Trees - Part 1

- Karun Karthik

Contents

Trees

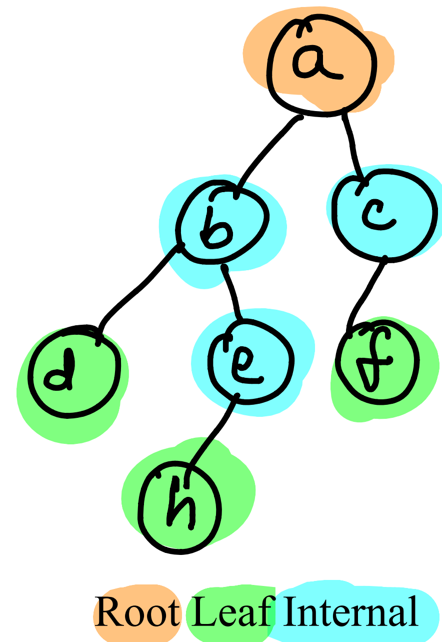
Tree - collection of tree-nodes

why trees?

1. Hirarchy
2. Computer system.
   (UNIX)

① Class Treenode
- → data
- → list<Treenode> children

② Binary Tree → atmost 2 children (0,1,2)
- data
- → leftchild
- → rightchild

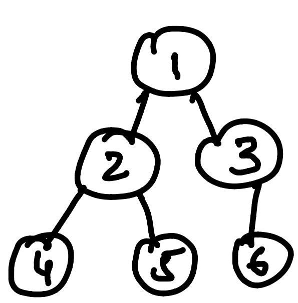


③ Types →

Ⓐ Complete Binary Tree

→ all levels are completely filled except last one

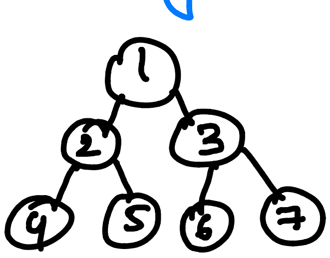
Eg



Ⓑ Perfect Binary Tree

→ every internal node has exactly 2 children

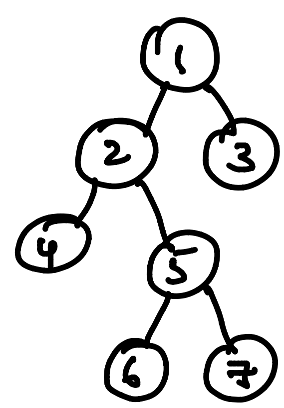
Eg



Ⓒ Full Binary tree

→ if every node has 0 or 2 children

Eg

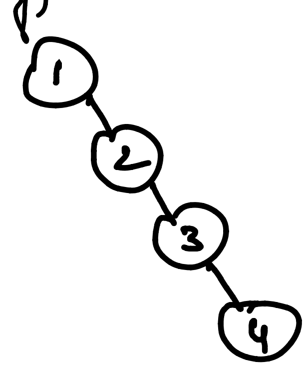


Ⓓ Skewed Binary Tree

(* used for finding complexity)

→ all nodes have either one or no child.

Eg

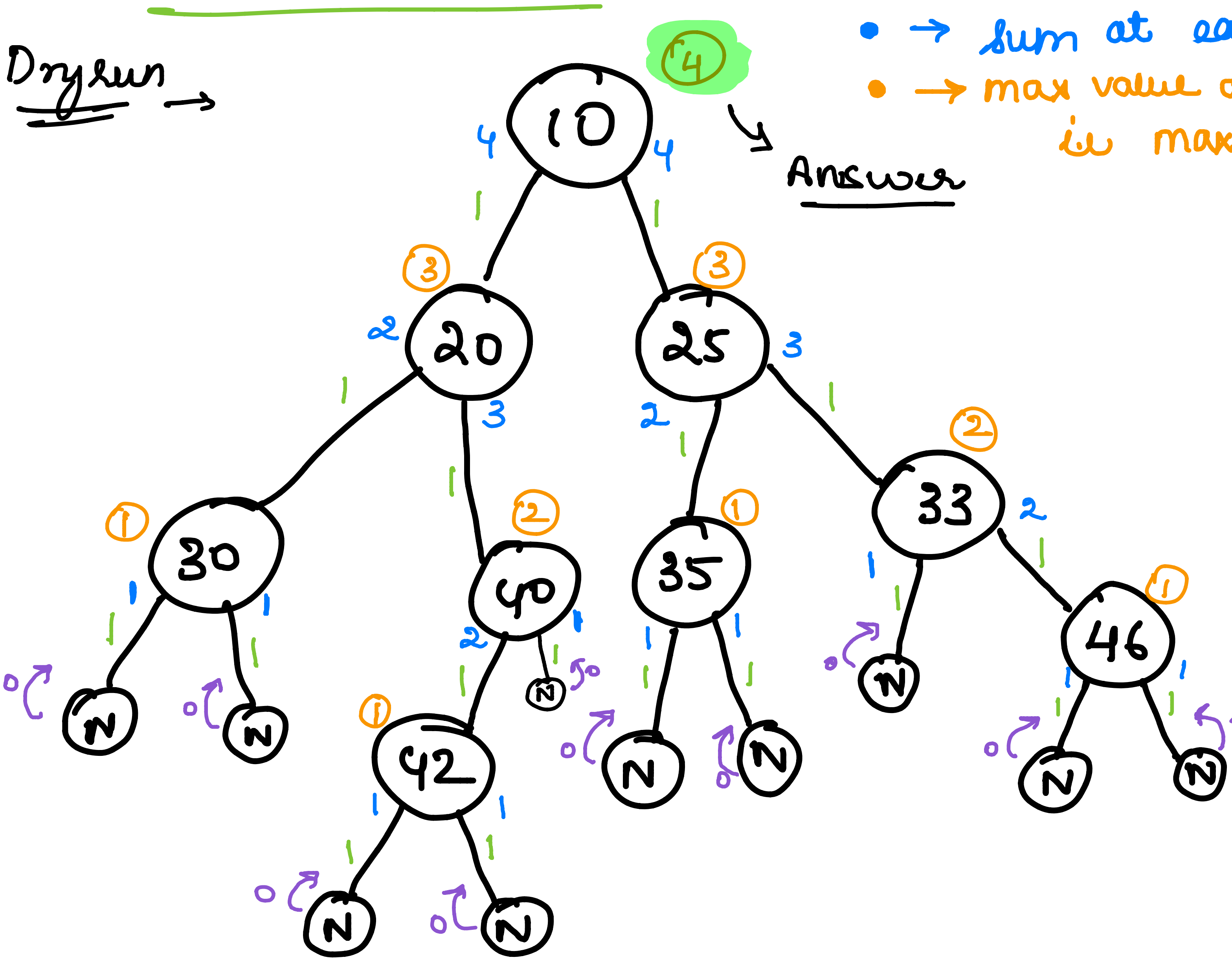


D1

① Depth of a binary tree (Max depth)

- → 1 added while returning.
- → sum at each node.
- → max value at node i.e max(left, right).

Dryrun →



if null then ht = 0

- consider max at a node i.e either left or right

TC → O(n)
SC → O(1)
AUX → O(h)
h → height

Code →

```cpp
C++ ˅
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(n
 *     TreeNode(int x) : val(x), left(nullptr), ri
 *     TreeNode(int x, TreeNode *left, TreeNode *r
 * };
 */
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 →

```cpp
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

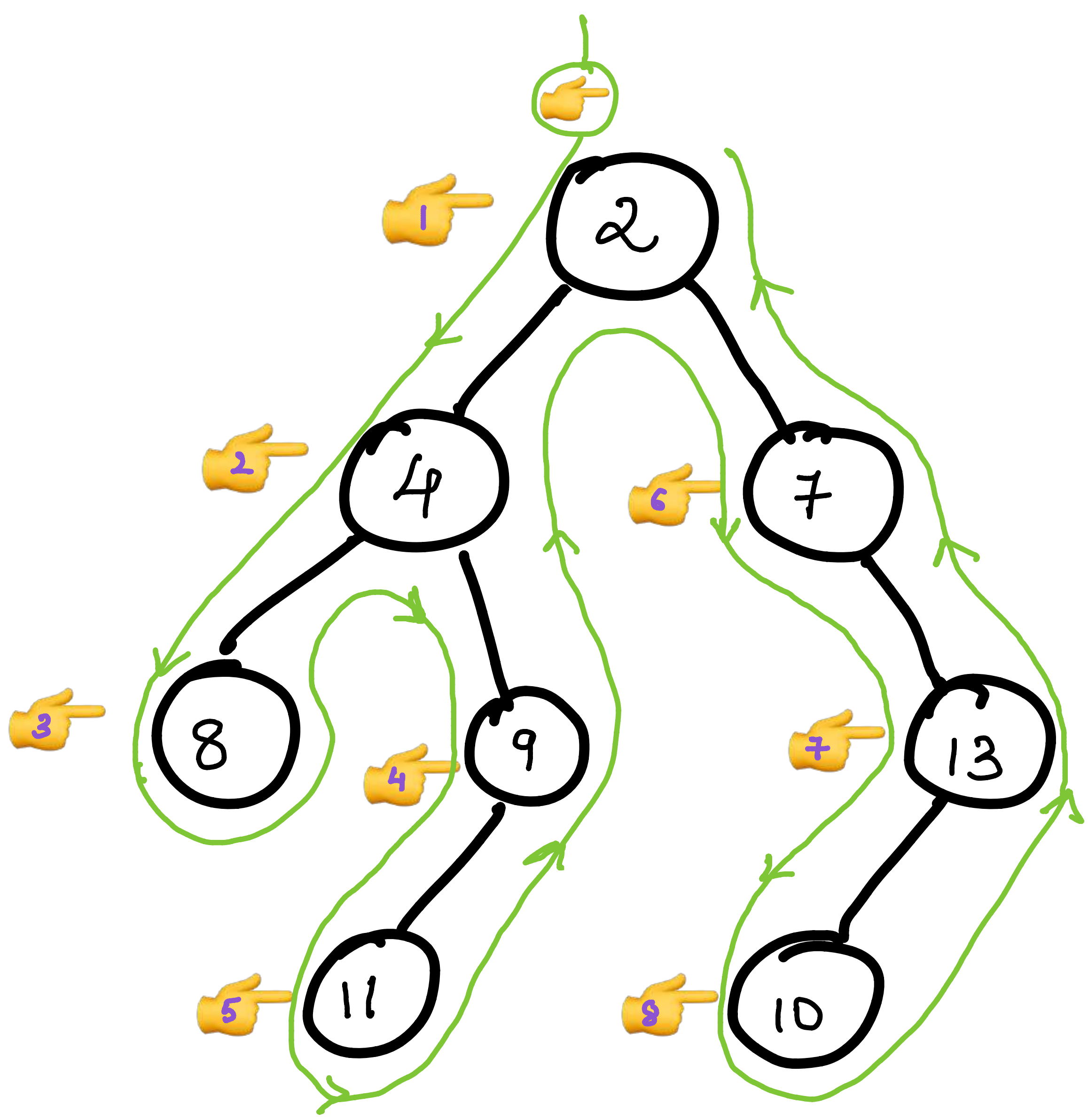
Traversals → DFS → Preorder
Inorder
Postorder

→ BFS → Levelorder

(A) Preorder →

processing order → node
left child
right child

Eg



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

* Order of visiting is the preorder traversal.

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

TC → O(n)

SC → O(n) 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;
    }
};
```

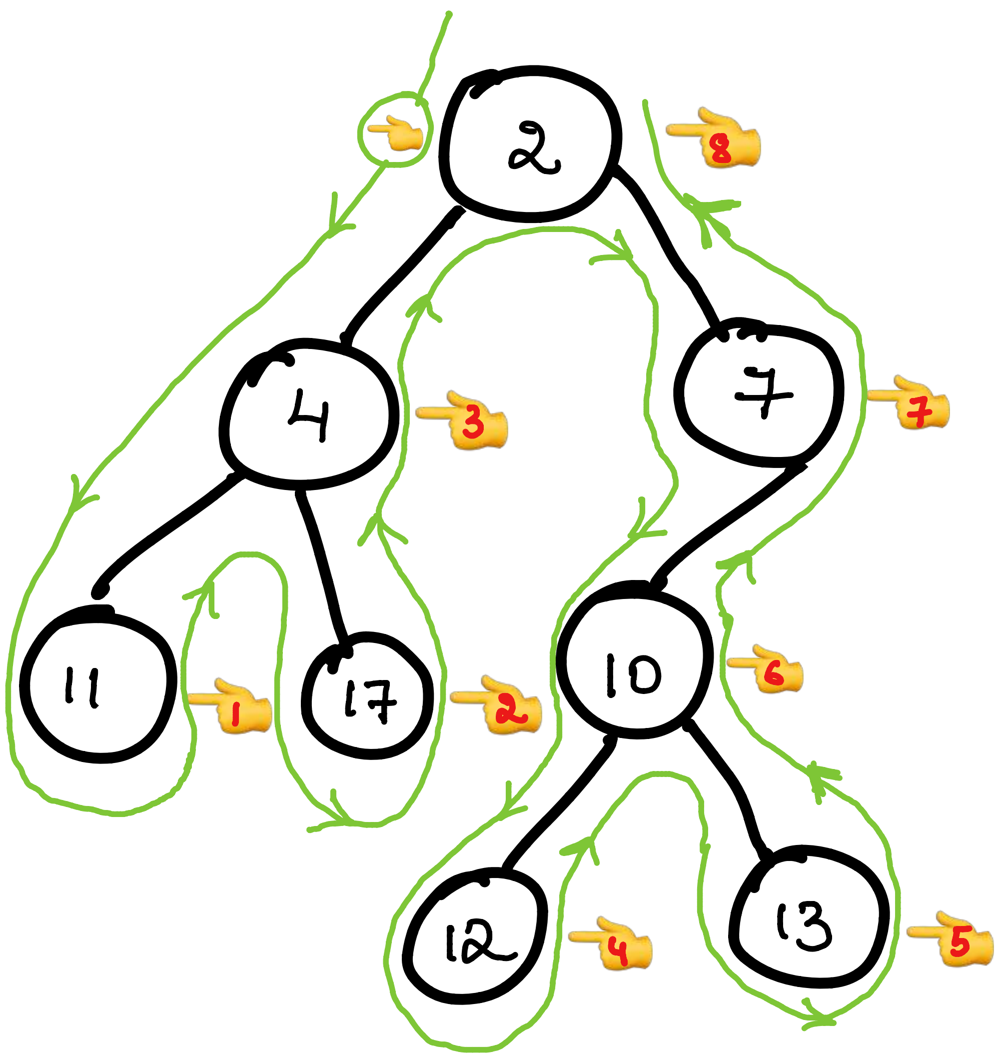
④ Pre-order traversal of n-ary tree

```
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.

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

TC → O(n)

SC → O(n) 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;
    }
};
```

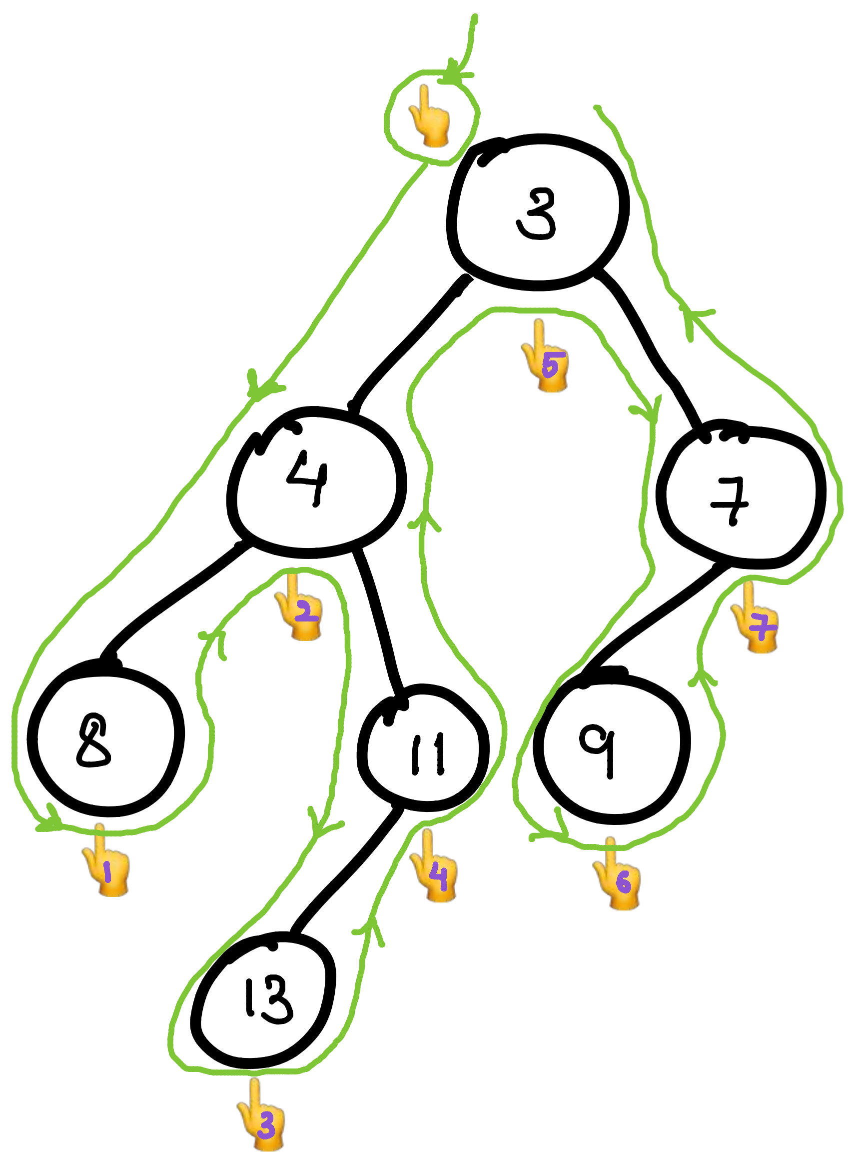
⑥ Postorder traversal of nary tree

```
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;
    }
};
```

ⓒ Inorder →

processing order → left child
node
right child

Eg



* Point finger 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;
    }
};
```

Inorder 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.

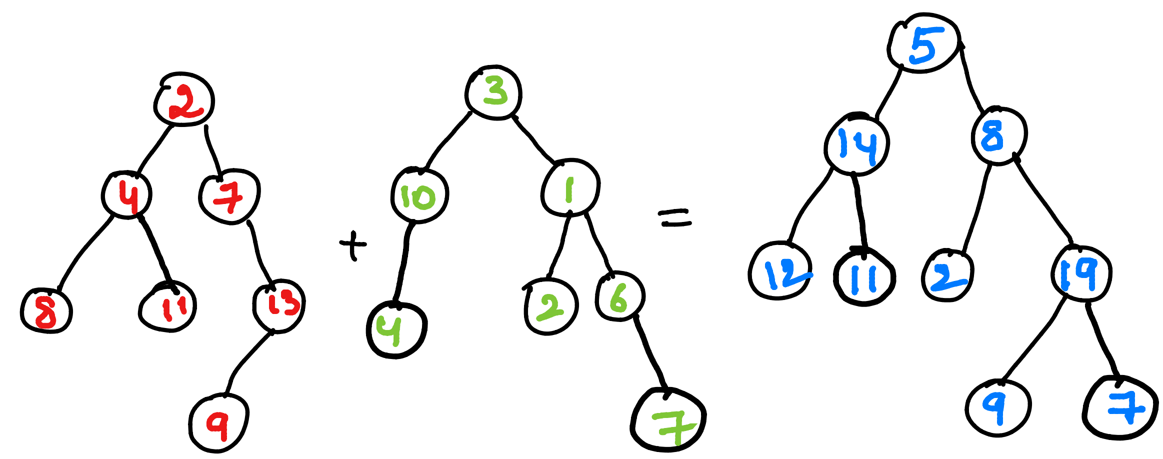
- Recursively visit the first child.
- Recursively visit the second child.
- …..
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```
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 →

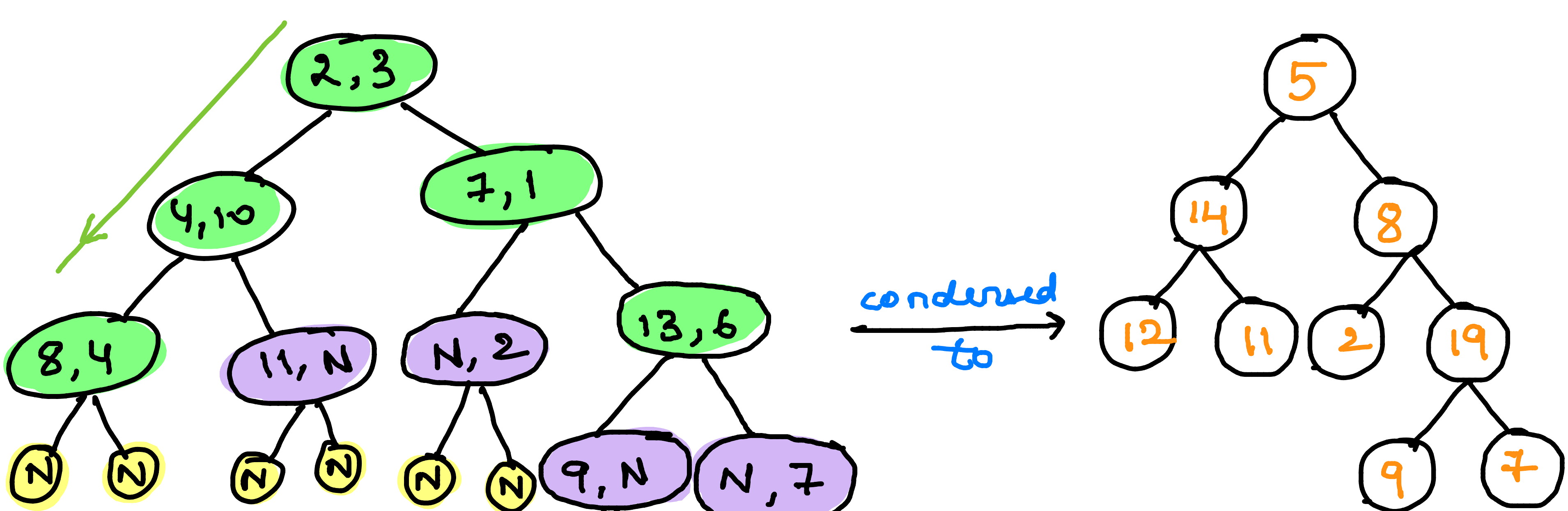
Given root nodes of 2 binary trees, return root of the sum tree

Eg



we will perform preorder traversal on the binary tree because the node/root need 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))

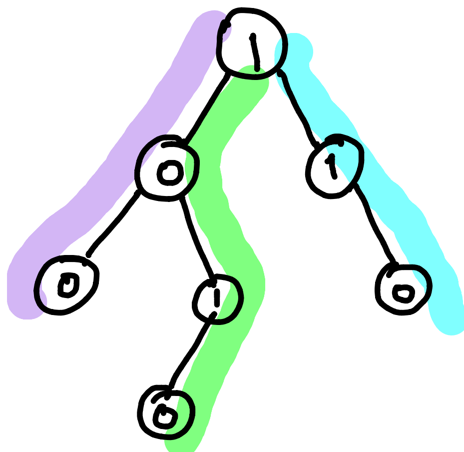
Recursive stack → O(max(h1, h2))

code →

```cpp
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);
    }
};
```

(9) Sum of root to leaf paths →

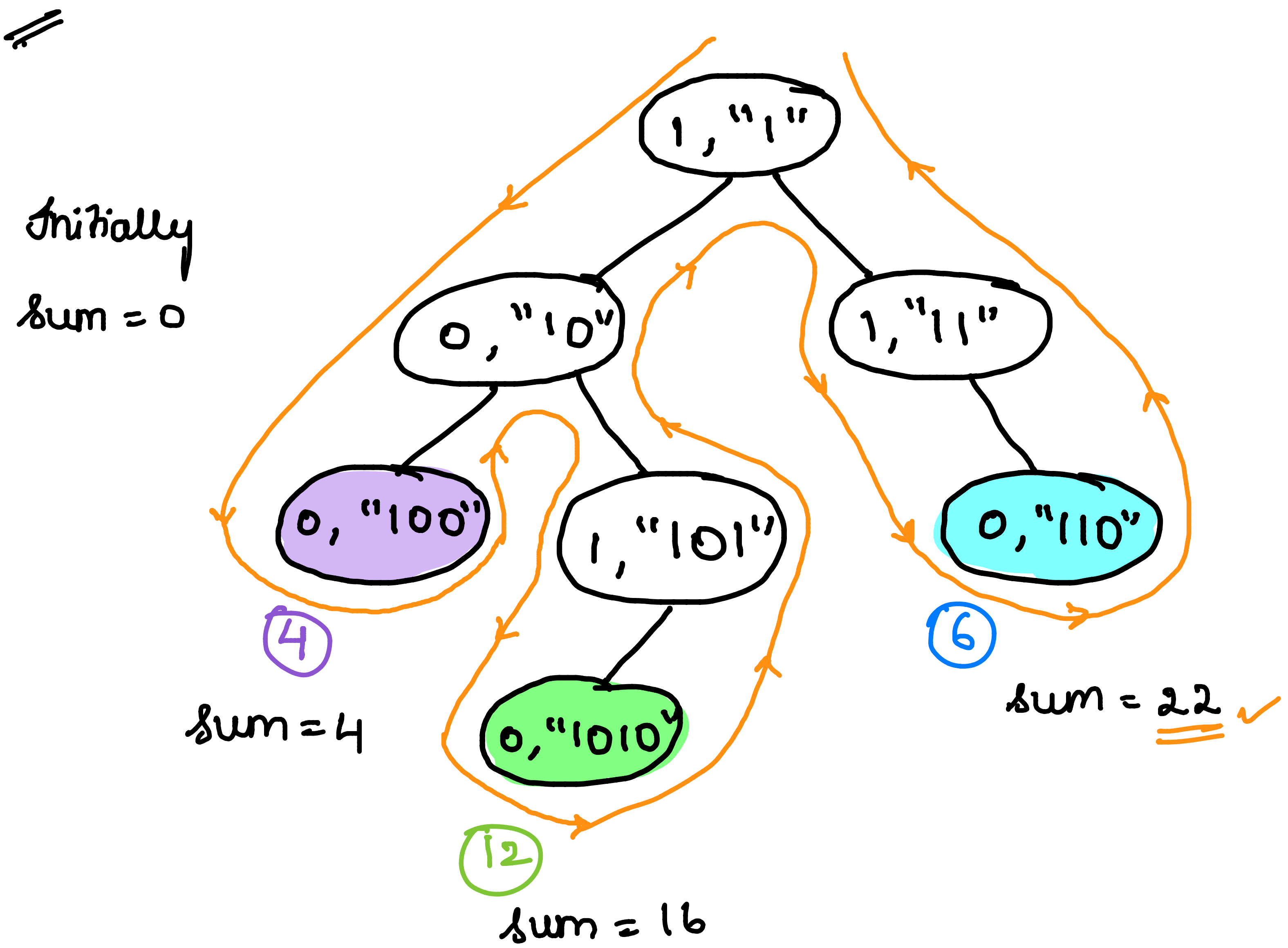
Eg



→ $(100)_2 + (1010)_2 + (110)_2$

4 + 12 + 6

= 22.



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

Time → O(n)

Space → O(n)

Recursive stack → O(h)

Code

```cpp
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];
    }
};
```

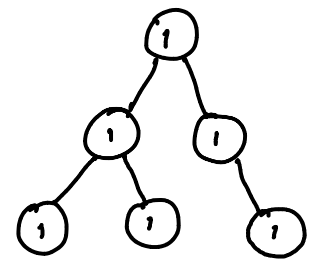
Note →

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

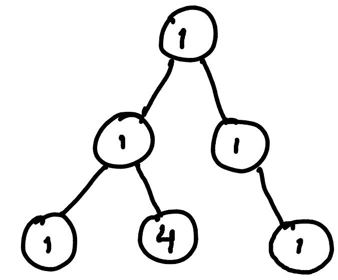
[to convert from binary to decimal we give it as 2]

(10) Univalued Binary Tree →

Eg

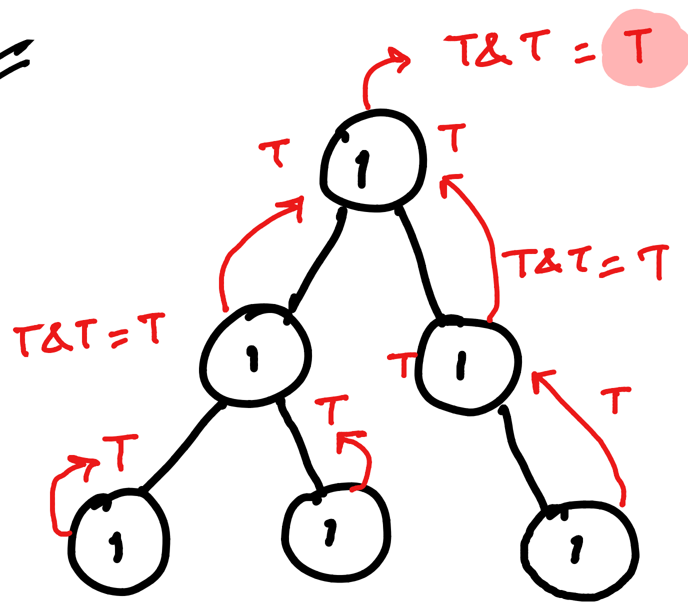


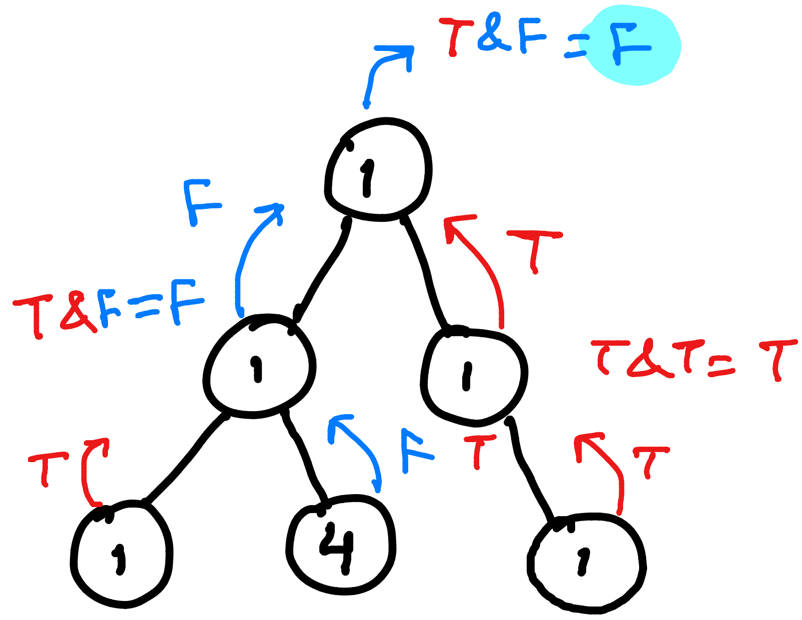
↳ returns true



↳ returns false

//





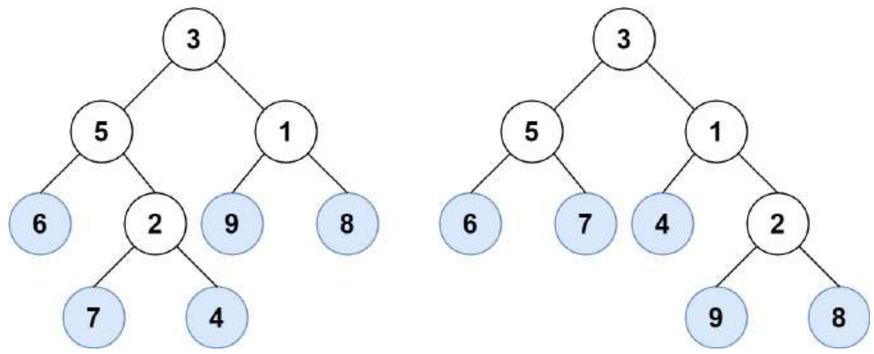
Code

```
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



$v_1 = 6, 7, 4, 9, 8$ $\Rightarrow$ $v_1 == v_2$

$v_2 = 6, 7, 4, 9, 8$ ↳ return true else false.

code →

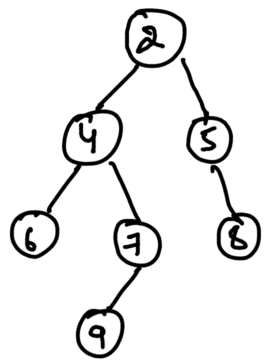
```
class Solution {
public:
    void traversal(TreeNode* root, vector<int>&v){
        if(root==NULL)
            return;

        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;
    }
};
```

D5

(12) Binary tree paths

↳ given root print all the paths from root to leaf

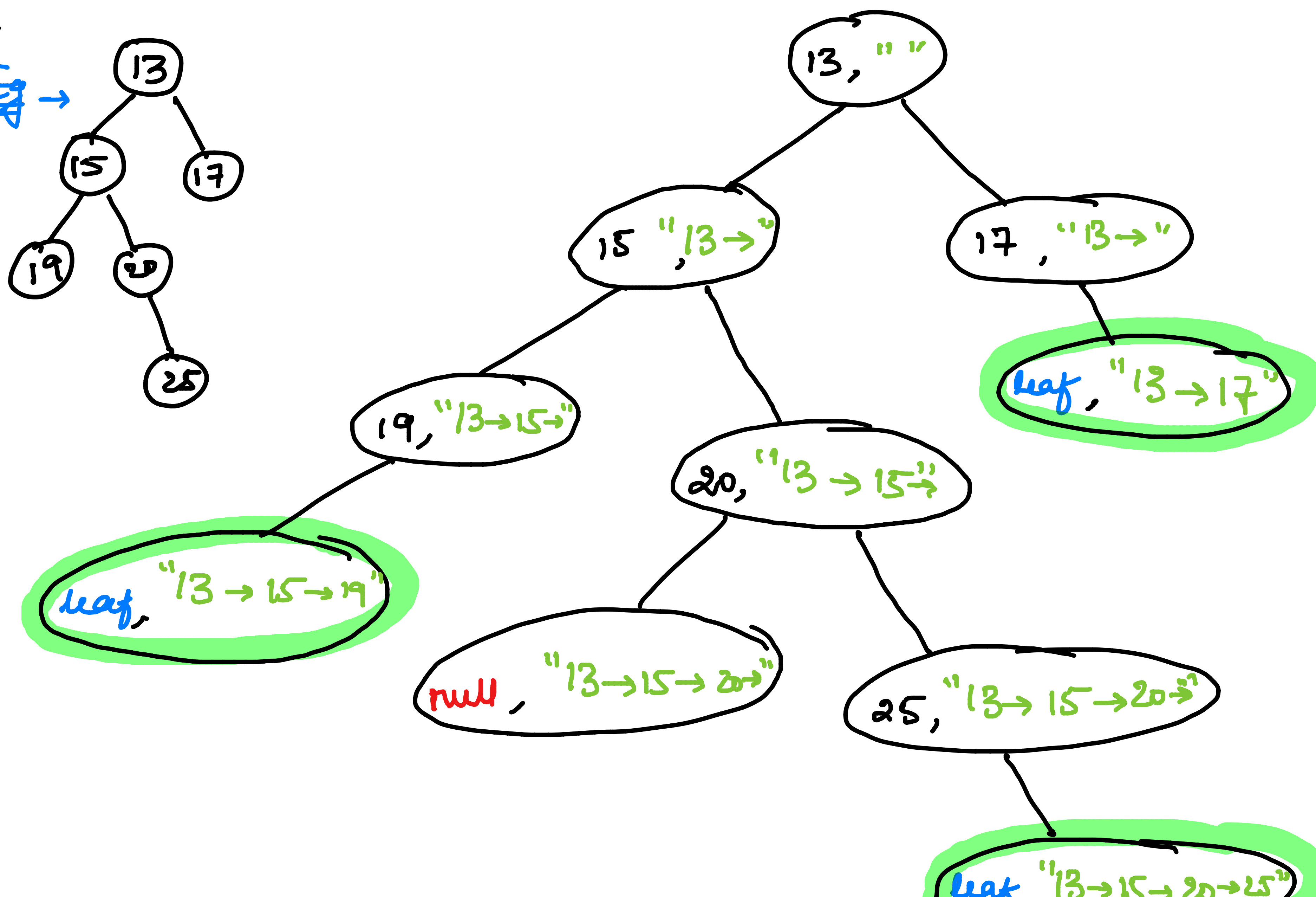
Eg



⇒ ["2→4→6", "2→4→7→9", "2→5→8"]

//

Eg →



Result =

["13→15→19", "13→15→20→25", "13→17"]

Time complexity = O(n)

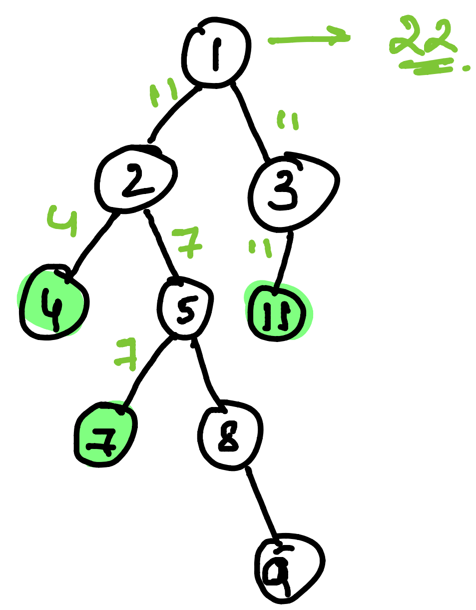
Space complexity = $\underbrace{O(x)}_{\text{↳ Answer array}}$ + O(h) → recursive stack.

Code →

```cpp
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;
    }
};
```

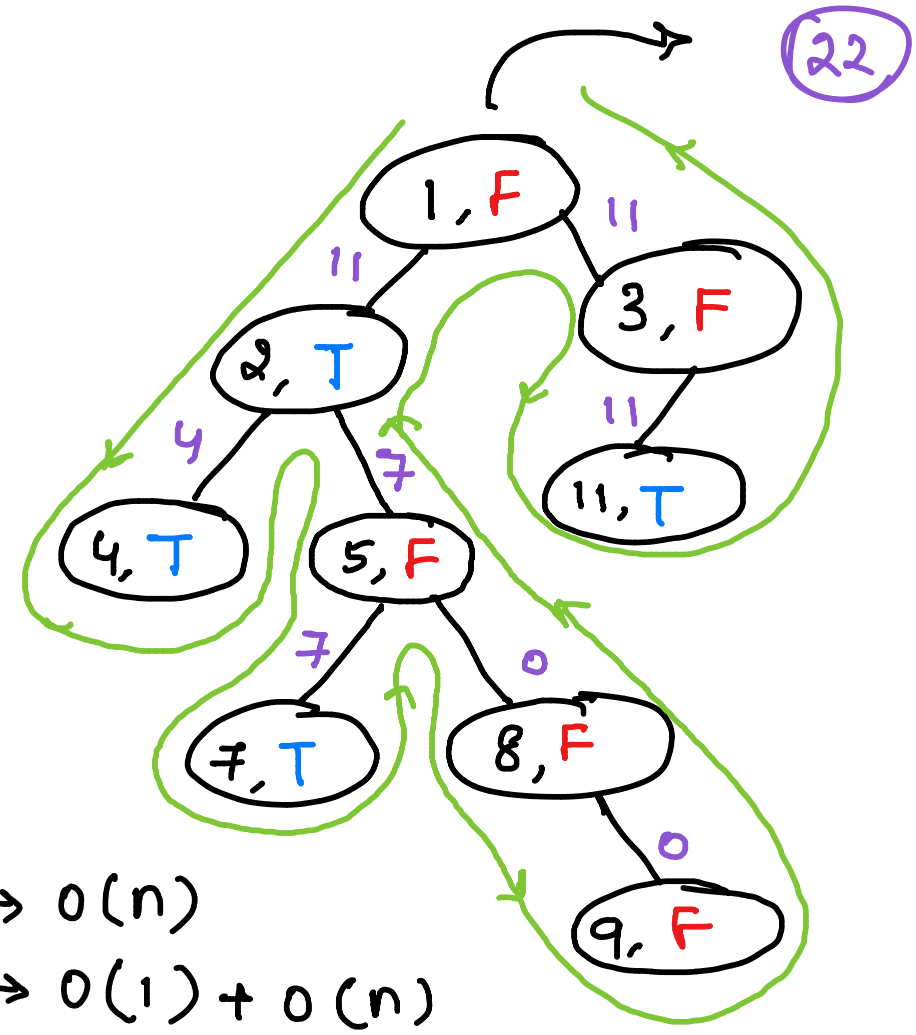
13 Sum of left leaves →

Eg



Result = 4+7+11

= 22.



Tc → O(n)

Sc → O(1) + O(n)

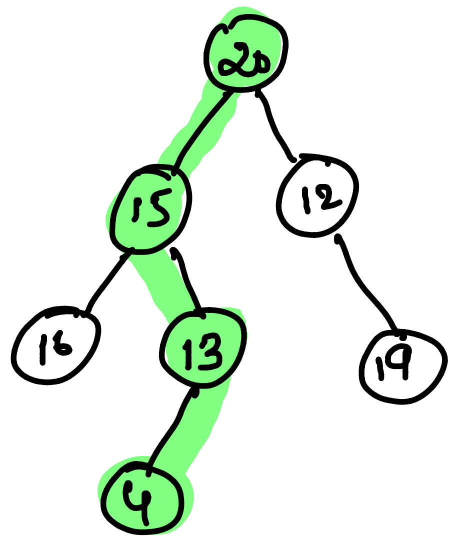
→ stack

Code →

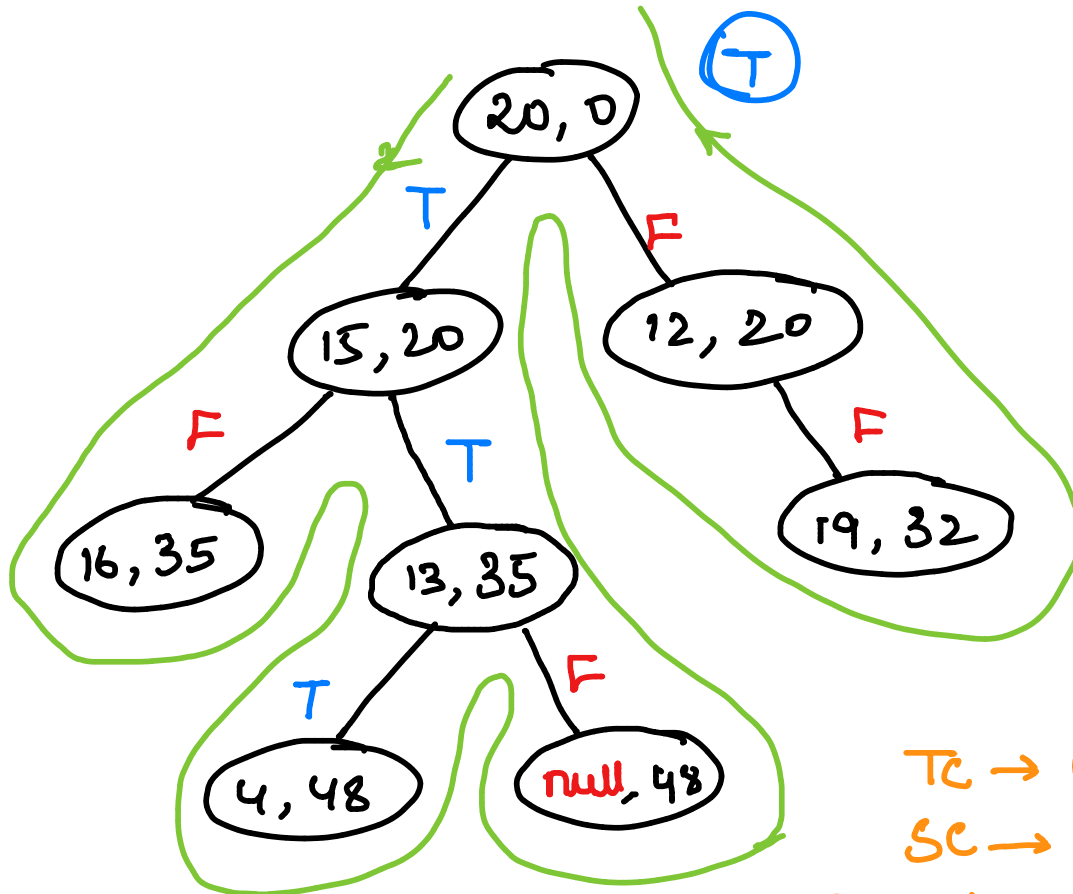
```
class Solution {
public:
    int leftLeafSum(TreeNode *root, bool leaf){
        if(root==NULL){
            return 0;
        }
        if(root->left==NULL && root->right==NULL && leaf){
            return root->val;
        }
        int ls = leftLeafSum(root->left, true);
        int rs = leftLeafSum(root->right,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

Eg



target = 52



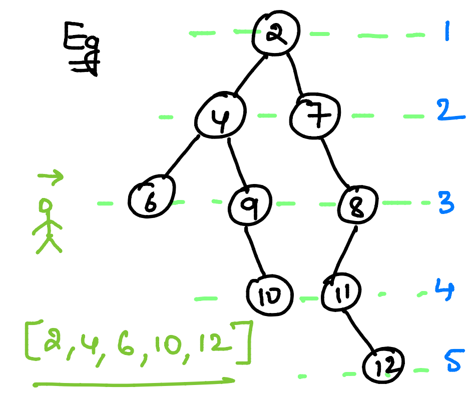
TC → O(n)
SC → O(1)
Recursive Stack → O(h)

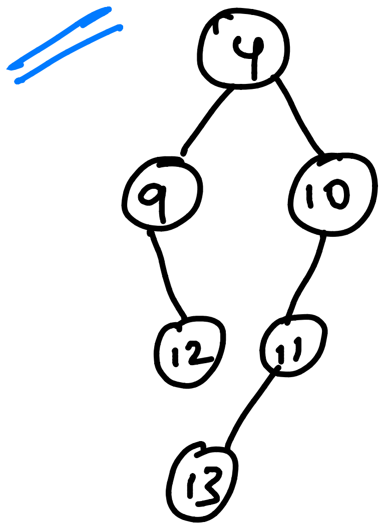
code

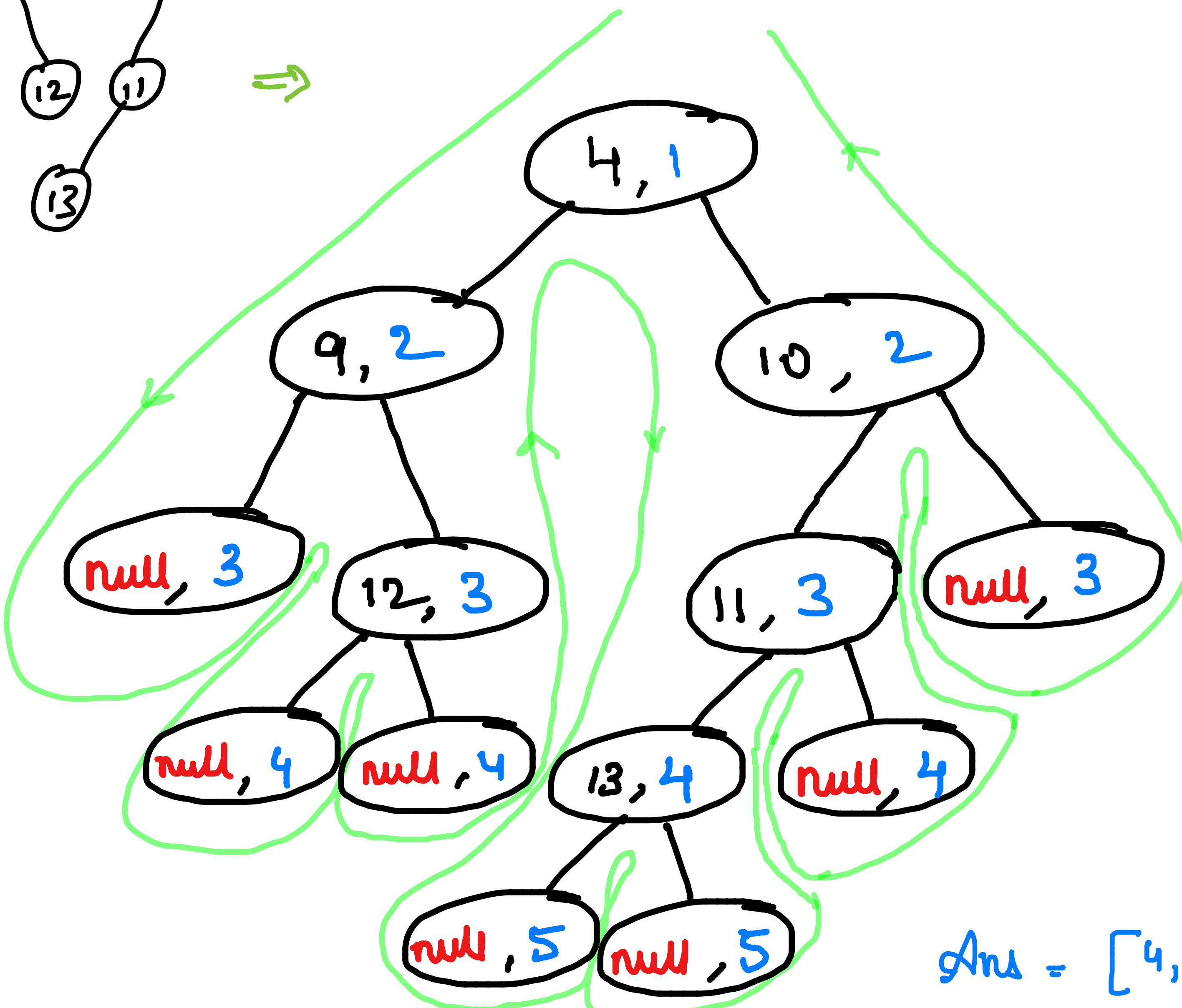
```
class Solution {
public:
    bool pathSumUtil(TreeNode* root, int currSum, int targetSum){
        if(root==NULL)
            return false;

        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);
    }
};
```

D6 (15) Left view of a Binary Tree

Eg







Ans = [4, 9, 12, 13]

→ 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 & into the set

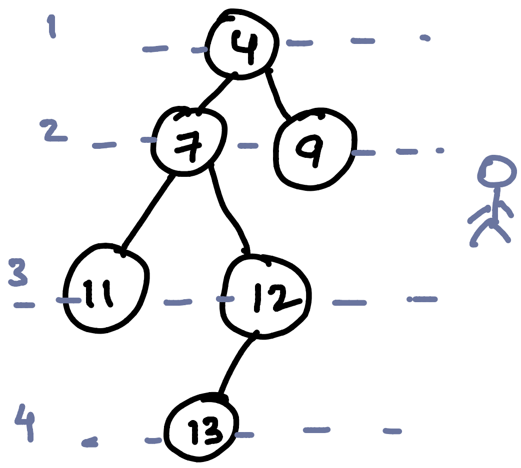
Tc → O(n)

Sc → $\underbrace{O(n)}_{\text{result}} + O(n) + O(h)$

Code →

```cpp
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

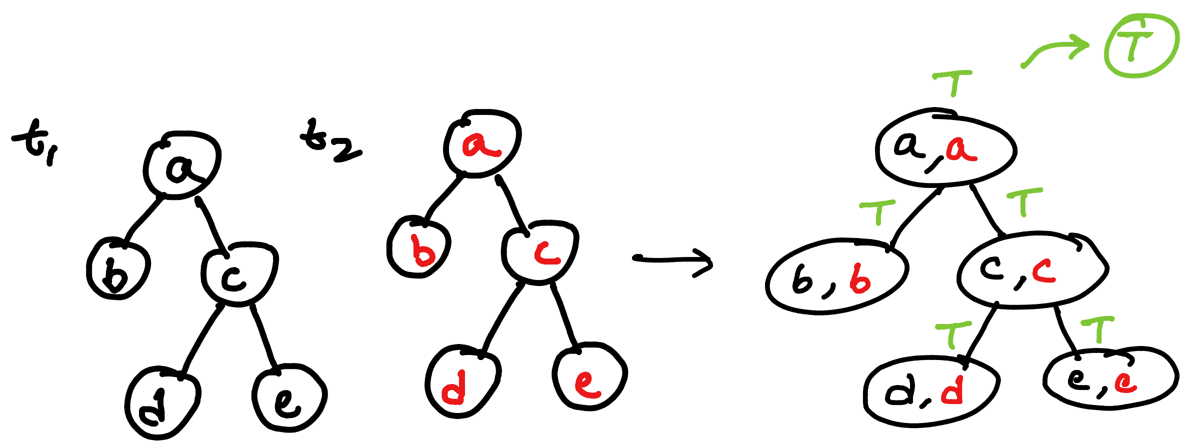
```
class Solution {
public:
    void viewGenerator (TreeNode* root, vector<int> &res, set<int> &s, int currLevel){
        if(root==NULL) return;
        // if level is not reached, then add to result and the set
        if(s.find(currLevel)==s.end()){
            s.insert(currLevel);
            res.push_back(root->val);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

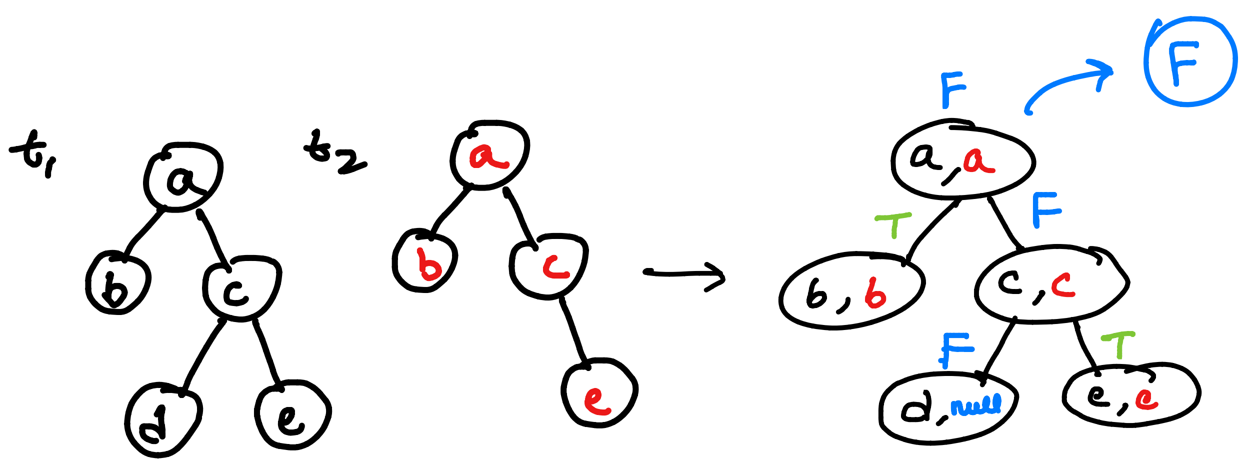
Tc → O(n)

Sc → $\underbrace{O(n)}_{\text{result}} + O(n) + O(h)$

17 Same tree →

return true if both trees are same else false





TC → $O(\min(m,n))$

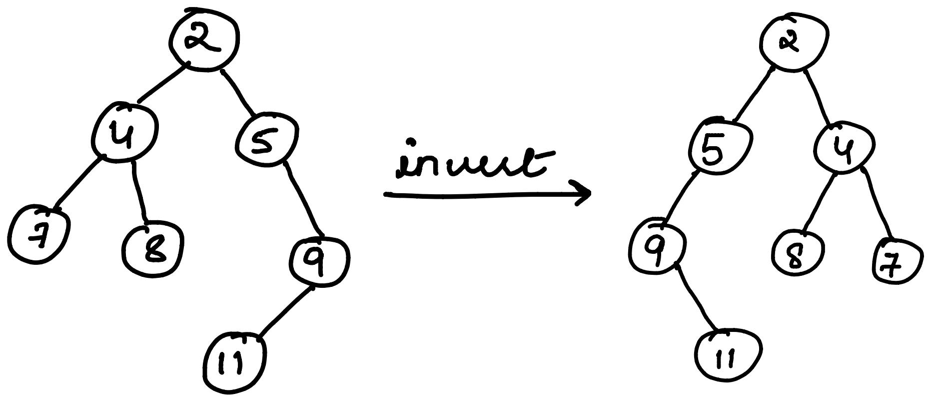
SC → $O(1) + O(\min(h1,h2))$

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);

    }
};
```

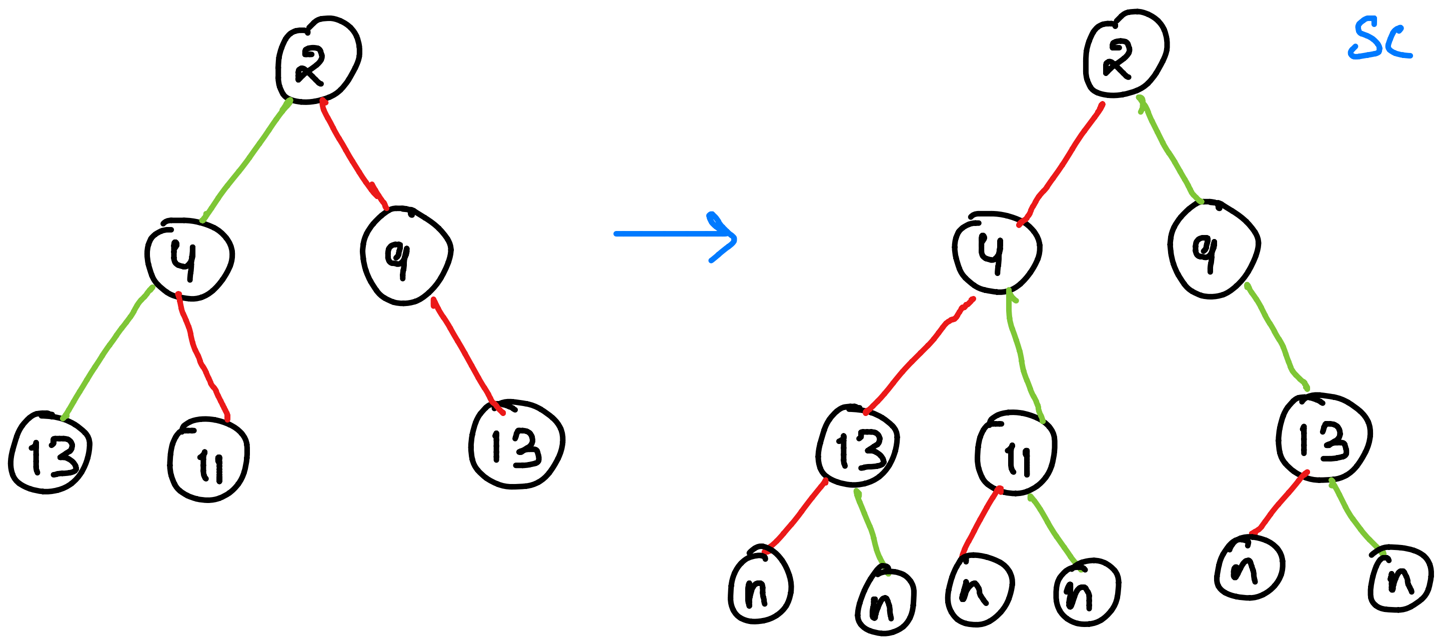
(18) Invert Binary Tree → given the root of BT, find its mirror img.

Eg



TC → $O(n)$

SC → $O(n) + O(h)$



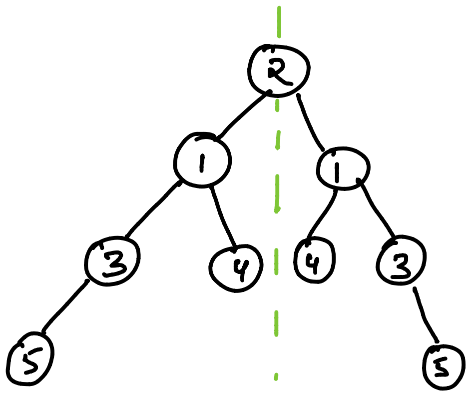
Code →

```
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;
    }
};
```

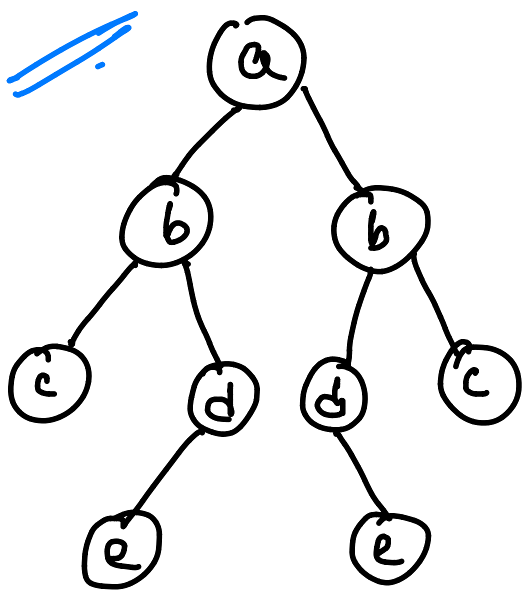
D7

19 Symmetric Tree →

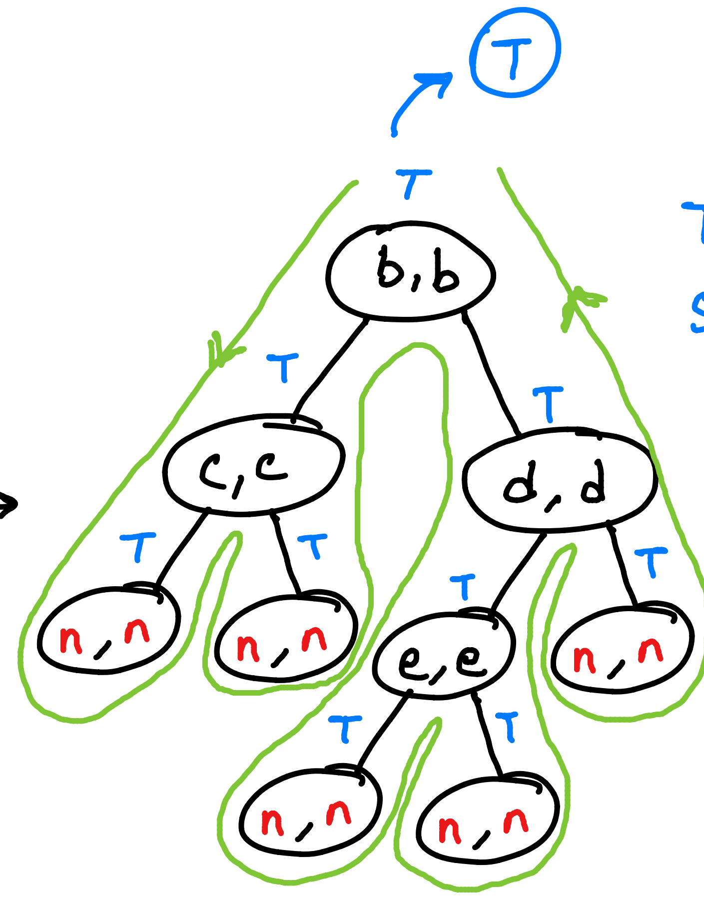
return true if left subtree is equal to right subtree, else return false



→ returns true



→



Tc → O(n)

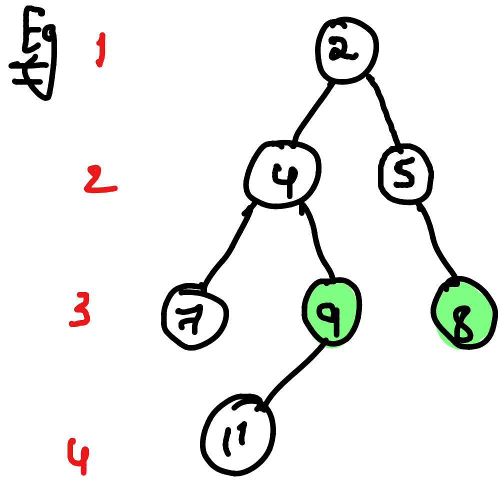
Sc → O(1) + O(h)

code →

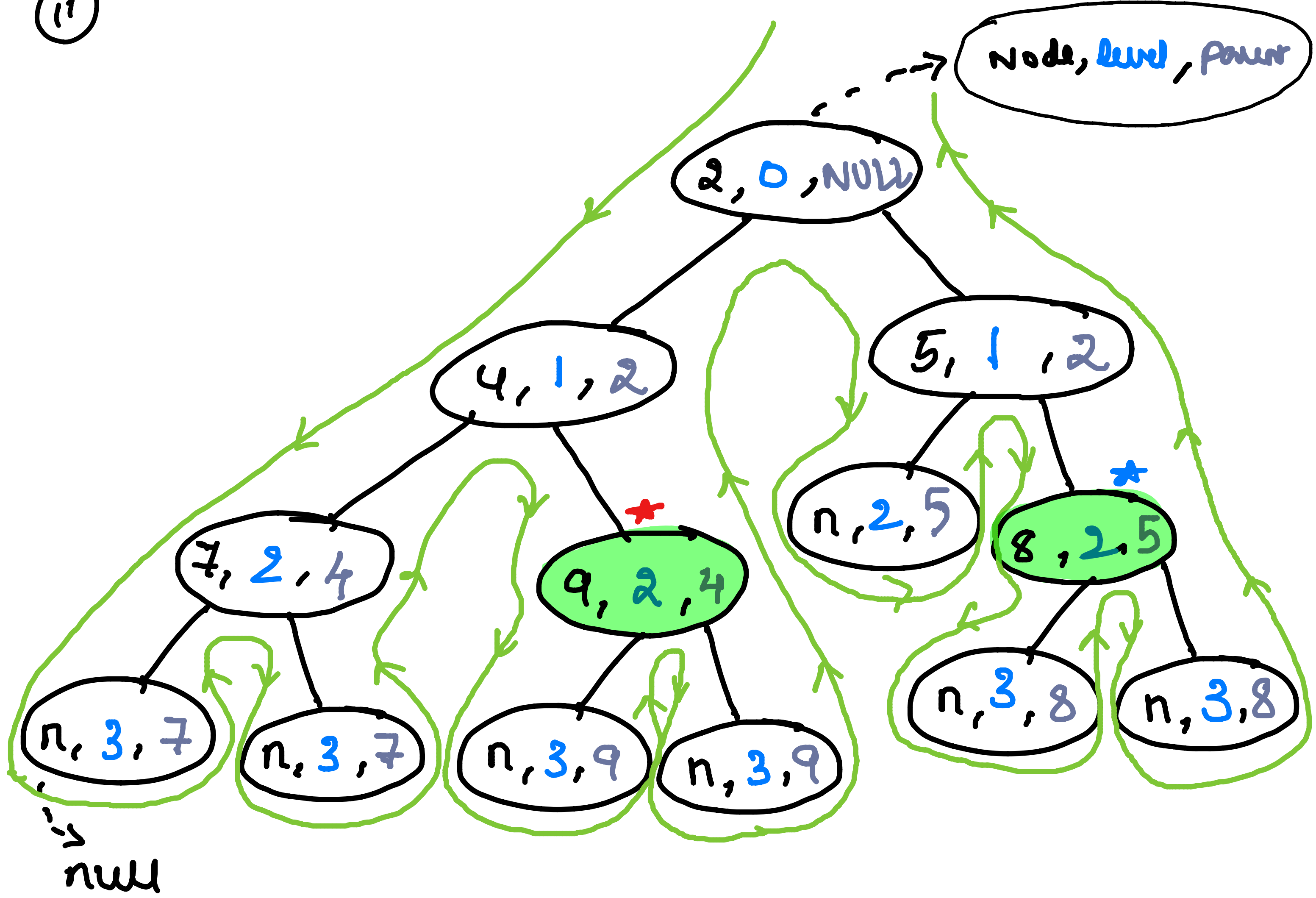
```cpp
class Solution {
public:
    bool isMirror(TreeNode* l, TreeNode* r){

        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.

same level but diff parent.



X = 9
Y = 8 } cousins



* at this step as value = 9 i.e
x is found store it's parent & level in separate variables

* later compare its value with other occurrence i.e Y
such that

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

Tc → O(n)
Sc → O(1)
Recursive Stack → 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;
    }
};
```