BINARY TREE

AND

BINARY SEARCH TREE

Handwritten Notes of Striver(TUF) Playlist

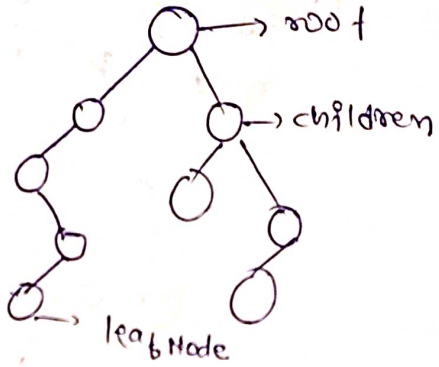
by: Aashish Kumar Nayak

NIT Srinagar

Linkedin Instagram

Introduction to Binary Trees

Full BT → either 0 or 2 children.

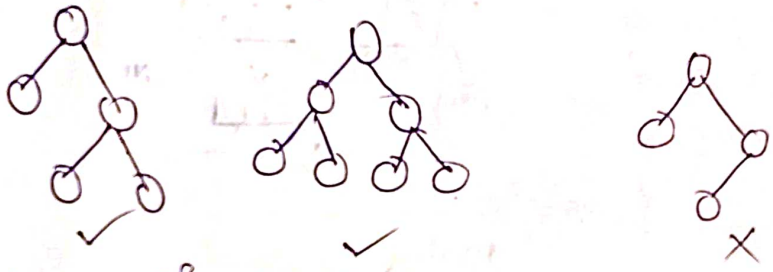


Five types of Binary Tree

- Full BT
- complete BT
- perfect BT
- Balanced BT
- Degenerate tree

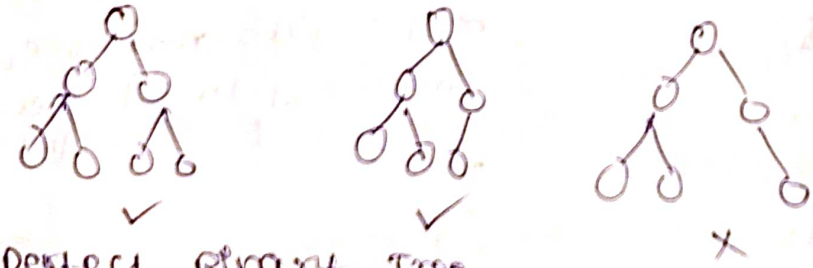
(•) Full Binary Tree

Either 0 or 2 children.



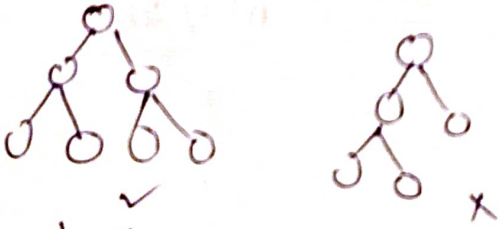
(•) Complete Binary Tree

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



(•) perfect Binary Tree

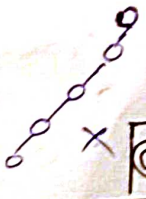
All leaf nodes are at same level.



(•) Balanced Binary Tree

Hight of tree at max $\log(N)$ (N: Nodes)

Ex: $n = 8$ $\log_2 8 = 3$



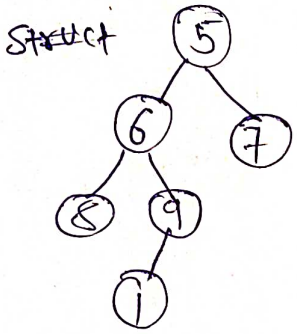
@ Aashish Kumar Nayak

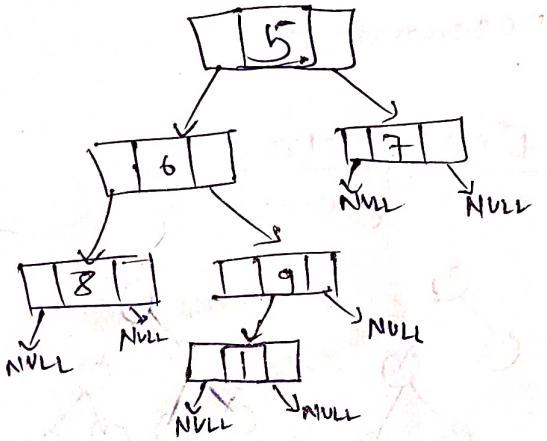
① Degenerate Tree

Every node have single children.

$n = 4$.

Binary Tree representation in C++



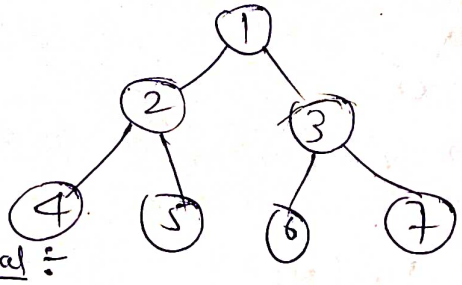



```
Struct Node {
    int data;
    Struct Node *left;
    Struct Node *right;

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);
}
```

Traversal Technique

BFS / DFS



DFS Traversal :-

① Inorder Traversal (~~left~~ LNR) 4 2 5 1 6 3 7

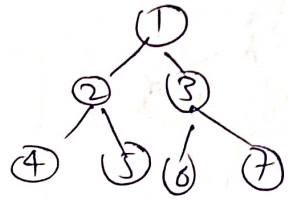
② pre order Traversal (NLR) 1 2 4 5 3 6 7

③ postorder Traversal (LRN) 4 5 2 6 7 3 1

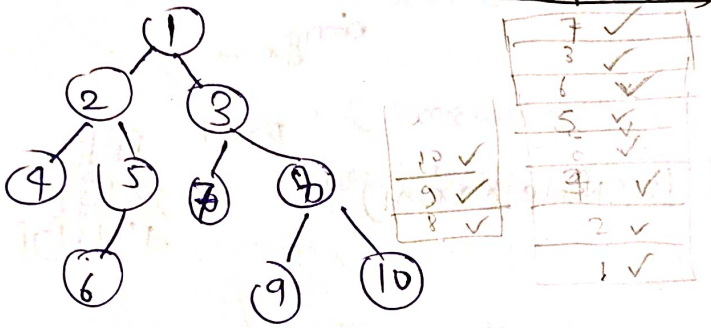
② BFS Traversal

① For lever wise order / level order Traversal.

1 2 3 4 5 6 7



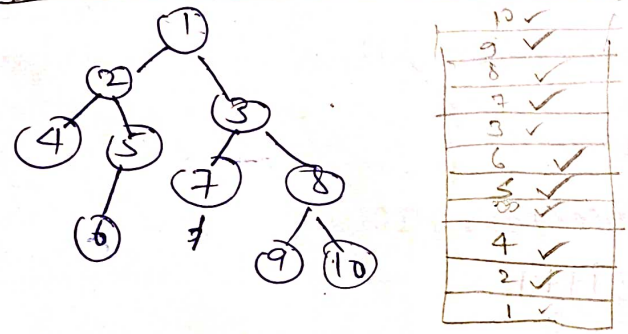
Pre-order Traversal : (NLR)



1 2 4 5 6 3 7 8 9 10

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

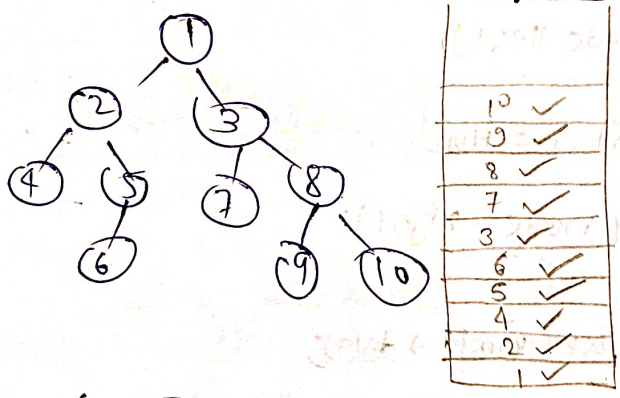
In-order Traversal (LNR) :



4 2 6 5 1 7 3 9 8 10

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

Post-order Traversal (LRN) :

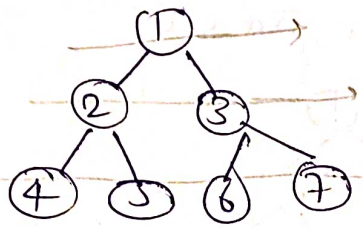


4 6 5 2 7 9 10 8 3 1

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

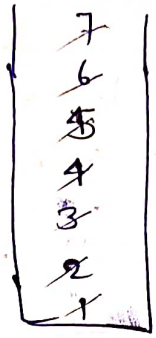
@Aashish Kumar Nayak

level order Traversal :-



Binary Tree

1
2 3
4 5 6 7

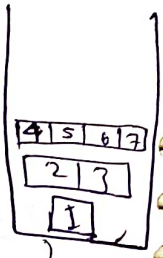


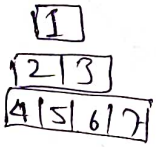
Queue

empty now

print

DS





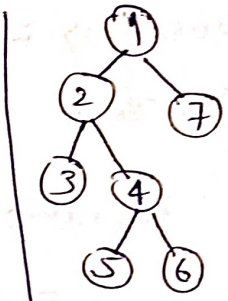
Code :-

1 2 3 4 5 6 7

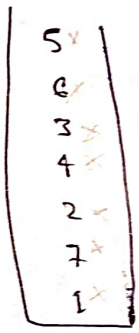
```
vector<vector<int>> levelorder(TreeNode *root)
{
    vector<vector<int>> ans;
    if(root == NULL)
        return ans;
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty)
    {
        int size = q.size();
        vector<int> level;
        for(int i=0; i<size; i++)
        {
            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)

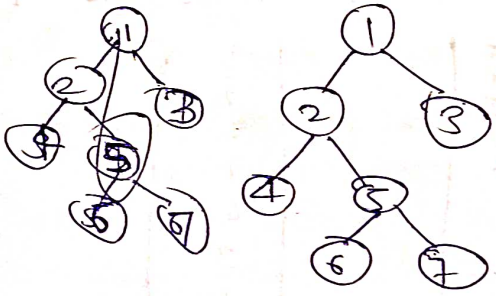
```
vector<int> preorder(TreeNode* root)
{
    vector<int> ans;
    if(root == NULL)
    { return ans;
    }
    stack<TreeNode*> st;
    st.push(root);
    while(!stack)
    {
        TreeNode*
            root = st.top();
            st.pop();
        ans.push_back(root->data);
        if(root->right != 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)

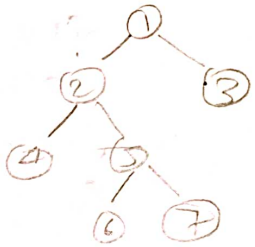


4 2 6 5 7 1 3

```
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;
            }
            stopop();
            node = st.top();
            st.pop();
            inorder.push_back(node->data);
            node = node->right;
        }
    }
    return inorder;
}
```

O(N)
O(N) – stack

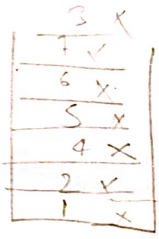
L N R



Step 1 :- left में चलते जाओ और stack में push भी करते जाओ।

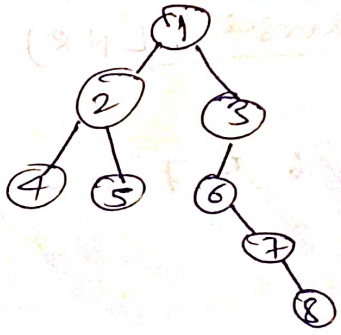
Step 2 :- जैसे ही NULL आ जाएगा top element pick करो ans array में insert करो और right में जाओ।

Step 3 :- stack.empty = True हुआ तो break;

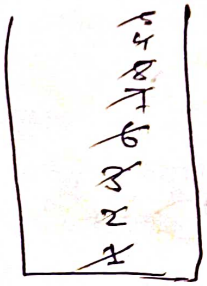


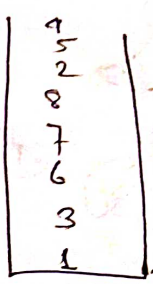
4	2	6	5	7	1	3

Iterative methode of post order (LRN) :-



→ 4 5 2 8 7 6 3 1





@Aashish Kumar Nayak

```
vector<int> postorder(TreeNode * root)
{
    vector<int> postorder;
    if(root == NULL)
        return postorder;
    Stack<int> st1, st2;
    Stack<TreeNode*> st1, st2;
    st1.push_back
        st1.push(root);
    while(!st1.empty())
    {
        root = st1.top();
        st2.push(root);
        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;
}
```

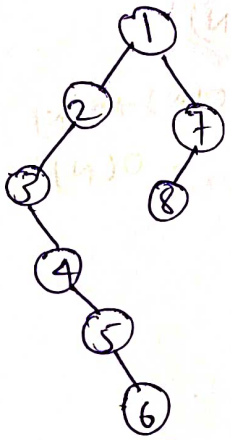
T.C. - O(N)

S.C. - O(N) + O(N) = O(N)

* It is same concept like Level order but here we use stack instead of queue.

* Stack(S_2) की जगह हम सीधे direct array में push कर देंगे और last में Reverse करके return कर देंगे।

Iterative postorder Traversal using 1 Stacks

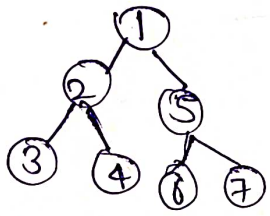


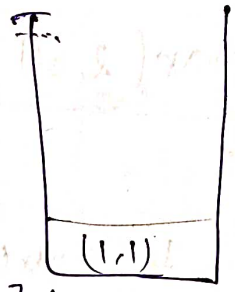
6 5 4 3 2 8 7 1

```
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();
post.add(temp->val);
else
    curr = temp;
```

@Aashish Kumar Nayak

Preorder, Inorder, Postorder Traversal in one Traversal





preorder: 1 2 3 4 5 6 7

inorder: 3 2 4 1 6 5 7

postorder: 3 4 2 6 7 5 1

if num == 1
preorder
++
left

if num == 2
inorder
++
right

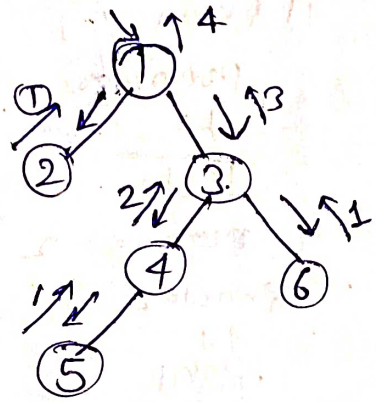
if num == 3
postorder

```
vector<int> preInPostOrderTraversal(TreeNode* root)
{
    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->data);
            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->data);
            it.second++;
            st.push(it);
            if(it.first->right != NULL) st.push({it.first->right, 1});
        }
        else { post.push_back(it.first->data);
        }
    }
    return pre/post/in;
```

@Aashish Kumar Nayak

Maximum depth of Binary Tree

$1 + \max(l, r)$



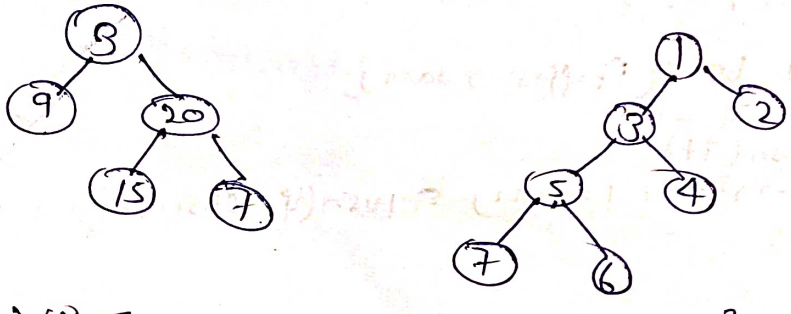
1 + max(1,0)
1 + max(0,0)
1 + max(2,1)

1 + max(1,3)
= 4 Ans

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

Cheaked for Balanced Binary Tree

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



3-1 = 2

Pseudo code

```
Bool cheak(Node)
{ if(Node == NULL) return true;
int Lh = findHLeft(Node->left);
int Rh = findHright(node->right);
   if( abs(lh-Rh) > 1 ) return false;
Bool left = cheak(Node->left);
Bool right= cheak(Node->right);
if( !left || !right) return false;
return true;
```

$O(N^2)$

Best sol :-

```
Bool isBalanced(TreeNode* root)
{ return dfsheight(root) != -1;
}
int dfsheight(TreeNode* root)
{ if(root == NULL)
  { return 0;
  }
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;
}
```

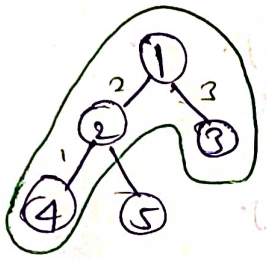
TC - O(N)

SC - O(N)

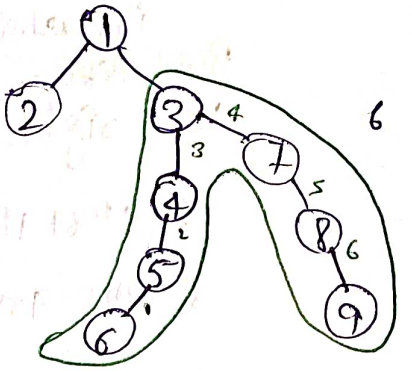
@Aashish Kumar Nayak

Diameter of a Binary Tree :-

Diameter :- The longest path between two nodes,
→ path does not need to pass via root.



3



6

$O(N^2)$

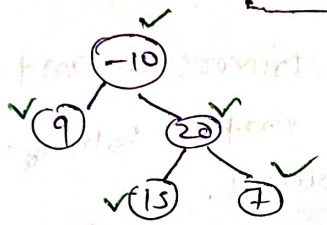
```
find max(node)
{ if(node == NULL)
    &return 0;
  }
lh = findleft(node->left);
rh = findright(node->right);
maxi = max(maxi, rh+lh);
findmax(node->left);
findmax(node->right);
}
```

Better Soln :-

```
int diameterOfBinaryTree(TreeNode* root)
{ int diameter = 0;
  height(root, diameter);
  return diameter;
}
int height(TreeNode * root, int & diameter)
{ if(root == NULL) return 0;
  int lh = height(root->left, diameter);
  int rh = height(root->right, diameter);
  diameter = max(diameter, (lh+rh));
  return 1+max(lh, rh);
}
```

Maximum path sum

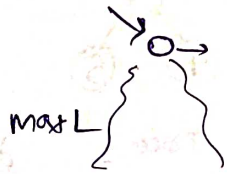
↳ node A → node B



15 + 20 + 7 = 42 ✓

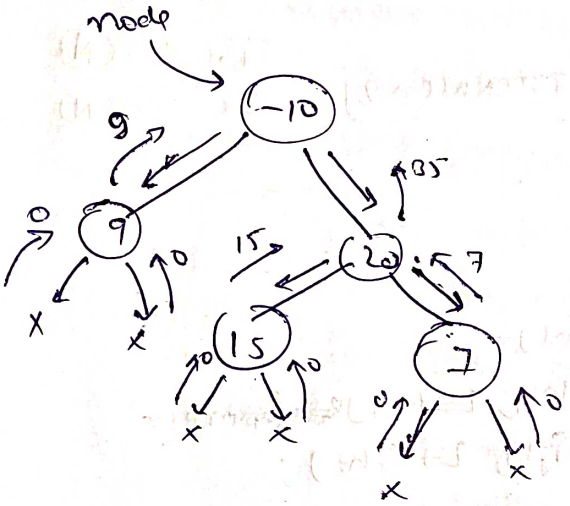
9 - 10 + 20 + 15 = 34

9 - 10 + 20 + 7 = 26



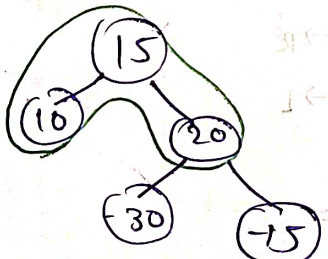
⇒ Val + (maxL + maxR)

maxi = 0 9 15 42




```
int pathmaxpath(node, maxi)
{
  if(node == NULL)
    return 0;
  leftsum = maxpath(node->left, maxi);
  rightsum = maxpath(node->right, maxi);
  Maxi = Max(maxi, leftsum + rightsum + node->val);
  return (node->val) + max(leftsum, rightsum);
}
```

1 taste case -ve



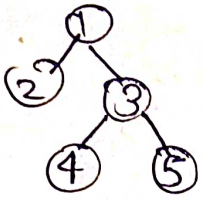
if some path returning -ve value than make it zero (ignore them)

Sol

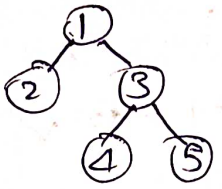
```
int maxpathsum(TreeNode *root)
{
  int maxi = INT_MIN;
  maxpathDown(root, maxi);
  return maxi;
}
```

```
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



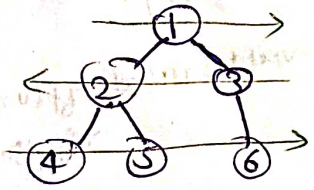
Tree 2

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

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

```
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);
}
```

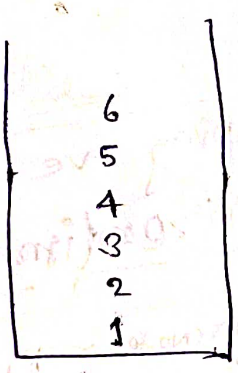
Zig-Zag or Spiral Traversal :



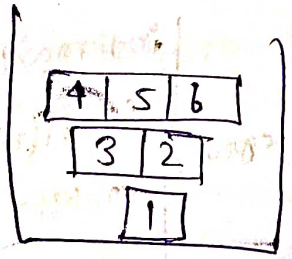
1
3 2
4 5 6

All same as level order traversal only one difference. Here we use flag variable just for printing output in alternate direction.

flag = 0 L → R
flag = 1 R → L



Queue



dll

```
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 = nodequeue.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;
}
```

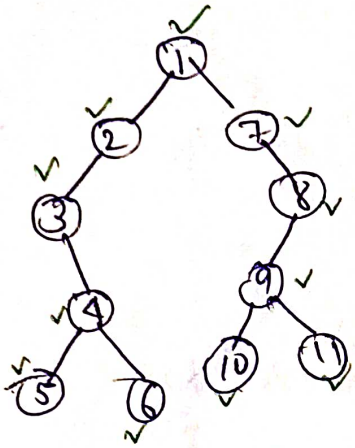
Boundary Traversal

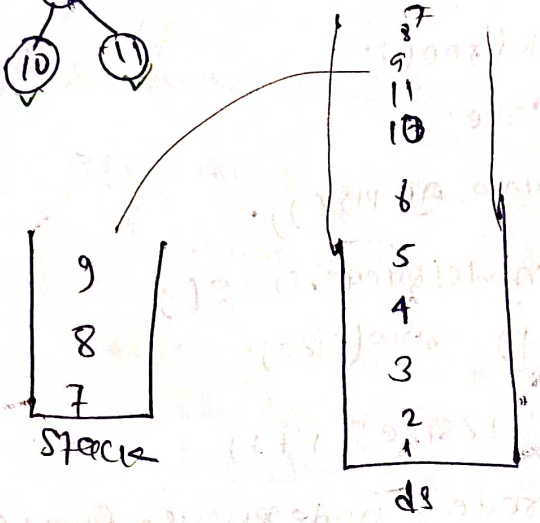
First point.

Left boundary excluding leaf

Leaf nodes

Right boundary on the reverse excluding leaf





use stack for reverse
or
vector

```
vector<int> printBoundary(Node *root)
{
    vector<int> res;
    if(root if(!root) ret
        if(root == NULL)
            return res;
    if(!isLeaf(root))
    {
        res.push_back(root->data);
    addBoundaryLeft(root, res);
    addLeaves(root, res);
    addRightBou
    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
    }
}
```

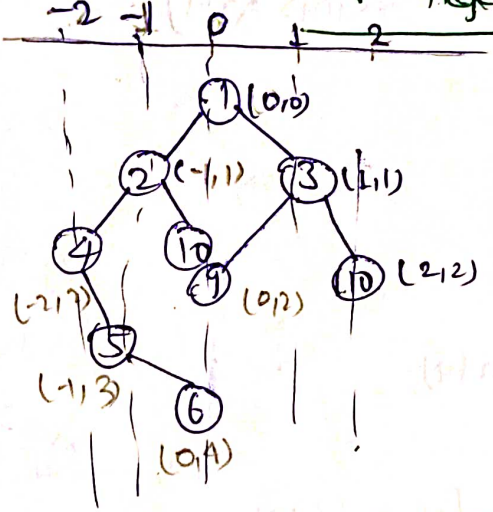
T.C. - O(H) + O(H) + O(N) = O(N)

S.C. - O(N)

```
void addRightBoundary(Node* root, vector<int> &res)
{
    Node* curr = curr->right
    ~~vector~~ stack<int> st;
    while(curr)
    {
        if(!isLeaf(curr))
            ~~res~~ 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);
}
```

@Aashish kumar Nayek

Vertical order traversal :-



4
2 5
1 9 10 6
3
10

9 9

set<int> s;

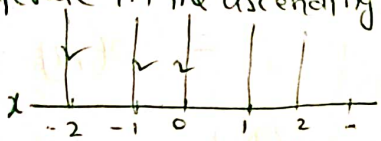
Sorted & unique elements

Multiset<int> s;

or priority queue

9 9 sorted only.

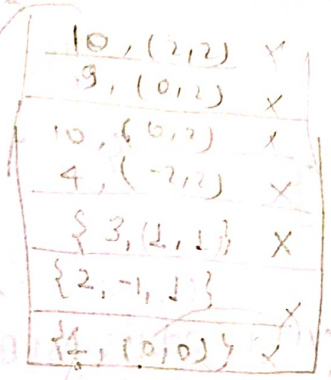
iterate in the ascending order of x.



-2 4
-1 2 5
0 1 9 10 6

Q(node, v, level)

6, (0,4)
5, (-1,3)



todo queue

node x vertical y level

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

vertical level multinodes

Treemap<int, treemap<int, po>>

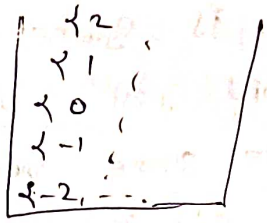
```
vector<vector<int>> verticleTraversal(TreeNode* root)
{   map<int, map<int, multiset<int>>> nodes;
    queue<pair<TreeNode*, pair<int, int>>> todo;
    todo.push({root, {0, 0}});
    while(!todo.empty()){
        auto p = todo.front();
        todo.pop();
        TreeNode* node = p.first;
        int x = p.second.first, y = p.second.second;
        nodes[x][y].insert(node->val);
        if(node->left) { todo.push({node->left, {x-1, y+1}}); }
        if(node->right)
        {
            todo.push({node->right, {x+1, y+1}});
        }
    }
```

nodes map multiset

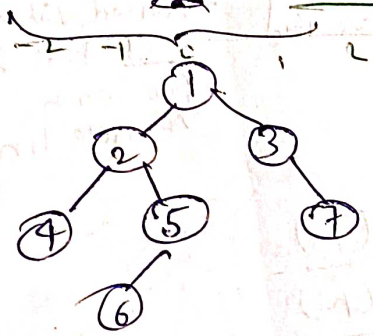
multiset में insert करवा दो

```
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;
```

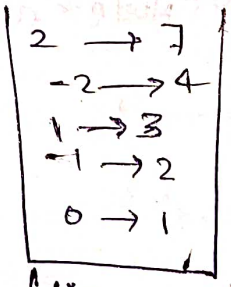
* We are using map so it will be in sorted order i.e.

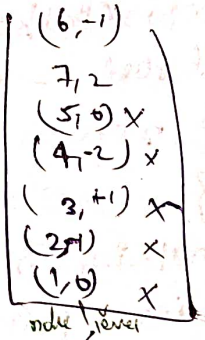


Top view of Binary Tree



4 2 1 3 7





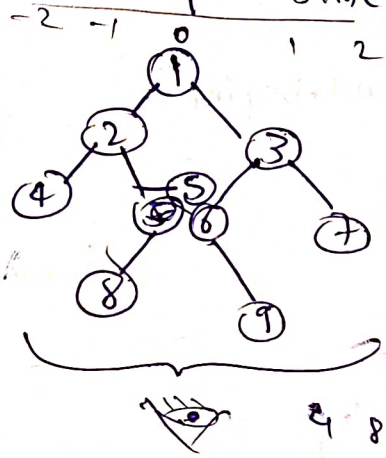
node = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ 6

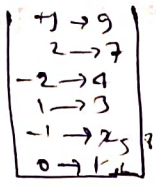
```
vector<int> topview(Node* root)
{
vector<int> ans;
if(root == NULL)
  { return ans;
  }
map<int, int> map;
queue<pair<Node*, int>> q;
q.push({root, 0});
 while(!q.empty)
   {
     auto it = q.front();
       q.pop();
 Node* node = it.first;
  int line = it.second;
 if(map.find(line) == map.end())
   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;
}
```

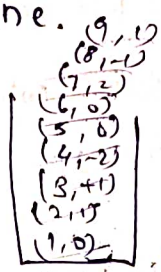
@Aashish Kumar Nayak

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 same vertical line.

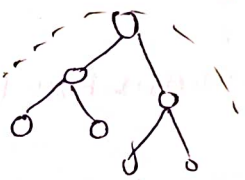






4 8 6 9 7

Top view



only first assigned value of map for each vertical line.

Bottom view

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

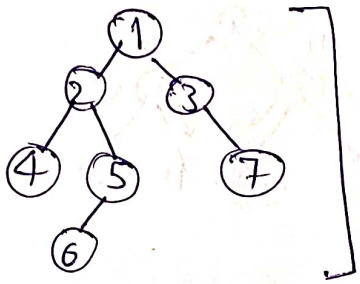
```
vector<int> bottomview(Node* root)
{  vector<int> ans;
   if(root == NULL)
    { return ans;
    }
   map<int,int> mpp;
   queue< pair<Node*,int>> q;
   q.push({root,0});
  {while(!q.empty()){
    auto it = q.front();
        q.pop();
    Node* node = it->first;
    int line = it.second;
    mpp[line]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

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

@Aashish Kumar Nayak

Right/left side view

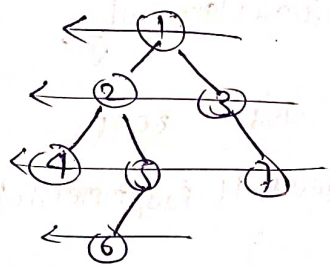
@Aashish Kumar Nayak



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 → O(N)

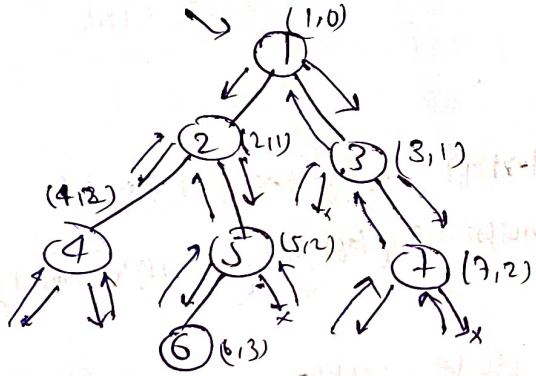
SC → O(N)

[Iterative]

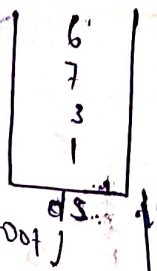
Level order

TC → O(N)

SC →




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



Right view

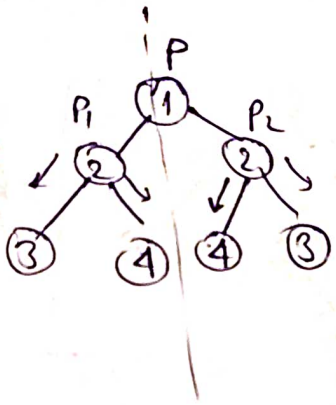
```
vector<int> rightview(node* root)
{ vector<int> res;
  recursion(root, 0, res);
  return res;
}
void recursion(node* root, int level, vector<int>& res)
{ if(root == NULL) return;
  if(res.size() == level)
    res.push_back(root->data);
  recursion(root->right, level+1);
  recursion(root->left, level+1);
}
```

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.

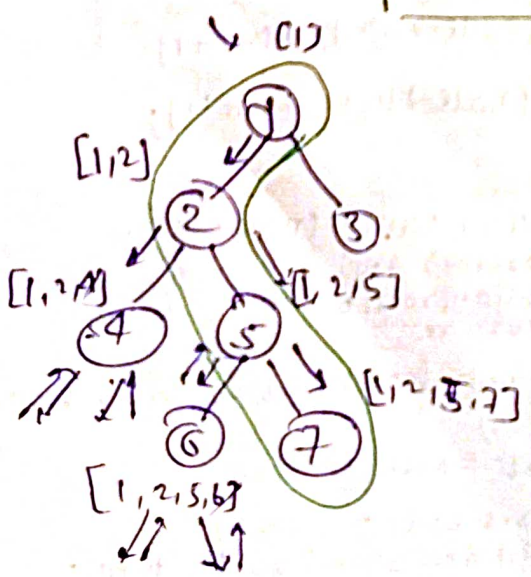


We will iterate P1 towards right while P2 towards left and then P1 towards left & P2 towards right simultaneously.

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

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

Print root to Node Path in BT

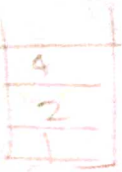


Node = 7

1 2 5 7

Inorder traverse
Recursive
stack space

T.C. - O(N)
S.C. — O(N)



@Adhish Kumar Nayak

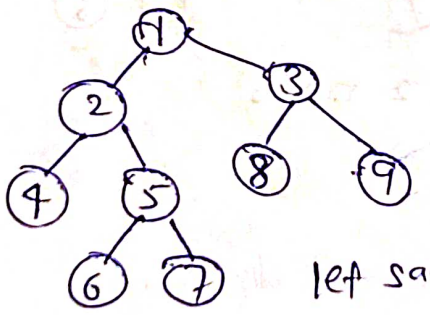
```
bool getpath (TreeNode* root, vector<int>&arr, int x)
{ if (root==NULL)
    return false;
  arr.push_back (root->val);
  if(root->val == x)
    return true;
  if (getpath(root->left, arr, x) ||
      getpath (root->right, arr, x))
    return true;
  arr.pop_back();
  return false;
}

vector<int> solve (TreeNode * A, int B)
{
  vector<int> arr;
  if(A == NULL)
    return arr;
  getpath (A, arr, B);
  return arr;
}
```

O(n)

O(n) + O(n) = O(n)

Lowest common Ancestor (Binary Tree)

The ancestor that exists at deepest level of



lca(4,7) = 2

lca(5,9) = 1

lca(2,6) = 2

let say lca(4,7)

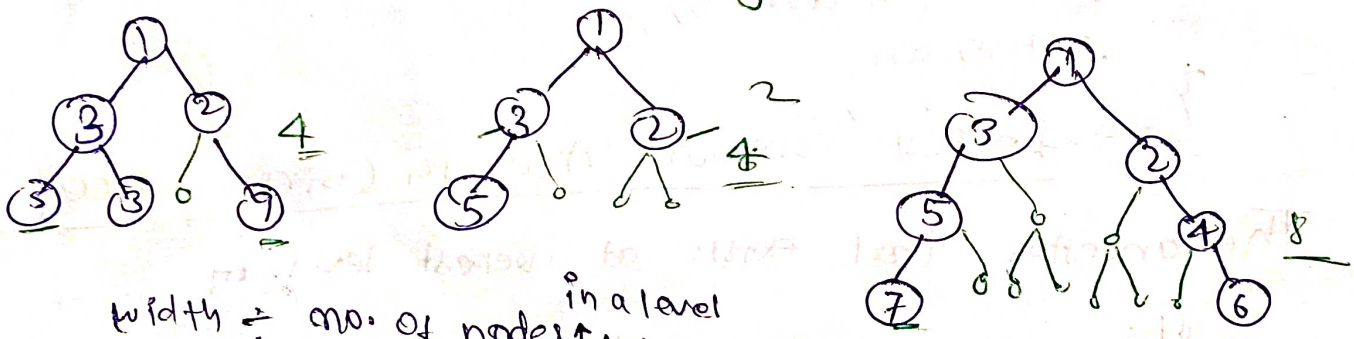
matching

path node = 4 (1 2 4)

path node = 7 (1 2 5 7)

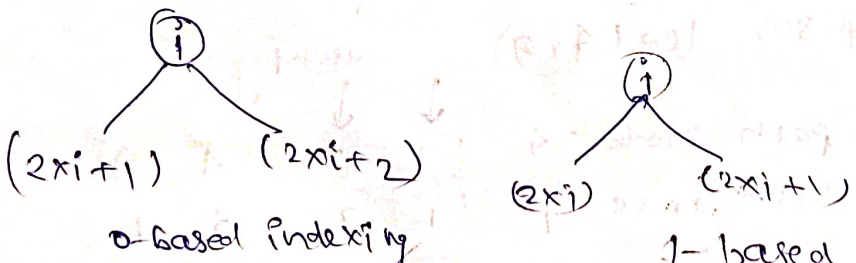
```
TreeNode * LowestCommonAncestor(TreeNode* root, TreeNode* p,
                                TreeNode* q)
{ if(root == NULL || root == p || root == q)
  { return root;
  }
  TreeNode* left = lowestCommonAncestor(root->left, p, q);
  TreeNode* right = LowestCommonAncestor(root->right, p, q);

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

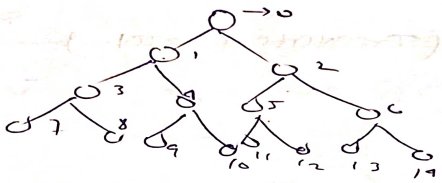
Maximum width of Binary Tree :



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

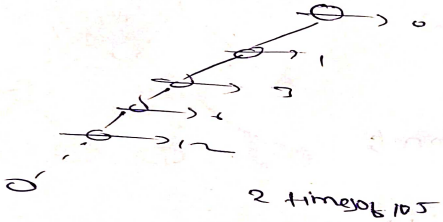
Think about : level order traversal





(14 - 7) + 1

Failed test case

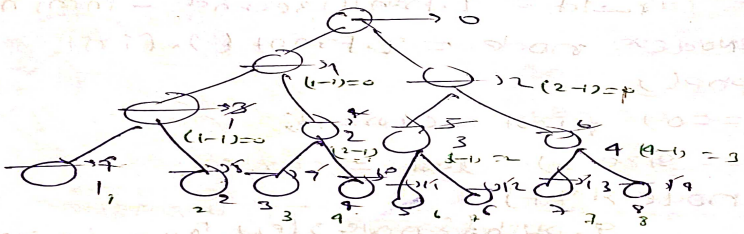


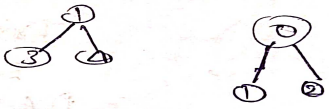
(2 x i) + 1

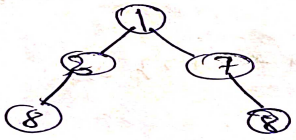
10 :-

2 times 10 will overflow.

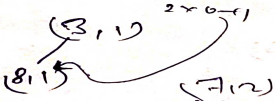
in order to reduced overflow

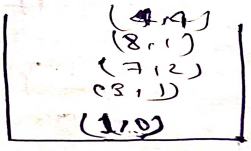




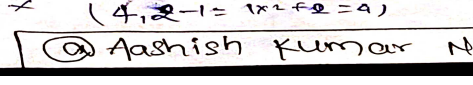


(left - right + 1)





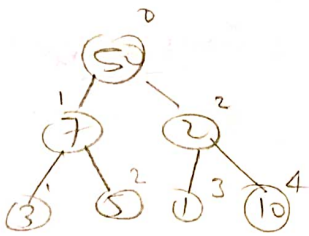
width = 4 x 4



(4, 2-1 = 1x2+2 = 4)

© Aashish Kumar Nayak

```
int WidthOfBinaryTree(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, (long long)cur_id*2+1});
      if(node->right)
        q.push({node->right, (long long)cur_id*2+2});
    }
    ans = max(ans, last-first+1);
  }
  return ans;
}
```



(100)
(1,3)
(5,2)
(3,1)
(2,2) ✓
(7,1) ✓
(50,0) ✓

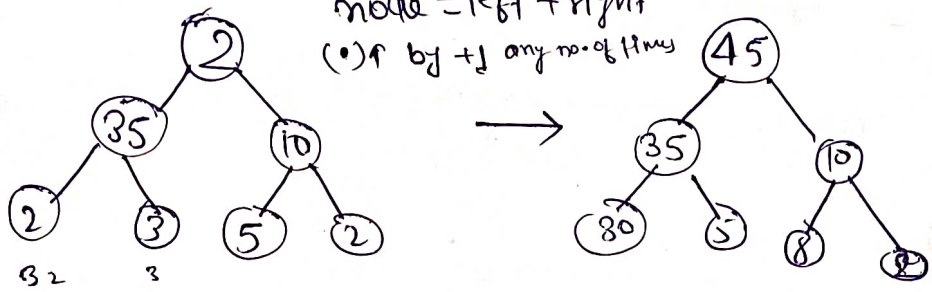
ans = 2
cur_id = 0
mmin = 1
④

Aashish Kumar Nayak

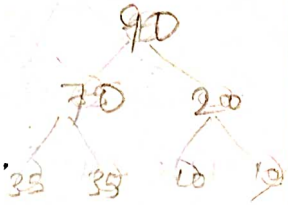
Children Sum property in Binary Tree

node = left + right

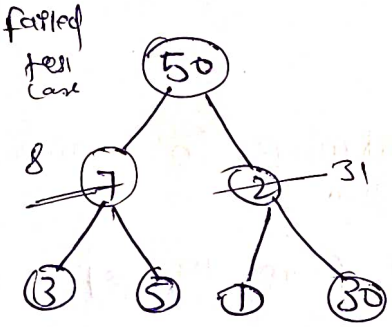
(+) or by (-) any no. of times



35 ⇒ 30 + 5



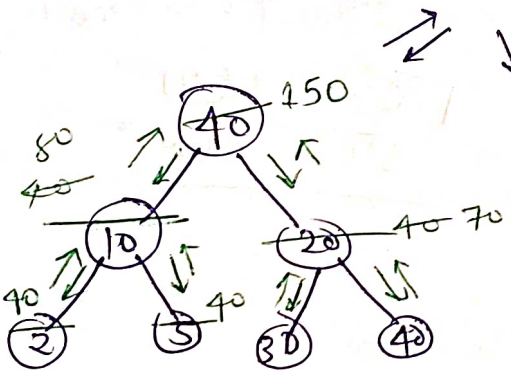
failed test case

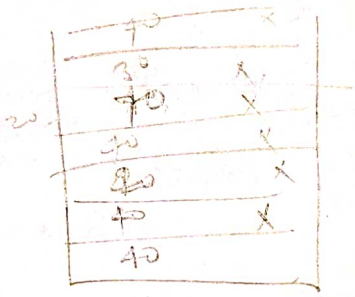


bruteforce and all solutions are $O(N^2)$.

we will solve in O(N).

solution





40

10 + 20 = 30 < 40

2 + 5 = 7 < 40

30 + 40 = 70 > 40

T.C. - O(N)

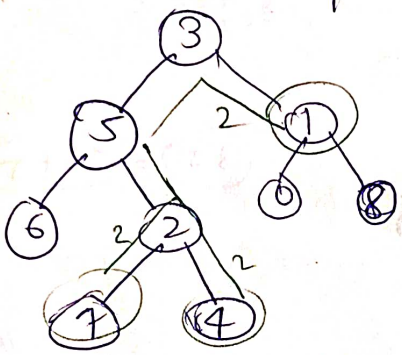
S.C

```
void reorder (BinaryTreeNode<int> *root)
{ if(root == NULL)
    return 0;
  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;
    if(root->right) root->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 or root->right)
      root->data = tot;
}
```

@Ashish Kumar Nayak

Print all nodes at a distance of K

K = 2, target = 5

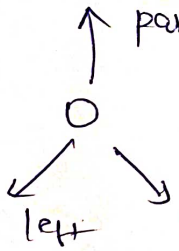


Output = 7, 4, 1

~~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 till K ~~dist~~ we reached ~~dist~~ upto distance K

T.C. = O(N)

S.C = O(N)

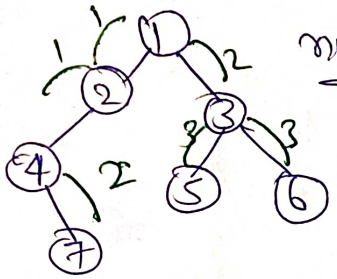
```
if(current->right && !visited[current->right])
    { queue.push(current->right);
      visited[current->right] = true;
    }
if( parent_track[current] && !visited[parent_track[current]])
    { queue.push(parent_track[current]);
      visited[parent_track[current]] = true;
    }
for loop }
while loop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

@Ashish Kumar Nayak

```
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;
        }
```

@Aashish Kumar Nayak

Minimum Time taken to Burn a Binary Tree From a node/Leaf node



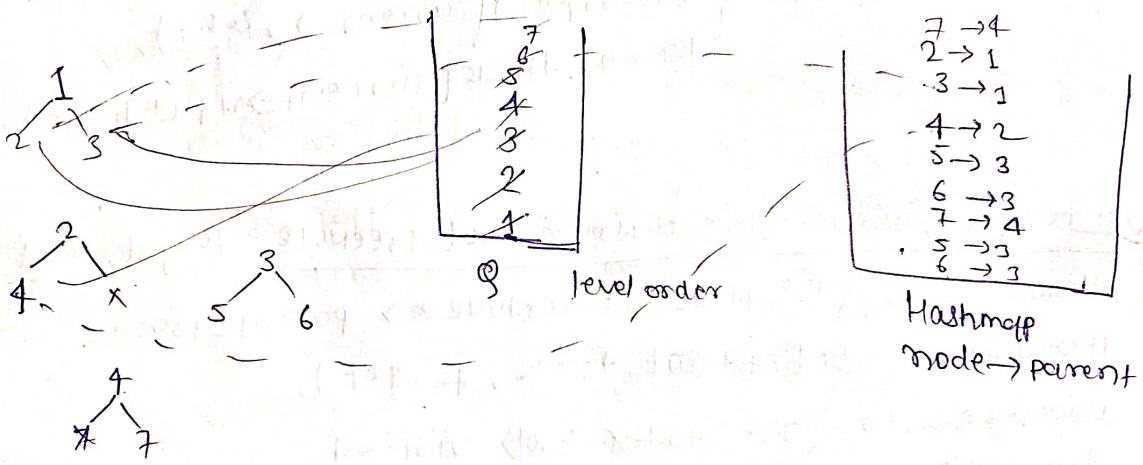
node = 2

@Aashish Kumar Nayak

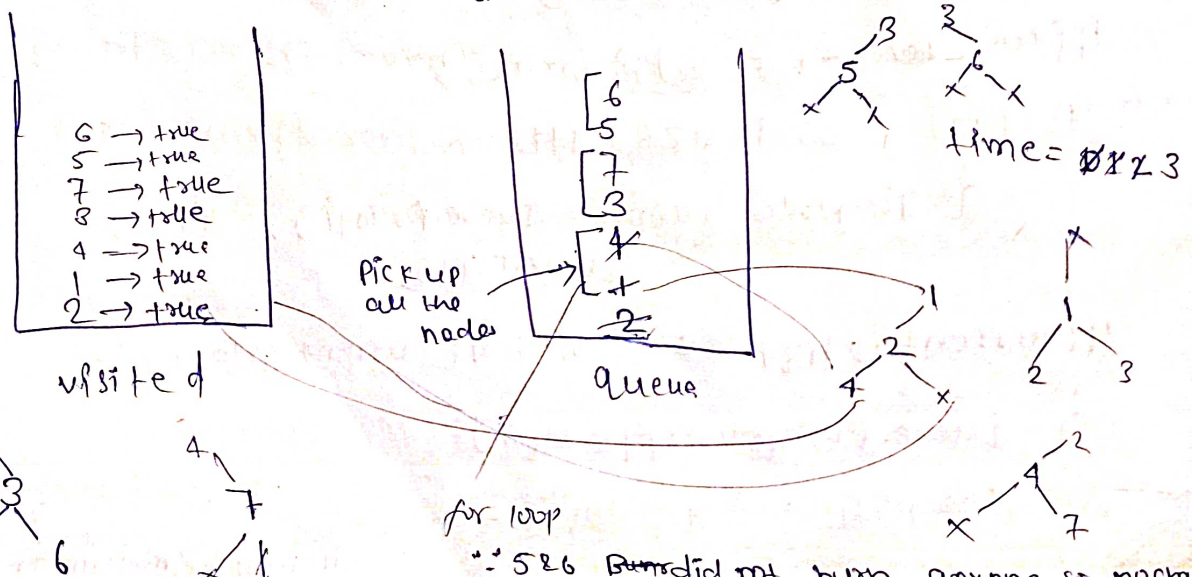
total 3 sec में पूरा Burn हो जायेगा।

As we will apply BFS just like previous question

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 visited hashmap.



or 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.

T.C. = O(N) + O(N)
≈ O(N)
S.C → O(N)

Code :-

```
int timeToBurnTree( ~~TreeNode*~~ BinaryTreeNode<int>* root, int start)
{ map<BinaryTreeNode<int>*, BinaryTreeNode<int>*> mapp;
  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>* ~~targ~~ 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;
}
```

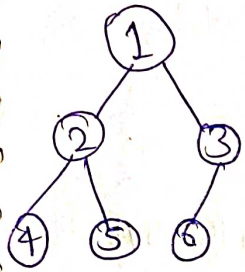
@Aashish Kumar Nayak

```
int findMaxDistance(map<BinaryTreeNode<int>*, BinaryTreeNode<int>*>
&mpp, BinaryTreeNode<int>* target)
{
    Queue<BinaryTreeNode<int>*> q;
    q.push(target);
    map<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;
}
```

@Aashish Kumar Nayak

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 ~~and~~ nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes ~~to~~ inclusive at the last level h.



O(log N)

TC → O(N)
ASC → O(h)

Because it is a complete BT.

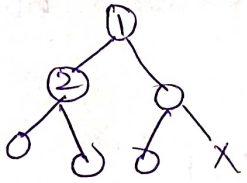
Brute force

```
inorder(node, &count)
{
    if(root == NULL)
        return;
    count++;
    inorder(root->left);
    inorder(root->right);
}
```

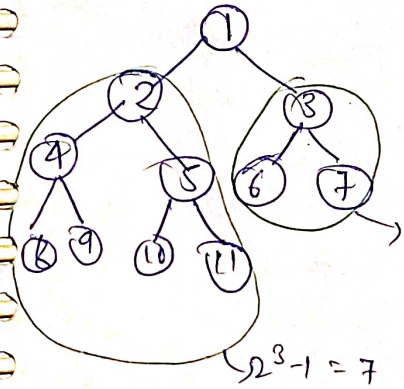
we will use a property of complete BT

∴ no. of nodes will be $2^h - 1$

But may be BT like this



still it is complete BT

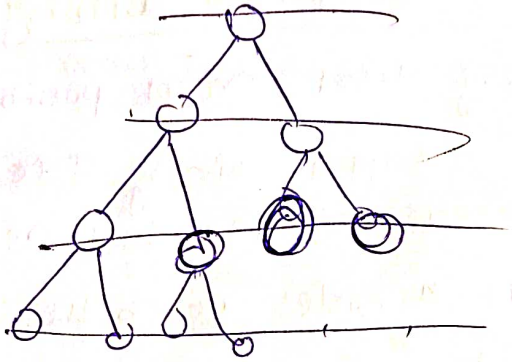


$2^2 - 1 = 3$

$2^3 - 1 = 7$

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;
}
```

$T.C. - O((\log N)^2)$

$\log N \times \log N$

For height ↓ $\log N$

$S.C. -$ Recursive space $O(N)$

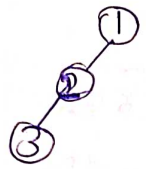
at any instant the tree we will traverse $\log N$

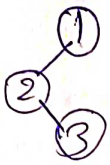
Requirements needed to construct a unique Binary Tree :-

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

(NLR) preorder 1 2 3

(LRN) postorder 3 2 1





Preorder 1 2 3

postorder 3 2 1

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

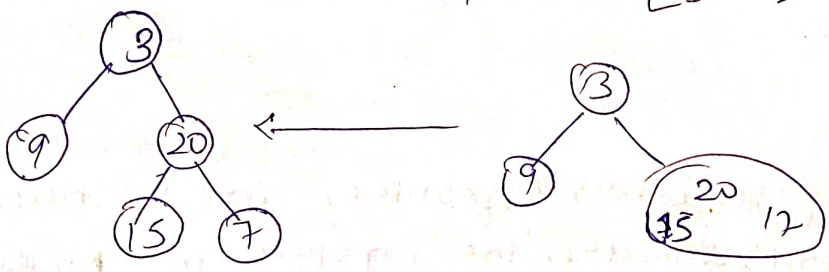
Now if given

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]

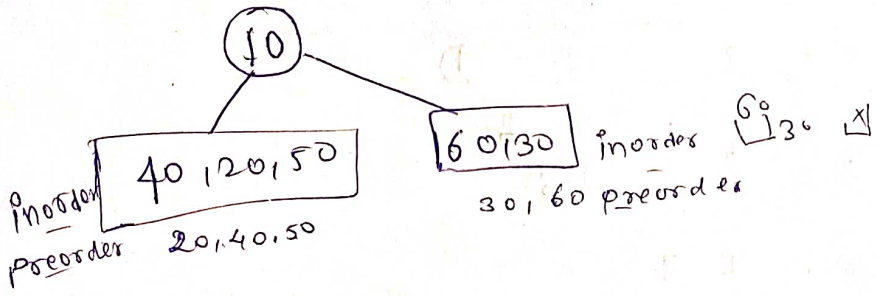


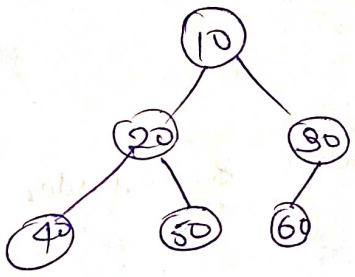
To create a unique B.T. it is necessary to root and about its left and right.

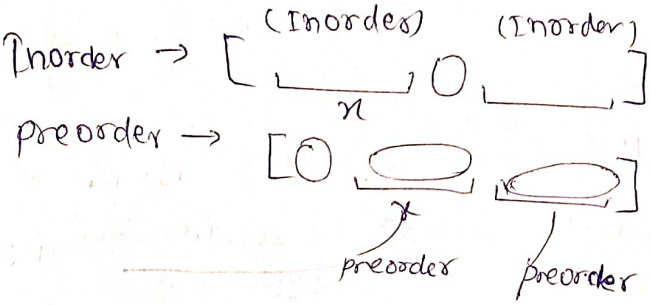
So we can use Inorder & Preorder to create unique B.T.

Construct Binary Tree from Inorder & preorder

(LNR) inorder - [40 20 50 10 60 30]
(NLR) preorder - [10 20 40 50 30 60]








```
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
{   map<int,int> inMap;
    for(int i=0; i<inorder.size(); i++)
    {   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]);
```

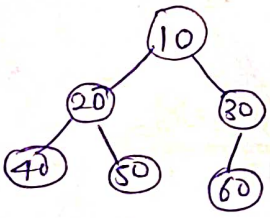
```
int inRoot = inmap[root->val];
int numleft = inroot - instart;

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

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

return root;
}
```

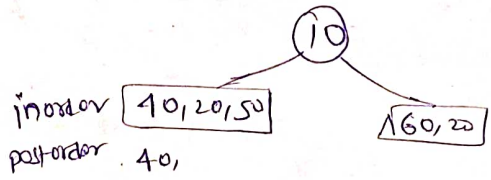
Construct a Binary Tree from postorder and Inorder :-

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

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

root

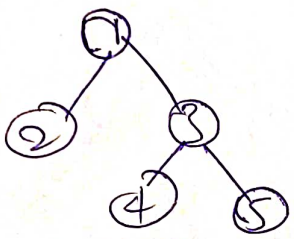





```
TreeNode* buildTree(vector<int> &inorder, vector<int> & postorder)
{
    if(inorder.size() != postorder.size())
        return NULL;
    map<int,int> hm;
    for(int i=0; i<inorder.size(); ++i)
    {
        hm[inorder[i]] = i;
    }
    return buildTreepostIn(inorder, 0, inorder.size()-1,
postorder, 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 = inRoot - is;
  root->left = buildtreepostIn(inorder, is, inRoot-1, postorder,
ps, ps+numsLeft-1, hm);
root->right = buildTreepostIn(inorder, inRoot+1, ie, postorder,
  ps+numsLeft, pe-1, hm);
    return root;
}
```

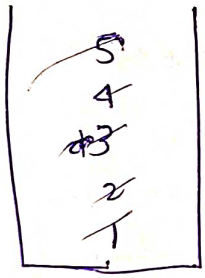
Serialize and De-Serialize Binary Tree



root → Serialize function (logic) → String → De-serialize function (reverse logic) → root → tree

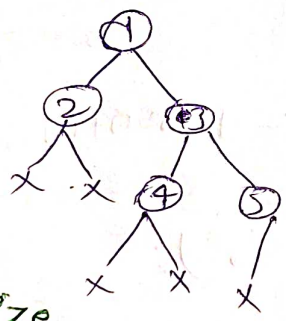
level order traversal

String :- 1 2 3 #,#,4,5,#,#,#,# (NULL)



Queue

1
2
4



It is 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;
    stringstream S(data); // string to be iterated over as a object.
    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* 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;
}
```

stringstream is a class used for insertion & extraction of data to/from string object.

Morris Traversal (Inorder) | preorder

Concept of ↳ Threaded Binary Tree

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

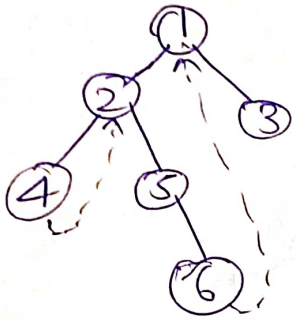
@Ashish Kumar Nayak

But

Morris traversal

T.C - O(N)

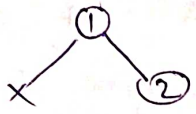
S.C. - O(1)



Inorder

4 2 5 6 1 3

1st case



left → null

print(1)

go to right.

~~vector<int> inorder;~~

~~Tree.~~

```
Vector <int> getInorder(TreeNode * root)
{
vector<int> inorder;
TreeNode *cur = root;
while(cur != NULL)
{
if(cur->left == NULL)
{
inorder.push_back(cur->val);
cur = cur->right;
}
else
{
TreeNode *prev = cur->left;
while( prev->right && prev->right != cur)
{
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;
}
```

2nd case

left ✓ right most guy on left subtree ↓

curr & curr = curr->left.

curr → make thread

exist → Remove threads

on the right most guy of left subtree if they dont have thread then make thread & curr = curr->left.

But if it has thread then remove it. and move to right

cur = cur->right.

it should not be pointing to itself.

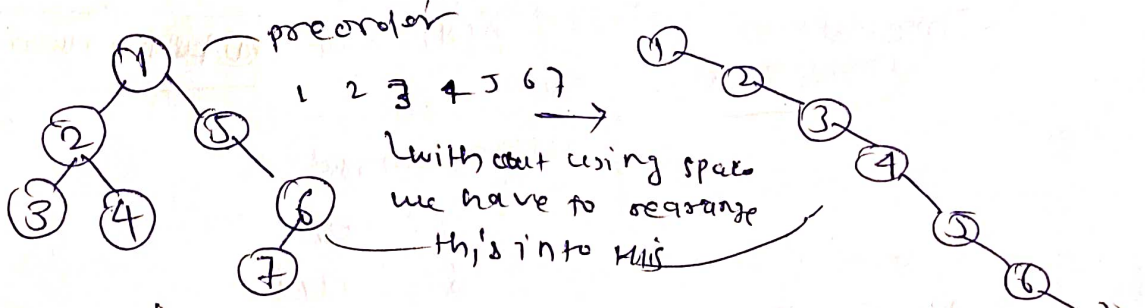
cut the thread

for preorder just 1 line change

preorder.push_back(cur.val);

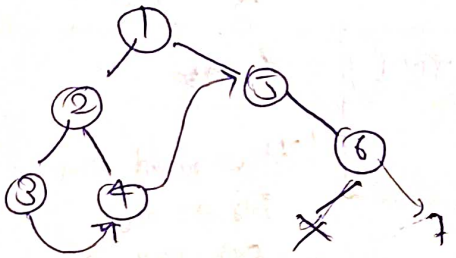
Remove & put there from here

flattening a Binary Tree to linked list :-



1 2 3 4 5 6 7

Right left Root → we can say it Reverse post order



T.C. - O(N)
S.C. - O(N) - Recursive

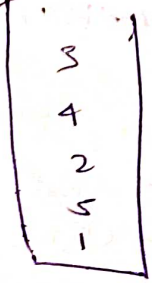
```
prev = null
flatten (node)
{ if(node == X)
      return;
  flatten(node → right);
  flatten(node → left);
  node → right = prev;
  node → left = null;
  prev = node;
}
```

using stack (another soln)

Extension of Recursion

```
St.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;
}
```

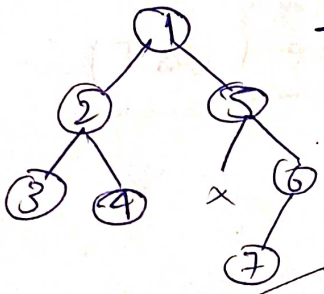
T.C. : O(N)
S.C. : O(N)



St → LIFO

@Aashish Kumar Nayak

Optimised way

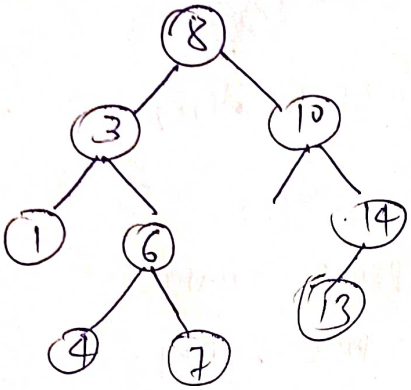


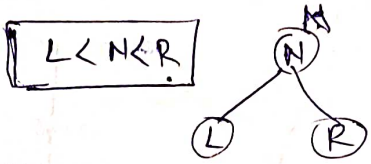
T.C. - O(N)
S.C. - O(1)

using Morris Traversal

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

Introduction to Binary Search Tree (BST) :-





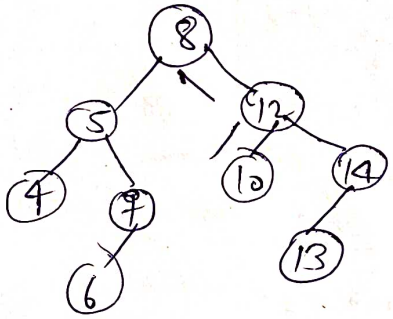
(•) left subtree → BST

(•) Right subtree → 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) :-

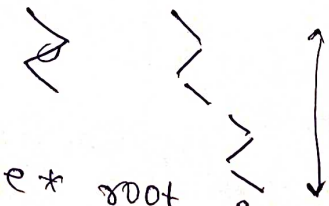


node = 10

We will cheak 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.

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

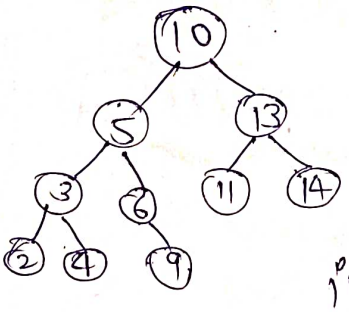
i.e. T.C. $O(\log_2 n)$



Code :-

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

Ceil in a Binary Search :



if key = 8 ans = 9

Val >= 8

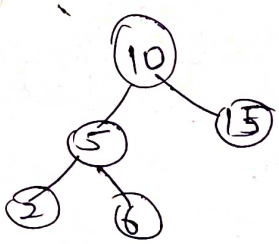
key = 11, Ans = 11

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

```
int findCeil( BinarySearchTree<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;
    }
  }
  return ceil;
}
```

Floor in a Binary search Tree

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

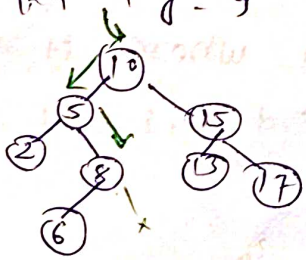


key = 7, Ans = 6

key = 14, Ans = 10

key = 9, Ans = 6

let key = 9



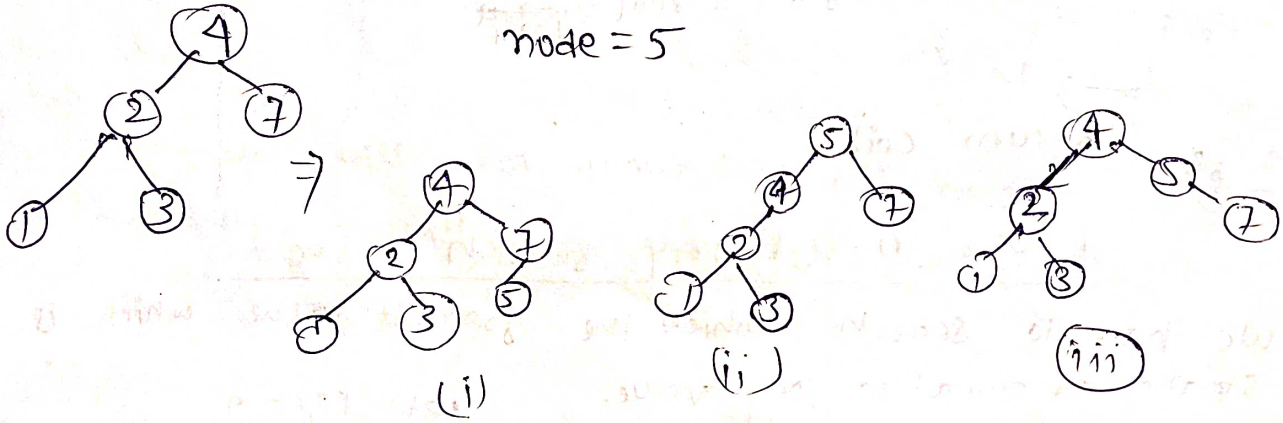
@Ashish Kumar Nayak

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

node = 5

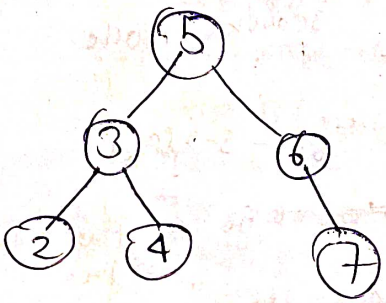


find where it can be insert and this will be always a leaf position.

~~TreeNode* insertBST~~

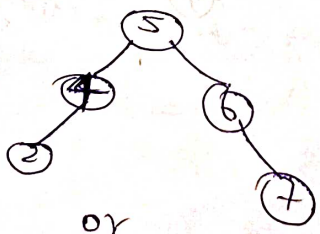
```
TreeNode* insertIntoBST (TreeNode* root, int val)
{
    if(root == NULL)
        return new TreeNode(val);
    TreeNode *cur = root;
    while (true)
    {
        if(cur->val <= val)
        {
            if( cur->right != NULL) cur = cur->right;
            else{
                cur->right = new TreeNode(val);
                break;
            }
        }
        else
        {
            if(cur->left != NULL )(cur= (cur->left;
            else{
                cur->left = new TreeNode(val);
                break;
            }
        }
    }
    return root;
}
```

Delete a Node in BST :

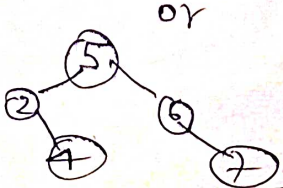


→ Delete
→ rearrange
→ return

node = 3

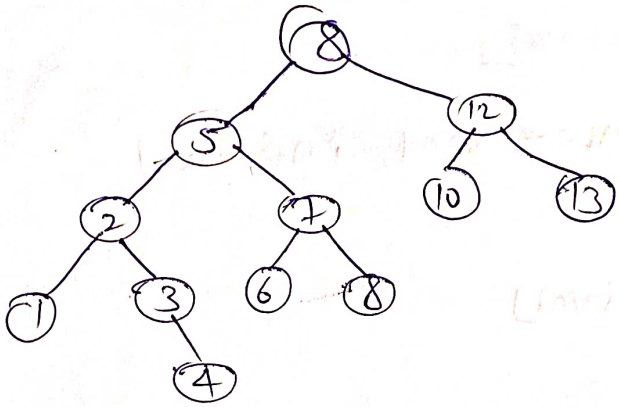


or



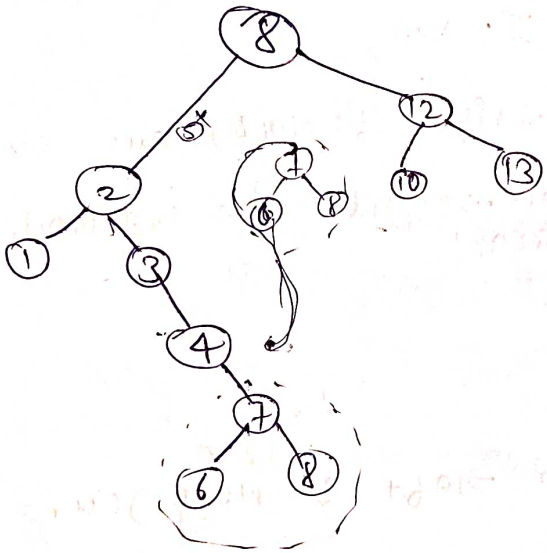
(i) Search node
(ii) Delete node

@Ashish Kumar Nayak

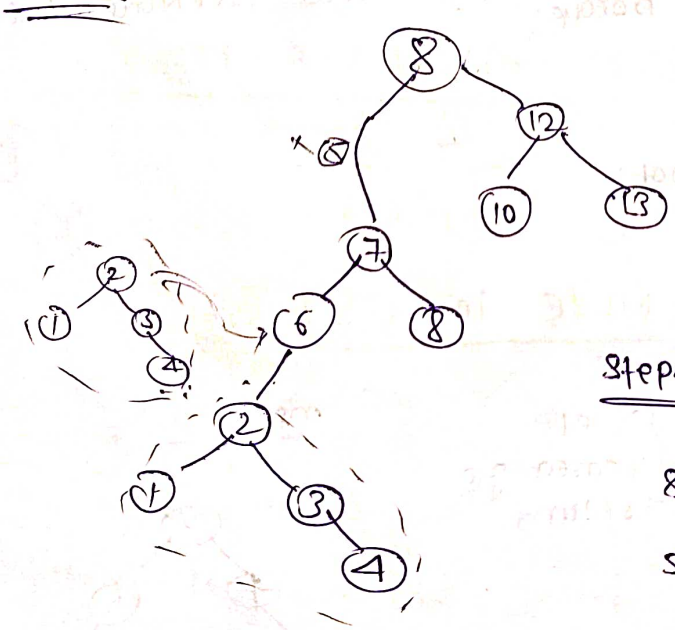


node = 5

1st way



2nd way



Steps :- search node

8 → left = 5 → left

5 ~~→ right~~ go to the extreme right.

4 → right = 5 → right

e ✓

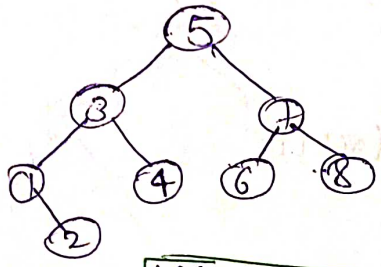
```
TreeNode* deleteNode(TreeNode* root, int key)
{
    if(root == NULL) return NULL;
    if(root->val == key) return helper(root);
    TreeNode* dummy = root;
    while(root != NULL)
    {
        if(root->val > key)
        {
            if(root->left != NULL && root->left->val == key)
           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

T.C. = O(Height of Tree)

S.C. = O(1)

@Aashish Kumar Nayak

K^th Smallest element in BST :-



K = 3

1 2 3 4 5 6 7 8

↑ you need to return

Approach 1

(i) → Do any DFS traversal

(ii) → Store all the node value in vector
→ Sort the vector
→ then return (K-1)th index value.

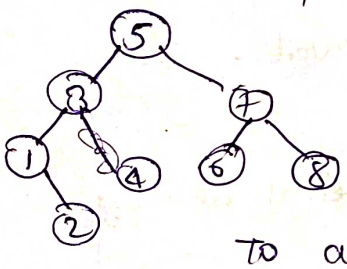
T.C. – O(N) + O(NlogN) (Traverse / sort)
S.C. – O(N) (vector)

Efficient approach

*** Property of BST

Inorder of Binary Search Tree is always in sorted order

inorder 1 2 3 4 5 6 7 8



T.C. – O(N)
S.C. – O(N) – vector

To avoid S.C. O(N) of vector.

Count = 0 left Node Right

```
count++        if( count == key)
                  ans = node;
```

Recursive } T.C. – O(N)
S.C. – O(N)

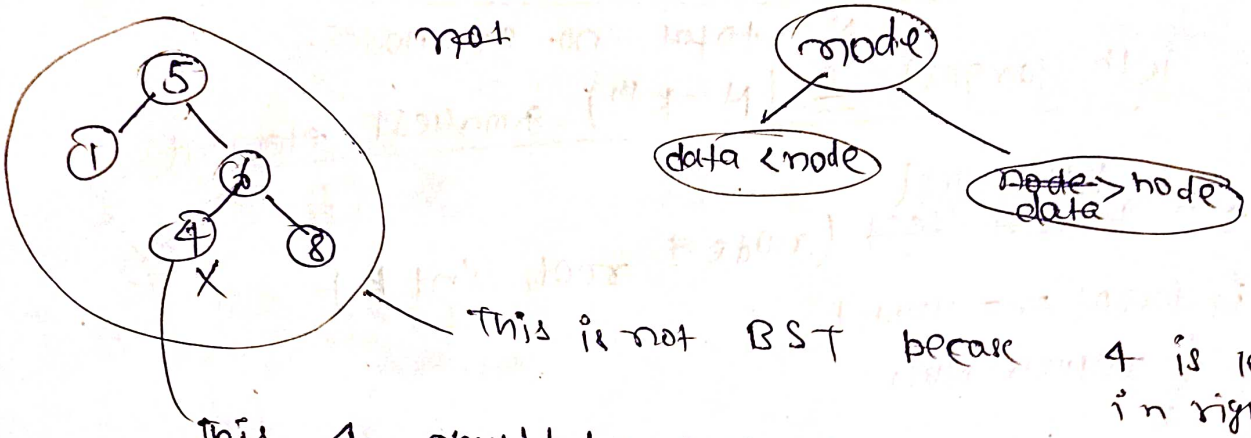
But if we do morris traversal

T.C. – O(N)
S.C. – O(1)

→ attached this piece of code. ✓

@Aashish Kumar Nayak

Cheak if a tree is BST or BT :



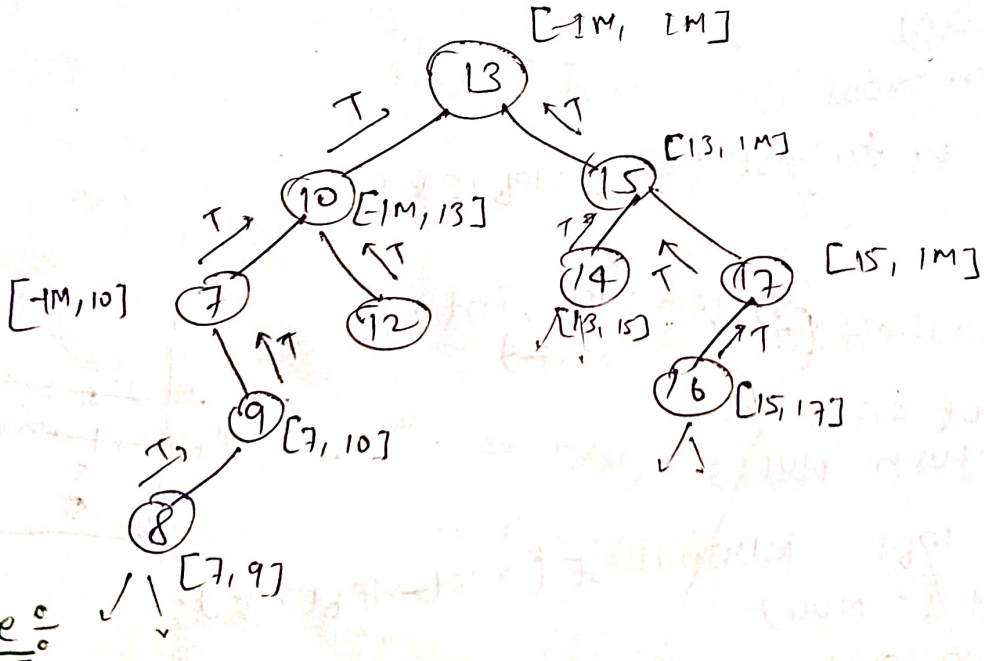
this is not BST because 4 is less than 5 in right of 5.

This 4 should be lessed than 6 and greater than 5.

Always maintain a range in this problem

[-INT_MIN, INT_MAX]

int MIN — int Max



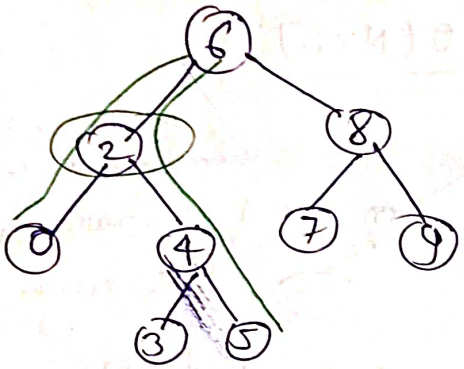
Code :

```
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);
}
```

@Aashish Kumar Nayak

LCA in a BST :-

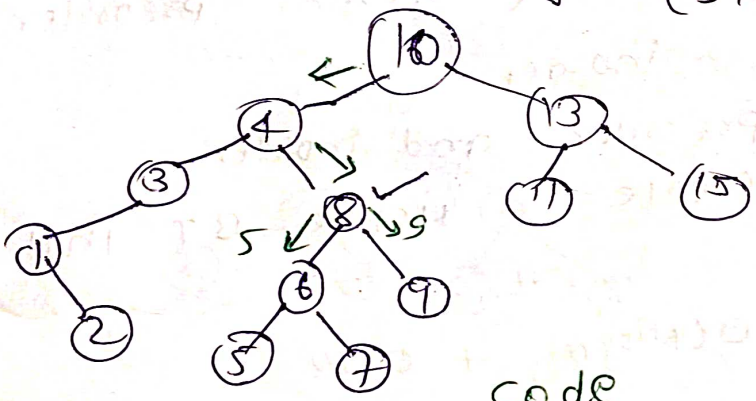
Lowest common Ancestor



first intersection point from the bottom in the Path.

we will traverse in BST and at the moment where both the element are not lies in left or right. we will return that node.

let say (5, 8)



T.C. O(H)
S.C. O(1)

code

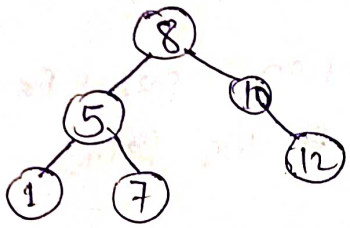
```
TreeNode * lowestCommonAncestor (TreeNode* root, TreeNode* p,
TreeNode* q)
{   if(root == NULL)
        return NULL;
    int curr = root->val;
    if(curr < p->val && curr < q->val)
    { return lowestCommonAncestor(root->right, p, q);
    }
    if(curr > p->val && curr > q->val)
    { return lowestCommonAncestor(root->left,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

@Aashish kumar Nayak

Construct a BST from preorder traversal

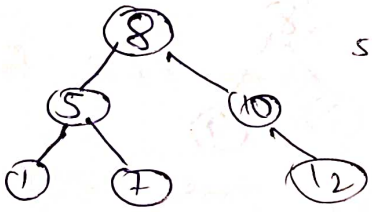
3 Methods

Preorder → { 8 5 1 7 10 12 }



1st method

T.C.
O(N×N)



5 is smaller so left
1 is smaller so left.
7 is greater than 5 so right
10 is greater than 8 so right,
12 is greater so right.

2nd Method

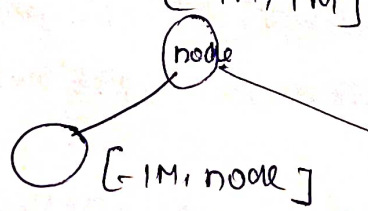
property

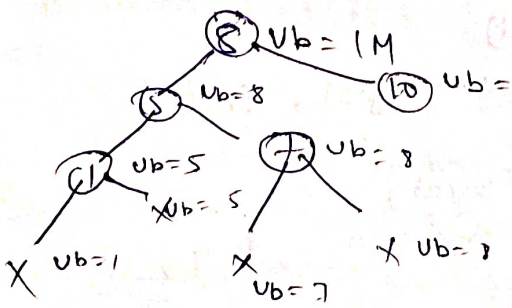
BST → inorder → sorted

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.

T.C. = O(NlogN) (sort) + O(N)
S.C. = O(N) (vector inorder)

3rd Method :-





T.C. → O(3N) ≈ O(N)

S.C. → O(1)

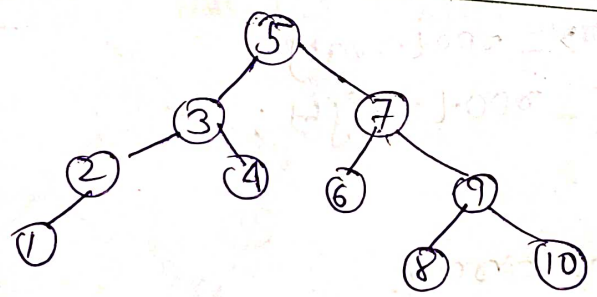
@Aakash Kumar Nayak

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

3 methods



node = 8

inorder 1 2 3 4 5 6 7 8 9 10

Node → Successor

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.

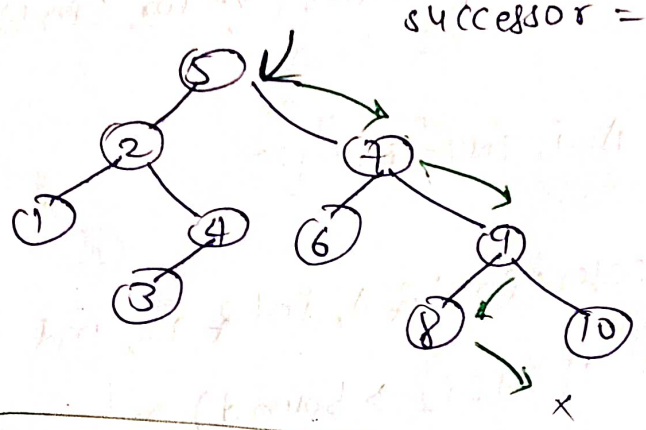
T.C. — O(N)
S.C — O(1)

@Ashish Kumar Nayak

Method 3 :-

let use a variable and initialize it as null.

successor = null.



val = 8

Successor = ~~5~~ ~~7~~ ~~9~~

traverse till we get NULL than return successor.

Code

T.C. = O(H) = O(log n)

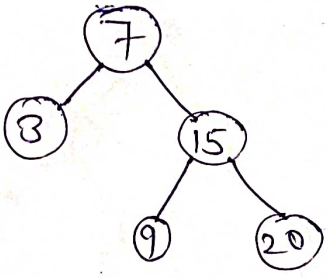
S.C. = O(1)

Code :-

```
TreeNode * inorderSuccessor(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;
}
```

5 4
5 7
1 - 2 3
3

BST Iterator :-

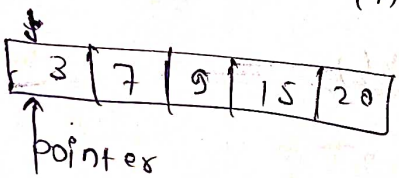


inorder 3 7 9 15 20

BSTIterator(7)
next → 3
next → 7
hasNext → True
next → 9
hasNext → True
next → 15
hasNext → True
next → 20
hasNext → false

~~Method 1 :-~~

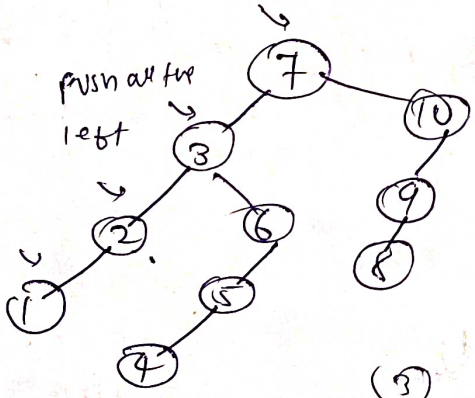
Store inorder in a vector



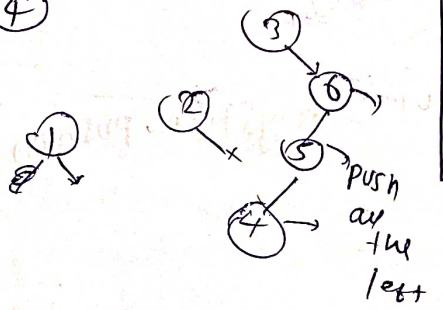
T.C. — O(1) for next
S.C. — O(N)

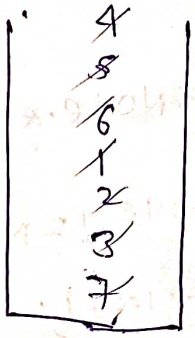
But if you are not allowed to store inorder.

Method 2 :- ~~Use~~ we will use stack



T.C → O(1) $(N/N) = O(1)$
S.C → O(H)



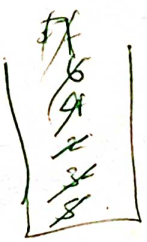


Stack LIFO

for cheaking hasNext
~~Use Stack~~ we cheak stack wheater it is empty or not.

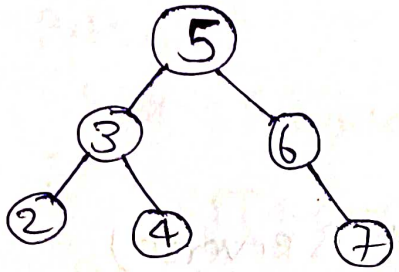
Code

```
class BSTIterator {
    private:
        stack<TreeNode*> myStack;
    public:
    BSTIterator (TreeNode * root)
    {
        pushAll(root);
    }
    bool hashNext()
    {
        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);
    }
};
```



@Aashish Kumar Nayak

Two sum in BST | Check if there exists a pair with sum K



K = 9

3+6 = 9 ✓ True

5+4 = 9 ✓ ~~Pair Found~~

K = 4

4 ✗ should be in pair.

false

Two sum problem

inorder Now it becomes Two sum problem.

2 3 4 5 6 7

Two pointer approach

BST to Array Vector store

Two pointer

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

S.C = O(N) — vector

Just like previous problem

push all right node instead of left in stack

next() ✓ before()

i = 1st j = last

it is same like Two pointer approach.

1 2 3 4 5 11

Answer is

T.C. → O(N)

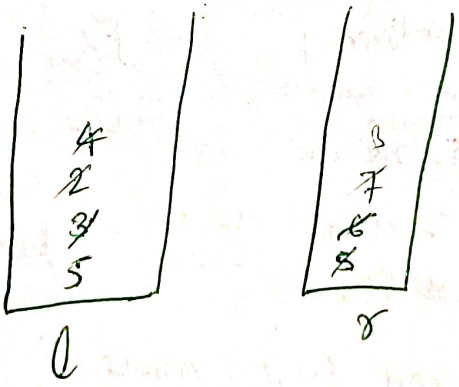
S.C → O(H)×2 ≈ O(H) = O(log n)

Code :

↓

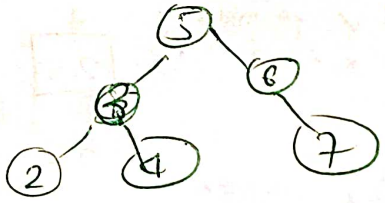
```
class BSTIterator
{
  stack<TreeNode *> myStack;
    bool reverse = true;
  public:
    BSTIterator( TreeNode *root , bool isReverse)
    {
        reverse = isReverse;
        pushAll(root);
    }

  bool hasNext().
  {
      return !mystack.empty();
  }

  int next()
  {
    TreeNode * tmpNode = myStack.top();
      myStack. pop().
   if(! reverse)
      pushAll(tmpNode -> right);
   else
      pushAll( tmpNode -> left);
      return tmpnode
   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();
    }
    return false;
  }
};
```



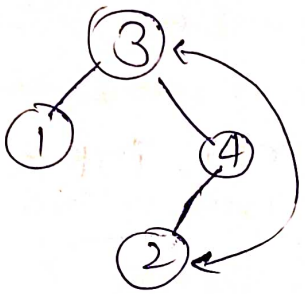
i j

i i i j j j
2 3 4 5 6 7



i j
2 + 7 = 9
3 + 7 = 10
3 + 6 = 9
4 + 6 = 10
4 + 5 = 9

Recover BST

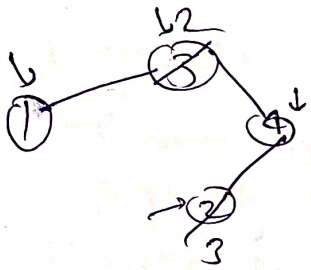


Two nodes swapped

we have to correct the BST.

→ Do Inorder traversal
→ sort the traversal → it will be incorrect Inorder of BST.

1 2 3 4



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

2 traversing sort

T.C. - O(2N) + NlogN)

S.C. — O(N)

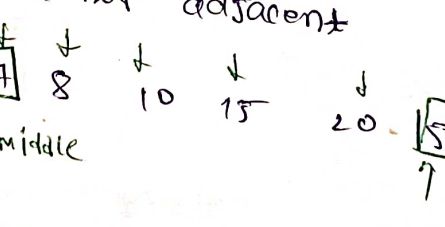
vector to store the Inorder.

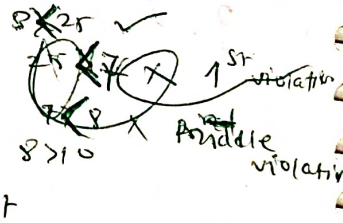
nodes 5 & 25

Swap can have two cases :-

1: Swapped nodes are not adjacent

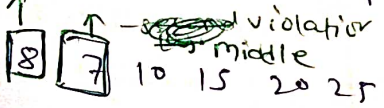
25 > 3 ✓ 1st middle
7 > 25 ✗ 25
8 > 7 ✓
10 > 8 ✓
15 > 10 ✓
20 > 15 ✓
5 > 20 ✗ last violation





2. Swapped nodes are adjacent

nodes 7 & 8



T.C. - O(N)

S.C. - O(1) - Morris traversal

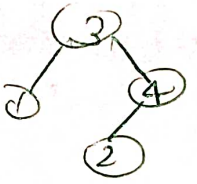
5 > 3 ✓
8 > 5 ✓ middle first
7 > 8 ✗ 7 8
10 > 7 ✓
15 > 10 ✓
20 > 15 ✓
25 > 20 ✓

if we dont get last violation then Swap first & middle node.

or

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);
  }
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);
  }
};
```

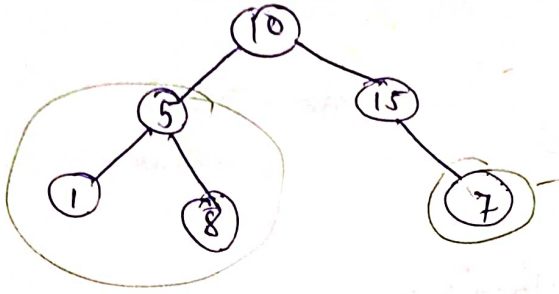


1 2 3 4

1 3 2 4

@Aashish kumar Nayak

largest BST in BT



BST of size 1

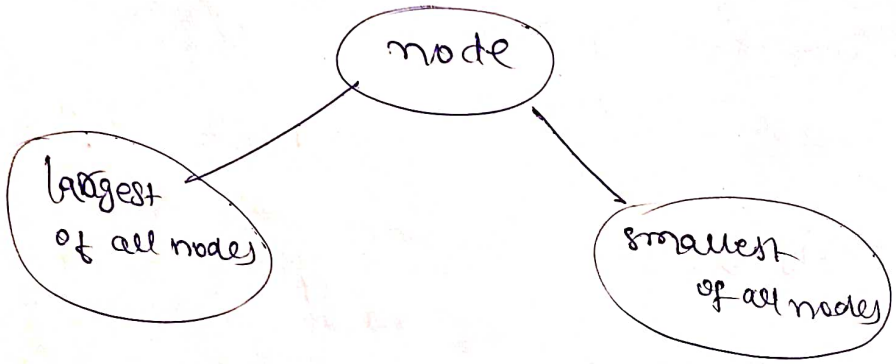
BST of size 3

Brute force

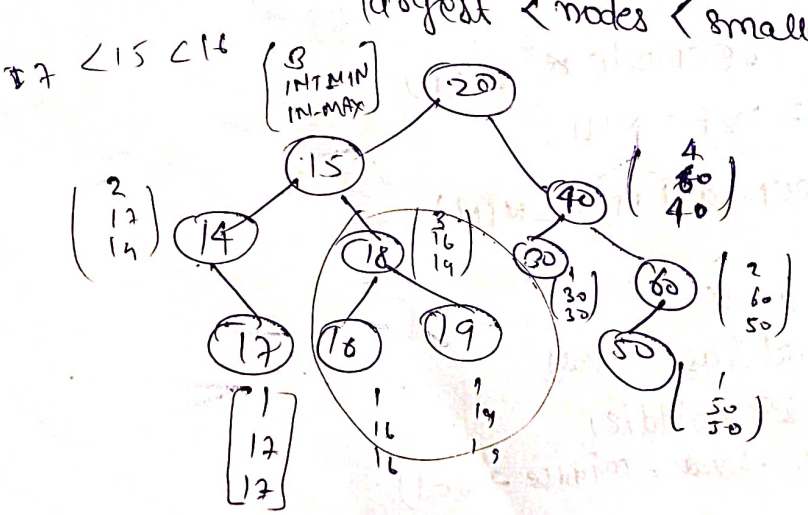
validate a BST T.C. O(N) X O(N) → Traverse every node. ← Validate

check every node for valid BST or not

then send the root to the function which return ⊕ total no. of nodes.



largest < nodes < smallest



(0, INT_MIN, INT_MAX)

@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;
  }
};
```