TREE

Tree

Tree data structure may be defined as-

Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

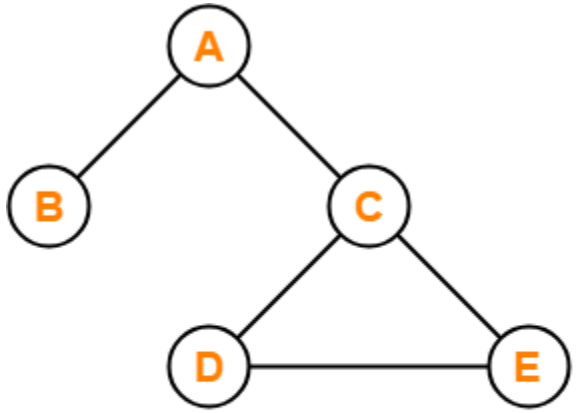
OR

A tree is a connected graph without any circuits.

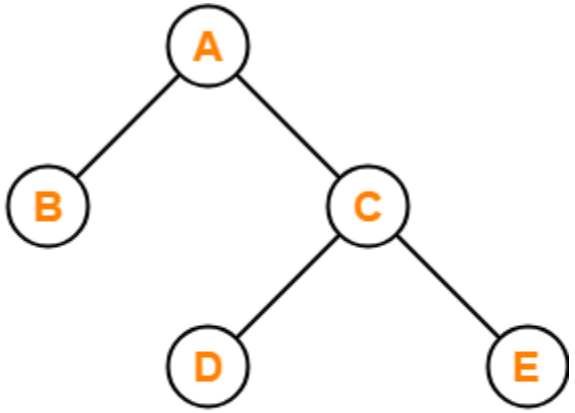
OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree

Example

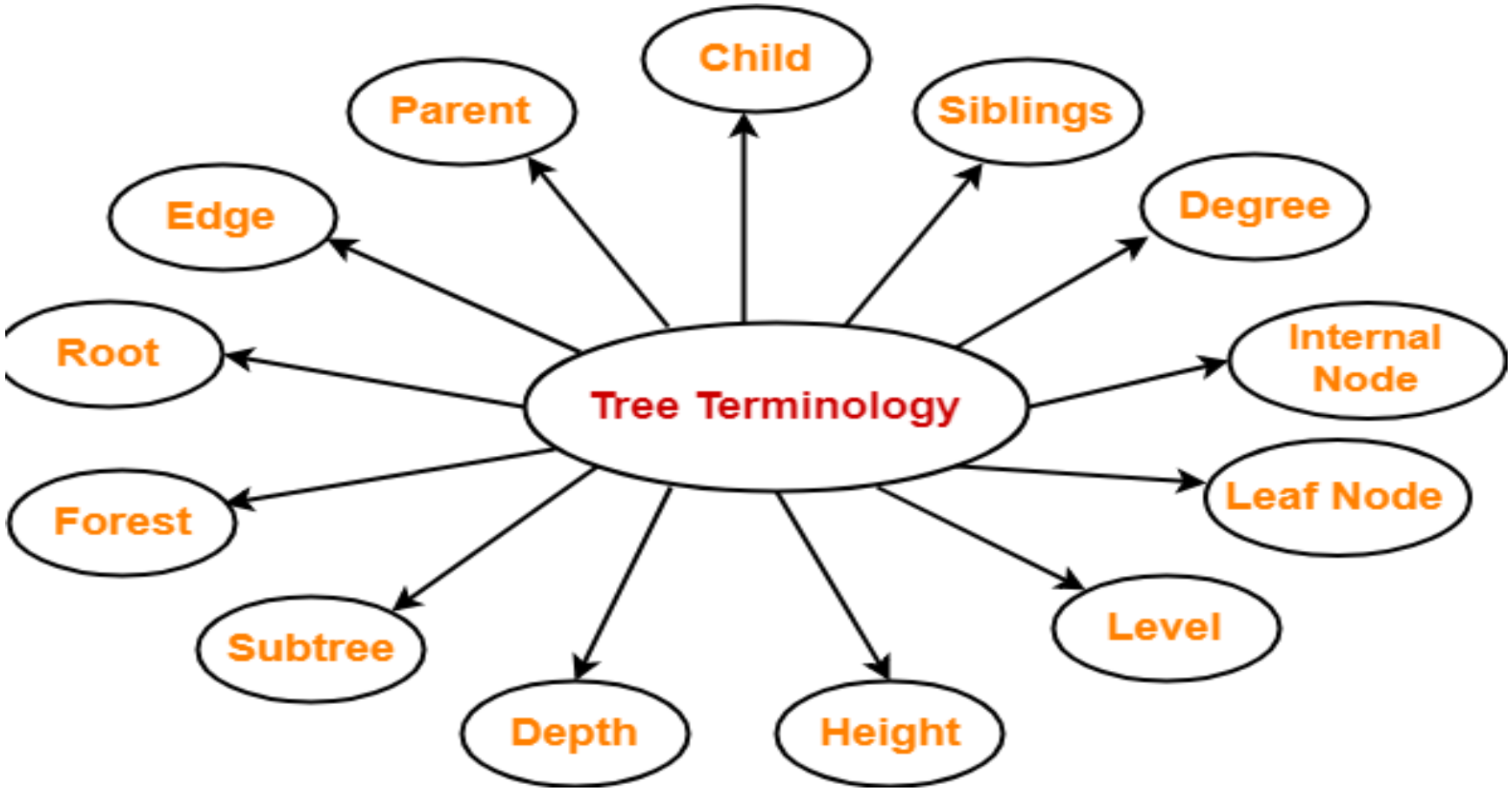


This graph is not a Tree



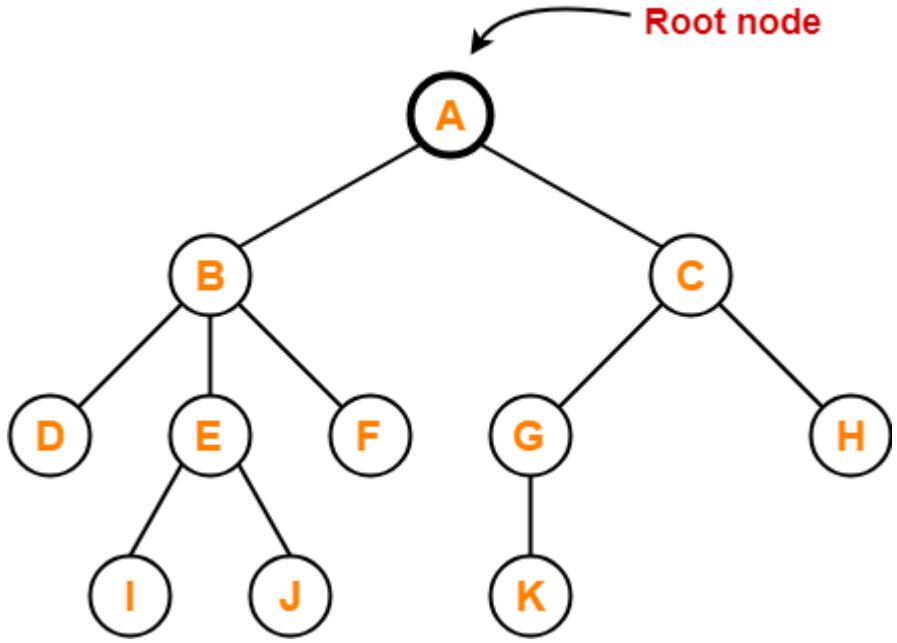
This graph is a Tree

Tree Terminology



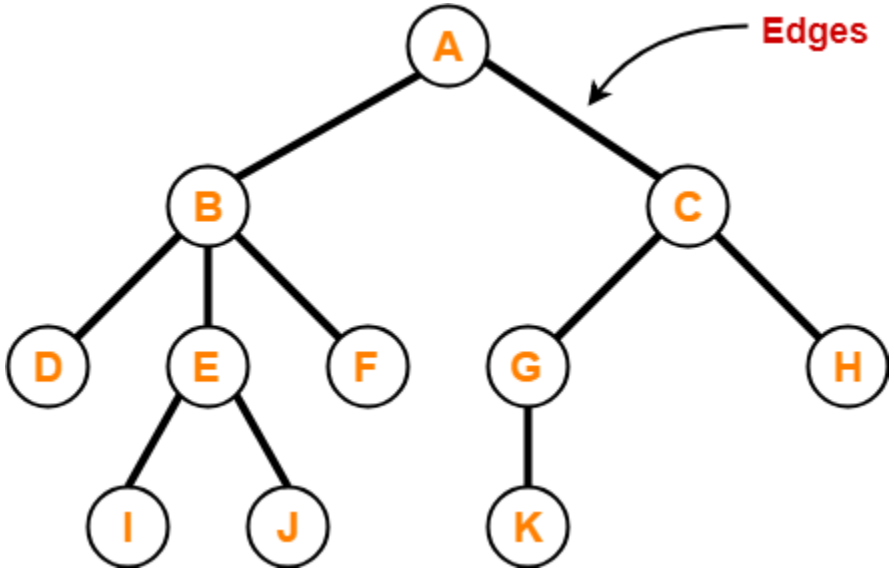
Root

- 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

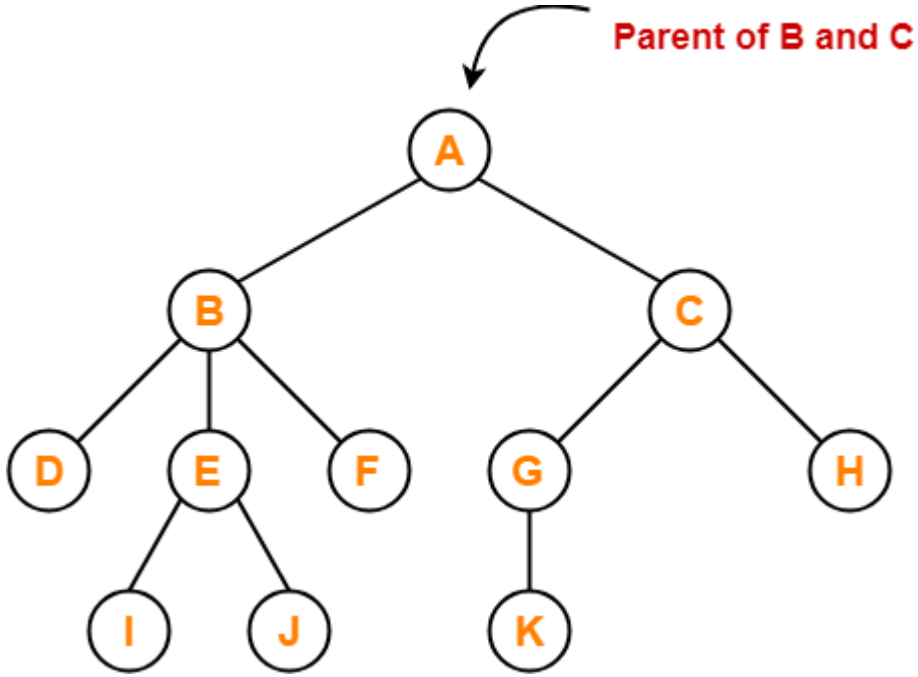
- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.



Parent Node

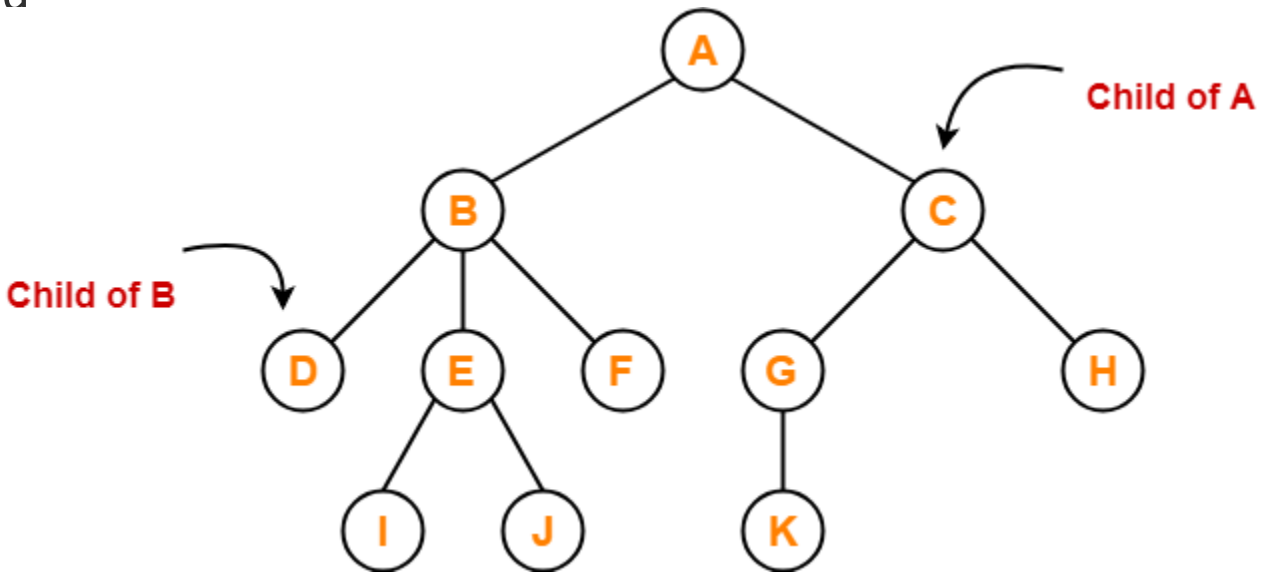
•The node which has a branch from it to any other node is called as a **parent node**.
•In other words, the node which has one or more children is called as a parent node.
•In a tree, a parent node can have any number of child nodes.

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.

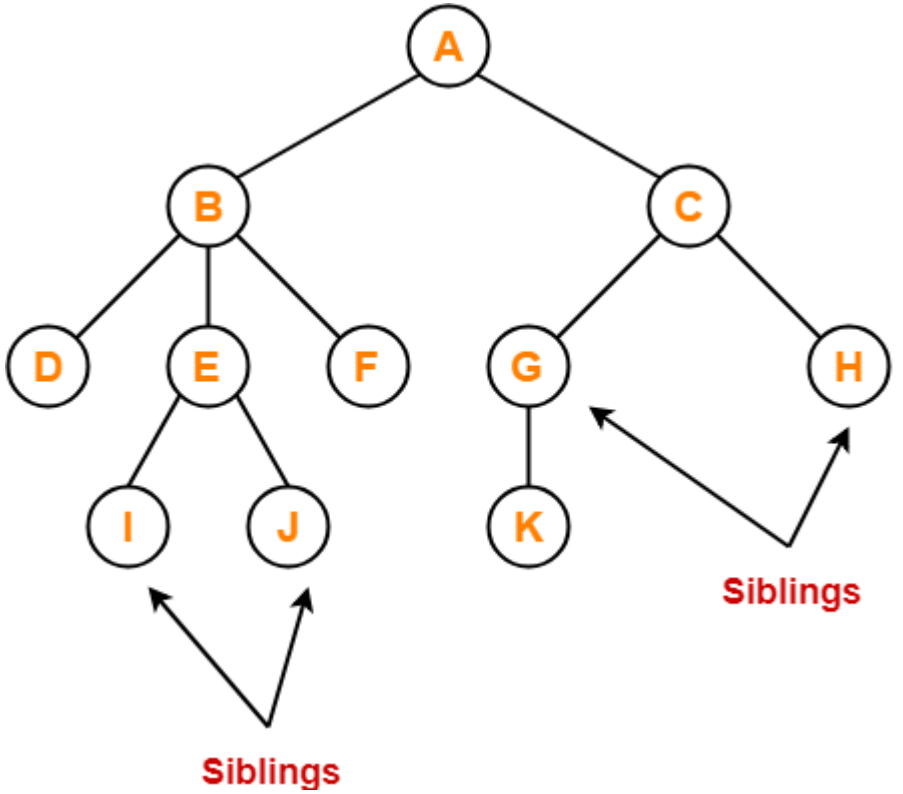


Here,
•Nodes B and C are the children of node A
•Nodes D, E and F are the children of node B
•Nodes G and H are the children of node C
•Nodes I and J are the children of node E
•Node K is the child of node G

Siblings

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

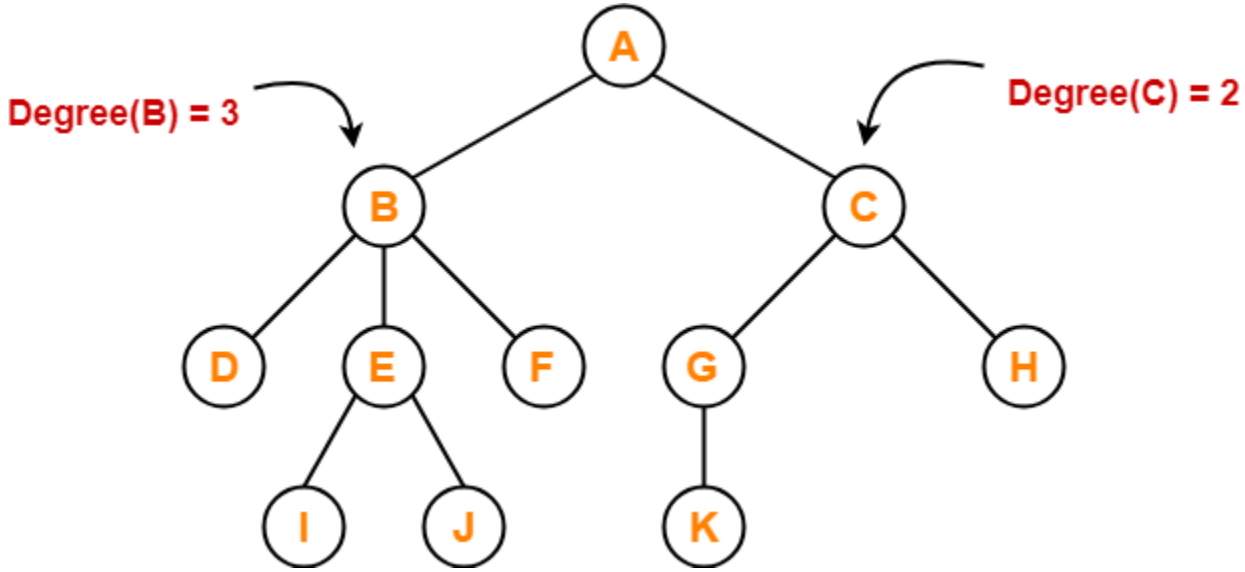
Here,
•Nodes B and C are siblings
•Nodes D, E and F are siblings
•Nodes G and H are siblings
•Nodes I and J are siblings



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.

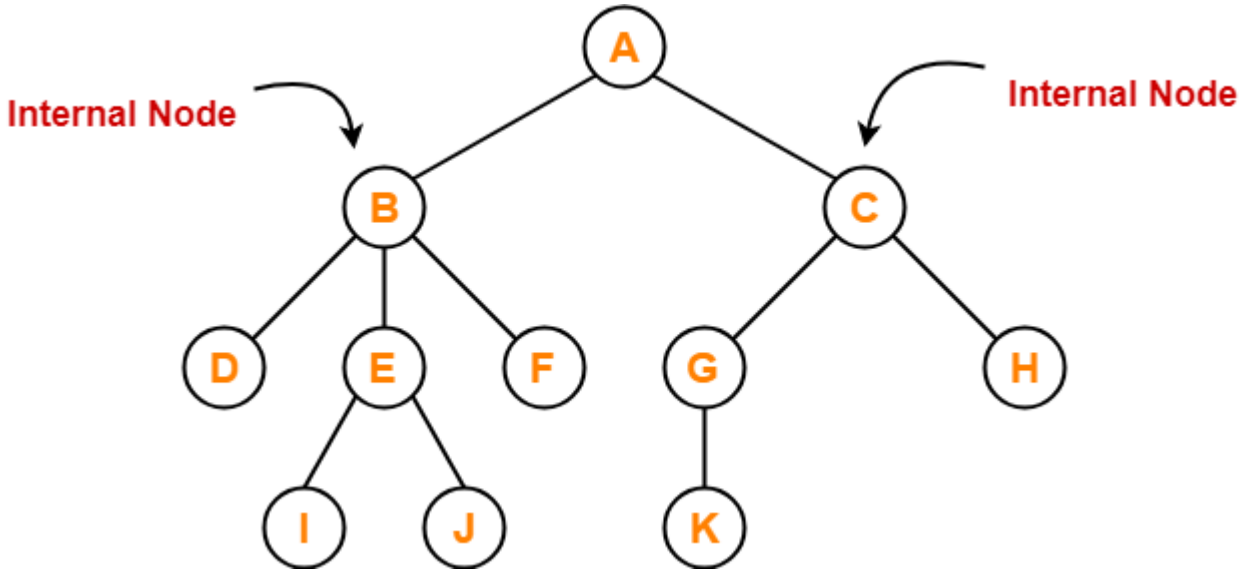
Here,
•Degree of node A = 2
•Degree of node B = 3
•Degree of node C = 2
•Degree of node D = 0
•Degree of node E = 2
•Degree of node F = 0
•Degree of node G = 1
•Degree of node H = 0
•Degree of node I = 0
•Degree of node J = 0
•Degree of node K = 0



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.

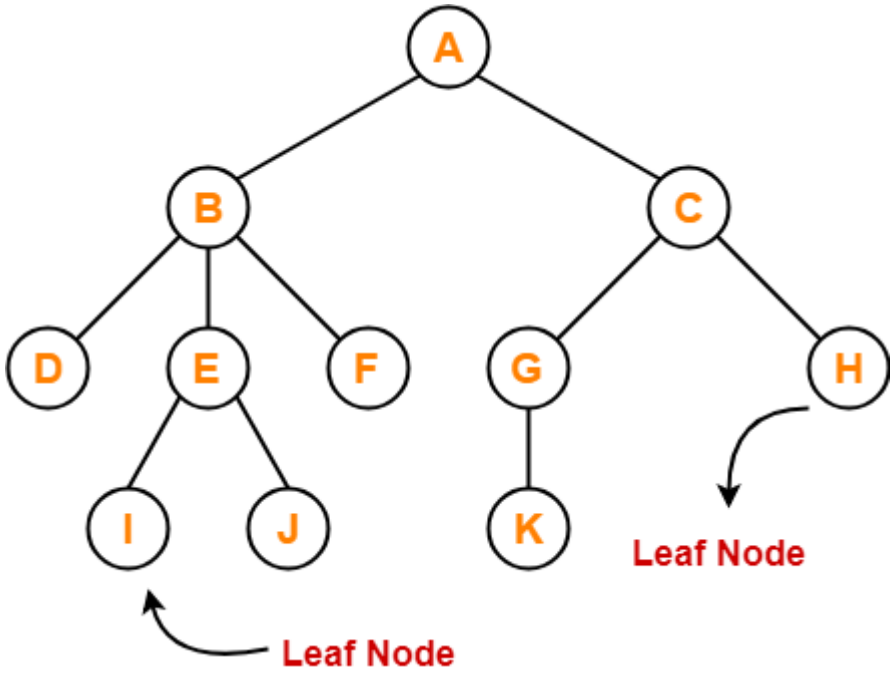
Here, nodes A, B, C, E and G are internal nodes.



Leaf Node

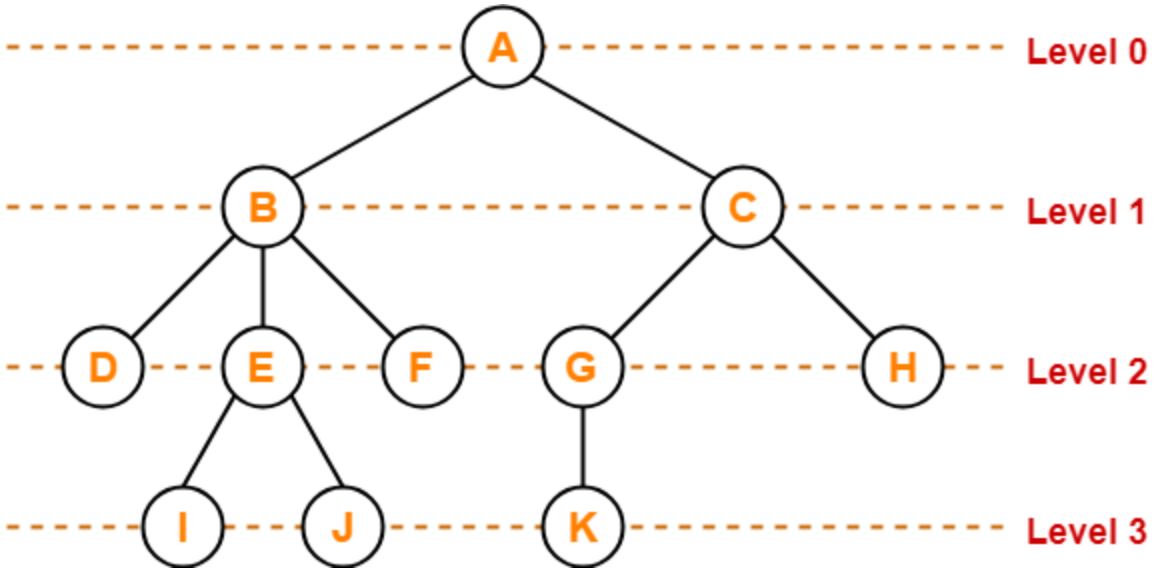
•The node which does not have any child is called as a **leaf node**.
•Leaf nodes are also called as **external nodes** or **terminal nodes**.

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



Level

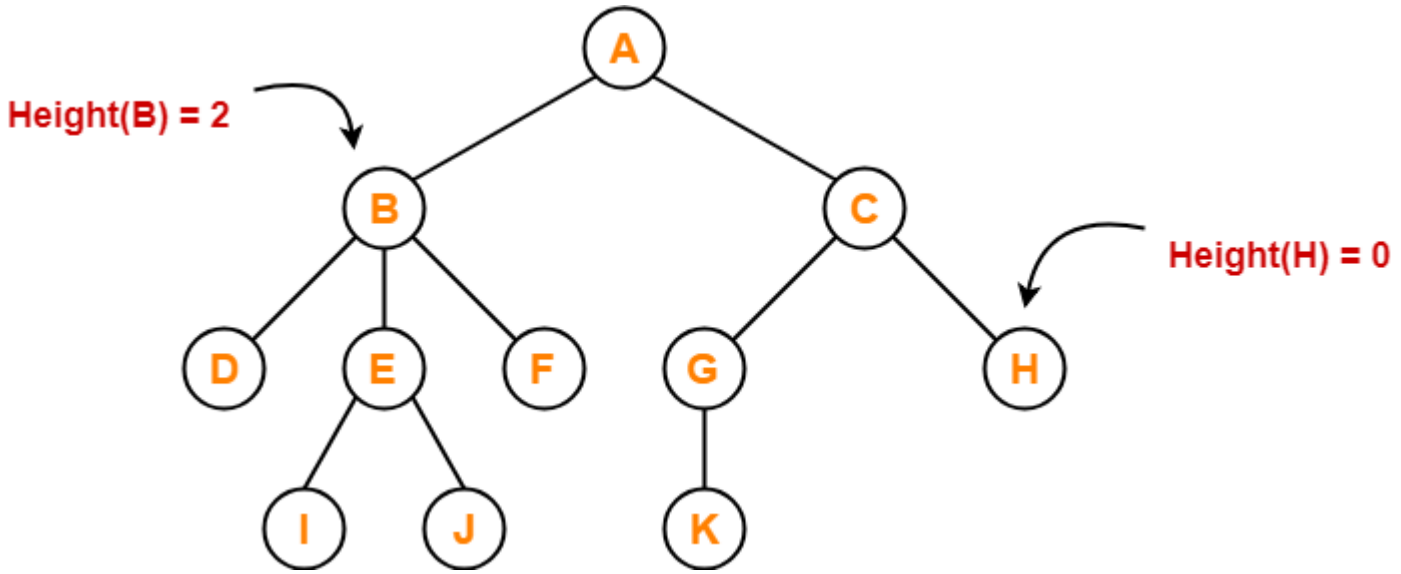
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



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

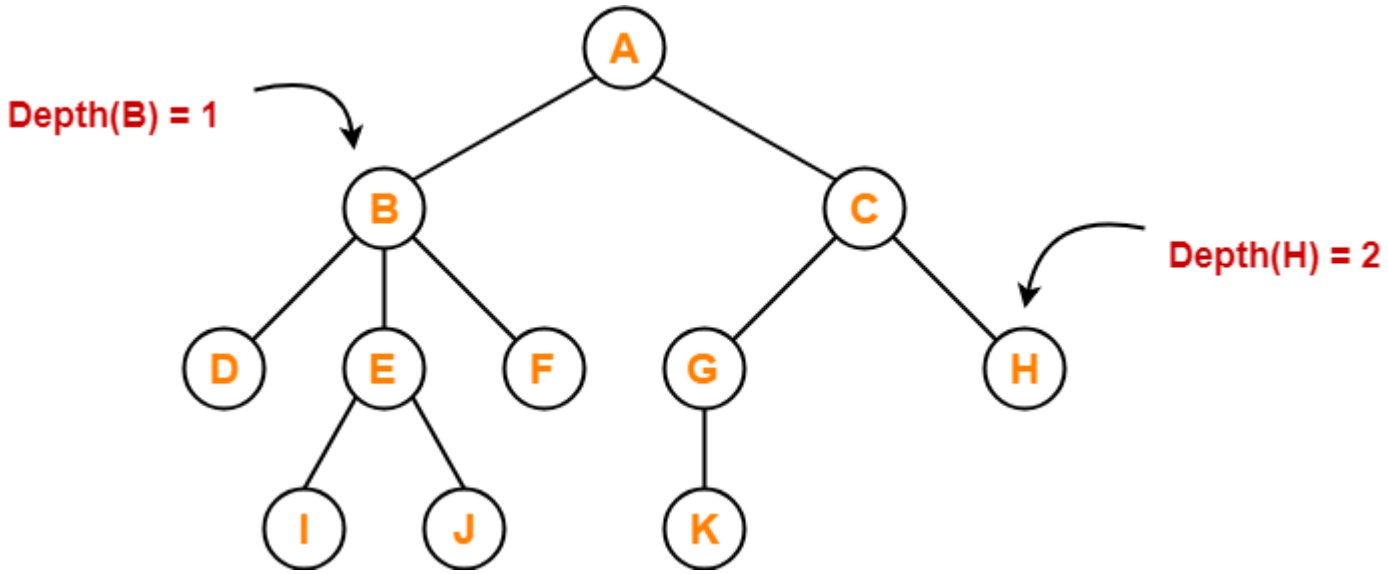
Here,
•Height of node A = 3
•Height of node B = 2
•Height of node C = 2
•Height of node D = 0
•Height of node E = 1
•Height of node F = 0
•Height of node G = 1
•Height of node H = 0
•Height of node I = 0
•Height of node J = 0
•Height of node K = 0



Depth

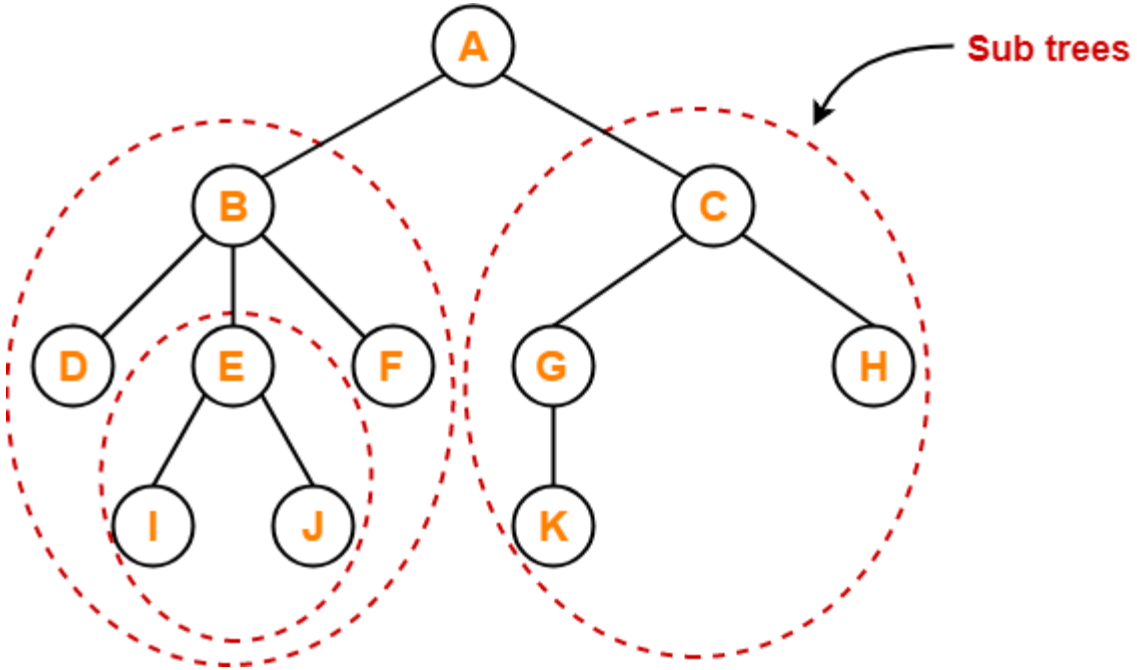
•Total number of edges from root node to a particular node is called as **depth of that node**.
•**Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
•Depth of the root node = 0
•The terms “level” and “depth” are used interchangeably.

Here,
•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



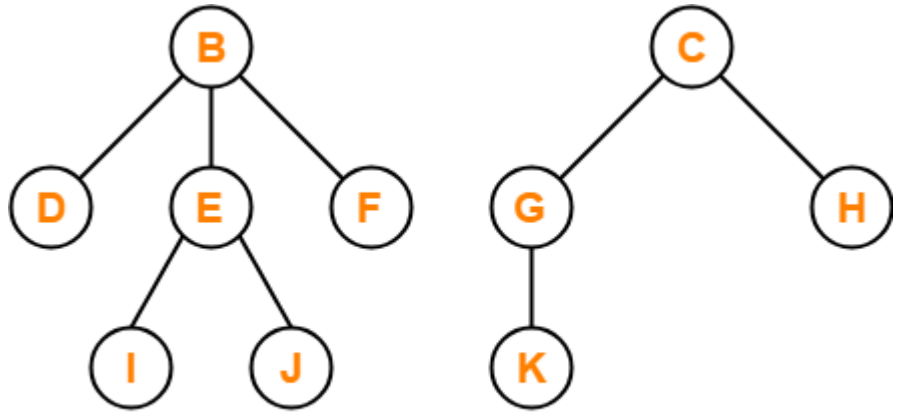
Subtree

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



Forest

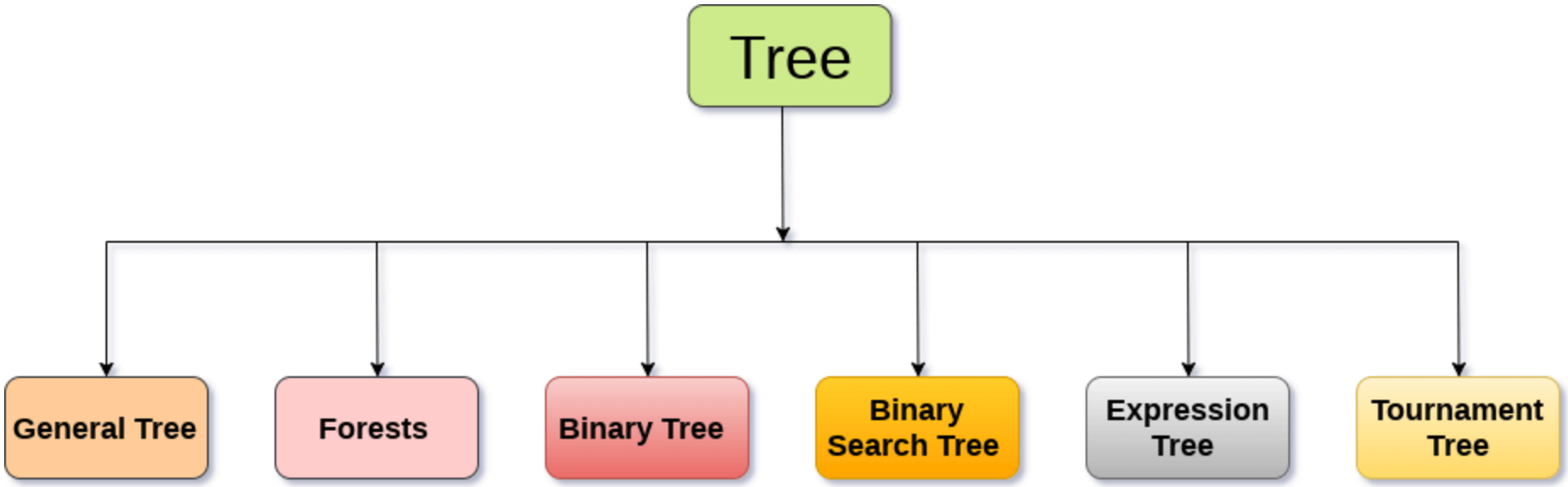
A forest is a set of disjoint trees.



Forest

Types of Tree

The tree data structure can be classified into six different categories.

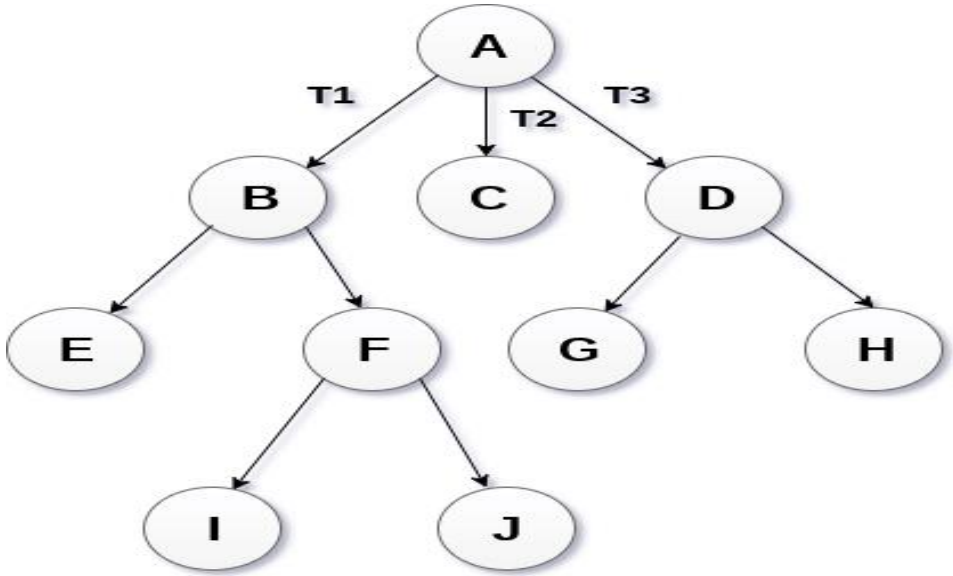


General Tree

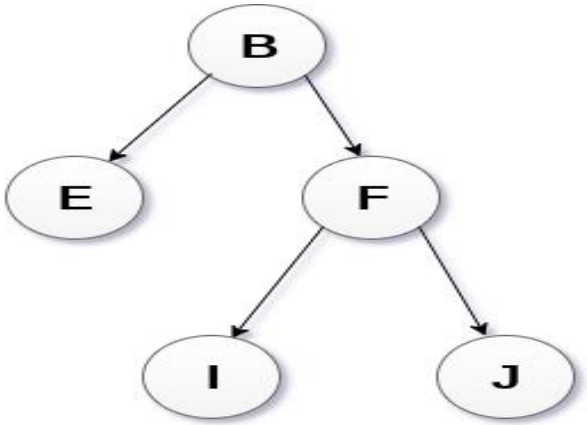
- General Tree stores the elements in a hierarchical order in which the top level element is always present at level 0 as the root element.
- All the nodes except the root node are present at number of levels.
- The nodes which are present on the same level are called siblings while the nodes which are present on the different levels exhibit the parent-child relationship among them.
- A node may contain any number of sub-trees.
- The tree in which each node contain 3 sub-tree, is called ternary tree.

Forest

Forest can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connects root node to the first level node.

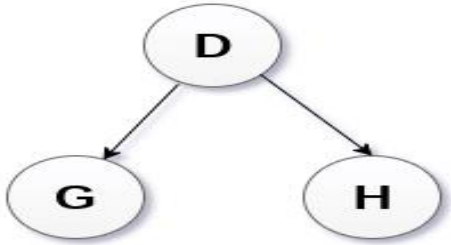


General Tree



Forest 1

Forest 2



Forest 3

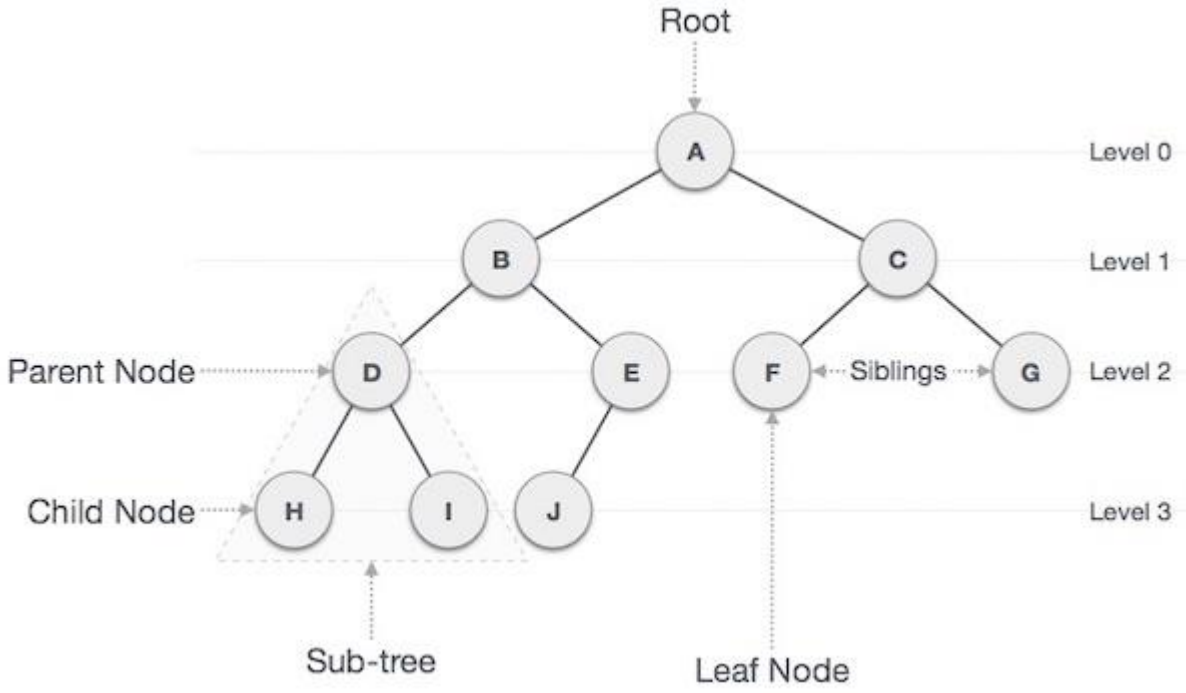
Binary Tree

Binary tree is a data structure in which each node can have at most 2 children.

The node present at the top most level is called the root node.

A node with the 0 children is called leaf node.

Binary Trees are used in the applications like expression evaluation and many more.

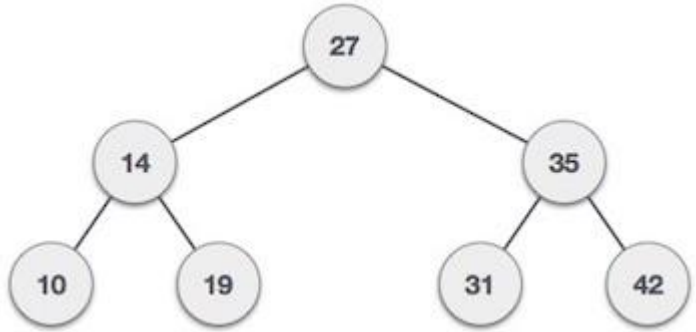


Binary Search Tree

Binary search tree is an ordered binary tree.

All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element.

Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.



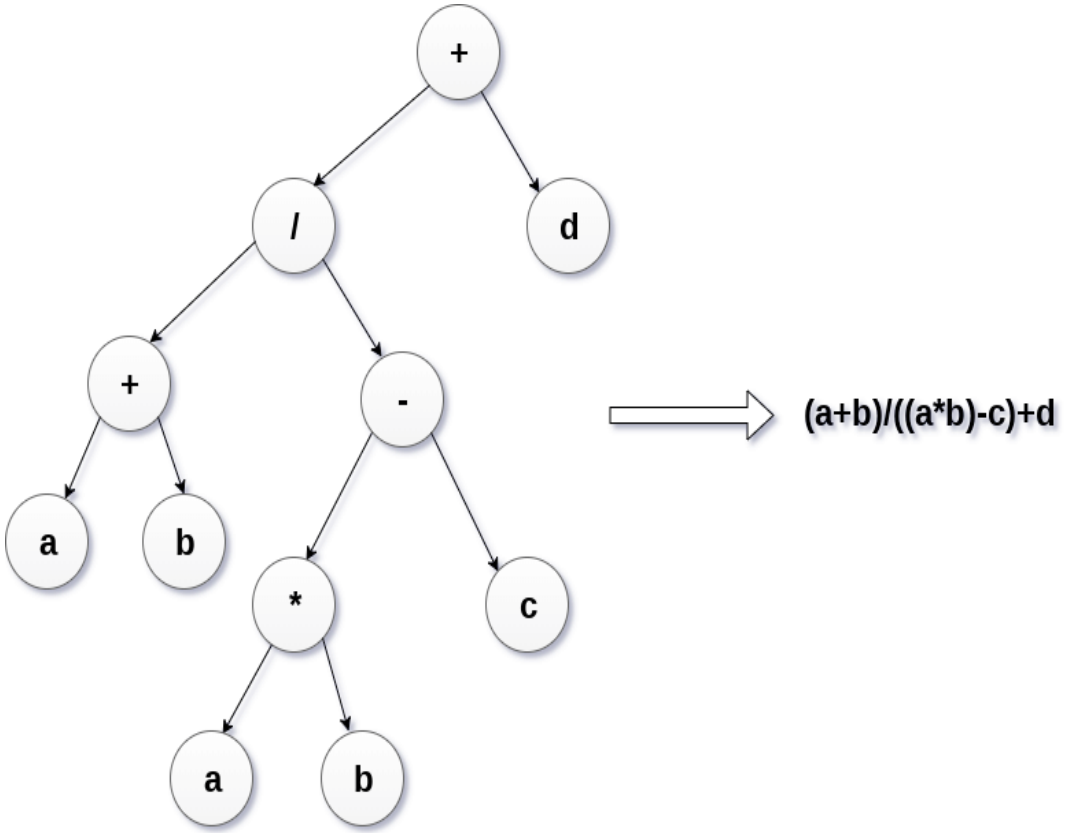
Expression Tree

Expression trees are used to evaluate the simple arithmetic expressions.

Expression tree is basically a binary tree where internal nodes are represented by operators while the leaf nodes are represented by operands.

Expression trees are widely used to solve algebraic expressions like (a+b)*(a-b).

Consider the following example.
Q. Construct an expression tree by using the following algebraic expression.
(a + b) / (a*b - c) + d



Tournament Tree

Tournament tree are used to record the winner of the match in each round being played between two players.

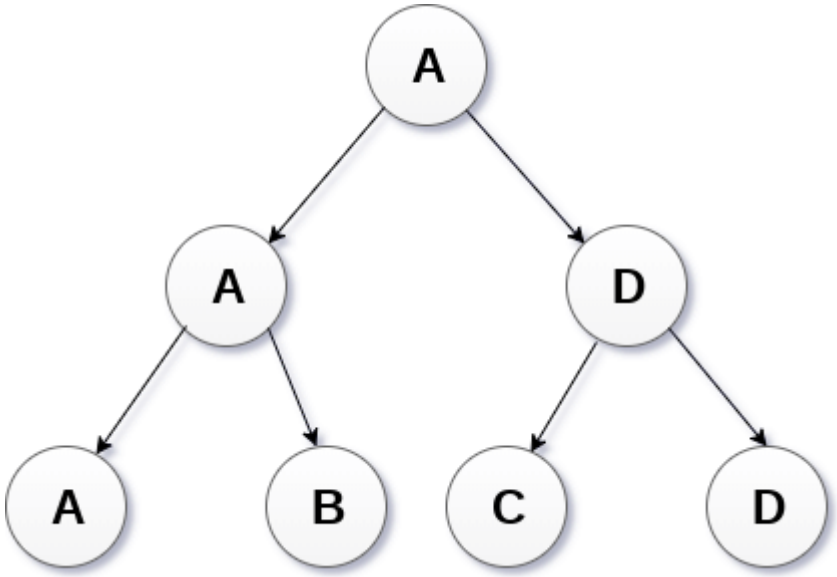
Tournament tree can also be called as selection tree or winner tree.

External nodes represent the players among which a match is being played while the internal nodes represent the winner of the match played.

At the top most level, the winner of the tournament is present as the root node of the tree.

For example, tree of a chess tournament being played among 4 players is shown as follows.

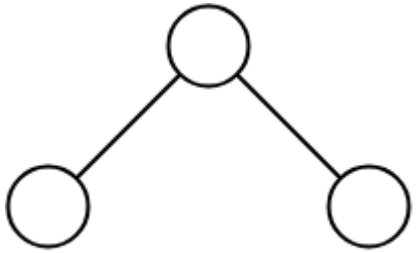
However,
the winner in the left sub-tree will play against the winner of right sub-tree.



Binary Tree in Detail

Unlabelled Binary Tree

A binary tree is unlabelled if its nodes are not assigned any label.



Unlabeled Binary Tree

Number of different Binary Trees possible with 'n' unlabeled nodes

$$= \frac{{}^{2n}C_n}{n+1}$$

Consider we want to draw all the binary trees possible with 3 unlabeled nodes.
Using the formula, we have-

Number of binary trees possible with 3 unlabeled nodes

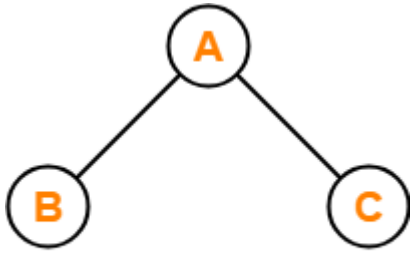
$= {}^{2 \times 3}C_3 / (3 + 1)$

$= {}^{6}C_3 / 4$

$= 5$

Labelled Binary Tree

A binary tree is labelled if all its nodes are assigned a label.



Labeled Binary Tree

Number of different Binary Trees possible with 'n' labeled nodes

$$= \frac{{}^{2n}C_n}{n+1} \times n!$$

Consider we want to draw all the binary trees possible with 3 labeled nodes.
Using the formula, we have-

Number of binary trees possible with 3 labeled nodes

$= \{ {}^{2 \times 3}C_3 / (3 + 1) \} \times 3!$

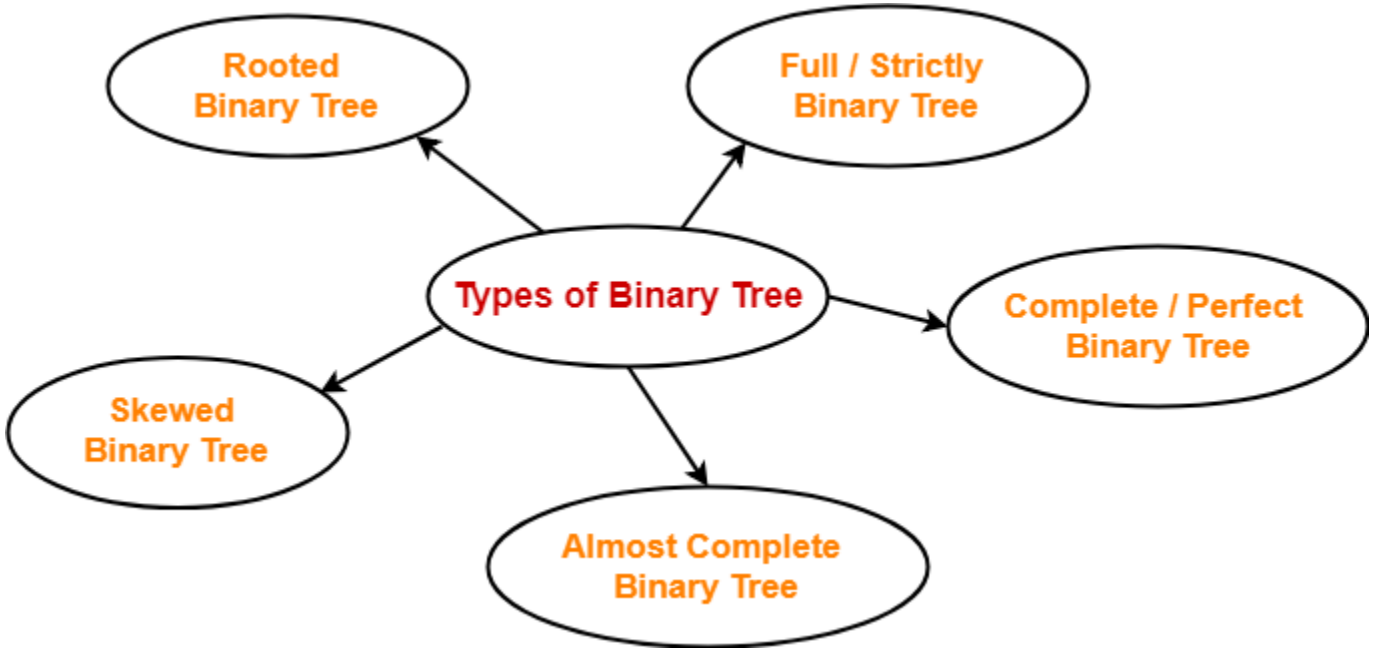
$= \{ {}^{6}C_3 / 4 \} \times 6$

$= 5 \times 6$

$= 30$

Types of Binary Tree

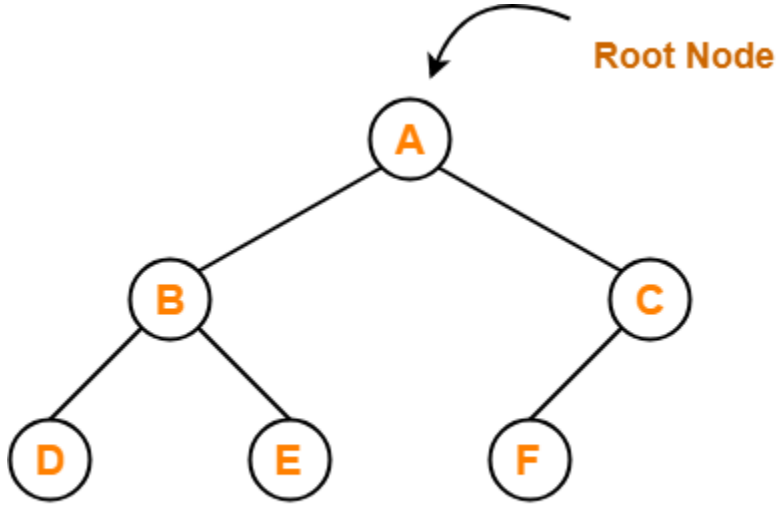
Binary trees can be of the following types-



Rooted Binary Tree

A **rooted binary tree** is a binary tree that satisfies the following 2 properties-

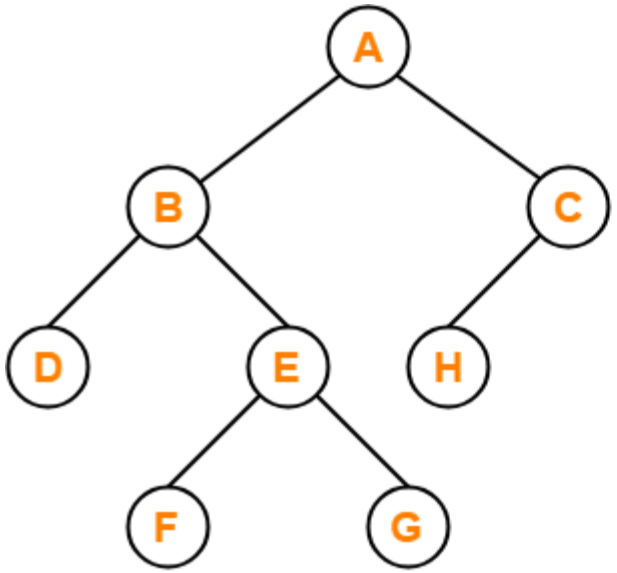
- It has a root node.
- Each node has at most 2 children.

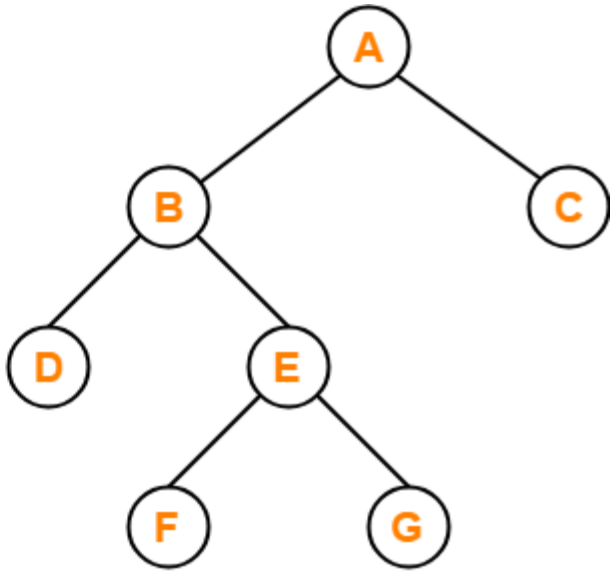


Rooted Binary Tree

Strictly Binary Tree

•A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
•Full binary tree is also called as **Strictly binary tree**.



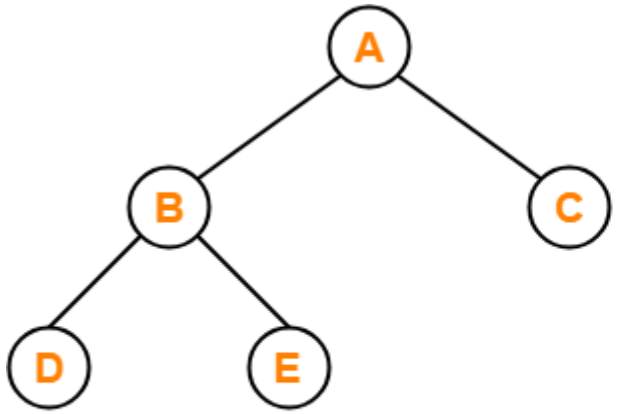


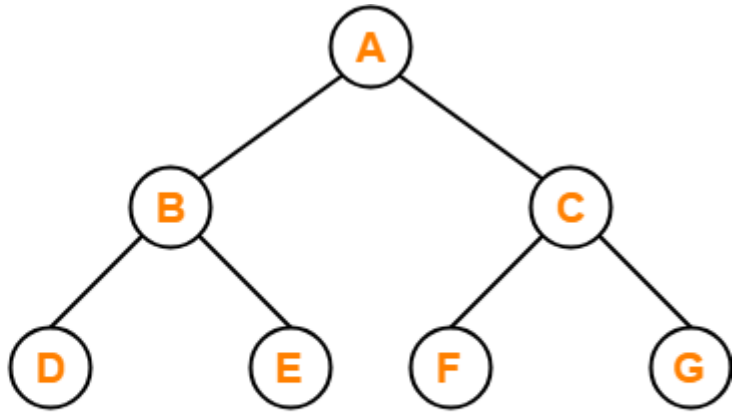
Perfect Binary Tree

A **complete binary tree** is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

Complete binary tree is also called as **Perfect binary tree**.

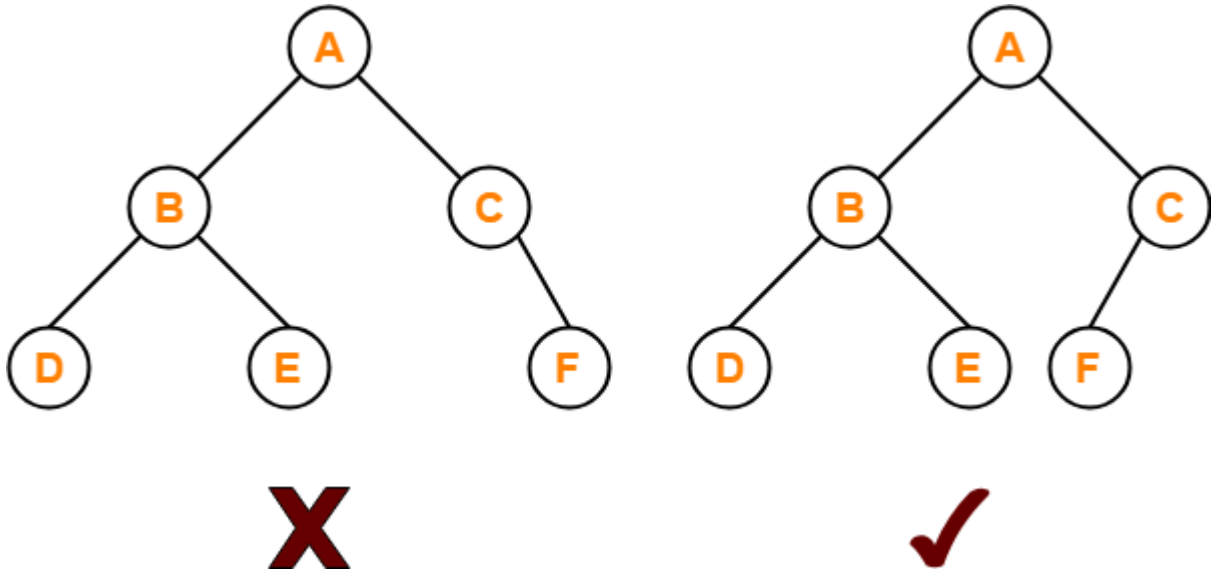




Almost Complete Binary Tree

An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.



Skewed Binary Tree

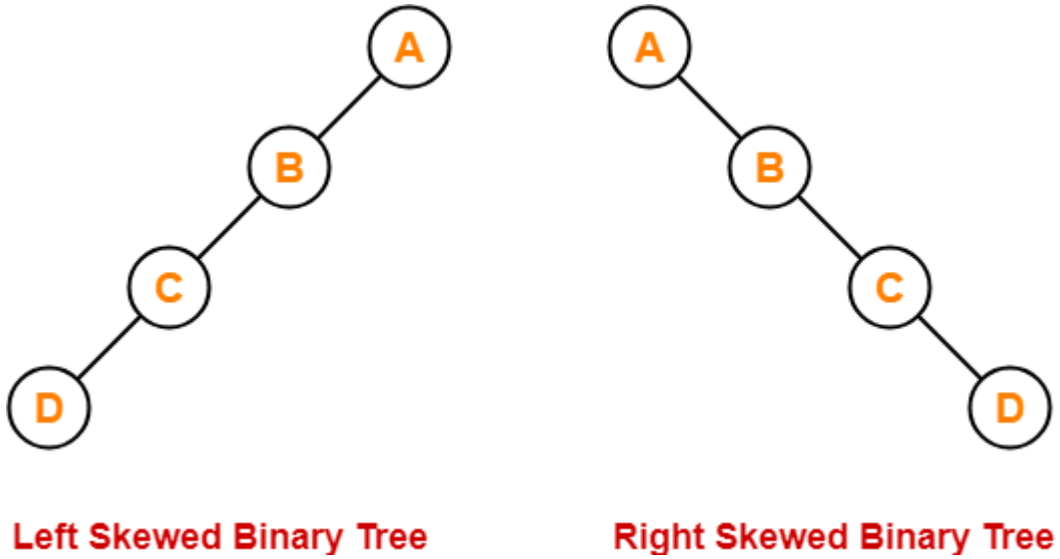
A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

•All the nodes except one node has one and only one child.

•The remaining node has no child.

OR

A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).



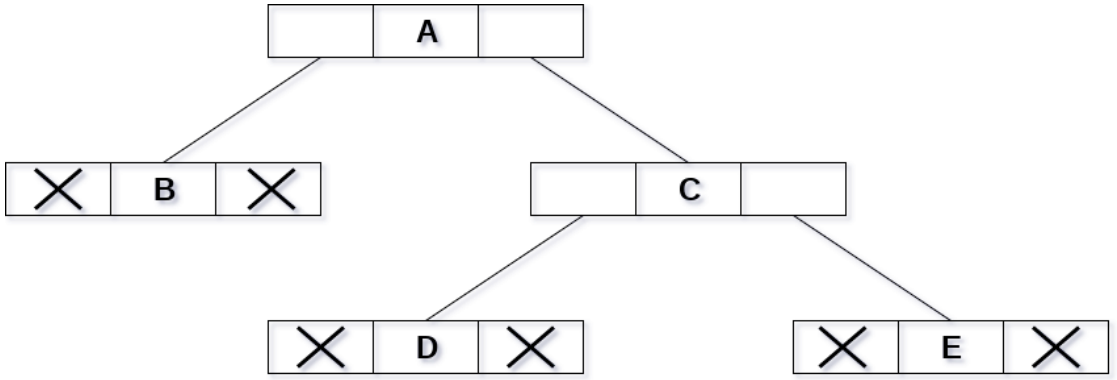
Left Skewed Binary Tree

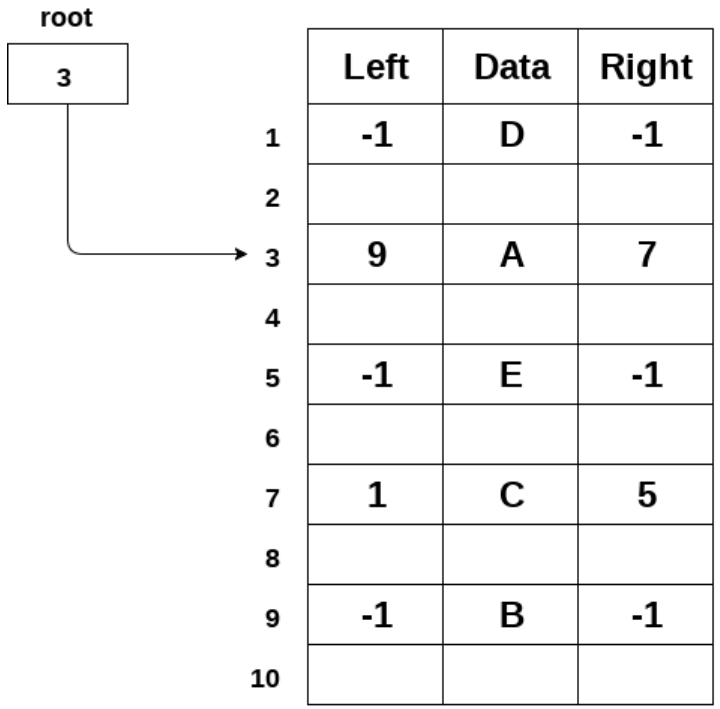
Right Skewed Binary Tree

Binary Tree Representation: Linked List

In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree.

every node contains three parts : pointer to the left node, data element and pointer to the right node.





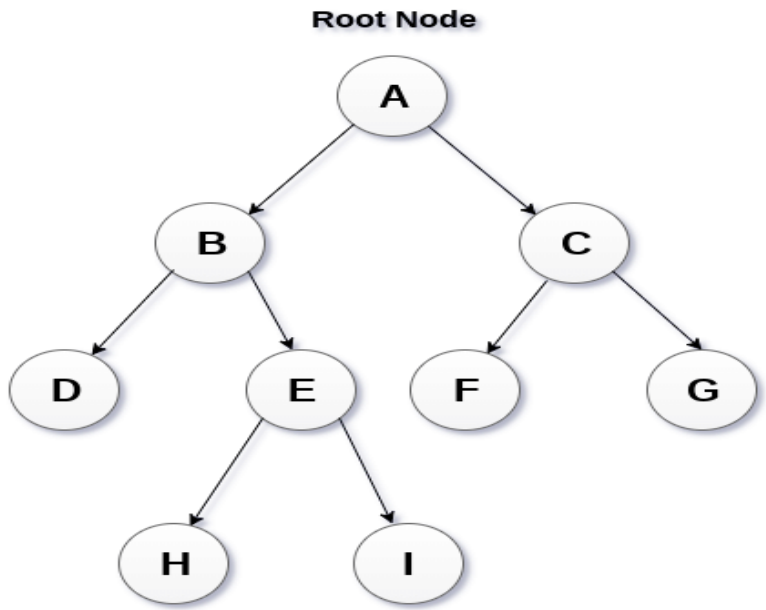
	Left	Data	Right
1	-1	D	-1
2			
3	9	A	7
4			
5	-1	E	-1
6			
7	1	C	5
8			
9	-1	B	-1
10			

Memory Allocation of Binary Tree using linked Representation

Binary Tree Representation: Sequential

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements.

A binary tree is shown in the following figure along with its memory allocation.



A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

Sequential Representation of Binary Tree

Tree Traversal

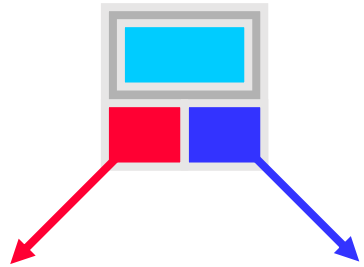
Traversal is a process to visit all the nodes of a tree and may print their values too.

Following three traversal techniques -

1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

The Scenario

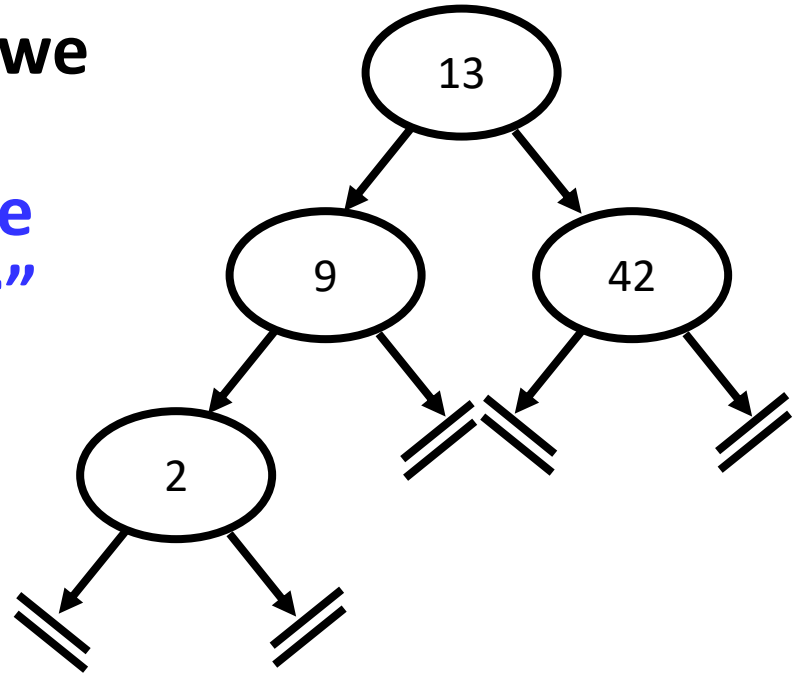
- **Imagine we have a binary tree**
- **We want to traverse the tree**
  - **It's not linear**
  - **We need a way to visit all nodes**



- **Three things must happen:**
  - **Deal with the entire left sub-tree**
  - **Deal with the current node**
  - **Deal with the entire right sub-tree**

Use the Activation Stack

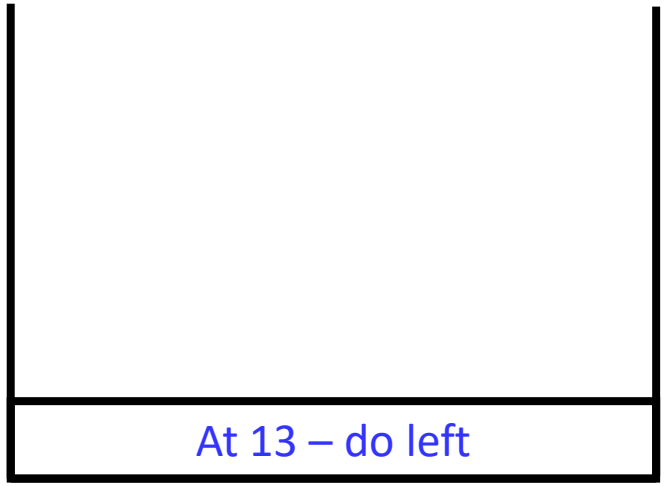
- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

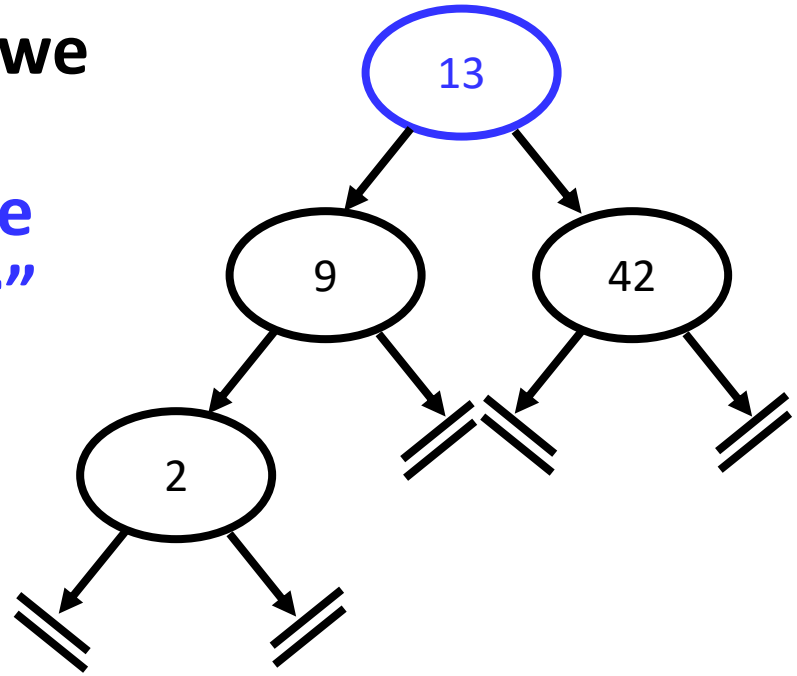


Output:

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

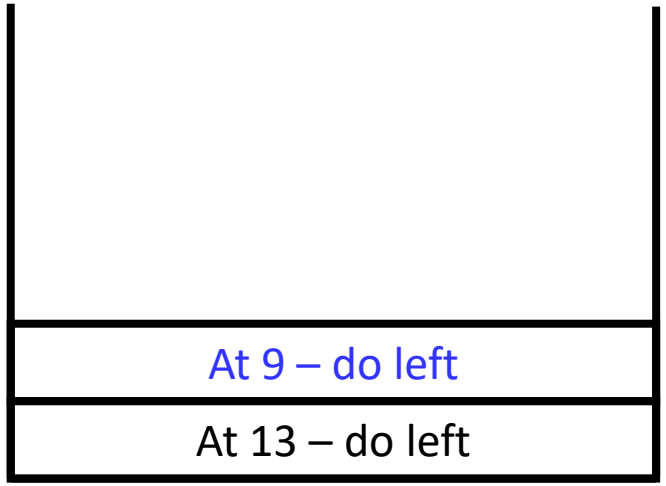


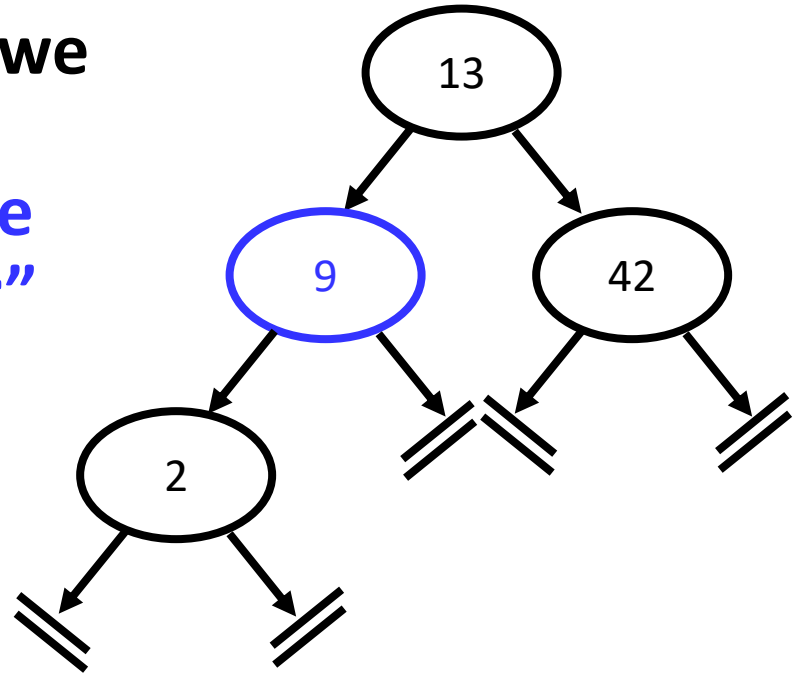


Output:

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

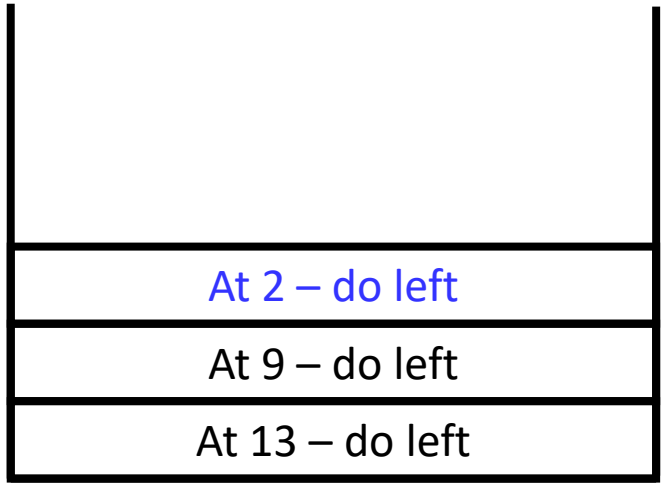


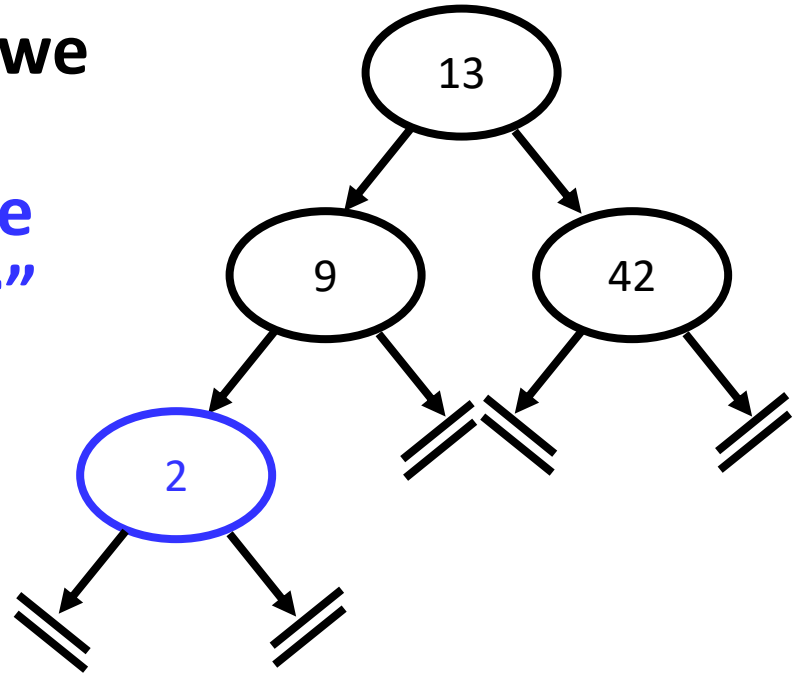


Output:

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

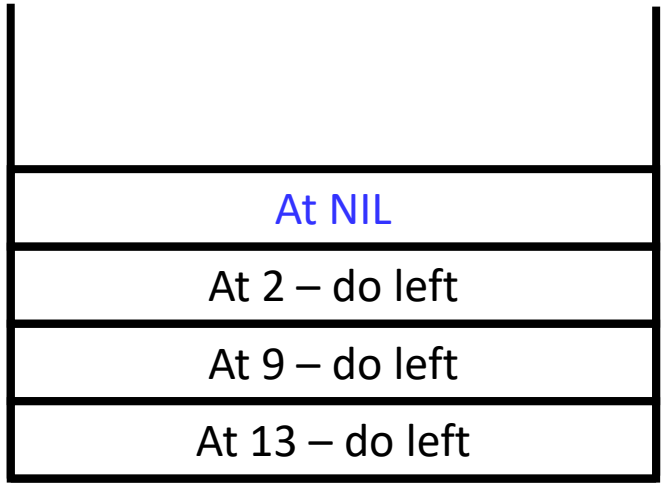


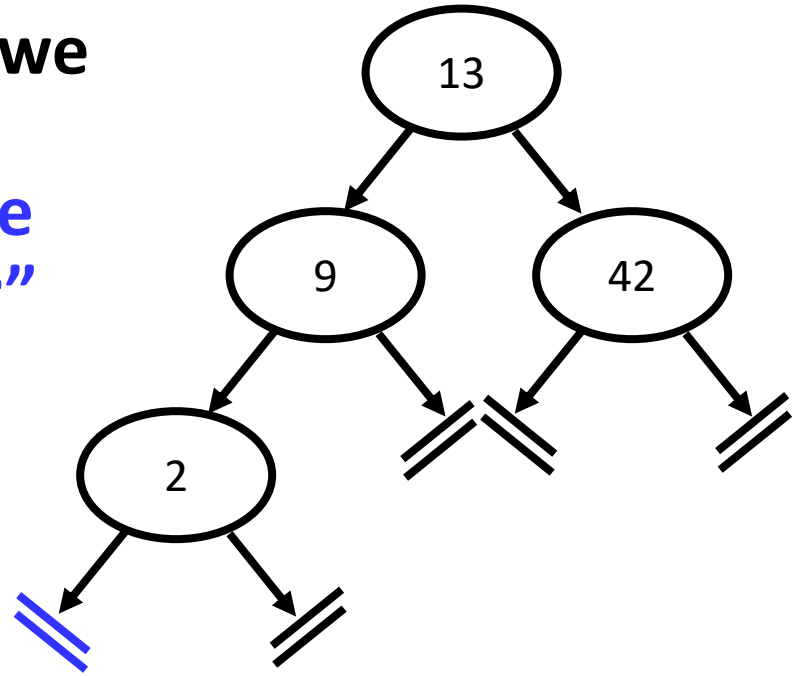


Output:

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

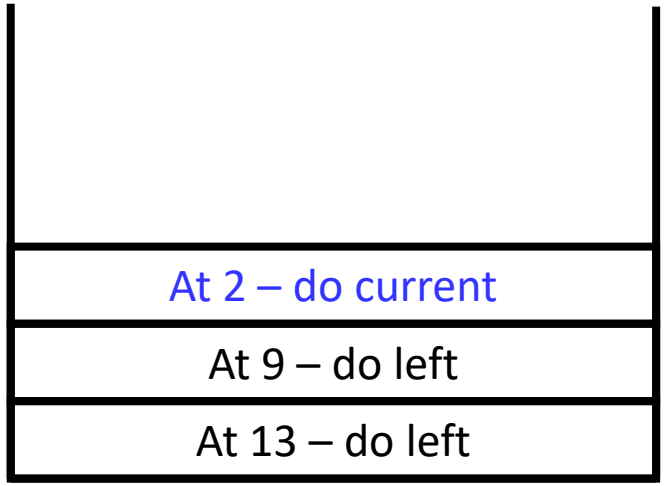


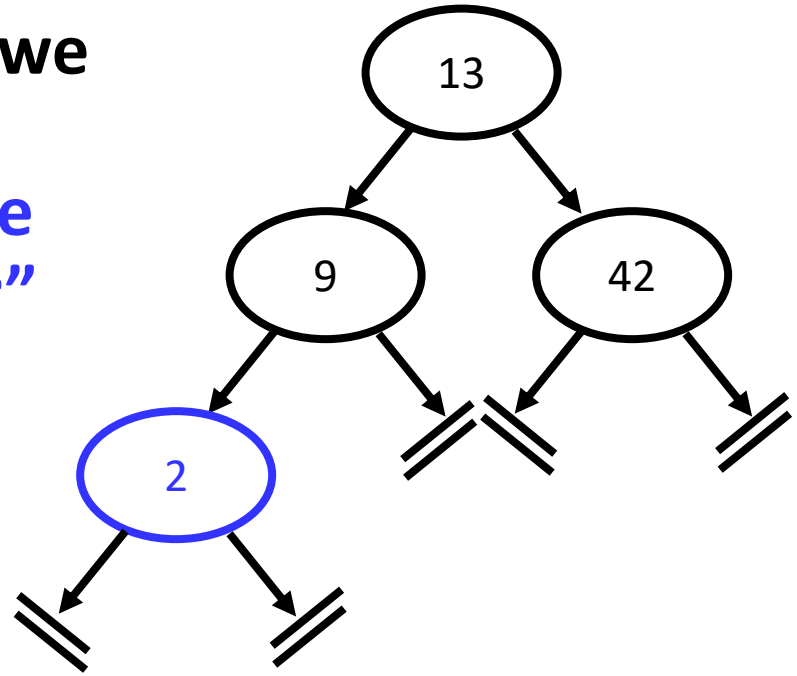


Output:

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

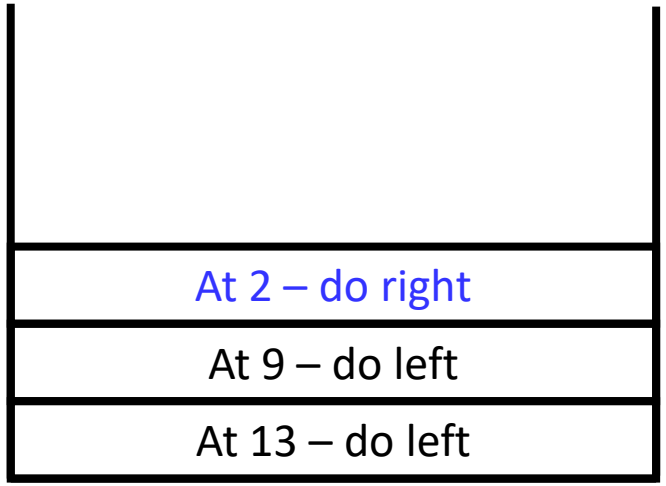


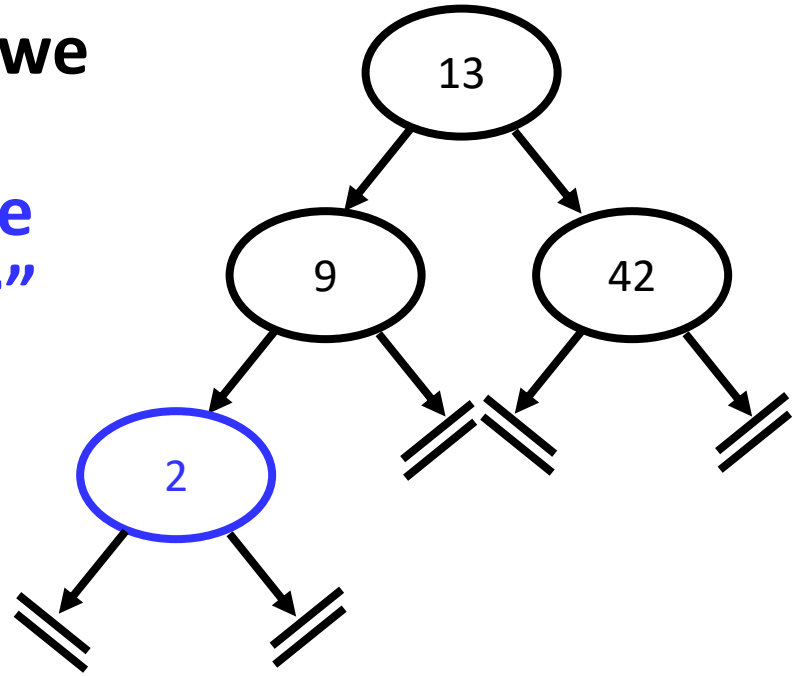


Output: 2

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

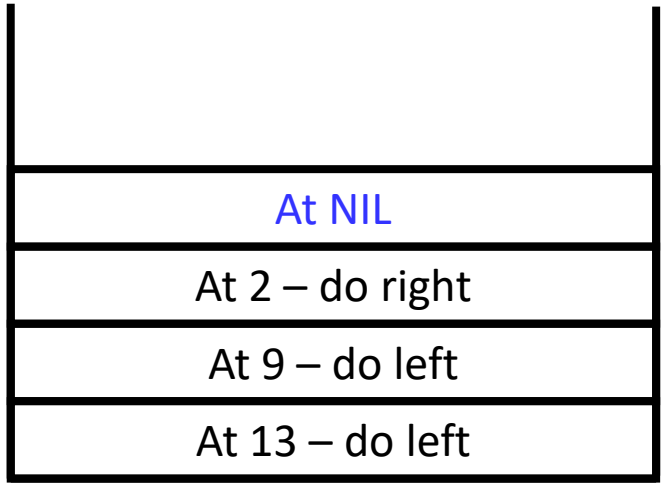


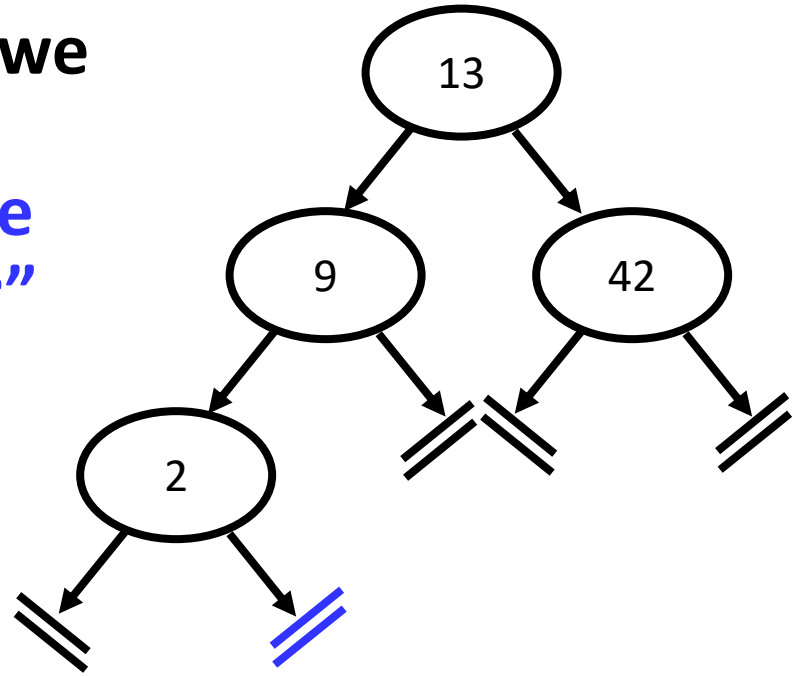


Output: 2

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

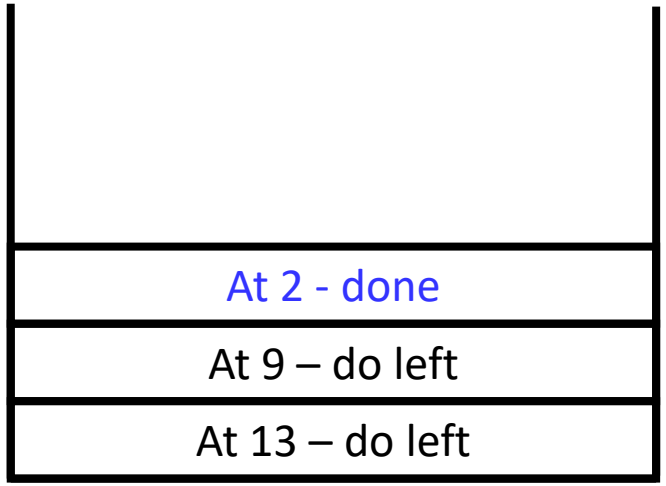


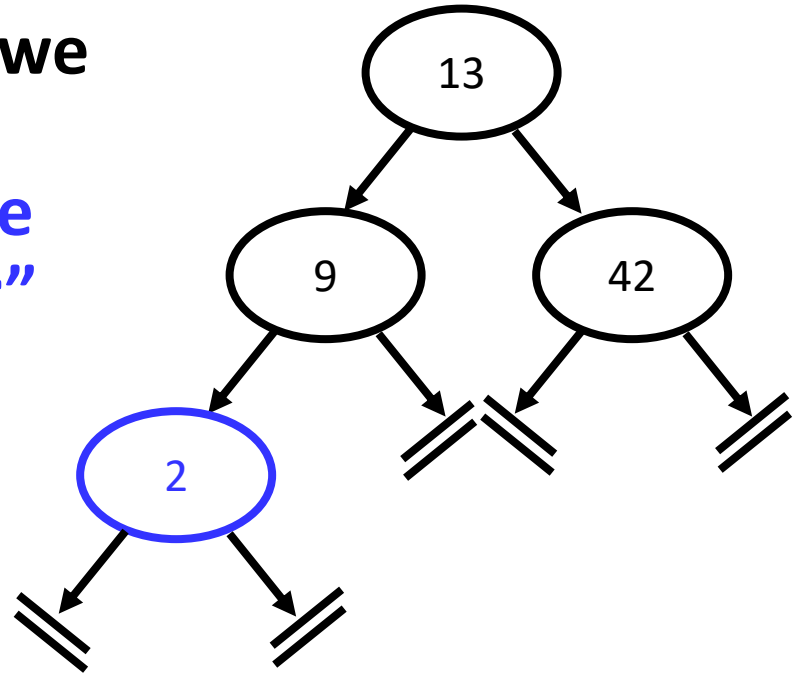


Output: 2

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

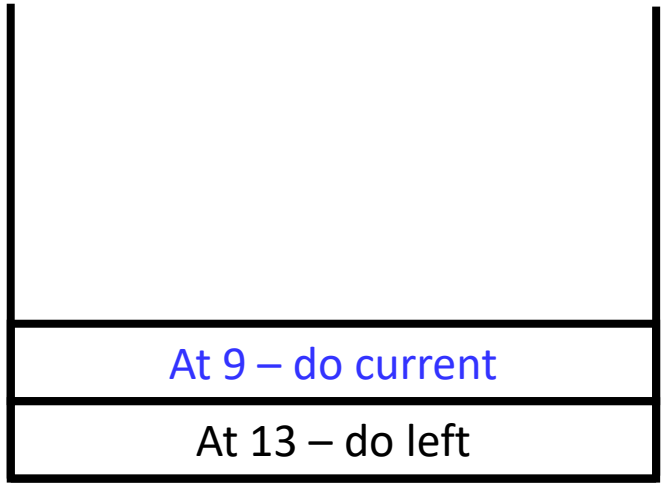


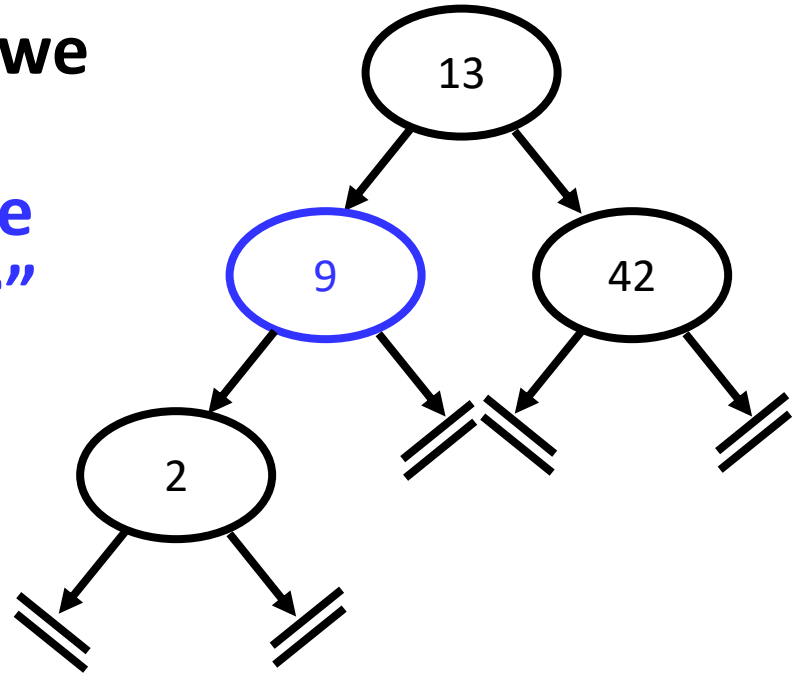


Output: 2

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

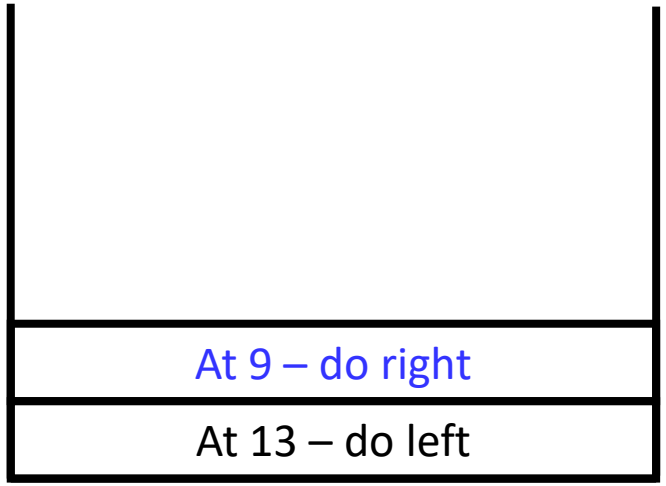


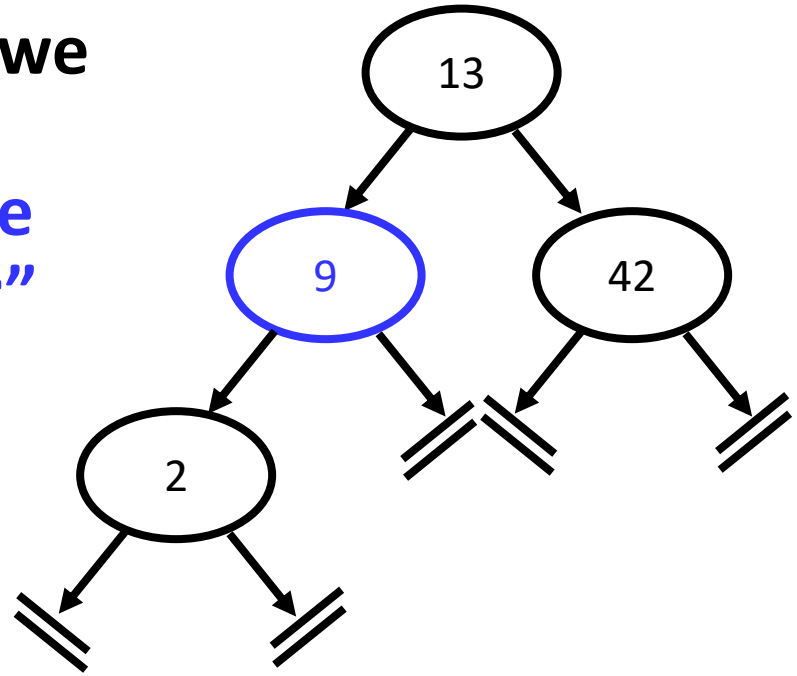


Output: 2 9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

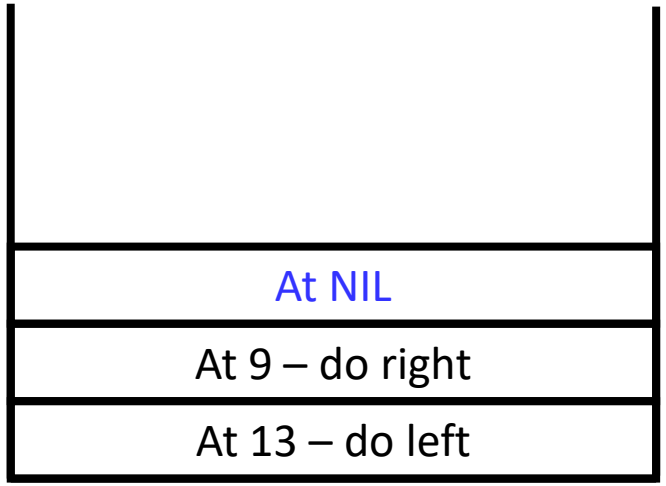


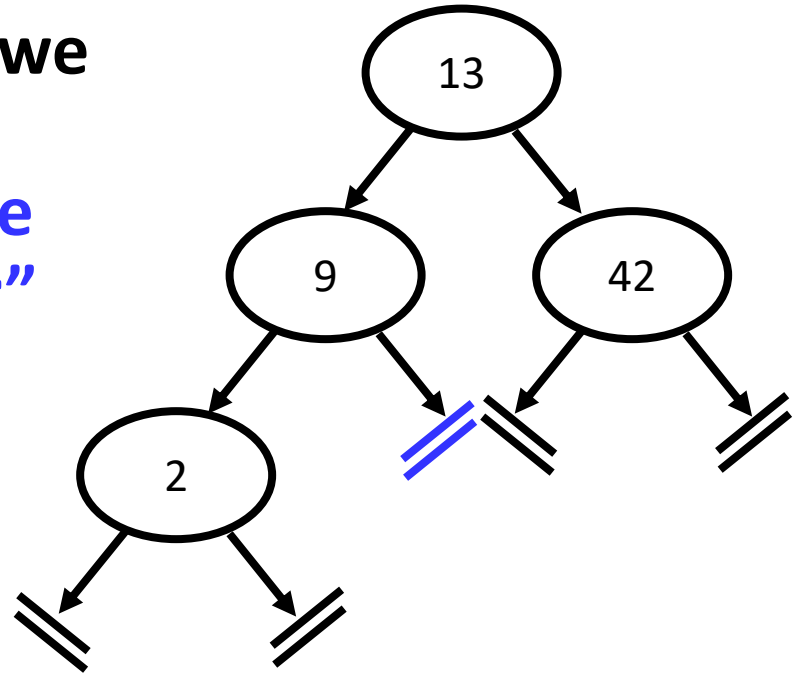


Output: 2 9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

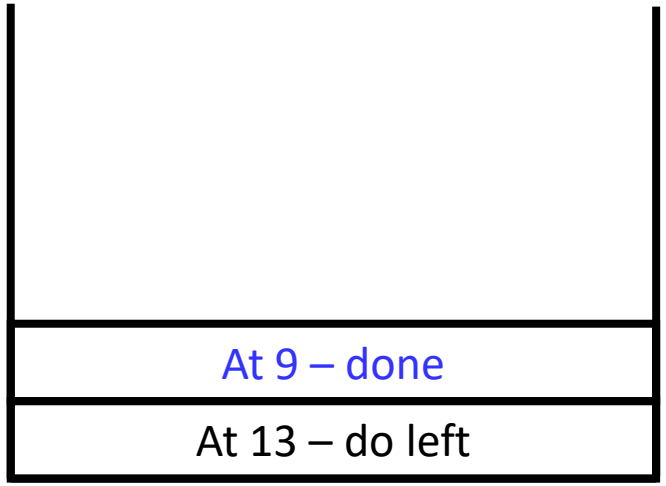


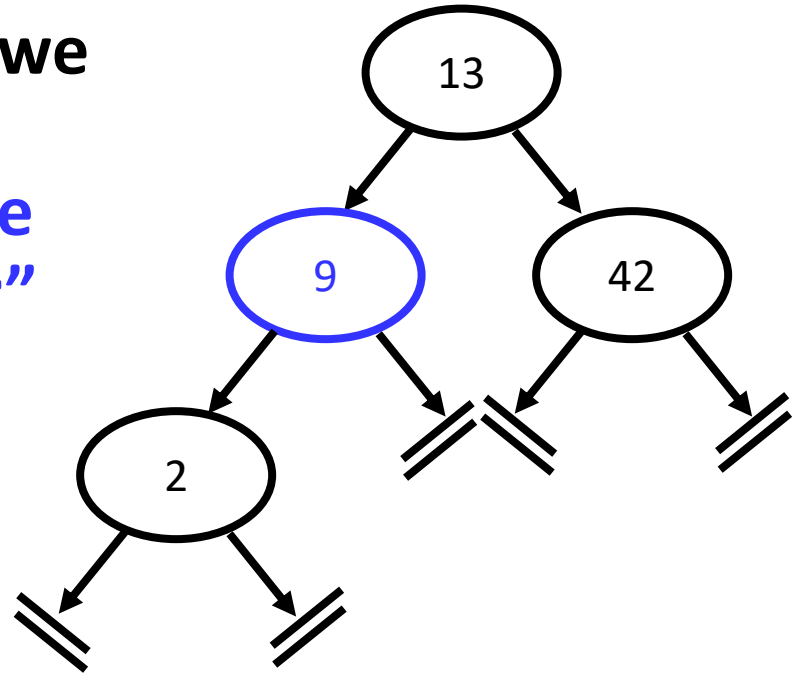


Output: 2 9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

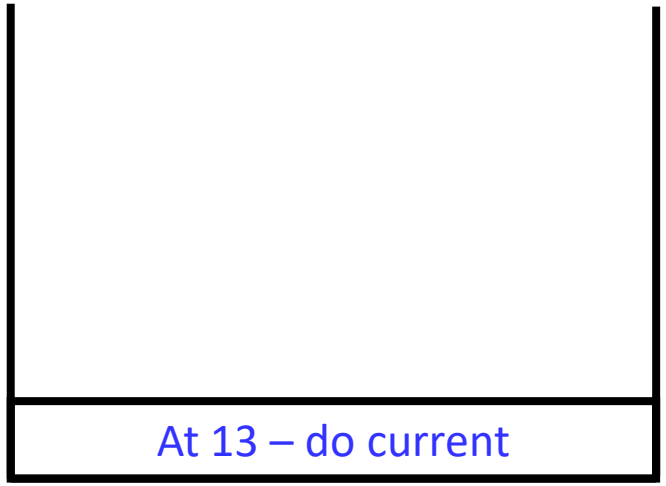


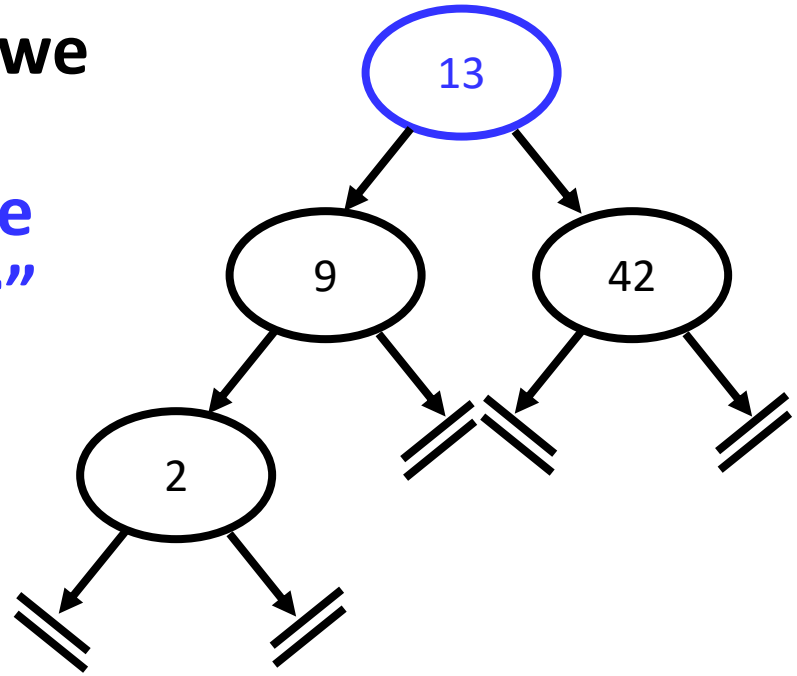


Output: 2 9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

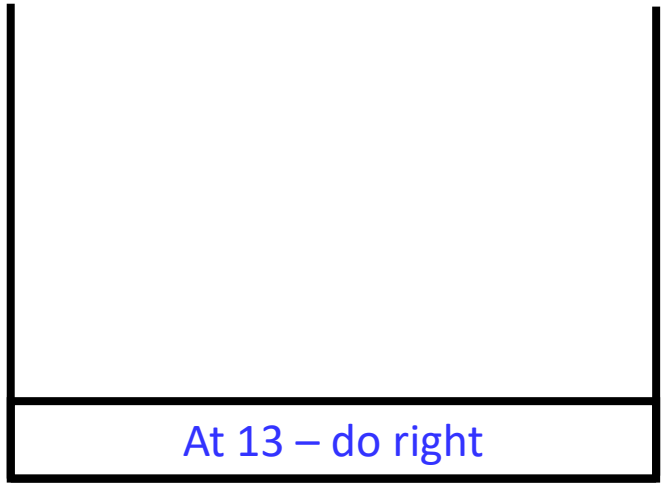


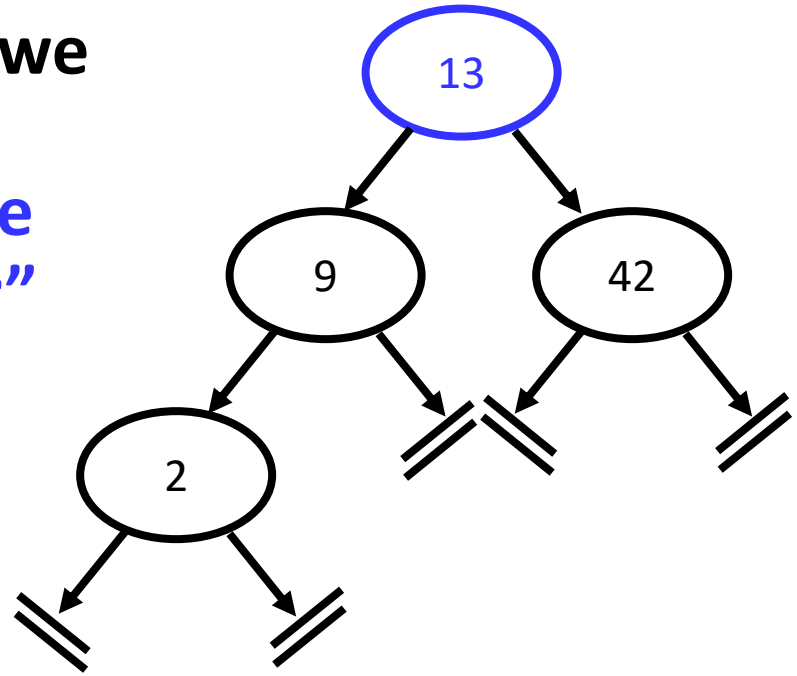


Output: 2 9 13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

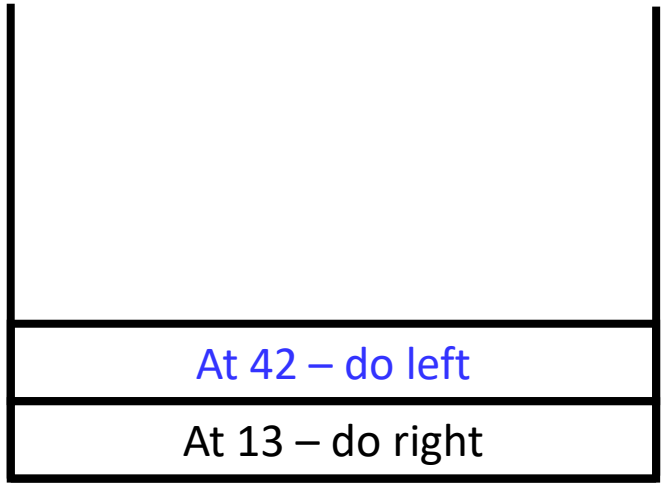


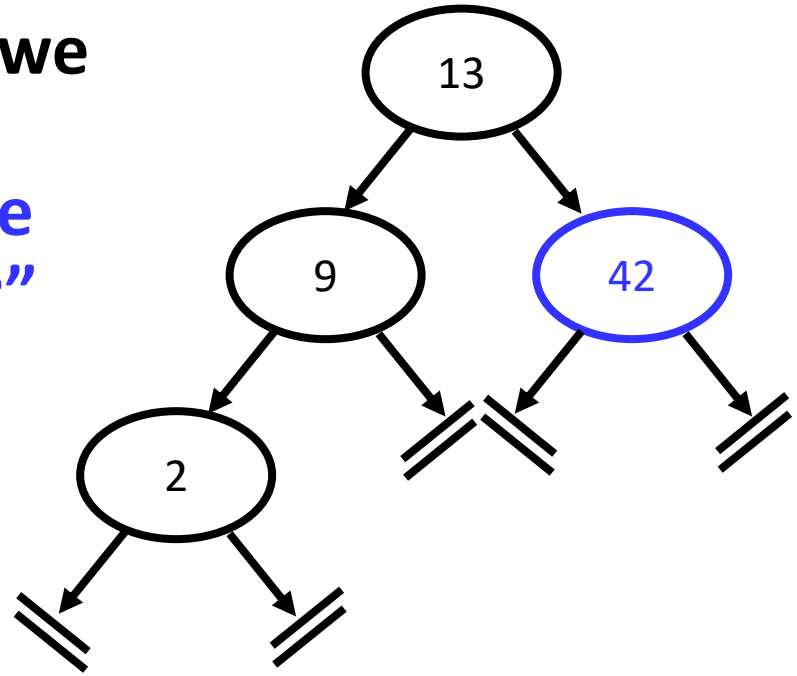


Output: 2 9 13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

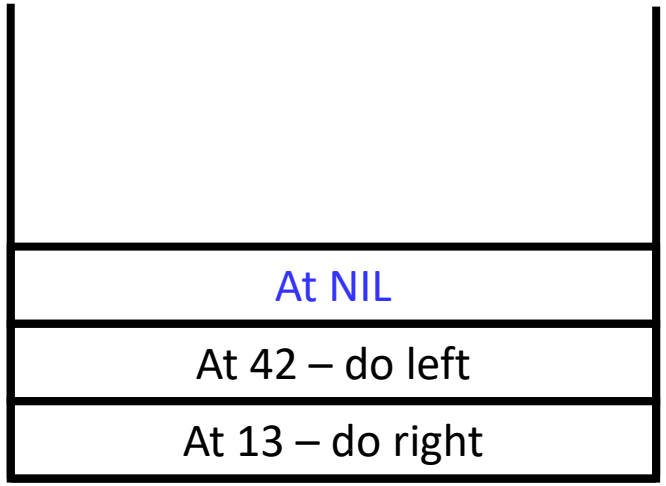


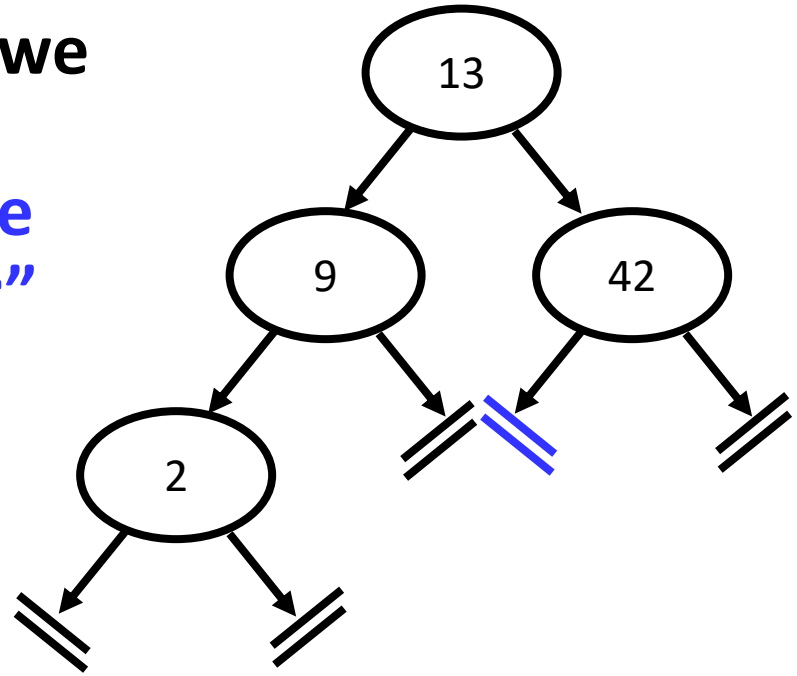


Output: 2 9 13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

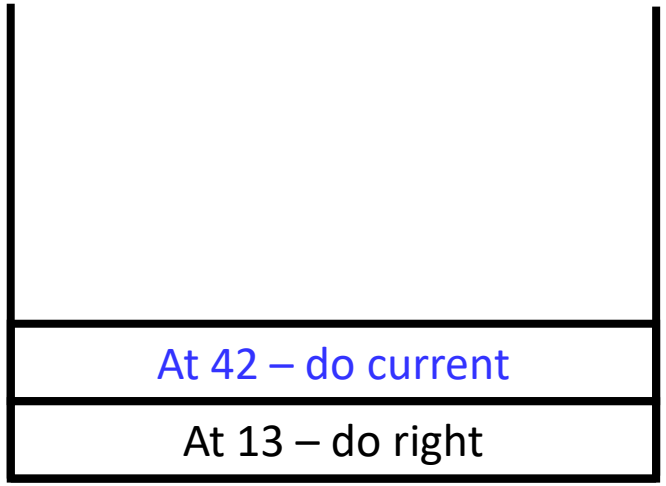


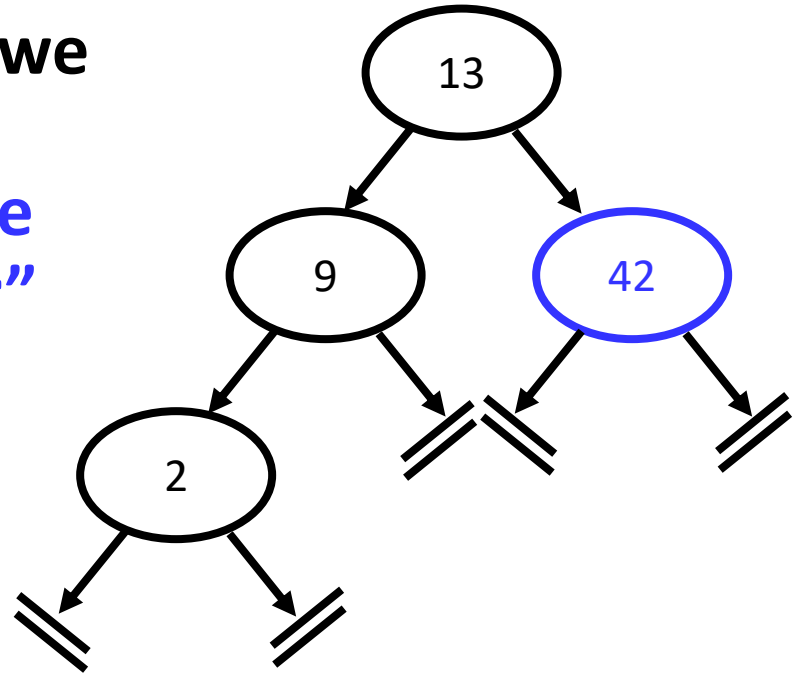


Output: 2 9 13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

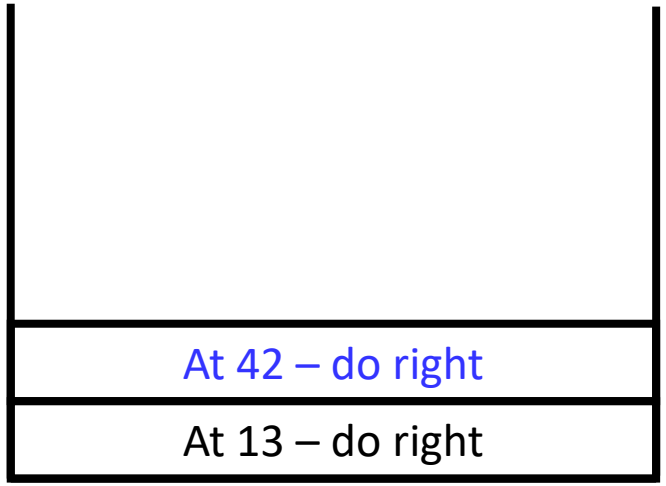


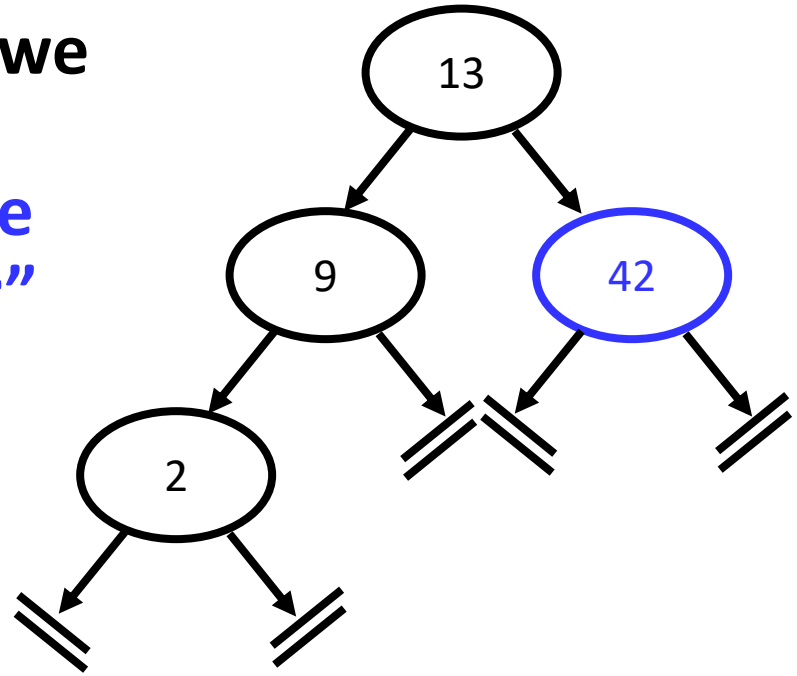


Output: 2 9 13 42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

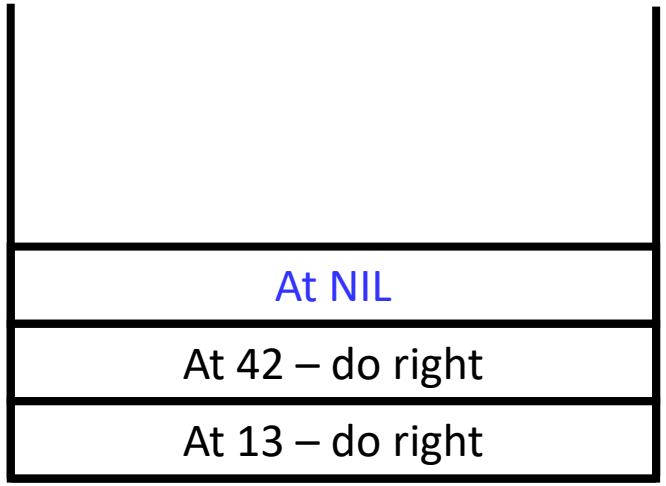


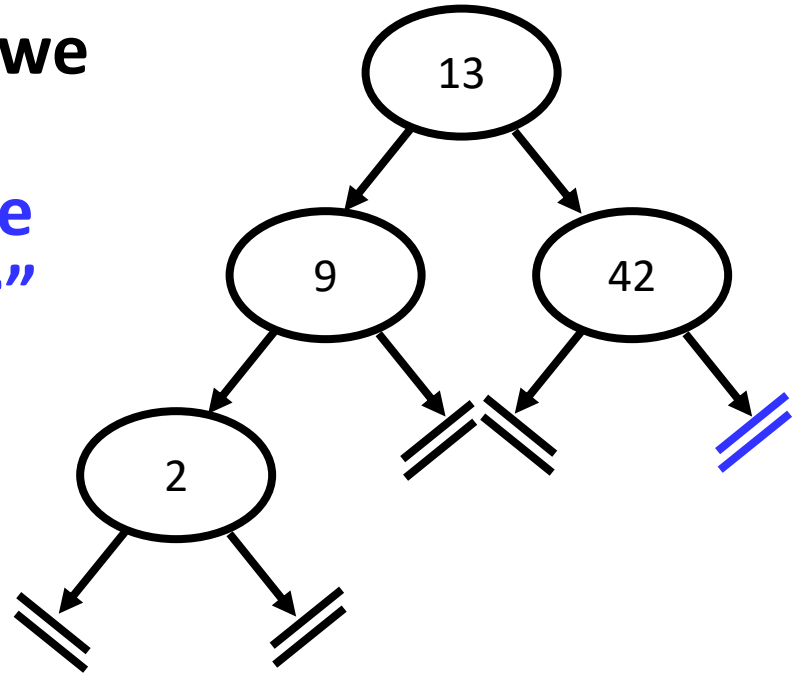


Output: 2 9 13 42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

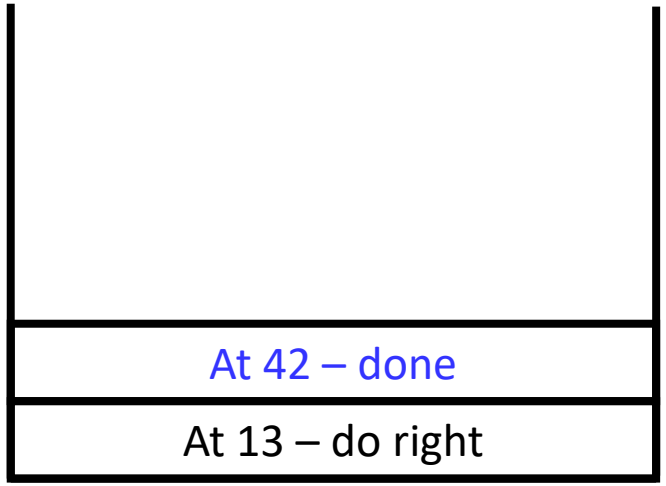


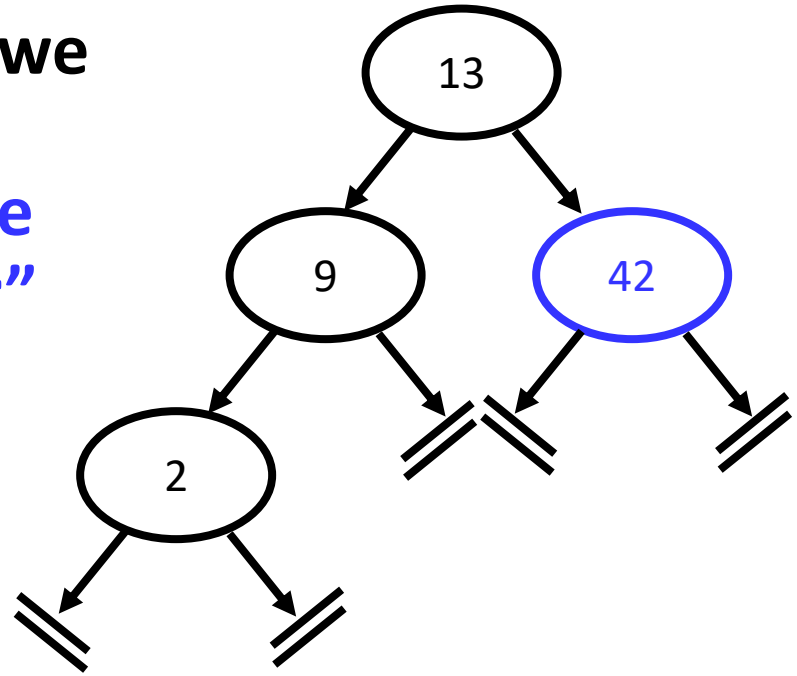


Output: 2 9 13 42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

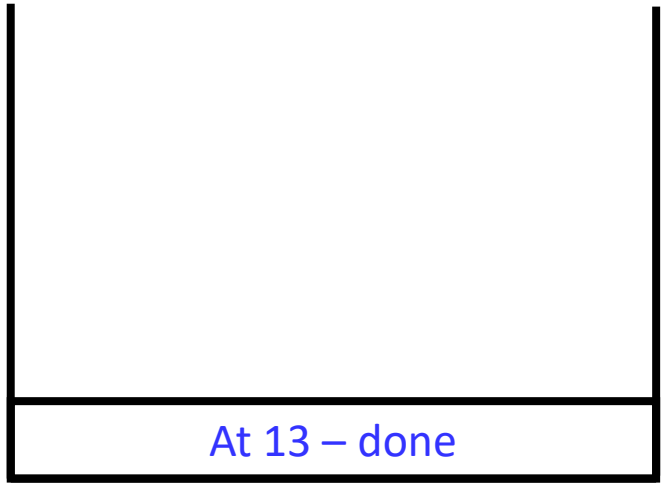


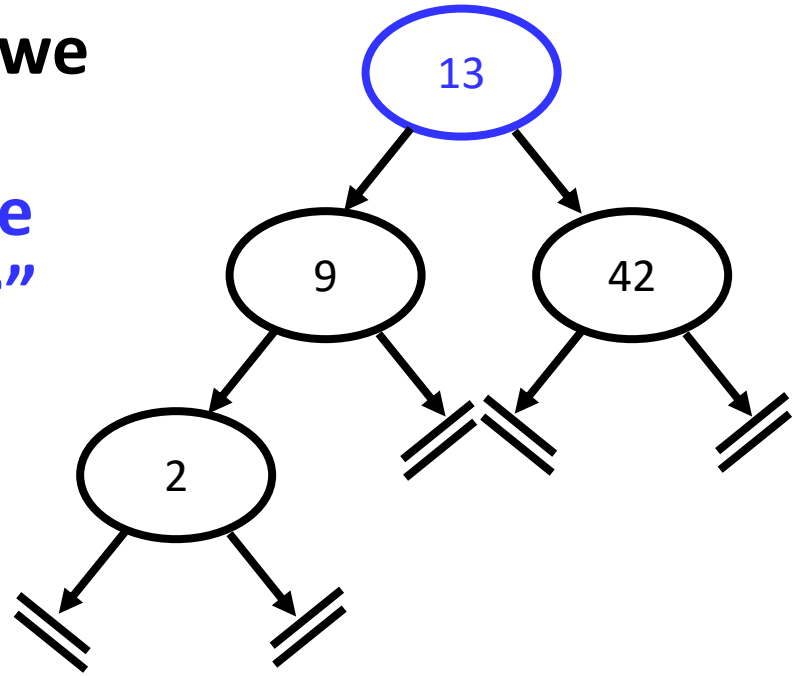


Output: 2 9 13 42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**





Output: 2 9 13 42