

Trees - Part 2

- Karun Karthik

Contents

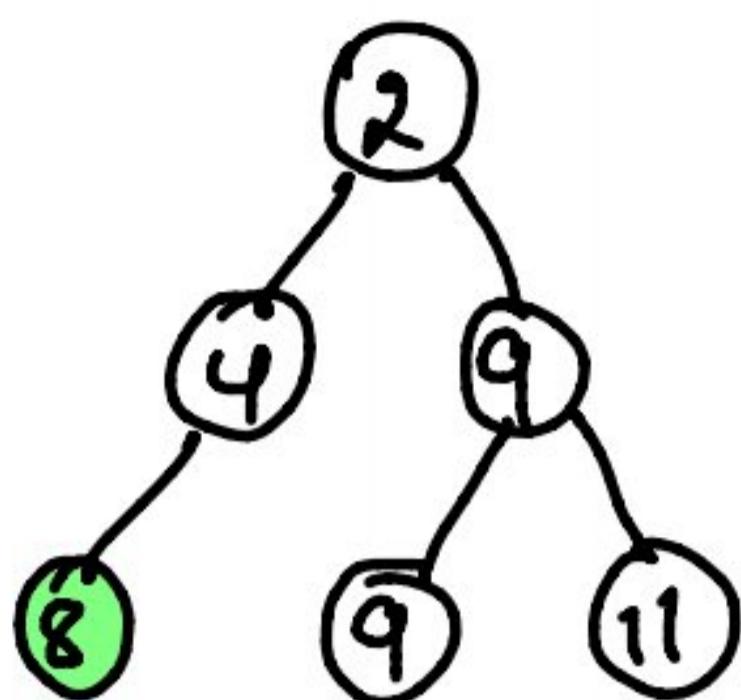
21. Print all nodes that do not have any siblings
22. All nodes at distance K in a Binary Tree
23. Lowest Common Ancestor
24. Level order traversal in Binary Tree
25. Level order traversal in N-ary Tree
26. Top view of Binary Tree
27. Bottom view of Binary Tree
28. Introduction to Binary Search Tree & 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

Q21

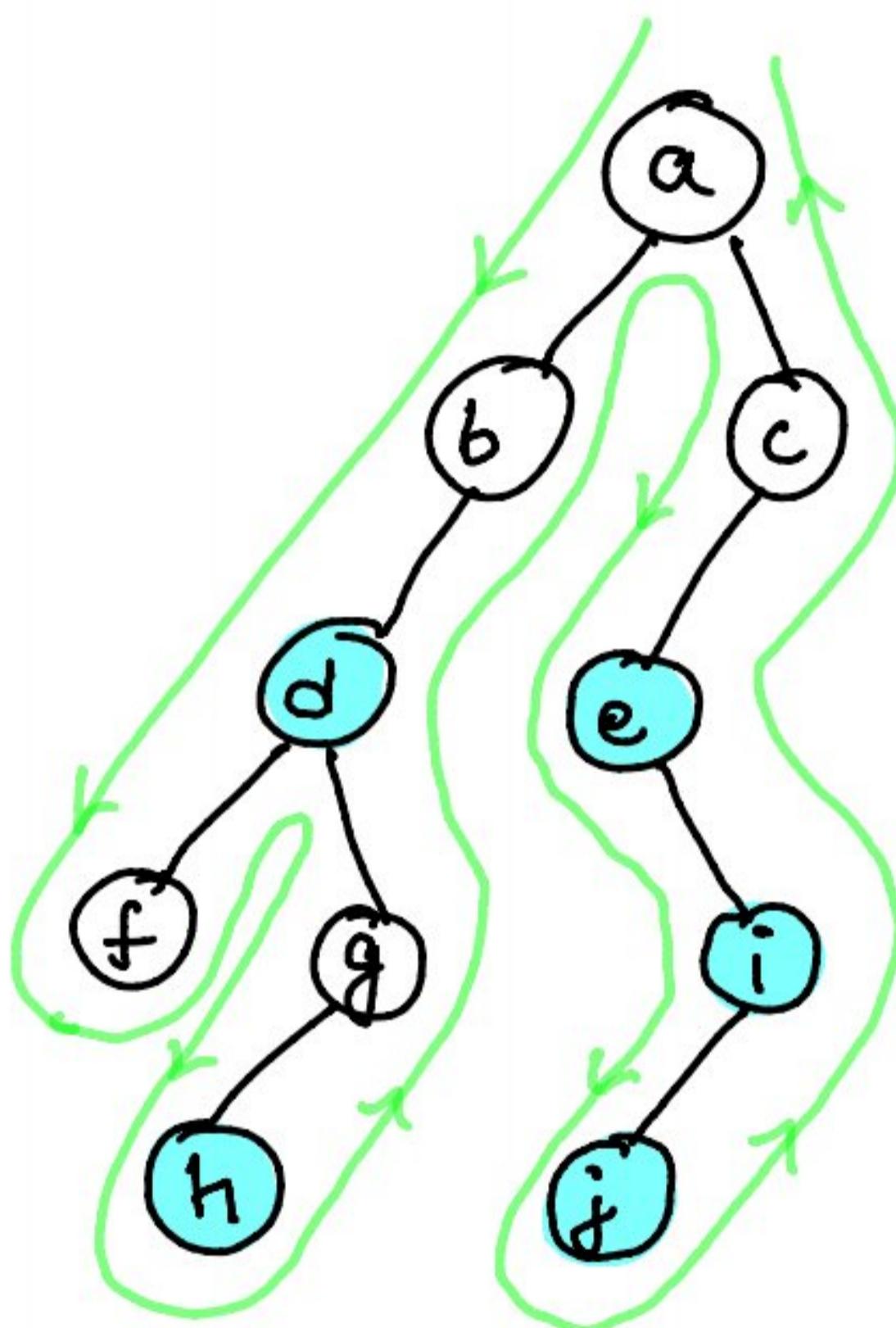
Print all nodes that donot have any siblings

Ex



Result ↗

Sibling \rightarrow same level, same parent



$T_C \rightarrow O(n)$

$S_C \rightarrow O(n)$

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 →



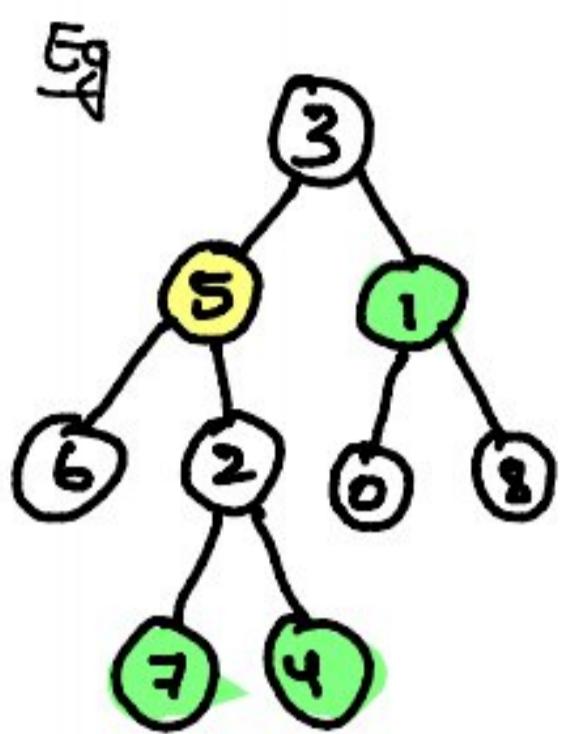
```
1 void findNode(Node* root, vector<int>&res){  
2  
3     if(root==NULL) return;  
4     if(root->left == NULL && root->right==NULL) return;  
5  
6     // both branches present then call recursively  
7     if(root->left!=NULL && root->right!=NULL)  
8     {  
9         findNode(root->left, res);  
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

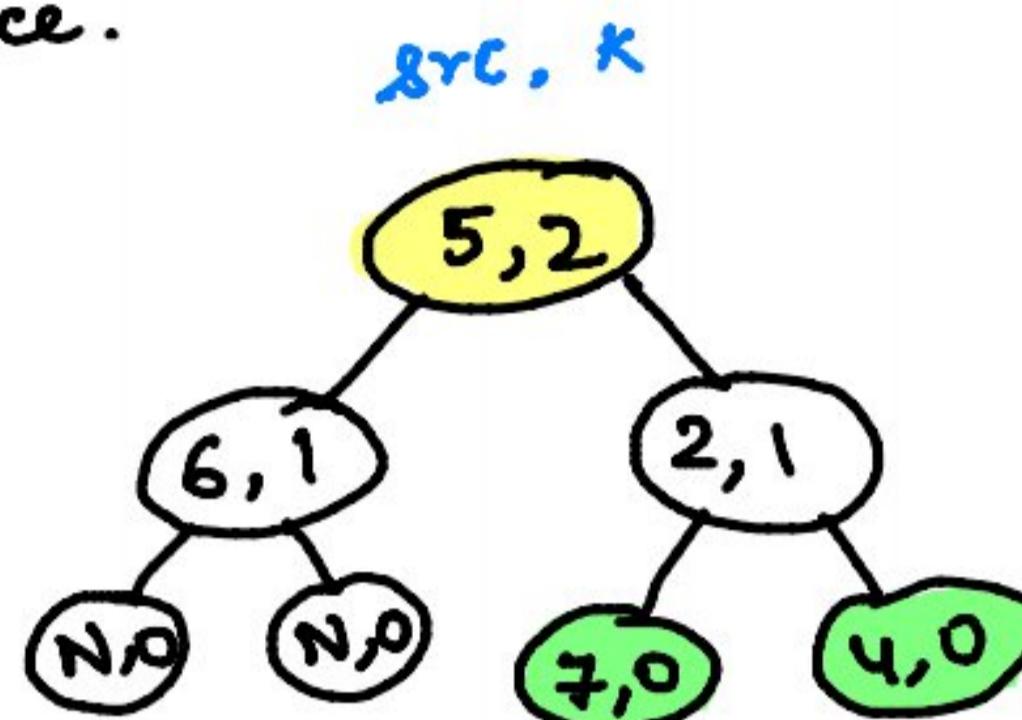
22 All nodes distance k in Binary Tree

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

- ① consider nodes in downward direction of source.

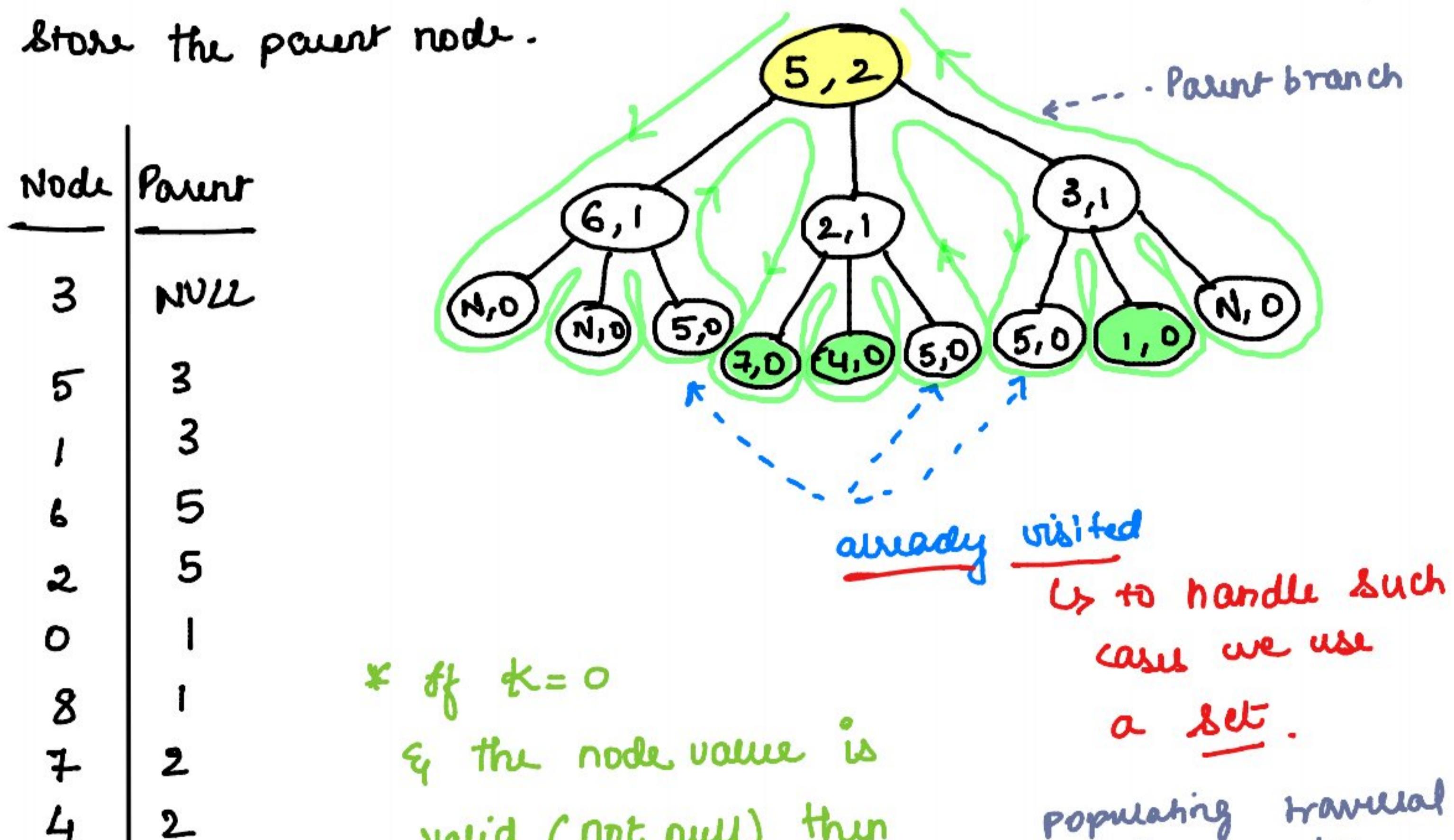


source = 5
 $k = 2$



- 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.



* if $k = 0$

& the node value is valid (not null) then add to result array

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

$$\begin{aligned} \text{TC} &\rightarrow O(n) + O(n) \\ \text{SC} &\rightarrow O(n) + O(n) + O(\alpha) \end{aligned}$$

result

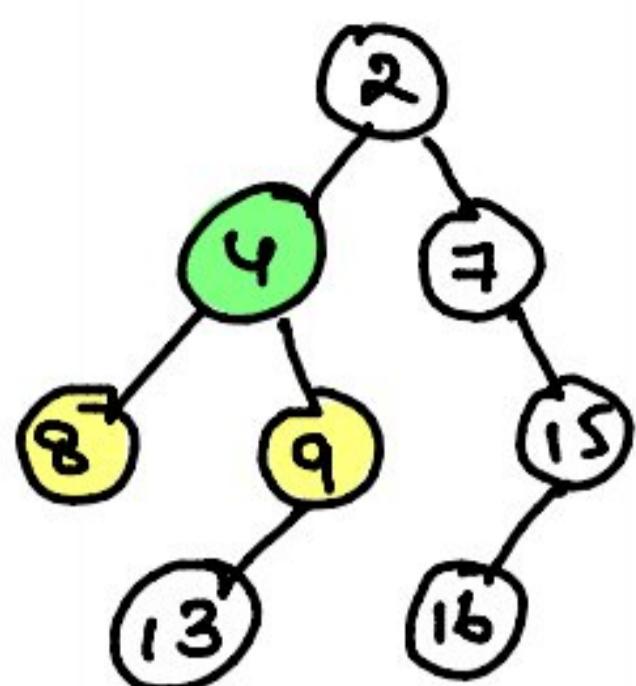
Code

```
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
 \downarrow

$n_1 = 8$, $n_2 = 9$ 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 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;
    }
};
  
```

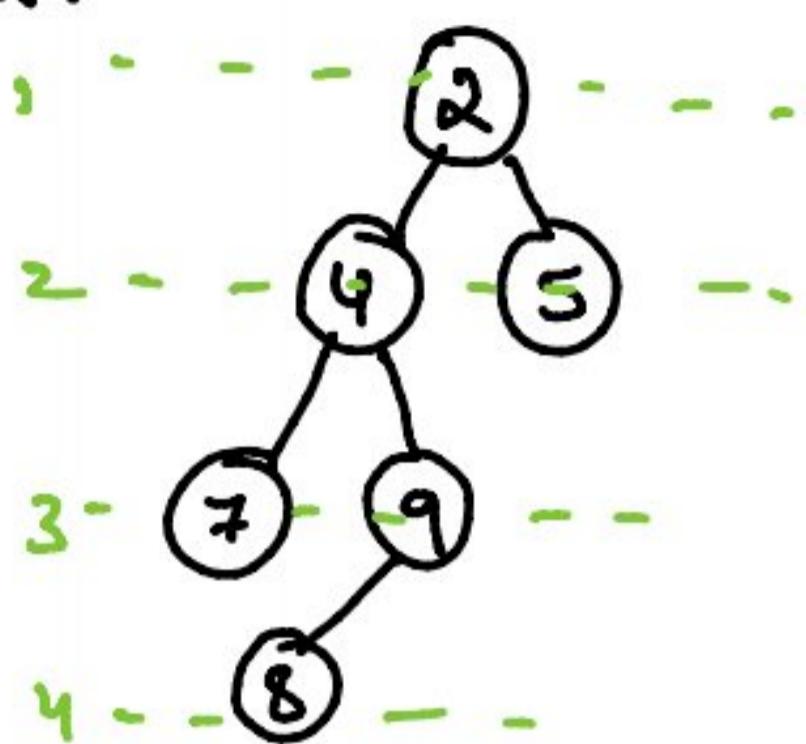
D9

Q4

Level order traversal Binary Tree

Given root node, find level order traversal.

Ex.



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

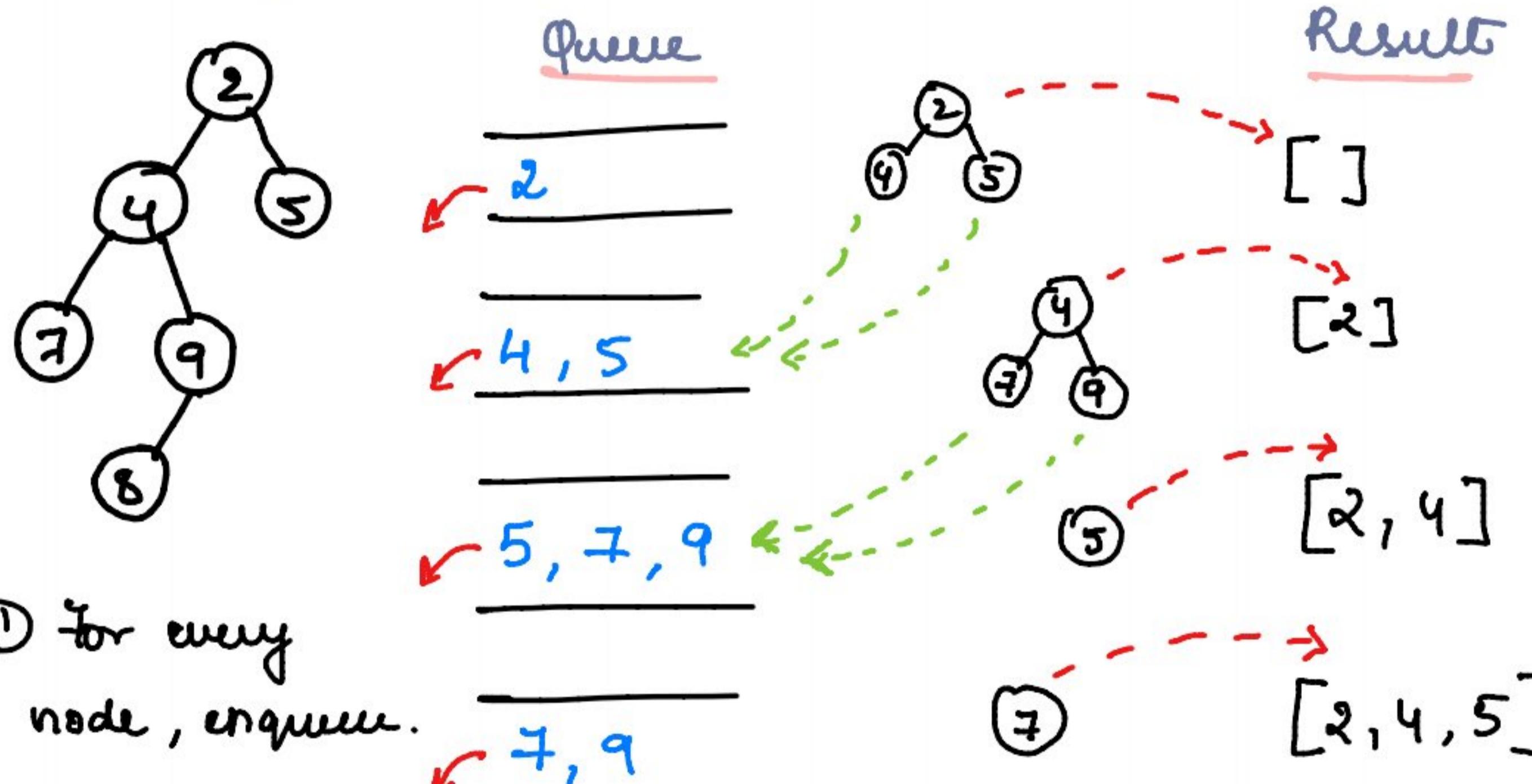
TC $\rightarrow O(n)$

SC $\rightarrow O(n)$

→ To find level order traversal use queue.

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

FIFO
datastructure
Inserting → rear
Removing → front



① For every node, enqueue.

② While dequeue, enqueue in 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

$T_C \rightarrow O(n)$

$S_C \rightarrow O(n)$

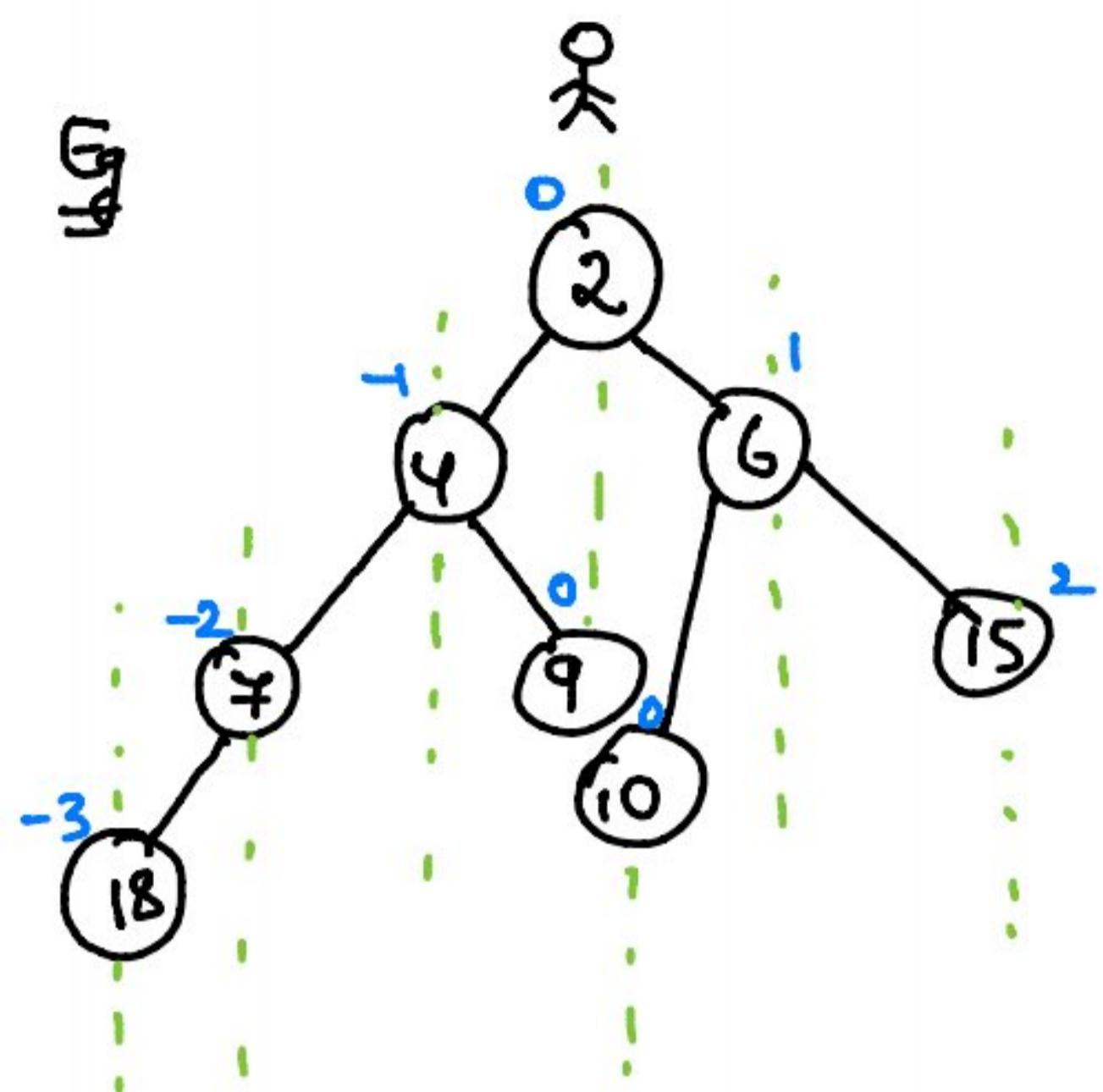
Code →

```
● ● ●

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)

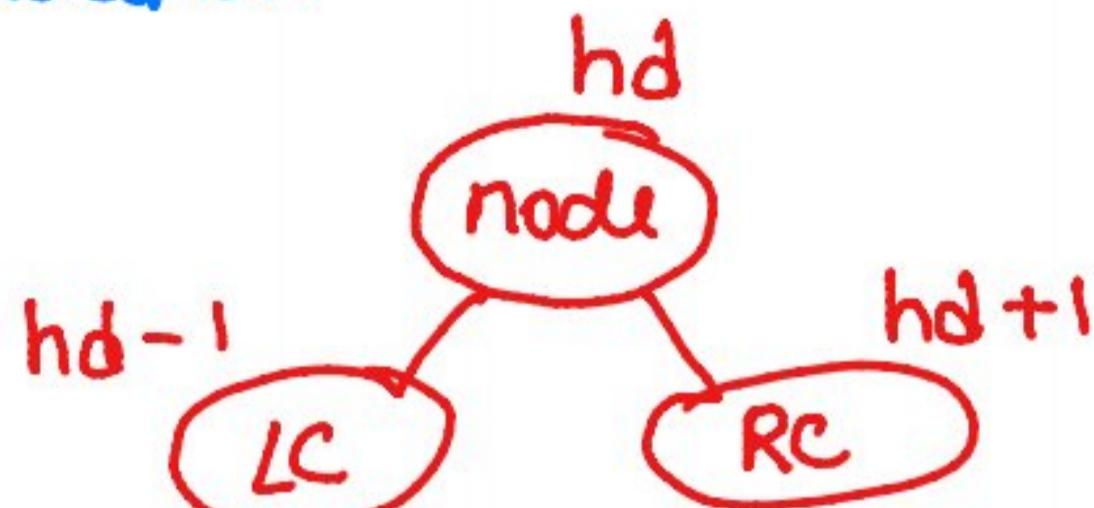
Top View of Binary Tree



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 result.

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

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        // store in vector or array
33        for(auto it:mp)
34            res.push_back(it.second);
35
36        return res;
37    }
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    };
```

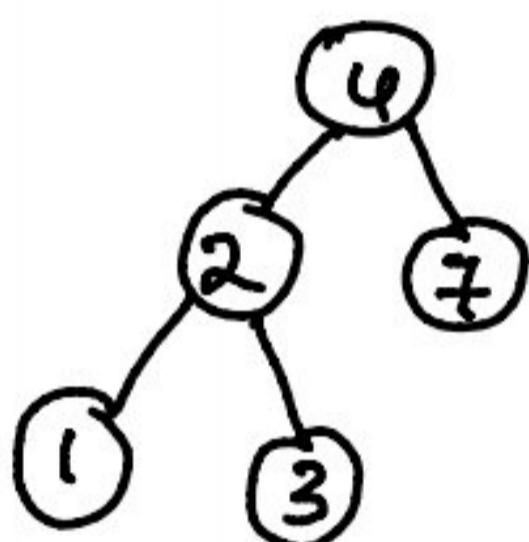
D10

Binary Search Tree

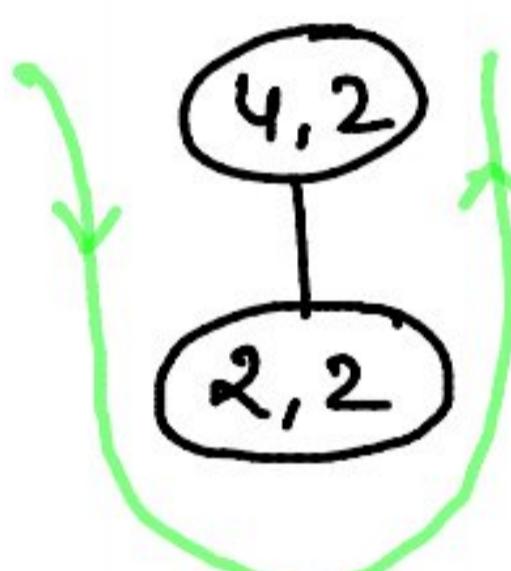
- every node is $>$ than previous node & $<$ than next node.
- if duplicates, then it'll be mentioned that it'll be included in LC or RC

- ① $LC < \text{node} < RC$
- ② $LC \leq \text{node} < RC$
- ③ $LC < \text{node} \leq RC$

(28) Search in a BST

 $\text{val} = 2$

\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

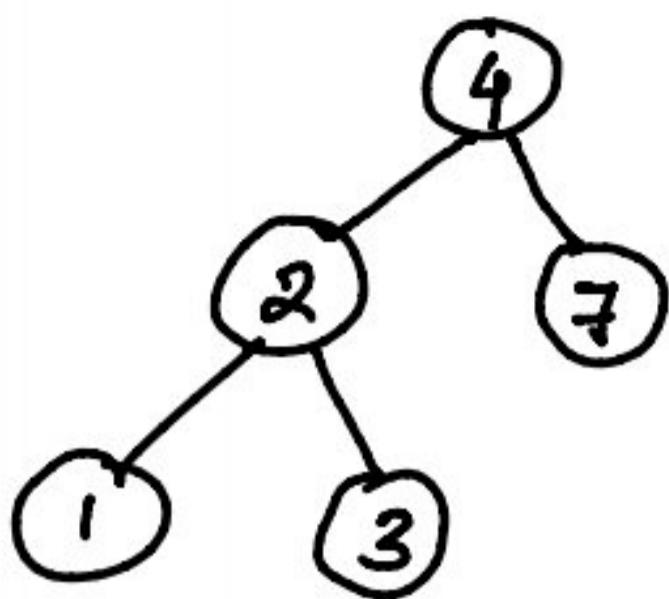
$Sc \rightarrow O(n)$

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);
    }
};
  
```

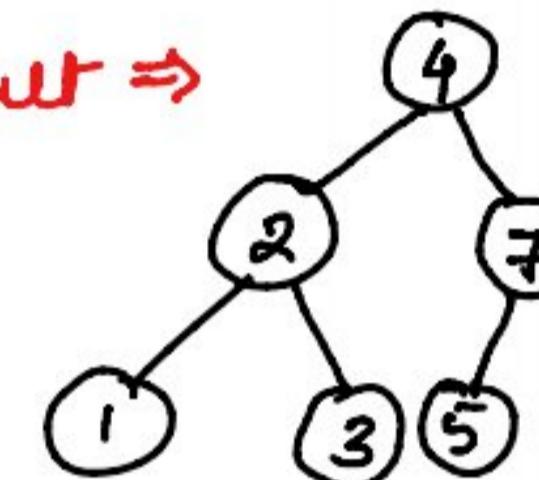
29

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. 
• Link 5 as LST of 7.
- 4) **Result \Rightarrow** 

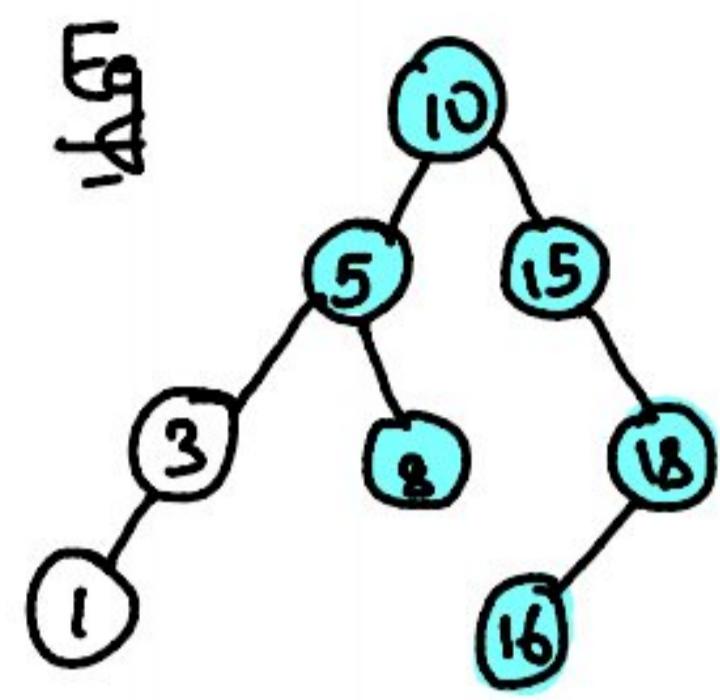
Code \rightarrow

```

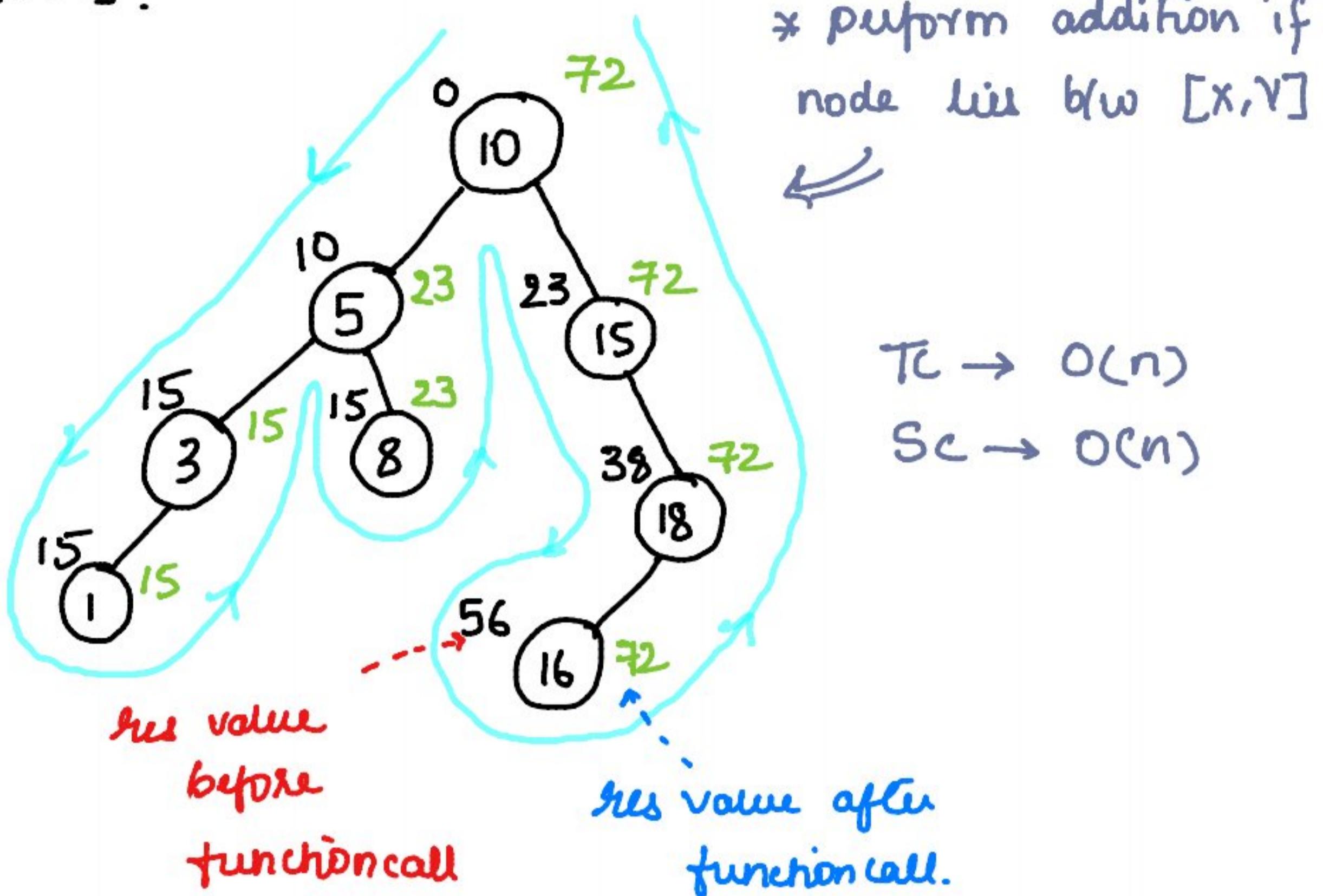
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* node, int val) {
        if(node==NULL){
            return new TreeNode(val);
        }
        if (val < node->val) {
            node->left = insertIntoBST(node->left, val);
        }
        else {
            node->right = insertIntoBST(node->right,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

30 Range sum of BST

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



range $\rightarrow [5, 18]$
sum = 72



Code →

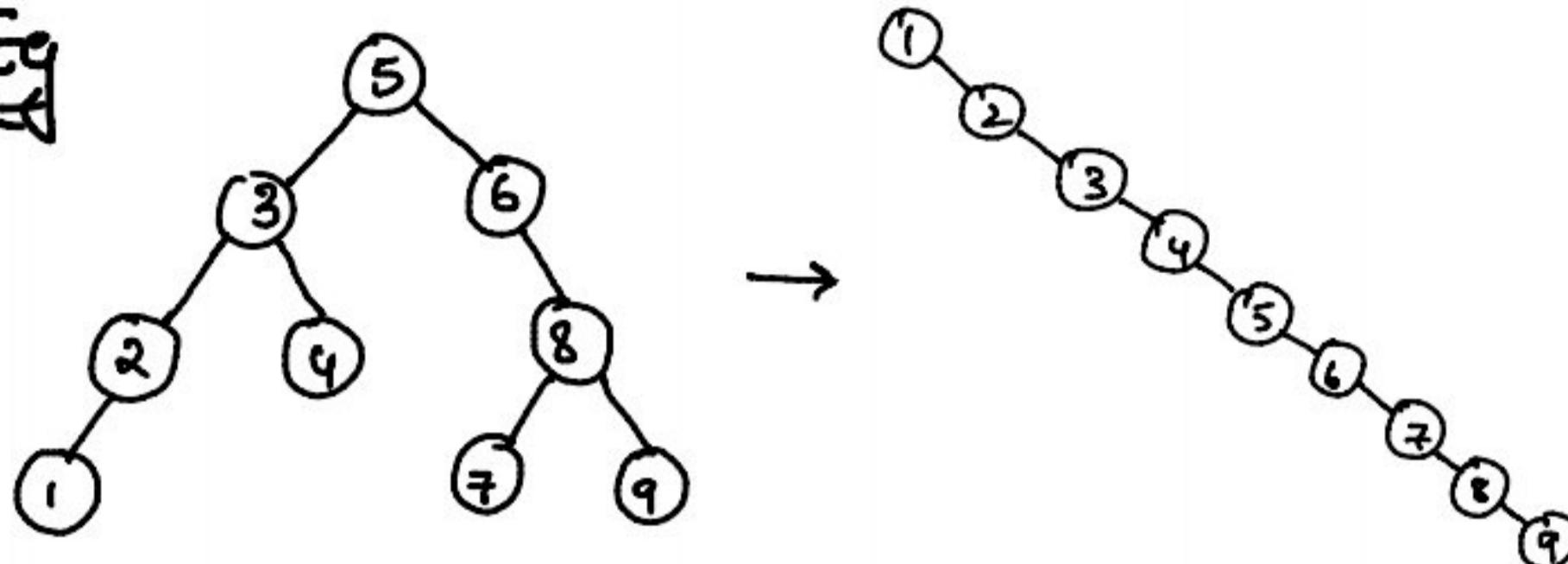
```

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.

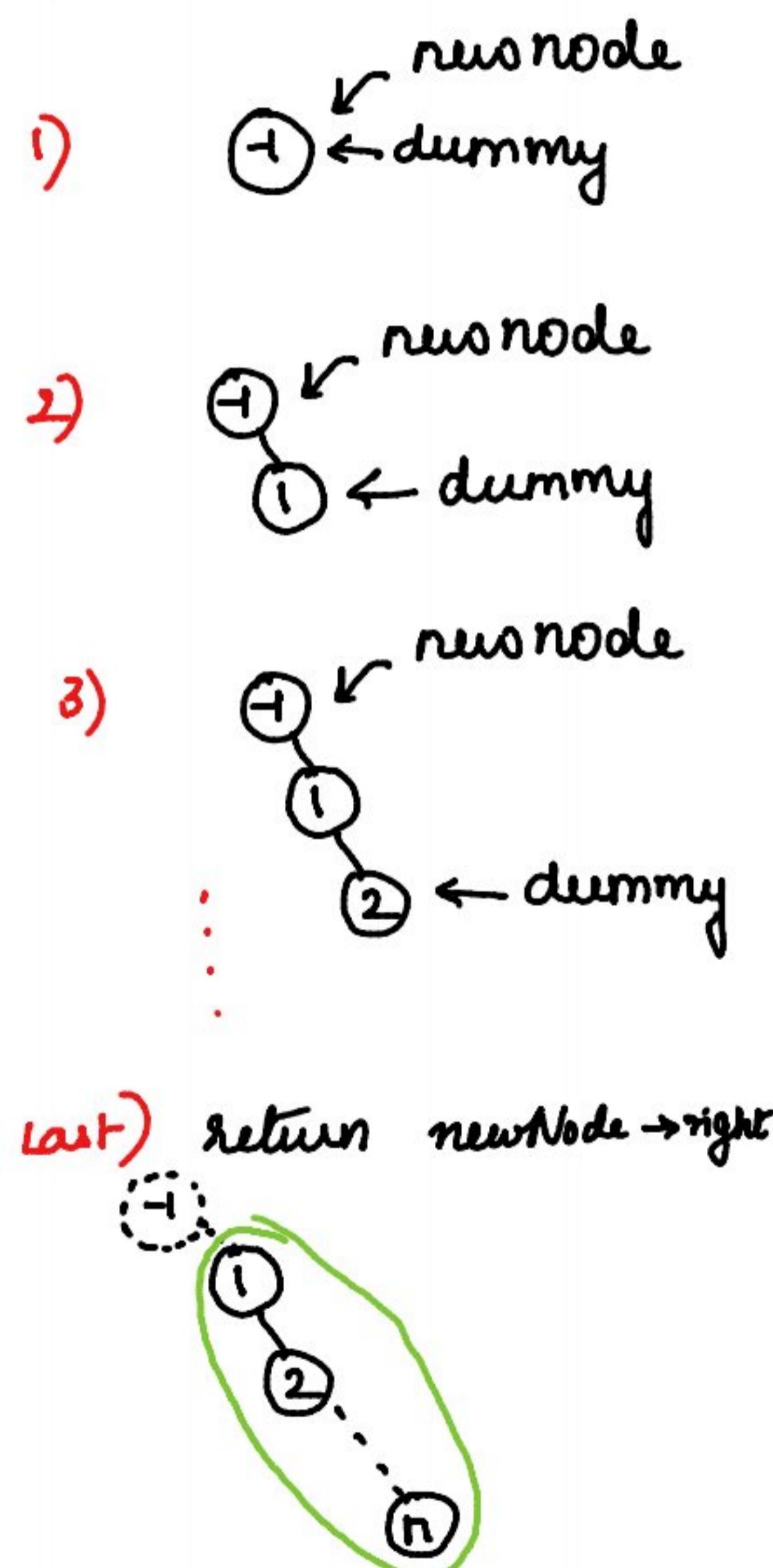
code

```

● ● ●
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

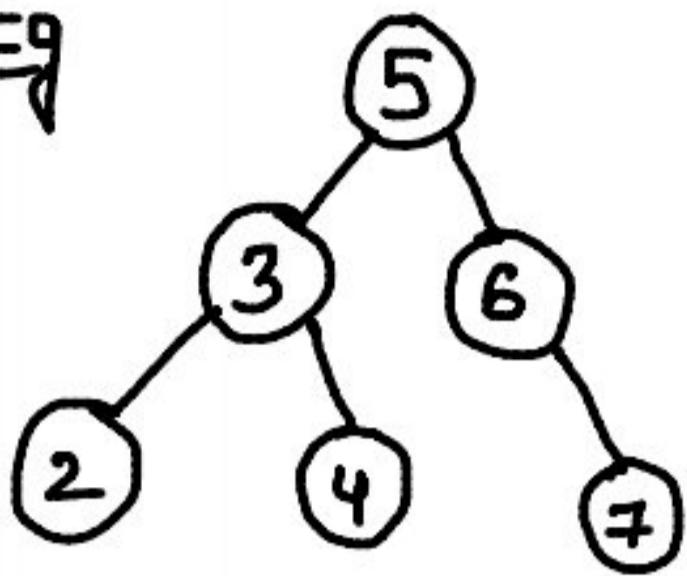


(32)

Two sum IV - Input is a BST

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

Ex



- ① Perform Inorder & store in array
- ② use 2-pointer approach

$$k = 9$$

$$\Rightarrow v = \begin{bmatrix} 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

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;
    }
};
  
```

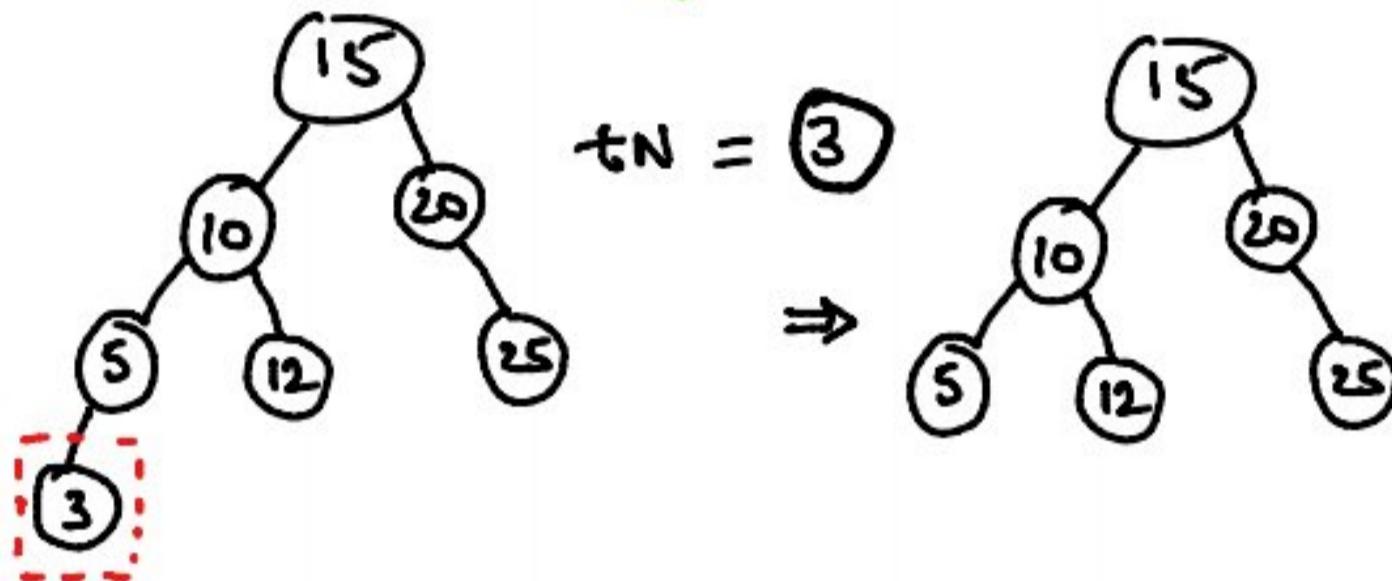
Tc → O(n)+O(n)
Sc → O(n)

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



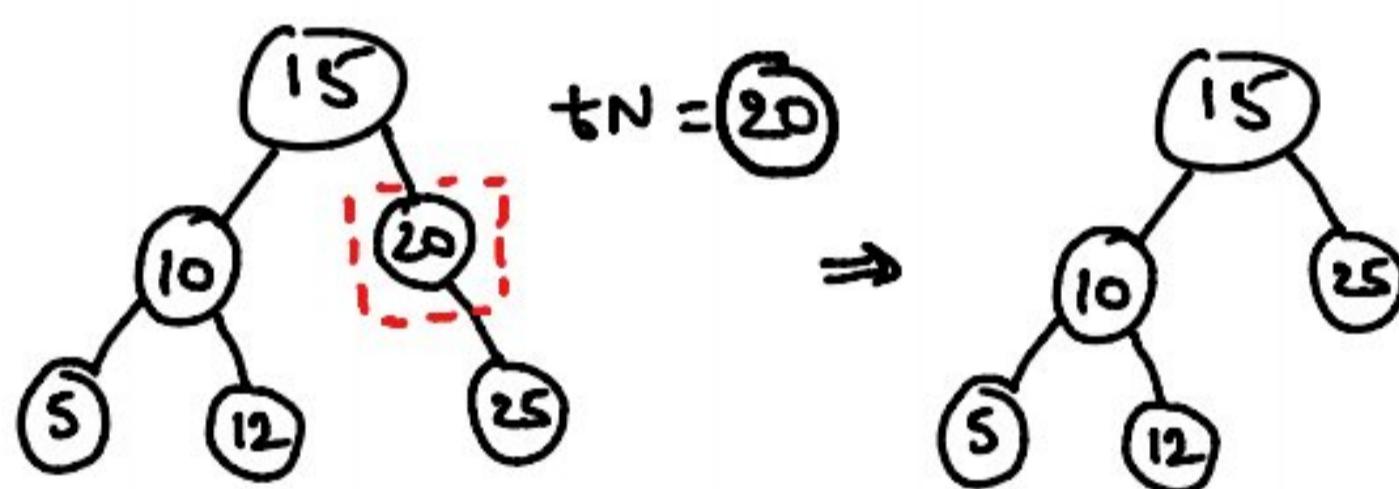
Tc →

Avg $\Rightarrow O(\log n)$

Worst $\Rightarrow O(n)$

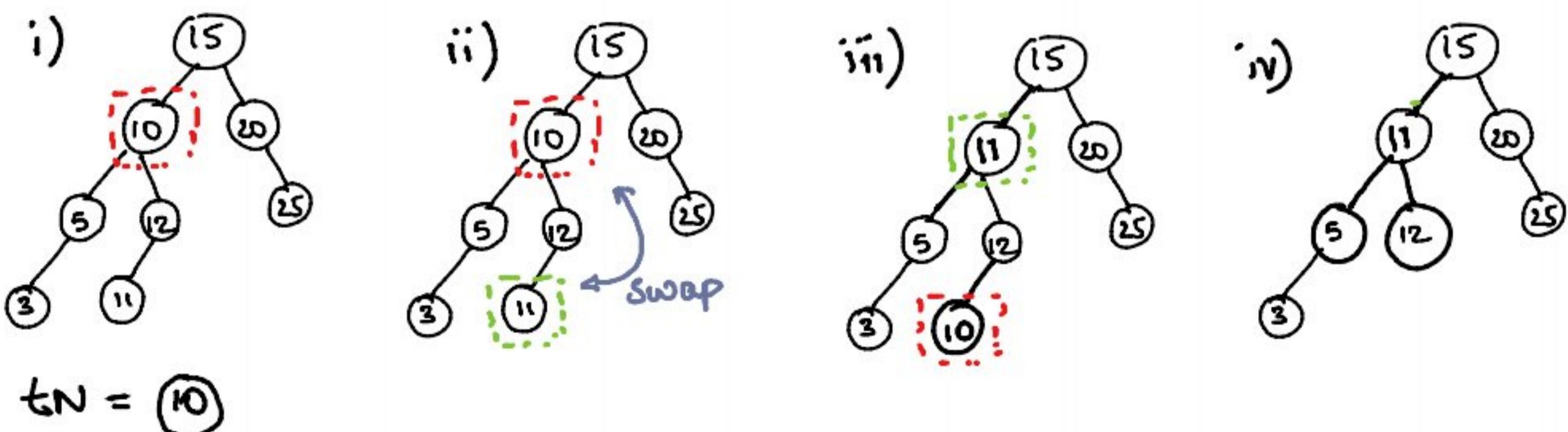
SC $\rightarrow O(h)$

- 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 ⁱⁿ value with target node & then delete it.



Code

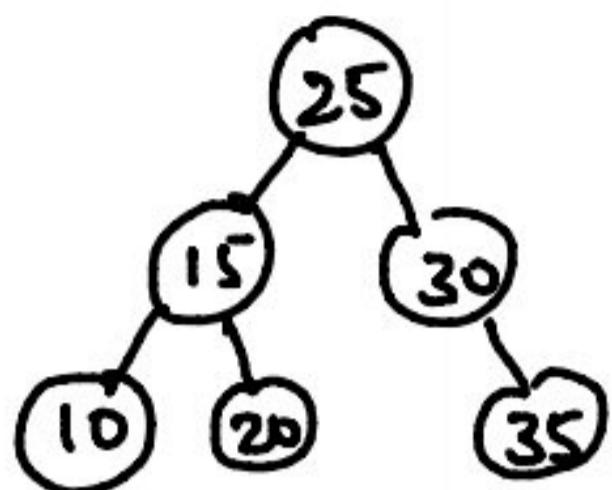
```
1 class Solution {
2 public:
3     TreeNode* findleftmostNode(TreeNode* root){
4         while(root->left!=NULL)
5             root = root->left;
6         return root;
7     }
8
9     TreeNode* deleteNode(TreeNode* root, int key) {
10
11         if(root==NULL)  return NULL;
12
13         if(root->val > 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         return root;
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);
    }

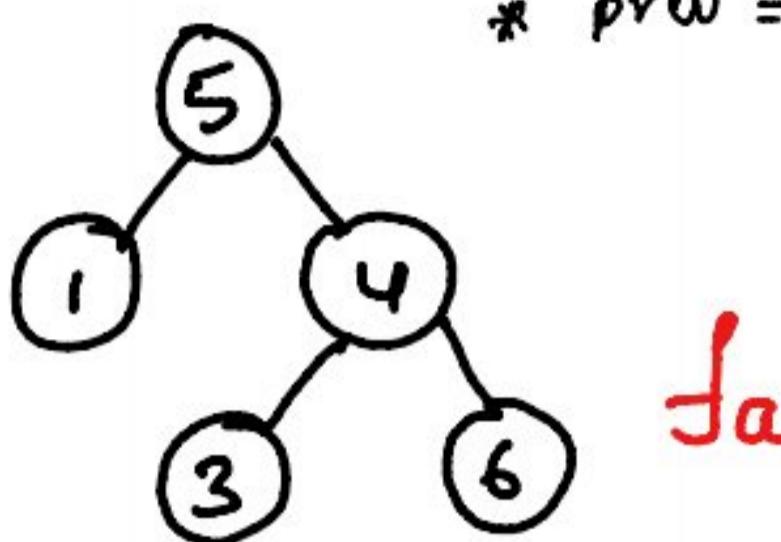
    Node * inOrderSuccessor(Node *root, Node *x)
    {
        vector<Node*> res;
        inorder(root, res);
        for(int i=0; i<res.size(); i++){
            if(res[i]==x && i<res.size()-1){
                return res[i+1];
            }
        }
        return NULL;
    }
};
```

D12

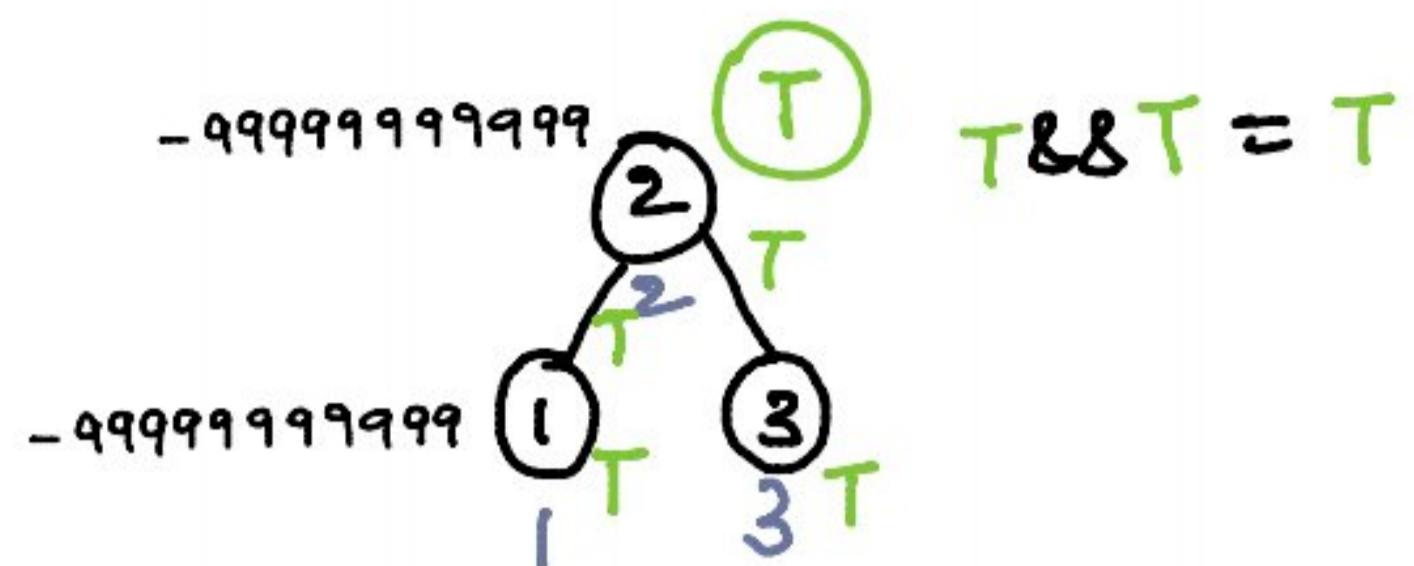
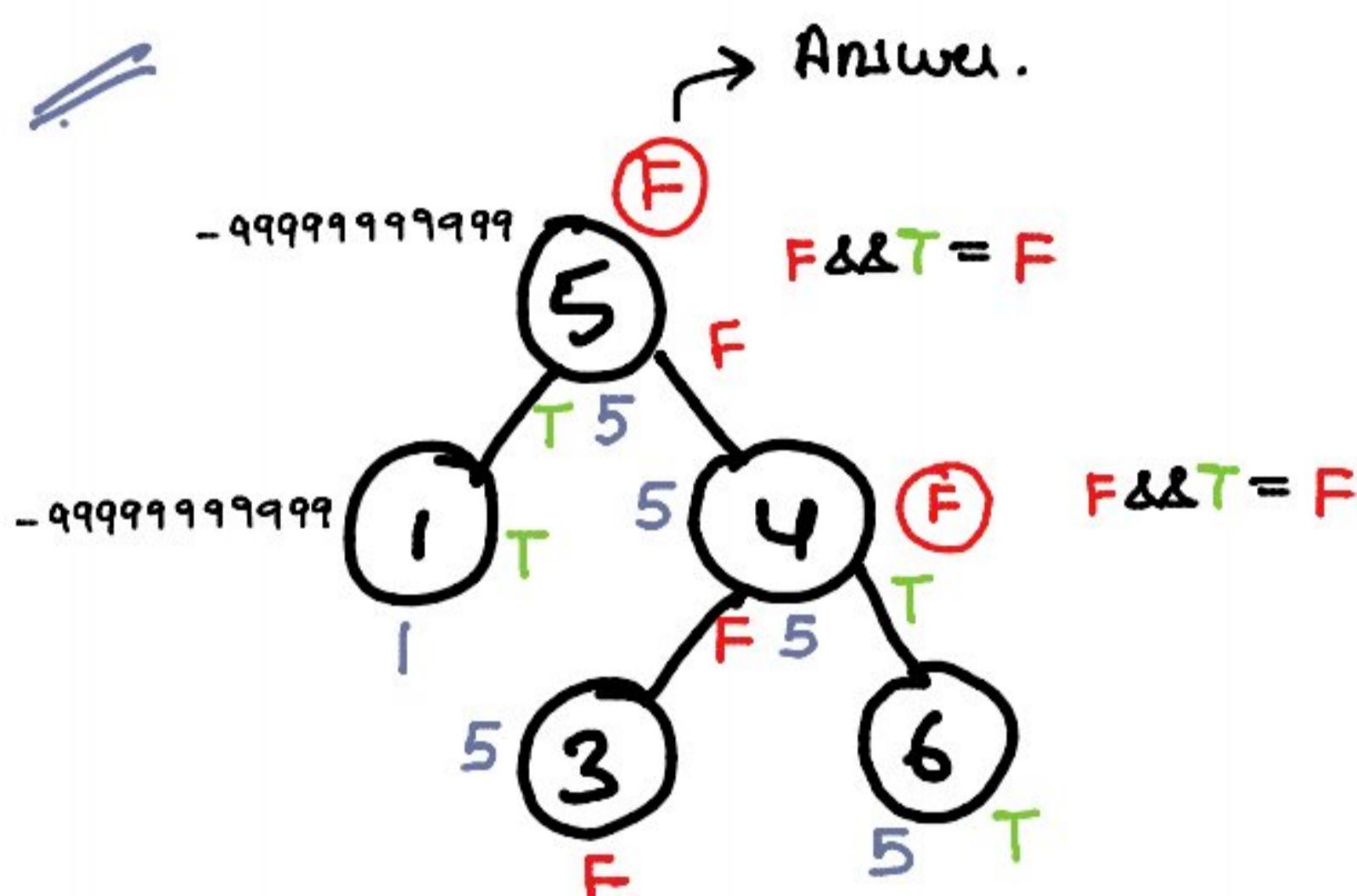
35 Validate BST

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

Eg



False



- Returns True on NULL nodes
- check for left subtree
- previous value gets updated before checking Right subtree & after checking left subtree
- if curVal \leq previous then return false
- return true if both LST & RST are BST

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;
    }

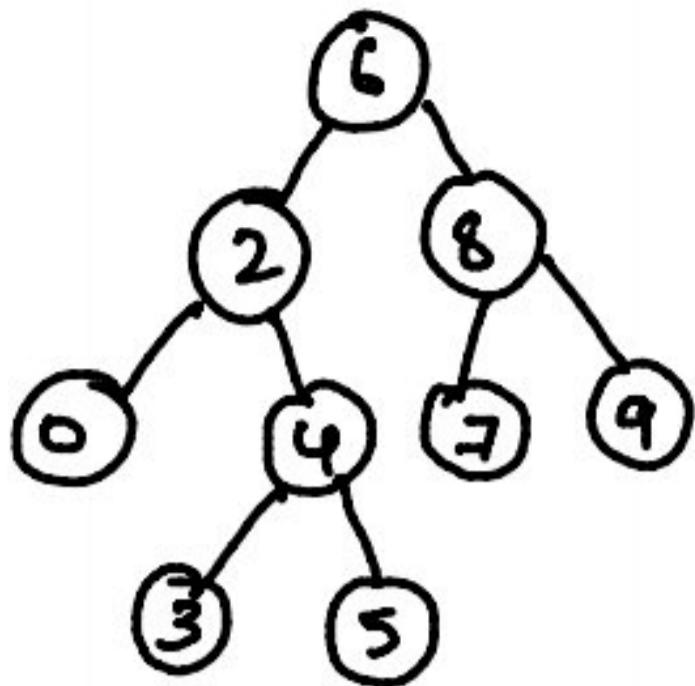
    bool isValidBST(TreeNode* root) {
        long int prev = -999999999999;
        return isBST(root, prev);
    }
};

```

36

LCA of BST →

Ex.

 $P=2, Q=8$

if $\text{currNode} > \text{both } P \& Q$
then LCA lies in LST

 if $\text{currNode} < \text{both } P \& Q$
then LCA lies in RST

 in every other case the currNode is
LCA as $P \& Q$ will be on

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

code

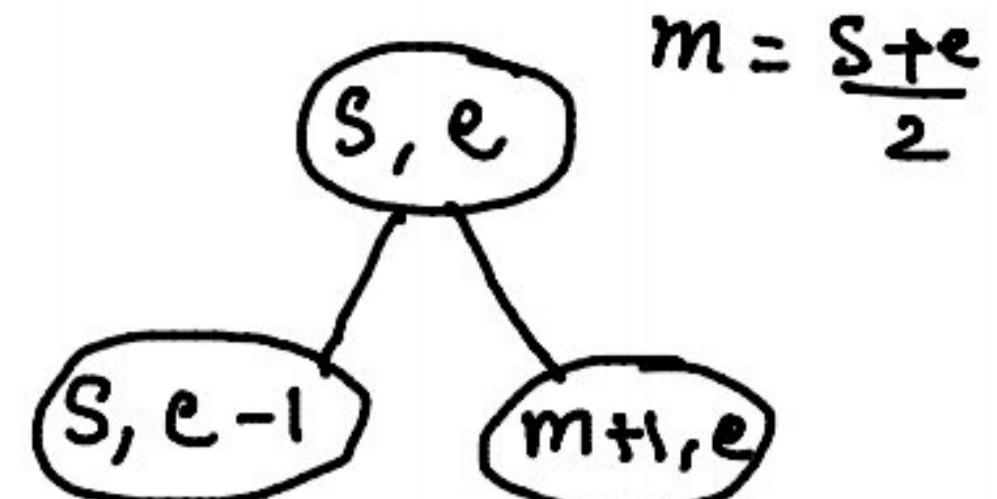
```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL) return NULL;

        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;
        }
    }
};
```

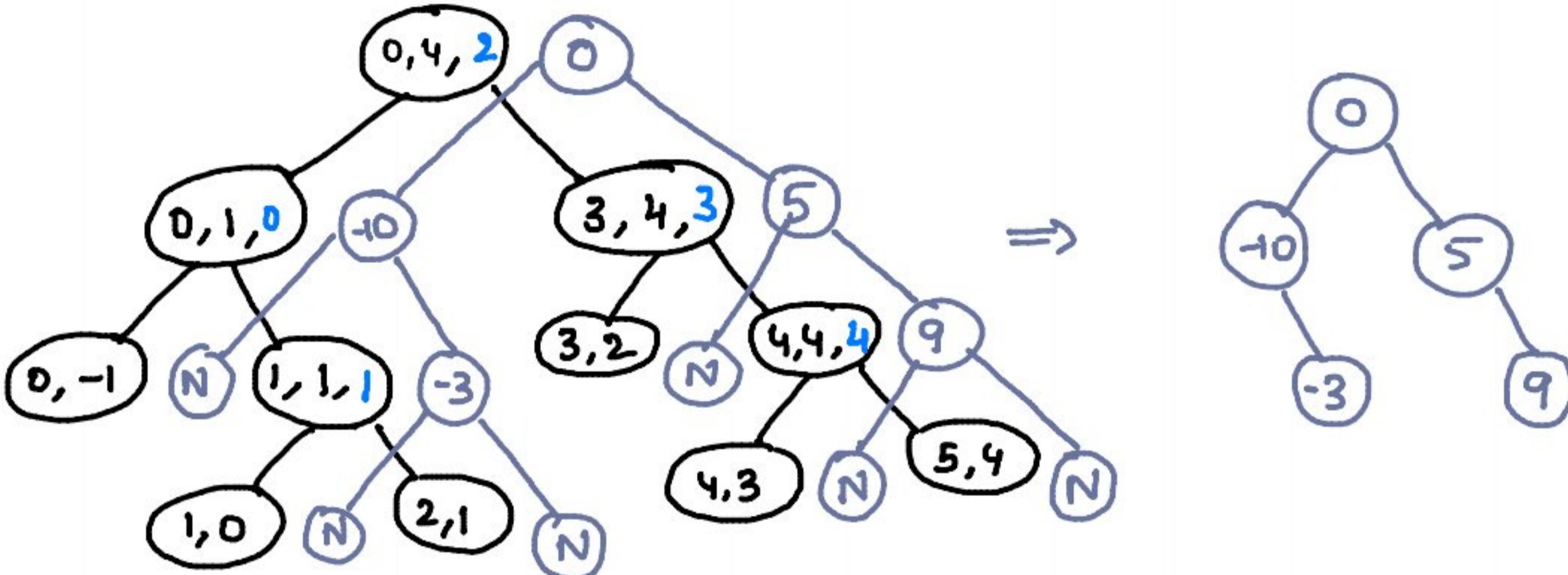
37 Convert Sorted array to BST

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);
    }
};
```

D13

(38) Construct Binary Tree from Pre & Inorder traversals

$\text{Pre} = [0, 1, 2, 3, 4]$
 $\text{pre} = [3, 9, 20, 15, 7]$
 $\text{in} = [9, 3, 15, 20, 7]$

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

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

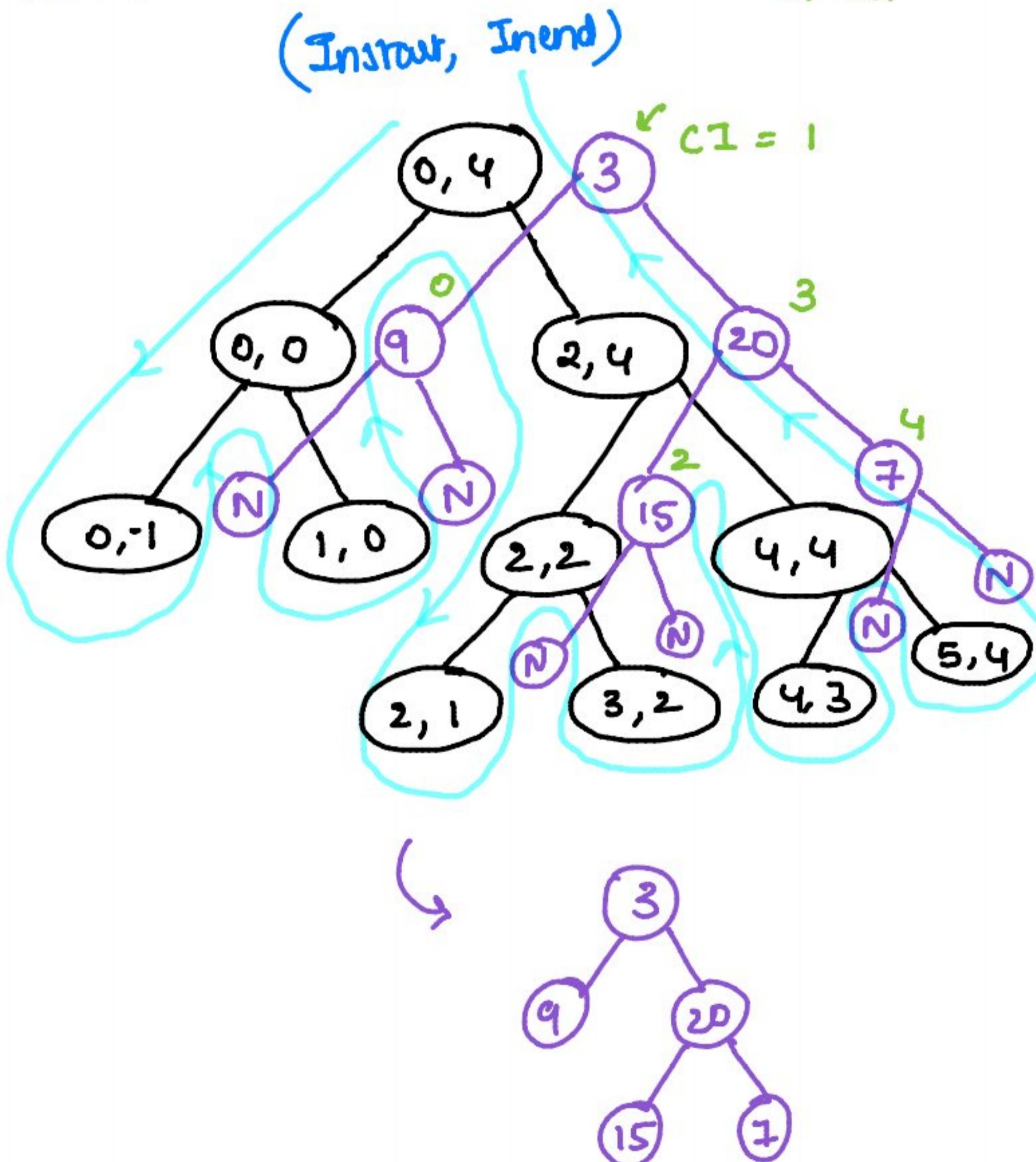
i.e. $3 \rightarrow [\begin{matrix} \text{LST} & \text{CI} & \text{RST} \\ \boxed{9}, & 3, & \boxed{15, 20, 7} \\ 0, & 1, & 2 \ 3 \ 4 \end{matrix}]$



$$\text{LST} = (\text{instar}, \text{ci}-1)$$

$$\text{RST} = (\text{ci}+1, \text{inend})$$

$\text{CI} = \text{index of pre[0]}$
 in In



$\text{Pre} = [3, 9, 20, 15, 7]$
 $\text{in} = [9, 3, 15, 20, 7]$

- 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 }
```

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

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]

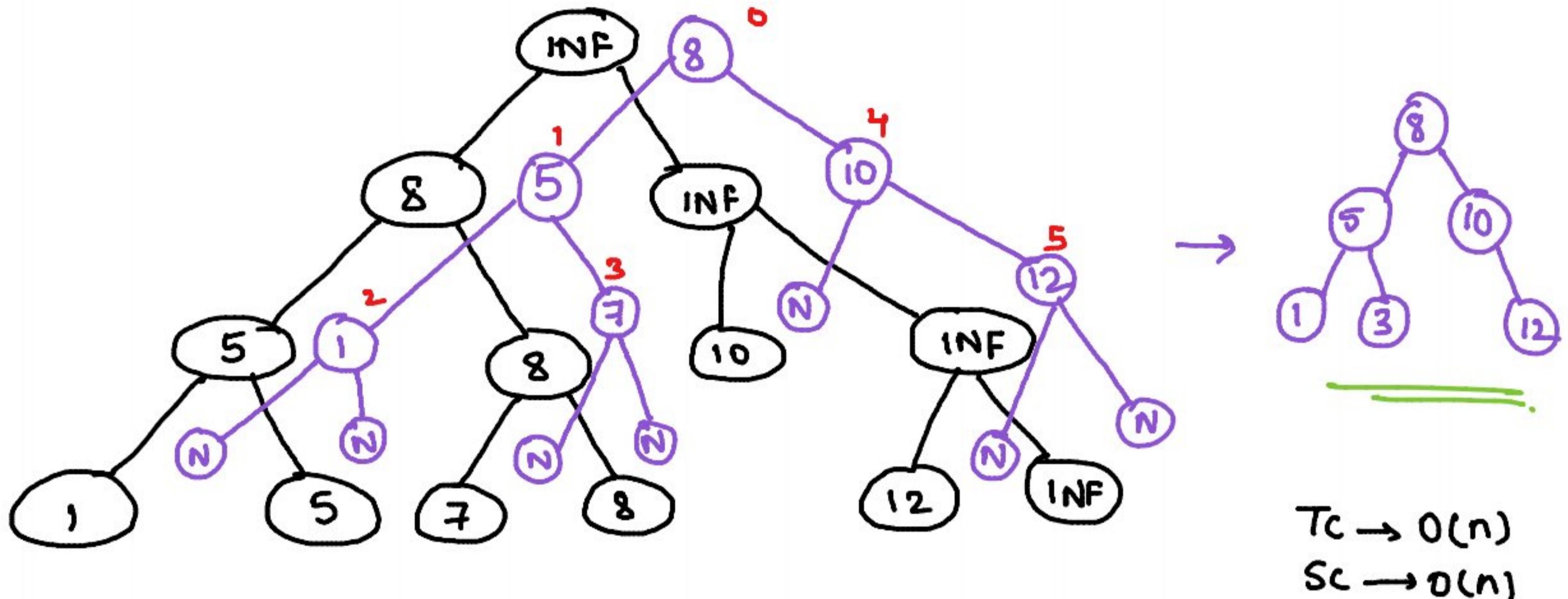
$T_C \rightarrow O(n \log n)$ (due to sorting)

Approach 1 → Sort given Preorder to get Inorder, now similar to problem 38.

Approach 2 →

[8, 5, 1, 7, 10, 12]
0 1 2 3 4 5

Boundary of LST \rightarrow val
RST \rightarrow boundVal \rightarrow initially (INF)



Code →

```

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

```

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