③ Pre-order traversal of Binary tree

```cpp
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;
    }
};
```

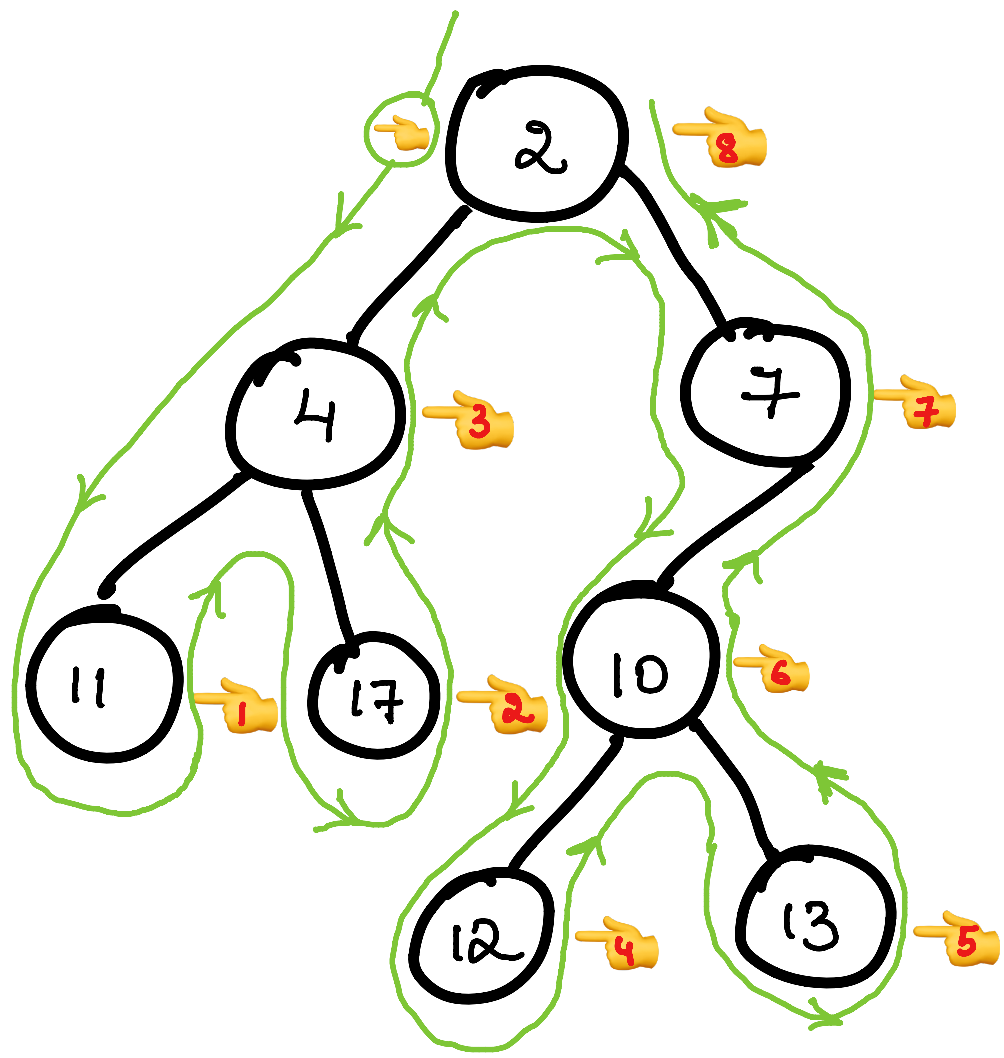
④ Pre-order traversal of n-ary tree

```cpp
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;
    }
};
```

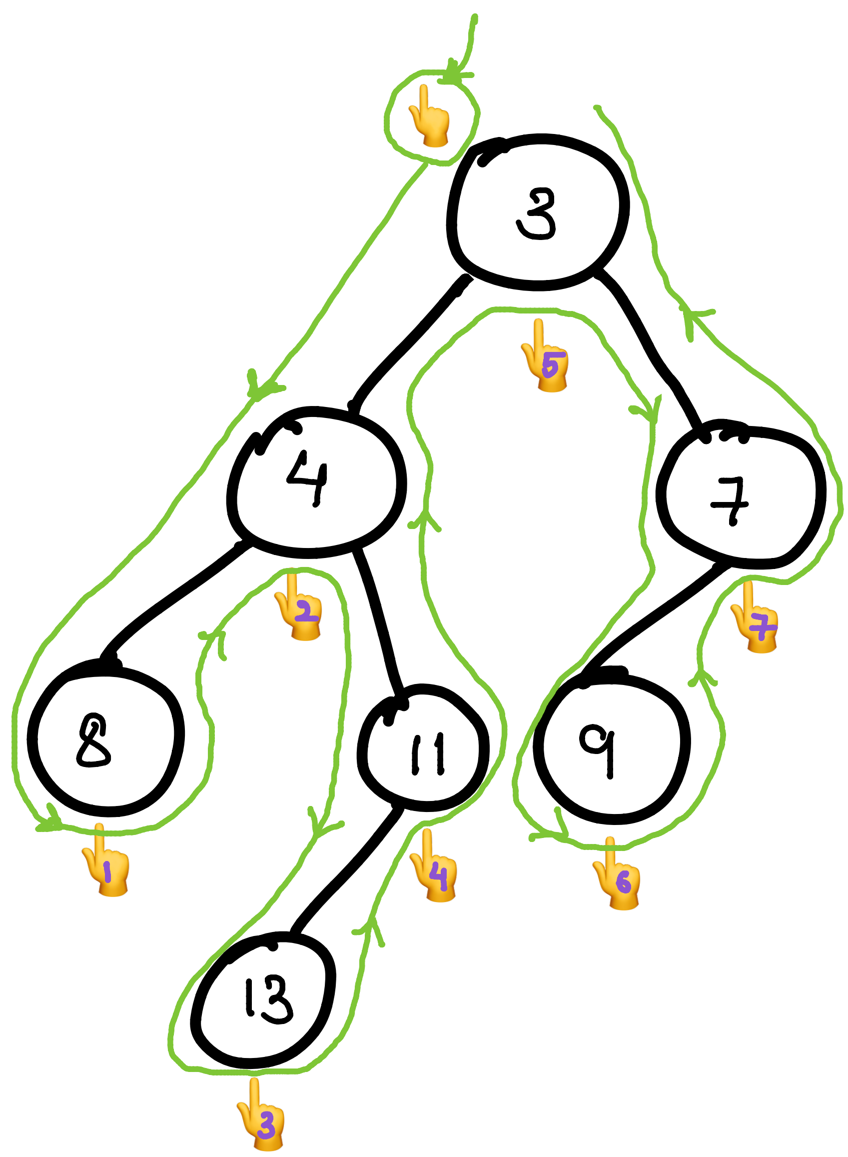
⑥ Postorder traversal of nary tree

```
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;
    }
};
```

(c) 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.

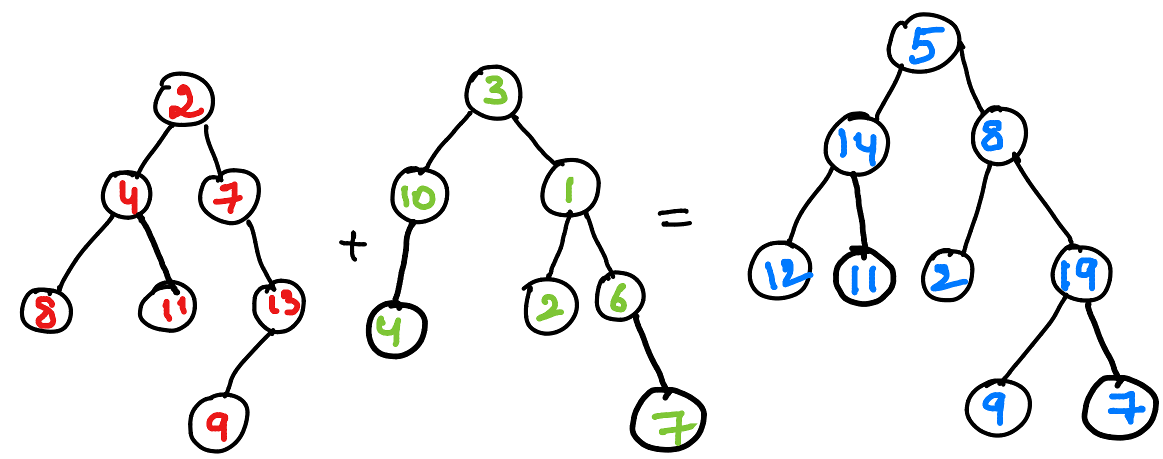
- Recursively visit the first child.
- Recursively visit the second child.
- .....
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```
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 →

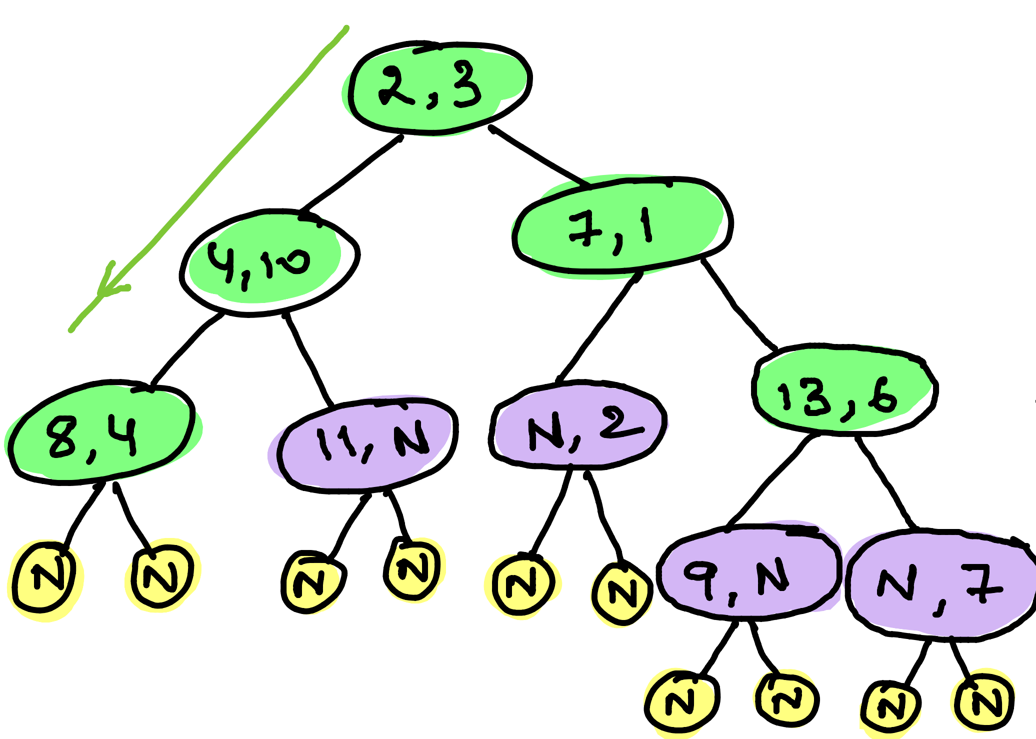
Given root nodes of 2 binary trees, return root of the sum tree

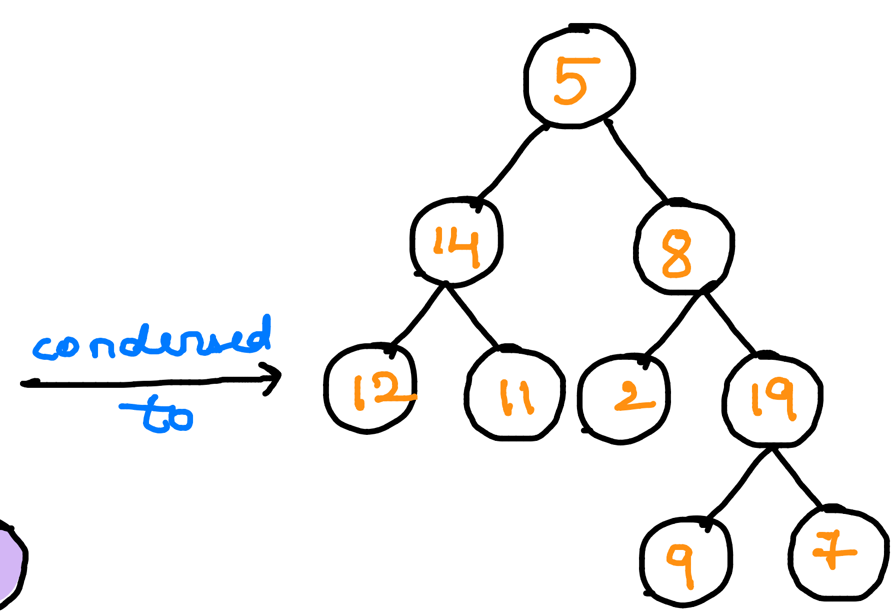
Eg



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))

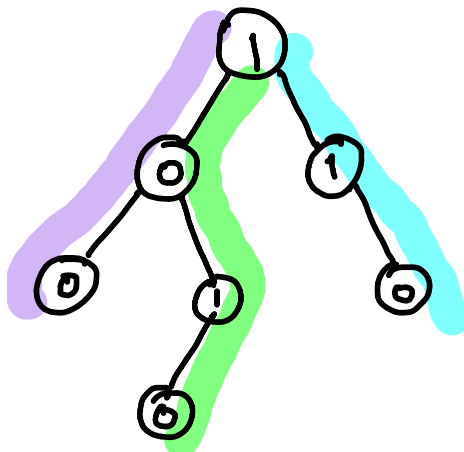
Recursive stack → O(max(h_1,h_2))

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);
    }
};
```

9) Sum of root to leaf paths →

Eg



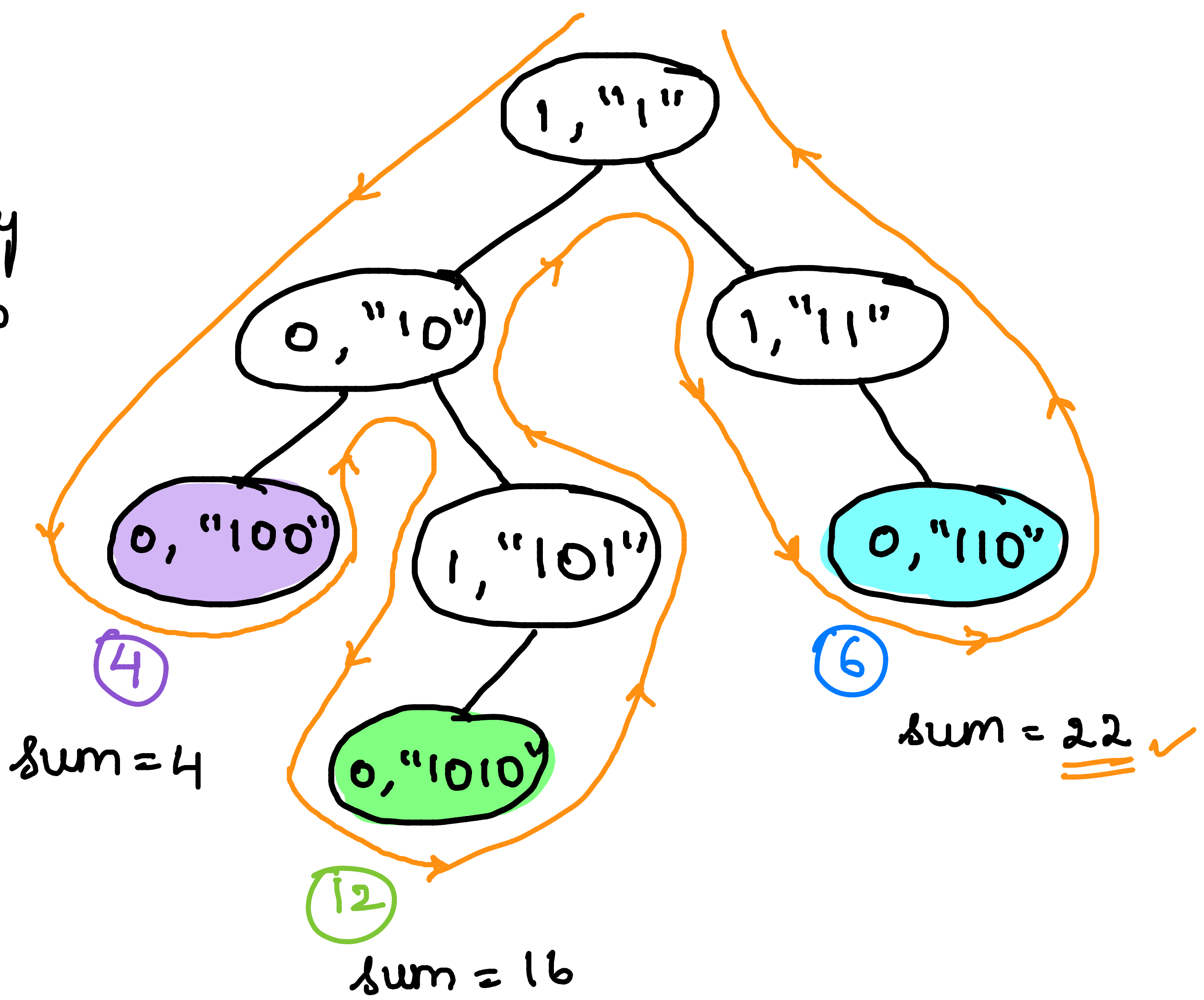
→ $(100)_2 + (1010)_2 + (110)_2$

$4 + 12 + 6$

$= 22$

Initially

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];
    }
};
```

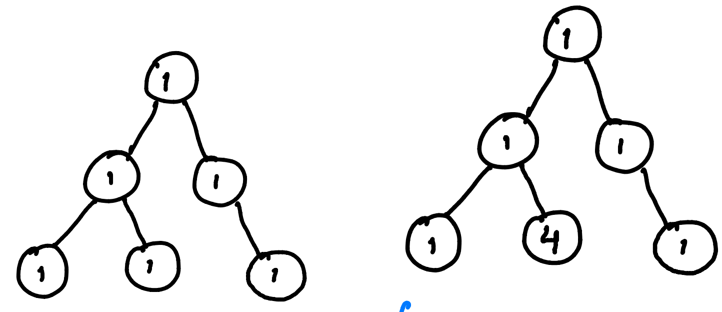
Note →

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

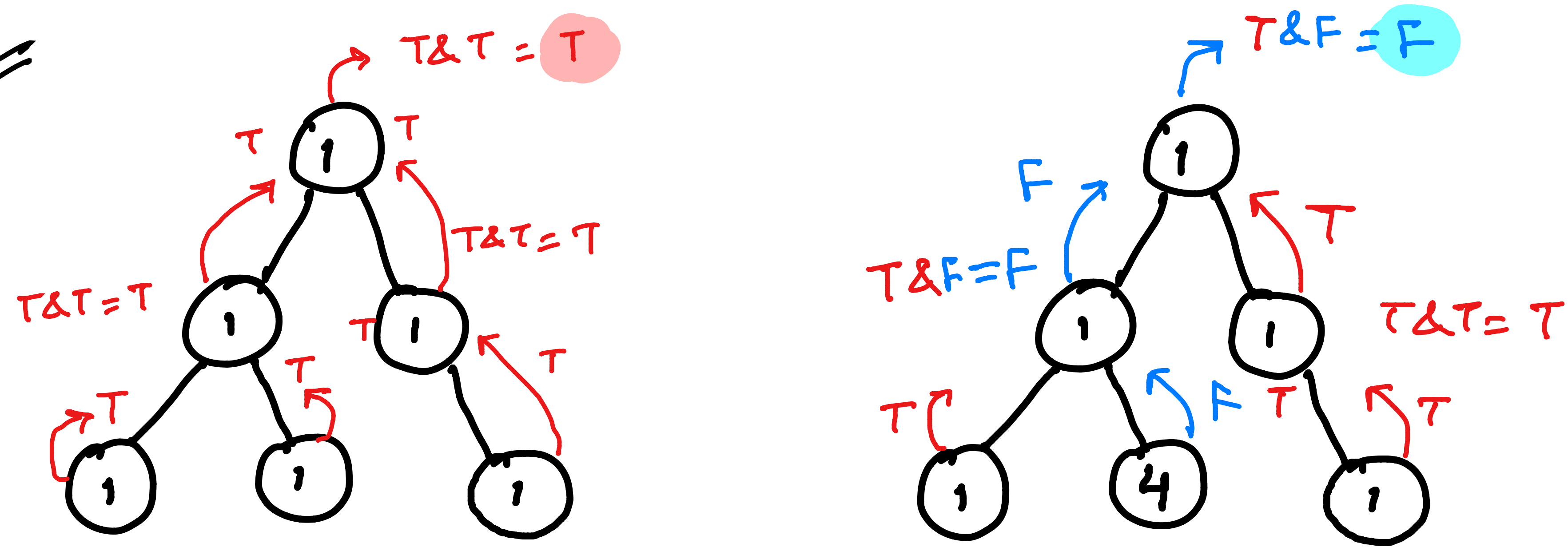
[to convert from binary to decimal we give it as 2]

10 Univalued Binary Tree →

Eg



↳ returns true ↳ returns false



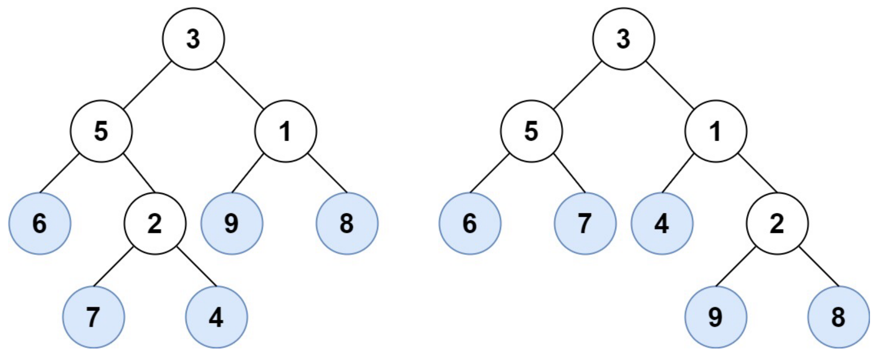
Code

```cpp
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 ⇒ $v_1 == v_2$

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

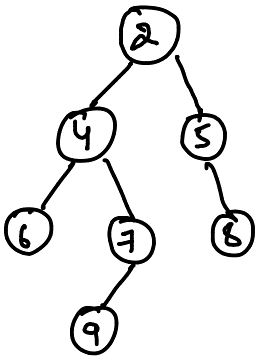
code →

```cpp
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

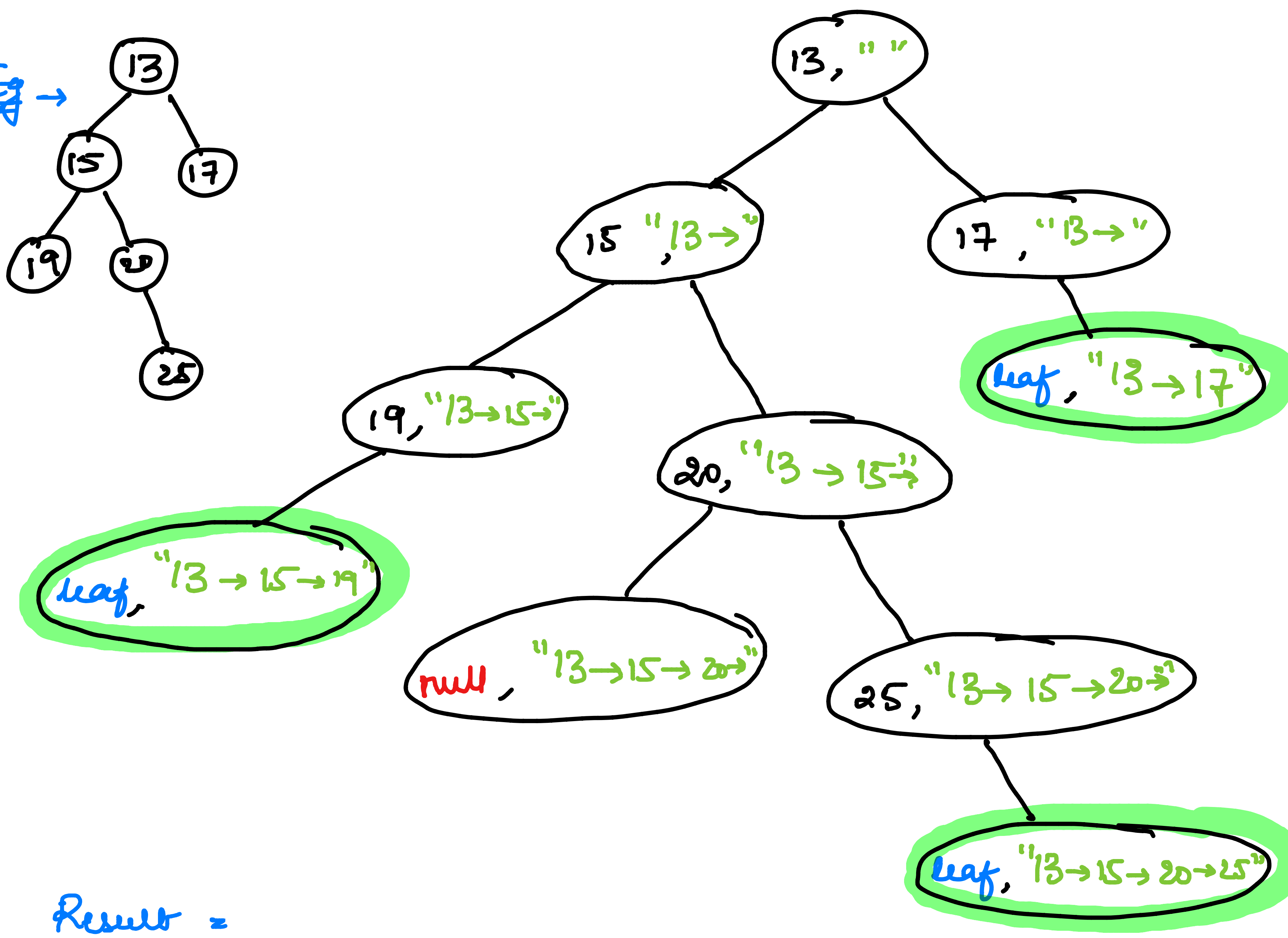
↳ given root print all the paths from root to leaf

Eg



⇒ ["2→4→6", "2→4→7→9", "2→5→8"]

Eg →



Result =

["13→15→19", "13→15→20→25", "13→17"]

Time complexity = $O(n)$

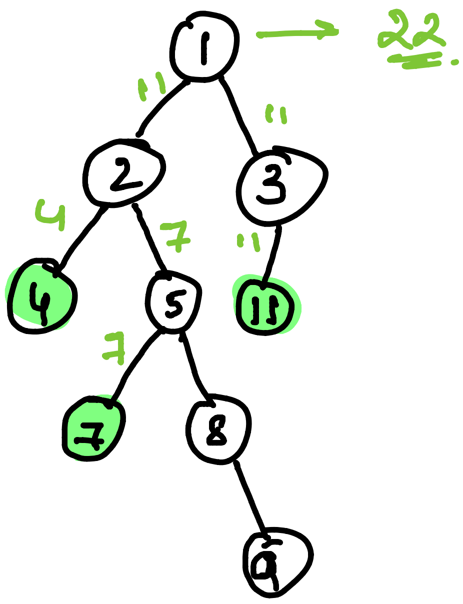
Space complexity = $\underbrace{O(x)}_{\text{Answer array}} + O(h)$ → recursive stack.

Code →

```
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;
    }
};
```

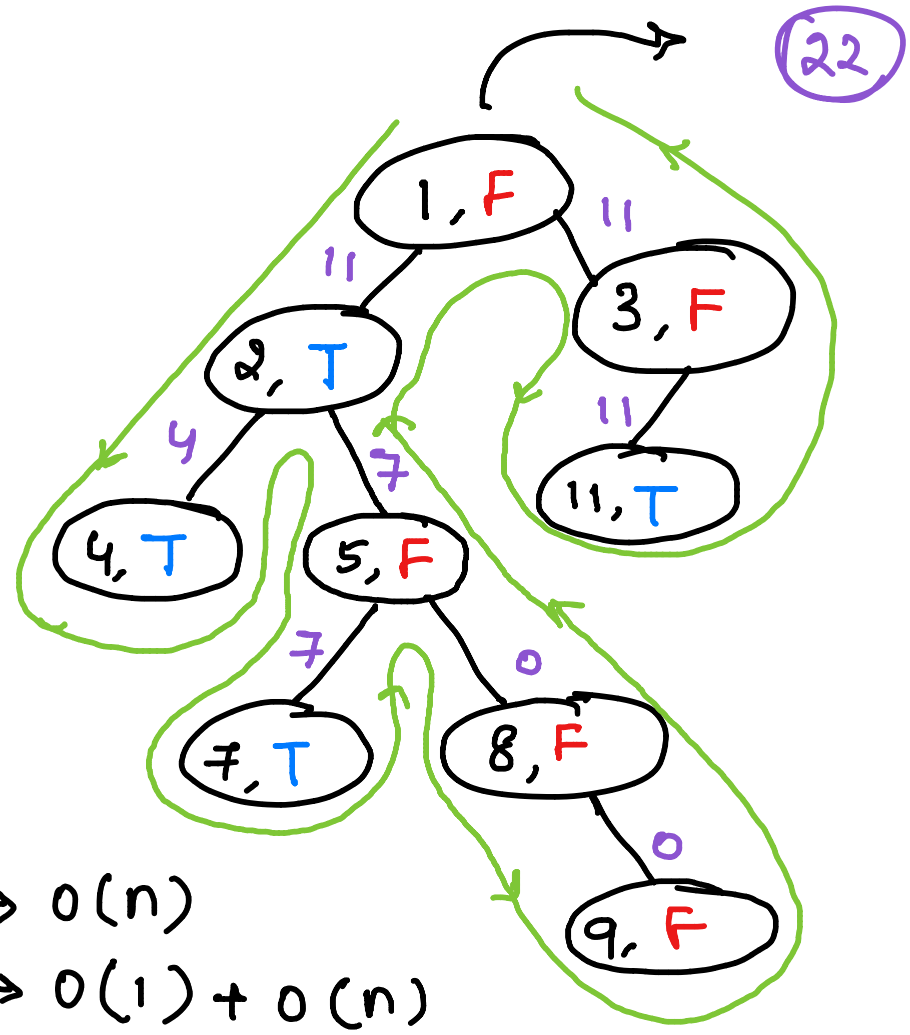
(13) Sum of left leaves →

Eg



Result = 4+7+11

= 22



Tc → O(n)

Sc → O(1) + O(n)

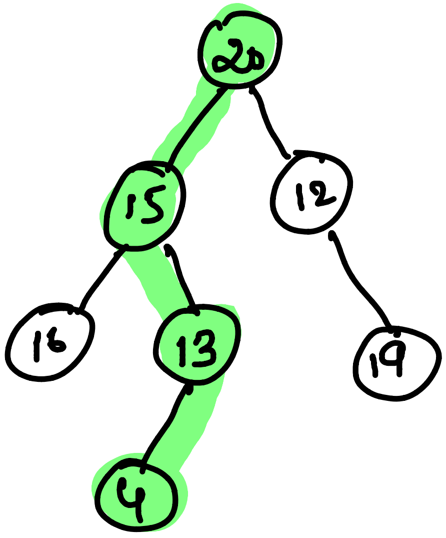
→ stack

code →

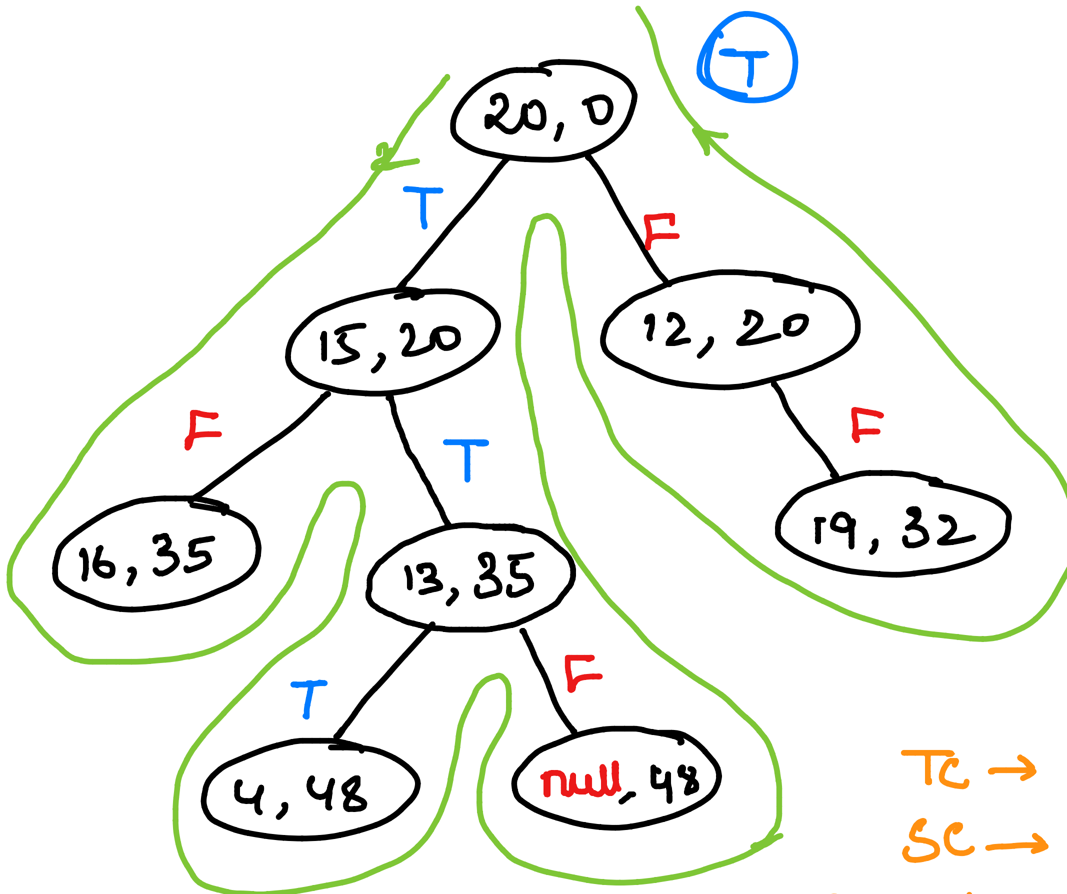
```
class Solution {
public:
    int leftLeafSum(TreeNode *root, bool leaf){
        if(root==NULL){
            return 0;
        }
        if(root->left==NULL && root->right==NULL && leaf){
            return root->val;
        }
        int ls = leftLeafSum(root->left, true);
        int rs = leftLeafSum(root->right,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

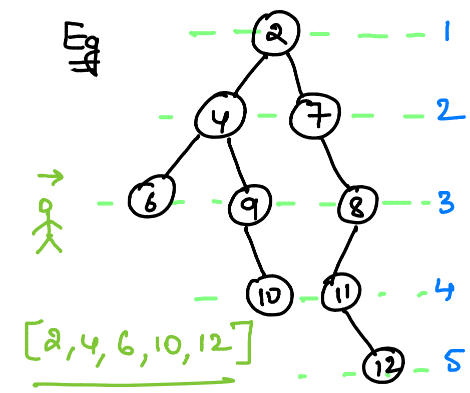
Recursive Stack → O(h)

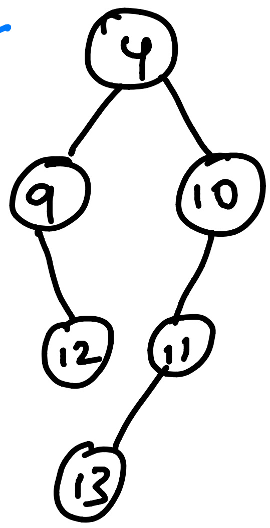
code

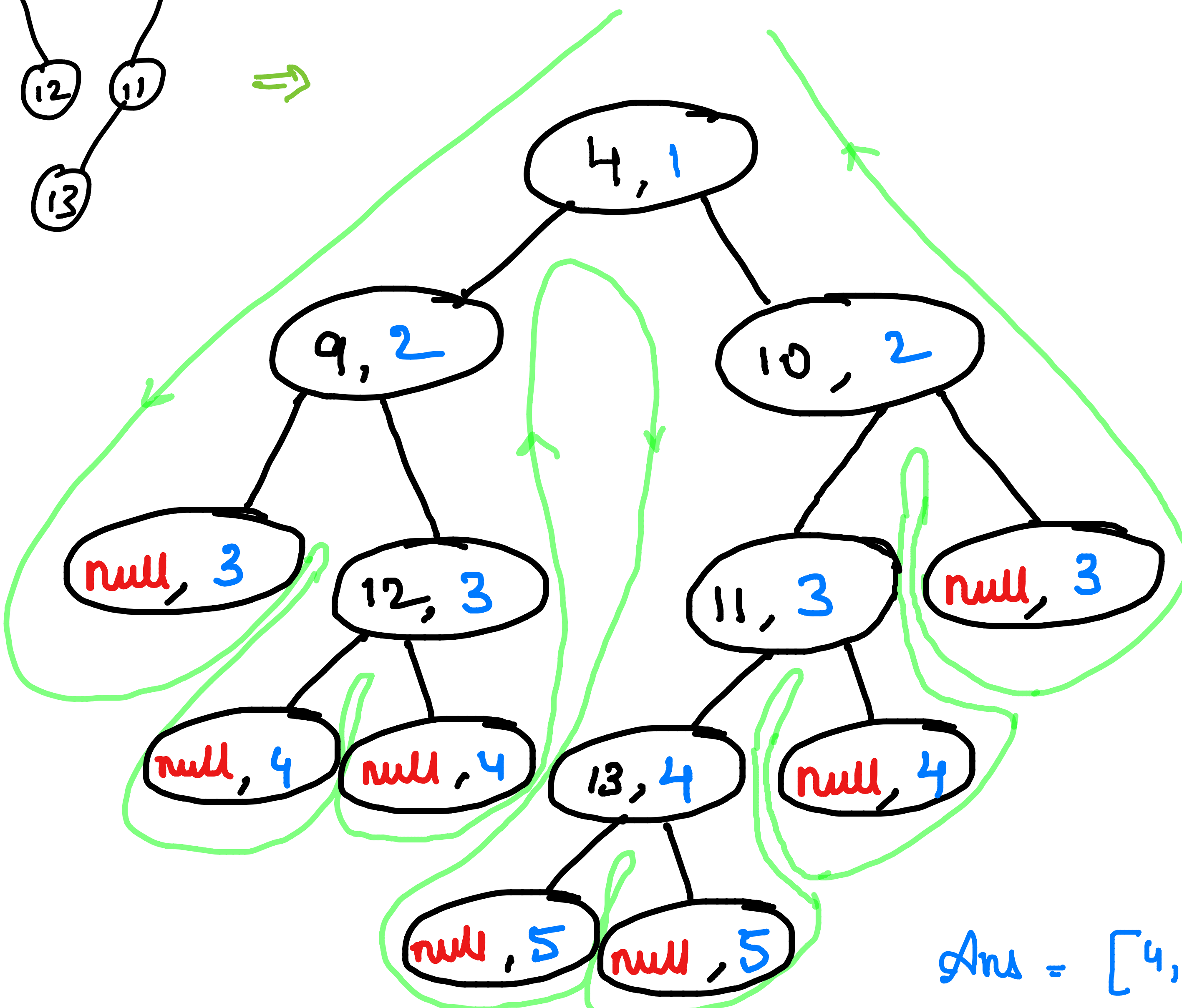
```
class Solution {
public:
    bool pathSumUtil(TreeNode* root, int currSum, int targetSum){
        if(root==NULL)
            return false;

        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

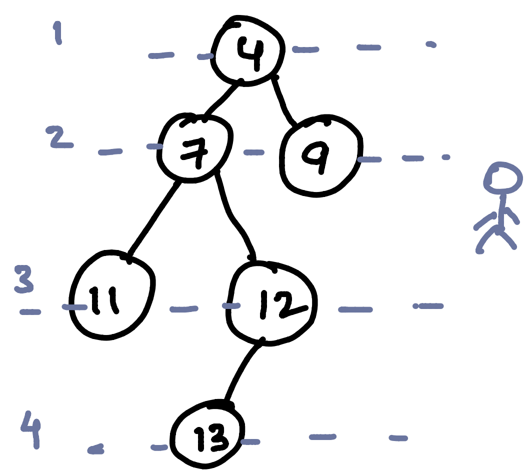
Tc → O(n)

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

Code →

```
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

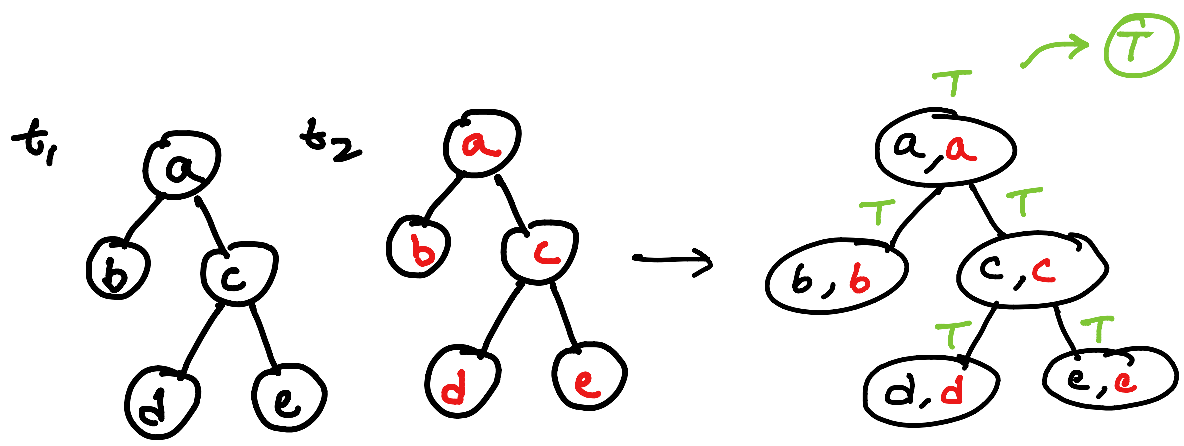
code

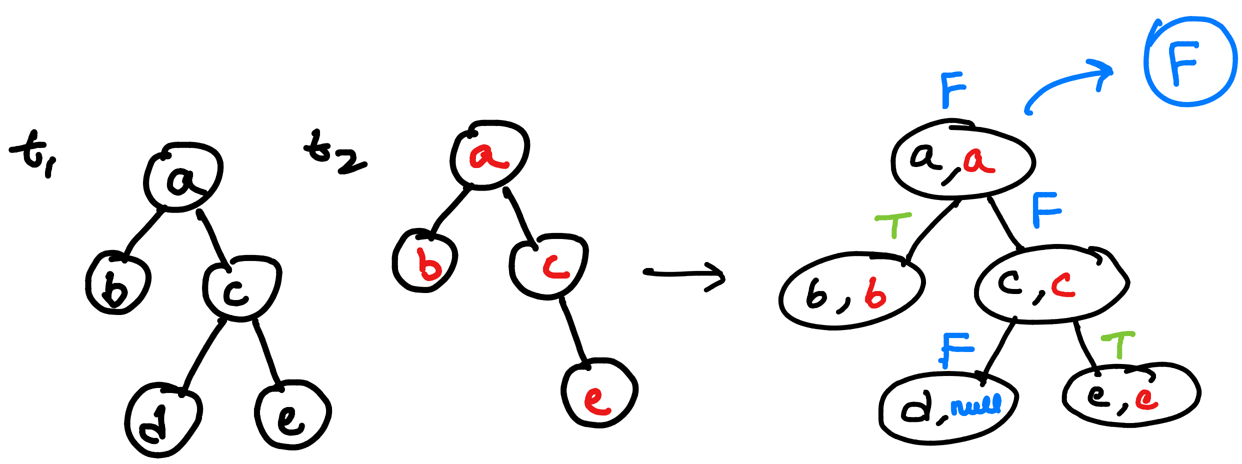
```
class Solution {
public:
    void viewGenerator (TreeNode* root, vector<int> &res, set<int> &s, int currLevel){
        if(root==NULL) return;
        // if level is not reached, then add to result and the set
        if(s.find(currLevel)==s.end()){
            s.insert(currLevel);
            res.push_back(root->val);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





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

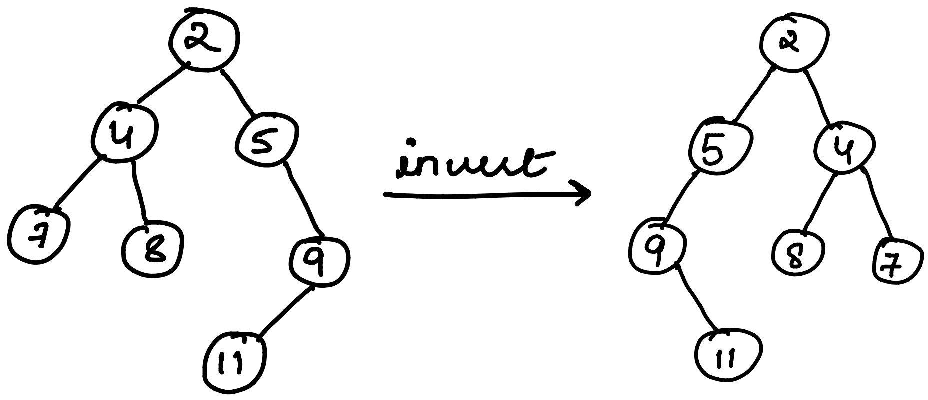
$SC \rightarrow O(1) + O(\min(h1, h2))$

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);

    }
};
```

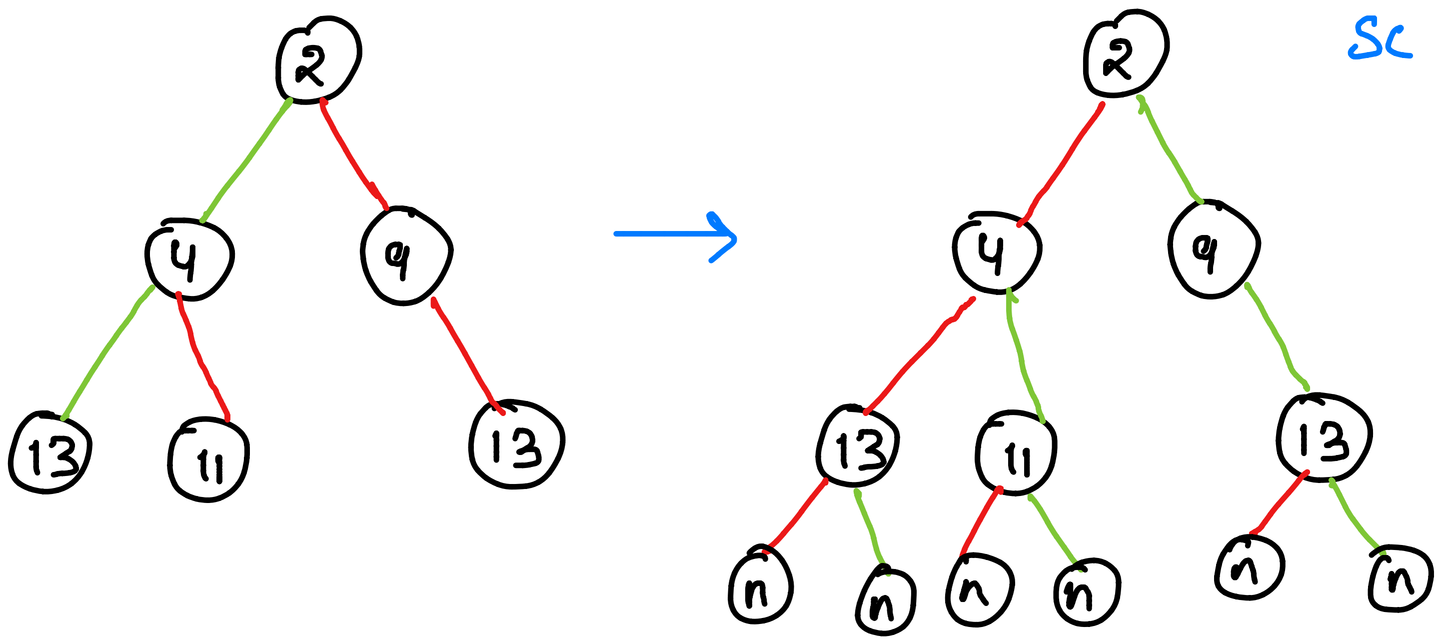
(18) Invert Binary Tree → given the root of BT, find its mirror img.

Eg



TC → O(n)

SC → O(n)+O(h)



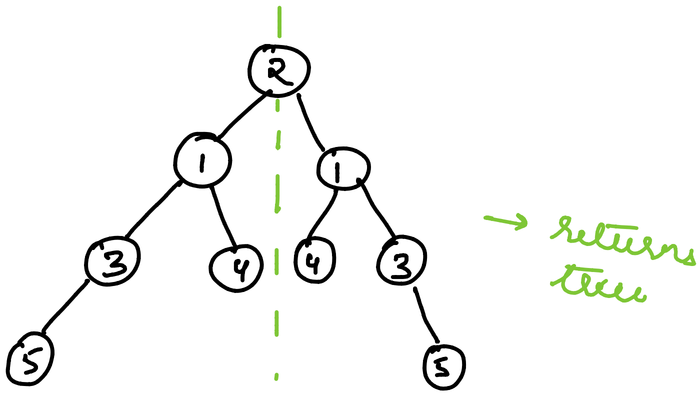
Code →

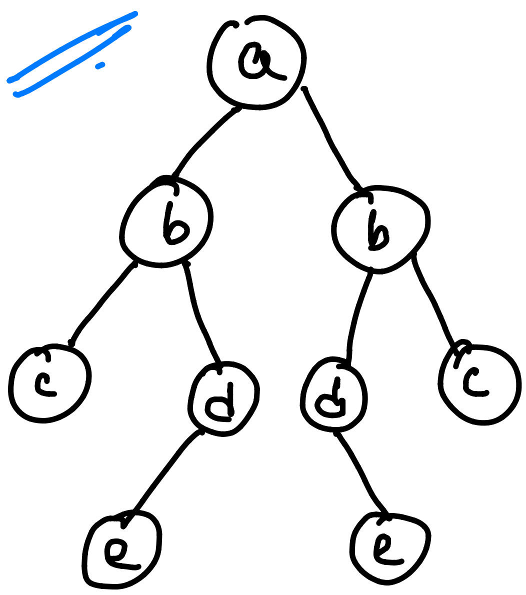
```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(root==NULL)  return root;

        /* 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;
    }
};
```

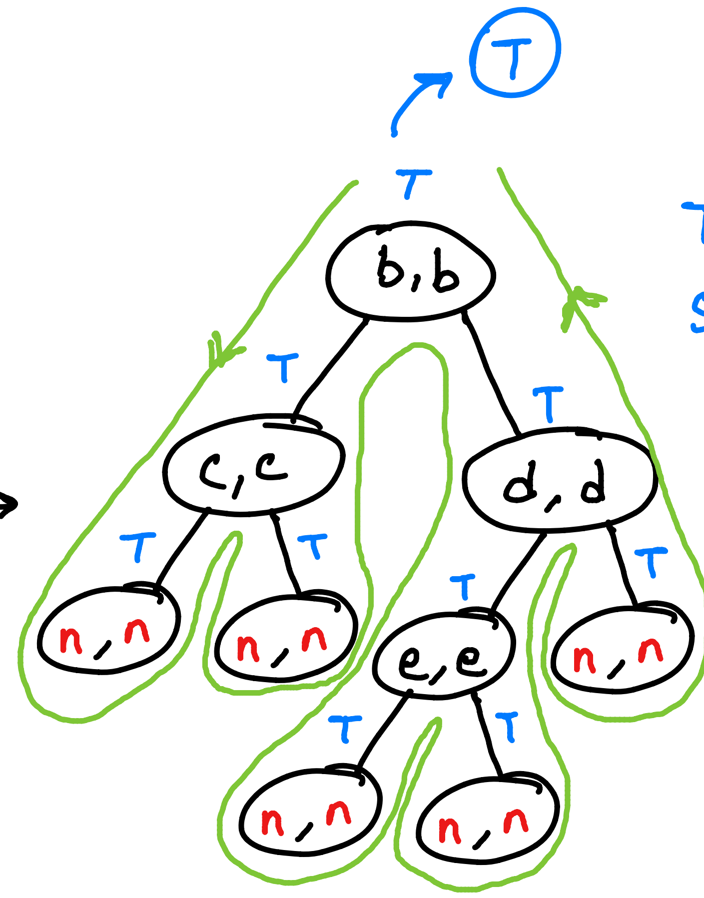
D7

19 Symmetric Tree

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







TC → O(n)

SC → O(1) + O(h)

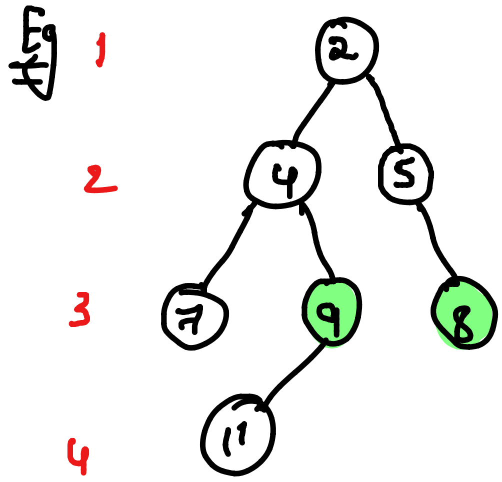
code →

```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);
    }
};
```

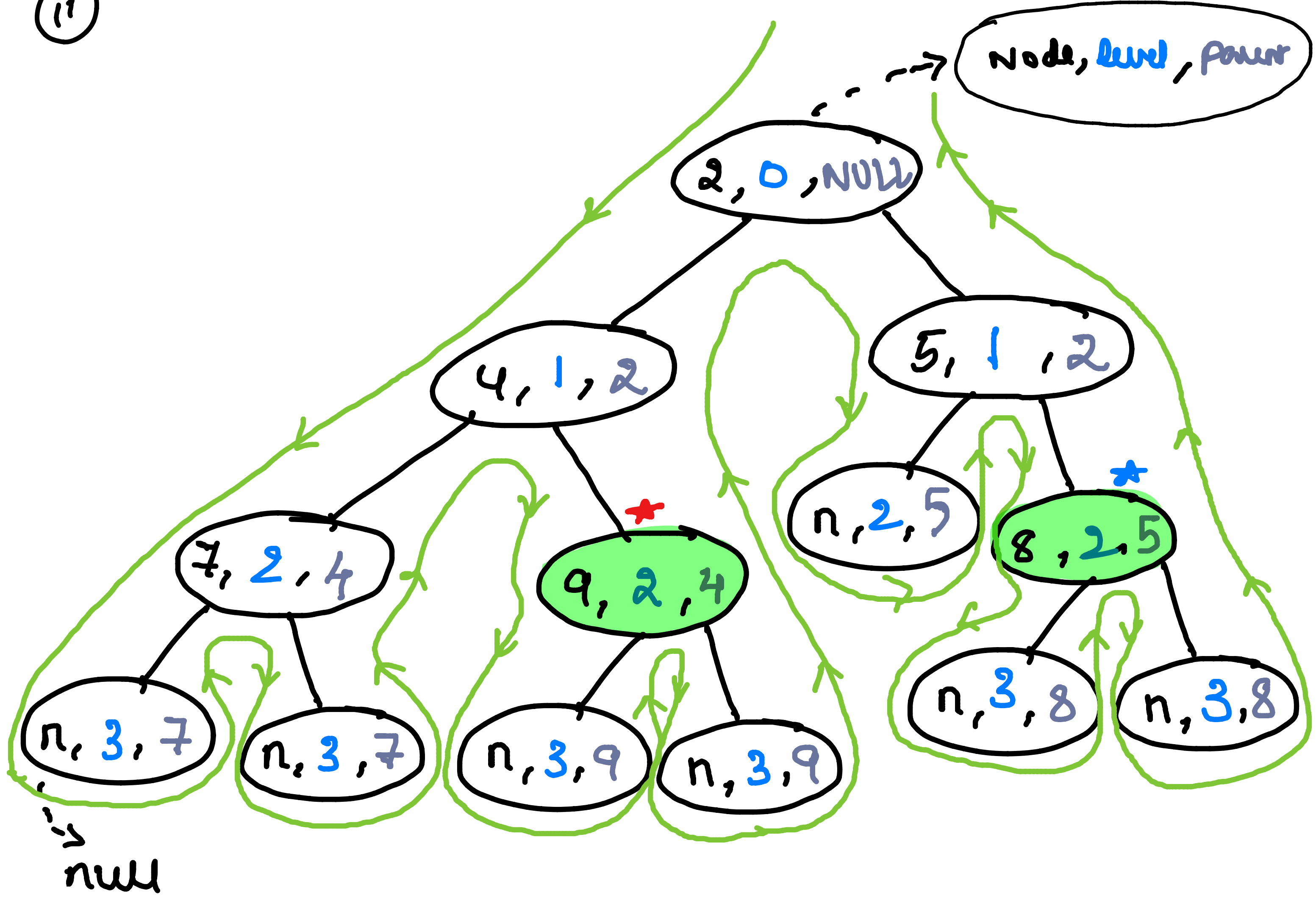
(20) Cousins of a Binary Tree → given two nodes, find if they are cousins of each other.

same level but diff parent.

Eg



x = 9 } cousins
y = 8 }



* 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;
    }
};
```

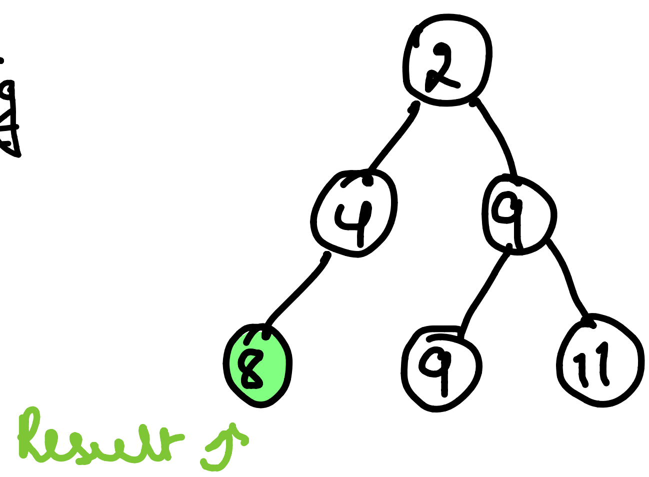
Trees - Part 2

- Karun Karthik

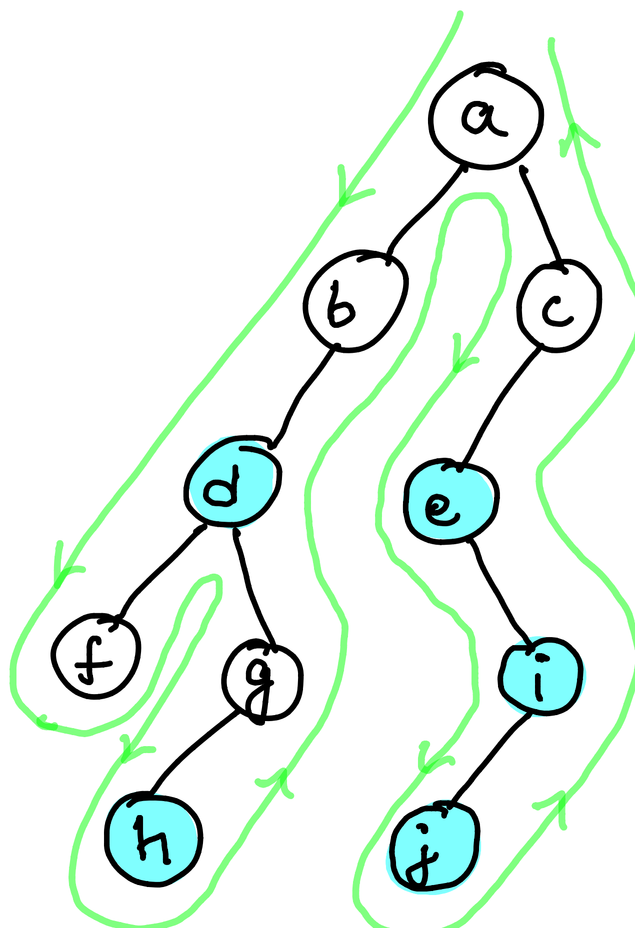
Contents

D7 (21) Print all nodes that donot have any siblings

Eg



Sibling → Same level, Same parent



Tc → O(n)

Sc → O(n)

At every node, check if

both branches exist → then call both of them recursively

only left branch exist → then call left branch recursively

only right branch exist → then call right branch recursively

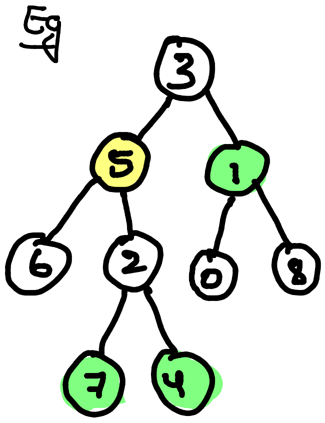
Code →

```cpp
void findNode(Node* root, vector<int>&res){

    if(root==NULL) return;
    if(root->left == NULL && root->right==NULL) return;

    // both branches present then call recursively
    if(root->left!=NULL && root->right!=NULL)
    {
        findNode(root->left, res);
        findNode(root->right, res);
    }
    else if(root->left!=NULL)   // right branch absent
    {
        res.push_back(root->left->data);
        findNode(root->left, res);

    } else if(root->right!=NULL)    // left branch absent
    {
        res.push_back(root->right->data);
        findNode(root->right, res);
    }
    return;
}

vector<int> noSibling(Node* node)
{
    vector<int> res;
    findNode(node, res);
    if(res.size()==0) res.push_back(-1);
    sort(res.begin(), res.end());
    return res;
}
```

D8 (22) All nodes distance K in Binary Tree

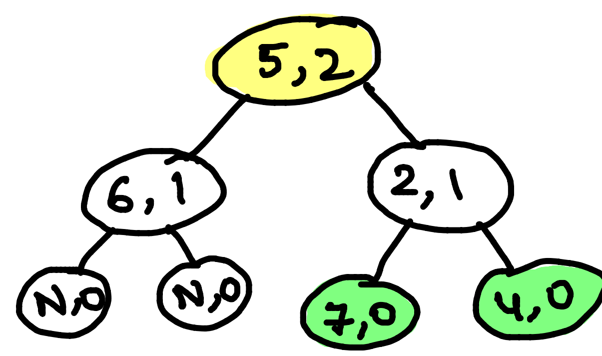
Given a source node, find all the nodes that are at a distance of K units.

Eg



Source = 5
$k = 2$

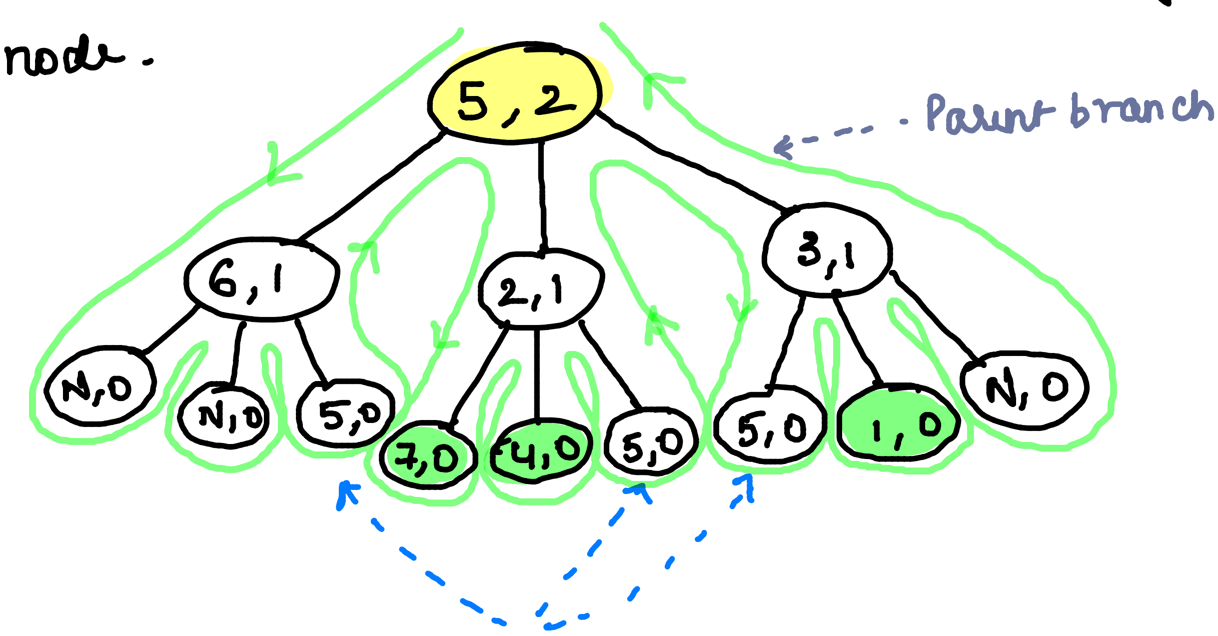
① Consider nodes in downward direction of source.



- 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.

Node	Parent
3	NULL
5	3
1	3
6	5
2	5
0	1
8	1
7	2
4	2



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

* If $k=0$ & the node value is valid (not null) then add to result array

$Tc \rightarrow \underbrace{O(n)}_{\text{populating}} + \underbrace{O(n)}_{\text{traversal}}$

$Sc \rightarrow O(n) + O(n) + \underbrace{O(x)}_{\text{result}}$

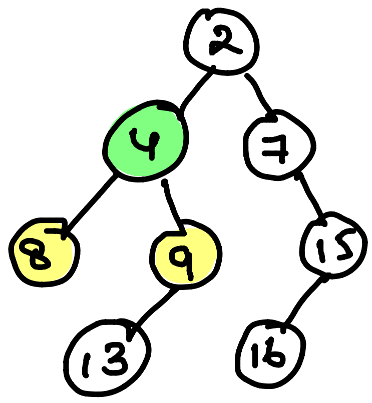
https://www.linkedin.com/in/karun-karthik-5a0794187/

Code.

```
class Solution {
public:
    // to create hashtable
    void populatemap(TreeNode* currnode, TreeNode* currparent,
                     unordered_map<TreeNode*,TreeNode*>&parentmap){
        if(currnode == NULL) return;
        parentmap[currnode] = currparent;
        populatemap(currnode->left,currnode,parentmap);
         populatemap(currnode->right,currnode,parentmap);
        return;
    }

    // finding all the nodes at distance K
    void printkdistance(TreeNode* currnode, int k,set<TreeNode*>&s,
        unordered_map<TreeNode*,TreeNode*>&parentmap,vector<int>&ans)
    {
        if(currnode == NULL || s.find(currnode)!=s.end()|| k<0)
            return;

        s.insert(currnode);

        if(k==0)
        {
            ans.push_back(currnode->val);
            return;
        }

        printkdistance(currnode->left,k-1,s,parentmap,ans);   // call left child
        printkdistance(currnode->right,k-1,s,parentmap,ans);   // call right child
        printkdistance(parentmap[currnode],k-1,s,parentmap,ans);  // call the parent
        return;
    }

    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
        vector<int>ans;
        set<TreeNode*>s;
        unordered_map<TreeNode*,TreeNode*>parentmap;
        populatemap(root,NULL,parentmap);
        printkdistance(target,k,s,parentmap,ans);
        return ans;
    }
};
```

(23) Lowest Common Ancestor

Eg



node to root paths

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

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

→ for every node, check if it matches n1 or n2.
if found return node
else call recursively in both branches.
if both return non-null value ⇒ root is LCA
else return the branch value that is non-null.

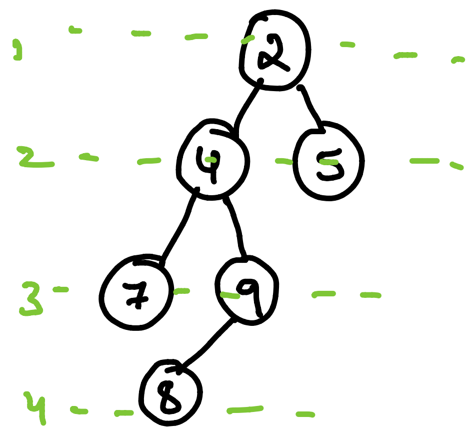
code

```
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)

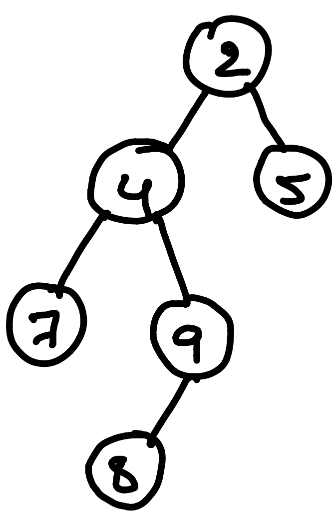
SC $\rightarrow$ O(n)

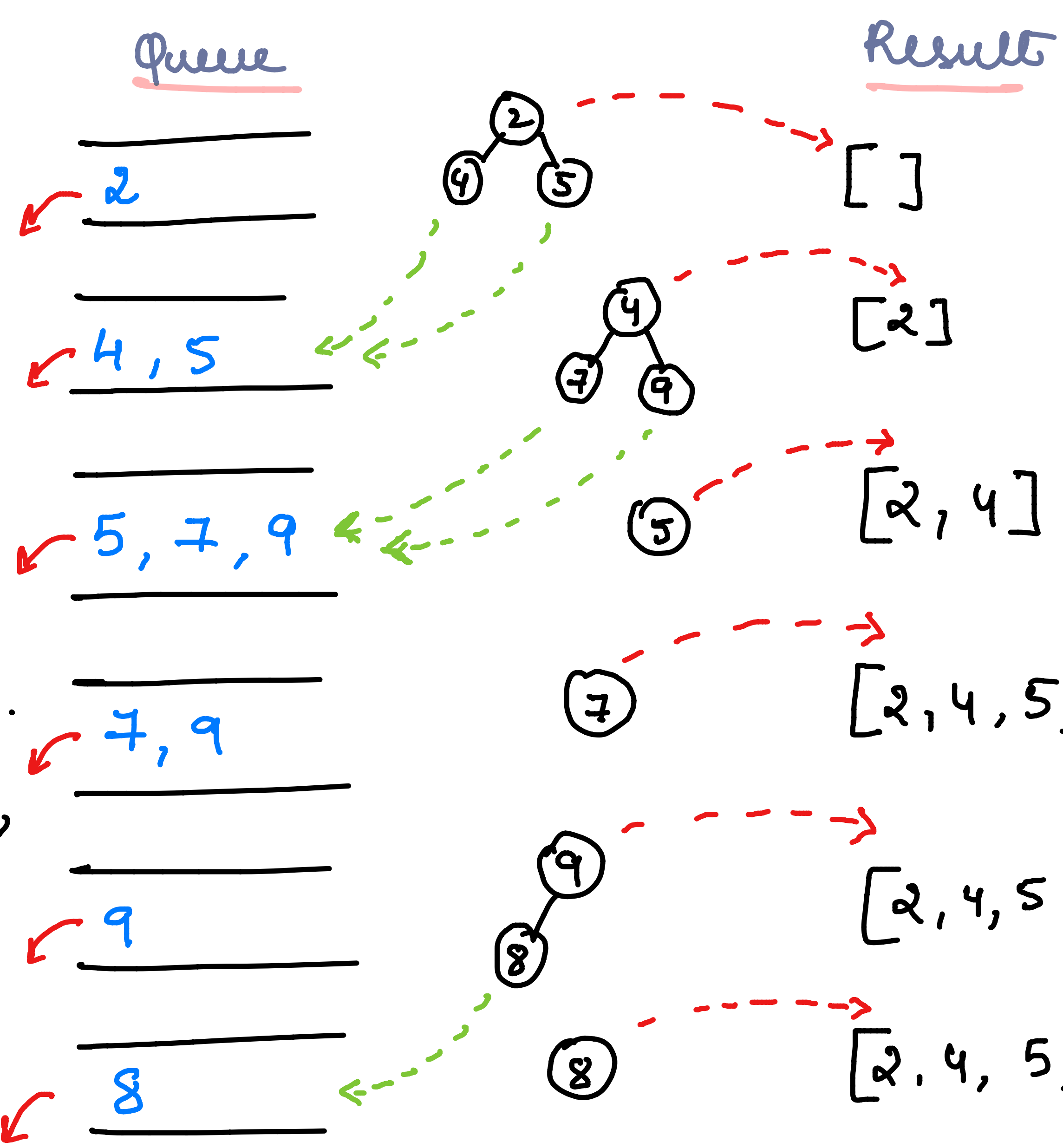
→ To find level order traversal use queue. → FIFO datastructure

Inserting → Rear

Removing → Front

→ Before removing from queue, add the children to the queue (BFS)





① For every node, enqueue.

② while dequeue, enqueue its branches.

empty →

Answer → [2, 4, 5, 7, 9, 8]

https://www.linkedin.com/in/karun-karthik-5a0794187/

Code →

```cpp
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        queue<TreeNode*> q;

        if(root==NULL)  return res;
        q.push(root);

        while(!q.empty()){

            int currsize = q.size();
            vector<int>currLevel;

            while(currsize>0)
            {
                TreeNode* currnode = q.front();
                q.pop();
                currLevel.push_back(currnode->val);
                currsize--;

                if(currnode->left!=NULL)
                q.push(currnode->left);

                if(currnode->right!=NULL)
                q.push(currnode->right);
            }
            res.push_back(currLevel);
        }
        return res;
    }
};
```

(25) Level Order traversal N-ary Tree

→ Everything is same as previous problem, intuition & complexity

TC → O(n)

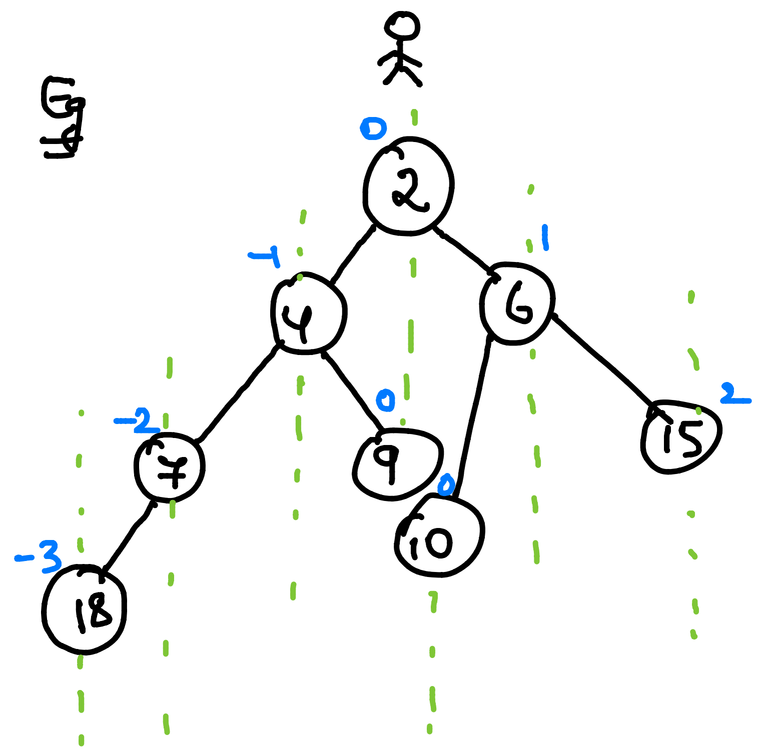
SC → O(n)

code →

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> res;
        queue<Node*>q;

        if(root == NULL) return res;
        q.push(root);

        while(!q.empty())
        {
            int currsize = q.size();
            vector<int>currLevel;
            while(currsize>0)
            {
                Node* currnode = q.front();
                q.pop();
                currLevel.push_back(currnode->val);
                currsize--;

                // enqueue all the children
                for(auto child:currnode->children)
                    q.push(child);
            }
            res.push_back(currLevel);
        }
         return res;
    }
};
```

26 Top View of Binary Tree

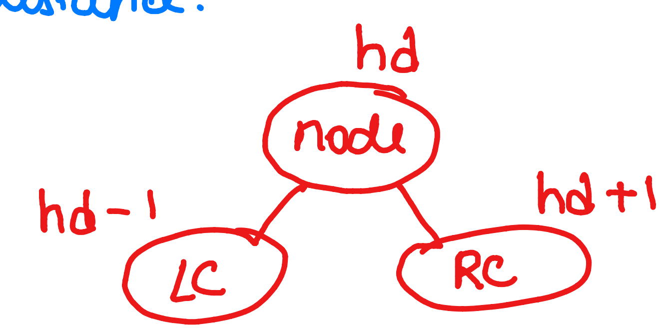
Eg



Left to Right
↳ [18, 7, 4, 2, 6, 15]

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

*



* hd of root = 0

* make a pair with node & it's 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 results.

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

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.

→ convert into array & return as result

https://www.linkedin.com/in/karun-karthik-5a0794187/

code

```cpp
class Solution
{
    public:
    vector<int> topView(Node *root)
    {
        vector<int> res;
        if(root==NULL) return res;

        map<int,int> mp;
        queue<pair<Node*,int>> q;

        q.push({root,0});

        while(!q.empty()){

            auto it = q.front();
            q.pop();

            Node* node = it.first;
            int hd = it.second;

            if(mp.find(hd) == mp.end())
                mp[hd] = node->data;

            if(node->left!=NULL)
                q.push({node->left,hd-1});


            if(node->right!=NULL)
                q.push({node->right,hd+1});
        }

        // store in vector or array
        for(auto it:mp)
            res.push_back(it.second);

        return res;
    }
};
```

logn → map.

Tc → O(nlogn)

Sc → 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 →

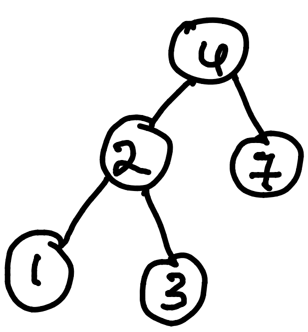
```cpp
class Solution {
  public:
    vector <int> bottomView(Node *root) {
        vector<int> res;
        if(root==NULL) return res;

        map<int, int> mp;
        queue<pair<Node*, int>>q;

        q.push({root, 0});
        while(!q.empty()){
            auto it = q.front();
            q.pop();

            Node* node = it.first;
            int hd = it.second;

            mp[hd] = node->data;

            if(node->left!=NULL)
                q.push({node->left, hd-1});

            if(node->right!=NULL)
                q.push({node->right, hd+1});
        }

        for(auto it:mp)
            res.push_back(it.second);

        return res;
    }
};
```

D10 **Binary Search Trees**

- 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

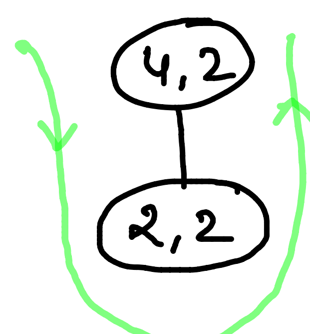
1. LC < node < RC
2. LC ≤ node < RC
3. LC < node ≤ RC

28 Search in a BST



Val = 2

⇒ return the subtree with given value.



- as 2 < 4, search in LST.
- as 2 == 2 return node.

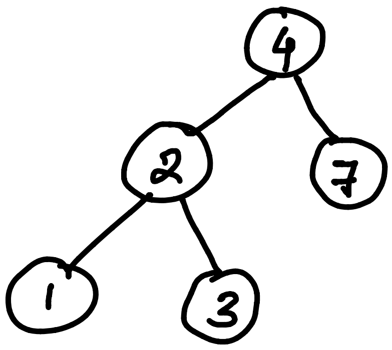
TC → $O(\log_2 n)$ avg, $O(n)$ worst

SC → $O(n)$

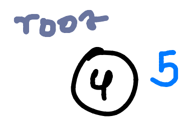
code

```
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);
    }
};
```

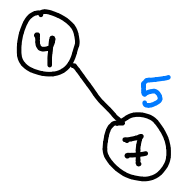
29 Insert into BST



val = 5.

1) 

As 5 > 4, go to RST

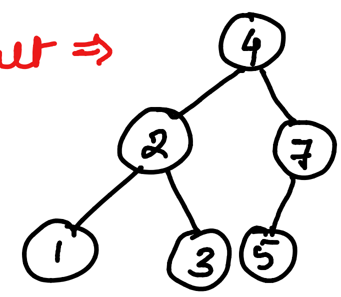
2) 

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 ⇒



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

SC → $O(1)$

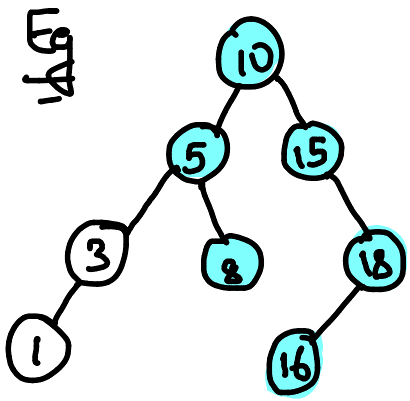
Code →

```
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

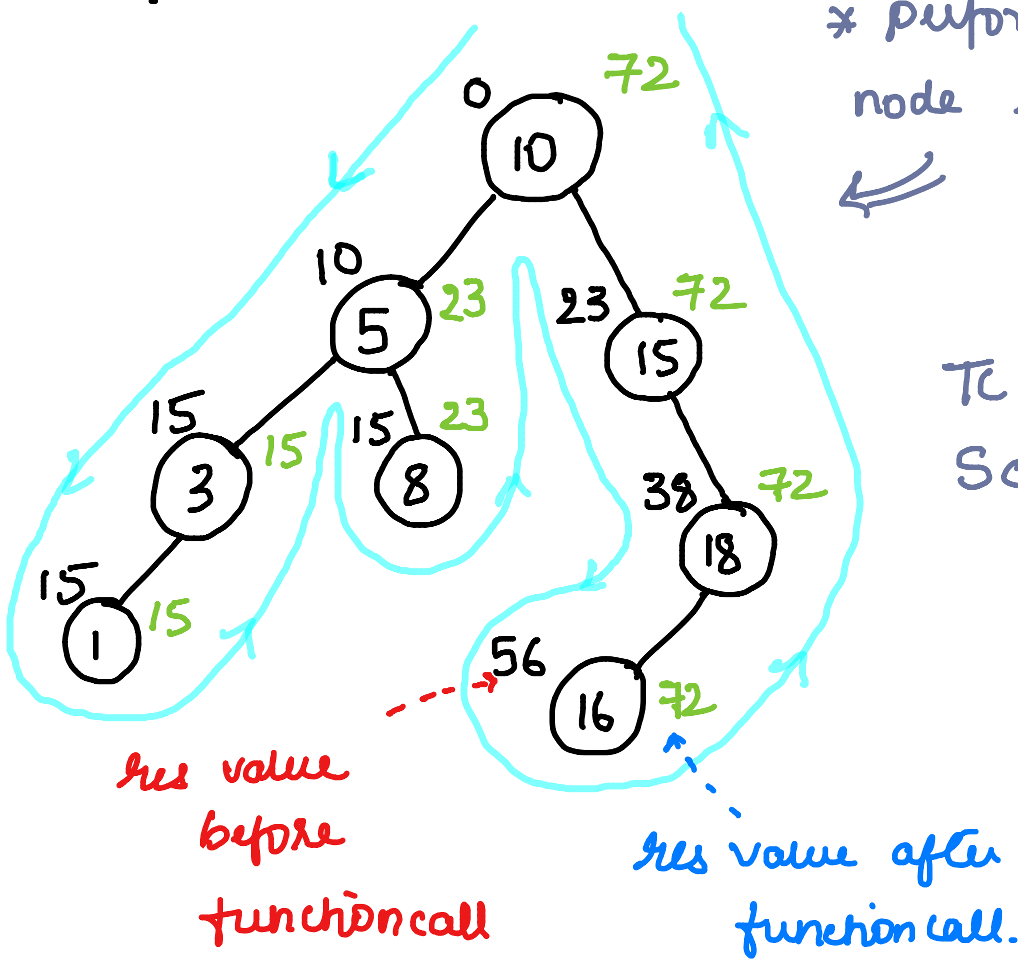
given a root node & interval [X,Y], find sum of all nodes that lies in [X,Y].

Eg



range → [5, 18]

sum = 72



* perform addition if node lies b/w [X,Y]

TC → O(n)

SC → O(n)

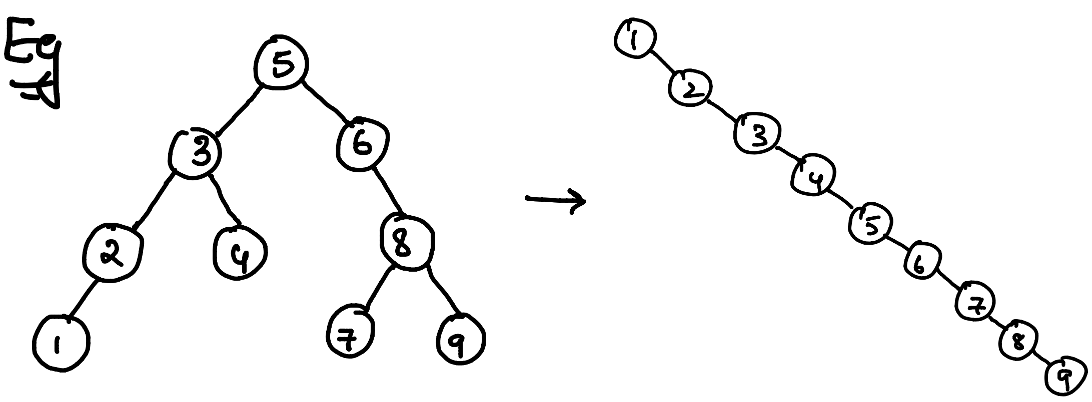
Code →

```cpp
class Solution {
public:
    void sumUtil(TreeNode* root, int low, int high, int &res){
        if(root==NULL)  return;
        if(root->val <= high && root->val >= low){
            res += root->val;
        }
        sumUtil(root->left, low, high, res);
        sumUtil(root->right, low, high, res);
    }

    int rangeSumBST(TreeNode* root, int low, int high) {
        int res = 0;
        sumUtil(root, low, high, res);
        return res;
    }
};
```

(31) Increasing order search tree

Given a BST, create an increasing order search tree.

Eg



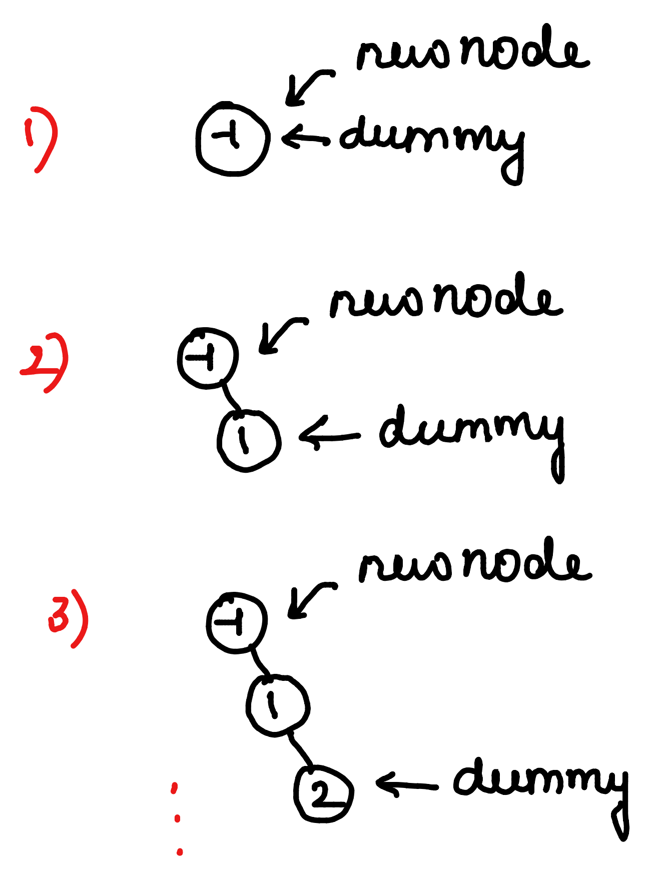
(1) Perform inorder traversal.

(2) create a skewed tree using elements in inorder traversal.

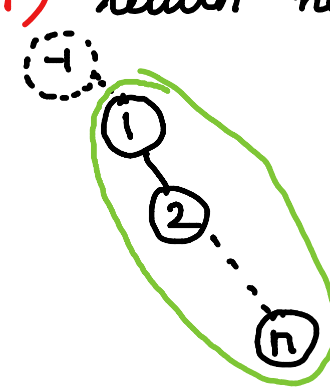
Code

```
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



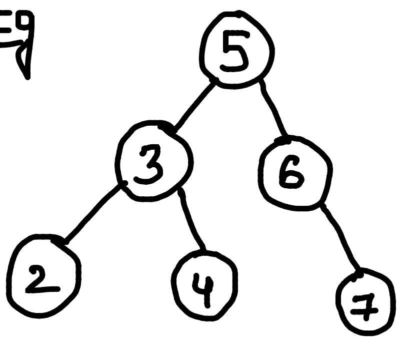
Last) return newNode->right



(32) Two sum IV - Input is a BST

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

Eg



① Perform Inorder & Store in array

② use 2-pointer approach

$k = 9$ ⇒

	0	1	2	3	4	5
v =	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

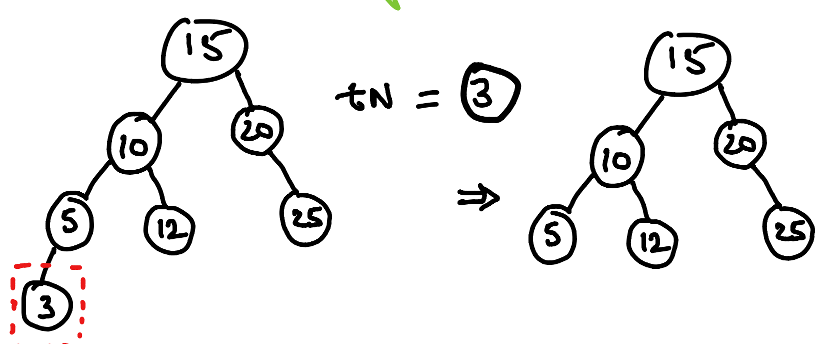
given root of BST & a target node, delete the target node & return the tree.

Tc →
Avg ⇒ O(log n)
worst ⇒ O(n)

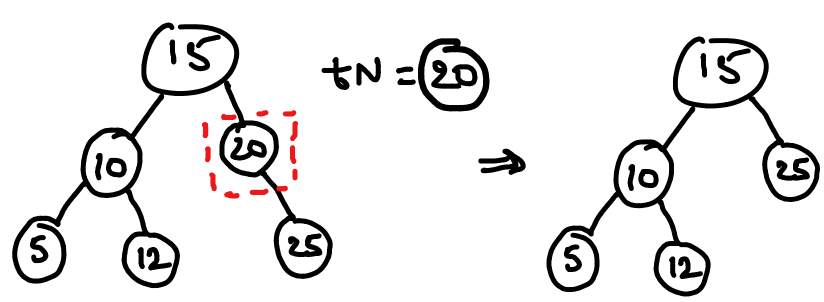
SC → O(h)

cases →

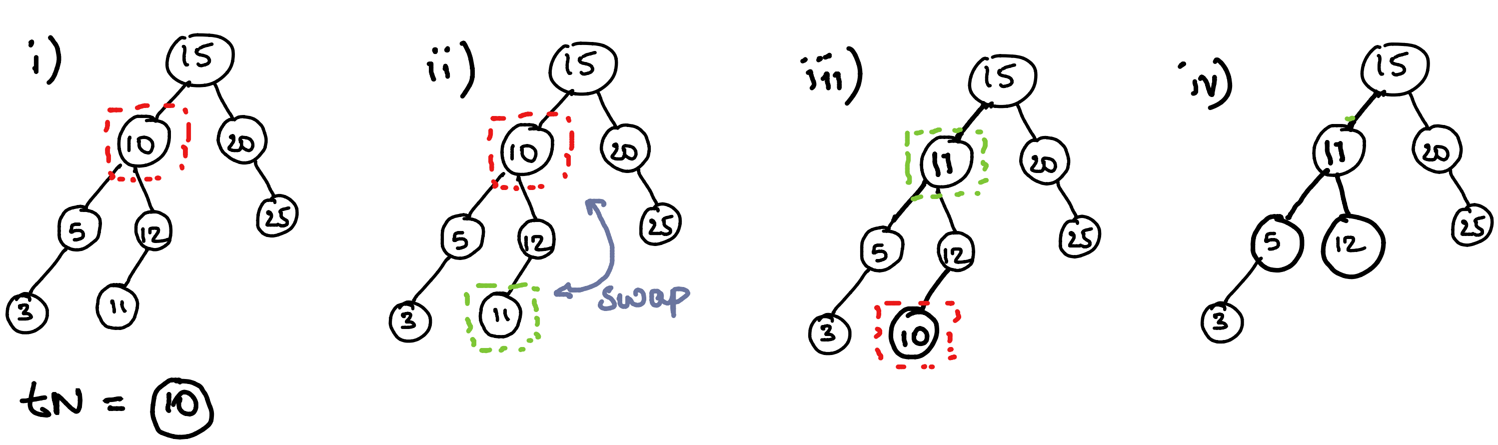
① If target node is leaf →
then simply delete it



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

```cpp
class Solution {
public:
    TreeNode* findleftmostNode(TreeNode* root){
        while(root->left!=NULL)
            root = root->left;
        return root;
    }

    TreeNode* deleteNode(TreeNode* root, int key) {

        if(root==NULL)  return NULL;

        if(root->val > key)
            root->left = deleteNode(root->left, key);

        else if(root->val < key)
            root->right = deleteNode(root->right, key);

        else {  // root->val == key
            if(root->left == NULL && root->right == NULL){
                root = NULL;
                return root;
            }
            if(root->left != NULL && root->right == NULL){
                root = root->left;
                return root;
            }
            if(root->right != NULL && root->left == NULL){
                root = root->right;
                return root;
            }

            // finding left most node in right subtree
            TreeNode* temp = findleftmostNode(root->right);

            //swapping root's value with left most node's val
            int tempVal = root->val;
            root->val = temp->val;
            temp->val = tempVal;

            // performing delete in right subtree
            root->right = deleteNode(root->right, key);
            return root;
        }
        return root;
    }
};
```