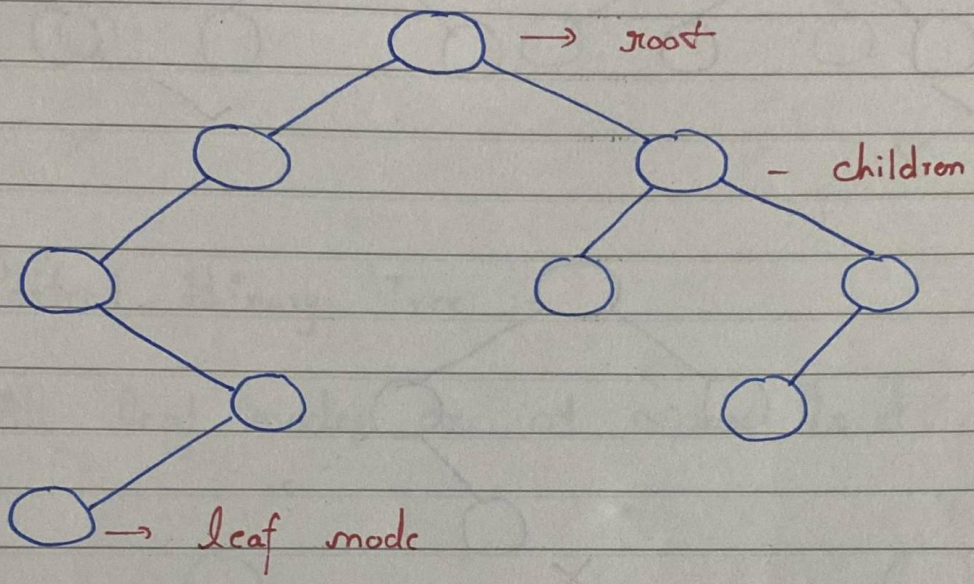
Trees

⊙ Introduction to Binary Trees



⊙ Types of Binary Trees :

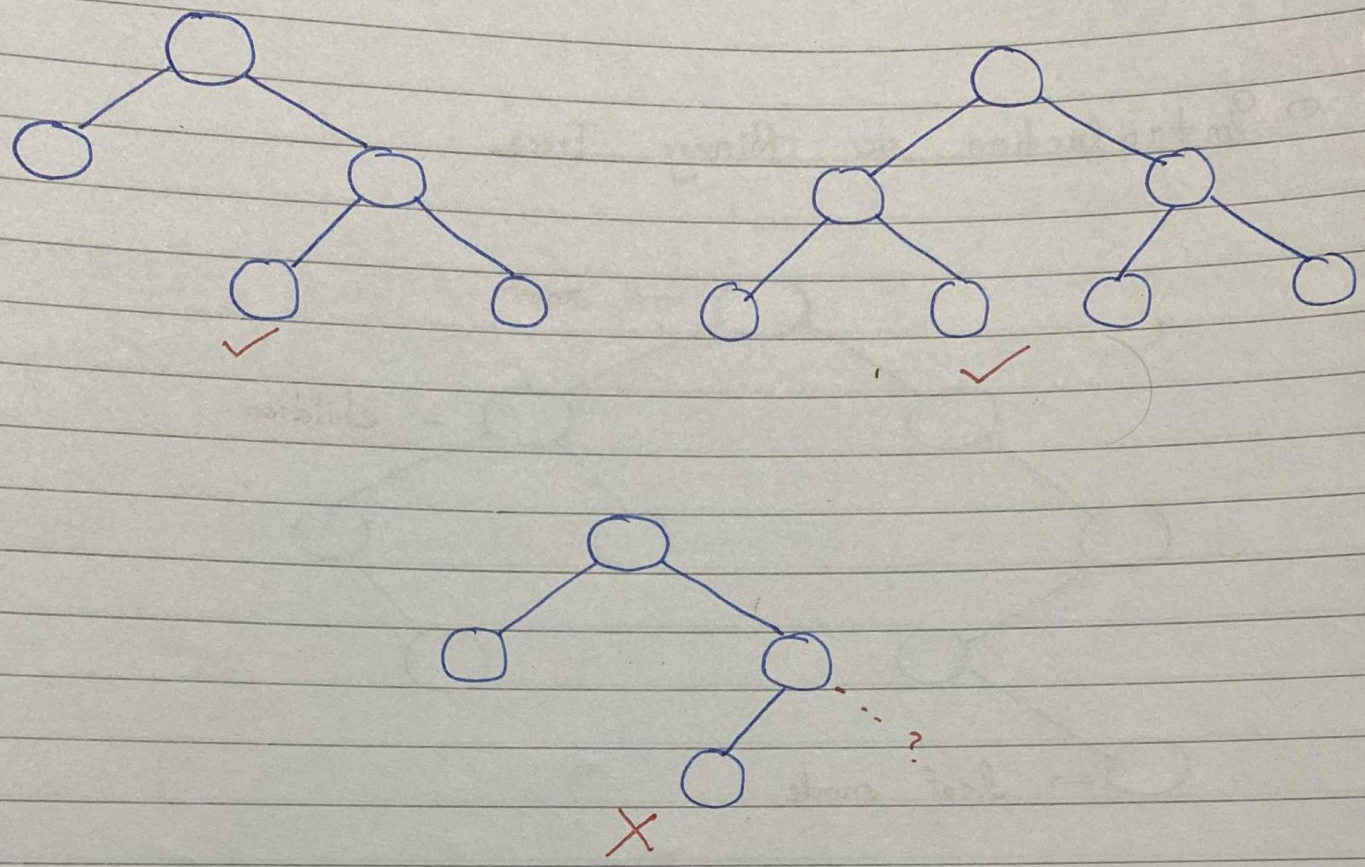
(·) Full Binary Tree
(·) Complete Binary Tree
(·) Perfect Binary Tree
(·) Balanced Binary Tree
(·) Degenerate Binary Tree

① Full Binary Tree :-

- Each node has either 0 or 2 children.

P.T.O →

Eg:

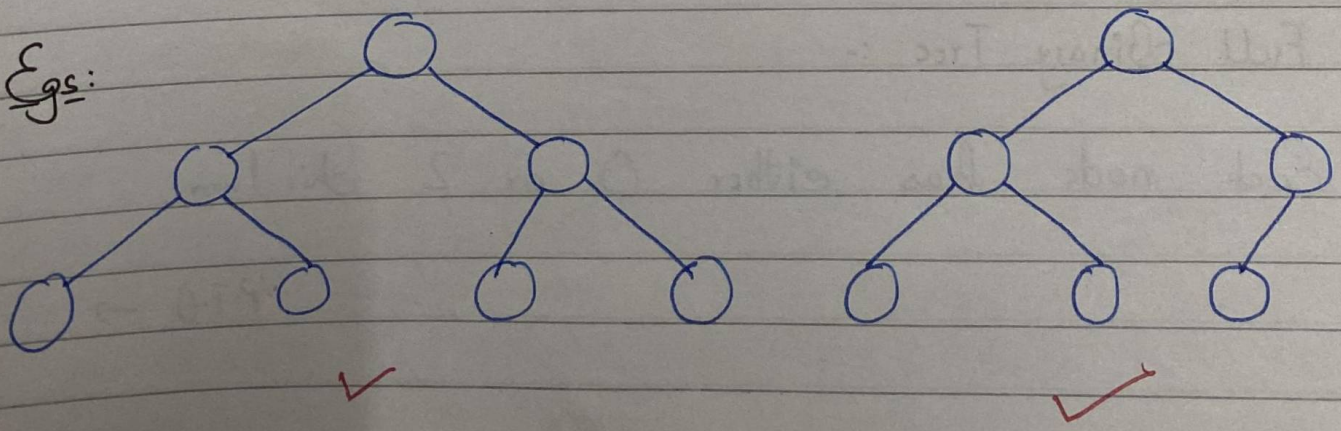


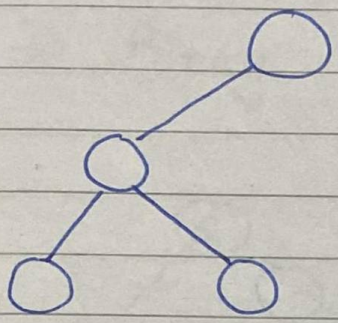
(11) Complete Binary Tree :-

(a) All levels are completely filled except the last level.

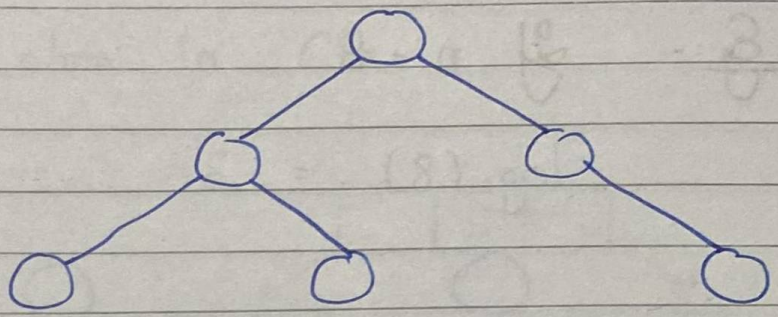
(b) the last level has all nodes in left as possible.

Eg:





X

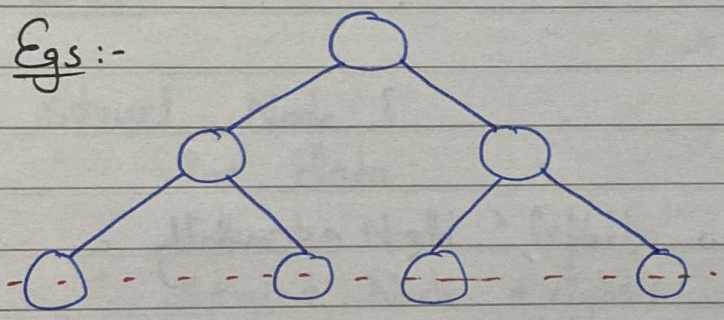


X

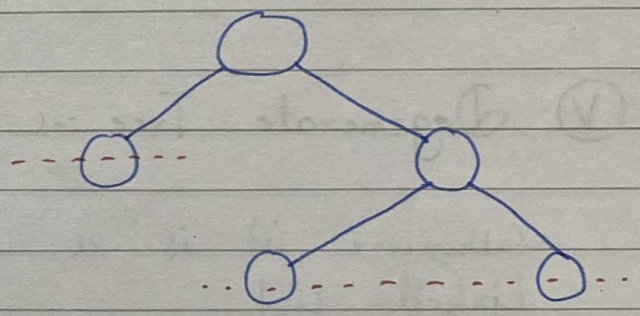
(III) Perfect Binary Tree :-

All leaf nodes are at same level.

Eg:-



✓



X

(IV) Balanced Binary Tree :-

Height of a tree at max $\log(N)$

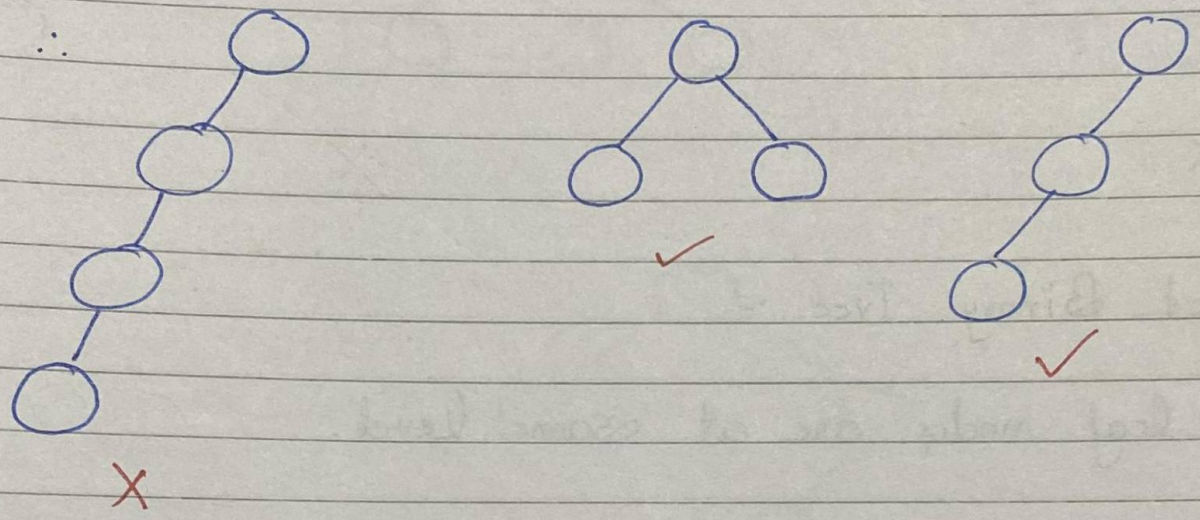
where N → no. of nodes

P.T.O →

Eg :- If $n = 8$

$\log_2(8) = 3$

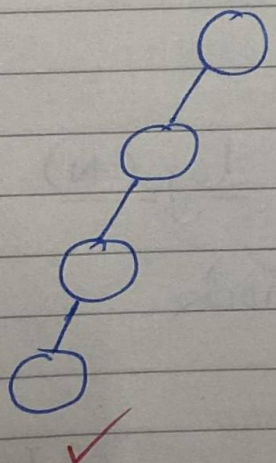
$\therefore$



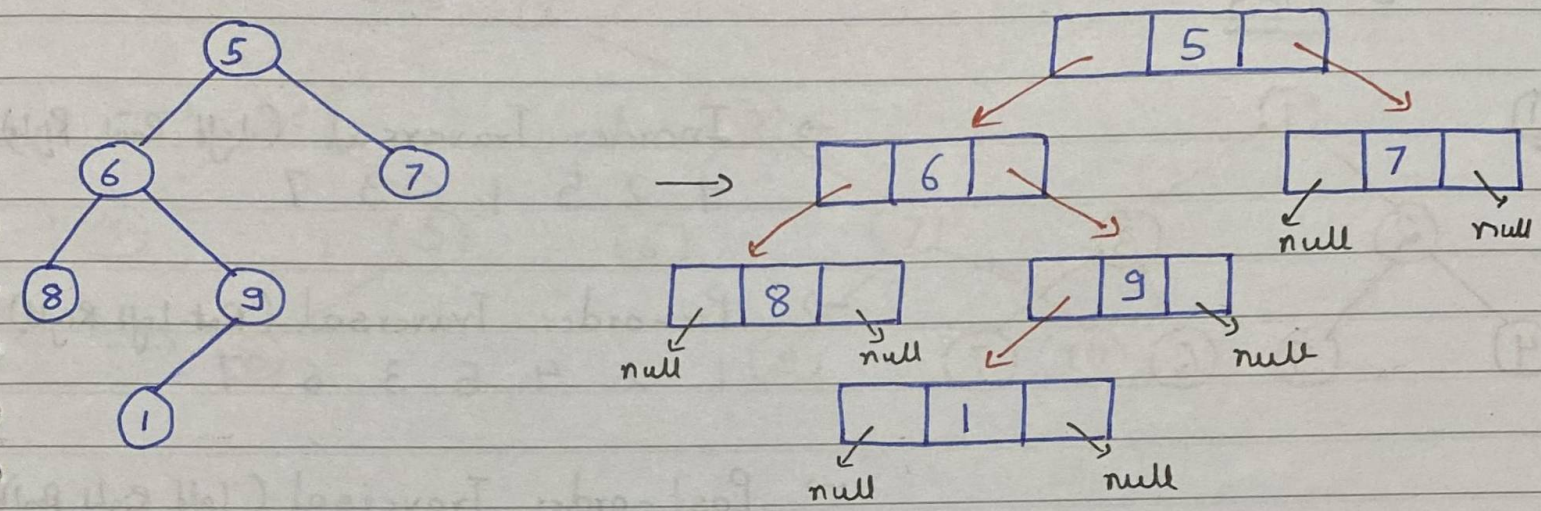
(V) Degenerate Tree :-

Whenever it is a skew tree, ide essentially a Linked List.

For eg: If $n = 4$

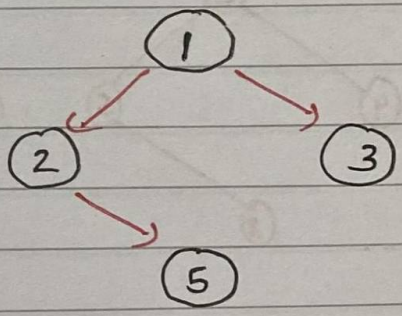


⊙ Binary Tree Representation in C++



CODE :-

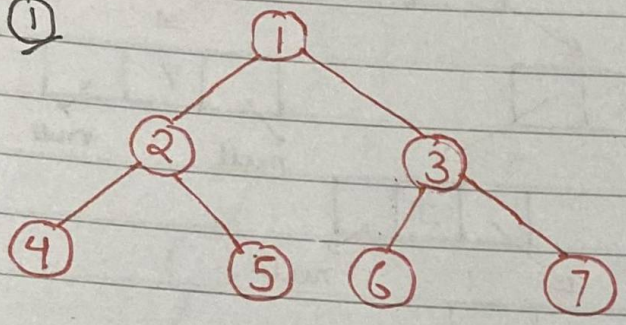
```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;

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

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



⊙ Traversal Techniques (BFS / DFS)

- DFS

(i)

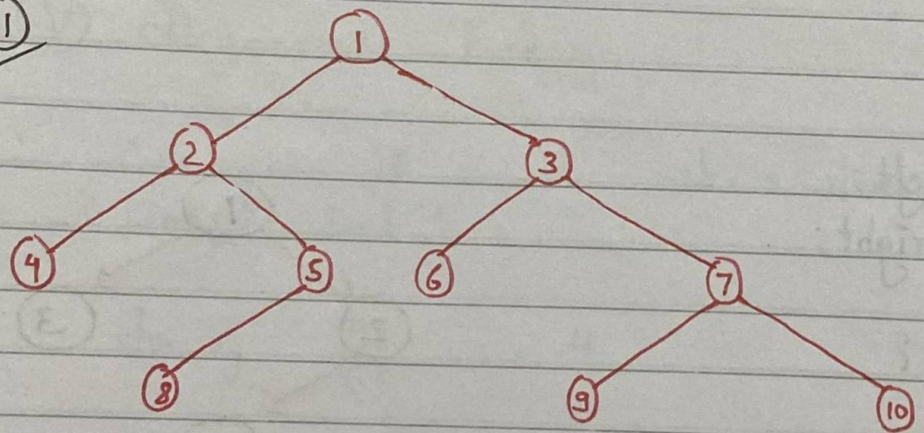


→ Inorder Traversal (Left Root Right)
4 2 5 1 6 3 7

→ Pre-order Traversal (Root Left Right)
1 2 4 5 3 6 7

→ Post-order Traversal (Left Right Root)
4 5 2 6 7 3 1

(ii)

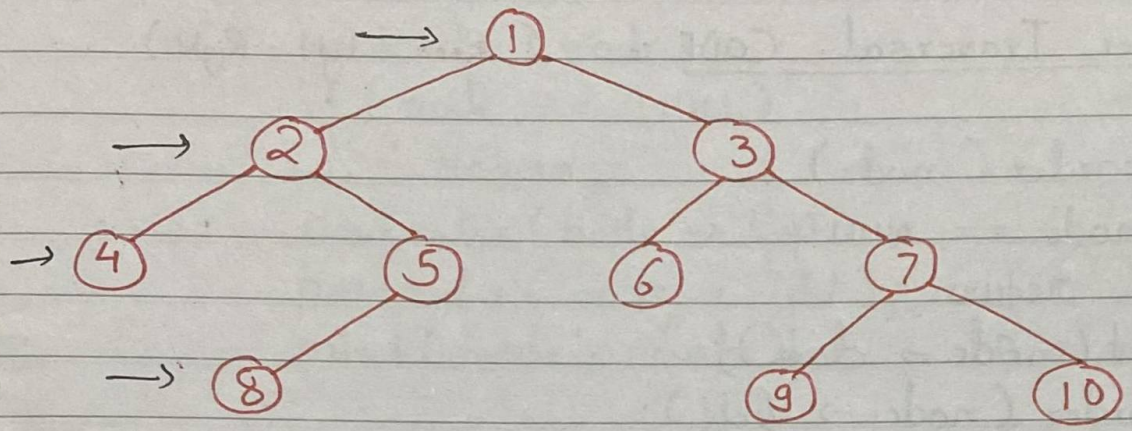


→ Inorder Traversal (Left Root Right)
4 2 8 5 1 6 3 9 7 10

→ Pre - order Traversal (Root Left Right)
1 2 4 5 8 3 6 7 9 10

→ Post - order Traversal (Left Right Root)
4 8 5 2 6 9 10 7 3 1

- **BFS**



BFS → 1 2 3 4 5 6 7 8 9 10

(Level / Breadth wise)

⊙ DFS

① Pre - Order Traversal CODE :- (Root Left Right)

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

* TC → O(N) , where N is no. of nodes.
  SC → O(N)
  ↳ worst height will be of a skew tree.

⑪ In - order Traversal CODE :- (Left Root Right)

```
void inorder ( node ) {
    if ( node == NULL )
        return ;
    inorder ( node → left );
    print ( node → data );
    inorder ( node → right );
}
```

TC and SC will be the same.

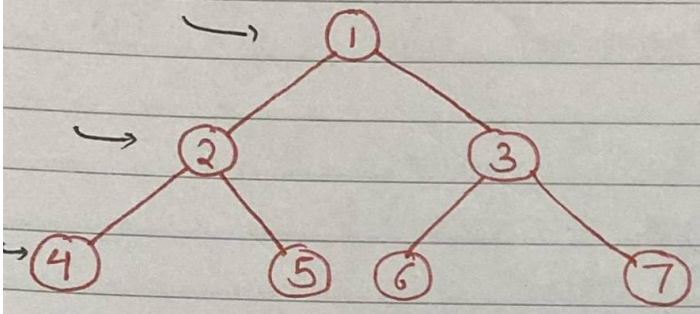
→

(iii) Post-order Traversal CODE :- (Left Right Root)

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

TC and SC same.

⊙ BFS (Level Order Traversal)

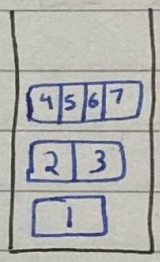


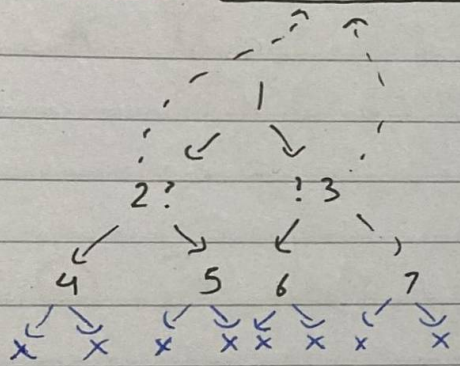
O/P → 1 2 3 4 5 6 7

We'll need a queue and a vector of vector to store the result level-wise.

Queue | 1 2 3 4 5 6 7 |

Result →





∴ Result is stored in our vector<vector<int>> result.

Whenever the queue is empty we are done with our iteration.

→

CODE :-

```
vector<vector<int>> levelorder (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->value);
             }
             ans.push_back(level);
        }
        return ans;
}
```

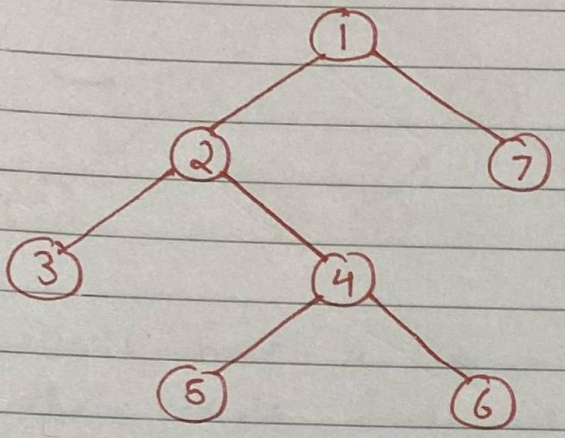
NOTE → TreeNode is an already created structure to store the node data.

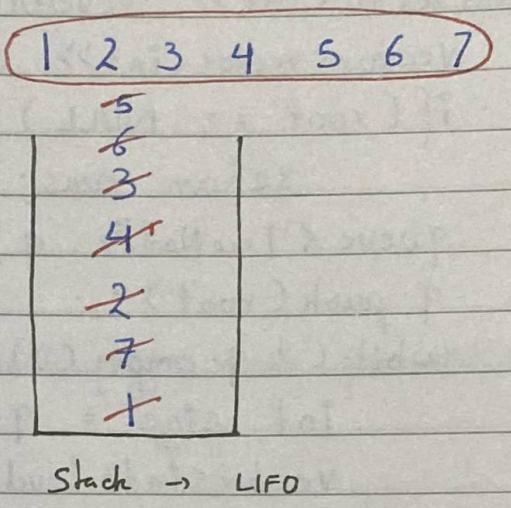
TC → O(N)
SC → O(N)

P.T.O →

→ (L9) - Iterative Pre-order Traversal

Root Left Right.



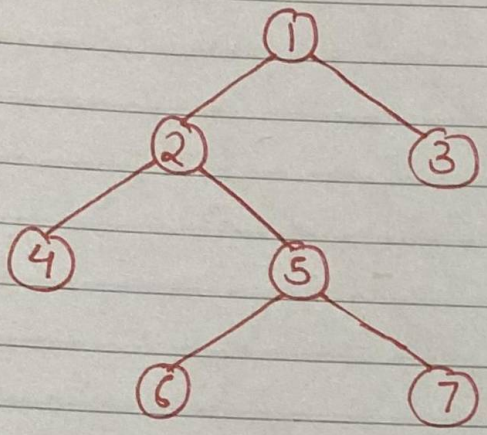


CODE:-

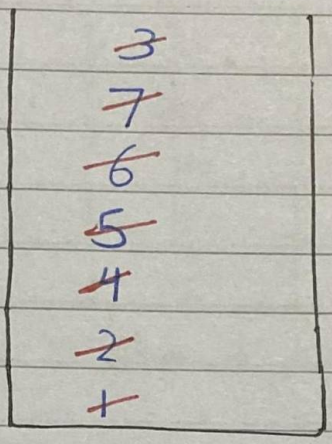
```
vector<int> preorder(root) {
        vector<int> ans;
        if(root == NULL) return ans;

        Stack<TreeNode*> st;
        st.push(root);
        while(!st.empty()) {
            root = st.top();
            st.pop();
            ans.push_back(root->val);
            if(root->right != NULL) st.push(root->right);
            if(root->left != NULL) st.push(root->left);
        }

        return ans;
}
```

→ 110 - Iterative Inorder Traversal

↳ Left Root Right



4 2 6 5 7 1 3



Stack → LIFO

↳ empty → break

node → 1 2 4 ~~null~~

~~null~~

~~5~~

~~6~~

~~null~~

~~null~~ → ~~null~~ ~~null~~ ~~3~~ ~~null~~ null

CODE :-

```
vector<int> inorder(root) {
    Stack<TreeNode*> st;
    TreeNode *node = root;
    vector<int> ans;
    while(true) {
        if(node != NULL){
            st.push(node);
            node = node->left;
        }
        else
        {
```

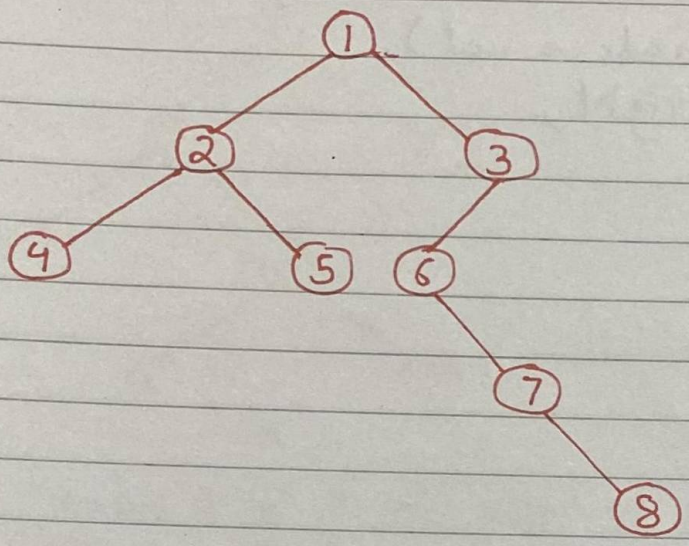
```
            if(st.empty()) break;
            node = st.top();
            st.pop();
            ans.push_back(node->val);
            node = node->right;
        }
    }
    return ans;
}
```

TC → O(N)

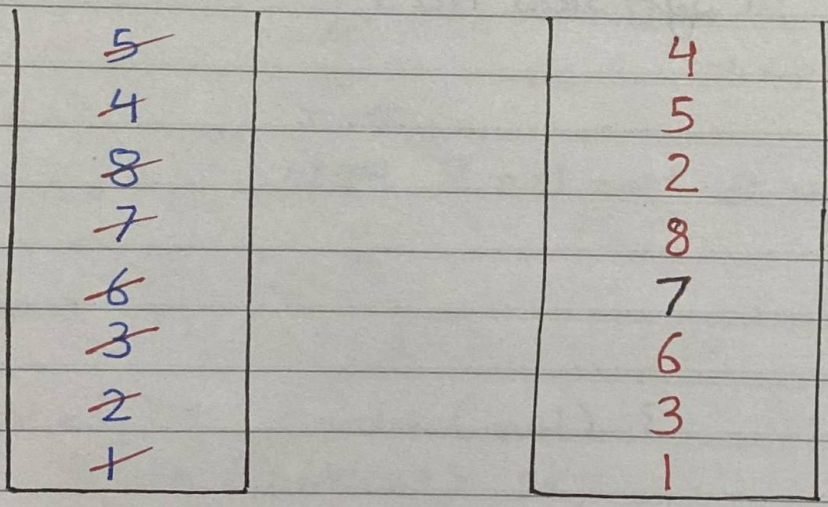
SC → O(N) → [Left Skew Tree]

→ (L11) - Iterative Postorder Traversal (Using 2 Stacks)

↳ Left Right Root



→ 4 5 2 8 7 6 3 1



St 1 St 2

pop()

4 5 2 8 7 6 3 1

=

CODE :-

```
vector<int> postorder(TreeNode *root) {
    vector<int> ans;
    if(root == NULL) return ans;

    stack<TreeNode*> st1, st2;
    st1.push(root);

    while(!st1.empty()) {
        root = st1.top();
        st1.pop();
        st2.push(root);
        if(root->left != NULL) st1.push(root->left);
        if(root->right != NULL) st1.push(root->right);
    }

    while(!st2.empty()) {
        ans.push_back(st2.top()->val);
        st2.pop();
    }

    return ans;
}
```

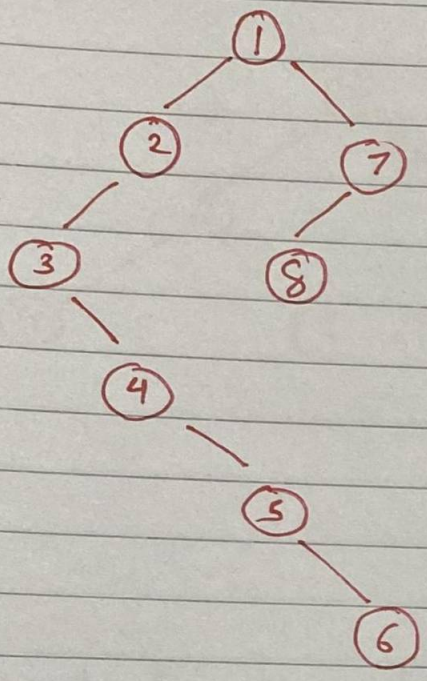
TC → O(N)

SC → O(2N)

↳ 2 Stacks

6 5 4 3 2 8 7 1

→ L 12 - Iterative Postorder Traversal (Using 1 stack)



curr = 1 2 3 null
4 null 5
null 6 null
7 8 null

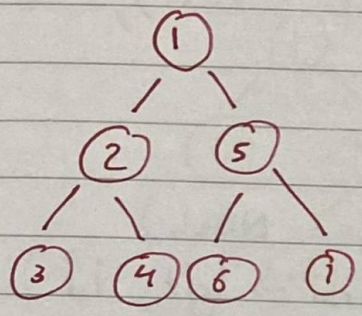
temp = 4 5 6 null
6 5 4 3
null 2 7 null 8 null 7 1

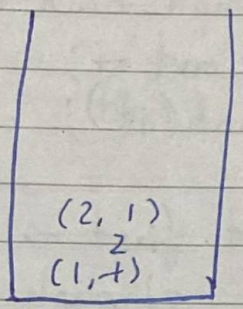
~~8~~
~~7~~
~~6~~
~~5~~
~~4~~
~~3~~
~~2~~
~~1~~

CODE:

```
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();
                post.add (temp);
                while ( !st.empty() && stemp == st.top()->right) {
                    temp = st.top(); st.pop();
                    post.add ( temp -> val);
                }
            } else {
                curr = temp;
            }
        }
```

→ L13 - Pre In Post in one Traversal





(node, num)

Stack

↳ LIFO

if (num = 1)
preorder
++
left

if (num == 2)
inorder
++
right

if (num == 3)
postorder

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

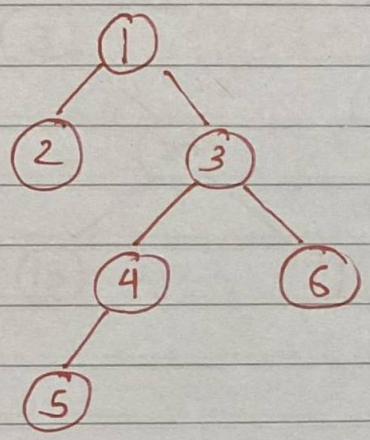
CODE :

```
void allTraversalInOne (TreeNode *root) {
    stack<pair<TreeNode*, int>> st;
    if(root == NULL)
        return;
    st.push({root, 1});

    vector<int> pre, in, post;

    while (!st.empty()) {
        auto it = st.top();   st.pop();
```

P.T.O →

```
if ( it.second == 1 ) {
        pre.push_back (it.first -> val );
        it.second ++;
        st.push(it);

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

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

→ L14 - Maximum Depth In Binary Tree



Height = 4

2 approaches

Recursive

O(height)
(Space)

Level-order Traversal

↳ O(N)
(Space)

→ Recursive

$1 + \max(l, r)$

CODE:

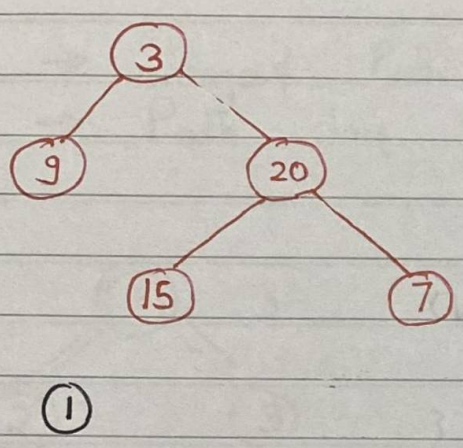
```
int maxDepth (TreeNode* root) {
    if( root == NULL) return 0;

    int left = maxDepth( root -> left);
    int right = maxDepth( root -> right);

    return 1 + max( left, right);
}
```

→ Level - Order Solution

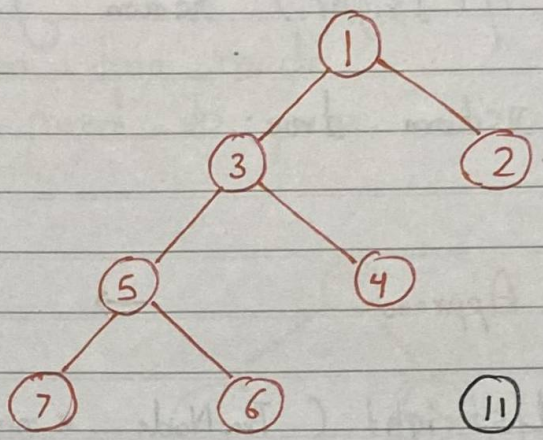
CODE :-

```
int levelOrder ( TreeNode * root ) {
        int res = 0;

        if ( root == NULL)
                return res;

        queue <TreeNode*> q;
        q.push ( root );

        while ( !q.empty ) {
                int size = q.size();
                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);
                }
                res ++;
        }

        return res;
}
```

→ L15 - Check for a balanced Binary Tree



(1)



(11)

Condition for Balanced BT → for every node,

$$|\text{height(left)} - \text{height(right)}| <= 1$$

⊙ Naive Approach → $O(N) \times O(N) \rightarrow O(N^2)$

```
bool isBalanced ( TreeNode * root ) {
    if( root == NULL ) return true;

    int lh = findHeight ( root -> left );
    int rh = findHeight ( root -> right );

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

    bool left = isBalanced ( root -> left );
    if ( ! left ) return false;
```

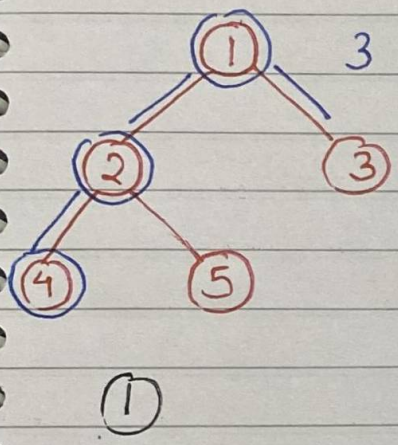
```
    bool right = isBalanced(root -> right);
    if(!right) return false;

    return true;
}
```

⊙ Better Approach → O(N)

```
int dfsHeight(TreeNode *node) {
    if(node == NULL)
        return 0;

    int lh = dfsHeight(node -> left);
    if(lh == -1) return -1;

    int rh = dfsHeight(node -> right);
    if(rh == -1) return -1;

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

    return 1 + max(lh, rh);
}

bool isBalanced(TreeNode *root) {
    return dfsHeight(root) != -1;
}
```

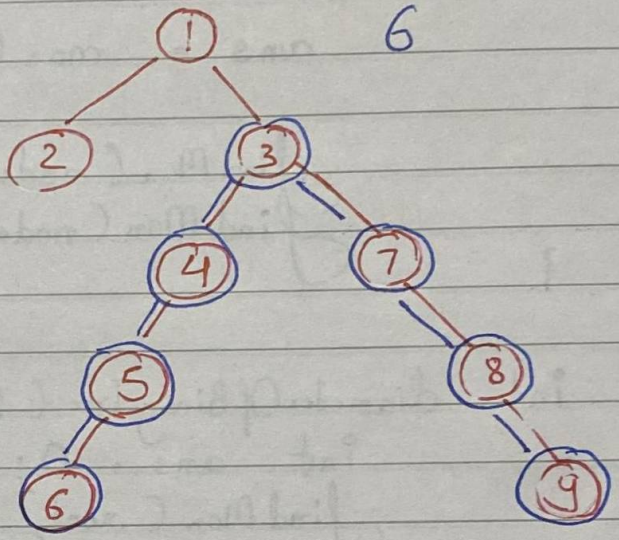
→ L16 - Diameter of Binary Tree

→ Longest Path b/w two nodes
→ Path does not need to pass via root.



3

(I)



6

(II)

⊙ Naive Approach → O(N) × O(N) → $O(N^2)$

```
int findHeight (TreeNode * node) {
    if (node == NULL) return 0;

    int lh = findHeight (node → left);
    int rh = findHeight (node → right);

    return 1 + max (lh, rh);
}
```

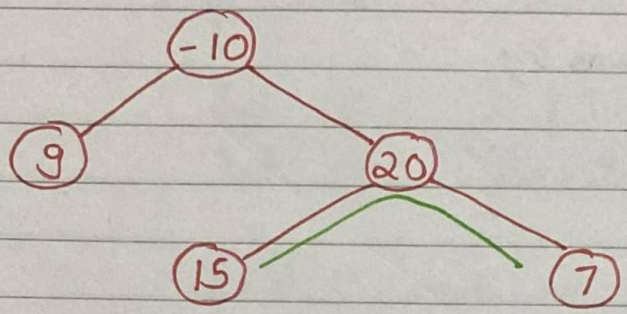
```
void findMax(TreeNode *node, int &ans){
    if(node == NULL) return;

    int lh = findHeight(node->left);
    int rh = findHeight(node->right);

    ans = max(ans, lh + rh);

    findMax(node->left, ans);
    findMax(node->right, ans);
}

int diameterOfBinaryTree(TreeNode *root){
    int ans = 0;
    findMax(root, ans);
    return ans;
}
```

⊙ Better Approach → O(N)

```
int findMax(TreeNode *node, int &ans){
    if(node == NULL) return 0;

    int lh = findMax(node->left, ans);
    int rh = findMax(node->right, ans);

    ans = max(ans, lh + rh);

    return 1 + max(lh, rh);
}
```

```
int solution (TreeNode *root) {
    int ans = 0;
    findMax (root, ans);
    return ans;
}
```

→ (L17) - Maximum Path Sum in Binary Tree



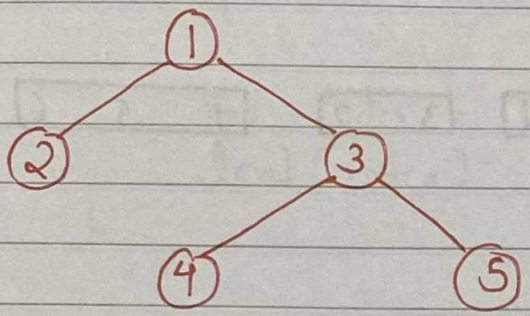
42

CODE :-

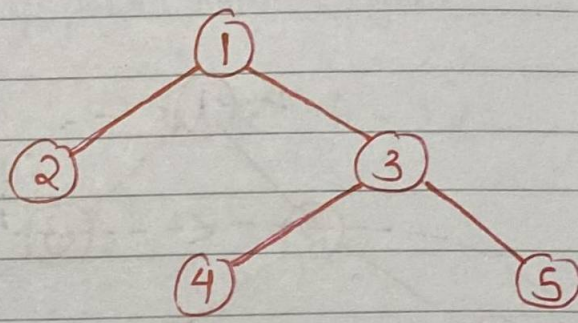
```
int Joobar (TreeNode *node, int &ans) {
        if (node == NULL) return 0;

        // Avoid negative as they won't help maximize
        int left = max(0, Joobar(node -> left, ans));
        int right = max(0, Joobar(node -> right, ans));

        ans = max(ans, left + right + node -> val);

        return max(left, right) + node -> val;
}
```

Time Complexity → O(N)

→ L18 - Check if two trees are identical or Not



Tree I



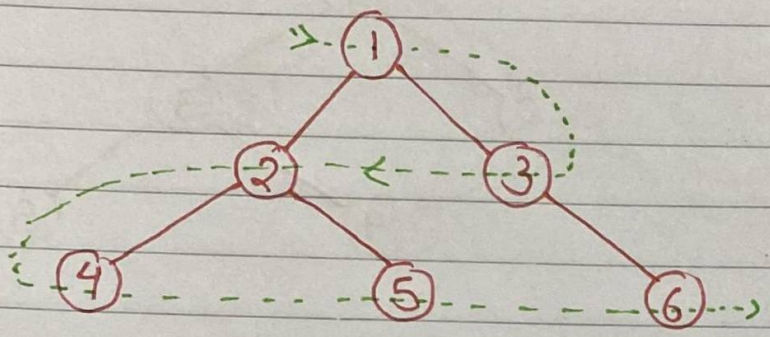
Tree II

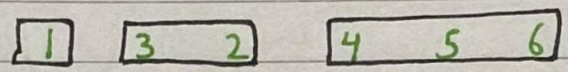
CODE :-

```
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);
}
```

* Its just a Pre-order Traversal (Time Complexity - O(N))

→ (L19) - Zig-Zag or Spiral Traversal in Binary Tree





* Can be done using slight variation in level order traversal

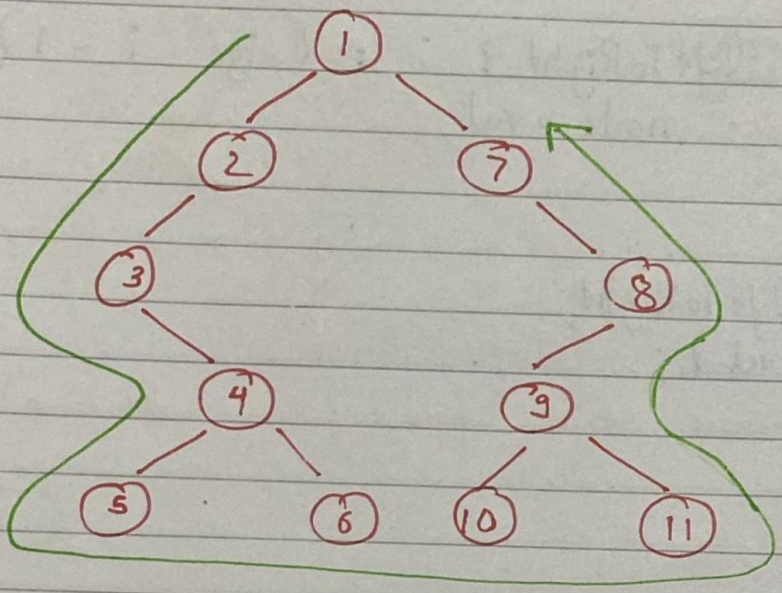
CODE :-

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root){
        vector<vector<int> ans;
        if(root == NULL)
            return ans;

        queue<TreeNode*> q;
        q.push(root);

        bool leftToRight = true;

        while(!q.empty()){
            int size = q.size()
            vector<int> level(size);

            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);

            int inden = leftToRight ? i : (size - i - 1);
            level[inden] = node->val;
        }

        leftToRight = !leftToRight;
        ans.push_back(level);
    }
    return ans;
}
```

TC → O(N)
SC → O(N)

→ L20 - Boundary Traversal in Binary Tree



1 2 3 4 5 6 10 11
9 8 7

First → Left Boundary excluding leaf.

→ Leaf Nodes

→ Right Boundary on the reverse end

TC → O(H) + O(H) + O(N) ≃ O(N)

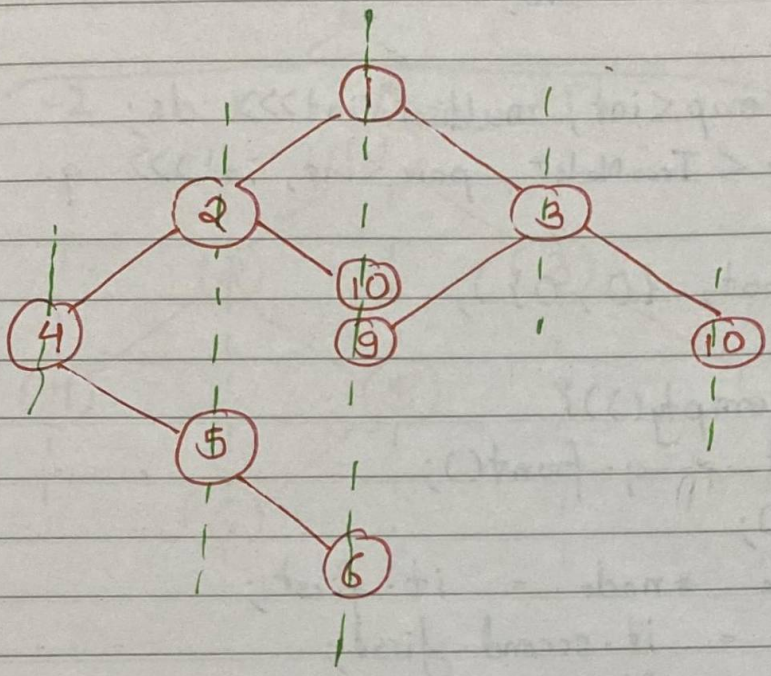
SC → O(N)

CODE:-

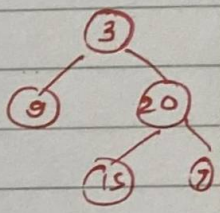
```
bool isLeaf ( Node *node ) {
        return ( !node -> left && !node -> right );
}

void leftBoundary ( Node *node, vector<int> &ans ) {
        while (node) {
                if ( !isLeaf (node)) ans.push_back ( node -> data );
                if (node -> left) node = node -> left;
                else node = node -> right;
        }
}

void leafTraversal (Node *node, vector<int> &ans ) {
        if ( isLeaf (node)) {
                ans.push_back( node -> data );
                return;
        }

if(node->left) leafTraversal ( node -> left, ans );
if(node ->right) leafTraversal ( node -> right, ans);
}

void rightBoundary ( Node *node, vector<int> &ans ) {
        vector <int> tmp;
        while (node) {
                if( !isleaf (node)) ans.push_back( node -> data );
                if( node -> right) node = node -> right;
                else node = node -> left;
        }
```

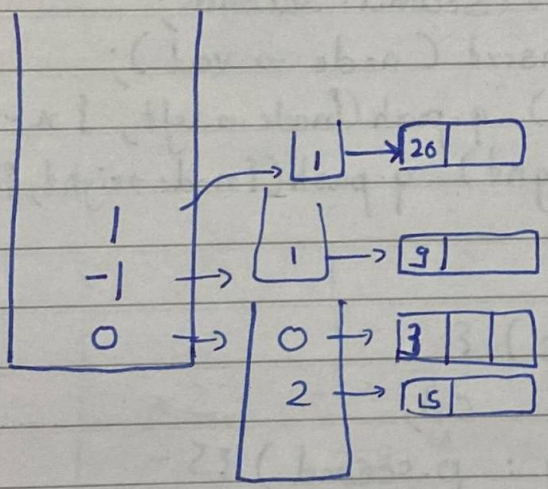
```
    for(int i = tmp.size() - 1;  i >= 0;  i--)
        ans.push_back(tmp[i]);
}

vector<int> boundary(Node *root){
    vector<int> ans;
    if(!root) return ans;
    if(!isLeaf(root)) ans.push_back(root->data);
    leftBoundary(root->left, ans);
    leafTraversal(root, ans);
    rightBoundary(root->right, ans);
    return ans;
}
```

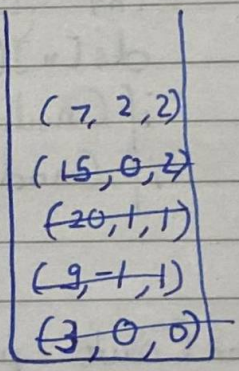
→ L21 - Vertical Order Traversal



O/P →
4
2 5
1 10 9 6
3
10





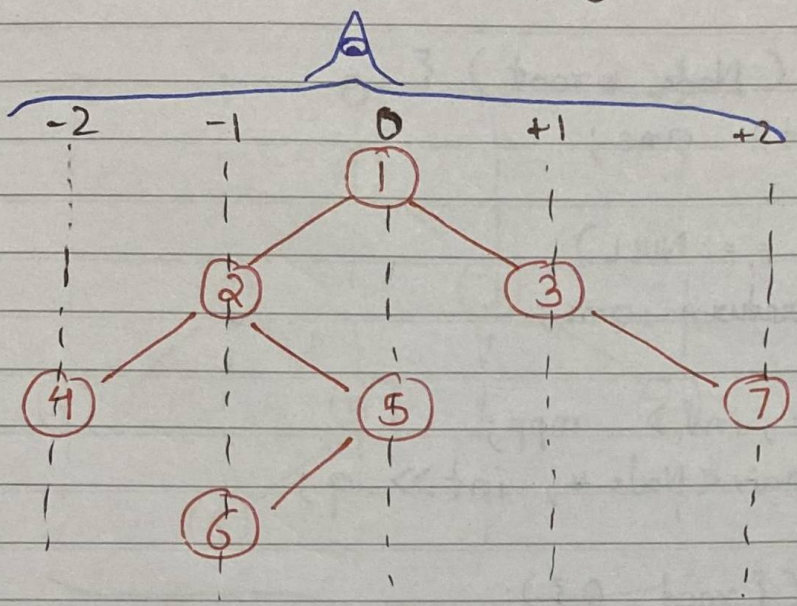


P.T.O →

CODE :-

```
vector<vector<int>> verticalTraversal(TreeNode *root) {
    vector<vector<int>> ans;

    map<int, map<int, multiset<int>>> ds;
    queue<pair<TreeNode*, pair<int, int>>> q;

    q.push(root, {0, 0});

    while (!q.empty()){
        auto it = q.front();
        q.pop();
        TreeNode *node = it.first;
        int x = it.second.first;
        int y = it.second.second;
        ds[x][y].insert(node->val);
        if(node->left) q.push({node->left, {x-1, y+1}});
        if(node->right) q.push({node->right, {x+1, y+1}});
    }

    for(auto p : ds) {
        vector<int> col;
        for(auto q : p.second) {
            col.insert(col.end(), q.second.begin(),
                       q.second.end());
        }
        ans.push_back(col);
    }

    return ans;
}
```

→ (L22) - Top View of Binary Tree



O/P → 4 2 1 3 7

Algo

+2	→ 7
-2	→ 4
+1	→ 3
-1	→ 2
0	→ 1

~~(7, +2)~~
~~(5, 0)~~
~~(4, -2)~~
~~(3, +1)~~
~~(2, -1)~~
~~(1, 0)~~

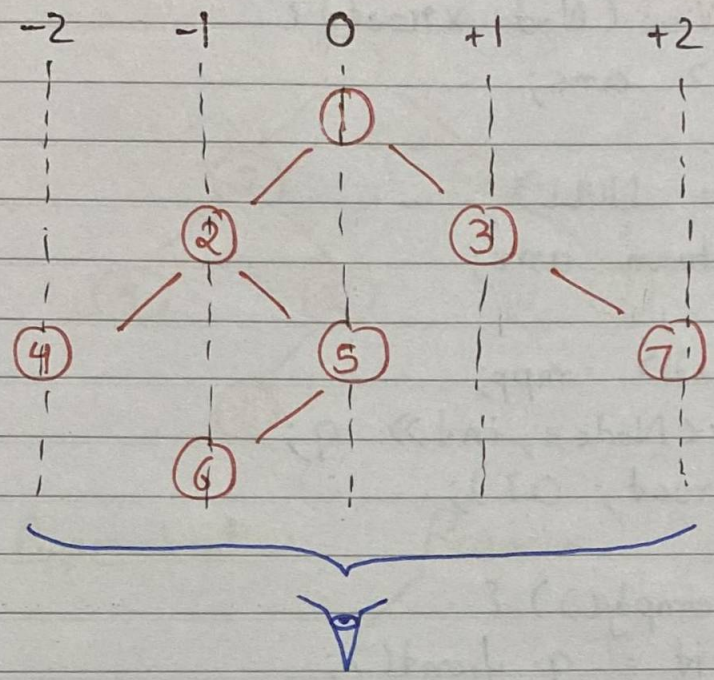
queue.

P.T.O →

CODE :-

```
vector<int> topView ( 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 level = it.second;

        if (mpp.find(level) == mpp.end())
            mpp[level] = node -> data;

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

    for (auto it : mpp)
        ans.push_back (it.second);

    return ans;
}
```

→ L23 - Bottom View of Binary Tree



O/P → 4 6 5 3 7

Algo → Same as the previous problem

BUT

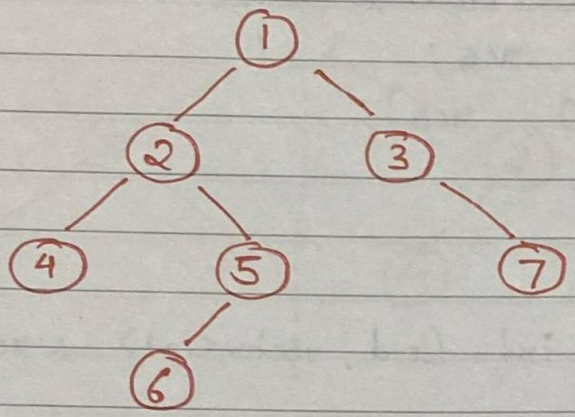
We just need to update every time even if it exists.

P.T.O →

CODE :-

```
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 level = it.second;

        mpp[level] = node->val;
        if(node->left) q.push({node->left, level-1});
        if(node->right) q.push({node->right, level+1});
    }

    for(auto p : mpp)
        ans.push_back(p.second);

    return ans;
}
```

→ L24 - Right / Left View of Binary Tree



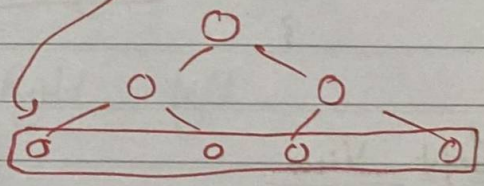
Approaches : Recursive | Iterative

TC → O(N)
SC → O(H)

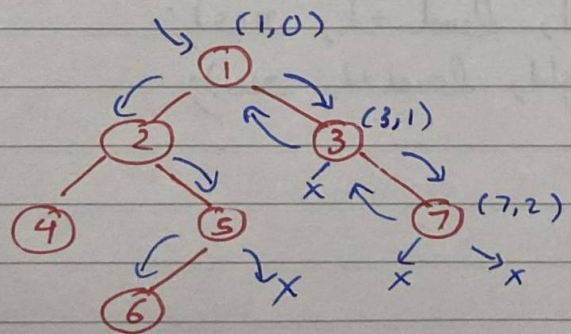
Code is simple

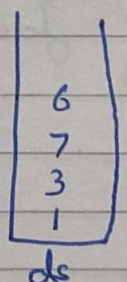
Iterative → Level Order

TC → O(N)
SC



Right Logic :-





CODE:

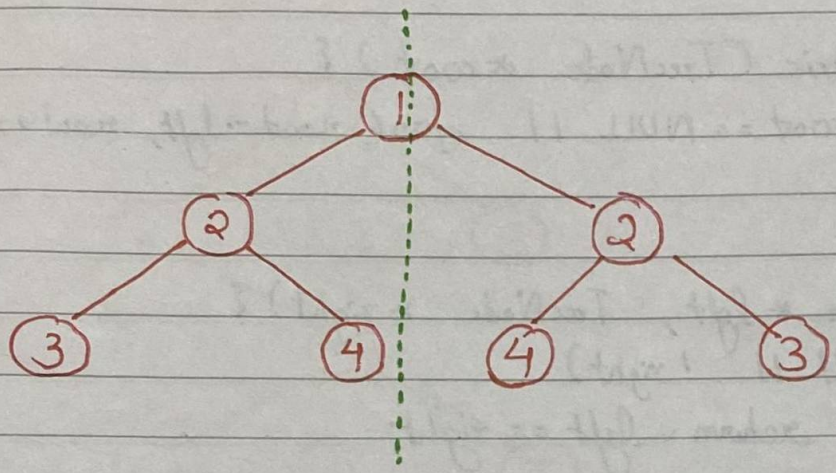
```
vector<int> rightSideView (TreeNode *root) {
        vector <int> res;
        foo(root, 0, res);
        return res;
}

void foo (TreeNode *node, int level, vector<int> &res){
        if(node == NULL)
                return;

        if( level == res.size())
                res.push_back(node->val);

        foo(node->right, level+1, res);
        foo(node->left, level+1, res);
}
```

Left View

↘ Same as above just interchange this

```
foo(node->left, level+1, res);
foo(node->right, level+1, res);
```

→ (L25) - Check for Symmetrical Binary Trees.



Approach :

Left ⇄ Right

(Root → Left)

Root Left Right

(Root → Right)

Root Right Left.

P.T.O →

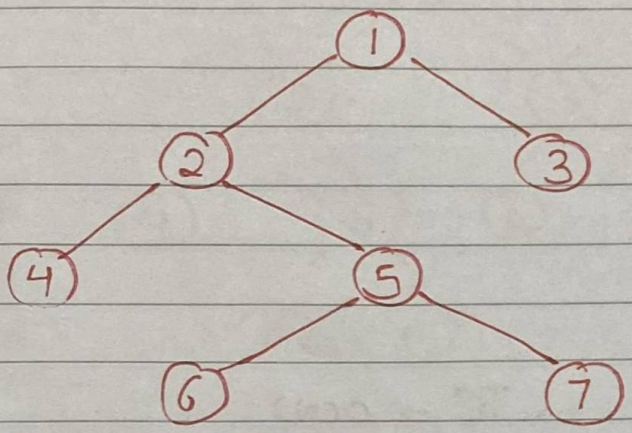
CODE :-

```
bool isSymmetric (TreeNode *root) {
    return root == NULL || foo(root->left, root->right);
}

bool foo(TreeNode *left, TreeNode *right) {
    if (!left || !right)
        return left == right;

    if (left.val != right.val) return false;

    return foo(left.left, right.right) &&
           foo(left.right, right.left);
}
```

TC → O(N)

SC → O(N)

→ L26 - Print Root to Node Path



Node => 7

* We use in order Traversal here because its easy to explain and also the code is simple.

CODE:-

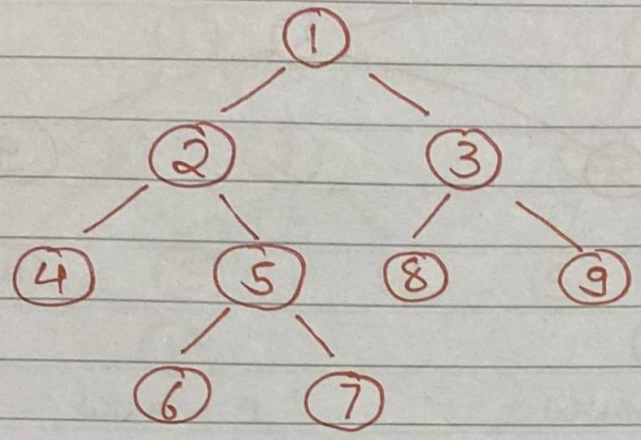
```
bool getPath(TreeNode *node, vector<int> &arr, int x) {
    if(!node)
        return false;

    arr.push_back(node->val);

    if(node->val == x)
        return true;

    if(getPath(node->left, arr, x) || getPath(node->right,
        arr, x))
        return true;

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

```
vector<int> solve ( TreeNode * A , int B ) {
        vector<int> arr;
        if ( !A )
            return arr;
        gdPath ( A , arr, B );
        return arr;
}
```

TC → O(N)

SC → O(H)

→ L27 - Lowest Common Ancestor (Binary Tree)



lca(4, 7) = 2

Solution 1

node = 4 Path → 1 2 4

TC → O(N)
SC → O(N)

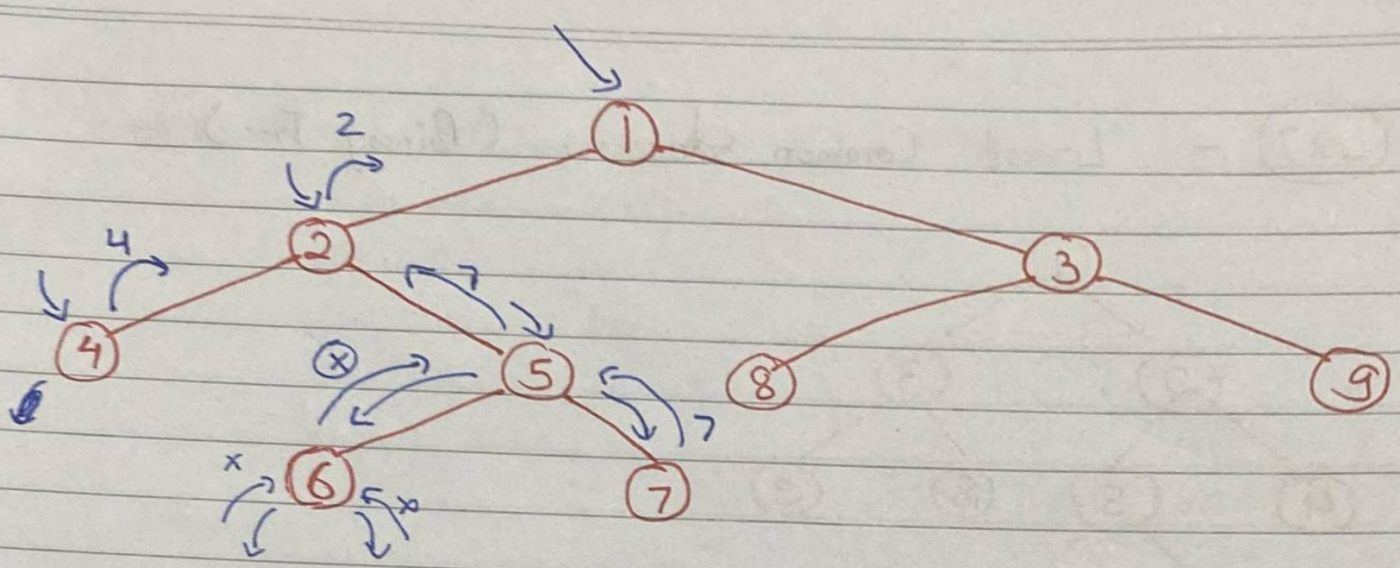
node = 7 Path → 1 2 5 7

TC → O(N)
SC → O(N)

★ Check for unequal values in this path and the answer will be the value before the unequal's occurence. i.e (2) in this case.

Extra Space is being used so we have to

P.T.O →

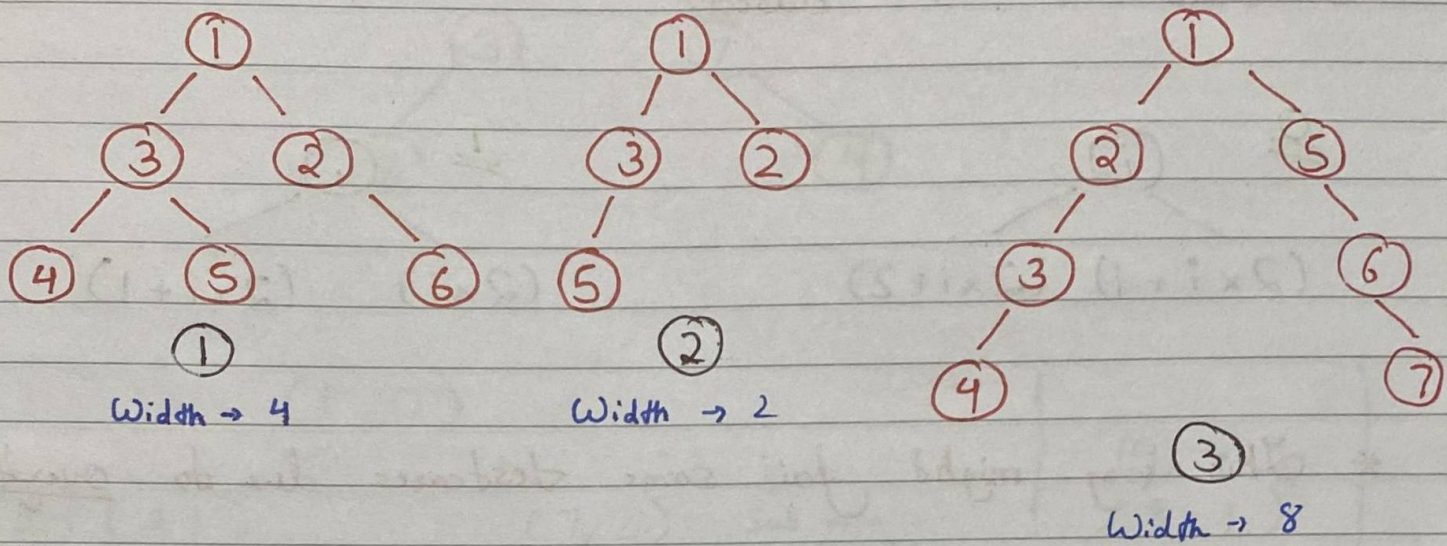


CODE:

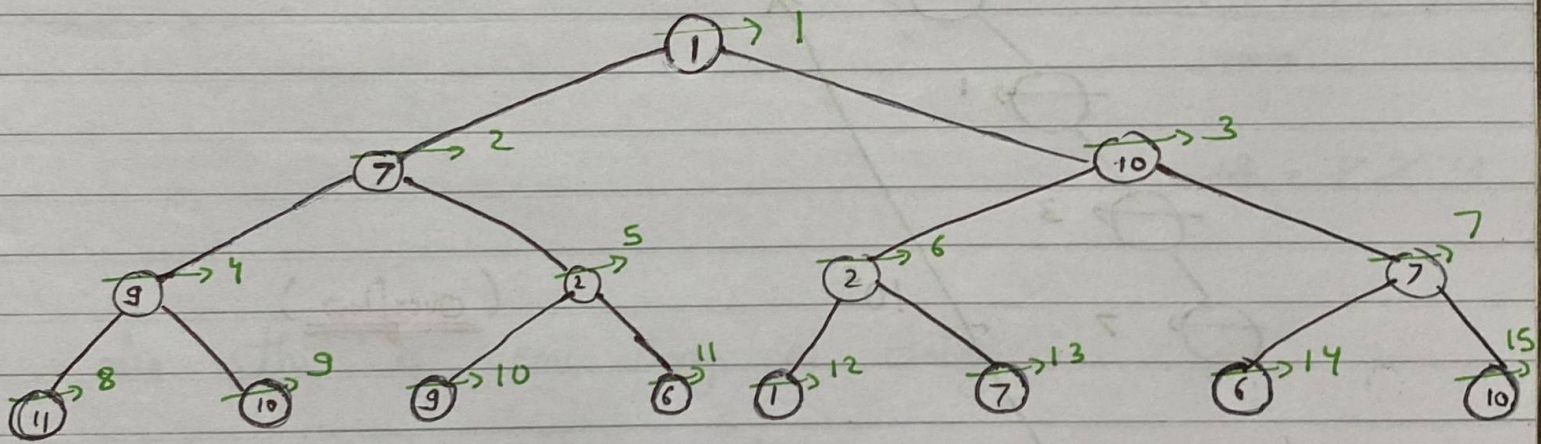
```
TreeNode*  lcs (TreeNode* root, TreeNode* p , TreeNode *q){
    if( root == NULL || root == p || root == q)
            return root;

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

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

TC → O(N)
SC → O(N)

→ L28 - Maximum Width of Binary Tree



Solution → Use level order traversal.



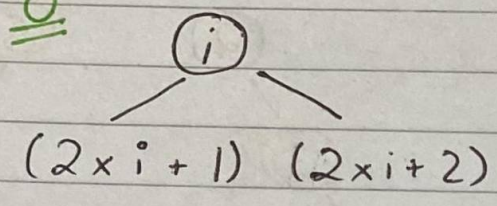
Now to find the width we just check for the max. in each level.

(High Idx - Low Idx + 1)

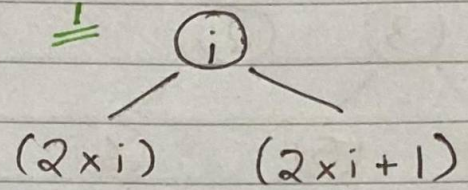
i.e 15 - 8 + 1 = 8

However this indexing can be done in two ways:-
0 based | 1 based

0

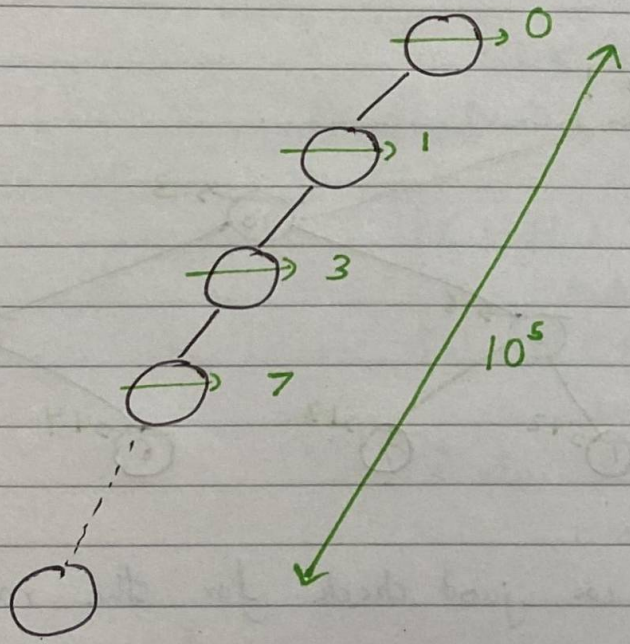


1



* This way might fail some testcases due to overflow.

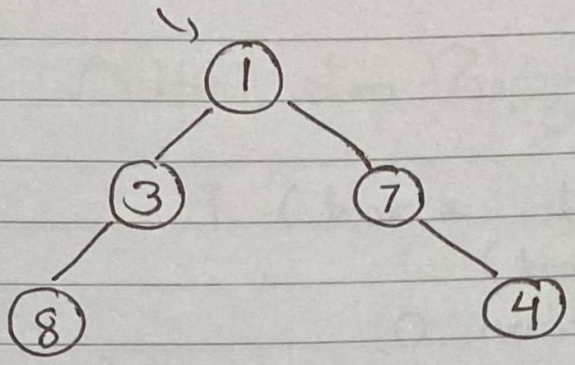
For eg: A skew tree

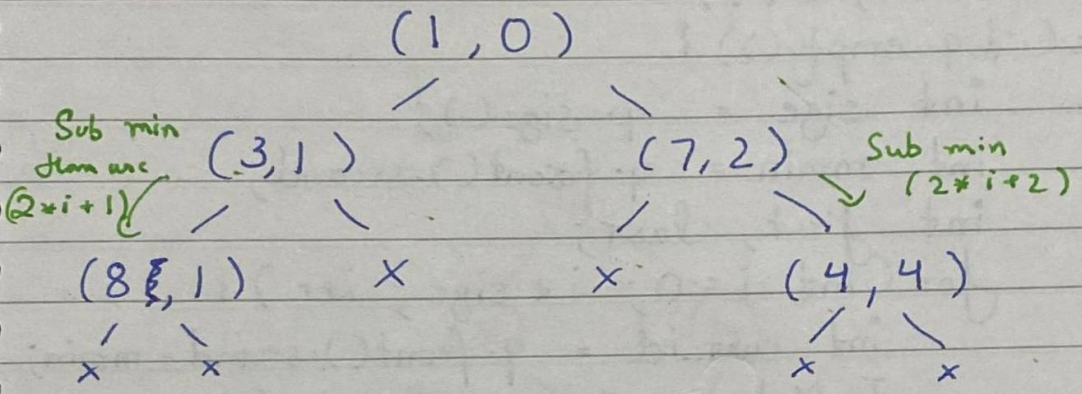


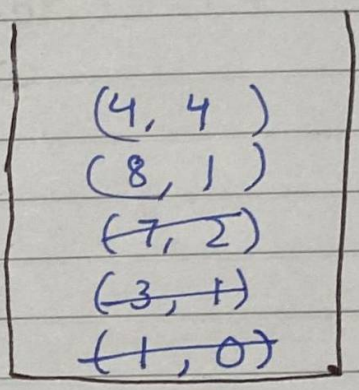
(overflow)

So to avoid this we use a different indexing method which 0 in each level.

ALGO
For eg:







queue.

width = 1 2 4

So this is one way of doing it.

Now lets see the code:

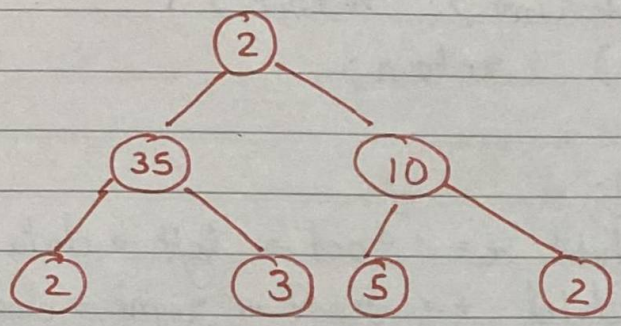
PTO
CODE →

CODE :-

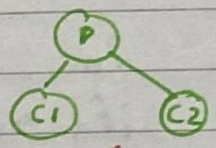
```
int foo ( TreeNode * root ) {
    if ( ! root )
        return 0;
    int ans = 1;
    queue < pair < TreeNode *, int >> q;
    q.push ( { root, 0 } );
    while ( ! q.empty() ) {
        int size = q.size();
        int mmin = q.front().second;
        int first, last;
        for ( int i = 0; i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

TC → O(N)
SC → O(N)

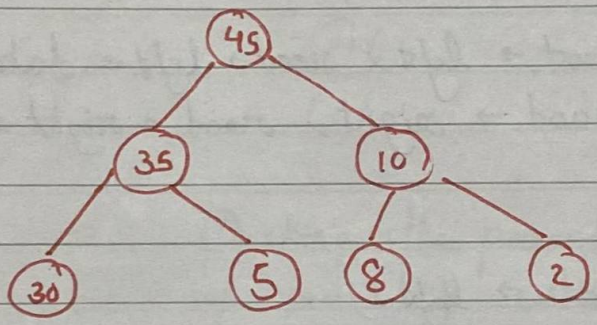
→ L29 - Children Sum Property in Binary Tree



(↑) by +1 any times



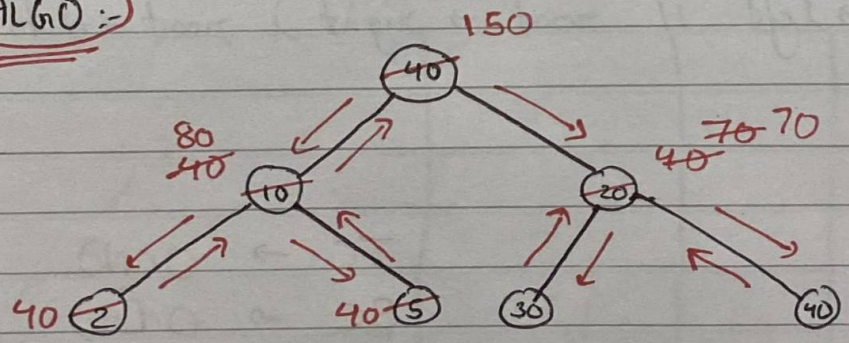
Such that (P = C1 + C2)



Solution → GFG → $O(N^2)$

Striver → $O(N)$

ALGO :-



10 + 20 = 30 < 40

2 + 5 = 7 < 40

30 + 40 = 70 >= 40

P.T.O →

CODE :-

```
void changeTree ( BinaryTreeNode<int> *root ) {
    if ( root == NULL ) return;

    int child = 0;
    if ( root -> left ) child += root -> left -> data;
    if ( root -> right ) child += root -> right -> data;

    if ( child >= root -> data ) root -> data = child;
    else {
        if ( root -> left ) root -> left -> data = root -> data;
        if ( root -> right ) root -> right -> data = root -> data;
    }

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

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

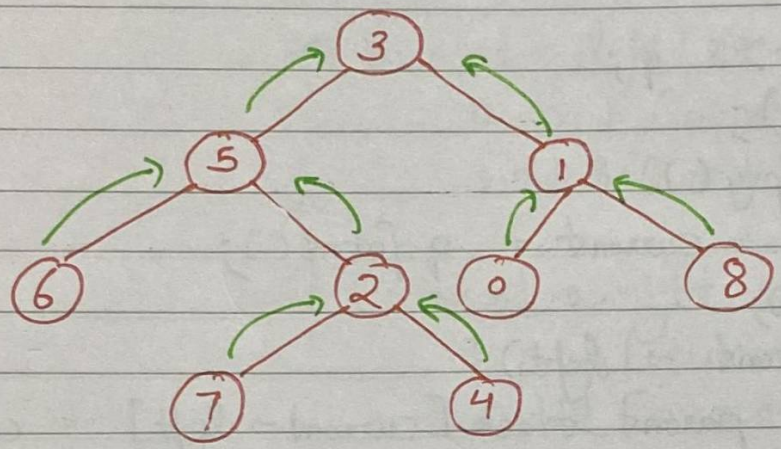
TC → O(N)

SC → O(H)

H ≤ N (worst)

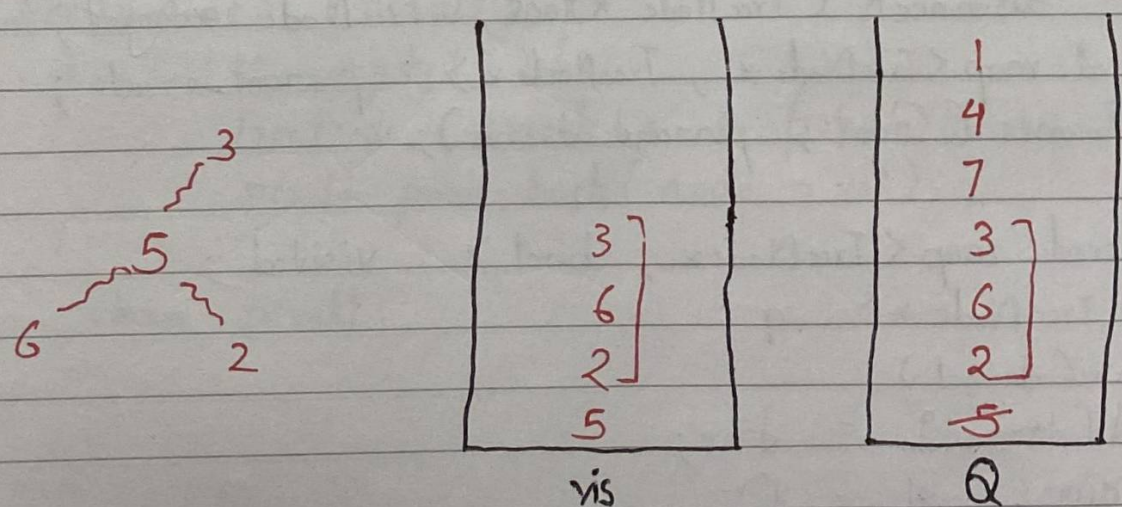
→ (L30) - Print all the Nodes at a distance of K in Binary Tree

For eg: K = 2, target = 5



STEP 1 → Marking the parent pointers so we can move upward as well.

STEP 2 → Radially move from target node till dist = k. Also keep a vis hash.



CODE:-

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

vector<int>  distanceK ( TreeNode *root, TreeNode *target, int k)
      unordered_map <TreeNode * , TreeNode *>  parent_track ;
      markParents ( root , parent_track );

      unordered_map < TreeNode* , bool >  visited ;
      queue <TreeNode *>  q ;
      q.push ( target );
      visited [target] .= true;
      int curr_level = 0;
```

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

vector<int> result;
while ( !q.empty()) {
        TreeNode *node = q.front(); q.pop();
        result.push_back(node->val);
}
return result;
}
```

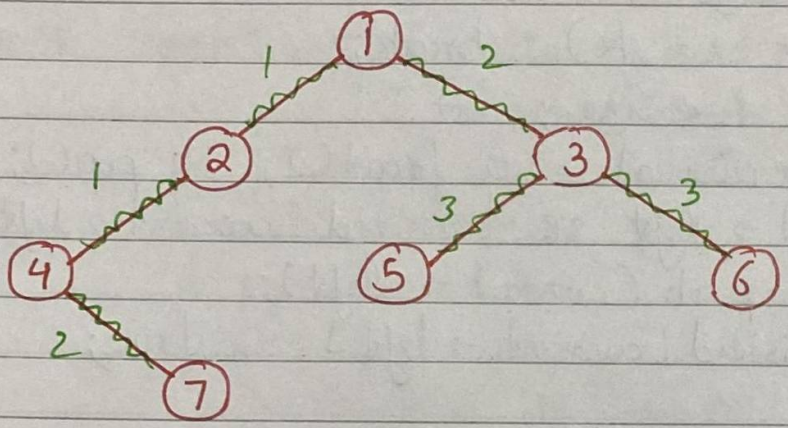
+hash

TC → O(N) + O(N)

SC → O(N) + O(N) + O(N)

≈ O(N)

→ (L31) - Min time taken to BURN a BT. from a node.



| Node → 2 |

⇒ 3 will be the ans.

ALGO :-

STEP 1 → Mark the parents for each node and store it in parent_track hash.

STEP 2 → Radially move till the tree is burnt entirely. (i.e BFS)

★ Very similar to the L30 problem.

CODE →

CODE :-

```
BinaryTreeNode<int>* bfsToMapParent( BinaryTreeNode<int>* root,
    unordered_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) {
                    mpp[node->left] = node;
                    q.push(node->left);
            }
            if(node->right) {
                    mpp[node->right] = node;
                    q.push(node->right);
            }
        }

        return res;
}
```

P.T.O →

```
int findMaxDistance(unordered_map<BinaryTreeNode<int>*, BinaryTreeNode
<int>*> &mpp, BinaryTreeNode<int>* target) {
queue<BinaryTreeNode<int>*> q;
q.push(target);
unordered_map<BinaryTreeNode<int>*, int> vis;
vis[target] = 1;
int result = 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] = 1;
            q.push(node->left);
        }
        if(node->right && !vis[node->right]) {
            fl = 1;
            vis[node->right] = 1;
            q.push(node->right);
        }
        if(mpp[node] && !vis[mpp[node]]) {
            fl = 1;
            vis[mpp[node]] = 1;
            q.push(mpp[node]);
        }
    }
    if(fl) result++;
}
```

```
    return result;
}

int timeToBurnTree(BinaryTreeNode<int>* root, int start) {
    unordered_map<BinaryTreeNode<int>*, BinaryTreeNode<int>*> mpp;
    BinaryTreeNode<int>* target = bfsToMapParents(root, mpp, start);
    return findMaxDistance(mpp, target);
}
```

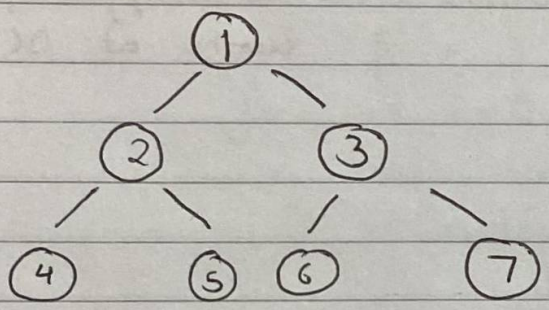
TC → O(N) + O(N) ≃ O(N) { assuming map works at O(1) }

SC → O(N)

→ (L32) Count total Nodes in a Complete Binary Tree

SOLUTION 1 → Using any traversal and counting all the Node.
However this will take O(N) TC and we need to provide a solution with better TC.

SOLUTION 2 →

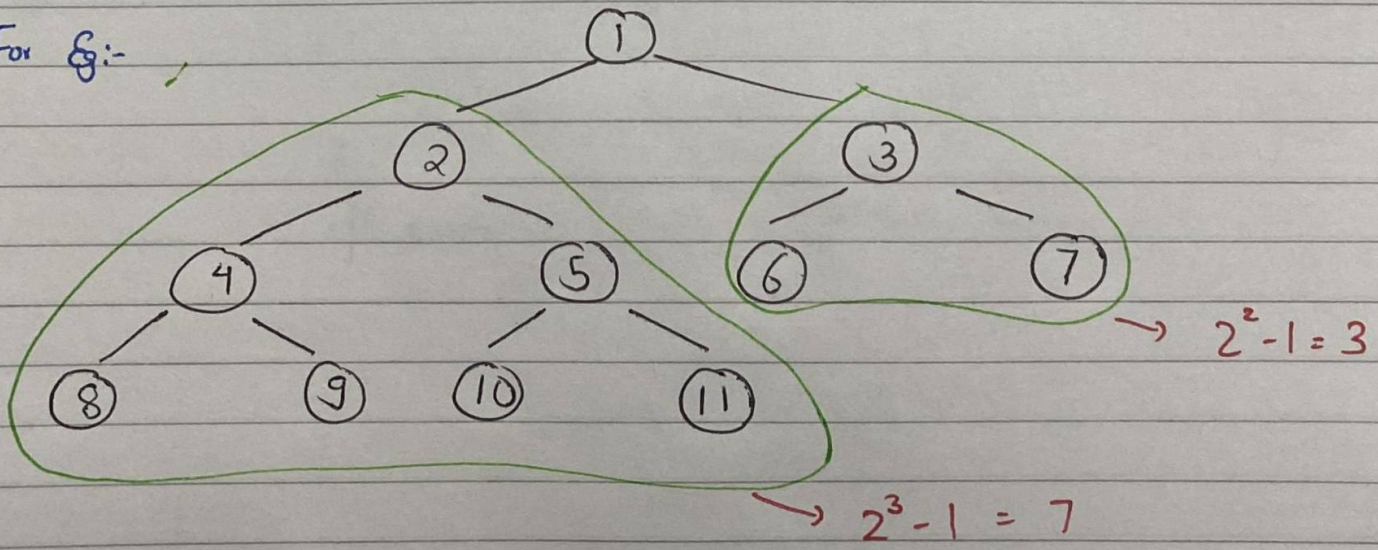


If we have a fully filled Complete B.T.
Then, No. of Nodes = $2^h - 1$
where h = height of the tree

$\therefore N = 2^3 - 1 = 7$

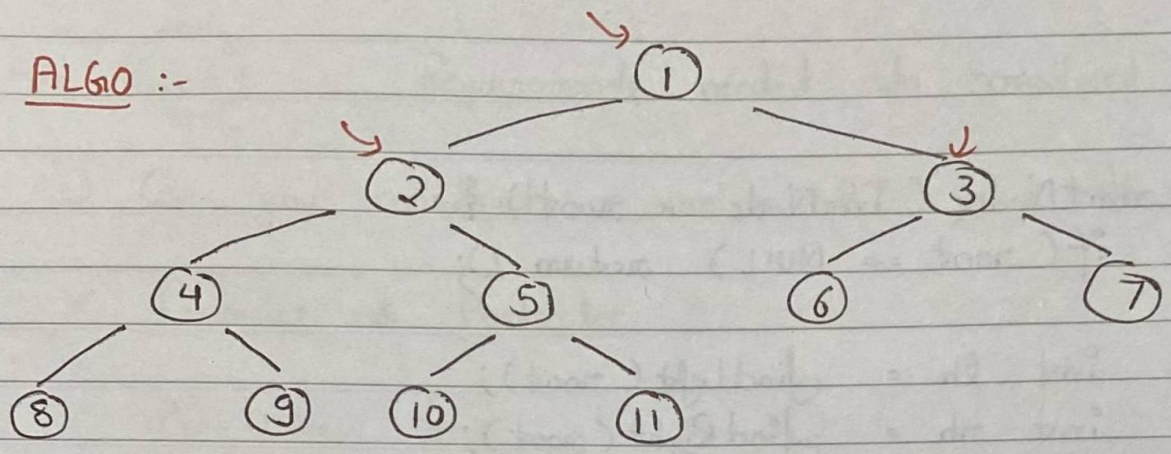
Now we can use this property smartly to find our answer.

For Eg:-



$\Rightarrow 1 + 3 + 7 = 11$

ALGO :-



(1) When on Node 1 → lh = 4
rh = 3
=> this subtree is not fully filled.

(2) When on Node 2 → lh = 3
rh = 3
=> this is a fully filled tree so return $(2^3 - 1) = 7$

(3) When on Node 3 → lh = 2
rh = 2
=> return $2^2 - 1 = 3$

(4) ∴ return 1 + 7 + 3

P.T.O →

★ NOTE: (a << b) means

$a * 2^b$

∴ $1 << lh \Rightarrow 1 * 2^{lh}$

CODE :-

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

    int lh = findLeft ( root );
    int rh = findRight ( root );

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

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

int findLeft ( TreeNode * node ) {
    int result = 0;
    while ( node ) {
        result ++;
        node = node -> left;
    }
    return result;
}

int findRight ( TreeNode * node ) {
    int result = 0;
    while ( node ) {
        result ++;
        node = node -> right;
    }
    return result;
}
```

TC → $O((\log N)^2)$

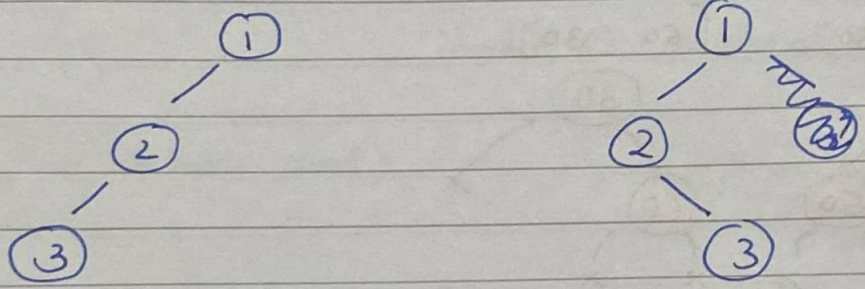
SC → $O(\log N)$

→ L33 - Requirements needed to construct a BT. (unique)

① Can you construct a unique BT from the following :-

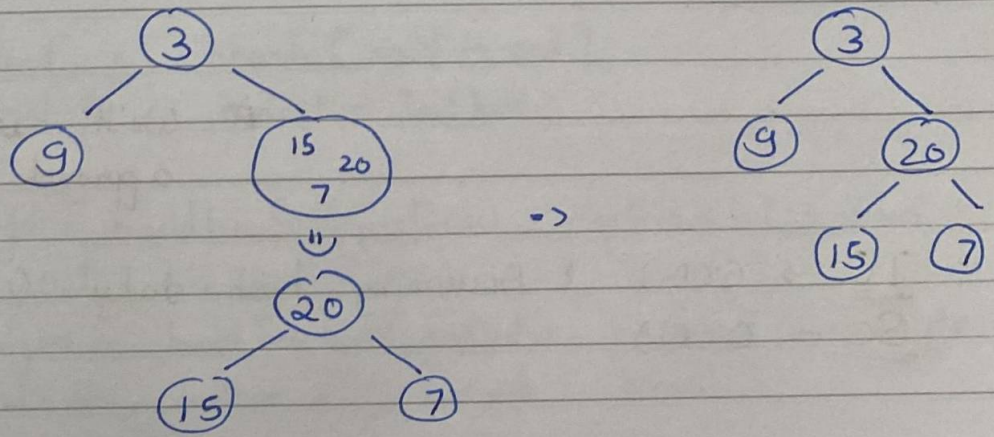
✗ Preorder & Postorder

Pre - Order → 1 2 3
Post - Order → 3 2 1



✓ InOrder & PreOrder

Pre Order → 3 9 20 15 7
In Order → 9 3 15 20 7



∴ We definitely need InOrder

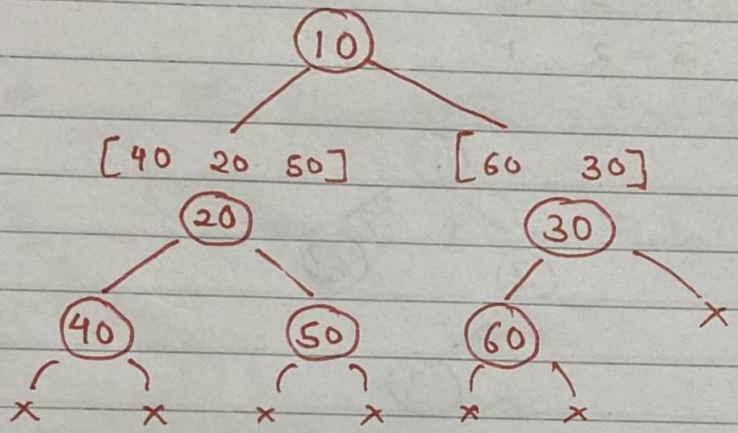
→ (L34) - Construct a BT from Pre and In order

(Left Root Right)

in → [40 20 50 10 60 30]

pre → [10 20 40 50 30 60]

(Root Left Right)



ALGO

in → [9 (3) 15 20 7]

pre → [3 9 20 15 7]

↓ root

☆ We'll use a recursive approach.

TC → O(N) { Assuming hash takes O(1) }

SC → O(N)

CODE:-

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

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

        TreeNode *root = buildTreefun(preorder, 0, preorder.size()-1,
                inorder, 0, inorder.size() -1 , inhash);

        return root;
}

TreeNode* buildTreefun(vector<int> &preorder, int preStart, int preEnd,
   vector<int> &inorder, int inStart, int inEnd, unordered_map
   <int, int> &inhash ) {
        if( preStart > preEnd || inStart > inEnd ) return NULL;

        TreeNode *root = new TreeNode( preOrder[preStart]);

        int inRoot = inhash[root->val];
        int numsLeft = inRoot - inStart;

        root->left = buildTreefun( preOrder, preStart+1, preStart +
   numsLeft, inOrder, inStart, inRoot -1 , inhash);
        root->right = buildTreefun(preOrder, preStart + numsLeft +1,
        preEnd, inOrder, inRoot +1, inEnd, inhash);

        return root;
}
```

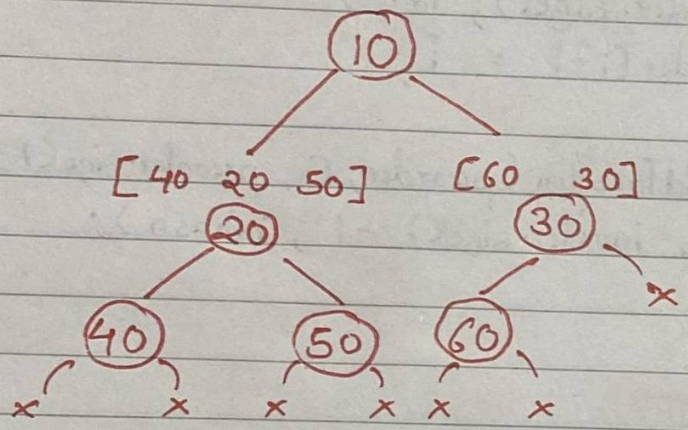
→ (L35) - Construct a BT from Post and In order

(Left Root Right)

in → [40 20 50 10 60 30]

post → [40 50 20 60 30 10]

(Left Right Root)



ALGO

in → [40 20 50 (10) 60 30]

post → [40 50 20 60 30 10]

As post order is Left Right Root. (10) will be the root of tree.

TC → O(N) {Assuming hash takes O(1)}

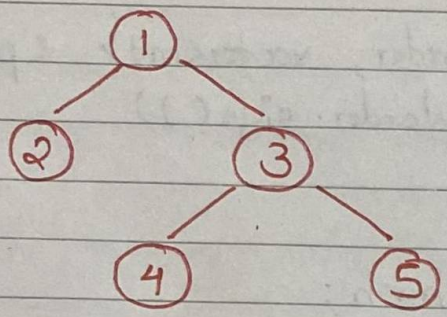
SC → O(N)

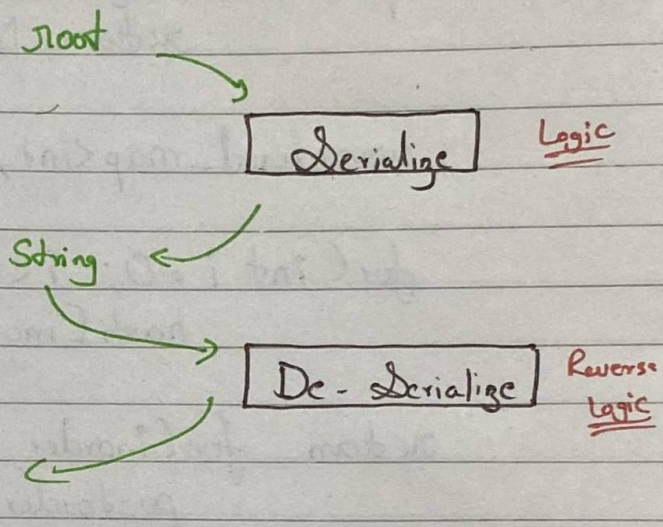
★ Again we'll use a recursive approach similar to the prev question.

CODE:-

```
TreeNode * buildTree(vector<int> &inorder, vector<int> &postorder){
        if( inorder.size() != postorder.size())
                return NULL;

        unordered_map<int,int> hash;

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

        return foo(inorder, 0, inorder.size()-1, postorder, 0,
                postorder.size()-1, hash);
}

TreeNode * foo(vector<int> &inorder, int is, int ie, vector<int>
&postorder, int ps, int pe, unordered_map<int,int> &hash){
        if( ps > pe || is > ie) return NULL;

        TreeNode *root = new TreeNode(postorder[pe]);
        int inRoot = hash[postorder[pe]];
        int numsLeft = inRoot - is;

        root->left = foo(inorder, is, inRoot-1, postorder, ps
                , ps+numsLeft-1, hash);
        root->right = foo(inorder, inRoot+1, ie, postorder
                , ps+numsLeft, pe-1, hash);

        return root;
}
```

→ (L36) - Serialize and De-Serialize a B.T.



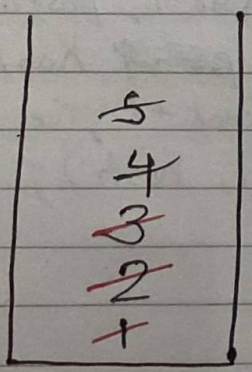


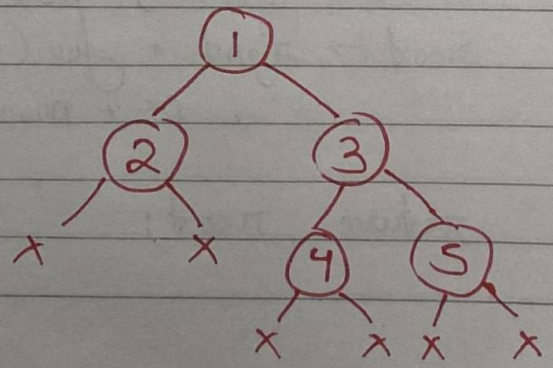
☆ We can use any traversal here we'll use LO (Level-order).

(1) Serialize

1, 2, 3, #, #, 4, 5, #, #, #, #,

(2) De-Serialize





CODE:-

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

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

            if(curNode) {
                q.push(curNode->left);
                q.push(curNode->right);
            }
        }

        return s;
}
```

PTO for De-serialize ↪

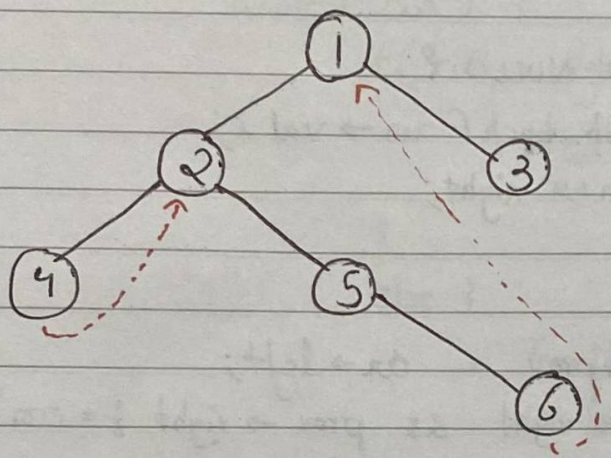
```
TreeNode* deserialize(string data) {
    if(data.size() == 0) return NULL;
    stringstream s(data);
    string str;
    getline(s, str, ',');
    TreeNode *root = new TreeNode(stoi(str));
    queue<TreeNode*> q;
    q.push(root);

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

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

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

TC → O(N)
SC → O(N)

→ (137) - Morris Traversal → O(N) & O(1)

↳ Threaded Binary Tree

① Inorder (Lgd Root Right)



In Order → 4 2 5 6 1 3

1st CASE :- left = NULL

- print (cur)
- → move right

2nd CASE :-

- left ↙ right most guy will be connected to cur.
- → move left
- If it is already pointing (i.e thread exists)
  - → remove thread
  - → move right

CODE :-

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

    while(cur) {
        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;
                cur = cur->right
                inorder.push_back(cur->val);
                cur = cur->right;
            }
        }
    }

    return inorder;
}
```

(2) PreOrder (Root Left Right)

```
vector<int> getPreorder ( TreeNode *root) {
        vector<int> preorder;
        TreeNode *cur = root;
        while ( cur ) {
            if( cur -> left == NULL ) {
                preorder. 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;
                    preorder . push_back ( cur -> val );
                    cur = cur -> left;
                } else {
                    prev -> right = NULL;
                    cur = cur -> right;
                }
            }
        }

        return preorder;
}
```

TC → O(N)
SC → O(1)

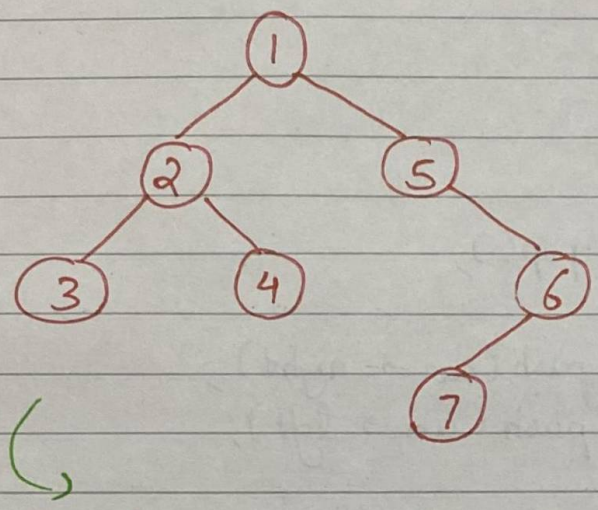
③ Post Order (Left Right Root)

```
vector<int> getPostorder (TreeNode *root) {
        vector<int> postorder;
        TreeNode *cur = root;

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

                    if( prev→left == NULL ) {
                            prev→left = cur;
                            postorder.push_back (cur→val);
                            cur = cur→right;
                    } else {
                            prev→left = NULL;
                            cur = cur→left;
                    }
            }
        }
        reverse (postorder.begin() , postorder.end());
        return postorder;
}
```

(Pre - order)

→ L38 - Flatten a Binary Tree to Linked List

Approach 1 → Recursion



1 2 3 4 5 6 7

PSEUDO CODE :-

```
prev = null
flatten (root) {
        if (root == X) return;

        flatten (root → right);
        flatten (root → left);

        root → right = prev;
        root → left = null;
        prev = root;
}
```

TC → O(N)

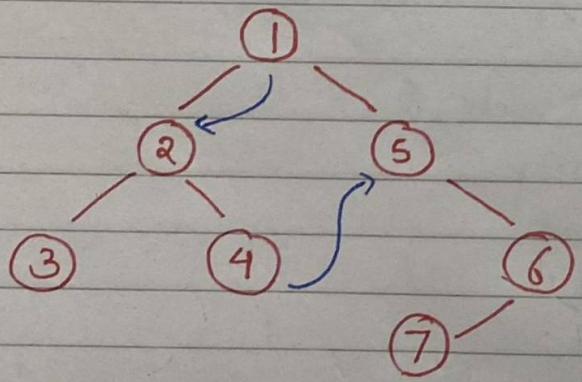
SC → O(N)

Approach 2 → Using Stack Same as 1 instead of using Recur.

PSEUDO CODE :-

```
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;
}
```

Approach 3 → Using Morris Traversal



→

PSEUDO CODE:-

TC → O(N)
SC → O(1)

```
cur = root;
while (cur) {
        if (cur → left != null) {
                prev = cur → left;
                while (prev → right)
                        prev = prev → right;

                prev → right = cur → right;
                cur → right = cur → left;
                cur → left = null;
        }
        cur = cur → right;
}
```

— X —