

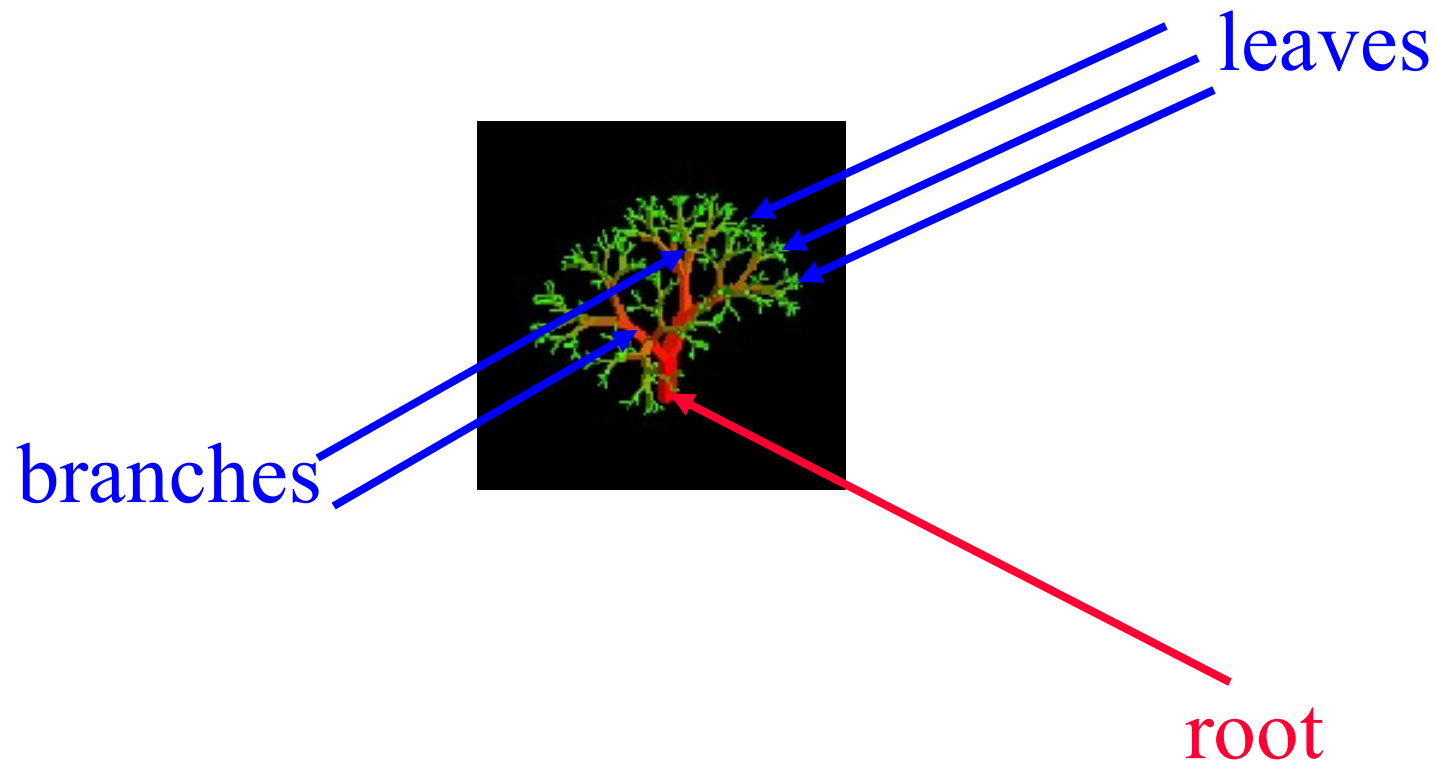
Trees

Prof. Ki-Hoon Lee

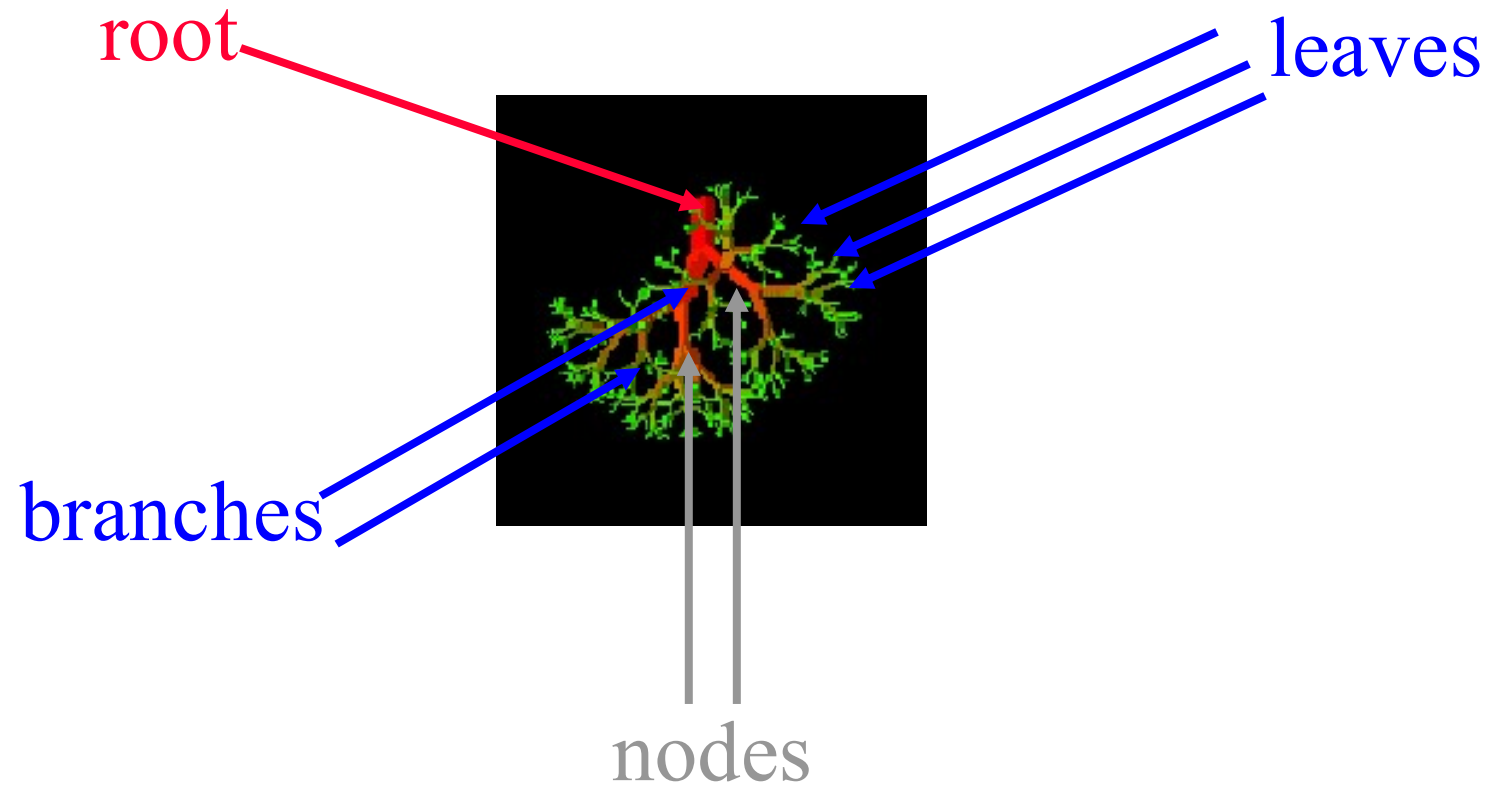
School of Computer and Information Engineering

Kwangwoon University

Nature Lover's View of a Tree



Computer Scientist's View



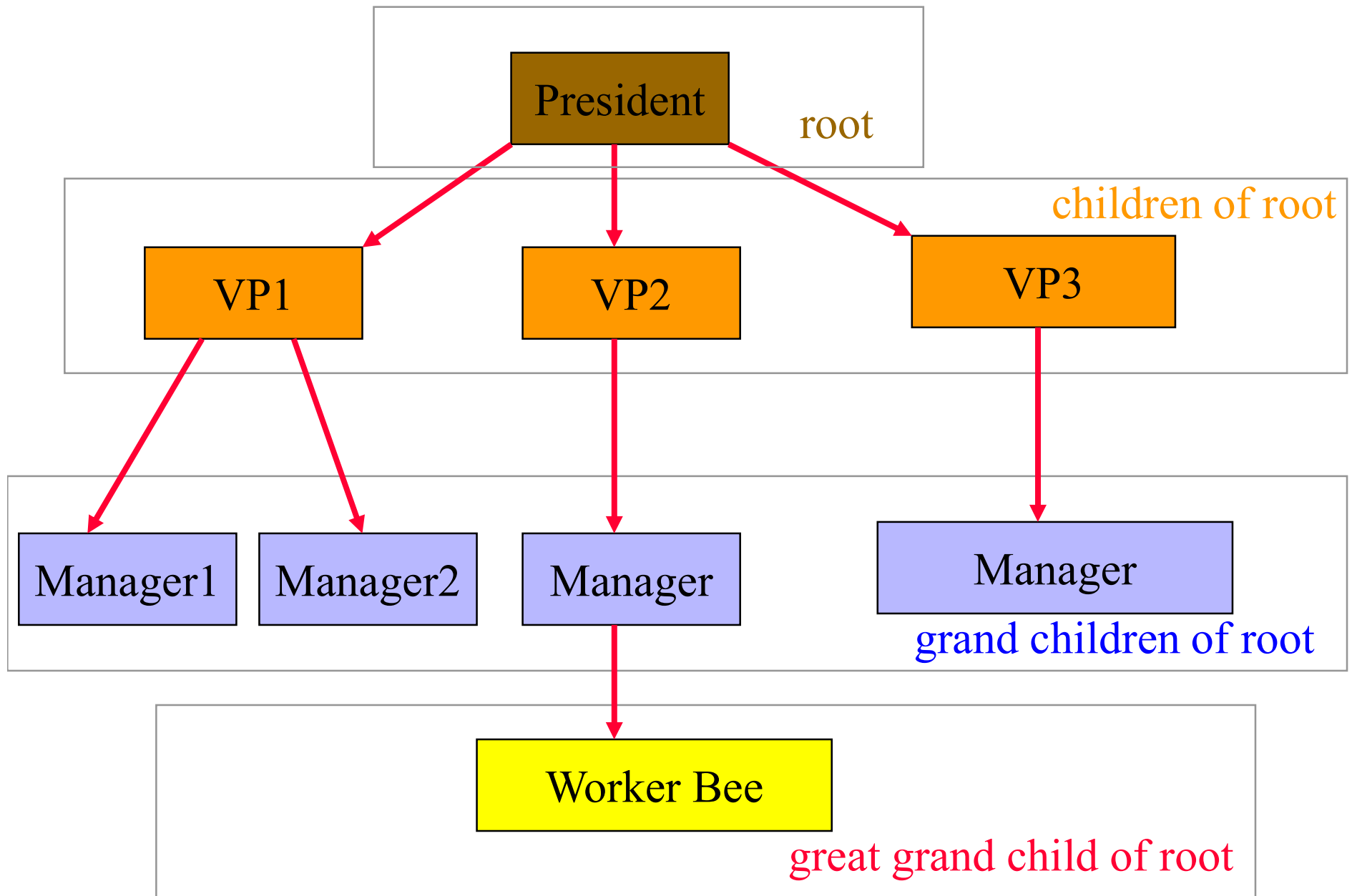


Hierarchical Data and Trees



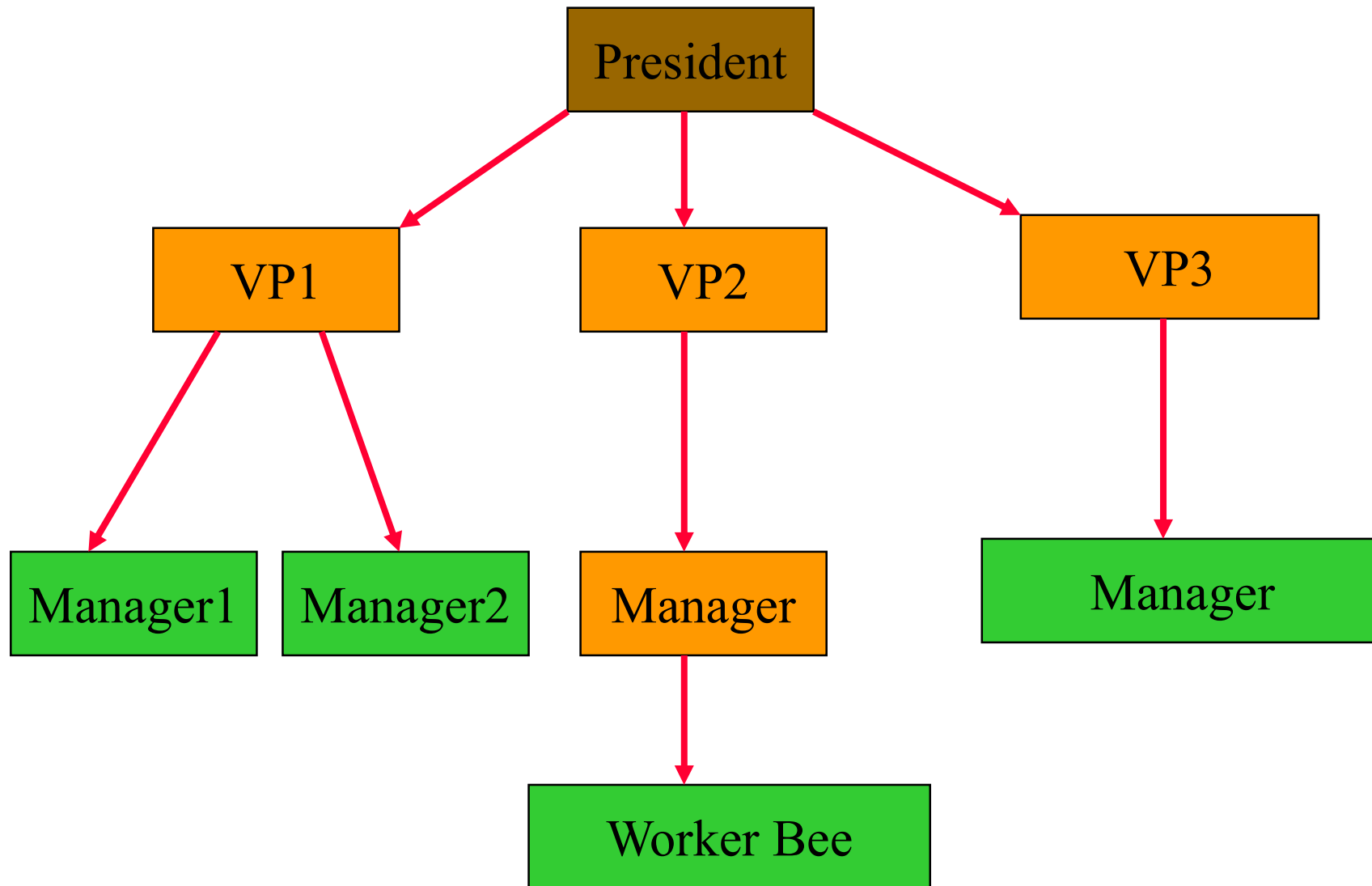
- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the **children** of the root.
- Elements next in the hierarchy are the **grandchildren** of the root, and so on.
- Elements that have no children are **leaves**.

Example Tree





Leaves



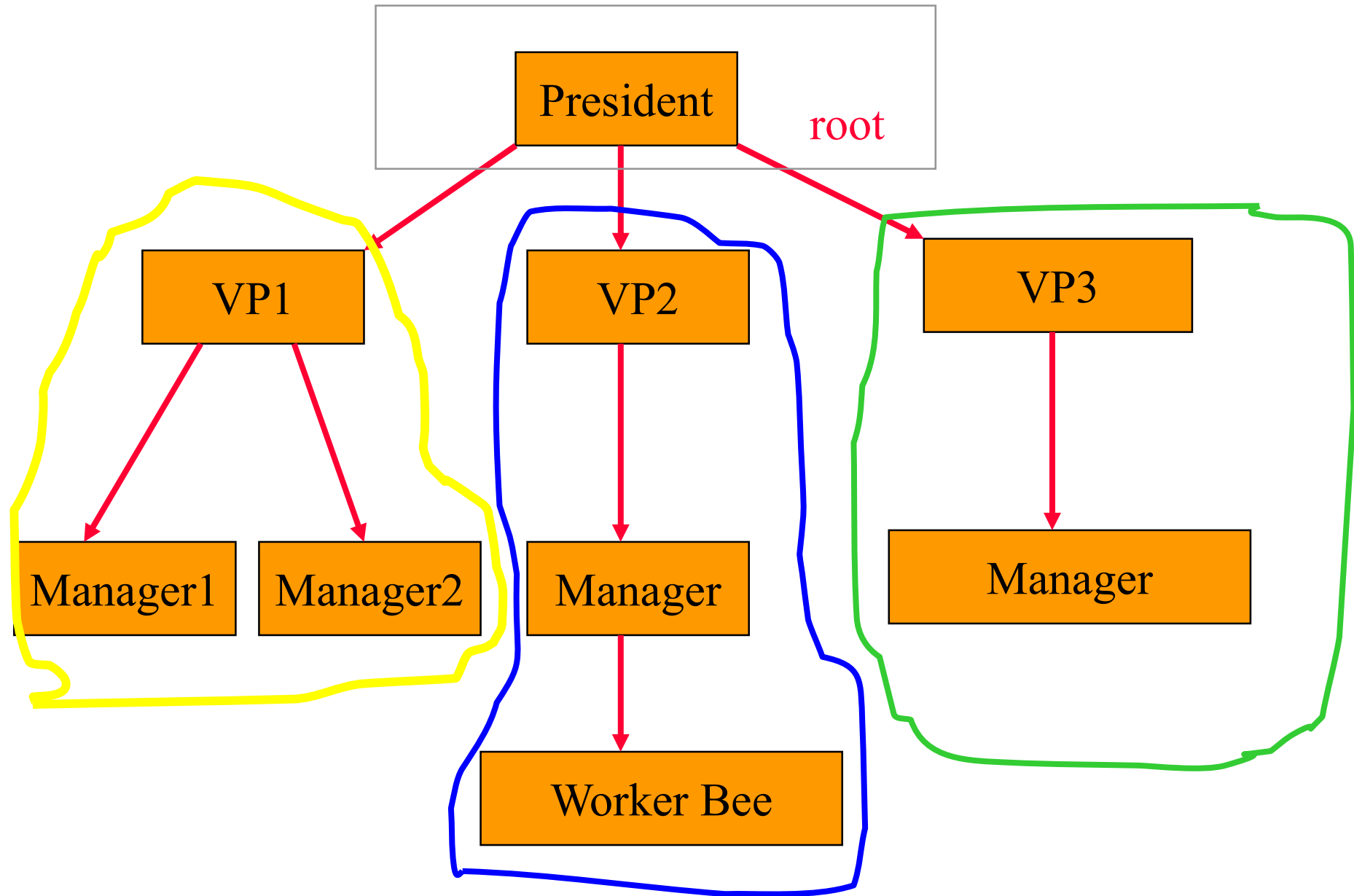
Definition of a Tree

A *tree* is a finite set of one or more nodes such that:

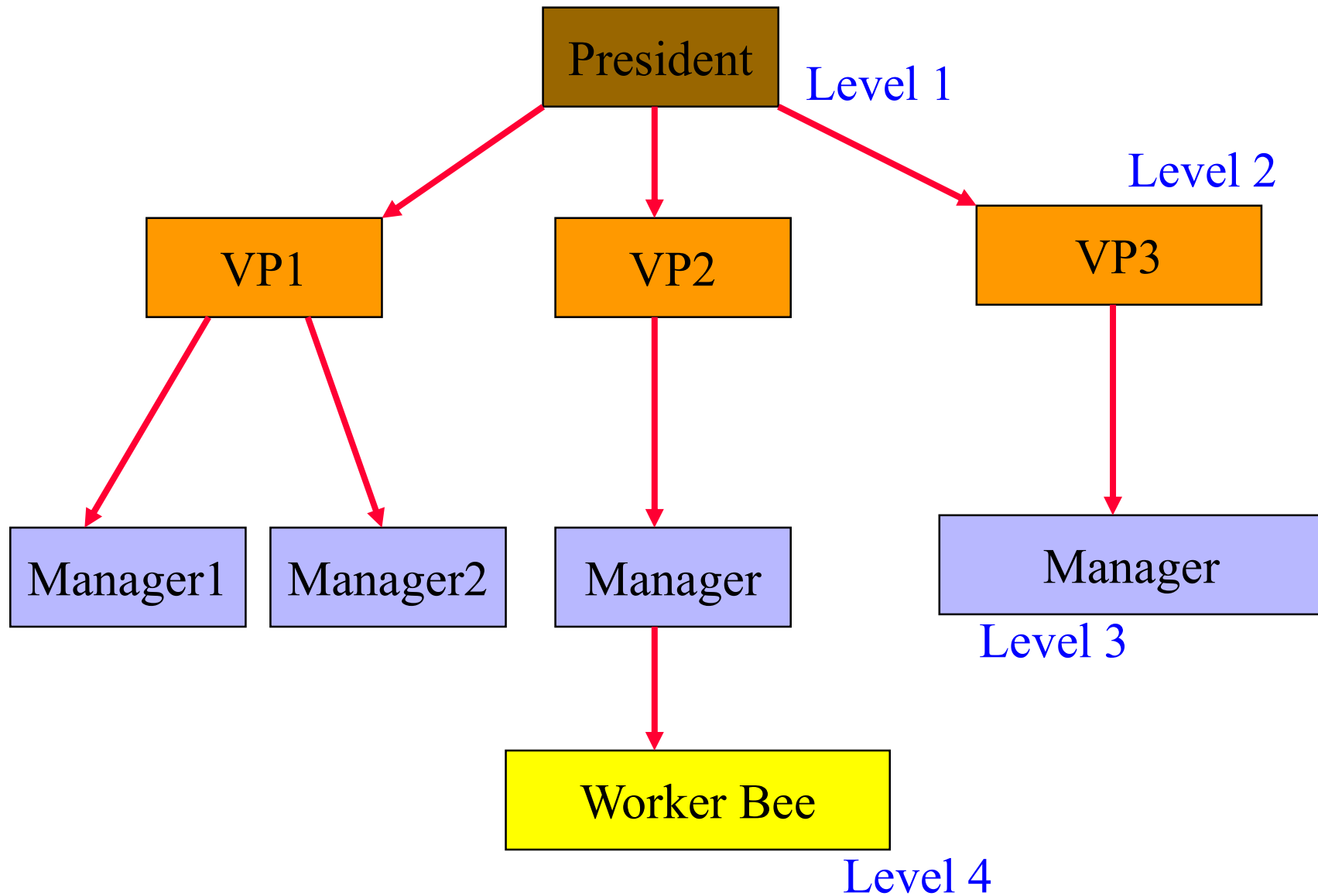
- (i) there is a specially designated node called the *root*;
- (ii) the remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n where each of these sets is a tree. T_1, \dots, T_n are called the *subtrees* of the root.



Subtrees



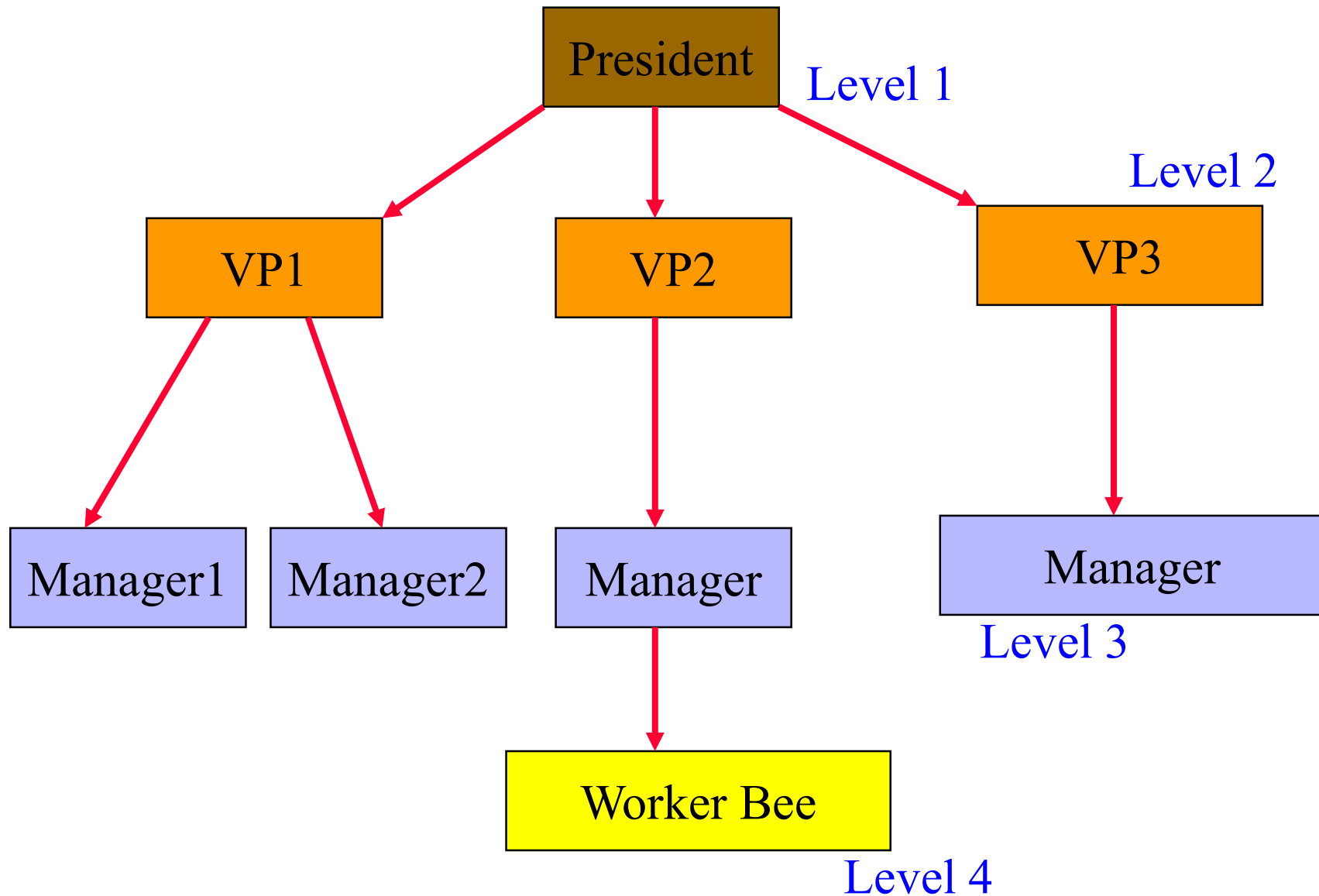
Levels



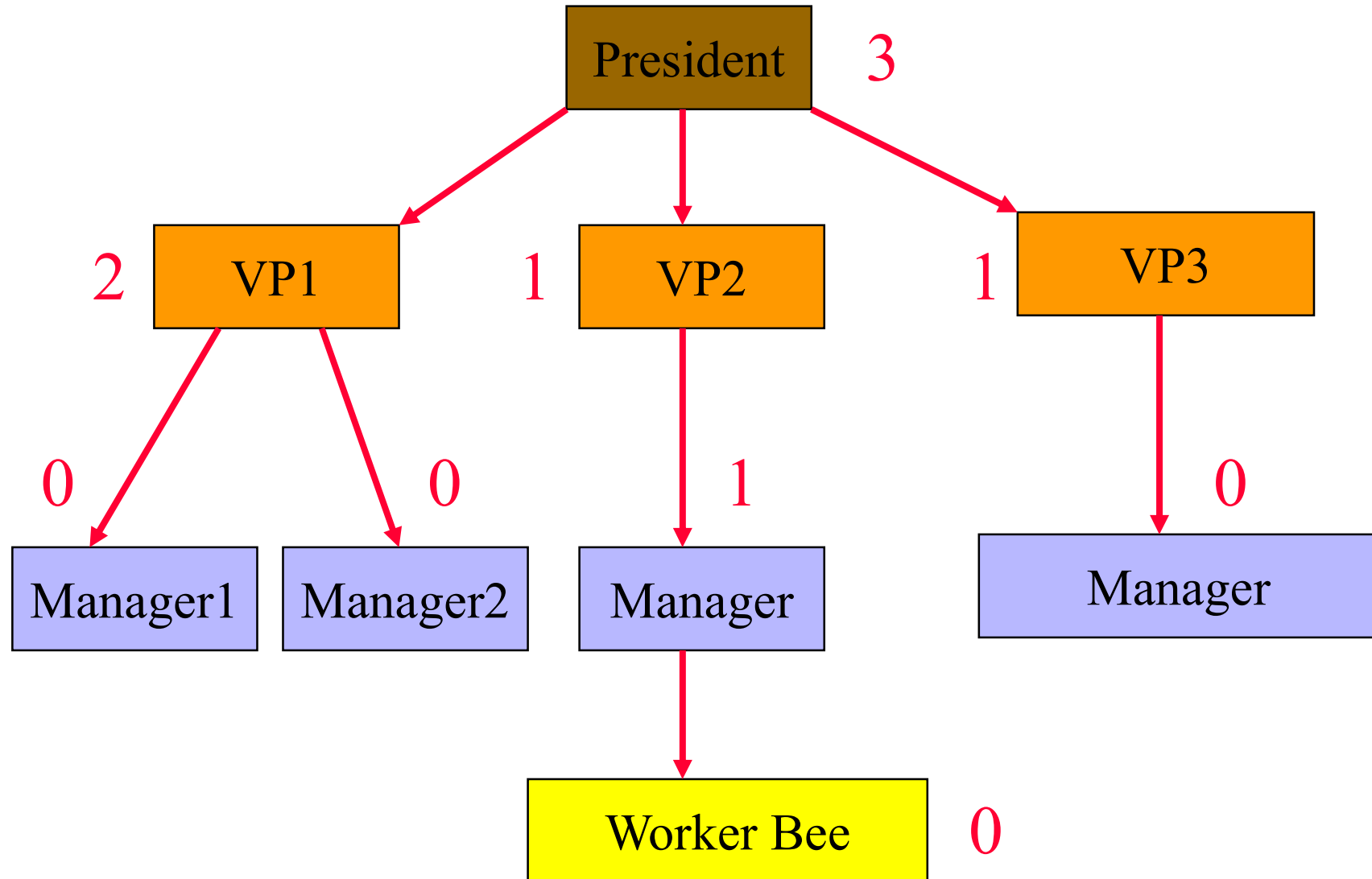
Caution

- Some texts start level numbers at 0 rather than at 1.
- Root is at level 0.
- Its children are at level 1.
- The grand children of the root are at level 2.
- And so on.
- We shall number levels with the root at level 1.

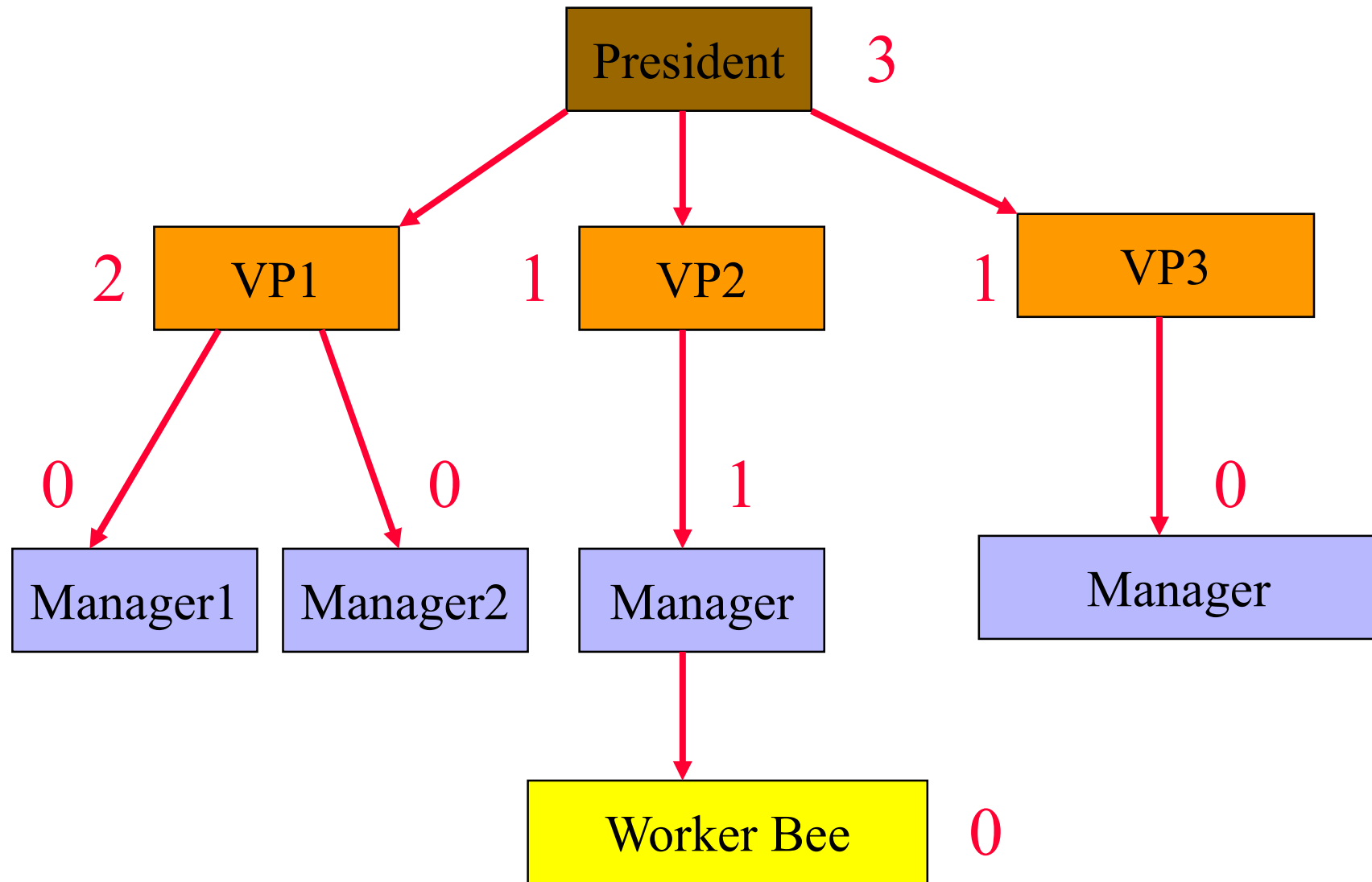
Height = Depth = Maximum Level



Node Degree = Number of Children



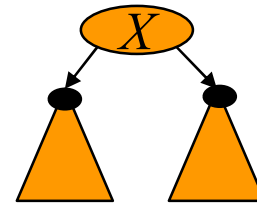
Tree Degree = Max Node Degree



Degree of tree = 3.

Terminology

- *Node*: the item of information (plus the branches to other items)
- *Degree* of a node: the number of subtrees of the node
- *Leaf* (or *terminal*) nodes: Nodes that have degree zero
 - The other nodes are referred to as *internal* (or *nonterminal*) nodes
- The roots of the subtrees of a node, X , are the *children* of X . X is the *parent* of its children.



- Children of the same parent are said to be *siblings*

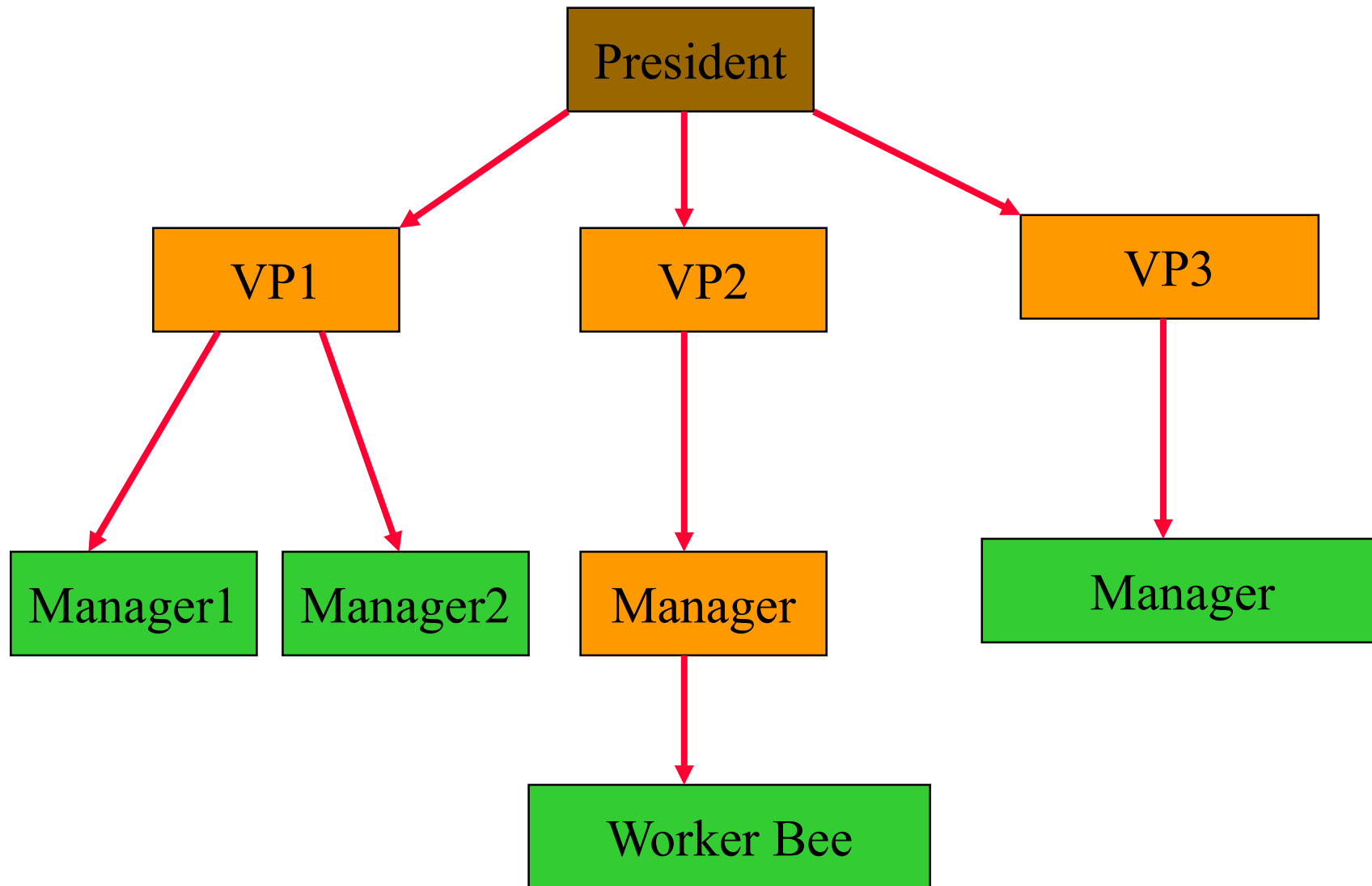
Terminology (Cont.)

- *Level*: letting the root be at level one. If a node is at level l , then its children are at level $l + 1$
- *Height* (or *depth*) of a tree is the maximum level of any node in the tree
- *Degree of a tree*: the maximum degree of the nodes in the tree

Terminology (Cont.)

- *Ancestors* of a node: all the nodes along the path from the root to the node
- *Descendants* of a node: all the nodes that are in its subtrees
- *Forest*: a set of disjoint trees. If we remove the root of a tree, we get a forest

Parent, Grandparent, Siblings, Ancestors, Descendants



Binary Tree

- Finite (possibly empty) collection of elements.
- A **nonempty** binary tree has a **root** element.
- The remaining elements (if any) are partitioned into **two** binary trees.
- These are called the **left** and **right** subtrees of the binary tree.

Differences Between a Tree & a Binary Tree

- No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
- A binary tree may be empty; a tree cannot be empty.

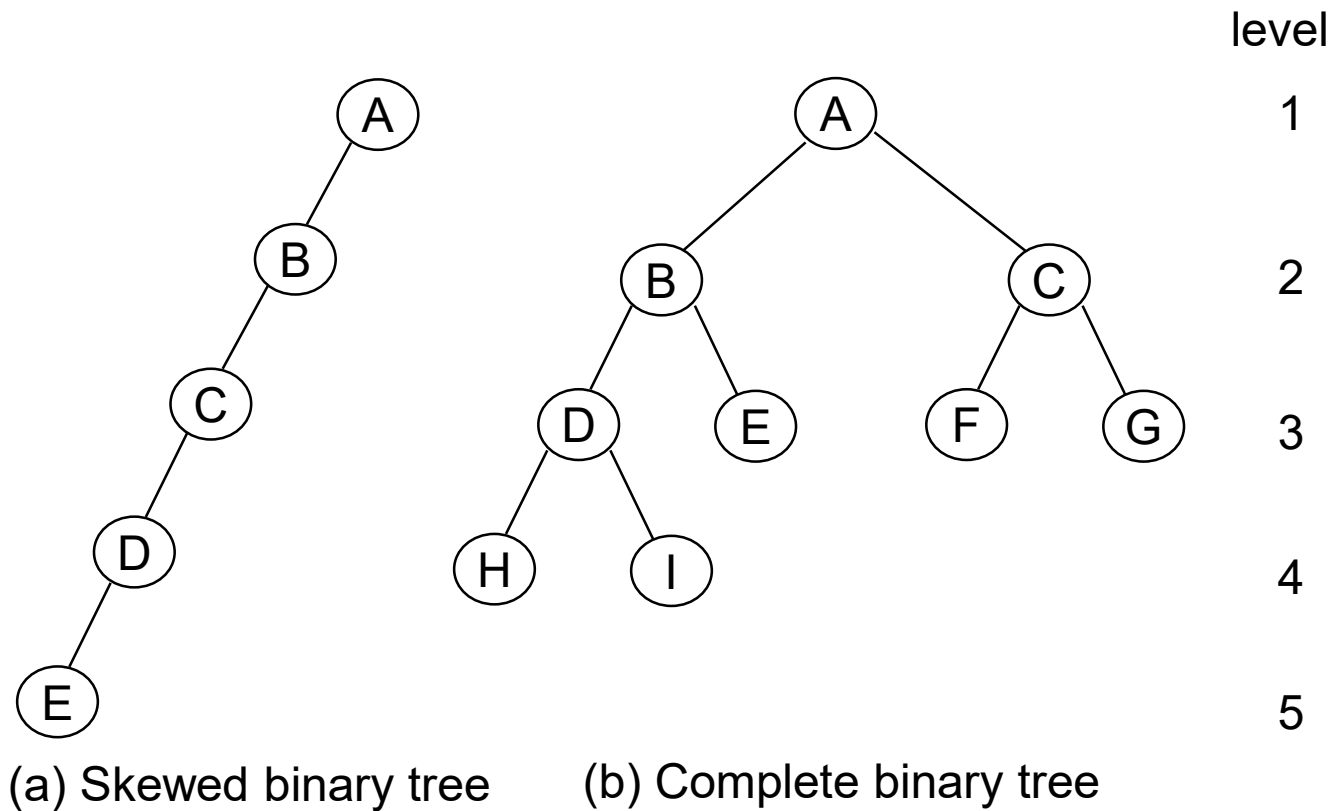
Differences Between a Tree & a Binary Tree (Cont.)

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



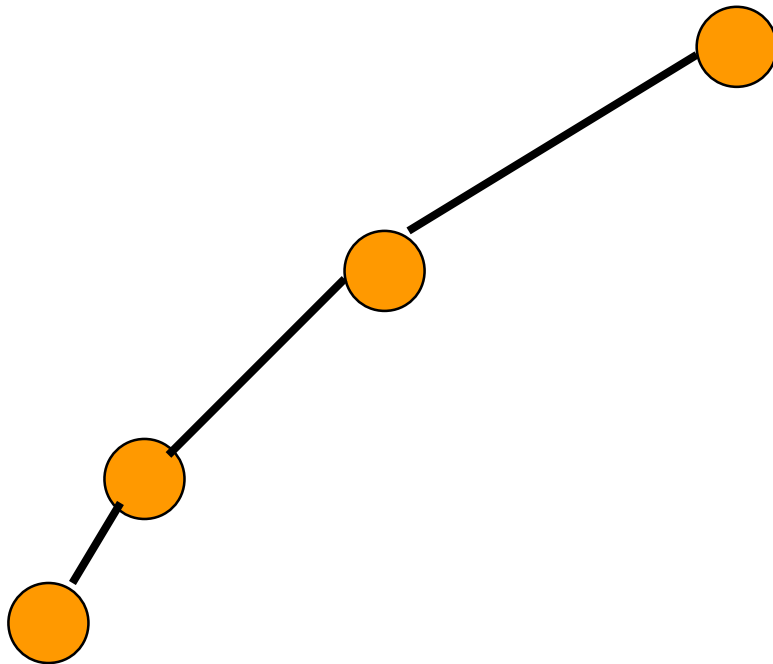
- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Examples of Binary Trees



Minimum Number of Nodes

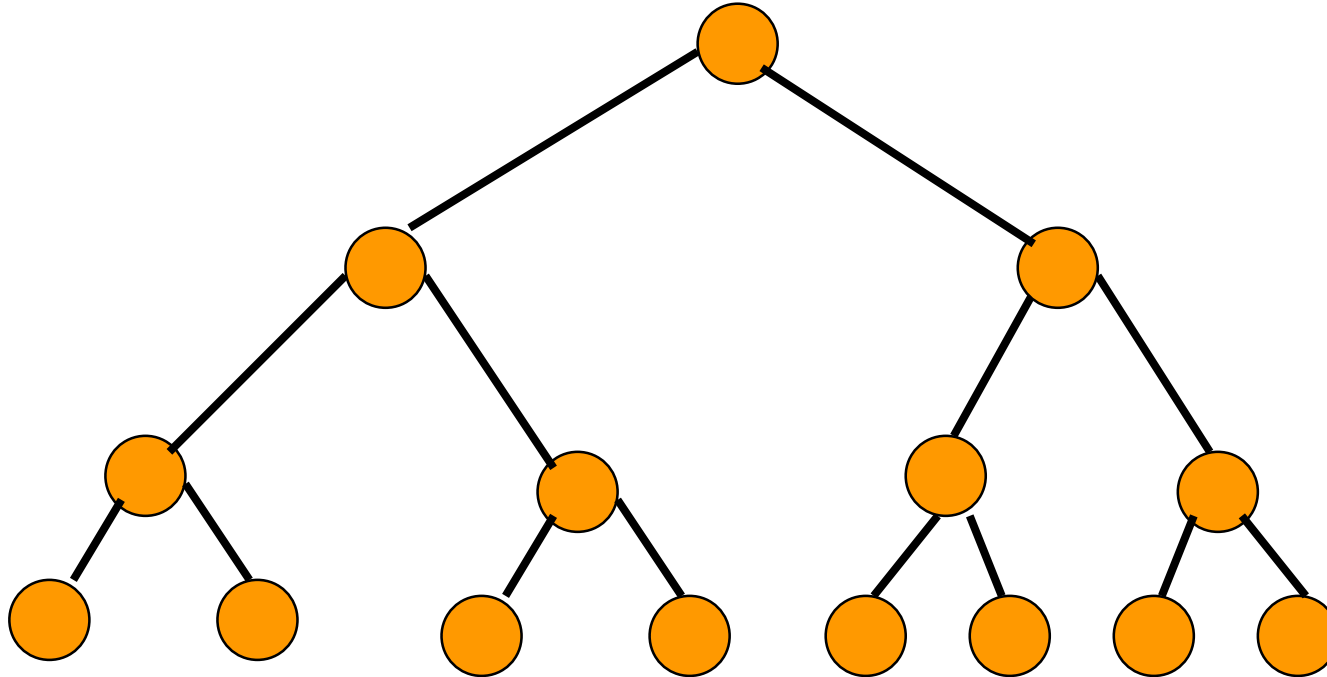
- Minimum number of nodes in a binary tree whose height is h .
- At least one node at each of first h levels.



minimum number of
nodes is h

Maximum Number of Nodes

- All possible nodes at first h levels are present.



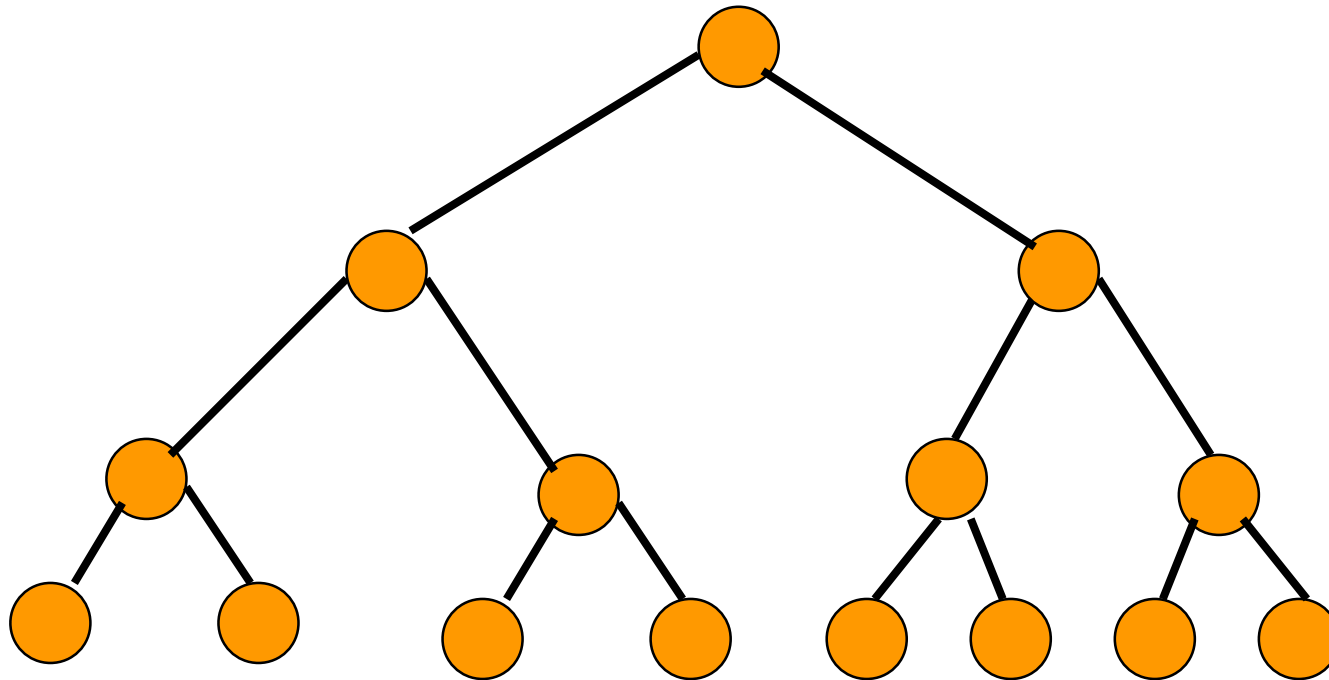
Maximum number of nodes

$$= 1 + 2 + 4 + 8 + \dots + 2^{h-1}$$

$$= 2^h - 1$$

Full Binary Tree

- A full binary tree of a given height h has $2^h - 1$ nodes.



Height 4 full binary tree has 15 nodes.

Number of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h .
- $h \leq n \leq 2^h - 1$
- $\log_2(n+1) \leq h \leq n$
- skewed tree: $h = n$
- full binary tree: $h = \log_2(n+1)$

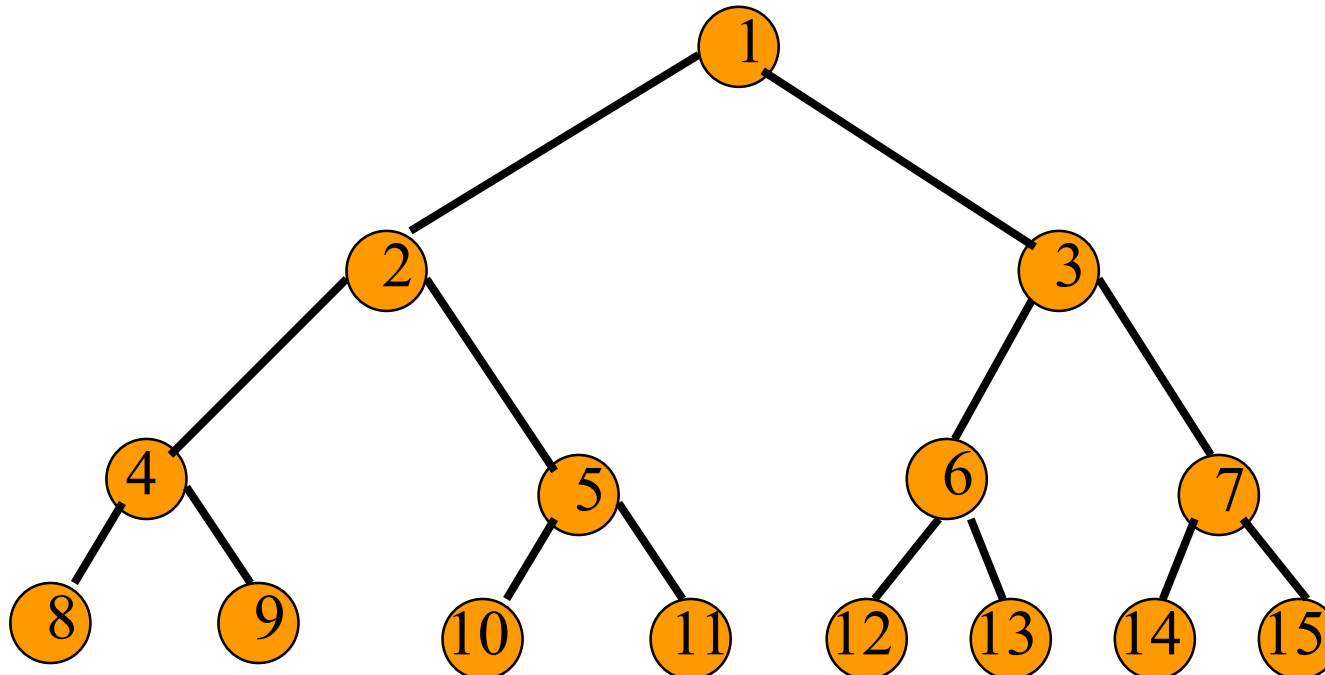
Properties of Binary Trees

- The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$
 - The proof is by induction on i
- The maximum number of nodes in a binary tree of depth h is $2^h - 1$

$$\sum_{i=1}^h 2^{i-1} = 2^h - 1$$

Numbering Nodes in a Full Binary Tree

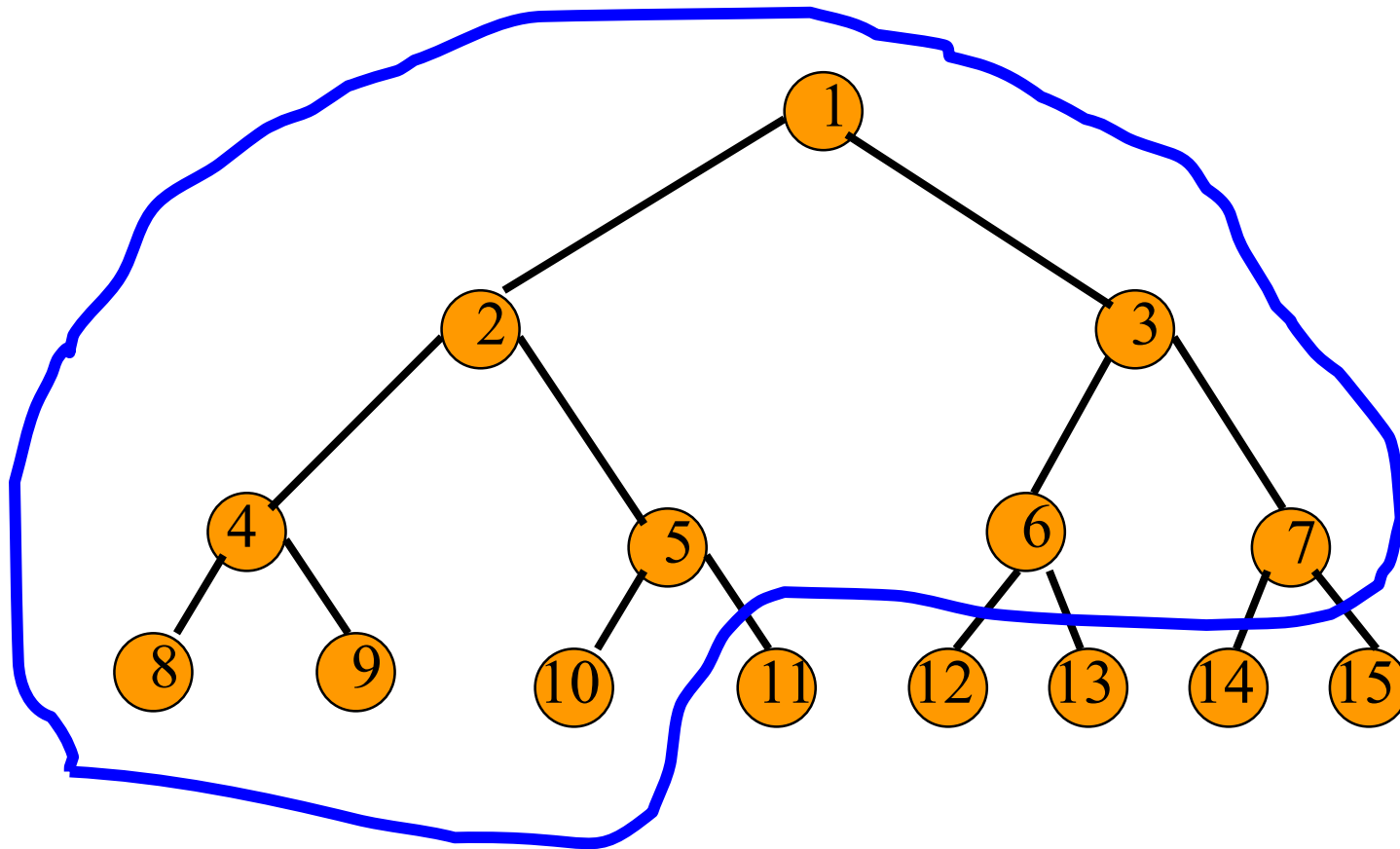
- Number the nodes **1** through $2^h - 1$.
- Number by levels from top to bottom.
- Within a level number from left to right.



Complete Binary Tree with n Nodes

- Start with a full binary tree that has at least n nodes.
- Number the nodes as described earlier.
- The binary tree defined by the nodes numbered 1 through n is the unique n node complete binary tree.

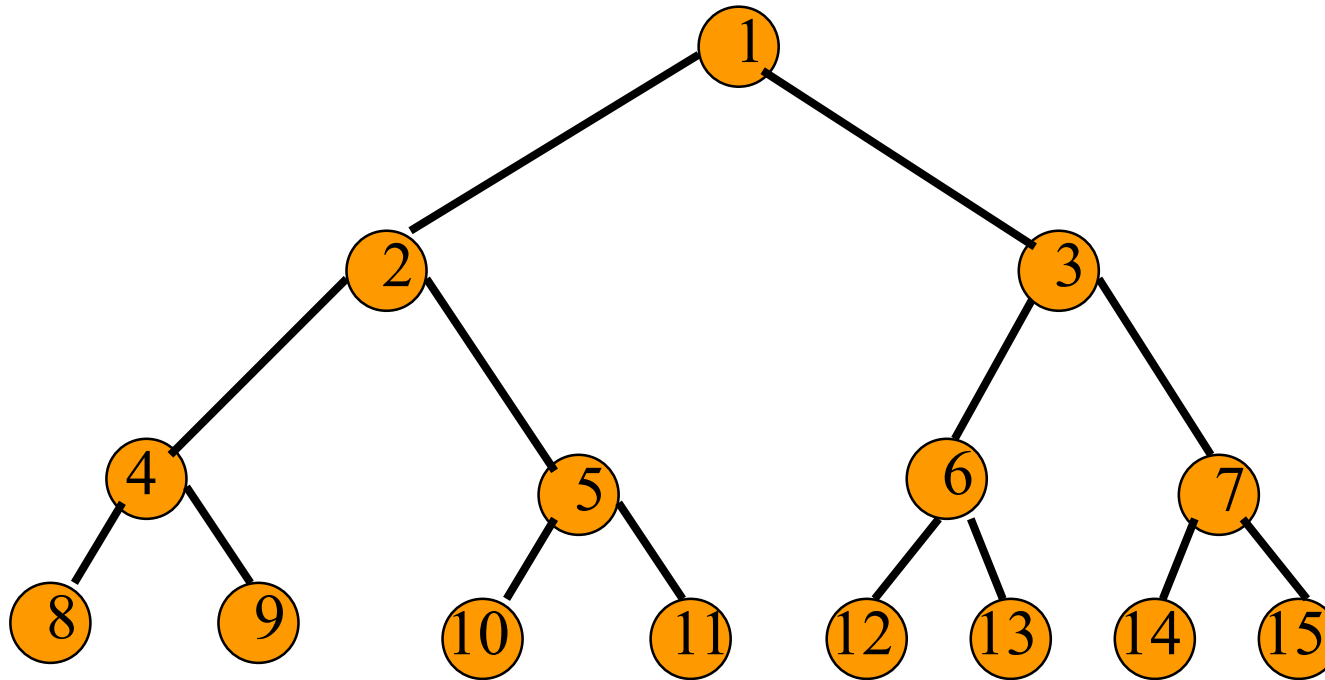
Example ($n = 10$)



Complete Binary Tree (Cont.)

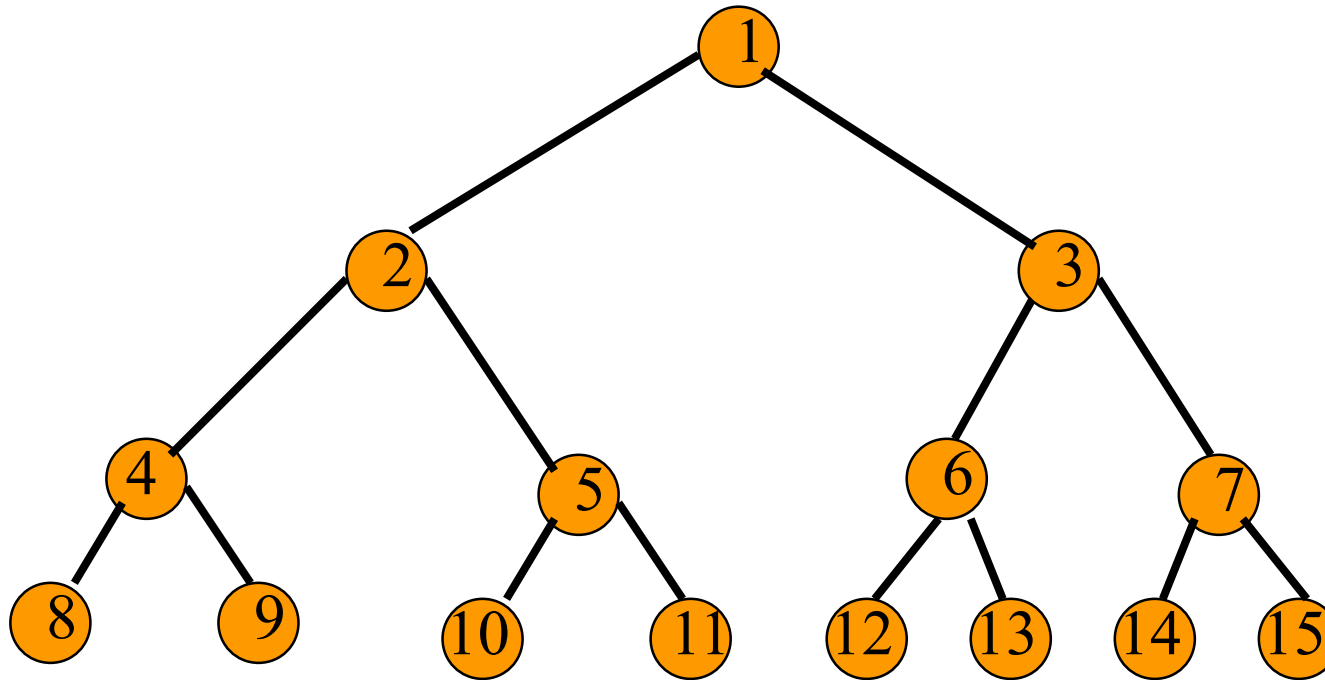
- Definition iff: if and only if
 - A binary tree with n nodes and of depth h is *complete* iff its nodes correspond to the nodes which numbered from 1 to n in the full binary tree of depth h .
- Note $\lceil x \rceil$: the smallest integer $\geq x$
 - $h = \lceil \log_2(n + 1) \rceil$

Node Number Properties



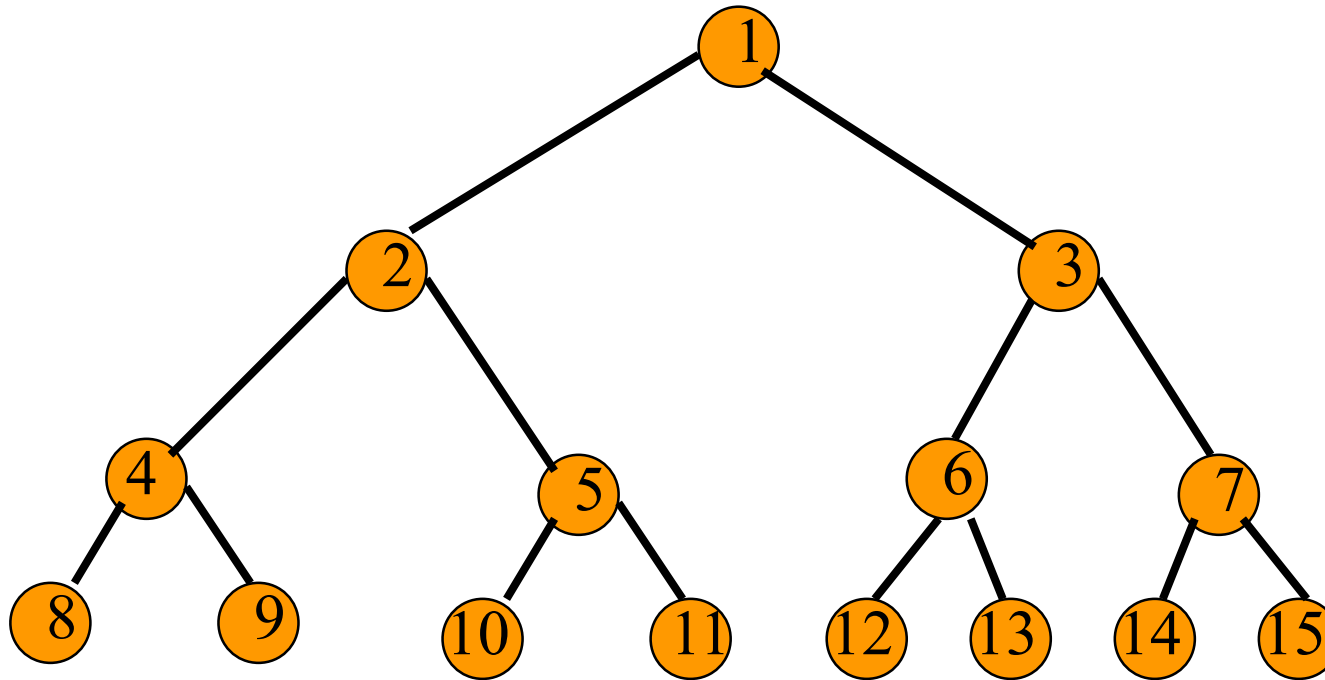
- Parent of node i is node $i / 2$, unless $i = 1$.
- Node 1 is the root and has no parent.

Node Number Properties (Cont.)



- Left child of node i is node $2i$, unless $2i > n$, where n is the number of nodes.
- If $2i > n$, node i has no left child.

Node Number Properties (Cont.)



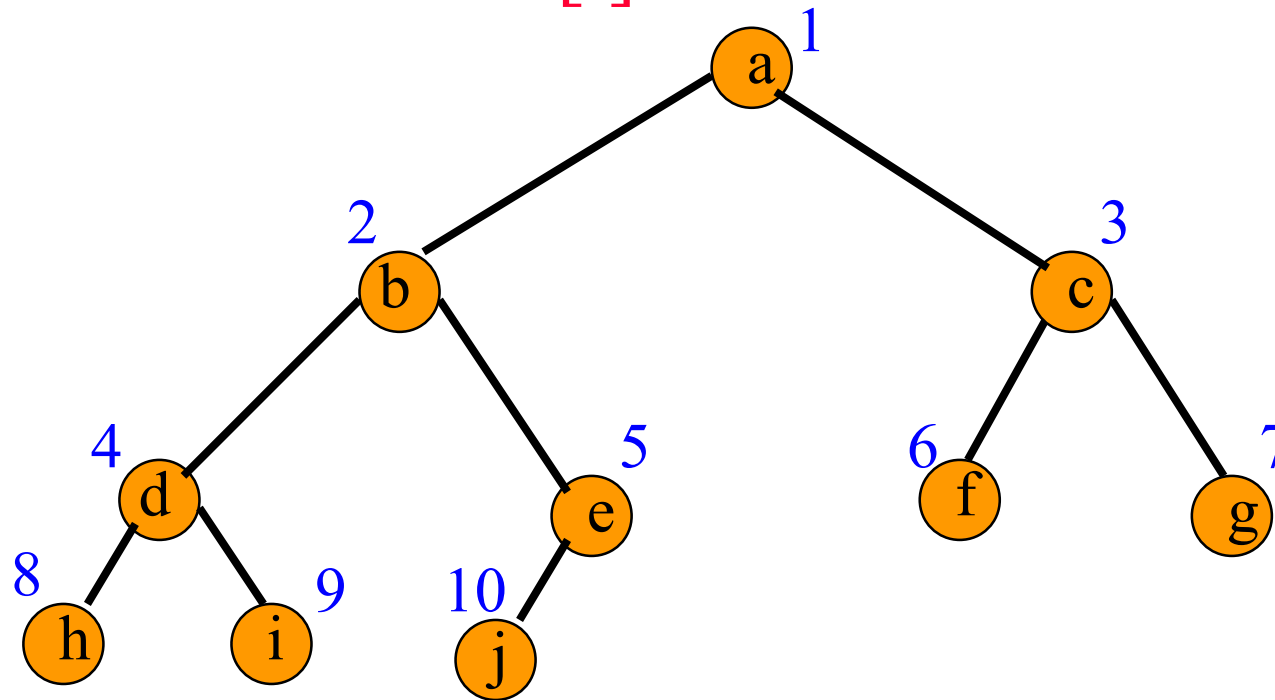
- Right child of node i is node $2i+1$, unless $2i+1 > n$, where n is the number of nodes.
- If $2i+1 > n$, node i has no right child.

Binary Tree Representation

- Array representation.
- Linked representation.

Array Representation

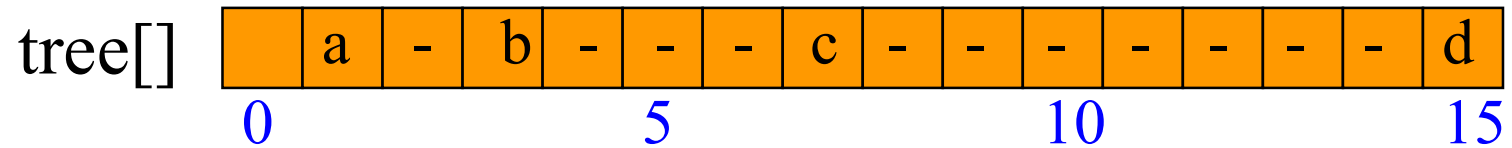
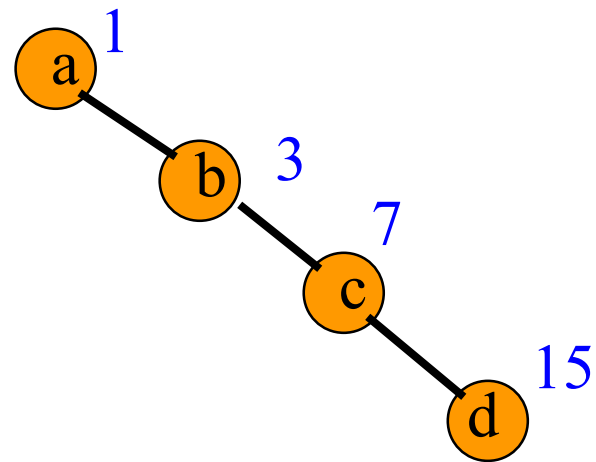
- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in $\text{tree}[i]$.



$\text{tree}[]$

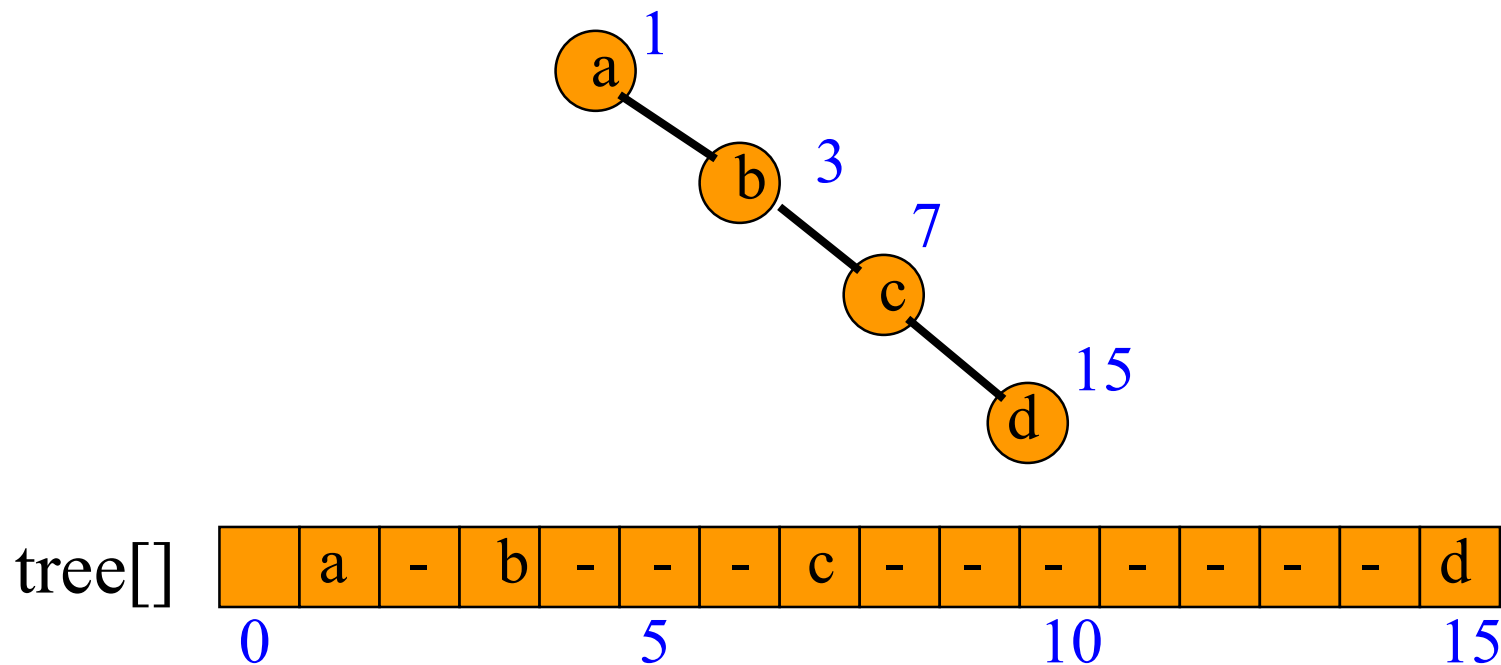
	a	b	c	d	e	f	g	h	i	j
0					5					10

Right-Skewed Binary Tree



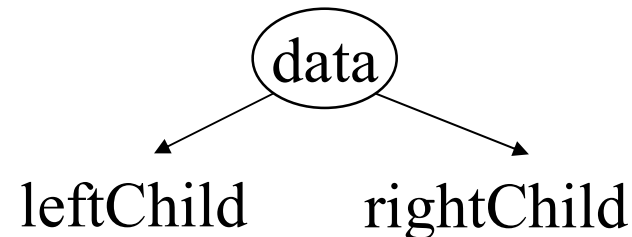
Disadvantages of Array Representation

- While the array representation appears to be good for complete binary trees, it is wasteful for many other binary trees



Linked Representation

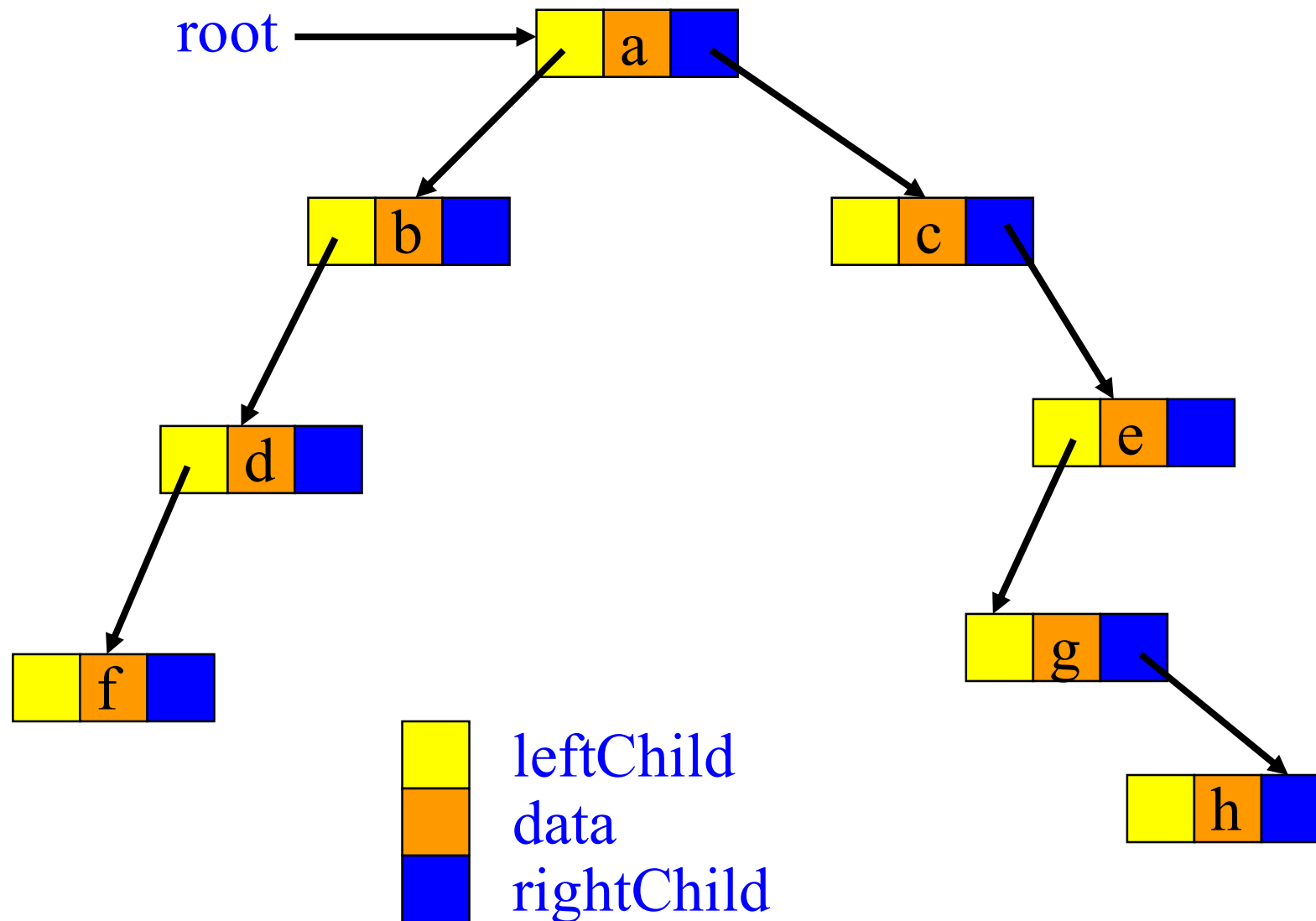
- Each binary tree node is represented as an object whose data type is `TreeNode`.
- The space required by an n node binary tree is $n * (\text{space required by one node})$



TreeNode Class

```
template <class T>
class TreeNode
{
    T data;
    TreeNode<T> *leftChild,
                *rightChild;
    TreeNode()
        {leftChild = rightChild = NULL;}
    // other constructors come here
};
```

Linked Representation Example



Binary Tree Traversal

- Arithmetic Expressions
- Inorder Traversal
- Preorder Traversal
- Postorder Traversal
- Level-order Traversal

Arithmetic Expressions

- $(a + b) * (c + d) + e - f/g * h + 3.25$
- Expressions comprise three kinds of entities.
 - Operators (+, -, /, *).
 - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).
 - Delimiters ((,)).

Operator Degree

- Number of operands that the operator requires.
- Binary operator requires two operands.
 - $a + b$
 - c / d
 - $e - f$
- Unary operator requires one operand.
 - $+ g$
 - $- h$

Infix Form

- Normal way to write an expression.
- Binary operators come **in** between their left and right operands.
 - $a * b$
 - $a + b * c$
 - $a * b / c$
 - $(a + b) * (c + d) + e - f/g * h + 3.25$

Operator Priorities

- How do you figure out the operands of an operator?
 - $a + b * c$
 - $a * b + c / d$
- This is done by assigning operator priorities.
 - $\text{priority}(*) = \text{priority}(/) > \text{priority}(+) = \text{priority}(-)$
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
 - $a + b - c$
 - $a * b / c / d$

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
 - $(a + b) * (c - d) / (e - f)$

Infix Expression Is Hard to Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier for a computer to evaluate expressions that are in these forms.

Postfix Form

- The postfix form of a variable or constant is the same as its infix form.
 - $a, b, 3.25$
- The relative order of operands is the same in infix and postfix forms.
- Operators come immediately **after** the postfix form of their operands.
 - Infix = $a + b$
 - Postfix = $ab+$

Postfix Examples

- Infix = $a + b * c$
- Postfix = $a b c * +$
- Infix = $a * b + c$
- Postfix = $a b * c +$
- Infix = $(a + b) * (c - d) / (e + f)$
- Postfix = $a b + c d - * e f + /$

Unary Operators

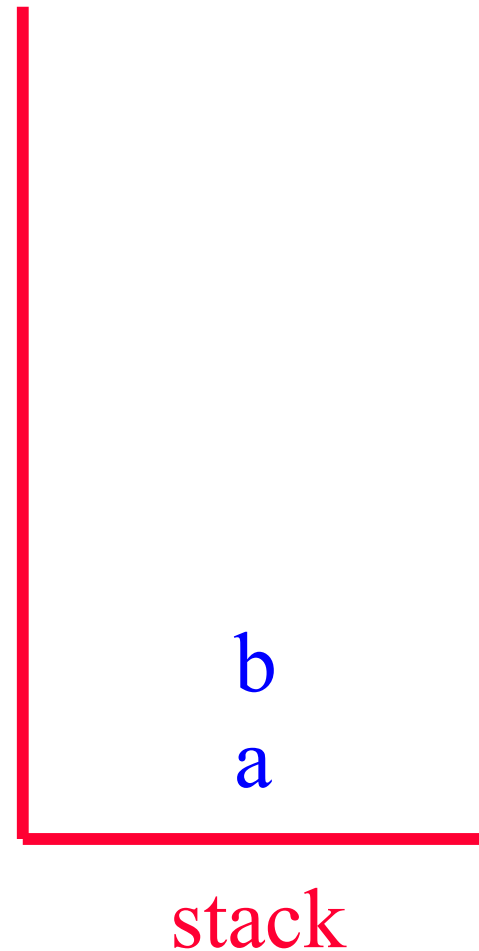
- Replace with new symbols.
 - $+ a \Rightarrow a @$
 - $+ a + b \Rightarrow a @ b +$
 - $- a \Rightarrow a ?$
 - $- a - b \Rightarrow a ? b -$

Postfix Evaluation

- Scan postfix expression from left to right pushing operands on to a stack.
- When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
- This works because, in postfix, operators come immediately after their operands.

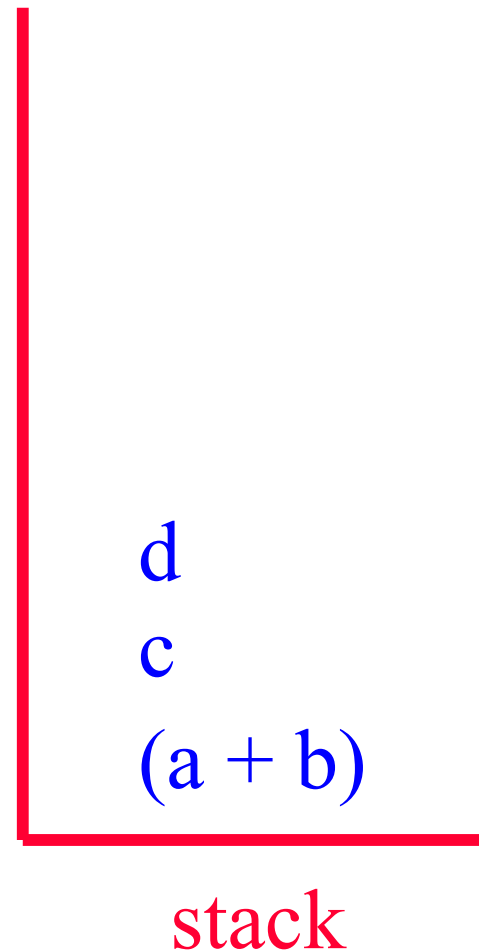
Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$



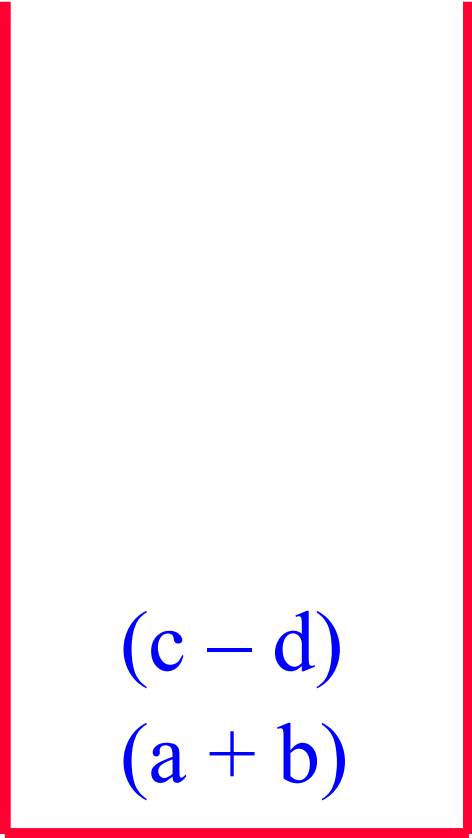
Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$
- $a\ b +\ c\ d -\ *\ e\ f +\ /$



Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$

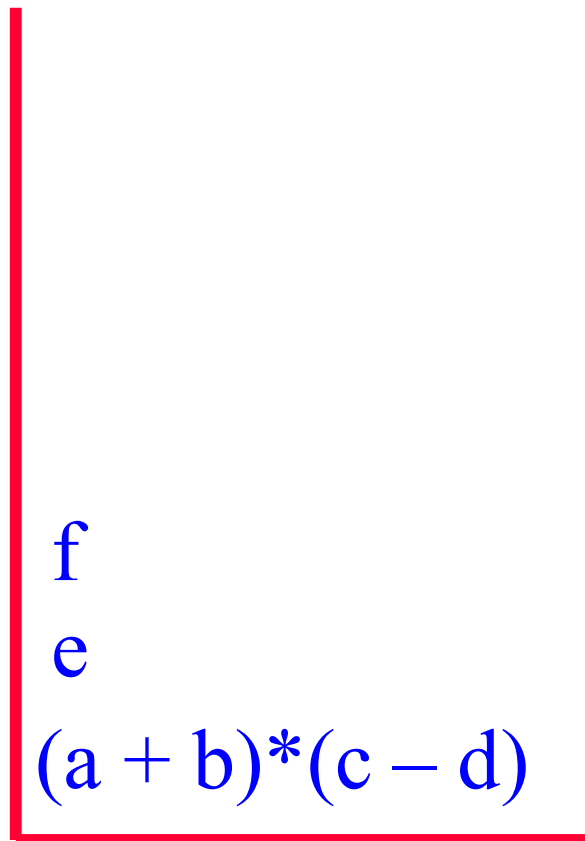


$(c - d)$
 $(a + b)$

stack

Postfix Evaluation

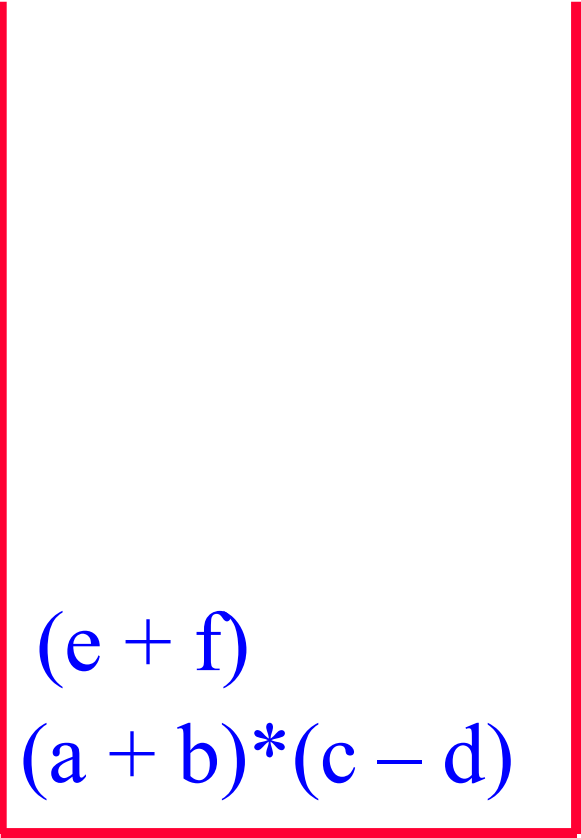
- $(a + b) * (c - d) / (e + f)$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$
- $a \ b \ + \ c \ d \ - \ * \ e \ f \ + \ /$



stack

Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a\ b +\ c\ d -\ * e\ f +\ /$
- $a\ b +\ c\ d -\ * e\ f +\ /$
- $a\ b +\ c\ d -\ * e\ f +\ /$
- $a\ b +\ c\ d -\ * e\ f +\ /$
- $a\ b +\ c\ d -\ * e\ f +\ /$
- $a\ b +\ c\ d -\ * e\ f +\ /$



$(e + f)$
 $(a + b) * (c - d)$

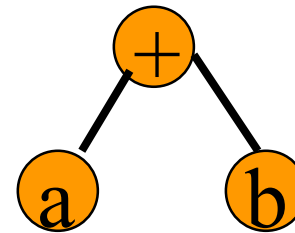
stack

Prefix Form

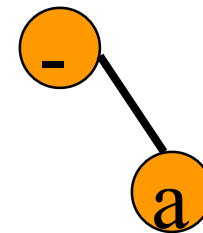
- The prefix form of a variable or constant is the same as its infix form.
 - $a, b, 3.25$
- The relative order of operands is the same in infix and prefix forms.
- Operators come immediately **before** the prefix form of their operands.
 - Infix = $a + b$
 - Postfix = $ab+$
 - Prefix = $+ab$

Binary Tree Form

- $a + b$

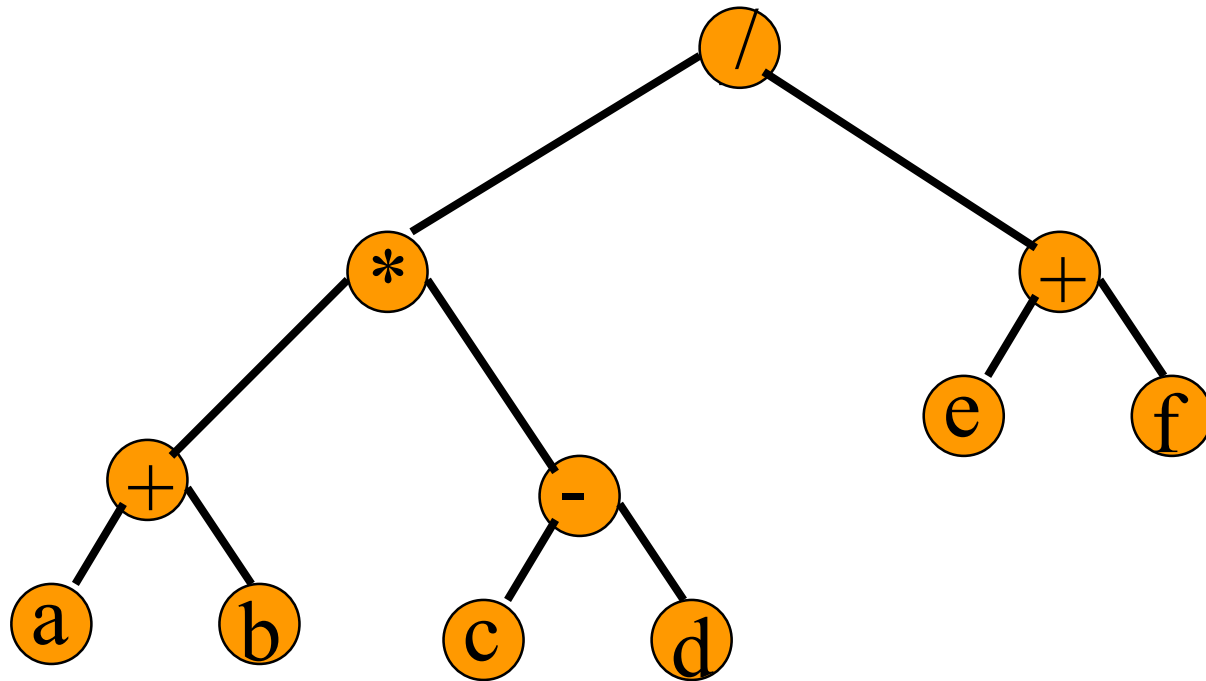


- $- a$



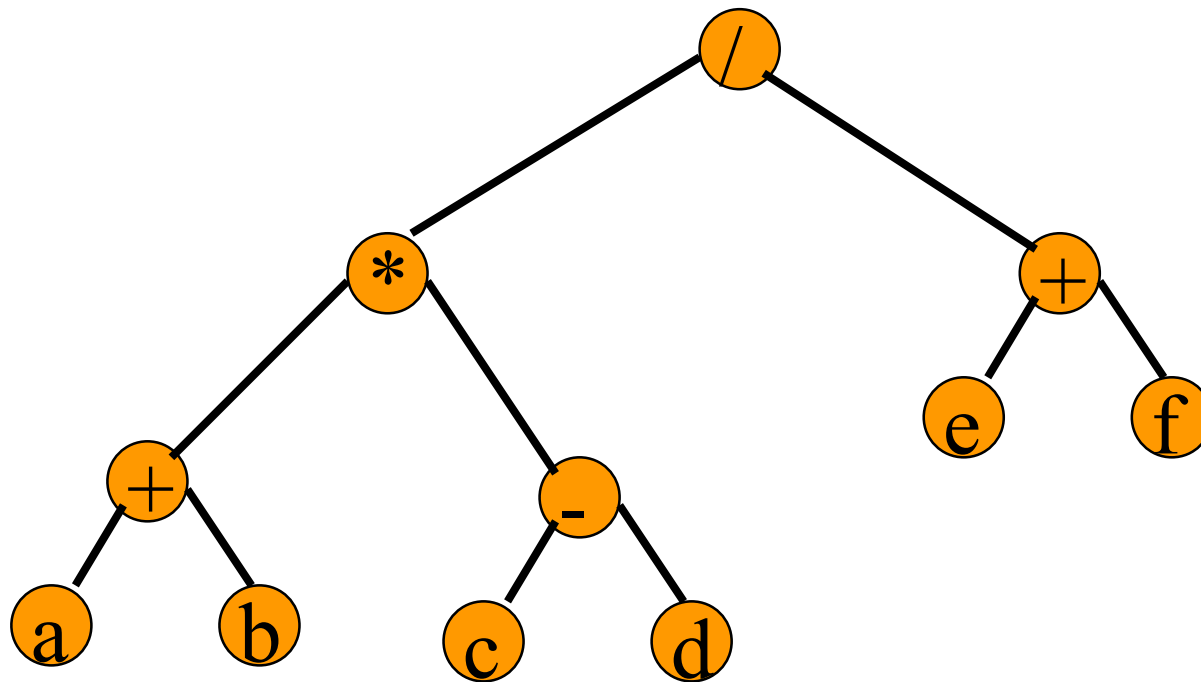
Binary Tree Form

- $(a + b) * (c - d) / (e + f)$



Merits of Binary Tree Form

- Left and right operands are easy to visualize.
- Simple recursive evaluation of expression.



Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

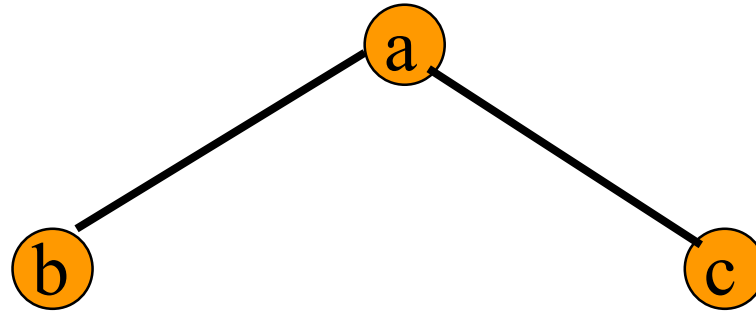
Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder
- Level order

Preorder Traversal

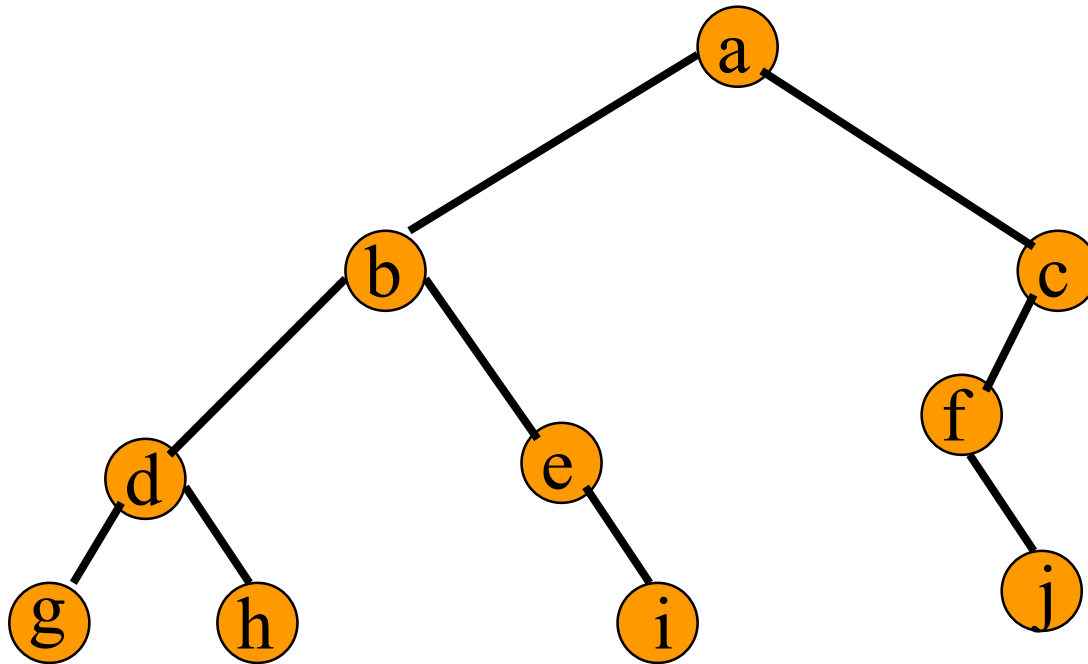
```
template <class T>
void PreOrder (TreeNode<T> *t)
{
    if (t != NULL)
    {
        Visit(t);
        PreOrder(t->leftChild);
        PreOrder(t->rightChild);
    }
}
```


Preorder Example (Visit = print)



a b c

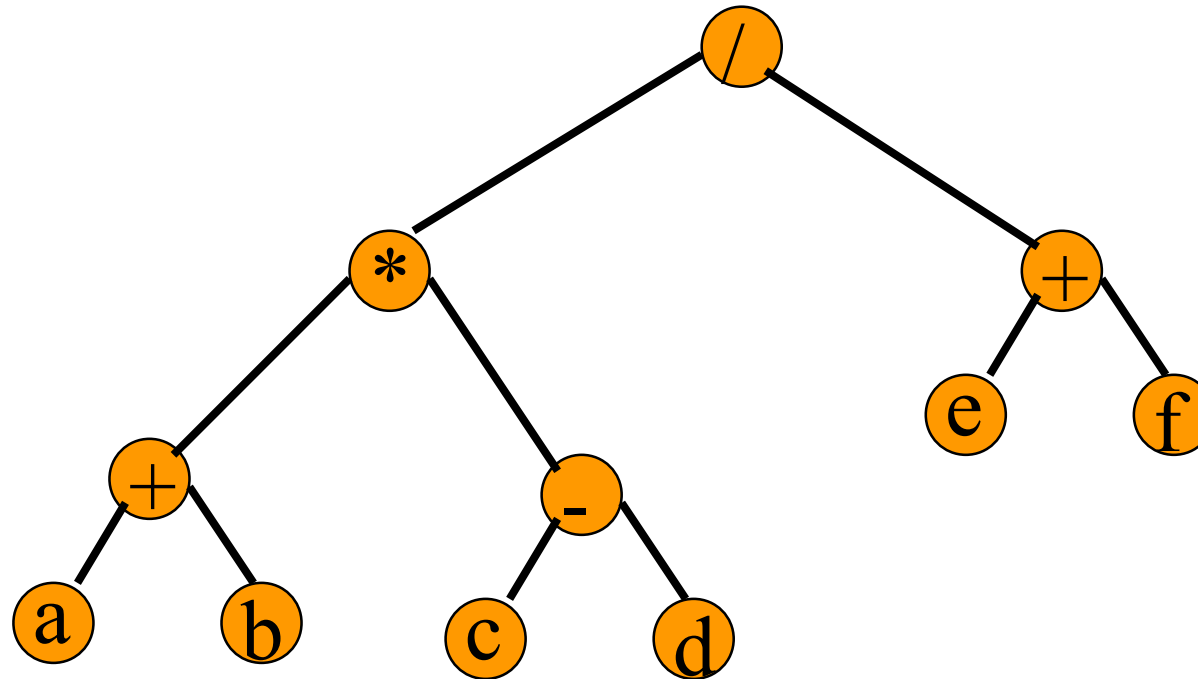
Preorder Example (Visit = print)



a b d g h e i c f j

```
template <class T>
void PreOrder(TreeNode<T> *t)
{
    if (t != NULL)
    {
        Visit(t);
        PreOrder(t->leftChild);
        PreOrder(t->rightChild);
    }
}
```

Preorder of Expression Tree



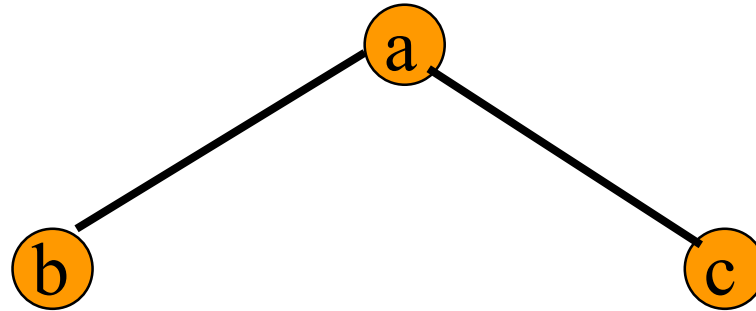
$/ * + a b - c d + e f$

Gives prefix form of expression!

Inorder Traversal

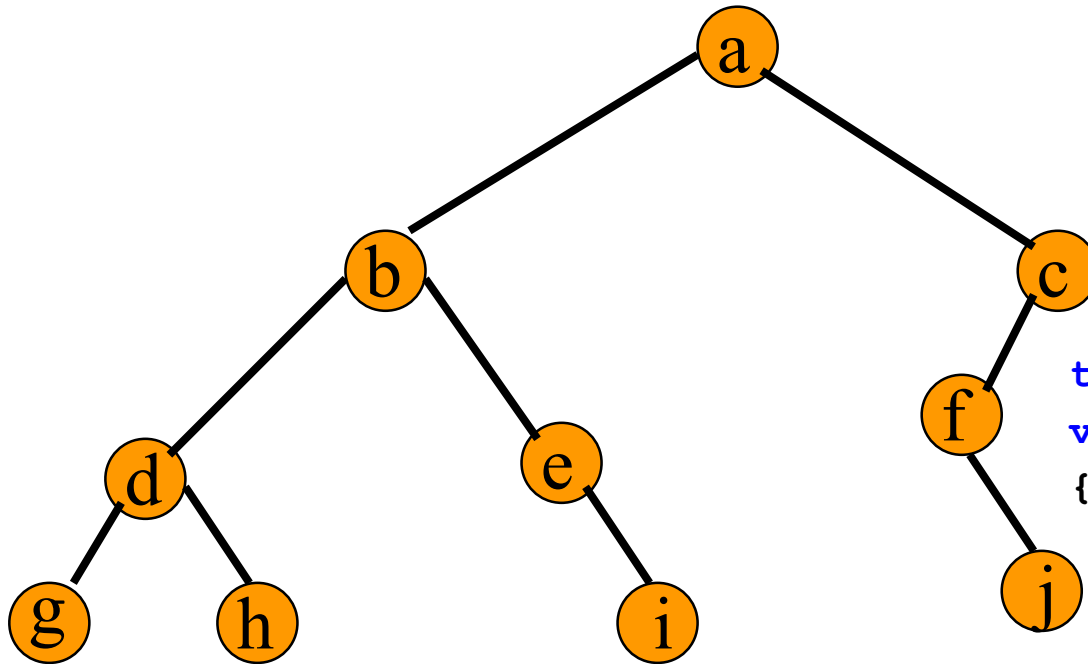
```
template <class T>
void InOrder (TreeNode<T> *t)
{
    if (t != NULL)
    {
        InOrder (t->leftChild) ;
        Visit (t) ;
        InOrder (t->rightChild) ;
    }
}
```

Inorder Example (Visit = print)



b a c

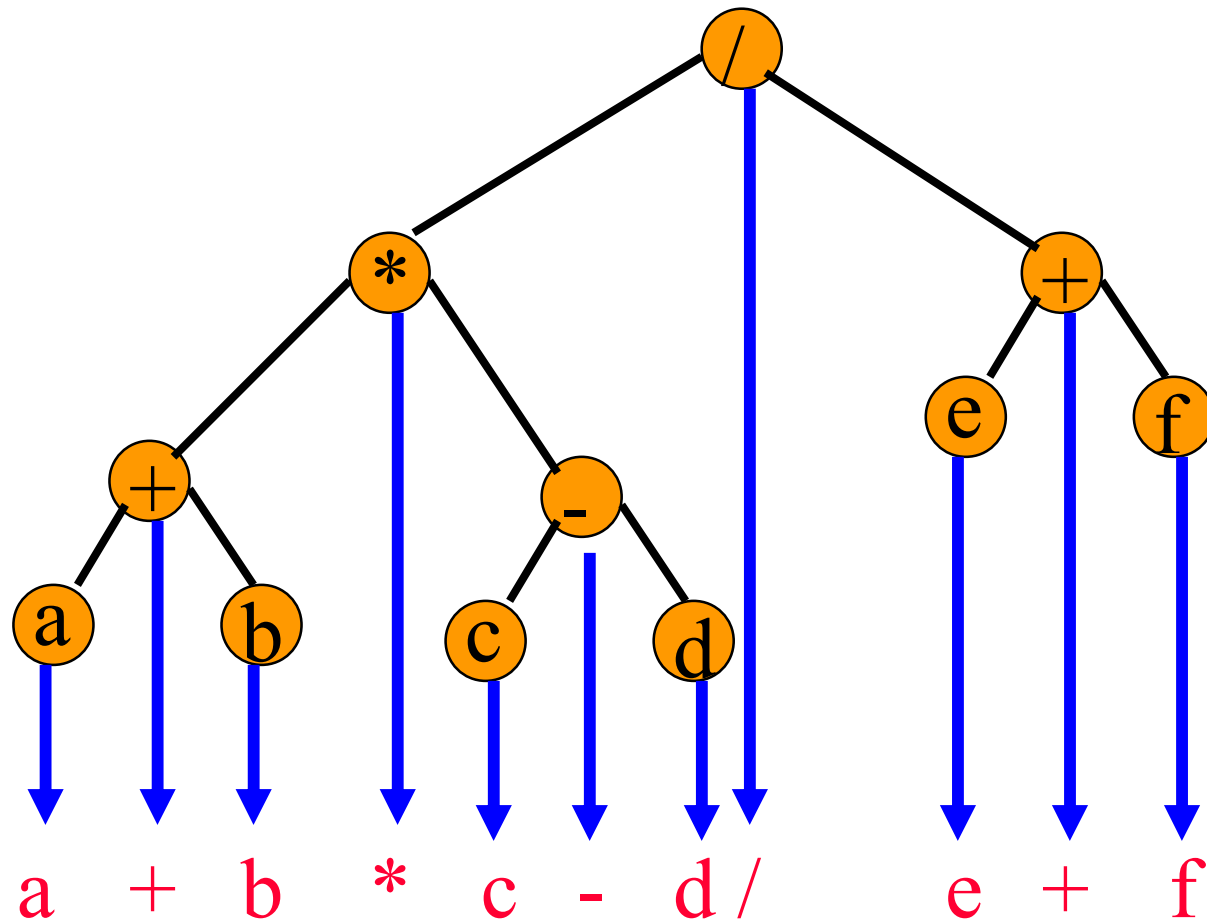
Inorder Example (Visit = print)



```
template <class T>
void InOrder (TreeNode<T> *t)
{
    if (t != NULL)
    {
        InOrder (t->leftChild);
        Visit(t);
        InOrder (t->rightChild);
    }
}
```

g d h b e i a f j c

Inorder of Expression Tree

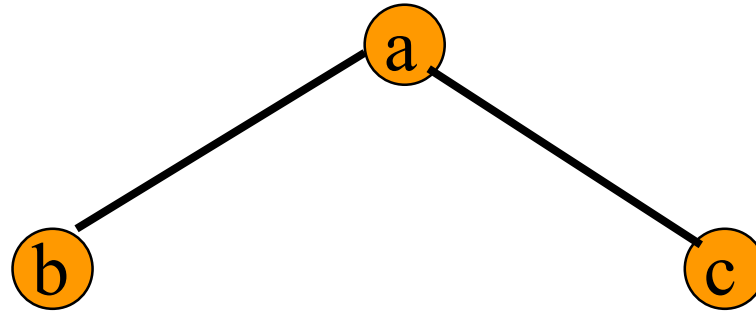


Gives infix form of expression (without parentheses)!

Postorder Traversal

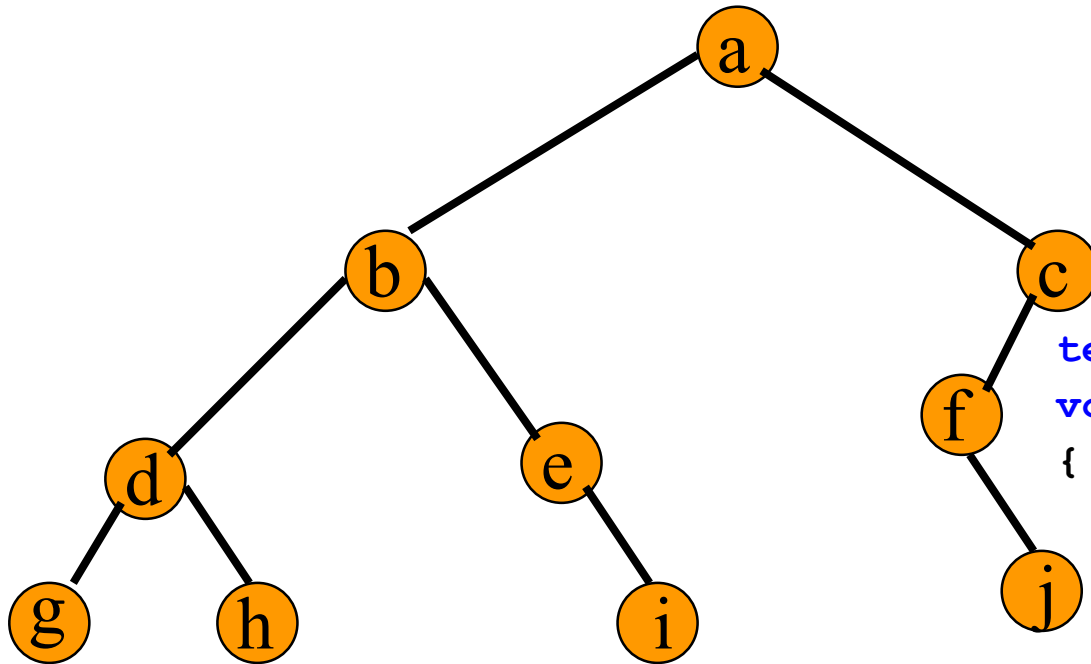
```
template <class T>
void PostOrder (TreeNode<T> *t)
{
    if (t != NULL)
    {
        PostOrder (t->leftChild);
        PostOrder (t->rightChild);
        Visit (t);
    }
}
```


Postorder Example (Visit = print)



b c a

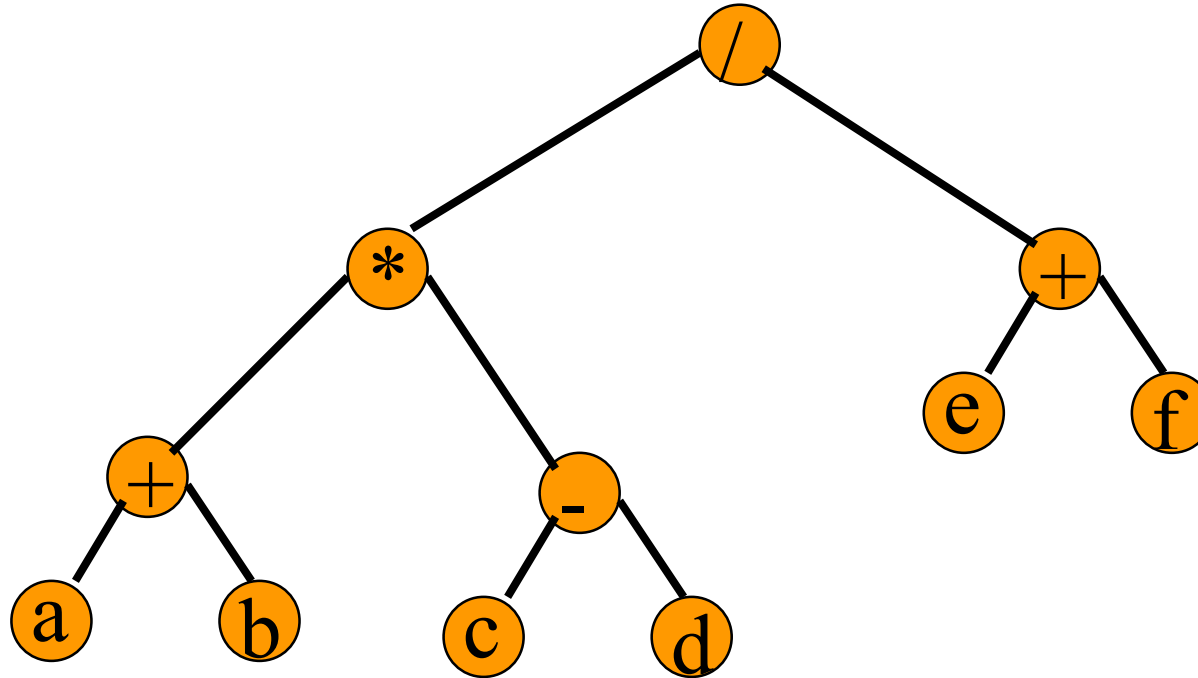
Postorder Example (Visit = print)



```
template <class T>
void PostOrder(TreeNode<T> *t)
{
    if (t != NULL)
    {
        PostOrder(t->leftChild);
        PostOrder(t->rightChild);
        Visit(t);
    }
}
```

g h d i e b j f c a

Postorder of Expression Tree



$a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /\$

Gives postfix form of expression!

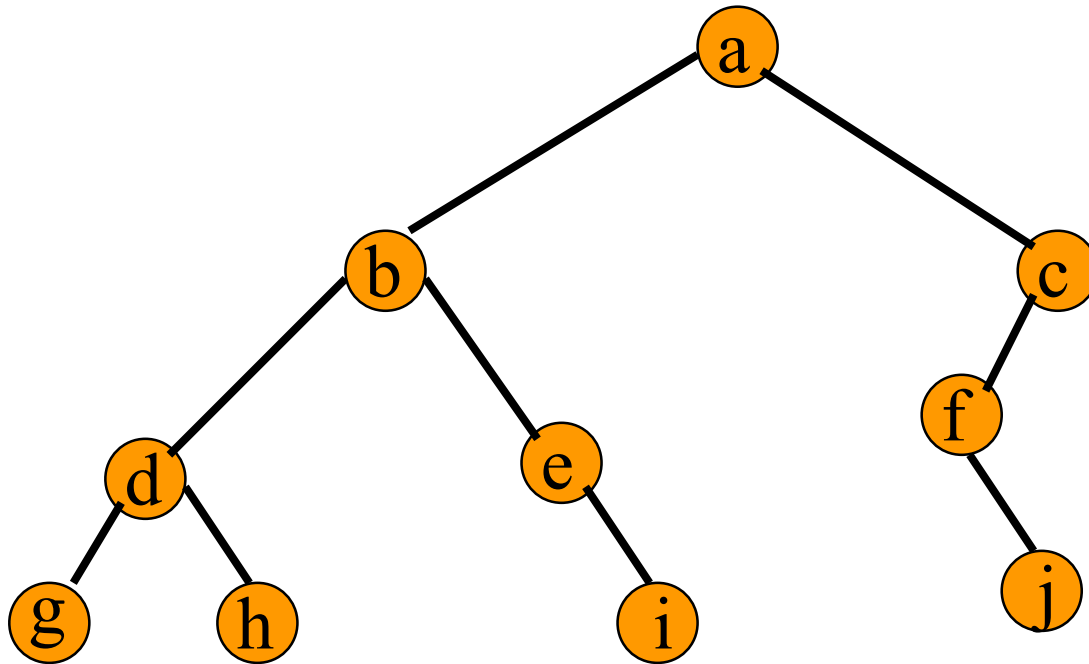
Level Order

- Visit the root first, then the root's left child, followed by the root's right child
- Continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node

Level Order (Cont.)

```
template <class T>
void Tree<T>::LevelOrder()
{
    Queue<TreeNode<T>*> q;
    TreeNode<T> * currentNode = root;
    while(currentNode){
        Visit(currentNode);
        if(currentNode->leftChild) q.Push(currentNode->leftChild);
        if(currentNode->rightChild) q.Push(currentNode->rightChild);
        if(q.IsEmpty()) return;
        currentNode = q.Front();
        q.Pop();
    }
}
```

Level-Order Example (Visit = print)



a b c d e f g h i j

```
template <class T>
void Tree<T>::LevelOrder()
{
    Queue<TreeNode<T>*> q;
    TreeNode<T> * currentNode = root;
    while(currentNode){
        Visit(currentNode);
        if(currentNode->leftChild)
            q.Push(currentNode->leftChild);
        if(currentNode->rightChild)
            q.Push(currentNode->rightChild);
        if(q.IsEmpty()) return;
        currentNode = q.Front();
        q.Pop();
    }
}
```

Homework #2

- Lemma 5.3: For any nonempty binary tree, T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$
 - Proof: Homework #2-1
- Homework #2-2
 - Implement and test Programs 5.1 – 5.7 in the textbook
- Homework을 제출할 필요는 없으나
중간/기말고사에 출제할 계획임