5

Binary Trees

The list representations of Chapter 4 have a fundamental limitation: Either search or insert can be made efficient, but not both at the same time. Tree structures permit both efficient access and update to large collections of data. Binary trees in particular are widely used and relatively easy to implement. But binary trees are useful for many things besides searching. Just a few examples of applications that trees can speed up include prioritizing jobs, describing mathematical expressions and the syntactic elements of computer programs, or organizing the information needed to drive data compression algorithms.

This chapter begins by presenting definitions and some key properties of binary trees. Section 5.2 discusses how to process all nodes of the binary tree in an organized manner. Section 5.3 presents various methods for implementing binary trees and their nodes. Sections 5.4 through 5.6 present three examples of binary trees used in specific applications: the Binary Search Tree (BST) for implementing dictionaries, heaps for implementing priority queues, and Huffman coding trees for text compression. The BST, heap, and Huffman coding tree each have distinctive structural features that affect their implementation and use.

5.1 Definitions and Properties

A **binary tree** is made up of a finite set of elements called **nodes**. This set either is empty or consists of a node called the **root** together with two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root. (Disjoint means that they have no nodes in common.) The roots of these subtrees are **children** of the root. There is an **edge** from a node to each of its children, and a node is said to be the **parent** of its children.

If n_1, n_2, ..., n_k is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a **path** from n_1 to n_k. The **length** of the path is $k - 1$. If there is a path from node *R* to node *M*, then *R* is an **ancestor** of *M*, and *M* is a **descendant** of *R*. Thus, all nodes in the tree are descendants of the

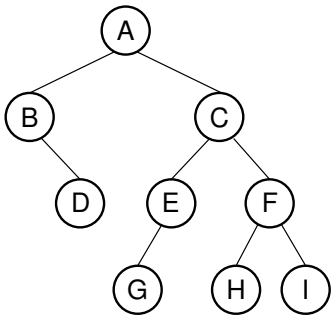


Figure 5.1 A binary tree. Node *A* is the root. Nodes *B* and *C* are *A*'s children. Nodes *B* and *D* together form a subtree. Node *B* has two children: Its left child is the empty tree and its right child is *D*. Nodes *A*, *C*, and *E* are ancestors of *G*. Nodes *D*, *E*, and *F* make up level 2 of the tree; node *A* is at level 0. The edges from *A* to *C* to *E* to *G* form a path of length 3. Nodes *D*, *G*, *H*, and *I* are leaves. Nodes *A*, *B*, *C*, *E*, and *F* are internal nodes. The depth of *I* is 3. The height of this tree is 4.

root of the tree, while the root is the ancestor of all nodes. The **depth** of a node *M* in the tree is the length of the path from the root of the tree to *M*. The **height** of a tree is one more than the depth of the deepest node in the tree. All nodes of depth d are at **level** d in the tree. The root is the only node at level 0, and its depth is 0. A **leaf** node is any node that has two empty children. An **internal** node is any node that has at least one non-empty child.

Figure 5.1 illustrates the various terms used to identify parts of a binary tree. Figure 5.2 illustrates an important point regarding the structure of binary trees. Because *all* binary tree nodes have two children (one or both of which might be empty), the two binary trees of Figure 5.2 are *not* the same.

Two restricted forms of binary tree are sufficiently important to warrant special names. Each node in a **full** binary tree is either (1) an internal node with exactly two non-empty children or (2) a leaf. A **complete** binary tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height d, all levels except possibly level $d-1$ are completely full. The bottom level has its nodes filled in from the left side.

Figure 5.3 illustrates the differences between full and complete binary trees.[1] There is no particular relationship between these two tree shapes; that is, the tree of Figure 5.3(a) is full but not complete while the tree of Figure 5.3(b) is complete but

[1] While these definitions for full and complete binary tree are the ones most commonly used, they are not universal. Because the common meaning of the words "full" and "complete" are quite similar, there is little that you can do to distinguish between them other than to memorize the definitions. Here is a memory aid that you might find useful: "Complete" is a wider word than "full," and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible.

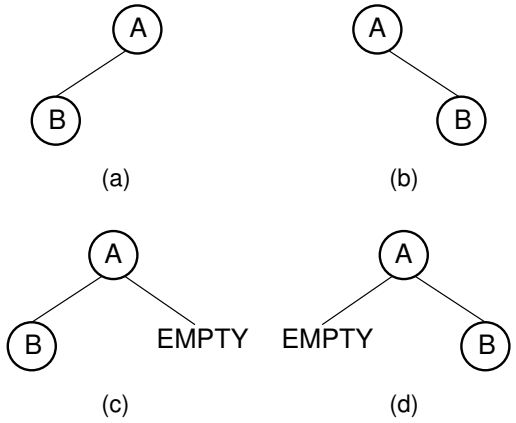


Figure 5.2 Two different binary trees. (a) A binary tree whose root has a non-empty left child. (b) A binary tree whose root has a non-empty right child. (c) The binary tree of (a) with the missing right child made explicit. (d) The binary tree of (b) with the missing left child made explicit.

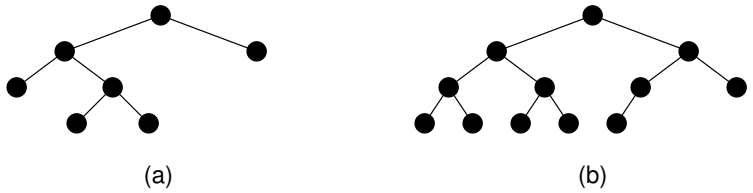


Figure 5.3 Examples of full and complete binary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

not full. The heap data structure (Section 5.5) is an example of a complete binary tree. The Huffman coding tree (Section 5.6) is an example of a full binary tree.

5.1.1 The Full Binary Tree Theorem

Some binary tree implementations store data only at the leaf nodes, using the internal nodes to provide structure to the tree. More generally, binary tree implementations might require some amount of space for internal nodes, and a different amount for leaf nodes. Thus, to analyze the space required by such implementations, it is useful to know the minimum and maximum fraction of the nodes that are leaves in a tree containing n internal nodes.

Unfortunately, this fraction is not fixed. A binary tree of n internal nodes might have only one leaf. This occurs when the internal nodes are arranged in a chain ending in a single leaf as shown in Figure 5.4. In this case, the number of leaves is low because each internal node has only one non-empty child. To find an upper bound on the number of leaves for a tree of n internal nodes, first note that the upper

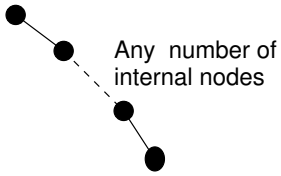


Figure 5.4 A tree containing many internal nodes and a single leaf.

bound will occur when each internal node has two non-empty children, that is, when the tree is full. However, this observation does not tell what shape of tree will yield the highest percentage of non-empty leaves. It turns out not to matter, because all full binary trees with n internal nodes have the same number of leaves. This fact allows us to compute the space requirements for a full binary tree implementation whose leaves require a different amount of space from its internal nodes.

Theorem 5.1 Full Binary Tree Theorem: *The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.*

Proof: The proof is by mathematical induction on n, the number of internal nodes. This is an example of an induction proof where we reduce from an arbitrary instance of size n to an instance of size $n-1$ that meets the induction hypothesis.

- **Base Cases**: The non-empty tree with zero internal nodes has one leaf node. A full binary tree with one internal node has two leaf nodes. Thus, the base cases for $n = 0$ and $n = 1$ conform to the theorem.
- **Induction Hypothesis**: Assume that any full binary tree **T** containing $n-1$ internal nodes has n leaves.
- **Induction Step**: Given tree **T** with n internal nodes, select an internal node *I* whose children are both leaf nodes. Remove both of *I*'s children, making *I* a leaf node. Call the new tree **T**′. **T**′ has $n-1$ internal nodes. From the induction hypothesis, **T**′ has n leaves. Now, restore *I*'s two children. We once again have tree **T** with n internal nodes. How many leaves does **T** have? Because **T**′ has n leaves, adding the two children yields $n+2$. However, node *I* counted as one of the leaves in **T**′ and has now become an internal node. Thus, tree **T** has $n+1$ leaf nodes and n internal nodes.

By mathematical induction the theorem holds for all values of $n \geq 0$. □

When analyzing the space requirements for a binary tree implementation, it is useful to know how many empty subtrees a tree contains. A simple extension of the Full Binary Tree Theorem tells us exactly how many empty subtrees there are in *any* binary tree, whether full or not. Here are two approaches to proving the following theorem, and each suggests a useful way of thinking about binary trees.

Theorem 5.2 *The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.*

Proof 1: Take an arbitrary binary tree **T** and replace every empty subtree with a leaf node. Call the new tree **T′**. All nodes originally in **T** will be internal nodes in **T′** (because even the leaf nodes of **T** have children in **T′**). **T′** is a full binary tree, because every internal node of **T** now must have two children in **T′**, and each leaf node in **T** must have two children in **T′** (the leaves just added). The Full Binary Tree Theorem tells us that the number of leaves in a full binary tree is one more than the number of internal nodes. Thus, the number of new leaves that were added to create **T′** is one more than the number of nodes in **T**. Each leaf node in **T′** corresponds to an empty subtree in **T**. Thus, the number of empty subtrees in **T** is one more than the number of nodes in **T**. □

Proof 2: By definition, every node in binary tree **T** has two children, for a total of $2n$ children in a tree of n nodes. Every node except the root node has one parent, for a total of $n - 1$ nodes with parents. In other words, there are $n - 1$ non-empty children. Because the total number of children is $2n$, the remaining $n + 1$ children must be empty. □

5.1.2 A Binary Tree Node ADT

Just as a linked list is comprised of a collection of link objects, a tree is comprised of a collection of node objects. Figure 5.5 shows an ADT for binary tree nodes, called **`BinNode`**. This class will be used by some of the binary tree structures presented later. Class **`BinNode`** is a template with parameter **`E`**, which is the type for the data record stored in the node. Member functions are provided that set or return the element value, set or return a pointer to the left child, set or return a pointer to the right child, or indicate whether the node is a leaf.

5.2 Binary Tree Traversals

Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a **traversal**. Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship. For example, we might wish to make sure that we visit any given node *before* we visit its children. This is called a **preorder traversal**.

```
// Binary tree node abstract class
template <typename E> class BinNode {
public:
  virtual ~BinNode() {} // Base destructor

  // Return the node's value
  virtual E& element() = 0;

  // Set the node's value
  virtual void setElement(const E&) = 0;

  // Return the node's left child
  virtual BinNode* left() const = 0;

  // Set the node's left child
  virtual void setLeft(BinNode*) = 0;

  // Return the node's right child
  virtual BinNode* right() const = 0;

  // Set the node's right child
  virtual void setRight(BinNode*) = 0;

  // Return true if the node is a leaf, false otherwise
  virtual bool isLeaf() = 0;
};
```

Figure 5.5 A binary tree node ADT.

Example 5.1 The preorder enumeration for the tree of Figure 5.1 is

ABDCEGFHI.

The first node printed is the root. Then all nodes of the left subtree are printed (in preorder) before any node of the right subtree.

Alternatively, we might wish to visit each node only *after* we visit its children (and their subtrees). For example, this would be necessary if we wish to return all nodes in the tree to free store. We would like to delete the children of a node before deleting the node itself. But to do that requires that the children's children be deleted first, and so on. This is called a **postorder traversal**.

Example 5.2 The postorder enumeration for the tree of Figure 5.1 is

DBGEHIFCA.

An **inorder traversal** first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree). The

binary search tree of Section 5.4 makes use of this traversal to print all nodes in ascending order of value.

Example 5.3 The inorder enumeration for the tree of Figure 5.1 is

BDAGECHFI.

A traversal routine is naturally written as a recursive function. Its input parameter is a pointer to a node which we will call **root** because each node can be viewed as the root of a some subtree. The initial call to the traversal function passes in a pointer to the root node of the tree. The traversal function visits **root** and its children (if any) in the desired order. For example, a preorder traversal specifies that **root** be visited before its children. This can easily be implemented as follows.

```
template <typename E>
void preorder(BinNode<E>* root) {
  if (root == NULL) return; // Empty subtree, do nothing
  visit(root);              // Perform desired action
  preorder(root->left());
  preorder(root->right());
}
```

Function **preorder** first checks that the tree is not empty (if it is, then the traversal is done and **preorder** simply returns). Otherwise, **preorder** makes a call to **visit**, which processes the root node (i.e., prints the value or performs whatever computation as required by the application). Function **preorder** is then called recursively on the left subtree, which will visit all nodes in that subtree. Finally, **preorder** is called on the right subtree, visiting all nodes in the right subtree. Postorder and inorder traversals are similar. They simply change the order in which the node and its children are visited, as appropriate.

An important decision in the implementation of any recursive function on trees is when to check for an empty subtree. Function **preorder** first checks to see if the value for **root** is **NULL**. If not, it will recursively call itself on the left and right children of **root**. In other words, **preorder** makes no attempt to avoid calling itself on an empty child. Some programmers use an alternate design in which the left and right pointers of the current node are checked so that the recursive call is made only on non-empty children. Such a design typically looks as follows:

```
template <typename E>
void preorder2(BinNode<E>* root) {
  visit(root);  // Perform whatever action is desired
  if (root->left() != NULL) preorder2(root->left());
  if (root->right() != NULL) preorder2(root->right());
}
```

At first it might appear that **preorder2** is more efficient than **preorder**, because it makes only half as many recursive calls. (Why?) On the other hand, **preorder2** must access the left and right child pointers twice as often. The net result is little or no performance improvement.

In reality, the design of **preorder2** is inferior to that of **preorder** for two reasons. First, while it is not apparent in this simple example, for more complex traversals it can become awkward to place the check for the **NULL** pointer in the calling code. Even here we had to write two tests for **NULL**, rather than the one needed by **preorder**. The more important concern with **preorder2** is that it tends to be error prone. While **preorder2** insures that no recursive calls will be made on empty subtrees, it will fail if the initial call passes in a **NULL** pointer. This would occur if the original tree is empty. To avoid the bug, either **preorder2** needs an additional test for a **NULL** pointer at the beginning (making the subsequent tests redundant after all), or the caller of **preorder2** has a hidden obligation to pass in a non-empty tree, which is unreliable design. The net result is that many programmers forget to test for the possibility that the empty tree is being traversed. By using the first design, which explicitly supports processing of empty subtrees, the problem is avoided.

Another issue to consider when designing a traversal is how to define the visitor function that is to be executed on every node. One approach is simply to write a new version of the traversal for each such visitor function as needed. The disadvantage to this is that whatever function does the traversal must have access to the **BinNode** class. It is probably better design to permit only the tree class to have access to the **BinNode** class.

Another approach is for the tree class to supply a generic traversal function which takes the visitor either as a template parameter or as a function parameter. This is known as the **visitor design pattern**. A major constraint on this approach is that the **signature** for all visitor functions, that is, their return type and parameters, must be fixed in advance. Thus, the designer of the generic traversal function must be able to adequately judge what parameters and return type will likely be needed by potential visitor functions.

Handling information flow between parts of a program can be a significant design challenge, especially when dealing with recursive functions such as tree traversals. In general, we can run into trouble either with passing in the correct information needed by the function to do its work, or with returning information to the recursive function's caller. We will see many examples throughout the book that illustrate methods for passing information in and out of recursive functions as they traverse a tree structure. Here are a few simple examples.

First we consider the simple case where a computation requires that we communicate information back up the tree to the end user.

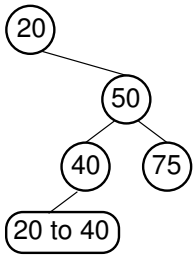


Figure 5.6 To be a binary search tree, the left child of the node with value 40 must have a value between 20 and 40.

Example 5.4 We wish to count the number of nodes in a binary tree. The key insight is that the total count for any (non-empty) subtree is one for the root plus the counts for the left and right subtrees. Where do left and right subtree counts come from? Calls to function **count** on the subtrees will compute this for us. Thus, we can implement **count** as follows.

```
template <typename E>
int count(BinNode<E>* root) {
  if (root == NULL) return 0;  // Nothing to count
  return 1 + count(root->left())
           + count(root->right());
}
```

Another problem that occurs when recursively processing data collections is controlling which members of the collection will be visited. For example, some tree "traversals" might in fact visit only some tree nodes, while avoiding processing of others. Exercise 5.20 must solve exactly this problem in the context of a binary search tree. It must visit only those children of a given node that might possibly fall within a given range of values. Fortunately, it requires only a simple local calculation to determine which child(ren) to visit.

A more difficult situation is illustrated by the following problem. Given an arbitrary binary tree we wish to determine if, for every node A, are all nodes in A's left subtree less than the value of A, and are all nodes in A's right subtree greater than the value of A? (This happens to be the definition for a binary search tree, described in Section 5.4.) Unfortunately, to make this decision we need to know some context that is not available just by looking at the node's parent or children. As shown by Figure 5.6, it is not enough to verify that A's left child has a value less than that of A, and that A's right child has a greater value. Nor is it enough to verify that A has a value consistent with that of its parent. In fact, we need to know information about what range of values is legal for a given node. That information might come from any of the node's ancestors. Thus, relevant range information must be passed down the tree. We can implement this function as follows.

```
template <typename Key, typename E>
bool checkBST(BSTNode<Key,E>* root, Key low, Key high) {
  if (root == NULL) return true; // Empty subtree
  Key rootkey = root->key();
  if ((rootkey < low) || (rootkey > high))
    return false; // Out of range
  if (!checkBST<Key,E>(root->left(), low, rootkey))
    return false; // Left side failed
  return checkBST<Key,E>(root->right(), rootkey, high);
}
```

5.3 Binary Tree Node Implementations

In this section we examine ways to implement binary tree nodes. We begin with some options for pointer-based binary tree node implementations. Then comes a discussion on techniques for determining the space requirements for a given implementation. The section concludes with an introduction to the array-based implementation for complete binary trees.

5.3.1 Pointer-Based Node Implementations

By definition, all binary tree nodes have two children, though one or both children can be empty. Binary tree nodes typically contain a value field, with the type of the field depending on the application. The most common node implementation includes a value field and pointers to the two children.

Figure 5.7 shows a simple implementation for the **BinNode** abstract class, which we will name **BSTNode**. Class **BSTNode** includes a data member of type **E**, (which is the second template parameter) for the element type. To support search structures such as the Binary Search Tree, an additional field is included, with corresponding access methods, to store a key value (whose purpose is explained in Section 4.4). Its type is determined by the first template parameter, named **Key**. Every **BSTNode** object also has two pointers, one to its left child and another to its right child. Overloaded **new** and **delete** operators could be added to support a freelist, as described in Section 4.1.2.Figure 5.8 illustrates the **BSTNode** implementation.

Some programmers find it convenient to add a pointer to the node's parent, allowing easy upward movement in the tree. Using a parent pointer is somewhat analogous to adding a link to the previous node in a doubly linked list. In practice, the parent pointer is almost always unnecessary and adds to the space overhead for the tree implementation. It is not just a problem that parent pointers take space. More importantly, many uses of the parent pointer are driven by improper understanding of recursion and so indicate poor programming. If you are inclined toward using a parent pointer, consider if there is a more efficient implementation possible.

```
// Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
  Key k;                    // The node's key
  E it;                     // The node's value
  BSTNode* lc;              // Pointer to left child
  BSTNode* rc;              // Pointer to right child

public:
  // Two constructors -- with and without initial values
  BSTNode() { lc = rc = NULL; }
  BSTNode(Key K, E e, BSTNode* l =NULL, BSTNode* r =NULL)
    { k = K; it = e; lc = l; rc = r; }
  ~BSTNode() {}             // Destructor

  // Functions to set and return the value and key
  E& element() { return it; }
  void setElement(const E& e) { it = e; }
  Key& key() { return k; }
  void setKey(const Key& K) { k = K; }

  // Functions to set and return the children
  inline BSTNode* left() const { return lc; }
  void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
  inline BSTNode* right() const { return rc; }
  void setRight(BinNode<E>* b) { rc = (BSTNode*)b; }

  // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};
```

Figure 5.7 A binary tree node class implementation.

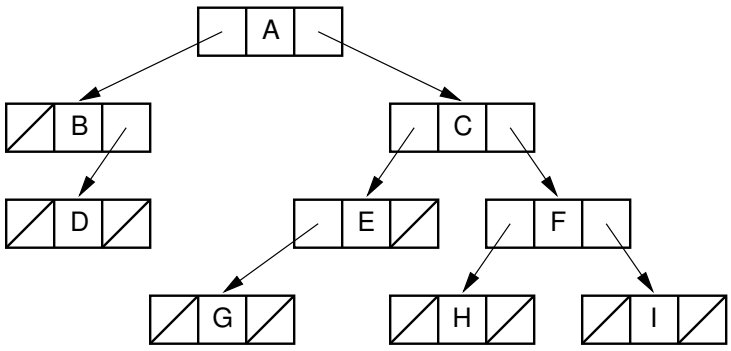


Figure 5.8 Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value.

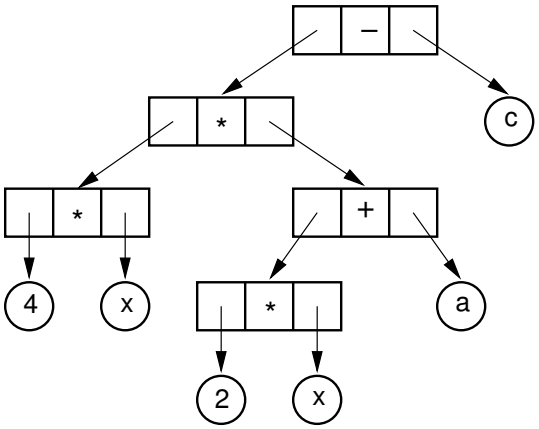


Figure 5.9 An expression tree for $4x(2x + a) - c$.

An important decision in the design of a pointer-based node implementation is whether the same class definition will be used for leaves and internal nodes. Using the same class for both will simplify the implementation, but might be an inefficient use of space. Some applications require data values only for the leaves. Other applications require one type of value for the leaves and another for the internal nodes. Examples include the binary trie of Section 13.1, the PR quadtree of Section 13.3, the Huffman coding tree of Section 5.6, and the expression tree illustrated by Figure 5.9. By definition, only internal nodes have non-empty children. If we use the same node implementation for both internal and leaf nodes, then both must store the child pointers. But it seems wasteful to store child pointers in the leaf nodes. Thus, there are many reasons why it can save space to have separate implementations for internal and leaf nodes.

As an example of a tree that stores different information at the leaf and internal nodes, consider the expression tree illustrated by Figure 5.9. The expression tree represents an algebraic expression composed of binary operators such as addition, subtraction, multiplication, and division. Internal nodes store operators, while the leaves store operands. The tree of Figure 5.9 represents the expression $4x(2x + a) - c$. The storage requirements for a leaf in an expression tree are quite different from those of an internal node. Internal nodes store one of a small set of operators, so internal nodes could store a small code identifying the operator such as a single byte for the operator's character symbol. In contrast, leaves store variable names or numbers, which is considerably larger in order to handle the wider range of possible values. At the same time, leaf nodes need not store child pointers.

C++ allows us to differentiate leaf from internal nodes through the use of class inheritance. A **base class** provides a general definition for an object, and a **subclass** modifies a base class to add more detail. A base class can be declared for binary tree nodes in general, with subclasses defined for the internal and leaf nodes. The base class of Figure 5.10 is named **`VarBinNode`**. It includes a virtual member function

named **isLeaf**, which indicates the node type. Subclasses for the internal and leaf node types each implement **isLeaf**. Internal nodes store child pointers of the base class type; they do not distinguish their children's actual subclass. Whenever a node is examined, its version of **isLeaf** indicates the node's subclass.

Figure 5.10 includes two subclasses derived from class **VarBinNode**, named **LeafNode** and **IntlNode**. Class **IntlNode** can access its children through pointers of type **VarBinNode**. Function **traverse** illustrates the use of these classes. When **traverse** calls method **isLeaf**, C++'s runtime environment determines which subclass this particular instance of **rt** happens to be and calls that subclass's version of **isLeaf**. Method **isLeaf** then provides the actual node type to its caller. The other member functions for the derived subclasses are accessed by type-casting the base class pointer as appropriate, as shown in function **traverse**.

There is another approach that we can take to represent separate leaf and internal nodes, also using a virtual base class and separate node classes for the two types. This is to implement nodes using the **composite design pattern**. This approach is noticeably different from the one of Figure 5.10 in that the node classes themselves implement the functionality of **traverse**. Figure 5.11 shows the implementation. Here, base class **VarBinNode** declares a member function **traverse** that each subclass must implement. Each subclass then implements its own appropriate behavior for its role in a traversal. The whole traversal process is called by invoking **traverse** on the root node, which in turn invokes **traverse** on its children.

When comparing the implementations of Figures 5.10 and 5.11, each has advantages and disadvantages. The first does not require that the node classes know about the **traverse** function. With this approach, it is easy to add new methods to the tree class that do other traversals or other operations on nodes of the tree. However, we see that **traverse** in Figure 5.10 does need to be familiar with each node subclass. Adding a new node subclass would therefore require modifications to the **traverse** function. In contrast, the approach of Figure 5.11 requires that any new operation on the tree that requires a traversal also be implemented in the node subclasses. On the other hand, the approach of Figure 5.11 avoids the need for the **traverse** function to know anything about the distinct abilities of the node subclasses. Those subclasses handle the responsibility of performing a traversal on themselves. A secondary benefit is that there is no need for **traverse** to explicitly enumerate all of the different node subclasses, directing appropriate action for each. With only two node classes this is a minor point. But if there were many such subclasses, this could become a bigger problem. A disadvantage is that the traversal operation must not be called on a **NULL** pointer, because there is no object to catch the call. This problem could be avoided by using a flyweight (see Section 1.3.1) to implement empty nodes.

Typically, the version of Figure 5.10 would be preferred in this example if **traverse** is a member function of the tree class, and if the node subclasses are

```
// Node implementation with simple inheritance
class VarBinNode {   // Node abstract base class
public:
  virtual ~VarBinNode() {}
  virtual bool isLeaf() = 0;    // Subclasses must implement
};


class LeafNode : public VarBinNode { // Leaf node
private:
  Operand var;                       // Operand value

public:
  LeafNode(const Operand& val) { var = val; } // Constructor
  bool isLeaf() { return true; }     // Version for LeafNode
  Operand value() { return var; }    // Return node value
};


class IntlNode : public VarBinNode { // Internal node
private:
  VarBinNode* left;                  // Left child
  VarBinNode* right;                 // Right child
  Operator opx;                      // Operator value

public:
  IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
    { opx = op; left = l; right = r; } // Constructor
  bool isLeaf() { return false; }    // Version for IntlNode
  VarBinNode* leftchild() { return left; }   // Left child
  VarBinNode* rightchild() { return right; } // Right child
  Operator value() { return opx; }           // Value
};


void traverse(VarBinNode *root) {    // Preorder traversal
  if (root == NULL) return;          // Nothing to visit
  if (root->isLeaf())                // Do leaf node
    cout << "Leaf: " << ((LeafNode *)root)->value() << endl;
  else {                             // Do internal node
    cout << "Internal: "
         << ((IntlNode *)root)->value() << endl;
    traverse(((IntlNode *)root)->leftchild());
    traverse(((IntlNode *)root)->rightchild());
  }
}
```

Figure 5.10 An implementation for separate internal and leaf node representations using C++ class inheritance and virtual functions.

```
// Node implementation with the composite design pattern
class VarBinNode {   // Node abstract base class
public:
  virtual ~VarBinNode() {}     // Generic destructor
  virtual bool isLeaf() = 0;
  virtual void traverse() = 0;
};


class LeafNode : public VarBinNode { // Leaf node
private:
  Operand var;                       // Operand value

public:
  LeafNode(const Operand& val) { var = val; } // Constructor
  bool isLeaf() { return true; }     // isLeaf for Leafnode
  Operand value() { return var; }    // Return node value
  void traverse() { cout << "Leaf: " << value() << endl; }
};


class IntlNode : public VarBinNode { // Internal node
private:
  VarBinNode* lc;                    // Left child
  VarBinNode* rc;                    // Right child
  Operator opx;                      // Operator value

public:
  IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
    { opx = op; lc = l; rc = r; }    // Constructor

  bool isLeaf() { return false; }    // isLeaf for IntlNode
  VarBinNode* left() { return lc; }  // Left child
  VarBinNode* right() { return rc; } // Right child
  Operator value() { return opx; }   // Value

  void traverse() { // Traversal behavior for internal nodes
    cout << "Internal: " << value() << endl;
    if (left() != NULL) left()->traverse();
    if (right() != NULL) right()->traverse();
  }
};


// Do a preorder traversal
void traverse(VarBinNode *root) {
  if (root != NULL) root->traverse();
}
```

Figure 5.11 A second implementation for separate internal and leaf node representations using C++ class inheritance and virtual functions using the composite design pattern. Here, the functionality of **`traverse`** is embedded into the node subclasses.

hidden from users of that tree class. On the other hand, if the nodes are objects that have meaning to users of the tree separate from their existence as nodes in the tree, then the version of Figure 5.11 might be preferred because hiding the internal behavior of the nodes becomes more important.

Another advantage of the composite design is that implementing each node type's functionality might be easier. This is because you can focus solely on the information passing and other behavior needed by this node type to do its job. This breaks down the complexity that many programmers feel overwhelmed by when dealing with complex information flows related to recursive processing.

5.3.2 Space Requirements

This section presents techniques for calculating the amount of overhead required by a binary tree implementation. Recall that overhead is the amount of space necessary to maintain the data structure. In other words, it is any space not used to store data records. The amount of overhead depends on several factors including which nodes store data values (all nodes, or just the leaves), whether the leaves store child pointers, and whether the tree is a full binary tree.

In a simple pointer-based implementation for the binary tree such as that of Figure 5.7, every node has two pointers to its children (even when the children are **NULL**). This implementation requires total space amounting to $n(2P + D)$ for a tree of n nodes. Here, P stands for the amount of space required by a pointer, and D stands for the amount of space required by a data value. The total overhead space will be $2Pn$ for the entire tree. Thus, the overhead fraction will be $2P/(2P + D)$. The actual value for this expression depends on the relative size of pointers versus data fields. If we arbitrarily assume that $P = D$, then a full tree has about two thirds of its total space taken up in overhead. Worse yet, Theorem 5.2 tells us that about half of the pointers are "wasted" **NULL** values that serve only to indicate tree structure, but which do not provide access to new data.

A common implementation is not to store any actual data in a node, but rather a pointer to the data record. In this case, each node will typically store three pointers, all of which are overhead, resulting in an overhead fraction of $3P/(3P + D)$.

If only leaves store data values, then the fraction of total space devoted to overhead depends on whether the tree is full. If the tree is not full, then conceivably there might only be one leaf node at the end of a series of internal nodes. Thus, the overhead can be an arbitrarily high percentage for non-full binary trees. The overhead fraction drops as the tree becomes closer to full, being lowest when the tree is truly full. In this case, about one half of the nodes are internal.

Great savings can be had by eliminating the pointers from leaf nodes in full binary trees. Again assume the tree stores a pointer to the data field. Because about half of the nodes are leaves and half internal nodes, and because only internal nodes

now have child pointers, the overhead fraction in this case will be approximately

$$\frac{\frac{n}{2}(2P)}{\frac{n}{2}(2P) + Dn} = \frac{P}{P + D}.$$

If $P = D$, the overhead drops to about one half of the total space. However, if only leaf nodes store useful information, the overhead fraction for this implementation is actually three quarters of the total space, because half of the "data" space is unused.

If a full binary tree needs to store data only at the leaf nodes, a better implementation would have the internal nodes store two pointers and no data field while the leaf nodes store only a pointer to the data field. This implementation requires $\frac{n}{2}2P+\frac{n}{2}(p+d)$ units of space. If $P = D$, then the overhead is $3P/(3P+D) = 3/4$. It might seem counter-intuitive that the overhead ratio has gone up while the total amount of space has gone down. The reason is because we have changed our definition of "data" to refer only to what is stored in the leaf nodes, so while the overhead fraction is higher, it is from a total storage requirement that is lower.

There is one serious flaw with this analysis. When using separate implementations for internal and leaf nodes, there must be a way to distinguish between the node types. When separate node types are implemented via **C**++ subclasses, the runtime environment stores information with each object allowing it to determine, for example, the correct subclass to use when the **`isLeaf`** virtual function is called. Thus, each node requires additional space. Only one bit is truly necessary to distinguish the two possibilities. In rare applications where space is a critical resource, implementors can often find a spare bit within the node's value field in which to store the node type indicator. An alternative is to use a spare bit within a node pointer to indicate node type. For example, this is often possible when the compiler requires that structures and objects start on word boundaries, leaving the last bit of a pointer value always zero. Thus, this bit can be used to store the node-type flag and is reset to zero before the pointer is dereferenced. Another alternative when the leaf value field is smaller than a pointer is to replace the pointer to a leaf with that leaf's value. When space is limited, such techniques can make the difference between success and failure. In any other situation, such "bit packing" tricks should be avoided because they are difficult to debug and understand at best, and are often machine dependent at worst.[2]

[2]In the early to mid 1980s, I worked on a Geographic Information System that stored spatial data in quadtrees (see Section 13.3). At the time space was a critical resource, so we used a bit-packing approach where we stored the nodetype flag as the last bit in the parent node's pointer. This worked perfectly on various 32-bit workstations. Unfortunately, in those days IBM PC-compatibles used 16-bit pointers. We never did figure out how to port our code to the 16-bit machine.

5.3.3 Array Implementation for Complete Binary Trees

The previous section points out that a large fraction of the space in a typical binary tree node implementation is devoted to structural overhead, not to storing data. This section presents a simple, compact implementation for complete binary trees. Recall that complete binary trees have all levels except the bottom filled out completely, and the bottom level has all of its nodes filled in from left to right. Thus, a complete binary tree of n nodes has only one possible shape. You might think that a complete binary tree is such an unusual occurrence that there is no reason to develop a special implementation for it. However, the complete binary tree has practical uses, the most important being the heap data structure discussed in Section 5.5. Heaps are often used to implement priority queues (Section 5.5) and for external sorting algorithms (Section 8.5.2).

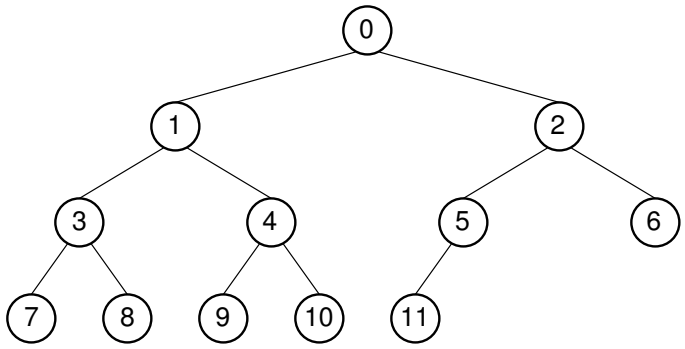
We begin by assigning numbers to the node positions in the complete binary tree, level by level, from left to right as shown in Figure 5.12(a). An array can store the tree's data values efficiently, placing each data value in the array position corresponding to that node's position within the tree. Figure 5.12(b) lists the array indices for the children, parent, and siblings of each node in Figure 5.12(a). From Figure 5.12(b), you should see a pattern regarding the positions of a node's relatives within the array. Simple formulas can be derived for calculating the array index for each relative of a node r from r's index. No explicit pointers are necessary to reach a node's left or right child. This means there is no overhead to the array implementation if the array is selected to be of size n for a tree of n nodes.

The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to $n - 1$.

- Parent$(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- Left child$(r) = 2r + 1$ if $2r + 1 < n$.
- Right child$(r) = 2r + 2$ if $2r + 2 < n$.
- Left sibling$(r) = r - 1$ if r is even.
- Right sibling$(r) = r + 1$ if r is odd and $r + 1 < n$.

5.4 Binary Search Trees

Section 4.4 presented the dictionary ADT, along with dictionary implementations based on sorted and unsorted lists. When implementing the dictionary with an unsorted list, inserting a new record into the dictionary can be performed quickly by putting it at the end of the list. However, searching an unsorted list for a particular record requires $\Theta(n)$ time in the average case. For a large database, this is probably much too slow. Alternatively, the records can be stored in a sorted list. If the list is implemented using a linked list, then no speedup to the search operation will



(a)

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

(b)

Figure 5.12 A complete binary tree and its array implementation. (a) The complete binary tree with twelve nodes. Each node has been labeled with its position in the tree. (b) The positions for the relatives of each node. A dash indicates that the relative does not exist.

result from storing the records in sorted order. On the other hand, if we use a sorted array-based list to implement the dictionary, then binary search can be used to find a record in only $\Theta(\log n)$ time. However, insertion will now require $\Theta(n)$ time on average because, once the proper location for the new record in the sorted list has been found, many records might be shifted to make room for the new record.

Is there some way to organize a collection of records so that inserting records and searching for records can both be done quickly? This section presents the binary search tree (BST), which allows an improved solution to this problem.

A BST is a binary tree that conforms to the following condition, known as the **Binary Search Tree Property**: All nodes stored in the left subtree of a node whose key value is K have key values less than K. All nodes stored in the right subtree of a node whose key value is K have key values greater than or equal to K. Figure 5.13 shows two BSTs for a collection of values. One consequence of the Binary Search Tree Property is that if the BST nodes are printed using an inorder traversal (see Section 5.2), the resulting enumeration will be in sorted order from lowest to highest.

Figure 5.14 shows a class declaration for the BST that implements the dictionary ADT. The public member functions include those required by the dictionary

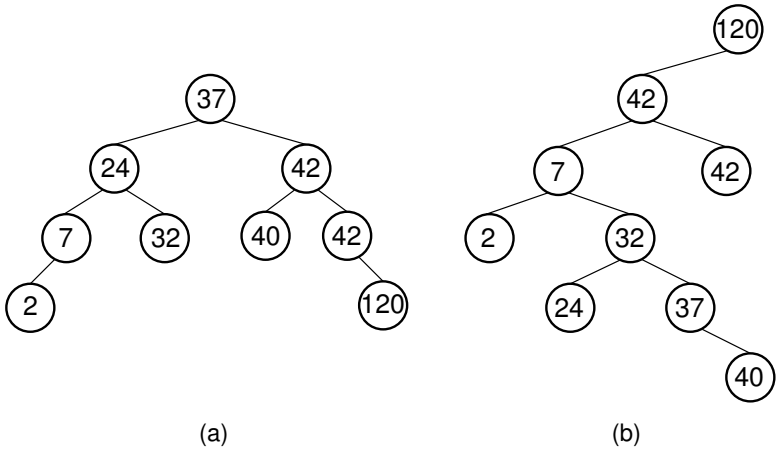


Figure 5.13 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

ADT, along with a constructor and destructor. Recall from the discussion in Section 4.4 that there are various ways to deal with keys and comparing records (three approaches being key/value pairs, a special comparison method, and passing in a comparator function). Our BST implementation will handle comparison by explicitly storing a key separate from the data value at each node of the tree.

To find a record with key value K in a BST, begin at the root. If the root stores a record with key value K, then the search is over. If not, then we must search deeper in the tree. What makes the BST efficient during search is that we need search only one of the node's two subtrees. If K is less than the root node's key value, we search only the left subtree. If K is greater than the root node's key value, we search only the right subtree. This process continues until a record with key value K is found, or we reach a leaf node. If we reach a leaf node without encountering K, then no record exists in the BST whose key value is K.

Example 5.5 Consider searching for the node with key value 32 in the tree of Figure 5.13(a). Because 32 is less than the root value of 37, the search proceeds to the left subtree. Because 32 is greater than 24, we search in 24's right subtree. At this point the node containing 32 is found. If the search value were 35, the same path would be followed to the node containing 32. Because this node has no children, we know that 35 is not in the BST.

Notice that in Figure 5.14, public member function **`find`** calls private member function **`findhelp`**. Method **`find`** takes the search key as an explicit parameter and its BST as an implicit parameter, and returns the record that matches the key.

```
// Binary Search Tree implementation for the Dictionary ADT
template <typename Key, typename E>
class BST : public Dictionary<Key,E> {
private:
  BSTNode<Key,E>* root;   // Root of the BST
  int nodecount;          // Number of nodes in the BST

  // Private "helper" functions
  void clearhelp(BSTNode<Key, E>*);
  BSTNode<Key,E>* inserthelp(BSTNode<Key, E>*,
                             const Key&, const E&);
  BSTNode<Key,E>* deletemin(BSTNode<Key, E>*);
  BSTNode<Key,E>* getmin(BSTNode<Key, E>*);
  BSTNode<Key,E>* removehelp(BSTNode<Key, E>*, const Key&);
  E findhelp(BSTNode<Key, E>*, const Key&) const;
  void printhelp(BSTNode<Key, E>*, int) const;

public:
  BST() { root = NULL; nodecount = 0; }  // Constructor
  ~BST() { clearhelp(root); }            // Destructor

  void clear()   // Reinitialize tree
    { clearhelp(root); root = NULL; nodecount = 0; }

  // Insert a record into the tree.
  // k Key value of the record.
  // e The record to insert.
  void insert(const Key& k, const E& e) {
    root = inserthelp(root, k, e);
    nodecount++;
  }

  // Remove a record from the tree.
  // k Key value of record to remove.
  // Return: The record removed, or NULL if there is none.
  E remove(const Key& k) {
    E temp = findhelp(root, k);   // First find it
    if (temp != NULL) {
      root = removehelp(root, k);
      nodecount--;
    }
    return temp;
  }
```

Figure 5.14 The binary search tree implementation.

```
  // Remove and return the root node from the dictionary.
  // Return: The record removed, null if tree is empty.
  E removeAny() {  // Delete min value
    if (root != NULL) {
      E temp = root->element();
      root = removehelp(root, root->key());
      nodecount--;
      return temp;
    }
    else return NULL;
  }

  // Return Record with key value k, NULL if none exist.
  // k: The key value to find. */
  // Return some record matching "k".
  // Return true if such exists, false otherwise. If
  // multiple records match "k", return an arbitrary one.
  E find(const Key& k) const { return findhelp(root, k); }

  // Return the number of records in the dictionary.
  int size() { return nodecount; }

  void print() const { // Print the contents of the BST
    if (root == NULL) cout << "The BST is empty.\n";
    else printhelp(root, 0);
  }
};
```

Figure 5.14 (continued)

However, the find operation is most easily implemented as a recursive function whose parameters are the root of a subtree and the search key. Member **findhelp** has the desired form for this recursive subroutine and is implemented as follows.

```
template <typename Key, typename E>
E BST<Key, E>::findhelp(BSTNode<Key, E>* root,
                        const Key& k) const {
  if (root == NULL) return NULL;          // Empty tree
  if (k < root->key())
    return findhelp(root->left(), k);   // Check left
  else if (k > root->key())
    return findhelp(root->right(), k);  // Check right
  else return root->element();  // Found it
}
```

Once the desired record is found, it is passed through return values up the chain of recursive calls. If a suitable record is not found, **null** is returned.

Inserting a record with key value k requires that we first find where that record would have been if it were in the tree. This takes us to either a leaf node, or to an

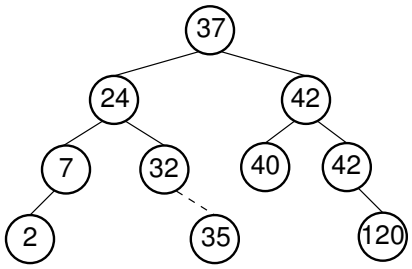


Figure 5.15 An example of BST insertion. A record with value 35 is inserted into the BST of Figure 5.13(a). The node with value 32 becomes the parent of the new node containing 35.

internal node with no child in the appropriate direction.[3] Call this node R'. We then add a new node containing the new record as a child of R'. Figure 5.15 illustrates this operation. The value 35 is added as the right child of the node with value 32. Here is the implementation for **inserthelp**:

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::inserthelp(
    BSTNode<Key, E>* root, const Key& k, const E& it) {
  if (root == NULL)  // Empty tree: create node
    return new BSTNode<Key, E>(k, it, NULL, NULL);
  if (k < root->key())
    root->setLeft(inserthelp(root->left(), k, it));
  else root->setRight(inserthelp(root->right(), k, it));
  return root;       // Return tree with node inserted
}
```

You should pay careful attention to the implementation for **inserthelp**. Note that **inserthelp** returns a pointer to a **BSTNode**. What is being returned is a subtree identical to the old subtree, except that it has been modified to contain the new record being inserted. Each node along a path from the root to the parent of the new node added to the tree will have its appropriate child pointer assigned to it. Except for the last node in the path, none of these nodes will actually change their child's pointer value. In that sense, many of the assignments seem redundant. However, the cost of these additional assignments is worth paying to keep the insertion process simple. The alternative is to check if a given assignment is necessary, which is probably more expensive than the assignment!

The shape of a BST depends on the order in which elements are inserted. A new element is added to the BST as a new leaf node, potentially increasing the depth of the tree. Figure 5.13 illustrates two BSTs for a collection of values. It is possible

[3]This assumes that no node has a key value equal to the one being inserted. If we find a node that duplicates the key value to be inserted, we have two options. If the application does not allow nodes with equal keys, then this insertion should be treated as an error (or ignored). If duplicate keys are allowed, our convention will be to insert the duplicate in the right subtree.

for the BST containing n nodes to be a chain of nodes with height n. This would happen if, for example, all elements were inserted in sorted order. In general, it is preferable for a BST to be as shallow as possible. This keeps the average cost of a BST operation low.

Removing a node from a BST is a bit trickier than inserting a node, but it is not complicated if all of the possible cases are considered individually. Before tackling the general node removal process, let us first discuss how to remove from a given subtree the node with the smallest key value. This routine will be used later by the general node removal function. To remove the node with the minimum key value from a subtree, first find that node by continuously moving down the left link until there is no further left link to follow. Call this node *S*. To remove *S*, simply have the parent of *S* change its pointer to point to the right child of *S*. We know that *S* has no left child (because if *S* did have a left child, *S* would not be the node with minimum key value). Thus, changing the pointer as described will maintain a BST, with *S* removed. The code for this method, named **deletemin**, is as follows:

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
    return rt->right();
  else {                        // Continue left
    rt->setLeft(deletemin(rt->left()));
    return rt;
  }
}
```

Example 5.6 Figure 5.16 illustrates the **deletemin** process. Beginning at the root node with value 10, **deletemin** follows the left link until there is no further left link, in this case reaching the node with value 5. The node with value 10 is changed to point to the right child of the node containing the minimum value. This is indicated in Figure 5.16 by a dashed line.

A pointer to the node containing the minimum-valued element is stored in parameter **S**. The return value of the **deletemin** method is the subtree of the current node with the minimum-valued node in the subtree removed. As with method **inserthelp**, each node on the path back to the root has its left child pointer reassigned to the subtree resulting from its call to the **deletemin** method.

A useful companion method is **getmin** which returns a pointer to the node containing the minimum value in the subtree.

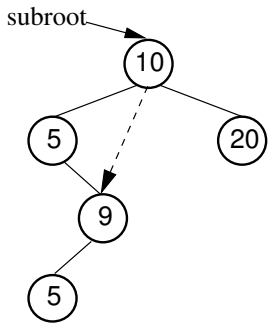


Figure 5.16 An example of deleting the node with minimum value. In this tree, the node with minimum value, 5, is the left child of the root. Thus, the root's **left** pointer is changed to point to 5's right child.

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
getmin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL)
    return rt;
  else return getmin(rt->left());
}
```

Removing a node with given key value R from the BST requires that we first find R and then remove it from the tree. So, the first part of the remove operation is a search to find R. Once R is found, there are several possibilities. If R has no children, then R's parent has its pointer set to **NULL**. If R has one child, then R's parent has its pointer set to R's child (similar to **deletemin**). The problem comes if R has two children. One simple approach, though expensive, is to set R's parent to point to one of R's subtrees, and then reinsert the remaining subtree's nodes one at a time. A better alternative is to find a value in one of the subtrees that can replace the value in R.

Thus, the question becomes: Which value can substitute for the one being removed? It cannot be any arbitrary value, because we must preserve the BST property without making major changes to the structure of the tree. Which value is most like the one being removed? The answer is the least key value greater than (or equal to) the one being removed, or else the greatest key value less than the one being removed. If either of these values replace the one being removed, then the BST property is maintained.

Example 5.7 Assume that we wish to remove the value 37 from the BST of Figure 5.13(a). Instead of removing the root node, we remove the node with the least value in the right subtree (using the **deletemin** operation). This value can then replace the value in the root. In this example we first remove the node with value 40, because it contains the least value in the

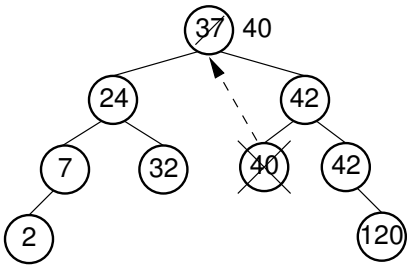


Figure 5.17 An example of removing the value 37 from the BST. The node containing this value has two children. We replace value 37 with the least value from the node's right subtree, in this case 40.

right subtree. We then substitute 40 as the new value for the root node. Figure 5.17 illustrates this process.

When duplicate node values do not appear in the tree, it makes no difference whether the replacement is the greatest value from the left subtree or the least value from the right subtree. If duplicates are stored, then we must select the replacement from the *right* subtree. To see why, call the greatest value in the left subtree *G*. If multiple nodes in the left subtree have value *G*, selecting *G* as the replacement value for the root of the subtree will result in a tree with equal values to the left of the node now containing *G*. Precisely this situation occurs if we replace value 120 with the greatest value in the left subtree of Figure 5.13(b). Selecting the least value from the right subtree does not have a similar problem, because it does not violate the Binary Search Tree Property if equal values appear in the right subtree.

From the above, we see that if we want to remove the record stored in a node with two children, then we simply call **`deletemin`** on the node's right subtree and substitute the record returned for the record being removed. Figure 5.18 shows an implementation for **`removehelp`**.

The cost for **`findhelp`** and **`inserthelp`** is the depth of the node found or inserted. The cost for **`removehelp`** is the depth of the node being removed, or in the case when this node has two children, the depth of the node with smallest value in its right subtree. Thus, in the worst case, the cost for any one of these operations is the depth of the deepest node in the tree. This is why it is desirable to keep BSTs **balanced**, that is, with least possible height. If a binary tree is balanced, then the height for a tree of n nodes is approximately $\log n$. However, if the tree is completely unbalanced, for example in the shape of a linked list, then the height for a tree with n nodes can be as great as n. Thus, a balanced BST will in the average case have operations costing $\Theta(\log n)$, while a badly unbalanced BST can have operations in the worst case costing $\Theta(n)$. Consider the situation where we construct a BST of n nodes by inserting records one at a time. If we are fortunate to have them arrive in an order that results in a balanced tree (a "random" order is

```
// Remove a node with key value k
// Return: The tree with the node removed
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
removehelp(BSTNode<Key, E>* rt, const Key& k) {
  if (rt == NULL) return NULL;    // k is not in tree
  else if (k < rt->key())
    rt->setLeft(removehelp(rt->left(), k));
  else if (k > rt->key())
    rt->setRight(removehelp(rt->right(), k));
  else {                            // Found: remove it
    BSTNode<Key, E>* temp = rt;
    if (rt->left() == NULL) {     // Only a right child
      rt = rt->right();          //  so point to right
      delete temp;
    }
    else if (rt->right() == NULL) { // Only a left child
      rt = rt->left();           //  so point to left
      delete temp;
    }
    else {                      // Both children are non-empty
      BSTNode<Key, E>* temp = getmin(rt->right());
      rt->setElement(temp->element());
      rt->setKey(temp->key());
      rt->setRight(deletemin(rt->right()));
      delete temp;
    }
  }
  return rt;
}
```

Figure 5.18 Implementation for the BST **removehelp** method.

likely to be good enough for this purpose), then each insertion will cost on average $\Theta(\log n)$, for a total cost of $\Theta(n \log n)$. However, if the records are inserted in order of increasing value, then the resulting tree will be a chain of height n. The cost of insertion in this case will be $\sum_{i=1}^{n} i = \Theta(n^2)$.

Traversing a BST costs $\Theta(n)$ regardless of the shape of the tree. Each node is visited exactly once, and each child pointer is followed exactly once.

Below are two example traversals. The first is member **clearhelp**, which returns the nodes of the BST to the freelist. Because the children of a node must be freed before the node itself, this is a postorder traversal.

```
template <typename Key, typename E>
void BST<Key, E>::
clearhelp(BSTNode<Key, E>* root) {
  if (root == NULL) return;
  clearhelp(root->left());
  clearhelp(root->right());
  delete root;
}
```

The next example is **printhelp**, which performs an inorder traversal on the BST to print the node values in ascending order. Note that **printhelp** indents each line to indicate the depth of the corresponding node in the tree. Thus we pass in the current level of the tree in **level**, and increment this value each time that we make a recursive call.

```
template <typename Key, typename E>
void BST<Key, E>::
printhelp(BSTNode<Key, E>* root, int level) const {
  if (root == NULL) return;           // Empty tree
  printhelp(root->left(), level+1);   // Do left subtree
  for (int i=0; i<level; i++)         // Indent to level
    cout << "  ";
  cout << root->key() << "\n";        // Print node value
  printhelp(root->right(), level+1);  // Do right subtree
}
```

While the BST is simple to implement and efficient when the tree is balanced, the possibility of its being unbalanced is a serious liability. There are techniques for organizing a BST to guarantee good performance. Two examples are the AVL tree and the splay tree of Section 13.2. Other search trees are guaranteed to remain balanced, such as the 2-3 tree of Section 10.4.

5.5 Heaps and Priority Queues

There are many situations, both in real life and in computing applications, where we wish to choose the next "most important" from a collection of people, tasks, or objects. For example, doctors in a hospital emergency room often choose to see next the "most critical" patient rather than the one who arrived first. When scheduling programs for execution in a multitasking operating system, at any given moment there might be several programs (usually called **jobs**) ready to run. The next job selected is the one with the highest **priority**. Priority is indicated by a particular value associated with the job (and might change while the job remains in the wait list).

When a collection of objects is organized by importance or priority, we call this a **priority queue**. A normal queue data structure will not implement a priority queue efficiently because search for the element with highest priority will take $\Theta(n)$ time. A list, whether sorted or not, will also require $\Theta(n)$ time for either insertion or removal. A BST that organizes records by priority could be used, with the total of n inserts and n remove operations requiring $\Theta(n \log n)$ time in the average case. However, there is always the possibility that the BST will become unbalanced, leading to bad performance. Instead, we would like to find a data structure that is guaranteed to have good performance for this special application.