Documentation

### **Topics Covered**

**Tree Overview** • Tree Structure  
 • Logical Representation  
 • Memory Representation  
 • Types of Tree (Binary, General)  
 • Terminology

* Root
* Child
* Parent
* Leaf (Terminal node)
* Non-Leaf
* Sibling
* Edge
* Level
* Height of Node
* Height of Tree
* Degree of Node
* Degree of Tree
* Subtree
* Ancestors
* Descendants
* Path
* Depth of Node
* Forest

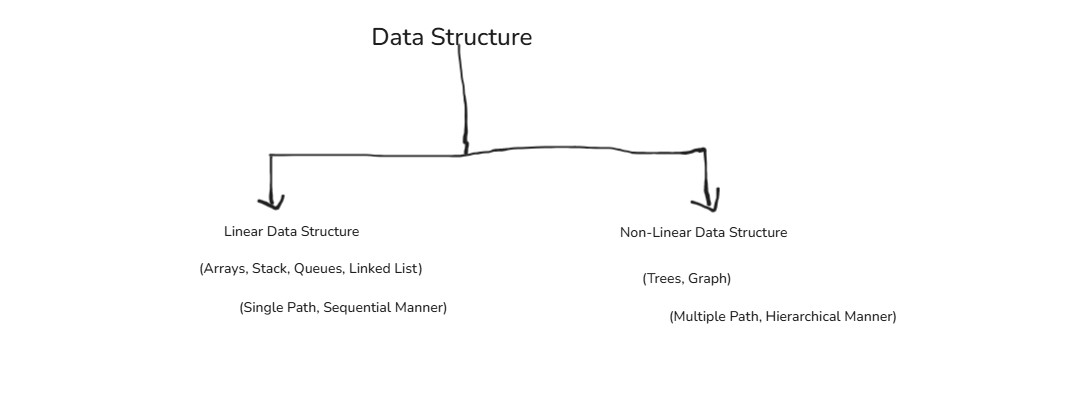
**Binary Trees** • Creation  
 • Traversal  
 • Pre-order Traversal  
 • In-order Traversal  
 • Post-order Traversal  
 • Logical Representation  
 • Memory Representation  
 • Code Practice

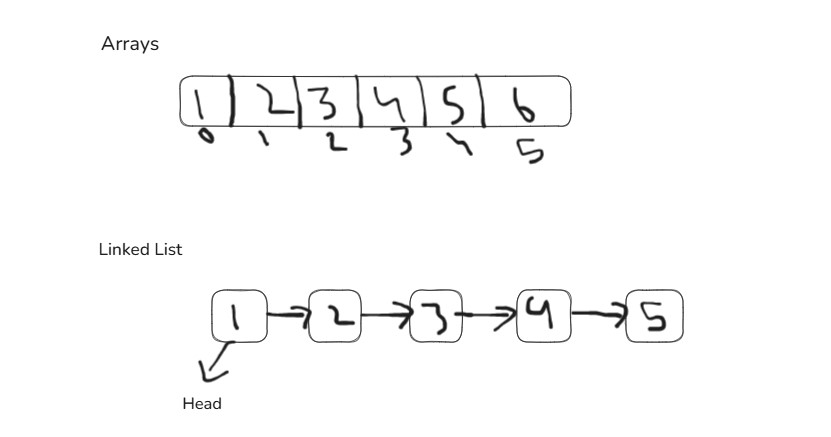
**Binary Search Trees (BST)** • Insertion  
 • Deletion  
 • Traversals

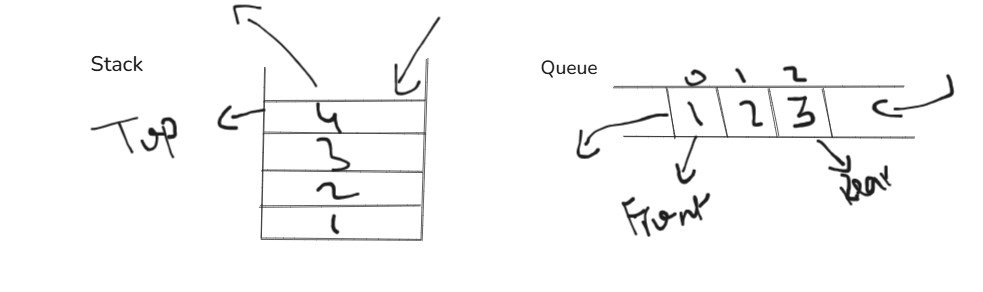
**AVL Trees (Advanced BST)** • Introduction  
 • Need & Features  
 • Self-Balancing Property  
 • Balance Factor (BF)  
 • Rotations (Left, Right, Left-Right, Right-Left)  
 • Traversals (Inorder, Preorder, Postorder)  
 • Time Complexity: O(log n) vs O(n) in skewed BST  
 • Insertion: AVL vs Skewed BST  
 • Example: 10 → 20 → 30 with Rotation  
 • Key Takeaways

Trees

**Overview**

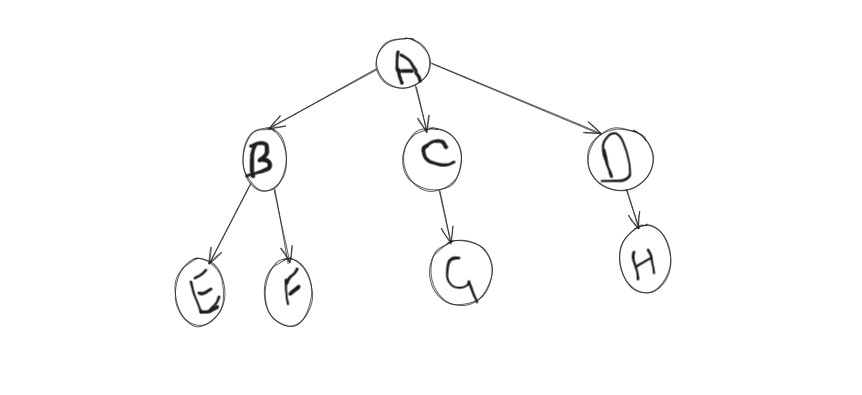
****

****

****-------------------------------------------------------------------------------

**Trees**

* Tree is also another Data Structure, but it’s a non-linear data structure.
* Trees can be defined as a collection of nodes (entities) linked together to form a hierarchy.
* Trees have multiple paths
* Trees also follows top to down direction

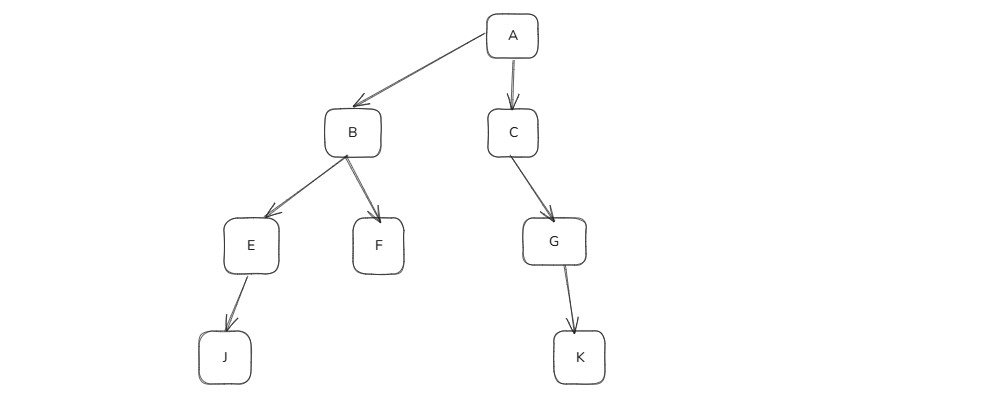


**Tree Structure**

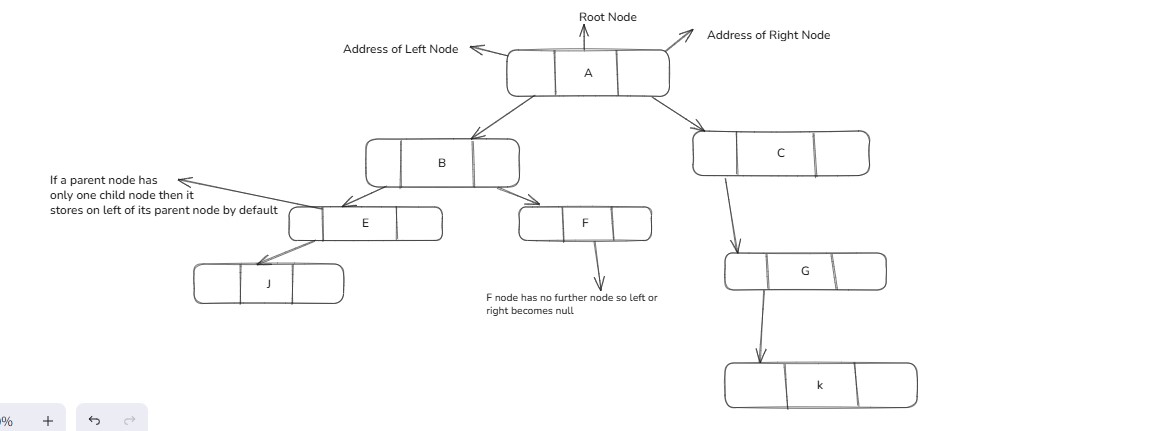
* **Root:** The very first node of a tree, or a node that has no parent and is not derived from any other node (A is a root node).
* **Node:** The entities or elements of a tree are called nodes (A, B, C, D, E, F, etc.).
* **Parent Node:** Nodes from which other nodes originate or are connected are called parent nodes (A is a parent from which B, C, and D originate; similarly, B is a parent from which E and F originate).
* **Child Node:** Nodes that are derived from or originate from a parent node are called child nodes (B, C, and D are children of A; E and F are children of B).
* **Leaf Nodes:** Also called *external nodes*, these are nodes that have no children or from which no other node originates (E, F, G, and H are leaf nodes).
* **Non-Leaf Nodes:** Also called *internal nodes*, these are nodes that have at least one child or from which at least one node originates (A, B, C, and D are non-leaf nodes).
* **Path:** The route or way to traverse from one node to another in a tree is called a path. For example, to go from A to E, first move from A to B, and then from B to E — that forms the path.
* **Edge**: The link or connection between two nodes in a tree is called an edge. (If A is connected to B, then the connection between A and B is an edge.)
* **Ancestor:** The nodes that form the path from the root to any specific node are called its ancestors. (For example, if the path to G is A → C → G, then the ancestors of G are A and C.)
* **Descendant:** The nodes that come after a specific node along the path toward the leaf nodes (nodes with no children) are called its descendants. (For example, if the path from A to the leaf node F is A → B → F, then the descendants of A are B and F.)
* **Subtrees:** The group of nodes connected to any particular node forms its subtree. (For example, if A is the root node connected to B, and B is connected to E and F, then A has subtrees rooted at B, and B has a subtree containing E and F.)
* **Sibling:** Nodes that originate from the same parent node are called siblings. (For example, B, C, and D are derived from the same node A, so B, C, and D are siblings.)
* **Degree:** The number of child nodes a parent node has is called its degree. (For example, if A has children B, C, and D, then the degree of A is 3.)
* **Degree of a Tree:** The degree of a tree is the maximum number of child nodes that any node in the tree has. (For example, if A has 3 children — B, C, and D — B has 2 children — E and F — C has 1 child — G — and D has 1child — H —then since A has the highest number of children (3), the degree of the tree is 3.)
* **Depth of a Node:** The depth of a node is the length or number of edges (connections) from the root node to the specific node. (For example, to find the depth from A to F — A → B → F — there are two edges: one from A to B and another from B to F, so the total **depth is 2**.)
* **Height of a Node:** The height of a node is the length or number of edges in the longest path from that node to any of its leaf nodes.  
   (For example, if B has children E and F, and both E and F are leaf nodes, then the longest path from B to a leaf node is one edge, so the height of B is **1**. But if E further has children I and J, then the longest path from B → E → I (or J) has two edges, so the height of B becomes **2**.)
* **Height of a Tree:** The height of a tree is the length or number of edges in the longest path from the root node to the farthest leaf node.  
   (For example, if A is the root node with children B, C, and D — B has E and F, C has G, and D has H — then the longest path is A → B → E (or F), which has two edges. Hence, the **height of the tree is 2**.)
* **Level of a Node:** The level of a node is the number of connections (or edges) from the root node to that specific node — it is the same concept as the depth of a node.  
   (For example, from A → B → E, there are two connections, so the **level of E is 2**.)
* **Level of a Tree:** The level of a tree represents how many levels (or layers) the tree contains from the root to the deepest leaf node.Level of a Tree is same as Height of Tree.  
   (For example, if A is at level 0, B, C, and D are at level 1, and E, F, G, and H are at level 2, then the tree has **2 levels**.)
* **Forest:** Forest means different multiple trees which are not connected with each other.

**Representation of a Tree**

**Logical**

****

**Memory**

****

Types of Tree

**Types of a Tree**

* Binary Tree
* General Tree

# **Binary Tree**

A binary tree is a tree in which each node has at most 2 children, usually called left and right.

### **Properties:**

* Max children per node = 2
* Used in **binary search tree, heaps, expression trees**
* Height of tree ≤ n (depends on shape)

**General Tree**

A general tree (or n-ary tree) allows any number of children per node.

**Key difference:**

| **Feature** | **Binary Tree** | **General Tree** |
| --- | --- | --- |
| Max children/node | 2 | Unlimited |
| Children storage | left & right | Array/list or firstChild+nextSibling |
| Example | A → B, C | A → B, C, D, E … |

Binary Trees

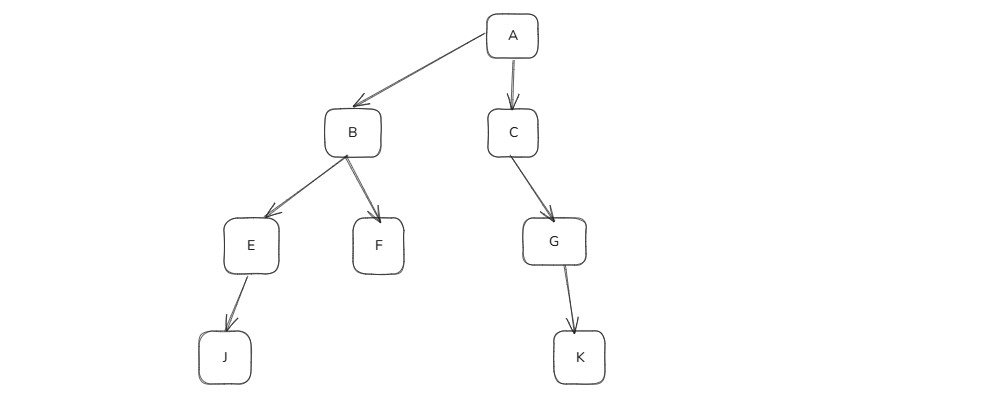
**Binary Trees**

A binary tree is a tree in which each node has at most 2 children, usually called left and right.

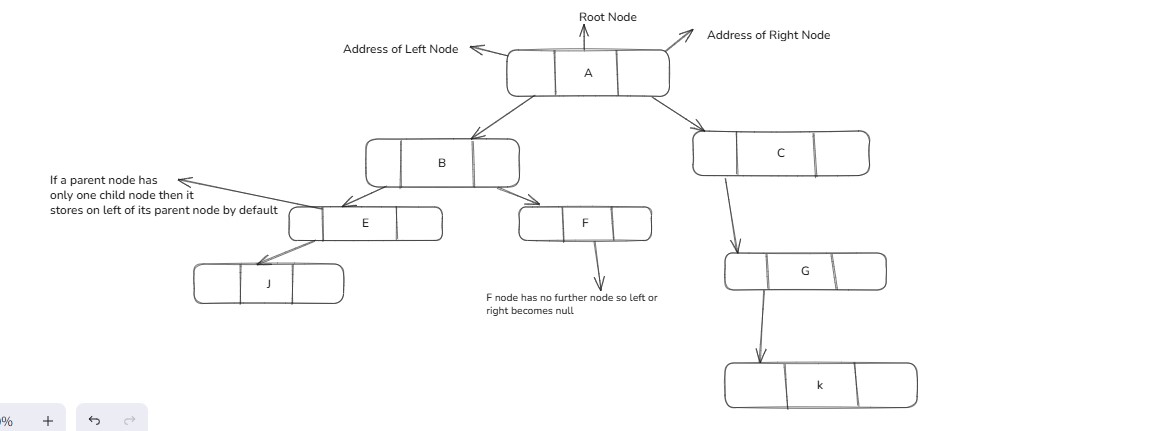
### **Properties:**

* Max children per node = 2
* Used in **binary search tree, heaps, expression trees**
* Height of tree ≤ n (depends on shape)

**Logical Representation**

****

**Memory Representation**

****

**Level of Binary Tree**

1

/ \

2 3

/ \ \

4 5 6

**Case 1:** If levels start from 0

Node Level

1 0

2, 3 1

4, 5, 6 2

**➡️** Total levels = 3 (0, 1, 2)

**➡️** Height = 2 (last level number)

**Case 2:** If levels start from 1

Node Level

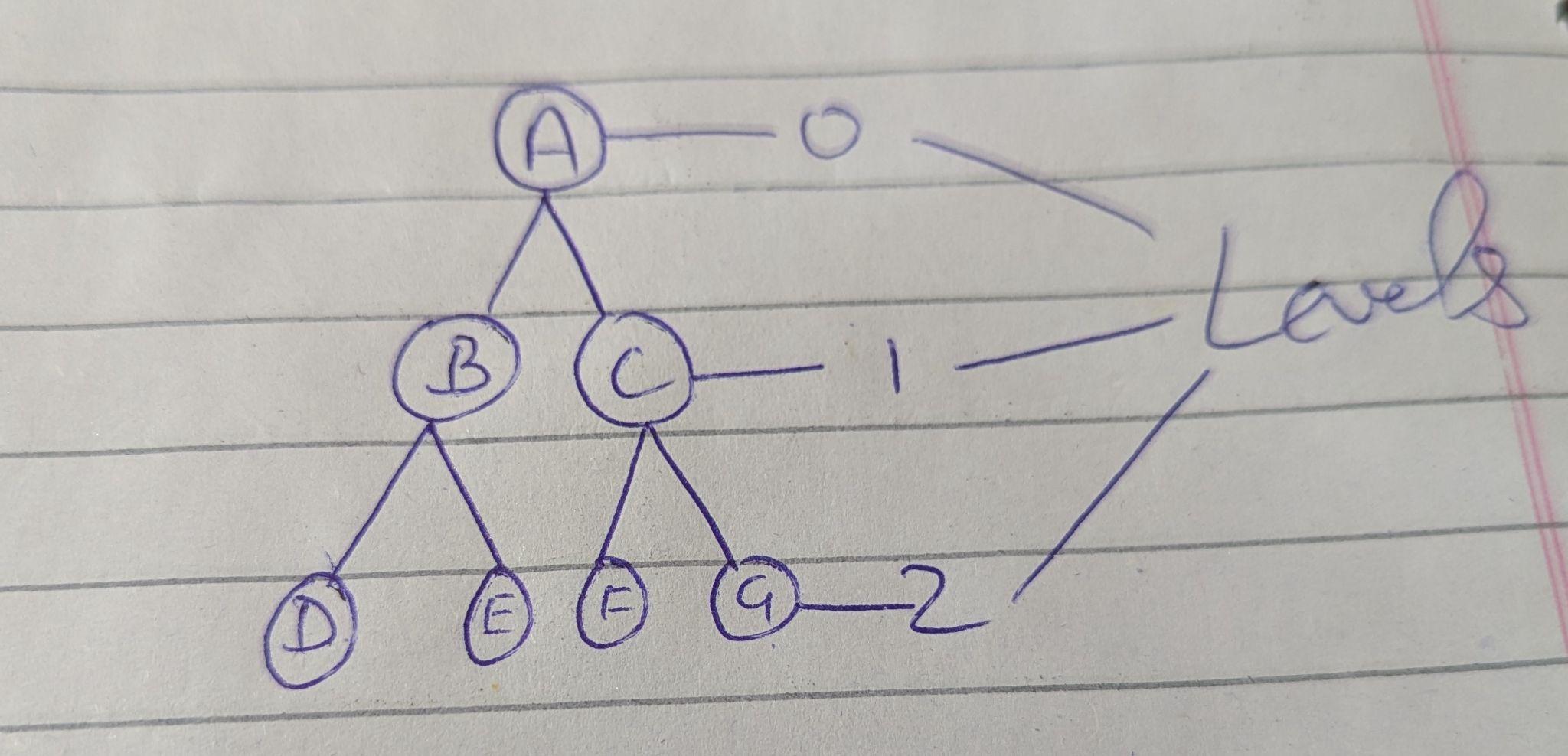
1 1

2, 3 2

4, 5, 6 3

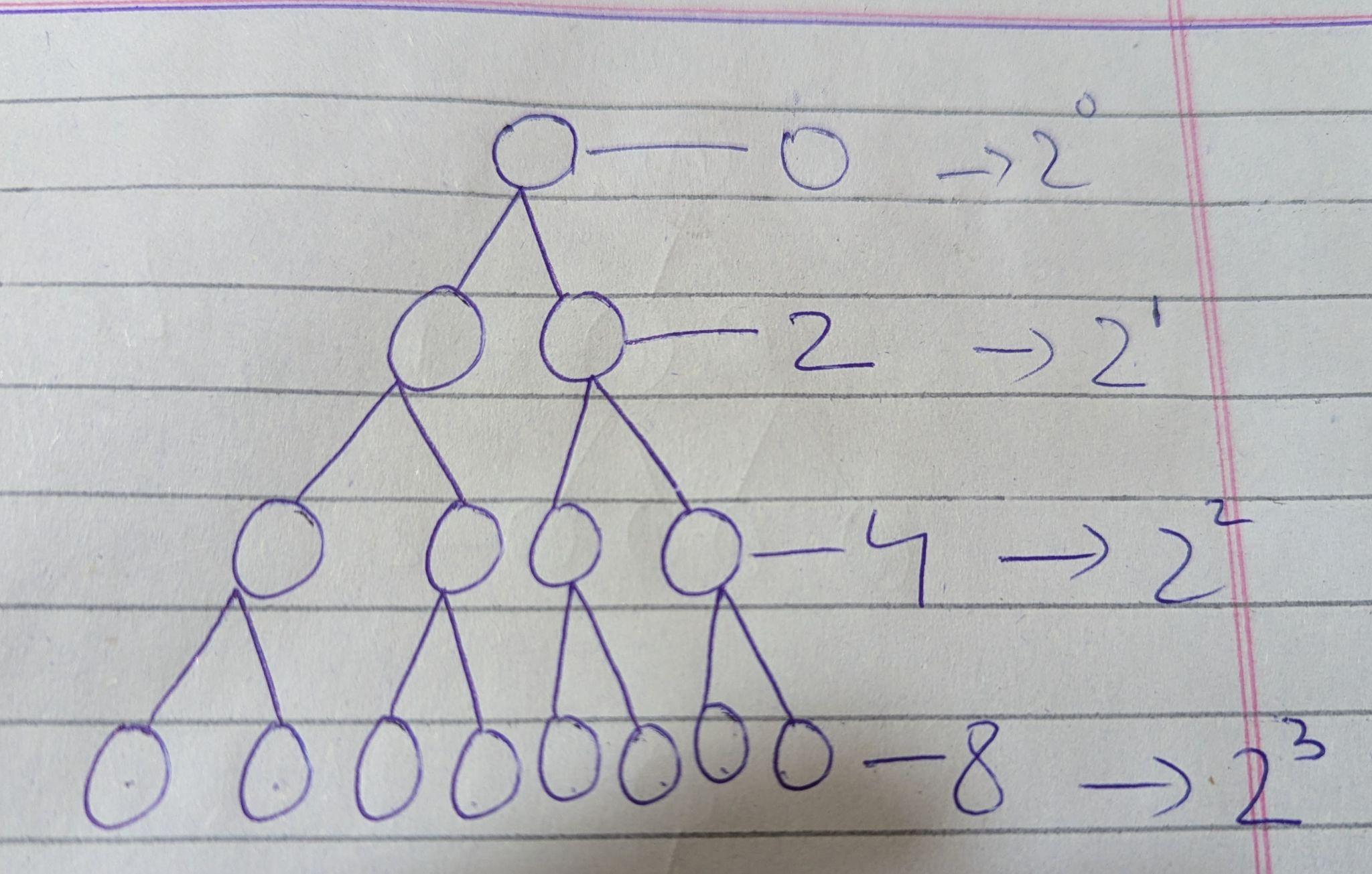
**➡️** Total levels = 3 (1, 2, 3)

**➡️** Height = 3 (last level number)



**Possibility of Nodes**

Max no of nodes possible at any level i is 2^i



**Build Tree Preorder**

Means How do we build a tree?

We have to build the tree using a **preorder traversal sequence**.  
In preorder traversal, the data sequence is **already given** — we use that sequence to **rebuild the tree**.

For example,  
Preorder sequence:

1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1

Here:

* The numbers represent **node values**,
* And -1 represents **NULL nodes** or **leaf nodes** (places where no child exists).

Rules:

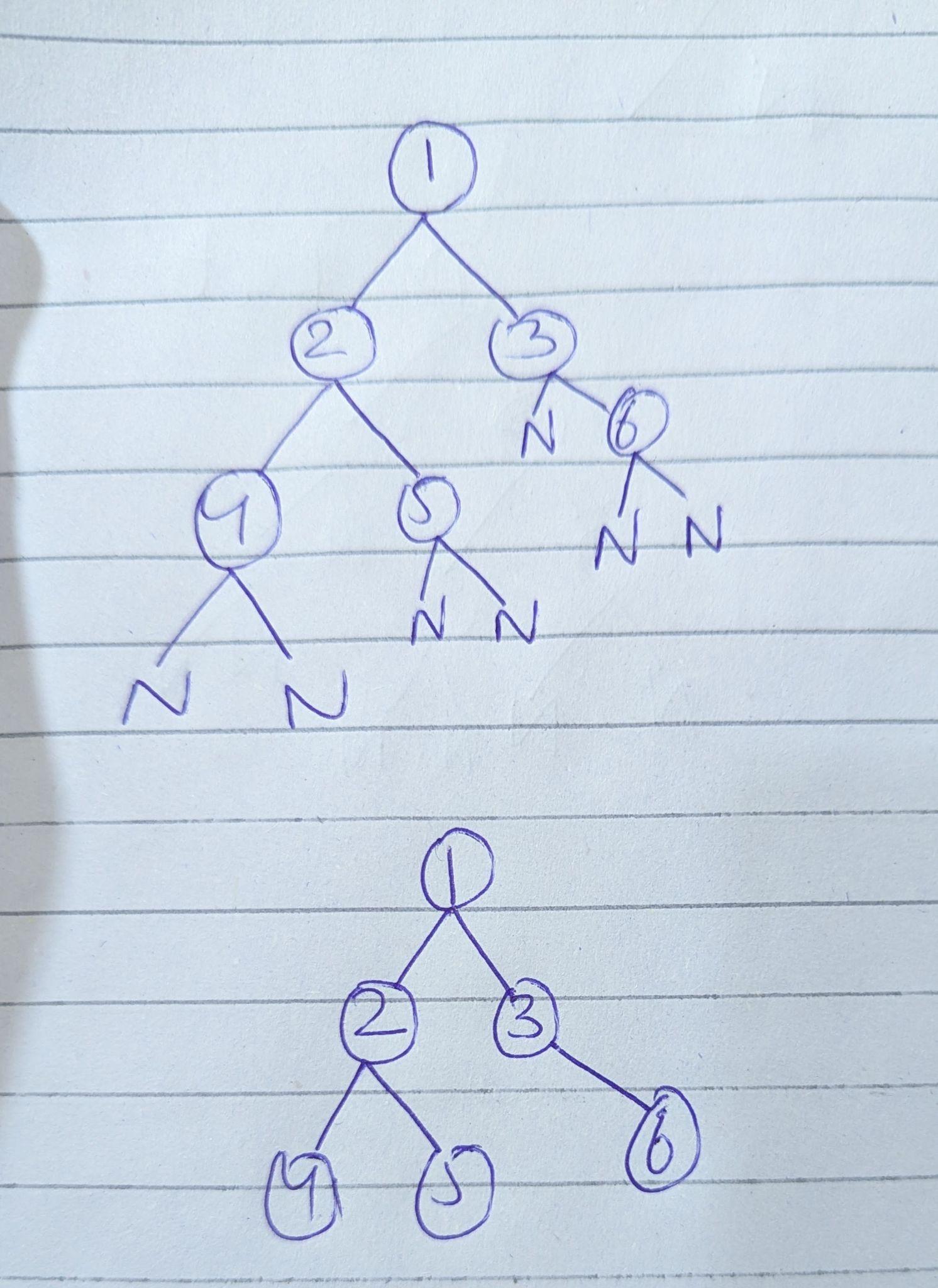
1. Take the **first element** as the root.
2. Recursively build its **left subtree** from the next elements.
3. Then recursively build its **right subtree**.

Whenever you see -1, it means that branch has **no child**, so return null.

Sequence:  
 [1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1]

Let’s trace:

1. 1 → root node
2. Build **left** of 1 → next is 2
3. Build **left** of 2 → next is 4
4. Next two -1 -1 → means 4 has no left/right child → leaf
5. Go back → build **right** of 2 → next is 5
6. Next two -1 -1 → 5 also a leaf
7. Go back → 2 done → go back to root (1)
8. Build **right** of 1 → next is 3
9. Build **left** of 3 → next is -1 → no left child
10. Build **right** of 3 → next is 6
11. Next two -1 -1 → 6 is a leaf
12. Done ✅



**Data - Preorder Sequence:**

[1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1]

**Python Code**

[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/01.Binary%20Trees.py)

Traversal

**Traversal:**

Traversal is the process of visiting all the nodes in a tree or graph systematically in a specific order so that we can access or process their data.

### **Types of Binary Tree Traversal**

1. **Inorder (Left → Root → Right)**
   * Visit the left subtree first, then the root, then the right subtree.
   * **Use:** Retrieve elements in sorted order in a Binary Search Tree (BST).
2. **Preorder (Root → Left → Right)**
   * Visit the root first, then left subtree, then right subtree.
   * **Use:** Copy the tree or store its structure.
3. **Postorder (Left → Right → Root)**
   * Visit left subtree, then right subtree, then root.
   * **Use:** Delete nodes, calculate sizes, bottom-up processing.
4. **Level Order (Breadth-First Traversal)**
   * Visit nodes level by level from top to bottom.
   * **Use:** BFS operations, shortest path, printing level-wise.

Pre-order Traversal

**Pre-order Traversal**

Traversal means travel in a tree.

Preorder Traversal has some **Rules:**

* Root
* Left Subtree
* Right Subtree

**Python Code**

[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/02.Preorder%20Travsersal.py)

### **Preorder Traversal Call Flow (print\_tree\_preorder(node))**

* Call: print\_tree\_preorder(1)  
  + Node = 1
  + Print 1
  + Call left subtree: print\_tree\_preorder(2)  
    - Node = 2
    - Print 2
    - Call left subtree: print\_tree\_preorder(4)  
      * Node = 4
      * Print 4
      * Call left → None → return
      * Call right → None → return
    - Return to previous call (node 2)
    - Call right subtree: print\_tree\_preorder(5)  
      * Node = 5
      * Print 5
      * Call left → None → return
      * Call right → None → return
    - Return to previous call (node 2 → node 1)
  + Back to node 1
  + Call right subtree: print\_tree\_preorder(3)  
    - Node = 3
    - Print 3
    - Call left → None → return
    - Call right subtree: print\_tree\_preorder(6)  
      * Node = 6
      * Print 6
      * Call left → None → return
      * Call right → None → return
    - Return to previous call (node 3 → node 1)
* Back to node 1
* Function ends

**Output**

1 2 4 5 3 6

In-order Traversal

**In-order Traversal**

Traversal means travel in a tree.

In-order Traversal has some **Rules:**

* Left Subtree
* Root
* Right Subtree

**Python Code**

[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/03.Inorder%20Traversal.py)

### **Inorder Traversal Call Flow (print\_tree\_inorder(node))**

* Call: print\_tree\_inorder(1)  
  + Node = 1
  + Call left subtree: print\_tree\_inorder(2)  
    - Node = 2
    - Call left subtree: print\_tree\_inorder(4)  
      * Node = 4
      * Call left → None → return
      * Print 4
      * Call right → None → return
    - Return to previous call (node 2)
    - Back to node 2
    - Left done
    - Print 2
    - Call right subtree: print\_tree\_inorder(5)  
      * Node = 5
      * Call left → None → return
      * Print 5
      * Call right → None → return
    - Return to previous call (node 2 → node 1)
  + Back to node 1
  + Left subtree done
  + Print 1
  + Call right subtree: print\_tree\_inorder(3)  
    - Node = 3
    - Call left → None → return
    - Print 3
    - Call right subtree: print\_tree\_inorder(6)  
      * Node = 6
      * Call left → None → return
      * Print 6
      * Call right → None → return
    - Return to previous call (node 3 → node 1)
* Back to node 1
* Right subtree done

**Output**

4 2 5 1 3 6

Post-order Travsersal

**Post-order Traversal**

Traversal means travel in a tree.

Post-order Traversal has some **Rules:**

* Left Subtree
* Right Subtree
* Root

**Python Code**

[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/04.Postorder%20Traversal.py)

### **Postorder Traversal Call Flow (print\_tree\_postorder(node))**

* Call: print\_tree\_postorder(1)  
  + Node = 1
  + Call left subtree: print\_tree\_postorder(2)  
    - Node = 2
    - Call left subtree: print\_tree\_postorder(4)  
      * Node = 4
      * Call left → None → return
      * Call right → None → return
      * Print 4
    - Return to previous call (node 2)
    - Call right subtree: print\_tree\_postorder(5)  
      * Node = 5
      * Call left → None → return
      * Call right → None → return
      * Print 5
    - Return to previous call (node 2 → node 1)
    - Print 2
  + Back to node 1
  + Call right subtree: print\_tree\_postorder(3)  
    - Node = 3
    - Call left → None → return
    - Call right subtree: print\_tree\_postorder(6)  
      * Node = 6
      * Call left → None → return
      * Call right → None → return
      * Print 6
    - Return to previous call (node 3 → node 1)
    - Print 3
* Back to node 1
* Print 1
* Function ends

**Output**

4 5 2 6 3 1

Binary Tree Construction

**Binary Tree Construction**

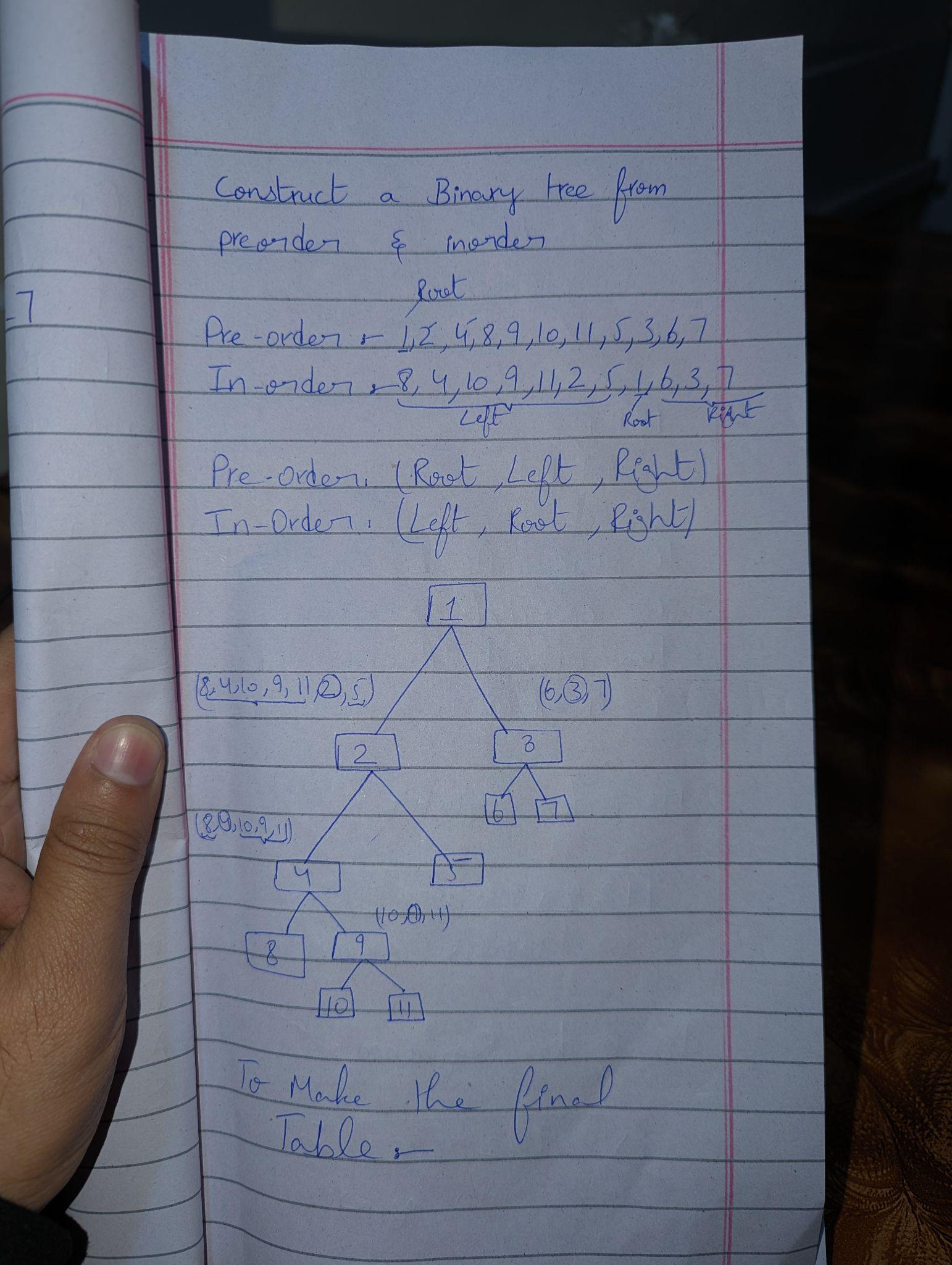
There are two ways to build binary trees.

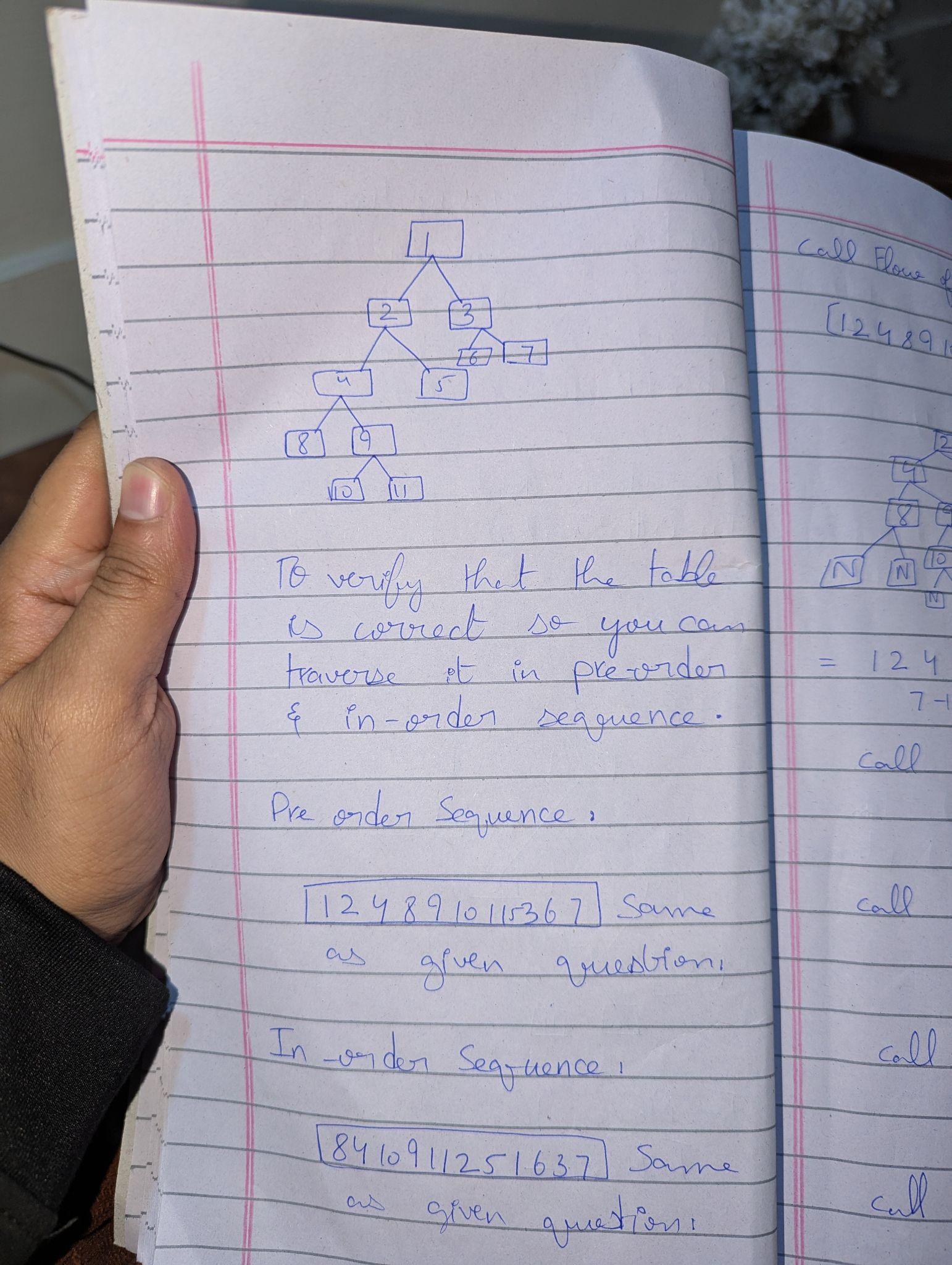
**Method 1**You give only the preorder sequence, but with null markers (-1).  
With this method, you can build the tree and then print preorder, inorder, and postorder all three.  
  
**Method 2** If you don’t want to provide null markers, then you must give two traversals.  
 Examples:

* Preorder + Inorder
* Inorder + Postorder
* Preorder + Postorder

Using any two valid traversals, you can build the complete binary tree without needing null markers.

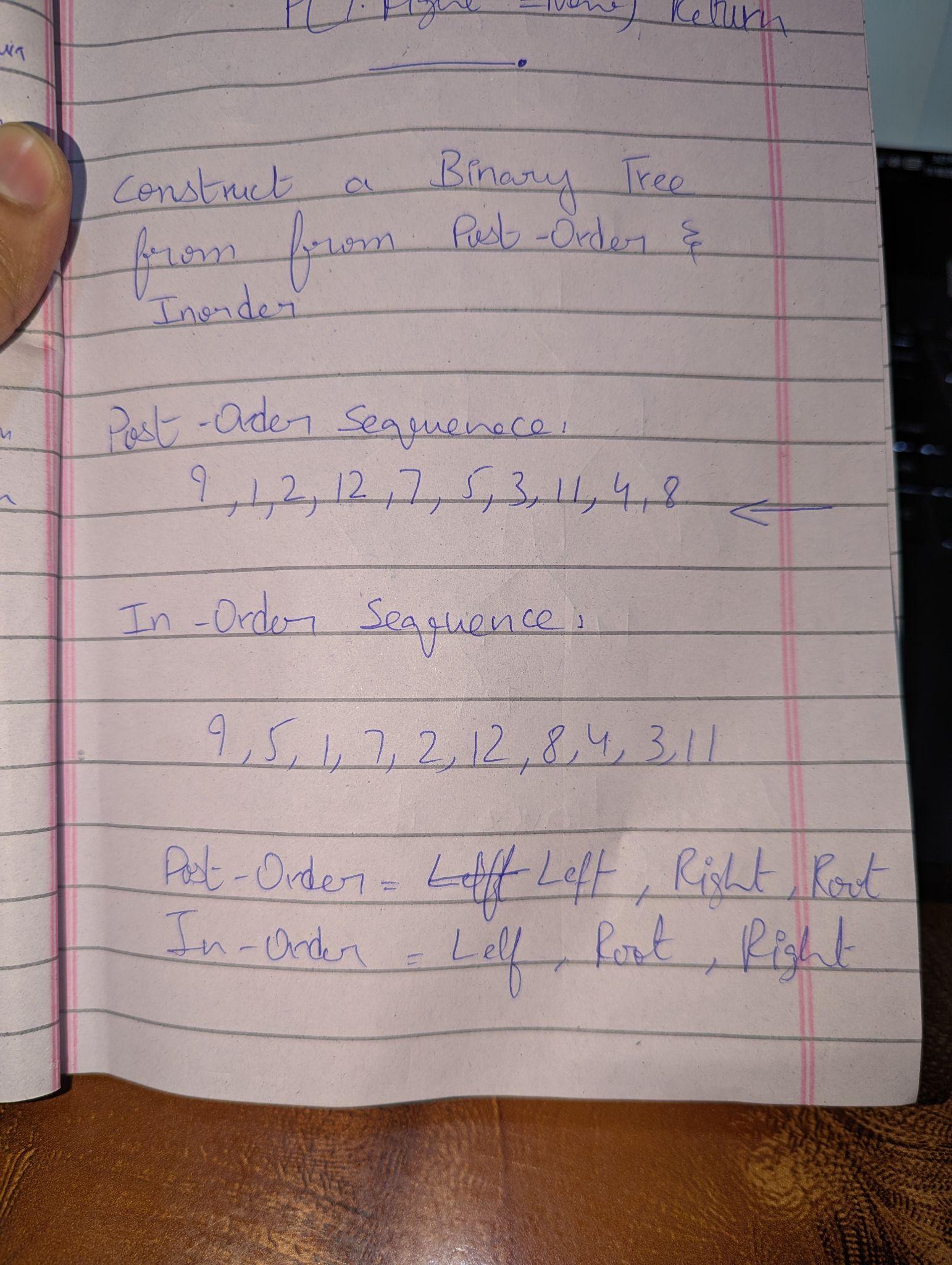
**Preorder + Inorder**

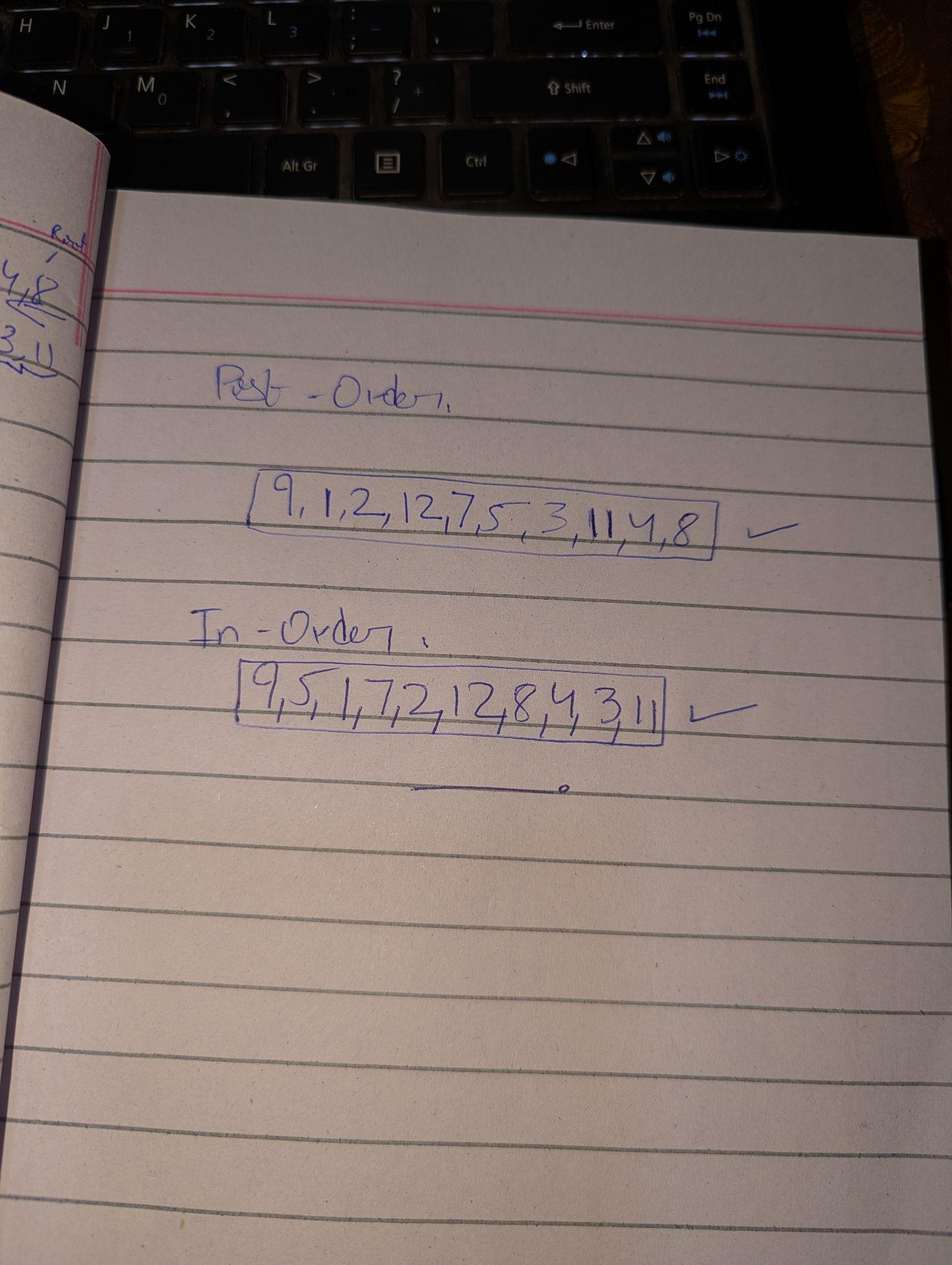
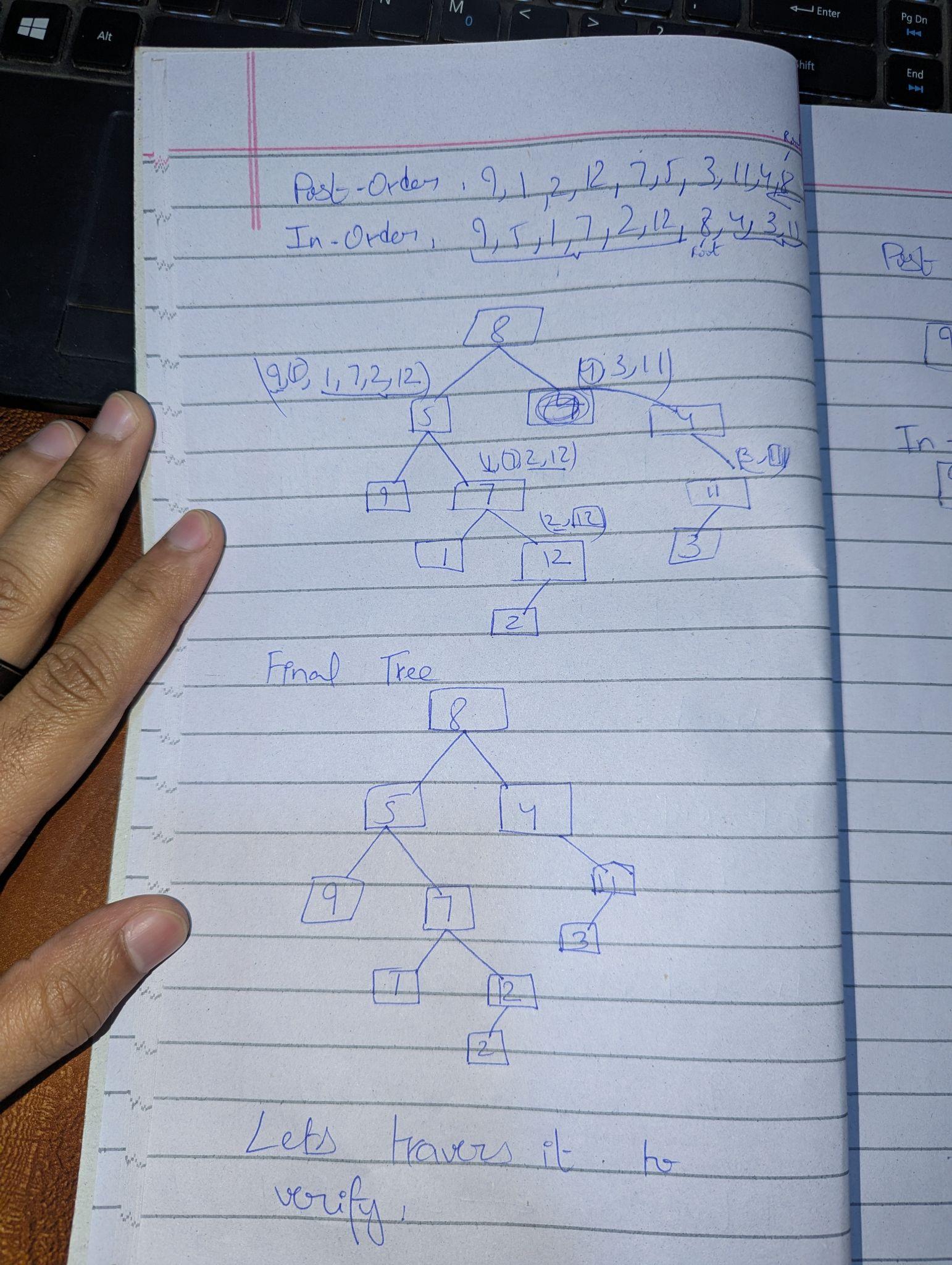
****

****

Python Code:[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/05.Preorder%20%2B%20Inorder.py)

**Postorder + Inorder**

****

****

### 

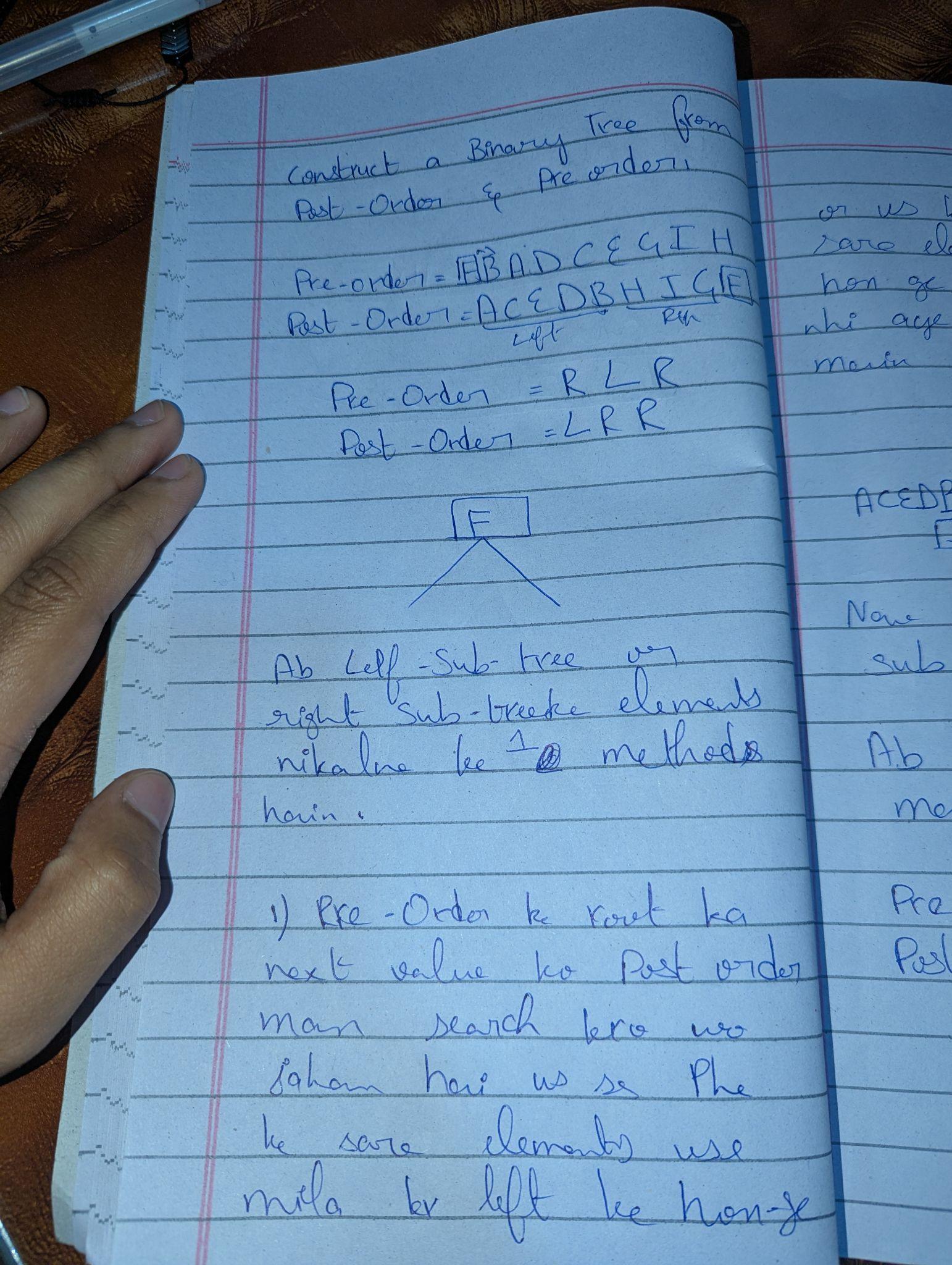
Python Code:[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/06.Postorder%20%2B%20Inorder.py)

### 

### 

### 

**Postorder + Preorder**

****

### 

### **Important Note**

* **Preorder + Inorder**
  + Preorder traversal: Root → Left → Right
  + When building the tree:  
    - Take the **first element** of preorder as root
    - **Build left subtree first**, then right subtree
  + Reason: Preorder lists the root first, then the left subtree, then the right subtree
* **Postorder + Inorder**
  + Postorder traversal: Left → Right → Root
  + When building the tree:  
    - Take the **last element** of postorder as root
    - **Build right subtree first**, then left subtree
  + Reason: Postorder lists the root last, so moving backward we encounter the root first, and the right subtree is immediately before the root in postorder

### **Unique Binary Tree**

* **Definition:** A binary tree is **unique** if a given set of traversal sequences (e.g., preorder + inorder, or inorder + postorder) can reconstruct **exactly one tree**.
* **Example:**
  + Preorder: [1, 2, 3]
  + Inorder: [2, 1, 3]
  + Only **one tree** can be made with these sequences:

1

/ \

2 3

* **Key point:** This tree can have **nodes with 0, 1, or 2 children**.
* Not every pair of sequences guarantees uniqueness.  
  + Example: Preorder + Postorder **alone** may not give a unique tree unless some extra rules are given.

### **Unique Full Binary Tree**

* **Definition:** A **full binary tree** is a tree in which **every node has either 0 or 2 children**.
* A unique full binary tree is a full binary tree that can be **uniquely reconstructed** from a pair of traversal sequences (usually preorder + postorder).
* **Example:**

Preorder: [1, 2, 4, 5, 3, 6, 7]  
 Postorder: [4, 5, 2, 6, 7, 3, 1]

* Only one **full binary tree** satisfies these sequences:

1

/ \

2 3

/ \ / \

4 5 6 7

* **Key point:**
  + The “full” property (each node has 0 or 2 children) removes ambiguity that occurs in general binary trees.
  + Without this property, preorder + postorder alone may give **multiple possible trees**.

Binary Search Tree

# **Binary Search Tree (BST)**

A **Binary Search Tree** is a special type of **binary tree** that makes searching very efficient.

## **BST Rules:**

1. The **left child** of a node contains a value **less than** the node.
2. The **right child** of a node contains a value **greater than** the node.
3. These rules apply **recursively** to all nodes in the tree.

# **Why BST (Binary Search Tree) is Useful**

1. **Efficient Searching:**
   * Searching for a value in a BST is very fast because we can ignore half of the tree at each step.
2. **Fast Insertion:**
   * Inserting a new value is efficient because we follow the same rules as searching to find the correct position.
3. **Fast Deletion:**
   * Deleting a value is also efficient, as we can locate the node quickly using the BST property.
4. **Ordered Structure:**
   * Inorder traversal of a BST gives values in **sorted order**, which is useful for many applications.

**Simple Structure of BST**

8

/ \

3 10

/ \ \

1 6 14

/ \ /

4 7 13

**Explanation:**

* Each node has at most 2 children: left and right.
* Left child values are less than the parent node.
* Right child values are greater than the parent node.

# **Deletion in BST – 3 Cases**

1. **Node with No Child (Leaf Node):**
   * If the node to be deleted has **no children**, it can be deleted **directly**.
2. **Node with One Child:**
   * If the node to be deleted has **one child**, delete the node and **replace it with its child**.
3. **Node with Two Children:**
   * If the node to be deleted has **two children**, there are **two options** to replace it:  
     1. **Inorder Predecessor:** Replace with the **largest node in the left subtree**.
     2. **Inorder Successor:** Replace with the **smallest node in the right subtree**.

**BST - Preorder:**

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/07.BST%20%2B%20Preorder.py)

**BST - Inorder:**

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/08.BST%20%2B%20Inorder.py)

**BST - Postorder:**

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/09.BST%20%2B%20Postorder.py)

**More Code Practice**

1. This code builds a binary tree from a list (using −1 as null) and prints its preorder, inorder, and postorder traversals.

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/Binary%20Trees%20Practice/main.py)

1. This code builds a binary tree from given preorder and inorder lists and prints its preorder, inorder, and postorder traversals.

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/Binary%20Trees%20Practice/main2.py)

1. This code builds a binary tree from given postorder and inorder lists and prints its preorder, inorder, and postorder traversals.

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/Binary%20Trees%20Practice/main3.py)

1. This code implements a Binary Search Tree (BST) with insertion & deletion and prints its preorder, inorder, and postorder traversals.

Python code[**Click Here**](https://github.com/Faaiz07-cloud/Trees-DSA/blob/master/Binary%20Trees%20Practice/main4.py)

AVL Tree

# **AVL Tree – Advanced Version of BST**

## **Introduction**

An **AVL Tree** is a **special type of Binary Search Tree (BST)**. It is designed to overcome some of the limitations of a standard BST.

## **Why AVL Tree is Needed**

In a regular BST, certain operations can become inefficient in the worst case:

* **Insertion:** O(n)O(n)O(n) in worst case
* **Search:** O(n)O(n)O(n) in worst case
* **Deletion:** O(n)O(n)O(n) in worst case

This happens when the BST becomes **skewed**, i.e., all nodes are on one side.

**AVL Trees solve this problem by keeping the tree balanced.**

## **Key Feature**

* **AVL Tree maintains balance**, ensuring that the height difference between left and right subtrees of any node is at most 1.
* This balance ensures that operations such as insertion, deletion, and search remain **efficient**.

## **Traversal for Learning Operations**

Before learning **Insert, Search, and Delete** in AVL trees, it is important to understand **tree traversal**:

* Traversal allows us to visit each node of the tree systematically.
* Typical traversal orders:  
  + **Inorder**
  + **Preorder**
  + **Postorder**

### **Time Complexity**

Even with traversal, the **worst-case time complexity** in a skewed BST is:

* **O(n)**

AVL trees **improve this** by maintaining balance, reducing the height, and thus making operations **O(log n)** in most cases.

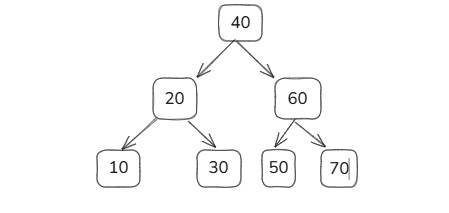
# **Limitations of Binary Search Tree (BST)**

* BSTs are efficient **only if balanced**.
* **Problem:** If a BST becomes skewed (like a linked list), operations become **slow**:  
  + **Search:** O(n)
  + **Insertion:** O(n)
  + **Deletion:** O(n)
* This is the main limitation of regular BSTs.

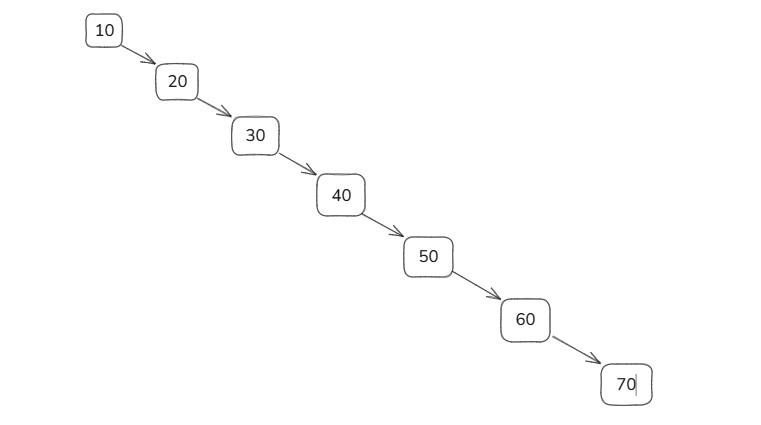
# **Here AVL comes**

* **AVL Tree** is a type of **self-balancing BST**.
* **Self-balancing property:**
  + The tree **automatically balances itself** after every insertion or deletion.
  + Ensures the **height difference** between left and right subtrees of any node is **at most 1**.

**Lets Suppose**Tree 1



Tree 2



* There are 2 BST - Binary Search Tree

# **Insertion in AVL Tree vs Skewed BST**

## **1. Insertion in a Skewed BST**

* Suppose we have a skewed BST (all nodes growing to one side).
* When we insert a new node, e.g., **80**:  
  + The tree starts from the **root** and **checks every node** sequentially.
  + It traverses the **entire tree** until it reaches the correct position.
  + In the worst case, if the tree has nnn nodes, insertion takes **O(n)** time.
* **Problem:** Skewed growth makes the tree inefficient for large datasets.

## **2. Insertion in a Balanced AVL Tree**

* Suppose we have a **well-balanced AVL tree**.
* When inserting **80**:  
  + The tree **does not need to traverse all nodes**.
  + Only a **few comparisons** (logarithmic with respect to height) are needed.
  + The tree **automatically balances itself** if needed using rotations.
* **Result:** Insertion time is **O(log n)**, much faster than skewed BST.

**Conceptual Diagrams**

**Skewed BST**

10

\

20

\

30

\

40

\

50

\

60

\

70

\

80

**Balanced AVL**

40

/ \

20 60

/ \ / \

10 30 50 70

\

80

# **Self-Balancing in AVL Trees**

## **1. What is Self-Balancing?**

* AVL trees are **self-balancing binary search trees**.
* **Goal:** Keep the tree **balanced** so that the height difference between the **left** and **right** subtrees of every node is **at most 1**.

## **2. Balance Factor**

* Each node in an AVL tree has a **balance factor**:

Balance Factor = Height of Left Subtree − Height of Right Subtree

**Interpretation:**

* + **-1, 0, 1** → Node is **balanced**
  + **< -1 or > 1** → Node is **unbalanced** and requires rotation

## **3. How AVL Maintains Balance**

* When a node is **inserted or deleted**, AVL tree checks **all ancestors** up to the root:  
  1. Calculate the **balance factor** of each node.
  2. If the balance factor becomes **less than -1 or greater than 1**, the tree is **unbalanced**.
  3. **Rotations** are applied to restore balance:  
     + Left Rotation
     + Right Rotation
     + Left-Right Rotation
     + Right-Left Rotation

# **AVL Tree Example: Checking Balance**

## **1. Initial Data**

We have the nodes:

10, 20, 30

* Suppose we insert them in order: 10 → 20 → 30.

## **2. Creating a Simple BST**

In a regular BST, inserting in order gives:

10

\

20

\

30

* **Observation:** All nodes are growing to the right.

## **3. Checking AVL Property**

* **Balance Factor (BF)** = Height of Left Subtree − Height of Right Subtree

### **Root Node (10)**

* Left Subtree Height = 0
* Right Subtree Height = 2
* BF = 0 − 2 = **−2** → **Not balanced**

✅ Conclusion: **This is not an AVL tree.**

## **4. Balancing the Tree**

* To make it AVL, we perform **rotations**.
* **Right-heavy imbalance at node 10** → Perform **Left Rotation**:

20

/ \

10 30

## **5. Checking Balance Again**

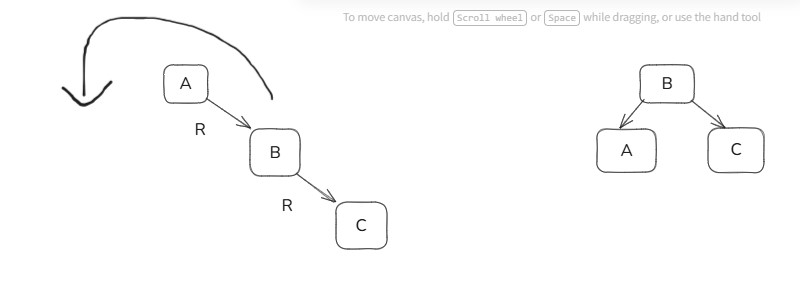
* **Root Node (20):**
  + Left Subtree Height = 1
  + Right Subtree Height = 1
  + BF = 1 − 1 = 0 → Balanced
* **Other Nodes (10 and 30):** Leaf nodes → BF = 0 → Balanced

✅ Conclusion: **Now it is a proper AVL tree.**

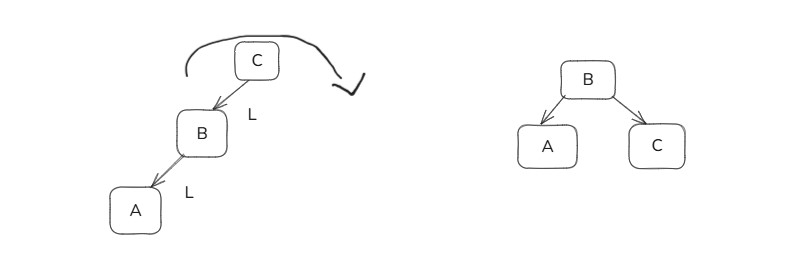
## **6. Key Points**

1. AVL trees must maintain **balance factor** −1, 0, or 1 at every node.
2. After insertion, check the **balance factor** from inserted node up to root.
3. If **BF < −1 or > 1**, apply **rotations** to restore balance.
4. The resulting tree is **height-balanced** and ensures **O(log n) operations**.

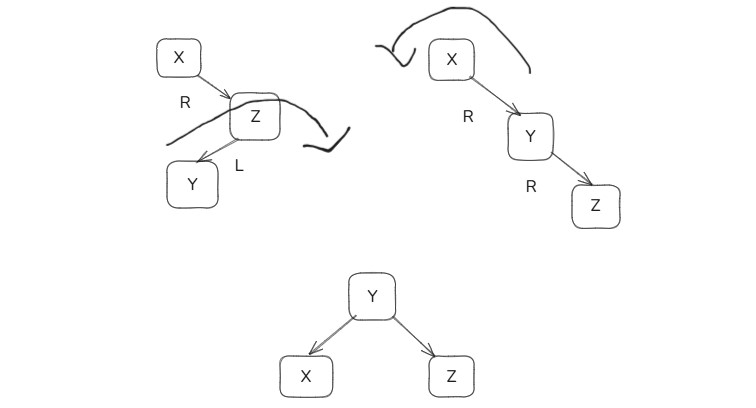
Types of Rotation  
  
1. Left Rotation – RR



2. Right Rotation – LL



3. RL Rotation (Right-Left Rotation)



4. LR Rotation (Left-Right Rotation)

