Data Structures

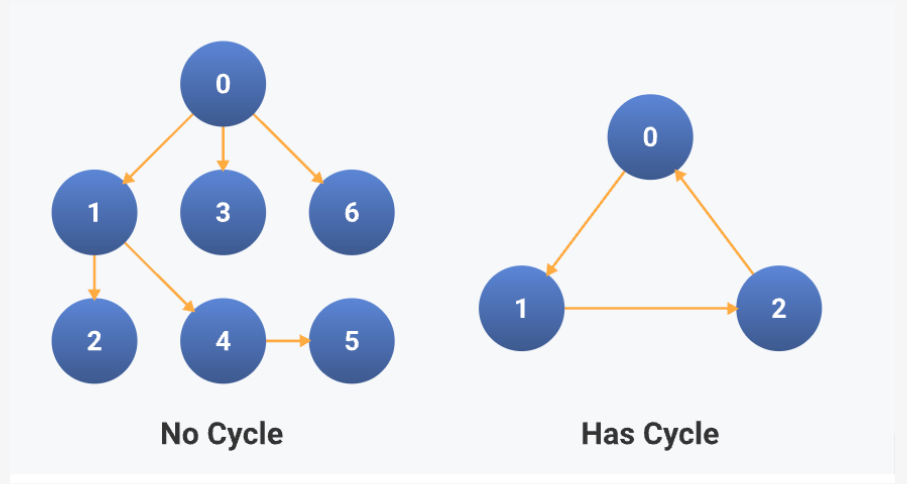
Sec 7 : Tree & BST

By: Eng.Rahma Osama Eng.Sandra Sameh

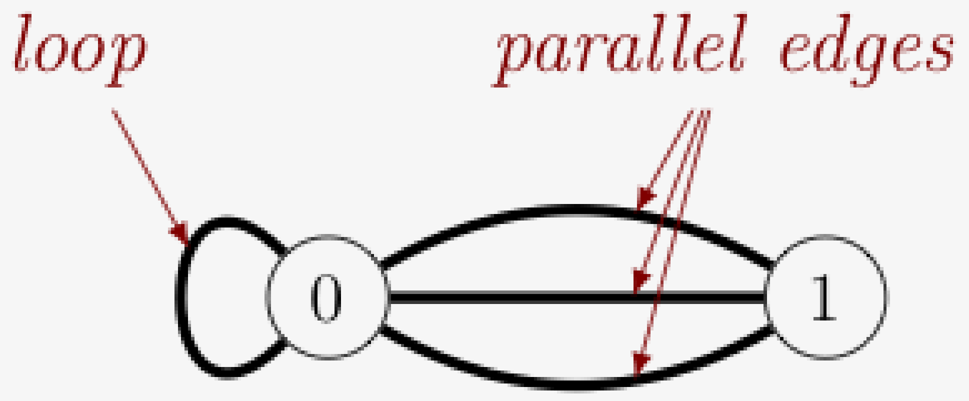
DEFINITION

Tree is : Connected & Undirected Graph with:

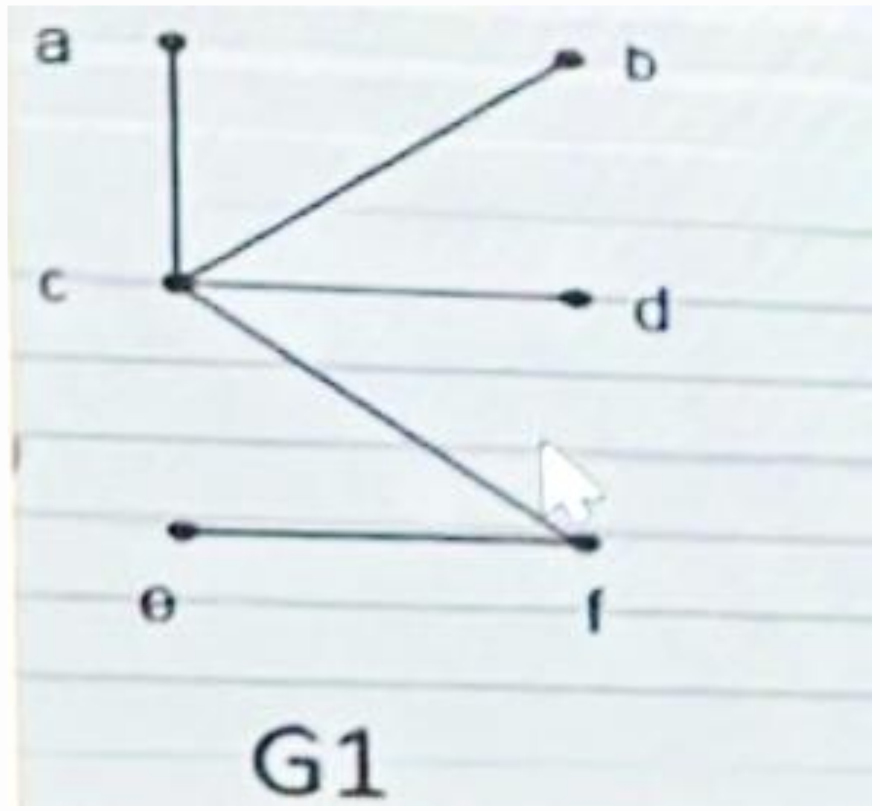
- No simple circuits.



- No Multiple Edges.
- No Loops.

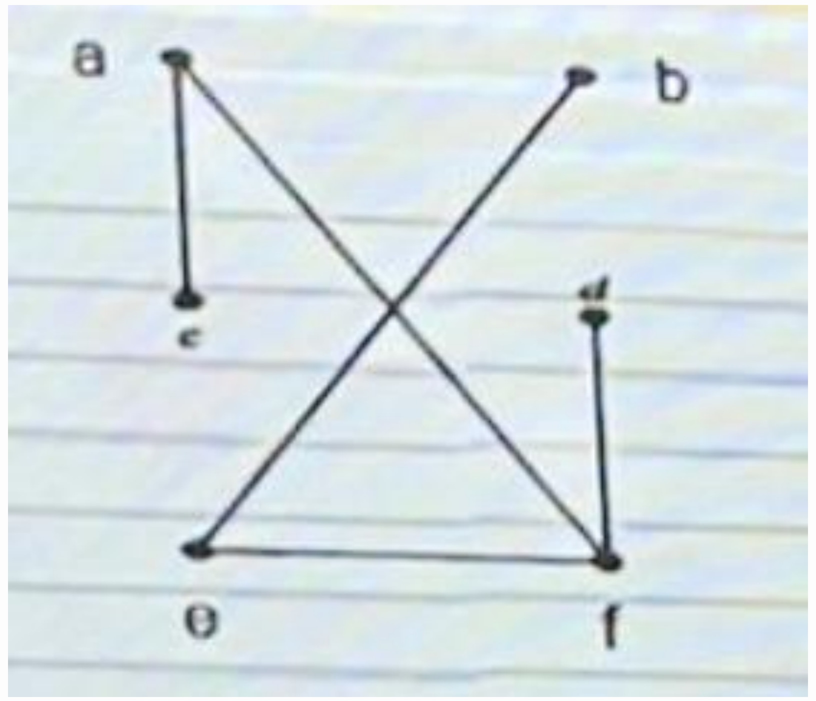


Tree Or Not?



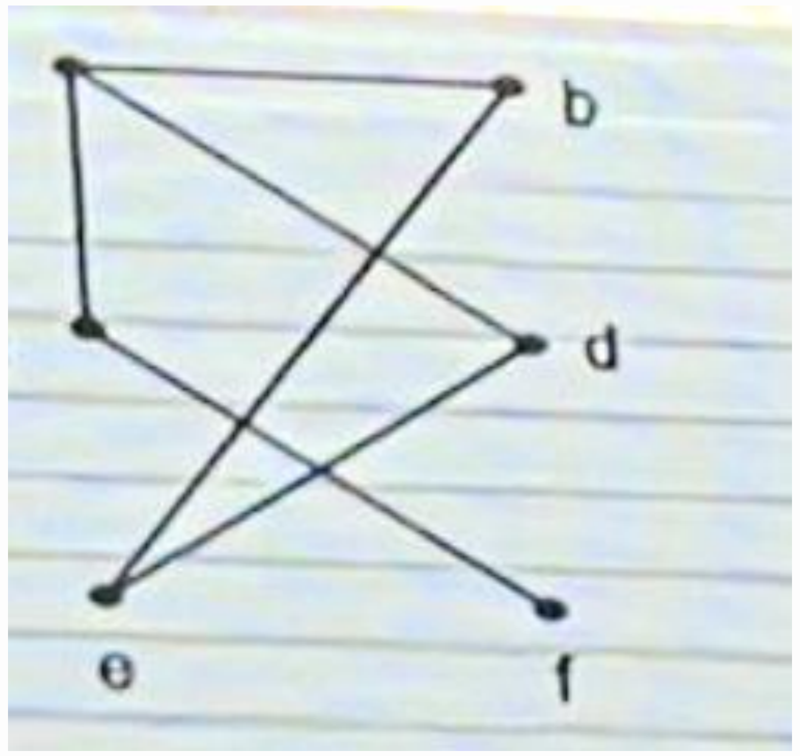
- Connected ✓
- Undirected ✓
- No simple circuits. ✓
- No Multiple Edges. ✓
- No loops ✓

Tree Or Not?



- Connected ✓
- Undirected ✓
- No simple circuits. ✓
- No Multiple Edges. ✓
- No loops ✓

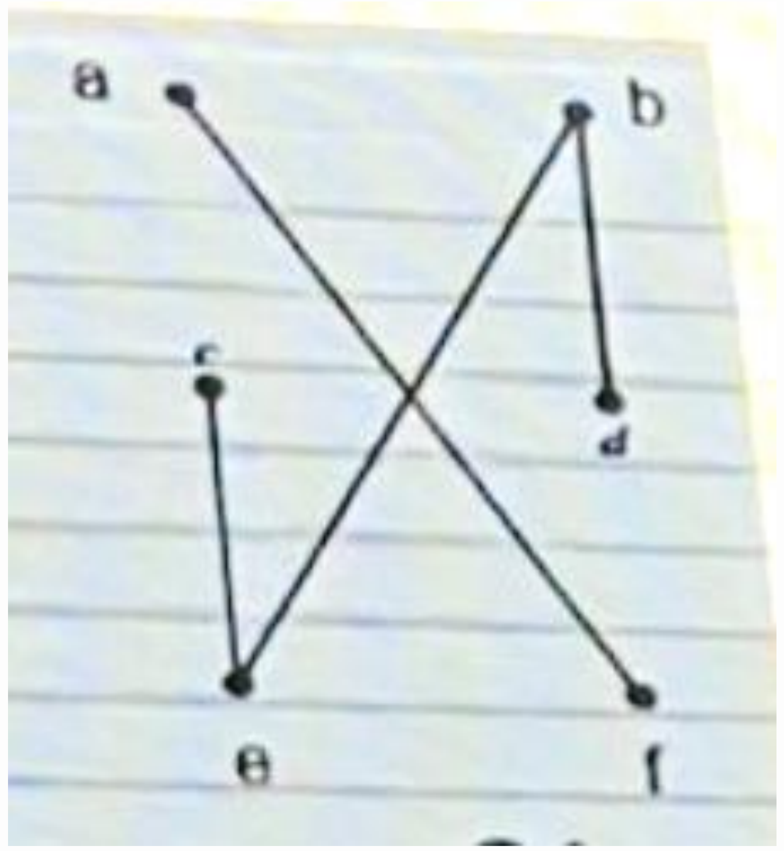
Tree Or Not?



- Connected ✓
- Undirected ✓
- No simple circuits. ✗

Not Tree.

Tree Or Not?

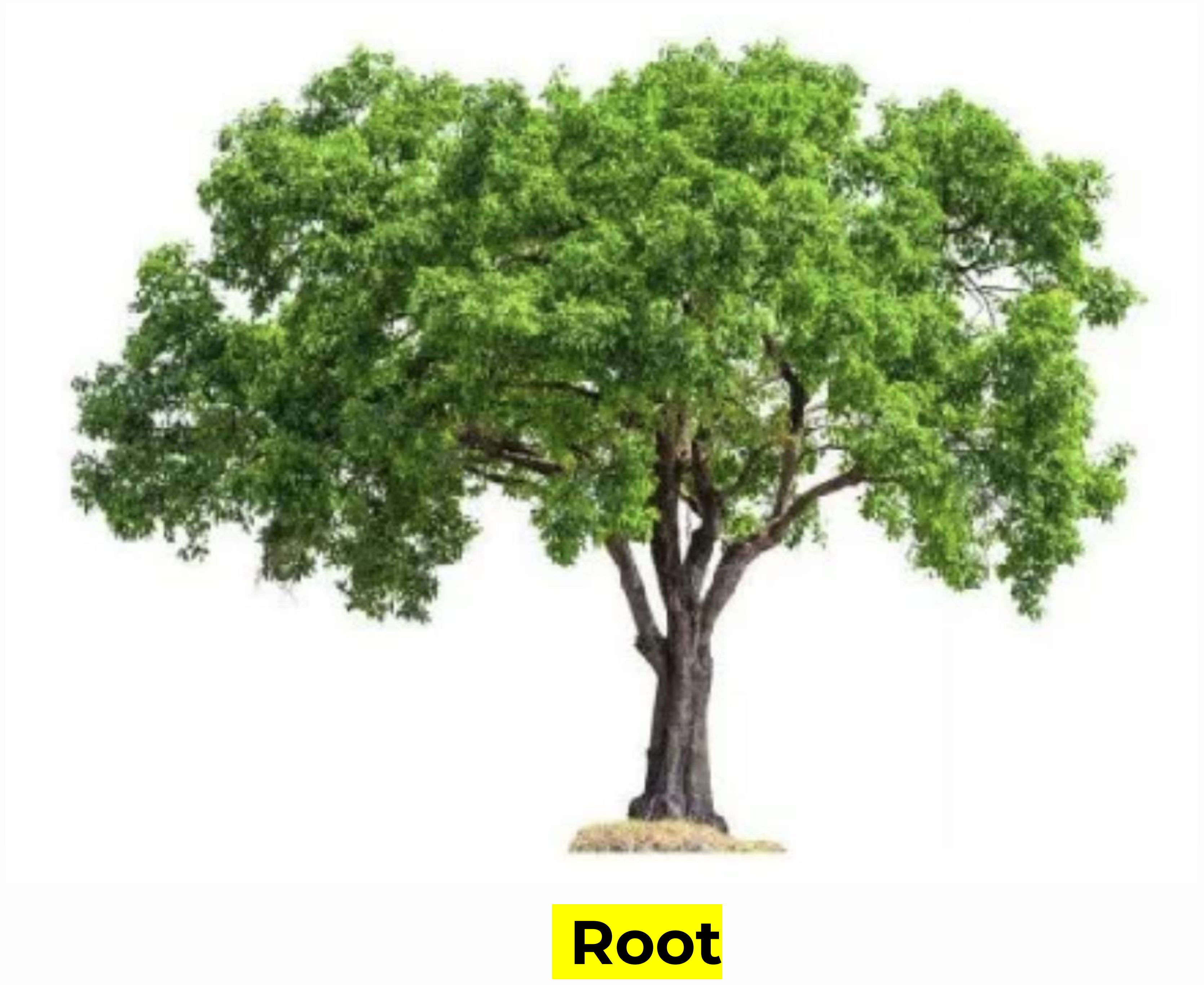


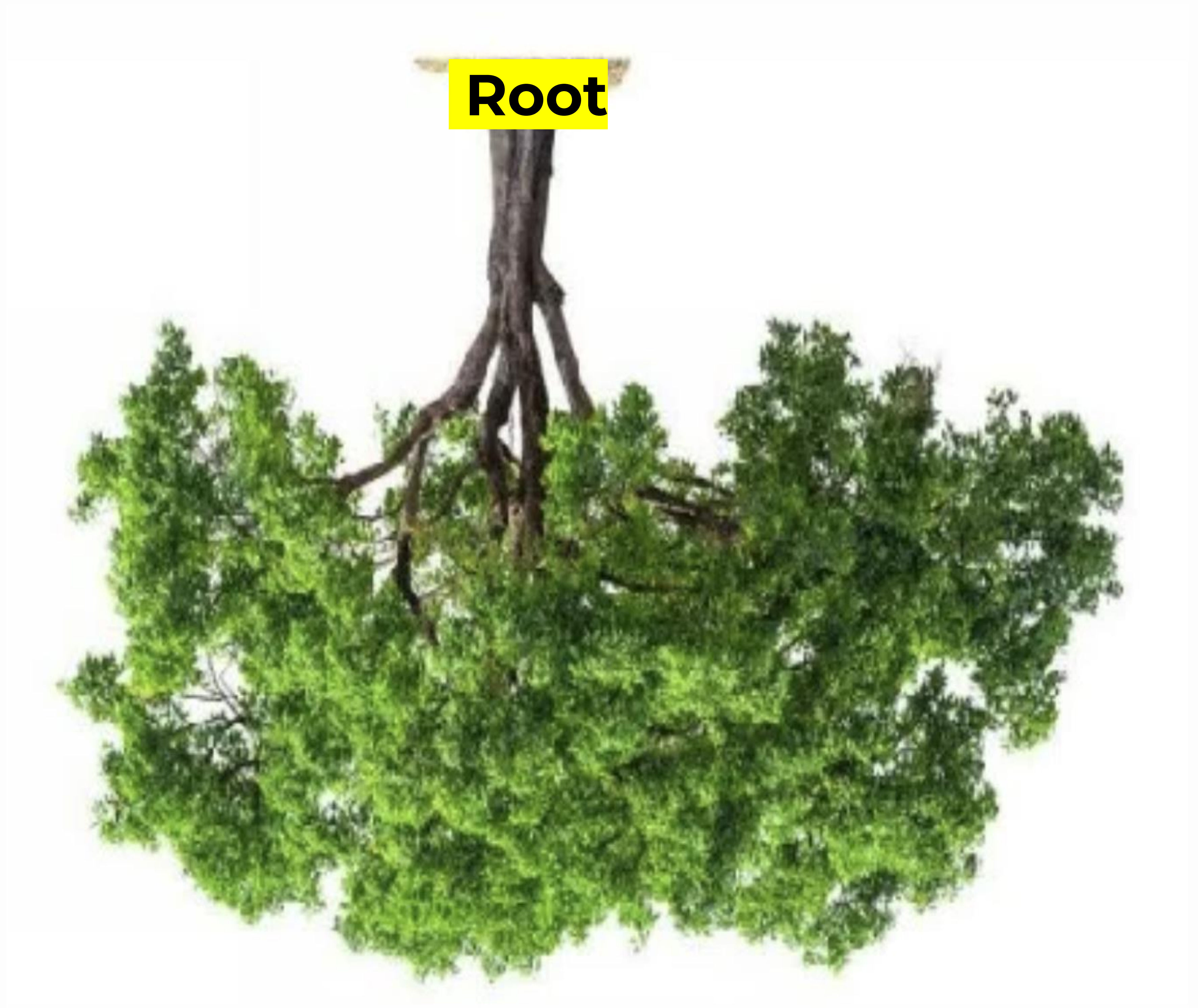
- Connected ✗

Not Tree.

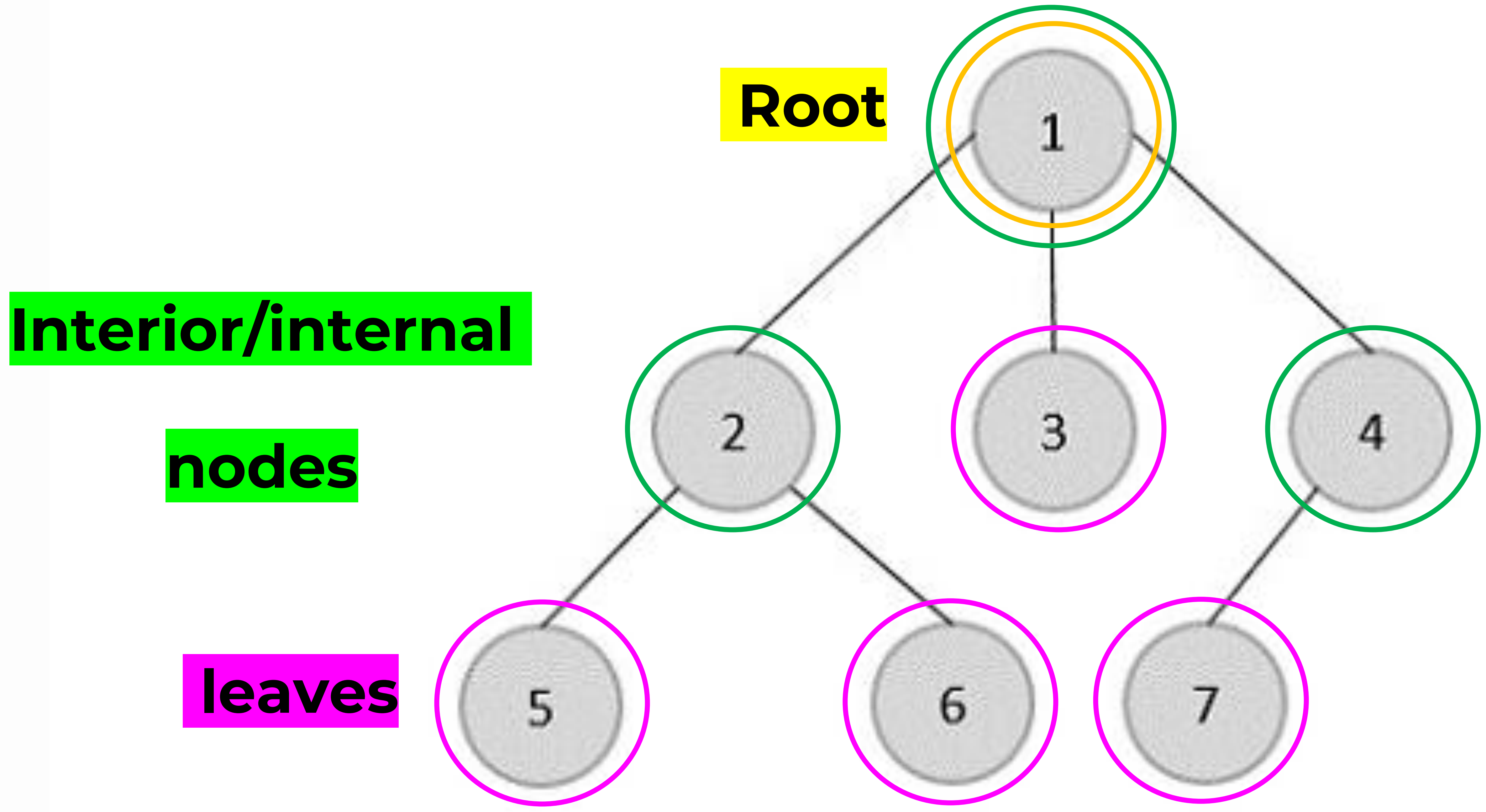
Trees in real life

Trees in computer science

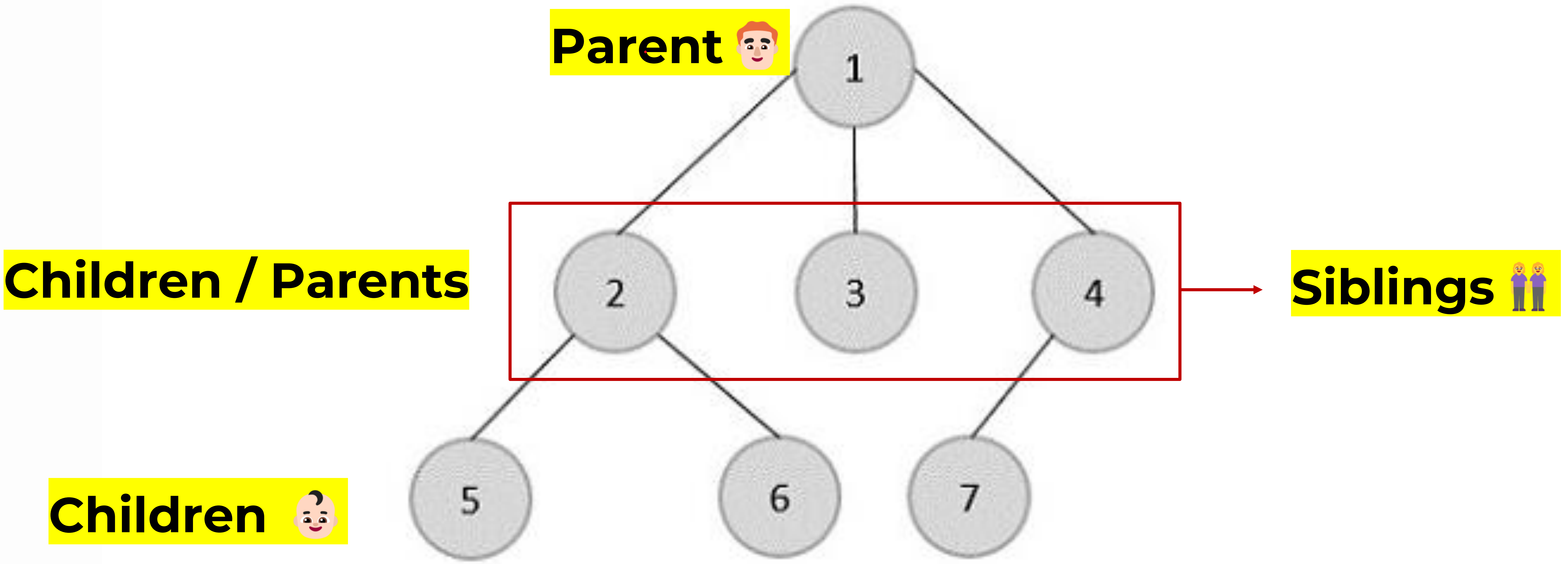




Tree Family



Tree Family



- 2 is parent of 5&6
- 5 is child of 2

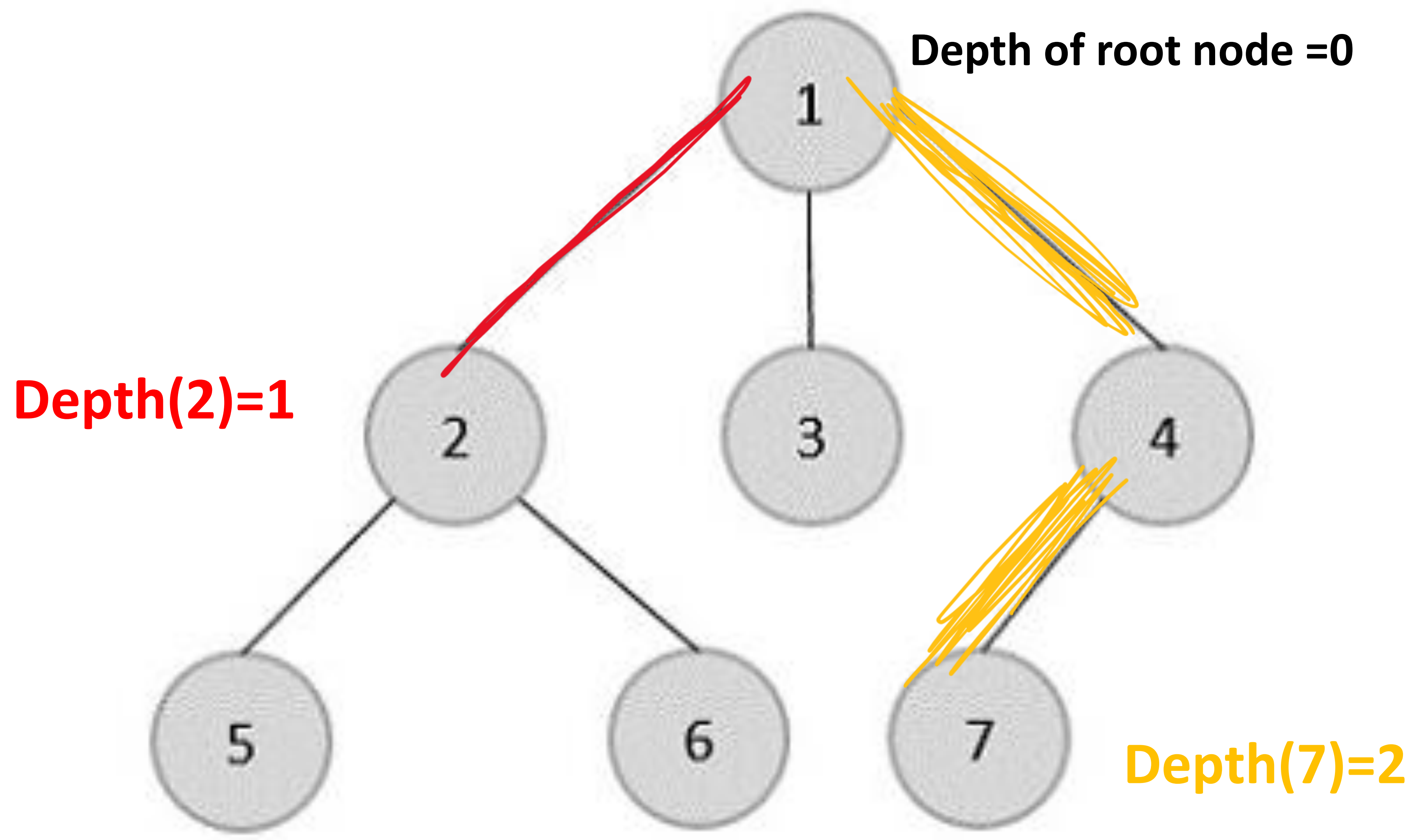
Ancestors 👴 👵 **: of 5 are 1 ,2**

descendants 👦 👦 **: of 1 are 2,3,4,5,6,7**

Tree Structure

🌳 **Depth of a Tree (Definition):**

The maximum number of edges from the root node to the farthest leaf node.

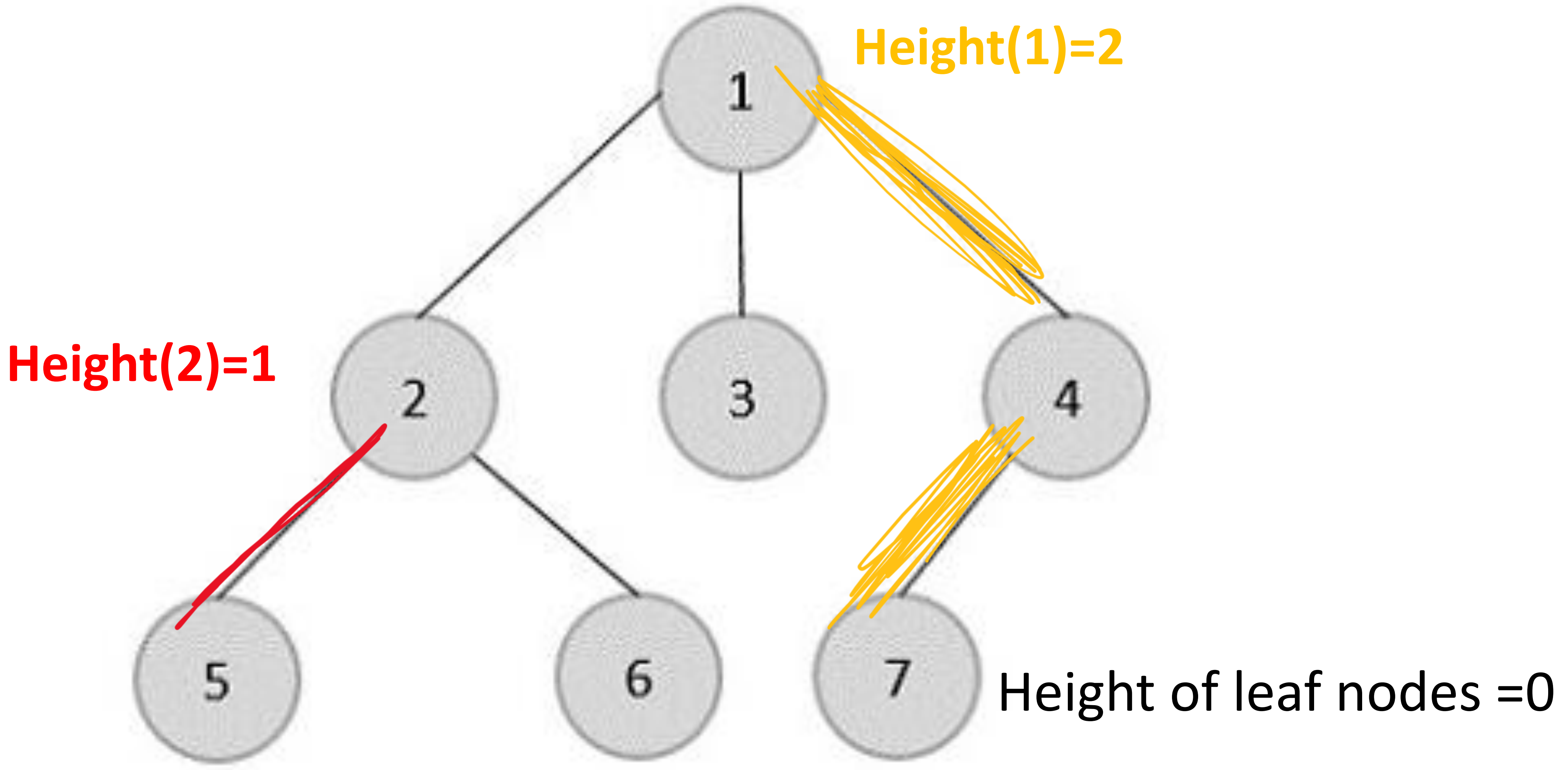


<u>So , the depth of this tree is 2</u>

Tree Structure

🌳 **Height of a Tree (Definition):**

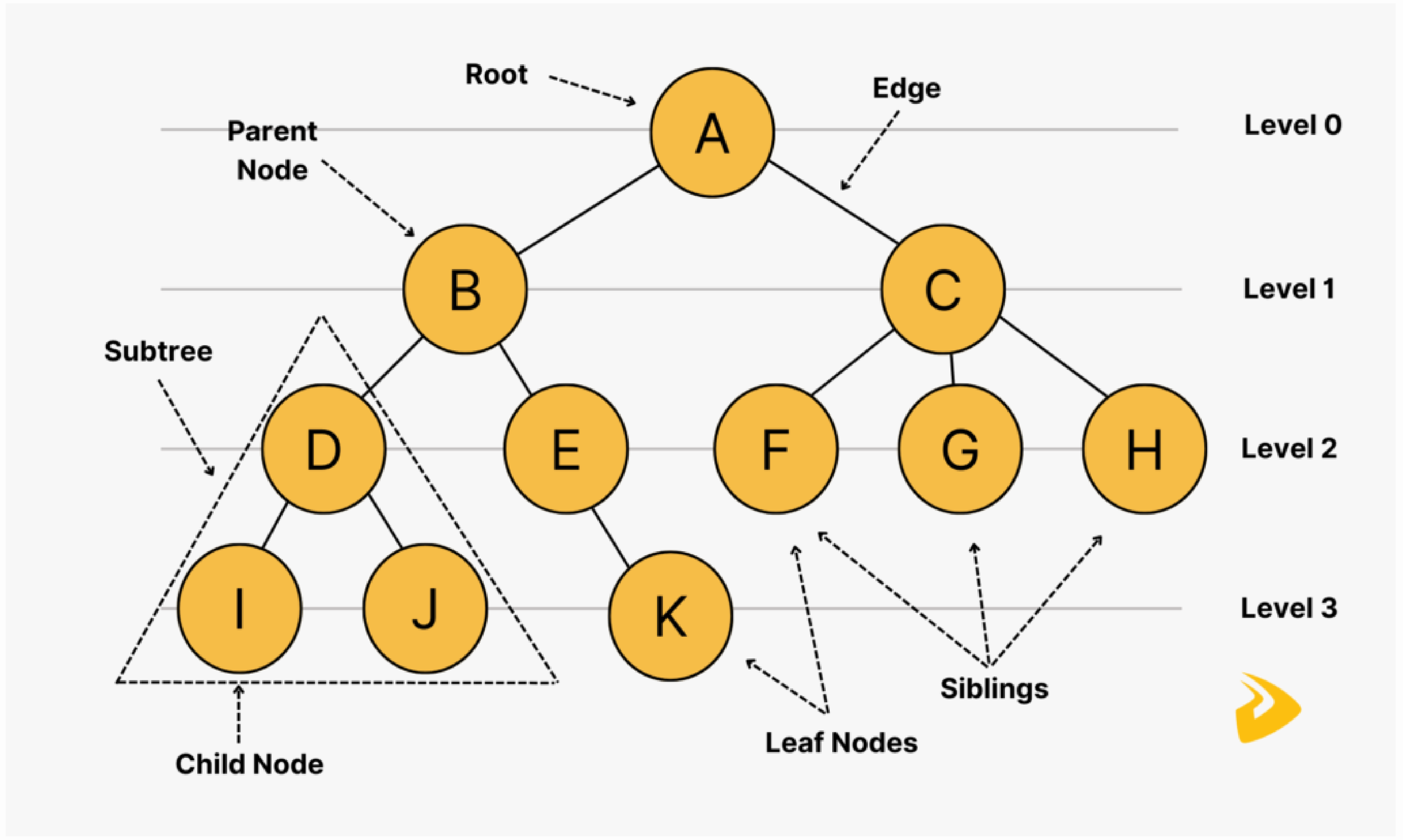
The height of a tree is the maximum number of edges from any leaf node up to the root.



<u>So , the height of this tree is 2</u>

Tree Structure

- 🌳 **Size of a Tree:** Total number of nodes. **-> 11**
- 🌳 The **width of a tree** is the **maximum number of nodes present at any single level** (depth) of the tree. -> 5
- 🌳 **n.of edges =** n.of nodes – 1**.**
- 🌳 **n.of nodes** = I + L . (internal + leaves)
- 🌳 A **rooted tree** is called **m-ary** tree if every internal vertex has **no more than m** children.
- 🌳 A **tree** is called **full m-ary** tree if every internal vertex has **exactly m** children.
- 🌳 **m-ary tree with m=2** is called **binary** tree.
- 🌳 **n.of nodes** for **full m-ary** tree with **i** internal nodes **= mi+1.**

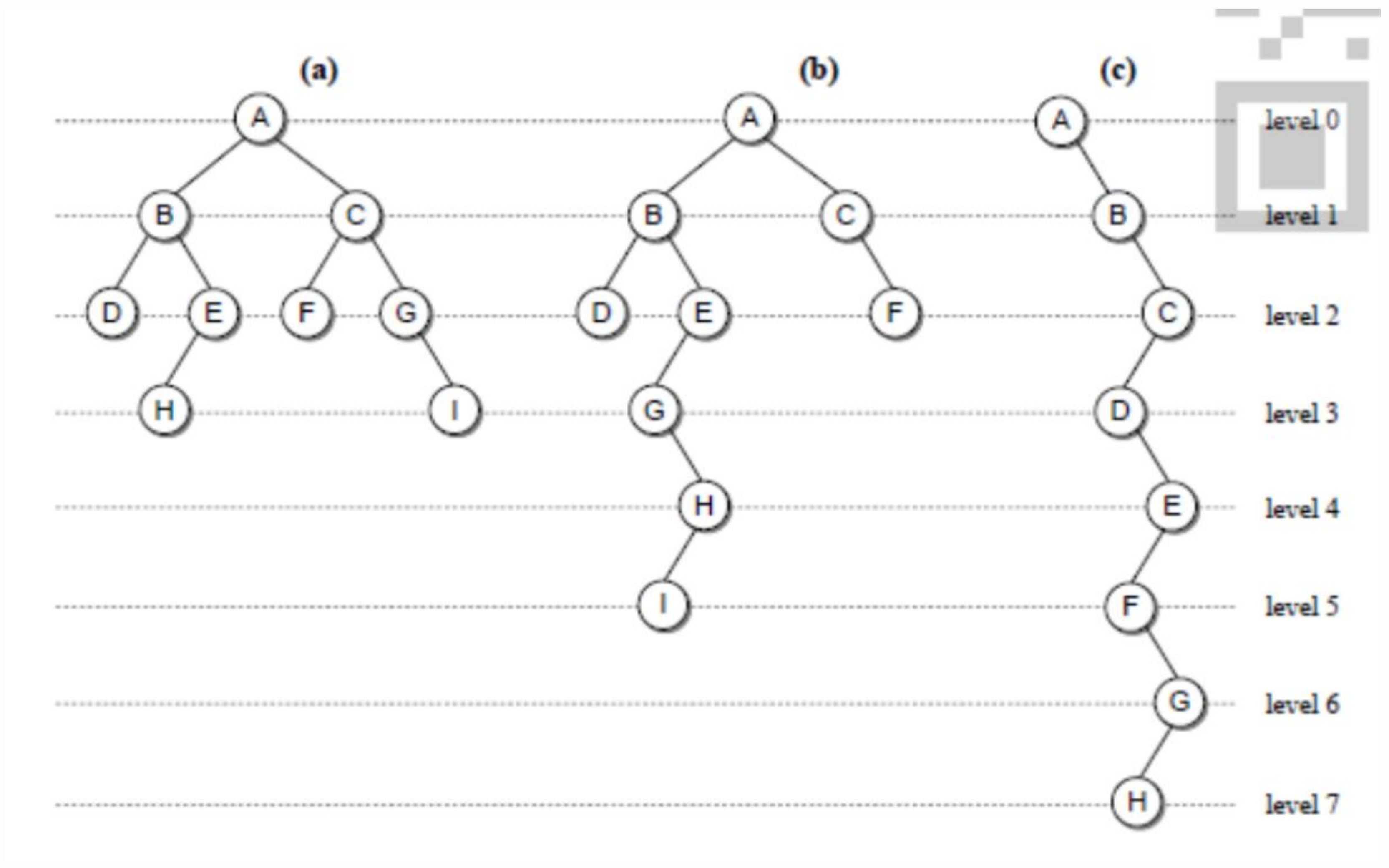


Binary Tree

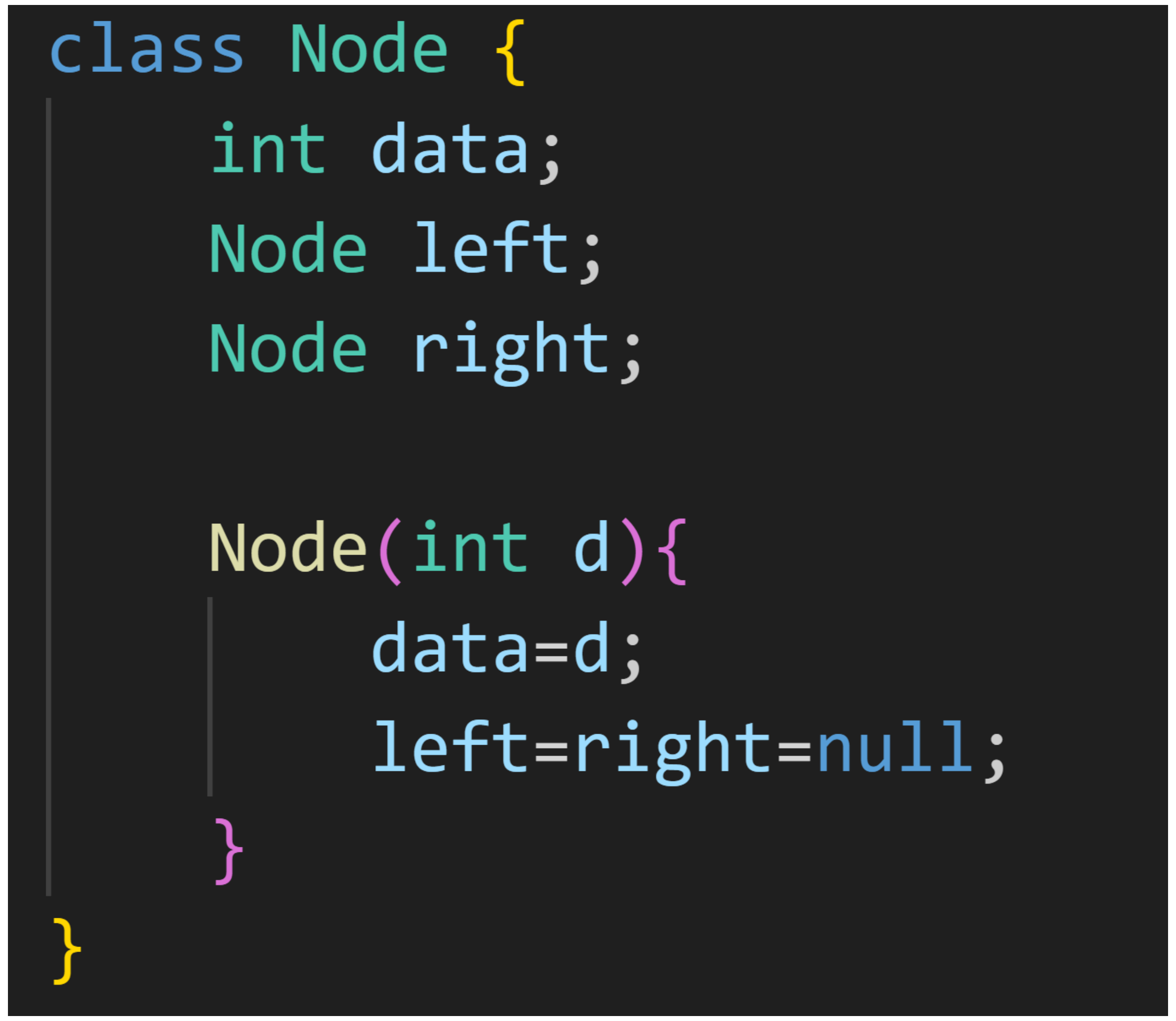
- A binary tree is composed of **zero or more nodes**.

- A binary tree may be empty (contain no nodes).
- If not empty, a binary tree has **a root node**.

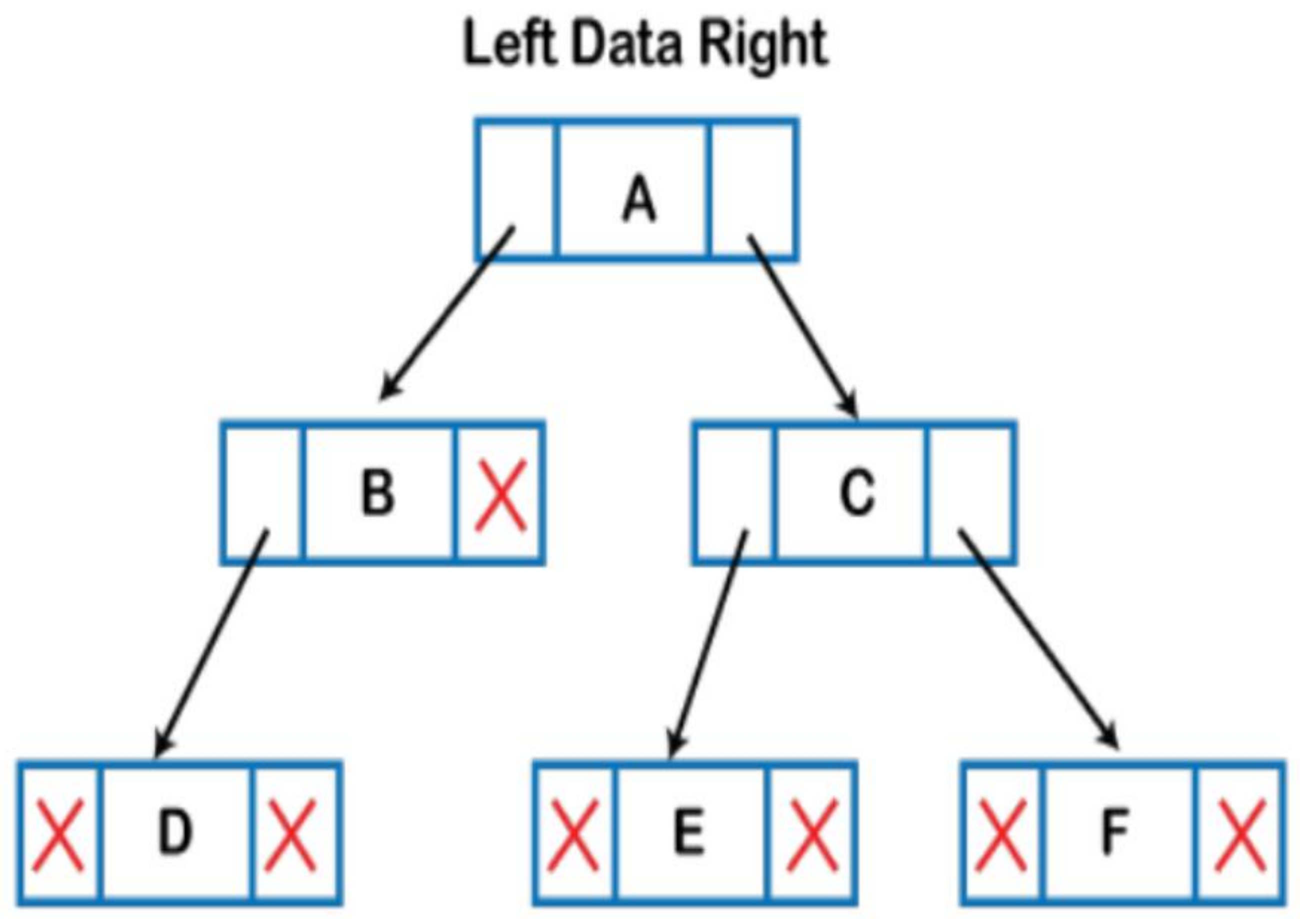
- Every node in the binary tree is reachable from the root node by **a unique path**.
- A node with no left child and no right child is called a **leaf**.



- **Each node contains**:
  * A **value** (some sort of data item)
  * A reference or pointer to a **left child** (may be null),
  * A reference or pointer to a **right child** (may be null).

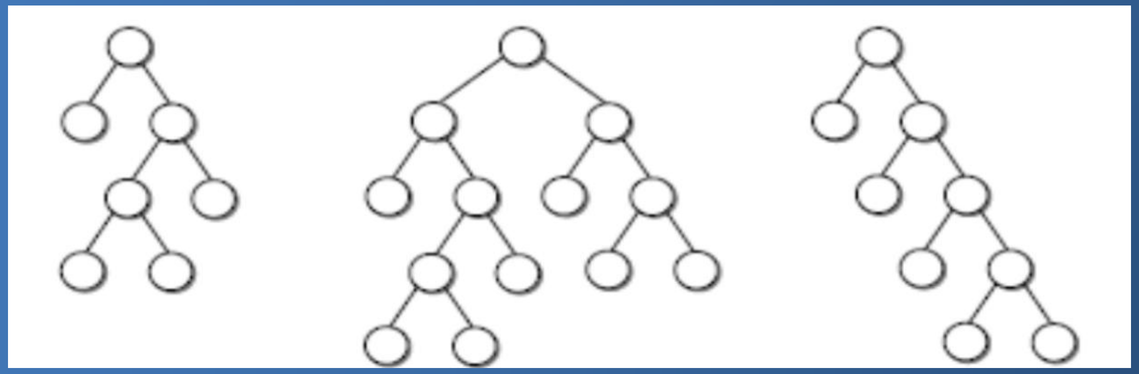


```
class Node {
    int data;
    Node left;
    Node right;

    Node(int d){
        data=d;
        left=right=null;
    }
}
```

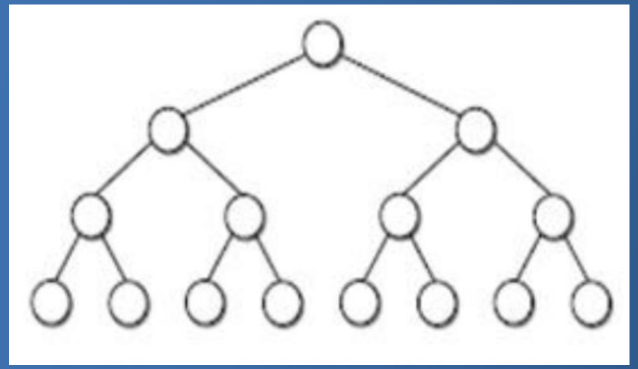


Binary tree

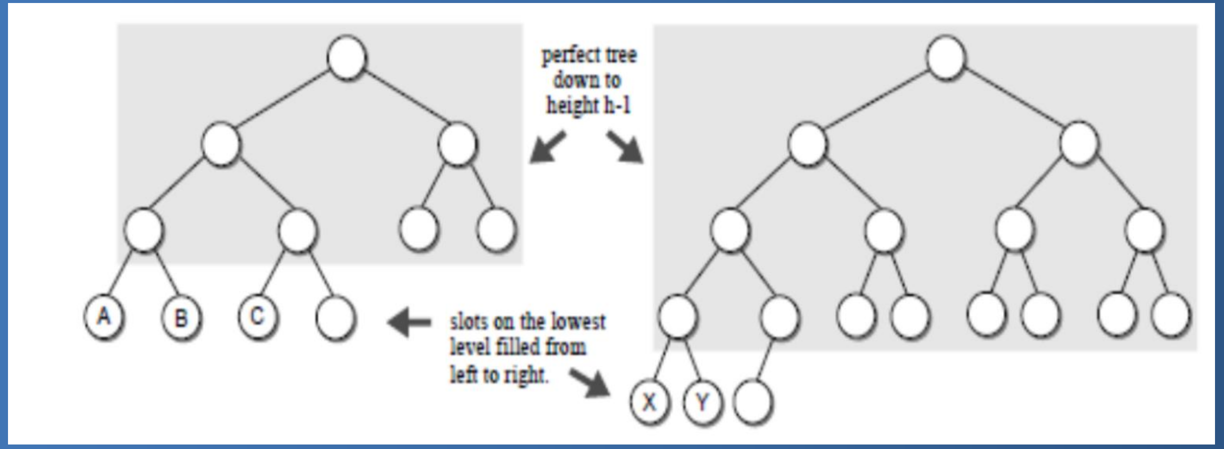
A **full binary tree** is a binary tree in which each interior node contains two children.



- A **perfect binary tree** is a full binary tree in which all leaf nodes are at the same level.
- The perfect tree has all possible node slots **filled** from top to bottom with no gaps,

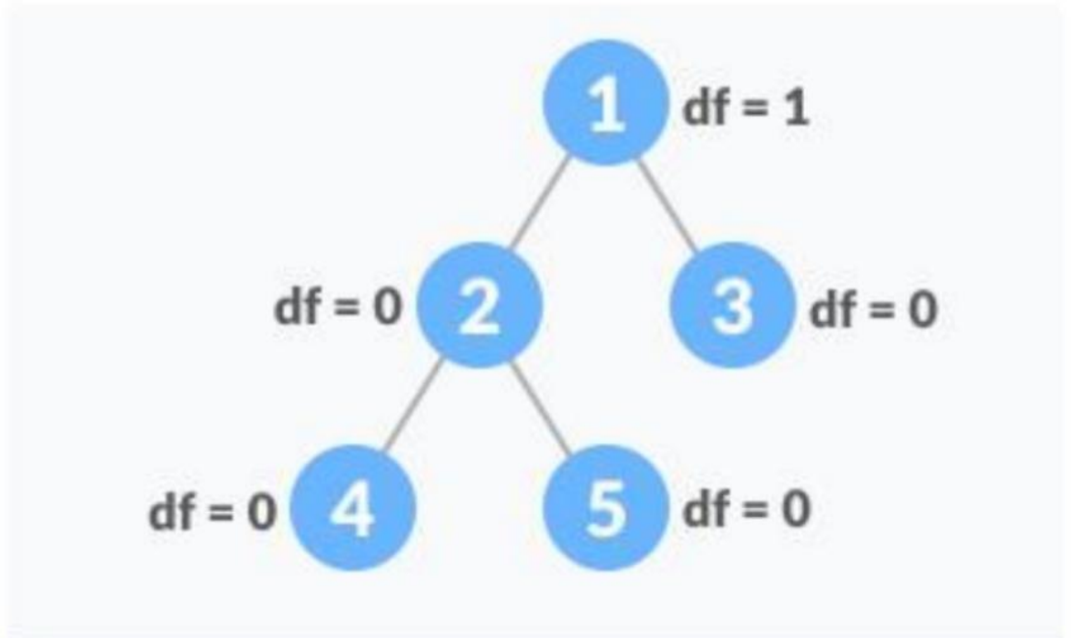


- A **complete binary tree** is a perfect binary tree down to height h-1 and the nodes on the lowest level fill the available slots **from left to right leaving no gaps**.

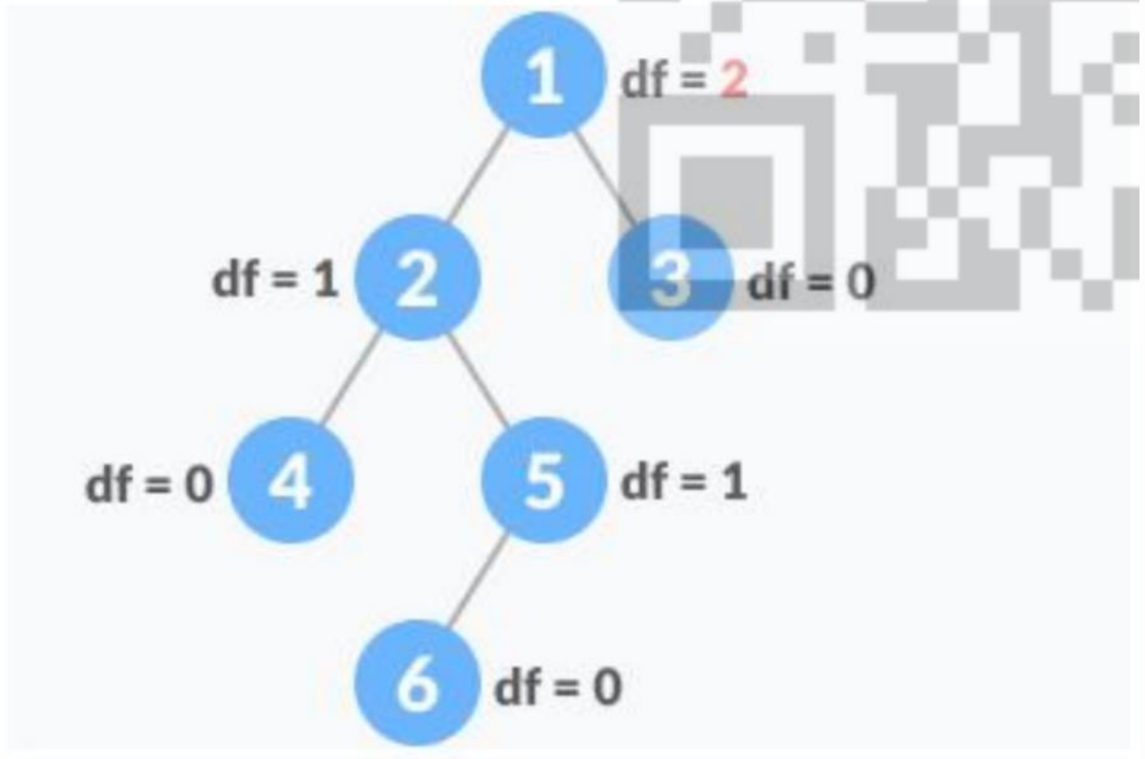

perfect tree down to height h-1
A
B
C
slots on the lowest level filled from left to right.
X
Y

Balanced Tree

🌳 **Balance factor =** height(left subtree) - height(right subtree) .



Balanced Binary Tree with depth at each level



Unbalanced Binary Tree with depth at each level

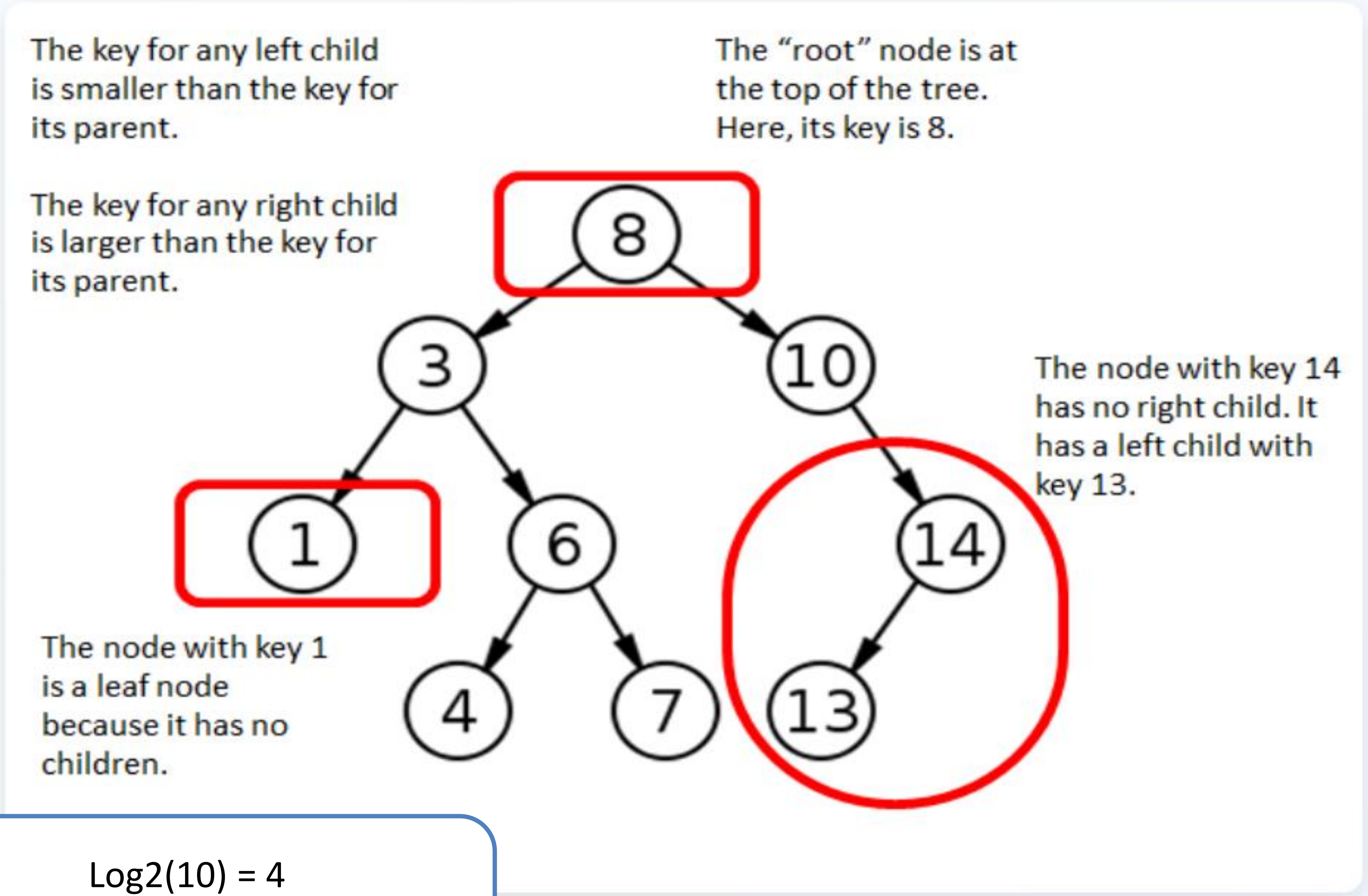
Balanced Binary trees are computationally efficient to perform operations on.
A balanced binary tree will follow the following conditions:
1. The **absolute difference** of heights of left and right subtrees at any node **is less than or equal 1. (difference=-1,0,1) “all nodes ends at level h or h-1”.**
2. For each node, its left subtree is a balanced binary tree.
3. For each node, its right subtree is a balanced binary tree.

Definition and Properties of Binary Search Tree

Definition

A Binary Search Tree is a special binary tree with the following properties:

- Each node's **left subtree** contains values **less than** the node's value
- Each node's **right subtree** contains values **greater than** the node's value
- Both left and right subtrees must also be binary search trees


The key for any left child is smaller than the key for its parent.
The key for any right child is larger than the key for its parent.
The "root" node is at the top of the tree. Here, its key is 8.
The node with key 14 has no right child. It has a left child with key 13.
The node with key 1 is a leaf node because it has no children.
8
3
10
1
6
14
4
7
13

Key Properties

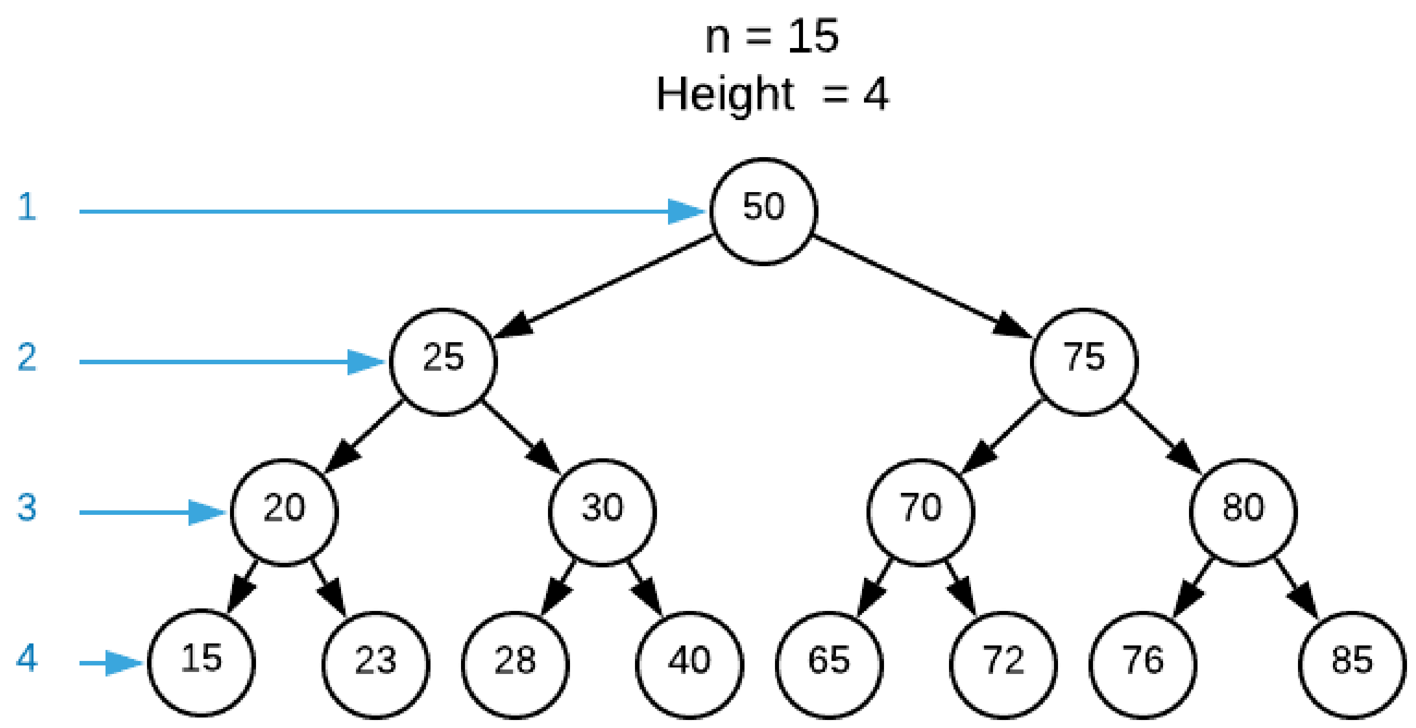
Unique Keys: Each node contains a unique key value

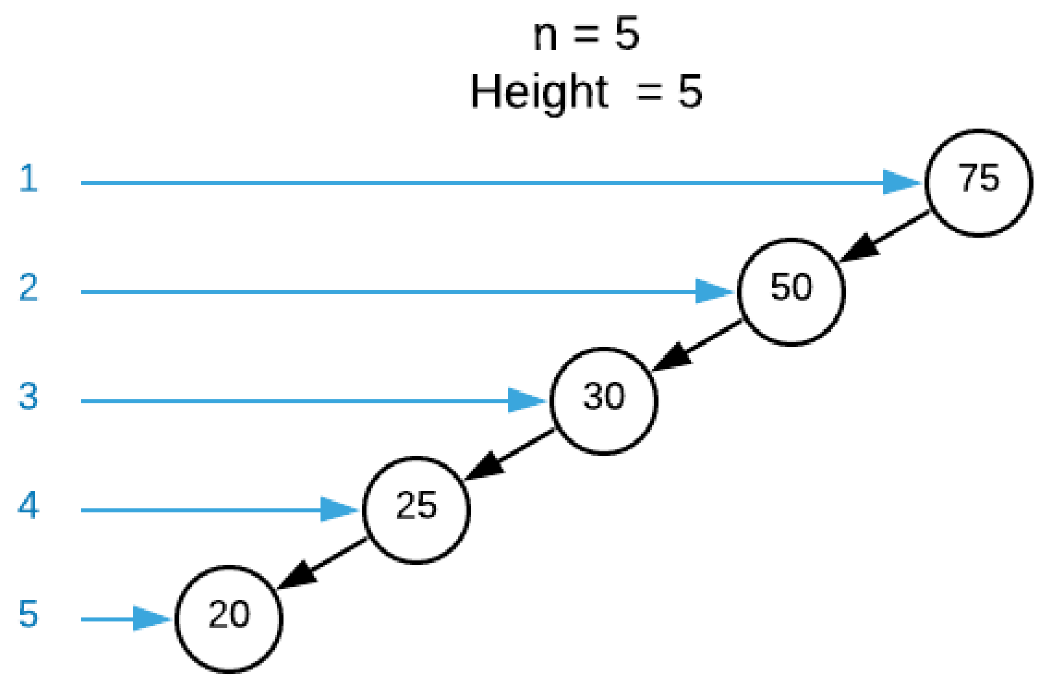
Ordered Structure: Inorder traversal yields sorted sequence

Efficient Search: Average time complexity O(log n)

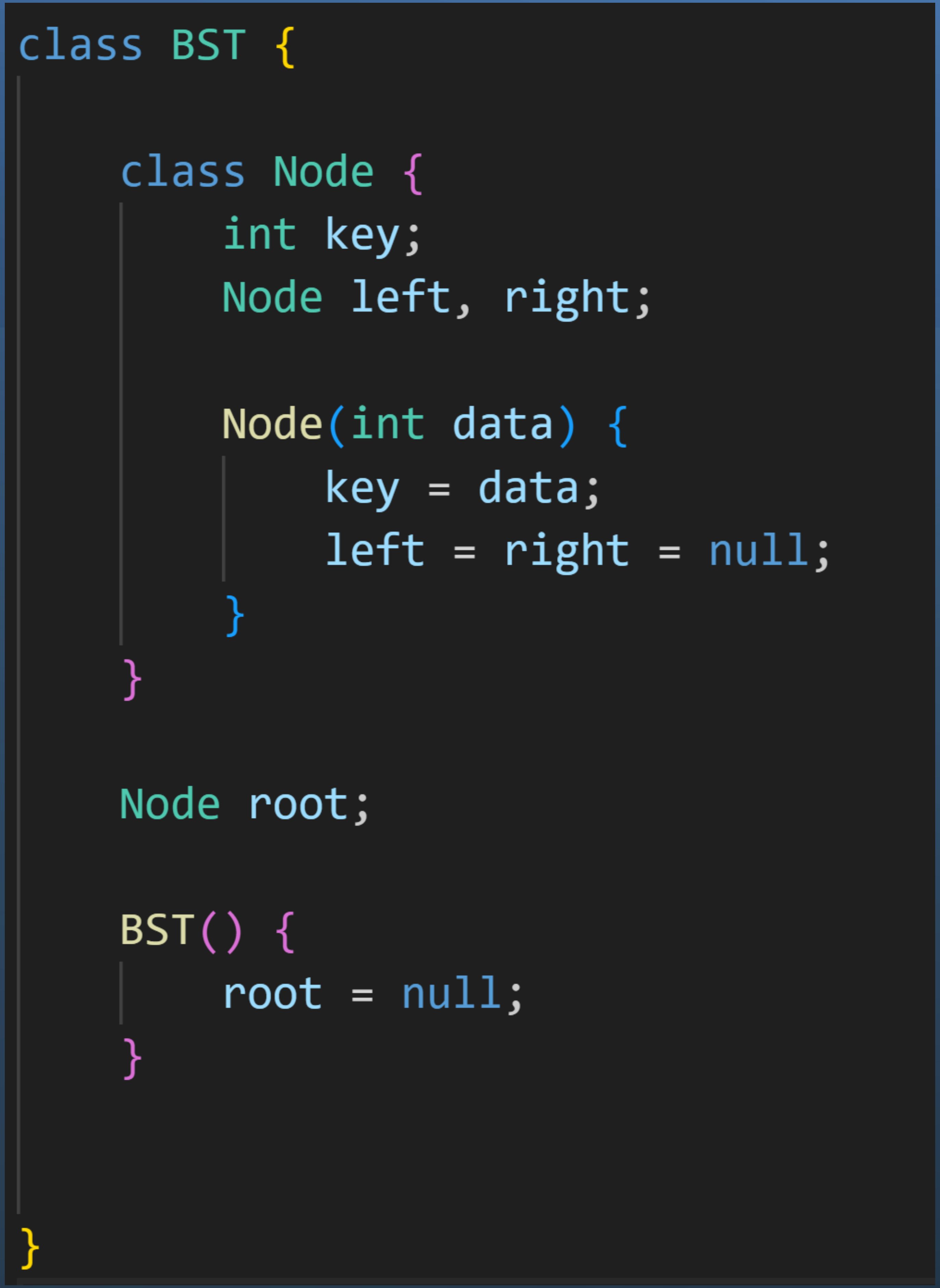
Log2(10) = 4
Log2(1000)=10
Log2(1000000)=20

Dynamic Structure: Supports efficient insertions and deletions





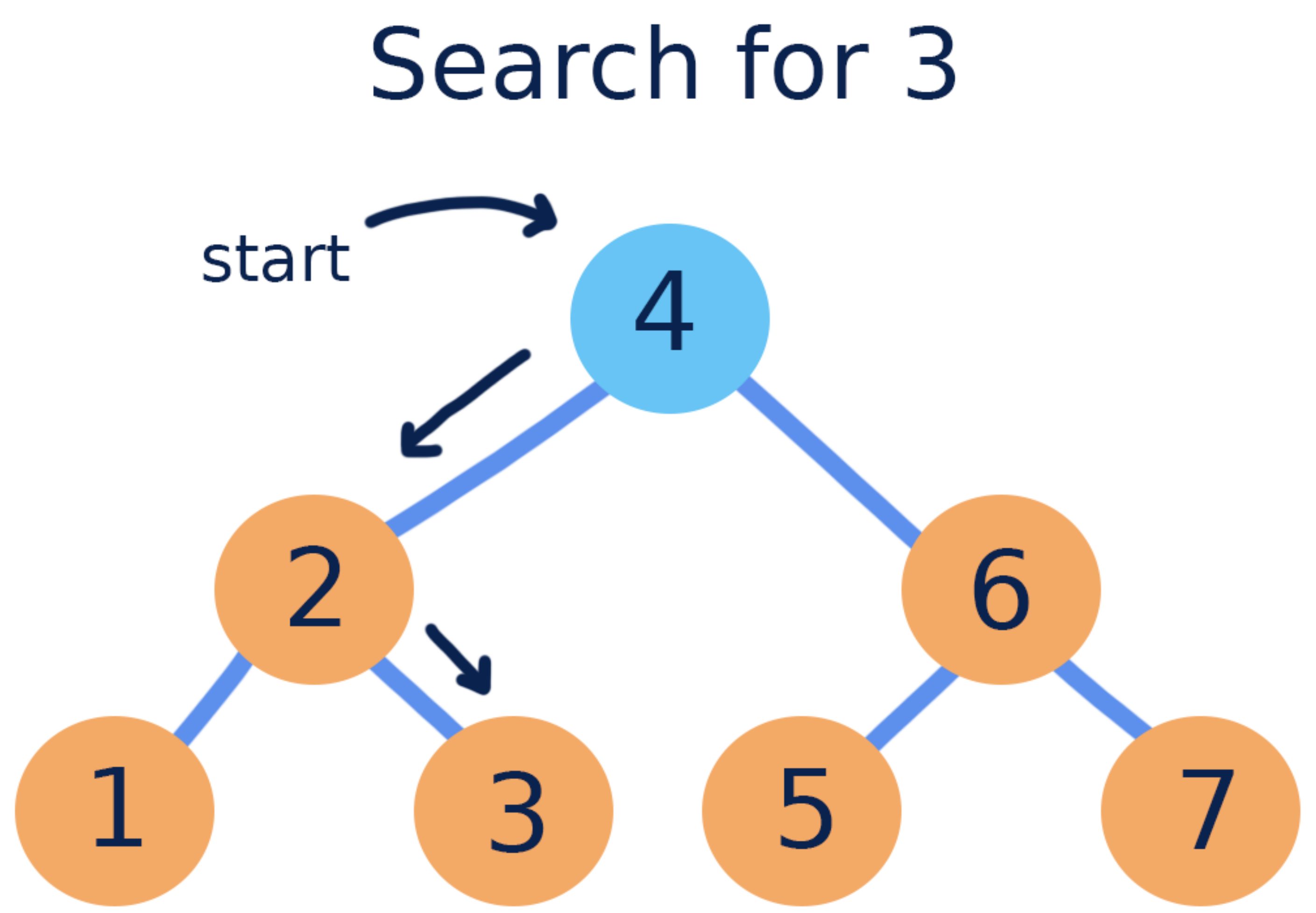
- **Worst Case:**
The BST becomes a skewed tree (like a linked list), making search/insert/delete = O(n).
- If the items are inserted in asc or desc order.


```
class BST {

    class Node {
        int key;
        Node left, right;

        Node(int data) {
            key = data;
            left = right = null;
        }
    }

    Node root;

    BST() {
        root = null;
    }


}
```

Search Operation in BST

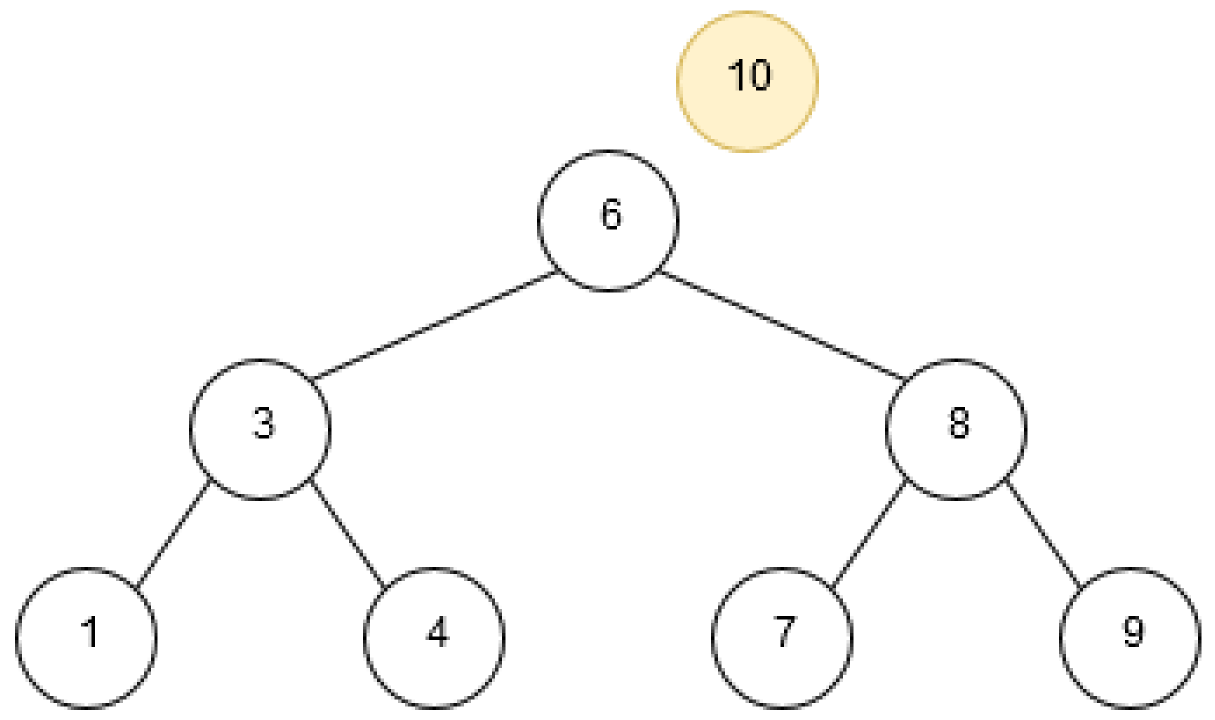
Search Algorithm Steps

1. Start from **root node**
2. Compare target value with current node
3. Target < current node → search **left subtree**
4. Target > current node → search **right subtree**
5. Stop when target found or null node reached



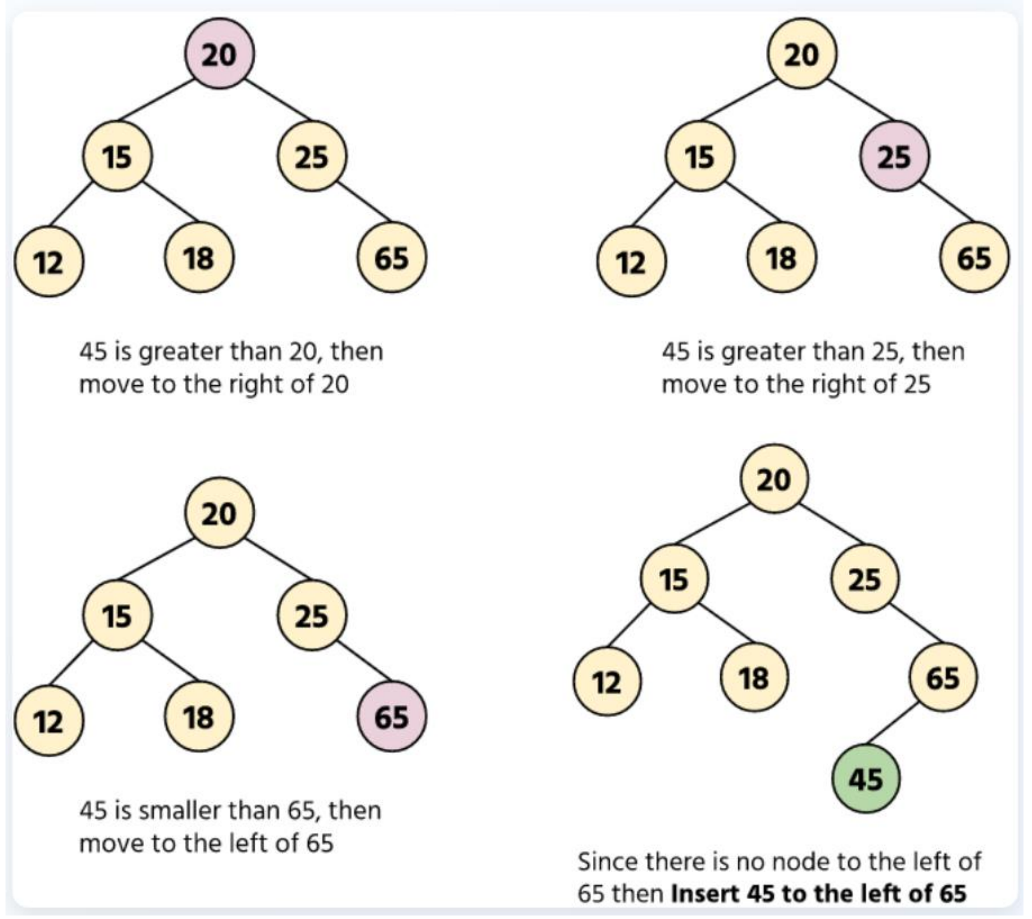
```java
Node search_Recursive(Node root, int key) {
    if (root == null || root.key == key) return root;
    if (key < root.key) return search_Recursive(root.left, key);
    return search_Recursive(root.right, key);
}
```

Insert Operation in BST



Insert Algorithm Steps

1. Start from **root node**
2. Compare new value with current node
3. New value < current node → go to **left subtree**
4. New value > current node → go to **right subtree**
5. Create new node at empty position



20
15
25
12
18
65
45 is greater than 20, then move to the right of 20
45 is greater than 25, then move to the right of 25
45 is smaller than 65, then move to the left of 65
45
Since there is no node to the left of 65 then **Insert 45 to the left of 65**

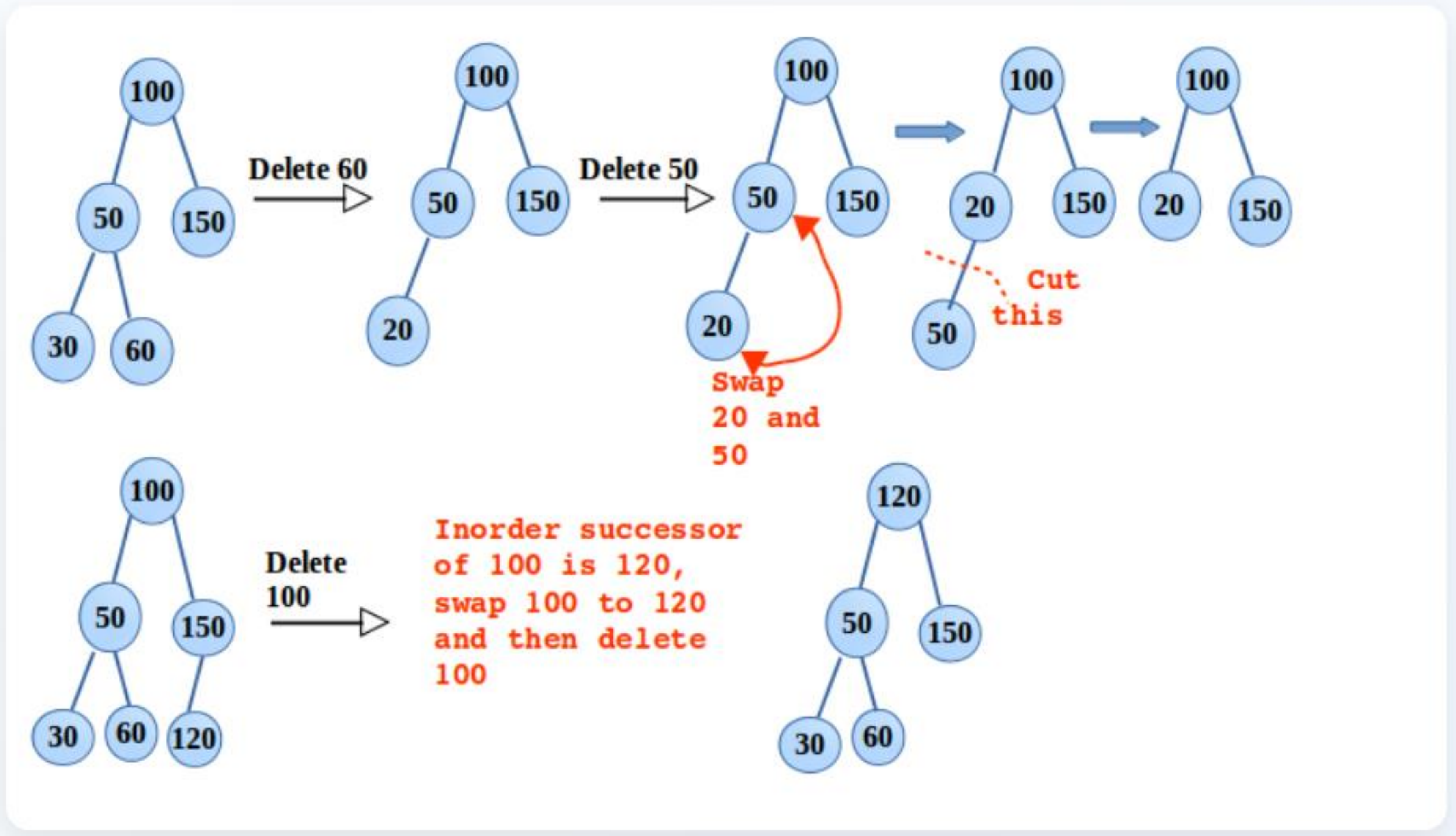
Task 1 : code for insert in BST

Delete Operation in BST

Three Deletion Cases

1. Delete **leaf node**: Simply remove
2. Delete node with **one child**: Replace with child
3. Delete node with **two children**: Replace with inorder successor or predecessor

Key point: Inorder successor is the minimum value in right subtree, inorder predecessor is the maximum value in left subtree



Task 1 : code for delete in BST

Traversal Operations in BST

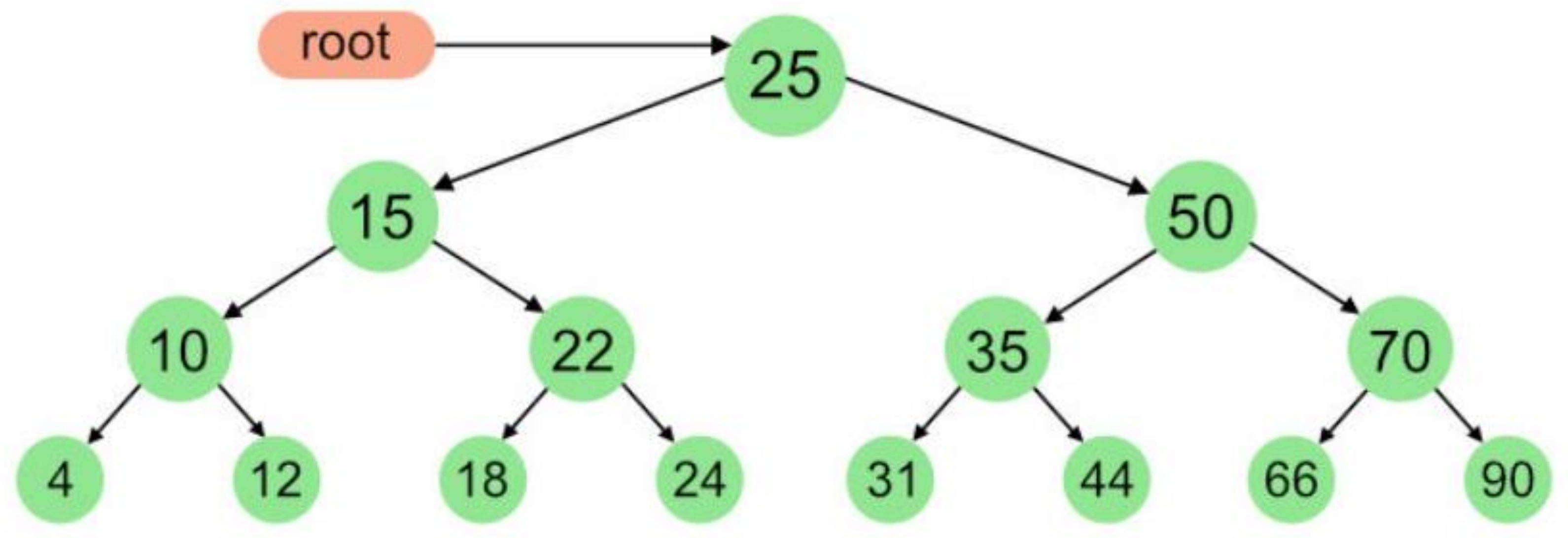
Three Traversal Methods

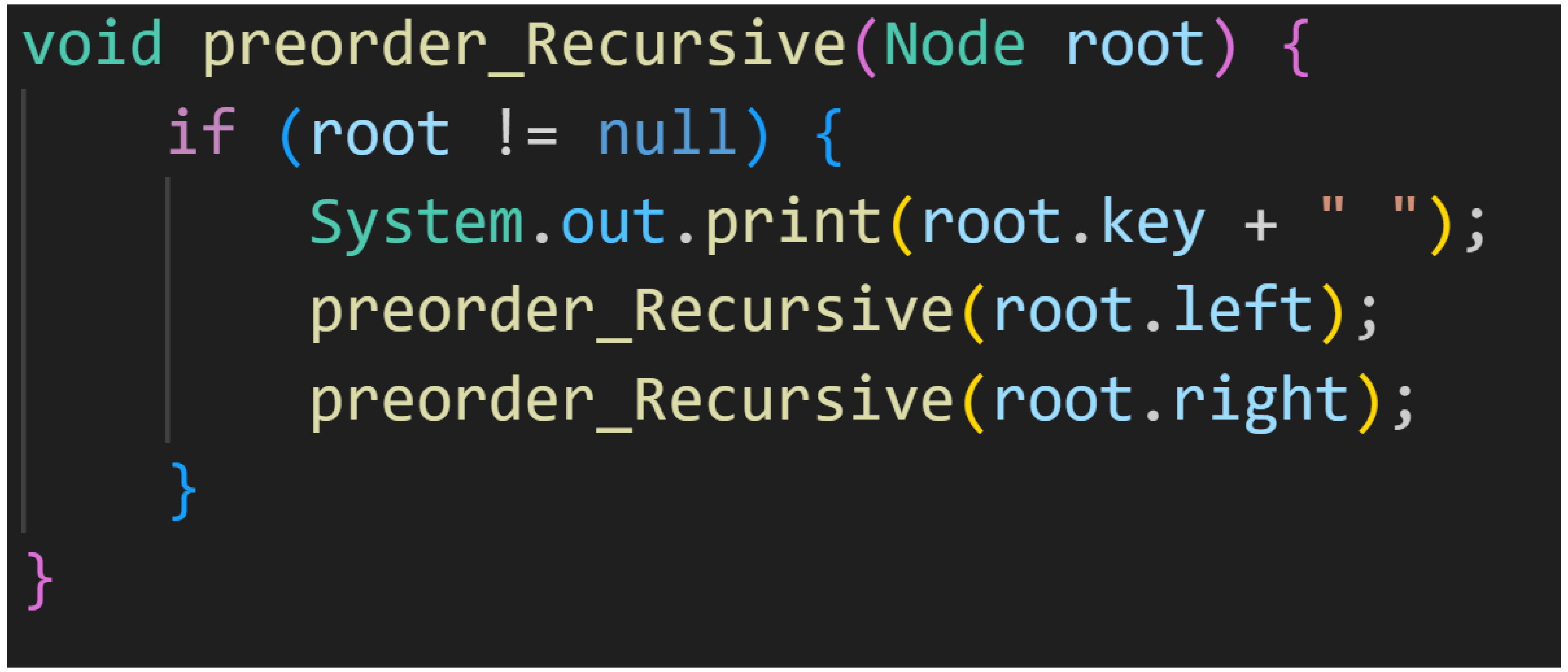
1. **Preorder**: Root → Left → Right
2. **Inorder**: Left → Root → Right
3. **Postorder**: Left → Right → Root

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

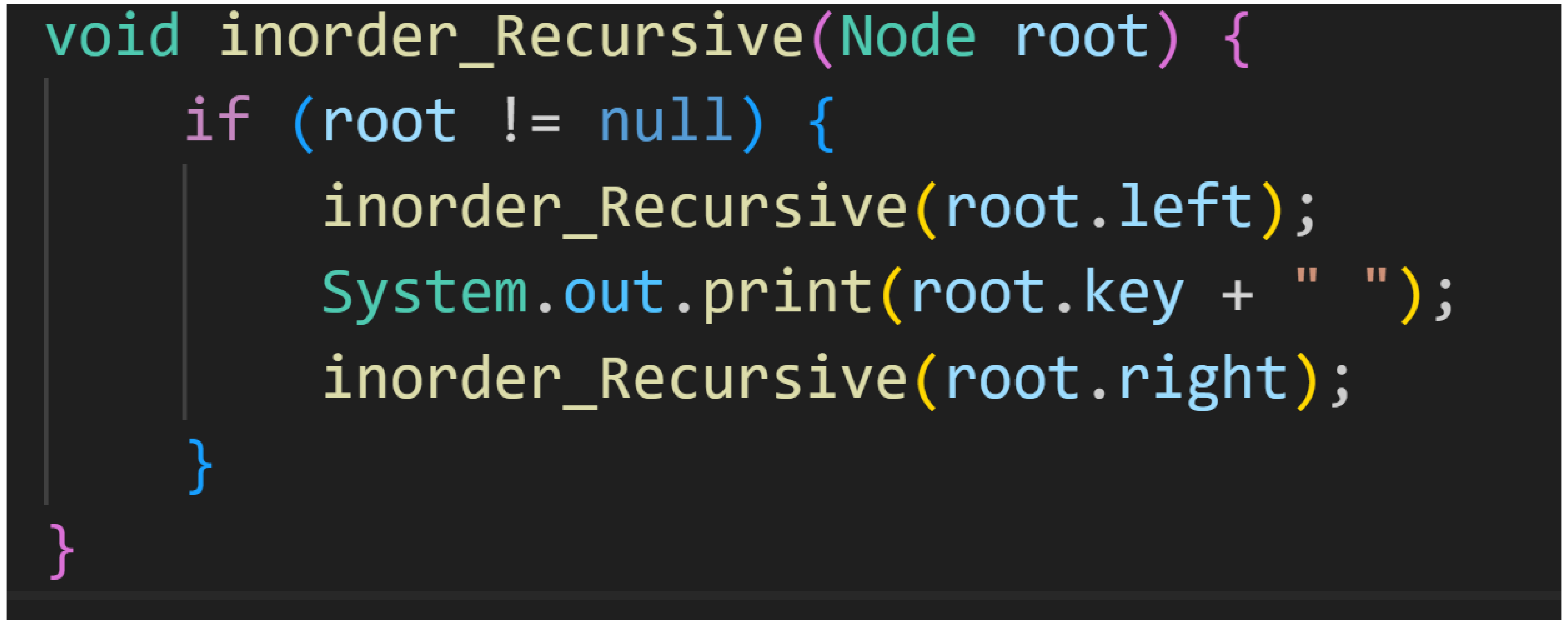
A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

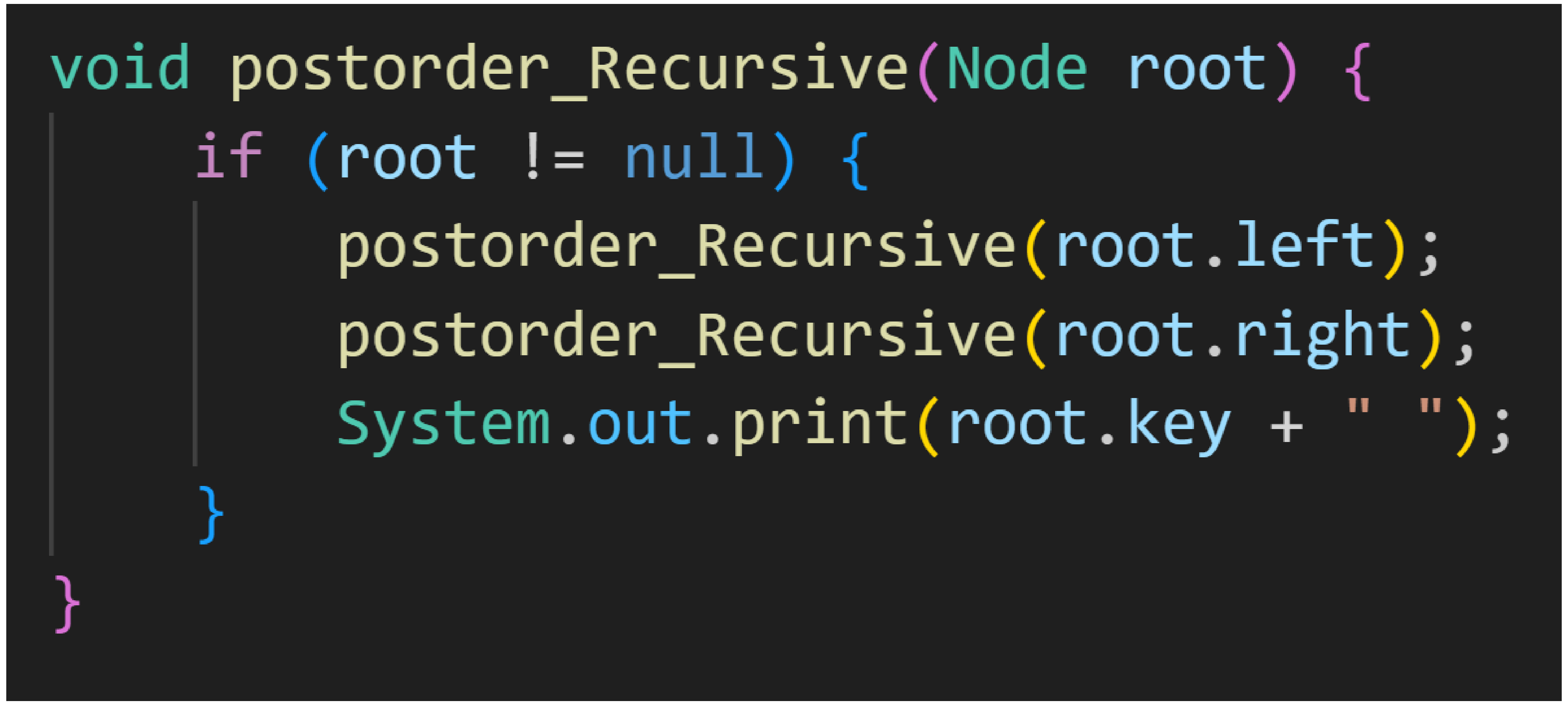




```
void preorder_Recursive(Node root) {
    if (root != null) {
        System.out.print(root.key + " ");
        preorder_Recursive(root.left);
        preorder_Recursive(root.right);
    }
}
```



```
void inorder_Recursive(Node root) {
    if (root != null) {
        inorder_Recursive(root.left);
        System.out.print(root.key + " ");
        inorder_Recursive(root.right);
    }
}
```



```
void postorder_Recursive(Node root) {
    if (root != null) {
        postorder_Recursive(root.left);
        postorder_Recursive(root.right);
        System.out.print(root.key + " ");
    }
}
```

THANK YOU!