6 Binary Trees

6.1 Trees, Binary Trees, and Binary Search Trees

Linked lists usually provide greater flexibility than arrays, but they are linear structures and so it is difficult to use them to organize a hierarchical representation of objects. Although stacks and queues reflect some hierarchy, they are limited to only one dimension. To overcome this limitation, we create a new data type called a *tree* that consists of *nodes* and *arcs.* Unlike natural trees, these trees are depicted upside down with the *root* at the top and the *leaves* (*terminal nodes*) at the bottom. The root is a node that has no parent; it can have only child nodes. Leaves, on the other hand, have no children, or rather, their children are empty structures. A tree can be defined recursively as the following:

1. An empty structure is an empty tree.
2. If $t_1, \ldots, t_k$ are disjointed trees, then the structure whose root has as its children the roots of $t_1, \ldots, t_k$ is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

Figure 6.1 contains examples of trees. Each node has to be reachable from the root through a unique sequence of arcs, called a *path.* The number of arcs in a path is called the *length* of the path. The *level* of a node is the length of the path from the root to the node plus 1, which is the number of nodes in the path. The *height* of a nonempty tree is the maximum level of a node in the tree. The empty tree is a legitimate tree of height 0 (by definition), and a single node is a tree of height 1. This is the only case in which a node is both the root and a leaf. The level of a node must be between 1 (the level of the root) and the height of the tree, which in the extreme case is the level of the only leaf in a degenerate tree resembling a linked list.

Figure 6.2 contains an example of a tree that reflects the hierarchy of a university. Other examples are genealogical trees, trees reflecting the grammatical structure of sentences, and trees showing the taxonomic structure of organisms, plants, or characters. Virtually all areas of science make use of trees to represent hierarchical structures.

FIGURE 6.1 Examples of trees.

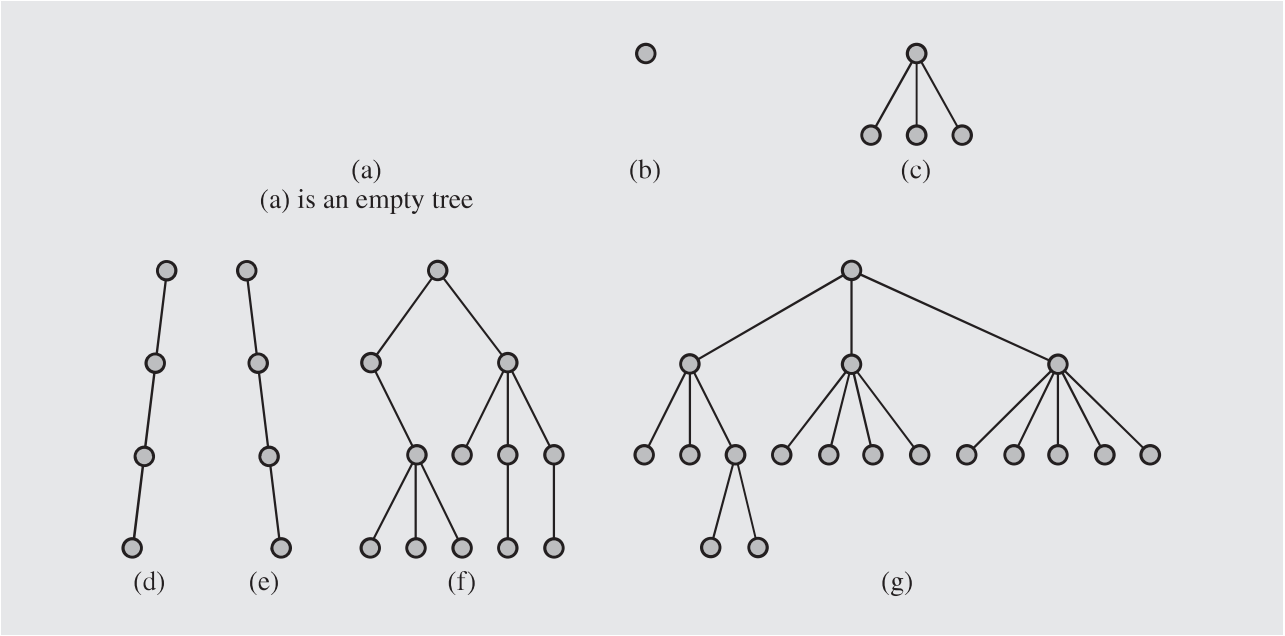
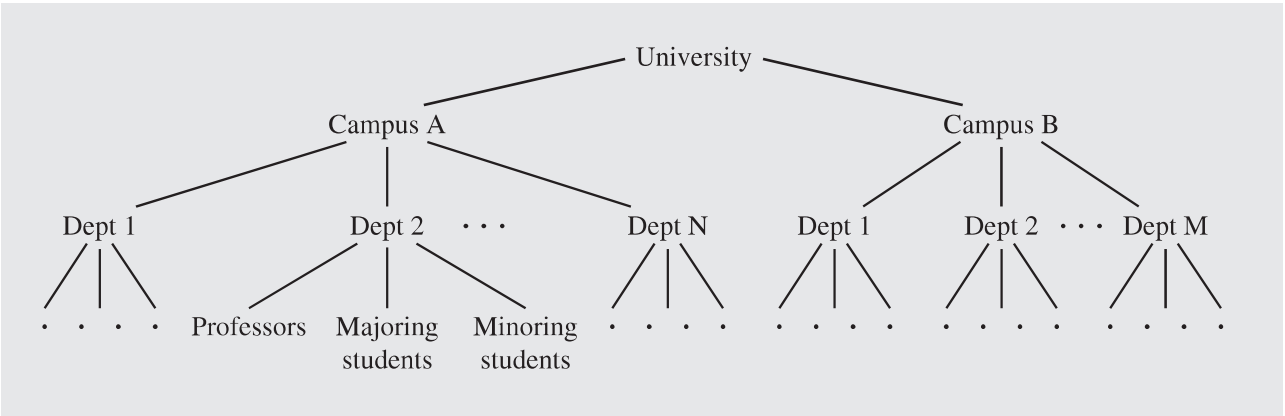


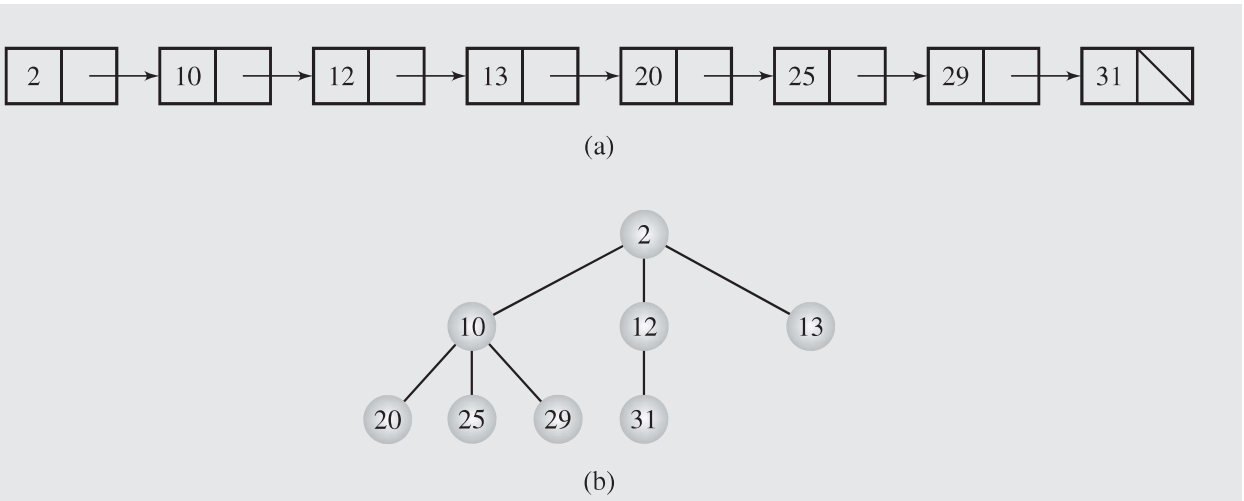
FIGURE 6.2 Hierarchical structure of a university shown as a tree.



The definition of a tree does not impose any condition on the number of children of a given node. This number can vary from 0 to any integer. In hierarchical trees, this is a welcome property. For example, the university has only two branches, but each campus can have a different number of departments. But representing hierarchies is not the only reason for using trees. In fact, in the discussion to follow, that aspect of trees is treated rather lightly, mainly in the discussion of expression trees. This chapter focuses on tree operations that allow us to accelerate the search process.

Consider a linked list of *n* elements. To locate an element, the search has to start from the beginning of the list, and the list must be scanned until the element is found or the end of the list is reached. Even if the list is ordered, the search of the list always has to start from the first node. Thus, if the list has 10,000 nodes and the information in the last node is to be accessed, then all 9,999 of its predecessors have to be traversed, an obvious inconvenience. If all the elements are stored in an *orderly tree,* a tree where all elements are stored according to some predetermined criterion of ordering, the number of tests can be reduced substantially even when the element to be located is the one farthest away. For example, the linked list in Figure 6.3a can be transformed into the tree in Figure 6.3b.

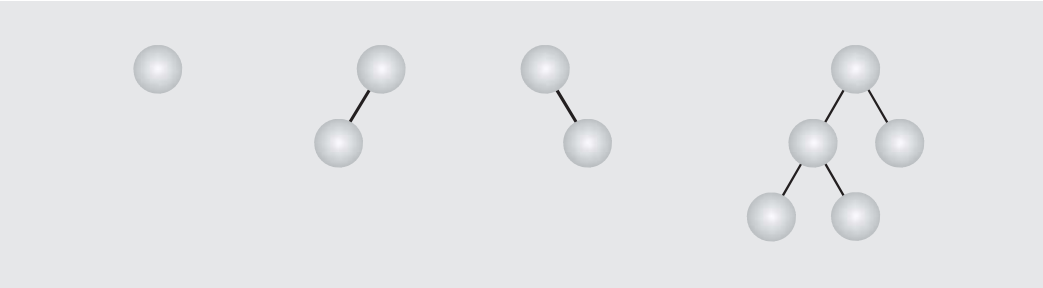
FIGURE 6.3 Transforming (a) a linked list into (b) a tree.



Was a reasonable criterion of ordering applied to construct this tree? To test whether 31 is in the linked list, eight tests have to be performed. Can this number be reduced further if the same elements are ordered from top to bottom and from left to right in the tree? What would an algorithm be like that forces us to make three tests only: one for the root, 2; one for its middle child, 12; and one for the only child of this child, 31? The number 31 could be located on the same level as 12, or it could be a child of 10. With this ordering of the tree, nothing really interesting is achieved in the context of searching. (The heap discussed later in this chapter uses this approach.) Consequently, a better criterion must be chosen.

Again, note that each node can have any number of children. In fact, there are algorithms developed for trees with a deliberate number of children (see the next chapter), but this chapter discusses only binary trees. A *binary tree* is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child. For example, the trees in Figure 6.4 are binary trees,

FIGURE 6.4 Examples of binary trees.



whereas the university tree in Figure 6.2 is not. An important characteristic of binary trees, which is used later in assessing an expected efficiency of sorting algorithms, is the number of leaves.

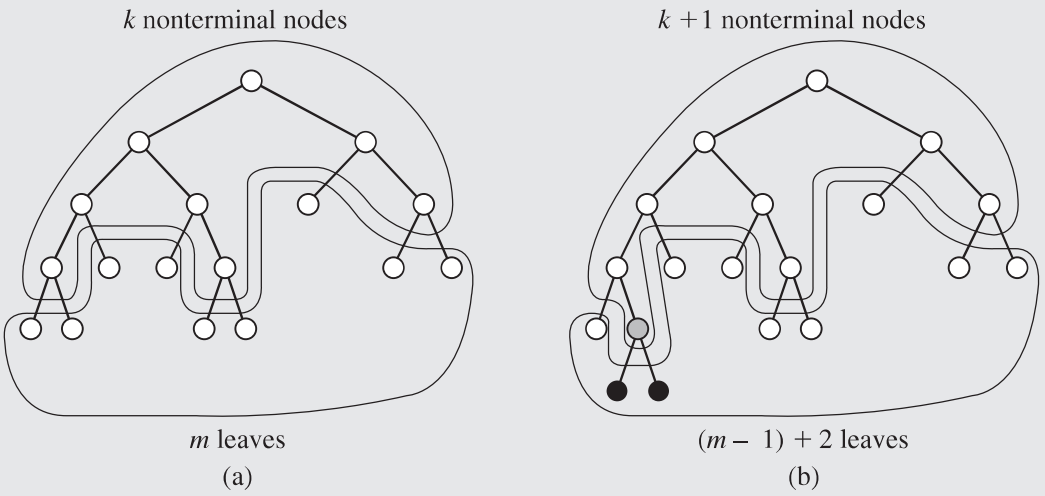
As already defined, the level of a node is the number of arcs traversed from the root to the node plus one. According to this definition, the root is at level 1, its nonempty children are at level 2, and so on. If all the nodes at all levels except the last had two children, then there would be $1 = 2^0$ node at level 1, $2 = 2^1$ nodes at level 2, $4 = 2^2$ nodes at level 3, and generally, 2^i nodes at level $i + 1$. A tree satisfying this condition is referred to as a *complete binary tree.* In this tree, all nonterminal nodes have both their children, and all leaves are at the same level. Consequently, in all binary trees, there are at most 2^i nodes at level $i + 1$. In Chapter 9, we calculate the number of leaves in a *decision tree,* which is a binary tree in which all nodes have either zero or two nonempty children. Because leaves can be interspersed throughout a decision tree and appear at each level except level 1, no generally applicable formula can be given to calculate the number of nodes because it may vary from tree to tree. But the formula can be approximated by noting first that

> For all the nonempty binary trees whose nonterminal nodes have exactly two nonempty children, the number of leaves m is greater than the number of nonterminal nodes k and $m = k + 1$.

If a tree has only a root, this observation holds trivially. If it holds for a certain tree, then after attaching two leaves to one of the already existing leaves, this leaf turns into a nonterminal node, whereby m is decremented by 1 and k is incremented by 1. However, because two new leaves have been grafted onto the tree, m is incremented by 2. After these two increments and one decrement, the equation $(m - 1) + 2 = (k + 1) + 1$ is obtained and $m = k + 1$, which is exactly the result aimed at (see Figure 6.5). It implies that an $i + 1$–level complete decision tree has 2^i leaves, and due to the preceding observation, it also has $2^i - 1$ nonterminal nodes, which makes $2^i + 2^i - 1 = 2^{i+1} - 1$ nodes in total (see also Figure 6.35).

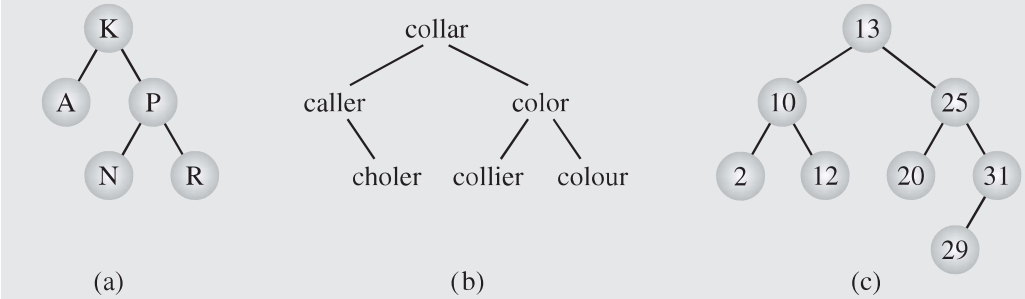
In this chapter, the *binary search trees,* also called *ordered binary trees,* are of particular interest. A binary search tree has the following property: for each node n of the tree, all values stored in its left subtree (the tree whose root is the left child) are less than value v stored in n, and all values stored in the right subtree are greater than or equal to v. For reasons to be discussed later, storing multiple copies of the same

FIGURE 6.5 (a) Adding a leaf to tree, (b) preserving the relation of the number of leaves to the number of nonterminal nodes.



value in the same tree is avoided. An attempt to do so can be treated as an error. The meanings of "less than" or "greater than" depend on the type of values stored in the tree. We use operators "<" and ">", which can be overloaded depending on the content. Alphabetical order is also used in the case of strings. The trees in Figure 6.6 are binary search trees. Note that Figure 6.6c contains a tree with the same data as the linked list in Figure 6.3a whose searching was to be optimized.

FIGURE 6.6 Examples of binary search trees.



6.2 Implementing Binary Trees

Binary trees can be implemented in at least two ways: as arrays and as linked structures. To implement a tree as an array, a node is declared as a structure with an information field and two "pointer" fields. These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any. For example, the tree from Figure 6.6c can be represented as the array in Figure 6.7. The root is always located in the first cell, cell 0, and –1 indicates a null child. In this representation, the two children of node 13 are located in positions 4 and 2, and the right child of node 31 is null.

FIGURE 6.7 Array representation of the tree in Figure 6.6c.

Index	Info	Left	Right
0	13	4	2
1	31	6	–1
2	25	7	1
3	12	–1	–1
4	10	5	3
5	2	–1	–1
6	29	–1	–1
7	20	–1	–1

However, this implementation may be inconvenient, even if the array is flexible—that is, a vector. Locations of children must be known to insert a new node, and these locations may need to be located sequentially. After deleting a node from the tree, a hole in the array would have to be eliminated. This can be done either by using a special marker for an unused cell, which may lead to populating the array with many unused cells, or by moving elements by one position, which also requires updating references to the elements that have been moved. Sometimes an array implementation is convenient and desirable, and it will be used when discussing the heap sort. But usually, another approach needs to be used.

In the new implementation, a node is an instance of a class composed of an information member and two pointer members. This node is used and operated on by member functions in another class that pertains to the tree as a whole (see Figure 6.8). For this reason, members of `BSTNode` are declared public because they can be accessible only from nonpublic members of objects of type `BST` so that the information-hiding principle still stands. It is important to have members of `BSTNode` be public because otherwise they are not accessible to classes derived from `BST`.

FIGURE 6.8 Implementation of a generic binary search tree.

```
//***********************  genBST.h  ************************
//                  generic binary search tree

#include <queue>
#include <stack>

using namespace std;

#ifndef BINARY_SEARCH_TREE
#define BINARY_SEARCH_TREE

template<class T>
class Stack : public stack<T> { ... } // as in Figure 4.21

template<class T>
class Queue : public queue<T> {
public:
    T dequeue() {
        T tmp = front();
        queue<T>::pop();
        return tmp;
    }
    void enqueue(const T& el) {
        push(el);
    }
};

template<class T>
class BSTNode {
public:
    BSTNode() {
        left = right = 0;
    }
    BSTNode(const T& e, BSTNode<T> *l = 0, BSTNode<T> *r = 0) {
        el = e; left = l; right = r;
    }
    T el;
    BSTNode<T> *left, *right;
};
```

FIGURE 6.8 (*continued*)

```
template<class T>
class BST {
public:
    BST() {
        root = 0;
    }
    ~BST() {
        clear();
    }
    void clear() {
        clear(root); root = 0;
    }
    bool isEmpty() const {
        return root == 0;
    }
    void preorder() {
        preorder(root);                      // Figure 6.11
    }
    void inorder() {
        inorder(root);                       // Figure 6.11
    }
    void postorder() {
        postorder(root);                     // Figure 6.11
    }
    T* search(const T& el) const {
        return search(root,el);              // Figure 6.9
    }
    void breadthFirst();                     // Figure 6.10
    void iterativePreorder();                // Figure 6.15
    void iterativeInorder();                 // Figure 6.17
    void iterativePostorder();               // Figure 6.16
    void MorrisInorder();                    // Figure 6.20
    void insert(const T&);                   // Figure 6.23
    void deleteByMerging(BSTNode<T>*&);      // Figure 6.29
    void findAndDeleteByMerging(const T&);   // Figure 6.29
    void deleteByCopying(BSTNode<T>*&);      // Figure 6.32
    void balance(T*,int,int);                // Section 6.7
    . . . . . .
protected:
    BSTNode<T>* root;
```

Continues

FIGURE 6.8 (*continued*)

```
    void clear(BSTNode<T>*);
    T* search(BSTNode<T>*, const T&) const;  // Figure 6.9
    void preorder(BSTNode<T>*);              // Figure 6.11
    void inorder(BSTNode<T>*);               // Figure 6.11
    void postorder(BSTNode<T>*);             // Figure 6.11
    virtual void visit(BSTNode<T>* p) {
        cout << p->el << ' ';
    }
    . . . . . .
};

#endif
```

6.3 Searching a Binary Search Tree

An algorithm for locating an element in this tree is quite straightforward, as indicated by its implementation in Figure 6.9. For every node, compare the element to be located with the value stored in the node currently pointed at. If the element is less than the value, go to the left subtree and try again. If it is greater than that value, try the right subtree. If it is the same, obviously the search can be discontinued. The search is also aborted if there is no way to go, indicating that the element is not in the tree. For example, to locate the number 31 in the tree in Figure 6.6c, only three tests are performed. First, the tree is checked to see if the number is in the root node. Next, because 31 is greater than 13, the root's right child containing the value 25 is tried. Finally, because 31 is again

FIGURE 6.9 A function for searching a binary search tree.

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
    while (p != 0)
        if (el == p->el)
            return &p->el;
        else if (el < p->el)
            p = p->left;
        else p = p->right;
    return 0;
}
```

greater than the value of the currently tested node, the right child is tried again, and the value 31 is found.

The worst case for this binary tree is when it is searched for the numbers 26, 27, 28, 29, or 30 because those searches each require four tests (why?). In the case of all other integers, the number of tests is fewer than four. It can now be seen why an element should only occur in a tree once. If it occurs more than once, then two approaches are possible. One approach locates the first occurrence of an element and disregards the others. In this case, the tree contains redundant nodes that are never used for their own sake; they are accessed only for testing. In the second approach, all occurrences of an element may have to be located. Such a search always has to finish with a leaf. For example, to locate all instances of 13 in the tree, the root node 13 has to be tested, then its right child 25, and finally the node 20. The search proceeds along the worst-case scenario: when the leaf level has to be reached in expectation that some more occurrences of the desired element can be encountered.

The complexity of searching is measured by the number of comparisons performed during the searching process. This number depends on the number of nodes encountered on the unique path leading from the root to the node being searched for. Therefore, the complexity is the length of the path leading to this node plus 1. Complexity depends on the shape of the tree and the position of the node in the tree.

The *internal path length (IPL)* is the sum of all path lengths of all nodes, which is calculated by summing $\sum(i-1)l_i$ over all levels i, where l_i is the number of nodes on level i. A depth of a node in the tree is determined by the path length. An average depth, called an *average path length,* is given by the formula IPL/n, which depends on the shape of the tree. In the worst case, when the tree turns into a linked list, $path_{worst} = \frac{1}{n}\sum_{i=1}^{n}(i-1) = \frac{n-1}{2} = O(n)$, and a search can take n time units.

The best case occurs when all leaves in the tree of height h are in at most two levels, and only nodes in the next to last level can have one child. To simplify the computation, we approximate the average path length for such a tree, $path_{best}$, by the average path of a complete binary tree of the same height.

By looking at simple examples, we can determine that for the complete binary tree of height h, IPL $= \sum_{i=1}^{h-1} i2^i$. From this and from the fact that $\sum_{i=1}^{h-1}2^i = 2^h - 2$, we have

$$\text{IPL} = 2\text{IPL} - \text{IPL} = (h-1)2^h - \sum_{i=1}^{h-1} 2^i = (h-2)2^h + 2$$

As has already been established, the number of nodes in the complete binary tree $n = 2^h - 1$, so

$$path_{best} = \text{IPL}/n = \Big((h-2)2^h + 2\Big)/(2^h - 1) \approx h - 2$$

which is in accordance with the fact that, in this tree, one-half of the nodes are in the leaf level with path length $h - 1$. Also, in this tree, the height $h = \lg(n+1)$, so $path_{best} = \lg(n+1) - 2$; the average path length in a perfectly balanced tree is $\lceil \lg(n+1) \rceil - 2 = O(\lg n)$ where $\lceil x \rceil$ is the closest integer greater than x.

The average case in an average tree is somewhere between $\frac{n-1}{2}$ and $\lg(n+1) - 2$. Is a search for a node in an average position in a tree of average shape closer to $O(n)$ or $O(\lg n)$? First, the average shape of the tree has to be represented computationally.

The root of a binary tree can have an empty left subtree and a right subtree with all $n - 1$ nodes. It can also have one node in the left subtree and $n - 2$ nodes in the

right, and so on. Finally, it can have an empty right subtree with all remaining nodes in the left. The same reasoning can be applied to both subtrees of the root, to the subtrees of these subtrees, down to the leaves. The average internal path length is the average over all these differently shaped trees.

Assume that the tree contains nodes 1 through n. If i is the root, then its left subtree has $i - 1$ nodes, and its right subtree has $n - i$ nodes. If $path_{i-1}$ and $path_{n-i}$ are average paths in these subtrees, then the average path of this tree is

$$path_n(i) = ((i-1)(path_{i-1} + 1) + (n-i)(path_{n-i} + 1))/n$$

Assuming that elements are coming randomly to the tree, the root of the tree can be any number i, $1 \le i \le n$. Therefore, the average path of an average tree is obtained by averaging all values of $path_n(i)$ over all values of i. This gives the formula

$$path_n = \frac{1}{n}\sum_{i=1}^{n} path_n(i) = \frac{1}{n^2}\sum_{i=1}^{n} ((i-1)(path_{i-1} + 1) + (n-i)(path_{n-i} + 1))$$

$$= \frac{2}{n^2}\sum_{i=1}^{n-1} i(path_i + 1)$$

from which, and from $path_1 = 0$, we obtain $2 \ln n = 2 \ln 2 \lg n = 1.386 \lg n$ as an approximation for $path_n$ (see section A.4 in Appendix A). This is an approximation for the average number of comparisons in an average tree. This number is $O(\lg n)$, which is closer to the best case than to the worst case. This number also indicates that there is little room for improvement, because $path_{best}/path_n \approx .7215$, and the average path length in the best case is different by only 27.85% from the expected path length in the average case. Searching in a binary tree is, therefore, very efficient in most cases, even without balancing the tree. However, this is true only for randomly created trees because, in highly unbalanced and elongated trees whose shapes resemble linked lists, search time is $O(n)$, which is unacceptable considering that $O(\lg n)$ efficiency can be achieved.

6.4 Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly one time. Traversal may be interpreted as putting all nodes on one line or linearizing a tree.

The definition of traversal specifies only one condition—visiting each node only one time—but it does not specify the order in which the nodes are visited. Hence, there are as many tree traversals as there are permutations of nodes; for a tree with n nodes, there are $n!$ different traversals. Most of them, however, are rather chaotic and do not indicate much regularity so that implementing such traversals lacks generality: for each n, a separate set of traversal procedures must be implemented, and only a few of them can be used for a different number of data. For example, two possible traversals of the tree in Figure 6.6c that may be of some use are the sequence 2, 10, 12, 20, 13, 25, 29, 31 and the sequence 29, 31, 20, 12, 2, 25, 10, 13. The first sequence lists even numbers and then odd numbers in ascending order. The second sequence lists all nodes from level to level right to left, starting from the lowest level up to the root. The sequence 13, 31, 12, 2, 10, 29, 20, 25 does not indicate any regularity in the order of

numbers or in the order of the traversed nodes. It is just a random jumping from node to node that in all likelihood is of no use. Nevertheless, all these sequences are the results of three legitimate traversals out of 8! = 40,320. In the face of such an abundance of traversals and the apparent uselessness of most of them, we would like to restrict our attention to two classes only, namely, breadth-first and depth-first traversals.

6.4.1 Breadth-First Traversal

Breadth-first traversal is visiting each node starting from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left). There are thus four possibilities, and one such possibility—a top-down, left-to-right breadth-first traversal of the tree in Figure 6.6c—results in the sequence 13, 10, 25, 2, 12, 20, 31, 29.

Implementation of this kind of traversal is straightforward when a queue is used. Consider a top-down, left-to-right, breadth-first traversal. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited. Considering that for a node on level n, its children are on level $n + 1$, by placing these children at the end of the queue, they are visited after all nodes from level n are visited. Thus, the restriction that all nodes on level n must be visited before visiting any nodes on level $n + 1$ is met.

An implementation of the corresponding member function is shown in Figure 6.10.

FIGURE 6.10 Top-down, left-to-right, breadth-first traversal implementation.

```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);
            if (p->right != 0)
                 queue.enqueue(p->right);
        }
    }
}
```

6.4.2 Depth-First Traversal

Depth-first traversal proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right). We repeat this process until all nodes are visited. This definition, however, does not clearly specify exactly when nodes are visited: before proceeding down the tree or after backing up? There are some variations of the depth-first traversal.

There are three tasks of interest in this type of traversal:

V—visiting a node

L—traversing the left subtree

R—traversing the right subtree

An orderly traversal takes place if these tasks are performed in the same order for each node. The three tasks can themselves be ordered in 3! = 6 ways, so there are six possible ordered depth-first traversals:

VLR VRL

LVR RVL

LRV RLV

If the number of different orders still seems like a lot, it can be reduced to three traversals where the move is always from left to right and attention is focused on the first column. The three traversals are given these standard names:

VLR—preorder tree traversal

LVR—inorder tree traversal

LRV—postorder tree traversal

Short and elegant functions can be implemented directly from the symbolic descriptions of these three traversals, as shown in Figure 6.11.

These functions may seem too simplistic, but their real power lies in recursion, in fact, double recursion. The real job is done by the system on the run-time stack. This simplifies coding but lays a heavy burden upon the system. To better understand this process, inorder tree traversal is discussed in some detail.

In inorder traversal, the left subtree of the current node is visited first, then the node itself, and finally, the right subtree. All of this, obviously, holds if the tree is not empty. Before analyzing the run-time stack, the output given by the inorder traversal is determined by referring to Figure 6.12. The following steps correspond to the letters in that figure:

(a) Node 15 is the root on which `inorder()` is called for the first time. The function calls itself for node 15's left child, node 4.

(b) Node 4 is not null, so `inorder()` is called on node 1. Because node 1 is a leaf (that is, both its subtrees are empty), invocations of `inorder()` on the subtrees do not result in other recursive calls of `inorder()`, as the condition in the `if` statement is not met. Thus, after `inorder()` called for the null left subtree is finished, node 1 is visited and then a quick call to `inorder()` is executed for the null right subtree of node 1. After

FIGURE 6.11 Depth-first traversal implementation.

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
     if (p != 0) {
         inorder(p->left);
         visit(p);
         inorder(p->right);
     }
}

template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}

template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

resuming the call for node 4, node 4 is visited. Node 4 has a null right subtree; hence, `inorder()` is called only to check that, and right after resuming the call for node 15, node 15 is visited.

(c) Node 15 has a right subtree, so `inorder()` is called for node 20.

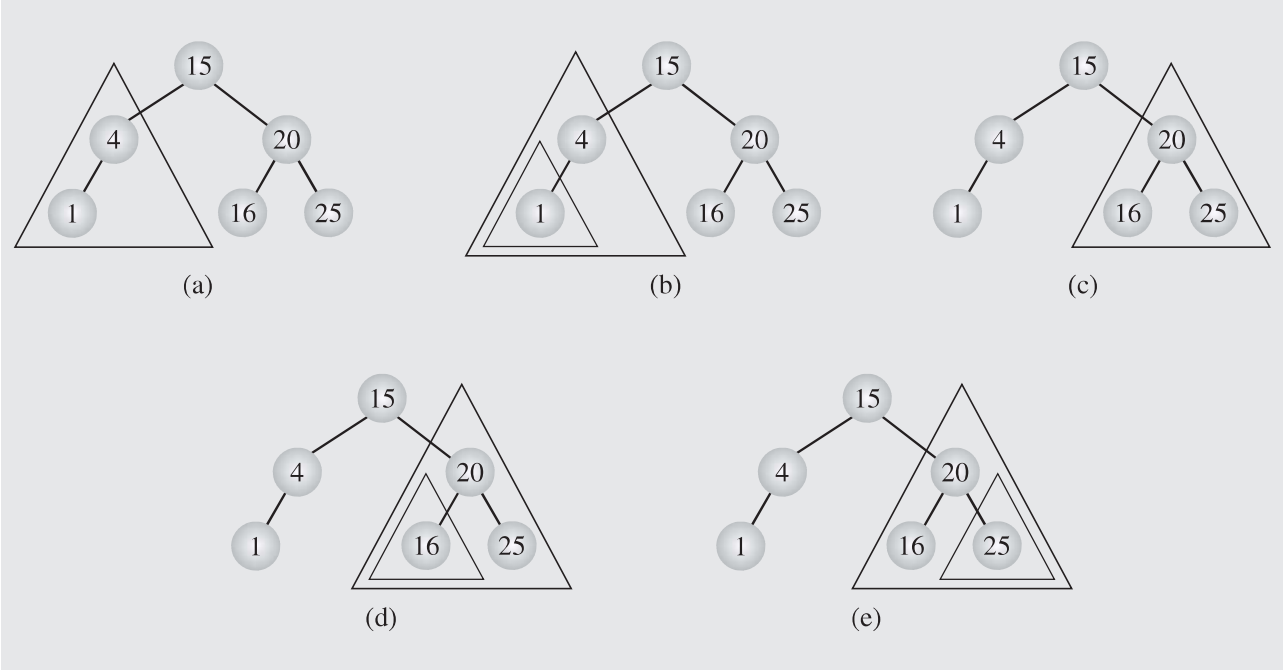
(d) `inorder()` is called for node 16, the node is visited, and then on its null left subtree, which is followed by visiting node 16. After a quick call to `inorder()` on the null right subtree of node 16 and return to the call on node 20, node 20 is also visited.

(e) `inorder()` is called on node 25, then on its empty left subtree, then node 25 is visited, and finally `inorder()` is called on node 25's empty right subtree.

If the visit includes printing the value stored in a node, then the output is:

```
1 4 15 16 20 25
```

FIGURE 6.12 Inorder tree traversal.



The key to the traversal is that the three tasks, L, V, and R, are performed for each node separately. This means that the traversal of the right subtree of a node is held pending until the first two tasks, L and V, are accomplished. If the latter two are finished, they can be crossed out as in Figure 6.13.

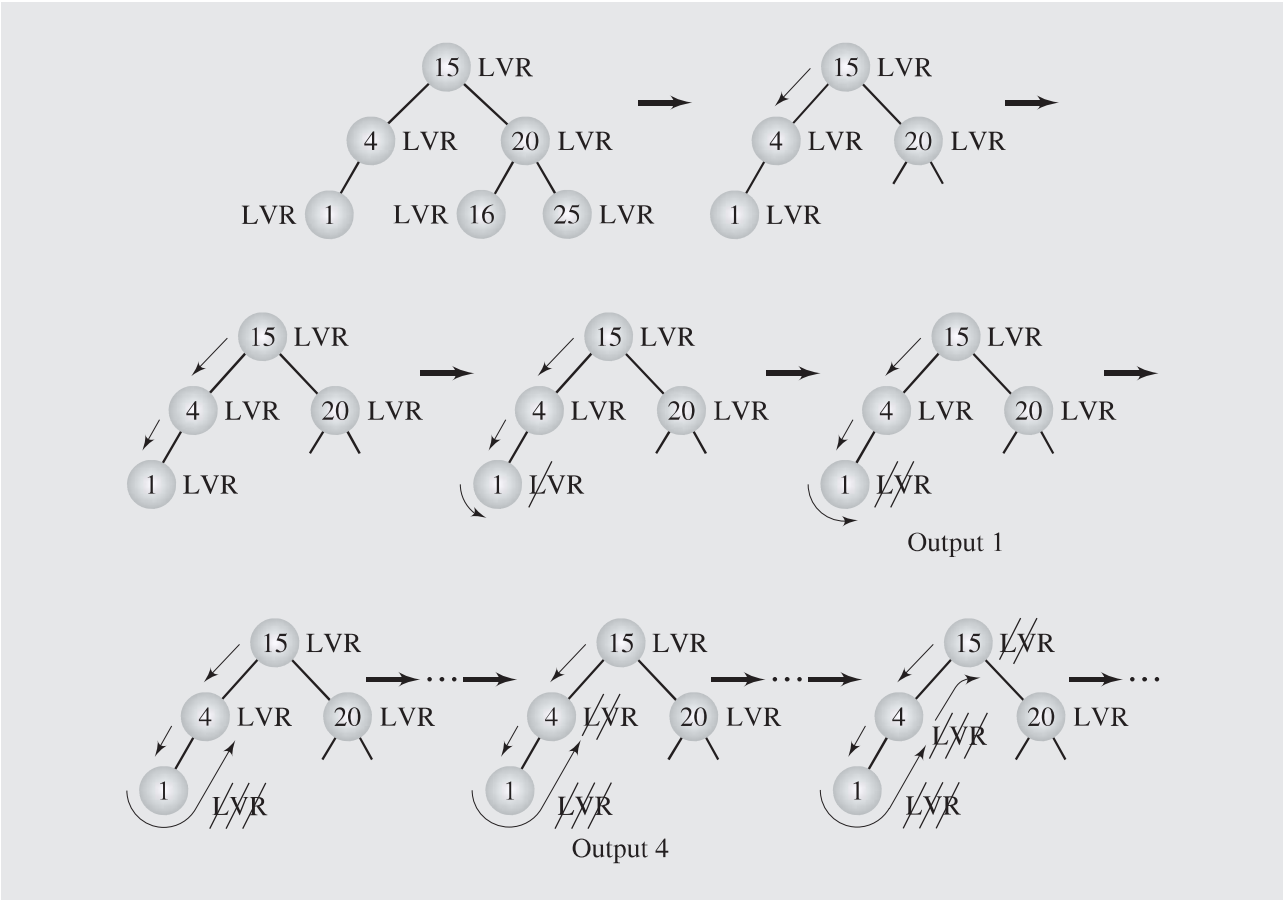
To present the way `inorder()` works, the behavior of the run-time stack is observed. The numbers in parentheses in Figure 6.14 indicate return addresses shown on the left-hand side of the code for `inorder()`.

```
        template<class T>
        void BST<T>::inorder(BSTnode<T> *node) {
             if (node != 0) {
/* 1 */          inorder(node->left);
/* 2 */          visit(node);
/* 3 */          inorder(node->right);
/* 4 */     }
        }
```

A rectangle with an up arrow and a number indicates the current value of `node` pushed onto the stack. For example, ↑4 means that `node` points to the node of the tree whose value is the number 4. Figure 6.14 shows the changes of the run-time stack when `inorder()` is executed for the tree in Figure 6.12.

(a) Initially, the run-time stack is empty (or rather it is assumed that the stack is empty by disregarding what has been stored on it before the first call to `inorder()`).

FIGURE 6.13 Details of several of the first steps of inorder traversal.



(b) Upon the first call, the return address of `inorder()` and the value of `node`, ↑15, are pushed onto the run-time stack. The tree, pointed to by `node`, is not empty, the condition in the `if` statement is satisfied, and `inorder()` is called again with node 4.

(c) Before it is executed, the return address, (2), and current value of `node`, ↑4, are pushed onto the stack. Because `node` is not null, `inorder()` is about to be invoked for `node`'s left child, ↑1.

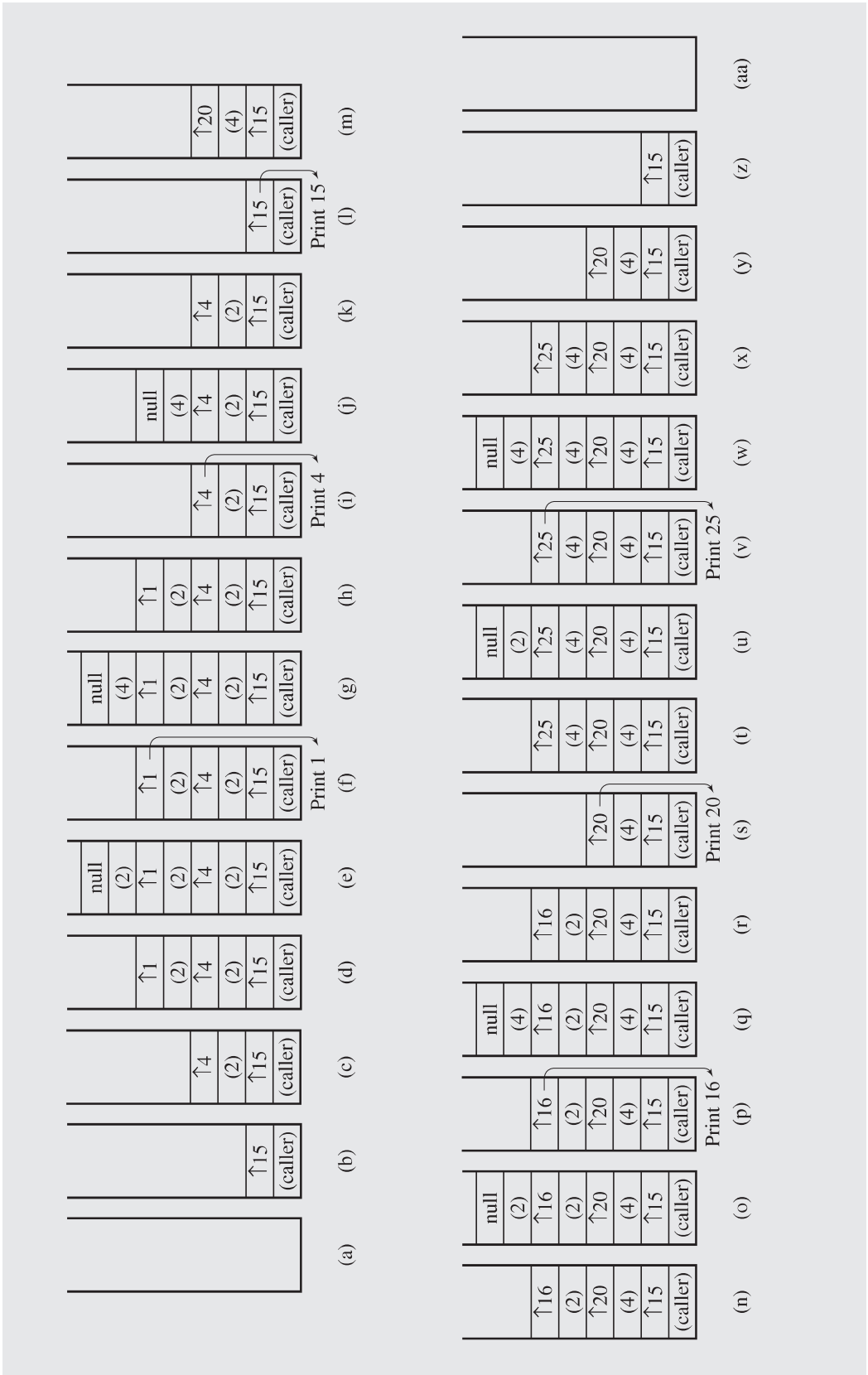
(d) First, the return address, (2), and the `node`'s value are stored on the stack.

(e) `inorder()` is called with node 1's left child. The address (2) and the current value of parameter `node`, null, are stored on the stack. Because `node` is null, `inorder()` is exited immediately; upon exit, the activation record is removed from the stack.

(f) The system goes now to its run-time stack, restores the value of the `node`, ↑1, executes the statement under (2), and prints the number 1. Because `node` is not completely processed, the value of `node` and address (2) are still on the stack.

(g) With the right child of `node` ↑1, the statement under (3) is executed, which is the next call to `inorder()`. First, however, the address (4) and `node`'s current value, null, are pushed onto the stack. Because `node` is null, `inorder()` is exited; upon exit, the stack is updated.

FIGURE 6.14 Changes in the run-time stack during inorder traversal.



(h) The system now restores the old value of the `node`, ↑1, and executes statement (4).

(i) Because this is `inorder()`'s exit, the system removes the current activation record and refers again to the stack; restores the `node`'s value, ↑4; and resumes execution from statement (2). This prints the number 4 and then calls `inorder()` for the right child of `node`, which is null.

These steps are just the beginning. All of the steps are shown in Figure 6.14.

At this point, consider the problem of a nonrecursive implementation of the three traversal algorithms. As indicated in Chapter 5, a recursive implementation has a tendency to be less efficient than a nonrecursive counterpart. If two recursive calls are used in a function, then the problem of possible inefficiency doubles. Can recursion be eliminated from the implementation? The answer has to be positive because if it is not eliminated in the source code, the system does it for us anyway. So the question should be rephrased: Is it expedient to do so?

Look first at a nonrecursive version of the preorder tree traversal shown in Figure 6.15. The function `iterativePreorder()` is twice as large as `preorder()`, but it is still short and legible. However, it uses a stack heavily. Therefore, supporting functions are necessary to process the stack, and the overall implementation is not so short. Although two recursive calls are omitted, there are now up to four calls per iteration of the `while` loop: up to two calls of `push()`, one call of `pop()`, and one call of `visit()`. This can hardly be considered an improvement in efficiency.

In the recursive implementations of the three traversals, note that the only difference is in the order of the lines of code. For example, in `preorder()`, first a node is

FIGURE 6.15 A nonrecursive implementation of preorder tree traversal.

```
template<class T>
void BST<T>::iterativePreorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    if (p != 0) {
        travStack.push(p);
        while (!travStack.empty()) {
            p = travStack.pop();
            visit(p);
            if (p->right != 0)
                 travStack.push(p->right);
            if (p->left != 0)   // left child pushed after right
                 travStack.push(p->left); // to be on the top of
        }                       // the stack;
    }
}
```

visited, and then there are calls for the left and right subtrees. On the other hand, in `postorder()`, visiting a node succeeds both calls. Can we so easily transform the nonrecursive version of a left-to-right preorder traversal into a nonrecursive left-to-right postorder traversal? Unfortunately, no. In `iterativePreorder()`, visiting occurs before both children are pushed onto the stack. But this order does not really matter. If the children are pushed first and then the node is visited—that is, if `visit(p)` is placed after both calls to `push()`—the resulting implementation is still a preorder traversal. What matters here is that `visit()` has to follow `pop()`, and the latter has to precede both calls of `push()`. Therefore, nonrecursive implementations of inorder and postorder traversals have to be developed independently.

A nonrecursive version of postorder traversal can be obtained rather easily if we observe that the sequence generated by a left-to-right postorder traversal (an LRV order) is the same as the reversed sequence generated by a right-to-left preorder traversal (a VRL order). In this case, the implementation of `iterativePreorder()` can be adopted to create `iterativePostorder()`. This means that two stacks have to be used, one to visit each node in the reverse order after a right-to-left preorder traversal is finished. It is, however, possible to develop a function for postorder traversal that pushes onto the stack a node that has two descendants, once before traversing its left subtree and once before traversing its right subtree. An auxiliary pointer `q` is used to distinguish between these two cases. Nodes with one descendant are pushed only once, and leaves do not need to be pushed at all (Figure 6.16).

FIGURE 6.16 A nonrecursive implementation of postorder tree traversal.

```
template<class T>
void BST<T>::iterativePostorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T>* p = root, *q = root;
    while (p != 0) {
        for ( ; p->left != 0; p = p->left)
            travStack.push(p);
        while (p->right == 0 || p->right == q) {
            visit(p);
            q = p;
            if (travStack.empty())
                 return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

A nonrecursive inorder tree traversal is also a complicated matter. One possible implementation is given in Figure 6.17. In cases like this, we can clearly see the power of recursion: `iterativeInorder()` is almost unreadable, and without thorough explanation, it is not easy to determine the purpose of this function. On the other hand, recursive `inorder()` immediately demonstrates a purpose and logic. Therefore, `iterativeInorder()` can be defended in one case only: if it is shown that there is a substantial gain in execution time and that the function is called often in a program; otherwise, `inorder()` is preferable to its iterative counterpart.

FIGURE 6.17 A nonrecursive implementation of inorder tree traversal.

```
template<class T>
void BST<T>::iterativeInorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    while (p != 0) {
        while (p != 0) {          // stack the right child (if any)
            if (p->right)         // and the node itself when going
               travStack.push(p->right); // to the left;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop();      // pop a node with no left child
        while (!travStack.empty() && p->right == 0) { // visit it
            visit(p);             // and all nodes with no right
            p = travStack.pop();  // child;
        }
        visit(p);                 // visit also the first node with
        if (!travStack.empty())   // a right child (if any);
             p = travStack.pop();
        else p = 0;
    }
}
```

6.4.3 Stackless Depth-First Traversal

Threaded Trees

The traversal functions analyzed in the preceding section were either recursive or nonrecursive, but both kinds used a stack either implicitly or explicitly to store information about nodes whose processing has not been finished. In the case of recursive functions, the run-time stack was utilized. In the case of nonrecursive variants, an explicitly defined and user-maintained stack was used. The concern is that some additional time has to be spent to maintain the stack, and some more space has to be set aside for the

stack itself. In the worst case, when the tree is unfavorably skewed, the stack may hold information about almost every node of the tree, a serious concern for very large trees.

It is more efficient to incorporate the stack as part of the tree. This is done by incorporating *threads* in a given node. Threads are pointers to the predecessor and successor of the node according to an inorder traversal, and trees whose nodes use threads are called *threaded trees.* Four pointers are needed for each node in the tree, which again takes up valuable space.

The problem can be solved by overloading existing pointers. In trees, left or right pointers are pointers to children, but they can also be used as pointers to predecessors and successors, thereby being overloaded with meaning. How do we distinguish these meanings? For an overloaded operator, context is always a disambiguating factor. In trees, however, a new data member has to be used to indicate the current meaning of the pointers.

Because a pointer can point to one node at a time, the left pointer is either a pointer to the left child or to the predecessor. Analogously, the right pointer points either to the right subtree or to the successor (Figure 6.18a).

FIGURE 6.18 (a) A threaded tree and (b) an inorder traversal's path in a threaded tree with right successors only.

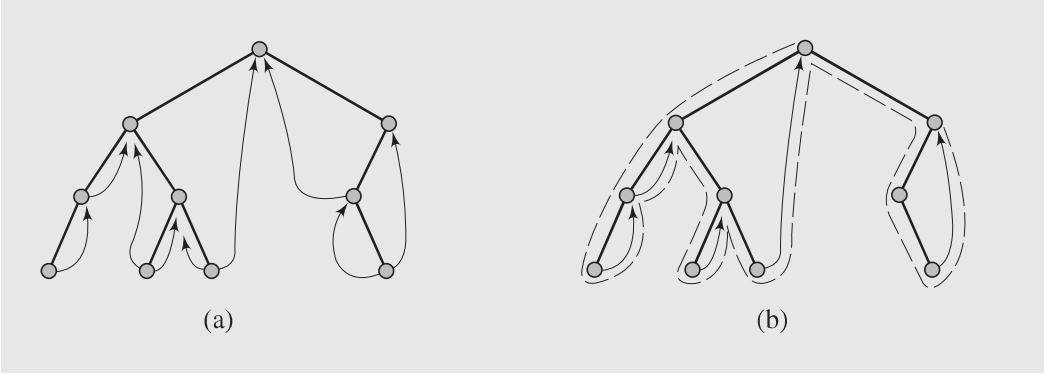


Figure 6.18a suggests that pointers to both predecessors and successors have to be maintained, which is not always the case. It may be sufficient to use only one thread, as shown in the implementation of the inorder traversal of a threaded tree, which requires only pointers to successors (Figure 6.18b).

The function is relatively simple. The dashed line in Figure 6.18b indicates the order in which `p` accesses nodes in the tree. Note that only one variable, `p`, is needed to traverse the tree. No stack is needed; therefore, space is saved. But is it really? As indicated, nodes require a data member indicating how the right pointer is being used. In the implementation of `threadedInorder()`, the Boolean data member `successor` plays this role, as shown in Figure 6.19. Hence, `successor` requires only one bit of computer memory, insignificant in comparison to other fields. However, the exact details are highly dependent on the implementation. The operating system almost certainly pads a bit structure with additional bits for proper alignment of machine words. If so, `successor` needs at least one byte, if not an entire word, defeating the argument about saving space by using threaded trees.

FIGURE 6.19 Implementation of the generic threaded tree and the inorder traversal of a threaded tree.

```
//***********************  genThreaded.h  *********************
//              Generic binary search threaded tree

#ifndef THREADED_TREE
#define THREADED_TREE

template<class T>
class ThreadedNode {
public:
    ThreadedNode() {
        left = right = 0;
    }
    ThreadedNode(const T& e, ThreadedNode *l = 0, ThreadedNode *r = 0) {
        el = e; left = l; right = r; successor = 0;
    }
    T el;
    ThreadedNode *left, *right;
    unsigned int successor : 1;
};

template<class T>
class ThreadedTree {
public:
    ThreadedTree()  {
        root = 0;
    }
    void insert(const T&);                  // Figure 6.24
    void inorder();
   . . . . . .
protected:
    ThreadedNode<T>* root;
    . . . . . .
};

#endif
```

Continues

FIGURE 6.19 (*continued*)

```
template<class T>
void ThreadedTree<T>::inorder() {
    ThreadedNode<T> *prev, *p = root;
    if (p != 0) {                // process only nonempty trees;
        while (p->left != 0)     // go to the leftmost node;
            p = p->left;
        while (p != 0) {
            visit(p);
            prev = p;
            p = p->right;        // go to the right node and only
            if (p != 0 && prev->successor == 0) // if it is a
                while (p->left != 0)// descendant go to the
                    p = p->left; // leftmost node, otherwise
        }                        // visit the successor;
    }
}
```

Threaded trees can also be used for preorder and postorder traversals. In preorder traversal, the current node is visited first and then traversal continues with its left descendant, if any, or right descendant, if any. If the current node is a leaf, threads are used to go through the chain of its already visited inorder successors to restart traversal with the right descendant of the last successor.

Postorder traversal is only slightly more complicated. First, a dummy node is created that has the root as its left descendant. In the traversal process, a variable can be used to check the type of the current action. If the action is left traversal and the current node has a left descendant, then the descendant is traversed; otherwise, the action is changed to right traversal. If the action is right traversal and the current node has a right nonthread descendant, then the descendant is traversed and the action is changed to left traversal; otherwise, the action changes to visiting a node. If the action is visiting a node, then the current node is visited, and afterward, its postorder successor has to be found. If the current node's parent is accessible through a thread (that is, current node is parent's left child), then traversal is set to continue with the right descendant of the parent. If the current node has no right descendant, then it is the end of the right-extended chain of nodes. First, the beginning of the chain is reached through the thread of the current node, then the right references of nodes in the chain are reversed, and finally, the chain is scanned backward, each node is visited, and then right references are restored to their previous setting.

Traversal through Tree Transformation

The first set of traversal algorithms analyzed earlier in this chapter needed a stack to retain some information necessary for successful processing. Threaded trees incorporated a stack as part of the tree at the cost of extending the nodes by one field to make a distinction between the interpretation of the right pointer as a pointer to the child

or to the successor. Two such tag fields are needed if both successor and predecessor are considered. However, it is possible to traverse a tree without using any stack or threads. There are many such algorithms, all of them made possible by making temporary changes in the tree during traversal. These changes consist of reassigning new values to some pointers. However, the tree may temporarily lose its tree structure, which needs to be restored before traversal is finished. The technique is illustrated by an elegant algorithm devised by Joseph M. Morris applied to inorder traversal.

First, it is easy to notice that inorder traversal is very simple for degenerate trees, in which no node has a left child (see Figure 6.1e). No left subtree has to be considered for any node. Therefore, the usual three steps, LVR (visit left subtree, visit node, visit right subtree), for each node in inorder traversal turn into two steps, VR. No information needs to be retained about the current status of the node being processed before traversing its left child, simply because there is no left child. Morris's algorithm takes into account this observation by temporarily transforming the tree so that the node being processed has no left child; hence, this node can be visited and its right subtree processed. The algorithm can be summarized as follows:

```
MorrisInorder()
    while not finished
      if node has no left descendant
            visit it;
            go to the right;
      else make this node the right child of the rightmost node in its left descendant;
            go to this left descendant;
```

This algorithm successfully traverses the tree, but only once, because it destroys its original structure. Therefore, some information has to be retained to allow the tree to restore its original form. This is achieved by retaining the left pointer of the node moved down its right subtree, as in the case of nodes 10 and 5 in Figure 6.21.

An implementation of the algorithm is shown in Figure 6.20, and the details of the execution are illustrated in Figure 6.21. The following description is divided into actions performed in consecutive iterations of the outer `while` loop:

1. Initially, `p` points to the root, which has a left child. As a result, the inner `while` loop takes `tmp` to node 7, which is the rightmost node of the left child of node 10, pointed to by `p` (Figure 6.21a). Because no transformation has been done, `tmp` has no right child, and in the inner `if` statement, the root, node 10, is made the right child of `tmp`. Node 10 retains its left pointer to node 5, its original left child. Now, the tree is not a tree anymore, because it contains a cycle (Figure 6.21b). This completes the first iteration.
2. Pointer `p` points to node 5, which also has a left child. First, `tmp` reaches the largest node in this subtree, which is 3 (Figure 6.21c), and then the current root, node 5, becomes the right child of node 3 while retaining contact with node 3 through its left pointer (Figure 6.21d).
3. Because node 3, pointed to by `p`, has no left child, in the third iteration, this node is visited, and `p` is reassigned to its right child, node 5 (Figure 6.21e).
4. Node 5 has a nonnull left pointer, so `tmp` finds a temporary parent of node 5, which is the same node currently pointed to by `tmp` (Figure 6.21f). Next, node 5 is visited, and

FIGURE 6.20 Implementation of the Morris algorithm for inorder traversal.

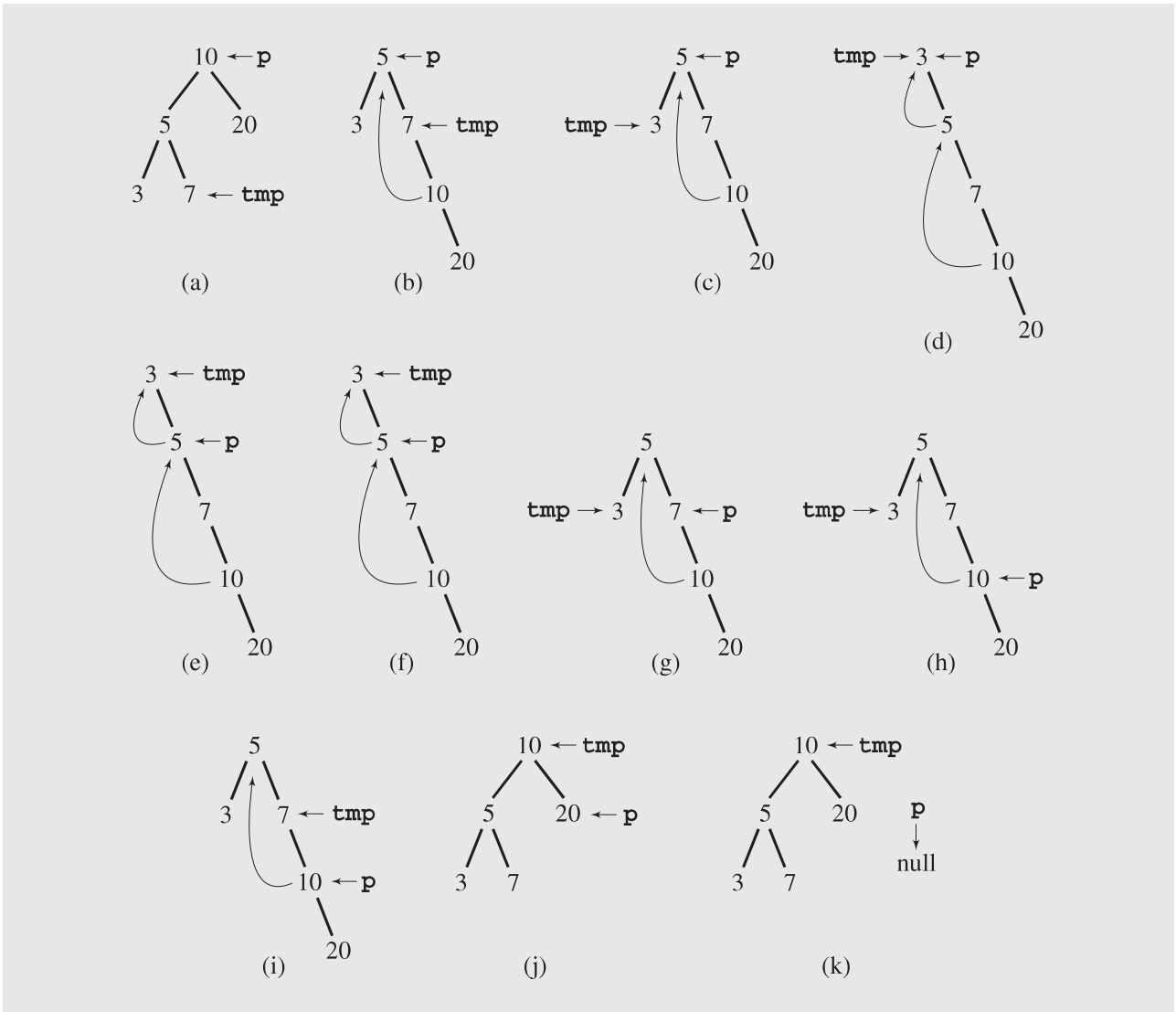
```
template<class T>
void BST<T>::MorrisInorder() {
    BSTNode<T> *p = root, *tmp;
    while (p != 0)
        if (p->left == 0) {
            visit(p);
            p = p->right;
        }
        else {
            tmp = p->left;
            while (tmp->right != 0 && // go to the rightmost node
                   tmp->right != p)   // of the left subtree or
                tmp = tmp->right;     // to the temporary parent
            if (tmp->right == 0) {    // of p; if 'true'
                tmp->right = p;       // rightmost node was
                p = p->left;          // reached, make it a
            }                         // temporary parent of the
            else {                    // current root, else
                                      // a temporary parent has
                visit(p);             // been found; visit node p
                tmp->right = 0;       // and then cut the right
                                      // pointer of the current
                p = p->right;         // parent, whereby it
            }                         // ceases to be a parent;
        }
}
```

configuration of the tree in Figure 6.21b is reestablished by setting the right pointer of node 3 to null (Figure 6.21g).

5. Node 7, pointed to now by `p`, is visited, and `p` moves down to its right child (Figure 6.21h).
6. `tmp` is updated to point to the temporary parent of node 10 (Figure 6.21i). Next, node 10 is visited and then reestablished to its status of root by nullifying the right pointer of node 7 (Figure 6.21j).
7. Finally, node 20 is visited without further ado, because it has no left child, nor has its position been altered.

This completes the execution of Morris's algorithm. Notice that there are seven iterations of the outer `while` loop for only five nodes in the tree in Figure 6.21. This is due to the fact that there are two left children in the tree, so the number of extra iterations depends on the number of left children in the entire tree. The algorithm performs worse for trees with a large number of such children.

FIGURE 6.21 Tree traversal with the Morris method.



Preorder traversal is easily obtainable from inorder traversal by moving `visit()` from the inner `else` clause to the inner `if` clause. In this way, a node is visited before a tree transformation.

Postorder traversal can also be obtained from inorder traversal by first creating a dummy node whose left descendant is the tree being processed and whose right descendant is null. Then this temporarily extended tree is the subject of traversal as in inorder traversal except that in the inner `else` clause, after finding a temporary parent, nodes between `p->left` (included) and `p` (excluded) extended to the right in a modified tree are processed in the reverse order. To process them in constant time, the chain of nodes is scanned down and right pointers are reversed to point to parents of nodes. Then the same chain is scanned upward, each node is visited, and the right pointers are restored to their original setting.

How efficient are the traversal procedures discussed in this section? All of them run in $\Theta(n)$ time, threaded implementation requires $\Theta(n)$ more space for threads than nonthreaded binary search trees, and both recursive and iterative traversals require $O(n)$ additional space (on the run-time stack or user-defined stack). Several dozens of runs on randomly generated trees of 5,000 nodes indicate that for preorder and inorder traversal routines (recursive, iterative, Morris, and threaded), the difference in the execution time is only on the order of 5–10%. Morris traversals have one undeniable advantage over other types of traversals: They do not require additional space. Recursive traversals rely on the run-time stack, which can be overflowed when traversing trees of large height. Iterative traversals also use a stack, and although the stack can be overflowed as well, the problem is not as imminent as in the case of the run-time stack. Threaded trees use nodes that are larger than the nodes used by nonthreaded trees, which usually should not pose a problem. But both iterative and threaded implementations are much less intuitive than their recursive counterparts; therefore, the clarity of implementation and comparable run time clearly favors, in most situations, recursive implementations over other implementations.

6.5 Insertion

Searching a binary tree does not modify the tree. It scans the tree in a predetermined way to access some or all of the keys in the tree, but the tree itself remains undisturbed after such an operation. Tree traversals can change the tree but they may also leave it in the same condition. Whether or not the tree is modified depends on the actions prescribed by `visit()`. There are certain operations that always make some systematic changes in the tree, such as adding nodes, deleting them, modifying elements, merging trees, and balancing trees to reduce their height. This section deals only with inserting a node into a binary search tree.

To insert a new node with key `el`, a tree node with a dead end has to be reached, and the new node has to be attached to it. Such a tree node is found using the same technique that tree searching used: the key `el` is compared to the key of a node currently being examined during a tree scan. If `el` is less than that key, the left child (if any) of `p` is tried; otherwise, the right child (if any) is tested. If the child of `p` to be tested is empty, the scanning is discontinued and the new node becomes this child. The procedure is illustrated in Figure 6.22. Figure 6.23 contains an implementation of the algorithm to insert a node.

In analyzing the problem of traversing binary trees, three approaches have been presented: traversing with the help of a stack, traversing with the aid of threads, and traversing through tree transformation. The first approach does not change the tree during the process. The third approach changes it, but restores it to the same condition as before it started. Only the second approach needs some preparatory operations on the tree to become feasible: it requires threads. These threads may be created each time before the traversal procedure starts its task and removed each time it is finished. If the traversal is performed infrequently, this becomes a viable option. Another approach is to maintain the threads in all operations on the tree when inserting a new element in the binary search tree.

FIGURE 6.22 Inserting nodes into binary search trees.

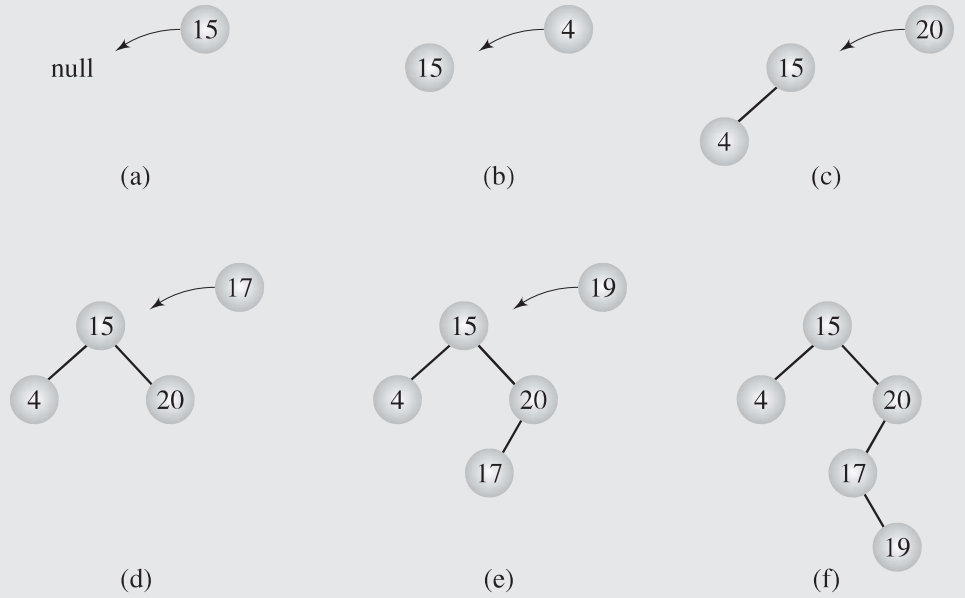


FIGURE 6.23 Implementation of the insertion algorithm.

```
template<class T>
void BST<T>::insert(const T& el) {
    BSTNode<T> *p = root, *prev = 0;
    while (p != 0) {  // find a place for inserting new node;
        prev = p;
        if (el < p->el)
             p = p->left;
        else p = p->right;
    }
    if (root == 0)    // tree is empty;
         root = new BSTNode<T>(el);
    else if (el < prev->el)
         prev->left   = new BSTNode<T>(el);
    else prev->right  = new BSTNode<T>(el);
}
```

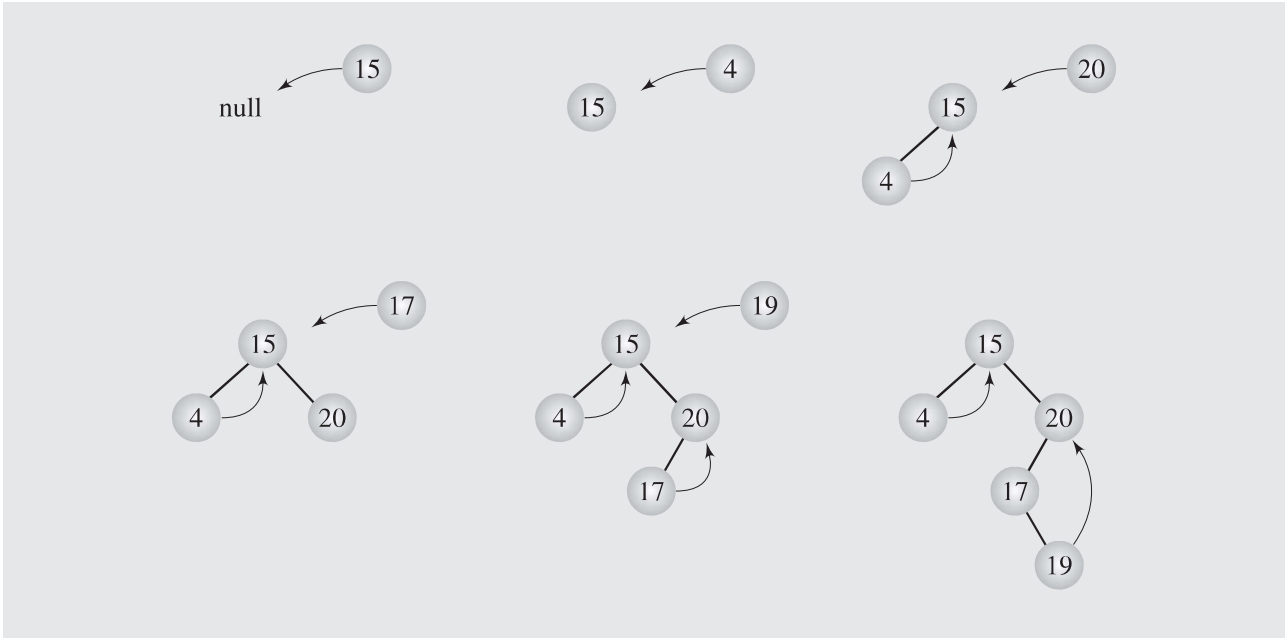
The function for inserting a node in a threaded tree is a simple extension of `insert()` for regular binary search trees to adjust threads whenever applicable. This function is for inorder tree traversal and it only takes care of successors, not predecessors.

A node with a right child has a successor some place in its right subtree. Therefore, it does not need a successor thread. Such threads are needed to allow climbing the tree, not going down it. A node with no right child has its successor somewhere above it. Except for one node, all nodes with no right children will have threads to their successors. If a node becomes the right child of another node, it inherits the successor from its new parent. If a node becomes a left child of another node, this parent becomes its successor. Figure 6.24 contains the implementation of this algorithm. The first few insertions are shown in Figure 6.25.

FIGURE 6.24 Implementation of the algorithm to insert a node into a threaded tree.

```
template<class T>
void ThreadedTree<T>::insert(const T& el) {
    ThreadedNode<T> *p, *prev = 0, *newNode;
    newNode = new ThreadedNode<T>(el);
    if (root == 0) {          // tree is empty;
        root = newNode;
        return;
     }
     p = root;                // find a place to insert newNode;
     while (p != 0) {
         prev = p;
         if (p->el > el)
              p = p->left;
         else if (p->successor == 0) // go to the right node only if it
              p = p->right;  // is a descendant, not a successor;
         else break;          // don't follow successor link;
     }
     if (prev->el > el) {     // if newNode is left child of
          prev->left  = newNode;    // its parent, the parent
          newNode->successor = 1;   // also becomes its successor;
          newNode->right = prev;
     }
     else if (prev->successor == 1) {// if the parent of newNode
          newNode->successor = 1;   // is not the rightmost node,
          prev->successor = 0;      // make parent's successor
          newNode->right = prev->right; // newNode's successor,
          prev->right = newNode;
     }
     else prev->right = newNode;    // otherwise it has no successor;
}
```

FIGURE 6.25 Inserting nodes into a threaded tree.

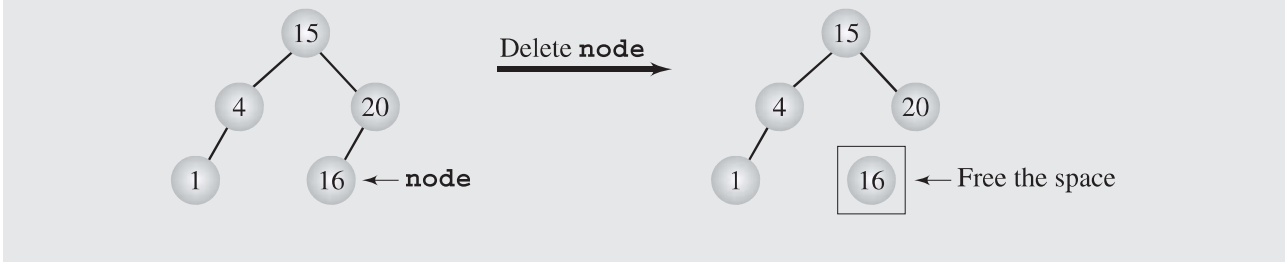


6.6 DELETION

Deleting a node is another operation necessary to maintain a binary search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. There are three cases of deleting a node from the binary search tree:

1. The node is a leaf; it has no children. This is the easiest case to deal with. The appropriate pointer of its parent is set to null and the node is disposed of by `delete` as in Figure 6.26.

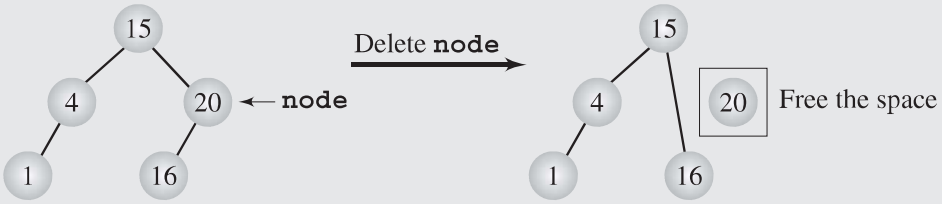
FIGURE 6.26 Deleting a leaf.



2. The node has one child. This case is not complicated. The parent's pointer to the node is reset to point to the node's child. In this way, the node's children are lifted up by one

level and all great-great-. . . grandchildren lose one "great" from their kinship designations. For example, the node containing 20 (see Figure 6.27) is deleted by setting the right pointer of its parent containing 15 to point to 20's only child, which is 16.

FIGURE 6.27 Deleting a node with one child.



3. The node has two children. In this case, no one-step operation can be performed because the parent's right or left pointer cannot point to both the node's children at the same time. This section discusses two different solutions to this problem.

6.6.1 Deletion by Merging

This solution makes one tree out of the two subtrees of the node and then attaches it to the node's parent. This technique is called *deleting by merging*. But how can we merge these subtrees? By the nature of binary search trees, every key of the right subtree is greater than every key of the left subtree, so the best thing to do is to find in the left subtree the node with the greatest key and make it a parent of the right subtree. Symmetrically, the node with the lowest key can be found in the right subtree and made a parent of the left subtree.

The desired node is the rightmost node of the left subtree. It can be located by moving along this subtree and taking right pointers until null is encountered. This means that this node will not have a right child, and there is no danger of violating the property of binary search trees in the original tree by setting that rightmost node's right pointer to the right subtree. (The same could be done by setting the left pointer of the leftmost node of the right subtree to the left subtree.) Figure 6.28 depicts this operation. Figure 6.29 contains the implementation of the algorithm.

It may appear that `findAndDeleteByMerging()` contains redundant code. Instead of calling `search()` before invoking `deleteByMerging()`, `findAndDeleteByMerging()` seems to forget about `search()` and searches for the node to be deleted using its private code. But using `search()` in function `findAndDeleteByMerging()` is a treacherous simplification. `search()` returns a pointer to the node containing `el`. In `findAndDeleteByMerging()`, it is important to have this pointer stored specifically in one of the pointers of the node's parent. In other words, a caller to `search()` is satisfied if it can access the node from any direction, whereas `findAndDeleteByMerging()` wants to access it from either its parent's left or right pointer data member. Otherwise, access to the entire subtree having this node as its root would be lost. One reason for this is the fact that `search()` focuses on the

FIGURE 6.28 Summary of deleting by merging.

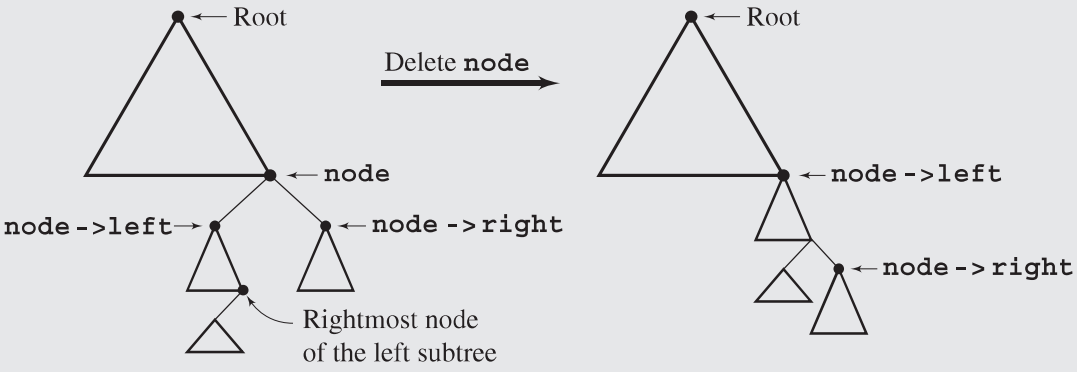


FIGURE 6.29 Implementation of an algorithm for deleting by merging.

```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (!node->right)           // node has no right child: its left
            node = node->left;      // child (if any) is attached to its
                                    // parent;
        else if (node->left == 0)   // node has no left child: its right
            node = node->right;     // child is attached to its parent;
        else {                      // be ready for merging subtrees;
            tmp = node->left;       // 1. move left
            while (tmp->right != 0)// 2. and then right as far as
                                    // possible;
               tmp = tmp->right;
            tmp->right =            // 3. establish the link between
               node->right;         //    the rightmost node of the left
                                    //    subtree and the right subtree;
            tmp = node;             // 4.
            node = node->left;      // 5.
        }
        delete tmp;                 // 6.
     }
}
```

Continues

FIGURE 6.29 *(continued)*

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
             break;
        prev = node;
        if (el < node->el)
             node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
         if (node == root)
              deleteByMerging(root);
         else if (prev->left == node)
              deleteByMerging(prev->left);
         else deleteByMerging(prev->right);
    else if (root != 0)
         cout << "element" << el << "is not in the tree\n";
    else cout << "the tree is empty\n";
}
```

node's key, and `findAndDeleteByMerging()` focuses on the node itself as an element of a larger structure, namely, a tree.

Figure 6.30 shows each step of this operation. It shows what changes are made when `findAndDeleteByMerging()` is executed. The numbers in this figure correspond to numbers put in comments in the code in Figure 6.29.

The algorithm for deletion by merging may result in increasing the height of the tree. In some cases, the new tree may be highly unbalanced, as Figure 6.31a illustrates. Sometimes the height may be reduced (see Figure 6.31b). This algorithm is not necessarily inefficient, but it is certainly far from perfect. There is a need for an algorithm that does not give the tree the chance to increase its height when deleting one of its nodes.

6.6.2 Deletion by Copying

Another solution, called *deletion by copying,* was proposed by Thomas Hibbard and Donald Knuth. If the node has two children, the problem can be reduced to one of two simple cases: the node is a leaf or the node has only one nonempty child. This can be done by replacing the key being deleted with its immediate predecessor (or successor). As already indicated in the algorithm deletion by merging, a key's predecessor is the key in the rightmost node in the left subtree (and analogically, its immediate successor is the key in the leftmost node in the right subtree). First, the predecessor has to be

FIGURE 6.30 Details of deleting by merging.

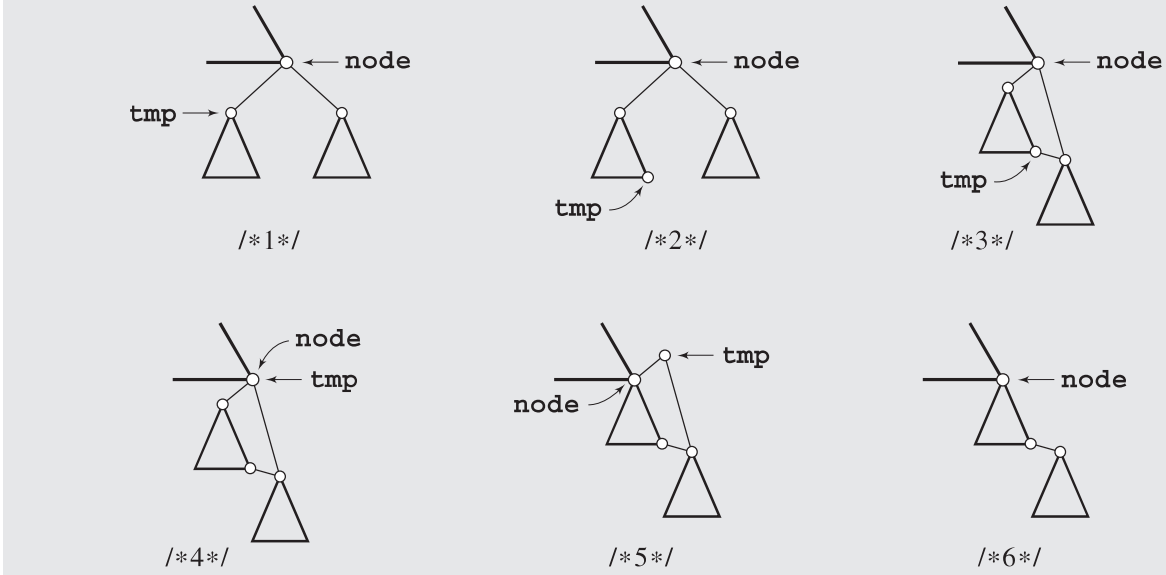
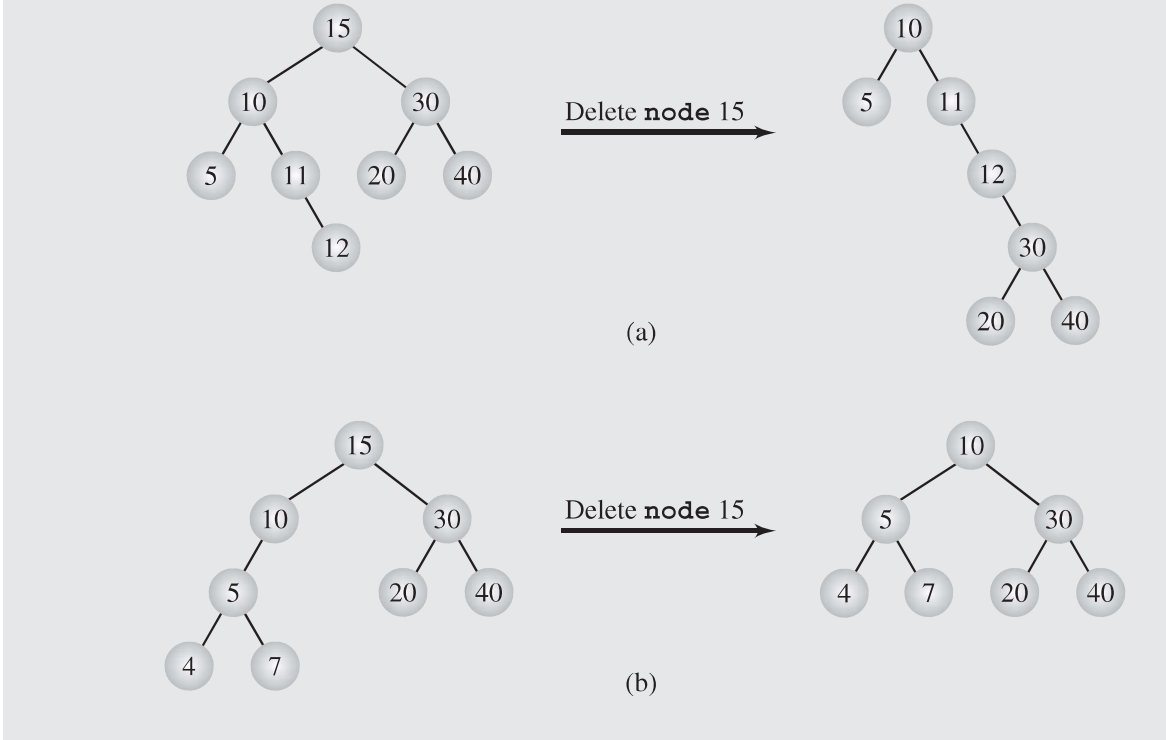


FIGURE 6.31 The height of a tree can be (a) extended or (b) reduced after deleting by merging.



located. This is done, again, by moving one step to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible. Next, the key of the located node replaces the key to be deleted. And that is where one of two simple cases comes into play. If the rightmost node is a leaf, the first case applies; however, if it has one child, the second case is relevant. In this way, deletion by copying removes a key k_1 by overwriting it by another key k_2 and then removing the node that holds k_2, whereas deletion by merging consisted of removing a key k_1 along with the node that holds it.

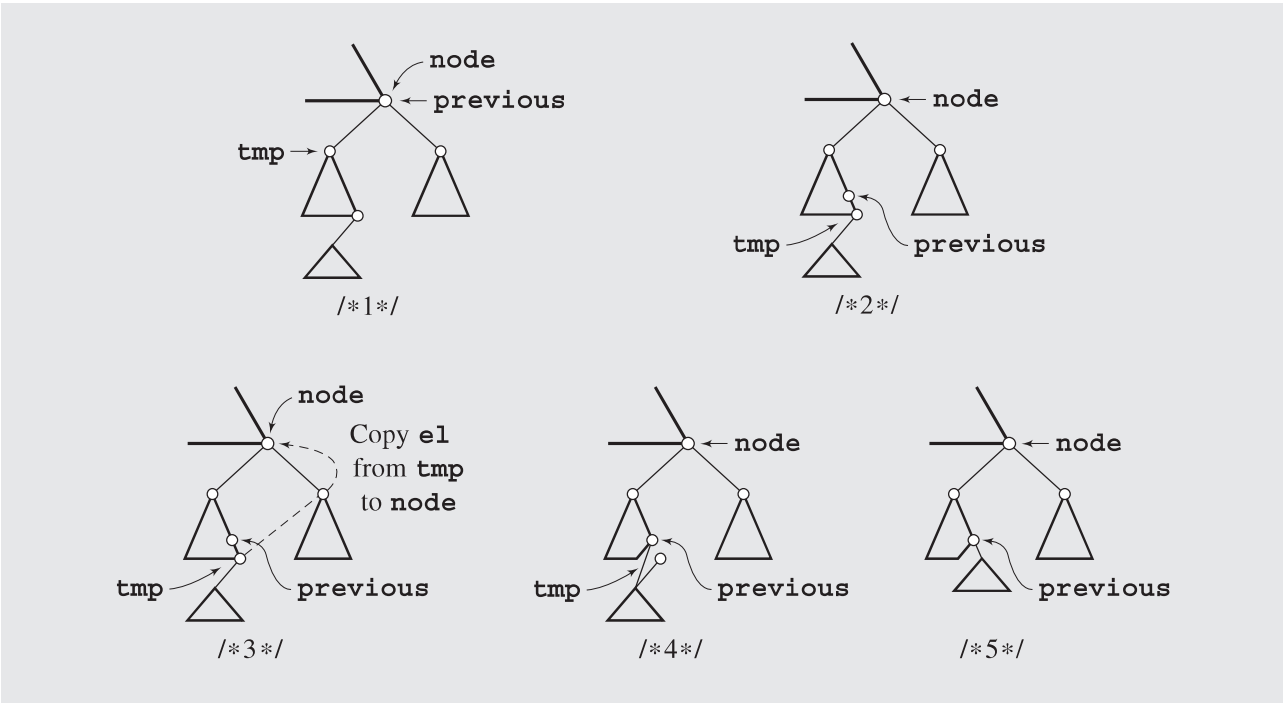
To implement the algorithm, two functions can be used. One function, `deleteByCopying()`, is illustrated in Figure 6.32. The second function, `findAndDeleteByCopying()`, is just like `findAndDeleteByMerging()`, but it calls `deleteByCopying()` instead of `deleteByMerging()`. A step-by-step trace is shown in Figure 6.33, and the numbers under the diagrams refer to the numbers indicated in comments included in the implementation of `deleteByCopying()`.

FIGURE 6.32 Implementation of an algorithm for deleting by copying.

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *tmp = node;
     if (node->right == 0)                    // node has no right child;
          node = node->left;
     else if (node->left == 0)                // node has no left child;
          node = node->right;
     else {
          tmp = node->left;                   // node has both children;
          previous = node;                    // 1.
          while (tmp->right != 0) {           // 2.
              previous = tmp;
              tmp = tmp->right;
          }
          node->el = tmp->el;                 // 3.
          if (previous == node)
               previous ->left  = tmp->left;
          else previous ->right = tmp->left;  // 4.
     }
     delete tmp;                              // 5.
}
```

This algorithm does not increase the height of the tree, but it still causes a problem if it is applied many times along with insertion. The algorithm is asymmetric; it always deletes the node of the immediate predecessor of the key in `node`, possibly reducing the height of the left subtree and leaving the right subtree unaffected. Therefore, the right subtree of `node` can grow after later insertions, and if the key

FIGURE 6.33 Deleting by copying.



in `node` is again deleted, the height of the right tree remains the same. After many insertions and deletions, the entire tree becomes right unbalanced, with the right subtree bushier and larger than the left subtree.

To circumvent this problem, a simple improvement can make the algorithm symmetrical. The algorithm can alternately delete the predecessor of the key in `node` from the left subtree and delete its successor from the right subtree. The improvement is significant. Simulations performed by Jeffrey Eppinger show that an expected internal path length for many insertions and asymmetric deletions is $\Theta(n \lg^3 n)$ for n nodes, and when symmetric deletions are used, the expected IPL becomes $\Theta(n \lg n)$. Theoretical results obtained by J. Culberson confirm these conclusions. According to Culberson, insertions and asymmetric deletions give $\Theta(n\sqrt{n})$ for the expected IPL and $\Theta(\sqrt{n})$ for the average search time (average path length), whereas symmetric deletions lead to $\Theta(\lg n)$ for the average search time, and as before, $\Theta(n \lg n)$ for the average IPL.

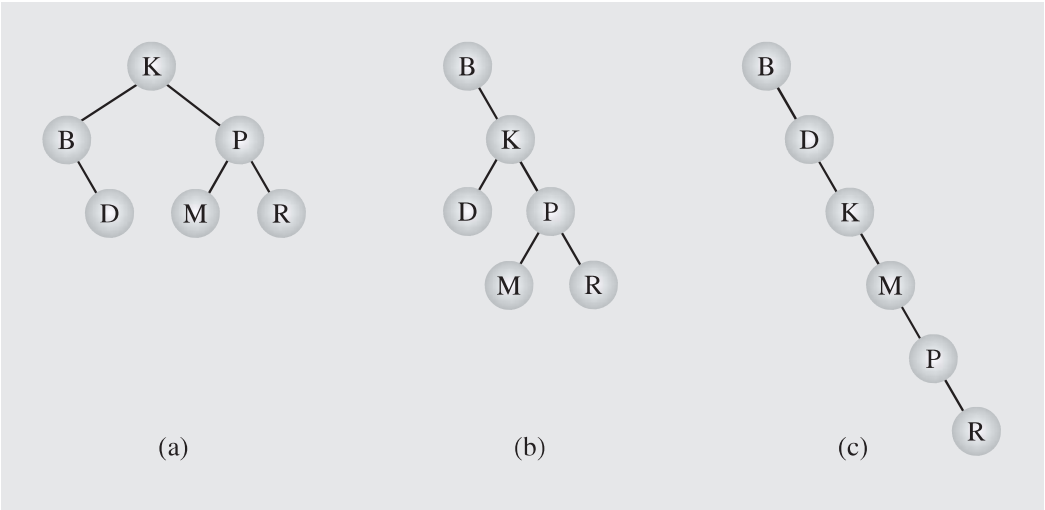
These results may be of moderate importance for practical applications. Experiments show that for a 2,048-node binary tree, only after 1.5 million insertions and asymmetric deletions does the IPL become worse than in a randomly generated tree.

Theoretical results are only fragmentary because of the extraordinary complexity of the problem. Arne Jonassen and Donald Knuth analyzed the problem of random insertions and deletions for a tree of only three nodes, which required using Bessel functions and bivariate integral equations, and the analysis turned out to rank among "the more difficult of all exact analyses of algorithms that have been carried out to date." Therefore, the reliance on experimental results is not surprising.

6.7 Balancing a Tree

At the beginning of this chapter, two arguments were presented in favor of trees: They are well suited to represent the hierarchical structure of a certain domain, and the search process is much faster using trees instead of linked lists. The second argument, however, does not always hold. It all depends on what the tree looks like. Figure 6.34 shows three binary search trees. All of them store the same data, but obviously, the tree in Figure 6.34a is the best and Figure 6.34c is the worst. In the worst case, three tests are needed in the former and six tests are needed in the latter to locate an object. The problem with the trees in Figures 6.34b and 6.34c is that they are somewhat unsymmetrical, or lopsided; that is, objects in the tree are not distributed evenly to the extent that the tree in Figure 6.34c practically turned into a linked list, although, formally, it is still a tree. Such a situation does not arise in balanced trees.

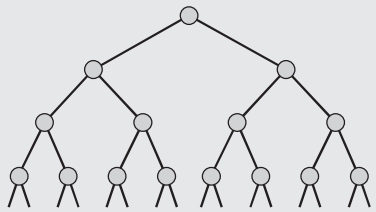
FIGURE 6.34 Different binary search trees with the same information.



A binary tree is *height-balanced* or simply *balanced* if the difference in height of both subtrees of any node in the tree is either zero or one. For example, for node *K* in Figure 6.34b, the difference between the heights of its subtrees being equal to one is acceptable. But for node *B* this difference is three, which means that the entire tree is unbalanced. For the same node *B* in 6.34c, the difference is the worst possible, namely, five. Also, a tree is considered *perfectly balanced* if it is balanced and all leaves are to be found on one level or two levels.

Figure 6.35 shows how many nodes can be stored in binary trees of different heights. Because each node can have two children, the number of nodes on a certain level is double the number of parents residing on the previous level (except, of course, the root). For example, if 10,000 elements are stored in a perfectly balanced tree, then the tree is of height $\lceil \lg(10{,}001) \rceil = \lceil 13.289 \rceil = 14$. In practical terms, this means that if 10,000 elements are stored in a perfectly balanced tree, then at most 14 nodes have to be checked to locate a particular element. This is a substantial difference compared

FIGURE 6.35 Maximum number of nodes in binary trees of different heights.



Height	Nodes at One Level	Nodes at All Levels
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
⋮		
11	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

to the 10,000 tests needed in a linked list (in the worst case). Therefore, it is worth the effort to build a balanced tree or modify an existing tree so that it is balanced.

There are a number of techniques to properly balance a binary tree. Some of them consist of constantly restructuring the tree when elements arrive and lead to an unbalanced tree. Some of them consist of reordering the data themselves and then building a tree, if an ordering of the data guarantees that the resulting tree is balanced. This section presents a simple technique of this kind.

The linked listlike tree of Figure 6.34c is the result of a particular stream of data. Thus, if the data arrive in ascending or descending order, then the tree resembles a linked list. The tree in Figure 6.34b is lopsided because the first element that arrived was the letter B, which precedes almost all other letters, except A; the left subtree of B is guaranteed to have just one node. The tree in Figure 6.34a looks very good, because the root contains an element near the middle of all the possible elements, and P is more or less in the middle of K and Z. This leads us to an algorithm based on binary search technique.

When data arrive, store all of them in an array. After all the data arrive, sort the array using one of the efficient algorithms discussed in Chapter 9. Now, designate for the root the middle element in the array. The array now consists of two subarrays: one between the beginning of the array and the element just chosen for the root and one between the root and the end of the array. The left child of the root is taken from the middle of the first subarray, its right child an element in the middle of the second subarray. Now, building the level containing the children of the root is finished. The next level, with children of children of the root, is constructed in the same fashion using four subarrays and the middle elements from each of them.

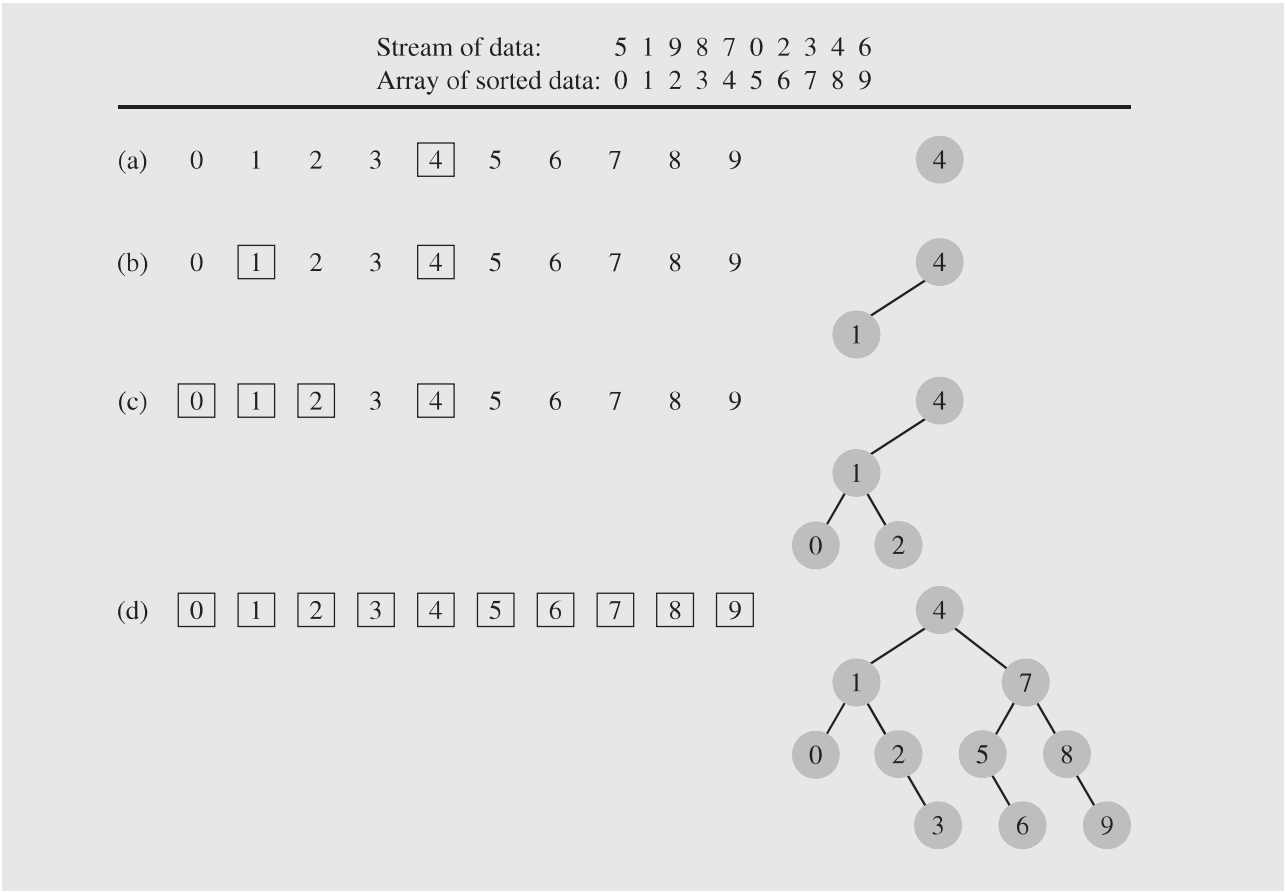
In this description, first the root is inserted into an initially empty tree, then its left child, then its right child, and so on level by level. An implementation of this algorithm is greatly simplified if the order of insertion is changed: first insert the root, then its left child, then the left child of this left child, and so on. This allows for using the following simple recursive implementation:

```
template<class T>
void BST<T>::balance(T data[], int first, int last) {
  if (first <= last) {
     int middle = (first + last)/2;
     insert(data[middle]);
     balance (data,first,middle-1);
     balance (data,middle+1,last);
  }
}
```

An example of the application of `balance()` is shown in Figure 6.36. First, number 4 is inserted (Figure 6.36a), then 1 (Figure 6.36b), then 0 and 2 (Figure 6.36c), and finally, 3, 7, 5, 6, 8, and 9 (Figure 6.36d).

This algorithm has one serious drawback: All data must be put in an array before the tree can be created. They can be stored in the array directly from the input. In this case, the algorithm may be unsuitable when the tree has to be used while the data to be included in the tree are still coming. But the data can be transferred from an unbalanced tree to the array using inorder traversal. The tree can now be deleted and recreated using `balance()`. This, at least, does not require using any sorting algorithm to put data in order.

FIGURE 6.36 Creating a binary search tree from an ordered array.



6.7.1 The DSW Algorithm

The algorithm discussed in the previous section was somewhat inefficient in that it required an additional array that needed to be sorted before the construction of a perfectly balanced tree began. To avoid sorting, it required deconstructing the tree after placing elements in the array using the inorder traversal and then reconstructing the tree, which is inefficient except for relatively small trees. There are, however, algorithms that require little additional storage for intermediate variables and use no sorting procedure. The very elegant DSW algorithm was devised by Colin Day and later improved by Quentin F. Stout and Bette L. Warren.

The building block for tree transformations in this algorithm is the *rotation* introduced by Adel'son-Vel'skii and Landis (1962). There are two types of rotation, left and right, which are symmetrical to one another. The right rotation of the node `Ch` about its parent `Par` is performed according to the following algorithm:

```
rotateRight(Gr, Par, Ch)
  if Par is not the root of the tree // i.e., if Gr is not null
    grandparent Gr of child Ch becomes Ch's parent;
  right subtree of Ch becomes left subtree of Ch's parent Par;
  node Ch acquires Par as its right child;
```

The steps involved in this compound operation are shown in Figure 6.37. The third step is the core of the rotation, when `Par`, the parent node of child `Ch`, becomes the child of `Ch`, when the roles of a parent and its child change. However, this exchange of roles cannot affect the principal property of the tree, namely, that it is a search tree. The first and the second steps of `rotateRight()` are needed to ensure that, after the rotation, the tree remains a search tree.

FIGURE 6.37 Right rotation of child `Ch` about parent `Par`.

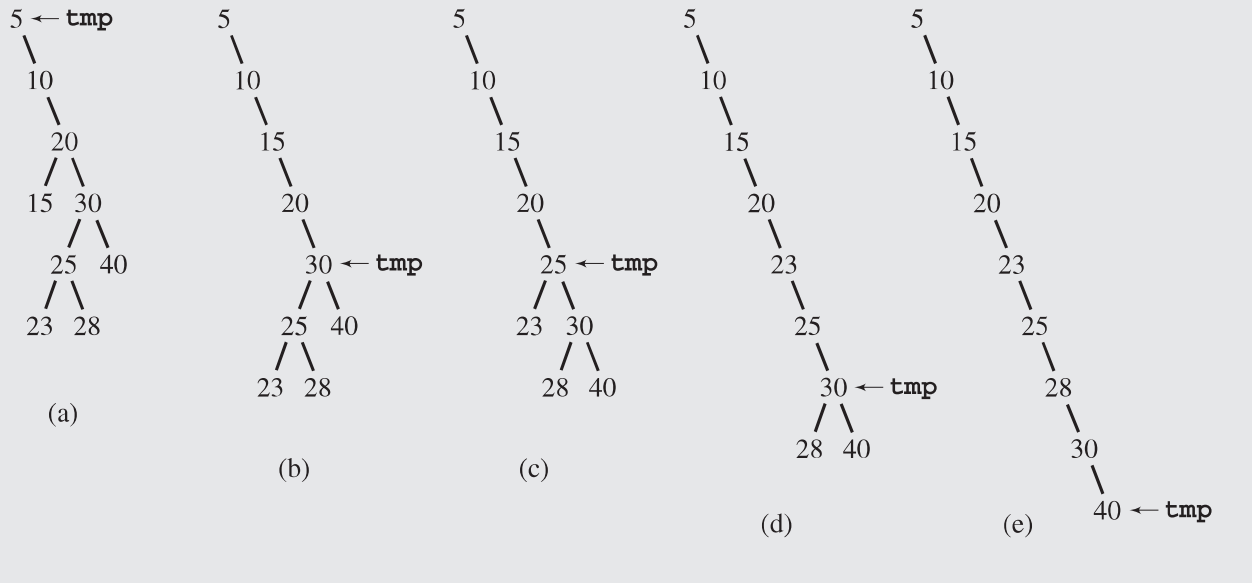
Basically, the DSW algorithm transfigures an arbitrary binary search tree into a linked listlike tree called a *backbone* or *vine*. Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

In the first phase, a backbone is created using the following routine:

```
createBackbone(root)
  tmp = root;
  while (tmp != 0)
    if tmp has a left child
        rotate this child about tmp; // hence the left child
                                     // becomes parent of tmp;
        set tmp to the child that just became parent;
    else set tmp to its right child;
```

This algorithm is illustrated in Figure 6.38. Note that a rotation requires knowledge about the parent of `tmp`, so another pointer has to be maintained when implementing the algorithm.

FIGURE 6.38 Transforming a binary search tree into a backbone.



In the best case, when the tree is already a backbone, the `while` loop is executed n times and no rotation is performed. In the worst case, when the root does not have a right child, the `while` loop executes $2n - 1$ times with $n - 1$ rotations performed, where n is the number of nodes in the tree; that is, the run time of the first phase is $O(n)$. In this case, for each node except the one with the smallest value, the left child of `tmp` is rotated about `tmp`. After all rotations are finished, `tmp` points to the root, and after n iterations, it descends down the backbone to become null.

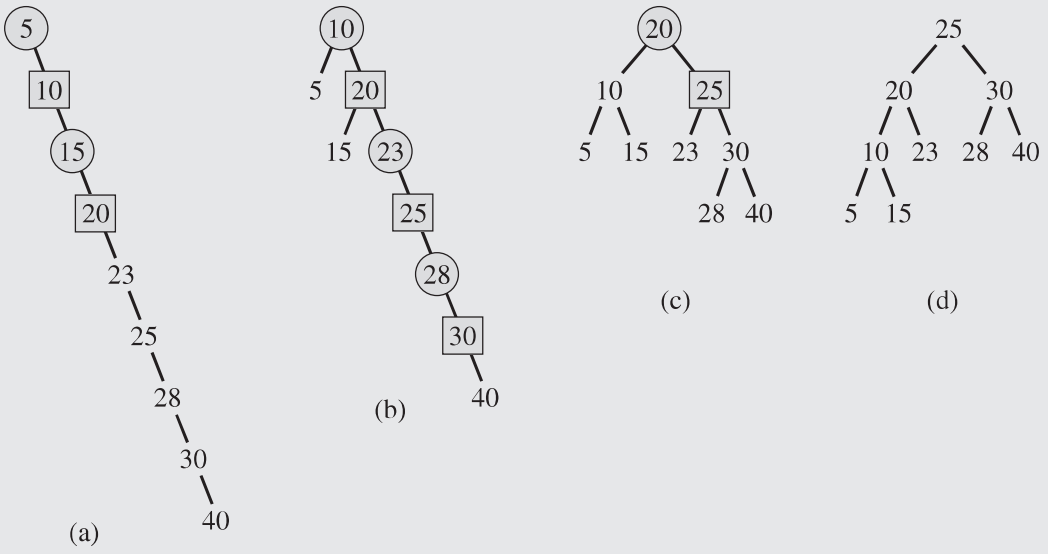
In the second phase, the backbone is transformed into a tree, but this time, the tree is perfectly balanced by having leaves only on two adjacent levels. In each pass down the backbone, every second node down to a certain point is rotated about its

parent. The first pass is used to account for the difference between the number n of nodes in the current tree and the number $2^{\lfloor \lg(n+1) \rfloor} - 1$ of nodes in the closest complete binary tree where $\lfloor x \rfloor$ is the closest integer less than x. That is, the overflowing nodes are treated separately.

```
createPerfectTree()
  n = number of nodes;
  m = 2^⌊lg(n+1)⌋-1;
  make n-m rotations starting from the top of backbone;
  while (m > 1)
    m = m/2;
    make m rotations starting from the top of backbone;
```

Figure 6.39 contains an example. The backbone in Figure 6.38e has nine nodes and is preprocessed by one pass outside the loop to be transformed into the backbone shown in Figure 6.39b. Now, two passes are executed. In each backbone, the nodes to be promoted by one level by left rotations are shown as squares; their parents, about which they are rotated, are circles.

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



To compute the complexity of the tree building phase, observe that the number of iterations performed by the `while` loop equals

$$(2^{\lg(m+1)-1} - 1) + \cdots + 15 + 7 + 3 + 1 = \sum_{i=1}^{\lg(m+1)-1} (2^i - 1) = m - \lg(m+1)$$

The number of rotations can now be given by the formula

$$n - m + (m - \lg(m+1)) = n - \lg(m+1) = n - \lfloor \lg(n+1) \rfloor$$

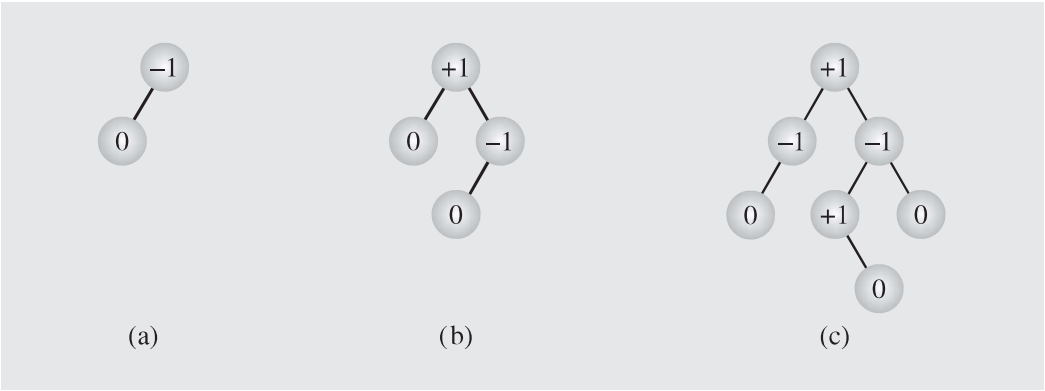
that is, the number of rotations is $O(n)$. Because creating a backbone also required at most $O(n)$ rotations, the cost of global rebalancing with the DSW algorithm is optimal in terms of time because it grows linearly with n and requires a very small and fixed amount of additional storage.

6.7.2 AVL Trees

The previous two sections discussed algorithms that rebalanced the tree globally; each and every node could have been involved in rebalancing either by moving data from nodes or by reassigning new values to pointers. Tree rebalancing, however, can be performed locally if only a portion of the tree is affected when changes are required after an element is inserted into or deleted from the tree. One classical method has been proposed by Adel'son-Vel'skii and Landis, which is commemorated in the name of the tree modified with this method: the AVL tree.

An *AVL tree* (originally called an *admissible tree*) is one in which the height of the left and right subtrees of every node differ by at most one. For example, all the trees in Figure 6.40 are AVL trees. Numbers in the nodes indicate the *balance factors* that are the differences between the heights of the left and right subtrees. A balance factor is the height of the right subtree minus the height of the left subtree. For an AVL tree, all balance factors should be +1, 0, or –1. Notice that the definition of the AVL tree is the same as the definition of the balanced tree. However, the concept of the AVL tree always implicitly includes the techniques for balancing the tree. Moreover, unlike the two methods previously discussed, the technique for balancing AVL trees does not guarantee that the resulting tree is perfectly balanced.

FIGURE 6.40 Examples of AVL trees.



The definition of an AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

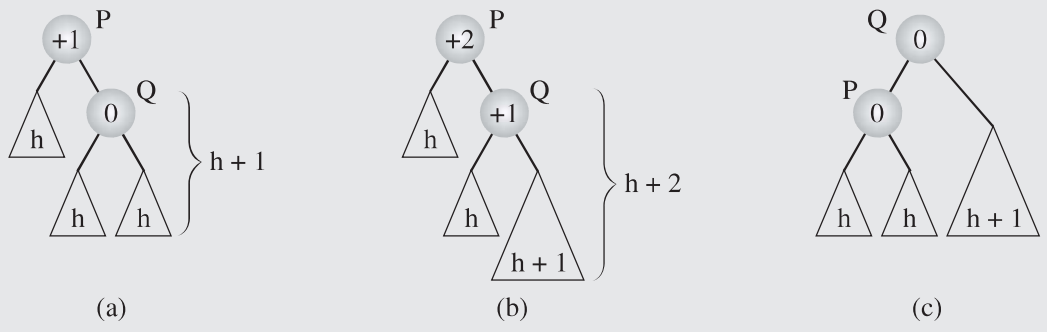
where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions.[1] This formula leads to the following bounds on the height h of an AVL tree depending on the number of nodes n (see Appendix A.5):

$$\lg(n + 1) \leq h < 1.44\lg(n + 2) - 0.328$$

Therefore, h is bounded by $O(\lg n)$; the worst case search requires $O(\lg n)$ comparisons. For a perfectly balanced binary tree of the same height, $h = \lceil \lg(n + 1) \rceil$. Therefore, the search time in the worst case in an AVL tree is 44% worse (it requires 44% more comparisons) than in the best case tree configuration. Empirical studies indicate that the average number of searches is much closer to the best case than to the worst and is equal to $\lg n + 0.25$ for large n (Knuth 1998). Therefore, AVL trees are definitely worth studying.

If the balance factor of any node in an AVL tree becomes less than –1 or greater than 1, the tree has to be balanced. An AVL tree can become out of balance in four situations, but only two of them need to be analyzed; the remaining two are symmetrical. The first case, the result of inserting a node in the right subtree of the right child, is illustrated in Figure 6.41. The heights of the participating subtrees are indicated within these subtrees. In the AVL tree in Figure 6.41a, a node is inserted somewhere in the right subtree of Q (Figure 6.41b), which disturbs the balance of the tree P. In this case, the problem can be easily rectified by rotating node Q about its parent P (Figure 6.41c) so that the balance factor of both P and Q becomes zero, which is even better than at the outset.

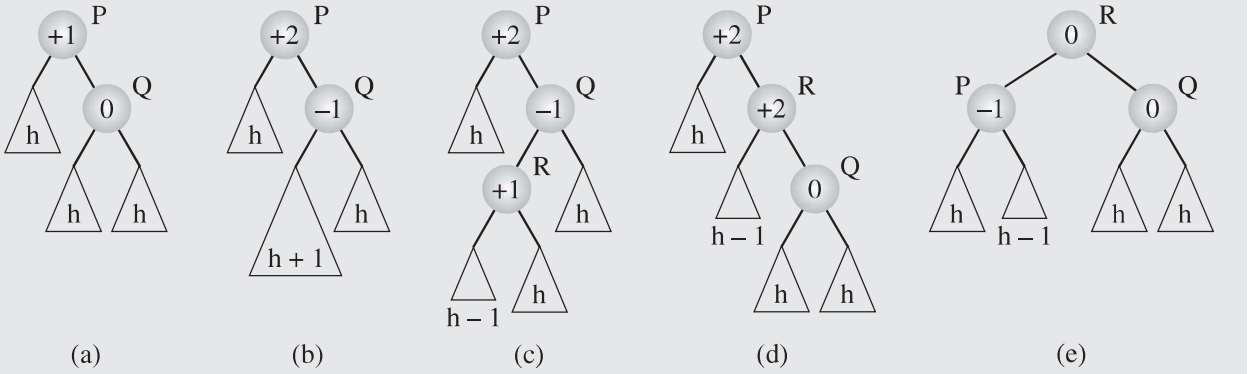
FIGURE 6.41 Balancing a tree after insertion of a node in the right subtree of node Q.



The second case, the result of inserting a node in the left subtree of the right child, is more complex. A node is inserted into the tree in Figure 6.42a; the resulting tree is shown in Figure 6.42b and in more detail in Figure 6.42c. Note that R's balance factor can also be –1. To bring the tree back into balance, a double rotation is performed. The balance of the tree P is restored by rotating R about node Q (Figure 6.42d) and then by rotating R again, this time about node P (Figure 6.42e).

[1]Numbers generated by this recurrence formula are called *Leonardo numbers.*

FIGURE 6.42 Balancing a tree after insertion of a node in the left subtree of node Q.



In these two cases, the tree *P* is considered a stand-alone tree. However, *P* can be part of a larger AVL tree; it can be a child of some other node in the tree. If a node is entered into the tree and the balance of *P* is disturbed and then restored, does extra work need to be done to the predecessor(s) of *P*? Fortunately not. Note that the heights of the trees in Figures 6.41c and 6.42e resulting from the rotations are the same as the heights of the trees before insertion (Figures 6.41a and 6.42a) and are equal to $h + 2$. This means that the balance factor of the parent of the new root (*Q* in Figure 6.41c and *R* in Figure 6.42e) remains the same as it was before the insertion, and the changes made to the subtree *P* are sufficient to restore the balance of the entire AVL tree. The problem is in finding a node *P* for which the balance factor becomes unacceptable after a node has been inserted into the tree.

This node can be detected by moving up toward the root of the tree from the position in which the new node has been inserted and by updating the balance factors of the nodes encountered. Then, if a node with a ±1 balance factor is encountered, the balance factor may be changed to ±2, and the first node whose balance factor is changed in this way becomes the root *P* of a subtree for which the balance has to be restored. Note that the balance factors do not have to be updated above this node because they remain the same.

To update the balance factors, the following algorithm can be used:

```
updateBalanceFactors()
   Q = the node just inserted;
   P = parent of Q;
   if Q is the left child of P
         P->balanceFactor--;
   else P->balanceFactor++;
   while P is not the root and P->balanceFactor ≠ ±2
       Q = P;
       P = P's parent;
       if Q->balanceFactor is 0
             return;
```

```
            if Q is the left child of P
                 P->balanceFactor--;
            else P->balanceFactor++;
      if P->balanceFactor is ±2
            rebalance the subtree rooted at P;
```

In Figure 6.43a, a path is marked with one balance factor equal to +1. Insertion of a new node at the end of this path results in an unbalanced tree (Figure 6.43b), and the balance is restored by one left rotation (Figure 6.43c).

However, if the balance factors on the path from the newly inserted node to the root of the tree are all zero, all of them have to be updated, but no rotation is needed for any of the encountered nodes. In Figure 6.44a, the AVL tree has a path of all zero balance factors. After a node has been appended to the end of this path (Figure 6.44b), no changes are made in the tree except for updating the balance factors of all nodes along this path.

FIGURE 6.43 An example of inserting (b) a new node in (a) an AVL tree, which requires one rotation (c) to restore the height balance.

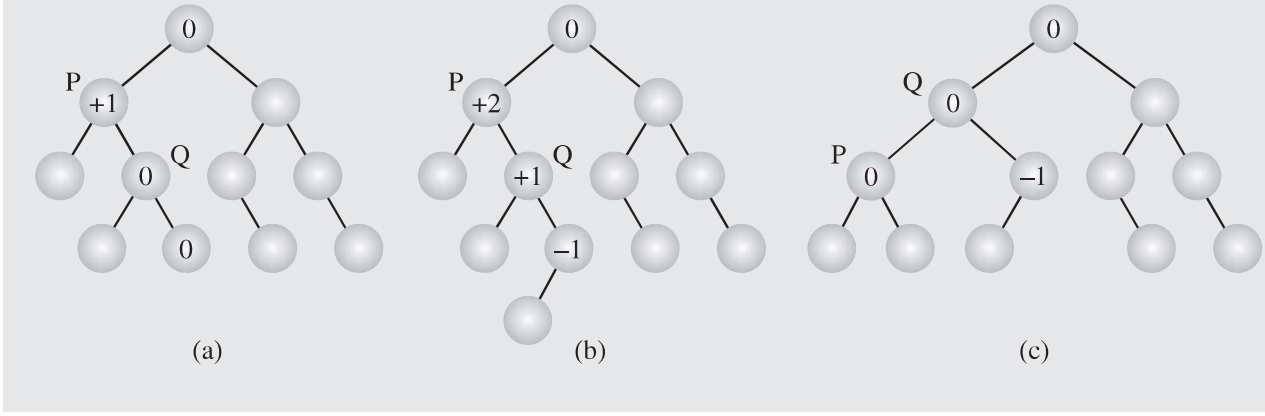
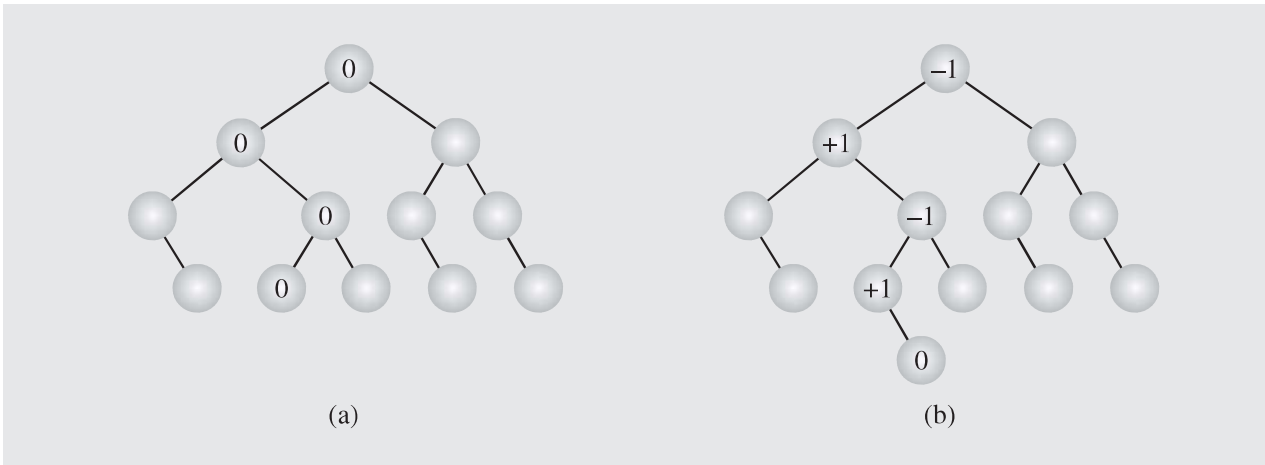


FIGURE 6.44 In an (a) AVL tree a (b) new node is inserted requiring no height adjustments.



Deletion may be more time-consuming than insertion. First, we apply `deleteByCopying()` to delete a node. This technique allows us to reduce the problem of deleting a node with two descendants to deleting a node with at most one descendant.

After a node has been deleted from the tree, balance factors are updated from the parent of the deleted node up to the root. For each node in this path whose balance factor becomes ±2, a single or double rotation has to be performed to restore the balance of the tree. Importantly, the rebalancing does not stop after the first node P is found for which the balance factor would become ±2, as is the case with insertion. This also means that deletion leads to at most $O(\lg n)$ rotations, because in the worst case, every node on the path from the deleted node to the root may require rebalancing.

Deletion of a node does not have to necessitate an immediate rotation because it may improve the balance factor of its parent (by changing it from ±1 to 0), but it may also worsen the balance factor for the grandparent (by changing it from ±1 to ±2). We illustrate only those cases that require immediate rotation. There are four such cases (plus four symmetric cases). In each of these cases, we assume that the left child of node P is deleted.

In the first case, the tree in Figure 6.45a turns, after deleting a node, into the tree in Figure 6.45b. The tree is rebalanced by rotating Q about P (Figure 6.45c). In the second case, P has a balance factor equal to +1, and its right subtree Q has a balance factor equal to 0 (Figure 6.45d). After deleting a node in the left subtree of P (Figure 6.45e), the tree is rebalanced by the same rotation as in the first case (Figure 6.45f). In this way, cases one and two can be processed together in an implementation after checking that the balance factor of Q is +1 or 0. If Q is –1, we have two other cases, which are more complex. In the third case, the left subtree R of Q has a balance factor equal to –1 (Figure 6.45g). To rebalance the tree, first R is rotated about Q and then about P (Figures 6.45h–i). The fourth case differs from the third in that R's balance factor equals +1 (Figure 6.45j), in which case the same two rotations are needed to restore the balance factor of P (Figures 6.45k–l). Cases three and four can be processed together in a program processing AVL trees.

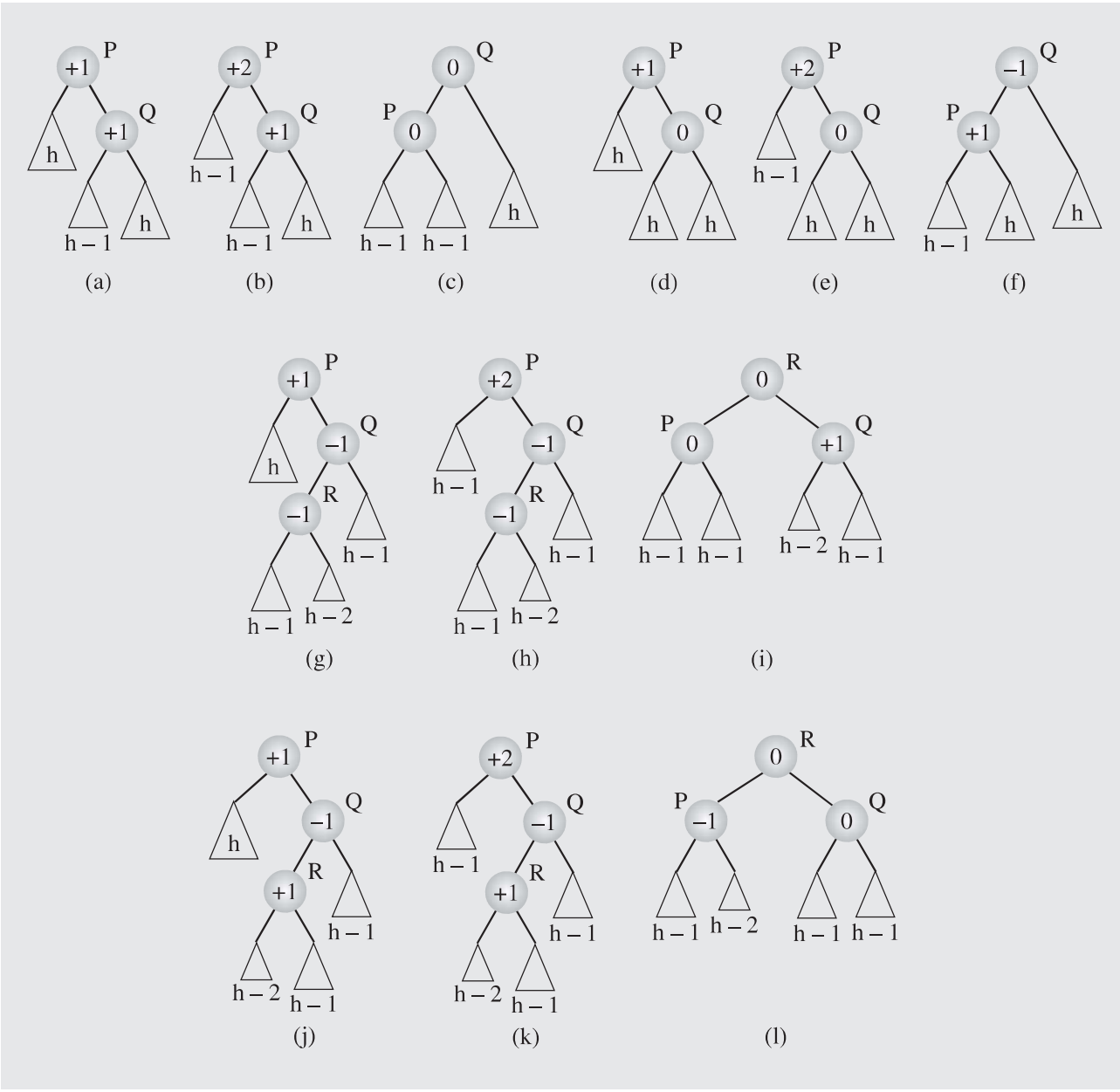
The previous analyses indicate that insertions and deletions require at most $1.44 \lg(n + 2)$ searches. Also, insertion can require one single or one double rotation, and deletion can require $1.44 \lg(n + 2)$ rotations in the worst case. But as also indicated, the average case requires $\lg(n) + .25$ searches, which reduces the number of rotations in case of deletion to this number. To be sure, insertion in the average case may lead to one single/double rotation. Experiments also indicate that deletions in 78% of cases require no rebalancing at all. On the other hand, only 53% of insertions do not bring the tree out of balance (Karlton et al. 1976). Therefore, the more time-consuming deletion occurs less frequently than the insertion operation, not markedly endangering the efficiency of rebalancing AVL trees.

AVL trees can be extended by allowing the difference in height $\Delta > 1$ (Foster 1973). Not unexpectedly, the worst-case height increases with Δ and

$$h = \begin{cases} 1.81 \lg(n) - 0.71 & \text{if } \Delta = 2 \\ 2.15 \lg(n) - 1.13 & \text{if } \Delta = 3 \end{cases}$$

As experiments indicate, the average number of visited nodes increases by one-half in comparison to pure AVL trees ($\Delta = 1$), but the amount of restructuring can be decreased by a factor of 10.

FIGURE 6.45 Rebalancing an AVL tree after deleting a node.



6.8 Self-Adjusting Trees

The main concern in balancing trees is to keep them from becoming lopsided and, ideally, to allow leaves to occur only at one or two levels. Therefore, if a newly arriving element endangers the tree balance, the problem is immediately rectified by restructuring the tree locally (the AVL method) or by re-creating the tree (the DSW method). However, we may question whether such a restructuring is always necessary. Binary search trees are used to insert, retrieve, and delete elements quickly, and the speed of performing these operations is the issue, not the shape of the tree. Performance can be improved by balancing the tree, but this is not the only method that can be used.

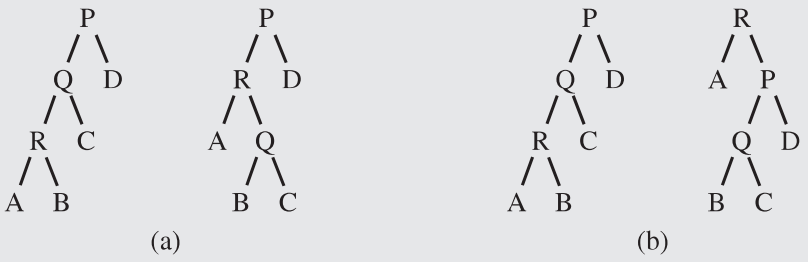
Another approach begins with the observation that not all elements are used with the same frequency. For example, if an element on the tenth level of the tree is used only infrequently, then the execution of the entire program is not greatly impaired by accessing this level. However, if the same element is constantly being accessed, then it makes a big difference whether it is on the tenth level or close to the root. Therefore, the strategy in self-adjusting trees is to restructure trees by moving up the tree only those elements that are used more often, creating a kind of "priority tree." The frequency of accessing nodes can be determined in a variety of ways. Each node can have a counter field that records the number of times the element has been used for any operation. Then the tree can be scanned to move the most frequently accessed elements toward the root. In a less sophisticated approach, it is assumed that an element being accessed has a good chance of being accessed again soon. Therefore, it is moved up the tree. No restructuring is performed for new elements. This assumption may lead to promoting elements that are occasionally accessed, but the overall tendency is to move up elements with a high frequency of access, and for the most part, these elements will populate the first few levels of the tree.

6.8.1 Self-Restructuring Trees

A strategy proposed by Brian Allen and Ian Munro and by James Bitner consists of two possibilities:

1. *Single rotation.* Rotate a child about its parent if an element in a child is accessed, unless it is the root (Figure 6.46a).
2. *Moving to the root.* Repeat the child–parent rotation until the element being accessed is in the root (Figure 6.46b).

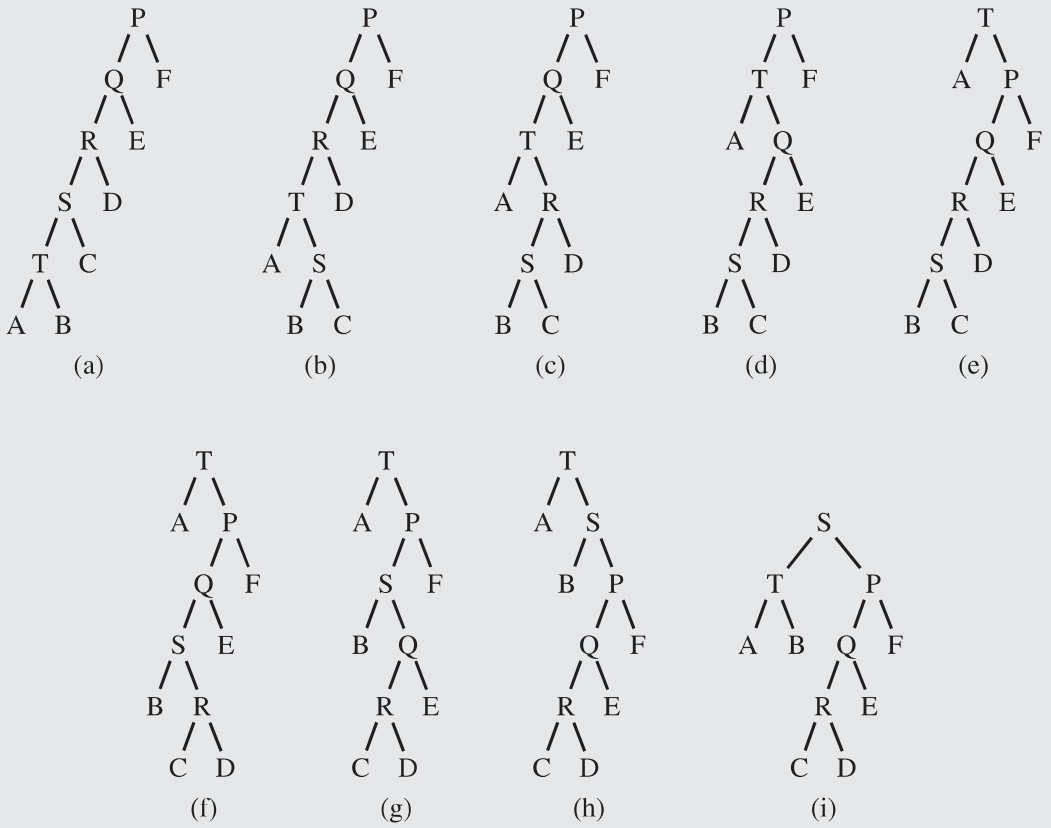
FIGURE 6.46 Restructuring a tree by using (a) a single rotation or (b) moving to the root when accessing node *R*.



Using the single-rotation strategy, frequently accessed elements are eventually moved up close to the root so that later accesses are faster than previous ones. In the move-to-the-root strategy, it is assumed that the element being accessed has a high probability to be accessed again, so it percolates right away up to the root. Even if it is not used in the next access, the element remains close to the root. These strategies, however, do not work very well in unfavorable situations, when the binary tree is elongated as in Figure 6.47. In this case, the shape of the tree improves slowly. Nevertheless, it has been determined that the cost of moving a node to the root converges to

the cost of accessing the node in an optimal tree times 2 ln 2; that is, it converges to (2 ln 2)lg n. The result holds for any probability distribution (that is, independently of the probability that a particular request is issued). However, the average search time when all requests are equally likely is, for the single rotation technique, equal to $\sqrt{\pi n}$.

FIGURE 6.47 (a–e) Moving element T to the root and then (e–i) moving element S to the root.



6.8.2 Splaying

A modification of the move-to-the-root strategy is called *splaying,* which applies single rotations in pairs in an order depending on the links between the child, parent, and grandparent (Sleator and Tarjan 1985). First, three cases are distinguished depending on the relationship between a node R being accessed and its parent Q and grandparent P (if any) nodes:

Case 1: Node R's parent is the root.

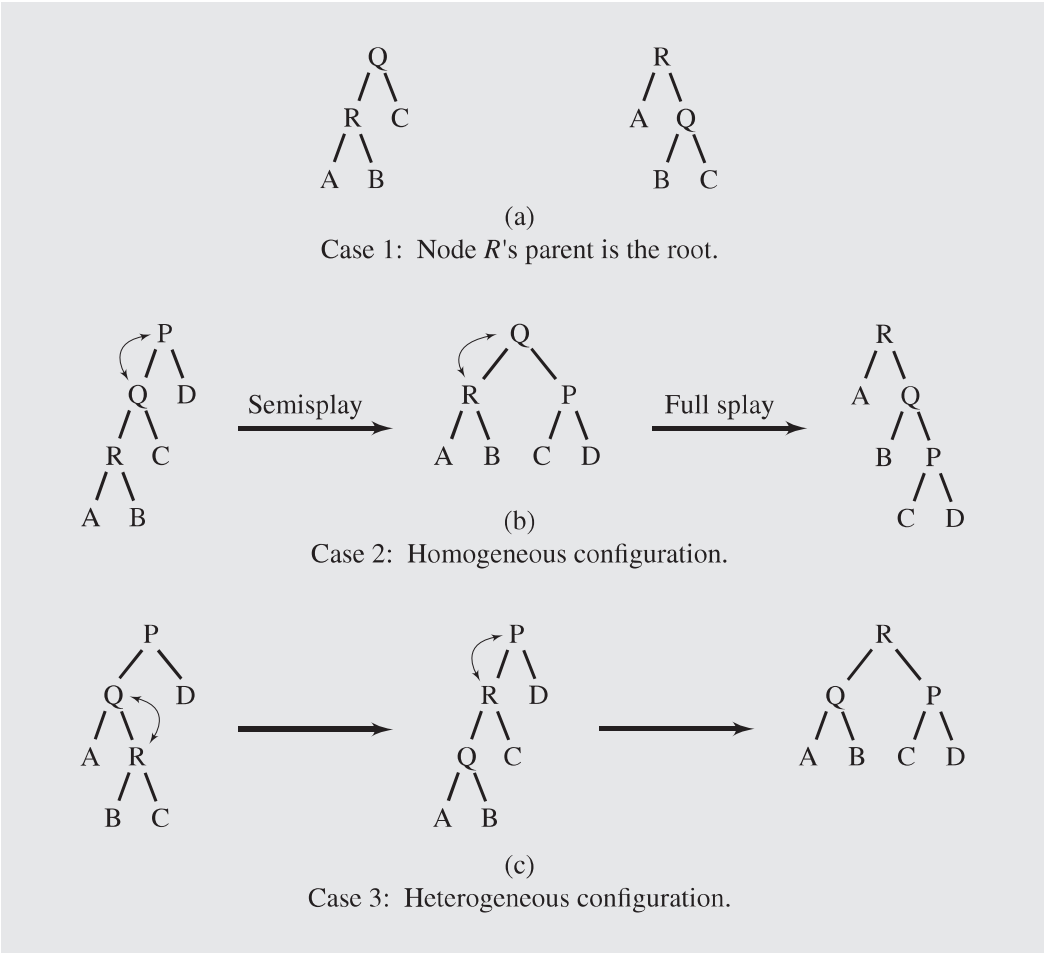
Case 2: *Homogeneous configuration.* Node R is the left child of its parent Q, and Q is the left child of its parent P, or R and Q are both right children.

Case 3: *Heterogeneous configuration.* Node R is the right child of its parent Q, and Q is the left child of its parent P, or R is the left child of Q, and Q is the right child of P.

The algorithm to move a node *R* being accessed to the root of the tree is as follows:

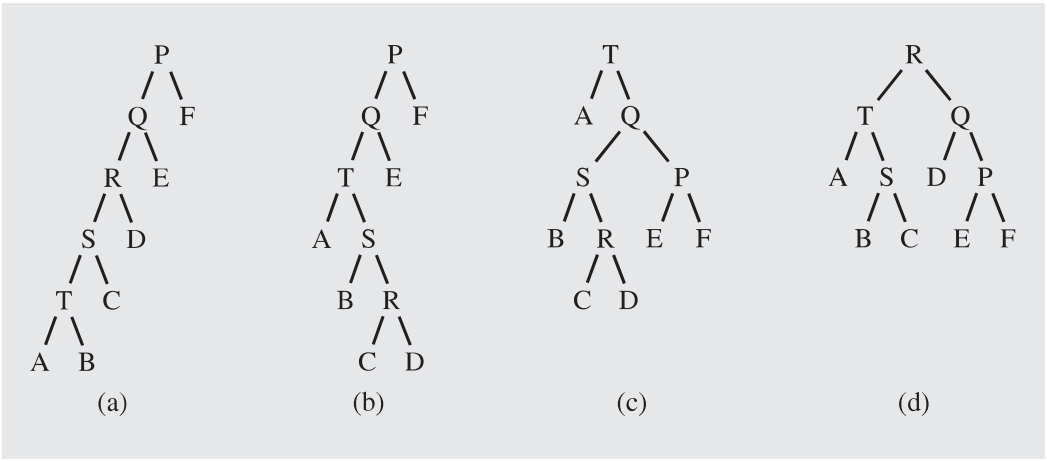
```
splaying(P,Q,R)
    while R is not the root
      if R's parent is the root
            perform a singular splay, rotate R about its parent (Figure 6.48a);
      else if R is in a homogeneous configuration with its predecessors
           perform a homogeneous splay, first rotate Q about P
           and then R about Q (Figure 6.48b);
      else // if R is in a heterogeneous configuration
           // with its predecessors
           perform a heterogeneous splay, first rotate R about Q
           and then about P (Figure 6.48c);
```

FIGURE 6.48 Examples of splaying.



The difference in restructuring a tree is illustrated in Figure 6.49, where the tree from Figure 6.47a is used to access node *T* located at the fifth level. The shape of the tree is immediately improved. Then, node *R* is accessed (Figure 6.49c) and the shape of the tree becomes even better (Figure 6.49d).

FIGURE 6.49 Restructuring a tree with splaying (a–c) after accessing *T* and (c–d) then *R*.



Although splaying is a combination of two rotations except when next to the root, these rotations are not always used in the bottom-up fashion, as in self-adjusting trees. For the homogeneous case (left-left or right-right), first the parent and the grandparent of the node being accessed are rotated, and only afterward are the node and its parent rotated. This has the effect of moving an element to the root and flattening the tree, which has a positive impact on the accesses to be made.

The number of rotations may seem excessive, and it certainly would be if an accessed element happened to be in a leaf every time. In the case of a leaf, the access time is usually $O(\lg n)$, except for some initial accesses when the tree is not balanced. But accessing elements close to the root may make the tree unbalanced. For example, in the tree in Figure 6.49a, if the left child of the root is always accessed, then eventually, the tree would also be elongated, this time extending to the right.

To establish the efficiency of accessing a node in a binary search tree that utilizes the splaying technique, an amortized analysis will be used.

Consider a binary search tree t. Let $nodes(x)$ be the number of nodes in the subtree whose root is x, $rank(x) = \lg(nodes(x))$, so that $rank(root(t)) = \lg(n)$, and $potential(t) = \sum_{x \text{ is a node of } t} rank(x)$. It is clear that $nodes(x) + 1 \leq nodes(parent(x))$; therefore, $rank(x) < rank(parent(x))$. Let the amortized cost of accessing node x be defined as the function

$$amCost(\mathrm{x}) = cost(x) + potential_s(t) - potential_0(t)$$

where $potential_s(t)$ and $potential_0(t)$ are the potentials of the tree before access takes place and after it is finished. It is very important to see that one rotation changes ranks of only the node x being accessed, its parent, and its grandparent. This is the reason for basing the definition of the amortized cost of accessing node x on the change in the potential of the tree, which amounts to the change of ranks of the nodes involved in splaying operations that promote x to the root. We can state now a lemma specifying the amortized cost of one access.

Access lemma (Sleator and Tarjan 1985). For the amortized time to splay the tree t at a node x,

$$amCost(x) < 3(\lg(n) - rank(x)) + 1$$

The proof of this conjecture is divided into three parts, each dealing with the different case indicated in Figure 6.48. Let $par(x)$ be a parent of x and $gpar(x)$ a grandparent of x (in Figure 6.48, $x = R$, $par(x) = Q$, and $gpar(x) = P$).

Case 1: One rotation is performed. This can be only the last splaying step in the sequence of such steps that move node x to the root of the tree t, and if there are a total of s splaying steps in the sequence, then the amortized cost of the last splaying step s is

$$amCost_s(x) = cost_s(x) + potential_s(t) - potential_{s-1}(t)$$
$$= 1 + (rank_s(x) - rank_{s-1}(x)) + (rank_s(par(x)) - rank_{s-1}(par(x)))$$

where $cost_s(x) = 1$ represents the actual cost, the cost of the one splaying step (which in this step is limited to one rotation); $potential_{s-1}(t) = rank_{s-1}(x) + rank_{s-1}(par(x)) + C$ and $potential_s(t) = rank_s(x) + rank_s(par(x)) + C$, because x and $par(x)$ are the only nodes whose ranks are modified. Now because $rank_s(x) = rank_{s-1}(par(x))$

$$amCost_s(x) = 1 - rank_{s-1}(x) + rank_s(par(x))$$

and because $rank_s(par(x)) < rank_s(x)$

$$amCost_s(x) < 1 - rank_{s-1}(x) + rank_s(\mathrm{x}).$$

Case 2: Two rotations are performed during a homogeneous splay. As before, number 1 represents the actual cost of one splaying step.

$$amCost_i(x) = 1 + (rank_i(x) - rank_{i-1}(x)) + (rank_i(par(x)) - rank_{i-1}(par(x))) + (rank_i(gpar(x)) - rank_{i-1}(gpar(x)))$$

Because $rank_i(x) = rank_{i-1}(gpar(x))$

$$amCost_i(x) = 1 - rank_{i-1}(x) + rank_i(par(x)) - rank_{i-1}(par(x)) + rank_i(gpar(x))$$

Because $rank_i(gpar(x)) < rank_i(par(x)) < rank_i(x)$

$$amCost_i(x) < 1 - rank_{i-1}(x) - rank_{i-1}(par(x)) + 2rank_i(x)$$

and because $rank_{i-1}(x) < rank_{i-1}(par(x))$, that is, $-rank_{i-1}(par(x)) < -rank_{i-1}(x)$

$$amCost_i(x) < 1 - 2rank_{i-1}(x) + 2rank_i(x).$$

To eliminate number 1, consider the inequality $rank_{i-1}(x) < rank_{i-1}(gpar(x))$; that is, $1 \le rank_{i-1}(gpar(x)) - rank_{i-1}(x)$. From this, we obtain

$$amCost_i(x) < rank_{i-1}(gpar(x)) - rank_{i-1}(x) - 2rank_{i-1}(x) + 2rank_i(x)$$

$$amCost_i(x) < rank_{i-1}(gpar(x)) - 3rank_{i-1}(x) + 2rank_i(x)$$

and because $rank_i(x) = rank_{i-1}(gpar(x))$

$$amCost_i(x) < -3rank_{i-1}(x) + 3rank_i(x)$$

Case 3: Two rotations are performed during a heterogeneous splay. The only difference in this proof is making the assumption that $rank_i(gpar(x)) < rank_i(x)$ and $rank_i(par(x)) < rank_i(x)$ instead of $rank_i(gpar(x)) < rank_i(par(x)) < rank_i(x)$, which renders the same result.

The total amortized cost of accessing a node x equals the sum of amortized costs of all the spaying steps executed during this access. If the number of steps equals s, then at most one (the last) step requires only one rotation (case 1) and thus

$$amCost(x) = \sum_{i=1}^{s} amCost_i(x) = \sum_{i=1}^{s-1} amCost_i(x) + amCost_s(x)$$
$$< \sum_{i=1}^{s-1} 3(rank_i(x) - rank_{i-1}(x)) + rank_s(x) - rank_{s-1}(x) + 1$$

Because $rank_s(x) > rank_{s-1}(x)$,

$$amCost(x) < \sum_{i=1}^{s-1} 3(rank_i(x) - rank_{i-1}(x)) + 3(rank_s(x) - rank_{s-1}(x)) + 1$$
$$= 3(rank_s(x) - rank_0(x)) + 1 = 3(\lg n - rank_0(x)) + 1 = O(\lg n)$$

This indicates that the amortized cost of an access to a node in a tree that is restructured with the splaying technique equals $O(\lg n)$, which is the same as the worst case in balanced trees. However, to make the comparison more adequate, we should compare a sequence of m accesses to nodes rather than one access because, with the amortize cost, one isolated access can still be on the order of $O(n)$. The efficiency of a tree that applies splaying is thus comparable to that of a balanced tree for a sequence of accesses and equals $O(m \lg n)$. ❑

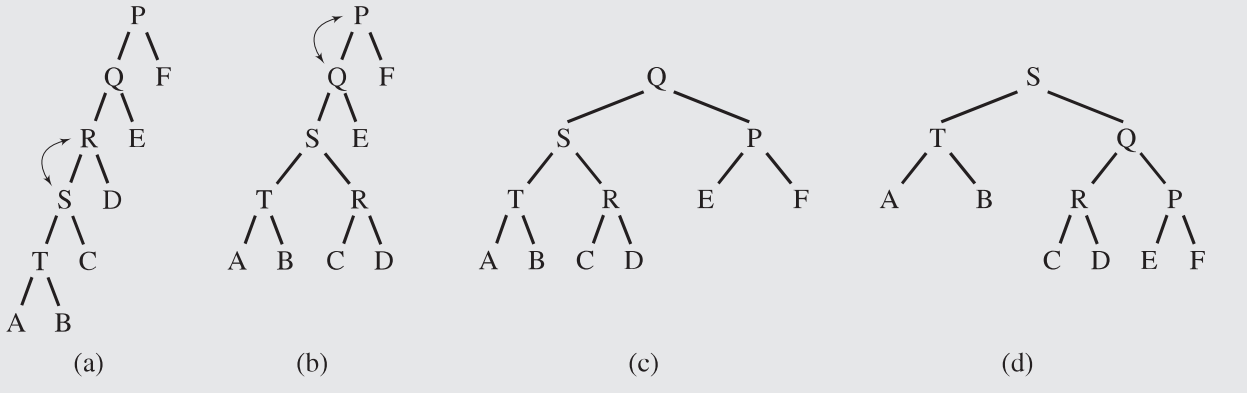
Splaying is a strategy focusing upon the elements rather than the shape of the tree. It may perform well in situations in which some elements are used much more frequently than others. If elements near the root are accessed with about the same frequency as elements on the lowest levels, then splaying may not be the best choice. In this case, a strategy that stresses balancing the tree rather than frequency is better; a modification of the splaying method is a more viable option.

Semisplaying is a modification that requires only one rotation for a homogeneous splay and continues splaying with the parent of the accessed node, not with the node itself. It is illustrated in Figure 6.48b. After R is accessed, its parent Q is rotated about P and splaying continues with Q, not with R. A rotation of R about Q is not performed, as would be the case for splaying.

Figure 6.50 illustrates the advantages of semisplaying. The elongated tree from Figure 6.49a becomes more balanced with semisplaying after accessing T (Figures 6.50a–c), and after T is accessed again, the tree in Figure 6.50d has basically the same number of levels as the tree in Figure 6.46a. (It may have one more level if E or F was a subtree higher than any of subtrees A, B, C, or D.) For implementation of this tree strategy, see the case study at the end of this chapter.

It is interesting that although the theoretical bounds obtained from self-organizing trees compare favorably with the bounds for AVL trees and random binary search

FIGURE 6.50 (a–c) Accessing *T* and restructuring the tree with semisplaying; (c–d) accessing *T* again.



trees—that is, with no balancing technique applied to it—experimental runs for trees of various sizes and different ratios of accessing keys indicate that almost always the AVL tree outperforms self-adjusting trees, and many times even a regular binary search tree performs better than a self-organizing tree (Bell and Gupta 1993). At best, this result indicates that computational complexity and amortized performance should not always be considered as the only measures of algorithm performance.

6.9 Heaps

A particular kind of binary tree, called a *heap,* has the following two properties:

1. The value of each node is greater than or equal to the values stored in each of its children.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

To be exact, these two properties define a *max heap*. If "greater" in the first property is replaced with "less," then the definition specifies a *min heap*. This means that the root of a max heap contains the largest element, whereas the root of a min heap contains the smallest. A tree has the *heap property* if each nonleaf has the first property. Due to the second condition, the number of levels in the tree is $O(\lg n)$.

The trees in Figure 6.51a are all heaps; the trees in Figure 6.51b violate the first property, and the trees in Figure 6.51c violate the second.

Interestingly, heaps can be implemented by arrays. For example, the array `data` = [2 8 6 1 10 15 3 12 11] can represent a nonheap tree in Figure 6.52. The elements are placed at sequential locations representing the nodes from top to bottom and in each level from left to right. The second property reflects the fact that the array is packed, with no gaps. Now, a heap can be defined as an array `heap` of length n in which

$$\texttt{heap[i]} \geq \texttt{heap[2·i + 1]}, \text{ for } 0 \leq i < \frac{n-1}{2}$$

FIGURE 6.51 Examples of (a) heaps and (b–c) nonheaps.

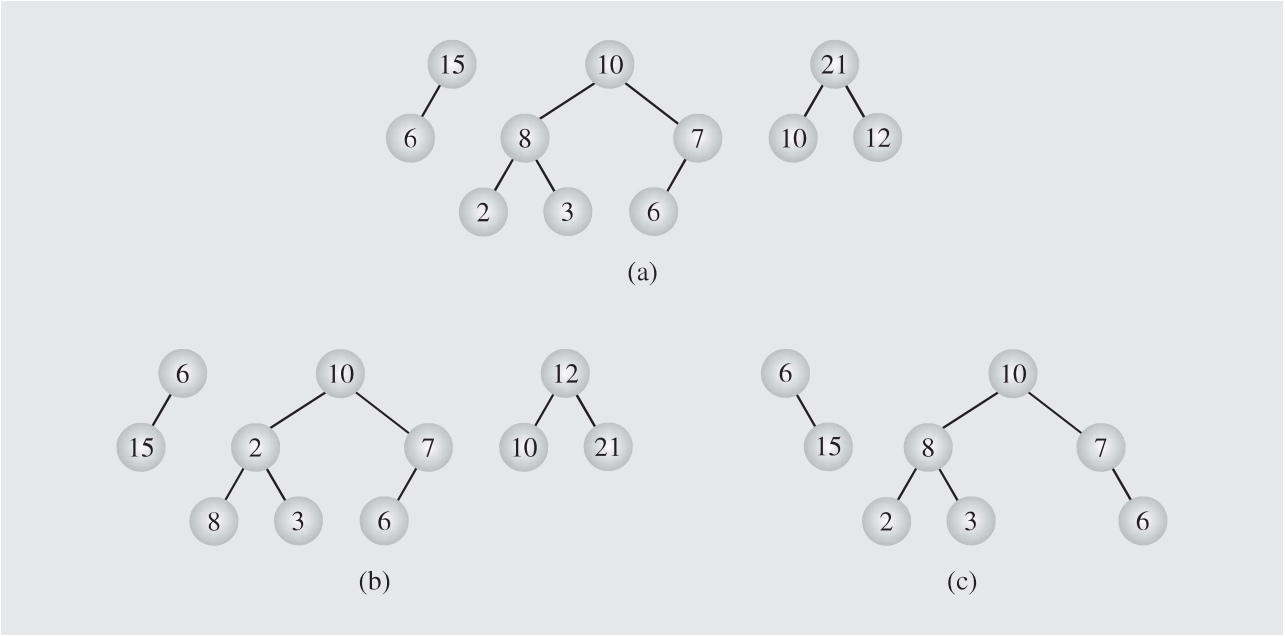
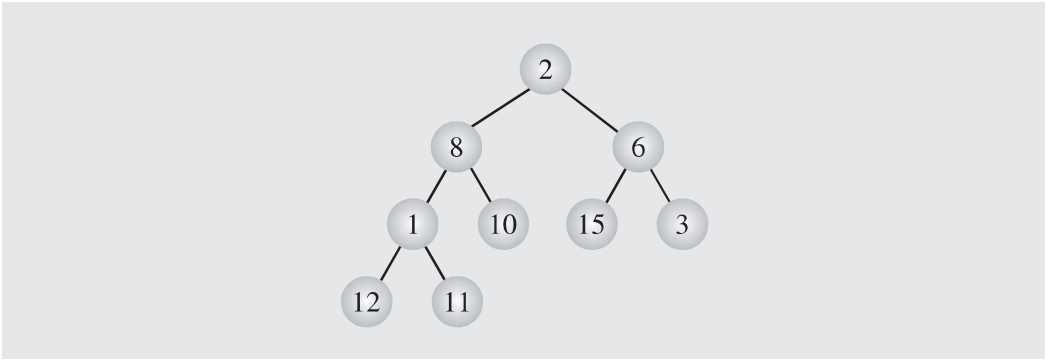


FIGURE 6.52 The array [2 8 6 1 10 15 3 12 11] seen as a tree.

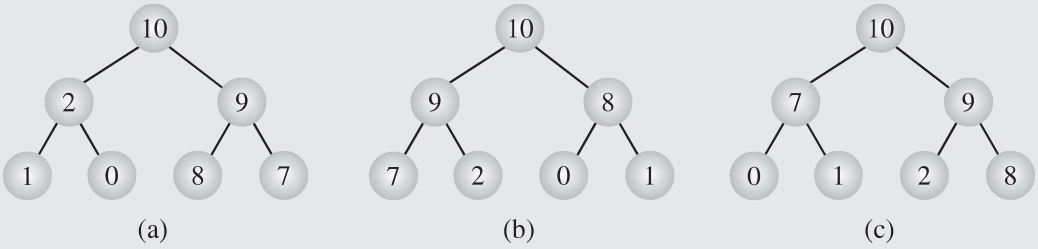


and

$$\texttt{heap[i]} \geq \texttt{heap[2·i + 2]}, \text{ for } 0 \leq i < \frac{n-2}{2}$$

Elements in a heap are not perfectly ordered. We know only that the largest element is in the root node and that, for each node, all its descendants are less than or equal to that node. But the relation between sibling nodes or, to continue the kinship terminology, between uncle and nephew nodes is not determined. The order of the elements obeys a linear line of descent, disregarding lateral lines. For this reason, all the trees in Figure 6.53 are legitimate heaps, although the heap in Figure 6.53b is ordered best.

FIGURE 6.53 Different heaps constructed with the same elements.



6.9.1 Heaps as Priority Queues

A heap is an excellent way to implement a priority queue. Section 4.3 used linked lists to implement priority queues, structures for which the complexity was expressed in terms of $O(n)$ or $O(\sqrt{n})$. For large n, this may be too ineffective. On the other hand, a heap is a perfectly balanced tree; hence, reaching a leaf requires $O(\lg n)$ searches. This efficiency is very promising. Therefore, heaps can be used to implement priority queues. To this end, however, two procedures have to be implemented to enqueue and dequeue elements on a priority queue.

To enqueue an element, the element is added at the end of the heap as the last leaf. Restoring the heap property in the case of enqueuing is achieved by moving from the last leaf toward the root.

The algorithm for enqueuing is as follows:

```
heapEnqueue(el)
  put el at the end of heap;
  while el is not in the root and el > parent(el)
    swap el with its parent;
```

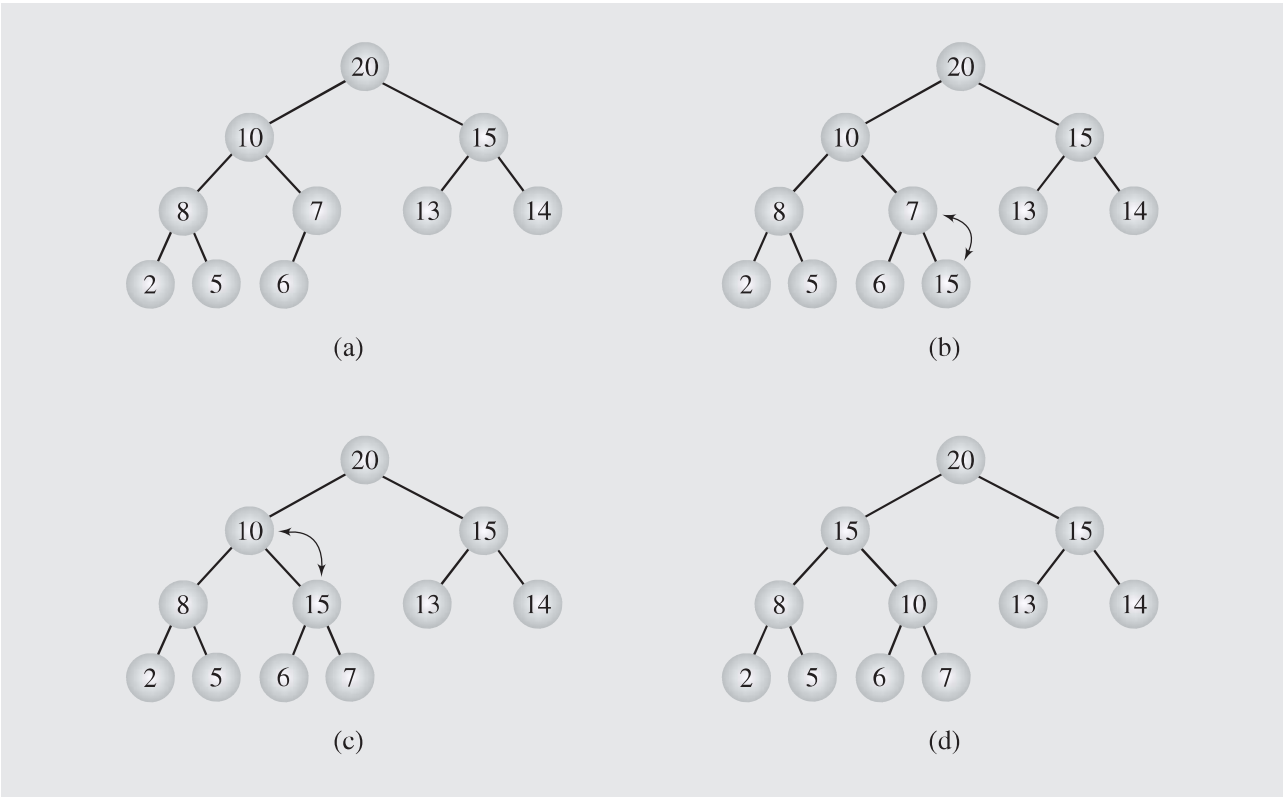
For example, the number 15 is added to the heap in Figure 6.54a as the next leaf (Figure 6.54b), which destroys the heap property of the tree. To restore this property, 15 has to be moved up the tree until it either ends up in the root or finds a parent that is not less than 15. In this example, the latter case occurs, and 15 has to be moved only twice without reaching the root.

Dequeuing an element from the heap consists of removing the root element from the heap, because by the heap property it is the element with the greatest priority. Then the last leaf is put in its place, and the heap property almost certainly has to be restored, this time by moving from the root down the tree.

The algorithm for dequeuing is as follows:

```
heapDequeue()
  extract the element from the root;
  put the element from the last leaf in its place;
  remove the last leaf;
  // both subtrees of the root are heaps;
```

FIGURE 6.54 Enqueuing an element to a heap.




```
p = the root;
while p is not a leaf and p < any of its children
   swap p with the larger child;
```

For example, 20 is dequeued from the heap in Figure 6.55a and 6 is put in its place (Figure 6.55b). To restore the heap property, 6 is swapped first with its larger child, number 15 (Figure 6.55c), and once again with the larger child, 14 (Figure 6.55d).

The last three lines of the dequeuing algorithm can be treated as a separate algorithm that restores the heap property only if it has been violated by the root of the tree. In this case, the root element is moved down the tree until it finds a proper position. This algorithm, which is the key to the heap sort, is presented in one possible implementation in Figure 6.56.

6.9.2 Organizing Arrays as Heaps

Heaps can be implemented as arrays, and in that sense, each heap is an array, but not all arrays are heaps. In some situations, however, most notably in heap sort (see Section 9.3.2), we need to convert an array into a heap (that is, reorganize the data in the array so that the resulting organization represents a heap). There are several ways to do this, but in light of the preceding section the simplest way is to start with an empty heap and sequentially include elements into a growing heap. This is a top-down

FIGURE 6.55 Dequeuing an element from a heap.

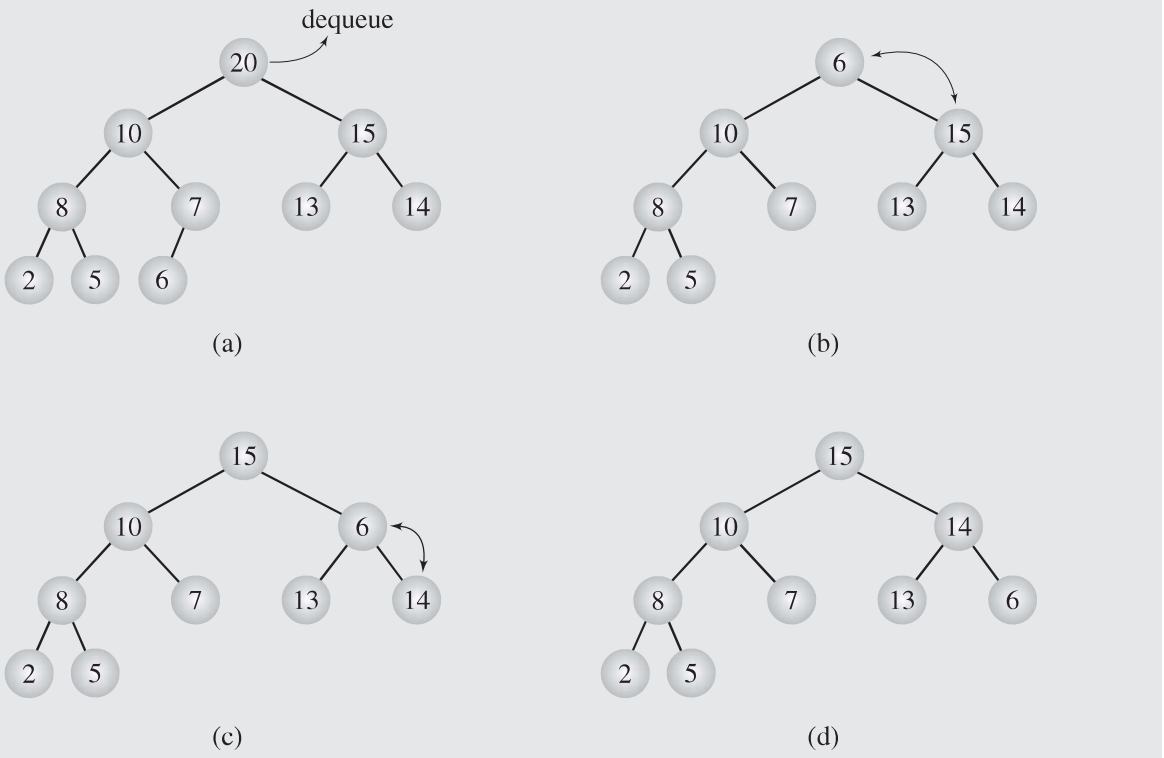


FIGURE 6.56 Implementation of an algorithm to move the root element down a tree.

```
template<class T>
void moveDown (T data[], int first, int last) {
    int largest = 2*first + 1;
    while (largest <= last) {
        if (largest < last && // first has two children (at 2*first+1 and
            data[largest] < data[largest+1]) // 2*first+2) and the second
             largest++;                      // is larger than the first;

        if (data[first] < data[largest]) {     // if necessary,
             swap(data[first],data[largest]);  // swap child and parent,
             first = largest;                  // and move down;
             largest = 2*first+1;
        }
        else largest = last+1;  // to exit the loop: the heap property
    }                           // isn't violated by data[first];
}
```

method and it was proposed by John Williams; it extends the heap by enqueuing new elements in the heap.

Figure 6.57 contains a complete example of the top-down method. First, the number 2 is enqueued in the initially empty heap (6.57a). Next, 8 is enqueued by putting it at the end of the current heap (6.57b) and then swapping with its parent (6.57c). Enqueuing the third and fourth elements, 6 (6.57d) and then 1 (6.57e), necessitates no swaps. Enqueuing the fifth element, 10, amounts to putting it at the end of the heap (6.57f), then swapping it with its parent, 2 (6.57g), and then with its new parent, 8 (6.57h) so that eventually 10 percolates up to the root of the heap. All remaining steps can be traced by the reader in Figure 6.57.

To check the complexity of the algorithm, observe that in the worst case, when a newly added element has to be moved up to the root of the tree, $\lfloor \lg k \rfloor$ exchanges are made in a heap of k nodes. Therefore, if n elements are enqueued, then in the worst case

$$\sum_{k=1}^{n} \lfloor \lg k \rfloor \leq \sum_{k=1}^{n} \lg k = \lg 1 + \cdots + \lg n = \lg(1 \cdot 2 \cdot \cdots \cdot n) = \lg(n!) = O(n \lg n)$$

exchanges are made during execution of the algorithm and the same number of comparisons. (For the last equality, $\lg(n!) = O(n \lg n)$, see Section A.2 in Appendix A.) It turns out, however, that we can do better than that.

In another algorithm, developed by Robert Floyd, a heap is built bottom-up. In this approach, small heaps are formed and repetitively merged into larger heaps in the following way:

```
FloydAlgorithm(data[])
    for i = index of the last nonleaf down to 0
        restore the heap property for the tree whose root is data[i] by calling
        moveDown(data,i,n-1);
```

Figure 6.58 contains an example of transforming the array `data[]` = [2 8 6 1 10 15 3 12 11] into a heap.

We start from the last nonleaf node, which is `data[n/2-1]`, n being the size of the array. If `data[n/2-1]` is less than one of its children, it is swapped with the larger child. In the tree in Figure 6.58a, this is the case for `data[3]` = 1 and `data[7]` = 12. After exchanging the elements, a new tree is created, shown in Figure 6.58b. Next the element `data[n/2-2]` = `data[2]` = 6 is considered. Because it is smaller than its child `data[5]` = 15, it is swapped with that child and the tree is transformed to that in Figure 6.58c. Now `data[n/2-3]` = `data[1]` = 8 is considered. Because it is smaller than one of its children, which is `data[3]` = 12, an interchange occurs, leading to the tree in Figure 6.58d. But now it can be noticed that the order established in the subtree whose root was 12 (Figure 6.58c) has been somewhat disturbed because 8 is smaller than its new child 11. This simply means that it does not suffice to compare a node's value with its children's, but a similar comparison needs to be done with grandchildren's, great-grandchildren's, and so on, until the node finds its proper position. Taking this into consideration, the next swap is made, after which the tree in Figure 6.58e is created. Only now is the element `data[n/2-4]` = `data[0]` = 2 compared with its children, which leads to two swaps (Figures 6.58f–g).

FIGURE 6.57 Organizing an array as a heap with a top-down method.

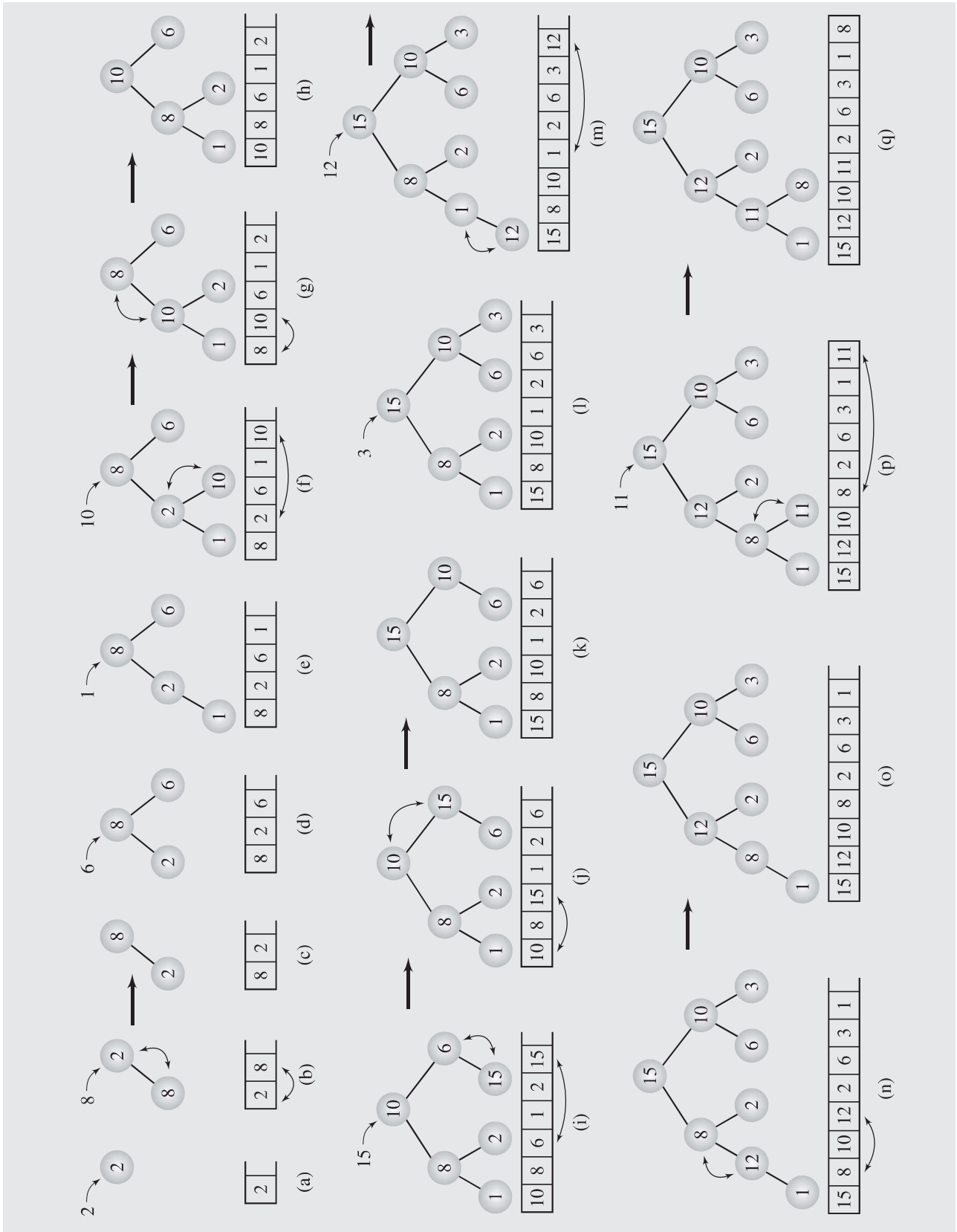
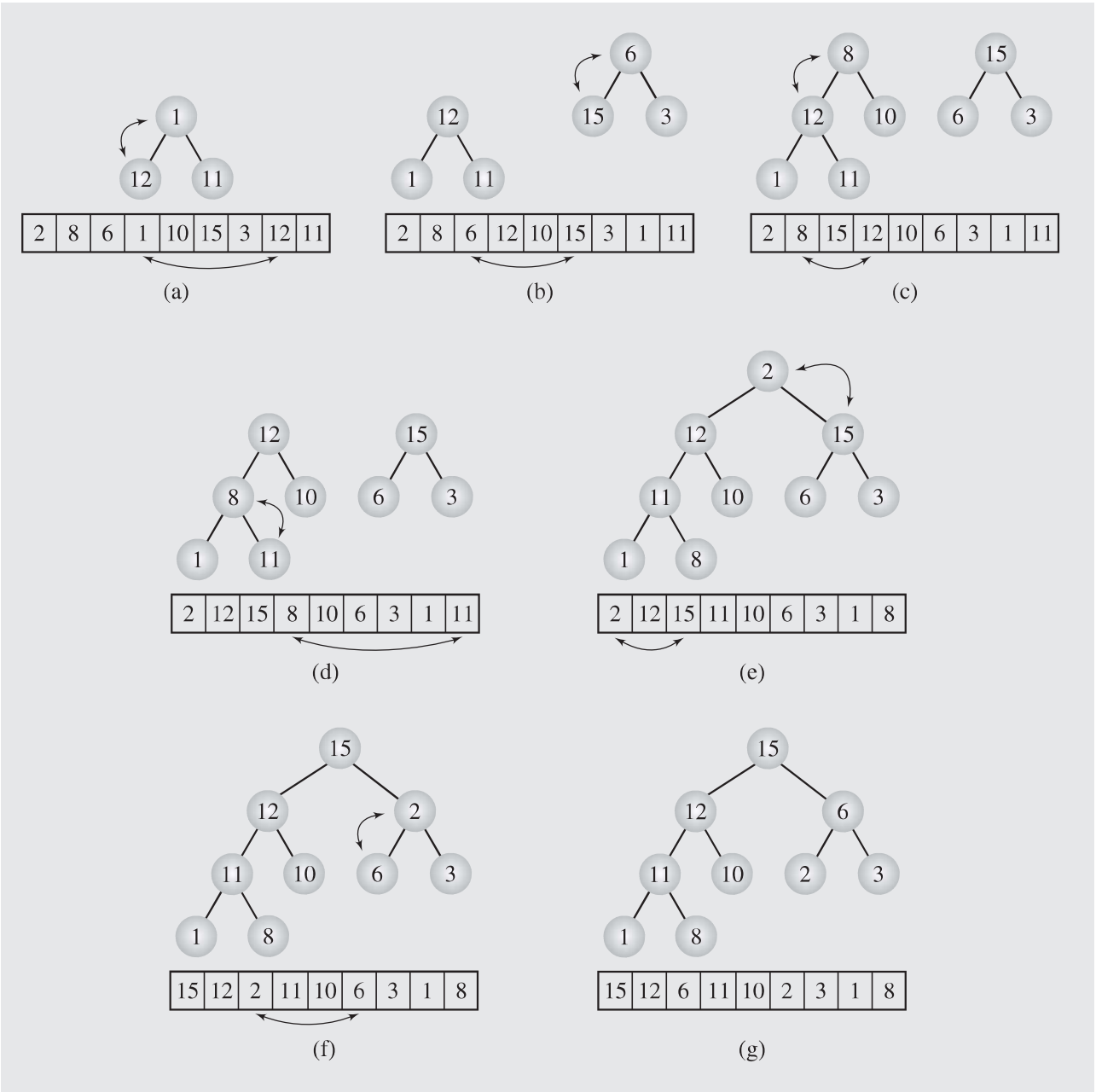


FIGURE 6.58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method.



When a new element is analyzed, both its subtrees are already heaps, as is the case with number 2, and both its subtrees with roots in nodes 12 and 15 are already heaps (Figure 6.58e). This observation is generally true: before one element is considered, its subtrees have already been converted into heaps. Thus, a heap is created from the bottom up. If the heap property is disturbed by an interchange, as in the transformation of the tree in Figure 6.58c to that in Figure 6.58d, it is immediately restored by shifting up elements that are larger than the element moved down. This is the case when

2 is exchanged with 15. The new tree is not a heap because node 2 has still larger children (Figure 6.58f). To remedy this problem, 6 is shifted up and 2 is moved down. Figure 6.58g is a heap.

We assume that a complete binary tree is created; that is, $n = 2^k - 1$ for some k. To create the heap, `moveDown()` is called $\frac{n+1}{2}$ times, once for each nonleaf. In the worst case, `moveDown()` moves data from the next to last level, consisting of $\frac{n+1}{4}$ nodes, down by one level to the level of leaves performing $\frac{n+1}{4}$ swaps. Therefore, all nodes from this level make $1 \cdot \frac{n+1}{4}$ moves. Data from the second to last level, which has $\frac{n+1}{8}$ nodes, are moved two levels down to reach the level of the leaves. Thus, nodes from this level perform $2 \cdot \frac{n+1}{8}$ moves and so on up to the root. The root of the tree as the tree becomes a heap is moved, again in the worst case, $\lg(n+1) - 1 = \lg \frac{n+1}{2}$ levels down the tree to end up in one of the leaves. Because there is only one root, this contributes $\lg \frac{n+1}{2} \cdot 1$ moves. The total number of movements can be given by this sum

$$\sum_{i=2}^{lg(n+1)} \frac{n+1}{2^i}(i-1) = (n+1)\sum_{i=2}^{lg(n+1)} \frac{i-1}{2^i}$$

which is $O(n)$ because the series $\sum_{i=2}^{\infty} \frac{i}{2^i}$ converges to 1.5 and $\sum_{i=2}^{\infty} \frac{1}{2^i}$ converges to 0.5. For an array that is not a complete binary tree, the complexity is all the more bounded by $O(n)$. The worst case for comparisons is twice this value, which is also $O(n)$, because for each node in `moveDown()`, both children of the node are compared to each other to choose the larger. That, in turn, is compared to the node. Therefore, for the worst case, Williams's method performs better than Floyd's.

The performance for the average case is much more difficult to establish. It has been found that Floyd's heap construction algorithm requires, on average, $1.88n$ comparisons (Doberkat 1984; Knuth 1998), and the number of comparisons required by Williams's algorithm in this case is between $1.75n$ and $2.76n$, and the number of swaps is $1.3n$ (Hayward and McDiarmid 1991; McDiarmid and Reed 1989). Thus, in the average case, the two algorithms perform at the same level.

6.10 Treaps

Heaps are very attractive in that they are perfectly balanced trees and they allow for immediate access to the largest element in the max-heap. However, they do not allow for quick access to any other element. Search is very efficiently performed in the binary search tree, but the shape of the tree depends on the order of insertions and deletions and the tree can become severely misshapen if no provision is made to balance it. It is, however, possible to combine a binary search tree and a heap into one data structure, a *treap*, which is also reflected in its name. Note, however, that heap is understood in this section in a weaker sense as the binary tree with the heap property, but the structural condition that requires the tree to be perfectly balanced and the leaves to be in the leftmost positions is disregarded, although heaps are intended to be as perfectly balanced as possible.

A treap is a binary search tree, that is, a tree that uses a data key as in the regular binary search tree, and an additional key, a priority, and the tree is also a