Chapter 12

Binary Trees

Concepts:
- ▷ Binary trees
- ▷ Tree traversals
- ▷ Recursion

I think that I shall never see
A poem lovely as a binary tree.
—Bill Amend as Jason Fox

RECURSION IS A BEAUTIFUL APPROACH TO STRUCTURING. We commonly think of recursion as a form of structuring the *control* of programs, but self-reference can be used just as effectively in the structuring of program *data*. In this chapter, we investigate the use of recursion in describing branching structures called *trees*.

Most of the structures we have already investigated are *linear*—their natural presentation is in a line. Trees branch. The result is that where there is an inherent ordering in linear structures, we find choices in the way we order the elements of a tree. These choices are an indication of the reduced "friction" of the structure and, as a result, trees provide us with the fastest ways to solve many problems.

Before we investigate the implementation of trees, we must develop a concise terminology.

12.1 Terminology

A tree is a collection of elements, called *nodes*, and relations between them, called *edges*. Usually, data are stored within the nodes of a tree. Two trees are *disjoint* if no node or edge is found common to both. A *trivial tree* has no nodes and thus no data. An isolated node is also a tree.

From these primitives we may recursively construct more complex trees. Let r be a new node and let $T_1, T_2, \ldots, T_n$ be a (possibly empty) set—a *forest*—of distinct trees. A new tree is constructed by making r the root of the tree, and establishing an edge between r and the root of each tree, T_i, in the forest. We refer to the trees, T_i, as *subtrees*. We draw trees with the root above and the trees below. Figure 12.1g is an aid to understanding this construction.

The *parent* of a node is the adjacent node appearing above it (see Figure 12.2). The *root* of a tree is the unique node with no parent. The *ancestors* of a node n are the roots of trees containing n: n, n's parent, n's parent's parent,

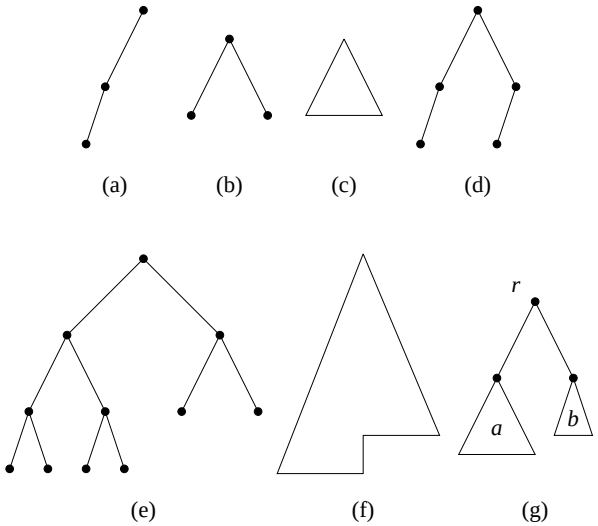


Figure 12.1 Examples of trees. Trees (a) and (b) are three-node trees. Trees are sometimes symbolized abstractly, as in (c). Tree (b) is *full*, but (d) is not. Tree (e) is not full but is *complete*. Complete trees are symbolized as in (f). Abstract tree (g) has root r and subtrees (a) and (b).

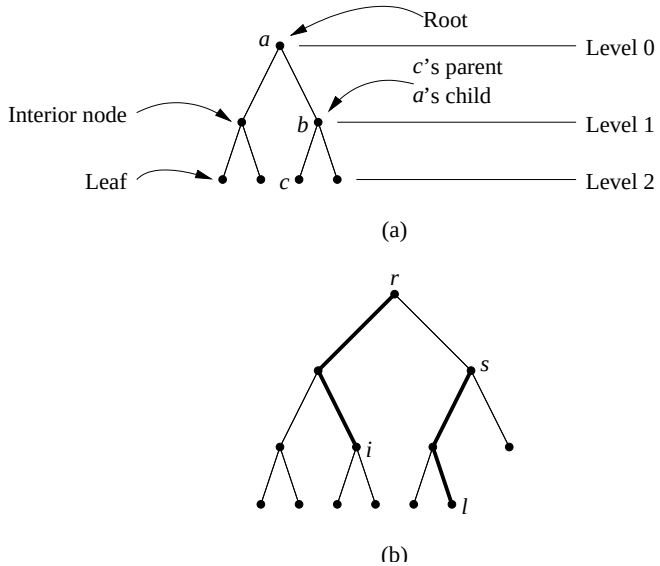


Figure 12.2 Anatomy of trees. (a) A full (and complete) tree. (b) A complete tree that is not full. Here, the unique path from node i to root r is bold: i has depth 2. Also, indicated in bold is a longest path from s to a leaf l: s has height 2 and depth 1. The subtree rooted at s has 5 nodes.

---

and so on. The root is the ancestor shared by every node in the tree. A *child* of a node n is any node that has n as its parent. The *descendants* of a node n are those nodes that have n as an ancestor. A *leaf* is a node with no children. Note that n is its own ancestor and descendant. A node m is the *proper ancestor* (*proper descendant*) of a node n if m is an ancestor (descendant) of n, but not vice versa. In a tree T, the descendants of n form the *subtree* of T rooted at n. Any node of a tree T that is not a leaf is an *interior node*. Roots can be interior nodes. Nodes m and n are *siblings* if they share a parent.

A *path* is the unique shortest sequence of edges from a node n to an ancestor. The *length* of a path is the number of edges it mentions. The *height of a node* n in a tree is the length of any longest path between a leaf and n. The *height of a tree* is the height of its root. This is the maximum height of any node in the tree. The *depth* (or *level*) of a node n in its tree T is the length of the path from n to T's root. The sum of a node's depth and height is no greater than the height of the tree. The *degree of a node* n is the number of its children. The *degree of a tree* (or its *arity*) is the maximum degree of any of its nodes. A *binary tree* is a tree with arity less than or equal to 2. A 1-ary binary tree is termed *degenerate*. A node n in a binary tree is *full* if it has degree 2. In an *oriented tree* we will call one child the *left child* and the other the *right child*. A *full binary tree* of height h

has leaves only on level h, and each of its internal nodes is full. The addition of a node to a full binary tree causes its height to increase. A *complete binary tree* of height h is a full binary tree with 0 or more of the rightmost leaves of level h removed.

12.2 Example: Pedigree Charts

With the growth of the Internet, many people have been able to make contact with long-lost ancestors, not through some new technology that allows contact with spirits, but through genealogical databases. One of the reasons that genealogy has been so successful on computers is that computers can organize treelike data more effectively than people.

One such organizational approach is a pedigree chart. This is little more than a binary tree of the relatives of an individual. The root is an individual, perhaps yourself, and the two subtrees are the pedigrees of your mother and father.[1] They, of course, have two sets of parents, with pedigrees that are rooted at your grandparents.

Steve points out: relationships in these trees is upside down!

To demonstrate how we might make use of a `BinaryTree` class, we might imagine the following code that develops the pedigree for someone named George Bush:[2]

Pedigree

```
// ancestors of George H. W. Bush
// indentation is provided to aid in understanding relations
   BinaryTree<String> JSBush = new BinaryTree<String>("Rev. James");
   BinaryTree<String> HEFay = new BinaryTree<String>("Harriet");
  BinaryTree<String> SPBush = new BinaryTree<String>("Samuel",JSBush,HEFay);

   BinaryTree<String> RESheldon = new BinaryTree<String>("Robert");
    BinaryTree<String> MEButler = new BinaryTree<String>("Mary");
   BinaryTree<String> FSheldon = new BinaryTree<String>("Flora",RESheldon,MEButler);

 BinaryTree<String> PSBush = new BinaryTree<String>("Prescott",SPBush,FSheldon);

   BinaryTree<String> DDWalker = new BinaryTree<String>("David");
   BinaryTree<String> MABeaky = new BinaryTree<String>("Martha");
  BinaryTree<String> GHWalker = new BinaryTree<String>("George",DDWalker,MABeaky);

   BinaryTree<String> JHWear = new BinaryTree<String>("James II");
   BinaryTree<String> NEHolliday = new BinaryTree<String>("Nancy");
  BinaryTree<String> LWear = new BinaryTree<String>("Lucretia",JHWear,NEHolliday);

 BinaryTree<String> DWalker = new BinaryTree<String>("Dorothy",GHWalker,LWear);
```

[1] At the time of this writing, modern technology has not advanced to the point of allowing nodes of degree other than 2.

[2] This is the Texan born in Massachusetts; the other Texan was born in Connecticut.

```
BinaryTree<String> GHWBush = new BinaryTree<String>("George",PSBush,DWalker);
```

For each person we develop a node that either has no links (the parents were not included in the database) or has references to other pedigrees stored as `BinaryTrees`. Arbitrarily, we choose to maintain the father's pedigree on the left side and the mother's pedigree along the right. We can then answer simple questions about ancestry by examining the structure of the tree. For example, who are the direct female relatives of the President?

```
// Question: What are George H. W. Bush's ancestors' names,
//   following the mother's side?
BinaryTree<String> person = GHWBush;
while (!person.right().isEmpty())
{
    person = person.right();    // right branch is mother
    System.out.println(person.value()); // value is name
}
```

The results are

```
Dorothy
Lucretia
Nancy
```

Exercise 12.1 *These are, of course, only some of the female relatives of President Bush. Write a program that prints* all *the female names found in a* `BinaryTree` *representing a pedigree chart.*

One feature that would be useful, would be the ability to add branches to a tree after the tree was constructed. For example, we might determine that James Wear had parents named William and Sarah. The database might be updated as follows:

```
// add individual directly
JHWear.setLeft(new BinaryTree<String>("William"));
// or keep a reference to the pedigree before the update:
BinaryTree<String> SAYancey = new BinaryTree<String>("Sarah");
JHWear.setRight(SAYancey);
```

A little thought suggests a number of other features that might be useful in supporting the pedigree-as-`BinaryTree` structure.

12.3 Example: Expression Trees

Most programming languages involve mathematical expressions that are composed of binary operations applied to values. An example from Java is the simple expression `R = 1 + (L - 1) * 2`. This expression involves four operators (`=`, `+`, `-`, and `*`), and 5 values (`R`, `1`, `L`, `1`, and `2`). Languages often represent expressions using binary trees. Each value in the expression appears as a leaf,

while the operators are internal nodes that represent the reduction of two values to one (for example, `L - 1` is reduced to a single value for use on the left side of the multiplication sign). The *expression tree* associated with our expression is shown in Figure 12.3a. We might imagine that the following code constructs the tree and prints -1:

Calc

```
BinaryTree<term> v1a,v1b,v2,vL,vR,t;

// set up values 1 and 2, and declare variables
v1a = new BinaryTree<term>(new value(1));
v1b = new BinaryTree<term>(new value(1));
v2 = new BinaryTree<term>(new value(2));
vL = new BinaryTree<term>(new variable("L",0));// L=0
vR = new BinaryTree<term>(new variable("R",0));// R=0

// set up expression
t = new BinaryTree<term>(new operator('-'),vL,v1a);
t = new BinaryTree<term>(new operator('*'),t,v2);
t = new BinaryTree<term>(new operator('+'),v1b,t);
t = new BinaryTree<term>(new operator('='),vR,t);

// evaluate and print expression
System.out.println(eval(t));
```

Once an expression is represented as an expression tree, it may be evaluated by *traversing* the tree in an agreed-upon manner. Standard rules of mathematical precedence suggest that the parenthesized expression (`L-1`) should be evaluated first. (The `L` represents a value previously stored in memory.) Once the subtraction is accomplished, the result is multiplied by 2. The product is then added to 1. The result of the addition is assigned to `R`. The assignment operator is treated in a manner similar to other common operators; it just has lower *precedence* (it is evaluated later) than standard mathematical operators. Thus an implementation of binary trees would be aided by a traversal mechanism that allows us to manipulate values as they are encountered.

12.4 Implementation

We now consider the implementation of binary trees. As with `List` implementations, we will construct a self-referential `BinaryTree` class. The recursive design motivates implementation of many of the `BinaryTree` operations as recursive methods. However, because the base case of recursion often involves an empty tree we will make use of a dedicated node that represents the empty tree. This simple implementation will be the basis of a large number of more advanced structures we see throughout the remainder of the text.

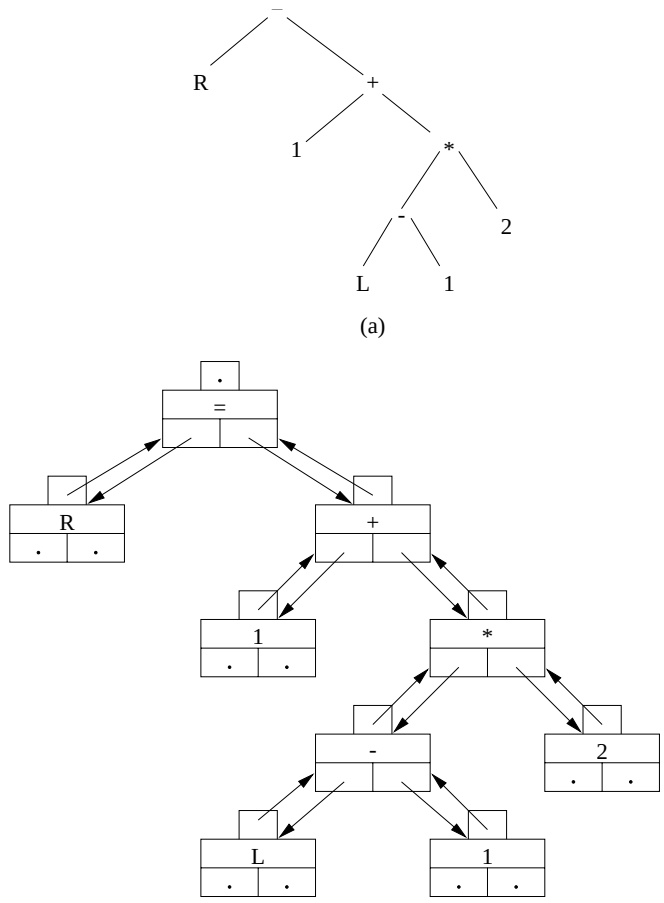
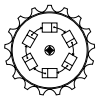


Figure 12.3 Expression trees. (a) An abstract expression tree representing `R=1+(L-1)*2`. (b) A possible connectivity of an implementation using references.

12.4.1 The BinaryTree Implementation

Our first step toward the development of a binary tree implementation is to represent an entire subtree as a reference to its root node. The node will maintain a reference to user data and related nodes (the node's parent and its two children) and directly provides methods to maintain a subtree rooted at that node. All empty trees will be represented by one or more instances of `BinaryTrees` called "empty" trees. This approach is not unlike the "dummy nodes" provided in our study of linked lists. Allowing the empty tree to be an object allows programs to call methods on trees that are empty. If the empty tree were represented by a `null` pointer, it would be impossible to apply methods to the structure. Here is the interface (again we have omitted right-handed versions of handed operations):



BinaryTree

```
public class BinaryTree<E>
{
    public BinaryTree()
    // post: constructor that generates an empty node

    public BinaryTree(E value)
    // post: returns a tree referencing value and two empty subtrees

    public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right)
    // post: returns a tree referencing value and two subtrees

    public BinaryTree<E> left()
    // post: returns reference to (possibly empty) left subtree

    public BinaryTree<E> parent()
    // post: returns reference to parent node, or null

    public void setLeft(BinaryTree<E> newLeft)
    // post: sets left subtree to newLeft
    //       re-parents newLeft if not null

    protected void setParent(BinaryTree<E> newParent)
    // post: re-parents this node to parent reference, or null

    public Iterator<E> iterator()
    // post: returns an in-order iterator of the elements

    public boolean isLeftChild()
    // post: returns true if this is a left child of parent

    public E value()
    // post: returns value associated with this node

    public void setValue(E value)
    // post: sets the value associated with this node
```

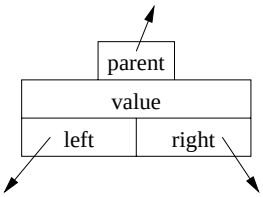


Figure 12.4 The structure of a `BinaryTree`. The parent reference opposes a left or right child reference in parent node.

```
}
```

Figure 12.3b depicts the use of `BinaryTrees` in the representation of an entire tree. We visualize the structure of a `BinaryTree` as in Figure 12.4. To construct such a node, we require three pieces of information: a reference to the data that the user wishes to associate with this node, and left and right references to binary tree nodes that are roots of subtrees of this node. The parent reference is determined implicitly from opposing references. The various methods associated with constructing a `BinaryTree` are as follows:

```
protected E val; // value associated with node
protected BinaryTree<E> parent; // parent of node
protected BinaryTree<E> left, right; // children of node

public BinaryTree()
// post: constructor that generates an empty node
{
    val = null;
    parent = null; left = right = this;
}

public BinaryTree(E value)
// post: returns a tree referencing value and two empty subtrees
{
    Assert.pre(value != null, "Tree values must be non-null.");
    val = value;
    right = left = new BinaryTree<E>();
    setLeft(left);
    setRight(right);
}

public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right)
// post: returns a tree referencing value and two subtrees
{
    Assert.pre(value != null, "Tree values must be non-null.");
```

```
        val = value;
        if (left == null) { left = new BinaryTree<E>(); }
        setLeft(left);
        if (right == null) { right = new BinaryTree<E>(); }
        setRight(right);
    }
```

The first constructor is called when an empty `BinaryTree` is needed. The result of this constructor are empty nodes that will represent members of the fringe of empty trees found along the edge of the binary tree. In the three-parameter variant of the constructor we make two calls to "setting" routines. These routines allow one to set the references of the left and right subtrees, but also ensure that the children of this node reference this node as their parent. This is the direct cost of implementing forward and backward references along every link. The return, though, is the considerable simplification of other code within the classes that make use of `BinaryTree` methods.

Principle 19 *Don't let opposing references show through the interface.*

When maintenance of opposing references is left to the user, there is an opportunity for references to become inconsistent. Furthermore, one might imagine implementations with fewer references (it is common, for example, to avoid the parent reference); the details of the implementation should be hidden from the user, in case the implementation needs to be changed.

Here is the code for `setLeft` (`setRight` is similar):

```
public void setLeft(BinaryTree<E> newLeft)
// post: sets left subtree to newLeft
//       re-parents newLeft if not null
{
    if (isEmpty()) return;
    if (left != null && left.parent() == this) left.setParent(null);
    left = newLeft;
    left.setParent(this);
}
```

If the setting of the left child causes a subtree to be disconnected from this node, and that subtree considers this node to be its parent (it should), we disconnect the node by setting its parent to `null`. We then set the left child reference to the value passed in. Any dereferenced node is explicitly told to set its `parent` reference to `null`. We also take care to set the opposite parent reference by calling the `setParent` method of the root of the associated non-trivial tree. Because we want to maintain consistency between the downward child references and the upward parent references, we declare `setParent` to be `protected` to make it impossible for the user to refer to directly:

```
protected void setParent(BinaryTree<E> newParent)
// post: re-parents this node to parent reference, or null
```

```
{
    if (!isEmpty()) {
        parent = newParent;
    }
}
```

It is, of course, useful to be able to access the various references once they have been set. We accomplish this through the accessor functions such as `left`:

```
public BinaryTree<E> left()
// post: returns reference to (possibly empty) left subtree
{
    return left;
}
```

Once the node has been constructed, its value can be inspected and modified using the value-based functions that parallel those we have seen with other types:

```
public E value()
// post: returns value associated with this node
{
    return val;
}

public void setValue(E value)
// post: sets the value associated with this node
{
    val = value;
}
```

Once the `BinaryTree` class is implemented, we may use it as the basis for our implementation of some fairly complex programs and structures.

12.5 Example: An Expert System

Anyone who has been on a long trip with children has played the game Twenty Questions. It's not clear why this game has this name, because the questioning often continues until the entire knowledge space of the child is exhausted. We can develop a very similar program, here, called `InfiniteQuestions`. The central component of the program is a database, stored as a `BinaryTree`. At each leaf is an object that is a possible guess. The interior nodes are questions that help distinguish between the guesses.

Figure 12.5 demonstrates one possible state of the database. To simulate a questioner, one asks the questions encountered on a path from the root to some leaf. If the response to the question is positive, the questioning continues along the left branch of the tree; if the response is negative, the questioner considers the right.

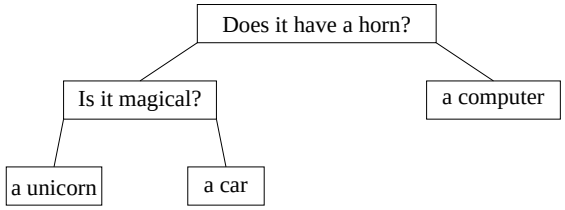


Figure 12.5 The state of the database in the midst of playing `InfiniteQuestions`.

Exercise 12.2 *What questions would the computer ask if you were thinking of a truck?*

Of course, we can build a very simple database with a single value—perhaps a computer. The game might be set up to play against a human as follows:

`Infinite-Questions`

```
public static void main(String args[])
{
    Scanner human = new Scanner(System.in);
    // construct a simple database -- knows only about a computer
    BinaryTree<String> database = new BinaryTree<String>("a computer");

    System.out.println("Do you want to play a game?");
    while (human.nextLine().equals("yes"))
    {
        System.out.println("Think of something...I'll guess it");
        play(human,database);
        System.out.println("Do you want to play again?");
    }
    System.out.println("Have a good day!");
}
```

When the game is played, we are likely to lose. If we lose, we can still benefit by incorporating information about the losing situation. If we guessed a computer and the item was a car, we could incorporate the car and a question "Does it have wheels?" to distinguish the two objects. As it turns out, the program is not that difficult.

```
public static void play(Scanner human, BinaryTree<String> database)
// pre: database is non-null
// post: the game is finished, and if we lost, we expanded the database
{
    if (!database.left().isEmpty())
    { // further choices; must ask a question to distinguish them
        System.out.println(database.value());
        if (human.nextLine().equals("yes"))
        {
```

```
                play(human,database.left());
            } else {
                play(human,database.right());
            }
        } else { // must be a statement node
            System.out.println("Is it "+database.value()+"?");
            if (human.nextLine().equals("yes"))
            {
                System.out.println("I guessed it!");
            } else {
                System.out.println("Darn.  What were you thinking of?");
                // learn!
                BinaryTree<String> newObject = new BinaryTree<String>(human.nextLine());
                BinaryTree<String> oldObject = new BinaryTree<String>(database.value());
                database.setLeft(newObject);
                database.setRight(oldObject);
                System.out.println("What question would distinguish "+
                                   newObject.value()+" from "+
                                   oldObject.value()+"?");
                database.setValue(human.nextLine());
            }
        }
    }
```

The program can distinguish questions from guesses by checking to see there is a left child. This situation would suggest this node was a question since the two children need to be distinguished.

The program is very careful to expand the database by adding new leaves at the node that represents a losing guess. If we aren't careful, we can easily corrupt the database by growing a tree with the wrong topology.

Here is the output of a run of the game that demonstrates the ability of the database to incorporate new information—that is to *learn*:

```
Do you want to play a game?
Think of something...I'll guess it
Is it a computer?
Darn.  What were you thinking of?
What question would distinguish a car from a computer?
Do you want to play again?
Think of something...I'll guess it
Does it have a horn?
Is it a car?
Darn.  What were you thinking of?
What question would distinguish a unicorn from a car?
Do you want to play again?
Think of something...I'll guess it
Does it have a horn?
Is it magical?
Is it a car?
I guessed it!
```

```
Do you want to play again?
Have a good day!
```

Exercise 12.3 *Make a case for or against this program as a (simple) model for human learning through experience.*

We now discuss the implementation of a general-purpose `Iterator` for the `BinaryTree` class. Not surprisingly a structure with branching (and therefore a choice in traversal order) makes traversal implementation more difficult. Next, we consider the construction of several `Iterators` for binary trees.

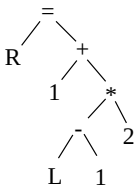
12.6 Traversals of Binary Trees

We have seen, of course, there is a great industry in selling calculators that allow users to enter expressions in what appear to be arbitrary ways. For example, some calculators allow users to specify expressions in *infix* form, where keys associated with operators are pressed between operands. Other brands of calculators advocate a *postfix*[3] form, where the operator is pressed only after the operands have been entered. Reconsidering our representation of expressions as trees, we observe that there must be a similar variety in the ways we traverse a `BinaryTree` structure. We consider those here.

When designing iterators for linear structures there are usually few useful choices: start at one end and visit each element until you get to the other end. Many of the linear structures we have seen provide an `elements` method that constructs an iterator for traversing the structure. For binary trees, there is no obvious order for traversing the structure. Here are four rather obvious but distinct mechanisms:

Preorder traversal. Each node is visited before any of its children are visited. Typically, we visit a node, and then each of the nodes in its left subtree, followed by each of the nodes in the right subtree. A preorder traversal of the expression tree in the margin visits the nodes in the order: $=$, R, $+$, 1, $*$, $-$, L, 1, and 2.

In-order traversal. Each node is visited after all the nodes of its left subtree have been visited and before any of the nodes of the right subtree. The in-order traversal is usually only useful with binary trees, but similar traversal mechanisms can be constructed for trees of arbitrary arity. An in-order traversal of the expression tree visits the nodes in the order: R, $=$, 1, $+$, L, $-$, 1, $*$, and 2. Notice that, while this representation is similar to the expression that actually generated the binary tree, the traversal has removed the parentheses.



[3] Reverse Polish Notation (RPN) was developed by Jan Lukasiewicz, a philosopher and mathematician of the early twentieth century, and was made popular by Hewlett-Packard in their calculator wars with Texas Instruments in the early 1970s.

Postorder traversal. Each node is visited after its children are visited. We visit all the nodes of the left subtree, followed by all the nodes of the right subtree, followed by the node itself. A postorder traversal of the expression tree visits the nodes in the order: R, 1, L, 1, $-$, 2, $*$, $+$, and $=$. This is precisely the order that the keys would have to be pressed on a "reverse Polish" calculator to compute the correct result.

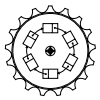
Level-order traversal. All nodes of level i are visited before the nodes of level $i + 1$. The nodes of the expression tree are visited in the order: $=$, R, $+$, 1, $*$, $-$, 2, L, and 1. (This particular ordering of the nodes is motivation for another implementation of binary trees we shall consider later and in Problem 12.12.)

As these are the most common and useful techniques for traversing a binary tree we will investigate their respective implementations. Traversing `BinaryTrees` involves constructing an iterator that traverses the entire set of subtrees. For this reason, and because the traversal of subtrees proves to be just as easy, we discuss implementations of iterators for `BinaryTrees`.

Most implementations of iterators maintain a linear structure that keeps track of the state of the iterator. In some cases, this auxiliary structure is not strictly necessary (see Problem 12.22) but may reduce the complexity of the implementation and improve its performance.

12.6.1 Preorder Traversal

For a preorder traversal, we wish to traverse each node of the tree before any of its proper descendants (recall the node is a descendant of itself). To accomplish this, we keep a stack of nodes whose right subtrees have not been investigated. In particular, the current node is the topmost element of the stack, and elements stored deeper within the stack are more distant ancestors.

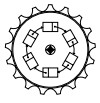


`BinaryTree`

We develop a new implementation of an `Iterator` that is not declared `public`. Since it will be a member of the `structure` package, it *is* available for use by the classes of the `structure` package, including `BinaryTree`. The `BinaryTree` class will construct and return a reference to the preorder iterator when the `preorderElements` method is called:

```
public AbstractIterator<E> preorderIterator()
// post: the elements of the binary tree rooted at node are
//       traversed in preorder
{
    return new BTPreorderIterator<E>(this);
}
```

Note that the constructor for the iterator accepts a single parameter—the root of the subtree to be traversed. Because the iterator only gives access to values stored within nodes, this is not a breach of the privacy of our binary tree implementation. The actual implementation of the `BTPreorderIterator` is short:



`BTPreorder-`
`Iterator`

```
class BTPreorderIterator<E> extends AbstractIterator<E>
{
    protected BinaryTree<E> root; // root of tree to be traversed
    protected Stack<BinaryTree<E>> todo; // stack of unvisited nodes whose

    public BTPreorderIterator(BinaryTree<E> root)
    // post: constructs an iterator to traverse in preorder
    {
        todo = new StackList<BinaryTree<E>>();
        this.root = root;
        reset();
    }

    public void reset()
    // post: resets the iterator to retraverse
    {
        todo.clear(); // stack is empty; push on root
        if (root != null) todo.push(root);
    }

    public boolean hasNext()
    // post: returns true iff iterator is not finished
    {
        return !todo.isEmpty();
    }

    public E get()
    // pre: hasNext()
    // post: returns reference to current value
    {
        return todo.get().value();
    }

    public E next()
    // pre: hasNext();
    // post: returns current value, increments iterator
    {
        BinaryTree<E> old = todo.pop();
        E result = old.value();

        if (!old.right().isEmpty()) todo.push(old.right());
        if (!old.left().isEmpty()) todo.push(old.left());
        return result;
    }
}
```

As we can see, `todo` is the private stack used to keep track of references to unvisited nodes whose nontrivial ancestors have been visited. Another way to think about it is that it is the frontier of nodes encountered on paths from the root that have not yet been visited. To construct the iterator we initialize the

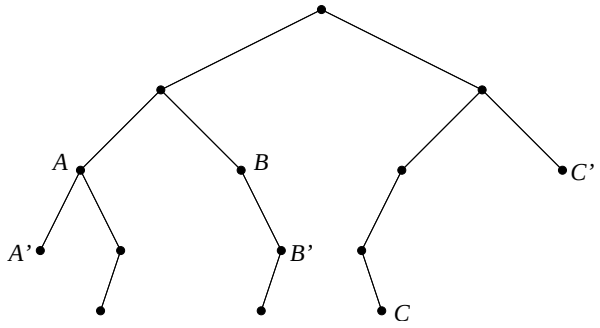


Figure 12.6 Three cases of determining the next current node for preorder traversals. Node A has a left child A' as the next node; node B has no left, but a right child B'; and node C is a leaf and finds its closest, "right cousin," C'.

---

stack. We also keep a reference to the root node; this will help reset the iterator to the correct node (when the root of the traversal is not the root of the tree, this information is vital). We then reset the iterator to the beginning of the traversal.

Resetting the iterator involves clearing off the stack and then pushing the root on the stack to make it the current node. The `hasNext` method needs only to check to see if there *is* a top node of the stack, and `value` returns the reference stored within the topmost `BinaryTree` of the `todo` stack.

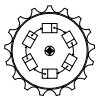
The only tricky method is `next`. Recall that this method returns the value of the current element and then increments the iterator, causing the iterator to reference the next node in the traversal. Since the current node has just been visited, we push on any children of the node—first any right child, then any left. If the node has a left child (see node A of Figure 12.6), that node (A') is the next node to be visited. If the current node (see node B) has only a right child (B'), it will be visited next. If the current node has no children (see node C), the effect is to visit the closest unvisited right cousin or sibling (C').

It is clear that over the life of the iterator each of the n values of the tree is pushed onto and popped off the stack exactly once; thus the total cost of traversing the tree is $O(n)$. A similar observation is possible for each of the remaining iteration techniques.

12.6.2 In-order Traversal

The most common traversal of trees is in order. For this reason, the `BTInorderIterator` is the value returned when the `elements` method is called on a `BinaryTree`. Again, the iterator maintains a stack of references to nodes. Here, the stack contains unvisited ancestors of the current (unvisited) node.

Thus, the implementation of this traversal is similar to the code for other iterators, except for the way the stack is `reset` and for the mechanism provided in the `nextElement` method:



BTInorder-Iterator

```
protected BinaryTree<E> root;    // root of subtree to be traversed

protected Stack<BinaryTree<E>> todo; // stack of unvisited ancestors

public void reset()
// post: resets the iterator to retraverse
{
    todo.clear();
    // stack is empty.  Push on nodes from root to
    // leftmost descendant
    BinaryTree<E> current = root;
    while (!current.isEmpty()) {
        todo.push(current);
        current = current.left();
    }
}

public E next()
// pre: hasNext()
// post: returns current value, increments iterator
{
    BinaryTree<E> old = todo.pop();
    E result = old.value();
    // we know this node has no unconsidered left children;
    // if this node has a right child,
    //   we push the right child and its leftmost descendants:
    // else
    //   top element of stack is next node to be visited
    if (!old.right().isEmpty()) {
        BinaryTree<E> current = old.right();
        do {
            todo.push(current);
            current = current.left();
        } while (!current.isEmpty());
    }
    return result;
}
```

Since the first element considered in an in-order traversal is the leftmost descendant of the root, resetting the iterator involves pushing each of the nodes from the root down to the leftmost descendant on the auxiliary stack.

When the current node is popped from the stack, the next element of the traversal must be found. We consider two scenarios:

1. If the current node has a right subtree, the nodes of that tree have not been visited. At this stage we should push the right child, and all the

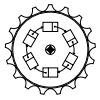
nodes down to and including its leftmost descendant, on the stack.

2. If the node has no right child, the subtree rooted at the current node has been fully investigated, and the next node to be considered is the closest unvisited ancestor of the former current node—the node just exposed on the top of the stack.

As we shall see later, it is common to order the nodes of a binary tree so that left-hand descendants of a node are smaller than the node, which is, in turn, smaller than any of the rightmost descendants. In such a situation, the in-order traversal plays a natural role in presenting the data of the tree in order. For this reason, the `elements` method returns the iterator constructed by the `inorderElements` method.

12.6.3 Postorder Traversal

Traversing a tree in postorder also maintains a stack of uninvestigated nodes. Each of the elements on the stack is a node whose descendants are currently being visited. Since the first element to be visited is the leftmost descendant of the root, the `reset` method must (as with the in-order iterator) push on each of the nodes from the root to the leftmost descendant. (Note that the leftmost descendant need not be a leaf—it does not have a left child, but it may have a right.)



BTPostorder-Iterator

```
protected BinaryTree<E> root; // root of traversed subtree
protected Stack<BinaryTree<E>> todo;  // stack of nodes
// whose descendants are currently being visited

public void reset()
// post: resets the iterator to retraverse
{
    todo.clear();
    // stack is empty; push on nodes from root to
    // leftmost descendant
    BinaryTree<E> current = root;
    while (!current.isEmpty()) {
        todo.push(current);
        if (!current.left().isEmpty())
            current = current.left();
        else
            current = current.right();
    }
}

public E next()
// pre: hasNext();
// post: returns current value, increments iterator
{
    BinaryTree<E> current = todo.pop();
```

```
        E result = current.value();
        if (!todo.isEmpty())
        {
            BinaryTree<E> parent = todo.get();
            if (current == parent.left()) {
                current = parent.right();
                while (!current.isEmpty())
                {
                    todo.push(current);
                    if (!current.left().isEmpty())
                         current = current.left();
                    else current = current.right();
                }
            }
        }
        return result;
    }
```

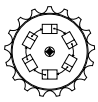
Here an interior node on the stack is potentially exposed twice before becoming current. The first time it may be left on the stack because the element recently popped off was the left child. The right child should now be pushed on. Later the exposed node becomes current because the popped element was its right child.

It is interesting to observe that the stack contains the ancestors of the current node. This stack describes, essentially, the path to the root of the tree. As a result, we could represent the state of the stack by a single reference to the current node.

12.6.4 Level-order Traversal

This is the family values traversal.

A level-order traversal visits the root, followed by the nodes of level 1, from left to right, followed by the nodes of level 2, and so on. This can be easily accomplished by maintaining a queue of the next few nodes to be visited. More precisely, the queue contains the current node, followed by a list of all siblings and cousins to the right of the current node, followed by a list of "nieces and nephews" to the left of the current node. After we visit a node, we enqueue the children of the node. With a little work it is easy to see that these are either nieces and nephews or right cousins of the next node to be visited.



BTLevelorder-Iterator

```
class BTLevelorderIterator<E> extends AbstractIterator<E>
{
    protected BinaryTree<E> root; // root of traversed subtree
    protected Queue<BinaryTree<E>> todo;  // queue of unvisited relatives

    public BTLevelorderIterator(BinaryTree<E> root)
    // post: constructs an iterator to traverse in level order
    {
        todo = new QueueList<BinaryTree<E>>();
```

```
            this.root = root;
            reset();
        }

        public void reset()
        // post: resets the iterator to root node
        {
            todo.clear();
            // empty queue, add root
            if (!root.isEmpty()) todo.enqueue(root);
        }

        public boolean hasNext()
        // post: returns true iff iterator is not finished
        {
            return !todo.isEmpty();
        }

        public E get()
        // pre: hasNext()
        // post: returns reference to current value
        {
            return todo.get().value();
        }

        public E next()
        // pre: hasNext();
        // post: returns current value, increments iterator
        {
            BinaryTree<E> current = todo.dequeue();
            E result = current.value();
            if (!current.left().isEmpty())
                todo.enqueue(current.left());
            if (!current.right().isEmpty())
                todo.enqueue(current.right());
            return result;
        }
    }
```

To `reset` the iterator, we need only empty the queue and add the root. When the queue is empty, the traversal is finished. When the next element is needed, we need only enqueue references to children (left to right). Notice that, unlike the other iterators, this method of traversing the tree is meaningful regardless of the degree of the tree.

12.6.5 Recursion in Iterators

Trees are recursively defined structures, so it would seem reasonable to consider recursive implementations of iterators. The difficulty is that iterators must

maintain their state across many calls to `nextElement`. Any recursive approach to traversal would encounter nodes while deep in recursion, and the state of the stack must be preserved.

One way around the difficulties of suspending the recursion is to initially perform the entire traversal, generating a list of values encountered. Since the entire traversal happens all at once, the list can be constructed using recursion. As the iterator pushes forward, the elements of the list are consumed.

Using this idea, we rewrite the in-order traversal:

Recursive-Iterators

```
protected BinaryTree<T> root; // root of traversed subtree
protected Queue<BinaryTree<T>> todo;  // queue of unvisited elements

public BTInorderIteratorR(BinaryTree<T> root)
// post: constructs an iterator to traverse in in-order
{
    todo = new QueueList<BinaryTree<T>>();
    this.root = root;
    reset();
}

public void reset()
// post: resets the iterator to retraverse
{
    todo.clear();
    enqueueInorder(root);
}

protected void enqueueInorder(BinaryTree<T> current)
// pre: current is non-null
// post: enqueue all values found in tree rooted at current
//       in in-order
{
    if (current.isEmpty()) return;
    enqueueInorder(current.left());
    todo.enqueue(current);
    enqueueInorder(current.right());
}
public T next()
// pre: hasNext();
// post: returns current value, increments iterator
{
    BinaryTree<T> current = todo.dequeue();
    return current.value();
}
```

The core of this implementation is the protected method `enqueueInorder`. It simply traverses the tree rooted at its parameter and enqueues every node encountered. Since it recursively enqueues all its left descendants, then itself, and then its right descendants, it is an in-order traversal. Since the queue is a FIFO, the order is preserved and the elements may be consumed at the user's leisure.

For completeness and demonstration of symmetry, here are the pre- and postorder counterparts:

```
protected void enqueuePreorder(BinaryTree<T> current)
// pre: current is non-null
// post: enqueue all values found in tree rooted at current
//       in preorder
{
    if (current.isEmpty()) return;
    todo.enqueue(current);
    enqueuePreorder(current.left());
    enqueuePreorder(current.right());
}

protected void enqueuePostorder(BinaryTree<T> current)
// pre: current is non-null
// post: enqueue all values found in tree rooted at current
//       in postorder
{
    if (current.isEmpty()) return;
    enqueuePostorder(current.left());
    enqueuePostorder(current.right());
    todo.enqueue(current);
}
```

It is reassuring to see the brevity of these implementations. Unfortunately, while the recursive implementations are no less efficient, they come at the obvious cost of a potentially long delay whenever the iterator is reset. Still, for many applications this may be satisfactory.

12.7 Property-Based Methods

At this point, we consider the implementation of a number of property-based methods. Properties such as the height and fullness of a tree are important to guiding updates of a tree structure. Because the binary tree is a recursively defined data type, the proofs of tree characteristics (and the methods that verify them) often have a recursive feel. To emphasize the point, in this section we allow theorems about trees and methods that verify them to intermingle. Again, the methods described here are written for use on `BinaryTrees`, but they are easily adapted for use with more complex structures.

Our first method makes use of the fact that the root is a common ancestor of every node of the tree. Because of this fact, given a `BinaryTree`, we can identify the node as the root, or return the root of the tree containing the node's parent.

BinaryTree

```
public BinaryTree<E> root()
// post: returns the root of the tree node n
{
    if (parent() == null) return this;
    else return parent().root();
}
```

A proof that this method functions correctly could make use of induction, based on the depth of the node involved.

If we count the number of times the `root` routine is recursively called, we compute the number of edges from the node to the root—the depth of the node. Not surprisingly, the code is very similar:

```
public int depth()
// post: returns the depth of a node in the tree
{
    if (parent() == null) return 0;
    return 1 + parent.depth();
}
```

The time it takes is proportional to the depth of the node. For full trees, we will see that this is approximately $O(\log n)$. Notice that in the empty case we return a height of -1. This is consistent with our recursive definition, even if it does seem a little unusual. We could avoid the strange case by avoiding it in the precondition. Then, of course, we would only have put off the work to the calling routine. Often, making tough decisions about base cases can play an important role in making your interface useful. Generally, a method is more robust, and therefore more usable, if you handle as many cases as possible.

Principle 20 *Write methods to be as general as possible.*

Having computed the depth of a node, it follows that we should be able to determine the height of a tree rooted at a particular `BinaryTree`. We know that the height is simply the length of a longest path from the root to a leaf, but we can adapt a self-referential definition: the height of a tree is one more than the height of the tallest subtree. This translates directly into a clean implementation of the height function:

```
public int height()
// post: returns the height of a node in its tree
{
    if (isEmpty()) return -1;
    return 1 + Math.max(left.height(),right.height());
}
```

This method takes $O(n)$ time to execute on a subtree with n nodes (see Problem 12.9).

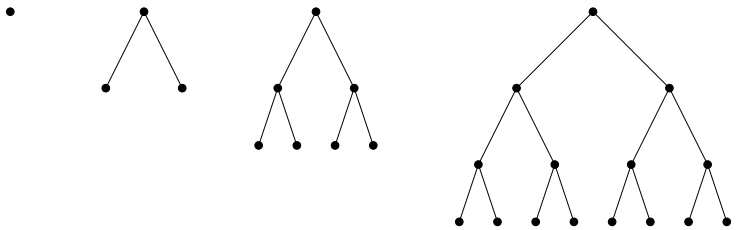


Figure 12.7 Several full (and complete) binary trees.

At this point, we consider the problem of identifying a tree that is full (see Figure 12.7). Our approach uses recursion:

```
public boolean isFull()
// post: returns true iff the tree rooted at node is full
{
    if (isEmpty()) return true;
    if (left().height() != right().height()) return false;
    return left().isFull() && right().isFull();
}
```

Again, the method is compact. Unfortunately, detecting this property appears to be significantly more expensive than computing the height. Note, for example, that in the process of computing this function on a full tree, the *height of every node* must be computed from scratch. The result is that the running time of the algorithm on full trees is $O(n \log n)$. Can it be improved upon?

To find the answer, we first prove a series of theorems about the structure of trees, with hope that we can develop an inexpensive way to test for a full tree. Our first result determines the number of nodes that are found in full trees:

Observation 12.1 *A full binary tree of height $h \geq 0$ has $2^{h+1} - 1$ nodes.*

Proof: We prove this by induction on the height of the tree. Suppose the tree has height 0. Then it has exactly one node, which is also a leaf. Since $2^1 - 1 = 1$, the observation holds, trivially.

Our inductive hypothesis is that full trees of height $k < h$ have $2^{k+1} - 1$ nodes. Since $h > 0$, we can decompose the tree into two full subtrees of height $h - 1$, under a common root. Each of the full subtrees has $2^{(h-1)+1} - 1 = 2^h - 1$ nodes, so there are $2(2^h - 1) + 1 = 2^{h+1} - 1$ nodes. This is the result we sought to prove, so by induction on tree height we see the observation must hold for all full binary trees. $\diamond$

This observation suggests that if we can compute the height and size of a tree, we have a hope of detecting a full tree. First, we compute the size of the tree using a recursive algorithm:

```
public int size()
// post: returns the size of the subtree
{
    if (isEmpty()) return 0;
    return left().size() + right().size() + 1;
}
```

This algorithm is similar to the height algorithm: each call to size counts one more node, so the complexity of the routine is $O(n)$. Now we have an alternative implementation of `isFull` that compares the height of the tree to the number of nodes:

```
public boolean isFull()
// post: returns true iff the tree rooted at n is full
{
    int h = height();
    int s = size();
    return s == (1<<(h+1))-1;
}
```

Notice the return statement makes use of shifting 1 to the left $h + 1$ binary places. This efficiently computes 2^{h+1}. The result is that, given a full tree, the function returns true in $O(n)$ steps. Thus, it *is* possible to improve on our previous implementation.

There is one significant disadvantage, though. If you are given a tree with height greater than 100, the result of the return statement cannot be accurately computed: 2^{100} is a large enough number to overflow Java integers. Even reasonably sized trees can have height greater than 100. The first implementation is accurate, even if it is slow. Problem 12.21 considers an efficient and accurate solution.

Redwoods and sequoias come to mind.

We now prove some useful facts about binary trees that help us evaluate performance of methods that manipulate them. First, we consider a pretty result: if a tree has lots of leaves, it must branch in lots of places.

Observation 12.2 *The number of full nodes in a binary tree is one less than the number of leaves.*

Proof: Left to the reader.$\diamond$

With this result, we can now demonstrate that just over half the nodes of a full tree are leaves.

Observation 12.3 *A full binary tree of height $h \geq 0$ has 2^h leaves.*

Proof: In a full binary tree, all nodes are either full interior nodes or leaves. The number of nodes is the sum of full nodes F and the number of leaves L. Since, by Observation 12.2, $F = L - 1$, we know that the count of nodes is $F + L = 2L - 1 = 2^{h+1} - 1$. This leads us to conclude that $L = 2^h$ and that $F = 2^h - 1$. $\diamond$ This result demonstrates that for many simple tree methods (like

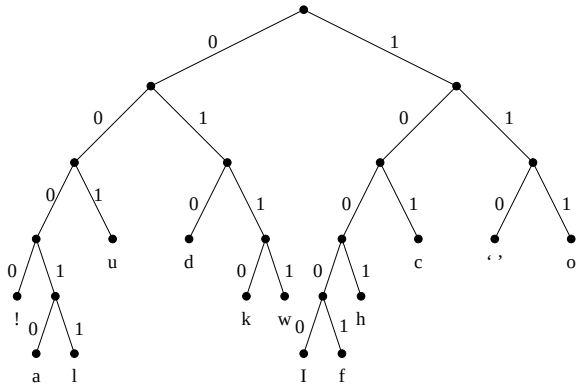


Figure 12.8 The woodchuck Huffman tree. Leaves are labeled with the characters they represent. Paths from root to leaves provide Huffman bit strings.

`size`) half of the time is spent processing leaves. Because complete trees can be viewed as full trees with some rightmost leaves removed, similar results hold for complete trees as well.

12.8 Example: Huffman Compression

Information within machines is stored as a series of `bits`, or 1's and 0's. Because the distribution of the patterns of 1's and 0's is not always uniform, it is possible to compress the bit patterns that are used and reduce the amount of storage that is necessary. For example, consider the following 32-character phrase:

`Huffman`

```
If a woodchuck could chuck wood!
```

If each letter in the string is represented by 8 bits (as they often are), the entire string takes 256 bits of storage. Clearly this catchy phrase does not use the full range of characters, and so perhaps 8 bits are not needed. In fact, there are 13 distinct characters so 4 bits would be sufficient (4 bits can represent any of 16 values). This would halve the amount of storage required, to 128 bits.

If each character were represented by a unique *variable-length* string of bits, further improvements are possible. *Huffman encoding* of characters allows us to reduce the size of this string to only 111 bits by assigning frequently occurring letters (like "`o`") short representations and infrequent letters (like "`a`") relatively long representations.

Huffman encodings can be represented by binary trees whose leaves are the characters to be represented. In Figure 12.8 left edges are labeled 0, while right edges are labeled 1. Since there is a unique path from the root to each leaf, there

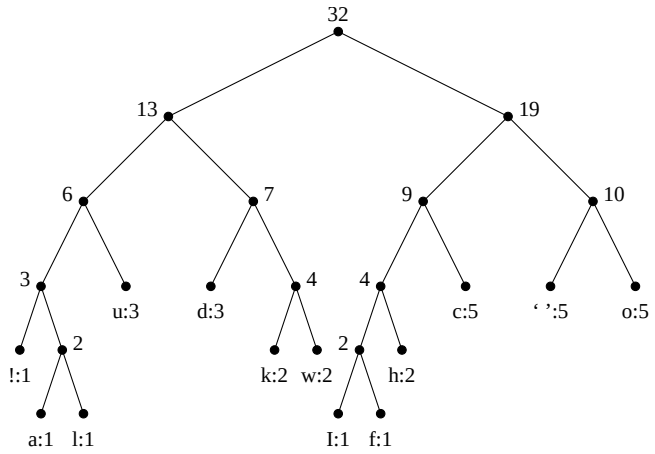


Figure 12.9 The Huffman tree of Figure 12.8, but with nodes labeled by total frequencies of descendant characters.

---

is a unique sequence of 1's and 0's encountered as well. We will use the string of bits encountered along the path to a character as its representation in the compressed output. Note also that no string is a prefix for any other (otherwise one character would be an ancestor of another in the tree). This means that, given the Huffman tree, decoding a string of bits involves simply traversing the tree and writing out the leaves encountered.

The construction of a Huffman tree is an iterative process. Initially, each character is placed in a Huffman tree of its own. The weight of the tree is the frequency of its associated character. We then iteratively merge the two most lightweight Huffman trees into a single new Huffman tree whose weight is the sum of weights of the subtrees. This continues until one tree remains. One possible tree for our example is shown in Figure 12.9.

Our approach is to use `BinaryTrees` to maintain the structure. This allows the use of recursion and easy merging of trees. Leaves of the tree carry descriptions of characters and their frequencies:

Huffman

```
class node
{
    int frequency; // frequency of char
    char ch;    // the character

    public node(int f)
    // post: construct an entry with frequency f

    public node(char c)
    // post: construct character entry with frequency 1
```

```
    public boolean equals(Object other)
    // post: return true if leaves represent same character
}
```

Intermediate nodes carry no data at all. Their relation to their ancestors determines their portion of the encoding. The entire tree is managed by a wrapper class, `huffmanTree`:

```
class huffmanTree implements Comparable<huffmanTree>
{
    BinaryTree<node> empty;
    BinaryTree<node> root; // root of tree
    int totalWeight;     // weight of tree

    public huffmanTree(node e)
    // post: construct a node with associated character

    public huffmanTree(huffmanTree left, huffmanTree right)
    // pre: left and right non-null
    // post: merge two trees together and merge their weights

    public int compareTo(huffmanTree other)
    // pre: other is non-null
    // post: return integer reflecting relation between values

    public boolean equals(Object that)
    // post: return true if this and that are same tree instance

    public void print()
    // post: print out strings associated with characters in tree

    protected void print(BinaryTree r, String representation)
    // post: print out strings associated with chars in tree r,
    //       prefixed by representation
}
```

This class is a `Comparable` because it implements the `compareTo` method. That method allows the trees to be ordered by their total weight during the merging process. The utility method `print` generates our output recursively, building up a different encoding along every path.

We now consider the construction of the tree:

```
public static void main(String args[])
{
    // read System.in one character at a time
    Scanner s = new Scanner(System.in).useDelimiter("");
    List<node> freq = new SinglyLinkedList<node>();

    // read data from input
```

```
        while (s.hasNext())
        {
            // s.next() returns string; we're interested in first char
            char c = s.next().charAt(0);
            // look up character in frequency list
            node query = new node(c);
            node item = freq.remove(query);
            if (item == null)
            {   // not found, add new node
                freq.addFirst(query);
            } else { // found, increment node
                item.frequency++;
                freq.addFirst(item);
            }
        }

        // insert each character into a Huffman tree
        OrderedList<huffmanTree> trees = new OrderedList<huffmanTree>();
        for (node n : freq)
        {
            trees.add(new huffmanTree(n));
        }

        // merge trees in pairs until one remains
        Iterator ti = trees.iterator();
        while (trees.size() > 1)
        {
            // construct a new iterator
            ti = trees.iterator();
            // grab two smallest values
            huffmanTree smallest = (huffmanTree)ti.next();
            huffmanTree small = (huffmanTree)ti.next();
            // remove them
            trees.remove(smallest);
            trees.remove(small);
            // add bigger tree containing both
            trees.add(new huffmanTree(smallest,small));
        }
        // print only tree in list
        ti  = trees.iterator();
        Assert.condition(ti.hasNext(),"Huffman tree exists.");
        huffmanTree encoding = (huffmanTree)ti.next();
        encoding.print();
    }
```

There are three phases in this method: the reading of the data, the construction of the character-holding leaves of the tree, and the merging of trees into a single encoding. Several things should be noted:

1. We store characters in a list. Since this list is likely to be small, keeping it

ordered requires more code and is not likely to improve performance.

2. The `huffmanTrees` are kept in an `OrderedList`. Every time we remove values we must construct a fresh iterator and remove the two smallest trees. When they are merged and reinserted, the wrappers for the two smaller trees can be garbage-collected. (Even better structures for managing these details in Chapter 13.)

3. The resulting tree is then printed out. In an application, the information in this tree would have to be included with the compressed text to guide the decompression.

When the program is run on the input

```
If a woodchuck could chuck wood!
```

it generates the output:

```
Encoding of ! is 0000 (frequency was 1)
Encoding of a is 00010 (frequency was 1)
Encoding of l is 00011 (frequency was 1)
Encoding of u is 001 (frequency was 3)
Encoding of d is 010 (frequency was 3)
Encoding of k is 0110 (frequency was 2)
Encoding of w is 0111 (frequency was 2)
Encoding of I is 10000 (frequency was 1)
Encoding of f is 10001 (frequency was 1)
Encoding of h is 1001 (frequency was 2)
Encoding of c is 101 (frequency was 5)
Encoding of   is 110 (frequency was 5)
Encoding of o is 111 (frequency was 5)
```

Again, the total number of bits that would be used to represent our compressed phrase is only 111, giving us a compression rate of 56 percent. In these days of moving bits about, the construction of efficient compression techniques is an important industry—one industry that depends on the efficient implementation of data structures.

12.9 Example Implementation: Ahnentafel

Having given, in Section 12.2, time to the Republican genealogists, we might now investigate the heritage of a Democrat, William Jefferson Clinton. In Figure 12.10 we see the recent family tree presented as a list. This arrangement is called an *ahnentafel*, or ancestor table. The table is generated by performing a level-order traversal of the pedigree tree, and placing the resulting entries in a table whose index starts at 1.

This layout has some interesting features. First, if we have an individual with index i, the parents of the individual are found in table entries $2i$ and

1	William Jefferson Clinton
2	William Jefferson Blythe III
3	Virginia Dell Cassidy
4	William Jefferson Blythe II
5	Lou Birchie Ayers
6	Eldridge Cassidy
7	Edith Grisham
8	Henry Patton Foote Blythe
9	Frances Ellen Hines
10	Simpson Green Ayers
11	Hattie Hayes
12	James M. Cassidy
13	Sarah Louisa Russell
14	Lemma Newell Grisham
15	Edna Earl Adams

Figure 12.10 The genealogy of President Clinton, presented as a linear table. Each individual is assigned an index i. The parents of the individual can be found at locations $2i$ and $2i + 1$. Performing an integer divide by 2 generates the index of a child. Note the table starts at index 1.

$2i + 1$. Given the index i of a parent, we can find the child (there is only one child for every parent in a pedigree), by dividing the index by 2 and throwing away the remainder.

We can use this as the basis of an implementation of short binary trees. Of course, if the tree becomes tall, there is potentially a great amount of data in the tree. Also, if a tree is not full, there will be empty locations in the table. These must be carefully managed to keep from interpreting these entries as valid data. While the math is fairly simple, our `Lists` are stored with the first element at location 0. The implementor must either choose to keep location 0 blank or to modify the indexing methods to make use of zero-origin indices.

One possible approach to storing tree information like this is to store entrees in key-value pairs in the list structure, with the key being the index. In this way, the tree can be stored compactly and, if the associations are kept in an ordered structure, they can be referenced with only a logarithmic slowdown.

Exercise 12.4 *Describe what would be necessary to allow support for trees with degrees up to eight (called* octtrees*). At what cost do we achieve this increased functionality?*

In Chapter 13 we will make use of an especially interesting binary tree called a *heap*. We will see the ahnentafel approach to storing heaps in a vector shortly.

12.10 Conclusions

The tree is a nonlinear structure. Because of branching in the tree, we will find it is especially useful in situations where decisions can guide the process of adding and removing nodes.

Our approach to implementing the binary tree—a tree with degree 2 or less—is to visualize it as a self-referential structure. This is somewhat at odds with an object-oriented approach. It is, for example, difficult to represent empty self-referential structures in a manner that allows us to invoke methods. To relieve the tension between these two approaches, we represent the empty tree with class instances that represent "empty" trees.

The power of recursion on branching structures is that significant work can be accomplished with very little code. Sometimes, as in our implementation of the `isFull` method, we find ourselves subtly pushed away from an efficient solution because of overzealous use of recursion. Usually we can eliminate such inefficiencies, but we must always verify that our methods act reasonably.

Self Check Problems

Solutions to these problems begin on page 448.

12.1 Can a tree have no root? Can a tree have no leaves?

12.2 Can a binary tree have more leaves than interior nodes? Can it have more interior nodes than leaves?

12.3 In a binary tree, which node (or nodes) have greatest height?

12.4 Is it possible to have two different paths between a root and a leaf?

12.5 Why are arithmetic expressions naturally stored in binary trees?

12.6 Many spindly trees look like lists. Is a `BinaryTree` a `List`?

12.7 Suppose we wanted to make a `List` from a `BinaryTree`. How might we provide indices to the elements of the tree?

12.8 Could the queue in the level-order traversal of a tree be replaced with a stack?

12.9 Recursion is used to compute many properties of trees. What portion of the tree is usually associated with the base case?

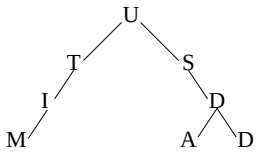
12.10 In code that recursively traverses binary trees, how many recursive calls are usually found within the code?

12.11 What is the average degree of a node in an n-node binary tree?

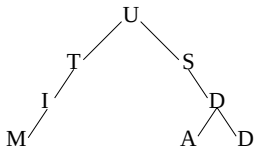
Problems

Solutions to the odd-numbered problems begin on page 477.

12.1 In the following binary tree containing character data, describe the characters encountered in pre-, post- and in-order traversals.



12.2 In the following tree what are the ancestors of the leaf D? What are the descendants of the node S? The root of the tree is the common ancestor of what nodes?



12.3 Draw an expression tree for each of the following expressions.

a. 1

b. $1 + 5 * 3 - 4/2$

c. $1 + 5 * (3 - 4)/2$

d. $(1 + 5) * (3 - 4/2)$

e. $(1 + (5 * (3 - (4/2))))$

Circle the nodes that are ancestors of the node containing the value 1.

12.4 What topological characteristics distinguish a tree from a list?

12.5 Demonstrate how the expression tree associated with the expression $R = 1 + (L - 1) * 2$ can be simplified using first the distributive property and then reduction of constant expressions to constants. Use pictures to forward your argument.

12.6 For each of the methods of `BinaryTree`, indicate which method can be implemented in terms of other `public` methods of that class or give a reasoned argument why it is not possible. Explain why it is useful to cast methods in terms of other `public` methods and not base them directly on a particular implementation.

12.7 The `BinaryTree` class is a *recursive data structure*, unlike the `List` class. Describe how the `List` class would be different if it were implemented as a recursive data structure.

12.8 The `parent` reference in a `BinaryTree` is declared `protected` and is accessed through the accessor methods `parent` and `setParent`. Why is this any different than declaring `parent` to be `public`.

12.9 Prove that efficient computation of the height of a `BinaryTree` must take time proportional to the number of nodes in the tree.

12.10 Write an `equals` method for the `BinaryTree` class. This function should return `true` if and only if the trees are similarly shaped and refer to equal values (every `Object`, including the `Objects` of the tree, has an `equals` method).

12.11 Write a static method, `copy`, that, given a binary tree, returns a copy of the tree. Because not every object implements the `copy` method, you should *not* copy objects to which the tree refers. This is referred to as a *shallow copy*.

12.12 Design a nonrecursive implementation of a binary tree that maintains node data in a `Vector`, `data`. In this implementation, element 0 of `data` references the root (if it exists). Every non-`null` element i of `data` finds its left and right children at locations $2i+1$ and $2(i+1)$, respectively. (The inverse of these index relations suggests the parent of a nonroot node at i is found at location $\lfloor (i-1)/2 \rfloor$.) Any element of data that is not used to represent a node should maintain a `null` reference.

12.13 Design an *interface* for general trees—trees with unbounded degree. Make this interface as consistent as possible with `BinaryTrees` when the degree of a tree is no greater than 2.

12.14 Implement the general tree structure of Problem 12.13 using `BinaryTreeNodes`. In this implementation, we interpret the left child of a `BinaryTreeNode` to be the leftmost child, and the right child of the `BinaryTree` to be the leftmost right sibling of the node.

12.15 Write a preorder iterator for the general tree implementation of Problem 12.14.

12.16 Implement the general tree structure of Problem 12.13 using a tree node of your own design. In this implementation, each node maintains (some sort of) collection of subtrees.

12.17 Write an in-order iterator for the general tree implementation of Problem 12.16.

12.18 Determine the complexity of each of the methods implemented in Problem 12.14.

12.19 Write a method, `isComplete`, that returns `true` if and only if the subtree rooted at a `BinaryTree` on which it acts is complete.

12.20 A tree is said to be an *AVL tree* or *height balanced* if, for every node n, the heights of the subtrees of n differ by no more than 1. Write a static `BinaryTree` method that determines if a tree rooted at the referenced node is height balanced.

12.21 The `BinaryTree` method `isFull` takes $O(n \log n)$ time to execute on full trees, which, as we've seen, is not optimal. Careful thought shows that calls to `height` (an $O(n)$ operation) are made more often than is strictly necessary. Write a recursive method `info` that computes two values—the height of the tree and whether or not the tree is full. (This might be accomplished by having the sign of the height be negative if it is not full. Make sure you do not call this method on empty trees.) If `info` makes no call to `height` or `isFull`, its performance is $O(n)$. Verify this on a computer by counting procedure calls. This

process is called *strengthening*, an optimization technique that often improves performance of recursive algorithms.

12.22 Demonstrate how, in an in-order traversal, the associated stack can be removed and replaced with a single reference. (Hint: We only need to know the top of the stack, and the elements below the stack top are determined *by* the stack top.)

12.23 Which other traversals can be rewritten by replacing their `Linear` structure with a single reference? How does this change impact the complexity of each of the iterations?

12.24 Suppose the nodes of a binary tree are unique and that you are given the order of elements as they are encountered in a preorder traversal and the order of the elements as they are encountered in a postorder traversal. Under what conditions can you accurately reconstruct the structure of the tree from these two traversal orders?

12.25 Suppose you are to store a k-ary tree where each internal node has k children and (obviously) each leaf has none. If $k = 2$, we see that Observation 12.2 suggests that there is one more leaf than internal node. Prove that a similar situation holds for k-ary trees with only full nodes and leaves: if there are n full nodes, there are $(k-1)n+1$ leaves. (Hint: Use induction.)

12.26 Assume that the observation of Problem 12.25 is true and that you are given a k-ary tree with only full nodes and leaves constructed with references between nodes. In a k-ary tree with n nodes, how many references are `null`? Considerable space might be saved if the k references to the children of an internal node were stored in a k-element array, instead of k fields. In leaves, the array needn't be allocated. In an 8-ary tree with only full nodes and leaves (an "octtree") with one million internal nodes, how many bytes of space can be saved using this array technique (assume all references consume 4 bytes).

12.11 Laboratory: Playing Gardner's Hex-a-Pawn

Objective. To use trees to develop a game-playing strategy.

Discussion. In this lab we will write a simulator for the game, Hex-a-Pawn. This game was developed in the early sixties by Martin Gardner. Three white and three black pawns are placed on a 3×3 chessboard. On alternate moves they may be either moved forward one square, or they may capture an opponent on the diagonal. The game ends when a pawn is promoted to the opposite rank, or if a player loses all his pieces, or if no legal move is possible.

In his article in the March 1962 *Scientific American*, Gardner discussed a method for teaching a computer to play this simple game using a relatively small number of training matches. The process involved keeping track of the different states of the board and the potential for success (a win) from each board state. When a move led directly to a loss, the computer forgot the move, thereby causing it to avoid that particular loss in the future. This *pruning* of moves could, of course, cause an intermediate state to lead indirectly to a loss, in which case the computer would be forced to prune out an intermediate move.

Gardner's original "computer" was constructed from matchboxes that contained colored beads. Each bead corresponded to a potential move, and the pruning involved disposing of the last bead played. In a modern system, we can use nodes of a tree stored in a computer to maintain the necessary information about each board state. The degree of each node is determined by the number of possible moves.

Procedure. During the course of this project you are to

1. Construct a tree of Hex-a-Pawn board positions. Each node of the tree is called a `GameTree`. The structure of the class is of your own design, but it is likely to be similar to the `BinaryTree` implementation.

2. Construct three classes of `Players` that play the game of Hex-a-Pawn. These three classes may interact in pairs to play a series of games.

Available for your use are three Javafiles:

`HexBoard` This class describes the state of a board. The default board is the 3×3 starting position. You can ask a board to print itself out (`toString`) or to return the `HexMoves` (`moves`) that are possible from this position. You can also ask a `HexBoard` if the current position is a win for a particular color—`HexBoard.WHITE` or `HexBoard.BLACK`. A static utility method, `opponent`, takes a color and returns the opposite color. The `main` method of this class demonstrates how `HexBoards` are manipulated.

`HexBoard`

`HexMove` This class describes a valid move. The components of the `Vector` returned from the `HexBoard.moves` contains objects of type `HexMove`. Given a `HexBoard` and a `HexMove` one can construct the resulting `HexBoard` using a `HexBoard` constructor.

`HexMove`

Player

`Player` When one is interested in constructing players that play Hex-a-Pawn, the `Player` interface describes the form of the `play` method that must be provided. The `play` method takes a `GameTree` node and an opposing `Player`. It checks for a loss, plays the game according to the `GameTree`, and then turns control over to the opposing player.

Read these class files carefully. You should not expect to modify them.

There are many approaches to experimenting with Hex-a-Pawn. One series of experiments might be the following:

1. Compile `HexBoard.java` and run it as a program. Play a few games against the computer. You may wish to modify the size of the board. Very little is known about the games larger than 3×3.

2. Implement a `GameTree` class. This class should have a constructor that, given a `HexBoard` and a color (a `char`, `HexBoard.WHITE` or `HexBoard.BLACK`), generates the tree of all boards reachable from the specified board position during normal game play. Alternate levels of the tree represent boards that are considered by alternate players. Leaves are winning positions for *the player at hand*. The references to other `GameTree` nodes are suggested by the individual moves returned from the `moves` method. A complete game tree for 3×3 boards has 252 nodes.

608? No win test
370? Wrong win test
150? early stop

3. Implement the first of three players. It should be called `HumanPlayer`. If it hasn't already lost (i.e., if the opponent hasn't won), this player prints the board, presents the moves, and allows a human (through a `ReadStream`) to select a move. The play is then handed off to the opponent.

4. The second player, `RandPlayer`, should play randomly. Make sure you check for a loss before attempting a move.

5. The third player, called `CompPlayer`, should attempt to have the `CompPlayer` object modify the game tree to remove losing moves.

Clearly, `Players` may be made to play against each other in any combination.

Thought Questions. Consider the following questions as you complete the lab:

1. How many board positions are there for the 3×4 board? Can you determine how many moves there are for a 3×5 board?

2. If you implement the learning machine, pit two machines against each other. Gardner called the computer to move first H.I.M., and the machine to move second H.E.R. Will H.I.M. or H.E.R. ultimately win more frequently? Explain your reasoning in a short write-up. What happens for larger boards?

3. In Gardner's original description of the game, each matchbox represented a board state *and its reflection*. What modifications to `HexBoard` and `HexMove` would be necessary to support this collapsing of the game tree?