Binary Trees + (BST)

~ Arbind

Trees

Binary Tree

```
struct Node {
  int data;
  Node * left;
  Node *       right;
  Node(int val){
   data = val;
   left = right = NULL;
  }
```

```
main() {
struct Node* root =
      new Node(1);
root → left = new Node(2);
root → right = new Node(3);
}
```

Traversal Techniques (BFS/DFS)

DFS

Inorder Traversal (Left root right)

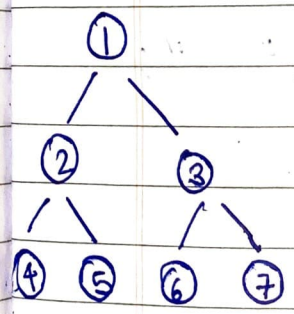
4 2 5 1 6 3 7

Preorder Traversal (Root Left right)

1 2 4 5 3 6 7

Postorder Traversal (Left right Root)

4 5 2 6 7 3 1



BFS (Level)

1 2 3 4 5 6 7

Pre-order Traversal

TC → O(N)
SC → O(N)
↳ auxiliary space

```
void preorder(node)
{
  if (node == NULL)
      return;

  cout << node → data << " ";
  preorder (node → left);
  preorder (node → right);
}
```

Inorder Traversal

```
void inorder (node)
{
  if (node == NULL)
      return;

  inorder (node → left);
  cout << node → data << " ";
  inorder (node → right);
}
```

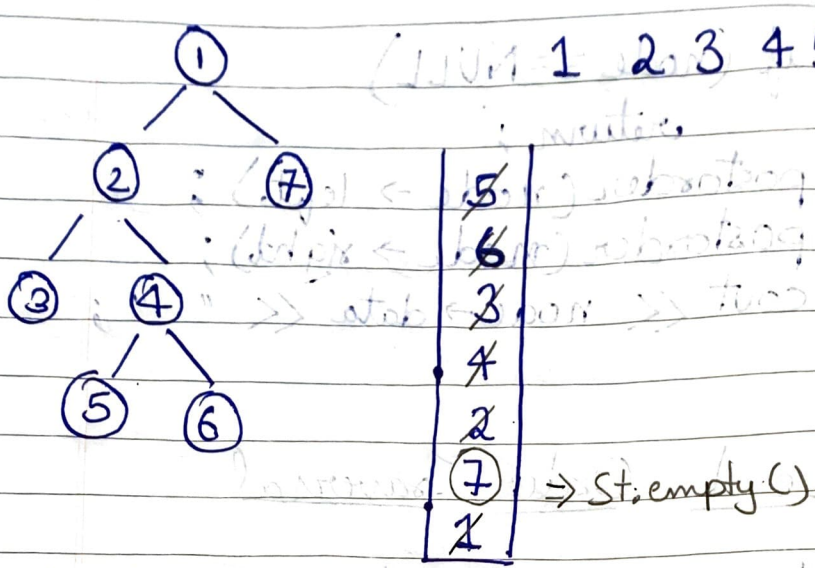
Post-order Traversal

```
void postorder (node)
{
  if (node == NULL)
      return;
  postorder (node -> left);
  postorder (node -> right);
  cout << node -> data << " ";
}
```

Level Order Traversal

```
vector<vector<int>> levelOrder(TreeNode * root){
    vector <vector<int>> ans;
    if(root==NULL)
        return ans;
    queue <TreeNode*> q;
    q.push(root);
    while(!q.empty()){
        int size = q.size();
        vector <int> level;
        for(int i=0; i<size; i++){
            TreeNode * node = q.front();
            q.pop();
            level.push_back( node->val);
            if(node->left) q.push(node->left);
            if (node->right)q.push(node->right);
        }
    ans.push_back(level) } return ans; }
```

Iterative Preorder Traversal
(Root Left Right)

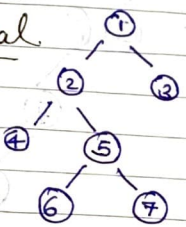


1 2 3 4 5 6 7

```
public:
    vector int<> preorder (TreeNode * root){
        vector <int> preorder;
        if (root == NULL)
            return preorder;

        stack <TreeNode *> st;
        st.push(root);
        while (!st.empty()){
            root = st.top();
            st.pop();
            preorder.push_back(root->val);
```

```
if (root -> right != NULL)
        st.push(root -> right);
if (root -> left != NULL)
        st.push(root -> left);
}
return preorder;
}
```

Iterative Inorder Traversal (Left Root Right)



4 2 6 5 7 1 3

Code

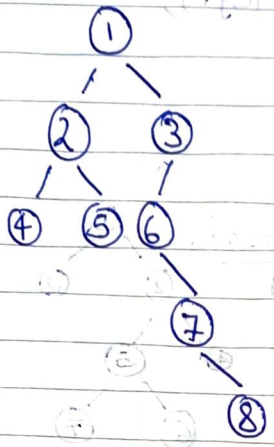
```
stack<TreeNode*> st;
TreeNode* node = root;
vector<int> inorder;
while (true) {
    if (node != NULL) {
        st.push(node);
        node = node -> left;
    }
    else {
        if (st.empty()) break;
        node = st.top();
        st.pop();
        inorder.push_back(node -> val);
        node = node -> right;
    }
return inorder;
```

3 7 6 5 4 2 1

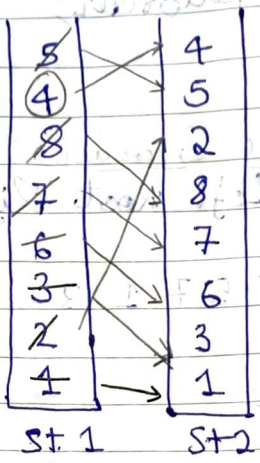
4 2 6 5 7 1 3

Iterative Postorder Traversal

2 Stack



4 5 2 8 7 6 3 1

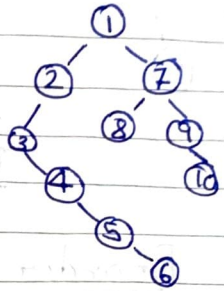



```
vector <int> postorder;
if (root == NULL)
        return postorder;
stack <TreeNode*> st1, st2;
st1.push(root);
while(!st1.empty()){
  root = st1.top();
  st2.push(root);
  st1.pop();
  if(root->left != NULL)
        st1.push(root->left);
  if (root->right != NULL)
        st1.push(root->right);
```

```
while (!st2.empty())
    postorder.push_back(st2.top->val);
    st2.pop();

return postorder;
```

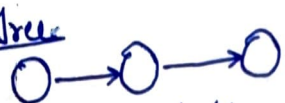
	Time	Space
1. stack	O(2n)	O(N)

6 5 4 3 2 8 10 9 7 1




```
while (cur != null || !st.empty())
{
if (cur != null)
    st.push(cur)
    cur = cur->left;
else
    temp = st.top()->right;
    if (temp == Null)
        temp = st.top()
        st.pop()
        ans.push_back(temp);
        while (!st.empty() && temp == st.top()->right)
            temp = st.top(), st.pop();
            ans.push_back(temp->val);

else
    cur = temp;
```

Skew Tree

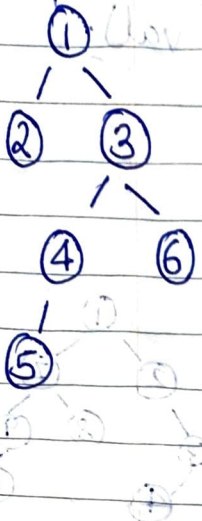


all node at max have only 1 child

Max Dept in Binary Tree

TC → O(N)
SC → O(N)
↓ for skew tree

1 + max (l + r)

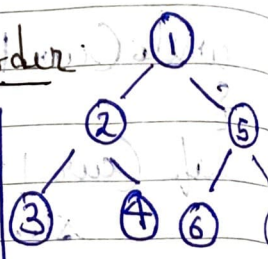


Code:

```
int maxDepth (TreeNode * root){
    if (root == NULL)
        return 0;
    int lh = maxDepth (root → left);
    int rh = maxDepth (root → right);
    return 1 + max (lh, rh);
}
```

Preorder, Inorder & Postorder

TC → O(3N) SC → O(4N)



Preorder → 1 2 3 4 5 6 7
Inorder → 3 2 4 1 6 5 7
Postorder → 3 4 2 6 7 5 1

(node, num)

(7,1) 3
(6,1) 3
(5,1) 3
(4,1) 3
(3,1) 3
(2,1) 3
(1,2) 3
(1,1)

if num == 1
preorder
++
left

if num == 2
inorder
++
right

if num == 3
postorder

Code:

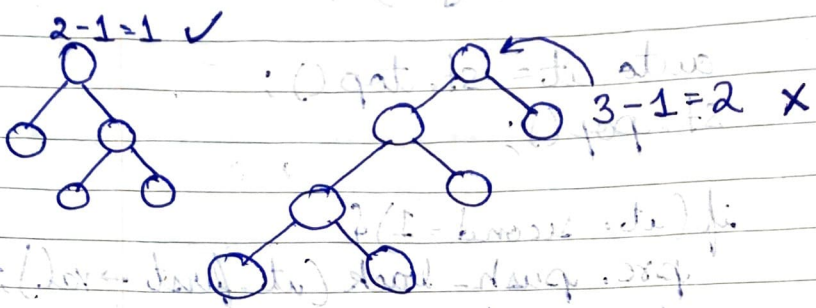
```
stack{ pair<TreeNode*, int>> st;
st.push({root, 1});
vector<int> pre, in, post

if(root == NULL)
    return;

while(!st.empty()){

    auto it= st.top();
    st.pop();

    if(it.second = 1){
      pre.push_back(it.first->val);
      it.second++;
      st.push(it);
      if(it.first -> left != NULL){
          st.push({it.first->left, 1});
      }
    }

    else if(it.second == 2){
      in.push_back(it.first->val);
      it.second++;
      st.push(it);
      if(it.first -> right != NULL){
          st.push({it.first->right, 1});
      }
```

else {
post.push_back(it.first->val);
}
}

time O(N)
space O(N)

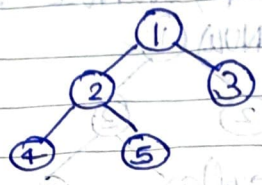
Check for Balanced Binary Tree

Balanced BT → for every node, height(left) - height(right) <= 1




```
int check(node){
  if(node == NULL)
    return 0;
  lh = check(node->left)
  rh = check(node->right)
  if(lh == -1)
    return -1;
  if(rh == -1)
    return -1;
  if(abs(lh-rh) > 1)
    return -1;
  return max(lh, rh) + 1;
}
```

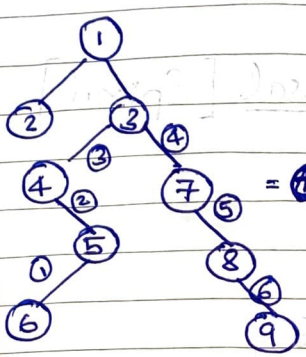
Diameter of Binary Tree



= 3 → longest path between 2 nodes → path does not need to pass via root

Code:

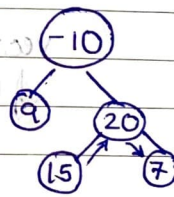
```
TreeNode*
int ( node, int & dia){
if (!node){
    return 0;
}
int lh = ht(node->left, dia);
int rh = ht(node->right, dia);
dia = max(dia, lh+rh);
return 1 + max(lh, rh);
}
```



= 6

Max Path Sum

any given node = val + (maxL + maxR)



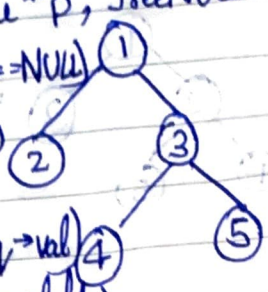
Code:

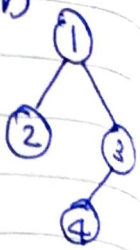
```
int maxPath (TreeNode* node, int & maxi)
if (node == NULL) return 0;
int left = max(0, maxPath(node->left, maxi));
int righ = max(0, maxPath(node->right, maxi));
maxi = (max, left + right + node->val)
return max(left, right) + node->val;
```

15+20+7 = 42

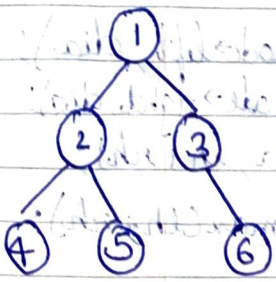
Check if two trees are Identical

```
bool issame( TreeNode *p, TreeNode* q)
if (p==NULL||q==NULL)
    return (p==q)
return (p->val==q->val)
&& issame(p->left, q->left)
&& issame( p->right, q->right);
```





Zig-Zig Traversal [Spiral]



Output :

1

3 2

4 5 6

```
vector <int>> ans;
if (Node==NULL)
    return ans;

queue < TreeNode *> q;
q.push (root);
bool flag = true;
```

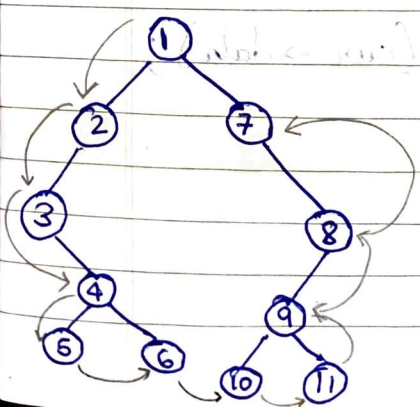
```
while (!q.empty()){
    int size = q.size();
    vector<int> row(size);
    for(int i=0 ; i<size ; i++){
        TreeNode* node = q.front();
        q.pop();

        int index = (flag) ? i : (size-1-i);
        row[index] = node -> val;

        if (node -> left)
            q.push(node -> left);
        if(node -> right)
            q.push(node -> right);
    }
    flag = !flag;
    ans.push_back(row);
}
return ans;
```

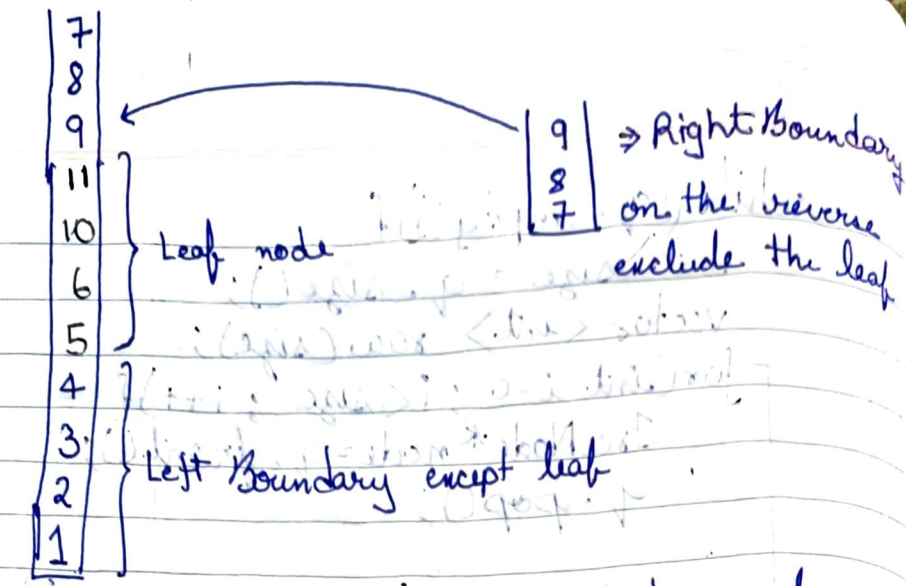
Boundary Traversal

→ Leaf Boundary excluding leaf

→ Leafnode

→ Right Boundary in the reverse exclude the leaf





ds - Leaf node ⇒ inorder traversal (or) preorder

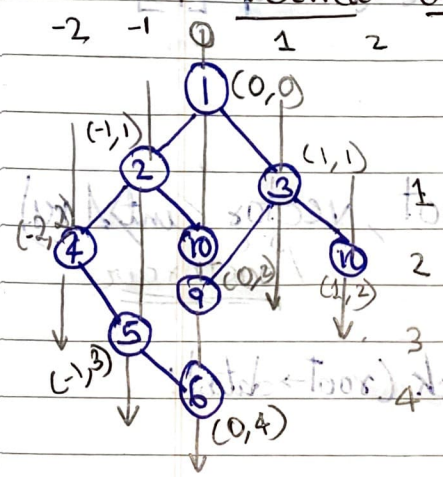
Code:

```
bool isleaf (Node* root){
    if(!root->left && !root->right)
        return true;
    return false;
}

void addleftBoundary(Node* root, vector<int>&res
{
    Node* curr= root->left;
while(curr){
    if(!isLeaf(curr))
        res.push_back(cur->data);
    if(cur->left)
        cur=cur->left
    else
        cur= cur->right;
}
```

```
void addrightBoundary(Node* root, vector<int>& res)
{
    Node* curr = root->right;
    vector<int> temp;
    while(curr){
        if(!isLeaf(curr))
            temp.push_back(curr->data);
        if(curr->right) curr = curr->right;
        else
            curr = curr->left;
    }
    for(int i = temp.size()-1; i>=0; i--){
        res.push_back(temp[i]);
    }
}

void addLeaves(Node* root, vector<int>& res)
                                    preorder
{
    if(isLeaf(root)){
        res.push_back(root->data);
        return;
    }
    if(root->left)
        addLeaves(root->left, res);
    if(root->right)
        addLeaves(root->right, res);
```

```
vector<int> printBoundary
(Node* root)
{
vector<int> res;
if(!root)
    return res;
if(!isLeaf(root))
    res.push_back(root->data);
addLeftBoundary(root, res);
addLeaves(root, res);
addRightBoundary(root, res);
return res;
}
```

Vertical order Traversal Left (-1, +1)
right (+1, +1)



4
2 5
1 9 10 6
2 3 → smaller value 1st
10
3
4

node=1 (0,0)
nod=2 (-1,1)

(1,0,0) (2,-1,1) (3,1,1) (4,-2,2) (10,0,2) (9,0,2) (10,2)

Q (node, v, level)

vertical ↓ level ↓

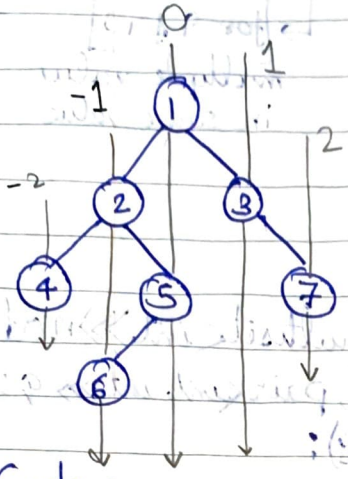
map<int, map<int, multiset<int>>

↳ [for 9 & 10] multiple value in same value

Code :

```
map<int, map<int, multiset<int>>> nodes;
queue<pair<TreeNode*, pair<int, int>>> q;
q.push({root, {0, 0}});
while(!q.empty()){
    auto p = q.front();
    q.pop();
    TreeNode* node = p.first;
    int x = p.second.first, y = p.second.second;

    nodes[x][y].insert(node->val);
    if(node->left)
        q.push({node->left, {x-1, y+1}});
    if(node->right)
        q.push({node->right, {x+1, y+1}});
}
vector<vector<int>> ans;
for(auto &p: nodes){
    vector<int> col;
    for(auto &q: p.second){
        col.insert(col.end(), q.second.begin(),
                   q.second.end());
    }
    ans.push_back(col);
}
return ans;
```

Top view of a Binary Tree



4 2 1 3 7

(5, 0) ⇒ will not be considered as we have 1

(4, -2)

(3, 1)

(2, 1)

(1, 0)

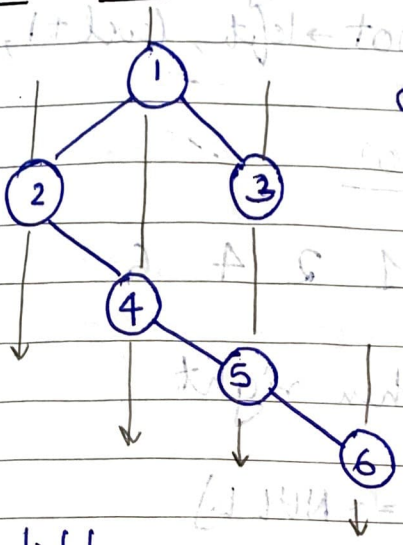
Code:

```
vector <int> ans;
if (root == NULL)
    return ans;
queue <pair<Node* int>> q;
q.push({root, 0});
while (!q.empty()) {
    auto it = q.front();
    q.pop();
    Node * node = it.first;
    int line = it.second;
    if(mp.find(line) == mp.end())
        mp[line] = node -> data;
    if (node -> left)
        q.push({node -> left, line - 1});
    if (node -> right)
        q.push({node -> right, line + 1});
}
```

map<int, int> mp;

```
for(auto &it : mp)
    ans.push_back(it.second);
return ans;
```

Bottom View



ans[]: 2 4 5 6

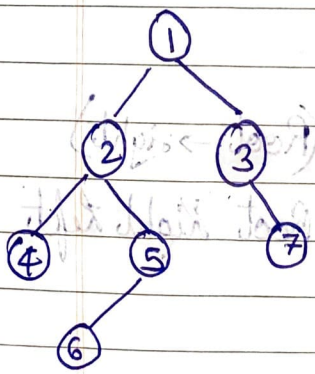
Code diff

```
if(mp.find(line) == mp.end())  } => mp[line] =
    mp[line] = node->data      }      node->data;
```

Time: O(N) Space: O(H) → height

Right Side

ans[]: 1 3 7 6




```
if(!node) return
if(level == ds.size()) ds.push_back(node
if(node->right, level+1);
if(node->left, level+1);
```

Code

```
void helper(TreeNode * root, int level, vector<int>& res){
    if(root == NULL)
        return;
    if (res.size() == level)
        res.push_back(root->val);
    helper(root->right, level+1, res);
    helper(root->left, level+1, res);
```

Left view

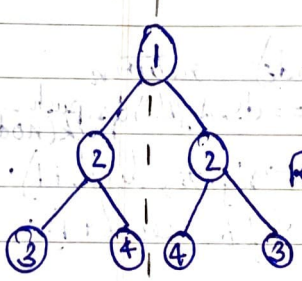
ans[]: 1 2 4 6

⇒ Left then right

Code:

```
if(root == NULL)
    return;
if(ans.size() == level)
    ans.push_back(root->val);
helper(root->left, level+1);
helper(root->right, level+1);
```

Symmetrical Binary Tree

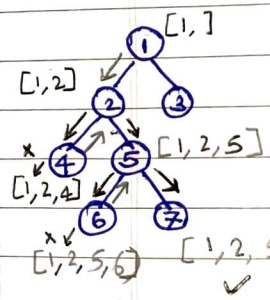


Mirror

(Root->left)	(Root->right)
Root Left Right	Root Right Left

```
class Solution{
public:
    bool isSymmetric(TreeNode* root){
return root==NULL || helper(root->left, root->right);
    }

    bool helper(TreeNode* left, TreeNode* right)
    {
        if(left==NULL || right==NULL)
            return left==right

        if(left->val !=right->val)
            return false

return helper(left->left, right->right) && helper(left->right, right->left);
    }
```

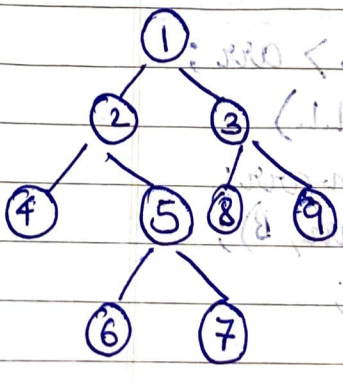
Root to Node Path




```
vector<int> arr;
if(A==NULL)
    return arr;
getPath(A, arr, B);
return arr;
```

```
bool getPath(TreeNode *root, vector<int> &arr,
             int x){
    if(!root)
        return false;

    arr.push_back(root->val);
    if(root->val == x)
        return true;
    if(getPath(root->left, arr, x) ||
       getPath(root->right, arr, x))
        return true;

    arr.pop_back();
    return false;
}
```

Lowest Common Ancestor

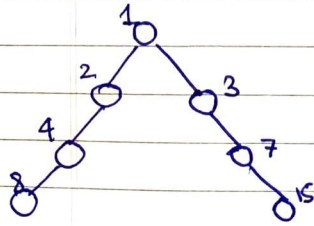


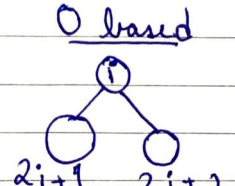
lca(4, 7) = 2
lca(5, 9) = 1
lca(2, 6) = 2

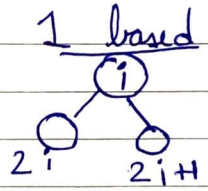
```
TreeNode * lca(TreeNode* root, TreeNode* p, TreeNode* q){
    if (root == NULL || root == p || root == q)
        return root;

TreeNode * left = lca(root->left, p, q);
TreeNode * right = lca(root->right, p, q);

if (!left)
    return right;
else if (!right)
    return left;
else
    return root;
}
```

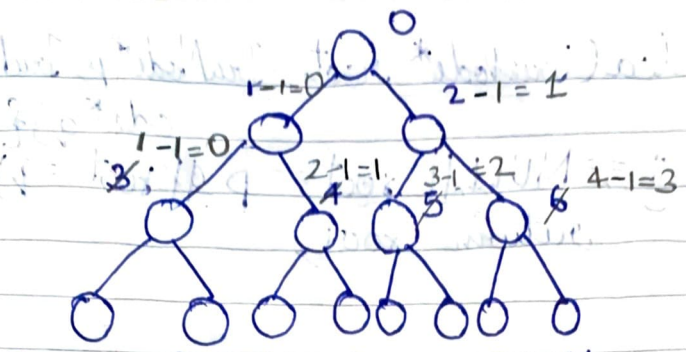
Max Width of Binary Tree

width = no. of nodes in a level between any 2 nodes





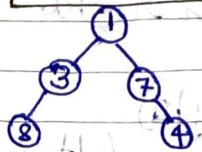


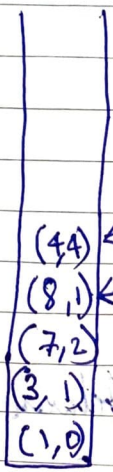


if(i - min)

2i+1 2i+2

(last - first + 1)





2*(2-1)+1+1 = 2*1+2 = 4

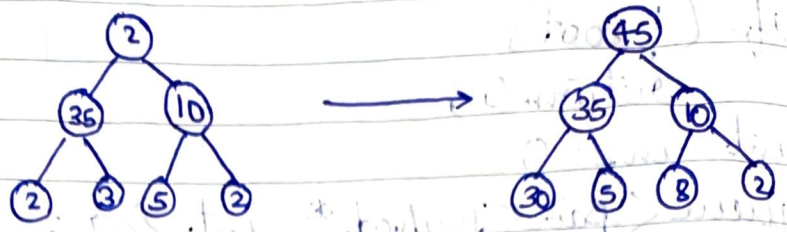
2*(1-1)+1
= 0+1 = 1

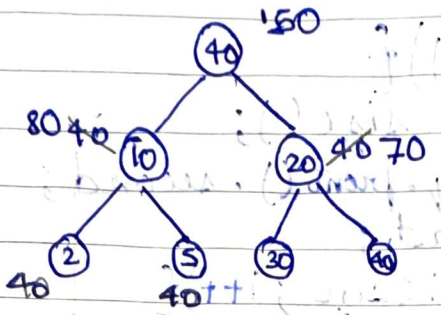
4-1+1 = 4

Code:

```
if (!root)
    return 0;
int ans = 0;
queue<pair<TreeNode*, int>>q;
q.push ({root, 0});

while (!q.empty()){
    int size = q.size();
    int mn = q.front().second;
    int first, last;
    for(int i=0; i<size; i++){
        int curid = q.front().second - mn;
        TreeNode* node = q.front().first;
        q.pop().
        if (i==0)
            first = curid;
        if(i==n-1)
            second = curid
        if (node->left)
            q.push({node->left, curid*2+1});
        if (node->right)
            q.push({node->right, curid*2+2});
    }
    ans = max(ans, last-first+1);
}
return ans;
```

Children Sum Property O(N)





10 + 20 = 30 < 40

2 + 5 = 7 < 40

30 + 40 = 70 >= 40

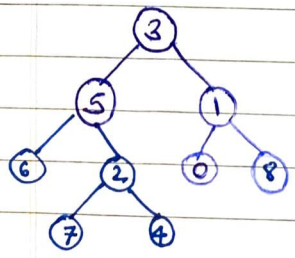
70 + 80 = 150 > 40

Code :

```
if (!root) return;
int child = 0;
if (root → left)
    child += root → left → data;
if (root → right)
    child += root → right → data;
```

```
if (child >= root → data)
    root → data = child;
else
    if (root → left)
        root → left → data = root → data;
    else if (root → right)
        root → right → data = root → data;

changeTree (root → left)
changeTree (root → right)

int tot = 0;
if (root → left)
    tot += root → left → data;

if (root → right)
    tot += root → right → data;

if (root → left || root → right)
    root → data = tot;
```

Nodes at a distance K



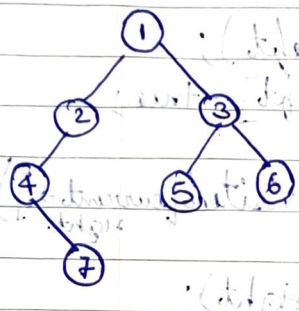
K=2, target = 5

```
void markparents(TreeNode* root, unordered_map
<TreeNode*, TreeNode*> &parents)
{
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty()) {
    TreeNode* current = q.front();
    q.pop();
    if(current->left) {
        parent[current->left] = current;
        q.push(current->left);
    }
    if(current->right) {
        parent[current->right] = current;
        q.push(current->right);
    }
}

public<int> distanceatK(TreeNode* root, TreeNode*
target, int k){
unordered_map<TreeNode*, TreeNode*> parent;
markparents(root, parent);

unordered_map<TreeNode*, bool> visited;
queue<TreeNode*> q;

q.push(target);
visited[target] = true;
```

```
while (!q.empty()) {
    int size = q.size();
    if (curr_level++ == k)
        break;

    for(int i=0; i< size; i++) {
        TreeNode* current = q.front();
        q.pop();
        if(current->left && !visited[current->left])
    {
        q.push(current->left);
        visited[current->left] = true;
    }
        if(current->right && !visited[current->right])
    {
        q.push(current->right);
        visited[current->right] = true;
    }
        if(parent[current] && !visited[parent[current]])
    {
        q.push(parent[current]);
        visited[parent[current]] = true;
    }
    }
}
```

```
vector <int> ans;
while ( ! queue . empty() ) {
    TreeNode* current = queue . front();
    queue . pop();
    ans . push_back (current → val);
}
return ans;
}
```

Minimum Time taken to BURN the BT



same as before

to find TreeNode ⇒ in parent

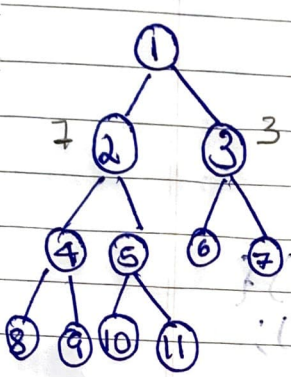
```
BinaryTreeNode* res;
while(!q.empty())
    BinaryTreeNode* node = q.front()
    if [node → data == target] {
        res = node;
```

To find TreeNode

```
int findMaxDistance(map<TreeNode*, TreeNode*> &parents, BinaryTreeNode * target)
{
    queue<TreeNode*> q;
    q.push(target);
    map<TreeNode*, int> vis;
    vis[target] = 1;
    int ans = 0;
    while(!q.empty()){
        int n = q.size();
        int fl = 0;
        for(int i=0; i<n; i++){
        TreeNode * temp = q.top();
        q.pop();
        if(node->left && !vis[node->left]){
            fl=1;
            vis[node->left] = 1;
            q.push(node->left);
        }
        if(node->right && !vis[node->right]){
            fl= 1;
            vis[node->right] = 1;
            q.push(node->right);
        }
        parent
        if(          [node] && !vis[parent[node]]){
            fl=1;
            vis[parent[node]] = 1;
            q.push(     parent[node]);}}
```

```
    if (fl)
        ans++;
}
return ans;
```

Total Nodes in a Complete BT



1+7+3
= 11

lh = 3 $2^3 - 1 = 7$
rh = 3

lh = 2 $2^2 - 1 = 3$
rh = 2

```
if (lh == rh)
    h^2 - 1
else
    root + root->left + root->right)
```

Code in CountN(TreeNode* root) {

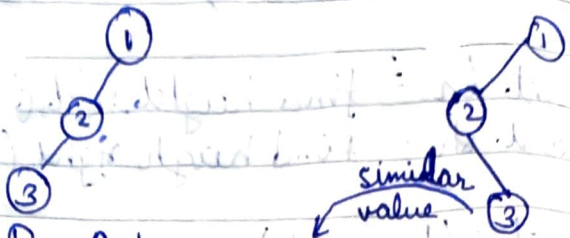
```
if (!root)
    return 0;

int lh = findheightleft(root);
int rh = findheighright(root);

if (lh == rh)
    return (1 << lh) - 1;
```

→ 2^{lh}

```
return 1 + countN(root->left) +
           countN(root->right);
}

int findheightleft(TreeNode* root){
    int hg = 0;
    while(root){
        hg++;
        root = root->left;
    }
    return hg;
}

int findheightright(TreeNode* root){
    int hg = 0;
    while(root){
        hg++
        root = root->right;
    }
    return hg;
```

Q) Can you construct Binary Tree?

Ans)



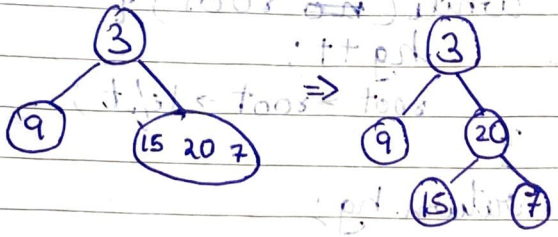
Pre-Order → 1 2 3

Post-Order → 3 2 1

So unique can't be constructed using post & pre

Inorder = [9 3 15 20 7]

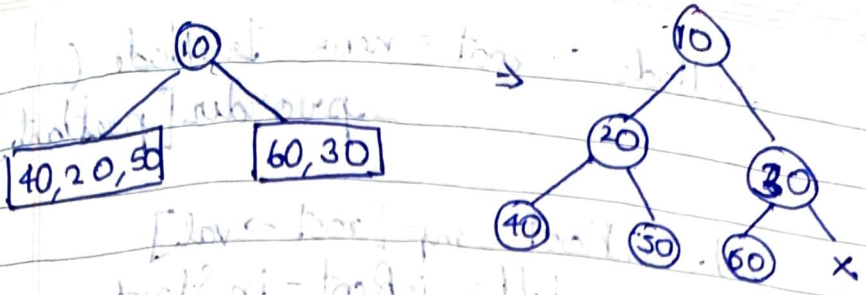
Preorder = [3 9 20 15 7]



Construct Binary Tree from Preorder & Inorder

inorder → [40 20 50 10 60 30]

preorder → [10 20 40 50 30 60]



Code

```
TreeNode* buildTree(vector<int>& preorder,
  vector<int>& inorder)
{
  map<int, int> mp;

  for(int i=0; i<inorder.size(); i++){
        mp[inorder[i]] = i;
  }
  TreeNode * root = buildtree(preorder, 0,
  preorder.size()-1, inorder, 0, inorder.size()-1,
  mp);

  return root;
}
```

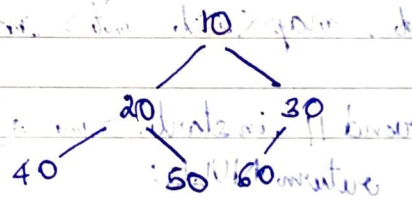
```
TreeNode* buildTree(vector<int>& preorder, int
prestart, int preend, vector<int>& inorder, int
instart, int inend, map<int, int> mp){

  if(prestart > preend || instart > inend)
        return NULL;
```

```
TreeNode * root = new TreeNode (
                    preorder[prestart])

int inRoot = mp[root->val]
int numsLeft = inRoot - inStart

root->left = buildTree(preorder, prestart+1,
    prestart + numsleft, inorder, inStart,
    inRoot - 1, mp);

root->right = buildTree(preorder, prestart +
    numsleft + 1, preEnd, inorder, inRoot+1,
    inEnd, mp);

return root;
}
```

Construct Binary Tree from Postorder & Inorder Traversal

Inorder : [40 20 50 10 60 30]
Postorder : [40 50 20 60 30 10]

40 20 50 10 60 30



Code :

```
{ if (inorder.size() != postorder.size())
        return NULL;

    map <int, int> mp;

    for(int i=0; i<inorder.size(); i++)
        mp[inorder[i]] = i

    return buildTreePostIn (inorder, 0,
    inorder.size()-1, postorder, 0, postorder.
                                    size()-1,
    mp);
}
TreeNode* buildTreePostIn( vector<int>& inorder,
int instart, int inend, vector<int>& postorder,
int poststart, int postend, map<int, int>
& mp ) {
    if (poststart > postend || instart
                                > inend)
        return NULL;

    TreeNode* root = new TreeNode(
                postorder[postend]);

int inRoot = mp[postorder[postend]];
int numsLeft = inRoot - instart;
```

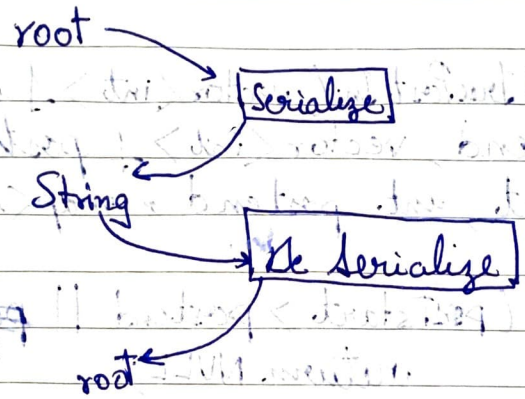
→ root → left = buildTreePostIn(inorder, inStart, inRoot - 1, postorder, postStart, postStart + numsLeft - 1, mp);

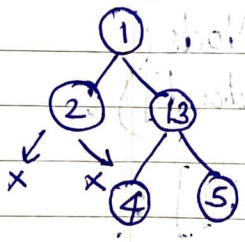
→ root → right = buildTreePostIn(inorder, inRoot + 1, inEnd, postorder, postorder + numsLeft, postEnd - 1, mp);

return root;

}

Serialize & De Serialize





String → 1, 2, 13, #, #, 4, 5, #, #, #, #

Code:

```
string serialize(TreeNode * root) {
    if(!root)
        return " ";

    string s = " ";
    queue<TreeNode *> q;
    q.push(root);
    while(!q.empty()) {

        TreeNode * curr = q.front();
        q.pop();
        if (!curr)
            s.append("#,");
        else
            s.append(to_string(curr->val)+
        ;
        if ( curr) {
          q.push (curr->left);
          q.push(curr->right);
        }

}
```

```
TreeNode * deserialize (string data) {
    if (data.size() != 0)
        return NULL;
    stringstream s(data);
    string str;
    getline (s, str, ',');
    TreeNode * root = new TreeNode(stoi(str));
    queue<TreeNode *> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode * node = q.front();
        q.pop();

        getline (s, str, ',');
        if (str == "#") {
            node -> left = NULL;
        }
        else {
            TreeNode * leftn = new TreeNode(stoi(str));
            node -> left = leftn;
            q.push (leftn)
        }

        getline (s, str, ',');
        if (str == "#") {
            node -> right = NULL;
        }
    }
```

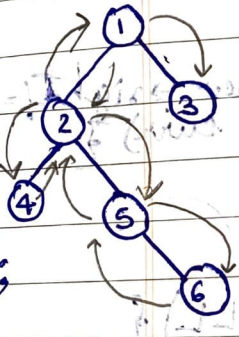
```
else {
    TreeNode * rightn = new TreeNode
                        (stoi(str));
    node -> right = rightn;
    q.push(rightn);
}
}
return root;
}
```

Morris Traversal

↳ Inorder Time O(N) Space O(1)

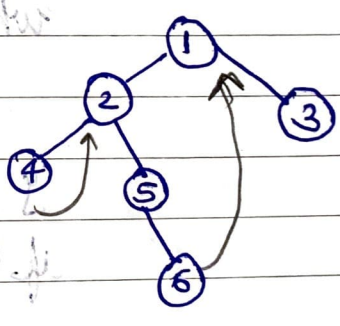
⇒ Threaded Binary Tree

4 2 5 1 6 1 3



Case (i)

left → null
print()
move to right.



Case (ii)

→ Before going to left, the right-most will be connected to cur from left subtree.

→ if thread exists then remove thread
curr to right

Code:

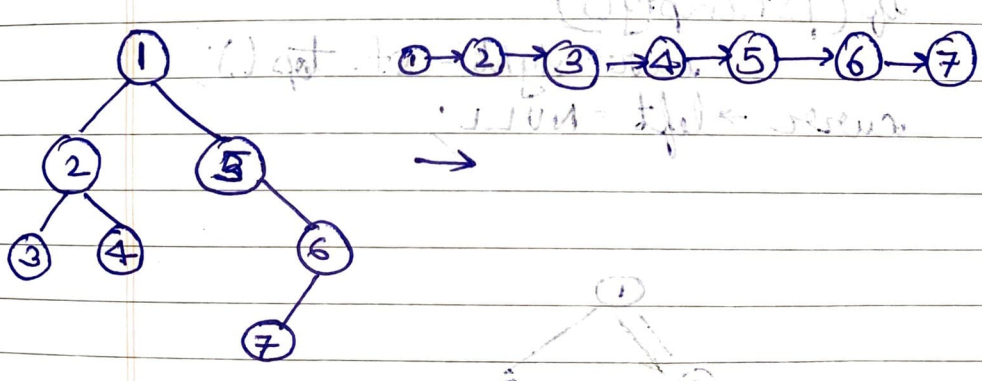
```
vector <int> getInorder(TreeNode *root){
  vector <int> inorder
  TreeNode * curr = root;

  while (curr != NULL) {
    if (curr → left == NULL) {
        inorder . push_back(curr→val);
        curr = curr → right;
    }
    else{
      TreeNode *prev = curr → left;
      while (prev → right && prev → right != curr) {
          prev = prev → right;
      }
      if (prev → right == NULL) {
          prev → right = curr;
          curr = curr → left;
      }
      else {
          prev → right = NULL;
          inorder . push_back (curr . val);
          curr = curr → right;
      }
```

```
        }
    }
    return inorder;
}
```

For preorder

```
if (prev->right == NULL)
{
    prev->right = curr;
    preorder.push_back(curr.val);
    curr = curr->next;
}
else {
    prev->right = NULL;
    cur = cur->right
}
```

Flatten a Binary Tree

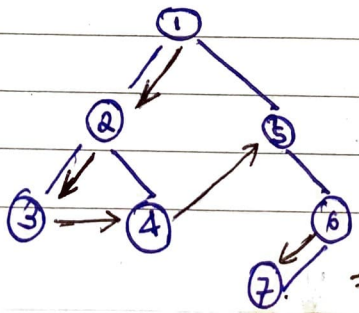


(i) TC → O(N) SC → O(N)

```
prev = null
flatten (node){
 if (node == NULL)
        return;
flatten (node → right);
flatten (node → left);
 node → right = prev;
 node → left = null;
 prev = node;
}
```

(ii) TC → O(N) SC → O(N)

```
st.push(root)
while(! st.empty){
 curr = st.top; st.pop();
 if(curr → right)
    st.push(curr → right)
if (curr → left)
    st.push(curr → left);
if (!st.empty())
        curr → right = st.top();
curr → left = NULL;
```



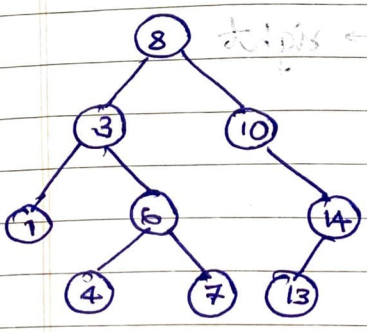
⇒ 7 right is NULL

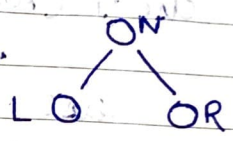
TC → O(N) SC → O(1)

(1,11)

```
curr = root;
while (curr){
    if(curr->left){
        prev = curr->left
        while(prev->right){
            prev = prev->right;
        }
        prev->right = curr->right;
        curr = curr->left;
    }
    curr = curr->right;
}
```

Binary Search Tree

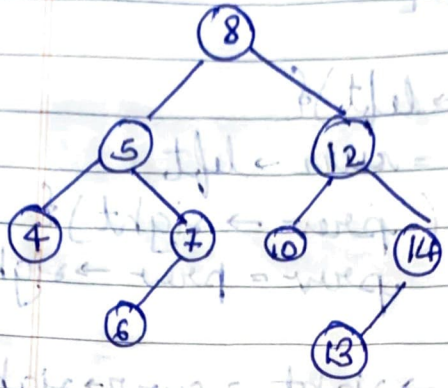




L<N<R

To store duplicate valued node we can use (node, freq)

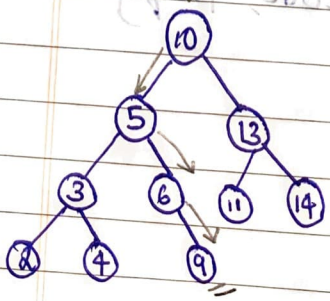
Search in a BST



Code:

```
while (root != NULL) {
    if (root->val == val)
        return root;
else if (root->val > val)
        root = root->left
    else
        root = root->right
}
return NULL;
```

Ceil in a BST

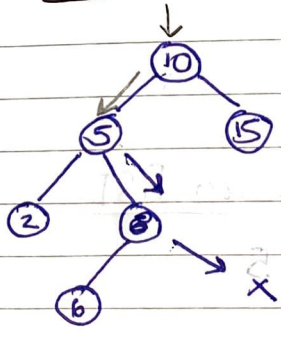


Key = 8

Code:

```
int ceil = -1;

while(root){
    if(root->data == key)
        return key;

    if(key > root->data)
        root = root->right
    else {
        ceil = root->data;
        root = root->left;
    }
return ceil;
}
```

Floor in BST



key = 9

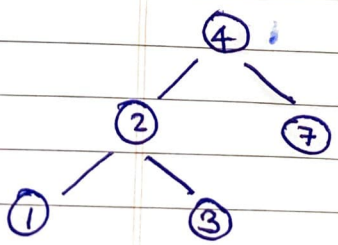
ans = 5 8

Code

```
int floor = -1;

while (floor){
    if ( root → val == key ){
        floor = root → val
        return floor;
    }
    if (key > root → val) {
        floor = root → val;
        root = root → right;
    }
    else {
        root = root → left;
    }
}
return floor;
```

Insert a given Node in BST

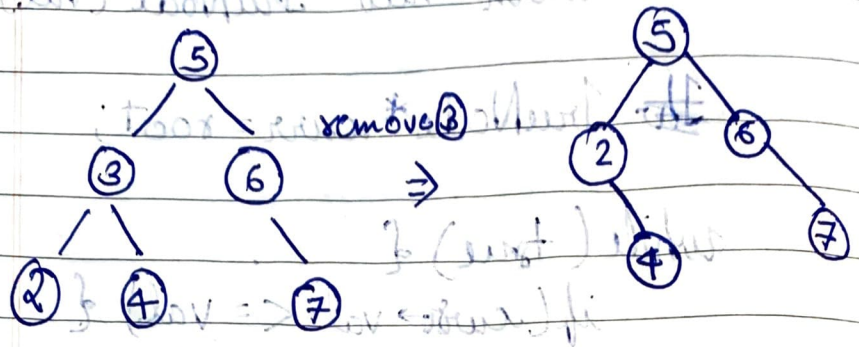
node = 5



Code:

```
if (node == NULL)
    return new TreeNode (val);

TreeNode* curr = root;

while (true) {
    if (curr->val <= val) {
        if (curr->right)
            curr = curr->right;
        else
            curr->right = new TreeNode(val);
            break;
    } else {
        if (curr->left)
            curr = curr->left;
        else
            curr->left = new TreeNode(val);
            break;
    }
}
return root;
}
```

Delete a Node in Binary Tree



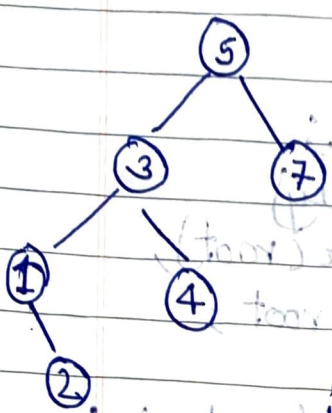
Code:

```
TreeNode * findLastRight (TreeNode* root)
{
    if (root → right)
        return root;
    return findLastRight (root → right);
};

TreeNode * helper (TreeNode * root) {
    if (! root → left)
        return root → right;
else if (! root → right)
        return root → left;

    TreeNode* child = root → right;
    TreeNode * lastright = findLastRight (root → left);
    lastright → right = child;
    return root → left;
```

```
TreeNode * deleteNode(TreeNode * root,
                      int key)
{
    if (!root)
        return NULL;
    if (root->val == key)
        return helper(root);
    TreeNode* temp = root;
    while (root) {
        if(root->val > key) {
            if (root->left && root->left->
                val == key) {
                root->left = helper(root->left);
                break;
            }
            else {
                root = root->left;
            }
        }
        else {
            if (root->right && root->right->val
                == key)
            { root->right = helper(root->right);
              break;
            }
            else {
                root = root->right;
            }
        }
    }
    return temp;
}
```

⇒ Inorder of a BST is always in a sorted order

Kth smallest element in BST



cnt = 0

Left Node Right

cnt ++

if (cnt == k)

ans = node

⇒ Code

```
stack <TreeNode*> s;
int cnt = 0;
while(true){
    if(root){
        s.push(root);
        root = root->left;
    }
    else{
        if(s.empty())
            break;
        root = s.top();
        s.pop();
        cnt++
        if(cnt==k)
            return root->val;
```

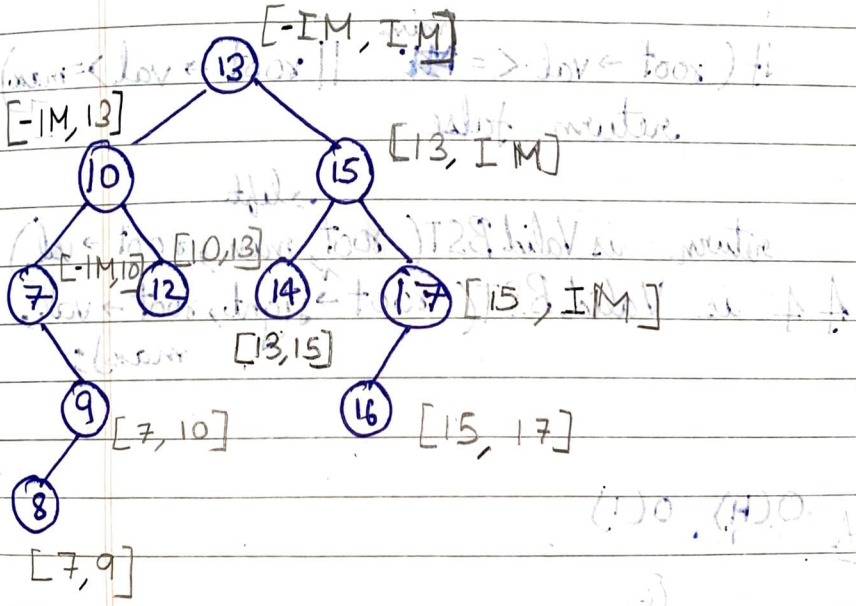
```
        root = root -> right;
    }
}
return -1;
```

Largest

K^{th} largest = $(N - K^{th})$ smallest

for the 1st traversal count the no. of nodes = N

Check if a tree is a BST

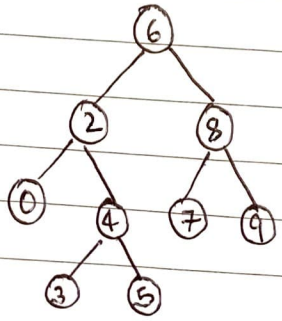


Code

public:

```
boolean isValidBST(TreeNode* root) {
    return isValidBST(root, LONG_MIN, LONG_MAX);
}

boolean isValidBST(TreeNode* root, long min, long max) {
    if(!root)
        return true;

    if(root->val <= min || root->val >= max)
        return false

    return isValidBST(root->left, min, root->val)
    && isValidBST(root->right, root->val, max);
}
```

LCA O(H) O(1)

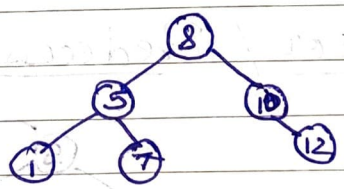


(0,5) ⇒ 2

```
TreeNode lCA ( TreeNode * root, TreeNode*
    p, TreeNode *q ){
    if (root == null)
        return NULL;
    int curr = root.val;
    if ( curr < p.val && curr < q.val)
        return LCA(root->right, p, q);
    if (curr > p.val && curr > q.val)
        return lCA( root->left; p, q);
    retur root;
}
```

Construct BST from preorder Traversal

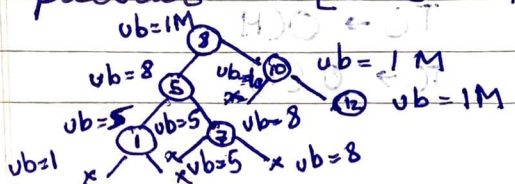
① ⇒ preorder → { 8, 5, 1, 7, 10, 12]



← Insert one by one TC ⇒ O(N×N)
SC ⇒ O(1)

② ⇒ Sorting = [1, 5, 7, 8, 10, 12]
tt gives inorder
TC ⇒ O(n log n) + O(n)
SC ⇒ O(N)

③ preorder → [8 5 1 7 10 12] ← In worst case we have to visit node 3 times



TC ⇒ O(3N) ≈ O(1)
SC ⇒ O(1)

Code :

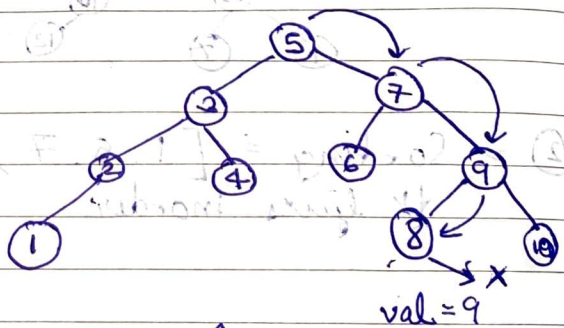
```
TreeNode* bstFromPreorder(vector<int>&A)
{
    int i=0;
    return build(A, i, INT_MAX);
}

TreeNode* build(vector<int>&A, int &i, int max)
{
    if(i==A.size() || A[i] > max)
        return NULL;
    TreeNode* root = new TreeNode(A[i++]);
root->left = build(A, i, root->val);
root->right = build(A, i, max);
    return root;
}
```

Inorder Successor / Predecessor in BST

first val > 8

TC → O(N)

SC → O(1)



TC → O(H)

TC → O(H)

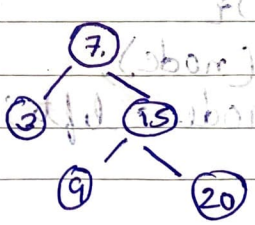
Code:

```
TreeNode* inorderS (TreeNode* root, TreeNode* P){
    TreeNode* successor = NULL;

    while (root){
        if (p->val >= root->val)
            root = root->right;
        else{
            successor = root;
            root = root->left;
        }
    }
    return successor;
}
```

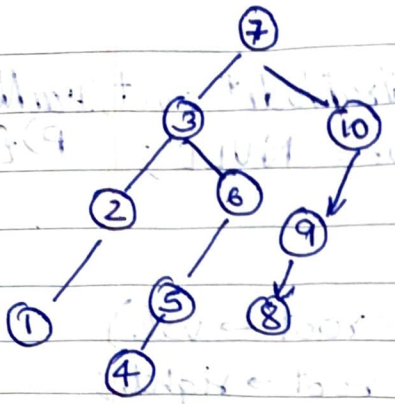
Predecessor

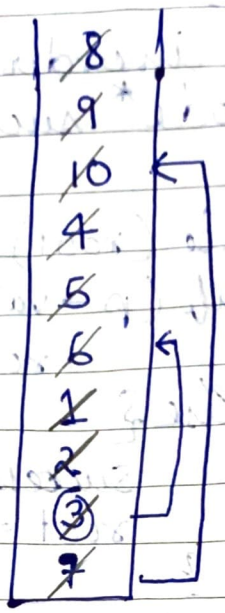
```
if (p->val <= root->val) {
        root = root->left;
}
else{
        predecessor = root;
        root = root->right;
}
```

BST iterator



next → 3
next → 7
hasnext → True
next → 9
next → 15
has next → True
next → 20
has next → false





when we remove node we go to int right

1 2 3 4 5 6 7 8 9 10

SC → O(H)

TC → O(N/N) → O(1)

↑ N push in stack in N calls

Code:

```
stack <TreeNode*> s;
void push_all(TreeNode* node){
    while(node){
        s.push(node);
        node = node->left;
    }
```

```
public:
    BSTIterator (TreeNode* root){
        push_all(root);
    }

int next(){
    TreeNode* temp = s.top();
    s.pop();
    push_all(root->right);
    return temp->val;
}

bool hasNext(){
    return !s.empty();
}
```

Two Sum IV

K = 9

Inorder

TC → O(N)

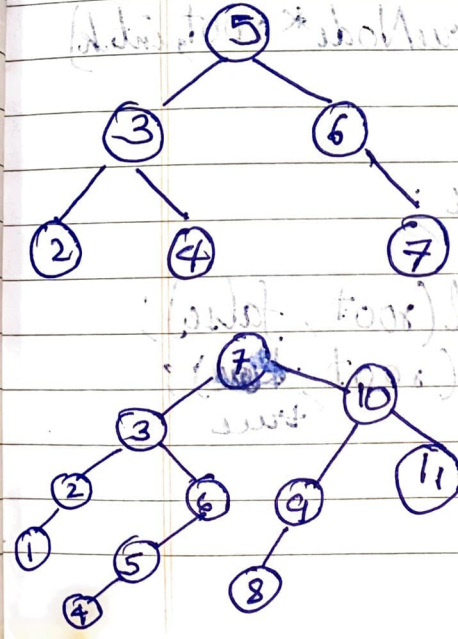
SC → O(N) + O(N)

Two pointer

Storing Inorder traversal

2 3 4 5 6 7

(Two pointer)



K = 16

i = next();

j = before();

if (sum < K)

next();

else if (sum > K) before();

TC → O(N)

SC → 2O(H)

Code:

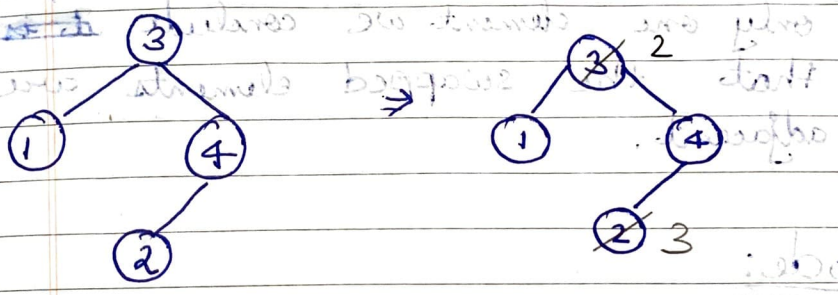
```
int next() {
    ...
    if (! reverse)
        pushAll(tmpNode -> right);
    else
        pushAll(tmpNode -> left);
    ...
}

pushAll(TreeNode * node) {
    if (reverse)
        node = node -> right
    else
        node = node -> left;
}
```

main

```
bool findTarget(TreeNode* root, int k)
{
    if (! root)
        return false;

    BST Iterator l(root, false);
    BST Iterator r(root, true);
    int i = l.next();
    int j = r.next();
```

```
while (i < j){
    if(i+j == k)
        return true;
    else if (i+j < k)
        i = l.next();
    else
        j = r.next();
}
return false;
```

Recover BST → Two nodes swapped

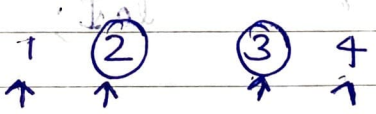


TC = 2N + N log N

Brute force SC = O(N)

→ Inorder Traversal

→ sort



TC → O(N)
SC → O(1)

Swap can have 2 cases

1. Swapped nodes are not adjacent

3 (25) 7 8 10 15 20 (5)

2. They are adjacent

3 5 (8) (7) 10 15 20 25

↑

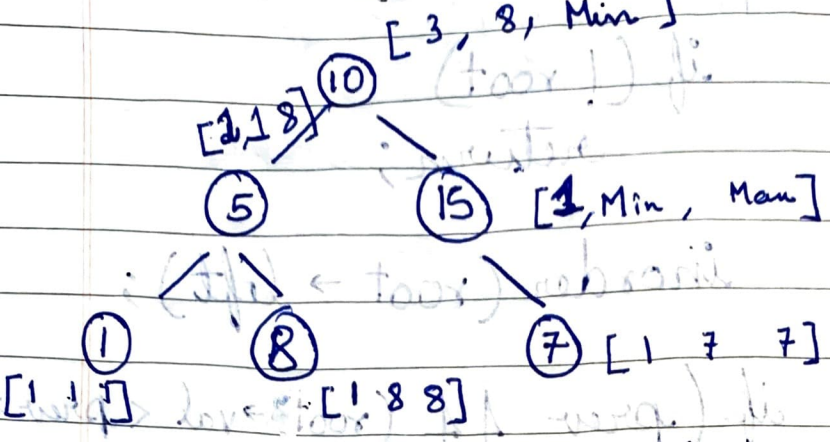
7 < 8 so when we find only one element we conclude that the swapped elements are adjacent.

Code:

```
TreeNode * first;
"        prev;
"        middle;
"        last;
```

```
void inorder (TreeNode* root){
    if (! root)
        return;
    inorder (root → left);
    if (prev && (root → val < prev → val))
    {
        if (! first) {
            first = prev;
            middle = root;
        }
        else
            last = root;
    }
    prev = root;
    inorder (root → right);
}
public:
    void recoverTree (TreeNode* root){
        first = middle = last = NULL;
        prev = new TreeNode (INT_MIN);
        inorder (root);
        if (first && last)
            swap
        else if (first && middle)
            swap (first → val, middle → val);
    }
```

Largest BST in BT



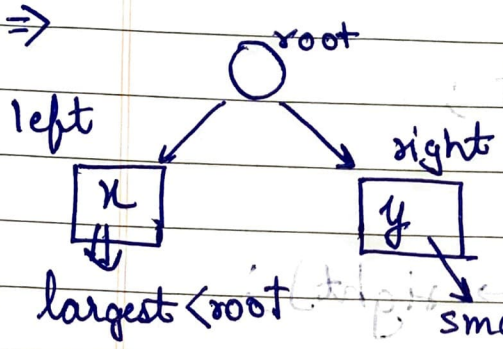
Brute force (going to every node & check if it's valid BST)

TC → O(N) × O(N) ≈ O(N²)

↳ going to every node

validate a BST ↵

⇒



largest < root < smallest

size = 1 + x + y

TC → O(N)

SC → O(1)

postorder

left

right

root

Code:

```
class NodeValue {
public:
    int maxNode, minNode, maxSize;

    NodeValue(int minNode, int maxNode,
              int maxSize) {
        this -> maxNode = maxNode;
        this -> minNode = minNode;
        this -> maxSize = maxSize;
    }
};
NodeValue largestBST(TreeNode* root) {
if(!root)
    return NodeValue(INT_MAX, INT_MIN
                     0);

auto left = largestBST(root -> left);
auto right = largestBST(root -> right);

if(left.maxNode < root -> val &&
    root -> val < right.minNode) {
    return NodeValue(min(root -> val, left.minNode)
    ,max(root -> val, right.maxNode),
    left.maxsize + right.maxsize + 1);
}
return (INT_MIN, INT_MAX, max(left.maxSize,
  right.maxsize));
}
```

in case of !root

```
int largestBSTSubtree(TreeNode *root){
    return largestBSTSubtree(root).maxSize;
}
```