Chapter Objectives

• Define trees as data structures
• Define the terms associated with trees
• Discuss the possible implementations of trees
• Analyze tree implementations of collections
• Discuss methods for traversing trees
• Examine a binary tree example
Trees

- A *Tree* is a non-linear structure defined by the concept that each *node* in the tree, other than the first node or *root* node, has exactly one parent.

- For trees, the operations are dependent upon the type of tree and it’s use.
Definitions

• In order to discuss trees, we must first have a common vocabulary

• We have already introduced a couple of terms:
  – *node* which refers to a location in the tree where an element is stored, and
  – *root* which refers to the node at the base of the tree or the one node in the tree that does not have a parent
Definitions

- Each node of the tree points to the nodes that are directly beneath it in the tree
- These nodes are referred to as its *children*
- A child of a child is then called a *grandchild*, a child of a grandchild called a *great-grandchild*
- A node that does not have at least one child is called a *leaf*
- A node that is not the root and has at least one child is called an *internal node*
Tree terminology

**FIGURE 9.1** Tree terminology
Definitions

- Any node below another node and on a path from that node is called a *descendant* of that node.
- Any node above another node on a connecting path from the root to that node is called an *ancestor* of that node.
- All children of the same node are called *siblings*.
- A tree that limits each node to no more than $n$ children is called an $n$-ary tree.
Definitions

- Each node of the tree is at a specific *level* or *depth* within the tree.
- The level of a node is the length of the path from the root to the node.
- This *pathlength* is determined by counting the number of links that must be followed to get from the root to the node.
- The root is considered to be level 0, the children of the root are at level 1, the grandchildren of the root are at level 2, and so on.
Definitions

- The *height* or *order* of a tree is the length of the longest path from the root to a leaf.
- Thus the height or order of the tree in the next slide is 3.
- The path from the root (A) to leaf (F) is of length 3.
- The path from the root (A) to leaf (C) is of length 1.
Path length and level

**Figure 9.2** Path length and level
Definitions

- A tree is considered to be *balanced* if all of the leaves of the tree are at roughly the same depth.

- While the use of the term “roughly” may not be intellectually satisfying, the actual definition is dependent upon the algorithm being used.

- Some algorithms define balanced as all of the leaves being at level $h$ or $h-1$ where $h$ is the height of the tree and where $h = \log_N n$ for an N-ary tree.
Balanced and unbalanced trees

**Figure 9.3** Balanced and unbalanced trees
Definitions

• The concept of a complete tree is related to the balance of a tree

• A tree is considered complete if it is balanced and all of the leaves at level $h$ are on the left side of the tree

• While a seemingly arbitrary concept, as we will discuss in later chapters, this definition has implications for how the tree is stored in certain implementations

• Tree a and c on the next slide are complete while tree b is not
Some example trees

**FIGURE 9.4** Some complete trees
Implementing Trees with Links

• While it is not possible to discuss the details of an implementation of a tree without defining the type of tree and its use, we can look at general strategies for implementing trees

• The most obvious implementation of tree is a linked structure

• Each node could be defined as a TreeNode class, as we did with the LinearNode class for linked lists
Implementing Trees with Links

- Each node would contain a pointer to the element to be stored in that node as well as pointers for each of the possible children of the node.

- Depending on the implementation, it may also be useful to store a pointer in each node to its parent.
Implementing Trees with Arrays

- For certain types of trees, specifically binary trees, a computational strategy can be used for storing a tree using an array.

- For any element stored in position $n$ of the array, that element's left child will be stored in position $((2*n) + 1)$ and that element's right child will be stored in position $(2*(n+1))$. 
Implementing Trees with Arrays

- This strategy can be managed in terms of capacity in much the same way that we did for other array-based collections.

- Despite the conceptual elegance of this solution, it is not without drawbacks.

- For example, if the tree that we are storing is not complete or relatively complete, we may be wasting large amounts of memory allocated in the array for positions of the tree that do not contain data.
Computational strategy for array implementation of trees

**Figure 9.5** Computational strategy for array implementation of trees
Implementing Trees with Arrays

- A second possible array implementation of trees is modeled after the way operating systems manage memory.
- Instead of assigning elements of the tree to array position by location in the tree, array positions are allocated contiguously on a first come first served basis.
- Each element of the array will be a node class similar to the TreeNode class that we discussed earlier.
Implementing Trees with Arrays

• However, instead of storing object reference variables as pointers to its children (and perhaps its parent), each node would store the array index of each child (and perhaps its parent)

• This approach allows elements to be stored contiguously in the array so that space is not wasted

• However, this approach increases the overhead for deleting elements in the tree since either remaining elements will have to be shifted to maintain contiguity or a free-list will have to be maintained
Simulated link strategy for array implementation of trees

**Figure 9.6** Simulated link strategy for array implementation of trees
Analysis of Trees

- Trees are a useful and efficient way to implement other collections
- In our analysis of list implementations in Chapter 6, we described the find operation as expected case n/2 or O(n)
- However, if we implemented an ordered list using a balanced binary search tree, a binary tree with the added property that the left child is always less than the parent which is always less than or equal to the right child, then we could improve the efficiency of the find operation to O(log n)
Analysis of Trees

• This is due to the fact that the height or order of such a tree will always be $\log_2 n$ where $n$ is the number of elements in the tree.
• This is very similar to our discussion of binary search in Chapter 8.
• In fact, for any balanced $N$-ary tree with $n$ elements, the tree’s height will be $\log_N n$.
• With the added ordering property of a binary search tree, you are guaranteed to at worst search one path from the root to a leaf.
Tree Traversals

- There are four basic algorithms for traversing a tree:
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal
  - Levelorder traversal
Preorder traversal

- Preorder traversal is accomplished by visiting each node, followed by its children, starting with the root.
- Given the complete binary tree on the next slide, a preorder traversal would produce the order: 
  A  B  D  E  C
A complete tree
Preorder traversal

• Stated in pseudocode, the algorithm for a preorder traversal of a binary tree is:

Visit node
Traverse(left child)
Traverse(right child)
Inorder traversal

- Inorder traversal is accomplished by visiting the left child of the node, then the node, then any remaining child nodes starting with the root

- An inorder traversal of the previous tree produces the order:
  
  D  B  E  A  C
Inorder traversal

- Stated in pseudocode, the algorithm for an inorder traversal of a binary tree is:

  Traverse(left child)
  Visit node
  Traverse(right child)
Postorder traversal

- Postorder traversal is accomplished by visiting the children, then the node starting with the root.
- Given the same tree, a postorder traversal produces the following order:
  
  D  E  B  C  A
Postorder traversal

• Stated in pseudocode, the algorithm for a postorder traversal of a binary tree is:

Traverse(left child)
Traverse(right child)
Visit node
Levelorder traversal

- Levelorder traversal is accomplished by visiting all of the nodes at each level, one level at a time, starting with the root.
- Given the same tree, a levelorder traversal produces the order:
  A  B  C  D  E
Levelorder traversal

- Stated in pseudocode, the algorithm for a level order traversal of a binary tree is:

Create a queue called nodes
Create an unordered list called results
Enqueue the root onto the nodes queue
While the nodes queue is not empty
{
    Dequeue the first element from the queue
    If that element is not null
        Add that element to the rear of the results list
        Enqueue the children of the element on the nodes queue
    Else
        Add null on the result list
}
Return an iterator for the result list
Implementing Binary Trees

• As an example of possible implementations of trees, let's explore a simple implementation of a binary tree.

• Having specified that we are implementing a binary tree, we can identify a set of possible operations that would be common for all binary trees.

• Notice however, that other than the constructors, none of these operations add any elements to the tree.

• It is not possible to define an operation to add an element to the tree until we know more about how the tree is to be used.
The operations on a binary tree

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getRoot</code></td>
<td>Returns a reference to the root of the binary tree</td>
</tr>
<tr>
<td><code>isEmpty</code></td>
<td>Determines if the tree is empty</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Returns the number of elements in the tree</td>
</tr>
<tr>
<td><code>contains</code></td>
<td>Determines if the specified target is in the tree</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Returns a reference to the specified target element if it is found</td>
</tr>
<tr>
<td><code>toString</code></td>
<td>Returns a string representation of the tree</td>
</tr>
<tr>
<td><code>iteratorInOrder</code></td>
<td>Returns an iterator for an inorder traversal of the tree</td>
</tr>
<tr>
<td><code>iteratorPreOrder</code></td>
<td>Returns an iterator for a preorder traversal of the tree</td>
</tr>
<tr>
<td><code>iteratorPostOrder</code></td>
<td>Returns an iterator for a postorder traversal of the tree</td>
</tr>
<tr>
<td><code>iteratorLevelOrder</code></td>
<td>Returns an iterator for a level-order traversal of the tree</td>
</tr>
</tbody>
</table>

**Figure 9.8** The operations on a binary tree
UML description of the BinaryTreeADT interface

FIGURE 9.9 UML description of the BinaryTreeADT interface

© 2010 Pearson Addison-Wesley. All rights reserved.
BinaryTreeADT

/**
 * BinaryTreeADT defines the interface to a binary tree data structure.
 * 
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
	package jss2;
	import java.util.Iterator;
	public interface BinaryTreeADT<T> {
		/**
		 * Returns a reference to the root element
		 * 
		 * @return a reference to the root
		 */
		public T getRoot();
/**
 * Returns true if this binary tree is empty and false otherwise.
 * @return true if this binary tree is empty
 */
public boolean isEmpty();

/**
 * Returns the number of elements in this binary tree.
 * @return the integer number of elements in this tree
 */
public int size();

/**
 * Returns true if the binary tree contains an element that matches
 * the specified element and false otherwise.
 * @param targetElement the element being sought in the tree
 * @return true if the tree contains the target element
 */
public boolean contains (T targetElement);
BinaryTreeADT (continued)

/**
 * Returns a reference to the specified element if it is found in
 * this binary tree. Throws an exception if the specified element
 * is not found.
 *
 * @param targetElement the element being sought in the tree
 * @return a reference to the specified element
 */
public T find (T targetElement);

/**
 * Returns the string representation of the binary tree.
 *
 * @return a string representation of the binary tree
 */
public String toString();

/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with the root.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorInOrder();
BinaryTreeADT (continued)

```java
/**
 * Performs a preorder traversal on this binary tree by calling an
 * overloaded, recursive preorder method that starts with the root.
 * 
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPreOrder();

/**
 * Performs a postorder traversal on this binary tree by calling an
 * overloaded, recursive postorder method that starts with the root.
 * 
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPostOrder();

/**
 * Performs a levelorder traversal on the binary tree, using a queue.
 * 
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorLevelOrder();
```
Expression Trees

• Now let's look at an example using a binary tree

• In Chapter 3, we used a stack to evaluate postfix expressions

• Now we modify that algorithm to construct an expression tree
An example expression tree
Expression Trees

• Our expression tree class will extend the LinkedBinaryTree class discussed later in the chapter.

• This class provides constructors that reference the constructors of the LinkedBinaryTree class.

• The class also provides an evaluate method to recursively evaluate an expression tree once it has been constructed.

• The ExpressionTreeObj class represents the expression tree objects to be stored in the binary tree.
The ExpressionTree class

```java
/**
 * ExpressionTree represents an expression tree of operators and operands.
 * *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

package jss2;

public class ExpressionTree extends LinkedBinaryTree<ExpressionTreeObj> {
    /**
     * Creates an empty expression tree.
     */
    public ExpressionTree()
    {
        super();
    }
```
The ExpressionTree class (continued)

```java
/**
 * Constructs a expression tree from the two specified expression trees.
 *
 * @param element       the expression tree for the center
 * @param leftSubtree   the expression tree for the left subtree
 * @param rightSubtree  the expression tree for the right subtree
 */
public ExpressionTree (ExpressionTreeObj element,
                        ExpressionTree leftSubtree, ExpressionTree rightSubtree)
{
    root = new BinaryTreeNode<ExpressionTreeObj> (element);
    count = 1;

    if (leftSubtree != null)
    {
        count = count + leftSubtree.size();
        root.left = leftSubtree.root;
    }
    else
        root.left = null;
```
The ExpressionTree class (continued)

```java
if (rightSubtree != null)
{
    count = count + rightSubtree.size();
    root.right = rightSubtree.root;
}
else
    root.right = null;

/**
 * Evaluates the expression tree by calling the recursive
 * evaluateNode method.
 *
 * @return the integer evaluation of the tree
 */
public int evaluateTree()
{
    return evaluateNode(root);
}
```
/**
 * Recursively evaluates each node of the tree.
 *
 * @param root the root of the tree to be evaluated
 * @return the integer evaluation of the tree
 */
public int evaluateNode(BinaryTreeNode root)
{
    int result, operand1, operand2;
    ExpressionTreeObj temp;

    if (root==null)
        result = 0;
    else
    {
        temp = (ExpressionTreeObj)root.element;

        if (temp.isOperator())
            {
                operand1 = evaluateNode(root.left);
                operand2 = evaluateNode(root.right);
                result = computeTerm(temp.getOperator(), operand1, operand2);
            }
else
    result = temp.getValue();
}

return result;
}

/**
 * Evaluates a term consisting of an operator and two operands.
 * 
 * @param operator  the operator for the expression
 * @param operand1  the first operand for the expression
 * @param operand2  the second operand for the expression
 */
The ExpressionTree class (continued)

```java
private static int computeTerm(char operator, int operand1, int operand2)
{
    int result=0;

    if (operator == '+')
        result = operand1 + operand2;

    else if (operator == '-')
        result = operand1 - operand2;
    else if (operator == '*')
        result = operand1 * operand2;
    else
        result = operand1 / operand2;

    return result;
}
```
The ExpressionTreeObj class

package jss2;

public class ExpressionTreeObj
{
    private int termType;
    private char operator;
    private int value;

    /**
     * Creates a new expression tree object with the specified data.
     * @param type the integer type of the expression
     * @param op the operand for the expression
     * @param val the value for the expression
     */
The ExpressionTreeObj class (cont.)

```java
public ExpressionTreeObj (int type,char op, int val)
{
    termType = type;
    operator = op;
    value = val;
}

/**
 * Returns true if this object is an operator and false otherwise.
 *
 * @return  true if this object is an operator
 */
public boolean isOperator()
{
    return (termType == 1);
}
```
The ExpressionTreeObj class (cont.)

```java
/**
 * Returns the operator of this expression tree object.
 * @return the character representation of the operator
 */
public char getOperator()
{
    return operator;
}

/**
 * Returns the value of this expression tree object.
 * @return the value of this expression tree object
 */
public int getValue()
{
    return value;
}
```
Building an Expression Tree from a postfix expression
The Postfix and PostfixEvaluator Classes

• The Postfix and PostfixEvaluator classes are modifications of those presented in chapter 3

• The solve method of the PostfixEvaluator class uses a pair of stacks to create an expression tree from a valid post-fix expression

• It then outputs the result using the evaluate method of the ExpressionTree class
The Postfix2 class

/**
 * Postfix2 uses the PostfixEvaluator2 class to solve a postfix expression
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

public class Postfix2
{
/**
 * Uses the PostfixEvaluator class to solve a postfix expression.
 */
 public static void main (String[] args)
 {
    PostfixEvaluator2 temp = new PostfixEvaluator2();
    temp.solve();
 }
}
The PostfixEvaluator2 class

/**
 * PostfixEvaluator2 this modification of our stack example uses a pair of
 * stacks to create an expression tree from a VALID postfix integer expression
 * and then uses a recursive method from the ExpressionTree class to
 * evaluate the tree.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

import jss2.*;
import jss2.exceptions.*;
import java.util.StringTokenizer;
import java.util.Iterator;
import java.io.*;

public class PostfixEvaluator2
{

The PostfixEvaluator2 class (cont.)

```java
/**
 * Retrieves and returns the next operand off of this tree stack.
 * @param treeStack the tree stack from which the operand will be returned
 * @return the next operand off of this tree stack
 */
private ExpressionTree getOperand(LinkedStack<ExpressionTree> treeStack) {
    ExpressionTree temp;
    temp = treeStack.pop();
    return temp;
}

/**
 * Retrieves and returns the next token, either an operator or operand from the user.
 * @return the next string token
 */
```
The PostfixEvaluator2 class (cont.)

private String getNextToken()
{
    String tempToken = "0", inString;
    StringTokenizer tokenizer;

    try
    {
        BufferedReader in =
            new BufferedReader( new InputStreamReader(System.in));
        inString = in.readLine();
        tokenizer = new StringTokenizer(inString);
        tempToken = (tokenizer.nextToken());
    }
    catch (Exception IOException)
    {
        System.out.println("An input/output exception has occurred");
    }

    return tempToken;
}
/**
 * Prompts the user for a valid post-fix expression, converts it to
 * an expression tree using a two stack method, then calls a
 * recursive method to evaluate the expression.
 */

public void solve ()
{
    ExpressionTree operand1, operand2;
    char operator;
    String tempToken;
    LinkedStack<ExpressionTree> treeStack = new LinkedStack<ExpressionTree>();

    System.out.println("Enter a valid post-fix expression one token " +
                      "at a time pressing the enter key after each token");
    System.out.println("Enter an integer, an operator (+,-,*,/) then ! to evaluate ");

    tempToken = getNextToken();
    operator = tempToken.charAt(0);
while (!(operator == '!'))
{
    if ((operator == '+') || (operator == '-') || (operator == '*') ||
        (operator == '/'))
    {
        operand1 = getOperand(treeStack);
        operand2 = getOperand(treeStack);
        treeStack.push(new ExpressionTree
                        (new ExpressionTreeObj(1,operator,0), operand2, operand1));
    }
    else
    {
        treeStack.push(new ExpressionTree (new ExpressionTreeObj
                        (2,' ',Integer.parseInt(tempToken)), null, null));
    }
    tempToken = getNextToken();
    operator = tempToken.charAt(0);
}
System.out.print("The result is ");
System.out.println(((ExpressionTree)treeStack.peek()).evaluateTree());
UML description of the Postfix example

FiguRE 9.12  UML description of the Postfix example
Implementing Binary Trees with Links

- Lets examine a linked implementation of a binary tree
- Our implementation will need to keep track of the node that is the root of the tree as well as the count of elements in the tree

```java
protected int count;
protected BinaryTreeNode<T> root;
```
LinkedBinaryTree

• We will provide two constructors
  – One to create a null tree
  – One to create a tree containing a single element
The LinkedBinaryTree class

```java
/**
 * LinkedBinaryTree implements the BinaryTreeADT interface
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

package jss2;
import java.util.Iterator;
import jss2.exceptions.*;

public class LinkedBinaryTree<T> implements BinaryTreeADT<T>
{
    protected int count;
    protected BinaryTreeNode<T> root;
```
/**
 * Creates an empty binary tree.
 */
public LinkedBinaryTree()
{
    count = 0;
    root = null;
}

/**
 * Creates a binary tree with the specified element as its root.
 * 
 * @param element the element that will become the root of the new binary tree
 */
public LinkedBinaryTree (T element)
{
    count = 1;
    root = new BinaryTreeNode<T> (element);
}
The BinaryTreeNode Class

• We will also need a class to represent each node in the tree
• Since this is a binary tree, we will create a BinaryTreeNode class that contain a reference to the element stored in the node as well as references for each of the children
The BinaryTreeNode class

/**
 * BinaryTreeNode represents a node in a binary tree with a left and
 * right child.
 * 
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

double jss2;

class BinaryTreeNode<T> {
    protected T element;
    protected BinaryTreeNode<T> left, right;
}
/**
 * Creates a new tree node with the specified data.
 * 
 * @param obj  the element that will become a part of the new tree node
 */
BinaryTreeNode (T obj)
{
    element = obj;
    left = null;
    right = null;
}
public int numChildren() {
    int children = 0;

    if (left != null)
        children = 1 + left.numChildren();

    if (right != null)
        children = children + 1 + right.numChildren();

    return children;
}
The LinkedBinaryTree Class

- Lets examine some of the methods of the LinkedBinaryTree Class
- Keep in mind that each node of a tree represents a sub-tree
The find and findagain Methods

• The find method provides an excellent example of the recursion that is possible given the nature of a tree

• We use the private method findagain to support the public find method

• This allows us to distinguish between the original invocation of the method and the subsequent recursive calls
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree. Throws a NoSuchElementException if
 * the specified target element is not found in the binary tree.
 *
 * @param targetElement the element being sought in this tree
 * @return a reference to the specified target
 * @throws ElementNotFoundException if an element not found exception occurs
 */

public T find(T targetElement) throws ElementNotFoundException {
    BinaryTreeNode<T> current = findAgain( targetElement, root );

    if( current == null )
        throw new ElementNotFoundException("binary tree");

    return (current.element);
}
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree.
 *
 * @param targetElement the element being sought in this tree
 * @param next the element to begin searching from
 */
private BinaryTreeNode<T> findAgain(T targetElement,
    BinaryTreeNode<T> next)
{
    if (next == null)
        return null;

    if (next.element.equals(targetElement))
        return next;

    BinaryTreeNode<T> temp = findAgain(targetElement, next.left);
    if (temp == null)
        temp = findAgain(targetElement, next.right);

    return temp;
}
The iteratorInOrder Method

- Like the find method, the iteratorInOrder method uses a private method, inorder, to support recursion.

- The traversals for a tree might be implemented as toString methods or iterators or both.
/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with
 * the root.
 * *
 * @return an in order iterator over this binary tree
 */
public Iterator<T> iteratorInOrder()
{
    ArrayUnorderedList<T> tempList = new ArrayUnorderedList<T>();
inorder (root, tempList);

    return tempList.iterator();
}
/**
 * Performs a recursive inorder traversal.
 * 
 * @param node the node to be used as the root for this traversal
 * @param tempList the temporary list for use in this traversal
 */
protected void inorder (BinaryTreeNode<T> node,
                      ArrayUnorderedList<T> tempList)
{
    if (node != null)
    {
        inorder (node.left, tempList);
        tempList.addToRear(node.element);
        inorder (node.right, tempList);
    }
}
Implementing Binary Trees with Arrays

• Lets examine an array implementation of a binary tree
• Our implementation will need to keep track of the array containing the tree as well as the count of elements in the tree

protected int count;
protected T[ ] tree;
ArrayBinaryTree

• We will provide two constructors
  – One to create a null tree
  – One to create a tree containing a single element
The ArrayBinaryTree class

/**
 * ArrayBinaryTree implements the BinaryTreeADT interface using an array
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
package jss2;
import java.util.Iterator;
import jss2.exceptions.*;

public class ArrayBinaryTree<T> implements BinaryTreeADT<T>
{
    protected int count;
    protected T[] tree;
    private final int capacity = 50;
}
/**
 * Creates an empty binary tree.
 */
public ArrayBinaryTree()
{
    count = 0;
    tree = (T[]) new Object[capacity];
}

/**
 * Creates a binary tree with the specified element as its root.
 *
 * @param element the element which will become the root of the new tree
 */
public ArrayBinaryTree (T element)
{
    count = 1;
    tree = (T[]) new Object[capacity];
    tree[0] = element;
}
The ArrayBinaryTree Class

• Lets examine some of the methods of the ArrayBinaryTree Class
• Like our linked implementation, keep in mind that each node of a tree represents a sub-tree
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree. Throws a NoSuchElementException if
 * the specified target element is not found in the binary tree.
 * 
 * @param targetElement the element being sought in the tree
 * @return true if the element is in the tree
 * @throws ElementNotFoundException if an element not found exception occurs
 */
public T find (T targetElement) throws ElementNotFoundException {
    T temp=null;
    boolean found = false;

    for (int ct=0; ct<count && !found; ct++)
        if (targetElement.equals(tree[ct]))
        {
            found = true;
            temp = tree[ct];
        }

    if (!found)
        throw new ElementNotFoundException("binary tree");

    return temp;
}
The iteratorInOrder Method

• Like our linked example, the iteratorInOrder method uses a private method, inorder, to support recursion

• The traversals for a tree might be implemented as toString methods or iterators or both
public Iterator<T> iteratorInOrder()
{
    ArrayUnorderedList<T> templist = new ArrayUnorderedList<T>();
    inorder (0, templist);
    return templist.iterator();
}
/**
 * Performs a recursive inorder traversal.
 *
 * @param node      the node used in the traversal
 * @param templist  the temporary list used in the traversal
 */

protected void inorder (int node, ArrayUnorderedList<T> templist) {
    if (node < tree.length) {
        if (tree[node] != null) {
            inorder (node*2+1, templist);
            templist.addToRear(tree[node]);
            inorder ((node+1)*2, templist);
        }
    }
}