CS2400 - Data Structures and Advanced Programming
Module 8: Trees (I)

Hao Ji
Computer Science Department
Cal Poly Pomona

Organization: Linear vs Nonlinear

- Linear organization
  - Such as Array, Linked List, Stack, Queue, and Dictionary
  - Objects appear one after the other.

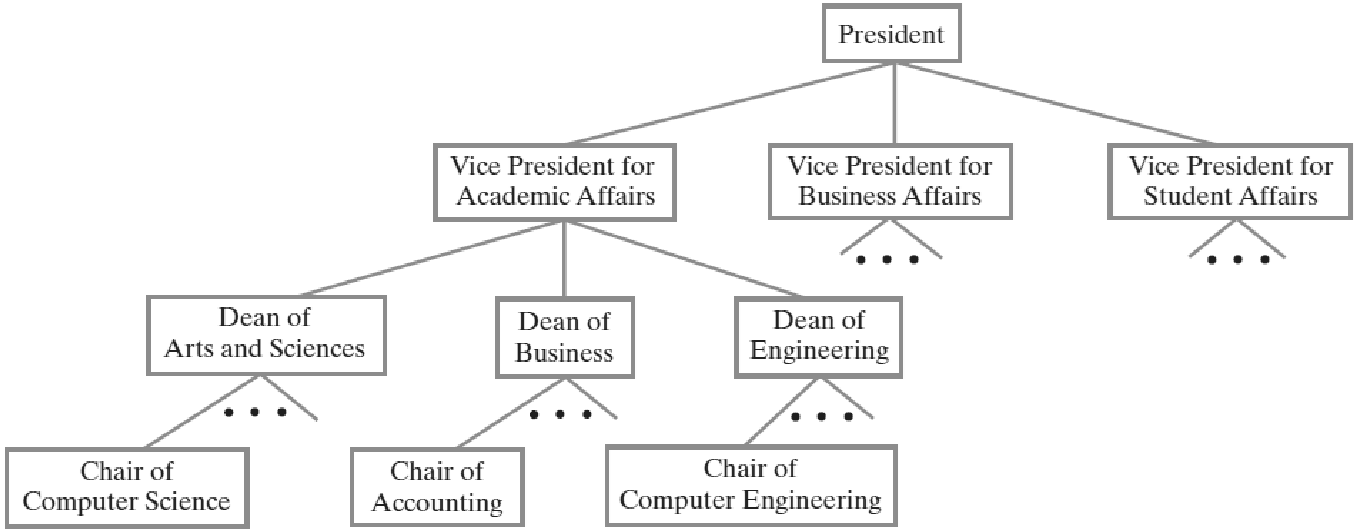
- Nonlinear organization
  - Such as Tree and Graph
  - Objects are arranged hierarchically

Today

- This Class
  - Tree Terminology
  - Tree Traversals
  - Tree Representations
  - Java Implementation of Binary Trees and Binary Nodes

Hierarchical Organizations

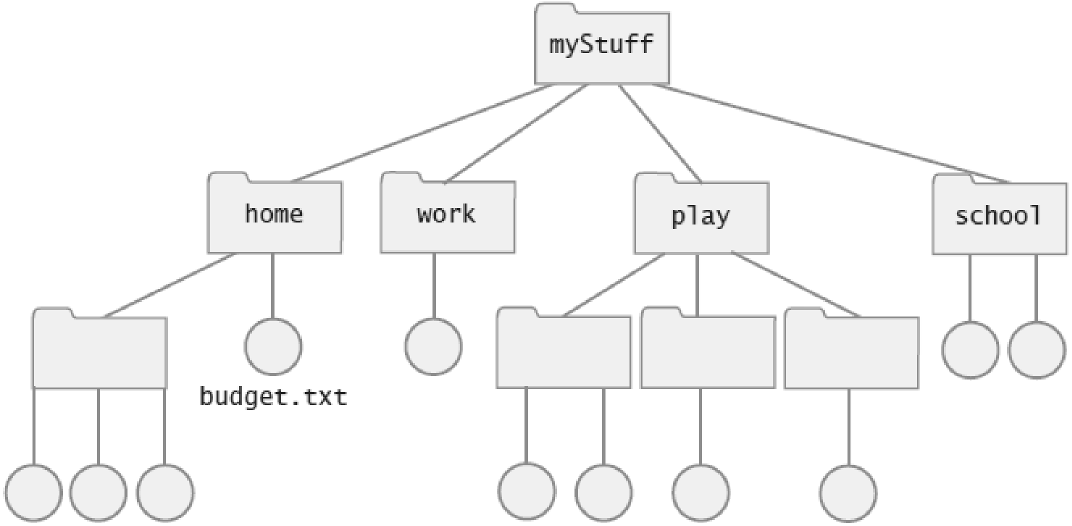
- (Example) A university's organization



A portion of a university's administrative structure

Hierarchical Organizations

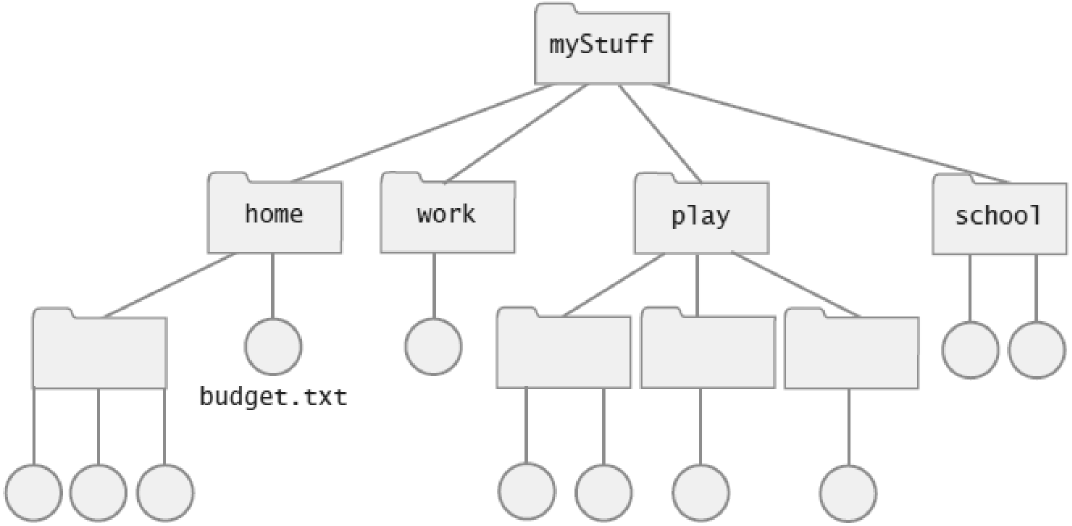
- (Example) File directories



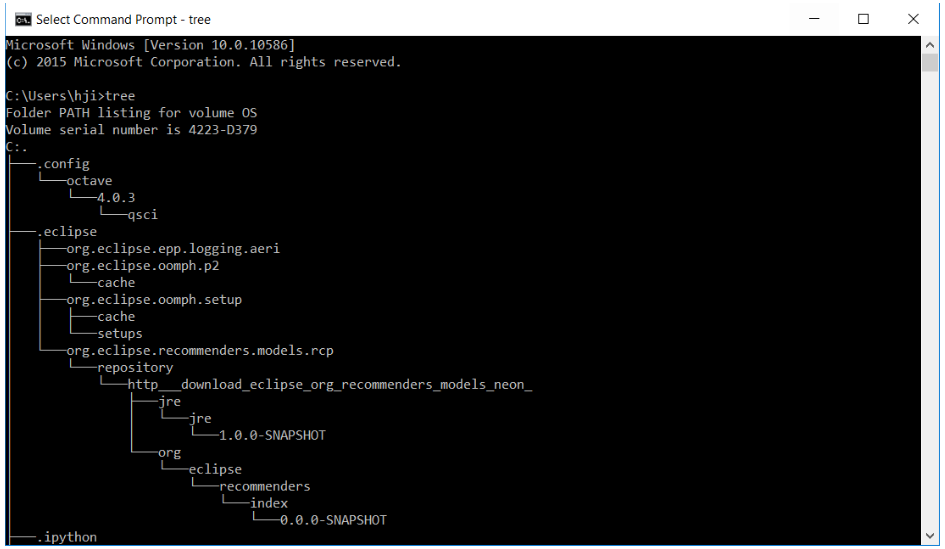
Computer files organized into folders

Hierarchical Organizations

- (Example) File directories



Computer files organized into folders

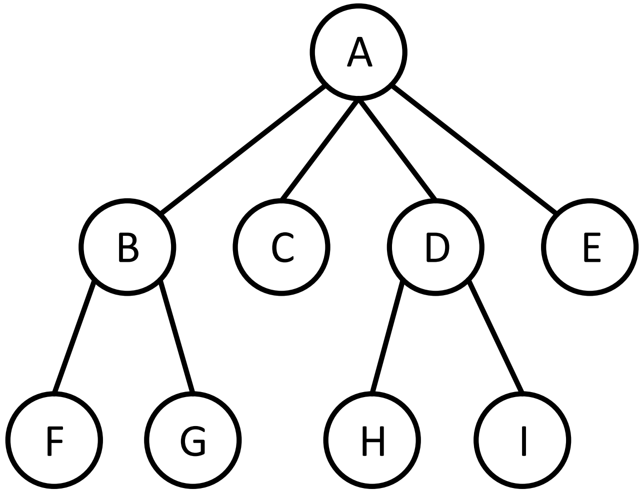


Tree command

Trees Definition

- In computer science, a tree is a widely-used data organization to place data in hierarchical structure.

- Tree
  - A collection of nodes connected by edges without having any cycle.

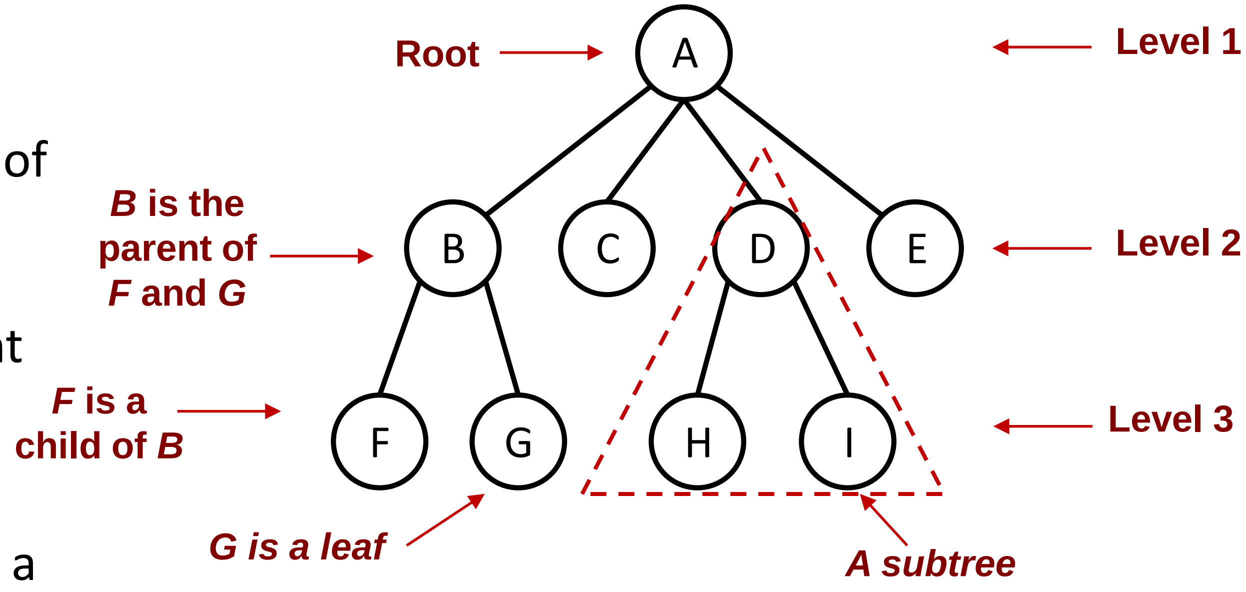


Tree (Data Structure)

Real Tree

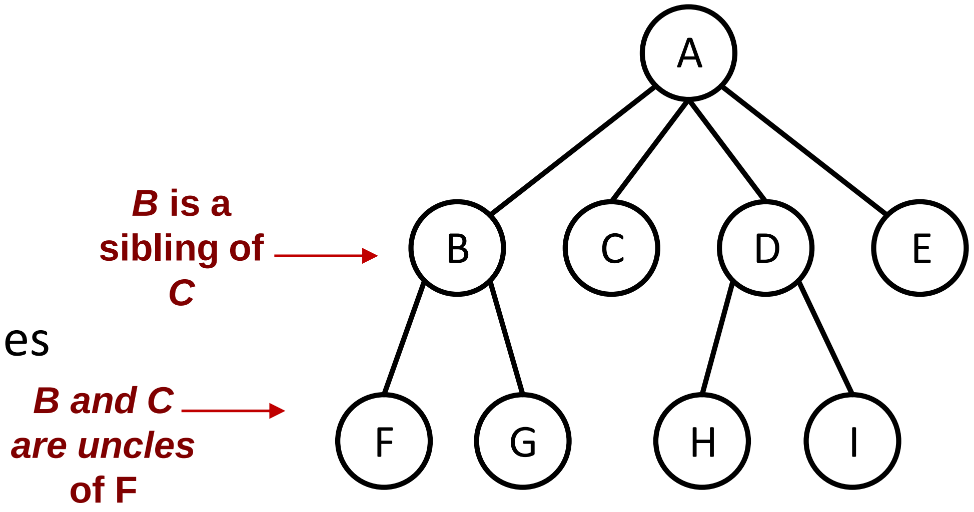
Tree Terminology

- Terminology
  - **Level**: the level of a node represents that node's hierarchy
  - **Root**: a single node at the top level
  - **Children**: the nodes at each successive level of a tree are the children of the nodes at the previous level.
  - **Parent**: a node that has children is the parent of those children
  - **Leaf**: a node has no children
  - **Subtree**: any node and its descendants form a subtree of the original tree



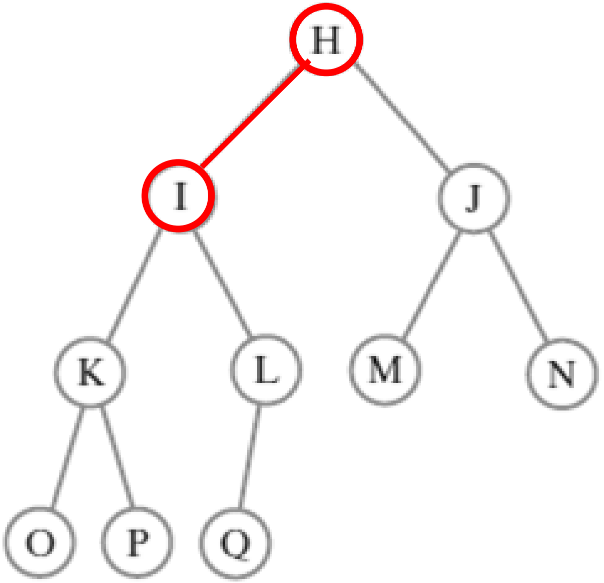
Tree Terminology

- Terminology
  - **Sibling**: Sibling nodes share the same parent node
  - **Uncles**: Siblings of that node's parent.
  - **Ancestor**: A node that is connected to all lower-level nodes.
  - **Descendant**: The connected lower-level nodes are "descendants" of the ancestor node.



Tree Terminology

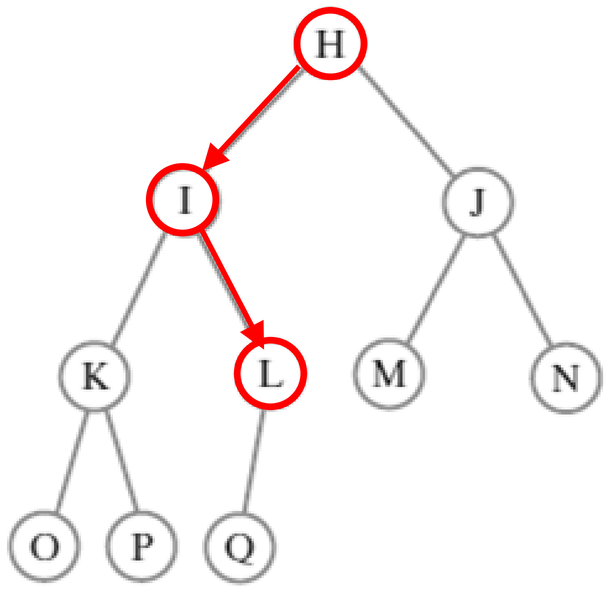
- **Edge**: connection between one node to another



Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Tree Terminology

- **Edge**: connection between one node to another
- **Path**: A sequence of nodes and edges connecting a node with a descendant.

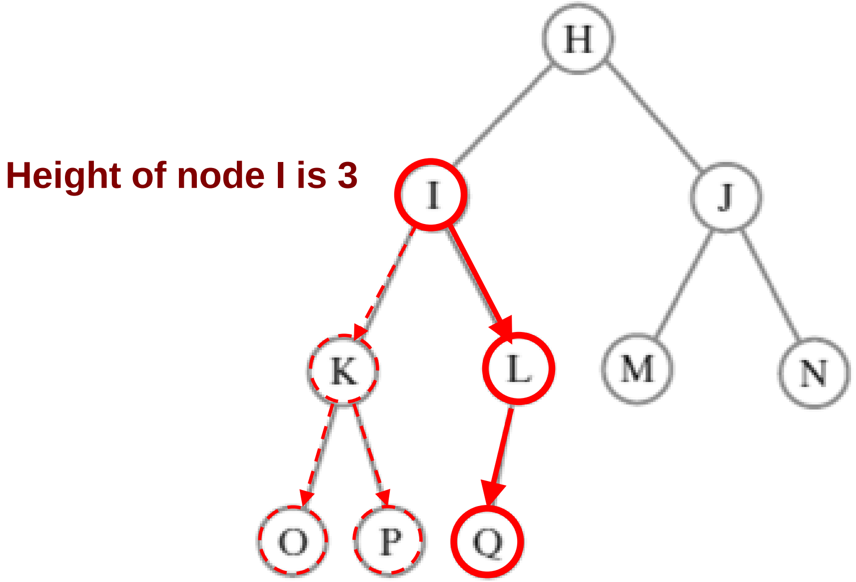


Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Tree Terminology

- **Edge**: connection between one node to another
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **Height of node**: The height of a node is the number of edges on the longest path between that node and a leaf + 1.

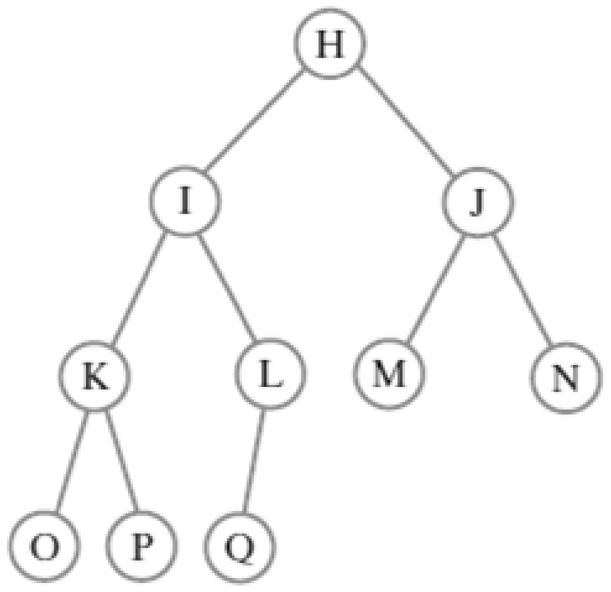
(or the number of nodes on the longest path between that node and a leaf)



Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Tree Terminology

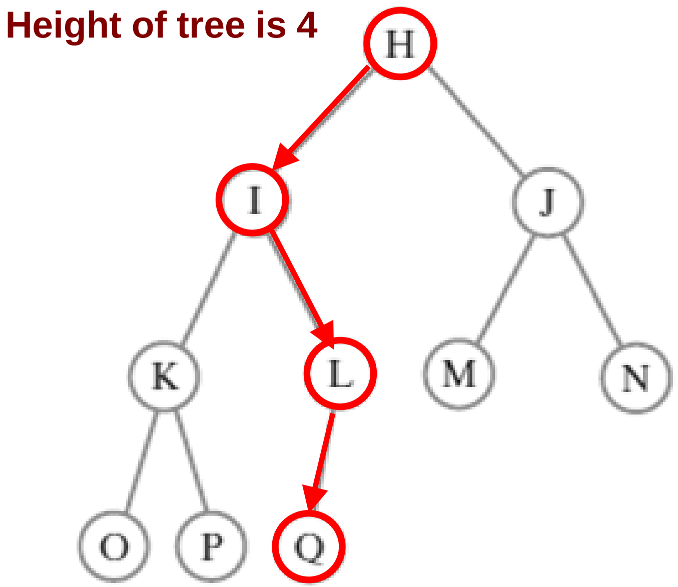
- **Edge**: connection between one node to another
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **Height of node**: The height of a node is the number of edges on the longest path between that node and a leaf + 1.
- **Height of tree**: The height of a tree is the height of its root node.



Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Tree Terminology

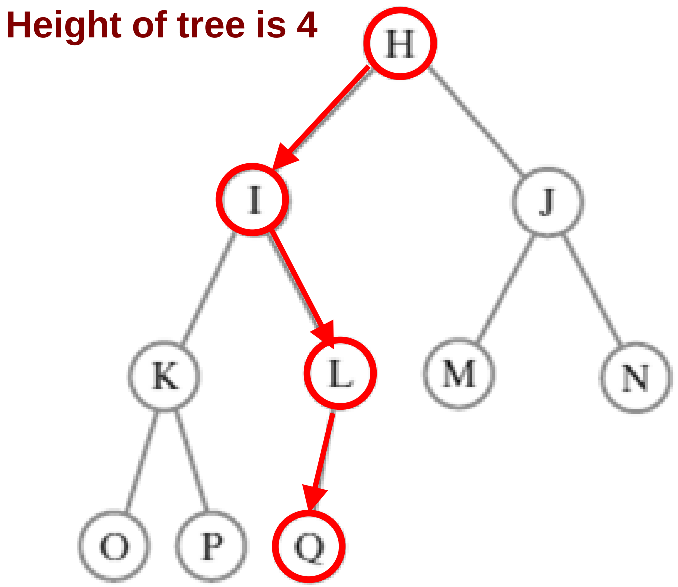
- **Edge**: connection between one node to another
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **Height of node**: The height of a node is the number of edges on the longest path between that node and a leaf + 1.
- **Height of tree**: The height of a tree is the height of its root node.



Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Tree Terminology

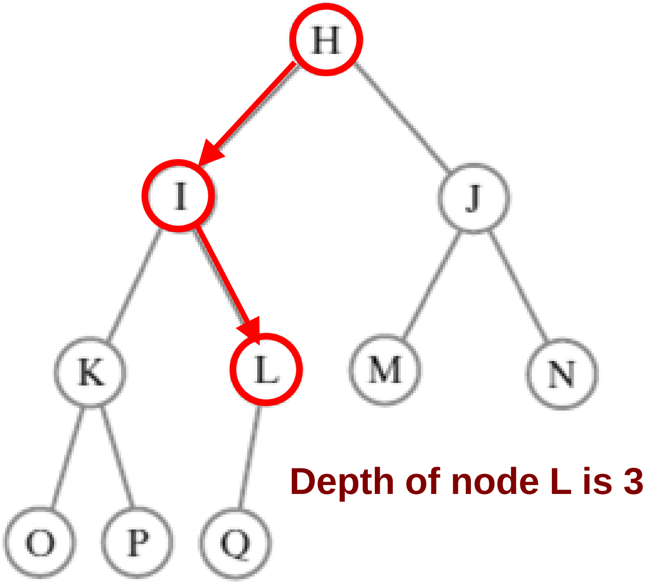
- **Edge**: connection between one node to another
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **Height of node**: The height of a node is the number of edges on the longest path between that node and a leaf + 1.
- **Height of tree**: The height of a tree is the height of its root node.



Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Tree Terminology

- **Edge**: connection between one node to another
- **Path**: A sequence of nodes and edges connecting a node with a descendant.
- **Height of node**: The height of a node is the number of edges on the longest path between that node and a leaf + 1.
- **Height of tree**: The height of a tree is the height of its root node.
- **Depth of node**: The depth of a node is the number of edges from the tree's root node to the node + 1.



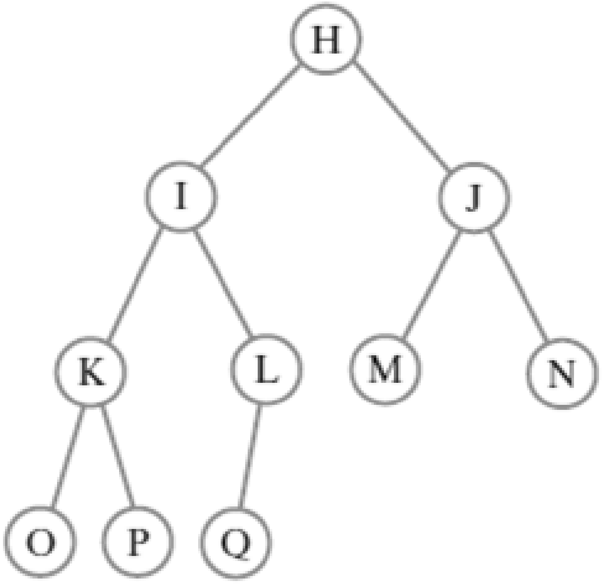
Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)
http://typeocaml.com/2014/11/26/height-depth-and-level-of-a-tree/

Common Operations in Trees

- Tree Terminology: Level, Root, Children, Parent, Leaf, Subtree, Edge, Path, Height of node, Height of tree, Depth of node

- An interface of methods common to all trees

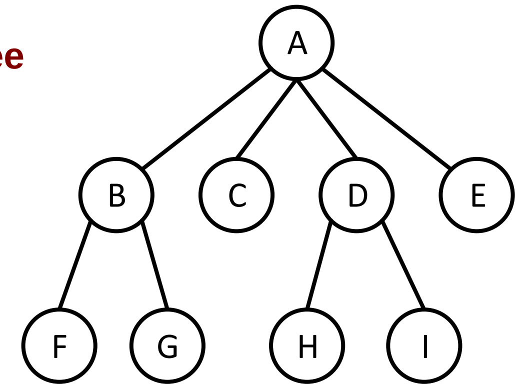
```java
package TreePackage;
public interface TreeInterface<T>
{
    public T getRootData();
    public int getHeight();
    public int getNumberOfNodes();
    public boolean isEmpty();
    public void clear();
} // end TreeInterface
```



Tree Terminology

- In general, a tree can have an arbitrary number of children.
  - **N-ary Tree**: each node has no more than n children.

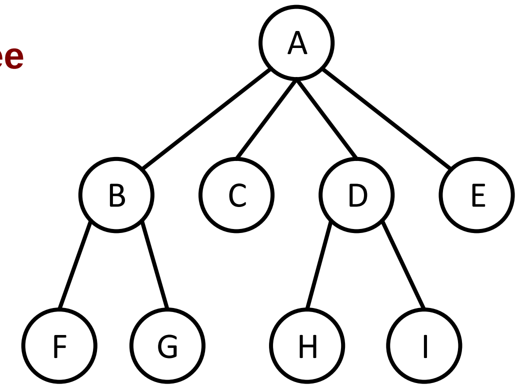
4-ary tree



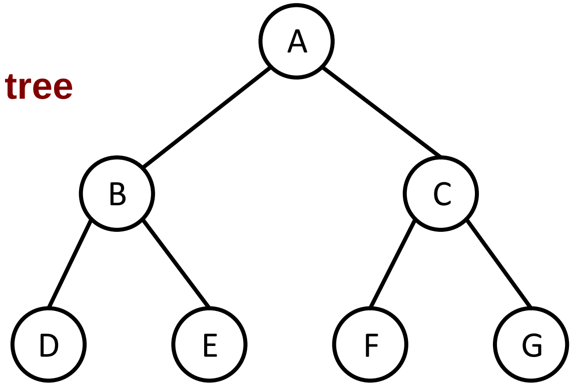
Tree Terminology

- In general, a tree can have an arbitrary number of children.
  - **N-ary Tree**: each node has no more than n children.
  - **Binary Tree**: each node has at most two children.

4-ary tree

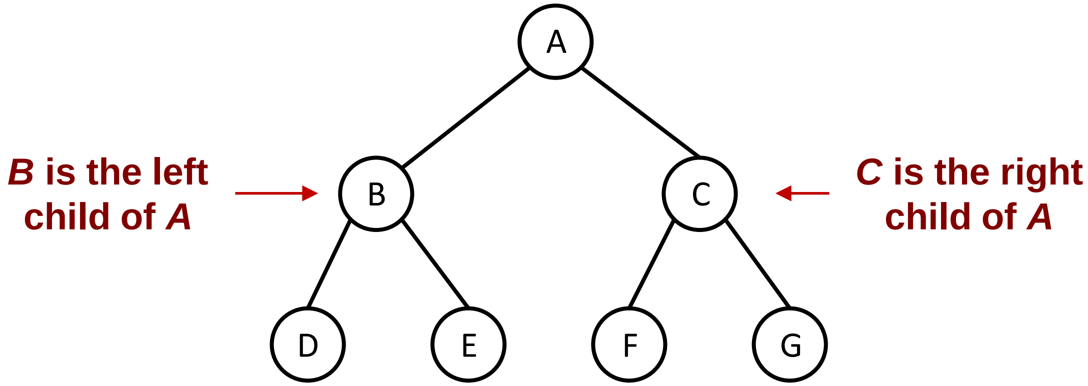


Binary tree



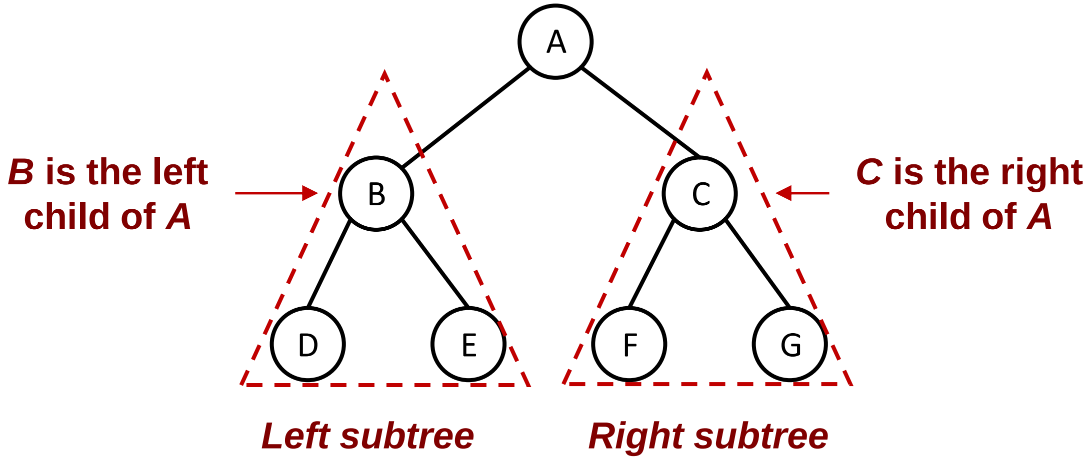
Tree Terminology

- **Binary Tree**
  - Every node in a tree can have at most two children.



Tree Terminology

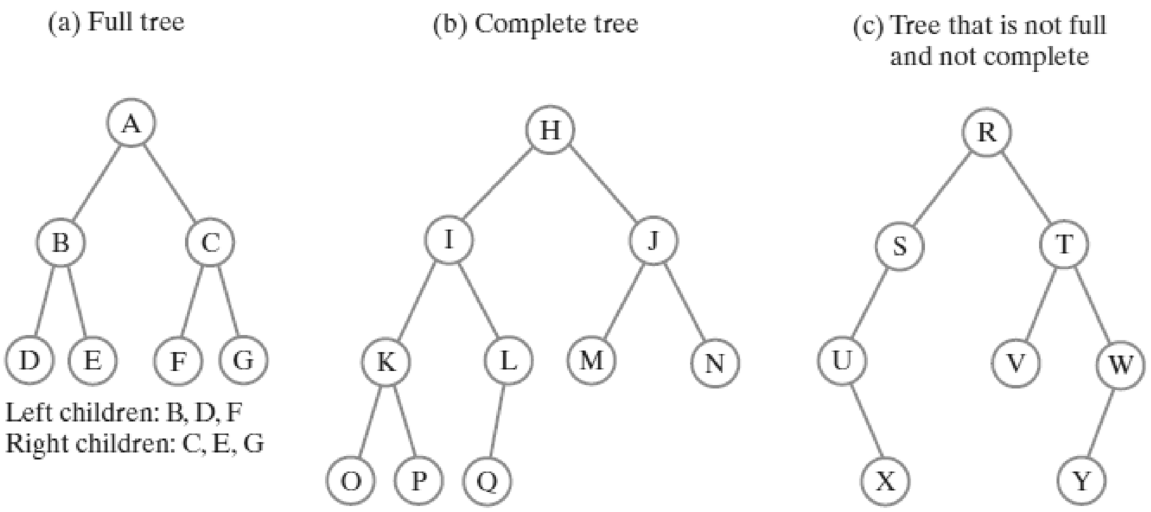
- **Binary Tree**
  - Every node in a tree can have at most two children.



Tree Terminology

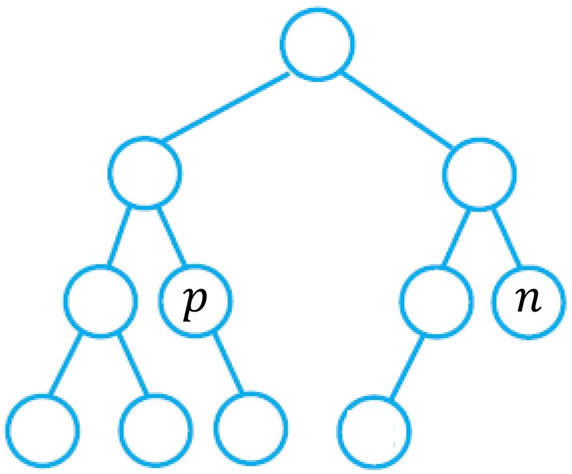
- Full Binary Tree
  - All internal nodes have two children and all leaves are at the same depth.

- Complete Binary Tree
  - An almost-full binary tree; the bottom level of the tree is filling from left to right but may not have its full complement of leaves.

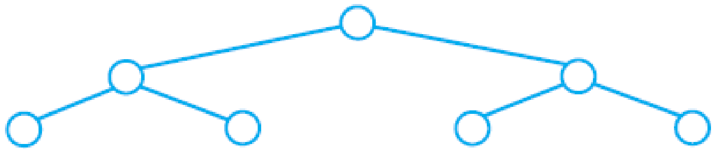


In-Class Exercise

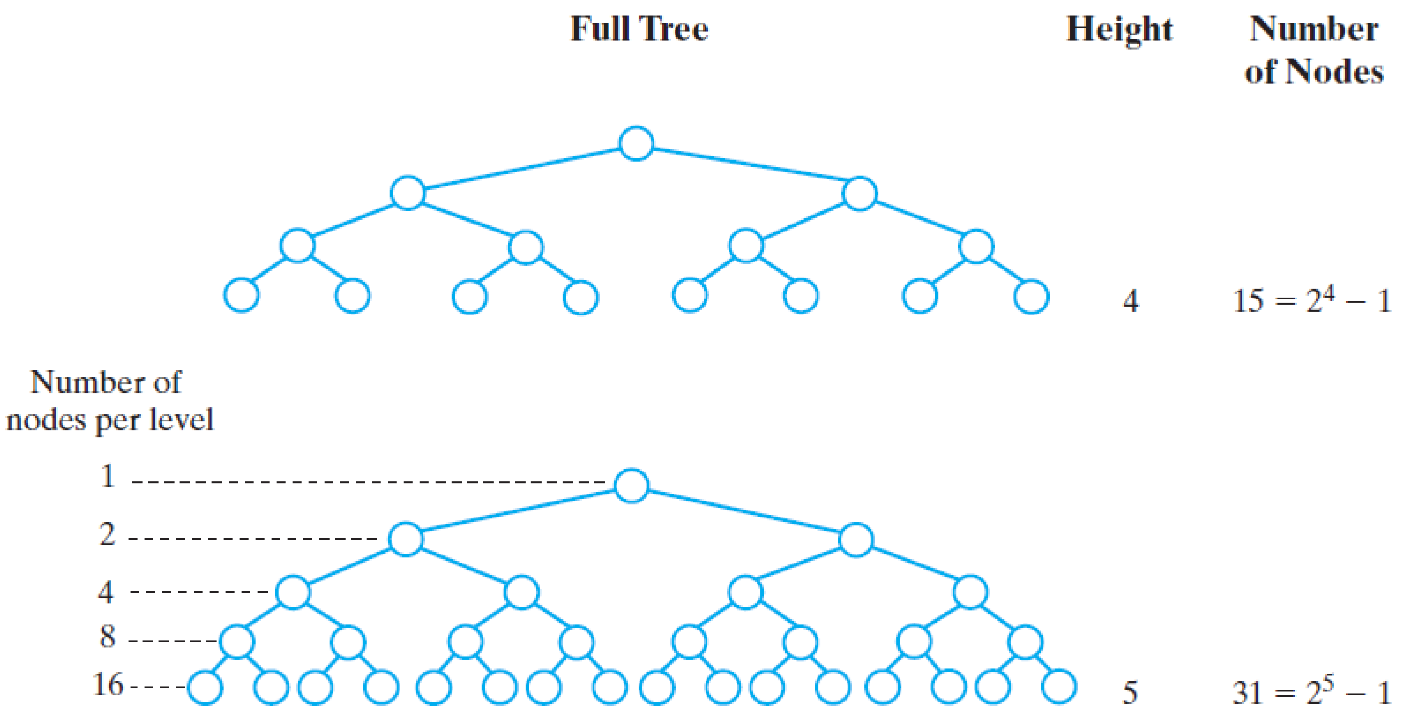
- What is the height of the tree?
- What is the height of node n?
- What is the height of node p?
- What is the depth of node n?
- What is the depth of node p?
- Is the tree complete?



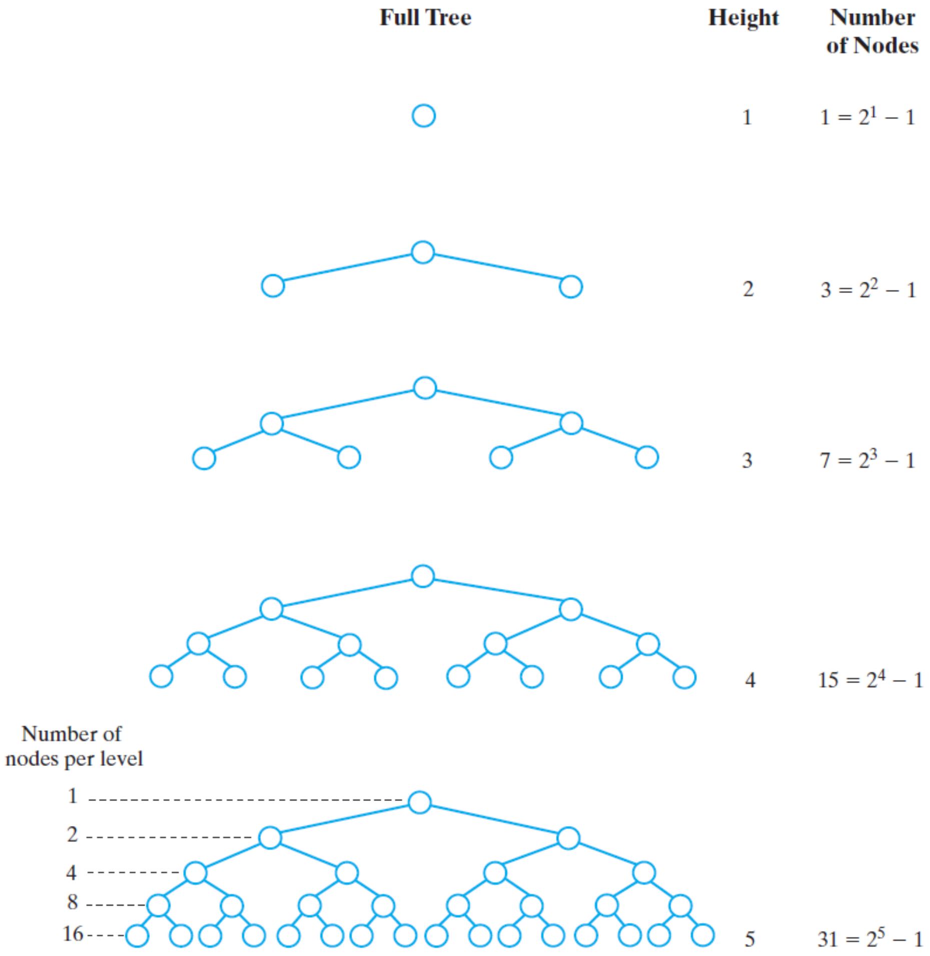
Height of Full or Complete Binary Trees

Full Tree	Height	Number of Nodes
	1	$1 = 2^1 - 1$
	2	$3 = 2^2 - 1$
	3	$7 = 2^3 - 1$

Height of Full or Complete Binary Trees



Height of Full or Complete Binary Trees



The height of a full or complete tree that has n nodes is $log_2(n+1)$ rounded up

In-Class Exercise

- In a full binary tree of height 4,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

- In a full binary tree with 10 nodes,
  - What is the height of the tree?
  - How many non-leaf nodes are in the tree?
  - How many leaves are in the tree?

Warm-up Example Questions I

- In a full binary tree of height 4,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

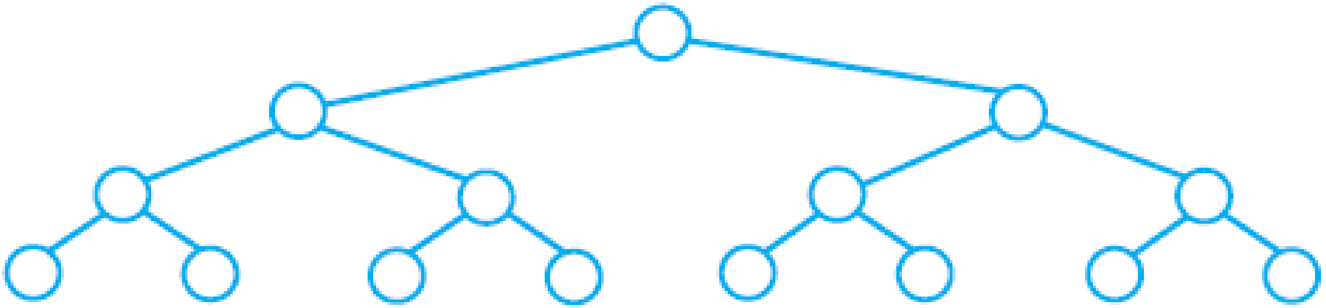


Fig.1 A Full Binary Tree of Height 4

Warm-up Example Questions II

- In a complete binary tree of height 4,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

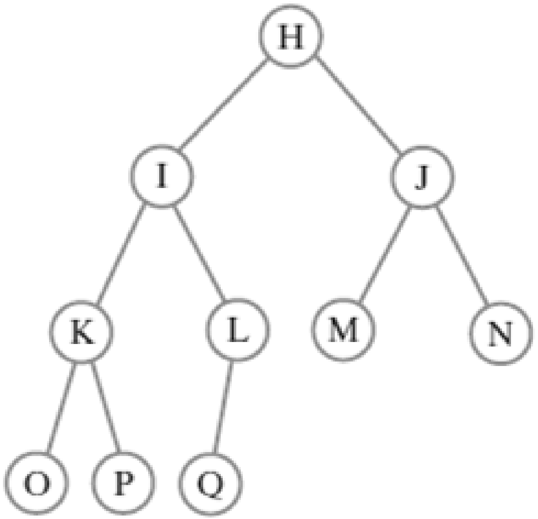


Fig.2 One of the cases

(Complete Binary Tree of Height 4)

Warm-up Example Questions II

- In a **complete** binary tree of height ***h***,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

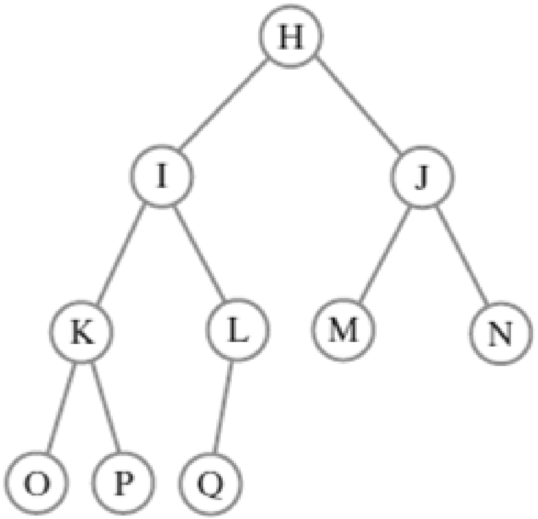


Fig.2 A Complete Binary Tree of Height 4

Warm-up Example Questions II

- In a complete binary tree of height 4,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

- In a complete binary tree with 10 nodes,
  - What is the height of the tree?
  - How many non-leaf nodes are in the tree?
  - How many leaves are in the tree?

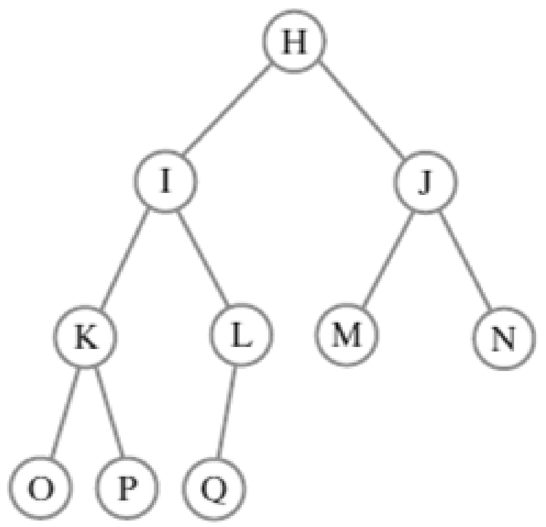


Fig.2 A Complete Binary Tree of Height 4

In-Class Exercise

- In a full binary tree of height h,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

- In a full binary tree with n nodes,
  - What is the height of the tree?
  - How many non-leaf nodes are in the tree?
  - How many leaves are in the tree?

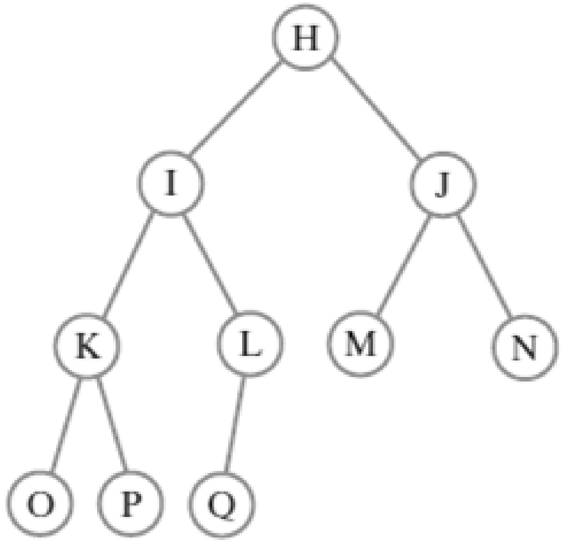
In-Class Exercise

- In a complete binary tree of height h,
  - How many nodes are in the tree?
  - How many leaves are in the tree?

- In a complete binary tree with n nodes,
  - What is the height of the tree?
  - How many non-leaf nodes are in the tree?
  - How many leaves are in the tree?

Tree Terminology

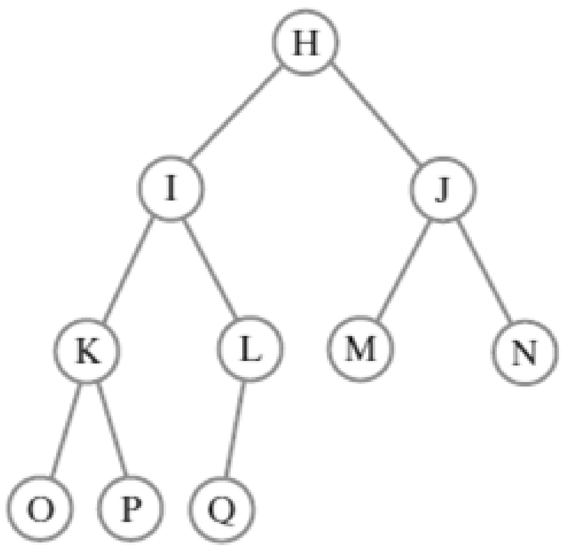
- **Balanced Binary Trees**
  - The substrees of each node in the tree differ in height by no more than 1.



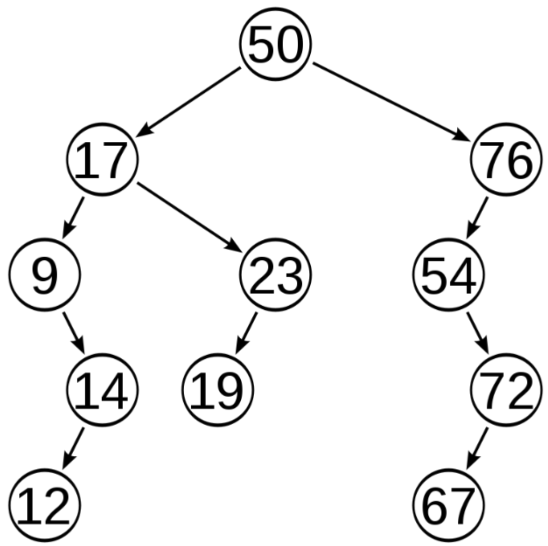
A balanced binary tree

Tree Terminology

- **Balanced Binary Trees**
  - The substrees of each node in the tree differ in height by no more than 1.



A balanced binary tree



Is this a balanced binary tree?

Tree Traversals

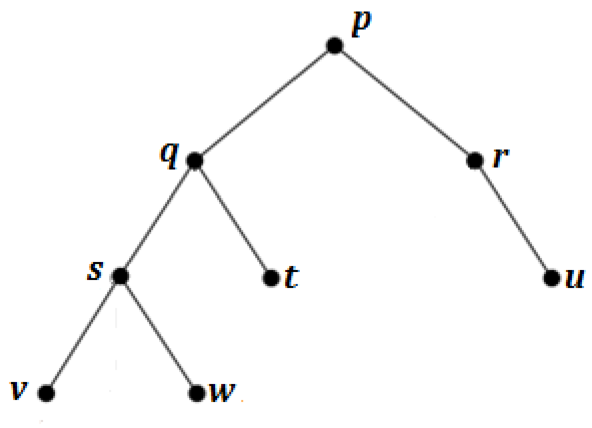
- If a tree structure is being used to store data, it is often helpful to have a systematic mechanism for writing out the data values stored at all the nodes.

- This can be accomplished by *traversing* the tree
  - Visiting each of the nodes in the tree structure.

- The common **tree traversals** are
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal
  - Level-order traversal

Tree Traversals

- Pre-order traversal:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.

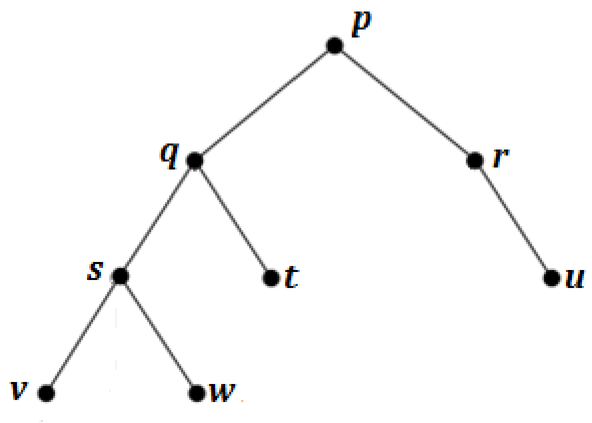


The preorder (root, left, right) traversal produces:

p, q, s, v, w, t, r, u

Tree Traversals

- In-order traversal:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.

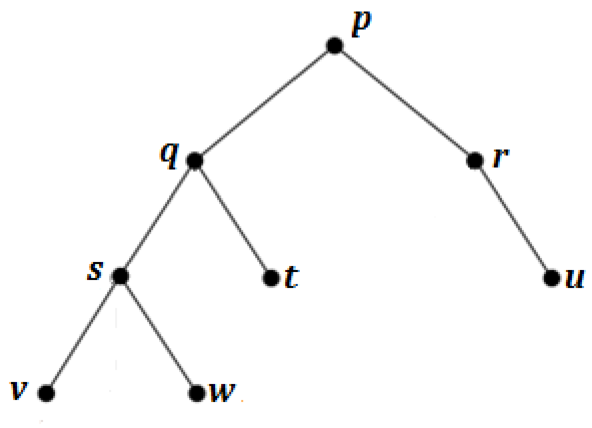


The inorder (left, root, right) traversal produces:

v, s, w, q, t, p, r, u

Tree Traversals

- Post-order traversal:
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  - Process the root.

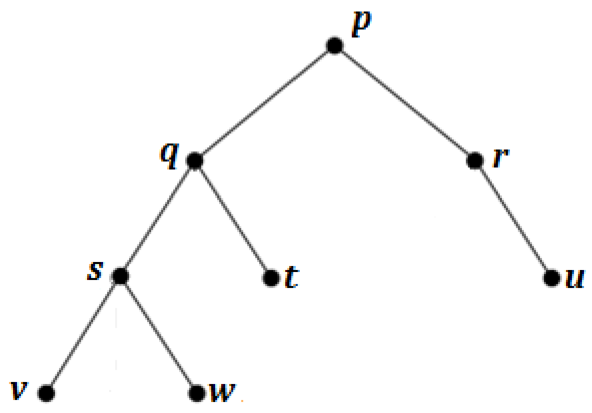


The postorder (left, right, root) traversal produces:

v, w, s, t, q, u, r, p

Tree Traversals

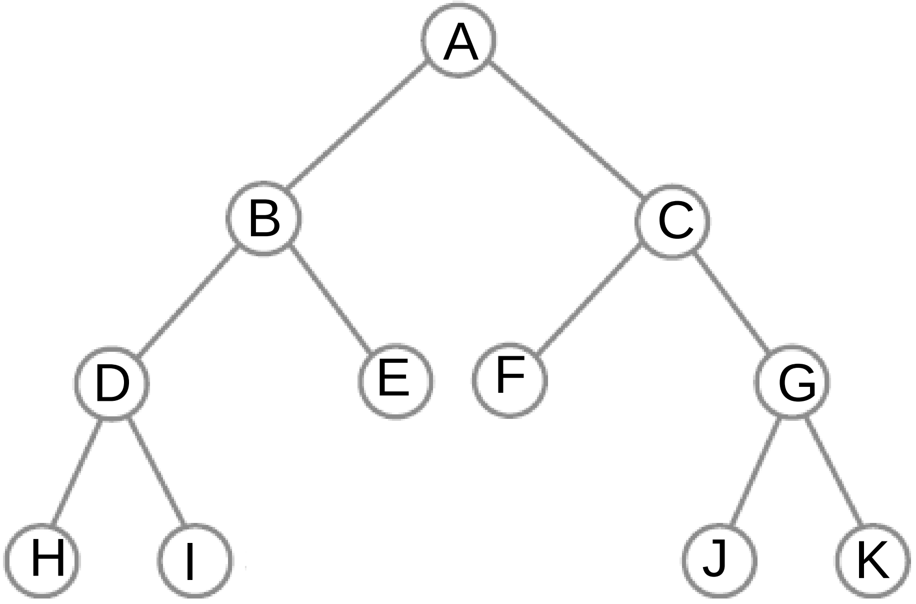
- Level-order traversal:
  - Process the root.
  - Process nodes one level at a time (visiting nodes in each level from left to right)



The level-order traversal produces:
p, q, r, s, t, u, v, w

In-Class Exercise

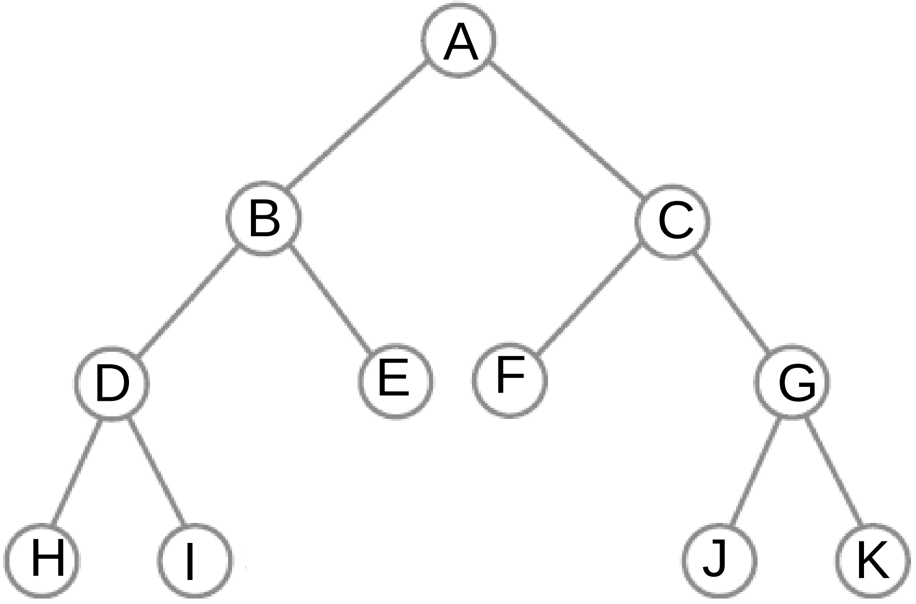
- Find the pre-order, in-order, post-order, and level-order traversal for the following tree:



In-Class Exercise

- Find the pre-order, in-order, post-order, and level-order traversal for the following tree:

Pre-order: *A, B, D, H, I, E, C, F, G, J, K*
In-order: *H, D, I, B, E, A, F, C, J, G, K*
Post-order: *H, I, D, E, B, F, J, K, G, C, A*
Level-order: *A, B, C, D, E, F, G, H, I, J, K*



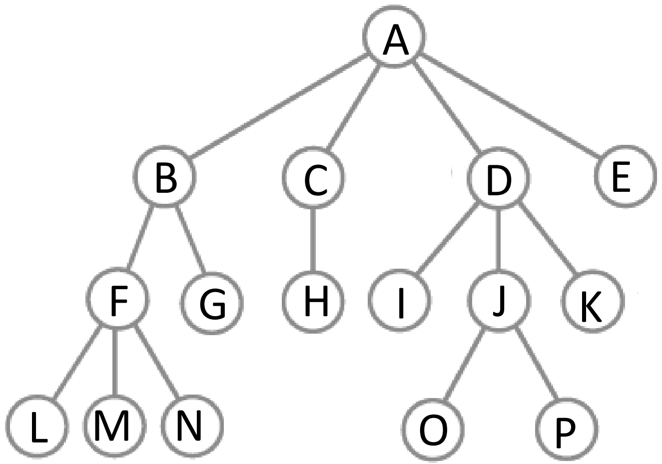
In-Class Exercise

- **Draw a binary tree** that gives the following traversals:

Pre-order: *A, B, D, H, I, E, C, F, G, J, K*
In-order: *H, D, I, B, E, A, F, C, J, G, K*

In-Class Exercise

- Find the pre-order, in-order, post-order, and level-order traversal for the following tree:



Tree Traversals

- An interface of traversal methods for a tree

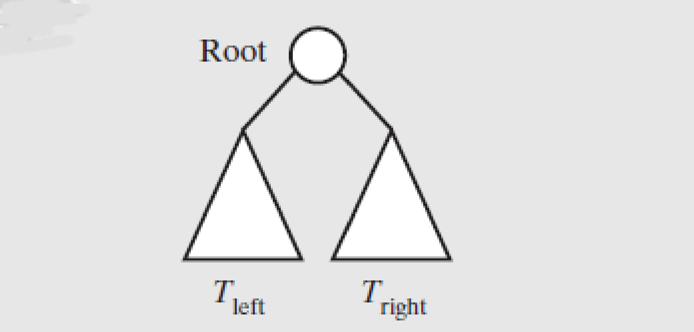
```java
package TreePackage;
import java.util.Iterator;
public interface TreeIteratorInterface<T>
{
    public Iterator<T> getPreorderIterator();
    public Iterator<T> getPostorderIterator();
    public Iterator<T> getInorderIterator();
    public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```

- Pre-order traversal:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
- In-order traversal:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.
- Post-order traversal:
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  - Process the root.
- Level-order traversal:
  - Process the root.
  - Process nodes one level at a time (visiting nodes in each level from left to right)

Constructing a Binary Tree

- An interface for a binary tree

```java
package TreePackage;
public interface BinaryTreeInterface<T> extends TreeInterface<T>,
                                                TreeIteratorInterface<T>
{
   /** Sets this binary tree to a new one-node binary tree.
       @param rootData  The object that is the data for the new tree's root.
   */
   public void setTree(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
```



where T_{left} and T_{right} are binary trees.

Constructing a Binary Tree

- An interface for a binary tree

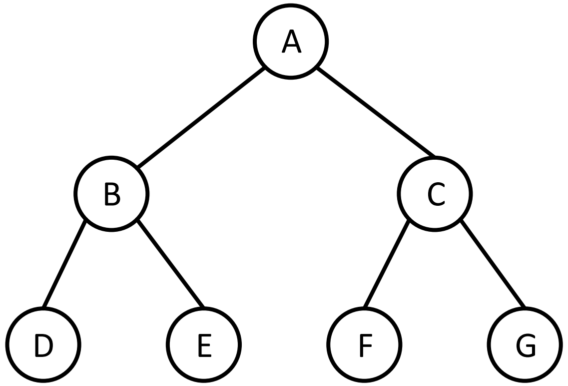
```java
package TreePackage;
public interface BinaryTreeInterface<T> extends TreeInterface<T>,
                                                TreeIteratorInterface<T>
{
   /** Sets this binary tree to a new one-node binary tree.
       @param rootData  The object that is the data for the new tree's root.
   */
   public void setTree(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
```

```java
package TreePackage;
import java.util.Iterator;
public interface TreeIteratorInterface<T>
{
   public Iterator<T> getPreorderIterator();
   public Iterator<T> getPostorderIterator();
   public Iterator<T> getInorderIterator();
   public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```

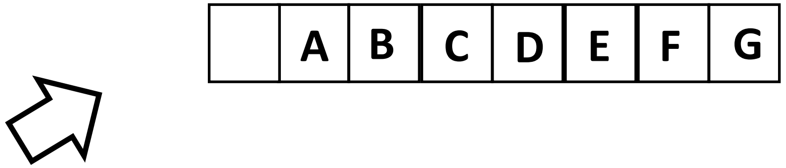
```java
package TreePackage;
public interface TreeInterface<T>
{
   public T getRootData();
   public int getHeight();
   public int getNumberOfNodes();
   public boolean isEmpty();
   public void clear();
} // end TreeInterface
```

Tree Representations

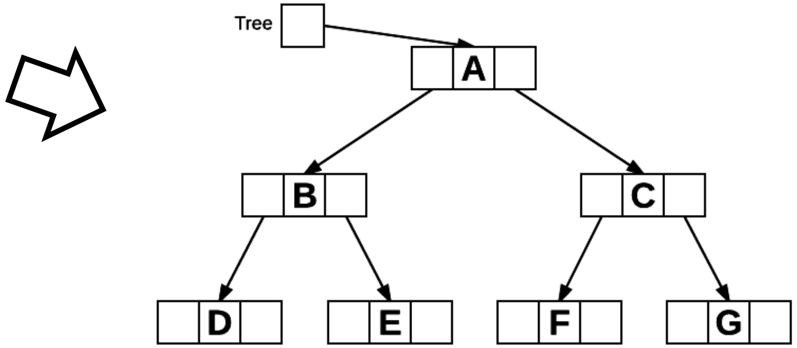
- Storage choices:
  - Array representation (contiguous structure)
  - Node representation (linked structure)



A binary tree



Array representation



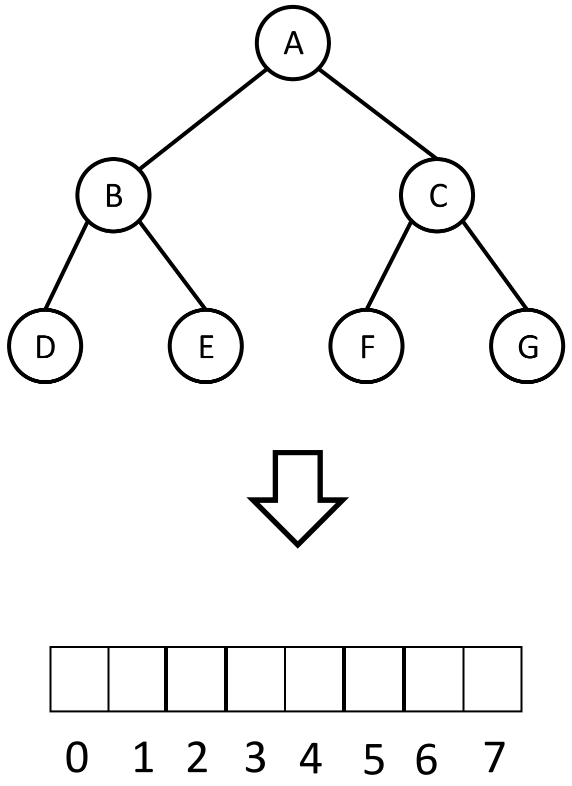
Node representation

Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]

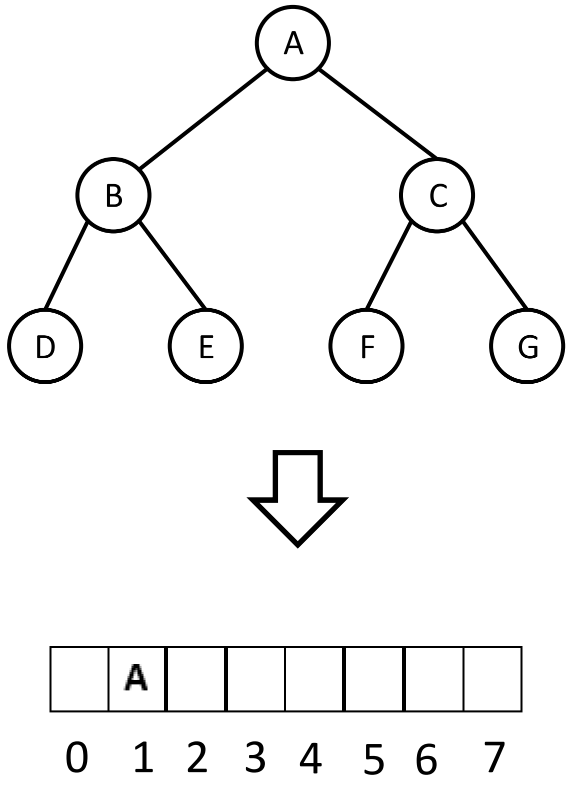
Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



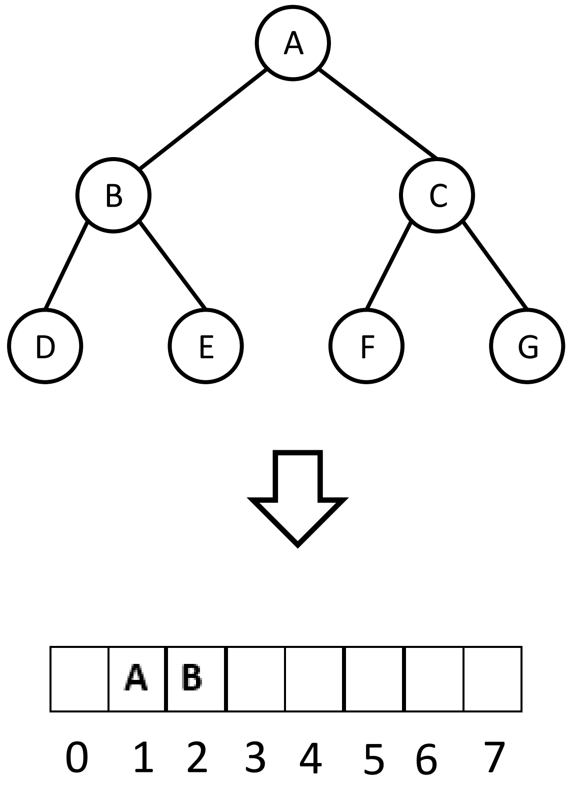
Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



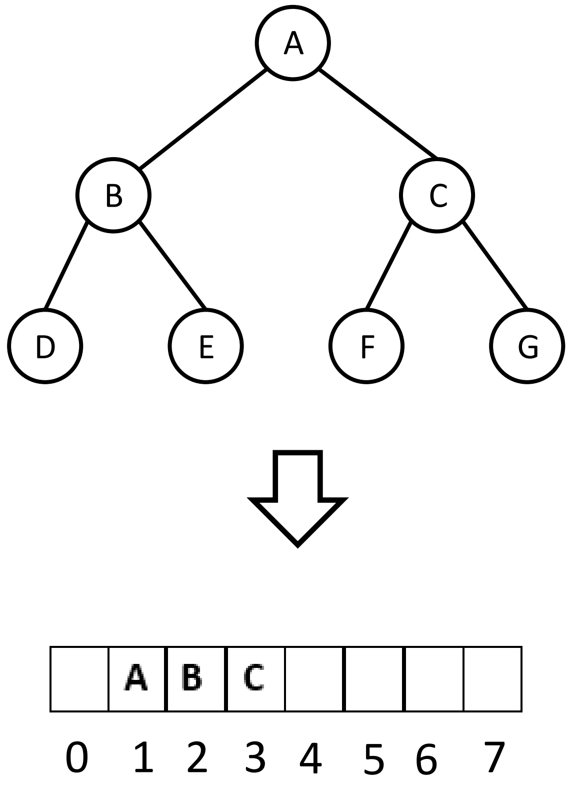
Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - **Left child** at component [2*i*]
    - Right child at component [2*i* + 1]



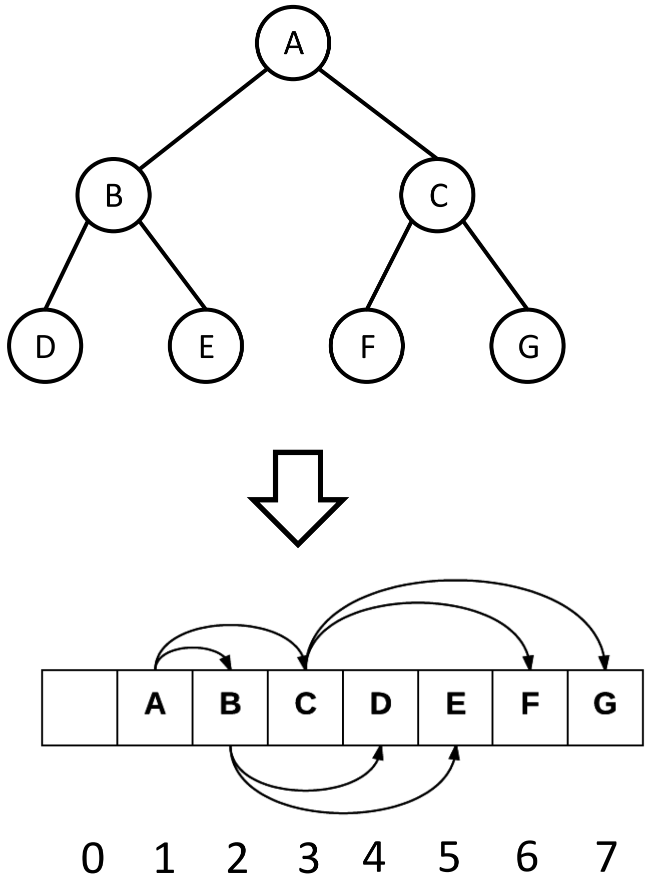
Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - **Right child** at component [2*i* + 1]



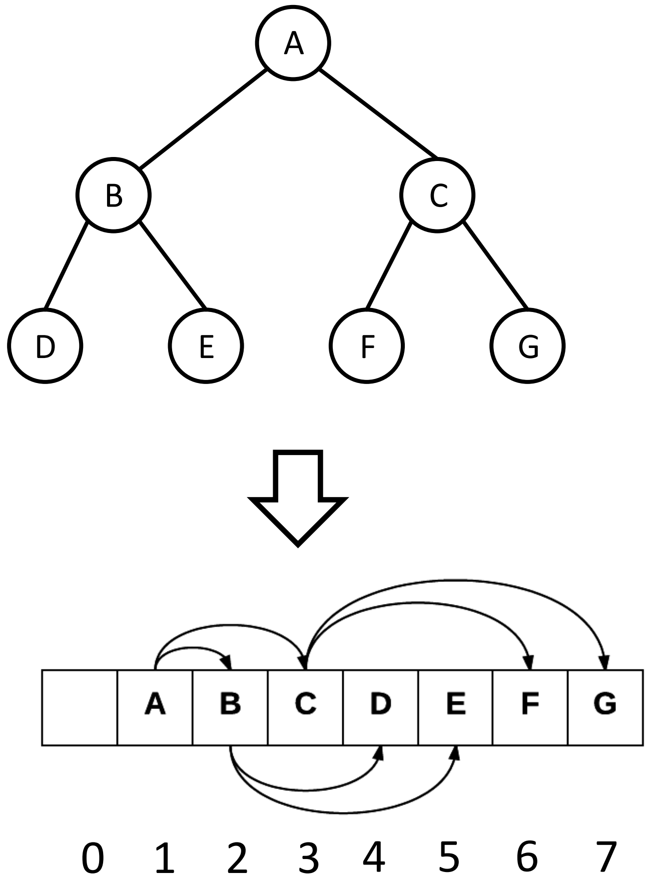
Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



Array Representation

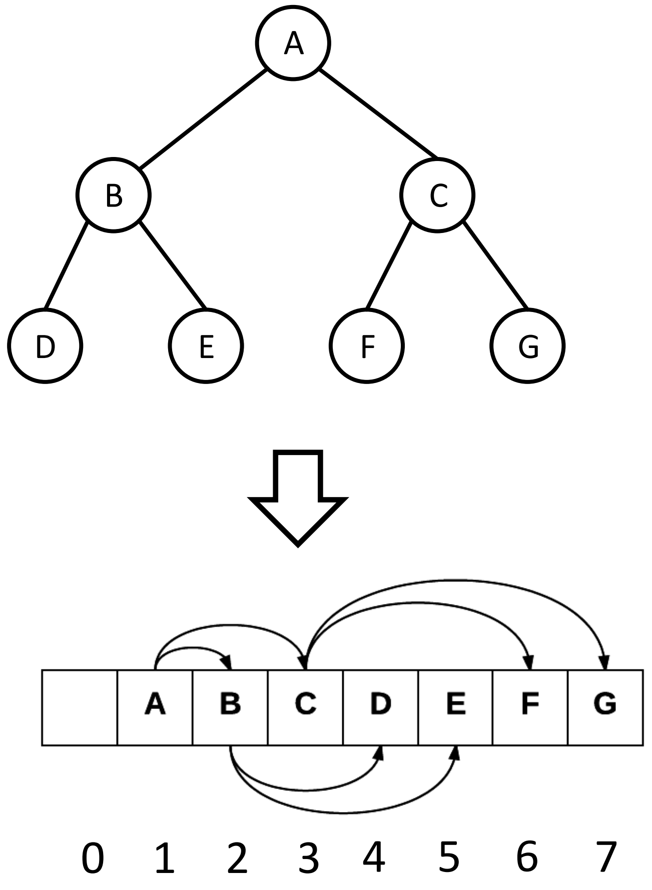
- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



- The actual links between the nodes are not stored, but are determined by the formula.

Array Representation

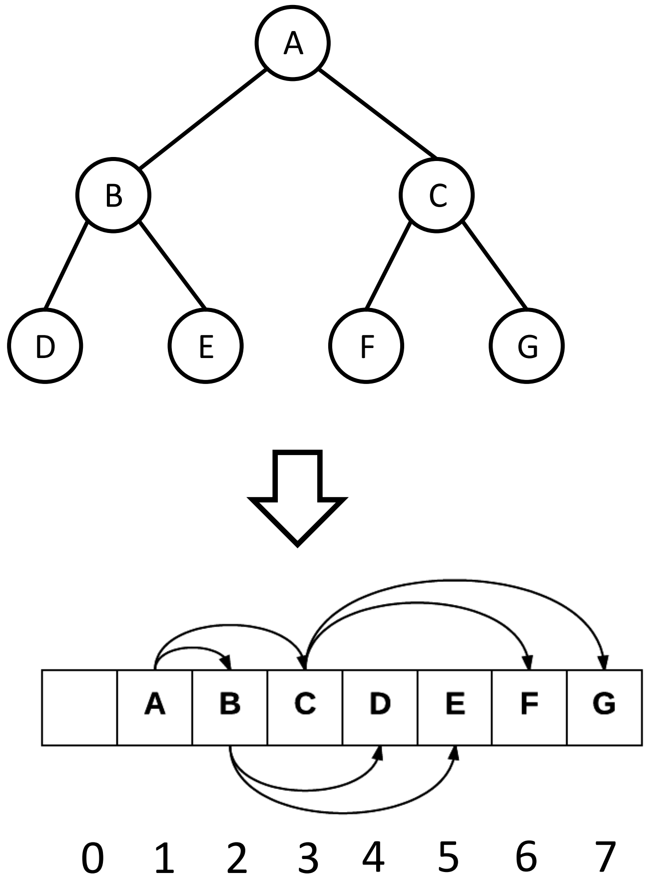
- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



Question: Suppose the data for a non-root node appears in component [*i*] of the array. What is the location of the data for its parent?

Array Representation

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



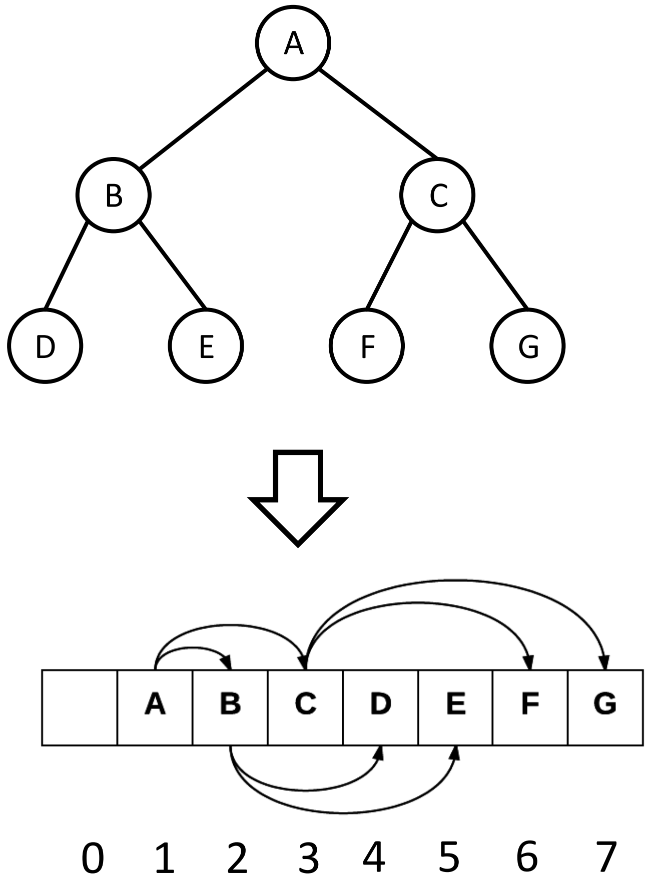
Question: Suppose the data for a non-root node appears in component [*i*] of the array. What is the location of the data for its parent?

At array index *i*/2

Array Representation

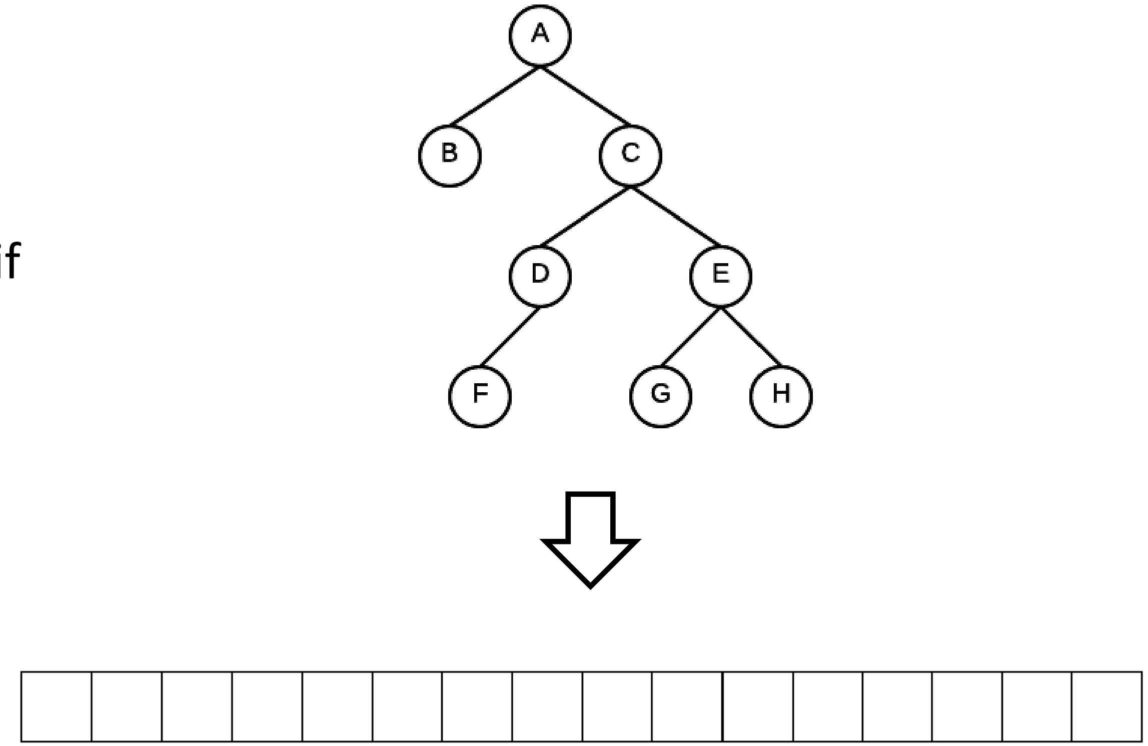
- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]

- The actual links between the nodes are not stored, but are determined by the formula.



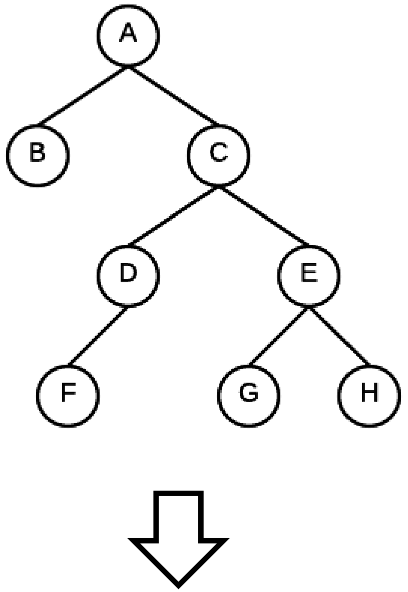
In-Class Exercise

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



In-Class Exercise

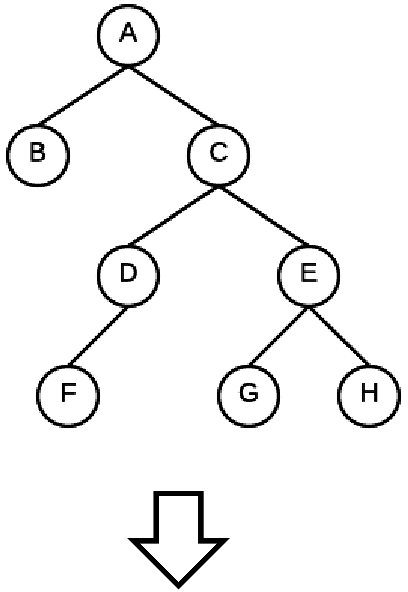
- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]



	A	B	C			D	E					F		G	H

In-Class Exercise

- Formulas for the array representation:
  - The data from the root always appears in the [1] component of the array.
  - Suppose the data for a node appears in component [*i*] of the array. Then its children (if they exist) always have their data at these locations:
    - Left child at component [2*i*]
    - Right child at component [2*i* + 1]

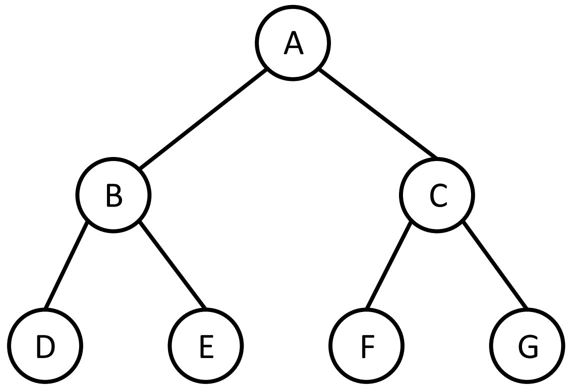


	A	B	C			D	E					F		G	H

- Disadvantages of an array implementation of a tree:
  - The memory wasted in an unbalanced tree.
  - fixed array size, which makes it harder to grow.

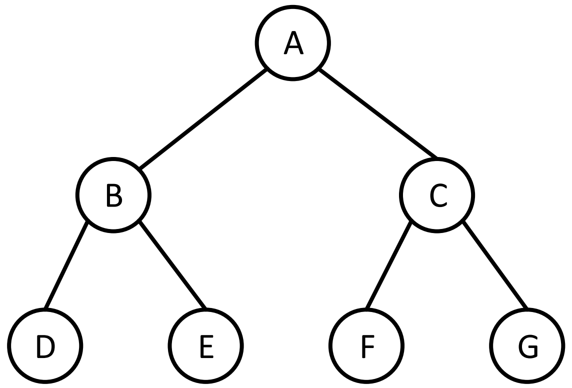
Node Representation

- Formulas for the node representation:
  - Each node of a binary tree can be stored as an object of a binary tree node class.
  - The class contains private instance variables that are references to other nodes in the tree.
  - An entire tree is represented as a reference to the root node.

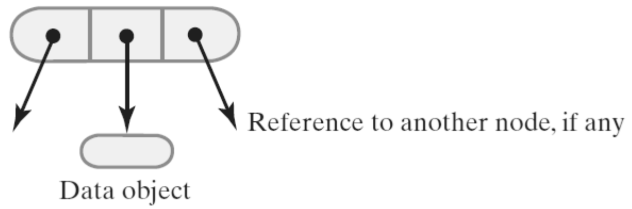


Node Representation

- Formulas for the node representation:
  - Each node of a binary tree can be stored as an object of a binary tree node class.
  - The class contains private instance variables that are references to other nodes in the tree.
  - An entire tree is represented as a reference to the root node.



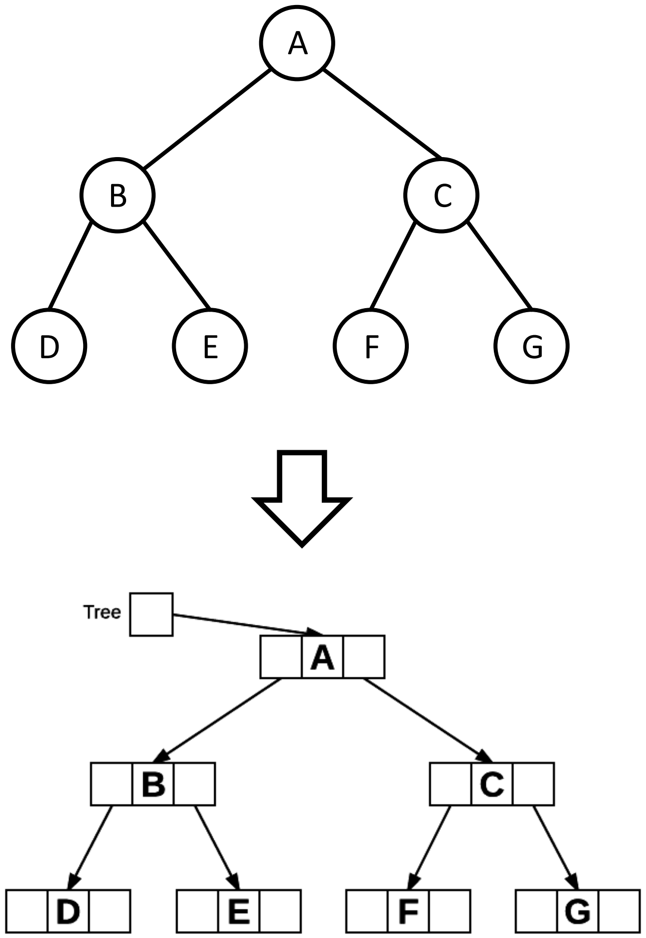

```
class BinaryNode<T>
{
  private T data;
  private BinaryNode<T> leftChild;
  private BinaryNode<T> rightChild;
  ...
}
```



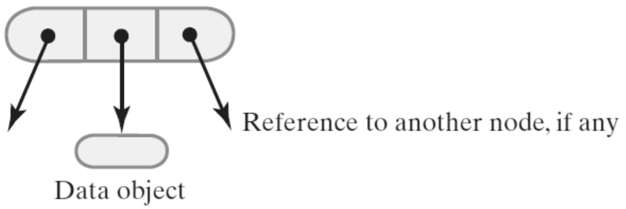
A node in a binary tree

Node Representation

- Formulas for the node representation:
  - Each node of a binary tree can be stored as an object of a binary tree node class.
  - The class contains private instance variables that are references to other nodes in the tree.
  - An entire tree is represented as a reference to the root node.



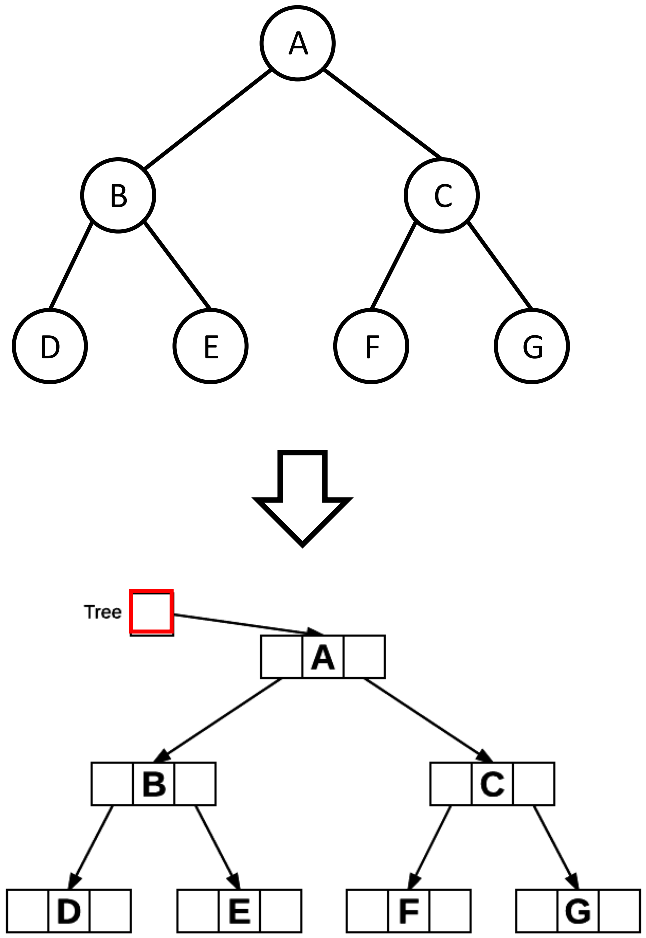

```
class BinaryNode<T>
{
  private T data;
  private BinaryNode<T> leftChild;
  private BinaryNode<T> rightChild;
  . . .
}
```



A node in a binary tree

Node Representation

- Formulas for the node representation:
  - Each node of a binary tree can be stored as an object of a binary tree node class.
  - The class contains private instance variables that are references to other nodes in the tree.
  - An entire tree is represented as a reference to the root node.
    - The reference to the root is similar to the head of a linked list, providing a starting point to access all the nodes in the tree.
    - We could include other things in the tree node.

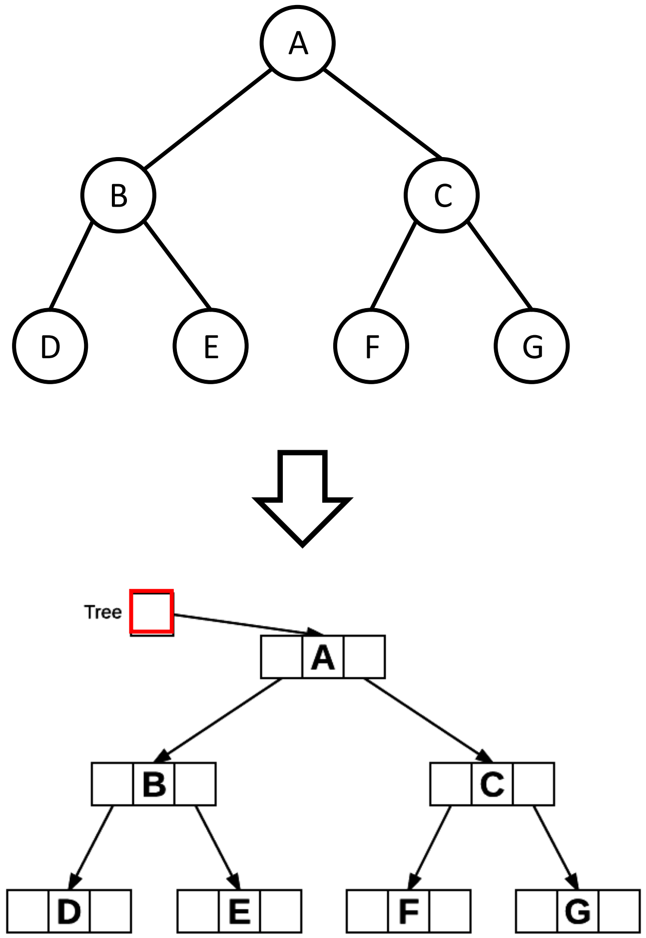


Today

- This Class
  - Tree Representations
  - **Implementation of Binary Nodes**

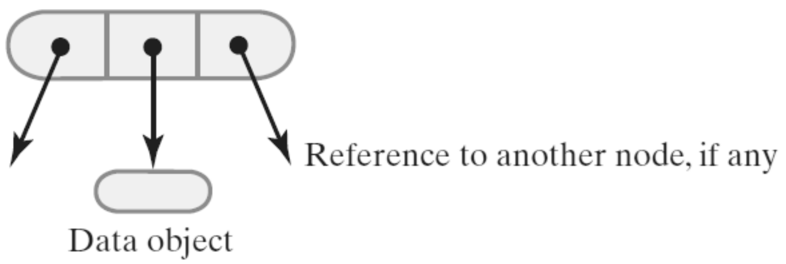
Node Representation

- Formulas for the node representation:
    - Each node of a binary tree can be stored as an object of a binary tree node class.
    - The class contains private instance variables that are references to other nodes in the tree.
    - An entire tree is represented as a reference to the root node.
        - The reference to the root is similar to the head of a linked list, providing a starting point to access all the nodes in the tree.
        - We could include other things in the tree node.



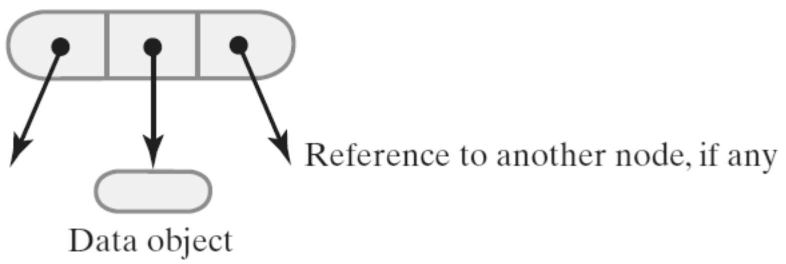
Implementation of Binary Nodes

```
class BinaryNode<T>
{
  private T data;
  private BinaryNode<T> leftChild;
  private BinaryNode<T> rightChild;
   ...









}
```



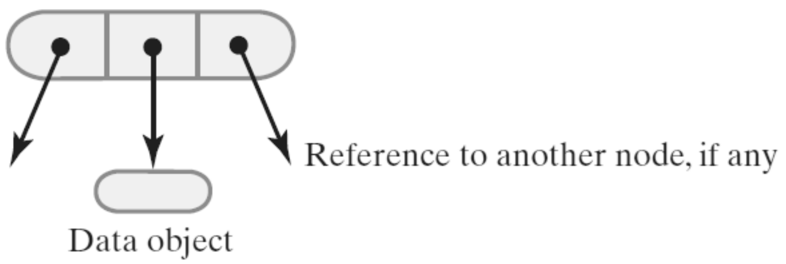
Implementation of Binary Nodes

```
class BinaryNode<T>
{
  private T data;
  private BinaryNode<T> leftChild;
  private BinaryNode<T> rightChild;
  ...
   // Constructors
   // Get and Set methods
   // Boolean method
   //    isLeaf(): return True if a leaf node, otherwise, False)
   // Other methods
   //    getNumberOfNodes(): Counts the nodes in the subtree rooted at this node.
   //    getHeight(): Computes the height of the subtree rooted at this node.
   //    copy(): Copies the subtree rooted at this node.
}
```



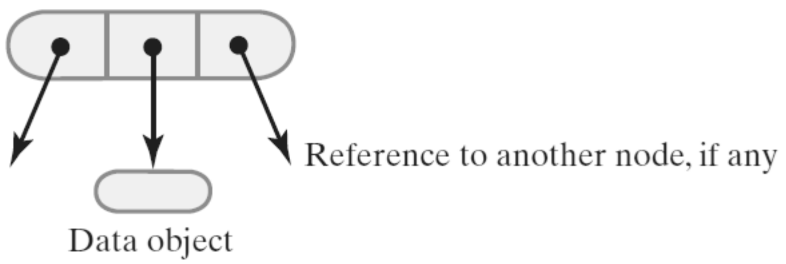
Constructors in BinaryNode Class

```java
package TreePackage;
class BinaryNode<T>
{
   private T data;
   private BinaryNode<T> leftChild;
   private BinaryNode<T> rightChild;

   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
```



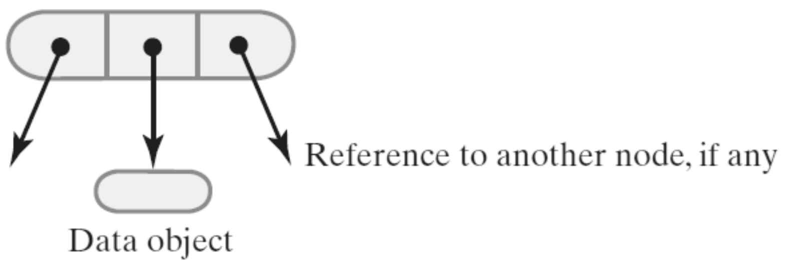
Methods in BinaryNode Class

```
/** 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  The node that is this node's left child. */
public BinaryNode<T> getLeftChild()
{
   return leftChild;
} // end getLeftChild
```



Methods in BinaryNode Class

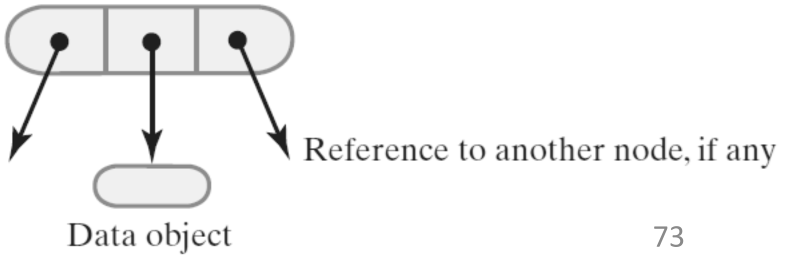
```java
public void setLeftChild(BinaryNode<T> newLeftChild)
{
   leftChild = newLeftChild;
} // end setLeftChild

/** 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);
} // end isLeaf
```



In-Class Exercise

- Write implementations of getRightChild, setRightChild, and hasRightChild.

```
class BinaryNode<T>
{
  private T data;
  private BinaryNode<T> leftChild;
  private BinaryNode<T> rightChild;
   . . .
}
```

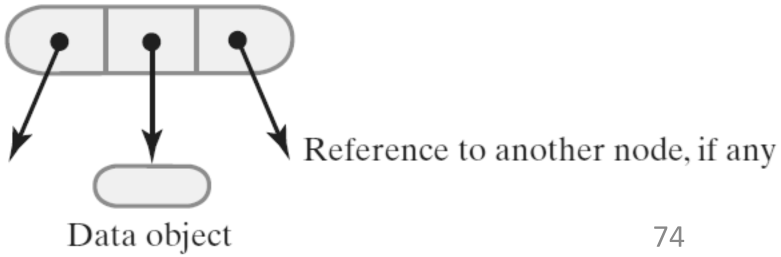


In-Class Exercise

- Write implementations of getRightChild, setRightChild, and hasRightChild.

```
public BinaryNode<T> getRightChild()
{
   return rightChild;
}

public void setRightChild(BinaryNode<T> newRightChild)
{
   rightChild = newRightChild;
}

public boolean hasRightChild()
{
   return rightChild != null;
}
```

```
class BinaryNode<T>
{
  private T data;
  private BinaryNode<T> leftChild;
  private BinaryNode<T> rightChild;
   . . .
}
```



Pre-order Traversing

```
public void preorderTraverse()
{
		preorderTraverse(root);
}

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

- Pre-order traversal:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.

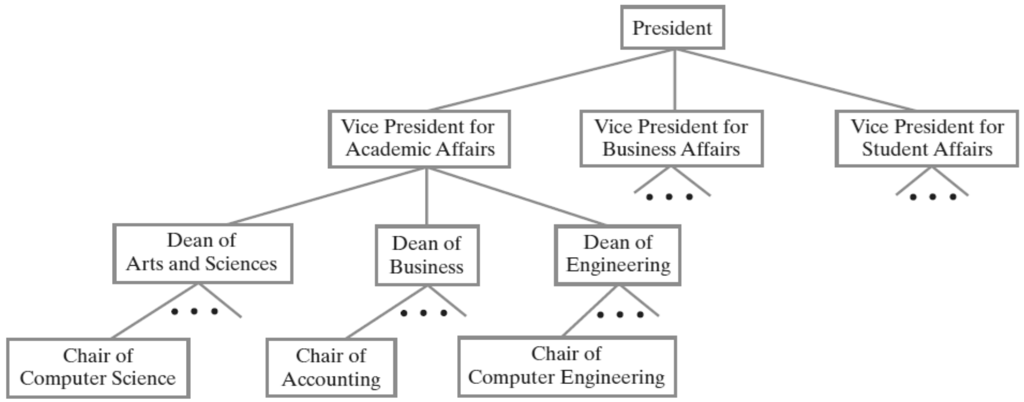
Pre-order Traversing

```
public void preorderTraverse()
{
		preorderTraverse(root);
}

private void preorderTraverse(BinaryNode<T> node)
{






}
```

- Pre-order traversal:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.

Pre-order Traversing

```
public void preorderTraverse()
{
        preorderTraverse(root);
}

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



- Pre-order traversal:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.

In-Class Exercise

- Write the method for In-order Traversing

- In-order traversal:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.

In-order Traversing

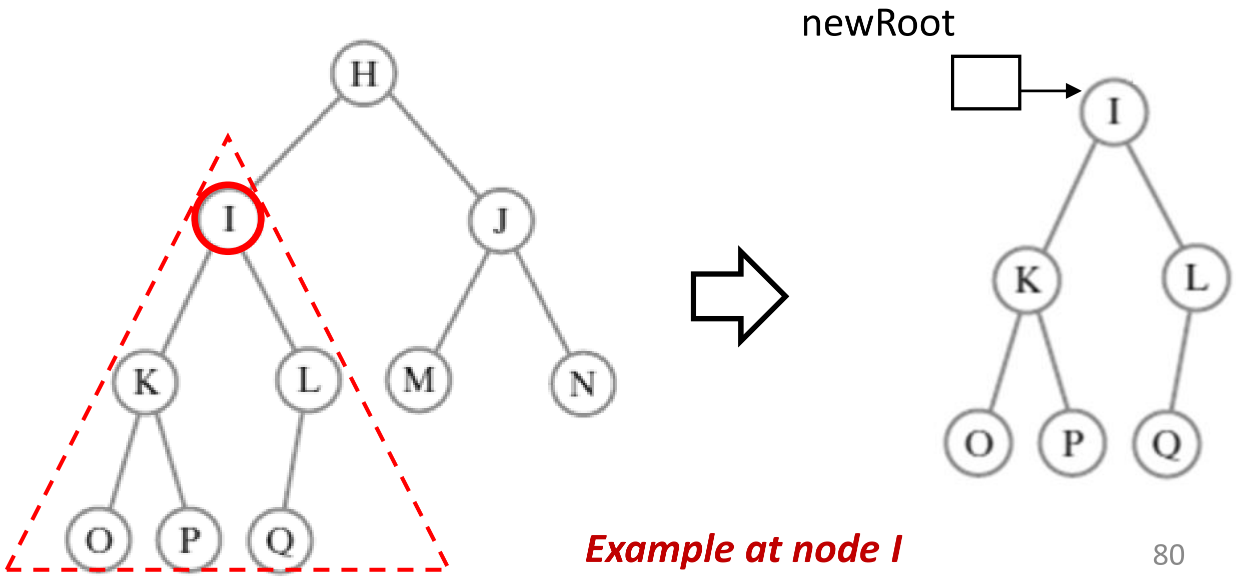
```
public void inorderTraverse()
{
   inorderTraverse(root);
} // end inorderTraverse

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

- In-order traversal:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```



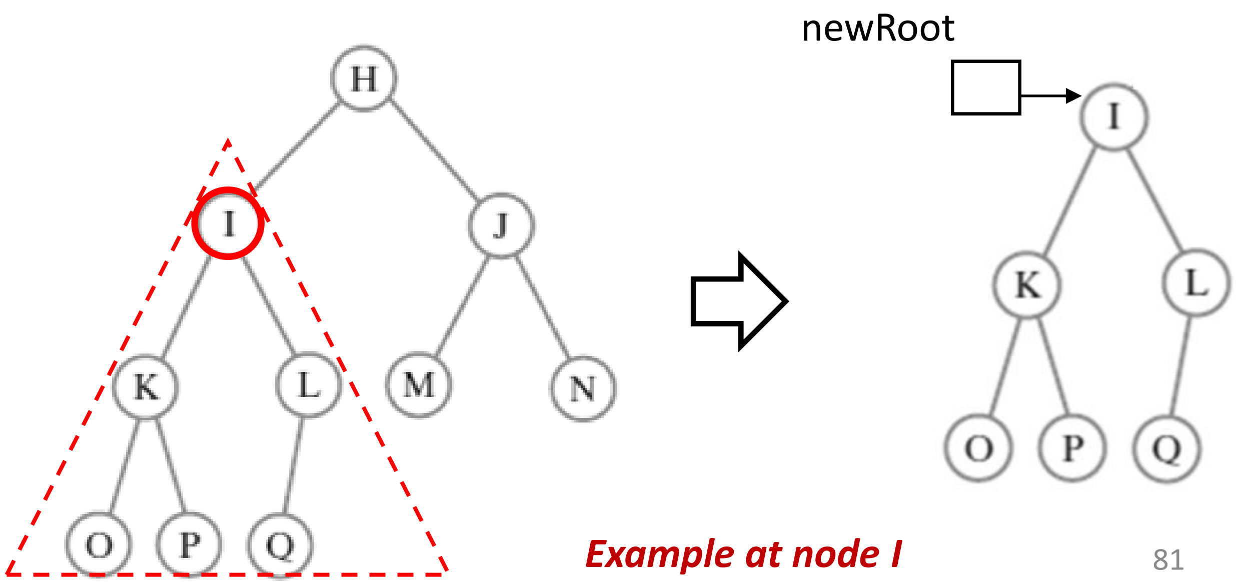
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



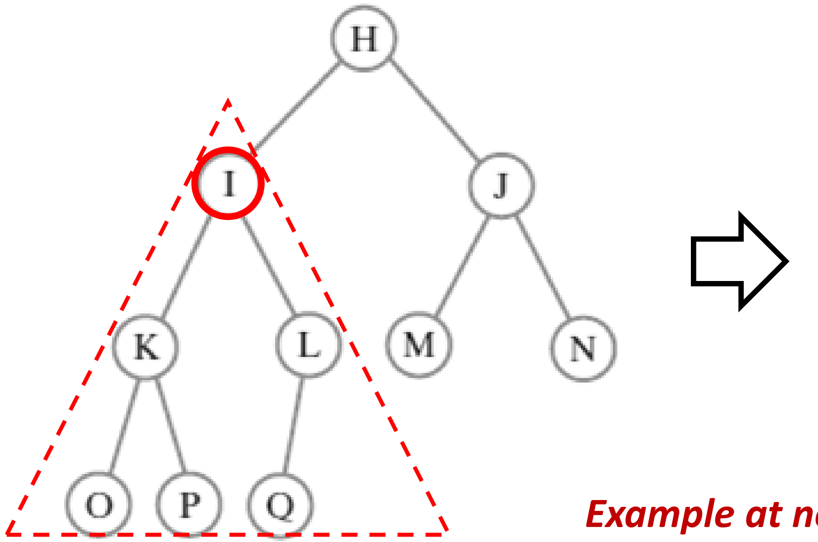
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



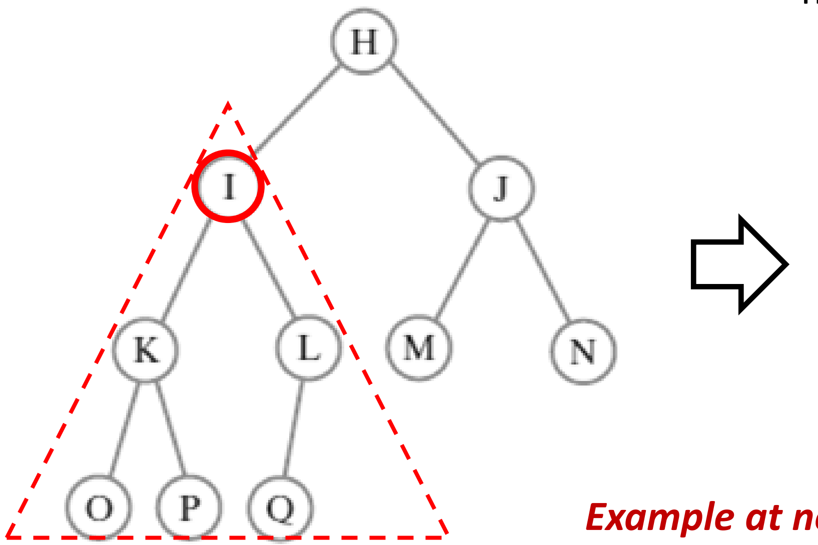
Example at node I

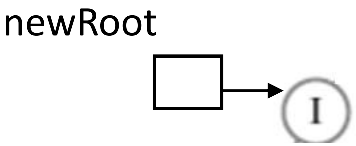
Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)





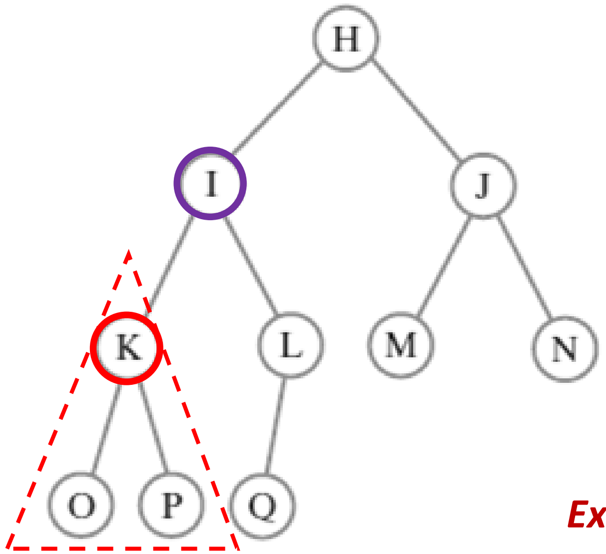
Example at node I

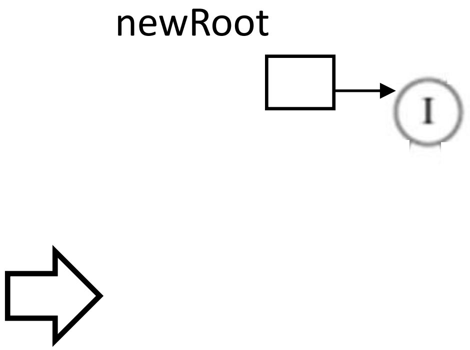
Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)





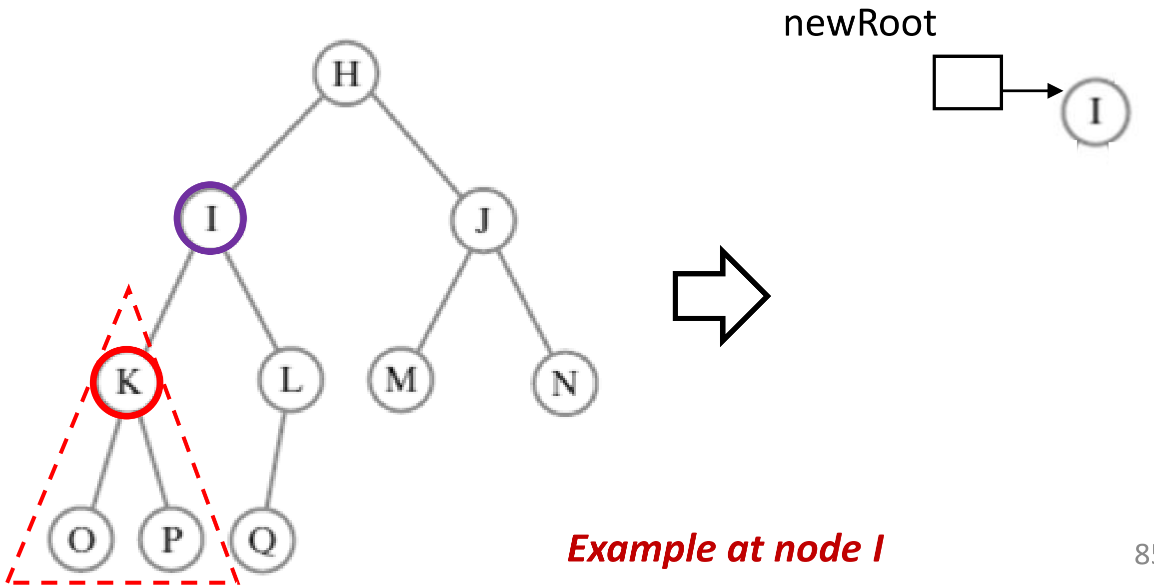
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



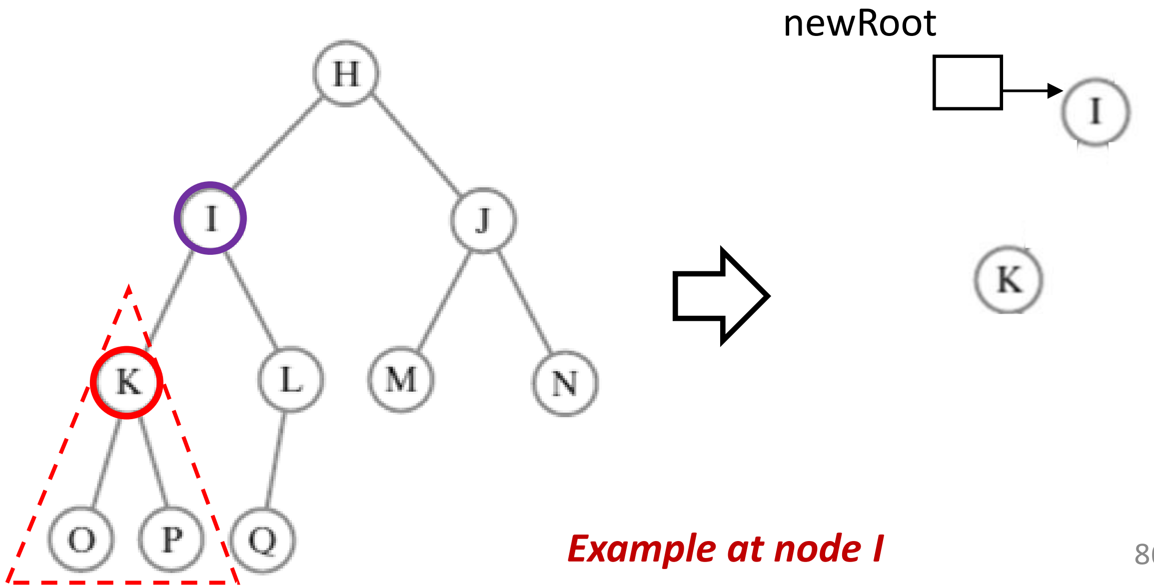
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



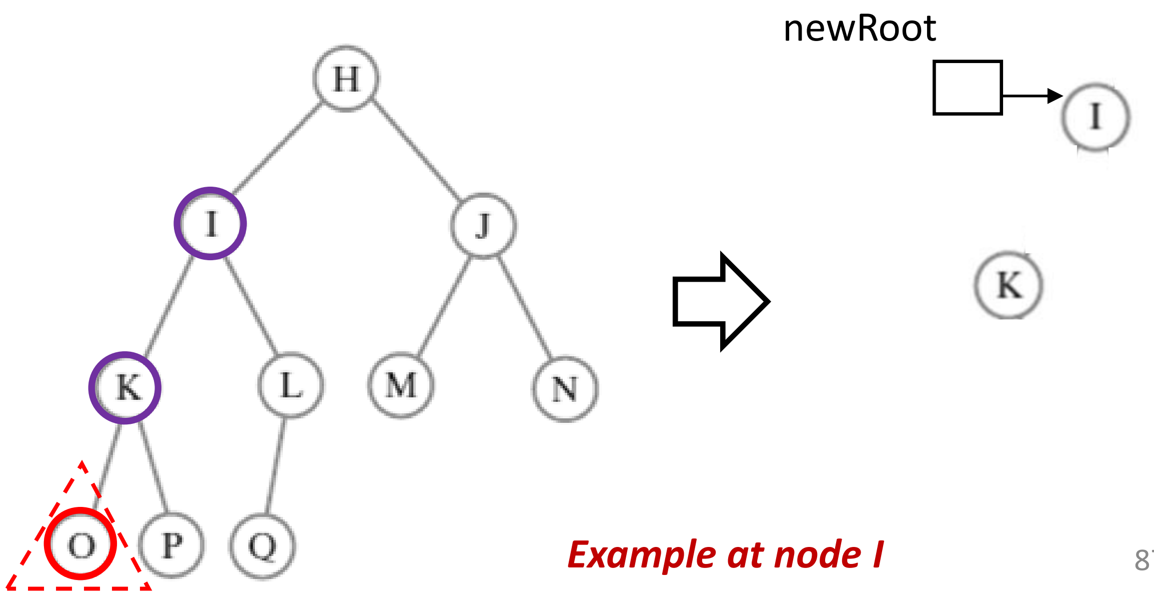
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



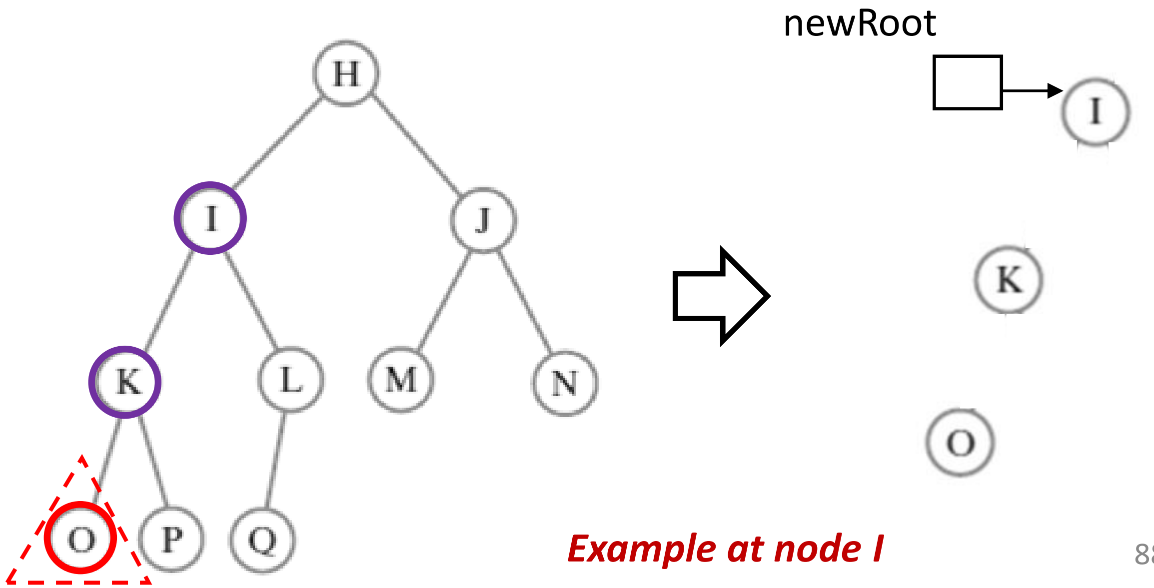
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



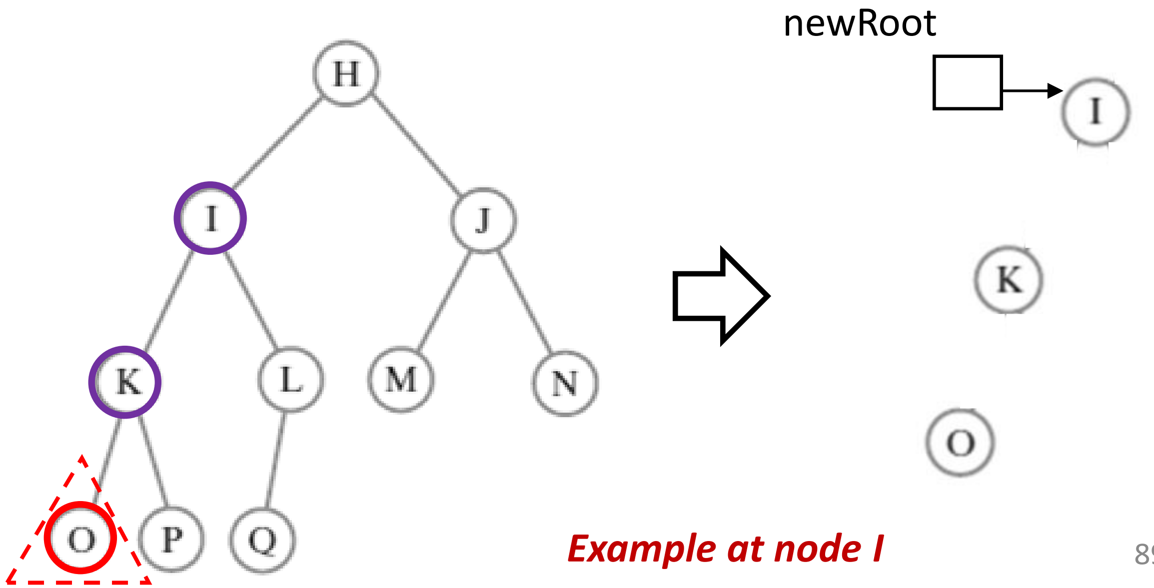
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



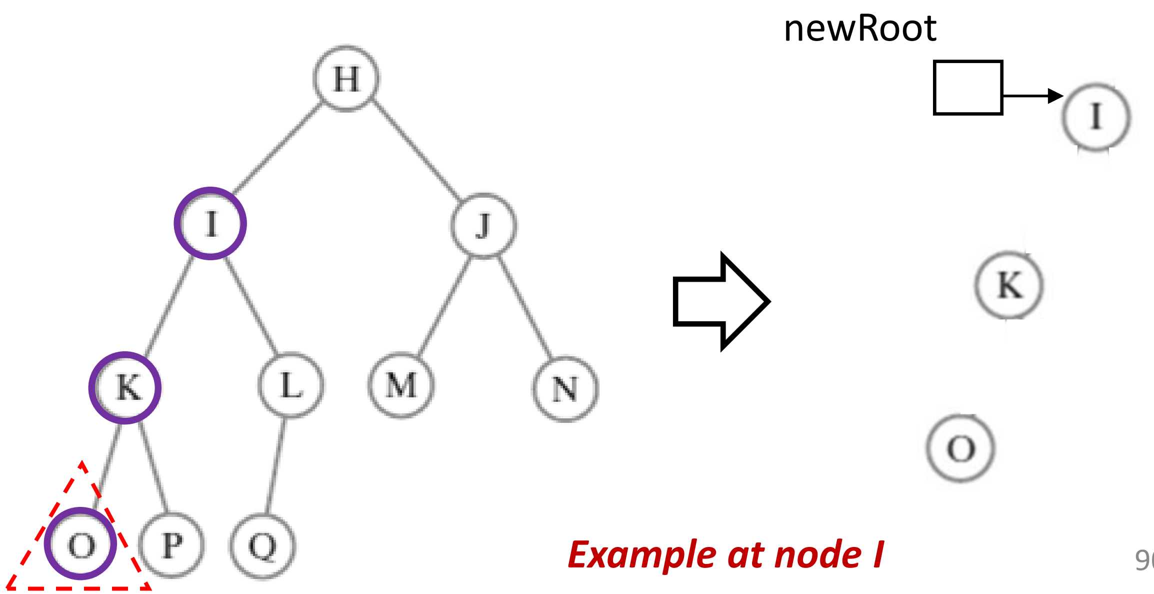
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



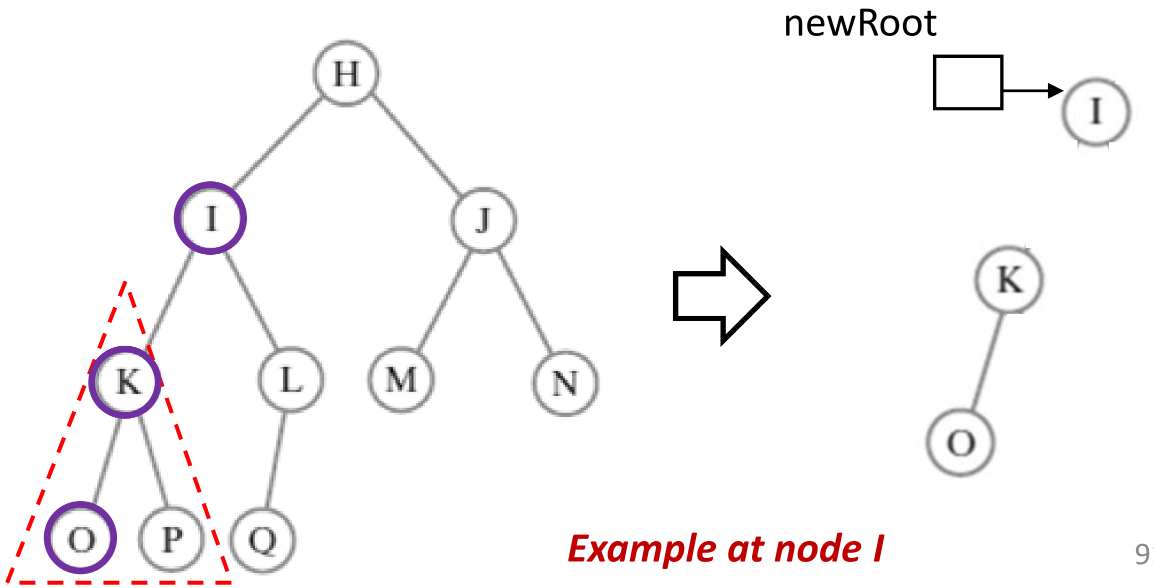
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



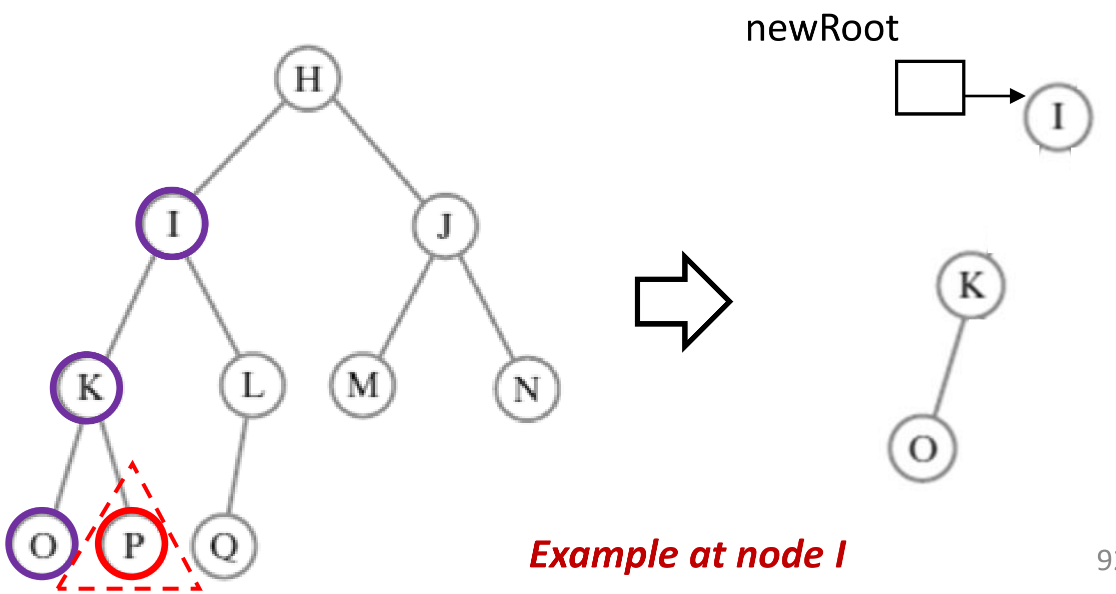
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



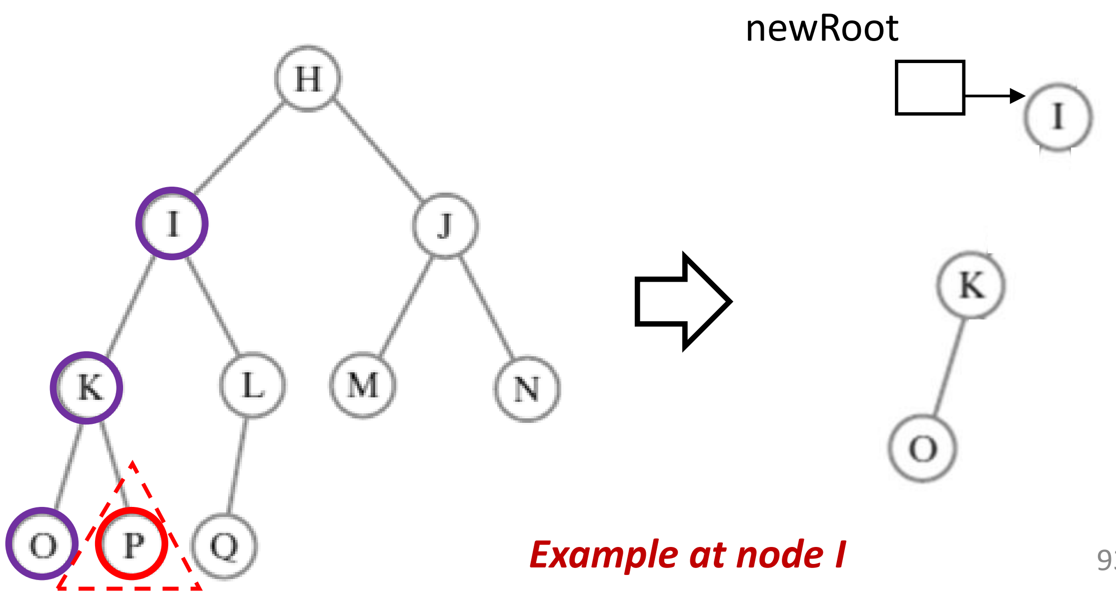
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



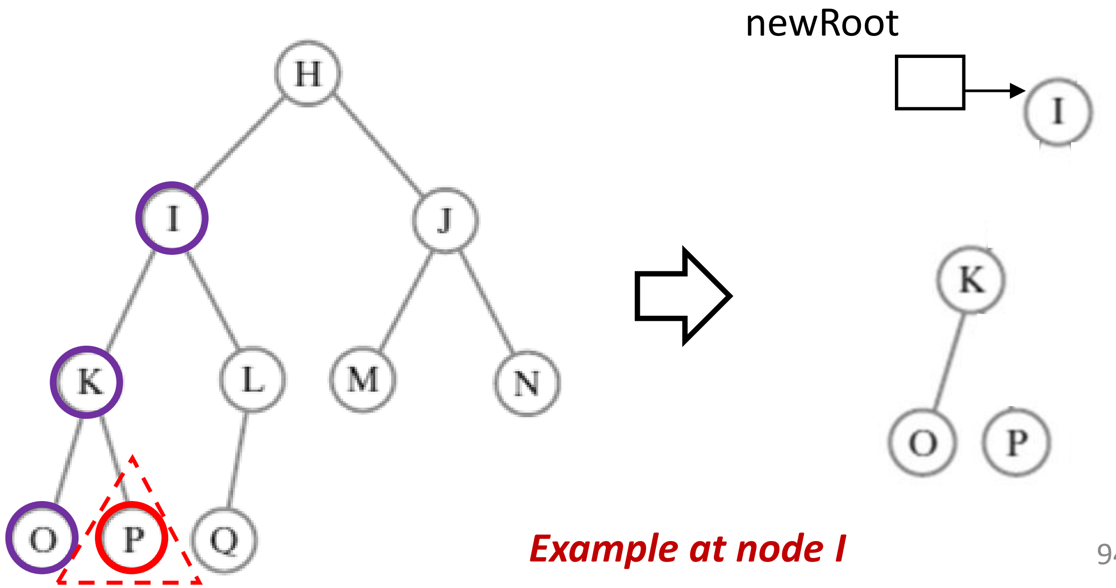
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



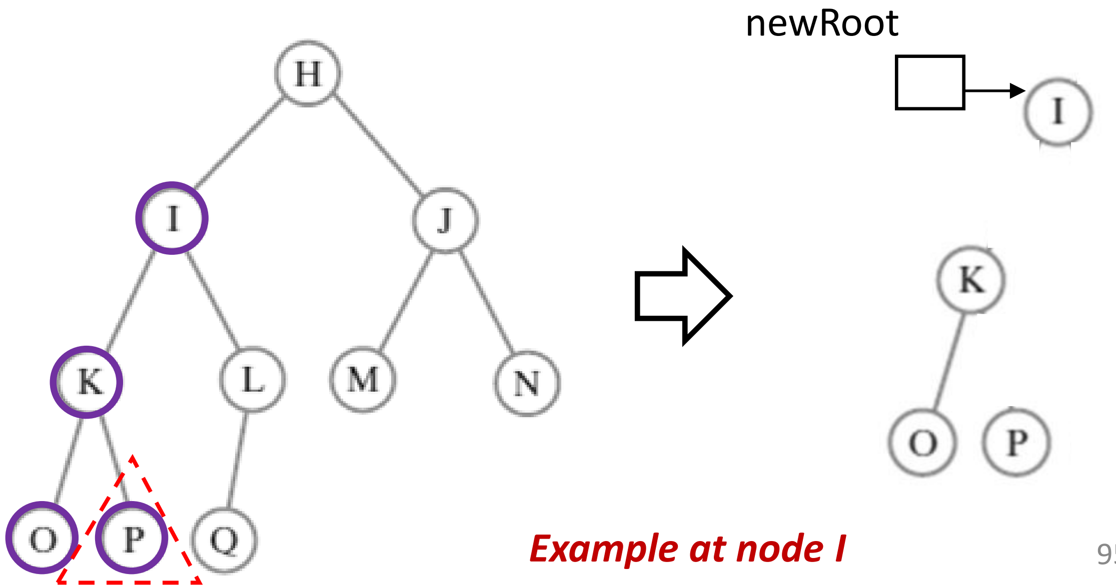
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



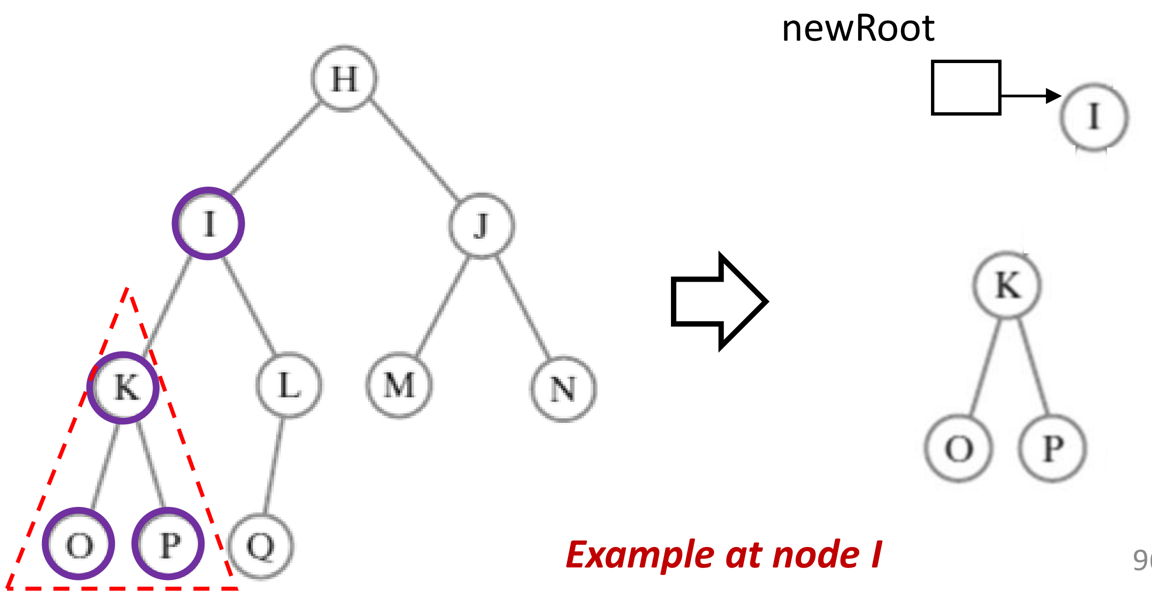
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



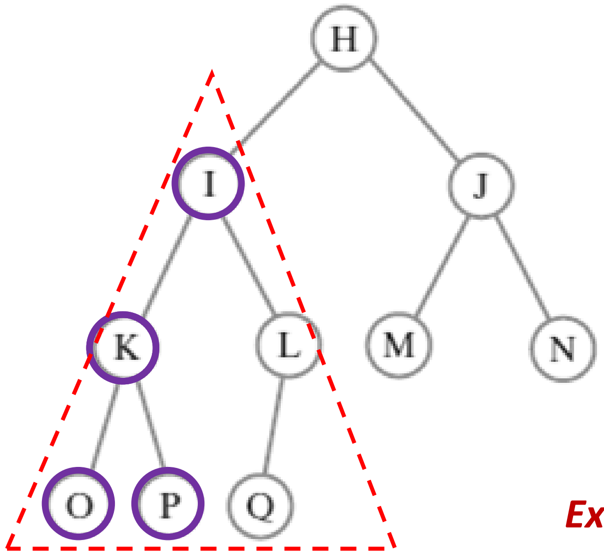
Example at node I

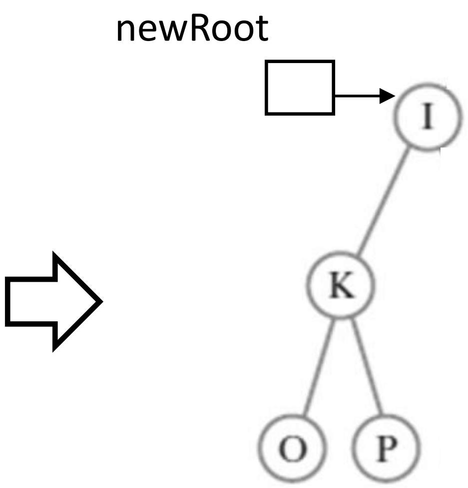
Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)





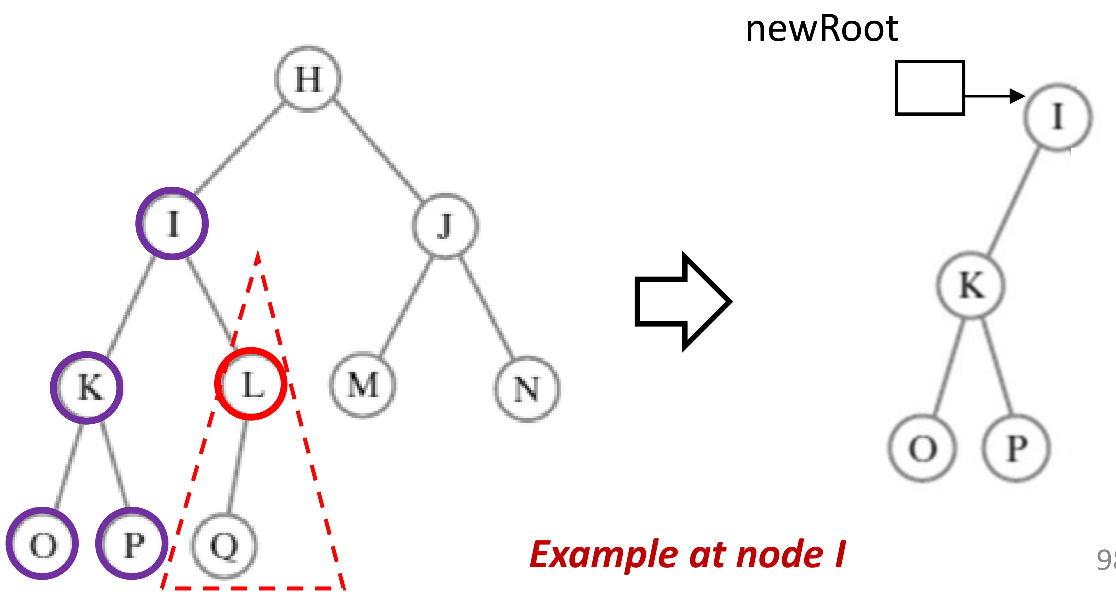
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)



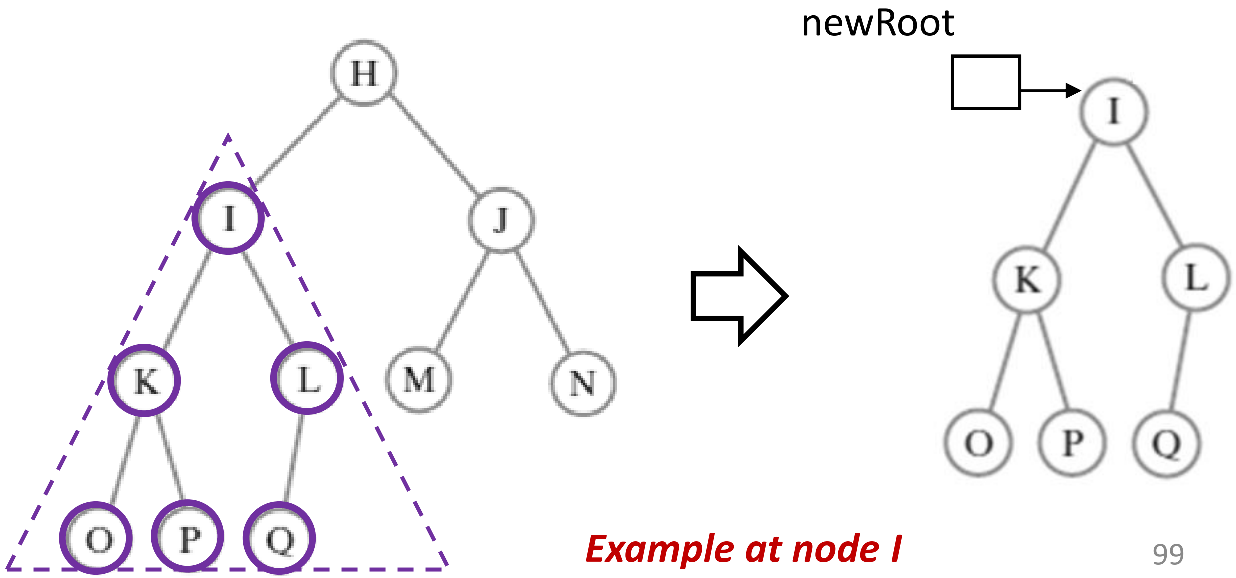
Example at node I

Method Copy in BinaryNode Class

- Generate copies the subtree rooted at a 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
```

This implementation uses **recursions**
(it performs a **pre-order traversal** of the subtree)

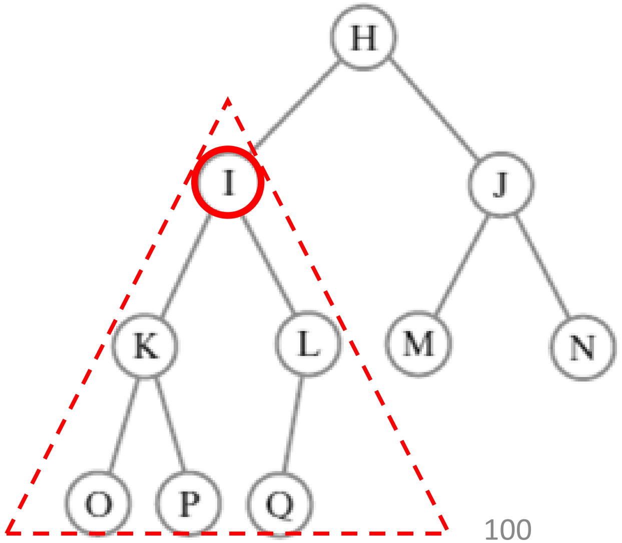


Example at node I

In-Class Exercises

Implementations of Tree-related methods really use a lot of "**recursions**".

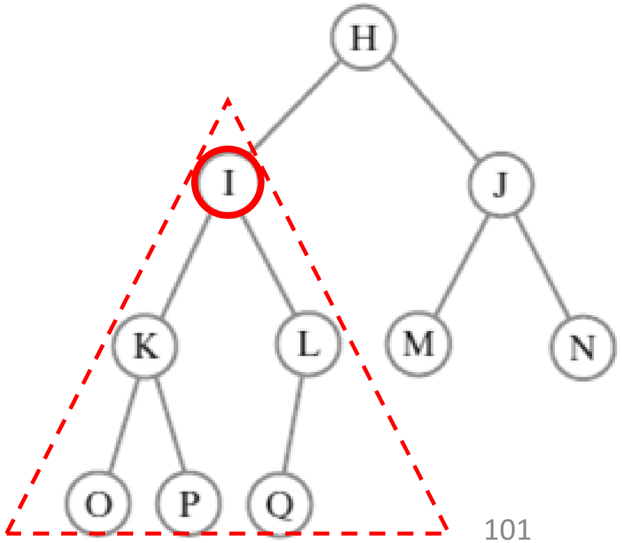
- getNumberOfNodes()
  - Returns the number of nodes in the subtree rooted at this node..
- getHeight()
  - Returns the height of the subtree rooted at this node.
  - Note: The height of a node is the number of nodes on the longest path between that node and a leaf + 1.
- getLeftmostData()
  - Returns the data from the leftmost node in the subtree rooted at this node
- getRightmostData()
  - Returns the data from the rightmost node in the subtree rooted at this node



In-Class Exercises

- getNumberOfNodes()
  - Returns the number of nodes in the subtree rooted at this node..

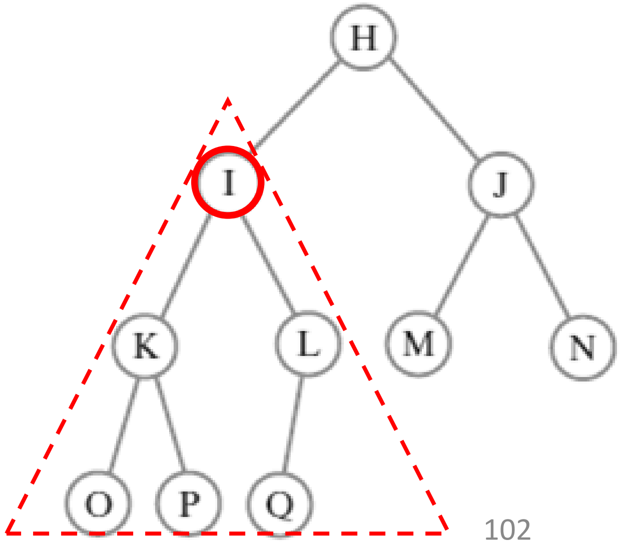
```
public int getNumberOfNodes()
 {
   int leftNumber = 0;
   int rightNumber = 0;

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

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

   return 1 + leftNumber + rightNumber;
 }
```



In-Class Exercises

- getHeight()
  - Returns the height of the subtree rooted at this node.

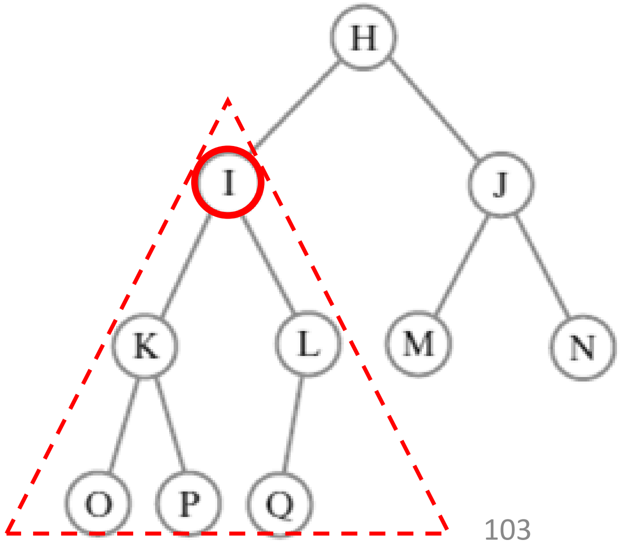
```
public int getHeight()
 {
   return getHeight(this); // Call private getHeight
 }

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

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

   return height;
 }
```



In-Class Exercises

- getLeftmostData()
  - Returns the data from the leftmost node in the subtree rooted at this node
- getRightmostData()
  - Returns the data from the rightmost node in the subtree rooted at this node

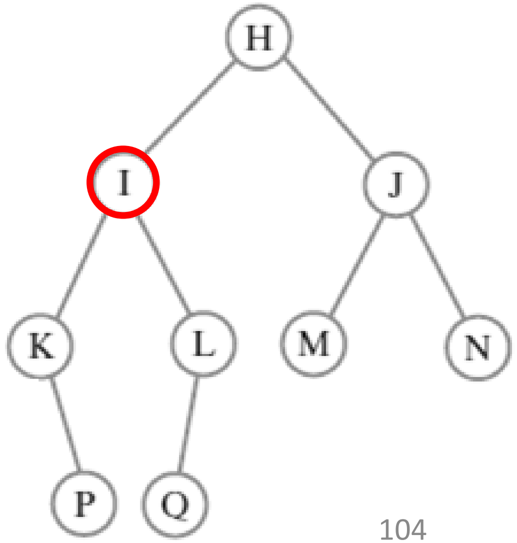
```
public T getLeftmostData( )
{
     if (leftChild == null)
         return data;
    else
         return leftChild.getLeftmostData( );
 }

public T getRightmostData( )
{
    if (rightChild == null)
                    return data;
    else
                    return rightChild.getRightmostData( );
}
```



Removing the Leftmost Node

- Remove the leftmost node of the subtree rooted at this node
- Postcondition: leftmost node removed.
  - (Note: the return value may be a reference to the root of the new tree. Return value could also be null or a reference to rightChild.)

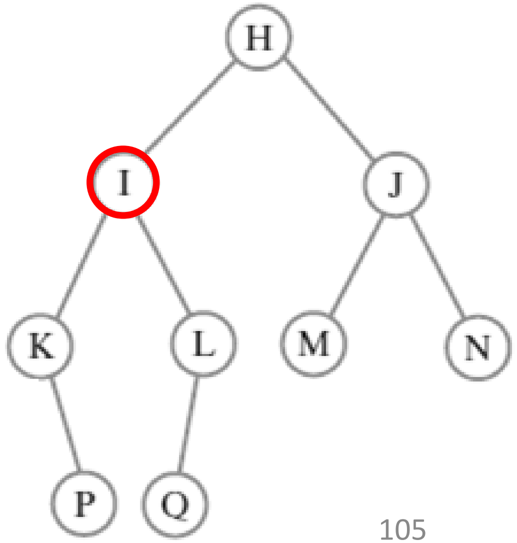
```
public BinaryNode<T> removeLeftmost( )
{
    if (leftChild == null) {
            return rightChild;
    }

    else {
            leftChild = leftChild.removeLeftmost( );
            return this;
    }
}
```



Removing the Leftmost Node

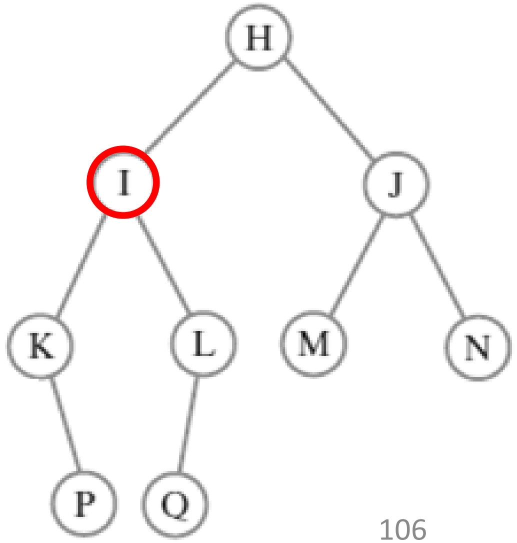
- Remove the leftmost node of the subtree rooted at this node
- Postcondition: leftmost node removed.
  - (Note: the return value may be a reference to the root of the new tree. Return value could also be null or a reference to rightChild.)

```
public BinaryNode<T> removeLeftmost( )
{
    if (leftChild == null) {
            return rightChild;        //Why we use rightChild here?
    }

    else {
            leftChild = leftChild.removeLeftmost( );
            return this;
    }
}
```



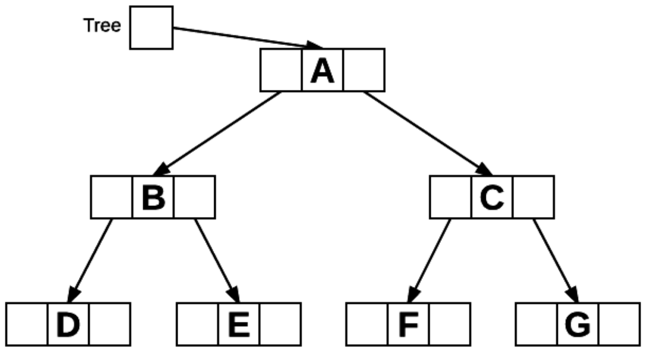
In-Class Exercise

- Remove the rightmost node of the subtree rooted at this node
- Postcondition: rightmost node removed.



Implementation of a Binary Tree

```
class BinaryTree<T>
{
  private BinaryNode<T> root;
  ...
  // Constructors
  // Get and Set methods
  // Boolean method
  //   isEmpty(): return True if an empty tree, otherwise, False)
  // Other methods
}
```



Constructing a Binary Tree

- An interface for a binary tree

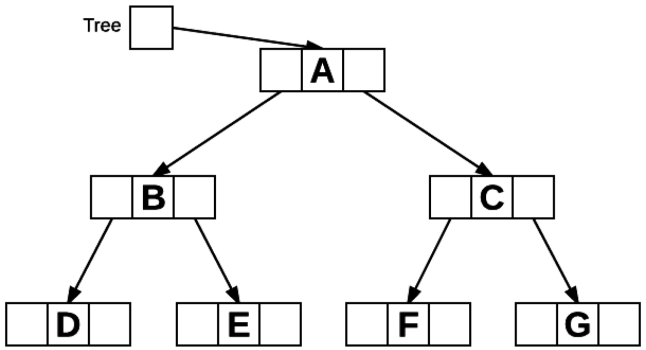
```
package TreePackage;
public interface BinaryTreeInterface<T> extends TreeInterface<T>,
                                                TreeIteratorInterface<T>
{
   /** Sets this binary tree to a new one-node binary tree.
       @param rootData  The object that is the data for the new tree's root.
   */
   public void setTree(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
```

```
package TreePackage;
import java.util.Iterator;
public interface TreeIteratorInterface<T>
{
   public Iterator<T> getPreorderIterator();
   public Iterator<T> getPostorderIterator();
   public Iterator<T> getInorderIterator();
   public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```

```
package TreePackage;
public interface TreeInterface<T>
{
   public T getRootData();
   public int getHeight();
   public int getNumberOfNodes();
   public boolean isEmpty();
   public void clear();
} // end TreeInterface
```

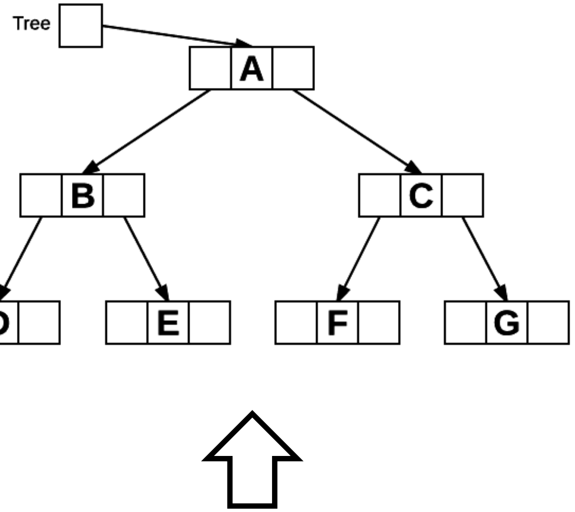
Constructors in BinaryTree Class

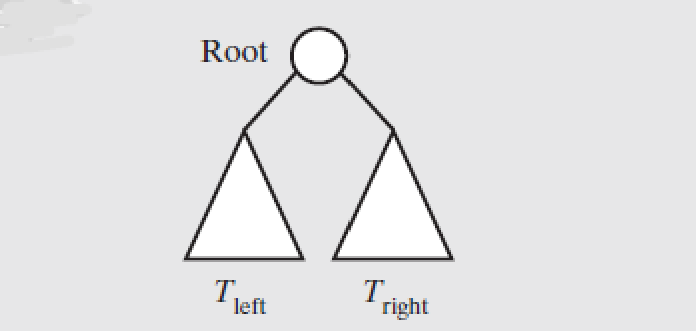
```java
package TreePackage;
import java.util.Iterator;
import java.util.NoSuchElementException;
import StackAndQueuePackage.*; // Needed by tree iterators
/**
    A class that implements the ADT binary tree.
    @author Frank M. Carrano.
*/
public class BinaryTree<T> implements BinaryTreeInterface<T>
{
   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
```



Constructors in BinaryTree Class

```java
package TreePackage;
import java.util.Iterator;
import java.util.NoSuchElementException;
import StackAndQueuePackage.*; // Needed by tree iterators
/**
    A class that implements the ADT binary tree.
    @author Frank M. Carrano.
*/
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
```

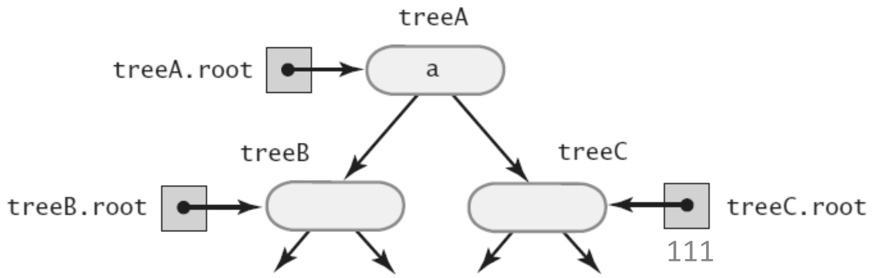




where T_{left} and T_{right} are binary trees.

Method: privateSetTree

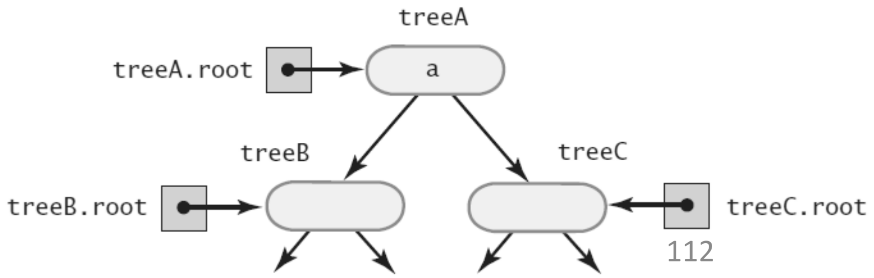
```
private void privateSetTree(T rootData, BinaryTree<T> leftTree,
                                        BinaryTree<T> rightTree)
{
   root = new BinaryNode<T>(rootData);

   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
```



Method: privateSetTree

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

// create a root node r constraining the given data

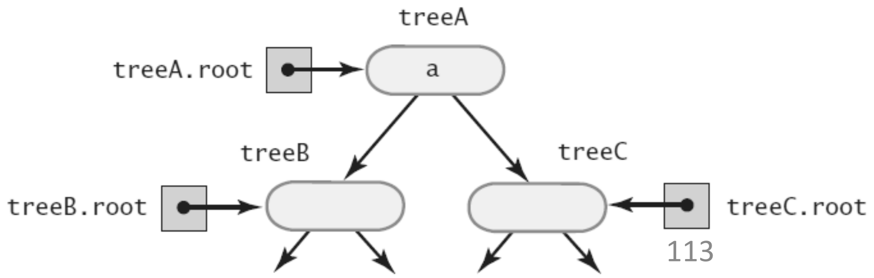


Method: privateSetTree

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

// create a root node r constraining the given data

// If left subtree exists and not empty, attach root node to r as left child.



Method: privateSetTree

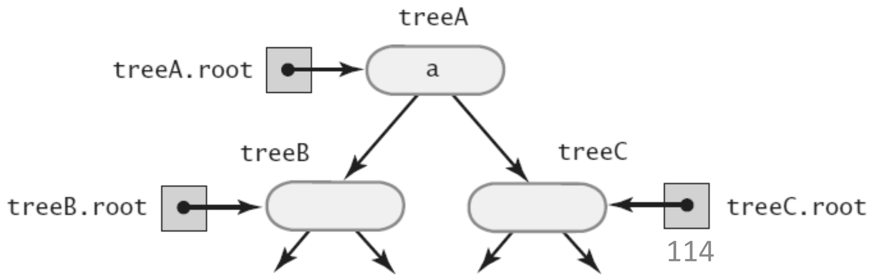
```java
private void privateSetTree(T rootData, BinaryTree<T> leftTree,
                                        BinaryTree<T> rightTree)
{
   root = new BinaryNode<T>(rootData);

   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
```

// create a root node r constraining the given data

// If left subtree exists and not empty, attach root node to r as left child.

// 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.



Method: privateSetTree

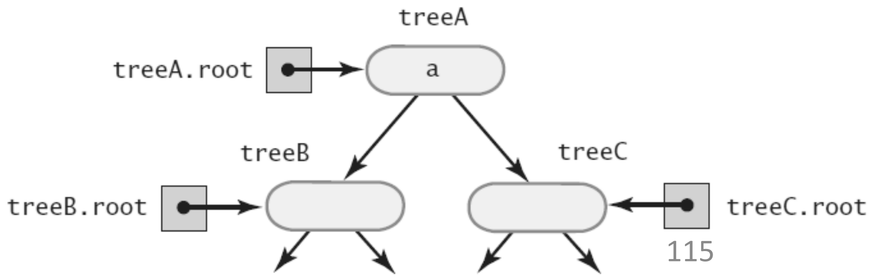
```
private void privateSetTree(T rootData, BinaryTree<T> leftTree,
                                        BinaryTree<T> rightTree)
{
   root = new BinaryNode<T>(rootData);

   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
```

// create a root node r constraining the given data

// If left subtree exists and not empty, attach root node to r as left child.

// 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.



Method: privateSetTree

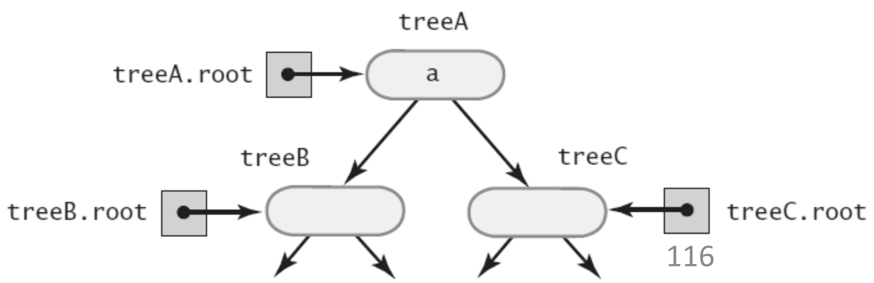
```
private void privateSetTree(T rootData, BinaryTree<T> leftTree,
                                        BinaryTree<T> rightTree)
{
   root = new BinaryNode<T>(rootData);
   if ((leftTree != null) && !leftTree.isEmpty())
      root.setLeftChild(leftTree.root);
   if ((rightTree != null) && !rightTree.isEmpty())
   {
      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
```

// create a root node r constraining the given data

// If left subtree exists and not empty, attach root node to r as left child.

// 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 a subtree (leftTree or rightTree) exists and differs from the tree object used to call privateSetTree, set the subtree's data field root to null



Method: privateSetTree

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

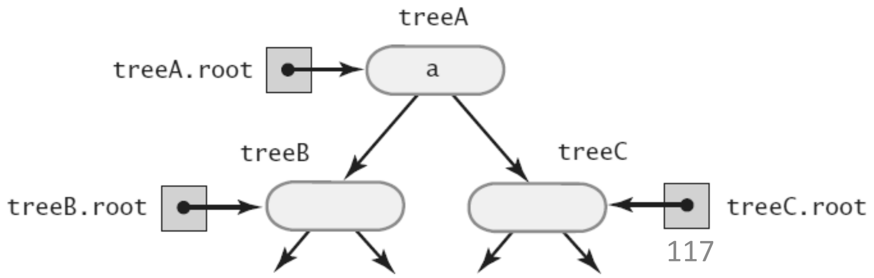
// create a root node r constraining the given data

// If left subtree exists and not empty, attach root node to r as left child.

// 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 a subtree (leftTree or rightTree) exists and differs from the tree object used to call privateSetTree, set the subtree's data field root to null



Constructing a Binary Tree

- An interface for a binary tree

```java
package TreePackage;
public interface BinaryTreeInterface<T> extends TreeInterface<T>,
                                                TreeIteratorInterface<T>
{
   /** Sets this binary tree to a new one-node binary tree.
       @param rootData  The object that is the data for the new tree's root.
   */
   public void setTree(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
```

```java
package TreePackage;
import java.util.Iterator;
public interface TreeIteratorInterface<T>
{
   public Iterator<T> getPreorderIterator();
   public Iterator<T> getPostorderIterator();
   public Iterator<T> getInorderIterator();
   public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```

```java
package TreePackage;
public interface TreeInterface<T>
{
   public T getRootData();
   public int getHeight();
   public int getNumberOfNodes();
   public boolean isEmpty();
   public void clear();
} // end TreeInterface
```

Summary

- Tree Terminology
- Tree Traversals
- Tree Representations
- Implementation of Binary Nodes

What I Want You to Do

- Review class slides
- Review chapters 24 and 25.

- Next Class
  - Trees (II)