Lecture 26: Trees

CS 0445: Data Structures

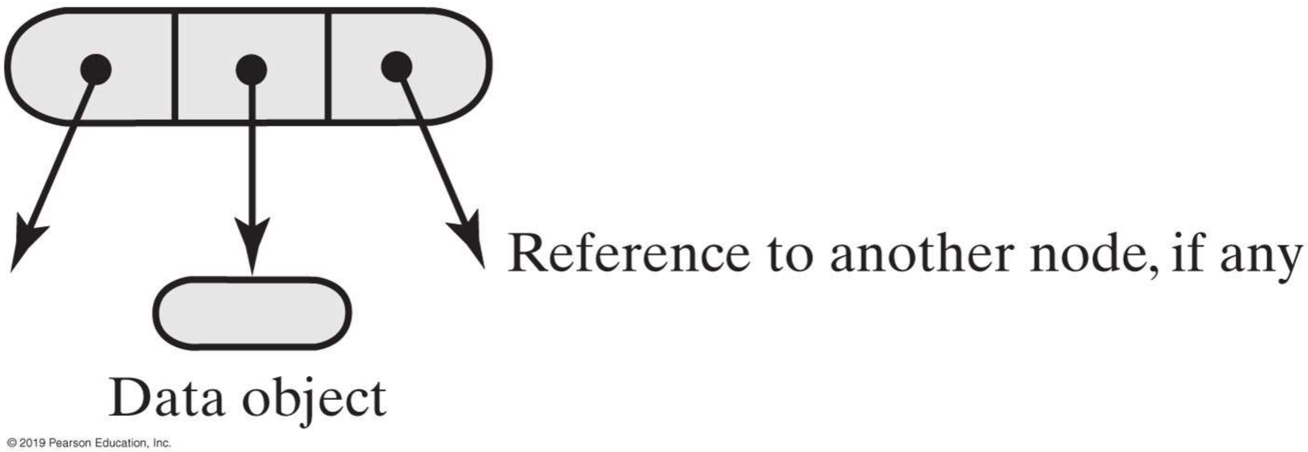
Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Nov 07, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

Nodes in a Binary Tree

- FIGURE 25-1 A node in a binary tree



Binary Tree Node (Part 1)

- LISTING 25-1 The class BinaryNode

```
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
```

Binary Tree Node (Part 2)

- LISTING 25-1 The class BinaryNode

```
/** 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
```

Binary Tree Node (Part 3)

- LISTING 25-1 The class BinaryNode

```
/** 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
```

Binary Tree Node (Part 4)

- LISTING 25-1 The class BinaryNode

```
   /** 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
```

Interface for a Basic Binary Tree

- Interface for a class of binary trees

```
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
```

Creating a Basic Binary Tree (Part 1)

- LISTING 25-2 A first draft of the class `BinaryTree`

```
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
```

Creating a Basic Binary Tree (Part 2)

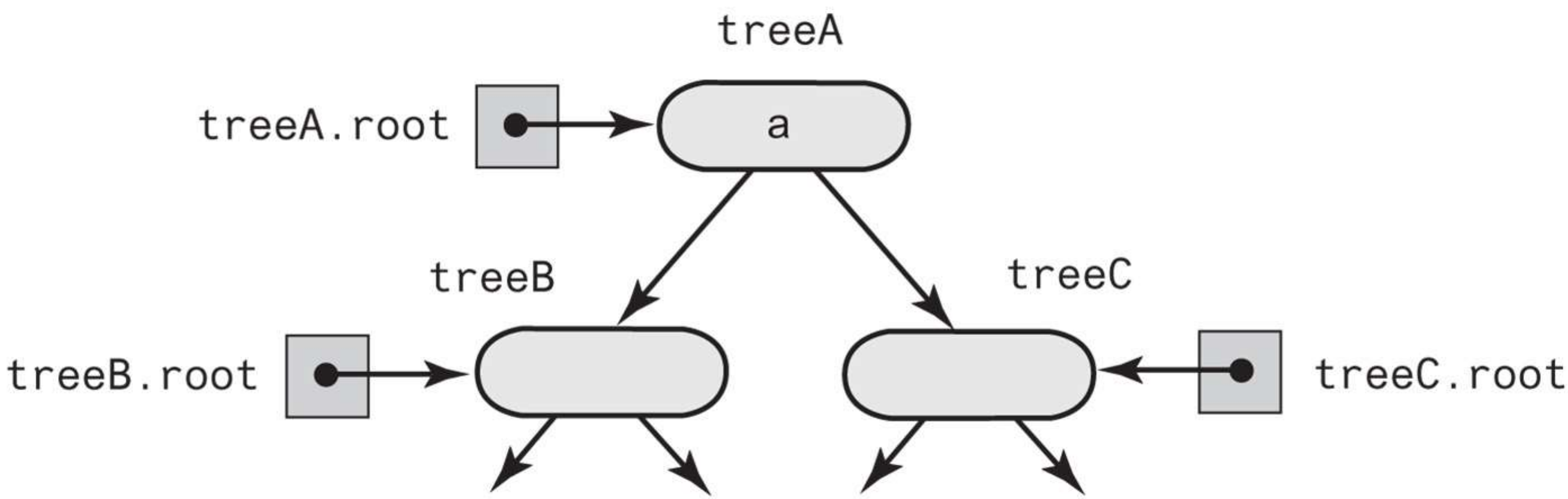
- LISTING 25-2 A first draft of the class `BinaryTree`

```
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
```

Creating a Binary Tree

- FIGURE 25-2 The binary tree `treeA` shares nodes with `treeB` and `treeC`

treeA.setTree(a, treeB, treeC);



The Method `copy`

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

```
/** 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
```

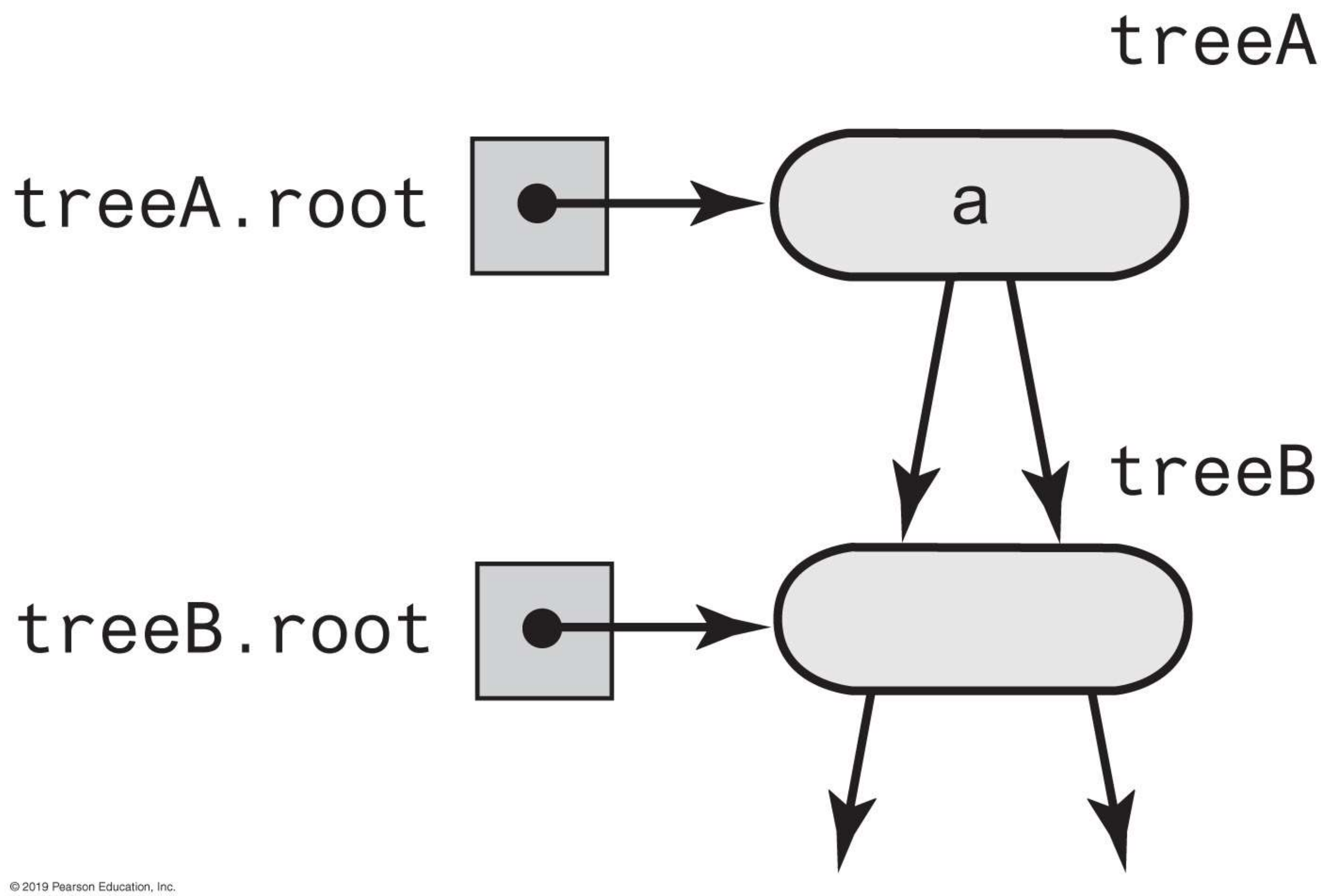
The Method `initializeTree`

- Method `initializeTree` can invoke `copy` to copy the nodes from the two given subtrees

```
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
```

Additional Challenges

- FIGURE 25-3 `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.

Method `initializeTree` Solution

- An implementation of `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);

  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
```

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

- Computing the Height and Counting Nodes

```
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
```

Methods within `BinaryNode`.

```
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

```
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
```

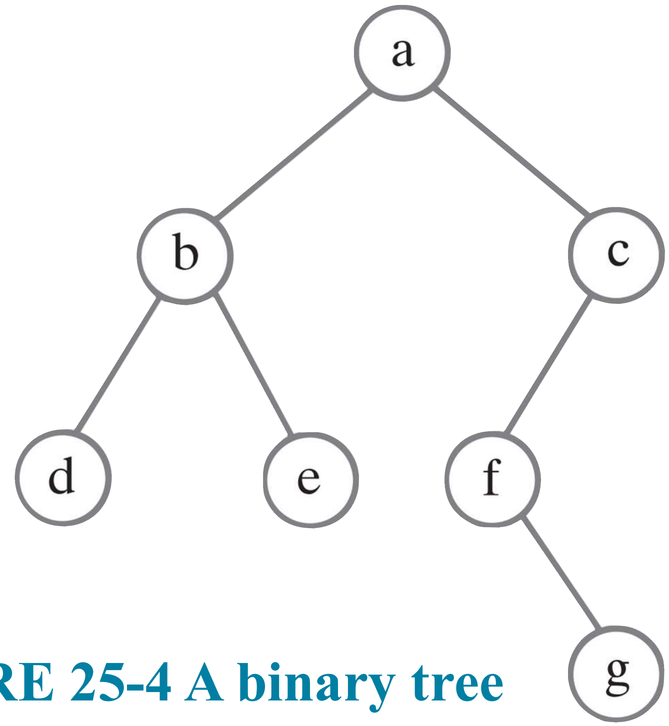
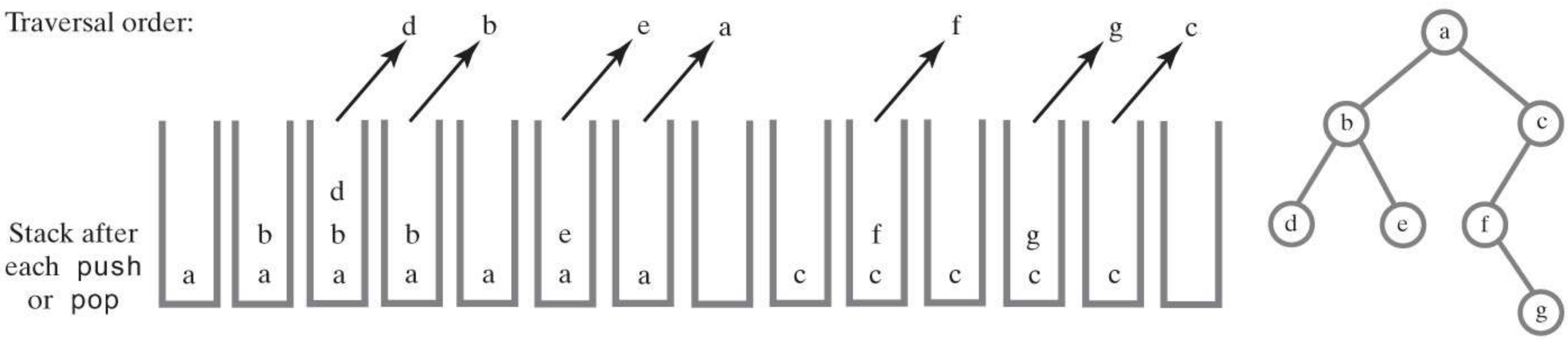


FIGURE 25-4 A binary tree

Non-recursive Traversal

- FIGURE 25-5 Using a stack to perform an in-order traversal of a binary tree



Non-recursive Traversal

- Iterative version …

```
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 (Part 1)

- LISTING 25-3 The private inner class `InorderIterator`

```
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
```

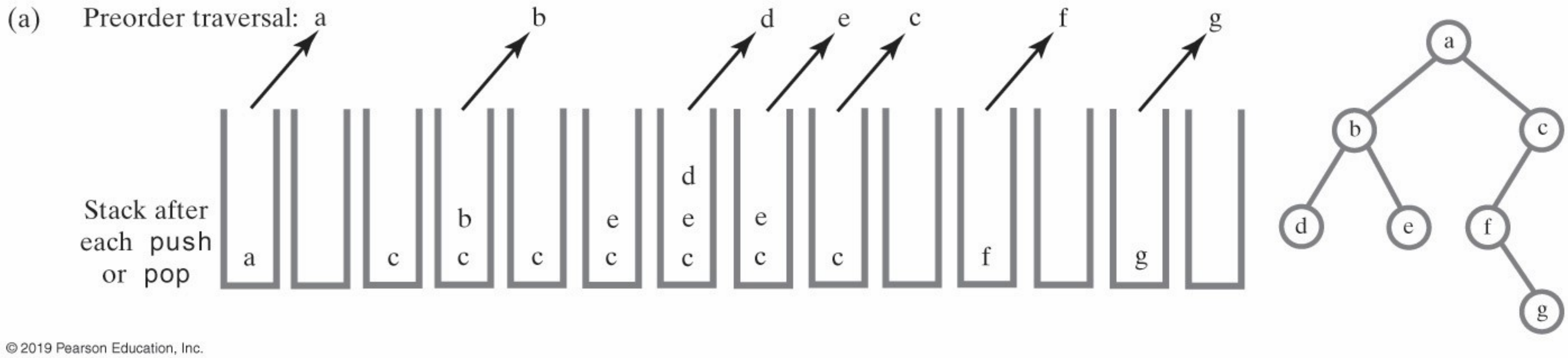
Traversals That Use An Iterator (Part 2)

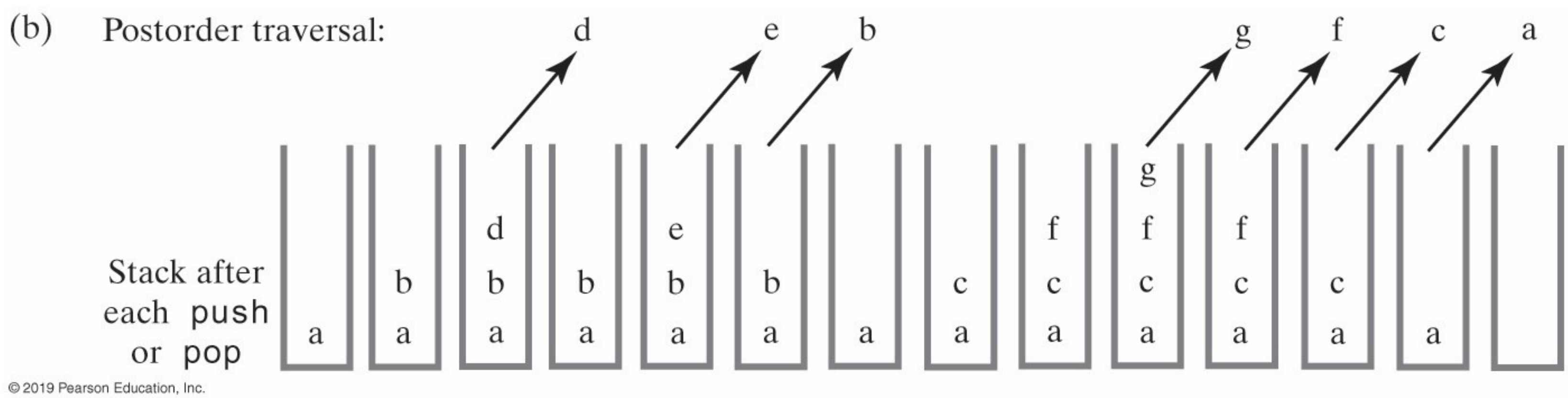
- LISTING 25-3 The private inner class `InorderIterator`

```
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
```

Using a Stack to Traverse a Binary Tree

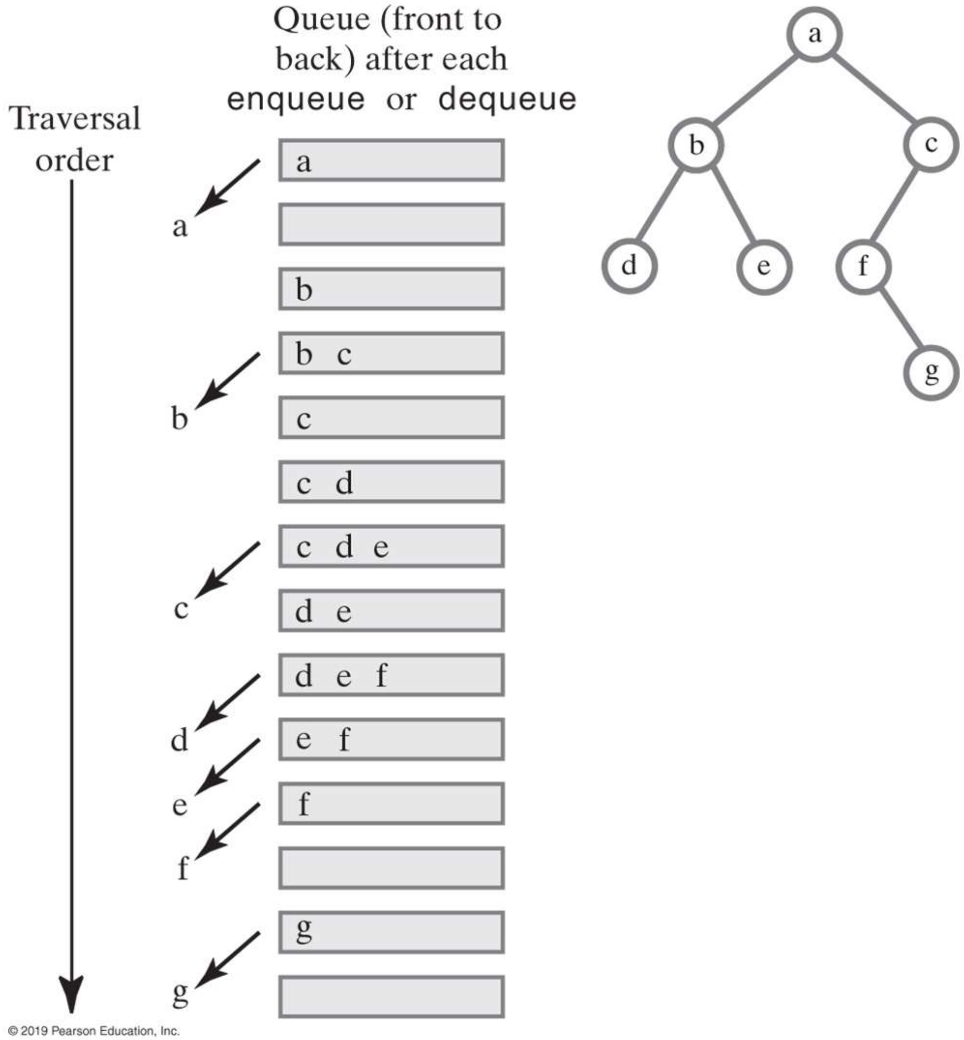
- FIGURE 25-6 Using a stack to traverse a binary tree in preorder and postorder





Using a Queue for Level-Order Traversal

- FIGURE 25-7 Using a queue to traverse a binary tree in level order



Implementation of an Expression Tree

- LISTING 25-4 An interface for an expression tree

```
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
```

Implementation of an Expression Tree (Part 1)

- LISTING 25-5 The class `ExpressionTree`

```
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
```

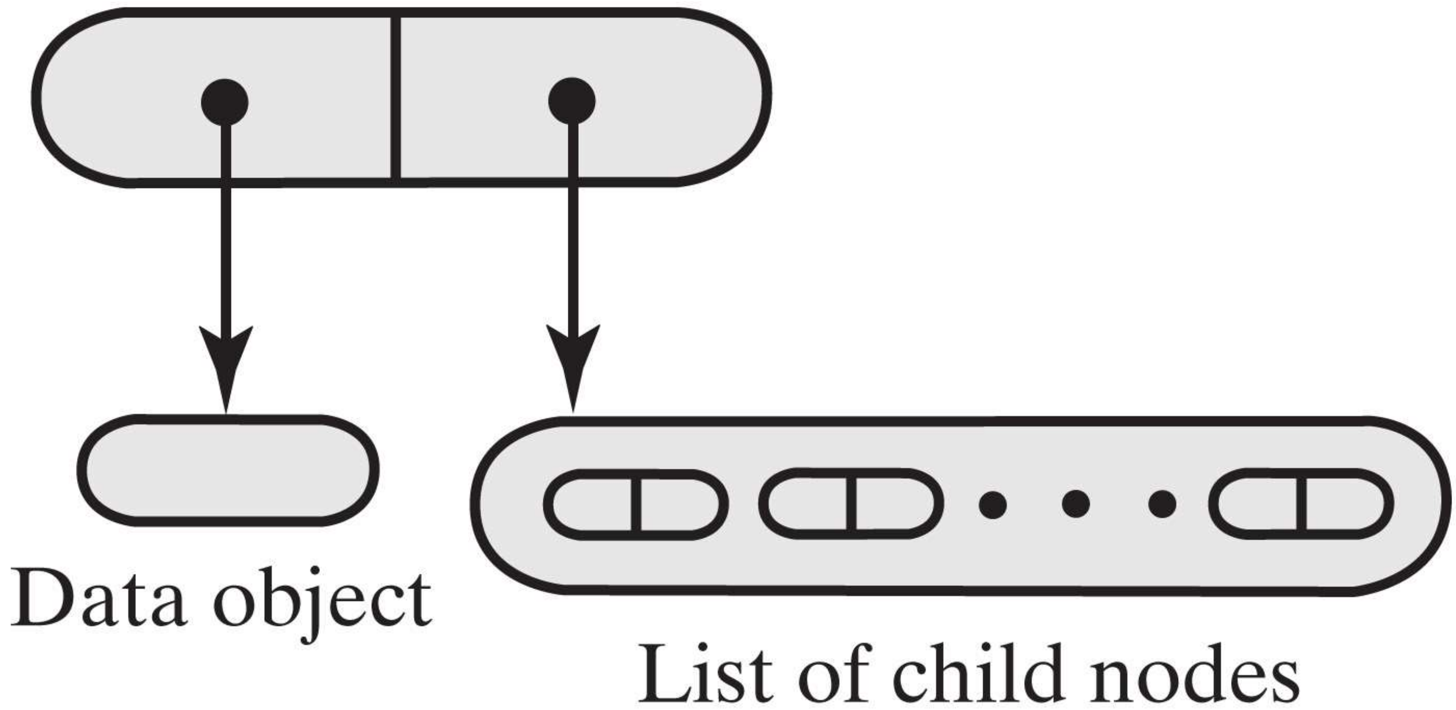
Implementation of an Expression Tree (Part 1)

- LISTING 25-5 The class `ExpressionTree`

```
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
```

Representing General Trees

- FIGURE 25-8 A node for a general tree



Representing General Trees

- LISTING 25-6 An interface for a node in a general tree

```
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
```

Representing General Trees

- FIGURE 25-9 A general tree and two views of an equivalent binary tree