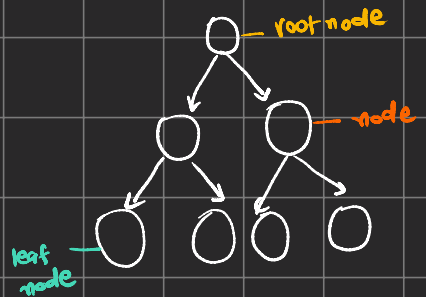
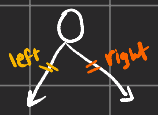
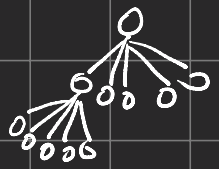
Binary Tree → non linear DS



→ combination of nodes those are attached through hierchecaly

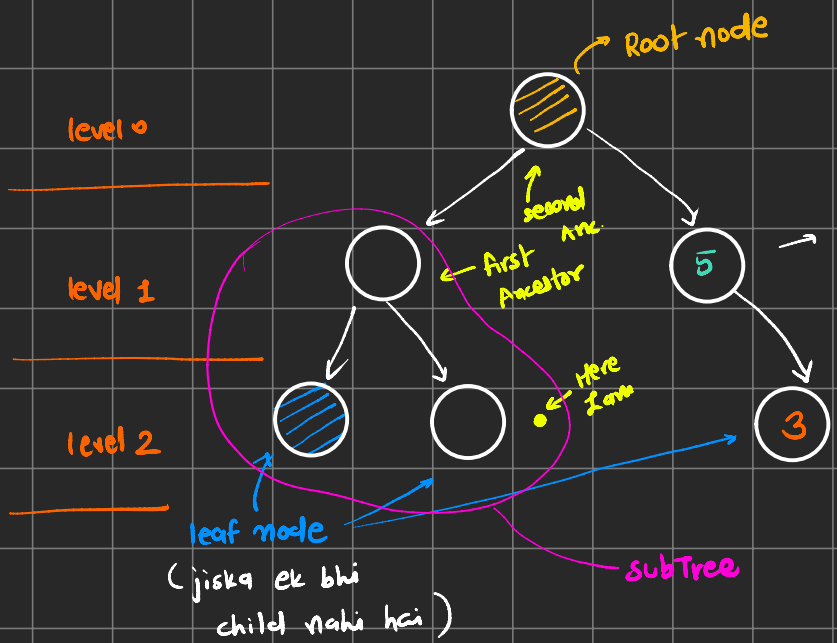



```
class Node
{
   int data ;
   Node* left ;
   Node * right ;
}
```



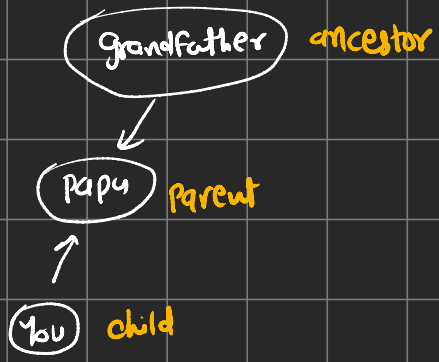
→ n-ary tree
generic tree

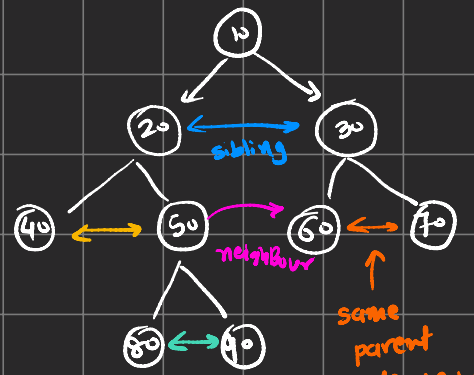
```
class Node
{
   int data ;
   vector < node * > child
}
```



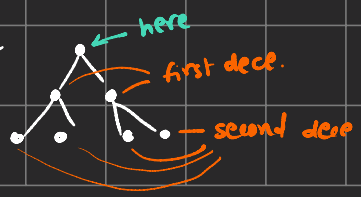
5 is parent node of 3

3 is child node of 5





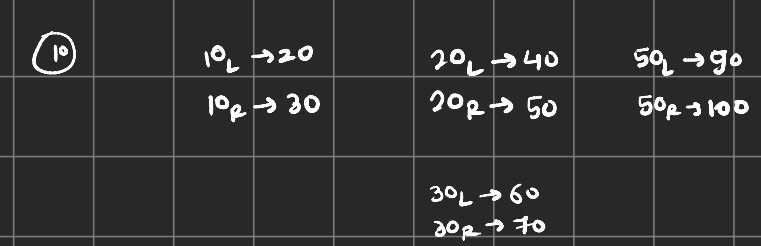
Ancestor opposite Decendent

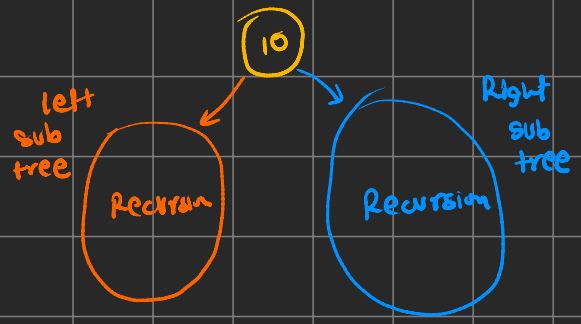


Binary Tree → Node ke 2 se jyada child nahi hai

means 0, 1, 2 childs he hai

Implementation

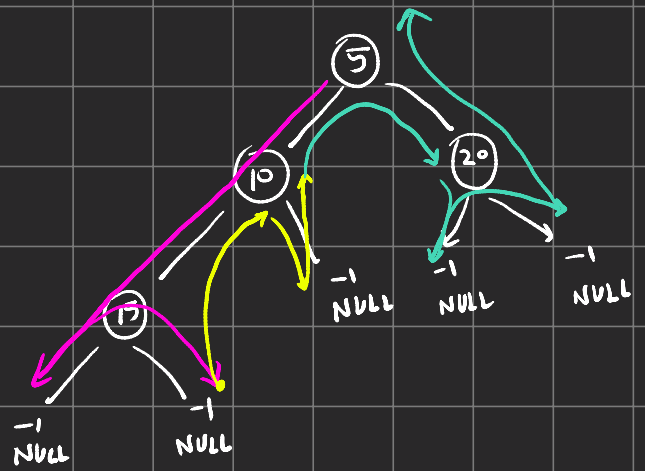





```cpp
class Node{
    public:
        int data;
        Node* left;
        Node* right;

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

// returns Root node of created Tree
Node* createTree(){
    cout << "Enter val of ROOT:" << endl;
    int data;
    cin >> data;

    if(data == -1){ // means donot want to add node
        return NULL;
    }

    // 1)create node
    Node* root = new Node(data);

    // 2) create left sub tree
    cout << "For LEFT node of" << root->data << endl;
    root->left = createTree();

    // 3) create right sub tree
    cout << "For RIGHT node of" << root->data << endl;
    root->right = createTree();

    return root;
}

int main(){
    Node* root = createTree();
    return 0;
```

```
ion } ; if ($?) { .\01_implementation }
Enter val of ROOT:
5
For LEFT node of5
Enter val of ROOT:
10
For LEFT node of10
Enter val of ROOT:
15
For LEFT node of15
Enter val of ROOT:
-1
For RIGHT node of15
Enter val of ROOT:
-1
For RIGHT node of10
Enter val of ROOT:
-1
For RIGHT node of5
Enter val of ROOT:
50
For LEFT node of50
Enter val of ROOT:
-1
For RIGHT node of50
Enter val of ROOT:
-1
```

Input : [5, 10, 15, -1, -1, -1, 50, -1, -1]



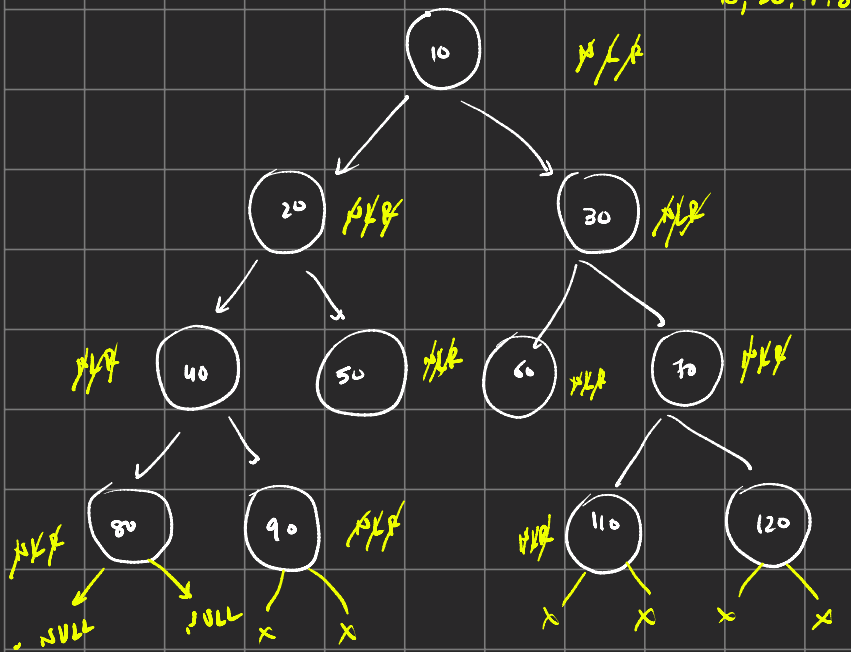
→ 3 traversal

current node, left part, right part

→ Pre order : N L R

→ In order : L N R

→ Post order : L R N



Pre order

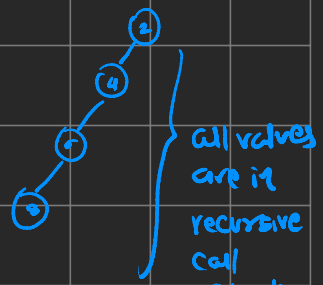
10, 20, 40, 80, 90, 50, 30, 60, 70, 110, 120

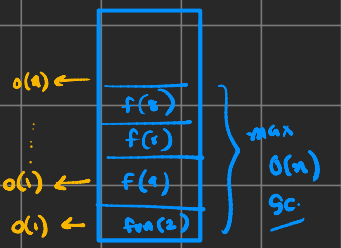
if (root = NULL)
return

TC = O(n)
SC = O(n)

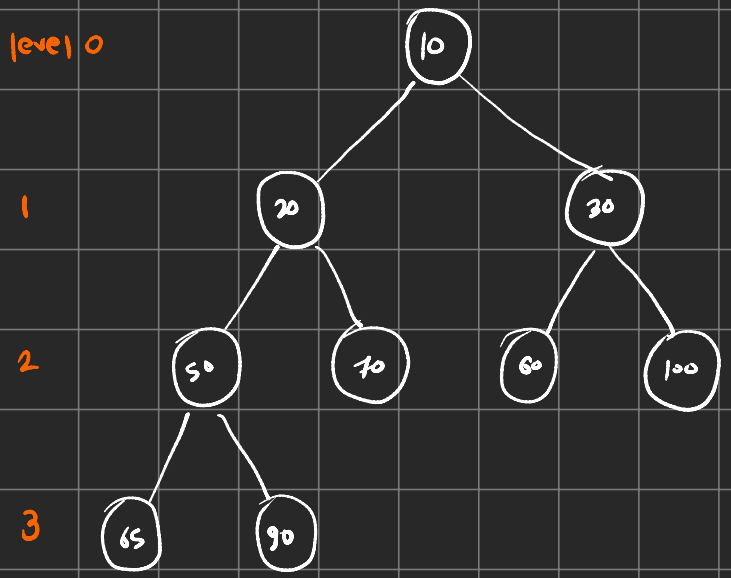
```
void preOrderTraversal(Node* root){
    if(root==NULL){
        return;
    }
    // NLR

    // N
    cout << root->data << " ";
    // L
    preOrderTraversal(root->left);
    // R
    preOrderTraversal(root->right);
}

void inOrderTraversal(Node* root){
    if(root==NULL) return;
    // LNR
    preOrderTraversal(root->left);
    cout << root->data << " ";
    preOrderTraversal(root->right);
}

void postOrderTraveral(Node* root){
    if(root==NULL) return;
    // LRN
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
    cout << root->data << " ";
}
```

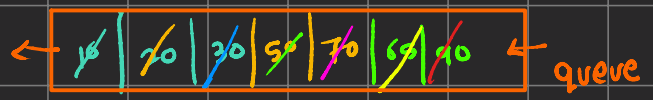
why SC = O(n)
worst case in skew tree





* Level order Traversal / Breadth First Traversal





initially push (root Node)

traversal
- fetch front (10)
- push left (10_L - 20)
- push right (10_R - 30)

(20) → $20_L = 50$, $20_R = 70$

(30) → $30_L = 60$, $30_R = 100$

(50) → $50_L = 65$, $50_R = 90$

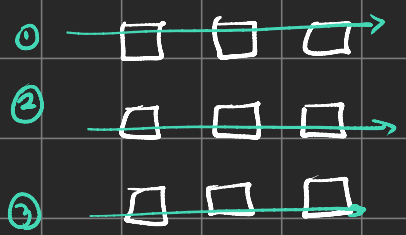
90 ← 65 ← 70

90_L = null, 90_R = null

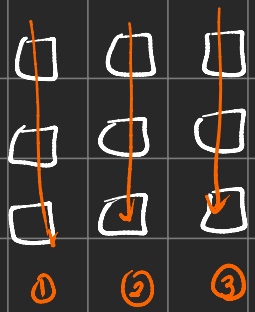
65_L = NULL, 65_R = NULL

70_L = NULL, 70_R = NULL

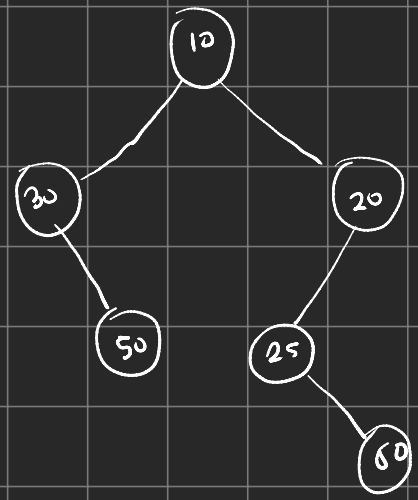
Breadth First Traversal (row wise) Search (BFS)



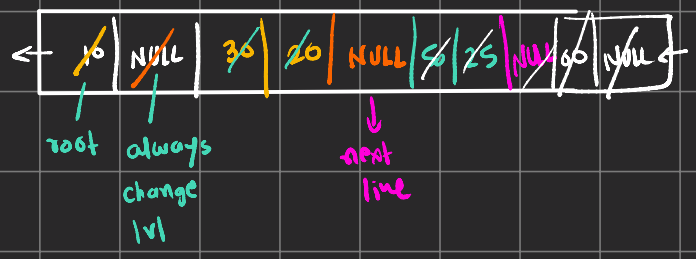
(DFS) Depth first search




```cpp
void levelOrderTraversal(Node* root){
    queue<Node*> q;
    q.push(root);

    while(!q.empty()){
        // front fetch
        Node* front = q.front();
        q.pop();
        cout << front->data << " ";

        // left
        if(front->left != NULL){
            q.push(front->left);
        }
        // right
        if(front->right != NULL){
            q.push(front->right);
        }
    }
}
```



NULL → marker → level change ho gaya



initial → q.push (root → data)
→ q.push (NULL)

Traversal → front / pop
→ NULL → print
→ push (null)
→ valid → print
→ push (left)
→ push (right)

```cpp
void levelOrderTraversal(Node* root){
    queue<Node*> q;
    q.push(root);
    q.push(NULL);

    while(q.size()>1){
        // front fetch
        Node* front = q.front();
        q.pop();

        if(front == NULL){
            cout << endl;
            q.push(NULL);
        }else{
            // valid
            cout << front->data << " ";

            // left
            if(front->left != NULL){
                q.push(front->left);
            }
            // right
            if(front->right != NULL){
                q.push(front->right);
            }
        }

    }
}
```

or

(!q.empty())

last mei NULL par fir se NULL daldega infinite loop issue

if (!q.empty()) then only q.push(NULL)

10
30 20
50 25
60

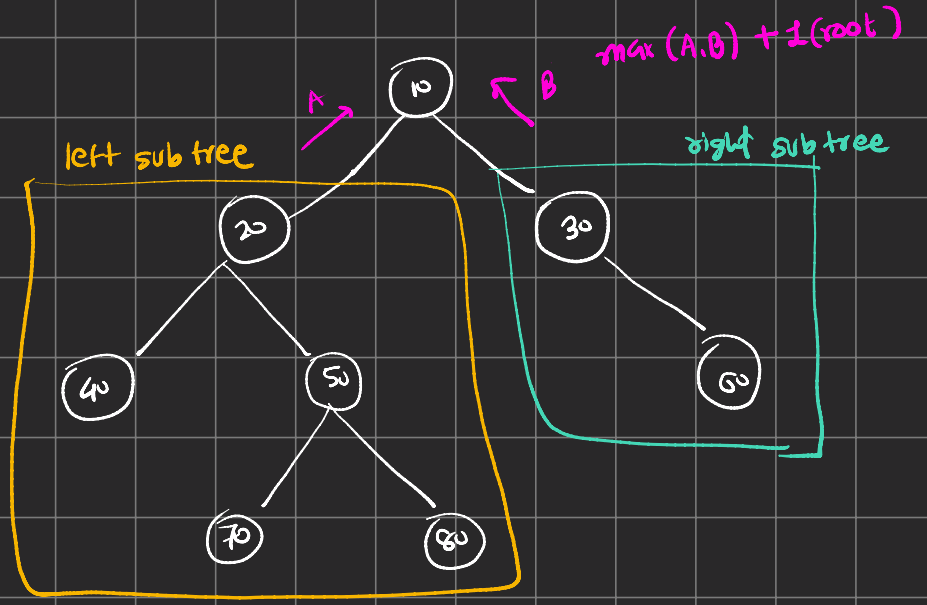
level wise printing

TC = O(n) - visiting each node

SC = max no of element in a single level

=

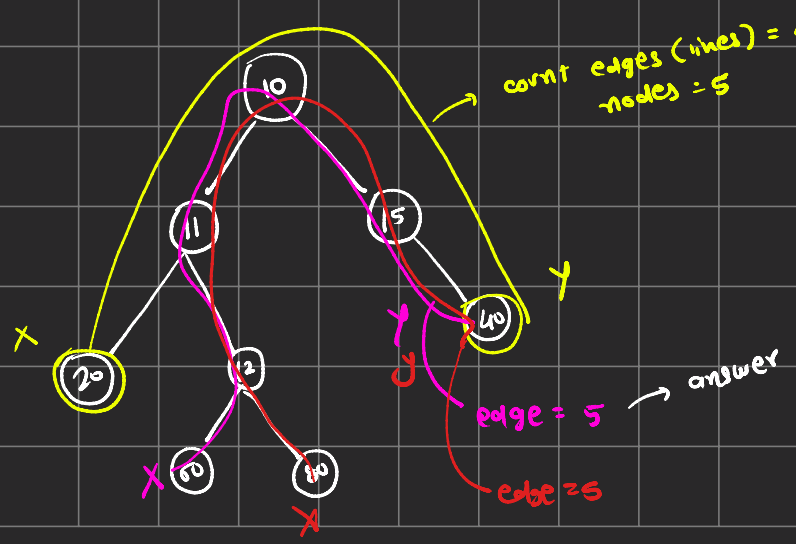
* Height of Tree / max depth of binary tree → total nums of level

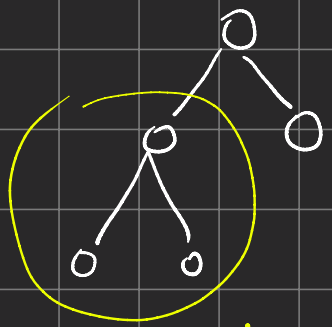



```
int maxDepth(TreeNode* root) {
    if(root==NULL) return 0;

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

    int heigth = max(leftHeigth, rightHeight) + 1;

    return heigth;
}
```

† we can solve with level order as above

* Diameter of tree





just think X, Y

both are in left

count distance between X, Y

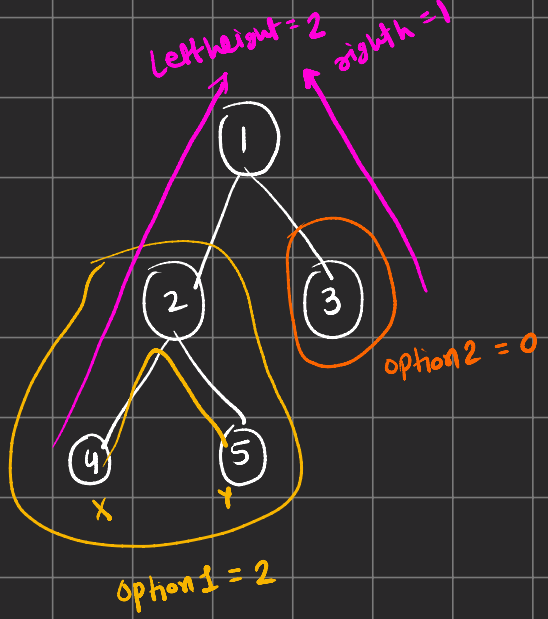
for X Y

① → XY Both in left sub tree

② → XY Both in right subtree

③ → X in left and Y in right
y in left and x in left

→ MAX will be Ans



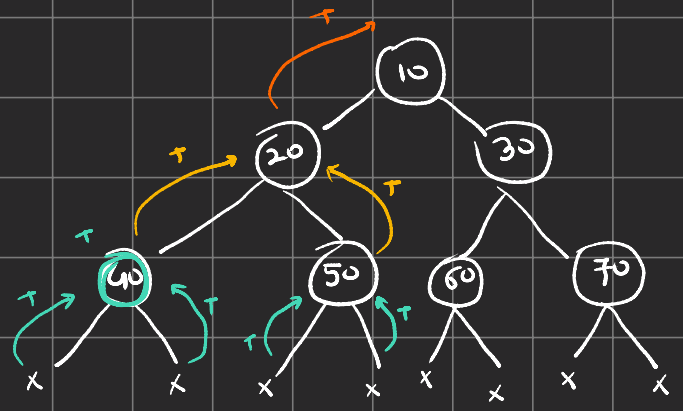
option 3 = leftheight + rightheight

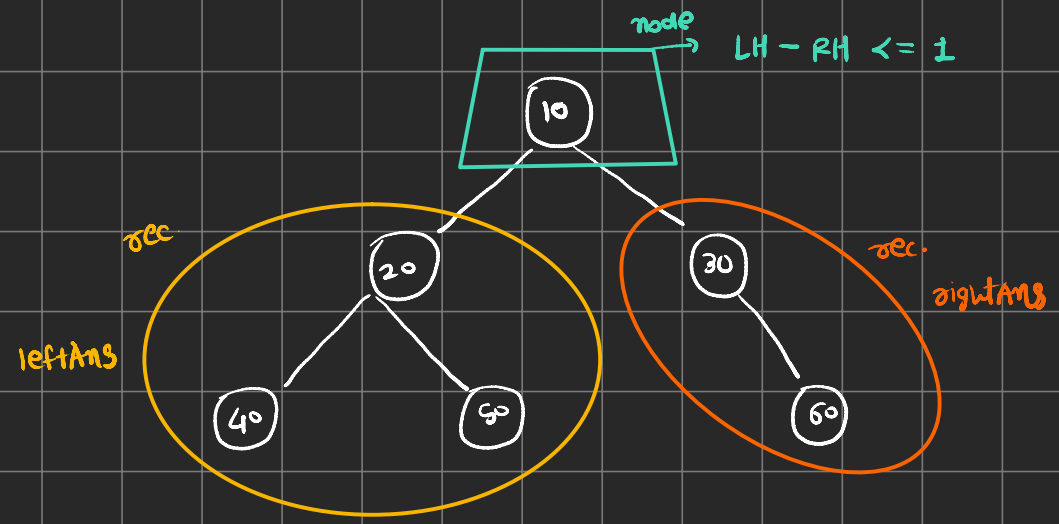
```
int height(TreeNode* root) {
        if(root==NULL) return 0;

        int leftHeigth = height(root->left);
        int rightHeight = height(root->right);

        int heigth = max(leftHeigth, rightHeight) + 1;

        return heigth;
    }

int diameterOfBinaryTree(TreeNode* root) {

    if(root==NULL) return 0;

    int option1 = diameterOfBinaryTree(root->left);
    int option2 = diameterOfBinaryTree(root->right);
    int option3 = height(root->left) + height(root->right);

    int diameter = max(option1, max(option2, option3));

    return diameter;

}
```

option 3 = left H + right H
+1 or -1
can be there according to definition of diameter

✱ Balanced Binary tree

- ek tree me kisi bhi (har node) pur jakar kch sakta hun ki left sub tree ki height and right sub tree ki height ka absolute difference $\underbrace{<= 1}_{\text{at most 1}}$ → height balanced BT

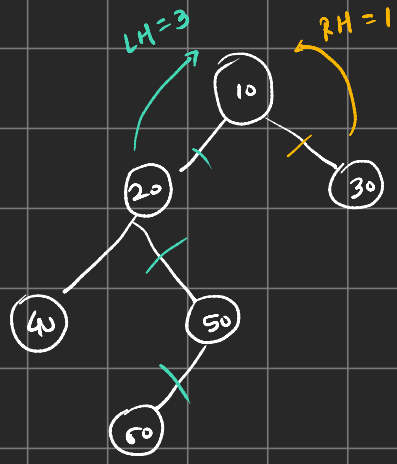




if (root = NULL) → Balanced

① left subtree → Balanced
② right subtree → Balanced
③ abs(LH - RH) <= 1

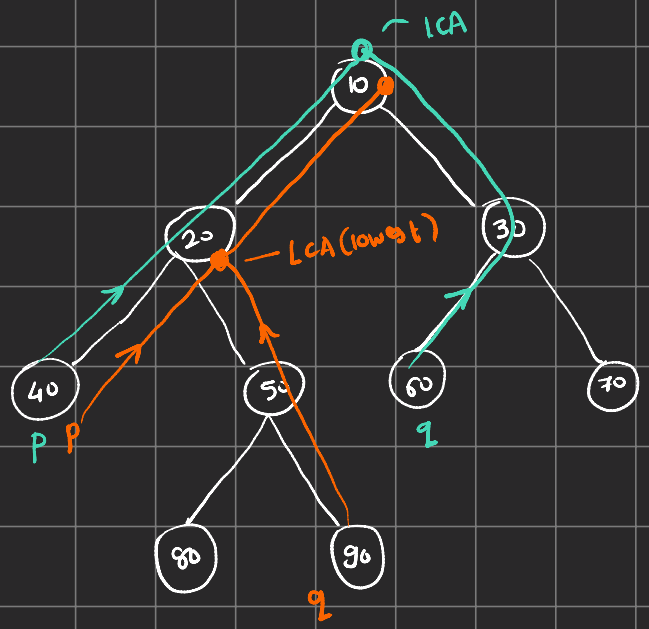
} entire tree Balanced



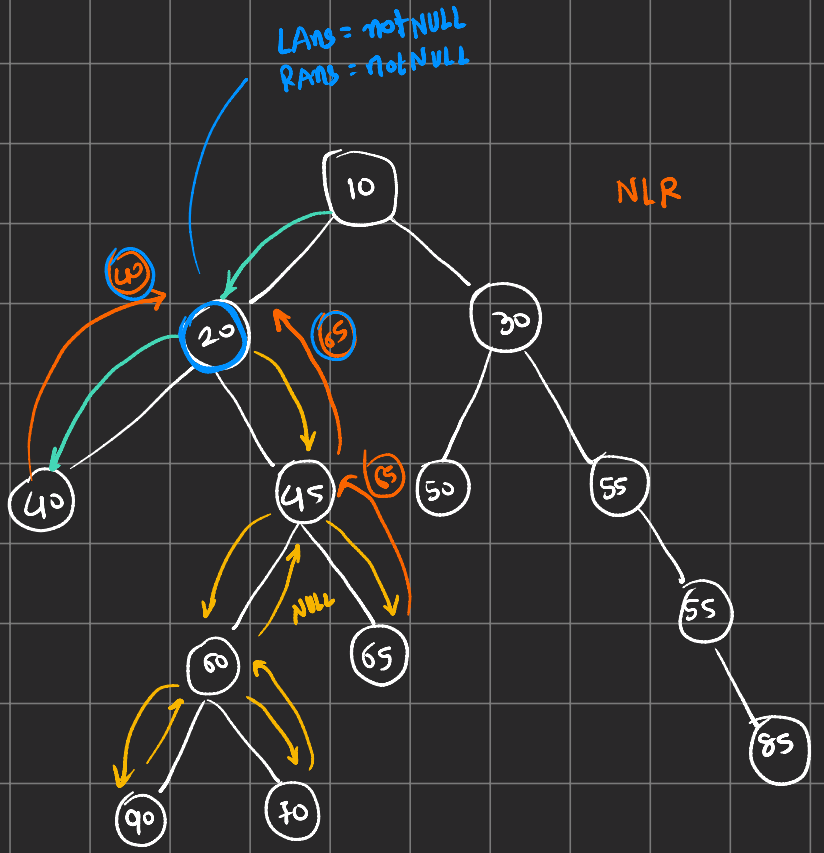
= abs(3-1)
= 2 <= 1 (False)

```
bool isBalanced(TreeNode* root) {

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

    // current node ans
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    int diff = abs(leftHeight-rightHeight);

    bool currentNodeAns = (diff <= 1);

    // left sub tree ans
    bool leftAns = isBalanced(root->left);

    // right sub tree ans
    bool rightAns = isBalanced(root->right);

    return currentNodeAns && leftAns && rightAns;

}
```

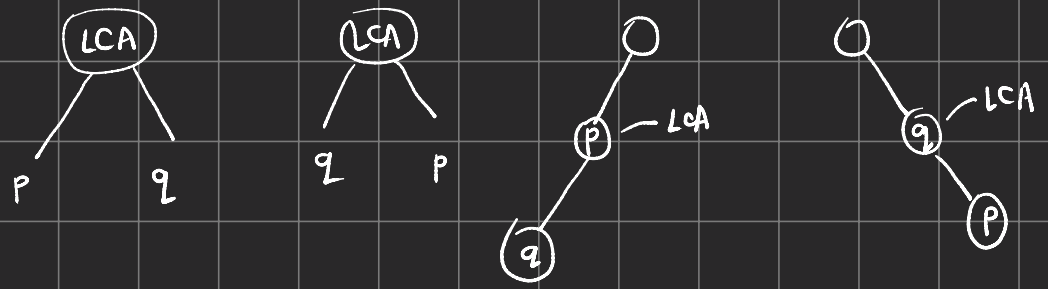
LCA (lowest common ancestor) in Binary tree

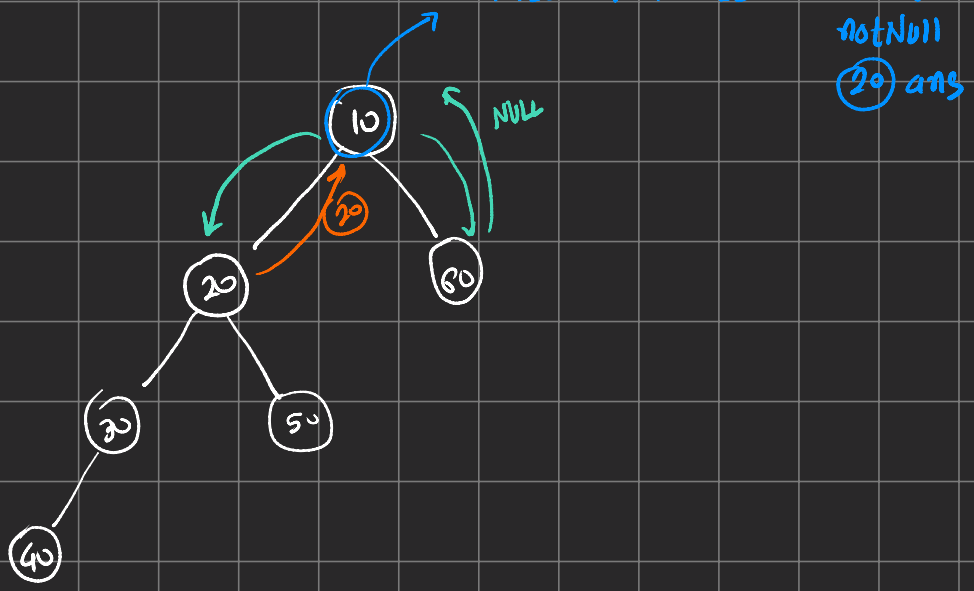


- P se upar and q se upar jaha sabse penle miljate hai wo LCA



cases

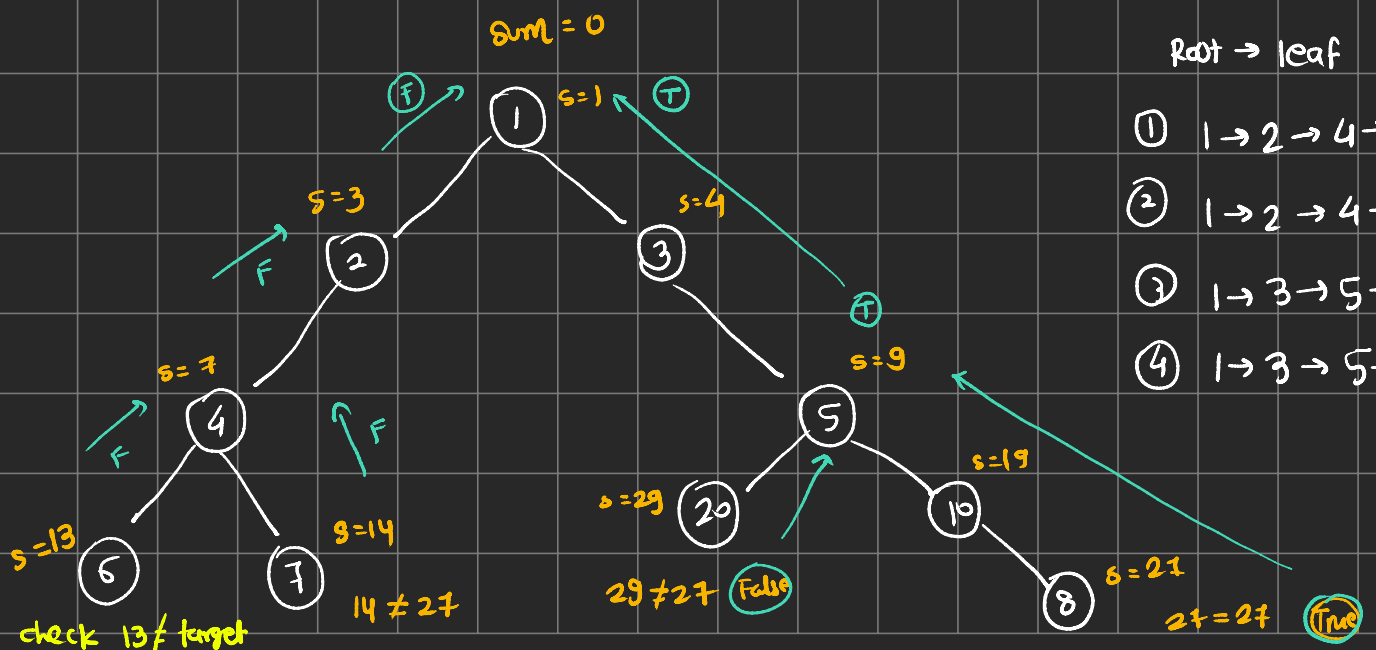





```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if(root==NULL) return NULL;

    if(root->val==p->val) return p;
    if(root->val==q->val) return q;

    TreeNode* leftAns = lowestCommonAncestor(root->left,p,q);
    TreeNode* rightAns = lowestCommonAncestor(root->right,p,q);

    if(leftAns==NULL && rightAns==NULL){
        return NULL;
    }else if(leftAns!=NULL && rightAns==NULL){
        return leftAns;
    }else if(leftAns==NULL && rightAns!=NULL){
        return rightAns;
    }else{
        // both are not null so actual node is ans
        return root;
    }
```

★ Path Sum

T || F = T



targetsum = 27

Root → leaf

① 1 → 2 → 4 → 6 = 13

② 1 → 2 → 4 → 7 = 14

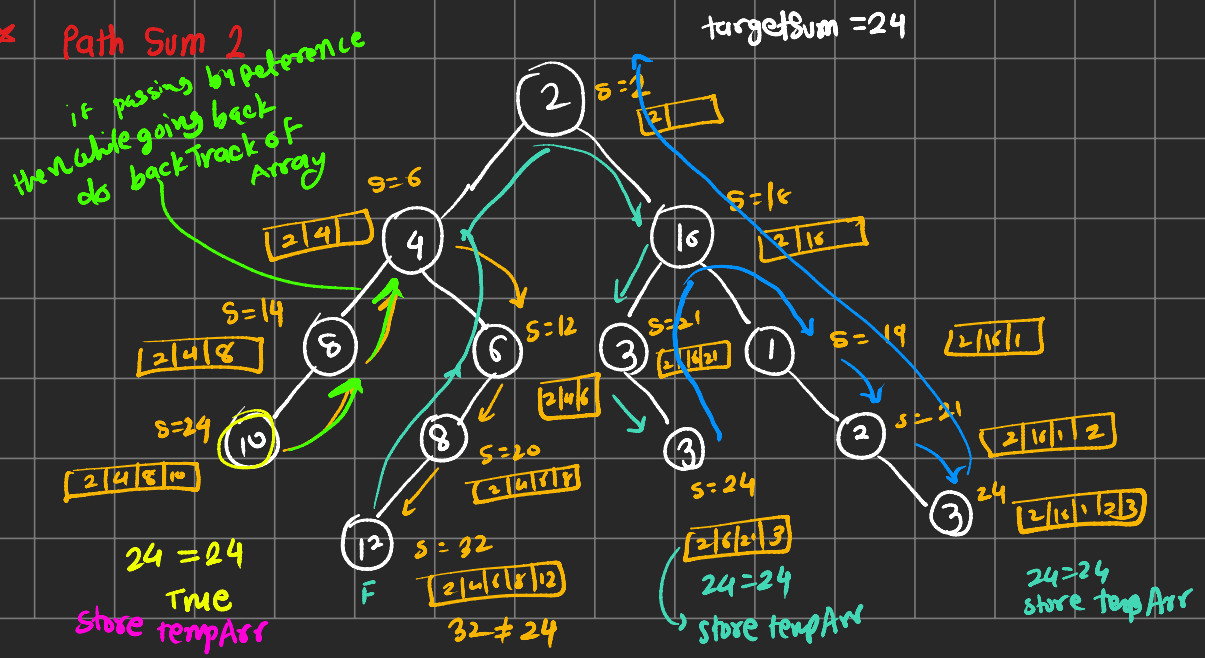
③ 1 → 3 → 5 → 20 = 29

④ 1 → 3 → 5 → 10 → 8 = 27 → existed

```cpp
bool solve(TreeNode* root, int targetSum, int sum){
    if(root==NULL){
        return false;
    }

    sum = sum + root->val;  // adding

    // leaf node
    if(root->left==NULL && root->right==NULL){
        if(sum==targetSum){
            return true;
        }else{
            return false;
        }
    }

    bool leftAns = solve(root->left, targetSum, sum);
    bool rightAns = solve(root->right, targetSum, sum);

    return leftAns || rightAns;

}

bool hasPathSum(TreeNode* root, int targetSum) {
    return solve(root,targetSum, 0);
```

Path Sum 2

if passing by reference then while going back do backTrack of Array

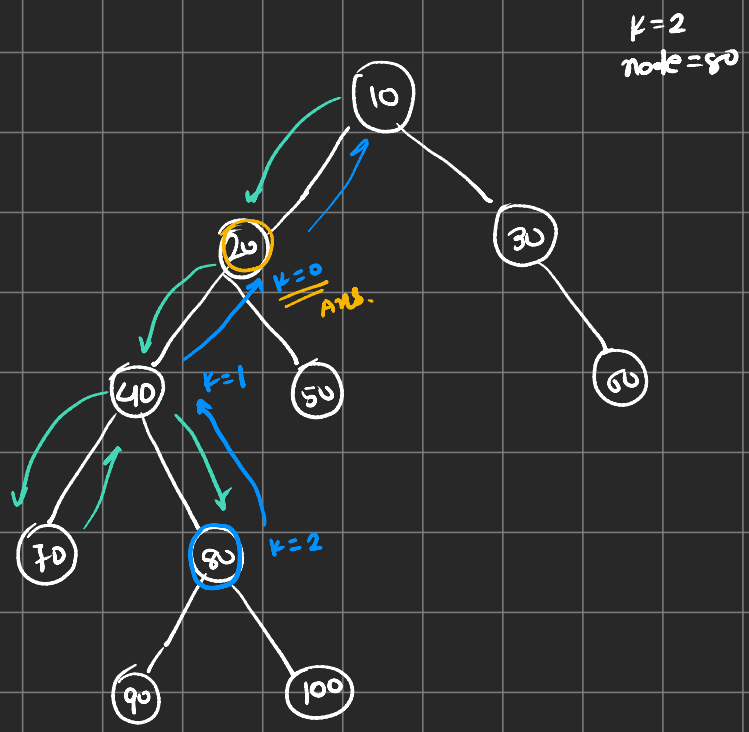



```cpp
void solve(TreeNode* root, int targetSum, vector<vector<int>>& ans, vector<int> temp, int sum){
    if(root==NULL){
        return;
    }

    sum = sum + root->val;
    temp.push_back(root->val);

    if(root->left==NULL && root->right==NULL){
        // leaf node
        if(sum==targetSum){
            ans.push_back(temp);
        }else{
            return;
        }
    }

    solve(root->left, targetSum, ans, temp, sum);
    solve(root->right, targetSum, ans, temp, sum);

}

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    vector<vector<int>> ans;
    vector<int> temp;
    int sum=0;
    solve(root, targetSum, ans, temp, sum);

    return ans;
}
```

```cpp
void solve(TreeNode* root, int targetSum, vector<vector<int>>& ans, vector<int>& temp, int sum){
    if(root==NULL){
        return;
    }

    sum = sum + root->val;
    temp.push_back(root->val);

    if(root->left==NULL && root->right==NULL){
        // leaf node
        if(sum==targetSum){
            ans.push_back(temp);
        }
    }

    solve(root->left, targetSum, ans, temp, sum);
    solve(root->right, targetSum, ans, temp, sum);

    temp.pop_back();
}
```

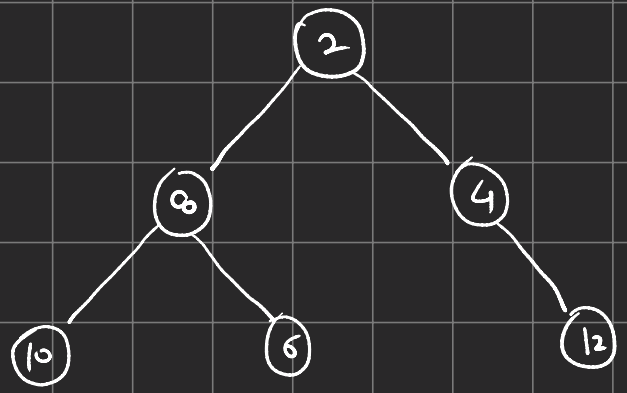
by ref.

✱ k^{th} Ancestor of a Node in Binary tree



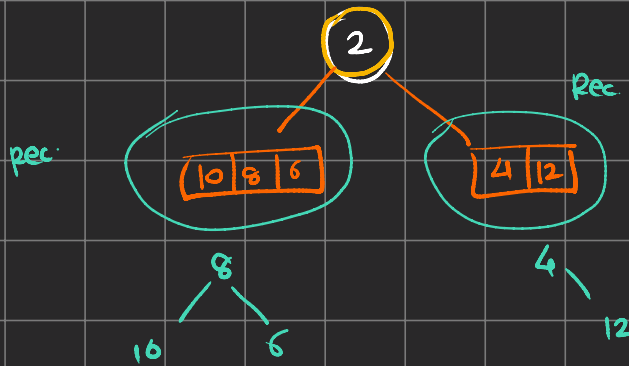
① node tak pahuncho
② then k ki value minus karte jao

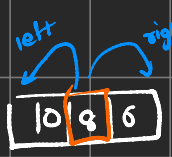
✱ Preorder and Inorder traversal you have, from them create Tree

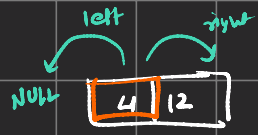


preorder (NLR) = 2, 8, 10, 6, 4, 12 (root node; left → right)

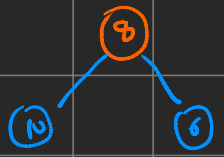
Inorder (LNR) = 10, 8, 6, 2, 4, 12 (L, N, R)







according to preorder (NLR) 2, 8, 10, 6, 4, 12 (left most)




```cpp
int searchInorder(vector<int>& inorder, int target){
    for(int i=0; i<inorder.size(); i++){
        if(inorder[i] == target){
            return i;
        }
    }
    return -1;
}

TreeNode* solve(vector<int>& preorder, vector<int>& inorder, int& preOrderIndex, int inorderStart, int inorderEnd){
    if(preOrderIndex>=preorder.size() || inorderStart>inorderEnd) return NULL;

    int ele = preorder[preOrderIndex];
    preOrderIndex++;

    TreeNode* root = new TreeNode(ele);

    // element search in inorder
    int position = searchInorder(inorder, ele);

    root->left = solve(preorder, inorder, preOrderIndex, inorderStart, position-1);
    root->right = solve(preorder, inorder, preOrderIndex, position+1, inorderEnd);

    return root;
}

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    int preIndex = 0;
    return solve(preorder, inorder, preIndex, 0, inorder.size()-1);
}
```

By ref. Imp

searching taking O(n) everytime
using mapping we can do in O(1)

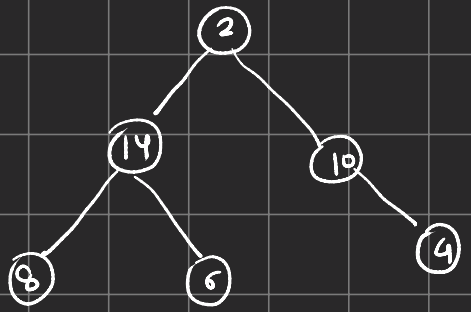
```
void createMapping(vector<int>& inorder, map<int, int>& valueToIndexMap){
    for(int i=0; i<inorder.size(); i++){
        int element = inorder[i];
        int index = i;
        valueToIndexMap[element] = index;
    }
}
```

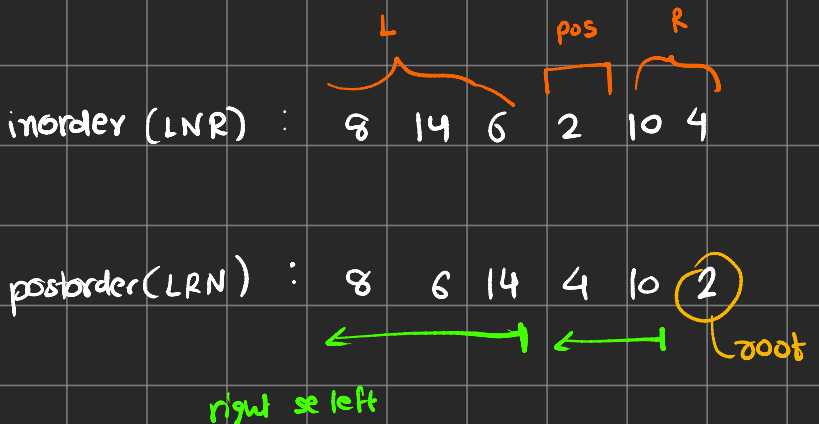
```
// element search in inorder
// int position = searchInorder(inorder, ele);
int position = valueToIndexMap[ele];
```

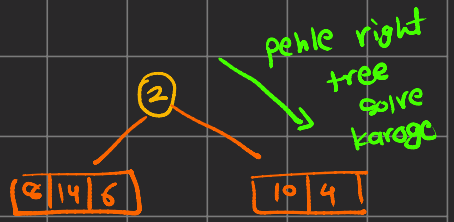
Why inorderIndex by reference?

- inorder k array par travel karte time kisi value ko dubara use karne bcoz jagah to issue hogi
- inorder ke ek element ko bas ek baar use karna chahta hu so us index par vapas naa aaysaku.

create tree using Postorder and Inorder





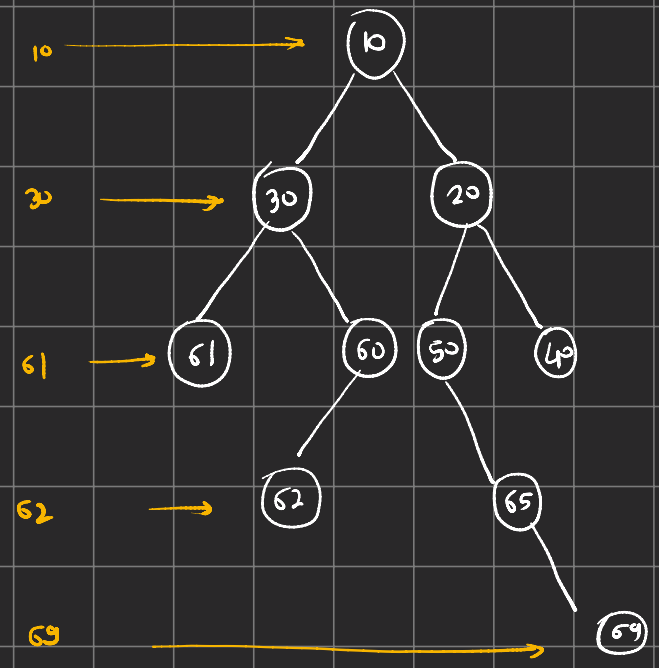


```cpp
void createMapping(vector<int>& inorder, map<int, int>& valueToIndexMap) {
    for (int i = 0; i < inorder.size(); i++) {
        int element = inorder[i];
        int index = i;
        valueToIndexMap[element] = index;
    }
}

TreeNode* solve(vector<int>& inorder, vector<int>& postorder, int& postOrderIndex, int inorderStart, int inorderEnd, map<int, int>&
valueToIndexMap) {
    if (postOrderIndex < 0 || inorderStart > inorderEnd)
        return NULL;

    int ele = postorder[postOrderIndex];
    postOrderIndex--;

    TreeNode* root = new TreeNode(ele);

    // element search in inorder
    int position = valueToIndexMap[ele];

    // right first
    root->right = solve(inorder, postorder, postOrderIndex, position + 1, inorderEnd, valueToIndexMap);
    root->left = solve(inorder, postorder, postOrderIndex, inorderStart, position - 1, valueToIndexMap);

    return root;
}

TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
    int postIndex = postorder.size()-1;
    map<int, int> valueToIndexMap;
    createMapping(inorder, valueToIndexMap);
    return solve(inorder, postorder, postIndex, 0, inorder.size()-1, valueToIndexMap);
}
```

☆ Views

M1 har level ka first element - do level traversal then first ele from each lvl

Left view 😊



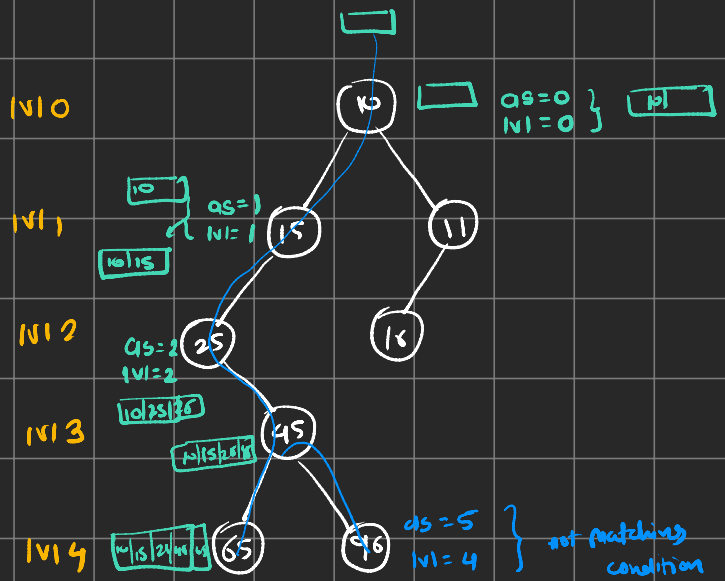
0

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

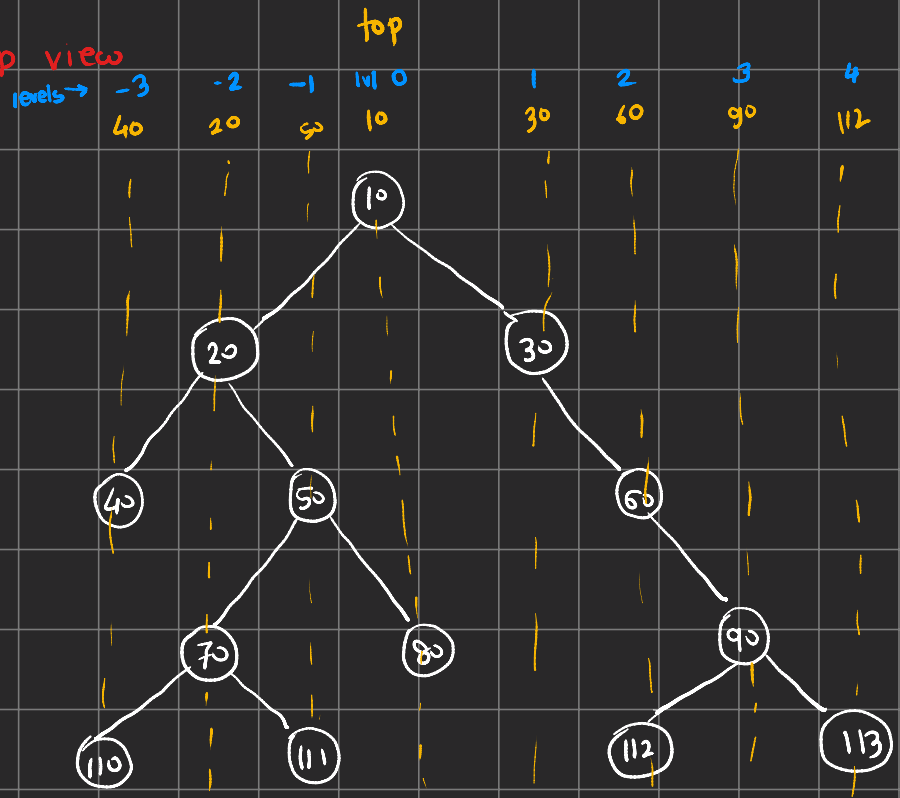
To print right view
we first need to call right
// right
// left

M2

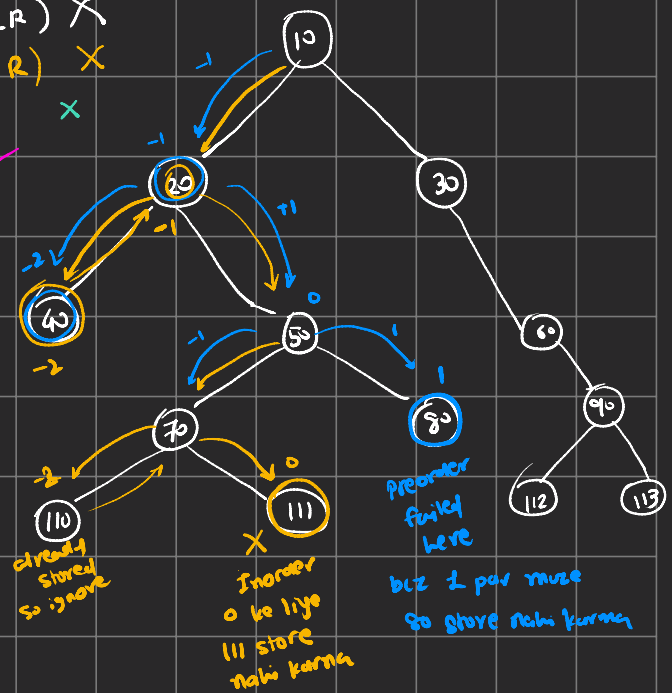
arr.size = lvl → push



★ Top view

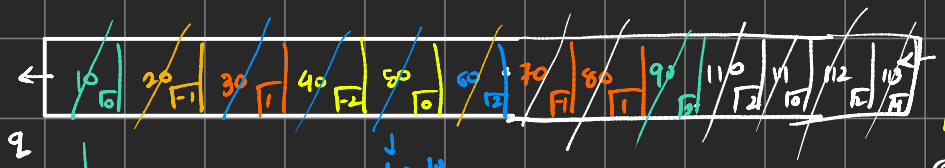


Preorder (NLR) ✓
Inorder (LNR) ✗
Postorder ✗
level order ✓



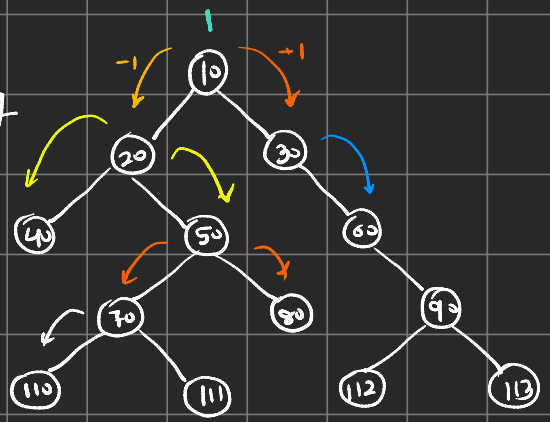
map

lvl	ans
-2	40
-1	20
0	10
1	30
2	60
3	90
4	113



pop karke hm uske left and right child ko que mei push karo

already value of 0 is there so no need to update



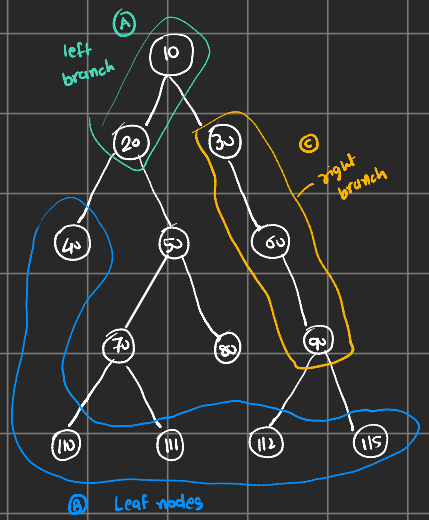
```cpp
void oragnizeAnsFromMap(map<int, int>& hdToNodemap, vector<int>& ans){
    for(auto i: hdToNodemap){
        ans.push_back(i.second);
    }
}

vector<int> topView(Node *root) {
    // code here

    map<int, int> hdToNodemap;  // node andhorizontal distance
    queue< pair<Node*, int> >q;

    q.push(make_pair(root,0));

    while(!q.empty()){
        pair<Node*, int> top = q.front();
        q.pop();

        Node* frontNode = top.first;
        int hd = top.second;

        // if there is no entry in map then create a new entry
        if(hdToNodemap.find(hd)==hdToNodemap.end()){
            hdToNodemap[hd] = frontNode->data;
        }

        // child ko dekhna he
        if(frontNode->left != NULL){
            q.push(make_pair(frontNode->left, hd-1));
        }
        if(frontNode->right != NULL){
            q.push(make_pair(frontNode->right, hd+1));
        }
    }

    vector<int> ans;
    oragnizeAnsFromMap(hdToNodemap, ans);
    return ans;
}
```

only

without the condition overwrite value / update value again for BOTTOMVIEW

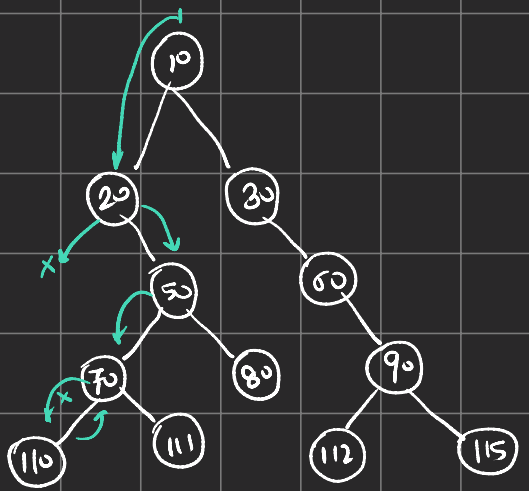
just remove if condition without if update value every time

✭ Boundry Traversal



→ PartA : leaf node milne se pehle tak
→ Part B
→ partC

```
void printLeftBoundry(Node *root, vector<int>& ans){
    if(root==NULL){
        return;
    }

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

    ans.push_back(root->data);

    if(root->left!=NULL){
        printLeftBoundry(root->left,ans);
    }else{
        printLeftBoundry(root->right,ans);
    }
}

void printLeafBoundry(Node *root, vector<int>& ans){
    if(root==NULL){
        return;
    }
    if(root->left==NULL && root->right==NULL){
        ans.push_back(root->data);
    }
    printLeafBoundry(root->left,ans);
    printLeafBoundry(root->right,ans);
}

void printRightBoundry(Node *root, vector<int>& ans){
    if(root==NULL){
        return;
    }
    if(root->left==NULL && root->right==NULL){
        return;
    }

    if(root->right!=NULL){
        printRightBoundry(root->right,ans);
    }else{
        printRightBoundry(root->left,ans);
    }
    ans.push_back(root->data);  // niche se print karna he
}
```



Leafnode bhi print nahi karni hai usse pehle se he niklajaega

Leafnode milte he vapas chale jao right me jane ki need nahi hai

follow this order wise isse hoga live repetation cool..

```
vector<int> boundaryTraversal(Node *root) {
    vector<int> ans;
    ans.push_back(root->data);
    printLeftBoundry(root->left,ans);
    printLeafBoundry(root->left,ans);
    printLeafBoundry(root->right,ans);
    printRightBoundry(root->right, ans);
    return ans;
}
```

Fast way to find diameter

```
int D = 0;
int height(TreeNode*root){
    if(!root) return 0;

    int lh = height(root->left);
    int rh = height(root->right);
    // diameter
    int currD = lh + rh;
    D = max(D, currD);
    return max(lh, rh) + 1;
}

int diameterOfBinaryTree(TreeNode* root) {
    height(root);
    return D;
}
```

currentdiameter = lh + rh

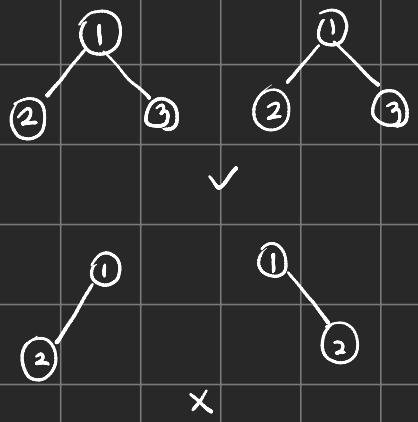
Fast way to find height balanced tree

node par abs(leftHeight - rightHeight) <= 1

```
bool isbalanced = true;
int height(TreeNode*root){
    if(!root) return 0;

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

    //check for current node, is it balanced?
    if(isbalanced && abs(lh - rh) > 1){
        isbalanced = false;
    }

    return max(lh, rh) + 1;
}

bool isBalanced(TreeNode* root) {
    height(root);
    return isbalanced;
}
```

ek node par bhi condition true ho gayi toh ye balanced tree nahi hai

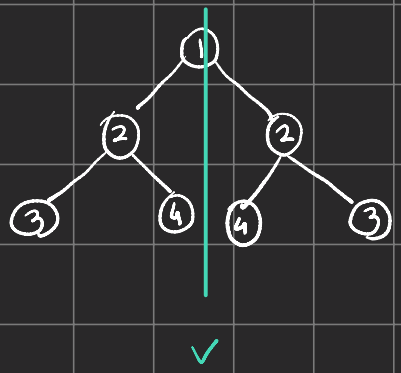
* same Tree identical

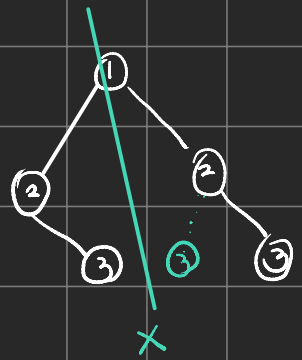


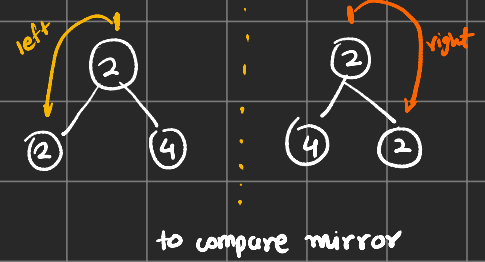
```cpp
bool isSameTree(TreeNode* p, TreeNode* q) {
    if(p==NULL && q==NULL) return true;

    if(!((p!=NULL && q!=NULL) && (p->val == q->val))){
        return false;
    }

    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
}
```

true to be a similar

* Symmetric Tree / mirror



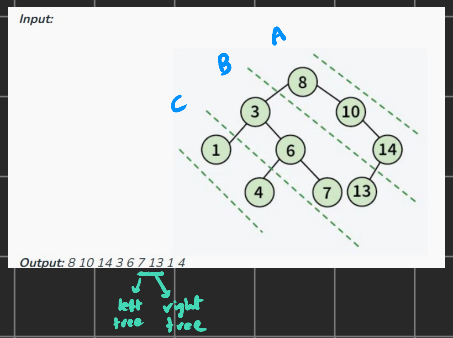




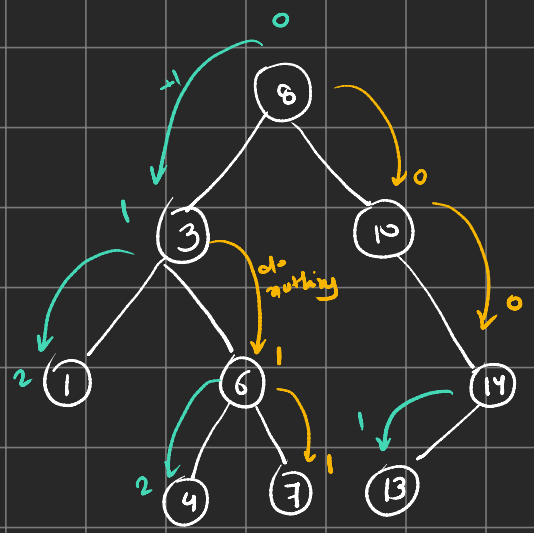
to compare mirror

```cpp
bool isMirror(TreeNode* p, TreeNode* q) {
    if(p==NULL && q==NULL) return true;

    if(!((p!=NULL && q!=NULL) && (p->val == q->val))){
        return false;
    }

    return isMirror(p->left, q->right) && isMirror(p->right, q->left);
}

bool isSymmetric(TreeNode* root) {
    return isMirror(root->left, root->right);
}
```

Diagonal Traversal



M1



→ going left : +1
→ going right : do nothing

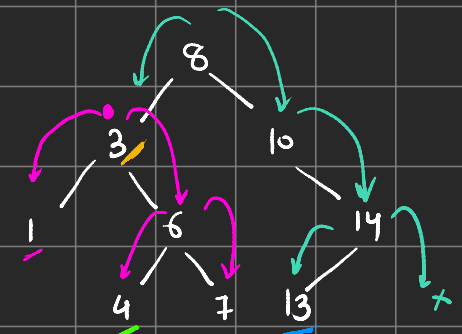
0 → diagonal A
1 → diagonal B
2 → diagonal C

0 →	8 10 14
1 →	3 6 7 13
2 →	1 4

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

M2

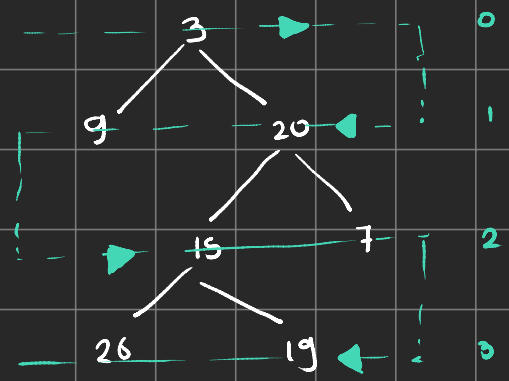
Level order Traversal (in which we were using queue)




```
temp = 8 3 13 X 4
while (temp != NULL)
{
    print temp -> val
    if (temp -> left) ~> q.push
    temp = temp -> right
}
```

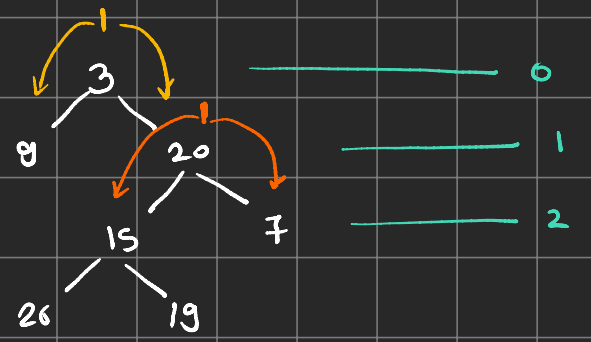

```
vector<int> diagonal(Node *root) {
    vector<int> ans;
    queue<Node*> q;

    q.push(root);

    while(!q.empty()){
        Node* temp = q.front();
        q.pop();

        while(temp){
            ans.push_back(temp->data);
            if(temp->left){
                q.push(temp->left); // baad me dekhenge
            }
            temp = temp->right;
        }
    }

    return ans;
```

* Zig Zag Traversal



[[3], [20,9], [15,7], [19,26]]

looks like level traversal
only alternatives in right to left



flag (leftToRight)

0 T
1 F
2 T

① 3 → [3] T

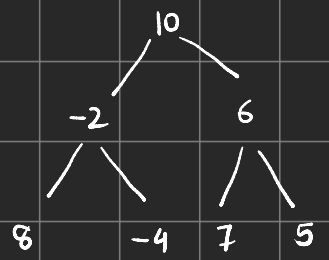
② 9 → [20,9] F

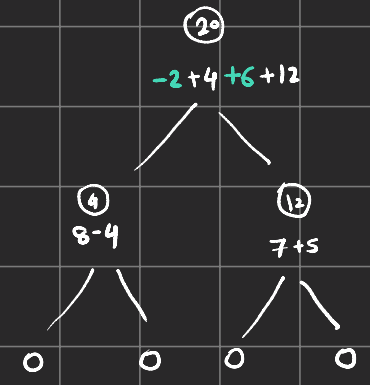
③ 15 → [15,7] T

④ 26 → [19,26] F

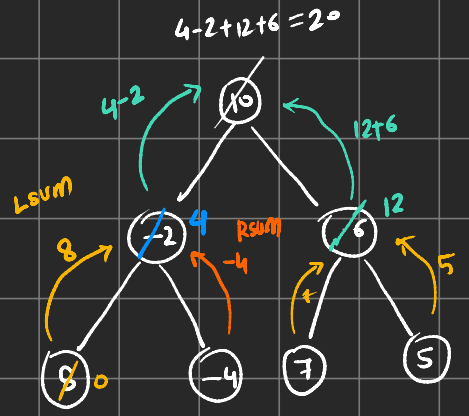
```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> ans;
    if(root==NULL) return ans;  // base case
    bool leftToRightDirection = true;
    queue<TreeNode*> q;
    q.push(root);

    while(!q.empty()){
        int width = q.size();   <- current level par total events
        vector<int> oneLevel(width);
        for(int i=0; i<width; i++){
            TreeNode* front = q.front();
            q.pop();

            int index = leftToRightDirection ? i : width-i-1;
            oneLevel[index] = front->val;

            if(front->left) {
                q.push(front->left);
            }
            if(front->right){
                q.push(front->right);
            }
        }
        // toogle when lvl change
        leftToRightDirection = !leftToRightDirection;
        ans.push_back(oneLevel);
    }

    return ans;
```

★ Transform to Sum tree



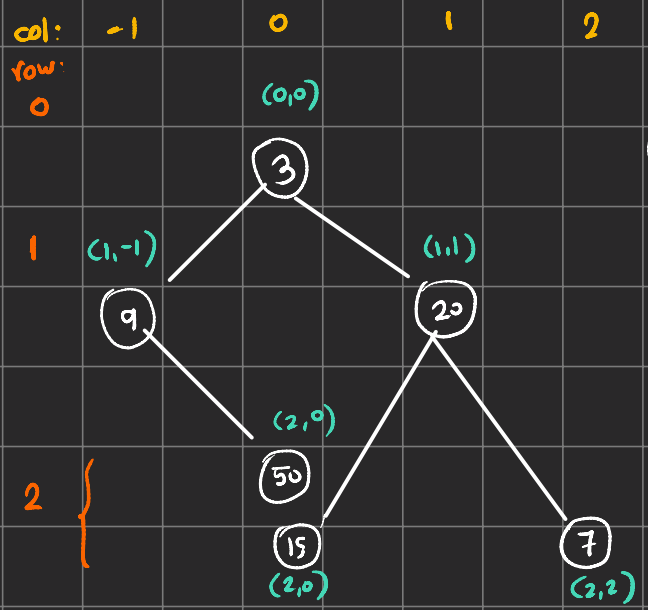


→leaf node = 0




```
int solve(Node* root){
    if(!root) return 0;

    if(root->left == NULL && root->right==NULL){    // leaf node
        int valOfNode = root->data;
        root->data = 0;
        return valOfNode;
    }

    int leftSum = solve(root->left);
    int rightSum = solve(root->right);

    int currentNodeVal = root->data;
    root->data = leftSum+rightSum;
    return currentNodeVal + (root->data);

}
void toSumTree(Node *node)
{
    solve(node);
}
```

✱ Vertical order Traversal of Binary Tree



(row, col)

(r+1, c-1) (r+1, c+1)

left most column se start → end (top → bottom)

-1 → 9

0 → 3, 15, 50

1 → 20

2 → 7

if same location par hai to sort by the value

multiset → able to store values in sorted order automatically

→ duplicates allowed (nor in SET)

```cpp
#include <set>

int main() {
    std::multiset<int> ms;

    // Inserting elements in random order
    ms.insert(30);
    ms.insert(10);
    ms.insert(20);
    ms.insert(10); // Duplicate

    // Elements are automatically sorted in ascending order
    std::cout << "Multiset elements (sorted): ";
    for (int x : ms) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output:

```
java

Multiset elements (sorted): 10 10 20 30
```

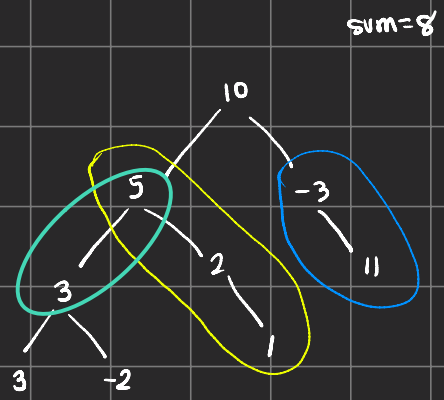
Map { col → map<row, multiset> }

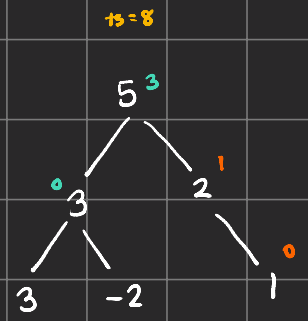
col	row	values
-2	2 →	4
-1	1 →	2
0	0 →	1
	2 →	5 6 → auto sorted multiset
1	1 →	3

```cpp
vector<vector<int>> verticalTraversal(TreeNode* root) {
    vector< vector<int> > ans;
    queue<pair <TreeNode*, pair<int,int>> > q;  // node, {row,col}
    q.push({root, {0,0}});

    map<int, map<int, multiset<int>> > mp;  // col -> {row: [x,y,z...]}

    while(!q.empty()){
        auto front = q.front();
        q.pop();

        TreeNode*& node = front.first;
        auto coordinate = front.second;
        int& row = coordinate.first;
        int& col = coordinate.second;

        mp[col][row].insert(node->val);

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

    // store final verticle order into ans vector
    for(auto it:mp){
        auto& colMap = it.second;
        vector<int> vLine;
        for(auto colMapIt: colMap){
            auto &mset = colMapIt.second;
            vLine.insert(vLine.end(), mset.begin(), mset.end());
        }
        ans.push_back(vLine);
    }

    return ans;
```

Tricky Implementation

Path sum 3 / K sum path






```
int ans=0;

void pathFromOneRoot(TreeNode* root, long long sum){
    if(root==NULL) return;

    if(sum==root->val){
        ans++;
    }

    pathFromOneRoot(root->left, sum-root->val);
    pathFromOneRoot(root->right, sum-root->val);
}

int pathSum(TreeNode* root, long long targetSum) {
    if(root){
        pathFromOneRoot(root, targetSum);
        pathSum(root->left, targetSum);
        pathSum(root->right, targetSum);
    }
    return ans;
}
```

condition that not only from root but from other nodes also