

**BINARY TREE
AND
BINARY SEARCH
TREE**

**Handwritten Notes of
Striver(TUF) Playlist**

by: Aashish Kumar Nayak

NIT Srinagar

Linkedin Instagram

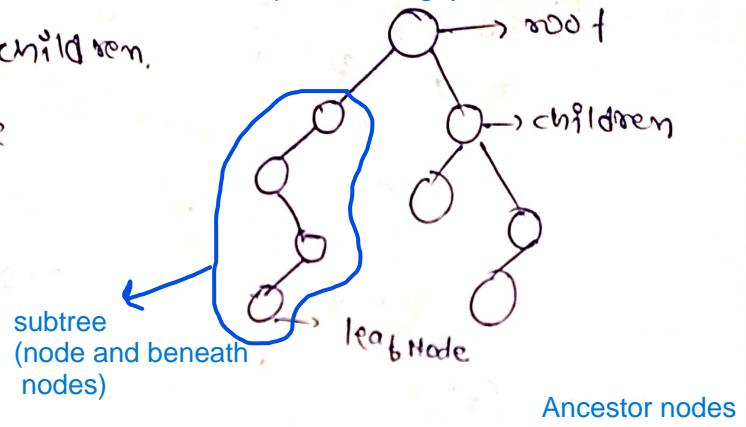
L1 Introduction to Binary Trees

at max 2 guys

FULL BT \rightarrow Either 0 or 2 children.

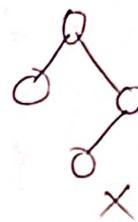
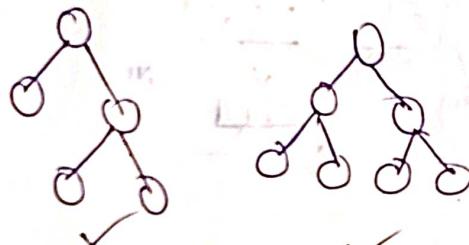
five types of Binary Tree

- (•) Full BT
- (•) complete BT
- (•) perfect BT
- (•) Balanced BT
- (•) Degenerate tree



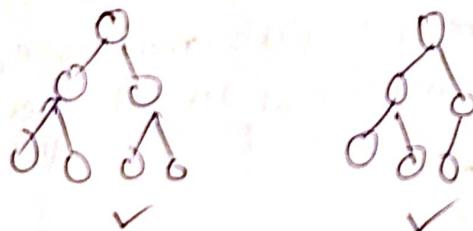
(•) Full Binary Tree

Either 0 or 2 children.



(•) Complete Binary Tree

- ① all levels are completely filled except the last level.
- ② the last level has all nodes on left as possible.



(•) Perfect Binary Tree

All leaf nodes are at same level.



(•) Balanced Binary Tree

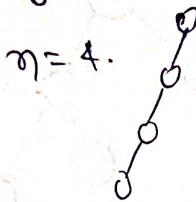
Height of tree at max $\log_2(N)$

$$\text{Ex: } n=8 \quad \log_2 8 = 3$$



① Degenerate Tree

Every node have single children.

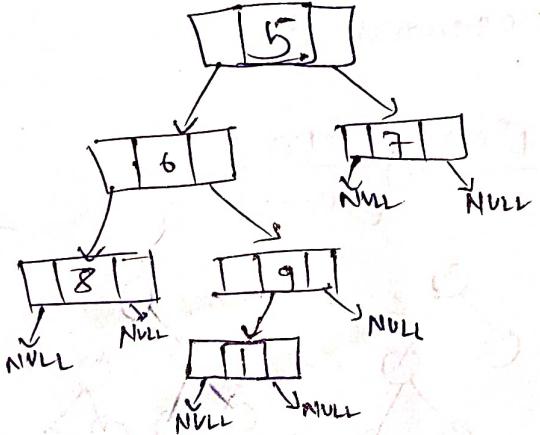
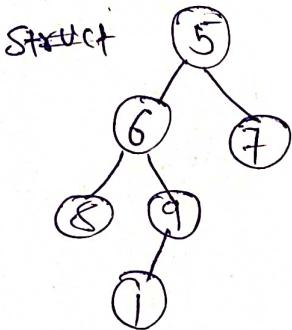


@Aashish Kumar Nayak

+

skew trees

L2 Binary Tree representation in C++



struct Node {

```

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

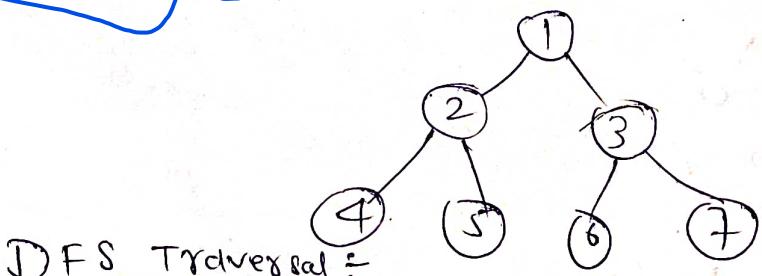
main()

```

{
    struct Node *root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->right = new
    Node(5);
}
  
```

L4 Traversal Technique

BFS / DFS



DFS Traversal :-

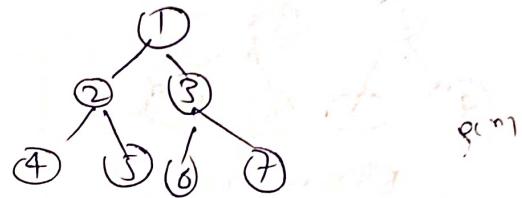
- ① Inorder Traversal (LNR)
- ② Preorder Traversal (NLR)
- ③ Postorder Traversal (LRN)

4	2	5	1	6	3	7
1	2	4	5	3	6	7
4	5	2	6	7	3	1

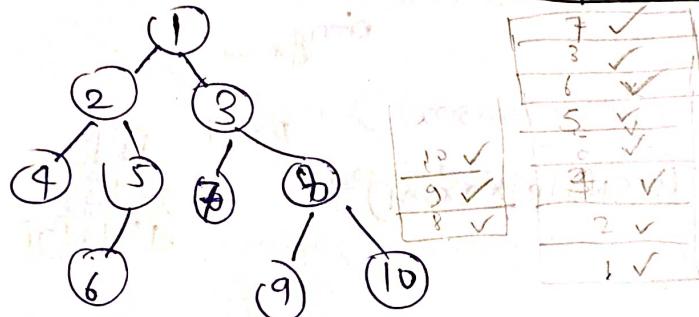
(a) BFS Traversal

① For level wise order, / level order traversal.

1 2 3 4 5 6 7 8

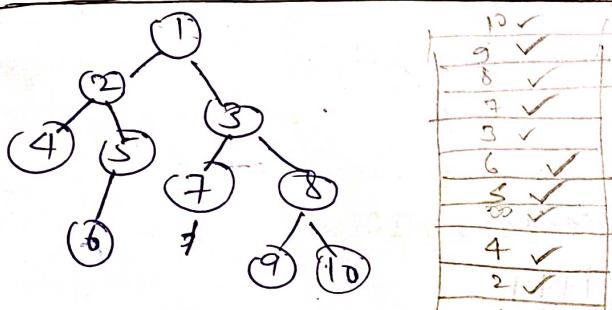


Pre-order Traversal (NLR)



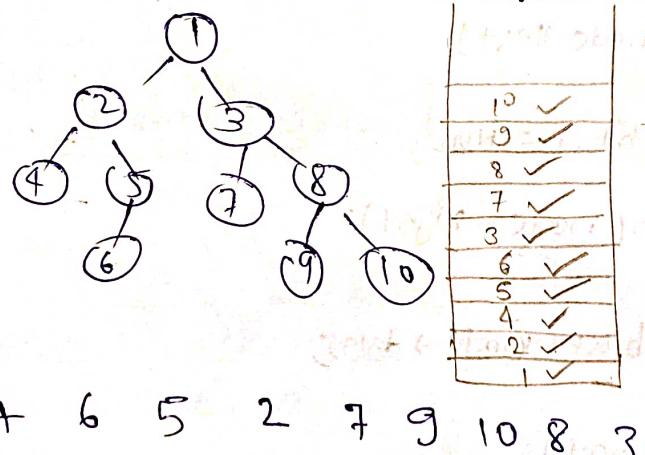
1 2 4 5 6 3 7 8 9 10

In-order Traversal (LNR)



4 2 6 5 1 7 3 9 8 10

Post-order Traversal (LRN)



4 6 5 2 7 9 10 8 3

void preorder(node)

```

{
  if (node == NULL)
    return;
  print(node->data);
  preorder(node->left);
  preorder(node->right);
}
  
```

TC: O(N), SC: O(N) -> max len of tree

void inorder(root)

```

{
  if (root == NULL)
    return;
  inorder(root->left);
  print(root->data);
  inorder(root->right);
}
  
```

TC: O(N), SC: O(N)

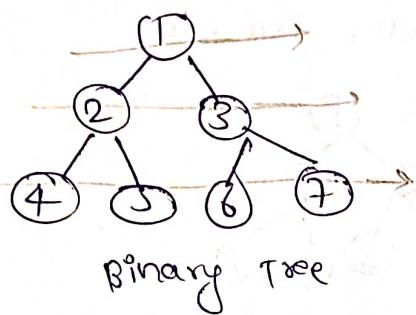
void postorder(root)

```

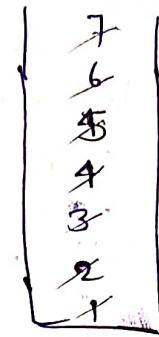
{
  if (root == NULL)
    return;
  postorder(root->left);
  postorder(root->right);
  print(root->data);
}
  
```

TC: O(N), SC: O(N) -> Aux space

level order traversal



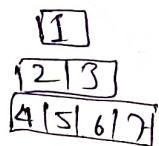
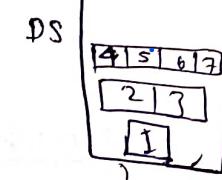
1
2 3
4 5 6 7



Queue

empty now

print



Code :-

```

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();
            if (node->left != NULL)
                q.push(node->left);
            if (node->right != NULL)
                q.push(node->right);
            level.push_back(node->data);
        }
        ans.push_back(level);
    }
    return ans;
}
  
```

Iterative method of Preorder Traversal (using stack)

```
vector<int> preorder(TreeNode* root)
```

```
vector<int> ans;
```

```
if (root == NULL)
```

```
{ return ans;  
}
```

```
Stack<TreeNode*> st;
```

```
st.push(root);
```

```
while (!stack)
```

```
{
```

```
TreeNode*
```

```
(root = st.top();  
st.pop();
```

```
ans.push_back(root->data);
```

```
if (root->right != NULL)
```

```
{  
    st.push(root->right);
```

```
if (root->left != NULL)
```

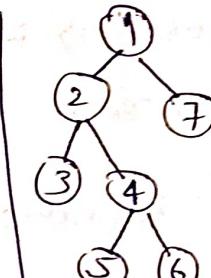
```
{  
    st.push(root->left);
```

```
}
```

```
return ans;
```

```
}
```

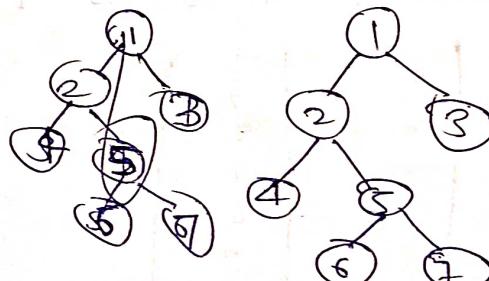
```
}
```



1 2 3 4 5 6 7



Iterative Method of Inorder Traversal (LNR)



4 2 6 5 7 1 3

`vector<int> inorderTraversal(TreeNode *root)`

`stack<TreeNode*> st;`

`TreeNode *node = root;`

`vector<int> inorder;`

`while (true)`

`if (node != NULL)`

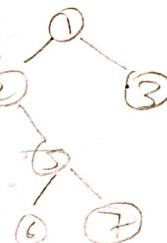
`st.push(node);
node = node->left;`

`else`

`if (st.empty() == true)`

`break;`

L N R



Step 1:

left में यात्रा
left में यात्रा नहीं
3 के stack में push करने वालों

Step 2:

पॉप करना वालों

top element Pick नहीं करा
A insert करों और Right में नहीं stopPop();
node = st.top();

Step 3:

stack.empty() == true होता है।
it break;

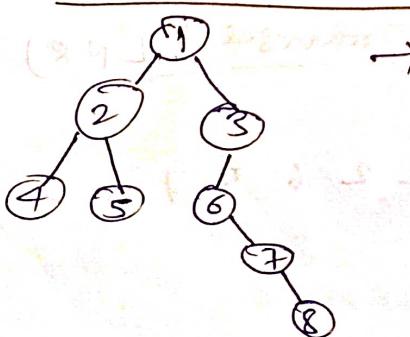
`inorder.push_back(node->data);
node = node->right;`

3	X
7	X
5	X
4	X
2	X
1	X

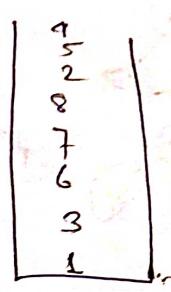
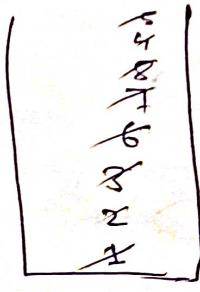
`return inorder;`

1 2 6 5 7 1 3

Iterative method of post order (LRN) :-



$\rightarrow 4 \ 5 \ 2 \ 8 \ 7 \ 6 \ 3 \ 1$



print

```
vector<int> postorder(TreeNode *root)
```

```
{
```

```
    vector<int> postorder;
```

```
    if (root == NULL)
```

```
        return postorder;
```

```
    Stack<int> s1, s2;
```

```
    Stack<TreeNode*> s1, s2;
```

```
s1.push_back;
```

```
s1.push(root);
```

```
while (!s1.empty())
```

```
{
```

```
    root = s1.top();
```

```
s2.push(root);
```

```
if (root->left != NULL)
```

```
{
```

```
s1.push(root->left);
```

```
if (root->right != NULL)
```

```
{
```

```
s1.push(root->right);
```

```
}
```

```
while (!s2.empty())
```

```
{
```

```
postorder.push_back(s2.top());
```

```
s2.pop();
```

```
return postorder;
```

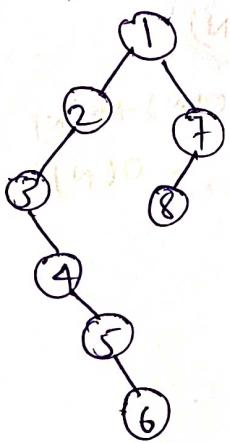
```
}
```

* STACK(S2) की जटि सुनिं दिक्षा अपने बैक पुश करे होंगे। तो
last अ० reverse करके return करेंगे।

T.C - $O(N)$

$$\begin{aligned} S.C &= O(N) + O(N) \\ &= O(N) \end{aligned}$$

Iterative postorder Traversal using 1 stack



6 5 4 3 2 8 7 1

```
while((curr != NULL) || !st.empty())
```

```
{ if(curr == NULL)
```

```
    st.push(curr)
```

```
    curr = curr->left;
```

```
else :
```

```
    temp = st.top() ->right;
```

```
    if (temp == NULL)
```

```
        temp = st.top()
```

```
        st.pop()
```

```
        postOrder(-temp);
```

```
while (!st.empty() || temp == st.top() ->right)
```

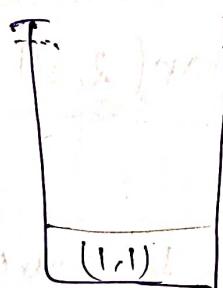
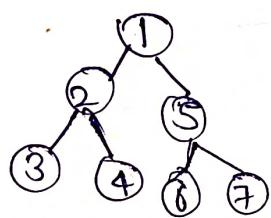
```
    temp = st.top(), st.pop();
```

```
    postOrder(temp->val);
```

```
else
```

```
    curr = temp;
```

Preorder, Inorder, Postorder Traversal in one Traversal



if bnum == 1

Poe order
++
left

if num == 2

in order ✓

++
right

if num == 3
postorder

preorder: 1 2 3 4 5 6 7 (node, num)
inorder: 3 2 1 5 4 6 7

inorder: 3 2 4 16 5 7 6 (minimum)

Postorder: 3 4 2 6 7 5 1

vector<int> PreInPostOrderTraversal(TreeNode *root)

Stacks pair { TreeNode*, int } > st;
st.push ({ root, 1 });

Vector<?> pop, in, post;

if (root == NULL)

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

\circ If its second = 1)

pre.push_back(g1.first->data);

first, second +);

Stopush(i, l);

if (it->first->left == NULL)

st.push({it, first->left, 1});

else if (it.second == 2)

5 In push back (ft. first → data ..)

first second etc

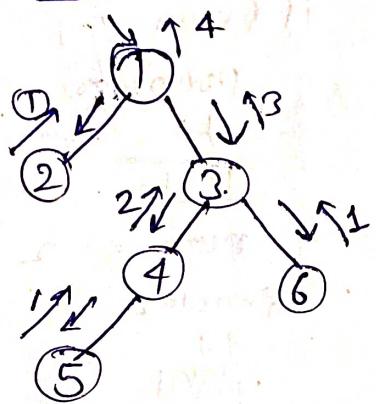
if f p t. st. push (it);

~~Step Push (if first → right, 12).~~

Post-push-back(ft, first->data);

return pre/post/in

Maximum depth of Binary Tree



$$1 + \max(l, r)$$

$$1 + \max(1, 0)$$

$$1 + \max(0, 0)$$

$$1 + \max(2, 1)$$

$$1 + \max(1, 3)$$

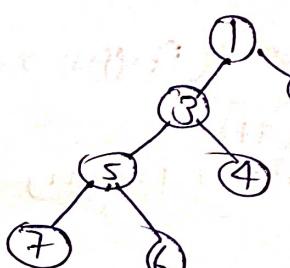
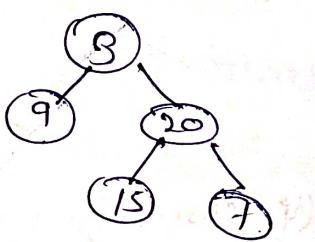
$$= 4 \quad \underline{\underline{Ans}}$$

class Solution

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

Cheaked for Balanced Binary Tree

Balanced BT \rightarrow for every node $height(left) - height(right) \leq 1$



$$3 - 1 = 2$$

X

start Shudo code

Bool check(Node)

{
if (Node == NULL) return true;

int lh = findHeight(Node->left);

int rh = findHeight(Node->right);

if (abs(lh-rh) > 1) return false;

Bool left = check(Node->left);

Bool right = check(Node->right);

if (!left || !right) return false;

return true;

$O(N^2)$

Best sol :-

int isBalanced(TreeNode *root)

{
bool isBalanced(TreeNode *root)

{
return dfsheight(root) != -1;

int dfsheight(TreeNode *root)

TC = $O(N)$

{
if (root == NULL)
{
return 0;
}

SC = $O(N)$

int leftHeight = dfsheight(root->left);

int rightHeight = dfsheight(root->right);

if (leftHeight == -1 || rightHeight == -1)

{
return -1;
}

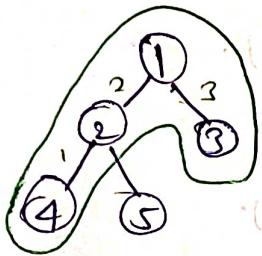
if (abs(leftHeight - rightHeight) > 1)
return -1;

return max(leftHeight, rightHeight) + 1;

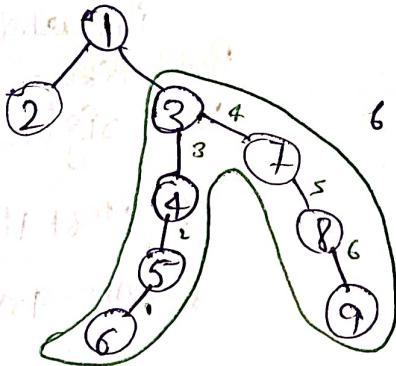
}

Diameter of a Binary Tree

Diameter \rightarrow The longest path between two nodes,
 \rightarrow path does not need to pass via root.



3. root of the left subtree
 4. left child of the left subtree
 5. right child of the left subtree
 6. right child of the right subtree
 7. left child of the right subtree
 8. right child of the right subtree
 9. right child of the right subtree



find max(node)

$O(N^2)$

{ if(node == null)
 return;

lh = findleft(node->left);

rh = findright(node->right);

maxi = Max(maxi, lh+rh);

findmax(node->left);

findmax(node->right);

Better soln

int diameterofBinaryTree(TreeNode* root)

{ int diameter = 0;

height(root, diameter);

return diameter;

int height(TreeNode* root, int & diameter)

{ int dh = height(root); if(root == null, return 0);

int lh = height(root->left, diameter);

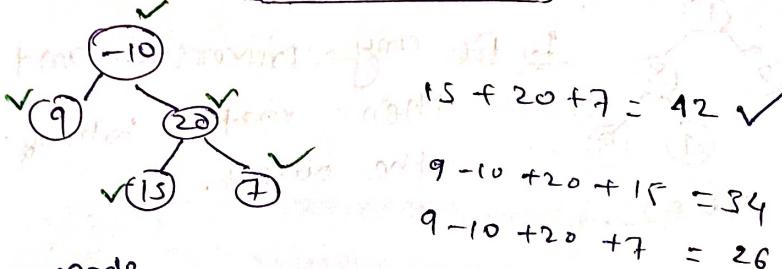
int rh = height(root->right, diameter);

maxi diameter = max(diameter, (lh+rh));

return 1+max(lh, rh);

Maximum path sum

↳ node A → node B



$\max L \rightarrow 0$
 $\max R$

$\Rightarrow \text{Val} + (\max L + \max R)$

$\max = 0 \text{ or } 15 \text{ or } 42$

int pathmaxpath(node, maxi)

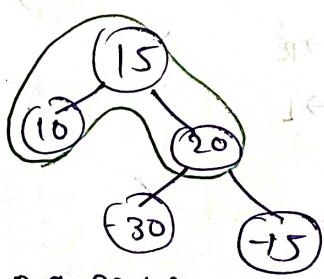
if (node == NULL)
return 0;

leftsum = maxpath(node->left, maxi);
rightsum = maxpath(node->right, maxi);

Max = Max(maxi, leftsum + rightsum +
node->val);

return (node->val) + max(leftsum,
rightsum);

1 test case



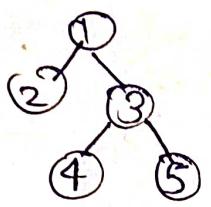
if some path returning -ve
value than make it 0 or ignore them

```
int maxpathsum(TreeNode* root)
{
    int maxi = INT_MIN;
    maxpathdown(root, maxi);
    return maxi;
}
```

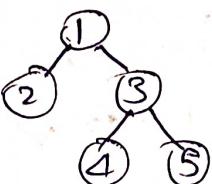
```
int maxpathdown(TreeNode* root, int &maxi)
{
    if (root == NULL) return 0;
    int left = max(0, maxpathdown(root->left, maxi));
    int right = max(0, maxpathdown(root->right, maxi));
    maxi = max(maxi, left + right + node->val);
    return max(left, right) + node->val;
}
```

Check if two trees are identical

or Not



Tree 1



Tree 2

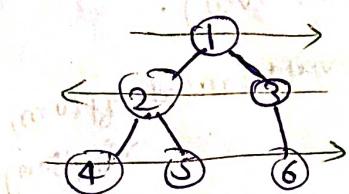
1. Do any Traversal and then match both of the output.

```

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

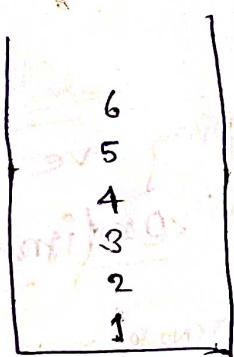
T.C = O(N)
S.C = O(N)

Zig-Zag or Spiral Traversal

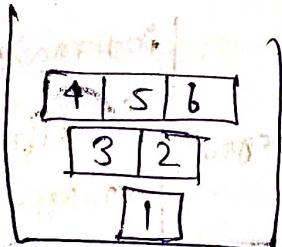


Alt. Same as level order traversal only one difference. Here we use flag variable just for printing in alternate direction.

flag = 0 L → R
flag = 1 R → L



Queue



vector<vector<int>> zigzagtraversal (TreeNode* root)

vector<vector<int>> result;

if (root == NULL)

{ return result; }

}

queue<TreeNode*> nodesqueue;

nodesqueue.push_back(root);

bool lefttoright = true;

while (!nodesqueue.empty()) {

int size = nodesqueue.size();

vector<int> row(size);

for (int i=0 ; i<size; i++)

TreeNode* node = nodesqueue.front();
nodesqueue.pop();

int index = (lefttoright) ? i : (size-1-i);

row[index] = node->val;

if (node->left)

{

nodesqueue.push(node->left);

if (node->right)

{

nodesqueue.push(node->right);

lefttoright = !lefttoright;

result.push_back(row);

return result;

Boundary Traversal

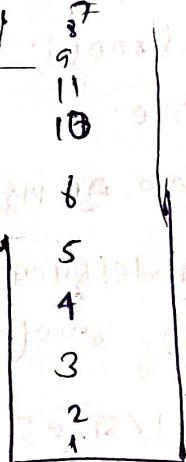
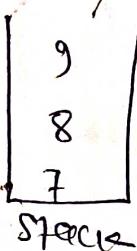
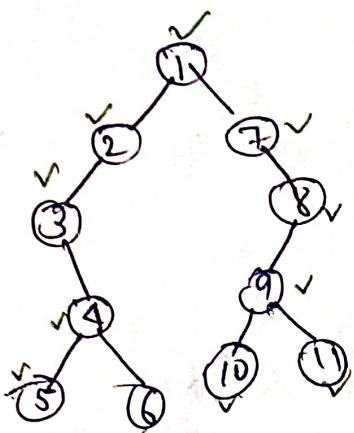
first point:

Left boundary excluding leaf

Leaf nodes

Right boundary on the reverse (excluding leaf)

else stack for reverse
vector



Vector<int> printBoundary (Node *root)

```

{
    Vector<int> res;
    if (root == NULL) return res;
    if (!isLeaf(root))
        res.push_back(root->data);
    addBoundaryLeft (root, res);
    addLeaves (root, res);
    addBoundaryRight (root, res);
    return res;
}
  
```

Bool

isLeaf (Node *root)

```

{
    if (root->left == NULL && root->right == NULL)
        return true;
    else
        false;
}
  
```

```

Void addLeftBoundary( Node* root, vector<int> &res )
{
    Node* curr = root->left;
    while( curr )
    {
        if( !isleaf( curr ) )
            res.push_back( curr->data );
        if( curr->left )
            curr = curr->left;
        else
            curr = curr->right;
    }
}

```

$$\begin{aligned}
T.C. &= O(H) + O(H) + O(N) \\
&= O(N)
\end{aligned}$$

$$S.C. = O(N)$$

```

Void addRightBoundary( Node* root, vector<int> &res )
{
    Node* curr = curr->right;
    stack<ptr> st;
    while( curr )
    {
        if( !isleaf( curr ) )
            st.push( curr->data );
        if( curr->right )
            curr = curr->right;
        else
            curr = curr->left;
    }
    while( !st.empty() )
        res.push_back( st.top() );
        st.pop();
}

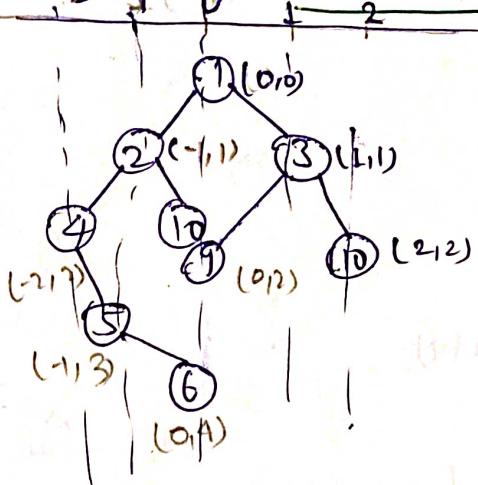
```

```

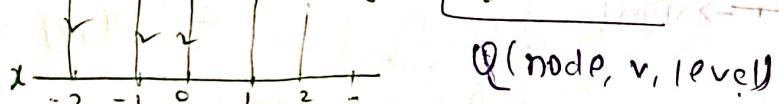
Void addLeaves( Node* root, vector<int> &res )
{
    if( !isleaf( root ) ) res.push_back( root->data );
    return;
    if( root->left )
        addLeaves( root->left, res );
    if( root->right )
        addLeaves( root->right, res );
}

```

¹⁰ Vertical order traversal :-



arrange in the ascending order of x .



-2 4 Map{int, map<int, multiset<int>>}
-1 2 5
0 1 9 10 6 weight

TreeMap (int, TreeMap<int, TreeMap<int, Po>>)

vector <treector<int>> verticleTraversed(TreeNode *root);

```
map<int, map<int, multiset<int>> nodes;
```

queue<pair<TreeNode*, pair<int, int>>> todo;

todo.push({root, \$0, 0});

```
while( ! todo.empty() ) {
```

auto p = todo.front();

TreeNode* node = p.first;

int x = p.second.first, y = p.second.second;

`moder[x][y].insert(moder->val);` \rightarrow multiply it insert ~~as~~

if (node->left) { todo.push(1); }

```
if(node->right) root.push({node->right, {x-1, y+1}}); }
```

19. 19. 19.

function push ({ node }) {
 stack.push({ x + 1, y + 1, node });

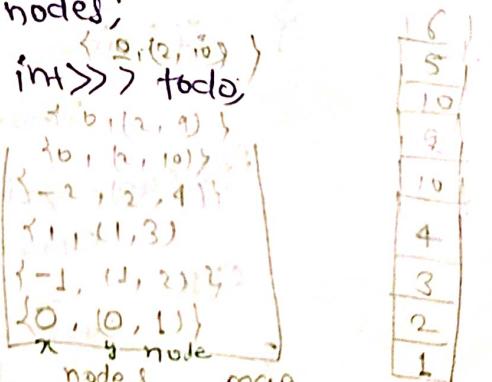
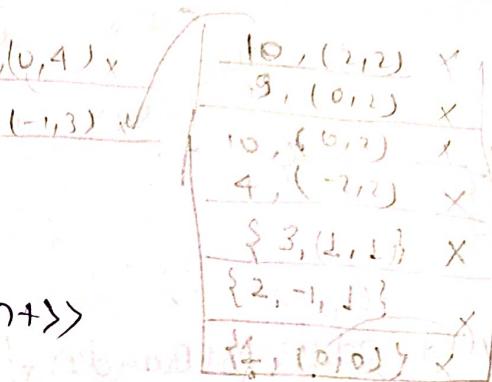
```
set<int> s;
```

Soated &
ring ~~uniggle~~
elements

Multiset<int> s;

or
positj
queue

199
posted on J.



```
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;
```

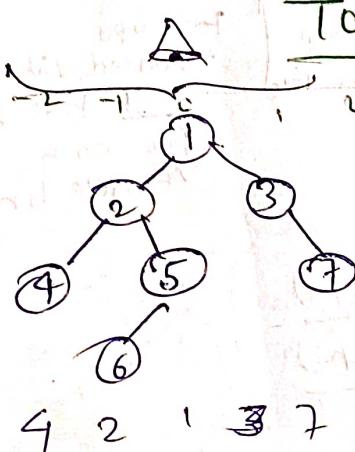
• we are using map so

it will be in sorted order

i.e.

```
{2, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100}
```

Top view of Binary Tree



mode = x / 2 / 3 / 4 / 5 / 6 / 7

```
2 → 7  
-2 → 4  
1 → 3  
-1 → 2  
0 → 1
```

(link, node)

```
(6, -1)  
(7, 2)  
(5, 0) X  
(4, -2) X  
(3, +1) X  
(2, 1) X  
(1, 0) X
```

node level

```
vector<int> topview(Node* root)
```

```
{
```

```
vector<int> ans;
```

```
if (root == NULL)
```

```
{
```

```
    return ans;
```

```
}
```

```
map<int, int> map;
```

```
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 (map.find(line) == map.end())

map[line] = node->data;

if (node->left) q.push({node->left, line-1});

if (node->right) q.push({node->right, line+1});

}

for (auto it : map) {

```
    ans.push_back(it.second);
```

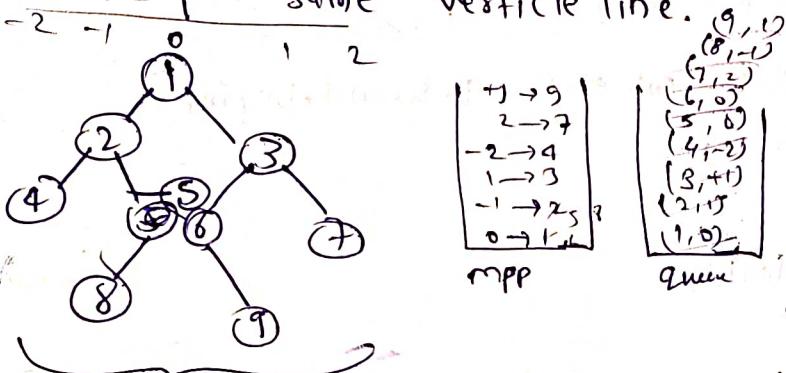
```
}
```

```
return ans;
```

@Aashish Kumar Nayak

Bottom view of Binary Tree

This is same as top view only difference is we will just update map data every time when we face some vertical line.



Top view		Bottom view	
$\begin{array}{l} 1 \rightarrow 9 \\ 2 \rightarrow 7 \\ 3 \rightarrow 4 \\ 4 \rightarrow 3 \\ 5 \rightarrow 2 \\ 6 \rightarrow 1 \\ 7 \rightarrow 0 \end{array}$ mpp		$\begin{array}{l} 1 \rightarrow 9 \\ 2 \rightarrow 7 \\ 3 \rightarrow 4 \\ 4 \rightarrow 3 \\ 5 \rightarrow 2 \\ 6 \rightarrow 1 \\ 7 \rightarrow 0 \end{array}$ queue	

only first assigned value of map for each vertical line.

last updated value of map till last for each vertical line.

vector<int> bottomview(Node* root)

```
{
    vector<int> ans;
    if(root == NULL)
        return ans;
    }
```

```

    map<int, int> mpp;
    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;
        mpp[line] = node->data;
    }
}
```

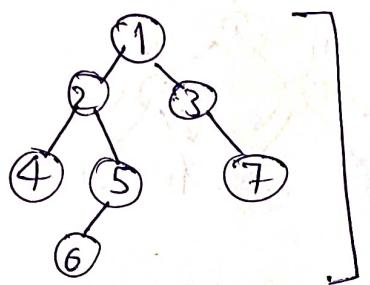
TC - O(N)
SC - O(N)

```

        if(node->left) q.push({node->left, line-1});
        if(node->right) q.push({node->right, line+1});
    }
    for(auto it : mpp) {ans.push_back(it.second);}
}
return ans;
}
```

Right/left side view

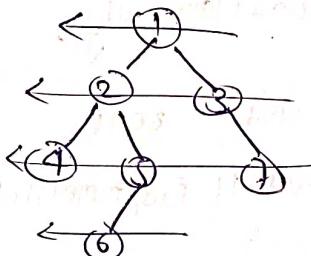
① Aashish Kumar Nayak



1 3 7 6

Last node of every level
is indeed my Right side view.

or if we traverse like this:



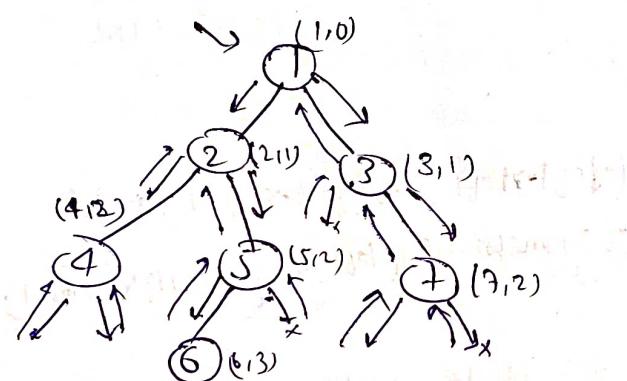
then 1st node of each level
is my right side view.

[Recursive]

TC $\rightarrow \Theta(N)$
SC $\rightarrow O(N)$

[Iterative]

level order
TC $\rightarrow O(N)$
SC \rightarrow



Right view

```
vector<int> rightview(node* root) {
    vector<int> res;
    recursion(root, 0, res);
    return res;
}

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

6
7
3
1

f(node, level)

```
if (node == NULL)
    return;
if (level == ds.size())
    ds.add(node);
f(node->right, level + 1);
f(node->left, level + 1);
```

Left view

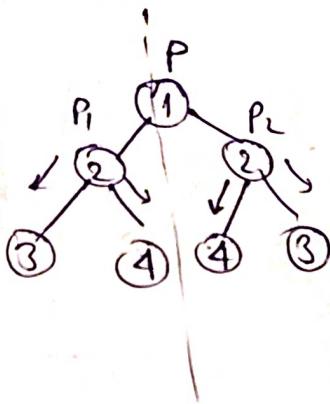
```
vector<int> leftview(node* root) {
    vector<int> res;
    recursion(root, 0, res);
    return res;
}

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

Symmetric Binary Tree

It forms a mirror of it self.

P₁ we will iterate P₁ towards right while P₂ towards left and then P₁ towards left & P₂ towards right simultaneously.



bool issymmetric(TreeNode* root)

{
return root == NULL || issymmetricHelp(root->left, root->right);
}

bool issymmetricHelp(TreeNode* left, TreeNode* right)

{
if (left == NULL || right == NULL)
 return left == right;
if (left->val != right->val)
 return false;

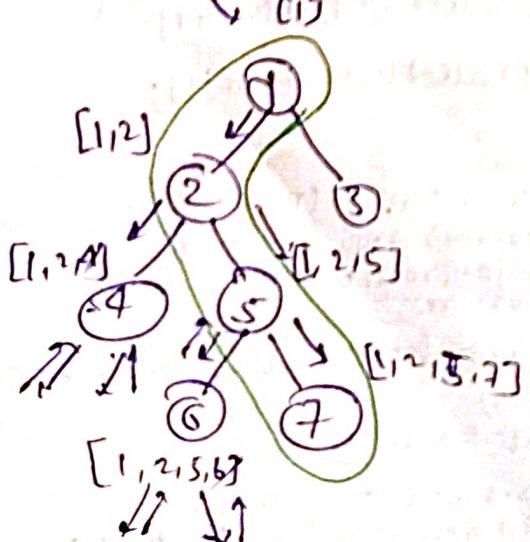
T.C. = O(N)
S.C. = O(1)

return issymmetricHelp(left->left, right->right) &&

issymmetricHelp(left->right, right->left);
}

}

Point root to Node Path in BT



NODE = 7

1 2 5 7

Inorder traversal
Recursive approach
Stack, space

T.C. = O(N)

S.C. = O(N)

4
2
1

@Aashish Kumar Nayak

```

bool getpath(TreeNode* root, vector<int>&arr, int x)
{
    if (root == NULL)
        return false;
    arr.push_back(root->val); ✓ O(n)
    if (root->val == x)
        return true;
    if (get if (getpath(root->left, arr, x) || getpath(root->right, arr, x)))
        return true;
    arr.pop_back(); ✓ O(n)
    return false;
}

```

$\text{vector} < \text{int} >$ solve (TreeNode * A, int B)

```

{
    vector<int> arr;
    if (A == NULL)
        return arr;

```

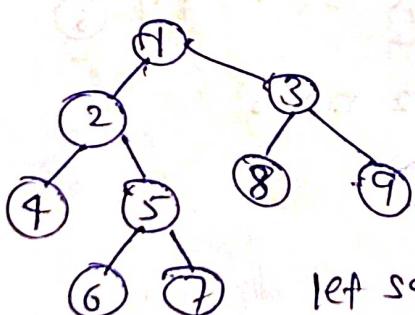
```

    getpath (A, arr, B);
    return arr;
}

```

Lowest common Ancestor (Binary Tree)

The ancestor that exists at deepest level.



$$\begin{aligned} \text{lca}(4, 7) &= 2 \\ \text{lca}(5, 8) &= 3 \\ \text{lca}(2, 6) &= 2 \end{aligned}$$

Let say $\text{lca}(4, 7)$

Path node = 4 ↓ matching
 Path node = 7 ↓ (1 2 4)
 Path node = 7 (1 2 5 7)

(a) Aashish Kumar Nayak

TreeNode* LowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)

```
{ if (root == NULL || root == p || root == q)
    { return root;
    }
```

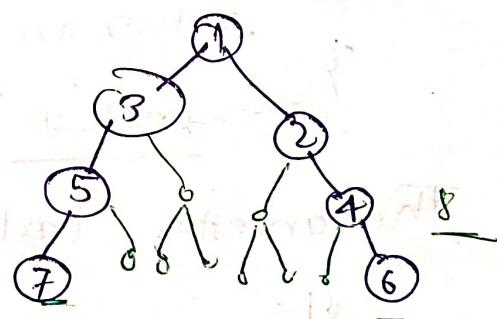
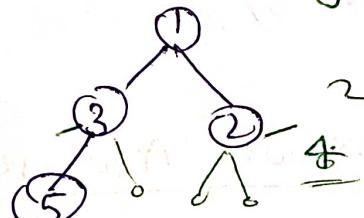
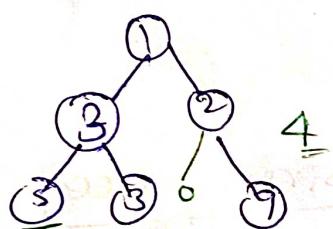
TreeNode* left = LowestCommonAncestor(root->left, p, q);

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

```
if (left == NULL)
{ return right;
}
```

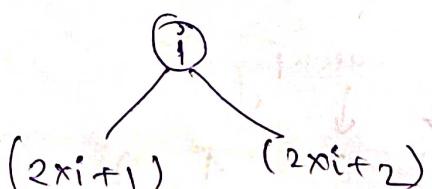
```
TreeNode* left;
else if (right == NULL)
{ return left;
}
else
{ return root;
}
```

Maximum width of Binary Tree

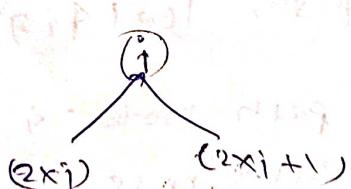


width \Rightarrow no. of nodes ^{in a level} between any 2 nodes

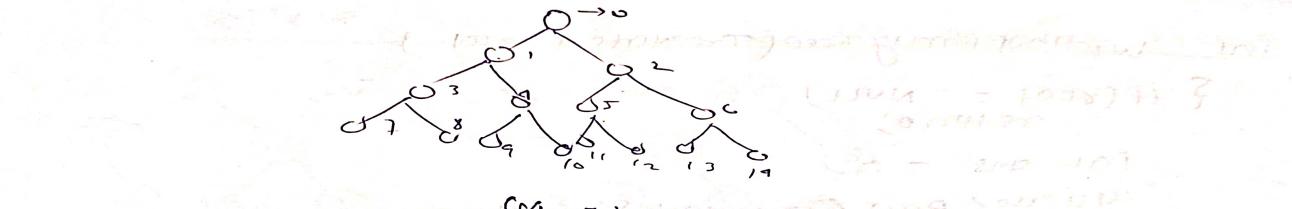
Think about: never order traversal



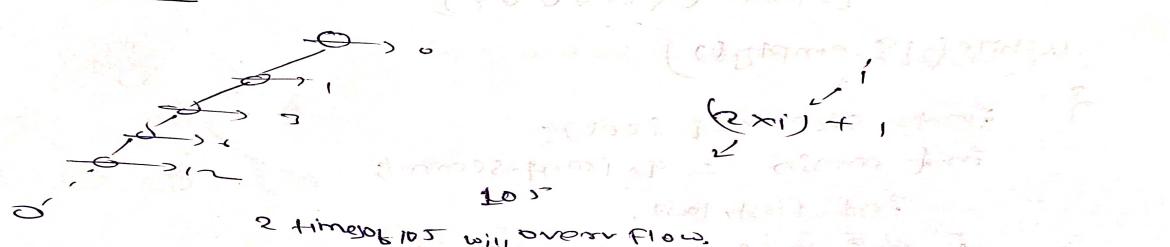
0-based indexing



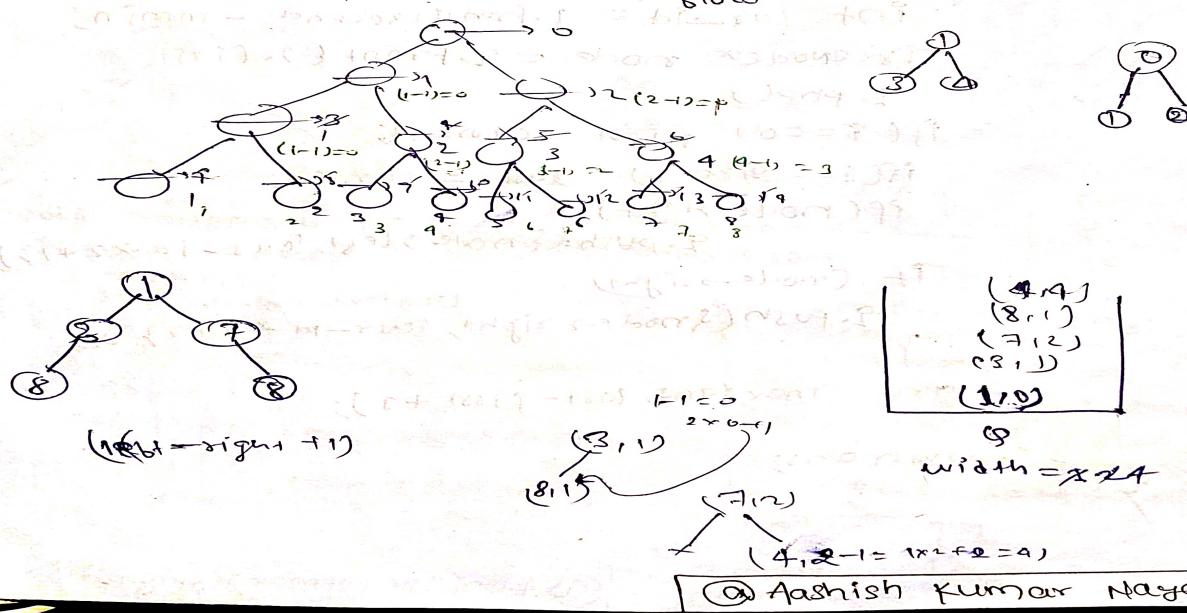
1-based indexing.



Failed test case



in order to reduced overflow



© Aashish Kumar Nayak

fun widthBinaryTree(TreeNode* root)

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

```
while(!q.empty())
```

```
{ int size = q.size();
int mmin = q.front().second;
int first, last;
```

```
for(int i=0; i<size; i++)
```

```
{ int cur_id = q.front().second - mmin;
TreeNode* node = q.front().first;
```

```
q.pop();
```

```
if(i==0) first = cur_id;
```

```
if(i==size-1) last = cur_id;
```

```
if(node->left)
```

```
q.push({node->left, cur_id * 2 + 1});
```

```
if(node->right)
```

```
q.push({node->right, cur_id * 2 + 2});
```

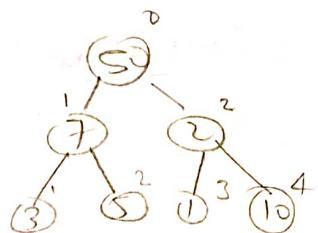
```
ans = max(ans, last - first + 1);
```

```
} return ans;
```

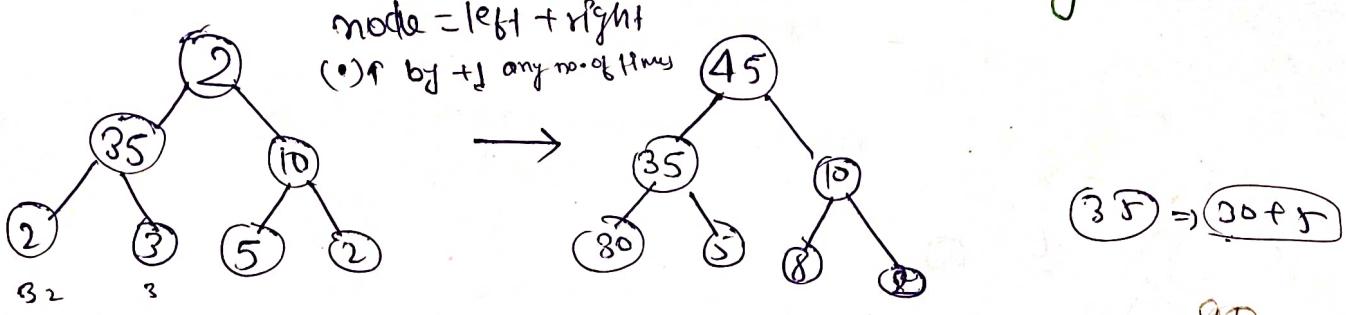
ans = 12
cur_id = 0

mmin = 0!

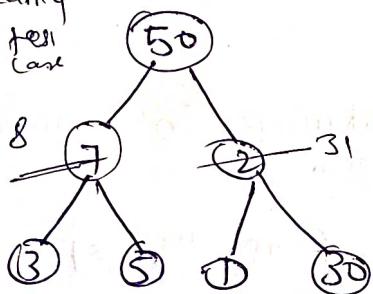
④



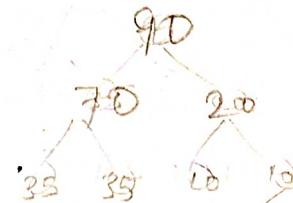
Children Sum Property in Binary Tree



failed
test case



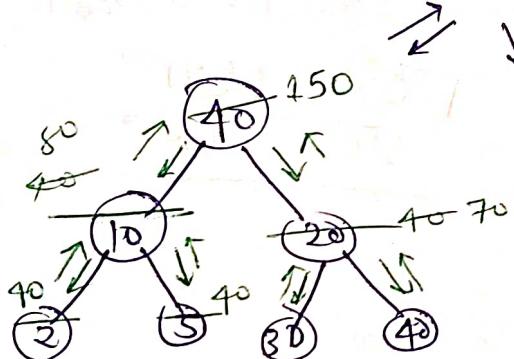
bifig and all solutions are $O(N^2)$.



we will solve in $O(N)$.

solution

70	80
30	X
40	X
30	X
40	X
40	



40

$$10 + 20 = 30 < 40$$

$$2 + 5 = 7 < 40$$

$$30 + 40 = 70 > 40$$

T.C. - $O(N)$

S.C.

void ~~change~~ reorder (BinaryTreeNode* root, int tot = 0001)

```

    if (root == NULL)
        return;
    if (root->left)
        child += root->left->data;
    if (root->right)
        child += root->right->data;
    if (child >= root->data)
        tot->data = child;
    else {
        if (root->left)
            tot->left->data = root->data;
        if (root->right)
            tot->right->data = root->data;
    }
}

```

```

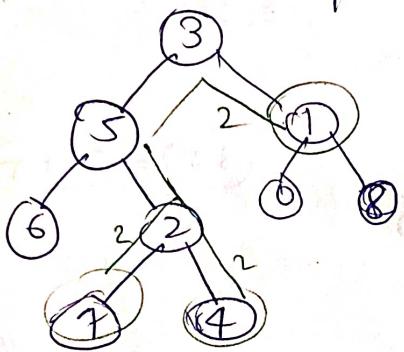
reorder (root->left);
reorder (root->right);
int tot = 0;
if (root->left) tot += root->left->data;
if (root->right) tot += root->right->data;
if (root->left || root->right)
    root->data = tot;
}

```

@Aashish Kumar Nagarkar

point all nodes at a distance of K

$K=2$, target = 5

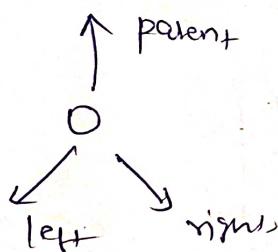


output = 7, 4, 1

BF

We will make a queue and hashmap in which we will connect all nodes with their parents.

then using BFS to go upto K level from target node using our hashtable info



we will go in three direction after till K distance reached upto distance K

$$T.C. = O(N)$$

$$S.C. = O(N)$$

```
if(current->right && !visited[current->right])  
    {  
        queue.push(current->right);  
        visited[current->right] = true;  
  
        if(parent_track[current] && !visited[parent_track[current]])  
            queue.push(parent_track[current]);  
        visited[parent_track[current]] = true;  
    }  
  
for loop  
while  
vector<int> result;  
while(!queue.empty())  
{  
    treenode* current = queue.front(); queue.pop();  
    result.push_back(current->val);  
}  
return result;
```

```
void markParents(TreeNode* root, unordered_map<TreeNode*, TreeNode*>& parent_track, TreeNode* target)
```

```
{  
    queue<TreeNode*> queue;  
    queue.push(root);  
    while (!queue.empty())  
    {  
        TreeNode* current = queue.front();  
        queue.pop();  
        if (current->left) { queue.push(current->left);  
            parent_track[current->left] = current; }  
        if (current->right) { queue.push(current->right);  
            parent_track[current->right] = current; }  
    }  
}
```

```
vector<int> distancek(TreeNode* root, TreeNode* target, int k)
```

```
{  
    unordered_map<TreeNode*, TreeNode*> parent_track;  
    markParents(root, parent_track, target);
```

```
unordered_map<TreeNode*, bool> visited;
```

```
queue<TreeNode*> queue;
```

```
queue.push(target);
```

```
visited[target] = true;
```

```
int curr_level = 0;
```

```
while (!queue.empty())
```

```
{  
    int size = queue.size();
```

```
    if (curr_level++ == k) break;
```

```
    for (int i=0; i<size; i++)
```

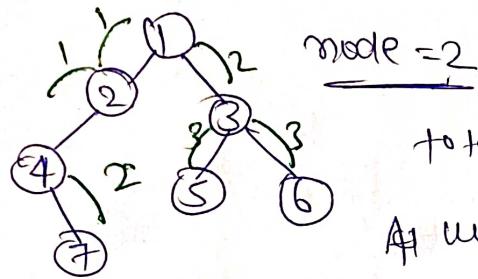
```
{  
    TreeNode* current = queue.front();  
    queue.pop();
```

```
    if (current->left && !visited[current->left])
```

```
{  
    queue.push(current->left);
```

```
    visited[current->left] = true;
```

Minimum Time taken to Burn a Binary Tree from a node/leaf node

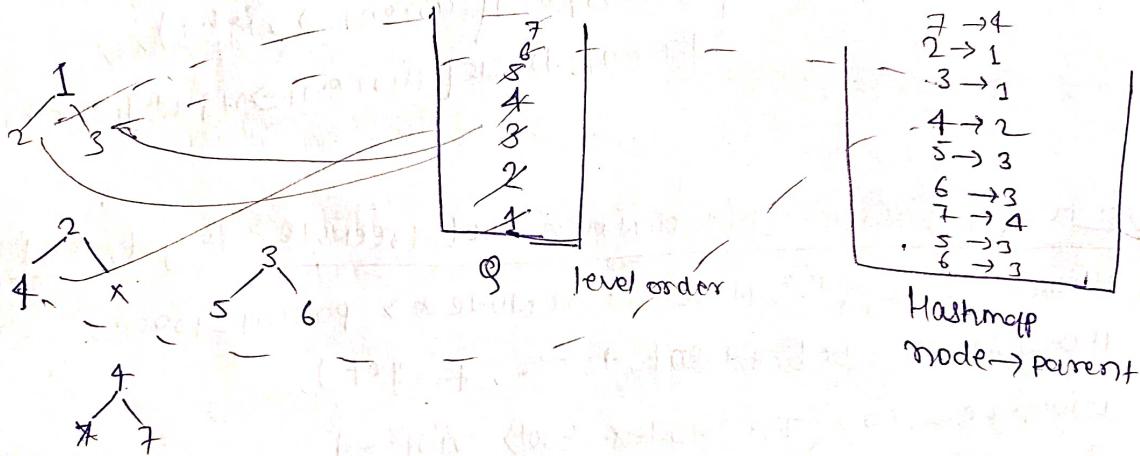


@Aashish Kumar Nayak

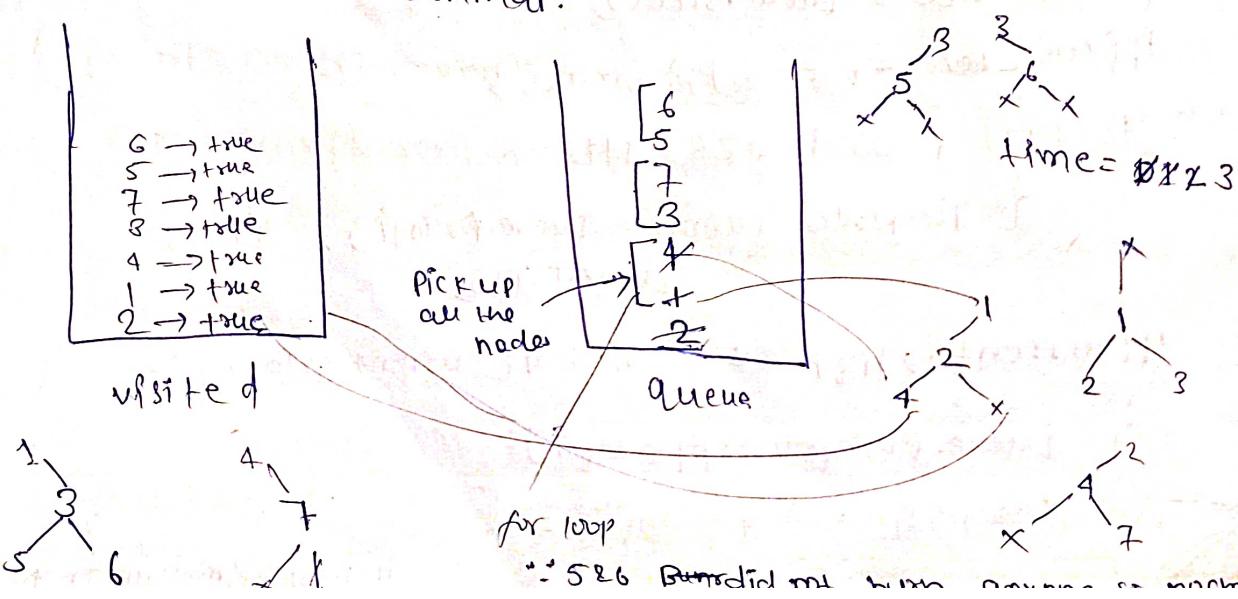
total 3 sec ही तक burn हो जायेगा।।।

As we will apply BFS just like previous

1st step We can't move upward so to solve this issue we will assign parent pointers



Step 2 Let again do a BFS traversal and take again a queue data structure and this time target node will be inserted first and the same time we have gone have visited hashmap.



so no increase in time.
 and queue becomes empty
 so when queue becomes empty
 return time any

it is kind of level wise movement so we can't use DFS. only BFS.

Code:

$$\begin{aligned}
 T.C. &= O(N) + O(N) \\
 &\approx O(N)
 \end{aligned}$$

```

int timeToBurnTree (BinaryTreeNode<int>* root, int start)
{
    map<BinaryTreeNode<int>*, BinaryTreeNode<int>*> mpp;
    BinaryTreeNode<int>* target = bfsMapParent(root, mpp, start);
    int maxi = findmaxdistance(mpp, target);
    return maxi;
}

BinaryTreeNode<int>* bfsMapParent (BinaryTreeNode<int>* root,
map<BinaryTreeNode<int>*, BinaryTreeNode<int>*> &mpp, int start)
{
    queue<BinaryTreeNode<int>*> q;
    q.push(root);
    BinaryTreeNode<int>* res;
    while (!q.empty())
    {
        BinaryTreeNode<int>* node = q.front();
        q.pop();
        if (node->data == start) { res = node; }
        if (node->left) { q.push(node->left); mpp[node->left] = node; }
        if (node->right) { q.push(node->right); mpp[node->right] = node; }
    }
    return res;
}
    
```

```

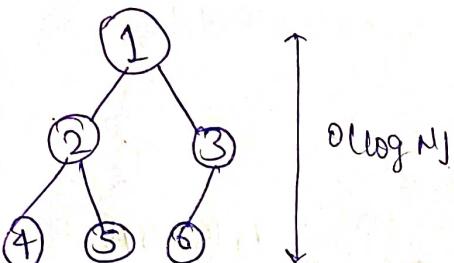
int findMaxDistance(mpp<BinaryTreeNode<int>*, BinaryTreeNode<int>*>
&mpp, BinaryTreeNode<int>* target)
{
    Queue<BinaryTreeNode<int>*> q;
    q.push(target);
    mpp<BinaryTreeNode<int>, bool> vis;
    vis[target] = true;
    int maxi = 0;

    while (!q.empty())
    {
        int sz = q.size();
        int fl = 0;
        for (int i = 0; i < sz; i++)
        {
            auto node = q.front();
            q.pop();
            if (node->left && !vis[node->left])
            {
                fl = 1;
                vis[node->left] = true;
                q.push(node->left);
            }
            if (node->right && !vis[node->right])
            {
                fl = 1;
                vis[node->right] = true;
                q.push(node->right);
            }
            if (mpp[node] && !vis[mpp[node]])
            {
                fl = 1;
                vis[mpp[node]] = true;
                q.push(mpp[node]);
            }
        }
        if (fl) maxi++;
    }
    return maxi;
}

```

Count total nodes in a complete Binary Tree

Complete Binary Tree: Every level except possibly the last, is completely filled in a Complete Binary Tree, and all node nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h .



TC - O(N)
ASC → O(h)

Because it is a complete BT.

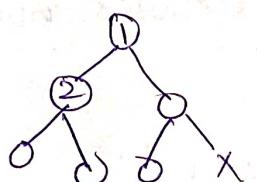
Brute force
inorder(node, &count)

```
if (root == NULL)
    return;
```

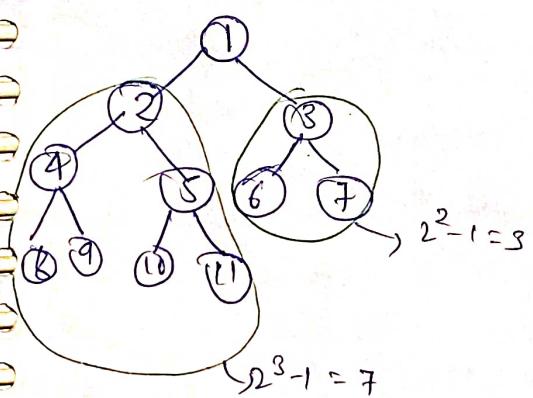
count++;

inorder(root->left);
 inorder(root->right);

property of complete BT
will be $\frac{2^n - 1}{3}$

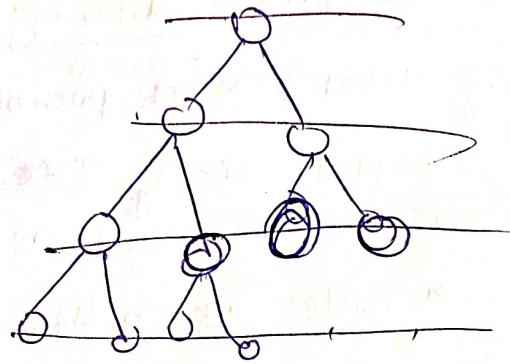


still it is complete BT



$$1 + 3 + 7 = 11$$

We will count ^{node} for every subtree
then we will add.



```
int countNodes(TreeNode* root)
{
    if (root == NULL)
    {
        return 0;
    }
}
```

```
int lh = findheightLeft(root);
int rh = findheightRight(root);
if (lh == rh)
{
    return (1 << lh) - 1;
}
return 1 + countNodes(root->left) + countNodes(root->right);
```

```
int findheightLeft(TreeNode* node)
```

```
{
    int height = 0;
    while (node)
    {
        height++;
        node = node->left;
    }
    return height;
}
```

```
int findheightRight(TreeNode* node)
```

```
{
    int height = 0;
    while (node)
    {
        height++;
        node = node->right;
    }
}
```

```
return height;
```

$$T(n) = O((\log n)^2)$$

$$\log N \times \log N$$

for height So C. - Recursive space
 \downarrow
 $O(N)$

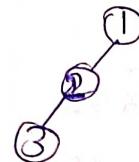
at any instant
 the tree we will traverse
 $\log N$

Requirements needed to construct a unique Binary Tree :-

Q. Can you construct a unique BT with given, (a) preorder & postorder following.

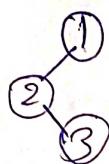
(PNR) preorder

1 2 3
3 2 1



(LRN) postorder

Preorder Postorder
1 2 3
3 2 1



We can create B.T. with preorder & postorder but can't create unique B.T.

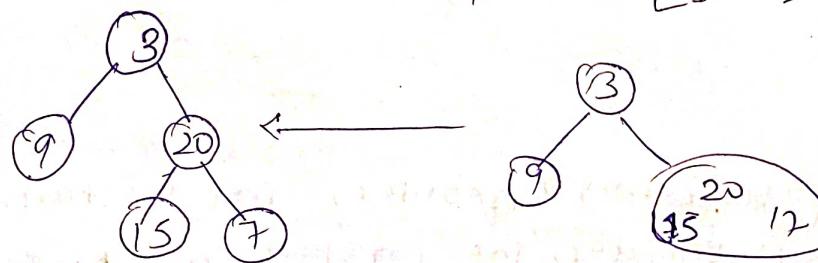
Note if given

Inorder & preorder

It is possible to create unique B.T. using inorder & preorder

Inorder [9 3 15 20 7]

Preorder [3 9 20 15 7]



To create a unique B.T. it is necessary to

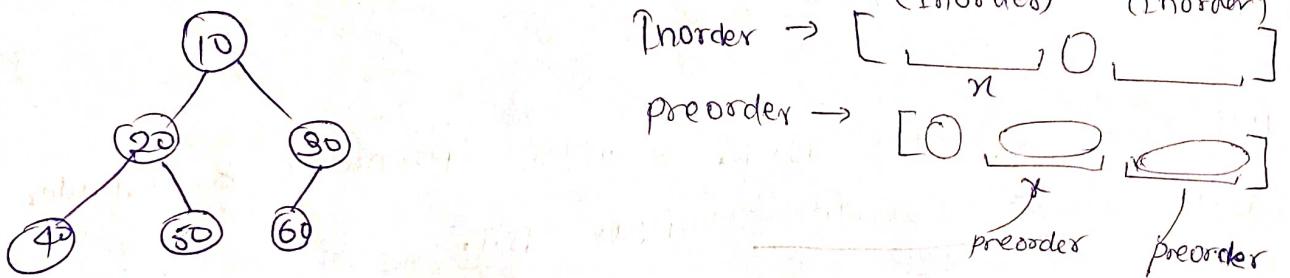
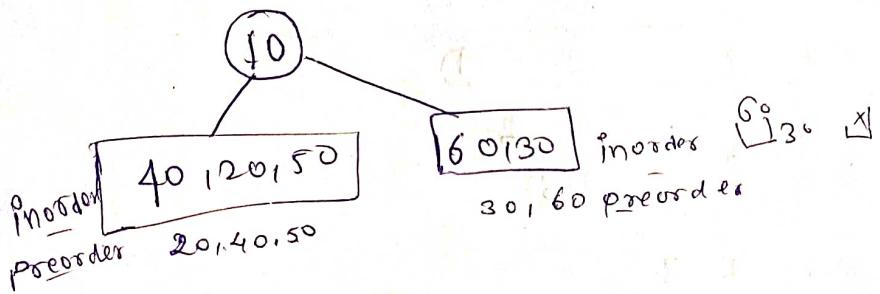
root and about its left and right.

so we can use inorder & preorder to create unique B.T.

~~CONTINUE~~ Construct Binary Tree from Inorder & Preorder

(LNR) Inorder - [40 20 50 10] ↓
 Left Right
 [60 30]

(NLR) Preorder - [10 20 40 50 30 60]
 Root



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

```
    for(int i=0; i<inorder.size(); i++)
        if(inmap[inorder[i]] == i)
            }
```

```
    TreeNode* root = buildTree(preorder, 0, preorder.size() - 1,
        inorder, 0, inorder.size() - 1, inmap);
```

```
    return root;
}
```

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

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

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

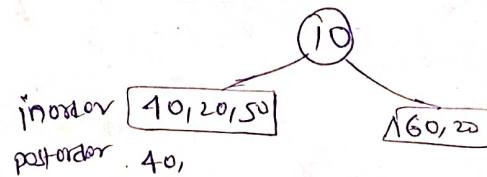
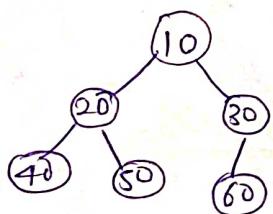
```

int inroot = inmap[roott->val];
int numleft = inroot - instart;
root->left
roott->left = buildTree(preorder, prestart + 1, prestart + numleft, inorder, instart, inroot - 1, inmap);
roott->right = buildTree(preorder, prestart + numleft + 1, prestart + numleft + 1, inorder, inroot + 1, inEnd, inmap);
return roott;
}

```

Construct a Binary Tree from postorder and Inorder

Inorder - [40 20 50 10] [60 80] (LNR)
 postorder - [40 50 20] [60 30 10] (LRN)



```

TreeNode* buildTree(vector<int> &inorder, vector<int> &postorder)
{
    if(inorder.size() != postorder.size())
        return NULL;
    map<int, int> hm;
    for(int i=0; i<inorder.size(); ++i)
    {
        hm[inorder[i]] = i;
    }
    return buildTreePostIn(postorder, 0, inorder.size() - 1,
                           postorder, 0, postorder.size() - 1, hm);
}

```

TreeNode* buildTreePostIn(vector<int> &inorder, int is, int ie,
vector<int> &postorder, int ps, int pe, map<int, int> &hm)

{

if(ps > pe || is > ie)
return NULL;

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

int inRoot = hm[postorder[pe]];

int numLeft = ie - is;

root->left = buildTreePostIn(inorder, is, inRoot - 1, postorder,

ps, ps + numLeft - 1, hm);

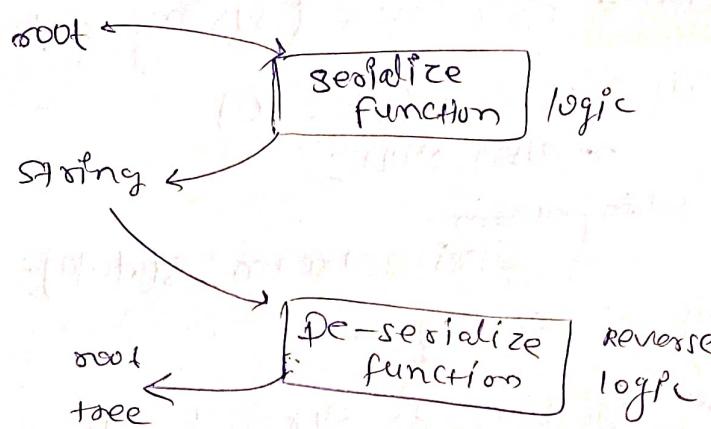
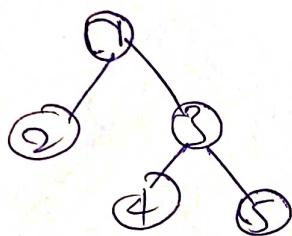
root->right = buildTreePostIn(inorder, inRoot + 1, ie, postorder,

ps + numLeft, pe - 1, hm);

return root;

{}

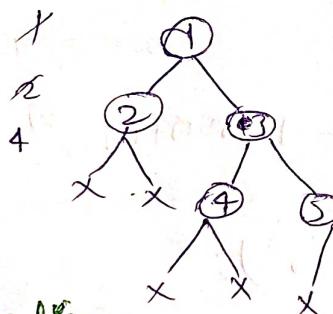
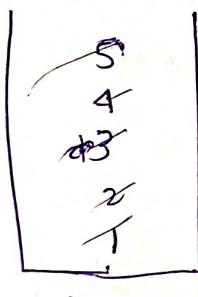
Serialize and De-Serialize Binary Tree



level order traversal

String : $\frac{1}{\downarrow} \frac{2}{\downarrow} \frac{3}{\downarrow} \frac{\#,\#}{\downarrow} \frac{4}{\downarrow} \frac{5}{\downarrow} \frac{\#,\#,\#}{\downarrow} \frac{\#}{\downarrow}$

NULL



If it's same like level order traversal,
only difference is
including # in place of NULL.

Serialize

String serialize(TreeNode* root)

```

if(root == NULL)
    { return "";}

string s = "";
queue<TreeNode*> q;
q.push(root);
while(!q.empty())
{
    TreeNode* currNode = q.front();
    q.pop();
    if(currNode == NULL) s.append("#,");
    else s.append(to_string(currNode->val) + ',');

    if( currNode != NULL )
    {
        q.push(currNode->left);
        q.push(currNode->right);
    }
}
return s;
  
```

Decodes or De-serialize

TreeNode* deserialize(string data)

{
if (data.size() == 0)
 return NULL;

String stream
StringStream s(data); // string to be iterated over
as a object.
String stream
is a class
used for insertion
& extraction of
data to/from
string object.
String Stream
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* leftNode = new TreeNode(stoi(str));
node->left = leftNode;
q.push(leftNode);

}

getline(s, str, ',');

if (str == "#")

{
node->right = NULL;

else {

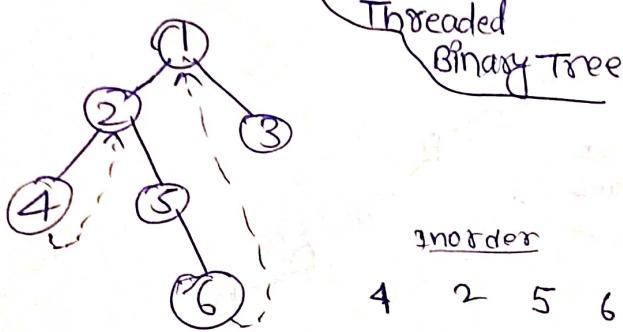
TreeNode* rightNode = new TreeNode(stoi(str));
node->right = rightNode;
q.push(rightNode);

}

} return root;

Morris Traversal (Inorder) / preorder

Concept of



T.C. $O(N)$
S.C. $O(1)$

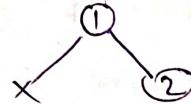
But

Morris traversal

T.C. - $O(N)$

S.C. - $O(1)$

1st case



left \rightarrow null
point(1)
go \rightarrow right

2nd case

left \leftarrow right most guy on left subtree
 \downarrow
curr & curr = curr \leftarrow left
 \downarrow
curr \rightarrow make it null exists \rightarrow remove thread

on the right most guy of left subtree if they don't have thread then make thread & curr = curr \leftarrow left. But if it has thread then remove it.
 \downarrow
and move to right
 \downarrow
curr = curr \rightarrow right.
 \downarrow
if should not be pointing to itself.

cut the thread

for preorder just 1 line change

~~preorder.push_back(curr.val);~~
preorder.push_back(curr.val);
Remove from here & put there

vector<int> inorder;

Tree.

Vector<int> getInorder(TreeNode* root)

TreeNode* cur = root;

while (cur != NULL)

if (cur->left == NULL)
inorder.push_back(cur->val);
cur = cur->right;
 \downarrow
else

TreeNode* prev = cur->left;
while (prev->right == prev->right)

prev = prev->right;

if (prev->right == NULL)

prev->right = cur;
cur = cur->left;

else

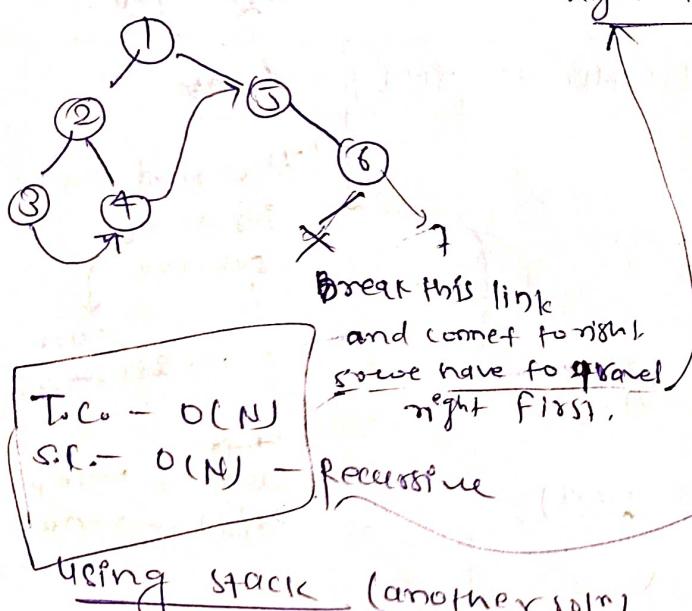
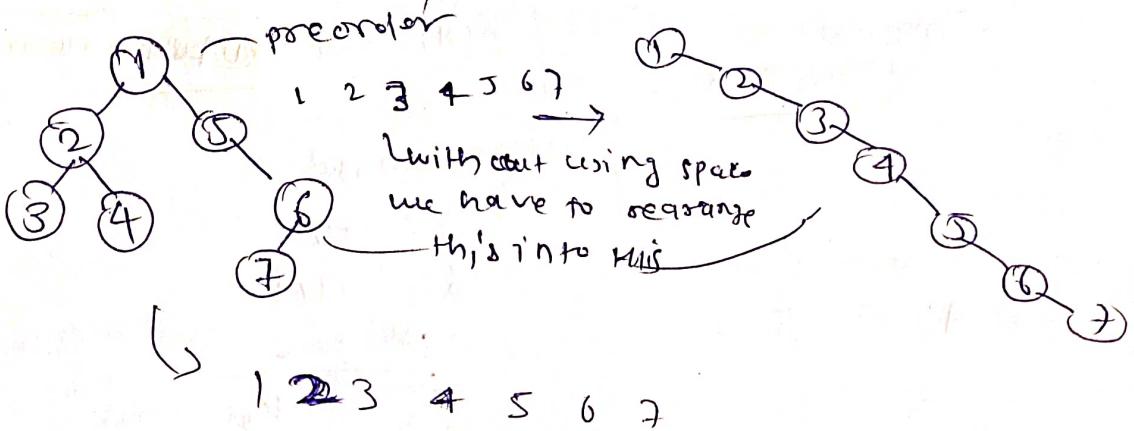
prev->right = NULL;

inorder.push_back(cur->val);

cur = cur->right;

return inorder;

Flattening a Binary Tree to Linkend List



```

S1. push (root)
while (!st.empty())
{
    cur = st.top();
    st.pop();
    if (cur → right)
        st.push (cur → right);
    if (cur → left)
        st.push (cur → left);
    if (!st.empty())
        cur → right = st.top();
    cur → left = null;
}
  
```

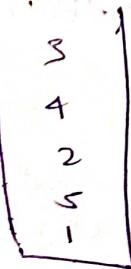
Right left Root → we can say it
Reverse post order

prev = null
flatten (node)

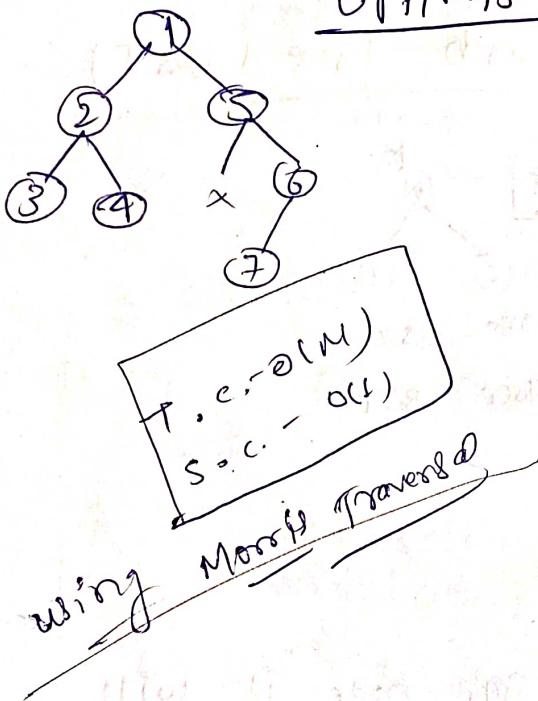
```

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

T.C. = $O(N)$
S.C. = $O(N)$



Optimized way



curr = root

while(curr != null)

{ if(curr->left != null)

 prev = curr->left;

 while(prev->right)

 prev = prev->right;

 prev->right = curr->right;

 curr->right = curr->left;

 curr = curr->right;

}

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

 }

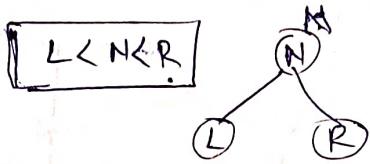
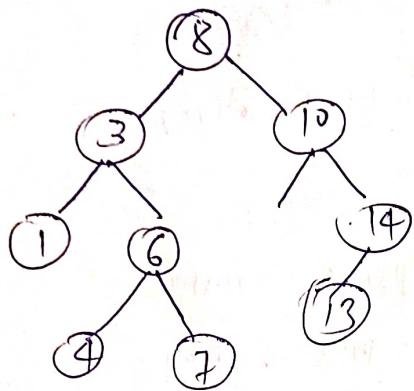
 }

 }

 }

 }

Introduction to Binary Search Tree (BST)



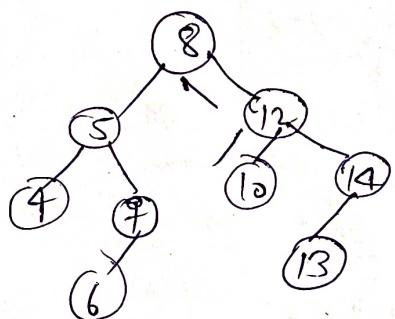
- (*) left subtree \rightarrow BST
- (*) right subtree \rightarrow BST

Why BST

If we want to search any node it will take $O(N)$ in BT, while $O(\log_2 n)$ in BST.

Search in a Binary Search Tree (BST)

node = go



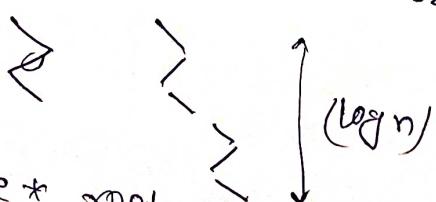
We will check given value to the node value if it is less then move to left part if it is greater, move to right part or if it is equal then return.

i.e. we will not traversing all node we are basically traversing height of the tree i.e. $O(\log_2 n)$

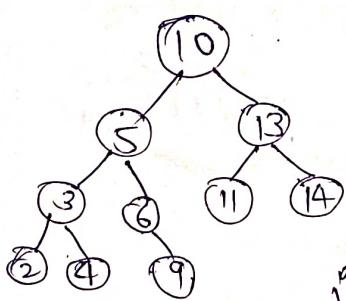
Code :-

```

TreeNode* searchBST (TreeNode* root, int val)
{
    while (root != NULL && root->val != val)
        if (root->val < val)
            root = root->right;
        else
            root = root->left;
    return root;
}
  
```



ceil in a Binary Search Tree



if
key = 8
ans = 9

Val ≥ 8

key = 11, Ans = 11

we have to find
the smallest no. which is largest than 8.
or equal to 8.

int findceil(BinarySearchTNode<int> *root,
int key)

int ceil = -1;

while (root)

{ if (root->data == key)

{ ceil = root->data;

return ceil;

if (key > root->data)

{ root = root->right;

else

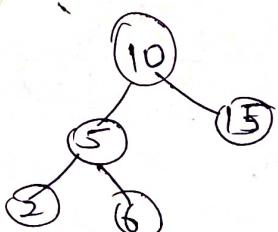
{ ceil = root->data;

root = root->left;

}

Floor in a Binary search Tree

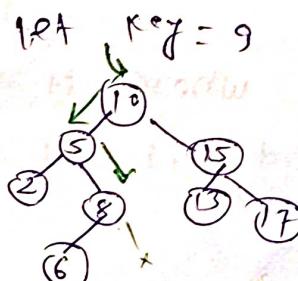
We have to search which the greatest value which is smaller or equal to key value.



key = 7, Ans = 6

key = 14, Ans = 10

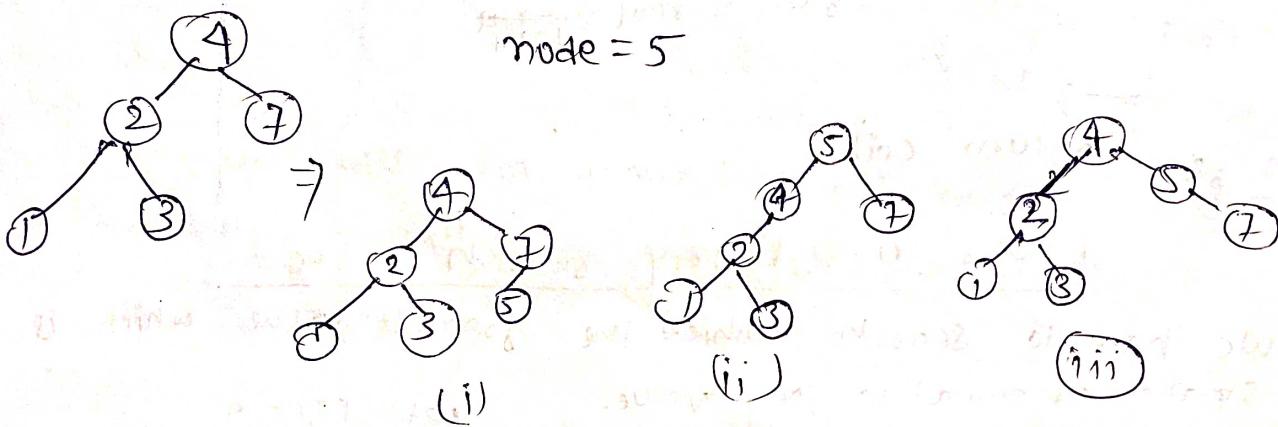
key = 9, Ans = 6



Code

```
int floorBST(Treenode<int> *root, int key)
{
    int floor = -1;
    while(root)
    {
        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 node in BST



find where it can be insert
and this will be always a leaf position.

TreeNode* insertBST

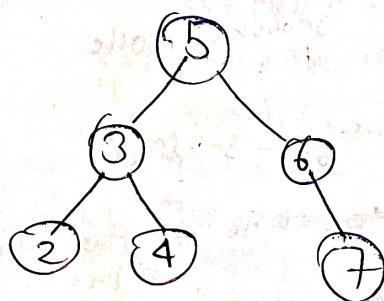
TreeNode* insertIntoBST(TreeNode* root, int val)

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

TreeNode* cur = root;
while(true)
    if(cur->val == val)
        if(cur->right == NULL)
            cur = cur->right;
        else
            cur->right = new TreeNode(val);
            break;
    else
        if(cur->left != NULL)
            cur = cur->left;
        else
            cur->left = new TreeNode(val);
    }

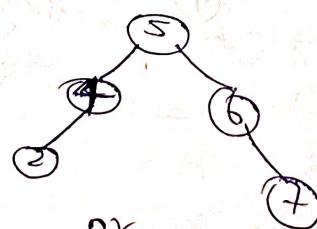
return root;
```

Delete a Node in BST

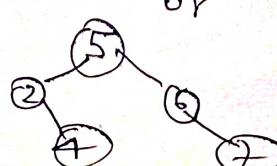


→ Delete
→ rearrange
→ return

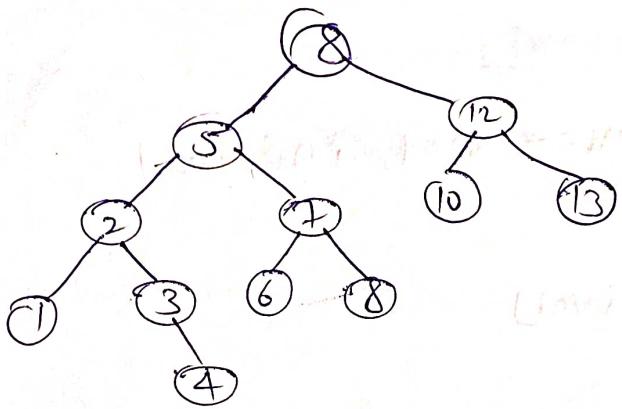
mode = 3



or

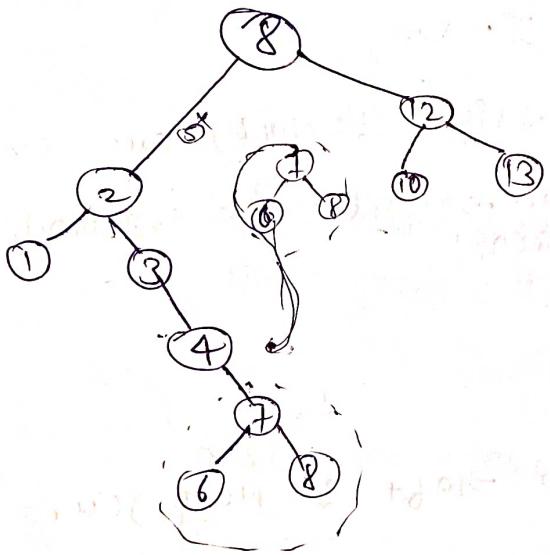


- (i) search node
(ii) Delete node

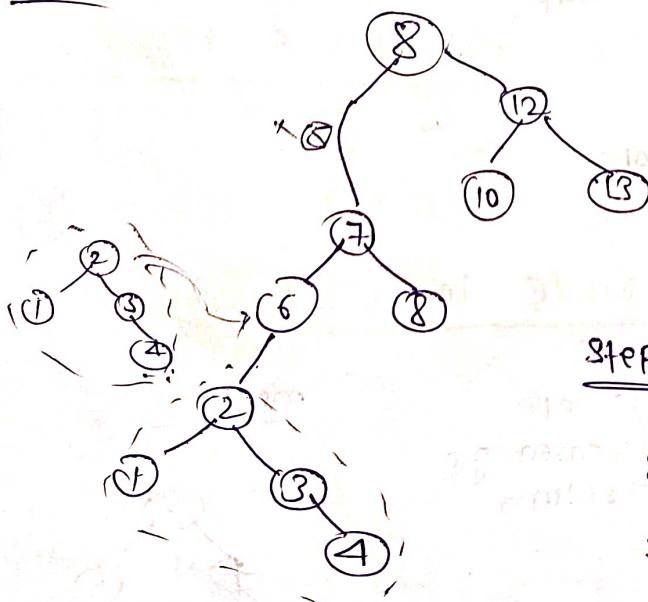


node = 5

1st way



2nd way



Steps :- search node

$8 \rightarrow \text{left} = 5 \rightarrow \text{left}$

~~5 → right~~ go to the extreme

$4 \rightarrow \text{right} = 5 \rightarrow \text{right}$

e ✓

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

```
{ if (root == NULL) return NULL;
```

```
if (root->val == key) return helper(root);
```

```
TreeNode* dummy = root;
```

```
while (f. while (root != NULL)
```

```
if (root->val > key)
```

```
{ if (root->left != NULL && root->left->val == key)
```

```
root->left = helper(root->left);
```

```
break;
```

```
else { root = root->left; }
```

```
else
```

```
{ if (root->right != NULL && root->right->val == key)
```

```
root->right = helper(root->right);
```

```
break;
```

```
else
```

```
{ root = root->right; }
```

```
else
```

```
return dummy;
```

T.S

$O(\text{Height of tree})$

S.C

$O(1)$

```
TreeNode* helper(TreeNode* root)
```

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

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

```
TreeNode* rightchild = root->right;
```

```
TreeNode* lastRight = findLastRight(root->left);
```

```
LastRight->right = rightchild;
```

```
return root->left;
```

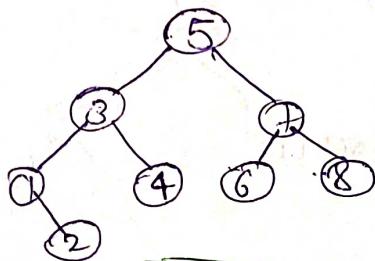
```
TreeNode* findLastRight(TreeNode* root)
```

```
{ if (root->right == NULL)
```

```
return root;
```

```
return findLastRight(root->right);
```

Kth smallest element in BST ?



$K = 3$

1 2 3 4 5 6 7 8

you need to return

① Do any DFS traversal

② Store all the node value in vector
→ Sort the vector
→ then return (K^{th}) index value.

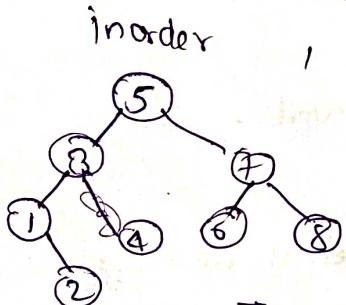
T.C. - $O(N)$ + $O(N \log N)$
S.C. - $O(N)$
vector

Efficient approach

* * * Property of BST

Inorder of Binary

Search Tree is always in sorted order



T.C. - $O(N)$

S.C. - $O(N)$ → vector

To avoid sc. $O(N)$ of vector.

Count = 0

left, Node, Right

Count++

if count == key
ans = node

Recursive { T.C. - $O(N)$

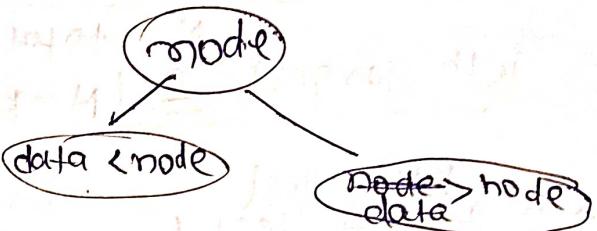
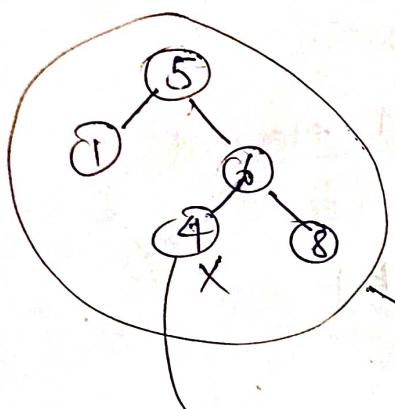
But if we do Morris traversal

T.C. - $O(N)$

S.C. - $O(1)$

→ attached this piece of code.

Check if a tree is BST or BT

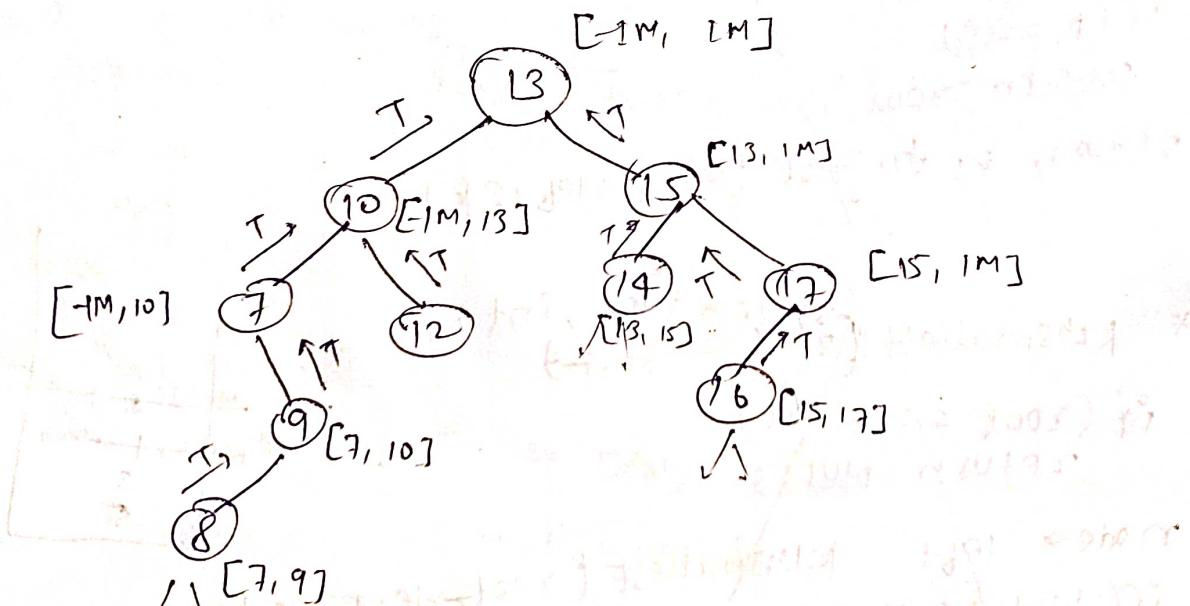


This is not BST because 4 is less than 5 in right of 5.

This 4 should be lesser than 6 and greater than 1.

Always maintain a range in this problem

$[l_m, r_m]$
int min max



Code:

```
boolean isValidBST (TreeNode* root)
```

```
{ return isValidBST (root, INT-MIN, INT-MAX); }
```

```
boolean isValidBST (TreeNode* root, long minval, long maxval)
```

```
{ if (root == NULL) return true;
```

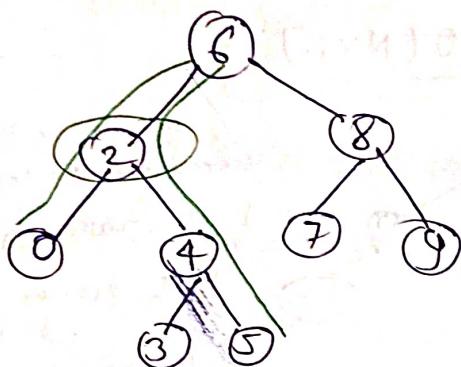
```
if (root->val >= maxval || root->val <= minval) return false;
```

```
return isValidBST (root->left, minval, root->right) &&
```

```
isValidBST (root->right, root->val, maxval); }
```

LCA in a BST

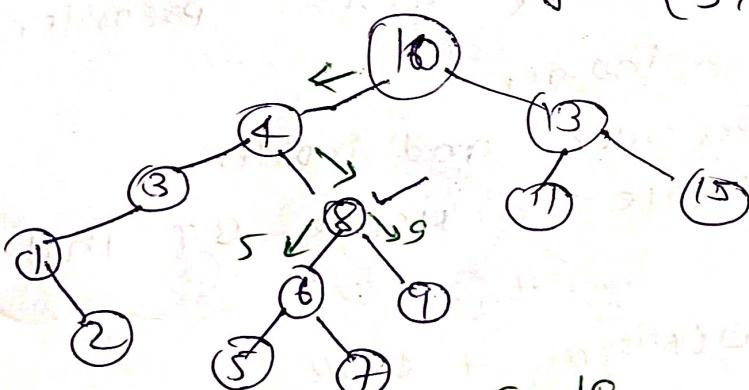
Lowest Common Ancestor



first intersection point from the bottom in the path.

5.02

We will traverse in BST and at the moment where left or right both the elements are not lies in let say (5, 8) we will return that node.



T.C. $O(H)$
S.C. $\underline{O(1)}$

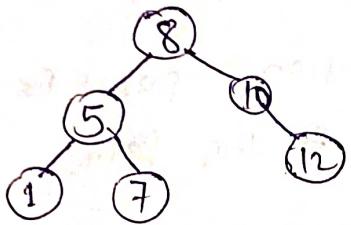
Code

```

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == NULL)
        return NULL;
    int curr = root->val;
    if (curr < p->val && curr < q->val)
        return lowestCommonAncestor(root->right, p, q);
    else if (curr > p->val && curr > q->val)
        return lowestCommonAncestor(root->left, p, q);
    else
        return root;
}
  
```

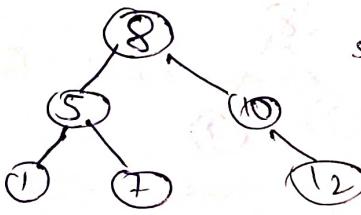
Construct a BST from Preorder traversal

Preorder $\rightarrow \{8, 5, 1, 7, 10, 12\}$ 3 methods



1st method

T.C. $O(N \times N)$



5 is smaller so left
1 is smaller so left.
7 is greater than 5 so right.
10 is greater than 5 so right.
12 is greater than 10 so right.

2nd Method

property

BST \rightarrow Inorder \rightarrow sorted

so first we will sort the given Preorder
it will become Inorder
Now we have Preorder and Inorder
so we can create a unique BT that will be BST.

$$\begin{aligned} T.C. &= O(N \log N) \\ S.C. &= O(N) \end{aligned}$$

post
 $O(N \log N) + O(N)$

$O(N)$

vector Inorder

3rd Method

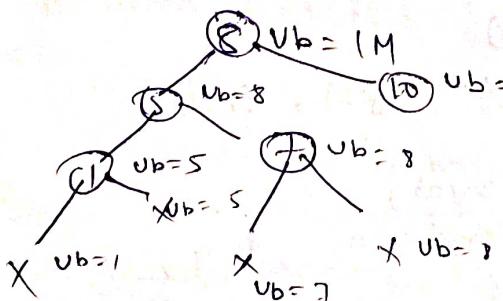
$[-\infty, \infty]$

node

$O(N \log N) + O(N)$

$O(N)$

vector Inorder



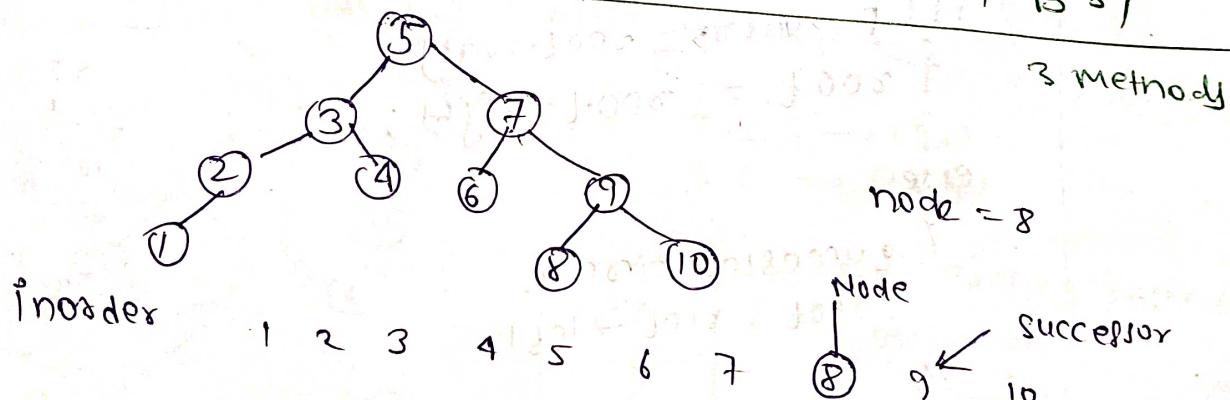
T.C. $\rightarrow O(3N) \approx O(N)$
S.C. $\rightarrow O(1)$

Code

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

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

Inorder & Successor/predecessor in BST



Method 1 :

- store the inorder (inorder of BST is sorted)
- find the value greater than node = 8

$$T.C = O(N) + O(N) S.C = O(N)$$

store find greater value.

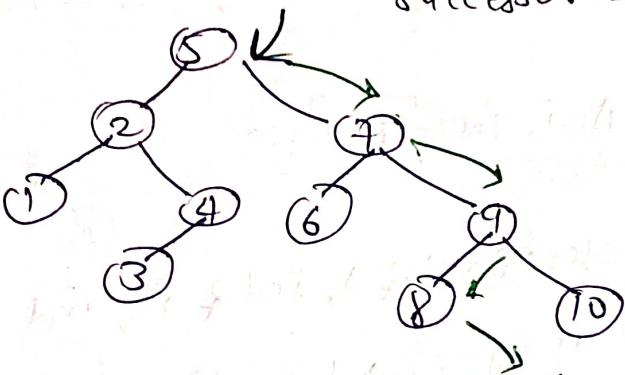
Method 2 :

- use Morris Traversal
- where we find value greater than node = 8 we will return

$$T.C = O(N) \\ S.C = O(1)$$

Method 3:

Let use a variable and initialize it as null.
 $\text{successor} = \text{null}$.



val = 8

successor

= 5 & 9

traverse till we get null

then return successor.

Code

$$\begin{aligned} T.C. &= O(H) = O(\log n) \\ S.C. &= O(1) \end{aligned}$$

Code:

`Treenode* findInorderSuccessor(Treenode* root, Treenode* p)`

`Treenode* successor = NULL;`

`while root != NULL`

```

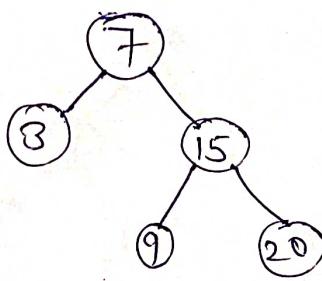
    if (P->val >= root->val)
        root = root->right;
    else
    
```

`successor = root;`

`root = root->left;`

`return successor;`

BST Iterator

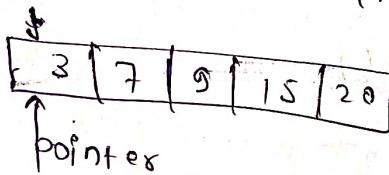


monday

3 7 9 15 20

Method 1 :-

store in order



Best Iterators (?)

next → ?

next →

hasnext → tail

next →

hashnext → true
new

has been \rightarrow 1

`isInNext` → True
`next` → -

hashmap

text \rightarrow false

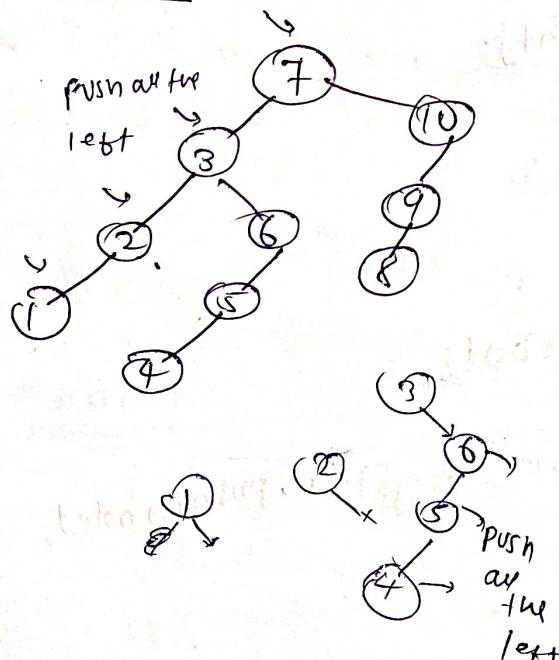
100 *100* *100* *100* *100* *100* *100* *100*

.c. - 011 } for next

But if you are not allowed to store disorder.

Method 2:

We will use stack



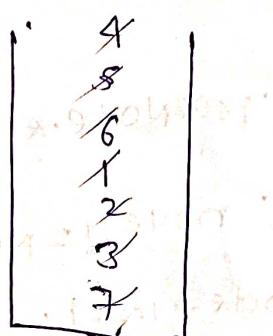
~~T.C.~~ → αγ

$$S \cdot C \rightarrow O(CH)$$

$$(N/n) \approx 0.1$$

for cheaping hardware

~~use~~ ~~stack~~ we check stack
whether it is empty
or not.



Sjaer 115

Code

Start Class BST iterator

Private :

Stack<TreeNode*> ~~root~~ myStack;

Public :

BST iterator(TreeNode* root)

{
 pushAll(root);
}

bool hasNext()

{
 return !myStack.empty();
}

int next()

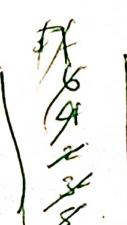
{
 TreeNode* tmpNode = myStack.top();
 myStack.pop();
 pushAll(tmpNode->right);
 return tmpNode->val;
}

Private :

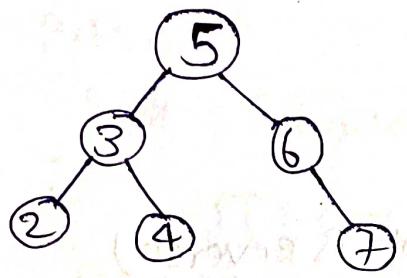
void pushAll(TreeNode* root)

{
 for(; node != NULL; myStack.push(node),
 node = node->left);
}

};



TWO SUM IN BST | check if there exists a pair with sum K



$$K = 9$$

$$3 + 6 = 9 \quad \text{True}$$

$$5 + 4 = 9 \quad \text{False}$$

$$K = 4$$

should be in pair.

inorder now it becomes two sum problem.

2 3 4 5 6 7

4 X

false

Two pointer approach

BS to array
vector store

Two pointer

$$T.C. = O(N) + O(N)$$

$$S.C. = O(N) \rightarrow \text{vector}$$

Just like previous problem

push all right node

next()
 $i = 1^{\text{st}}$

before()
 $j = 1^{\text{st}}$

instead of keeping in stack
it is same like two pointer approach.

X 2 3 4 5 11

Answer is

Code :-



$T.C. \rightarrow O(N)$
 $S.C. \rightarrow O(N) \times 2 \approx O(4N)$
 $= O(\log n)$

Class BSTIterator:

{

 stack<TreeNode*> mystack;

 bool reverse = true;

Public:

 BSTIterator(TreeNode* root, bool isReverse)

 {

 reverse = isReverse;

 pushAll(root);

 bool hasNext():

 {

 return !mystack.empty();

 int next():

 {

 TreeNode* tmpNode = mystack.top();

 mystack.pop();

 if(!reverse)

 pushAll(~~tmpNode -> right~~);

 else

 pushAll(~~tmpNode -> left~~);

 return tmpNode->val;

Private:

 void pushAll(TreeNode* node)

 {

 for(; node != NULL;)

 mystack.push(node);

 if(reverse == true)

 node = node->right;

 else

 node = node->left;

}

class solution :

{

public :

 bool findTarget (TreeNode* root, int k)

 { if (!root) return false;

 BSTIterator l (root, false);

 BSTIterator 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();

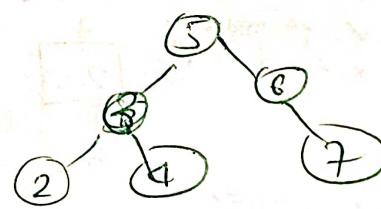
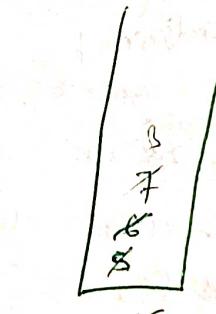
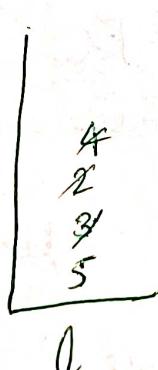
 else

 j = r.next();

 }

}

}



$$2 + 7 = 9$$

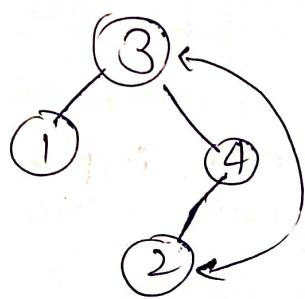
$$3 + 7 = 10$$

$$3 + 6 = 9$$

$$4 + 6 = 10$$

$$5 + 5 = 10$$

Recover BST



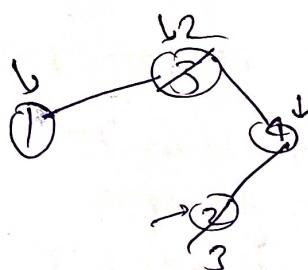
Two nodes swapped

We have to correct the BST.

→ Do inorder traversal

→ Sort the traversal

1 2 3 4



Simultaneously match and if it is not matching just correct the traversal

T.C. = $O(2N) + N \log(N)$

S.C. = $O(N)$

vector to store the inorder.

nodes
5 & 25

Swap can have two cases :-

1. Swapped nodes are not adjacent

25 > 3 ✓ 1st mismatch
7725 X 25 2
8 > 7 ✓
10 > 8 ✓
15 > 10 ✓
20 > 15 ✓
25 > 20 ✗ last violation.
2. Swapped nodes are adjacent

25
↑
first
7 8
↑ middle
10 15
↑
20
↑
last

25 ✗
7 8 X 1st violation
10 15 ✗ middle violation
20 ✗ last violation

nodes
7 & 8

3 5 8 7 10 15 20 25
first violation
middle violation
last violation

T.C. = $O(N)$

S.C. = $O(1)$ - memory + traversal

5 > 3 ✓
8 > 5 ✓ middle first
7 > 8 ✗ 7 8
10 > 7 ✓
15 > 10 ✓
20 > 15 ✓
25 > 20 ✓

If we don't get last violation then swap first & middle node.

else swap first and last node.

class solution :-

private :

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

private :

```
void inorder(TreeNode* root)
{ if (root == NULL)
    return;
```

```
    inorder (root->left);
    if (prev == NULL && root->val < prev->val)
```

```
{ if (first == NULL)
    { first = prev;
    middle = root;
    }
}
```

else

```
    last = root;
```

}

```
    prev = root;
    inorder (root->right);
```

3

public :

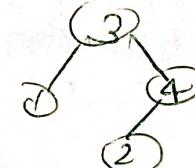
```
void recovertree(TreeNode* root)
{ first = middle = last = NULL;
    prev = new TreeNode(INT_MIN);
```

```
    inorder (root);
    if (first && last) &
```

```
        Swap (first->val, last->val);
    else if (first && middle)
        Swap (first->val, middle->val);
```

3

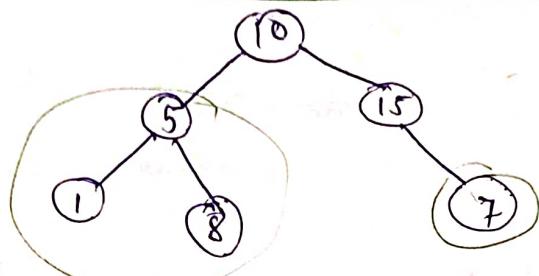
```
    }
```



1 2 3 4

1 3 2 4

Largest BST in BT

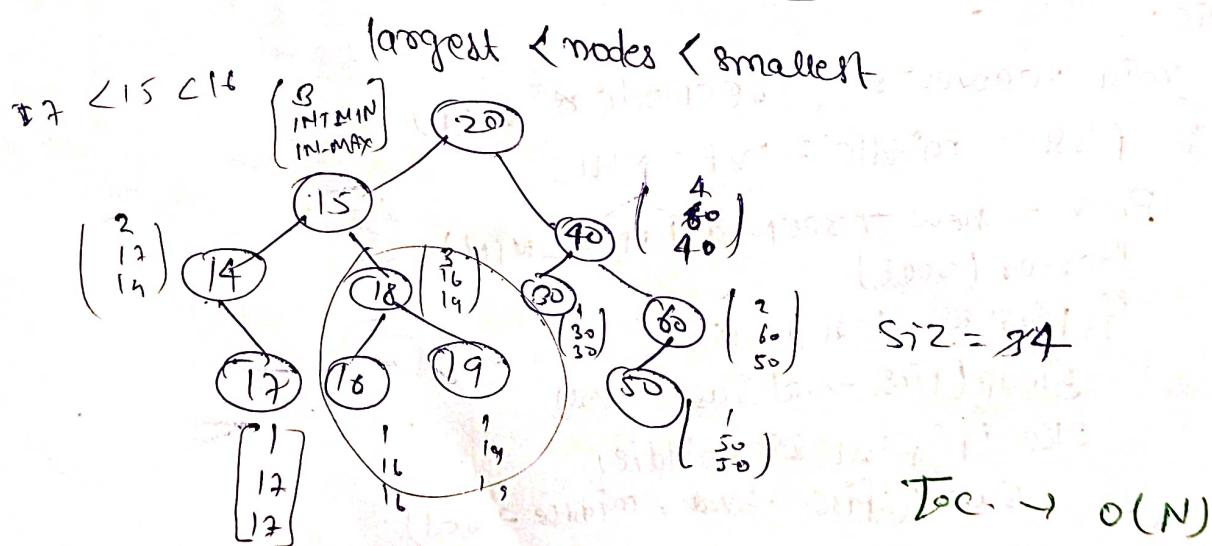
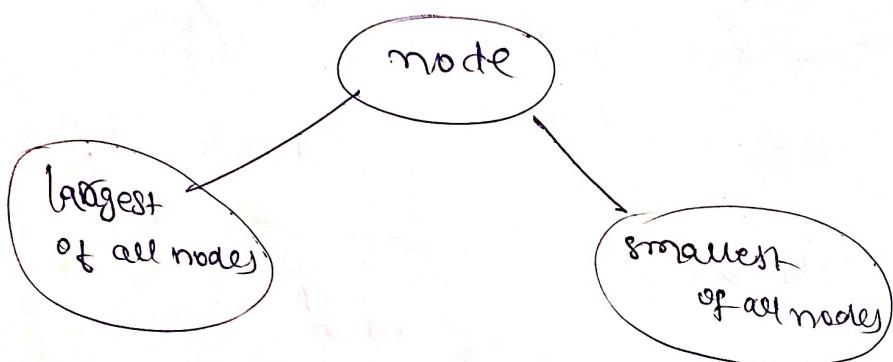


BST of size 3

BST of size 1

Brute force

Validate a BST → Validate every node for valid BST or not
check every node for valid BST or not → Traverse every node.
 then → return the root to the function which
 @ total. no. of nodes.



T.C. $\rightarrow O(N)$

S.C. $\rightarrow O(1)$

(0, INT_MIN, INT_MAX)

Avoid recursion

DO Morris traversal

@Aashish Kumar Nayak

Code :-

```
class NodeValue
{
public:
    int maxNode, minNode, maxSize;
    NodeValue(int minNode, int maxNode, int maxSize)
    {
        this->maxNode = maxNode;
        this->minNode = minNode;
        this->maxSize = maxSize;
    }
};

class Solution
{
private:
    NodeValue largestBSTSubtreeHelper(TreeNode* root)
    {
        if(!root)
            return NodeValue(INT_MAX, INT_MIN, 0);
        auto left = largestBSTSubtreeHelper(root->left);
        auto right = largestBSTSubtreeHelper(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 NodeValue(INT_MIN, INT_MAX, max(left.maxSize, right.maxSize));
    }
public:
    int largestBSTSubtree(TreeNode* root)
    {
        return largestBSTSubtreeHelper(root).maxSize;
    }
};
```