Chapter 13
Trees

I like trees because they seem more resigned to the way they have to live than other things do.

— Willa Cather, *O Pioneers!,* 1913

Objectives

- To understand the concept of trees and the standard terminology used to describe them.
- To appreciate the recursive nature of a tree and how that recursive structure is reflected in its underlying representation.
- To become familiar with the data structures and algorithms used to implement binary search trees.
- To recognize that it is possible to maintain balance in a binary search tree as new keys are inserted.
- To learn how binary search trees can be implemented as a general abstraction.

As you have seen in several earlier chapters, linked lists make it possible to represent an ordered collection of values without using arrays. The link pointers associated with each cell form a linear chain that defines the underlying order. Although linked lists require more memory space than arrays and are less efficient for operations such as selecting a value at a particular index position, they have the advantage that insertion and deletion operations can be performed in constant time.

The use of pointers to define the ordering relationship among a set of values is considerably more powerful than the linked-list example suggests and is by no means limited to creating linear structures. In this chapter, you will learn about a data structure that uses pointers to model hierarchical relationships. That structure is called a **tree,** which is defined to be a collection of individual entries called **nodes** for which the following properties hold:

- As long as the tree contains any nodes at all, there is a specific node called the **root** that forms the top of a hierarchy.
- Every other node is connected to the root by a unique line of descent.

Tree-structured hierarchies occur in many contexts outside of computer science. The most familiar example is the family tree, which is discussed in the next section. Other examples include

- *Game trees*. The game trees introduced in the section on "The minimax strategy" in Chapter 7 have a branching pattern that is typical of trees. The current position is the root of the tree; the branches lead to positions that might occur later in the game.
- *Biological classifications*. The classification system for living organisms, which was developed in the eighteenth century by the Swedish botanist Carolus Linnaeus, is structured as a tree. The root of the tree is all living things. From there, the classification system branches to form separate kingdoms, of which animals and plants are the most familiar. From there, the hierarchy continues down through several additional levels until it defines an individual species.
- *Organization charts*. Many businesses are structured so that each employee reports to a single supervisor, forming a tree that extends up to the company president, who represents the root.
- *Directory hierarchies*. On most modern computers, files are stored in directories that form a tree. There is a top-level directory that represents the root, which can contain files along with other directories. Those directories may contain subdirectories, which gives rise to the hierarchical structure representative of trees.

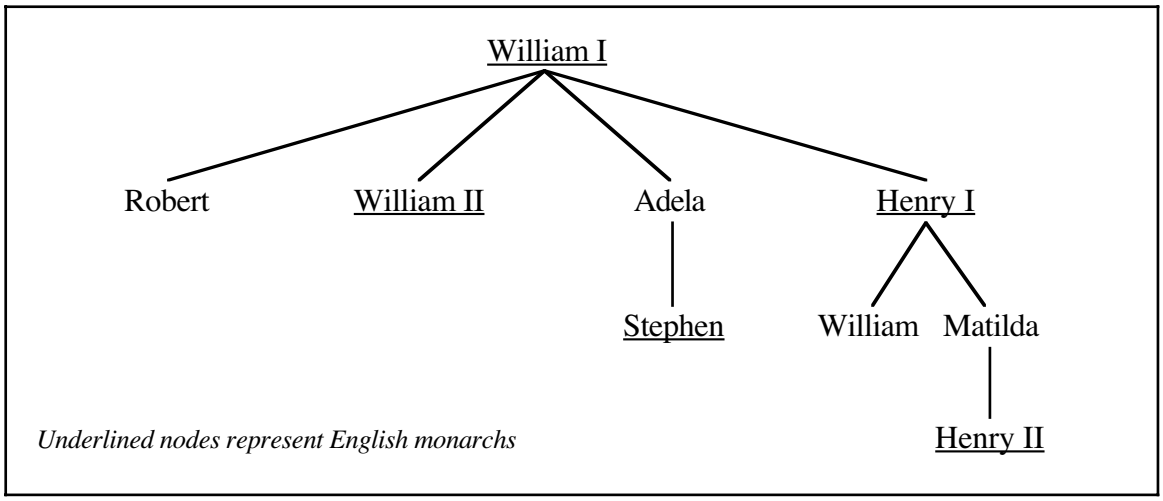
13.1 Family trees

Family trees provide a convenient way to represent the lines of descent from a single individual through a series of generations. For example, the diagram in Figure 13-1 shows the family tree of the House of Normandy, which ruled England after the accession of William I at the Battle of Hastings in 1066. The structure of the diagram fits the definition of a tree given in the preceding section. William I is the root of the tree, and all other individuals in the chart are connected to William I through a unique line of descent.

Terminology used to describe trees

The family tree in Figure 13-1 makes it easy to introduce the terminology computer scientists use to describe tree structures. Each node in a tree may have several **children,** but only a single **parent** in the tree. In the context of trees, the words **ancestor** and

Figure 13-1 Family tree for the House of Normandy

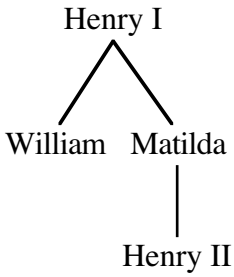


descendant have exactly the same meaning as they do in English. The line of descent through Henry I and Matilda shows that Henry II is a descendant of William I, which in turn implies that William I is an ancestor of Henry II. Similarly, the term **siblings** is used to refer to two nodes that share the same parent, such as Robert and Adela.

Although most of the terms used to describe trees come directly from the family-tree analogue, others—like the word *root*—come from the botanical metaphor instead. At the opposite end of the tree from the root, there are nodes that have no children, which are called **leaves.** Nodes that are neither the root nor a leaf are called **interior** nodes. For example, in Figure 13-1, Robert, William II, Stephen, William, and Henry II represent leaf nodes; Adela, Henry I, and Matilda represent interior nodes. The **height** of a tree is defined to be the length of the longest path from the root to a leaf. Thus, the height of the tree shown in Figure 13-1 is 4, because there are four nodes on the path from William I to Henry II, which is longer than any other path from the root.

The recursive nature of a tree

One of the most important things to notice about any tree is that the same branching pattern occurs at every level of the decomposition. If you take any node in a tree together with all its descendants, the result fits the definition of a tree. For example, if you extract the portion of Figure 13-1 beginning at Henry I, you get the following tree:



A tree formed by extracting a node and its descendants from an existing tree is called a **subtree** of the original one. The tree in this diagram, for example, is the subtree rooted at Henry I.

The fact that each node in a tree can be considered the root of its own subtree underscores the recursive nature of tree structures. If you think about trees from a

recursive perspective, a tree is simply a node and a set—possibly empty in the case of a leaf node—of attached subtrees. The recursive character of trees is fundamental to their underlying representation as well as to most algorithms that operate on trees.

Representing family trees in C++

In order to represent any type of tree in C++, you need some way to model the hierarchical relationships among the data values. In most cases, the easiest way to represent the parent/child relationship is to include a pointer in the parent that points to the child. If you use this strategy, each node is a structure that contains—in addition to other data specific to the node itself—pointers to each of its children. In general, it works well to define a node as the structure itself and to define a tree as a pointer to that structure. This definition is mutually recursive even in its English conception because of the following relationship:

- Trees are pointers to nodes.
- Nodes are structures that contain trees.

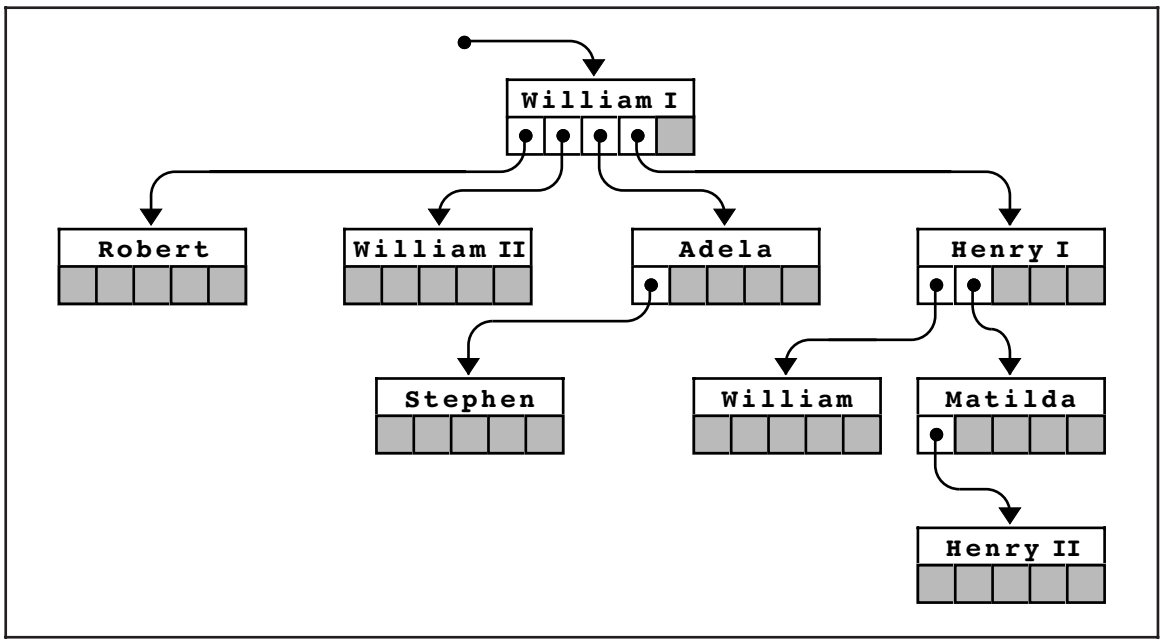
How would you use this recursive insight to design a structure suitable for storing the data in a family tree such as the one shown in Figure 13-1? Each node consists of a name of a person and a set of pointers to its children. If you store the child pointers in a vector, a node has the following form as a C++ structure:

```
struct familyTreeNodeT {
   string name;
   Vector<familyTreeNodeT *> children;
};
```

A family tree is simply a pointer to one of these nodes.

A diagram showing the internal representation of the royal family tree appears in Figure 13-2. To keep the figure neat and orderly, Figure 13-2 represents the children as if they were stored in a five-element array; in fact, the **children** field is a vector that grows to

Figure 13-2 Pointer-based representation of the family tree



accommodate any number of children. You will have a chance to explore other strategies for storing the children, such as keeping them in a linked list rather than a vector, in the exercises at the end of this chapter.

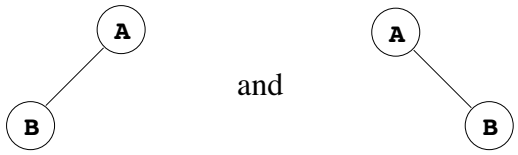
13.2 Binary search trees

Although it is possible to illustrate tree algorithms using family trees, it is more effective to do so in a simpler environment that has more direct application to programming. The family-tree example provides a useful framework for introducing the terminology used to describe trees, but suffers in practice from the complication that each node can have an arbitrary number of children. In many programming contexts, it is reasonable to restrict the number of children to make the resulting trees easier to implement.

One of the most important subclasses of trees—which has many practical applications—is a **binary tree,** which is defined to be a tree in which the following additional properties hold:

- Each node in the tree has at most two children.
- Every node except the root is designated as either a *left child* or a *right child* of its parent.

The second condition emphasizes the fact that child nodes in a binary tree are ordered with respect to their parents. For example, the binary trees



are different trees, even though they consist of the same nodes. In both cases, the node labeled `B` is a child of the root node labeled `A`, but it is a left child in the first tree and a right child in the second.

The fact that the nodes in a binary tree have a defined geometrical relationship makes it convenient to represent ordered collections of data using binary trees. The most common application uses a special class of binary tree called a **binary search tree,** which is defined by the following properties:

1. Every node contains—possibly in addition to other data—a special value called a *key* that defines the order of the nodes.
2. Key values are *unique,* in the sense that no key can appear more than once in the tree.
3. At every node in the tree, the key value must be greater than all the keys in the subtree rooted at its left child and less than all the keys in the subtree rooted at its right child.

Although this definition is formally correct, it almost certainly seems confusing at first glance. To make sense of the definition and begin to understand why constructing a tree that meets these conditions might be useful, it helps to go back and look at a specific problem for which binary search trees represent a potential solution strategy.

The underlying motivation for using binary search trees

In Chapter 12, one of the strategies proposed for representing maps—before the hashing algorithm made other options seem far less attractive—was to store the key/value pairs in

an array. This strategy has a useful computational property: if you keep the keys in sorted order, you can write an implementation of **get** that runs in $O(\log N)$ time. All you need to do is employ the binary search algorithm, which was introduced in Chapter 5. Unfortunately, the array representation does not offer any equally efficient way to code the **put** function. Although **put** can use binary search to determine where any new key fits into the array, maintaining the sorted order requires $O(N)$ time because each subsequent array element must be shifted to make room for the new entry.

This problem brings to mind a similar situation that arose in Chapter 11. When arrays were used to represent the editor buffer, inserting a new character was a linear-time operation. In that case, the solution was to replace the array with a linked list. Is it possible that a similar strategy would improve the performance of **put** for the map? After all, inserting a new element into a linked list—as long as you have a pointer to the cell prior to the insertion point—is a constant-time operation.

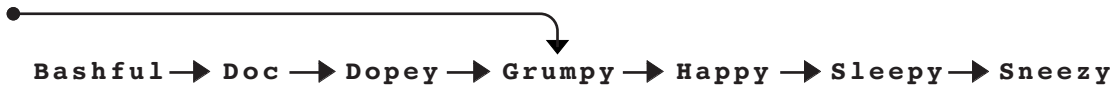
The trouble with linked lists is that they do not support the binary search algorithm in any efficient way. Binary search depends on being able to find the middle element in constant time. In an array, finding the middle element is easy. In a linked list, the only way to do so is to iterate through all the link pointers in the first half of the list.

To get a more concrete sense of why linked lists have this limitation, suppose that you have a linked list containing the following seven elements:

The elements in this list appear in lexicographic order, which is the order imposed by their internal character codes.

Given a linked list of this sort, you can easily find the first element, because the initial pointer gives you its address. From there, you can follow the link pointer to find the second element. On the other hand, there is no easy way to locate the element that occurs halfway through the sequence. To do so, you have to walk through each chain pointer, counting up to $N/2$. This operation requires linear time, which completely negates the efficiency advantage of binary search. If binary search is to offer any improvement in efficiency, the data structure must enable you to find the middle element quickly.

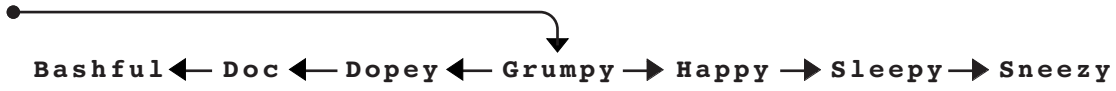
Although it might at first seem silly, it is useful to consider what happens if you simply point at the middle of the list instead of the beginning:



In this diagram, you have no problem at all finding the middle element. It's immediately accessible through the list pointer. The problem, however, is that you've thrown away the first half of the list. The pointers in the structure provide access to **Grumpy** and any name that follows it in the chain, but there is no longer any way to reach **Bashful**, **Doc**, and **Dopey**.

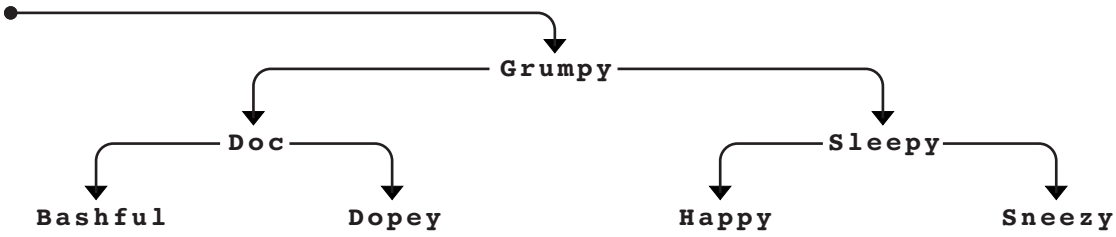
If you think about the situation from **Grumpy**'s point of view, the general outline of the solution becomes clear. What you need is to have two chains emanating from the **Grumpy** cell: one that consists of the cells whose names precede **Grumpy** and another for the cells

whose names follow **Grumpy** in the alphabet. In the conceptual diagram, all you need to do is reverse the arrows:



Each of the strings is now accessible, and you can easily divide the entire list in half.

At this point, you need to apply the same strategy recursively. The binary search algorithm requires you to find the middle of not only the original list but its sublists as well. You therefore need to restructure the lists that precede and follow **Grumpy**, using the same decomposition strategy. Every cell points in two directions: to the midpoint of the list that precedes it and to the midpoint of the list that follows it. Applying this process transforms the original list into the following binary tree:



The most important feature about this particular style of binary tree is that it is ordered. For any particular node in the tree, the string it contains must follow all the strings in the subtree descending to the left and precede all strings in the subtree to the right. In this example, **Grumpy** comes after **Doc**, **Bashful**, and **Dopey** but before **Sleepy**, **Happy**, and **Sneezy**. The same rule, however, applies at each level, so the node containing **Doc** comes after the **Bashful** node but before the **Dopey** node. The formal definition of a binary search tree, which appears at the end of the preceding section, simply ensures that every node in the tree obeys this ordering rule.

Finding nodes in a binary search tree

The fundamental advantage of a binary search tree is that you can use the binary search algorithm to find a particular node. Suppose, for example, that you are looking for the node containing the string **Happy** in the tree diagram shown at the end of the preceding section. The first step is to compare **Happy** with **Grumpy**, which appears at the root of the tree. Since **Happy** comes after **Grumpy** in lexicographic order, you know that the **Happy** node, if it exists, must be in the right subtree. The next step, therefore, is to compare **Happy** and **Sleepy**. In this case, **Happy** comes before **Sleepy** and must therefore be in the left subtree of this node. That subtree consists of a single node, which contains the correct name.

Because trees are recursive structures, it is easy to code the search algorithm in its recursive form. For concreteness, let's suppose that the type definition for **nodeT** looks like this:

```
struct nodeT {
    string key;
    nodeT *left, *right;
};
```

Given this definition, you can easily write a function **FindNode** that implements the binary search algorithm, as follows:

```
nodeT *FindNode(nodeT *t, string key) {
    if (t == NULL) return NULL;
    if (key == t->key) return t;
    if (key < t->key) {
        return FindNode(t->left, key);
    } else {
        return FindNode(t->right, key);
    }
}
```

If the tree is empty, the desired node is clearly not there, and **FindNode** returns the value **NULL** as a sentinel indicating that the key cannot be found. If the tree is not equal to **NULL**, the implementation checks to see whether the desired key matches the one in the current node. If so, **FindNode** returns a pointer to the current node. If the keys do not match, **FindNode** proceeds recursively, looking in either the left or right subtree depending on the result of the key comparison.

Inserting new nodes in a binary search tree

The next question to consider is how to create a binary search tree in the first place. The simplest approach is to begin with an empty tree and then call an **InsertNode** function to insert new keys into the tree, one at a time. As each new key is inserted, it is important to maintain the ordering relationship among the nodes of the tree. To make sure the **FindNode** function continues to work, the code for **InsertNode** must use binary search to identify the correct insertion point.

As with **FindNode**, the code for **InsertNode** can proceed recursively beginning at the root of the tree. At each node, **InsertNode** must compare the new key to the key in the current node. If the new key precedes the existing one, the new key belongs in the left subtree. Conversely, if the new key follows the one in the current node, it belongs in the right subtree. Eventually, the process will encounter a **NULL** subtree that represents the point in the tree where the new node needs to be added. At this point, the **InsertNode** implementation must replace the **NULL** pointer with a new node initialized to contain a copy of the key.

The code for **InsertNode**, however, is a bit tricky. The difficulty comes from the fact that **InsertNode** must be able to change the value of the binary search tree by adding a new node. Since the function needs to change the values of the argument, it must be passed by reference. Instead of taking a **nodeT *** as its argument the way **FindNode** does, **InsertNode** must instead take a **nodeT * &**. The prototype for **InsertNode** therefore looks like this:

```
void InsertNode(nodeT * & t, string key);
```

Once you understand the prototype for the **InsertNode** function, writing the code is not particularly hard. A complete implementation of **InsertNode** appears in Figure 13-3. If **t** is **NULL**, **InsertNode** creates a new node, initializes its fields, and then replaces the **NULL** pointer in the existing structure with a pointer to the new node. If **t** is not **NULL**, **InsertNode** compares the new key with the one stored at the root of the tree **t**. If the keys match, the key is already in the tree and no further operations are required. If not, **InsertNode** uses the result of the comparison to determine whether to insert the key in the left or the right subtree and then makes the appropriate recursive call.

Figure 13-3 Standard algorithm for inserting a node into a binary search tree

```
/*
 * Inserts the specified key at the appropriate location in the
 * binary search tree rooted at t.  Note that t must be passed
 * by reference, since it is possible to change the root.
 */

void InsertNode(nodeT * & t, string key) {
    if (t == NULL) {
        t = new nodeT;
        t->key = key;
        t->left = t->right = NULL;
        return;
    }
    if (key == t->key) return;
    if (key < t->key) {
        InsertNode(t->left, key);
    } else {
        InsertNode(t->right, key);
    }
}
```

Because the code for **InsertNode** seems complicated until you've seen it work, it makes sense to go through the process of inserting a few keys in some detail. Suppose, for example, that you have declared and initialized an empty tree as follows:

```
nodeT *dwarfTree = NULL;
```

These statements create a local variable **dwarfTree** that lives at some address in memory, as illustrated by the following diagram:

The actual address depends on the structure of the computer's memory system and the way the compiler allocates addresses. In this example, each memory word has been assigned an arbitrary address to make it easier to follow the operation of the code.

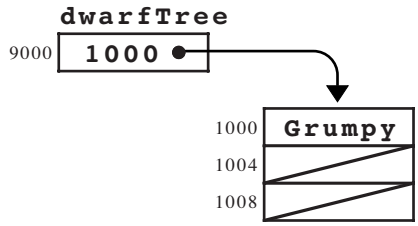
What happens if you call

```
InsertNode(dwarfTree, "Grumpy");
```

starting with this initial configuration in which **dwarfTree** is empty? In the frame for **InsertNode**, the variable **t** is a reference parameter, aliased to the variable **dwarfTree** stored at address 9000. The first step in the code checks if **t** is set to **NULL**, which is true is this case, so it executes the body of the **if** statement

```
if (t == NULL) {
    t = new nodeT;
    t->key = key;
    t->left = t->right = NULL;
    return;
}
```

which has the effect of creating a new node, initializing it to hold the key **Grumpy**, and then storing the address of the new node into the reference parameter **t**. When the function returns, the tree looks like this:

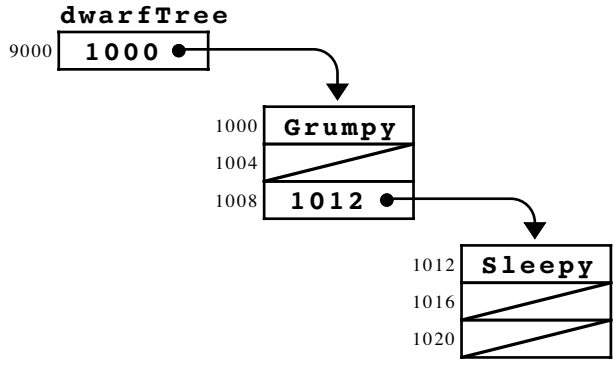


This structure correctly represents the binary search tree with a single node containing **Grumpy**.

What happens if you then use **InsertNode** to insert **Sleepy** into the tree? As before, the initial call generates a stack frame in which the reference parameter **t** is aliased to **dwarfTree**. This time, however, the value of the tree **t** is no longer **NULL**. **dwarfTree** now contains the address of the node containing **Grumpy**. Because **Sleepy** comes after **Grumpy** in the lexicographical order, the code for **InsertNode** continues with the following recursive call:

```
InsertNode(t->right, key);
```

At this point, the recursive call looks much like the insertion of **Grumpy** into the original empty tree. The only difference is that the reference parameter **t** now refers to a field within an existing node. The effect of the recursive call is therefore to store a pointer to a new node containing **Sleepy** in the **right** field of the root node, which gives rise to the following configuration:

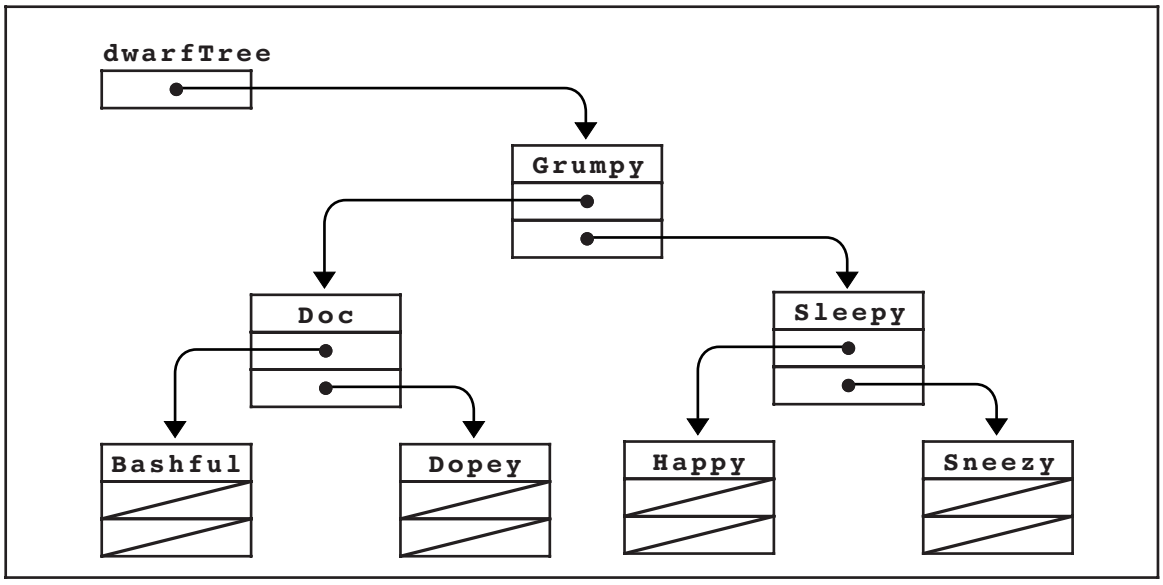


Additional calls to **InsertNode** will create additional nodes and insert them into the structure in a way that preserves the ordering constraint required for binary search trees. For example, if you insert the names of the five remaining dwarves in the order **Doc**, **Bashful**, **Dopey**, **Happy**, and **Sneezy**, you end up with the binary search tree shown in Figure 13-4.

Tree traversals

The structure of a binary search tree makes it easy to go through the nodes of the tree in the order specified by the keys. For example, you can use the following function to display the keys in a binary search tree in lexicographic order:

Figure 13-4 Binary search tree for the seven dwarves




```
void DisplayTree(nodeT *t) {
    if (t != NULL) {
        DisplayTree(t->left);
        cout << t->key << endl;
        DisplayTree(t->right);
    }
}
```

Thus, if you call **DisplayTree** on the tree shown in Figure 13-4, you get the following output:

```
Bashful
Doc
Dopey
Grumpy
Happy
Sleepy
Sneezy
```

At each recursive level, **DisplayTree** checks to see whether the tree is empty. If it is, **DisplayTree** has no work to do. If not, the ordering of the recursive calls ensures that the output appears in the correct order. The first recursive call displays the keys that precede the current node, all of which must appear in the left subtree. Displaying the nodes in the left subtree before the current one therefore maintains the correct order. Similarly, it is important to display the key from the current node before making the last recursive call, which displays the keys that occur later in the ASCII sequence and therefore appear in the right subtree.

The process of going through the nodes of a tree and performing some operation at each node is called **traversing** or **walking** the tree. In many cases, you will want to traverse a tree in the order imposed by the keys, as in the **DisplayTree** example. This approach, which consists of processing the current node between the recursive calls to the left and right subtrees, is called an **inorder traversal.** There are, however, two other types of tree traversals that occur frequently in the context of binary trees, which are

called **preorder** and **postorder** traversals. In the preorder traversal, the current node is processed before either of its subtrees; in the postorder traversal, the subtrees are processed first, followed by the current node. These traversal styles are illustrated by the functions **PreOrderWalk** and **PostOrderWalk**, which appear in Figure 13-5. The sample output associated with each function shows the result of applying that function to the balanced binary search tree in Figure 13-4.

13.3 Balanced trees

Although the recursive strategy used to implement **InsertNode** guarantees that the nodes are organized as a legal binary search tree, the structure of the tree depends on the order in which the nodes are inserted. The tree in Figure 13-4, for example, was generated by inserting the names of the dwarves in the following order:

Grumpy, **Sleepy**, **Doc**, **Bashful**, **Dopey**, **Happy**, **Sneezy**

Suppose that you had instead entered the names of the dwarves in alphabetical order. The first call to **InsertNode** would insert **Bashful** at the root of the tree. Subsequent calls would insert **Doc** after **Bashful**, **Dopey** after **Doc**, and so on, appending each new node to the **right** chain of the previously one.

The resulting figure, which is shown in Figure 13-6, looks more like a linked list than a tree. Nonetheless, the tree in Figure 13-6 maintains the property that the key field in any node follows all the keys in its left subtree and precedes all the keys in its right subtree. It therefore fits the definition of a binary search tree, so the **FindNode** function will operate correctly. The running time of the **FindNode** algorithm, however, is proportional to the height of the tree, which means that the structure of the tree can have a significant impact on the algorithmic performance. If a binary search tree is shaped like the one shown in Figure 13-4, the time required to find a key in the tree will be $O(\log N)$. On the other hand, if the tree is shaped like the one in Figure 13-6, the running time will deteriorate to $O(N)$.

The binary search algorithm used to implement **FindNode** achieves its ideal performance only if the left and right subtrees have roughly the same height at each level of the tree. Trees in which this property holds—such as the tree in Figure 13-4—are said

Figure 13-5 Preorder and postorder traversals of a binary search tree

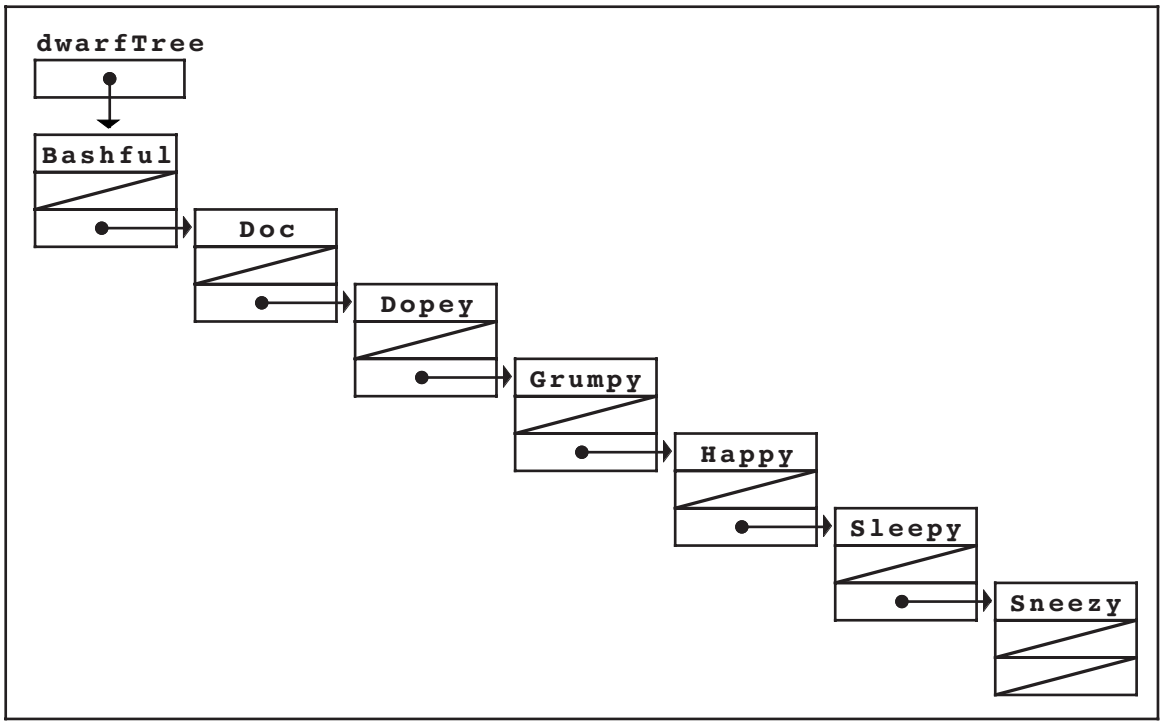
```
void PreOrderWalk(nodeT *t) {
   if (t != NULL) {
      cout << t->key << endl;
      PreOrderWalk(t->left);
      PreOrderWalk(t->right);
   }
}
```

```
Grumpy
Doc
Bashful
Dopey
Sleepy
Happy
Sneezy
```

```
void PostOrderWalk(nodeT *t) {
   if (t != NULL) {
      PostOrderWalk(t->left);
      PostOrderWalk(t->right);
      cout << t->key << endl;
   }
}
```

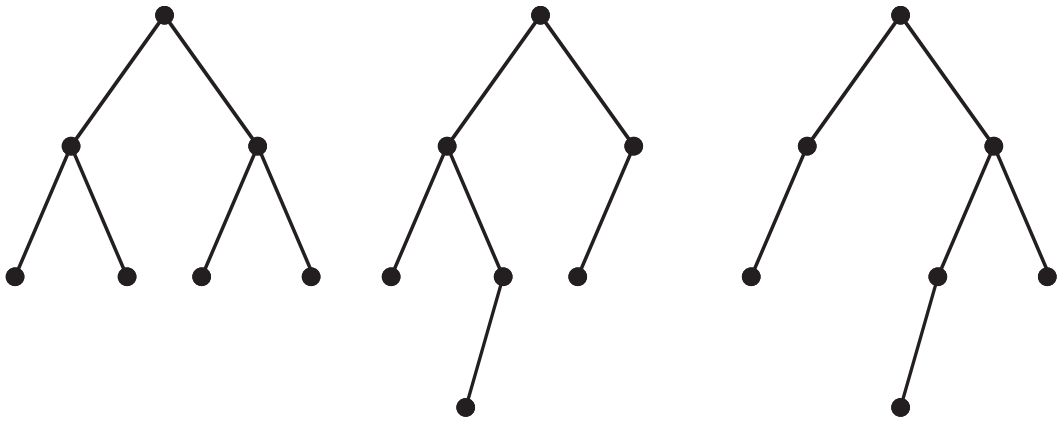
```
Bashful
Dopey
Doc
Happy
Sneezy
Sleepy
Grumpy
```

Figure 13-6 Unbalanced tree that results if the names are inserted in lexicographic order



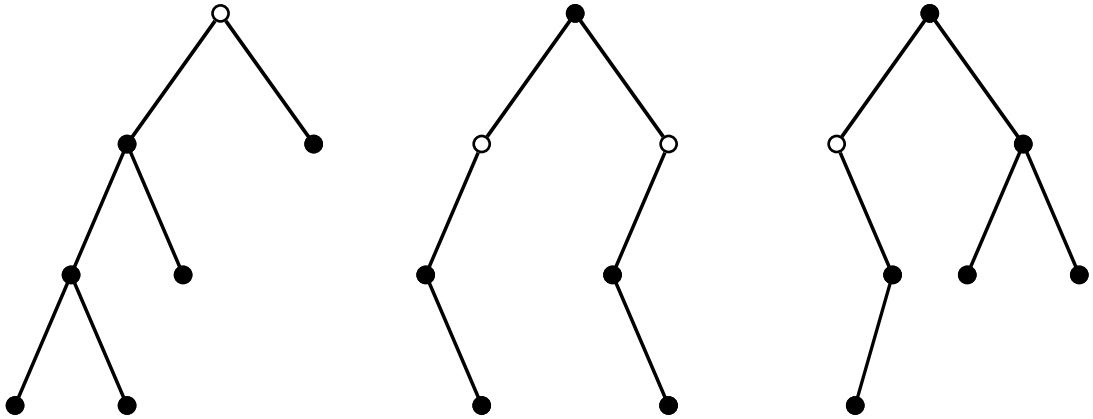
to be **balanced.** More formally, a binary tree is defined to be balanced if, at each node, the height of the left and right subtrees differ by at most one.

To illustrate this definition of a balanced binary tree, each of the following diagrams shows a balanced arrangement of a tree with seven nodes:



The tree diagram on the left is optimally balanced in the sense that the heights of the two subtrees at each node are always equal. Such an arrangement is possible, however, only if the number of nodes is one less than a power of two. If the number of nodes does not meet this condition, there will be some point in the tree where the heights of the subtrees differ to some extent. By allowing the heights of the subtrees to differ by one, the definition of a balanced tree provides some flexibility in the structure of a tree without adversely affecting its computational performance.

The diagrams at the top of the next page represent unbalanced trees.



In each diagram, the nodes at which the balanced-tree definition fails are shown as open circles. In the leftmost tree, for example, the left subtree of the root node has height 3 while the right subtree has height 1.

Tree-balancing strategies

Binary search trees are useful in practice only if it is possible to avoid the worst-case behavior associated with unbalanced trees. As trees become unbalanced, the **`FindNode`** and **`InsertNode`** operations become linear in their running time. If the performance of binary trees deteriorates to $O(N)$, you might as well use a sorted array to store the values. With a sorted array, it requires $O(\log N)$ time to implement **`FindNode`** and $O(N)$ time to implement **`InsertNode`**. From a computational perspective, the performance of the array-based algorithms is therefore superior to that of unbalanced trees, even though the array implementation is considerably easier to write.

What makes binary search trees useful as a programming tool is the fact that you can keep them balanced as you build them. The basic idea is to extend the implementation of **`InsertNode`** so that it keeps track of whether the tree is balanced while inserting new nodes. If the tree ever becomes out of balance, **`InsertNode`** must rearrange the nodes in the tree so that the balance is restored without disturbing the ordering relationships that make the tree a binary search tree. Assuming that it is possible to rearrange a tree in time proportional to its height, both **`FindNode`** and **`InsertNode`** can be implemented in $O(\log N)$ time.

Algorithms for maintaining balance in a binary tree have been studied extensively in computer science. The algorithms used today to implement balanced binary trees are quite sophisticated and have benefited enormously from theoretical research. Most of these algorithms, however, are difficult to explain without reviewing mathematical results beyond the scope of this text. To demonstrate that such algorithms are indeed possible, the next few sections present one of the first tree-balancing algorithms, which was published in 1962 by the Russian mathematicians Georgii Adel'son-Vel'skii and Evgenii Landis and has since been known by the initials AVL. Although the AVL algorithm has been largely replaced in practice by more modern approaches, it has the advantage of being considerably easier to explain. Moreover, the operations used to implement the basic strategy reappear in many other algorithms, which makes it a good model for more modern techniques.

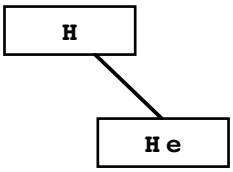
Illustrating the AVL idea

Before you attempt to understand the implementation of the AVL algorithm in detail, it helps to follow through the process of inserting nodes into a binary search tree to see

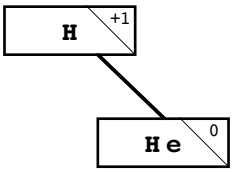
what can go wrong and, if possible, what steps you can take to fix any problems that arise. Let's imagine that you want to create a binary search tree in which the nodes contain the symbols for the chemical elements. For example, the first six elements are

H (Hydrogen)
He (Helium)
Li (Lithium)
Be (Beryllium)
B (Boron)
C (Carbon)

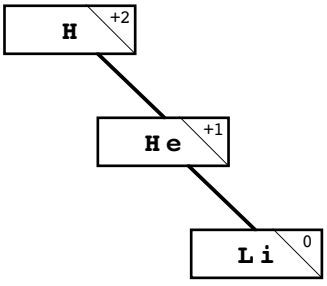
What happens if you insert the chemical symbols for these elements in the indicated order, which is how these elements appear in the periodic table? The first insertion is easy because the tree is initially empty. The node containing the symbol **H** becomes the root of the tree. If you call **InsertNode** on the symbol **He**, the new node will be added after the node containing **H**, because **He** comes after **H** in lexicographic order. Thus, the first two nodes in the tree are arranged like this:



To keep track of whether the tree is balanced, the AVL algorithm associates an integer with each node, which is simply the height of the right subtree minus the height of the left subtree. This value is called the **balance factor** of the node. In the simple tree that contains the symbols for the first two elements, the balance factors, which are shown here in the upper right corner of each node, look like this:

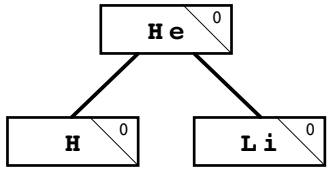


So far, the tree is balanced because none of the nodes has a balance factor whose absolute value is greater than 1. That situation changes, however, when you add the next element. If you follow the standard insertion algorithm, adding **Li** results in the following configuration:



Here, the root node is out of balance because its right subtree has height 2 and its left subtree has height 0, which differ by more than one.

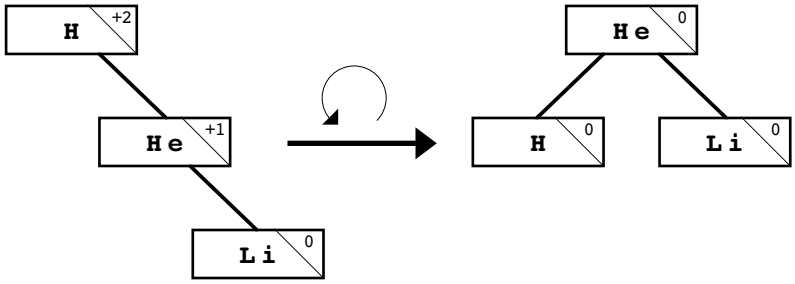
To fix the imbalance, you need to restructure the tree. For this set of nodes, there is only one balanced configuration in which the nodes are correctly ordered with respect to each other. That tree has **He** at the root, with **H** and **Li** in the left and right subtrees, as follows:



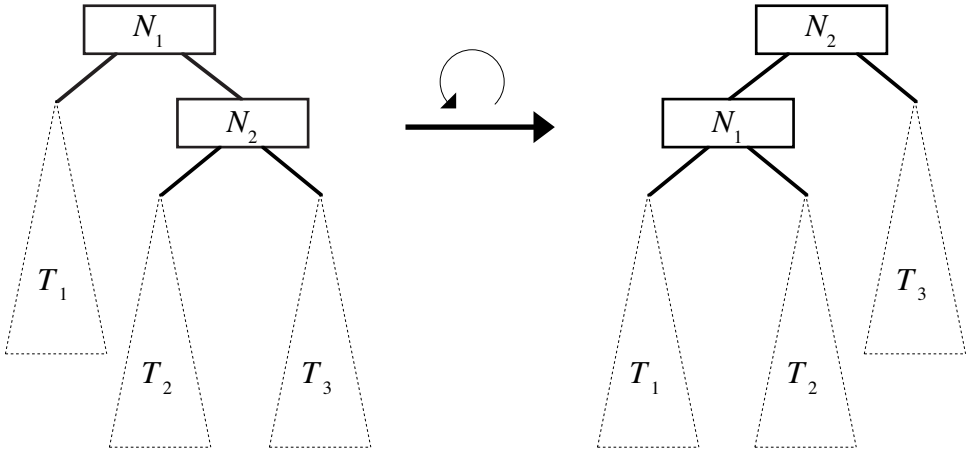
This tree is once again balanced, but an important question remains: how do you know what operations to perform in order to restore the balance in a tree?

Single rotations

The fundamental insight behind the AVL strategy is that you can always restore balance to a tree by a simple rearrangement of the nodes. If you think about what steps were necessary to correct the imbalance in the preceding example, it is clear that the **He** node moves upward to become the root while **H** moves downward to become its child. To a certain extent, the transformation has the characteristic of rotating the **H** and **He** nodes one position to the left, like this:

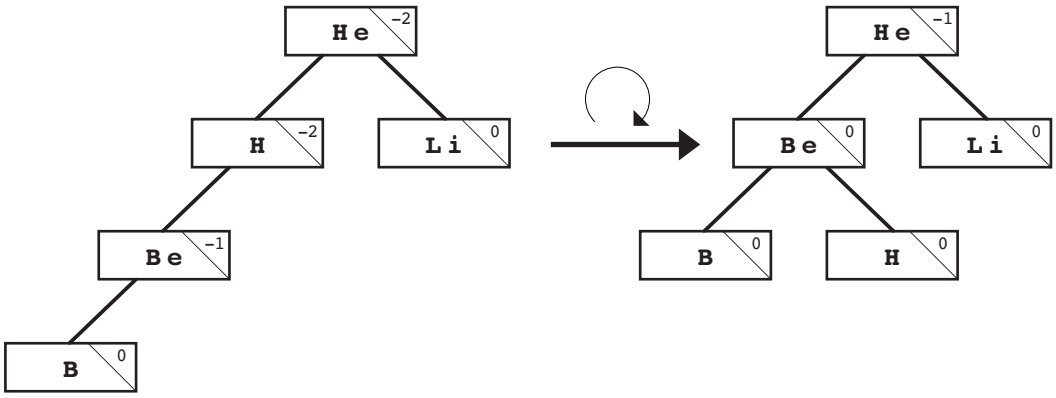


In general, you can always perform this type of rotation operation on any two nodes in a binary search tree without invalidating the relative ordering of the nodes, even if the nodes have subtrees descending from them. In a tree with additional nodes, the basic operation looks like this:

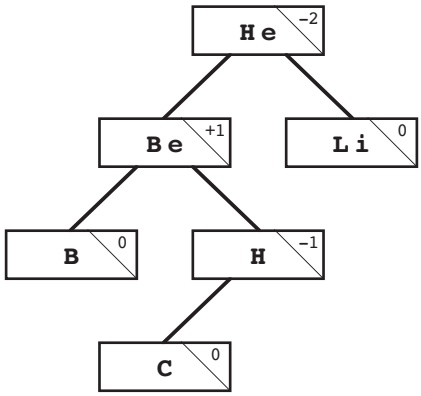


Note that the T_2 subtree, which would otherwise be orphaned by this process, must be reattached to N_1 after N_1 and N_2 change positions.

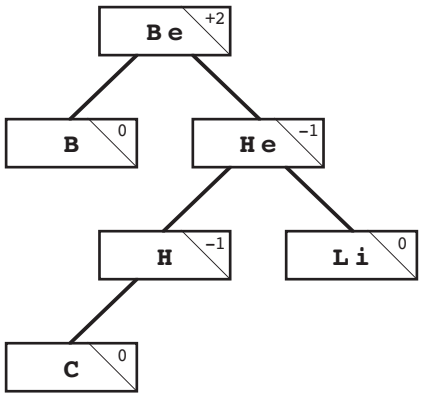
The two nodes involved in the rotation operation are called the **axis** of the rotation. In the example consisting of the elements **H**, **He**, and **Li**, the rotation was performed around the **H-He** axis. Because this operation moves nodes to the left, the operation illustrated by this diagram is called a **left rotation.** If a tree is out of balance in the opposite direction, you can apply a symmetric operation called a **right rotation,** in which all the operations are simply reversed. For example, the symbols for the next two elements—**Be** and **B**—each get added at the left edge of the tree. To rebalance the tree, you must perform a right rotation around the **Be-H** axis, as illustrated in the following diagram:



Unfortunately, simple rotation operations are not always sufficient to restore balance to a tree. Consider, for example, what happens when you add **C** to the tree. Before you perform any balancing operations, the tree looks like this:



The **He** node at the root of the tree is out of balance. If you try to correct the imbalance by rotating the tree to the right around the **Be-He** axis, you get the following tree:



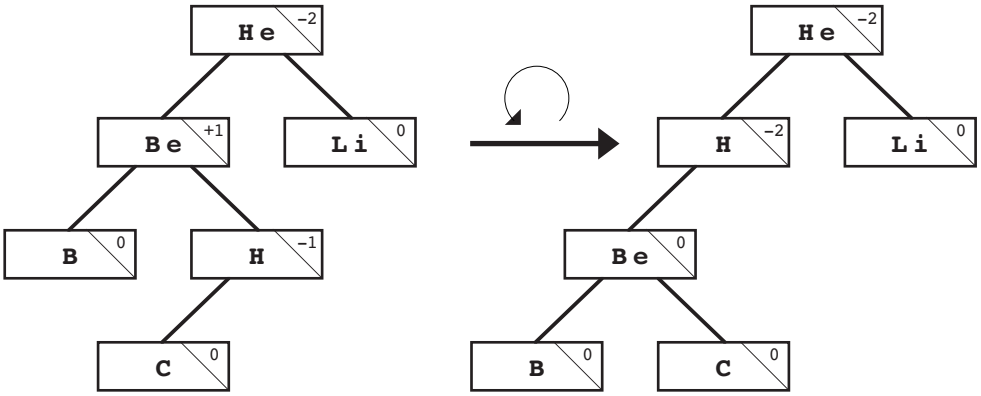
The tree is still unbalanced.

After the rotation, the tree is just as unbalanced as it was before. The only difference is that the root node is now unbalanced in the opposite direction.

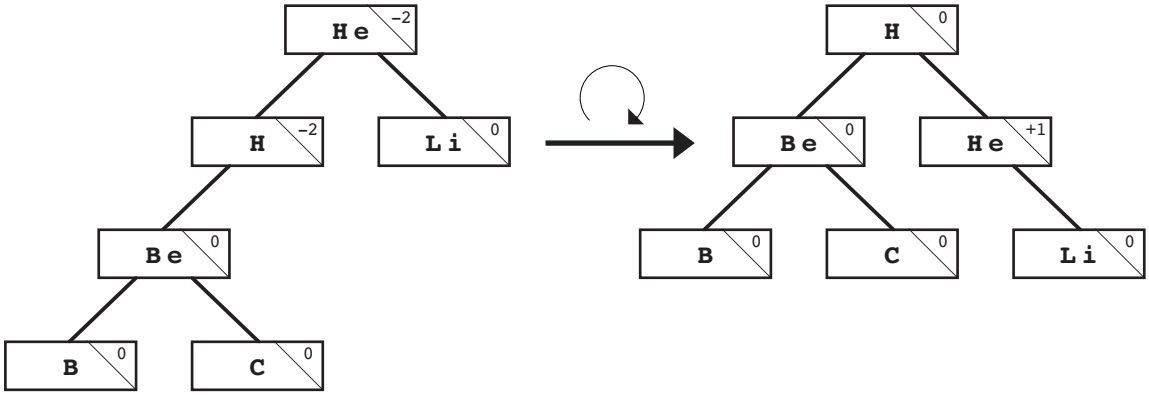
Double rotations

The problem in this last example arises because the nodes involved in the rotation have balance factors with opposite signs. When this situation occurs, a single rotation is not enough. To fix the problem, you need to make two rotations. Before rotating the out-of-balance node, you rotate its child in the opposite direction. Rotating the child gives the balance factors in the parent and child the same sign, which means that the following rotation will succeed. This pair of operations is called a **double rotation.**

As an illustration of the double-rotation operation, consider the preceding unbalanced tree of elements just after adding the symbol **C**. The first step is to rotate the tree to the left around the **Be**-**H** axis, like this:



The resulting tree is still out of balance at the root node, but the **H** and **He** nodes now have balance factors that share the same sign. In this configuration, a single rotation to the right around the **H**-**He** axis restores balance to the tree, as follows:



In their paper describing these trees, Adel'son-Vel'skii and Landis demonstrated the following properties of their tree-balancing algorithm:

- If you insert a new node into an AVL tree, you can always restore its balance by performing at most one operation, which is either a single or a double rotation.
- After you complete the rotation operation, the height of the subtree at the axis of rotation is always the same as it was before inserting the new node. This property ensures that none of the balance factors change at any higher levels of the tree.

Implementing the AVL algorithm

Although the process involves quite a few details, implementing **InsertNode** for AVL trees is not as difficult as you might imagine. The first change you need to make is to include a new field in the node structure that allows you to keep track of the balance factor, as follows:

```
struct nodeT {
    string key;
    nodeT *left, *right;
    int bf;
};
```

The code for **InsertNode** itself appears in Figure 13-7. As you can see from the code, **InsertNode** is implemented as a wrapper to a function **InsertAVL**, which at first glance seems to have the same prototype. The parameters to the two functions are indeed the same. The only difference is that **InsertAVL** returns an integer value that represents the change in the height of the tree after inserting the node. This return value, which will always be 0 or 1, makes it easy to fix the structure of the tree as the code makes its way back through the level of recursive calls. The simple cases are

1. Adding a node in place of a previously **NULL** tree, which increases the height by one
2. Encountering an existing node containing the key, which leaves the height unchanged

In the recursive cases, the code first adds the new node to the appropriate subtree, keeping track of the change in height in the local variable **delta**. If the height of the subtree to which the insertion was made has not changed, then the balance factor in the current node must also remain the same. If, however, the subtree increased in height, there are three possibilities:

1. *That subtree was previously shorter than the other subtree in this node*. In this case, inserting the new node actually makes the tree more balanced than it was previously. The balance factor of the current node becomes 0, and the height of the subtree rooted there remains the same as before.
2. *The two subtrees in the current node were previously the same size*. In this case, increasing the size of one of the subtrees makes the current node slightly out of balance, but not to the point that any corrective action is required. The balance factor becomes –1 or +1, as appropriate, and the function returns 1 to show that the height of the subtree rooted at this node has increased.
3. *The subtree that grew taller was already taller than the other subtree*. When this situation occurs, the tree has become seriously out of balance, because one subtree is now two nodes higher than the other. At this point, the code must execute the appropriate rotation operations to correct the imbalance. If the balance factors in the current node and the root of the subtree that expanded have the same sign, a single rotation is sufficient. If not, the code must perform a double rotation. After performing the rotations, the code must correct the balance factors in the nodes whose positions have changed. The effect of the single and double rotation operations on the balance factors in the node is shown in Figure 13-8.

Using the code for the AVL algorithm shown in Figure 13-7 ensures that the binary search tree remains in balance as new nodes are added. As a result, both **FindNode** and **InsertNode** will run in $O(\log N)$ time. Even without the AVL extension, however, the code will continue to work. The advantage of the AVL strategy is that it guarantees good performance, at some cost in the complexity of the code.

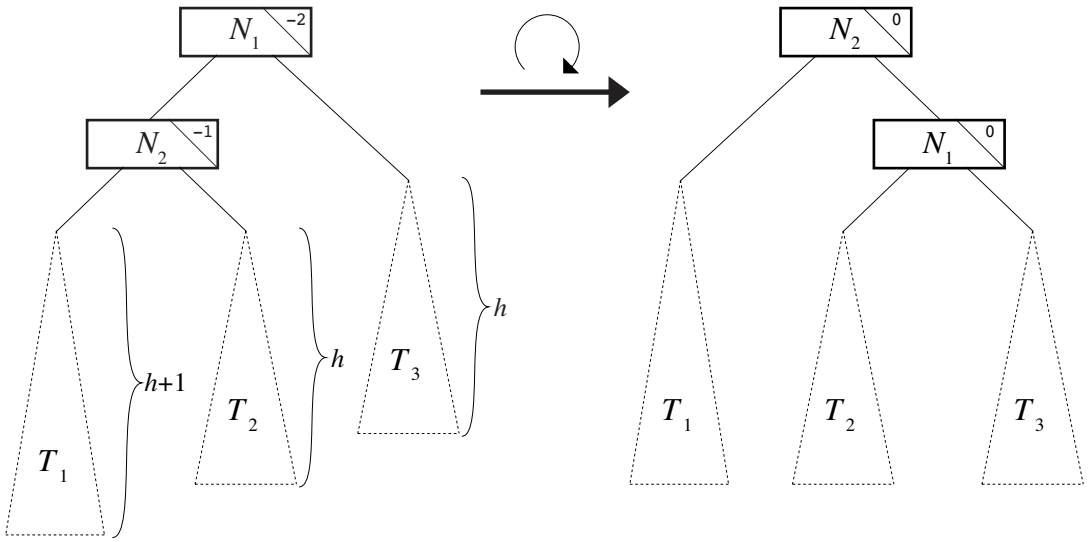
Figure 13-7 Code to insert a node into an AVL tree

```
/*
 * Function: InsertNode
 * Usage: InsertNode(t, key);
 * --------------------------
 * This function calls InsertAVL and discards the result.
 */

void InsertNode(nodeT * & t, string key) {
    InsertAVL(t, key);
}

/*
 * Function: InsertAVL
 * Usage: delta = InsertAVL(t, key);
 * ---------------------------------
 * This function enters the key into the tree whose is passed by
 * reference as the first argument.  The return value is the change
 * in depth in the tree, which is used to correct the balance
 * factors in ancestor nodes.
 */

int InsertAVL(nodeT * & t, string key) {
    if (t == NULL) {
        t = new nodeT;
        t->key = key;
        t->bf = 0;
        t->left = t->right = NULL;
        return +1;
    }
    if (key == t->key) return 0;
    if (key < t->key) {
        int delta = InsertAVL(t->left, key);
        if (delta == 0) return 0;
        switch (t->bf) {
          case +1: t->bf =  0; return 0;
          case  0: t->bf = -1; return +1;
          case -1: FixLeftImbalance(t); return 0;
        }
    } else {
        int delta = InsertAVL(t->right, key);
        if (delta == 0) return 0;
        switch (t->bf) {
          case -1: t->bf =  0; return 0;
          case  0: t->bf = +1; return +1;
          case +1: FixRightImbalance(t); return 0;
        }
    }
}
```

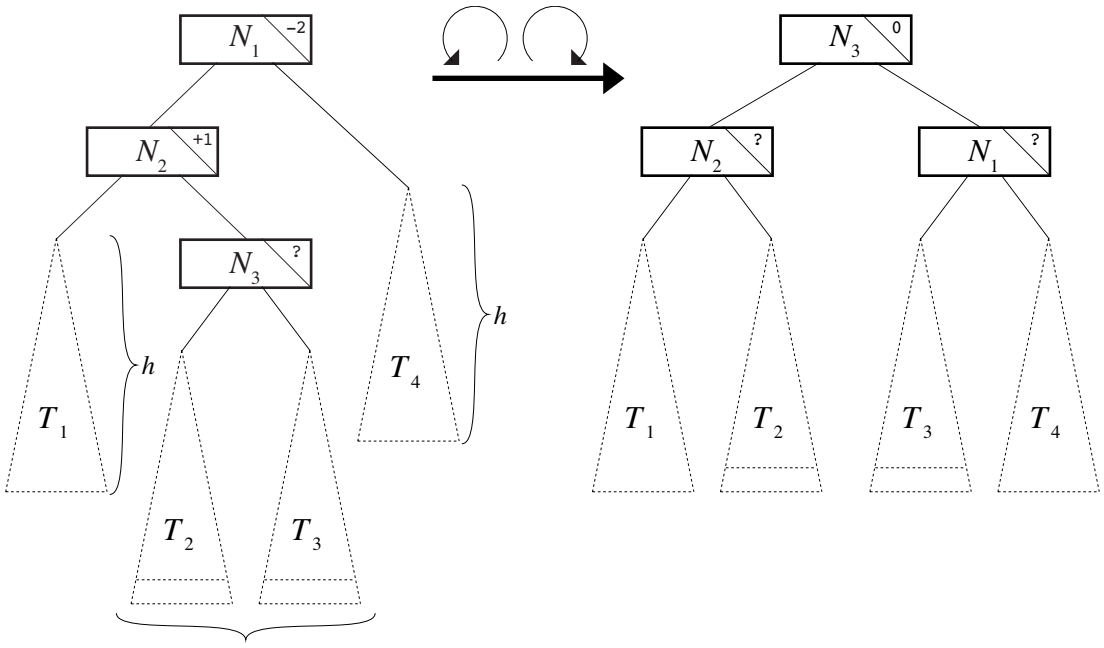
```
/*
 * Function: FixLeftImbalance
 * Usage: FixLeftImbalance(t);
 * ---------------------------
 * This function is called when a node has been found that
 * is out of balance with the longer subtree on the left.
 * Depending on the balance factor of the left child, the
 * code performs a single or double rotation.
 */

void FixLeftImbalance(nodeT * & t) {
    nodeT *child = t->left;
    if (child->bf != t->bf) {
        int oldBF = child->right->bf;
        RotateLeft(t->left);
        RotateRight(t);
        t->bf = 0;
        switch (oldBF) {
          case -1: t->left->bf = 0; t->right->bf = +1; break;
          case  0: t->left->bf = t->right->bf = 0; break;
          case +1: t->left->bf = -1; t->right->bf = 0; break;
        }
    } else {
        RotateRight(t);
        t->right->bf = t->bf = 0;
    }
}

/*
 * Function: RotateLeft
 * Usage: RotateLeft(t);
 * ---------------------
 * This function performs a single left rotation of the tree
 * that is passed by reference.  The balance factors
 * are unchanged by this function and must be corrected at a
 * higher level of the algorithm.
 */

void RotateLeft(nodeT * & t) {
    nodeT *child = t->right;
    t->right = child->left;
    child->left = t;
    t = child;
}

/* FixRightImbalance and RotateRight are defined similarly */
```

Figure 13-8 The effect of rotation operations on balance factors

Single rotation:



Double rotation:



Unless both T_2 and T_3 are empty ($h = 0$), one will have height h and the other height h–1

The new balance factors in the N_1 and N_2 nodes depend on the relative heights of the subtrees T_2 and T_3

13.4 Defining a general interface for binary search trees

The code in these last several sections has given you an idea of how binary search trees work. Up to this point, however, the focus has been entirely on the implementation level. What would you do if you wanted to build an application program that used a binary search tree? As things stand, you would almost certainly have to change the definition of the **`nodeT`** type to include the data required by your application. Such a change would require you to edit the source code for the implementation, which violates the basic principles of interface-based design. As a client, you should never have to edit the implementation code. In general, you should not need to know the details of the implementation at all.

Since Chapter 9, this text has used the public and private sections of a class to separate the client and the implementation. If you want to make binary search trees usable as a general tool, the ideal approach is to define a **`BST`** class that allows clients to invoke the basic operations without having to understand the underlying detail. That interface, moreover, needs to be as general as possible to offer the client maximum flexibility. In particular, the following features would certainly make the **`BST`** class more useful:

- *The class should allow the client to define the structure of the data in a node.* The binary search trees you've seen so far have included no data fields except the key itself. In most cases, clients want to work with nodes that contain additional data fields as well.
- *The keys should not be limited to strings.* Although the implementations in the preceding sections have used strings as keys, there is no reason that a general package would need to impose this constraint. To maintain the proper order in a binary tree, all the implementation needs to know is how to compare two keys. As long as the client provides a comparison function, it should be possible to use any type as a key.
- *It should be possible to remove nodes as well as to insert them.* Some clients—particularly including the set package introduced in Chapter 15—need to be able to remove entries from a binary search tree. Removing a node requires some care, but is easy enough to specify in the implementation.
- *The details of any balancing algorithm should lie entirely on the implementation side of the abstraction boundary.* The interface itself should not reveal what strategy, if any, the implementation uses to keep the tree in balance. Making the process of balancing the tree private to the implementation allows you to substitute new algorithms that perform more effectively than the AVL strategy without forcing clients to change their code.

Figure 13-9 defines an interface that includes each of these features. To understand why the interface looks the way it does, it is important to consider some of the issues that arise in its design and implementation. The sections that follow review these issues.

Allowing the client to define the node data

As it stands, the code for the binary search tree algorithms introduced earlier in this chapter defines a node structure in which the key is the only data field. As a client, you almost certainly want to include other information as well. How would you go about incorporating this additional information into the structure of a binary search tree? If you think back to how the container classes introduced in the earlier chapters have enabled clients to store arbitrary types within them, the answer to this question is quite obvious. As the implementer, you define the BST as a class template with a template parameter for the type of the data. As a client, you define a record type that contains the information you need, and then use that type to fill in the placeholder when creating a new BST.

Figure 13-9 Interface for a binary search tree class template

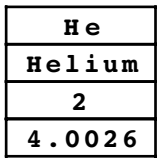
```
/*
 * File: bst.h
 * -----------
 * This file provides an interface for a general binary search
 * tree class template.
 */

#define _bst_h

#include "cmpfn.h"
#include "disallowcopy.h"

/*
 * Class: BST
 * ----------
 * This interface defines a class template for a binary search tree.
 * For maximum generality, the BST is supplied as a class template.
 * The data type is set by the client.  The client specializes the
 * tree to hold a specific type, e.g. BST<int> or BST<studentT>.
 * The one requirement on the type is that the client must supply a
 * a comparison function that compares two elements (or be willing
 * to use the default comparison function that relies on < and ==).
 */

template <typename ElemType>
class BST {

  public:

/*
 * Constructor: BST
 * Usage: BST<int> bst;
 *        BST<song> songs(CompareSong)
 *        BST<string> *bp = new BST<string>;
 * ----------------------------------------
 * The constructor initializes a new empty binary search tree.
 * The one argument is a comparison function, which is called
 * to compare data values.  This argument is optional, if not
 * given, OperatorCmp from cmpfn.h is used, which applies the
 * built-in operator < to its operands.  If the behavior of <
 * on your type is defined and sufficient, you do not need to
 * supply your own comparison function.
 */

    BST(int (*cmpFn)(ElemType one, ElemType two) = OperatorCmp);

/*
 * Destructor: ~BST
 * Usage: delete bp;
 * -----------------
 * This function deallocates the storage for a tree.
 */

    ~BST();
```

```
/*
 * Method: find
 * Usage:  if (bst.find(key) != NULL) . . .
 * ----------------------------------------
 * This method applies the binary search algorithm to find a key
 * in this tree.  The argument is the key you're looking for.  If
 * a node matching key appears in the tree, find returns a pointer
 * to the data in that node; otherwise, find returns NULL.
 */

    ElemType *find(ElemType key);

/*
 * Method: add
 * Usage: bst.add(elem);
 * ---------------------
 * This method adds a new node to this tree.  The elem argument
 * is compared with the data in existing nodes to find the proper
 * position.  If a node with the same value already exists, the
 * contents are overwritten with the new copy, and the add method
 * returns false.  If no matching node is found, a new node is
 * allocated and added to the tree, and the method returns true.
 */

    bool add(ElemType elem);

/*
 * Method: remove
 * Usage: bst.remove(key);
 * -----------------------
 * This method removes a node in this tree that matches the
 * specified key.  If a node matching key is found, the node
 * is removed from the tree and true is returned.  If no match
 * is found, no changes are made and false is returned.
 */

    bool remove(ElemType key);

/*
 * Method: mapAll
 * Usage: bst.mapAll(PrintToFile, outputStream);
 * ---------------------------------------------
 * This method iterates through the binary search tree and
 * calls the function fn once for each element, passing the
 * element and the client's data.  That data can be of whatever
 * type is needed for the client's callback.  The order of calls
 * is determined by an InOrder walk of the tree.
 */

    template <typename ClientElemType>
    void mapAll(void (*fn)(ElemType elem, ClientElemType &data),
                ClientElemType &data);

private:

#include "bstpriv.h"

}

#include "bstimpl.cpp"
```

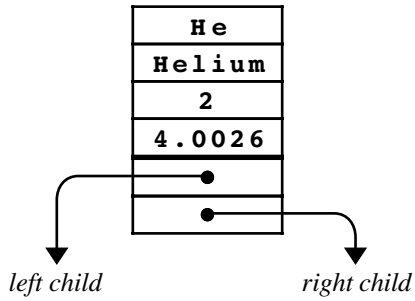
For example, let's go back to the idea of inserting the symbols for the chemical elements into a binary search tree. It is hard to imagine why anyone would want to create a tree that contained *only* the symbols for the elements, even though the symbol—being unique—makes a perfectly reasonable key. If you were writing an application that needed to know something about the elements, you would almost certainly want to store additional information. For example, in addition to the symbol for the element, you might want to store its name, atomic number, and atomic weight, which suggests the following data structure for the node:

```
struct elementT {
    string symbol;
    string name;
    int atomicNumber;
    double atomicWeight;
};
```

Thus, from the client's view, the data for the element helium would look like this:



The client would create an object of **`BST<elementT>`** which indicates each node in the tree will store the client's record plus the additional node information, such the pointers to its left and right children. A node in the tree might look like this:



The heavy line in the diagram divides the client data from the implementation data. Everything above that line belongs to the client. Everything below it—which might also include other fields necessary to keep the tree in balance—belongs to the implementation.

Generalizing the types used for keys

Allowing the client to use keys of any type is not particularly difficult. For the most part, the basic strategy is to have the client supply a comparison function. By storing a pointer to the client's comparison function as a data member for the tree as a whole, the implementation can invoke that function whenever it needs to compare keys. The comparison function is passed as an argument when constructing a new BST object. For each new data type to be stored in a BST, the client will have to supply the appropriate comparison function. For example, the **`BST<elementT>`** described above would require a function for comparing two **`elementT`** structs. The key field within an element is its symbol, so the **`CompareElements`** function below returns the result of comparing the symbol fields within the **`elementT`** structures.

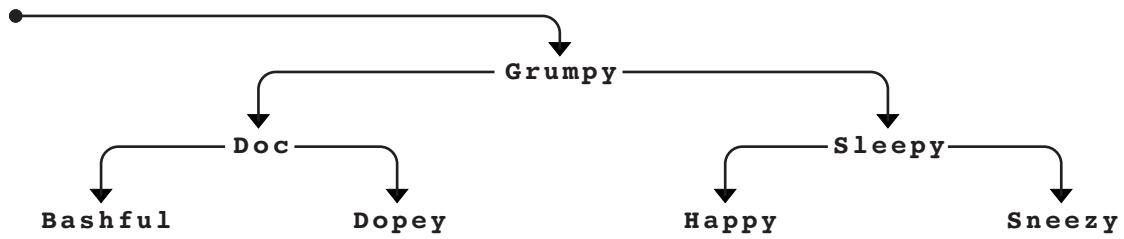
```
int CompareElements(elementT one, elementT two) {
    if (one.symbol == two.symbol) return 0;
    return (one.symbol < two.symbol) ? –1 : 1;
};
```

The comparison function argument to the constructor is optional. If not supplied, the default comparison function is used. The default comparison function was introduced for the general sorting routine in Chapter 12, it simply compares two values using the built-in operator **<**. If the BST is storing types such as **int** or **string** that can be compared using **<**, this default function can be used.

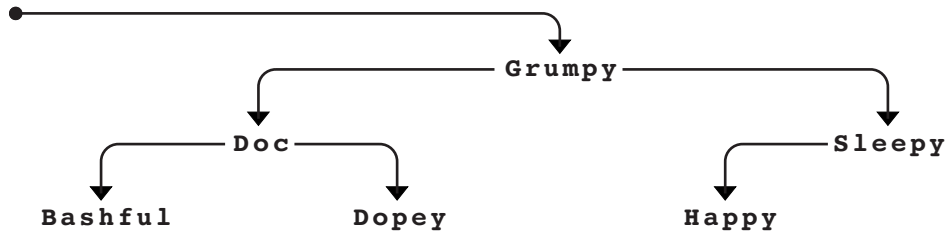
Removing nodes

The operation of removing a node from a binary search tree is not hard to define in the **BST** class interface. The interesting issues that arise in removing a node are primarily the concern of the implementation. Finding the node to be removed requires the same binary-search strategy as finding a node. Once you find the appropriate node, however, you have to remove it from the tree without violating the ordering relationship that defines a binary search tree. Depending on where the node to be removed appears in the tree, removing it can get rather tricky.

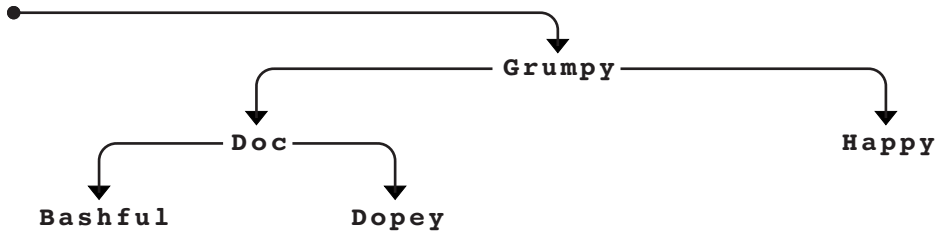
To get a sense of the problem, suppose that you are working with a binary search tree whose nodes have the following structure:



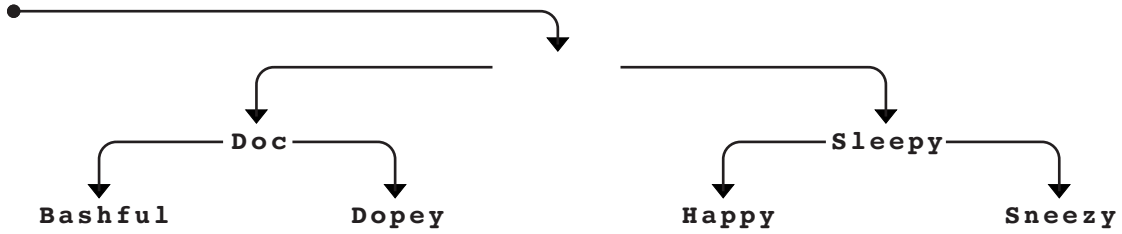
Removing **Sneezy** (for creating an unhealthy work environment) is easy. All you have to do is replace the pointer to the **Sneezy** node with a **NULL** pointer, which produces the following tree:



Starting from this configuration, it is also relatively easy to remove **Sleepy** (who has trouble staying awake on the job). If either child of the node you want to remove is **NULL**, all you have to do is replace it with its non-**NULL** child, like this:

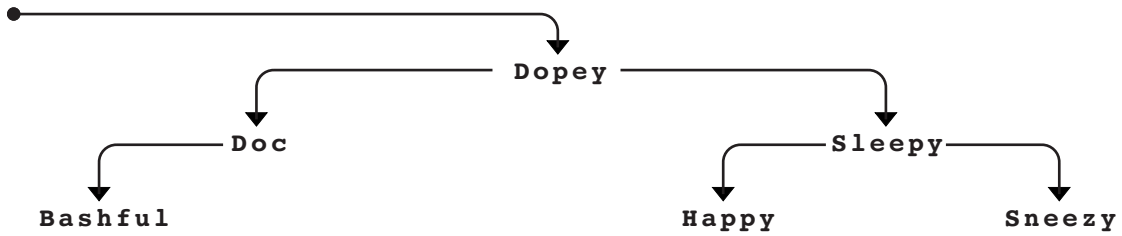


The problem arises if you try to remove a node with both a left and a right child. Suppose, for example, that you instead want to remove **Grumpy** (for failure to whistle while working) from the original tree containing all seven dwarves. If you simply remove the **Grumpy** node, you're left with two partial search trees, one rooted at **Doc** and one rooted at **Sleepy**, as follows:



How can you patch things back together so that you have a valid binary search tree?

At this point, what you would like to do is find a node that can be inserted into the empty space left behind by the removal of the **Grumpy** node. To ensure that the resulting tree remains a binary search tree, there are only two nodes you can use: the rightmost node in the left subtree or the leftmost node in the right subtree. These two nodes work equally well, and you can write the removal algorithm using either one. For example, if you choose the rightmost node in the left subtree, you get the **Dopey** node, which is guaranteed to be larger than anything else in the left subtree but smaller than the values in the right subtree. To complete the removal, all you have to do is replace the **Dopey** node with its left child—which may be **NULL**, as it is in this example—and then move the **Dopey** node into the deleted spot. The resulting picture looks like this:



Implementing the binary search tree package

The code for the generalized binary search tree package, including the details of the removal algorithm described in the preceding section, appears in Figure 13-10 and 13-11. This implementation does not include the AVL extension that keeps the nodes balanced. Allowing the tree to become out of balance increases the running time of the algorithm but does not compromise its correctness. You will have a chance to implement the self-balancing features in the exercises.

Figure 13-10 Private data for the BST class

```
/*
 * File: bstpriv.h
 * ---------------
 * This file contains the private section for the BST class.
 */

/* Type for tree node */

    struct nodeT {
       ElemType data;
       nodeT *left, *right;
    };

/* Instance variables */

    nodeT *root;                          /* Pointer to root node */
    int (*cmpFn)(ElemType, ElemType);     /* Comparison function  */

/* Private function prototypes */

    nodeT *recFindNode(nodeT *t, ElemType key);
    bool recAddNode(nodeT * & t, ElemType key);
    bool recRemoveNode(nodeT * & t, ElemType key);
    void removeTargetNode(nodeT * & t);
    void recFreeTree(nodeT *t);

    template <typename ClientType>
    void recMapAll(nodeT *t, void (*fn)(ElemType, ClientType &),
                   ClientType &data);
```

Figure 13-11 Implementation of the BST class

```
/*
 * File: bstimpl.cpp
 * -----------------
 * This file implements the bst.h interface, which provides a
 * general implementation of binary search trees.
 */

/*
 * Implementation notes for the constructor
 * ----------------------------------------
 * The constructor sets root to NULL to indicate an empty tree.
 * It also stores the client-supplied comparison function for
 * later use; this value defaults to OperatorCmp if no value
 * is supplied.
 */

template <typename ElemType>
BST<ElemType>::BST(int (*cmpFn)(ElemType, ElemType)) {
    root = NULL;
    this.cmpFn = cmpFn;
}
```

```
/*
 * Implementation notes for the destructor
 * ---------------------------------------
 * The destructor must delete all the nodes in the tree.  To
 * do so, it calls recFreeNode, which does a recursive postorder
 * walk on the tree, deleting all nodes in the subtrees before
 * deleting the current node.
 */

template <typename ElemType>
BST<ElemType>::~BST() {
    recFreeTree(root);
}

template <typename ElemType>
void BST<ElemType>::recFreeTree(nodeT *t) {
    if (t != NULL) {
        recFreeTree(t->left);
        recFreeTree(t->right);
        delete t;
    }
}


/*
 * Implementation notes: find, recFindNode
 * ---------------------------------------
 * The find function simply calls recFindNode to do the work.  The
 * recursive function takes the current node along with the original
 * argument. If found, it returns a pointer to the matching data.
 */

template <typename ElemType>
ElemType *BST<ElemType>::find(ElemType key) {
    nodeT *found = recFindNode(root, key);
    if (found != NULL) {
        return &found->data;
    }
    return NULL;
}

template <typename ElemType>
typename BST<ElemType>::nodeT *
         BST<ElemType>::recFindNode(nodeT *t, ElemType key) {
    if (t == NULL) return NULL;
    int sign = cmpFn(key, t->data);
    if (sign == 0) return t;
    if (sign < 0) {
        return recFindNode(t->left, key);
    } else {
        return recFindNode(t->right, key);
    }
}
```

```
/*
 * Implementation notes: add, recAddNode
 * -------------------------------------
 * The add function is implemented as a simple wrapper to recAddNode,
 * which does all the work.  The recAddNode function takes an extra
 * argument, which is a reference to the root of the current subtree.
 */

template <typename ElemType>
bool BST<ElemType>::add(ElemType data) {
    return recAddNode(root, data);
}
template <typename ElemType>
bool BST<ElemType>::recAddNode(nodeT * & t, ElemType data) {
    if (t == NULL) {
        t = new nodeT;
        t->data = data;
        t->left = t->right = NULL;
        return true;
    }
    int sign = cmpFn(data, t->data);
    if (sign == 0) {
        t->data = data;
        return false;
    } else if (sign < 0) {
        return recAddNode(t->left, data);
    } else {
        return recAddNode(t->right, data);
    }
}

/*
 * Implementation notes: remove, recRemoveNode
 * -------------------------------------------
 * The first step in removing a node is to find it using binary
 * search, which is performed by these two functions.  If the
 * node is found, removeTargetNode does the actual deletion.
 */

template <typename ElemType>
bool BST<ElemType>::remove(ElemType data) {
    return recRemoveNode(root, data);
}

template <typename ElemType>
bool BST<ElemType>::recRemoveNode(nodeT *& t, ElemType data) {
    if (t == NULL) return false;
    int sign = cmpFn(data, t->data);
    if (sign == 0) {
        removeTargetNode(t);
        return true;
    } else if (sign < 0) {
        return recRemoveNode(t->left, data);
    } else {
        return recRemoveNode(t->right, data);
    }
}
```

```
/*
 * Implementation notes: removeTargetNode
 * --------------------------------------
 * This function removes the node which is passed by reference as t.
 * The easy case occurs when either of the  children is NULL: all
 * you need to do is replace the node with its non-NULL child.
 * If both children are non-NULL, this code finds the rightmost
 * descendent of the left child; this node may not be a leaf, but
 * will have no right child.  Its left child replaces it in the
 * tree, after which the replacement node is moved to the position
 * occupied by the target node.
 */

template <typename ElemType>
void BST<ElemType>::removeTargetNode(nodeT * & t) {
    nodeT *toDelete = t;
    if (t->left == NULL) {
        t = t->right;
    } else if (t->right == NULL) {
        t = t->left;
    } else {
        nodeT *newRoot = t->left;
        nodeT *parent = t;
        while (newRoot->right != NULL) {
            parent = newRoot;
            newRoot = newRoot->right;
        }
        if (parent != t) {
            parent->right = newRoot->left;
            newRoot->left = t->left;
        }
        newRoot->right = t->right;
        t = newRoot;
    }
    delete toDelete;
}

/* Implementation of the mapping functions */

template <typename ElemType>
template <typename ClientType>
void BST<ElemType>::mapAll(void (*fn)(ElemType, ClientType &),
                           ClientType &data) {
    recMapAll(root, fn, data);
}

template <typename ElemType>
template <typename ClientType>
void BST<ElemType>::recMapAll(nodeT *t,
                              void (*fn)(ElemType, ClientType &),
                              ClientType &data) {
    if (t != NULL) {
        recMapAll(t->left, fn, data);
        fn(t->data, data);
        recMapAll(t->right, fn, data);
    }
}
```

Implementing the `map.h` interface using binary trees

Once you have defined and implemented a class template for binary search trees, you can use the class as part of other applications. For example, you can easily use the **`BST`** class to reimplement the **`Map`** class from Chapter 12. If you think about the problem in terms of what the map needs—as opposed to the details of the binary search tree itself—an entry consists of a key and a value. You would define the type **`pairT`** as this pair and change the data members of the **`Map`** class to a **`BST`** containing such pairs. Here is the private section of the **`Map`** class with these changes:

```
private:

  struct pairT {
      string key;
      ValueType value;
  };

  BST<pairT> bst;

/* private helper function to compare two pairTs */

  static int ComparePairByKey(pairT one, pairT two);
```

When you have access to the **`BST`** class template, writing the code to implement the **`Map`** constructor, **`put`**, and **`get`** methods becomes a simple task. The code for these functions appears in Figure 13-12.

Using the `static` keyword

If you examine the implementation of the **`Map`** class template closely, you will note that the declaration of the **`ComparePairByKey`** method in the class interface is marked with

Figure 13-12 Implementation of maps using binary search trees

```
/*
 * File: bstmap.cpp
 * ----------------
 * This file implements the Map class layered on top of BSTs.
 */

template <typename ValueType>
Map<ValueType>::Map() : bst(ComparePairByKey) {
    /* Empty */
}

template <typename ValueType>
Map<ValueType>::~Map() {
    /* Empty */
}

template <typename ValueType>
void Map<ValueType>::put(string key, ValueType value) {
    pairT pair;
    pair.key = key;
    pair.value = value;
    bst.add(pair);
}
```

```
template <typename ValueType>
ValueType Map<ValueType>::get(string key) {
    pairT pair, *found;
    pair.key = key;
    if ((found = bst.find(pair)) != NULL) {
        return found->value;
     }
     Error("getValue called on non-existent key");
  }

/*
 * This static function is used to compare two pairs. It
 * ignores the value fields and just returns the ordering
 * of the two key fields. This function must be declared static
 * so that is _not_ a method of the class and thus
 * is not expected to be invoked on a receiver object.
 */

template <typename ValueType>
int Map<ValueType>::ComparePairByKey(pairT one, pairT two) {
    if (one.key == two.key) return 0;
    return (one.key < two.key) ? -1 : 1;
}
```

the keyword **static**. Within a class interface, the **static** keyword is used to identify members that are shared across the class, and not specific to a particular object or instance. You have previously used the **static** keyword when declaring class constants.

The distinction between static and non-static functions of a class is subtle, but important. By default, a method is assumed to operate on a particular object. When you call such a method, you identify which object is being acted upon by specifying the receiver in the call, as in

```
obj.performAction()
```

By declaring **ComparePairByKey** as **static**, the function is modified so that it becomes associated with the class itself and is not be invoked on a specific receiver object. This function must be declared **static** in order to be compatible with the type of comparison function used by the **BST** class. The **BST** class assumes the comparison callback has the form of a free function that takes two arguments, the two values being compared, and does not invoke the callback on a receiver object.

Any **static** methods you declare are still considered part of the class implementation, and thus have access to the private internals, such as the definition of the **Map** class **pairT** type. However, in the body of a **static** method, there is no reference to **this** (because there is no receiving object) and thus no access to data members.

Summary

In this chapter, you have been introduced to the concept of *trees,* which are hierarchical collections of nodes that obey the following properties:

- There is a single node at the top that forms the root of the hierarchy.
- Every node in the tree is connected to the root by a unique line of descent.

Important points in this chapter include:

- Many of the terms used to describe trees, such as *parent, child, ancestor, descendant,* and *sibling,* come directly from family trees. Other terms, including *root* and *leaf,* are derived from trees in nature. These metaphors make the terminology used for trees easy to understand because the words have the same interpretation in computer science as they do in these more familiar contexts.
- Trees have a well-defined recursive structure because every node in a tree is the root of a subtree. Thus, a tree consists of a node together with its set of children, each of which is a tree. This recursive structure is reflected in the underlying representation for trees, which are defined as a pointer to a **`nodeT`**, and the type **`nodeT`** is defined as a record containing values of type pointer to a **`nodeT`**.
- Binary trees are a subclass of trees in which nodes have at most two children and every node except the root is designated as either a left child or a right child of its parent.
- If a binary tree is organized so that every node in the tree contains a key field that follows all the keys in its left subtree and precedes all the keys in its right subtree, that tree is called a *binary search tree*. As its name implies, the structure of a binary search tree permits the use of the binary search algorithm, which makes it possible to find individual keys more efficiently. Because the keys are ordered, it is always possible to determine whether the key you're searching for appears in the left or right subtree of any particular node.
- Using recursion makes it easy to step through the nodes in a binary search tree, which is called *traversing* or *walking* the tree. There are several types of traversals, depending on the order in which the nodes are processed. If the key in each node is processed before the recursive calls to process the subtrees, the result is a *preorder* traversal. Processing each node after both recursive calls gives rise to a *postorder* traversal. Processing the current node between the two recursive calls represents an *inorder* traversal. In a binary search tree, the inorder traversal has the useful property that the keys are processed in order.
- Depending on the order in which nodes are inserted, given the same set of keys, binary search trees can have radically different structures. If the branches of the tree differ substantially in height, the tree is said to be unbalanced, which reduces its efficiency. By using techniques such as the AVL algorithm described in this chapter, you can keep a tree in balance as new nodes are added.
- It is possible to design an interface for binary search trees that allows the client to control the data of the individual nodes by using C++ templates. The **`BST`** class template that appears in Figure 13-9 exports a flexible implementation of the binary search tree structure that can be used in a wide variety of applications.

Review questions

1. What two conditions must be satisfied for a collection of nodes to be a tree?

2. Give at least four real-world examples that involve tree structures.

3. Define the terms *parent, child, ancestor, descendant,* and *sibling* as they apply to trees.

4. The family tree for the House of Tudor, which ruled England in Shakespeare's time, is shown in Figure 13-13. Identify the root, leaf, and interior nodes. What is the height of this tree?

5. What is it about trees that makes them recursive?

6. Diagram the internal structure of the tree shown in Figure 13-13 when it is represented as a **`familyNodeT`**.

7. What is the defining property of a binary search tree?

8. Why are different type declarations used for the first argument in **`FindNode`** and **`InsertNode`**?

9. In *The Hobbit* by J. R. R. Tolkien, 13 dwarves arrive at the house of Bilbo Baggins in the following order: **`Dwalin`**, **`Balin`**, **`Kili`**, **`Fili`**, **`Dori`**, **`Nori`**, **`Ori`**, **`Oin`**, **`Gloin`**, **`Bifur`**, **`Bofur`**, **`Bombur`**, and **`Thorin`**. Diagram the binary search tree that results from inserting the names of these dwarves into an empty tree.

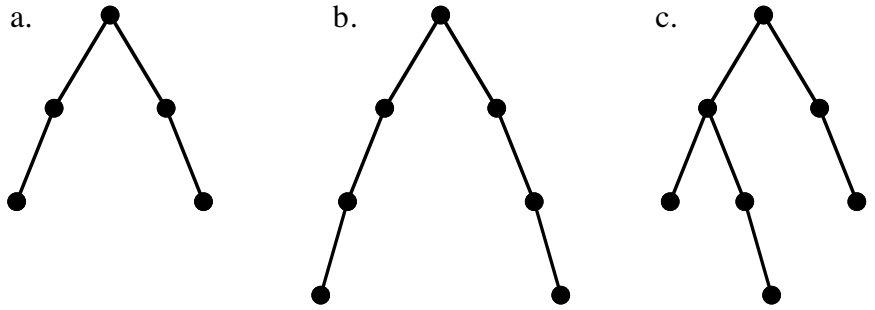
10. Given the tree you created in the preceding question, what key comparisons are made if you call **`FindNode`** on the name **`Bombur`**?

11. Write down the preorder, inorder, and postorder traversals of the binary search tree you created for question 9.

12. One of the three standard traversal orders—preorder, inorder, or postorder—does not depend on the order in which the nodes are inserted into the tree. Which one is it?

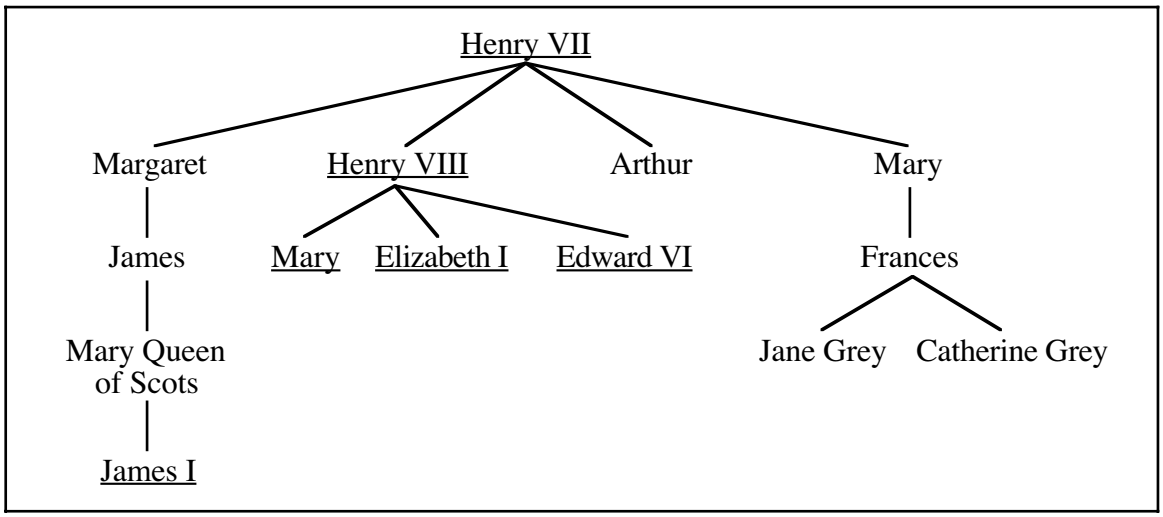
13. What does it mean for a binary tree to be balanced?

14. For each of the following tree structures, indicate whether the tree is balanced:



For any tree structure that is out of balance, indicate which nodes are out of balance.

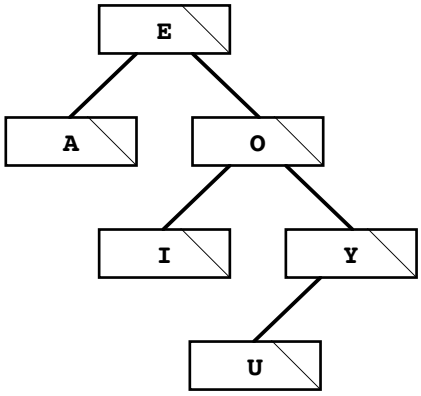
Figure 13-13 Family tree for the House of Tudor



15. True or false: If a binary search tree becomes unbalanced, the algorithms used in the functions **`FindNode`** and **`InsertNode`** will fail to work correctly.

16. How do you calculate the balance factor of a node?

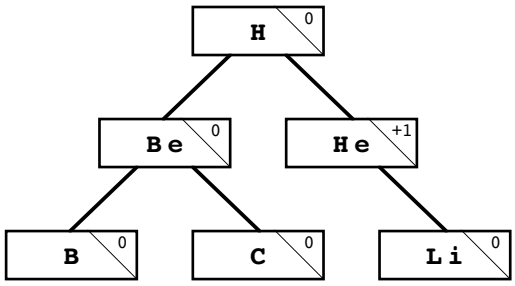
17. Fill in the balance factors for each node in the following binary search tree:



18. If you use the AVL balancing strategy, what rotation operation must you apply to the tree in the preceding question to restore its balanced configuration? What is the structure of the resulting tree, including the updated balance factors?

19. True or false: When you insert a new node into a balanced binary tree, you can always correct any resulting imbalance by performing one operation, which will be either a single or a double rotation.

20. As shown in the section on "Illustrating the AVL idea," inserting the symbols for the first six elements into an AVL tree results in the following configuration:



Show what happens to the tree as you add the next six element symbols:

`N` (Nitrogen)
`O` (Oxygen)
`F` (Fluorine)
`Ne` (Neon)
`Na` (Sodium)
`Mg` (Magnesium)

21. Describe in detail what happens when the **`add`** method is called.

22. What strategy does the text suggest to avoid having a binary search tree become disconnected if you remove an interior node?

Programming exercises

1. Working from the definition of **familyNodeT** given in the section entitled "Representing family trees in C++," write a function

```
familyNodeT *ReadFamilyTree(string filename);
```

that reads in a family tree from a data file whose name is supplied as the argument to the call. The first line of the file should contain a name corresponding to the root of the tree. All subsequent lines in the data file should have the following form:

child:*parent*

where *child* is the name of the new individual being entered and *parent* is the name of that child's parent, which must appear earlier in the data file. For example, if the file **normandy.dat** contains the lines

```
William I
Robert:William I
William II:William I
Adela:William I
Henry I:William I
Stephan:Adela
William:Henry I
Matilda:Henry I
Henry II:Matilda
```

calling **ReadFamilyTree("normandy.dat")** should return the family-tree structure shown in Figure 13-2.

2. Write a function

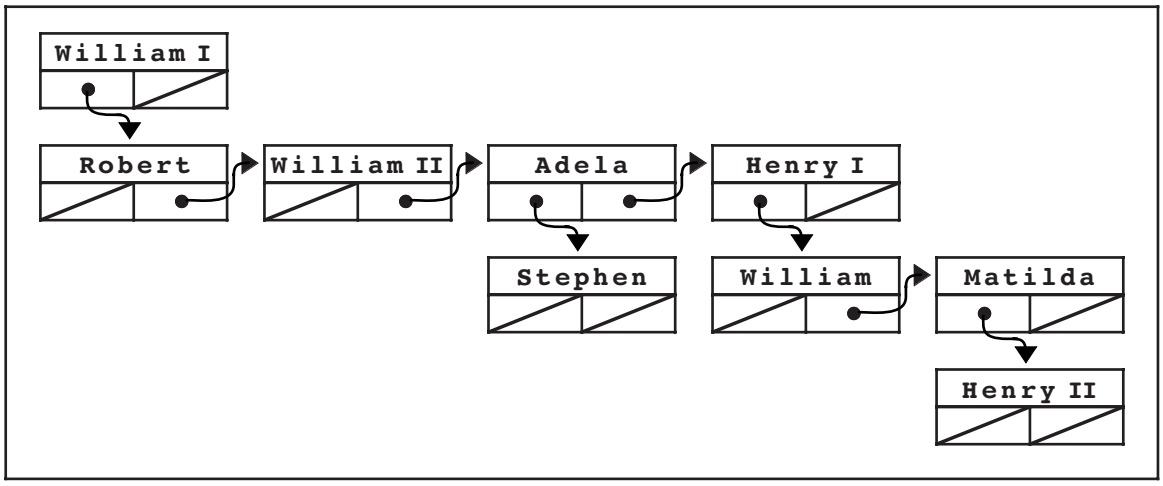
```
void DisplayFamilyTree(familyNodeT *tree);
```

that displays all the individuals in a family tree. To record the hierarchy of the tree, the output of your program should indent each generation so that the name of each child appears two spaces to the right of the corresponding parent, as shown in the following sample run:

```
William I
  Robert
  William II
  Adela
    Stephan
  Henry I
    William
    Matilda
      Henry II
```

3. As defined in the chapter, the **familyNodeT** structure uses a vector to store the children. Another possibility is to include an extra pointer in these nodes that will allow them to form a linked list of the children. Thus, in this design, each node in the tree needs to contain only two pointers: one to its eldest child and one to its next younger sibling. Using this representation, the House of Normandy appears as shown in Figure 13-14. In each node, the pointer on the left always points down to a child; the pointer on the right indicates the next sibling in the same generation. Thus, the eldest child of William I is Robert, which you obtain by following the link at the left of the diagram. The remaining children are linked together through the link cells

Figure 13-14 House of Normandy using a linked list of siblings



shown at the right of the node diagram. The chain of children ends at Henry I, which has the value **NULL** in its next-sibling link.

Using the linked design illustrated in this diagram, write new definitions of **familyNodeT**, **ReadFamilyTree**, and **DisplayFamilyTree**.

4. In exercise 3, the changes you made to **familyNodeT** forced you to rewrite the functions—specifically **ReadFamilyTree** and **DisplayFamilyTree**—that depend on that representation, such as . If the family tree were instead represented as a class that maintained its interface despite any changes in representation, you could avoid much of this recoding. Such an interface appears in Figure 13-15. Write the corresponding implementation using a vector to store the list of children.

   Note that the class exported by the **famtree.h** interface corresponds to an individual node rather than to the tree as a whole. From each node, you can find the parent using **getParent** and the children using **getChildren**.

5. Using the **famtree.h** interface defined in the preceding exercise, write a function

   ```
   FamilyTreeNode *FindCommonAncestor(FamilyTreeNode *p1,
                                      FamilyTreeNode *p2);
   ```

   that returns the closest ancestor shared by **p1** and **p2**.

6. Write a function

   ```
   int Height(nodeT *tree);
   ```

   that takes a binary search tree—using the definition of **nodeT** from section 13.2—and returns its height.

7. Write a function

   ```
   bool IsBalanced(nodeT *tree);
   ```

   that determines whether a given tree is balanced according to the definition in the section on "Balanced trees." To solve this problem, all you really need to do is translate the definition of a balanced tree more or less directly into code. If you do so, however, the resulting implementation is likely to be relatively inefficient because it has to make several passes over the tree. The real challenge in this

Figure 13-15 Interface for a class that supports the representation of family trees

```
class FamilyTreeNode {

public:

/*
 * Constructor: FamilyTreeNode
 * Usage: FamilyTreeNode *person = new FamilyTreeNode(name);
 * ---------------------------------------------------------
 * This function constructs a new FamilyTreeNode with the specified
 * name.  The newly constructed entry has no children, but clients
 * can add children by calling the addChild method.
 */

    FamilyTreeNode(string name);

/*
 * Method: getName
 * Usage: string name = person->getName();
 * --------------------------------------
 * Returns the name of the person.
 */

    string getName();

/*
 * Method: addChild
 * Usage: person->addChild(child);
 * -------------------------------
 * Adds child to the end of the list of children for person, and
 * makes person the parent of child.
 */

    void addChild(FamilyTreeNode *child);

/*
 * Method: getParent
 * Usage: FamilyTreeNode *parent = person->getParent();
 * ----------------------------------------------------
 * Returns the parent of the specified person.
 */

    FamilyTreeNode *getParent();

/*
 * Method: getChildren
 * Usage: Vector<FamilyTreeNode *> children = person->getChildren();
 * -----------------------------------------------------------------
 * Returns a vector of the children of the specified person.
 * Note that this vector is a copy of the one in the node, so
 * that the client cannot change the tree by adding or removing
 * children from this vector.
 */

    Vector<FamilyTreeNode *> getChildren();

/* Whatever private section you need */

}
```

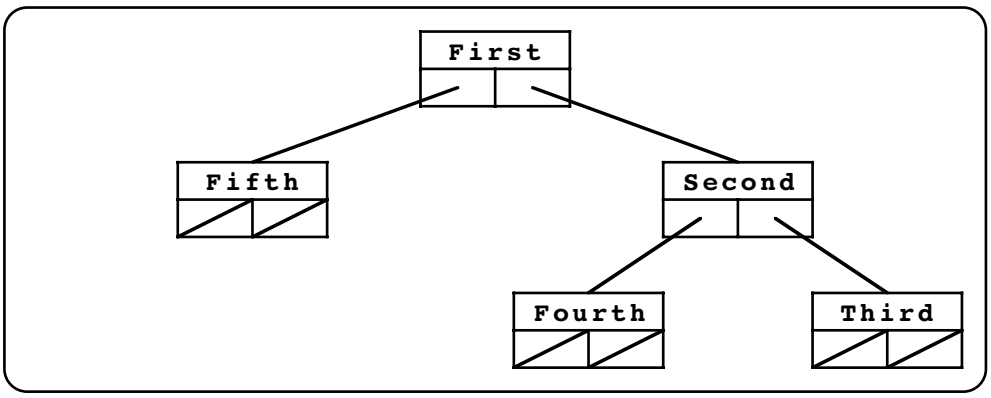
problem is to implement the **IsBalanced** function so that it determines the result without looking at any node more than once.

8. Write a function

```
bool HasBinarySearchProperty(nodeT *tree);
```

that takes a tree and determines whether it maintains the fundamental property that defines a binary search tree: that the key in each node follows every key in its left subtree and precedes every key in its right subtree.

9. Write a test program for the **BST** class that uses the graphics library described in section 6.3 to display the structure of the tree. For example, if you insert the keys **First**, **Second**, **Third**, **Fourth**, and **Fifth** into a binary search tree without balancing, your program should display the following diagram in the graphics window:



Including the keys as part of the node diagram will require you to use the extended version of the graphics library interface, **extgraph.h**, which is available for many systems as part of the Addison-Wesley software archive. Even without it, you can construct the line drawing for the nodes from the simple commands available in the simpler **graphics.h** interface.

10. Extend the implementation of the **BST** class template so that it uses the AVL algorithm to keep the tree balanced as new nodes are inserted. The algorithm for balanced insertion is coded in Figure 13-7. Your task in this problem is simply to integrate this algorithm into the more general implementation of binary search trees given in Figure 13-11.

11. Integrating the AVL algorithm for inserting a node into the **bst.cpp** implementation only solves part of the balancing problem for the generalized **BST** class. Because the **BST** class interface also exports a function to remove a node from the tree, the complete implementation of the package must also rebalance the tree when a node is removed. The structure of the algorithm to rebalance after removal is quite similar to that for insertion. Removing a node either may have no effect on the height of a tree or may shorten it by one. If a tree gets shorter, the balance factor in its parent node changes. If the parent node becomes out of balance, it is possible to rebalance the tree at that point by performing either a single or a double rotation.

Revise the implementation of the **remove** method so that it keeps the underlying AVL tree balanced. Think carefully about the various cases that can arise and make sure that your implementation handles each of these cases correctly.

12. From a practical standpoint, the AVL algorithm is too aggressive. Because it requires that the heights of the subtrees at each node never differ by more than one, the AVL algorithm spends quite a bit of time performing rotation operations to correct imbalances that occur as new nodes are inserted. If you allow trees to become somewhat more unbalanced—but still keep the subtrees relatively similar—you can reduce the balancing overhead significantly.

    One of the most popular techniques for managing binary search trees is called *red-black trees*. The name comes from the fact that every node in the tree is assigned a color, either red or black. A binary search tree is a legal red-black tree if all three of the following properties hold:

    1. The root node is black.
    2. The parent of every red node is black.
    3. Every path from the root to a leaf contains the same number of black nodes.

    These properties ensure that the longest path from the root to a leaf can never be more than twice the length of the shortest path. Given the rules, you know that every such path has the same number of black nodes, which means that the shortest possible path is composed entirely of black nodes, and the longest has black and red nodes alternating down the chain. Although this condition is less strict than the definition of a balanced tree used in the AVL algorithm, it is sufficient to guarantee that the operations of finding and inserting new nodes both run in logarithmic time.

    The key to making red-black trees work is finding an insertion algorithm that allows you to add new nodes while maintaining the conditions that define red-black trees. The algorithm has much in common with the AVL algorithm and uses the same rotation operations. The first step is to insert the new node using the standard insertion algorithm with no balancing. The new node always replaces a **`NULL`** entry at some point in the tree. If the node is the first node entered into the tree, it becomes the root and is therefore colored black. In all other cases, the new node must initially be colored red to avoid violating the rule that every path from the root to a leaf must contain the same number of black nodes.

    As long as the parent of the new node is black, the tree as a whole remains a legal red-black tree. The problem arises if the parent node is also red, which means that the tree violates the second condition, which requires that every red node have a black parent. In this case, you need to restructure the tree to restore the red-black condition. Depending on the relationship of the red-red pair to the remaining nodes in the tree, you can eliminate the problem by performing one of the following operations:

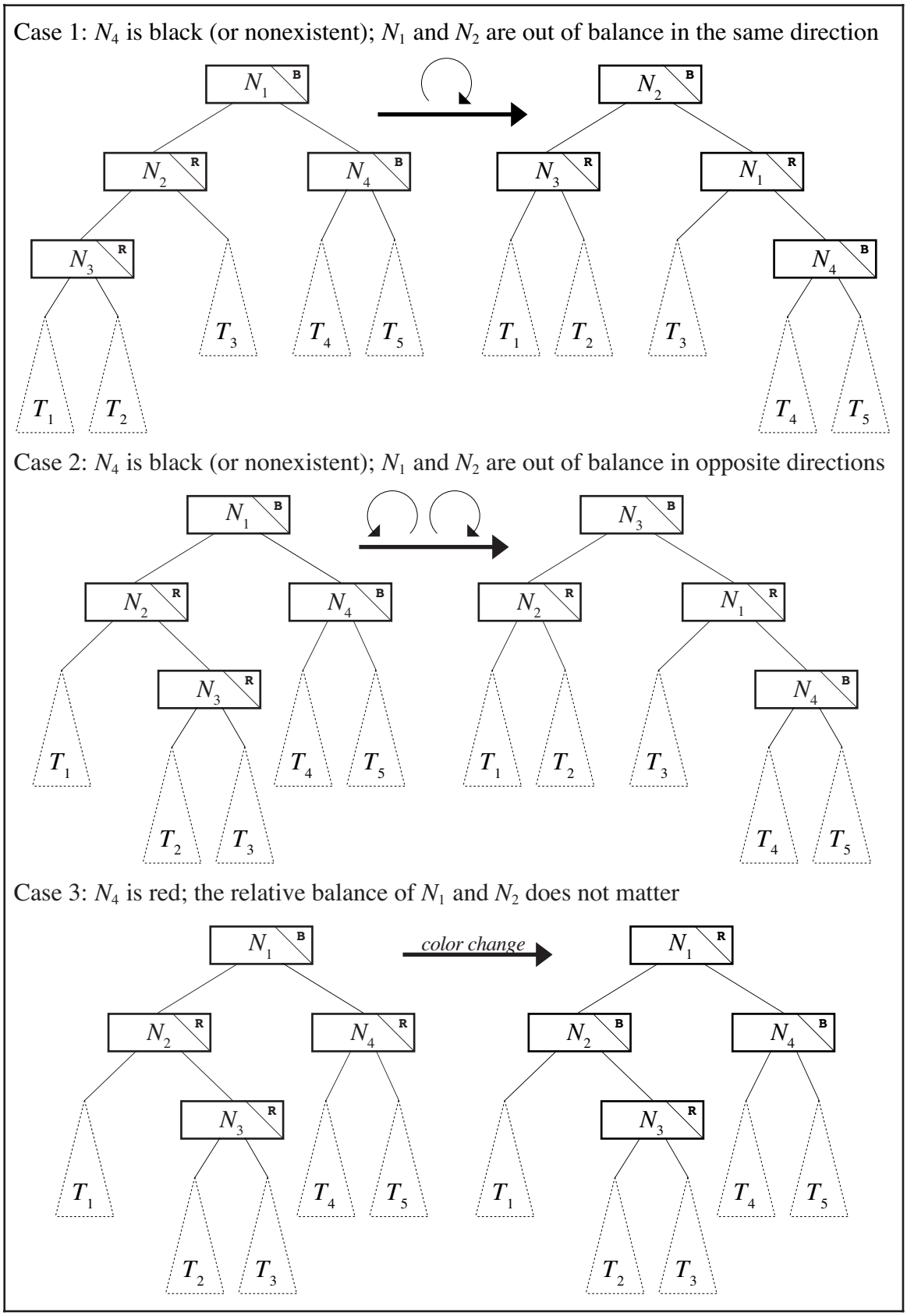
    1. A single rotation, coupled with a recoloring that leaves the top node black.
    2. A double rotation, coupled with a recoloring that leaves the top node black.
    3. A simple change in node colors that leaves the top node red and may therefore require further restructuring at a higher level in the tree.

    These three operations are illustrated in Figure 13-16. The diagram shows only the cases in which the imbalance occurs on the left side. Imbalances on the right side are treated symmetrically.

    Change the implementation of the **`BST`** class template in Figures 13-10 and 13-11 so that it uses red-black trees to maintain balance.

13. Complete the BST-based implementation of the **`Map`** class, which appears in a partial form in Figure 13-12. The missing methods are **`mapAll`** and **`remove`**.

Figure 13-16 Balancing operations on red-black trees



14. Trees have many applications beyond those listed in this chapter. For example, trees can be used to implement a lexicon, which was introduced in Chapter 4. The resulting structure, first developed by Edward Fredkin in 1960, is called a **trie.** (Over time, the pronunciation of this word has evolved to the point that it is now pronounced like *try,* even though the name comes from the central letters of *retrieval*.) The trie-based implementation of a lexicon, while somewhat inefficient in its use of space, makes it possible to determine whether a word is in the lexicon much more quickly than you can using a hash table.

    At one level, a trie is simply a tree in which each node branches in as many as 26 ways, one for each possible letter of the alphabet. When you use a trie to represent a lexicon, the words are stored implicitly in the structure of the tree and represented as a succession of links moving downward from the root. The root of the tree corresponds to the empty string, and each successive level of the tree corresponds to the subset of the entire word list formed by adding one more letter to the string represented by its parent. For example, the *A* link descending from the root leads to the subtree containing all the words beginning with *A,* the B link from that node leads to the subtree containing all the words beginning with *AB,* and so forth. Each node is also marked with a flag indicating whether the substring that ends at that particular point is a legitimate word.

    The structure of a trie is much easier to understand by example than by definition. Figure 13-17 shows a trie containing the symbols for the first six elements—**`H`**, **`He`**, **`Li`**, **`Be`**, **`B`**, and **`C`**. The root of the tree corresponds to the empty string, which is not a legal symbol, as indicated by the designation **`no`** in the field at the extreme right end of the structure. The link labeled **`B`** from the node at the root of the trie descends to a node corresponding to the string **`"B"`**. The rightmost field of this node contains **`yes`**, which indicates that the string **`"B"`** is a complete symbol in its own right. From this node, the link labeled **`E`** leads to a new node, which indicates that the string **`"BE"`** is a legal symbol as well. The **`NULL`** pointers in the trie indicate that no legal symbols appear in the subtree beginning with that substring and therefore make it possible to terminate the search process.

    Remplement the **`Lexicon`** class that uses a trie as its internal representation. Your implementation should be able to read text files but not the binary ones.

Figure 13-17 Trie containing the element symbols `H`, `He`, `Li`, `Be`, `B`, and `C`