



Northeastern
University

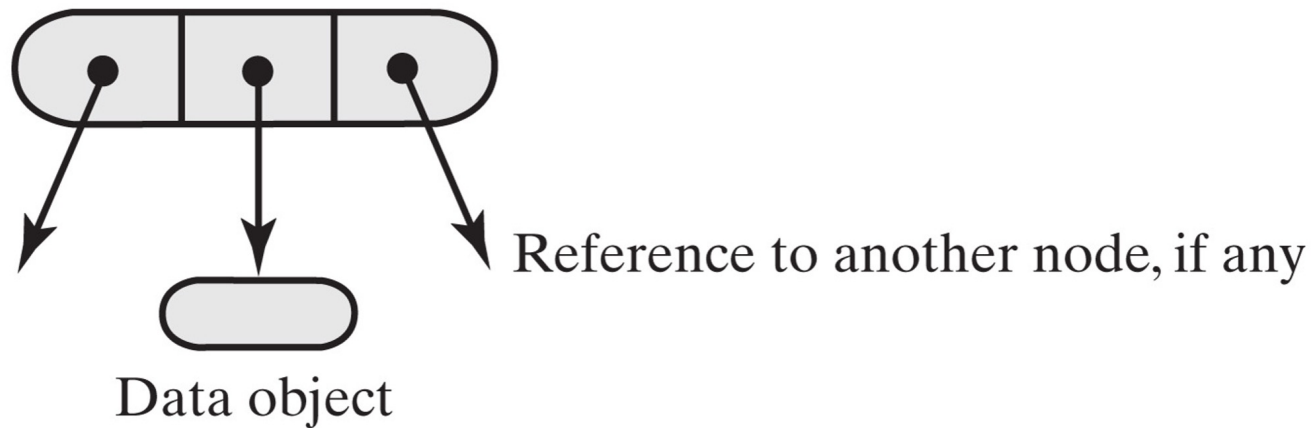
Lecture 31: Trees Implementation

Prof. Chen-Hsiang (Jones) Yu, Ph.D.
College of Engineering

Materials are edited by Prof. Jones Yu from

Data Structures and Abstractions with Java, 5th edition. By Frank M. Carrano and Timothy M. Henry.
ISBN-13 978-0-13-483169-5 © 2019 Pearson Education, Inc.

Node in a Binary Tree



A node in a binary tree

Binary Tree Node - 1

```
package TreePackage;
/** A class that represents nodes in a binary tree. */
class BinaryNode<T>
{
    private T          data;
    private BinaryNode<T> leftChild; // Reference to left child
    private BinaryNode<T> rightChild; // Reference to right child

    public BinaryNode()
    {
        this(null); // Call next constructor
    } // end default constructor

    public BinaryNode(T dataPortion)
    {
        this(dataPortion, null, null); // Call next constructor
    } // end constructor

    public BinaryNode(T dataPortion, BinaryNode<T> newLeftChild,
                      BinaryNode<T> newRightChild)
    {
        data = dataPortion;
        leftChild = newLeftChild;
        rightChild = newRightChild;
    } // end constructor
}
```

The class BinaryNode

Binary Tree Node - 2

```
/** Retrieves the data portion of this node.
    @return The object in the data portion of the node. */
public T getData(). {
    return data;
} // end getData

/** Sets the data portion of this node.
    @param newData The data object. */
public void setData(T newData) {
    data = newData;
} // end setData

/** Retrieves the left child of this node.
    @return A reference to this node's left child. */
public BinaryNode<T> getLeftChild()
{
    return leftChild;
} // end getLeftChild

/** Sets this node's left child to a given node.
    @param newLeftChild A node that will be the left child. */
public void setLeftChild(BinaryNode<T> newLeftChild)
{
    leftChild = newLeftChild;
} // end setLeftChild
```

The class BinaryNode

Binary Tree Node - 3

```
/** Detects whether this node has a left child.
    @return True if the node has a left child. */
public boolean hasLeftChild()
{
    return leftChild != null;
} // end hasLeftChild

/* Implementations of getRightChild, setRightChild, and hasRightChild
   are here and are analogous to their left-child counterparts. */

/** Detects whether this node is a leaf.
    @return True if the node is a leaf. */
public boolean isLeaf()
{
    return (leftChild == null) && (rightChild == null);
} // end isLeaf

/** Counts the nodes in the subtree rooted at this node.
    @return The number of nodes in the subtree rooted at this node. */
public int getNumberOfNodes()
{
    // < Coming later — See Segment 25.10. >
} // end getNumberOfNodes
```

The class BinaryNode

Binary Tree Node - 4

```
/** Computes the height of the subtree rooted at this node.
    @return The height of the subtree rooted at this node. */
public int getHeight()
{
    // < Coming later — See Segment 25.10. >
} // end getHeight

/** Copies the subtree rooted at this node.
    @return The root of a copy of the subtree rooted at this node. */
public BinaryNode<T> copy()
{
    // < Coming later — See Segment 25.5. >
} // end copy
} // end BinaryNode
```

The class BinaryNode

Interface for a Basic Binary Tree

```
package TreePackage;
/** An interface for the ADT binary tree. */
public interface BinaryTreeInterface<T> extends TreeInterface<T>,
                                              TreeIteratorInterface<T>
{
    /** Sets the data in the root of this binary tree.
        @param rootData The object that is the data for the tree's root.
    */
    public void setRootData(T rootData);

    /** Sets this binary tree to a new binary tree.
        @param rootData The object that is the data for the new tree's root.
        @param leftTree The left subtree of the new tree.
        @param rightTree The right subtree of the new tree. */
    public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
                      BinaryTreeInterface<T> rightTree);
} // end BinaryTreeInterface
```

Interface for a class of binary trees

Creating a Basic Binary Tree - 1

```
package TreePackage;
import java.util.Iterator;
import java.util.NoSuchElementException;
import StackAndQueuePackage.*; // Needed by tree iterators
/** A class that implements the ADT binary tree. */
public class BinaryTree<T> implements BinaryTreeInterface<T>
{
    private BinaryNode<T> root;

    public BinaryTree()
    {
        root = null;
    } // end default constructor

    public BinaryTree(T rootData)
    {
        root = new BinaryNode<>(rootData);
    } // end constructor

    public BinaryTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)
    {
        initializeTree(rootData, leftTree, rightTree);
    } // end constructor
}
```

A first draft of the class BinaryTree

Creating a Basic Binary Tree - 2

```
public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
                    BinaryTreeInterface<T> rightTree)
{
    initializeTree(rootData, (BinaryTree<T>)leftTree,
                    (BinaryTree<T>)rightTree);
} // end setTree

private void initializeTree(T rootData, BinaryTree<T> leftTree,
                            BinaryTree<T> rightTree)
{
    // < FIRST DRAFT – See Segments 25.4 – 25.7 for improvements. >
    root = new BinaryNode<T>(rootData);

    if (leftTree != null)
        root.setLeftChild(leftTree.root);

    if (rightTree != null)
        root.setRightChild(rightTree.root);
} // end initializeTree

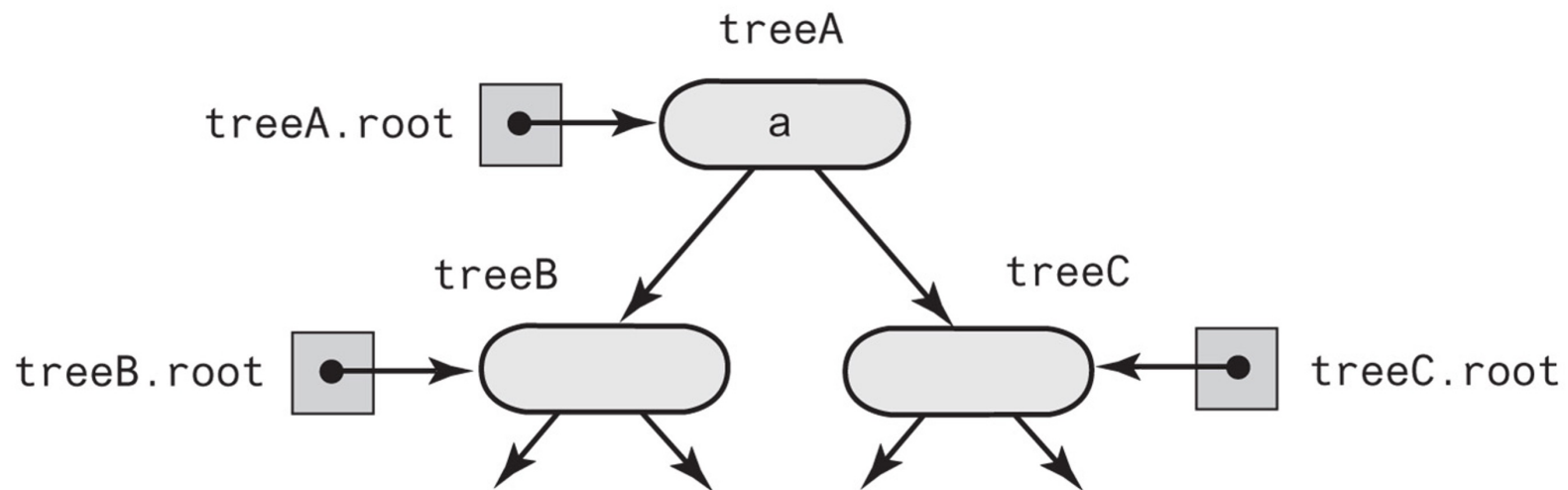
/* Implementations of setRootData, getRootData, getHeight, getNumberOfNodes,
isEmpty, clear, and the methods specified in TreeIteratorInterface are here.
. . . */

} // end BinaryTree
```

A first draft of the class BinaryTree

Creating a Binary Tree

```
treeA.setTree(a, treeB, treeC);
```



The binary tree `treeA` shares nodes with `treeB` and `treeC`

The Method `copy`

```
/** Copies the subtree rooted at this node. */
public BinaryNode<T> copy()
{
    BinaryNode<T> newRoot = new BinaryNode<>(data);
    if (leftChild != null)
        newRoot.setLeftChild(leftChild.copy());

    if (rightChild != null)
        newRoot.setRightChild(rightChild.copy());

    return newRoot;
} // end copy
```

Definition of the method `copy` in the class `BinaryNode`

The Method `initializeTree`

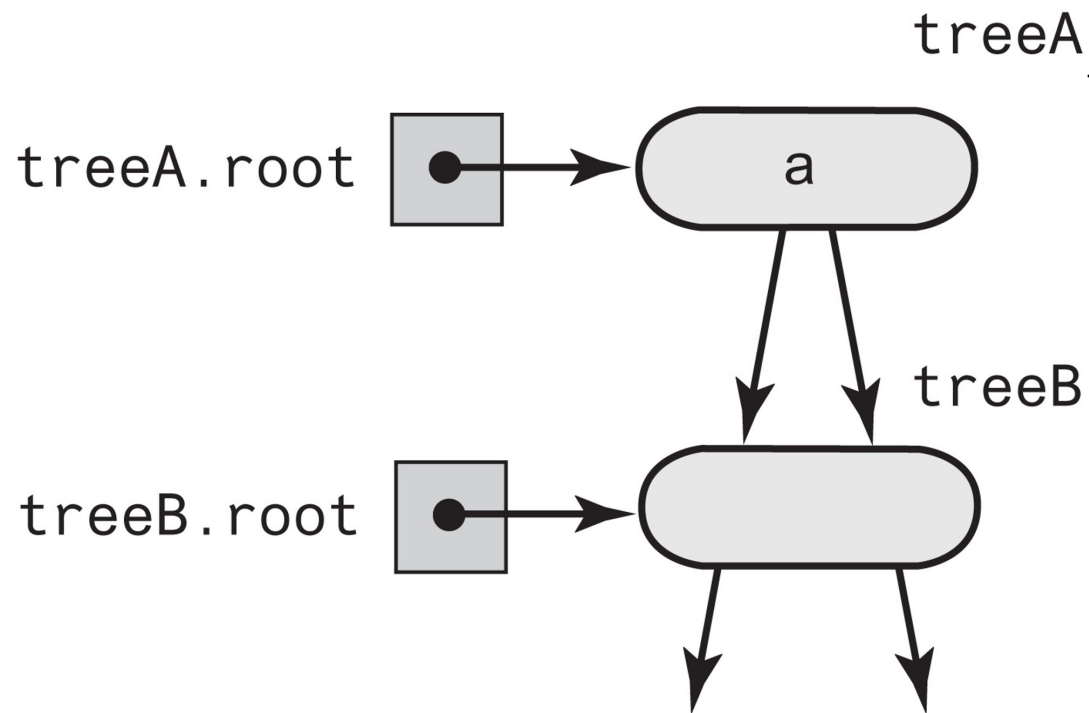
```
private void initializeTree(T rootData,
                           BinaryTree<T> leftTree,
                           BinaryTree<T> rightTree)
{
    root = new BinaryNode<>(rootData);

    if ((leftTree != null) && !leftTree.isEmpty())
        root.setLeftChild(leftTree.root.copy());

    if ((rightTree != null) && !rightTree.isEmpty())
        root.setRightChild(rightTree.root.copy());
} // end initializeTree
```

Definition of the method `copy` in the class `BinaryNode`

Additional Challenges



treeA has identical subtrees

Method `initializeTree` Solution

- If left subtree exists and not empty,
 - attach root node to `r` as left child.
- Create root node `r` containing given data.
- If right subtree exists, not empty, and distinct from left subtree,
 - attach root node to `r` as a right child.
- But if right and left subtrees are same,
 - attach copy of right subtree to `r` instead.
- If the left subtree exists and differs from the tree object used to call `initializeTree`,
 - set the subtree's data field root to null.
- If right subtree exists and differs from the tree object used to call `initializeTree`,
 - set subtree's data field root to null.

treeA has identical subtrees

Method `initializeTree` Solution (cont.)

```
private void initializeTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)
{
    root = new BinaryNode<>(rootData);

    if ((leftTree != null) && !leftTree.isEmpty())
        root.setLeftChild(leftTree.root);

    if ((rightTree != null) && !rightTree.isEmpty())
    {
        if (rightTree != leftTree)
            root.setRightChild(rightTree.root);
        else
            root.setRightChild(rightTree.root.copy());
    } // end if

    if ((leftTree != null) && (leftTree != this))
        leftTree.clear();

    if ((rightTree != null) && (rightTree != this))
        rightTree.clear();

} // end initializeTree
```

An implementation of `initializeTree`

BinaryTree Accessor and Mutator Methods

```
public void setRootData(T rootData)
{
    root.setData(rootData);
} // end setRootData

public T getRootData()
{
    if (isEmpty())
        throw new EmptyTreeException();
    else
        return root.getData();
} // end getRootData

protected void setRootNode(BinaryNode<T> rootNode)
{
    root = rootNode;
} // end setRootNode

protected BinaryNode<T> getRootNode()
{
    return root;
} // end getRootNode
```

More BinaryTree Methods

```
public int getHeight()
{
    int height = 0;
    if (root != null)
        height = root.getHeight();
    return height;
} // end getHeight

public int getNumberOfNodes()
{
    int numberOfNodes = 0;
    if (root != null)
        numberOfNodes = root.getNumberOfNodes();
    return numberOfNodes;
} // end getNumberOfNodes
```

Computing the Height and Counting Nodes

Methods within BinaryNode - 1

```
public int getHeight()
{
    return getHeight(this); // Call private getHeight
} // end getHeight

private int getHeight(BinaryNode<T> node)
{
    int height = 0;

    if (node != null)
        height = 1 + Math.max(getHeight(node.getLeftChild()),
                               getHeight(node.getRightChild()));

    return height;
}
```

Methods within `BinaryNode` - 2

```
public int getNumberOfNodes()
{
    int leftNumber = 0;
    int rightNumber = 0;

    if (left != null)
        leftNumber = left.getNumberOfNodes();

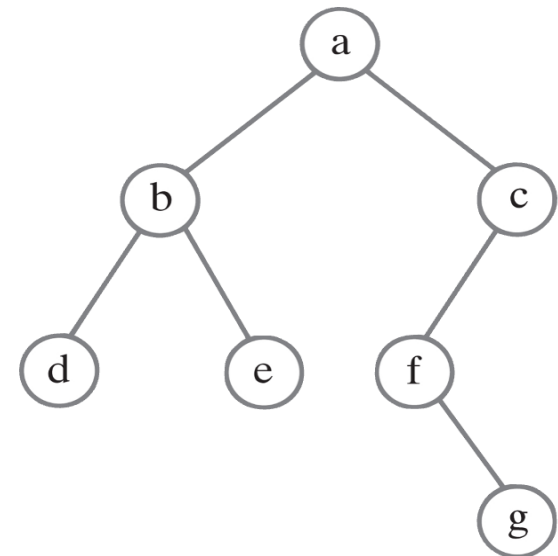
    if (right != null)
        rightNumber = right.getNumberOfNodes();

    return 1 + leftNumber + rightNumber;
} // end getNumberOfNodes
```

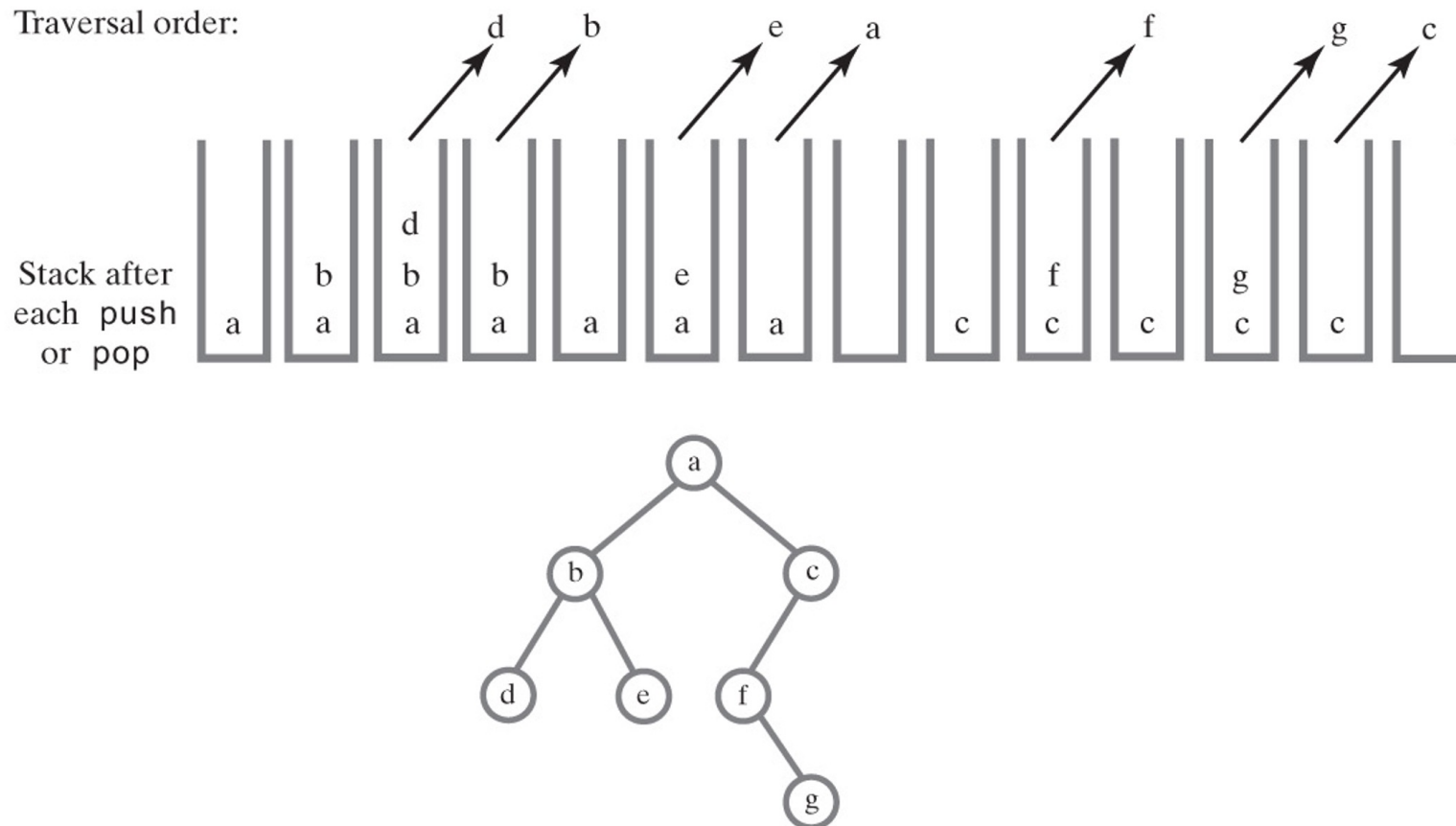
Traversing a binary tree recursively

```
public void inorderTraverse()
{
    inorderTraverse(root);
} // end inorderTraverse

private void inorderTraverse(BinaryNode<T> node)
{
    if (node != null)
    {
        inorderTraverse(node.getLeftChild());
        System.out.println(node.getData());
        inorderTraverse(node.getRightChild());
    } // end if
} // end inorderTraverse
```



A binary tree



Using a stack to perform an in-order traversal of a binary tree

Non-recursive Traversal - 2

```
public void iterativeInorderTraverse()
{
    StackInterface<BinaryNode<T>> nodeStack = new LinkedStack<>();
    BinaryNode<T> currentNode = root;

    while (!nodeStack.isEmpty() || (currentNode != null))
    {
        // Find leftmost node with no left child
        while (currentNode != null)
        {
            nodeStack.push(currentNode);
            currentNode = currentNode.getLeftChild();
        } // end while

        // Visit leftmost node, then traverse its right subtree
        if (!nodeStack.isEmpty())
        {
            BinaryNode<T> nextNode = nodeStack.pop();
            // Assertion: nextNode != null, since nodeStack was not empty
            // before the pop
            System.out.println(nextNode.getData());
            currentNode = nextNode.getRightChild();
        } // end if
    } // end while
} // end iterativeInorderTraverse
```

Traversals That Use An Iterator - 1

```
private class InorderIterator implements Iterator<T>
{
    private StackInterface<BinaryNode<T>> nodeStack;
    private BinaryNode<T> currentNode;

    public InorderIterator()
    {
        nodeStack = new LinkedStack<>();
        currentNode = root;
    } // end default constructor

    public void remove()
    {
        throw new UnsupportedOperationException();
    } // end remove

    public boolean hasNext()
    {
        return !nodeStack.isEmpty() || (currentNode != null);
    } // end hasNext
}
```

The private inner class `InorderIterator`

Traversals That Use An Iterator - 2

```
public T next()
{
    BinaryNode<T> nextNode = null;

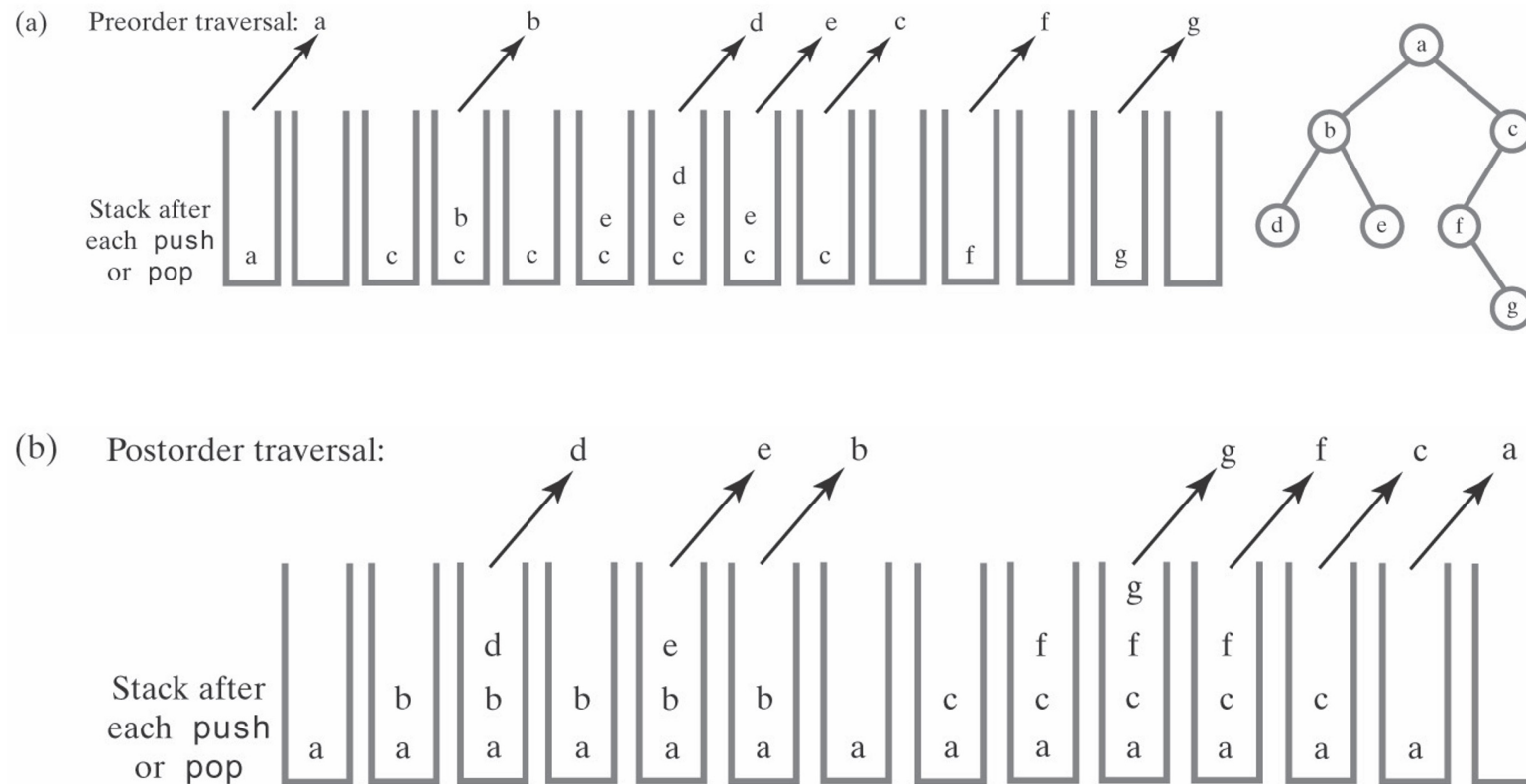
    // Find leftmost node with no left child
    while (currentNode != null)
    {
        nodeStack.push(currentNode);
        currentNode = currentNode.getLeftChild();
    } // end while

    // Get leftmost node, then move to its right subtree
    if (!nodeStack.isEmpty())
    {
        nextNode = nodeStack.pop();
        // Assertion: nextNode != null, since nodeStack was not empty
        // before the pop
        currentNode = nextNode.getRightChild();
    }
    else
        throw new NoSuchElementException();

    return nextNode.getData();
} // end next
} // end InorderIterator
```

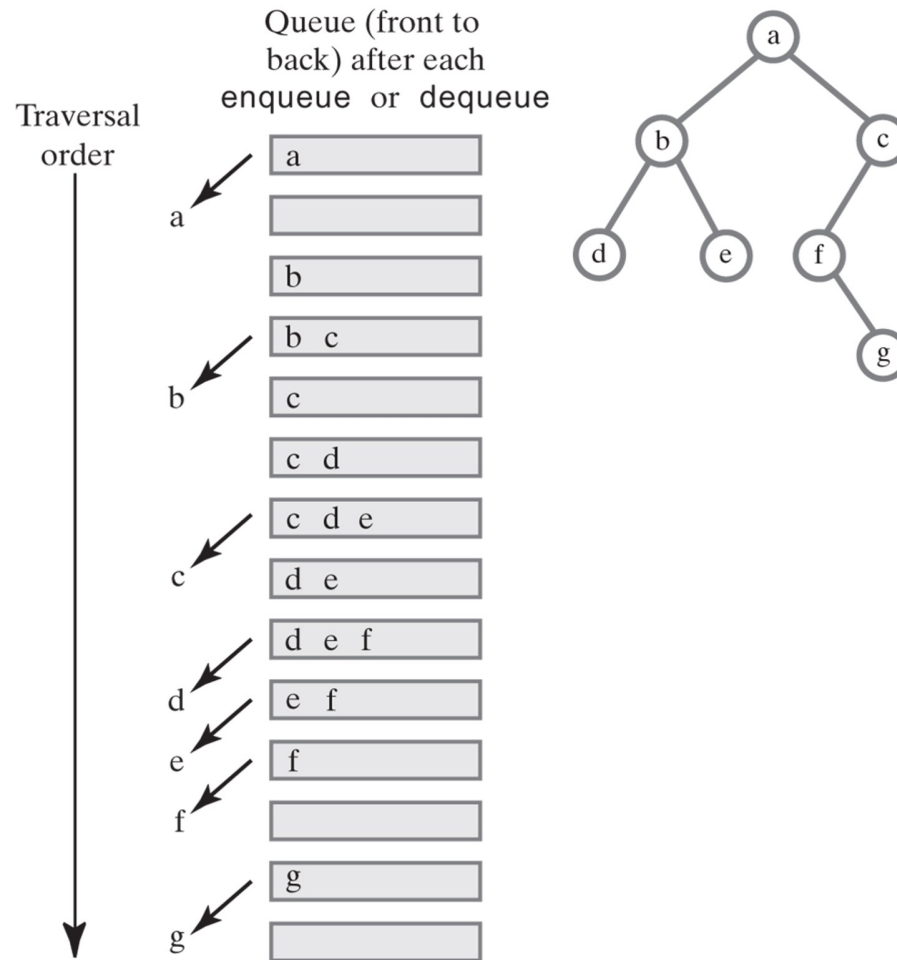
The private inner class `InorderIterator`

Using a Stack to Traverse a Binary Tree



Using a stack to traverse a binary tree in preorder and postorder

Using a Queue for Level-Order Traversal



Using a queue to traverse a binary tree in level order

Implementation of an Expression Tree - 1

```
package TreePackage;
/** An interface for an expression tree. */
public interface ExpressionTreeInterface
    extends BinaryTreeInterface<String>
{
    /** Computes the value of the expression in this tree.
        @return The value of the expression. */
    public double evaluate();
} // end ExpressionTreeInterface
```

An interface for an expression tree

Implementation of an Expression Tree - 2

```
package TreePackage;
/** A class that implements an expression tree by extending BinaryTree. */
public class ExpressionTree extends BinaryTree<String>
    implements ExpressionTreeInterface
{
    public ExpressionTree()
    {
    } // end default constructor

    public double evaluate()
    {
        return evaluate(getRootNode());
    } // end evaluate

    private double getValueOf(String variable)
    { // Strings allow multicharacter variables

        double result = 0;

        // To be defined.

        return result;
    } // end getValueOf
```

The class `ExpressionTree`

Implementation of an Expression Tree - 3

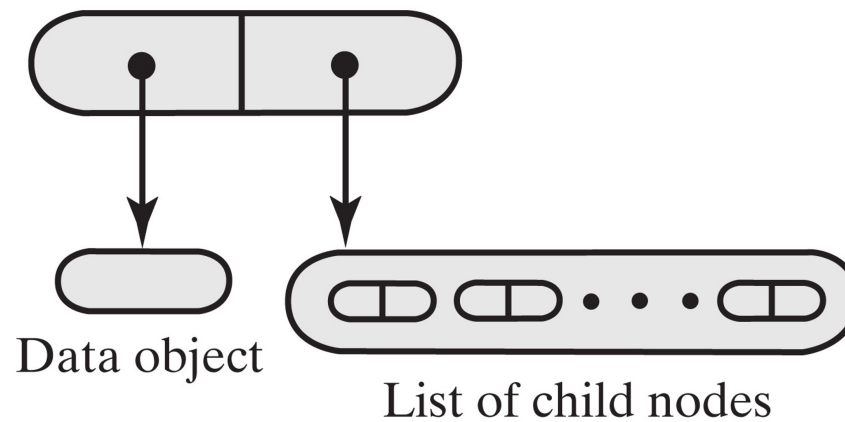
```
private double compute(String operator, double firstOperand, double
secondOperand)
{
    double result = 0;
    // To be defined.
    return result;
} // end compute

private double evaluate(BinaryNode<String> rootNode) {
    double result;
    if (rootNode == null)
        result = 0;
    else if (rootNode.isLeaf()) {
        String variable = rootNode.getData();
        result = getValueOf(variable);
    }
    else
    {
        double firstOperand = evaluate(rootNode.getLeftChild());
        double secondOperand = evaluate(rootNode.getRightChild());
        String operator = rootNode.getData();
        result = compute(operator, firstOperand, secondOperand);
    } // end if

    return result;
} // end evaluate
} // end ExpressionTree
```

The class `ExpressionTree`

Representing General Trees



A node for a general tree

Representing General Trees (cont.)

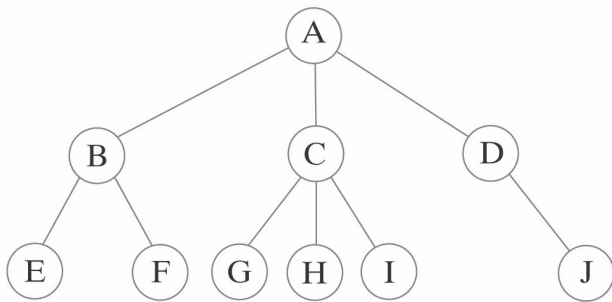
```
package TreePackage;
import java.util.Iterator;

/** An interface for a node in a general tree.*/
interface GeneralNodeInterface<T>
{
    public T getData();
    public void setData(T newData);
    public boolean isLeaf();
    public Iterator<GeneralNodeInterface<T>> getChildrenIterator();
    public void addChild(GeneralNodeInterface<T> newChild);
} // end GeneralNodeInterface
```

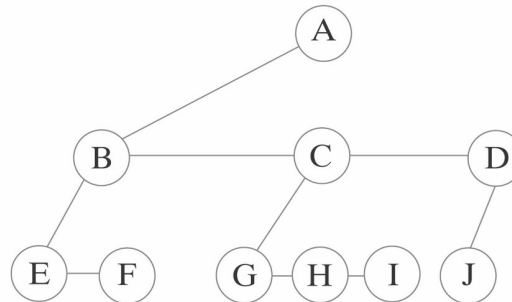
An interface for a node in a general tree

Representing General Trees (cont.)

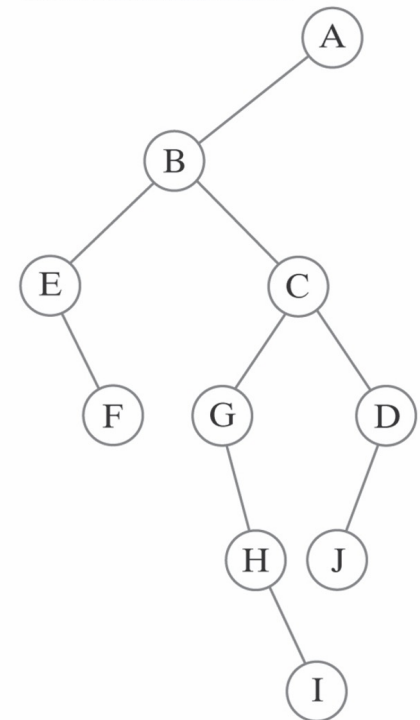
(a) A general tree



(b) An equivalent binary tree



(c) The same binary tree in a conventional form



A general tree and two views of an equivalent binary tree