6/05/2023

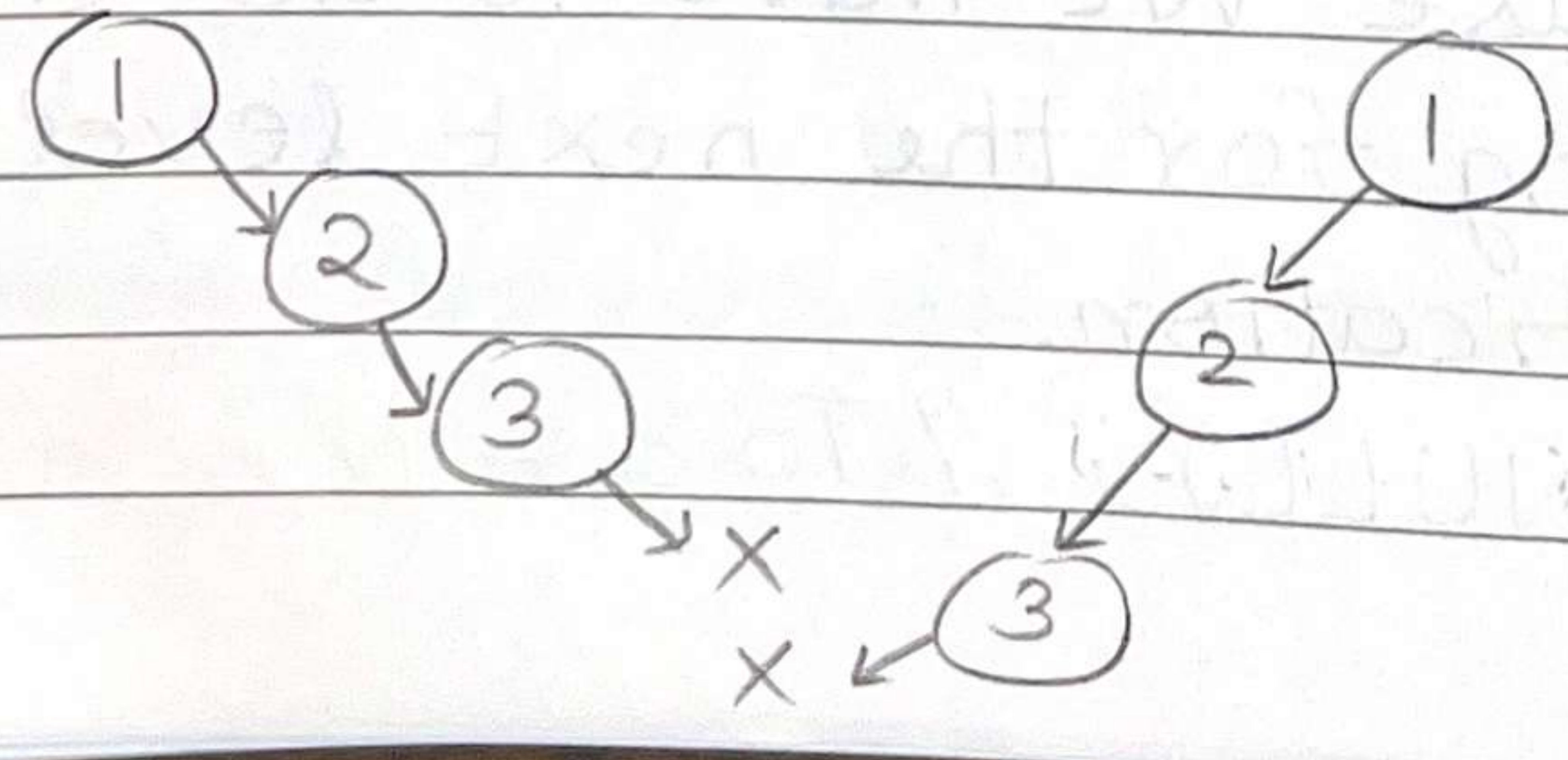
n-ary trees

Tree data structure that allow us to have upto n-children nodes for each node.

```
class Node {
        int data;
        vector <Node *> child; // Array of
                                  pointers
};
```

Note→ There is no official algorithm to create the tree.

Skew Trees

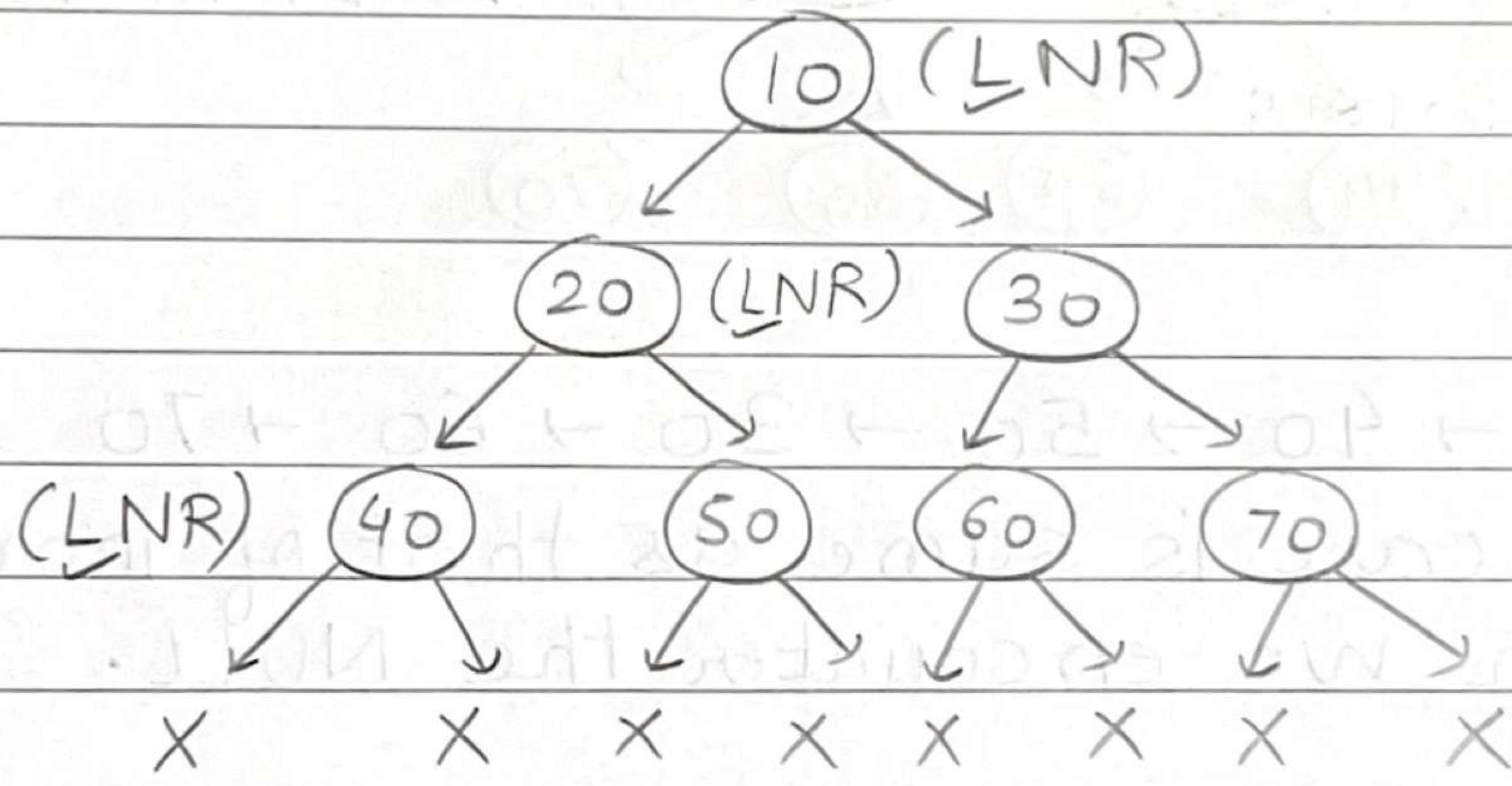


The binary tree in which each node has either one child or no child is known as skewed binary tree. In this type of tree, either all nodes are positioned to the left or to the right.

Types of traversal (continued)

2) Inorder traversal

Here the mapping is like LNR i.e left, node & right.



(B.C) & return

40 → 20 → 50 → 10 → 60 → 30 → 70

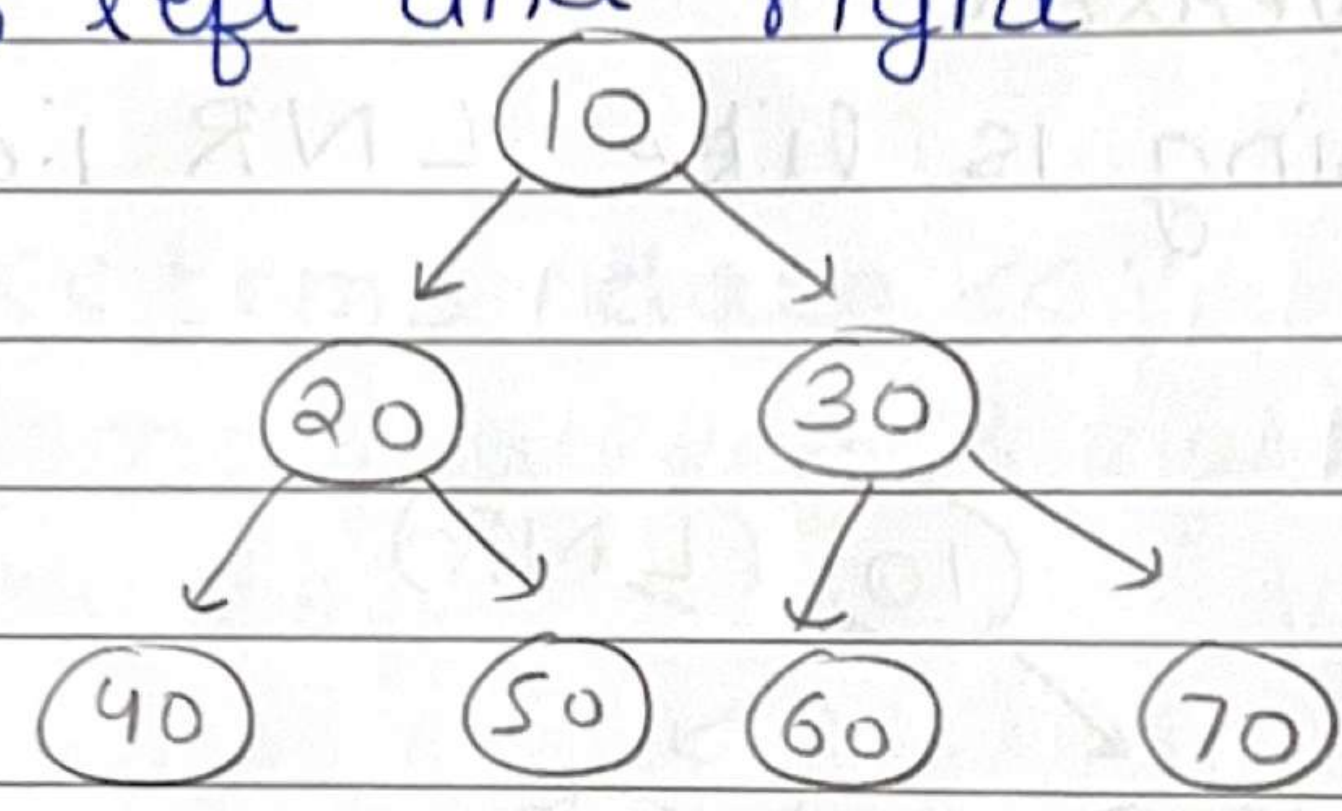
At each node we have to follow LNR and the base case is when we encounter NULL.

Code

```
void inorderTraversal(Node *root){
    // Base case
    if(root == NULL){
        return;
    }
    // Left child
```

```
    inorderTraversal (root→left);
    //Node
    cout << root→data << " ";
    // Right child
    inorder Traversal (root→right);
}
```

3) Preorder traversal The mapping here is NLR i.e node, left and right



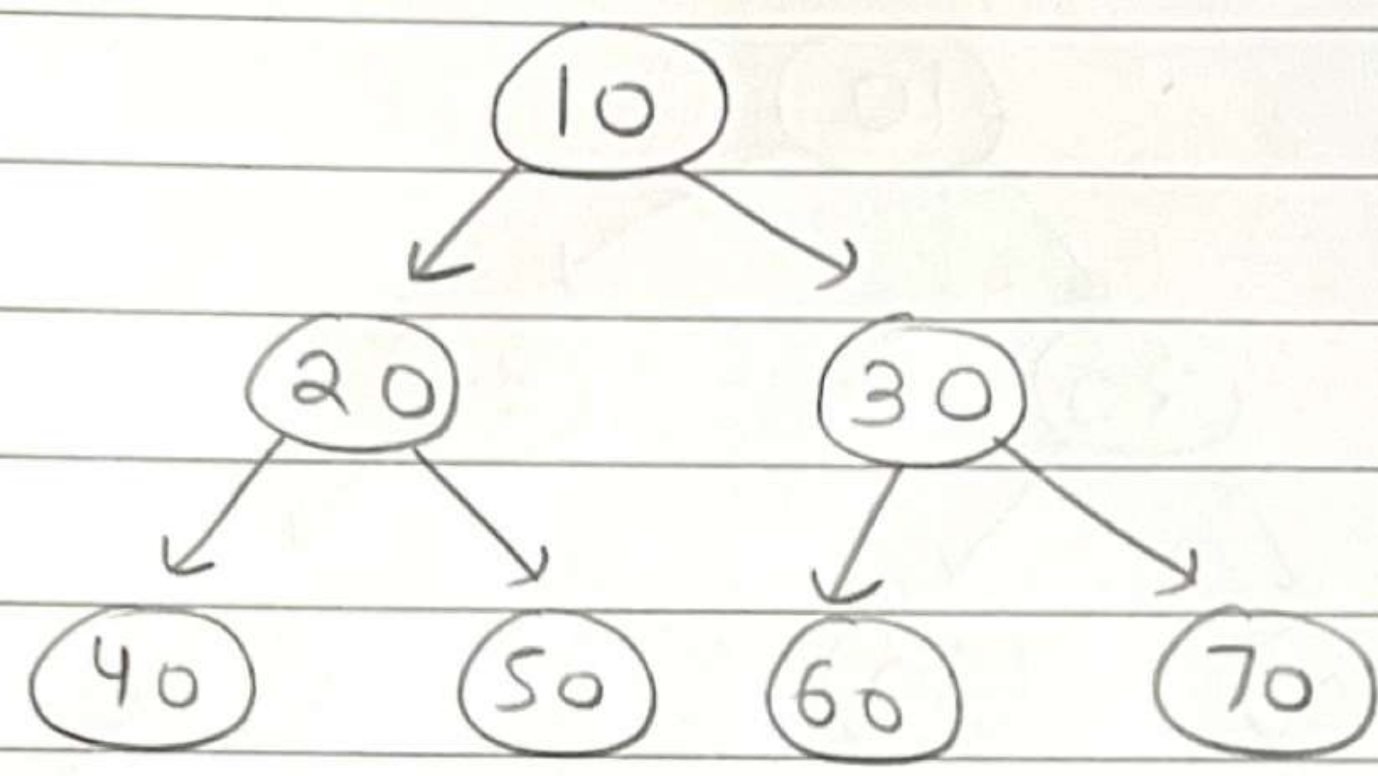
10 → 20 → 40 → 50 → 30 → 60 → 70

The base case is same as that of inorder i.e when we encounter the NULL.

Code

```
void preOrderTraversal (Node * root){
    // Base case
    if (root == NULL){
        return;
    }
    //Node
    cout << root→data << " ";
    // Left child
    preOrderTraversal (root → left);
    //Right child
    preOrderTraversal (root→right);
}
```

4) Postorder traversal The mapping here is LRN.
i.e. Left, Right and node.



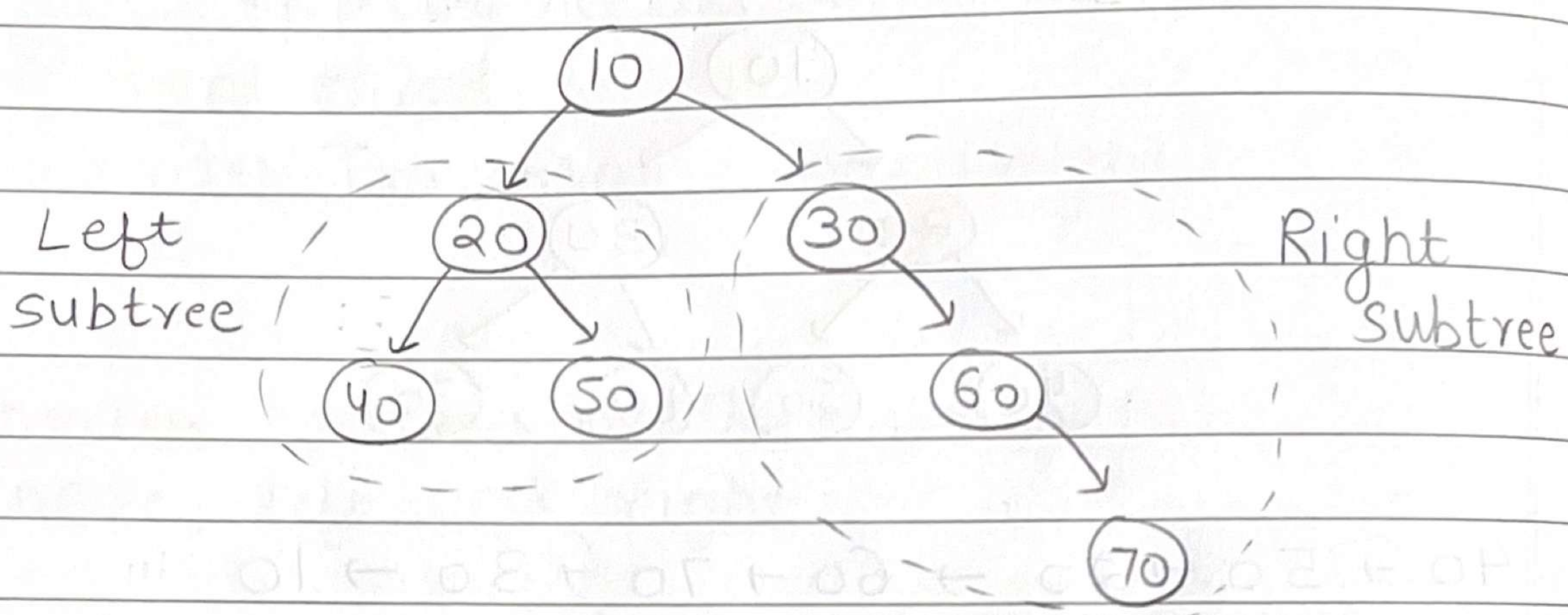
40 → 50 → 20 → 60 → 70 → 30 → 10

Code

```
void postOrderTraversal (Node * root){
        // Base case
        if (root == NULL){
              return;
        }
      // Left child
      postOrderTraversal (root → left);
     //Right child
      postOrderTraversal (root → right);
     //Node
     cout << root → data << " ";
}
```

Height of the tree
The height of binary tree is defined as maximum depth of any leaf node from root node.

Note → At some places the height of tree is defined in terms of the number of links instead of no. of nodes.

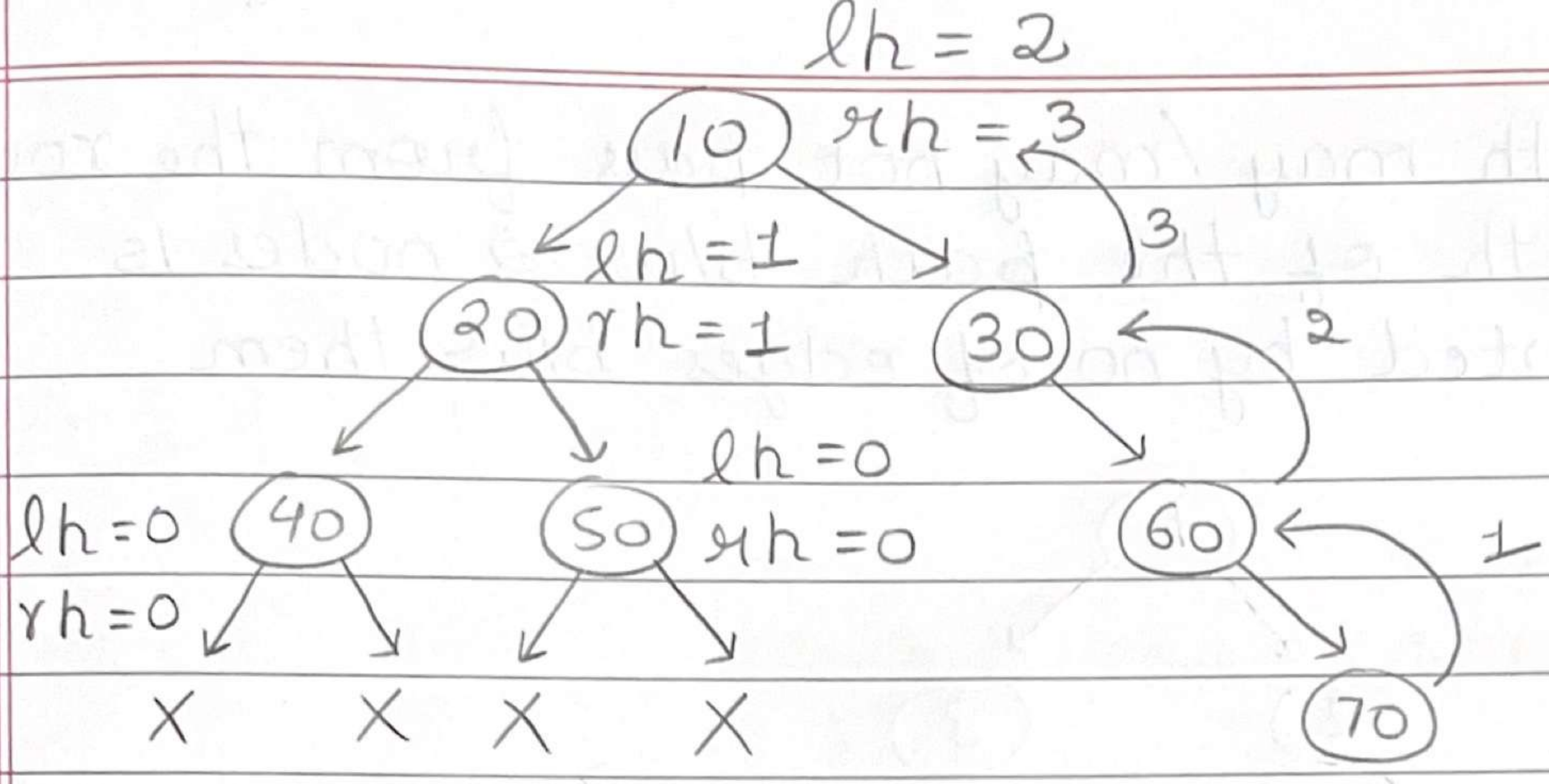


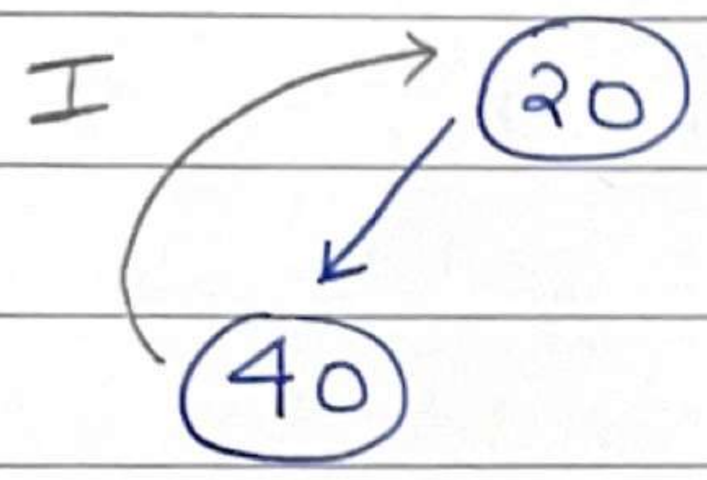
height → max (left, right) + 1.
Here we have done +1 to take in consideration the root node.

Code

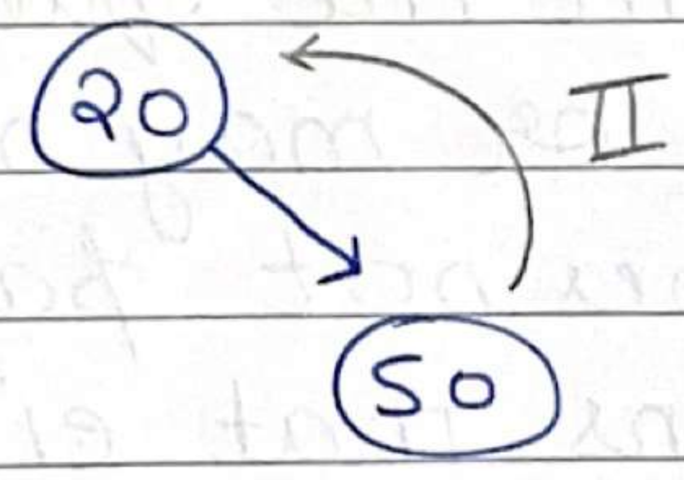
```
int height (Node * root) {
    // Base case → Empty tree
    if (root == NULL) {
        return 0; // 0 is height of empty
                     tree
    }
    // Left subtree height
    int lh = height (root → left);
    // Right subtree height
    int rh = height (root → right);
    // Max of both the heights
    int ans = max (lh, rh) + 1;
    // Height should be returned ↳ To consider root
    return ans;                    node (1 case
}                                  solve)
```

Dry run

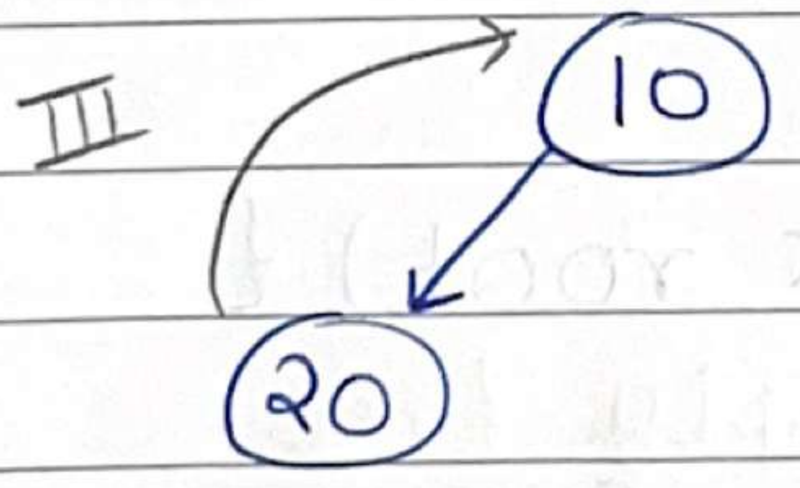




I → max (lh, rh) + 1
max (0, 0) + 1 = 0 + 1



II → max (lh, rh) + 1
max (0, 0) + 1 = 0 + 1 = 1

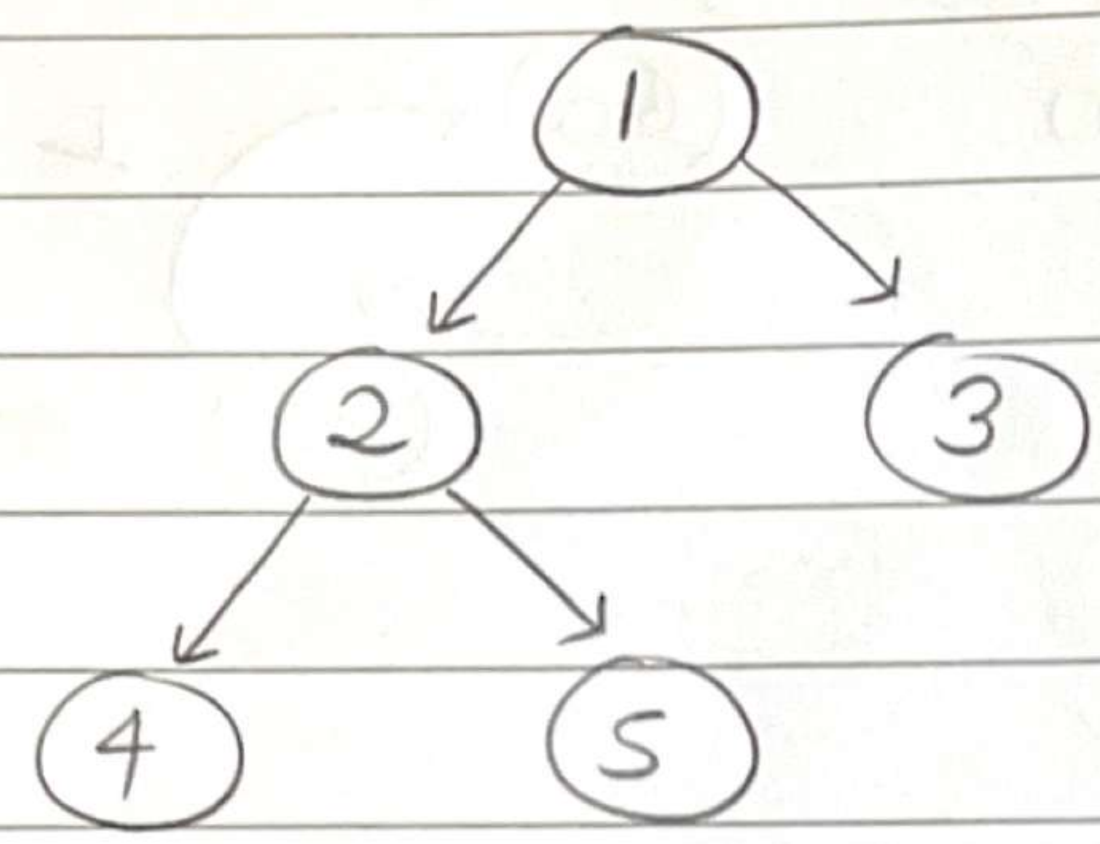


III → max(lh, rh) + 1
max (1, 1) + 1 = 1 + 1 = 2

Now at root node we have lh = 2 and rh = 3
max (2, 3) + 1 = 3 + 1 = 4 is the height.

Diameter of binary tree
The diameter of a binary tree is length of longest path between any 2 nodes in a tree.

This path may /may not pass from the root.
The length of the path b/w 2 nodes is represented by no. of edges b/w them.



Diameter = 3
$\underbrace{4 \rightarrow 2 \rightarrow 1 \rightarrow 3}_{\text{3 edges}}$

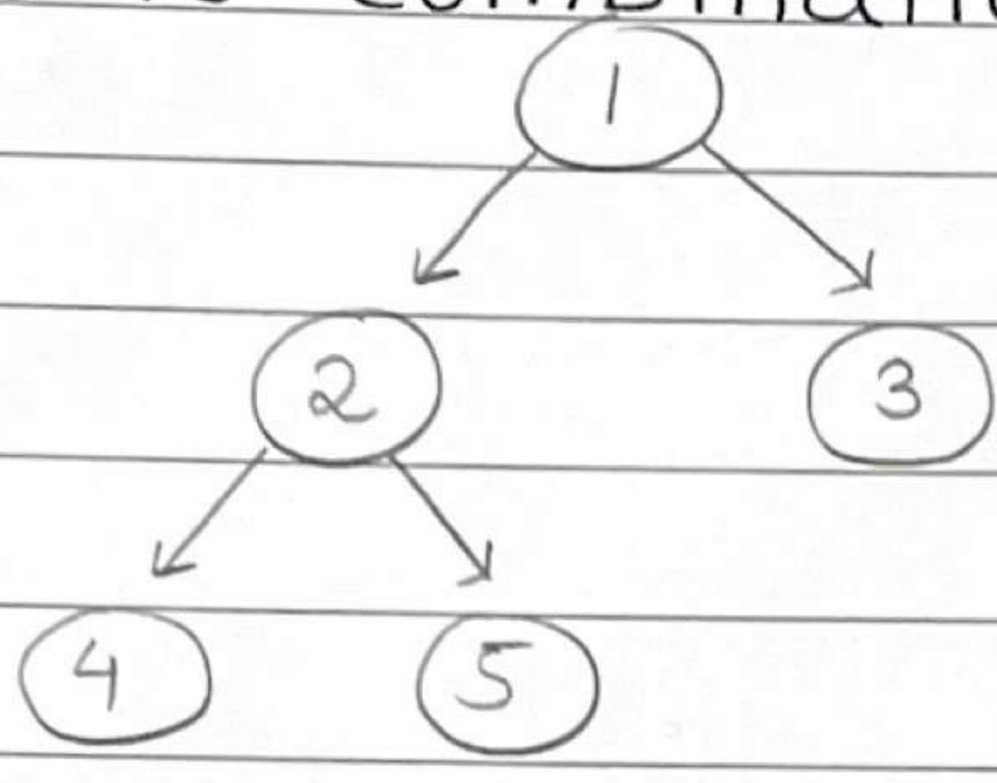
Important hint in the question
Longest path may or may not pass through the root. If it does not pass through the root, then it means that either the path is in left subtree or the right subtree.

Code

```
int diameter (Node * root) {
    // Base case → Empty tree
    if (root == NULL) {
        return 0;
    }
    //left subtree check
    int op1 = diameter (root → left);
    // right subtree check
    int op2 = diameter (root → right);
    // root is included in answer
    int op3 = height (root → left) + height (root → right
```

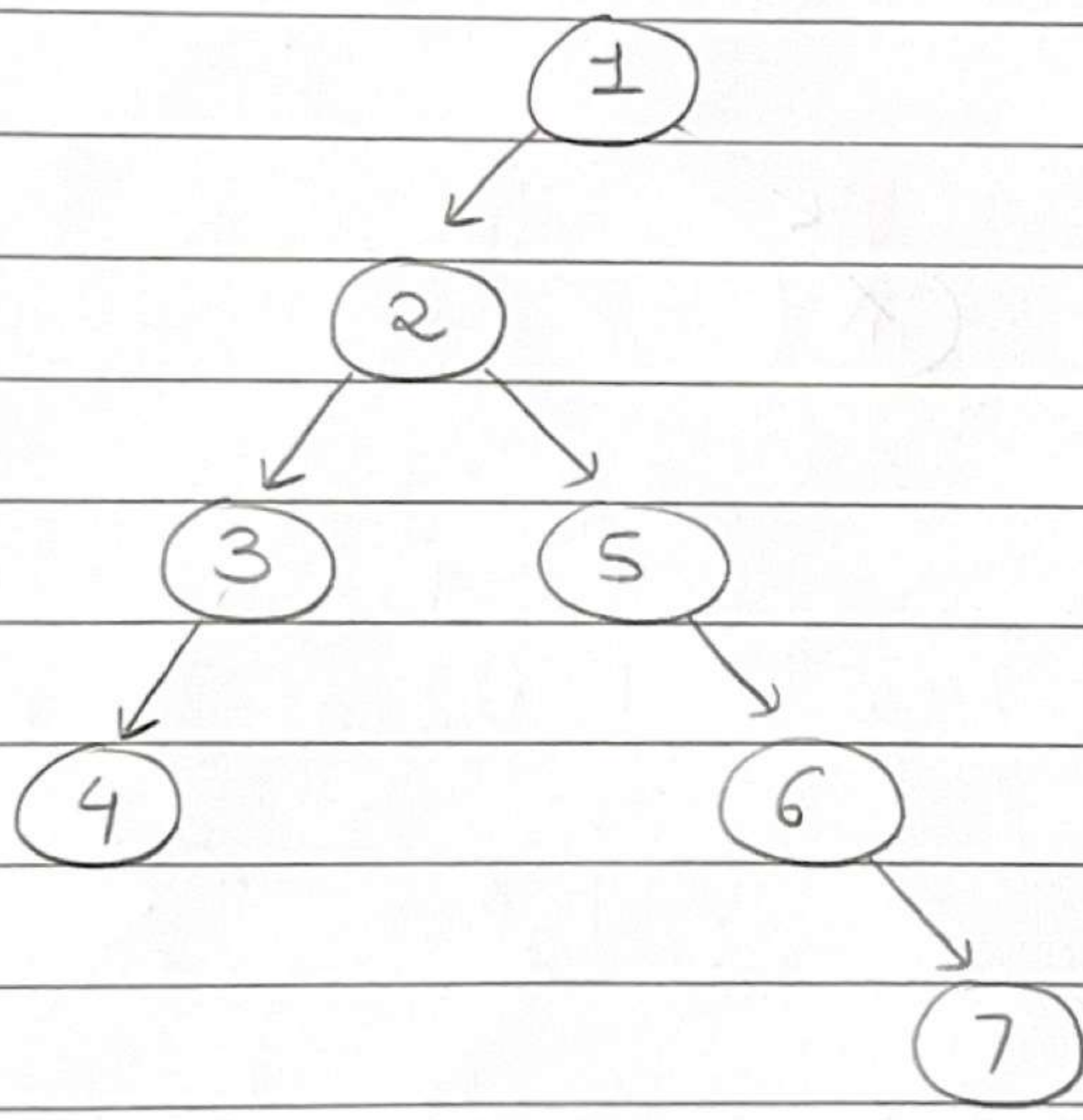
```
// Longest path & hence maximum is taken
int ans = max(op1, max(op1, op2));
// Return diameter
return ans;
}
```

(1) When ans is combination of both subtrees



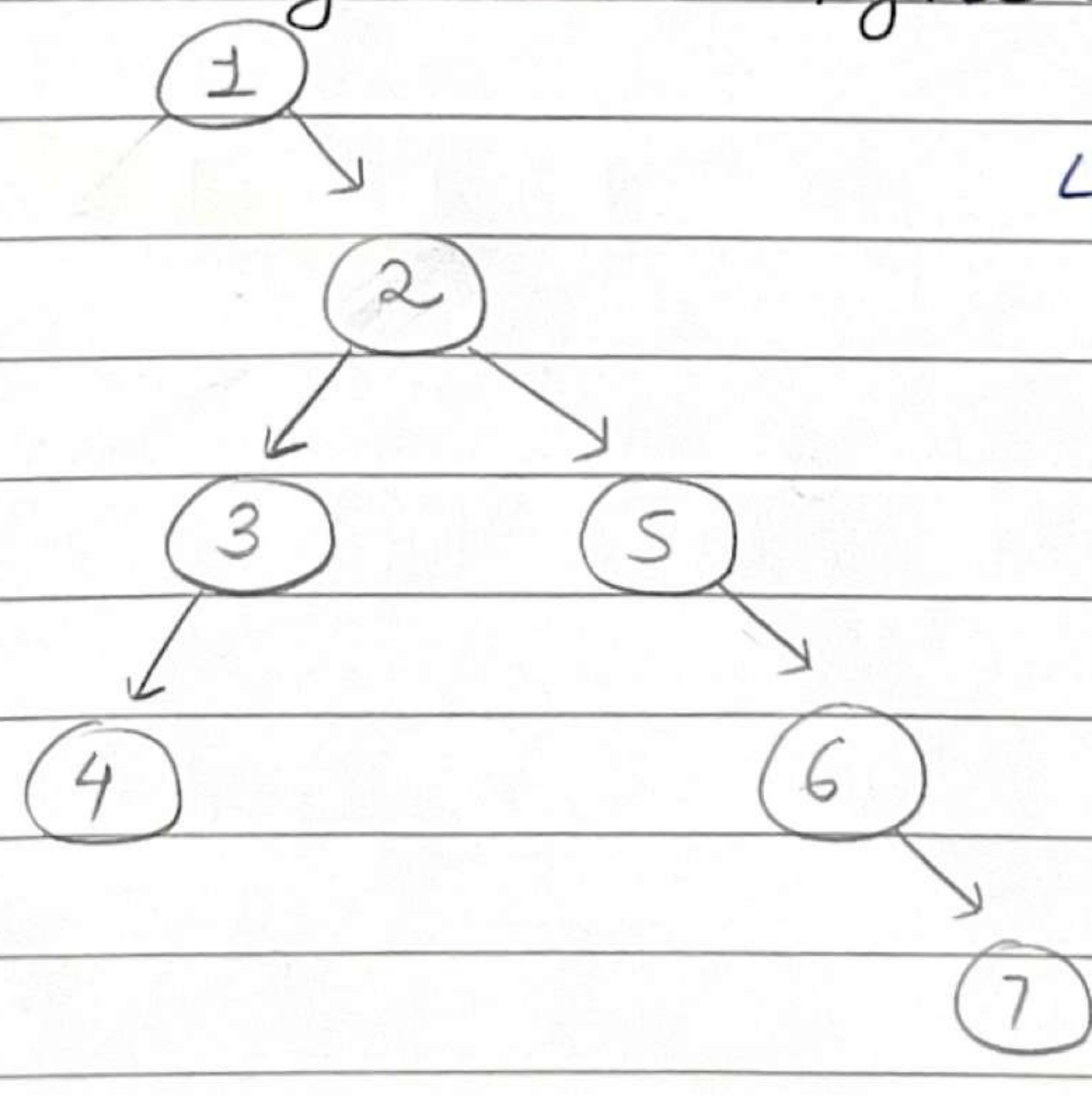
4 → 2 → 1 → 3

(2) When ans is coming from left subtree



4 → 3 → 2 → 5 → 6 → 7

(3) When ans is coming from right sub tree.



4 → 3 → 2 → 5 → 6 → 7