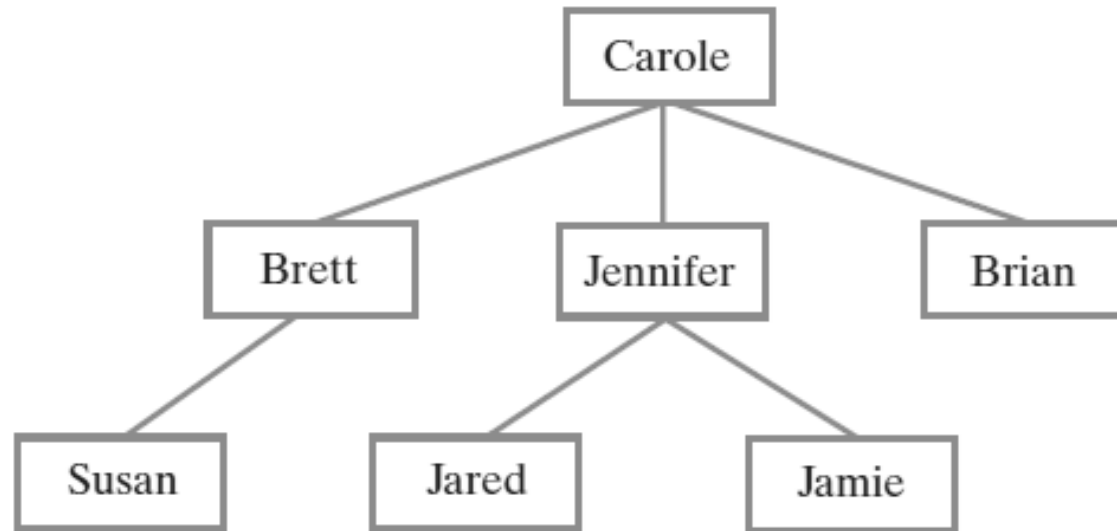


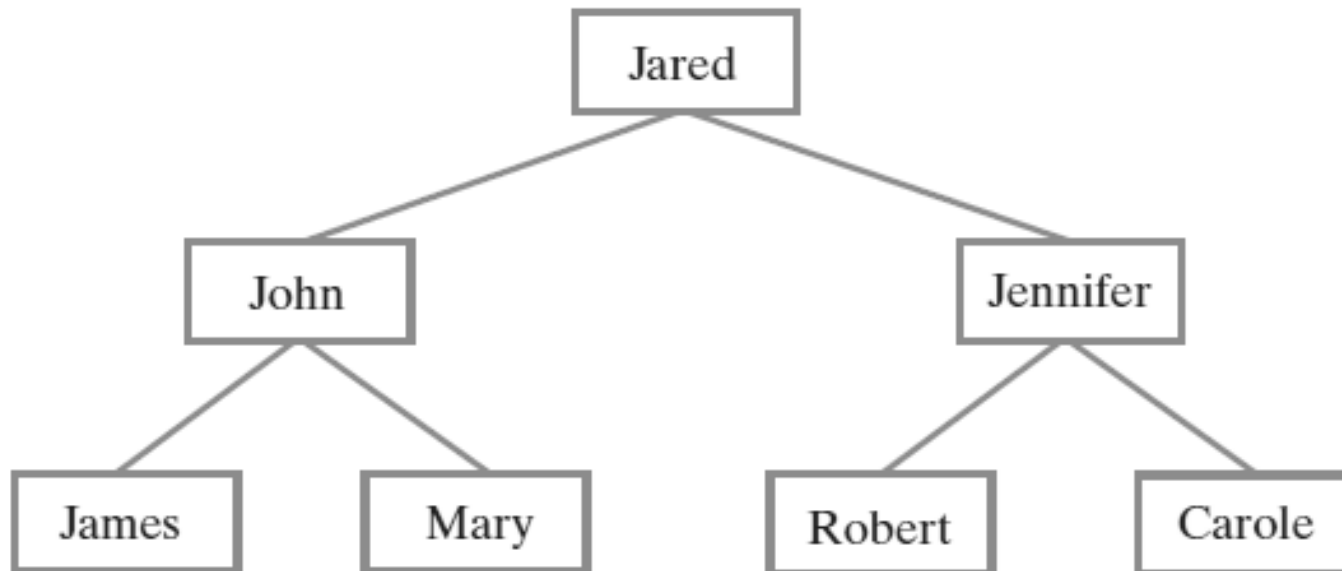
# Trees

# Intro

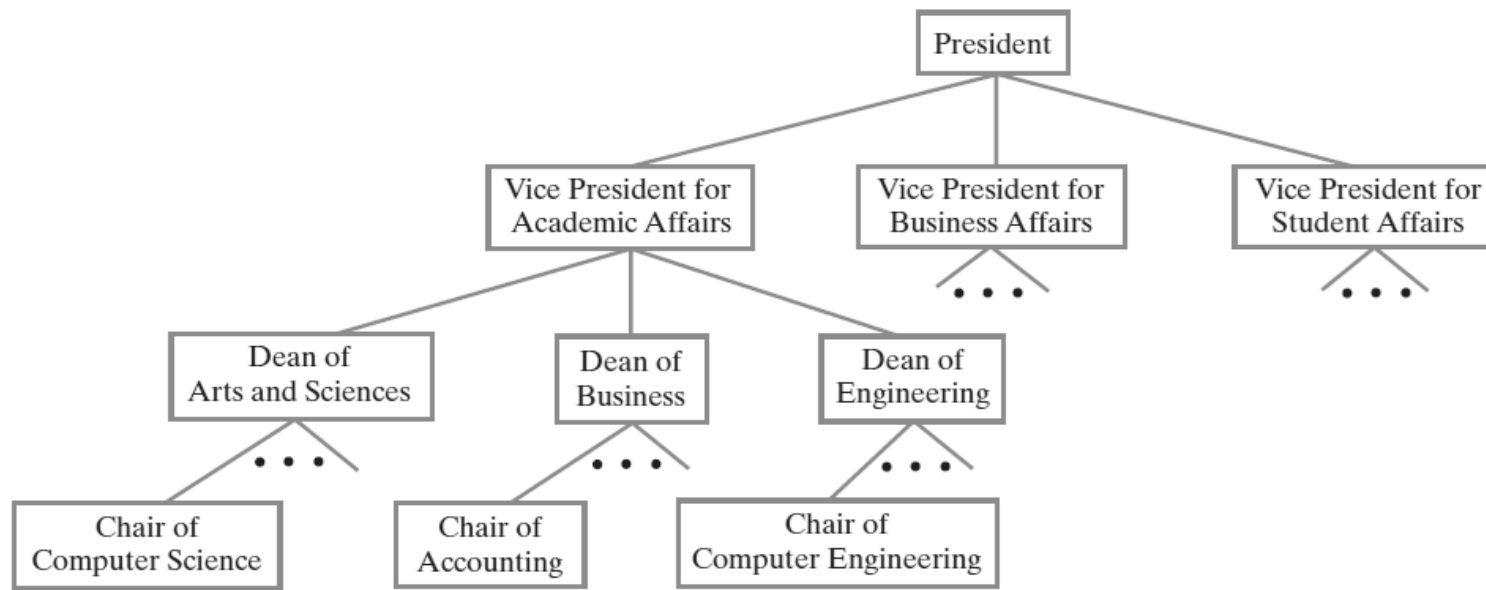
- As a plant, a **tree** is well known.
- Data organization that we have seen so far have placed data in a linear order.
  - For example, objects in a stack, queue, list, or dictionary appear one after the other.
- As useful as these organizations are, you often must categorize data into groups and subgroups. Such a classification is **hierarchical**, or **nonlinear**, since the data items appear at various levels within the organization.



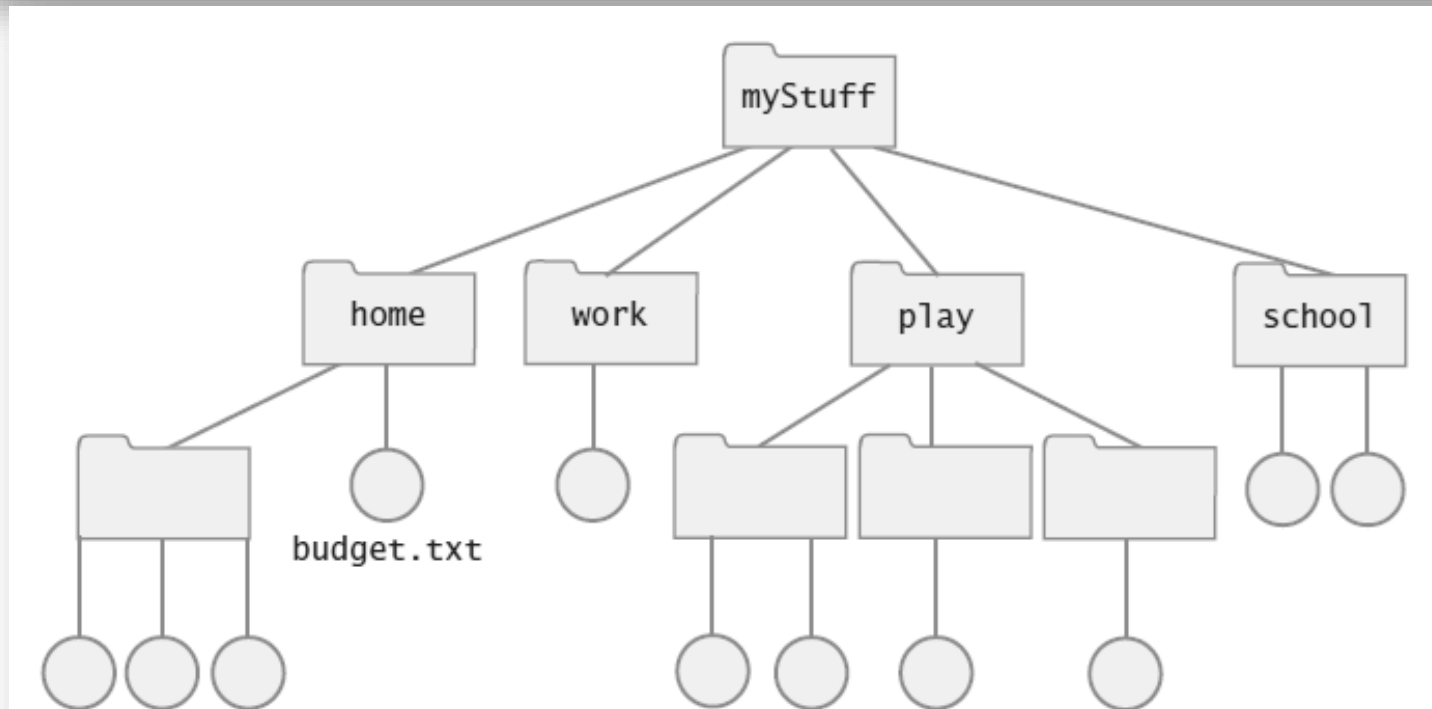
Family tree: Carole's children and grandchildren.



Family tree: Jared's parents and grandparents.



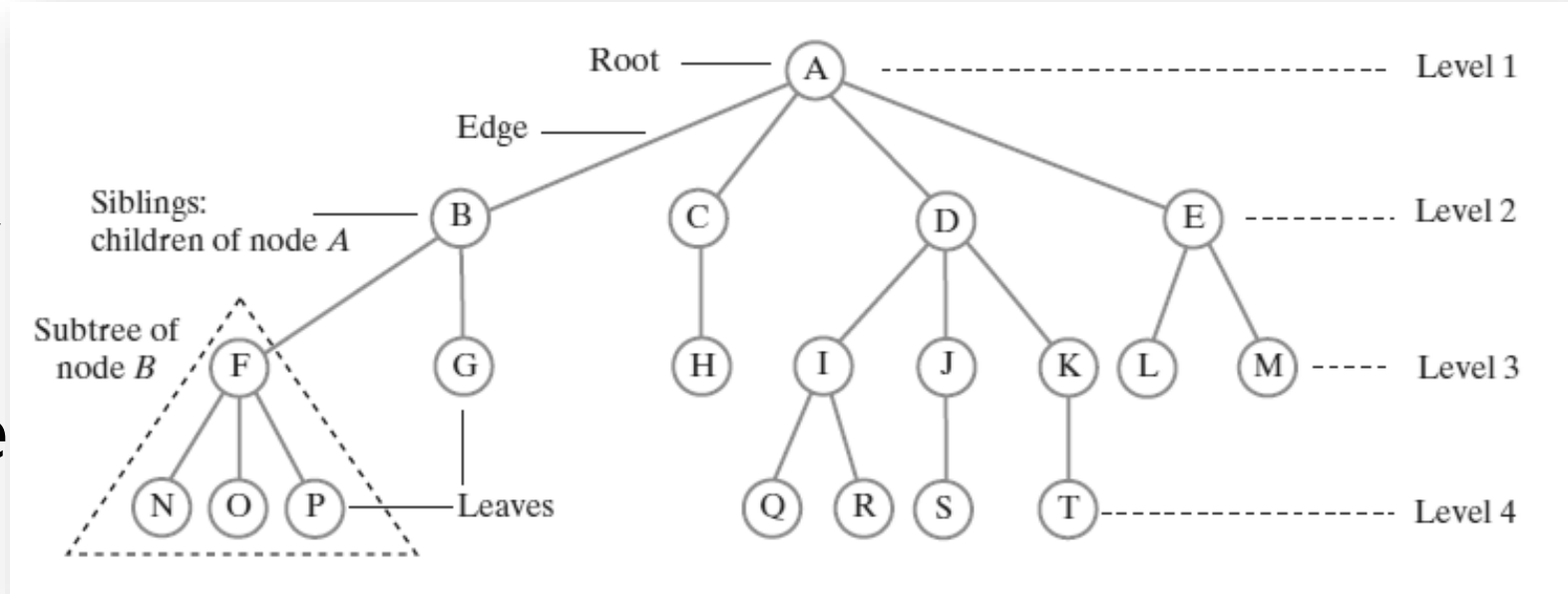
A portion of university's administrative structure.



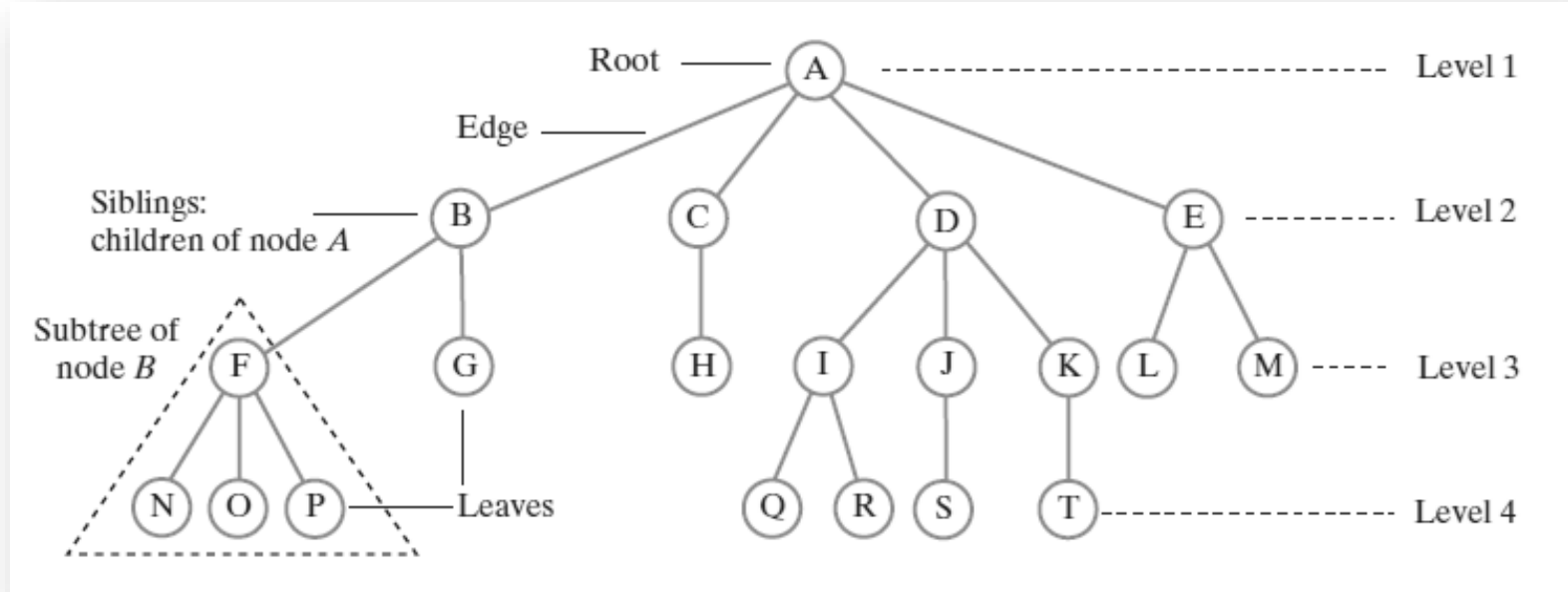
Computer files organized into folders.

# Tree Terminology

- At the top level is a single node called the **root**.
- A **tree** is a set of **nodes** connected by **edges** that indicate the relationships among the nodes.
- The nodes are arranged in **levels** that indicate the nodes' hierarchy.
- The nodes at each successive level of a tree are the **children** of the nodes at the previous level.
- A node that has children is the **parent** of those children.
- For example, node A is the parent of nodes B, C, D, and E. Since these children have the same parent, they are called **siblings**.

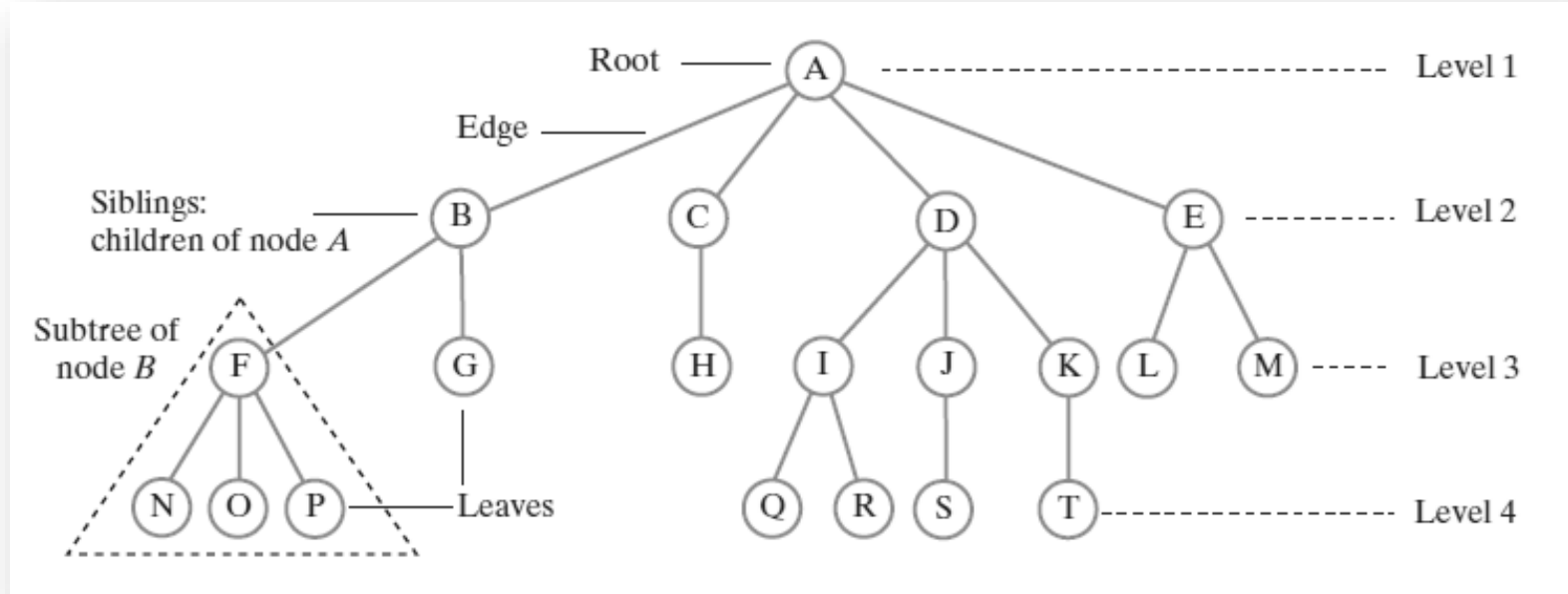


# Tree Terminology (cont.)



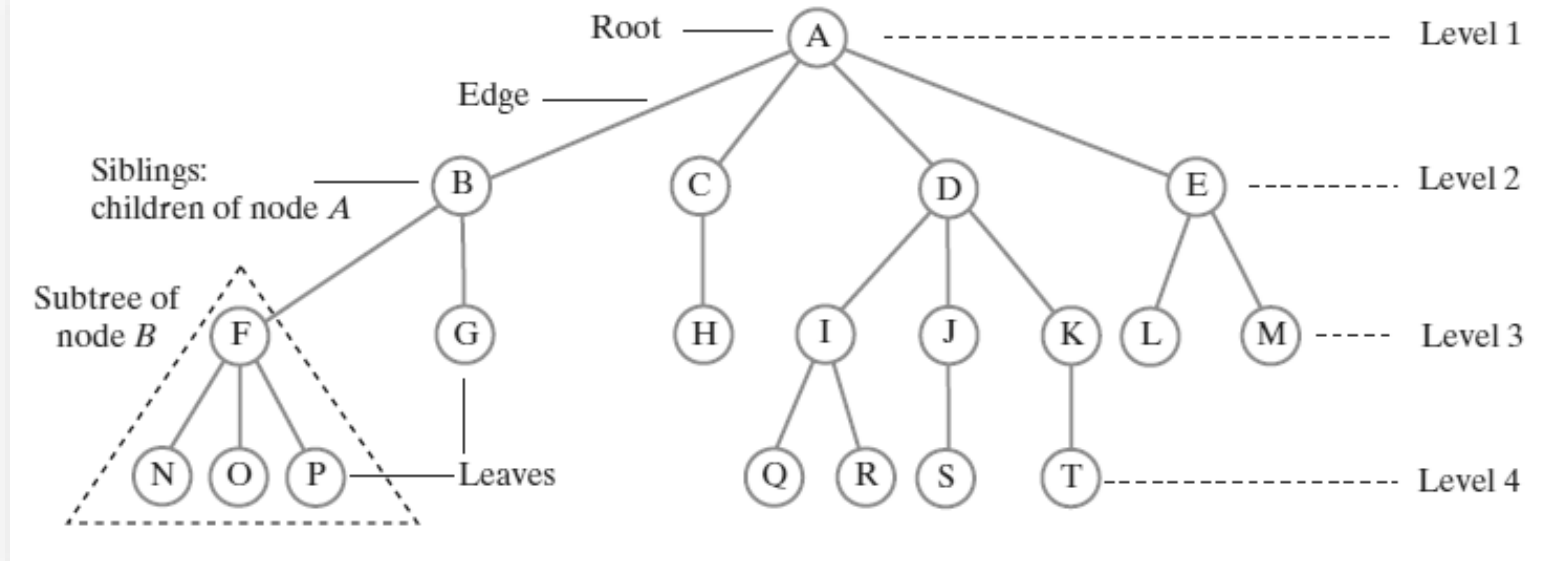
- Nodes B, C, D, and E are also **descendants** of node A, and node A is their **ancestor**.
  - Furthermore, node P is a descendant of A, and A is an ancestor of P. Notice that P has no children.
- A node that has no children is called a **leaf**, for example, a node P above.
- A node that is not a leaf – that is, one that has children – is called either an **interior node** or a **nonleaf**.

# Tree Terminology (cont.)



- The **root** is the only node that has no parent; all other nodes have one parent each. The root is the origin of a hierarchical organization.
- A tree can be empty.
- Any node and its descendants form a **subtree** of the original tree.
  - A **subtree of a node** is a tree rooted at a child of that node. For example, one subtree of node B is the tree rooted at F.
- A **subtree of a tree** is a subtree of the tree's root.

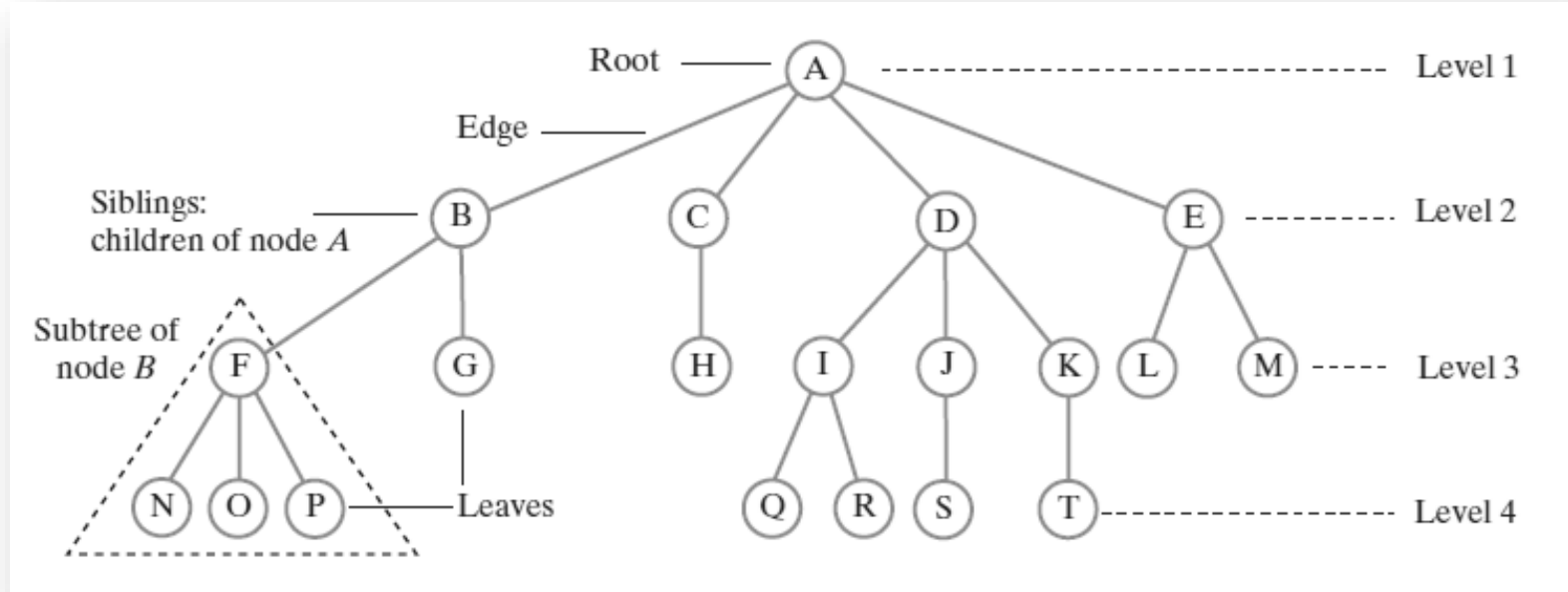
# Tree Terminology (cont.)



- In general, each node in a tree can have an arbitrary number of children. We sometimes call such a tree a **general tree**.
- If each node has no more than  $n$  children, the tree is called an  **$n$ -ary tree**.
- If each node has at most two children, the tree is called a **binary tree**.

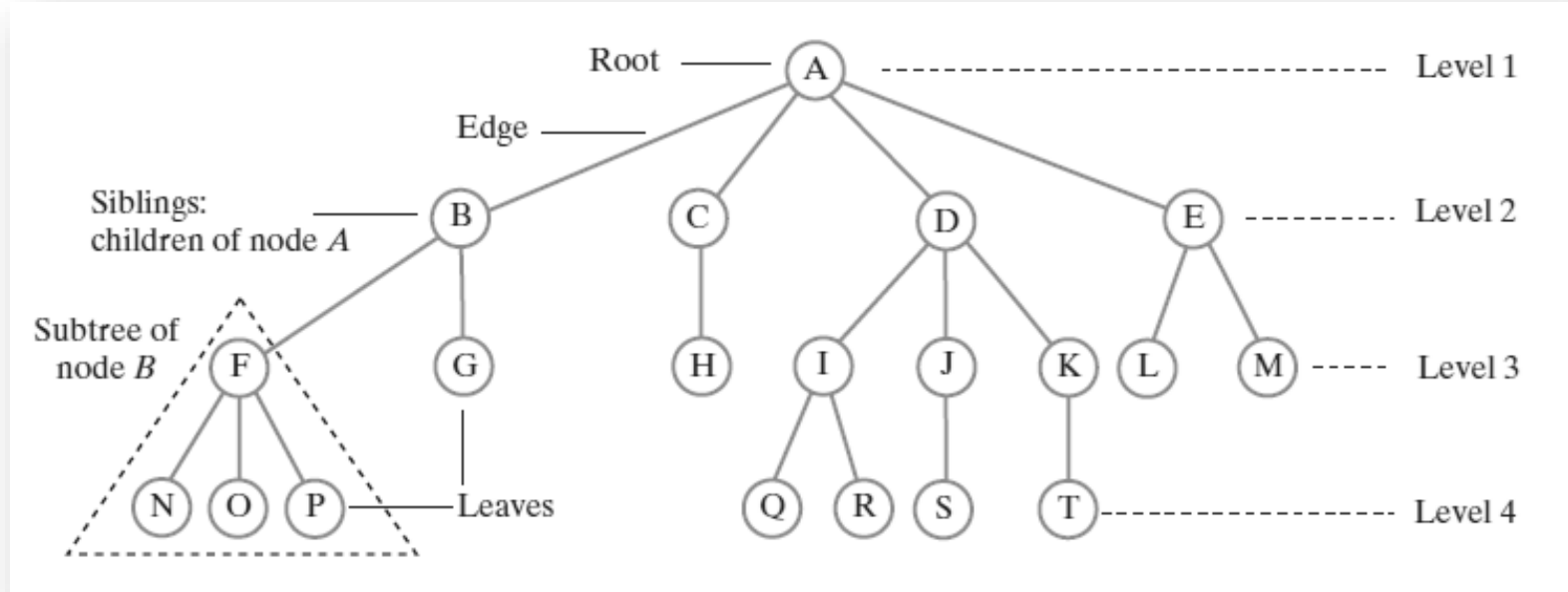


# Tree Terminology (cont.)



- The **height** of a tree is the number of levels in the tree.
  - The tree in above figure has 4 levels, and so its height is 4.
  - We number the levels in a tree beginning at level 1. The height of a one-node tree is 1, and the height of an empty tree is 0.
- Recursively: ***Height of tree  $T = 1 + \text{height of the tallest subtree of } T$*** .
  - The root of the tree in above figure has four subtrees of heights 3, 2, 3, and 2. Since the tallest of these subtrees has height 3, the tree height is 4.

# Tree Terminology (cont.)

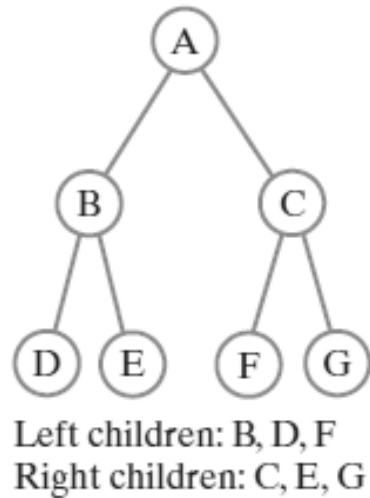


- We can reach any node in a tree by following a **path** that begins at the root and goes from node to node along the edges that join them.
  - The path between the root and any other node is unique.
- The **length of a path** is the number of edges that compose it.
  - In above figure the path that passes through the nodes A, B, F, and N has length 3. No other path from the root to a leaf is longer than this particular path.
  - The tree has height 4, which is 1 more than the length of this longest path.

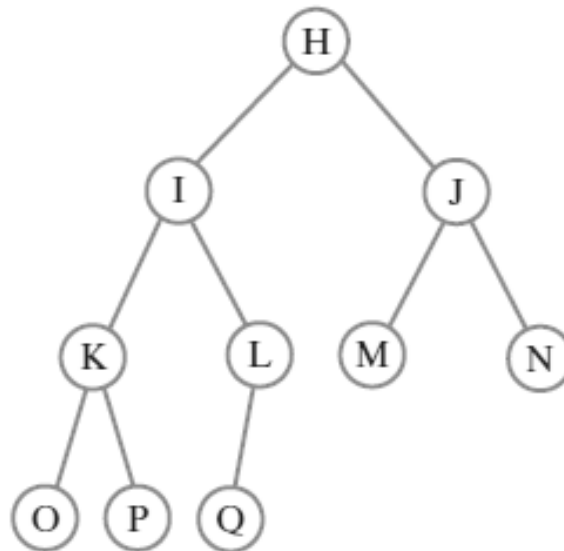
# Binary Trees

- Each node in a **binary tree** has at most two children. They are called **left child** and the **right child**.

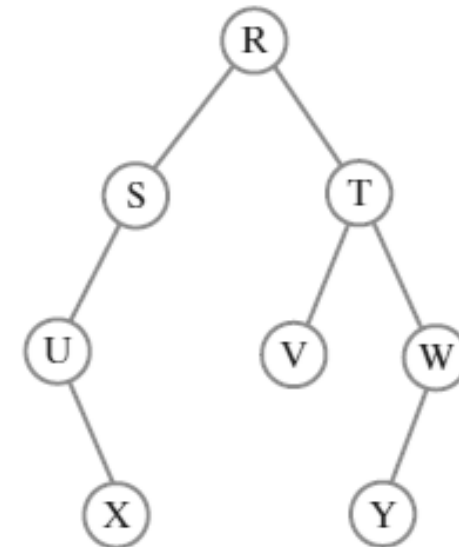
(a) Full tree



(b) Complete tree

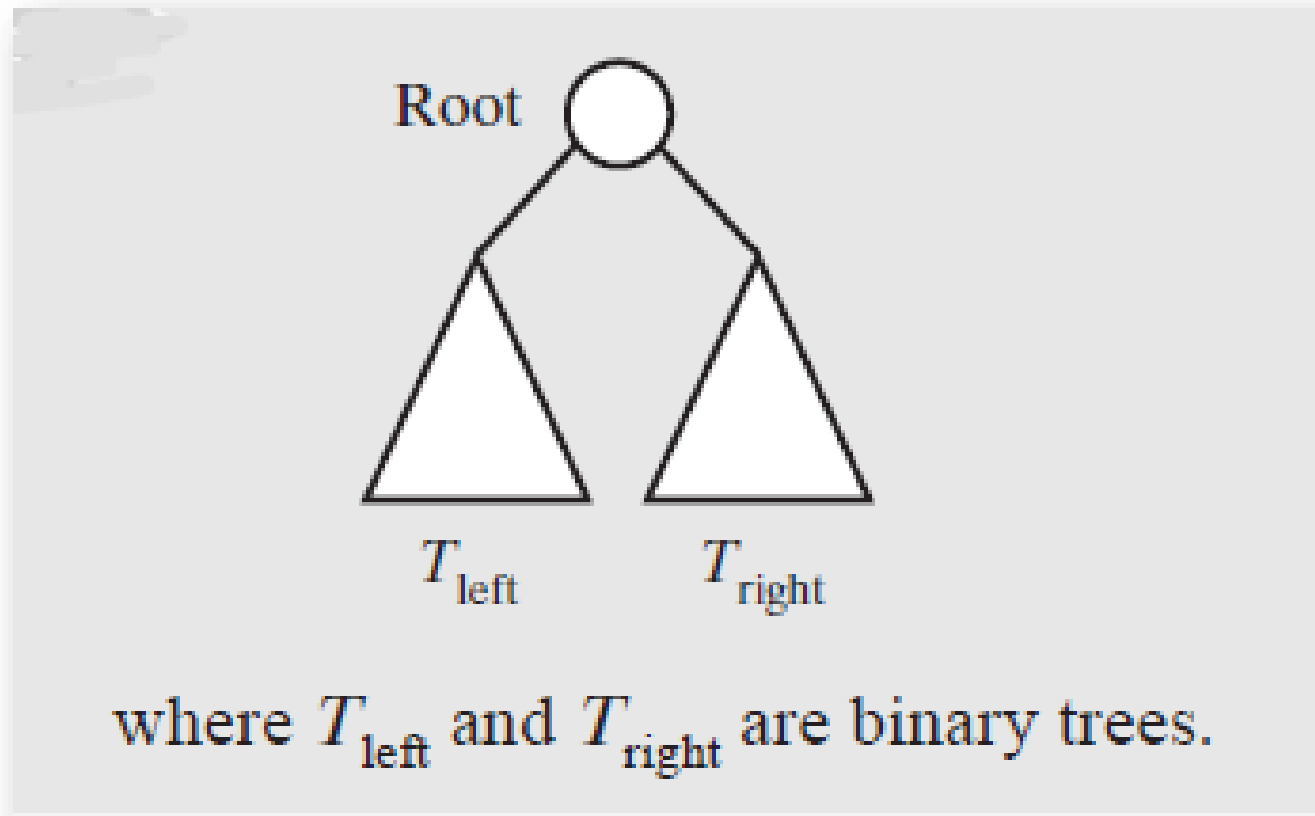


(c) Tree that is not full and not complete



# Binary Trees (cont.)

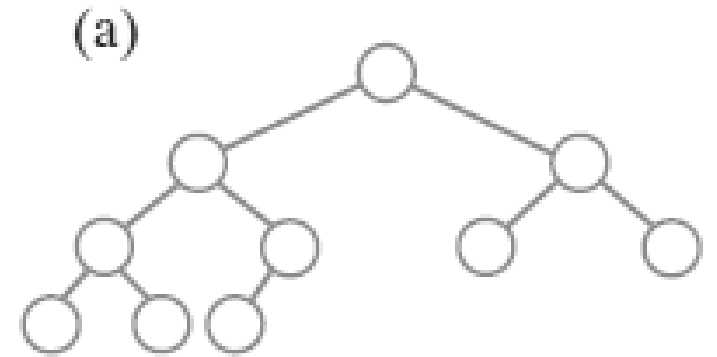
- A **binary tree** is either empty or has the following form:



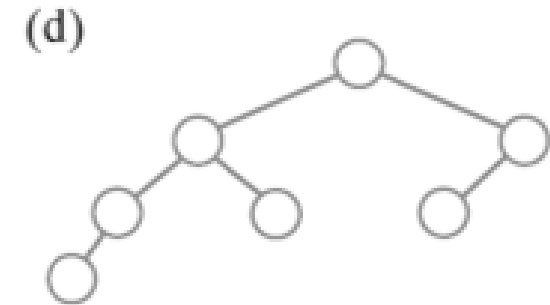
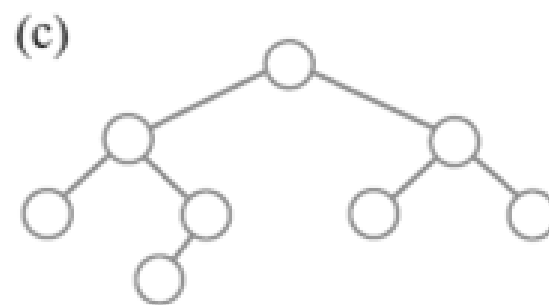
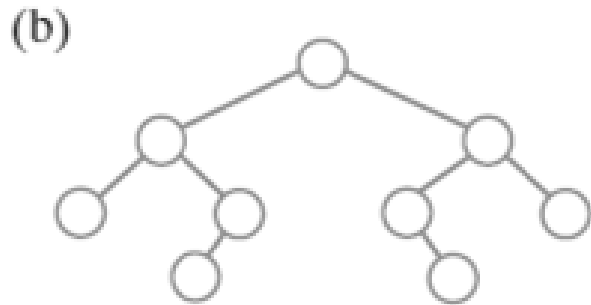
# Balanced Binary Trees

- *When each node in a binary tree* has two subtrees whose heights are exactly the same, the tree is said to be **completely balanced**.
  - The only completely balanced binary trees are full.
- Other trees said to be **height balanced**, or simply **balanced**, if the subtrees of each node in the tree differ in height no more than 1.

# Balanced Binary Trees (cont.)






Balanced and complete

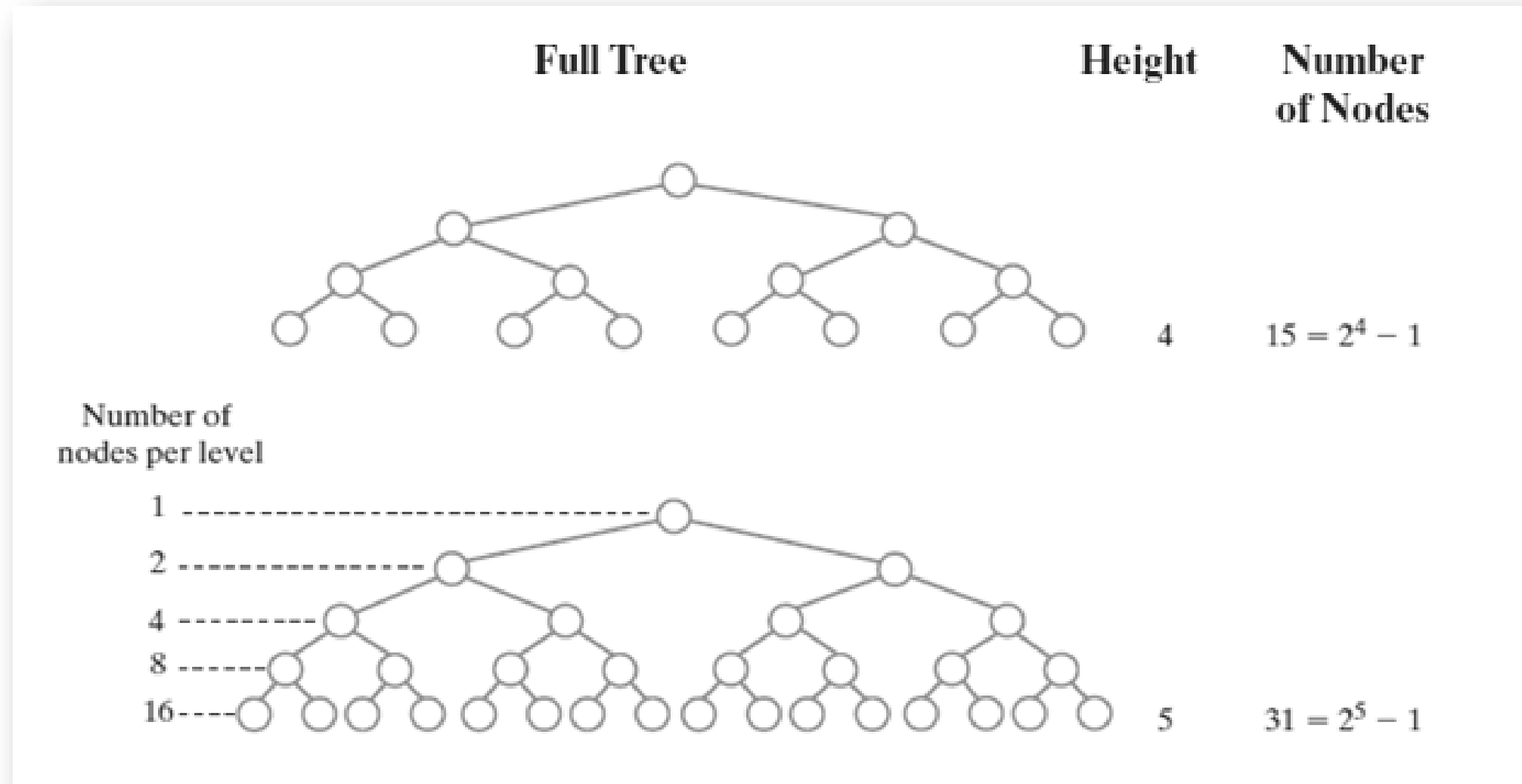


Balanced but not complete

# Binary Trees (cont.)

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

# Binary Trees (cont.)





# Binary Trees (cont.)

- Now, if  $n$  is the number of nodes in a full tree, we have the following results:

$$\begin{aligned}n &= 2^h - 1 \\2^h &= n + 1 \\\log_2(2^h) &= \log_2(n + 1) \\h &= \log_2(n + 1)\end{aligned}$$

- That is, the height of a full tree that has  $n$  nodes is  $\log_2(n + 1)$ .

# Programming Tip

- To compute  $\log_2(x)$  in Java, first observe that

$$\log_a(x) = \log_b(x) / \log_b(a)$$

- In Java, *Math.log*(*x*) returns the natural logarithm of *x*. So,

*Math.Log(x)/Math.Log(2.0)*

computes the base 2 logarithm of *x*.

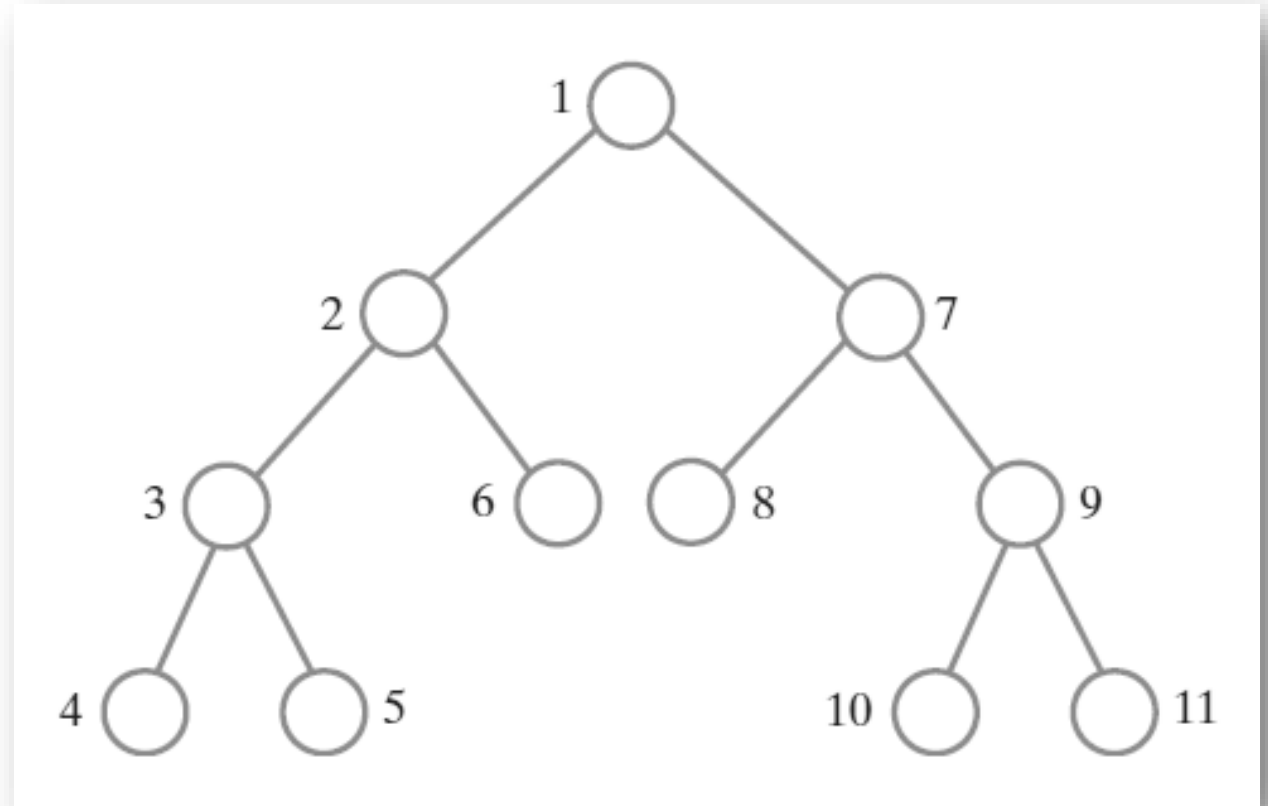
# Traversals of a Tree

- In defining a traversal, or iteration, of a tree, we must **visit**, or process, each data item exactly once.
- The order in which we visit items is not unique.
- Because traversals of a binary tree are somewhat easier to understand than traversals of a general tree, we begin there.

**Note:** "Visiting a node" means "processing the data within a node". It is an action that we perform during a traversal of a tree. A traversal can pass through a node without visiting it at that moment. Realize that traversals of a tree are based on the positions of its nodes, but not on the nodes' data values.

# Traversals of a Binary Tree

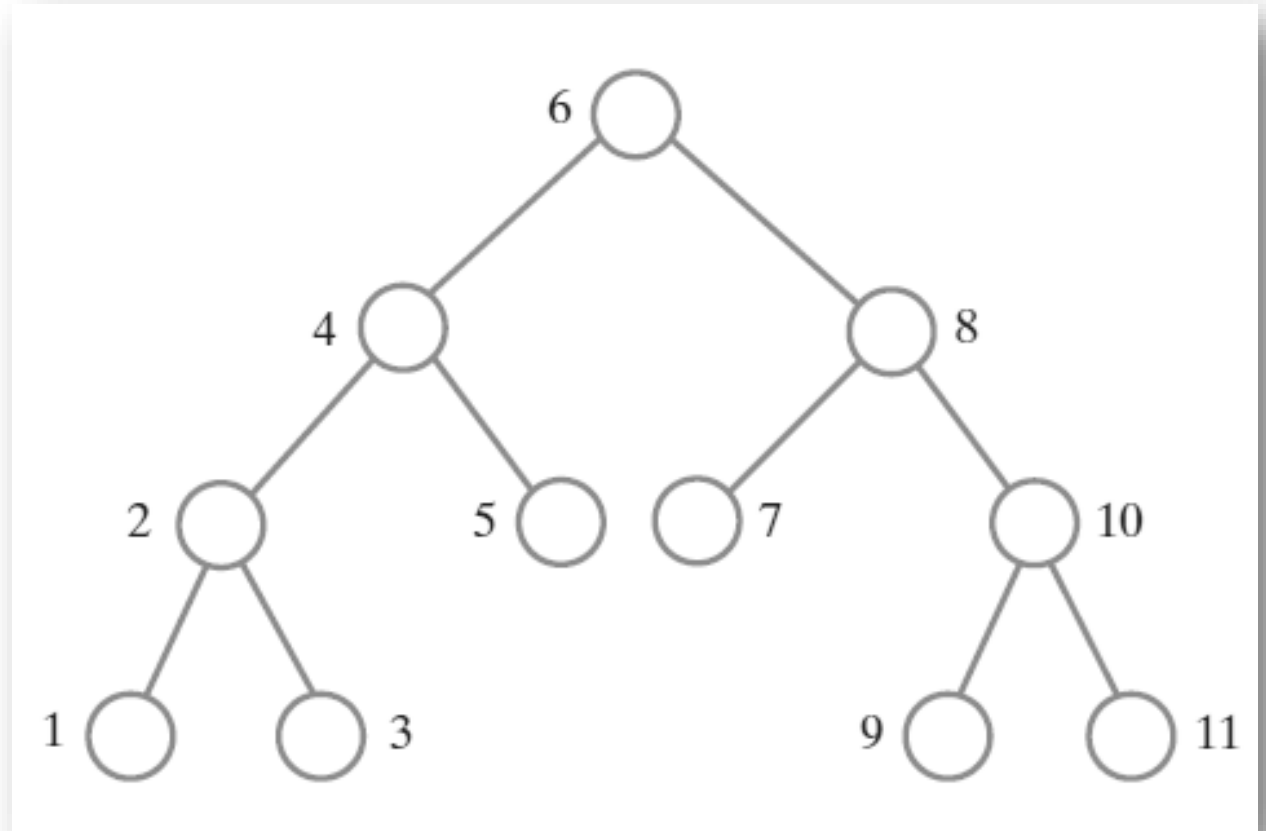
- In a **preorder traversal** we visit nodes in the following order:
  1. Visit the root
  2. Visit all the nodes in the root's left subtree
  3. Visit all the nodes in the root's right subtree



The visitation order of a preorder traversal.

# Traversals of a Binary Tree (cont.)

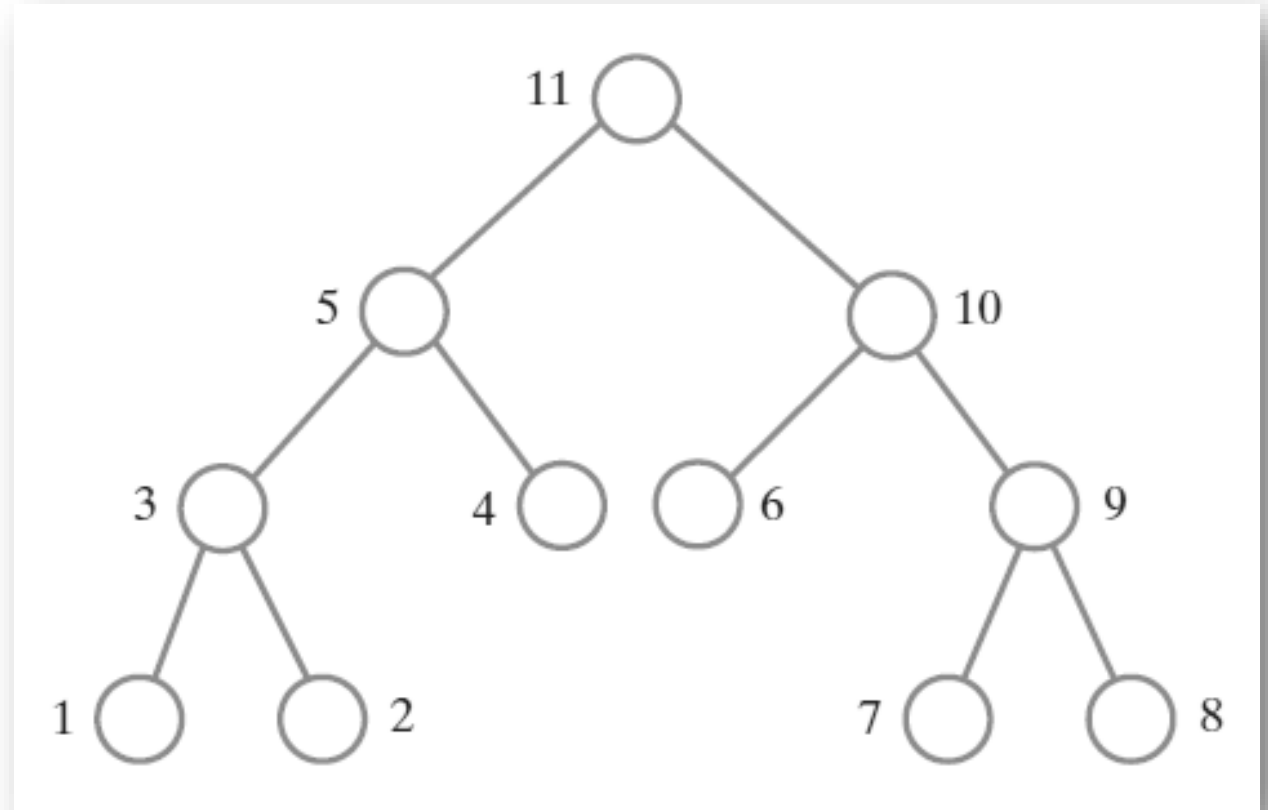
- In an **inorder traversal** we visit nodes in the following order:
  1. Visit all the nodes in the root's left subtree
  2. Visit the root
  3. Visit all the nodes in the root's right subtree



The visitation order of an inorder traversal.

# Traversals of a Binary Tree (cont.)

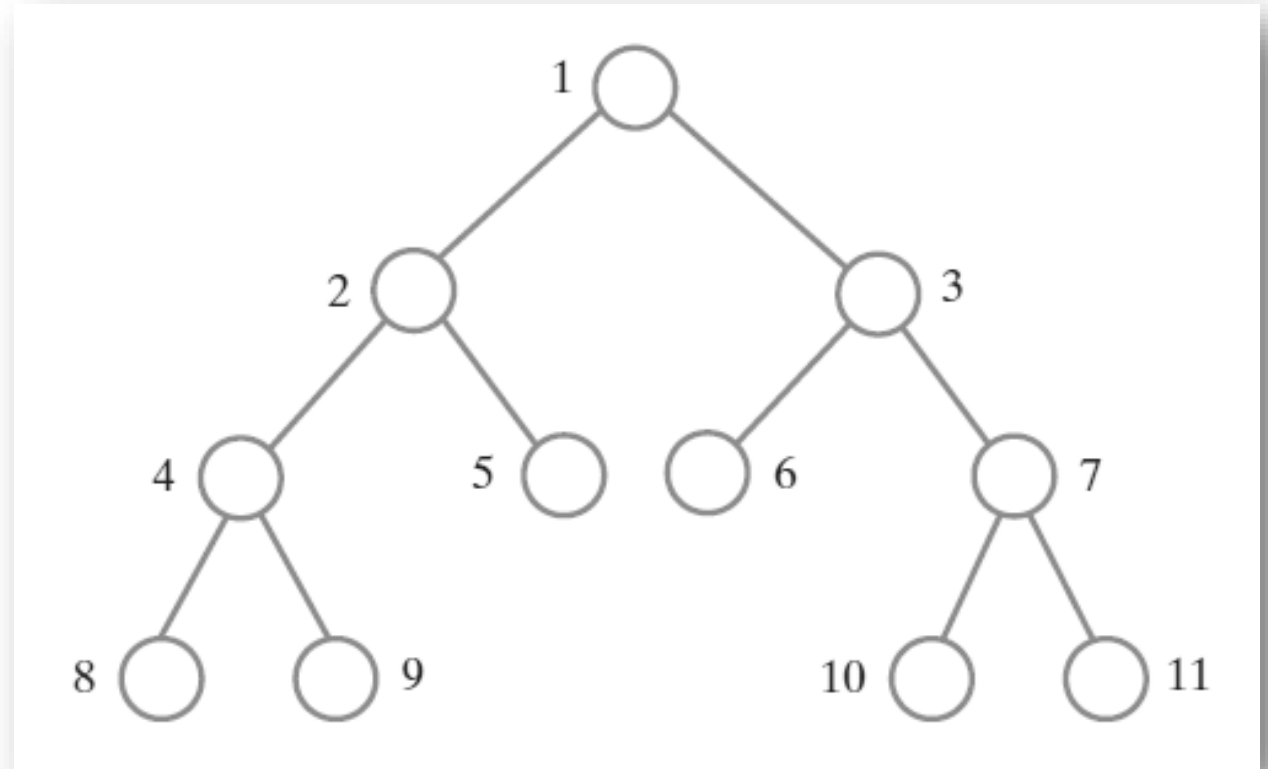
- In a **postorder traversal** we visit nodes in the following order:
  1. Visit all the nodes in the root's left subtree
  2. Visit all the nodes in the root's right subtree
  3. Visit the root



The visitation order of a postorder traversal.

# Traversals of a Binary Tree (cont.)

- The **level-order traversal** – the last traversal we will consider – begins at the root and visits nodes one level at a time.
- Within a level, it visits nodes from left to right.



The visitation order of a level-order traversal.

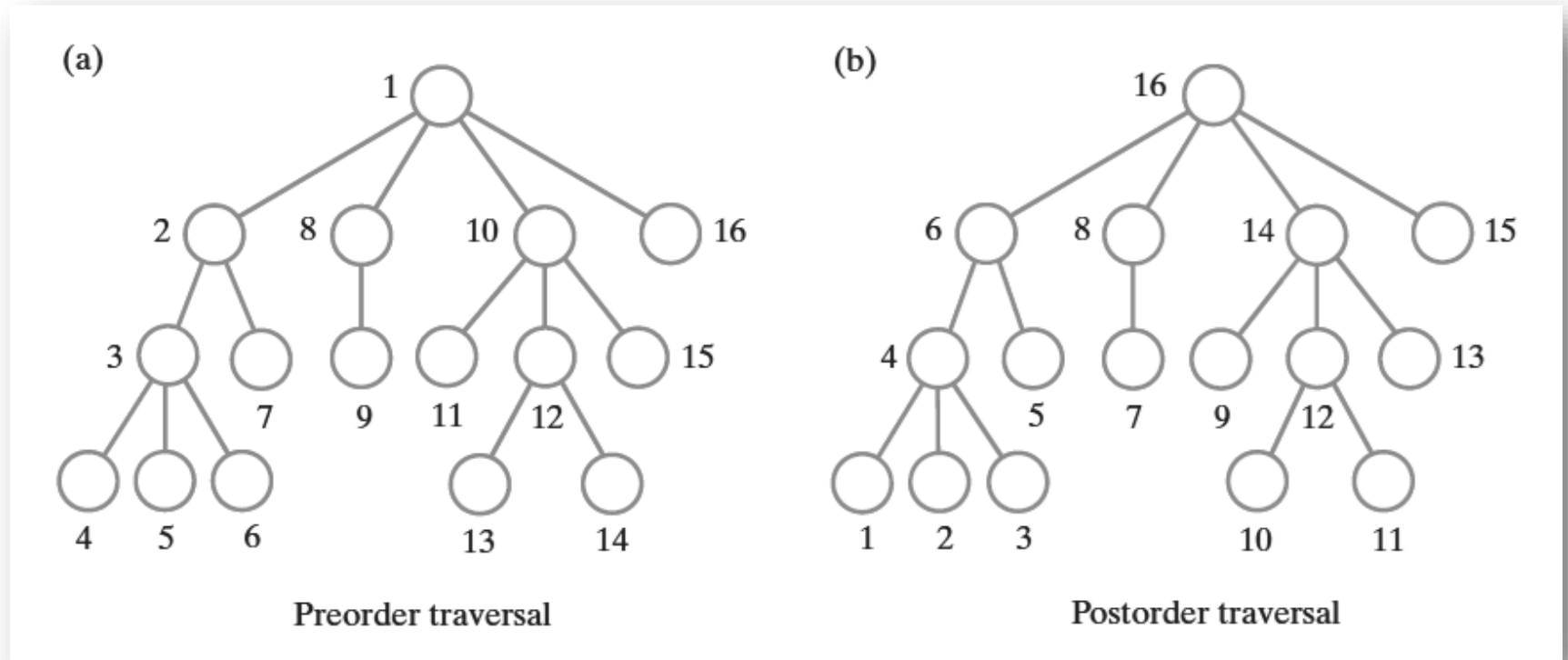
# Traversal of a Binary Tree (cont.)

- The **level-order traversal** is an example of a **breadth-first traversal**.
  - It follows a path that explores an entire level before moving to the next level.
- The **preorder traversal** is an example of a **depth-first traversal**.
  - This kind of traversal fully explores one subtree before exploring another. That is, the traversal follows a path that descends the levels of a tree as deeply as possible until it reaches the leaf.



# Traversals of General Tree

- A general tree has traversals that are in *level order*, *preorder*, and *postorder*. An *inorder* traversal is not well defined for a general tree.



The visitation order of two traversals of a general tree: (a) preorder; (b) postorder.

# Java Interfaces for Trees: Interface for All Trees

```
package TreePackage;

/**
 * An interface of basic methods for the ADT tree.
 */
public interface TreeInterface<T>
{
    public T getRootData();

    public int getHeight();

    public int getNumberOfNodes();

    public boolean isEmpty();

    public void clear();
} // end TreeInterface
```

# Traversals

```
package TreePackage;

import java.util.Iterator;

/**
 * An interface of iterators for the ADT tree.
 */
public interface TreeIteratorInterface<T>
{
    public Iterator<T> getPreorderIterator();

    public Iterator<T> getPostorderIterator();

    public Iterator<T> getInorderIterator();

    public Iterator<T> getLevelOrderIterator();
} // end TreeIteratorInterface
```

# An Interface for Binary Trees

```
package TreePackage;
/**
 * An interface for the ADT binary tree.
 */
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
```

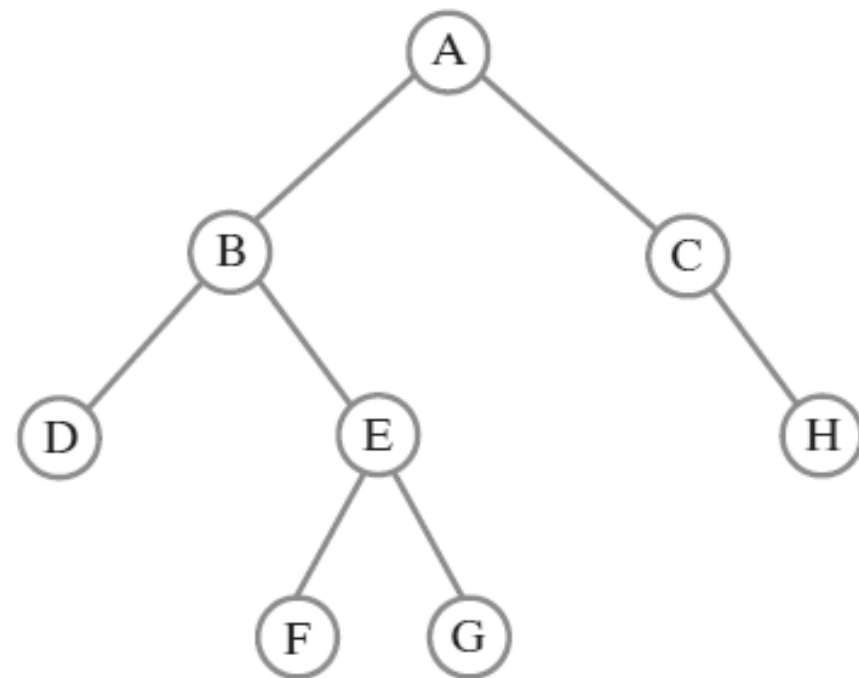
```

//Represent each leaf as a one-node tree
BinaryTreeInterface<String> dTree = new BinaryTree<>();
dTree.setTree("D");
BinaryTreeInterface<String> fTree = new BinaryTree<>();
fTree.setTree("F");
BinaryTreeInterface<String> gTree = new BinaryTree<>();
gTree.setTree("G");
BinaryTreeInterface<String> hTree = new BinaryTree<>();
hTree.setTree("H");
BinaryTreeInterface<String> emptyTree = new BinaryTree<>();

//Form larger subtrees
BinaryTreeInterface<String> eTree = new BinaryTree<>();
eTree.setTree("E", fTree, gTree); // Subtree rooted at E
BinaryTreeInterface<String> bTree = new BinaryTree<>();
bTree.setTree("B", dTree, eTree); // Subtree rooted at B
BinaryTreeInterface<String> cTree = new BinaryTree<>();
cTree.setTree("C", emptyTree, hTree); // Subtree rooted at C
BinaryTreeInterface<String> aTree = new BinaryTree<>();
aTree.setTree("A", bTree, cTree); // Desired tree rooted at A

//Display root, height, number of nodes
System.out.println("Root of tree contains " + aTree.getRootData());
System.out.println("Height of tree is " + aTree.getHeight());
System.out.println("Tree has " + aTree.getNumberOfNodes() + " nodes");

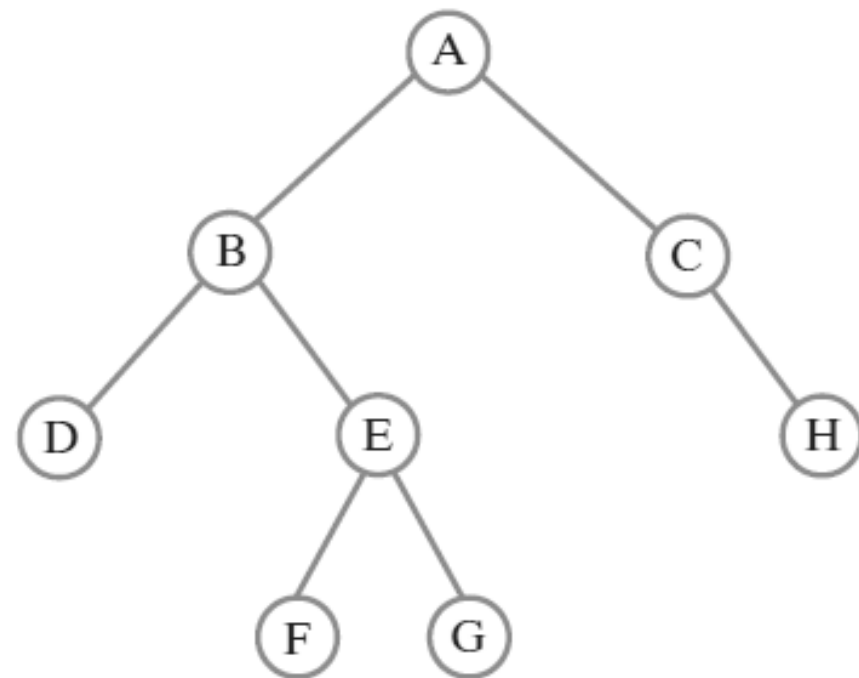
```



```
//Represent each leaf as a one-node tree
BinaryTreeInterface<String> dTree = new BinaryTree<>();
dTree.setTree("D");
BinaryTreeInterface<String> fTree = new BinaryTree<>();
fTree.setTree("F");
BinaryTreeInterface<String> gTree = new BinaryTree<>();
gTree.setTree("G");
BinaryTreeInterface<String> hTree = new BinaryTree<>();
hTree.setTree("H");
BinaryTreeInterface<String> emptyTree = new BinaryTree<>();

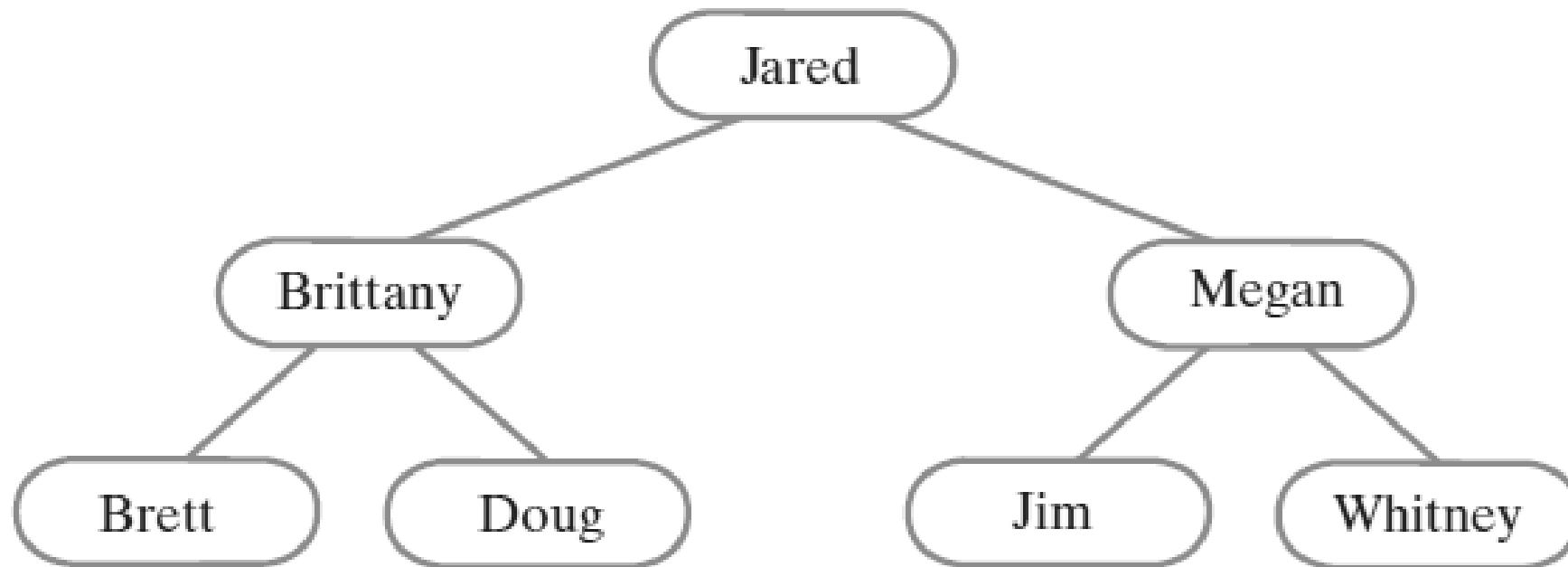
//Form larger subtrees
BinaryTreeInterface<String> eTree = new BinaryTree<>();
eTree.setTree("E", fTree, gTree); // Subtree rooted at E
BinaryTreeInterface<String> bTree = new BinaryTree<>();
bTree.setTree("B", dTree, eTree); // Subtree rooted at B
BinaryTreeInterface<String> cTree = new BinaryTree<>();
cTree.setTree("C", emptyTree, hTree); // Subtree rooted at C
BinaryTreeInterface<String> aTree = new BinaryTree<>();
aTree.setTree("A", bTree, cTree); // Desired tree rooted at A
```

```
//Display nodes in preorder
System.out.println("A preorder traversal visits nodes in this order:");
Iterator<String> preorder = aTree.getPreorderIterator();
while (preorder.hasNext())
    System.out.print(preorder.next() + " ");
System.out.println();
```



# Binary Search Trees

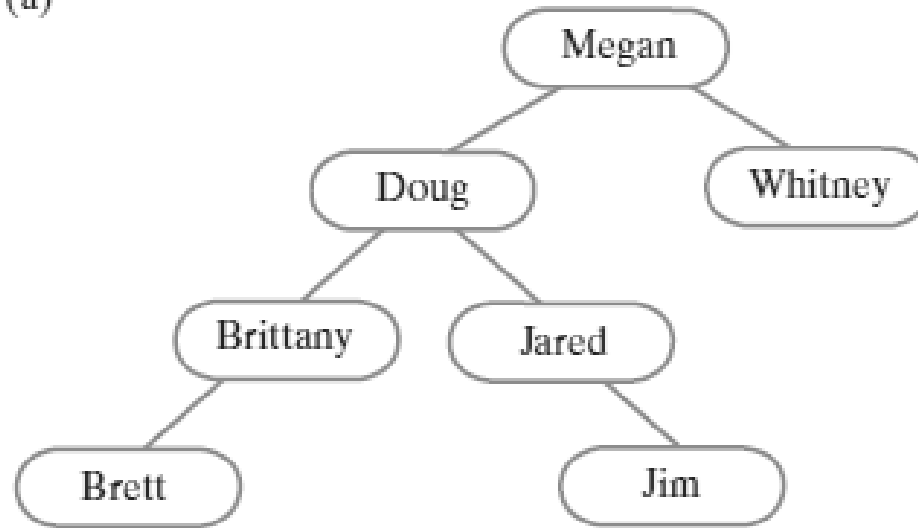
- A **binary search tree** is a *binary tree* whose nodes contain comparable objects.
- For each node in a binary search tree,
  - The node's data is greater than all the data in the node's left subtree
  - The node's data is less than all the data in the node's right subtree
- Every node in a binary search tree is the root of a binary search tree.



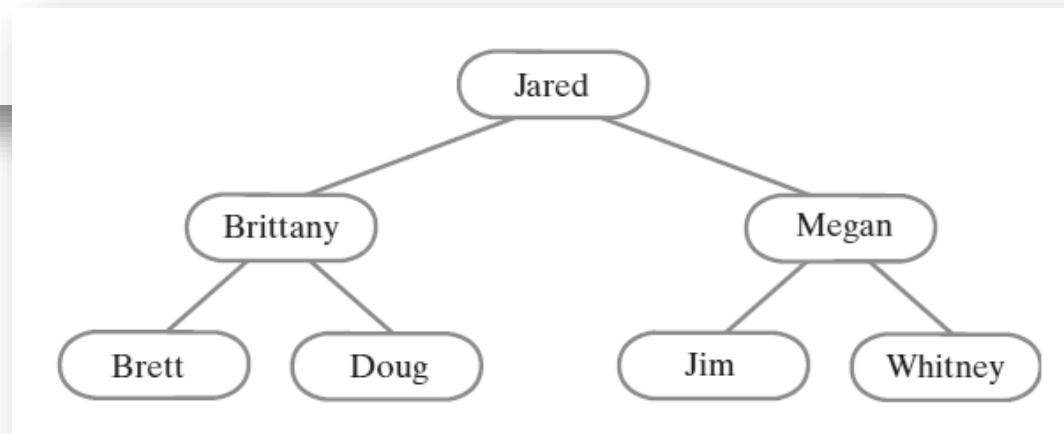
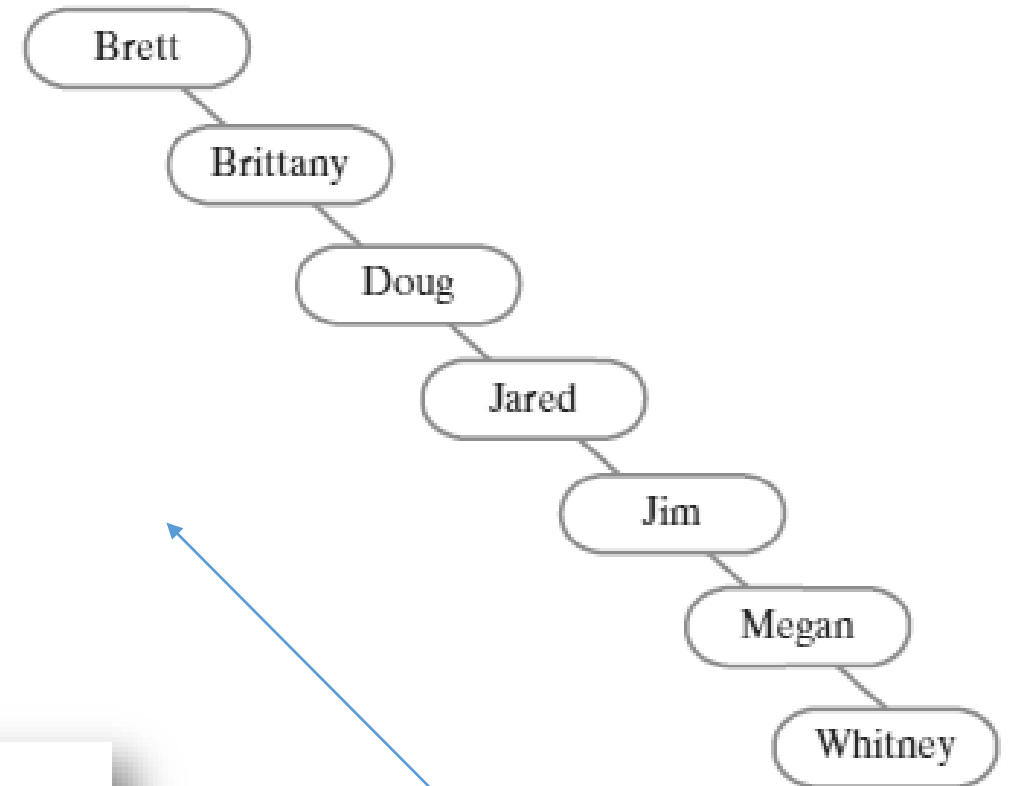
A binary search tree of names.



(a)



(b)



Two binary search trees ((a) and (b)) containing the same data as the tree on the left.

# Searching a binary search tree

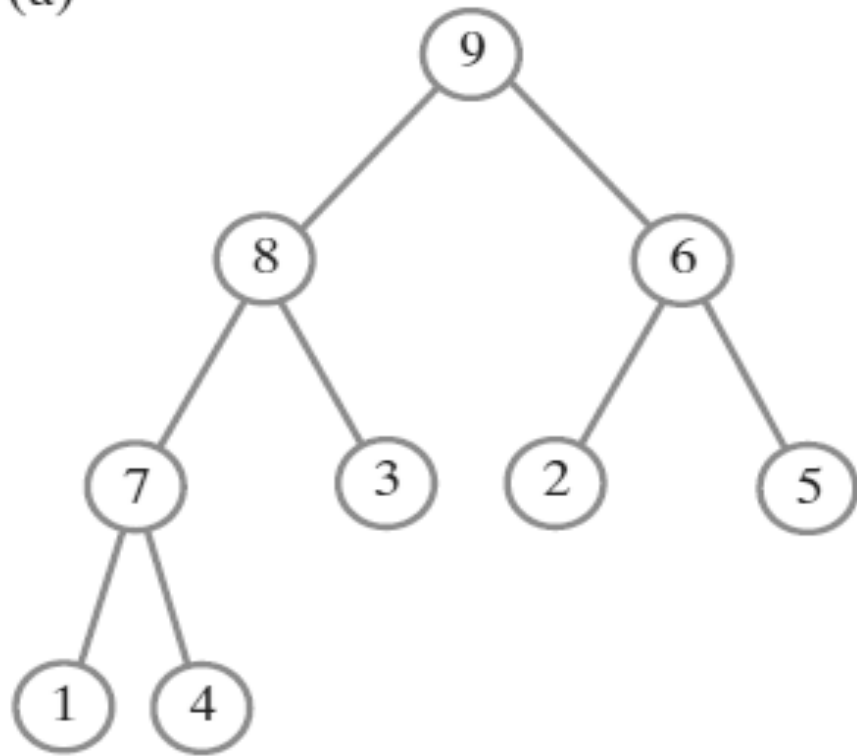
```
Algorithm bstSearch(binarySearchTree, desiredObject)  
// Searches a binary search tree for a given object.  
// Returns true if the object is found.  
  
if (binarySearchTree is empty)  
    return false  
else if (desiredObject == object in the root of binarySearchTree)  
    return true  
else if (desiredObject < object in the root of binarySearchTree)  
    return bstSearch(left subtree of binarySearchTree, desiredObject)  
else  
    return bstSearch(right subtree of binarySearchTree, desiredObject)
```

# Heaps

- A **heap** is a *complete binary tree* whose nodes contain comparable objects.
- Each node contains an object that is no smaller (or no larger) than the objects in its descendants.
- In a **maxheap**, the object in a node is greater than or equal to its descendant objects.
- In a **minheap**, the relation is less than or equal to its descendant objects.

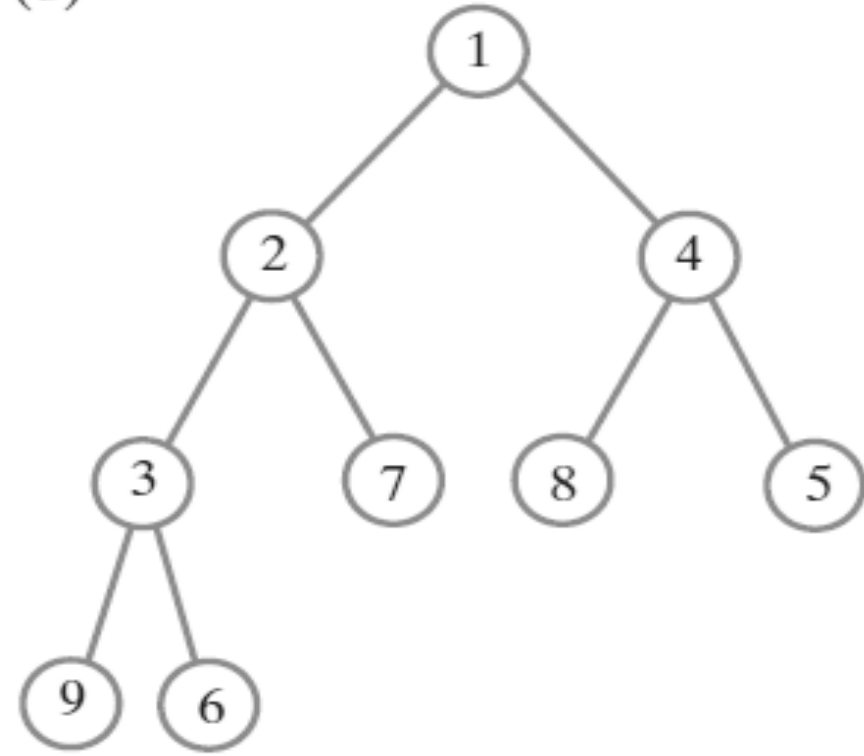
# Heaps (cont.)

(a)



Maxheap

(b)



Minheap

```
/**
 * An interface for the ADT maxheap.
 */
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
    /**
     * Adds a new entry to this heap.
     *
     * @param newEntry An object to be added.
     */
    public void add(T newEntry);

    /**
     * Removes and returns the largest item in this heap.
     *
     * @return Either the largest object in the heap or, if the heap is empty
     *         before the operation, null.
     */
    public T removeMax();

    /**
     * Retrieves the largest item in this heap.
     *
     * @return Either the largest object in the heap or, if the heap is empty, null.
     */
    public T getMax();
}
```

```
/**
 * Detects whether this heap is empty.
 *
 * @return True if the heap is empty, or false otherwise.
 */
public boolean isEmpty();

/**
 * Gets the size of this heap.
 *
 * @return The number of entries currently in the heap.
 */
public int getSize();

/** Removes all entries from this heap. */
public void clear();
} // end MaxHeapInterface
```

# References

- F. C. Carrano and T. M. Henry, "Data Structures and Abstractions with Java", 4<sup>th</sup> ed., 2015. Pearson Education, Inc.