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

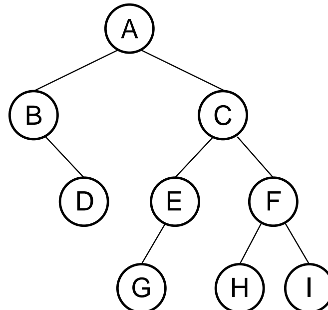


Figure 5.1 A binary tree. Node *A* is the root. Nodes *B* and *C* are *A*'s children. Nodes *B* and *D* together form a subtree. Node *B* has two children: Its left child is the empty tree and its right child is *D*. Nodes *A*, *C*, and *E* are ancestors of *G*. Nodes *D*, *E*, and *F* make up level 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.

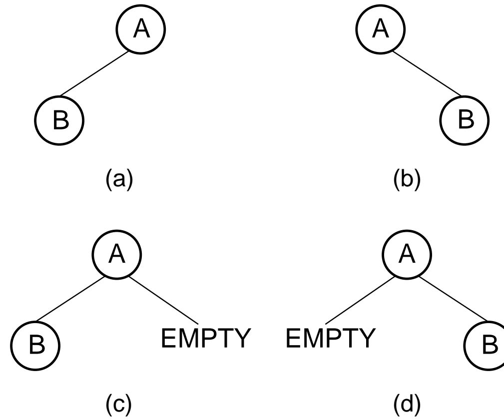


Figure 5.2 Two different binary trees. (a) A binary tree whose root has a non-empty left child. (b) A binary tree whose root has a non-empty right child. (c) The binary tree of (a) with the missing right child made explicit. (d) The binary tree of (b) with the missing left child made explicit.

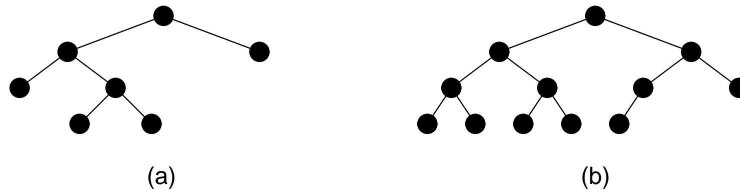


Figure 5.3 Examples of full and complete binary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

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

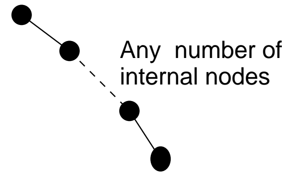


Figure 5.4 A tree containing many internal nodes and a single leaf.

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 $\mathbf{T}'$. $\mathbf{T}'$ has $n - 1$ internal nodes. From the induction hypothesis, $\mathbf{T}'$ 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 $\mathbf{T}'$ has n leaves, adding the two children yields $n+2$. However, node *I* counted as one of the leaves in $\mathbf{T}'$ 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 $\mathbf{T}'$. All nodes originally in **T** will be internal nodes in $\mathbf{T}'$ (because even the leaf nodes of **T** have children in $\mathbf{T}'$). $\mathbf{T}'$ is a full binary tree, because every internal node of **T** now must have two children in $\mathbf{T}'$, and each leaf node in **T** must have two children in $\mathbf{T}'$ (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 $\mathbf{T}'$ is one more than the number of nodes in **T**. Each leaf node in $\mathbf{T}'$ 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 generic 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 reference to the left child, set or return a reference 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**.

```
/** ADT for binary tree nodes */
public interface BinNode<E> {
  /** Get and set the element value */
  public E element();
  public void setElement(E v);

  /** @return The left child */
  public BinNode<E> left();

  /** @return The right child */
  public BinNode<E> right();

  /** @return True if a leaf node, false otherwise */
  public boolean isLeaf();
}
```

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 reference to a node which we will call `rt` because each node can be

viewed as the root of a some subtree. The initial call to the traversal function passes in a reference to the root node of the tree. The traversal function visits **rt** and its children (if any) in the desired order. For example, a preorder traversal specifies that **rt** be visited before its children. This can easily be implemented as follows.

```
/** @param rt is the root of the subtree */
void preorder(BinNode rt)
{
  if (rt == null) return; // Empty subtree - do nothing
  visit(rt);              // Process root node
  preorder(rt.left());    // Process all nodes in left
  preorder(rt.right());   // Process all nodes in 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 **rt** is **null**. If not, it will recursively call itself on the left and right children of **rt**. 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:

```
void preorder2(BinNode rt)
{
  visit(rt);
  if (rt.left() != null) preorder2(rt.left());
  if (rt.right() != null) preorder2(rt.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 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.

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.

```
int count(BinNode rt) {
  if (rt == null) return 0;  // Nothing to count
  return 1 + count(rt.left()) + count(rt.right());
}
```

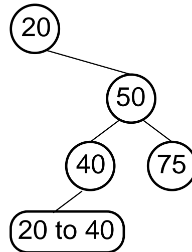


Figure 5.6 To be a binary search tree, the left child of the node with value 40 must have a value between 20 and 40.

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.

```
boolean checkBST(BinNode<Integer> rt,
                 int low, int high) {
  if (rt == null) return true; // Empty subtree
  int rootkey = rt.element();
  if ((rootkey < low) || (rootkey > high))
    return false; // Out of range
  if (!checkBST(rt.left(), low, rootkey))
    return false; // Left side failed
  return checkBST(rt.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 generic 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 generic parameter, named **Key**. Every **BSTNode** object also has two pointers, one to its left child and another to its right child. 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.

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.

```
/** Binary tree node implementation: Pointers to children
    @param E The data element
    @param Key The associated key for the record */
class BSTNode<Key, E> implements BinNode<E> {
  private Key key;              // Key for this node
  private E element;            // Element for this node
  private BSTNode<Key,E> left;  // Pointer to left child
  private BSTNode<Key,E> right; // Pointer to right child

  /** Constructors */
  public BSTNode() {left = right = null; }
  public BSTNode(Key k, E val)
  { left = right = null; key = k; element = val; }
  public BSTNode(Key k, E val,
                 BSTNode<Key,E> l, BSTNode<Key,E> r)
  { left = l; right = r; key = k; element = val; }

  /** Get and set the key value */
  public Key key() { return key; }
  public void setKey(Key k) { key = k; }

  /** Get and set the element value */
  public E element() { return element; }
  public void setElement(E v) { element = v; }

  /** Get and set the left child */
  public BSTNode<Key,E> left() { return left; }
  public void setLeft(BSTNode<Key,E> p) { left = p; }

  /** Get and set the right child */
  public BSTNode<Key,E> right() { return right; }
  public void setRight(BSTNode<Key,E> p) { right = p; }

  /** @return True if a leaf node, false otherwise */
  public boolean isLeaf()
  { return (left == null) && (right == null); }
}
```

Figure 5.7 A binary tree node class implementation.

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.

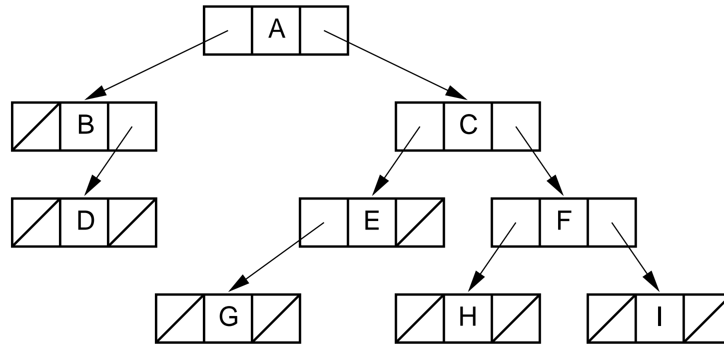


Figure 5.8 Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value.

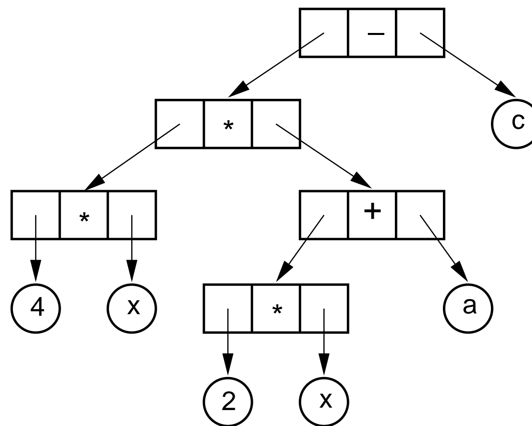


Figure 5.9 An expression tree for $4x(2x + a) - c$.

Java 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`**, Java'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

```
/** Base class for expression tree nodes */
public interface VarBinNode {
  public boolean isLeaf(); // All subclasses must implement
}

/** Leaf node */
class VarLeafNode implements VarBinNode {
  private String operand;                 // Operand value

  public VarLeafNode(String val) { operand = val; }
  public boolean isLeaf() { return true; }
  public String value() { return operand; }
};

/** Internal node */
class VarIntlNode implements VarBinNode {
  private VarBinNode left;                // Left child
  private VarBinNode right;               // Right child
  private Character operator;             // Operator value

  public VarIntlNode(Character op,
                     VarBinNode l, VarBinNode r)
    { operator = op; left = l; right = r; }
  public boolean isLeaf() { return false; }
  public VarBinNode leftchild() { return left; }
  public VarBinNode rightchild() { return right; }
  public Character value() { return operator; }
}

/** Preorder traversal */
public static void traverse(VarBinNode rt) {
  if (rt == null) return;          // Nothing to visit
  if (rt.isLeaf())                 // Process leaf node
    Visit.VisitLeafNode(((VarLeafNode)rt).value());
  else {                           // Process internal node
    Visit.VisitInternalNode(((VarIntlNode)rt).value());
    traverse(((VarIntlNode)rt).leftchild());
    traverse(((VarIntlNode)rt).rightchild());
  }
}
```

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

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 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.

```
/** Base class: Composite */
public interface VarBinNode {
  public boolean isLeaf();
  public void traverse();
}

/** Leaf node: Composite */
class VarLeafNode implements VarBinNode {
  private String operand;                // Operand value

  public VarLeafNode(String val) { operand = val; }
  public boolean isLeaf() { return true; }
  public String value() { return operand; }

  public void traverse() {
    Visit.VisitLeafNode(operand);
  }
}

/** Internal node: Composite */
class VarIntlNode implements VarBinNode { // Internal node
  private VarBinNode left;               // Left child
  private VarBinNode right;              // Right child
  private Character operator;            // Operator value

  public VarIntlNode(Character op,
                     VarBinNode l, VarBinNode r)
    { operator = op; left = l; right = r; }
  public boolean isLeaf() { return false; }
  public VarBinNode leftchild() { return left; }
  public VarBinNode rightchild() { return right; }
  public Character value() { return operator; }

  public void traverse() {
    Visit.VisitInternalNode(operator);
    if (left != null) left.traverse();
    if (right != null) right.traverse();
  }
}

/** Preorder traversal */
public static void traverse(VarBinNode rt) {
  if (rt != null) rt.traverse();
}
```

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

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.

In Java, the most typical implementation is not to store any actual data in a node, but rather a reference 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 reference 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 reference to the data field. This implementation requires

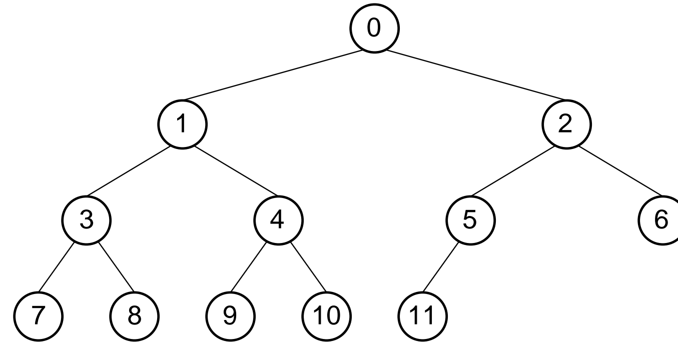
$\frac{n}{2}2P + \frac{n}{2}(p+d)$ units of space. If $P = D$, then the overhead is $3P/(3P+D) = 3/4$. It might seem counter-intuitive that the overhead ratio has gone up while the total amount of space has gone down. The reason is because we have changed our definition of "data" to refer only to what 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 Java 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]

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).

[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.



(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.

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 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 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 such as using the `Comparator` class, 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

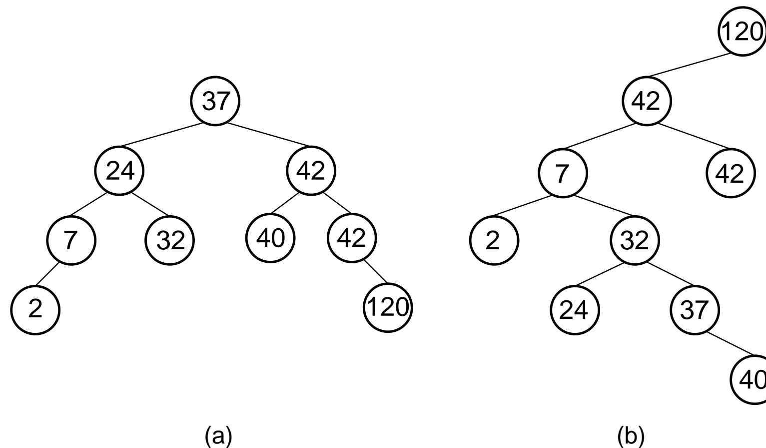


Figure 5.13 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

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. 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.

```
private E findhelp(BSTNode<Key,E> rt, Key k) {
  if (rt == null) return null;
  if (rt.key().compareTo(k) > 0)
    return findhelp(rt.left(), k);
  else if (rt.key().compareTo(k) == 0) return rt.element();
  else return findhelp(rt.right(), k);
}
```

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.

```
/** Binary Search Tree implementation for Dictionary ADT */
class BST<Key extends Comparable<? super Key>, E>
         implements Dictionary<Key, E> {
  private BSTNode<Key,E> root; // Root of the BST
  private int nodecount;       // Number of nodes in the BST

  /** Constructor */
  BST() { root = null; nodecount = 0; }

  /** Reinitialize tree */
  public void clear() { root = null; nodecount = 0; }

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

  /** Remove a record from the tree.
      @param k Key value of record to remove.
      @return The record removed, null if there is none. */
  public E remove(Key k) {
    E temp = findhelp(root, k);   // First find it
    if (temp != null) {
      root = removehelp(root, k); // Now remove it
      nodecount--;
    }
    return temp;
  }

  /** Remove and return the root node from the dictionary.
      @return The record removed, null if tree is empty. */
  public E removeAny() {
    if (root == null) return null;
    E temp = root.element();
    root = removehelp(root, root.key());
    nodecount--;
    return temp;
  }

  /** @return Record with key value k, null if none exist.
      @param k The key value to find. */
  public E find(Key k) { return findhelp(root, k); }

  /** @return The number of records in the dictionary. */
  public int size() { return nodecount; }
}
```

Figure 5.14 The binary search tree implementation.

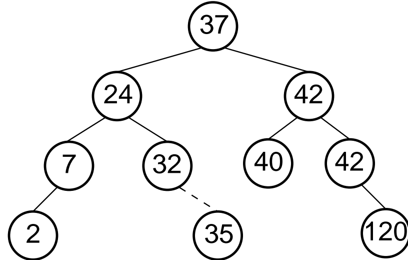


Figure 5.15 An example of BST insertion. A record with value 35 is inserted into the BST of Figure 5.13(a). The node with value 32 becomes the parent of the new node containing 35.

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 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**:

```
/** @return The current subtree, modified to contain
   the new item */
private BSTNode<Key,E> inserthelp(BSTNode<Key,E> rt,
                                  Key k, E e) {
  if (rt == null) return new BSTNode<Key,E>(k, e);
  if (rt.key().compareTo(k) > 0)
    rt.setLeft(inserthelp(rt.left(), k, e));
  else
    rt.setRight(inserthelp(rt.right(), k, e));
  return rt;
}
```

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!

[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.

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 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:

```
private BSTNode<Key,E> deletemin(BSTNode<Key,E> rt) {
  if (rt.left() == null) return rt.right();
  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 reference to the node containing the minimum value in the subtree.

```
private BSTNode<Key,E> getmin(BSTNode<Key,E> rt) {
  if (rt.left() == null) return rt;
  return getmin(rt.left());
}
```

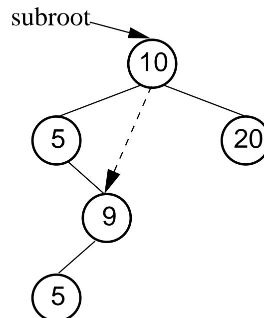


Figure 5.16 An example of deleting the node with minimum value. In this tree, the node with minimum value, 5, is the left child of the root. Thus, the root's **`left`** pointer is changed to point to 5's right child.

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 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

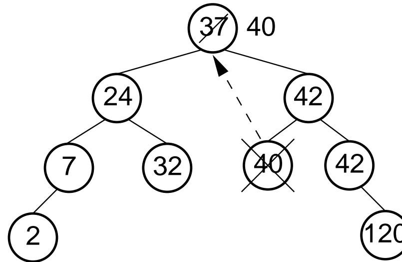


Figure 5.17 An example of removing the value 37 from the BST. The node containing this value has two children. We replace value 37 with the least value from the node's right subtree, in this case 40.

```
/** Remove a node with key value k
    @return The tree with the node removed */
private BSTNode<Key,E> removehelp(BSTNode<Key,E> rt,Key k) {
  if (rt == null) return null;
  if (rt.key().compareTo(k) > 0)
    rt.setLeft(removehelp(rt.left(), k));
  else if (rt.key().compareTo(k) < 0)
    rt.setRight(removehelp(rt.right(), k));
  else { // Found it
    if (rt.left() == null) return rt.right();
    else if (rt.right() == null) return rt.left();
    else { // Two children
      BSTNode<Key,E> temp = getmin(rt.right());
      rt.setElement(temp.element());
      rt.setKey(temp.key());
      rt.setRight(deletemin(rt.right()));
    }
  }
  return rt;
}
```

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

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 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 is an example traversal, named **`printhelp`**. It performs an inorder traversal on the BST to print the node values in ascending order.

```
private void printhelp(BSTNode<Key,E> rt) {
  if (rt == null) return;
  printhelp(rt.left());
  printVisit(rt.element());
  printhelp(rt.right());
}
```

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.

This section presents the **heap**[4] data structure. A heap is defined by two properties. First, it is a complete binary tree, so heaps are nearly always implemented using the array representation for complete binary trees presented in Section 5.3.3. Second, the values stored in a heap are **partially ordered**. This means that there is a relationship between the value stored at any node and the values of its children. There are two variants of the heap, depending on the definition of this relationship.

A **max-heap** has the property that every node stores a value that is *greater* than or equal to the value of either of its children. Because the root has a value greater than or equal to its children, which in turn have values greater than or equal to their children, the root stores the maximum of all values in the tree.

A **min-heap** has the property that every node stores a value that is *less* than or equal to that of its children. Because the root has a value less than or equal to its children, which in turn have values less than or equal to their children, the root stores the minimum of all values in the tree.

Note that there is no necessary relationship between the value of a node and that of its sibling in either the min-heap or the max-heap. For example, it is possible that the values for all nodes in the left subtree of the root are greater than the values for every node of the right subtree. We can contrast BSTs and heaps by the strength of their ordering relationships. A BST defines a total order on its nodes in that, given the positions for any two nodes in the tree, the one to the "left" (equivalently, the one appearing earlier in an inorder traversal) has a smaller key value than the one to the "right." In contrast, a heap implements a partial order. Given their positions, we can determine the relative order for the key values of two nodes in the heap *only* if one is a descendant of the other.

Min-heaps and max-heaps both have their uses. For example, the Heapsort of Section 7.6 uses the max-heap, while the Replacement Selection algorithm of Section 8.5.2 uses a min-heap. The examples in the rest of this section will use a max-heap.

Be careful not to confuse the logical representation of a heap with its physical implementation by means of the array-based complete binary tree. The two are not

[4]The term "heap" is also sometimes used to refer to a memory pool. See Section 12.3.

synonymous because the logical view of the heap is actually a tree structure, while the typical physical implementation uses an array.

Figure 5.19 shows an implementation for heaps. The class is a generic with one type parameter, **E**, which defines the type for the data elements stored in the heap. **E** must extend the **Comparable** interface, and so we can use the **compareTo** method for comparing records in the heap.

This class definition makes two concessions to the fact that an array-based implementation is used. First, heap nodes are indicated by their logical position within the heap rather than by a pointer to the node. In practice, the logical heap position corresponds to the identically numbered physical position in the array. Second, the constructor takes as input a pointer to the array to be used. This approach provides the greatest flexibility for using the heap because all data values can be loaded into the array directly by the client. The advantage of this comes during the heap construction phase, as explained below. The constructor also takes an integer parameter indicating the initial size of the heap (based on the number of elements initially loaded into the array) and a second integer parameter indicating the maximum size allowed for the heap (the size of the array).

Method **heapsize** returns the current size of the heap. **H.isLeaf(pos)** returns **true** if position **pos** is a leaf in heap **H**, and **false** otherwise. Members **leftchild**, **rightchild**, and **parent** return the position (actually, the array index) for the left child, right child, and parent of the position passed, respectively.

One way to build a heap is to insert the elements one at a time. Method **insert** will insert a new element *V* into the heap. You might expect the heap insertion process to be similar to the insert function for a BST, starting at the root and working down through the heap. However, this approach is not likely to work because the heap must maintain the shape of a complete binary tree. Equivalently, if the heap takes up the first n positions of its array prior to the call to **insert**, it must take up the first $n+1$ positions after. To accomplish this, **insert** first places *V* at position n of the array. Of course, *V* is unlikely to be in the correct position. To move *V* to the right place, it is compared to its parent's value. If the value of *V* is less than or equal to the value of its parent, then it is in the correct place and the insert routine is finished. If the value of *V* is greater than that of its parent, then the two elements swap positions. From here, the process of comparing *V* to its (current) parent continues until *V* reaches its correct position.

Since the heap is a complete binary tree, its height is guaranteed to be the minimum possible. In particular, a heap containing n nodes will have a height of $\Theta(\log n)$. Intuitively, we can see that this must be true because each level that we add will slightly more than double the number of nodes in the tree (the ith level has 2^i nodes, and the sum of the first i levels is $2^{i+1}-1$). Starting at 1, we can double only $\log n$ times to reach a value of n. To be precise, the height of a heap with n nodes is $\lceil \log(n+1) \rceil$.

```
/** Max-heap implementation */
public class MaxHeap<E extends Comparable<? super E>> {
  private E[] Heap;   // Pointer to the heap array
  private int size;   // Maximum size of the heap
  private int n;      // Number of things in heap

/** Constructor supporting preloading of heap contents */
public MaxHeap(E[] h, int num, int max)
{ Heap = h;  n = num;  size = max;  buildheap(); }

/** @return Current size of the heap */
public int heapsize() { return n; }

/** @return True if pos a leaf position, false otherwise */
public boolean isLeaf(int pos)
{ return (pos >= n/2) && (pos < n); }

/** @return Position for left child of pos */
public int leftchild(int pos) {
  assert pos < n/2 : "Position has no left child";
  return 2*pos + 1;
}

/** @return Position for right child of pos */
public int rightchild(int pos) {
  assert pos < (n-1)/2 : "Position has no right child";
  return 2*pos + 2;
}

/** @return Position for parent */
public int parent(int pos) {
  assert pos > 0 : "Position has no parent";
  return (pos-1)/2;
}

/** Insert val into heap */
public void insert(E val) {
  assert n < size : "Heap is full";
  int curr = n++;
  Heap[curr] = val;              // Start at end of heap
  // Now sift up until curr's parent's key > curr's key
  while ((curr != 0)  &&
         (Heap[curr].compareTo(Heap[parent(curr)]) > 0)) {
    DSutil.swap(Heap, curr, parent(curr));
    curr = parent(curr);
  }
}
```

Figure 5.19 An implementation for the heap.

```
/** Heapify contents of Heap */
public void buildheap()
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }

/** Put element in its correct place */
private void siftdown(int pos) {
  assert (pos >= 0) && (pos < n) : "Illegal heap position";
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) && (Heap[j].compareTo(Heap[j+1]) < 0))
      j++; // j is now index of child with greater value
    if (Heap[pos].compareTo(Heap[j]) >= 0) return;
    DSutil.swap(Heap, pos, j);
    pos = j;  // Move down
  }
}

/** Remove and return maximum value */
public E removemax() {
  assert n > 0 : "Removing from empty heap";
  DSutil.swap(Heap, 0, --n); // Swap maximum with last value
  if (n != 0)      // Not on last element
    siftdown(0);   // Put new heap root val in correct place
  return Heap[n];
}

/** Remove and return element at specified position */
public E remove(int pos) {
  assert (pos >= 0) && (pos < n) : "Illegal heap position";
  if (pos == (n-1)) n--; // Last element, no work to be done
  else
  {
    DSutil.swap(Heap, pos, --n); // Swap with last value
    // If we just swapped in a big value, push it up
    while ((pos > 0) &&
           (Heap[pos].compareTo(Heap[parent(pos)]) > 0)) {
      DSutil.swap(Heap, pos, parent(pos));
      pos = parent(pos);
    }
    if (n != 0) siftdown(pos); // If it is little, push down
  }
  return Heap[n];
}
}
```

Figure 5.19 (continued)

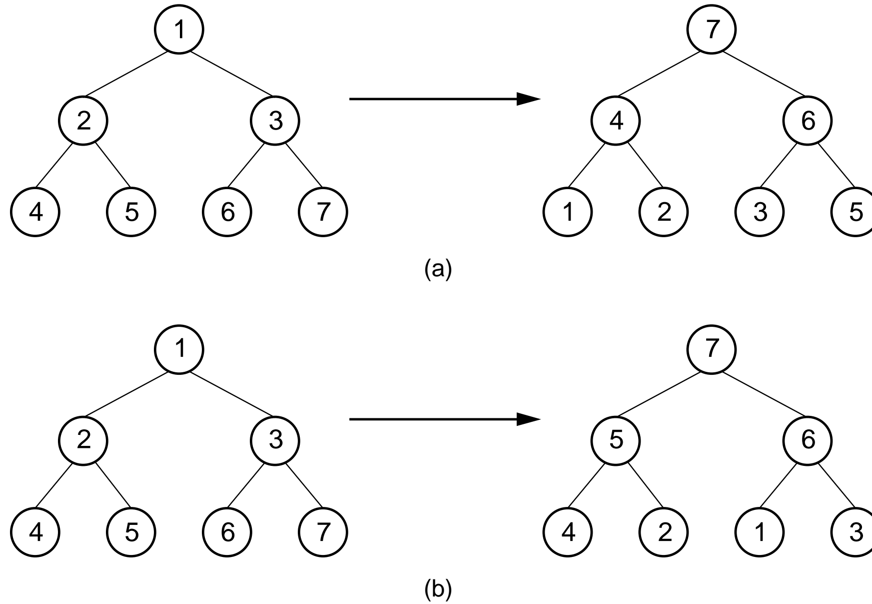


Figure 5.20 Two series of exchanges to build a max-heap. (a) This heap is built by a series of nine exchanges in the order (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6). (b) This heap is built by a series of four exchanges in the order (5-2), (7-3), (7-1), (6-1).

Each call to **`insert`** takes $\Theta(\log n)$ time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree. Thus, to insert n values into the heap, if we insert them one at a time, will take $\Theta(n \log n)$ time in the worst case.

If all n values are available at the beginning of the building process, we can build the heap faster than just inserting the values into the heap one by one. Consider Figure 5.20(a), which shows one series of exchanges that could be used to build the heap. All exchanges are between a node and one of its children. The heap is formed as a result of this exchange process. The array for the right-hand tree of Figure 5.20(a) would appear as follows:

7	4	6	1	2	3	5

Figure 5.20(b) shows an alternate series of exchanges that also forms a heap, but much more efficiently. The equivalent array representation would be

7	5	6	4	2	1	3

From this example, it is clear that the heap for any given set of numbers is not unique, and we see that some rearrangements of the input values require fewer exchanges than others to build the heap. So, how do we pick the best rearrangement?

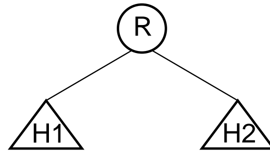


Figure 5.21 Final stage in the heap-building algorithm. Both subtrees of node *R* are heaps. All that remains is to push *R* down to its proper level in the heap.

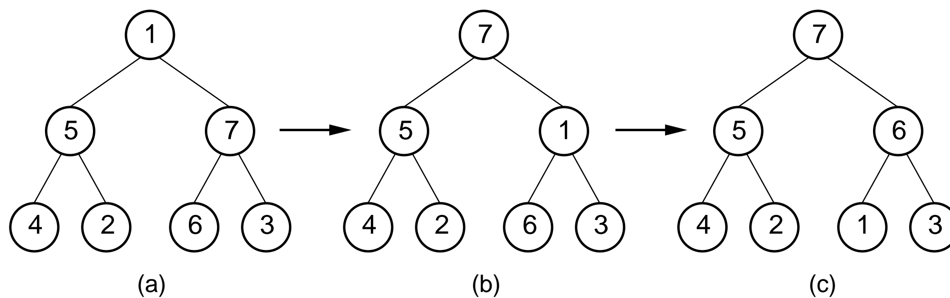


Figure 5.22 The siftdown operation. The subtrees of the root are assumed to be heaps. (a) The partially completed heap. (b) Values 1 and 7 are swapped. (c) Values 1 and 6 are swapped to form the final heap.

One good algorithm stems from induction. Suppose that the left and right subtrees of the root are already heaps, and *R* is the name of the element at the root. This situation is illustrated by Figure 5.21. In this case there are two possibilities. (1) *R* has a value greater than or equal to its two children. In this case, construction is complete. (2) *R* has a value less than one or both of its children. In this case, *R* should be exchanged with the child that has greater value. The result will be a heap, except that *R* might still be less than one or both of its (new) children. In this case, we simply continue the process of "pushing down" *R* until it reaches a level where it is greater than its children, or is a leaf node. This process is implemented by the private method **`siftdown`**. The siftdown operation is illustrated by Figure 5.22.

This approach assumes that the subtrees are already heaps, suggesting that a complete algorithm can be obtained by visiting the nodes in some order such that the children of a node are visited *before* the node itself. One simple way to do this is simply to work from the high index of the array to the low index. Actually, the build process need not visit the leaf nodes (they can never move down because they are already at the bottom), so the building algorithm can start in the middle of the array, with the first internal node. The exchanges shown in Figure 5.20(b) result from this process. Method **`buildHeap`** implements the building algorithm.

What is the cost of **`buildHeap`**? Clearly it is the sum of the costs for the calls to **`siftdown`**. Each **`siftdown`** operation can cost at most the number of levels it

takes for the node being sifted to reach the bottom of the tree. In any complete tree, approximately half of the nodes are leaves and so cannot be moved downward at all. One quarter of the nodes are one level above the leaves, and so their elements can move down at most one level. At each step up the tree we get half the number of nodes as were at the previous level, and an additional height of one. The maximum sum of total distances that elements can go is therefore

$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} = \frac{n}{2}\sum_{i=1}^{\log n}\frac{i-1}{2^{i-1}}.$$

From Equation 2.9 we know that this summation has a closed-form solution of approximately 2, so this algorithm takes $\Theta(n)$ time in the worst case. This is far better than building the heap one element at a time, which would cost $\Theta(n \log n)$ in the worst case. It is also faster than the $\Theta(n \log n)$ average-case time and $\Theta(n^2)$ worst-case time required to build the BST.

Removing the maximum (root) value from a heap containing n elements requires that we maintain the complete binary tree shape, and that the remaining $n-1$ node values conform to the heap property. We can maintain the proper shape by moving the element in the last position in the heap (the current last element in the array) to the root position. We now consider the heap to be one element smaller. Unfortunately, the new root value is probably *not* the maximum value in the new heap. This problem is easily solved by using **`siftdown`** to reorder the heap. Because the heap is $\log n$ levels deep, the cost of deleting the maximum element is $\Theta(\log n)$ in the average and worst cases.

The heap is a natural implementation for the priority queue discussed at the beginning of this section. Jobs can be added to the heap (using their priority value as the ordering key) when needed. Method **`removemax`** can be called whenever a new job is to be executed.

Some applications of priority queues require the ability to change the priority of an object already stored in the queue. This might require that the object's position in the heap representation be updated. Unfortunately, a max-heap is not efficient when searching for an arbitrary value; it is only good for finding the maximum value. However, if we already know the index for an object within the heap, it is a simple matter to update its priority (including changing its position to maintain the heap property) or remove it. The **`remove`** method takes as input the position of the node to be removed from the heap. A typical implementation for priority queues requiring updating of priorities will need to use an auxiliary data structure that supports efficient search for objects (such as a BST). Records in the auxiliary data structure will store the object's heap index, so that the object can be deleted from the heap and reinserted with its new priority (see Project 5.5). Sections 11.4.1 and 11.5.1 present applications for a priority queue with priority updating.

5.6 Huffman Coding Trees

The space/time tradeoff principle from Section 3.9 states that one can often gain an improvement in space requirements in exchange for a penalty in running time. There are many situations where this is a desirable tradeoff. A typical example is storing files on disk. If the files are not actively used, the owner might wish to compress them to save space. Later, they can be uncompressed for use, which costs some time, but only once.

We often represent a set of items in a computer program by assigning a unique code to each item. For example, the standard ASCII coding scheme assigns a unique eight-bit value to each character. It takes a certain minimum number of bits to provide unique codes for each character. For example, it takes $\lceil \log 128 \rceil$ or seven bits to provide the 128 unique codes needed to represent the 128 symbols of the ASCII character set.[5]

The requirement for $\lceil \log n \rceil$ bits to represent n unique code values assumes that all codes will be the same length, as are ASCII codes. This is called a **fixed-length** coding scheme. If all characters were used equally often, then a fixed-length coding scheme is the most space efficient method. However, you are probably aware that not all characters are used equally often in many applications. For example, the various letters in an English language document have greatly different frequencies of use.

Figure 5.23 shows the relative frequencies of the letters of the alphabet. From this table we can see that the letter 'E' appears about 60 times more often than the letter 'Z.' In normal ASCII, the words "DEED" and "MUCK" require the same amount of space (four bytes). It would seem that words such as "DEED," which are composed of relatively common letters, should be storable in less space than words such as "MUCK," which are composed of relatively uncommon letters.

If some characters are used more frequently than others, is it possible to take advantage of this fact and somehow assign them shorter codes? The price could be that other characters require longer codes, but this might be worthwhile if such characters appear rarely enough. This concept is at the heart of file compression techniques in common use today. The next section presents one such approach to assigning **variable-length** codes, called Huffman coding. While it is not commonly used in its simplest form for file compression (there are better methods), Huffman coding gives the flavor of such coding schemes. One motivation for studying Huffman coding is because it provides our first opportunity to see a type of tree structure referred to as a **search trie**.

[5]The ASCII standard is eight bits, not seven, even though there are only 128 characters represented. The eighth bit is used either to check for transmission errors, or to support "extended" ASCII codes with an additional 128 characters.

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

Figure 5.23 Relative frequencies for the 26 letters of the alphabet as they appear in a selected set of English documents. "Frequency" represents the expected frequency of occurrence per 1000 letters, ignoring case.

5.6.1 Building Huffman Coding Trees

Huffman coding assigns codes to characters such that the length of the code depends on the relative frequency or **weight** of the corresponding character. Thus, it is a variable-length code. If the estimated frequencies for letters match the actual frequency found in an encoded message, then the length of that message will typically be less than if a fixed-length code had been used. The Huffman code for each letter is derived from a full binary tree called the **Huffman coding tree**, or simply the **Huffman tree**. Each leaf of the Huffman tree corresponds to a letter, and we define the weight of the leaf node to be the weight (frequency) of its associated letter. The goal is to build a tree with the **minimum external path weight**. Define the **weighted path length** of a leaf to be its weight times its depth. The binary tree with minimum external path weight is the one with the minimum sum of weighted path lengths for the given set of leaves. A letter with high weight should have low depth, so that it will count the least against the total path length. As a result, another letter might be pushed deeper in the tree if it has less weight.

The process of building the Huffman tree for n letters is quite simple. First, create a collection of n initial Huffman trees, each of which is a single leaf node containing one of the letters. Put the n partial trees onto a priority queue organized by weight (frequency). Next, remove the first two trees (the ones with lowest weight) from the priority queue. Join these two trees together to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two trees. Put this new tree back into the priority queue. This process is repeated until all of the partial Huffman trees have been combined into one.

Letter	C	D	E	K	L	M	U	Z
Frequency	32	42	120	7	42	24	37	2

Figure 5.24 The relative frequencies for eight selected letters.

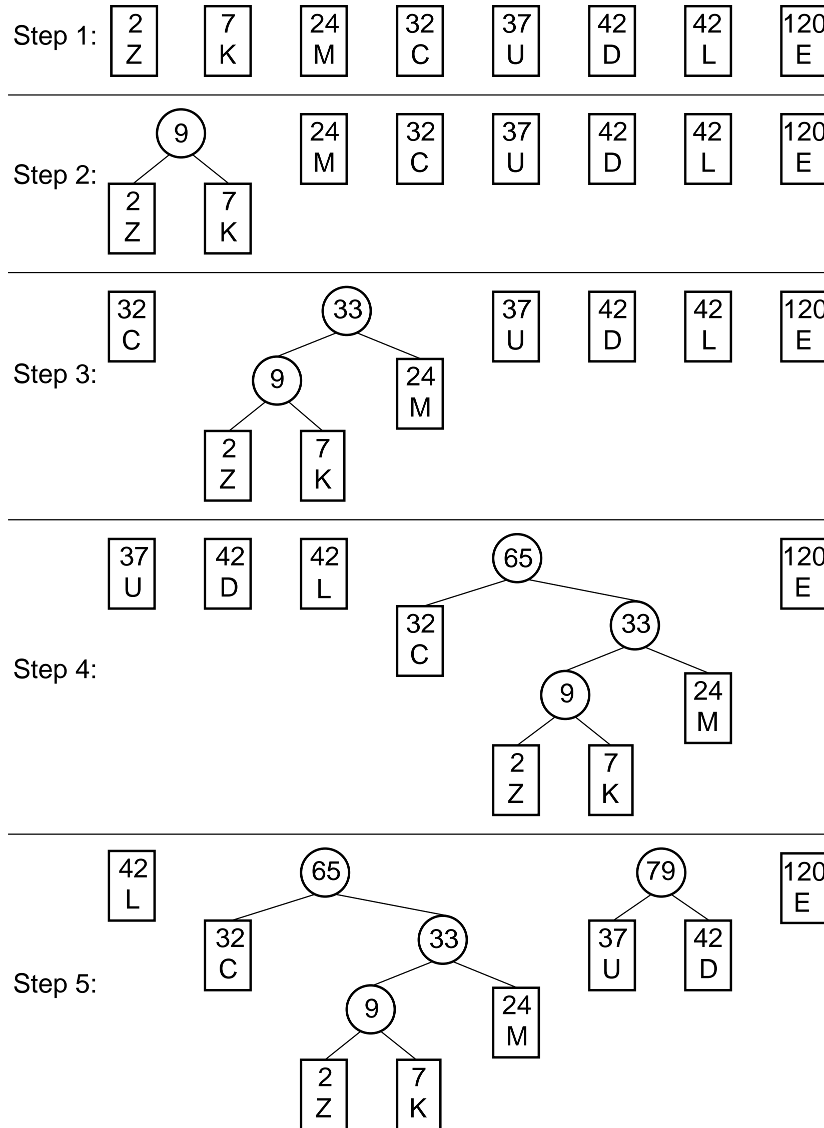


Figure 5.25 The first five steps of the building process for a sample Huffman tree.

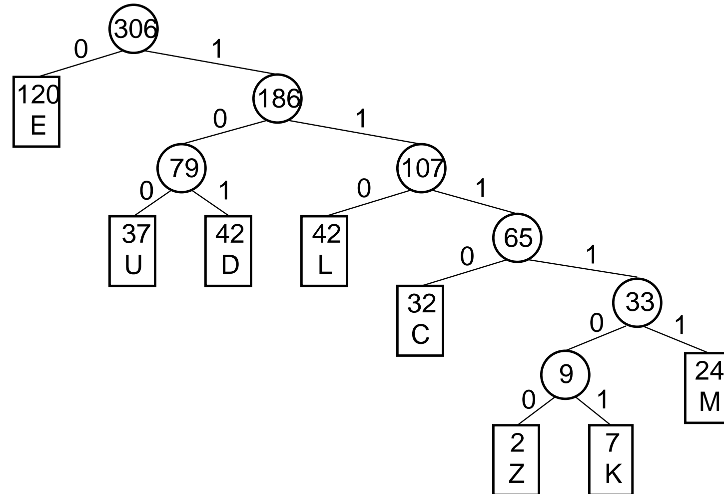


Figure 5.26 A Huffman tree for the letters of Figure 5.24.

Example 5.8 Figure 5.25 illustrates part of the Huffman tree construction process for the eight letters of Figure 5.24. Ranking D and L arbitrarily by alphabetical order, the letters are ordered by frequency as

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Because the first two letters on the list are Z and K, they are selected to be the first trees joined together.[6] They become the children of a root node with weight 9. Thus, a tree whose root has weight 9 is placed back on the list, where it takes up the first position. The next step is to take values 9 and 24 off the list (corresponding to the partial tree with two leaf nodes built in the last step, and the partial tree storing the letter M, respectively) and join them together. The resulting root node has weight 33, and so this tree is placed back into the list. Its priority will be between the trees with values 32 (for letter C) and 37 (for letter U). This process continues until a tree whose root has weight 306 is built. This tree is shown in Figure 5.26.

Figure 5.27 shows an implementation for Huffman tree nodes. This implementation is similar to the **`VarBinNode`** implementation of Figure 5.10. There is an abstract base class, named **`HuffNode`**, and two subclasses, named **`LeafNode`**

[6]For clarity, the examples for building Huffman trees show a sorted list to keep the letters ordered by frequency. But a real implementation would use a heap to implement the priority queue for efficiency.

```
/** Huffman tree node implementation: Base class */
public interface HuffBaseNode<E> {
  public boolean isLeaf();
  public int weight();
}

/** Huffman tree node: Leaf class */
class HuffLeafNode<E> implements HuffBaseNode<E> {
  private E element;          // Element for this node
  private int weight;         // Weight for this node

  /** Constructor */
  public HuffLeafNode(E el, int wt)
    { element = el; weight = wt; }

  /** @return The element value */
  public E element() { return element; }

  /** @return The weight */
  public int weight() { return weight; }

  /** Return true */
  public boolean isLeaf() { return true; }
}

/** Huffman tree node: Internal class */
class HuffInternalNode<E> implements HuffBaseNode<E> {
  private int weight;             // Weight (sum of children)
  private HuffBaseNode<E> left;  // Pointer to left child
  private HuffBaseNode<E> right; // Pointer to right child

  /** Constructor */
  public HuffInternalNode(HuffBaseNode<E> l,
                          HuffBaseNode<E> r, int wt)
    { left = l; right = r; weight = wt; }

  /** @return The left child */
  public HuffBaseNode<E> left() { return left; }

  /** @return The right child */
  public HuffBaseNode<E> right() { return right; }

  /** @return The weight */
  public int weight() { return weight; }

  /** Return false */
  public boolean isLeaf() { return false; }
}
```

Figure 5.27 Implementation for Huffman tree nodes. Internal nodes and leaf nodes are represented by separate classes, each derived from an abstract base class.

```
/** A Huffman coding tree */
class HuffTree<E> implements Comparable<HuffTree<E>>{
  private HuffBaseNode<E> root;  // Root of the tree

  /** Constructors */
  public HuffTree(E el, int wt)
    { root = new HuffLeafNode<E>(el, wt); }
  public HuffTree(HuffBaseNode<E> l,
                  HuffBaseNode<E> r, int wt)
    { root = new HuffInternalNode<E>(l, r, wt); }

  public HuffBaseNode<E> root() { return root; }
  public int weight() // Weight of tree is weight of root
    { return root.weight(); }
  public int compareTo(HuffTree<E> that) {
    if (root.weight() < that.weight()) return -1;
    else if (root.weight() == that.weight()) return 0;
    else return 1;
  }
}
```

Figure 5.28 Class declarations for the Huffman tree.

and `IntlNode`. This implementation reflects the fact that leaf and internal nodes contain distinctly different information.

Figure 5.28 shows the implementation for the Huffman tree. Figure 5.29 shows the Java code for the tree-building process.

Huffman tree building is an example of a **greedy algorithm**. At each step, the algorithm makes a "greedy" decision to merge the two subtrees with least weight. This makes the algorithm simple, but does it give the desired result? This section concludes with a proof that the Huffman tree indeed gives the most efficient arrangement for the set of letters. The proof requires the following lemma.

Lemma 5.1 *For any Huffman tree built by function* `buildHuff` *containing at least two letters, the two letters with least frequency are stored in siblings nodes whose depth is at least as deep as any other leaf nodes in the tree.*

Proof: Call the two letters with least frequency l_1 and l_2. They must be siblings because `buildHuff` selects them in the first step of the construction process. Assume that l_1 and l_2 are not the deepest nodes in the tree. In this case, the Huffman tree must either look as shown in Figure 5.30, or in some sense be symmetrical to this. For this situation to occur, the parent of l_1 and l_2, labeled *V*, must have greater weight than the node labeled *X*. Otherwise, function `buildHuff` would have selected node *V* in place of node *X* as the child of node *U*. However, this is impossible because l_1 and l_2 are the letters with least frequency. □

Theorem 5.3 *Function* `buildHuff` *builds the Huffman tree with the minimum external path weight for the given set of letters.*

```
/** Build a Huffman tree from list hufflist */
static HuffTree<Character> buildTree() {
  HuffTree tmp1, tmp2, tmp3 = null;

  while (Hheap.heapsize() > 1) { // While two items left
    tmp1 = Hheap.removemin();
    tmp2 = Hheap.removemin();
    tmp3 = new HuffTree<Character>(tmp1.root(), tmp2.root(),
                                   tmp1.weight() + tmp2.weight());
    Hheap.insert(tmp3);   // Return new tree to heap
  }
  return tmp3;            // Return the tree
}
```

Figure 5.29 Implementation for the Huffman tree construction function. **`buildHuff`** takes as input **`fl`**, the min-heap of partial Huffman trees, which initially are single leaf nodes as shown in Step 1 of Figure 5.25. The body of function **`buildTree`** consists mainly of a **`for`** loop. On each iteration of the **`for`** loop, the first two partial trees are taken off the heap and placed in variables **`temp1`** and **`temp2`**. A tree is created (**`temp3`**) such that the left and right subtrees are **`temp1`** and **`temp2`**, respectively. Finally, **`temp3`** is returned to **`fl`**.

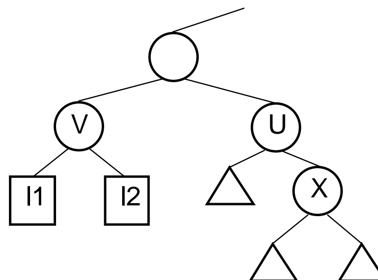


Figure 5.30 An impossible Huffman tree, showing the situation where the two nodes with least weight, l_1 and l_2, are not the deepest nodes in the tree. Triangles represent subtrees.

Proof: The proof is by induction on n, the number of letters.

- **Base Case**: For $n = 2$, the Huffman tree must have the minimum external path weight because there are only two possible trees, each with identical weighted path lengths for the two leaves.
- **Induction Hypothesis**: Assume that any tree created by **`buildHuff`** that contains $n - 1$ leaves has minimum external path length.
- **Induction Step**: Given a Huffman tree **T** built by **`buildHuff`** with n leaves, $n \geq 2$, suppose that $w_1 \leq w_2 \leq \cdots \leq w_n$ where w_1 to w_n are the weights of the letters. Call *V* the parent of the letters with frequencies w_1 and w_2. From the lemma, we know that the leaf nodes containing the letters with frequencies w_1 and w_2 are as deep as any nodes in **T**. If any other leaf

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

Figure 5.31 The Huffman codes for the letters of Figure 5.24.

nodes in the tree were deeper, we could reduce their weighted path length by swapping them with w_1 or w_2. But the lemma tells us that no such deeper nodes exist. Call $\mathbf{T}'$ the Huffman tree that is identical to $\mathbf{T}$ except that node V is replaced with a leaf node V' whose weight is $w_1 + w_2$. By the induction hypothesis, $\mathbf{T}'$ has minimum external path length. Returning the children to V' restores tree $\mathbf{T}$, which must also have minimum external path length.

Thus by mathematical induction, function **buildHuff** creates the Huffman tree with minimum external path length. □

5.6.2 Assigning and Using Huffman Codes

Once the Huffman tree has been constructed, it is an easy matter to assign codes to individual letters. Beginning at the root, we assign either a '0' or a '1' to each edge in the tree. '0' is assigned to edges connecting a node with its left child, and '1' to edges connecting a node with its right child. This process is illustrated by Figure 5.26. The Huffman code for a letter is simply a binary number determined by the path from the root to the leaf corresponding to that letter. Thus, the code for E is '0' because the path from the root to the leaf node for E takes a single left branch. The code for K is '111101' because the path to the node for K takes four right branches, then a left, and finally one last right. Figure 5.31 lists the codes for all eight letters.

Given codes for the letters, it is a simple matter to use these codes to encode a text message. We simply replace each letter in the string with its binary code. A lookup table can be used for this purpose.

Example 5.9 Using the code generated by our example Huffman tree, the word "DEED" is represented by the bit string "10100101" and the word "MUCK" is represented by the bit string "111111001110111101."

Decoding the message is done by looking at the bits in the coded string from left to right until a letter is decoded. This can be done by using the Huffman tree in

a reverse process from that used to generate the codes. Decoding a bit string begins at the root of the tree. We take branches depending on the bit value — left for '0' and right for '1' — until reaching a leaf node. This leaf contains the first character in the message. We then process the next bit in the code restarting at the root to begin the next character.

Example 5.10 To decode the bit string "1011001110111101" we begin at the root of the tree and take a right branch for the first bit which is '1.' Because the next bit is a '0' we take a left branch. We then take another right branch (for the third bit '1'), arriving at the leaf node corresponding to the letter D. Thus, the first letter of the coded word is D. We then begin again at the root of the tree to process the fourth bit, which is a '1.' Taking a right branch, then two left branches (for the next two bits which are '0'), we reach the leaf node corresponding to the letter U. Thus, the second letter is U. In similar manner we complete the decoding process to find that the last two letters are C and K, spelling the word "DUCK."

A set of codes is said to meet the **prefix property** if no code in the set is the prefix of another. The prefix property guarantees that there will be no ambiguity in how a bit string is decoded. In other words, once we reach the last bit of a code during the decoding process, we know which letter it is the code for. Huffman codes certainly have the prefix property because any prefix for a code would correspond to an internal node, while all codes correspond to leaf nodes. For example, the code for M is '11111.' Taking five right branches in the Huffman tree of Figure 5.26 brings us to the leaf node containing M. We can be sure that no letter can have code '111' because this corresponds to an internal node of the tree, and the tree-building process places letters only at the leaf nodes.

How efficient is Huffman coding? In theory, it is an optimal coding method whenever the true frequencies are known, and the frequency of a letter is independent of the context of that letter in the message. In practice, the frequencies of letters in an English text document do change depending on context. For example, while E is the most commonly used letter of the alphabet in English documents, T is more common as the first letter of a word. This is why most commercial compression utilities do not use Huffman coding as their primary coding method, but instead use techniques that take advantage of the context for the letters.

Another factor that affects the compression efficiency of Huffman coding is the relative frequencies of the letters. Some frequency patterns will save no space as compared to fixed-length codes; others can result in great compression. In general, Huffman coding does better when there is large variation in the frequencies of letters. In the particular case of the frequencies shown in Figure 5.31, we can

determine the expected savings from Huffman coding if the actual frequencies of a coded message match the expected frequencies.

Example 5.11 Because the sum of the frequencies in Figure 5.31 is 306 and E has frequency 120, we expect it to appear 120 times in a message containing 306 letters. An actual message might or might not meet this expectation. Letters D, L, and U have code lengths of three, and together are expected to appear 121 times in 306 letters. Letter C has a code length of four, and is expected to appear 32 times in 306 letters. Letter M has a code length of five, and is expected to appear 24 times in 306 letters. Finally, letters K and Z have code lengths of six, and together are expected to appear only 9 times in 306 letters. The average expected cost per character is simply the sum of the cost for each character (c_i) times the probability of its occurring (p_i), or

$$c_1 p_1 + c_2 p_2 + \cdots + c_n p_n.$$

This can be reorganized as

$$\frac{c_1 f_1 + c_2 f_2 + \cdots + c_n f_n}{f_T}$$

where f_i is the (relative) frequency of letter i and f_T is the total for all letter frequencies. For this set of frequencies, the expected cost per letter is

$$[(1\times120)+(3\times121)+(4\times32)+(5\times24)+(6\times9)]/306 = 785/306 \approx 2.57$$

A fixed-length code for these eight characters would require $\log 8 = 3$ bits per letter as opposed to about 2.57 bits per letter for Huffman coding. Thus, Huffman coding is expected to save about 14% for this set of letters.

Huffman coding for all ASCII symbols should do better than this. The letters of Figure 5.31 are atypical in that there are too many common letters compared to the number of rare letters. Huffman coding for all 26 letters would yield an expected cost of 4.29 bits per letter. The equivalent fixed-length code would require about five bits. This is somewhat unfair to fixed-length coding because there is actually room for 32 codes in five bits, but only 26 letters. More generally, Huffman coding of a typical text file will save around 40% over ASCII coding if we charge ASCII coding at eight bits per character. Huffman coding for a binary file (such as a compiled executable) would have a very different set of distribution frequencies and so would have a different space savings. Most commercial compression programs use two or three coding schemes to adjust to different types of files.

In the preceding example, "DEED" was coded in 8 bits, a saving of 33% over the twelve bits required from a fixed-length coding. However, "MUCK" requires

18 bits, more space than required by the corresponding fixed-length coding. The problem is that "MUCK" is composed of letters that are not expected to occur often. If the message does not match the expected frequencies of the letters, than the length of the encoding will not be as expected either.

5.6.3 Search in Huffman Trees

When we decode a character using the Huffman coding tree, we follow a path through the tree dictated by the bits in the code string. Each '0' bit indicates a left branch while each '1' bit indicates a right branch. Now look at Figure 5.26 and consider this structure in terms of searching for a given letter (whose key value is its Huffman code). We see that all letters with codes beginning with '0' are stored in the left branch, while all letters with codes beginning with '1' are stored in the right branch. Contrast this with storing records in a BST. There, all records with key value less than the root value are stored in the left branch, while all records with key values greater than the root are stored in the right branch.

If we view all records stored in either of these structures as appearing at some point on a number line representing the key space, we can see that the splitting behavior of these two structures is very different. The BST splits the space based on the key values as they are encountered when going down the tree. But the splits in the key space are predetermined for the Huffman tree. Search tree structures whose splitting points in the key space are predetermined are given the special name **trie** to distinguish them from the type of search tree (like the BST) whose splitting points are determined by the data. Tries are discussed in more detail in Chapter 13.

5.7 Further Reading

See Shaffer and Brown [SB93] for an example of a tree implementation where an internal node pointer field stores the value of its child instead of a pointer to its child when the child is a leaf node.

Many techniques exist for maintaining reasonably balanced BSTs in the face of an unfriendly series of insert and delete operations. One example is the AVL tree of Adelson-Velskii and Landis, which is discussed by Knuth [Knu98]. The AVL tree (see Section 13.2) is actually a BST whose insert and delete routines reorganize the tree structure so as to guarantee that the subtrees rooted by the children of any node will differ in height by at most one. Another example is the splay tree [ST85], also discussed in Section 13.2.

See Bentley's Programming Pearl "Thanks, Heaps" [Ben85, Ben88] for a good discussion on the heap data structure and its uses.

The proof of Section 5.6.1 that the Huffman coding tree has minimum external path weight is from Knuth [Knu97]. For more information on data compression

techniques, see *Managing Gigabytes* by Witten, Moffat, and Bell [WMB99], and *Codes and Cryptography* by Dominic Welsh [Wel88]. Tables 5.23 and 5.24 are derived from Welsh [Wel88].

5.8 Exercises

5.1 Section 5.1.1 claims that a full binary tree has the highest number of leaf nodes among all trees with n internal nodes. Prove that this is true.

5.2 Define the **degree** of a node as the number of its non-empty children. Prove by induction that the number of degree 2 nodes in any binary tree is one less than the number of leaves.

5.3 Define the **internal path length** for a tree as the sum of the depths of all internal nodes, while the **external path length** is the sum of the depths of all leaf nodes in the tree. Prove by induction that if tree **T** is a full binary tree with n internal nodes, I is **T**'s internal path length, and E is **T**'s external path length, then $E = I + 2n$ for $n \geq 0$.

5.4 Explain why function `preorder2` from Section 5.2 makes half as many recursive calls as function `preorder`. Explain why it makes twice as many accesses to left and right children.

5.5 **(a)** Modify the preorder traversal of Section 5.2 to perform an inorder traversal of a binary tree.

(b) Modify the preorder traversal of Section 5.2 to perform a postorder traversal of a binary tree.

5.6 Write a recursive function named `search` that takes as input the pointer to the root of a binary tree (*not* a BST!) and a value K, and returns `true` if value K appears in the tree and `false` otherwise.

5.7 Write an algorithm that takes as input the pointer to the root of a binary tree and prints the node values of the tree in **level** order. Level order first prints the root, then all nodes of level 1, then all nodes of level 2, and so on. *Hint*: Preorder traversals make use of a stack through recursive calls. Consider making use of another data structure to help implement the level-order traversal.

5.8 Write a recursive function that returns the height of a binary tree.

5.9 Write a recursive function that returns a count of the number of leaf nodes in a binary tree.

5.10 Assume that a given binary tree stores integer values in its nodes. Write a recursive function that sums the values of all nodes in the tree.

5.11 Assume that a given binary tree stores integer values in its nodes. Write a recursive function that traverses a binary tree, and prints the value of every node who's grandparent has a value that is a multiple of five.

5.12 Write a recursive function that traverses a binary tree, and prints the value of every node which has at least four great-grandchildren.

5.13 Compute the overhead fraction for each of the following full binary tree implementations.

(a) All nodes store data, two child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.

(b) All nodes store data and two child pointers. The data field requires sixteen bytes and each pointer requires four bytes.

(c) All nodes store data and a parent pointer, and internal nodes store two child pointers. The data field requires eight bytes and each pointer requires four bytes.

(d) Only leaf nodes store data; internal nodes store two child pointers. The data field requires eight bytes and each pointer requires four bytes.

5.14 Why is the BST Property defined so that nodes with values equal to the value of the root appear only in the right subtree, rather than allow equal-valued nodes to appear in either subtree?

5.15 **(a)** Show the BST that results from inserting the values 15, 20, 25, 18, 16, 5, and 7 (in that order).

(b) Show the enumerations for the tree of (a) that result from doing a preorder traversal, an inorder traversal, and a postorder traversal.

5.16 Draw the BST that results from adding the value 5 to the BST shown in Figure 5.13(a).

5.17 Draw the BST that results from deleting the value 7 from the BST of Figure 5.13(b).

5.18 Write a function that prints out the node values for a BST in sorted order from highest to lowest.

5.19 Write a recursive function named `smallcount` that, given the pointer to the root of a BST and a key K, returns the number of nodes having key values less than or equal to K. Function `smallcount` should visit as few nodes in the BST as possible.

5.20 Write a recursive function named `printRange` that, given the pointer to the root of a BST, a low key value, and a high key value, prints in sorted order all records whose key values fall between the two given keys. Function `printRange` should visit as few nodes in the BST as possible.

5.21 Write a recursive function named `checkBST` that, given the pointer to the root of a binary tree, will return `true` if the tree is a BST, and `false` if it is not.

5.22 Describe a simple modification to the BST that will allow it to easily support finding the Kth smallest value in $\Theta(\log n)$ average case time. Then write a pseudo-code function for finding the Kth smallest value in your modified BST.

5.23 What are the minimum and maximum number of elements in a heap of height h?

5.24 Where in a max-heap might the smallest element reside?

5.25 Show the max-heap that results from running **`buildHeap`** on the following values stored in an array:

$$10 \quad 5 \quad 12 \quad 3 \quad 2 \quad 1 \quad 8 \quad 7 \quad 9 \quad 4$$

5.26 **(a)** Show the heap that results from deleting the maximum value from the max-heap of Figure 5.20b.

(b) Show the heap that results from deleting the element with value 5 from the max-heap of Figure 5.20b.

5.27 Revise the heap definition of Figure 5.19 to implement a min-heap. The member function **`removemax`** should be replaced by a new function called **`removemin`**.

5.28 Build the Huffman coding tree and determine the codes for the following set of letters and weights:

Letter	A	B	C	D	E	F	G	H	I	J	K	L
Frequency	2	3	5	7	11	13	17	19	23	31	37	41

What is the expected length in bits of a message containing n characters for this frequency distribution?

5.29 What will the Huffman coding tree look like for a set of sixteen characters all with equal weight? What is the average code length for a letter in this case? How does this differ from the smallest possible fixed length code for sixteen characters?

5.30 A set of characters with varying weights is assigned Huffman codes. If one of the characters is assigned code 001, then,

(a) Describe all codes that *cannot* have been assigned.

(b) Describe all codes that *must* have been assigned.

5.31 Assume that a sample alphabet has the following weights:

Letter	Q	Z	F	M	T	S	O	E
Frequency	2	3	10	10	10	15	20	30

(a) For this alphabet, what is the worst-case number of bits required by the Huffman code for a string of n letters? What string(s) have the worst-case performance?

(b) For this alphabet, what is the best-case number of bits required by the Huffman code for a string of n letters? What string(s) have the best-case performance?

(c) What is the average number of bits required by a character using the Huffman code for this alphabet?

5.32 You must keep track of some data. Your options are:

(1) A linked-list maintained in sorted order.
(2) A linked-list of unsorted records.
(3) A binary search tree.
(4) An array-based list maintained in sorted order.
(5) An array-based list of unsorted records.

For each of the following scenarios, which of these choices would be best? Explain your answer.

(a) The records are guaranteed to arrive already sorted from lowest to highest (i.e., whenever a record is inserted, its key value will always be greater than that of the last record inserted). A total of 1000 inserts will be interspersed with 1000 searches.
(b) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1,000,000 insertions are performed, followed by 10 searches.
(c) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1000 insertions are interspersed with 1000 searches.
(d) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1000 insertions are performed, followed by 1,000,000 searches.

5.9 Projects

5.1 Re-implement the composite design for the binary tree node class of Figure 5.11 using a flyweight in place of `null` pointers to empty nodes.

5.2 One way to deal with the "problem" of `null` pointers in binary trees is to use that space for some other purpose. One example is the **threaded** binary tree. Extending the node implementation of Figure 5.7, the threaded binary tree stores with each node two additional bit fields that indicate if the child pointers `lc` and `rc` are regular pointers to child nodes or threads. If `lc` is not a pointer to a non-empty child (i.e., if it would be `null` in a regular binary tree), then it instead stores a pointer to the **inorder predecessor** of that node. The inorder predecessor is the node that would be printed immediately before the current node in an inorder traversal. If `rc` is not a pointer to a child, then it instead stores a pointer to the node's **inorder successor**. The inorder successor is the node that would be printed immediately after the current node in an inorder traversal. The main advantage of threaded binary

trees is that operations such as inorder traversal can be implemented without using recursion or a stack.

Re-implement the BST as a threaded binary tree, and include a non-recursive version of the preorder traversal

5.3 Implement a city database using a BST to store the database records. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x- and y-coordinates. The BST should be organized by city name. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. Another operation that should be supported is to print all records within a given distance of a specified point. Collect running-time statistics for each operation. Which operations can be implemented reasonably efficiently (i.e., in $\Theta(\log n)$ time in the average case) using a BST? Can the database system be made more efficient by using one or more additional BSTs to organize the records by location?

5.4 Create a binary tree ADT that includes generic traversal methods that take a visitor, as described in Section 5.2. Write functions **`count`** and **`BSTcheck`** of Section 5.2 as visitors to be used with the generic traversal method.

5.5 Implement a priority queue class based on the max-heap class implementation of Figure 5.19. The following methods should be supported for manipulating the priority queue:

```
void enqueue(int ObjectID, int priority);
int dequeue();
void changeweight(int ObjectID, int newPriority);
```

Method **`enqueue`** inserts a new object into the priority queue with ID number **`ObjectID`** and priority **`priority`**. Method **`dequeue`** removes the object with highest priority from the priority queue and returns its object ID. Method **`changeweight`** changes the priority of the object with ID number **`ObjectID`** to be **`newPriority`**. The type for **`E`** should be a class that stores the object ID and the priority for that object. You will need a mechanism for finding the position of the desired object within the heap. Use an array, storing the object with **`ObjectID`** i in position i. (Be sure in your testing to keep the **`ObjectID`**s within the array bounds.) You must also modify the heap implementation to store the object's position in the auxiliary array so that updates to objects in the heap can be updated as well in the array.

5.6 The Huffman coding tree function **`buildHuff`** of Figure 5.29 manipulates a sorted list. This could result in a $\Theta(n^2)$ algorithm, because placing an intermediate Huffman tree on the list could take $\Theta(n)$ time. Revise this algorithm to use a priority queue based on a min-heap instead of a list.

5.7 Complete the implementation of the Huffman coding tree, building on the code presented in Section 5.6. Include a function to compute and store in a table the codes for each letter, and functions to encode and decode messages. This project can be further extended to support file compression. To do so requires adding two steps: (1) Read through the input file to generate actual frequencies for all letters in the file; and (2) store a representation for the Huffman tree at the beginning of the encoded output file to be used by the decoding function. If you have trouble with devising such a representation, see Section 6.5.