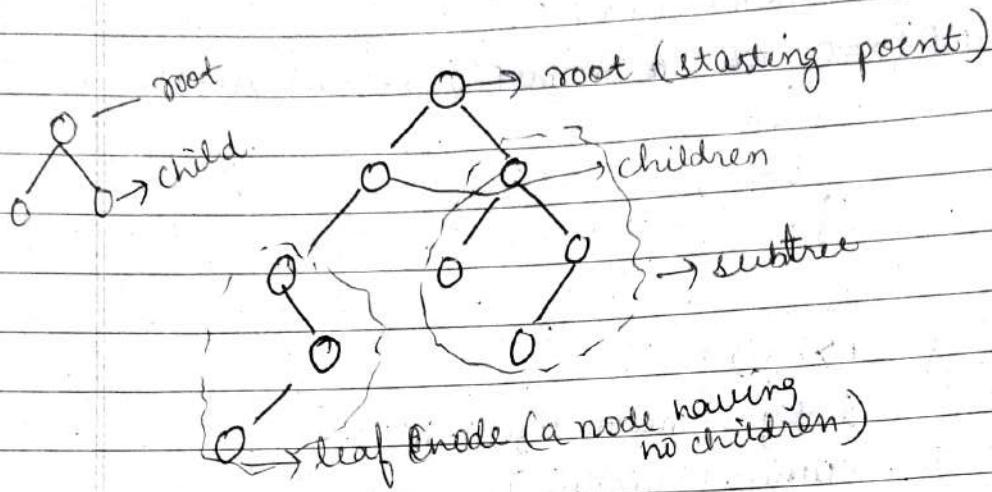


## Tree Data Structure

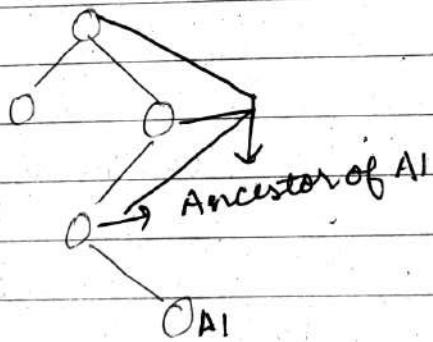
### Tree series

#### Introduction to Binary trees.

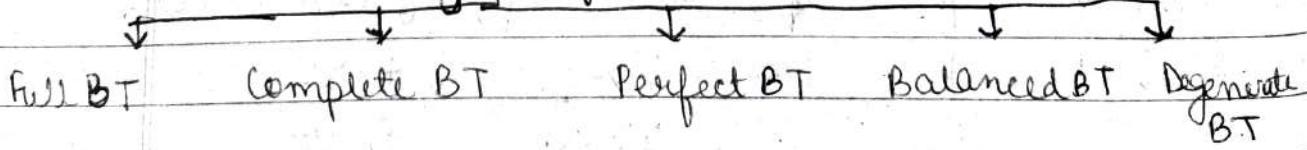


Tree means a hierarchy and Binary means 2.  
So a binary tree means a node can have at max 2 childs

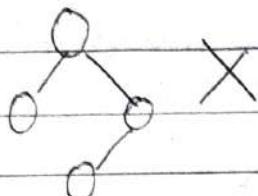
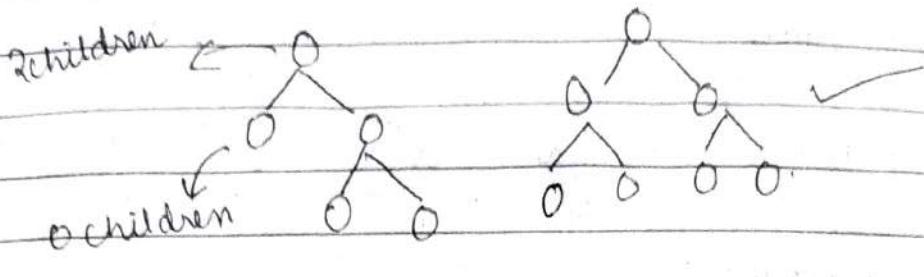
Ancestor  $\rightarrow$  All the parents are called ancestor



#### Type of BT

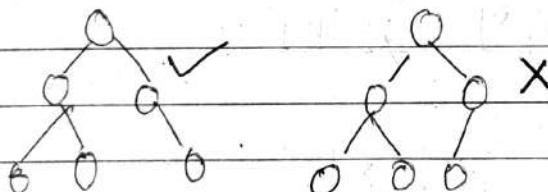
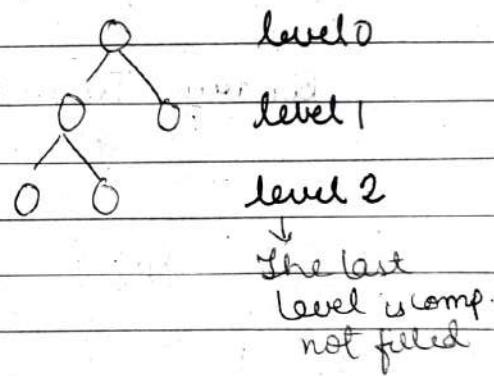
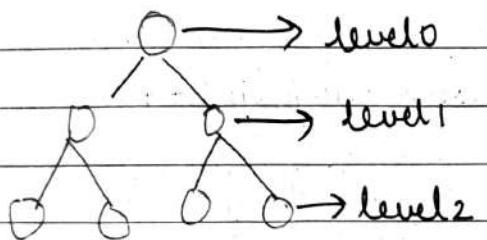


Full BT - Every node has 0 or 2 children

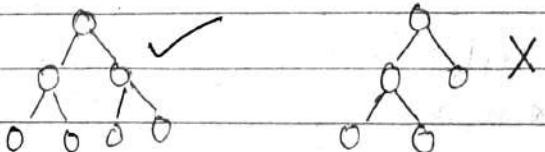


Complete BT - All levels are completely filled except the last level.

- i) The last level has all nodes on <sup>as</sup> left as possible.



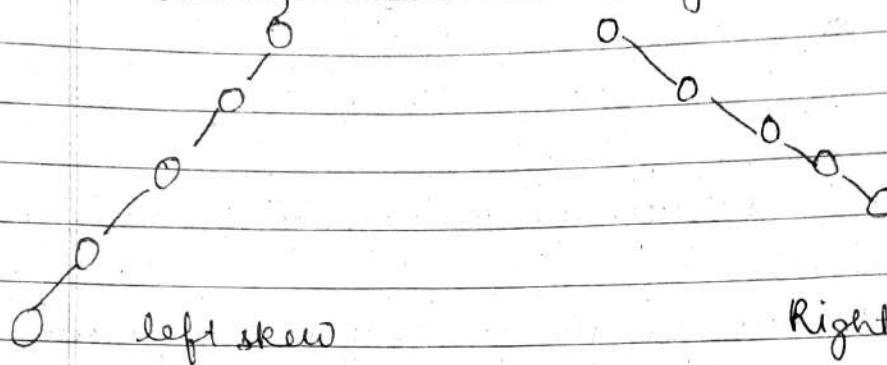
Perfect BT - All leaf nodes are at the same level.



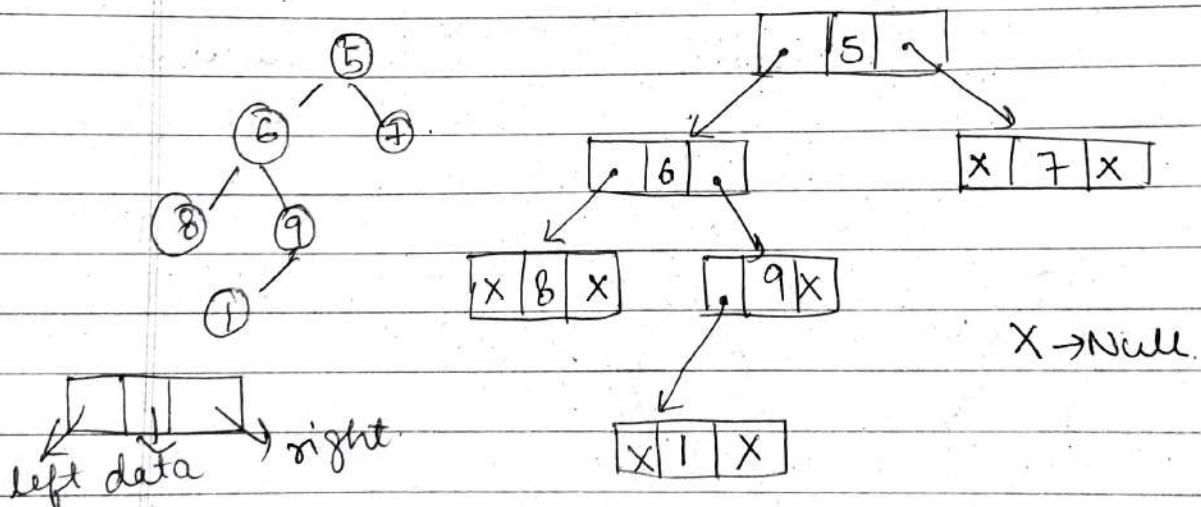
Balanced BT  $\rightarrow$  height of tree is at max  $\log(N)$   
 $\downarrow$   
 nodes.

Let  $N = 8$        $\log_2 N = \log_2 8 = 3$

Degenerate BT  $\rightarrow$  when it is a skew tree.  
 Every node has single children.



### Binary Tree Representation in C++



Struct Node {

    int data;

    Struct Node\* left;

    Struct Node\* right;

}

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

```
int main(){
```

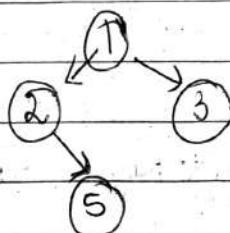
```
Struct Node* root = new Node(1);
```

```
root->left = new Node(2);
```

```
root->right = new Node(3);
```

```
root->left->right = new Node(5);
```

```
}
```

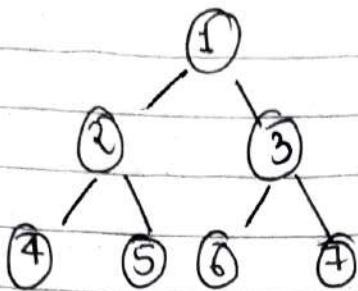


## Binary Tree traversals in BT (BFS|DFS)

To solve any tree problem we tend to traverse the tree. So to solve it there are basically couple of techniques

- ① DFS (Inorder, Preorder, Postorder)
- ② BFS (level order)

1. DFS (Depth first search) → Traverses depth wise



1) Inorder (left root right)

4 2 5 1 6 3 7

2) Pre order traversal (root left right)

1 2 4 5 3 6 7

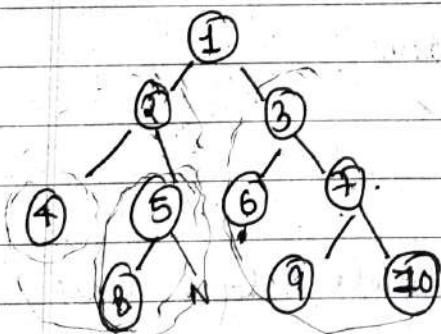
3) Post order traversal (left right root)

4 5 2 6 3 1

Pre → Root at first

Post → Root at last

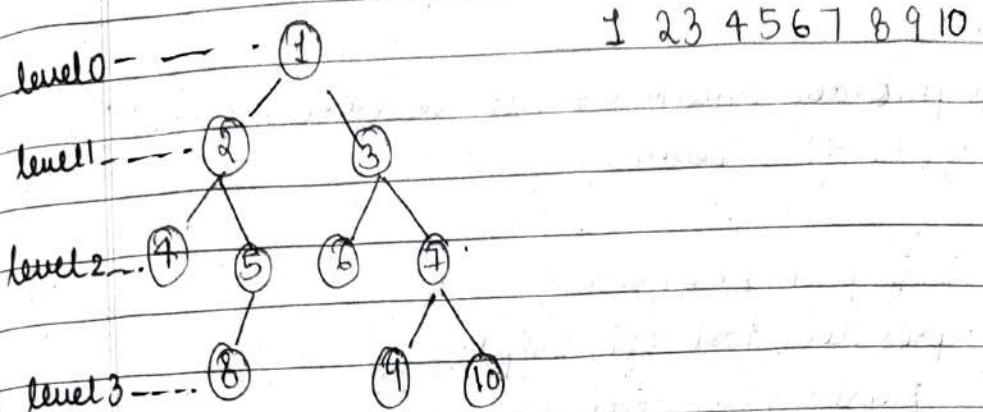
In → Root in between



Inorder - (L R O R I)

4 2 8 5 1 6 3 9 7 10

ii) BFS (Breadth first search) → Travels levels wise



Pre order traversal (Root, left, Right)

```
void preorder( node){
```

```
    if (node == null)  
        return;
```

```
}
```

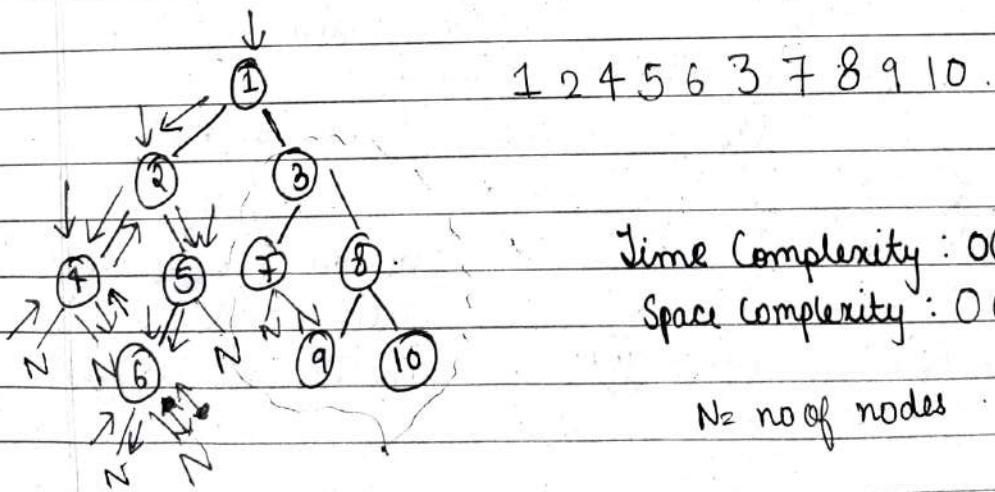
```
print( node-> data);
```

~~preorder~~

```
preorder( node-> left);
```

```
preorder( node-> right);
```

```
}
```

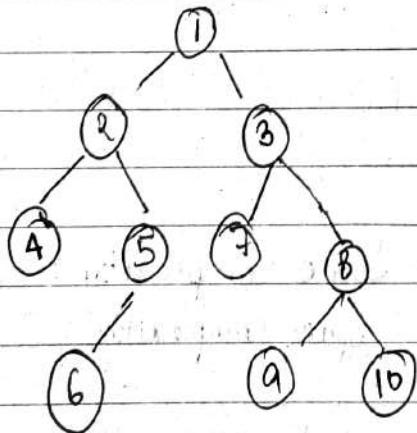


• Code

```
void preorder (TreeNode* root, vector<int>& temp){  
    if (!root) return;  
  
    temp.push_back (root->val);  
    preorder (root->left, temp);  
    preorder (root->right, temp);  
}
```

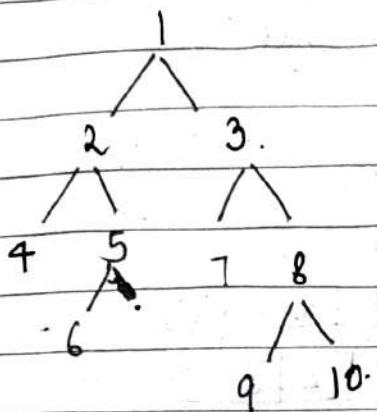
```
vector<int> preOrderTraversal (TreeNode* root)  
{  
    vector<int> temp;  
    preorder (root, temp);  
    return temp;  
}
```

Inorder traversal (left <sup>Root</sup> Right)



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

## Post Order Traversal of Binary Tree



```
void postorder(node)
```

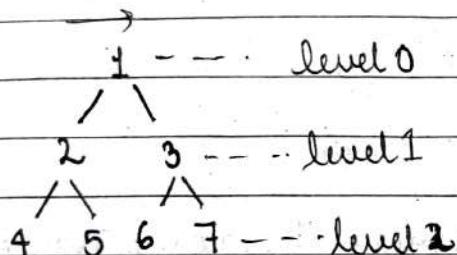
```
{
```

```
if (node == null)  
    return;
```

```
postorder (node->left);  
postorder (node->right);  
print (node->data);
```

```
}
```

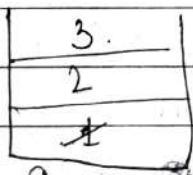
## Level Order traversal (BFS)



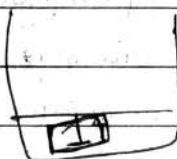
```
1  
2 3  
4 5 6 7
```

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

We would require a queue ds. It would be having queue ds.

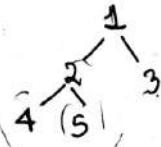


Queue



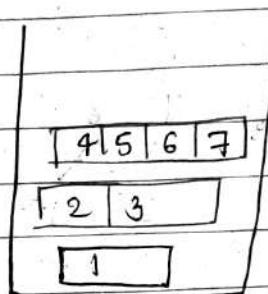
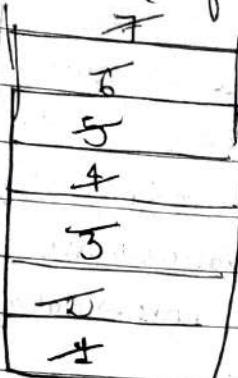
vector<vector<int>>

If the left exists put it into queue. then check if right exists & put it into queue



Pre : 1 2 4 5 3. (R<sub>0</sub>, L, R)  
 In : 4 2 5 1 3 (L R<sub>0</sub> R)  
 Post : 4 5 2 3 1 (L, R, R<sub>0</sub>)

In the next iteration whatever is present in the queue take off. Now onto the left check left & right & put it into queue.



Queue .

taking  
out of  
queue

1

2 3

4 5 6 7

vector<vector<int>>  
(level order  
traversal  
stored)

```
vector<vector<int>> levelOrder (TreeNode* root) {
    vector<vector<int>> ans;
    if (!root) 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->val);
        }
    }
}
```

```
ans.push_back(level);
```

```
}
```

```
return ans;
```

```
}
```

## Iterative Preorder Traversal

Root left Right

1  
/\

2 7

1 2 3 4 5 6 7

/ \

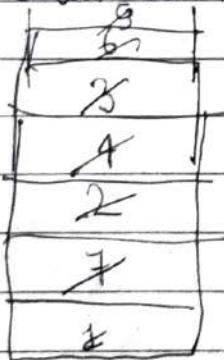
3 4

We would be doing it using stack

/ \

5 6

Take a stack (LIFO) DS. Whatever is at the root put it in stack. At first whatever is at the top put it out. Then ~~put~~ out right & then the left node.



1 2 3 4 5 6 7

stack becomes empty  $\rightarrow$  Entire pre order

traversal is printed

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

```
{
```

```
    vector<int> preorder;
```

```
    if (!root) return preorder;
```

```
    stack<TreeNode*> st;
```

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

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

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

```
        st.pop ();
```

```
        preorder.push_back (root->val);
```

```
        if (root->right != NULL) st.push (root->right);
```

```
        if (root->left != NULL) st.push (root->left);
```

```
}
```

```
return preorder;
```

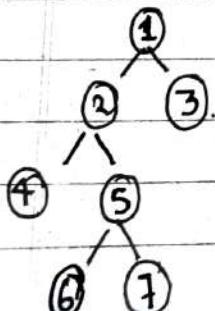
```
}
```

TC: O(N)

SC: O(N) or O(Height of BT)

### Iterative Inorder Traversal

left Root Right

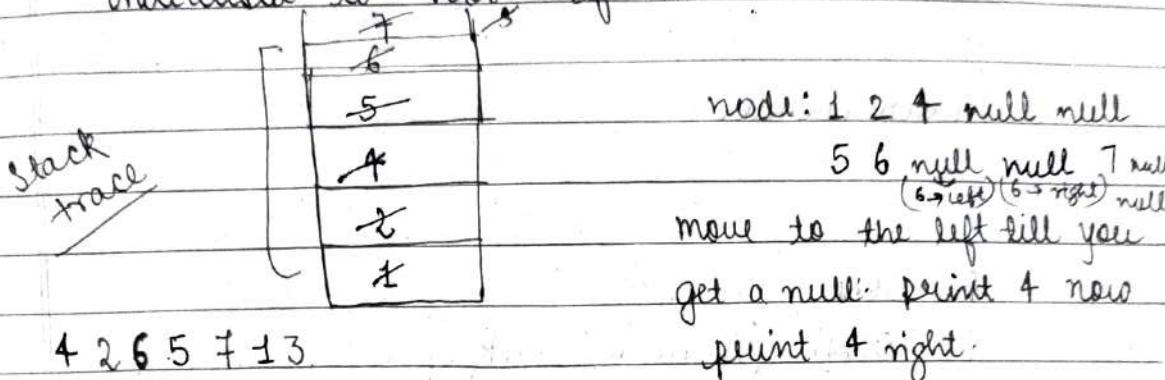


• 4 2 6 5 7 1 3

In recursion: left <sub>root</sub> point right

4
2
1

we would use the Stack (LIFO) • Have a node initialised to root if  $\text{node} = \text{null}$



node: 1 2 4 N N 5 6 N N 7 N N 3 N N

when we see null whatever is there we print.

```
vector<int> InorderTraversal(TreeNode *root) {
```

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

~~if (root == null)~~

```
    TreeNode *node = root;
```

```
    vector<int> inorder;
```

```
    while (true) {
```

```
        if (node != NULL) {
```

```
            st.push(node);
```

```
            node = node->left;
```

```
}
```

```

else {
    if (st.empty() == true) break;
    node = st.top();
    st.pop();
    inorder.push_back(node->val);
    node = node->right;
}
return inorder;
}.

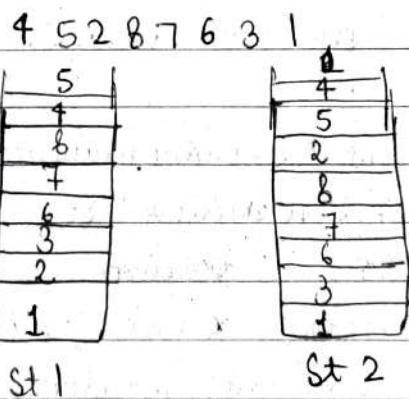
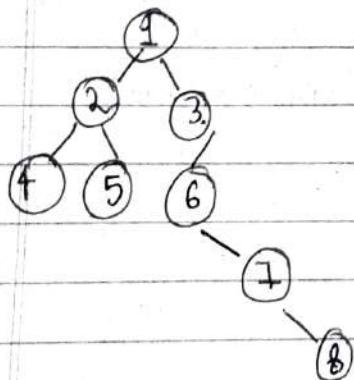
```

TC - O(N)

SC - O(N) (Auxiliary stack space)

Iterative postorder . (left Right Root)

Using 2 stack



4 5 2 8 7 6 3 1

Take two stacks. In first stack push root now take out root push the root->left & then root->right . & do it vice versa.

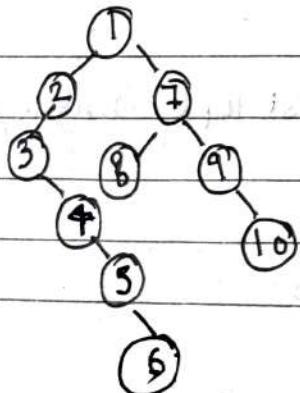
```

vector<int> postorderTraversal(TreeNode* root) {
    vector<int> postorder;
    if (!root) return postorder;
    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()) {
        postorder.push_back(st2.top()->val);
        st2.pop();
    }
    return postorder;
}

```

$T C \rightarrow O(N)$  (Traversing for every nodes)  
 $S C \rightarrow O(2N)$

Using 1 stack (left Right Root)



Revison: left Right point

We will use an external stack.

Take a curv called current & point to the root of the node.

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

postorder vec : 6 5 4 3 2 8 7 1



temp = st.top()  $\rightarrow$  right

while ( curr != null || !st.isEmpty() ) {

if ( curr == null)

st.push(curr);

curr  $\rightarrow$  left;

else

temp = st.top()  $\rightarrow$  right;

if (temp == null)

temp = st.top();

st.pop();

postorder(temp);

while (!st.empty && temp == st.top()  $\rightarrow$  right)

temp = st.top();

st.pop();

postorder(temp  $\rightarrow$  val);

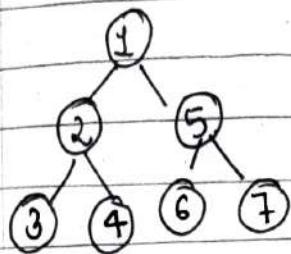
else

curr = temp;

T.C  $\rightarrow O(2N)$

SC  $\rightarrow O(N)$

Preorder, Inorder, Postorder (In one video)



A single stack would be used, it would be storing (node, num)

(R, L, R) Preorder list - 1 2 3 4 5 6 7

(L, R, R) Inorder list - 3 2 4 1 6 5 7

(L, R, R) Postorder list - 3 4 2 6 7 5 1

x	$(7, x)^{x^3}$
x	$(6, +)^{x^3}$
x	$(5, +)^{x^3}$
x	$(4, +)^{x^3}$
x	$(3, +)^{x^3}$
x	$(2, +)^{x^3}$
x	$(1, +)^{x^3}$

if num = 1

pre-order

++ (push that num)

!left enter left

if num = 2

Inorder

++

!right push right

if num = 3

postorder

TC: O(3N) } linear TC  
SC: O(4N) }

```
vector<int> preInPost(TreeNode* root) {  
    Stack<pair<TreeNode*, int>> st;  
    st.push({root, 1});
```

```
vector<int> pre, in, post;  
if (root == null) return;
```

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

|| this is a part of pre. Increment 1 to 2 & push the left side of BT

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

```
if (it.first->left != NULL) st.push({it.first->left, 1});
```

|| this is a part of in. Increment 2 to 3 & push the right side of BT

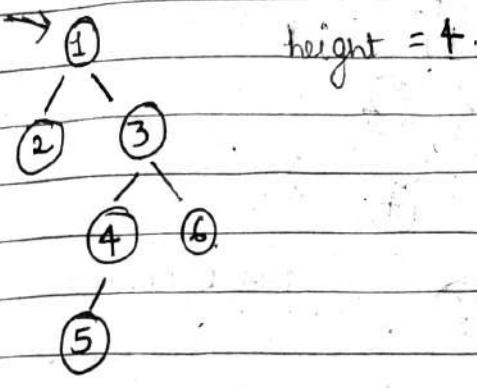
```
else if (it.second == 2)  
    in.push_back(it.first->val);  
    it.second++;  
    st.push(it);
```

```
if (it.first->right != NULL) st.push({it.second->right, 3});
```

|| dont push it back again

```
else post.push_back(it.first->val);
```

## Maximum Depth in BT (Height of BT)



Recursive | level order

Auxiliary  
space

$SC \rightarrow O(N)$   
 $O(\text{height})$

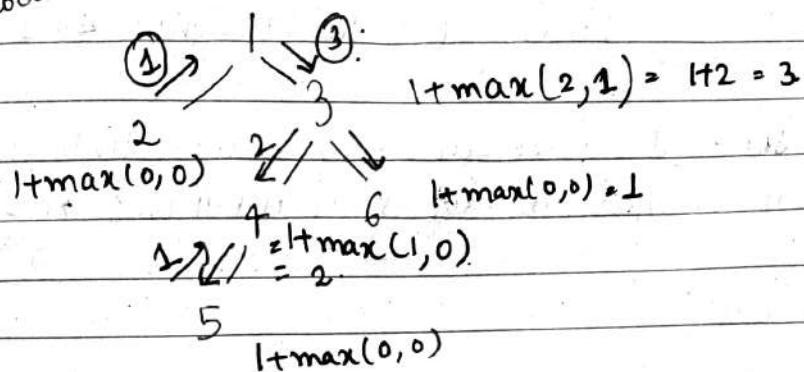
~~stack space~~  
level order as we  
use queue.

$SC \rightarrow O(N)$

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

for the  
current  
node

$$1 + \max(3, 1) = 1 + 3 = 4$$



```
int maxHeight (Treenode* root) {
    if (root == NULL) return 0;
```

```
    int lh = maxHeight (root -> left);
```

```
    int rh = maxHeight (root -> right);
```

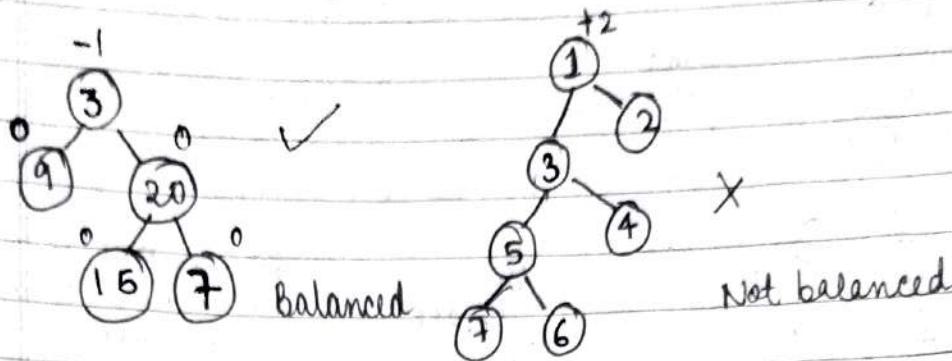
```
    return 1 + max (lh, rh);
```

}

TC -  $O(N)$

SC -  $O(N)$

## Check for Balanced Tree



(1)

(2)

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

Naive sol - Traverse for every node: find lh  
 find rh: If their abs diff is greater than 1 return false.

pseudocode

Bool check (Node)

if node == NULL

return true;

$lh = \text{findHLeft} (\text{node} \rightarrow \text{left});$  J ocn

$rh = \text{findHRight} (\text{node} \rightarrow \text{right}),$

if ( $|\text{abs}(rh - lh)| > 1$ ) return false;

bool left = check (node  $\rightarrow$  left);

bool right = check (node  $\rightarrow$  right);

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

return true;

$$TC \rightarrow O(N) \times O(N) \approx O(N^2)$$

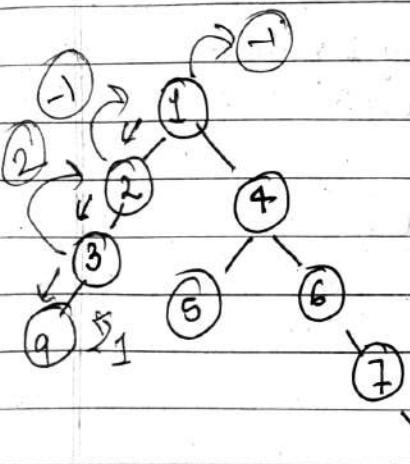
↓                    ↓  
traversal      for every height

if balanced

Yes

return  
height

return -1



```
int check (TreeNode* root) {  
    if (!root) return 0;
```

```
    int lh = check (node->left);  
    int rh = check (node->right);  
    if (!lh || !rh) return -1;  
    if (abs(lh - rh) > 1) return -1;
```

return max (lh, rh);

```
bool isBalanced (TreeNode* root) {
```

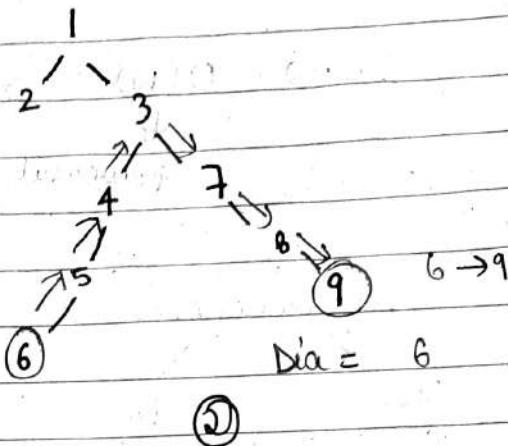
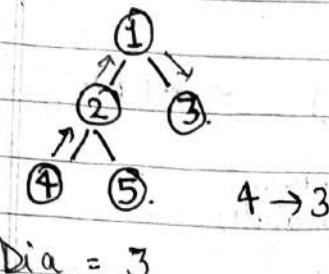
```
    return check (root) != -1;
```

}

TC -  $O(N)$

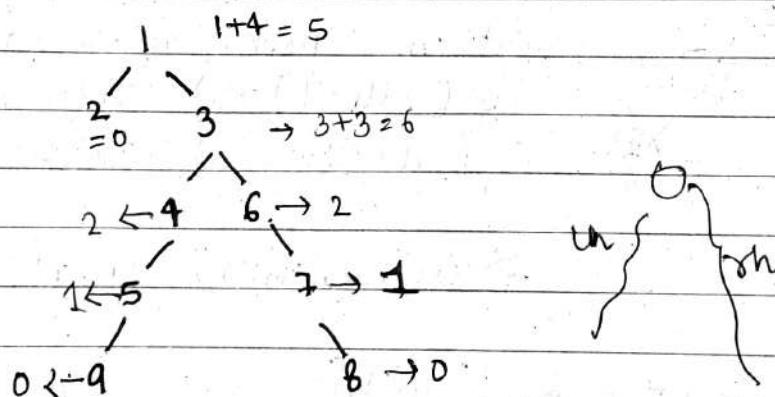
SC -  $O(N)$  Auxiliary space

## Diameter of Binary Tree



Diameter = longest path between any 2 nodes  
path doesn't need to pass via root.

Name: Traverse the entire node



$$\max = \max(h_l + h_r, \max)$$

find Max(node)  
{

if (root == null)  
return;

$lh = \text{findLeftH}(\text{node} \rightarrow \text{left})$

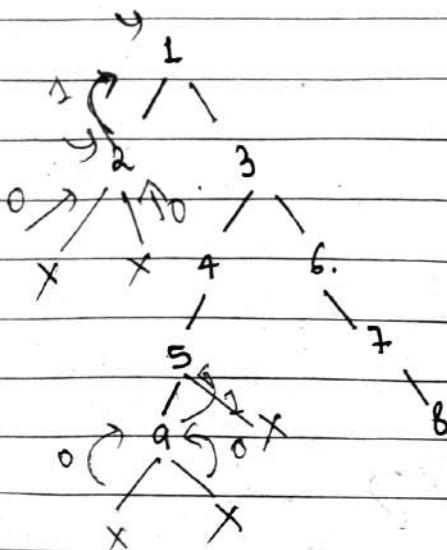
$rh = \text{findRightH}(\text{node} \rightarrow \text{right})$

$\max = \max (\max, lh + rh);$

$\cdot \text{findMax}(\text{node} \rightarrow \text{left})$   
 $\cdot \text{findMax}(\text{node} \rightarrow \text{right})$   
 }

$$T.C = O(N^2)$$

Optimised Approach - using height of BT



```

int diameter (Tree* node * root) {
    int dia = 0;
    height (root, dia);
    return dia;
}
  
```

```

int height (Tree* node, int & dia)
if (!node) return 0;
  
```

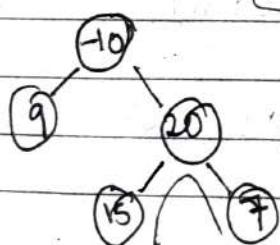
TC  $\rightarrow O(N)$

SC  $\rightarrow O(N)$

```

int lh = height (node → left, dia);
int rh = height (node → right, dia);
dia = max (dia, lh + rh);
return 1 + max (lh, rh);
}
  
```

maximum path sum in BT



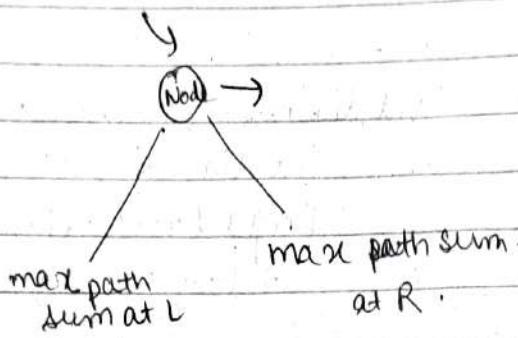
$\rightarrow \text{node A} \rightarrow \text{node B}$  (in a path  
 a single node  
 only appear  
 once)

15	20	7	$(15 \rightarrow 7)$
15	20	-10	$(15 \rightarrow 9)$

Out of all the path, find where you get the maximum sum.

Brute force: Try out every possible combination of A & B. whichever path is giving the max sum is the max.

Optimal Approach: find the max<sup>m</sup> height of BT and find the diameter



$$\Rightarrow \text{val} + (\text{maxL} + \text{maxR}) \rightarrow \text{maximum}$$

```
int maxPath(node, maxi)
```

```
{
```

```
    if (node == null) return 0;
```

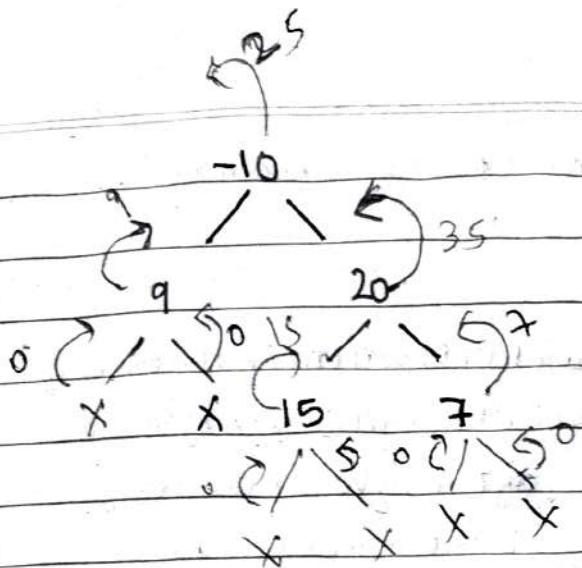
```
    ls = maxPath(node->left, maxi);
```

```
    rs = maxPath(node->right, maxi);
```

```
    maxi = max(maxi, ls+rs+node->val);
```

```
    return (node->val) + max(ls, rs);
```

```
}
```



~~maxi = 9 + 15 + 42~~

$$= 15 + 7 + 20$$

$$= \underline{42}$$

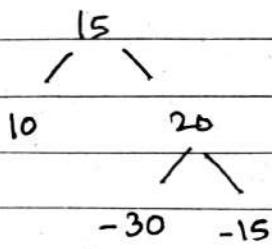
$$\text{max} = 20 + \max(15, 7)$$

$$\text{return} = \begin{aligned} & 20 + 15 \\ & \underline{20 + 15} \\ & = 35 \end{aligned}$$

$$\begin{aligned} \text{max} &= 9 + 35 - 10 \\ &= 25 + 9 \\ &= \underline{34} \end{aligned}$$

$$\begin{aligned} \text{return} &= -10 + \max(9, 35) \\ &= -10 + 35 \\ &= \underline{25} \end{aligned}$$

max path in some cases.



→ for this tc don't consider a negative path sum.

so ignore & make it zero.

```
int maxPathSum (TreeNode *root) {
    int maxi = INT_MIN;
    maxPathDown (root, maxi);
    return maxi;
}
```

```

int maxPathDown (TreeNode* node, int &maxi) {
    if (!node) return 0;
    int left = max (0, maxPathDown (node->left, maxi));
    int right = max (0, maxPathDown (node->right, maxi));
    maxi = max (maxi, left + right + node->val);
    return max (left, right) + node->val;
}

```

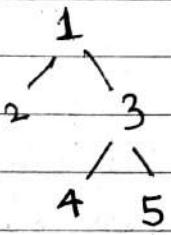
T.C  $\rightarrow O(N)$

S.C  $\rightarrow O(N)$  Auxiliary space.

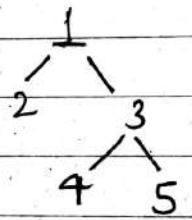
Check if 2 trees are identical or not

TC  $\rightarrow O(N)$

SC  $\rightarrow O(N)$



Tree 1



Tree 2

preorder - Root, Left, Right

Do traversal & see if the traversal is same or not.

bool isSame (TreeNode\* p, TreeNode\* q) {

if ( $p == \text{NULL}$  ||  $q == \text{NULL}$ ) {

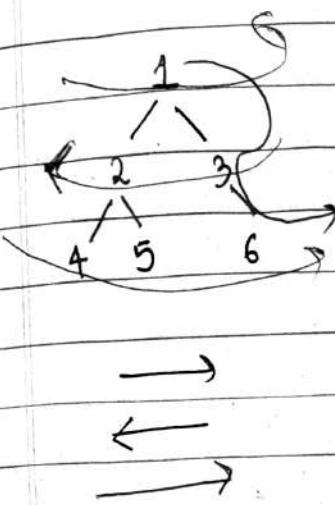
return ( $p == q$ );

}

return ( $p->\text{val} == q->\text{val}$ ) && isSame ( $p->\text{left}$ ,  $q->\text{left}$ )  
&& isSame ( $p->\text{right}$ ,  $q->\text{right}$ );

}

## Zig Zag or spiral traversal



1 3 2 4 5 6.

Take a queue ds & store

root	6
	5
	4
	3
	2
	1

Queue. 1  
2 3  
4 5 6

We introduce a flag variable.

flag = 0

if put 1 in queue first ie. root

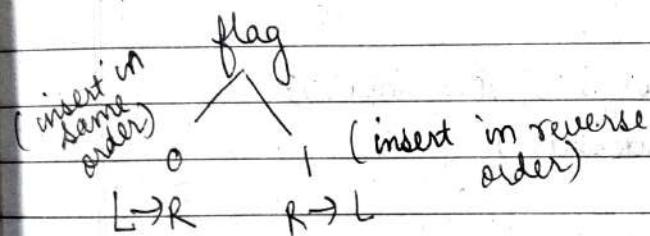
new put 1 out if left exists

push left & flag change the

same.

4   5   6	→ flag = 0
3   2	→ flag = 1
1	→ flag = 0

vector<vector<int>>



After pushing in vector update flag to 1.

```
vector<vector<int>> zigzag (TreeNode* root) {
```

```
    vector<vector<int>> result;
```

```
    if (!root) return result;
```

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

```
    q.push(root);
```

```
    int flag = 1;
```

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

```
        int size = q.size();
```

```
        vector<int> row(size);
```

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

```
            TreeNode* node = q.front();
```

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

```
            row.push_back(node->val);
```

```
            if (node->left) q.push(node->left);
```

```
            if (node->right) q.push(node->right);
```

```
}
```

```
        if (flag == 0) {
```

```
            reverse(row.begin(), row.end());
```

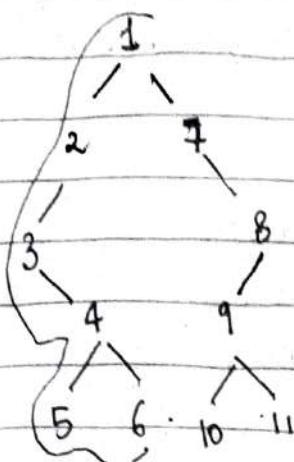
```
            result.push_back(row);
```

```
        } flag = !flag;
```

```
    }
```

```
    return result;
```

Boundary Traversal



Anticlockwise boundary traversal

1 2 3 4 5 6

- 1) Take the left boundary excluding leaf node.
- 2) Take leaf node
- 3) Take right boundary in the reverse direction excluding leaf node.

Take the ds which stores the data.

4	
3	
2	
1	
ds	
7	
8	
9	
11	
10	
6	
5	
4	
3	
2	
1	

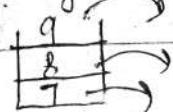
put on left if left doesn't exist put right. whenever you encounter leaf node stop left boundary traversal.

For leaf node, use inorder traversal (Root, left right)

For right boundary use a

vector or stack

If right doesn't exist go to the left



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

```
void addRightBoundary (Node* root, vector<int>& res)
{
    vector<int> temp;
    Node* curr = root->right;
    while (curr) {
        if (isLeaf(curr))
            temp.push_back (curr->data);
        if (curr->right) right;
        curr=curr->right;
        if else
            curr=curr->left;
    }
    res.push_back (reverse (temp.begin(), temp.end()));
}
```

```
void addLeave (Node* root, vector<int>& res) {
    if (!isLeaf(root))
        res.push_back (root->data)
    return;
    if (root->left) addLeave (root->left, res);
    if (root->right) addLeave (root->right, res);
}
```

vector<int> pointBoundary (Node\* root)

{

vector<int> res;

if (!root) return res;

if (!isleaf (root)) res.push\_back (root->data);

addleftboundary (root, res);

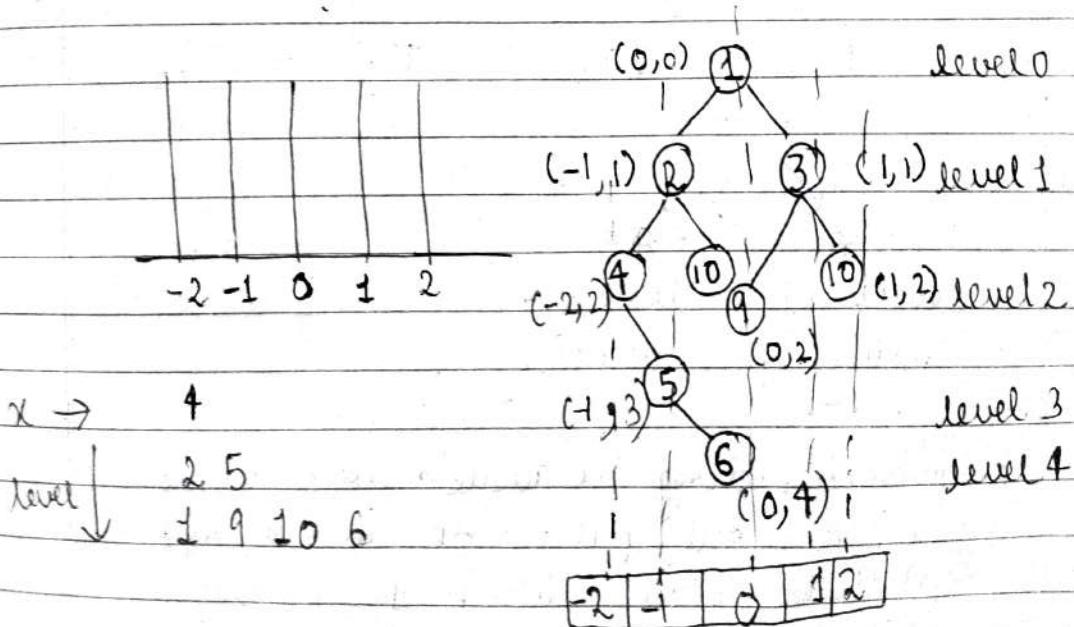
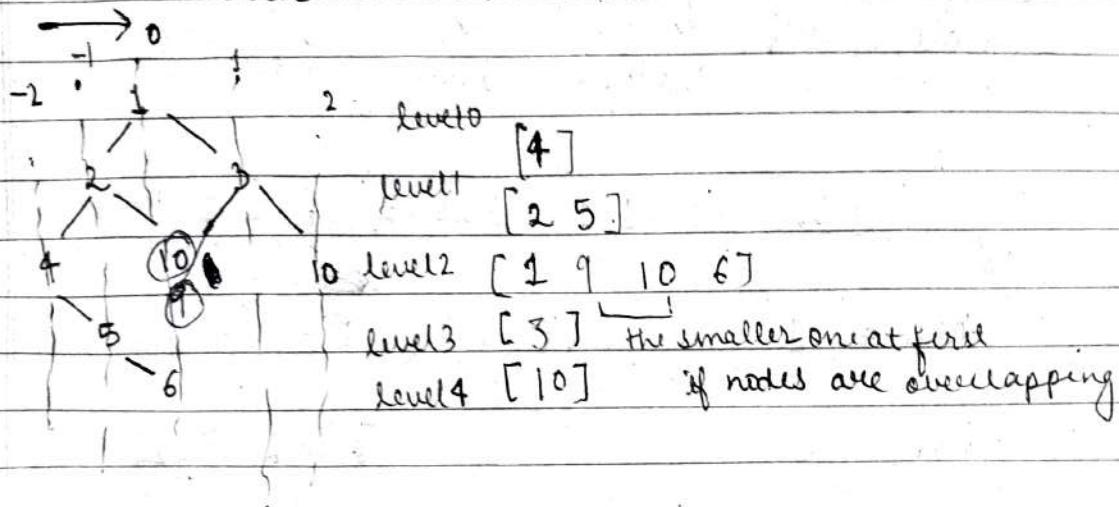
addleaves (root, res);

addrighboundary (root, res);

return res;

}

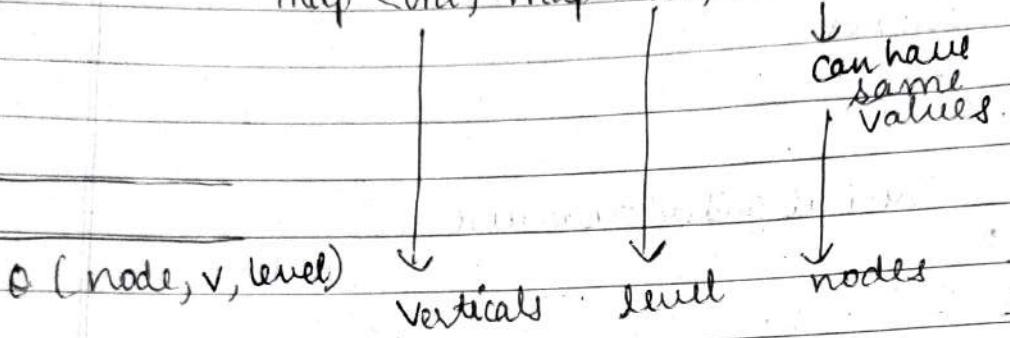
### Vertical Order Traversal



Level order traversal to move left & right  
 Take a queue ds. In this queue ds store  
 the node vertical & level

Carry a map, on every vertical there  
 would be multiple nodes.

$\text{map } \langle \text{int}, \text{map } \langle \text{int}, \text{multiset } \langle \text{int} \rangle \rangle$



$(1, 0, 0) (2, -1, 1) (3, 1, 1) (4, -2, 2) (10, 0, 2) (9, 0, 2)$

Q  
 $\text{node} = 1$        $(0, 0)$       ~~l~~  $0 \rightarrow 0 \rightarrow \{1\}$

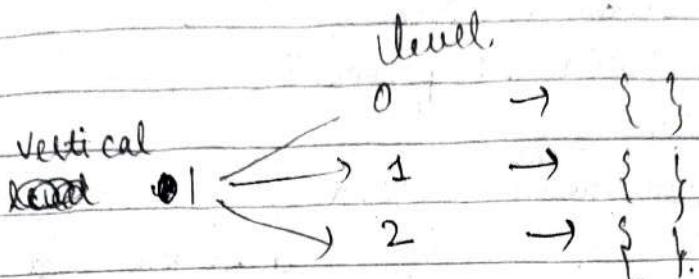
$\text{node} = 2$        $(-1, 1)$       ~~l~~  $0 \rightarrow 1 \rightarrow \{2\}$

$\text{node} = 3$        $(1, 1)$        $0 \rightarrow 1 \rightarrow \{3\}$

node  
 left      right  
 level by +1      level by +1  
 vertical by -1      vertical by +1

keep on iterating the queue. Every time  
 you pop it out put it into vertical after  
 that level after that in the multiset.

$\rightarrow$  p. first       $\rightarrow$  p. second  
 map < int, map < int, multiset > nodes  
 ↓                  ↓                  ↓  
 vertical      level      node.



vector <vector <int>> verticalTraversal(TreeNode\* root);

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

queue < TreeNode\*, pair < int, int > > q;

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

while (!q.empty());

~~Temporary~~.

auto p = q.front();

q.pop();

TreeNode\* node = p.first;

int x = p.second.first;

int y = p.second.second;

nodes[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}});

}

vector<vector<int>> ans;

for(auto p : nodes)

vector<int> col;

for(auto q : p.second) {

col.insert(col.end(), q.second.begin(),

q.second.end());

}

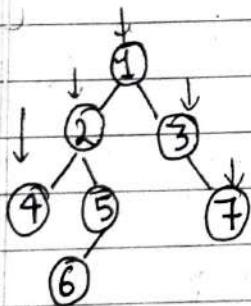
ans.push\_back(col);

}

return ans;

}

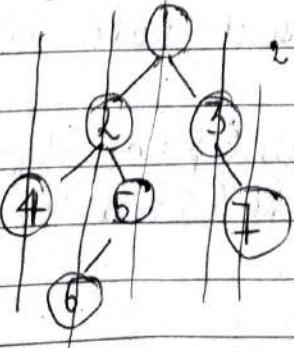
Top view of Binary Tree



4 2 1 3 7

To solve any BT ques we use traversal. We use level order traversal.

-2 -1 0 1



We require a queue

(4, -2) (5, 0)

(3, 1)

nodebt, 2

(2, -1)

0, -1

(1, 0)

(line, node)  
map

↓  
node line

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

map

map →  
duplicate  
elements  
is deleted

Queen (root, level)

node = 1, 2, 3, 4, 5, 7, 6

$$\text{cline} = 0, -1+1, -2, 0, 1$$

0	-1	1	-2	0	2	-1	6
1	2	3	4	5	7	6	0

```

vector<int> topview(reenode* root) {
    vector<int> ans;
    if(!root) return ans;
    map<int, int> mpp;
    queue<reenode*> q;
    q.push({root, {0, 0}});

```

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

~~The modern French~~

auto it  
q.pop();

```
auto it = q.front();
```

```
int line = it.second;
```

Free node \* node = it. first;

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

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

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

```
}
```

```
for (auto it : mpp)
```

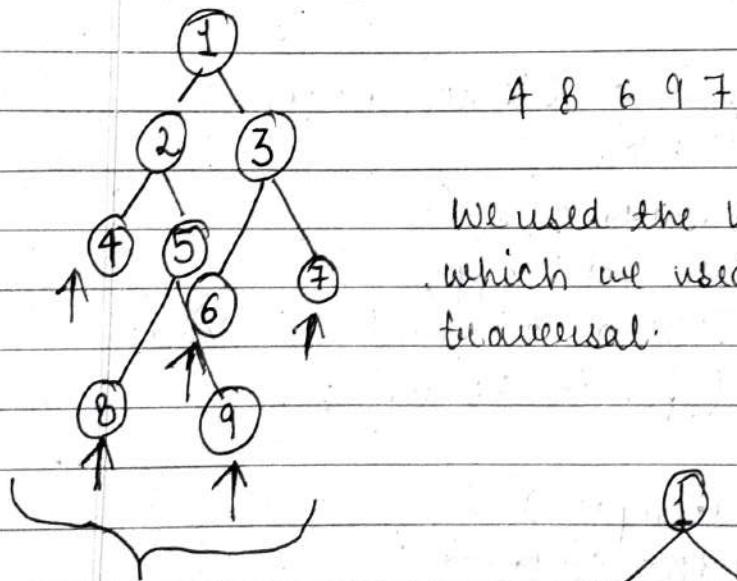
```
    ans.push_back(it.second)
```

```
}
```

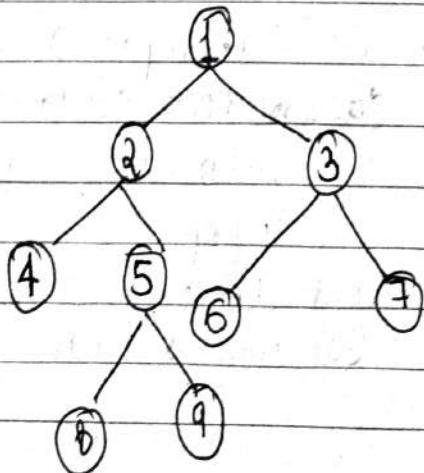
```
return ans;
```

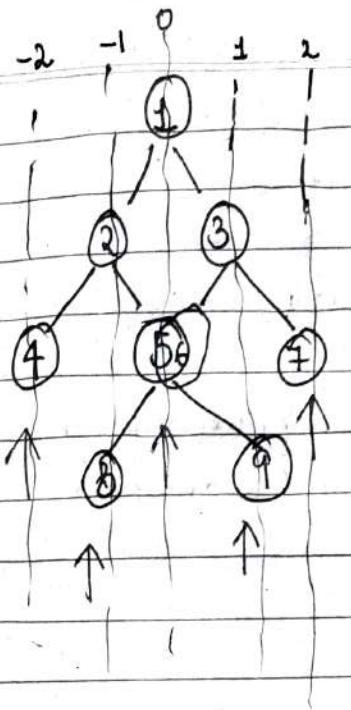
```
}
```

Bottom view of BT



We used the line concept  
which we used in vertical order  
traversal.





(9, 1)
(8, -1)
(7, 2)
(6, 0)
(5, 0)
(4, -2)
(3, 1)
(2, -1)
(1, 0)
(0, 2)

Queue.

map (line, node)

replace (5, 0) with (4, 0)

node = 2 3 4 5 6 7 8 9  
0 1 +1 -2 0 0 2 1 +1

-2	-1	0	1	2	
4	8	6	9	7	

whatever line you are getting replace it with the node.

In top view we first check if the element with given line is not there, then we insert the value.

```
vector<int> bottomView( Node* root) {
```

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

```
    if (!root) return ans;
```

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

```
    queue<Treenode*, int> q;
```

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

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

```
        auto it = q.front();
```

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

```
Node* node = it.first;
int line = it.second;
```

```
mpp[line] = node->data;
```

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

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

```
return ans;
```

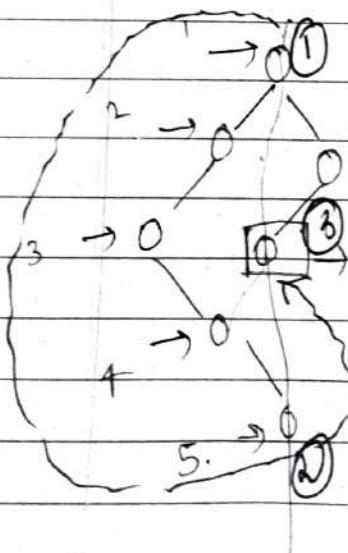
TC → O(N)  
SC → O(N)

Q. Will a Recursive traversal work?

Ans: No it won't work with

(left, line + 1)

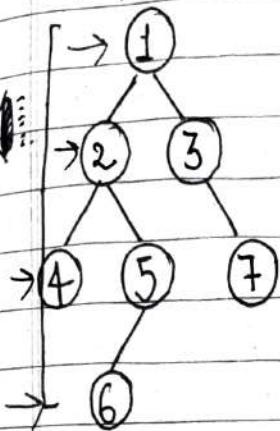
(right, line + 1)



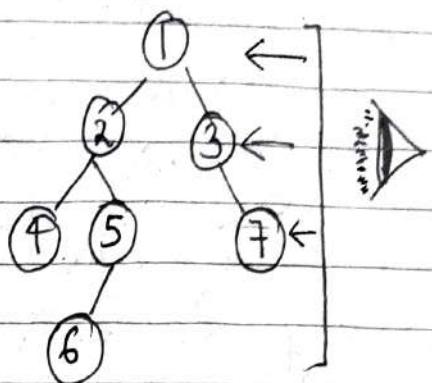
at the end this node  
would be visited  
in inorder traversal

## Right / left view of Binary

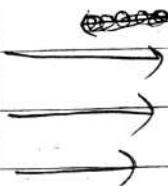
left view



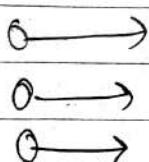
Right view



left view: 1 2 4 6

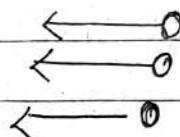


first node of every level in



right view: 1 3 7

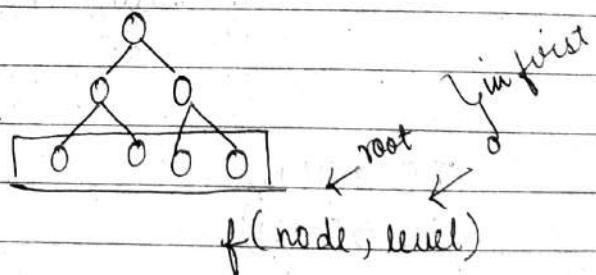
first node of every level



SC TC

$O(H)$   $O(N)$  ① Recursive traversal  
 $O(N)$   $O(N)$  ② Iterative traversal.

We wont use level order traversal



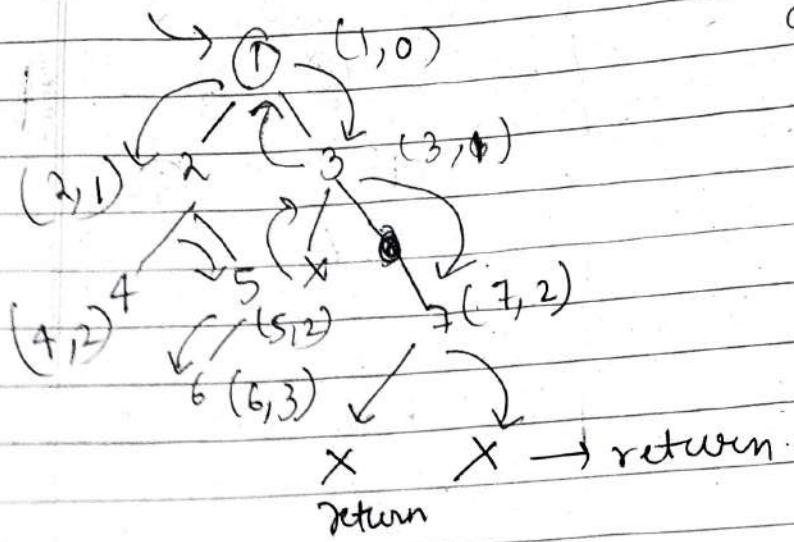
we would Reverse  
preorder traversal (Root Left Right)  
(Root Right Left)

Right view of binary tree

```
if (level == ds.size())
    ds.add(node)
```

6	7	3
7		2
3		1
1		0

ds.



left side

```
f (node, level)
```

```
if (node == null) return;
```

```
if (level == ds.size())
    ds.add(node)
```

```
if (node->left, level+1)
    if (node->right, level+1)
```

TC  $\rightarrow O(N)$

SC  $\rightarrow O(H)$

```
vector<int> rightSideView (Node *root) {
```

```
    vector<int> res;
```

```
    recursion (root, 0, res);
```

```
    return res;
```

```
}
```

```
void recursion (Node *root, int level, vector<int>&res)
```

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

```
if (res.size () == level) res.push_back (root->val),
```

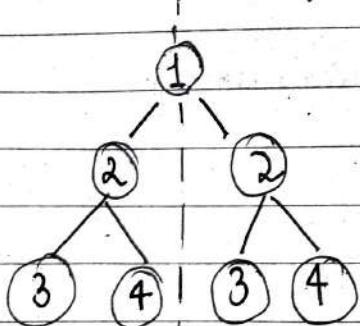
```
recursion (root->right, level+1, res),
```

```
recursion (root->left, level+1, res);
```

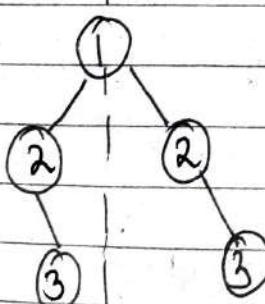
```
}
```

Check for Symmetrical Binary Trees.

(whether it forms mirror  
of itself along the  
center or not)



it forms mirror  
of itself around its center



left  $\Rightarrow$  right

Root  $\rightarrow$  left  $\nparallel$  Root  $\rightarrow$  Right

| not symmetric around its  
center

Root → left

Preorder Root Left Right

Root → right

Root Right Left

When we do a simultaneous traversal on both the trees nodes would be marked.

```
bool isSymmetrical(TreeNode *root) {
```

```
    if (!root) return true;
```

```
    return helper(root→left, root→right);
```

```
}
```

```
bool helper(TreeNode *p, TreeNode *q) {
```

```
    if (!p && !q) {
```

```
        return true;
```

```
}
```

```
    else if (!p || !q) {
```

```
        return false;
```

```
}
```

```
    if (p→val != q→val) {
```

```
        return false;
```

```
}
```

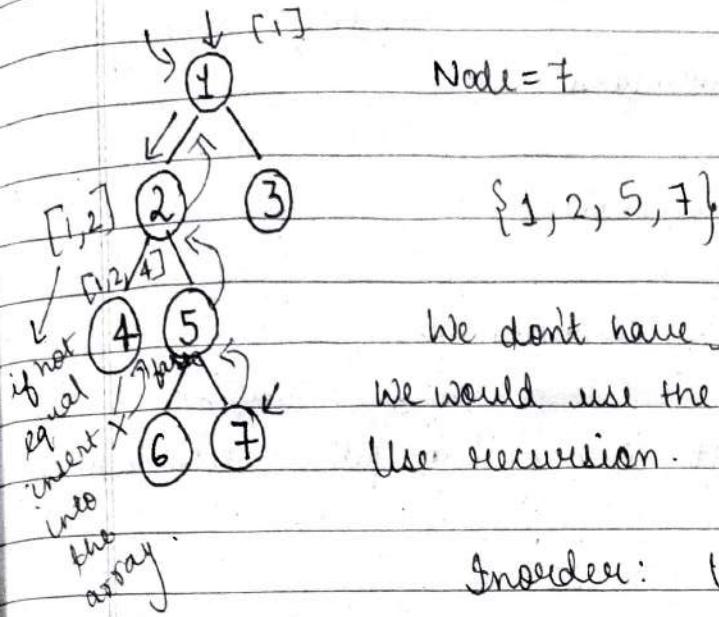
```
    return helper(p→left, q→right);
```

```
    p→right, q→left)
```

```
&& helper(p→right, q→left);
```

```
}
```

Point Root to Node path. (Root to leaf path)



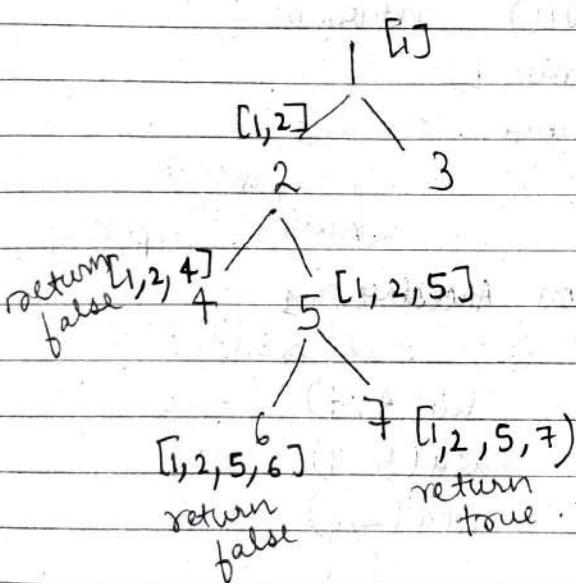
We don't have the parent node.

We would use the inorder traversal  
Use recursion.

Inorder: left Root Right

false (left  $\rightleftarrows$ )  
false (right  $\rightleftarrows$ ) } (if any of them give true go back)

When you reach 4 you encounter false in the left as well as in the right. So you return. Now when you return don't forget to remove 4.



In question it. says the value always exists

```

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

```

vector<int> Solution::solve (TreeNode\* A, int B) {  
 vector<int> arr;

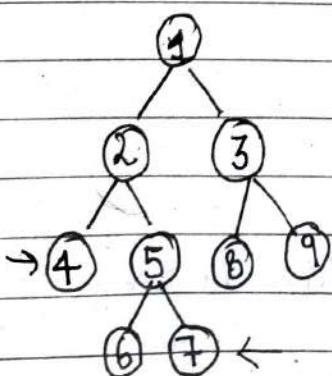
```

    if (A == NULL) return arr;
    getpath (A, arr, B);
    return arr;
}

```

ancestor  
 (node at the deepest level)

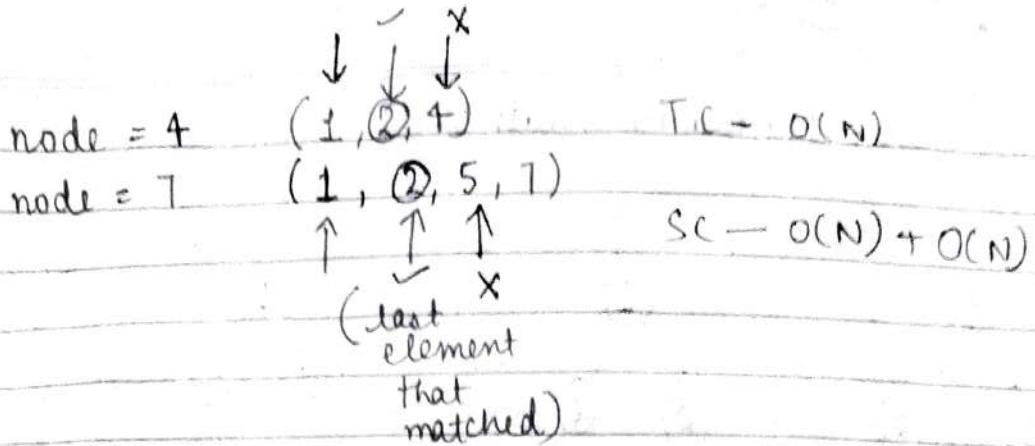
lowest common ancestor



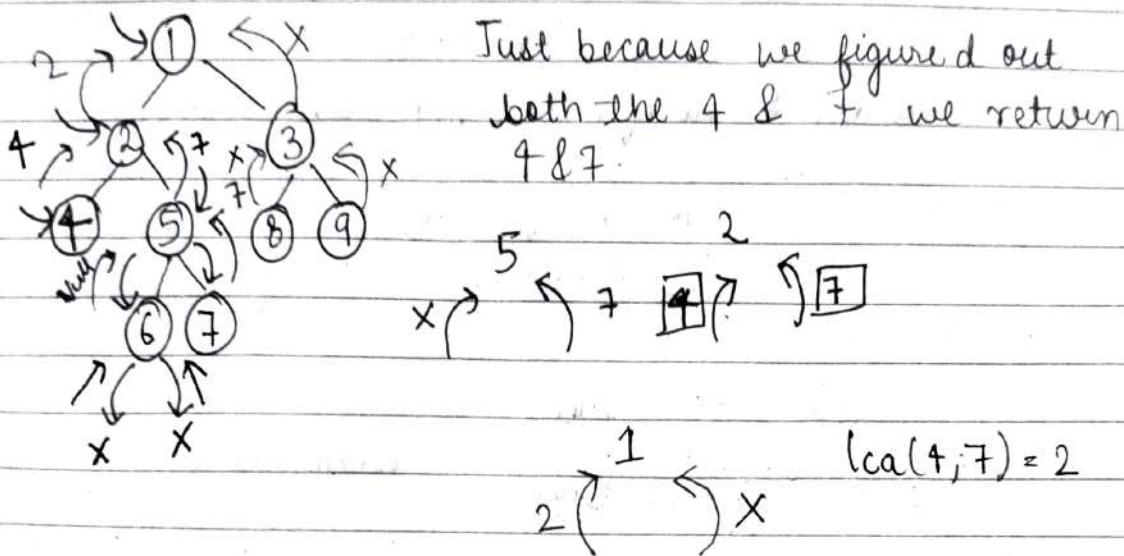
$$\begin{aligned}
 lca(4, 7) &= 2 \\
 lca(5, 9) &= 1 \\
 lca(2, 6) &= 2
 \end{aligned}$$

$$lca(2, 1) = 1$$

$$lca(5, 2) = 1$$



$\text{lca}(4, 7) = 2$  (Recursive traversal i.e dfs traversal)



Node \* lowestCommonAncestor (Node \* root, Node \* p, Node \* q);

// base case

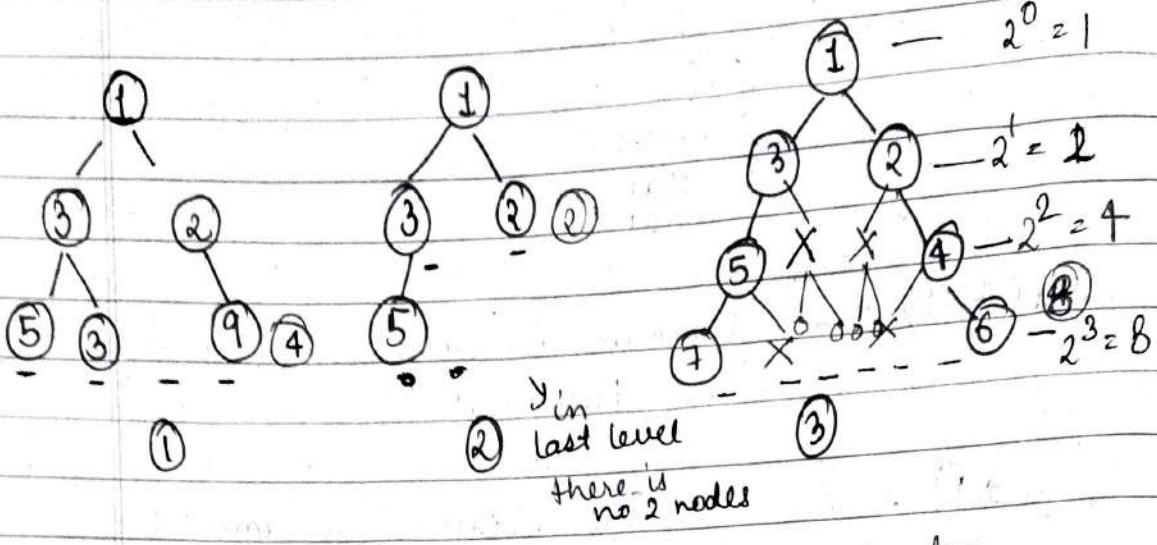
```
if (root == NULL || p == NULL || q == NULL)
  return root;
```

Node \* left = lowestCommonAncestor (root->left, p, q);  
 Node \* right = lowestCommonAncestor (root->right, p, q);

// result

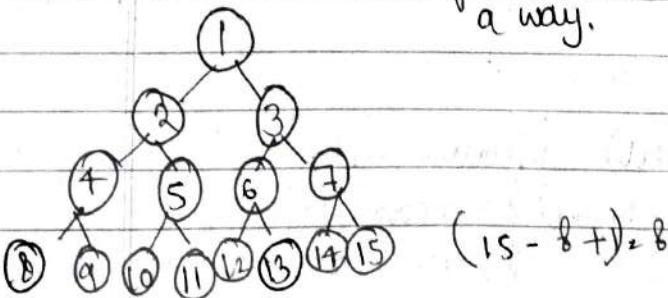
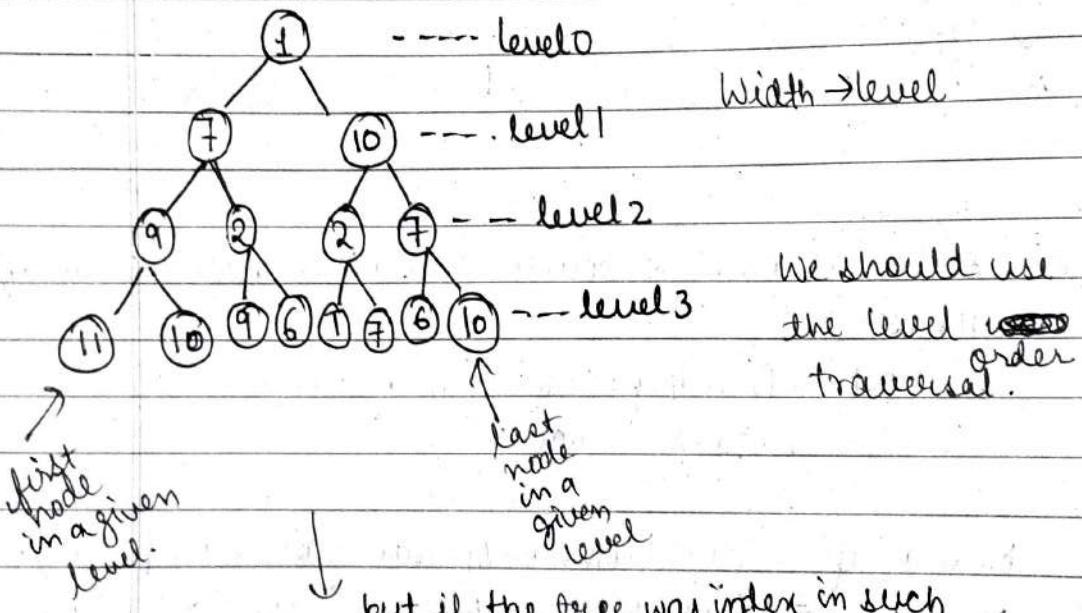
```
if (left == NULL) { return right; }
else if (right == NULL) { return left; }
else { return root; }
```

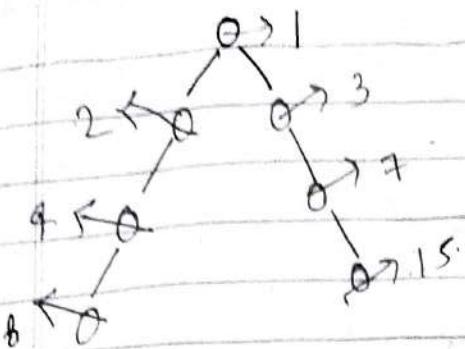
## 28 → Maximum Width of BT



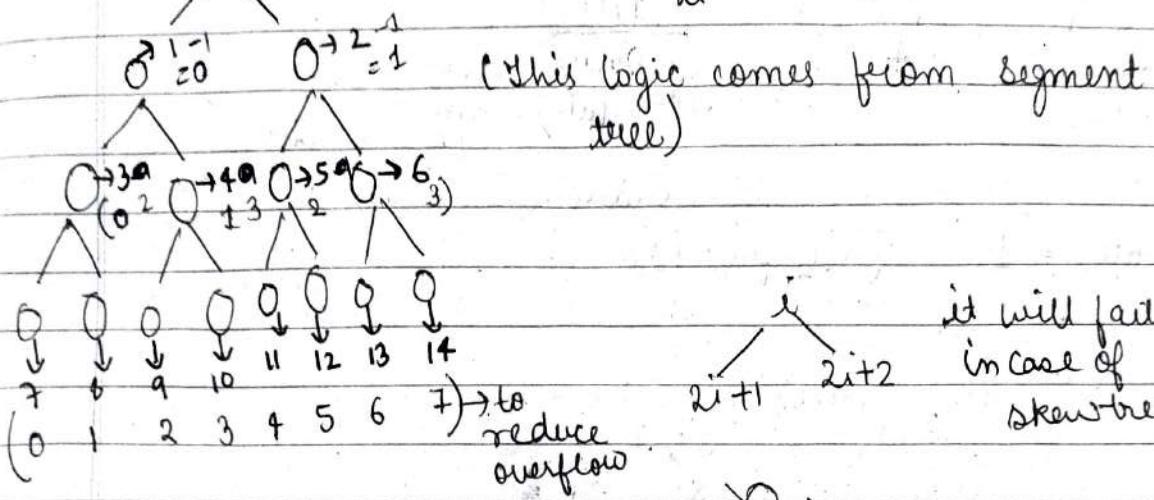
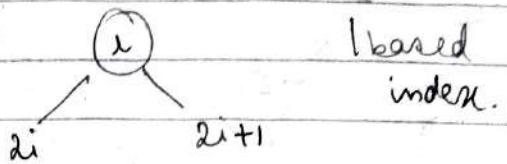
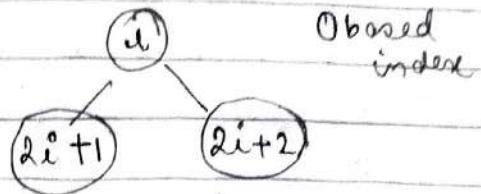
width → no of nodes bet<sup>n</sup> any 2 nodes

max no of nodes in a level bet<sup>n</sup> 2 nodes

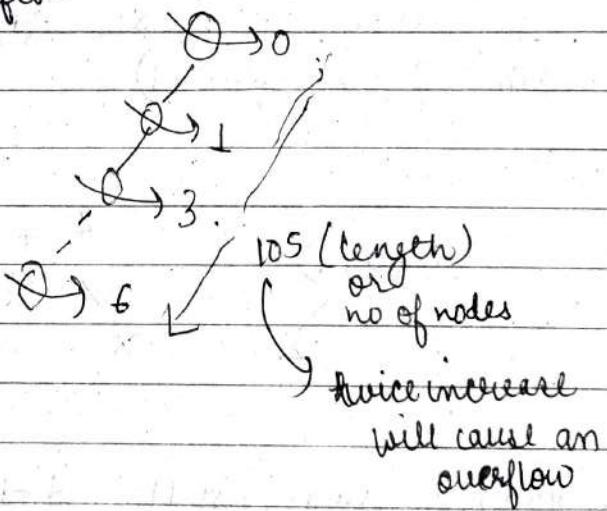
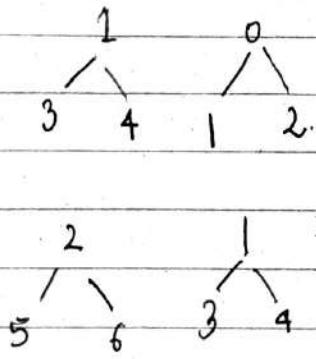


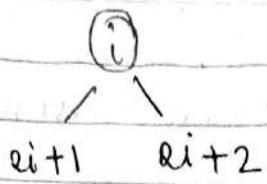


(last node - first node + 1)

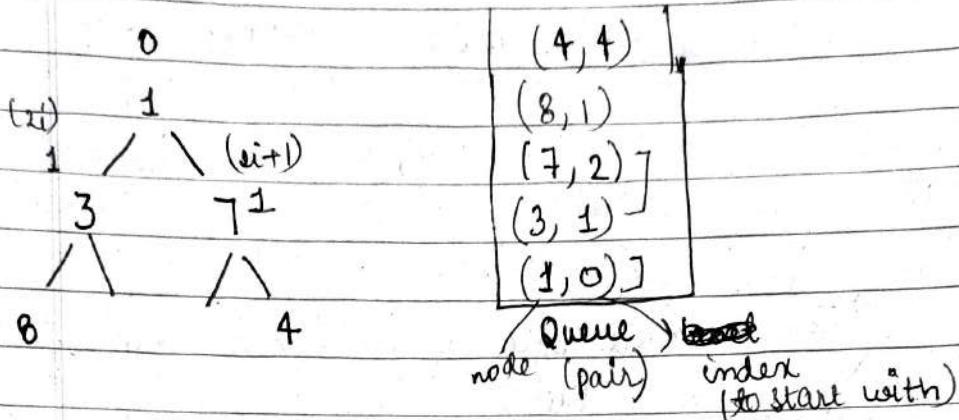


$i$   
it will fail  
in case of  
skew tree



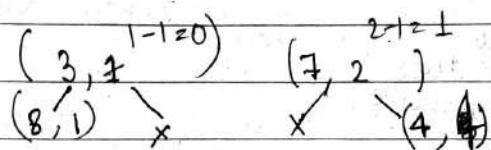


$i = i - \min \text{ initial value}$

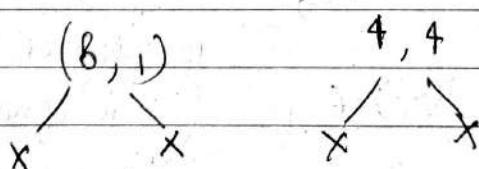


level order traversal.

width = 1 (last - first + 1)



width = 2 (last - first + 1)



Width = last - first + 1 = 4 - 1 + 1 = 4

$T C - O(N)$   
 $S C - O(N)$

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

```
queue < pair < Node*, int > q; ;  
q.push ( {root, 0} );
```

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

```
int sz = q.size();
```

```
int min = q.front().second;
```

```
int first, last;
```

```
for ( int i = 0; i < sz; i++ ) {
```

```
    int curr_id = q.front().second - min;
```

```
    Node* node = q.front().first;
```

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

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

```
    if ( i == sz - 1 ) last = curr_id;
```

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

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

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

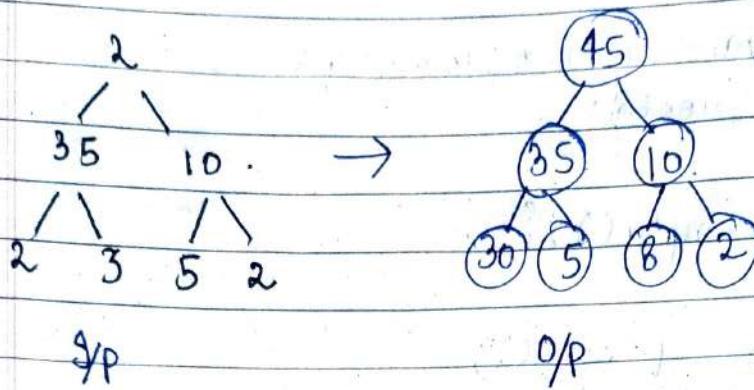
```
            q.push ( { node->right, 2 * curr_id + 2 } );
```

```
}
```

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

```
return ans;
```

## 1.29 Children Sum Property



Given a BT, maintain the children sum property.

At any node its value should be left child + right child. You can't change the structure. You can increase the node val;

node = left + right

- ) increment  
any node by +1 any no of times

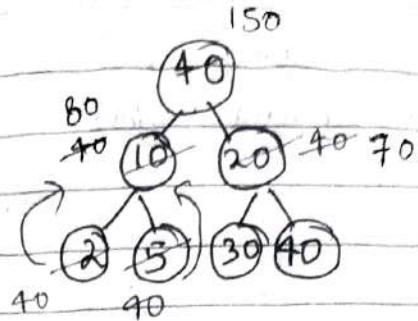
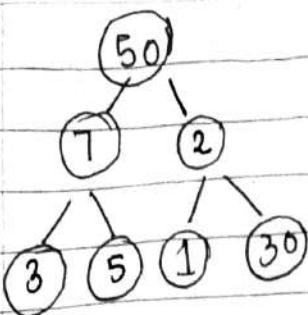
$$\frac{2}{\uparrow} \neq 35 + 10$$

## Instrument 2

$$\boxed{45 \cancel{+} 35 + 10.}$$

$$\begin{array}{r} 35 \\ \times 2 + 3 \\ \hline 30 \quad 5 \end{array}$$

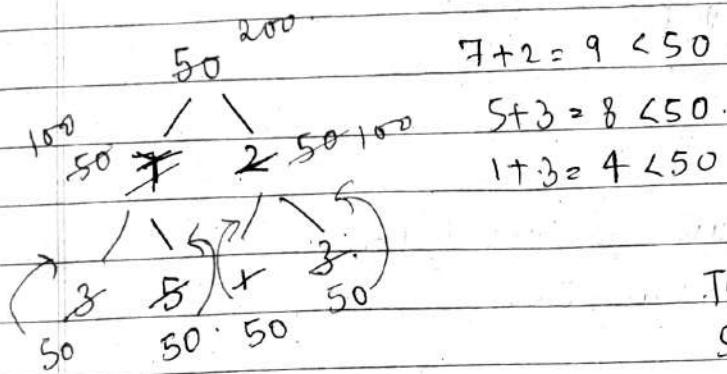
## Some of the corner cases:



$$10 + 20 = 30 < 40$$

$$2 + 5 = 7 < 40$$

$$30 + 40 = 70 > 40$$



$T.C \rightarrow O(N)$

$S.C \rightarrow O(1) \text{ or } O(N)$

↓  
for skew

void changeTree (Node<int>\*& root) {

if (root == NULL) return;

int child = 0;

50      if (root->left) { child += root->left->data; }

  \      if (root->right) { child += root->right->data; }

50 50      if (child >= root->data)      root->data = child;

  else {

    if (root->left)      root->left->data = root->data;

    else if (root->right)      root->right->data = root->data;

}

change tree (root  $\rightarrow$  left);  
change tree (root  $\rightarrow$  right);

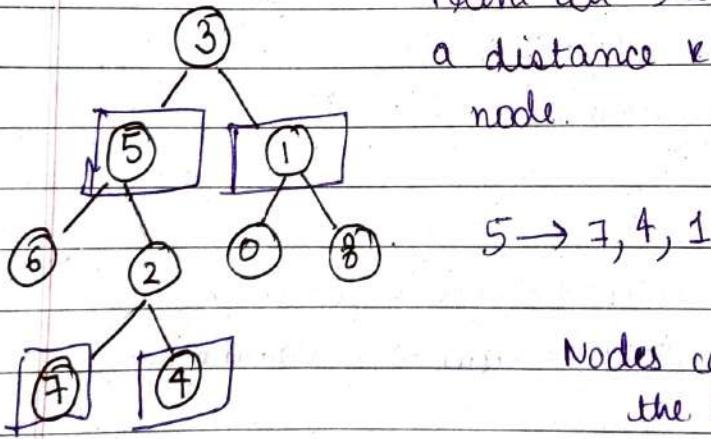
int tot = 0;

if (root  $\rightarrow$  left) tot += root  $\rightarrow$  left  $\rightarrow$  data;  
if (root  $\rightarrow$  right) tot += root  $\rightarrow$  right  $\rightarrow$  data;  
if (root  $\rightarrow$  left || root  $\rightarrow$  right) root  $\rightarrow$  data = tot;  
(to check leaf node toh nahi  
hai)

b.

30. Print all nodes at a distance K  
in Binary Tree

Point all the nodes that are  
a distance k from a given  
node.



5  $\rightarrow$  7, 4, 1

Nodes can be at  
the top or bottom

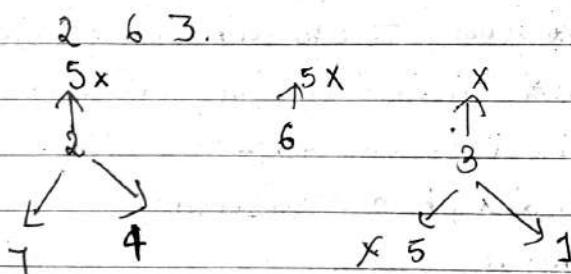
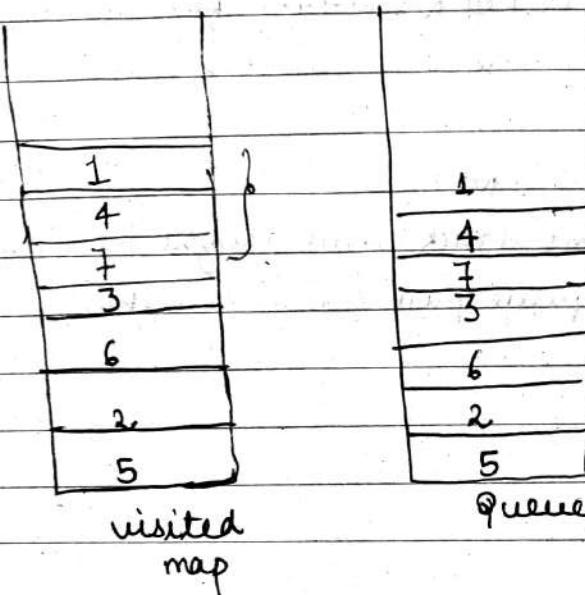
We can't travel towards the parent like from  
5 to 3. We would mark parent nodes.  
We would do BFS traversal.

parent of 5 and 1 can be stored by using map.

7
8
0
2
6
1
5
3

Queue  
(BFS)

Step 1 : Mark the parents pointer using map. ~~Backtrace~~  
visited hash.



i) (i) parent pointer

ii) ↑ distance (radially traverse upwards, downwards by a distance of 1 everytime).

$$TC \rightarrow O(N) + O(N) = O(N)$$

$$SC \rightarrow O(N) + O(N) + O(N) = O(N)$$

Date: 2023  
Page: 3

void markParents(TreeNode\* root, unordered\_map<TreeNode\*, TreeNode\*> &parent\_track, TreeNode\* target);

queue<TreeNode\*> queue;

queue.push(root);

while (!queue.empty()) {

TreeNode\* curr = queue.front();

if (curr->left) {

parent\_track[curr->left] = curr;

queue.push(curr->left);

}

if (curr->right) {

parent\_track[curr->right] = curr;

queue.push(curr->right);

}

}

vector<int> distanceK(TreeNode\* root, TreeNode\* target, int k);

(node->parent).unordered\_map<TreeNode\*, TreeNode\*> parent\_track;

markParents(root, parent\_track, target);

unordered\_map<TreeNode\*, bool> visited;

queue<TreeNode\*> queue;

queue.push(target);

visited[target] = true;

int curr\_level = 0;

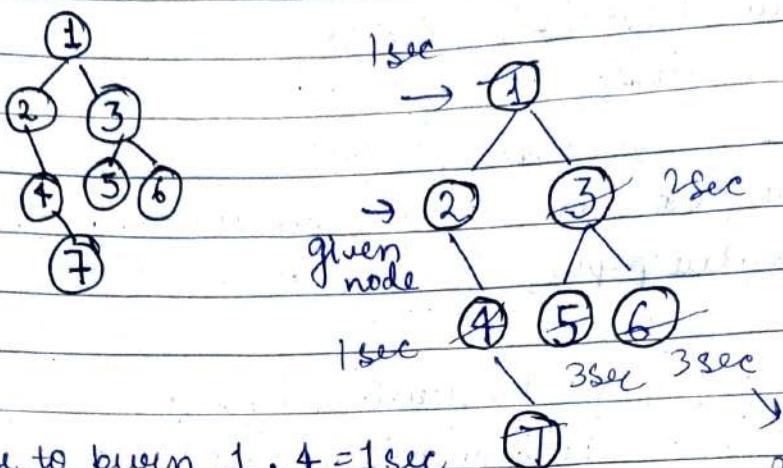
|| Second BFS is go upto k level from targetnode & using our hash table

```

while (!queue.empty()) {
    int size = queue.size();
    if (curr_level == k) break;
    curr_level++;
    for (int i = 0; i < size; i++) {
        TreeNode* curr = queue.front();
        queue.pop();
        if (curr->left && !visited[curr->left]) {
            queue.push(curr->left);
            visited[curr->left] = true;
        }
        if (curr->right && !visited[curr->right]) {
            queue.push(curr->right);
            visited[curr->right] = true;
        }
        if (parent_track[curr] && !visited[parent_track[curr]]) {
            queue.push(parent_track[curr]);
            visited[parent_track[curr]] = true;
        }
    }
    vector<int> res;
    while (!queue.empty()) {
        TreeNode* curr = queue.front();
        queue.pop();
        res.push_back(curr->val);
    }
    return res;
}

```

### 31. Minimum time taken to burn a tree



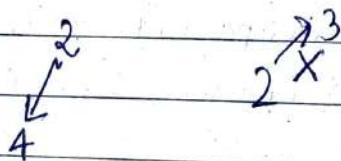
$$\text{Time to burn } 1, 4 = 1 \text{ sec}$$

$$\text{Time to burn } 7, 3 = 2 \text{ sec}$$

$$\text{Time to burn } 5, 6 = 3 \text{ sec}$$

Using bfs traversal. Say node = 2, we would start coloring nodes radially outwards.

Now the problem is we can't go upwards as



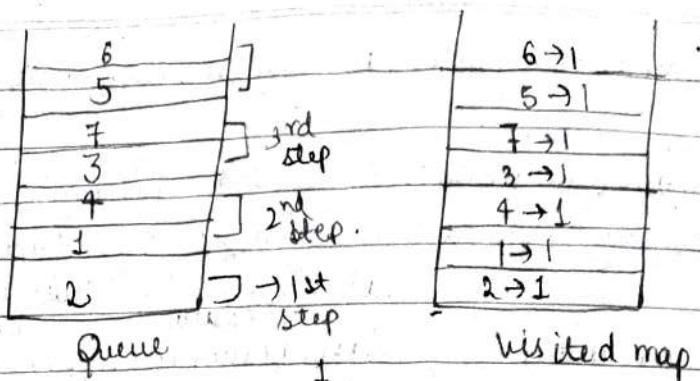
Maintain the parent pointers using hashmap.

$2 \rightarrow 1$	$3 \rightarrow 1$	$4 \rightarrow 2$	$5 \rightarrow 3$	$6 \rightarrow 3$	$7 \rightarrow 4$
X X X A G					

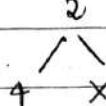
map  
(parent, node)

queue

Step 2.



time = 0 |



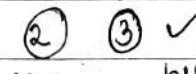
x

|

(1)

(2) x

time = 1 | 2 .



x

(2)

(3) ✓

burn

(4)

x

already  
burnt

time = 2 | 3 .

(1) x

(1) x

3

7

(6) ✓

x

time = 3

(3) x

(3) x

5

6

x x

x x

didn't burn anyone  
as they couldn't burn  
anyone.

Now the queue is empty.

We are doing level wise burning so we won't use dfs.

$$TC \rightarrow OCN + O(N) = O(2N) \approx O(N)$$

$$SC \rightarrow O(N) + O(N) = O(N)$$

Date \_\_\_\_\_  
Page \_\_\_\_\_

```
int timeToBivenTrees (BinaryTreeNode<int>* root, int start)
{
    map<BinaryTreeNode<int>*, BinaryTreeNode<int*>> mpp;
    BinaryTreeNode<int>* target = bftSToMapParents(root, mpp);
    int maxi = findMaxDistance(mpp, target);
    return maxi;
}
```

address of given data      all the nodes parents start

```
BinaryTreeNode<int>* bftSToMapParents (BinaryTreeNode<int>* root)
{
    map<BinaryTreeNode<int>*, BinaryTreeNode<int*>> mpp;
    int start;
}

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

```
int findMaxDistance( map<BinaryTreeNode<int>*, BinaryTreeNode<int>*>
                      &mpp,
                      BinaryTreeNode<int>* target);
```

```
queue<BinaryTreeNode<int>*> q;
```

```
q.push(target);
```

```
map<BinaryTreeNode<int>*, int> vis;
```

```
vis[target] = 1;
```

```
int maxi = 0;
```

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

```
    int sz = q.size();
```

```
    int fl = 0;
```

```
    for (int i = 0; i < sz; i++) {
```

```
        auto node = q.front(),
```

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

```
// left
```

```
        if (node->left && !vis[node->left]) {
```

```
            fl = 1;
```

```
            vis[node->left] = 1;
```

```
            q.push(node->left); }
```

```
// right
```

```
        if (node->right && !vis[node->right]) {
```

```
            fl = 1;
```

```
            vis[node->right] = 1;
```

```
            q.push(node->right); }
```

```
// top
```

```
        if (mpp[node] && !vis[mpp[node]]) {
```

```
            fl = 1;
```

```
            vis[mpp[node]] = 1;
```

```
            q.push(mpp[node]);
```

```
}
```

```
, if (fl) maxi++;
```

```
} return maxi;
```