

Chapter 19

Trees

Chapter Scope

- Trees as data structures
- Tree terminology
- Tree implementations
- Analyzing tree efficiency
- Tree traversals
- Expression trees

Trees

- A *tree* is a non-linear structure in which elements are organized into a hierarchy
- A tree is comprised of a set of *nodes* in which elements are stored and *edges* connect one node to another
- Each node is located on a particular *level*
- There is only one *root* node in the tree

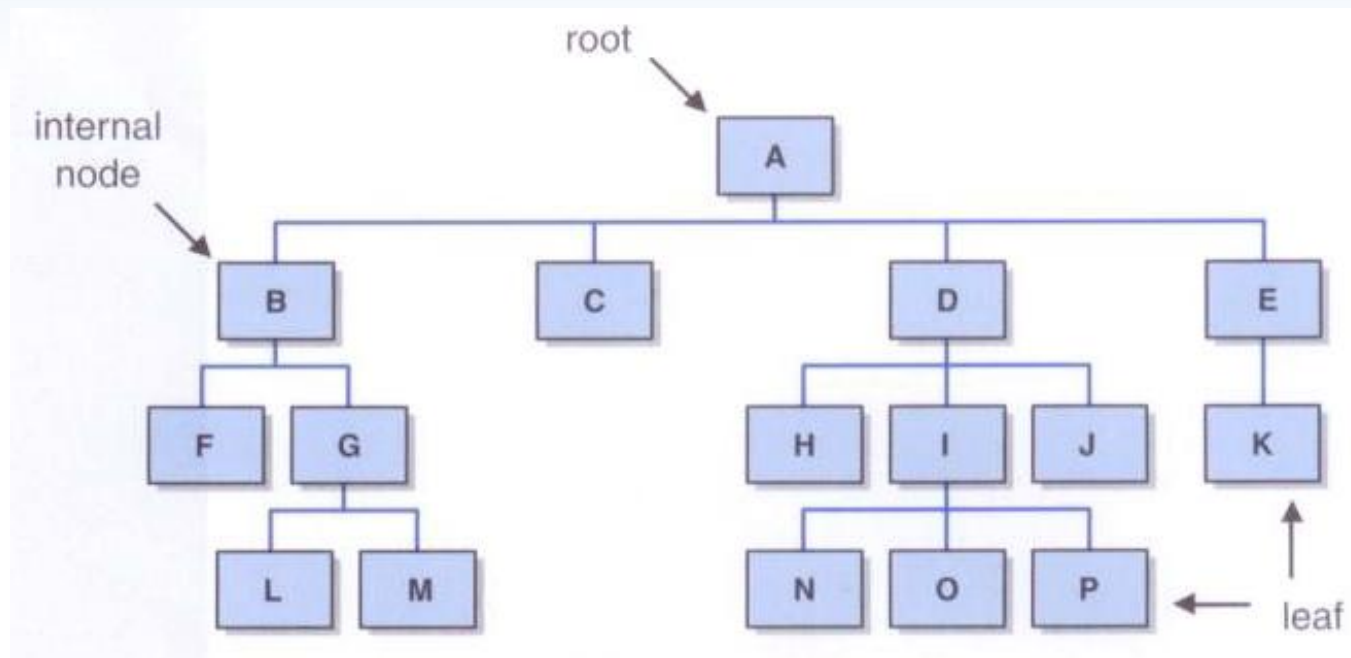
Trees

- Nodes at the lower level of a tree are the *children* of nodes at the previous level
- A node can have only one *parent*, but may have multiple children
- Nodes that have the same parent are *siblings*
- The root is the only node which has no parent

Trees

- A node that has no children is a *leaf* node
- A node that is not the root and has at least one child is an *internal node*
- A *subtree* is a tree structure that makes up part of another tree
- We can follow a *path* through a tree from parent to child, starting at the root
- A node is an *ancestor* of another node if it is above it on the path from the root.

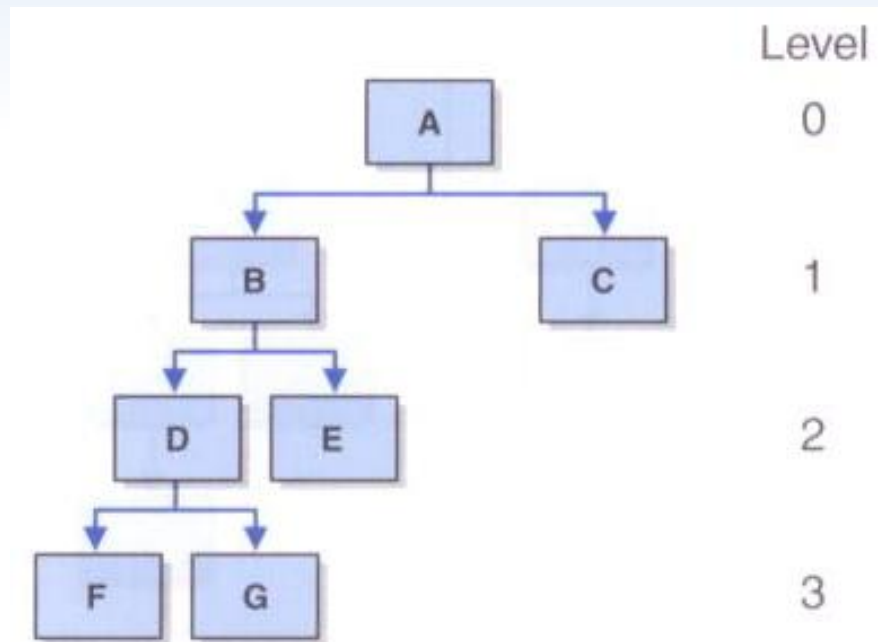
Trees



Trees

- Nodes that can be reached by following a path from a particular node are the *descendants* of that node
- The *level* of a node is the length of the path from the root to the node
- The *path length* is the number of edges followed to get from the root to the node
- The *height* of a tree is the length of the longest path from the root to a leaf

Trees

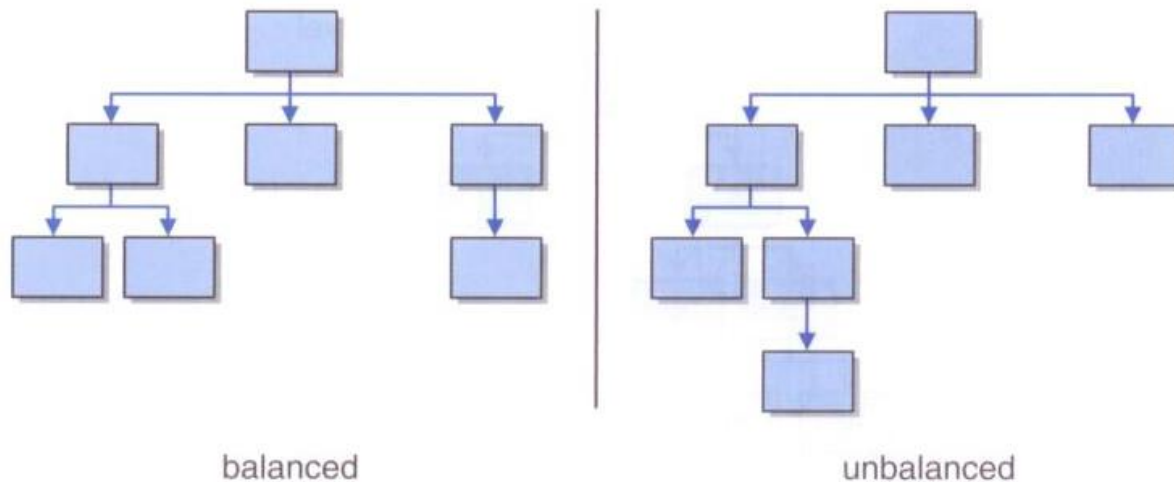


Classifying Trees

- Trees can be classified in many ways
- One important criterion is the maximum number of children any node in the tree may have
- This may be referred to as the *order of the tree*
- *General trees* have no limit to the number of children a node may have
- A tree that limits each node to no more than n children is referred to as an *n -ary tree*

Balanced Trees

- Trees in which nodes may have at most two children are called *binary trees*
- A tree is *balanced* if all of the leaves of the tree are on the same level or within one level of each other

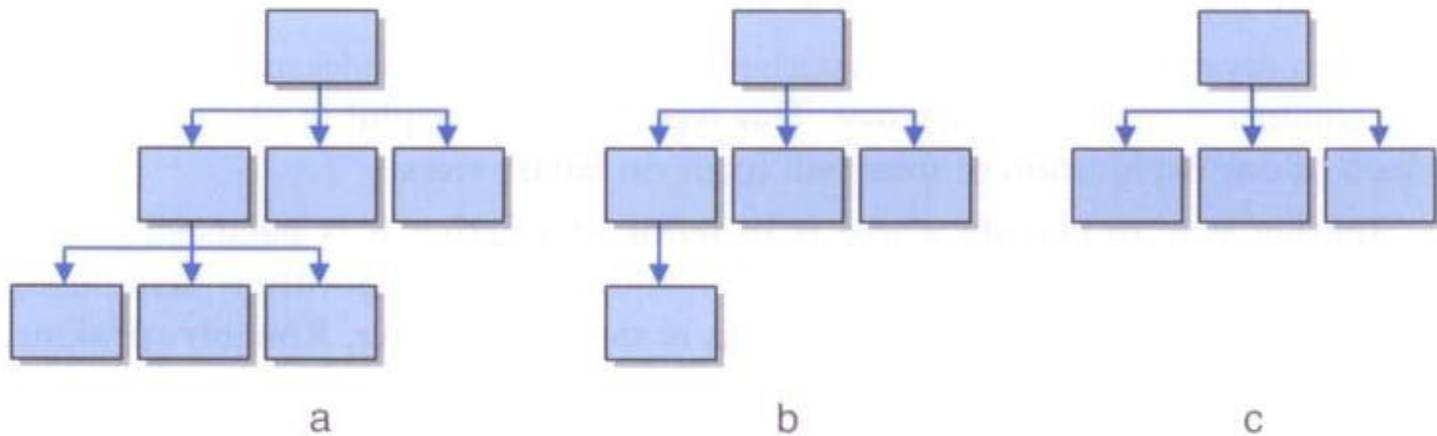


Full and Complete Trees

- A balanced n-ary tree with m elements will have a height of $\log_n m$
- A balanced binary tree with n nodes has a height of $\log_2 n$
- An n-ary tree is *full* if all leaves of the tree are at the same height and every non-leaf node has exactly n children
- A tree is *complete* if it is full, or full to the next-to-last level with all leaves at the bottom level on the left side of the tree

Full and Complete Trees

- Three complete trees:



- Only tree c is full

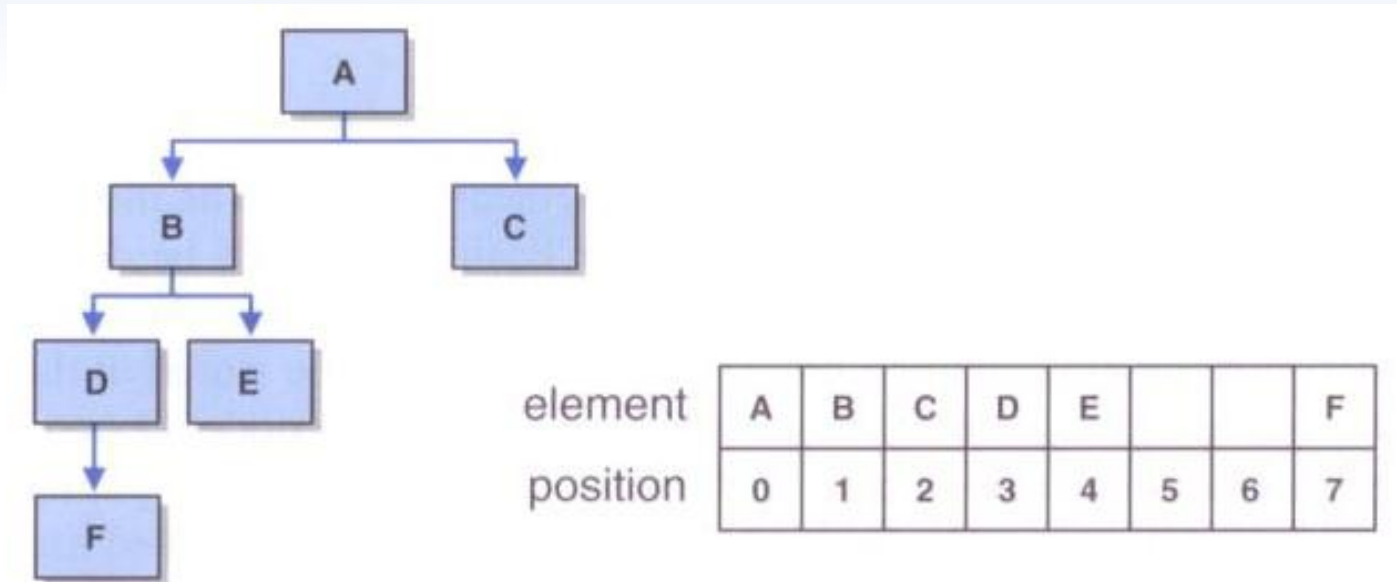
Implementing Trees

- An obvious choice for implementing trees is a linked structure
- Array-based implementations are the less obvious choice, but are sometimes useful
- Let's first look at the strategy behind two array-based implementations

Computed Child Links

- For full or complete binary trees, we can use an array to represent a tree
- For any element stored in position n ,
 - the element's left child is stored in array position $(2n+1)$
 - the element's right child is stored in array position $(2*(n+1))$
- If the represented tree is not complete or relatively complete, this approach can waste large amounts of array space

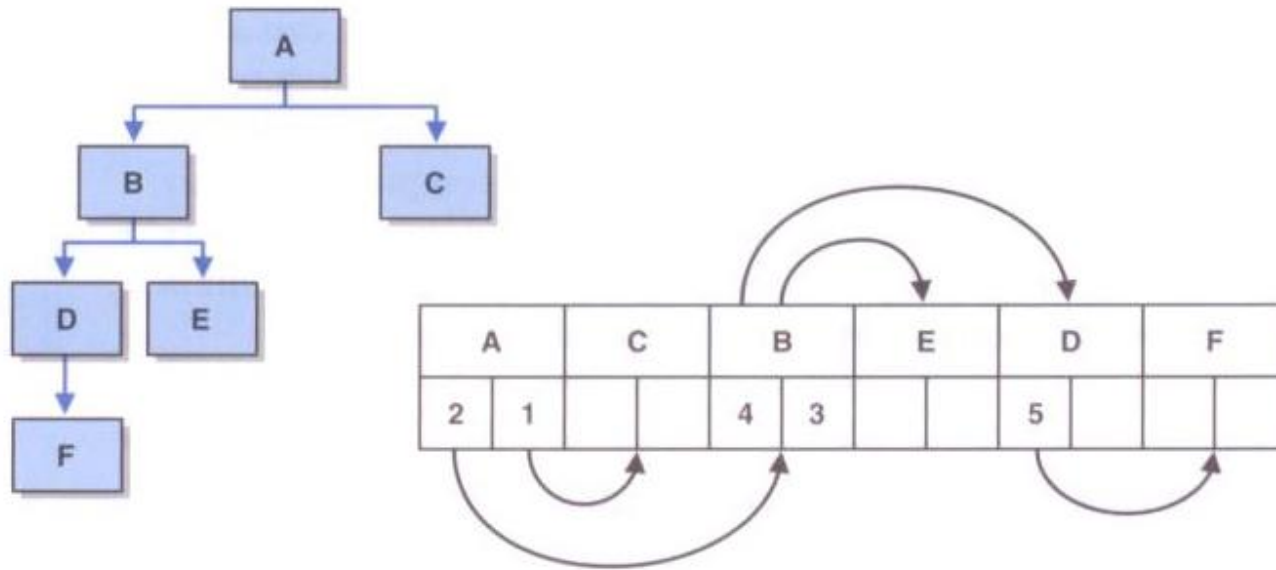
Computed Child Links



Simulated Child Links

- Each element of the array is an object that stores a reference to the tree element and the array index of each child
- This approach is modeled after the way operating systems manage memory
- Array positions are allocated on a first-come, first-served basis

Simulated Child Links



Tree Traversals

- For linear structures, the process of iterating through the elements is fairly obvious (forwards or backwards)
- For non-linear structures like a tree, the possibilities are more interesting
- Let's look at four classic ways of *traversing* the nodes of a tree
- All traversals start at the root of the tree
- Each node can be thought of as the root of a subtree

Tree Traversals

- *Preorder*: visit the root, then traverse the subtrees from left to right
- *Inorder*: traverse the left subtree, then visit the root, then traverse the right subtree
- *Postorder*: traverse the subtrees from left to right, then visit the root
- *Level-order*: visit each node at each level of the tree from top (root) to bottom and left to right

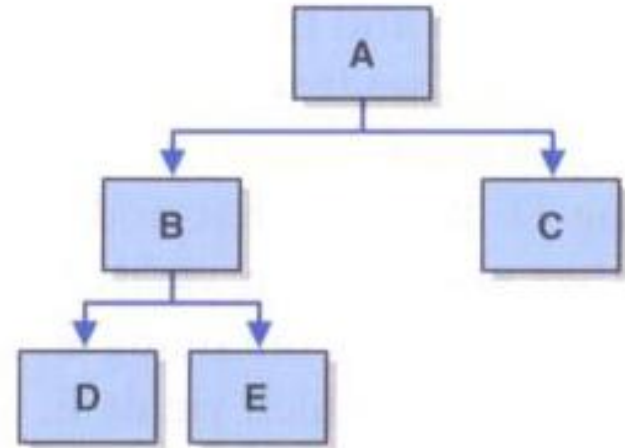
Tree Traversals

Preorder: A B D E C

Inorder: D B E A C

Postorder: D E B C A

Level-Order: A B C D E



Tree Traversals

- Recursion simplifies the implementation of tree traversals
- Preorder (pseudocode):

Visit node

Traverse (left child)

Traverse (right child)

- Inorder:

Traverse (left child)

Visit node

Traverse (right child)

Tree Traversals

- Postorder:

```
    Traverse (left child)
```

```
    Traverse (right child)
```

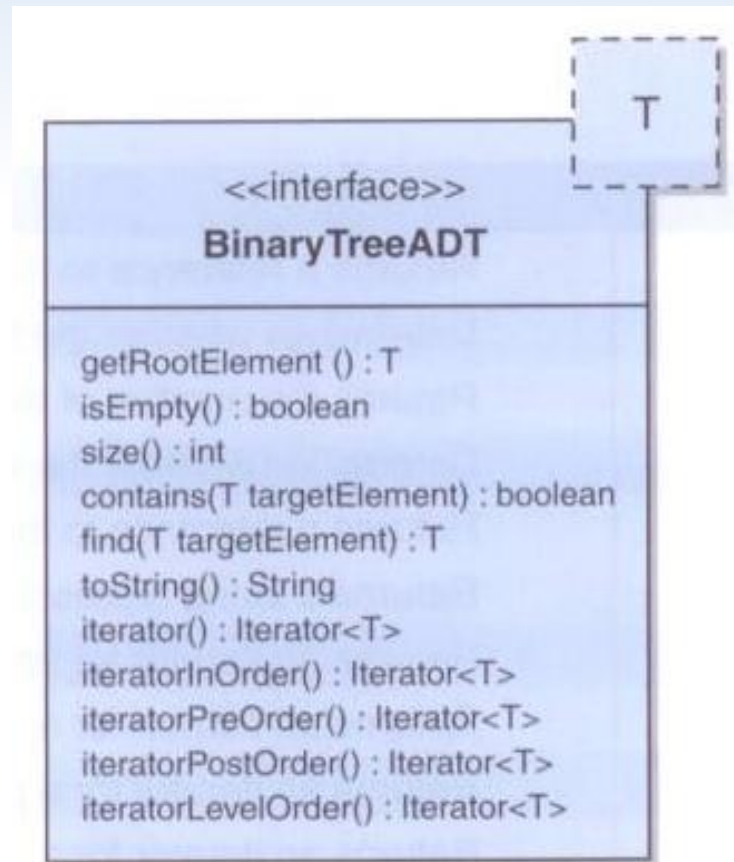
```
    Visit node
```

- A level-order traversal is more complicated
- It requires the use of extra structures (such as queues and/or lists) to create the necessary order

A Binary Tree ADT

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

A Binary Tree ADT




```

package jsjf;

import java.util.Iterator;

/**
 * BinaryTreeADT defines the interface to a binary tree data structure.
 *
 * @author Java Foundations
 * @version 4.0
 */
public interface BinaryTreeADT<T>
{
    /**
     * Returns a reference to the root element
     *
     * @return a reference to the root
     */
    public T getRootElement();

    /**
     * Returns true if this binary tree is empty and false otherwise.
     *
     * @return true if this binary tree is empty, false otherwise
     */
    public boolean isEmpty();
}

```

```

/**
 * Returns the number of elements in this binary tree.
 *
 * @return the number of elements in the 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);

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

/**
 * Returns an iterator over the elements of this tree.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iterator();

/**
 * Returns an iterator that represents an inorder traversal on this binary tree.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorInOrder();

/**
 * Returns an iterator that represents a preorder traversal on this binary tree.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPreOrder();

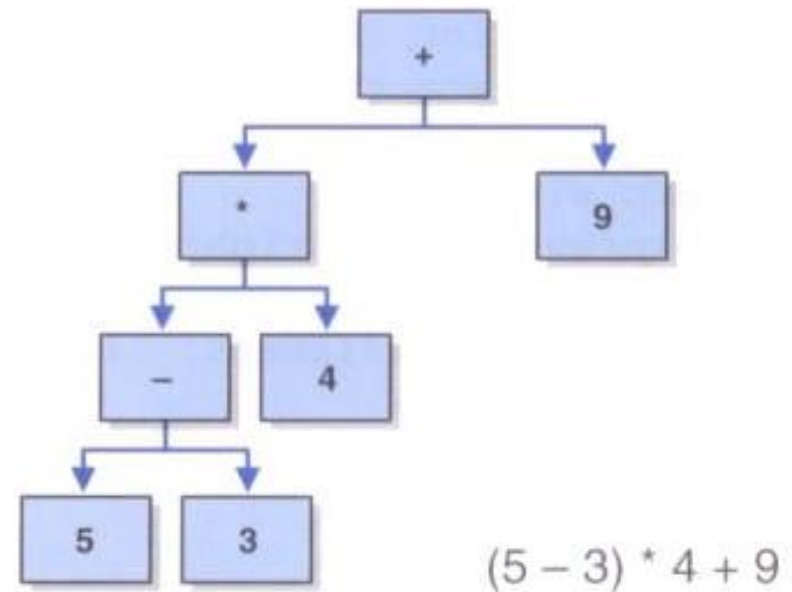
```

```
/**
 * Returns an iterator that represents a postorder traversal on this binary tree.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPostOrder();

/**
 * Returns an iterator that represents a levelorder traversal on the binary tree.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorLevelOrder();
}
```

Expression Trees

- An *expression tree* is a tree that shows the relationships among operators and operands in an expression
- An expression tree is evaluated from the bottom up



```
import jsjf.*;

/**
 * ExpressionTree represents an expression tree of operators and operands.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class ExpressionTree extends LinkedBinaryTree<ExpressionTreeOp>
{
    /**
     * Creates an empty expression tree.
     */
    public ExpressionTree()
    {
        super();
    }
}
```

```

/**
 * 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(ExpressionTreeOp element,
                      ExpressionTree leftSubtree, ExpressionTree rightSubtree)
{
    root = new BinaryTreeNode<ExpressionTreeOp>(element, leftSubtree, rightSubtree);
}

/**
 * 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;
    ExpressionTreeOp temp;

    if (root==null)
        result = 0;
    else
    {
        temp = (ExpressionTreeOp)root.getElement();

        if (temp.isOperator())
        {
            operand1 = evaluateNode(root.getLeft());
            operand2 = evaluateNode(root.getRight());
            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
 */
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;
}

```

```

/**
 * Generates a structured string version of the tree by performing
 * a levelorder traversal.
 *
 * @return a string representation of this binary tree
 */
public String printTree()
{
    UnorderedListADT<BinaryTreeNode<ExpressionTreeOp>> nodes =
        new ArrayUnorderedList<BinaryTreeNode<ExpressionTreeOp>>();
    UnorderedListADT<Integer> levelList =
        new ArrayUnorderedList<Integer>();
    BinaryTreeNode<ExpressionTreeOp> current;
    String result = "";
    int printDepth = this.getHeight();
    int possibleNodes = (int)Math.pow(2, printDepth + 1);
    int countNodes = 0;

    nodes.addToRear(root);
    Integer currentLevel = 0;
    Integer previousLevel = -1;
    levelList.addToRear(currentLevel);

```

```

while (countNodes < possibleNodes)
{
    countNodes = countNodes + 1;
    current = nodes.removeFirst();
    currentLevel = levelList.removeFirst();
    if (currentLevel > previousLevel)
    {
        result = result + "\n\n";
        previousLevel = currentLevel;
        for (int j = 0; j < ((Math.pow(2, (printDepth - currentLevel))) - 1); j++)
            result = result + " ";
    }
    else
    {
        for (int i = 0; i < ((Math.pow(2, (printDepth - currentLevel + 1)) - 1)) ;
i++)
        {
            result = result + " ";
        }
    }
    if (current != null)
    {
        result = result + (current.getElement()).toString();
        nodes.addToRear(current.getLeft());
        levelList.addToRear(currentLevel + 1);
        nodes.addToRear(current.getRight());
        levelList.addToRear(currentLevel + 1);
    }
}

```

```
        else
        {
            nodes.addToRear(null);
            levelList.addToRear(currentLevel + 1);
            nodes.addToRear(null);
            levelList.addToRear(currentLevel + 1);
            result = result + " ";
        }

    }

    return result;
}
}
```

```

import jsjf.*;

/**
 * ExpressionTreeOp represents an element in an expression tree.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class ExpressionTreeOp
{
    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
     */
    public ExpressionTreeOp(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, false otherwise
 */
public boolean isOperator()
{
    return (termType == 1);
}

/**
 *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;
}


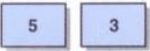
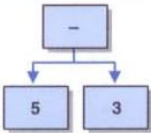
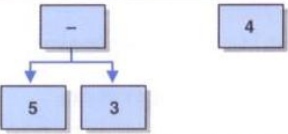
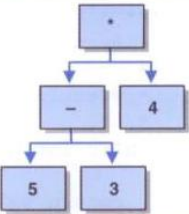
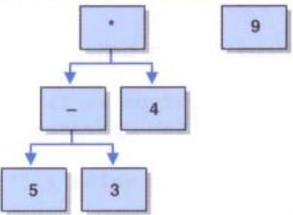
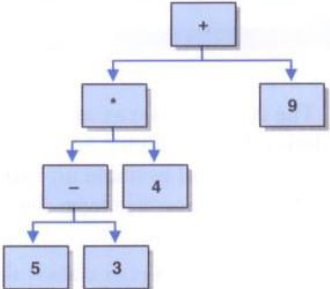
```

```
public String toString()
{
    if (termType == 1)
        return operator + "";
    else
        return value + "";
}
}
```

Expression Tree Evaluation

- Let's look at an example that creates an expression tree from a postfix expression and then evaluates it
- This is a modification of an earlier solution
- It uses a stack of expression trees to build the complete expression tree

Input in Postfix: 5 3 - 4 * 9 +

Token	Processing Steps	Expression Tree Stack (top at right)
5	push(new ExpressionTree(5, null, null))	
3	push(new ExpressionTree(3, null, null))	
-	op2 = pop op1 = pop push(new ExpressionTree(-, op1, op2))	
4	push(new ExpressionTree(4, null, null))	
*	op2 = pop op1 = pop push(new ExpressionTree(*, op1, op2))	
9	push(new ExpressionTree(9, null, null))	
+	op2 = pop op1 = pop push(new ExpressionTree(+, op1, op2))	

```

import java.util.Scanner;

/**
 * Demonstrates the use of an expression tree to evaluate postfix expressions.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class PostfixTester
{
    /**
     * Reads and evaluates multiple postfix expressions.
     */
    public static void main(String[] args)
    {
        String expression, again;
        int result;

        Scanner in = new Scanner(System.in);

        do
        {
            PostfixEvaluator evaluator = new PostfixEvaluator();
            System.out.println("Enter a valid post-fix expression one token " +
                               "at a time with a space between each token (e.g. 5 4 + 3 2 1 - + *)");
            System.out.println("Each token must be an integer or an operator (+,-,*,/)");
            expression = in.nextLine();

```

```
        result = evaluator.evaluate(expression);
        System.out.println();
        System.out.println("That expression equals " + result);

        System.out.println("The Expression Tree for that expression is: ");
        System.out.println(evaluator.getTree());

        System.out.print("Evaluate another expression [Y/N]? ");
        again = in.nextLine();
        System.out.println();
    }
    while (again.equalsIgnoreCase("y"));
}
```

```

import jsjf.*;
import jsjf.exceptions.*;
import java.util.*;
import java.io.*;

/**
 * PostfixEvaluator this modification of our stack example uses a
 * stack to create an expression tree from a VALID integer postfix expression
 * and then uses a recursive method from the ExpressionTree class to
 * evaluate the tree.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class PostfixEvaluator
{
    private String expression;
    private Stack<ExpressionTree> treeStack;

    /**
     * Sets up this evaluator by creating a new stack.
     */
    public PostfixEvaluator()
    {
        treeStack = new Stack<ExpressionTree>();
    }
}

```

```

/**
 * 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(Stack<ExpressionTree> treeStack)
{
    ExpressionTree temp;
    temp = treeStack.pop();

    return temp;
}

/**
 * Evaluates the specified postfix expression by building and evaluating
 * an expression tree.
 *
 * @param expression string representation of a postfix expression
 * @return value of the given expression
 */
public int evaluate(String expression)
{
    ExpressionTree operand1, operand2;
    char operator;
    String tempToken;

    Scanner parser = new Scanner(expression);

```

```

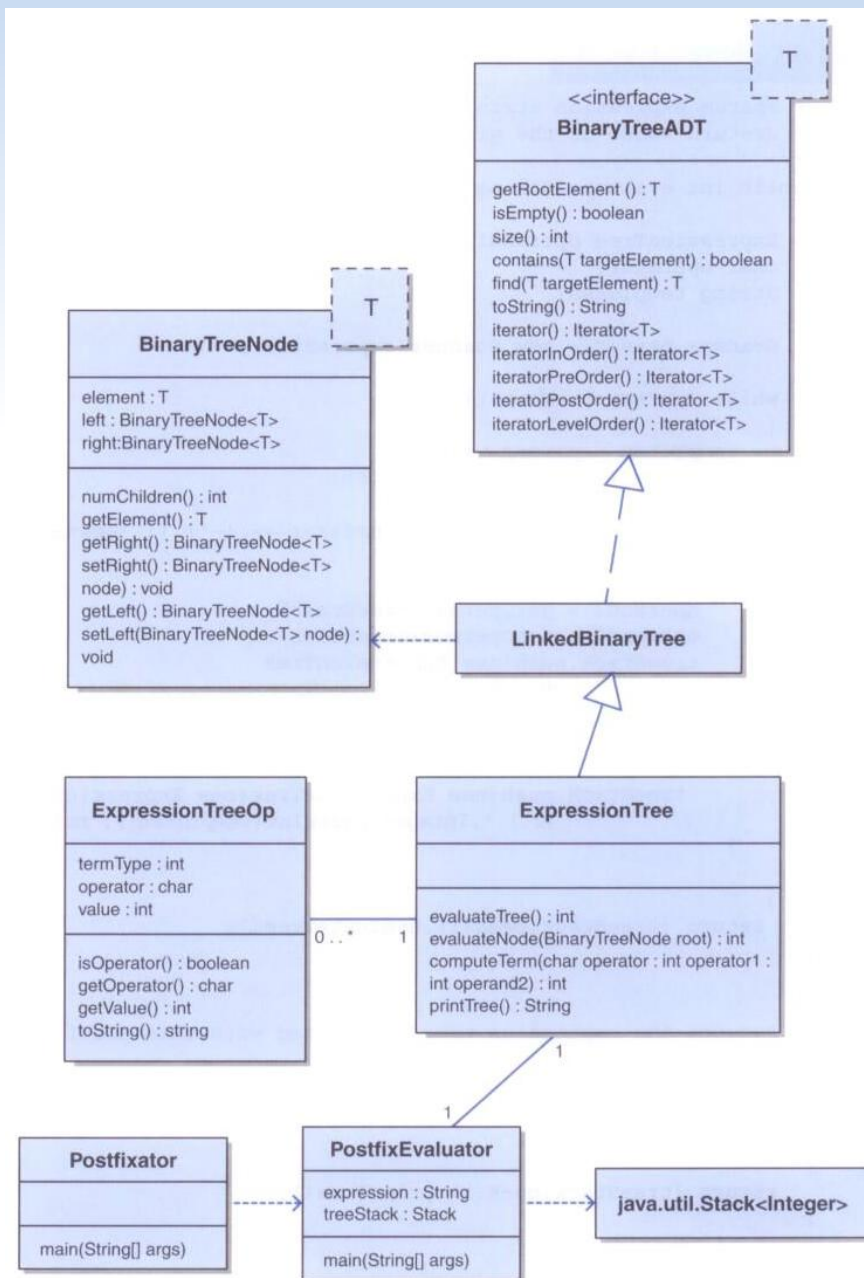
while (parser.hasNext())
{
    tempToken = parser.next();
    operator = tempToken.charAt(0);

    if ((operator == '+') || (operator == '-') || (operator == '*') ||
        (operator == '/'))
    {
        operand1 = getOperand(treeStack);
        operand2 = getOperand(treeStack);
        treeStack.push(new ExpressionTree
                        (new ExpressionTreeOp(1,operator,0), operand2, operand1));
    }
    else
    {
        treeStack.push(new ExpressionTree(new ExpressionTreeOp
                        (2,' ',Integer.parseInt(tempToken)), null, null));
    }

}
return (treeStack.peek()).evaluateTree();
}

/**
 * Returns the expression tree associated with this postfix evaluator.
 *
 * @return string representing the expression tree
 */
public String getTree()
{
    return (treeStack.peek()).printTree();
}
}

```

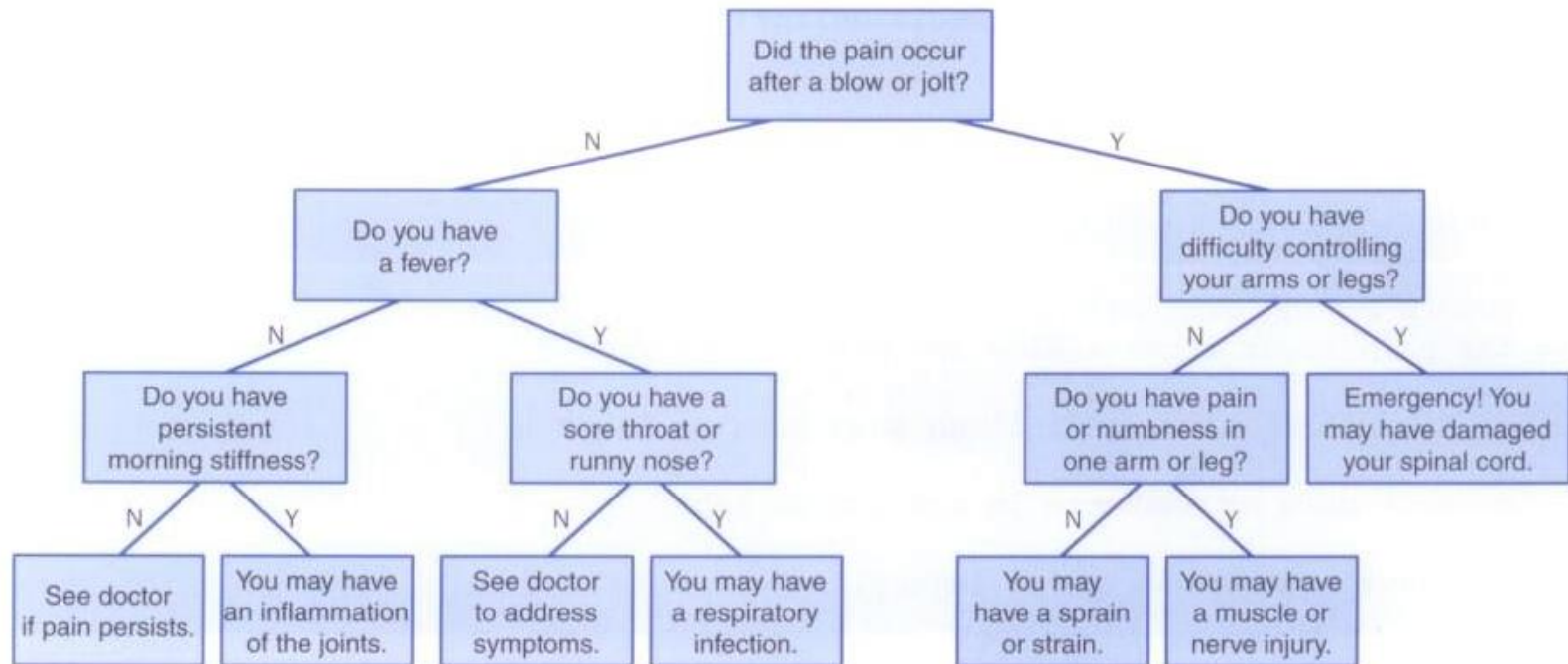


Decision Trees

- A *decision tree* is a tree whose nodes represent decision points, and whose children represent the options available
- The leaves of a decision tree represent the possible conclusions that might be drawn
- A simple decision tree, with yes/no questions, can be modeled by a binary tree
- Decision trees are useful in diagnostic situations (medical, car repair, etc.)

Decision Trees

- A simplified decision tree for diagnosing back pain:



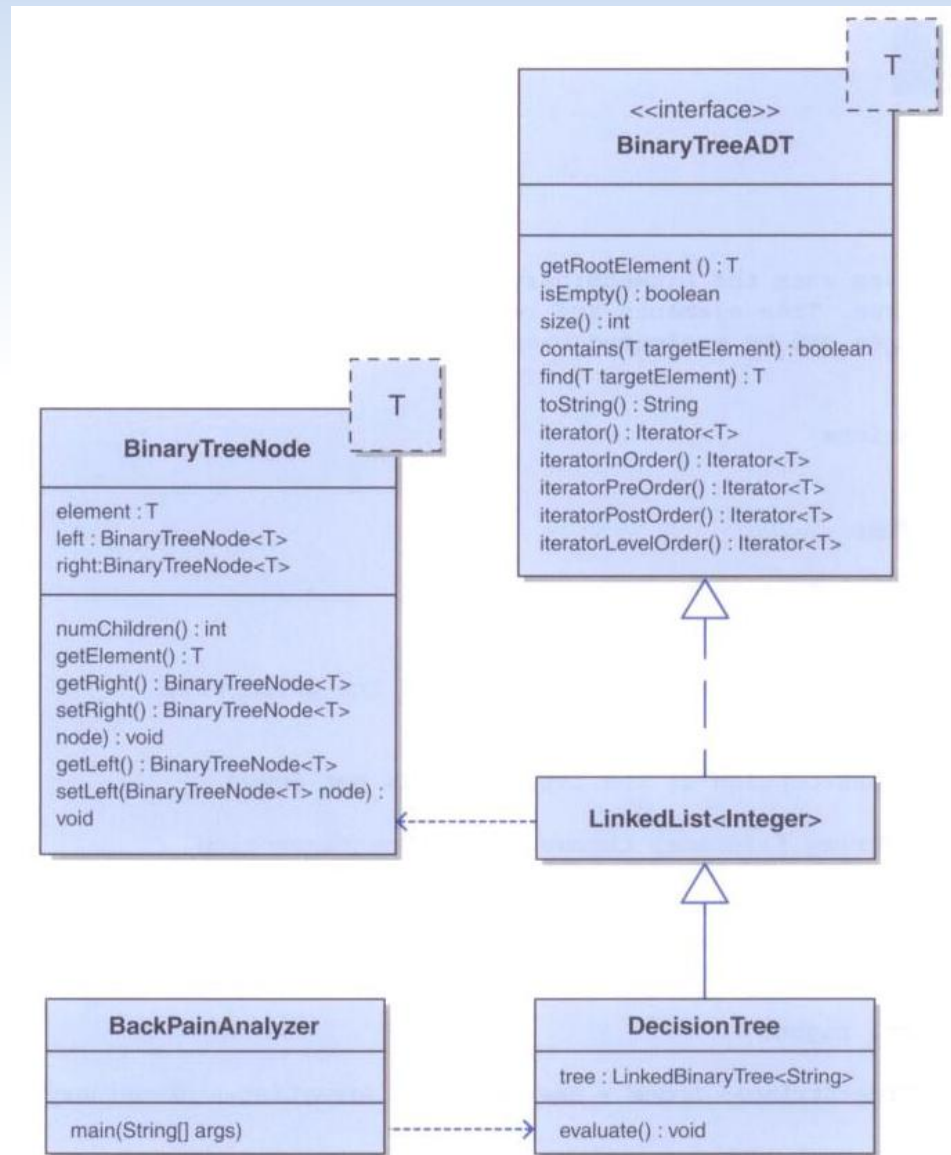
```

import java.io.*;

/**
 * BackPainAnalyzer demonstrates the use of a binary decision tree to
 * diagnose back pain.
 */
public class BackPainAnalyzer
{
    /**
     * Asks questions of the user to diagnose a medical problem.
     */
    public static void main (String[] args) throws FileNotFoundException
    {
        System.out.println ("So, you're having back pain.");

        DecisionTree expert = new DecisionTree("input.txt");
        expert.evaluate();
    }
}

```



```

import jsjf.*;
import java.util.*;
import java.io.*;

/**
 * The DecisionTree class uses the LinkedBinaryTree class to implement
 * a binary decision tree. Tree elements are read from a given file and
 * then the decision tree can be evaluated based on user input using the
 * evaluate method.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class DecisionTree
{
    private LinkedBinaryTree<String> tree;

    /**
     * Builds the decision tree based on the contents of the given file
     *
     * @param filename the name of the input file
     * @throws FileNotFoundException if the input file is not found
     */
    public DecisionTree(String filename) throws FileNotFoundException
    {
        File inputFile = new File(filename);
        Scanner scan = new Scanner(inputFile);
    }

```

```

int numberNodes = scan.nextInt();
scan.nextLine();
int root = 0, left, right;

List<LinkedExceptionTree<String>> nodes = new
    java.util.ArrayList<LinkedExceptionTree<String>>();
for (int i = 0; i < numberNodes; i++)
    nodes.add(i, new LinkedExceptionTree<String>(scan.nextLine()));

while (scan.hasNext())
{
    root = scan.nextInt();
    left = scan.nextInt();
    right = scan.nextInt();
    scan.nextLine();

    nodes.set(root, new LinkedExceptionTree<String>((nodes.get(root)).getRootElement(),
                                                    nodes.get(left), nodes.get(right)));
}
tree = nodes.get(root);
}

```

```

/**
 * Follows the decision tree based on user responses.
 */
public void evaluate()
{
    LinkedBinaryTree<String> current = tree;
    Scanner scan = new Scanner(System.in);

    while (current.size() > 1)
    {
        System.out.println (current.getRootElement());
        if (scan.nextLine().equalsIgnoreCase("N"))
            current = current.getLeft();
        else
            current = current.getRight();
    }

    System.out.println (current.getRootElement());
}
}

```

Implementing Binary Trees with Links

- Now let's explore an implementation of a binary tree using links
- The `LinkedBinaryTree` class holds a reference to the root node
- The `BinaryTreeNode` class represents each node, with links to a possible left and/or right child

```

package jsjf;

/**
 * BinaryTreeNode represents a node in a binary tree with a left and
 * right child.
 *
 * @author Java Foundations
 * @version 4.0
 */
public 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
     */
    public BinaryTreeNode(T obj)
    {
        element = obj;
        left = null;
        right = null;
    }
}

```



```

/**
 * Creates a new tree node with the specified data.
 *
 * @param obj the element that will become a part of the new tree node
 * @param left the tree that will be the left subtree of this node
 * @param right the tree that will be the right subtree of this node
 */
public BinaryTreeNode(T obj, LinkedBinaryTree<T> left, LinkedBinaryTree<T> right)
{
    element = obj;
    if (left == null)
        this.left = null;
    else
        this.left = left.getRootNode();

    if (right == null)
        this.right = null;
    else
        this.right = right.getRootNode();
}

```

```

/**
 * Returns the number of non-null children of this node.
 *
 * @return the integer number of non-null children of this node
 */
public int numChildren()
{
    int children = 0;

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

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

    return children;
}

/**
 * Return the element at this node.
 *
 * @return the element stored at this node
 */
public T getElement()
{
    return element;
}

```

```

/**
 * Return the right child of this node.
 *
 * @return the right child of this node
 */
public BinaryTreeNode<T> getRight()
{
    return right;
}

/**
 * Sets the right child of this node.
 *
 * @param node the right child of this node
 */
public void setRight(BinaryTreeNode<T> node)
{
    right = node;
}

```

```

/**
 * Return the left child of this node.
 *
 * @return the left child of the node
 */
public BinaryTreeNode<T> getLeft()
{
    return left;
}

/**
 * Sets the left child of this node.
 *
 * @param node the left child of this node
 */
public void setLeft(BinaryTreeNode<T> node)
{
    left = node;
}
}

```

```

package jsjf;

import java.util.*;
import jsjf.exceptions.*;

/**
 * LinkedBinaryTree implements the BinaryTreeADT interface
 *
 * @author Java Foundations
 * @version 4.0
 */
public class LinkedBinaryTree<T> implements BinaryTreeADT<T>, Iterable<T>
{
    protected BinaryTreeNode<T> root;
    protected int modCount;

    /**
     * Creates an empty binary tree.
     */
    public LinkedBinaryTree()
    {
        root = null;
    }
}

```

```

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

/**
 * Creates a binary tree with the specified element as its root and the
 * given trees as its left child and right child
 *
 * @param element the element that will become the root of the binary tree
 * @param left the left subtree of this tree
 * @param right the right subtree of this tree
 */
public LinkedBinaryTree(T element, LinkedBinaryTree<T> left,
                        LinkedBinaryTree<T> right)
{
    root = new BinaryTreeNode<T>(element);
    root.setLeft(left.root);
    root.setRight(right.root);
}

```

```

/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree. Throws a ElementNotFoundException 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 the element is not in the tree
 */
public T find(T targetElement) throws ElementNotFoundException
{
    BinaryTreeNode<T> current = findNode(targetElement, root);

    if (current == null)
        throw new ElementNotFoundException("LinkedBinaryTree");

    return (current.getElement());
}

```

```

/**
 * 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> findNode(T targetElement,
                                   BinaryTreeNode<T> next)
{
    if (next == null)
        return null;

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

    BinaryTreeNode<T> temp = findNode(targetElement, next.getLeft());

    if (temp == null)
        temp = findNode(targetElement, next.getRight());

    return temp;
}

```



```

/**
 * 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 new TreeIterator(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.getLeft(), tempList);
        tempList.addToRear(node.getElement());
        inOrder(node.getRight(), tempList);
    }
}

```