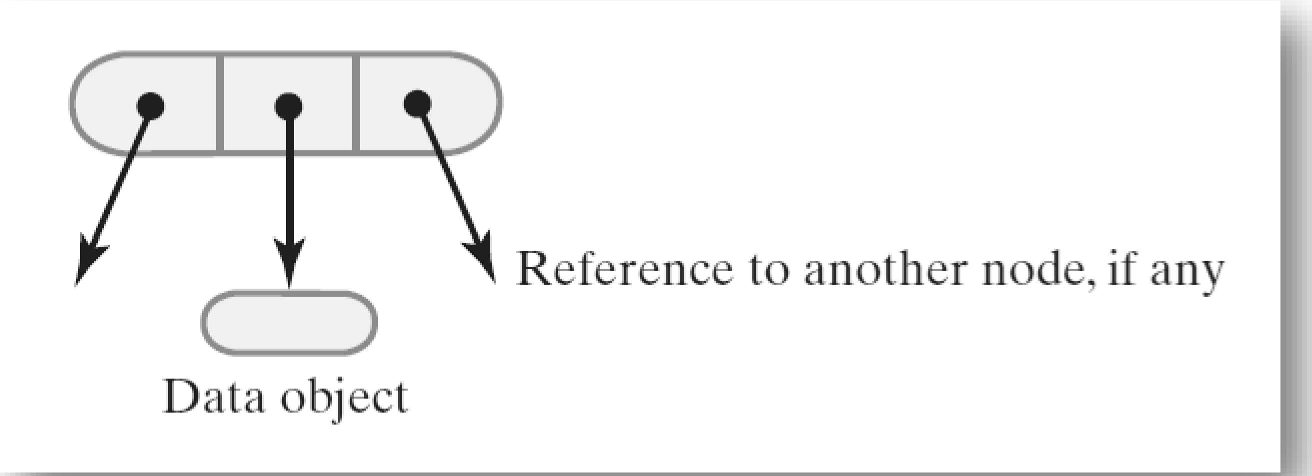
A Node in a Binary Tree



First Impl.

```
public class BinTree<E>
{
 private Node root;

 private class Node
 {
  public E data;
  public Node left;
  public Node right;

  public Node(E item)
  {
   if (item == null)
    throw new NullPointerException("Item 0");
   data = item;
   left = null;
   right = null;
  }
 }

 public BinTree()
 {
  root = null;
 }
```

```java
public BinTree(E item)
{
 if (item == null)
  throw new NullPointerException("Item 1");
 root = new Node(item);
}
```

```java
public BinTree(BinTree<E> lefttree, BinTree<E> righttree, E item)
{
 if (item == null) throw new NullPointerException("Item 2");
 if (lefttree == null) throw new NullPointerException("Ltree");
 if (righttree == null) throw new NullPointerException("Rtree");
 root = new Node(item);

 root.left = copyTree(lefttree.root);
 root.right = copyTree(righttree.root);
}
```

```java
private Node copyTree(Node r)
{
 if (r == null) return null;
 Node retval = new Node(r.data);
 retval.left = copyTree(r.left);
 retval.right = copyTree(r.right);
 return retval;
}
```

```java
public void preOrderTraversal()
{
 System.out.println("\nPre-order traversal");
 recPreOrderTraversal(root);
}
```

```java
private void recPreOrderTraversal(Node r)
{
 if (r == null) return;
 System.out.println(r.data.toString()); // "Visit the node"
 recPreOrderTraversal(r.left);
 recPreOrderTraversal(r.right);
}
```

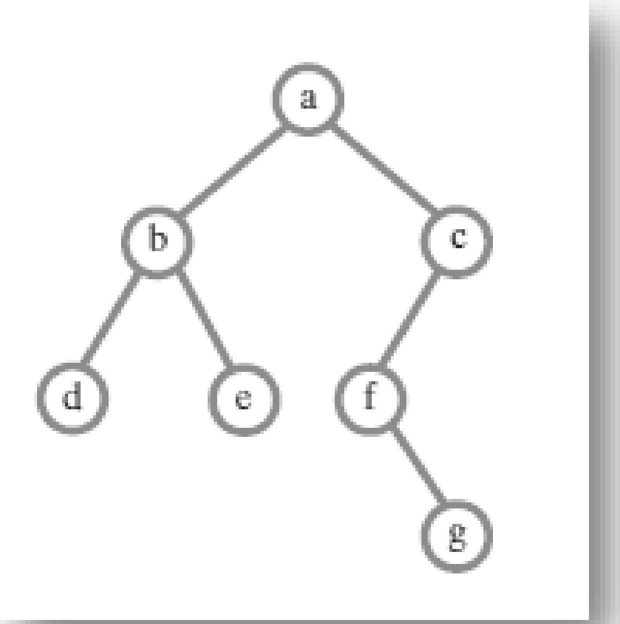
```java
public void inOrderTraversal()
{
 System.out.println("\nIn-order traversal");
 recInOrderTraversal(root);
}
```

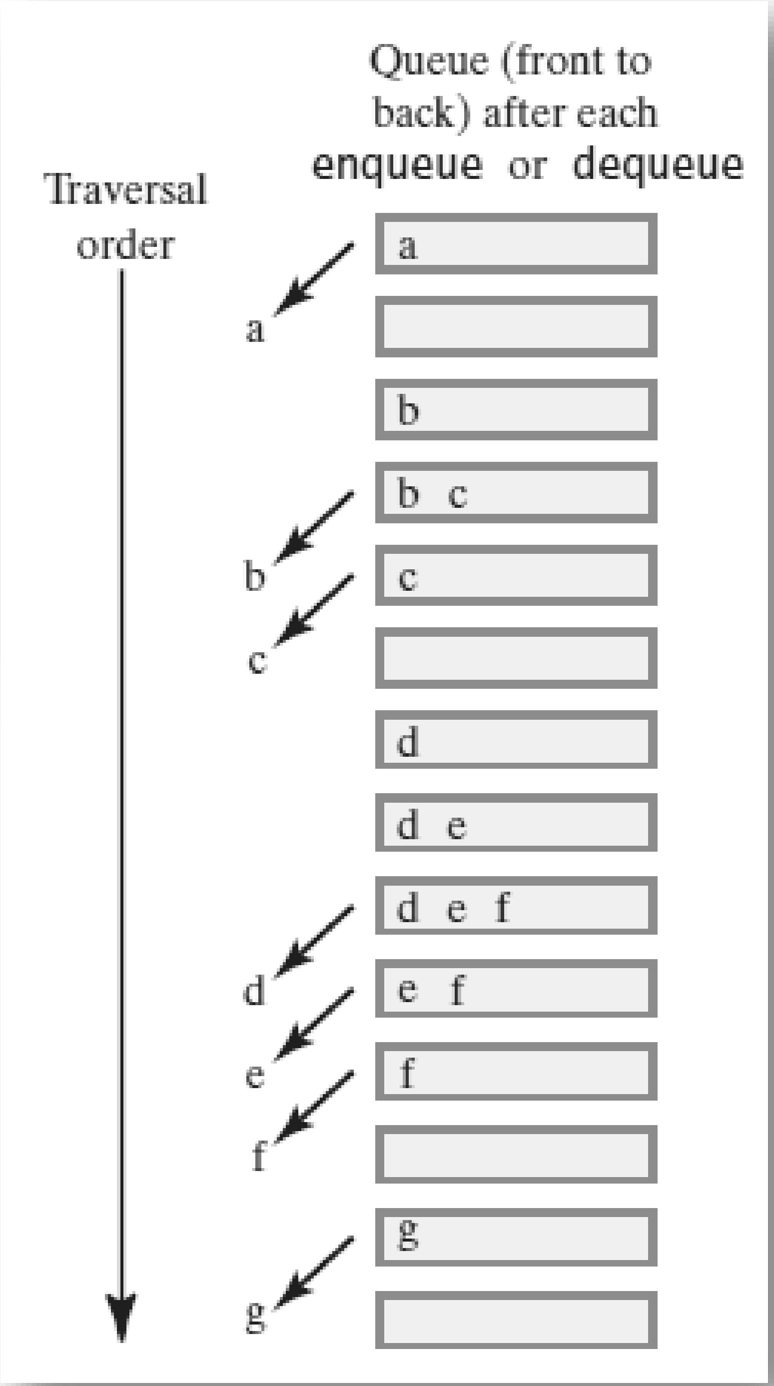
```java
private void recInOrderTraversal(Node r)
{
 if (r == null) return;
 recInOrderTraversal(r.left);
 System.out.println(r.data.toString()); // "Visit the node"
 recInOrderTraversal(r.right);
}
```

```java
public void postOrderTraversal()
{
 System.out.println("\nPost-order traversal");
 recPostOrderTraversal(root);
}
```

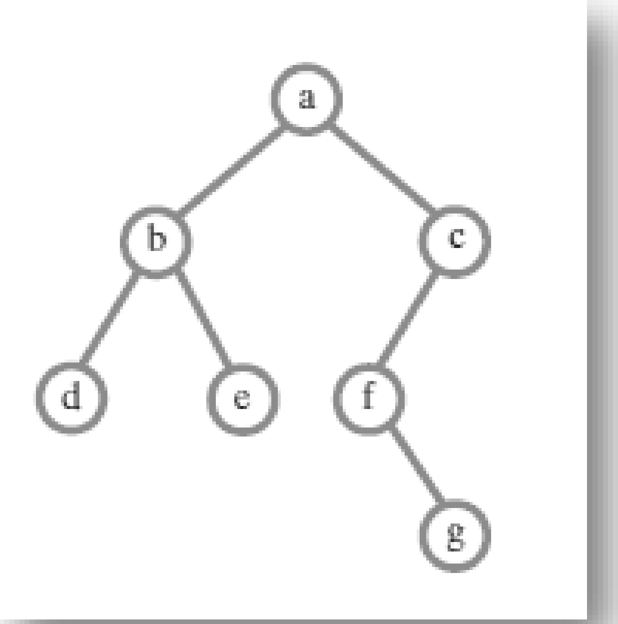
```java
private void recPostOrderTraversal(Node r)
{
 if (r == null)
  return;
 recPostOrderTraversal(r.left);
 recPostOrderTraversal(r.right);
 System.out.println(r.data.toString()); // "Visit the node"
}
```

Iterative level-order traversal

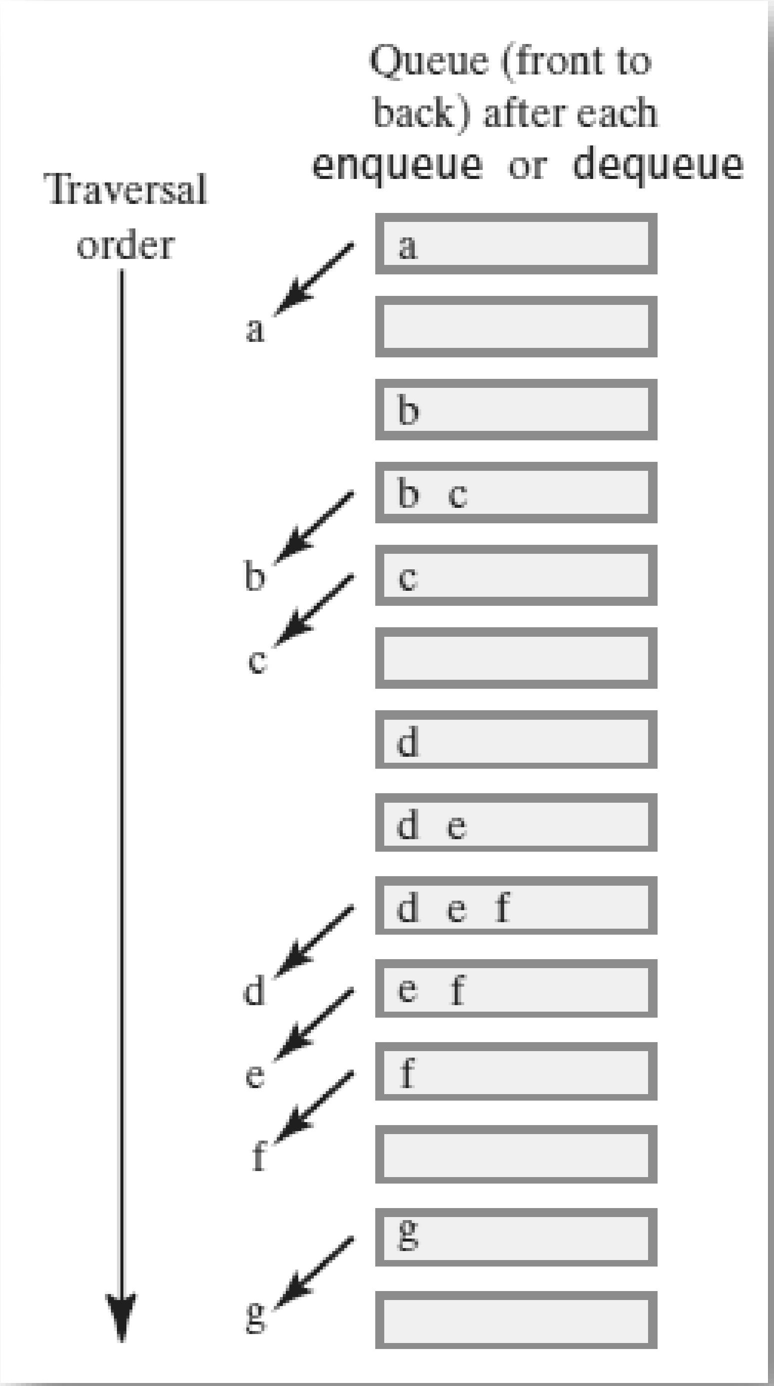




Iterative level-order traversal




```
public void breadthFirstTraversal()
{
 System.out.println("\nBreadth-first traversal");
 java.util.LinkedList<Node> q = new java.util.LinkedList<Node>();
 q.addLast(root);
 while (!q.isEmpty())
 {
  Node removed = q.removeFirst();
  System.out.println(removed.data.toString());
  if (removed.left != null) q.addLast(removed.left);
  if (removed.right != null) q.addLast(removed.right);
 }
}
```



```java
public static void main(String[] args)
{
  BinTree<String> a1 = new BinTree<String>("Phineas");
  BinTree<String> a2 = new BinTree<String>("Ferb");
  BinTree<String> a3 = new BinTree<String>(a1, a2, "Doof");
  BinTree<String> a4 = new BinTree<String>("Perry");
  BinTree<String> a5 = new BinTree<String>(a3, a4, "Candace");

  a5.preOrderTraversal();
  a5.inOrderTraversal();
  a5.postOrderTraversal();
  a5.breadthFirstTraversal();
}
```

Another Impl.

Creating a Basic Binary Node

```java
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
```

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

/**
 * 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
```

```java
/**
 * Retrieves the left child of this node.
 * @return The 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; }

/**
 * 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

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

```java
/**
 * Copies the subtree rooted at this node.
 *
 * @return The root of a copy of 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
```

```java
/**
 * 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()
{
 int leftNumber = 0;
 int rightNumber = 0;

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

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

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

```java
/**
 * Computes the height of the subtree rooted at this node.
 * @return The height of the subtree rooted at this node.
 */
public int getHeight()
{
 return getHeight(this);
} // end getHeight

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

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

 return height;
} // end getHeight


} // end BinaryNode
```

Creating a Basic Binary Tree

```
package TreePackage;

import java.util.Iterator;
import java.util.EmptyStackException;
import java.util.NoSuchElementException;
import java.util.Stack;

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)
 {
  privateSetTree(rootData, leftTree, rightTree);
 } // end constructor

 public void setTree(T rootData)
 {
  root = new BinaryNode<>(rootData);
 } // end setTree
```

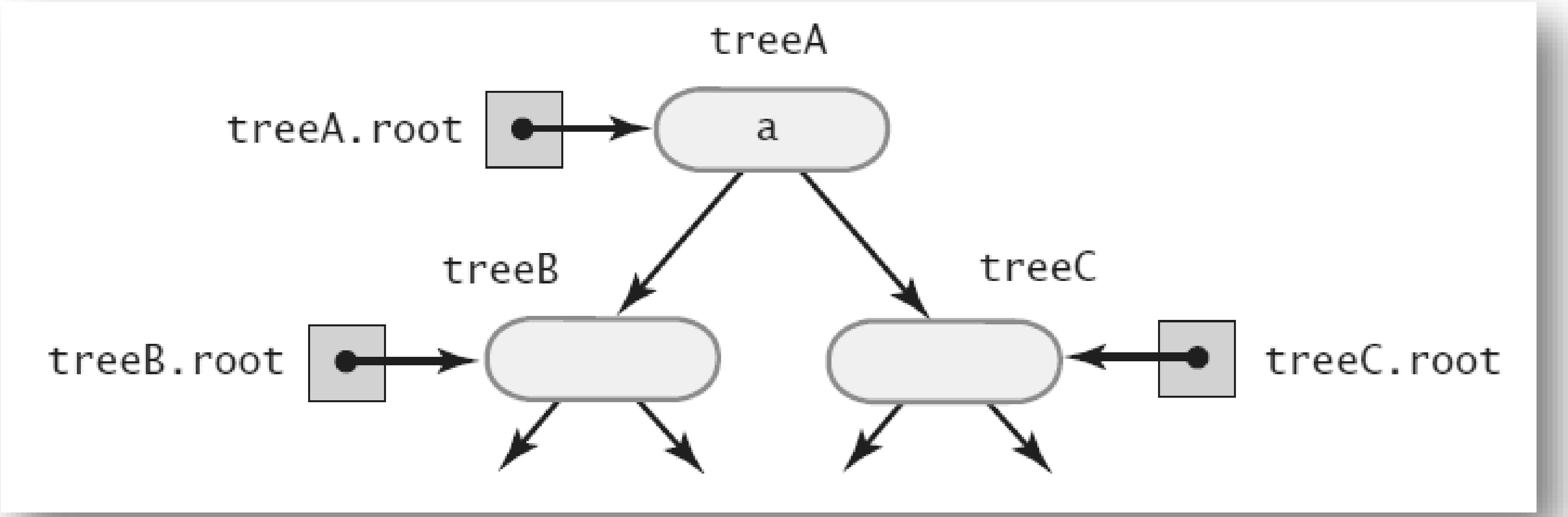
```java
public void setTree(T rootData, BinaryTreeInterface<T> leftTree, BinaryTreeInterface<T> rightTree)
{
 privateSetTree(rootData, (BinaryTree<T>) leftTree, (BinaryTree<T>) rightTree);
} // end setTree

private void privateSetTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)
{
 // < FIRST DRAFT >
 root = new BinaryNode<T>(rootData);

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

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

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

public boolean isEmpty() { return root == null; } // end isEmpty

public void clear() { root = null; } // end clear

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

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

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

public int getHeight() { return (root!=null)?root.getHeight():0; } // end getHeight

public int getNumberOfNodes() { return (root!=null)?root.getNumberOfNodes():0; }

} // end BinaryTree
```

The method **privateSetTree**

- The implementation of a **privateSetTree** given in previous slide is really not sufficient to handle all possible uses of the method.

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



The method **privateSetTree** (cont.)

```
private void privateSetTree(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 privateSetTree
```

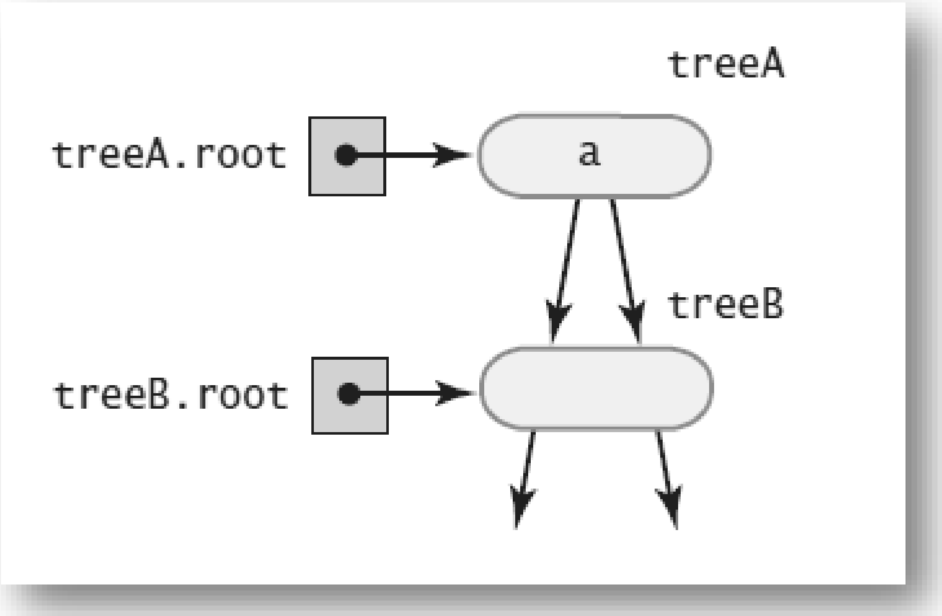
```
public BinaryNode<T> copy()
{
 BinaryNode<T> newRoot = new BinaryNode<>(data);

 if (leftChild != null)
  newRoot.setLeftChild(leftChild.copy());

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

 return newRoot;
} // end copy
```

The method **privateSetTree**

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

The method **privateSetTree**

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



The method **privateSetTree** (cont.)

1. Create a root node r containing the given data.
2. If the left subtree exists and is not empty, attach its root node to r as a left child.
3. If the right subtree exists, is not empty, and is distinct from the left subtree, attach its root node to r as a right child. But if the right and left subtrees are the same, attach a copy of the right subtree to r instead.
4. If the left subtree exists and differs from the tree object used to call privateSetTree, set the subtree's data field root to null.
5. If the right subtree exists and differs from the tree object used to call privateSetTree, set the subtree's data field root to null.

The method **privateSetTree** (cont.)

```
private void privateSetTree(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 privateSetTree
```

Traversing a binary tree recursively

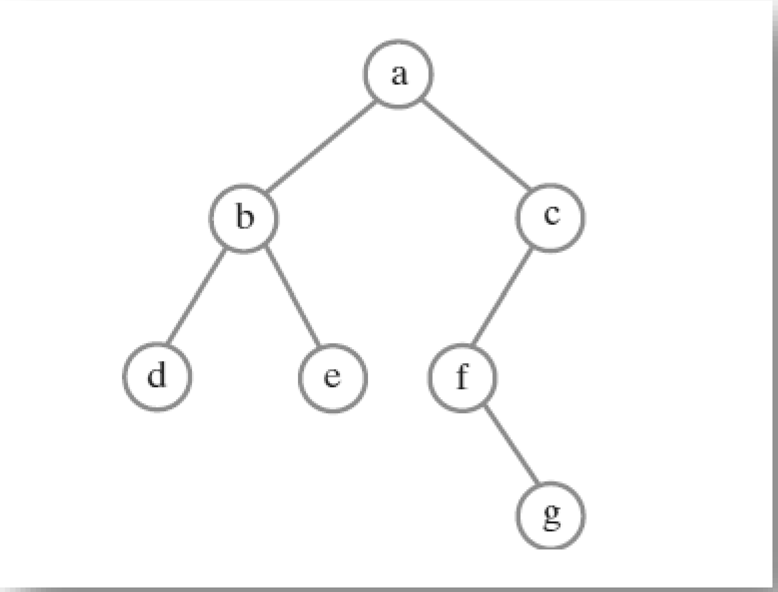
```
public void inorderTraverse()
{
 inorderTraverse(root);
} // end inorderTraverse

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

Traversing a binary tree recursively

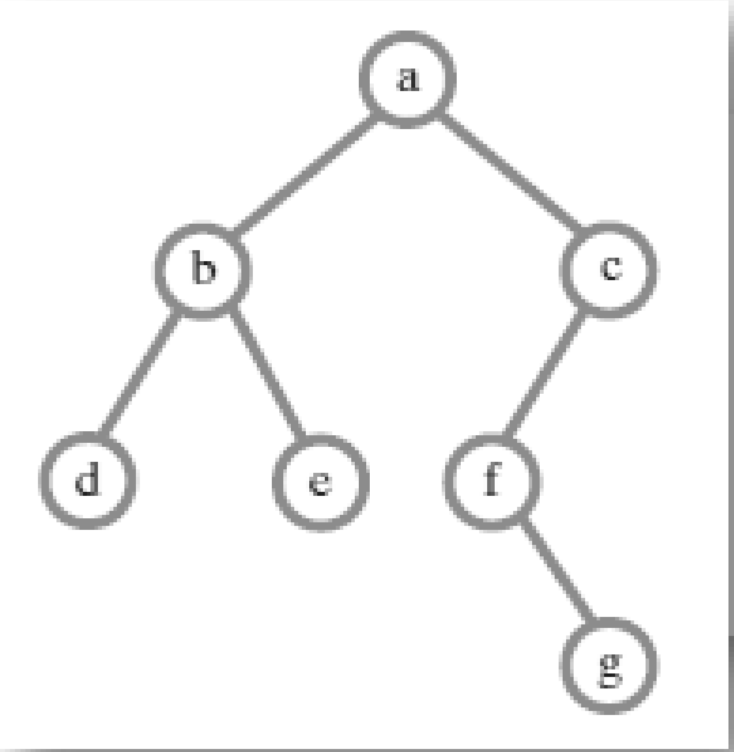
```
public void inorderTraverse()
{
 inorderTraverse(root);
} // end inorderTraverse

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

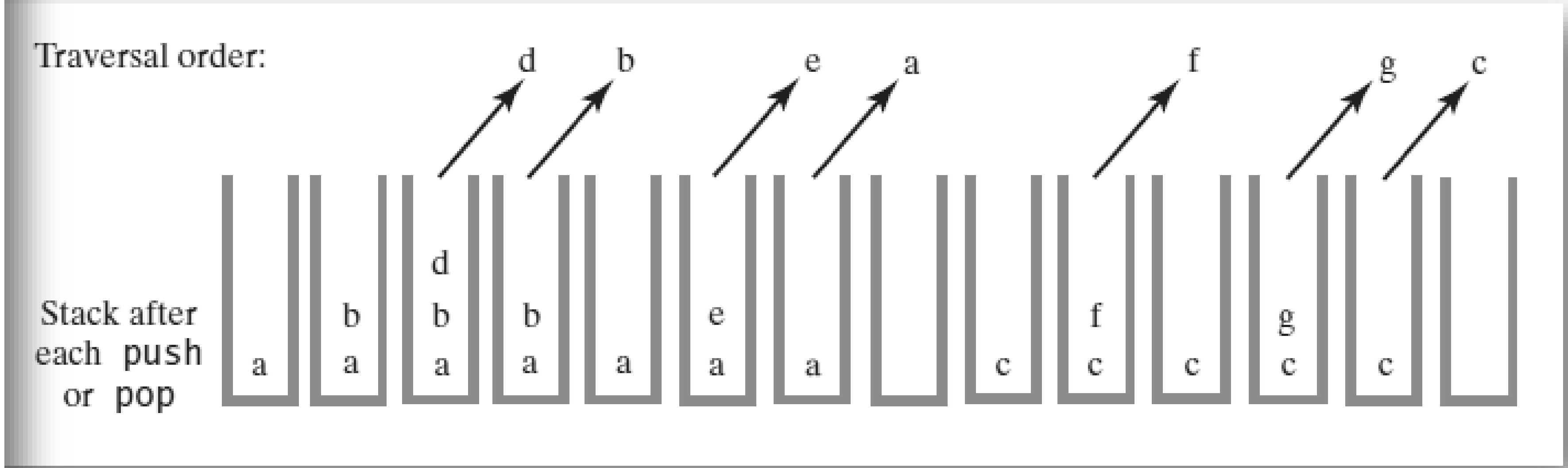
QUESTION

- Trace the method **inorderTraverse** with the binary tree below.
- What data is displayed?



An Iterative version of an inorder traversal

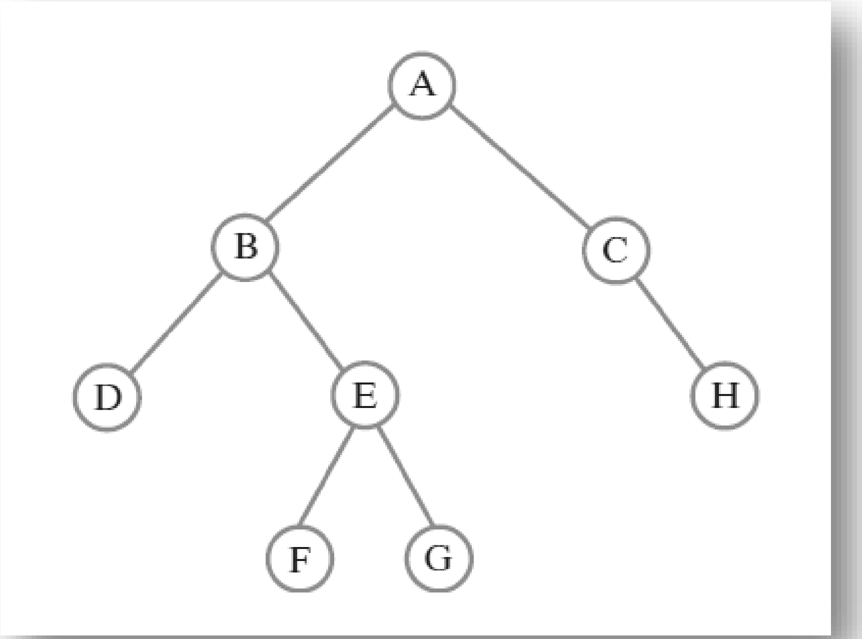




```java
public void iterativeInorderTraverse()
{
 StackInterface<BinaryNode<T>> nodeStack = new LinkedStack<>();
 BinaryNode<T> currentNode = root;

 while (!nodeStack.isEmpty() || (currentNode != null))
 {
  while (currentNode != null)
  {
   nodeStack.push(currentNode);
   currentNode = currentNode.getLeftChild();
  } // end while

  if (!nodeStack.isEmpty())
  {
   BinaryNode<T> nextNode = nodeStack.pop();
   assert nextNode != null;
   System.out.println(nextNode.getData());
   currentNode = nextNode.getRightChild();
  } // end if
 } // end while
} // end iterativeInorderTraverse
```

QUESTION Trace this method with the binary tree shown below.

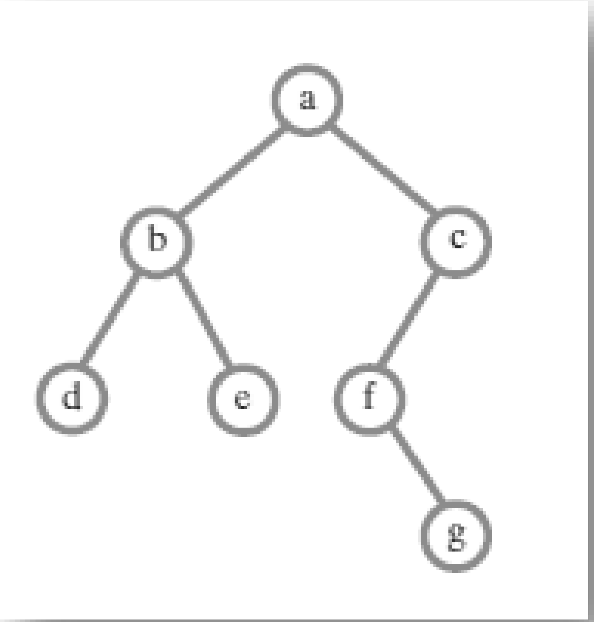
```java
public void iterativeInorderTraverse()
{
 StackInterface<BinaryNode<T>> nodeStack = new LinkedStack<>();
 BinaryNode<T> currentNode = root;

 while (!nodeStack.isEmpty() || (currentNode != null))
 {
  while (currentNode != null)
  {
   nodeStack.push(currentNode);
   currentNode = currentNode.getLeftChild();
  } // end while

  if (!nodeStack.isEmpty())
  {
   BinaryNode<T> nextNode = nodeStack.pop();
   assert nextNode != null;
   System.out.println(nextNode.getData());
   currentNode = nextNode.getRightChild();
  } // end if
 } // end while
} // end iterativeInorderTraverse
```

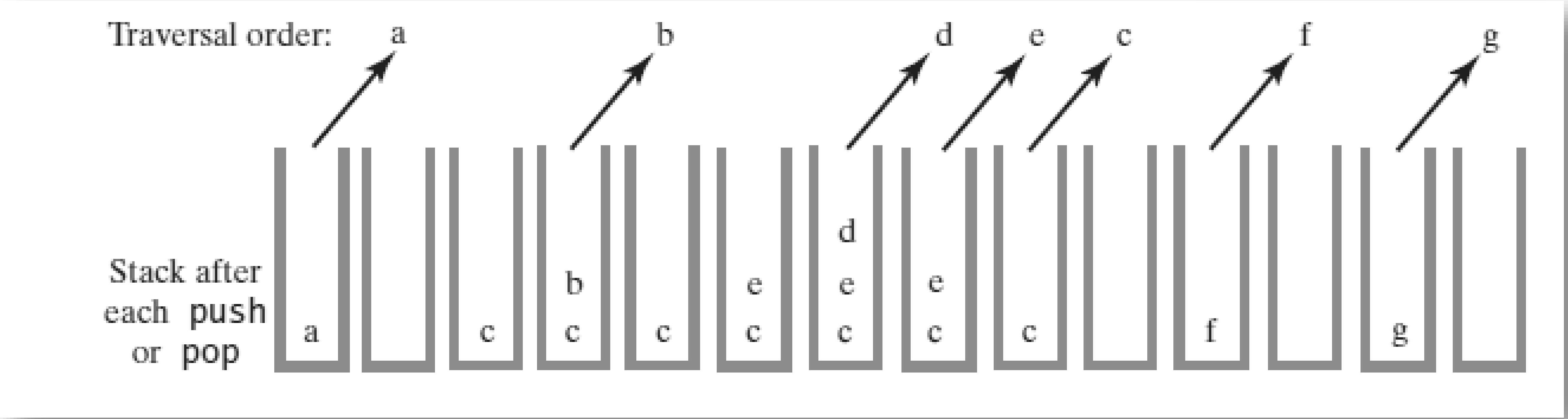


```java
public Iterator<T> getInorderIterator() { return new InorderIterator(); }

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

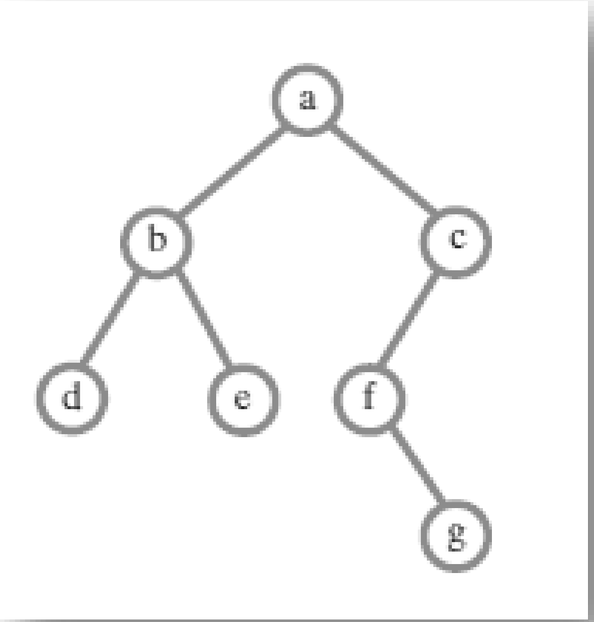
```java
public T next()
{
 BinaryNode<T> nextNode = null;

 while (currentNode != null)
 {
  nodeStack.push(currentNode);
  currentNode = currentNode.getLeftChild();
 } // end while

 if (!nodeStack.isEmpty())
 {
  nextNode = nodeStack.pop();
  assert nextNode != null;
  currentNode = nextNode.getRightChild();
 } else
  throw new NoSuchElementException();

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

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

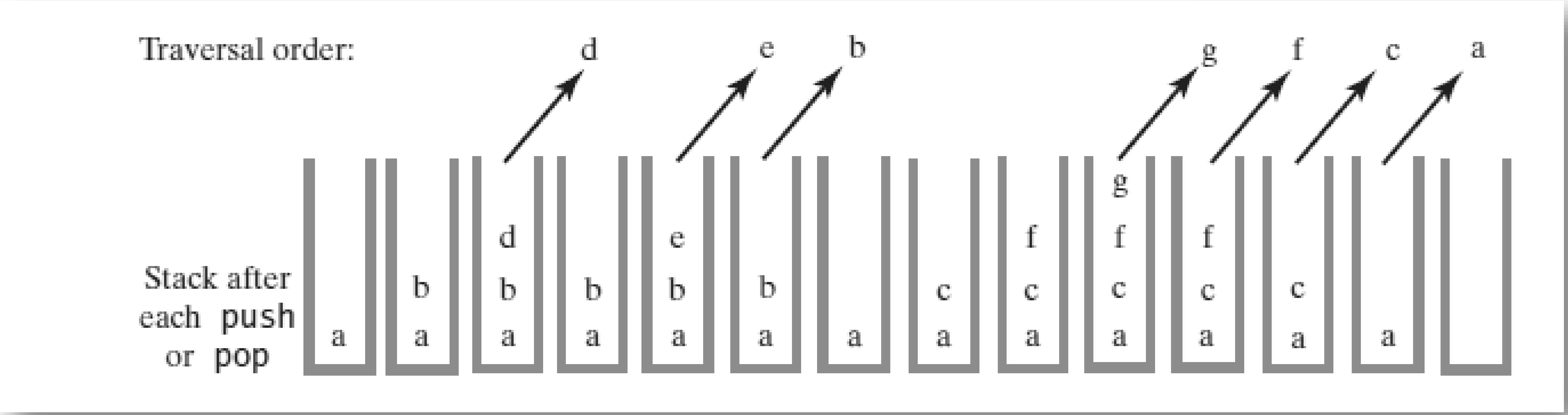
Iterative preorder traversal





Iterative postorder traversal





Reference

- F. C. Carrano & T. M. Henry, “Data Structures and Abstractions with Java”, 4th ed., 2015. Pearson Education, Inc.
- BinTree.java