Table of Contents

1. Trees: Non – Linear Data Structure 4
2. Definition 4
3. Advantages of Tree 5
4. Basic Terminologies & Properties of Tree Data Structure 5
    1. Root 6
    2. Edge 6
    3. Parent 7
    4. Child 7
    5. Siblings 7
    6. Leaf / External Node 7
    7. Internal Nodes 8
    8. Degree 8
    9. Level 8
    10. Height 9
    11. Depth 9
    12. Path 10
    13. Sub Tree 10
    14. Ancestor of a Node 12
    15. Descendant 12
5. Types of Tree Data Structure 13
    1. General tree 13
    2. Binary tree 13
    3. Balanced Binary tree 14
    4. Binary search tree 14
6. Types of Binary Trees 15
    1. Full Binary Tree (Proper binary tree or 2-tree or strictly binary tree) 15
    a) Maximum No. of Nodes at Each Level 16
    b) Maximum & Minimum No. of Nodes 16
    c) Heigh Calculation 16
    d) Finding Parent and Child Nodes 17
    f) Relation Between Internal Nodes & Leaf Nodes 17
    2. Complete Binary Tree 18

b) Maximum & Minimum No. of Nodes 19
c) Height Calculation 19
d) Finding Parent and Child Nodes 20
e) Finding the Depth of the Last Level 20
f) Finding the Number of Leaf Nodes & Internal Nodes 21
3. Perfect Binary Tree 22
a) Total Number of Nodes: 23
b) Height Calculation: 23
c) Finding Parent and Child Nodes: 23
d) Number of Leaf Nodes: 23
f) Number of Internal Nodes 24
4. Balanced Binary Tree 25
5. A degenerate (or pathological) Tree 26
6. Skewed Binary Tree 27
7. Check whether a binary tree is a full binary tree or not 28
8. Check whether a given binary tree is perfect or not 28
9. Representation of Binary Tree 30
1. Linked Representation 30
2. Array Representation 32
10. Convert General Tree to Binary Tree 35
a) Rules for Conversion 35
11. Binary Tree Traversals 42
12. Depth First Traversal 42
[1]. Pre Order Traversal (Root-Left-Right) 42
[2]. In order Traversal (Left-Root-Right) 42
[3]. Post Order Traversal (Left-Right-Root) 43
13. Breadth First Traversal 44
12. Numerical Based on Finding Traversal from Given Tree 45
a) Rules to Find Traversal. 45
13. Numerical Based on Construction of Tree When Traversals are given 84
14. Expression Tree 95
a) Application of Expression Trees 95
15. Construction of Expression Tree 97

16. Threaded Binary Tree 107
a) Why do we need Threaded Binary Tree? 107
b) Types of threaded binary trees 108
c) Find In-Order Successor and In-Order Predecessor 109

1. Trees: Non – Linear Data Structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order.

A data structure is said to be nonlinear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure below shows a tree and a non-tree.

2. Definition

- Tree is a collection of **Nodes (or) vertices and edges (or) links** where each node is connected with other node in such a way that there is no formation of a cycle/circuit/self-loop.
- In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.
- A tree with **N** node must have N-1 edges and all the n nodes must be connected with each other.

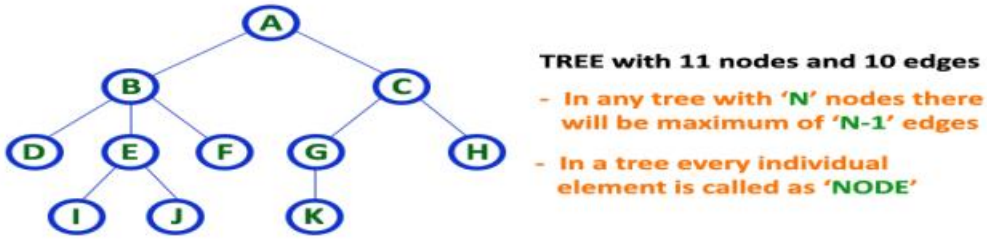
3. Advantages of Tree

- Trees are so useful and frequently used, because they have some very serious advantages:
  - Trees reflect structural relationships in the data
  - Trees are used to represent hierarchies
  - Trees provide an efficient insertion and searching
  - Trees are very flexible data, allowing to move sub trees around with minimum effort

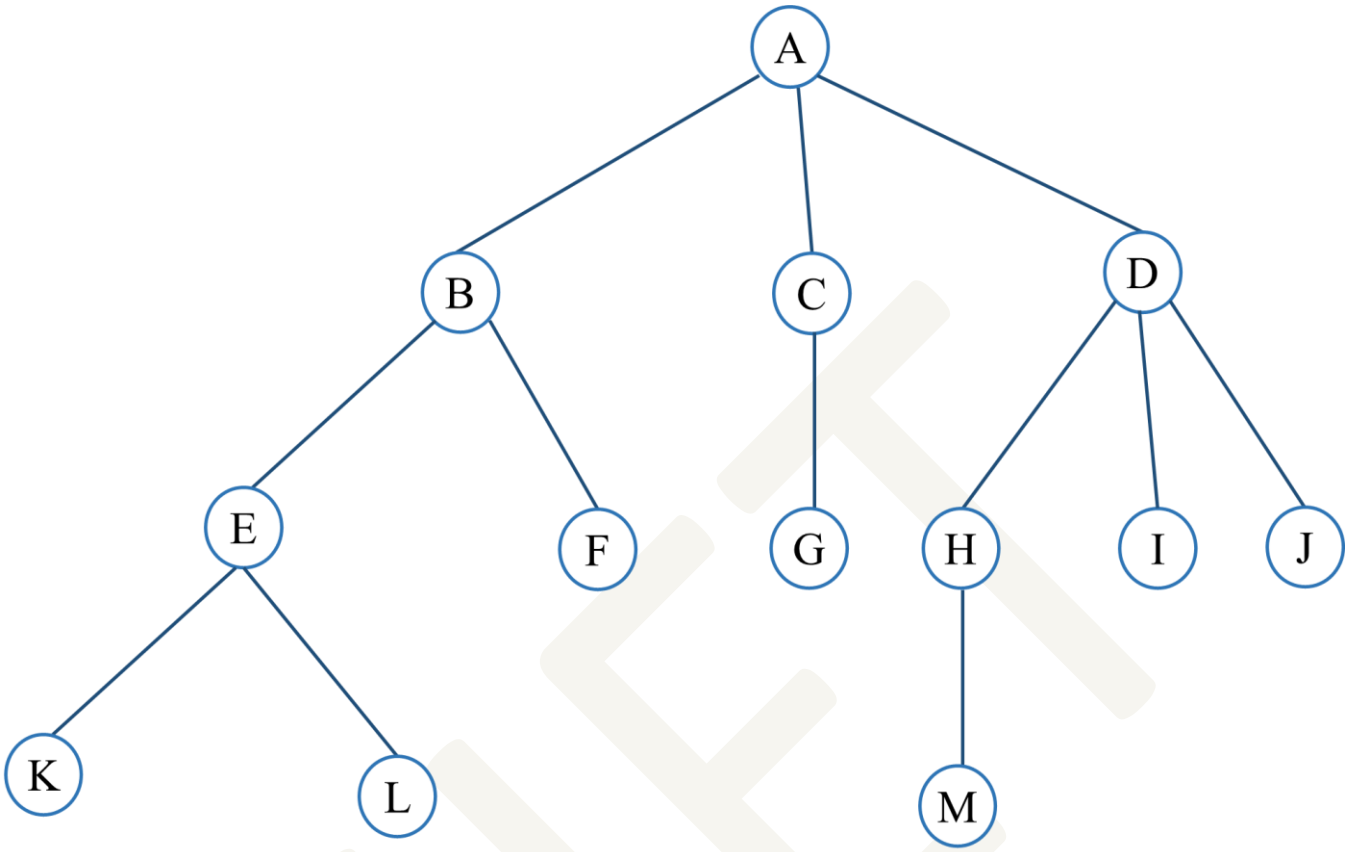
4. Basic Terminologies & Properties of Tree Data Structure

In a Tree, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

Example,



To Understand the Basic terminologies let us consider a Tree for example as below:



1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure.

In any tree, there must be only one root node. We never have multiple root nodes in a tree. In above tree, A is a Root node.

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. **In simple words, the node which has branch from it to any other node is called as parent node.**

Parent node can also be defined as "The node which has child / children". e.g., Parent (A, B, C, D, E, H).

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node.

In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

e.g., Children of “D” are (H, I, J).

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS.

In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B,C, D)

6. Leaf / External Node

In a tree data structure, the node which does not have a child (or) node with degree zero is called as LEAF Node.

In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes.

External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K, L, F, G, M, I, J)

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes.

The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'non-Terminal' nodes. Ex: B, C, D, E, H

8. Degree

In a tree data structure, the total number of children of a node (or)number of subtrees of a node is called as DEGREE of that Node.

In simple words, the Degree of a node is total number of children it has.

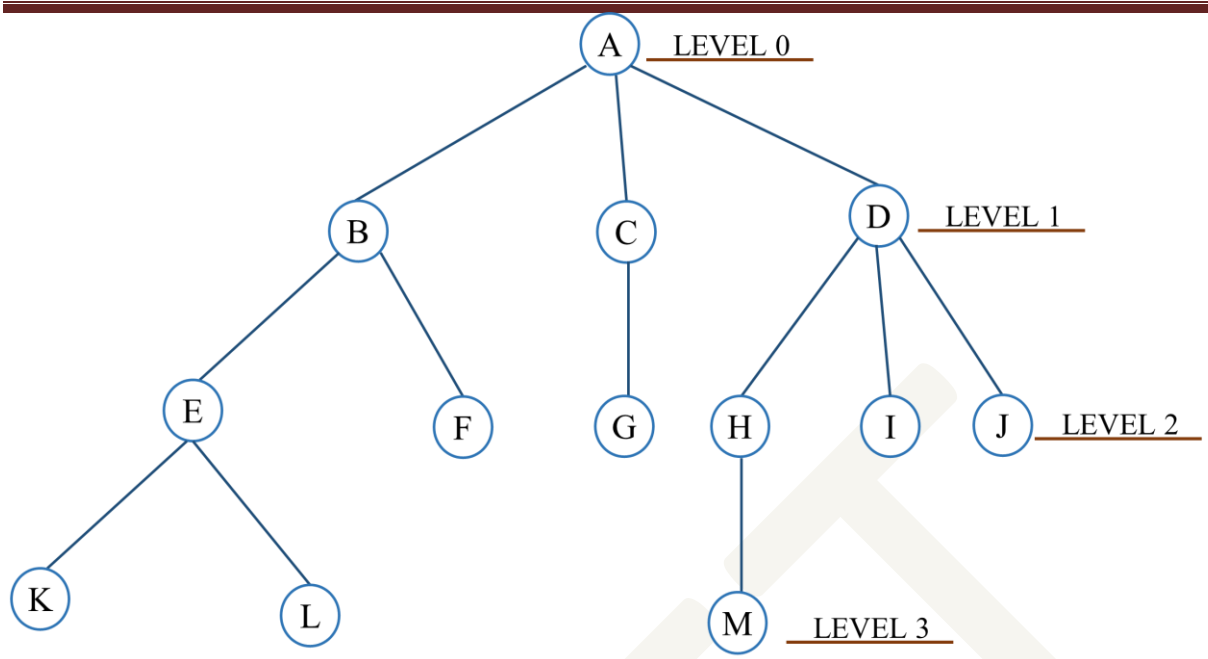
The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.

9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...

In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).

Here in our figure Levels can be shown as below:



Node A at Level 0

Node B, C & D at Level 1

Node E, F, G, H, I & J at Level 2

Node K, L, & M at Level 3

10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node.

In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node.

In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree.

In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

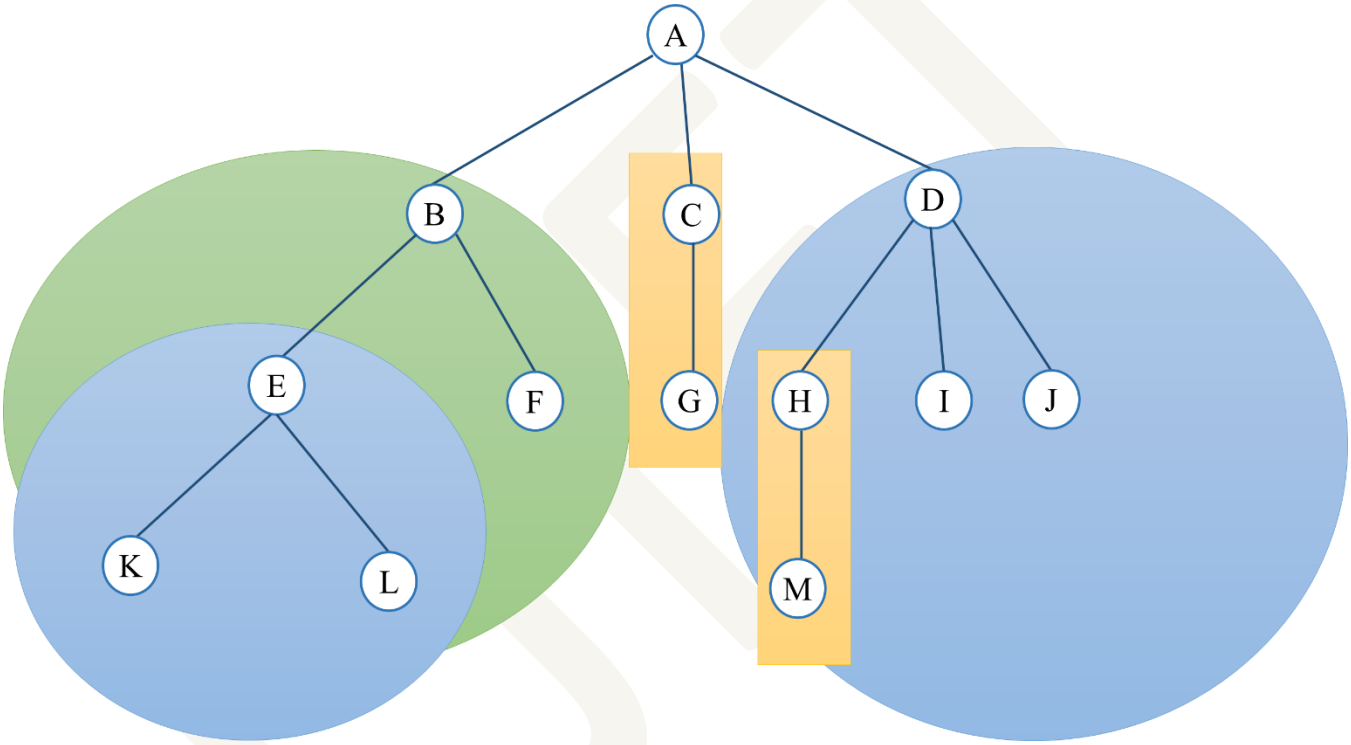
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes.

Length of a Path is total number of nodes in that path. In below example the path A - B - E - L has length 4.

13. Sub Tree

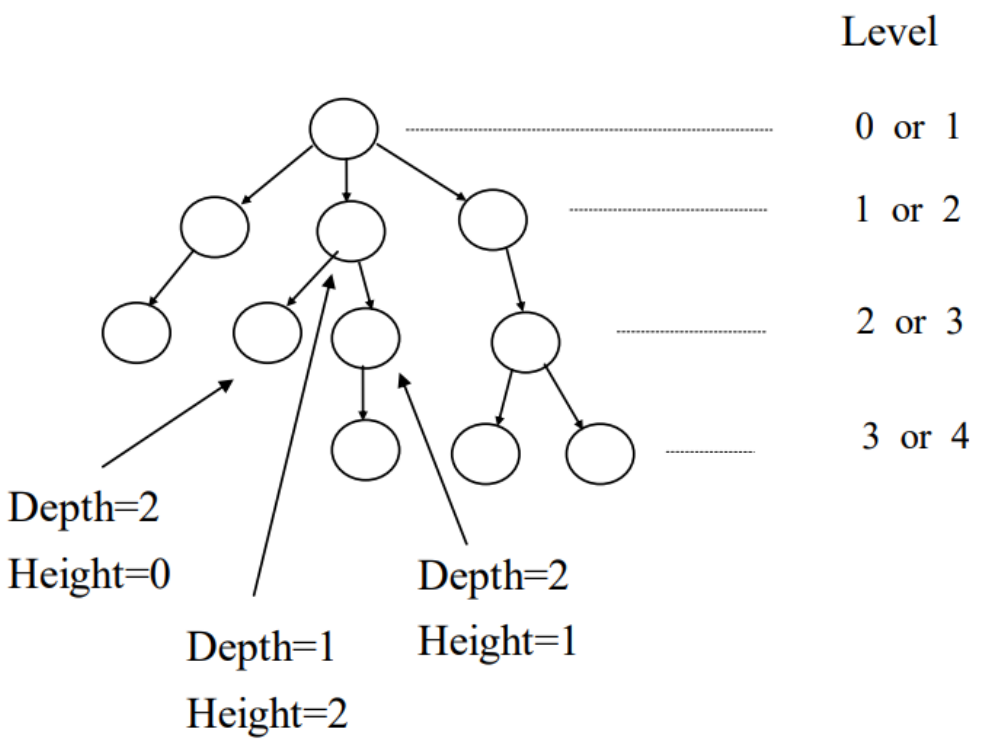
In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Every Highlighted part is a sub-tree.

Important Points about Height & Depth.

- The depth of a node is the number of edges from the root to the node.
- The root node has depth zero
- The height of a node is the number of edges from the node to
- the deepest leaf.
- The height of a tree is a height of the root.
- The height of the root is the height of the tree
- Leaf nodes have height zero
- A tree with only a single node (hence both a root and leaf) has
- depth and height zero.
- An empty tree (tree with no nodes) has depth and height −1.
- It is the maximum level of any node in the tree.



- Please note that if you label the level starting from 1, the depth (height) is level-1 (max level -1)

14. Ancestor of a Node

Any predecessor nodes on the path of the root to that node are called Ancestors of that node.

15. Descendant

Any successor node on the path from the leaf node to that node.

5. Types of Tree Data Structure

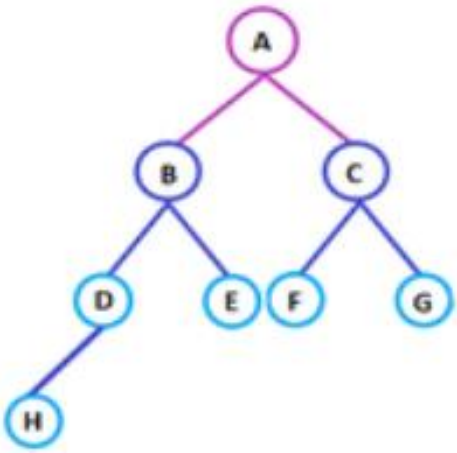
The different types of tree data structures are as follows:

1. General tree

A general tree data structure has no restriction on the number of nodes. It means that a parent node can have any number of child nodes.

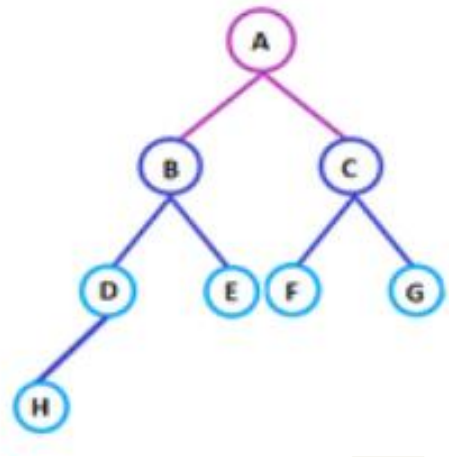
2. Binary tree

A node of a binary tree can have a maximum of two child nodes. In the given tree diagram, node B, D, F and H are left children, while E, C, and G are the right children.

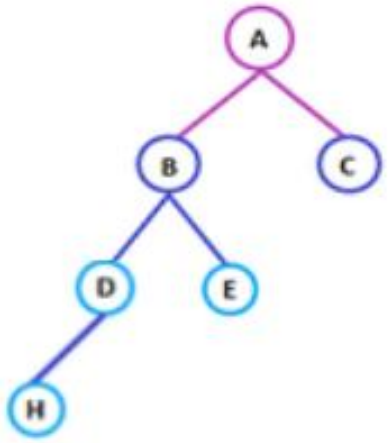


3. Balanced Binary tree

If the height of the left sub-tree and the right sub-tree is equal or differs at most by 1, the tree is known as a balanced tree.



Balanced Tree



Unbalanced Tree

4. Binary search tree

Binary search trees shows that the value of the left node is less than its parent, while the value of the right node is greater than its parent. They are used for various searching and sorting algorithms.

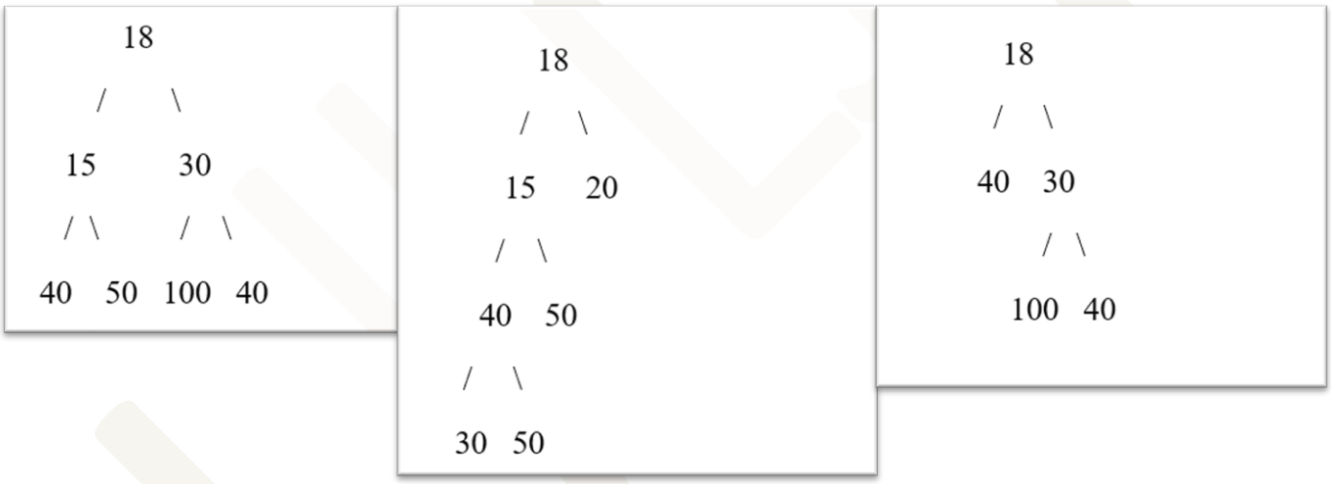
- The examples include AVL tree and red-black tree.
- It is a non-linear data structure.

6. Types of Binary Trees

1. Full Binary Tree (Proper binary tree or 2-tree or strictly binary tree)

- **A Binary Tree is a Full Binary tree if every node has 0 or 2 children.**
- We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.
- A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.

The following are the examples of a full binary tree.



a) Maximum No. of Nodes at Each Level

In a strict binary tree, the number of nodes at each level follows a pattern.

If we consider the root as level 0, then the number of nodes at level i is given by 2^i.

For example, at level 0 (root), there is one node ($2^0 = 1$), at level 1, there are two nodes ($2^1 = 2$), at level 2, there are four nodes ($2^2 = 4$), and so on.

b) Maximum & Minimum No. of Nodes

The Maximum number of nodes in a strict binary tree of height h is given by the formula: $2^{(h+1)} - 1$.

This formula is derived by summing the number of nodes at all levels from 0 to h.

For example, a strict binary tree of height 2 will have Maximum of $2^{(2+1)} - 1 = 7$ nodes.

The Minimum number of nodes in a strict binary tree of height h is given by the formula: $2h+1$.

For example, a strict binary tree of height 2 will have Maximum of $2(2) +1 = 5$ nodes.

c) Heigh Calculation

The height (or depth) of a strict binary tree with **n** nodes **(considering n is maximum no. of nodes)** can be calculated using the formula: $\lfloor \log_2 (n+1) \rfloor - 1$.

The symbol indicates floor down, which means you have to round down the answer if value of $\log_2 (n+1)$ is in decimal. If answer is already in integer then no need to round down.

For general strict binary trees, the height can be larger, and the formula only gives the minimum possible height.

d) Finding Parent and Child Nodes

For any node with index **i** (0-based indexing) in a strict binary tree,

- its left child's index is **2i + 1.**
- its right child's index is **2i + 2.**

Conversely, the parent of a node with index i has an index of **(i - 1) / 2 (where i>0)**. These formulas are useful when representing the tree in an array-based implementation.

f) Relation Between Internal Nodes & Leaf Nodes

In any Full Binary Tree, If **Leaf Nodes are denoted by L & Internal Nodes are denoted by I** then the relation between Internal Nodes and Leaf Nodes can be given by – **L = I +1**

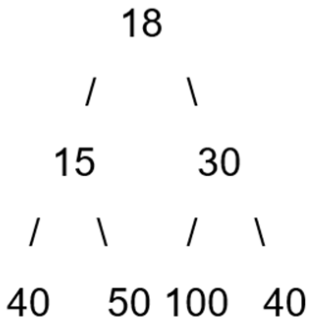
2. Complete Binary Tree

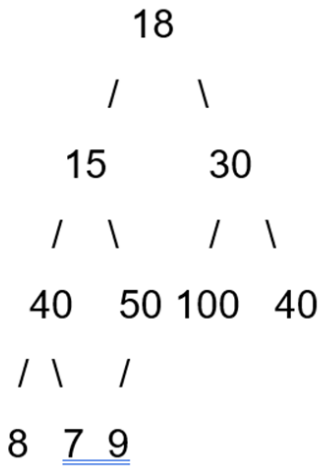
A Binary Tree is a **Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible (Left filled keys).**

A complete binary tree is just like a full binary tree, but with two major differences:

- Every level must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree. But vice versa is not true.

The following are examples of Complete Binary Trees.





Practical example of Complete Binary Tree is Binary Heap.

a) Maximum No. of Nodes at Each Level

In a complete binary tree, the number of nodes at each level follows a pattern.

If we consider the root as level 0, then the number of nodes at level i is given by **2^i.**

For example, at level 0 (root), there is one node ($2^0 = 1$), at level 1, there are two nodes ($2^1 = 2$), at level 2, there are four nodes ($2^2 = 4$), and so on.

b) Maximum & Minimum No. of Nodes

The Maximum number of nodes in a complete binary tree of height h is given by the formula: **$2^{(h+1)} - 1$**.

This formula is derived by summing the number of nodes at all levels from 0 to h.

For example, a complete binary tree of height 2 will have Maximum of $2^{(2+1)} - 1 = 7$ nodes.

The Minimum number of nodes in a complete binary tree of height h is given by the formula: **2^h**.

For example, a complete binary tree of height 2 will have Maximum of $2^2 = 4$ nodes.

c) Height Calculation

The height (or depth) of a complete binary tree with **n nodes** can be approximated using the formula: **$\lfloor \log_2(n) \rfloor$**

This formula gives an estimation of the height based on the total number of nodes in the tree.

For example, the total number of Nodes are given 5 then height can be calculated by = **$\lfloor \log_2(n) \rfloor = \lfloor \log_2(5) \rfloor = \left\lfloor \frac{log\ 5}{log\ 2} \right\rfloor = \lfloor 2.32 \rfloor = 2$**

Since complete binary trees are usually balanced, the height is close to $\log_2(n)$, but it may differ slightly if the last level is incomplete.

But, in case of No. Nodes <= 3 we need to Round Up the answer.

OR

THERE IS ANOTHER UNIVERSAL FORMULA

$$h=\lceil \log_2(n+1) \rceil -1$$

d) Finding Parent and Child Nodes

For any node with index **i** (0-based indexing) in a complete binary tree,

- its left child's index is **2i + 1.**
- its right child's index is **2i + 2.**

Conversely, the parent of a node with index i has an index of **(i - 1) / 2**. These formulas are useful when representing the tree in an array-based implementation.

e) Finding the Depth of the Last Level

Method 1:

In a complete binary tree, the depth of the last level (deepest level) can be found by counting the number of digits in the binary representation of the total number of nodes (including the root).

For example, if the total number of nodes is 13 (1101 in binary), the depth of the last level is 3.

To find the depth of the last level, we count the number of digits in the binary representation. In this case, the binary representation has 4 digits. However, we are interested in the depth of the last level, which is the number of digits minus 1.

Depth of the last level = Number of digits - 1 = 4 - 1 = 3.

Method 2:

Prepare a table with following columns: (We are considering the same example where No. of Nodes are given 13)

Level No.	No. of Nodes at that level(2^i)	Total Cumulative Nodes in Tree
0	1	1
1	2	3
2	4	7
3	8	15

This means the At Level 3 the total No. of nodes will be 15. Hence, the Height of the Tree having 13 Nodes will have the depth at last level = Level No. (which is 3 in our case)

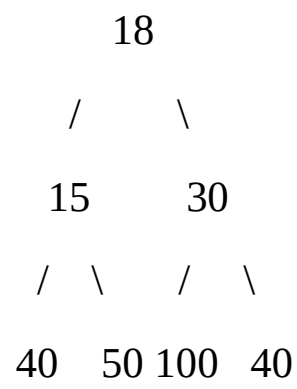
f) Finding the Number of Leaf Nodes & Internal Nodes

In a complete binary tree, When Total No. of Nodes are given, the Number of Leaf & Internal (Non-Leaf Nodes) are given by

Value of N	No. of Leaf Nodes	No. of Internal Nodes
n (Odd Value)	(n+1)/2	(n-1)/2
n (Even Value)	n/2	n/2

3. Perfect Binary Tree

- A Perfect Binary Tree is a Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.
- The following are the examples of Perfect Binary Trees.



a) Total Number of Nodes:

The number of nodes in a perfect binary tree of height h is given by the formula: **$2^{(h+1)} - 1$.** This formula counts all the nodes in the tree, including both internal nodes and leaf nodes.

The above formula holds true for Max value and Min value both.

b) Height Calculation:

The height (or depth) of a perfect binary tree with **n** nodes (considering n is maximum no. of Nodes) can be calculated using the formula: **$\log_2(n+1) - 1$.**

This formula gives the height of the tree based on the total number of nodes in it. For example, a perfect binary tree **with maximum 15 nodes** will have a height of **$\log_2(15+1) - 1 = \log_2(16) - 1 = \log_2(2^4) - 1 = 4\log_2(2) - 1 = 4(1) - 1 = 4-1 = 3$**.

c) Finding Parent and Child Nodes:

For any node with index **i** (0-based indexing) in a strict binary tree,

- its left child's index is **$2i + 1$.**
- its right child's index is **$2i + 2$.**

Conversely, the parent of a node with index i has an index of **$(i - 1) / 2$**. These formulas are useful when representing the tree in an array-based implementation.

d) Number of Leaf Nodes:

In a perfect binary tree, the number of leaf nodes (nodes with no children) is given by **(2^h)**, where h is the height of the tree.

This formula can be derived from the fact that all levels of the tree are filled, and the last level contains only leaf nodes.

For example, if a perfect binary tree is having height 4 (h=4), then total number of leaf nodes would be = $2^h = 2^4 = 16$. Hence the tree would have 16 no of leaf nodes.

e) Relation Between Internal Nodes & Leaf Nodes:

In any Full Binary Tree, If **Leaf Nodes are denoted by L & Internal Nodes are denoted by I** then the relation between Internal Nodes and Leaf Nodes can be given by – **L = I +1.**

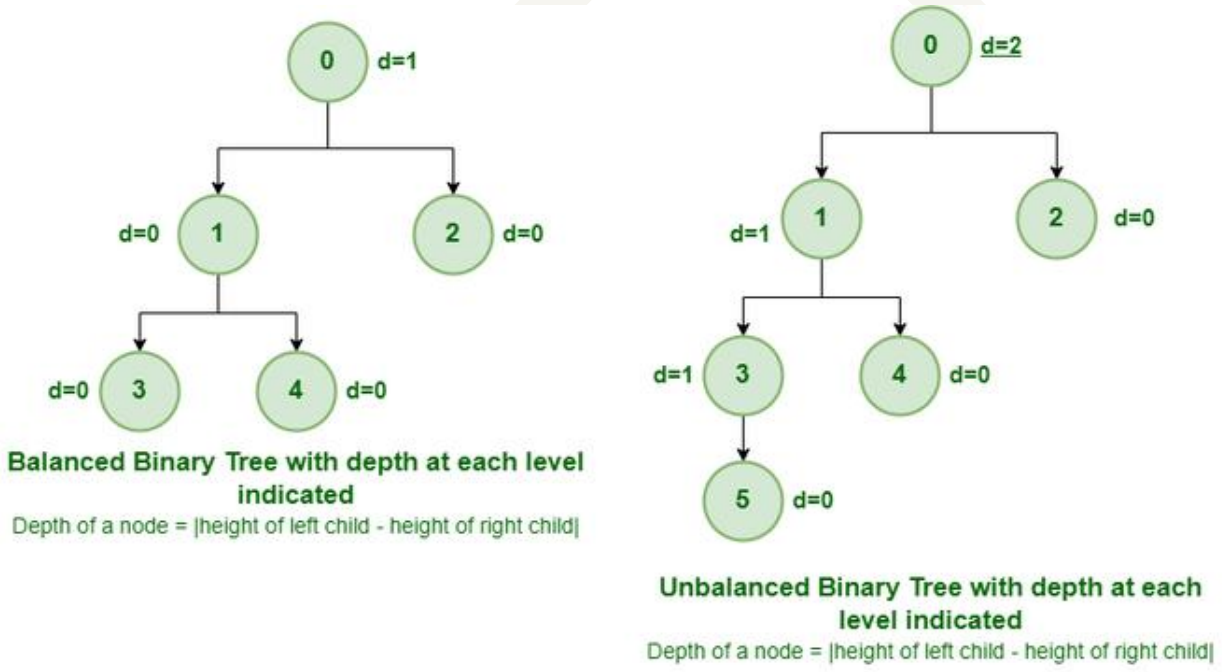
f) Number of Internal Nodes

The number of internal nodes (non-leaf nodes) in a perfect binary tree of height h is given by **2^h - 1.**

For example, if a perfect binary tree is having height 3 (h=3), then total number of leaf nodes would be = 2^h -1= 2^3-1 = 8-1=7. Hence the tree would have 7 no of Internal Nodes.

4. Balanced Binary Tree

- A binary tree is balanced if the maximum height of the tree is Log n where n is the number of nodes.
- For Example, the AVL tree maintains maximum Log n height by making sure that the difference between the heights of the left and right subtrees is at most 1.
- Red-Black trees maintain maximum Log n height by making sure that the number of Black nodes on every root to leaf paths is the same and there are no adjacent red nodes.
- Balanced Binary Search trees are performance-wise good as they provide maximum log n time for search, insert and delete.



Example of Balanced and Unbalanced Binary Tree

- It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1. In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

Tree Height and Number of Nodes Relation:

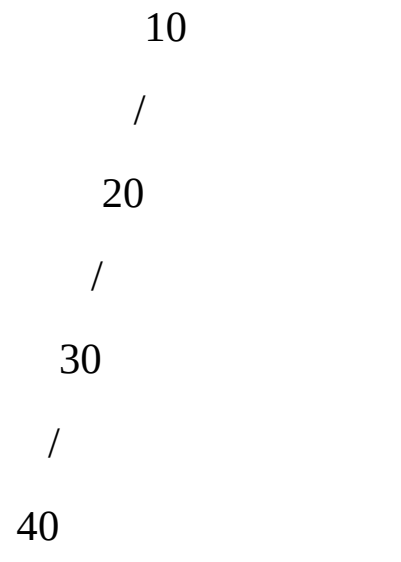
For a balanced binary tree with n nodes, the height of the tree is approximately log2(n). The logarithmic height ensures efficient operations, and the tree remains relatively balanced even with a large number of nodes.

5. A degenerate (or pathological) Tree

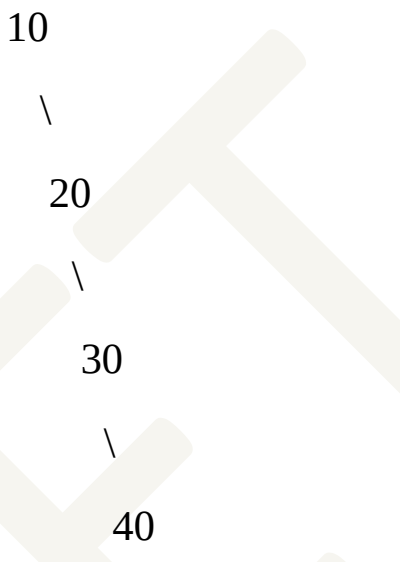
- A Tree where every internal node has one child. Such trees are performance-wise same as linked list.
- A degenerate or pathological tree is the tree having a single child either left or right.

6. Skewed Binary Tree

- A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes.
- Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



Left-Skewed Binary Tree



Right-Skewed Binary Tree

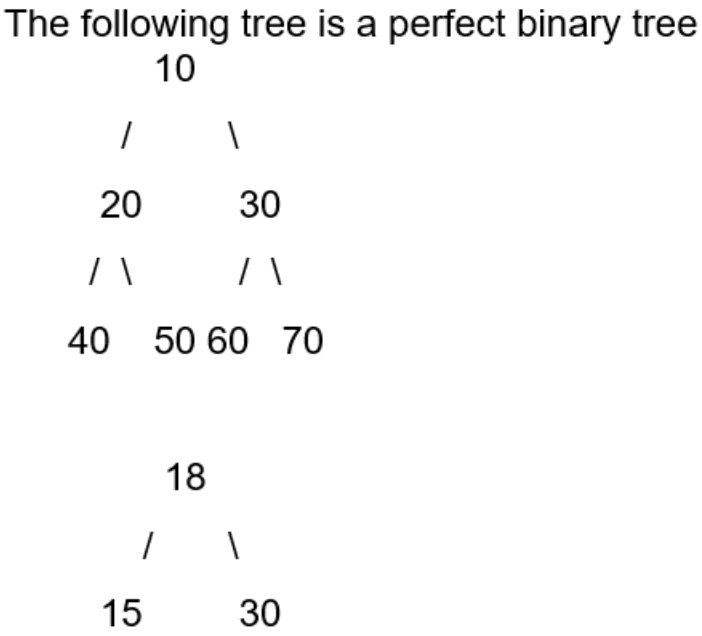
7. Check whether a binary tree is a full binary tree or not

- A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has one child node.
- To check whether a binary tree is a full binary tree we need to test the following cases: -

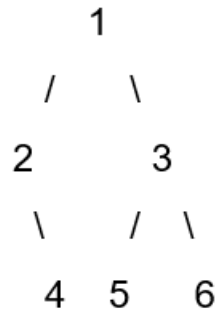
1. If a binary tree node is NULL then it is a full binary tree.
2. If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition.
3. If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.
4. In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

8. Check whether a given binary tree is perfect or not

- A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.



The following tree is **not** a perfect binary tree



- A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has **$2^{h+1} - 1$ node.** (Considering the height of leaf node as 0)
- Below is an idea to check whether a given Binary Tree is perfect or not.
  1. Find depth of any node (in below tree we find depth of leftmost node). Let this depth be "d".
  2. Now recursively traverse the tree and check for following two conditions.
     - Every internal node should have both children non-empty
     - All leaves are at depth 'd'

9. Representation of Binary Tree

Tree can be represented in 2 ways:

1. Dynamic node Representation (Linked Representation)
2. Array Representation

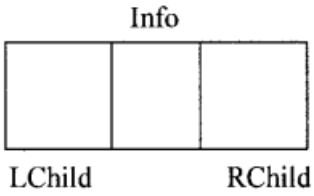
1. Linked Representation

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as:

A Tree node contains the following parts.

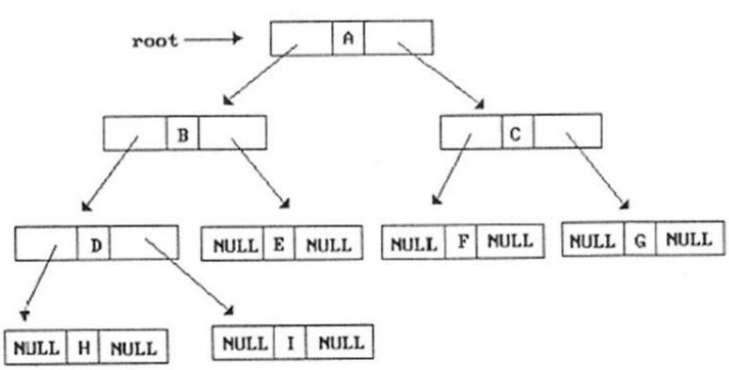
- Left Child (LChild)
- Data or Information of the Node (Info)
- Right Child (RChild)

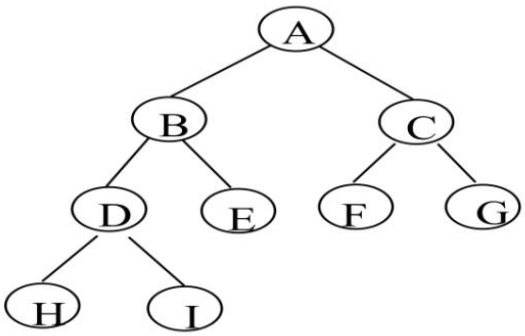
The LChild links to the Left Child Node of the Parent Node, Info holds the information of every node and RChild holds the address of right child of the parent node.



In Java, we can represent a tree node using Inner class. Below is an example of a tree node with integer data.

```
//Inner class node

class node

{

        int data;

        node left;

        node right;

}
```



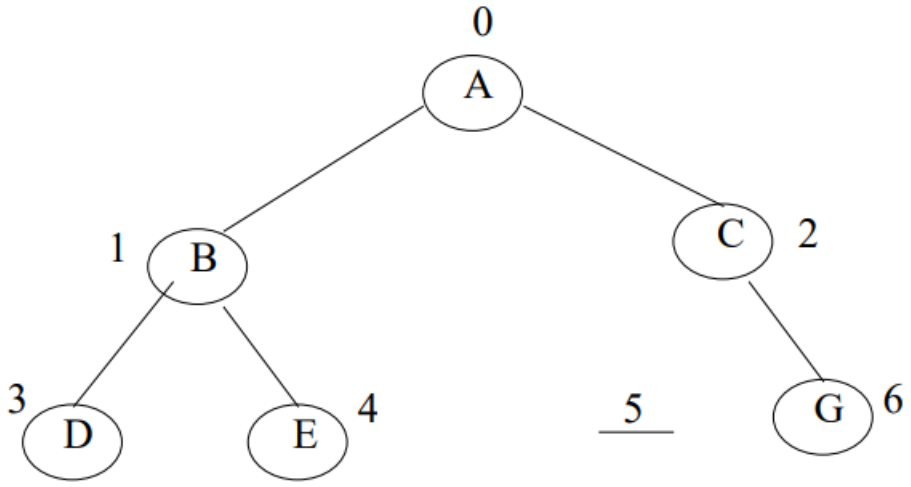


2. Array Representation

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially.

Suppose a binary tree T of depth d. Then at most **$2^d - 1$** node can be there in T. **Hence the Size of Array would be $2^d - 1$.**

Consider a binary tree in below figure of depth 3. Then SIZE = $2^3 - 1 = 7$. Then the array A [7] is declared to hold the nodes.



A[ ]	A	B	C	D	E		F
	[0]	[1]	[2]	[3]	[4]	[5]	[6]

Array representation of the binary tree

The array representation of the binary tree is shown above. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.

- The father of a node having index n can be obtained by (n – 1)/2. For example, to find the father of D, where array index n = 3. Then the father nodes index can be obtained

  = (n – 1)/2

  = 3 – 1/2

  = 2/2

  = 1

  i.e., A [1] is the father D, which is B.

- The left child of a node having index n can be obtained by (2n+1). For example, to find the left child of C, where array index n = 2. Then it can be obtained by

  = (2n +1)

  = 2*2 + 1

  = 4 + 1

  = 5

  i.e., A [5] is the left child of C, which is NULL. So, no left child for C.

- The right child of a node having array index n can be obtained by the formula (2n + 2). For example, to find the right child of B, where the array index n = 1. Then

  = (2n + 2)

  = 2*1 + 2

  = 4

  i.e., A [4] is the right child of B, which is E.

- If the left child is at array index n, then its right brother is at (n + 1). Similarly, if the right child is at index n, then its left brother is at (n – 1).

The array representation is more ideal for the complete binary tree. The Figure we have taken is not a complete binary tree. Since there is no left child for node C, i.e., A [5] is vacant. Even though memory is allocated for A [5] it is not used, so wasted unnecessarily.

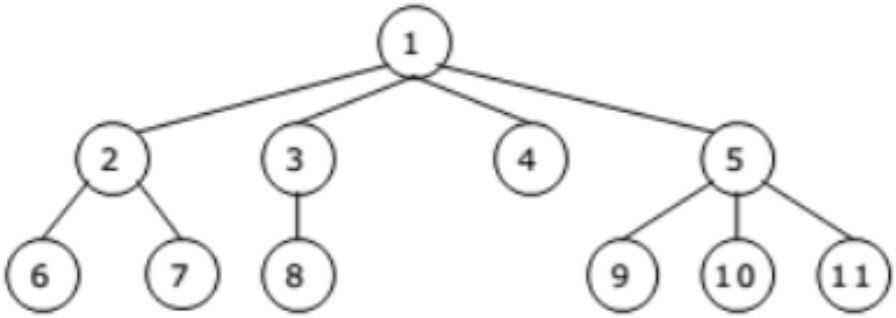
10. Convert General Tree to Binary Tree

Now, in this section we will learn how to convert a General or Generic Tree in to Binary Tree.

a) Rules for Conversion

1. Root of Generic Tree = Root of Binary Tree.
2. Left Child of a Particular Node = Left Child of that particular Node in Binary Tree.
3. Right Sibling of a Particular Node = Right Child of that particular Node in Binary Tree.

Q.1(Q.B 115) Construct a Binary tree from given General tree.

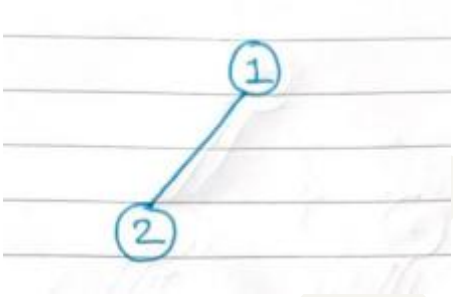


Solution:

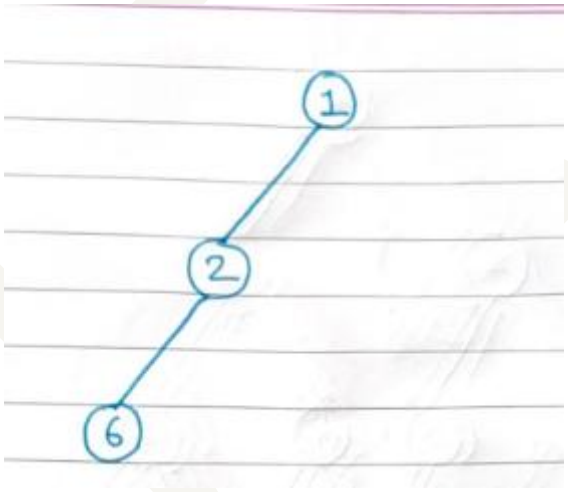
For the very first numerical we will draw the binary tree step-by-step.

Step 1 – Root of Generic Tree = Root of Binary Tree.

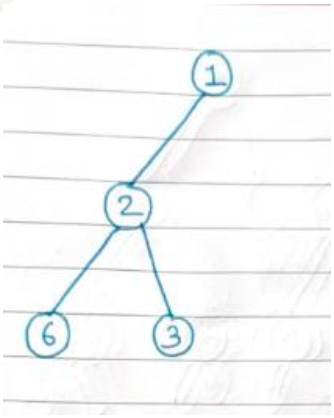
Step 2 – Left child of a particular node in Generic Tree = Left Child of a Particular Node in Binary Tree (Node 1 has left child as Node 2) Hence it will go as left child of 1.



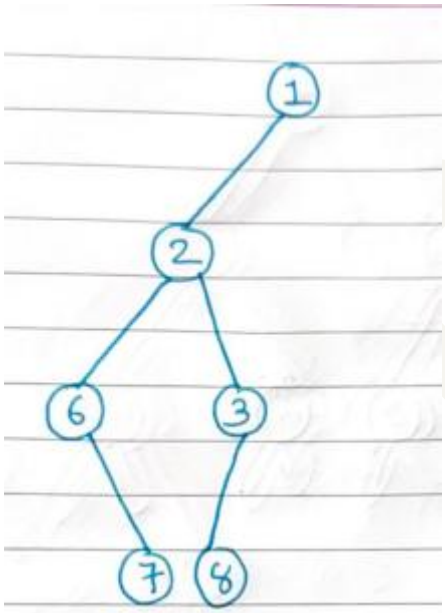
Step 3 – Now, if you check Node 1 has not any right sibling. Hence, in Binary Tree Node 1 will note have right child. So, we are moving ahead with Node 2. Now, Node 2 has left child as Node 6. Hence it will be in the left on Node 2.



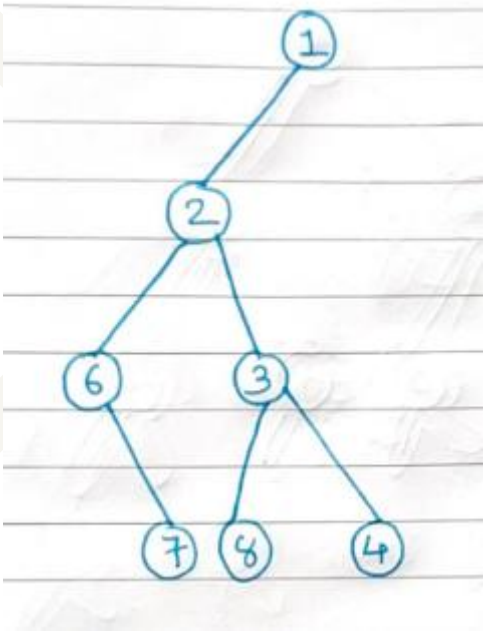
Step 4 – Now, If you see the generic Tree, Node 2 has the right sibling as Node 3. Hence, Node 3 will be the right child in Binary Tree.



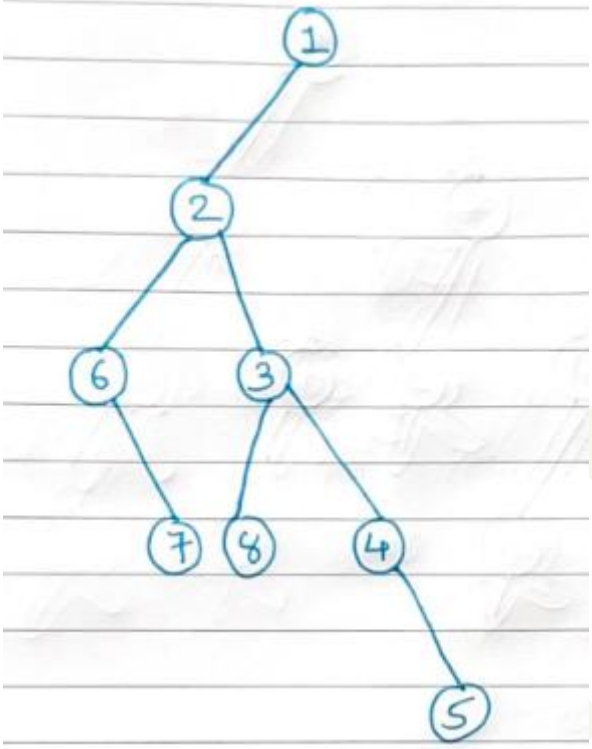
Step 5 – Now, Node 6 doesn't have any left child but it has a right sibling as Node 7. Also, Node 3 has only 1 child as Node 8. Hence, Node 8 will the left child.



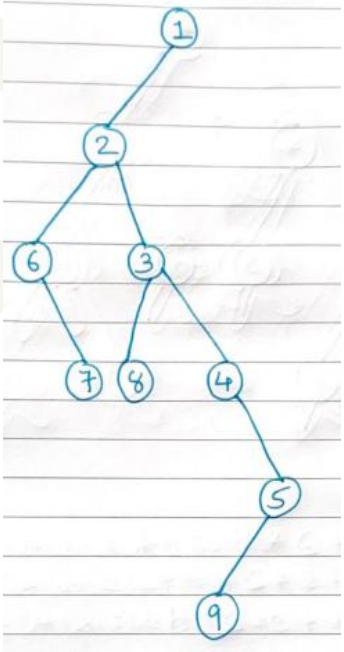
Step 6 – Now, Node 3 has the right sibling as Node 4. Hence, it will be the right child of Node 3 in binary tree.



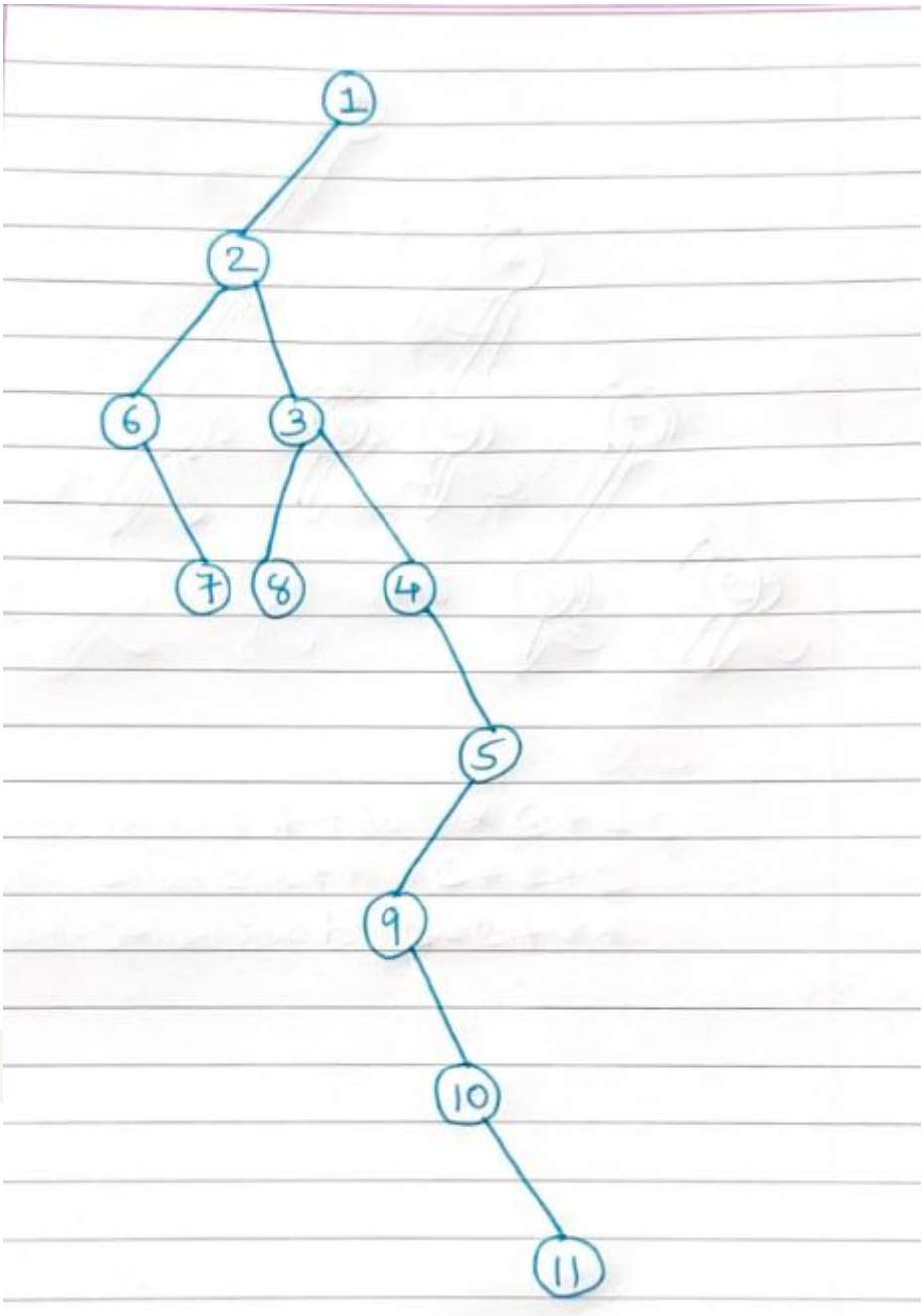
Step 7 – Now, Node 4 doesn't have any left child but it has right sibling as Node 5.



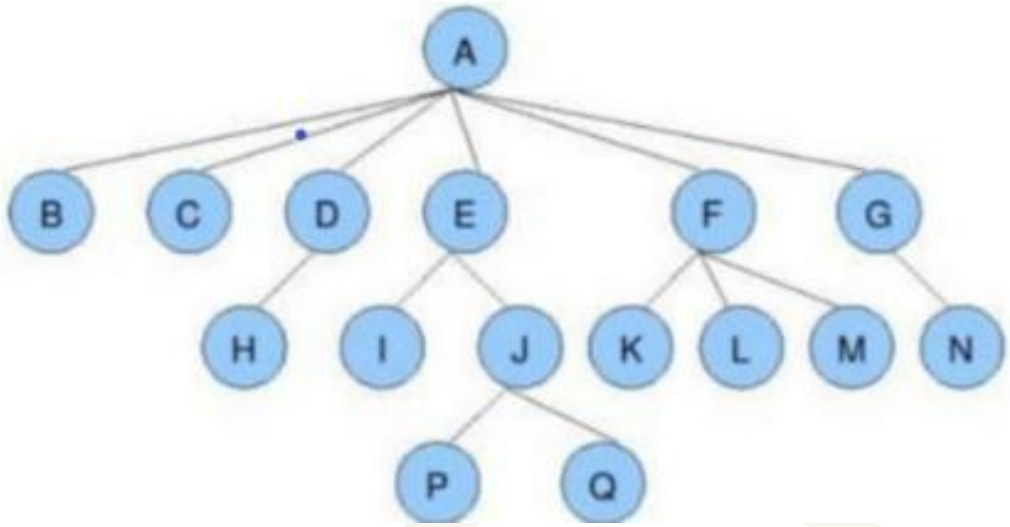
Step 8 – Now, Node 5 has the left child as Node 9.



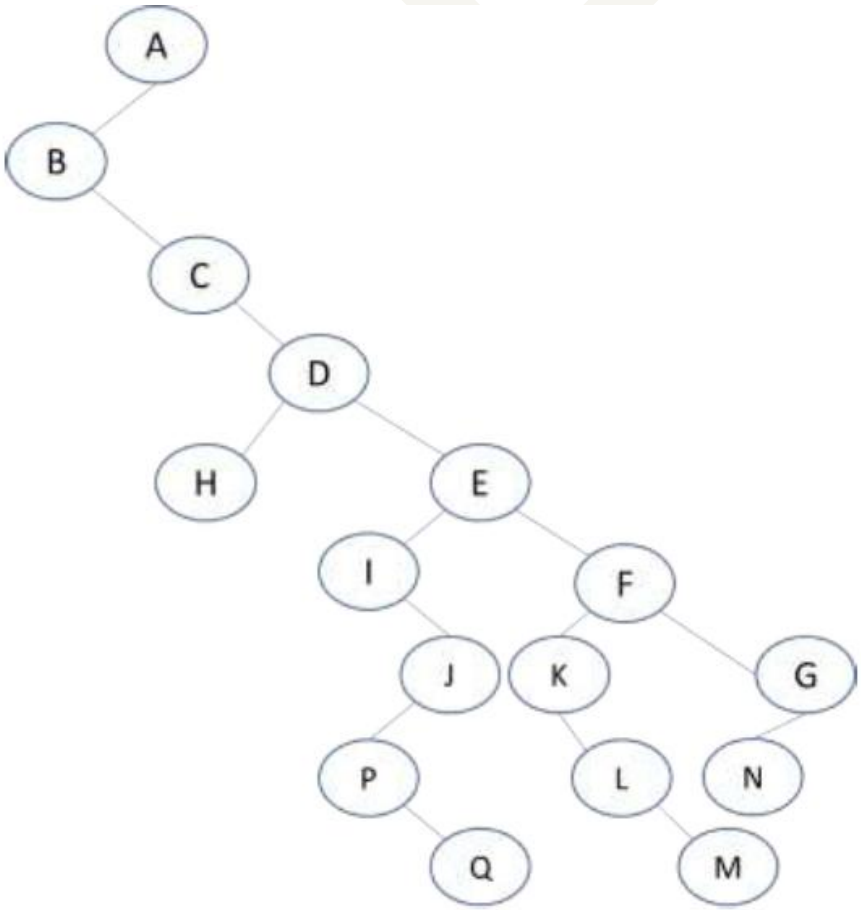
Step 9 – Now, Node 9 doesn't have any left child but it has right sibling has Node 10. Also, Node 10 doesn't have any left child but it has right sibling has Node 11.



Q.2(Q.B 117) Construct a Binary tree from given General tree.



Solution:



11. Binary Tree Traversals

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are two standard ways of traversing a binary tree. They are:

12. Depth First Traversal

[1]. Pre Order Traversal (Root-Left-Right)

Here, the traversal is: Root – Left child – Right child. It means that the root node is traversed first then its left child and finally the right child.

To traverse a non-empty binary tree in pre order following steps one to be processed:

1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

That is, in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively.

[2]. In order Traversal (Left-Root-Right)

Here, the traversal is: left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.

The in-order traversal of a non-empty binary tree is defined as follows:

1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order

In In-order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root, the right sub tree is traversed recursively, in order fashion.

[3]. Post Order Traversal (Left-Right-Root)

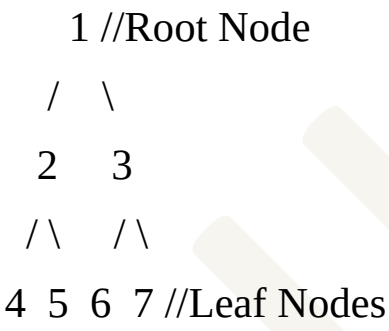
Here, the traversal is: left child – right child – root. It means that the left child is traversed first then the right child and finally its root node.

The post order traversal of a non-empty binary tree can be defined as:

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

Let us traverse the following tree with all the three traversal methods:



Pre-Order Traversal of the above tree: 1-2-4-5-3-6-7
In-Order Traversal of the above tree: 4-2-5-1-6-3-7
Post-Order Traversal of the above tree: 4-5-2-6-7-3-1

13. Breadth First Traversal

In this type of traversal, the tree is traversed by level by level. Hence values on Level 0 are traversed first, then Level 1 then level 2 and so on…

It is also known as Level-Order-Traversal.

Hence for the same example Level-Order Traversal or Breadth First Traversal would be: 1-2-3-4-5-6-7.

12. Numerical Based on Finding Traversal from Given Tree.

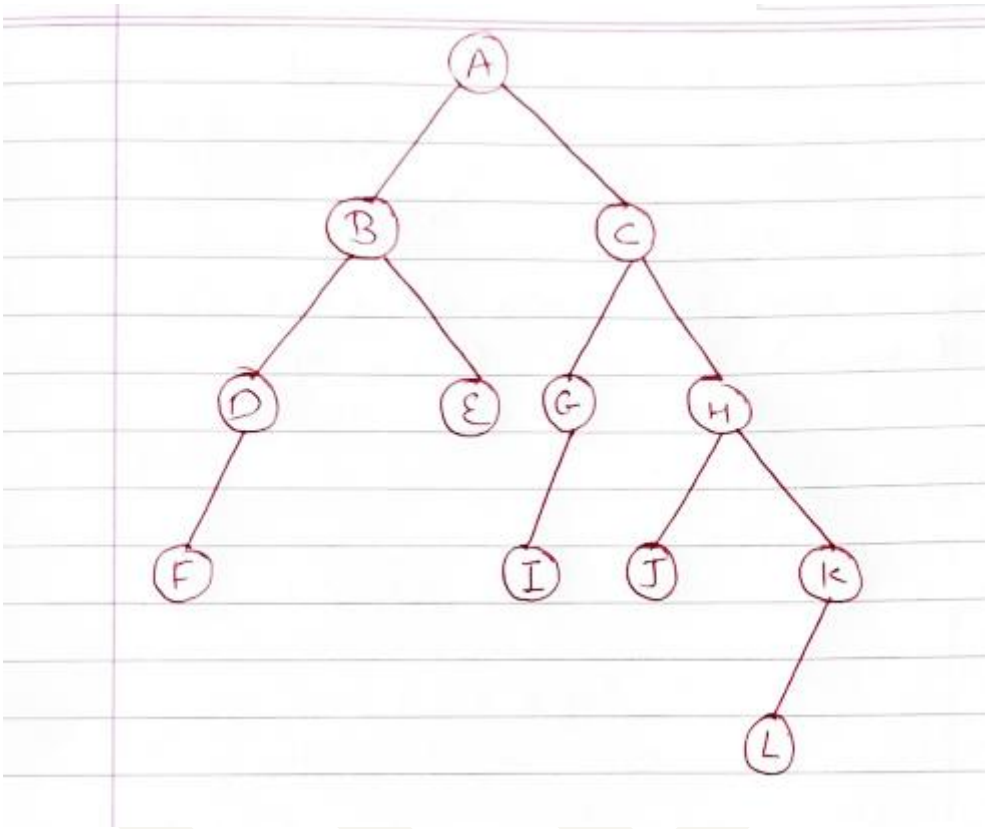
To find the Traversal of the given Tree, you may follow the definition provided in the Topic 9.

Also, alternatively the following method can be used to find traversals.

a) Rules to Find Traversal.

1. **Make Sure that Each Node in Root must have 2 children.**
    a. **A Leaf Node do not have children hence we need to make 2 dummy children of it.**
    b. **If any internal node has only 1 Left child, then we need to make another dummy right child. OR if any internal node has only 1 right child, then we need to make another dummy left child.**
2. **Start Traversing from Root Node.**
3. **While traversing the Tree if any value is met for 1st time, we shall write it in Pre-Order Traversal.**
4. **If any value is met for 2nd time, we shall write it in In-Order Traversal.**
5. **And, if any value is met for 3rd time, we shall write it in Post-Order Traversal.**

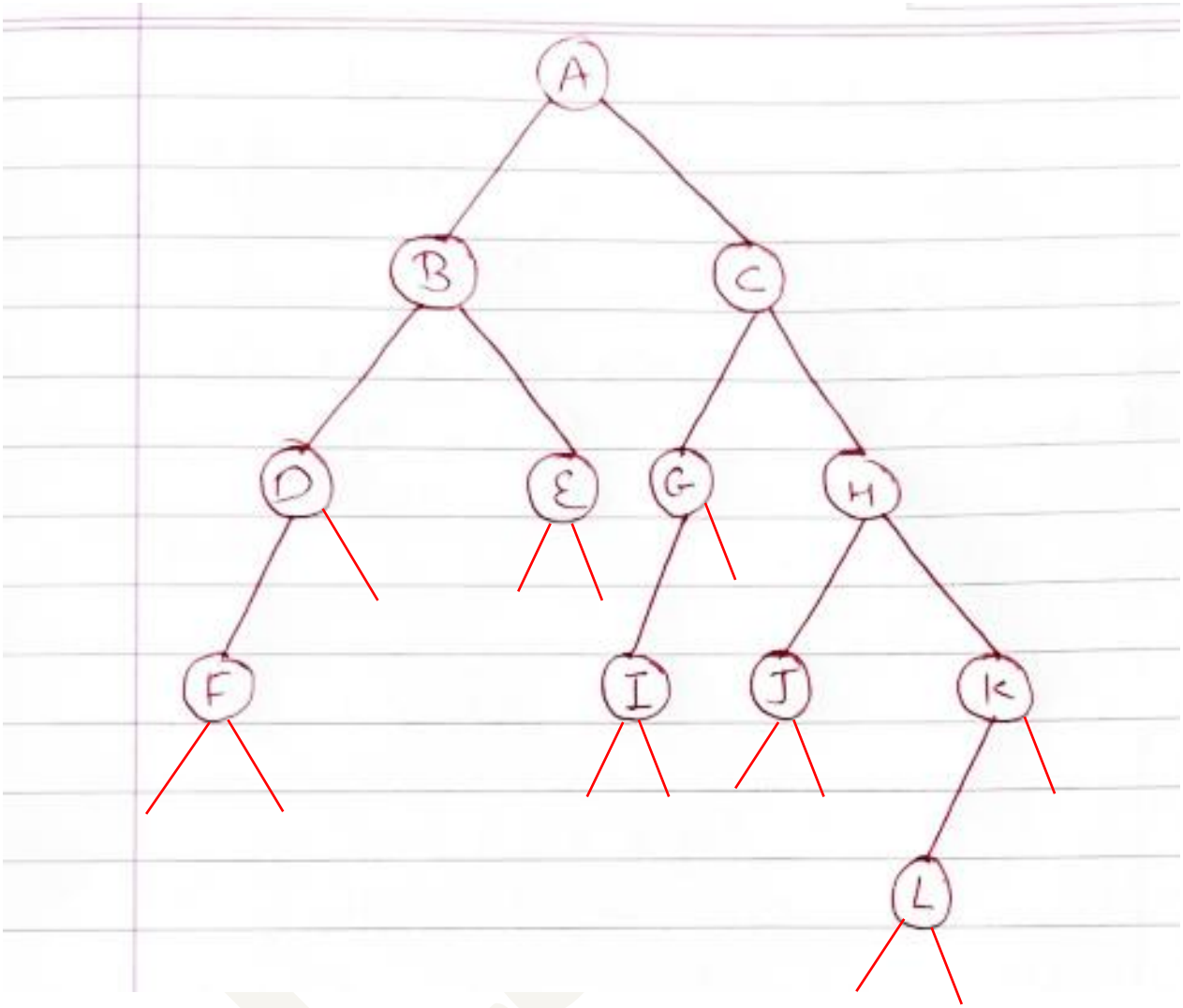
Q.1(Q.B 110) Find the Pre-Order, In-Order & Post-Order Traversal of the given tree.



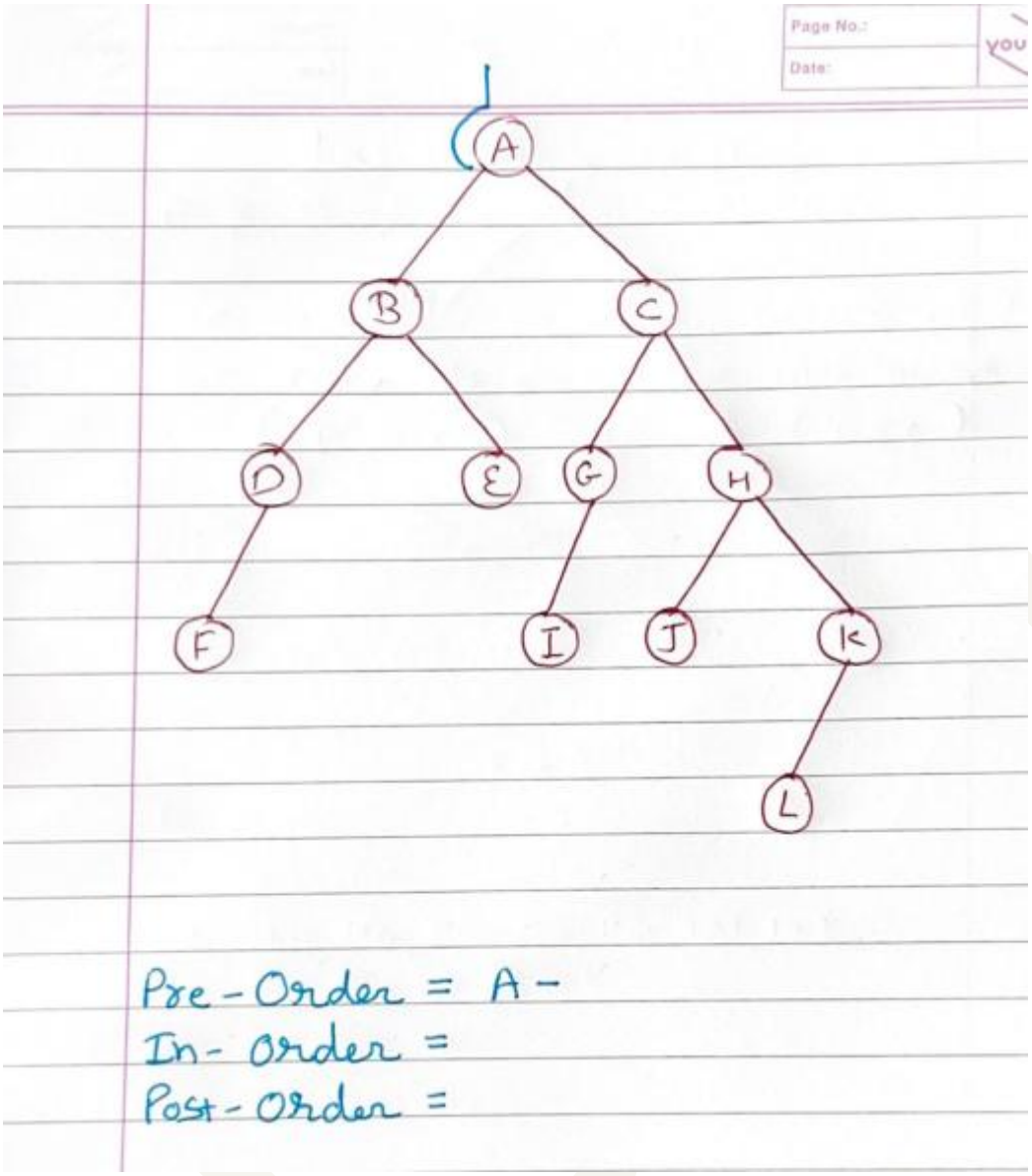
Solution:

Note – For the very first numerical, I have shown the step-by-step process.

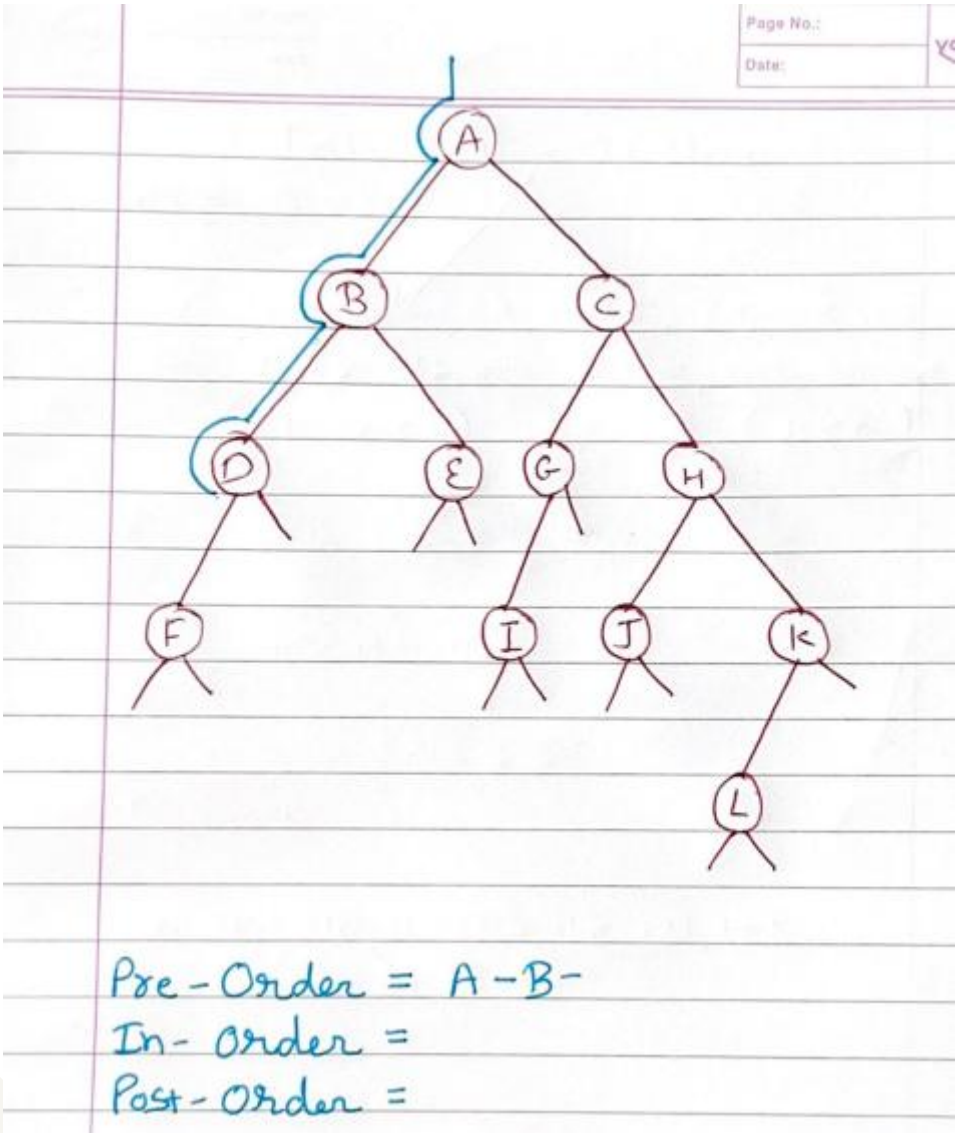
Step 1 – Make dummy children as necessary.



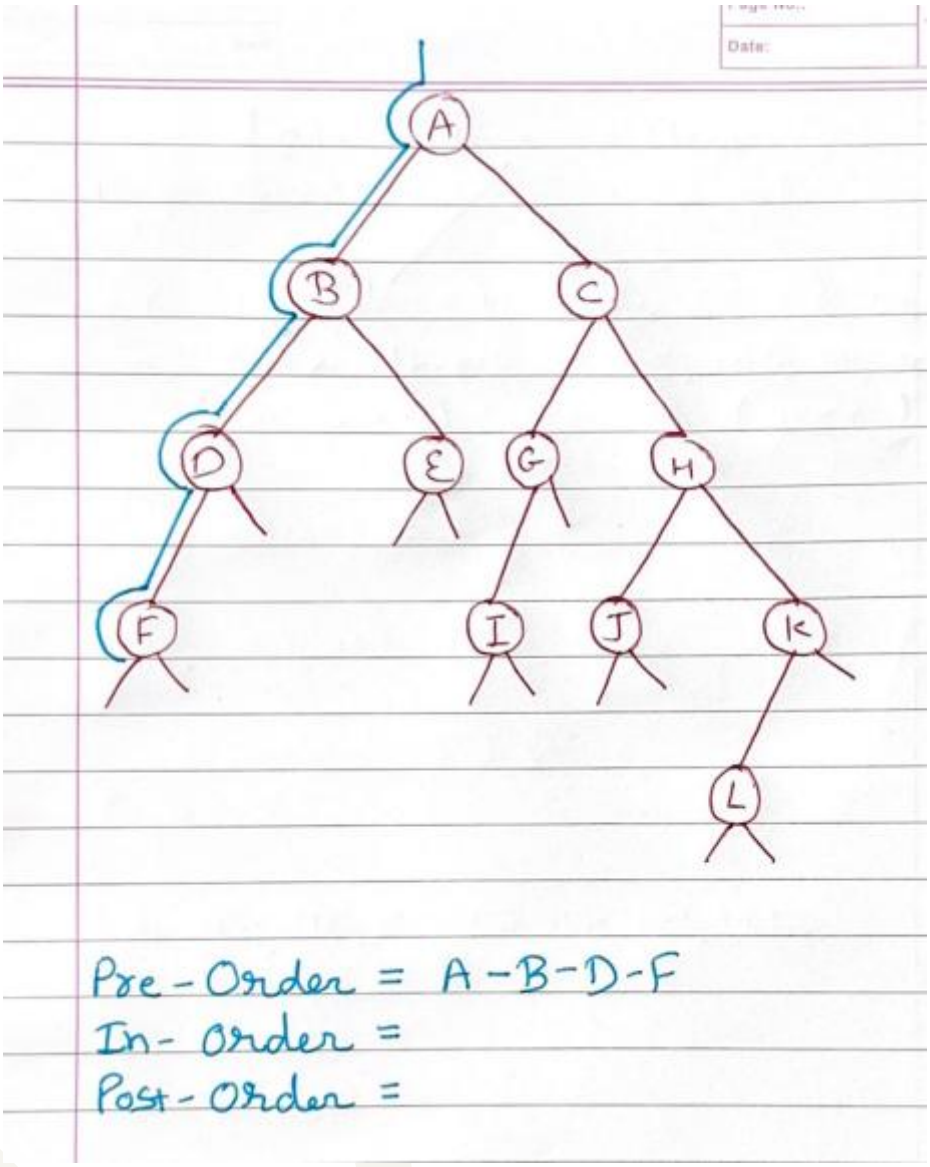
Step 2 – Start from the Root Node.



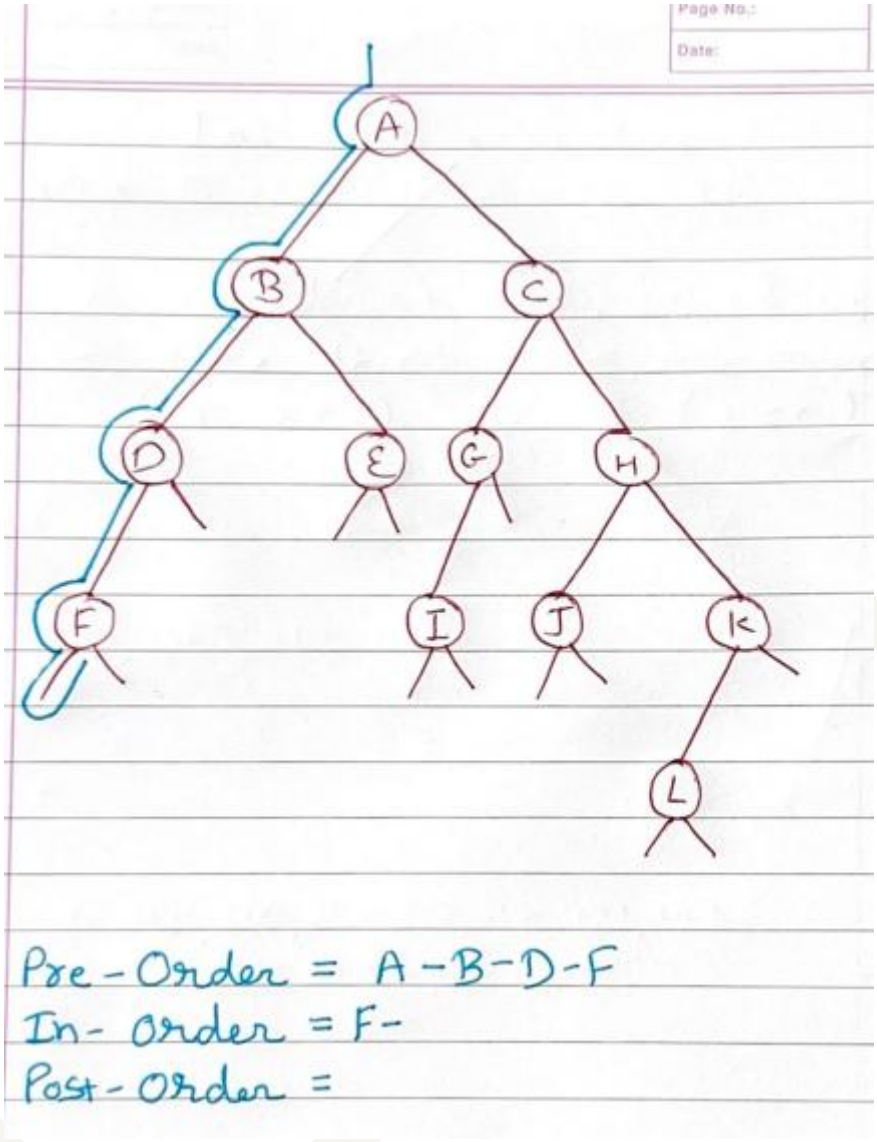
Step 3 – Start Traversing from Left to Right. Next, we met Node B for 1st time. Hence, we will write B in pre-order.



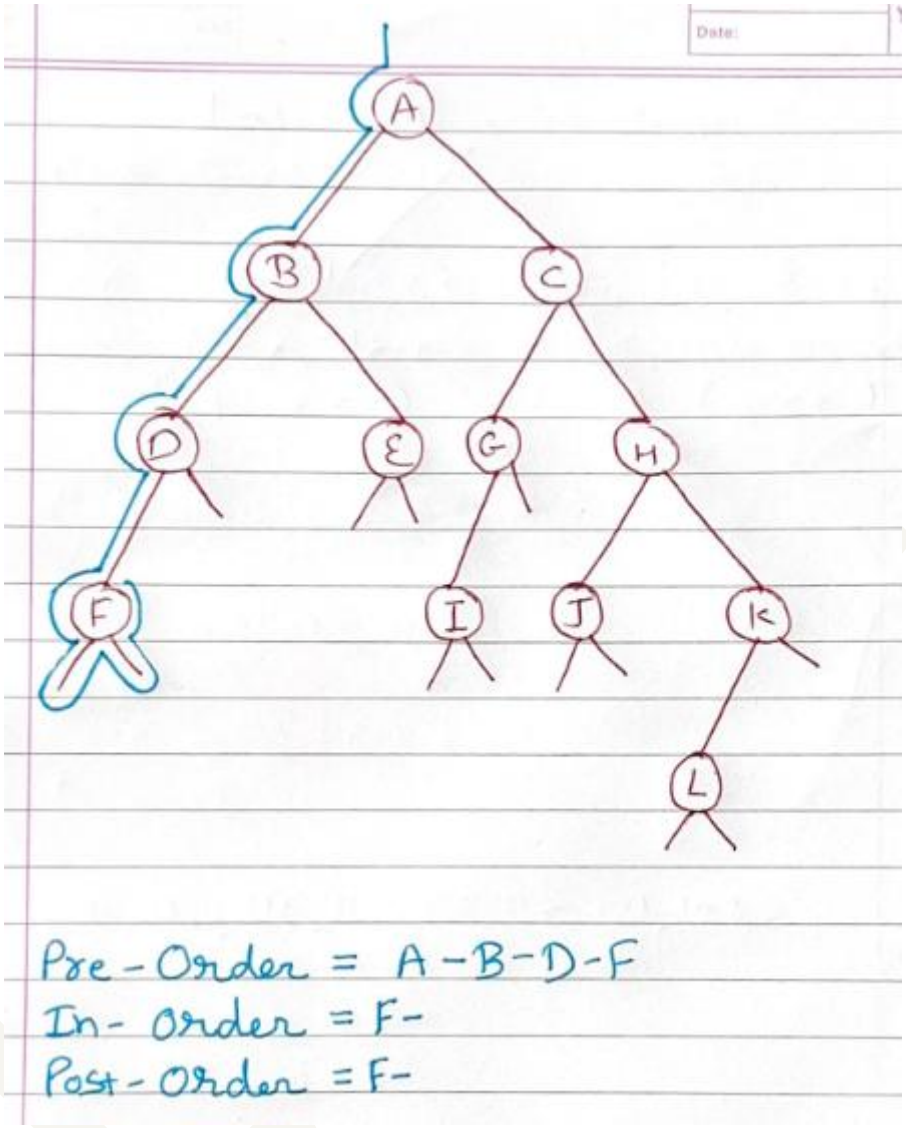
Step 4 – Now, similarly we met D & F for the first time.



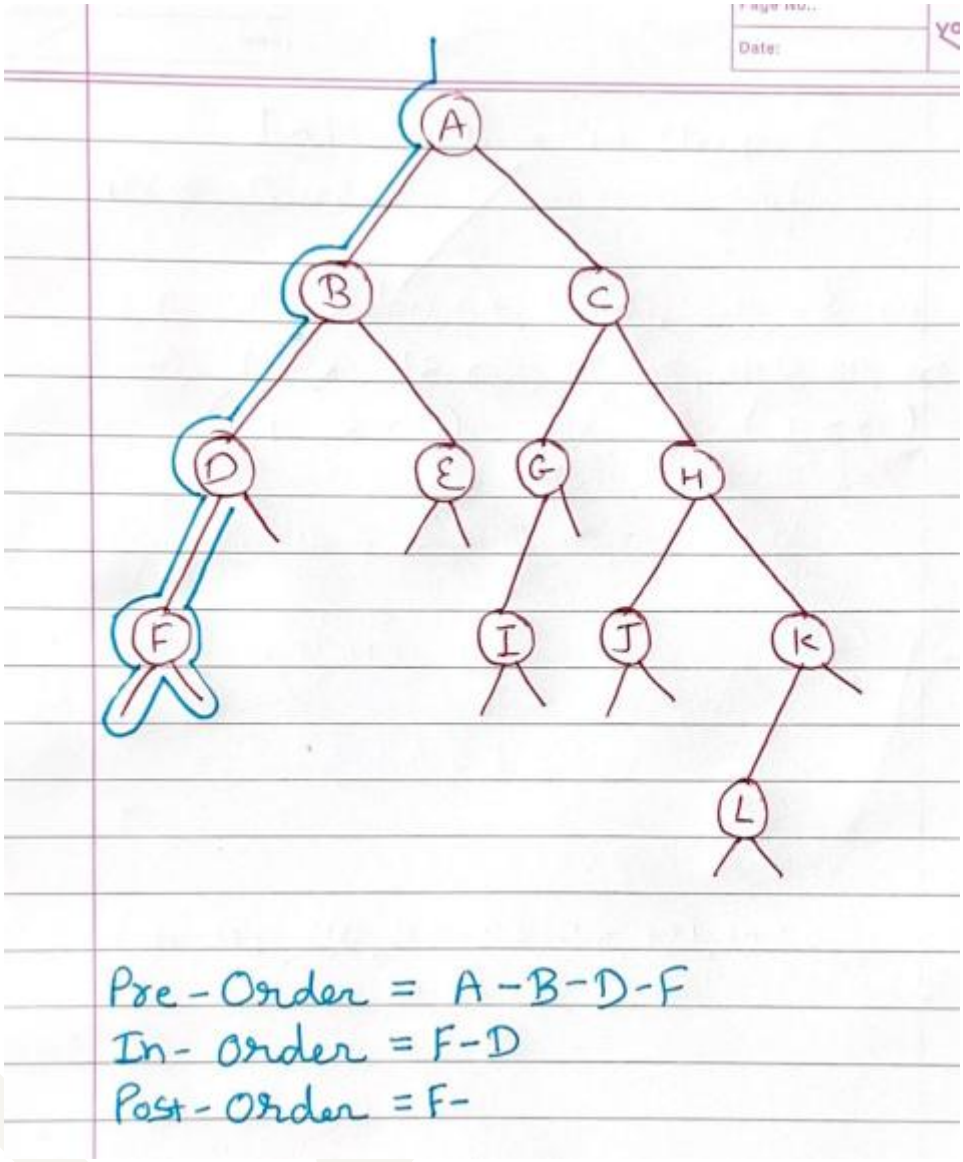
Step 5 – Now, as you can see, we met F for the 2nd time as we turned inside the tree. So, we shall write F in In-Order traversal.



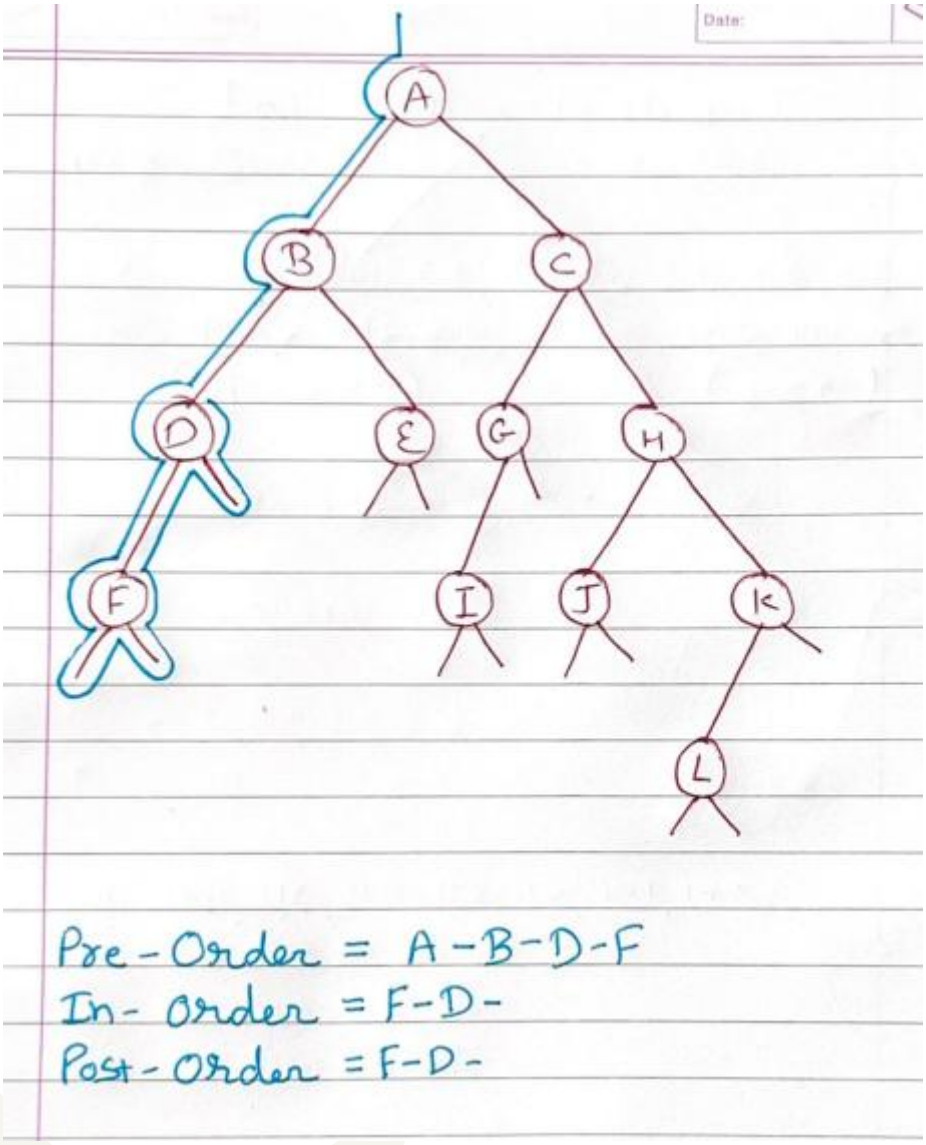
Step 6 – Now, we go ahead as seen in figure and met F for the 3rd time. Hence, we write it in Post-Order traversal.



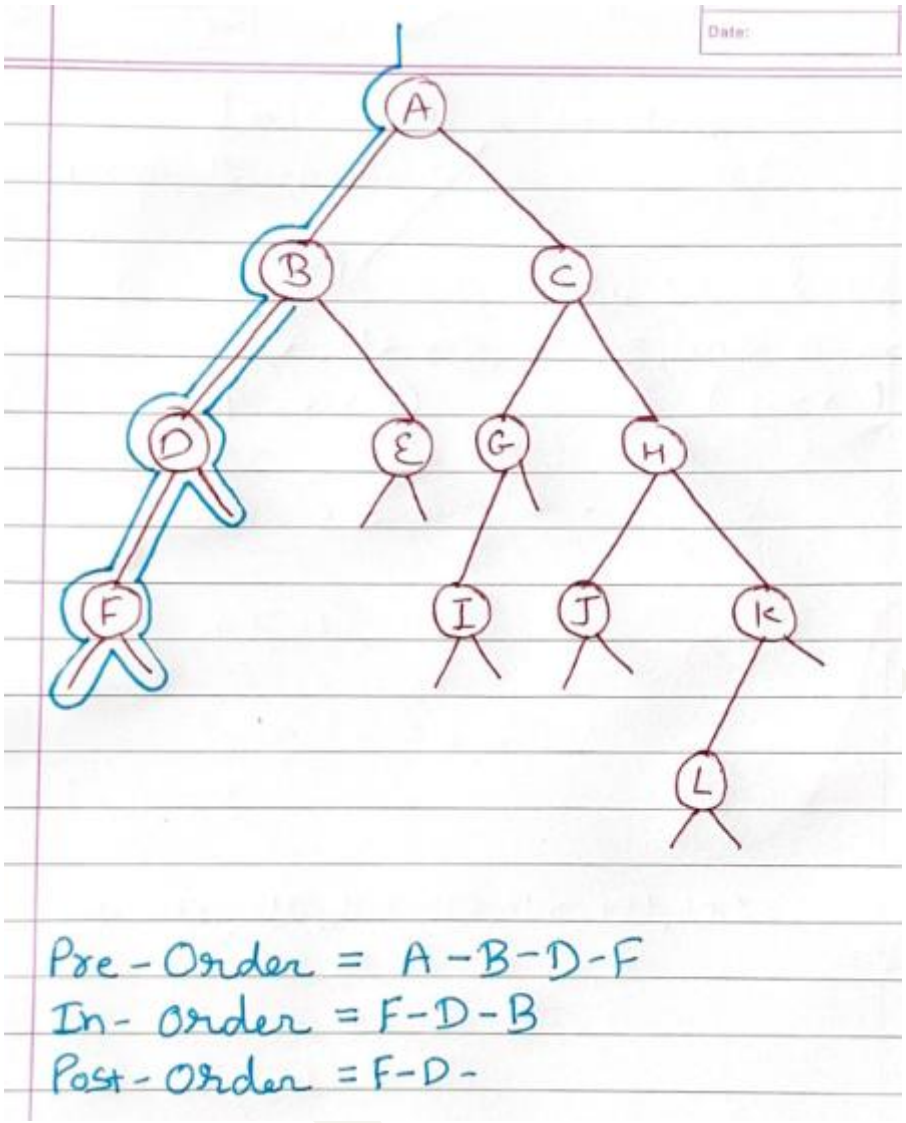
Step 7 – Now, we go ahead and met D for the 2nd time. Hence write it in In-Order.



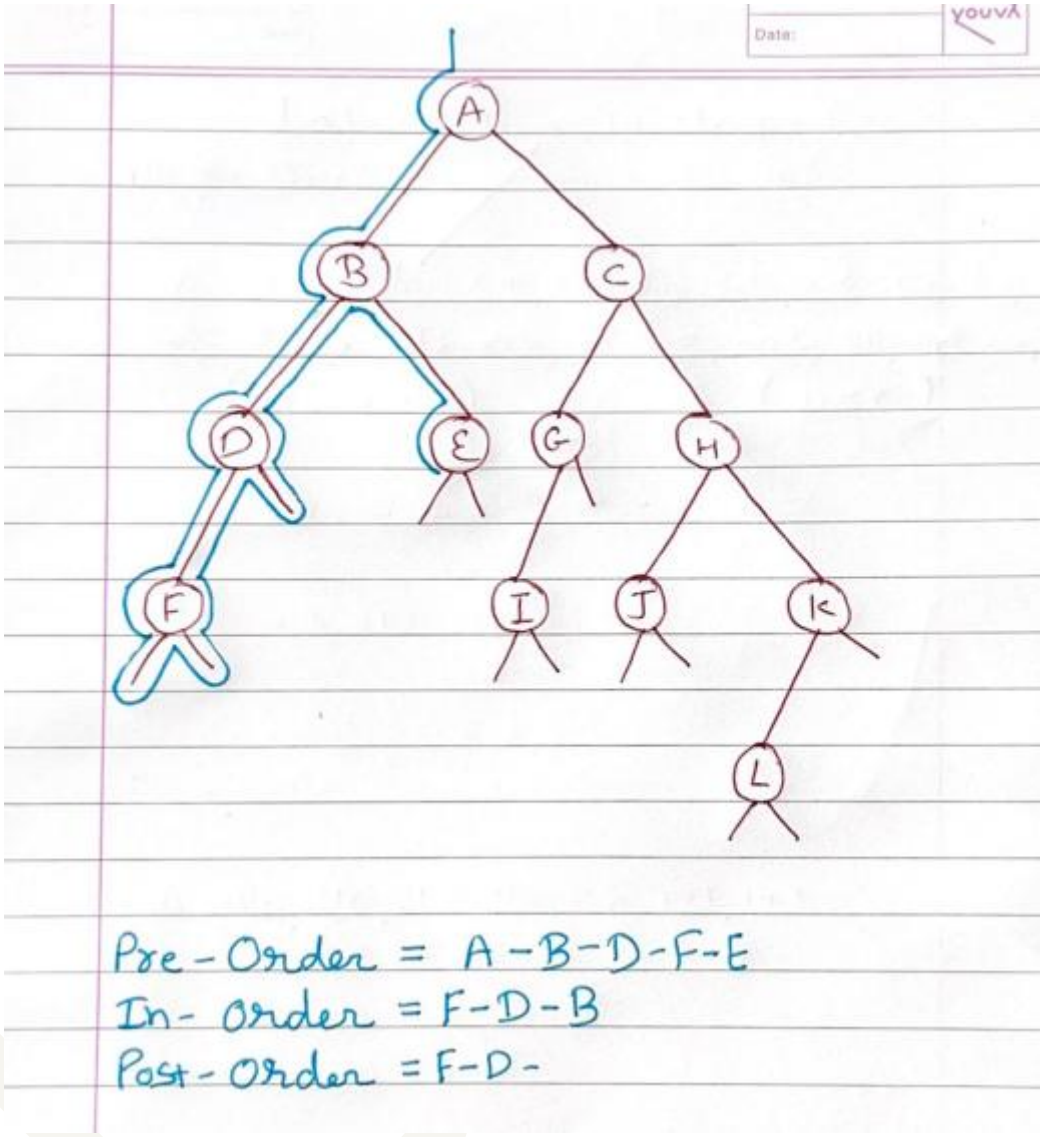
Step 8 – Now, we go ahead and met D for the 3rd time. Hence, write it in Post-Order.



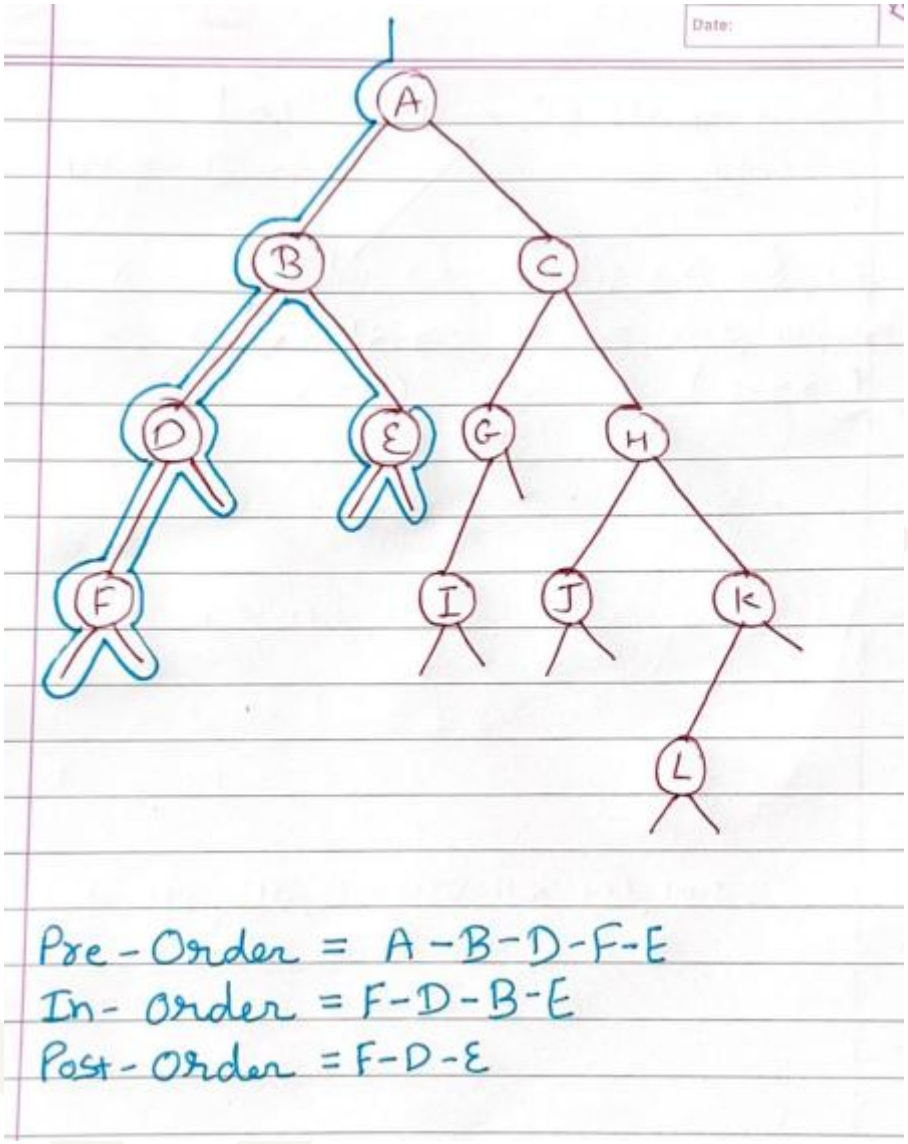
Step 9 – Now, we met B for the 2nd time. Hence write it in In-Order traversal.



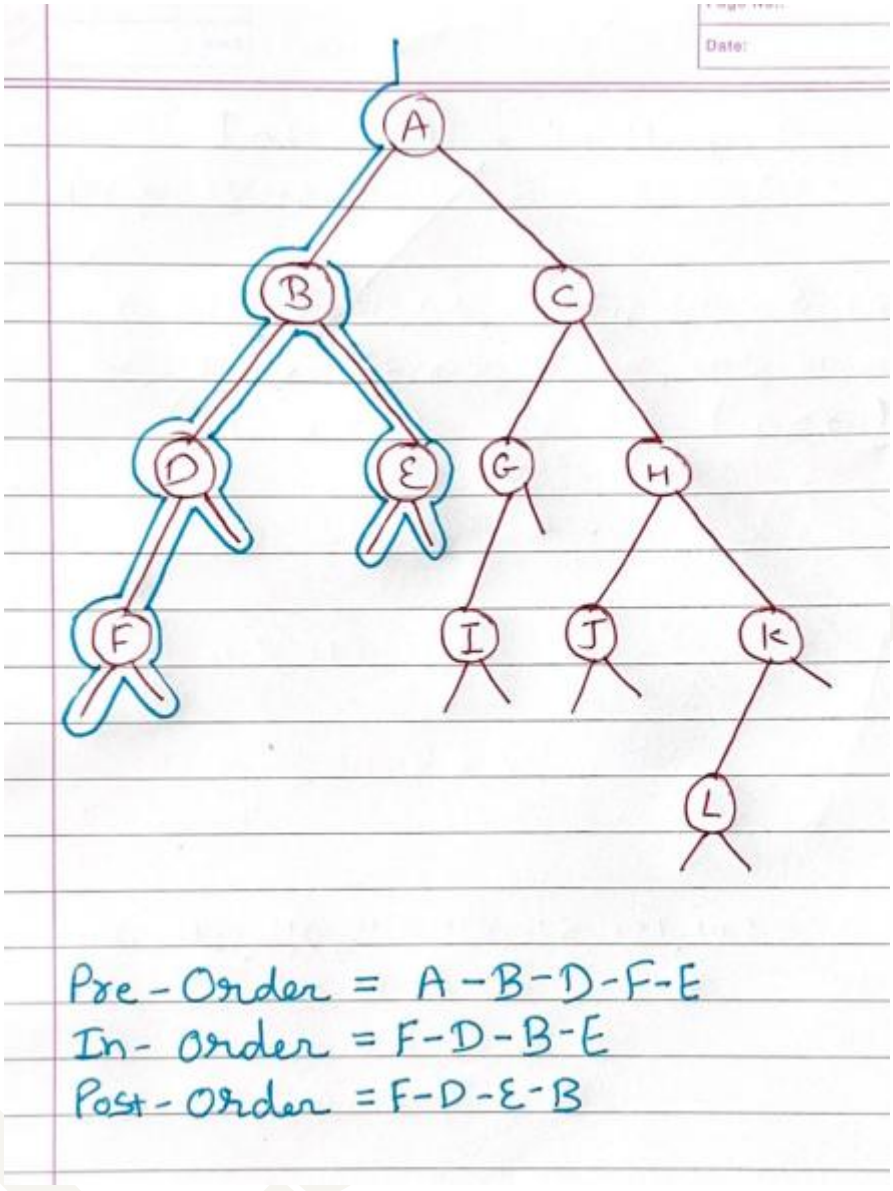
Step 10 – Now, we go ahead and met E for the 1st time hence we will write it in Pre-Order.



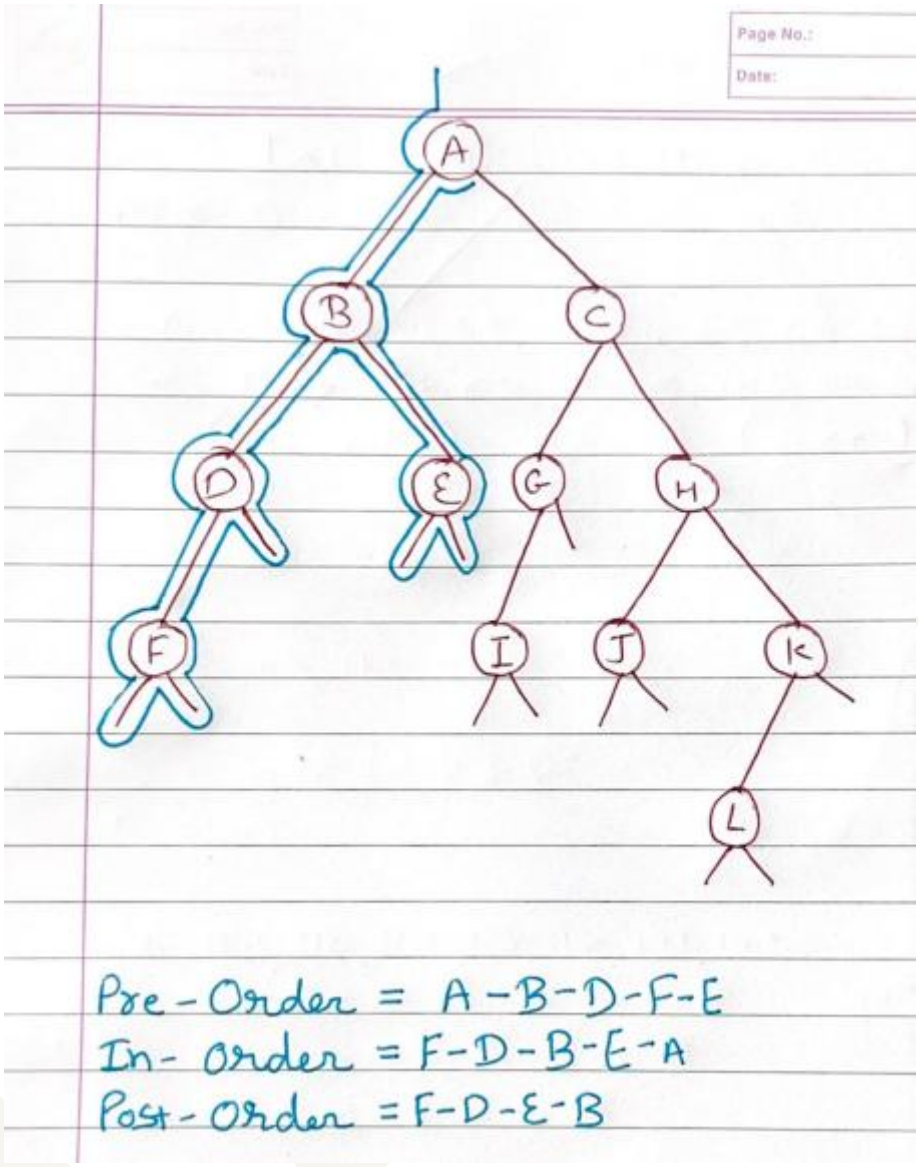
Step 11 – Now, we met E for the 2nd and 3rd time respectively. Hence, we will write E in In-Order and Post -Order.



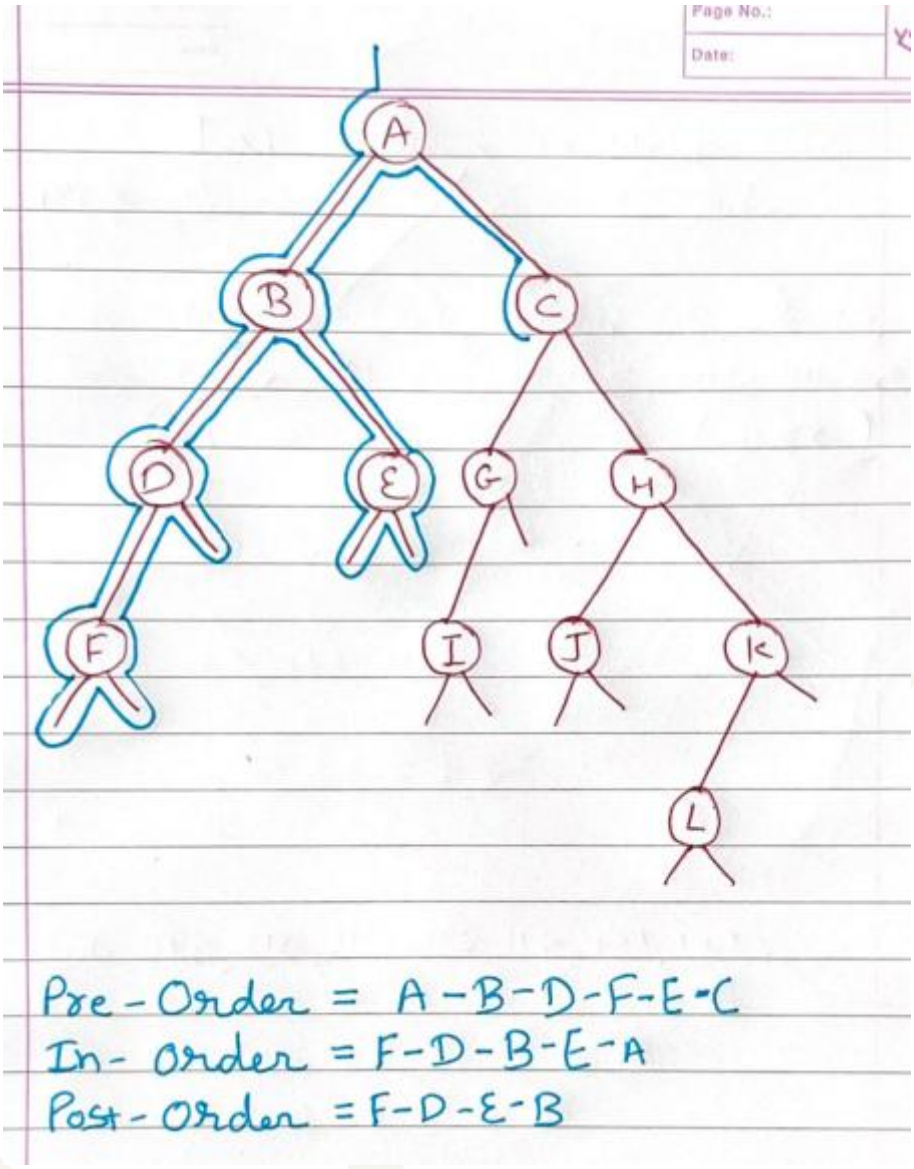
Step 12 – Now, we met B for the 3rd time in the traversal. Hence, write it in Post-Order.



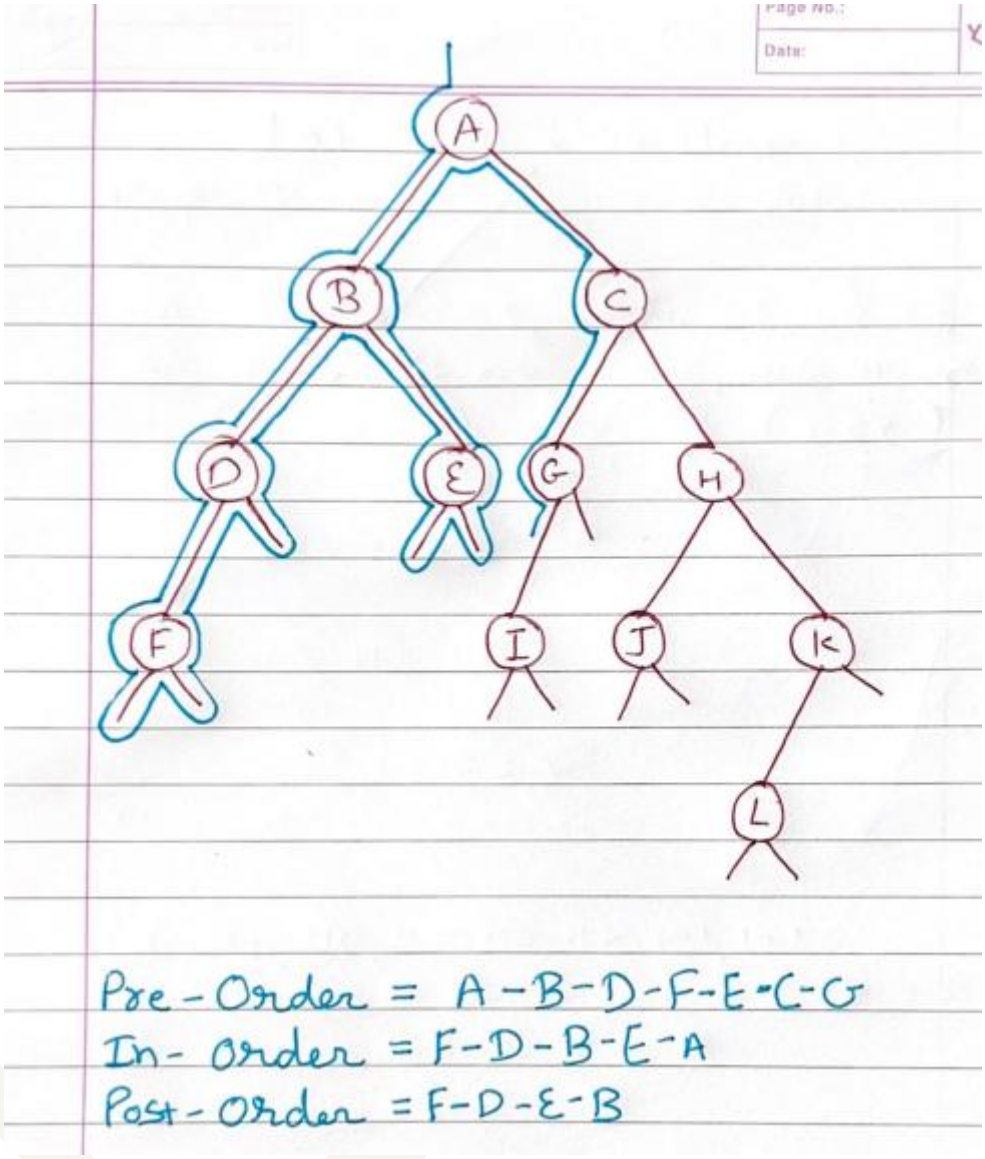
Step 13 - Now, going ahead we met A for the 2nd time. Hence, write it in In-Order.



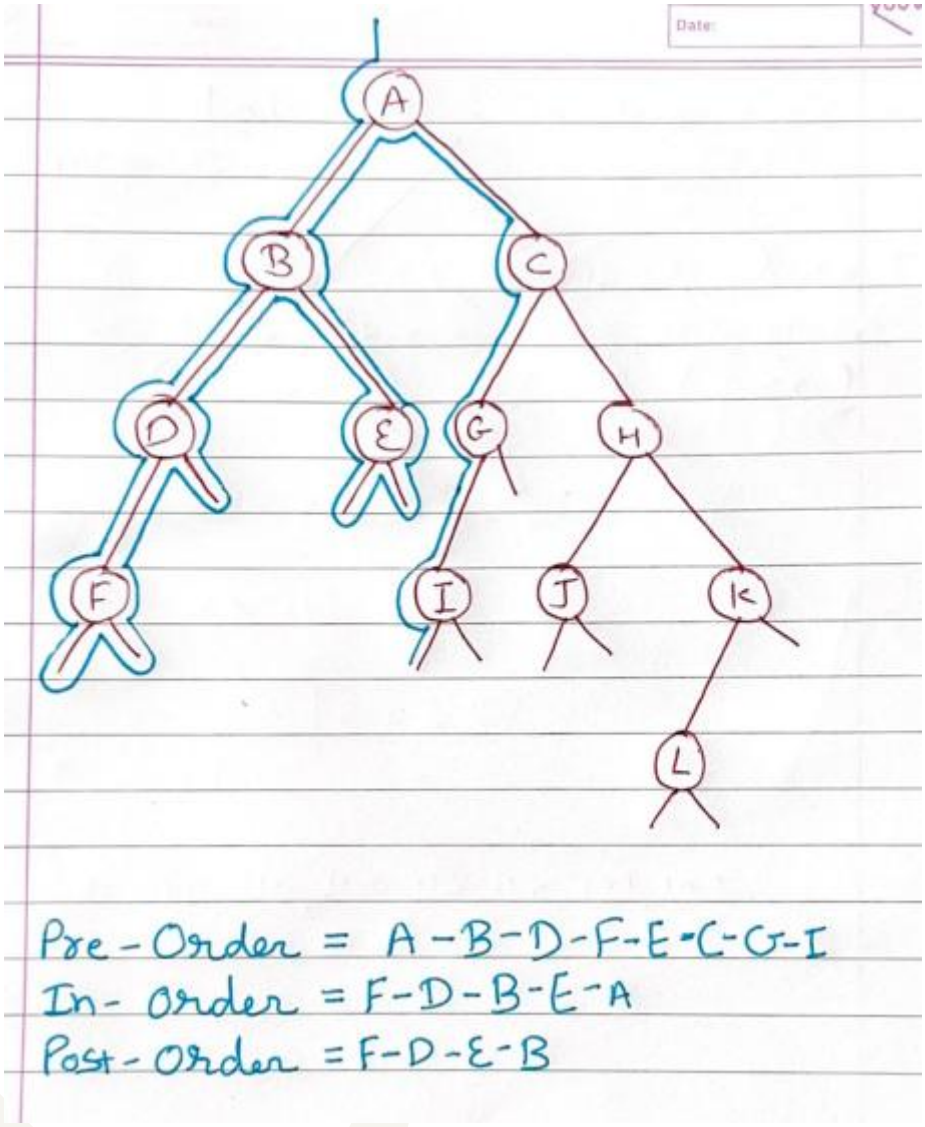
Step 14 – Now, we met C for the 1st time. Hence, we will write it in Pre-Order.



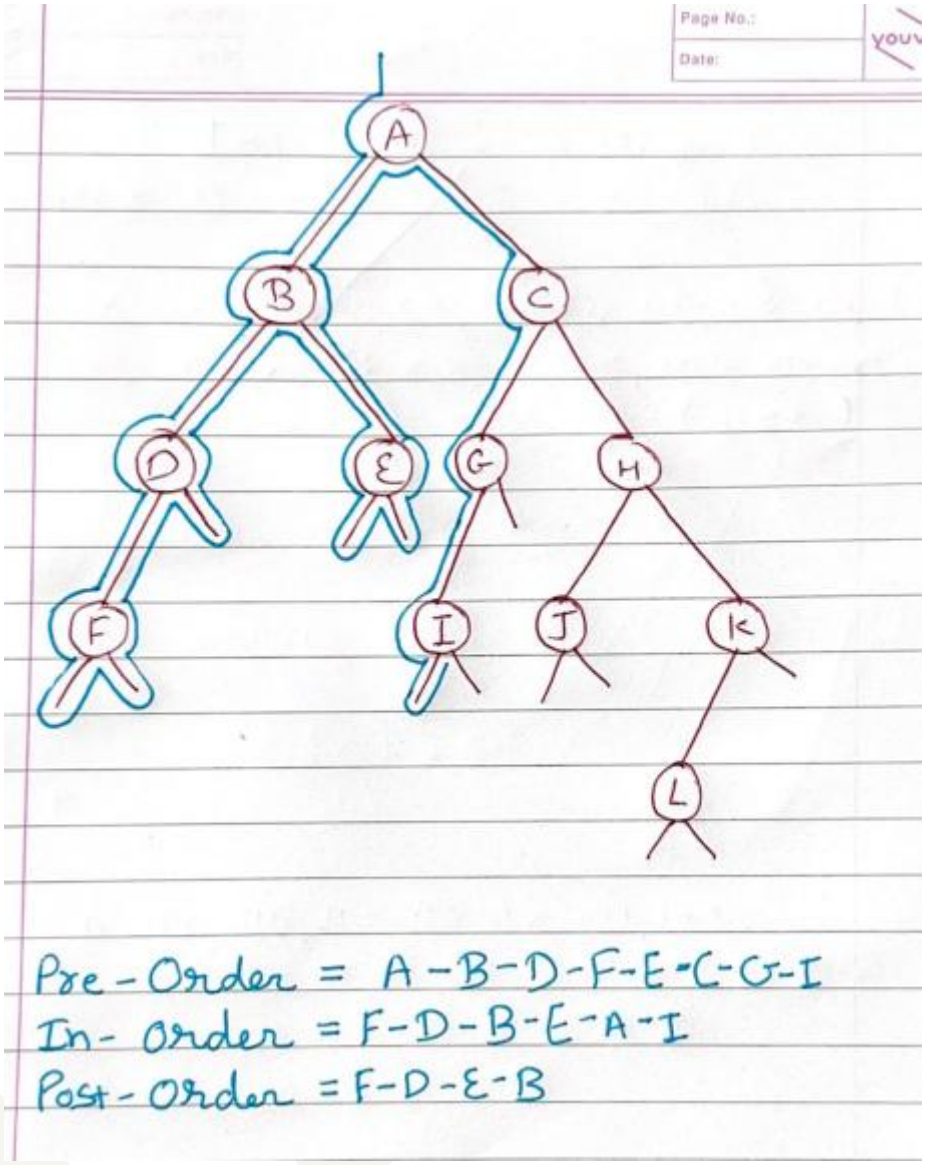
Step 15 – Now, going ahead we met G, for the 1st time. Hence, we will write it in Pre-Order.



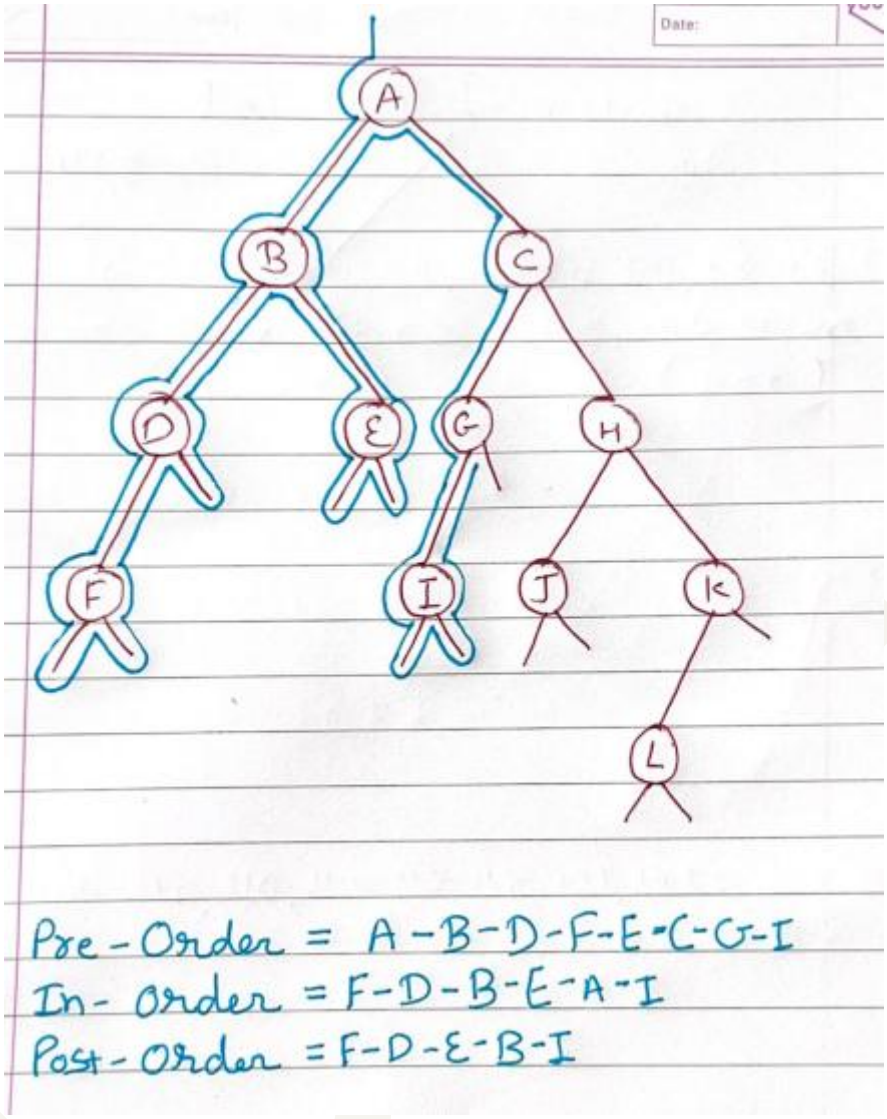
Step 16 – Now, going ahead we met I, for the 1st time. Hence, we will write it in Pre-Order.



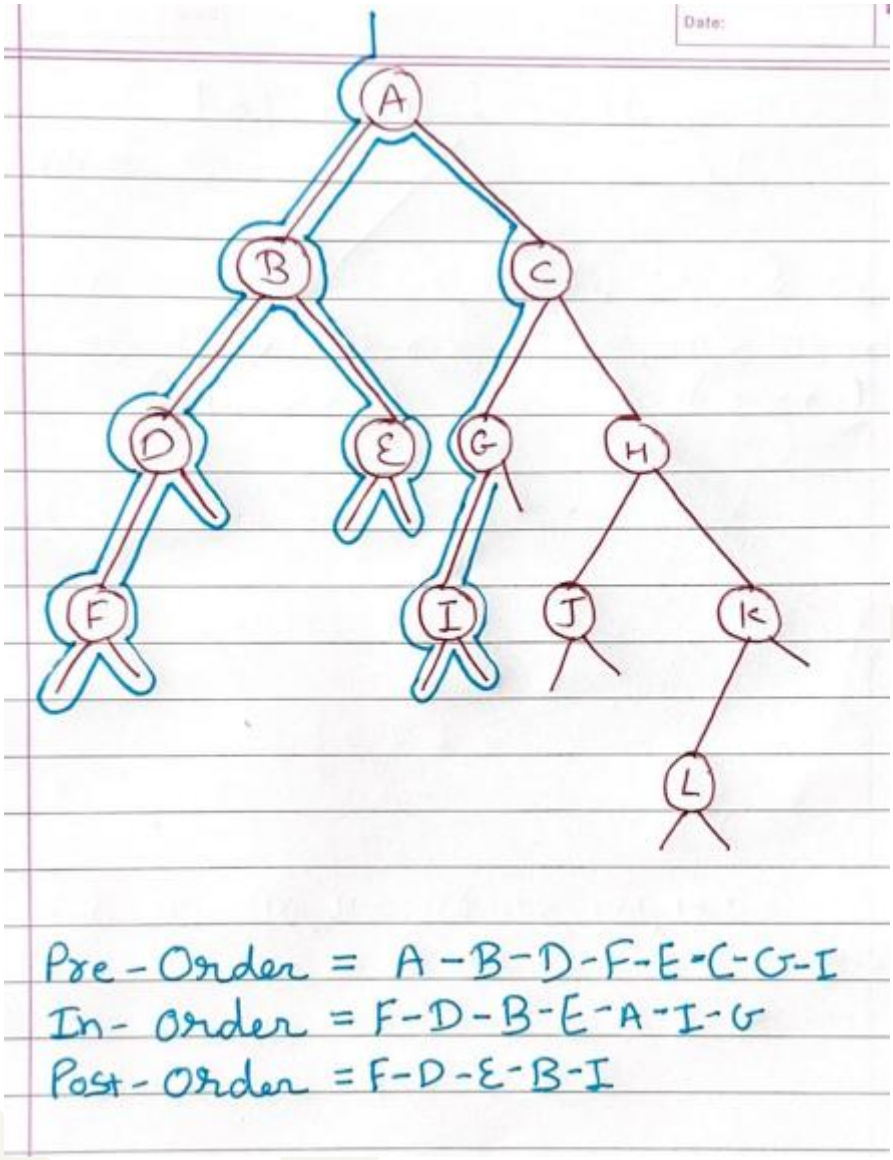
Step 17– Now, going ahead we met I, for the 2nd time. Hence, we will write it in In-Order.



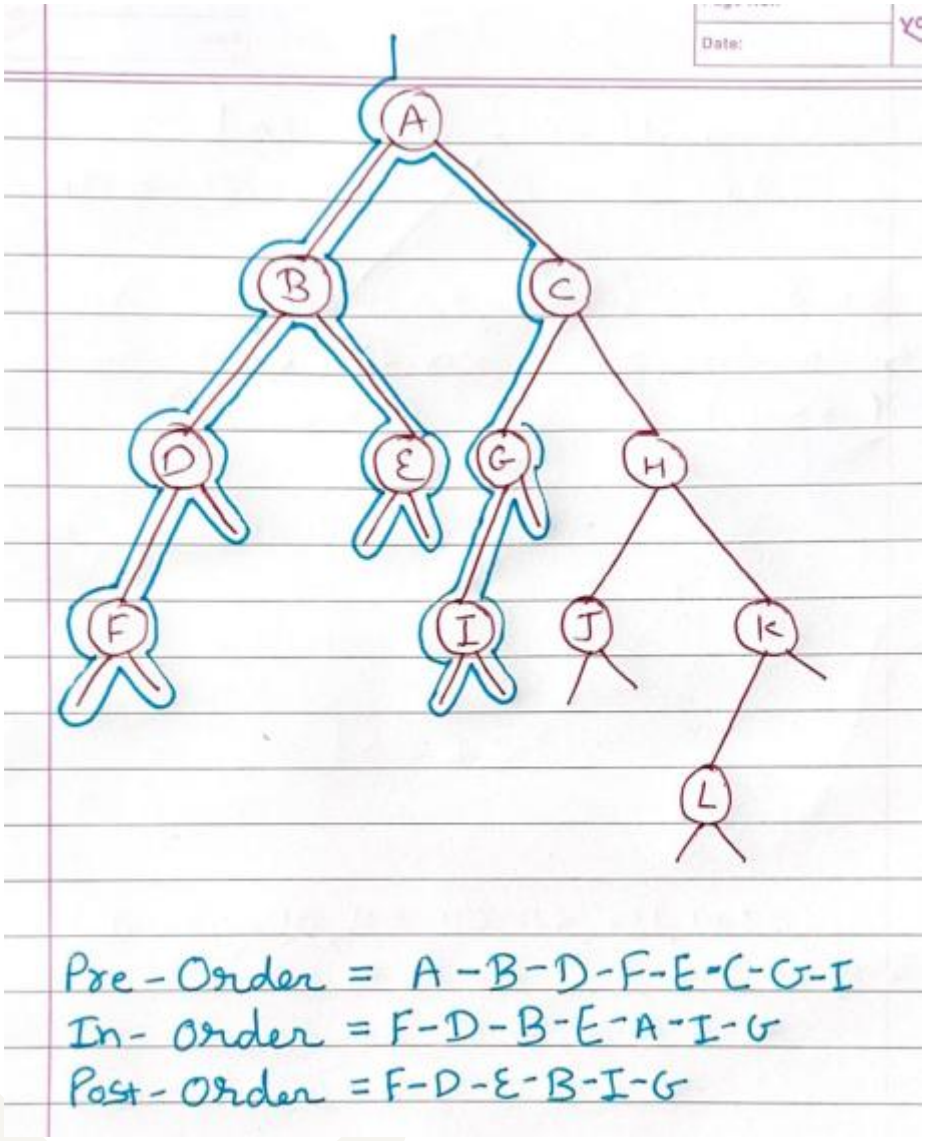
Step 18– Now, going ahead we met I, for the 3rd time. Hence, we will write it in Post-Order.



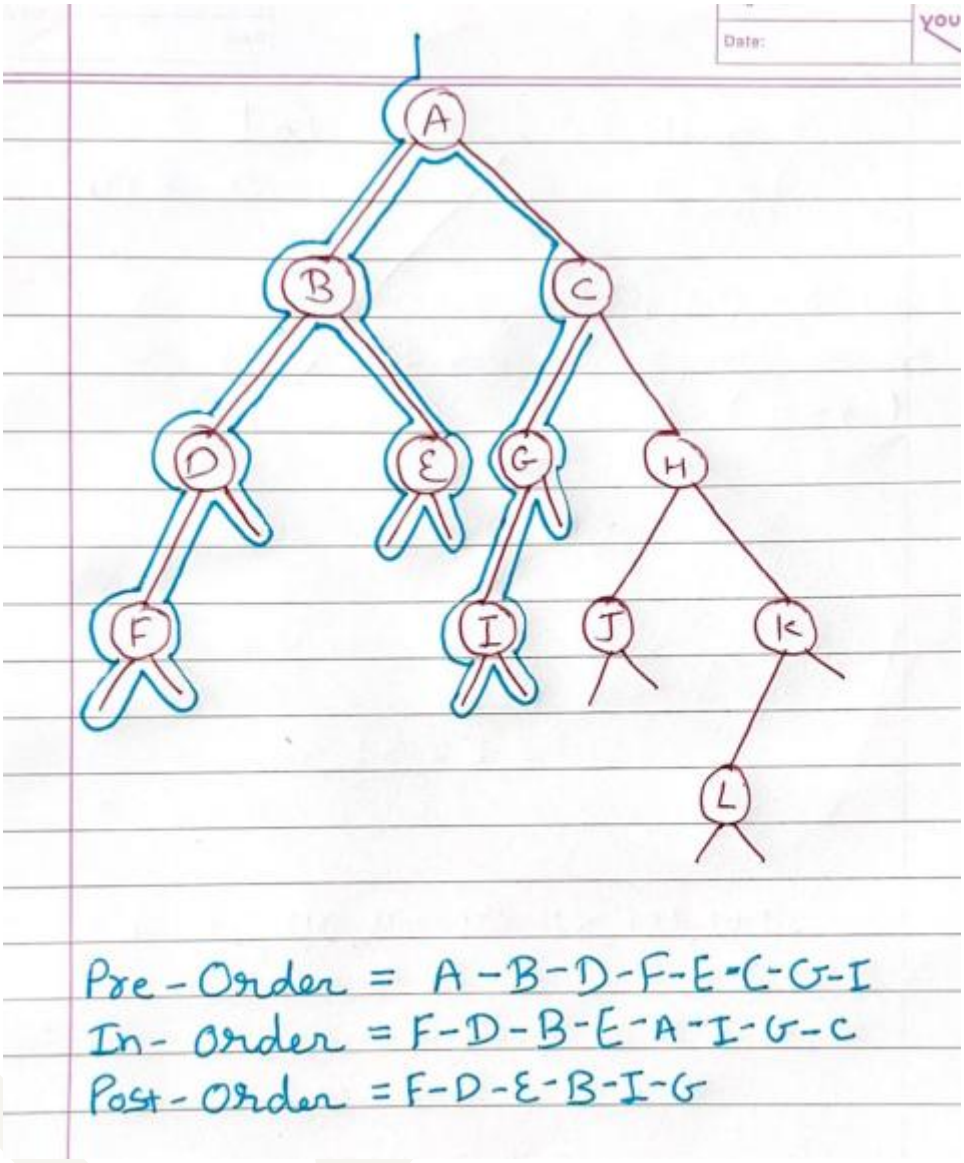
Step 19– Now, going ahead we met G, for the 2nd time. Hence, we will write it in In-Order.



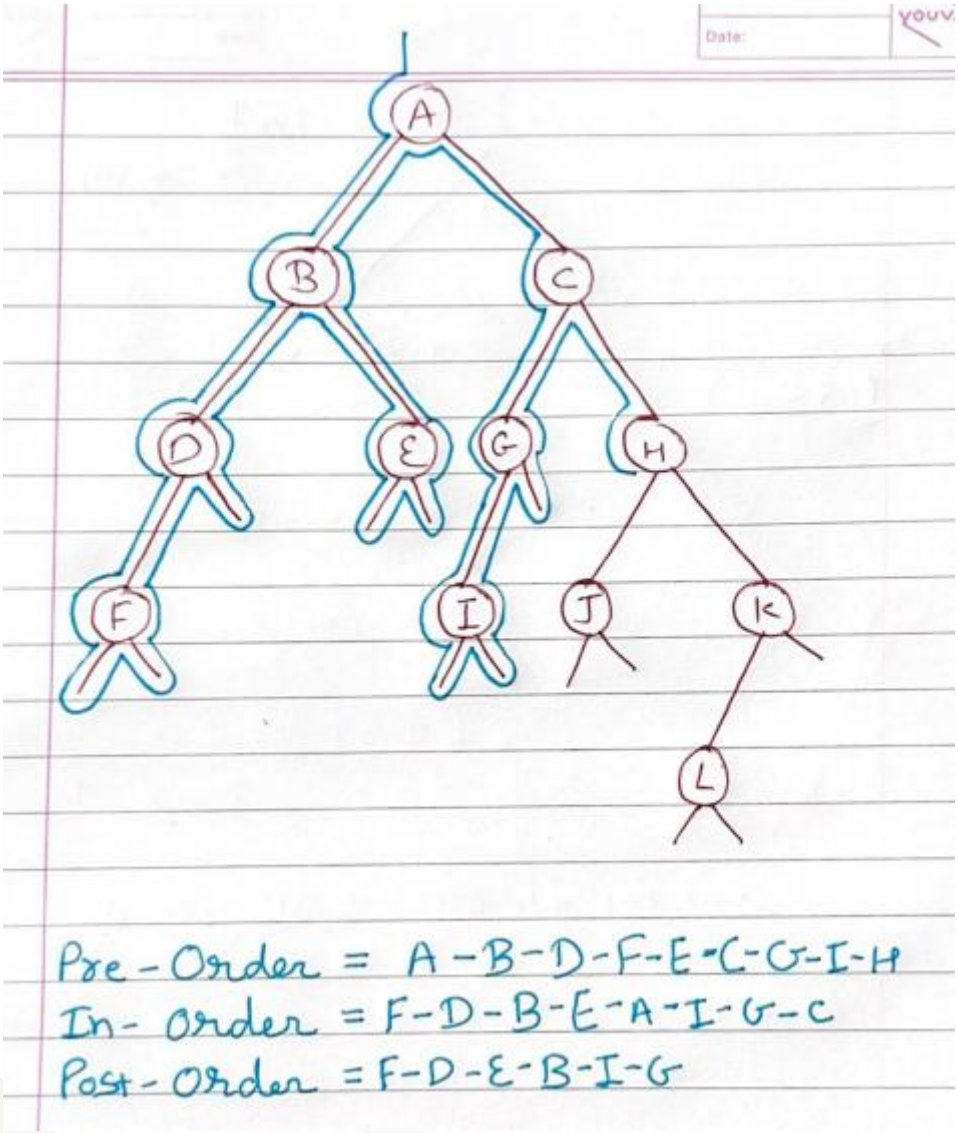
Step 20– Now, going ahead we met G, for the 3rd time. Hence, we will write it in Post-Order.



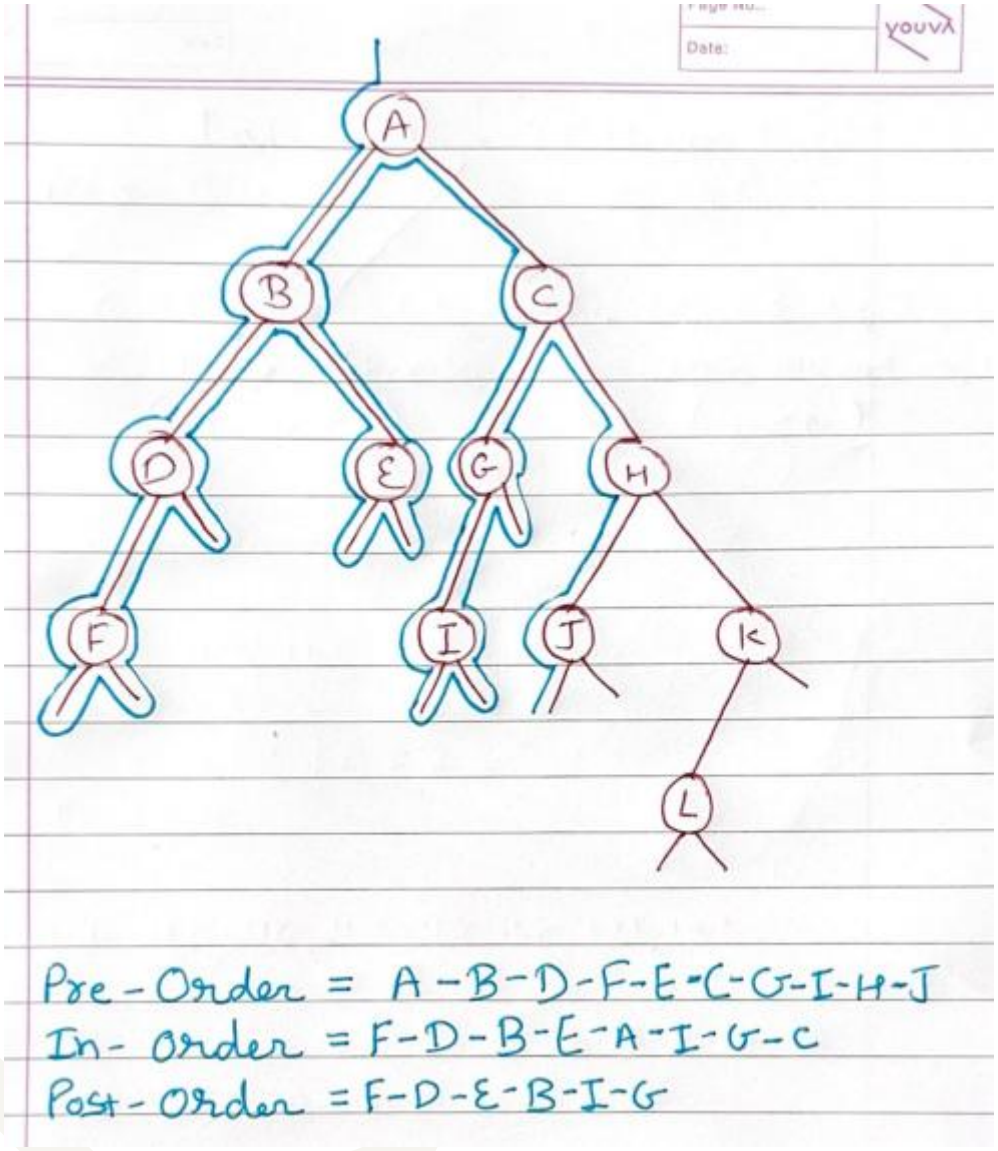
Step 21– Now, going ahead we met C, for the 2nd time. Hence, we will write it in In-Order.



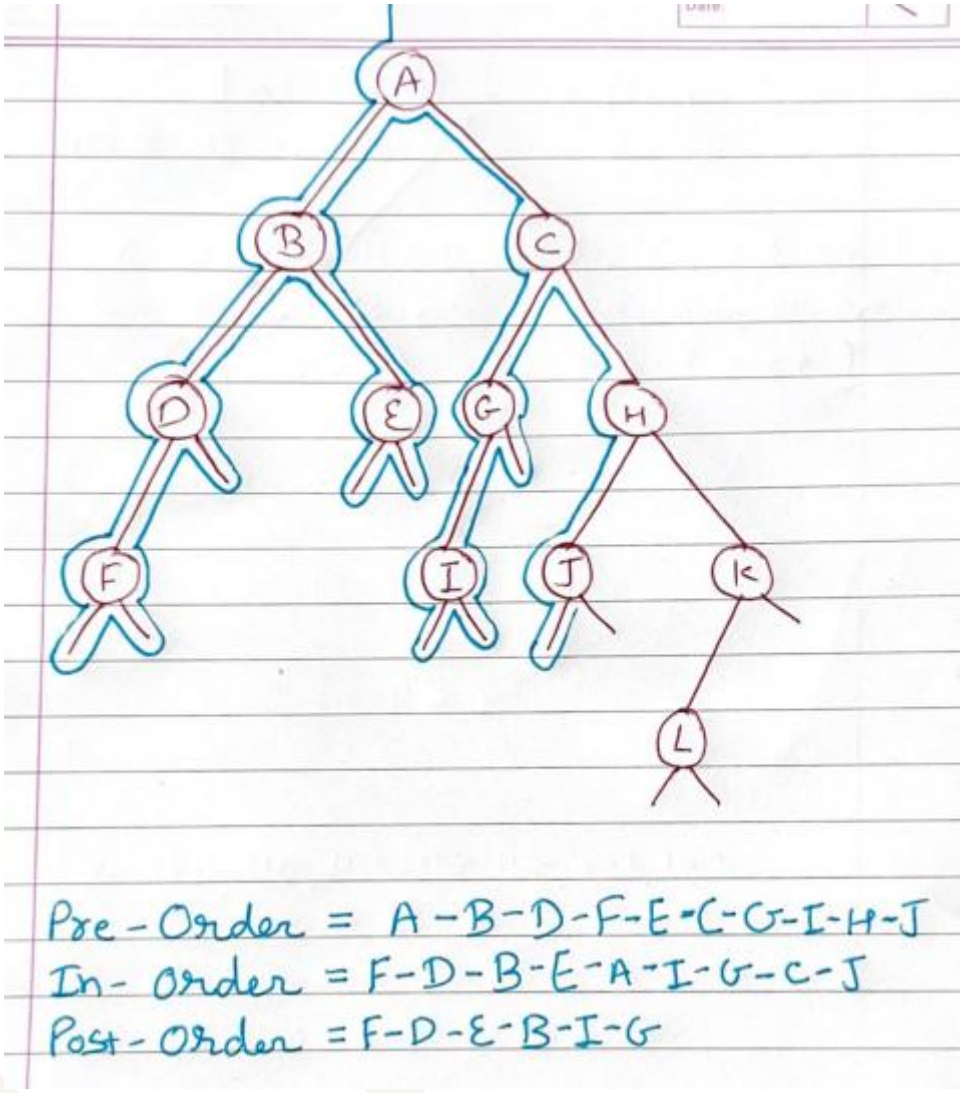
Step 22– Now, going ahead we met H, for the 1st time. Hence, we will write it in Pre-Order.



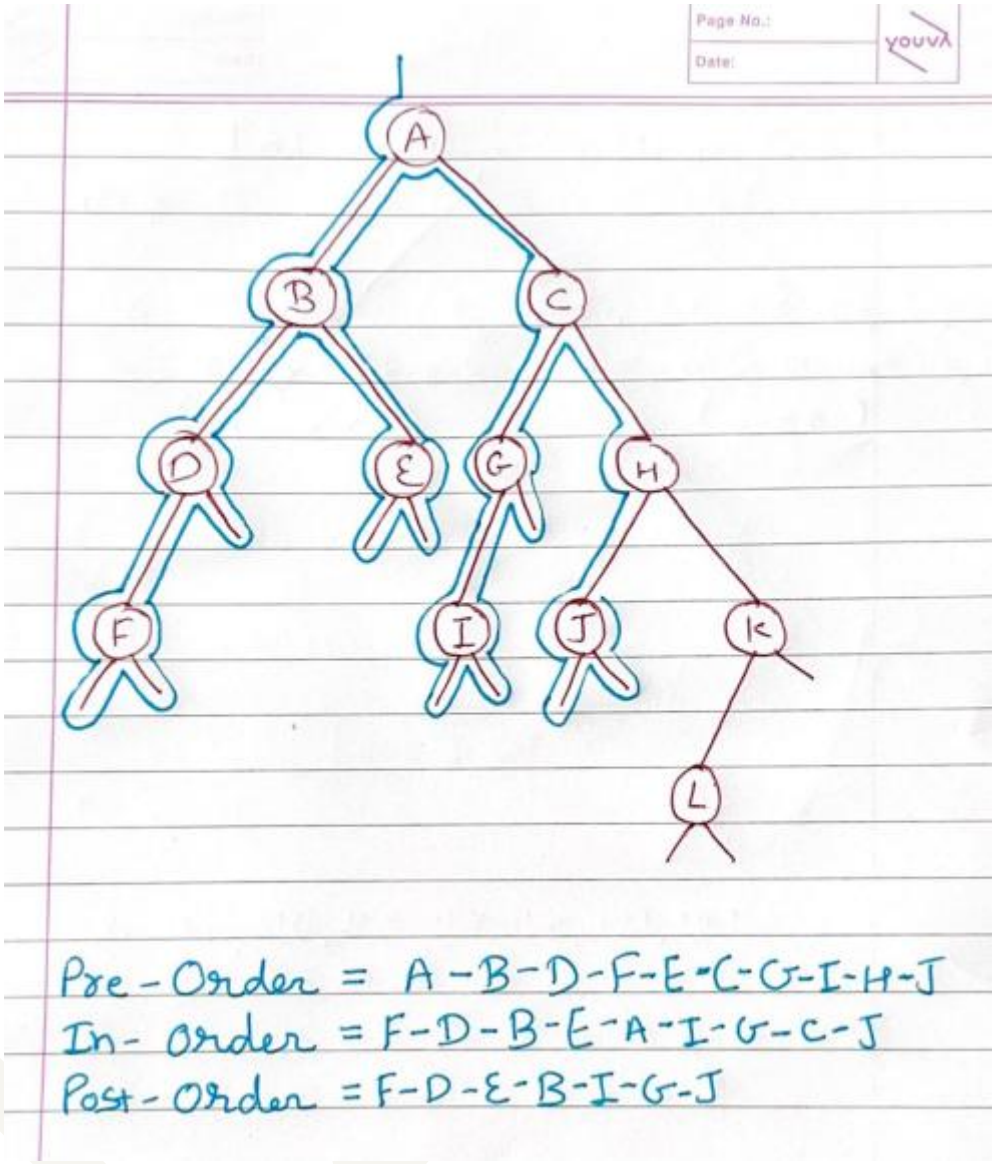
Step 23– Now, going ahead we met J, for the 1st time. Hence, we will write it in Pre-Order.



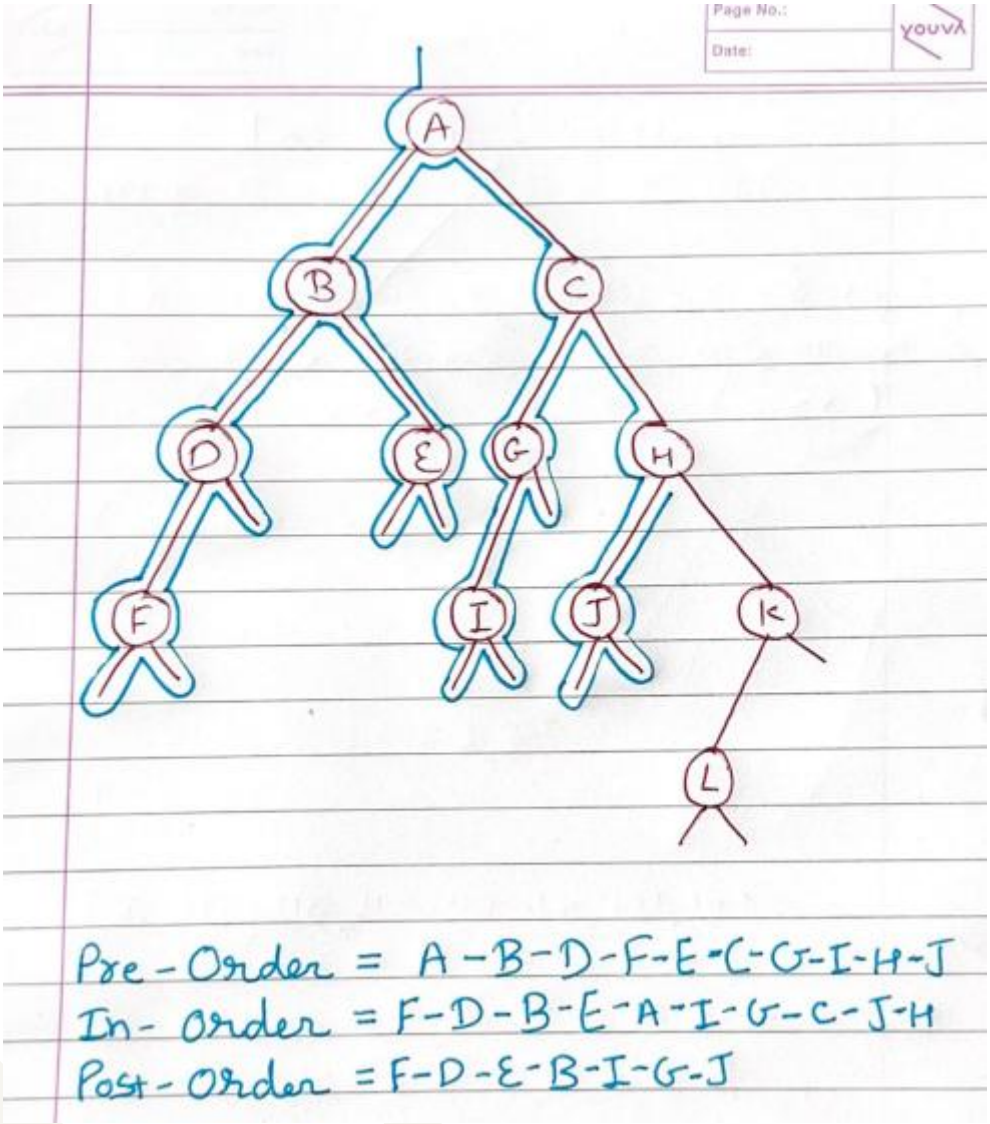
Step 24– Now, going ahead we met J, for the 2nd time. Hence, we will write it in In-Order.



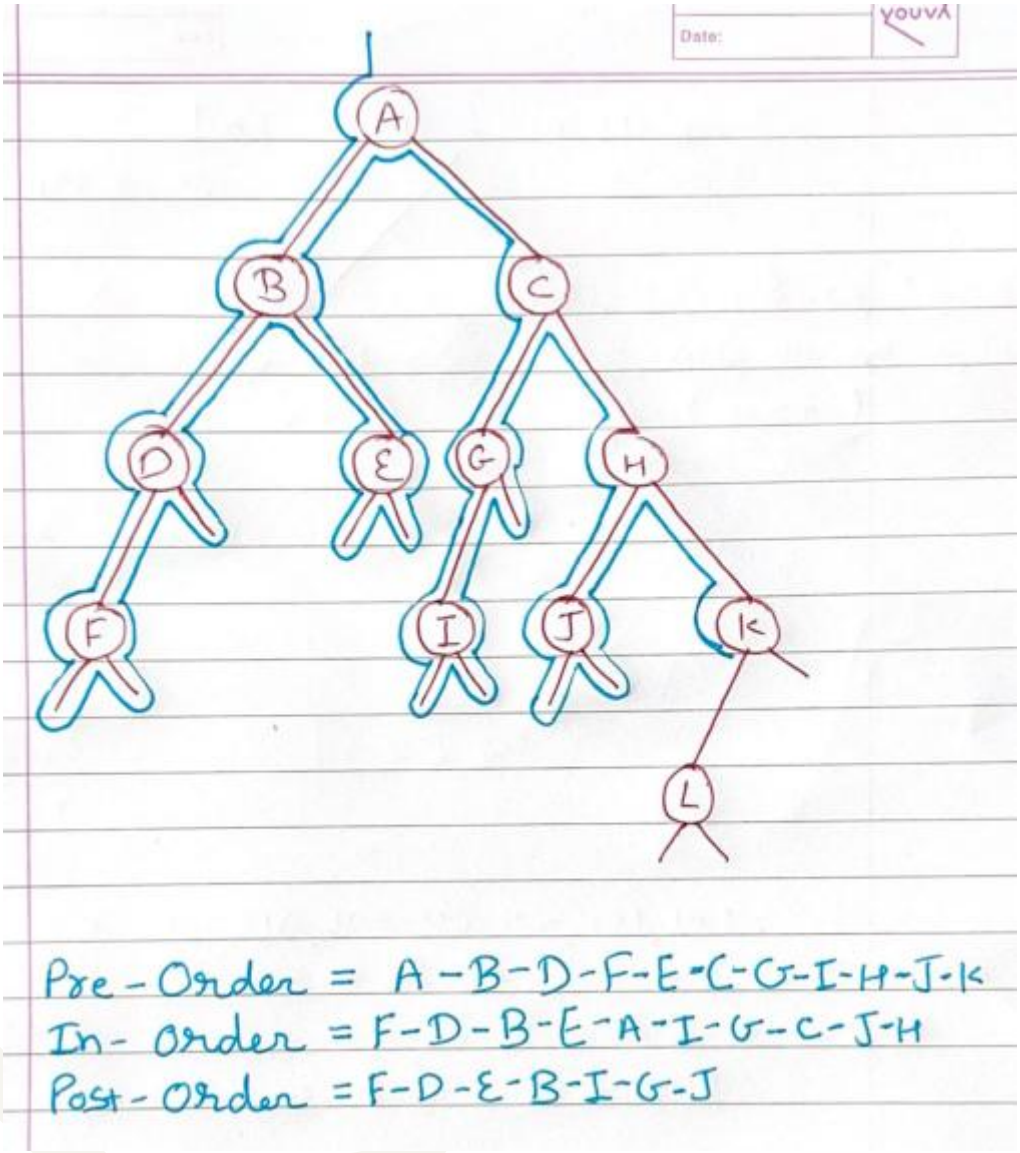
Step 25– Now, going ahead we met J, for the 3rd time. Hence, we will write it in Post-Order.



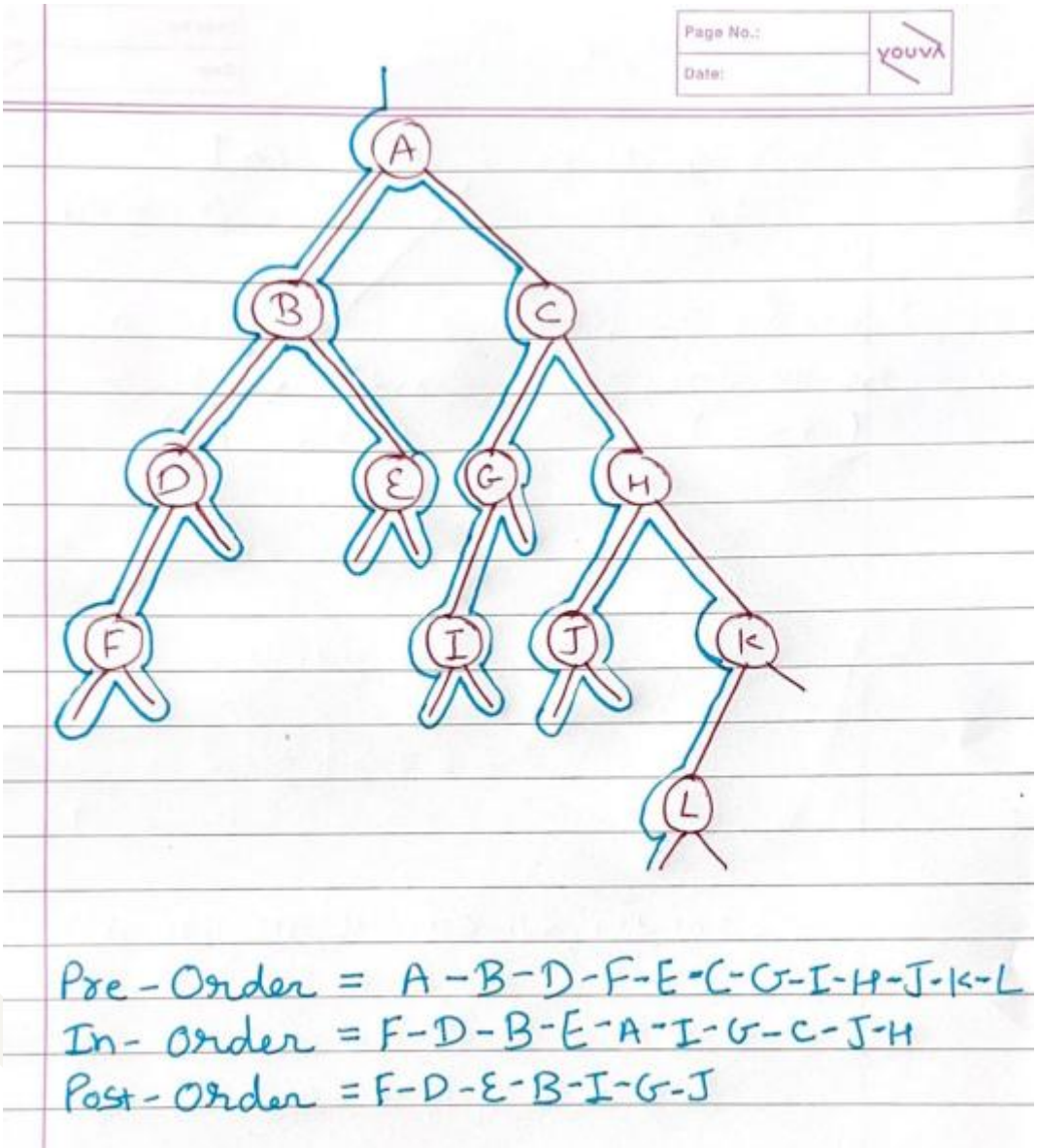
Step 26– Now, going ahead we met H, for the 2nd time. Hence, we will write it in In-Order.



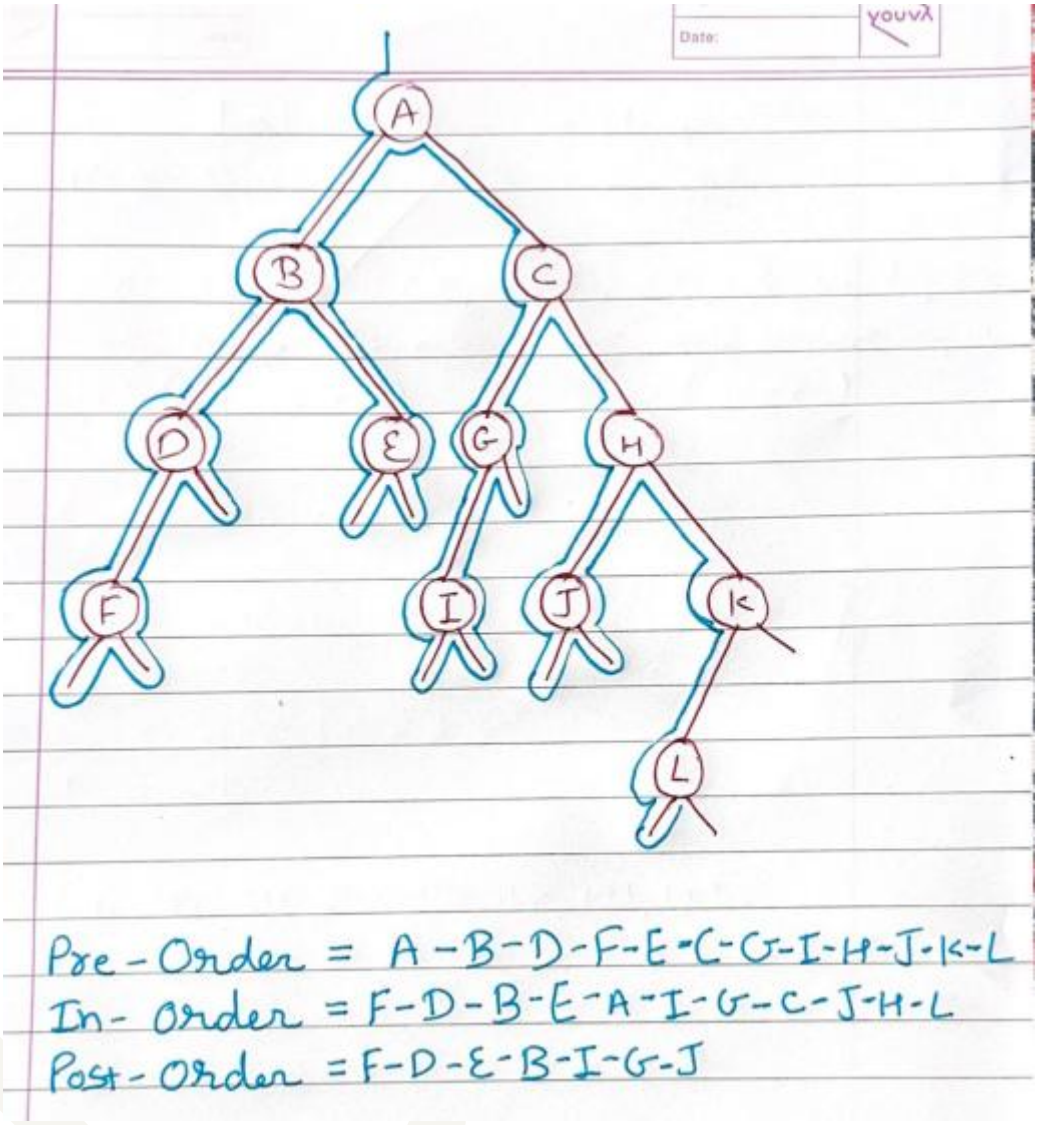
Step 27– Now, going ahead we met K, for the 1st time. Hence, we will write it in Pre-Order.



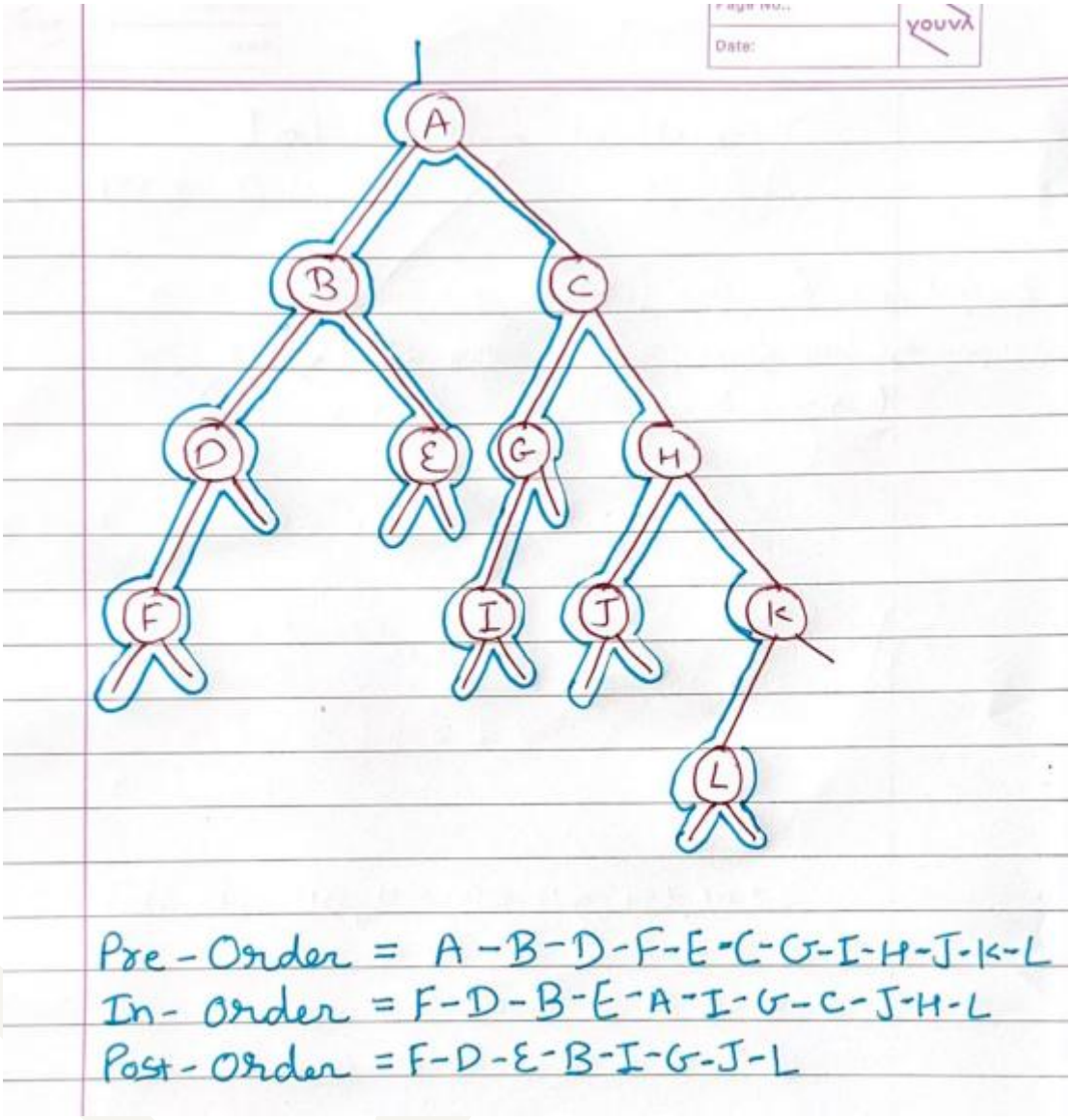
Step 28– Now, going ahead we met L, for the 1st time. Hence, we will write it in Pre-Order.



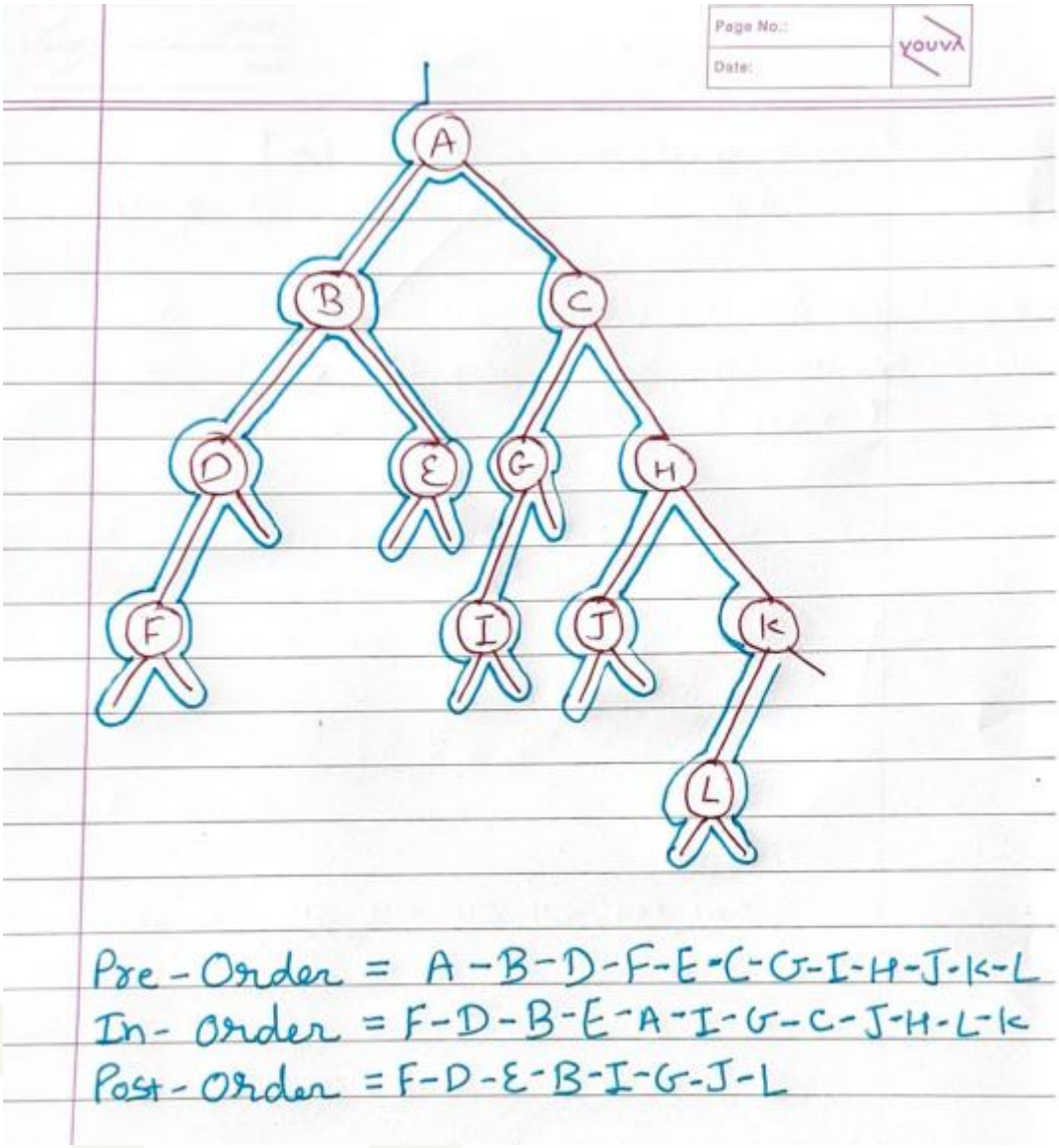
Step 29– Now, going ahead we met L, for the 2nd time. Hence, we will write it in In-Order.



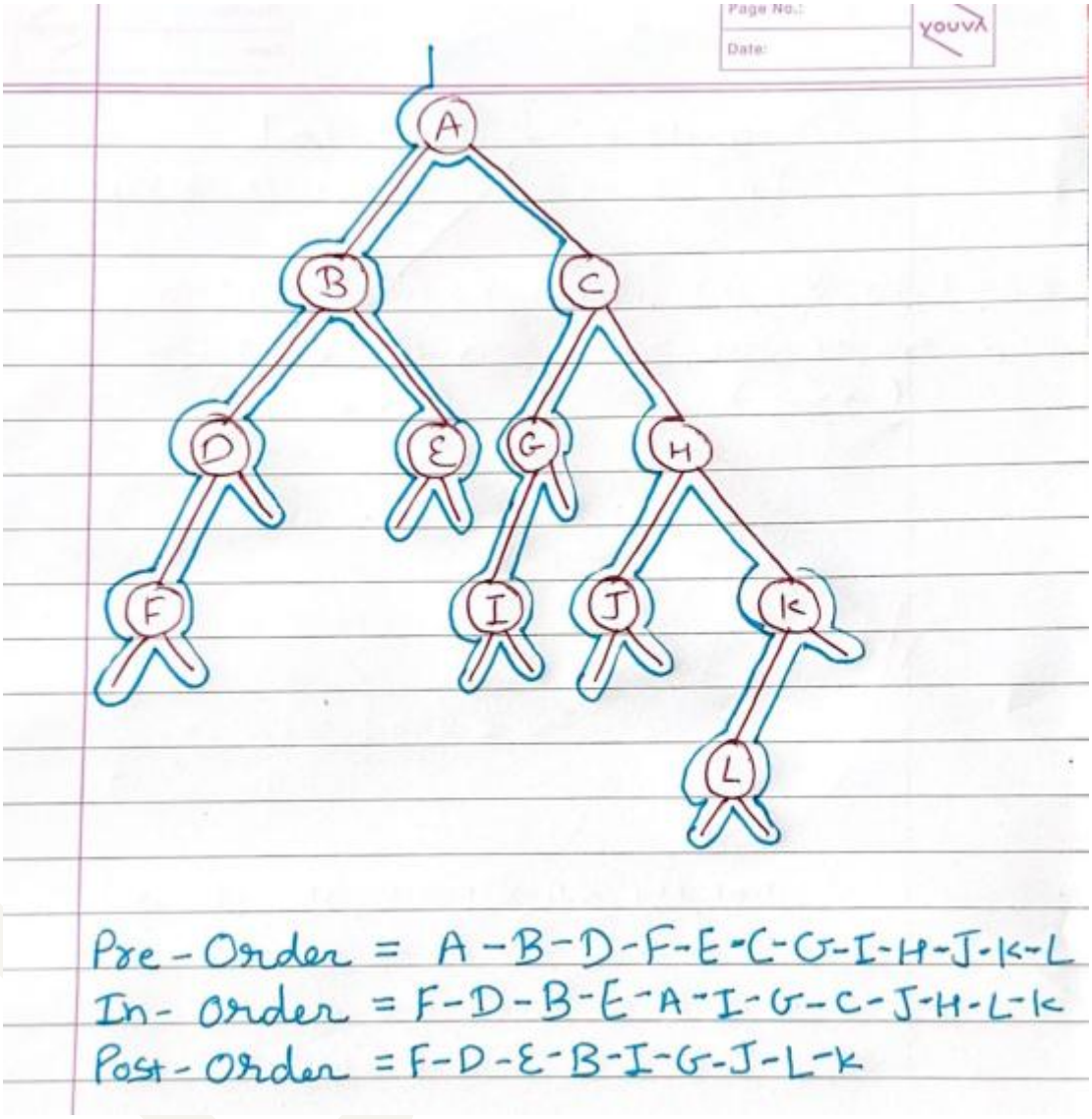
Step 30– Now, going ahead we met L, for the 3rd time. Hence, we will write it in Post-Order.



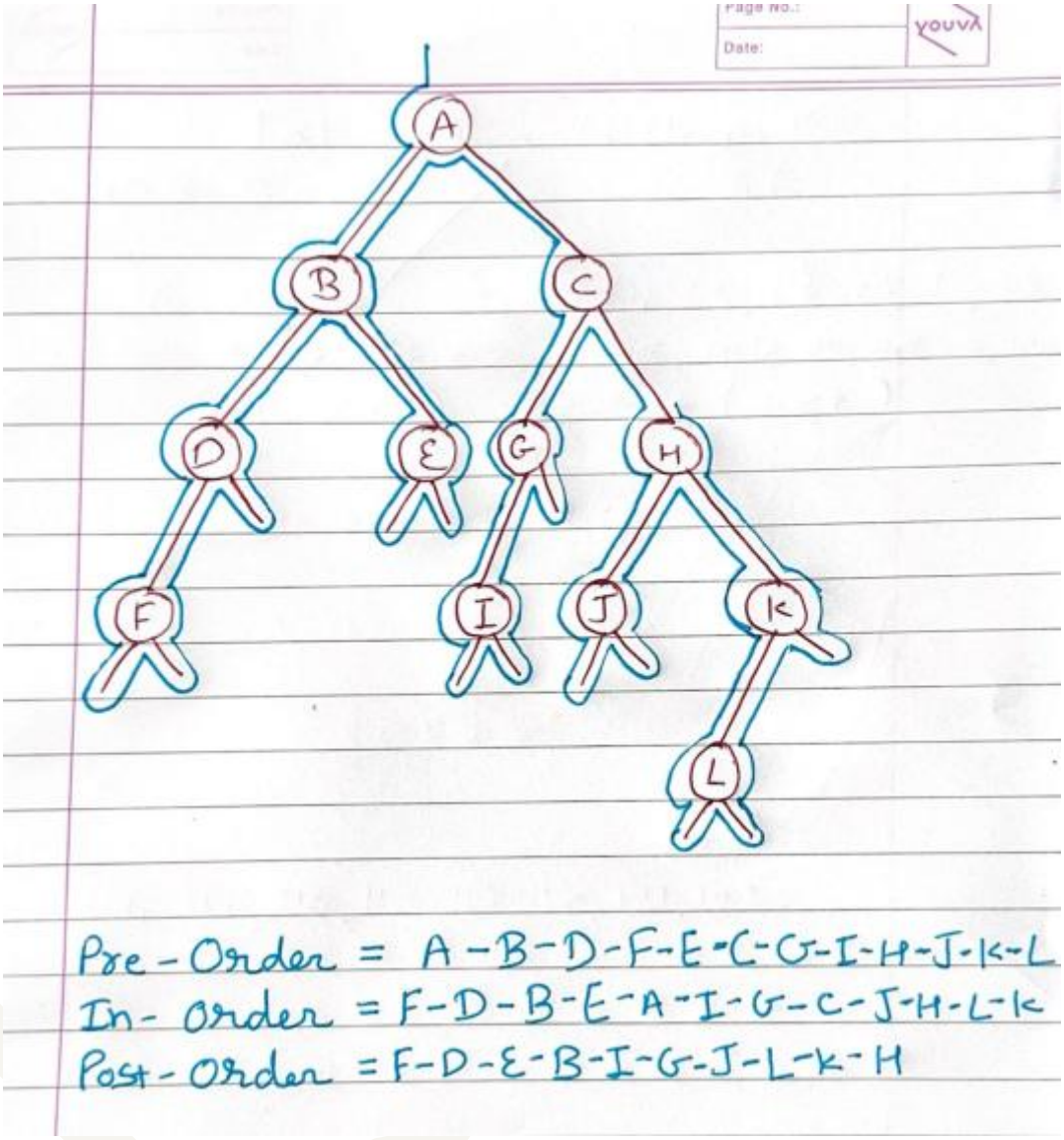
Step 31– Now, going ahead we met K, for the 2nd time. Hence, we will write it in In-Order.



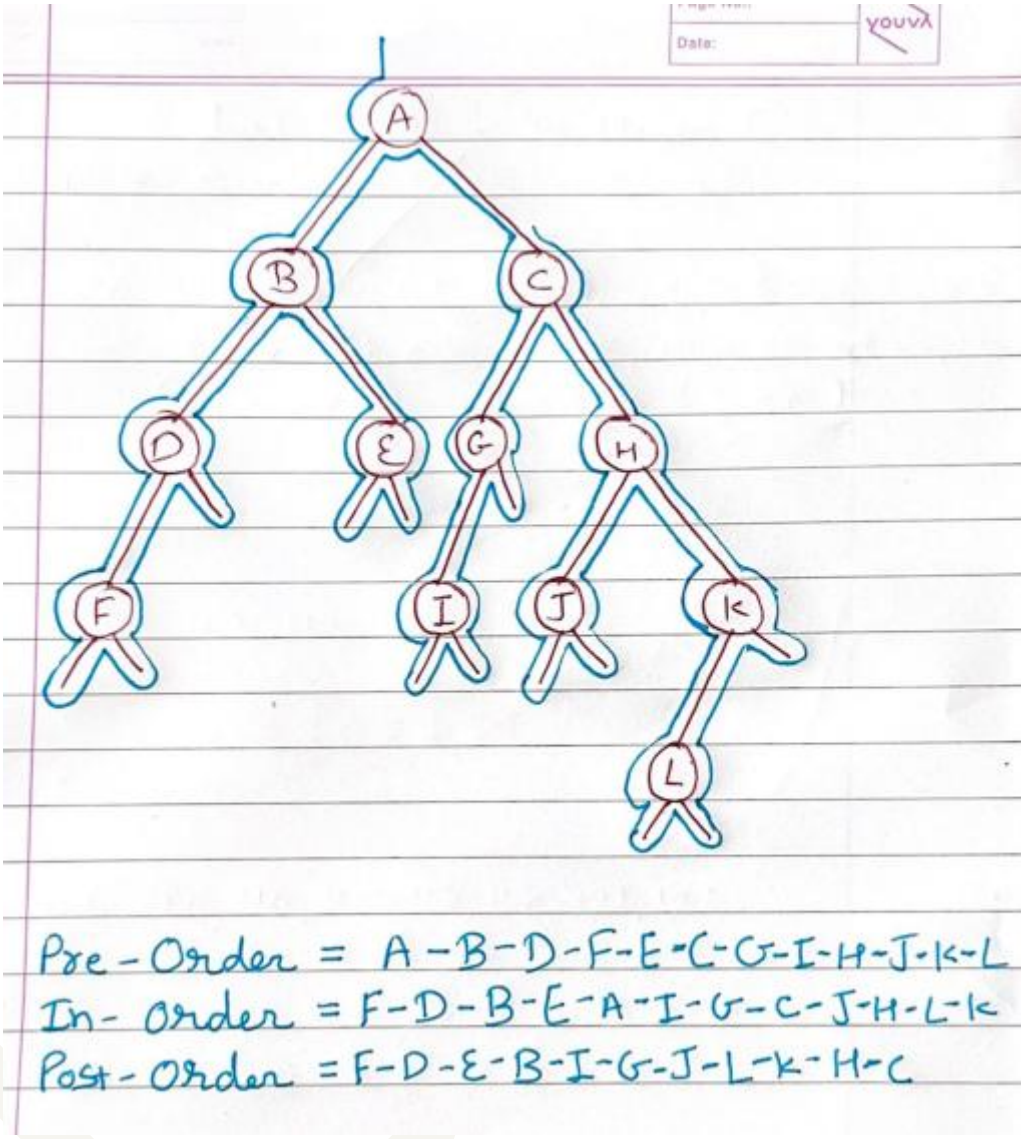
Step 32– Now, going ahead we met K, for the 3rd time. Hence, we will write it in Post-Order.



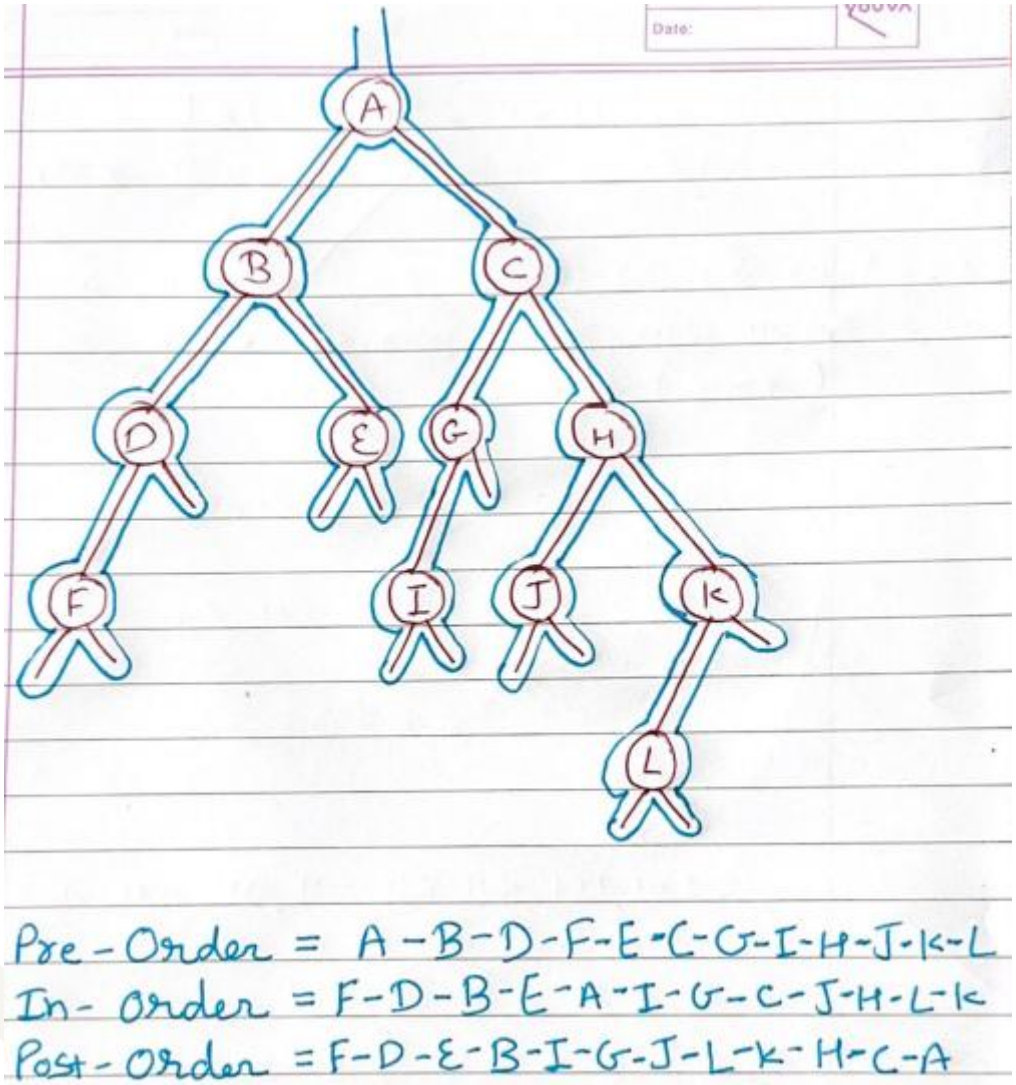
Step 33– Now, going ahead we met H, for the 3rd time. Hence, we will write it in Post-Order.



Step 34– Now, going ahead we met C, for the 3rd time. Hence, we will write it in Post-Order.



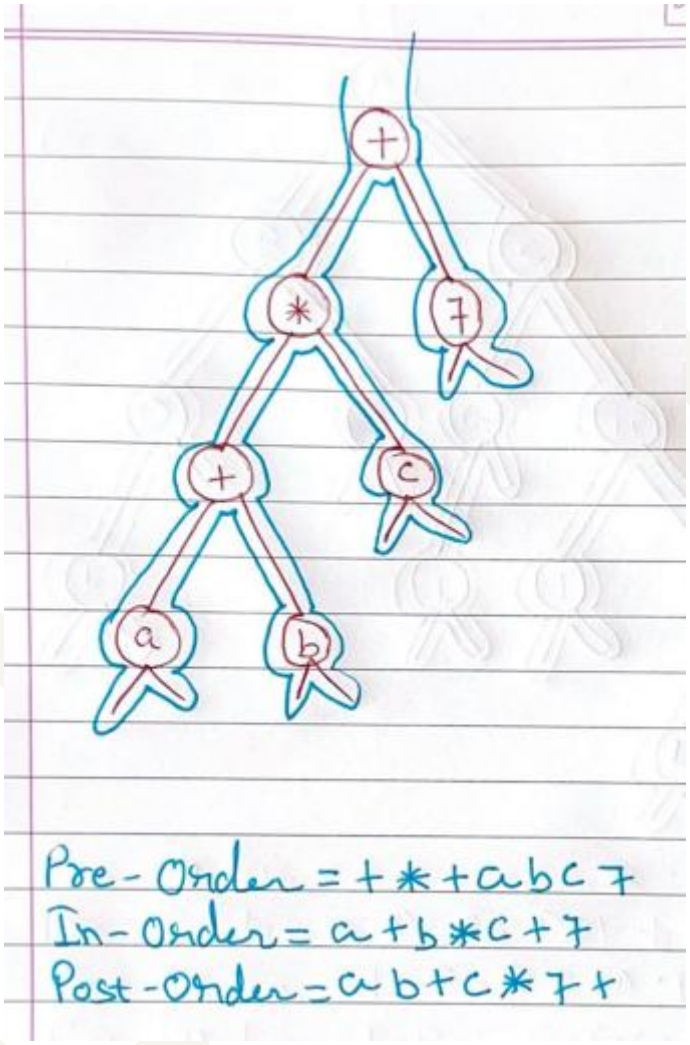
Step 35– Now, going ahead we met A, for the 3rd time. Hence, we will write it in Post-Order.



From now onwards we will directly write the traversal.

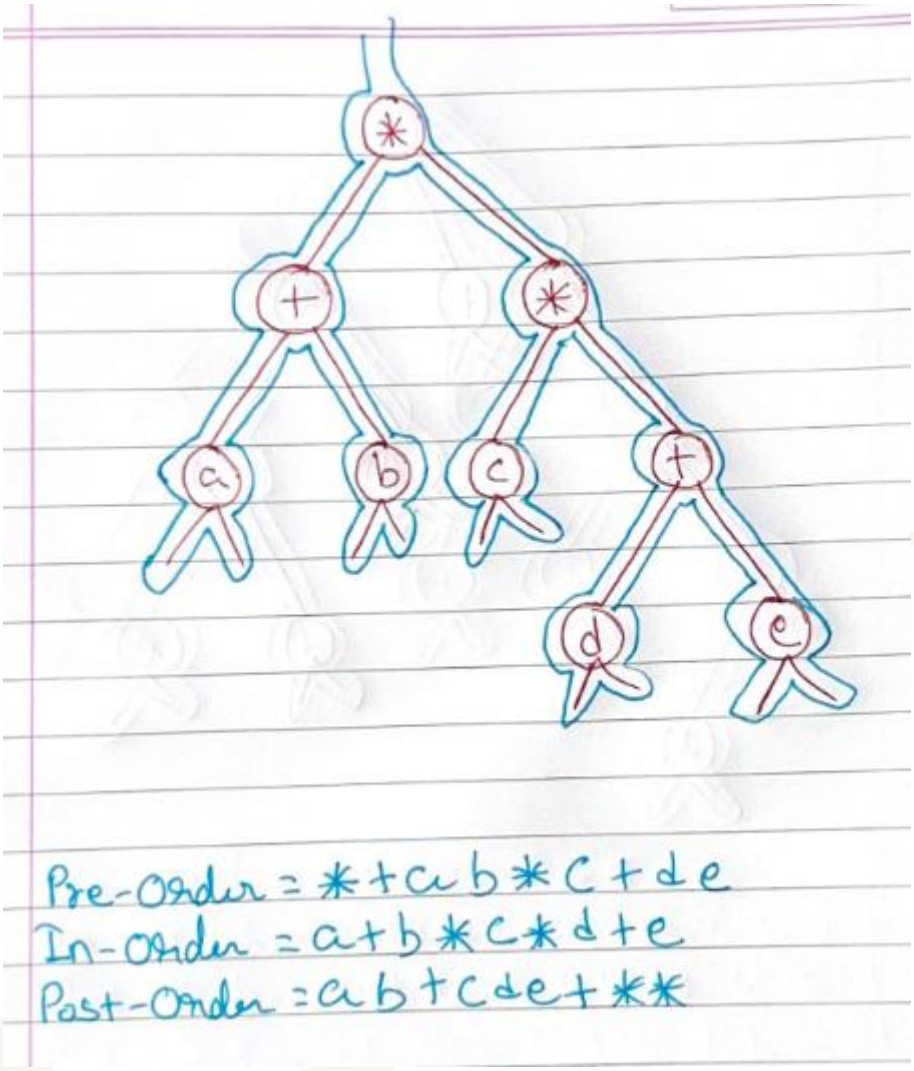
Q.2(Q.B 104) Find the Pre-Order, In-Order & Post-Order Traversal of the given tree.

Solution:



Q.3(Q.B 114) Find the Pre-Order, In-Order & Post-Order Traversal of the given tree.

Solution:



13. Numerical Based on Construction of Tree When Traversals are given

Let us understand this process step by step with an example.

Q.1(Q.B 59) Construct a Tree using given Traversals. (Show all steps)

The preorder traversal = {7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10}

Inorder traversal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Solution:

To construct a Binary Tree using Pre-Order & In-Order, the very first step is to identify the Root of the Tree.

Remember the definition of Pre-Order Traversal = Root – Left Child – Right Child.

Hence, the 1st value from the given Pre-Order Traversal will be the Root of the binary tree. Here in our case, it is 7.

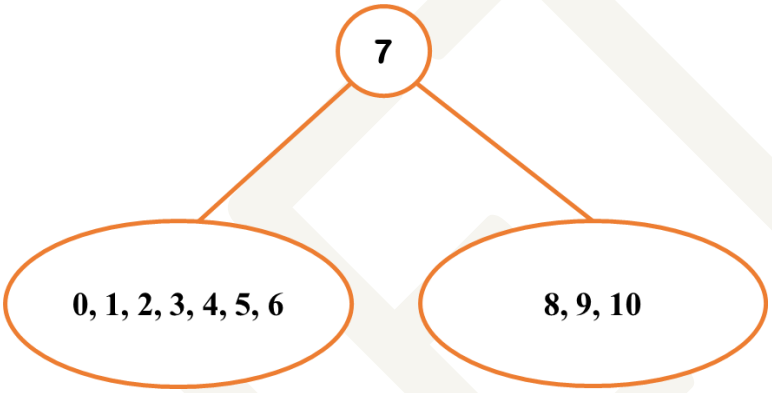
Step 1 – Root of the Binary Tree (1st value of the Pre-Order Traversal) = 7.

The preorder traversal = {7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10}

Inorder traversal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Now, highlight the same value in In-Order Traversal. As you can see above, when I highlighted 7 in In-Order traversal (0,1,2,3,4,5,6) are on the left side of tree and (8,9,10) is on the right side of the tree.

And this will hold true when we construct the tree.

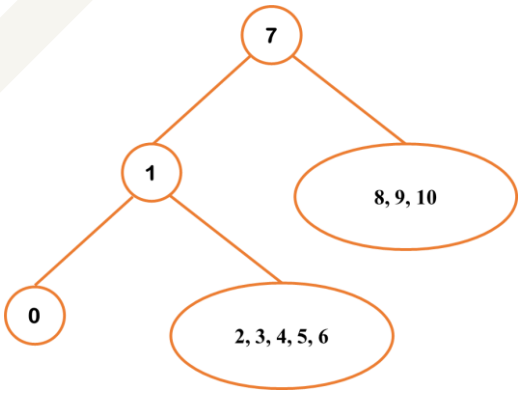


Step 2 – Now, again see the Pre-Order Traversal to get the next root. Here in our case, it is 1.

The preorder traversal = {7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10}

Inorder traversal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

As you can see in step 1, the value 1 is already on the left of the Node 7. Hence (0) will be on the left of 1 and (2, 3, 4, 5, 6) will be on the right of 1.

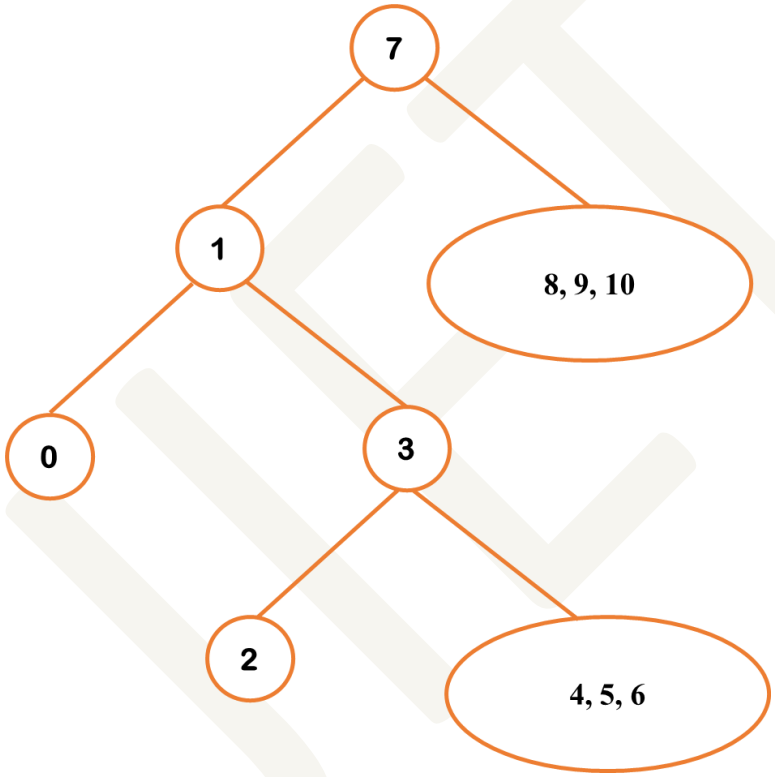


Step 3 – Next root from Pre-Order is 0. Which we already have shown on the left of 1. Hence, we will be seeing next value from Pre-Order. That value is 3.

Hence making root value 3 which is on right side of 1. Therefore (2) will be on left side and (4,5,6) will be on the right side.

The preorder traversal = {7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10}

Inorder traversal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

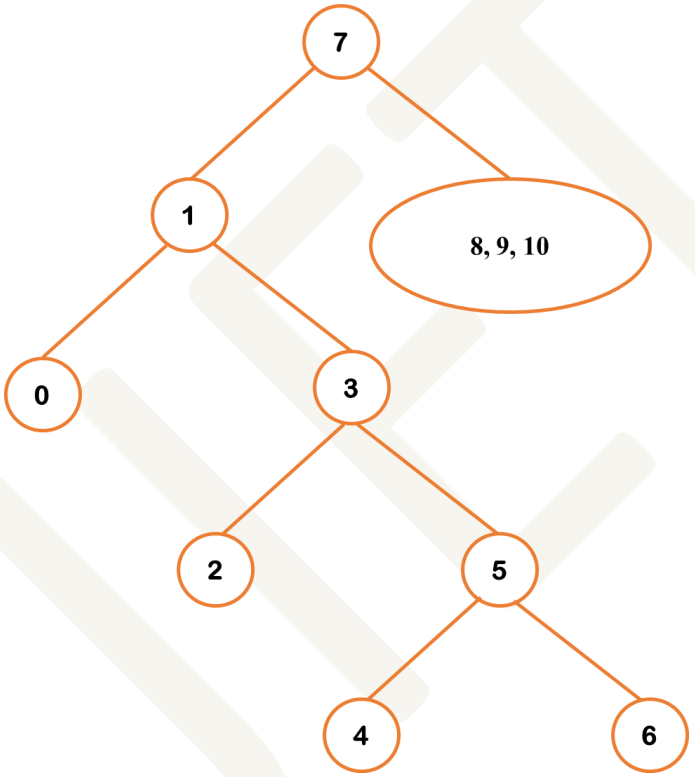


Step 4 – Next root from Pre-Order is 2. Which we already have shown on the left of 3. Hence, we will be seeing next value from Pre-Order. That value is 5.

Hence making root value 5 which is on left side of 3. Therefore (4) will be on left side and (6) will be on the right side.

The preorder traversal = {7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10}

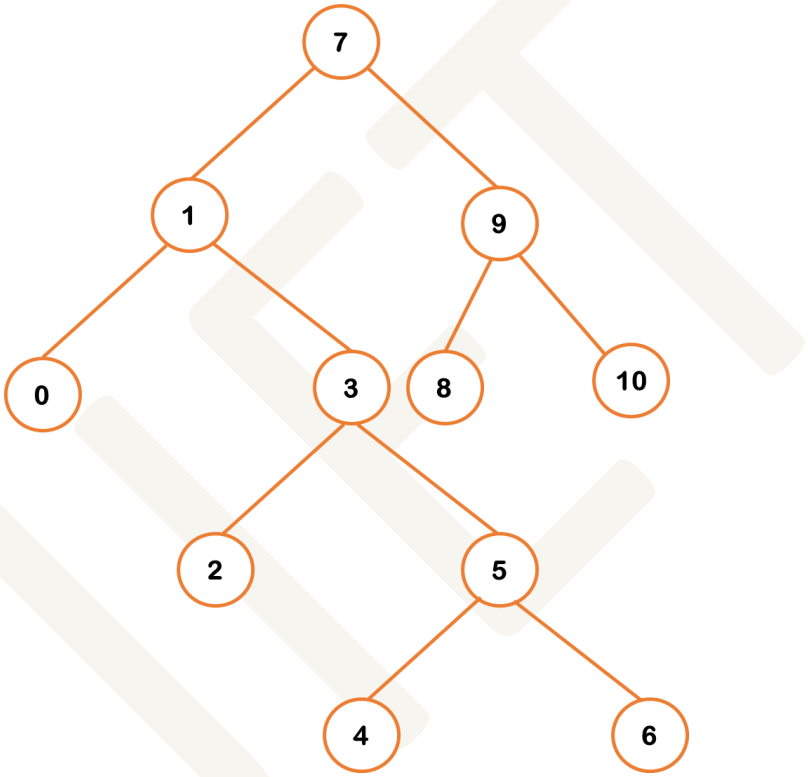
Inorder traversal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}



Step 5 – Next root from Pre-Order is 4. Which we already have shown on the left of 5. Hence, we will be seeing next value from Pre-Order. That value is 6. That is also shown on the right side of 5. Hence moving on to the next value from Pre-Order which is 9.

The preorder traversal = {7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10}

Inorder traversal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}



Now, in the next step the root value is 8 from the Pre-Order. Which is already on the left of 9. And then the last value is 10 which is already on the right of 9.

So, figure in step 5 is the final binary tree from the given traversal.

Q.1(Q.B 74) Construct binary tree from given post-order, in-order pair. (Show all steps)

Post order: I-D-B-G-C-H-F-E-A

Inorder: B-I-D-A-C-G-E-H-F

Solution:

To construct a Binary Tree using Post-Order & In-Order, the very first step is to identify the Root of the Tree.

Remember the definition of Post-Order Traversal = Left Child – Right Child - Root.

Hence, the Last value from the given Post-Order Traversal will be the Root of the binary tree. Here in our case, it is A.

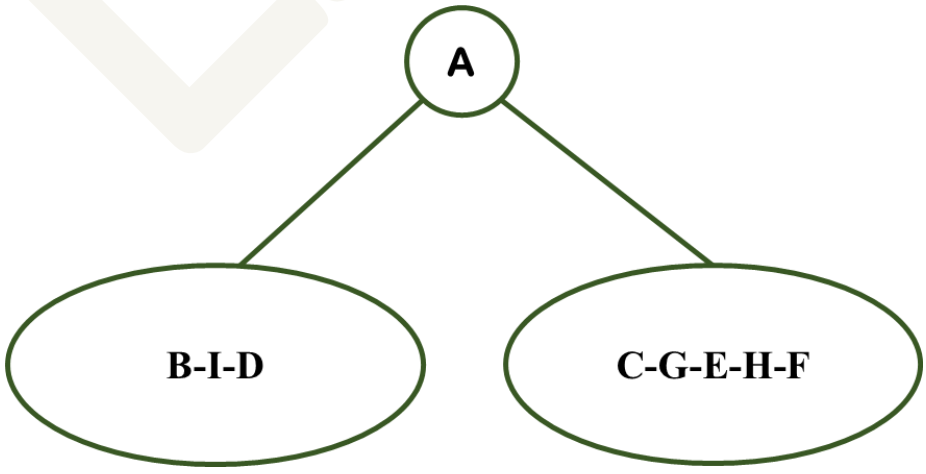
Step 1 – Root of the Binary Tree (Last value of the Pre-Order Traversal) = A.

Post order: I-D-B-G-C-H-F-E-A

Inorder: B-I-D-A-C-G-E-H-F

Now, highlight the same value in In-Order Traversal. As you can see above, when I highlighted A in In-Order traversal (B-I-D) are on the left side of tree and (C-G-E-H-F) is on the right side of the tree.

And this will hold true when we construct the tree.

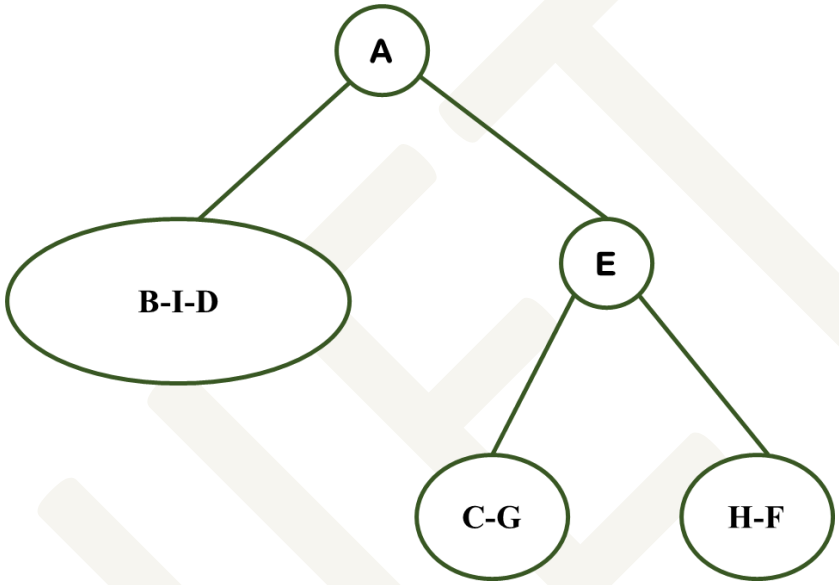


Step 2 – Now, again see the Post-Order Traversal to get the next root. Here in our case, it is E.

Post order: I-D-B-G-C-H-F-E-A

Inorder: B-I-D-A-C-G-E-H-F

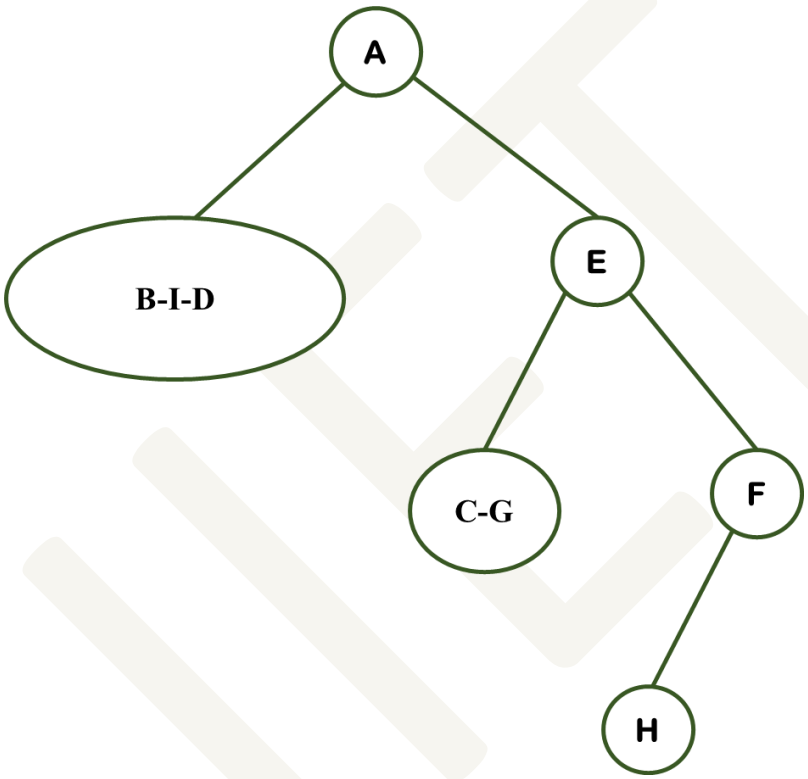
As you can see in step 1, the value E is already on the right of the Node A. Hence (C-G) will be on the left of E and (H-F) will be on the right of A.



Step 3 – Now, again see the Post-Order Traversal to get the next root. Here in our case, it is F. & also you see the Tree in step 2. H is on the left of F.

Post order: I-D-B-G-C-H-F-E-A

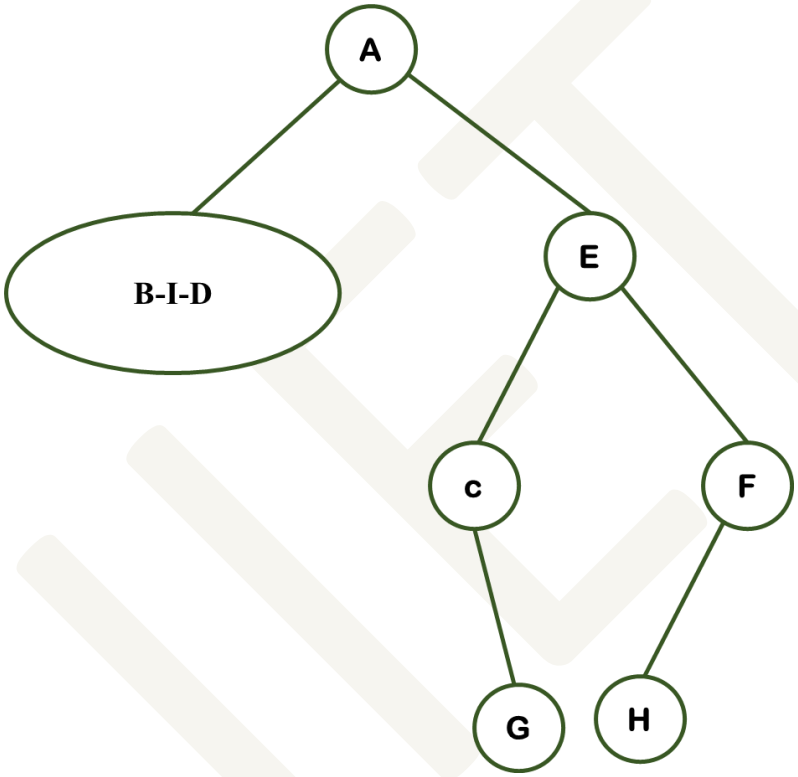
Inorder: B-I-D-A-C-G-E-H-F



Step 4 – Now, again see the Post-Order Traversal to get the next root. Here in our case, it is C. & also you see the Tree in step 3. G is on the right of C.

Post order: I-D-B-G-C-H-F-E-A

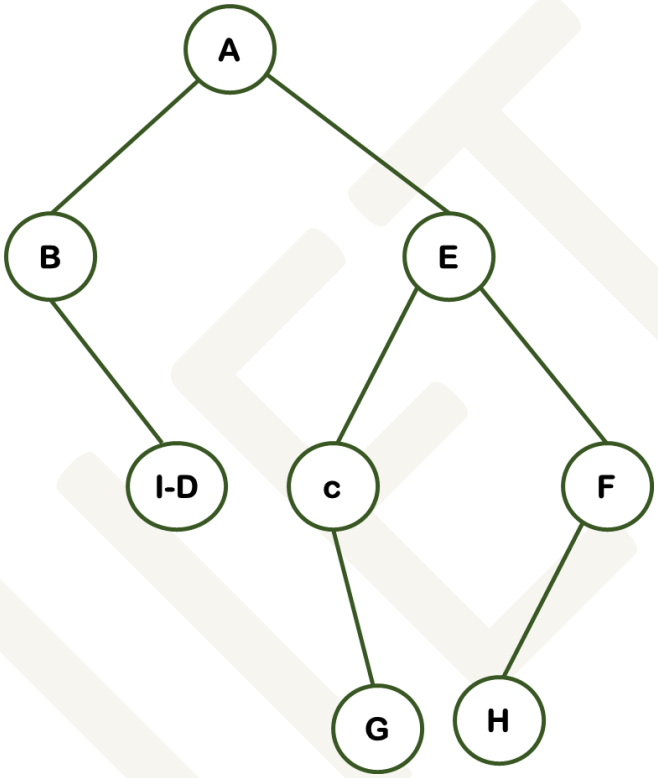
Inorder: B-I-D-A-C-G-E-H-F



Step 5 – Now, again see the Post-Order Traversal to get the next root. Here in our case, it is B. So, when we will make B the root, I-D both will be on the right of B as seen in In-Order.

Post order: I-D-B-G-C-H-F-E-A

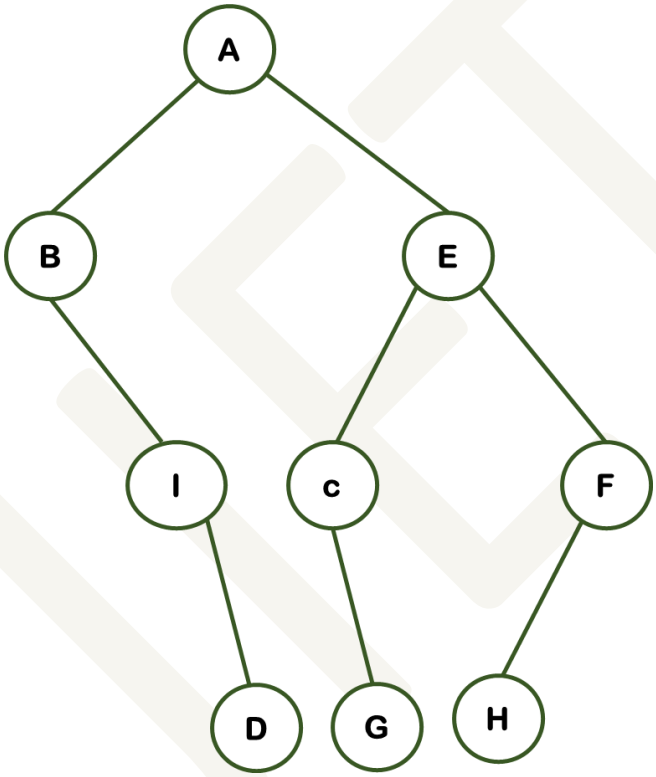
Inorder: B-I-D-A-C-G-E-H-F



Step 6 – Now, again see the Post-Order Traversal to get the next root. Here in our case, it is D. So, when we will make D the root, I both will be on the right of D as seen in In-Order.

Post order: I-D-B-G-C-H-F-E-A

Inorder: B-I-D-A-C-G-E-H-F



This is our Final Binary Tree from the given Traversals.

14. Expression Tree

An expression binary tree, also known as an expression tree or parse tree, is a type of binary tree used to represent expressions in a way that facilitates their evaluation.

It is commonly used in computer science and programming to represent mathematical and logical expressions.

In an expression binary tree, each node represents an operator or an operand of the expression. Operators (such as +, -, *, /) are represented by internal nodes, while operands (numeric values or variables) are represented by the leaf nodes.

The tree's structure reflects the hierarchical nature of the expression, with the operators at higher levels and operands at lower levels.

a) Application of Expression Trees

Expression Evaluation: Expression trees provide a natural and efficient way to evaluate mathematical and logical expressions. By understanding how to construct and traverse expression trees, you can evaluate complex expressions in a structured and systematic manner.

Parsing and Compilation: In programming languages and compilers, expressions need to be parsed and converted into machine-readable code. Expression trees play a crucial role in this parsing process, allowing compilers to understand the hierarchical structure of expressions and generate optimized machine code.

Optimization: Expression trees can be used to optimize expressions by identifying common subexpressions and eliminating redundant calculations. This is particularly relevant in algebraic simplification and query optimization in databases.

Algorithm Design: Understanding expression trees helps in designing algorithms that involve mathematical or logical expressions. For example, in scientific computing and symbolic computation, expression trees are used to solve complex equations and perform algebraic manipulations.

Interpreter Development: Interpreters for programming languages use expression trees to interpret and execute code efficiently. Expression trees allow interpreters to understand the order of operations and apply them correctly during execution.

Mathematical Modeling: In mathematical modeling and problem-solving, expression trees can represent complex relationships between variables, making it easier to analyze and solve problems.

Debugging: Expression trees can be instrumental in debugging programs. By visualizing the expression tree, programmers can better understand how expressions are being evaluated and identify any potential issues.

Learning Other Data Structures: Expression trees are a type of binary tree, and learning about them serves as a foundation for understanding other tree-based data structures and algorithms like binary search trees, AVL trees, and Red-Black trees.

Interviews and Assessments: Expression trees and related concepts are often included in technical interviews and coding assessments for software engineering positions. Familiarity with expression trees can be beneficial for excelling in such evaluations.

Overall, learning about expression trees expands your understanding of data structures, programming languages, and algorithm design. It equips you with valuable skills for problem-solving, software development, and various other areas in computer science and engineering.

15. Construction of Expression Tree

To Construct an Expression Tree from the given data, follow these simple rules.

- Scan the equation from Left to Right
- Check the priority level of Operators. (If the same operand is appearing multiple times, then apply associativity rule)
- Provide numbers to Operators as per their priority. (Hence the operator with Highest Priority will have 1 and the Lowest Priority will have lower value)
- The Operator with Lowest Priority will be the root of Binary Tree.

To understand the above scenario, let us take an example and apply the rules.

Q.1(Q.B 76) Construct an Expression tree for a*b-c*d-e/f+g-h/i.

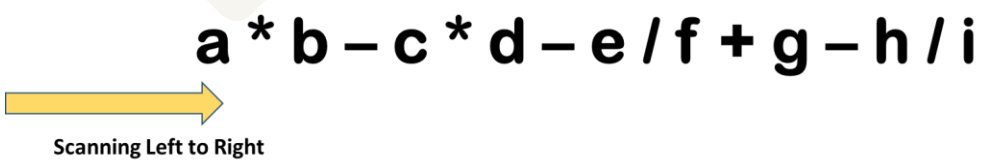
Solution:

Step 1 – Scan the equation left to right. Also, take a note that there are no parenthesis () in the given equation.

Now, If we recall the fundamentals of priority –

1. Power Operators: ^, $ ↑ have the highest priority and their associativity is Right to Left.
2. Multiply and Divide: * / have the next highest priority and their associativity is Left to Right.
3. Plus and Minus: + - have the least priority and their associativity is Left to Right.

Hence, Applying the same logic to our given equation --



There are No power operators hence we will check for * / operators and as we can see there are more than 1 * / in the given equation. Hence, we will follow Left to right associativity rule.

$$a \underset{1}{*} b - c \underset{2}{*} d - e \underset{3}{/} f + g - h \underset{4}{/} i$$

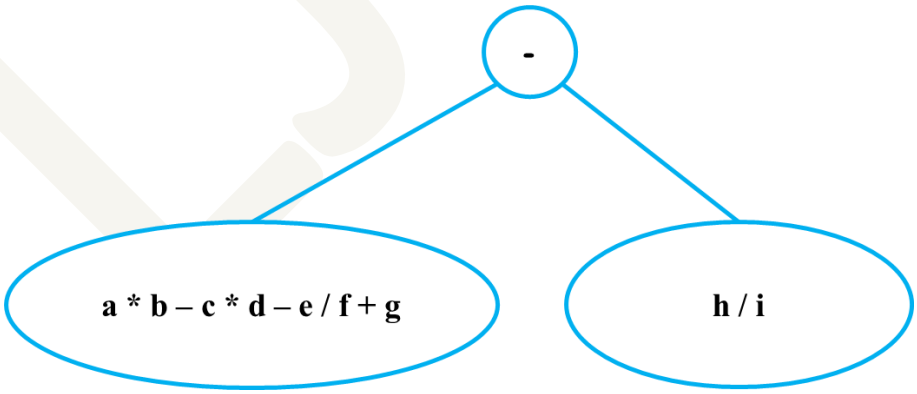
Now, there are only + and – operators left. Again, applying Left to Right associativity rule.

$$a \underset{1}{*} b \underset{5}{-} c \underset{2}{*} d \underset{6}{-} e \underset{3}{/} f \underset{7}{+} g \underset{8}{-} h \underset{4}{/} i$$

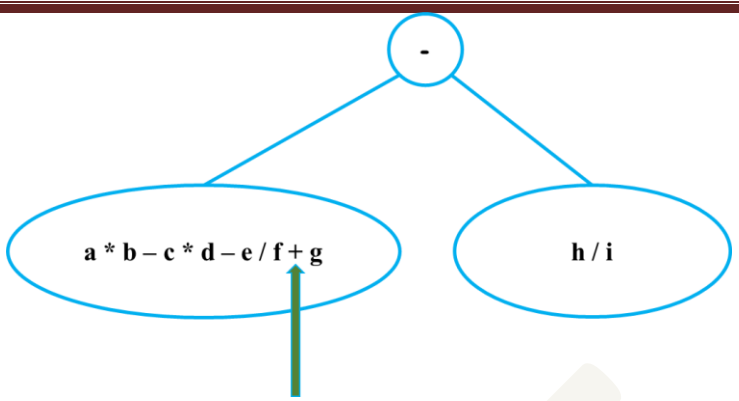
Now, we have numbered all the operators. Hence our Step 1 is completed.

Step 2 – Root of the Expression Tree = Highest number of operator in Step 1. (In our case it is - .

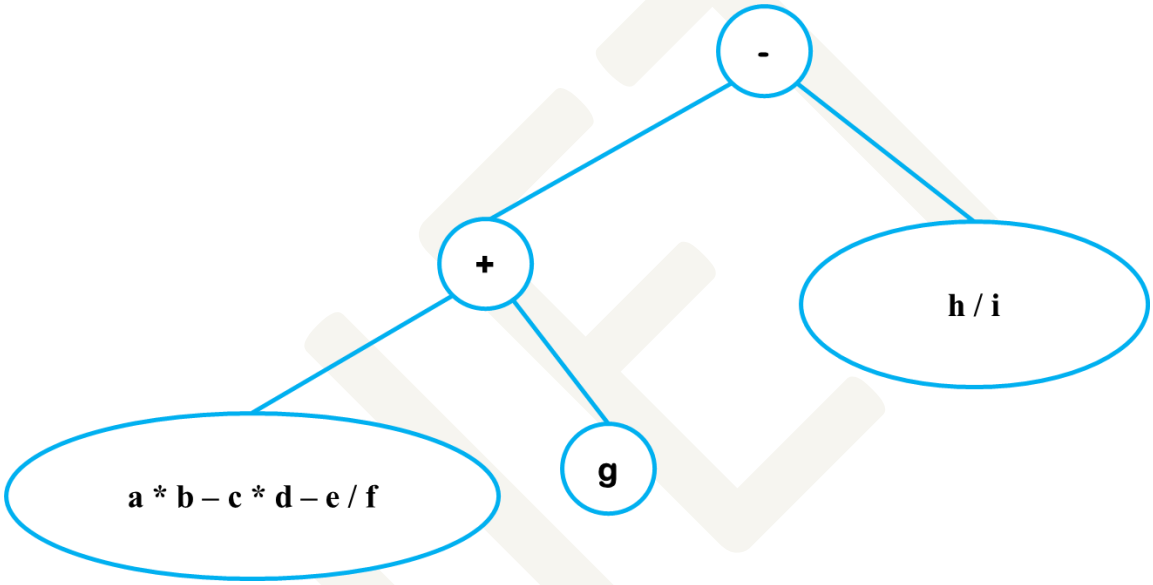
$$a \underset{1}{*} b \underset{5}{-} c \underset{2}{*} d \underset{6}{-} e \underset{3}{/} f \underset{7}{+} g \underset{8}{-} h \underset{4}{/} i$$



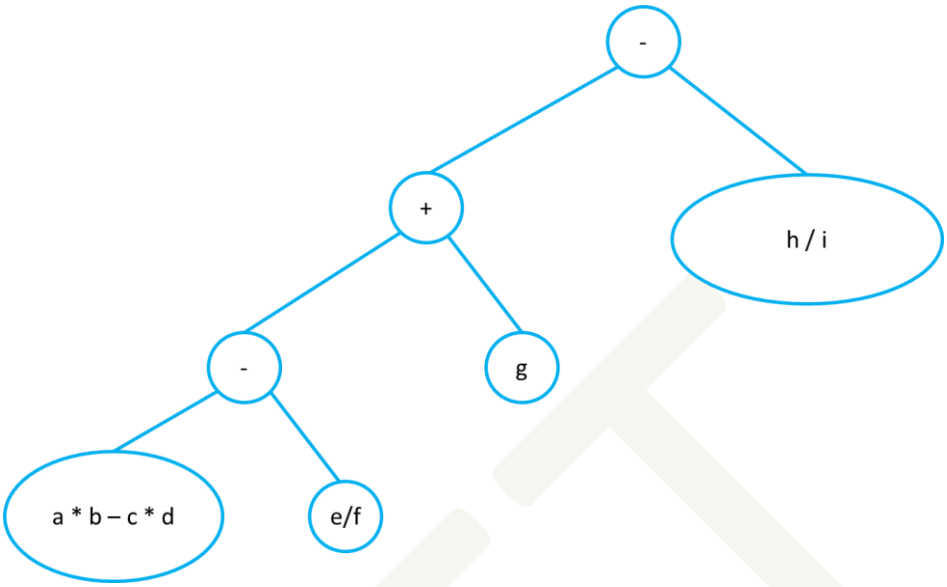
Step 3 – Now, Solving the next lowest operator number which is 7. And, the operator is +. (Refer numbering given in step 1)



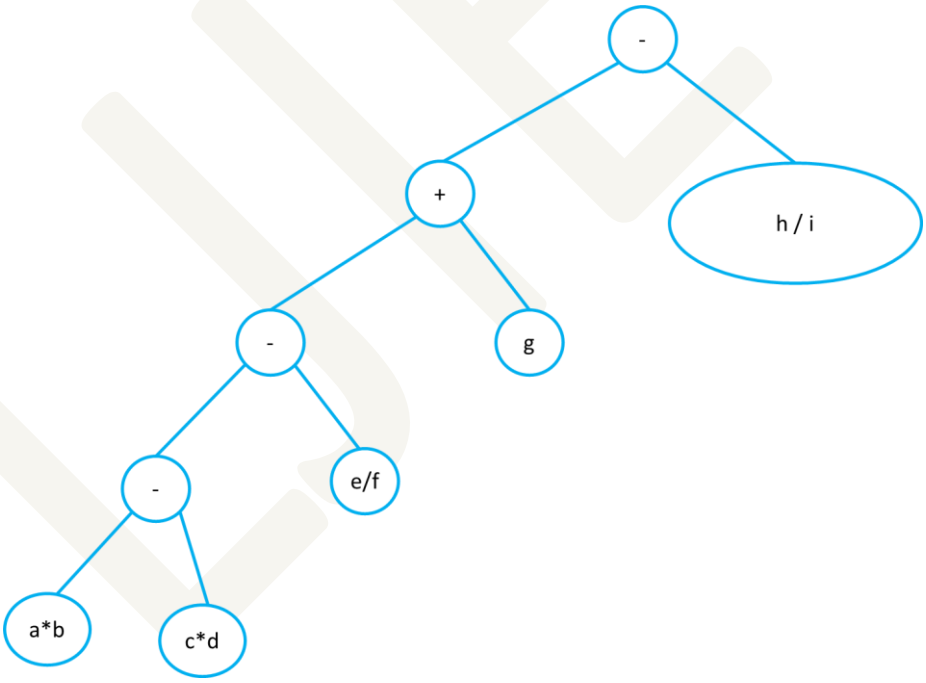
As you can see it is already on the left of the root node.



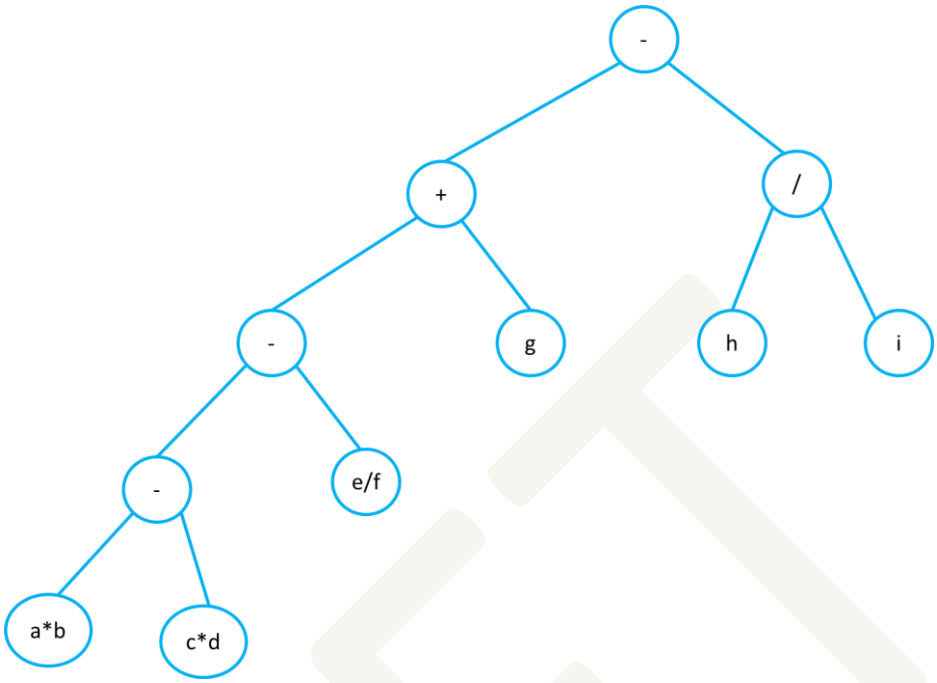
Step 3 – The next lowest number for the operators is 6 which is given to - .



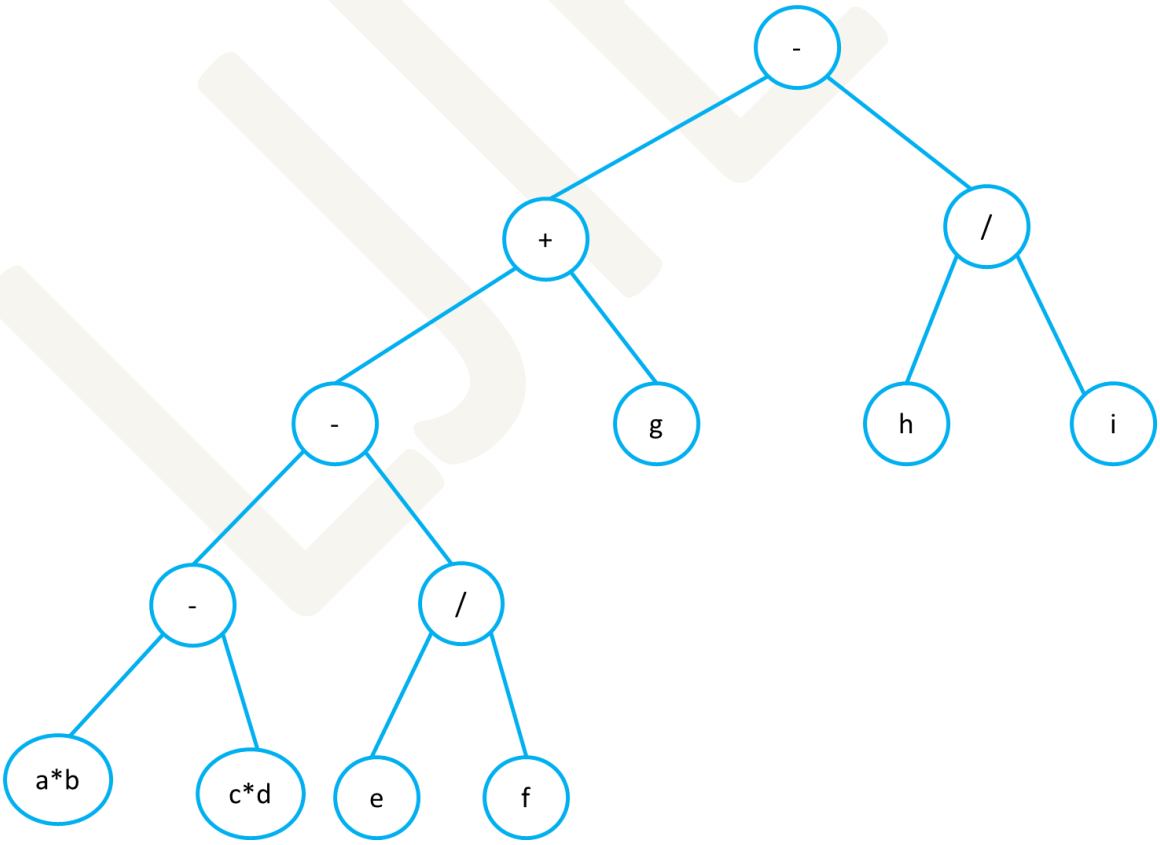
Step 4 – Solve for the next lowest number for the operator which is 5 and it is given to - .



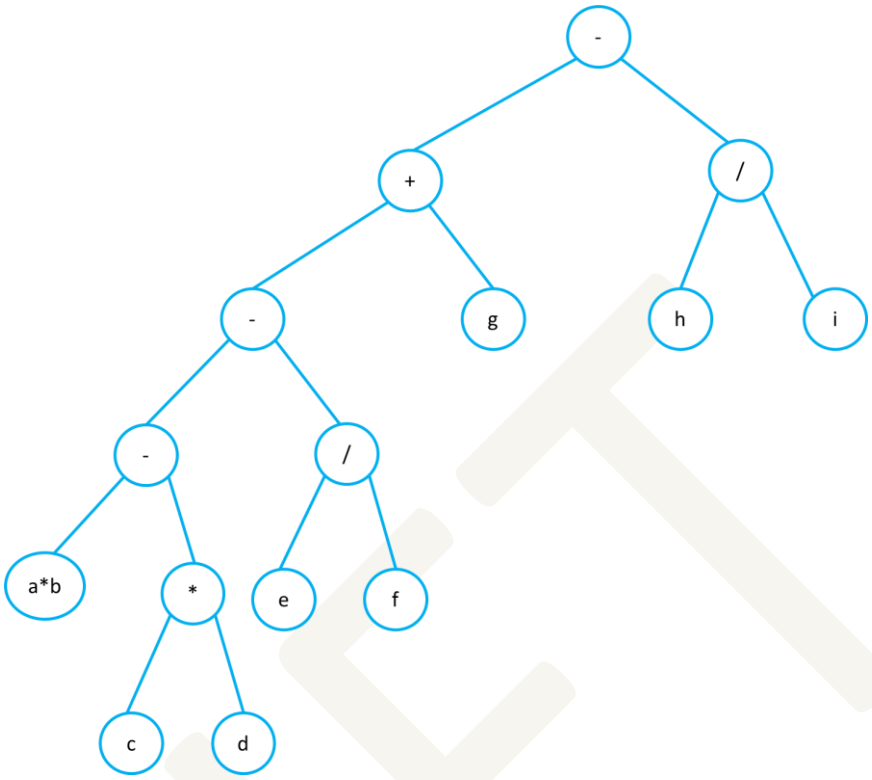
Step 5 – Again solving for number 4 which is given to / operator.



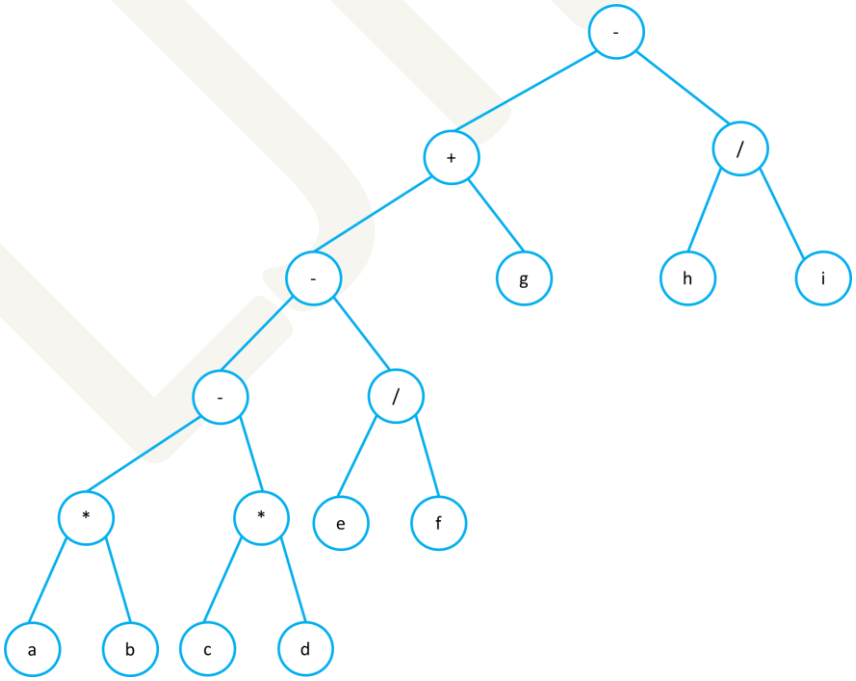
Step 6 – Again, solving for number 3 which is given to / operator.



Step 7 – Solving for operator number 2, which is given to *.



Step 8 – At Last, we will be solving for number 1, which is give to * and that will be our final tree.



Q.2(Q.B 82) Construct an Expression tree for a * (b + c) + (d * e) / f + g*h.

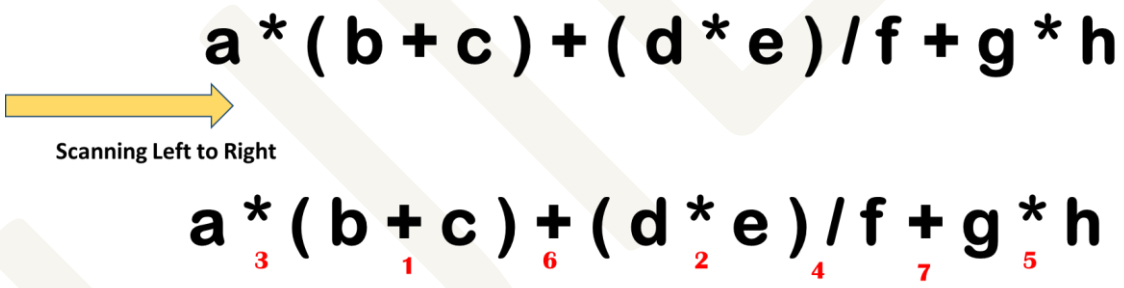
Solution:

Step 1 – Scan the equation left to right. Also, take a note that there is parenthesis () in the given equation.

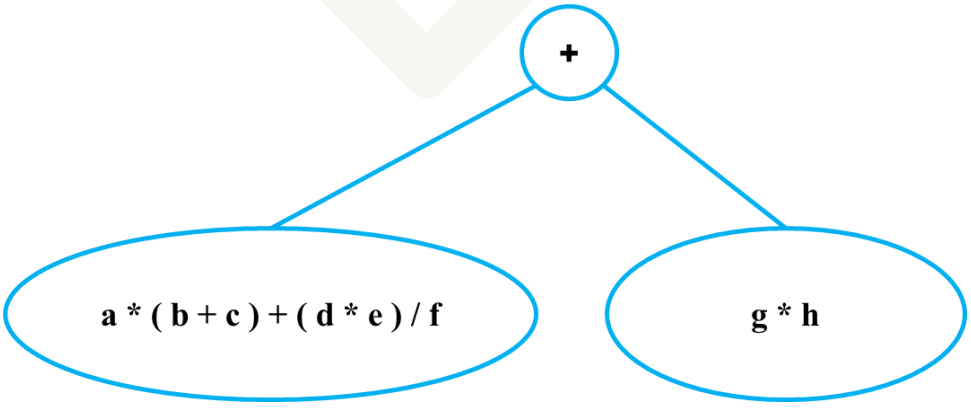
Now, If we recall the fundamentals of priority –

1. () are given the top priority with associativity Left to Right.
2. Power Operators: ^, $ ↑ have the highest priority and their associativity is Right to Left.
3. Multiply and Divide: * / have the next highest priority and their associativity is Left to Right.
4. Plus and Minus: + - have the least priority and their associativity is Left to Right.

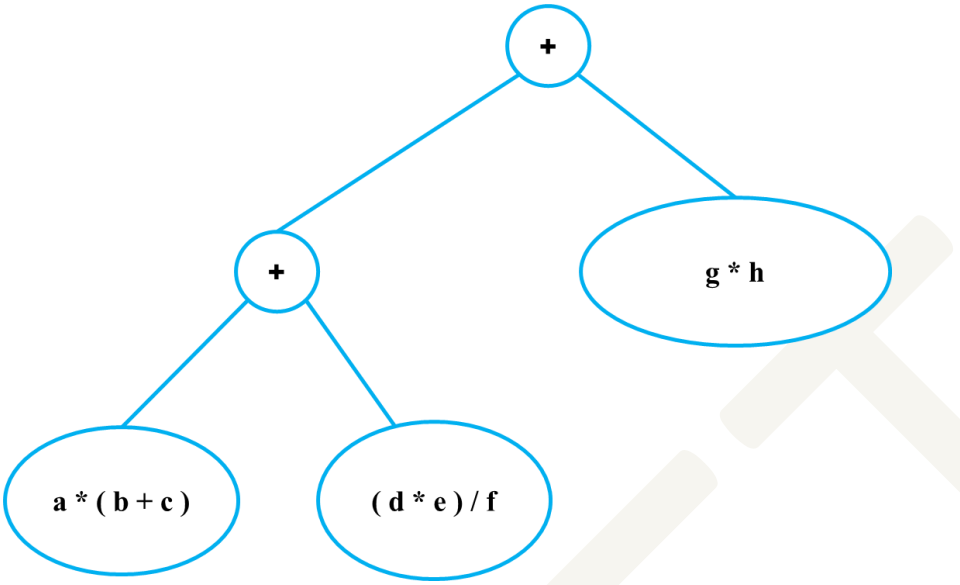
Hence, Applying the same logic to our given equation --



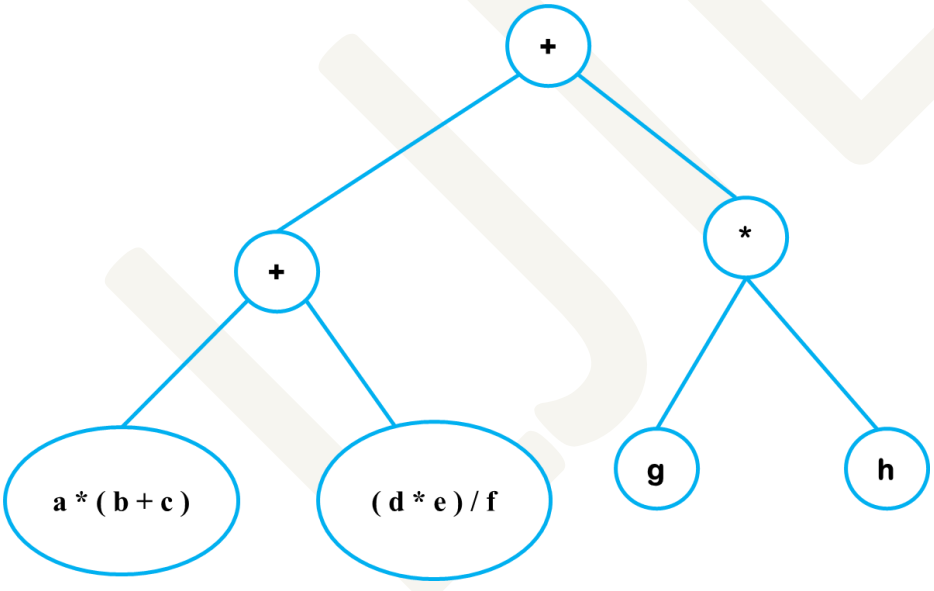
Step 2 – Root of the Expression Tree = Lowest Operator number. Here in our case, it is 7 (+).



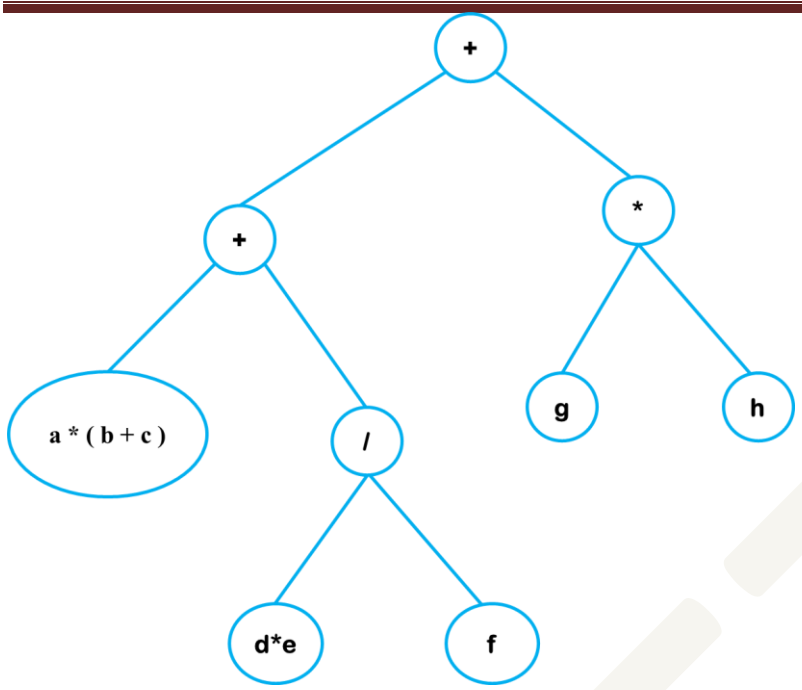
Step 3 – Now, Solving for the next lowest number of operator which is 6. In our case 6 is given to +. And that + is already on the left part of the tree.



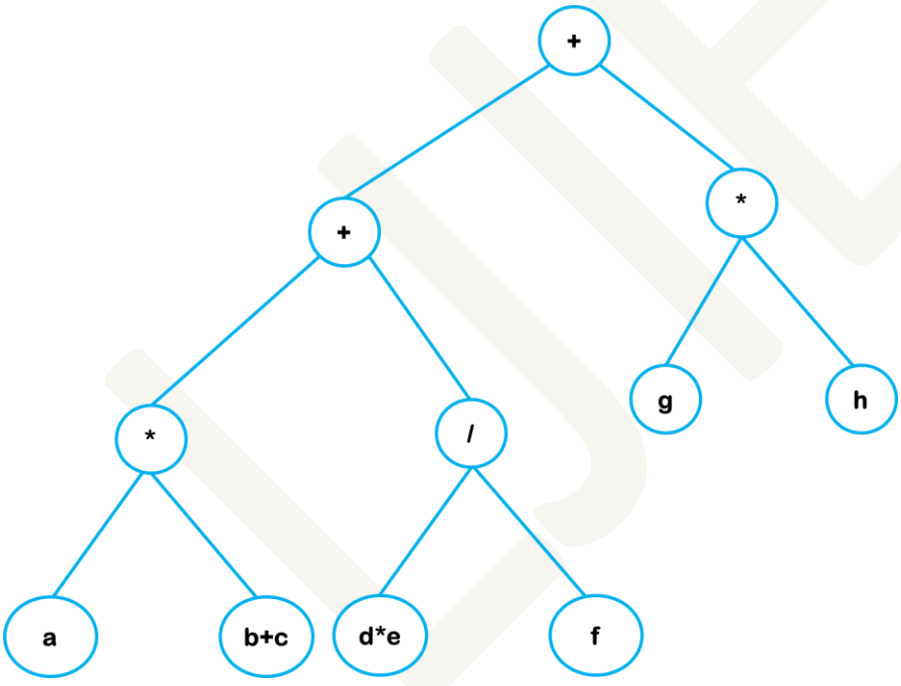
Step 3 – Now, again solving for the next lowest number of operator which is 5. In our case 5 is given to * which is on the right part of the tree.



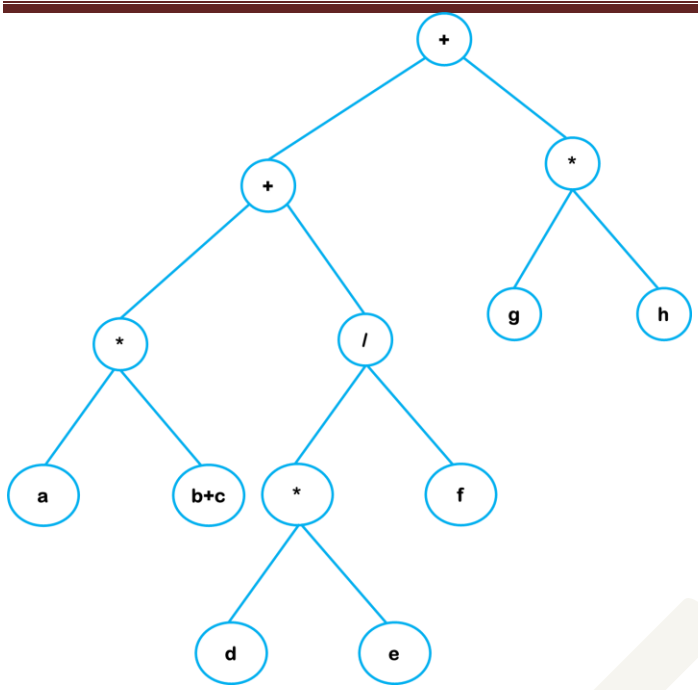
Step 4 – Now, again solving for the next lowest number of operator which is 4. In our case 4 is given to the /.



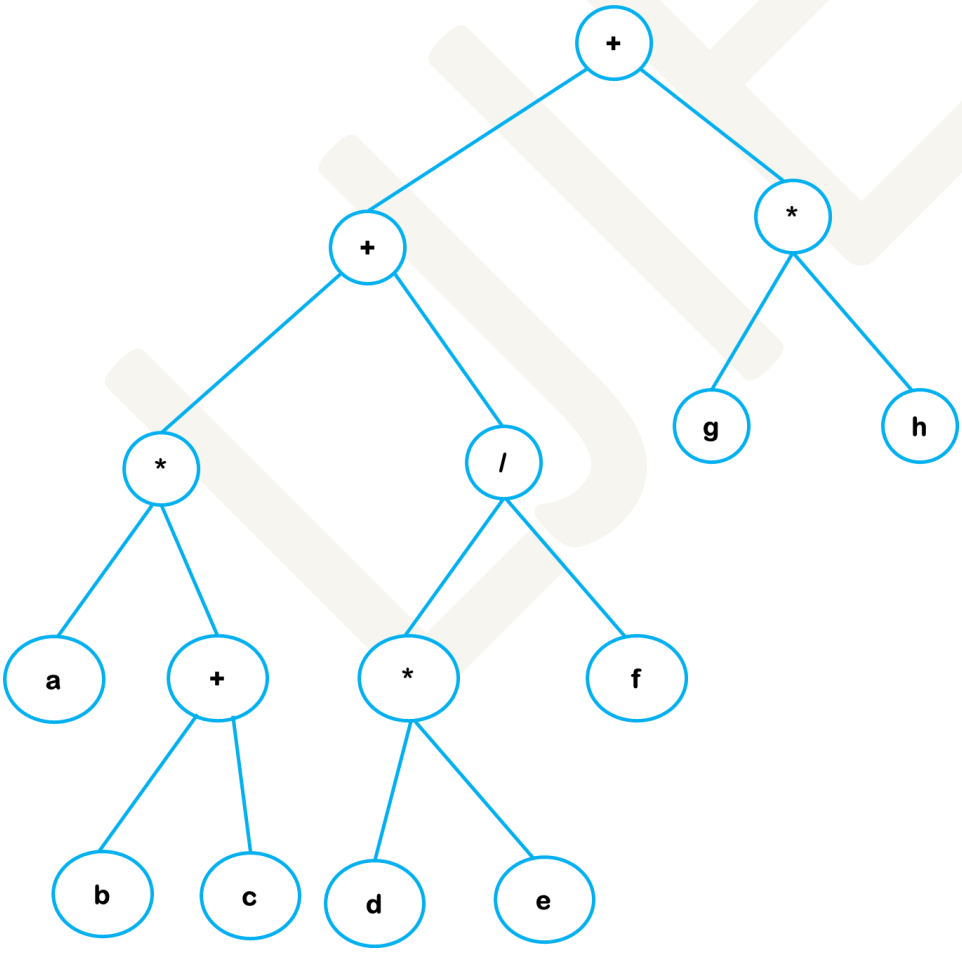
Step 5 – Now, again solve for operator number 3.



Step 6 – Now, again solving for number 2.



Step 7 – Now, last solving for number 1 operator would give us our final tree.



16. Threaded Binary Tree

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child, we use NULL pointer in that position. In any binary tree linked list representation, there are a greater number of NULL pointer than actual pointers.

Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play any role except indicating there is no link (no child).

And, that is why the concept of threaded binary tree has emerged. "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called threads.

A threaded binary tree defined as follows:

"A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

a) Why do we need Threaded Binary Tree?

Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals. Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.

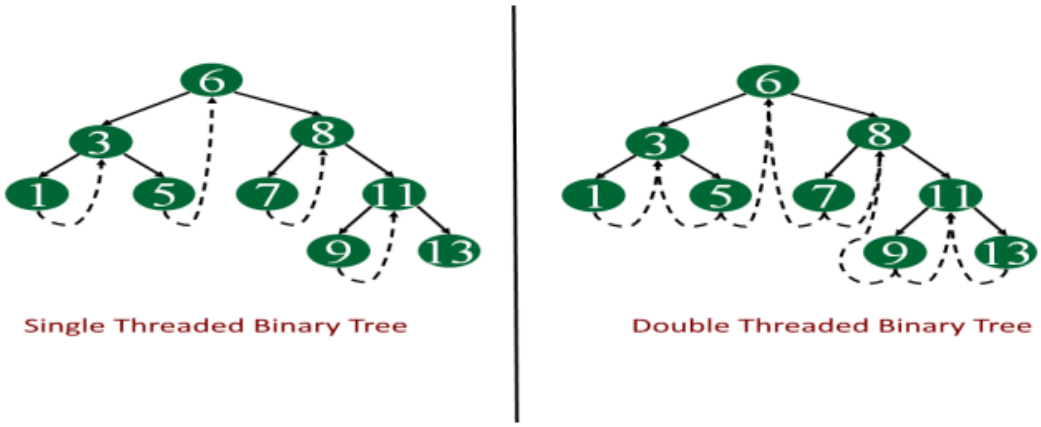
b) Types of threaded binary trees

Single Threaded: Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

Ex: Left threaded Tree and Right threaded Tree

Double threaded: Each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.

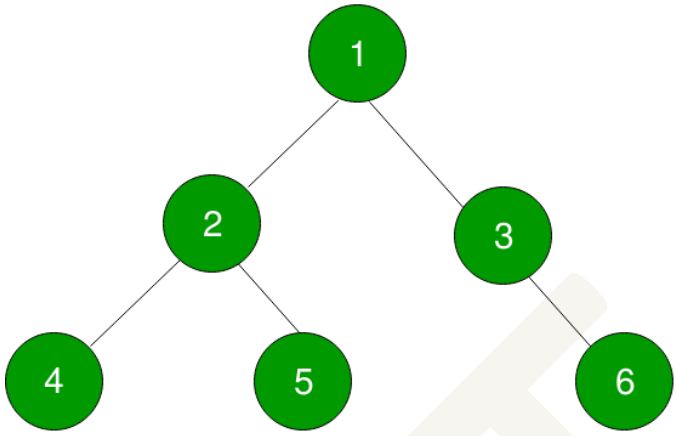
Ex: Fully threaded Tree



If you have noticed in the above definitions, there is mention of In-Order successor and In-Order Predecessor.

Let's learn how to find the In-Order successor and In-Order predecessor of a given value in any Binary Tree.

c) Find In-Order Successor and In-Order Predecessor



Step 1 – Find the In-Order Traversal for the given tree.

Here in our case the In-Order Traversal would be: 4-2-5-1-3-6

Step 2 – Highlight the value for which we need to find the traversal.

For example, we need to find the traversal for value 2.

In-Order Traversal would be: 4-**2**-5-1-3-6

Step 3 – The In-Order successor will be the next value of 2 in In-Order Traversal. And In-order predecessor will be the previous value of 2 in In-Order Traversal.

Hence, 4 would be In-Order predecessor and 5 would be In-Order successor of 2.