CHAPTER 4

Trees

For large amounts of input, the linear access time of linked lists is prohibitive. In this chapter, we look at a simple data structure for which the average running time of most operations is $O(\log N)$. We also sketch a conceptually simple modification to this data structure that guarantees the above time bound in the worst case and discuss a second modification that essentially gives an $O(\log N)$ running time per operation for a long sequence of instructions.

The data structure that we are referring to is known as a **binary search tree**. The binary search tree is the basis for the implementation of two library collections classes, `set` and `map`, which are used in many applications. *Trees* in general are very useful abstractions in computer science, so we will discuss their use in other, more general applications. In this chapter, we will . . .

- See how trees are used to implement the file system of several popular operating systems.
- See how trees can be used to evaluate arithmetic expressions.
- Show how to use trees to support searching operations in $O(\log N)$ average time and how to refine these ideas to obtain $O(\log N)$ worst-case bounds. We will also see how to implement these operations when the data are stored on a disk.
- Discuss and use the `set` and `map` classes.

4.1 Preliminaries

A **tree** can be defined in several ways. One natural way to define a tree is recursively. A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node, r, called the **root,** and zero or more nonempty (sub)trees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by a directed **edge** from r.

The root of each subtree is said to be a **child** of r, and r is the **parent** of each subtree root. Figure 4.1 shows a typical tree using the recursive definition.

From the recursive definition, we find that a tree is a collection of N nodes, one of which is the root, and $N - 1$ edges. That there are $N - 1$ edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent (see Fig. 4.2).

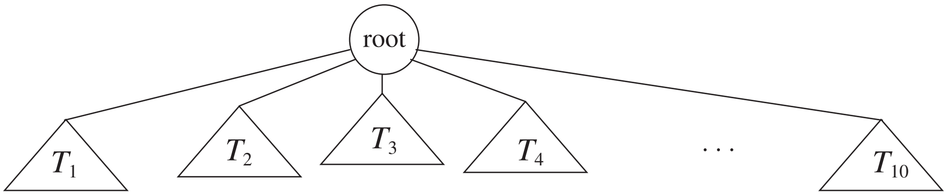


Figure 4.1 Generic tree

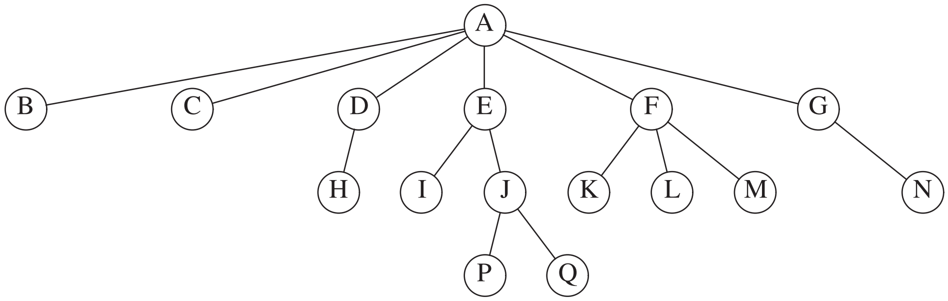


Figure 4.2 A tree

In the tree of Figure 4.2, the root is *A*. Node *F* has *A* as a parent and *K*, *L*, and *M* as children. Each node may have an arbitrary number of children, possibly zero. Nodes with no children are known as **leaves;** the leaves in the tree above are *B*, *C*, *H*, *I*, *P*, *Q*, *K*, *L*, *M*, and *N*. Nodes with the same parent are **siblings;** thus, *K*, *L*, and *M* are all siblings. **Grandparent** and **grandchild** relations can be defined in a similar manner.

A **path** from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The **length** of this path is the number of edges on the path, namely, $k - 1$. There is a path of length zero from every node to itself. Notice that in a tree there is exactly one path from the root to each node.

For any node n_i, the **depth** of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0. The **height** of n_i is the length of the longest path from n_i to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root. For the tree in Figure 4.2, *E* is at depth 1 and height 2; *F* is at depth 1 and height 1; the height of the tree is 3. The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

If there is a path from n_1 to n_2, then n_1 is an **ancestor** of n_2 and n_2 is a **descendant** of n_1. If $n_1 \neq n_2$, then n_1 is a **proper ancestor** of n_2 and n_2 is a **proper descendant** of n_1.

4.1.1 Implementation of Trees

One way to implement a tree would be to have in each node, besides its data, a link to each child of the node. However, since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the children direct links in the data

```
struct TreeNode
{
    Object    element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
};
```

Figure 4.3 Node declarations for trees

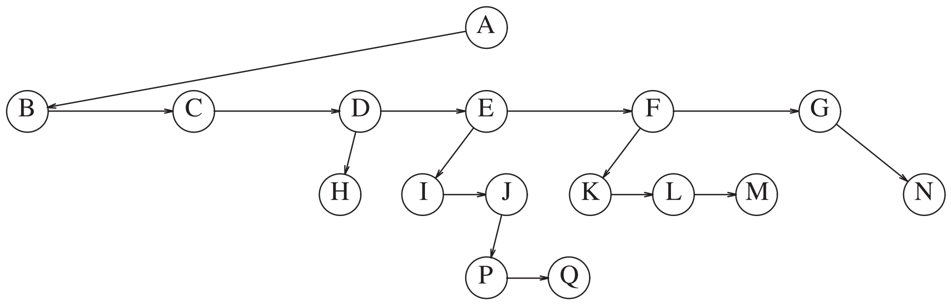


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

structure, because there would be too much wasted space. The solution is simple: Keep the children of each node in a linked list of tree nodes. The declaration in Figure 4.3 is typical.

Figure 4.4 shows how a tree might be represented in this implementation. Horizontal arrows that point downward are `firstChild` links. Arrows that go left to right are `nextSibling` links. Null links are not drawn, because there are too many.

In the tree of Figure 4.4, node *E* has both a link to a sibling (*F*) and a link to a child (*I*), while some nodes have neither.

4.1.2 Tree Traversals with an Application

There are many applications for trees. One of the popular uses is the directory structure in many common operating systems, including UNIX and DOS. Figure 4.5 is a typical directory in the UNIX file system.

The root of this directory is */usr*. (The asterisk next to the name indicates that */usr* is itself a directory.) */usr* has three children, *mark*, *alex*, and *bill*, which are themselves directories. Thus, */usr* contains three directories and no regular files. The filename */usr/mark/book/ch1.r* is obtained by following the leftmost child three times. Each / after the first indicates an edge; the result is the full **pathname**. This hierarchical file system is very popular because it allows users to organize their data logically. Furthermore, two files in different directories can share the same name, because they must have different paths from the root and thus have different pathnames. A directory in the UNIX file system is just a file with a list of all its children, so the directories are structured almost exactly in accordance

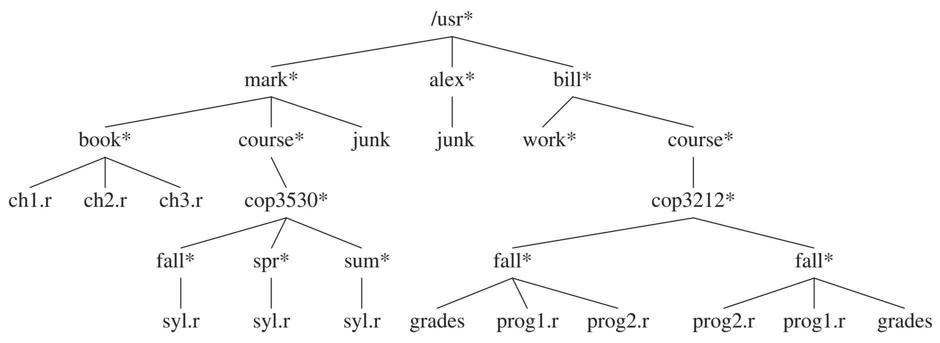


Figure 4.5 UNIX directory

```
    void FileSystem::listAll( int depth = 0 ) const
    {
1       printName( depth );  // Print the name of the object
2       if( isDirectory( ) )
3           for each file c in this directory (for each child)
4               c.listAll( depth + 1 );
    }
```

Figure 4.6 Pseudocode to list a directory in a hierarchical file system

with the type declaration above.[1] Indeed, on some versions of UNIX, if the normal command to print a file is applied to a directory, then the names of the files in the directory can be seen in the output (along with other non-ASCII information).

Suppose we would like to list the names of all of the files in the directory. Our output format will be that files that are depth d_i will have their names indented by d_i tabs. Our algorithm is given in Figure 4.6 as pseudocode.

The recursive function `listAll` needs to be started with a depth of 0 to signify no indenting for the root. This depth is an internal bookkeeping variable, and is hardly a parameter that a calling routine should be expected to know about. Thus, the default value of 0 is provided for `depth`.

The logic of the algorithm is simple to follow. The name of the file object is printed out with the appropriate number of tabs. If the entry is a directory, then we process all children recursively, one by one. These children are one level deeper, and thus need to be indented an extra space. The output is in Figure 4.7.

This traversal strategy is known as a **preorder traversal**. In a preorder traversal, work at a node is performed before (*pre*) its children are processed. When this program is run, it is clear that line 1 is executed exactly once per node, since each name is output once. Since line 1 is executed at most once per node, line 2 must also be executed once per

[1] Each directory in the UNIX file system also has one entry that points to itself and another entry that points to the parent of the directory. Thus, technically, the UNIX file system is not a tree, but is treelike.

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Figure 4.7 The (preorder) directory listing

node. Furthermore, line 4 can be executed at most once for each child of each node. But the number of children is exactly one less than the number of nodes. Finally, the `for` loop iterates once per execution of line 4 plus once each time the loop ends. Thus, the total amount of work is constant per node. If there are N file names to be output, then the running time is $O(N)$.

Another common method of traversing a tree is the **postorder traversal**. In a postorder traversal, the work at a node is performed after (*post*) its children are evaluated. As an example, Figure 4.8 represents the same directory structure as before, with the numbers in parentheses representing the number of disk blocks taken up by each file.

Since the directories are themselves files, they have sizes too. Suppose we would like to calculate the total number of blocks used by all the files in the tree. The most natural way to do this would be to find the number of blocks contained in the subdirectories */usr/mark* (30), */usr/alex* (9), and */usr/bill* (32). The total number of blocks is then the total in the

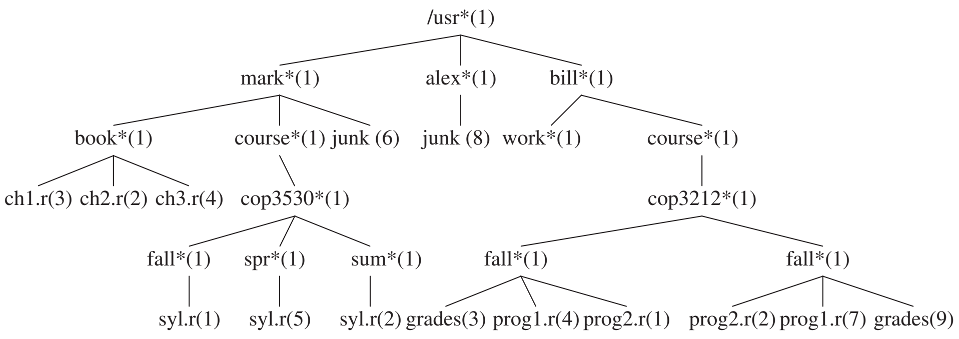


Figure 4.8 UNIX directory with file sizes obtained via postorder traversal

```
int FileSystem::size( ) const
{
    int totalSize = sizeOfThisFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

     return totalSize;
}
```

Figure 4.9 Pseudocode to calculate the size of a directory

subdirectories (71) plus the one block used by */usr*, for a total of 72. The pseudocode method `size` in Figure 4.9 implements this strategy.

If the current object is not a directory, then `size` merely returns the number of blocks it uses in the current object. Otherwise, the number of blocks used by the directory is added to the number of blocks (recursively) found in all the children. To see the difference between the postorder traversal strategy and the preorder traversal strategy, Figure 4.10 shows how the size of each directory or file is produced by the algorithm.

4.2 Binary Trees

A binary tree is a tree in which no node can have more than two children.

Figure 4.11 shows that a binary tree consists of a root and two subtrees, T_L and T_R, both of which could possibly be empty.

A property of a binary tree that is sometimes important is that the depth of an average binary tree is considerably smaller than N. An analysis shows that the average depth is $O(\sqrt{N})$, and that for a special type of binary tree, namely the *binary search tree,* the average value of the depth is $O(\log N)$. Unfortunately, the depth can be as large as $N - 1$, as the example in Figure 4.12 shows.

```
            ch1.r             3
            ch2.r             2
            ch3.r             4
        book                 10
                    syl.r     1
                fall          2
                    syl.r     5
                spr           6
                    syl.r     2
                sum           3
            cop3530          12
        course               13
        junk                  6
    mark                     30
        junk                  8
    alex                      9
        work                  1
                    grades    3
                    prog1.r   4
                    prog2.r   1
                fall          9
                    prog2.r   2
                    prog1.r   7
                    grades    9
                fall         19
            cop3212          29
        course               30
    bill                     32
/usr                         72
```

Figure 4.10 Trace of the `size` function

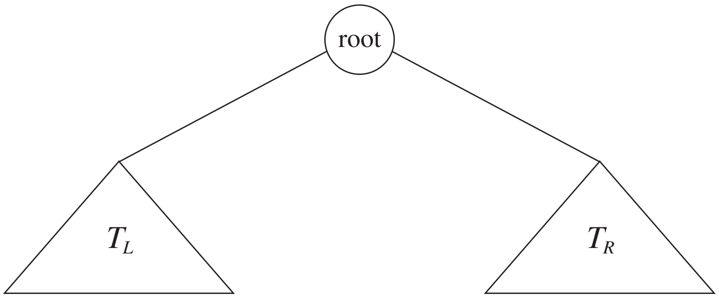


Figure 4.11 Generic binary tree

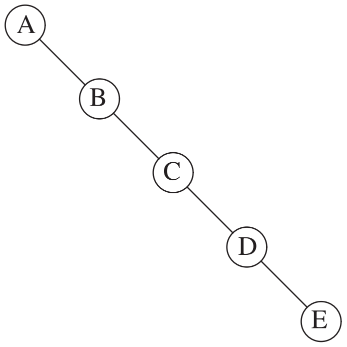


Figure 4.12 Worst-case binary tree

4.2.1 Implementation

Because a binary tree node has at most two children, we can keep direct links to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the `element` information plus two pointers (`left` and `right`) to other nodes (see Fig. 4.13).

We could draw the binary trees using the rectangular boxes that are customary for linked lists, but trees are generally drawn as circles connected by lines, because they are actually graphs. We also do not explicitly draw `nullptr` links when referring to trees, because every binary tree with N nodes would require $N + 1$ `nullptr` links.

Binary trees have many important uses not associated with searching. One of the principal uses of binary trees is in the area of compiler design, which we will now explore.

4.2.2 An Example: Expression Trees

Figure 4.14 shows an example of an **expression tree**. The leaves of an expression tree are **operands**, such as constants or variable names, and the other nodes contain **operators**. This particular tree happens to be binary, because all the operators are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the **unary minus** operator. We can evaluate an expression tree, T, by applying the operator at the root to the values

```
struct BinaryNode
{
    Object      element;      // The data in the node
    BinaryNode *left;         // Left child
    BinaryNode *right;        // Right child
};
```

Figure 4.13 Binary tree node class (pseudocode)

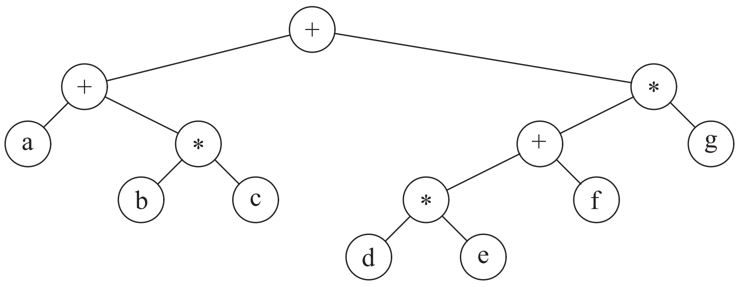


Figure 4.14 Expression tree for `(a + b * c) + ((d * e + f ) * g)`

obtained by recursively evaluating the left and right subtrees. In our example, the left subtree evaluates to `a + (b * c)` and the right subtree evaluates to `((d * e) + f) * g`. The entire tree therefore represents `(a + (b * c)) + (((d * e) + f) * g)`.

We can produce an (overly parenthesized) infix expression by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This general strategy (left, node, right) is known as an **inorder traversal**; it is easy to remember because of the type of expression it produces.

An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. If we apply this strategy to our tree above, the output is `a b c * + d e * f + g * +`, which is easily seen to be the postfix representation of Section 3.6.3. This traversal strategy is generally known as a *postorder* traversal. We have seen this traversal strategy earlier in Section 4.1.

A third traversal strategy is to print out the operator first and then recursively print out the left and right subtrees. The resulting expression, `+ + a * b c * + * d e f g`, is the less useful *prefix* notation, and the traversal strategy is a *preorder* traversal, which we have also seen earlier in Section 4.1. We will return to these traversal strategies later in the chapter.

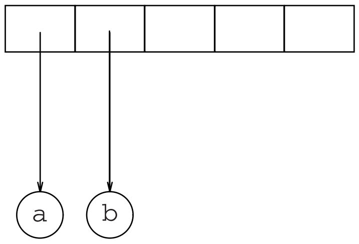
Constructing an Expression Tree

We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. The method we describe strongly resembles the postfix evaluation algorithm of Section 3.6.3. We read our expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop (pointers) to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1, respectively. A pointer to this new tree is then pushed onto the stack.

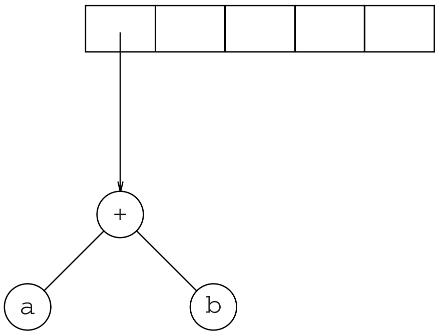
As an example, suppose the input is

```
a b + c d e + * *
```

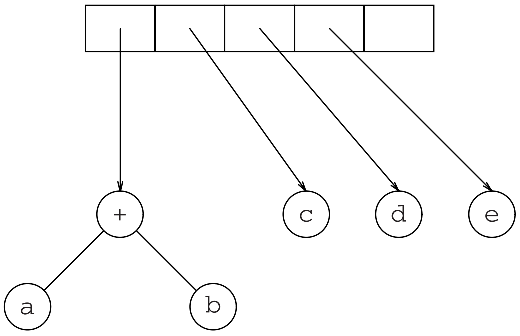
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.[2]



Next, a + is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

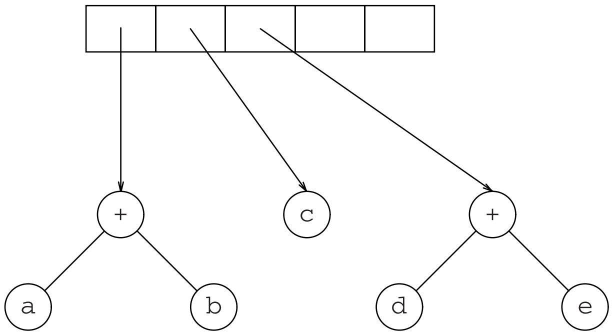


Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

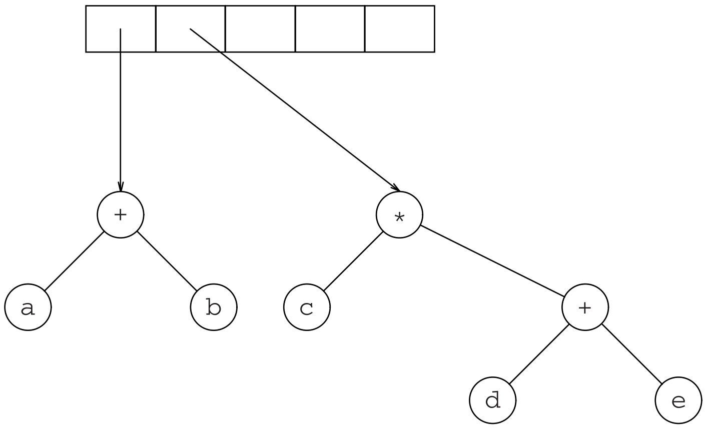


Now a + is read, so two trees are merged.

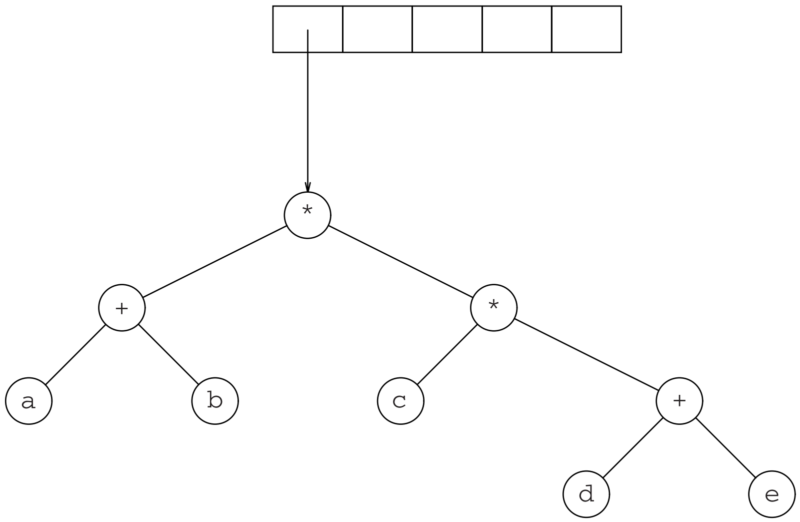
[2] For convenience, we will have the stack grow from left to right in the diagrams.



Continuing, a * is read, so we pop two tree pointers and form a new tree with a * as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



4.3 The Search Tree ADT—Binary Search Trees

An important application of binary trees is their use in searching. Let us assume that each node in the tree stores an item. In our examples, we will assume, for simplicity, that these are integers, although arbitrarily complex items are easily handled in C++. We will also assume that all the items are distinct, and we will deal with duplicates later.

The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the items in its left subtree are smaller than the item in X, and the values of all the items in its right subtree are larger than the item in X. Notice that this implies that all the elements in the tree can be ordered in some consistent manner. In Figure 4.15, the tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with item 7 in the left subtree of a node with item 6 (which happens to be the root).

We now give brief descriptions of the operations that are usually performed on binary search trees. Note that because of the recursive definition of trees, it is common to write these routines recursively. Because the average depth of a binary search tree turns out to be $O(\log N)$, we generally do not need to worry about running out of stack space.

Figure 4.16 shows the interface for the `BinarySearchTree` class template. There are several things worth noticing. Searching is based on the `<` operator that must be defined for the particular `Comparable` type. Specifically, item `x` matches `y` if both `x<y` and `y<x` are `false`. This allows `Comparable` to be a complex type (such as an employee record), with a comparison function defined on only part of the type (such as the social security number data member or salary). Section 1.6.3 illustrates the general technique of designing a class that can be used as a `Comparable`. An alternative, described in Section 4.3.1, is to allow a function object.

The data member is a pointer to the root node; this pointer is `nullptr` for empty trees. The `public` member functions use the general technique of calling `private` recursive functions. An example of how this is done for `contains`, `insert`, and `remove` is shown in Figure 4.17.

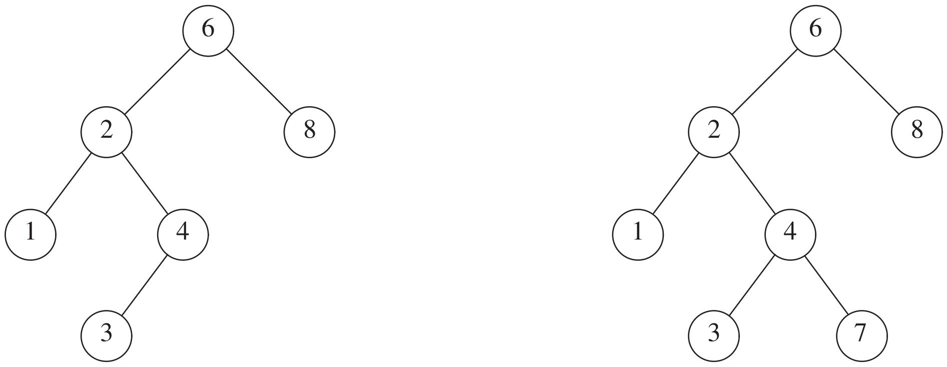


Figure 4.15 Two binary trees (only the left tree is a search tree)

```cpp
template <typename Comparable>
class BinarySearchTree
{
  public:
    BinarySearchTree( );
    BinarySearchTree( const BinarySearchTree & rhs );
    BinarySearchTree( BinarySearchTree && rhs );
    ~BinarySearchTree( );

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ostream & out = cout ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void insert( Comparable && x );
    void remove( const Comparable & x );

    BinarySearchTree & operator=( const BinarySearchTree & rhs );
    BinarySearchTree & operator=( BinarySearchTree && rhs );

  private:
    struct BinaryNode
    {
        Comparable element;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ theElement }, left{ lt }, right{ rt } { }

        BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ std::move( theElement ) }, left{ lt }, right{ rt } { }
    };

    BinaryNode *root;

    void insert( const Comparable & x, BinaryNode * & t );
    void insert( Comparable && x, BinaryNode * & t );
    void remove( const Comparable & x, BinaryNode * & t );
    BinaryNode * findMin( BinaryNode *t ) const;
    BinaryNode * findMax( BinaryNode *t ) const;
    bool contains( const Comparable & x, BinaryNode *t ) const;
    void makeEmpty( BinaryNode * & t );
    void printTree( BinaryNode *t, ostream & out ) const;
    BinaryNode * clone( BinaryNode *t ) const;
};
```

Figure 4.16 Binary search tree class skeleton

```
/**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
```

Figure 4.17 Illustration of public member function calling private recursive member function

Several of the `private` member functions use the technique of passing a pointer variable using call-by-reference. This allows the `public` member functions to pass a pointer to the root to the `private` recursive member functions. The recursive functions can then change the value of the root so that the `root` points to another node. We will describe the technique in more detail when we examine the code for `insert`.

We can now describe some of the `private` methods.

4.3.1 `contains`

This operation requires returning `true` if there is a node in tree T that has item X, or `false` if there is no such node. The structure of the tree makes this simple. If T is empty, then we can just return `false`. Otherwise, if the item stored at T is X, we can return `true`. Otherwise, we make a recursive call on a subtree of T, either left or right, depending on the relationship of X to the item stored in T. The code in Figure 4.18 is an implementation of this strategy.

```
/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}
```

Figure 4.18 `contains` operation for binary search trees

Notice the order of the tests. It is crucial that the test for an empty tree be performed first, since otherwise, we would generate a run time error attempting to access a data member through a `nullptr` pointer. The remaining tests are arranged with the least likely case last. Also note that both recursive calls are actually tail recursions and can be easily removed with a `while` loop. The use of tail recursion is justifiable here because the simplicity of algorithmic expression compensates for the decrease in speed, and the amount of stack space used is expected to be only $O(\log N)$.

Figure 4.19 shows the trivial changes required to use a function object rather than requiring that the items be `Comparable`. This mimics the idioms in Section 1.6.4.

4.3.2 `findMin` and `findMax`

These `private` routines return a pointer to the node containing the smallest and largest elements in the tree, respectively. To perform a `findMin`, start at the root and go left as long as there is a left child. The stopping point is the smallest element. The `findMax` routine is the same, except that branching is to the right child.

Many programmers do not bother using recursion. We will code the routines both ways by doing `findMin` recursively and `findMax` nonrecursively (see Figs. 4.20 and 4.21).

Notice how we carefully handle the degenerate case of an empty tree. Although this is always important to do, it is especially crucial in recursive programs. Also notice that it is safe to change `t` in `findMax`, since we are only working with a copy of a pointer. Always be extremely careful, however, because a statement such as `t->right = t->right->right` will make changes.

```
template <typename Object, typename Comparator=less<Object>>
class BinarySearchTree
{
  public:

    // Same methods, with Object replacing Comparable

  private:

    BinaryNode *root;
    Comparator isLessThan;

    // Same methods, with Object replacing Comparable

    /**
     * Internal method to test if an item is in a subtree.
     * x is item to search for.
     * t is the node that roots the subtree.
     */
    bool contains( const Object & x, BinaryNode *t ) const
    {
        if( t == nullptr )
            return false;
        else if( isLessThan( x, t->element ) )
            return contains( x, t->left );
        else if( isLessThan( t->element, x ) )
            return contains( x, t->right );
        else
            return true;    // Match
    }
};
```

Figure 4.19 Illustrates use of a function object to implement binary search tree

4.3.3 `insert`

The insertion routine is conceptually simple. To insert *X* into tree *T*, proceed down the tree as you would with a `contains`. If *X* is found, do nothing. Otherwise, insert *X* at the last spot on the path traversed. Figure 4.22 shows what happens. To insert 5, we traverse the tree as though a `contains` were occurring. At the node with item 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot to place 5.

Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence. This adds some extra space to the entire tree but is better than putting duplicates in the tree (which tends to make the tree very deep). Of course,

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}
```

Figure 4.20 Recursive implementation of `findMin` for binary search trees

```
/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}
```

Figure 4.21 Nonrecursive implementation of `findMax` for binary search trees

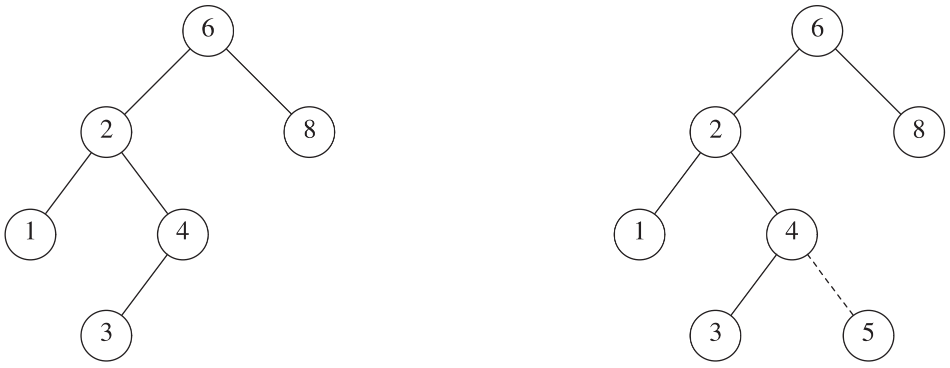


Figure 4.22 Binary search trees before and after inserting 5

this strategy does not work if the key that guides the < operator is only part of a larger structure. If that is the case, then we can keep all of the structures that have the same key in an auxiliary data structure, such as a list or another search tree.

Figure 4.23 shows the code for the insertion routine. Lines 12 and 14 recursively insert and attach x into the appropriate subtree. Notice that in the recursive routine, the only time that t changes is when a new leaf is created. When this happens, it means that the recursive routine has been called from some other node, p, which is to be the leaf's parent. The call

```
 1  /**
 2   * Internal method to insert into a subtree.
 3   * x is the item to insert.
 4   * t is the node that roots the subtree.
 5   * Set the new root of the subtree.
 6   */
 7  void insert( const Comparable & x, BinaryNode * & t )
 8  {
 9      if( t == nullptr )
10          t = new BinaryNode{ x, nullptr, nullptr };
11      else if( x < t->element )
12          insert( x, t->left );
13      else if( t->element < x )
14          insert( x, t->right );
15      else
16          ;  // Duplicate; do nothing
17  }
18
19  /**
20   * Internal method to insert into a subtree.
21   * x is the item to insert by moving.
22   * t is the node that roots the subtree.
23   * Set the new root of the subtree.
24   */
25  void insert( Comparable && x, BinaryNode * & t )
26  {
27      if( t == nullptr )
28          t = new BinaryNode{ std::move( x ), nullptr, nullptr };
29      else if( x < t->element )
30          insert( std::move( x ), t->left );
31      else if( t->element < x )
32          insert( std::move( x ), t->right );
33      else
34          ;  // Duplicate; do nothing
35  }
```

Figure 4.23 Insertion into a binary search tree

will be `insert(x,p->left)` or `insert(x,p->right)`. Either way, `t` is now a reference to either `p->left` or `p->right`, meaning that `p->left` or `p->right` will be changed to point at the new node. All in all, a slick maneuver.

4.3.4 `remove`

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a link to bypass the node (we will draw the link directions explicitly for clarity). See Figure 4.24.

The complicated case deals with a node with two children. The general strategy is to replace the data of this node with the smallest data of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second `remove` is an easy one. Figure 4.25 shows an initial tree and the result of a deletion. The node to be deleted is the left child of the root; the key value is 2. It is replaced with the smallest data in its right subtree (3), and then that node is deleted as before.

The code in Figure 4.26 performs deletion. It is inefficient because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It is easy to remove this inefficiency by writing a special `removeMin` method, and we have left it in only for simplicity.

If the number of deletions is expected to be small, then a popular strategy to use is **lazy deletion:** When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate items are present, because then the data member that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant (why?), so there is a very small time penalty associated with lazy deletion. Also, if a deleted item is reinserted, the overhead of allocating a new cell is avoided.

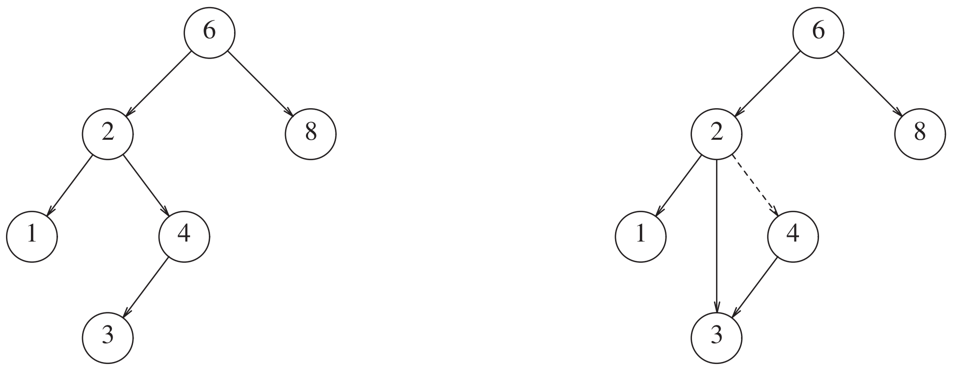


Figure 4.24 Deletion of a node (4) with one child, before and after

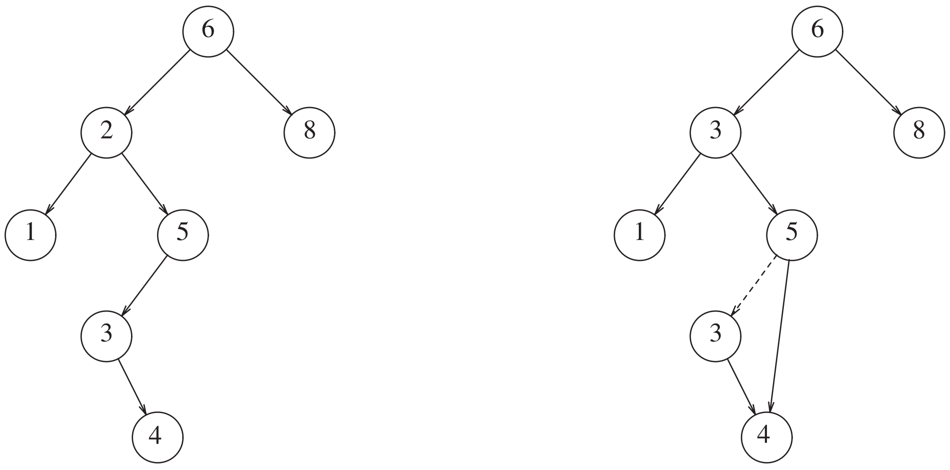


Figure 4.25 Deletion of a node (2) with two children, before and after

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Figure 4.26 Deletion routine for binary search trees

4.3.5 Destructor and Copy Constructor

As usual, the destructor calls `makeEmpty`. The public `makeEmpty` (not shown) simply calls the private recursive version. As shown in Figure 4.27, after recursively processing `t`'s children, a call to `delete` is made for `t`. Thus all nodes are recursively reclaimed. Notice that at the end, `t`, and thus `root`, is changed to point at `nullptr`. The copy constructor, shown in Figure 4.28, follows the usual procedure, first initializing `root` to `nullptr` and then making a copy of `rhs`. We use a very slick recursive function named `clone` to do all the dirty work.

4.3.6 Average-Case Analysis

Intuitively, we expect that all of the operations described in this section, except `makeEmpty` and copying, should take $O(\log N)$ time, because in constant time we descend a level in the tree, thus operating on a tree that is now roughly half as large. Indeed, the running time of all the operations (except `makeEmpty` and copying) is $O(d)$, where d is the depth of the node containing the accessed item (in the case of `remove`, this may be the replacement node in the two-child case).

We prove in this section that the average depth over all nodes in a tree is $O(\log N)$ on the assumption that all insertion sequences are equally likely.

The sum of the depths of all nodes in a tree is known as the **internal path length.** We will now calculate the average internal path length of a binary search tree, where the average is taken over all possible insertion sequences into binary search trees.

```
/**
 * Destructor for the tree
 */
~BinarySearchTree( )
{
    makeEmpty( );
}
/**
 * Internal method to make subtree empty.
 */
void makeEmpty( BinaryNode * & t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = nullptr;
}
```

Figure 4.27 Destructor and recursive `makeEmpty` member function

```
/**
 * Copy constructor
 */
BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
{
    root = clone( rhs.root );
}

/**
 * Internal method to clone subtree.
 */
BinaryNode * clone( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    else
        return new BinaryNode{ t->element, clone( t->left ), clone( t->right ) };
}
```

Figure 4.28 Copy constructor and recursive `clone` member function

Let $D(N)$ be the internal path length for some tree T of N nodes. $D(1) = 0$. An N-node tree consists of an i-node left subtree and an $(N - i - 1)$-node right subtree, plus a root at depth zero for $0 \leq i < N$. $D(i)$ is the internal path length of the left subtree with respect to its root. In the main tree, all these nodes are one level deeper. The same holds for the right subtree. Thus, we get the recurrence

$$D(N) = D(i) + D(N - i - 1) + N - 1$$

If all subtree sizes are equally likely, which is true for binary search trees (since the subtree size depends only on the relative rank of the first element inserted into the tree), but not binary trees, then the average value of both $D(i)$ and $D(N - i - 1)$ is $(1/N)\sum_{j=0}^{N-1} D(j)$. This yields

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

This recurrence will be encountered and solved in Chapter 7, obtaining an average value of $D(N) = O(N \log N)$. Thus, the expected depth of any node is $O(\log N)$. As an example, the randomly generated 500-node tree shown in Figure 4.29 has nodes at expected depth 9.98.

It is tempting to say immediately that this result implies that the average running time of all the operations discussed in the previous section is $O(\log N)$, but this is not entirely true. The reason for this is that because of deletions, it is not clear that all binary search trees are equally likely. In particular, the deletion algorithm described above favors making the left subtrees deeper than the right, because we are always replacing a deleted node with a node from the right subtree. The exact effect of this strategy is still unknown, but

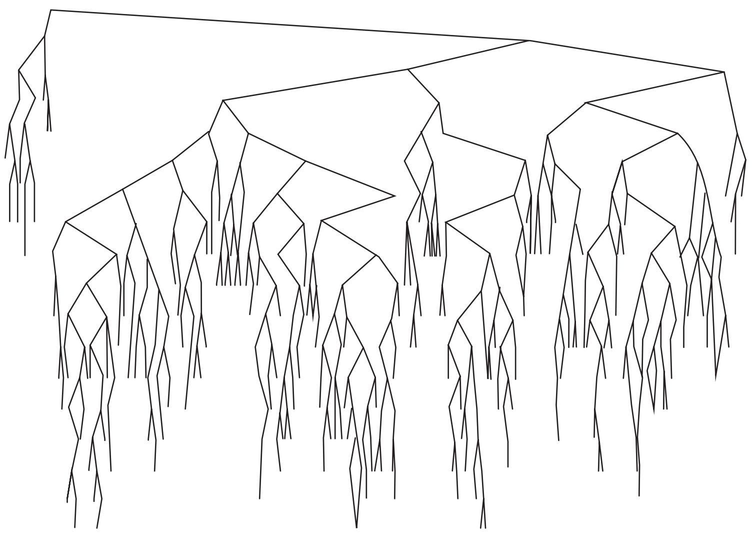


Figure 4.29 A randomly generated binary search tree

it seems only to be a theoretical novelty. It has been shown that if we alternate insertions and deletions $\Theta(N^2)$ times, then the trees will have an expected depth of $\Theta(\sqrt{N})$. After a quarter-million random `insert/remove` pairs, the tree that was somewhat right-heavy in Figure 4.29 looks decidedly unbalanced (average depth = 12.51) in Figure 4.30.

We could try to eliminate the problem by randomly choosing between the smallest element in the right subtree and the largest in the left when replacing the deleted element. This apparently eliminates the bias and should keep the trees balanced, but nobody has

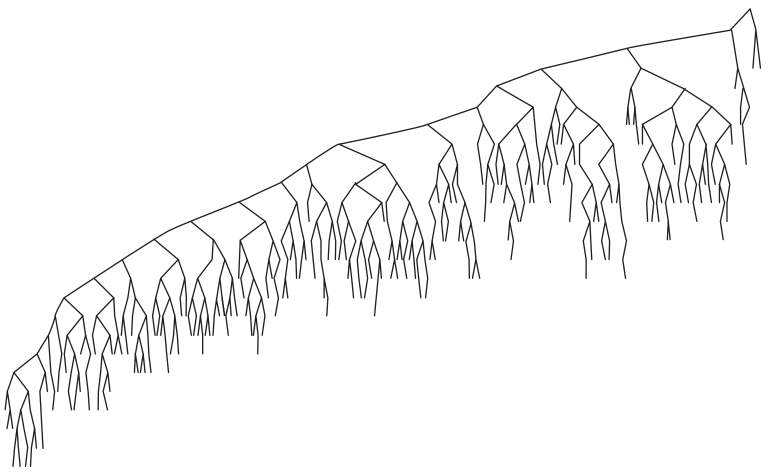


Figure 4.30 Binary search tree after $\Theta(N^2)$ `insert/remove` pairs

actually proved this. In any event, this phenomenon appears to be mostly a theoretical novelty, because the effect does not show up at all for small trees, and, stranger still, if $o(N^2)$ `insert/remove` pairs are used, then the tree seems to gain balance!

The main point of this discussion is that deciding what "average" means is generally extremely difficult and can require assumptions that may or may not be valid. In the absence of deletions, or when lazy deletion is used, we can conclude that the average running times of the operations above are $O(\log N)$. Except for strange cases like the one discussed above, this result is very consistent with observed behavior.

If the input comes into a tree presorted, then a series of `inserts` will take quadratic time and give a very expensive implementation of a linked list, since the tree will consist only of nodes with no left children. One solution to the problem is to insist on an extra structural condition called *balance:* No node is allowed to get too deep.

There are quite a few general algorithms to implement balanced trees. Most are quite a bit more complicated than a standard binary search tree, and all take longer on average for updates. They do, however, provide protection against the embarrassingly simple cases. Below, we will sketch one of the oldest forms of balanced search trees, the AVL tree.

A second method is to forgo the balance condition and allow the tree to be arbitrarily deep, but after every operation, a restructuring rule is applied that tends to make future operations efficient. These types of data structures are generally classified as **self-adjusting.** In the case of a binary search tree, we can no longer guarantee an $O(\log N)$ bound on any single operation but can show that any *sequence* of M operations takes total time $O(M \log N)$ in the worst case. This is generally sufficient protection against a bad worst case. The data structure we will discuss is known as a *splay tree;* its analysis is fairly intricate and is discussed in Chapter 11.

4.4 AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a **balance condition**. The balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log N)$. The simplest idea is to require that the left and right subtrees have the same height. As Figure 4.31 shows, this idea does not force the tree to be shallow.

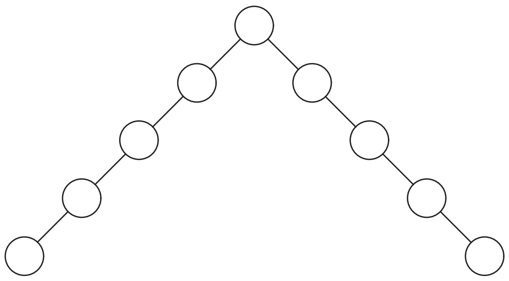


Figure 4.31 A bad binary tree. Requiring balance at the root is not enough.

Another balance condition would insist that every node must have left and right subtrees of the same height. If the height of an empty subtree is defined to be -1 (as is usual), then only perfectly balanced trees of $2^k - 1$ nodes would satisfy this criterion. Thus, although this guarantees trees of small depth, the balance condition is too rigid to be useful and needs to be relaxed.

An **AVL tree** is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.) In Figure 4.32 the tree on the left is an AVL tree but the tree on the right is not. Height information is kept for each node (in the node structure). It can be shown that the height of an AVL tree is at most roughly $1.44 \log(N + 2) - 1.328$, but in practice it is only slightly more than $\log N$. As an example, the AVL tree of height 9 with the fewest nodes (143) is shown in Figure 4.33. This tree has as a left subtree an AVL tree of height 7 of minimum size. The right subtree is an AVL tree of height 8 of minimum size. This tells us that the minimum number of nodes, $S(h)$, in an AVL tree of height h is given by $S(h) = S(h - 1) + S(h - 2) + 1$. For $h = 0$, $S(h) = 1$. For $h = 1$, $S(h) = 2$. The function $S(h)$ is closely related to the Fibonacci numbers, from which the bound claimed above on the height of an AVL tree follows.

Thus, all the tree operations can be performed in $O(\log N)$ time, except possibly insertion and deletion. When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6 into the AVL tree in Figure 4.32 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a **rotation.**

After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition. We will show how to rebalance the tree at the first (i.e., deepest) such node, and we will prove that this rebalancing guarantees that the entire tree satisfies the AVL property.

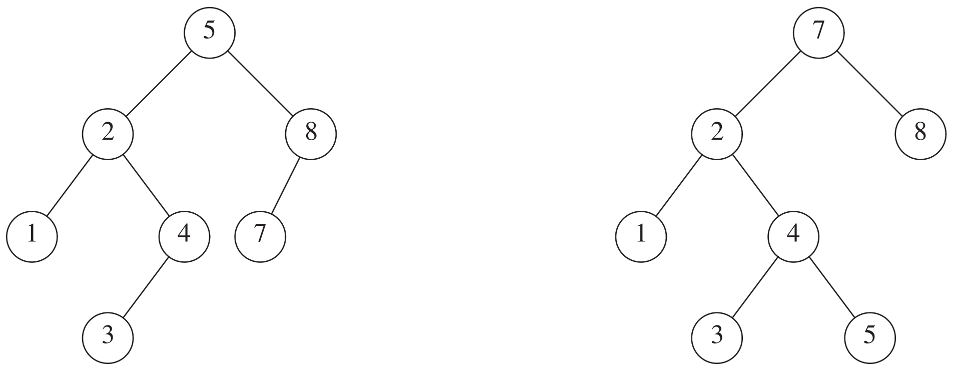


Figure 4.32 Two binary search trees. Only the left tree is AVL.

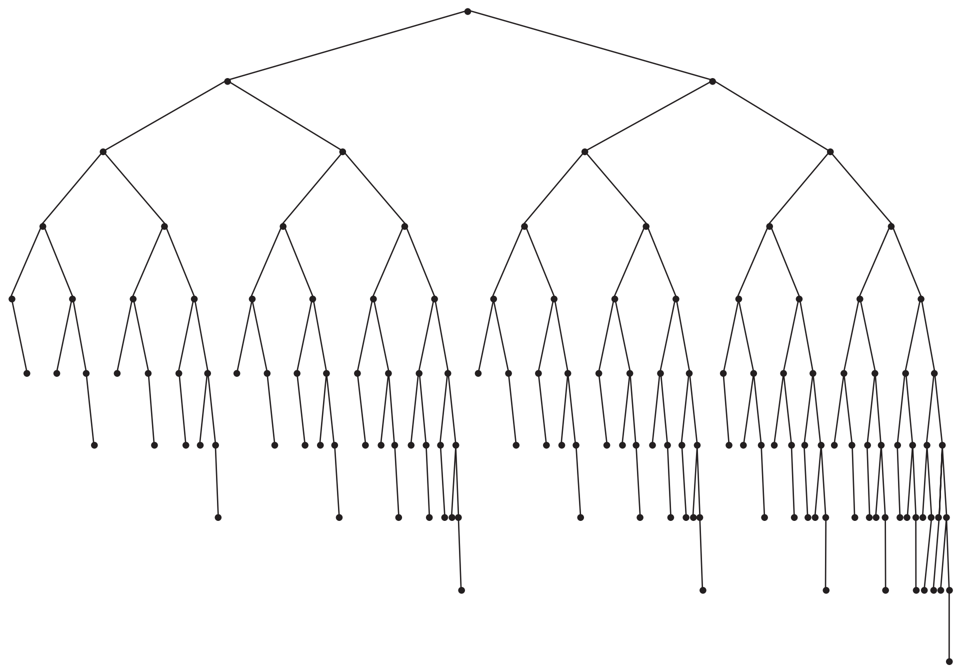


Figure 4.33 Smallest AVL tree of height 9

Let us call the node that must be rebalanced α. Since any node has at most two children, and a height imbalance requires that α's two subtrees' heights differ by two, it is easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α
2. An insertion into the right subtree of the left child of α
3. An insertion into the left subtree of the right child of α
4. An insertion into the right subtree of the right child of α

Cases 1 and 4 are mirror image symmetries with respect to α, as are cases 2 and 3. Consequently, as a matter of theory, there are two basic cases. From a programming perspective, of course, there are still four cases.

The first case, in which the insertion occurs on the "outside" (i.e., left–left or right–right), is fixed by a **single rotation** of the tree. The second case, in which the insertion occurs on the "inside" (i.e., left–right or right–left) is handled by the slightly more complex **double rotation.** These are fundamental operations on the tree that we'll see used several times in balanced-tree algorithms. The remainder of this section describes these rotations, proves that they suffice to maintain balance, and gives a casual implementation of the AVL tree. Chapter 12 describes other balanced-tree methods with an eye toward a more careful implementation.

4.4.1 Single Rotation

Figure 4.34 shows the *single rotation* that fixes case 1. The before picture is on the left and the after is on the right. Let us analyze carefully what is going on. Node k_2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree (the dashed lines in the middle of the diagram mark the levels). The situation depicted is the only possible case 1 scenario that allows k_2 to satisfy the AVL property before an insertion but violate it afterwards. Subtree X has grown to an extra level, causing it to be exactly two levels deeper than Z. Y cannot be at the same level as the new X because then k_2 would have been out of balance *before* the insertion, and Y cannot be at the same level as Z because then k_1 would be the first node on the path toward the root that was in violation of the AVL balancing condition.

To ideally rebalance the tree, we would like to move X up a level and Z down a level. Note that this is actually more than the AVL property would require. To do this, we rearrange nodes into an equivalent tree as shown in the second part of Figure 4.34. Here is an abstract scenario: Visualize the tree as being flexible, grab the child node k_1, close your eyes, and shake it, letting gravity take hold. The result is that k_1 will be the new root. The binary search tree property tells us that in the original tree $k_2 > k_1$, so k_2 becomes the right child of k_1 in the new tree. X and Z remain as the left child of k_1 and right child of k_2, respectively. Subtree Y, which holds items that are between k_1 and k_2 in the original tree, can be placed as k_2's left child in the new tree and satisfy all the ordering requirements.

As a result of this work, which requires only a few pointer changes, we have another binary search tree that is an AVL tree. This happens because X moves up one level, Y stays at the same level, and Z moves down one level. k_2 and k_1 not only satisfy the AVL requirements, but they also have subtrees that are exactly the same height. Furthermore, the new height of the entire subtree is *exactly the same* as the height of the original subtree prior to the insertion that caused X to grow. Thus no further updating of heights on the path to the root is needed, and consequently *no further rotations are needed.* Figure 4.35 shows that after the insertion of 6 into the original AVL tree on the left, node 8 becomes unbalanced. Thus, we do a single rotation between 7 and 8, obtaining the tree on the right.

As we mentioned earlier, case 4 represents a symmetric case. Figure 4.36 shows how a single rotation is applied. Let us work through a rather long example. Suppose we start with an initially empty AVL tree and insert the items 3, 2, 1, and then 4 through 7 in sequential order. The first problem occurs when it is time to insert item 1 because the AVL

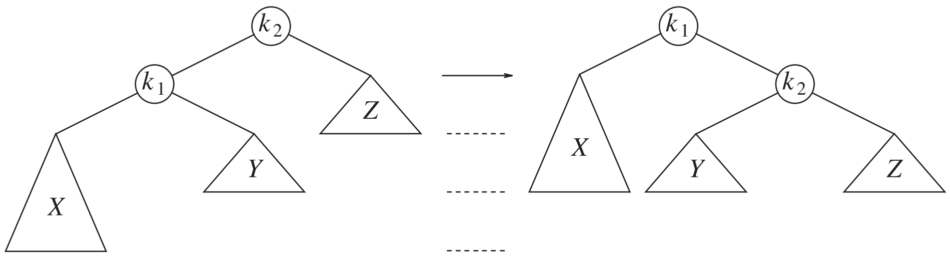


Figure 4.34 Single rotation to fix case 1

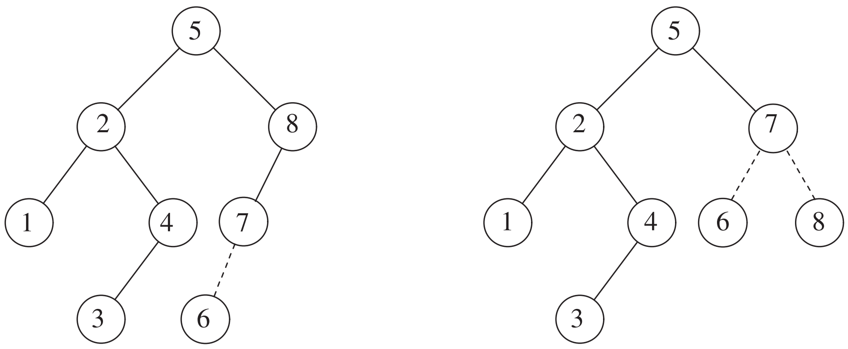


Figure 4.35 AVL property destroyed by insertion of 6, then fixed by a single rotation

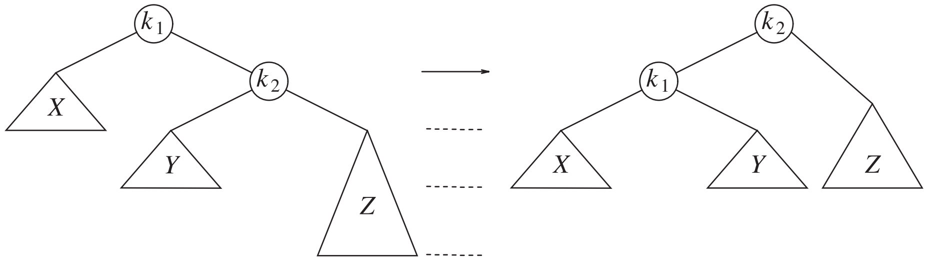
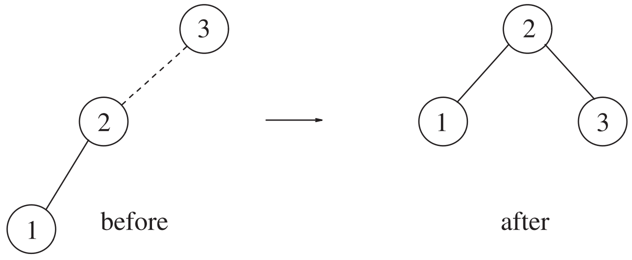
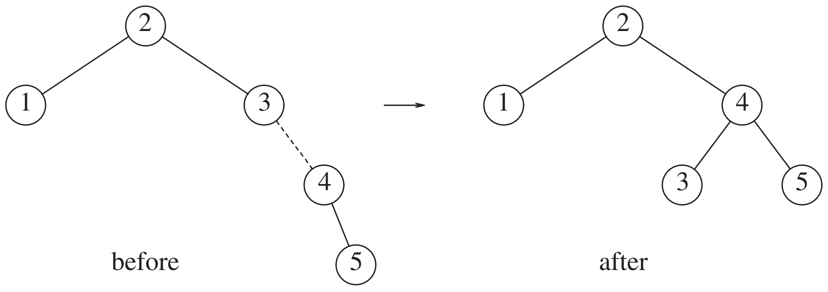


Figure 4.36 Single rotation fixes case 4

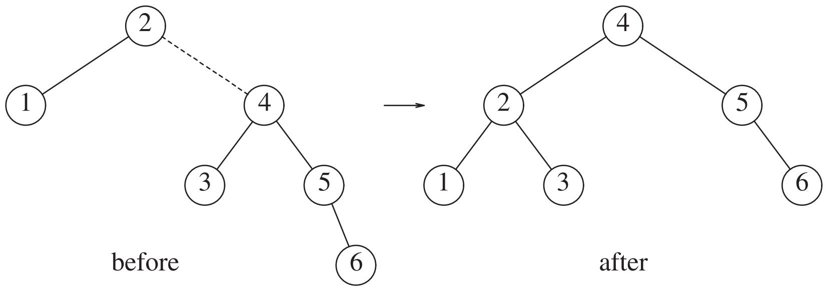
property is violated at the root. We perform a single rotation between the root and its left child to fix the problem. Here are the before and after trees:



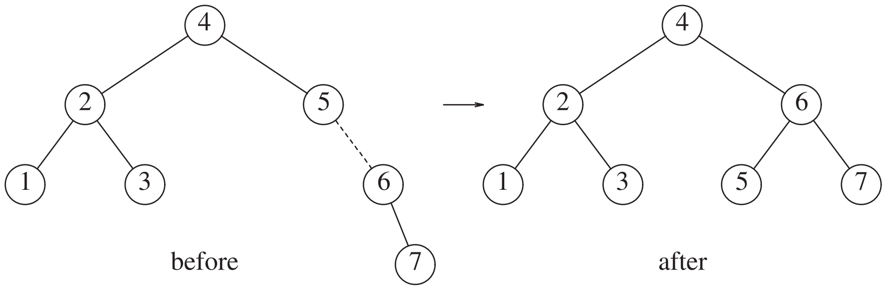
A dashed line joins the two nodes that are the subject of the rotation. Next we insert 4, which causes no problems, but the insertion of 5 creates a violation at node 3 that is fixed by a single rotation. Besides the local change caused by the rotation, the programmer must remember that the rest of the tree has to be informed of this change. Here this means that 2's right child must be reset to link to 4 instead of 3. Forgetting to do so is easy and would destroy the tree (4 would be inaccessible).



Next we insert 6. This causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be height 2. Therefore, we perform a single rotation at the root between 2 and 4.



The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2. Every item in this subtree must lie between 2 and 4, so this transformation makes sense. The next item we insert is 7, which causes another rotation:



4.4.2 Double Rotation

The algorithm described above has one problem: As Figure 4.37 shows, it does not work for cases 2 or 3. The problem is that subtree Y is too deep, and a single rotation does not make it any less deep. The *double rotation* that solves the problem is shown in Figure 4.38.

The fact that subtree Y in Figure 4.37 has had an item inserted into it guarantees that it is nonempty. Thus, we may assume that it has a root and two subtrees. Consequently, the

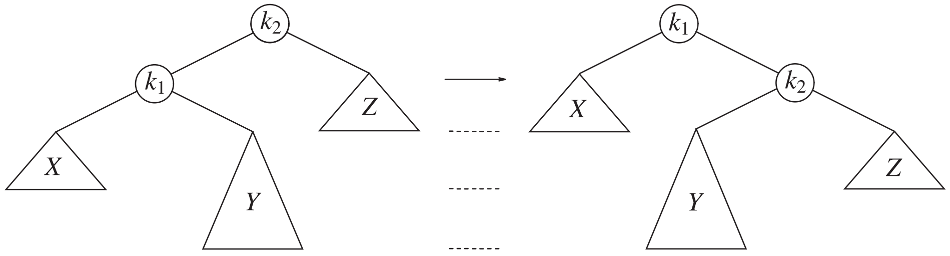


Figure 4.37 Single rotation fails to fix case 2

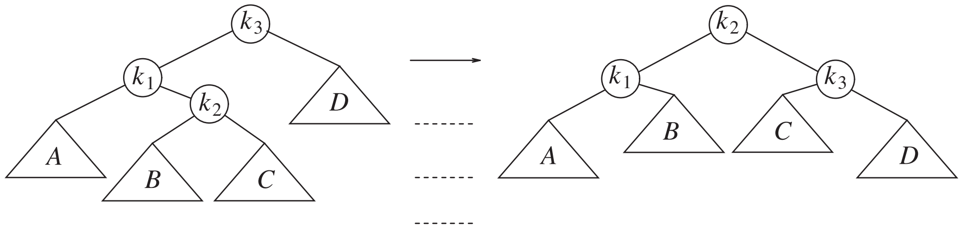


Figure 4.38 Left–right double rotation to fix case 2

tree may be viewed as four subtrees connected by three nodes. As the diagram suggests, exactly one of tree *B* or *C* is two levels deeper than *D* (unless all are empty), but we cannot be sure which one. It turns out not to matter; in Figure 4.38, both *B* and *C* are drawn at $1\frac{1}{2}$ levels below *D*.

To rebalance, we see that we cannot leave k_3 as the root, and a rotation between k_3 and k_1 was shown in Figure 4.37 to not work, so the only alternative is to place k_2 as the new root. This forces k_1 to be k_2's left child and k_3 to be its right child, and it also completely determines the resulting locations of the four subtrees. It is easy to see that the resulting tree satisfies the AVL tree property, and as was the case with the single rotation, it restores the height to what it was before the insertion, thus guaranteeing that all rebalancing and height updating is complete. Figure 4.39 shows that the symmetric case 3 can also be fixed

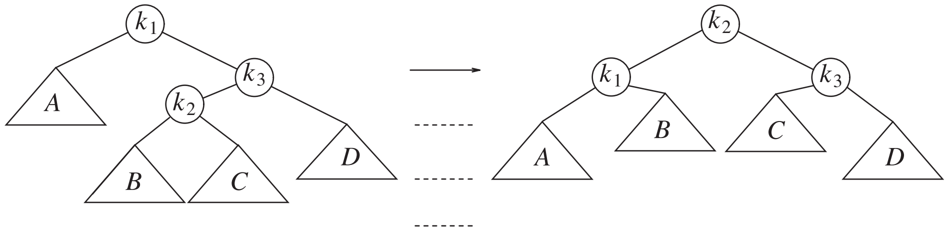
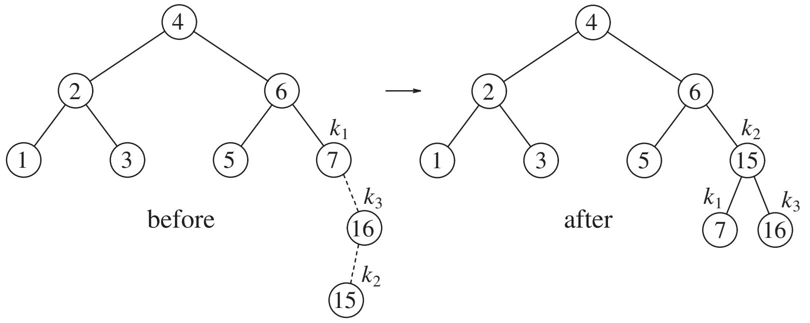


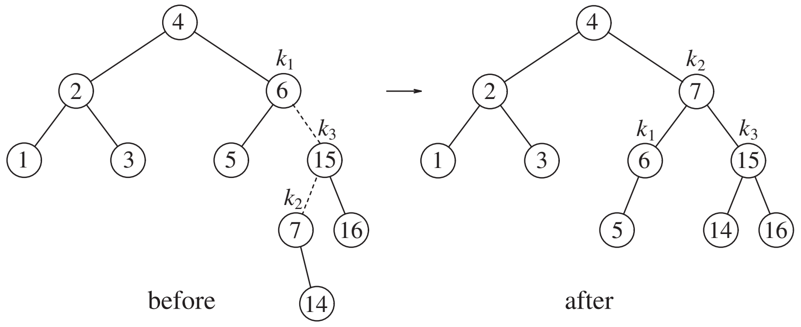
Figure 4.39 Right–left double rotation to fix case 3

by a double rotation. In both cases the effect is the same as rotating between α's child and grandchild, and then between α and its new child.

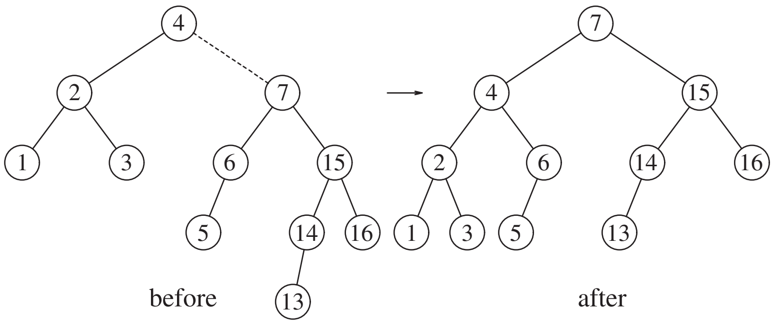
We will continue our previous example by inserting 10 through 16 in reverse order, followed by 8 and then 9. Inserting 16 is easy, since it does not destroy the balance property, but inserting 15 causes a height imbalance at node 7. This is case 3, which is solved by a right–left double rotation. In our example, the right–left double rotation will involve 7, 16, and 15. In this case, k_1 is the node with item 7, k_3 is the node with item 16, and k_2 is the node with item 15. Subtrees *A, B, C,* and *D* are empty.



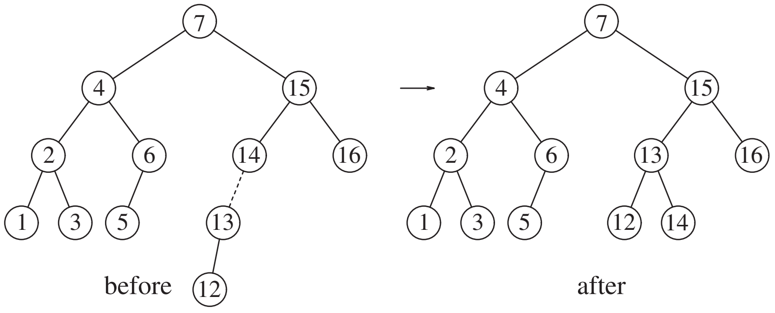
Next we insert 14, which also requires a double rotation. Here the double rotation that will restore the tree is again a right–left double rotation that will involve 6, 15, and 7. In this case, k_1 is the node with item 6, k_2 is the node with item 7, and k_3 is the node with item 15. Subtree *A* is the tree rooted at the node with item 5; subtree *B* is the empty subtree that was originally the left child of the node with item 7, subtree *C* is the tree rooted at the node with item 14, and finally, subtree *D* is the tree rooted at the node with item 16.



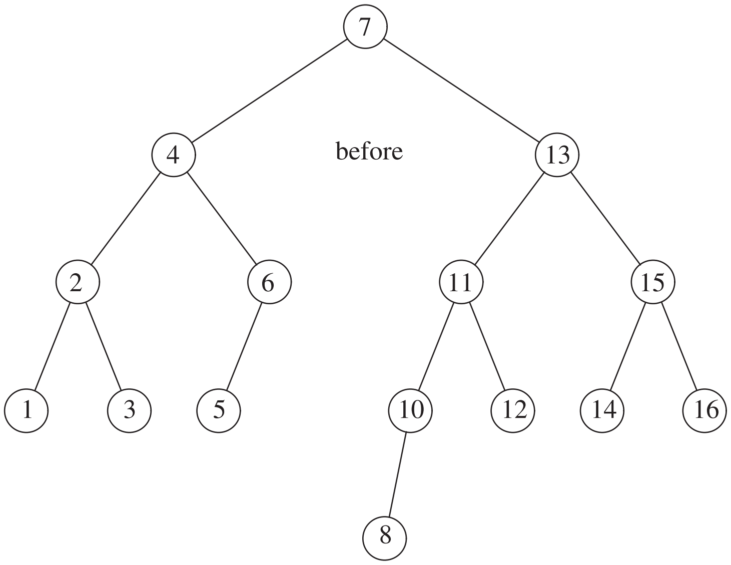
If 13 is now inserted, there is an imbalance at the root. Since 13 is not between 4 and 7, we know that the single rotation will work.



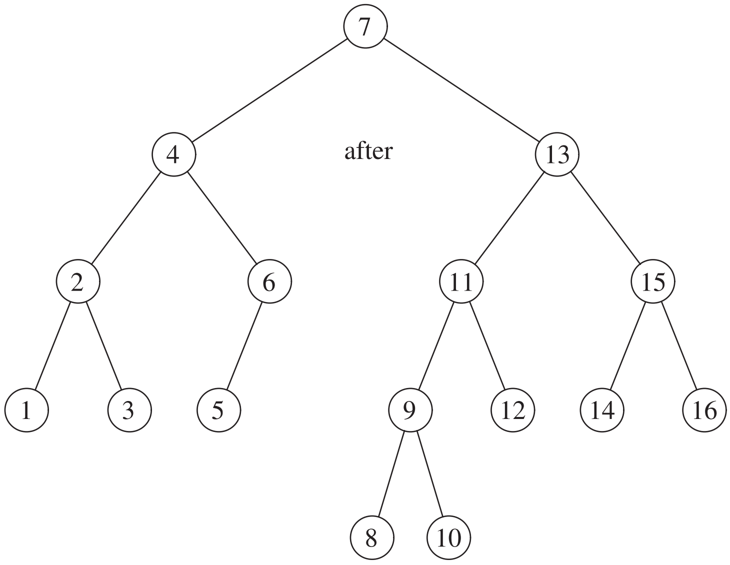
Insertion of 12 will also require a single rotation:



To insert 11, a single rotation needs to be performed, and the same is true for the subsequent insertion of 10. We insert 8 without a rotation, creating an almost perfectly balanced tree:



Finally, we will insert 9 to show the symmetric case of the double rotation. Notice that 9 causes the node containing 10 to become unbalanced. Since 9 is between 10 and 8 (which is 10's child on the path to 9), a double rotation needs to be performed, yielding the following tree:



Let us summarize what happens. The programming details are fairly straightforward except that there are several cases. To insert a new node with item X into an AVL tree T, we recursively insert X into the appropriate subtree of T (let us call this T_{LR}). If the height of T_{LR} does not change, then we are done. Otherwise, if a height imbalance appears in T, we do the appropriate single or double rotation depending on X and the items in T and T_{LR}, update the heights (making the connection from the rest of the tree above), and we are done. Since one rotation always suffices, a carefully coded nonrecursive version generally turns out to be faster than the recursive version, but on modern compilers the difference is not as significant as in the past. However, nonrecursive versions are quite difficult to code correctly, whereas a casual recursive implementation is easily readable.

Another efficiency issue concerns storage of the height information. Since all that is really required is the difference in height, which is guaranteed to be small, we could get by with two bits (to represent $+1$, 0, -1) if we really try. Doing so will avoid repetitive calculation of balance factors but results in some loss of clarity. The resulting code is somewhat more complicated than if the height were stored at each node. If a recursive routine is written, then speed is probably not the main consideration. In this case, the slight speed advantage obtained by storing balance factors hardly seems worth the loss of clarity and relative simplicity. Furthermore, since most machines will align this to at least an 8-bit boundary anyway, there is not likely to be any difference in the amount of space used. An 8-bit (signed) `char` will allow us to store absolute heights of up to 127. Since the tree is balanced, it is inconceivable that this would be insufficient (see the exercises).

```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int       height;

    AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
      : element{ ele }, left{ lt }, right{ rt }, height{ h } { }

    AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
      : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
};
```

Figure 4.40 Node declaration for AVL trees

```
/**
 * Return the height of node t or -1 if nullptr.
 */
int height( AvlNode *t ) const
{
    return t == nullptr ? -1 : t->height;
}
```

Figure 4.41 Function to compute height of an AVL node

With all this, we are ready to write the AVL routines. We show some of the code here; the rest is online. First, we need the `AvlNode` class. This is given in Figure 4.40. We also need a quick function to return the height of a node. This function is necessary to handle the annoying case of a `nullptr` pointer. This is shown in Figure 4.41. The basic insertion routine (see Figure 4.42) adds only a single line at the end that invokes a balancing method. The balancing method applies a single or double rotation if needed, updates the height, and returns the resulting tree.

For the trees in Figure 4.43, `rotateWithLeftChild` converts the tree on the left to the tree on the right, returning a pointer to the new root. `rotateWithRightChild` is symmetric. The code is shown in Figure 4.44.

Similarly, the double rotation pictured in Figure 4.45 can be implemented by the code shown in Figure 4.46.

Since deletion in a binary search tree is somewhat more complicated than insertion, one can assume that deletion in an AVL tree is also more complicated. In a perfect world, one would hope that the deletion routine in Figure 4.26 could easily be modified by changing the last line to return after calling the `balance` method, as was done for insertion. This would yield the code in Figure 4.47. This change works! A deletion could cause one side

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == nullptr )
        t = new AvlNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );

    balance( t );
}

static const int ALLOWED_IMBALANCE = 1;

// Assume t is balanced or within one of being balanced
void balance( AvlNode * & t )
{
    if( t == nullptr )
        return;

    if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
        if( height( t->left->left ) >= height( t->left->right ) )
            rotateWithLeftChild( t );
        else
            doubleWithLeftChild( t );
    else
    if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
        if( height( t->right->right ) >= height( t->right->left ) )
            rotateWithRightChild( t );
        else
            doubleWithRightChild( t );

    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

Figure 4.42 Insertion into an AVL tree

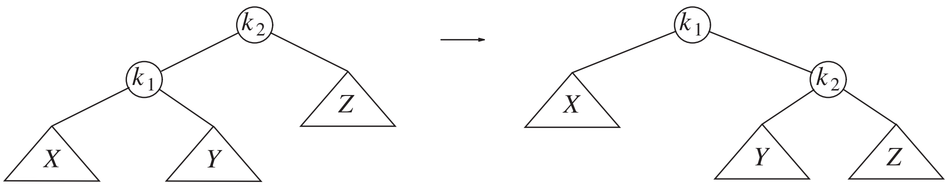


Figure 4.43 Single rotation

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```

Figure 4.44 Routine to perform single rotation

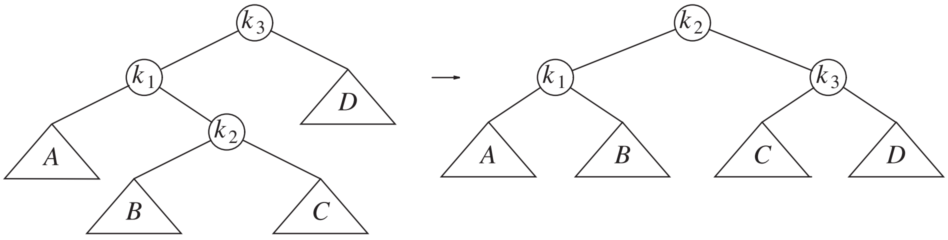


Figure 4.45 Double rotation

of the tree to become two levels shallower than the other side. The case-by-case analysis is similar to the imbalances that are caused by insertion, but not exactly the same. For instance, case 1 in Figure 4.34, which would now reflect a deletion from tree *Z* (rather than an insertion into *X*), must be augmented with the possibility that tree *Y* could be as deep as tree *X*. Even so, it is easy to see that the rotation rebalances this case and the symmetric case 4 in Figure 4.36. Thus the code for balance in Figure 4.42 lines 28 and 34 uses >=

```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Figure 4.46 Routine to perform double rotation

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, AvlNode * & t )
{
   if( t == nullptr )
       return;   // Item not found; do nothing

   if( x < t->element )
       remove( x, t->left );
   else if( t->element < x )
       remove( x, t->right );
   else if( t->left != nullptr && t->right != nullptr )  // Two children
   {
       t->element = findMin( t->right  )->element;
       remove( t->element, t->right );
   }
   else
   {
       AvlNode *oldNode = t;
       t = ( t->left != nullptr ) ? t->left : t->right;
       delete oldNode;
   }

    balance( t );
}
```

Figure 4.47 Deletion in an AVL tree

instead of > specifically to ensure that single rotations are done in these cases rather than double rotations. We leave verification of the remaining cases as an exercise.

4.5 Splay Trees

We now describe a relatively simple data structure known as a **splay tree** that guarantees that any M consecutive tree operations starting from an empty tree take at most $O(M \log N)$ time. Although this guarantee does not preclude the possibility that any *single* operation might take $\Theta(N)$ time, and thus the bound is not as strong as an $O(\log N)$ worst-case bound per operation, the net effect is the same: There are no bad input sequences. Generally, when a sequence of M operations has total worst-case running time of $O(Mf(N))$, we say that the **amortized** running time is $O(f(N))$. Thus, a splay tree has an $O(\log N)$ amortized cost per operation. Over a long sequence of operations, some may take more, some less.

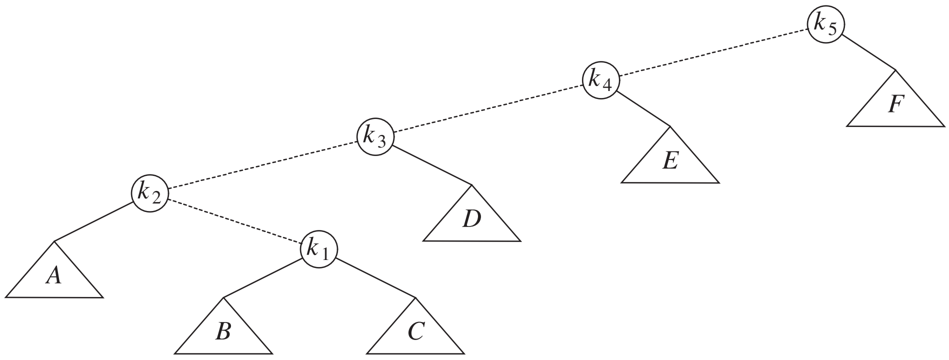
Splay trees are based on the fact that the $O(N)$ worst-case time per operation for binary search trees is not bad, as long as it occurs relatively infrequently. Any one access, even if it takes $\Theta(N)$, is still likely to be extremely fast. The problem with binary search trees is that it is possible, and not uncommon, for a whole sequence of bad accesses to take place. The cumulative running time then becomes noticeable. A search tree data structure with $O(N)$ worst-case time, but a *guarantee* of at most $O(M \log N)$ for any M consecutive operations, is certainly satisfactory, because there are no bad sequences.

If any particular operation is allowed to have an $O(N)$ worst-case time bound, and we still want an $O(\log N)$ amortized time bound, then it is clear that whenever a node is accessed, it must be moved. Otherwise, once we find a deep node, we could keep performing accesses on it. If the node does not change location, and each access costs $\Theta(N)$, then a sequence of M accesses will cost $\Theta(M \cdot N)$.

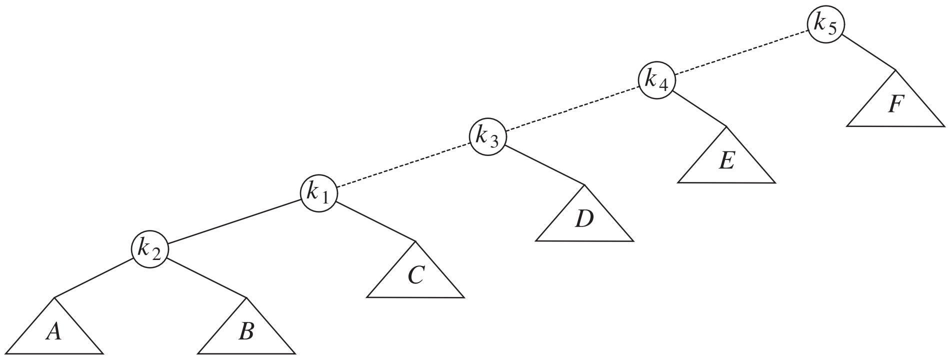
The basic idea of the splay tree is that after a node is accessed, it is pushed to the root by a series of AVL tree rotations. Notice that if a node is deep, there are many nodes on the path that are also relatively deep, and by restructuring we can make future accesses cheaper on all these nodes. Thus, if the node is unduly deep, then we want this restructuring to have the side effect of balancing the tree (to some extent). Besides giving a good time bound in theory, this method is likely to have practical utility, because in many applications, when a node is accessed, it is likely to be accessed again in the near future. Studies have shown that this happens much more often than one would expect. Splay trees also do not require the maintenance of height or balance information, thus saving space and simplifying the code to some extent (especially when careful implementations are written).

4.5.1 A Simple Idea (That Does Not Work)

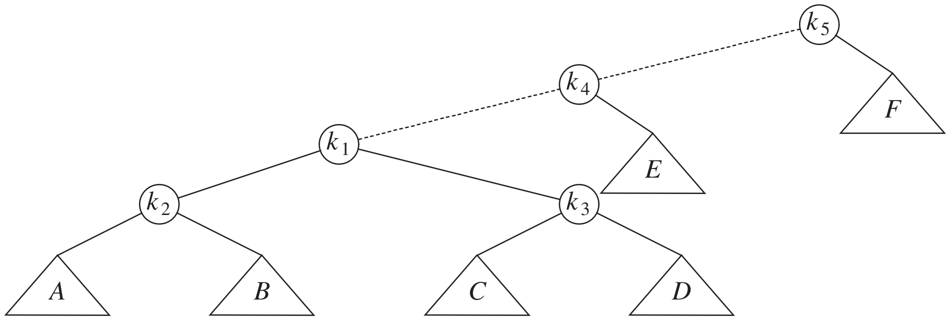
One way of performing the restructuring described above is to perform single rotations, bottom up. This means that we rotate every node on the access path with its parent. As an example, consider what happens after an access (a `find`) on k_1 in the following tree:



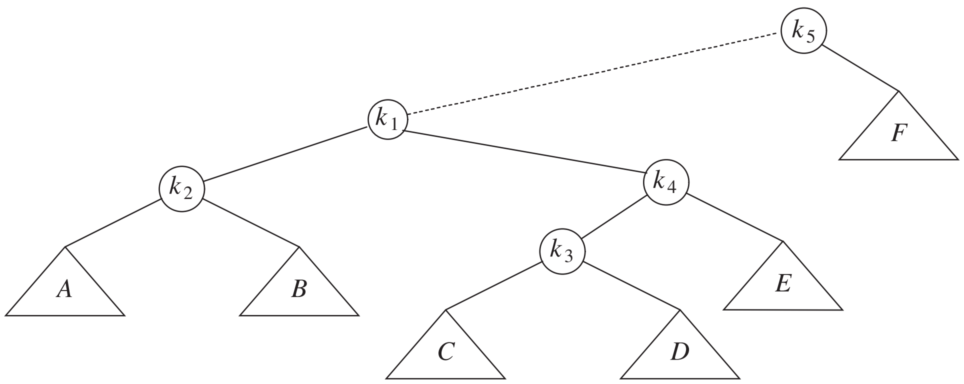
The access path is dashed. First, we would perform a single rotation between k_1 and its parent, obtaining the following tree:

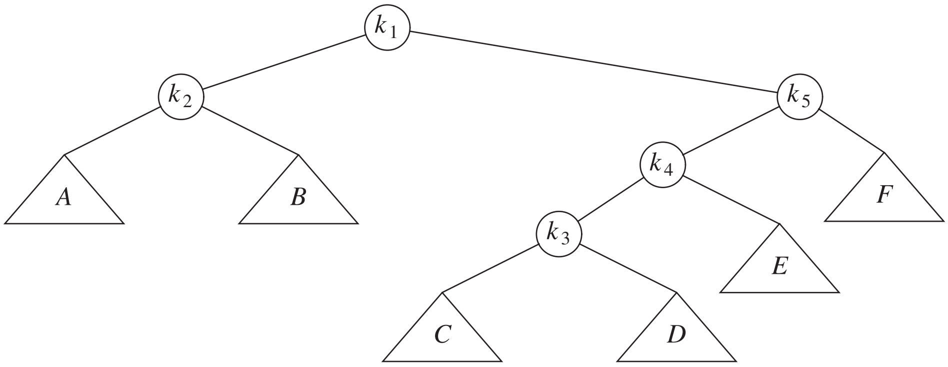


Then, we rotate between k_1 and k_3, obtaining the next tree:



Then two more rotations are performed until we reach the root:





These rotations have the effect of pushing k_1 all the way to the root, so that future accesses on k_1 are easy (for a while). Unfortunately, it has pushed another node (k_3) almost as deep as k_1 used to be. An access on that node will then push another node deep, and so on. Although this strategy makes future accesses of k_1 cheaper, it has not significantly improved the situation for the other nodes on the (original) access path. It turns out that it is possible to prove that using this strategy, there is a sequence of M operations requiring $\Omega(M \cdot N)$ time, so this idea is not quite good enough. The simplest way to show this is to consider the tree formed by inserting keys $1, 2, 3, \ldots, N$ into an initially empty tree (work this example out). This gives a tree consisting of only left children. This is not necessarily bad, though, since the time to build this tree is $O(N)$ total. The bad part is that accessing the node with key 1 takes N units of time, where each node on the access path counts as one unit. After the rotations are complete, an access of the node with key 2 takes N units of time, key 3 takes $N - 1$ units, and so on. The total for accessing all the keys in order is $N + \sum_{i=2}^{N} i = \Omega(N^2)$. After they are accessed, the tree reverts to its original state, and we can repeat the sequence.

4.5.2 Splaying

The splaying strategy is similar to the rotation idea above, except that we are a little more selective about how rotations are performed. We will still rotate bottom up along the access

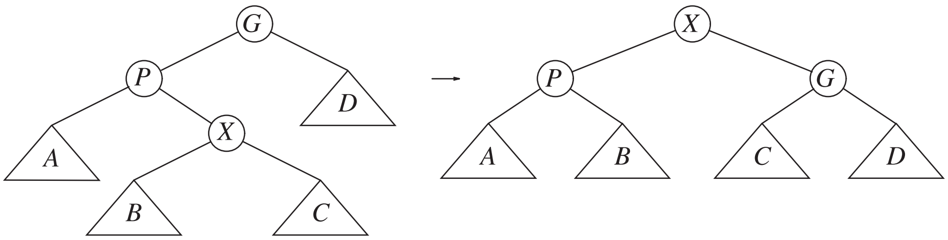


Figure 4.48 Zig-zag

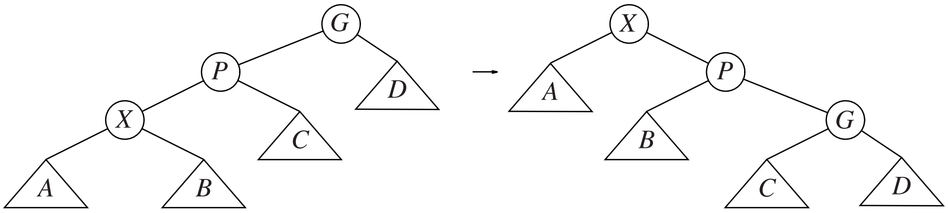
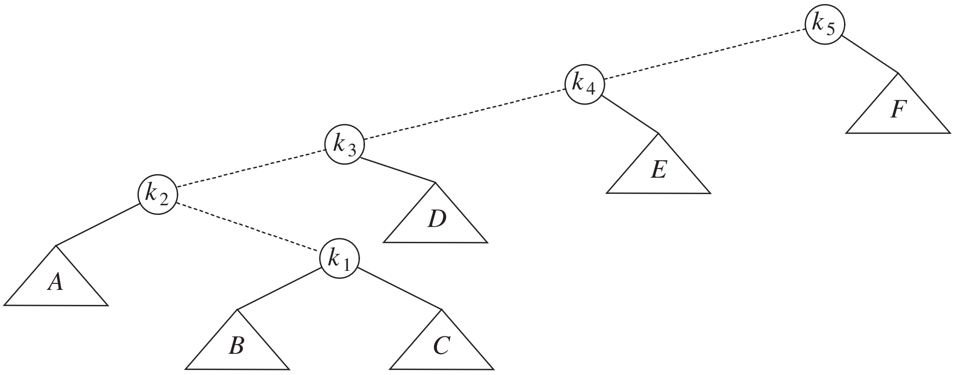


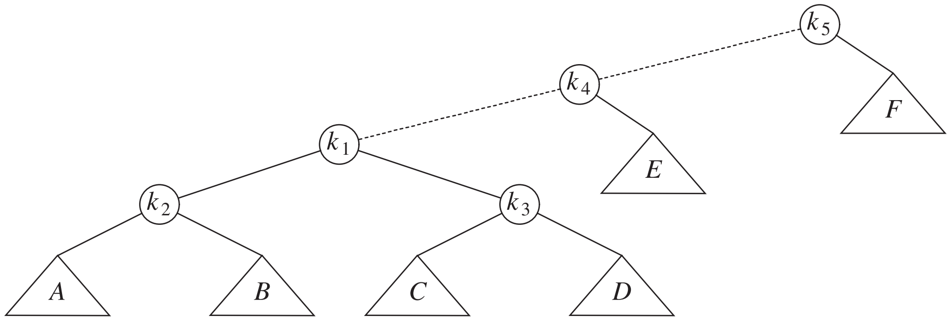
Figure 4.49 Zig-zig

path. Let X be a (non-root) node on the access path at which we are rotating. If the parent of X is the root of the tree, we merely rotate X and the root. This is the last rotation along the access path. Otherwise, X has both a parent (P) and a grandparent (G), and there are two cases, plus symmetries, to consider. The first case is the **zig-zag** case (see Fig. 4.48). Here X is a right child and P is a left child (or vice versa). If this is the case, we perform a double rotation, exactly like an AVL double rotation. Otherwise, we have a **zig-zig** case: X and P are both left children (or, in the symmetric case, both right children). In that case, we transform the tree on the left of Figure 4.49 to the tree on the right.

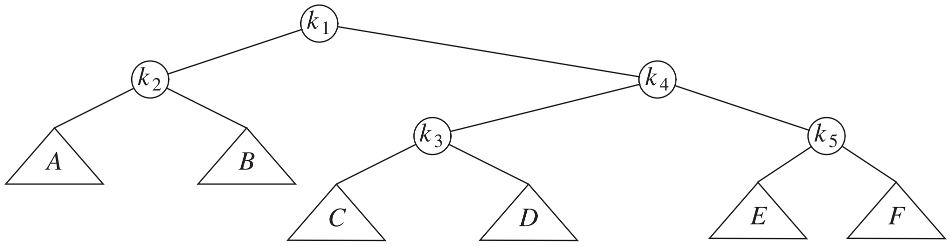
As an example, consider the tree from the last example, with a `contains` on k_1:



The first splay step is at k_1 and is clearly a *zig-zag*, so we perform a standard AVL double rotation using k_1, k_2, and k_3. The resulting tree follows:



The next splay step at k_1 is a *zig-zig*, so we do the *zig-zig* rotation with k_1, k_4, and k_5, obtaining the final tree:



Although it is hard to see from small examples, splaying not only moves the accessed node to the root but also has the effect of roughly halving the depth of most nodes on the access path (some shallow nodes are pushed down at most two levels).

To see the difference that splaying makes over simple rotation, consider again the effect of inserting items $1, 2, 3, \ldots, N$ into an initially empty tree. This takes a total of $O(N)$, as before, and yields the same tree as simple rotations. Figure 4.50 shows the result of splaying at the node with item 1. The difference is that after an access of the node with item 1, which

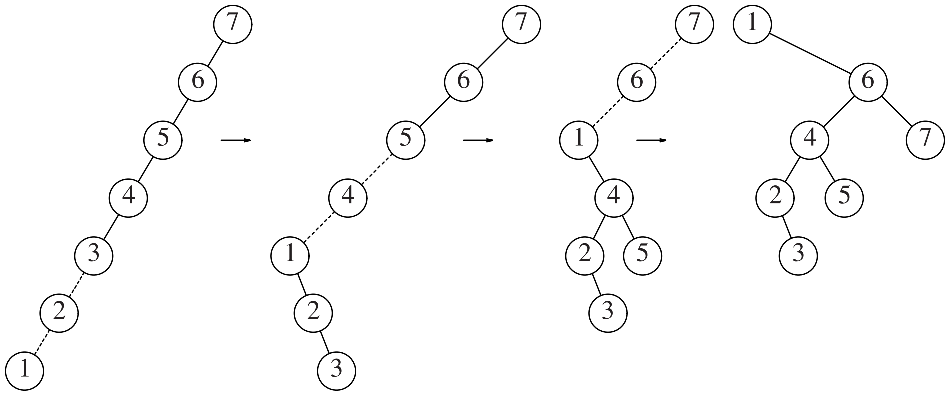


Figure 4.50 Result of splaying at node 1

takes N units, the access on the node with item 2 will only take about $N/2$ units instead of N units; there are no nodes quite as deep as before.

An access on the node with item 2 will bring nodes to within $N/4$ of the root, and this is repeated until the depth becomes roughly $\log N$ (an example with $N = 7$ is too small to see the effect well). Figures 4.51 to 4.59 show the result of accessing items 1 through 9 in a 32-node tree that originally contains only left children. Thus we do not get the same bad behavior from splay trees that is prevalent in the simple rotation strategy. (Actually, this turns out to be a very good case. A rather complicated proof shows that for this example, the N accesses take a total of $O(N)$ time.)

These figures highlight the fundamental and crucial property of splay trees. When access paths are long, thus leading to a longer-than-normal search time, the rotations tend to be good for future operations. When accesses are cheap, the rotations are not as good and can be bad. The extreme case is the initial tree formed by the insertions. All the insertions were constant-time operations leading to a bad initial tree. At that point in time, we had a very bad tree, but we were running ahead of schedule and had the compensation of less total running time. Then a couple of really horrible accesses left a nearly balanced tree, but the cost was that we had to give back some of the time that had been saved. The main theorem, which we will prove in Chapter 11, is that we never fall behind a pace of $O(\log N)$ per operation: We are always on schedule, even though there are occasionally bad operations.

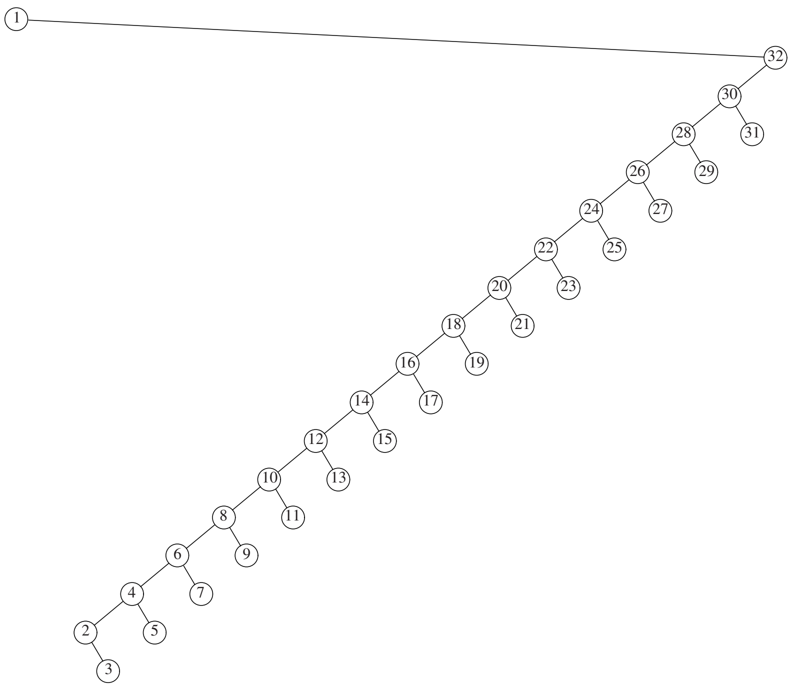


Figure 4.51 Result of splaying at node 1 a tree of all left children

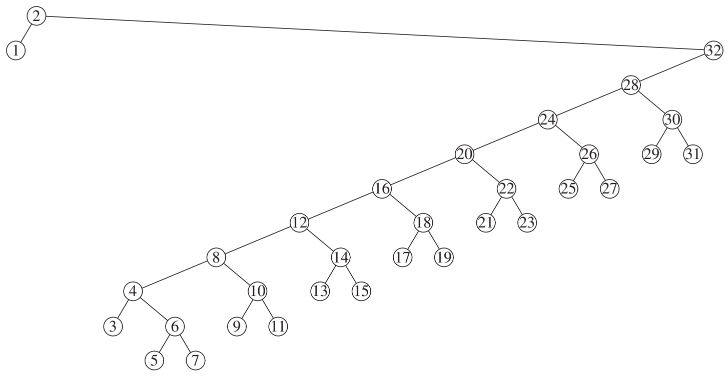


Figure 4.52 Result of splaying the previous tree at node 2

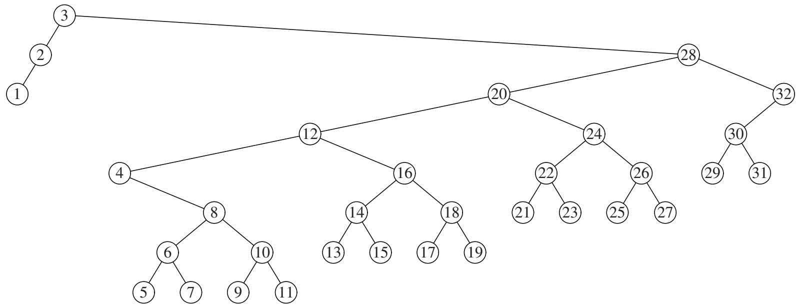


Figure 4.53 Result of splaying the previous tree at node 3

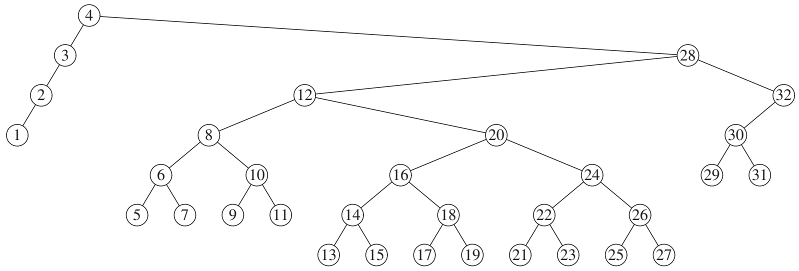


Figure 4.54 Result of splaying the previous tree at node 4

We can perform deletion by accessing the node to be deleted. This puts the node at the root. If it is deleted, we get two subtrees T_L and T_R (left and right). If we find the largest element in T_L (which is easy), then this element is rotated to the root of T_L, and T_L will now have a root with no right child. We can finish the deletion by making T_R the right child.

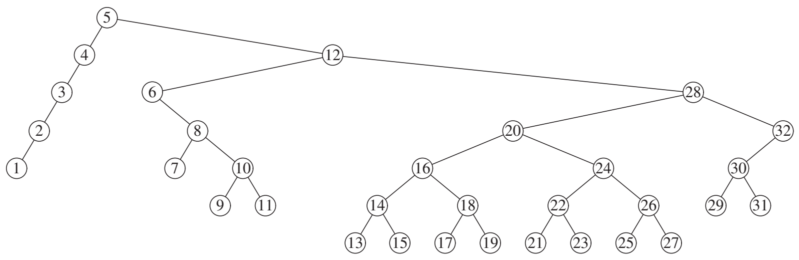


Figure 4.55 Result of splaying the previous tree at node 5

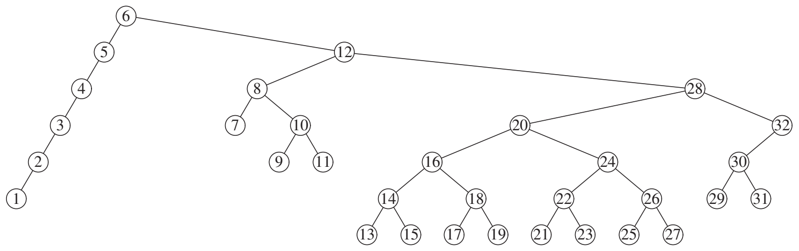


Figure 4.56 Result of splaying the previous tree at node 6

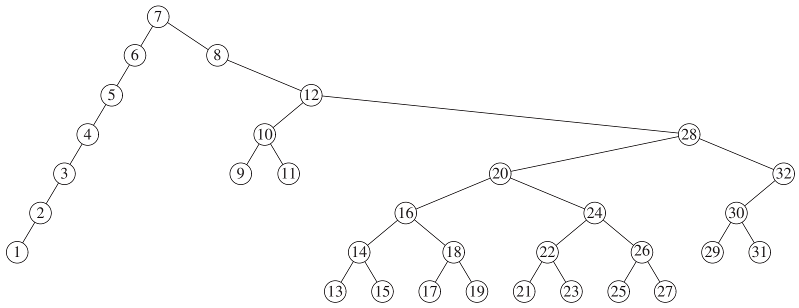


Figure 4.57 Result of splaying the previous tree at node 7

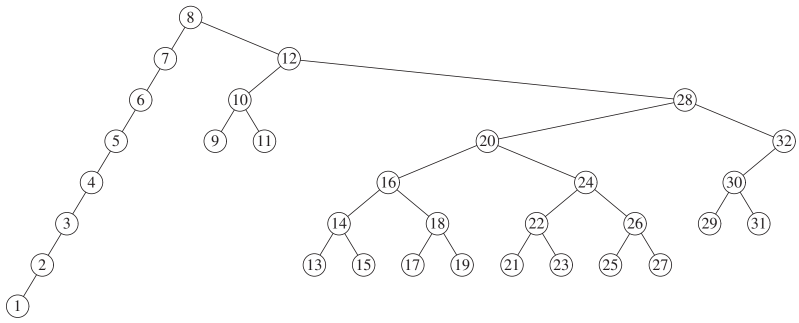


Figure 4.58 Result of splaying the previous tree at node 8

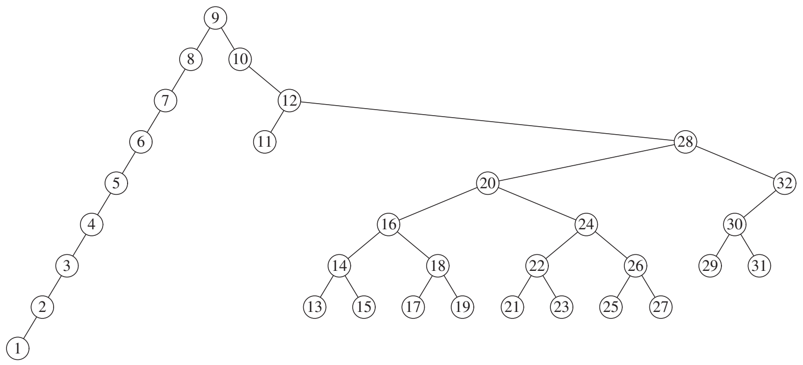


Figure 4.59 Result of splaying the previous tree at node 9

The analysis of splay trees is difficult, because it must take into account the ever-changing structure of the tree. On the other hand, splay trees are much simpler to program than most balanced search trees, since there are fewer cases to consider and no balance information to maintain. Some empirical evidence suggests that this translates into faster code in practice, although the case for this is far from complete. Finally, we point out that there are several variations of splay trees that can perform even better in practice. One variation is completely coded in Chapter 12.

4.6 Tree Traversals (Revisited)

Because of the ordering information in a binary search tree, it is simple to list all the items in sorted order. The recursive function in Figure 4.60 does the real work.

Convince yourself that this function works. As we have seen before, this kind of routine when applied to trees is known as an **inorder traversal** (which makes sense, since it lists the items in order). The general strategy of an inorder traversal is to process the left subtree first, then perform processing at the current node, and finally process the right subtree. The interesting part about this algorithm, aside from its simplicity, is that the total running time is $O(N)$. This is because there is constant work being performed at every node in the tree. Each node is visited once, and the work performed at each node is testing against `nullptr`, setting up two function calls, and doing an output statement. Since there is constant work per node and N nodes, the running time is $O(N)$.

Sometimes we need to process both subtrees first before we can process a node. For instance, to compute the height of a node, we need to know the height of the subtrees first. The code in Figure 4.61 computes this. Since it is always a good idea to check the special cases—and crucial when recursion is involved—notice that the routine will declare the height of a leaf to be zero, which is correct. This general order of traversal, which we have also seen before, is known as a **postorder traversal**. Again, the total running time is $O(N)$, because constant work is performed at each node.

```
/**
 * Print the tree contents in sorted order.
 */
void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

/**
 * Internal method to print a subtree rooted at t in sorted order.
 */
void printTree( BinaryNode *t, ostream & out ) const
{
    if( t != nullptr )
    {
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}
```

Figure 4.60 Routine to print a binary search tree in order

```
/**
 * Internal method to compute the height of a subtree rooted at t.
 */
int height( BinaryNode *t )
{
    if( t == nullptr )
        return -1;
    else
        return 1 + max( height( t->left ), height( t->right ) );
}
```

Figure 4.61 Routine to compute the height of a tree using a postorder traversal

The third popular traversal scheme that we have seen is **preorder traversal**. Here, the node is processed before the children. This could be useful, for example, if you wanted to label each node with its depth.

The common idea in all of these routines is that you handle the `nullptr` case first and then the rest. Notice the lack of extraneous variables. These routines pass only the pointer to the node that roots the subtree, and do not declare or pass any extra variables. The more compact the code, the less likely that a silly bug will turn up. A fourth, less often used, traversal (which we have not seen yet) is **level-order traversal**. In a level-order traversal, all nodes at depth d are processed before any node at depth $d + 1$. Level-order traversal differs from the other traversals in that it is not done recursively; a queue is used, instead of the implied stack of recursion.

4.7 B-Trees

So far, we have assumed that we can store an entire data structure in the main memory of a computer. Suppose, however, that we have more data than can fit in main memory, and, as a result, must have the data structure reside on disk. When this happens, the rules of the game change, because the Big-Oh model is no longer meaningful.

The problem is that a Big-Oh analysis assumes that all operations are equal. However, this is not true, especially when disk I/O is involved. Modern computers execute billions of instructions per second. That is pretty fast, mainly because the speed depends largely on electrical properties. On the other hand, a disk is mechanical. Its speed depends largely on the time it takes to spin the disk and to move a disk head. Many disks spin at 7,200 RPM. Thus, in 1 min it makes 7,200 revolutions; hence, one revolution occurs in 1/120 of a second, or 8.3 ms. On average, we might expect that we have to spin a disk halfway to find what we are looking for, but this is compensated by the time to move the disk head, so we get an access time of 8.3 ms. (This is a very charitable estimate; 9–11 ms access times are more common.) Consequently, we can do approximately 120 disk accesses per second. This sounds pretty good, until we compare it with the processor speed. What we have is billions instructions equal to 120 disk accesses. Of course, everything here is a rough calculation, but the relative speeds are pretty clear: Disk accesses are incredibly expensive. Furthermore, processor speeds are increasing at a much faster rate than disk speeds (it is disk *sizes* that are increasing quite quickly). So we are willing to do lots of calculations just to save a disk access. In almost all cases, it is the number of disk accesses that will dominate the running time. Thus, if we halve the number of disk accesses, the running time will halve.

Here is how the typical search tree performs on disk: Suppose we want to access the driving records for citizens in the state of Florida. We assume that we have 10,000,000 items, that each key is 32 bytes (representing a name), and that a record is 256 bytes. We assume this does not fit in main memory and that we are 1 of 20 users on a system (so we have 1/20 of the resources). Thus, in 1 sec we can execute many millions of instructions or perform six disk accesses.

The unbalanced binary search tree is a disaster. In the worst case, it has linear depth and thus could require 10,000,000 disk accesses. On average, a successful search would require $1.38 \log N$ disk accesses, and since $\log 10000000 \approx 24$, an average search would require 32 disk accesses, or 5 sec. In a typical randomly constructed tree, we would expect that a few nodes are three times deeper; these would require about 100 disk accesses, or 16 sec. An AVL tree is somewhat better. The worst case of $1.44 \log N$ is unlikely to occur, and the typical case is very close to $\log N$. Thus an AVL tree would use about 25 disk accesses on average, requiring 4 sec.

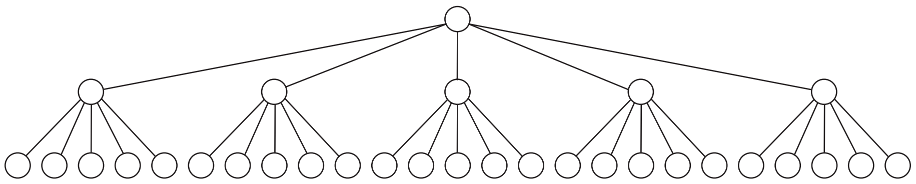


Figure 4.62 5-ary tree of 31 nodes has only three levels

We want to reduce the number of disk accesses to a very small constant, such as three or four. We are willing to write complicated code to do this, because machine instructions are essentially free, as long as we are not ridiculously unreasonable. It should probably be clear that a binary search tree will not work, since the typical AVL tree is close to optimal height. We cannot go below $\log N$ using a binary search tree. The solution is intuitively simple: If we have more branching, we have less height. Thus, while a perfect binary tree of 31 nodes has five levels, a 5-ary tree of 31 nodes has only three levels, as shown in Figure 4.62. An **M-ary search tree** allows M-way branching. As branching increases, the depth decreases. Whereas a complete binary tree has height that is roughly $\log_2 N$, a complete M-ary tree has height that is roughly $\log_M N$.

We can create an M-ary search tree in much the same way as a binary search tree. In a binary search tree, we need one key to decide which of two branches to take. In an M-ary search tree, we need $M - 1$ keys to decide which branch to take. To make this scheme efficient in the worst case, we need to ensure that the M-ary search tree is balanced in some way. Otherwise, like a binary search tree, it could degenerate into a linked list. Actually, we want an even more restrictive balancing condition. That is, we do not want an M-ary search tree to degenerate to even a binary search tree, because then we would be stuck with $\log N$ accesses.

One way to implement this is to use a **B-tree.** The basic B-tree[3] is described here. Many variations and improvements are known, and an implementation is somewhat complex because there are quite a few cases. However, it is easy to see that, in principle, a B-tree guarantees only a few disk accesses.

A B-tree of order M is an M-ary tree with the following properties:[4]

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M - 1$ keys to guide the searching; key i represents the smallest key in subtree $i + 1$.
3. The root is either a leaf or has between two and M children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data items, for some L (the determination of L is described shortly).

[3] What is described is popularly known as a B^+ tree.

[4] Rules 3 and 5 must be relaxed for the first L insertions.

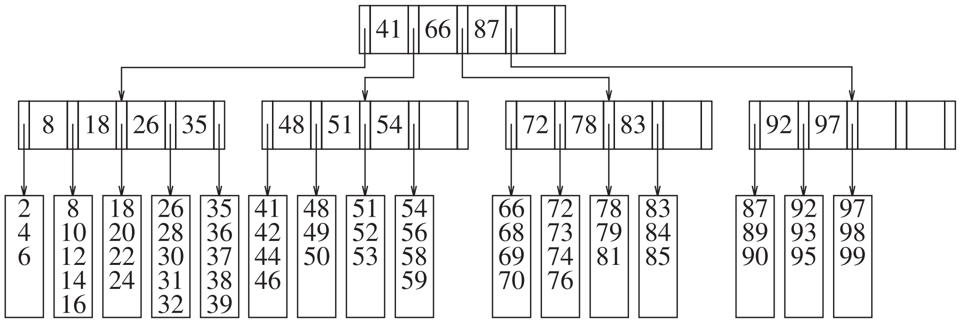


Figure 4.63 B-tree of order 5

An example of a B-tree of order 5 is shown in Figure 4.63. Notice that all nonleaf nodes have between three and five children (and thus between two and four keys); the root could possibly have only two children. Here, we have $L = 5$. It happens that L and M are the same in this example, but this is not necessary. Since L is 5, each leaf has between three and five data items. Requiring nodes to be half full guarantees that the B-tree does not degenerate into a simple binary tree. Although there are various definitions of B-trees that change this structure, mostly in minor ways, this definition is one of the popular forms.

Each node represents a disk block, so we choose M and L on the basis of the size of the items that are being stored. As an example, suppose one block holds 8,192 bytes. In our Florida example, each key uses 32 bytes. In a B-tree of order M, we would have $M-1$ keys, for a total of $32M - 32$ bytes, plus M branches. Since each branch is essentially a number of another disk block, we can assume that a branch is 4 bytes. Thus the branches use $4M$ bytes. The total memory requirement for a nonleaf node is thus $36M-32$. The largest value of M for which this is no more than 8,192 is 228. Thus we would choose $M = 228$. Since each data record is 256 bytes, we would be able to fit 32 records in a block. Thus we would choose $L = 32$. We are guaranteed that each leaf has between 16 and 32 data records and that each internal node (except the root) branches in at least 114 ways. Since there are 10,000,000 records, there are, at most, 625,000 leaves. Consequently, in the worst case, leaves would be on level 4. In more concrete terms, the worst-case number of accesses is given by approximately $\log_{M/2} N$, give or take 1. (For example, the root and the next level could be cached in main memory, so that over the long run, disk accesses would be needed only for level 3 and deeper.)

The remaining issue is how to add and remove items from the B-tree. The ideas involved are sketched next. Note that many of the themes seen before recur.

We begin by examining insertion. Suppose we want to insert 57 into the B-tree in Figure 4.63. A search down the tree reveals that it is not already in the tree. We can add it to the leaf as a fifth item. Note that we may have to reorganize all the data in the leaf to do this. However, the cost of doing this is negligible when compared to that of the disk access, which in this case also includes a disk write.

Of course, that was relatively painless, because the leaf was not already full. Suppose we now want to insert 55. Figure 4.64 shows a problem: The leaf where 55 wants to go is already full. The solution is simple: Since we now have $L+1$ items, we split them into two

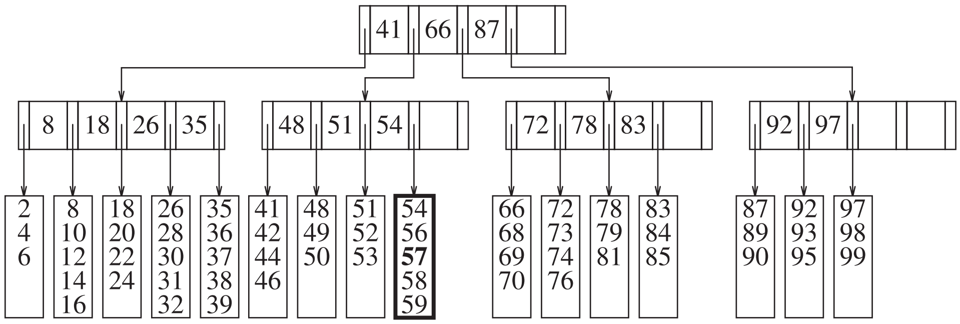


Figure 4.64 B-tree after insertion of 57 into the tree in Figure 4.63

leaves, both guaranteed to have the minimum number of data records needed. We form two leaves with three items each. Two disk accesses are required to write these leaves, and a third disk access is required to update the parent. Note that in the parent, both keys and branches change, but they do so in a controlled way that is easily calculated. The resulting B-tree is shown in Figure 4.65. Although splitting nodes is time-consuming because it requires at least two additional disk writes, it is a relatively rare occurrence. If L is 32, for example, then when a node is split, two leaves with 16 and 17 items, respectively, are created. For the leaf with 17 items, we can perform 15 more insertions without another split. Put another way, for every split, there are roughly $L/2$ nonsplits.

The node splitting in the previous example worked because the parent did not have its full complement of children. But what would happen if it did? Suppose, for example, that we insert 40 into the B-tree in Figure 4.65. We must split the leaf containing the keys 35 through 39, and now 40, into two leaves. But doing this would give the parent six children, and it is allowed only five. The solution is to split the parent. The result of this is shown in Figure 4.66. When the parent is split, we must update the values of the keys and also the parent's parent, thus incurring an additional two disk writes (so this insertion costs five disk writes). However, once again, the keys change in a very controlled manner, although the code is certainly not simple because of a host of cases.

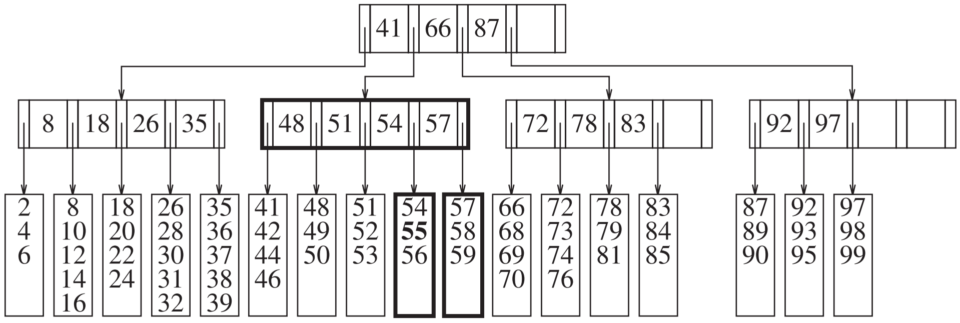


Figure 4.65 Insertion of 55 into the B-tree in Figure 4.64 causes a split into two leaves

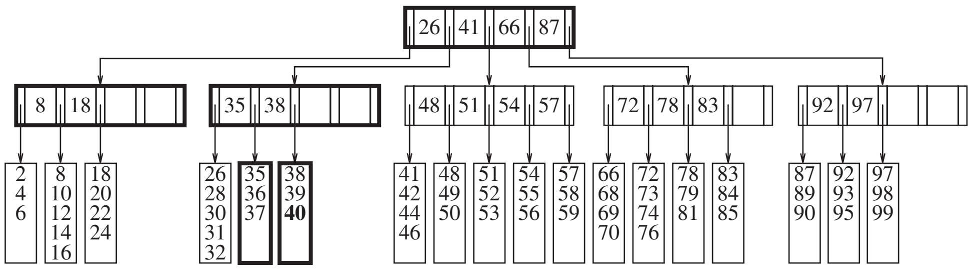


Figure 4.66 Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

When a nonleaf node is split, as is the case here, its parent gains a child. What if the parent already has reached its limit of children? Then we continue splitting nodes up the tree until either we find a parent that does not need to be split or we reach the root. If we split the root, then we have two roots. Obviously, this is unacceptable, but we can create a new root that has the split roots as its two children. This is why the root is granted the special two-child minimum exemption. It also is the only way that a B-tree gains height. Needless to say, splitting all the way up to the root is an exceptionally rare event. This is because a tree with four levels indicates that the root has been split three times throughout the entire sequence of insertions (assuming no deletions have occurred). In fact, the splitting of any nonleaf node is also quite rare.

There are other ways to handle the overflowing of children. One technique is to put a child up for adoption should a neighbor have room. To insert 29 into the B-tree in Figure 4.66, for example, we could make room by moving 32 to the next leaf. This technique requires a modification of the parent, because the keys are affected. However, it tends to keep nodes fuller and saves space in the long run.

We can perform deletion by finding the item that needs to be removed and then removing it. The problem is that if the leaf it was in had the minimum number of data items, then it is now below the minimum. We can rectify this situation by adopting a neighboring item, if the neighbor is not itself at its minimum. If it is, then we can combine with the neighbor to form a full leaf. Unfortunately, this means that the parent has lost a child. If this causes the parent to fall below its minimum, then it follows the same strategy. This process could

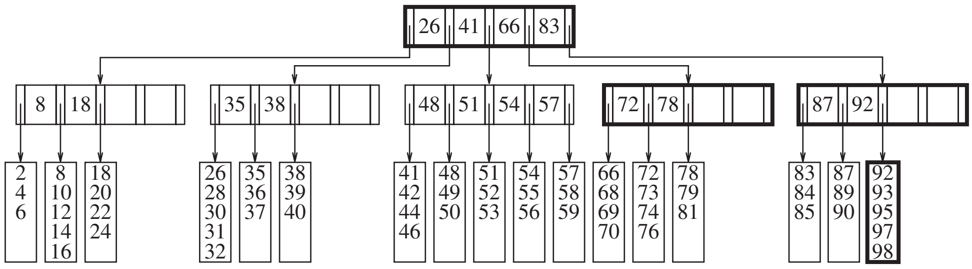


Figure 4.67 B-tree after the deletion of 99 from the B-tree in Figure 4.66

percolate all the way up to the root. The root cannot have just one child (and even if this were allowed, it would be silly). If a root is left with one child as a result of the adoption process, then we remove the root and make its child the new root of the tree. This is the only way for a B-tree to lose height. For example, suppose we want to remove 99 from the B-tree in Figure 4.66. Since the leaf has only two items and its neighbor is already at its minimum of three, we combine the items into a new leaf of five items. As a result, the parent has only two children. However, it can adopt from a neighbor, because the neighbor has four children. As a result, both have three children. The result is shown in Figure 4.67.

4.8 Sets and Maps in the Standard Library

The STL containers discussed in Chapter 3—namely, `vector` and `list`—are inefficient for searching. Consequently, the STL provides two additional containers, `set` and `map`, that guarantee logarithmic cost for basic operations such as insertion, deletion, and searching.

4.8.1 Sets

The `set` is an ordered container that does not allow duplicates. Many of the idioms used to access items in `vector` and `list` also work for a `set`. Specifically, nested in the `set` are `iterator` and `const_iterator` types that allow traversal of the `set`, and several methods from `vector` and `list` are identically named in `set`, including `begin`, `end`, `size`, and `empty`. The `print` function template described in Figure 3.6 will work if passed a `set`.

The unique operations required by the `set` are the abilities to insert, remove, and perform a basic search (efficiently).

The insert routine is aptly named `insert`. However, because a `set` does not allow duplicates, it is possible for the `insert` to fail. As a result, we want the return type to be able to indicate this with a Boolean variable. However, `insert` has a more complicated return type than a `bool`. This is because `insert` also returns an `iterator` that represents where `x` is when `insert` returns. This `iterator` represents either the newly inserted item or the existing item that caused the `insert` to fail, and it is useful, because knowing the position of the item can make removing it more efficient by avoiding the search and getting directly to the node containing the item.

The STL defines a class template called `pair` that is little more than a `struct` with members `first` and `second` to access the two items in the `pair`. There are two different `insert` routines:

```
pair<iterator,bool> insert( const Object & x );
pair<iterator,bool> insert( iterator hint, const Object & x );
```

The one-parameter `insert` behaves as described above. The two-parameter `insert` allows the specification of a hint, which represents the position where `x` should go. If the hint is accurate, the insertion is fast, often $O(1)$. If not, the insertion is done using the normal insertion algorithm and performs comparably with the one-parameter `insert`. For instance, the following code might be faster using the two-parameter `insert` rather than the one-parameter `insert`:

```
set<int> s;
for( int i = 0; i < 1000000; ++i )
    s.insert( s.end( ), i );
```

There are several versions of `erase`:

```
int erase( const Object & x );
iterator erase( iterator itr );
iterator erase( iterator start, iterator end );
```

The first one-parameter `erase` removes `x` (if found) and returns the number of items actually removed, which is obviously either 0 or 1. The second one-parameter `erase` behaves the same as in `vector` and `list`. It removes the object at the position given by the `iterator`, returns an `iterator` representing the element that followed `itr` immediately prior to the call to `erase`, and invalidates `itr`, which becomes stale. The two-parameter `erase` behaves the same as in a `vector` or `list`, removing all the items starting at `start`, up to but not including the item at `end`.

For searching, rather than a `contains` routine that returns a Boolean variable, the `set` provides a `find` routine that returns an `iterator` representing the location of the item (or the endmarker if the search fails). This provides considerably more information, at no cost in running time. The signature of `find` is

```
iterator find( const Object & x ) const;
```

By default, ordering uses the `less<Object>` function object, which itself is implemented by invoking `operator<` for the `Object`. An alternative ordering can be specified by instantiating the `set` template with a function object type. For instance, we can create a `set` that stores `string` objects, ignoring case distinctions by using the `CaseInsensitiveCompare` function object coded in Figure 1.25. In the following code, the `set s` has size 1:

```
set<string,CaseInsensitiveCompare> s;
s.insert( "Hello" ); s.insert( "HeLLo" );
cout << "The size is: " << s.size( ) << endl;
```

4.8.2 Maps

A `map` is used to store a collection of ordered entries that consists of keys and their values. Keys must be unique, but several keys can map to the same values. Thus values need not be unique. The keys in the `map` are maintained in logically sorted order.

The `map` behaves like a `set` instantiated with a `pair`, whose comparison function refers only to the key.[5] Thus it supports `begin`, `end`, `size`, and `empty`, but the underlying iterator is a key-value pair. In other words, for an `iterator itr`, `*itr` is of type `pair<KeyType,ValueType>`. The `map` also supports `insert`, `find`, and `erase`. For `insert`, one must provide a `pair<KeyType,ValueType>` object. Although `find` requires only a key, the

[5] Like a `set`, an optional template parameter can be used to specify a comparison function that differs from `less<KeyType>`.

`iterator` it returns references a `pair`. Using only these operations is often not worthwhile because the syntactic baggage can be expensive.

Fortunately, the `map` has an important extra operation that yields simple syntax. The array-indexing operator is overloaded for `map`s as follows:

```
ValueType & operator[] ( const KeyType & key );
```

The semantics of `operator[]` are as follows. If `key` is present in the `map`, a reference to the corresponding value is returned. If `key` is not present in the `map`, it is inserted with a default value into the `map` and then a reference to the inserted default value is returned. The default value is obtained by applying a zero-parameter constructor or is zero for the primitive types. These semantics do not allow an accessor version of `operator[]`, so `operator[]` cannot be used on a `map` that is constant. For instance, if a `map` is passed by constant reference, inside the routine, `operator[]` is unusable.

The code snippet in Figure 4.68 illustrates two techniques to access items in a `map`. First, observe that at line 3, the left-hand side invokes `operator[]`, thus inserting `"Pat"` and a `double` of value 0 into the `map`, and returning a reference to that `double`. Then the assignment changes that `double` inside the `map` to 75000. Line 4 outputs 75000. Unfortunately, line 5 inserts `"Jan"` and a salary of 0.0 into the `map` and then prints it. This may or may not be the proper thing to do, depending on the application. If it is important to distinguish between items that are in the `map` and those not in the `map`, or if it is important not to `insert` into the `map` (because it is immutable), then an alternate approach shown at lines 7 to 12 can be used. There we see a call to `find`. If the key is not found, the `iterator` is the endmarker and can be tested. If the key is found, we can access the second item in the pair referenced by the `iterator`, which is the value associated with the key. We could also assign to `itr->second` if, instead of a `const_iterator`, `itr` is an `iterator`.

4.8.3 Implementation of `set` and `map`

C++ requires that `set` and `map` support the basic `insert`, `erase`, and `find` operations in logarithmic worst-case time. Consequently, the underlying implementation is a balanced

```
 1      map<string,double> salaries;
 2
 3      salaries[ "Pat" ] = 75000.00;
 4      cout << salaries[ "Pat" ] << endl;
 5      cout << salaries[ "Jan" ] << endl;
 6
 7      map<string,double>::const_iterator itr;
 8      itr = salaries.find( "Chris" );
 9      if( itr == salaries.end( ) )
10          cout << "Not an employee of this company!" << endl;
11      else
12          cout << itr->second << endl;
```

Figure 4.68 Accessing values in a `map`

binary search tree. Typically, an AVL tree is not used; instead, top-down red-black trees, which are discussed in Section 12.2, are often used.

An important issue in implementing `set` and `map` is providing support for the iterator classes. Of course, internally, the iterator maintains a pointer to the "current" node in the iteration. The hard part is efficiently advancing to the next node. There are several possible solutions, some of which are listed here:

1. When the iterator is constructed, have each iterator store as its data an array containing the `set` items. This doesn't work: It makes it impossible to efficiently implement any of the routines that return an iterator after modifying the `set`, such as some of the versions of `erase` and `insert`.
2. Have the iterator maintain a stack storing nodes on the path to the current node. With this information, one can deduce the next node in the iteration, which is either the node in the current node's right subtree that contains the minimum item or the nearest ancestor that contains the current node in its left subtree. This makes the iterator somewhat large and makes the iterator code clumsy.
3. Have each node in the search tree store its parent in addition to the children. The iterator is not as large, but there is now extra memory required in each node, and the code to iterate is still clumsy.
4. Have each node maintain extra links: one to the next smaller, and one to the next larger node. This takes space, but the iteration is very simple to do, and it is easy to maintain these links.
5. Maintain the extra links only for nodes that have `nullptr` left or right links by using extra Boolean variables to allow the routines to tell if a left link is being used as a standard binary search tree left link or a link to the next smaller node, and similarly for the right link (Exercise 4.49). This idea is called a **threaded tree** and is used in many of the STL implementations.

4.8.4 An Example That Uses Several Maps

Many words are similar to other words. For instance, by changing the first letter, the word `wine` can become `dine`, `fine`, `line`, `mine`, `nine`, `pine`, or `vine`. By changing the third letter, `wine` can become `wide`, `wife`, `wipe`, or `wire`, among others. By changing the fourth letter, `wine` can become `wind`, `wing`, `wink`, or `wins`, among others. This gives 15 different words that can be obtained by changing only one letter in `wine`. In fact, there are over 20 different words, some more obscure. We would like to write a program to find all words that can be changed into at least 15 other words by a single one-character substitution. We assume that we have a dictionary consisting of approximately 89,000 different words of varying lengths. Most words are between 6 and 11 characters. The distribution includes 8,205 six-letter words, 11,989 seven-letter words, 13,672 eight-letter words, 13,014 nine-letter words, 11,297 ten-letter words, and 8,617 eleven-letter words. (In reality, the most changeable words are three-, four-, and five-letter words, but the longer words are the time-consuming ones to check.)

The most straightforward strategy is to use a `map` in which the keys are words and the values are vectors containing the words that can be changed from the key with a

```
 1  void printHighChangeables( const map<string,vector<string>> & adjacentWords,
 2                             int minWords = 15 )
 3  {
 4      for( auto & entry : adjacentWords )
 5      {
 6          const vector<string> & words = entry.second;
 7
 8          if( words.size( ) >= minWords )
 9          {
10              cout << entry.first << " (" << words.size( ) << "):";
11              for( auto & str : words )
12                  cout << " " << str;
13              cout << endl;
14          }
15      }
16  }
```

Figure 4.69 Given a `map` containing words as keys and a `vector` of words that differ in only one character as values, output words that have `minWords` or more words obtainable by a one-character substitution

one-character substitution. The routine in Figure 4.69 shows how the `map` that is eventually produced (we have yet to write code for that part) can be used to print the required answers. The code uses a range `for` loop to step through the `map` and views entries that are pairs consisting of a word and a vector of words. The constant references at lines 4 and 6 are used to replace complicated expressions and avoid making unneeded copies.

The main issue is how to construct the `map` from an array that contains the 89,000 words. The routine in Figure 4.70 is a straightforward function to test if two words are identical except for a one-character substitution. We can use the routine to provide the simplest algorithm for the `map` construction, which is a brute-force test of all pairs of words. This algorithm is shown in Figure 4.71.

If we find a pair of words that differ in only one character, we can update the `map` at lines 12 and 13. The idiom we are using at line 12 is that `adjWords[str]` represents the `vector` of words that are identical to `str`, except for one character. If we have previously seen `str`, then it is in the `map`, and we need only add the new word to the `vector` in the `map`, and we do this by calling `push_back`. If we have never seen `str` before, then the act of using `operator[]` places it in the `map`, with a `vector` of size 0, and returns this `vector`, so the `push_back` updates the `vector` to be size 1. All in all, a super-slick idiom for maintaining a `map` in which the value is a collection.

The problem with this algorithm is that it is slow and takes 97 seconds on our computer. An obvious improvement is to avoid comparing words of different lengths. We can do this by grouping words by their length, and then running the previous algorithm on each of the separate groups.

To do this, we can use a second `map`! Here the key is an integer representing a word length, and the value is a collection of all the words of that length. We can use a `vector` to

```
// Returns true if word1 and word2 are the same length
// and differ in only one character.
bool oneCharOff( const string & word1, const string & word2 )
{
    if( word1.length( ) != word2.length( ) )
        return false;

    int diffs = 0;

    for( int i = 0; i < word1.length( ); ++i )
        if( word1[ i ] != word2[ i ] )
            if( ++diffs > 1 )
                return false;

    return diffs == 1;
}
```

Figure 4.70 Routine to check if two words differ in only one character

```
// Computes a map in which the keys are words and values are vectors of words
// that differ in only one character from the corresponding key.
// Uses a quadratic algorithm.
map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
{
    map<string,vector<string>> adjWords;

    for( int i = 0; i < words.size( ); ++i )
        for( int j = i + 1; j < words.size( ); ++j )
            if( oneCharOff( words[ i ], words[ j ] ) )
            {
                adjWords[ words[ i ] ].push_back( words[ j ] );
                adjWords[ words[ j ] ].push_back( words[ i ] );
            }

    return adjWords;
}
```

Figure 4.71 Function to compute a `map` containing words as keys and a `vector` of words that differ in only one character as values. This version runs in 1.5 minutes on an 89,000-word dictionary.

store each collection, and the same idiom applies. The code is shown in Figure 4.72. Line 8 shows the declaration for the second `map`, lines 11 and 12 populate the `map`, and then an extra loop is used to iterate over each group of words. Compared to the first algorithm, the second algorithm is only marginally more difficult to code and runs in 18 seconds, or about six times as fast.

```cpp
// Computes a map in which the keys are words and values are vectors of words
// that differ in only one character from the corresponding key.
// Uses a quadratic algorithm, but speeds things up a little by
// maintaining an additional map that groups words by their length.
map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
{
    map<string,vector<string>> adjWords;
    map<int,vector<string>> wordsByLength;

      // Group the words by their length
    for( auto & thisWord : words )
        wordsByLength[ thisWord.length( ) ].push_back( thisWord );

      // Work on each group separately
    for( auto & entry : wordsByLength )
    {
        const vector<string> & groupsWords = entry.second;

        for( int i = 0; i < groupsWords.size( ); ++i )
            for( int j = i + 1; j < groupsWords.size( ); ++j )
                if( oneCharOff( groupsWords[ i ], groupsWords[ j ] ) )
                {
                    adjWords[ groupsWords[ i ] ].push_back( groupsWords[ j ] );
                    adjWords[ groupsWords[ j ] ].push_back( groupsWords[ i ] );
                }
    }

    return adjWords;
}
```

Figure 4.72 Function to compute a `map` containing words as keys and a `vector` of words that differ in only one character as values. It splits words into groups by word length. This version runs in 18 seconds on an 89,000-word dictionary.

Our third algorithm is more complex and uses additional `map`s! As before, we group the words by word length, and then work on each group separately. To see how this algorithm works, suppose we are working on words of length 4. First we want to find word pairs, such as `wine` and `nine`, that are identical except for the first letter. One way to do this is as follows: For each word of length 4, remove the first character, leaving a three-character word representative. Form a `map` in which the key is the representative, and the value is a `vector` of all words that have that representative. For instance, in considering the first character of the four-letter word group, representative `"ine"` corresponds to `"dine"`, `"fine"`, `"wine"`, `"nine"`, `"mine"`, `"vine"`, `"pine"`, `"line"`. Representative `"oot"` corresponds to `"boot"`, `"foot"`, `"hoot"`, `"loot"`, `"soot"`, `"zoot"`. Each individual `vector` that is a value in this latest `map` forms a clique of words in which any word can be changed to any other word by a one-character substitution, so after this latest `map` is constructed, it is easy to traverse it and add entries to the original `map` that is being computed. We would then proceed to the second character of the four-letter word group, with a new `map`, and then the third character, and finally the fourth character.

The general outline is

```
for each group g, containing words of length len
    for each position p (ranging from 0 to len-1)
    {
        Make an empty map<string,vector<string>> repsToWords
        for each word w
        {
            Obtain w's representative by removing position p
            Update repsToWords
        }
        Use cliques in repsToWords to update adjWords map
    }
```

Figure 4.73 contains an implementation of this algorithm. The running time improves to two seconds. It is interesting to note that although the use of the additional `maps` makes the algorithm faster, and the syntax is relatively clean, the code makes no use of the fact that the keys of the `map` are maintained in sorted order.

```cpp
// Computes a map in which the keys are words and values are vectors of words
// that differ in only one character from the corresponding key.
// Uses an efficient algorithm that is O(N log N) with a map
map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
{
    map<string,vector<string>> adjWords;
    map<int,vector<string>> wordsByLength;

      // Group the words by their length
    for( auto & str : words )
        wordsByLength[ str.length( ) ].push_back( str );

      // Work on each group separately
    for( auto & entry : wordsByLength )
    {
        const vector<string> & groupsWords = entry.second;
        int groupNum = entry.first;

        // Work on each position in each group
        for( int i = 0; i < groupNum; ++i )
```

Figure 4.73 Function to compute a map containing words as keys and a vector of words that differ in only one character as values. This version runs in 2 seconds on an 89,000-word dictionary.

```
        {
            // Remove one character in specified position, computing representative.
            // Words with same representatives are adjacent; so populate a map ...
            map<string,vector<string>> repToWord;

            for( auto & str : groupsWords )
            {
                string rep = str;
                rep.erase( i, 1 );
                repToWord[ rep ].push_back( str );
            }

            // and then look for map values with more than one string
            for( auto & entry : repToWord )
            {
                const vector<string> & clique = entry.second;
                if( clique.size( ) >= 2 )
                    for( int p = 0; p < clique.size( ); ++p )
                        for( int q = p + 1; q < clique.size( ); ++q )
                        {
                            adjWords[ clique[ p ] ].push_back( clique[ q ] );
                            adjWords[ clique[ q ] ].push_back( clique[ p ] );
                        }
            }
        }
    }
    return adjWords;
}
```

Figure 4.73 *(continued)*

As such, it is possible that a data structure that supports the `map` operations but does not guarantee sorted order can perform better, since it is being asked to do less. Chapter 5 explores this possibility and discusses the ideas behind the alternative map implementation that C++11 adds to the Standard Library, known as an `unordered_map`. An unordered map reduces the running time of the implementation from 2 sec to 1.5 sec.

Summary

We have seen uses of trees in operating systems, compiler design, and searching. Expression trees are a small example of a more general structure known as a **parse tree**, which is a central data structure in compiler design. Parse trees are not binary, but are relatively simple extensions of expression trees (although the algorithms to build them are not quite so simple).

Search trees are of great importance in algorithm design. They support almost all the useful operations, and the logarithmic average cost is very small. Nonrecursive implementations of search trees are somewhat faster, but the recursive versions are sleeker, more elegant, and easier to understand and debug. The problem with search trees is that their performance depends heavily on the input being random. If this is not the case, the running time increases significantly, to the point where search trees become expensive linked lists.

We saw several ways to deal with this problem. AVL trees work by insisting that all nodes' left and right subtrees differ in heights by at most one. This ensures that the tree cannot get too deep. The operations that do not change the tree, as insertion does, can all use the standard binary search tree code. Operations that change the tree must restore the tree. This can be somewhat complicated, especially in the case of deletion. We showed how to restore the tree after insertions in $O(\log N)$ time.

We also examined the splay tree. Nodes in splay trees can get arbitrarily deep, but after every access the tree is adjusted in a somewhat mysterious manner. The net effect is that any sequence of M operations takes $O(M \log N)$ time, which is the same as a balanced tree would take.

B-trees are balanced M-way (as opposed to 2-way or binary) trees, which are well suited for disks; a special case is the 2–3 tree ($M = 3$), which is another way to implement balanced search trees.

In practice, the running time of all the balanced-tree schemes, while slightly faster for searching, is worse (by a constant factor) for insertions and deletions than the simple binary search tree, but this is generally acceptable in view of the protection being given against easily obtained worst-case input. Chapter 12 discusses some additional search tree data structures and provides detailed implementations.

A final note: By inserting elements into a search tree and then performing an inorder traversal, we obtain the elements in sorted order. This gives an $O(N \log N)$ algorithm to sort, which is a worst-case bound if any sophisticated search tree is used. We shall see better ways in Chapter 7, but none that have a lower time bound.

Exercises

4.1 For the tree in Figure 4.74:
a. Which node is the root?
b. Which nodes are leaves?

4.2 For each node in the tree of Figure 4.74:
a. Name the parent node.
b. List the children.
c. List the siblings.
d. Compute the depth.
e. Compute the height.

4.3 What is the depth of the tree in Figure 4.74?

4.4 Show that in a binary tree of N nodes, there are $N + 1$ `nullptr` links representing children.

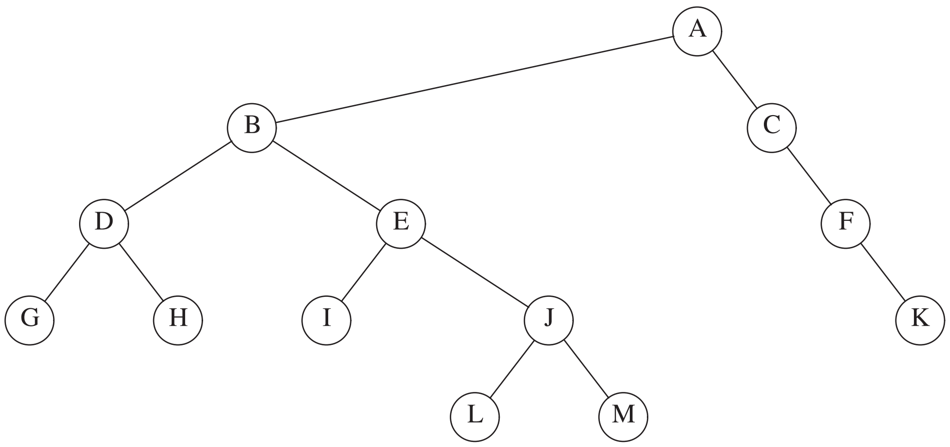


Figure 4.74 Tree for Exercises 4.1 to 4.3

4.5 Show that the maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$.

4.6 A *full node* is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a nonempty binary tree.

4.7 Suppose a binary tree has leaves $l_1, l_2, \ldots, l_M$ at depths $d_1, d_2, \ldots, d_M$, respectively. Prove that $\sum_{i=1}^{M} 2^{-d_i} \leq 1$ and determine when the equality is true.

4.8 Give the prefix, infix, and postfix expressions corresponding to the tree in Figure 4.75.

4.9 a. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.

b. Show the result of deleting the root.

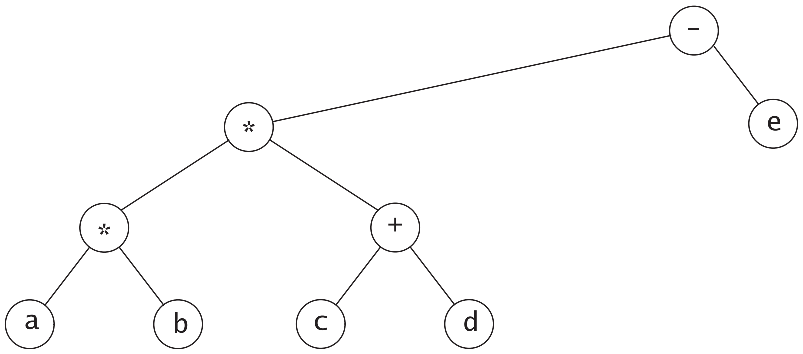


Figure 4.75 Tree for Exercise 4.8

4.10 Let $f(N)$ be the average number of full nodes in an N-node binary search tree.

a. Determine the values of $f(0)$ and $f(1)$.

b. Show that for $N > 1$

$$f(N) = \frac{N-2}{N} + \frac{1}{N} \sum_{i=0}^{N-1} (f(i) + f(N-i-1))$$

c. Show (by induction) that $f(N) = (N-2)/3$ is a solution to the equation in part (b), with the initial conditions in part (a).

d. Use the results of Exercise 4.6 to determine the average number of leaves in an N-node binary search tree.

4.11 Write an implementation of the `set` class, with associated iterators using a binary search tree. Add to each node a link to the parent node.

4.12 Write an implementation of the `map` class by storing a data member of type `set<Pair<KeyType,ValueType>>`.

4.13 Write an implementation of the `set` class, with associated iterators using a binary search tree. Add to each node a link to the next smallest and next largest node. To make your code simpler, add a header and tail node which are not part of the binary search tree, but help make the linked list part of the code simpler.

4.14 Suppose you want to perform an experiment to verify the problems that can be caused by random `insert/remove` pairs. Here is a strategy that is not perfectly random, but close enough. You build a tree with N elements by inserting N elements chosen at random from the range 1 to $M = \alpha N$. You then perform N^2 pairs of insertions followed by deletions. Assume the existence of a routine, `randomInteger(a,b)`, which returns a uniform random integer between `a` and `b` inclusive.

a. Explain how to generate a random integer between 1 and M that is not already in the tree (so a random insertion can be performed). In terms of N and α, what is the running time of this operation?

b. Explain how to generate a random integer between 1 and M that is already in the tree (so a random deletion can be performed). What is the running time of this operation?

c. What is a good choice of α? Why?

4.15 Write a program to evaluate empirically the following strategies for removing nodes with two children:

a. Replace with the largest node, X, in T_L and recursively remove X.

b. Alternately replace with the largest node in T_L and the smallest node in T_R, and recursively remove the appropriate node.

c. Replace with either the largest node in T_L or the smallest node in T_R (recursively removing the appropriate node), making the choice randomly.

Which strategy seems to give the most balance? Which takes the least CPU time to process the entire sequence?

4.16 Redo the binary search tree class to implement lazy deletion. Note carefully that this affects all of the routines. Especially challenging are `findMin` and `findMax`, which must now be done recursively.

** **4.17** Prove that the depth of a random binary search tree (depth of the deepest node) is $O(\log N)$, on average.

4.18 * a. Give a precise expression for the minimum number of nodes in an AVL tree of height h.

b. What is the minimum number of nodes in an AVL tree of height 15?

4.19 Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.

* **4.20** Keys $1, 2, \ldots, 2^k - 1$ are inserted in order into an initially empty AVL tree. Prove that the resulting tree is perfectly balanced.

4.21 Write the remaining procedures to implement AVL single and double rotations.

4.22 Design a linear-time algorithm that verifies that the height information in an AVL tree is correctly maintained and that the balance property is in order.

4.23 Write a nonrecursive function to insert into an AVL tree.

4.24 Show that the deletion algorithm in Figure 4.47 is correct

4.25 a. How many bits are required per node to store the height of a node in an N-node AVL tree?

b. What is the smallest AVL tree that overflows an 8-bit height counter?

4.26 Write the functions to perform the double rotation without the inefficiency of doing two single rotations.

4.27 Show the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in Figure 4.76.

4.28 Show the result of deleting the element with key 6 in the resulting splay tree for the previous exercise.

4.29 a. Show that if all nodes in a splay tree are accessed in sequential order, the resulting tree consists of a chain of left children.

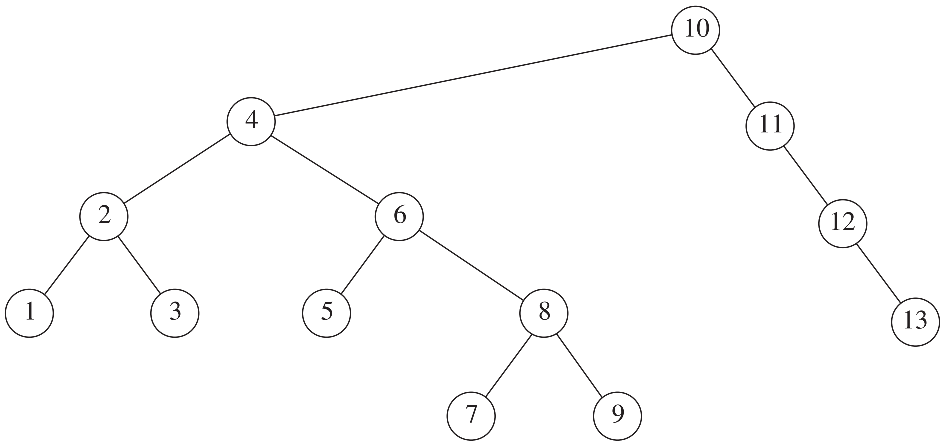


Figure 4.76 Tree for Exercise 4.27

** b. Show that if all nodes in a splay tree are accessed in sequential order, then the total access time is $O(N)$, regardless of the initial tree.

4.30 Write a program to perform random operations on splay trees. Count the total number of rotations performed over the sequence. How does the running time compare to AVL trees and unbalanced binary search trees?

4.31 Write efficient functions that take only a pointer to the root of a binary tree, T, and compute

a. the number of nodes in T
b. the number of leaves in T
c. the number of full nodes in T

What is the running time of your routines?

4.32 Design a recursive linear-time algorithm that tests whether a binary tree satisfies the search tree order property at every node.

4.33 Write a recursive function that takes a pointer to the root node of a tree T and returns a pointer to the root node of the tree that results from removing all leaves from T.

4.34 Write a function to generate an N-node random binary search tree with distinct keys 1 through N. What is the running time of your routine?

4.35 Write a function to generate the AVL tree of height h with fewest nodes. What is the running time of your function?

4.36 Write a function to generate a perfectly balanced binary search tree of height h with keys 1 through $2^{h+1} - 1$. What is the running time of your function?

4.37 Write a function that takes as input a binary search tree, T, and two keys, k_1 and k_2, which are ordered so that $k_1 \leq k_2$, and prints all elements X in the tree such that $k_1 \leq Key(X) \leq k_2$. Do not assume any information about the type of keys except that they can be ordered (consistently). Your program should run in $O(K + \log N)$ average time, where K is the number of keys printed. Bound the running time of your algorithm.

4.38 The larger binary trees in this chapter were generated automatically by a program. This was done by assigning an (x, y) coordinate to each tree node, drawing a circle around each coordinate (this is hard to see in some pictures), and connecting each node to its parent. Assume you have a binary search tree stored in memory (perhaps generated by one of the routines above) and that each node has two extra fields to store the coordinates.

a. The x coordinate can be computed by assigning the inorder traversal number. Write a routine to do this for each node in the tree.
b. The y coordinate can be computed by using the negative of the depth of the node. Write a routine to do this for each node in the tree.
c. In terms of some imaginary unit, what will the dimensions of the picture be? How can you adjust the units so that the tree is always roughly two-thirds as high as it is wide?
d. Prove that using this system no lines cross, and that for any node, X, all elements in X's left subtree appear to the left of X and all elements in X's right subtree appear to the right of X.

4.39 Write a general-purpose tree-drawing program that will convert a tree into the following graph-assembler instructions:

a. *Circle*(X, Y)

b. *DrawLine*(i, j)

The first instruction draws a circle at (X, Y), and the second instruction connects the ith circle to the jth circle (circles are numbered in the order drawn). You should either make this a program and define some sort of input language or make this a function that can be called from any program. What is the running time of your routine?

4.40 Write a routine to list out the nodes of a binary tree in *level-order.* List the root, then nodes at depth 1, followed by nodes at depth 2, and so on. You must do this in linear time. Prove your time bound.

4.41 ⋆a. Write a routine to perform insertion into a B-tree.

⋆b. Write a routine to perform deletion from a B-tree. When an item is deleted, is it necessary to update information in the internal nodes?

⋆c. Modify your insertion routine so that if an attempt is made to add into a node that already has M entries, a search is performed for a sibling with less than M children before the node is split.

4.42 A *B^*-tree* of order M is a B-tree in which each interior node has between $2M/3$ and M children. Describe a method to perform insertion into a B^*-tree.

4.43 Show how the tree in Figure 4.77 is represented using a child/sibling link implementation.

4.44 Write a procedure to traverse a tree stored with child/sibling links.

4.45 Two binary trees are similar if they are both empty or both nonempty and have similar left and right subtrees. Write a function to decide whether two binary trees are similar. What is the running time of your function?

4.46 Two trees, T_1 and T_2, are *isomorphic* if T_1 can be transformed into T_2 by swapping left and right children of (some of the) nodes in T_1. For instance, the two trees in Figure 4.78 are isomorphic because they are the same if the children of A, B, and G, but not the other nodes, are swapped.

a. Give a polynomial time algorithm to decide if two trees are isomorphic.

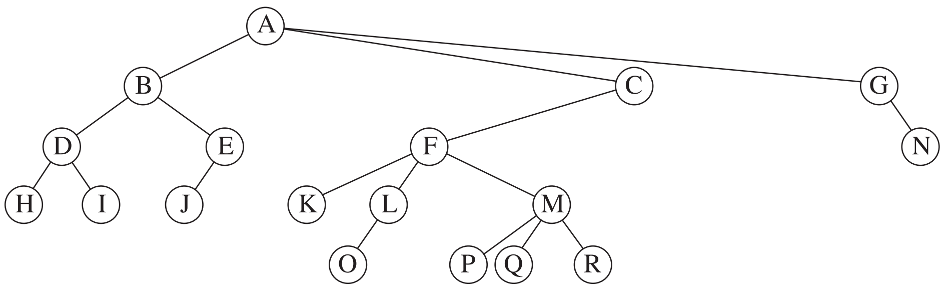


Figure 4.77 Tree for Exercise 4.43

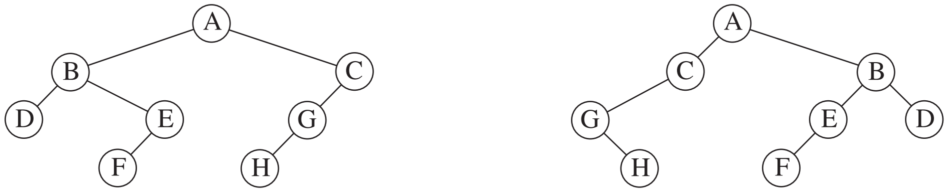


Figure 4.78 Two isomorphic trees

* b. What is the running time of your program (there is a linear solution)?

4.47 * a. Show that via AVL single rotations, any binary search tree T_1 can be transformed into another search tree T_2 (with the same items).

* b. Give an algorithm to perform this transformation using $O(N \log N)$ rotations on average.

** c. Show that this transformation can be done with $O(N)$ rotations, worst-case.

4.48 Suppose we want to add the operation `findKth` to our repertoire. The operation `findKth(k)` returns the `k`th smallest item in the tree. Assume all items are distinct. Explain how to modify the binary search tree to support this operation in $O(\log N)$ average time, without sacrificing the time bounds of any other operation.

4.49 Since a binary search tree with N nodes has $N + 1$ `nullptr` pointers, half the space allocated in a binary search tree for pointer information is wasted. Suppose that if a node has a `nullptr` left child, we make its left child link to its inorder predecessor, and if a node has a `nullptr` right child, we make its right child link to its inorder successor. This is known as a *threaded tree* and the extra links are called *threads*.

a. How can we distinguish threads from real children pointers?

b. Write routines to perform insertion and deletion into a tree threaded in the manner described above.

c. What is the advantage of using threaded trees?

4.50 Write a program that reads a C++ source code file and outputs a list of all identifiers (that is, variable names, but not keywords, that are not found in comments or string constants) in alphabetical order. Each identifier should be output with a list of line numbers on which it occurs.

4.51 Generate an index for a book. The input file consists of a set of index entries. Each line consists of the string `IX:`, followed by an index entry name enclosed in braces, followed by a page number that is enclosed in braces. Each `!` in an index entry name represents a sublevel. A `|(` represents the start of a range, and a `|)` represents the end of the range. Occasionally, this range will be the same page. In that case, output only a single page number. Otherwise, do not collapse or expand ranges on your own. As an example, Figure 4.79 shows sample input and Figure 4.80 shows the corresponding output.

```
IX: {Series|(}            {2}
IX: {Series!geometric|(}  {4}
IX: {Euler's constant}    {4}
IX: {Series!geometric|)}  {4}
IX: {Series!arithmetic|(} {4}
IX: {Series!arithmetic|)} {5}
IX: {Series!harmonic|(}   {5}
IX: {Euler's constant}    {5}
IX: {Series!harmonic|)}   {5}
IX: {Series|)}            {5}
```

Figure 4.79 Sample input for Exercise 4.51

```
Euler's constant: 4, 5
Series: 2-5
   arithmetic: 4-5
   geometric: 4
   harmonic: 5
```

Figure 4.80 Sample output for Exercise 4.51

References

More information on binary search trees, and in particular the mathematical properties of trees, can be found in the two books by Knuth, [22] and [23].

Several papers deal with the lack of balance caused by biased deletion algorithms in binary search trees. Hibbard's paper [19] proposed the original deletion algorithm and established that one deletion preserves the randomness of the trees. A complete analysis has been performed only for trees with three nodes [20] and four nodes [5]. Eppinger's paper [14] provided early empirical evidence of nonrandomness, and the papers by Culberson and Munro [10], [11] provided some analytical evidence (but not a complete proof for the general case of intermixed insertions and deletions).

Adelson-Velskii and Landis [1] proposed AVL trees. Recently it was shown that for AVL trees, if rebalancing is performed only on insertions, and not on deletions, under certain circumstances the resulting structure still maintains a depth of $O(\log M)$ where M is the number of insertions [28]. Simulation results for AVL trees, and variants in which the height imbalance is allowed to be at most k for various values of k, are presented in [21]. Analysis of the average search cost in AVL trees is incomplete, but some results are contained in [24].

[3] and [8] considered self-adjusting trees like the type in Section 4.5.1. Splay trees are described in [29].

B-trees first appeared in [6]. The implementation described in the original paper allows data to be stored in internal nodes as well as leaves. The data structure we have described is sometimes known as a B$^+$-tree. A survey of the different types of B-trees is presented in [9]. Empirical results of the various schemes are reported in [17]. Analysis of 2–3 trees and B-trees can be found in [4], [13], and [32].

Exercise 4.17 is deceptively difficult. A solution can be found in [15]. Exercise 4.29 is from [32]. Information on B*-trees, described in Exercise 4.42, can be found in [12]. Exercise 4.46 is from [2]. A solution to Exercise 4.47 using $2N - 6$ rotations is given in [30]. Using threads, à la Exercise 4.49, was first proposed in [27]. k-d trees, which handle multidimensional data, were first proposed in [7] and are discussed in Chapter 12.

Other popular balanced search trees are red-black trees [18] and weight-balanced trees [26]. More balanced-tree schemes can be found in the books [16] and [25].

1. G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet. Mat. Doklady,* 3 (1962), 1259–1263.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
3. B. Allen and J. I. Munro, "Self Organizing Search Trees," *Journal of the ACM,* 25 (1978), 526–535.
4. R. A. Baeza-Yates, "Expected Behaviour of B^+-trees under Random Insertions," *Acta Informatica,* 26 (1989), 439–471.
5. R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't: A Continuation," *BIT,* 29 (1989), 88–113.
6. R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica,* 1 (1972), 173–189.
7. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM,* 18 (1975), 509–517.
8. J. R. Bitner, "Heuristics that Dynamically Organize Data Structures," *SIAM Journal on Computing,* 8 (1979), 82–110.
9. D. Comer, "The Ubiquitous B-tree," *Computing Surveys,* 11 (1979), 121–137.
10. J. Culberson and J. I. Munro, "Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations," *Computer Journal,* 32 (1989), 68–75.
11. J. Culberson and J. I. Munro, "Analysis of the Standard Deletion Algorithms in Exact Fit Domain Binary Search Trees," *Algorithmica,* 5 (1990), 295–311.
12. K. Culik, T. Ottman, and D. Wood, "Dense Multiway Trees," *ACM Transactions on Database Systems,* 6 (1981), 486–512.
13. B. Eisenbath, N. Ziviana, G. H. Gonnet, K. Melhorn, and D. Wood, "The Theory of Fringe Analysis and Its Application to 2–3 Trees and B-trees," *Information and Control,* 55 (1982), 125–174.
14. J. L. Eppinger, "An Empirical Study of Insertion and Deletion in Binary Search Trees," *Communications of the ACM,* 26 (1983), 663–669.
15. P. Flajolet and A. Odlyzko, "The Average Height of Binary Trees and Other Simple Trees," *Journal of Computer and System Sciences,* 25 (1982), 171–213.
16. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* 2d ed., Addison-Wesley, Reading, Mass., 1991.
17. E. Gudes and S. Tsur, "Experiments with B-tree Reorganization," *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200–206.

18. L. J. Guibas and R. Sedgewick, “A Dichromatic Framework for Balanced Trees,” *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8–21.
19. T. H. Hibbard, “Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting,” *Journal of the ACM,* 9 (1962), 13–28.
20. A. T. Jonassen and D. E. Knuth, “A Trivial Algorithm Whose Analysis Isn’t,” *Journal of Computer and System Sciences,* 16 (1978), 301–322.
21. P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler, “Performance of Height Balanced Trees,” *Communications of the ACM,* 19 (1976), 23–28.
22. D. E. Knuth, *The Art of Computer Programming: Vol. 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
23. D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
24. K. Melhorn, “A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions,” *SIAM Journal of Computing,* 11 (1982), 748–760.
25. K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching,* Springer-Verlag, Berlin, 1984.
26. J. Nievergelt and E. M. Reingold, “Binary Search Trees of Bounded Balance,” *SIAM Journal on Computing,* 2 (1973), 33–43.
27. A. J. Perlis and C. Thornton, “Symbol Manipulation in Threaded Lists,” *Communications of the ACM,* 3 (1960), 195–204.
28. S. Sen and R. E. Tarjan, “Deletion Without Rebalancing in Balanced Binary Trees,” *Proceedings of the Twentieth Symposium on Discrete Algorithms* (2010), 1490–1499.
29. D. D. Sleator and R. E. Tarjan, “Self-adjusting Binary Search Trees,” *Journal of the ACM,* 32 (1985), 652–686.
30. D. D. Sleator, R. E. Tarjan, and W. P. Thurston, “Rotation Distance, Triangulations, and Hyperbolic Geometry,” *Journal of the AMS* (1988), 647–682.
31. R. E. Tarjan, “Sequential Access in Splay Trees Takes Linear Time,” *Combinatorica,* 5 (1985), 367–378.
32. A. C. Yao, “On Random 2–3 Trees,” *Acta Informatica,* 9 (1978), 159–170.