**5\_Binary\_Search\_Trees**

|  |  |  |  |
| --- | --- | --- | --- |
| **Level 1** | | | |
| 1. Height of Binary Tree |  | 1. Check for BST |  |
| 1. Determine if two trees are identical |  | 1. Array to BST |  |
| 1. Mirror tree |  | 1. Largest value in each level of binary tree |  |
| 1. Symmetric Tree |  | 1. Maximum GCD of siblings of a binary tr |  |
| 1. Diameter of tree |  | 1. Zigzag Tree Traversal |  |
| 1. Checked for Balanced tree |  | 1. Inorder Successor in BST |  |
| 1. Children Sum Parent |  | 1. Kth Largest Element in a BST |  |
| **Level 2** | | | |
| 1. Check if subtree |  | 1. Maximum sum leaf to root path |  |
| 1. Single Valued Subtree |  | 1. Odd Even Level Difference |  |
| 1. Unique BSTs |  | 1. Lowest Common Ancestor of a Binary Tree |  |
| 1. Inorder Traversal (iterative) |  | 1. Ancestors in Binary Tree |  |
| 1. Preorder Traversal (iterative) |  | 1. Remove BST keys outside the given range |  |
| 1. Postorder Traversal(iterative) |  | 1. Pair with given target in BST |  |
| 1. Vertical Traversal of a Binary Tree |  | 1. Sum Tree |  |
| 1. Boundary Traversal |  | 1. BST to greater sum tree |  |
| 1. Construct Binary Tree from Parent array |  | 1. BST to max heap |  |
| 1. Construct Binary Tree from Preorder and Inorder Traversal |  | 1. Clone binary tree with random pointer |  |
| 1. Preorder Traversal and BST |  | 1. Maximum sum of non adjacent nodes |  |
| 1. Construct tree from preorder traversal |  | 1. Largest BST in a Binary Tree |  |
| 1. Minimum distance between two given nodes |  | 1. Extreme nodes in alternate order |  |
| **Level 3** | | | |
| 1. Connect nodes at same level |  | 1. K-Sum Paths |  |
| 1. Nodes at given distance in a Binary Tree |  | 1. Number of turns in a binary tree |  |
| 1. Sorted Linked List to BST |  | 1. Merge two BST’s |  |
| 1. Binary Tree to Doubly Linked List |  | 1. Fixing two nodes of a BST |  |
| 1. Maximum sum path between two leaf nodes |  | 1. Burn Binary Tree |  |

|  |
| --- |
| Link : <https://www.geeksforgeeks.org/top-50-tree-coding-problems-for-interviews/> |

**5.1 Tree in Data Structure | Introduction to Trees**

* It is a nonlinear data structure
* It’s having multiple level
* Trees are used to represent the data items which are having hierarchical relationships between them
* The notes can contain data
* By default, the direction of the tree is top to bottom

Logical representation of tree in data structure

A diagram of a structure

Description automatically generated

**Tree - Terminology**

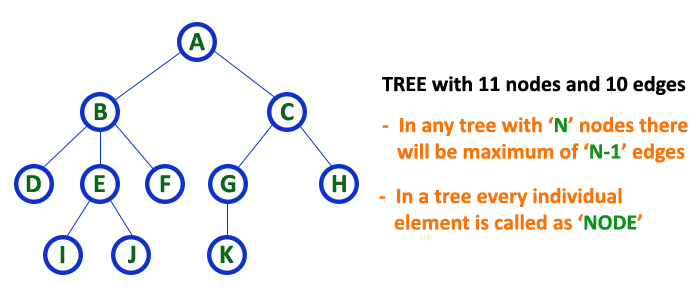
**Link:** [**http://www.btechsmartclass.com/data\_structures/tree-terminology.html**](http://www.btechsmartclass.com/data_structures/tree-terminology.html)

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

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

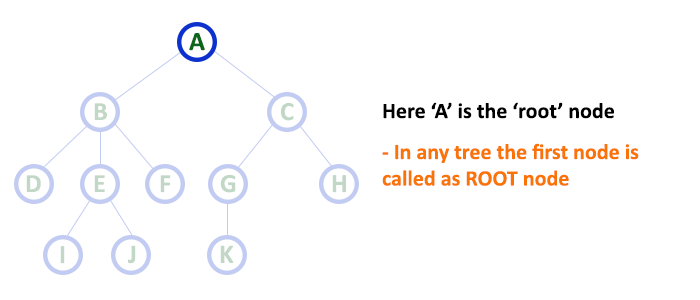
A tree data structure can also be defined as follows...

**Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively.**

In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that element and link to next element in hierarchical structure.  
  
In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links. **Example**

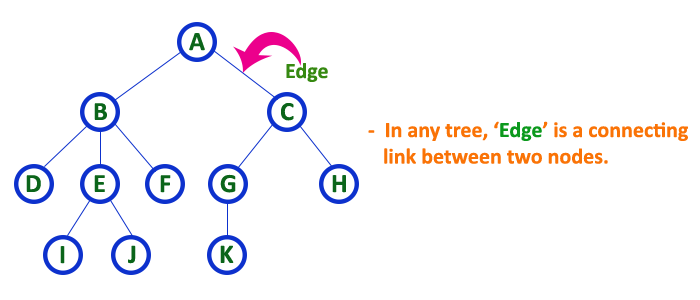
**Terminology**

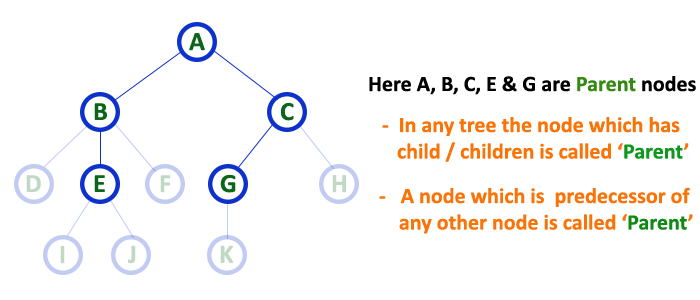
In a tree data structure, we use the following terminology...

**1. Root**

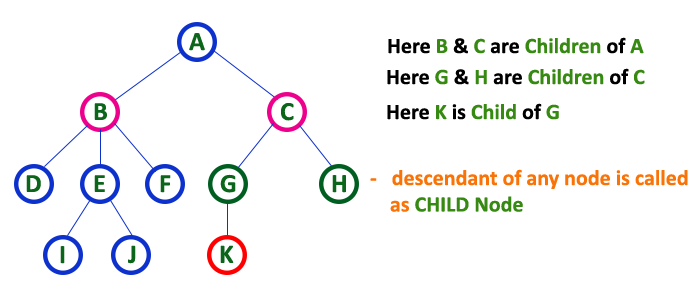
In a tree data structure, the first node is called **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

**2. Edge**

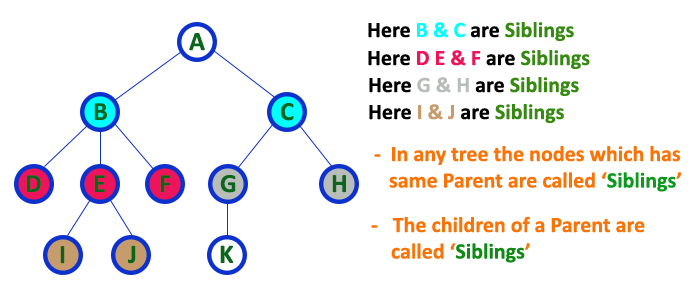
In a tree data structure, the connecting link between any two nodes is called **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.

**3. Parent**

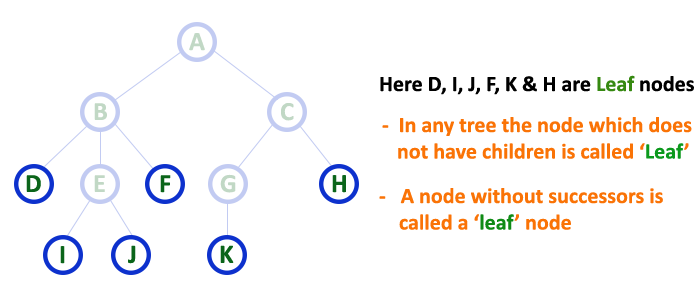
In a tree data structure, the node which is a predecessor of any node is called **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".

**4. Child**

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

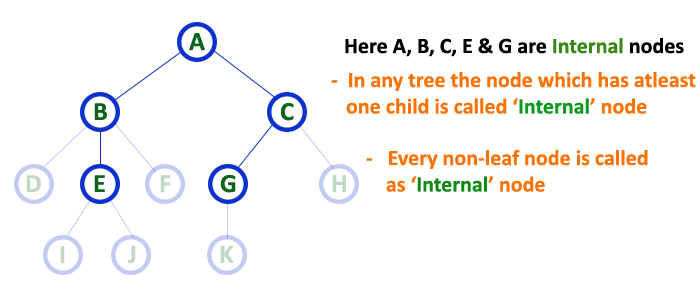
**5. Siblings**

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.

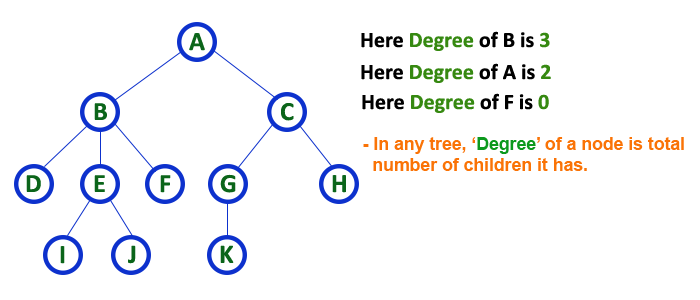
**6. Leaf**

In a tree data structure, the node which does not have a child is called **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.

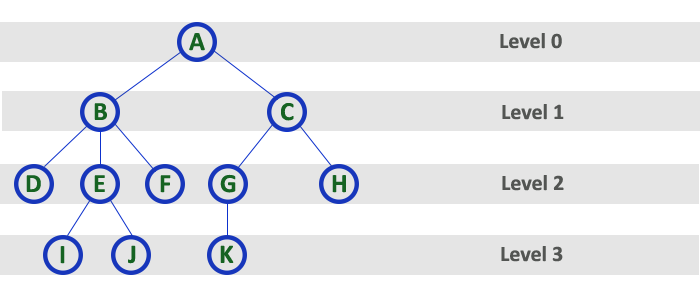
**7. Internal Nodes**

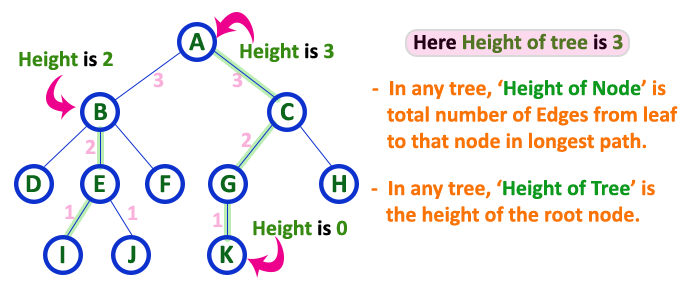
In a tree data structure, the node which has at least one child is called **INTERNAL Node**. In simple words, an internal node is a node with at least one child.  
  
In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**.**The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes

**8. Degree**

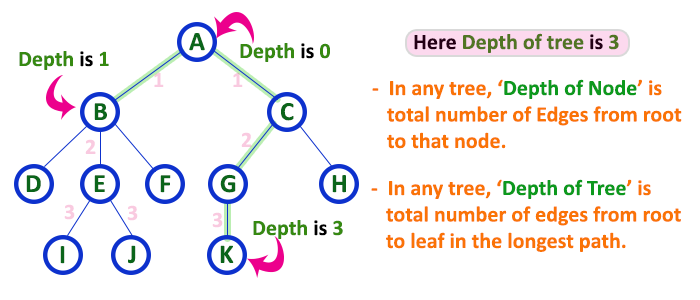
In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is the total number of children it has. The highest degree of a node among all the nodes in a tree is called '**Degree of Tree.**'

**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 a Level and the Level count starts with '0' and incremented by one at each level (Step).

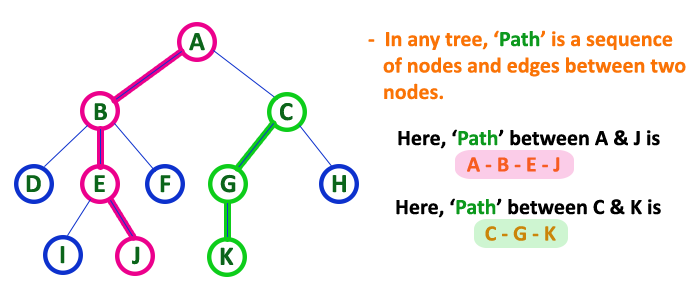
**10. Height**

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'.**

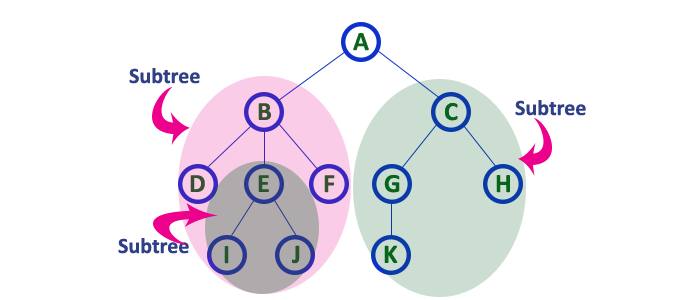
**11. Depth**

In a tree data structure, the total number of egdes 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 the 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 **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In the example **the path A - B - E - J 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.

|  |  |
| --- | --- |
| 1. **Root**: top most element (the node which does not have any parent) 2. Elements of trees are known as nodes. 3. **Parent Nod**e**:** immediate predecessor 4. **Child Node:** immediate successor 5. **Leaf Node / External Node:** The node which doesn’t have any child. 6. Degrees of the leaf node is 0. 7. **Non-leaf Node / Internal Node:** it has at least one child (all the other node except leaf node) 8. **Edge :** Link between two nodes (likes are uni (one angle) direction) 9. **Path:** It is a sequence of consecutive edges from source node to destination node 10. **Ancestor :** Any predecessor node on the path from root to the node 11. **Descended :** Any successor node on the path from the node to leaf node 12. **Sub-tree:** containing a tree and all of its descendants | 1. **Sibling:** All the children of same parent 2. **Degree:** Number of children of that node 3. **Degree of the tree :** Maximum degree of this tree 4. **Depth of node**: The length of a path from root to that node (number of edges from root to that node)    1. Depth of root is 0 5. **Hight of node:** no of edges in the longest path form that node to a leaf (path between that node to its leaf node) (max distance) 6. **Hight of the tree :** height of root node (Longest path from root node) 7. **Level of node :** number of edges between root to its the given node    1. (each hierarchy known as level) 8. **Level of node =** **Depth of node** 9. **Level of tree = Height of a tree**    1. If the has n nodes the tree will contain (n-1) edges (it can not be a cycle) |

|  |  |
| --- | --- |
| **Binary tree:**  **a**ctual implementation  binary tree example | **Application of tree:**   1. File system 2. Routing potocall 3. Organize the data for quick search (for insertion and deletion) |

* 1. = DONE
  2. **Binary Tree in Data Structure| Types of Binary Tree**

**Binary Tree Representations**

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...

**1. Array Representation of Binary Tree**

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.  
Consider the above example of a binary tree and it is represented as follows...



To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

**2. Linked List Representation of Binary Tree**

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.  
In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows...



Each node has a maximum of 2 children.



**Properties of binary tree:**

* Maximum number of solution possible at any level i is
* Maximum number of nodes: )
* Minimum number of nodes: ) ( h = height)
* Max height: n - 1
* Min height: (Taking the celling function )

|  |
| --- |
| **Types of binary tree:** |
| 1. **Full / Poper / Strict**   Lightbox  Each node contains 0 or 2 children or (each node will contain exactly 2 node except leaf node)  Number of leaf nodes = number of internal nodes + 1 |
| 1. **A blue lines with dots     Description automatically generatedComplete binary tree**   If all the level are filled except possibly the last level  Last level has nodes at left as possible |
| 1. **Perfect Binary Tree (Recursive Representation)Perfect Binary tree**   If all the internal node has two children and all the leaf node are in same level  Perfect binary tree = Complete binary tree = Full binary tree |
| 1. **Degenerate binary tree / left skewed binary tree /right skewed binary tree**   Degenerate (or pathological) treeAll the internal node has only 1 child (containing only the left child) (containing only the right child) |
| 1. **Balanced tree** |
| A screenshot of a computer screen  Description automatically generated |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Max nodes | Min nodes | Min height | Max height |
| Binary tree | ) | h + 1 |  | n - 1 |
| Full binary tree | ) | 2h + 1 |  |  |
| Complete binary tree | ) |  |  |  |

* 1. = DONE

**5.7 Construct Binary Tree from Preorder and Inorder Traversal | Example**

Pre order: 1 2 4 8 9 10 11 5 3 6 7 (Root Left Right)

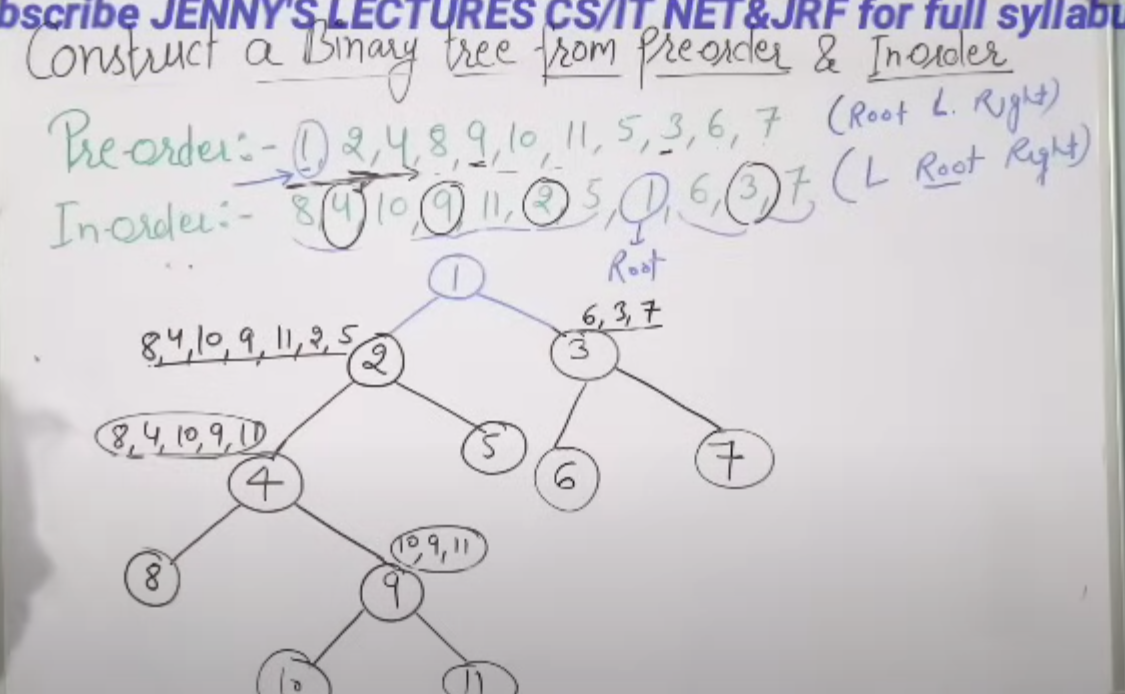
In order: 8 4 10 9 11 2 5 1 6 3 7 (Left Root Right)

We will be using be using **preorder for finding the roots**

Scan preorder left to right

And we will use inorder to **find the left and right portion of the tree**

In left and right portion which value comes first will be the root (Scan Left 🡪 right)



**5.8 Construct Binary Tree from Postorder and Inorder with example**

Post order: 9 1 2 12 7 5 3 11 4 8 (Left Right Root)

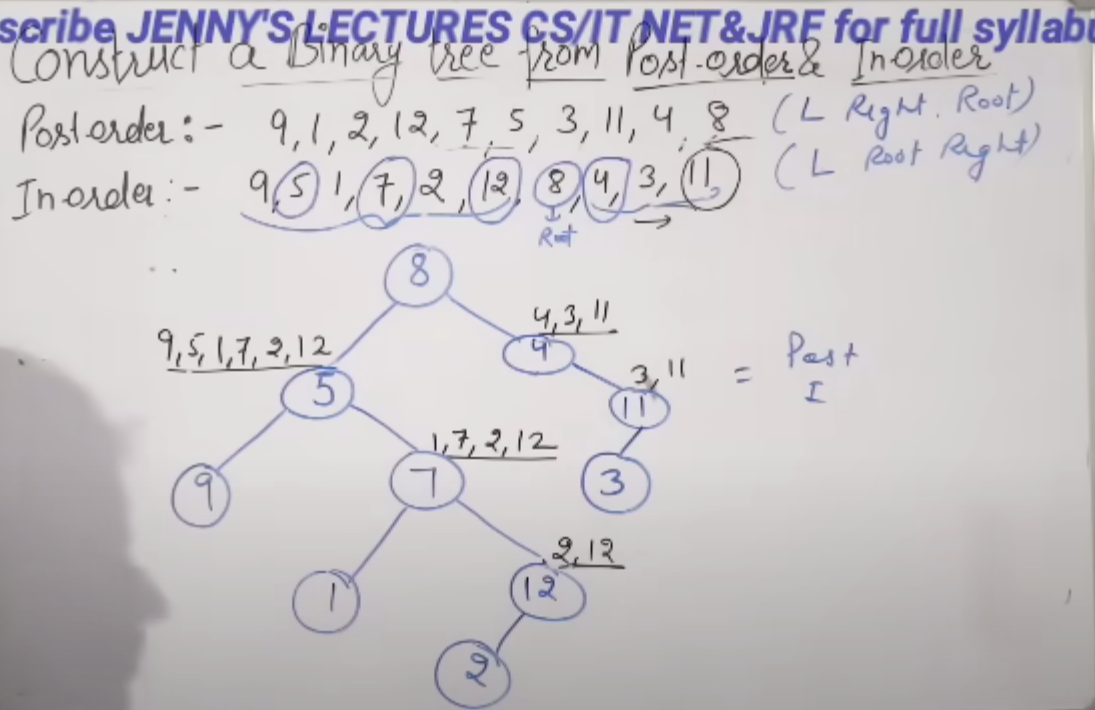
In order: 9 5 1 7 2 12 8 4 3 11 (Left Root Right)

Find root from post order

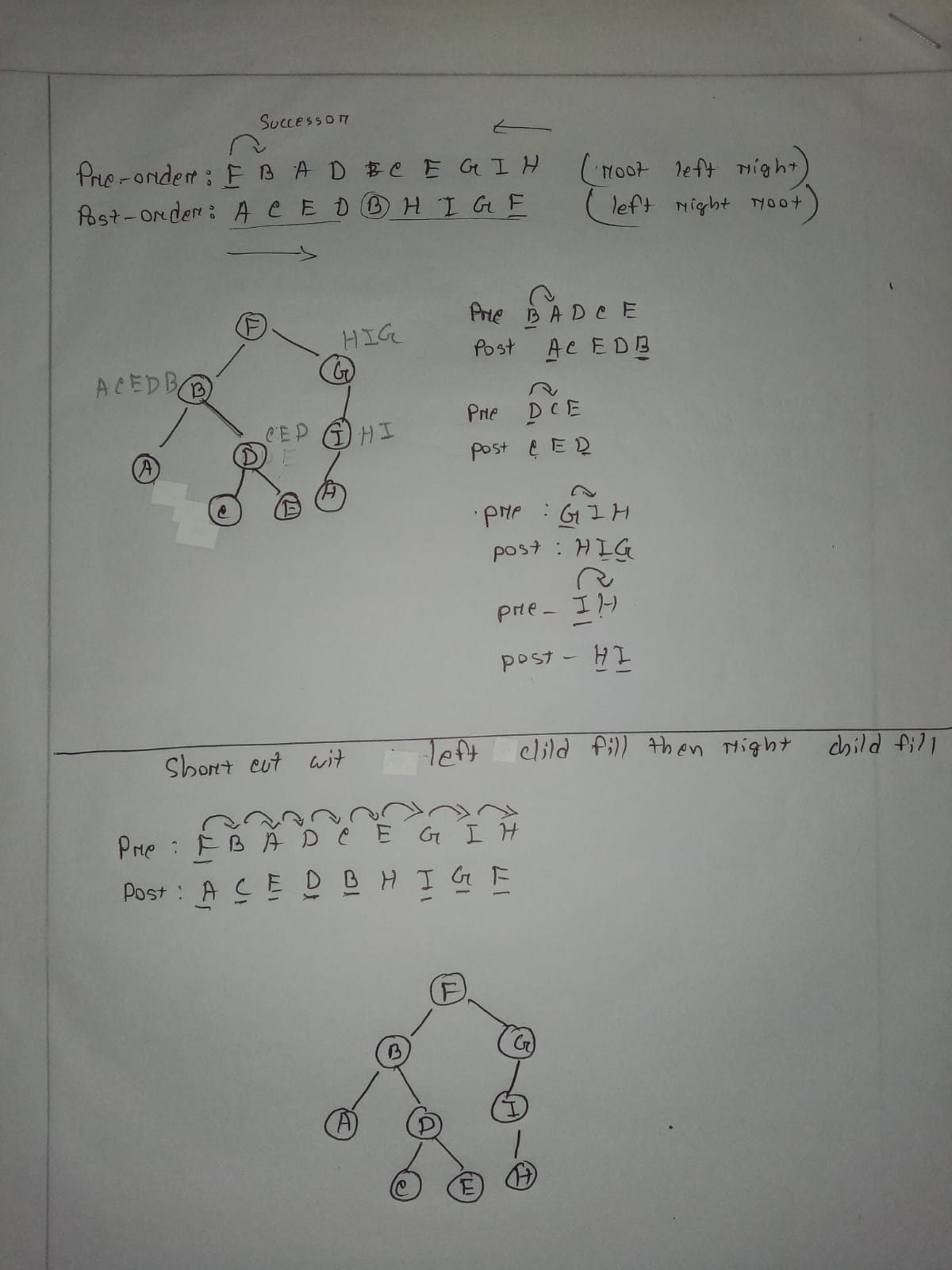
Scan post order from right to left

Left and right portion of the tree from in order

In left and right portion which value comes first will be the root (Scan right 🡪 left)



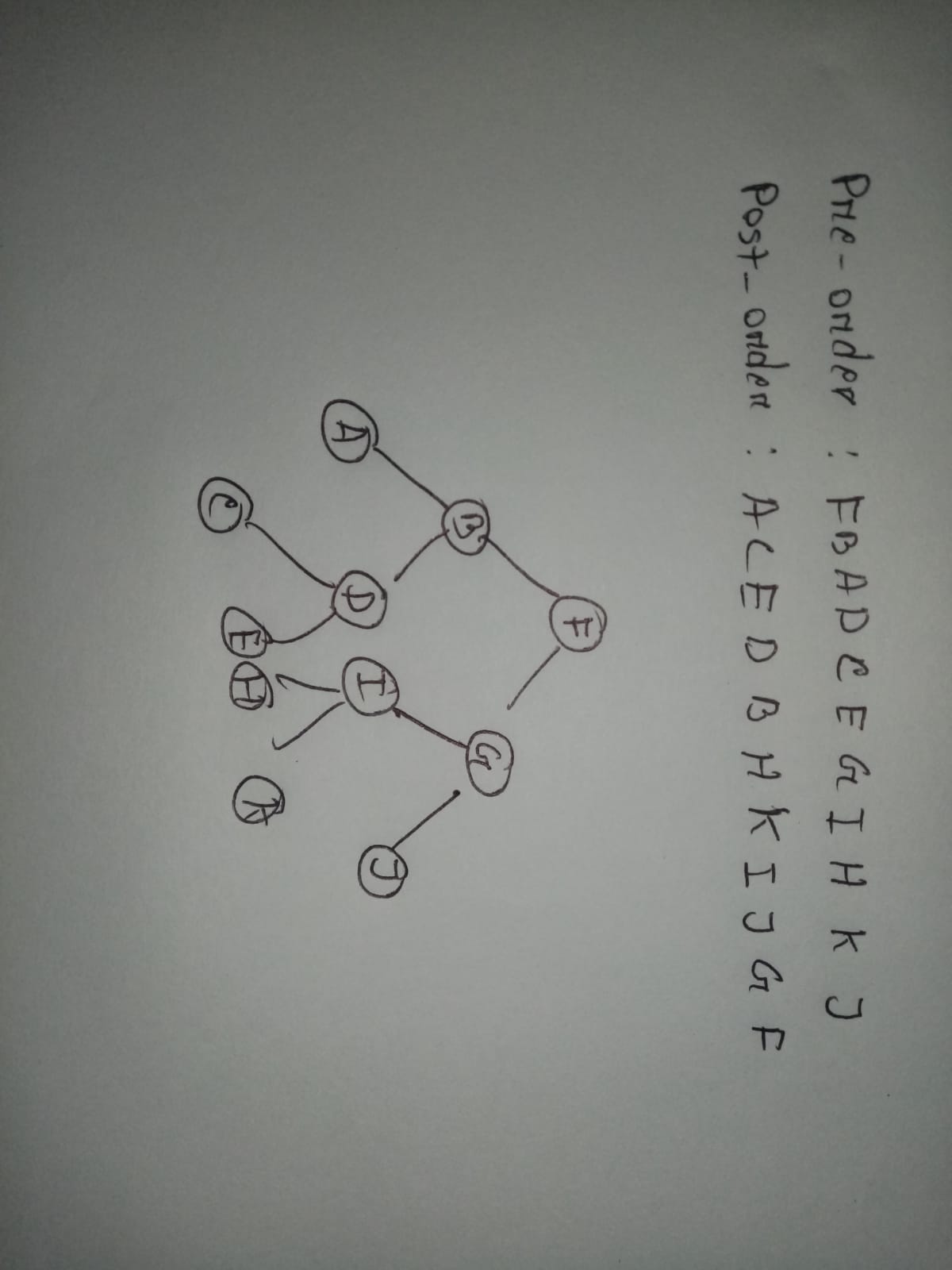
**5.9 Construct Binary Tree from Preorder and Postorder traversal**

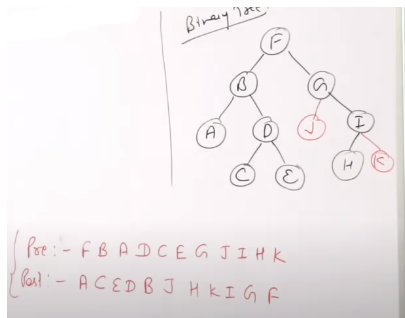


We can get only unique binary tree

It’s full binary tree condition:

Each node has 0 or 2 childern





**Height is binary tree**

/// Find the height of a tree defined by the root node

int treeHeight(Node\* root) {

if (root == nullptr)

return 0;

else {

// Find the height of left and right subtrees

int leftHeight = treeHeight(root->left);

int rightHeight = treeHeight(root->right);

// Find max(subtree\_height) + 1 to get the height of the tree

return std::max(leftHeight, rightHeight) + 1;

}

}

**Depth if a binary tree**

// Calculate the maximum depth (height) of the binary tree

int maxDepth(TreeNode\* root) {

if (root == nullptr)

return 0; // Empty tree has height 0

else {

// Recursively calculate the depth of left and right subtrees

int leftDepth = maxDepth(root->left);

int rightDepth = maxDepth(root->right);

// Return the maximum depth plus 1 (for the current node)

return std::max(leftDepth, rightDepth) + 1;

}

}

|  |  |  |
| --- | --- | --- |
| **Preorder traversal**  // Preorder traversal: Root -> Left -> Right  void preorderTraversal(TreeNode\* root) {  if (root == nullptr){  return;  }    cout << root->data << " ";  preorderTraversal(root->left);  preorderTraversal(root->right);  } | **In- order**  void inorderTraversal(TreeNode\* root) {  if (root == nullptr)  return;  inorderTraversal(root->left);  cout << root->val << " ";  inorderTraversal(root->right);  } | **Post order**  void postOrderTraversal(TreeNode\* root) {  if (root == nullptr)  return;  postOrderTraversal(root->left);  postOrderTraversal(root->right);  cout << root->val << " ";  } |

**Built a binary tree**

Node\* buildTree() {

int d;

cin >> d;

if (d == -1)

return NULL;

Node\* root = new Node(d);

root->left = buildTree();

root->right = buildTree();

return root;

}