Date
Page

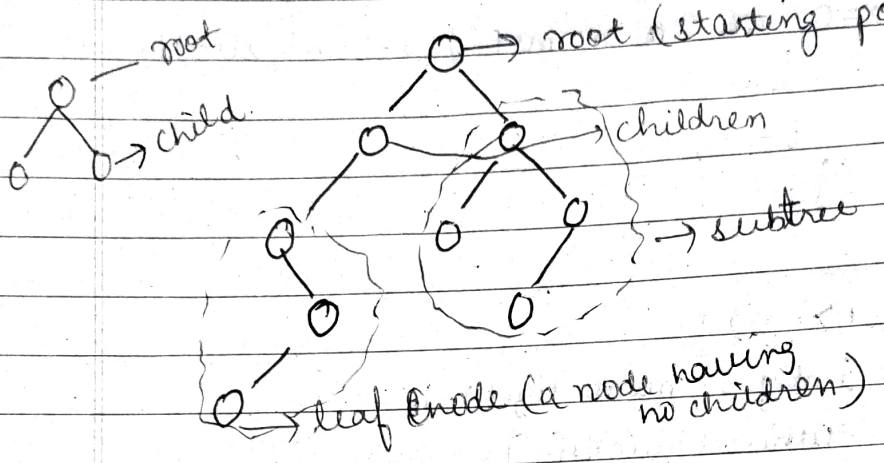
Notes Made By

– Riti Kumari

~~Binary Traversals~~

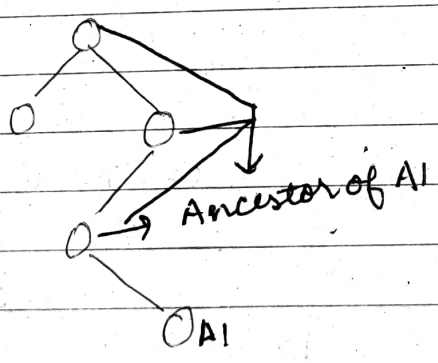
Tree Series

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

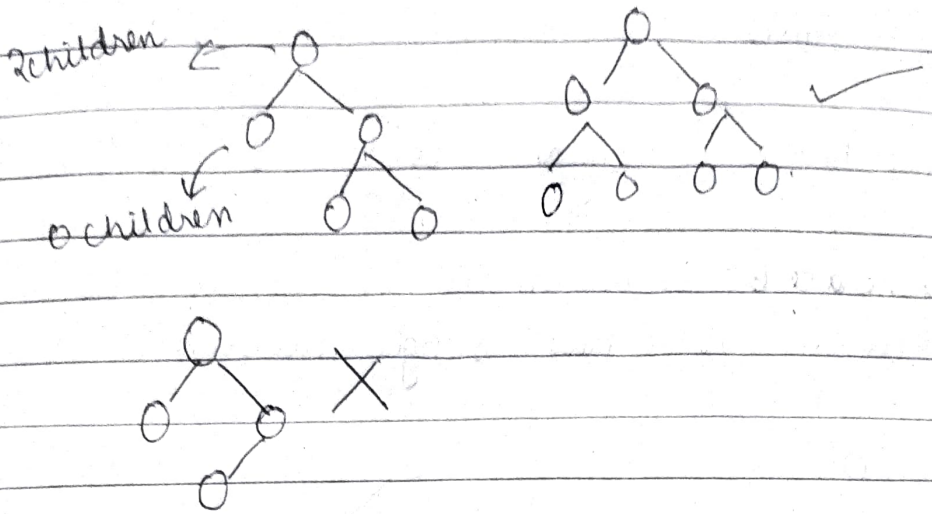
Ancestor → All the parents are called ancestor



Types of BT

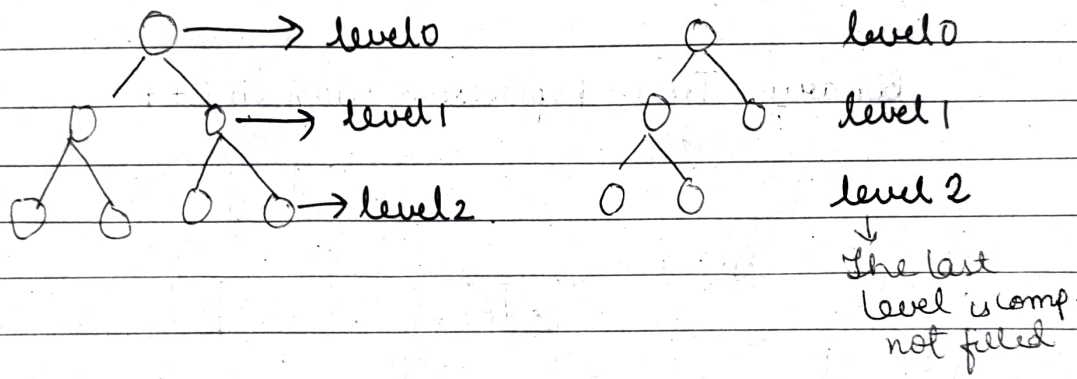
- Full BT
- Complete BT
- Perfect BT
- Balanced BT
- Degenerate BT

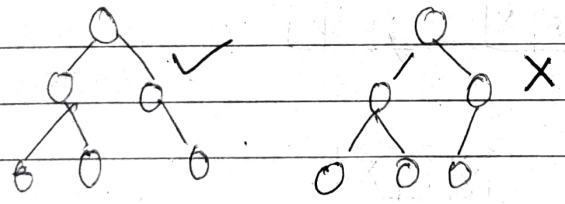
Full BT - Every node has 0 or 2 children



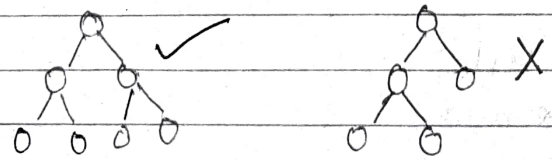
Complete BT - All levels are completely filled except the last level.

1) The last level has all nodes on left as possible.





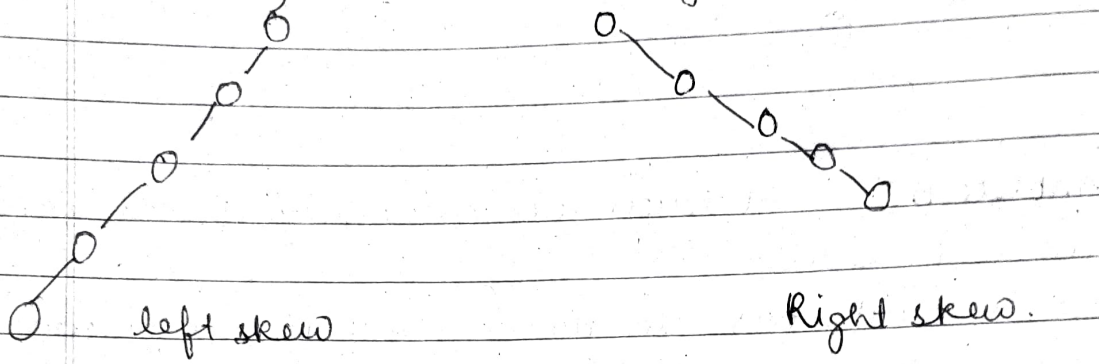
Perfect BT - All leaf nodes are at the same level.



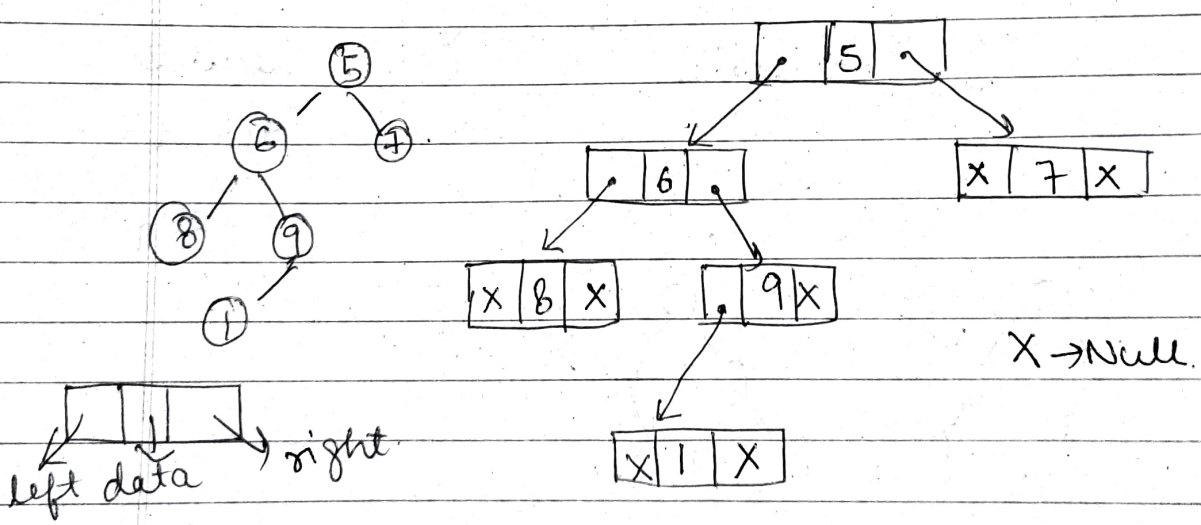
Balanced BT → height of tree is at max $\log(N)$ (N → nodes)

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

Degenerate BT → when it is a skew tree. Every node has single children.

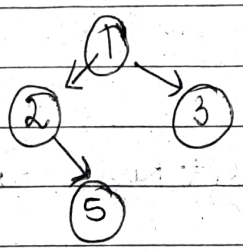


Binary Tree Representation in C++




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

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

int main(){

Struct Node* root = newNode(1);
 root → left = newNode(2);
 root → right = newNode(3);
 root → left → right = newNode(5);
 }
```

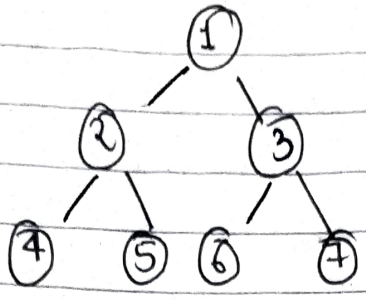


Binary Tree traversals in BT (BFS | DFS)

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

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

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



1) Inorder (left Root Right)

4 2 5 1 6 3 7

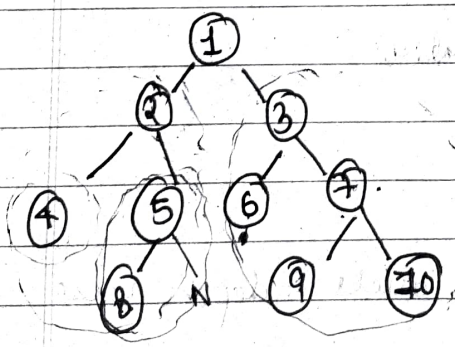
2) Pre order traversal (Root left Right)

1 2 4 5 3 6 7

3) Post order traversal (left Right Root)

4 5 2 6 7 3 1

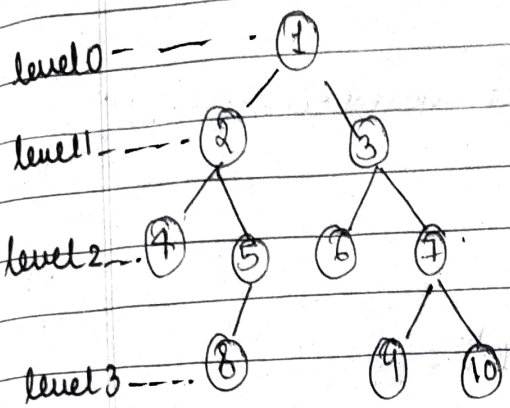
Pre → Root at first
Post → Root at last
In → Root in between



Inorder- (LRo Ri)

4 2 8 5 1 6 3 9 7 10

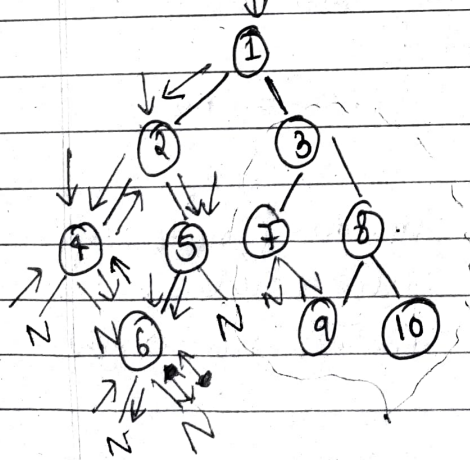
ii) BFS (Breadth first search) → Travels levels wise



1 2 3 4 5 6 7 8 9 10

Pre order traversal (Root, left, Right)

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



1 2 4 5 6 3 7 8 9 10.

Time Complexity : O(N)
Space Complexity : O(N)

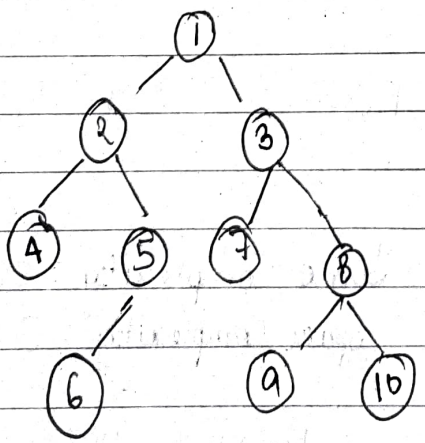
N = no of nodes

leetcode.

```
void preorder (TreeNode* root, vector<int>& temp) {
    if (!root)    return;

    temp.push_back (root->val);
    preorder (root->left, temp);
     preorder (root->right, temp);

}.
```

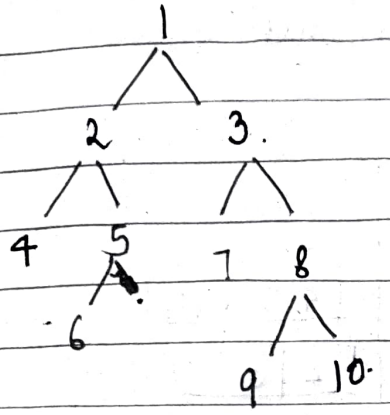
```
vector <int> preOrderTraversal (TreeNode* root)
  {
  vector <int> temp;
     preorder (root, temp);
     return temp;
  }.
```

Inorder traversal (left Root Right)



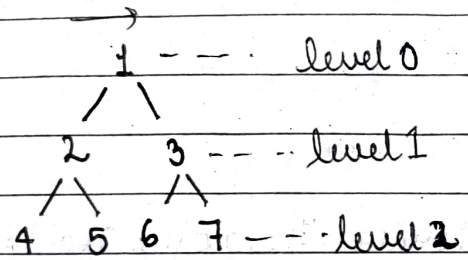

```
void inorder (node)
  {
  if (node == null)
      return;
  inorder (node->left)
  print (node->data)
  inorder (node->right)
```

Post order Traversal of Binary Tree




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

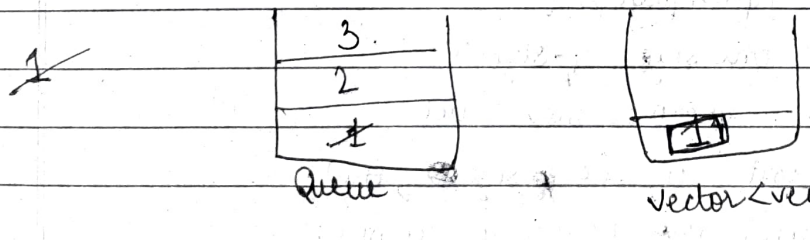
Level order traversal (BFS)



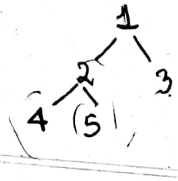
1
2 3
4 5 6 7

1 2 3 4 5 6 7

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

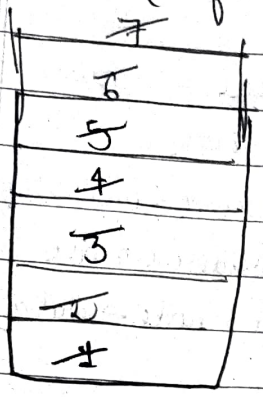


If the left exists put it into queue. then check if right exists & put it into queue

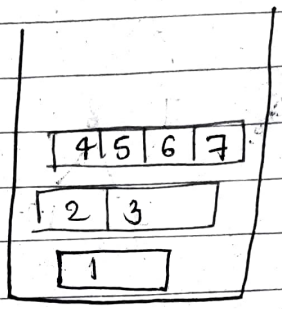


Pre : 1 2 4 5 3. (Ro, L, R)
In : 4 2 5 1 3 (L Ro R)
Post : 4 5 2 3 1 (L, R, Ro)

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



Queue.



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

Taking out of queue 1

2 3

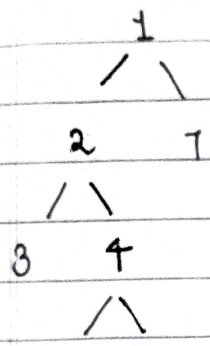
4 5 6 7

```
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

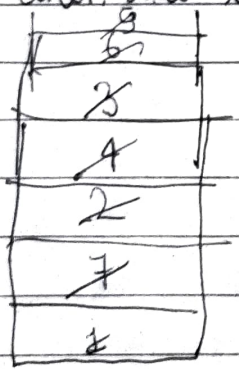
Root left Right



1 2 3 4 5 6 7

We would be doing it using stack

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 ~~took~~ put out right & then the left node.



1 2 3 4 5 6 7

Stack becomes empty → Entire pre order traversal is printed.

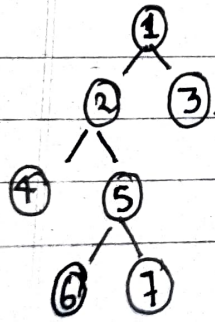
```
vector <int> preOrderTraversal( Treenode * root)
{
   Vector <int >preorder;

  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)

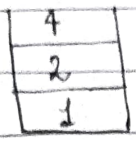
Iterative Inorder Traversal

Left Root Right

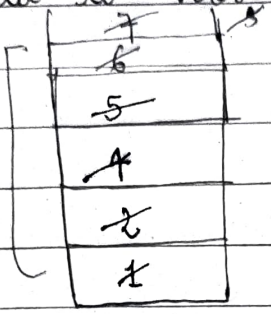


4 2 6 5 7 1 3

In recursion: left print right
root



we would use the stack (LIFO). Have a node initialised to root. if node != null



Stack trace

4 2 6 5 7 1 3

node: 1 2 4 null null 5 6 null null 7 null null
(6→left) (6→right)

move to the left till you get a null. print 4 now print 4 right.

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.

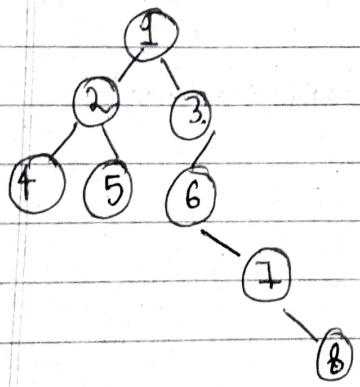
```
vector <int> InorderTraversal (TreeNode * root) {
    Stack < TreeNode * > St;
    TreeNode * node = root;
    vector <int> inorder ;
    while(true) {
        if (node != NULL) {
            St.push(node)
        node = node -> left ;
    }
```

```
else {
    if (st.empty() == true) break;

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

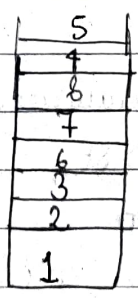
TC - O(N)
SC - O(N) (Auxillary stack space)

Iterative postorder. (Left Right Root)

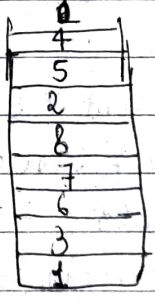
Using 2 stack



4 5 2 8 7 6 3 1



St 1



St 2

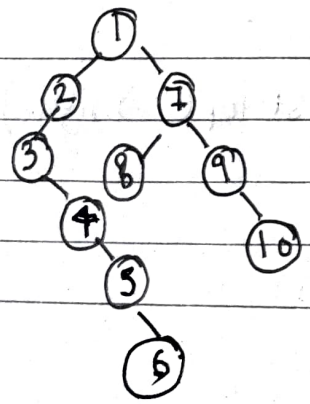
4 5 2 8 7 6 3 1

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;
}
```

TC → O(N) (Traversing for every nodes)
SC → O(2N)

Using 1 stack (left Right Root)



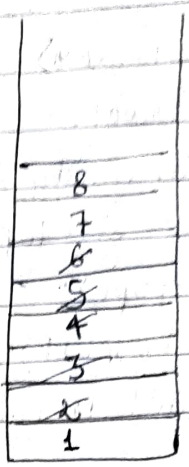
Recursion : left Right print

We will use an external stack.

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

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

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



postorder vec : 6 5 4 3 2 8 7 1

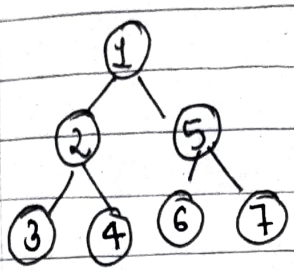
temp = st.top() → right

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

    if ( curr != null)
        st.push (curr);
        curr → left;
    else
        temp = st.top() → right;
        if (temp == null)
            temp = st.top ();
            st.pop();
        postorder ( temp);

    while (!st.empty && temp == st.top() → right)
            temp = st.top();
            st.pop();
        postorder (temp → val);
    else
        curr = temp;
```

T.C → O(2N)

S.C → O(N)

PreOrder, Inorder, Postorder (In one video)



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

(Ro L Ri) Preorder list - 1 2 3 4 5 6 7

(L Ro Ri) Inorder list - 3 2 4 1 6 5 7

(L Ri Ro) Postorder list - 3 4 2 6 7 5 1

x	(7, x) z 3
x	(6, +) z 3
x	(5, +) z 3
x	(4, +) z 3.
x	(3, +) z 3
x	(2, +) z 3
x	(1, +) 3

if num == 1
    preorder
    ++ (push that num)
    !left enter left.

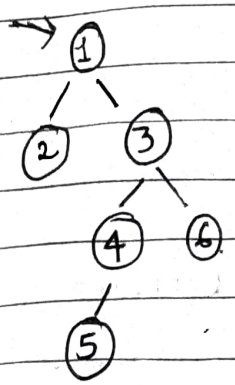
if num == 2
    Inorder
    ++
    !right push right

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 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++;
        st.push(it);
    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, 1});

// dont push it back again
    else    post.push_back(it.first->val);
}
```

Maximum Depth in BT (Height of BT)



height = 4.

Recursive | levelorder

↓ Auxiliary space

SC → O(N)

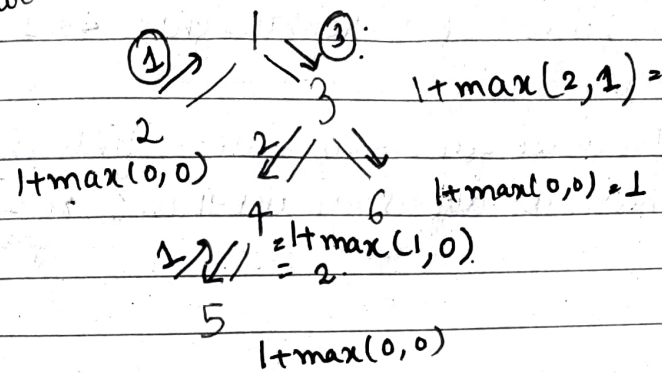
O(height)

↓ level order as we use queue.

SC → O(N)

1 + max (l, r)

↓ for the current node

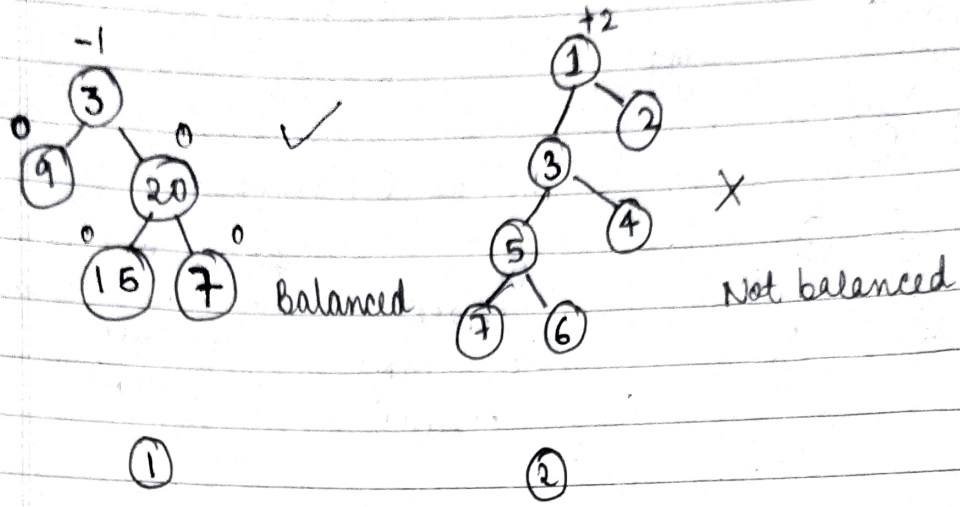



```
int maxHeight (Treenode * root) {
    if (root == NULL) return 0;

    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



Balanced BT → for every node
height(left) − height(right) <= 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 = findHLeft (node → left);
Rh = findHRight (node → right);   ] O(N)

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

bool left = check (node → left);
bool right = check (node → right);
```

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

return true;

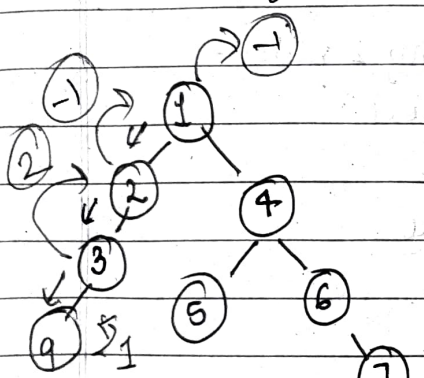
TC → O(N) X O(N) ≃ O(N²)

↓ traversal ↓ for every height

if balanced

Yes — return height

No — return -1

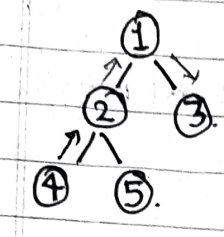



```
int check (Treenode * root) {
    if (!root) return 0;

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

    return 1+max(lh, rh);
}

bool isbalanced(Treenode * root) {
    return check(root) != -1;
}
```

TC – O(N)

SC → O(N) Auxiliary space

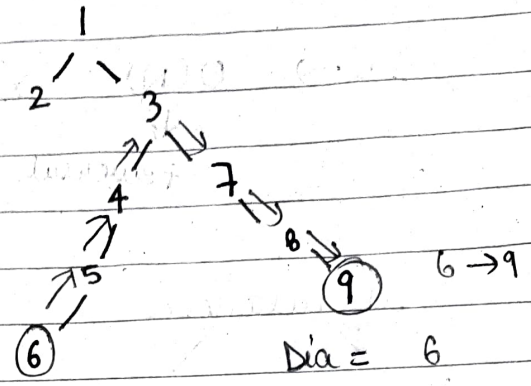
Diameter of Binary Tree



4 → 3

Dia = 3

①



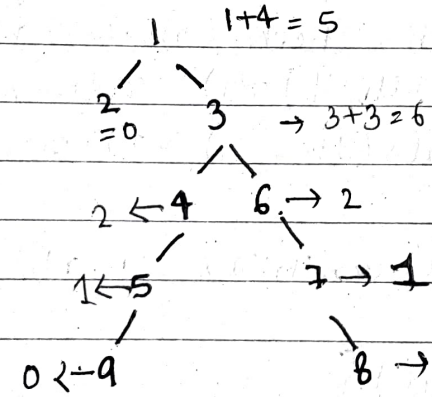
6 → 9

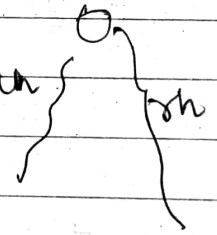
Dia = 6

②

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

Naive: Traverse the entire node





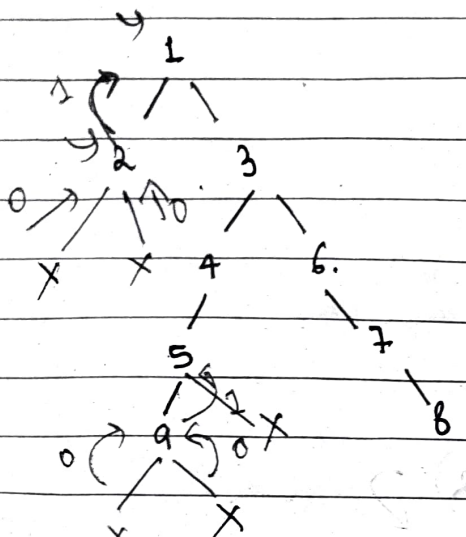
maxi = max (lh + rh, maxi)

```
find Max (node)
    {
    if (root == null)
            return;
lh = findleftH( node → left)
rh = findrightH( node → right)
    maxi = max (maxi, lh+rh);
```

findMax(node→left)
findMax(node→right)
}

T.C = O(N²)

Optimised Approach - using height of BT

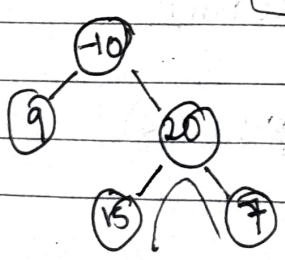



```
int diameter(TreeNode* root){
    int dia = 0;
    height(root, dia);
    return dia;
}

int height(TreeNode* node, int& dia){
    if(!node) return 0;
    int lh = height(node→left, dia);
    int rh = height(node→right, dia);
    dia = max(dia, lh+rh);
    return 1+max(lh, rh);
}
```

TC → O(N)
SC → O(N)

Maximum Path Sum in BT

→ node A → node B (in a path a single node only appear once)

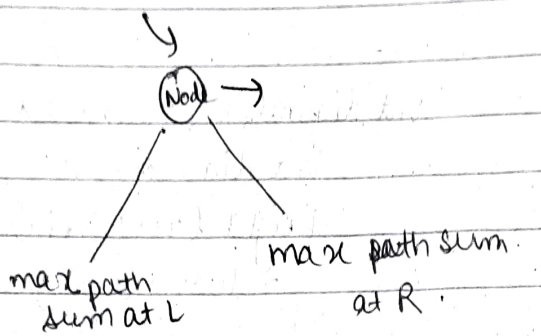


15 20 7 (15→7)
15 20 -10 9 (15→9)

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

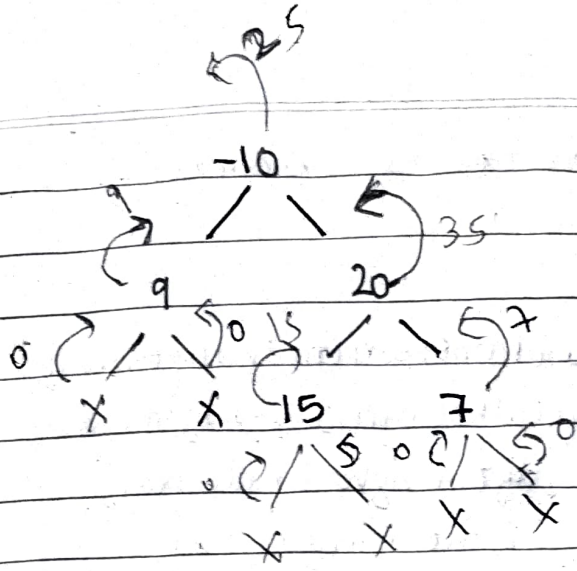
Brute force: Try out every possible combination of A & B. Whichever path is giving the max sum is the max.

Optimal Approach: find the maxm height of BT and find the diameter



⇒ val + (maxL + maxR) → 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 = ~~0~~ ~~9~~ ~~15~~ 42

max = 9 + 35 - 10
= 25 + 9
= 34

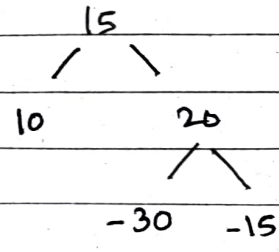
return = -10 + max(9, 35)
= -10 + 35
= 25

= 15 + 7 + 20
= 42

max

return = 20 + max(15, 7)
= 20 + 15
= 35

max path in some cases.



→ for this tc don't consider a negative path sum.
So ignore & make it zero.

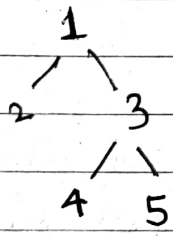
```
int maxPathSum (TreeNode * root) {
    int maxi = INT_MIN;
    maxPathDown( root , maxi );
    return maxi;
}
```

```
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;
}
```

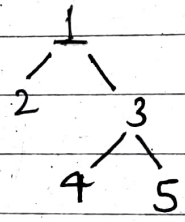
T.C → O(N)
S.C → O(N) Auxiliary space.

Check if 2 trees are identical or not

T.C → O(N)
S.C → O(N)



Tree 1



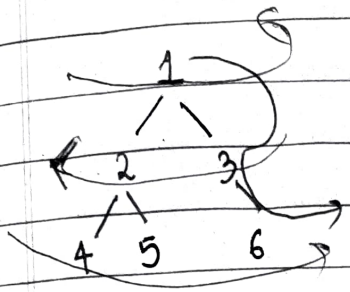
Tree 2

preorder – Root, Left, Right

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

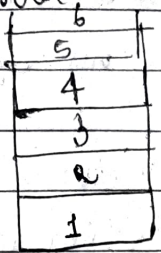
```
bool isSame (TreeNode* p, TreeNode* q) {
 if (p == NULL || q == NULL) {
        return (p == q);
    }
return (p->val == q->val) && isSame (p->left, q->left)
                          && isSame (p->right, q->right);
}.
```

Zig Zag or spiral traversal



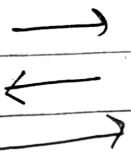
1 3 2 4 5 6.

Take a queue ds & store root



Queue.

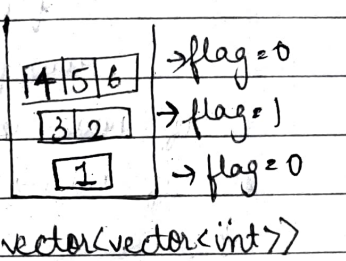
1
~~2~~ ~~3~~
~~4~~ ~~5~~ ~~6~~



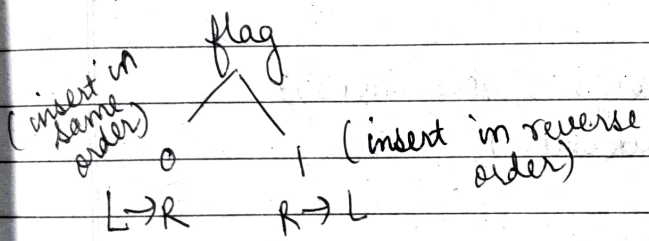
We introduce a flag variable.

flag = 0

If put 1 in queue first i.e. root. Now put 1 out if left exsists push left & for right the same.



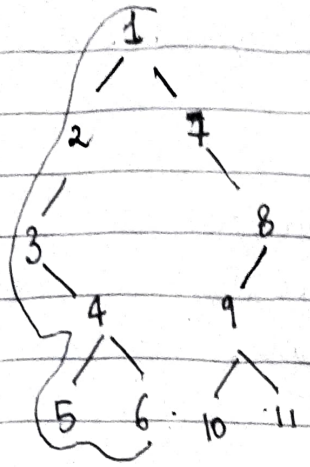
vector<vector<int>>



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(f%2==0){
  reverse(row.begin(), row.end());
  result.push_back(row);
} f=!f;

return result;
}
```

Boundary Traversal

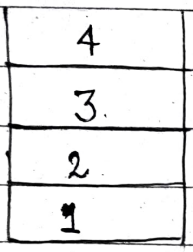


Anticlockwise boundary traversal

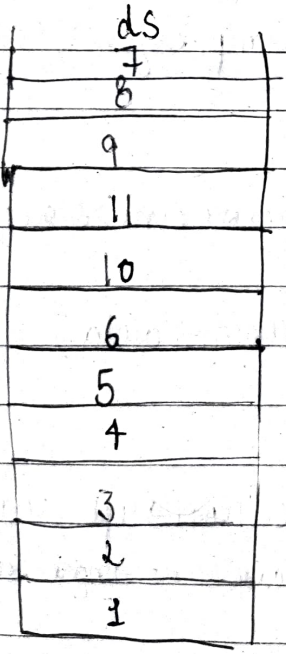
1 2 3 4 5 6

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.

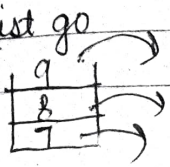


put on left if left doesnt exsist put right. Whenever you encounter leaf node stop left boundary traversal



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

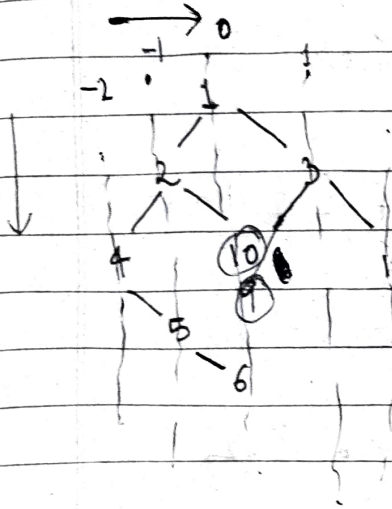
For right boundary use a vector or stack
If right doesnt exsist go to the left



```
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)
            curr = curr->right;
        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);
}
```

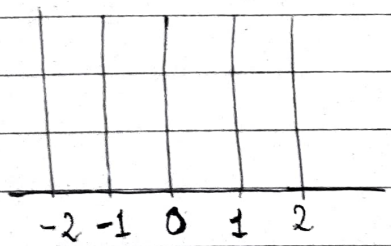
```
vector<int> printBoundary(Node *root)
{
    vector<int> res;
    if(!root) return res;
    if(!isLeaf(root)) res.push_back(root->data);
    addLeftBoundary(root, res);
    addLeaves(root, res);
    addRightBoundary(root, res);
    return res;
}
```

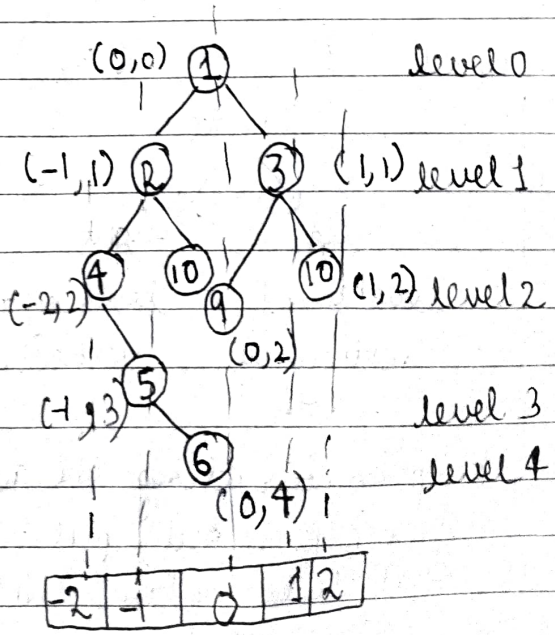
Vertical Order Traversal



level0 [4]
level1 [2 5]
level2 [1 9 10 6]
level3 [3]
level4 [10]

the smaller one at first if nodes are overlapping





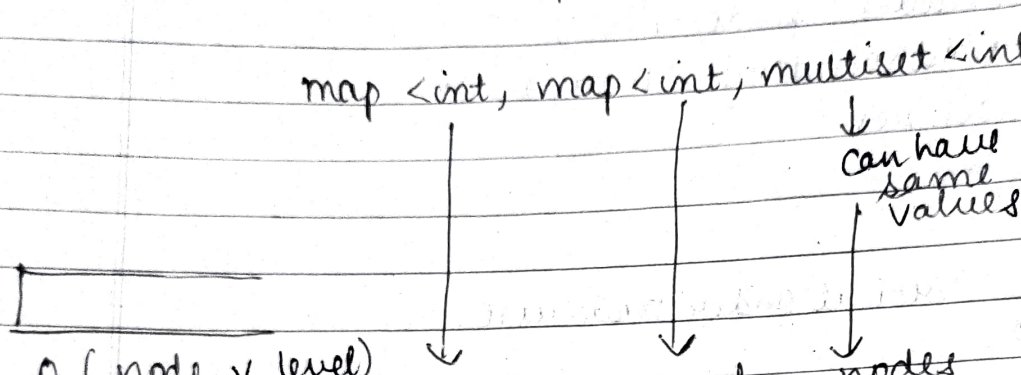
x → 4

level ↓ 2 5

1 9 10 6

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.



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

Q

node = 1 (0, 0) 0 → 0 → {1}

node = 2 (-1, 1) -1 → 1 → {2}

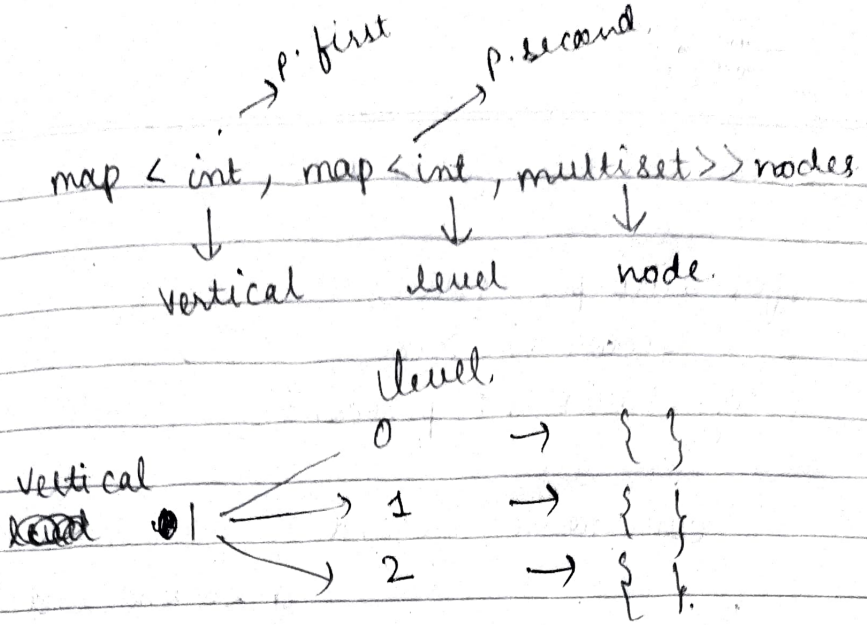
node = 3 (1, 1) 1 → 1 → {3}

node

left
level by +1
vertical by -1

right
level by +1
vertical by +1

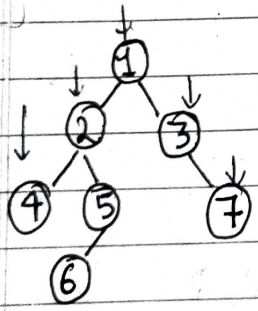
keep on iterating the queue. Every time you pop it out put it into vertical after that level after that in the multiset




```
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());
        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}});

}
```

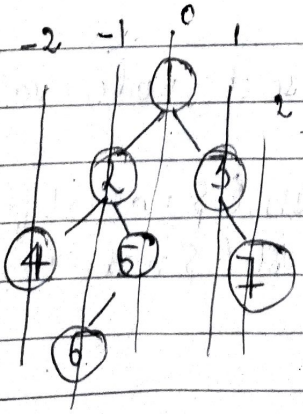
```
vector<vector<int>> ans;

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.



We require a queue.

(4, -2) (5, 0)
(3, 1)
(2, -1)
(1, 0)

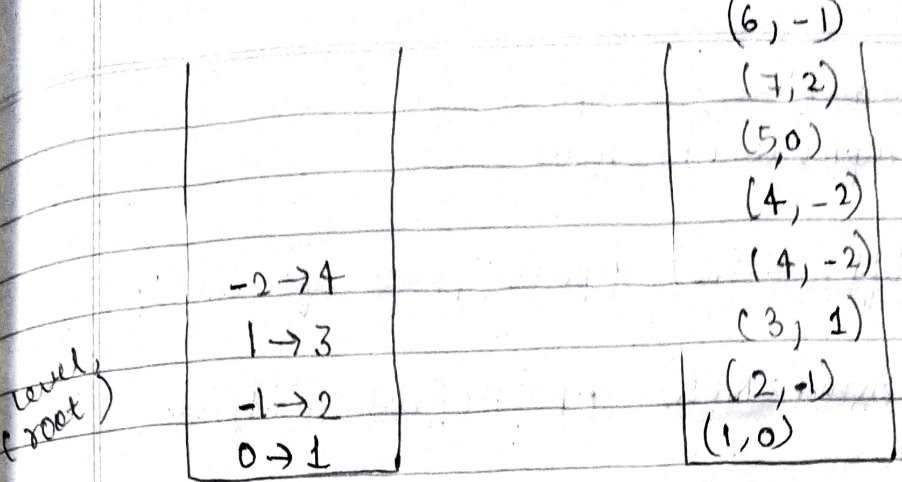
node=4, 2
0, -1

-1 → 2
0 → 1

(Line, node) map

↓ node ↓ line

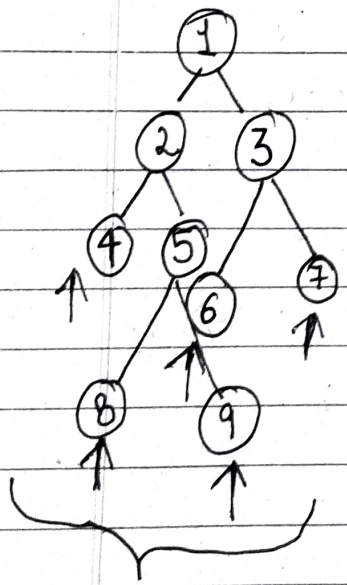
q




```
vector<int> topView(Treenode* root) {
    vector<int> ans;
if(!root)    return ans;
    map<int, int> mpp;
    queue< Treenode*, pair<int, int>> q;
    q.push({root, {0, 0}});

    while(!q.empty()) {
        auto it = q.front();
        q.pop();
    int line = it.second;
    Treenode* node = it.first;
```

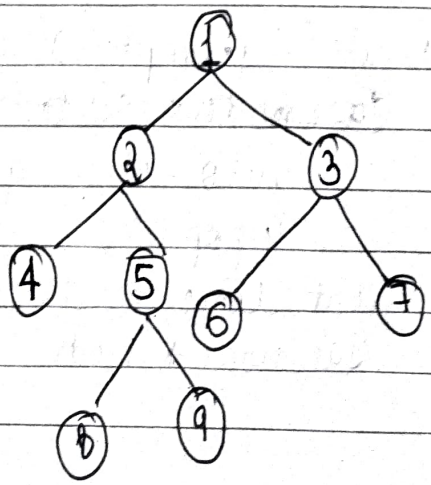
```
if (mpp.find(line) == mpp.end())
        mpp[line] = node->data;

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;
}
```

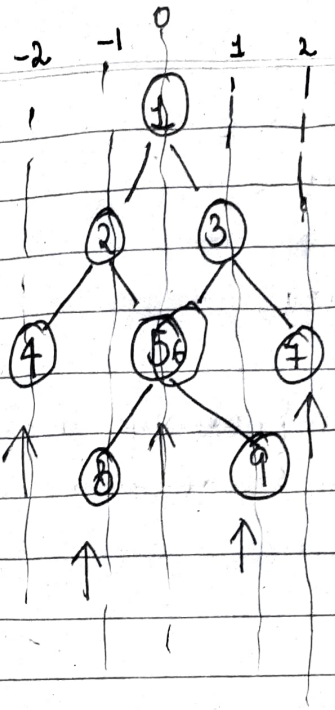
Bottom view of BT



4 8 6 9 7

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)

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

Queue. map (line, node)

replace (5,0) with (1,0)

node = ~~1~~ ~~2~~ ~~3~~ 4 5 6 7 8 9

0 -1 +1 -2 0 0 2 -1 +1

o/p

-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.

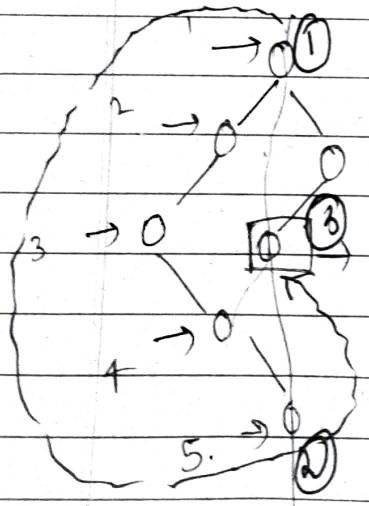
```
vector<int> bottomView( Node *root){
    vector<int>ans;
  if(!root)        return ans;
    map<int, int> mpp;
  queue<Treenode*, int> q;
  q.push({root, 0});

  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?

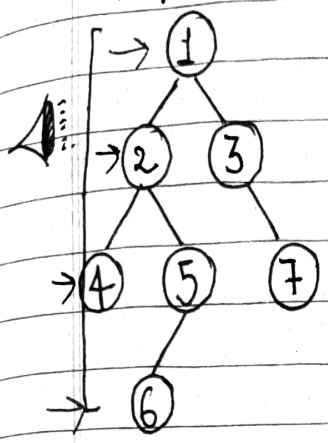
Ans: No it won't work with
(left, line+1)
(right, line+1)



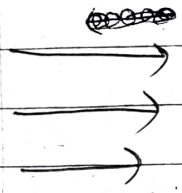
at the end this node would be visited in inorder traversal

Right / left view of Binary

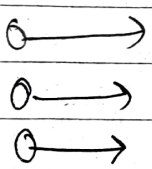
Left view



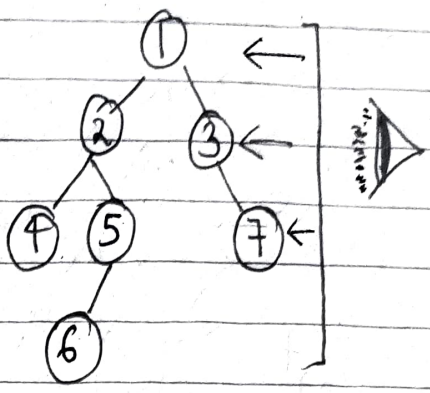
Leftview: 1 2 4 6



first node of every level in

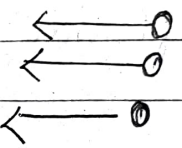


Right view



Right view: 1 3 7

first node of every level.

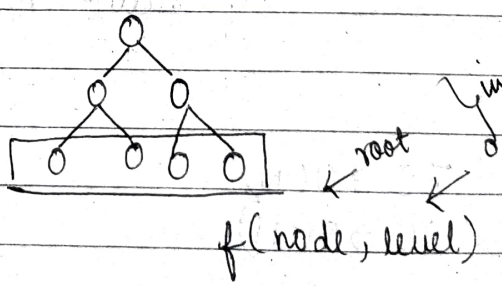


SC TC

O(H) O(N) ← ① Recursive traversal

O(N) O(N) ② Iterative traversal.

We wont use level order traversal

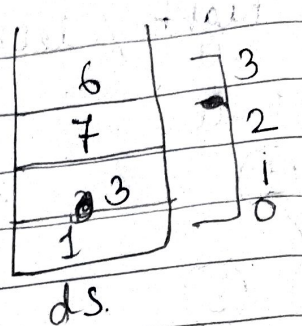


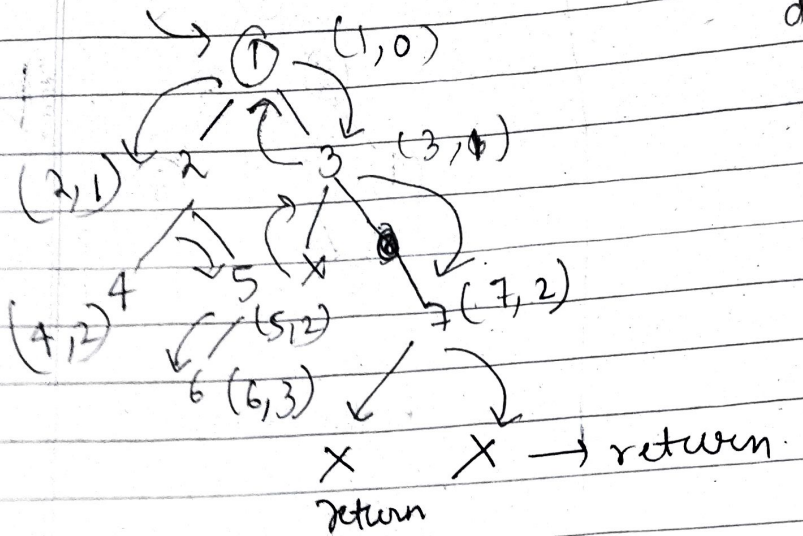
we would Reverse preorder traversal (Root Left Right)

(Root Right left)

Right view of binary tree

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





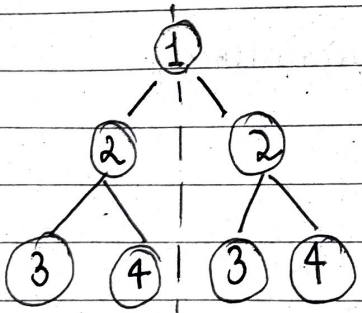
Left side

```
f (node, level)
if (node == null) return;

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

if (node → left , level+1)
if (node → right , level+1)
```

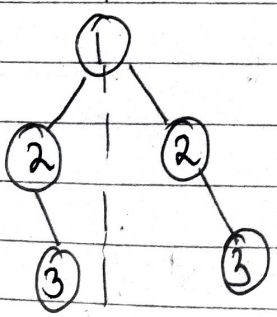
TC → O(N)
SC → O(H)

```
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



not symmetric around its center

Left ⇌ Right

Root→left ⇌ Root→Right

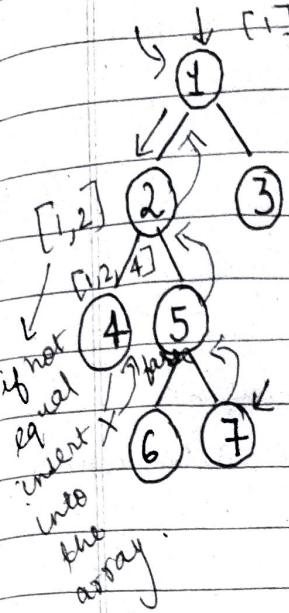
Root → left | Root Right

Preorder Root left 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( root.
                p → left, q → right)
            && helper ( q → right, q → left );
}
```

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



Node = 7

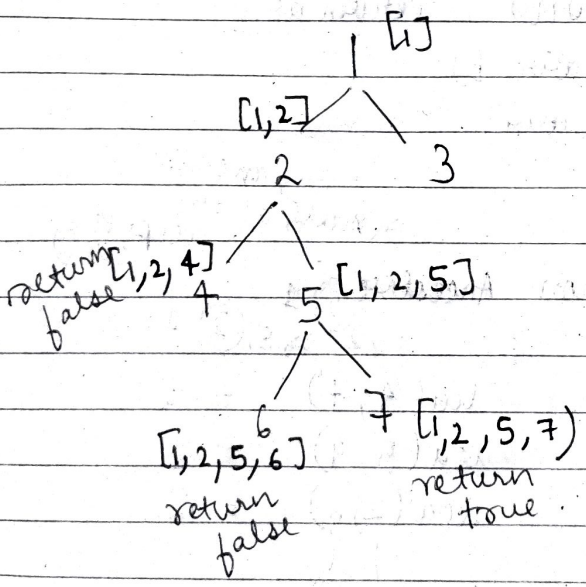
{1, 2, 5, 7}

We don't have the parent node. We would use the inorder traversal Use recursion.

Inorder: left Root Right

false (left ⇄)
false (right ⇄)
} (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.

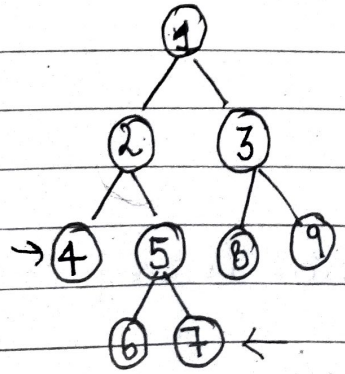


In question it says the value always exists.

```
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;
}
```

Lowest Common Ancestor → (node/ancestor at the deepest level)



lca(4, 7) = 2.
lca(5, 9) = 1
lca(2, 6) = 2.

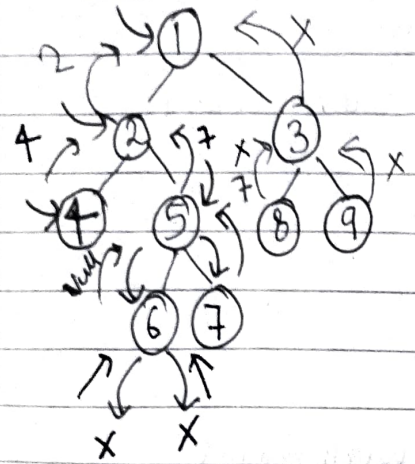
2, 1
5, 2, 1
2, 1
6, 5, 2, 1

node = 4 (1, 2, 4) TC - O(N)

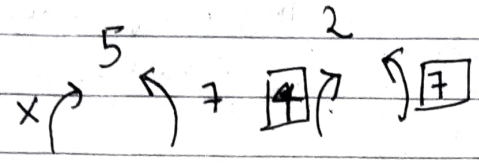
node = 7 (1, 2, 5, 7) SC - O(N) + O(N)

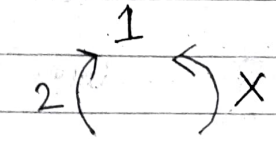
(last element that matched)

lca(4, 7) = 2 (Recursive traversal i.e dfs traversal)



Just because we figured out both the 4 & 7 we return 4 & 7

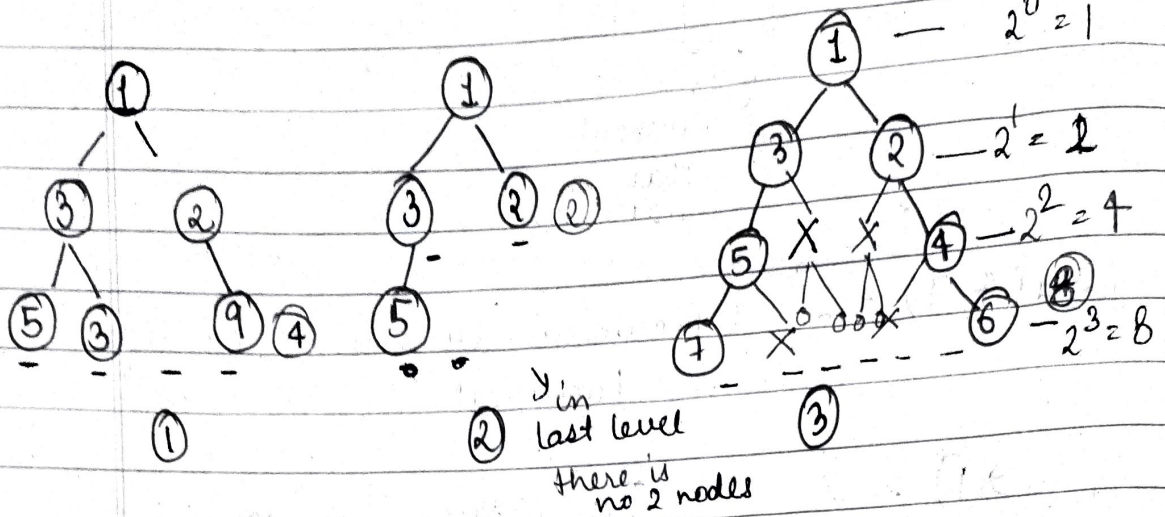




lca(4, 7) = 2

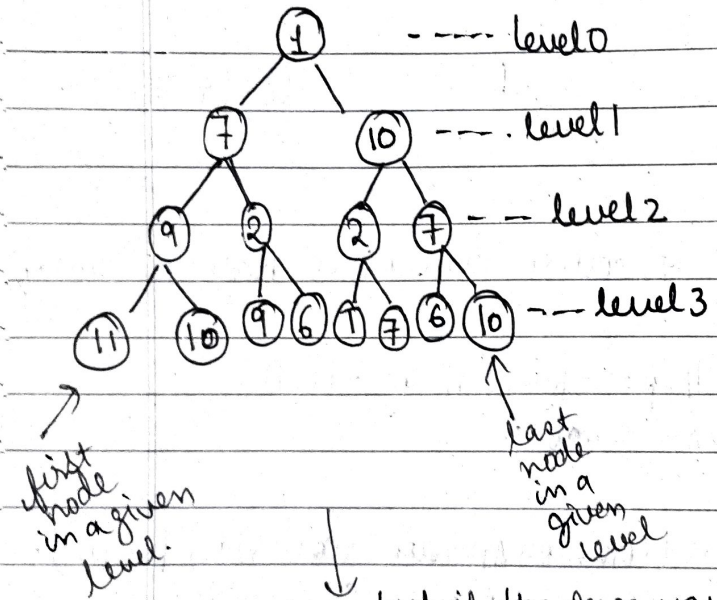
```
Node* lowestCommonAncestor(Node* root, Node* p, Node* q)
    // base case
    if(root == NULL || p == NULL || q == NULL)
        return root;

    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^n any 2 nodes

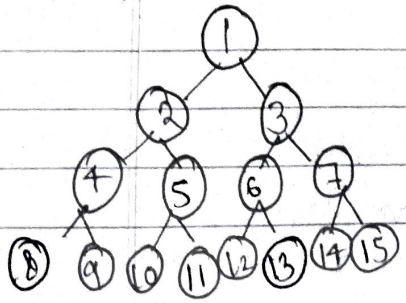
max no of nodes in a level bet^n 2 nodes.



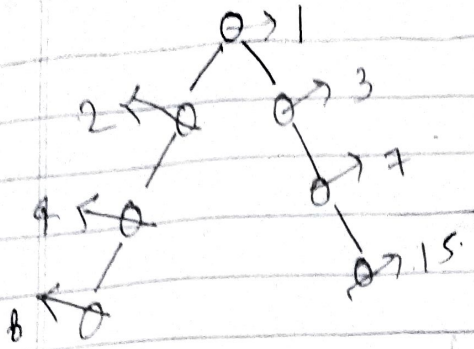
Width → level

We should use the level order traversal.

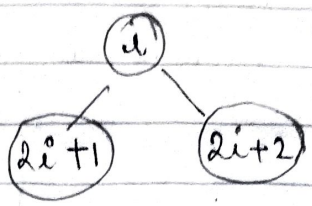
but if the tree was index in such a way.



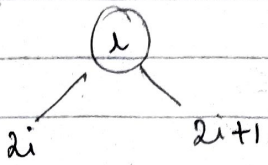
$(15 - 8 + 1) = 8$



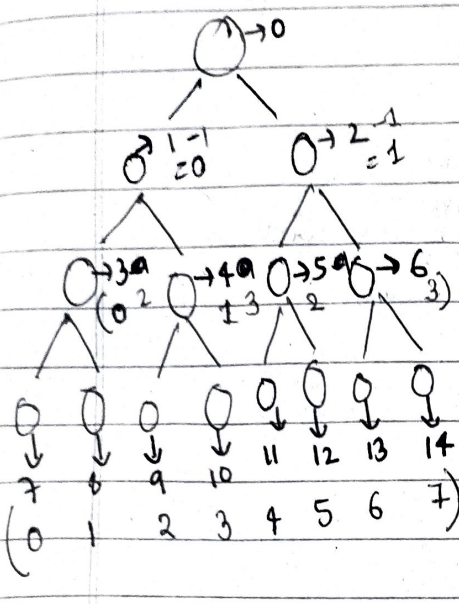
(last node − first node + 1)



0 based index

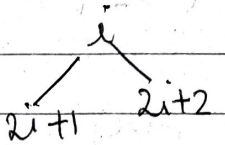


1 based index.

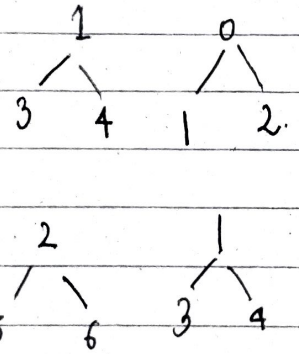


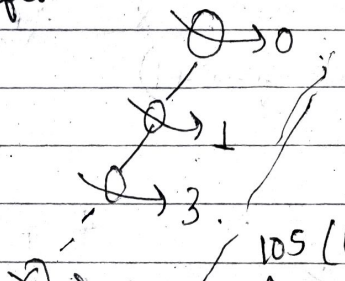
(This logic comes from segment tree)

to reduce overflow.



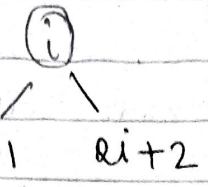
it will fail in case of skew tree



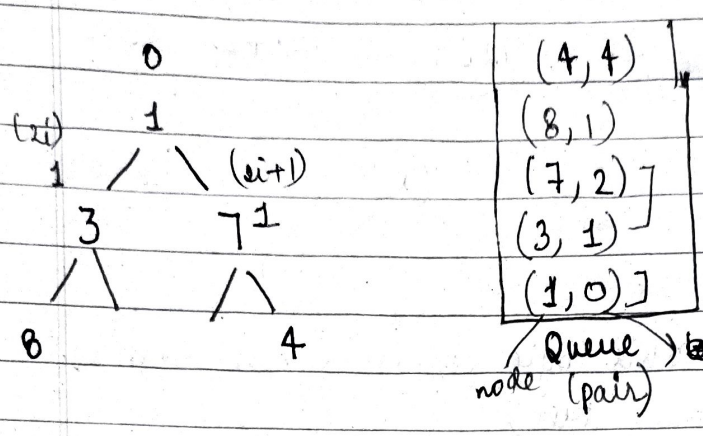


105 (length) or no of nodes

twice increase will cause an overflow

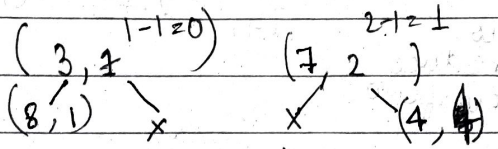


i = i - minimal value

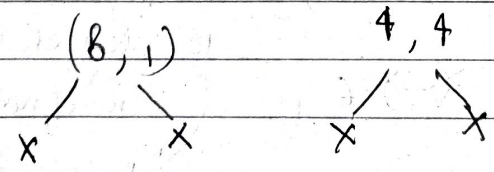


level order traversal.

width = 1 (last - first + 1)



width = 2 (last - first + 1)

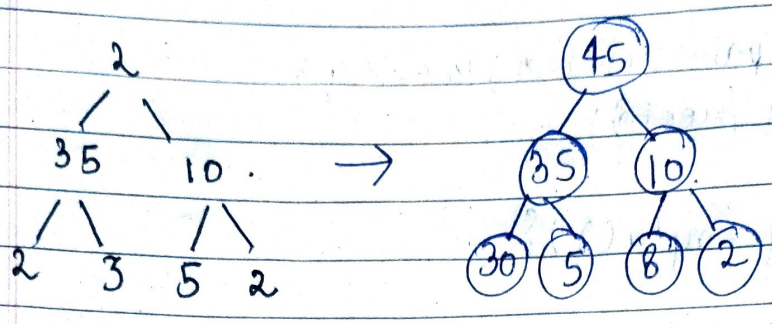


Width = last - first + 1 = 4 - 1 + 1 = 4

TC - O(N)
SC - 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_idx;
            if(i==sz-1) last = curr_idx;

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

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

    return ans;
}
```

L.29 Children Sum Property



I/P O/P

Given a BT, maintain the children sum property. At any node its value should be left child + right child. You can't chang the structure. You can increase the node→val;

node = left + right

1) Increement any node by +1 any no of times

$2 \neq 35 + 10$

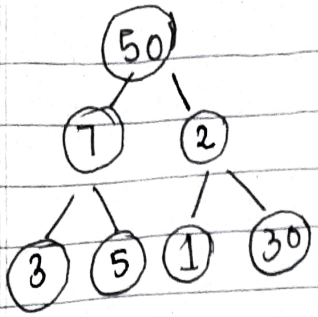
↑
increement 2.

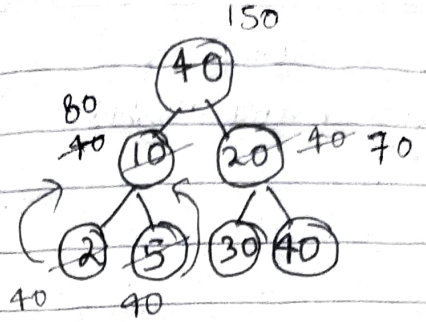
$45 = 35 + 10$

$35 \neq 2 + 3$

↓ ↓
30 5.

Some of the corner cases:

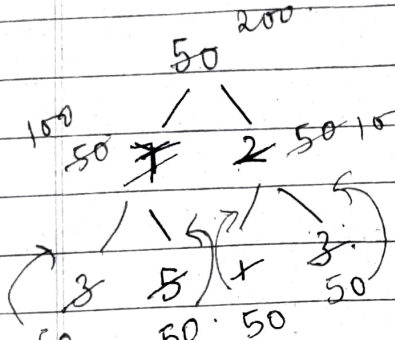




$10+20 = 30 < 40$

$2+5 = 7 < 40$

$30+40 = 70 > 40$



$7+2 = 9 < 50$

$5+3 = 8 < 50$

$1+3 = 4 < 50$

$TC \rightarrow O(N)$

$SC \rightarrow O(H)$ or $O(N)$

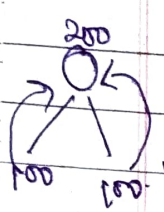
↓ for skew

```
void changeTree(Node<int>* root) {
    if(root == NULL) return;
    int child = 0;

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

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

    }
```

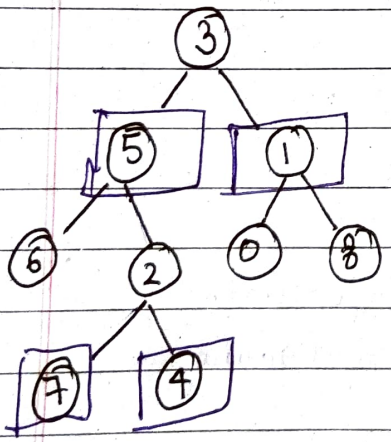
```
changetree(root->left);
changetree(root->right);

int tot = 0;

if(root->left)  tot += root->left->data;
if(root->right) tot += root->right->data;

if(root->left || root->right) root->data = tot;
```

(to check leaf node toh nahi hai)

}



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



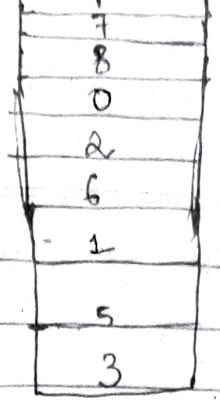
Print all the nodes that are a distance k from a given node.

5 → 7, 4, 1

Nodes can be at the top or bottom

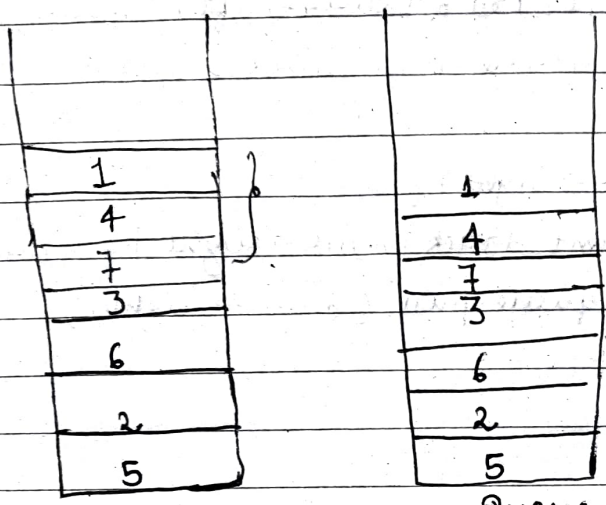
We can't travel towards the parent like from 5 to 3. We would mark parent nodes. We would do BFS traversal.

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



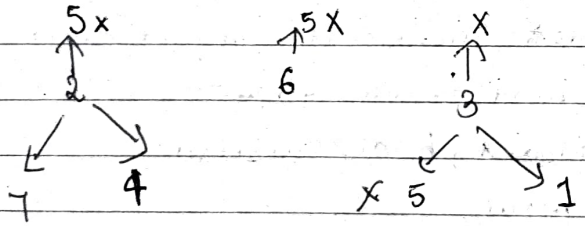
Queue
(BFS)

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



dis = 0 1 2.

2 6 3.



1) (•) parent pointer
2) ↑ distance (radially traverse upwards/downwards by a distance of 1 everytime)
↙ ↘

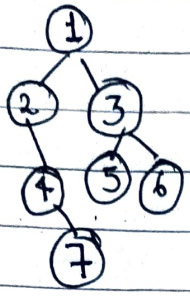
TC → O(N) + O(N) = O(N)
SC → O(N) + O(N) + O(N) = O(N)

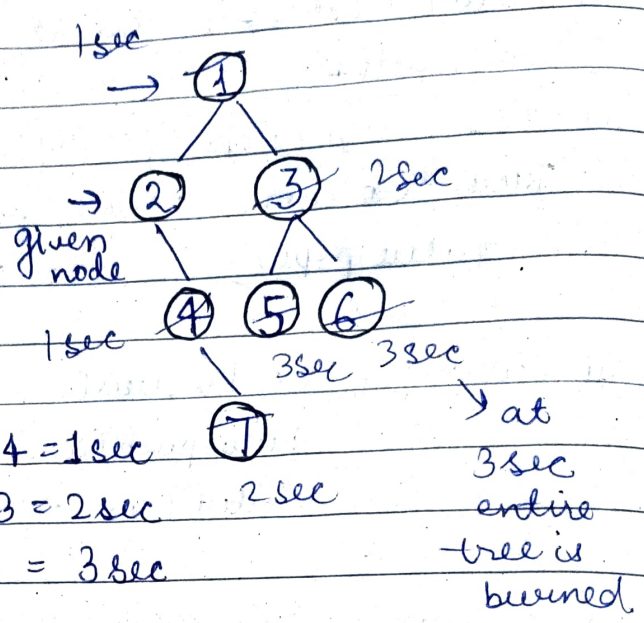
```
void markParents(TreeNode* root, unordered_map<TreeNode*, TreeNode*> &parent_track,
                 TreeNode* target) {

    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){
    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

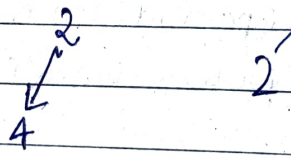




Time to burn 1, 4 = 1sec
Time to burn 7, 3 = 2sec
Time to burn 5, 6 = 3sec

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

Now the problem is 2 can't go upwards as



Maintain the parent pointers using hashmap.

2→1
3→1
4→2
5→3
6→3
7→4

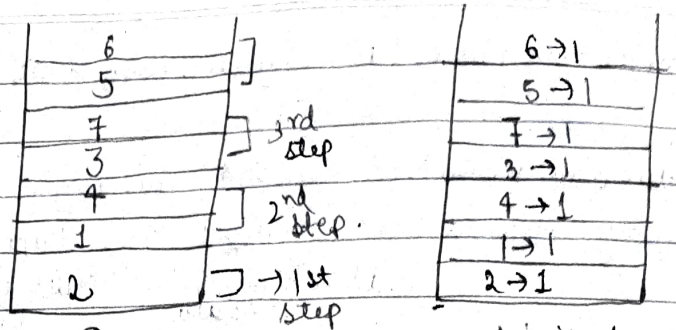
map
(parent, node)

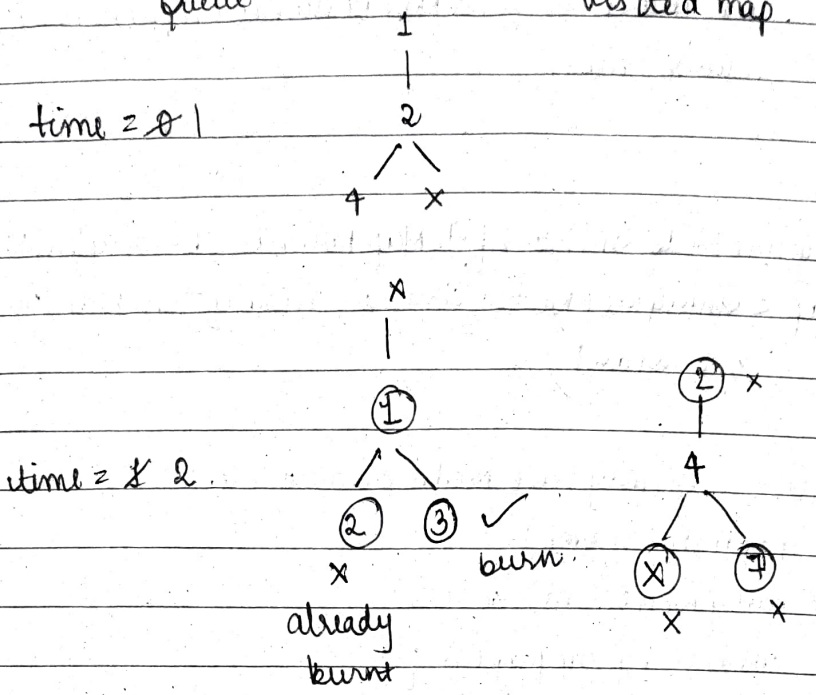
1 2 3 4 5
6

6
5
4
3
2
1

Queueue

Step 2.





Now the queue is empty.

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

TC → O(N) + O(N) = O(2N) ≃ O(N)
SC → O(N) + O(N) = O(N)

```
int timeToBurnTree(BinaryTreeNode<int>* root, int start) {
    map<BinaryTreeNode<int>*, BinaryTreeNode<int>*> mpp;
    BinaryTreeNode<int>* target = bfsToMapParents(root, mpp, start);
                                  // address of given data   // all the nodes parents
    int maxi = findMaxDistance(mpp, target);
    return maxi;
}

BinaryTreeNode<int>* bfsToMapParents(BinaryTreeNode<int>* root,
    map<BinaryTreeNode<int>*, BinaryTreeNode<int>*> &mpp,
    int start) {
    queue<BinaryTreeNode<int>*> q;
    q.push(root);
    BinaryTreeNode<int>* res;
    while(!q.empty()) {
        BinaryTreeNode<int>* node = q.front();

        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;
}
```