

# Trees - Part 1

---

- Karun Karthik

## Contents

---

0. Introduction
1. Max depth of Binary tree
2. Max depth of N-ary tree
3. Preorder of binary tree
4. Preorder of N-ary tree
5. Postorder of binary tree
6. Postorder of N-ary tree
7. Inorder of Binary tree
8. Merge two binary trees
9. Sum of root to leaf paths
10. Uni-valued Binary tree
11. Leaf similar trees
12. Binary tree paths
13. Sum of Left leaves
14. Path sum
15. Left view of Binary tree
16. Right view of Binary tree
17. Same tree
18. Invert Binary tree
19. Symmetric tree
20. Cousins of Binary tree

# Trees

why trees?

Tree - collection of tree-nodes

① class Treenode

```

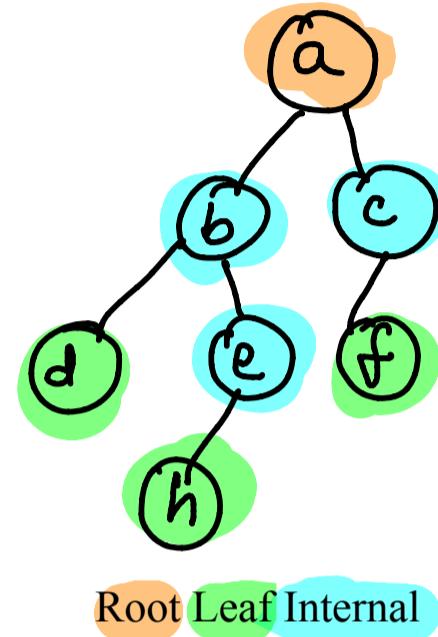
    ↴ data
    ↴ list<Treenode> children
  
```

② Binary Tree → almost 2 children (0,1,2)

```

    ↴ data
    ↴ leftchild
    ↴ rightchild
  
```

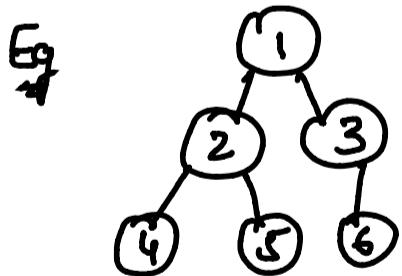
1. Hierarchy
2. Computer system (UNIX)



③ Types →

A) Complete Binary Tree

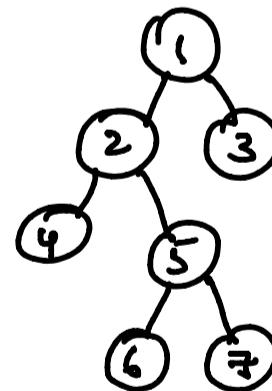
↳ all levels are completely filled except last one



C) Full Binary tree

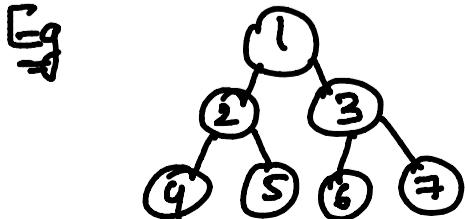
↳ if every node has 0 or 2 children

Eg



B) Perfect Binary Tree

↳ every internal node has exactly 2 children

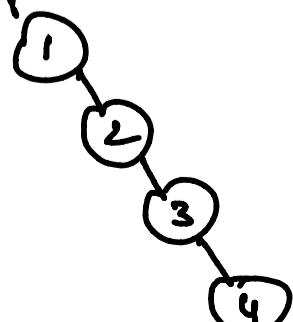


D) Skewed Binary Tree

(\* used for finding complexity)

↳ all nodes have either one or no child.

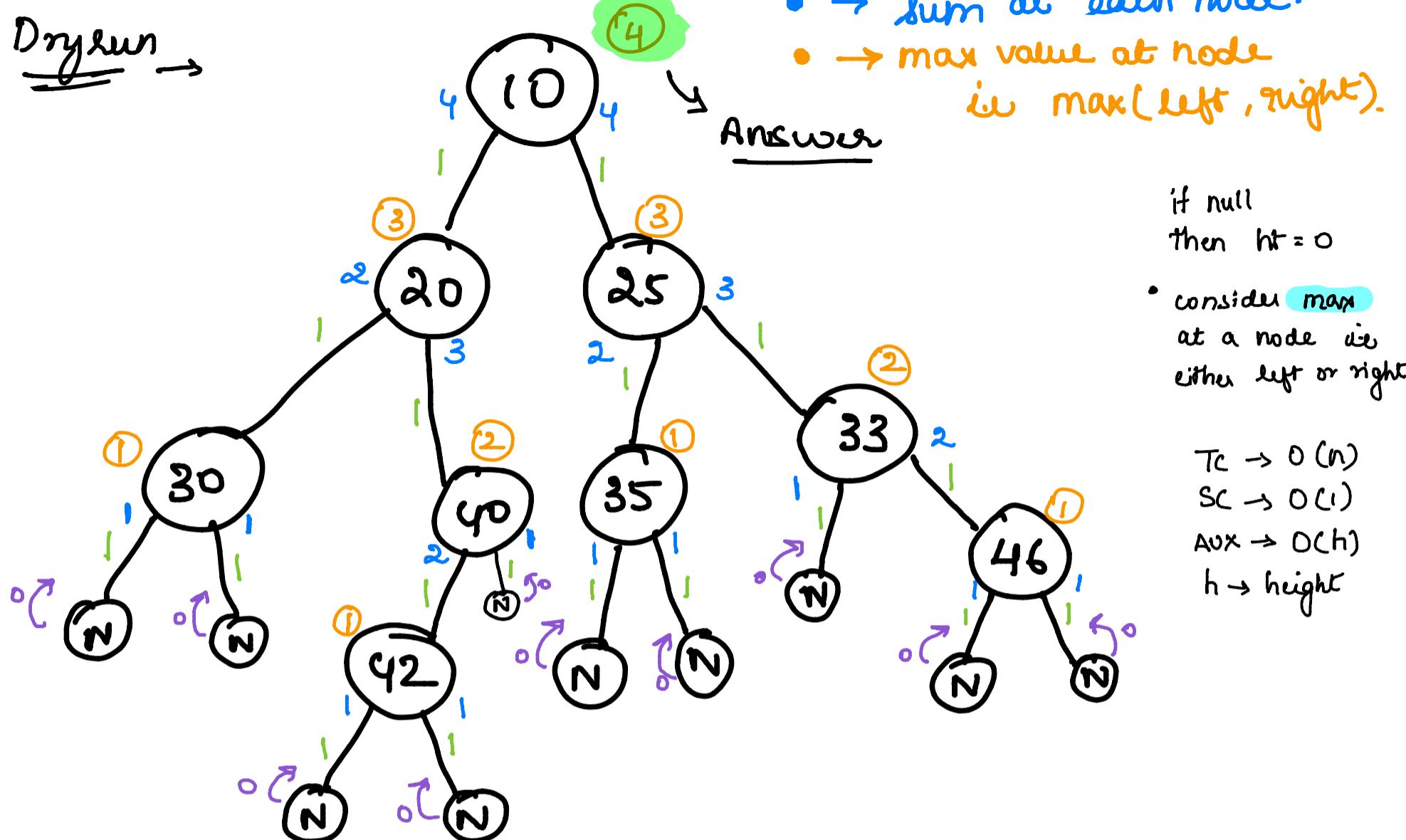
Eg



D1

# ① Depth of a binary tree (Max depth)

Dry run →



Code →

```
C++ v
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;

        int lefth= 1+ maxDepth(root->left);
        int righth = 1+maxDepth(root->right);
        return max(lefth,righth);
    }
};
```

2

## Maximum depth of n-ary tree

Idea is same as previous problem, only implementation changes

Code →

```
C++ ▾

/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

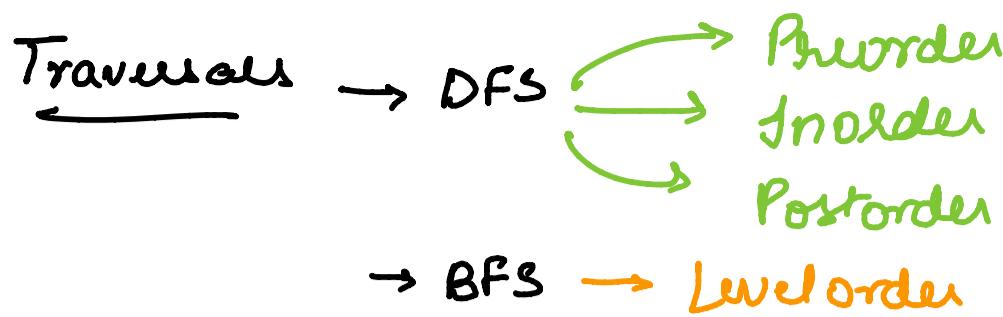
    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
public:
    int maxDepth(Node* root) {
        if(root==NULL) return 0;
        int ans=0;
        for(int i=0;i<root->children.size();i++)
        {
            int tempans = maxDepth(root->children[i]);
            ans = max(ans,tempans);
        }
        return ans+1;
    }
};
```

D2



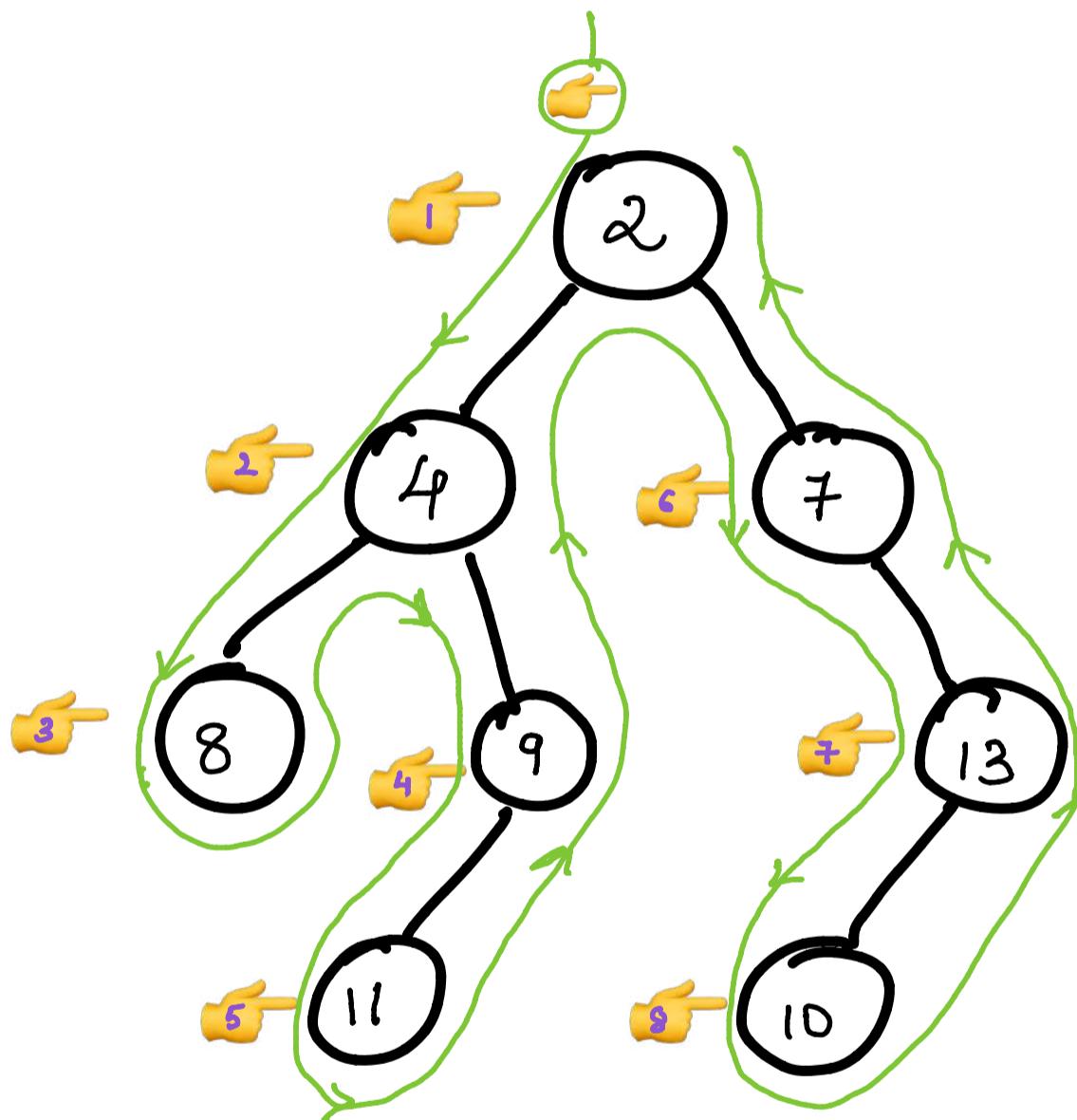
Q4

Preorder →

processing order

node  
left child  
right child

Eg



\* Point fingers as shown  
and traverse the  
tree starting from Root

\* Order of visiting is the  
preorder traversal.

Tc → O(n)

Sc → O(n)

~~[2, 4, 8, 9, 11, 6, 13, 10]~~

Recursive Stack space → O(h) h → height.

### ③ Pre-order traversal of Binary tree

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;
        ans.push_back(root->val);
        Preorder(root->left,ans);
        Preorder(root->right,ans);
        return;
    }
};
```

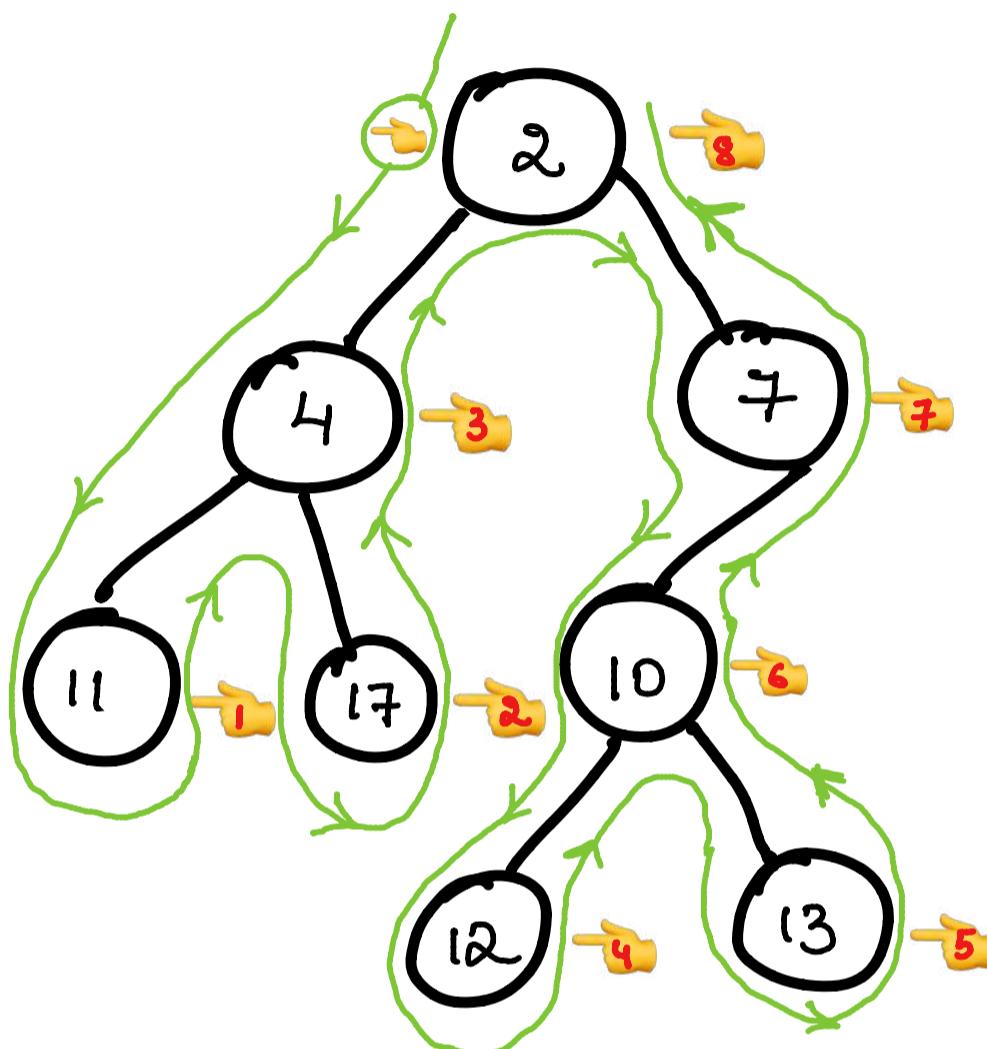
### ④ Pre-order traversal of n-ary tree

```
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(Node* root, vector<int>&ans)
    {
        if(root==NULL) return;
        ans.push_back(root->val);
        for(int i=0;i<root->children.size();i++)
        {
            Preorder(root->children[i],ans);
        }
        return;
    }
};
```

(B) Postorder →  
processing order

left child  
right child  
node

Eg



\* Point finger as shown  
and traverse the  
tree starting from Root

\* Order of visiting is the  
postorder traversal.

Tc → O(n)

Sc → O(n)

~~[11, 17, 4, 12, 13, 10, 7, 2]~~

Recursive Stack space → O(h) h → height .

## ⑤ Postorder traversal of Binary tree

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;

        Postorder(root->left,ans);
        Postorder(root->right,ans);
        ans.push_back(root->val);
        return;
    }
};
```

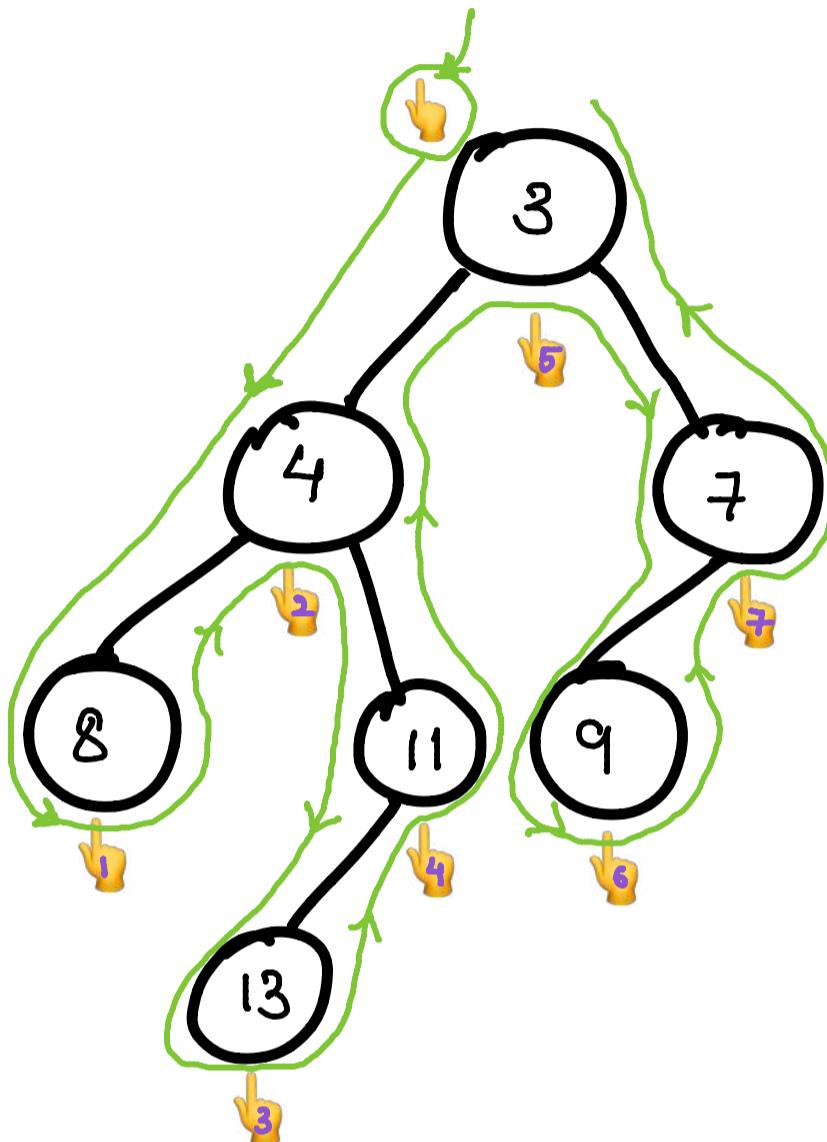
## ⑥ Postorder traversal of nary tree

```
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(Node* root, vector<int>&ans)
    {
        if(root == NULL) return;
        for(int i=0;i<root->children.size();i++)
        {
            Postorder(root->children[i],ans);
        }
        ans.push_back(root->val);
        return;
    }
};
```

(c) Inorder →

processing order →  
 left child  
 node  
 right child

Eg



\* Point fingers as shown  
 and traverse the  
 tree starting from Root

\* Order of visiting is the  
 Inorder traversal.

↙ [8, 4, 13, 11, 3, 9, 7 ]

Tc → O(n)

Sc → O(n)

Recursive Stack space → O(h) h → height .

7

## In-order traversal of Binary tree

```

class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> ans;
        Inorder(root, ans);
        return ans;
    }
    void Inorder(TreeNode* root, vector<int>& ans)
    {
        if (root == NULL) return;
        Inorder(root->left, ans);
        ans.push_back(root->val);
        Inorder(root->right, ans);
        return;
    }
};

```

## In-order traversal of n-ary tree

### Approach:

The inorder traversal of an N-ary tree is defined as visiting all the children except the last then the root and finally the last child recursively.

- Recursively visit the first child.
- Recursively visit the second child.
- .....
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```

void inorder(Node *node)
{
    if (node == NULL)
        return;

    // Total children count
    int total = node->length;

    // All the children except the last
    for (int i = 0; i < total - 1; i++)
        inorder(node->children[i]);

    // Print the current node's data
    cout << node->data << " ";

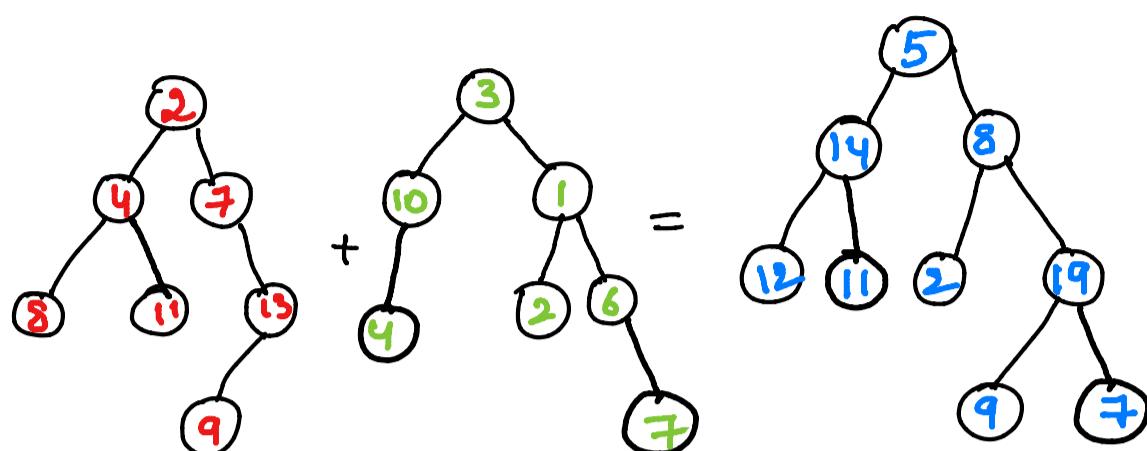
    // Last child
    inorder(node->children[total - 1]);
}

```

### D3 ⑧ Merge two Binary trees →

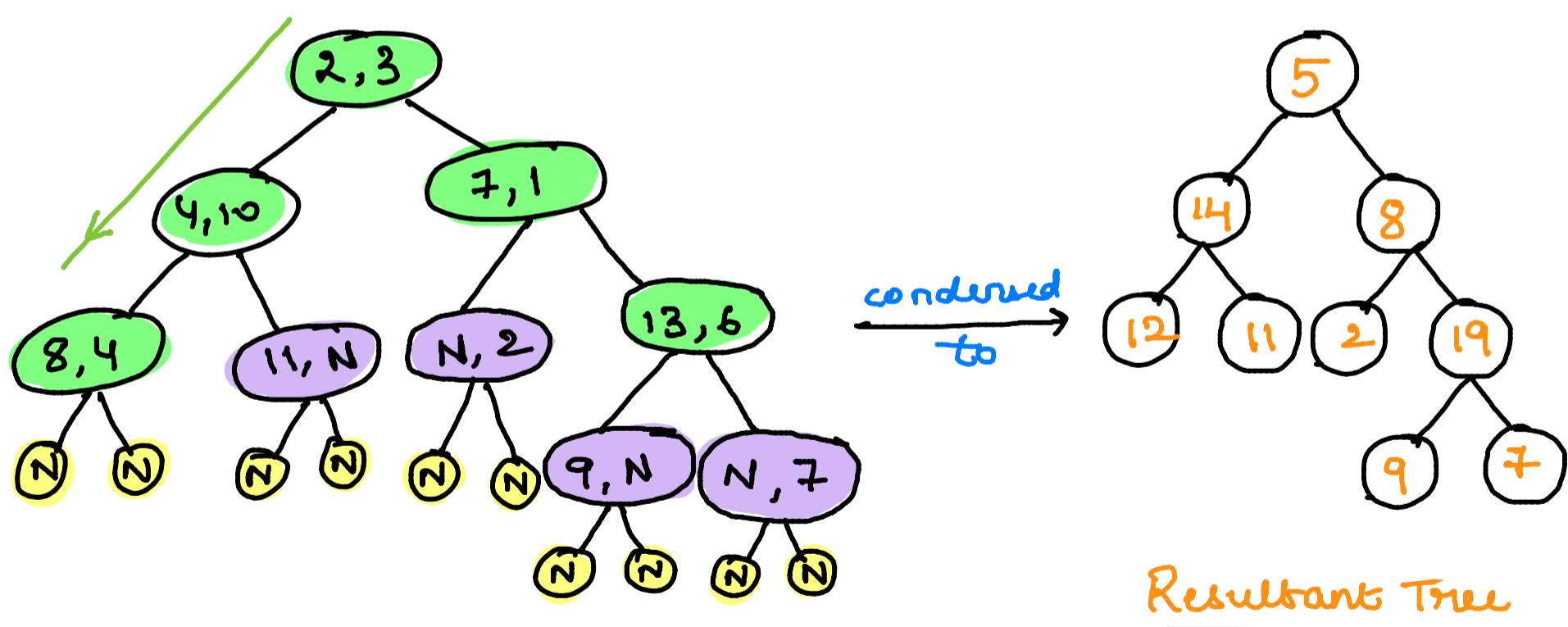
Given root nodes of 2 binary trees, return root of the sum tree

Eg



we will perform preorder traversal on the binary tree because the node/root needs to be processed first.

The recursive tree structure would be like :



Resultant Tree

- NULL & NULL
- Node & NULL
- Node & Node

TC → O(n+m)

SC → O(max(n,m))

Recursive stack → O(max(h<sub>1</sub>, h<sub>2</sub>))

Code →

```
class Solution {
public:
    TreeNode* merge(TreeNode* root1, TreeNode* root2){

        if(root1==NULL && root2==NULL) return NULL;
        if(root1==NULL) return root2;
        if(root2==NULL) return root1;

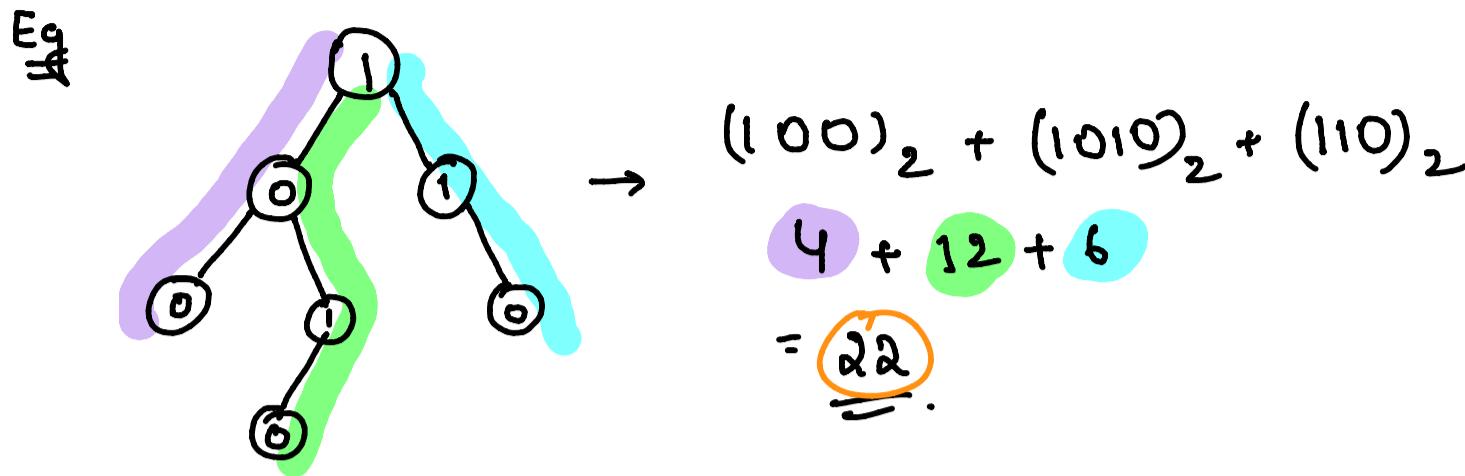
        // Create new node to store sum
        TreeNode *newNode = new TreeNode(root1->val+root2->val);

        // Recursively call the left sub-trees and right sub-trees
        newNode->left = merge(root1->left, root2->left);
        newNode->right = merge(root1->right, root2->right);

        // return the new node
        return newNode;
    }

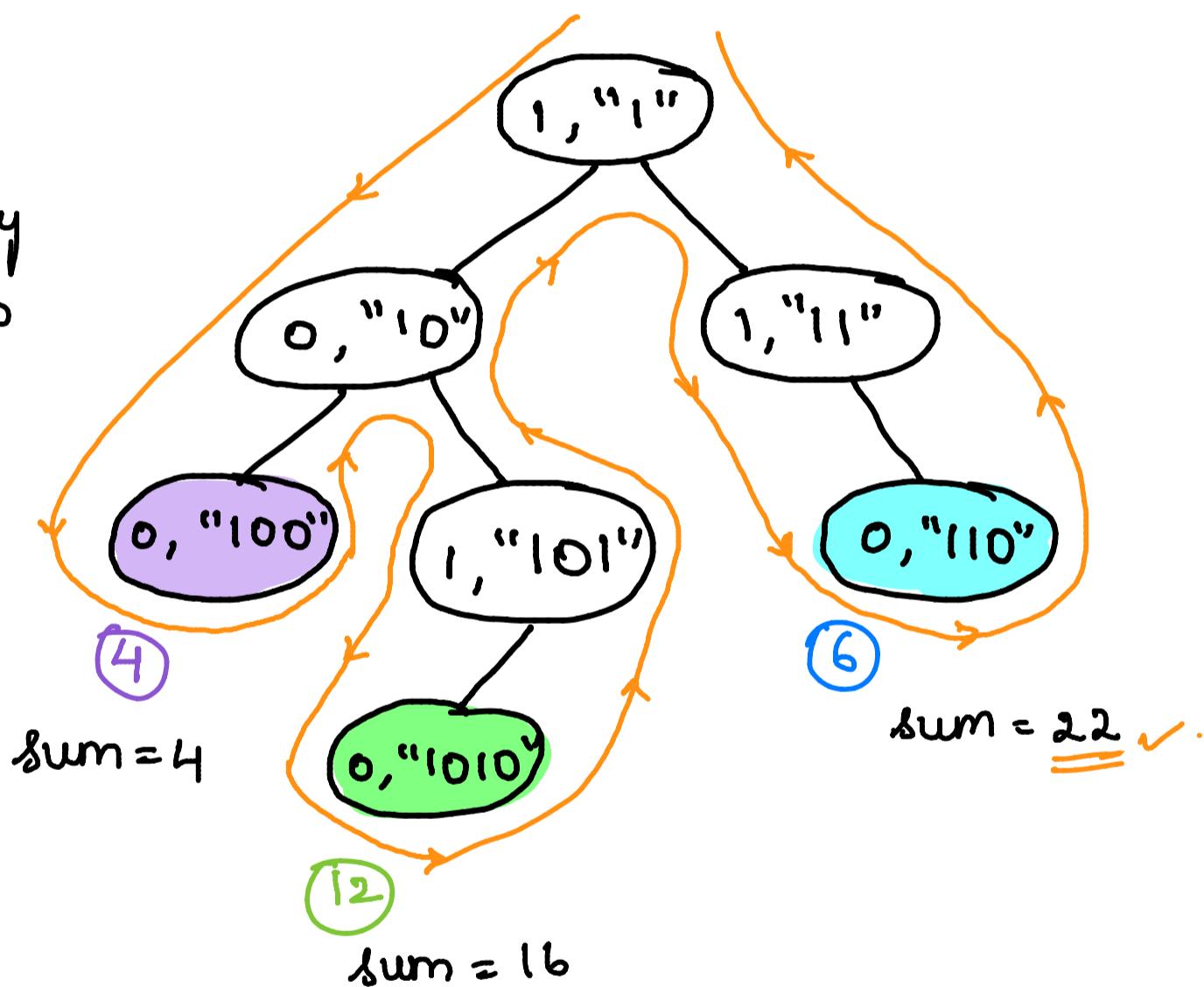
    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        return merge(root1, root2);
    }
};
```

Q) Sum of root to leaf paths →



=

Initially  
 $\text{sum} = 0$



\* If root becomes null convert string to integer & add to sum.

Time →  $O(n)$

Space →  $O(n)$

Recursive stack →  $O(h)$

## Code

```
class Solution {
public:
    void rootToLeaf(TreeNode* root, string currentString,int* ans)
    {
        if(root->left== NULL && root->right==NULL)
        {
            currentString+=to_string(root->val);
            ans[0]+=stoi(currentString,0,2);
            return;
        }
        string curr=to_string(root->val);
        if(root->left!=NULL)
            rootToLeaf(root->left,currentString+curr,ans);
        if(root->right!=NULL)
            rootToLeaf(root->right,currentString+curr,ans);

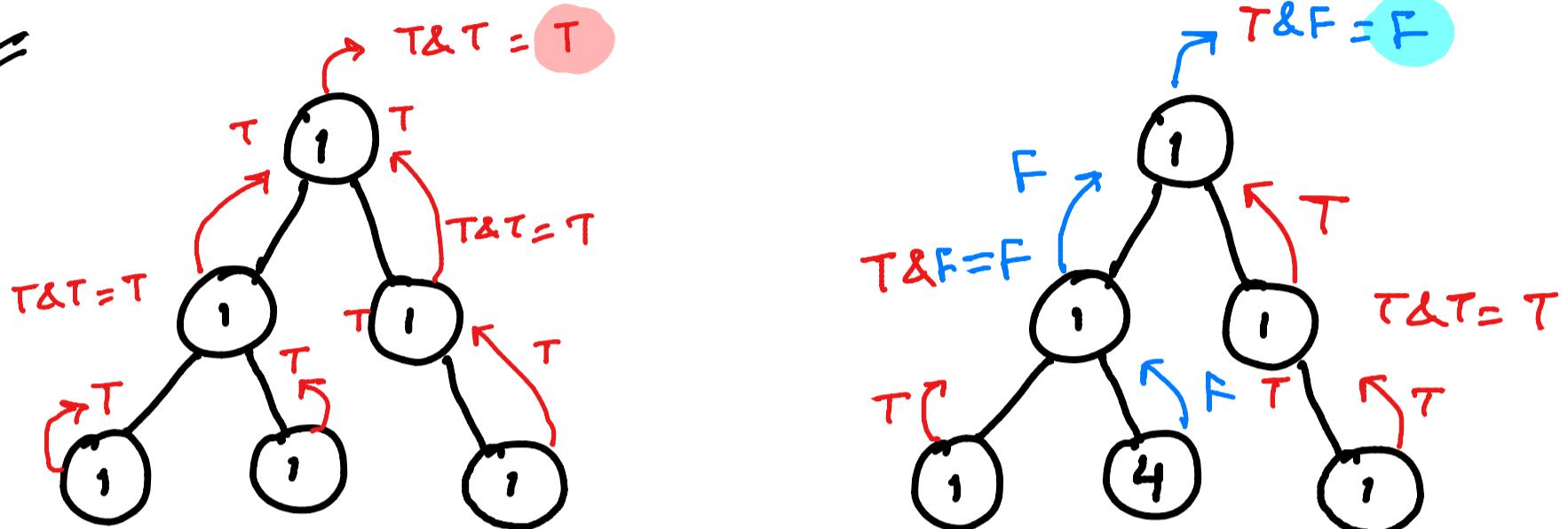
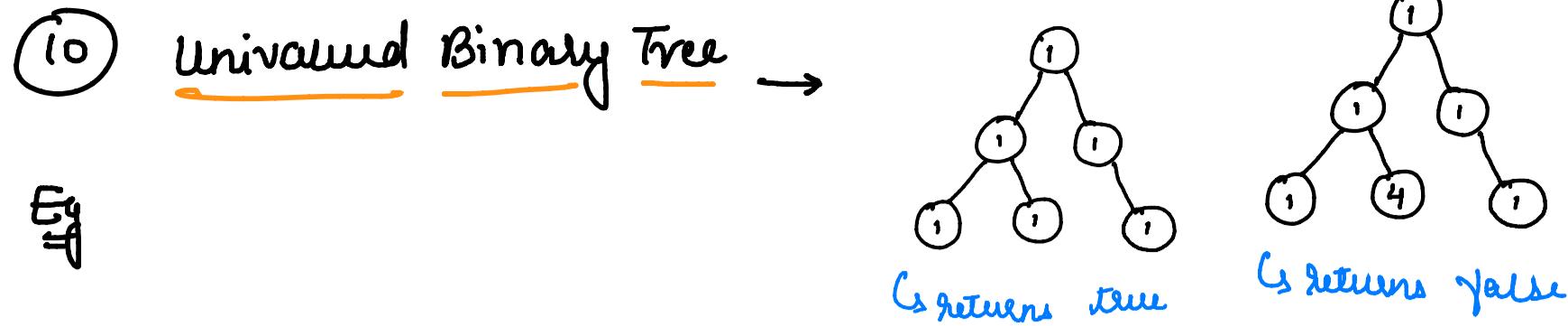
    }
    int sumRootToLeaf(TreeNode* root) {
        int* ans=new int[1];
        ans[0]=0;
        rootToLeaf(root,"",ans);
        return ans[0];
    }
};
```

## Note →

stoi() can take upto three parameters, the second parameter is for starting index and third parameter is for base of input number.



[to convert from binary to decimal we give it as 2]



## Code

```

class Solution {
public:
    bool isSame(TreeNode* root, int val){
        if(root==NULL) return true;
        if(root->val!=val) return false;

        bool left = isSame(root->left, val);
        bool right = isSame(root->right, val);

        return left && right;
    }

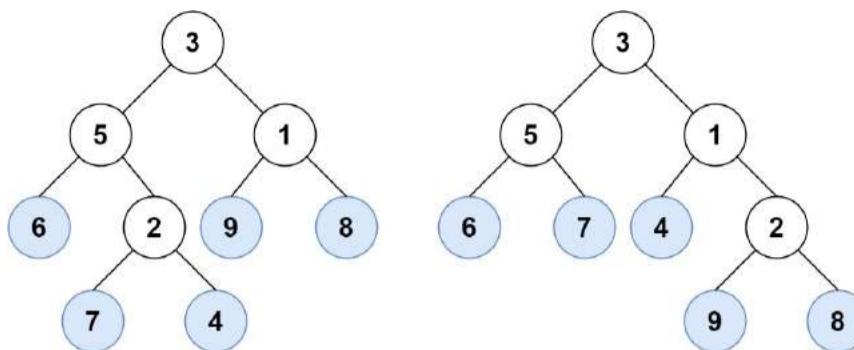
    bool isUnivalTree(TreeNode* root) {
        return isSame(root, root->val);
    }
};

```

## ⑪ Leaf Similar trees

→ return true if all leaves are in same order for both trees.

Eg



$$V_1 = 6, 7, 4, 9, 8 \rightarrow V_1 = V_2$$

$$V_2 = 6, 7, 4, 9, 8 \quad \text{↳ returns true else false.}$$

Code →

```
class Solution {
public:
    void traversal(TreeNode* root, vector<int>&v){
        if(root==NULL)
            return;

        if(root->left==NULL && root->right==NULL)
            v.push_back(root->val);

        if(root->left!=NULL)
            traversal(root->left, v);

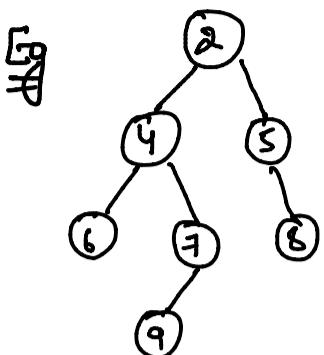
        if(root->right!=NULL)
            traversal(root->right, v);
    }

    bool leafSimilar(TreeNode* root1, TreeNode* root2) {
        vector<int> a;
        vector<int> b;
        traversal(root1,a);
        traversal(root2,b);
        return a==b;
    }
};
```

DS

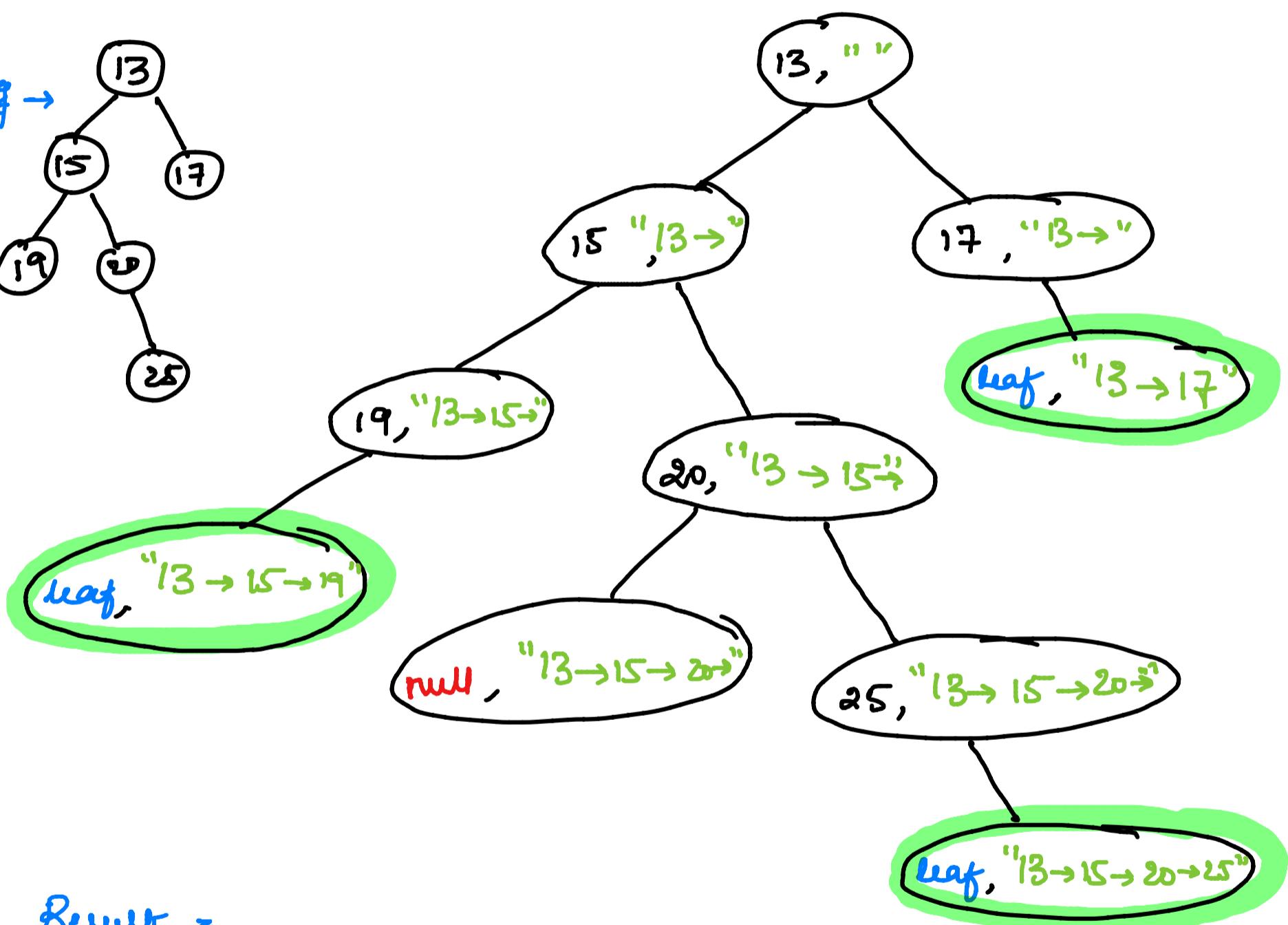
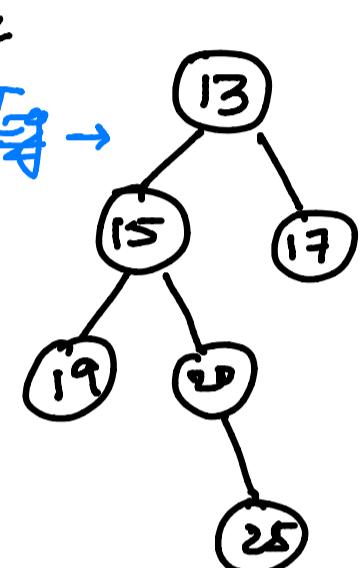
## 12 Binary tree paths

Given root print all the paths from root to leaf



$\Rightarrow [ "2 \rightarrow 4 \rightarrow 6", "2 \rightarrow 4 \rightarrow 7 \rightarrow 9", "2 \rightarrow 5 \rightarrow 8" ]$

=



Result =

$[ "13 \rightarrow 15 \rightarrow 19", "13 \rightarrow 15 \rightarrow 20 \rightarrow 25", "13 \rightarrow 17" ]$

Time complexity =  $O(n)$

Space complexity =  $O(\alpha) + O(h)$   $\rightarrow$  recursive stack.  
 $\downarrow$  Answer array

## Code →

```
class Solution {
public:
    void pathFinder(TreeNode *root, vector<string> &res, string currPath){

        if(root==NULL)  return;

        // if leaf then add it's value to currentPath
        if(root->left == NULL && root->right==NULL){
            currPath += to_string(root->val);
            res.push_back(currPath);
            return;
        }

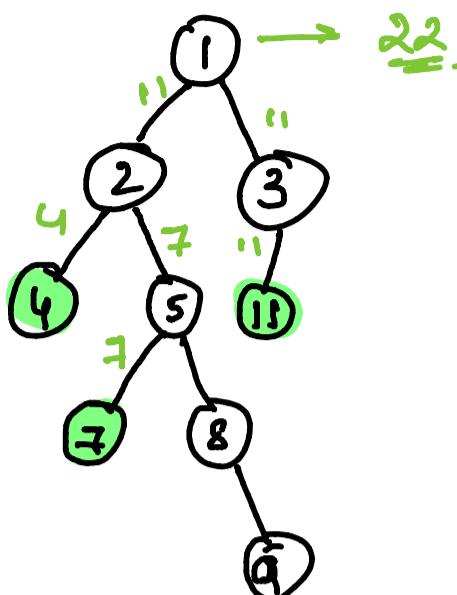
        // else add the node's value to path
        currPath += to_string(root->val)+"->";

        if(root->left)  pathFinder(root->left, res, currPath);
        if(root->right) pathFinder(root->right, res, currPath);
    }

    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res;
        pathFinder(root, res, "");
        return res;
    }
};
```

(13) sum of left leaves →

Eg

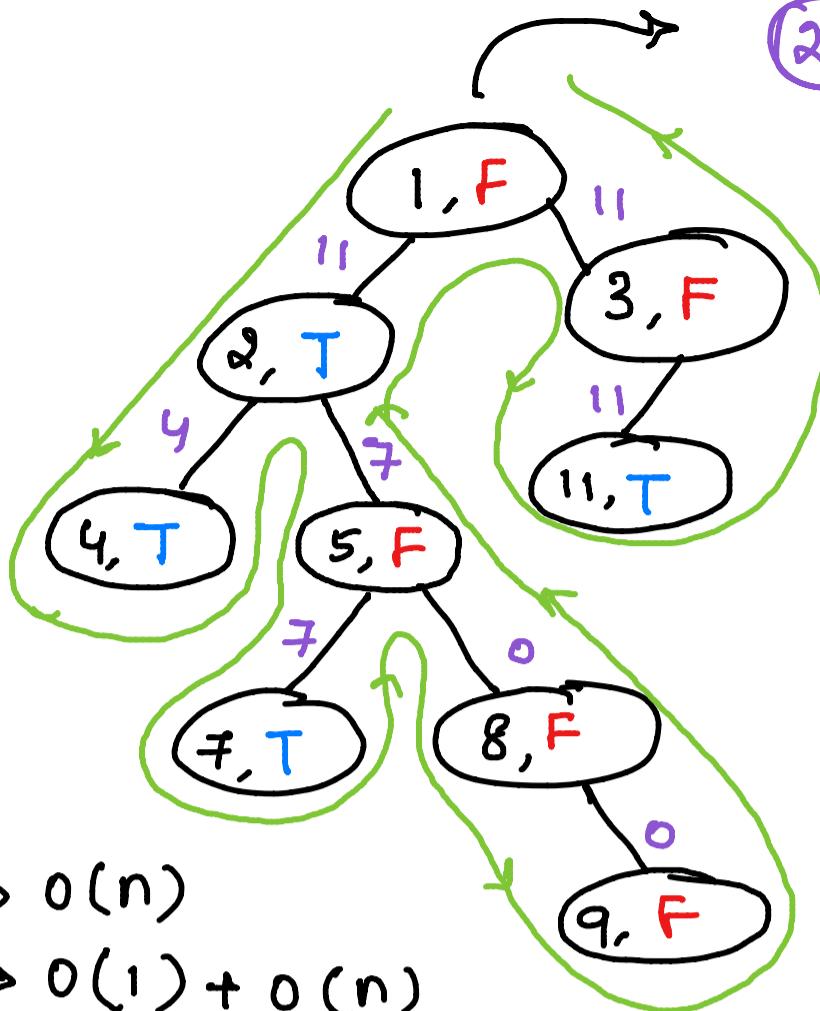


$$\text{Result} = 4 + 7 + 11 \\ = \underline{\underline{22}}.$$

$$Tc \rightarrow O(n) \\ Sc \rightarrow O(1) + O(n)$$

L → Stack

(22)

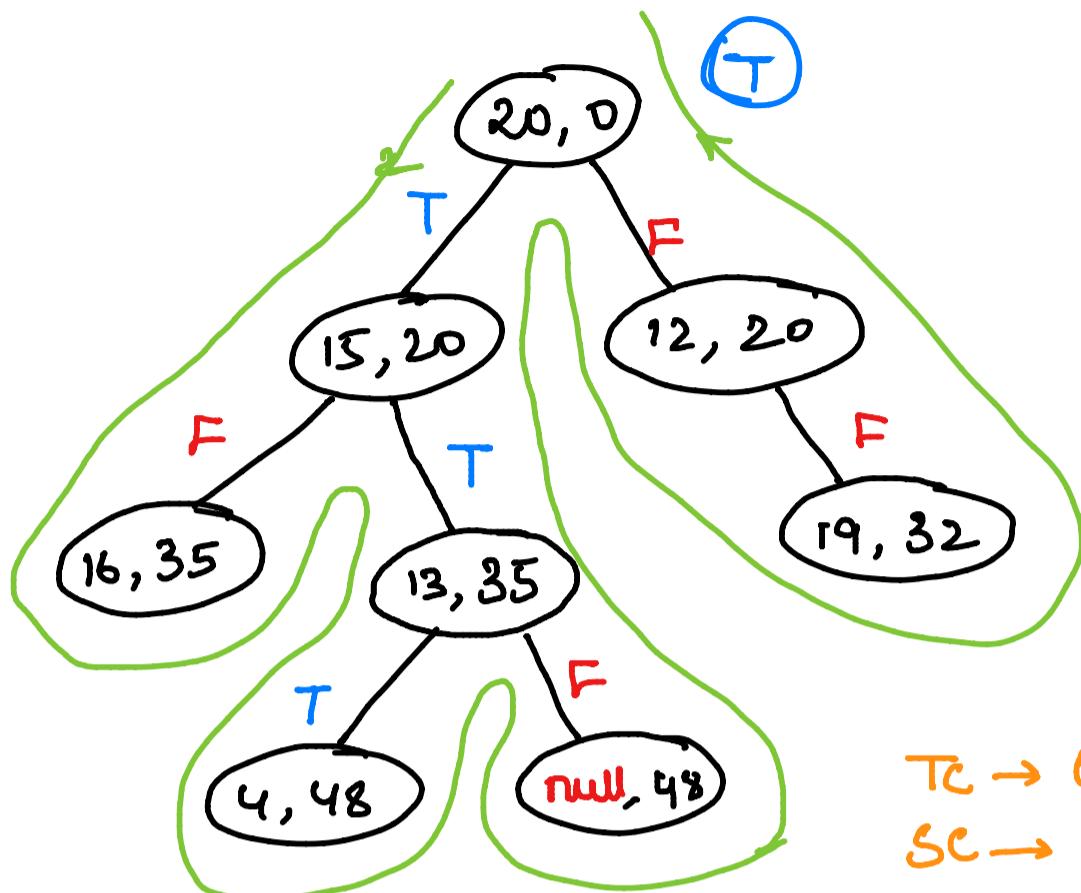
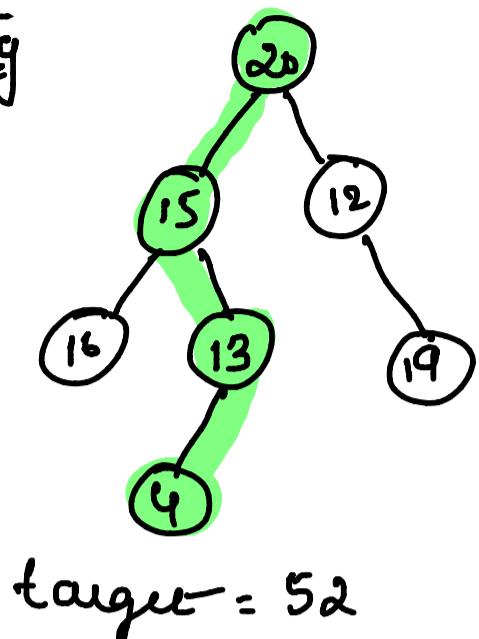


Code →

```
class Solution {  
public:  
    int leftLeafSum(TreeNode *root, bool leaf){  
        if(root==NULL){  
            return 0;  
        }  
        if(root->left==NULL && root->right==NULL && leaf){  
            return root->val;  
        }  
        int ls = leftLeafSum(root->left, true);  
        int rs = leftLeafSum(root->right, false);  
        return ls+rs;  
    }  
  
    int sumOfLeftLeaves(TreeNode* root) {  
        return leftLeafSum(root, false);  
    }  
};
```

14 Path sum → sum of all nodes from root to leaf is equal to target sum → then T else F.

Ex



TC → O(n)

SC → O(1)

Recursive → O(h)  
Stack

Code

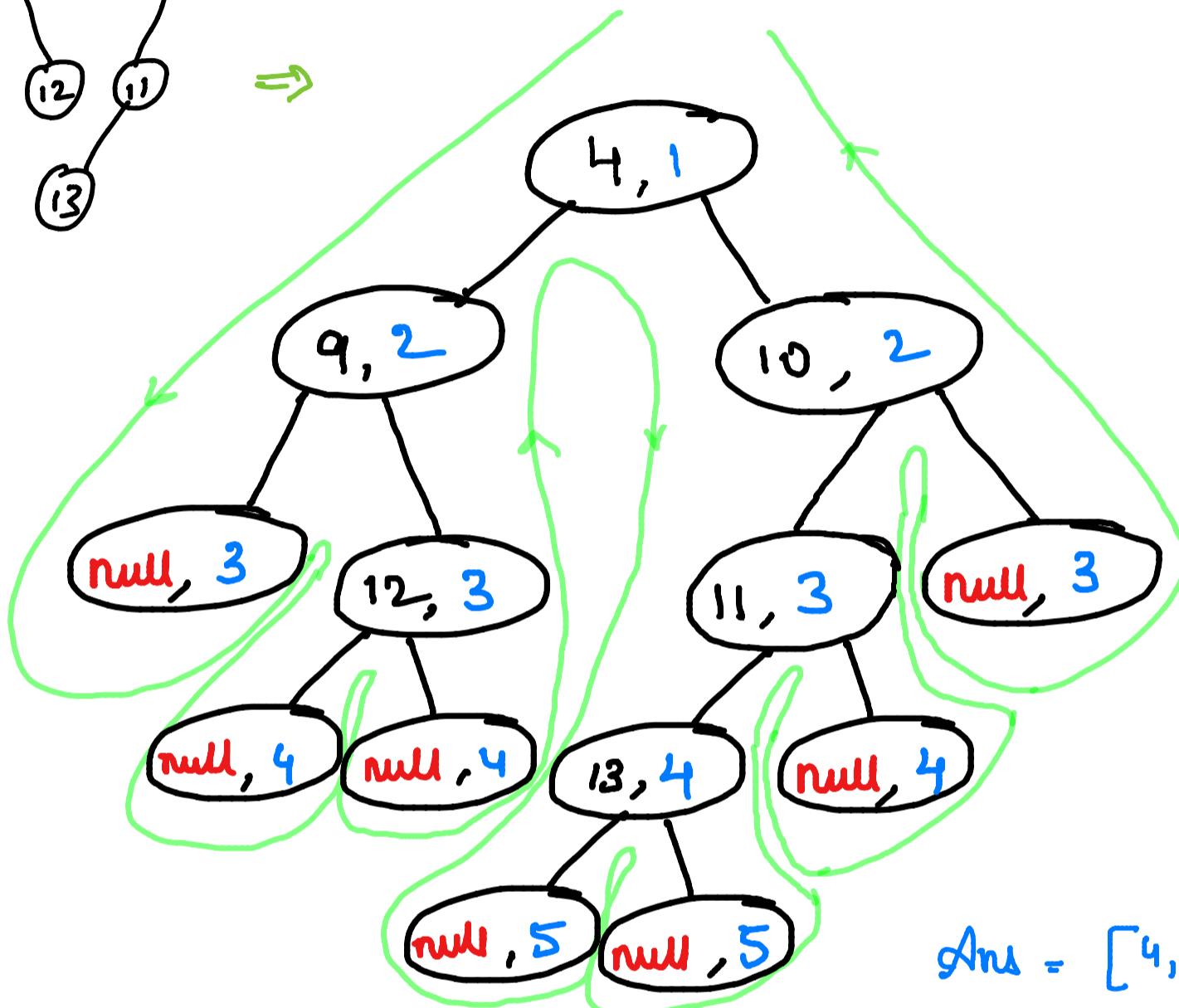
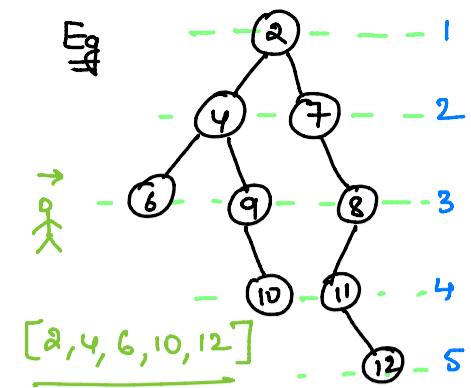
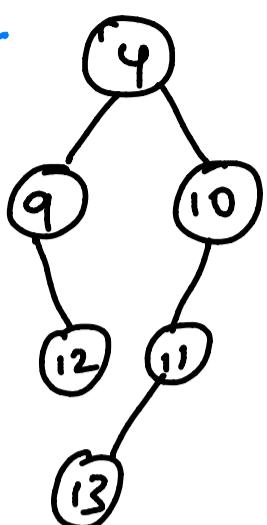
```
class Solution {
public:
    bool pathSumUtil(TreeNode* root, int currSum, int targetSum){
        if(root==NULL)
            return false;

        if(root->left==NULL && root->right==NULL){
            return (currSum+root->val)==targetSum;
        }

        return pathSumUtil(root->left, currSum+root->val, targetSum)
            ||pathSumUtil(root->right, currSum+root->val, targetSum);
    }

    bool hasPathSum(TreeNode* root, int targetSum) {
        return pathSumUtil(root, 0, targetSum);
    }
};
```

DL

(15) Left view of a Binary Tree

→ For every level traversed,  
check if it already exist in the set,

if already exist then continue,  
else add the root's value  
to array q into the set

$T_C \rightarrow O(n)$

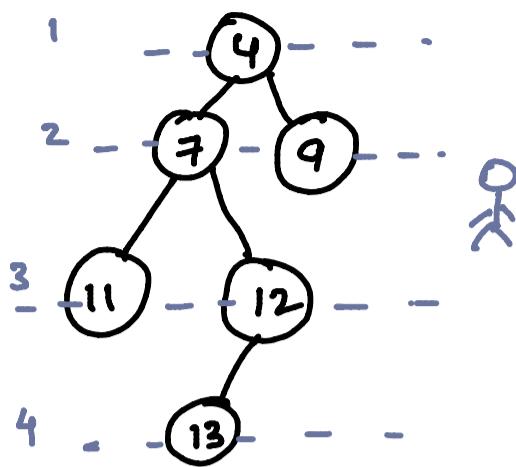
$S_C \rightarrow O(n) + O(n) + O(h)$

$\downarrow$   
result

## Code →

```
void viewGenerator(Node *root, vector<int> &res, set<int> &s, int currLevel){  
    if(root==NULL) return;  
    // if level is not reached, then add to result and the set  
    if(s.find(currLevel)==s.end()){  
        s.insert(currLevel);  
        res.push_back(root->data);  
    }  
    // traverse the remaining branches  
    viewGenerator(root->left, res, s, currLevel+1);  
    viewGenerator(root->right, res, s, currLevel+1);  
    return;  
}  
  
vector<int> leftView(Node *root)  
{  
    vector<int> res;  
    set<int> s;  
    viewGenerator(root, res, s, 0);  
    return res;  
}
```

## 16 Right view of Binary Tree →



Result = [4, 9, 12, 13].

- The entire approach to solve the problem is same as the left view of binary tree. Even the time complexities.
- Only order of calling the branches change.
  - ① right
  - ② left

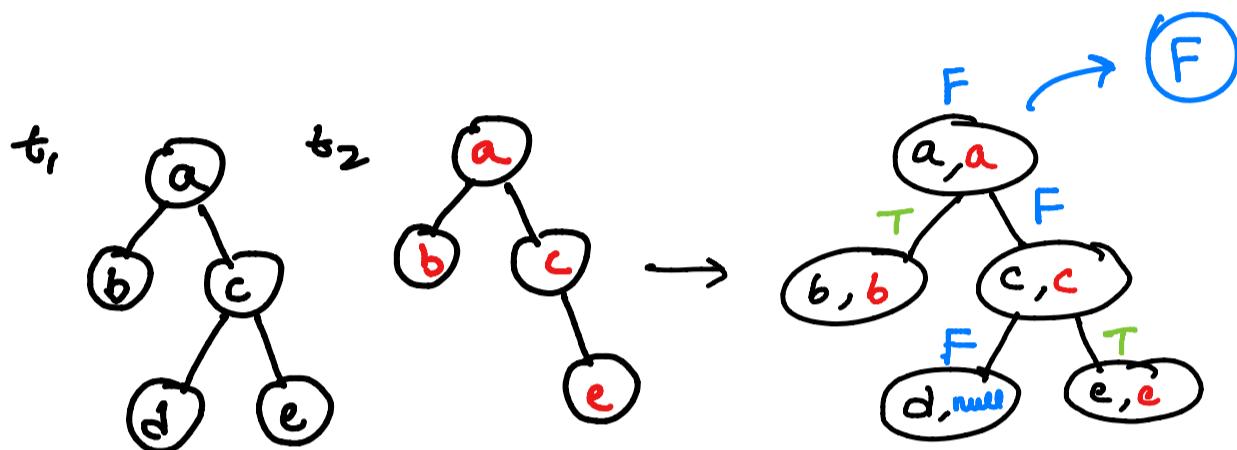
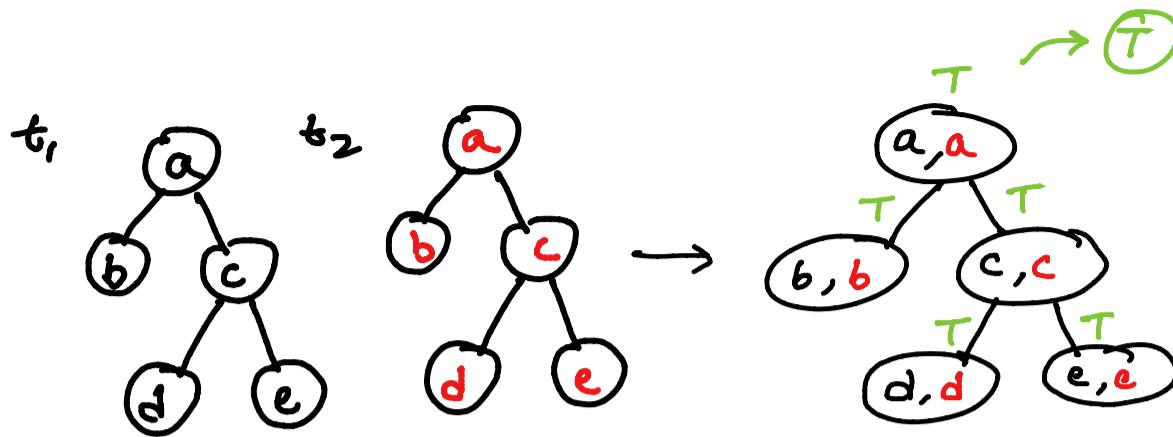
### Code

```

class Solution {
public:
    void viewGenerator (TreeNode* root, vector<int> &res, set<int> &s, int currLevel){
        if(root==NULL) return;
        // if level is not reached, then add to result and the set
        if(s.find(currLevel)==s.end()){
            s.insert(currLevel);
            res.push_back(root->val);
        }
        // traverse the remaining branch
        viewGenerator(root->right, res, s, currLevel+1);
        viewGenerator(root->left, res, s, currLevel+1);
        return;
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        set<int> s;
        viewGenerator(root, res, s, 0);
        return res;
    }
};
  
```

$T_C \rightarrow O(n)$   
 $S_C \rightarrow O(n) + O(n) + O(h)$   
 ↴  
 result

17 Same tree → return true if both trees are same  
else false



$$TC \rightarrow O(\min(m, n))$$

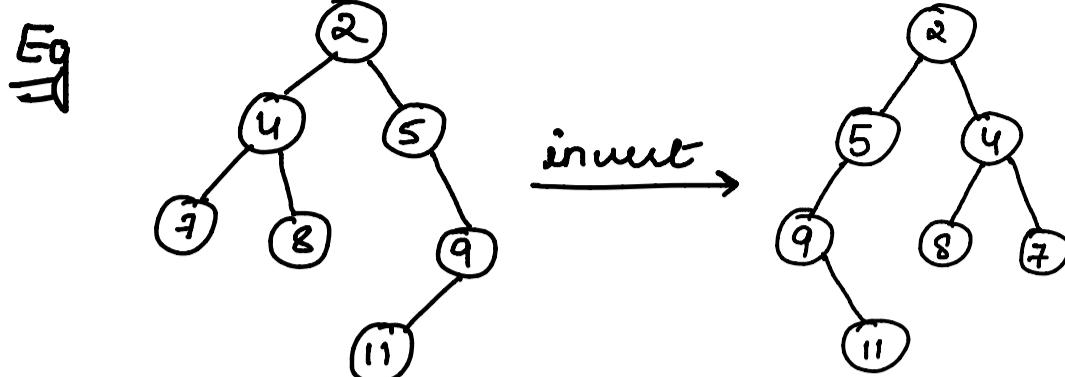
$$SC \rightarrow O(1) + O(\min(h_1, h_2))$$

code →

```
class Solution {
public:

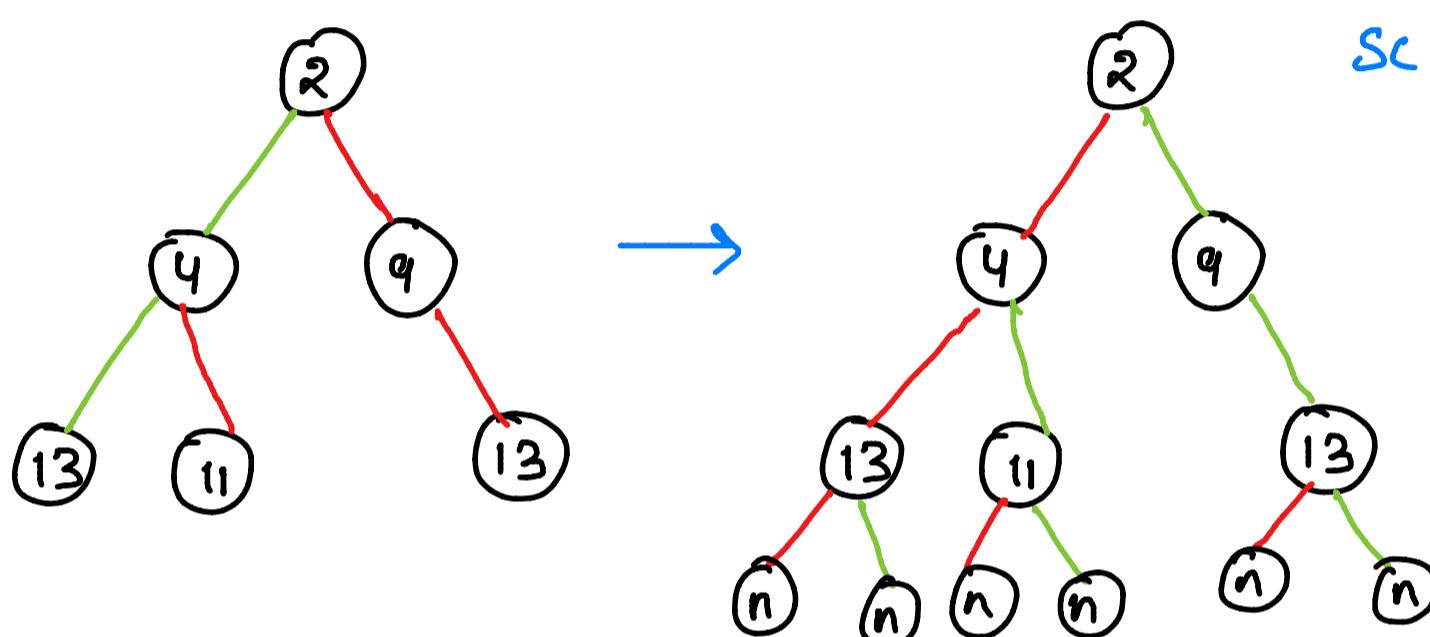
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p==NULL && q==NULL) return true;
        if(p==NULL || q==NULL || p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

(18) Invert Binary Tree → given the root of BT, find its mirror img.



TC → O(n)

SC → O(n)+O(h)



Code →

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(root==NULL) return root;

        /* invert the left and right sub-trees and store
           them separately */
        TreeNode *leftSub = invertTree(root->right);
        TreeNode *rightSub = invertTree(root->left);

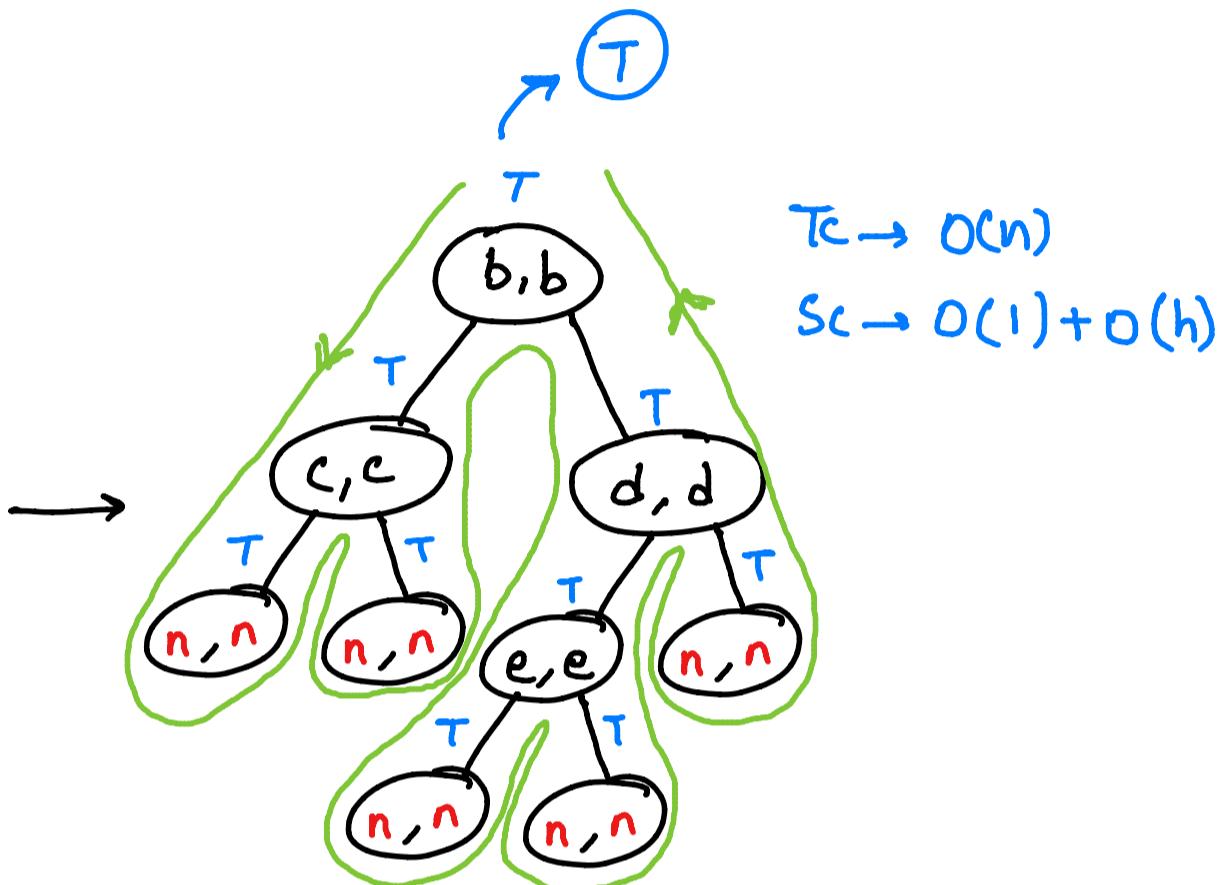
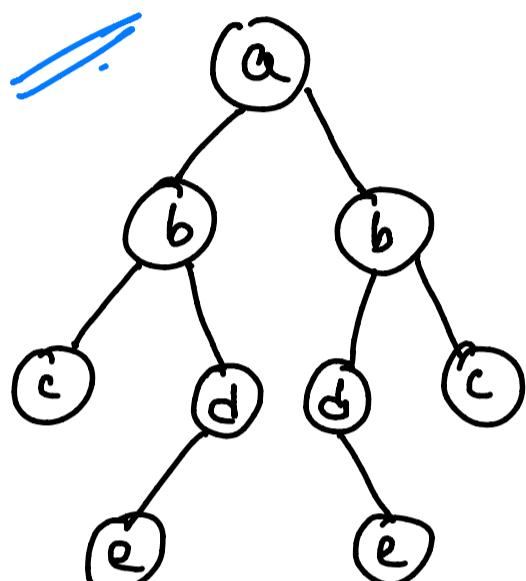
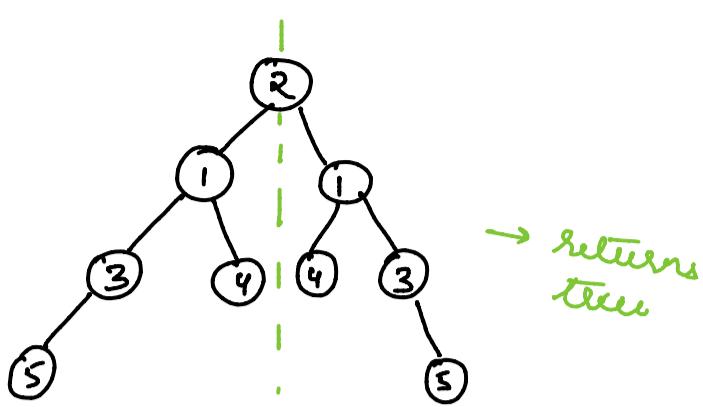
        // attach the branches to root
        root->left = leftSub;
        root->right = rightSub;

        return root;
    }
};
```

D7

19 Symmetric Tree →

return true if left subtree  
is equal to right subtree,  
else return false



Code →

```

class Solution {
public:
    bool isMirror(TreeNode* l, TreeNode* r){

        if(l== NULL && r==NULL)
            return true;
        else if(l==NULL || r==NULL)
            return false;
        else if(l->val != r->val)
            return false;

        return isMirror(l->left,r->right) && isMirror(l->right, r->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(root==NULL) return true;
        return isMirror(root->left, root->right);
    }
};

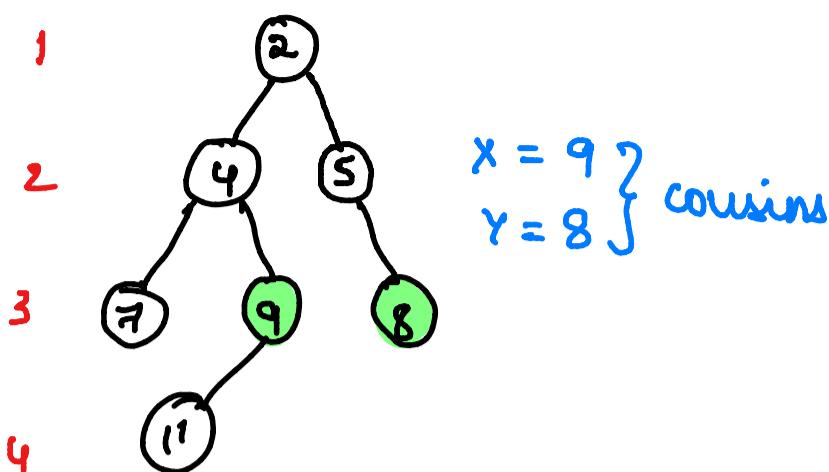
```

20

## Cousins of a Binary Tree

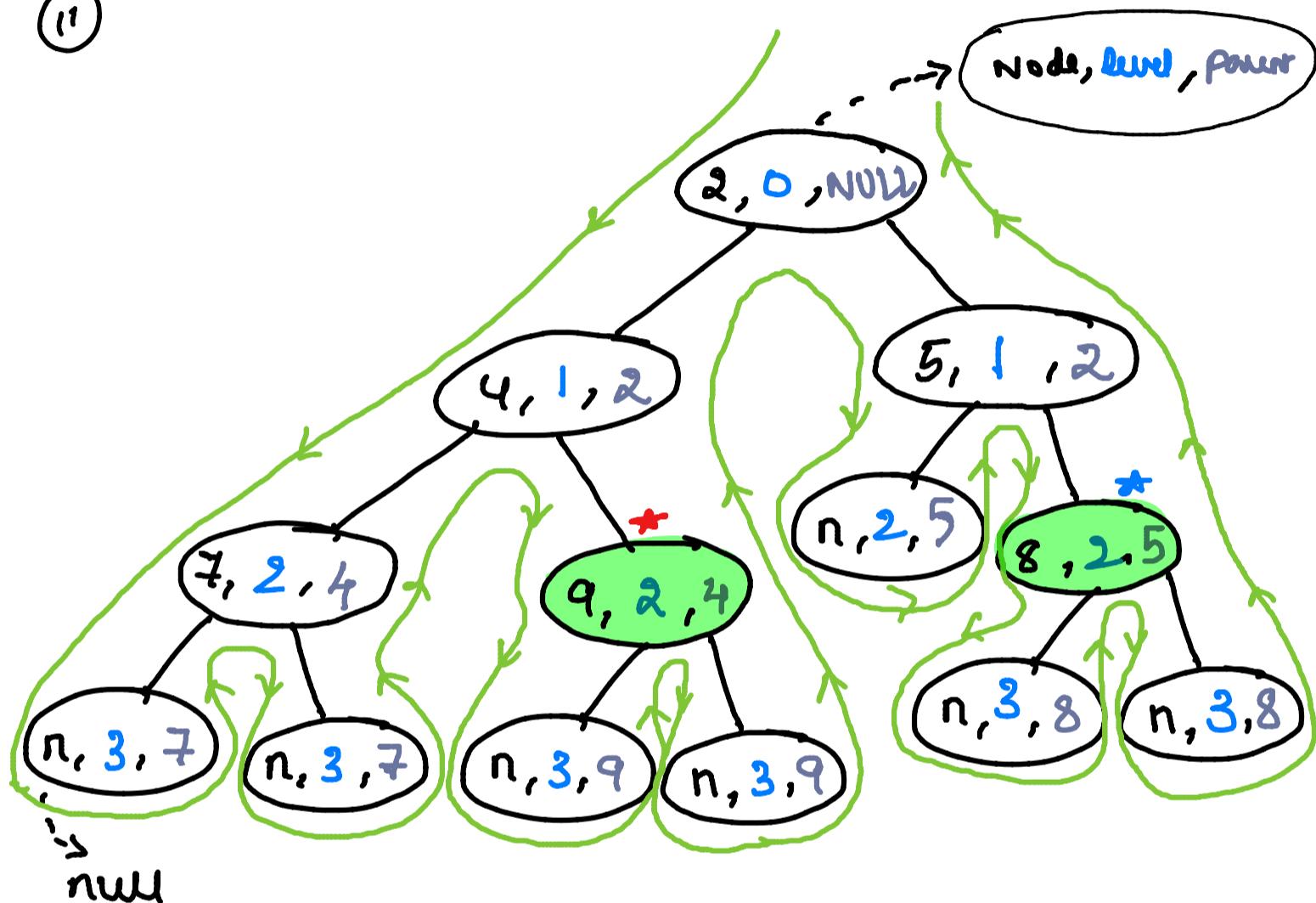
→ given two nodes, find if they are cousins of each other.

Ex:



$$\begin{aligned} x = 9 \\ y = 8 \end{aligned} \} \text{ cousins}$$

same level but diff parents.



- \* at this step as value = 9 is found store it's parent & level in separate variables
- \* later compare its value with other occurrence in Y such that

- 1) x.parent != y.parent
- 2) x.level == y.level.

TC  $\rightarrow O(n)$

SC  $\rightarrow O(1)$

Recursive stack  $\rightarrow O(n)$

## Code

```
class Solution {
public:
    void findNodes(TreeNode* root, int x, int y,int level[2],int parents[2],int currlevel,TreeNode* currparent)
    {
        if(root==NULL) return;
        if(root->val == x)
        {
            level[0]=currlevel;
            parents[0]=currparent->val;
        }
        if(root->val == y)
        {
            level[1]=currlevel;
            parents[1]=currparent->val;
        }
        findNodes(root->left, x, y, level, parents, currlevel+1, root);
        findNodes(root->right, x, y, level, parents, currlevel+1, root);
    }
    bool isCousins(TreeNode* root, int x, int y) {
        int level [2] = {-1,-1};
        int parents[2] = {-1,-1};
        findNodes(root, x, y, level, parents, 0, new TreeNode(-1));
        if(level[0]==level[1] && parents[0]!=parents[1])
            return true;
        return false;
    }
};
```

# Trees - Part 2

- Karun Karthik

## Contents

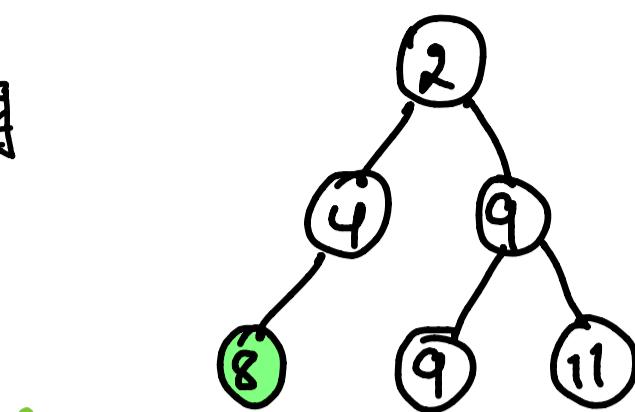
21. Print all nodes that do not have any siblings
22. All nodes at distance K in a Binary Tree
23. Lowest Common Ancestor
24. Level order traversal in Binary Tree
25. Level order traversal in N-ary Tree
26. Top view of Binary Tree
27. Bottom view of Binary Tree
28. Introduction to Binary Search Tree & Search in a BST
29. Insert into a BST
30. Range Sum of BST
31. Increasing order search tree
32. Two Sum IV
33. Delete Node in a BST
34. Inorder successor in BST
35. Validate BST
36. Lowest Common Ancestor of BST
37. Convert Sorted Array to BST
38. Construct BT from Preorder and Inorder traversal
39. Construct BT from Inorder and Postorder traversal
40. Construct BST from Preorder traversal

D7

21

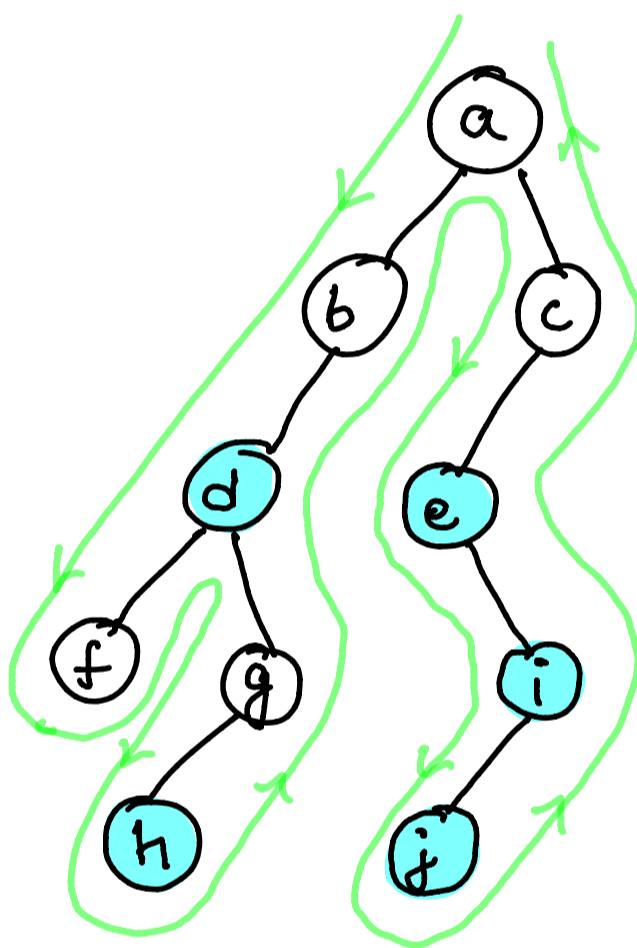
Print all nodes that donot have any siblings

Ex



Result ↗

Sibling  $\rightarrow$  same level, same parent



$Tc \rightarrow O(n)$

$Sc \rightarrow O(n)$

At every node, check if

both branches exist  $\rightarrow$  then call both of them recursively

only left branch exist  $\rightarrow$  then call left branch recursively

only right branch exist  $\rightarrow$  then call right branch recursively

Code →



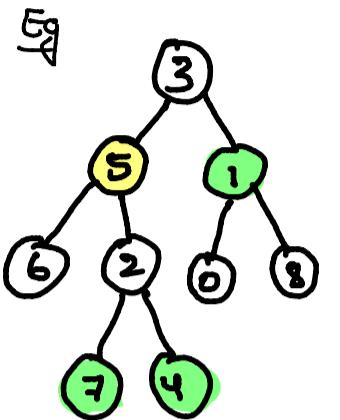
```
1 void findNode(Node* root, vector<int>&res){  
2  
3     if(root==NULL) return;  
4     if(root->left == NULL && root->right==NULL) return;  
5  
6     // both branches present then call recursively  
7     if(root->left!=NULL && root->right!=NULL)  
8     {  
9         findNode(root->left, res);  
10        findNode(root->right, res);  
11    }  
12    else if(root->left!=NULL) // right branch absent  
13    {  
14        res.push_back(root->left->data);  
15        findNode(root->left, res);  
16    }  
17    } else if(root->right!=NULL) // left branch absent  
18    {  
19        res.push_back(root->right->data);  
20        findNode(root->right, res);  
21    }  
22    return;  
23 }  
24  
25 vector<int> noSibling(Node* node)  
26 {  
27     vector<int> res;  
28     findNode(node, res);  
29     if(res.size()==0) res.push_back(-1);  
30     sort(res.begin(), res.end());  
31     return res;  
32 }  
33
```

D8

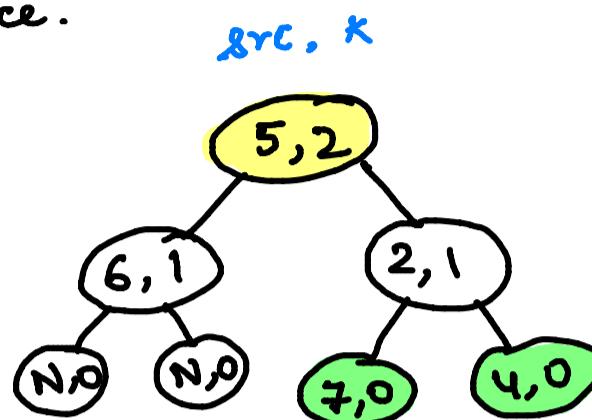
## 22 All nodes distance K in Binary Tree

given a source node, find all the nodes that are at a distance of  $K$  units.

- ① consider nodes in downward direction of source.

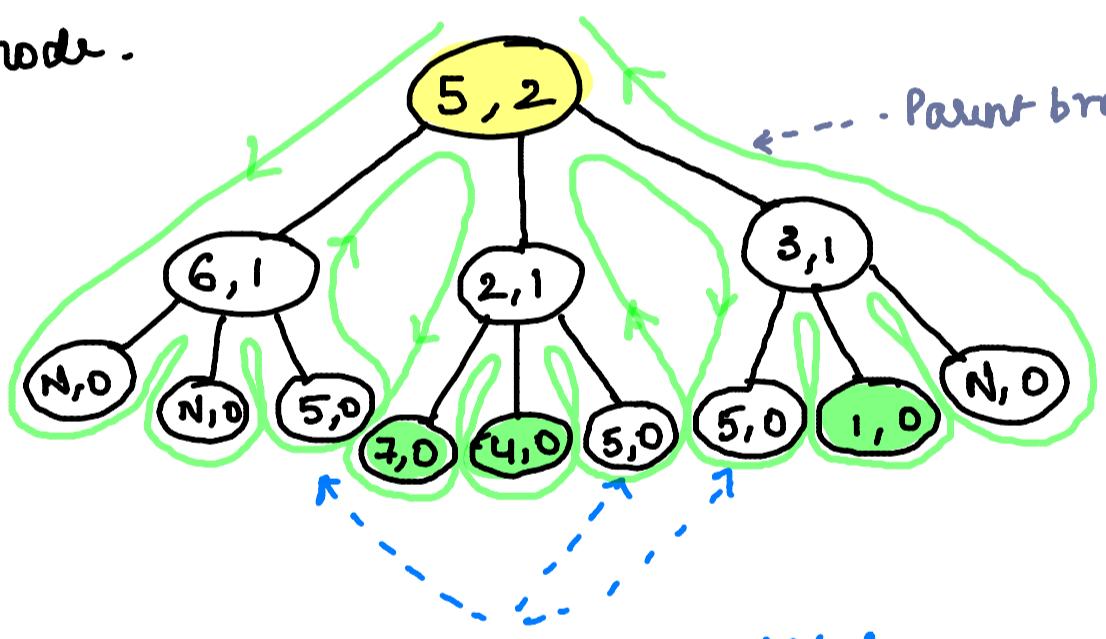
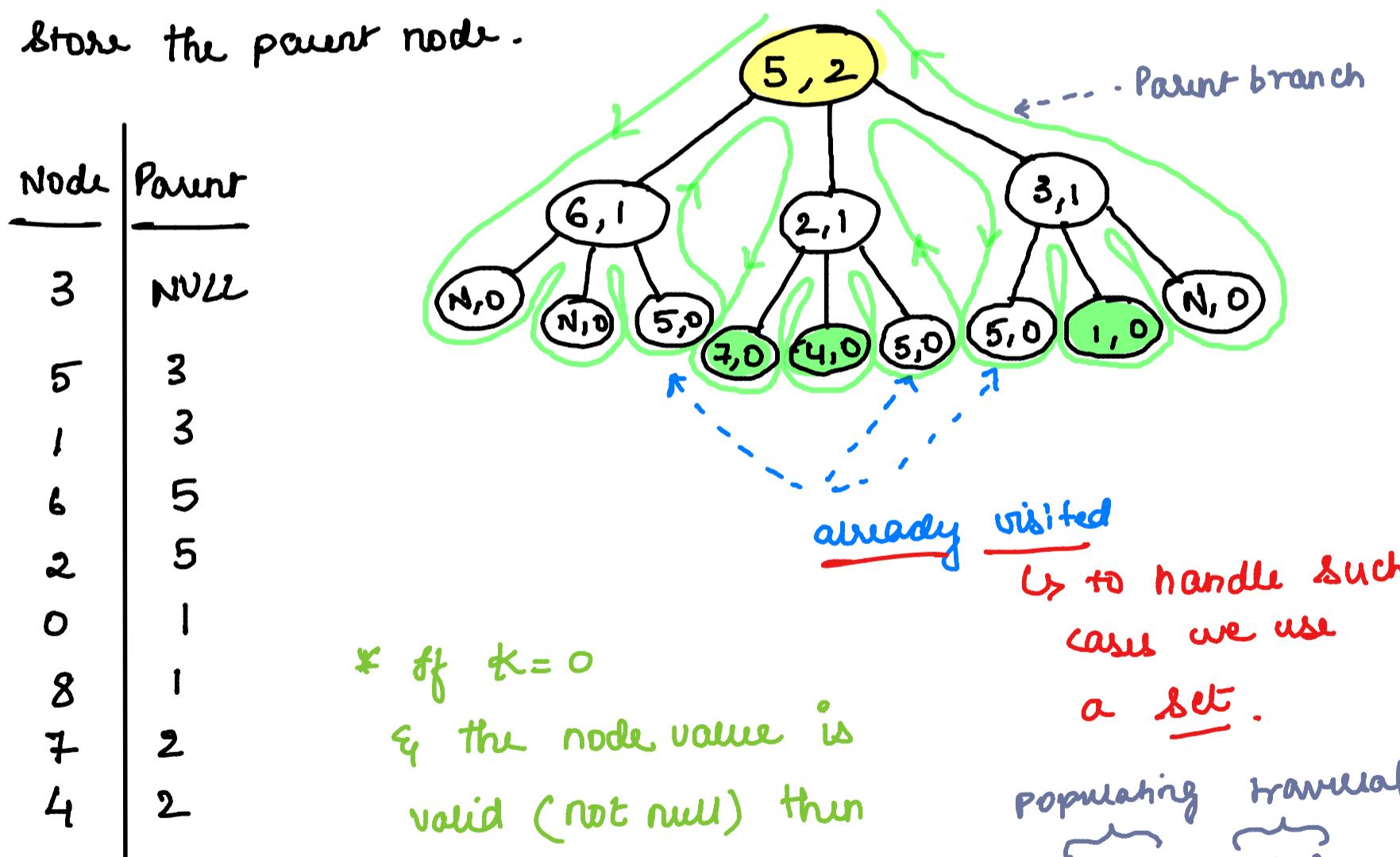


src = 5  
 $k = 2$



- Store in result if  $k = 0$
- Return if  $k < 0$

- ② To solve for the upward direction we can use hashing to store the parent node.



already visited  
↳ to handle such cases we use a set.

populating traversal  
 $Tc \rightarrow O(n) + O(n)$   
 $Sc \rightarrow O(n) + O(n) + O(z)$   
 result

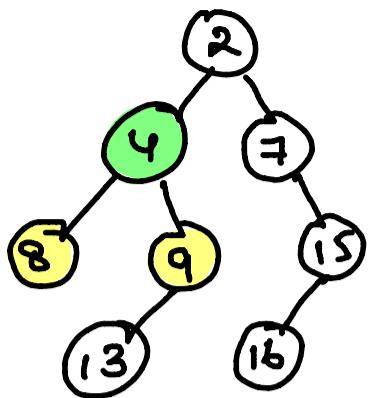
## Code

```
1 class Solution {
2 public:
3     // to create hashtable
4     void populatemap(TreeNode* currnode, TreeNode* currparent,
5                      unordered_map<TreeNode*,TreeNode*>&parentmap){
6         if(currnode == NULL) return;
7         parentmap[currnode] = currparent;
8         populatemap(currnode->left,currnode,parentmap);
9         populatemap(currnode->right,currnode,parentmap);
10        return;
11    }
12
13    // finding all the nodes at distance K
14    void printkdistance(TreeNode* currnode, int k, set<TreeNode*>&s,
15                        unordered_map<TreeNode*,TreeNode*>&parentmap, vector<int>&ans)
16    {
17        if(currnode == NULL || s.find(currnode)!=s.end()|| k<0)
18            return;
19
20        s.insert(currnode);
21
22        if(k==0)
23        {
24            ans.push_back(currnode->val);
25            return;
26        }
27
28        printkdistance(currnode->left,k-1,s,parentmap,ans);    // call left child
29        printkdistance(currnode->right,k-1,s,parentmap,ans);   // call right child
30        printkdistance(parentmap[currnode],k-1,s,parentmap,ans); // call the parent
31        return;
32    }
33
34    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
35        vector<int>ans;
36        set<TreeNode*>s;
37        unordered_map<TreeNode*,TreeNode*>parentmap;
38        populatemap(root,NULL,parentmap);
39        printkdistance(target,k,s,parentmap,ans);
40        return ans;
41    }
42};
```

23

## Lowest Common Ancestor

Ex



node to root paths  
 $\bar{n}_1 = [8, 4, 2]$   
 $\bar{n}_2 = [9, 4, 2]$

$n_1 = 8, n_2 = 9$  then  
 $\bar{n}_1 = [9, 4, 2]$   
 $\bar{n}_2 = [13, 9, 4, 2]$

→ for every node, check if it matches  $n_1$  or  $n_2$ .  
 if found return node  
 else call recursively in both branches.  
 if both return non-null value  $\Rightarrow$  root is LCA  
 else return the branch value that is non-null.

Code

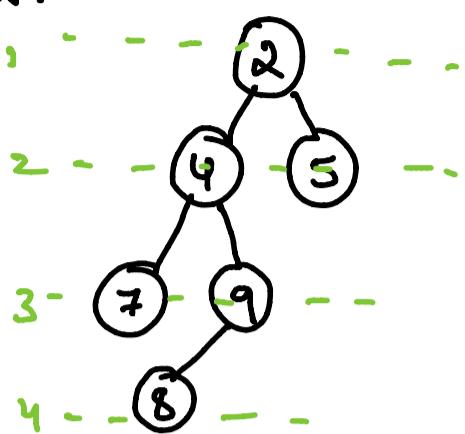
```

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL) return NULL;
        if(root->val == p->val || root->val == q->val) return root;
        TreeNode* leftSubTree = lowestCommonAncestor(root->left, p, q);
        TreeNode* rightSubTree = lowestCommonAncestor(root->right, p, q);
        if(leftSubTree!=NULL && rightSubTree!=NULL) return root;
        if(leftSubTree!=NULL) return leftSubTree;
        if(rightSubTree!=NULL) return rightSubTree;
        return NULL;
    }
};
  
```

D9 24 Level order traversal Binary Tree

Given root node, find level order traversal.

Eg.

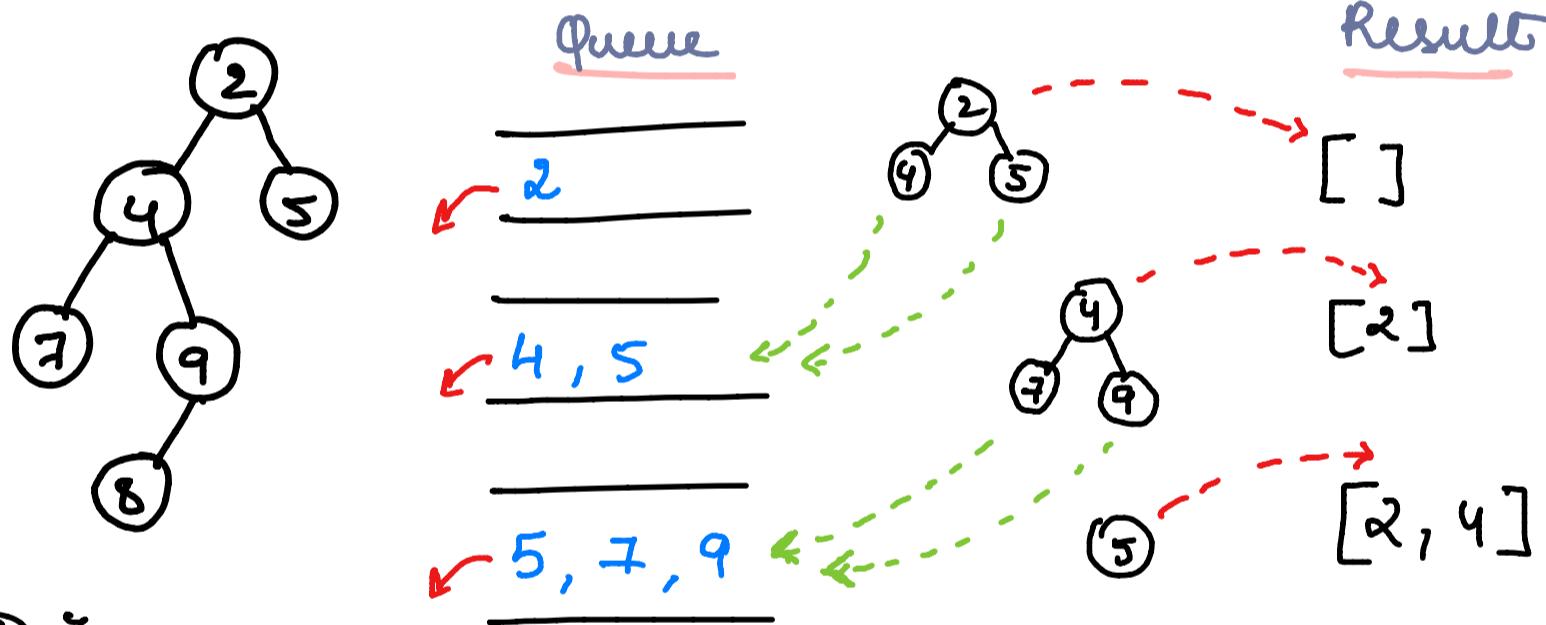


$\Rightarrow [[2], [4, 5], [7, 9], [8]]$

TC  $\rightarrow O(n)$

SC  $\rightarrow O(n)$

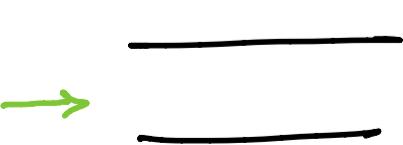
- To find level order traversal use queue. FIFO datastructure
- Before removing from queue, add the children to the queue (BFS)
- Inserting  $\rightarrow$  rear  
Removing  $\rightarrow$  front



① For every node, enqueue.

② While dequeue, enqueue its branches.

empty  $\rightarrow$



Answer  $\rightarrow [2, 4, 5, 7, 9, 8]$

Code →

```
1 class Solution {
2 public:
3     vector<vector<int>> levelOrder(TreeNode* root) {
4         vector<vector<int>> res;
5         queue<TreeNode*> q;
6
7         if(root==NULL) return res;
8         q.push(root);
9
10        while(!q.empty()){
11
12            int currsize = q.size();
13            vector<int>currLevel;
14
15            while(currsize>0)
16            {
17                TreeNode* currnode = q.front();
18                q.pop();
19                currLevel.push_back(currnode->val);
20                currsize--;
21
22                if(currnode->left!=NULL)
23                    q.push(currnode->left);
24
25                if(currnode->right!=NULL)
26                    q.push(currnode->right);
27            }
28            res.push_back(currLevel);
29        }
30        return res;
31    }
32};
```

25

## Level Order Traversal N-ary Tree

→ Everything is same as previous problem, intuition & complexity

$T_C \rightarrow O(n)$

$S_C \rightarrow O(n)$

code →



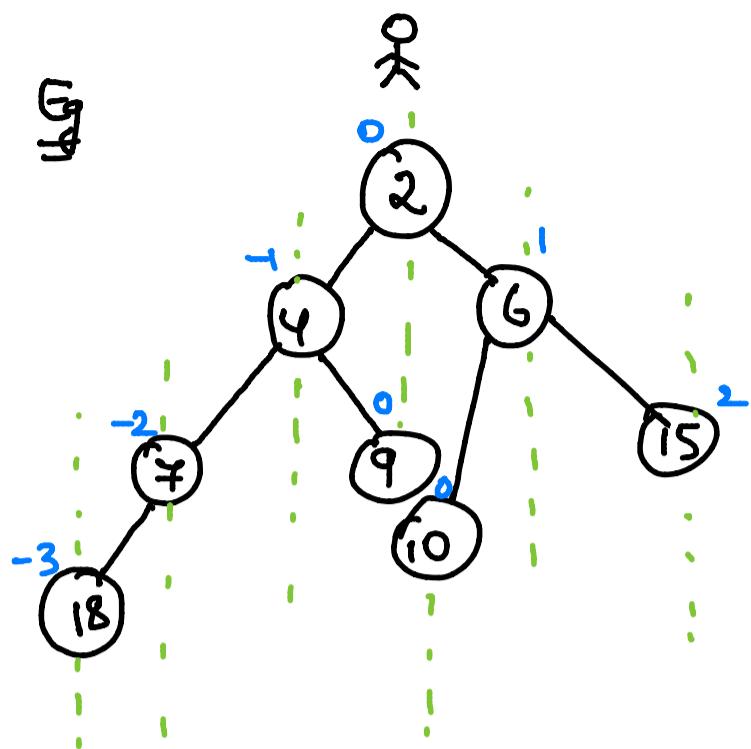
```

1  class Solution {
2  public:
3      vector<vector<int>> levelOrder(Node* root) {
4          vector<vector<int>> res;
5          queue<Node*>q;
6
7          if(root == NULL) return res;
8          q.push(root);
9
10         while(!q.empty())
11         {
12             int currsize = q.size();
13             vector<int>currLevel;
14             while(currsize>0)
15             {
16                 Node* currnode = q.front();
17                 q.pop();
18                 currLevel.push_back(currnode->val);
19                 currsize--;
20
21                 // enqueue all the children
22                 for(auto child:currnode->children)
23                     q.push(child);
24             }
25             res.push_back(currLevel);
26         }
27         return res;
28     }
29 };

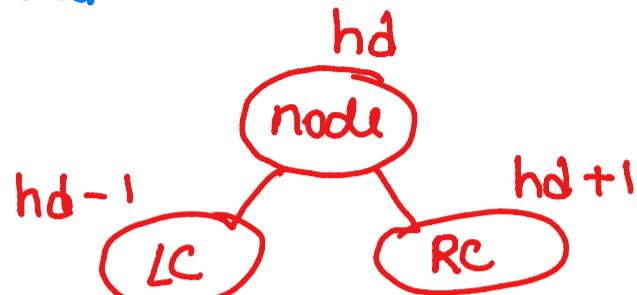
```

26

## Top View of Binary Tree



\* For top view or bottom view we use concept of horizontal distance.



\* hd of root = 0

Left to Right  
 $\hookrightarrow [18, 7, 4, 2, 6, 15]$  \* make a pair with node & its hd.  
 & perform bfs.

<node, hd>

①	②	③	④	⑤	⑥	⑦	⑧
(2, 0)	(4, -1)	(6, 1)	(7, -2)	(9, 0)	(10, 0)	(15, 2)	(18, -3)

use a hashmap to store result.

- ① As hd = 0 is not present in map add 2 to map.
- ② As hd = -1 is not present in map add 4 to map.
- ③ As hd = 1 is not present in map add 6 to map.
- ④ As hd = -2 is not present in map add 7 to map.
- ⑤ hd = 0 is already present.
- ⑥ hd = 0 is already present.
- ⑦ As hd = 2 is not present in map add 15 to map.
- ⑧ As hd = -3 is not present in map add 18 to map.

HD	NODE
0	2
-1	4
1	6
-2	7
2	15
-3	18

convert into array & return as result

## code

```
1 class Solution
2 {
3     public:
4     vector<int> topView(Node *root)
5     {
6         vector<int> res;
7         if(root==NULL) return res;
8
9         map<int,int> mp;
10        queue<pair<Node*,int>> q;
11
12        q.push({root,0});
13
14        while(!q.empty()){
15
16            auto it = q.front();
17            q.pop();
18
19            Node* node = it.first;
20            int hd = it.second;
21
22            if(mp.find(hd) == mp.end())
23                mp[hd] = node->data;
24
25            if(node->left!=NULL)
26                q.push({node->left,hd-1});
27
28            if(node->right!=NULL)
29                q.push({node->right,hd+1});
30        }
31
32
33        // store in vector or array
34        for(auto it:mp)
35            res.push_back(it.second);
36
37        return res;
38    }
39 };
40
```

$\log n \rightarrow \text{map.}$

$T_C \rightarrow O(n \log n)$

$S_C \rightarrow O(n)$

## 27 Bottom View of Binary Tree

→ Similar to top view, but replace entries in hashmap so you'll get last possible element with particular hd.

Code →



```
1 class Solution {
2     public:
3         vector <int> bottomView(Node *root) {
4             vector<int> res;
5             if(root==NULL) return res;
6
7             map<int, int> mp;
8             queue<pair<Node*, int>>q;
9
10            q.push({root, 0});
11            while(!q.empty()){
12                auto it = q.front();
13                q.pop();
14
15                Node* node = it.first;
16                int hd = it.second;
17
18                mp[hd] = node->data;
19
20                if(node->left!=NULL)
21                    q.push({node->left, hd-1});
22
23                if(node->right!=NULL)
24                    q.push({node->right, hd+1});
25            }
26
27            for(auto it:mp)
28                res.push_back(it.second);
29
30            return res;
31        }
32    };
```

D10

## Binary Search Tree

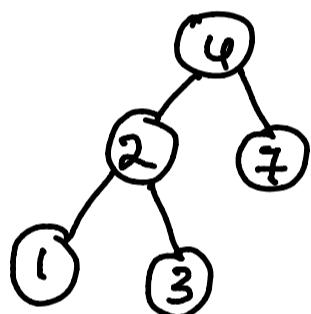
- every node is  $>$  than previous node &  $<$  than next node.
- if duplicates, then it'll be mentioned that it'll be included in LC or RC

$$\textcircled{1} \quad LC < \text{node} < RC$$

$$\textcircled{2} \quad LC \leq \text{node} < RC$$

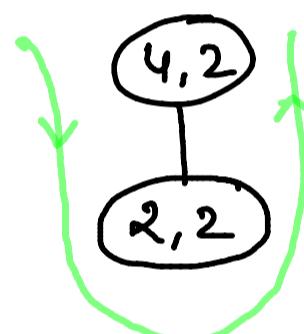
$$\textcircled{3} \quad LC < \text{node} \leq RC$$

## (28) Search in a BST



val = 2

$\Rightarrow$  return the subtree with given value.



• as  $2 < 4$ , search in LST.

• as  $2 == 2$  return node.

TC  $\rightarrow O(\log_2 n)$ , O(n)  
avg worst

SC  $\rightarrow O(n)$

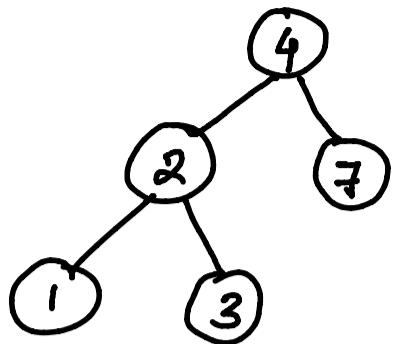
code

```

class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root==NULL) return NULL;
        if(root->val == val) return root;
        if(root->val < val) return searchBST(root->right, val);
        return searchBST(root->left, val);
    }
};
  
```

29

## Insert into BST



val = 5.

TC  $\rightarrow O(\log_2 n)$ , avg  
worst

SC  $\rightarrow O(1)$ 

- 1)   
As  $5 > 4$ , go to RST
- 2)   
As  $5 < 7$ , go to LST
- 3) • As LST of 7 is null, create node with value = 5. (5)  
• Link (5) as LST of 7.
- 4) Result  $\Rightarrow$ 

```

graph TD
    4((4)) --- 2((2))
    4 --- 7((7))
    2 --- 1((1))
    2 --- 3((3))
    7 --- 5((5))
  
```

Code  $\rightarrow$

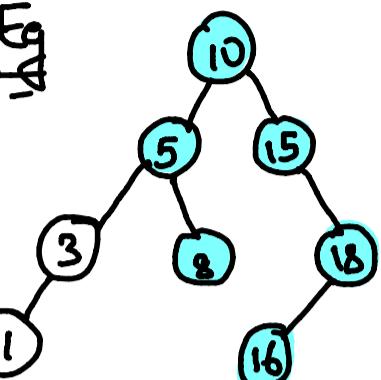
```

class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* node, int val) {
        if(node==NULL){
            return new TreeNode(val);
        }
        if (val < node->val) {
            node->left = insertIntoBST(node->left, val);
        }
        else {
            node->right = insertIntoBST(node->right, val);
        }
        return node;
    }
};
  
```

### 30 Range sum of BST

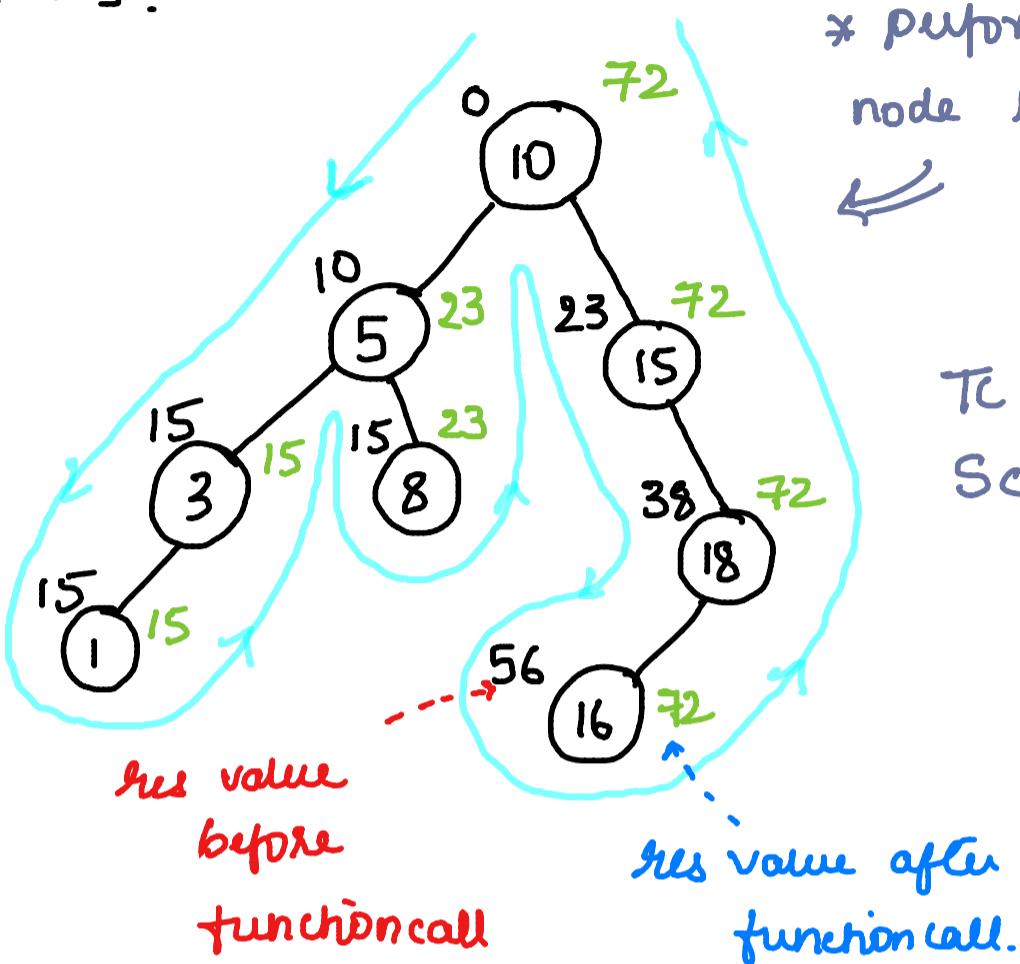
given a root node & interval  $[x, y]$ , find sum of all nodes that lies in  $[x, y]$ .

Eg



range  $\rightarrow [5, 18]$

sum = 72



Code  $\rightarrow$

```

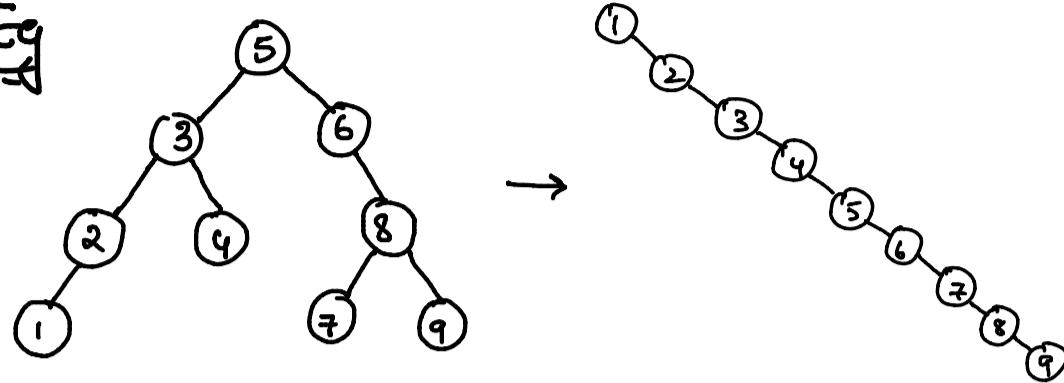
● ● ●
1 class Solution {
2 public:
3     void sumUtil(TreeNode* root, int low, int high, int &res){
4         if(root==NULL) return;
5         if(root->val <= high && root->val >= low){
6             res += root->val;
7         }
8         sumUtil(root->left, low, high, res);
9         sumUtil(root->right, low, high, res);
10    }
11
12    int rangeSumBST(TreeNode* root, int low, int high) {
13        int res = 0;
14        sumUtil(root, low, high, res);
15        return res;
16    }
17 };
  
```

31

## Increasing order search tree

Given a BST, create an increasing order search tree.

Ex



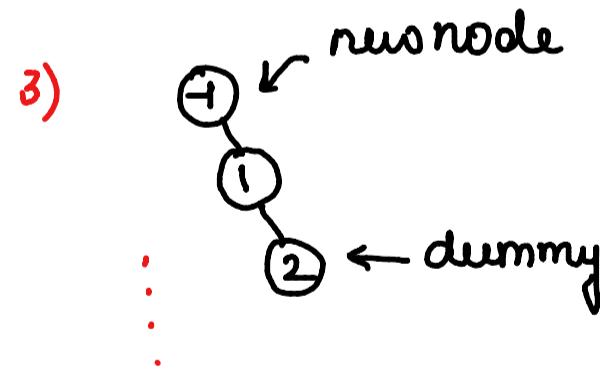
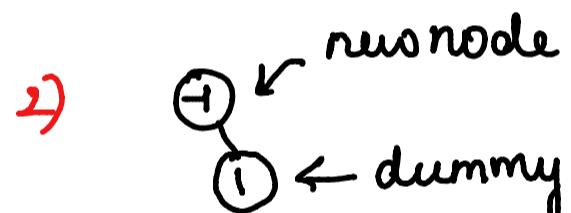
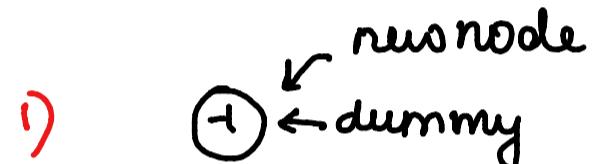
- ① Perform inorder traversal.
- ② Create a skewed tree using elements in inorder traversal.

Code

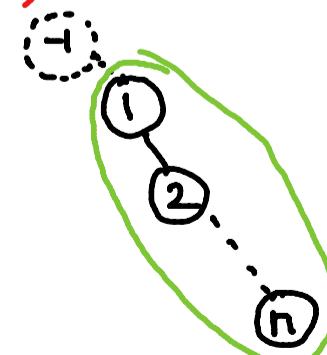
```

1 class Solution {
2 public:
3     void inorder(TreeNode* root, vector<int> &res){
4         if(root==NULL) return;
5         inorder(root->left, res);
6         res.push_back(root->val);
7         inorder(root->right, res);
8     }
9     TreeNode* increasingBST(TreeNode* root) {
10        vector<int> res;
11        inorder(root, res);
12
13        // create right skewed tree
14        TreeNode* dummy = new TreeNode(-1);
15        TreeNode* newNode = dummy;
16        for(auto it: res){
17            dummy->right = new TreeNode(it);
18            dummy = dummy->right;
19        }
20        return newNode->right;
21    }
22 };

```

Lines 16-20

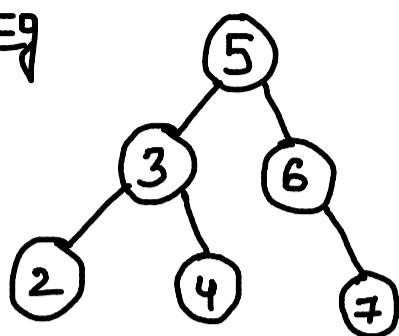
last) return newNode->right



32

## Two sum IV - Input is a BST

Ex



↳ returns true if sum of any 2 values == k

① Perform Inorder & store in array

② use 2-pointer approach

k = 9

$\Rightarrow v = \boxed{2 | 3 | 4 | 5 | 6 | 7}$

$f$                                      $r$

as  $v[f] + v[r] == k$ , return true, else  $f++$  or  $r--$   
as per sum & k.

Code →

```

class Solution {
public:
    void inorder(TreeNode* root, vector<int> &res){
        if(root==NULL) return;
        inorder(root->left, res);
        res.push_back(root->val);
        inorder(root->right, res);
    }
    bool findTarget(TreeNode* root, int k) {
        vector<int> res;
        inorder(root, res);
        int front = 0;
        int rear = res.size()-1;
        while(front<rear){
            if(res[front]+res[rear]==k) return true;
            if(res[front]+res[rear]>k) rear--;
            else front++;
        }
        return false;
    }
};
  
```

Tc → O(n)+O(n)

Sc → O(n)

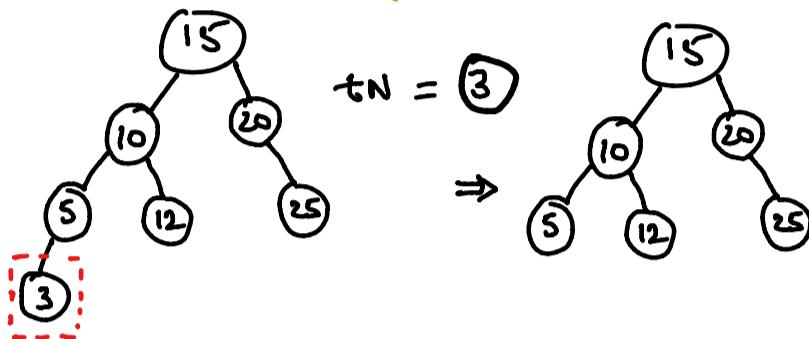
D11

### 33 Delete Node in BST

given root of BST & a target node, delete the target node & return the tree.

Cases →

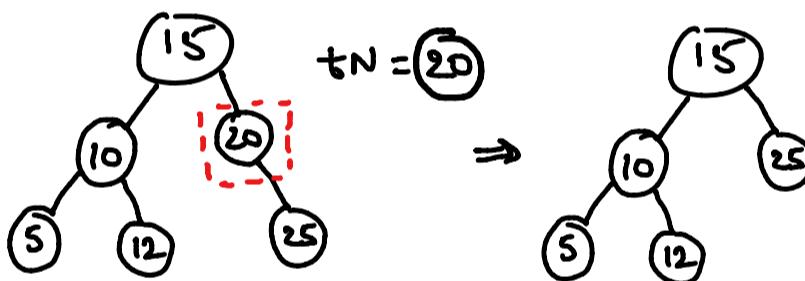
- ① If target node is leaf →  
then simply delete it



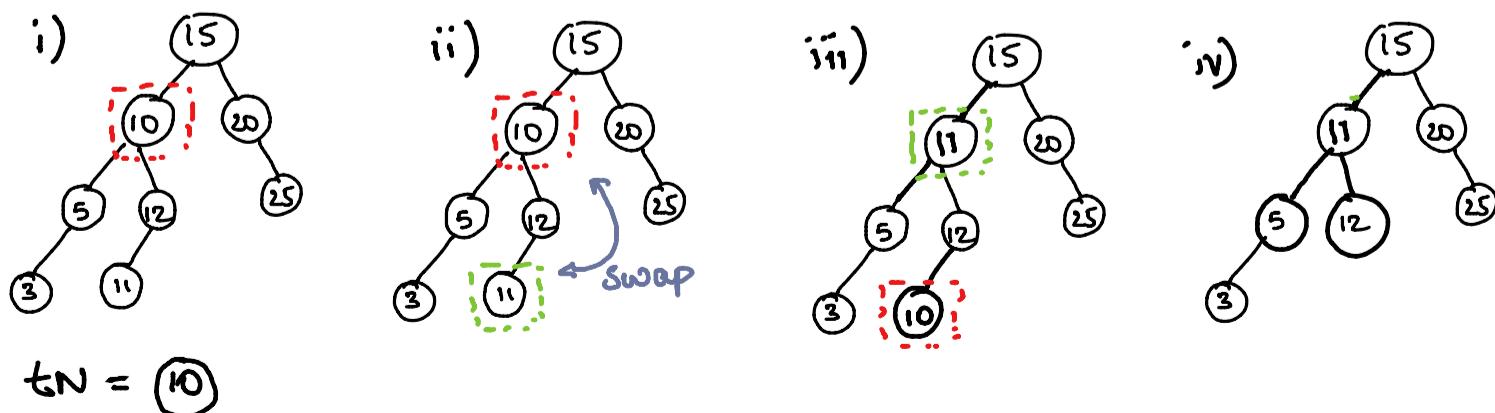
Tc →  
Avg  $\Rightarrow O(\log n)$   
Worst  $\Rightarrow O(n)$

SC  $\rightarrow O(h)$

- ② If target node has 1 child →  
then remove node & return the subtree



- ③ If target node has 2 children →  
then go to right child's left subtree & swap its  
value with target node & then delete it.



## Code

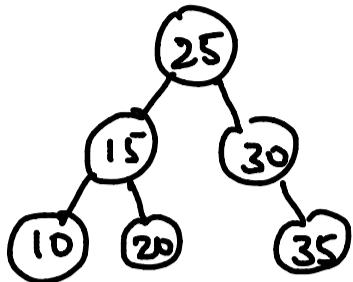
```
● ○ ●
1 class Solution {
2 public:
3     TreeNode* findleftmostNode(TreeNode* root){
4         while(root->left!=NULL)
5             root = root->left;
6         return root;
7     }
8
9     TreeNode* deleteNode(TreeNode* root, int key) {
10
11         if(root==NULL)  return NULL;
12
13         if(root->val > key)
14             root->left = deleteNode(root->left, key);
15
16         else if(root->val < key)
17             root->right = deleteNode(root->right, key);
18
19         else { // root->val == key
20             if(root->left == NULL && root->right == NULL){
21                 root = NULL;
22                 return root;
23             }
24             if(root->left != NULL && root->right == NULL){
25                 root = root->left;
26                 return root;
27             }
28             if(root->right != NULL && root->left == NULL){
29                 root = root->right;
30                 return root;
31             }
32
33             // finding left most node in right subtree
34             TreeNode* temp = findleftmostNode(root->right);
35
36             //swapping root's value with left most node's val
37             int tempVal = root->val;
38             root->val = temp->val;
39             temp->val = tempVal;
40
41             // performing delete in right subtree
42             root->right = deleteNode(root->right, key);
43             return root;
44         }
45     }
46 }
47 };
```

### 34 Inorder successor of BST

given root, find inorder successor of given node

↳ the element just after the node in  
inorder traversal.

Eg



$n = 15 \quad O/p \rightarrow 20$

$n = 35 \quad O/p \rightarrow \text{null}$ .

Code →

```
class Solution{
public:

    void inorder(Node *root, vector<Node*> &res){
        if(root==NULL) return;
        inorder(root->left, res);
        res.push_back(root);
        inorder(root->right, res);
    }

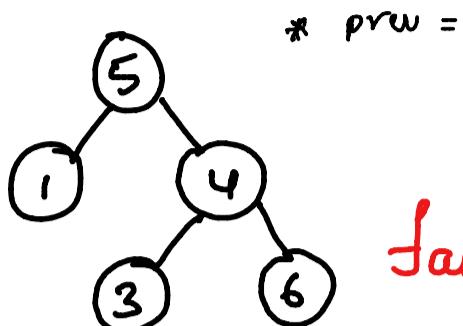
    Node * inOrderSuccessor(Node *root, Node *x)
    {
        vector<Node*> res;
        inorder(root, res);
        for(int i=0; i<res.size(); i++){
            if(res[i]==x && i<res.size()-1){
                return res[i+1];
            }
        }
        return NULL;
    }
};
```

D12 35 Validate BST

Given a root node ,  
returns true if it is valid BST

- \* Every value should be less than previous one in Inorder traversal

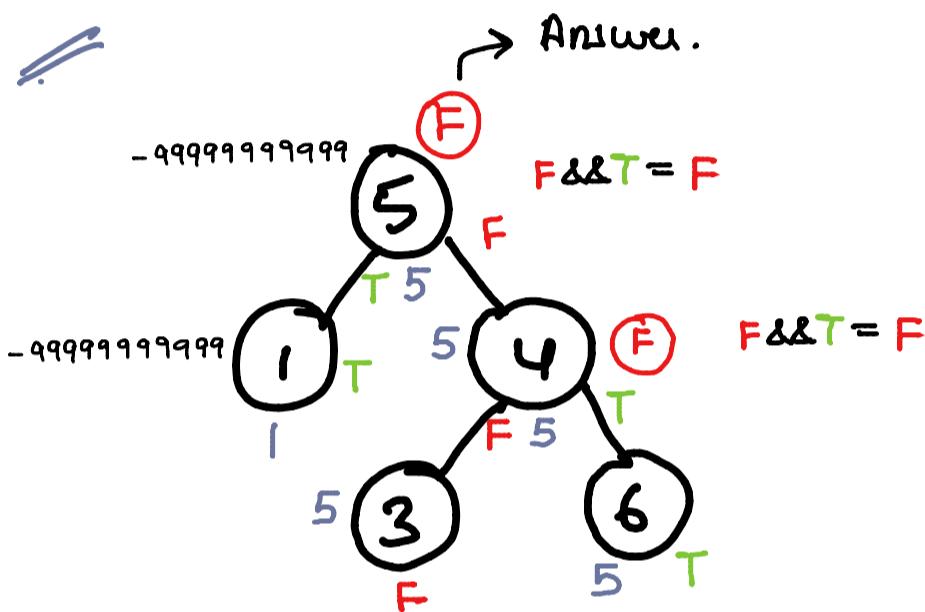
Eg



\* prw = -999999999999

False

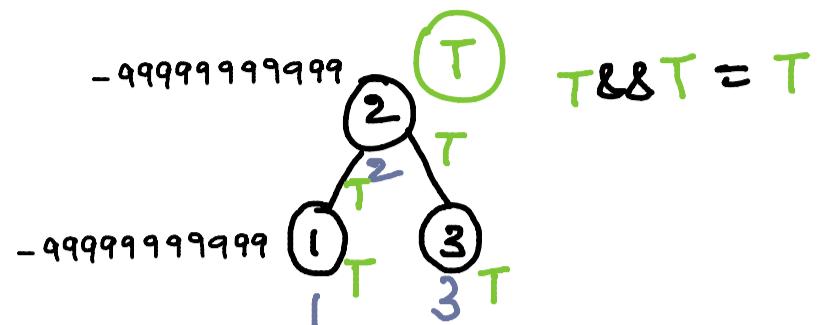
Ans.



code

```
class Solution {
public:
    bool isBST(TreeNode* root, long int &prev){
        if(root==NULL) return true;
        bool isLeftBalanced = isBST(root->left, prev);
        if(root->val <= prev) return false;
        prev = root->val;
        bool isRightBalanced = isBST(root->right, prev);
        return isLeftBalanced && isRightBalanced;
    }

    bool isValidBST(TreeNode* root) {
        long int prev = -999999999999;
        return isBST(root, prev);
    }
};
```



-999999999999

-999999999999

T && T = T

→ Returns True on NULL nodes

→ Check for Left subtree

→ previous value gets updated  
before checking Right subtree  
& after checking Left subtree

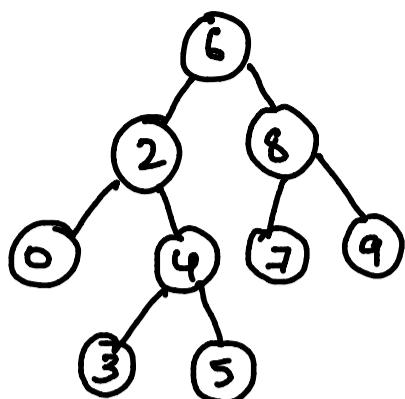
→ if curVal <= previous then  
return false

→ return true if both LST & RST  
are BST

36

## LCA of BST →

Ex.



$$P=2, Q=8$$

if  $\text{currNode} > \text{both } P \text{ & } Q$   
 then LCA lies in LST  
 if  $\text{currNode} < \text{both } P \text{ & } Q$   
 then LCA lies in RST  
 in every other case the currNode is  
 LCA as  $P \text{ & } Q$  will be on

	worst	Avg
$T_C \rightarrow O(n)$		$O(\log n)$
$S_C \rightarrow O(n)$		

code

```

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL) return NULL;

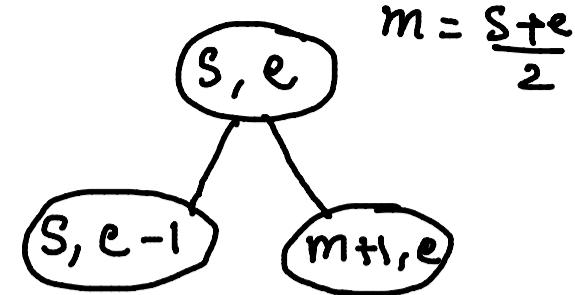
        if(root->val < p->val && root->val < q->val){
            return lowestCommonAncestor(root->right, p, q);
        }
        else if(root->val > p->val && root->val > q->val){
            return lowestCommonAncestor(root->left, p, q);
        }
        else {
            return root;
        }
    }
};
  
```

37

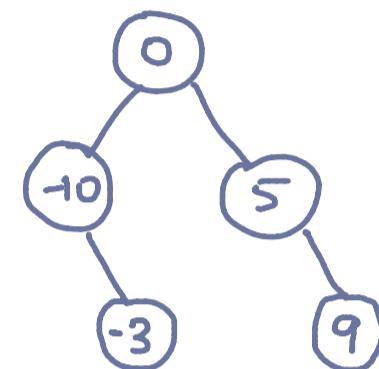
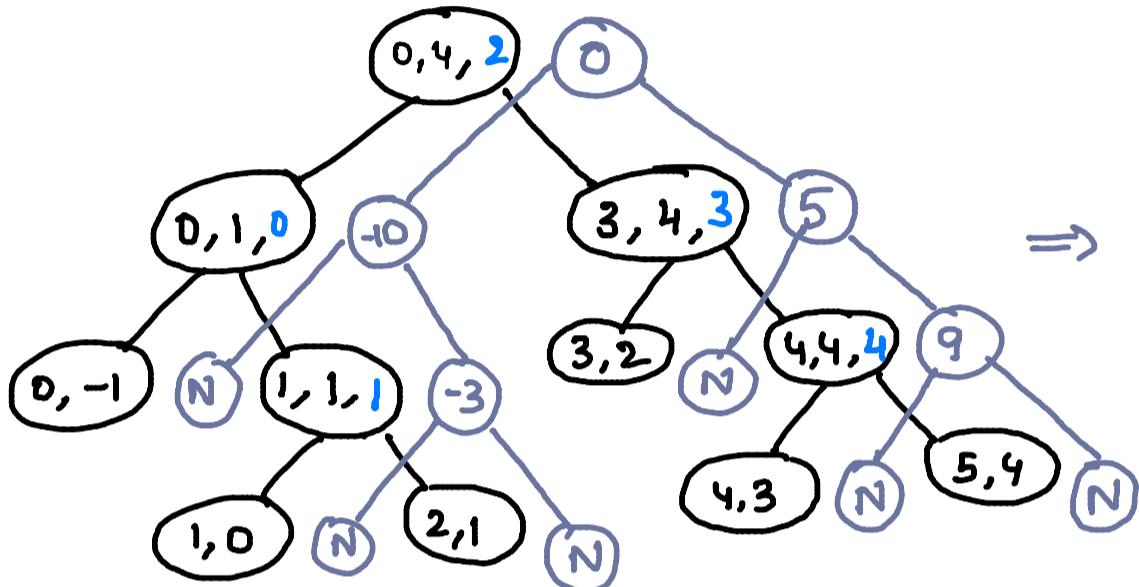
## Convert Sorted array to BST

Given sorted array, create a BST

Eg  $[-10, -3, 0, 5, 9]$



start, end, mid



Code →

```
class Solution {
public:
    TreeNode* createBST(vector<int>& nums, int start, int end){
        if(start > end)    return NULL;

        int mid = (start + end)/2;
        TreeNode* root = new TreeNode(nums[mid]);

        root->left = createBST(nums, start, mid-1);
        root->right = createBST(nums, mid+1, end);
        return root;
    }

    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return createBST(nums, 0, nums.size()-1);
    }
};
```

DI3

### (38) Construct Binary Tree from Pre & Inorder traversal

Eg   
 pre = [3, 9, 20, 15, 7]   
 in = [9, 3, 15, 20, 7]

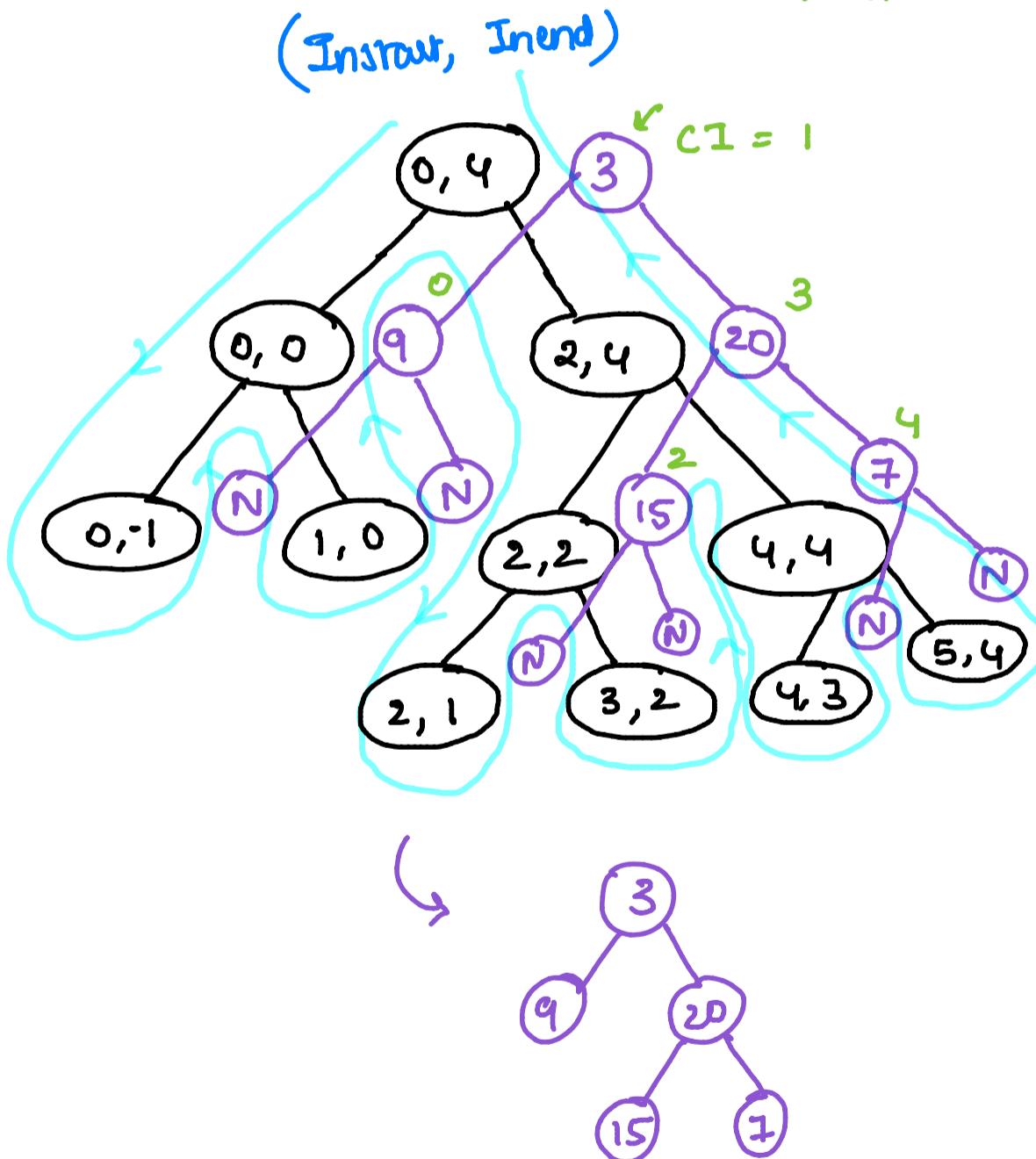
Tc  $\rightarrow O(n^2)$ Sc  $\rightarrow O(n)$ 

\* for every node in Pre, the corresponding LST & RST are in In

i.e. 3  $\rightarrow$  [ LST CI RST ]  
 [ 9, 3, 15, 20, 7 ]  
 0 1 2 3 4

$$\begin{aligned} LST &= (instart, CI-1) \\ RST &= (CI+1, inend) \end{aligned}$$

CI = index of pre[0]  
in In



0 1 2 3 4  
 pre = [3, 9, 20, 15, 7]  
 in = [9, 3, 15, 20, 7]  
 ,

- ① for pre-order index = 0,  
in-order boundary = [0, 4]
- ② find root value in  
Inorder array & its  
index is currIndex
- ③ if instart > CI-1  
or CI+1 < inend  
returns NULL

To reduce Tc  
we can use  
hashTable to find  
indexing

Tc  $\rightarrow O(n)$   
 Sc  $\rightarrow O(n) + O(n)$

## Code →

```
1 class Solution {
2 public:
3     TreeNode* constructTree(vector<int>& preorder, unordered_map<int, int> &mp,
4     int start, int end, int &preIdx ){
5
6         if(start>end)    return NULL;
7         TreeNode* root = new TreeNode(preorder[preIdx]);
8
9         // find currIndex as per inorder array
10        int currIdx = mp[preorder[preIdx]];
11        // increment preIdx to find next root
12        preIdx++;
13
14        // recursively call LST & RST
15        root->left = constructTree(preorder, mp, start, currIdx-1, preIdx);
16        root->right = constructTree(preorder, mp, currIdx+1, end, preIdx);
17        return root;
18    }
19
20    unordered_map<int,int> populate(vector<int>&inorder){
21        unordered_map<int,int> mp;
22        for(int i=0; i<inorder.size(); i++){
23            mp[inorder[i]] = i;
24        }
25        return mp;
26    }
27
28    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
29        unordered_map<int,int> mp = populate(inorder);
30        int preIdx = 0;
31        return constructTree(preorder, mp, 0, inorder.size()-1, preIdx);
32    }
33 };
34 }
```

### 39) Construct Binary Tree from In & Postorder traversals

Intuition is same as previous program, only changes are

- traverse from last element in postorder array
- process RST & then go for LST

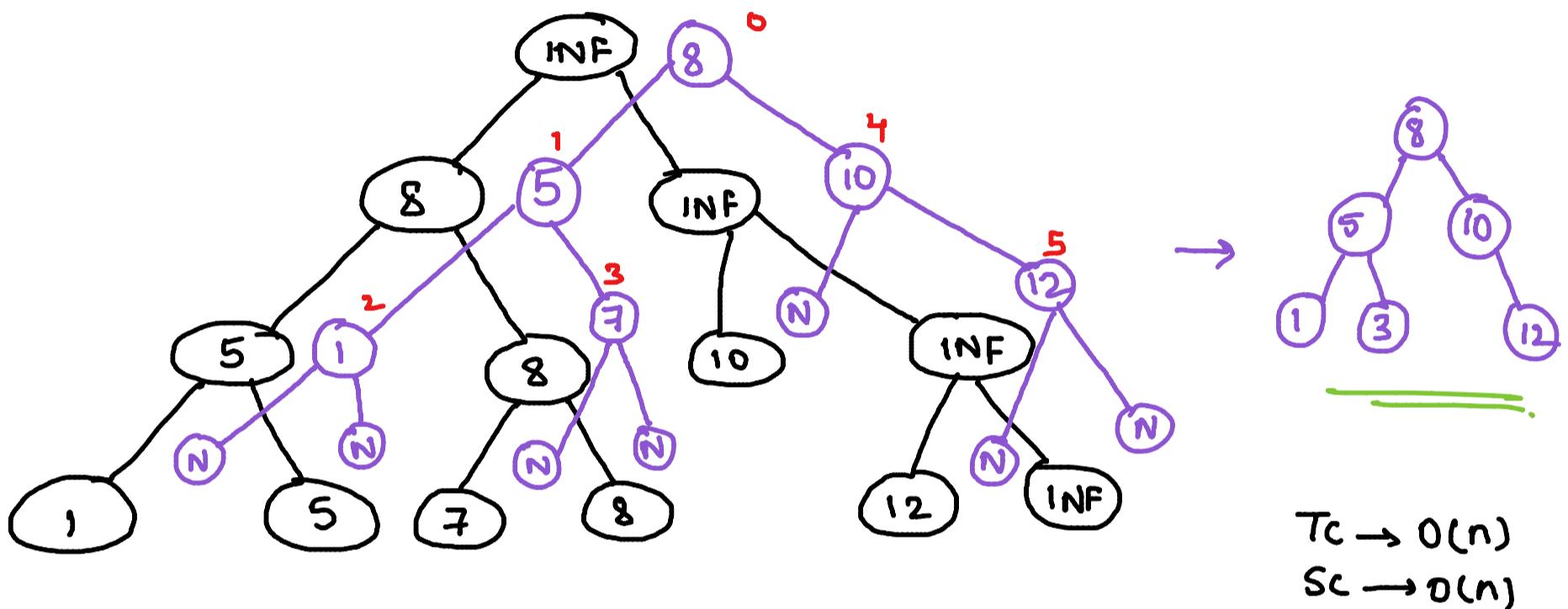
Code →

```
1 class Solution {
2 public:
3
4     TreeNode* constructTree(vector<int>& postorder, unordered_map<int, int> &mp,
5     int start, int end, int &postIdx ){
6
7         if(start>end)    return NULL;
8         TreeNode* root = new TreeNode(postorder[postIdx]);
9
10        // find currIndex as per inorder array
11        int currIdx = mp[postorder[postIdx]];
12        postIdx--;
13
14        // recursively call RST & LST
15        root->right = constructTree(postorder, mp, currIdx+1, end, postIdx);
16        root->left = constructTree(postorder, mp, start, currIdx-1, postIdx);
17        return root;
18    }
19
20    unordered_map<int,int> populate(vector<int>&inorder){
21        unordered_map<int,int> mp;
22        for(int i=0; i<inorder.size(); i++){
23            mp[inorder[i]] = i;
24        }
25        return mp;
26    }
27
28    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
29        unordered_map<int,int> mp = populate(inorder);
30        int postIdx = postorder.size()-1;
31        return constructTree(postorder, mp, 0, inorder.size()-1, postIdx);
32    }
33};
```

(40)

## Construct BST from Preorder traversal

[8, 5, 1, 7, 10, 12]

TC  $\rightarrow O(n \log n)$  (due to sorting)Approach 1  $\rightarrow$  Sort given Preorder to get Inorder, now similar to problem 38.Approach 2  $\rightarrow$ [8, 5, 1, 7, 10, 12]  
0 1 2 3 4 5Boundary of LST  $\rightarrow$  Val  
RST  $\rightarrow$  boundVal  $\rightarrow$  initially (INF)Code  $\rightarrow$ 

```

1 class Solution {
2 public:
3     TreeNode* buildTree(vector<int>& preorder, int &preIdx, int boundary){
4         if(preIdx >= preorder.size() || preorder[preIdx] >= boundary)
5             return NULL;
6
7         // create root using preIdx
8         TreeNode* root = new TreeNode(preorder[preIdx]);
9         preIdx++;
10
11        // recursively call LST & RST
12        root->left = buildTree(preorder, preIdx, root->val);
13        root->right = buildTree(preorder, preIdx, boundary);
14        return root;
15    }
16
17    TreeNode* bstFromPreorder(vector<int>& preorder) {
18        int preIdx = 0;
19        return buildTree(preorder, preIdx, 1001);
20    }
21 };
22

```

Find the rest on  
<https://linktr.ee/KarunKarthik>

Follow **Karun Karthik** for more!