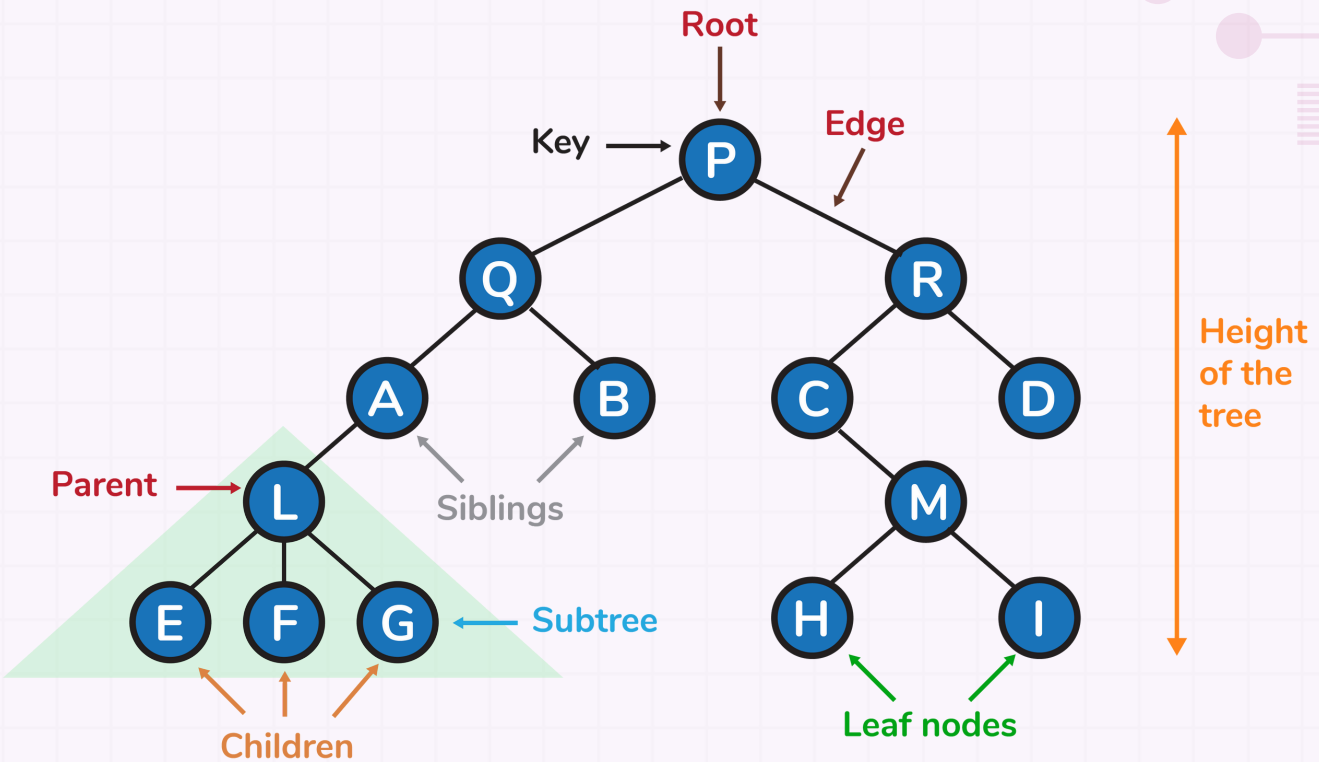
InterviewCafe
Where Learning Never Takes a Break!

Top Tree Interview Questions & Explanations



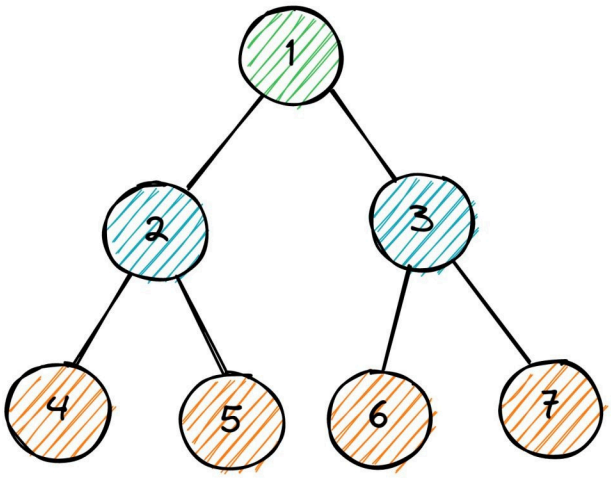
Santosh Kumar Mishra
Software Engineer at Microsoft • Author • Founder of InterviewCafe

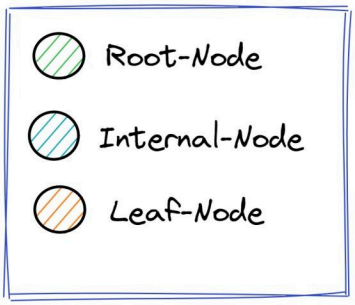
Table of Contents

Tree Data Structure

→ In simple words we can say it is used to explain the hierarchical relationships. example - family tree.

→ This hierarchical structure of trees is used in Computer science as an abstract data type for various applications like data storage, search and sort algorithms.





1. Sibling :-

If we take node (2) & (3) . They are sibling because node (1) is its parent.

Same as for node (4) & (5). Its parent node is (2)

2. Cousion :-

If we take node (4) & (5) . They are cousions of node (6) & (7) ,

Because node (2) & (3) are sibling and its parent node is (1) .

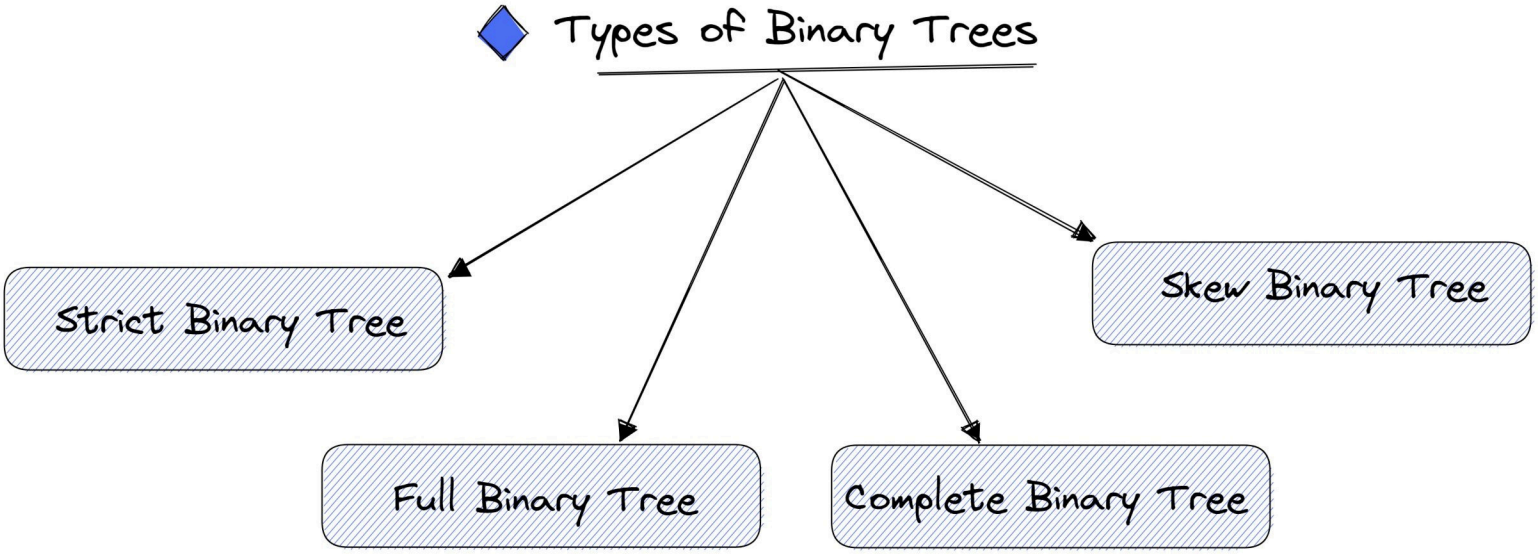
3. Ancestor :-

for node (4) & (5) , (6) & (7) , There Ancestor is node (1) .

Binary Trees

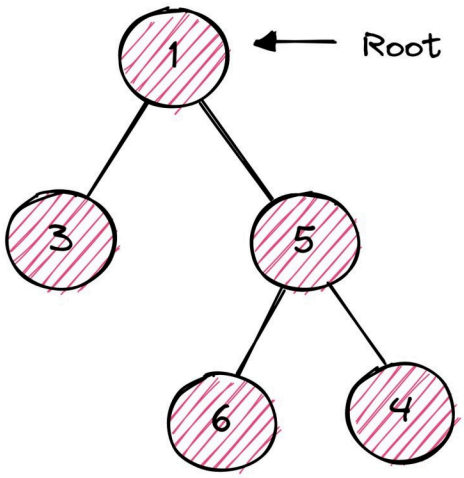
→ A tree is called binary tree if node has zero, one or two children.

→ We can visualize a binary tree as consisting of root node, left child & right child.



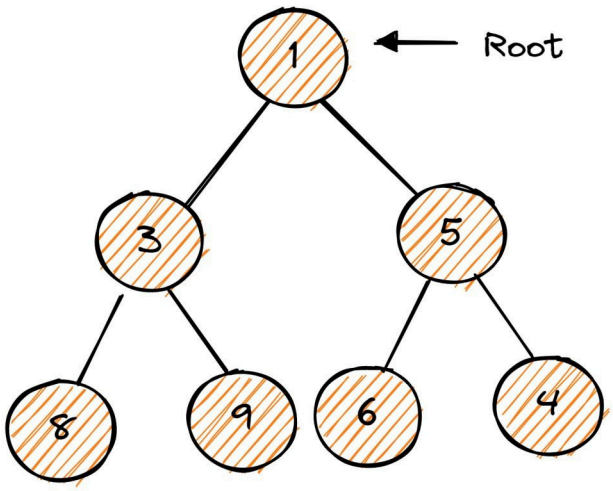
1. Strict Binary Tree :-

A binary tree is called strict binary tree if each node has exactly two children or no children.



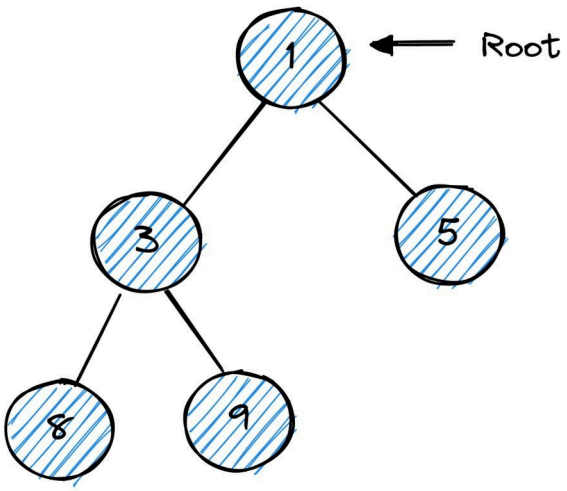
2. Full Binary Tree :-

A binary tree in which each node have two children and all the leaf nodes are on the same level.



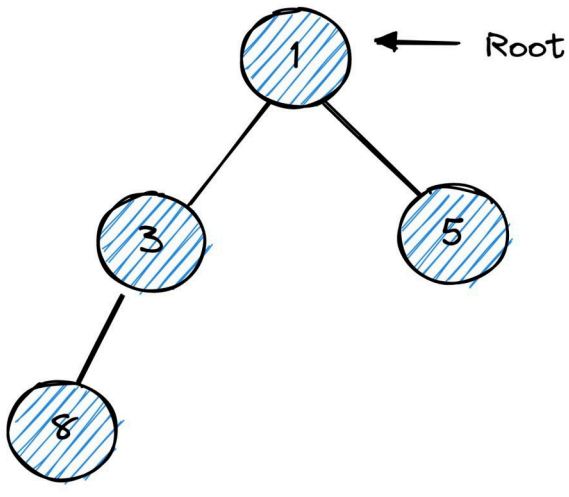
3. Complete Binary Tree :-

→ Binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.



✓ Complete Binary tree

✓ Full Binary tree

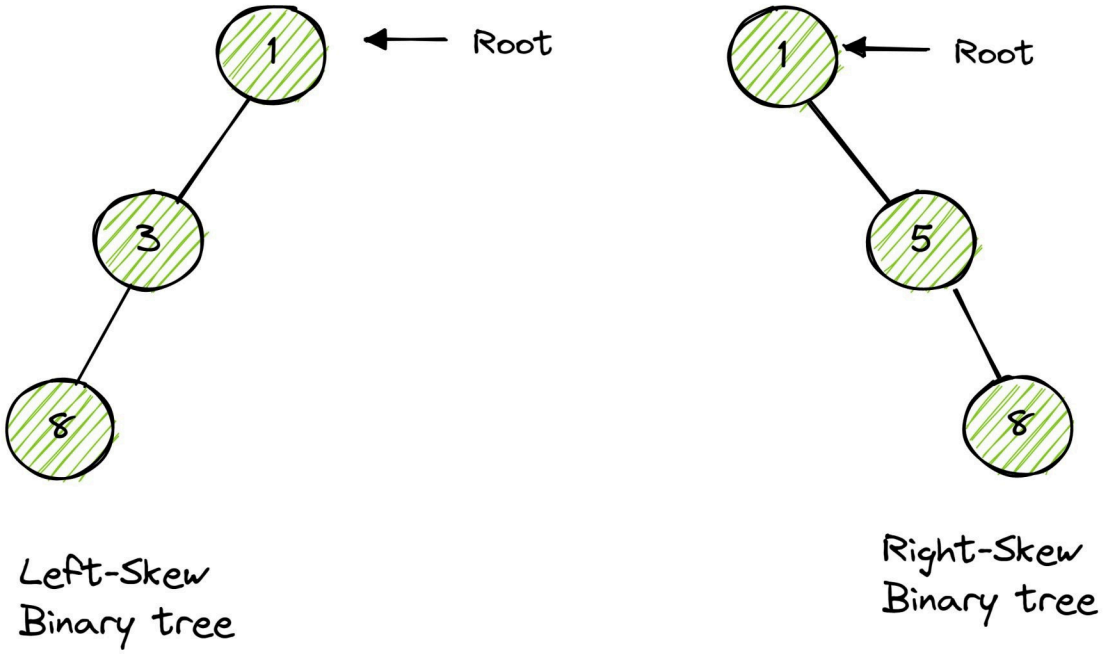


✓ Complete Binary tree

✗ Full Binary tree

4. Skew Binary Tree :-

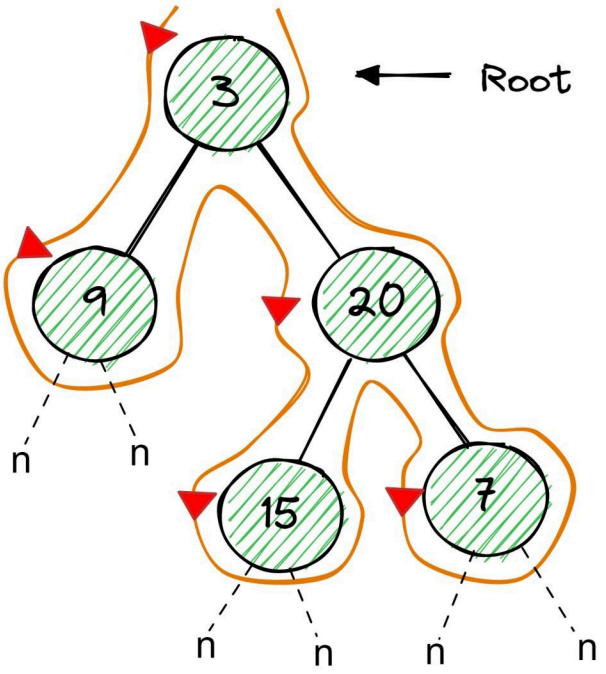
→ Binary tree in which every parent has exactly one child.

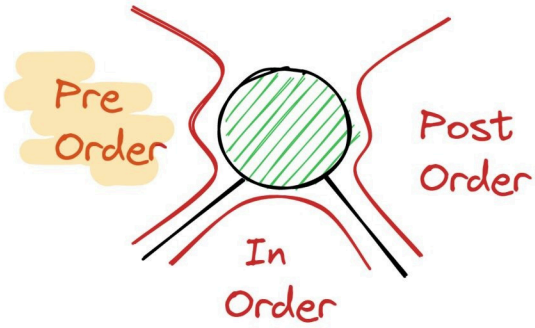


Binary Tree Preorder Traversal

Easy

→ Given the root of a binary tree, return the preorder traversal of its nodes' values.



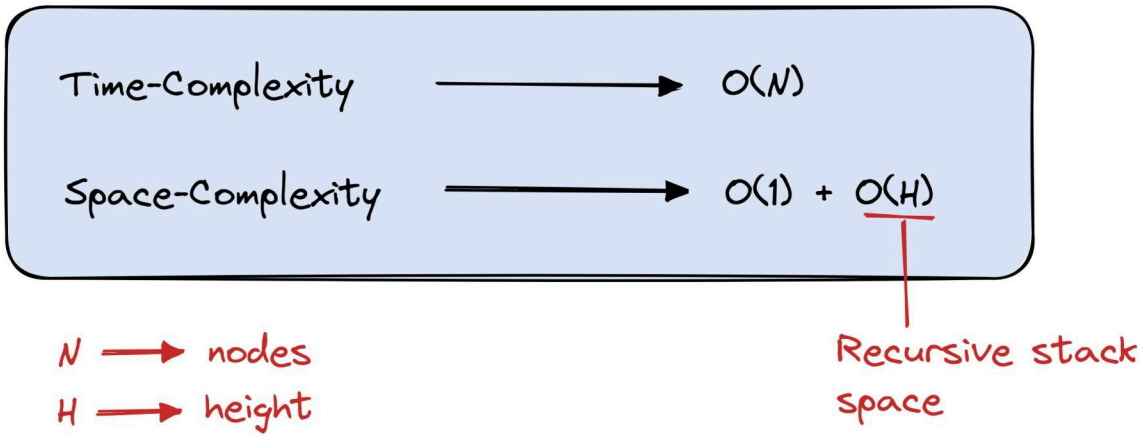



```
Input: root = [3,9,20,null,null,15,7]
Output: [3,9,20,15,7]
```

Pre-Order ⟶ Node
Left-Child
Right-Child

- The steps to perform the preorder traversal are listed as follows -
  - → First, visit the root node.
  - → Then, visit the left subtree.
  - → At last, visit the right subtree.

```java
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {

        List<Integer> list = new ArrayList<>();
        preOrder(root,list);
        return list;
    }

    public void preOrder(TreeNode root,List<Integer> list){

        if(root == null){
            return;
        }
        list.add(root.val);
        preOrder(root.left,list);
        preOrder(root.right,list);
    }
}
```



Note: we can also solve this problem using Stack.

Approach-2 (using Stack)

```java
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {

        List<Integer> list = new ArrayList<>();
        if(root == null) return list;

        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        while(!stack.isEmpty()){

            root = stack.pop();
            list.add(root.val);
            if(root.right != null) stack.push(root.right);
            if(root.left != null) stack.push(root.left);
        }
        return list;
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(H)

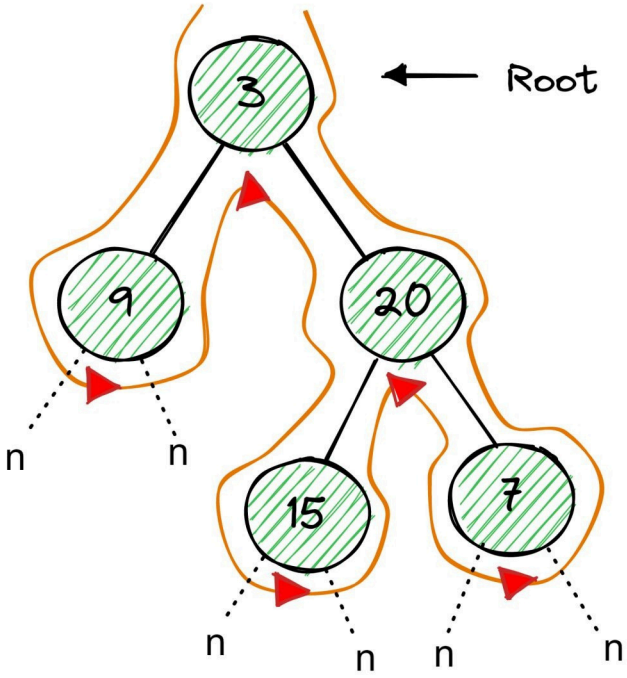
N ⟶ nodes

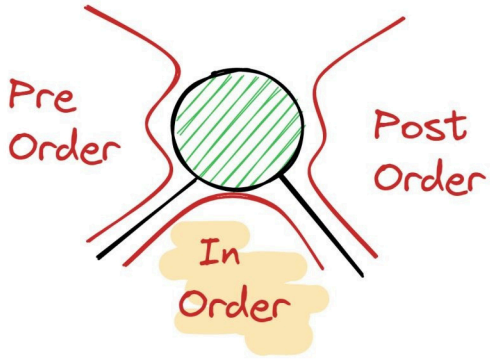
H ⟶ height

Binary Tree Inorder Traversal

Easy

→ Given the root of a binary tree, return the inorder traversal of its nodes' values.



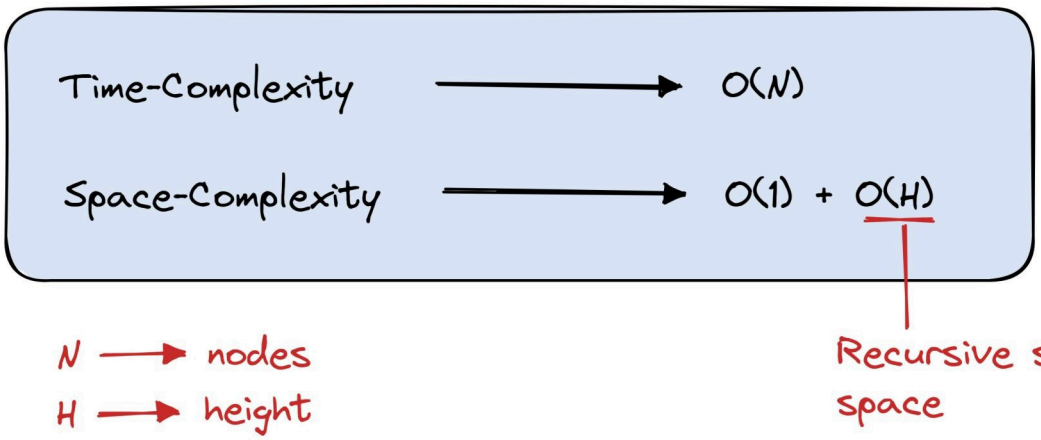



```
Input: root = [3,9,20,null,null,15,7]
Output: [9,3,15,20,7]
```

In-Order → Left-Child Node Right-Child

- The steps to perform the inorder traversal are listed as follows -
  - → First, visit the left subtree.
  - → Then, visit the root node.
  - → At last, visit the right subtree.

```java
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {

        List<Integer> list = new ArrayList<>();
        helper(root,list);
        return list;
    }

    public void helper(TreeNode root,List<Integer> list ){

        if(root == null){
            return;
        }

        helper(root.left,list);
        list.add(root.val);
        helper(root.right,list);
    }
}
```

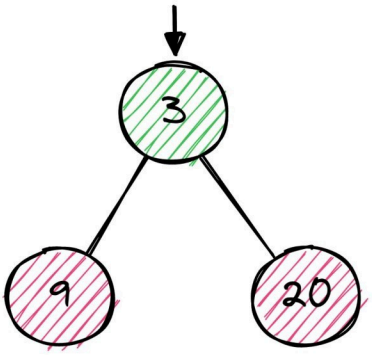


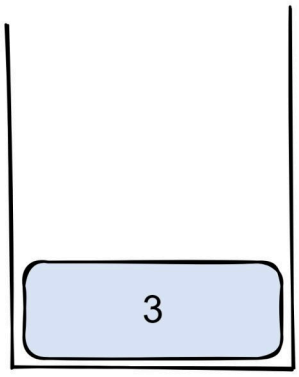
Note: we can also solve this problem using Stack.

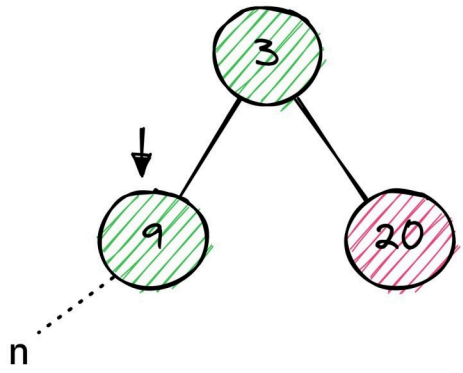
Approach-2 (Using Stack)

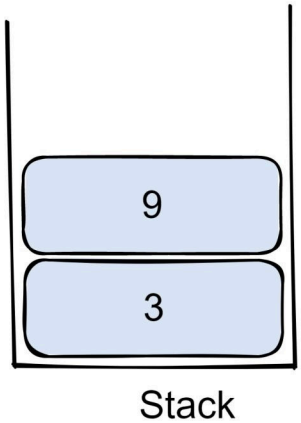
→ The strategy is very similiar to the first method, the different is using stack.

- Initialize an empty stack.
- Push the current node (starting from the root node) onto the stack. Continue pushing nodes to the left of the current node until a NULL value is reached.
- If the current node is NULL and the stack is not empty:
  - → Remove and print the last item from the stack.
  - → Set the current node to be the node to the right of the removed node.
  - → Repeat the second step of the algorithm.
- If the current node is NULL and the stack is empty, then the algorithm has finished.

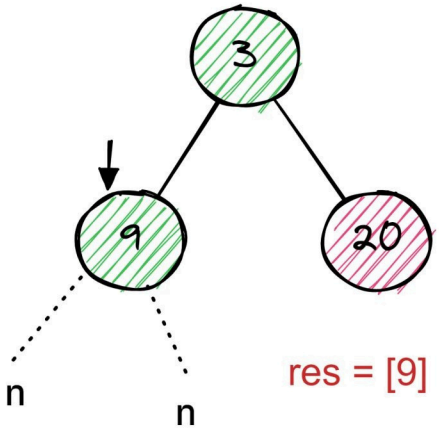




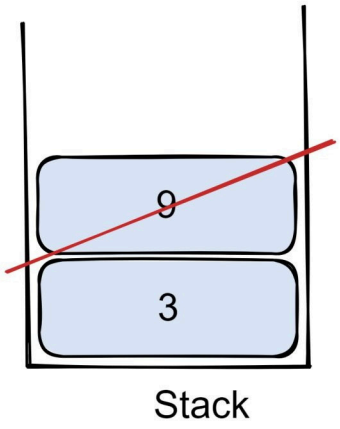




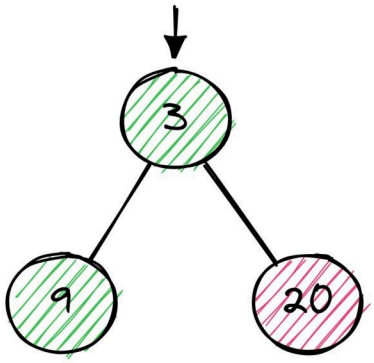
now, curr.left == null , so pop the top element from stack and store in result array.



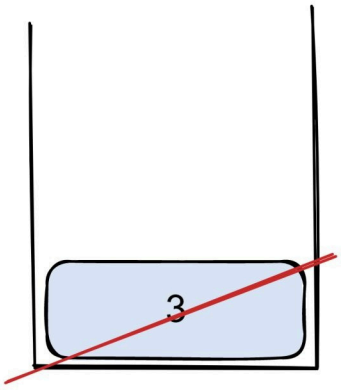
res = [9]

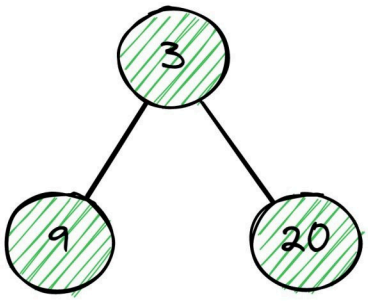


since, curr.right is also equals to null , pop the top element from stack and store to result.

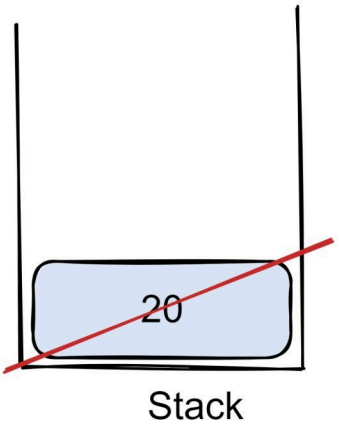


res = [9,3]



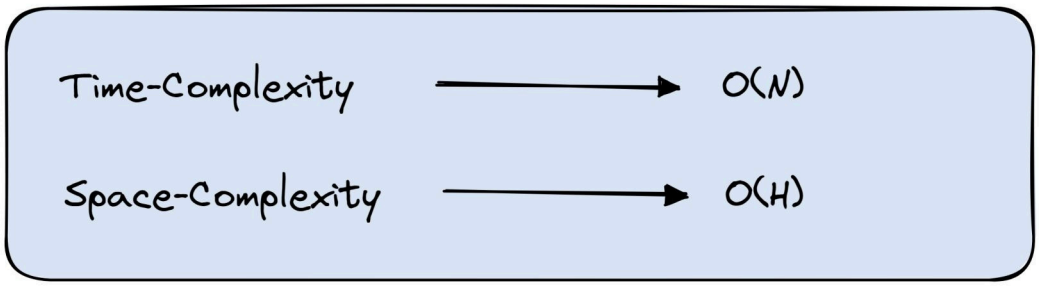


res = [9,3,20]



since, curr element became empty and stack is empty. Then, just return our result array.

```java
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {

        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode curr = root;

        while (curr != null || !stack.isEmpty()) {
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }
            curr = stack.pop();
            res.add(curr.val);
            curr = curr.right;
        }
        return res;
    }
}
```

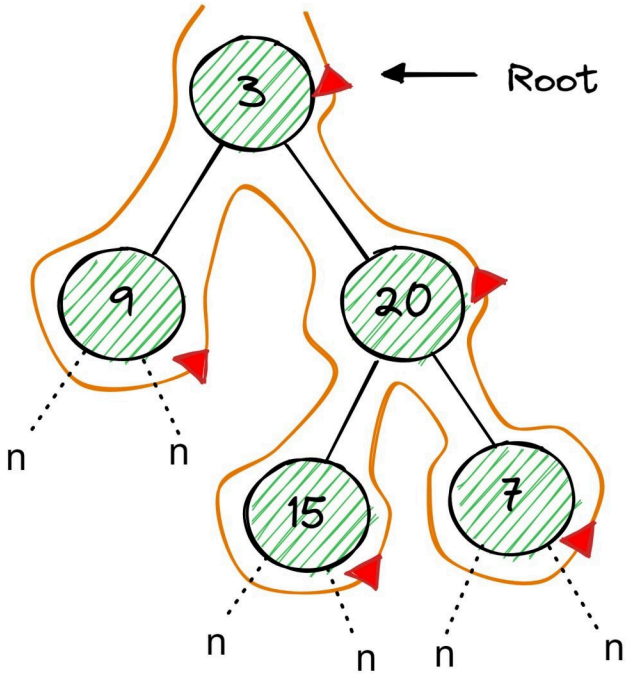


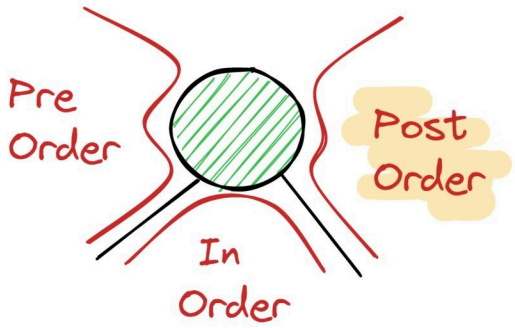
N → nodes
H → height

Binary Tree Postorder Traversal

Easy

→ Given the root of a binary tree, return the postorder traversal of its nodes' values.



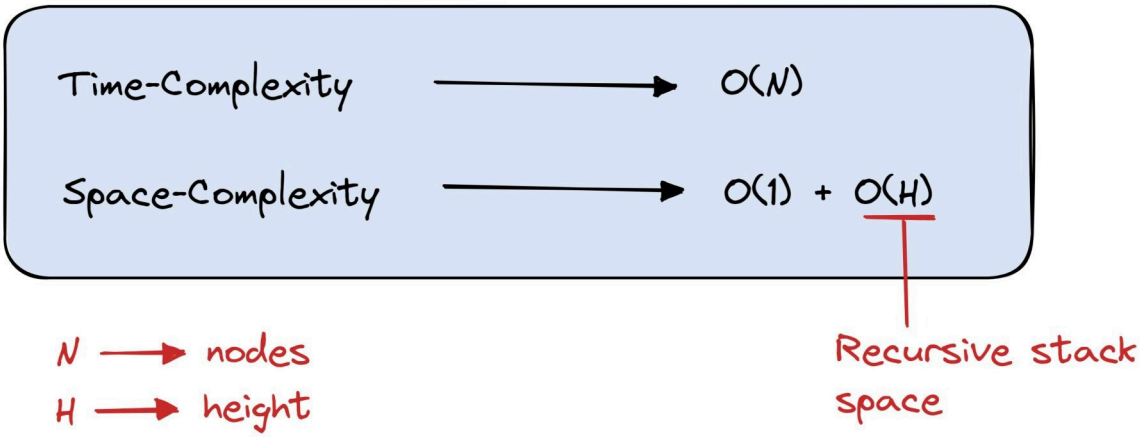



```
Input: root = [3,9,20,null,null,15,7]
Output: [9,15,7,20,3]
```

In-Order ⟶ Left-Child Right-Child Node

- The steps to perform the inorder traversal are listed as follows -
  - → First, visit the left subtree.
  - → Then, visit the right subtree.
  - → At last, visit the root node.

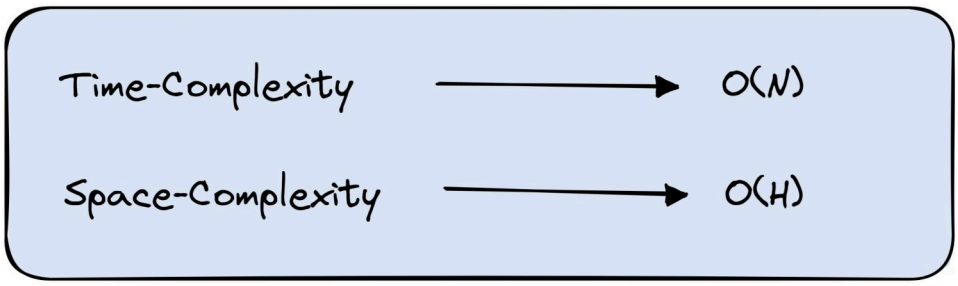
```java
class Solution {
    public List<Integer> postorderTraversal(TreeNode root){

        List<Integer> list = new ArrayList<>();
        postOrder(root,list);
        return list;
    }

    public void postOrder(TreeNode root,List<Integer> list){
        if(root == null){
            return;
        }
        postOrder(root.left,list);
        postOrder(root.right,list);
        list.add(root.val);
    }
}
```



Note: we can also solve this problem using Stack.

Approach-2 (using Stack)

```java
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {

        ArrayList<Integer> ans = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();

        if(root == null) return ans;
        stack.push(root);

        while(!stack.empty()){

            TreeNode currNode = stack.pop();
            ans.add(currNode.val);

            if(currNode.left != null) stack.push(currNode.left);
            if(currNode.right != null) stack.push(currNode.right);
        }
        Collections.reverse(ans);
        return ans;
    }
}
```

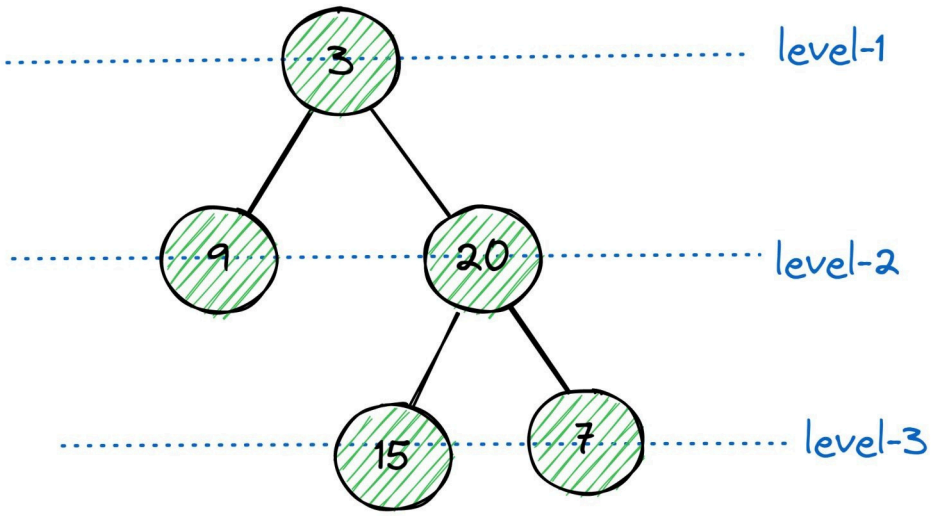


N → nodes
H → height

Binary Tree Level Order Traversal

Medium

→ Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).



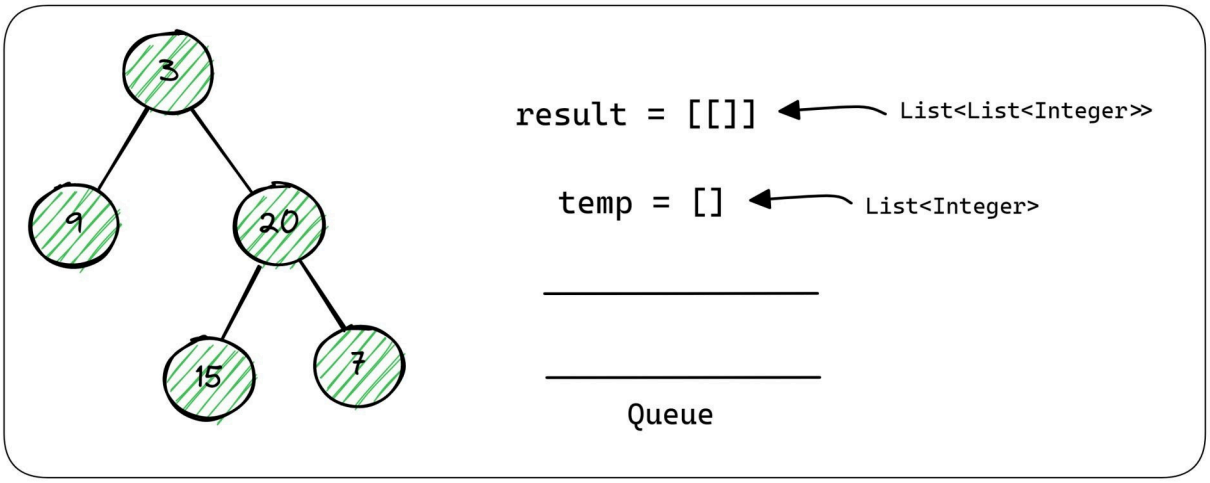

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

→ We are going to use Queue to perform level order traversal.

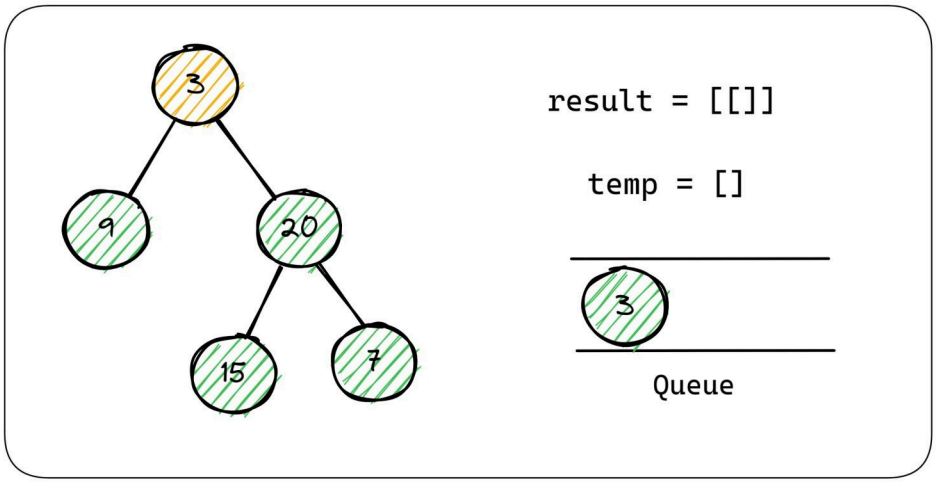
Follow the below steps to Implement the above idea:

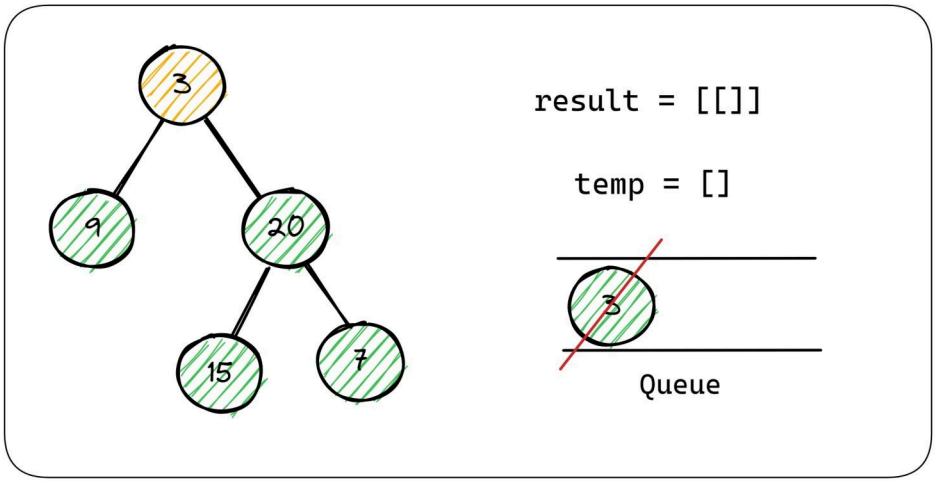
- Create an empty queue q and push root in q.
- Run While loop until q is not empty.
  - Initialize node = q.pop() and store node.data.
  - Push temp_node's children i.e. temp_node.left then temp_node.right to q
  - Pop front node from q.

in Santosh Kumar Mishra

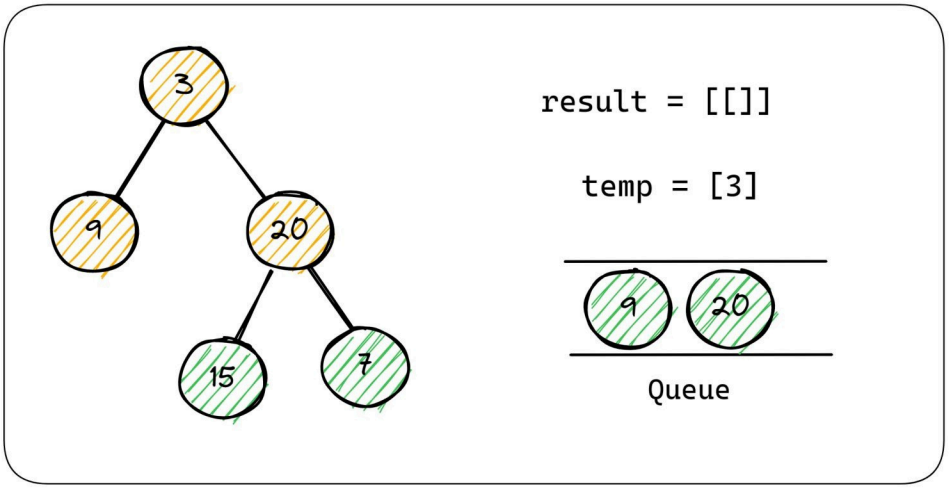


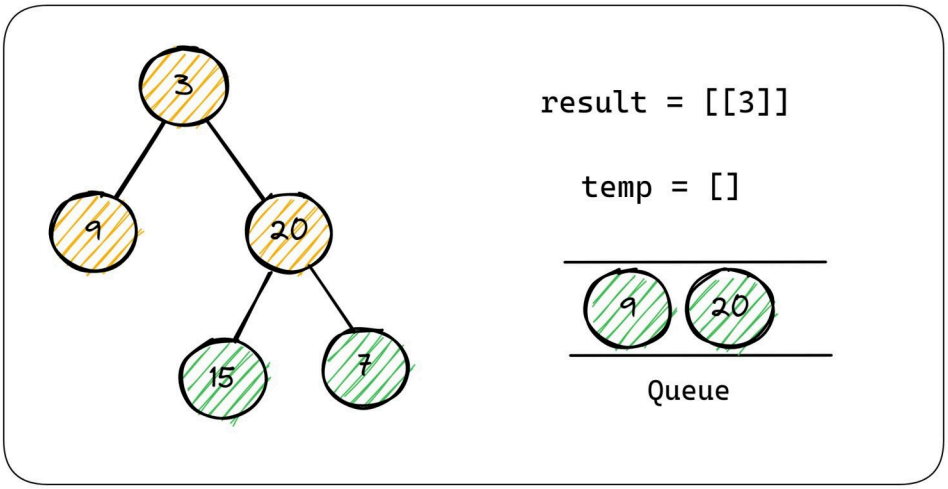
- Initially we are going to add root Node 3 in our Queue.
  Now, run a while loop until queue became empty.



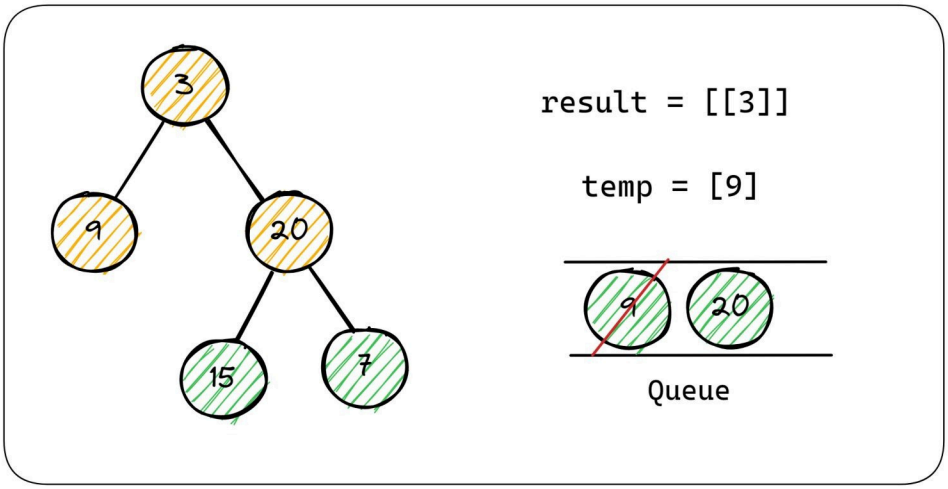


- Dequeue the top element from the queue, and store it in a temp ArrayList<Integer> , And add that top element right & left children in a queue.

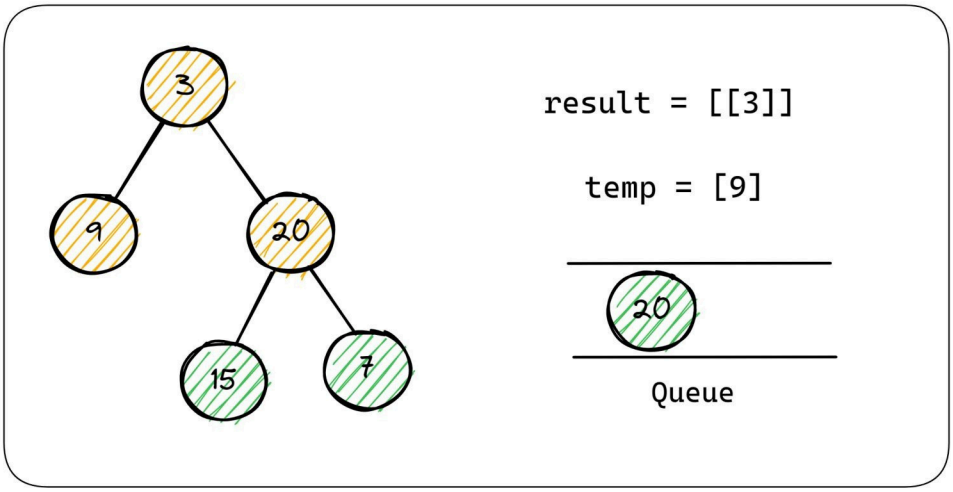




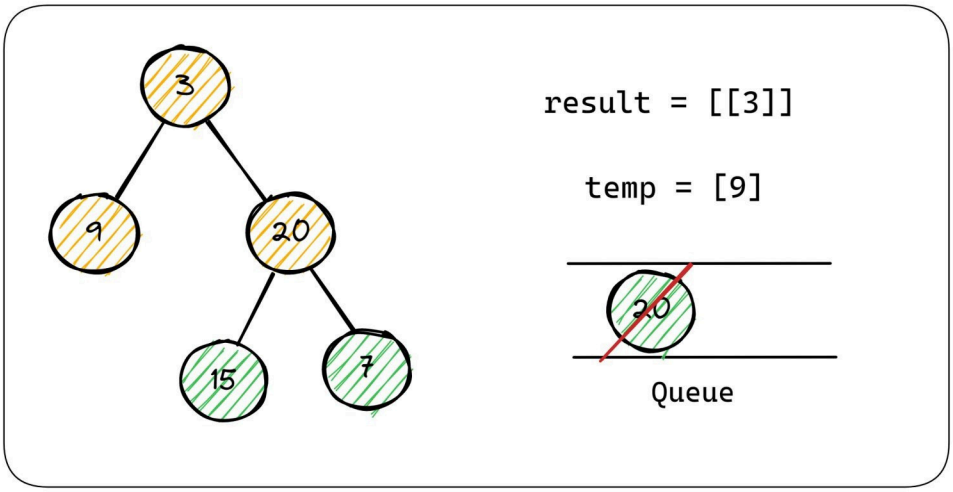
- Now , dequeue node 9 from the queue and add its children in queue. and store node 9 in a temp arraylist.

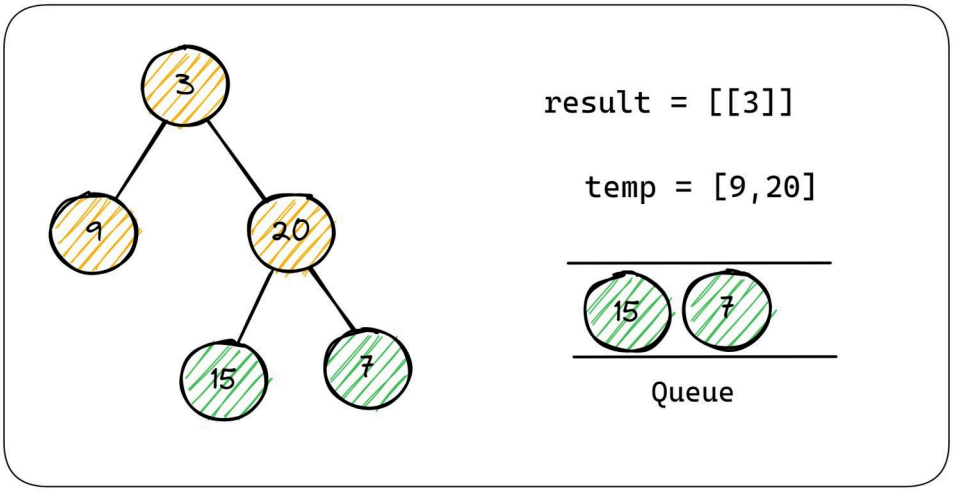


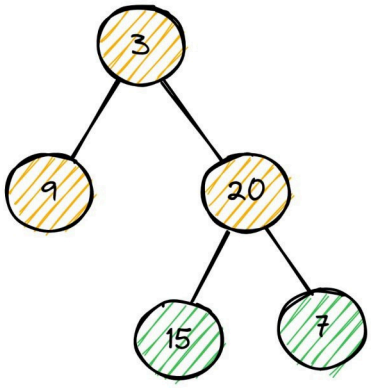
- Since node 9 left & right children are null we are going to move further with another element in our queue.



- Now , dequeue node 20 from the queue and add its children in queue. and store node 20 in a temp arraylist.

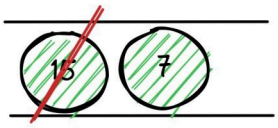




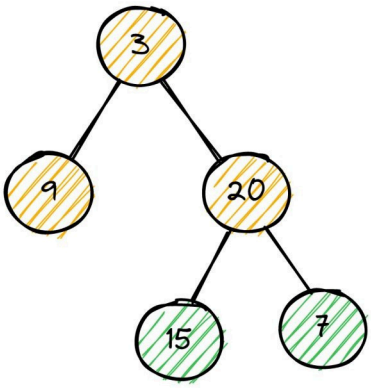


result = [[3],[9,20]]

temp = []



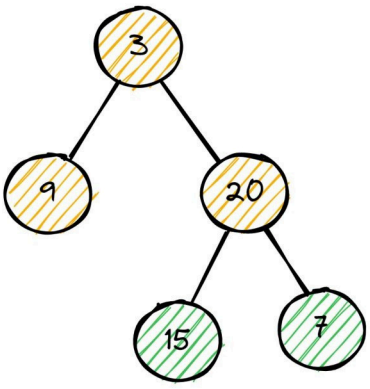
Queue



result = [[3],[9,20]]

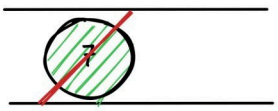
temp = [15]

Queue

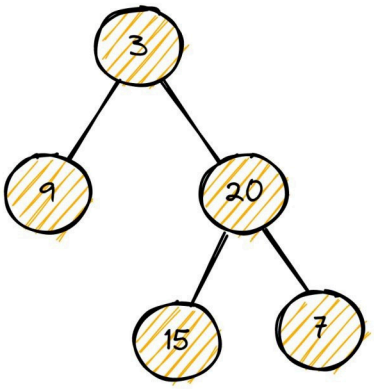


result = [[3],[9,20]]

temp = [15]



Queue



result = [[3],[9,20],[15,7]]

temp = []

Queue

```java
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {

        List<List<Integer>> res = new ArrayList<>();
        if(root == null) return res;

        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);

        while(!q.isEmpty()){

            int current_size = q.size();
            List<Integer> temp = new ArrayList<>();

            while(current_size > 0){

                TreeNode node = q.remove();
                temp.add(node.val);

                if(node.left != null) q.add(node.left);
                if(node.right != null) q.add(node.right);

                current_size--;
            }
            res.add(temp);
        }
        return res;
    }
}
```

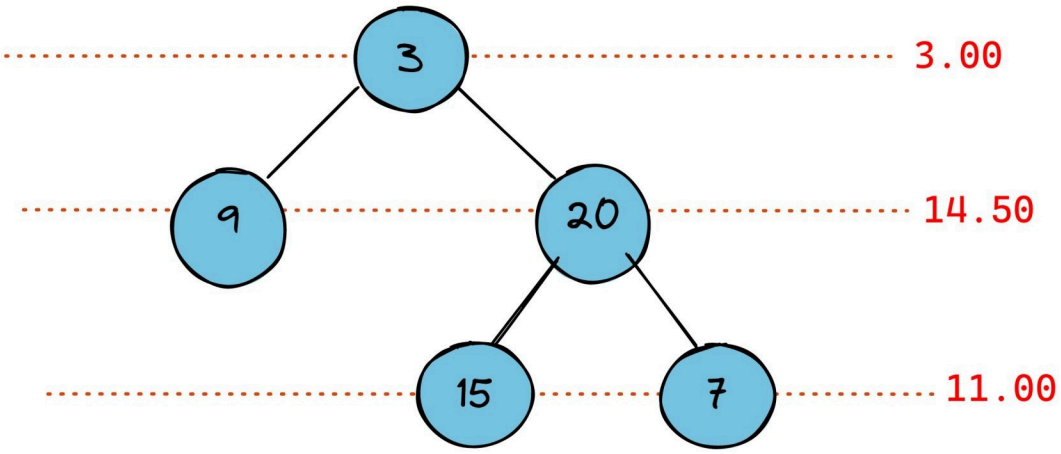
Time-Complexity → O(N)

Space-Complexity → O(N)

Average of Levels in Binary Tree

Easy

Given the root of a binary tree, return the average value of the nodes on each level in the form of an array.




```
Input: root = [3,9,20,null,null,15,7]

Output: [3.00000,14.50000,11.00000]
```

→ Explanation: The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11.

Hence return [3, 14.5, 11].

Approach-1 (BFS)

→ This problem follows the Binary Tree Level Order Traversal pattern. We can follow the same BFS approach.

→ The only difference will be that instead of keeping track of all nodes of a level, we will only track the running sum of the values of all nodes in each level.

→ In the end, we will append the average of the current level to the result array.

Santosh Kumar Mishra

```java
class Solution {
    public List<Double> averageOfLevels(TreeNode root) {

        List<Double> list = new ArrayList<>();
        Queue<TreeNode> q = new LinkedList<>();

        q.add(root);
        Double avg = 0.0000;

        while(!q.isEmpty()){

            int size = q.size();

            for(int i = 0;i<size;i++){

                TreeNode node = q.remove();
                avg += node.val;

                if(node.left != null) q.add(node.left);
                if(node.right != null) q.add(node.right);
            }
            list.add(avg/size);
            avg = 0.000;
        }
        return list;
    }
}
```

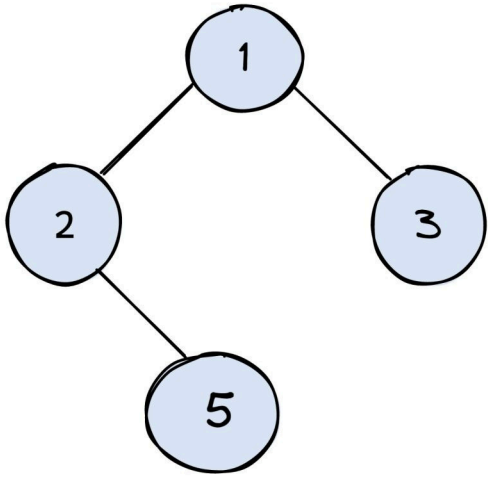
- This algorithm has a time complexity of O(n), where n is the number of nodes in the tree, because it processes each node exactly once.

- It also has a space complexity of O(n), because at worst case, the queue will contain all the nodes in the tree.

Binary Tree Paths

Easy

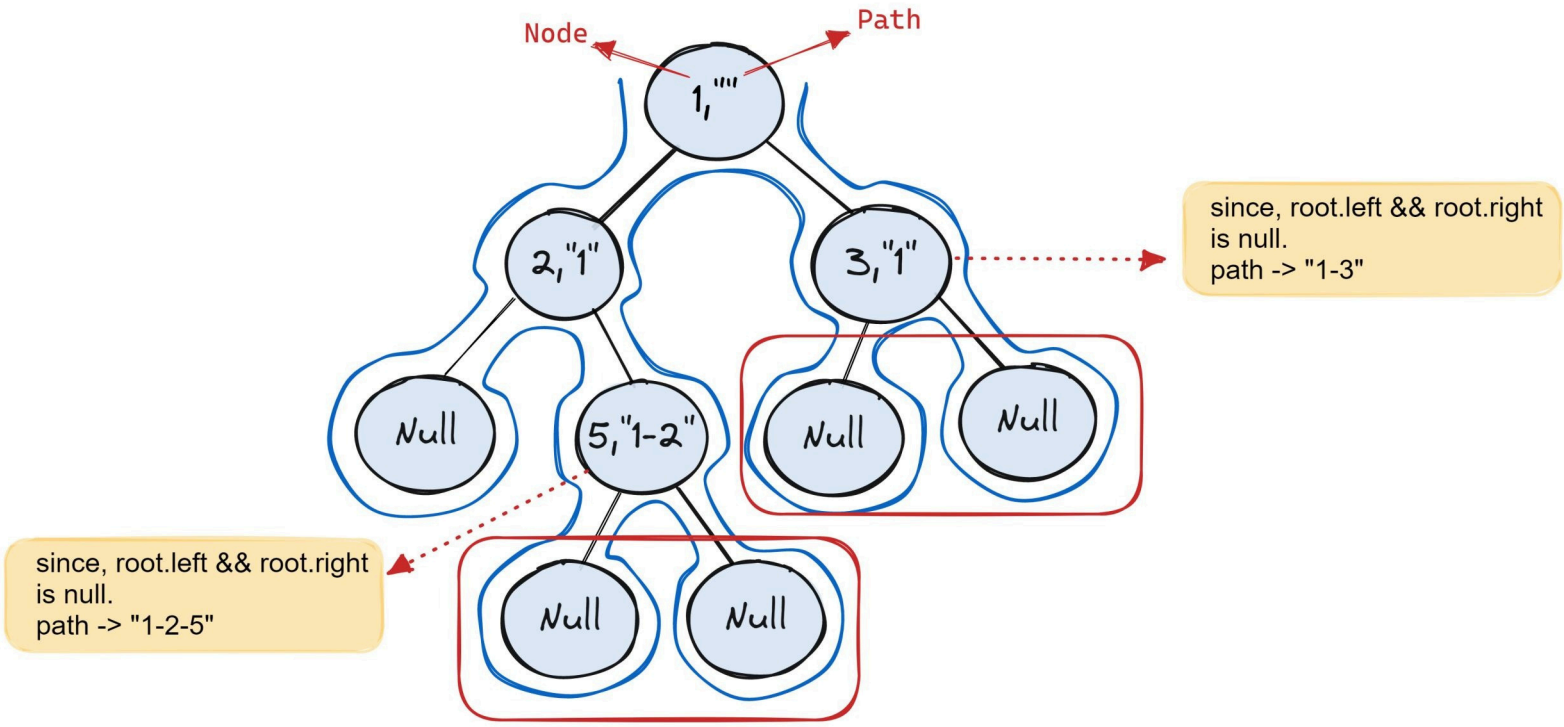
→ Given the root of a binary tree, return all root-to-leaf paths in any order. A leaf is a node with no children.




```
Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]
```

Approach (Using DFS)

1. Define a list res to store the paths, and a helper function pathFinder that takes in a node, the list res, and a path.
2. Inside the pathFinder function, check if the node is null. If it is, then return.
3. If the node is not null, check if it is a leaf. If it is, append the path + node's value to the list res.
4. if node left child exists then, call pathfinder for left child. (path + root.val + "->").
5. if node right child exists then, call pathfinder for right child. (path + root.val + "->").
6. Return the list res.




```java
class Solution {
    public List<String> binaryTreePaths(TreeNode root) {

        ArrayList<String> res = new ArrayList<>();
        pathFinder(root,res,"");
        return res;
    }

    public void pathFinder(TreeNode root,ArrayList<String> res,String path){

        if(root == null) return;

        if(root.left == null && root.right == null){
            res.add(path + root.val);
        }

        if(root.left!=null){
            pathFinder(root.left,res,path + root.val + "->");
        }
        if(root.right!=null){
            pathFinder(root.right,res,path + root.val + "->");
        }
    }
}
```

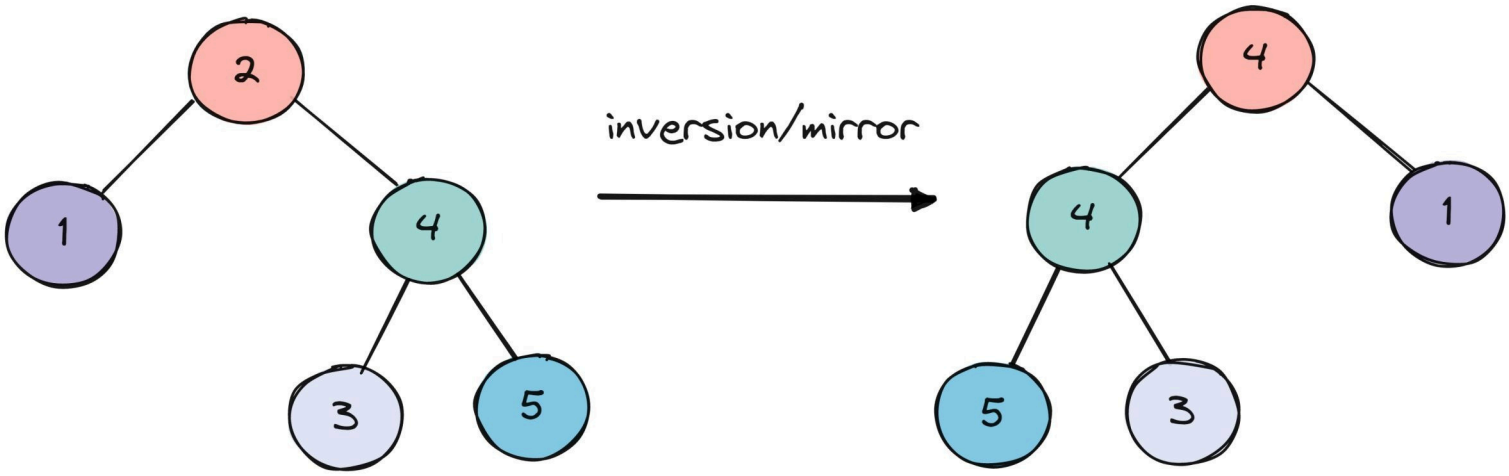
- This algorithm has a time complexity of O(n), where n is the number of nodes in the tree, because it processes each node exactly once.
- It also has a space complexity of O(n), because at worst case, the queue will contain all the nodes in the tree.

Santosh Kumar Mishra

Invert Binary Tree

Easy

→ Given the root of a binary tree, invert the tree, and return its root.



For this solution, we do a pre-order traversal of the binary tree. The algorithm is as follows:

→ The solution is a simple recursive approach:

- Call invert for left-subtree.
- Call invert for right-subtree.
- Swap left and right subtrees.

Code:- Using DFS

```java
class Solution {
    public TreeNode invertTree(TreeNode root) {

        if(root == null){
            return null;
        }
        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;

       invertTree(root.left);
        invertTree(root.right);

        return root;
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(H)

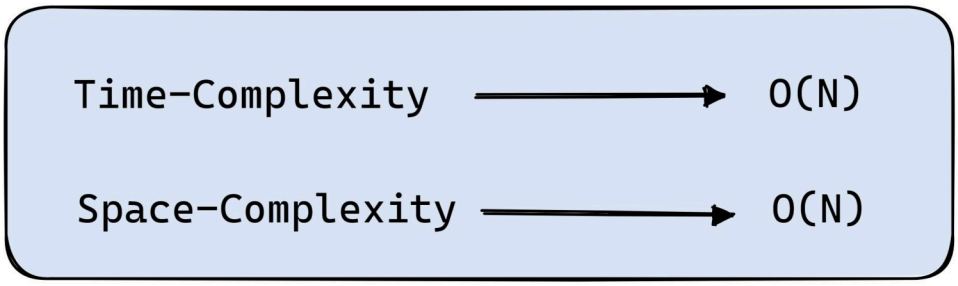
➙ where `N` is the number of nodes in the tree.

➙ `H` because of recursive stack space.

Code:- Using BFS

→ The idea is to do queue-based level order traversal. While doing traversal, swap left and right children of every node.

```java
class Solution {
    public TreeNode invertTree(TreeNode root) {

        if (root == null) return null;

        Queue<TreeNode> q = new LinkedList<TreeNode>();
        queue.add(root);

        while (!q.isEmpty()) {

            TreeNode current = q.poll();

            TreeNode temp = current.left;
            current.left = current.right;
            current.right = temp;

            if (current.left != null) q.add(current.left);
            if (current.right != null) q.add(current.right);
        }
        return root;
    }
}
```



→ Time-Complexity for Traversing over the tree of size N

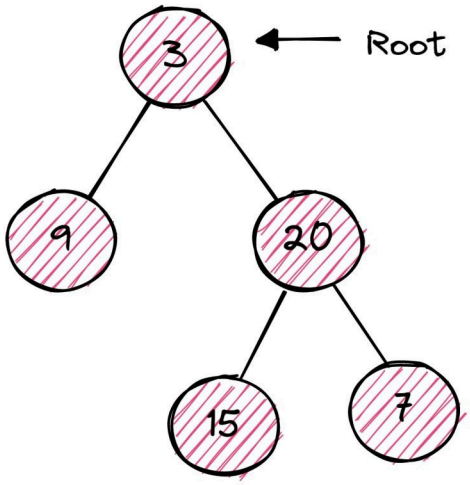
→ Space-Complexity Using queue to store the nodes of the tree

Maximum Depth of Binary Tree

Easy

→ Given the root of a binary tree, return its maximum depth.

→ A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.




```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

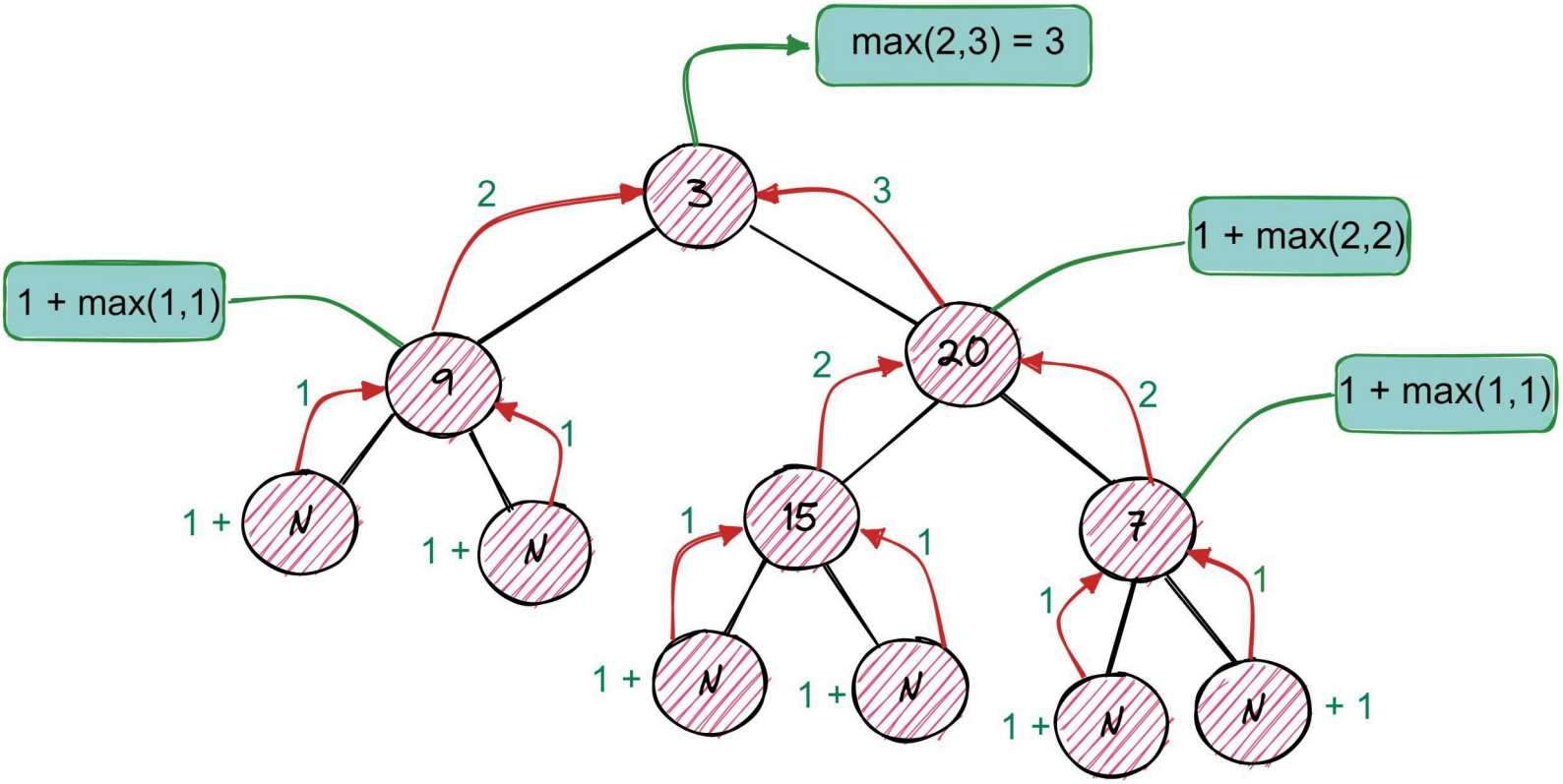
Approach (DFS)

- The algorithm uses recursion to calculate the maximum height:

  → Recursively calculate the height of the tree to the left of the root and add one to it.

  → Recursively calculate the height of the tree to the right of the root and add one to it.

  → Pick the larger height from the two answers.



Time-Complexity	⟶	O(N)
Space-Complexity	⟶	O(1) + O(H)

N ⟶ nodes
H ⟶ height

Recursive stack space

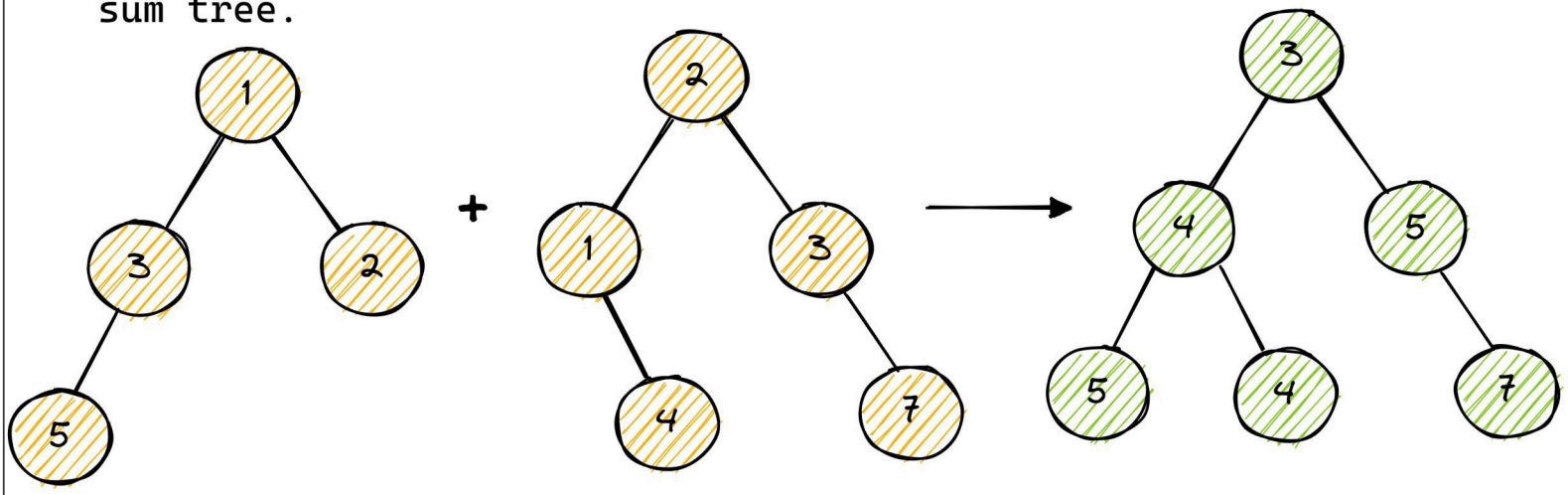
Note: we can also solve this problem using BFS(Breadth first search) technique.

```java
class Solution {
    public int maxDepth(TreeNode root) {
        return findDepth(root);
    }
    public int findDepth(TreeNode root){
        if(root == null){
            return 0;
        }
        int left = 1 + findDepth(root.left);
        int right = 1 + findDepth(root.right);

        return Math.max(left,right);
    }
}
```

Merge two sorted Binary Tree

Easy

→ You are given two binary trees root1 and root2. return root of the sum tree.



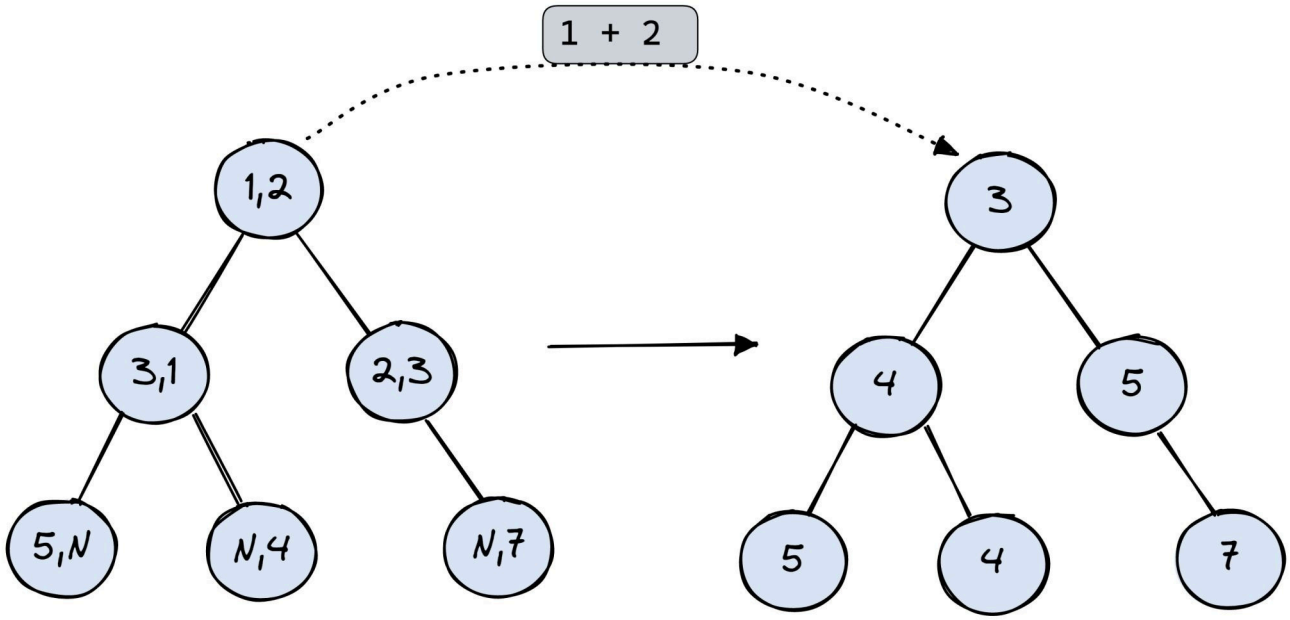

```
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

Output: [3,4,5,5,4,null,7]
```

→ We can start traversing from the root node of both the trees (root1 and root2) in a preorder function.

→ At each step of the traversal, we will compare the corresponding nodes of the trees. We will use recursion for proceeding to the next nodes. We will call the same method recursively for both left and right subtrees.

```
root1.val = root1.val + root2.val
```

Santosh Kumar Mishra



Time-Complexity → O(m+n)

Space-Complexity → O(max(m,n))

m → root1 nodes

n → root2 nodes

```java
class Solution {
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2)
{
        if(root1 == null && root2 != null){
            return root2;
        }

        if(root2 == null && root1 != null){
            return root1;
        }

        if(root1 == null && root2 == null){
            return null;
        }

        root1.val += root2.val;

        root1.left = mergeTrees(root1.left,root2.left);
        root1.right = mergeTrees(root1.right,root2.right);

        return root1;
    }
}
```

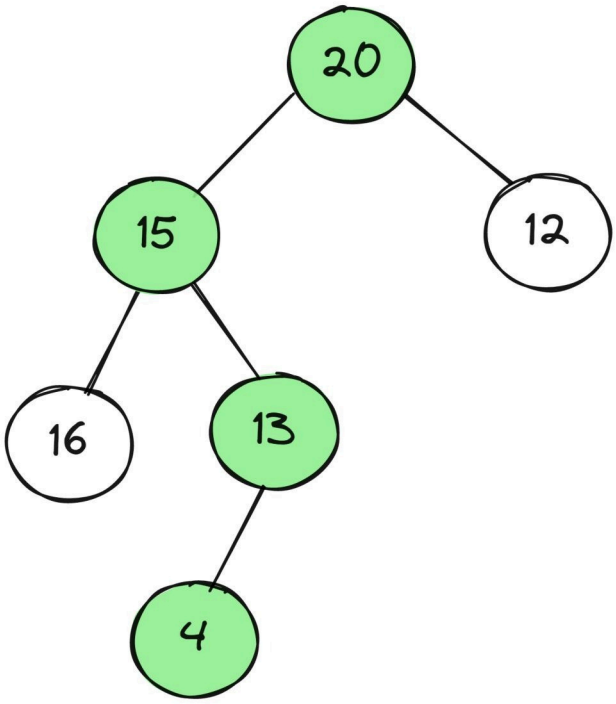
Santosh Kumar Mishra

Path Sum

Easy

→ Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.



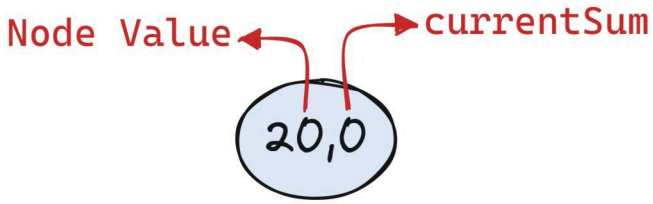
targetSum = 52

Output = True

- Follow the given steps to solve the problem using the above approach (Using DFS):

→ Recursively move to the left and right subtree and at each call increment the sum by the value of the current node.

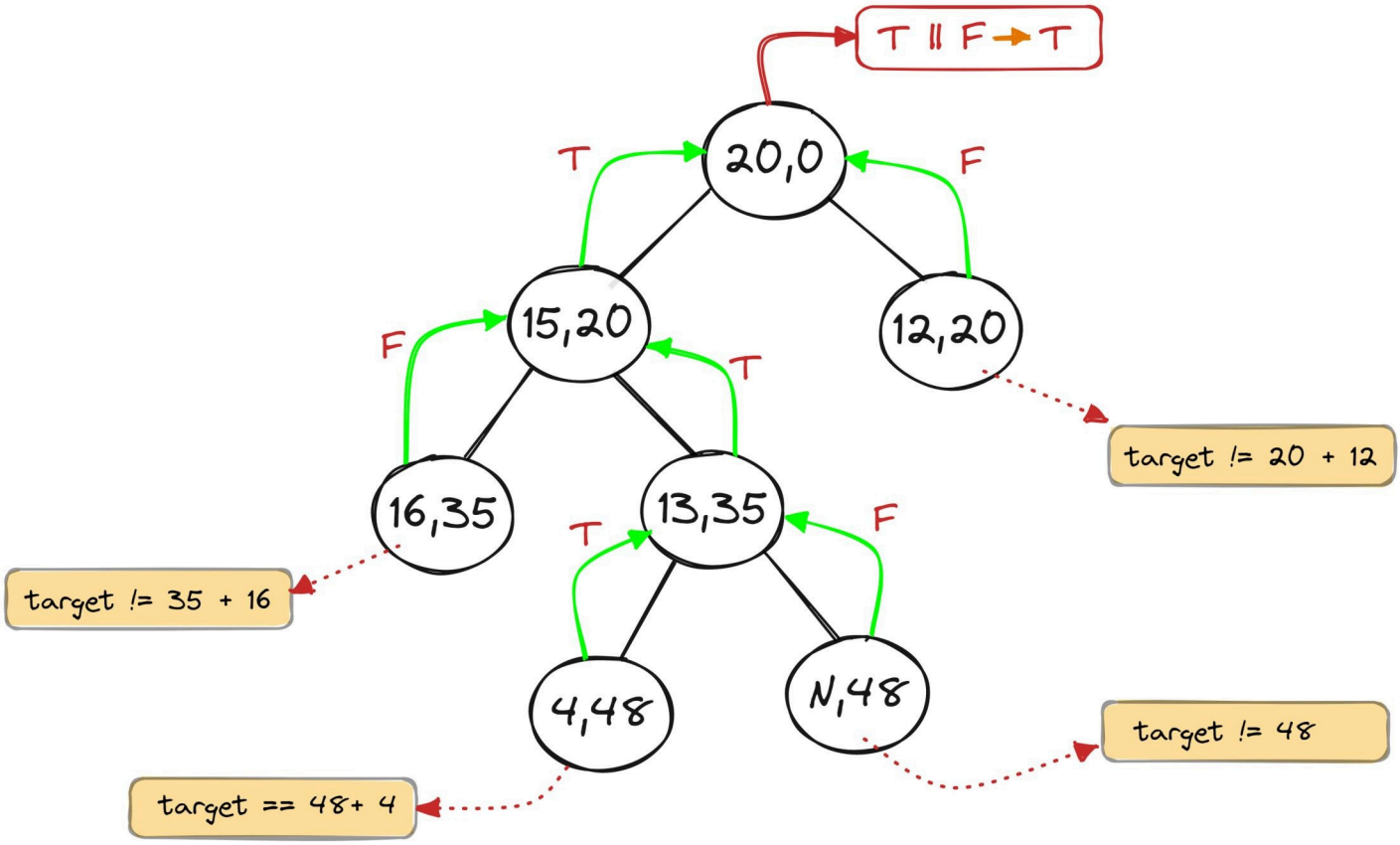
→ If at any level the current node is a leaf node and the remaining currentSum + Node Value is equal to target then return true.



Time-Complexity → O(N)

Space-Complexity → O(N)

in Santosh Kumar Mishra



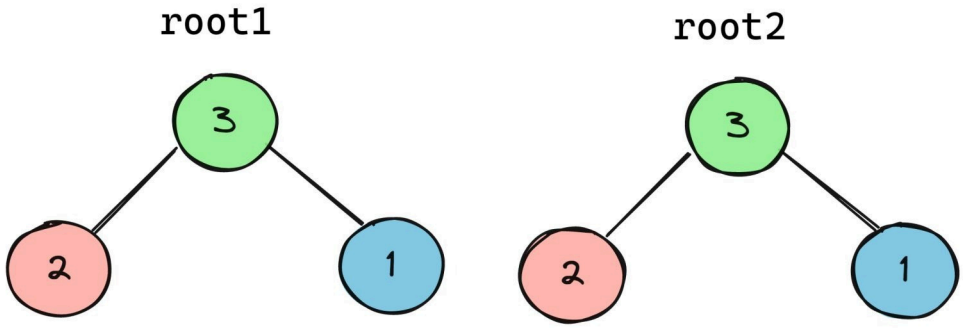

```java
class Solution {
    public boolean hasPathSum(TreeNode root, int targetSum) {

        return checkPath(root,0,targetSum);
    }

    public boolean checkPath(TreeNode root,int current, int target){

        if(root == null) return false;

        if(root.left == null && root.right == null){
             if(current + root.val == target){
               return true;
             }
             else{
               return false;
             }
         }
        boolean left = checkPath(root.left , current + root.val,target);
        boolean right = checkPath(root.right , current + root.val, target);

        return left || right;

    }
}
```

Same Tree

Easy

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.



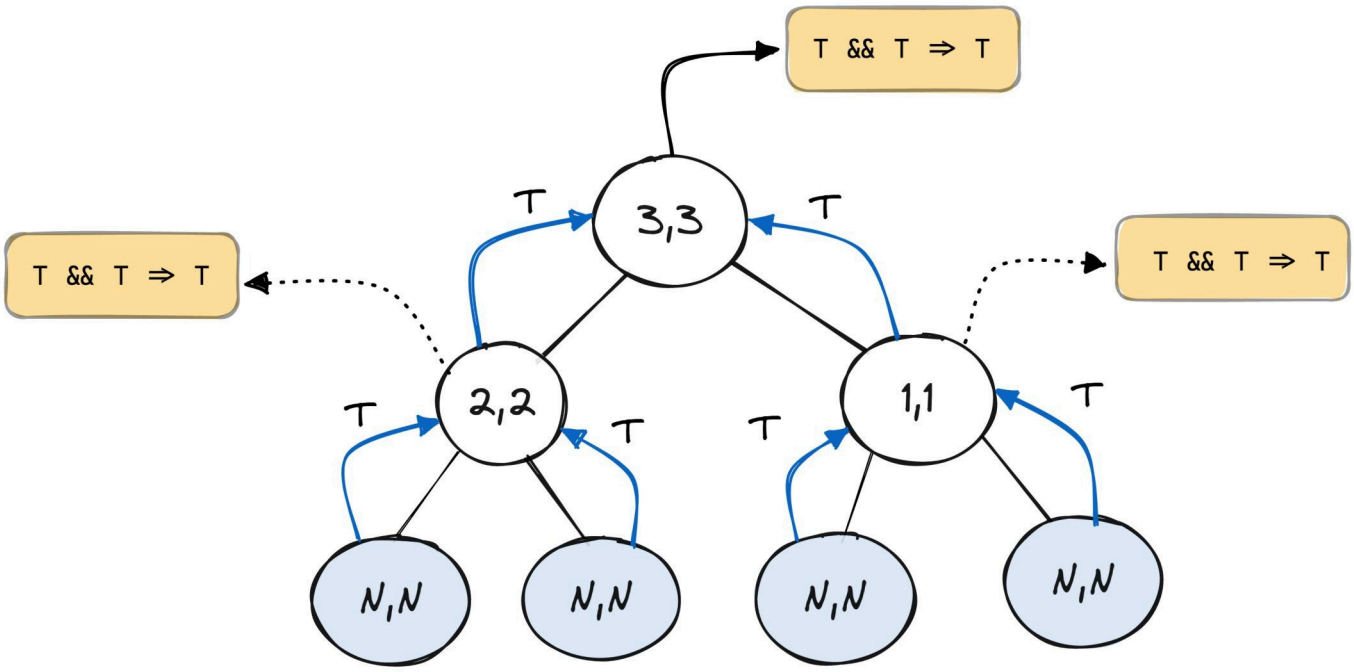

```
Input: p = [1,2,3], q = [1,2,3]

Output: true
```

Approach-1 (Using Recursion)

To solve the "Same Tree" problem we can use the following steps:

1. If both trees are empty, return true.
2. If one tree is empty and the other is not, return false.
3. If the values of the root nodes of the two trees are not equal, return false.
4. Recursively compare,
   - → left subtree of the first tree with the left subtree of the second tree.
   - → right subtree of the first tree with the right subtree of the second tree.

5. If both of the above recursive calls return true, return true. Otherwise, return false.



Time-Complexity ⟶ O(N)
Space-Complexity ⟶ O(N)

```java
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {

        if(p == null && q == null) return true;
        if(p == null || q == null) return false;

        if(p.val != q.val) return false;

        boolean leftAns = isSameTree(p.left,q.left);
        boolean rightAns = isSameTree(p.right,q.right);

        return (leftAns && rightAns);
    }
}
```

Approach-2 (Using BFS)

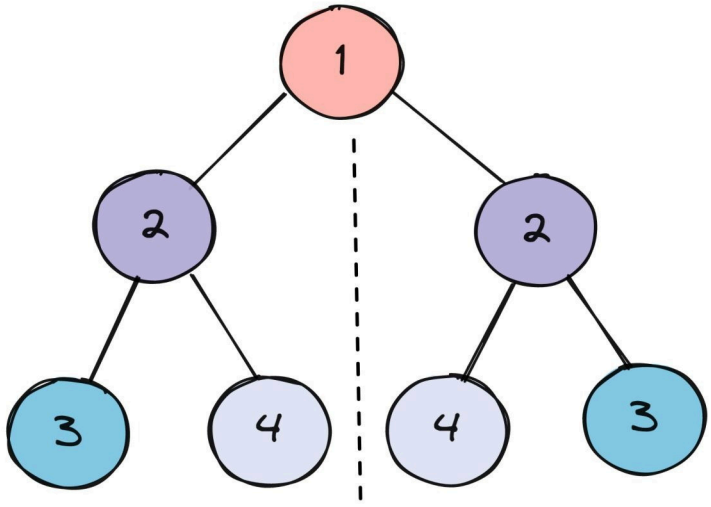
```java
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {

        if (p == null && q == null) return true;
        if (p == null || q == null) return false;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(p);
        queue.offer(q);

        while (!queue.isEmpty()) {

            TreeNode node1 = queue.poll();
            TreeNode node2 = queue.poll();

            if (node1 == null && node2 == null) continue;
            if (node1 == null || node2 == null) return false;

            if (node1.val != node2.val) return false;

            queue.offer(node1.left);
            queue.offer(node2.left);
            queue.offer(node1.right);
            queue.offer(node2.right);
        }

        return true;
    }
}
```

→ This solution has a time complexity of O(n) and a space complexity of O(n), where n is the number of nodes in the tree.

Symmetric Tree

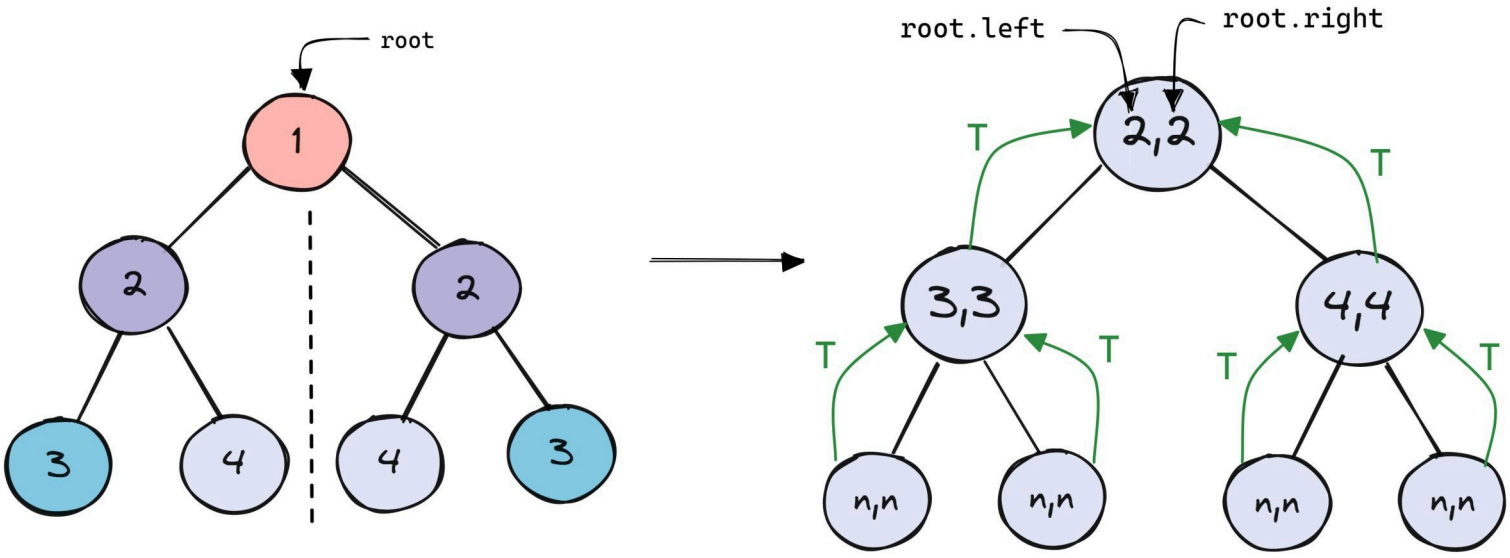
Easy

→ Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

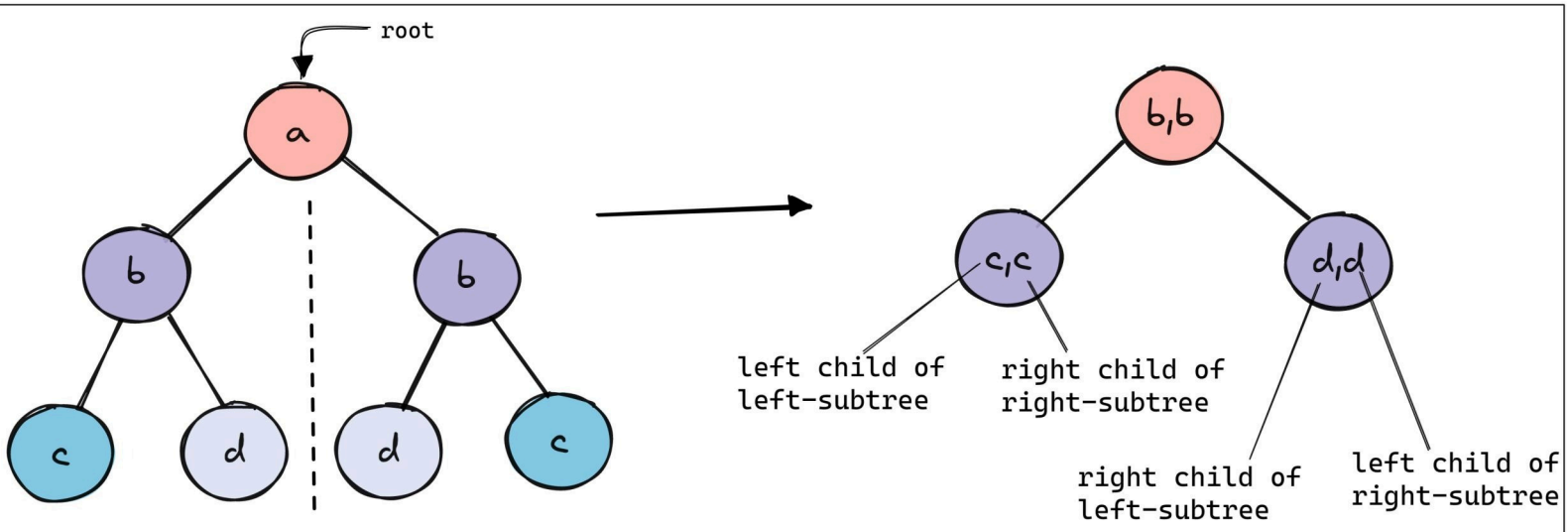



```
Input: root = [1,2,2,3,4,4,3]
Output: true
```

→ A binary tree is symmetric if it's left and right subtrees are identical-mirror images of each other.



in Santosh Kumar Mishra



→ Two trees are a mirror reflection of each other if:

- Their two roots have the same value.
- The right subtree of each tree is a mirror reflection of the left subtree of the other tree.

```java
class Solution {
    public boolean isSymmetric(TreeNode root) {

        if(root == null) return true;

        return solve(root.left,root.right);
    }

    public boolean solve(TreeNode root1,TreeNode root2){

        if(root1 == null && root2 == null) return true;
        if(root1 == null || root2 == null) return false;

        if(root1.val != root2.val) return false;

        boolean left = solve(root1.left,root2.right);
        boolean right = solve(root1.right,root2.left);

        return left && right;
    }
}
```

→ where `N` is the total number of nodes in the tree.

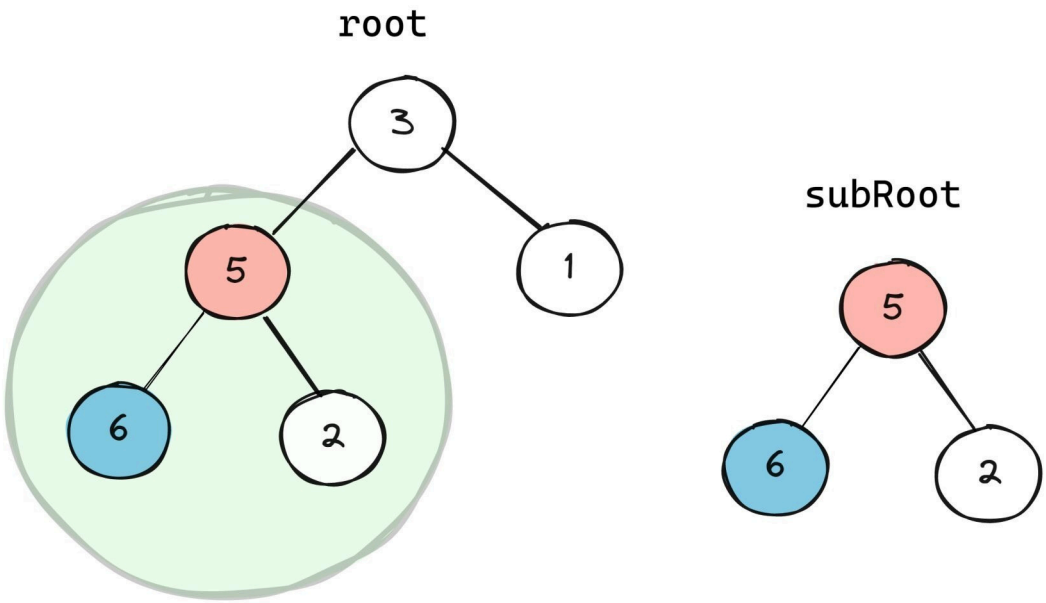
Time-Complexity → O(N)

Space-Complexity → O(N)

Subtree of Another Tree

Easy

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.




```
Input: root = [3,4,5,1,2], subRoot = [4,1,2]

Output: true
```

Approach:

1. If the root node of root is null, return false.
2. Check if subRoot is a subtree of root using the dfs function, which compares the values of the nodes at each level and returns true if they are the same, and false otherwise. If subRoot is a subtree of root, return true.
3. Recursively check the left and right subtrees of root to see if subRoot is a subtree of either of them. If subRoot is a subtree of either the left or right subtree, return true.
4. If subRoot is not a subtree of root, return false.

in Santosh Kumar Mishra

- We need to check for every node whether the subtree beneath it matches the subtree given in the question.
  - Case 1: root == null && subroot == null, return true.
    →If both the nodes are null then we need to return true.
  - Case 2:- root == null || subroot == null, return false.
    →If any one of the nodes is null and the other one is not null then we return false.
  - Case 3:- root.val ≠ subRoot.val , return false.
    →If both the nodes are not null then in that case we need to check for the value of the given two nodes.

```java
class Solution {
    public boolean isSubtree(TreeNode root, TreeNode subRoot) {

    if (root == null) return false;
    if (dfs(root, subRoot)) return true;

    return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
    }

   public boolean dfs(TreeNode root, TreeNode subRoot) {

    if (root == null && subRoot == null) return true;
    if (root == null || subRoot == null) return false;
    if (root.val != subRoot.val) return false;

    return dfs(root.left, subRoot.left) && dfs(root.right, subRoot.right);
  }

}
```

Time-Complexity → O(n * m)
Space-Complexity → O(n)

→ where n and m are the number of nodes in trees root and subRoot

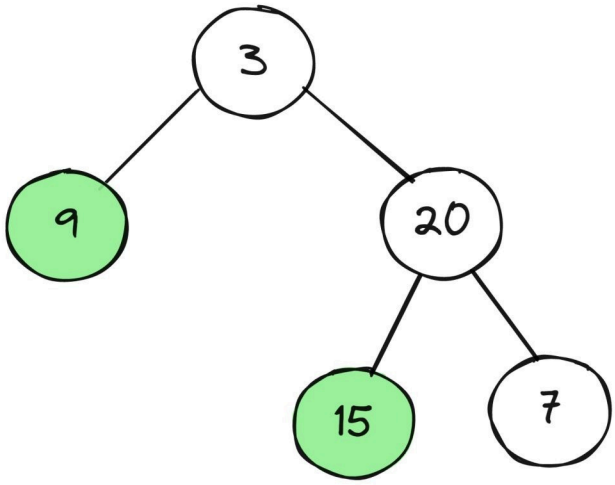
→ The space complexity is O(n) since the maximum depth of the recursion is n.

Sum of Left Leaves

Easy

→ Given the root of a binary tree, return the sum of all left leaves.

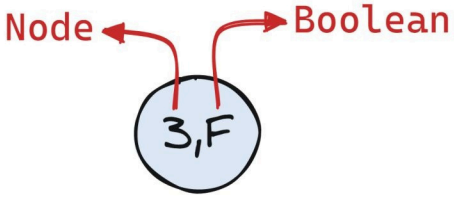
A leaf is a node with no children. A left leaf is a leaf that is the left child of another node.

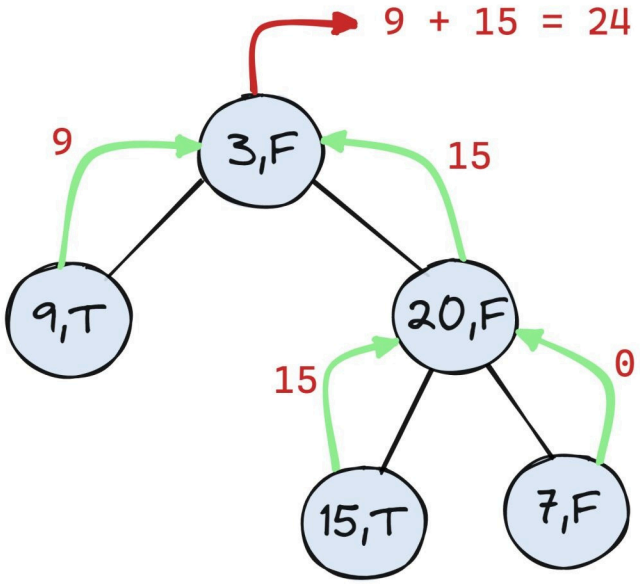



```
Input: root = [3,9,20,null,null,15,7]
Output: 24
Explanation: There are two left leaves in the binary
tree, with values 9 and 15 respectively.
```

→ We are going to use boolean to keep a track of right & left child.

if left child then mark it as a `True` else `False`.





in Santosh Kumar Mishra

Code → Using DFS

```java
class Solution {

    public int sumOfLeftLeaves(TreeNode root) {
        return leftSum(root,false);
    }

    public int leftSum(TreeNode root,boolean isLeft){

        if(root == null) return 0;

        if(root.left == null && root.right == null){
            if(isLeft){
                return root.val;
            }
            else{
                return 0;
            }
        }
        int left = leftSum(root.left,true);
        int right = leftSum(root.right,false);

        return left+right;
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

⟶ where `N` is the total number of nodes in the tree.

Santosh Kumar Mishra

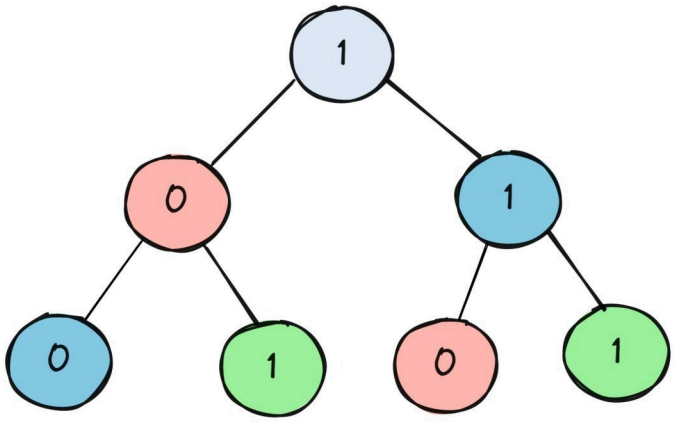
Sum of Root To Leaf Binary Numbers

Medium

- You are given the root of a binary tree where each node has a value 0 or 1. Each root-to-leaf path represents a binary number starting with the most significant bit.

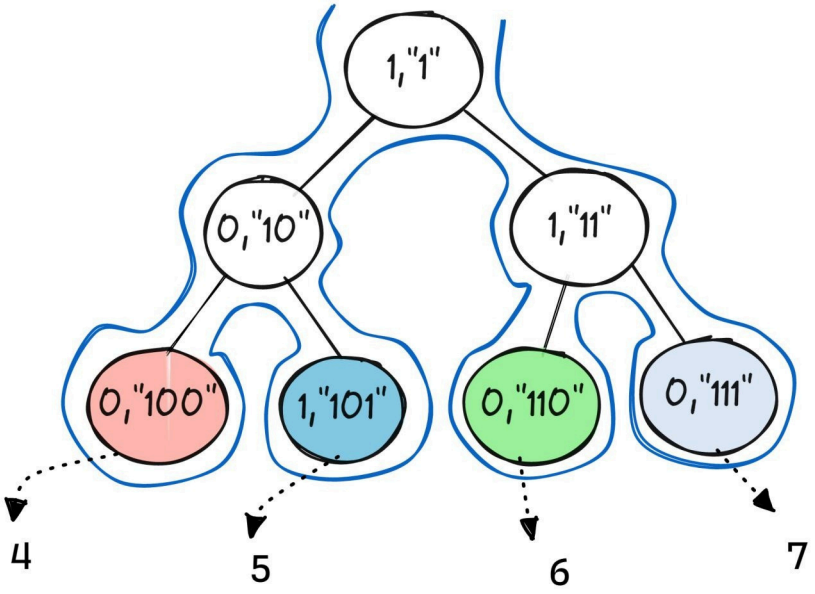
- For example, if the path is 0 → 1 → 1 → 0 → 1, then this could represent 01101 in binary, which is 13.

- For all leaves in the tree, consider the numbers represented by the path from the root to that leaf. Return the sum of these numbers.




```
Input: root = [1,0,1,0,1,0,1]
Output: 22
Explanation: (100) + (101) + (110) + (111) =
                4 + 5 + 6 + 7 = 22
```

in Santosh Kumar Mishra



→ initialize s = 0, if root became null then convert string to integer and add to s.

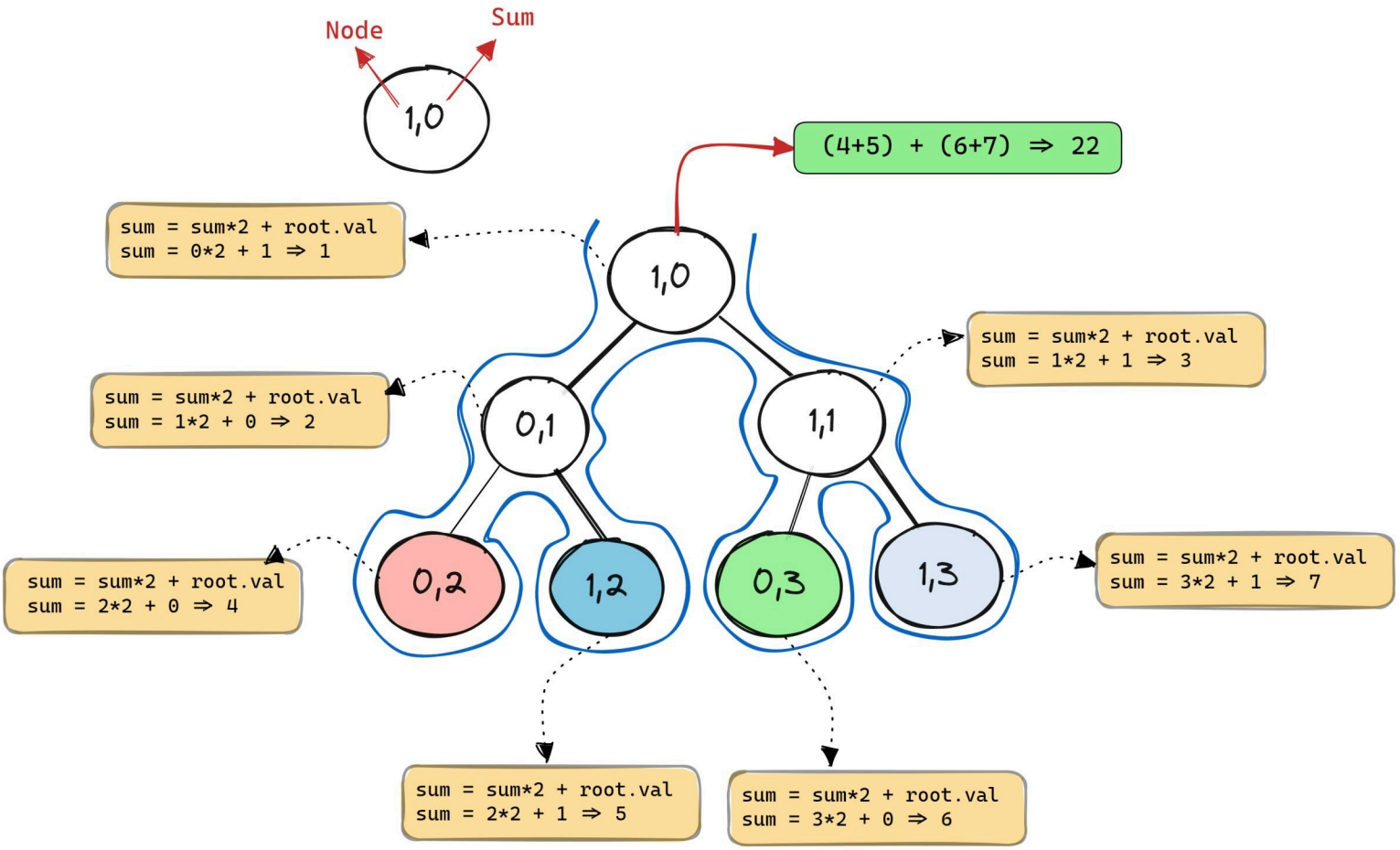
```java
class Solution {
    public int sumRootToLeaf(TreeNode root) {
        return sum(root, "");
    }

    public static int sum(TreeNode root , String sum){

        if(root.left==null && root.right == null){
            int s = Integer.parseInt(sum + root.val,2);
            return s;
        }

        if(root.left == null){
            return sum(root.right,sum + root.val);
        }
        if(root.right == null){
            return sum(root.left,sum + root.val);
        }

        int l = sum(root.left, sum + root.val);
        int r = sum(root.right,sum + root.val);

        return l + r ;
    }

}
```

Time-Complexity ⟶ O(N)
Space-Complexity ⟶ O(H)

Approach-2

In this problem, we will do a preorder traversal and keep track of the current sum from the root till the current node in a variable called curr .

At a particular node, the value of the current sum will be 2*currSum+node.val. When we reach any of the leaf nodes, we return te current sum value i.e curr, because we have completed a valid root to leaf path and we need a return and add this value to the overall result.



```java
class Solution {
    public int sumRootToLeaf(TreeNode root) {
        return sum(root, 0);
    }

    public static int sum(TreeNode root , int sum){

        if(root == null) return 0;

        sum = sum*2 + root.val;
        if(root.left == null && root.right == null){
            return sum;
        }

        int left = sum(root.left,sum);
        int right = sum(root.right,sum);

        return left+right;
    }
}
```

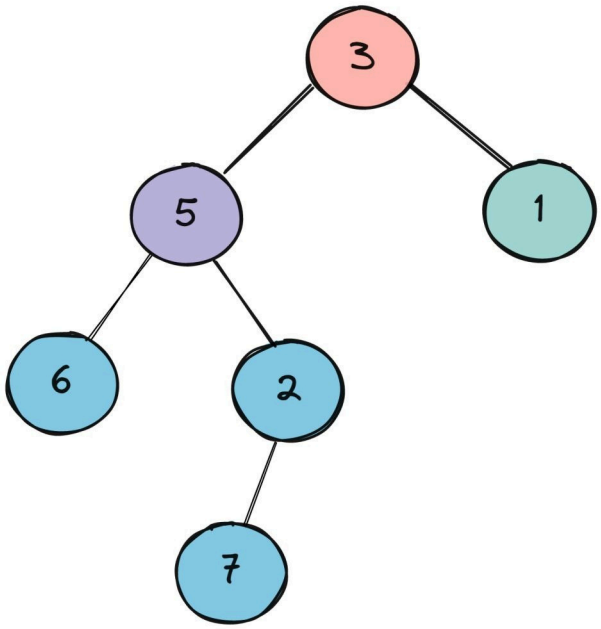
Time-Complexity ⟶ O(N)
Space-Complexity ⟶ O(H)

in Santosh Kumar Mishra

Lowest Common Ancestor

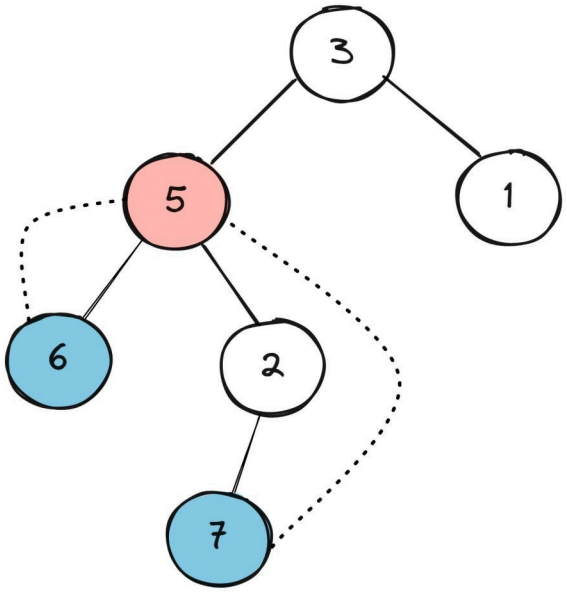
Medium

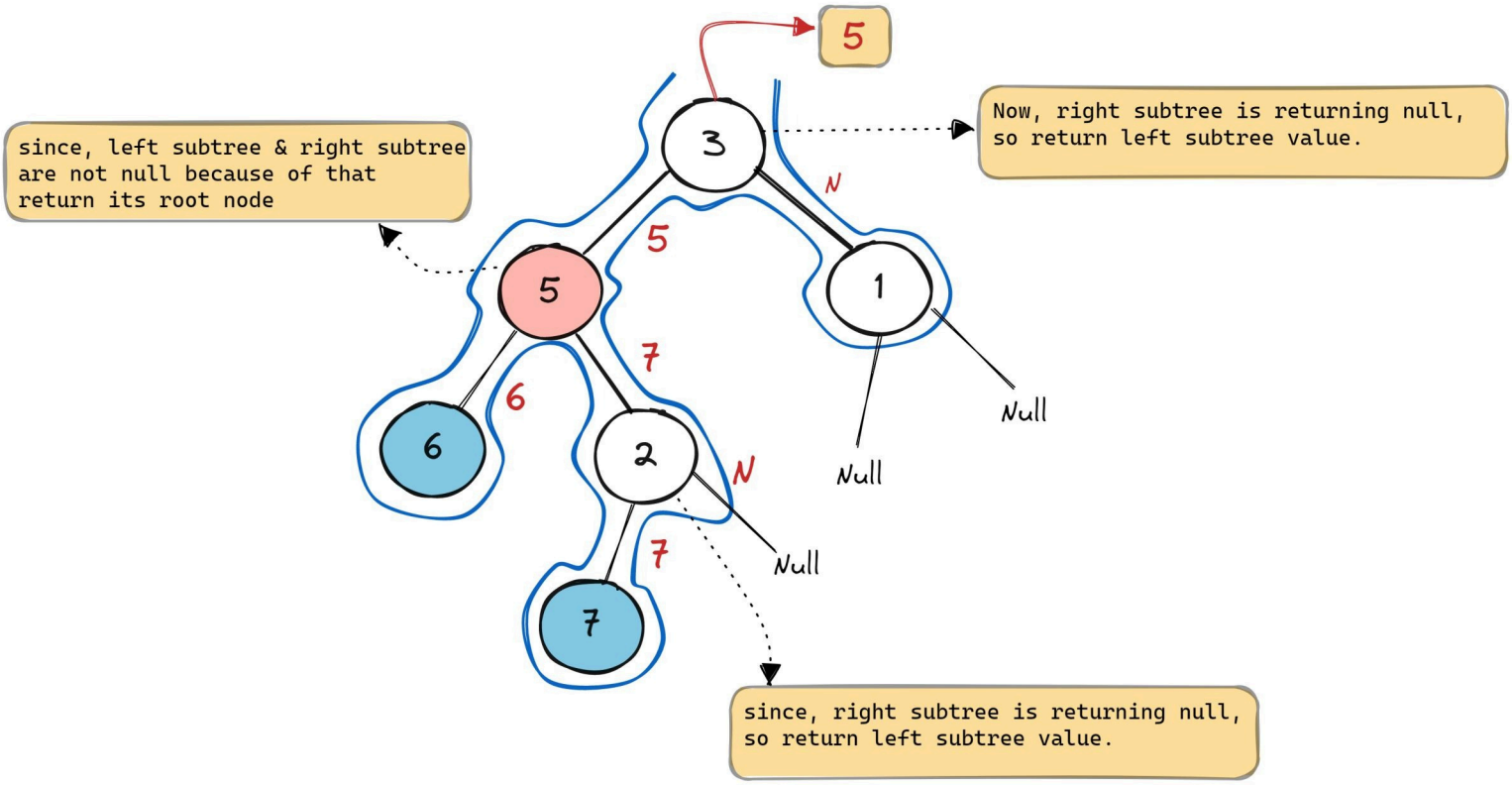
- Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
- The lowest common ancestor is defined between two nodes x and y as the lowest node in T that has both x and y as descendants (where we allow a node to be a descendant of itself.




```
Input: x = 6 , y = 7

Output: 5
```

- Explanation: All ancestors for 6,7 are 5,3. But we need Lowest Common ancestor, So we will consider lowest and also common ancestor that is 5






```java
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root,TreeNode p,TreeNode q){

        if(root == p || root == q || root == null){
            return root;
        }

        TreeNode left = lowestCommonAncestor(root.left,p,q);
        TreeNode right = lowestCommonAncestor(root.right,p,q);

        if(left == null){
            return right;
        }
        else if(right == null){
            return left;
        }
        return root;
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

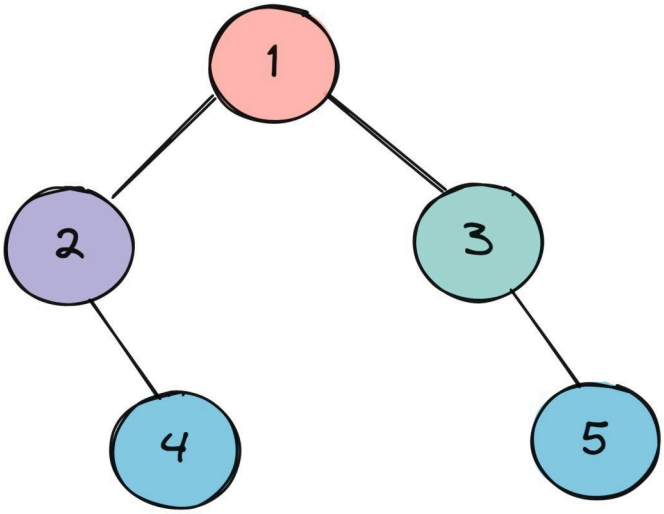
Santosh Kumar Mishra

Cousins in Binary Tree

Easy

→ Given the root of a binary tree with unique values and the values of two different nodes of the tree x and y, return true if the nodes corresponding to the values x and y in the tree are cousins, or false otherwise.

→ Two nodes are cousins if they have the same depth with different parents.



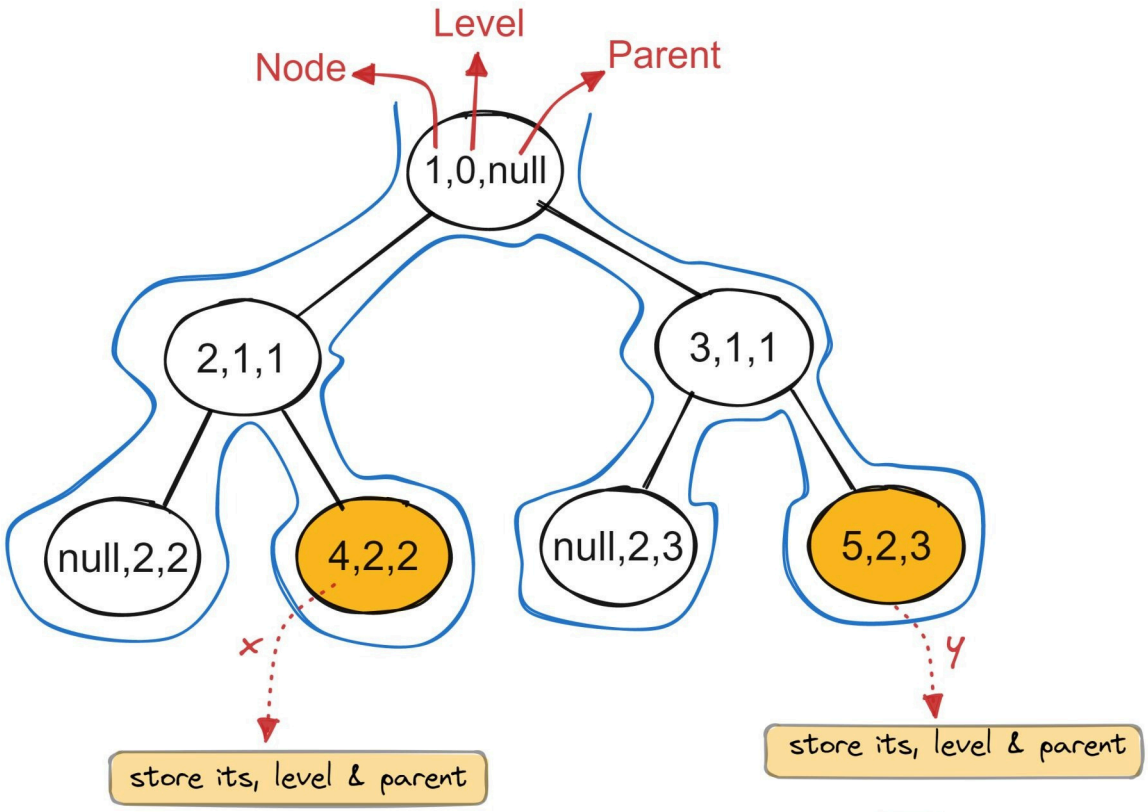
x = 4

y = 5

Output = True

● x & y are cousins if it follows two conditions:-

1. Having Different Parent.
2. Having Same Level.



in Santosh Kumar Mishra

Code:- Using DFS

```java
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {

        int[] level = {-1,-1};
        TreeNode[] parent = {null,null};

        findNodes(root,null,0,x,y,level,parent);

        if(level[0] == level[1] && parent[0] != parent[1]){
            return true;
        }
        return false;
    }

    public void findNodes(TreeNode root,TreeNode parentNode,int currentlevel,int
                x,int y, int[] level, TreeNode[] parent){

        if(root == null) return;

        if(root.val == x){
            level[0] = currentlevel;
            parent[0] = parentNode;
        }

         if(root.val == y){
            level[1] = currentlevel;
            parent[1] = parentNode;
        }

        findNodes(root.left,root,currentlevel+1,x,y,level,parent);
        findNodes(root.right,root,currentlevel+1,x,y,level,parent);
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

→ Space Complexity `O(N)` because of recursive stack space.

→ Time Complexity where `O(N)` is the number of nodes in the tree.

Santosh Kumar Mishra

Code:- Using BFS

```java
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {

        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);

        while(!q.isEmpty()){

            int size = q.size();
            boolean Aexist = false;
            boolean Bexist = false;

            for(int i = 0;i<size;i++){

                TreeNode curr = q.remove();

                if(curr.val == x) Aexist = true;
                if(curr.val == y) Bexist = true;

                if(curr.left != null && curr.right != null){
                    if(curr.left.val == x && curr.right.val == y){
                            return false;
                    }
                    if(curr.left.val == y && curr.right.val == x){
                            return false;
                    }
                }
                if(curr.left != null) q.add(curr.left);
                if(curr.right != null) q.add(curr.right);
            }
            if(Aexist && Bexist) return true;
        }
        return false;
    }
}
```

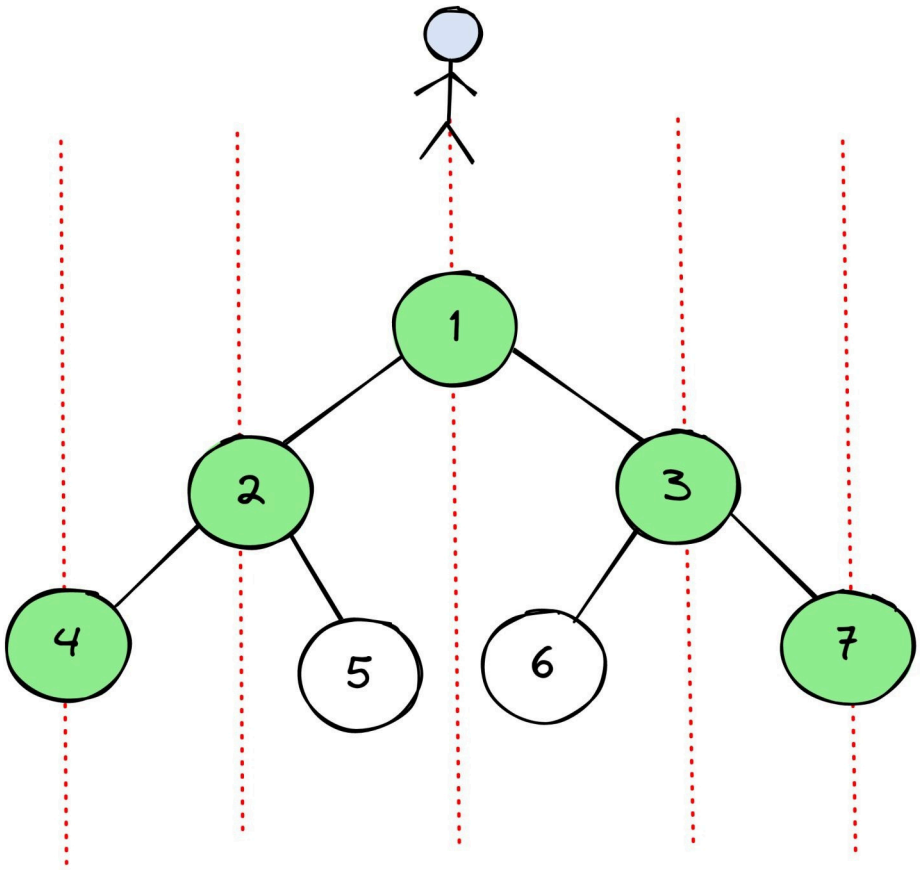
Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

Top view of Binary Tree

Easy

➔ The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree



Top view will be: 4 2 1 3 7

Approach-1 (BFS)

To find the top view of a binary tree using a breadth-first search (BFS) approach, you can follow these steps:

1. Create a queue and add the root node to it.
2. Create a map to store the horizontal distance (HD) of each node as the key and the node's value as the value.
3. Set the HD of the root node as 0.

4. While the queue is not empty, do the following:

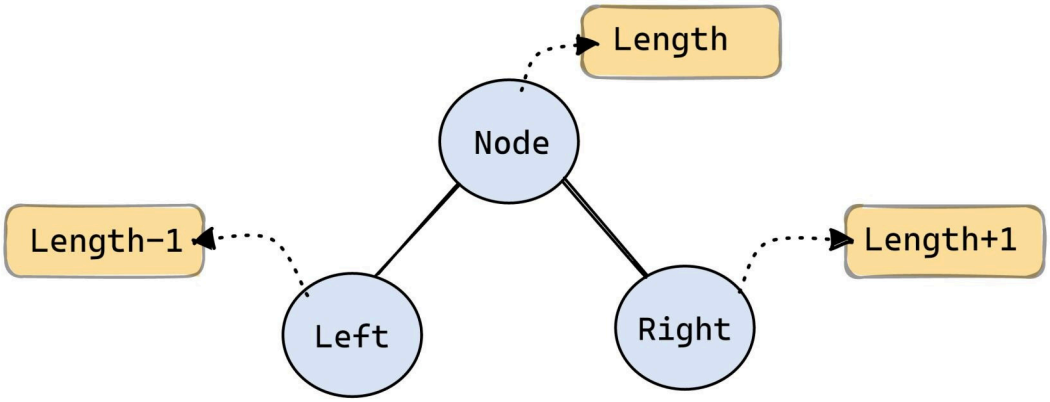
   → Remove the front node from the queue.

   → If the map does not contain the HD of the current node, add the node's value to the map with its HD as the key.

   → If the node has a left child, add it to the queue and set its HD as the HD of the parent node minus 1.

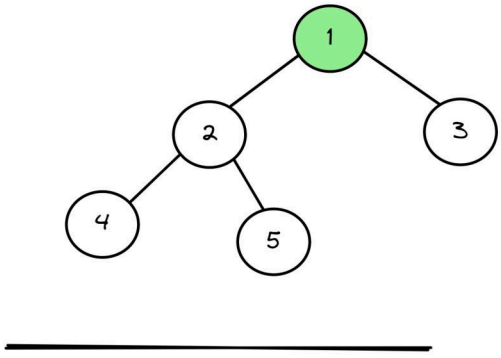
   → If the node has a right child, add it to the queue and set its HD as the HD of the parent node plus 1.

5. Print the values in the map, sorted by the keys in ascending order.

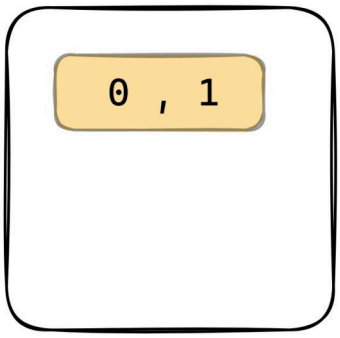


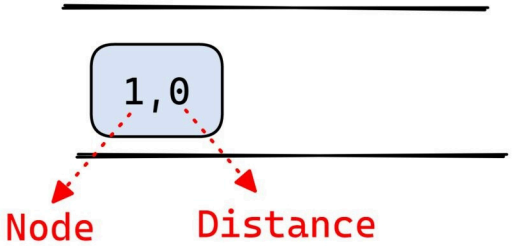
→ initially we are going to add root node with its horizontal distance in Queue.

Let's take one small example for the dry-run.

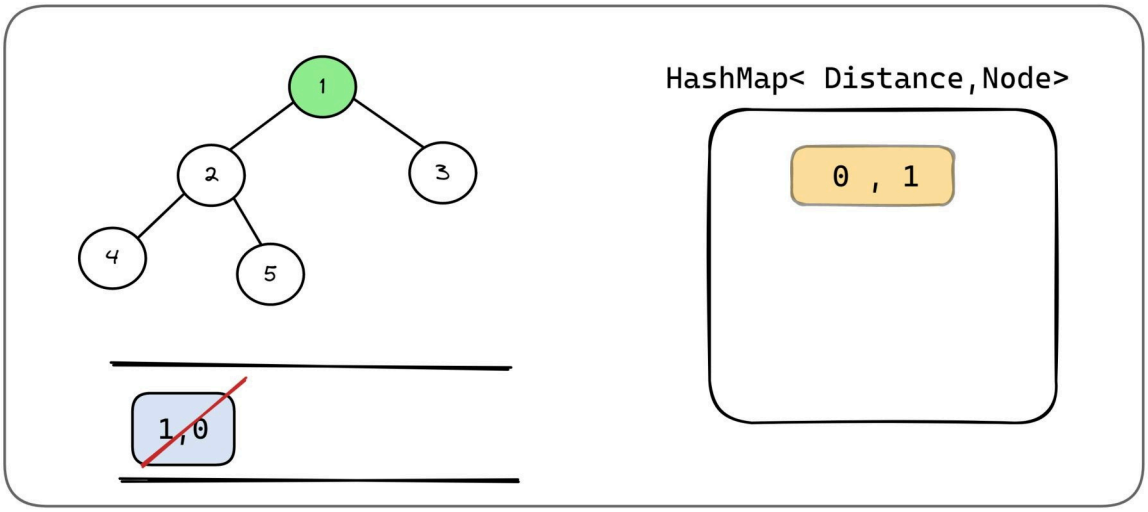


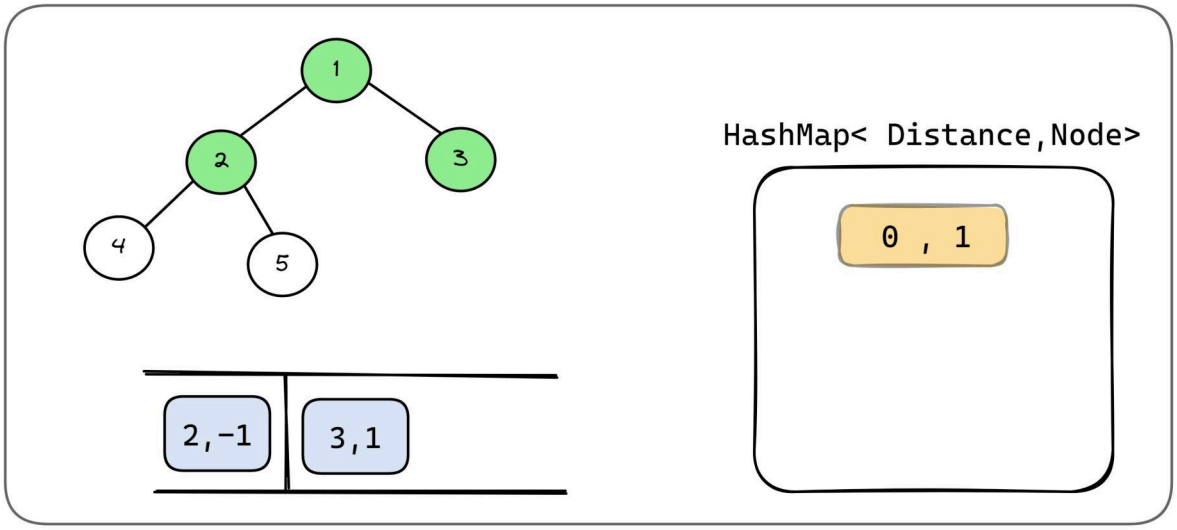
HashMap< Distance,Node>



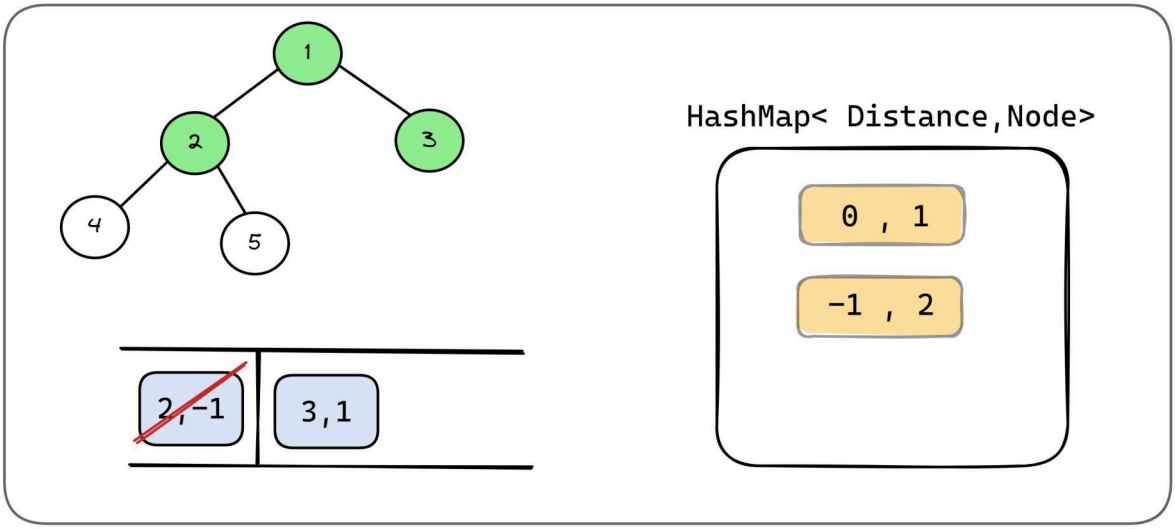


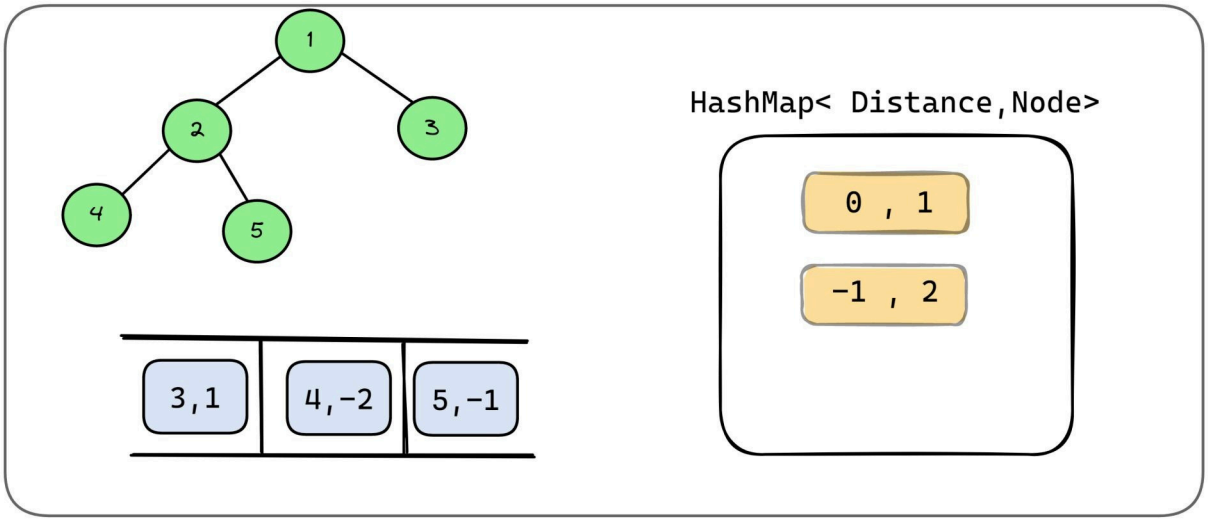
- → Pop node 1 from queue.
  → And check if map contains 0th horizontal distance in hashmap or not. if not than add it in our hashmap.
  → Push left & right children of node 1 in queue.



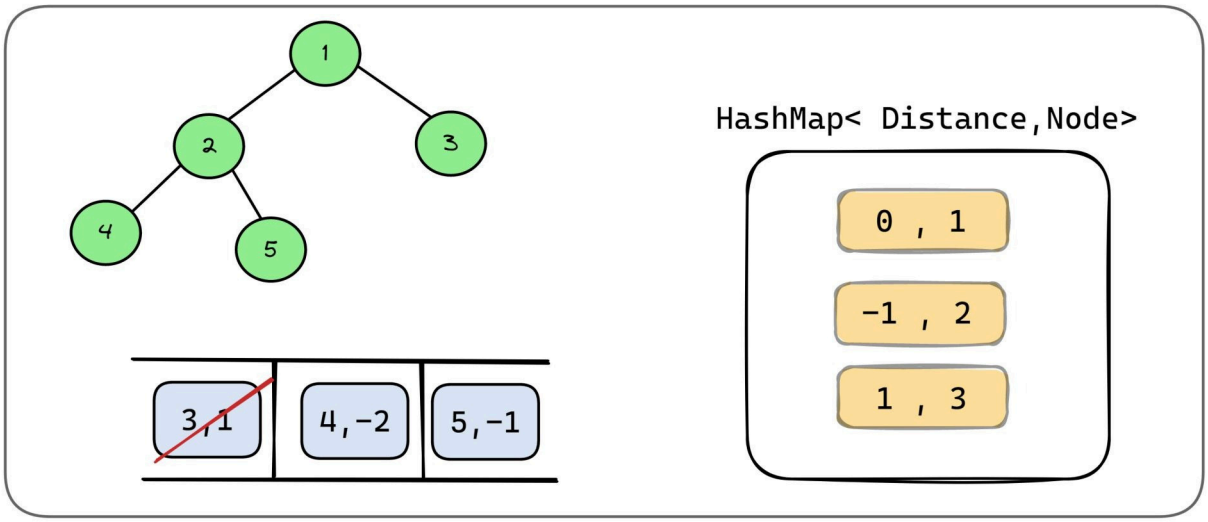


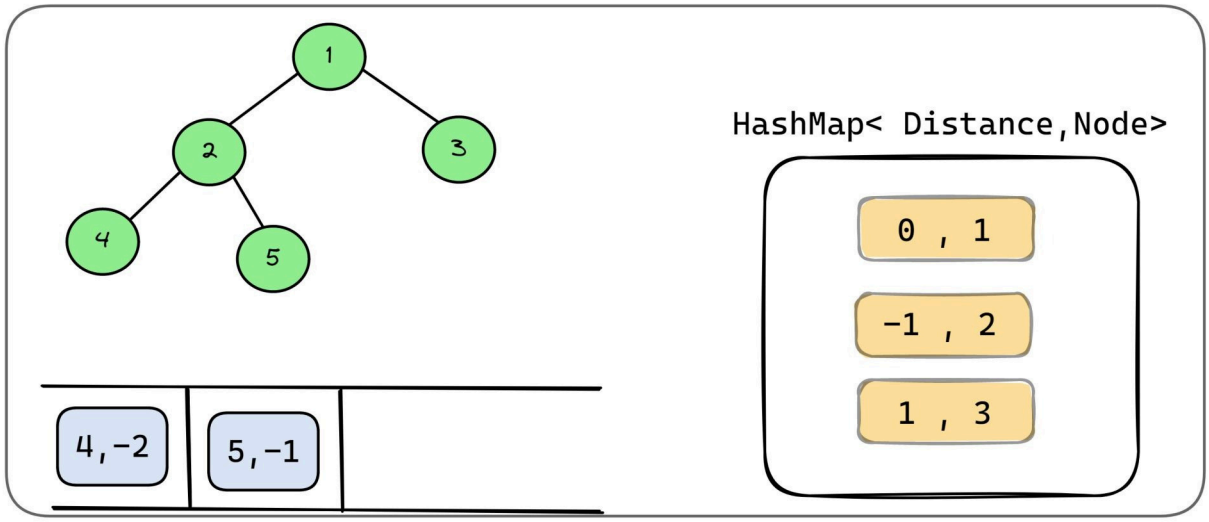
→ Pop node 2 from queue, and add its left & right children.

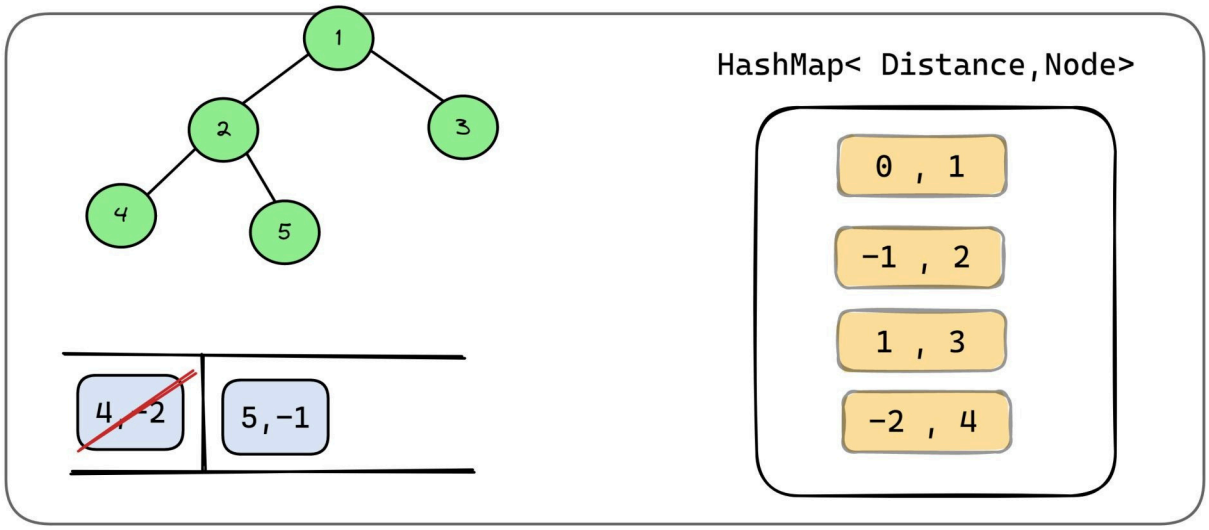


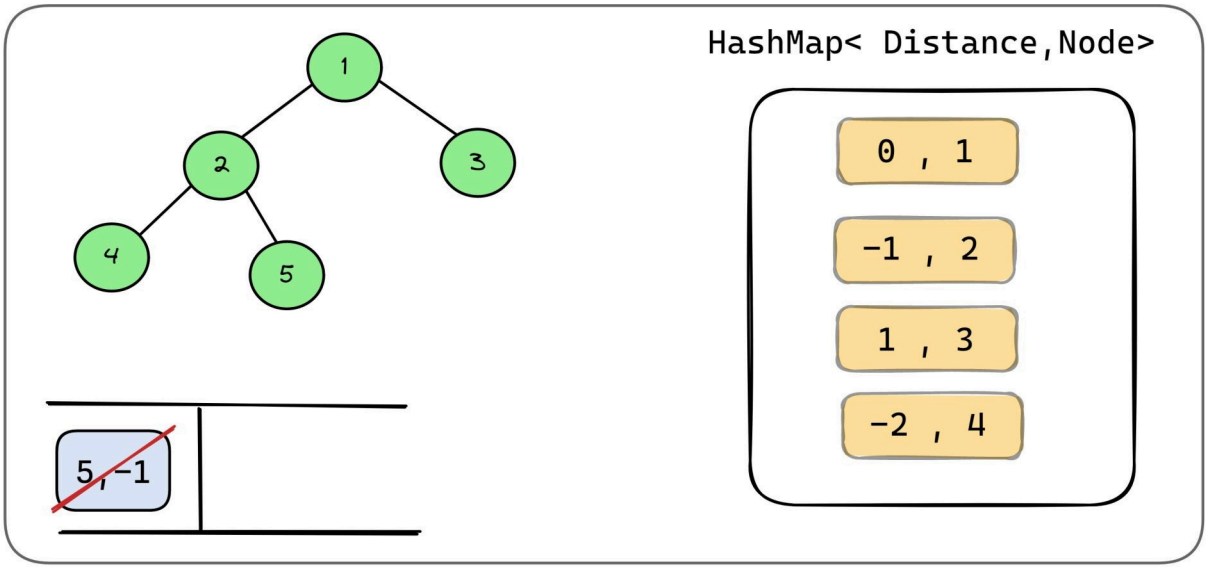


➡ Pop node 3 from queue, and add its left & right children.



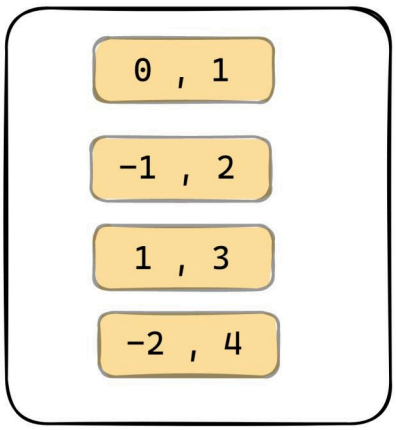






Base-Case → Since, -1 horizontal distance is already present in our hashmap so we are not going to add Node 5.

Now, queue became empty. so the values present in our hashmap is our answer.



HashMap< Distance,Node>

```
class Pair{
    Node node;
    int distance;

    Pair(Node node,int distance){
        this.node = node;
        this.distance = distance;
    }
}

class Solution{
    static ArrayList<Integer> topView(Node root){

        ArrayList<Integer> ans = new ArrayList<>();
        Queue<Pair> queue = new LinkedList<>();
        HashMap<Integer,Integer> memo = new HashMap<>();

        if(root == null) return ans;
        queue.add(new Pair(root,0));

        while(!queue.isEmpty()){

            Pair currentPair = queue.remove();
            Node Node = currentPair.node;
            int distance = currentPair.distance;

            if(!memo.containsKey(distance)){
                memo.put(distance,Node.data);
            }

            if(Node.left != null){
                queue.add(new Pair(Node.left,distance-1));
            }

            if(Node.right != null){
                queue.add(new Pair(Node.right,distance+1));
            }
         }

       for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
           ans.add(entry.getValue());
       }
        return ans;
    }
}
```

→ This solution has a time complexity of O(n) and a space complexity of O(n), where n is the number of nodes in the tree.

Approach-2 (DFS)

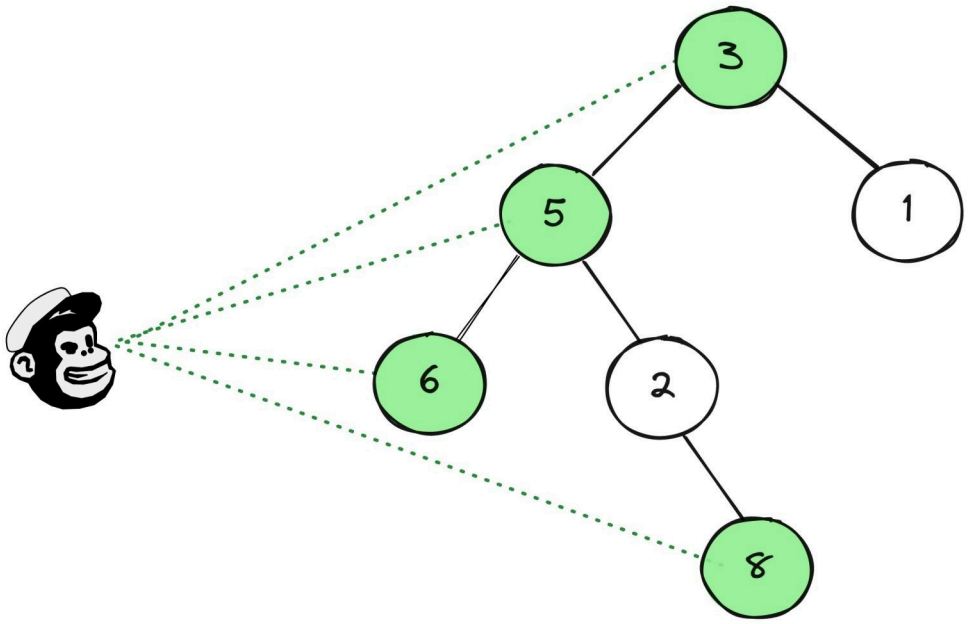
```java
class Solution {
    public void topView(Node root) {

        if (root == null)  return;

        ArrayList<Integer> ans = new ArrayList<>();
        Map<Integer, Integer> hdMap = new TreeMap<>();

        dfs(root, hdMap, 0);

        for (Map.Entry<Integer, Integer> entry : hdMap.entrySet()) {
          ans.add(entry.getValue());
         }
     }

     public void dfs(Node node, Map<Integer, Integer> hdMap, int hd) {
        if (node == null) return;

        if (!hdMap.containsKey(hd)) {
           hdMap.put(hd, node.data);
         }

        dfs(node.left, hdMap, hd - 1);
        dfs(node.right, hdMap, hd + 1);
 }
}
```

→ This solution has a time complexity of O(n) and a space complexity of O(n), where n is the number of nodes in the tree.

Left View of Binary Tree

Medium

- Given a Binary Tree, the task is to print the left view of the Binary Tree. The left view of a Binary Tree is a set of leftmost nodes for every level.



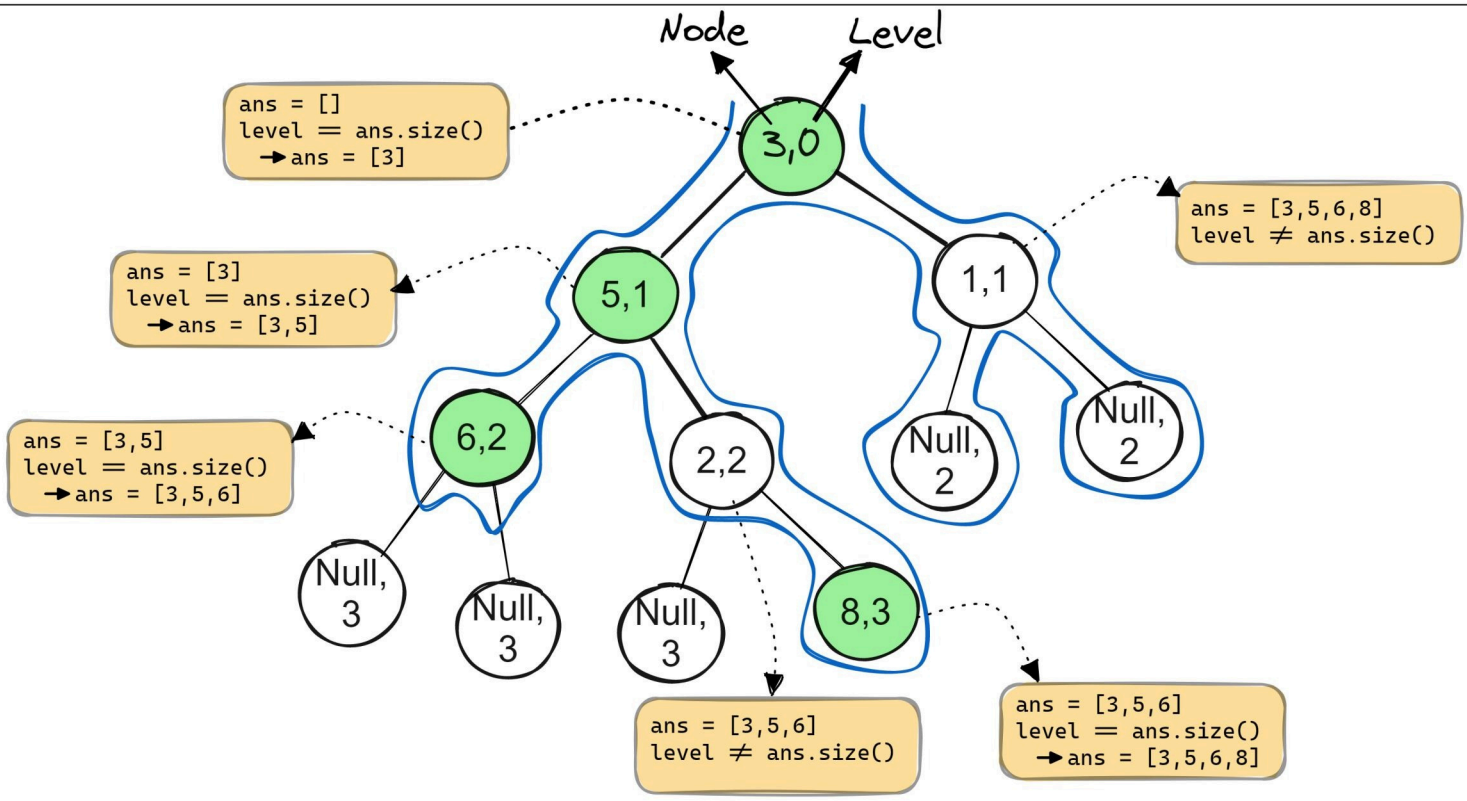
Output: [3,5,6,8]

- Keep track of the level of a node by passing the level as a parameter to all recursive calls and also keep track of the maximum level. Whenever, we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level
- On every level that person is able to see only one node.
- for every node check if its level is equal to size of our ans ArrayList.
  if ans.size() == level then, ans.add(node).

Note: We traverse the left subtree before right subtree. because we want left view of binary tree.

we can also use HashSet, so when ever we encounter new level we are going to that level into our hashset and going to add that node in ans arraylist.

in Santosh Kumar Mishra




```java
class Tree
{
    public ArrayList<Integer> leftView(Node root){

      ArrayList<Integer> ans = new ArrayList<>();
      viewSide(root,0,ans);
      return ans;
    }

    public void viewSide(Node root,int level,ArrayList<Integer> ans){

        if(root == null) return;

        if(level == ans.size()){
            ans.add(root.data);
        }

        viewSide(root.left,level+1,ans);
        viewSide(root.right,level+1,ans);
    }
}
```

Time-Complexity → O(N)
Space-Complexity → O(H)

→ O(H), due to the stack space during recursive call.

Code(Using HashSet):-

```
class Tree
{
    public ArrayList<Integer> leftView(Node root){

     ArrayList<Integer> list = new ArrayList<>();
     leftSide(root,1,list,new HashSet<>());
     return list;
    }

    public void leftSide(Node root,int level,ArrayList<Integer> list,
      HashSet<Integer> set){

        if(root == null) return;

        if(!set.contains(level)){
            set.add(level);
            list.add(root.data);
        }

        leftSide(root.left,level+1,list,set);
        leftSide(root.right,level+1,list,set);
    }
}
```

The Art Of Data Structures and Algorithms

Visual Aids and Detailed Dry Runs for Easy Understanding

The Art of Structures Algorithms

ate Quick Reference Guide

₹399 ~~₹999~~

sh Kumar Mishra

gineer at Microsoft, Author

@iamsantoshmishra

+91-9701101993

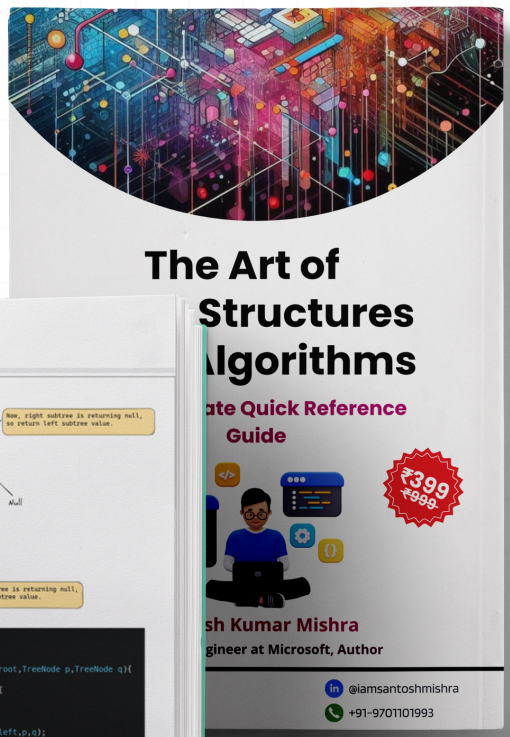
Questions Explained from Brute Force to Optimized Solutions

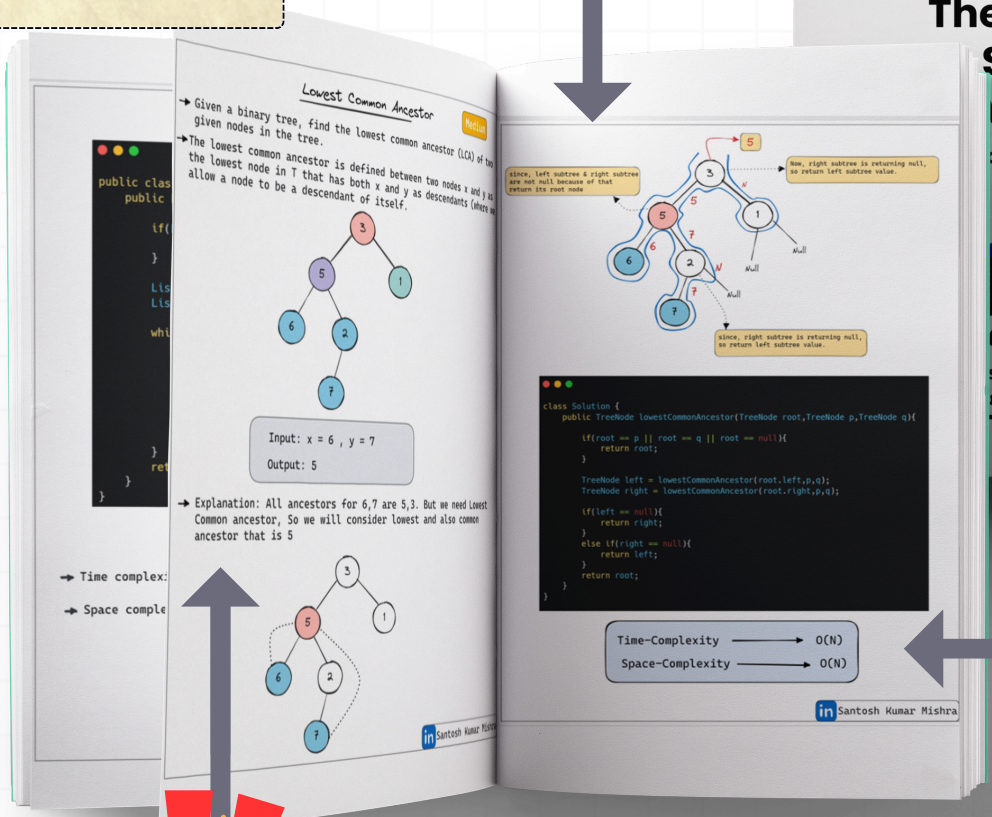
Perfect for Beginners to Advanced Learners

BUY NOW

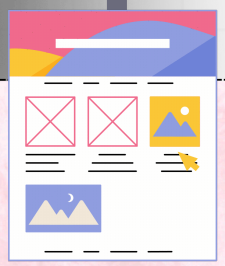
The Art Of Data Structures and Algorithms

One-Time Purchase with Lifetime Access





Comprehensive Coverage of Important DSA Questions



Expertly Designed by Industry Professionals

The Art of Data Structures and Algorithms

Readers' Reviews and Insights

Prem Kumar

Software Engineer

I've never seen a book that covers DSA questions so thoroughly! Santosh breaks down each problem from the basic brute-force solution to the most optimized version, using proper diagrams and dry runs. It's a fantastic resource if you struggle with understanding complex algorithms.

Saumya Awasthi

Software Engineer

If you're preparing for interviews, this is the book you need. It contains all the key DSA problems explained with easy-to-follow graphics. The step-by-step approach, from brute force to optimized solutions, along with well-done dry runs, makes difficult concepts very digestible.

Archy Gupta

Software Engineer

Thank you, Santosh, for sending me a copy of this book. It's truly amazing! The book contains all the important DSA questions, and each one is explained from brute-force to optimized solutions with very clear dry runs and diagrams. It's the perfect resource for mastering DSA concepts and acing coding interviews.

The Art of Data Structures and Algorithms
Readers' Reviews and Insights

Aishwarya Tripathi
Software Engineer

"This book was a game-changer for my Amazon interview preparation. The collection of important DSA problems and their step-by-step solutions from brute-force to optimized helped me build a solid understanding. The graphical dry runs made complex concepts easy to follow. I highly recommend this book if you're aiming for top tech companies!

Parth Chaturvedi
Software Engineer

"The structured explanations in this book played a huge role in my Amazon interview success. The dry runs and graphical representations provided a clear understanding of complex problems, and the progression from brute-force to optimal solutions was exactly what I needed. This book is a must-read for serious interview prep.

Namrata Tiwari
Software Engineer

"This book's strength lies in its ability to break down complex problems into simple, understandable steps. The questions are explained starting from brute-force solutions and are then optimized with clear, graphical dry runs. It's an essential guide for interview prep and mastering DSA.

And Many More Success Stories from Satisfied Readers!

The Art of Data Structures and Algorithms

The Ultimate Quick Reference Guide

BUY NOW

Buy From Gumroad **Buy From Topmate**