Note: Height + left Node = 0
→ height (node) = 1 + max (height, child nodes)

```
int height(node *root){
    if(root == NULL){ return -1 }
    int l = height(root->left)
    int r = height(root->right)
    // Find height of current node
    return 1 + max(l,r)
```

Q5) Given root node find length of longest path which is going across the root node.

Eg:

Eg 2:

ans: -1+0+1 = 1

* 5:30 pm
→ 4:55 pm Doubts
8:00 pm Sharp:

* For root node: length of max path going across the root node
→ height of left subtree + height of right subtree + 2

Method

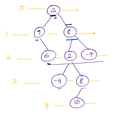
For any node: length of max path going across the root node
→ height of left subtree + height of right subtree + 2

```
int ans = 0
int height(node *root){
    if(root == NULL){ return -1 }
    int l = height(root->left)
    int r = height(root->right)
    // length of max path going
    // with the current node :
    ans = max(ans, l+r+2)
    // Find height of current node
    return 1 + max(l,r)
}
```

→ O(N) → TC
→ SC: O(H)
height of tree

{ ans: length of max path across entire BT. // Diameter of Binary Tree }

Level order traversal:



Expected output:
3 9 8 6 2 -7 -4 8 10

Approach 1

3	X	X	9	X	8	X	X	X

output:
3 9 8 6 2 -7 -4 8 10

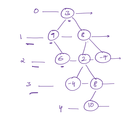
3	level 0
9 8	level 1
6 2 -7 -4	level 2
8 10	level 3
10	level 4

Idea: Take Queue<Node*> que;
que.push(root);

Repeat: Repeat entire approach, until que is empty.

At every step
1. Get front node
2. Delete front element from queue.
3. Print front node
4. (Insert left child) (Insert right child) } Make sure they are not null

TC: O(N) SC: O(Breadth) → In above ex 4 + 8 + 2 → 3



Expected output:
3
9 8
6 2 -7
-4 8
10

3	NULL	9	8	NULL	6	2	-7	NULL	-4	8	NULL	10	NULL

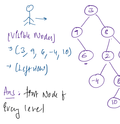
output: 3 \n
9 8 \n
6 2 -7 \n
-4 8 \n
10 \n

To avoid infinite loop

Note 1: Insert root node & NULL in que
Note 2: While(que.size()>1)

Note: Inside while loop when ever we get NULL
→ print("\n")
→ pop NULL from front
→ push NULL at rear.

Q6)



(Middle View)
→ (3, 9, 6, -4, 10)
→ (Left View)

Ans: First Node of every level

Q: level order from right → left?
→ After popping a node
Step 1: Insert right child
Step 2: Insert left child

Q: Right View of BT?
output: 3, 8, -7, 8, 10
Ans: First Node of every level from R→L
level order. ~
Last Node at every level to L→R first order

Q5) Vertical level order traversal (L→R)

[from left → Right]

Expected output

-10	
2	6
4	9 10
6	3 8
8	22
9	25
10	

Bottom View

Top View

Top View: First node of each level in vertical level order traversal
Bottom View: last node of each level in vertical level order traversal.

Idea:

In function we need to pass d as well.

Idea: store based on depth values.

Ans:

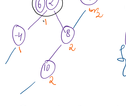
Expected output:
6
9 2
4 8

observe:
postorder/preorder/Inorder
Now it has one working?
→ now if the above horizontals are because it gives a higher preference to other left children or right children
Note: Data should be filled level by level, so that can solve the problem.

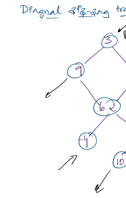
0 : 1, 2
-1 : 6
1 : 8, 4
} output we get based on preorder. Not matching

0 : 2 4
-1 : 6
1 : 8 4
} output we get based on inorder. Not matching

→ Diagonal traversal



output:
3 9
8 2 6 -4
-7 8 10



} Only change

} do the modification to vertical level order traversal & entire code works.

Diagonal zig-zag traversal

Alternative levels print in reversed order.

0 : 3 9
1 : 8 2 6 -4
2 : -7 8 10
3 :

Note: For your future reference
→ level order traversal < fb to bottom / preorder / Inorder / Morris traversal
→ Left View < preorder + Recursion + Note: For optimal approach we need to use map<int, int>
preorder(node *root, int depth)
Passing all the above information it can be done easily
int: Key based on depth
int: Node Value
→ Right View — postorder + Recursion
preorder(node *root, int depth)
Passing all the above information it can be done easily
→ Top View — } can be done with any of the
→ Bottom View — } PO/POST/In orders

```
void traversal(Node *root, int depth, int level){
    if(root == NULL) return;
    // For current node is at depth, & level l
    if( hm.find(l) == hm.end()){
        // That means the new level l node didn't come
        hm[l] = {root->data, depth}
        Insert node along its current depth
    }
    else {
        int &r, int &d = hm[l];   // node already stored at level l
        if( depth < d ){
            hm[l] = {root->data, level}
        }
    }
    traversal(root->left, depth+1, level-1);
    traversal(root->right, depth+1, level+1);
}
// Top View: In above code, for every level we are storing node at min depth
```

Bottom View: For every level we are getting node at max depth.

```
if( depth > d ){
    hm[l] = {root->data, level}
}
```

Note: All views, left, right, top, bottom, they can be done recursively.
level order traversal can not done
→ preorder/Inorder + (External Storage)

Recollate:

```
int noOfL = 1, maxL = 0;
Queue<Node*> que;
que.enque(root);
unordered_map<int, vector<int>> ump;
while(que.size()>0){
    int &r, int &l = que.front();
    que.deque();// Delete from front
    Node *temp = f.first;
    int l = f.second;
    ump[l].push_back(temp->val);
    noOfL = min(noOfL, l);
    maxL = max(maxL, l);
    if( temp->left != NULL){
        que.enque(temp->left, l-1)
    }
    if( temp->right != NULL){
        que.enque(temp->right, l+1)
    }
}
// Printing
for(int i = noOfL; i <= maxL; i++){
    vector<int> &v = ump[i];
    for(int j = 0; j < v.size(); j++){
        cout << v[j];
    }
    cout << "\n";
}
```

Key is based on level L→R
→ data & Nodes level by level
→ accessing ump at level i

Note: If any min level = a
Any max level = b
All levels from a to b have to present

Simple modification
If at all top & bottom View value can only be right
// Based on that there will be small modification

Top View	Bottom View
cout << data[0]	cout << data[data.size()-1]