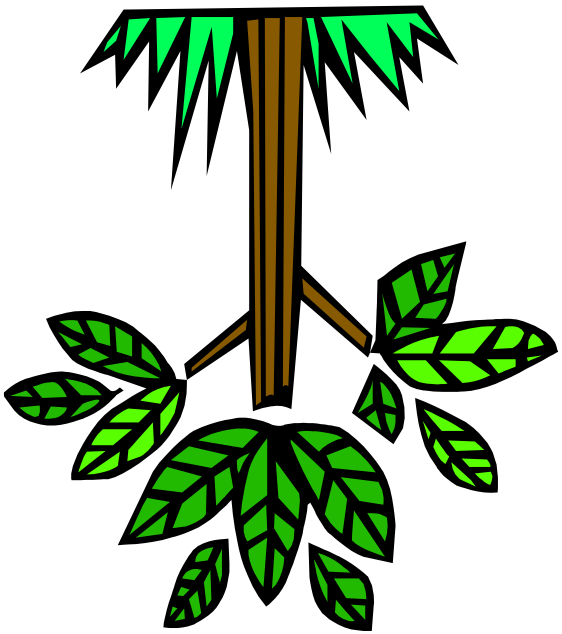
Data Structures - TREES

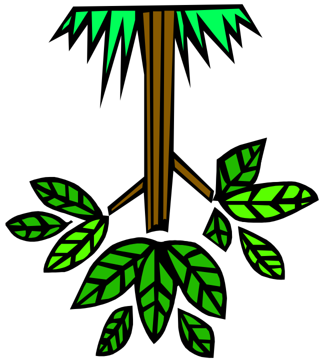
Introduction to Binary Trees

Why Trees?

- Limitations of
  - Arrays
  - Linked lists
  - Stacks
  - Queues
- LINEAR STORAGE

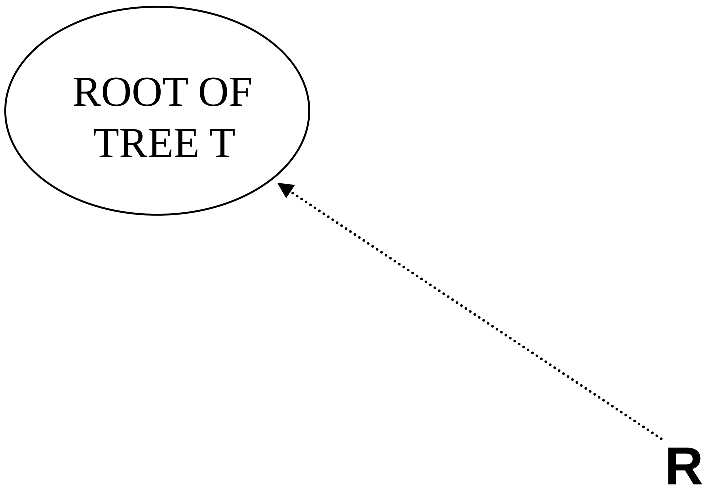


Trees: Recursive Definition

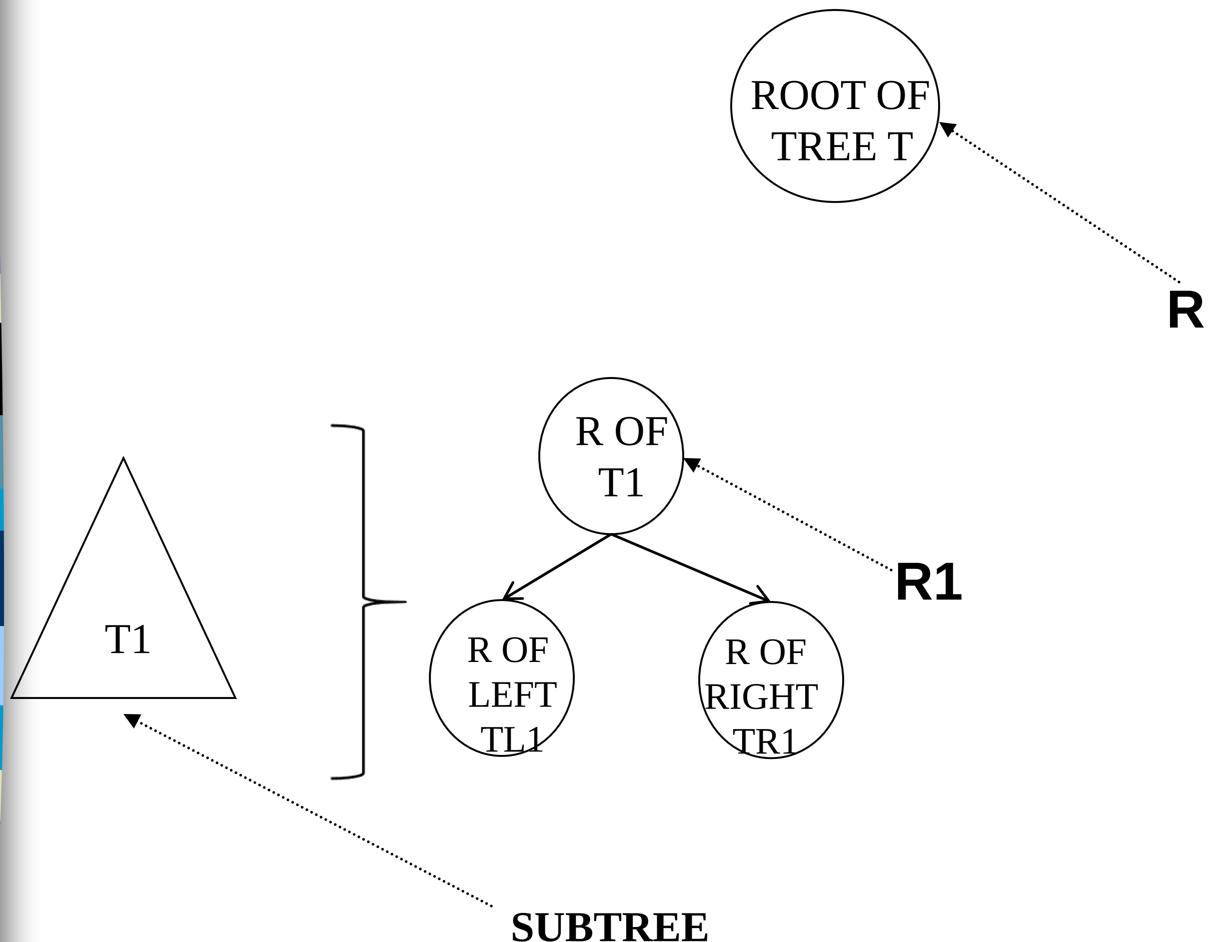


- A tree is a collection of nodes.
- The collection can be empty, or consist of a "root" node *R*.
- There is a "directed edge" from *R* to the root of each subtree. The root of each subtree is a "child" of *R*. *R* is the "parent" of each subtree root.

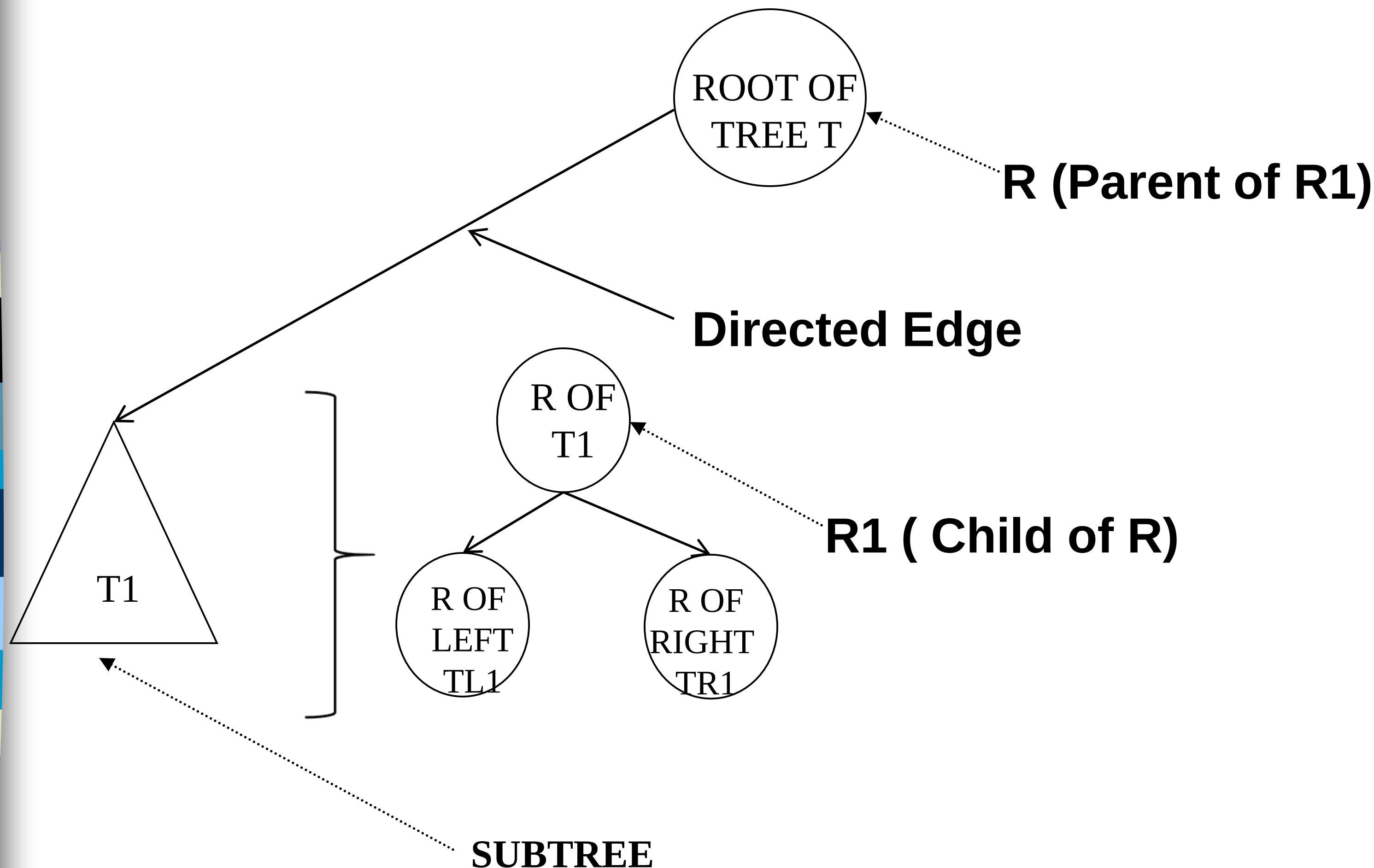
Trees: Recursive Definition (cont.)



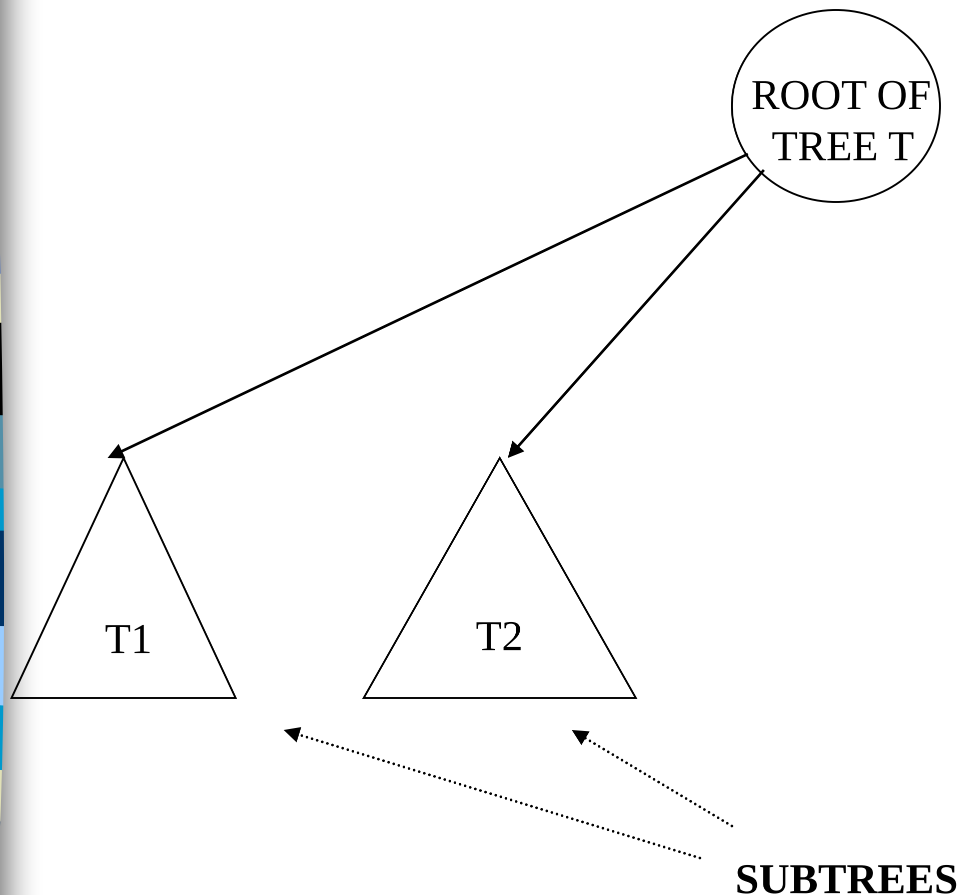
Trees: Recursive Definition (cont.)



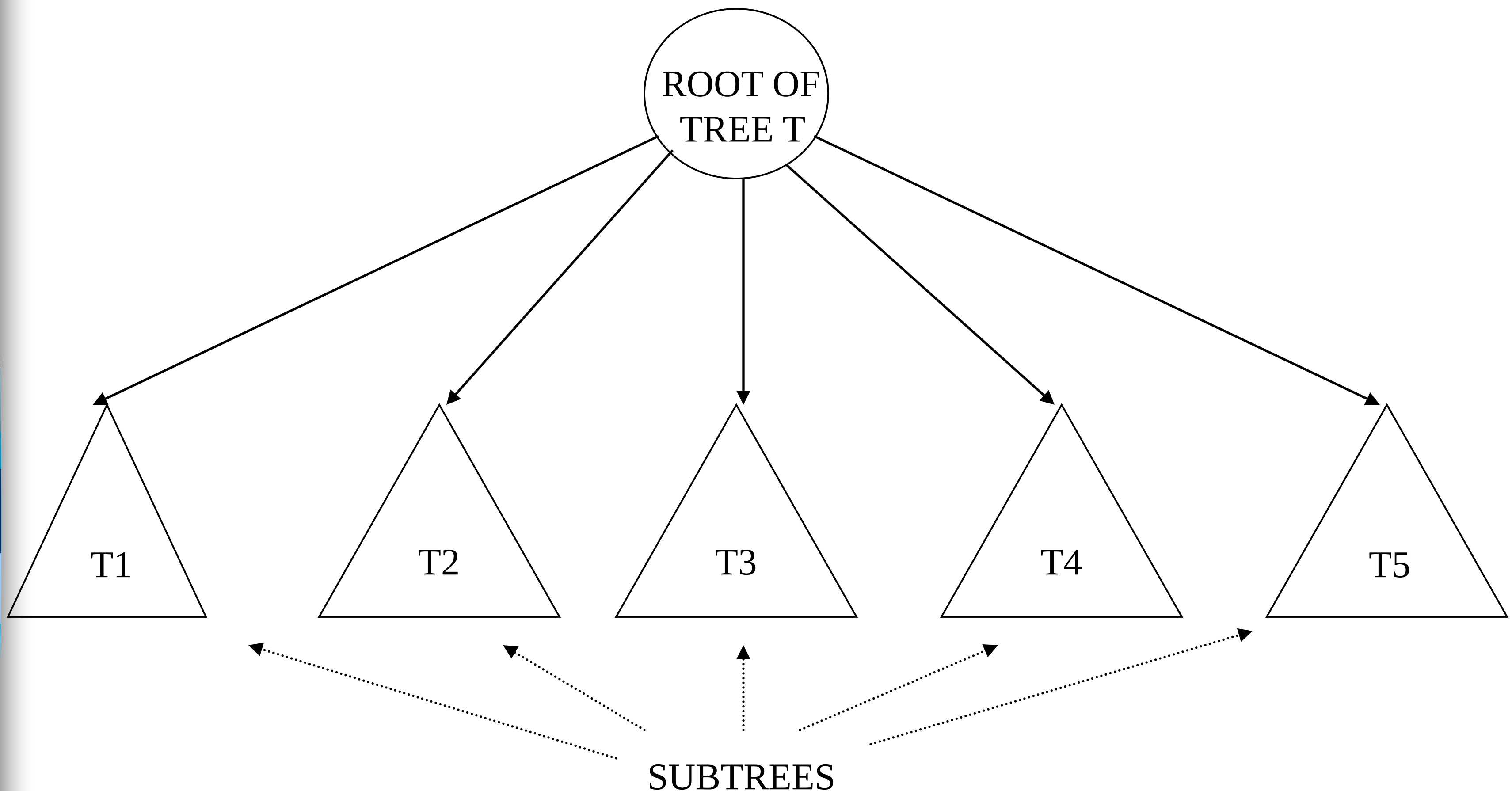
Trees: Recursive Definition (cont.)



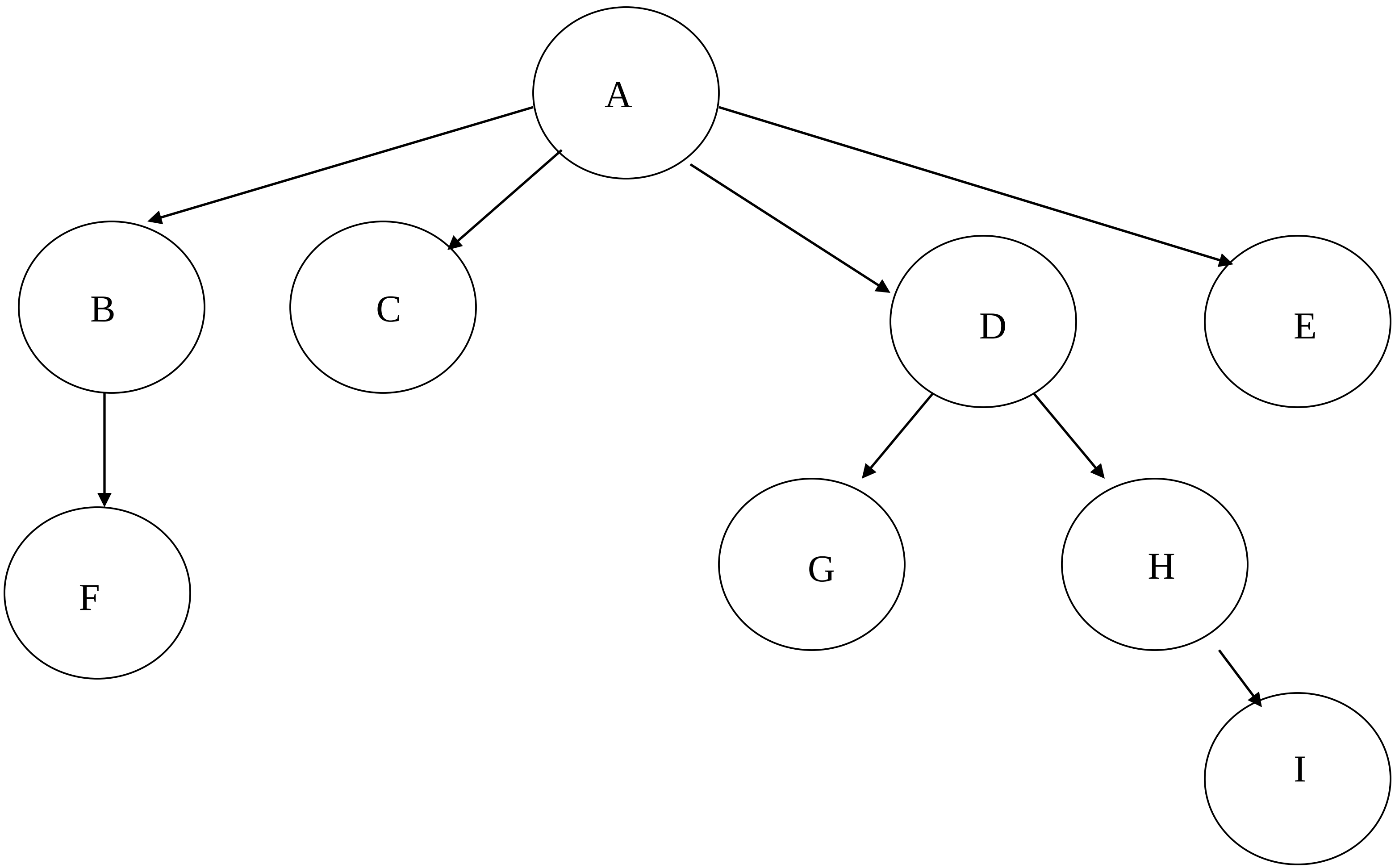
Trees: Recursive Definition (cont.)



Trees: Recursive Definition (cont.)



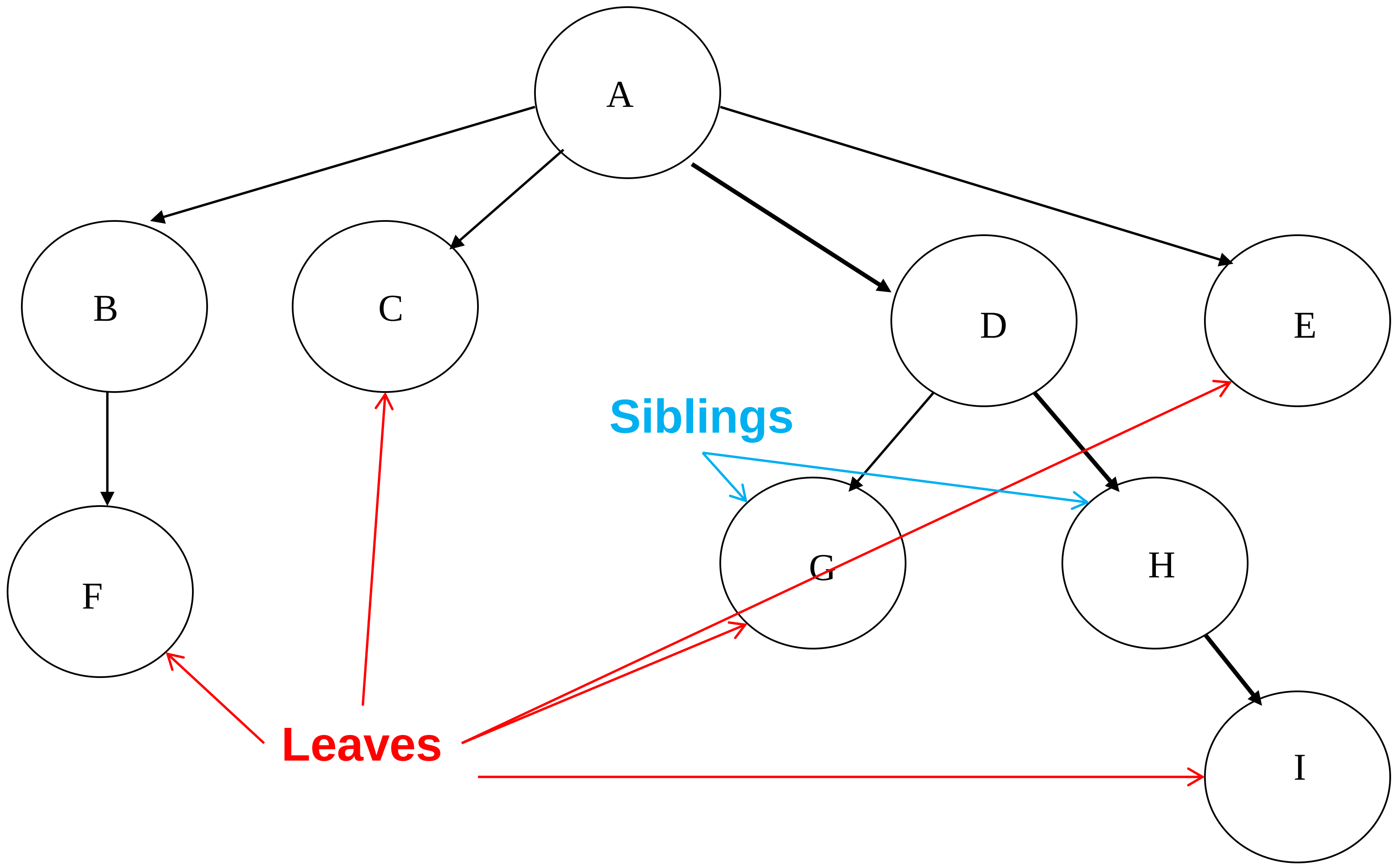
Trees: An Example



Trees: More Definitions

- Nodes with no children are leaves: (C,E,F,H,I).
- Nodes with the same parents are siblings: (B,C,D,E) and (G,H).
- A path from node *n* to node *m* is the sequence of directed edges from *n* to *m*.
- A length of a path is the number of edges in the path

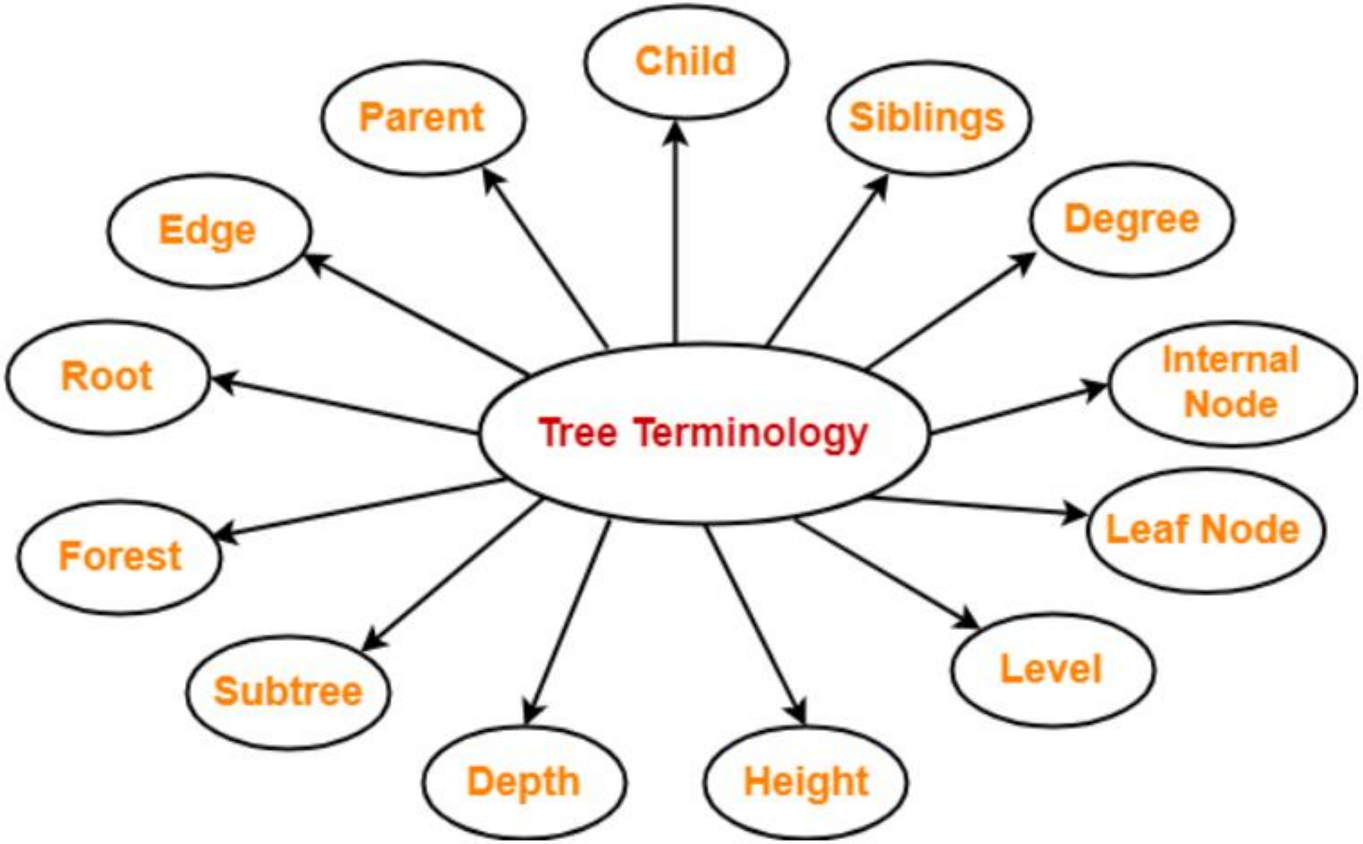
Trees: An Example



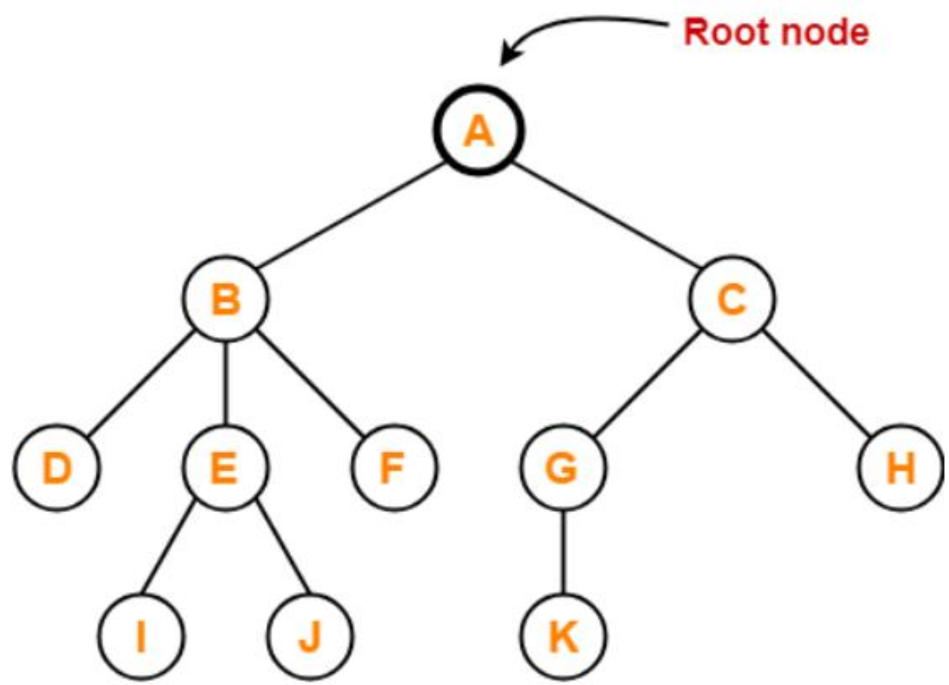
Trees: More Definitions (cont.)

- The level/depth of node *n* is the length of the path from the root to *n*. The level of the root is 0.
- The height/depth of a tree is equal to the maximum level of a node in the tree.
- The height of a node *n* is the length of the longest path from *n* to a leaf. The height of a leaf node is 0.
- The height of a tree is equal to the height of the root node.

Tree Terminology

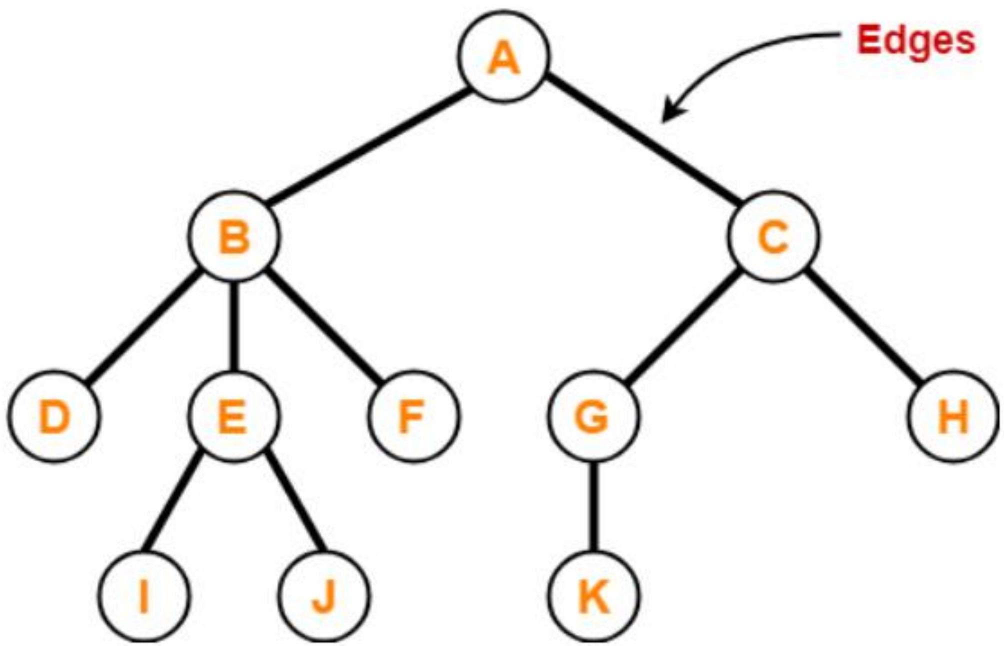


- **<u>Root-</u>**
- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



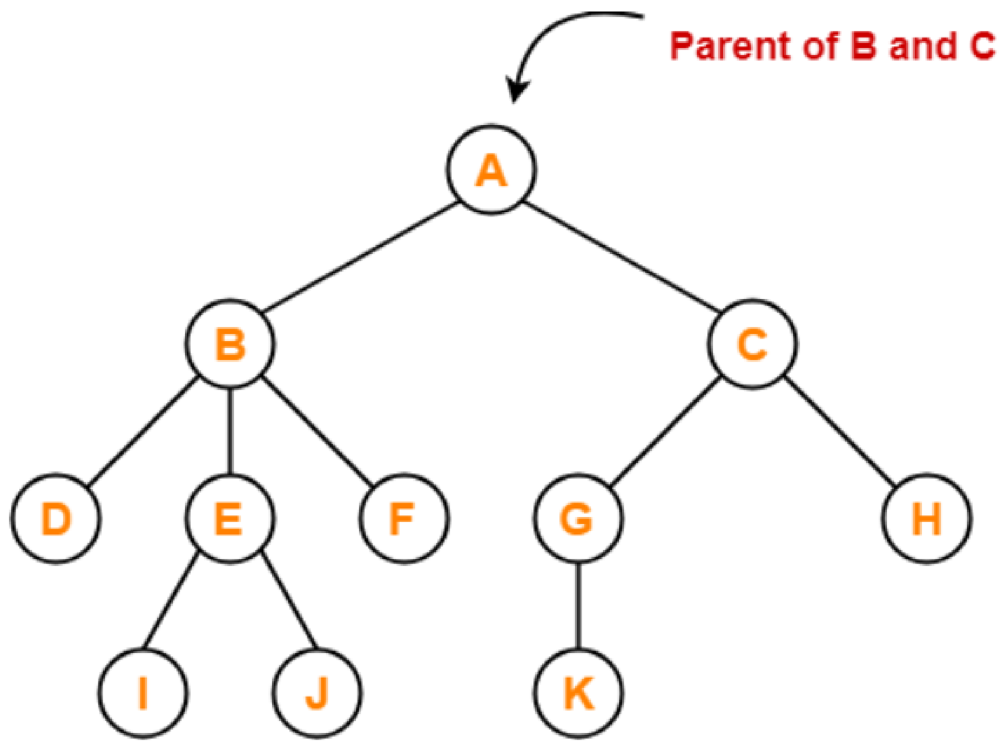
Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.



- **Parent-**
- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

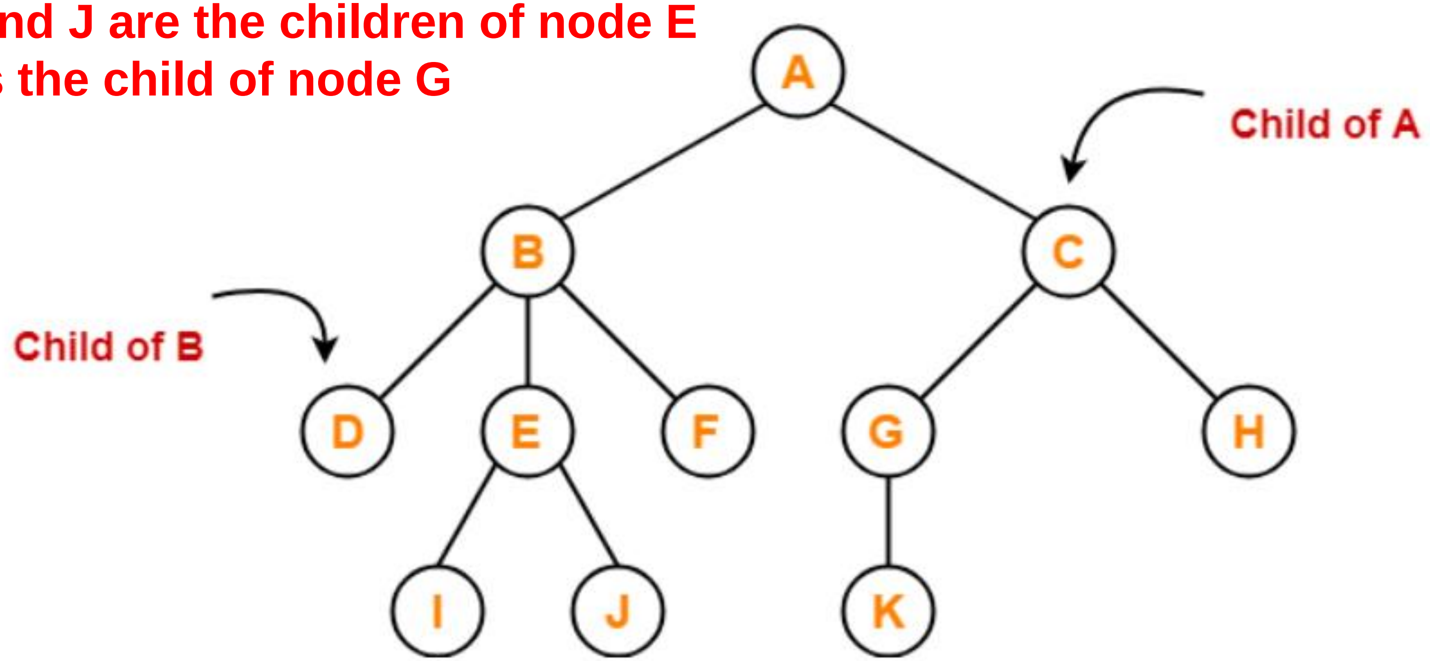
Here,
•Node A is the parent of nodes B and C
•Node B is the parent of nodes D, E and F
•Node C is the parent of nodes G and H
•Node E is the parent of nodes I and J
•Node G is the parent of node K



- **Child-**
- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

Here,

•Nodes B and C are the children of node A
•Nodes D, E and F are the children of node B
•Nodes G and H are the children of node C
•Nodes I and J are the children of node E
•Node K is the child of node G



- **Siblings-**
-
- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

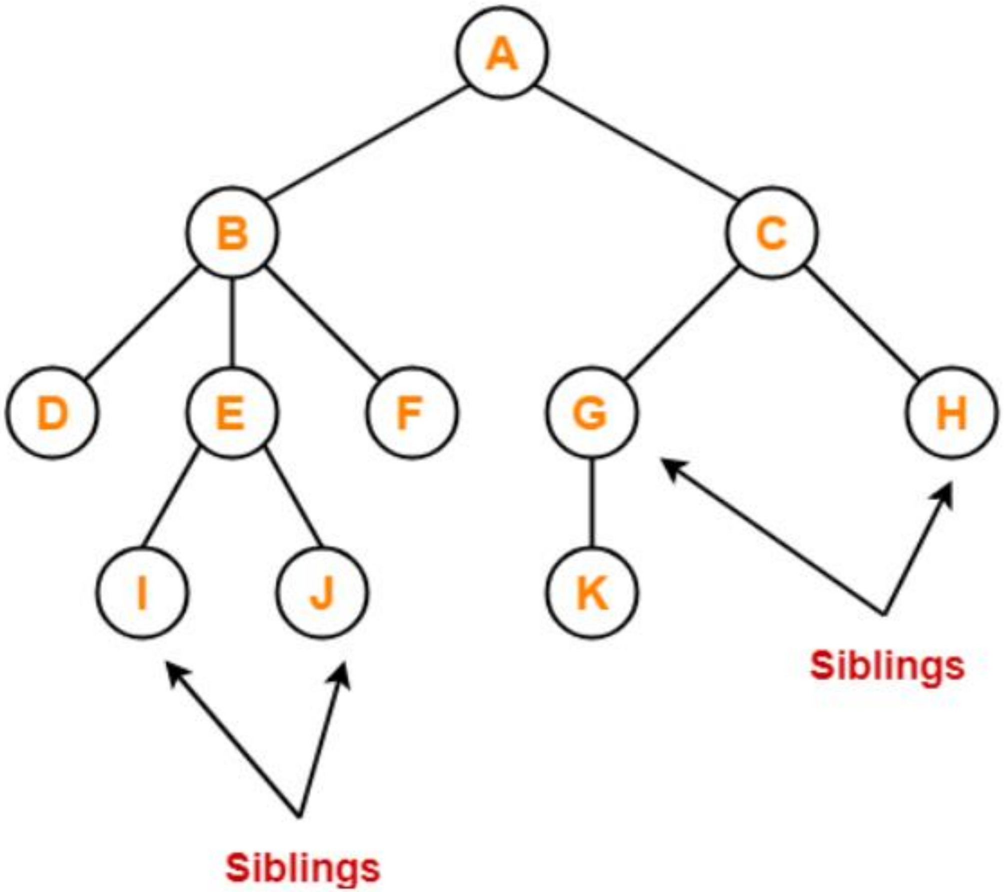
Here,

•Nodes B and C are siblings

•Nodes D, E and F are siblings

•Nodes G and H are siblings

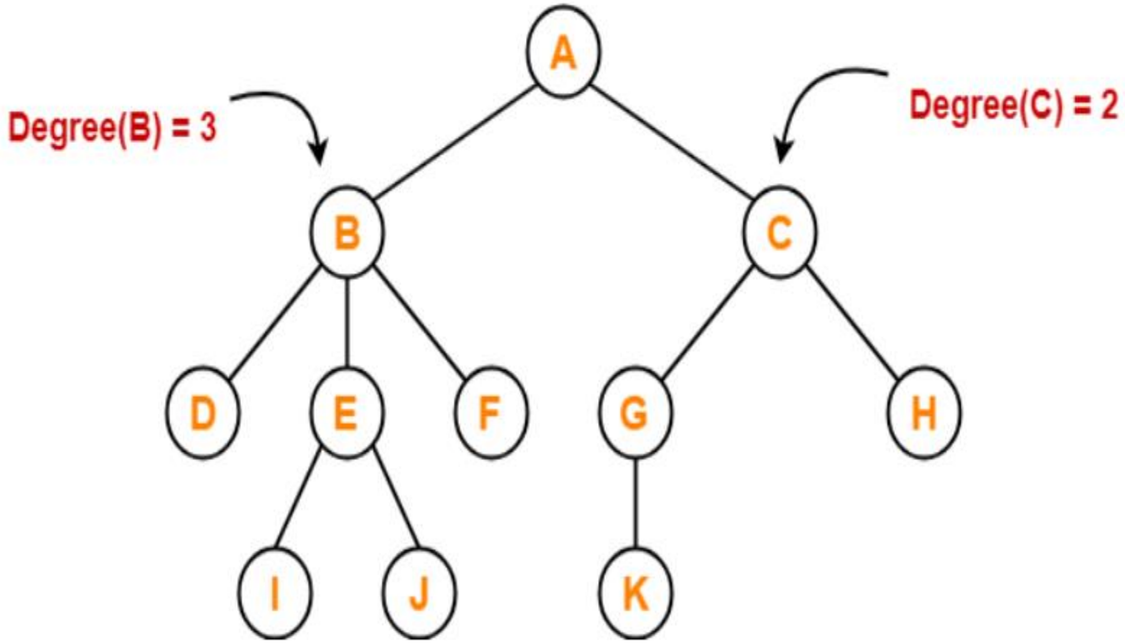
•Nodes I and J are siblings



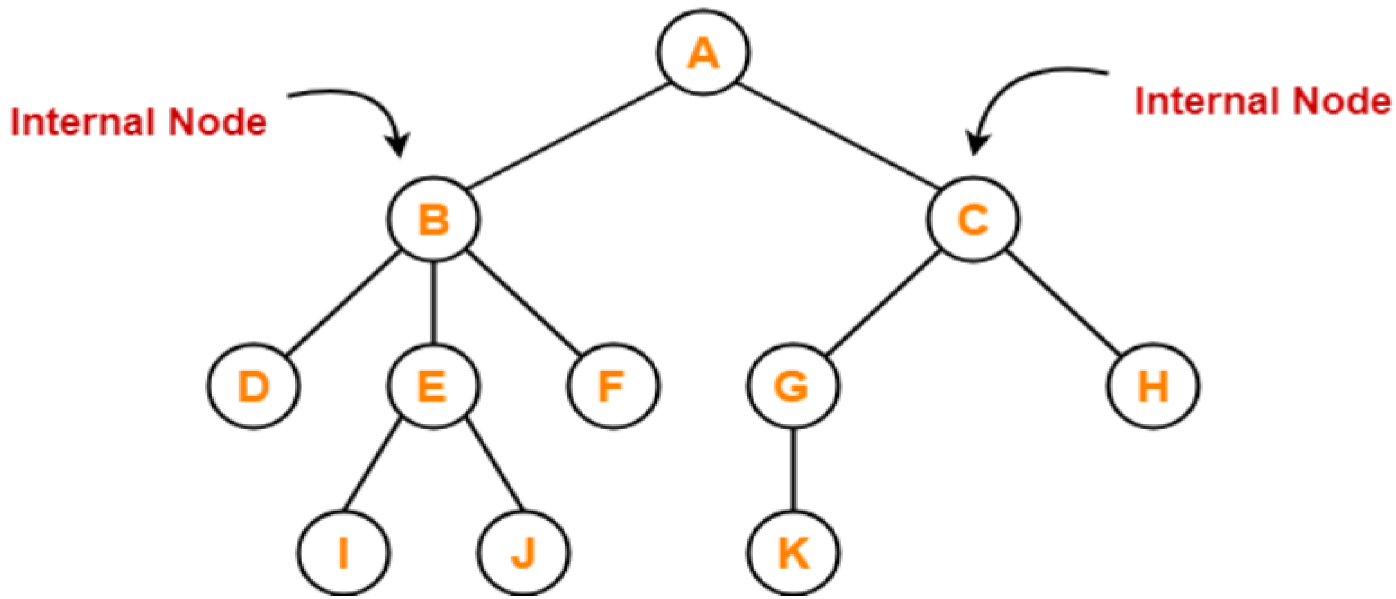
- **Degree-**
- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

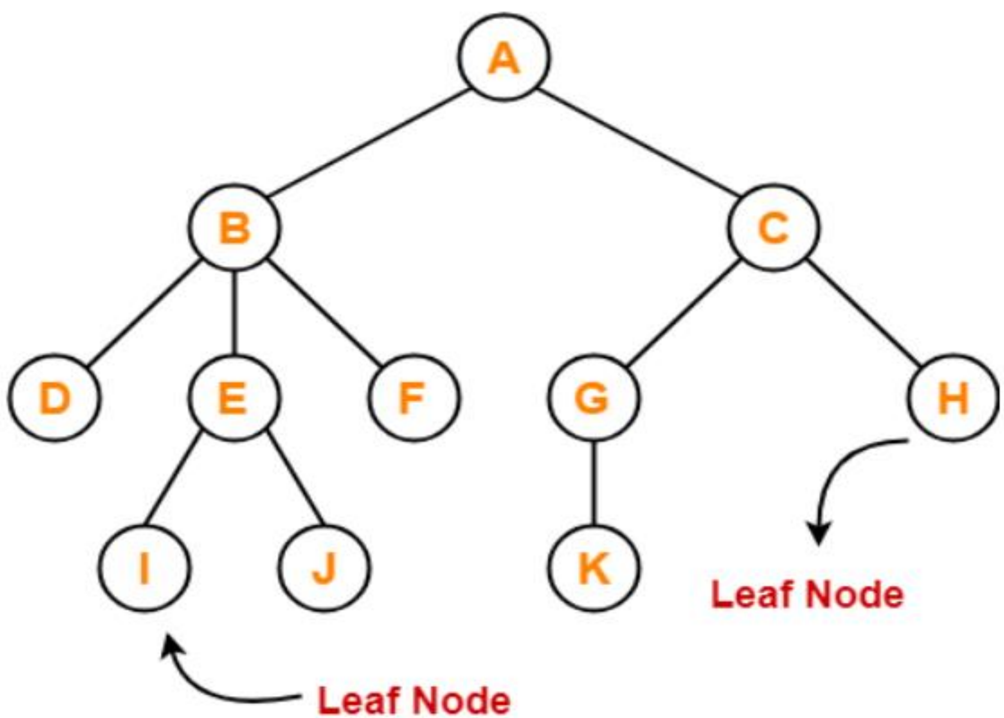


- **Internal Node-**
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



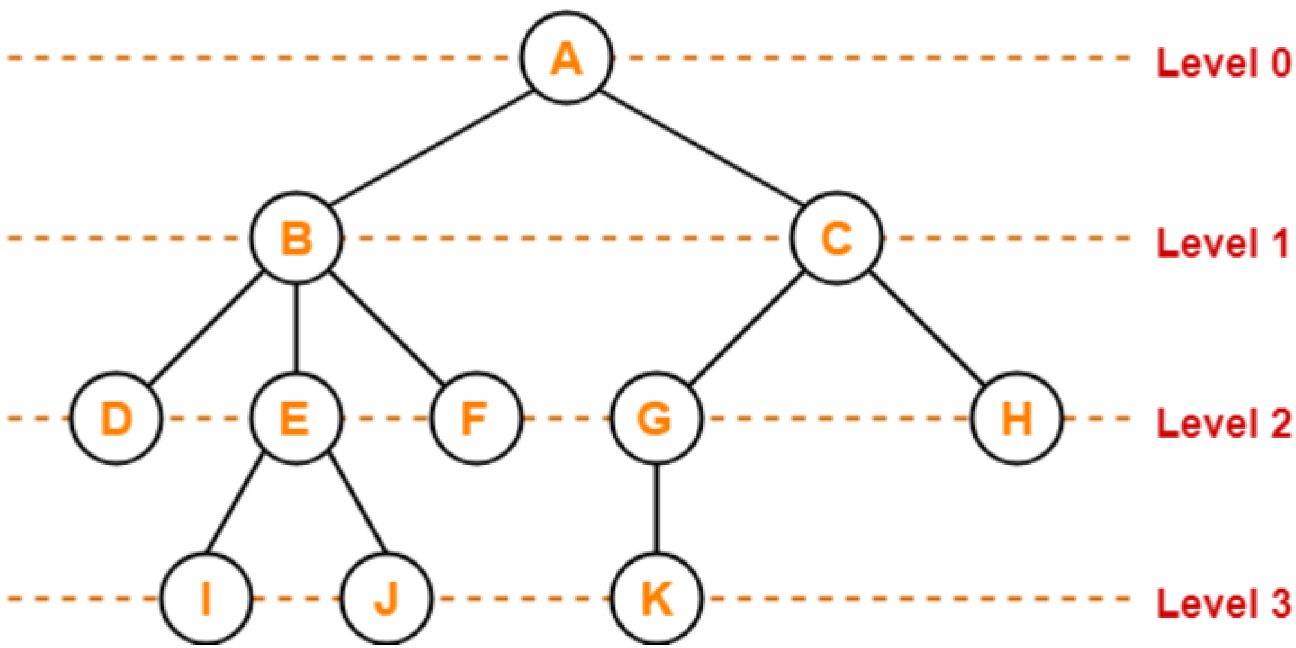
Here, nodes A, B, C, E and G are internal nodes.

- **Leaf Node-**
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



Here, nodes D, I, J, F, K and H are leaf nodes.

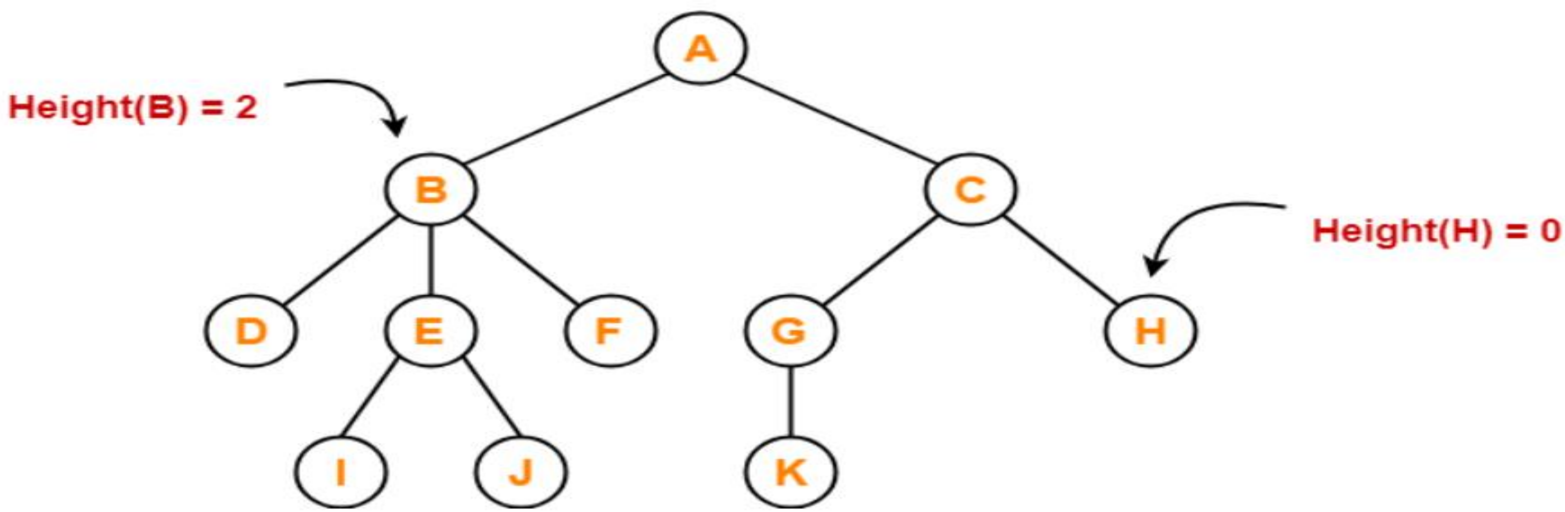
- **<u>Level-</u>**
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



Height-

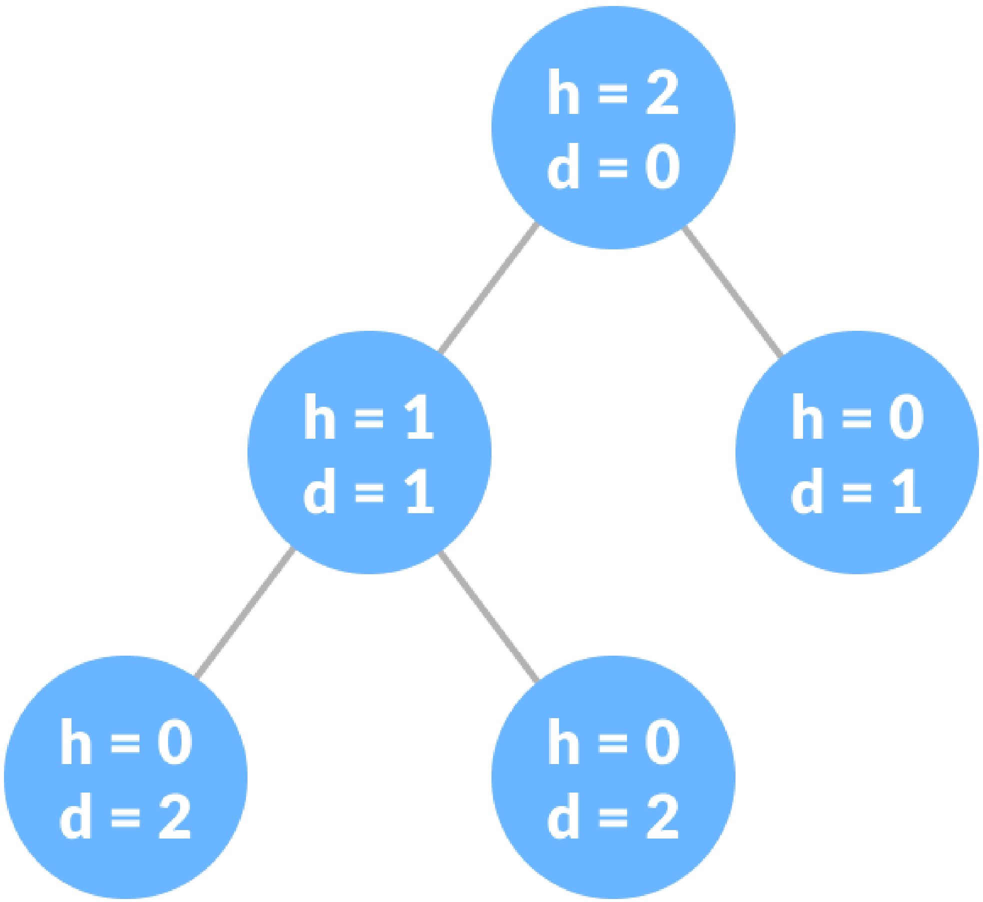
- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0

Here,
- **Height of node A = 3**
- **Height of node B = 2**
- **Height of node C = 2**
- **Height of node D = 0**
- **Height of node E = 1**
- **Height of node F = 0**
- **Height of node G = 1**
- **Height of node H = 0**
- **Height of node I = 0**
- **Height of node J = 0**
- **Height of node K = 0**



The height of a Tree is the height of the root node or the depth of the deepest node.

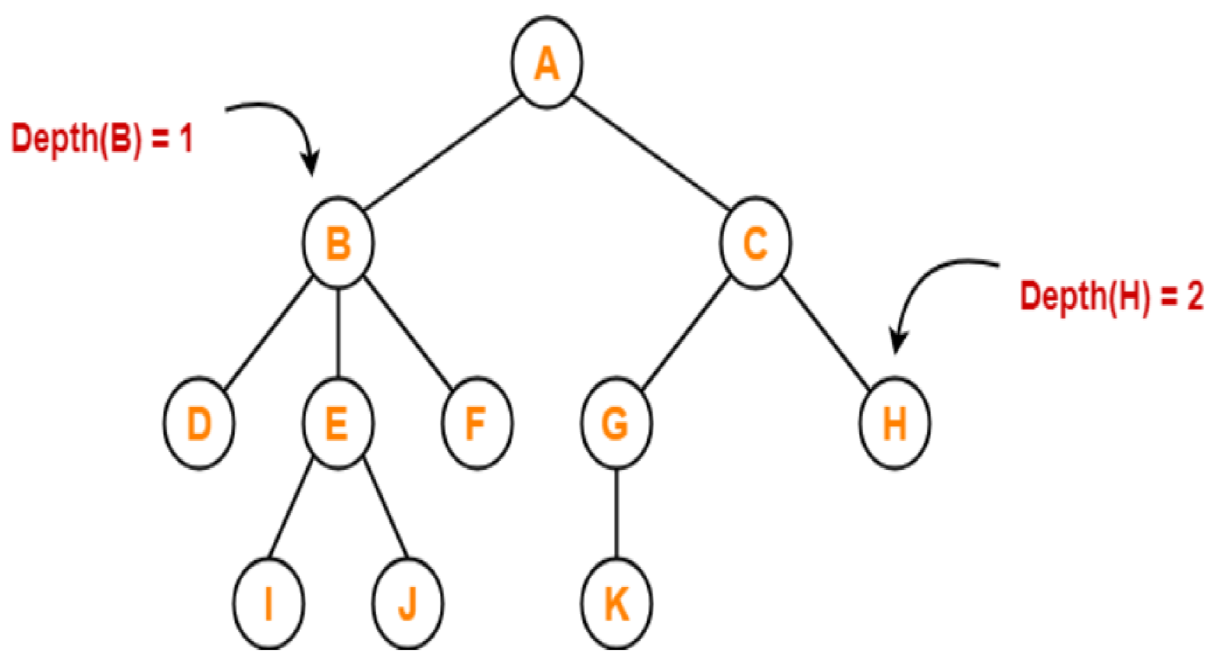
Height and Depth of a TREE



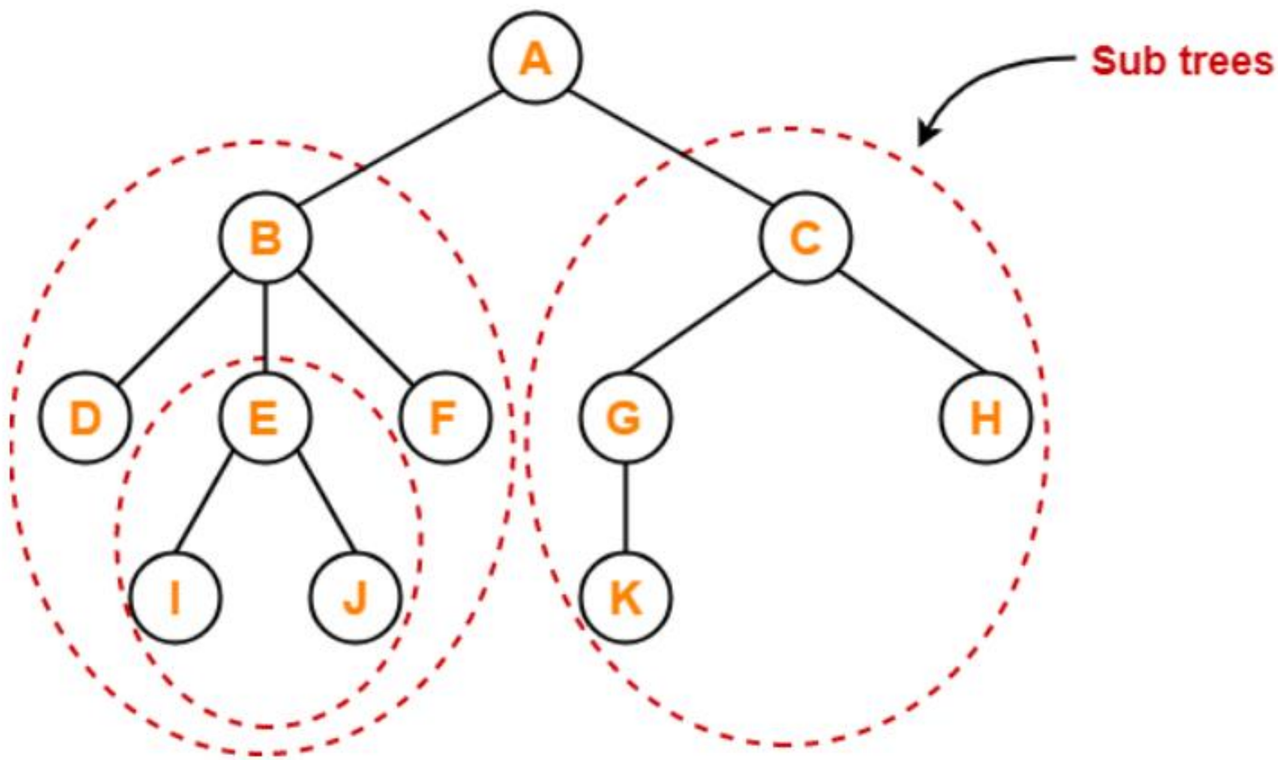
Depth-

•Total number of edges from root node to a particular node is called as depth of that node.
•Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
•Depth of the root node = 0
•The terms “level” and “depth” are used interchangeably.

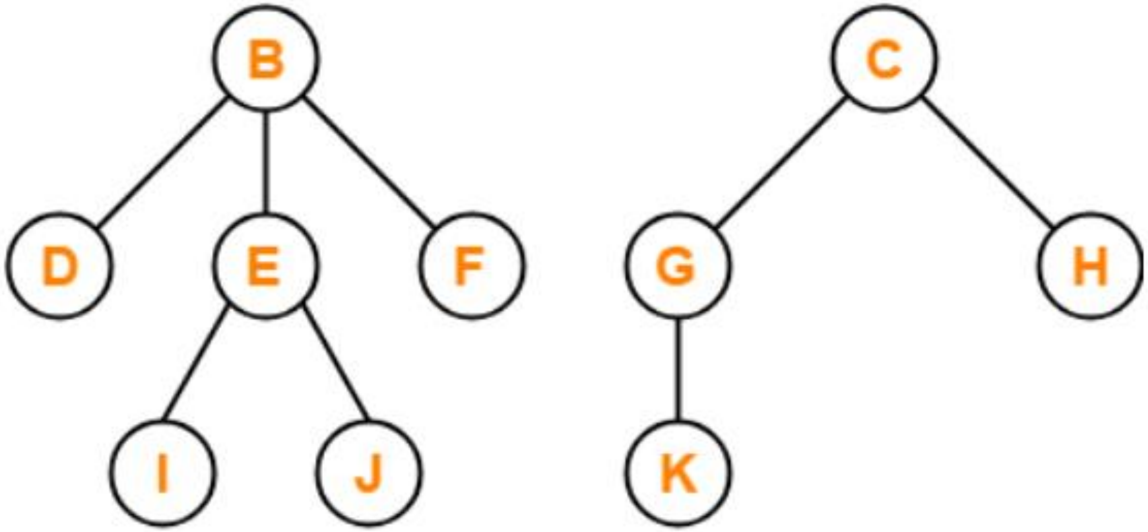
•Depth of node A = 0
•Depth of node B = 1
•Depth of node C = 1
•Depth of node D = 2
•Depth of node E = 2
•Depth of node F = 2
•Depth of node G = 2
•Depth of node H = 2
•Depth of node I = 3
•Depth of node J = 3
•Depth of node K = 3



- **Subtree-**
- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



- **Forest-**
- A forest is a set of disjoint trees.



Forest

Binary Trees – A Informal Definition

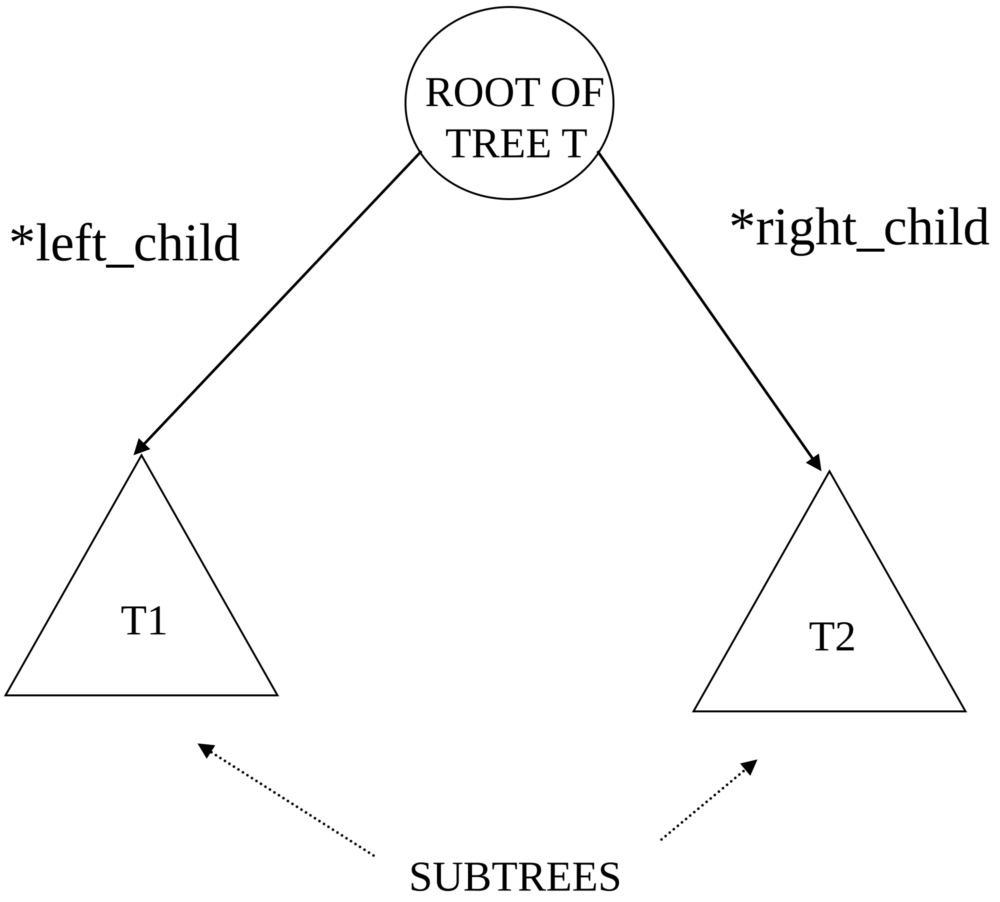
- A binary tree is a tree in which no node can have more than two children.
- Each node has 0, 1, or 2 children
  - In this case we can keep direct links to the children:

```
struct TreeNode
{
    data_type element;
    TreeNode *left_child;
    TreeNode *right_child;
};
```

Binary Trees – Formal Definition

- A binary tree is a structure that
  - contains no nodes, or
  - is comprised of three disjoint sets of nodes:
    - a **root**
    - a binary tree called its **left subtree**
    - a binary tree called its **right subtree**
- A binary tree that contains no nodes is called **empty**

Binary Trees: Recursive Definition

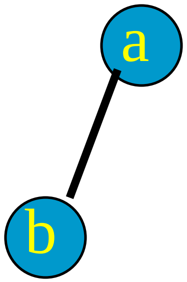


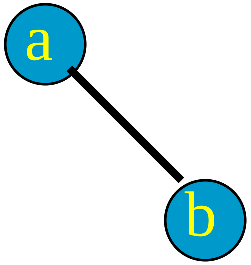
Differences Between A Tree & A Binary Tree

- No node in a binary tree may have more than 2 children, whereas there is no limit on the number of children of a node in a tree.
- The subtrees of a binary tree are ordered; those of a tree are not ordered.

Differences Between A Tree & A Binary Tree (cont.)

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

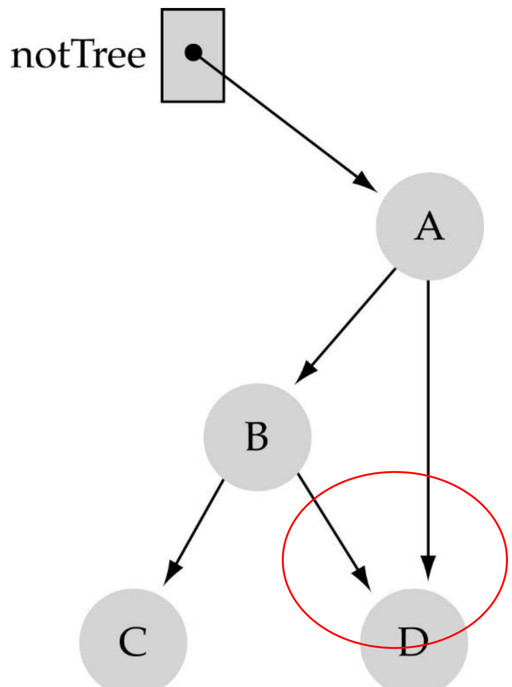




- Are different when viewed as binary trees
- Are the same when viewed as trees

What is a binary tree? (cont.)

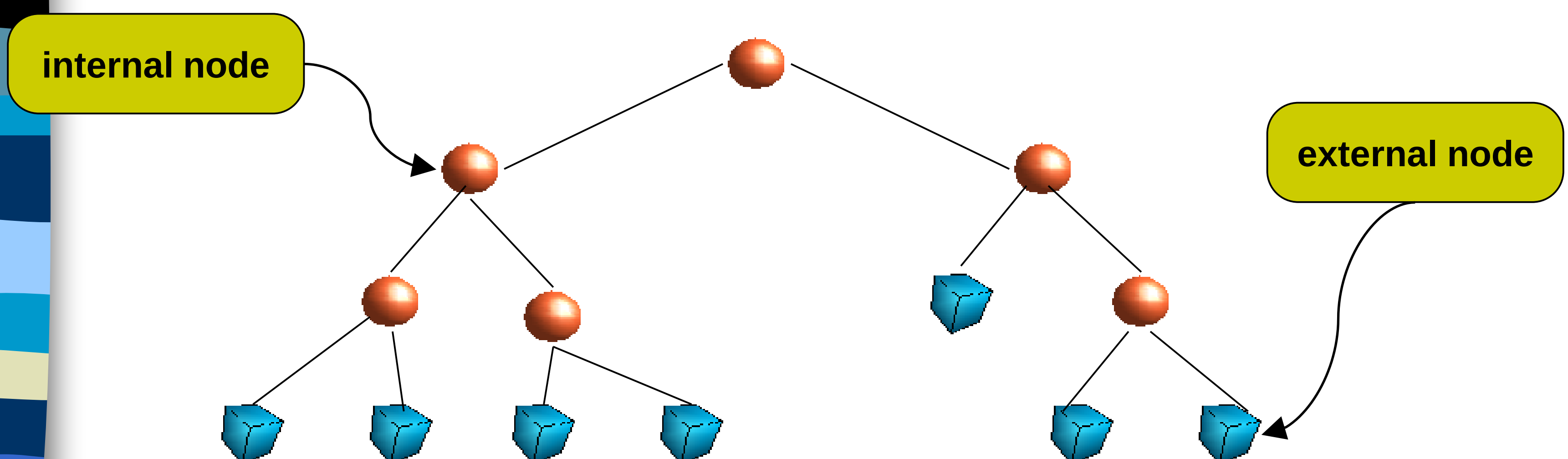
- *Property2:* a unique path exists from the root to every other node



As there are 2 paths to reach node ‘D’, this is NOT a tree Rather it is a graph.

Internal and External Nodes

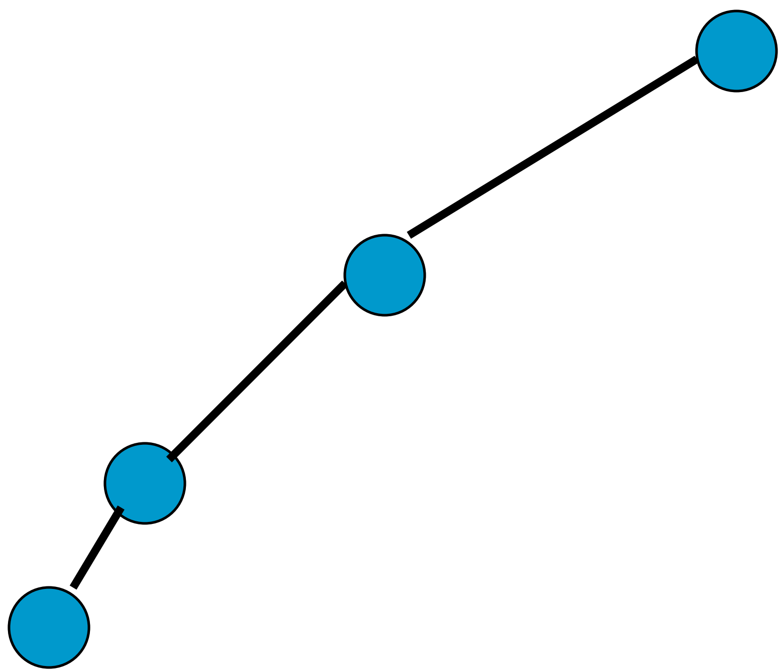
- Because in a binary tree all the nodes must have the same number of children we are forced to change the concepts slightly
  - We say that all internal nodes have two children
  - External nodes have no children



Mathematical Properties of Binary Trees

Minimum Number Of Nodes

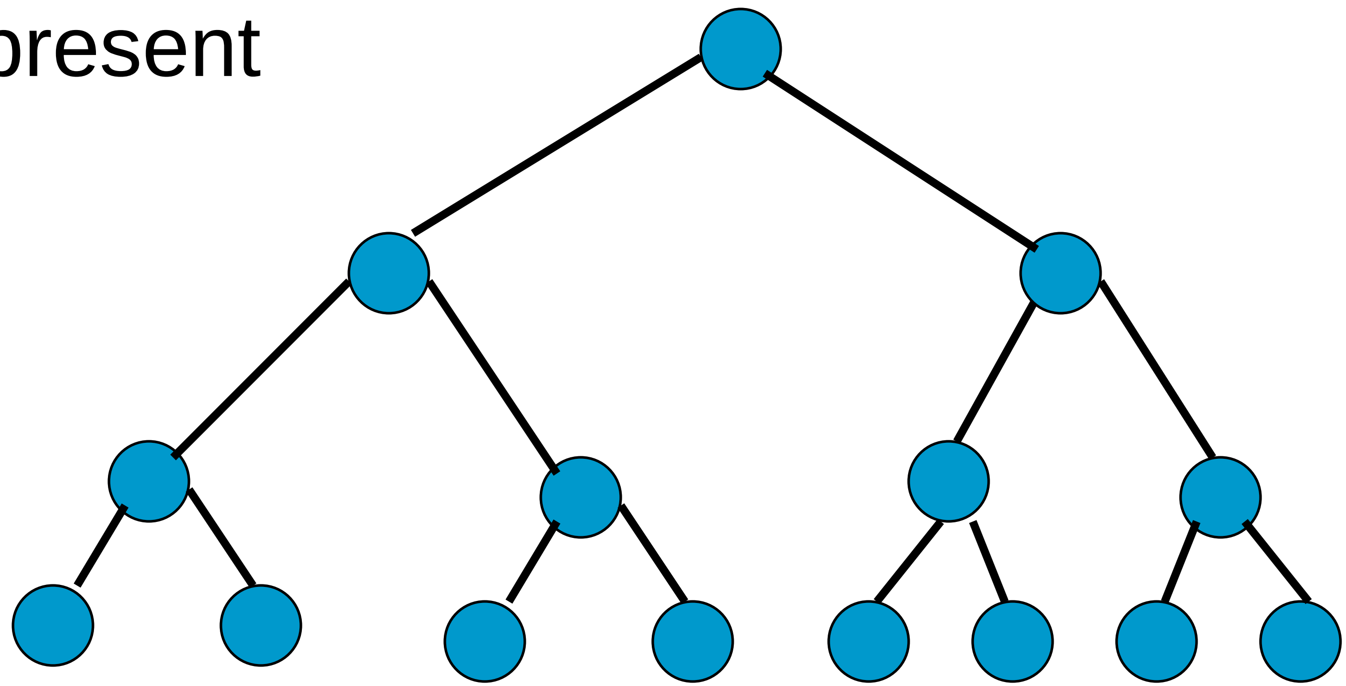
- Minimum number of nodes in a binary tree whose height is h.
- At least one node at each level.



minimum number of nodes is h + 1

Maximum Number Of Nodes

- All possible nodes at first h levels are present



- Maximum number of nodes

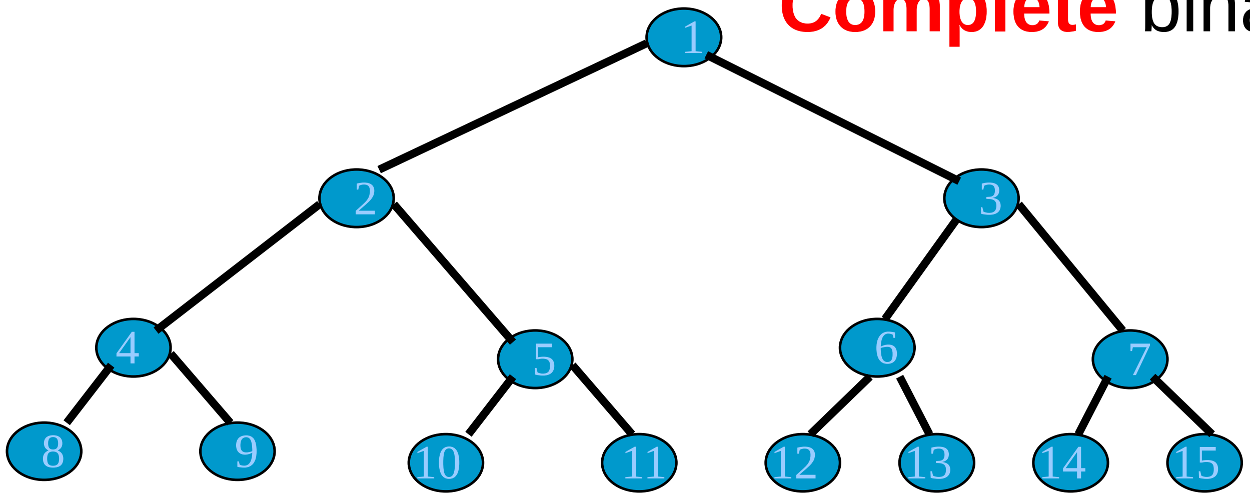
$= 1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1} - 1$

Binary Trees

- **Complete** binary tree :
  - All leaves have the same level
  - All internal nodes have two children

- **Full** binary tree :
  - All internal nodes have two children.

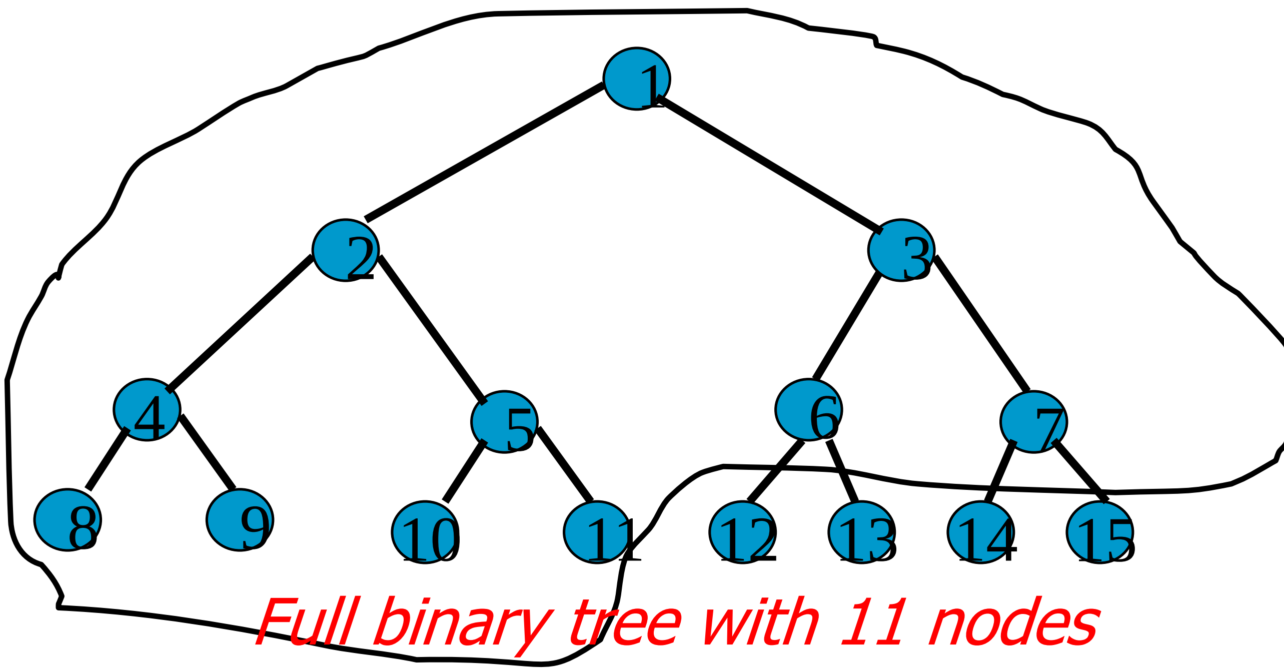
Node Number Properties



- Parent of node i is node i/2
  - But node 1 is the root and has no parent
- Left child of node i is node 2i
  - But if 2i > n, node i has no left child
- Right child of node i is node 2i+1
  - But if 2i+1 > n, node i has no right child

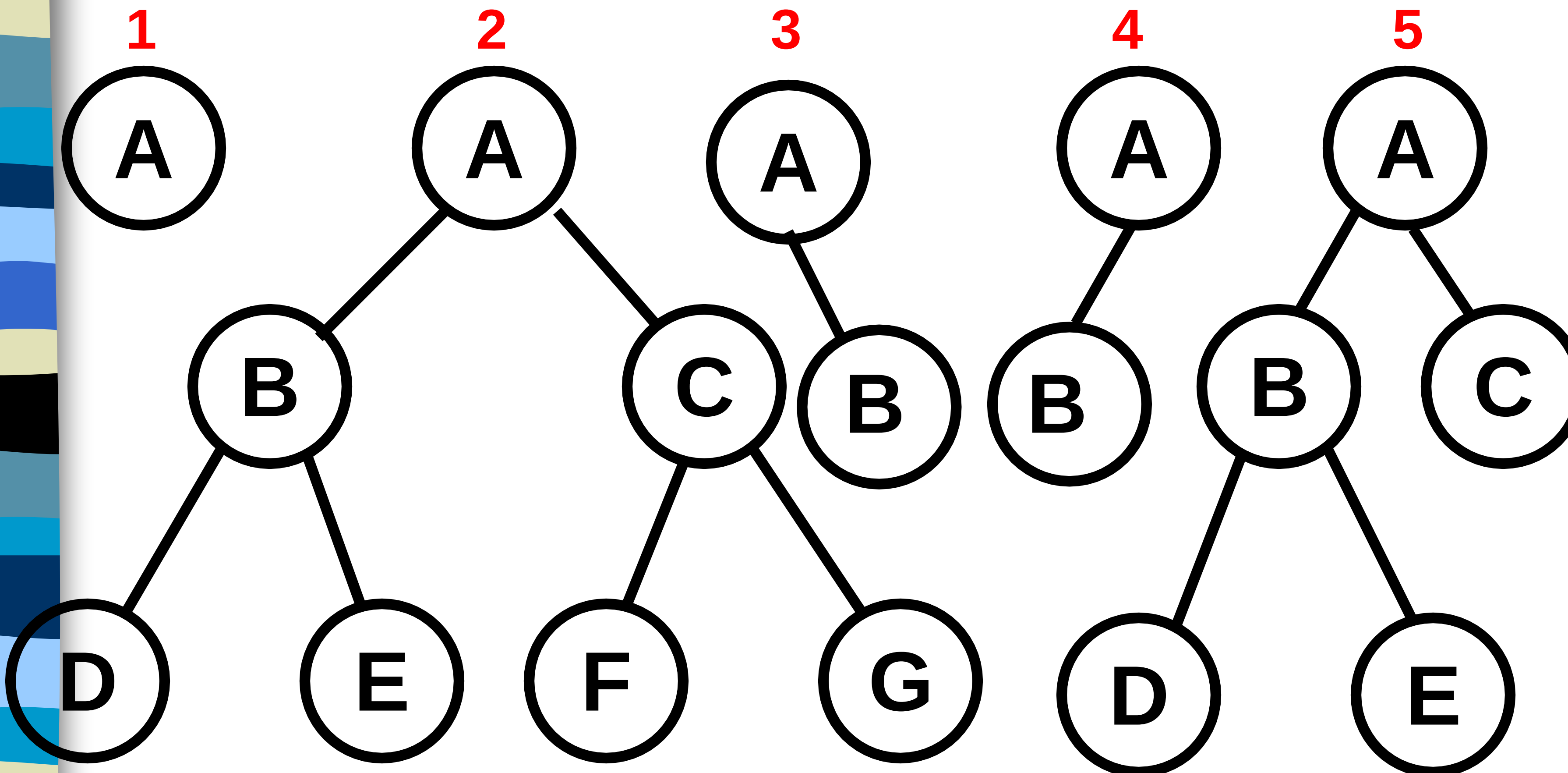
Full Binary Tree With n Nodes

- Start with a complete 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 full binary tree.



Full binary tree with 11 nodes

Tree Examples



A full binary tree is always a complete binary tree.

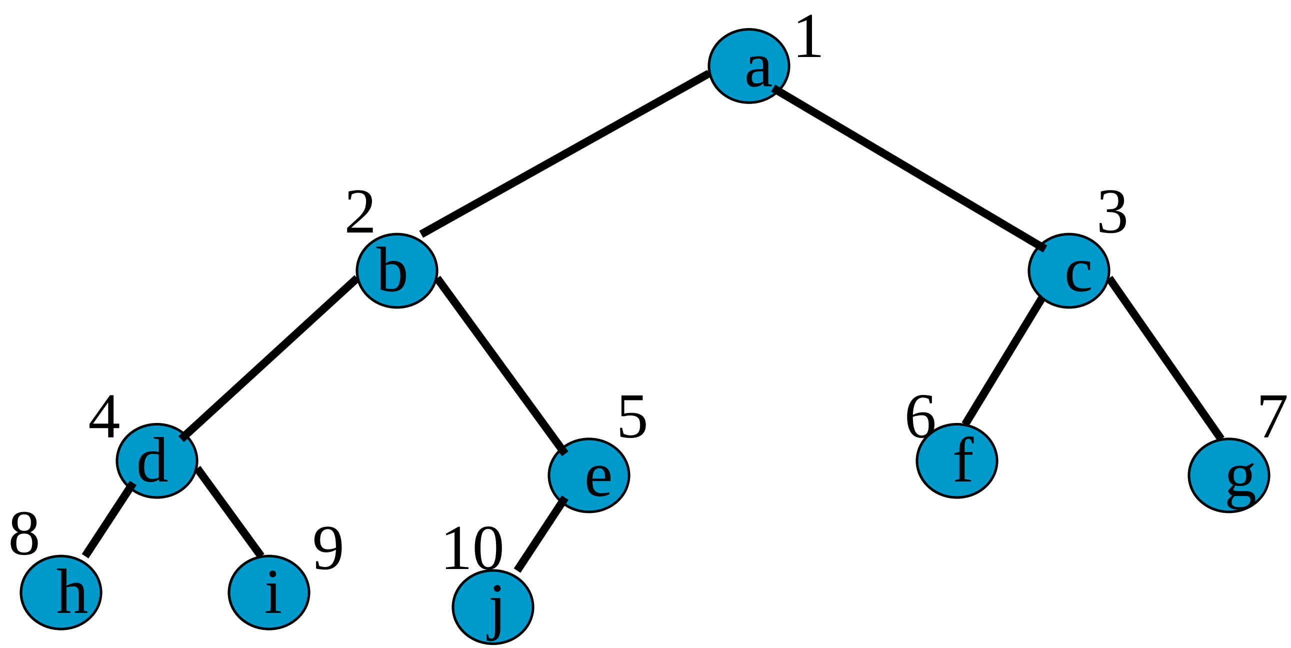
What is the number of nodes in a full binary tree?

Binary Tree Representation

- Array representation
- Linked representation

Array Representation

- Number the nodes using the numbering scheme for a full binary tree
- Store the node numbered i in tree[i]

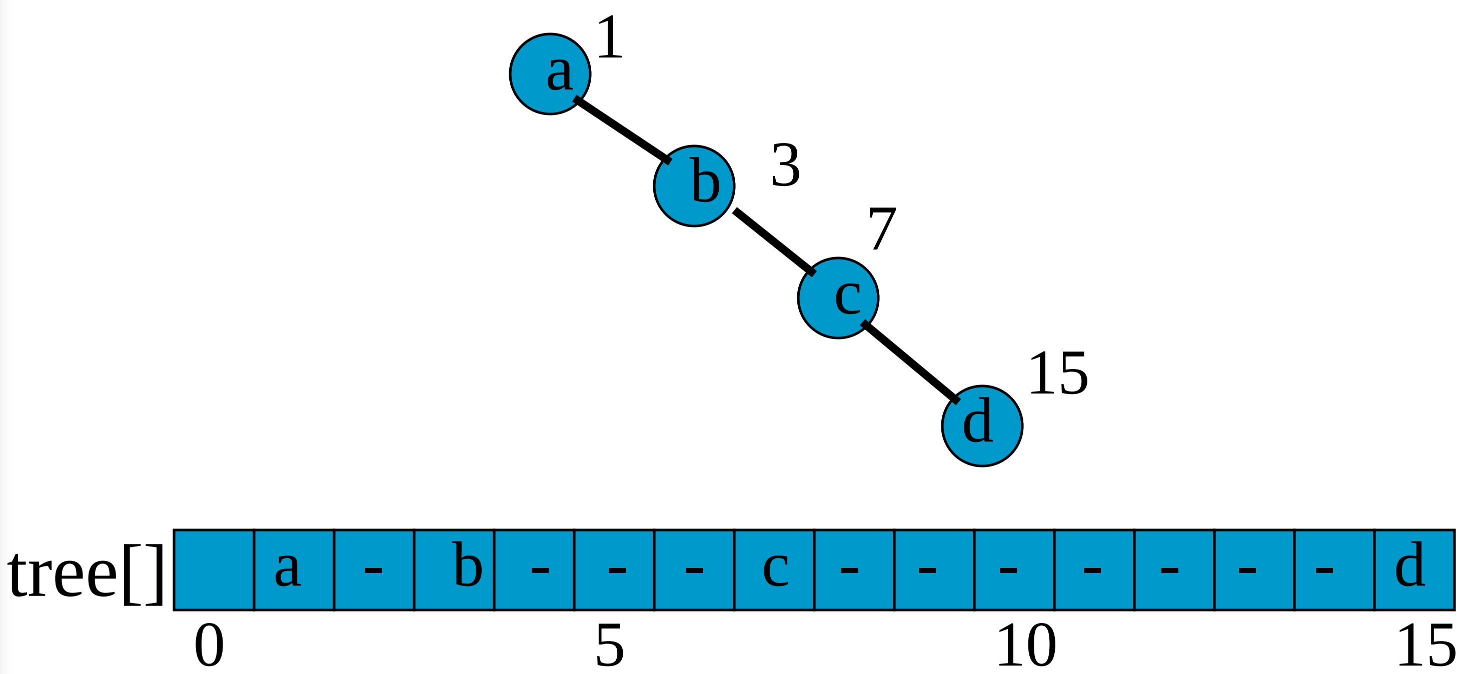


tree[]

	a	b	c	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9	10

Right-Skewed Binary Tree

- An n node binary tree needs an array whose length is between n+1 and 2^n
- If h = n-1 then skewed binary tree



Array Representation- Exercise:

tree[]		a	b			c					d
	0					5					10

tree[]		a	b	c	d		e		h	i			g
	0	1	2	3	4	5	6	7	8	9	10	11	12

tree[]		a	b	c		e	f	g			j	k
	0	1	2	3	4	5	6	7	8	9	10	11

Linked Representation

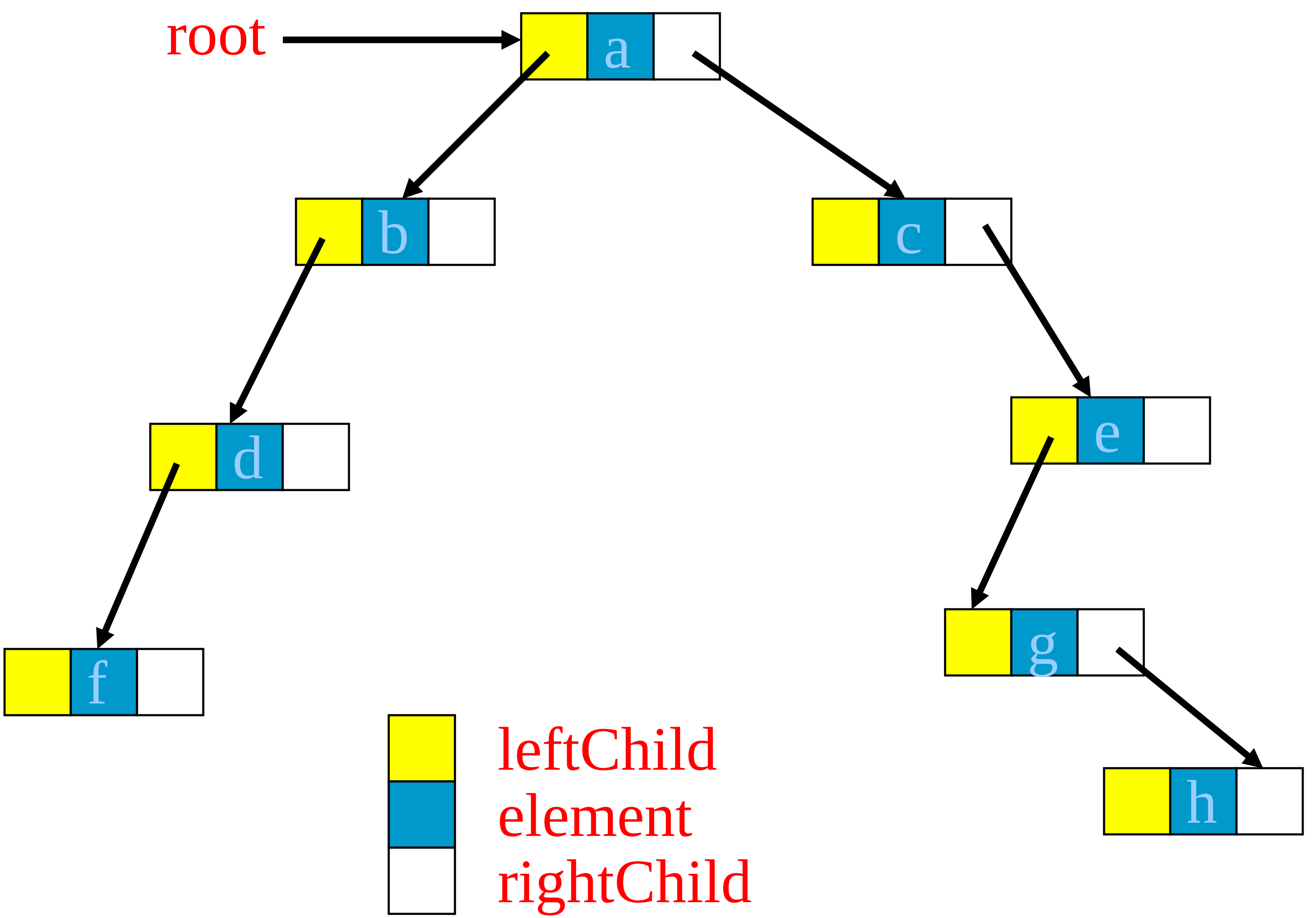
- Each tree node is represented as an structure whose data type is struct TreeNode
- The space required by an n node binary tree is n * (space required by one node)

Binary Trees

- A ***binary tree*** is a tree whose nodes have at most two offspring
- **Example**

```
struct nodeType {
    data_type element;
    struct nodeType *left, *right;
};
struct nodeType *ROOT= NULL;
```

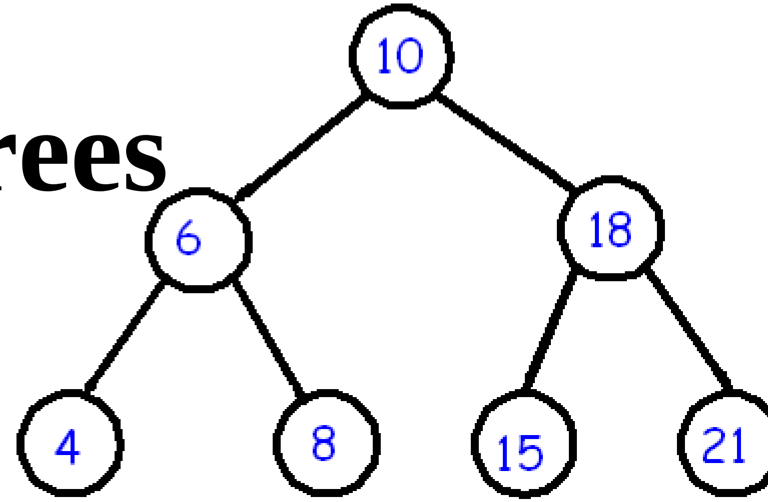
Linked Representation Example



Binary Search Trees

- Particular kind of a binary tree called a Binary Search Tree (BST).
- Provides the efficient way of data sorting, searching and retrieving.
- A BST is a binary tree where nodes are ordered in the following way:
  - each node contains one key (also known as data)
  - the keys in the left subtree are less then the key in its parent node, in short L < P;
  - the keys in the right subtree are greater the key in its parent node, in short P < R;
  - duplicate keys are not allowed.

Binary Search Trees



- In the above tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10.
- Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

Some Binary Tree Operations

- Determine the height.
- Determine the number of nodes.
- Display the binary tree.
- Represent an arithmetic expression using a binary tree.
- Obtain the infix form of an expression.
- Obtain the prefix form of an expression.
- Obtain the postfix form of an expression.
- Evaluate the arithmetic expression represented by a binary tree

Traversing a BINARY TREE

- INORDER
- PREORDER
- POSTORDER

Inorder Traversal

1. Traverse the LEFT subtree of R in Inorder
2. Process the ROOT
3. Traverse the RIGHT subtree of R in Inorder

Inorder Traversal

Inorder(node)

Step 1: repeat through step 4

if node != NULL

Step 2: Call Inorder(Leftchild(node))

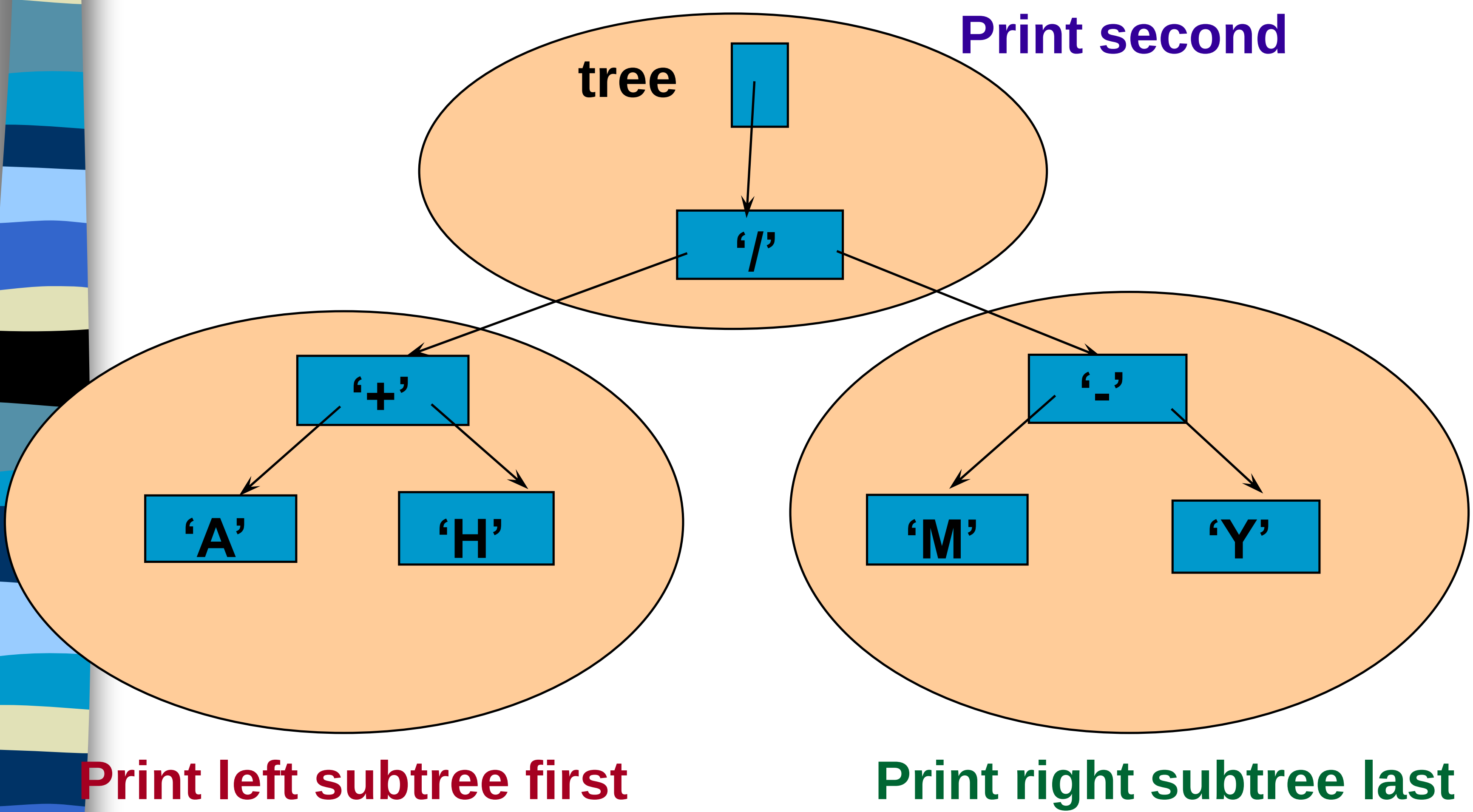
Step 3: Output info(node)

Step 4: Call Inorder(Rightchild(node))

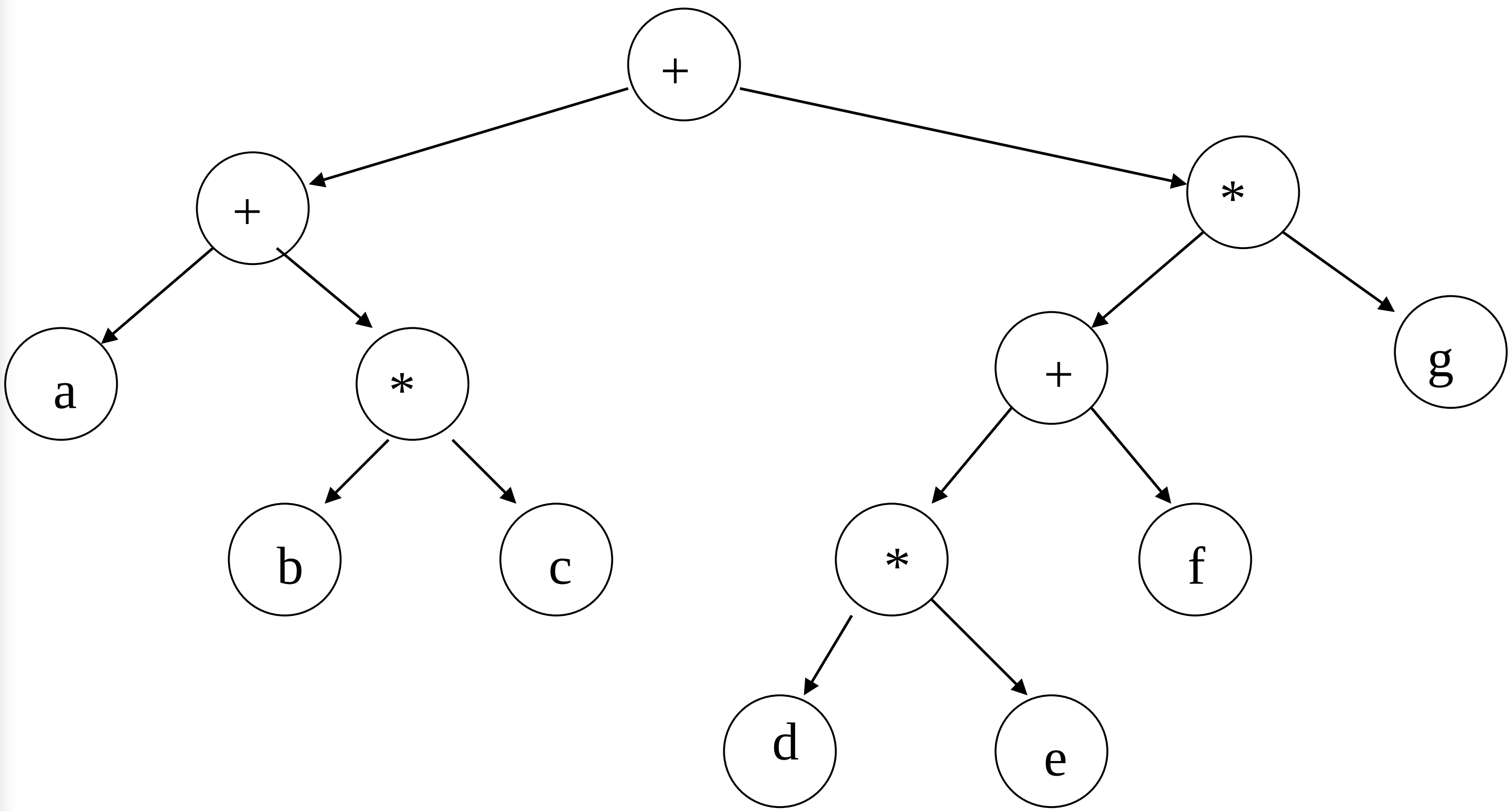
Step 5: Exit

Inorder Traversal: (A + H) / (M - Y)

LNR



Inorder Traversal (cont.)



Inorder traversal yields: (a + (b * c)) + (((d * e) + f) * g)

Preorder Traversal

1. Process the ROOT
2. Traverse the LEFT subtree of R in Preorder
3. Traverse the RIGHT subtree of R in Preorder

Preorder Traversal

Preorder(node)

Step 1: repeat through step 3

    if node != NULL

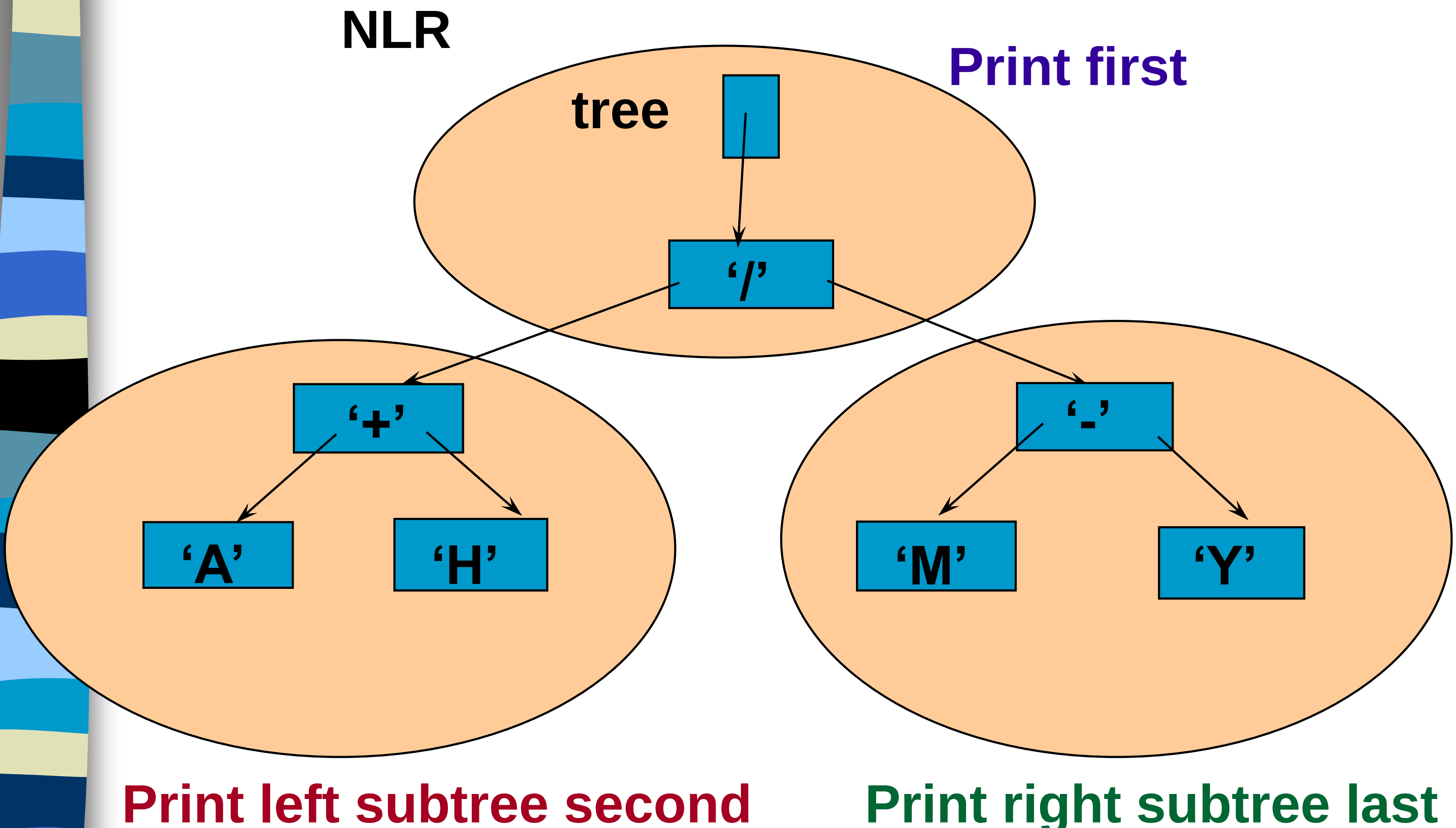
        Output info(node)

Step 2: Call Preorder(Leftchild(node))

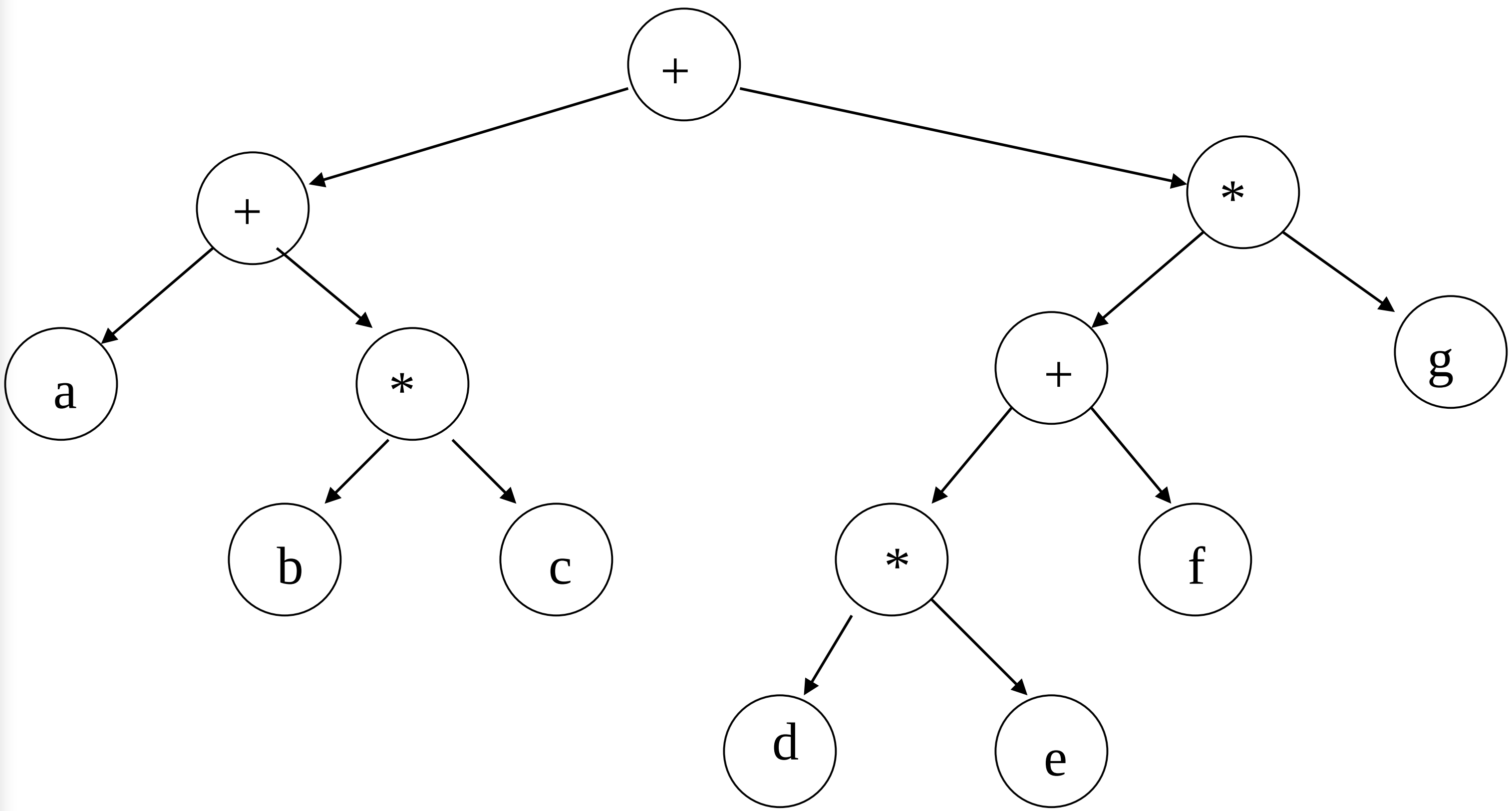
Step 3: Call Preorder(Rightchild(node))

Step 4: Exit

Preorder Traversal: / + A H - M Y



Preorder Traversal (cont.)



Preorder traversal yields: (+ (+ a (* b c)) (* (+ (* d e) f) g))

Postorder Traversal

1. Traverse the LEFT subtree of R in Postorder
2. Traverse the RIGHT subtree of R in Postorder
3. Process the ROOT

Postorder Traversal

Postorder(node)

Step 1: repeat through step 4

if node != NULL

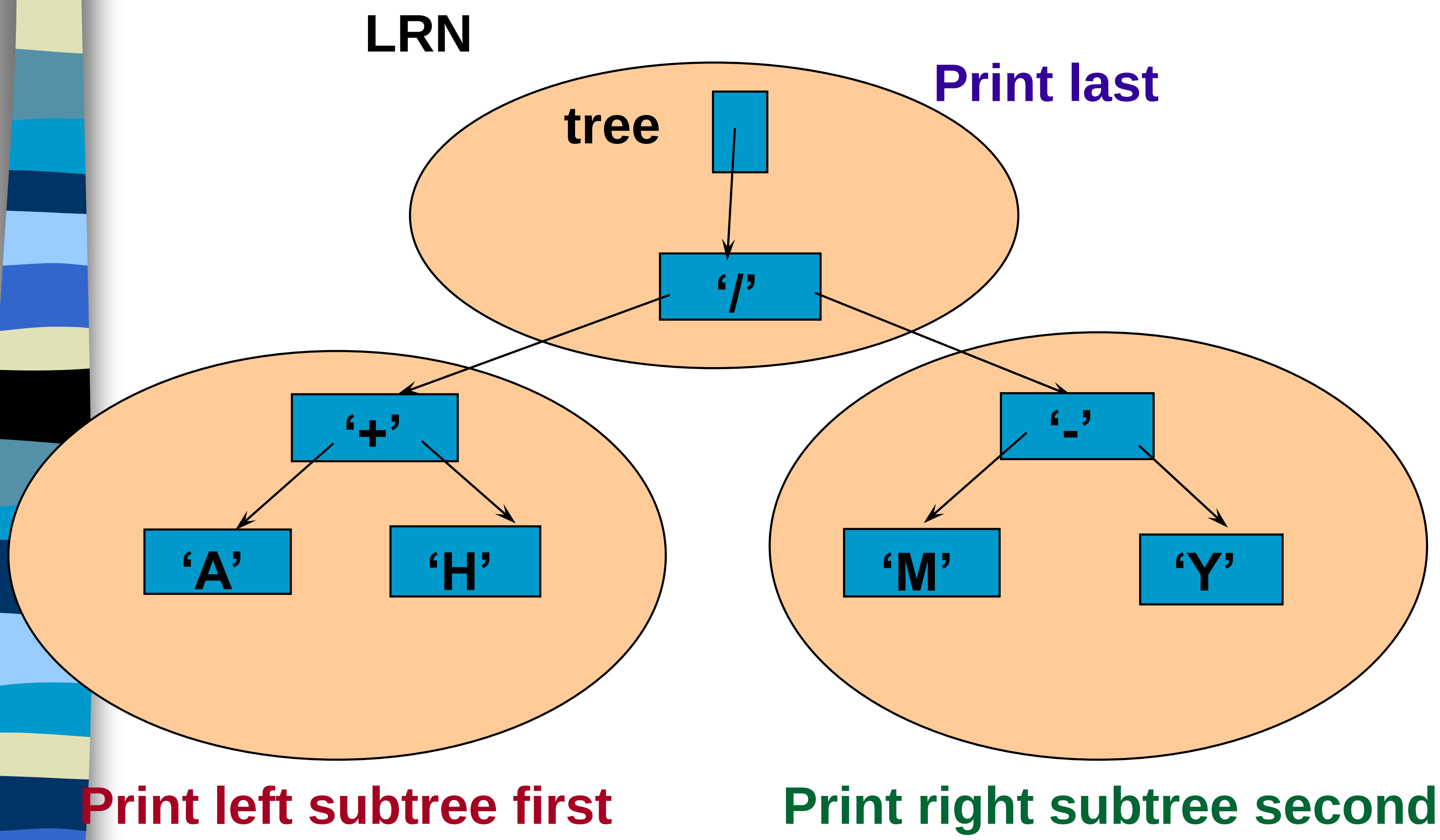
Step 2: Call Postorder(Leftchild(node))

Step 3: Call Postorder(Rightchild(node))

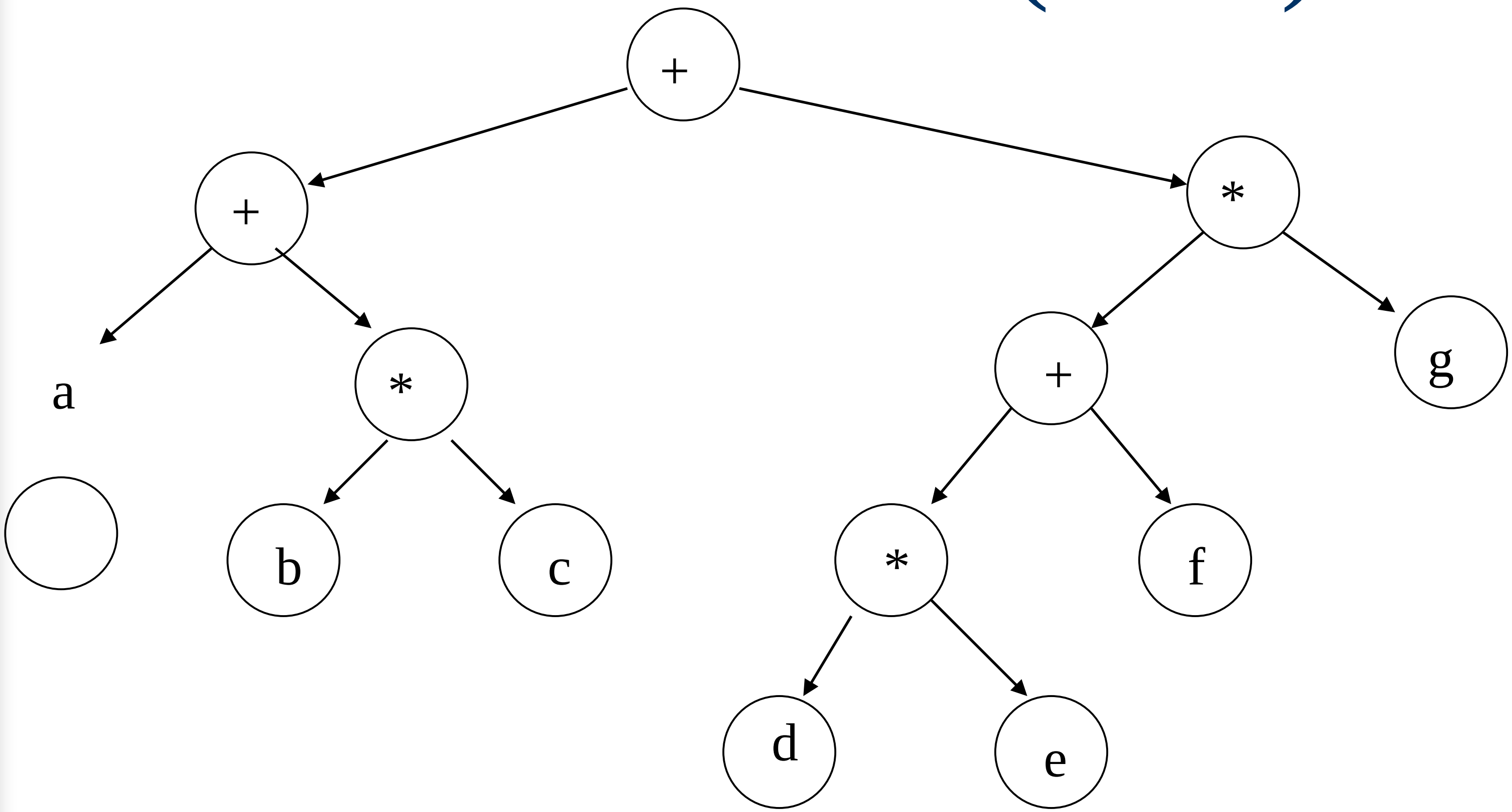
Step 4: Output info(node)

Step 5: Exit

Postorder Traversal: A H + M Y - /



Postorder Traversal (cont.)

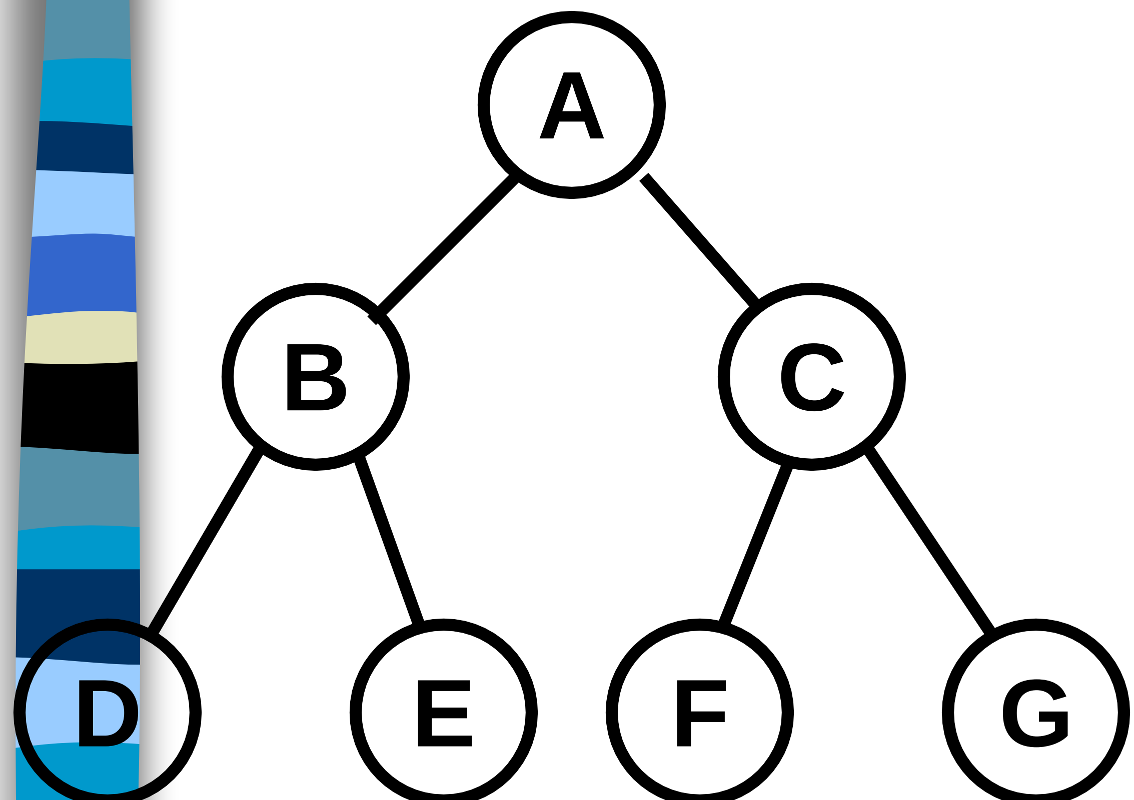


Postorder traversal yields: a b c * + d e * f + g * +

Tree Traversals - Algorithms

- preorder traversal
  1. Visit the root.
  2. Perform a preorder traversal of the left subtree.
  3. Perform a preorder traversal of the right subtree.
- inorder traversal
  1. Perform an inorder traversal of the left subtree.
  2. Visit the root.
  3. Perform an inorder traversal of the right subtree.
- postorder traversal
  1. Perform a postorder traversal of the left subtree.
  2. Perform a postorder traversal of the right subtree.
  3. Visit the root.

Traversal Example



preorder

A B D E C F G

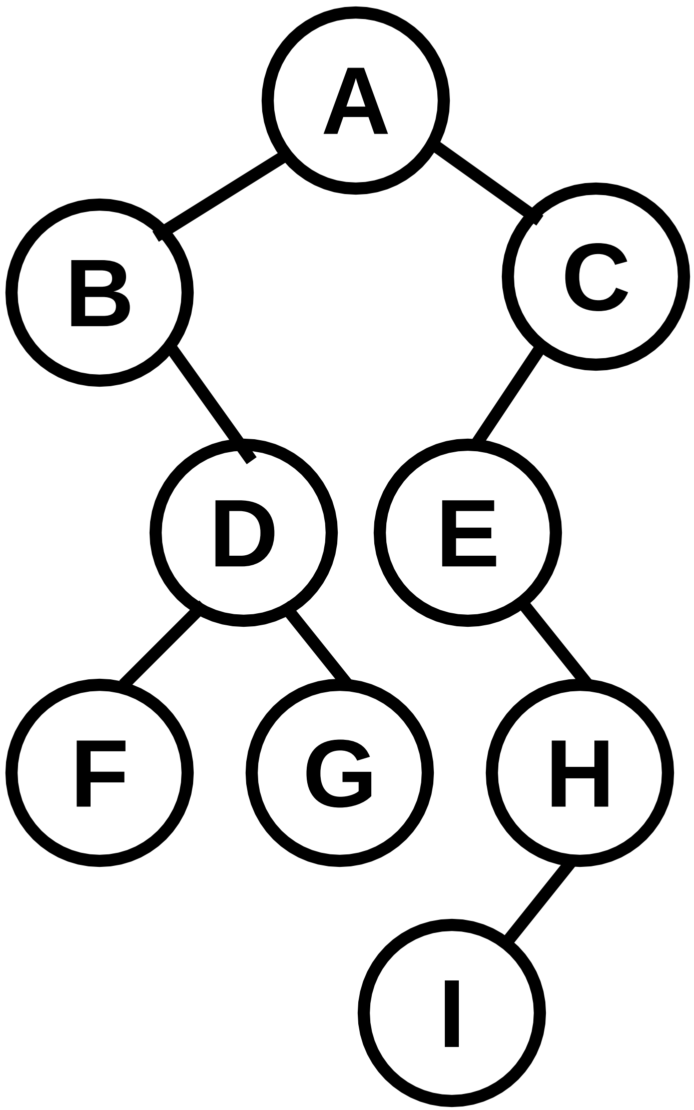
inorder

D B E A F C G

postorder

D E B F G C A

Traversal Example



preorder

ABDFGCEHI

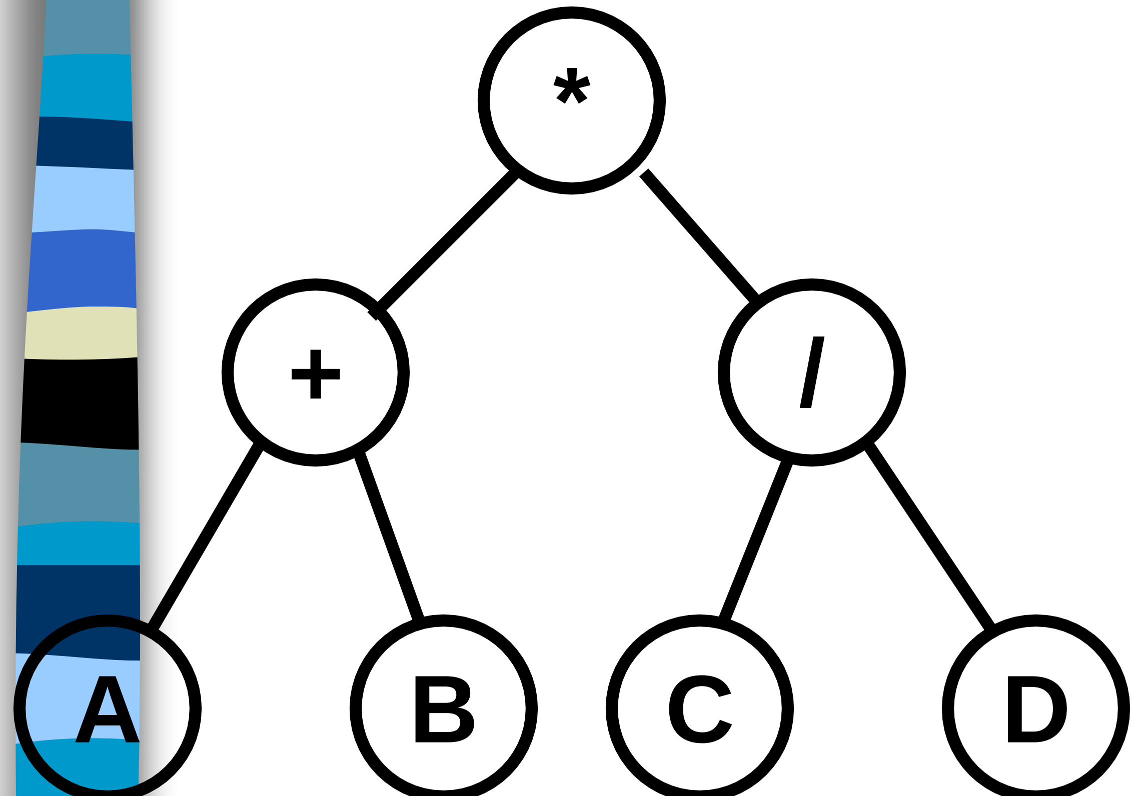
inorder

BFDGAEIHC

postorder

FGDBIHECA

Traversal Example
(expression tree)



preorder

***+AB/CD**

inorder

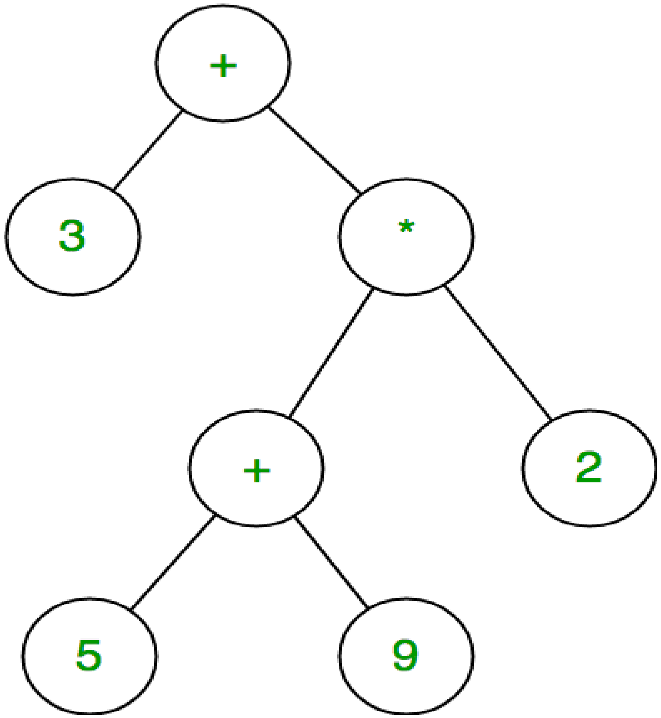
A+B*C/D

postorder

AB+CD/*

Expression Tree

- The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:

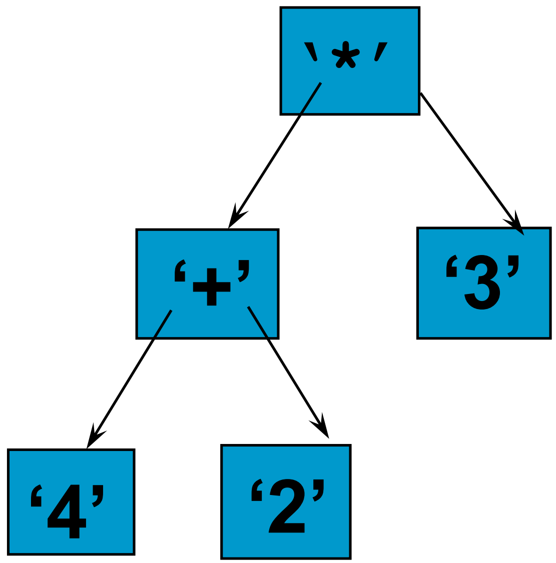


Evaluating the expression represented by an expression tree:

```
Let t be the expression tree
If  t is not null then
    If t.value is operand then
        Return  t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

A Binary Expression Tree



What value does it have?

(4 + 2) * 3 = 18

Construct Binary Tree by Inspection Method

- [a+(b-c)]*[(d-e)/(f+g-h)]
- $(2x + y)(5a - b)^3$
- Inorder: E A C K F H D B G
  Preorder: F A E K C D H G B
- Inorder: B D A E H G I F C
  Postorder: D B H I G F E C A

Construct a Binary Tree from Postorder and Inorder

- Given Postorder and Inorder traversals, construct the tree.
- **Example:**

Input:
in[]   = {2, 1, 3}
post[] = {2, 3, 1}

Output: Root of below tree

```
      1
     / \
    2   3
```

Construct a Binary Tree from Postorder and Inorder

- Given Postorder and Inorder traversals, construct the tree.
- **Example:**

Input:
in[] = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}

Construct a Binary Tree from Postorder and Inorder

Input:
in[]   = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}

- **Approach:** To solve the problem follow the below idea:

Follow the below steps:

The process of constructing a tree from **in[]** = {4, 8, 2, 5, 1, 6, 3, 7} and **post[]** = {8, 4, 5, 2, 6, 7, 3, 1}:

•First find the last node in post[].

  •The last node is “1”, this value is the root as the root always appears at the end of postorder traversal.

  •Now search “1” in **in[]** to find the left and right subtrees of the root.

  •Everything on the left of “1” in **in[]** is in the left subtree and everything on right is in the right subtree.

Construct a Binary Tree from Postorder and Inorder

Input:
in[] = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}

Follow the below steps:

The process of constructing a tree from **in[]** = {4, 8, 2, 5, 1, 6, 3, 7} and **post[]** = {8, 4, 5, 2, 6, 7, 3, 1}:

- **First find the last node in post[].**
- **The last node is "1",** this value is the root as the root always appears at the end of postorder traversal.
- Now search "1" in **in[]** to find the left and right subtrees of the root.
- Everything on the left of "1" in **in[]** is in the left subtree and everything on right is in the right subtree.

```
        1
      /    \
[4, 8, 2, 5]   [6, 3, 7]
```

Construct a Binary Tree from Postorder and Inorder

Recur the process for the following two.

- Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3} Make the created tree as right child of root.
- Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}. Make the created tree the left child of the root.

```
                            1
                          /    \
                 [4, 8, 2, 5]  [6, 3, 7]

          2                                   3
        /    \                              /    \
    [4, 8]    [5]                         [6]     [7]

  4
    \
     [8]
```

Construct a Binary Tree from Postorder and Inorder

Input:
in[]   = {4, 8, 2, 5, **1,** 6, 3, 7**}**
post[] = {8, 4, 5, 2, 6, 7, 3, **1}**

```
              1
             /   \
   [4, 8, 2, 5]   [6, 3, 7]

        2                    3
       /   \                /   \
   [4, 8]   [5]          [6]    [7]

   4
     \
      [8]
```

Construct a Binary Tree from Postorder and Inorder

- Input:
- inorder = [9,3,15,20,7],
- postorder = [9,15,7,20,3]

Construct Tree from given Inorder and Preorder traversals

- Consider the below traversals:
- Inorder sequence: D B E **A** F C
- Preorder sequence: **A** B D E C F

In a Preorder sequence, the leftmost element is the root of the tree. So we know 'A' is the root for given sequences.
By searching 'A' in the Inorder sequence, we can find out all elements on the left side of 'A' is in the left subtree and elements on right in the right subtree.
So we know the below structure now.

```
      A
    /   \
D B E    F C
```

We recursively follow the above steps and get the following tree.

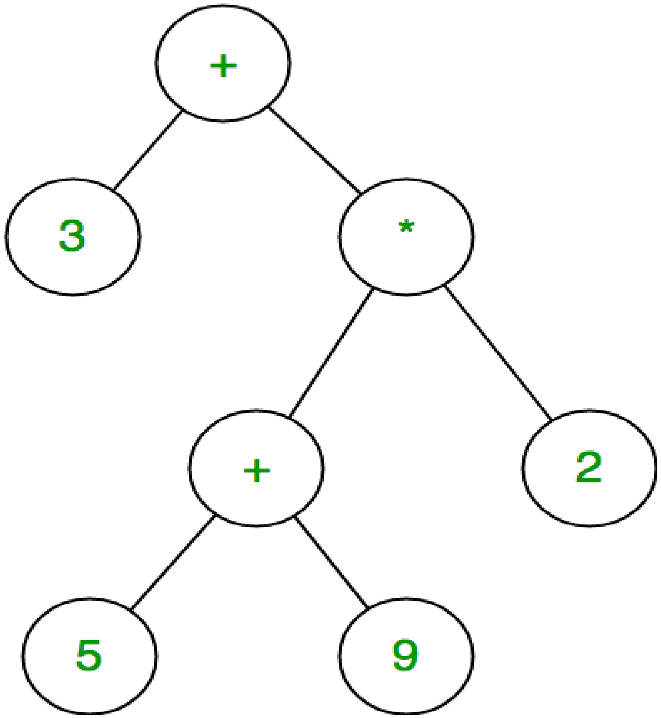
```
        A
      /   \
     B     C
    / \   /
   D   E F
```

Construct Tree from given Inorder and Preorder traversals

- Input:
- preorder = [3,9,20,15,7],
- inorder = [9,3,15,20,7]

Expression Tree

- The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:

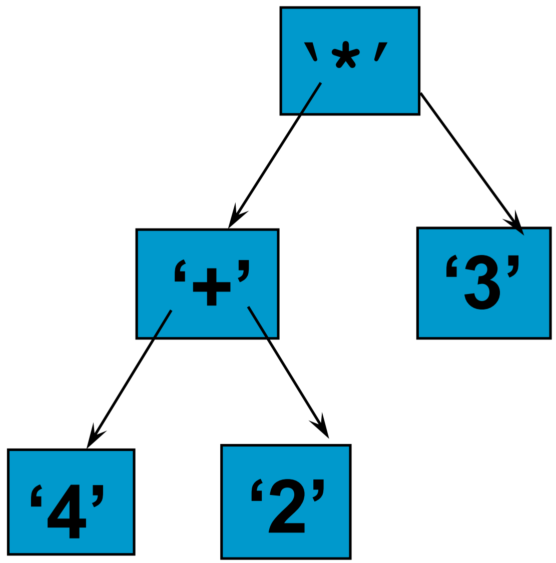


Evaluating the expression represented by an expression tree:

```
Let t be the expression tree
If  t is not null then
      If t.value is operand then
                Return  t.value
      A = solve(t.left)
      B = solve(t.right)

      // calculate applies operator 't.value'
      // on A and B, and returns value
      Return calculate(A, B, t.value)
```

A Binary Expression Tree

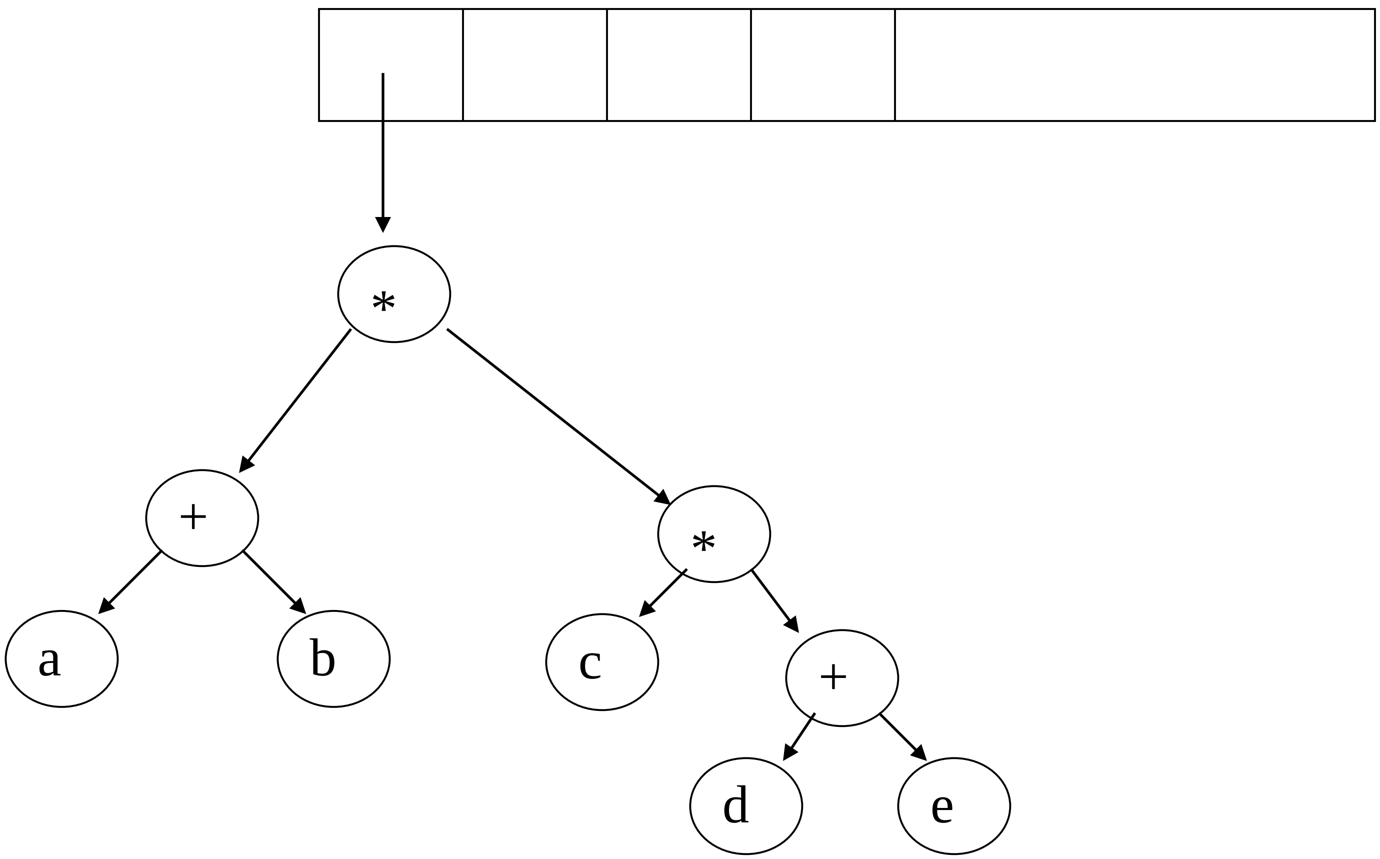


What value does it have?

(4 + 2) * 3 = 18

Example

a b + c d e + * * :



Exercise

- [a+(b-c)]*[(d-e)/(f+g-h)]
- Inorder: E A C K F H D B G

  Preorder: F A E K C D H G B
- $(2x + y)(5a – b)^3$

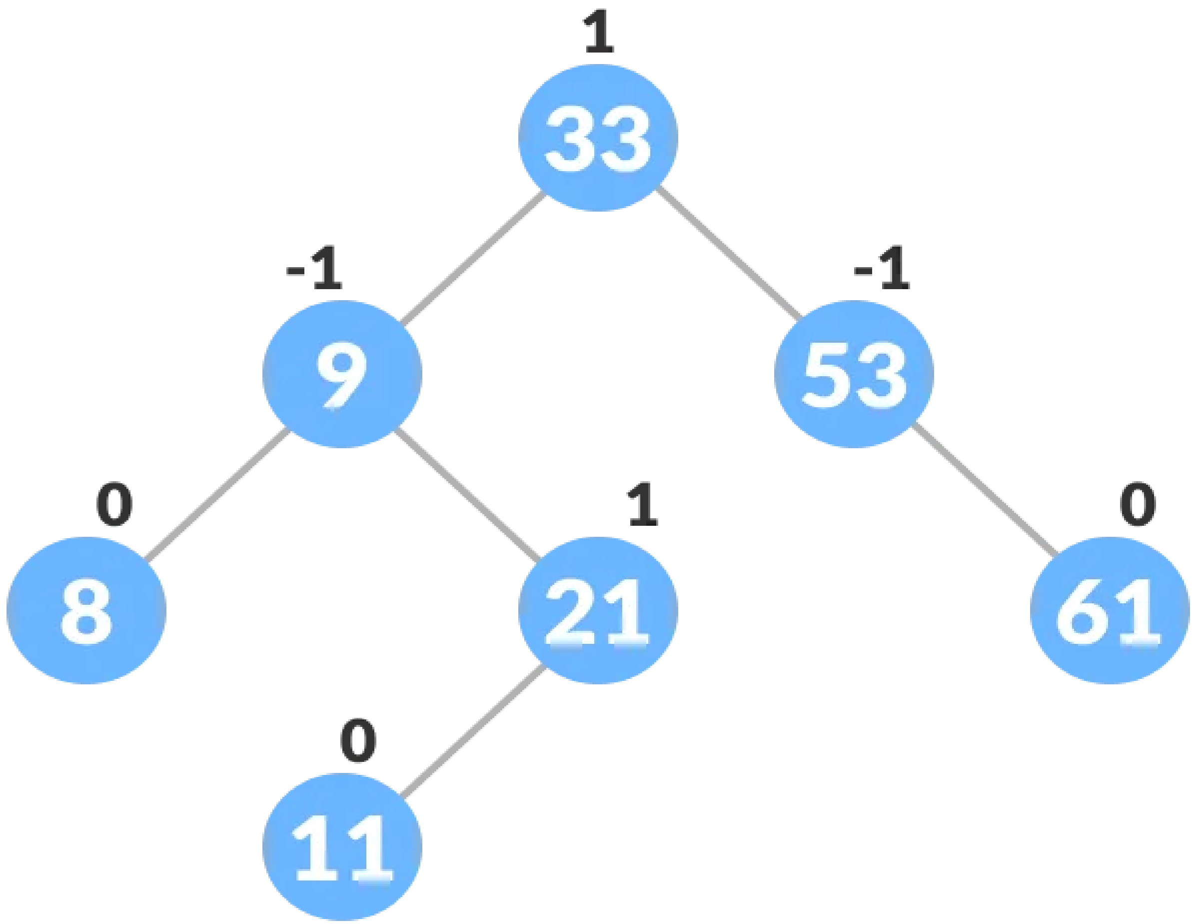
AVL Tree

- AVL tree is a self-balancing binary search tree in which each node maintains extra information called a **balance factor** whose value is either -1, 0 or +1.
- AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

AVL Tree - Balance Factor

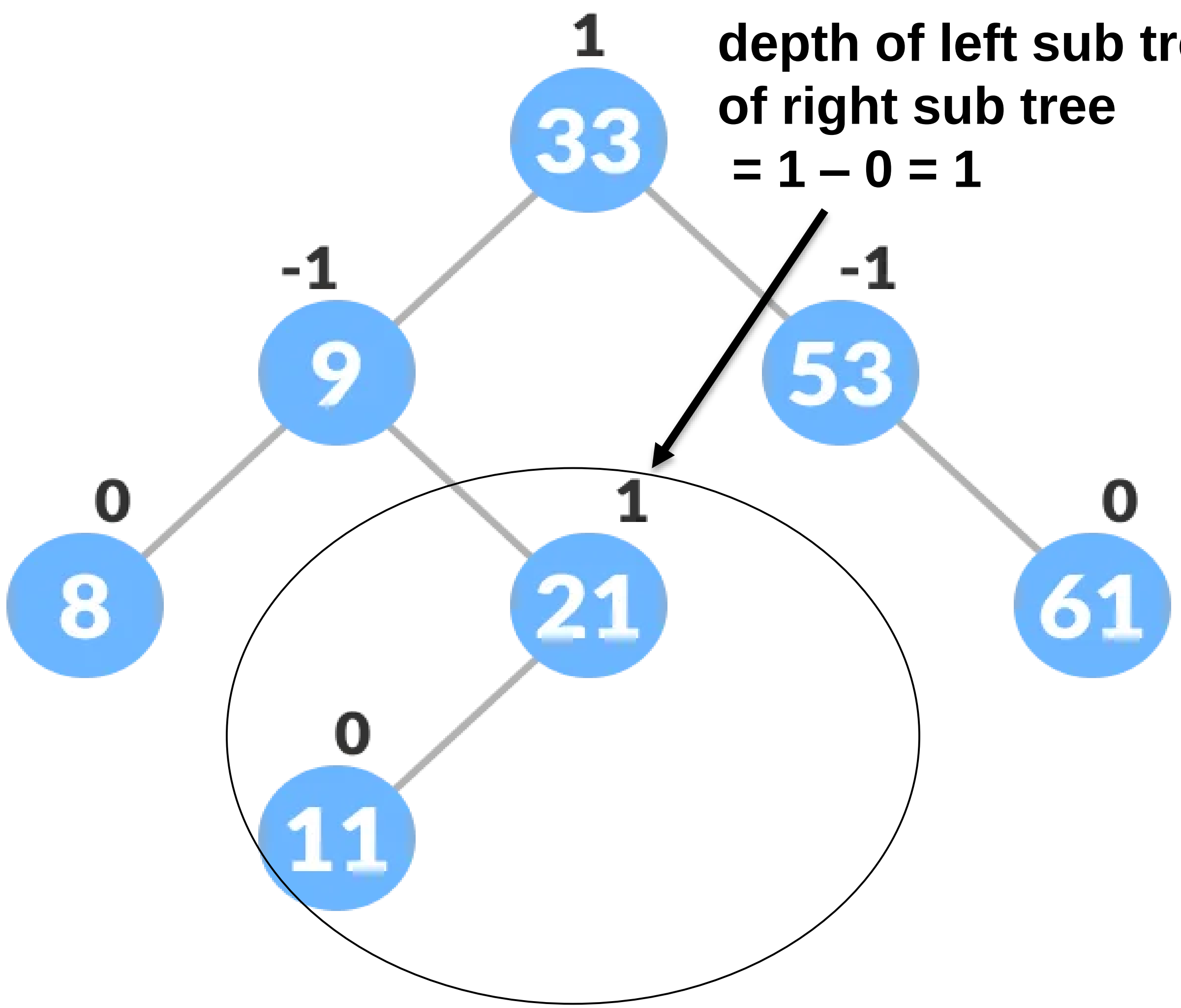
- **Balance factor** of a node in an AVL tree is the **difference between the depth of the left subtree and that of the right subtree of that node**.
- **Balance Factor** = **(Depth of Left Subtree - Depth of Right Subtree)** or **(Depth of Right Subtree - Depth of Left Subtree)**
- The self balancing property of an AVL tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

AVL Tree - example of a balanced AVL tree:



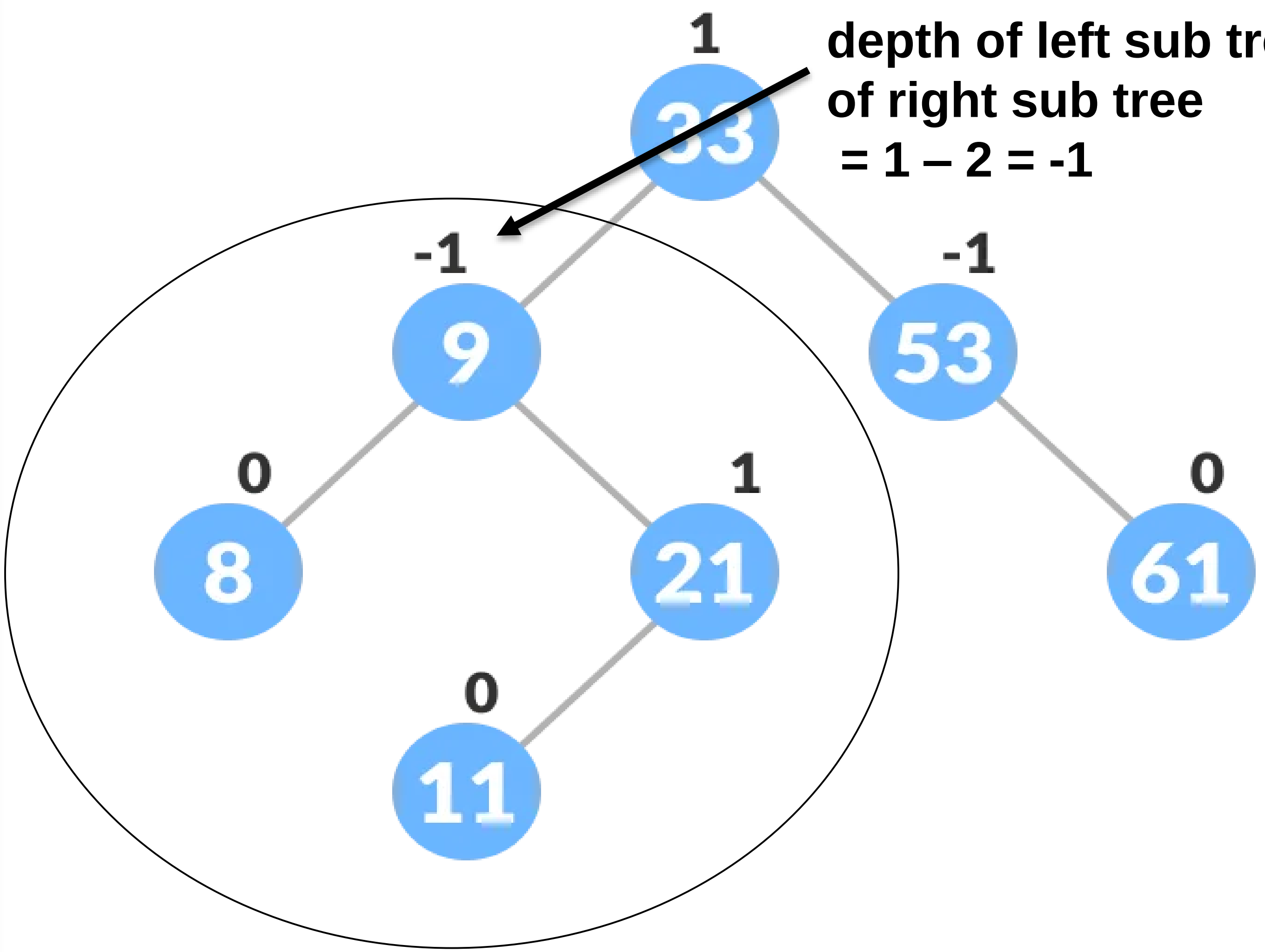
AVL Tree - example of a balanced AVL tree:

Balance factor of Node 21 = depth of left sub tree - depth of right sub tree
= 1 – 0 = 1



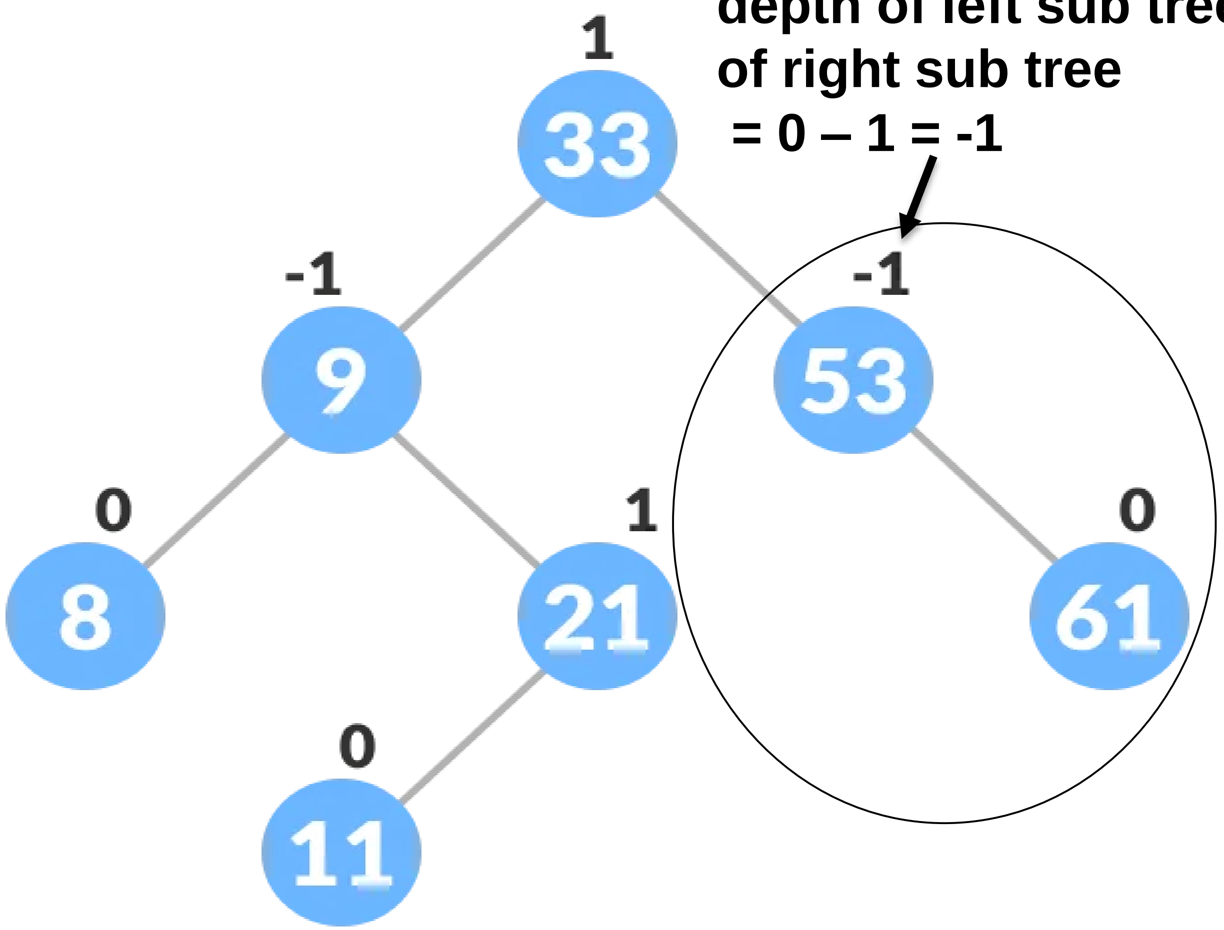
AVL Tree - example of a balanced AVL tree:

Balance factor of Node 9 = depth of left sub tree - depth of right sub tree
= 1 – 2 = -1



AVL Tree - example of a balanced AVL tree:

Balance factor of Node 53 = depth of left sub tree - depth of right sub tree
= 0 – 1 = -1



AVL Tree - example of a balanced AVL tree:

Balance factor of Node 33 = depth of left sub tree - depth of right sub tree = 3 – 2 = 1