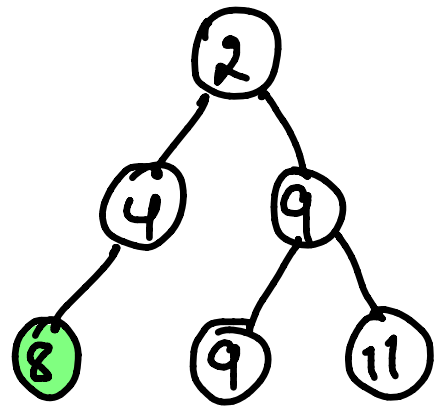
Trees - Part 2

- Karun Karthik

Contents

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

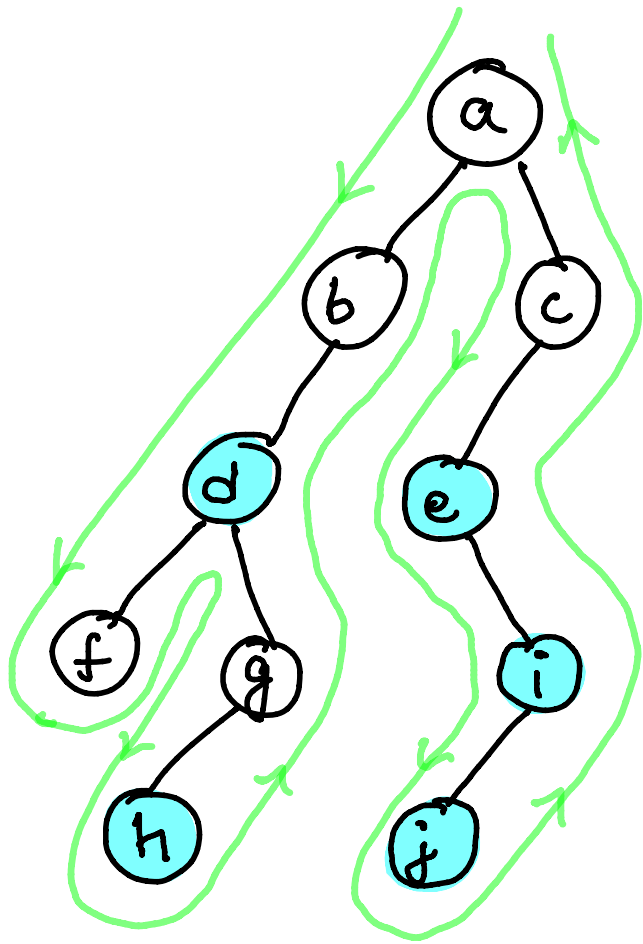
D7 (21) Print all nodes that donot have any siblings



Eg

Result →

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

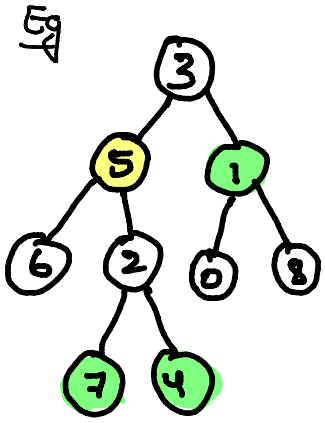
Code →

```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

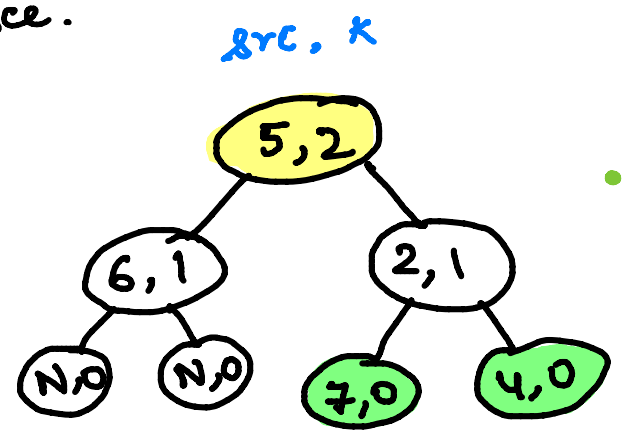
given a source node, find all the nodes that are at a distance of K units.

Eg



Source = 5
K = 2

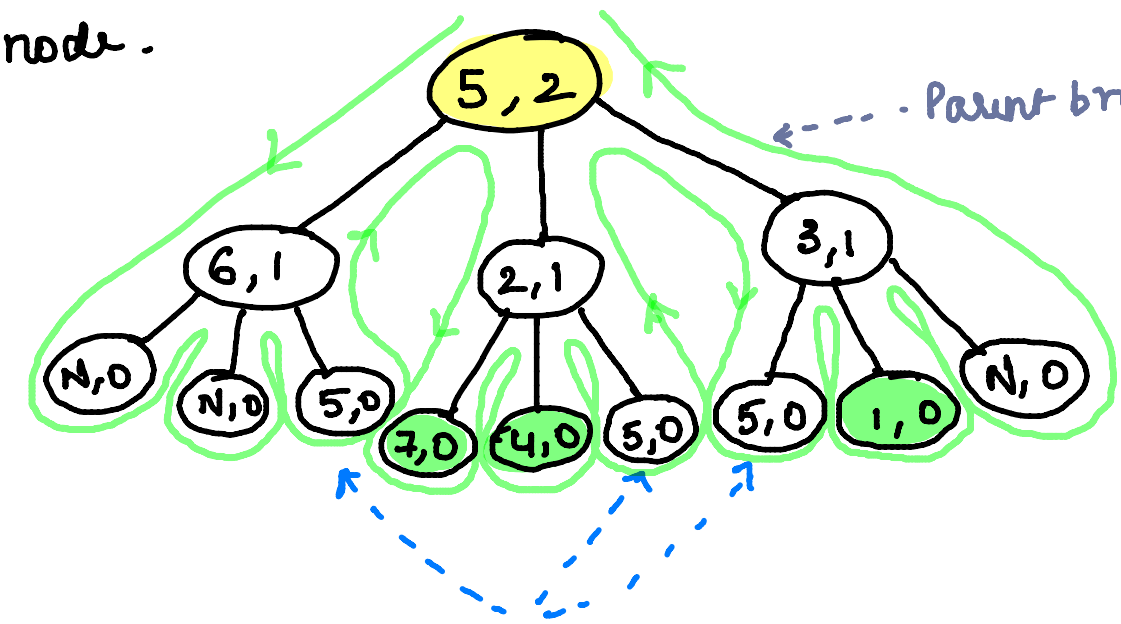
① consider nodes in downward direction of source.



- 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

populating traversal

Tc → O(n) + O(n)

Sc → O(n) + O(n) + O(x)

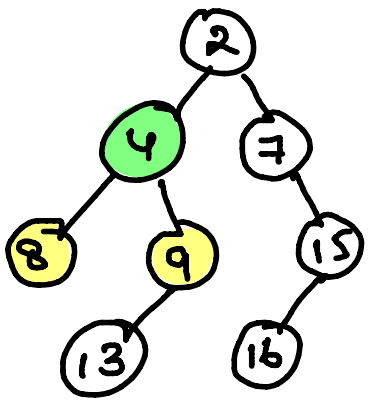
x → result

Code.

```cpp
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 $\bar{n}_1 = [8, 4, 2]$, $\bar{n}_2 = [9, 4, 2]$ → 4

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

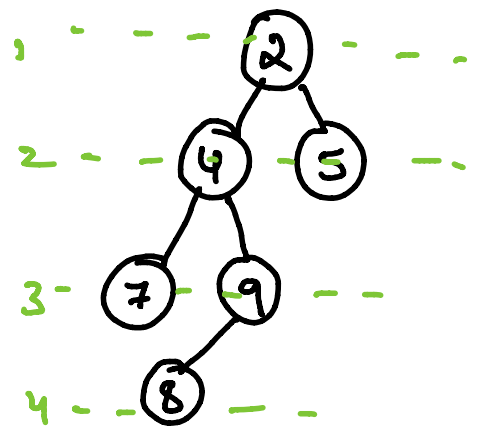
Code

```
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.



⇒ [[2], [4,5], [7,9], [8]]

TC → O(n)

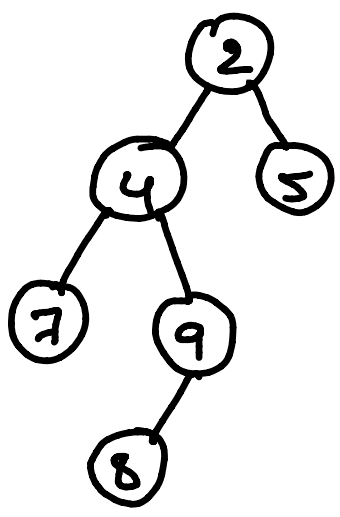
SC → O(n)

→ To find level order traversal use queue. → FIFO data structure

Inserting → Rear

Removing → Front

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



Queue

2

4, 5

5, 7, 9

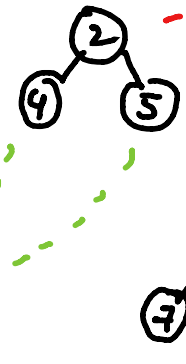
7, 9

9

8

empty →

Result



[ ]

[2]

[2, 4]

[2, 4, 5]

[2, 4, 5, 7]

[2, 4, 5, 7, 9]

Answer → [2, 4, 5, 7, 9, 8]

① for every node, enqueue.

② while dequeue, enqueue its branches.

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

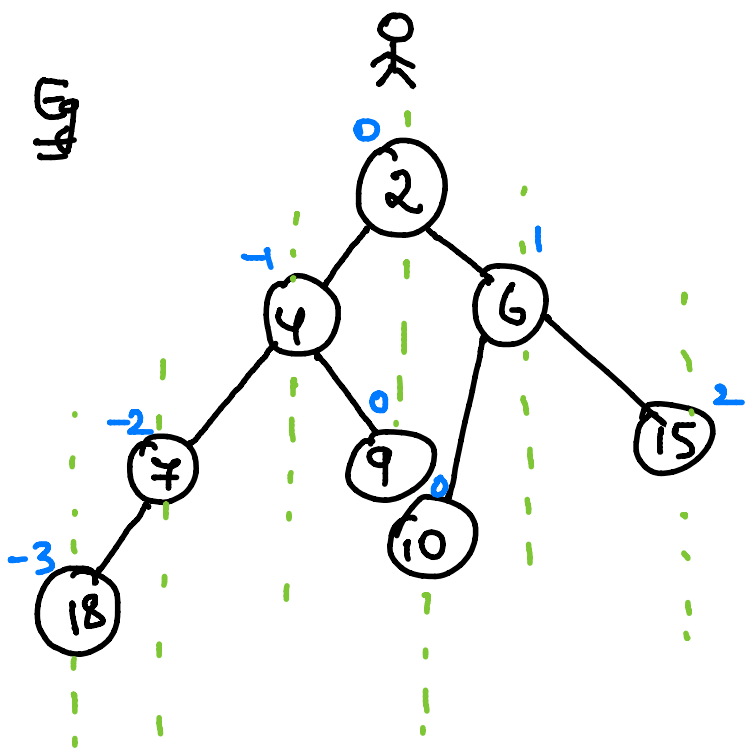
TC → O(n)

SC → O(n)

code →

```cpp
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

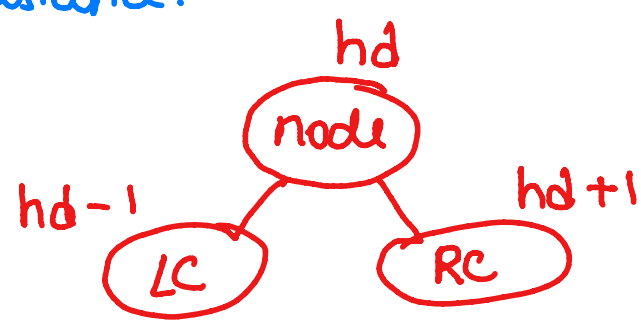


Left to Right

↳ [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

```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 →

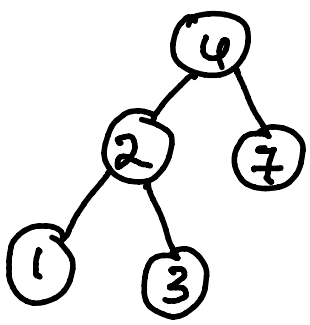
```cpp
class Solution {
  public:
    vector <int> bottomView(Node *root) {
        vector<int> res;
        if(root==NULL) return res;

        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 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

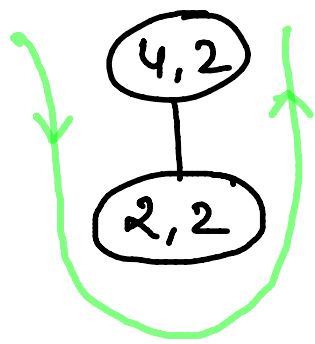
① LC < node < RC
② LC ≤ node < RC
③ LC < node ≤ RC

(28) Search in a BST



Val = 2

⇒ return the subtree with given value.



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

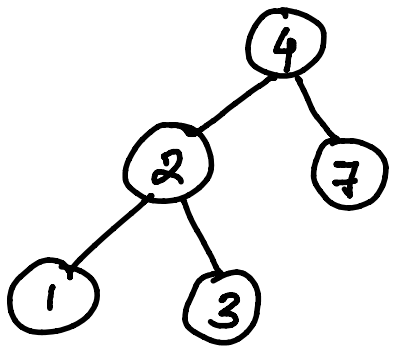
TC → O(log₂n) avg , O(n) worst

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.

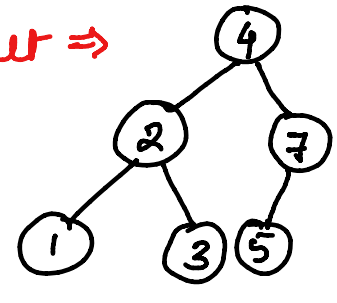
1) root 4 5 — As 5 > 4, go to RST

2) 4 — 5 — 7 — 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

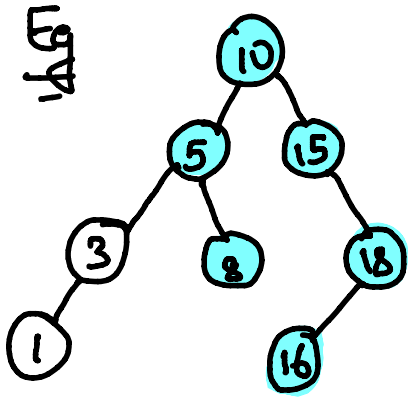
SC → O(1)

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;
    }
};
```

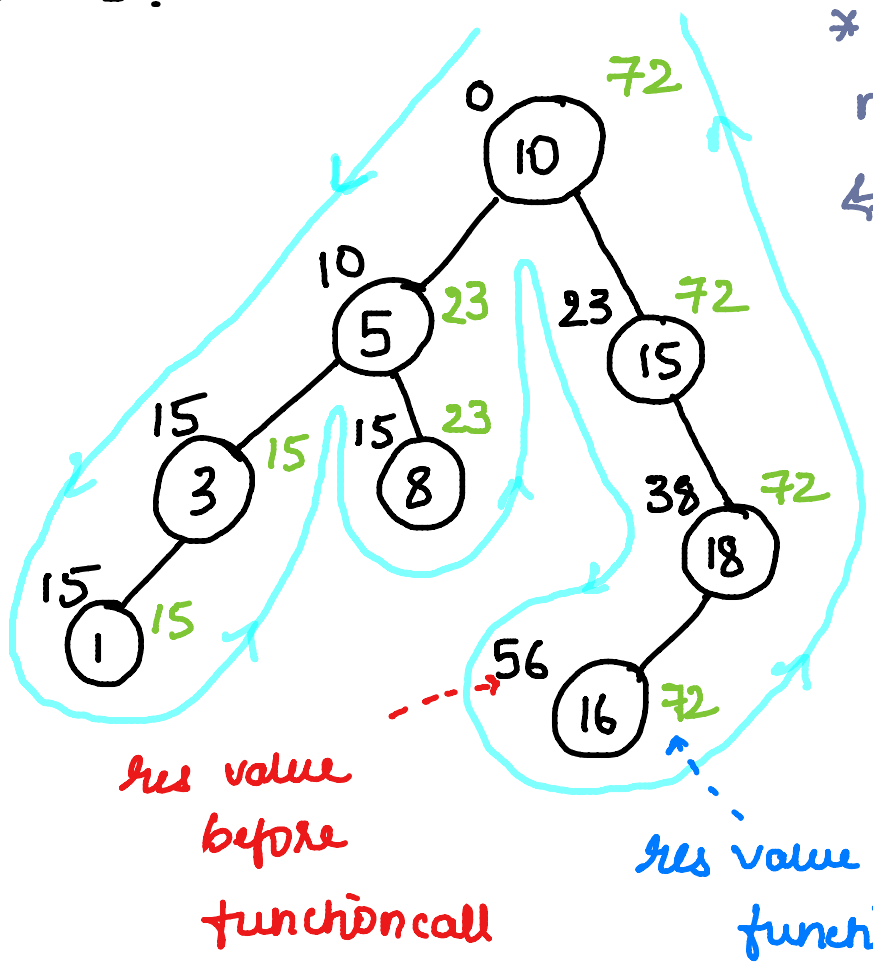
(30) Range Sum of BST

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



range → [5, 18]

sum = 72



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

TC → O(n)

SC → O(n)

res value before function call

res value after function call.

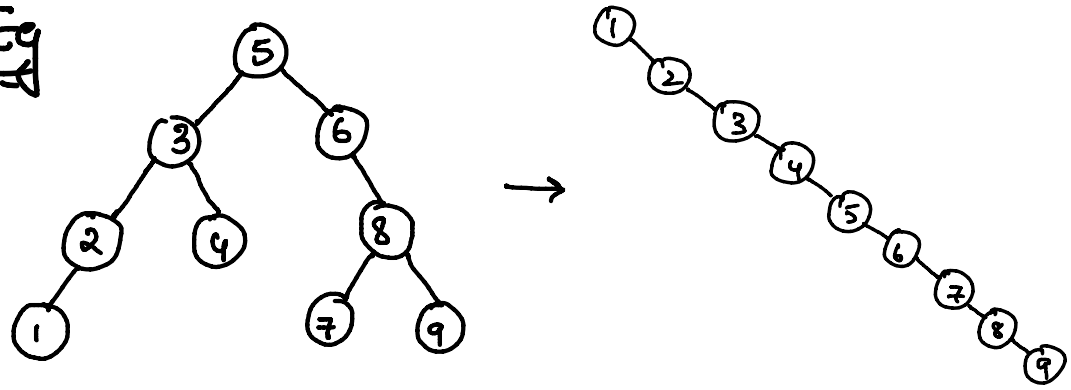
Code →

```
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

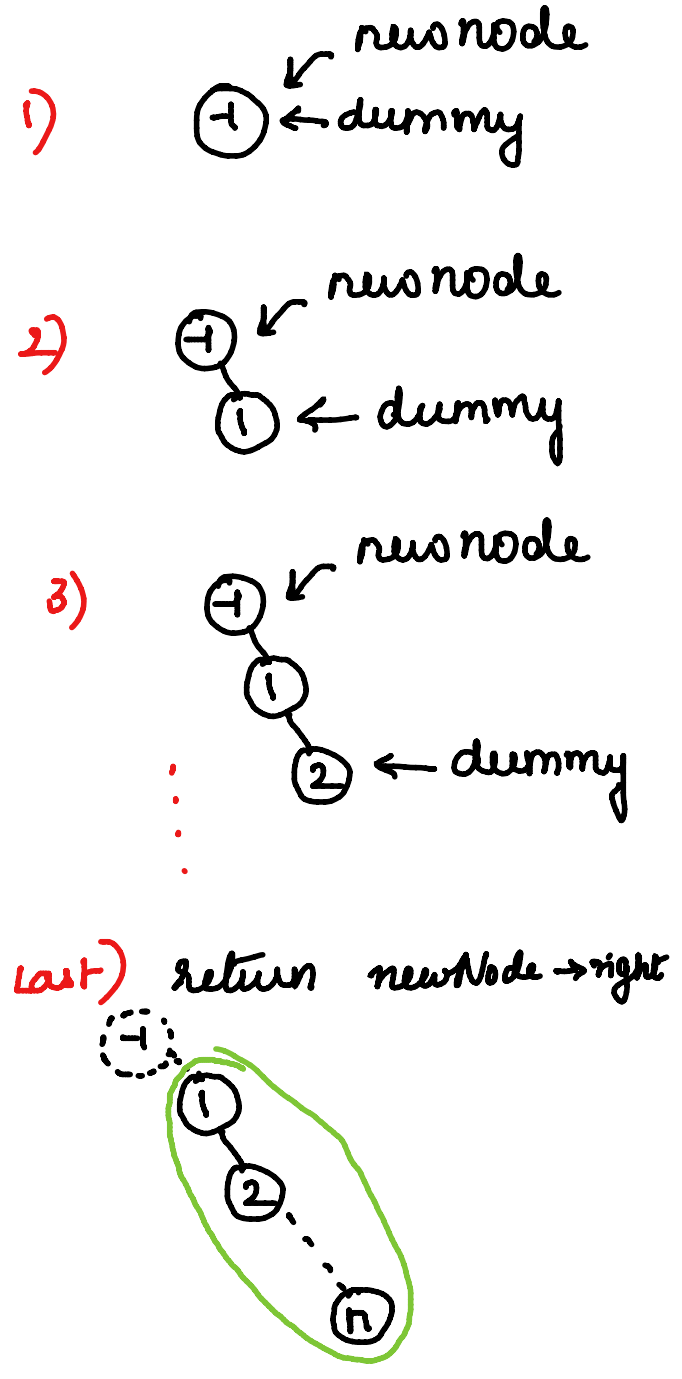


1. Perform inorder traversal.
2. Create a skewed tree using elements in inorder traversal.

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 };
```

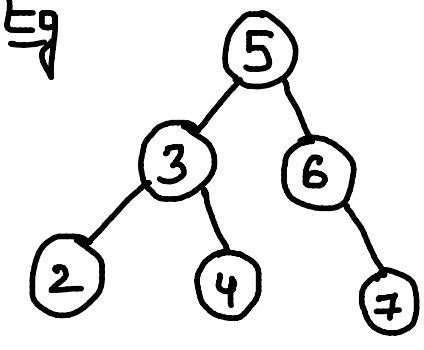
Lines 16-20



(32) Two sum IV - Input is a BST

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

Eg



k = 9

① Perform Inorder & store in array

② use 2-pointer approach

⇒ v =

0	1	2	3	4	5
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)

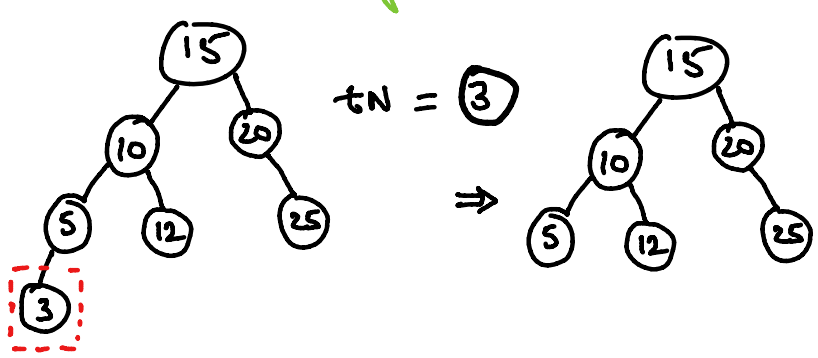
worst ⇒ O(n)

SC → O(h)

Cases →

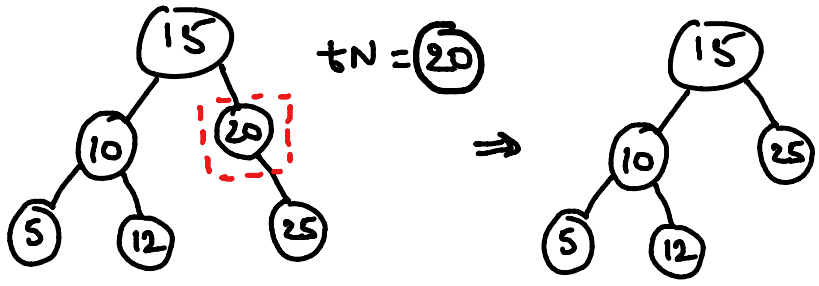
(1) If target node is leaf →

then simply delete it



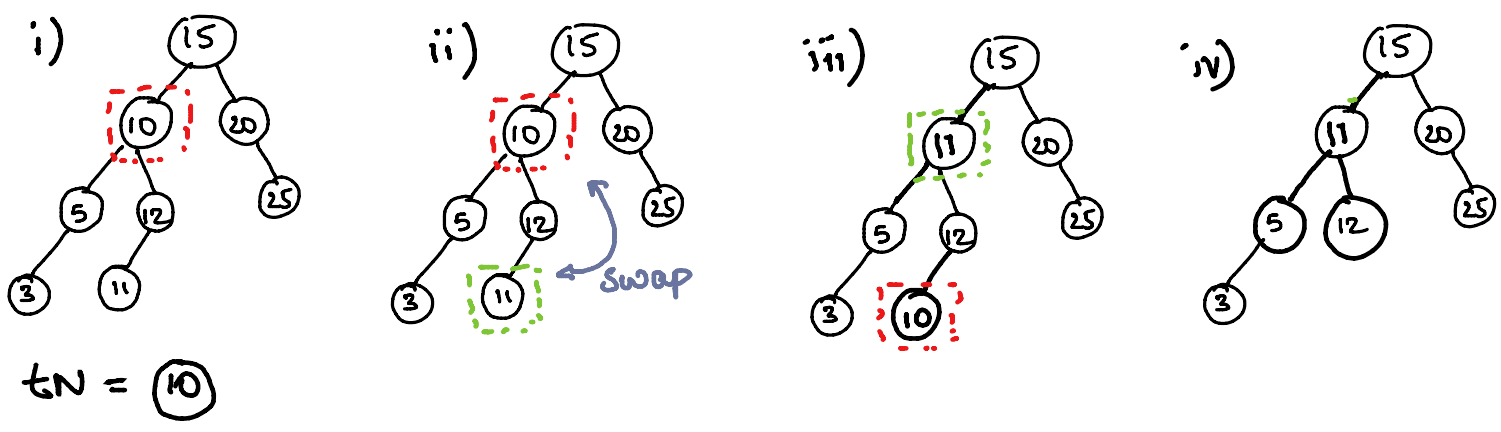
(2) If target node has 1 child →

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

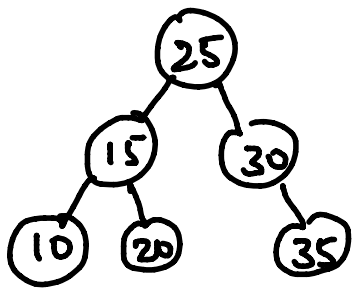
```cpp
class Solution {
public:
    TreeNode* findleftmostNode(TreeNode* root){
        while(root->left!=NULL)
            root = root->left;
        return root;
    }

    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;
    }
};
```

(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 o/p → 20

n = 35 o/p → null.

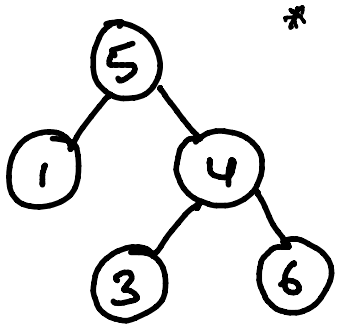
Code →

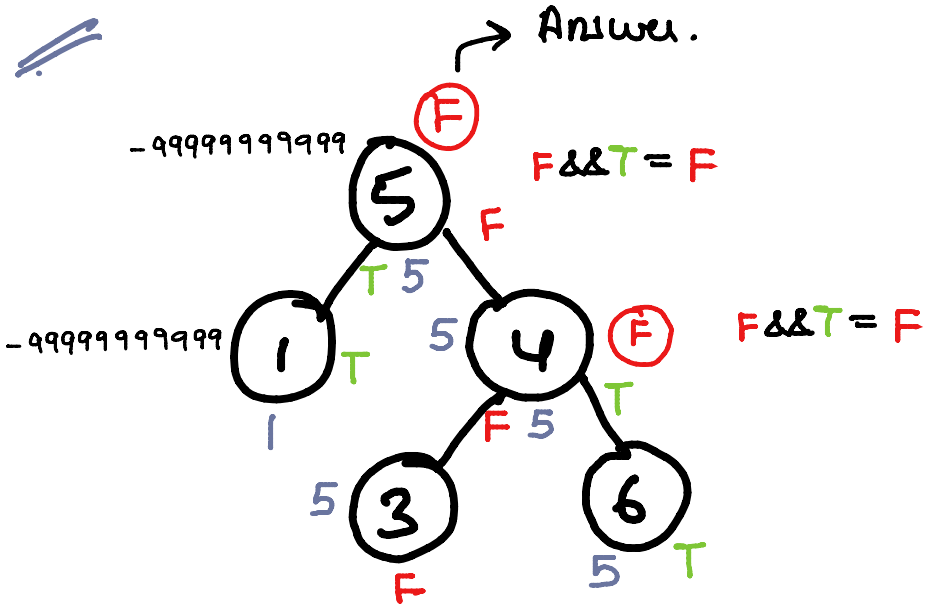
```
class Solution{
  public:

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

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

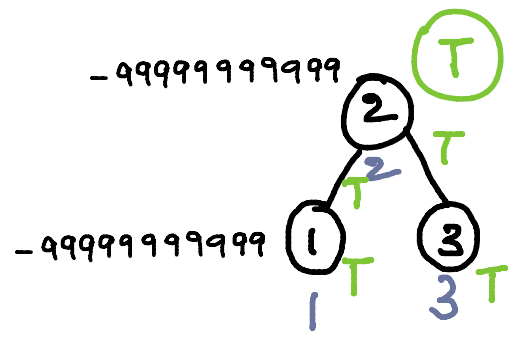
D12 (35) **Validate BST** Given a root node, return true if it is valid BST

* Every value should be less than previous one in Inorder Traversal

Eg







→ Return True on NULL nodes

→ Check for left subtree

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

→ if curVal <= Previous then return false

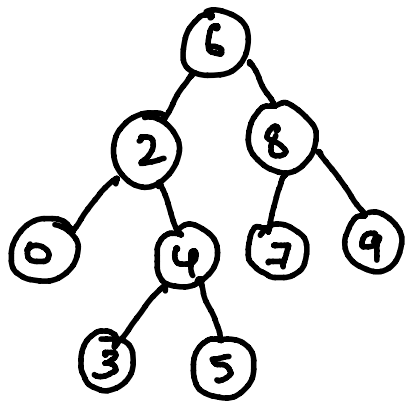
→ return true if both LST & RST are BST

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 = -99999999999;
        return isBST(root, prev);
    }
};
```

(36) LCA of BST →

Eg.



P = 2, q = 8

if currNode > both p & q
then LCA lies in LST

if currNode < both p & q
then LCA lies in RST

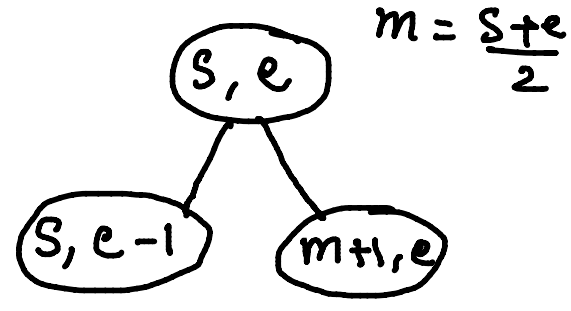
in every other case the currNode is LCA as p & q will be on

	worst	Avg
TC →	O(n)	O(logn)
SC →	O(n)	

code

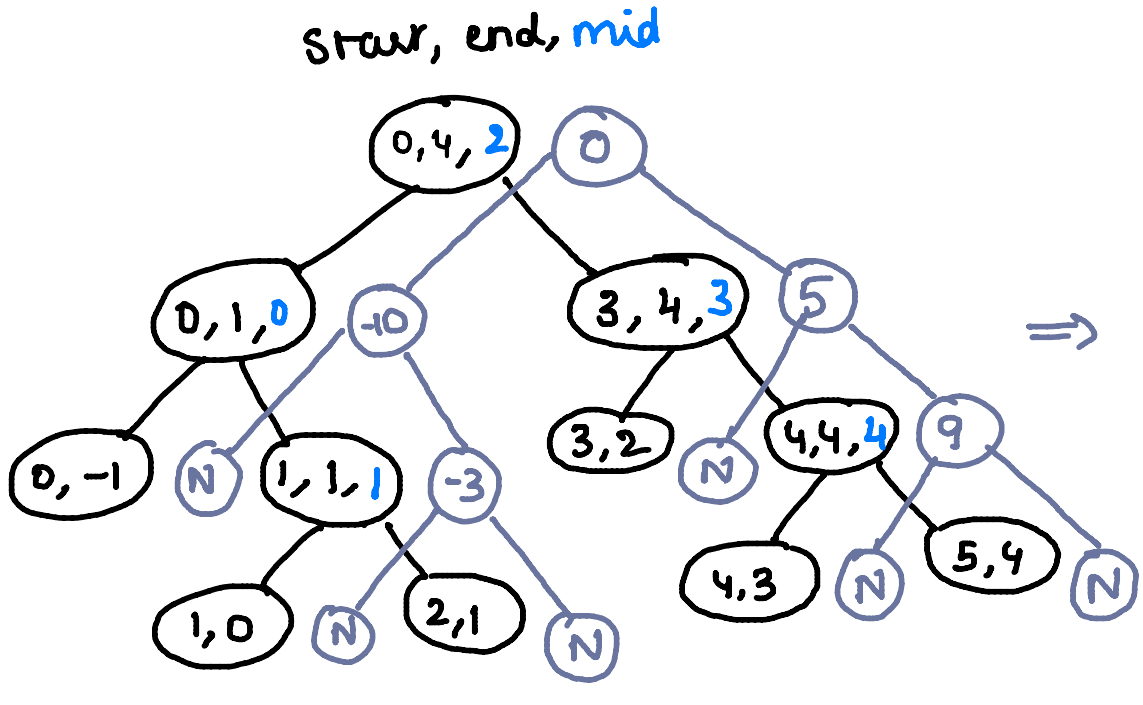
```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL)  return NULL;

        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]
0 1 2 3 4



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 BinaryTree from Pre & Inorder traversal

Eg

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

Tc → O(n²)

Sc → O(n)

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

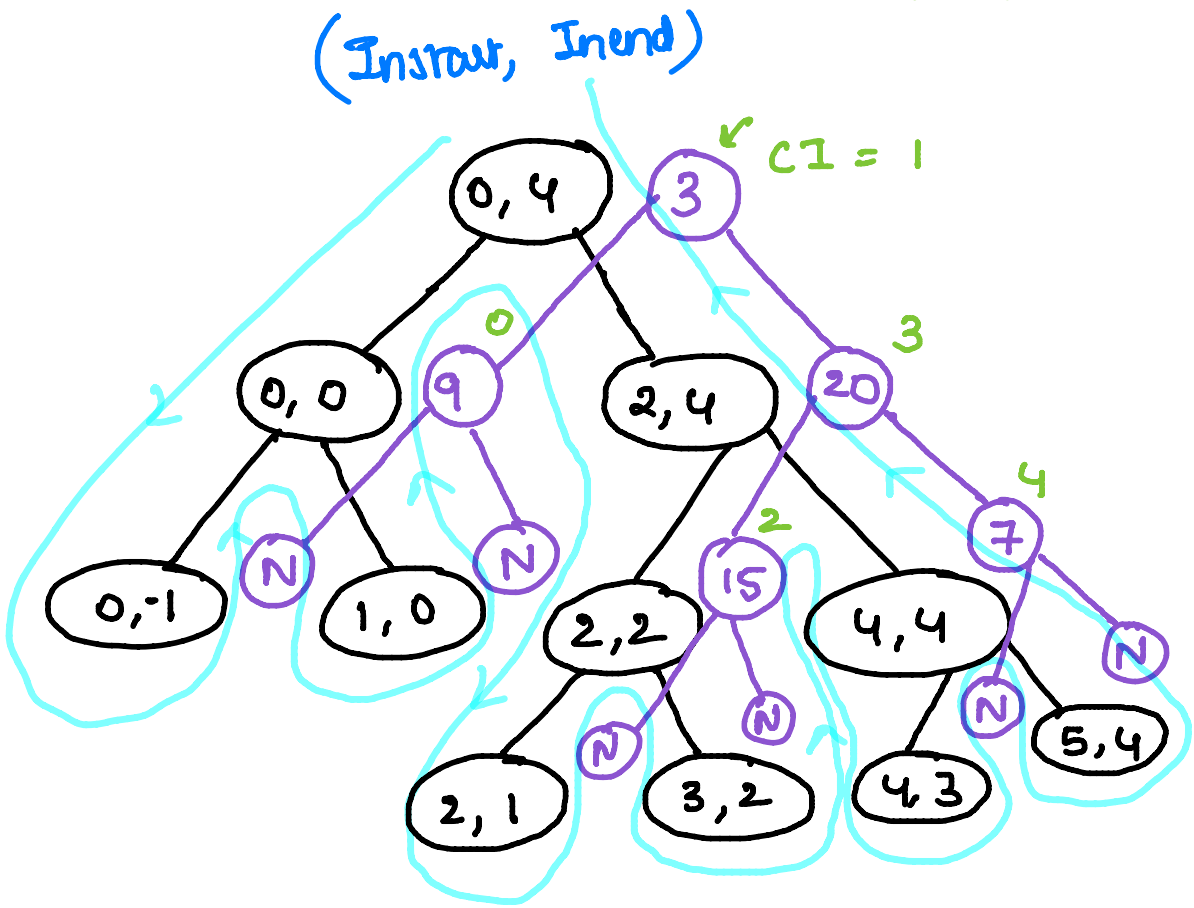
ie 3 → [LST: 9 (idx 0) | CI: 3 (idx 1) | RST: 15, 20, 7 (idx 2, 3, 4)]

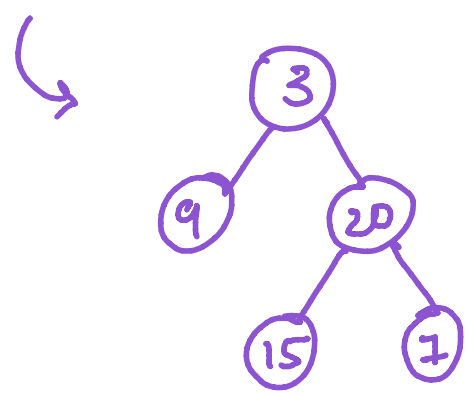
LST = (instart, CI-1)

RST = (CI+1, inend)

CI = index of pre[0] in In

(Instart, Inend)





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

↑ (pre) ↑ (in)

1

① for preorder index = 0, inorder boundry = [0, 4]

② find root value in Inorder array & it's index is CurrIndex

③ if instart > CI-1 or CI+1 < inend return NULL

To reduce Tc we can use hashtable to find indexing

Tc → O(n)

Sc → O(n)+O(n)

code →

```cpp
class Solution {
public:
    TreeNode* constructTree(vector<int>& preorder, unordered_map<int, int> &mp,
    int start, int end, int &preIdx ){

        if(start>end)   return NULL;
        TreeNode* root = new TreeNode(preorder[preIdx]);

        // find currIndex as per inorder array
        int currIdx = mp[preorder[preIdx]];
        // increment preIdx to find next root
        preIdx++;

        // recursively call LST & RST
        root->left = constructTree(preorder, mp, start, currIdx-1, preIdx);
        root->right = constructTree(preorder, mp, currIdx+1, end, preIdx);
        return root;
    }

    unordered_map<int,int> populate(vector<int>&inorder){
        unordered_map<int,int> mp;
        for(int i=0; i<inorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return mp;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int,int> mp = populate(inorder);
        int preIdx = 0;
        return constructTree(preorder, mp, 0, inorder.size()-1, preIdx);
    }
};

```

(39) Construct BinaryTree from In & Postorder traversal

Intuition is same as previous program, only changes are

→ traverse from last element in postorder array

→ process RST & then go for LST

Code →

```cpp
class Solution {
public:

    TreeNode* constructTree(vector<int>& postorder, unordered_map<int, int> &mp,
    int start, int end, int &postIdx ){

        if(start>end)   return NULL;
        TreeNode* root = new TreeNode(postorder[postIdx]);

        // find currIndex as per inorder array
        int currIdx = mp[postorder[postIdx]];
        postIdx--;

        // recursively call RST & LST
        root->right = constructTree(postorder, mp, currIdx+1, end, postIdx);
        root->left = constructTree(postorder, mp, start, currIdx-1, postIdx);
        return root;
    }

    unordered_map<int,int> populate(vector<int>&inorder){
        unordered_map<int,int> mp;
        for(int i=0; i<inorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return mp;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int,int> mp = populate(inorder);
        int postIdx = postorder.size()-1;
        return constructTree(postorder, mp, 0, inorder.size()-1, postIdx);
    }
};
```

(40) Construct BST from Preorder traversal

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

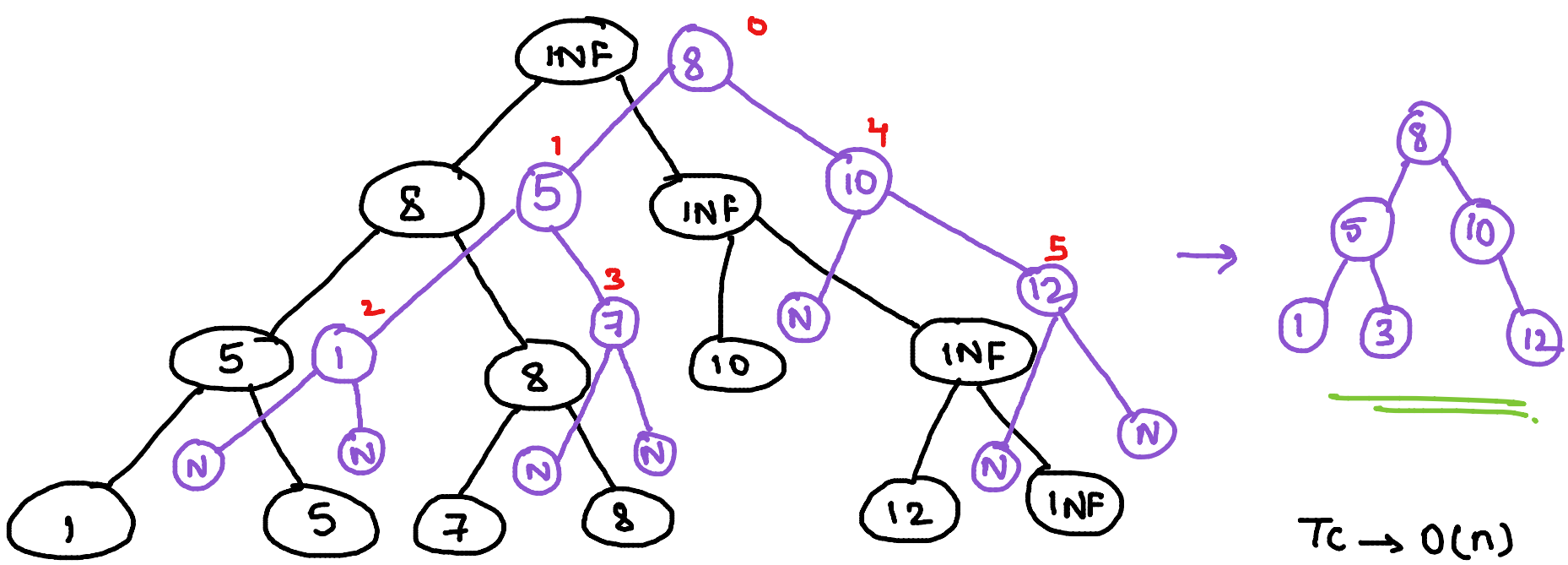
Tc → O(nlogn) (due to sorting)

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 → Val
RST → boundVal → 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
```

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

Follow Karun Karthik for more!

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