[CHAPTER] 7

Hierarchical Data Structures

7.1 Introduction

7.2 Trees

7.3 Binary Trees

- 7.3.1 Introduction
- 7.3.2 Operations on Binary Trees
- 7.3.3 The Binary Tree Representation of General Trees
- 7.3.4 The Reference-Based Implementation of Binary Trees
- 7.3.5 An Array-Based Implementation of Binary Trees

7.4 Binary Search Trees

- 7.4.1 Definition
- 7.4.2 Using Binary Search Trees in Searching Operations
- 7.4.3 Tree Sort

7.5 Balanced Binary Search Trees

- 7.5.1 Introduction
- 7.5.2 Red-Black Trees

7.6 Heaps

- 7.6.1 Definition
- 7.6.2 Implementation of Heaps Using One-Dimensional Arrays
- 7.6.3 Application of Heaps

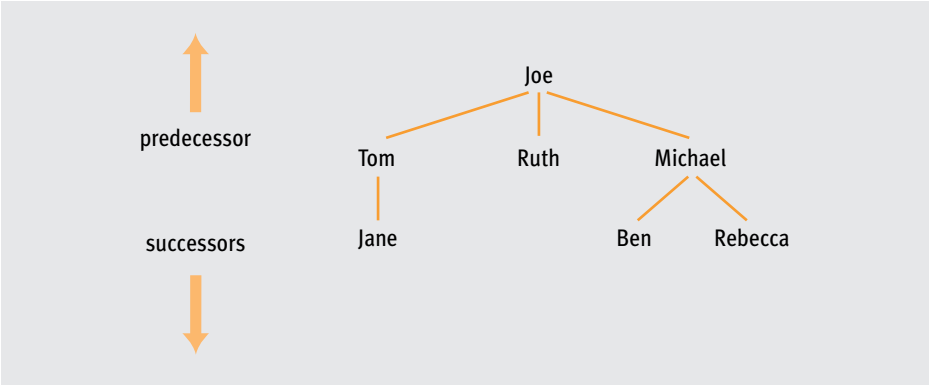
7.7 The Importance of Good Approximations

7.8 Summary

7.1 Introduction

This chapter investigates a class of data structures that are quite different from the lists, stacks, and queues of Chapter 6. Those structures are linear—if they are not empty, they have a first and last node, and every node except the first and last has a unique successor and predecessor node. Now we begin our look at **hierarchical data structures**, which we first diagrammed in Figure 6-2. Although each element in these structures still has a single predecessor, it may have zero, one, or more successors. In computer science, hierarchical structures are usually referred to as **trees**.

Trees should be familiar from everyday life. For example, everyone has a family tree of the type shown in Figure 7-1. Sporting competitions, management structures, and term paper outlines are often displayed as trees as well.

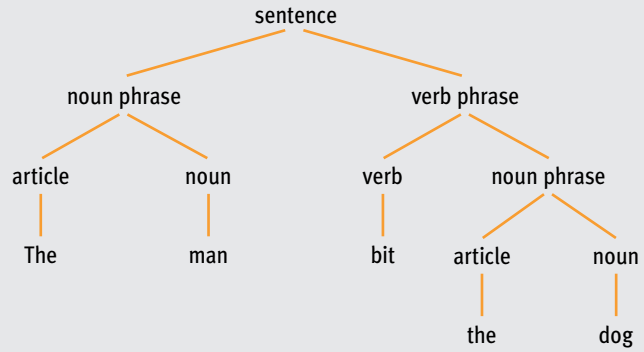


[FIGURE 7-1] Family tree

We can see in Figure 7-1 that every node in the tree except Joe has exactly one predecessor, but that a node may have zero (Ruth), one (Tom), or many (Joe, Michael) successors.

> *Unlike in nature, trees in computer science grow down, not up. Therefore, tree diagrams in this chapter display the node's predecessor above the node, while the node's successor(s) are below the node.*

Trees have many applications in computer science. For example, a compiler constructs a **parse tree** as part of the translation process. A parse tree is a hierarchical data structure that reflects the grammatical relationships between syntactic elements of a language. You probably first encountered parse trees in elementary school when you learned how to diagram a sentence. For example, Figure 7-2 is a parse tree for the sentence "The man bit the dog."



[FIGURE 7-2] Example parse tree

When a compiler analyzes a computer program to determine if it is syntactically correct, it attempts to generate a parse tree like the one in Figure 7-2.

Let's illustrate this process by showing how a compiler might determine the syntactic validity of some simple arithmetic expressions. A four-rule grammar for these expressions follows:

rule 1: <expression> → <term> { **+** | **–** <term> }
rule 2: <term> → <factor> { ***** <factor> }
rule 3: <factor> → <letter> | **(** <expression> **)**
rule 4: <letter> → **a** | **b** | **c**

Each of these four lines is a rule that defines part of the syntax of an arithmetic expression. The symbol → stands for "is defined as"; the grammatical symbol on the left side of the → operator can be replaced by, or expanded into, the sequence of symbols on the right side. The braces ({}) mean that zero, one, or more repetitions of the information are enclosed inside the braces, and the vertical bar (|) means that you select exactly one of the symbols on either side. Thus, the grammatical expression $a\ \{b\}$ can represent the infinite sequence of strings a, ab, abb, $abbb$, $abbbb$, and so on, while $a|b$ means that you may choose either a or b, but not both.

Symbols set in boldface are **terminal** symbols, which represent the actual elements of the language being defined. Symbols inside < > are **nonterminal** symbols, which represent intermediate grammatical objects defined further by other rules. These nonterminals are

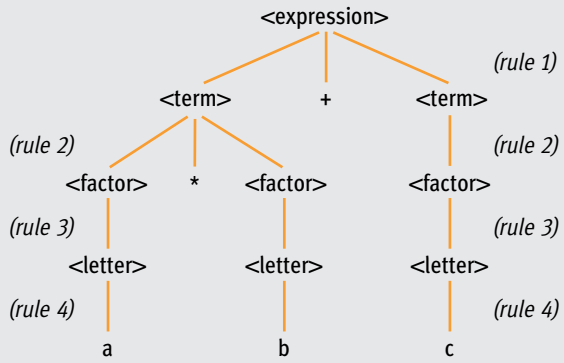
equivalent to the terms *sentence*, *article*, *noun*, *verb phrase*, and *noun phrase* in Figure 7-2. The syntax of every programming language, including Java, is defined by such rules. The entire collection of rules is called the **grammar** of the language.

When you write an arithmetic expression in your program, such as $a + b * c$, the compiler uses a grammar like the one shown previously to try to construct a valid parse tree for the expression. If the compiler can build a parse tree, then the expression is syntactically valid in the language. If the compiler cannot build such a tree, then the expression is invalid, and an appropriate error message is displayed.

To construct the parse tree, the compiler starts with the nonterminal symbol called <expression> in the following example, the top-level object it is trying to validate. It then applies a rule of the grammar to expand this symbol into a sequence of one or more terminal and nonterminal symbols. For example, we can expand <expression> by applying the previous rule 1—selecting the plus sign alternative—to produce the following tree:

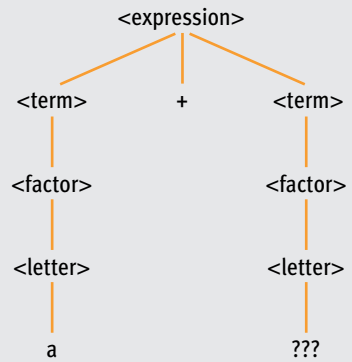
We repeat this process of using rules to expand the nonterminal symbols until we have either constructed the desired expression (generated all the terminal symbols) or we can go no further with the parse.

For example, given the expression $a * b + c$ and the four-rule grammar shown earlier, a compiler could construct the parse tree shown in Figure 7-3, thereby proving that $a * b + c$ is a valid <expression> in this language.



[FIGURE 7-3] Parse tree for $a * b + c$

However, try as we might, no valid parse tree can ever be found for the two-character expression "a +". For example:



We are at a dead end. We have built a parse tree for the expression a + <letter>, where <letter> can be expanded into either a, b, or c, but we cannot construct one for the sequence "a +". Every attempt to construct a tree will fail, and the compiler will reject such expressions with an error message like `***Error: Missing term in expression***`.

Parsing is just one example of how hierarchical data structures are used in computer science. More examples are provided later in this chapter.

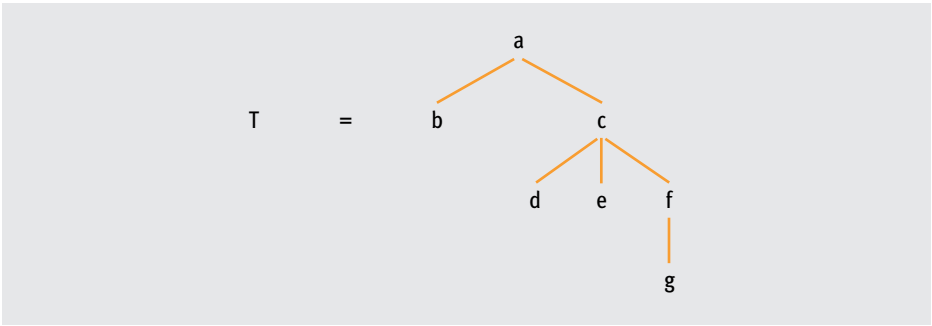
7.2 Trees

A **general tree**, or more simply a **tree**, is a hierarchical data structure containing $k \geq 1$ nodes $N = \{n_1, n_2, \ldots, n_k\}$, connected to each other by exactly $(k - 1)$ links $E = \{e_1, e_2, \ldots, e_{k-1}\}$. One special node in the tree is called the **root** and has no predecessor. The tree also has one or more special nodes called **leaves** that have no successors. All other nodes in the tree are **internal nodes**; they have exactly one predecessor and one or more successors.

Nodes are the components of a tree that store information; they serve the same role as a node in a linked list. Each node contains an information field and zero, one, or more links to other nodes. The links of a tree are often referred to as **edges**.

A fundamental characteristic of a tree T is that its set of N nodes, except for the root $(N - \{r\})$, can be partitioned into zero, one, or more disjoint subsets T_1, T_2, T_3, …, each of which is itself a tree and is called a **subtree** of T. This disjoint partitioning property holds for all subtrees as well.

Let's look at this definition more closely, using the tree T in Figure 7-4.



[FIGURE 7-4] Sample tree structure

The tree T in Figure 7-4 contains the following seven nodes:

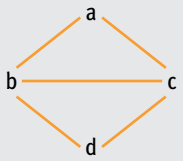
$$T = \{a, b, c, d, e, f, g\}$$

It also has exactly six edges that connect the nodes to each other. The root of the tree is the node containing the value a. It has two disjoint subtrees:

$$T_1 = \{b\} \qquad T_2 = \{c, d, e, f, g\}$$

The root of T_1 is b, and it has no subtrees. The root of T_2 is c, and it has three disjoint subtrees $\{d\}$, $\{e\}$, and $\{f, g\}$.

In contrast, we cannot partition the following four-node structure:



into a root and a collection of disjoint subtrees. If we call the node that contains a the root, then its subtrees would be $\{b, c, d\}$ and $\{c, d, b\}$, which are not disjoint. This type of structure is a graph, not a tree; graphs are discussed in Chapter 8.

The number of successors of a node is called the node's **degree**. Hence, node a in Figure 7-4 has degree 2, node b has degree 0, and node c has degree 3. The leaf nodes of a tree, such as b, d, e, and g in Figure 7-4, are the nodes that have degree 0. All other nodes are **internal nodes**; their degree is always greater than or equal to 1. In Figure 7-4, the internal nodes are a, c, and f.

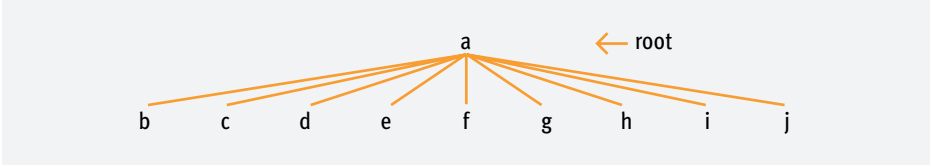
When discussing trees, we often use familial terms to describe relationships between elements in the collection, perhaps because family trees were among the earliest uses of tree structures. For example, node a in Figure 7-4 is said to be the **parent** of nodes b and c, rather than their predecessor, and nodes b and c are called the **children** of node a instead of its successors. Nodes that have the same parent are called **siblings**. In Figure 7-4, nodes b and c are siblings. So are d, e, and f.

A **path** is a sequence of nodes $n_1, n_2, n_3, \ldots, n_k$, such that either n_{i+1} is a child of n_i or n_{i+1} is the parent of n_i, for $i = 1, 2, \ldots, k - 1$. In Figure 7-4, *bac* is a path, as are *dcfg* and *gfcace*. A path in which all nodes are unique is called a **simple path**. Usually, the term *path* means a simple path.

An alternative way to define a tree is to use the definition of path just given: A tree is a set of k nodes and $(k - 1)$ edges, $k \geq 1$, in which there is exactly one simple path between any two nodes. That is, given any two nodes n_i and n_j in the tree, there is only one sequence of unique nodes $n_i \rightarrow n_{i+1} \rightarrow n_{i+2} \rightarrow \ldots \rightarrow n_j$ that take you from node n_i to node n_j. For example, given the tree in Figure 7-4, the unique simple path from a to g is $a \rightarrow c \rightarrow f \rightarrow g$. The unique simple path from d to b is $d \rightarrow c \rightarrow a \rightarrow b$. We can now see why a structure like the four-node graph shown previously is not a tree. Namely, there are multiple paths between nodes. For example, for nodes d and a, there are four simple paths: $d \rightarrow b \rightarrow a$, $d \rightarrow c \rightarrow a$, $d \rightarrow c \rightarrow b \rightarrow a$, and $d \rightarrow b \rightarrow c \rightarrow a$.

The **ancestors** of a node n are all the nodes in the path from node n to the root of the tree, excluding node n itself. For example, in Figure 7-4 the ancestors of node g are $\{f, c, a\}$. The **descendants** of node n are all the nodes contained in the subtree rooted at n, again excluding node n itself. The descendants of node c are $\{d, e, f, g\}$. The **height** of a tree is defined as the length, in terms of the total number of nodes, of the longest path from any node n to the root of the tree. The height of the tree in Figure 7-4 is 4 because the longest simple path from any node to the root contains four nodes: $\{g, f, c, a\}$. Finally, the **level** of a node measures the distance of that node from the root. We assign the root a level number of 1. Then, the level of any node n is defined recursively as 1 + (level of the parent of n). An equivalent definition is that the level of a node is the total number of nodes on the path from that node to the root of the tree. The height of a tree would then be the maximum level number of any node in the tree.

The type of tree we have been describing is called a **general tree**. It is characterized by the fact that its nodes can have arbitrary degree—that is, they may have any number of children, as shown in the following diagram:



General trees are easy to describe, but they can be somewhat difficult to implement, so they are not widely used in computer science. To understand why general trees can be problematic, let's look at how we might implement one. When we attempt to specify the structure of an individual node, we run into trouble.

```
/**
 * A linked node for use in describing general trees
 */

public class GTreeNode<T> {
    // Links to this node's children
    private GTreeNode child1;  // child number 1
    private GTreeNode child2;  // child number 2
    private GTreeNode child3;  // child number 3
    private GTreeNode child4;  // child number 4
    private GTreeNode child5;  // child number 5
    // This node's data
    private T data;

} // GTreeNode
```

[FIGURE 7-5] Possible representation of a node in a general tree

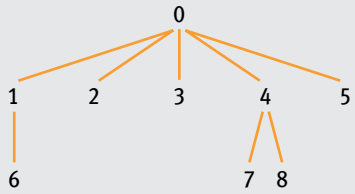
The implementation of a node shown in Figure 7-5 is unsatisfactory because a general tree can have an *arbitrary* number of children, not just five. It does no good to add more fields to Figure 7-5 because there are no restrictions on the maximum number of successors. For any number n of child references we choose to allocate, we might need $n + 1$. Similarly, it does no good to construct an array of child references:

```
private GTreeNode[] child = new GTreeNode[SIZE];
```

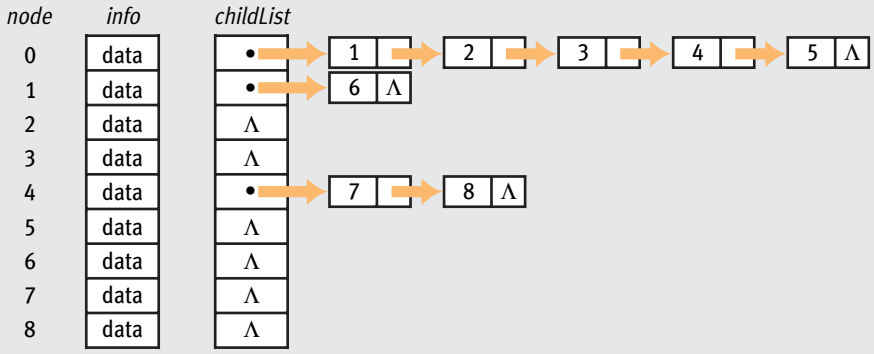
because an array is a fixed-size structure; when created, we have to specify its size—the constant SIZE in the previous declaration. What happens if we have a node with (SIZE + 1) children?

There are certainly ways to represent general trees without encountering this problem. For example, we could create two parallel arrays called `info` and `childList`. Row i of the `info` array contains the information field, and row i of `childList` is the head of a linked list that contains all the children of this node. Because the length of a list is unbounded, the

node could have an arbitrary number of children. For example, to represent the following general tree:



we could use the following representation:



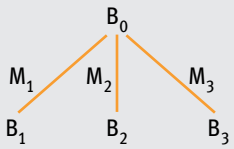
However, an even better solution, and one that leads to a much simpler structure, is to restrict the degree of a node—in other words, to place a limit on the maximum number of children of any node. One of the most important and widely used restricted hierarchical structures is the binary tree, which places a limit of two on the degree of all nodes in the tree. As you will see in the next section, binary trees are easy to work with, and any general tree can be represented as a binary tree.

TREES AND HUMAN INTELLIGENCE

In the early days of artificial intelligence (AI), getting a computer to play chess at the level of a human master was considered one of the ultimate challenges. The complex strategies of the game were thought to be impossible to program into a computer. In 1968, the international grandmaster David Levy bet $10,000 of his own money that no program would beat him in the next 10 years. He won the bet, but in 1989 he was defeated by a program called Deep Thought. In 1996, a program called Deep Blue beat Garry Kasparov, the reigning world champion, in a game. The following year, in a full six-game match, Deep Blue beat Kasparov 3.5 to 2.5, signaling that computers could play chess at the grandmaster level.

Although chess software is a challenging technical feat, most computer scientists and AI researchers agree that it has contributed little to our understanding of human intelligence. A software program's approach to representing knowledge seems to be quite different from how a human's brain works.

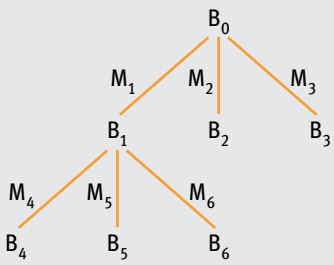
Virtually all chess programs use a **game tree** to represent a game board and to select a next move. In a game tree, nodes represent board positions and edges represent moves. For example, if we are at board position B_0 and feel that our opponent has three possible moves—M_1, M_2, M_3, which would result in board positions B_1, B_2, and B_3, respectively—this knowledge would be represented by the following game tree:



An actual game tree, however, would require many levels, or *plys*, to do a complete and thorough analysis. Using the preceding diagram, we must consider what moves we might make in response to our opponent's moves. For example, if our opponent made move

continued

M_1, we might have three potential responses—M_4, M_5, and M_6, leading to the following game tree:



As you can imagine, a game tree constructed from a given board position can become quite large. If we looked ahead 10 moves for each player (20 plys), and there were 10 possible moves to consider at each level, our game tree would have 10^{20} leaf nodes to evaluate before we made the next move. The fastest chess program in the world (called Hydra) can evaluate 2×10^8 positions per second, so examining 10^{20} alternatives would take about 16,000 years! Therefore, most AI chess research is directed at designing algorithms to eliminate large sections of the tree so that a search can be completed in a reasonable time. However, even after eliminating most nodes from consideration, the tree search used by a computer is still based on the concepts of brute force and great speed—evaluating a massive number of positions in relatively simple ways. Grandmasters like Garry Kasparov use a different technique—they analyze far fewer moves but in a much more focused and detailed manner.

Essentially, AI researchers have learned that computer software and humans represent knowledge in a chess game in very different ways. This is not surprising, however, given that a Von Neumann machine and a human brain are based on quite different architectures.

7.3 Binary Trees

7.3.1 Introduction

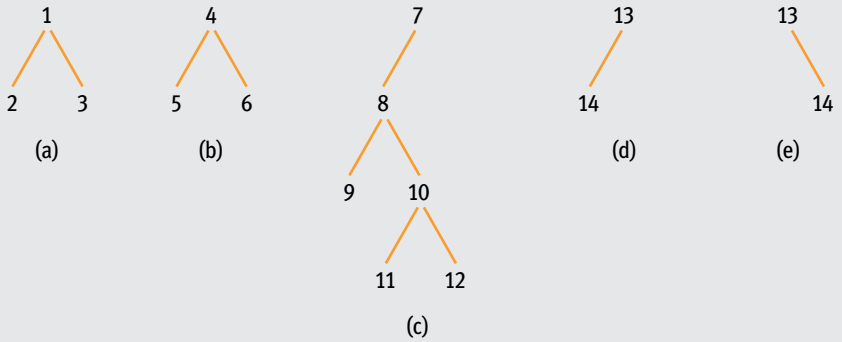
Binary trees are distinct from general trees in three ways:

- Binary trees can be empty.
- All nodes in a binary tree have a degree less than or equal to two.

- Binary trees are **ordered trees** in which every node is explicitly identified as being either the left child or the right child of its parent.

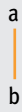
In other words, a **binary tree** is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left and right subtrees.

Some examples of binary trees are shown in Figure 7-6.

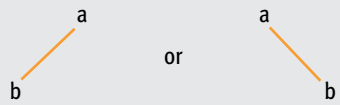


[FIGURE 7-6] Examples of binary trees

In Figure 7-6, the binary trees (a) and (b) are structurally identical, and differ only in the data values contained in their nodes. However, binary trees (d) and (e) are not structurally identical. The root of the binary tree in (d) has an empty right subtree, whereas the one in (e) has an empty left subtree. This is an important point because binary trees are ordered. A tree is identified not only by the data it contains but by the *position* of its nodes. Therefore, when drawing a binary tree, be careful to indicate clearly whether a node is the right or left subtree of its parent. Diagrams such as the following:



are ambiguous and can lead to errors because it is unclear whether we meant to write:



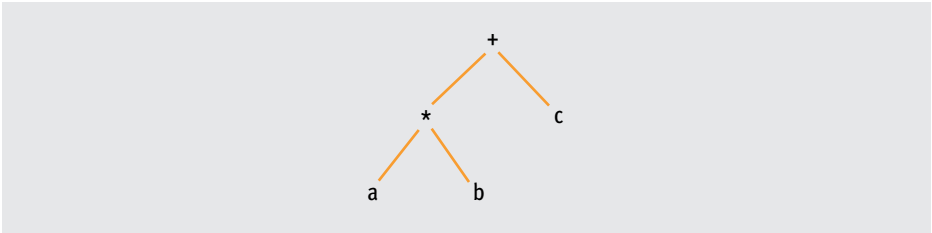
Finally, observe that a binary tree is actually not a tree, technically speaking. Reread the definitions of *tree* in Section 7.2 and *binary tree* in Section 7.3 and notice that a general tree must have at least one node, whereas a binary tree can be empty.

7.3.2 Operations on Binary Trees

When designing a binary tree class, we first must decide on the methods we want to include. Like the list structure of Chapter 6, the binary tree is a general structure with a large number of possible methods.

One of the most common operations on binary trees is **tree traversal**. This operation starts at the root and visits every node in the tree exactly once. By *visit*, we mean to perform some processing operation on the data value in that node. The exact nature of this operation depends on the application; for example, it could mean printing the contents of each node or adding the node values in the entire tree. Traversal of a tree is equivalent to iterating through a linear structure from the first element to the last. Tree traversal forms the basis for many other tree operations. For example, to print the contents of a tree, we would traverse it, printing the contents of each node as it is visited.

There are a number of ways to traverse a tree. One method starts at the root and visits it. Then we have a choice: We can traverse the nodes in either the left subtree or the right subtree. If we choose the left subtree, we traverse it in exactly the same way as the original tree. That is, we visit the root of the subtree and then traverse all the nodes in its left subtree, followed by all the nodes in its right subtree. This approach is called a **preorder traversal**. Let's see how this algorithm behaves using the expression tree in Figure 7-7.



[FIGURE 7-7] Sample tree used for tree traversal

The three-step algorithm for a preorder tree traversal is as follows:

1. Visit the root: +.
2. Traverse the left subtree: (* *a b*).
3. Traverse the right subtree: (*c*).

Steps 2 and 3 each involve traversing a subtree, which you can accomplish by recursively reapplying the same three steps:

1. Visit the root: +.
2. Traverse the left subtree: (* *a b*).
   - 2.1 Visit the root: *.
   - 2.2 Traverse the left subtree: (*a*).
   - 2.3 Traverse the right subtree: (*b*).
3. Traverse the right subtree: (*c*).
   - 3.1 Visit the root: *c*.
   - 3.2 Traverse the left subtree: (empty).
   - 3.3 Traverse the right subtree: (empty).

When a subtree is empty, as in Steps 3.2 and 3.3, we have reached the base case of our recursive algorithm, and we are finished. Therefore, the complete preorder traversal of the tree in Figure 7-7 is:

1. Visit the root: +.
2. Traverse the left subtree: (* *a b*).
   - 2.1 Visit the root: *.
   - 2.2 Traverse the left subtree: (*a*).
     - 2.2.1 Visit the root: a.
     - 2.2.2 Traverse the left subtree: (empty).
     - 2.2.3 Traverse the right subtree: (empty).
   - 2.3 Traverse the right subtree: (*b*).
     - 2.3.1 Visit the root: *b*.
     - 2.3.2 Traverse the left subtree: (empty).
     - 2.3.3 Traverse the right subtree: (empty).
3. Traverse the right subtree: (*c*).
   - 3.1 Visit the root: *c*.
   - 3.2 Traverse the left subtree: (empty).
   - 3.3 Traverse the right subtree: (empty).

This traversal visits the nodes in the order +**abc*.

If R is a pointer to the root of a binary tree, the preorder traversal algorithm of the tree referenced by R is shown in Figure 7-8.

```
// The preorder traversal algorithm of a
// binary tree whose root is referenced by R

preorderTraversal(R) {
  if (R refers to an empty tree)
    // Base case
    return;
  else {
    // Recursive case
    visit the node referenced by R;
    preorderTraversal(left subtree of R)
    preorderTraversal(right subtree of R)
  }
}
```

[FIGURE 7-8] Preorder tree traversal algorithm

The list structures of Chapter 6 provided only one way to iterate through a list—from the first node to its unique successor, then to that node's unique successor, and so on. However, hierarchical structures such as the binary tree in Figure 7-7 offer multiple ways to traverse the nodes. A preorder traversal scheme walks through the tree by visiting the root node before visiting the left and right subtrees. We can produce other traversal methods by simply permuting the order of these operations. Two permutations of interest are the **postorder traversal**:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

and the **inorder traversal**:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

There is a natural correspondence between the preorder, postorder, and inorder traversals of an expression tree and the prefix, postfix, and infix representations of an arithmetic expression. In a **prefix representation** of an expression, a binary operator is written *before*

its two operands. For example, $a + b$ would be represented in prefix notation as $+ab$. In a **postfix representation** of an expression, the operator is written *after* its operands. So, $a + b$ would be written $ab+$. Finally, in **infix representation** (the one we are most used to), a binary operator is written *between* its two operands. So the expression $a + b$ is in infix notation.

Using the expression tree in Figure 7-7, and assuming that *visit the root* means to output the data field of the root, the three tree traversal methods just described produce the following:

*Pre*fix representation of $a * b + c$:	$+*abc$	Operators precede operands
*Pre*order traversal of the tree:	$+*abc$	
*Post*fix representation of $a * b + c$:	$ab*c+$	Operators follow operands
*Post*order traversal of the tree:	$ab*c+$	
*In*fix representation of $a * b + c$:	$a*b+c$	Operators between operands
*In*order traversal of the tree:	$a*b+c$	

The recursive algorithms for postorder and inorder traversals of a binary tree are shown in Figure 7-9.

```
// The postorder traversal algorithm of a
// binary tree whose root is referenced by R

postorderTraversal(R)   {
  if (R refers to an empty tree)
    // Base case
    return;
  else {
    // Recursive case
    postorderTraversal(left subtree of R);
    postorderTraversal(right subtree of R);
    visit the node referenced by R;
  }
}


// The inorder traversal algorithm for a
// binary tree whose root is referenced by R
inorderTraversal(R)   {
  if (R refers to an empty tree)
    // Base case
    return;
```

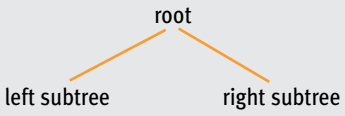
continued

```
  else {
    // Recursive case
    inorderTraversal(left subtree of R);
    visit the node referenced by R;
    inorderTraversal(right subtree of R);
  }
}
```

[FIGURE 7-9] Postorder and inorder binary tree traversal algorithms

An important question is: What is the complexity of the traversal algorithms of Figures 7-8 and 7-9? Because these algorithms are recursive, we will answer this question using the recurrence relation technique for analyzing recursive algorithms, which was presented in Section 5.5.

Assume our binary tree contains n nodes and looks like this:



Furthermore, assume that the binary tree and all its subtrees are roughly in balance, which means that every left and right subtree is approximately the same size. To traverse the tree we first visit the root, which takes one step, and then visit both the left and right subtrees, each of size $n/2$, because we assumed that the tree is balanced. If we use the notation $T(n)$ to mean the time required to traverse a tree of size n, then the recurrence relations describing the time complexity of our traversal algorithms are given by:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= \underset{\text{root}}{1} + \underset{\text{left subtree}}{T(n/2)} + \underset{\text{right subtree}}{T(n/2)} \\
&= 1 + 2\,T(n/2)
\end{aligned}
$$

To solve this recurrence relation, we use the method of repeated substitution:

$$T(n) = 1 + 2\,T(n/2)$$

Solving for $T(n/2)$:

$$
\begin{aligned}
T(n/2) &= 1 + 2\,T(n/4), \text{ which we now plug into the preceding formula} \\
T(n) &= 1 + 2\,(1 + 2\,T(n/4)) \\
&= 3 + 4\,T(n/4)
\end{aligned}
$$

Repeat this substitution process, yielding the following sequence of equations:

$$\begin{aligned} T(n) &= 7 + 8\,T(n/8) \\ &= 15 + 16\,T(n/16) \\ &= 31 + 32\,T(n/32) \\ &= \ldots \end{aligned}$$

The general formula for all the terms in the series is:

$$T(n) = (2^k - 1) + 2^k\,T(n/2^k) \qquad k = 1, 2, 3, \ldots$$

We now let $n = 2^k$:

$$T(n) = (n - 1) + n\,T(1)$$

The original problem statement said that $T(1) = 1$. This yields the following:

$$\begin{aligned} T(n) &= (n - 1) + n = 2\,n - 1 \\ &= O(n) \end{aligned}$$

We have shown that the traversal of a binary tree is a linear-time, $O(n)$ operation. Intuitively, this is what we would expect, because the traversal methods of Figures 7-8 and 7-9 visit every one of the n nodes in the tree exactly once.

Even if the binary tree were unbalanced, we would still get the same result. See Exercise 5b at the end of the chapter.

In addition to traversal, many other important operations could be part of a binary tree interface. However, as you saw with the list structure in Chapter 6, the enormous flexibility of this hierarchical data structure means there is no universal agreement on exactly which methods should be included. In general, we need more information about how this structure will be used to determine which methods are best to include or omit. This section presents some typical operations on binary trees. We then suggest additional possibilities in Exercise 9 at the end of the chapter.

To describe the behavior of our methods, we use an idea first presented in Chapter 6: a **current position indicator**, also called a **cursor** or an **iterator**. This is simply an instance variable that references, or points to, one of the nodes, called the **current node**, in the binary tree. Most of the following methods operate on this current node.

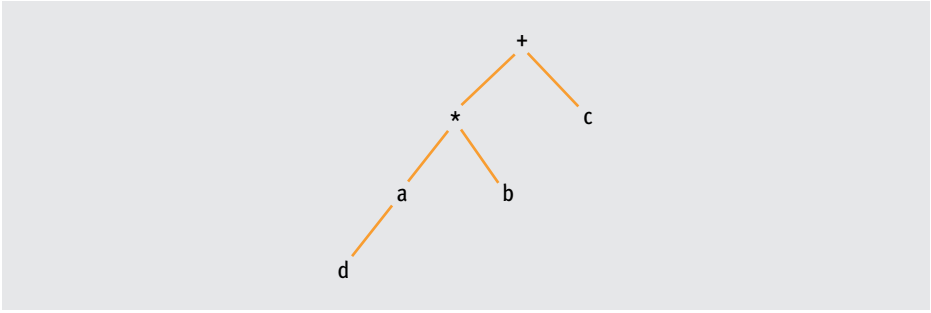
Our binary tree interface includes the following positioning, checking, retrieval, mutator, and traversal methods:

POSITIONING:	`toRoot()`	Reposition the cursor to the root of the entire tree
	`toParent()`	Reposition the cursor to the parent of the current node or to **null** if the current node has no parent
	`toLeftChild()`	Reposition the cursor to the left child of the current node or to **null** if the current node has no left child
	`toRightChild()`	Reposition the cursor to the right child of the current node or to **null** if the current node has no right child
	`find(T o)`	Reposition the cursor to the first node containing `o` in the information field when doing an inorder traversal
CHECKING:	`hasParent()`	True if current node has a parent; false otherwise
	`hasLeftChild()`	True if current node has a left child; false otherwise
	`hasRightChild()`	True if current node has a right child; false otherwise
	`isOffList()`	True if cursor points to a node in the tree; false otherwise
	`equals(Object o)`	True if the given object and **this** tree are identical, false otherwise
	`size()`	Return the total number of nodes in the tree
	`height()`	Return the height of the tree
RETRIEVAL:	`get()`	Return the information field of the current node

continued

MUTATOR:	`set(T o)`	Reset the information field of current node to the value o
	`insertRight(T o)`	Add `o` as the right child of the current node if the right child is **null**; otherwise, do nothing
	`insertLeft(T o)`	Add `o` as the left child of the current node if the right child is **null**; otherwise, do nothing
	`prune()`	Delete the entire subtree rooted at the current node
TRAVERSAL:	`preorder()`	Do a preorder traversal of the tree
	`postorder()`	Do a postorder traversal of the tree
	`inorder()`	Do an inorder traversal of the tree

Three important mutator methods on binary trees are `insertLeft()`, `insertRight()`, and `prune()`. The methods `insertLeft()` and `insertRight()` create a node that contains the object `o` in the information field and then insert the new node into the tree as either the left child or right child of the current node. The cursor is reset to point to the newly added node. A necessary precondition for both methods is that the new node is inserted in an unoccupied position. For example, given the binary tree in Figure 7-7, and assuming that the cursor references the node containing `a`, the call `insertLeft("d")` would insert a new node containing the value `"d"` as the left child of node `a`, producing the following tree:



These two methods are the primary means of constructing binary trees. You start with an empty tree and insert new nodes, one at a time, into their proper location.

The `prune()` method is a mutator that removes nodes from a binary tree. However, we cannot simply delete an individual node *n* unless it is a leaf. Instead, we must delete the entire subtree rooted at *n*. For example, given the six-node tree shown previously, deleting the node labeled * also requires us to delete the entire subtree rooted at that node, namely

the four nodes {*, *a*, *b*, *d*}. It is possible to describe a method that deletes a single node, but it involves reconstruction of the tree as well. In our interface, `prune()` deletes the entire subtree rooted at the node referenced by the cursor. The cursor is then reset to point to the parent of the root of the deleted subtree.

The `equals()` method determines if two binary trees are identical. Trees are identical if their shape is identical and if the data value contained in the information field of every node is the same. The operator returns a Boolean value that is true if the trees are identical and false otherwise. This method is called a **deep compare**, and was introduced in Section 3.3.5.

Two methods that help characterize the size and shape of a binary tree are `size()` and `height()`, which return the number of nodes in the tree and the height of the tree, respectively. Remember from Section 7.2 that the height of a tree is defined as the maximum level number of any individual node. For example, given the tree shown in Figure 7-7, `size()` returns a 5, as there are five nodes, and `height()` returns a 3.

Finally, our interface includes the three traversal operations—`preorder()`, `inorder()`, and `postorder()`—described at the beginning of this section. However, we have chosen to implement them in an interesting way. If you remember our earlier discussion, a traversal visits every node, where *visit* means to perform an unspecified operation on the data field. Now that we are implementing these traversals, not simply explaining them, how do we specify the operation that the visit method should carry out? We could simply assume that the visit performs a specific operation, such as printing the data value stored in a node. However, this approach is not very flexible and does not permit the user to do something else.

We have designed these methods to use a **callback** to specify what should be done when a node is visited. A callback is the code we want to execute when a node is visited during a traversal. In an object-oriented language such as Java, we use an object to specify the code that should be executed. At the point when a node needs to be visited, the traversal method invokes the `visit()` method of the callback object, and passes the data contained in the node being visited to the method. Instead of defining `visit()` as an instance method of our binary tree class (and not knowing exactly what it should do), we invoke `cb.visit(data)`, which invokes the user-provided `visit()` method of the object named as a parameter. Because `visit()` is free to do anything it wants, the three traversal operations are completely general. The `Callback` interface, shown in Figure 7-10a, specifies that a callback object must provide a public instance method called `visit()`. The `visit()` method represents the callback.

```
/**
 * A callback object specifies what to do when a traversal
 * "visits" a node.
 */
public interface Callback<T> {
```

continued

```
    /**
     * Called when a node is visited during a traversal.
     *
     * @param T the data contained in the node being visited.
     */
    public void visit( T data );

} // Callback
```

[FIGURE 7-10a] The `Callback` interface

Figure 7-10b is a `BinaryTree` interface specification that includes the 20 methods just described. This interface should not be viewed as definitive or complete. On the contrary, many other methods would be good candidates for inclusion in this interface. We would need more information about how these classes will be used to know whether other methods should be added to our interface.

```
/**
 * An interface for the binary tree abstract data type (ADT)
 */
public interface BinaryTree<T> {
    /**
     * Position the cursor at the root of the tree.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   If the tree is empty, the cursor is invalid.
     *     Otherwise, the cursor refers to the root of the tree.
     *   The tree structure is unchanged.
     */
    public void toRoot();

    /**
     * Determine if the current node has a left child.
     *
     * Preconditions:
     *   The cursor is valid.
     *
     * Postconditions:
     *   The tree structure is unchanged.
     *   The cursor is unchanged.
     *
```

continued

```
     * @return true if the current node has a left child.
     */
    public boolean hasLeftChild();

    /**
     * Determine if the current node has a right child.
     *
     * Preconditions:
     *   The cursor is valid.
     *
     * Postconditions:
     *   The tree structure is unchanged.
     *   The cursor is unchanged.
     *
     * @return true if the current node has a right child.
     */
    public boolean hasRightChild();

    /**
     * Determine if the current node has a parent.
     *
     * Preconditions:
     *   The cursor is valid.
     *
     * Postconditions:
     *   The tree structure is unchanged.
     *   The cursor is unchanged.
     *
     * @return true if the current node has a parent.
     */
    public boolean hasParent();

    /**
     * Determine if the cursor is on the tree.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The tree structure is unchanged.
     *   The cursor is unchanged.
     *
     * @return true if the cursor is on the tree.
     */
    public boolean isValid();
```

continued

```
/**
 * Position the cursor at the current node's parent, if any.
 *
 * Preconditions:
 *   The cursor is valid.
 *
 * Postconditions:
 *   If the cursor has no parent (i.e., it is root node),
 *     the cursor is invalid. Otherwise, the cursor is
 *     changed to refer to the parent of the current node.
 *   The structure of the tree is unchanged.
 */
public void toParent();

/**
 * Position the cursor at the left child of the current node.
 *
 * Preconditions:
 *   The cursor is valid.
 *
 * Postconditions:
 *   If the left child of the current node is invalid, the
 *     cursor is invalid. Otherwise, the cursor is changed
 *     to refer to the left child of the current node.
 *   The structure of the tree is unchanged.
 */
public void toLeftChild();

/**
 * Position the cursor at the right child of the current node.
 *
 * Preconditions:
 *   The cursor is valid.
 *
 * Postconditions:
 *   If the right child of the current node is invalid, the
 *     cursor is invalid. Otherwise, the cursor is changed
 *     to refer to the right child of the current node.
 *   The structure of the tree is unchanged.
 */
public void toRightChild();

/**
 * Insert an item in the left child of the current node.
 * If the cursor is null and the tree has no root, a new root
 * is created containing this data.
 *
```

continued

```
     * Preconditions:
     *   The tree is empty, or the cursor is valid and the left
     *     child is not empty.
     *
     * Postconditions:
     *   The cursor has not changed.
     *   The size of the tree has increased by one.
     *   No other structure of the tree has changed.
     *
     * @param data the data to put in the left child.
     */
    public void insertLeft( T data );

    /**
     * Insert an item in the right child of the current node.
     * If the cursor is null and the tree has no root, a new root
     * is created containing this data.
     *
     * Preconditions:
     *   The tree is empty, or the cursor is valid and the right
     *     child is not empty.
     *
     * Postconditions:
     *   The cursor has not changed.
     *   The size of the tree has increased by one.
     *   No other structure of the tree has changed.
     *
     * @param data the data to put in the right child.
     */
    public void insertRight( T data );

    /**
     * Return a reference to the data stored in the current node.
     *
     * Preconditions:
     *   The cursor is on the tree.
     *
     * Postconditions:
     *   The tree is unchanged.
     *
     * @return the data stored in the current node.
     */
    public T get();

    /**
     * Set the data stored at the current node.
     *
```

continued

```
     * Preconditions:
     *   The cursor is on the tree.
     *
     * Postconditions:
     *   The reference of the current node is changed.
     *   The rest of the tree is unchanged.
     *
     * @param data the data to store in the current node.
     */
    public void set( T data );

    /**
     * Removes the subtree rooted at (including) the cursor.
     *
     * Preconditions:
     *   The cursor is on the tree.
     *
     * Postconditions:
     *   The specified subtree has been removed. If the cursor
     *     referred to the root node, the tree is empty.
     *   The tree's size has decreased.
     *   No other structure of the tree has changed.
     *   If the resulting tree is empty, the cursor is invalid.
     *     Otherwise, the cursor refers to the parent of the
     *     current node.
     */
    public void prune();

    /**
     * Determine if the given object is identical to this one.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The cursors of both trees refer to the root of their
     *     respective trees.
     *
     * @param o the object to compare to.
     *
     * @return true if and only if all of the following are true:
     *               The other object is a BinaryTree
     *               The structure of the two trees is identical
     *               The data contained in corresponding nodes of
     *                 the two trees is identical.
     */
    public boolean equals( Object o );
```

continued

```
/**
 * Return the number of nodes in this tree.
 *
 * Preconditions:
 *      None
 *
 * Postconditions:
 *      The tree is unchanged.
 *
 * @return the number of nodes in the tree.
 */
public int size();

/**
 * Return the height of the tree.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *   The tree is unchanged.
 *
 * @return the height of the tree.
 */
 public int height();

/**
 * Position the cursor on the first (as seen by an inorder
 * traversal) occurrence of the given data.
 *
 * Preconditions:
 *   Key is not null.
 *
 * Postconditions:
 *   The tree is unchanged.
 *   If key is found, the cursor refers to the first
 *     occurrence of the key. If key is not found, the
 *     cursor is off the tree.
 *
 * @param target the data to be searched for.
 */
public void find( T target );

/**
 * Perform a preorder traversal.
 *
```

continued

```
     * Preconditions:
     *   cb is not null
     *
     * Postconditions:
     *   The tree is unchanged.
     *
     * @param cb the callback object used to visit each node.
     */
    public void preOrder( Callback<T> cb );

    /**
     * Perform an inorder traversal.
     *
     * Preconditions:
     *   cb is not null
     *
     * Postconditions:
     *   The tree is unchanged.
     *
     * @param cb the callback object used to visit each node.
     */
    public void inOrder( Callback<T> cb );

    /**
     * Perform a postorder traversal.
     *
     * Preconditions:
     *   cb is not null
     *
     * Postconditions:
     *   The tree is unchanged.
     *
     * @param cb the callback object used to visit each node.
     */
    public void postOrder( Callback<T> cb );

} // BinaryTree
```

[FIGURE 7-10b] Interface specifications for a binary tree

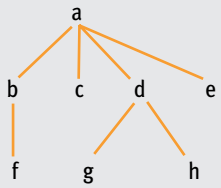
7.3.3 The Binary Tree Representation of General Trees

We have covered the topic of binary trees in depth because of an important property of hierarchical data structures—every general tree can be rewritten as a binary tree without any loss of information. Therefore, instead of working with the cumbersome general tree of Section 7.2, we can convert it into a binary tree and work with a much simpler structure.

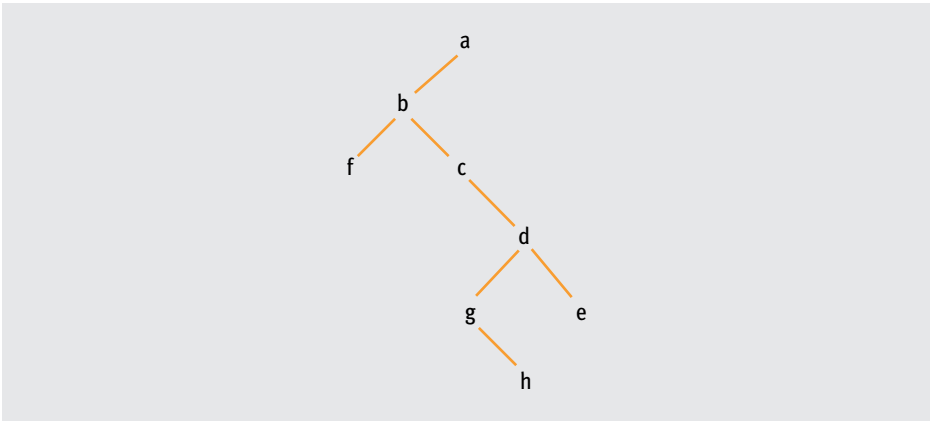
Recall that every node in a binary tree has at most two links—one that points to the left subtree and one that points to the right subtree. We can use these two links to convert any general tree into a binary tree using the **oldest child/next sibling algorithm**. The algorithm uses the two links of a binary tree in the following way:

- The left pointer of the binary tree node points to the "oldest" (far-left) child of that node in the general tree, or **null** if the node has no children.
- The right pointer of the binary tree node points to the "next" sibling (the one immediately to the right) of a node in the general tree, or **null** if the node has no siblings to its right.

For example, consider the following eight-node general tree:



We can convert this into a binary tree in the following manner. In the general tree, the oldest (far-left) child of node *a* is node *b*. So, we use the left pointer of *a* in the new binary tree to point to *b*. Because *a* has no siblings, its right pointer in the binary tree is set to **null**. The right pointers of *b*, *c*, and *d* in the binary tree point to their next (far-right) sibling in the general tree, *c*, *d*, and *e*, respectively, while *e*, which has no right sibling, has its right pointer set to **null**. The left pointer of *b* points to *f*, its oldest (far-left) child. The left pointers of *c* and *e* are **null** because they have no children. The left pointer of *d* points to *g*, its oldest child. The right and left pointers of node *f* are **null** because it has neither children nor siblings. The right pointer of *g* points to its next sibling *h*, while its left pointer is **null**. Finally, the right and left pointers of *h* are **null** because it has no siblings to its right and no children. The final binary tree representation of the preceding eight-node general tree is shown in Figure 7-11.



[FIGURE 7-11] The binary tree representation of a general tree

This conversion algorithm allows us to represent a general tree of arbitrary degree as a binary tree without any loss of information. All relationships in the original general tree are still present in the new binary tree. To identify the children of a given node in the general tree, we follow the left pointer of the node in the binary tree to its first child and then follow the right pointer of nodes until we reach a **null**. These are the children of the original node. Similar algorithms exist for determining all other important structural relationships.

7.3.4 The Reference-Based Implementation of Binary Trees

To construct a class that implements the binary tree interface of Figure 7-10b, we first must select an internal representation for the tree's nodes. As with linear data structures in Chapter 6, we can implement the nodes using either array-based or reference-based techniques. This section describes a reference-based method; the following section describes how to implement a binary tree using an array. Figure 7-12 shows the declarations for a binary tree node containing two reference fields that point to the left and right subtrees of the node.

```
// This class defines the structure of a node in a binary tree

public class BinaryTreeNode<T> {
   private T data;                    // the information field
   private BinaryTreeNode<T> right;   // link to the right subtree
   private BinaryTreeNode<T> left;    // link to the left subtree
```

continued

```
    // The public methods go here
}
```

[FIGURE 7-12] `BinaryTreeNode` class

Although the declarations in Figure 7-12 are a common way to implement the nodes of a binary tree, they are not the only way. For example, using the declarations in Figure 7-12, it is easy to locate the children of a node—just follow the `left` and `right` pointers. However, moving *up* the tree is more difficult because there is no pointer from a child to its parent. The only way to locate the parent of a node is to traverse the entire tree beginning from the root. If moving upward from the leaves toward the root is a common occurrence, then we may want to consider adding a `parent` field to each node. The addition of a `parent` pointer is similar to our inclusion of a `previous` pointer in the doubly linked list of Section 6.2.3.3, which allowed us to identify both the predecessor and successor of a node. However, as we saw with the doubly linked list, this increase in functionality did not come without cost. Adding a `parent` pointer caused a 50% increase in the number of reference fields in each node, from two to three. For large trees this increased memory could be significant.

We have now specified the behavior of our binary tree class (the 20 operations in the interface of Figure 7-10b), and we have selected a representation for the individual nodes. The only remaining task is to decide exactly what state information we want to maintain for a binary tree object. We certainly need a reference to the root of the tree, and we need the current position indicator, or cursor. No other state variables are required, because once we identify the root of the tree, we can locate all other nodes. However, additional information might be useful to include, such as the height of the tree or the total number of nodes. Keeping this state information simplifies the implementation of the instance methods `height()` and `size()`, as they only need to return these values. Of course, every time we add or remove a node or a subtree, we need to recompute and store the height and size of the new tree.

A complete reference-based implementation of the binary tree interface of Figure 7-10b is shown in Figure 7-13. It includes the class `BinaryTreeNode`, which provides the state and behaviors for individual nodes in the tree.

```
/**
 * A node for use in binary trees.
 */
public class BinaryTreeNode<T> {
    private BinaryTreeNode<T> left;   // The left child
    private BinaryTreeNode<T> right;  // The right child
    private T data;                   // The data in this node
```

continued

```
/**
 * Create a new node.
 */
public BinaryTreeNode() {
  this( null, null, null );
}

/**
 * Create a new node containing the specified data.
 *
 * @param theData the data to place in this node.
 */
public BinaryTreeNode( T theData ) {
  this( theData, null, null );
}

/**
 * Create a new node with the specified data and children.
 *
 * @param theData the data to place in this node.
 * @param leftChild the left child.
 * @param rightChild the right child.
 */
public BinaryTreeNode( T theData,
                       BinaryTreeNode<T> leftChild,
                       BinaryTreeNode<T> rightChild ) {
      data = theData;
      left = leftChild;
      right = rightChild;
}

/**
 * Return the data stored in this node.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *       This node is unchanged.
 *
 * @return the data stored in this node.
 */
public T getData() {
    return data;
}
```

continued

```
/**
 * Return a reference to the left child.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *       This node is unchanged.
 *
 * @return a reference to this node's left child.
 */
public BinaryTreeNode<T> getLeft() {
      return left;
}

/**
 * Return a reference to the right child.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *       This node is unchanged.
 *
 * @return a reference to this node's right child.
 */
public BinaryTreeNode<T> getRight() {
    return right;
}

/**
 * Set this node's left child to the given node.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *       This node's left subchild has been set to the given node.
 *
 * @param newLeft the node to become this node's left child.
 */
public void setLeft( BinaryTreeNode<T> newLeft ) {
    left = newLeft;
}
```

continued

```
/**
 * Set this node's right child to the given node.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *       This node's right subchild has been set to the given node.
 *
 * @param newRight the node to become this node's right child.
 */
public void setRight( BinaryTreeNode<T> newRight ) {
    right = newRight;
}

/**
 * Set the data field of the current node.
 *
 * Preconditions:
 *   None
 *
 * Postconditions:
 *       This node's data field has been updated.
 *
 * @param newData Node to become this node's right child.
 */
public void setData( T newData ) {
    data = newData;
}

/**
 * Perform an inorder traversal of the tree rooted at this node.
 * Each visited node is visited using the given callback.
 *
 * Preconditions:
 *      cb is not null
 *
 * Postconditions:
 *       All nodes in the tree have been visited by the given
 *         callback in the specified order.
 *
 * @param cb the callback object used to process nodes.
 */
public void inOrder( Callback<T> cb ) {
    // First we visit the left subtree, if any
    if ( left != null ) {
        left.inOrder( cb );
    }
```

continued

```
        // Then we visit this node
        cb.visit(data);

        // Last we visit the right subtree, if any
        if ( right != null ) {
            right.inOrder( cb );
        }
    }

    /**
     * Perform a preorder traversal of the tree rooted at this node.
     * Each visited node is visited using the given callback.
     *
     * Preconditions:
     *       cb is not null
     *
     * Postconditions:
     *       All nodes in the tree have been visited by the given
     *         callback in the specified order.
     *
     * @param cb the callback object used to process nodes.
     */
    public void preOrder( Callback<T> cb ) {
        // First we visit the current node
        cb.visit( data );

        // Then we visit our left subtree, if any
        if ( left != null )  {
            left.preOrder( cb );
        }

        // Last we visit our right subtree, if any
        if ( right != null ) {
            right.preOrder( cb );
        }
    }

    /**
     * Perform a postorder traversal of the tree rooted at this node.
     * Each visited node is visited using the given callback.
     *
     * Preconditions:
     *       cb is not null
     *
```

continued

```
     * Postconditions:
     *       All nodes in the tree have been visited by the given
     *         callback in the specified order.
     *
     * @param cb the callback object used to process nodes.
     */
    public void postOrder( Callback<T> cb ) {
        // First we visit our left subtree, if any
        if ( left != null ) {
            left.postOrder( cb );
        }

        // Then we visit our right subtree, if any
        if ( right != null ) {
            right.postOrder( cb );
        }

        // Last we visit this node
        cb.visit( data );
    }

    /**
     * Return the height of this tree.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The tree rooted at this node is unchanged.
     *
     * @return the height of this tree.
     */
    public int height() {
        int leftHeight = 0;   // Height of the left subtree
        int rightHeight = 0;  // Height of the right subtree
        int height = 0;       // The height of this subtree

        // If we have a left subtree, determine its height
        if ( left != null )  {
            leftHeight = left.height();
        }

        // If we have a right subtree, determine its height
        if ( right != null )  {
            rightHeight = right.height();
        }
```

continued

```
        // The height of the tree rooted at this node is one more
        // than the height of the 'taller' of its children.
        if (leftHeight > rightHeight) {
            height = 1 + leftHeight;
        }
        else {
            height = 1 + rightHeight;
        }

        // Return the answer
        return height;
    }

    /**
     * Return the number of nodes in this tree.
     *
     * Preconditions:
     *     None
     *
     * Postconditions:
     *     The tree rooted at this node is unchanged.
     *
     * @return the number of nodes in this tree.
     */
    public int size() {
        int size = 0;  // The size of the tree

        // The size of the tree rooted at this node is one more than the
        // sum of the sizes of its children.
        if ( left != null ) {
            size = size + left.size();
        }

        if ( right != null ) {
            size = size + right.size();
        }

        return size + 1;
    }

    /**
     * Position the cursor on the first (as seen by an inorder
     * traversal) occurrence of the given data.  Equality is
     * determined by invoking the equals method.
     *
     * Preconditions:
     *     Key is not null
     *
```

continued

```
     * Postconditions:
     *     The tree is unchanged.
     *     If key is found, cursor refers to the first occurrence of
     *       the key.  If key is not found, the cursor is off the tree
     *
     * @param target object containing the data to be searched for.
     */
    public BinaryTreeNode<T> find( T target ) {
        BinaryTreeNode<T> location = null;  // The target node

        // Is the target in the left subtree?
        if ( left != null ) {
            location = left.find( target );
        }

        // If we haven't found it, is it in this node?
        if (location == null && target.equals( data ) ) {
            location = this;
        }

        // If we haven't found it – check the right child
        if ( location == null && right != null ){
            location = right.find( target );
        }

        // Return the location of the target
        return location;
    }

    /**
     * Locate the parent of the given node looking in the tree whose
     * root is this node.
     *
     * Preconditions:
     *    The target is not null
     *
     * Postconditions:
     *    The tree rooted at this node is unchanged.
     *
     * @param target the node whose parent is to be located.
     *
     * @return the parent of the given node.
     */
    public BinaryTreeNode<T> findParent( BinaryTreeNode<T> target ) {
        BinaryTreeNode<T> parent = null;
```

continued

```
        // Are we parent to the target node?
        if ( left == target || right == target ) {
            parent = this;
        }

        // If we have not found the parent, check the left subtree
        if ( parent == null && left != null ) {
            parent = left.findParent( target );
        }

        // If we have not found the parent, check the right subtree
        if ( parent == null && right != null ) {
            parent = right.findParent( target );
        }

        // Return the parent
        return parent;
    }

    /**
     * Return a string representation of this node.
     *
     * @return a string representation of this node.
     */
    public String toString() {
        return data.toString();
    }

} // BinaryTreeNode

/**
 * A reference-based implementation of a binary tree. Javadoc comments
 * for methods specified in the BinaryTree interface have been omitted.
 *
 * This code assumes that the preconditions stated in the comments are
 * true when a method is invoked and therefore does not check the
 * preconditions.
 */
public class LinkedBinaryTree<T> implements BinaryTree<T> {
    private BinaryTreeNode<T> root;    // The root of the tree
    private BinaryTreeNode<T> cursor;  // The current node

    public void toRoot() {
        cursor = root;
    }
```

continued

```
public boolean hasLeftChild() {
    return cursor.getLeft() != null;
}

public boolean hasRightChild() {
    return cursor.getRight() != null;
}

public boolean hasParent() {
    return root.findParent( cursor ) != null;
}

public boolean isValid() {
    return cursor != null;
}

public void toParent() {
    cursor = root.findParent( cursor );
}

public void toLeftChild() {
    cursor = cursor.getLeft();
}

public void toRightChild() {
    cursor = cursor.getRight();
}

public void insertLeft( T data ) {
    // Create the node that will hold the data
    BinaryTreeNode<T> newNode = new BinaryTreeNode<T>( data );

    // If the tree is empty, this is the only node in the
    // tree; otherwise, the new node is the left child of the cursor
    if ( root == null ) {
        root = newNode;
    }
    else {
        cursor.setLeft( newNode );
    }
}

public void insertRight( T data ) {
    // Create the node that will hold the data
    BinaryTreeNode<T> newNode = new BinaryTreeNode<T>( data );
```

continued

```
        // If the tree is empty, this is the only node in the tree;
        // otherwise, the new node is the right child of the cursor
        if ( root == null ) {
            root = newNode;
        }
        else {
            cursor.setRight( newNode );
        }
    }

    public T get() {
      return cursor.getData();
    }

    public void set( T data ) {
      cursor.setData( data );
    }

    public void prune() {
        // Are we trying to delete the root node?
        if ( cursor == root ) {
            // Delete the root and invalidate the cursor
            root = null;
            cursor = null;
        }
        else {
            // Find the parent of the node to delete
            BinaryTreeNode<T> parent = root.findParent( cursor );

            // Is it the parent's left child?
            if ( parent.getLeft() == cursor ) {
                // Delete left child
                parent.setLeft( null );
            }
            else {
                // Delete right child
                parent.setRight( null );
            }
            // Update the cursor
            cursor = parent;
        }
    }

    public boolean equals( Object o ) {
        boolean retVal = false;
```

continued

```
        // We can only do the comparison if the other object is a tree
        if ( o instanceof BinaryTree ) {
            BinaryTree other = (BinaryTree<?>)o;

            // Start at the top of both trees
            toRoot();
            other.toRoot();

            // Use the recursive helper method to do the actual work
            retVal = equalHelper( other );

            // Reset the cursors
            toRoot();
            other.toRoot();
        }

        return retVal;
    }

    /**
     * Compare the content and structure of this tree to a second
     * binary tree.
     *
     * @param tree the tree to compare this tree with
     *
     * @return true if the content and structure of a given tree are
     *         identical to this tree and false otherwise.
     */
    private boolean equalHelper( BinaryTree<?> tree ) {
        // Handle the case in which both trees are empty.  If both
        // trees are empty, they are equal.
        boolean retVal = ( cursor == null && !tree.isValid() );

        // If both trees have something in them - compare them
        if ( cursor != null && tree.isValid() ) {

            // Both nodes should have the same data and both
            // should have the same type of children
            retVal = cursor.getData().equals( tree.get() ) &&
                     hasLeftChild() == tree.hasLeftChild() &&
                     hasRightChild() == tree.hasRightChild();

            // If they are equal and have left children - compare them
            if ( retVal && hasLeftChild() ) {
                toLeftChild();
                tree.toLeftChild();
                retVal = equalHelper( tree );
```

continued

```
            // Back up the cursors
            toParent();
            tree.toParent();
        }

        // If they are equal and have right children - compare them
        if ( retVal && hasRightChild() ) {
            toRightChild();
            tree.toRightChild();
            retVal = retVal && equalHelper( tree );

            // Back up the cursors
            toParent();
            tree.toParent();
        }
    }

    return retVal;
}


public int size() {
    int size = 0;

    // If the root is null, the size of the tree is zero.
    // Otherwise, the size is the size of the root node.
    if ( root != null ) {
        size = root.size();
    }

    return size;
}

public int height() {
    int height = 0;
    // If the root is null, the height of the tree is zero.
    // Otherwise, the height is the height of the root node.
    if ( root != null ) {
        height = root.height();
    }

    return height;
}

public void find( T key) {
    // If the root is null, the key is not in the tree.
    // Otherwise, check the tree rooted at the root node.
```

continued

```
        if (root != null) {
            cursor = root.find(key);
        }
        else {
            cursor = null;
        }
    }

    public void preOrder( Callback<T> cb ) {
        // Start the traversal at the root node
        if ( root != null ) {
            root.preOrder( cb );
        }
    }

    public void inOrder( Callback<T> cb ) {
        // Start the traversal at the root node
        if ( root != null ) {
            root.inOrder( cb );
        }
    }

    public void postOrder( Callback<T> cb ) {
        // Start the traversal at the root node
        if ( root != null ) {
            root.postOrder( cb );
        }
    }

    /**
     * Return a string representation of this tree.  The string
     * returned by this method will show the structure of the
     * tree if the string is rotated 90 degrees to the right.
     *
     * @return a string representation of this tree.
     */
    public String toString() {
            StringBuffer retVal = new StringBuffer();

            // Get the string
            treeToString( root, retVal, "" );

            // Convert the string buffer to a string
            return retVal.toString();
    }
```

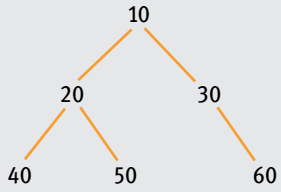
continued

```
    /**
     * A recursive method that does an RVL traversal of the tree
     * to create a string that shows the contents and structure of
     * the tree.
     *
     * For a tree that has the following structure:
     *
     *                      A
     *                B           C
     *            D       E
     *
     * This method will return the following string:
     *
     *     C
     * A
     *         E
     *     B
     *         D
     *
     * @param cur the tree being converted
     * @param str the converted tree
     * @param indent the indentation for the current level
     */
    private void treeToString( BinaryTreeNode<T> cur,
                               StringBuffer str,
                               String indent ) {

        if ( cur != null ) {
            // Get the string representation of the right child.
            // Indent is increased by 4 because this subtree is
            // one level deeper in the tree.
            treeToString( cur.getRight(), str, indent + "    " );

            // Convert the information in the current node
            str.append( indent );
            str.append( cur.getData().toString() );
            str.append( "\n" );

            // Get the string representation of the left child.
            treeToString( cur.getLeft(), str, indent + "    " );
        }
    }

} // LinkedBinaryTree
```

[FIGURE 7-13] A reference-based binary tree class

7.3.5 An Array-Based Implementation of Binary Trees

The reference-based implementation of binary trees discussed in Section 7.3.4 and shown in Figure 7-13 is highly efficient, but it is not the only choice. For example, we could implement a binary tree using an array. (We used a similar technique in Section 6.2.3.1 to implement a list.) We create a one-dimensional array of objects, called `data`, to hold the information field of a node, and a two-dimensional $k \times 2$ array, called `children`, in which the first column holds the array index of the root of the left subtree, and the second column holds the array index of the root of the right subtree. Thus, the following six-node tree:



could be represented using 10-element arrays, as shown in Figure 7-14.

root = 3

	data	*children* (left)	*children* (right)
0	60	–1	–1
1	20	6	2
2	50	–1	–1
3	10	1	5
4	—	—	—
5	30	–1	0
6	40	–1	–1
7	—	—	—
8	—	—	—
9	—	—	—

[FIGURE 7-14] Implementation of a binary tree using arrays

The root of the tree, the integer 10, has been stored in row 3 of the data array, as indicated by the state variable `root` that has the value 3. (We can place the root wherever we want, because the variable `root` indicates where it is located.) The root of the left subtree,

the value 20, is in row 1, as specified by the entry 1 in the column labeled left in row 3, while the root of the right subtree is in row 5. The rest of the tree is stored in similar fashion. If a node does not have a left child or a right child, the value −1 is stored in the appropriate column of the children array. We can use this value without the possibility of misinterpretation because Java array indices start at 0. Finally, to indicate that a row of the `data` array is empty and not being used, Figure 7-14 uses the character value −. To implement this concept, we could use the Java value **null**, assuming that **null** is not a data value that can appear in the data field of a node.

Figure 7-15 shows the code for an array-based `BinaryTree` class that implements the `BinaryTree` interface of Figure 7-10b. It uses the approach that is diagrammed in Figure 7-14.

```
/**
 * An array-based implementation of a binary tree. Javadoc comments
 * for methods specified in the BinaryTree interface have been omitted.
 *
 * This code assumes that the preconditions stated in the comments are
 * true when a method is invoked and therefore does not check the
 * preconditions.
 */
public class ArrayBasedBinaryTree<T> implements BinaryTree<T> {
    // Constants for selecting children
    public static final int LEFT = 0;
    public static final int RIGHT = 1;
    public static final int FREE = 0;

    // Constants for selecting traversals
    public static final int IN_ORDER = 0;
    public static final int PRE_ORDER = 1;
    public static final int POST_ORDER = 2;

    // Marks an empty position in the array
    public static final int OFF_TREE = -1;

    // Initial size of the arrays
    public static final int INITIAL_SIZE = 100;

    private T data[];           // Array containing node data
    private int children[][];   // The children

    private int root;           // The root node
    private int cursor;         // Current node
    private int count;          // Number of nodes in the tree

    private int availHead;      // Start of free list
```

continued

```
/**
 * Create a new tree.
 */
public ArrayBasedBinaryTree() {
  // Initialize storage arrays

  // Note that the cast is necessary because you cannot
  // create generic arrays in Java.  This statement will
  // generate a compiler warning.
  data  = (T[])new Object[ INITIAL_SIZE ];
  children = new int[ INITIAL_SIZE ][ 2 ];

  // Tree is empty
  root = OFF_TREE;
  cursor = OFF_TREE;
  count = 0;

  // Initialize children array
  availHead = 0;
  for ( int i = 0; i < INITIAL_SIZE - 1; i++ ) {
      // Add this node to our free list.
      children[ i ][ FREE ] = i + 1;
  }

  // The end of the free list
  children[ INITIAL_SIZE - 1 ][ FREE ] = OFF_TREE;
}

public void toRoot() {
  cursor = root;
}

public boolean hasLeftChild() {
  return children[ cursor ][ LEFT ] != OFF_TREE;
}

public boolean hasRightChild() {
  return children[ cursor ][ RIGHT ] != OFF_TREE;
}

public boolean hasParent() {
  int parent = OFF_TREE;

  // If there is a tree to search, look for the parent
  if ( root != OFF_TREE ) {
      parent = findParent( root, cursor );
  }
```

continued

```
  // If we found a parent, return true
  return parent != OFF_TREE;
}

public boolean isValid() {
  return cursor != OFF_TREE;
}

public void toParent() {
  // If there is no tree to search, invalidate the cursor
  if ( root == OFF_TREE ) {
      cursor = OFF_TREE;
  }
  else {
      // Set the cursor to the parent
      cursor = findParent( root, cursor );
  }
}

public void toLeftChild() {
  cursor = children[ cursor ][ LEFT ];
}

public void toRightChild() {
  cursor = children[ cursor ][ RIGHT ];
}

public void insertLeft( T element ) {
  // Find the place where the new node will be placed
  int pos = nextIndex();
  data[ pos ] = element;

  // If the tree is empty, this becomes the only node in the
  // tree; otherwise, the new node is the left child of the cursor
  if ( root == OFF_TREE ) {
      root = pos;
  }
  else {
      children[ cursor ][ LEFT ] = pos;
  }
}

public void insertRight( T element ) {
  // Find the place where the new node will be placed
  int pos = nextIndex();
  data[ pos ] = element;
```

continued

```
    // If the tree is empty, this becomes the only node in the
    // tree; otherwise, the new node is the right child of the cursor
    if ( root == OFF_TREE ) {
        root = pos;
    }
    else {
        children[ cursor ][ RIGHT ] = pos;
    }
  }

  public T get() {
    return data[ cursor ];
  }

  public void set( T element ) {
    data[ cursor ] = element;
  }

  public void prune() {
    int parent = findParent( root, cursor );

    // Add the nodes in the tree being deleted to the free list
    freeIndex( cursor );

    // Did we delete the entire tree?
    if ( parent == OFF_TREE ) {
        // Invalidate the root and the cursor
        root = OFF_TREE;
        cursor = OFF_TREE;
    }
    else {
        // Is it the parent's left child?
        if ( children[ parent ][ LEFT ] == cursor ) {
          // Deleted the left child
          children[ parent ][ LEFT ] = OFF_TREE;
        }
        else {
          // Deleted the right child
          children[ parent ][ RIGHT ] = OFF_TREE;
        }

        // Update the cursor
        cursor = parent;
    }
  }

  public boolean equals( Object o ) {
    boolean retVal = false;
```

continued

```
  // We can only do the comparison if the other object is a tree
  if ( o instanceof BinaryTree ) {
      BinaryTree other = (BinaryTree<?>)o;

      // Start at the top of both trees
      toRoot();
      other.toRoot();

      // Use the recursive helper method to do the actual work
      retVal = equalHelper( other );

      // Reset the cursors
      toRoot();
      other.toRoot();
  }

  return retVal;
}

/**
 * Compare the content and structure of this tree to a second
 * binary tree.
 *
 * @param tree the tree to compare this tree with
 *
 * @return true if the content and structure of a given tree are
 *         identical to this tree and false otherwise.
 */
private boolean equalHelper( BinaryTree<?> tree ) {
  // Handle the case in which both trees are empty.  If both
  // trees are empty, they are equal.
  boolean retVal = ( cursor == OFF_TREE && !tree.isValid() );

  // If both trees have something in them, then compare them
  if ( cursor != OFF_TREE && tree.isValid() ) {

      // Both nodes should have the same data and both
      // should have the same type of children
      retVal = data[ cursor ].equals( tree.get() ) &&
        hasLeftChild() == tree.hasLeftChild() &&
        hasRightChild() == tree.hasRightChild();

      // If they are equal and have left children, compare them
      if ( retVal && hasLeftChild() ) {
        toLeftChild();
        tree.toLeftChild();
        retVal = equalHelper( tree );
```

continued

```
        // Back up the cursors
        toParent();
        tree.toParent();
      }

      // If they are equal and have right children, compare them
      if ( retVal && hasRightChild() ) {
        toRightChild();
        tree.toRightChild();
        retVal = retVal && equalHelper( tree );

        // Back up the cursors
        toParent();
        tree.toParent();
      }
    }

    return retVal;
  }

  public int size() {
    return count;
  }

  public int height() {
    int height = 0;

    // If there is a tree, determine its height
    if ( root != OFF_TREE ) {
        height = height( root );
    }

    return height;
  }

  /**
   * Determine the height of the tree with the specified root.
   *
   * @param the root of the tree.
   *
   * @return the height of the tree.
   */
  private int height( int root ) {
    int leftHeight = 0;
    int rightHeight = 0;
    int height = 0;
```

continued

```
    // Determine the height of the left subtree
    if ( children[ root ][ LEFT ] != OFF_TREE ) {
        leftHeight = height( children[ root ][ LEFT ] );
    }

    // Determine the height of the right subtree
    if ( children[ root ][ RIGHT ] != OFF_TREE ) {
        rightHeight = height( children[ root ][ RIGHT ] );
    }

    // The height of the tree rooted at this node is one more than the
    // height of the taller of its children.
    if (leftHeight > rightHeight) {
        height = 1 + leftHeight;
    }

    else {
        height = 1 + rightHeight;
    }

    // Return the answer
    return height;
  }

  public void find( T target ) {
    // If there is no tree, cursor should be off the tree
    if ( root == OFF_TREE ) {
        cursor = OFF_TREE;
    }
    else {
        // Find the target and set cursor
        cursor = search( root, target );
    }
  }

  /**
   * Search the tree at the specified root for the given target
   *
   * @param root the root of the tree to search.
   * @param target the element being looked for.
   *
   * @return the location of the node that contains the target,
   *         or OFF_TREE if the target cannot be found.
   */
  private int search( int root, T target ) {
    int location = OFF_TREE;
```

continued

```
    // Is the target in the left subtree?
    if ( children[ root ][ LEFT ] != OFF_TREE ) {
        location = search( children[ root ][ LEFT ], target );
    }

    // If we haven't found it, is it in this node?
    if (location == OFF_TREE && target.equals( data[ root ] ) ) {
        location = root;
    }

    // If we haven't found it, and we have a right child, check there
    if ( location == OFF_TREE &&
         children[ root ][ RIGHT ] != OFF_TREE ){
        location = search( children[ root ][ RIGHT ], target );
    }

    // Return the location of the target
    return location;
  }

  /**
   * Return a string representation of this tree. The string
   * returned by this method will show the structure of the
   * tree if the string is rotated 90 degrees to the right.
   *
   * @return a string representation of this tree.
   */
  public String toString() {
    StringBuffer retVal = new StringBuffer();

    // Get the string
    treeToString( root, retVal, "" );

    // Convert the string buffer to a string
    return retVal.toString();
  }

  /**
   * A recursive method that does an RVL traversal of the tree
   * to create a string that shows the contents and structure of
   * the tree.
   *
   * For a tree that has the following structure:
   *
   *                      A
   *                B           C
   *            D       E
   *
```

continued

```
 * This method will return the following string:
 *
 *     C
 * A
 *         E
 *     B
 *         D
 * @param cur the tree being converted
 * @param str the converted tree
 * @param indent the indentation for the current level
 */
private void treeToString( int cur,
                           StringBuffer str,
                           String indent ) {
  if ( cur != OFF_TREE ) {
      // Get the string representation of the right child. Indent
      // is increased by 4 because this subtree is one level
      // deeper in the tree
      treeToString( children[ cur ][ RIGHT ],
                    str,
                    indent + "    " );

      // Convert the information in the current node
      str.append( indent );
      str.append( data[ cur ].toString() );
      str.append( "\n" );

      // Get the string representation of the left child
      treeToString( children[ cur ][ LEFT ], str, indent + "    " );
  }
}

public void preOrder( Callback<T> cb ) {
  // Start the traversal at the root node
  if ( root != OFF_TREE )  {
      preOrder( root, cb );
  }
}

/**
 * Do a pre-order traversal of the tree with the given root.
 *
 * @param root the root of the tree to traverse.
 * @param cb the callback object used to process each node.
 */
```

continued

```
private void preOrder( int root, Callback<T> cb ) {
  // Process the data in the current node
  cb.visit( data[ root ] );

  // Process the left child - if there is one
  if ( children[ root ][ LEFT ] != OFF_TREE ) {
      preOrder( children[ root ][ LEFT ], cb );
  }

  // Process the right child - if there is one
  if ( children[ root ][ RIGHT ] != OFF_TREE ) {
      preOrder( children[ root ][ RIGHT ], cb );
  }
}

public void inOrder( Callback<T> cb ) {
  // Start the traversal at the root node
  if ( root != OFF_TREE ) {
      inOrder( root, cb );
  }
}

/**
 * Do an inorder traversal of the tree with the given root.
 *
 * @param root the root of the tree to traverse.
 * @param cb the callback object used to process each node.
 */
private void inOrder( int root, Callback<T> cb ) {
  // Process the left child, if there is one
  if ( children[ root ][ LEFT ] != OFF_TREE ) {
      inOrder( children[ root ][ LEFT ], cb );
  }

  // Process the current node
  cb.visit( data[ root ] );

  // Process the right child, if there is one
  if ( children[ root ][ RIGHT ] != OFF_TREE ) {
      inOrder( children[ root ][ RIGHT ], cb );
  }
}

public void postOrder(Callback<T> cb) {
  // Start the traversal at the root node
```

continued

```
    if (root != OFF_TREE) {
        postOrder( root, cb );
    }
}

/**
 * Do a postorder traversal of the tree with the given root.
 *
 * @param root the root of the tree to traverse.
 * @param cb the callback object used to process each node.
 */
private void postOrder( int root, Callback<T> cb ) {
    // Process the left child, if there is one
    if ( children[ root ][ LEFT ] != OFF_TREE ) {
        postOrder( children[ root ][ LEFT ], cb );
    }

    // Process the right child, if there is one
    if ( children[ root ][ RIGHT ] != OFF_TREE ) {
        postOrder( children[ root ][ RIGHT ], cb );
    }

    // Process the current node
    cb.visit( data[ root ] );
}

/**
 * Return the position of the parent given a root of the tree
 * and the child.
 *
 * @param root the root of the tree to search.
 * @param child the child whose parent we are looking for.
 *
 * @return the location of the parent in the tree or OFF_TREE
 *         if the parent cannot be found.
 */
protected int findParent( int root, int child ) {
    int parent = OFF_TREE;
    // Is the root the parent?
    if ( children[ root ][ LEFT ] == child ||
         children[ root ][ RIGHT ] == child ) {
        parent = root;
    }
```

continued

```
    else {
        // Check left child, if there is one
        if ( children[ root ][ LEFT ] != OFF_TREE ) {
          parent = findParent( children[ root ][ LEFT ], child );
        }
        // If it has not been found, check the right child
        if ( parent == OFF_TREE &&
           children[ root ][ RIGHT ] != OFF_TREE ) {
          parent = findParent( children[ root ][ RIGHT ], child );
        }
    }

    return parent;
  }

  /**
   * Locate the next open index in the arrays holding the tree nodes.
   * If there is not enough room in the arrays for the new node,
   * the arrays will be expanded.
   *
   * @return the next open index in the node arrays.
   */
  private int nextIndex() {
    int retVal;

    // If the free space is gone, expand the arrays
    if ( availHead == OFF_TREE ) {
        expand();
    }

    // Take the location at the front of the list
    retVal = availHead;
    availHead = children[ availHead ][ FREE ];

    // Ensure the location doesn't link to the available list anymore
    children[ retVal ][ LEFT ] = OFF_TREE;
    children[ retVal ][ RIGHT ] = OFF_TREE;

    // One more node in the tree
    count = count + 1;

    // Return the index
    return retVal;
  }

  /**
   * Double the capacity of the arrays that hold the data and the
```

continued

```
     * links.
     */
    private void expand() {
      // Make the new arrays twice the size of the old arrays

      // Note that the cast is necessary because you cannot
      // create generic arrays in Java.  This statement will
      // generate a compiler warning.

      T newData[] = (T[])new Object[ data.length * 2 ];
      int newChildren[][] = new int[ newData.length ][ 2 ];

      // Copy the contents of the old arrays to the new arrays
      for ( int i = 0; i < data.length; i = i + 1 ) {
          newData[ i ] = data[ i ];
          newChildren[ i ][ LEFT ] = children[ i ][ LEFT ];
          newChildren[ i ][ RIGHT ] = children[ i ][ RIGHT ];
      }

      // Add the empty space to the available list
      for ( int i = data.length; i < newData.length; i++ ) {
          newChildren[ i ][ FREE ] = availHead;
          availHead = i;
      }

      // Start using the new arrays
      data = newData;
      children = newChildren;
    }

    /**
     * Add the array locations occupied by the specified tree to the
     * free list.
     *
     * @param root the root of the tree to delete.
     */
    private void freeIndex( int root ) {
      // Add the nodes in the left tree to the free list
      if ( children[ root ][ LEFT ] != OFF_TREE ) {
          freeIndex( children[ root ][ LEFT ] );
          children[ root ][ LEFT ] = OFF_TREE;
      }

      // Add the nodes in the right tree to the free list
      if ( children[ root ][ RIGHT ] != OFF_TREE ) {
          freeIndex( children[ root ][ RIGHT ] );
          children[ root ][ RIGHT ] = OFF_TREE;
      }
```

continued

```
        // Add this node to the free list
        children[ root ][ FREE ] = availHead;
        availHead = root;

        // One less node in the tree
        count = count - 1;
      }

} // ArrayBasedBinaryTree
```

[FIGURE 7-15] An array-based binary tree class

The array implementation of Figure 7-15 suffers from the same problem mentioned in Section 6.2.3.1 with respect to linked lists. Specifically, the programmer must know in advance exactly how much array space to allocate (the declaration `INITIAL_SIZE = 100` in Figure 7-15). If we attempt to insert 101 nodes into our binary tree, we overflow the array and must resize it. If, instead, the tree has only a few nodes, a good deal of memory is wasted. Because of these problems, the reference-based implementation of Figure 7-13 is usually preferred.

We have now looked at tree structures with few, if any, restrictions on either the location where you can perform operations or the type of data stored in a node. In a general tree you may insert, delete, or access any node. You have the same options with a binary tree, with the one restriction that no node may have more than two children.

In the following sections, we examine special-purpose tree structures that place more severe restrictions on either the type of data stored in a node or the location where you can insert new nodes. You can use these trees to efficiently solve problems in such areas as searching, sorting, and finding maxima or minima.

7.4 Binary Search Trees

7.4.1 Definition

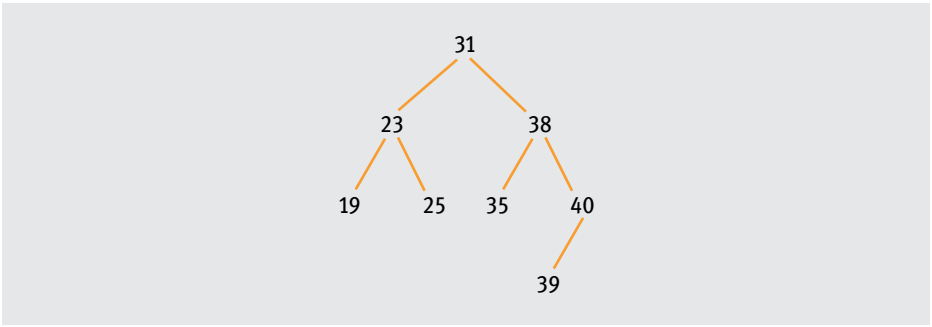
One important variation of the binary tree, called the **binary search tree** (BST), is extremely useful for carrying out search operations. In a BST, one of the data values stored in the information field of a node is a special value called the **key**. Every key value in the left subtree of a node is less than the key value in that node, and every key value in the right subtree of a node is greater than the key value in that node. This ordering is called the **binary search tree property**.

A characteristic of the key is that you can always compare two of them against each other to determine if one key is less than, equal to, or greater than another. To ensure that this comparison always makes sense, we can require the key field to be a primitive value, such as an integer, character, or real, which can always be compared using the operators $<$, $=$, and $>$. Another way is to require that the data type of the key field implements the Java interface `Comparable`. This interface imposes a natural ordering on the objects of any class that implements it. It contains a single method, `compareTo(T t)`, that compares two objects of type `T` and then returns a negative value, 0, or positive value to indicate that the **this** object is less than, equal to, or greater than `t`. For example, if `x` and `y` are objects of type `T`, which implements `Comparable`, then `x.compareTo(y)` returns a negative number if `x` $<$ `y`, 0 if `x` $=$ `y`, and a positive number if `x` $>$ `y`. In this chapter we assume that keys are simple integer values.

In addition to requiring that keys be comparable, it is usually assumed (but not required) that the keys in a BST are unique, and that duplicates are not allowed. We make this assumption in all of our examples, so two keys should never test equal. If duplicates were allowed, we would modify the definition of a BST in one (but not both) of the following ways (the change is shown in boldface):

- Every key value in the left subtree of a node is less than *or equal to* the key value in that node.
- Every key value in the right subtree of a node is less than *or equal to* the key value in that node.

Figure 7-16 is an example of a binary search tree.



[FIGURE 7-16] Example binary search tree

Every key in the left subtree of the root of Figure 7-16, {19, 23, 25}, is less than the root value 31, and every key in the right subtree of the root, {35, 38, 39, 40}, is greater than the root value 31. We can see that this ordering property holds not just for the root node but for *every* node in the tree. For example, every key in the left subtree of the tree rooted at 23 {19} is less than 23, and every key in the right subtree {25} is greater than 23.

Every key in the left subtree of the tree rooted at 38 {35} is less than 38, and every key in the right subtree {39, 40} is greater than 38.

7.4.2 Using Binary Search Trees in Searching Operations

There are two important differences between the binary search tree just introduced and the binary tree in Section 7.3. First, a new node can be inserted anywhere into a binary tree, but insertion into a binary search tree must guarantee that the `insert()` method maintains the binary search tree property. Second, the `contains()` method for a binary search tree class, which searches the tree to locate a specific key `t`, can exploit the binary search tree property to speed up the process of locating a value in the tree. A `BinarySearchTree` interface that includes both of these methods—`insert()` and `contains()`—is shown in Figure 7-17. It also includes two utility methods: `size()`, which returns the number of nodes in the tree, and `height()`, which returns the height of the tree.

```
/**
 * An interface for the binary search tree ADT.
 * This tree does not allow duplicate entries.
 */
public interface BinarySearchTree<T extends Comparable> {
    /**
     * Insert an item into the correct position within the tree.
     *
     * Preconditions:
     *   The item is not currently in the tree.
     *
     * Postconditions:
     *   The size of the tree has increased by one.
     *   The item is in the correct position within the tree.
     *
     * @param data the item to insert into the tree.
     */
    public void insert( T info );

    /**
     * Determine the size of the tree.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The tree is unchanged.
     *
```

continued

```
     * @return the number of elements in the tree.
     */
    public int size();

    /**
     * Determine the height of the tree.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The tree is unchanged.
     *
     * @return the height of the tree.
     */
    public int height();

    /**
     * Determine if the given item is in the tree.
     *
     * Preconditions:
     *   Target is not null.
     *
     * Postconditions:
     *   The tree is unchanged.
     *
     * @param target the element being searched for.
     *
     * @return true if the item is found.
     */
    public boolean contains( T key );

} // BinarySearchTree
```

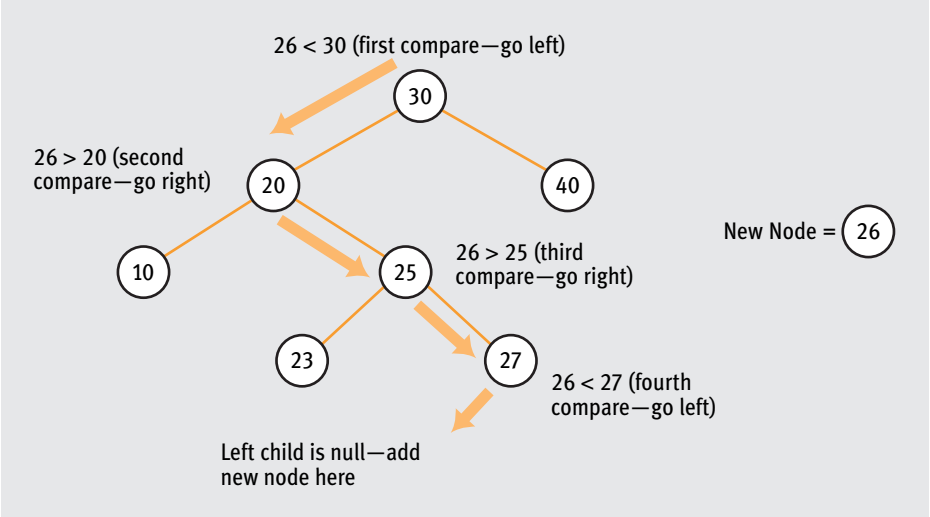
[FIGURE 7-17] Interface for a binary search tree

When inserting a node into the binary tree of Section 7.3, we used the instance variable `cursor` to identify where the new node should be attached. However, when inserting a node into a binary search tree, we do not specify the location of the insertion. Instead, we provide the new data value to be inserted and a reference to the root of the tree. The `insert()` method itself must determine where to insert this new value so the binary search tree property is maintained. This is done by comparing the value to be inserted, v, to the data value in the current node, n. If $v < n$, then follow the left branch of the node and repeat the

process. If $v > n$, then follow the right branch of the node and repeat the process. This continues until one of two things happens:

- $v = n$. This means the new item is already in the tree. We assumed that duplicates are not stored, so our method returns without modifying the tree. In some circumstances, this may be treated as either an update operation or an error.
- child pointer = **null**. This is the correct location to insert the new node.

Figure 7-18 shows the sequence of four comparisons required to correctly insert the value 26 into the binary search tree.



[FIGURE 7-18] Insertion of a new node in a binary search tree

We initially compare 26 to the value in the root of the tree, 30. Because it is less, we must go to the left. We repeat the process, comparing 26 to the value in the current node, which this time is 20. Because 26 is greater, we must move to the right. This process continues as we arrive at nodes containing a 25 and then a 27. When we compare our data value 26 to the node value 27 and see that it is less, we attempt to move to the left. However, the left pointer of this node is **null**. This means that we have found the correct place to insert the new node—as the left child of the node whose key value is 27. Examine the figure to confirm that the binary search tree property still holds for this new tree after the insertion.

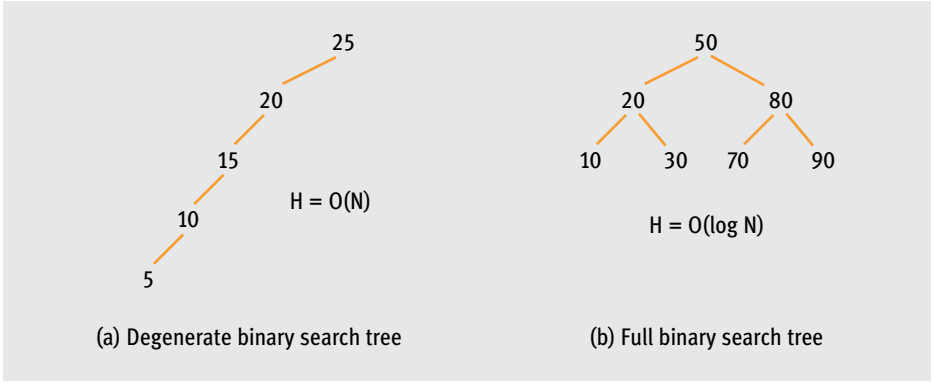
The questions we must now answer are: What is the advantage of using a binary search tree? What gains justify the price of maintaining the binary search tree property whenever we insert a new node? After all, the `insertLeft()` and `insertRight()` methods for ordinary binary trees, shown in Figure 7-10b, are efficient O(1) operations. In a binary search tree, we must search through it to locate the correct insertion point.

The answer lies in the speed and efficiency of the `contains()` method. Searching through a collection of elements to locate a specific value is one of the most widely performed operations in computer science. When we search an unordered list of n values for a special key, we must search an average of $n/2$ items until we find what we want. In the worst case, we must look at all n items; because the complexity of a search on unordered lists is $O(n)$, this can be slow for large values of n. This condition also holds for binary trees, such as those in Figure 7-6. If we do not know exactly where a value is within a binary tree, the best we can do is search the entire tree using the traversal algorithms of Figures 7-8 or 7-9, which are also $O(n)$, as shown earlier.

However, in a binary search tree we do not have to examine all its nodes to locate a given key; we only examine nodes on the path followed by the `insert()` method when the key value was initially stored. Thus, we never need to look at more than H items, where H is the height of the binary search tree, the distance required to reach the leaf node farthest from the root. We saw this in Figure 7-18, where we traveled from the root node to a leaf node looking for the correct place to insert the value 26. The distance traveled was H, the height of the tree.

Given that conclusion, the next question is: What is the value of H, and is it always the case that $H < n$? More specifically, can we execute the `contains` operation on a binary search tree more quickly than $O(n)$?

Figure 7-19 shows two extremes for the shape of a binary search tree. The tree in Figure 7-19(a) is called a **degenerate binary search tree**—one in which every nonterminal node has exactly one child. In a degenerate BST, $H = n$, where n is the number of nodes in the tree; we gain nothing because the BST has effectively become a linked list. (In fact, it is even worse, because we still have to allocate memory for all those unused references.) The time to locate a node is still $O(n)$.



[FIGURE 7-19] Height, *H*, of a binary search tree

The tree in Figure 7-19(b) represents the other extreme, a structure called a **full binary search tree**. In this type of tree, all leaf nodes are on the same level and all nonleaf nodes have degree = 2. Let's determine how many nodes n are contained in a full binary search tree of height H.

	Level	Number of Nodes on this level
(root)	1	1
	2	2
	3	4
	4	8
	…	…
(H)	i	2^{i-1}

The number of nodes on level i in a full binary search tree is 2^{i-1}, so the total number of nodes n in the tree of Figure 7-19(b) is the sum of the number of nodes on all levels from 1 to H, the height of the tree:

$$n = \sum_{i=1}^{H} 2^{i-1} \qquad = 2^0 + 2^1 + 2^2 + \quad + 2^{H-1}$$

$$= 2^H - 1$$

When solving for H, the height of the tree, we get:

$$H = \log_2(n + 1) \quad \text{or}$$

$$H = O(\log n)$$

Thus, the maximum distance from the root to the leaves is a logarithmic function of n, the number of nodes in the tree, rather than a linear function of n. If our binary search tree looks more like the one in Figure 7-19(b) than the one in Figure 7-19(a), we have gained a great deal. We can locate any item in the tree with at most $O(\log n)$ comparisons, rather than the $O(n)$ required by either the sequential search of an unordered list or the traversal of an unordered binary tree.

For large n, $\log n$ is much smaller than n. For example, if $n = 100{,}000$, the search of either an unordered n-element list or a binary tree with n nodes takes an average of 50,000 comparisons. However, we can locate any node in a balanced binary search tree in $\log_2 100{,}000 = 17$ comparisons, an improvement of more than three orders of magnitude. As you saw in Chapter 5, the "efficiency game" in software is most strongly affected by your choice of data representations, not by the details of coding or the speed of the machine that runs the final program. This example clearly illustrates the point.

Figure 7-20 shows the code for `LinkedBinarySearchTree`, a linked list implementation of the `BinarySearchTree` interface in Figure 7-17. The only instance variable needed in this implementation is `root`, a pointer to the root of the entire tree. We no longer need a

current position indicator, `cursor`, because the `insert()` method itself determines the proper location for the insertion operation. Note that `insert()` first checks to see if `root == null`, because a binary search tree, like a binary tree, can be empty. Also, the generic type for the class extends the `Comparable` interface, which guarantees that the parameters to both `insert()` and `contains()` provide a `compareTo()` method. This method can compare objects to determine their proper ordering.

```
/**
 * A node in a binary search tree.
 */
public class BinarySearchTreeNode<T extends Comparable> {
    T data;                          // Data stored in this node
    BinarySearchTreeNode<T> left;    // Left child
    BinarySearchTreeNode<T> right;   // Right child

    /**
     * Create a new node.
     */
    public BinarySearchTreeNode() {
        this( null );
    }

    /**
     * Create a new node that contains the specified element.
     *
     * @param element the element to place in the node.
     */
    public BinarySearchTreeNode( T element ) {
        data = element;
        left = null;
        right = null;
    }

    /**
     * Get the data stored in this node.
     *
     * @return the data stored in this node.
     */
    public T getData() {
        return data;
    }

    /**
     * Get the left child of this node.
     *
```

continued

```
     * @return the left child of this node.
     */
    public BinarySearchTreeNode<T> getLeft() {
        return left;
    }

    /**
     * Get the right child of this node.
     *
     * @return the right child of this node.
     */
    public BinarySearchTreeNode<T> getRight() {
        return right;
    }

    /**
     * Set the data to the specified value.
     *
     * @param newData the data to place in this node.
     */
    public void setData( T newData ) {
        data = newData;
    }

    /**
     * Set the left child of this node.
     *
     * @param newLeft the left child of this node.
     */
    public void setLeft( BinarySearchTreeNode<T> newLeft ) {
        left = newLeft;
    }

    /**
     * Set the right child of this node.
     *
     * @param newRight the right child of this node.
     */
    public void setRight( BinarySearchTreeNode<T> newRight ) {
        right = newRight;
    }

} // BinarySearchTreeNode

/**
 * A linked list-based binary search tree implementation.
 * Javadoc comments for methods specified in the BinarySearchTree
```

continued

```
 * interface have been omitted.
 *
 * This code assumes that the preconditions stated in the
 * comments are true when a method is invoked and therefore does
 * not check the preconditions.
 */
public class LinkedBinarySearchTree<T extends Comparable>
    implements BinarySearchTree<T> {

    private BinarySearchTreeNode<T> root; // The root of the tree

    public void insert( T data ) {
        BinarySearchTreeNode<T> cur = root;
        BinarySearchTreeNode<T> newNode =
            new BinarySearchTreeNode<T>( data );

        if ( root == null ) {
            root = newNode;
        }
        else {
            while ( cur != null ) {
                int compare = data.compareTo( cur.getData() );

                if ( compare < 0 ) {
                    // The new item is less
                    if ( cur.getLeft() == null ) {
                        // No left child, so insert item here
                        cur.setLeft( newNode );
                        cur = null;
                    }
                    else {
                        // There is a left child - insert into it
                        cur = cur.getLeft();
                    }
                }
                else if ( compare > 0 ) {
                    // The new item is greater
                    if ( cur.getRight() == null ) {
                        // No right child, so insert item here
                        cur.setRight( newNode );
                        cur = null;
                  }
                  else {
                      // There is a right child - insert into it
                      cur = cur.getRight();
                  }
                }
```

continued

```
            else {
                // Item is already in the tree - do not add
                cur = null;
            }
        }
    }
}

public int size() {
    int size = 0;

    // If there is a tree, determine the size of it
    if (root != null) {
        size = size( root );
    }

    return size;
}

/**
 * Determine the size of the tree with the specified root.
 *
 * @param root the root of the tree.
 *
 * @return the size of the tree.
 */
private int size( BinarySearchTreeNode<T> root ) {
    int size = 0;  // The size of the tree

    // The size of the tree is one more than the sum of
    // the sizes of its children.
    if ( root.getLeft() != null ) {
        size = size( root.getLeft() );
    }

    if ( root.getRight() != null ) {
        size = size + size( root.getRight() );
    }

    return size + 1;
}

public int height() {
    int height = 0;
```

continued

```
        // If there is a tree, determine the height
        if ( root != null ) {
            height = height( root );
        }

        return height;
    }

    /**
     * Determine the height of the tree with the specified root.
     *
     * @param root the root of the tree.
     *
     * @return the height of this tree.
     */
    private int height( BinarySearchTreeNode<T> root ) {
        int leftHeight = 0;   // Height of the left subtree
        int rightHeight = 0;  // Height of the right subtree
        int height = 0;       // The height of this subtree

        // If we have a left subtree, determine its height
        if ( root.getLeft() != null )  {
            leftHeight = height( root.getLeft() );
        }

        // If we have a right subtree, determine its height
        if ( root.getRight() != null )  {
            rightHeight = height( root.getRight() );
        }

        // The height of the tree rooted at this node is one more
        // than the height of the 'taller' of its children.
        if (leftHeight > rightHeight) {
            height = 1 + leftHeight;
        }
        else {
            height = 1 + rightHeight;
        }

        // Return the answer
        return height;
    }

    public boolean contains( T target ) {
        boolean found = false;
        BinarySearchTreeNode<T> cur = root;
```

continued

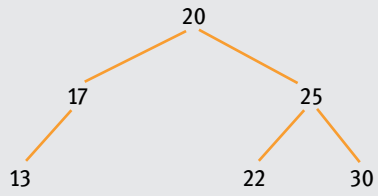
```
        // Keep looking until we find it or fall off the tree
        while ( !found && cur != null ) {
            int compare = target.compareTo( cur.getData() );

            if ( compare < 0 ) {
                // The target is smaller - look left
                cur = cur.getLeft();
            }
            else if ( compare > 0 ) {
                // The target is greater - look right
                cur = cur.getRight();
            }
            else {
                // Found it!!
                found = true;
            }
        }

        return found;
    }

    /**
     * Return a string representation of this tree. The string
     * contains the elements in the tree listed in the order they
     * were found during an inorder traversal of the tree.
     *
     * @return a string representation of this tree.
     */
    public String toString() {
        // Use a string buffer to avoid temporary strings
        StringBuffer result = new StringBuffer();

        // If there is a tree, traverse it
        if ( root != null ) {
            inorder( root, result );
        }

        // Return the result
        return result.toString();
    }

    /**
     * Perform an inorder traversal of the tree.  When a node is
     * processed, the contents of the node are converted to a
     * string and appended to the specified string buffer.
     *
```

continued

```
     * @param root the root of the tree to traverse.
     * @param result the string buffer that will contain the
     *        string representation of the strings in the tree.
     */
    private void inorder( BinarySearchTreeNode<T> root,
                          StringBuffer result ) {
        // If there is a left child, traverse it
        if ( root.getLeft() != null ) {
            inorder( root.getLeft(), result );
        }

        // Convert the data to a string and append the result
        // to the string buffer. A space is placed between
        // consecutive elements in the string
        result.append( root.getData() );
        result.append( " " );

        // If there is a right child, traverse it
        if ( root.getRight() != null ) {
            inorder( root.getRight(), result );
        }
    }

} // LinkedBinarySearchTree
```

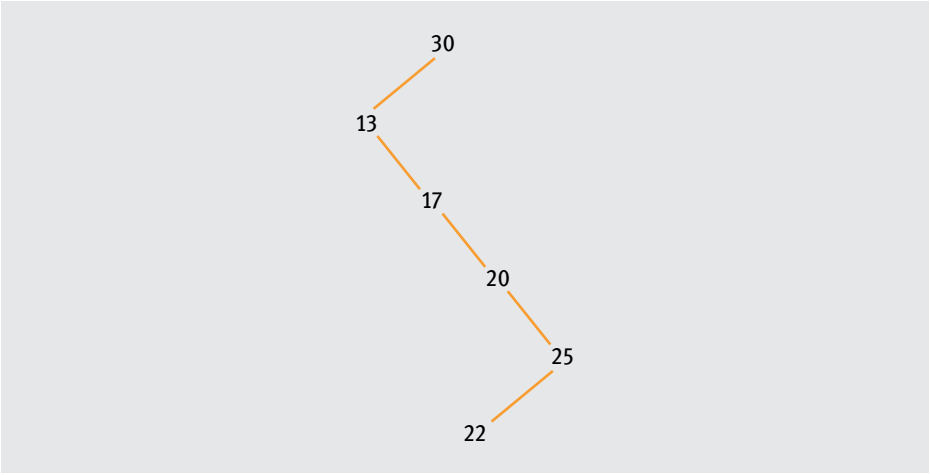
[FIGURE 7-20] Linked list implementation of a binary search tree

We will address one final question: Assuming that we are given a random sequence of n data values and we construct a binary search tree from this random sequence, will the final height H of the tree be closer to n [Figure 7-19(a)] or to $\log_2 n$ [Figure 7-19(b)]? That is, will its final shape tend to be an unbalanced degenerate form or a more balanced full structure? For example, given the following six-element sequence {20, 17, 25, 22, 13, 30}, the `insert` method of Figure 7-20 produces the following binary tree structure:



This balanced structure has the least possible height for a binary tree that contains six nodes: $H = 3$, which is $O(\log_2 n)$. This is called a **minimum height binary search tree**.

However, if the same six input values are used, but in a different order—{30, 13, 17, 20, 25, 22}—we end up with the following structure after inserting each number into our BST:



This is a degenerate tree with a height $H = 6$, which is $O(n)$. Given a random collection of six values, are these two cases equally likely? If not, which structure can we expect to observe?

This is a difficult and complex mathematical problem that we will not solve here. Instead, we simply state the result. A binary search tree constructed from a random sequence of values is roughly balanced. It probably will look more like the balanced tree in Figure 7-19(b) than the degenerate one in Figure 7-19(a), and its height H will satisfy the relationship $H = O(\log n)$. Although a binary search tree built from a random input sequence will almost certainly not be a minimum-height tree, its height H will be within a small constant factor k of that minimum, $H = k \times \log_2 n$, which makes it $O(\log n)$. Exercise 13 at the end of this chapter asks you to demonstrate this behavior empirically by analyzing the height of several binary search trees constructed from random input sequences. The exercise also allows you to approximate the value of k, the factor by which a randomly constructed binary search tree exceeds the minimum $\log_2 n$ height of a perfectly balanced binary search tree that contains n nodes.

Thus, for searching arbitrary lists of values, a binary search tree can be a highly efficient data structure that lets you locate any item in the collection in logarithmic time. However, remember that some input sequences produce a degenerate tree, leading to a worst-case linear-time behavior. To avoid this problem, the Java Collection Framework makes use of a BST variation called a **red-black tree**. This type of binary search tree is always kept in a balanced state. When we insert a new value, we see if the tree is still balanced; if it is not, we immediately perform a rebalancing operation. We discuss the red-black binary search tree in Section 7.5.2.

SPACE TO SPARE

Large problems are common in computer science; they occur often enough to make it worthwhile to study advanced data structures. For example, Chapter 5 included a feature entitled "The Mother of all Computations," which described a 1000-year climate model that requires 10^{20} computations. However, not only do massive processing demands drive the need for higher efficiency, so does the existence of massive amounts of *data*. Applications such as high-resolution graphics, data mining, and remote telemetry generate incredibly large volumes of information that must be stored and processed in a reasonable amount of time.

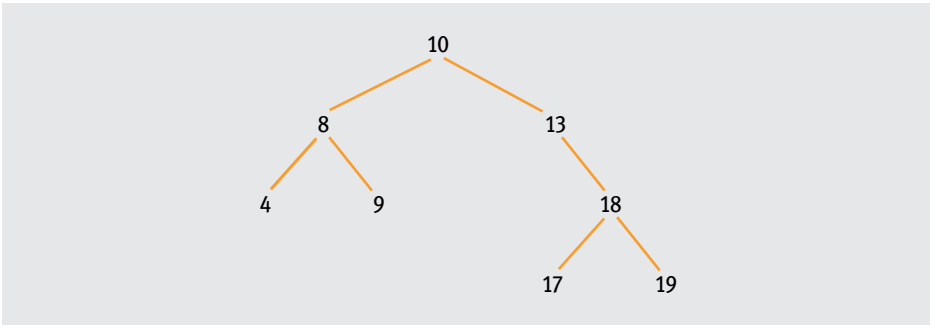
We are all familiar with the term **gigabyte**, which is one billion bytes of information. The first low-cost, widely available gigabyte storage devices appeared in the mid-1990s. Now virtually all desktop hard drives have capacities of at least 80 to 300 GB, at a cost of only a few cents per gigabyte. In the last three to four years, a number of companies have begun to produce low-cost **terabyte** storage devices, desktop units that can store a minimum of 1000 GB. To get an idea of how much information this is, consider that the entire U.S. Library of Congress contains only about 20 terabytes of text!

However, text is not driving this explosion of data; imaging is. For example, recording just two hours of HDTV requires about 1 terabyte. We are beginning to see storage devices based on the "next unit" of metric storage—the **petabyte**, or one million gigabytes. The first commercial petabyte storage array came to market in January 2006, and others have appeared since. Many applications require this massive volume of storage. For example, the Internet has a 2-petabyte storage device (jokingly called the "Wayback Machine") to archive its records. Google reportedly maintains about 5 petabytes of data storage. Some computer scientists estimate that a low-cost, desktop petabyte storage device will be available for home use in a couple of years.

Next on the horizon is the **exabyte**—an almost unimaginable one billion gigabytes. Although an exabyte storage device has no practical applications now, remember that there were none for a gigabyte storage device in the 1960s and 1970s. Future applications could certainly demand this level of data storage. For example, if we want to construct a robot that can store its total visual input over a 40-year period (think of the android Commander Data from the *Star Trek* TV series), scientists estimate it would require 10 to 20 exabytes. So, maybe the exabyte is not unimaginable after all.

7.4.3 Tree Sort

In addition to searching for a key value, binary search trees can also be used to sort a set of values into ascending or descending order. For example, assume we have constructed a BST, such as the following:



If we perform an inorder traversal as shown in Figure 7-9, printing each node as we visit it, we generate the following output:

4, 8, 9, 10, 13, 17, 18, 19

These numbers are the original input values in ascending order.

An algorithm to sort n numbers, called a **tree sort**, is shown in Figure 7-21. It uses instance methods drawn from the binary search tree class in Figure 7-20.

```
// The tree sort algorithm. Assume that you start
// with an empty tree t.

// Phase I. Construct a binary search tree t
// from the n input values
for (int i = 1; i <= n; i++) {
     num = getInput()
     t.insert(num)
}

// Phase II. Traverse the binary search tree using
// an inOrder traversal, printing each node's data
// value as you visit it.
t.inorder()
```

[FIGURE 7-21] The tree sort algorithm

Let's determine the complexity of the tree sort algorithm in Figure 7-21. The insert method in Phase I is called n times, once for each input value. To do the insert operation, we must travel from the root or the tree to the leaf node where we perform the insertion, exactly as pictured in Figure 7-18. We have already shown that this distance, H, is $O(\log_2 n)$ for the average case. Therefore, the time complexity of Phase I, building the n-node binary search tree, is n times the complexity of a single insertion operation, or $O(n \log_2 n)$. The inorder traversal of this tree in Phase II involves visiting every node in the tree exactly once. Because there are n nodes in the tree, the inorder traversal is $O(n)$. We performed this analysis in Section 7.3.2.

In Section 5.3.2, we stated that if an algorithm is in two sections whose complexities are $O(f(n))$ and $O(g(n))$, then the complexity of the entire algorithm is $\max(O(f(n)), O(g(n)))$. That is, the overall complexity of the algorithm is determined by its most time-consuming section. Using this result, we can say that the complexity of the tree sort algorithm in Figure 7-21 is $\max(O(n \log n), O(n)) = O(n \log n)$. This is the same complexity as the merge sort and Quicksort algorithms presented in Section 5.3.1.

However, remember that this analysis applies only to the *average* case behavior. In the worst case, a binary search tree degenerates into a linear structure, and the insert procedure becomes $O(n)$ rather than $O(\log n)$. Because we must repeat this process n times, the time required by the insertion becomes $O(n^2)$, and the complexity of the tree sort is $\max(O(n^2), O(n)) = O(n^2)$, an inefficient quadratic algorithm.

Figure 7-22 shows the performance of the tree sort algorithm on input sequences of length $n = 100{,}000$ in random and reverse order. This is exactly the same performance test we applied to the merge sort and Quicksort in Figure 5-3, which is repeated here for ease of comparison. (We have also included the timing of a sorting method called heap sort, which we introduce in the following section.)

ALGORITHM	n = 100,000 (ALL TIMES IN SECONDS) RANDOM ORDER	REVERSE ORDER
Merge sort	6.7	7.2
Quicksort	5.4	89.5
Tree sort	9.6	101.7
Heap sort	7.5	6.8

[FIGURE 7-22] Performance of the tree sort algorithm

Notice that for numbers in random order, the performance of the tree sort is roughly comparable to that of the other two O(n log n) algorithms we examined—merge sort and Quicksort. However, when the lists are in reverse order, the insert method produces a degenerate tree, and the performance of the algorithm degrades dramatically—it runs more slowly by a factor of about 10. This worst-case behavior is one of the reasons that the tree sort is not as widely used as other sorting algorithms. The tree sort also suffers from memory space problems because it requires storage for two pointers per node.

In Section 7.6, we will introduce one more tree-based sorting method called the heap sort, which does not suffer from the worst-case time problem or excessive memory demands. Thus, the heap sort is a popular sorting algorithm in computer science.

7.5 Balanced Binary Search Trees

7.5.1 Introduction

The binary search tree (BST) of Section 7.4 performs well most of the time. In the previous section, we described how a BST built from a random collection of values has a height H = O(log n). Therefore, the method `contains()` in Figure 7-17, which locates a specific value in a BST, usually runs in logarithmic time, a very efficient performance. However, that is only its *average* case behavior, and we cannot guarantee that our software will always deliver this level of service. Sometimes, the binary search tree we construct will be highly unbalanced, and our search performance will be much worse than expected. In fact, as we proved in the previous section, the worst-case search performance of a BST is O(n).

In some applications, worst-case linear behavior is totally unacceptable. Examples include time-sensitive applications such as **real-time programs**, which must deliver their results within a guaranteed time period, or programs in which the specification requires that the software produce its answers in less than a specified amount of time. In these cases we would be unable to meet the specifications. We could only say that most of the time our software would perform acceptably, but not all the time. If the application was the crash avoidance software of a Boeing 777 jetliner, "most of the time" would be unacceptable. To meet these fixed upper bounds on run time, we must be able to guarantee that our BST is always balanced and always has height H = O(log n). Then we can guarantee that the search process is O(log n) in both the average and worst case.

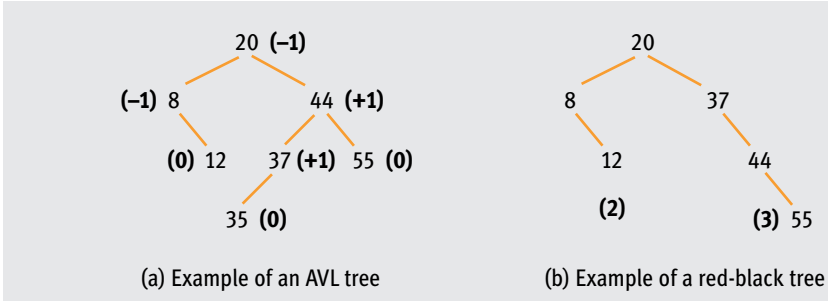
There are a number of types of balanced search trees, including **AVL trees**, **red-black trees**, and **B-trees**. They all work in a similar fashion: They initially insert a new node into the BST, much as we have described. Then they examine the tree that is produced and ask,

"Is the tree still in balance?" If the answer is no, they immediately rebuild the tree using a **rebalancing algorithm**. The differences in the search trees lie in how these rebalancing algorithms work.

The term **balance** is not precisely defined with regard to binary search trees; what is balanced with respect to one type of tree structure may not be balanced in another. For example, in a **B-tree**, which uses the strictest definition of the term *balanced*, every single leaf node must be on the same level. In a binary tree, this requirement is impossible to achieve, except when the number of nodes in the tree is exactly $2^k - 1$ for integer k. Thus, B-trees are not *binary* search trees and allow more than two children per node. (See the Challenge Work Exercise at the end of the chapter for more information about the B-tree.)

In an **AVL tree**, which is a balanced binary search tree, the definition of balance is that for every node x in the search tree, the **difference factor**, defined as (height of the left subtree of x) − (height of the right subtree of x) must be −1, 0, or +1. So, the tree shown in Figure 7-23(a) is a valid AVL tree because the difference factors of each node (the quantity in parenthesis) all have one of these three values.

The **red-black tree** uses a slightly different definition of *balanced*. It says that for any two paths P1 and P2, from the root to a leaf node, the difference in length between the two paths can never be more than a factor of 2. That is, if P1 is the longer of the two paths, then length(P1) ≤ 2 × length(P2). An example of a valid red-black binary search tree is shown in Figure 7-23(b). Notice that the ratio of the difference in path length between the longest path (3) and the shortest path (2) is less than 2.



(a) Example of an AVL tree

(b) Example of a red-black tree

[FIGURE 7-23] Examples of balanced binary search trees

Figure 7-23 demonstrates the differences in the definition of the term *balanced*. The red-black tree of Figure 7-23(b) would not be viewed as balanced if it were an AVL tree, because the difference in height between the left subtree and right subtree of the tree rooted at 37 is −2, an unacceptable value. However, by the definition used to construct a red-black tree, the structure in Figure 7-23(b) is balanced to an acceptable level. Regardless of

which definition we use, though, the height H of both AVL trees and red-black trees is always $H = O(\log n)$. This property allows us to guarantee that all search operations on either tree can be completed in $O(\log n)$ time.

Rather than describe all the balanced binary search trees mentioned previously, we examine only one—the red-black tree of Figure 7-23(b)—in the next section. We have chosen this balanced tree structure because it is used in the Java Collection Framework to provide guaranteed $O(\log n)$ performance. We describe the Java Collection Framework and its use of red-black trees in detail in Chapter 9.

7.5.2 Red-Black Trees

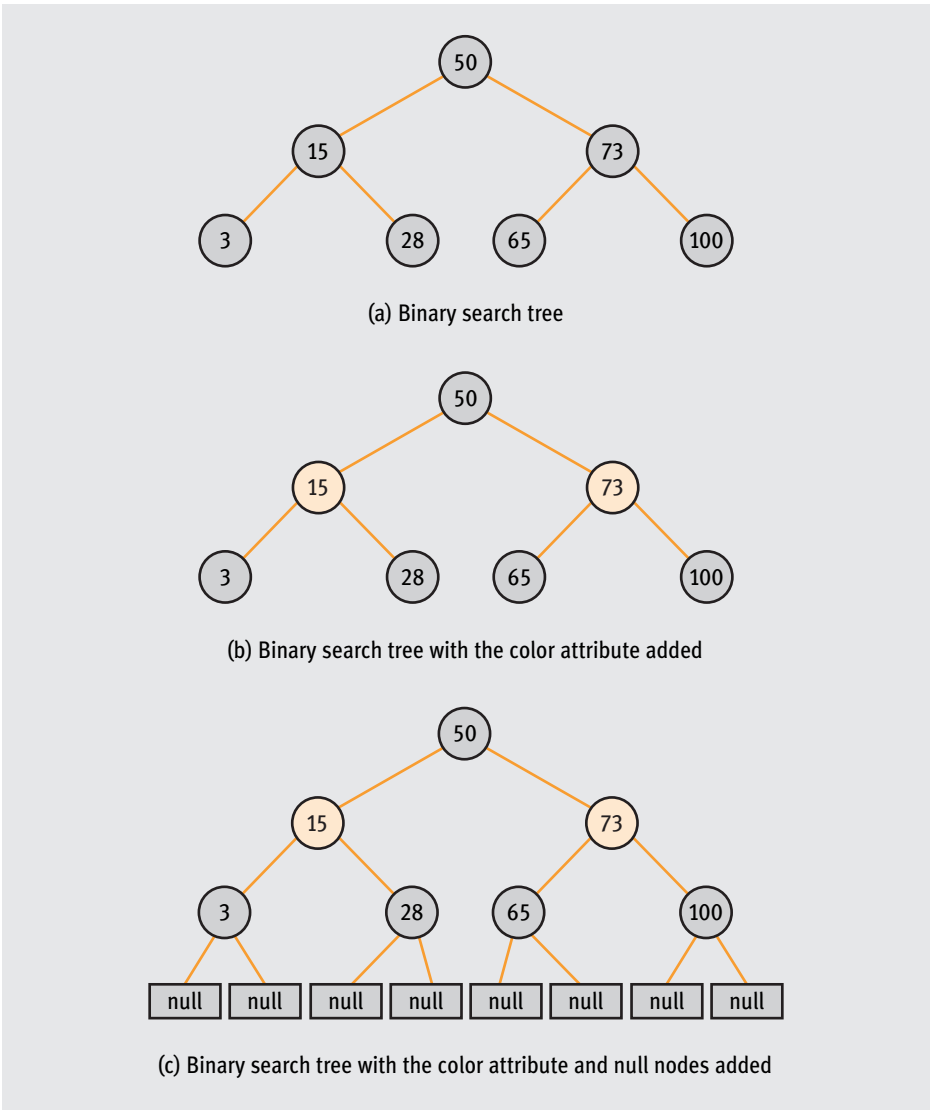
A red-black tree is a binary search tree that is automatically rebalanced whenever we add or remove a node. Because of this rebalancing operation, all searches, insertions, and deletions are guaranteed to run in $O(\log n)$ time, where n is the number of nodes in the tree. The red-black tree is an extremely complex data structure. We will not cover every aspect of its behavior in this section; instead, we take a look at its properties and performance and overview its rebalancing operation. This will allow you to understand the tree's efficiency properties and to determine when it would be the appropriate data structure for a given problem.

Because a red-black tree *is* a binary search tree, it must implement all the BST characteristics described in Section 7.4. To these we add the following two properties:

- Every node in a red-black tree has an associated color attribute that is designated red or black. We see this characteristic in Figure 7-24(b), which takes the BST of Figure 7-24(a) and colors each of the nodes red or black.

Because this book uses a limited color palette, red components in red-black trees are shown in orange and black components are shown in gray.

- The **null** pointers in the tree leaves are explicitly shown as null nodes in the tree diagram. This representation is used to simplify the explanation of the insertion algorithms that follow. It does not change the tree structure. We can see this situation in Figure 7-24(c), which makes the null nodes explicit.



[FIGURE 7-24] Additional attributes of red-black trees

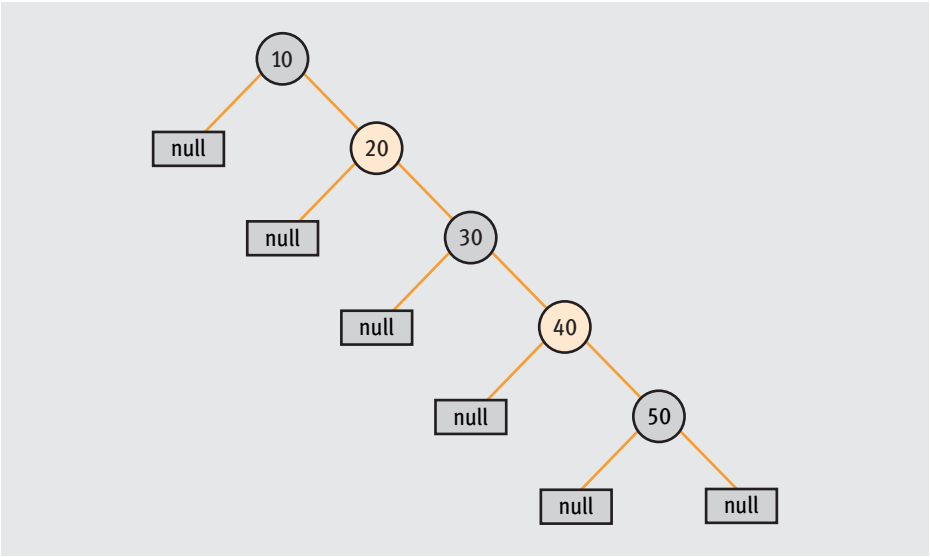
Given these two additional characteristics, we can now formally define a red-black tree. A red-black tree is a binary search tree that has the following five properties:

1. Every node in the tree is red or black.
2. The root of the tree must be black.
3. Every leaf in the tree (in other words, every null node) must be black.
4. Every red node must have two children, and both must be black.
5. Every path from a node x to a leaf must contain the exact same number of black nodes.

We can observe these properties in the tree of Figure 7-24(c). Every node in the tree is colored, and the root is black. The eight leaves, which are all null nodes, are also black. The two red nodes, 15 and 73, both have two black children. Finally, if we take any node in the tree, say the root 50, we see that the paths from it to its eight leaves each contain exactly three black nodes. This property holds for every other node as well. For example, from the red node 15, the paths to the four leaf nodes in its subtree each contain exactly two black nodes. Thus, the tree in Figure 7-24(c) is a red-black tree.

The previous five properties guarantee that a red-black tree will be approximately in balance. We can prove this as follows. Every path from the root to a leaf must contain the same number of black nodes (property 5). Let's call that value n. The shortest possible path from the root to a leaf will therefore have n black nodes and 0 red nodes, for a total length of n. Because every red node must have two black children (property 4), no path in the tree can have two consecutive red nodes. Thus, the longest path from the root to a leaf must alternate between red and black. Because that longest path must contain n black nodes and because both the root and the leaf must be black (properties 2 and 3), the longest path must start with a black node, end with a black node, and alternate between red and black in between. Thus, it can contain at most $(n - 1)$ red nodes, and the total length of the longest path can be no greater than $(2n - 1)$. This shows that the shortest path and the longest path from the root to a leaf cannot differ by more than a factor of 2, and a red-black tree will be in balance within a factor of 2.

Furthermore, the properties of red-black trees will not allow them to become degenerate. In Figure 7-25, the numbers of black nodes on the paths from the root to the six null nodes in the degenerate tree are {2, 2, 3, 3, 4, 4}. This violates property 5, which specifies that the number of black nodes be the same for every path. There is no possible way to color the nodes in Figure 7-25 to make this happen. Although we offer no proof here, it can be shown that the height H of any red-black tree is $H \leq 2 \log_2(n + 1)$, where n is the number of nodes. Thus, $H = O(\log n)$, and any operation that is $O(H)$, such as searching, inserting, or deleting, is guaranteed to run in logarithmic time.



[FIGURE 7-25] Degenerate binary search tree

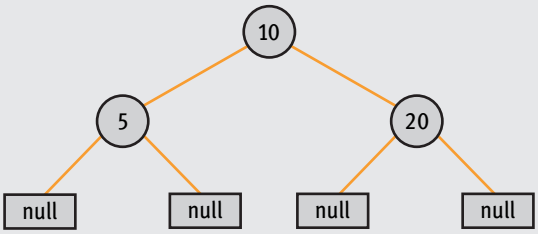
The problem we must now address is ensuring that all the properties of a red-black tree are maintained following an insertion that can change the tree's shape and structure.

If the new node being added is the root of the entire tree, insertion is trivial. Just color the node black (because the root must be black), and we are finished. Therefore, let's now assume that the node is not being added as the root.

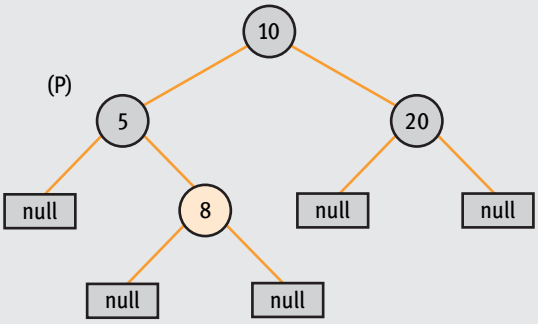
Because a red-black tree is a binary search tree, the addition of a new node N to a non-empty tree always occurs at a leaf, which is a null node in our model. (You can see this in the BST insertion diagram of Figure 7-18.) Therefore, any insertion will replace a black null node in the tree with a new data node N, which itself will have two null black children. To determine what we need to do, we must examine the current color of up to three other nodes in the tree related to the new node N. We need to look at the parent of N, which we call P; the grandparent of N (the parent of P), which we designate G; and the sibling of P, which we denote U, for the uncle of N. Our actions depend on the current color of the three nodes P, G, and U. The three possible actions are: (1) do nothing; (2) recolor one or more of the nodes P, G, and U; and (3) rotate the position of the nodes P, G, and U within the tree. Sometimes, we only need to recolor one or more nodes; at other times we may need to both recolor and rotate.

Let's take a look at some examples. Because insertion will replace a black null node, we cannot color the new node N black without violating the rule that we must have the same number of black nodes on every path to a leaf. This is because we have replaced a black null node with a new black node N and two new black null children, increasing the black path length by 1 to these new leaves. Therefore, the new node must be red. If the parent P

of the newly inserted node is black, then the insertion operation is trivial and nothing needs to be done. We do not have two consecutive red nodes, and the path length to the new black leaves will be the same as that to the existing leaves because we replaced a black null node with a new red node and two black null leaves. We see this condition in Figure 7-26(b), where we have added the value 8 to the tree of Figure 7-26(a). The number of black nodes on the path to all of the leaves is 3 for every path in the tree. This operation can be completed in O(1) time.



(a) Original red-black tree

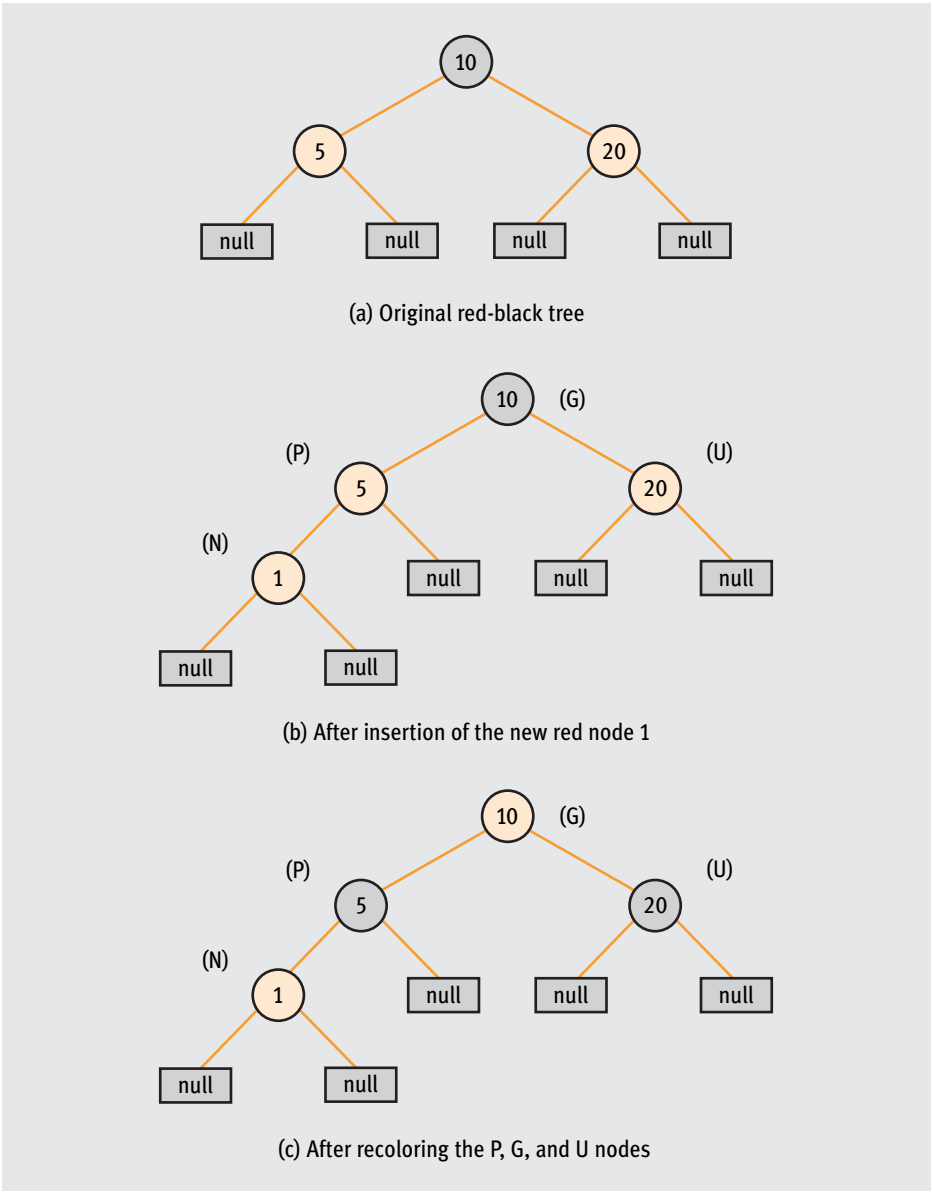


(b) After insertion of the value 8

[FIGURE 7-26] Insertion of a node where *P* is black

However, what do we do if the red node *N* is added to a red parent *P*, as shown in Figure 7-27(b), which added the new value 1 to the red-black tree of Figure 7-27(a)? The answer depends on the color of the uncle node *U*. If that node is also red [as in Figure 7-27(b)], we can repair the violation using only recoloring operations. We simply recolor both *P* and *U* to be black, and recolor the grandparent *G* red. (It must originally have been black because we cannot have two consecutive red nodes.) This guarantees that all paths from *G* to its leaves have the same number of black nodes, because we have replaced a black *G* node and a black null node [Figure 7-27(a)] with either a black *P* node and a black null node or a black *U* node

and a black null node. The total number of black nodes on all paths remains the same—2 in the case of the tree in Figure 7-27(c).



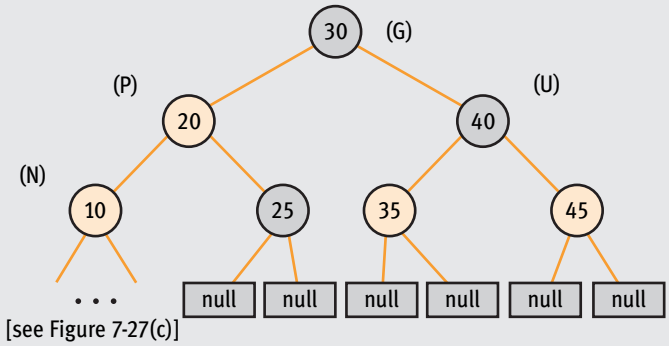
[FIGURE 7-27] Insertion of a node where both *P* and *U* are red

However, it is possible that in recoloring node G, we violated one of the other rules for red-black trees. For example, G might be the root of the entire tree, and so it must be black, not red. We can solve that problem easily by recoloring G black. Because G is on every path from the root to a leaf, it simply adds one black node to every path, which keeps the total number of black nodes on all paths from G to its leaves the same.

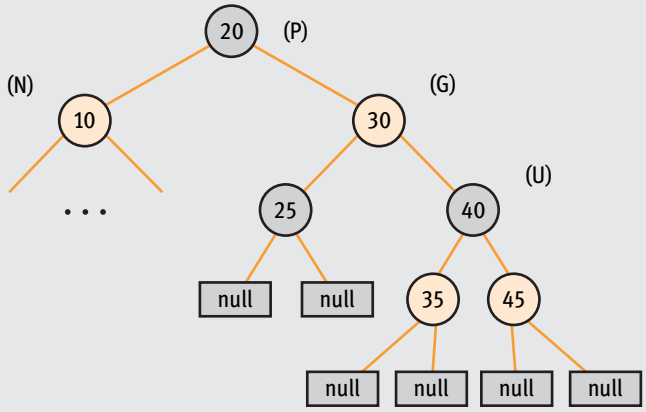
Recoloring, though, cannot solve all the problems we might encounter when rebuilding a red-black tree. For example, it is possible that node G may not be the root of the entire tree. If the parent of node G in Figure 7-27(c) is red, we will have two consecutive red nodes. We can solve this problem by repeating the rebalancing algorithm but this time assuming that G, rather than N, is the newly added node. That is, set $G = N$ and recursively execute the entire algorithm.

This situation is diagrammed in Figure 7-28(a), where the node 10 that had been labeled G in Figure 7-27(c) has been relabeled N. We again have the problem of two consecutive red nodes. The parent P (20) is red, but this time the uncle node U (40) is black, so the recoloring approach shown in Figure 7-27(c) does not work. Because recoloring alone is not enough to rectify this violation, we must do a rotation. In a rotation operation, the actual position of the P, U, and/or G nodes changes within the tree. There are a number of different rotations, including both single and double rotations (move up or down one or two levels) and right and left rotations (move into the right or left subtree).

In this example, we can solve the problem using a single right rotation. We take the parent node P and move it up one level to become the root of the subtree. Both the G and the U nodes are moved down into the right subtree to become the right child and the far-right grandchild of P. We also must flip the colors of both P and G, making P black and G red [see Figure 7-28(b)]. We have eliminated the problem of having consecutive red nodes. We can also see that the number of black nodes on every path of Figure 7-28(b) has remained the same as in Figure 7-28(a). In Figure 7-28(a), every path from the root went through G, a black node. In Figure 7-28(b), every path through the root goes through P, also a black node. The paths to the subtrees rooted at 25, 35, and 45 previously had two black nodes, and they still do. The red-black tree is still a red-black tree after we carry out the single right rotation.



(a) Violation caused by the recoloring in Figure 7-27(c)



(b) Doing a single right rotation on G to rebuild the tree

[FIGURE 7-28] Using a single right rotation to rebuild the red-black tree

The maximum number of times we recolor and rotate in Figures 7-26, 7-27, and 7-28 depends on the height of the tree. We can demonstrate this point by noting that we relabeled the grandparent G as the new node N and recursively executed the rebalancing algorithm. The node G is two levels closer to the root than N, and when we reach the root we stop the recursion. (Remember, when we add a node as the root, we color it black and stop.) So, the maximum number of times we would have to recursively execute the rebalancing algorithm is $H/2$. Because we have already shown that the height of a red-black tree is $H = O(\log n)$, we can conclude that insertion into a red-black tree can be completed in $O(\log n)$ time.

The preceding arguments certainly do not constitute a proof of the logarithmic complexity of red-black tree insertion. Twelve additional cases must be described and analyzed before we can state the result with any formal certainty. However, our goal in this section was not to work through the details of every possible case and explain the steps required for each one. That would be a long and complex presentation, which would be unnecessary in most situations. As we mentioned at the beginning of the section, red-black trees are part of the Java Collection Framework, and the code required to implement the rebalancing and recoloring operations described in Figures 7-26, 7-27, and 7-28 have already been written.

Instead, our purpose was to introduce you to the concept of a balanced binary search tree and help you understand the work involved in rebalancing the tree after an insertion operation. You need this information to appreciate the efficiency gains that come from a balanced tree and to make intelligent and informed decisions about what data structure is best for a given problem.

In a modern software development environment such as Java 1.5, users often do not need to design and implement their own data structures; instead, they need to be able to understand, analyze, and select from a library the data structures they use to solve a given problem.

7.6 Heaps

7.6.1 Definition

The heap data structure is another example of a balanced binary tree. It is particularly useful in solving three types of problems:

- Finding a minimum or maximum value within a collection of scalar values
- Sorting numerical values into ascending or descending order
- Implementing another important data structure called a priority queue, as introduced in Section 6.4.4

This section shows examples of all three of these important applications.

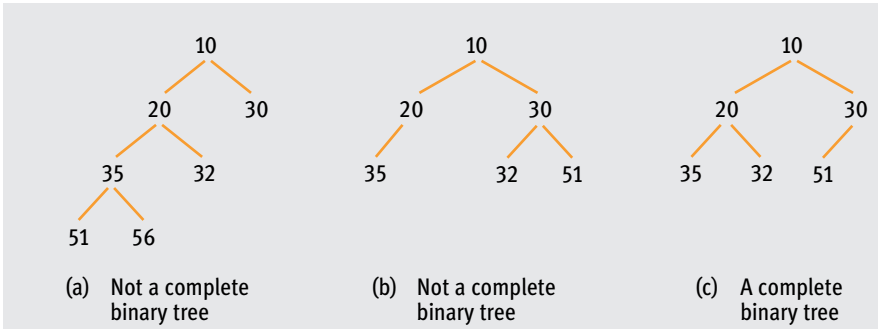
A **heap** is a binary tree that satisfies the following two conditions:

- The data value stored in a node is less than or equal to the data values stored in all of that node's descendants. Unlike the binary search tree of Section 7.4, no restrictions are placed on the ordering of values within left or right subtrees, but only between a parent and its children. Therefore, the value stored in the root is always the smallest value in the heap. This is called the **order property** of heaps.

We could just as easily define a heap in which a node's value is greater than *or equal to the data values stored in all of that node's descendants. In this case, all algorithms would simply change the* < *operator to a* >*, and every occurrence of the word smallest would be replaced by largest).*

- A heap is a **complete binary tree**, a binary tree of height i in which all leaf nodes are located on level i or level $i - 1$, and all the leaves on level i are as far to the left as possible. This is called the **structure property** of heaps.

Figure 7-29 shows examples of complete binary trees as well as those that violate the preceding constraints.



[FIGURE 7-29] Examples of valid and invalid complete binary trees

The tree in Figure 7-29(a) is not a complete binary tree because the leaves occur on levels 2, 3, and 4 rather than just levels 3 and 4. Figure 7-29(b) is not a complete binary tree because the leaves on level 3 have not been placed as far to the left as possible. (Notice that the right child of a node—the one that contains a 20—is empty.) However, the tree in Figure 7-29(c) is a valid heap because it satisfies both the order and the structure properties. Informally, we can say that you produce a valid heap structure when you insert new nodes by moving across level i in a strictly left-to-right fashion until it is full, and only then begin to insert new nodes on level $i + 1$, again strictly from left to right.

The two most important mutator methods on heaps are: (1) inserting a new value into the heap and (2) retrieving the smallest value from the heap (in other words, removing the root).

The `insertHeapNode()` method adds a new data value to the heap. It must ensure that the insertion maintains both the order and structure properties of the heap. The retrieval method, `getSmallest()`, removes and returns the smallest value in the heap, which must be the value stored in the root. This method also rebuilds the heap because it removes the root, and all nonempty trees must have a root by definition. (We will learn how

to carry out these operations in the next section.) The other important operation on heaps is the Boolean method `empty()`, which returns true if the heap is empty and false otherwise.

An interface for a heap data structure is shown in Figure 7-30. It includes the three basic operations just described—`insertHeapNode()`, `getSmallest()`, and `empty()`. The generic type for the class extends the `Comparable` interface, which guarantees that the parameter to `insertHeapNode()` and the return value for `getSmallest()` provide a `compareTo()` method. This method can determine whether the value stored in a node is smaller than that stored in the other nodes of the heap.

```
/**
 * An interface for a heap data structure
 */
public interface Heap<T extends Comparable> {
    /**
     * Adds the given information to the heap.
     *
     * Preconditions:
     *   Data is not null.
     *
     * Postconditions:
     *   Data has been added to the heap.
     *   The heap property has been preserved.
     *
     * @param data the information to be added
     */
    public void insertHeapNode( T info );

    /**
     * Remove and return the smallest element in the heap
     *
     * Preconditions:
     *   The heap is not empty.
     *
     * Postconditions:
     *   The element has been removed.
     *
     * @return the smallest value in the heap
     */
    public T getSmallest();

    /**
     * Determine whether the heap is empty.
     *
     * Preconditions:
     *   None
     *
```

continued

```
     * Postconditions:
     *   The heap is unchanged.
     *
     * @return true if the heap is empty and false otherwise
     */
    public boolean empty();

} // Heap
```

[FIGURE 7-30] Interface for a heap data structure

7.6.2 Implementation of Heaps Using One-Dimensional Arrays

One problem with the tree structures described in this chapter is the amount of memory space required for reference fields. A reference is simply a memory address, and it typically requires four bytes, sometimes more. Given that all binary tree nodes need two reference fields, the total amount of required memory can become rather large.

However, we can take advantage of a heap's restricted structure to produce an extremely efficient one-dimensional array that requires no reference fields at all. We can store the elements of our heap in a one-dimensional array in strict left-to-right, **level order**. That is, we store all of the nodes on level i from left to right before storing the nodes on level $i + 1$. This one-dimensional array representation of a heap is called a **heapform**, as diagrammed in Figure 7-31(b).

(a) Heap represented as a tree

(b) Heap represented as an array

[FIGURE 7-31] A tree and a one-dimensional array representation of a heap

We do not need pointers in this array-based representation because the parent, children, and siblings of a given node must be placed into array locations that can be determined with some simple calculations.

For example, the root of the heap X_0 is in location 0 of the array, as shown in Figure 7-31(b), and its left child, X_1, is in location 1. The left child of X_1 is X_3, which is in location 3. Similarly, the left child of X_2, in location 2, is X_5, in location 5:

Node location	Left child location
0	1
1	3
2	5

Simple reasoning should convince you that if a node is in location i of the heapform array, the left child of that node, if it exists, must be in location $2i + 1$. The reason for this is the restriction on where new nodes can be placed in a heap—only in the far-left available slot of level i or the far-left position of level $i + 1$ if level i is full.

Similar expressions exist for all the other important relationships in a binary tree. For a node stored in array location i, $0 \leq i < n$, where n is the total number of nodes in the heap, the locations of the parent, left child, right child, and sibling of that node are given by the following expressions:

`Parent(i)` = **int** $((i - 1)/2)$	if $(i > 0)$, else i has no parent
`LeftChild`$(i) = 2i + 1$	if $(2i + 1) < n$ else i has no left child
`RightChild` $(i) = 2i + 2$	if $(2i + 2) < n$ else i has no right child
`Sibling (i)` =	
if odd(i) then $i + 1$	if $i < n$ else i has no sibling
if even (i) then $i - 1$	if $i > 0$ else i has no sibling

Using these formulas, we can recover all the hierarchical relationships in the original tree representation of a heap shown in Figure 7-31(a).

For example, look at the node labeled X_2 in Figure 7-31(a). It is stored in slot 2 of the array. We can reconstruct all of the original tree-based relationships shown in Figure 7-31(a) using the previous formulas:

`Parent` = **int** $(((2 - 1) / 2)) = 0$	The node in location 0 is the parent of X_2.
`LeftChild` $= 2 \times 2 + 1 = 5$	The node in location 5 is the left child of X_2.
`RightChild` $= 2 \times 2 + 2 = 6$	Because 6 is not less than n, X_2 has no right child.
`Sibling` $= 2 - 1 = 1$	The node in location 1 is the sibling of X_2.

As a second example, take a look at node X_5. It is stored in slot 5 of the array. We can locate its parent, children, and siblings as follows:

`Parent` = **int** $(((5 - 1) / 2)) = 2$	The node X_2 is the parent of X_5.
`LeftChild` $= 2 \times 5 + 1 = 11$	Because $11 > 6$, we know that node X_5 has no left child.

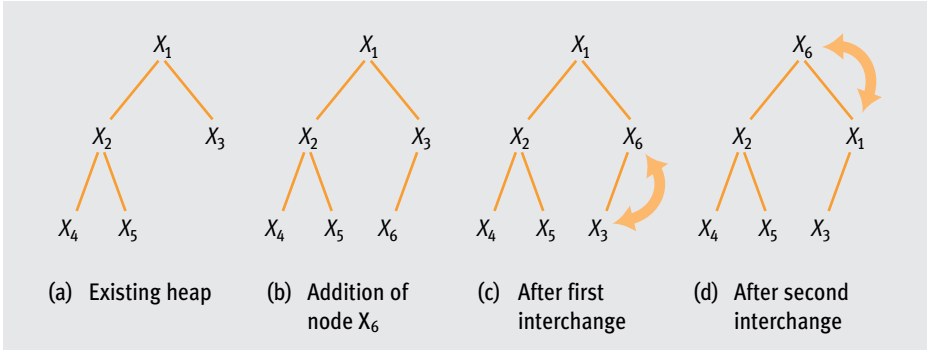
`RightChild` $= 2 \times 5 + 2 = 12$	Because $12 > 6$, we know that node X_5 has no right child.
`Sibling` $= 5 + 1 = 6$	Because 6 is not less than n, node X_5 has no sibling.

Assume that we use the following declarations to create a heapform array h:

```
private static final int INITIAL_SIZE = 100;
private T theHeap[] = (T[]) new Comparable[INITIAL_SIZE];
private int size = 0;
```

We now can describe the implementation of the two basic mutator methods on heaps—`insertHeapNode()` and `getSmallest()`.

To insert a new value into the heap `h`, initially place the value in the unique location that maintains the structure property of heaps. This is either the far-left unoccupied slot on level i or the far-left slot on level $(i + 1)$ if level i is full. In either case, this location corresponds to element `h[size]` in the heapform array, where `size` is the number of nodes stored in the heap prior to the insertion operation. This situation is shown in Figures 7-32(a) and 7-32b, which start with a five-element heap (`size` $= 5$) and add a sixth item.

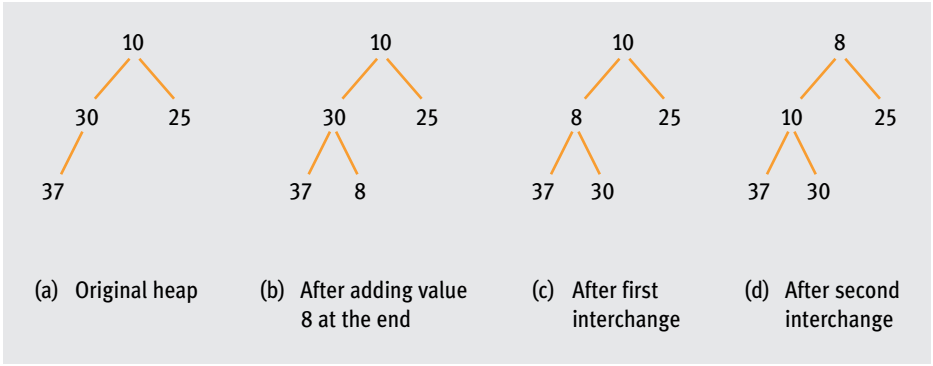


[FIGURE 7-32] The insert operation on heaps

However, in Figure 7-32(b), it may be true that $X_6 < X_3$, in which case the placement of this node violates the order property of heaps. If so, we must interchange the child and parent nodes that are out of order, as shown in Figure 7-32(c). We now know that X_3 and X_6 are in their proper order, but X_1 and X_6 may not be. If not, then repeat the interchange of child and parent nodes one more time [Figure 7-32(d)]. Continue this process until we either find the correct location for the new value or we reach the root (as we did here). In Figure 7-32(d), we know that X_6 and X_1 are in the correct order because they were explicitly compared to each other, but are X_6 and its left child X_2 in the proper order? We know that

$X_1 < X_2$ because we assumed that Figure 7-32(a) was a valid heap and X_1 was the root, which means that its data value is less than every other node. We also know that $X_6 < X_1$, as we just mentioned. Because $X_6 < X_1$ and $X_1 < X_2$, the commutive property of the $<$ operator proves that $X_6 < X_2$ and demonstrates that the swapping operation just described correctly restores the order property of heaps.

Figure 7-33 shows a specific numeric example of this insertion operation. Here we insert the new value 8 into the heap structure. After two interchanges, we have restored the order property, and the tree in Figure 7-33(b) is a valid heap.



[FIGURE 7-33] Example of a heap insertion

The code for the `insertHeapNode()` method is shown in Figure 7-34. Note that if we attempt to insert a new node and the heap is full, then we dynamically enlarge the heap-form array via a call to the method `expandHeap()`.

```
public void insertHeapNode( T data ) {
    // Is there room in the heap for another element?
    if ( size == heapForm.length ) {
        // No - resize the heap
        expandHeap();
    }

    heapForm[ size ] = data;  // Put new data into the heap
    size = size + 1;          // One more element in the heap

    interchangeUp();          // Ensure that the ordering property
                              // of the heap holds after the item
                              // has been inserted
}
```

continued

```
/**
 * Ensure that the ordering property of the heap holds after
 * the insertion of a new element into the heap.
 */
private void interchangeUp() {
    int cur = size - 1;            // Location of last item added
    int parent = ( cur - 1 ) / 2;  // Parent of the new item

    // We only need to check heaps with more than 1 element
    if ( size > 1 ) {
        // Walk up the heap until you reach the top or stop finding
        // values that are out of place
        while ( parent >= 0 &&
                heapForm[ cur ].compareTo( heapForm[ parent ] ) < 0 ) {
            // Swap the parent and child values
            swap( parent, cur );

            // Move up one level in the heap
            cur = parent;
            parent = ( cur - 1 ) / 2;
        }
    }
}

/**
 * Swap two elements in the heap.
 *
 * @param pos1 the position of the first element.
 * @param pos2 the position of the second element.
 */
private void swap( int pos1, int pos2 ) {
    T temp = heapForm[ pos1 ];
    heapForm[ pos1 ] = heapForm[ pos2 ];
    heapForm[ pos2 ] = temp;
}

/**
 * Double the size of the array used to hold the heap.
 */
private void expandHeap() {
    // Create a bigger array to hold the heap

    // Note that the cast is necessary because you cannot
    // create generic arrays in Java. This statement will
    // generate a compiler warning.
    T newHeap[] = (T[]) new Comparable[ heapForm.length * 2 ];
```

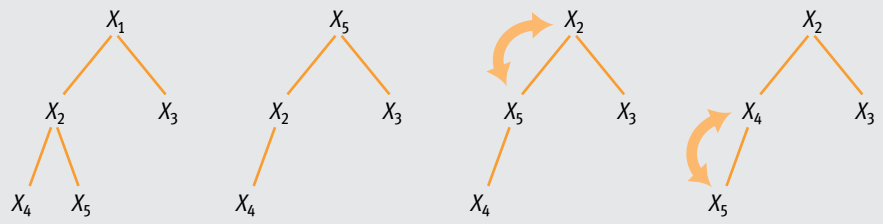
continued

```
    // Copy the elements over from the existing heap
    for ( int i = 0; i < size; i = i + 1 ) {
        newHeap[ i ] = heapForm[ i ];
    }

    // Use the bigger heap
    heapForm = newHeap;
}
```

[FIGURE 7-34] The `insertHeapNode` method

The code in Figure 7-34 shows that, in the worst case, we may need to exchange node pairs starting at a leaf and traveling all the way up to the root. A heap is a balanced tree by definition, because the structure property keeps all leaf nodes on two levels at most. Therefore, its height is O(log n), where n is the number of nodes, and the maximum number of times we must perform the exchange operation is O(log n). Because the time for a single exchange is a constant, O(log n) is the overall complexity of the `insertHeapNode` method in Figure 7-34.

The `getSmallest()` method works in a similar fashion. We first remove the smallest element from the heap, which by definition is the value stored in the root. To reconstruct the tree, we need to move a new node into the root position; the only node we can move and still maintain the structure property of the heap is the "last" one (the far-right node on the lowest level i). When this value is moved from its current position in the array into the root position, the heap now contains (`size` − 1) elements rather than `size`. This process is diagrammed in Figures 7-35(a) and 7-35(b), which show the removal of the smallest element from a five-element heap and the shifting of the last element into the root position.



(a) Existing heap

(b) Removal of root and shifting of last element into the root position

(c) After one interchange with its smaller child

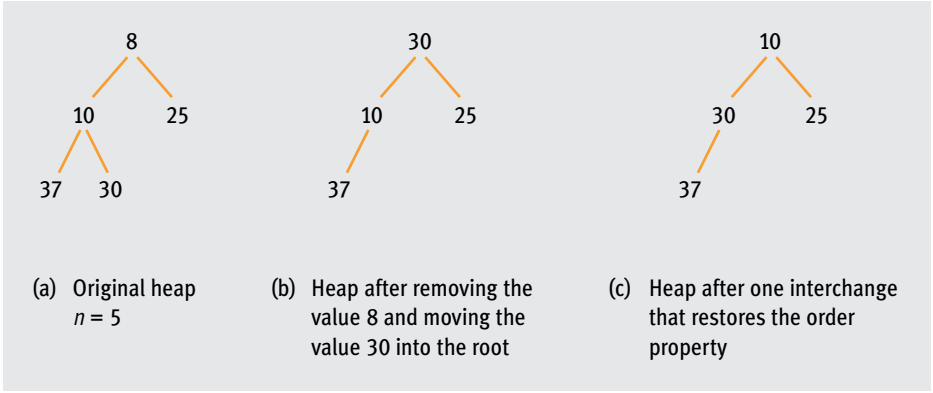
(d) After the second interchange

[FIGURE 7-35] The `getSmallest()` method on heaps

As with insertion, we must now determine the correct location for the value that was moved into the root—namely, X_5 in Figure 7-35(b). If $X_5 < X_2$ and $X_5 < X_3$, then this value is in the correct location. If not, we interchange X_5 with the smaller of its one or two children, as shown in Figure 7-35(c). (This diagram assumes that $X_2 < X_3$.) We repeat this downward interchange process until we either find the correct location for X_5 or reach a leaf node. This algorithm restores the order property of the heap.

The maximum number of times we must exchange a node with its smaller child is equal to the height of the heap. We have already shown that the height is O(log n), where n is the number of elements. Because the time for a single exchange is constant, the time complexity of the `getSmallest` operation in Figure 7-35 is O(log n).

Figure 7-36 is another example of the `getSmallest()` method. It diagrams the removal of the smallest value from the heap in Figure 7-33(d) and the rebuilding of the structure. In this example, the rebuilding requires only a single exchange.



(a) Original heap $n = 5$

(b) Heap after removing the value 8 and moving the value 30 into the root

(c) Heap after one interchange that restores the order property

[FIGURE 7-36] Example of the `getSmallest()` method

The code for the `getSmallest()` method is shown in Figure 7-37.

```
public T getSmallest() {
    // The smallest element is always the root
    T smallest = heapForm[ 0 ];

    // One less time in the heap
    size = size - 1;

    // Replace the smallest (root) node with the last node
    heapForm[ 0 ] = heapForm[ size ];
    heapForm[ size ] = null;
```

continued

```
    // Ensure that the ordering property holds
    interchangeDown();

    return smallest;
}

/**
 * Make sure the ordering property holds after removing the
 * smallest element from the heap.
 */
private void interchangeDown() {
    int parent = 0;        // Parent position
    int left;              // Left child
    int right;             // Right child

    int minPos;            // Position of the smallest child
    T minValue;            // Value of the smallest child

    boolean continueScan;  // When true, heap is ordered

    // Only need to look if the heap size is greater than 1
    if ( size > 1 ) {
        do {
            left = parent * 2 + 1;
            right = parent * 2 + 2;
            continueScan = false;
            if ( left < size ) {
                // I have at least one child
                if ( right < size ) {
                    // I have two children, which is the minimum?
                    if (
                      heapForm[left].compareTo(heapForm[right]) < 0 ) {
                        // Left is the smallest
                        minValue = heapForm[ left ];
                        minPos = left;
                    }
                    else {
                        // Right is the smallest
                        minValue = heapForm[ right ];
                        minPos = right;
                    }

                    // If the parent is larger than the smallest child,
                    // swap them and continue scan
```

continued

```
                    if ( heapForm[parent].compareTo(minValue) > 0 ) {
                        swap( parent, minPos );
                        parent = minPos;
                        continueScan = true;
                    }
                }
                else {
                    // Only one child (must be the smaller). Is the
                    // parent larger than the left child?
                    if (
                      heapForm[parent].compareTo(heapForm[left]) > 0) {
                        // Yes, swap them
                        swap( parent, left );
                        parent = left;
                        continueScan = true;
                    }
                }
            }
        } while ( continueScan );
    }
}
```

[FIGURE 7-37] The `getSmallest()` method

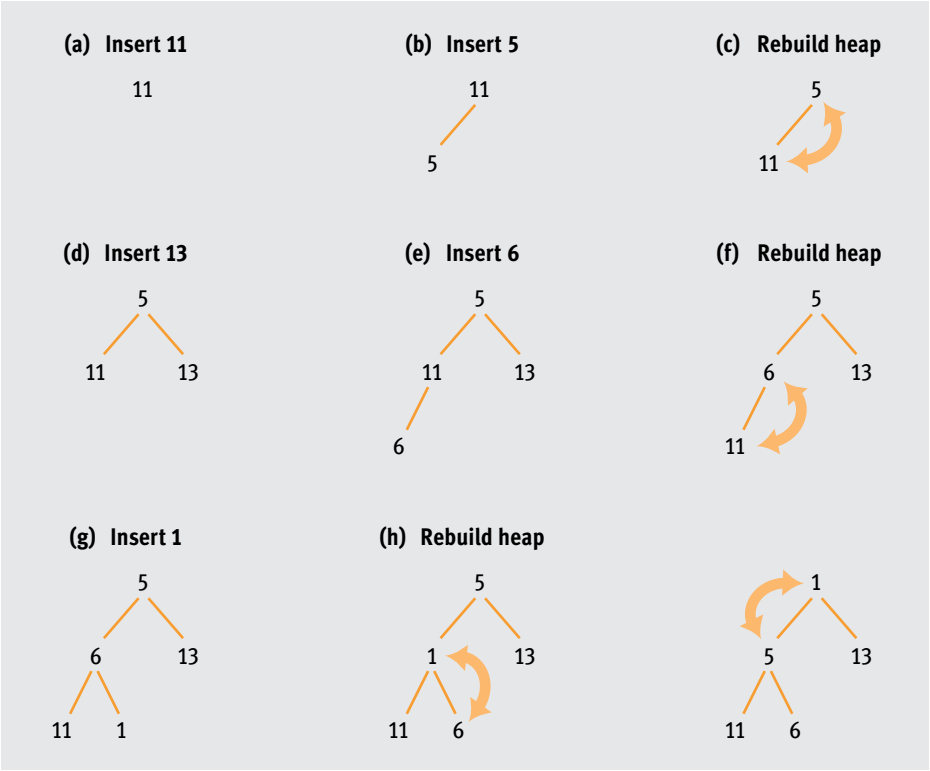
7.6.3 Application of Heaps

One of the most important uses of the heap data structure is as the foundation for a popular sorting algorithm known as the **heap sort**.

Assume we are given a random sequence of n values $\{i_1, i_2, \ldots, i_n\}$ that we want to sort in ascending order. The heap sort performs this task in two phases: the *building phase* and the *removing phase*. During the building phase, we build a heap structure that contains the n elements to be sorted. We start with an empty heap and insert the elements from the sequence into the heap, one at a time, using the `insertHeapNode()` method in Figure 7-34. This phase can be summarized as follows:

```
// Phase I. The building phase of heap sort in which
// we create a heap h from the n numbers to be sorted

for (int k = 0; k < N; k++) {   // Sort n numbers
    num = getInput(num);        // the next number
                                // in the sequence
    h.insertHeapNode(num);      // and insert into
                                // our heap h
}
```

This heap-building phase is illustrated in Figure 7-38 using the five-element set of integer values {11, 5, 13, 6, 1}. (Although we show the heap as a tree for clarity, the values are stored internally as a one-dimensional heapform array, as described in the previous section.)

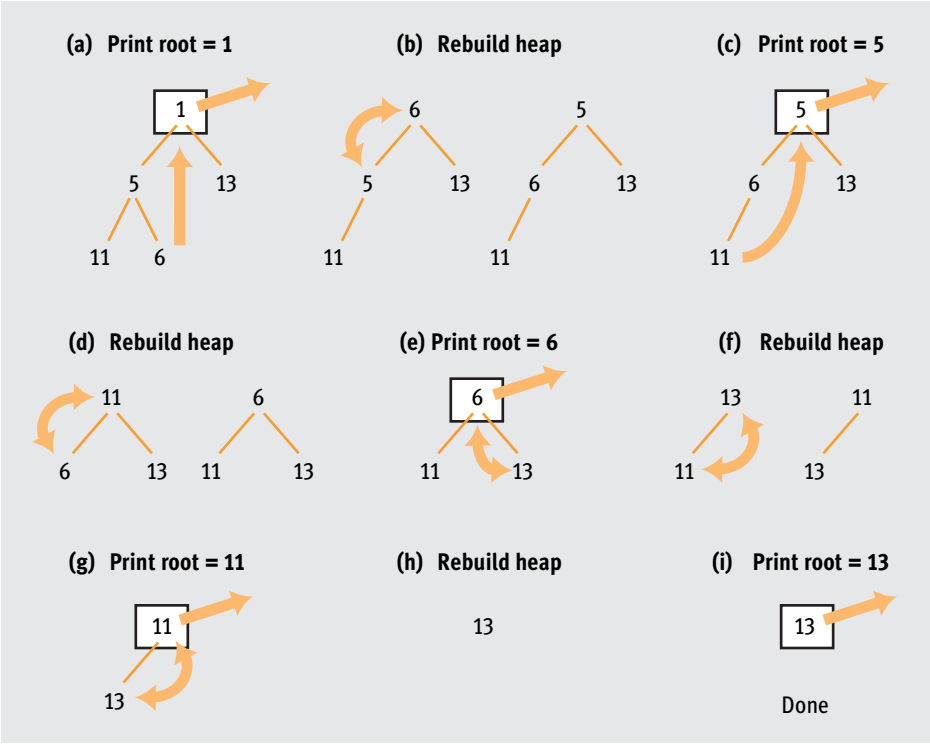


[FIGURE 7-38] The building phase of a heap sort

After we have built the heap, it becomes simple to obtain the elements in sorted order. We simply remove the root, which by definition is the smallest value, print it, and rebuild the heap, which now contains one less item. These operations are identical to the `getSmallest()` method in Figure 7-37. We can summarize the removal phase of the heap sort as follows:

```
// Phase II. The removal phase of heap sort. We
// remove the elements in the heap, one at a time
// This will produce the values in ascending order

for (int k = N; k > 0; k--)   {
    smallest = h.getSmallest();
    output(smallest);
}
```

This process of removing the smallest item and rebuilding the heap is diagrammed in Figure 7-39. It uses the heap built in Figure 7-38.



[FIGURE 7-39] The removal phase of a heap sort

Both `insertHeapNode()` and `getSmallest()` methods are O(log n). These method calls are encased inside loops that are executed n times. Therefore, both phases of this algorithm—building and removing—are O(n log n), and the complexity of the heap sort is O(n log n). Because a heap is a balanced tree by definition, this behavior is achieved in both the average and the worst case. Therefore, heap sort, like merge sort, is an excellent sorting algorithm for applications in which we must guarantee efficient performance under all conditions.

We measured the running time of the heap sort using lists of 100,000 values in both random and reverse order, which was exactly the same test we ran on our other O(n log n) sorting techniques—merge sort, Quicksort, and tree sort. The results for all four algorithms were summarized in Figure 7-22. Note that the performance of the tree sort and Quicksort degraded when presented with ordered data, but the performance of the merge sort and heap sort stayed approximately the same. This property, as well as its memory efficiency, makes heap sort a popular sorting algorithm in programming libraries.

The final application to discuss is the use of a heap to implement the priority queue data structure introduced in Section 6.4.4. Remember that in a traditional queue structure, all objects are kept in time-ordered sequence. In a sense, you can think of time as the priority mechanism—the earlier you arrive, the higher your priority, and the closer you are to the front of the line. With a priority queue, we are not restricted to using time as the priority field. Instead, we allow the user to specify the value of the priority field, typically by modifying the calling sequence of the `enqueue` method to include a priority value `p`.

```
pq.enqueue (o, p);        // Place object o in the priority
                          // queue with priority p
```

Next, we modify the behavior of both the `front` and `dequeue` methods as follows, assuming that `pq` is a priority queue object:

```
o = pq.front();  // Return the item o in the priority
                 // queue with the highest priority.
pq.dequeue();    // Remove the item that has the
                 // highest priority
```

Priority queues are an important data structure because they model situations that occur frequently in computer systems. In many designs, time alone is insufficient to obtain optimal behavior for a system. Instead, we need the ability to prioritize requests to indicate that some are more important and must be serviced before others, regardless of the time order in which they arrived.

For example, a process that controls the setting of wing or tail flaps on a commercial airliner cannot be delayed for any significant amount of time. It must be given a processor as soon as one becomes available, because it is obviously more important than processes that control the showing of in-flight movies or that regulate air conditioning in the cabin. A simple way to implement this is to assign priority values to the different types of operations. For example:

Request type	**Priority level**
Start an in-flight movie	4 (lowest)
Adjust oxygen/pressurization software	3
Adjust wing/tail control software	2
Invoke crash avoidance program	1 (highest)

All incoming requests from the plane's sensors are kept in a priority queue rather than a regular queue. When we retrieve the next item to carry out, via a `front()` command, we are not given the next request in line, as in a regular queue. Instead, we are given a priority 1 process before any others, if one exists. If there are no priority 1 requests, we are given

requests at priority level 2. If there are none, we move to a priority 3 process. Only after the higher-priority jobs are serviced do we get to priority level 4 and begin showing the in-flight movie.

We need to select a data structure that allows us to efficiently implement the `enqueue`, `front`, and `dequeue` operations on priority queues. If we select a linked list, as described in Chapter 6, we have two possibilities:

- Keep the list sorted in order of priority. To accomplish this, the `enqueue` operation must traverse the list to find the correct location. Now, the `front` and `dequeue` operations are simplified because they only have to return the first element in line, which is the highest-priority item. Thus, if we choose to keep the elements in sorted order, we have the following complexities:

```
enqueue = O(N)      front, dequeue = O(1)
```

- Instead, we could simply place each new request at the end of the list, regardless of its priority. However, when we need to locate the highest-priority element, we have to search the entire list for the highest-priority value. Thus, if we keep the elements in unsorted order, we have the following complexities, which are exactly the reverse of the previous complexities:

```
enqueue = O(1)      front, dequeue = O(N)
```

In both cases, we obtain efficient O(1) behavior for one operation at the expense of spending O(n) time on another.

However, there is a third possibility—implementing our priority queue as a heap. We implement `enqueue()` as the heap insertion operation `insertHeapNode()` using the priority field p as the key field of the heap. As shown in the previous section, this takes O(log n) time. Our `front()` and `dequeue()` methods become the heap operation `getSmallest`, which removes and returns the object with the smallest value of p (the highest priority) and rebuilds the heap. This also takes O(log n) time.

The result is that we have traded a priority queue implementation (sorted/unsorted lists) that produces one excellent (O(1)) behavior and one less efficient (O(n)) behavior for another implementation (heap) that demonstrates very good (O(log n)) behavior on both. This is a worthwhile trade-off, which is why heaps are frequently used to store collections of objects ordered by priority field and retrieved in order of the smallest (or largest) value in that field.

DONALD E. KNUTH

Donald Knuth is one of the most influential and well-known computer scientists in the world. He received his PhD in mathematics from the California Institute of Technology and taught there from 1963 to 1968. In 1968 he joined the faculty of Stanford University, where he taught until 1992. He is currently a Professor Emeritus at Stanford.

Knuth has made innumerable contributions to his field, but he is probably best known for his multivolume series *The Art of Computer Programming*, the definitive description of classical computer science and one of the most widely cited references in computing. The first volume, *Fundamental Algorithms*, appeared in 1968 and is in its third edition. Succeeding volumes, entitled *Seminumerical Algorithms* and *Sorting and Searching*, were released in 1969 and 1973, respectively. Volume 4, *Combinatorial Algorithms*, is planned for release in 2006. Forty years after their initial appearance, they remain among the most widely used references in the field, and are a standard part of virtually every computer professional's library. (We use Knuth's text as an excellent reference for the Challenge Work Exercise on B-trees at the end of the chapter.)

However, Knuth is known for much more than these groundbreaking volumes. He was a pioneer in the field of algorithm analysis and made enormously important contributions to our understanding of data structures, computational complexity, and the asymptotic behavior of algorithms. (Many of the techniques in this chapter are based on his contributions.) He was deeply involved in theoretical computer science, including language semantics, translation, and models of computation. Knuth was not just a theoretician, though; he made fundamental contributions in such practical areas as compiler design and typesetting. He created the typesetting tools T_EX and METAFONT and made important contributions in the field of software development, including the concept of literate programming—how to write elegant source code for the benefit of human readers.

Among the many honors bestowed on Knuth are the National Medal of Science, the John Von Neumann Medal from the IEEE, and the ACM's A.M. Turing Award. Knuth's reputation is international—he is a Fellow of the British Royal Society and an associate of the French Academy of Science.

7.7 The Importance of Good Approximations

This chapter has demonstrated the importance of data structure design to the success of a software development project. Your choices of algorithms and data structures can produce enormous differences in efficiency. For example, assume you want to keep one million objects in a priority queue. A list requires you to perform about 500,000 comparisons, either when you originally store the object or when you attempt to locate the highest-priority object in the collection. If, instead, you are more clever and store those million items in a heap, you can both store and locate the highest-priority item in only ($\log_2$ 1,000,000) or about 20 comparisons—a reduction of almost five orders of magnitude! When designing and building programs, be sure to spend adequate time on both their structure and the selection of all key data structures. This latter decision can be instrumental to the success (or failure) of your project.

We sometimes forget that computer science is both a theoretical and an applied discipline. While a theoretical mathematician is only concerned with proving that a solution is correct, the computer scientist wants *both* a provably correct solution and a program that produces correct results in a reasonable and practical amount of time. A method that could produce correct results in 100 years would have no value.

Therefore, the ability to make good approximations (sometimes jokingly referred to as **back-of-the-envelope calculations**) is an important skill in computer science. When deciding on an algorithm or data structure, you should first try to approximate how long it will take to solve the problem using your proposed solution. If an approximation shows that the program produces an answer 10 to 20 times faster than specifications require, you can be reasonably confident that this approach will meet the user's needs. On the other hand, if the approximation shows that the finished program runs 10 to 20 times slower than required, you should probably reject the technique and look for something better.

A simple example illustrates this point. Assume that we are asked to build a spell checker for a company that sells word-processing software. Our program inputs a word from a text file and looks it up in a dictionary. If the word is there, we assume it is spelled correctly; otherwise, the program states that the word is incorrect. The dictionary contains about 100,000 words, and the software specifications state that the finished program must check the spelling of one page of error-free text (500 words) in under one second. How should we choose to store the dictionary entries? As an unordered array? As a sorted list? As a binary search tree? It is hard to say which method is best without doing some rough approximations. We should not waste our time designing and building a program that has no chance of meeting the performance requirements.

If we store our dictionary as a simple, unordered list of words, then we do not have to resort to the list whenever the user adds a new word. Although this is a nice feature, each time we attempt to locate a word to see if it is correctly spelled, we typically have to search

half the dictionary—roughly 50,000 words. If we found that it takes 1.0 μsec (10^{-6} sec) to compare, character by character, one word from the text with one word in the dictionary, then for each word in the text, we would spend an average of $50{,}000 \times 1.0 \times 10^{-6}$ seconds, or about 0.05 seconds to check its spelling. Because we assumed an average of 500 words per page, the total time to check the spelling of one page of text is $500 \times 0.05 = 25$ seconds. This is 25 times larger than the specification, and a good indication that an unordered list will not work in our dictionary. Even if our back-of-the-envelope calculations are off by an order of magnitude, we would still not meet the design goals. This simple approximation saved us many hours of programming that would not have met our needs.

Instead, what if we implemented our dictionary using a binary search tree, as described in Section 7.4? Instead of examining 50,000 words of text on average, as needed with an unordered list, we would only have to examine ($\log_2 100{,}000$) or about 17 dictionary entries to determine if a word is correctly spelled. (To find a word in the dictionary, use the `contains()` method shown in Figure 7-17.) Now the time needed to check the spelling of one page of text is:

$$500 \text{ words/page} \times 17 \text{ comparisons/word} \times 1.0 \times 10^{-6} \text{ seconds/comparison} = 0.009 \text{ seconds}$$

This is more than 100 times faster than the specification, so even if our approximation is off by two orders of magnitude, we should still be able to meet the user's needs. We can confidently continue with our design and implementation.

Remember this discussion before investing a good deal of time and effort to implement a solution in your program.

7.8 Summary

This chapter introduced the topic of hierarchical data structures. The chapter examined general trees, trees that place restrictions on the maximum number of successors (binary trees), and trees that restrict where you can add new nodes (binary search trees, red-black trees, and heaps). The chapter also demonstrated a number of interesting applications for trees in computer science, such as program compilations (parse trees), sorting (tree sort, heap sort), searching (binary search trees, red-black trees), and retrieving objects in order of their priority (heaps/priority queues). There are many other tree structures, and we encourage you to read more about this important topic.

However, it is time to move on and look at the final two classifications presented in the data structure taxonomy of Section 6.1. The next chapter investigates two interesting ways to organize data using collections called sets and graphs.

EXERCISES

1 a Using the four-rule grammar from Section 7.1, show a parse tree for the following expression:

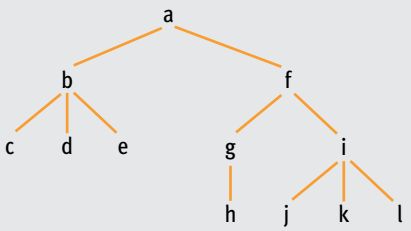
$a + (b + c * d)$

b Using the same grammar, show how a compiler could determine that the following expression:

$a + b +$

is *not* a valid statement of the language.

2 Given the following general tree:



Answer the following questions:

a What are the terminal nodes?

b What are the nonterminal nodes?

c What is the root of the tree?

d What is the degree of node a, node b, and node c?

e Who are the siblings of node c and node i?

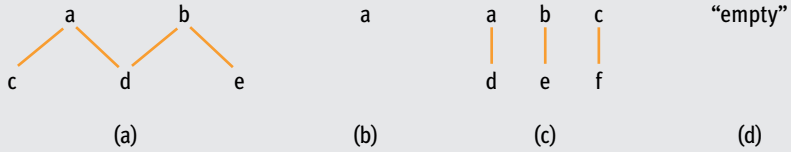
f Who are the ancestors of node h, node l, and node a?

g Who are the descendants of node f and node d?

h What is the level of node b and node c?

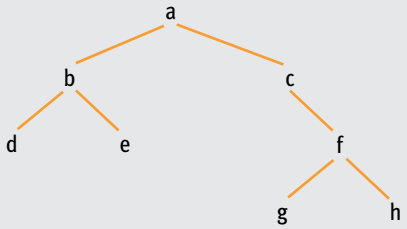
i What is the height of the tree?

3 Which of the following are trees? If a structure is not a tree, explain why.



4 Convert the general tree shown in Exercise 2 to a binary tree using the oldest child/next sibling algorithm presented in Section 7.3.3.

5 a Given the following binary tree:

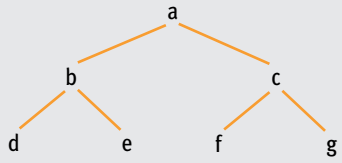


What is the order of visitation of nodes in the following:

- A preorder traversal
- A postorder traversal
- An inorder traversal

b Prove that, even if a binary tree is unbalanced, the time complexity of the three traversal algorithms used in Exercise 5a is still O(n).

6 a Given the following binary tree:



Sketch an algorithm that visits the node of a tree in *level order*; that is, it visits all nodes on level i before visiting nodes at level $i + 1$, $i = 1, 2, 3, \ldots$. In the preceding tree, a level order traversal would visit the nodes in the order *abcdefg*. Assume that you start with R, a reference to the root of the tree.

b Write a `levelOrder` method that implements the level-order algorithm you developed in Exercise 6a. Your method will be given a reference to the root of a binary tree, and it should output the contents of each node in level order.

7 A binary tree is called a **full binary tree** if all internal nodes have two children and all its leaf nodes occur on the same level. For example, the binary tree in Exercise 6 is full because all the internal nodes (a, b, c) have two children, and all leaf nodes (d, e, f, g) occur on level 3. Write a Boolean method called `fullChecker` that takes a pointer to the root of a binary tree and determines whether the tree is full. It returns true if the tree is full and false otherwise.

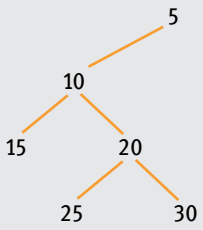
8 In Exercise 6, there are $n = 7$ nodes in a full binary tree of height 3. Develop a formula for the relationship between n, the total number of nodes in a full binary tree, and i, the level number on which the terminal nodes occur.

9 **a** Specify the pre- and postconditions for the following operations on binary trees, and add them to the binary tree interface of Figure 7-10b:

- A `treeCopy` method that makes an identical copy of a binary tree and returns a reference to the root of the copy
- A `treePrint` method that prints the information field of every node in the tree
- An `empty` method that is true if the tree is empty, and false otherwise

b Implement these three methods using the linked list representation of a binary tree presented in Section 7.3.4.

10 Show what the following binary tree might look like when stored in the array representation discussed in Section 7.3.5 and diagrammed in Figure 7-14.



11 Show the binary search tree that results from inserting the following values in exactly the order shown, beginning from an empty tree:

a 38, 65, 27, 29, 81, 70, 14, 53, 12, 20

b 81, 73, 70, 52, 51, 40, 42, 38, 35, 20

What do these two cases say about the structure of the binary search tree as a function of the input data's order?

12 Write a `distantNode` method that uses the resources of `LinkedBinaryTree` to identify the node farthest from the root. The specifications for this method are:

```
/*  Precondition: none
    Postcondition: The method returns the identity of the node
                   that is farthest from the root. If more than
                   one node is the same distance from the root,
                   this method can return the identity of any
                   one of them. */
public BinaryTreeNode distantNode ()
```

13 In a binary search tree with 1023 nodes, the minimum height is 10 if the tree is full. The maximum height is 1023 if the tree is degenerate. Generate 100 distinct sequences of exactly 1023 random integers, build a binary search tree from each sequence, and determine the height, H, of each binary search tree using the `height()` method in the `LinkedBinarySearchTree` class. Use your data to show empirically that the expected height of a BST built from a random sequence of values is closer to $(\log_2 n)$ than to n. How close to $(\log_2 n)$ was the average height?

14 Show that if we delete a node in a binary search tree and replace it with its successor in an inorder traversal, the result is still a binary search tree. Does this also hold true for either the preorder or postorder traversal successor?

15 Compute the total amount of memory required to store a dictionary that contains 100,000 words using:

a A singly linked list

b A doubly linked list

c A binary search tree

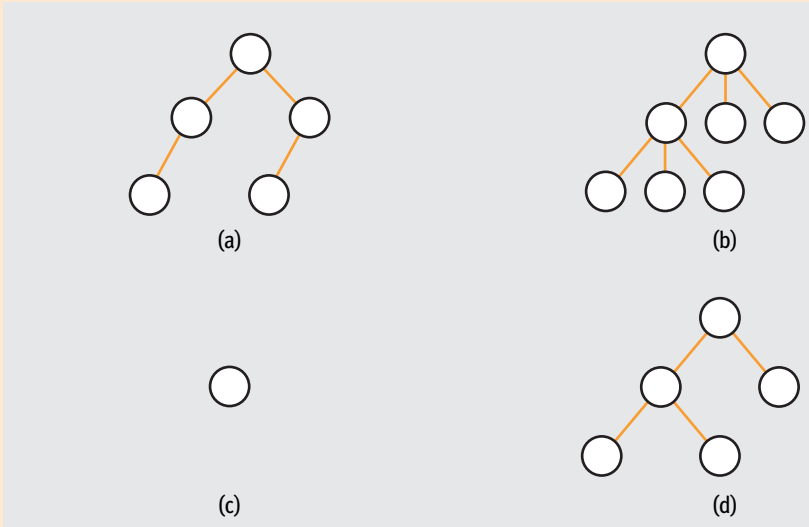
When performing the computations, assume the following:

- A reference variable requires 4 bytes.
- A character requires 1 byte.
- Each English word is 5 bytes in length.

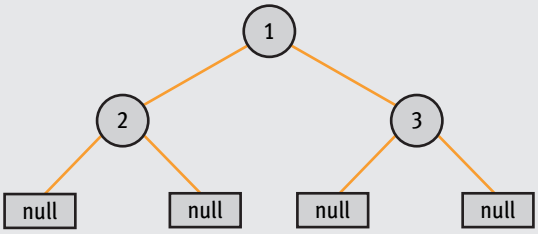
16 **a** Estimate how long it would take to locate a specific word in the binary search tree of Exercise 15, assuming that the tree is balanced and it takes 1 μsec to access any node. Does this allow us to meet the requirement that we can check the spelling of a page of text (500 words) in under 1 second?

b What if a tree is out of balance instead of being balanced? Instead of a height $H = \log_2 n$, the height is $(k \log_2 n)$ for $k > 1$. How large can k get and still allow us to meet the performance requirements specified in Exercise 16a in the worst case?

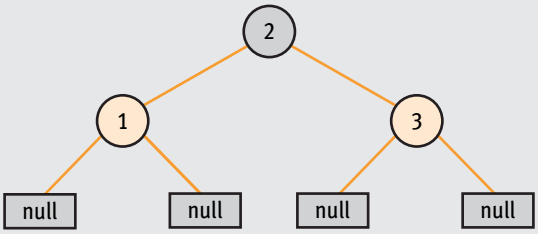
17 Do each of the following shapes represent a *complete* binary tree? If not, explain why.



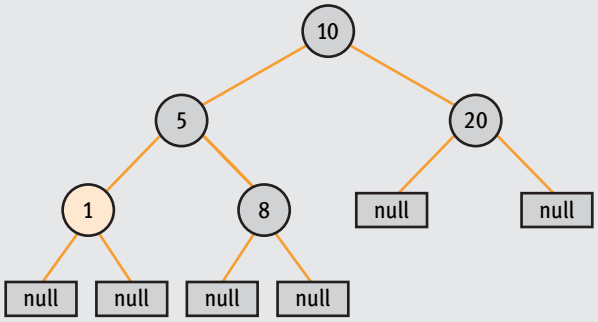
18 Are each of the following valid red-black trees? If not, state why.



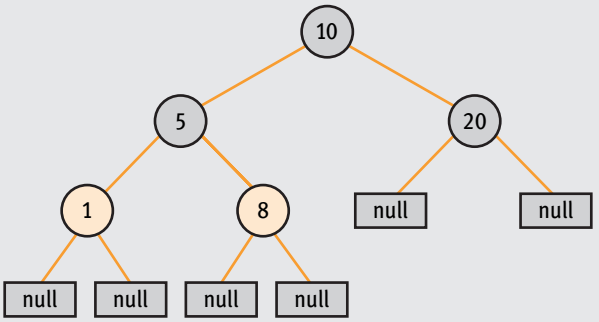
(a)



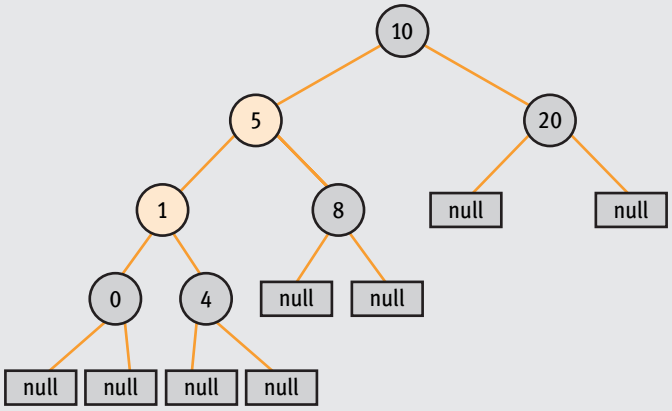
(b)



(c)

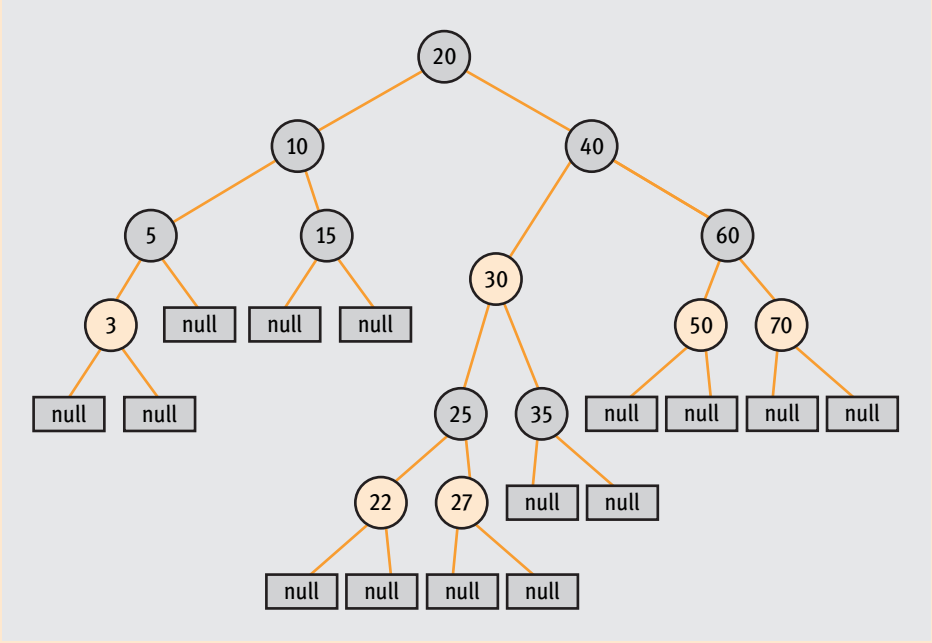

10
5
20
1
8
null
null
null
null
null
null

(d)


10
5
20
1
8
null
null
0
4
null
null
null
null
null
null

(e)

19 **a** Is the following tree a red-black tree?



Review the five properties of a red-black tree and determine whether the tree satisfies all of them.

b Show what the tree would look like after you inserted the following values:

- The new value 7
- The new value 48
- The new value 21

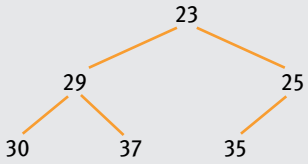
20 **a** Show the structure that results from beginning with an empty tree and constructing a heap from the following seven integers in exactly the order shown:

$S = \{20, 19, 17, 23, 18, 22, 15\}$

b Show how the final heap is stored internally as a heapform.

21 Develop the formula that determines whether a heap node, stored as a heapform, does or does not have *first cousin*s. Two nodes are said to be first cousins if they have the same grandparent but different parents. In Figure 7-31(a), nodes (X_3, X_5) and (X_4, X_5) are first cousins. None of the other nodes stand in this relationship. Develop a formula `firstCousin(i)` that returns all of the first cousins of I in the heapform.

22 What would the following heap look like after you completed each of the following three operations in sequence—in other words, do operation (b) on the heap produced by operation (a), and so on?



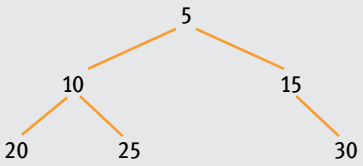
a Insert a 26.

b Insert a 22.

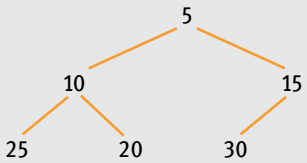
c Delete the smallest value.

d Delete the smallest value.

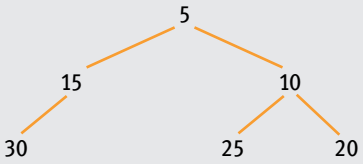
23 Write a `mirrorImage` method that is given a binary tree T and that returns the *mirror image* of that tree. A mirror image is one in which every left and right subtree have been interchanged. For example, given the following binary tree:



You first switch the left and right subtrees of the trees rooted at 10 and 15. This produces the following:



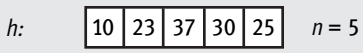
Next, move up one level and switch the left and right subtrees of the tree rooted at 5:



This is the mirror image of the original tree that is returned by `mirrorImage`.

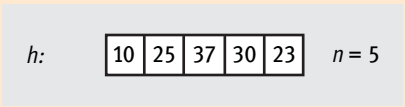
24 Write a `leafCounter` method that is given an arbitrary binary tree T and that counts how many leaf nodes are present in T. Remember, a leaf node is a node with degree 0.

25 Write a `heapChecker` method that is given an array h of size n. The method determines whether the n integer values stored in h constitute a legal heap. For example, given the following five-element integer array:



The method returns true because the five values form a heap. To confirm this, draw the heap as a tree and check the order property.

However, given the following array:



The method should return false because the two values 25 and 23 violate the order property of heaps.

26 Implement two or more of the four O($n \log n$) sorts presented in previous chapters: the merge sort, Quicksort, tree sort, and heap sort. Generate a file of at least 100,000 random numbers, sort them using each of the sorting methods, and collect timing information on how quickly they work. (If you have a fast machine, you may have to enlarge the file to get accurate timing data.) Which one sorts the fastest? If we assume that the relationship between problem size n and solution time t is $t = k\ n \log n$, how large a file can we sort in five minutes using each of the sorting methods? How large a file can we sort in one hour?

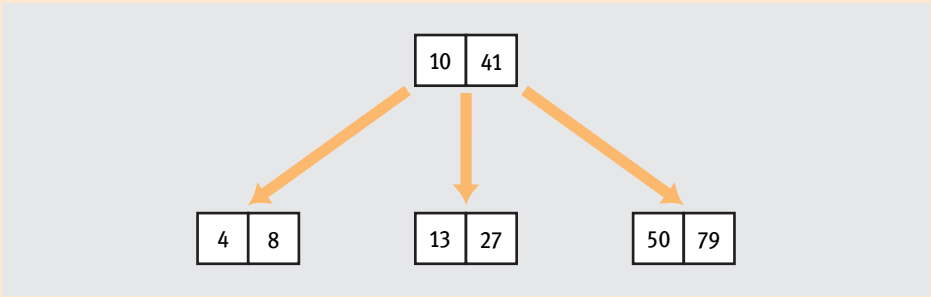
CHALLENGE WORK EXERCISE

The search trees discussed in this chapter (BST, red-black trees) are binary trees that had two children at the most. However, an interesting variation of the search tree is definitely not binary. It is the **B-tree**, a search tree that can be of any degree $m > 2$.

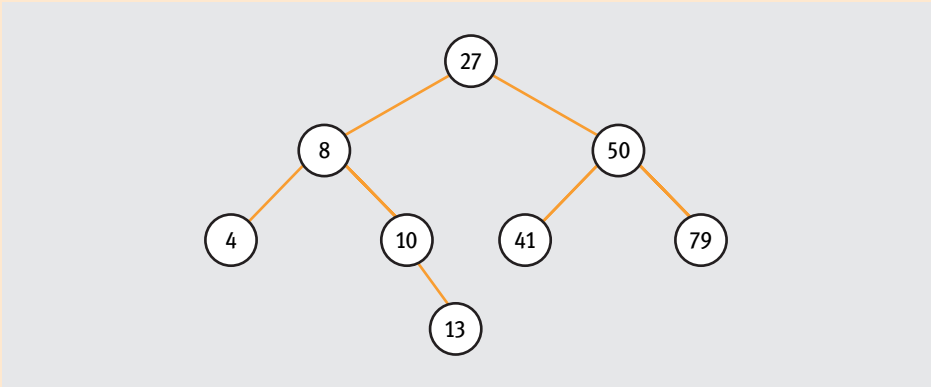
A binary search tree has one key value in each node, and uses this key to divide the remaining elements into two distinct groups: those less than the key and those greater than the key. We can generalize this structure in the following way. Assume that $m - 1$ key values $\{k_1, k_2, \ldots, k_{m-1}\}$ are stored in ascending order in each node of the tree. This is called a B-tree of degree m. Use these $m - 1$ keys to divide the remaining elements of the tree into the following m distinct groups:

- All values v less than k_1
- All values v in the range $k_1 \leq v < k_2$
- All values v in the range $k_2 \leq v < k_3$
- All values v in the range $k_2 \leq v < k_3$
- …
- All values v in the range $k_{m-2} \leq v < k_{m-1}$
- All values v greater than or equal to k_{m-1}

In addition to allowing more than a single key to be stored in a node, a B-tree has other important characteristics, including the two balancing rules, which say that a B-tree must be **node balanced**—each node must be at least half filled—and a B-tree must be height balanced—every leaf must be on the same level. The following diagram shows a B-tree of degree 3, which means we can store up to two data elements per node:



The B-tree is widely used in computer science to store the directory information contained on a hard disk, because the higher the degree of the B-tree, the "flatter" the resulting tree structure. That is, the tree is wider and not as deep as one with a lower degree *m*. If we assume that each node of the search tree is stored as a single disk sector, then the higher the degree tree, the fewer sectors we need to read. For example, in the degree 3 B-tree shown previously, the height of the tree is 2, and we can locate any value in the tree by reading in and searching two sectors at most. However, if the same information were stored as a binary search tree, it would have a height of at least 4:



This might require us to read in and search up to four disk sectors.

Disk access can be five or six orders of magnitude slower than the time required to perform an in-memory computation. Therefore, a data structure that allows us to search a collection of values stored on a disk by examining the fewest number of nodes, which means the fewest disk accesses, is extremely efficient. The B-tree offers this benefit.

Read about the B-tree data structure and answer the following questions:

- What rules exist for balancing the content of each node as well as the nodes in the tree?
- What is the algorithm for searching or inserting into a B-tree?
- What algorithm is used to balance the B-tree after an insertion?
- What is the time complexity of a search operation? An insertion operation?

Write a report that explains how B-trees can be used to represent the directory information stored on a hard disk.

Two good sources of information on B-trees are:

- *http://en.wikipedia.org/wiki/B-tree*
- Donald Knuth, *The Art of Computer Programming*, Volume 3, Sorting and Searching, 3rd Edition, Addison-Wesley, ISBN 0-201-89685-0, Section 6.2.4: Multiway Trees.