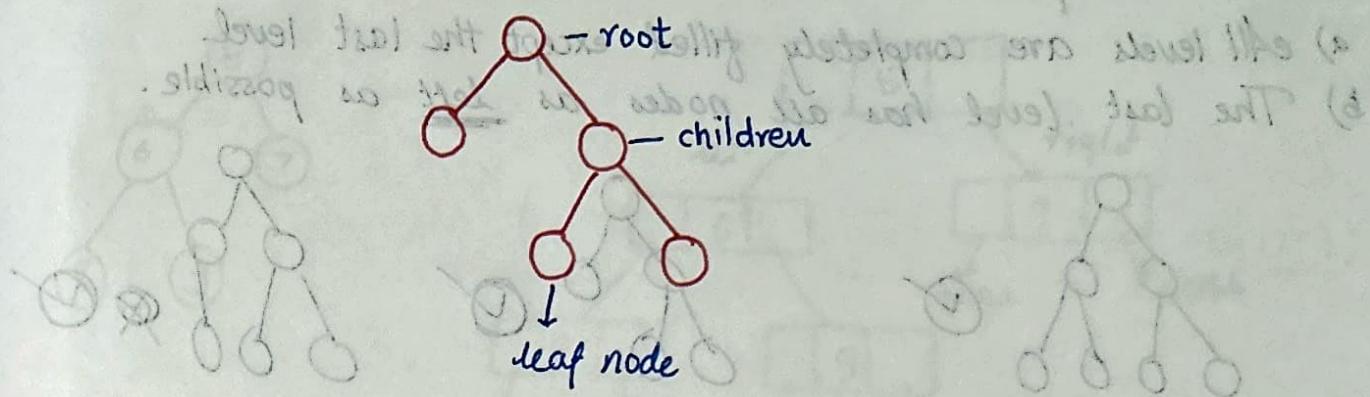
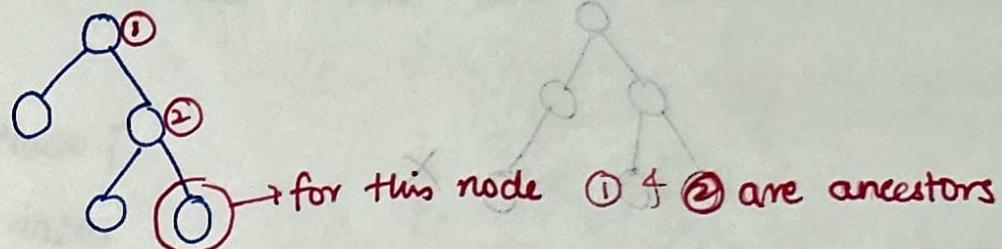


## Introduction to trees :-

:- sort pyramid diagram

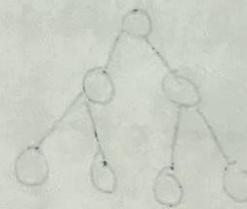


## Ancestors:-



## # Types of binary trees :-

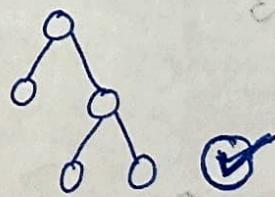
- ① Full binary tree
- ② Complete binary tree
- ③ Perfect binary tree
- ④ Balanced binary tree
- ⑤ Degenerate tree



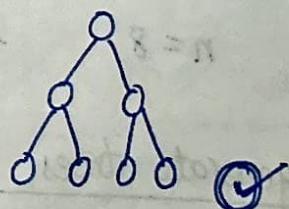
## ① Full binary tree :-

(n) pal xam to sort to nish.

Either have zero or two childrens.

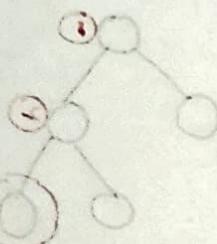
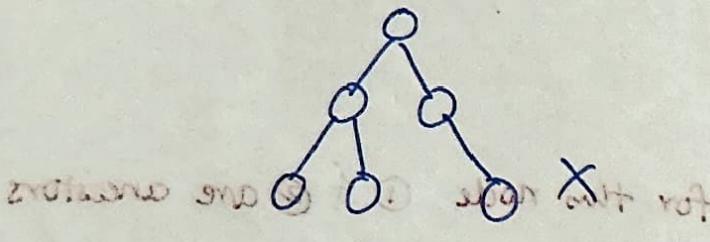
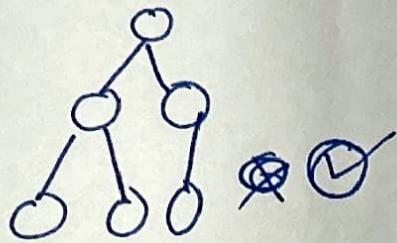
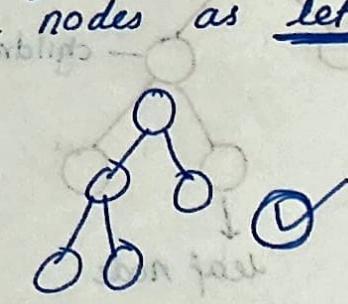
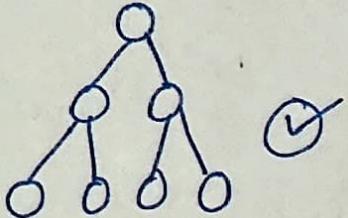


$E = 8, \text{pal}$



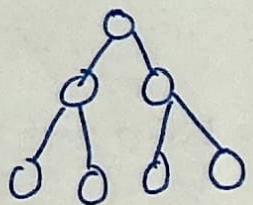
## ② Complete binary tree :-

- a) All levels are completely filled except the last level.
- b) The last level has all nodes as left as possible.



## ③ Perfect binary tree :-

All leaf nodes are at same level

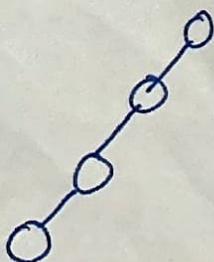


## ④ Balanced binary tree :-

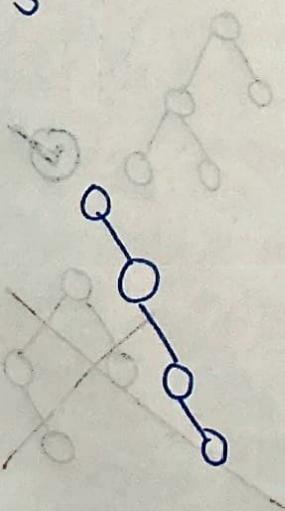
Height of tree at max  $\log(n)$

$$n=8 \quad \log_2 8 = 3$$

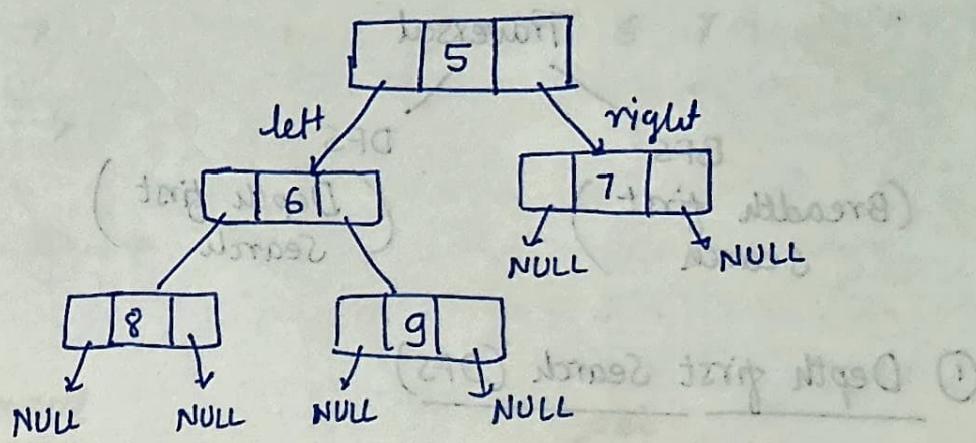
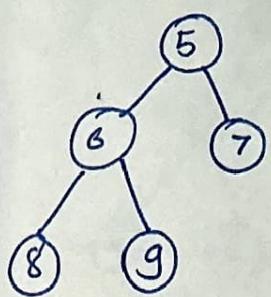
## ⑤ Degenerate trees :-



Skewed



## Binary tree representation in C++ :-

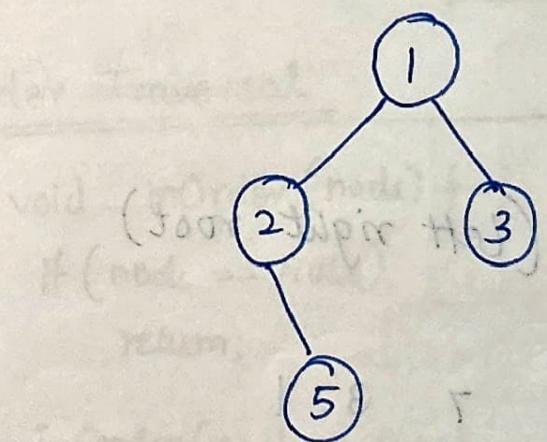


```

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

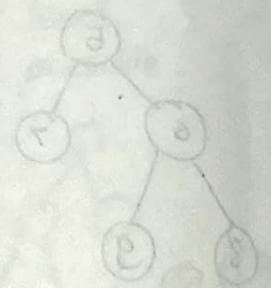
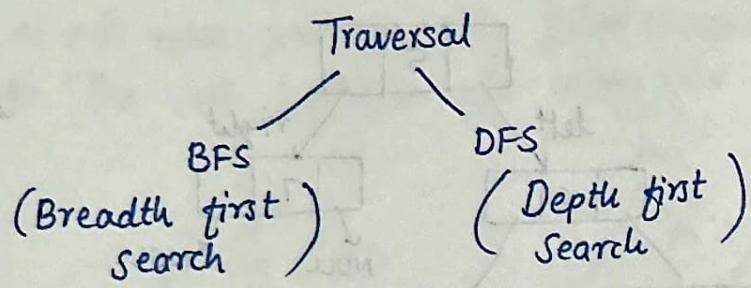
```

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



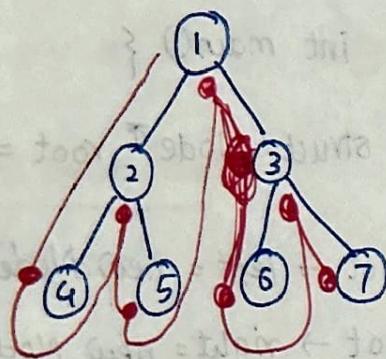
Locksworit rebro - tag

## Traversal Techniques



### ① Depth first Search (DFS)

① Inorder traversal :- (Left - Root - right)



4 2 5 1 6 3 7

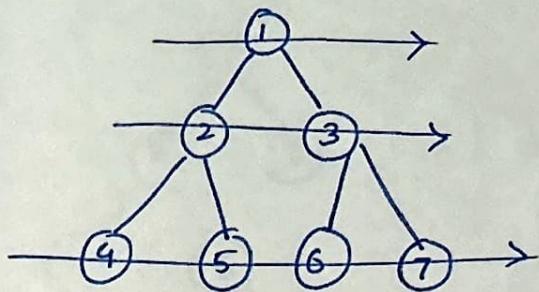
② preorder traversal (root left right)

1 2 4 5 3 6 7

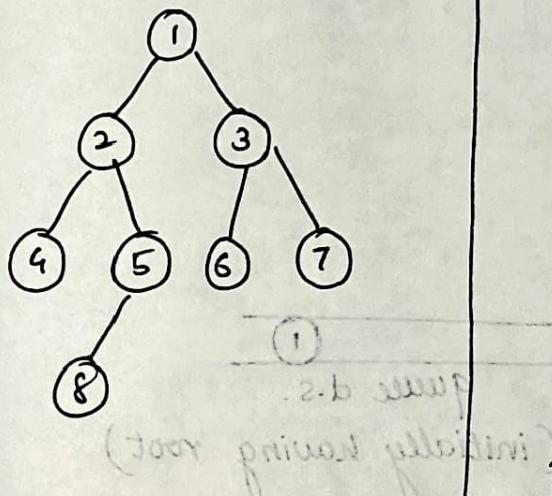
③ post-order traversal (left right root)

4 5 2 6 7 3 1

## ② Breadth First Search



## ① Pre-Order Traversal



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

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

$$S.C = O(N)$$

## ② Inorder Traversal

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

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

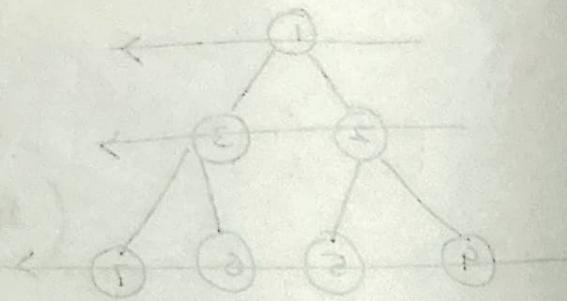
$$S.C = O(N)$$

### ③ Post Order Traversal

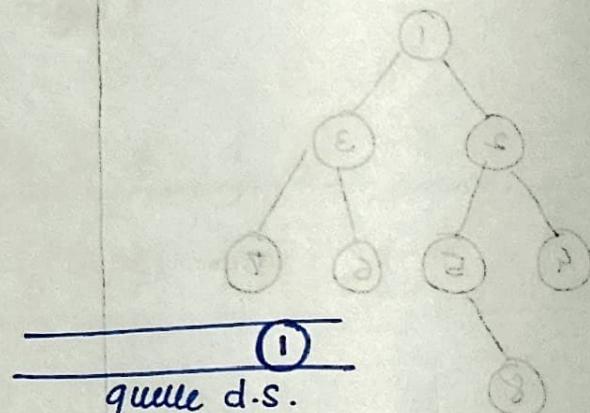
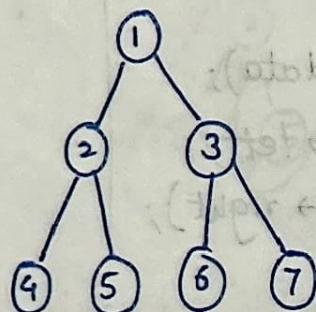
```

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

↑ or



### # Level-Order traversal:



Start

(1) 0 = 0.T  
Take out = 0  $\Rightarrow$

① Now check left side exists for this 1

Yes! put in queue

(1) 0 = 0.T

(1) 0 = 22

[4, 5, 6, 7]

[2, 3]

[1]

vector

3  
2  
1  
Q

Now do same for 2 4 3

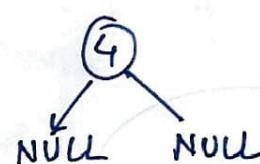
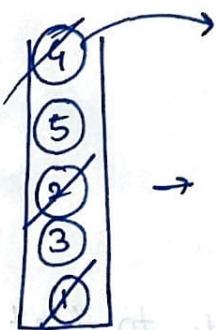
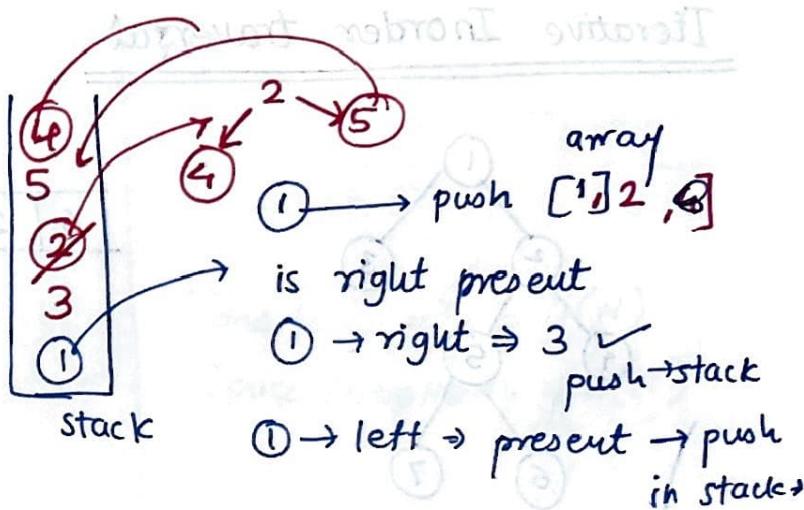
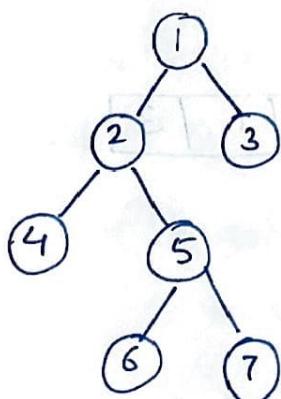
for ②

7  
6  
5  
4



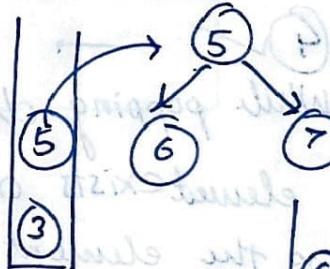
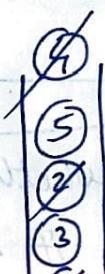
Whenever empty  $\rightarrow$  Completed !!

## Pre-Order traversal (Iterative)



→ pop this and  
push into array

[1, 2, 4]



⇒ Push 5 in ans.

[1, 2, 4, 5]



While doing the level-order traversal

→ we have to iterate over the size of the queue and push left and then right

But in preorder iterative traversal,

Use stack and

push the RIGHT node first and then left node.

vector<int> ans;

stack<TreeNode\*> st;

st.push(root)

while(!st.empty()) {

root = st.top();

st.pop();

ans.push\_back(root->val);

if (root->right) → push into st.

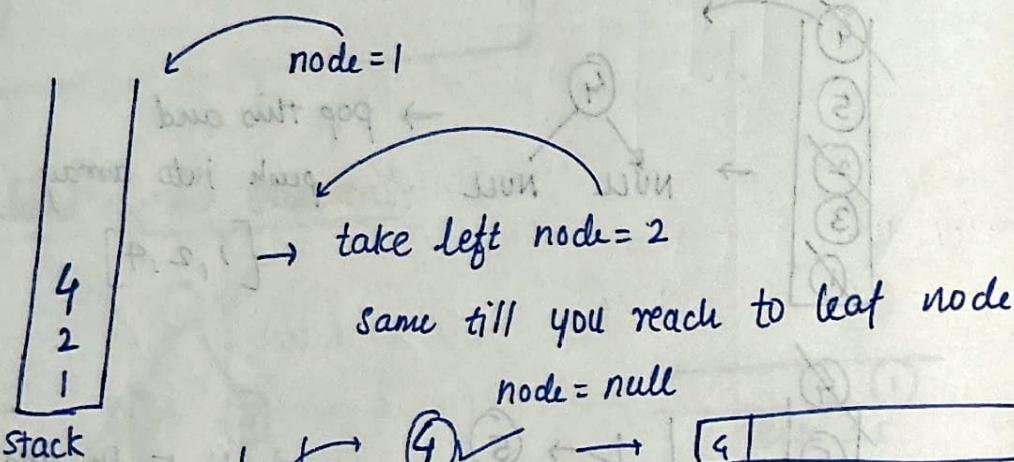
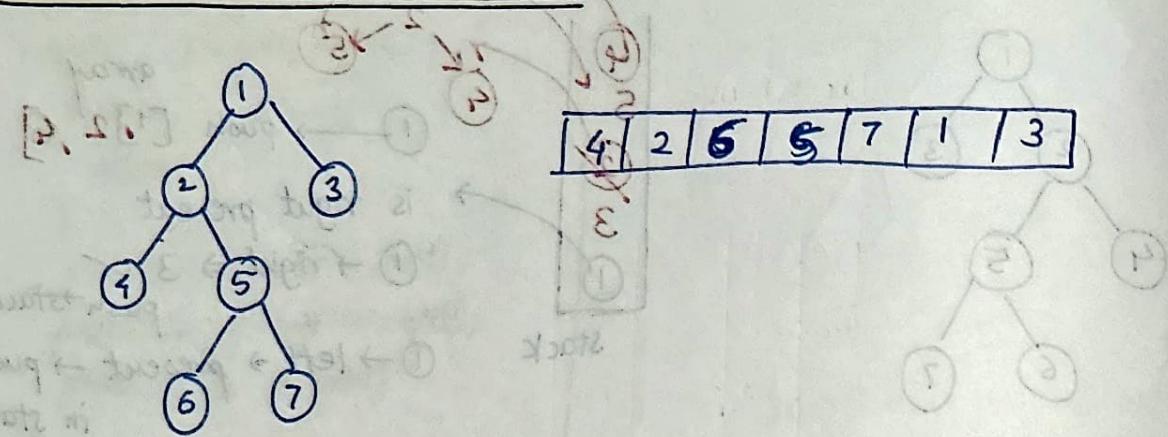
if (root->left) → push into st,

}

return ans;

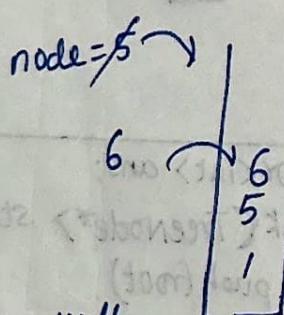
## Iterative Inorder traversal

(written II) Learning notes 0-99



while popping check whether right element exists or not if it is then pop the element and assign the right to node = 5

4	2
---	---



4	2	6	5
---	---	---	---

node = 7



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

likethis.

```

stack <TreeNode*> st;
vector <int> result;
TreeNode* node = root; // -> node

```

```
while(true) {
```

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

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

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

```
    } else {
```

```
        if(st.empty()) break;
```

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

```
        st.pop(); result.push_back(node->val);
```

```
        node = node->right;
```

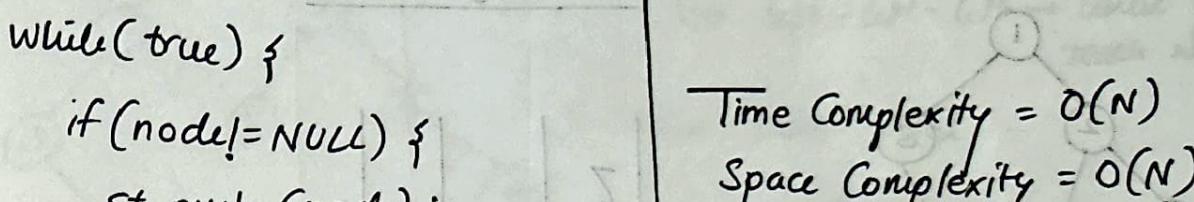
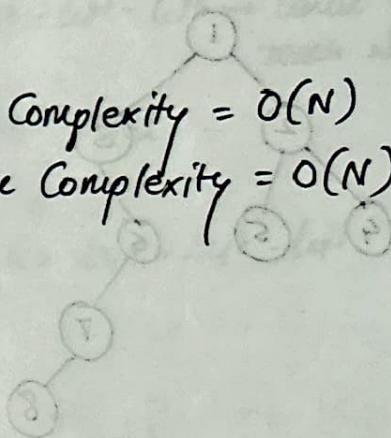
```
    }
```

```
}
```

```
return result;
```

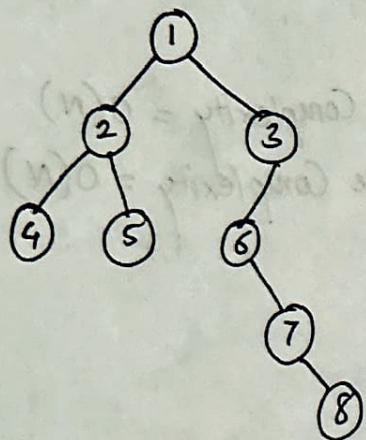
Time Complexity =  $O(N)$

Space Complexity =  $O(N)$

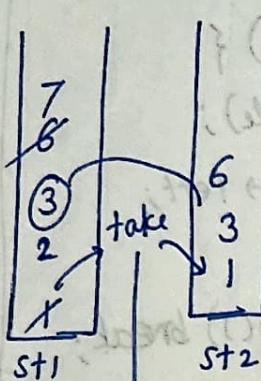


## • Iterative post-order traversal

### ① Using Two-stacks



Left - Right - Root

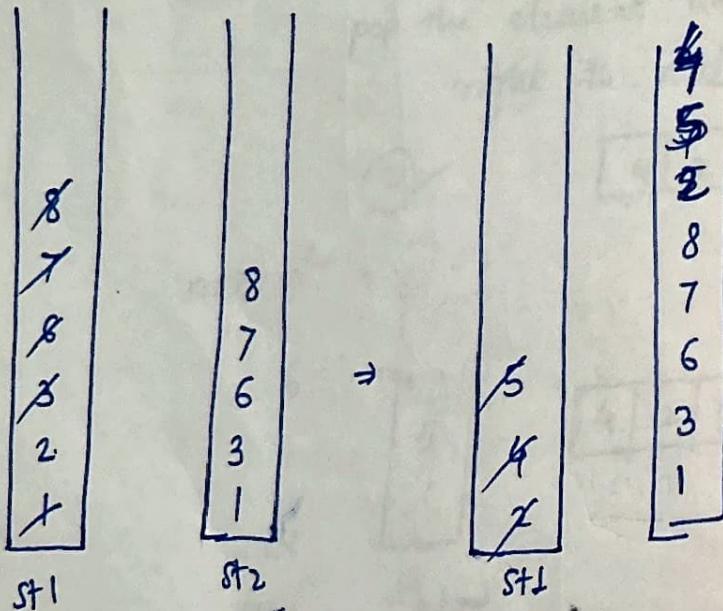


Check if '1' has left  $\rightarrow$  yes push into 'St1'  
check if '1' has right  $\rightarrow$  \_\_\_\_\_

$\Rightarrow 3 \rightarrow \text{left} \rightarrow 6 \rightarrow \text{push in } St1$

$\Rightarrow 6 \rightarrow \text{right} \rightarrow 7 \rightarrow \text{push in } St1$

Perform this



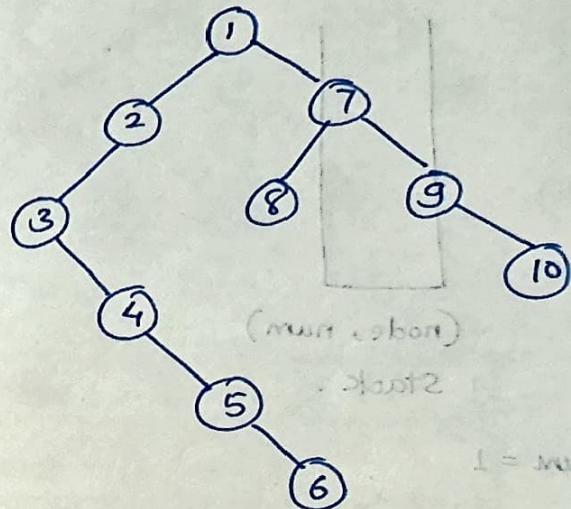
4	5	2	8	7	6	3	1
---	---	---	---	---	---	---	---

This is post order

```

vector<int> postorder;
if (root == NULL) return postorder;
Stack<Node*> St1, St2;
St1.push (root);
while (!St1.empty()) {
    root = St1.pop();
    St1.pop();
    St2.push (root);
    if (root->left)
        St1.push (root->left);
    if (root->right)
        St1.push (root->right);
}
while (!St2.empty()) {
    postorder.push_back (St2.top()->data);
    St2.pop();
}
  
```

## ② Using One stack



Go left - left - left → until you reach Null  
 then take right  
 AGAIN Go left - left - left

curr = 1 ← root

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

if (curr == NULL) {

st.push(curr);

curr = curr → left;

} if of return <

else {

temp = st.top() → right; i →

if (temp == NULL) {

temp = st.top();

st.pop();

duplication of return

postOrder.push\_back(temp → val);

while (!st.isEmpty() && temp == st.top() → right) {

temp = st.top();

st.pop();

} postOrder.push\_back(temp → val);

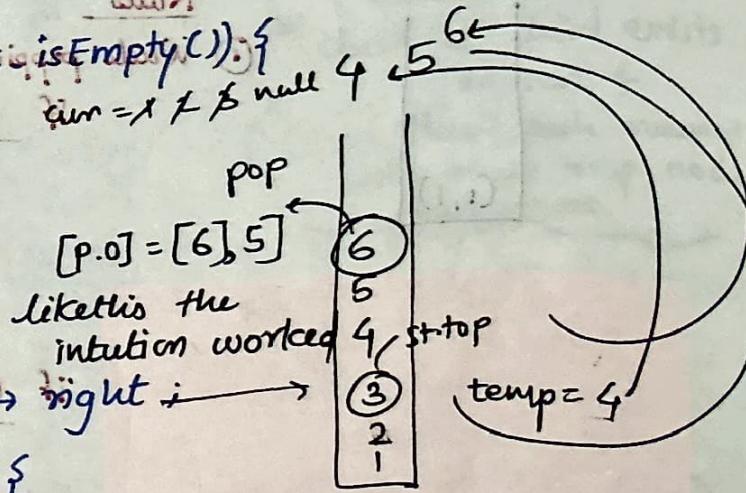
}

else

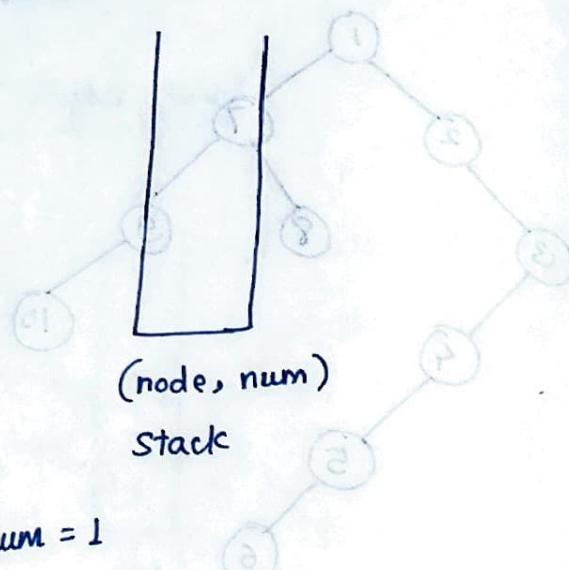
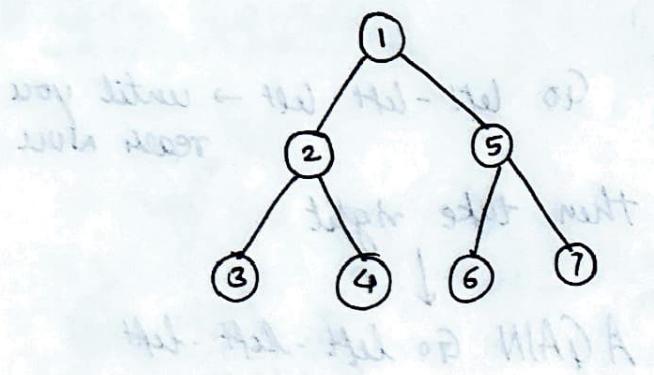
curr = temp;

}

}



## PreOrder | InOrder | PostOrder in One traversal



At first push the root node with num = 1

(1, 1)

### Rules

① While popping the element from stack

if num == 1

↳ PRE-ORDER

push that num++  
if (left) exists

↳ enter to left

if (num == 2)

↳ IN-ORDER

Increment num++

if (right)

↳ enter to right

While ~~skipping~~ the num(1/2/3)

```

if (num == 1) {
    push (into preOrder)
    do num++;
    push again into stack
    check if (left is present)
    push (left, 1)
    ↳ num
}
  
```

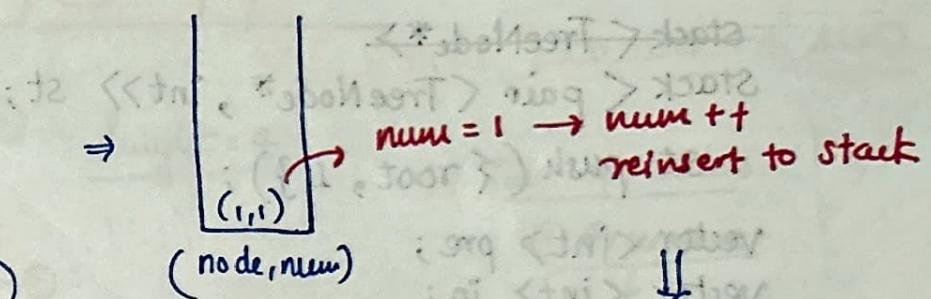
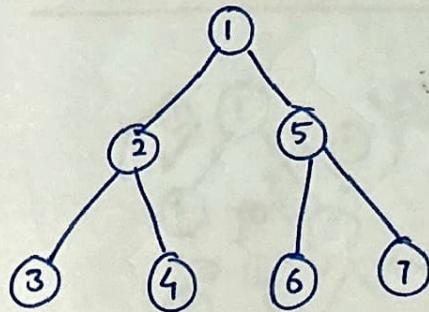
Same for num == 2

```

push it into inOrder
do num++;
push again into stack
check if right is present
→ push (right, 1)
  
```

else simply push-back  
to the postOrder <>

## DRY RUN



PreOrder : 1 2 3 4 5 6 7

InOrder : 3 2 4 1 6 5 7

PostOrder: 3 4 2 1 6 7 5

$\text{num} = 2$   
INORDER  
 $\text{num}++ = 3$   
 check right  
 $\downarrow$   
 No

(3, 1)  
(2, 2)  
(1, 2)

(5, 1)  
(2, 2)  
(1, 2)

! pop-it  $\rightarrow$  check num  
 increment it  
 check if child exists  
 or not &  
 then push num=1  
 with their resp. node  
~~add it to~~

(3, 3)  
(2, 2)  
(1, 2)

(3, 2)  
(2, 2)  
(1, 2)

POST ORDER

pop the element  
& node

$\text{num} = 3$

(4, 1)  
(2, 2)  
(1, 2)

INORDER  $\therefore 2$   
 $\downarrow$   
 $\text{num}++ = 3$   
 check right  
 $\downarrow$   
(4, 1)

(4, 1)  
(2, 3)  
(1, 2)

$\Rightarrow \text{num} 2$

INORDER

post order

$\downarrow$   
 $\Rightarrow (4, 3)$

likewise  
you have  
to  
perform

(1, 3)

post post  $\rightarrow$  pop  
Inorder

(5, 1)  
(1, 2)

$\rightarrow$  post  $\rightarrow$  pop  
 $\Rightarrow$

$\Rightarrow$

Time Complexity =  $O(N)$

Space Complexity =  $O(N)$

```
stack <TreeNode*>
```

```
Stack < pair <TreeNode*, int>> st;
```

$\dagger \text{num} \leftarrow 1 = \text{num}$

root or root.push({root, 1});

```
vector <int> pre;
```

```
vector <int> in;
```

```
vector <int> post;
```

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

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

auto it  
~~TreeNode\*~~ top = st.top();

were here <= st.pop();

$\dagger$  traversal

time block for

$\dagger$  far to

$\dagger$  num next  
more long next  
more que next via  
 $\dagger$  time

```
if (top
```

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

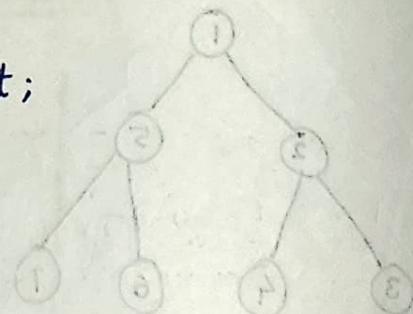
$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time

$\dagger$  far to  
more long next  
more que next via  
 $\dagger$  time



7 6 5 4 3 2 1 : rebreak

7 5 3 1 2 4 6 : rebreak

3 5 2 1 4 6 7 : rebreak

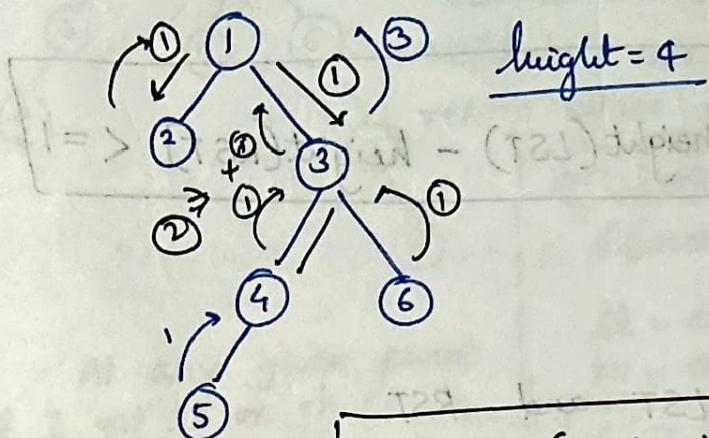
(E,E)  
(S,S)  
(S,I)

E = NULL

RETRY T209

## • Maximum Depth of Binary Tree

Chalk for 2nd year  
Topic: Maximum Depth of Binary Tree



Two approaches

① Recursive

Auxiliary Space

② Iterative

Queue D.S.

SC.  $\Rightarrow O(N)$

$$1 + \max(\text{left}, \text{right})$$

At ④  $\rightarrow 1 + \max(1, 0)$

$\Rightarrow 1 + 1 = 2 \rightarrow$  for 4  $\rightarrow \text{height} = 2$

for ⑥  $\rightarrow 1 + \max(0, 0) \rightarrow 1$

for ③  $\rightarrow 1 + \max(\text{left}, \text{right})$

$(0) \quad (1) \quad 1 + \max(2, 1) \Rightarrow 1 + 2 = 3$

```
int maxDepth(TreeNode* root) {
    if (root == NULL) return;
    int leftHeight = maxDepth(root -> left);
    int rightHeight = maxDepth(root -> right);
    return 1 + max(leftHeight, rightHeight);
}
```

~~sort priority to height maximum~~

## • Check for balanced binary Tree :-

Balanced binary Tree :-

$$\text{height(LST)} - \text{height(RST)} \leq 1$$

Naive solution :-

Compute the height of LST and RST  
and check the difference.

pseudo code

```
check(node) {  
    if (node == null) return true;
```

$Lh = \text{findHeight}(\text{node} \rightarrow \text{left});$        $O(n)$   
 $Rh = \text{findHeight}(\text{node} \rightarrow \text{right});$        $O(n)$        $\Rightarrow O(n^2)$

```
    if ( $|\text{abs}(Lh - Rh)| > 1$ ) return false;
```

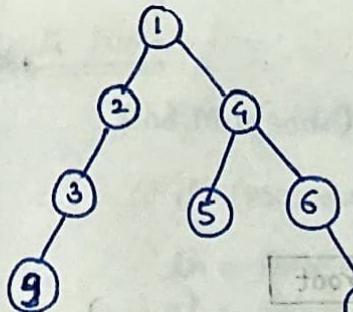
```
    Bool left = check(\text{node} \rightarrow \text{left});  
    Bool right = check(\text{node} \rightarrow \text{right}), } →
```

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

```
    return true;
```

}

we have to  
check for  
every node.



Optimal code

~~lets dry run~~ The idea is to add the conditions. So before we were returning the true/false. but now instead of this we will return either -1 or height.

(Any tree won't do)  
Wow very good!!

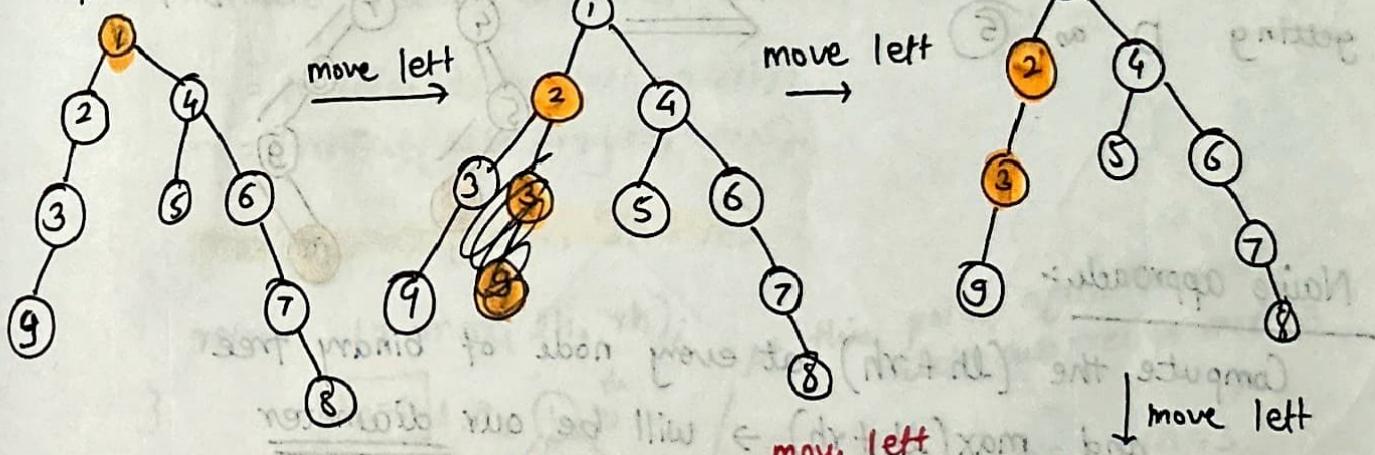
At any given point  
if I got lh or rh as  
-1 return -1

return -1 if  
absolute diff exceeds

```
int check(node) {
    if(node == null) return 0;
    lh = check(node->left);
    rh = check(node->right);
    if(lh == -1 || rh == -1) return -1;
    if(abs(lh-rh) > 1) return -1;
    return max(lh,rh) + 1
}
```

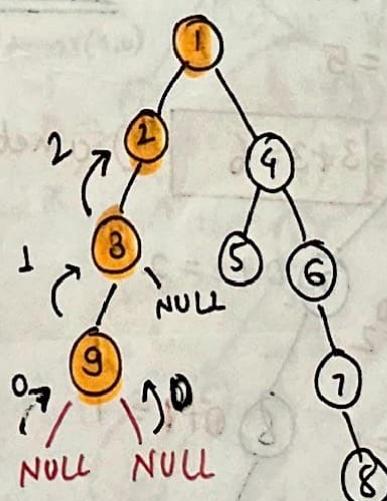
T.C = O(N) → Recursive traversal  
S.C = O(N) → Auxiliary space for skewed tree

Dry run :-



At ③  
from left = 1  
from right = 0

$$\max(1,0) + 1 = 1 + 1 = 2$$

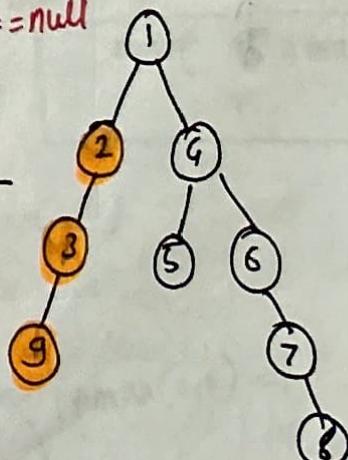


move leftt  
here, node == null  
return 0;

$$\Rightarrow lh=0$$

$$\Rightarrow rh=0$$

$$\text{return } \max(0,0)+1 = 1$$



At ② → the height

$$lh=2 \\ rh=1$$

$\rightarrow \text{if}(\text{abs}(lh-rh)>1) \text{return } -1$   
This condition is executed!

Tree is ~~not~~ not a balanced binary tree!

## Diameter of binary tree

Diameter :-

→ Longest path between 2 nodes.

→ Path does not need to pass via root  
optional (It can pass or may not pass)

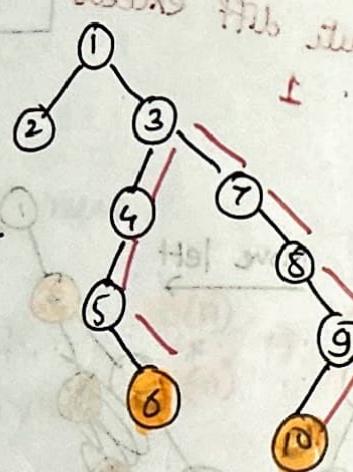
↓  
This is tricky I'll show you How?!

(Height of left & right roots = 3)

(Height of 3 = 1)  $\Rightarrow$  3

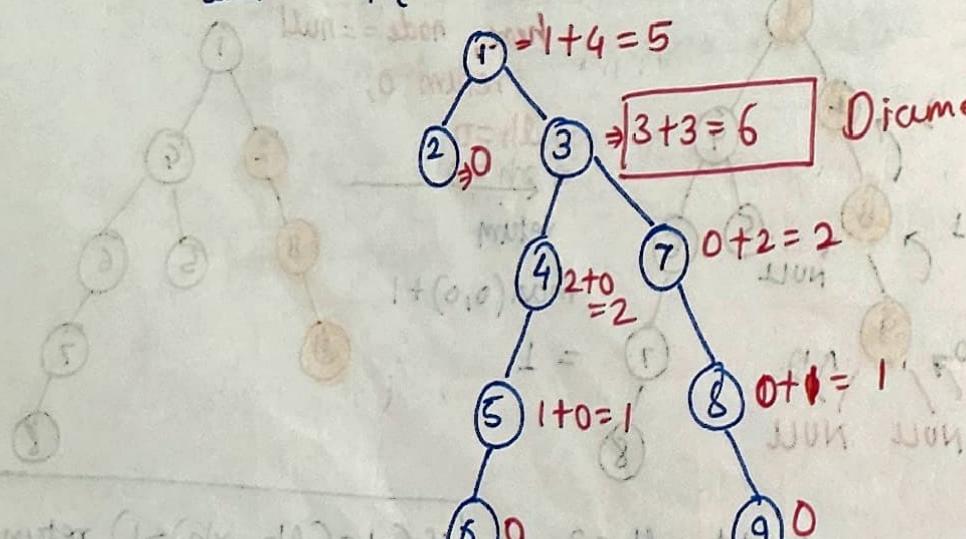
(So you might think we should always pass through root node)

If I would have chosen the path which passes through root then I would have ended up getting D as 5



Naive approach :-

Compute the  $(lh + rh)$  at every node of binary tree and  $\max(lh + rh) \rightarrow$  will be our diameter



- diameter  $(\leq (N-1)/2)$   
(between 2 non-blanks out)

sort priority bounded to tail of string

## Brute Force:-

findMax(node) {

    if (root == null) return 0;

    lh = findLeftH(node → left);

    rh = findRightH(node → right);

    maxi = max(maxi, lh + rh);

    findMax(node → left); ] → Go to left and right

    findMax(node → right); and compute the max height  
    EVERYTIME!!

now think about to sing you by

T.C = O(N<sup>2</sup>)

## Optimal ⇒

int findMax(node, maxi) {

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

    int lh = findMax(node → left, maxi);

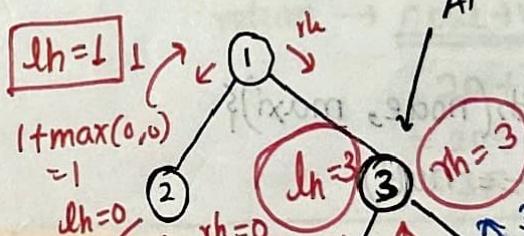
    int rh = findMax(node → right, maxi);

    maxi = max(maxi, lh + rh);

    return 1 + max(lh, rh);

At this point after traversal  
left sub tree  $\Rightarrow$  maxi = 3

3



lh = findLeftH(node → left);

rh = findRightH(node → right);

maxi = max(maxi, lh + rh);

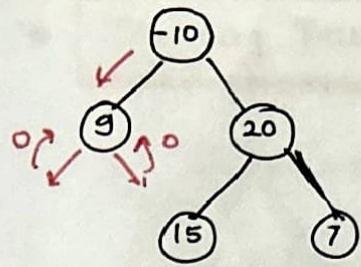
findMax(node → left); ] → Go to left and right  
and compute the max height  
EVERYTIME!!

T.C = O(N<sup>2</sup>)

now think about to sing you by

(mult1, mult2, abon) N<sup>2</sup> to O(N<sup>2</sup>)





At ⑨

$$\text{maxi} = \max(0, 9 + 0 + 0)$$

$$= \max(0, 9)$$

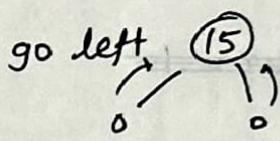
$$\text{maxi} = 9$$

$$\text{return } \rightarrow 9 + \max(0, 0) = 9$$

return to -10

At ⑩ → leftSum = 9

right, ⑯ → go left



$$\text{maxi} = 9 \text{ (old)}$$

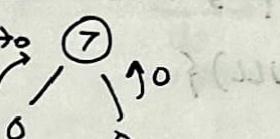
$$\text{maxi} = \max(9, 15 + 0 + 0)$$

$$= \max(9, 15)$$

$$\text{maxi} = 15 \rightarrow \text{returned } 15$$

$$\text{return } 15 + \max(0, 0)$$

⑯ → right



$$\text{maxi} = 15 \text{ (old)}$$

$$\text{maxi} = \max(15, 7 + 0 + 0)$$

$$= \max(15, 7)$$

$$= 15 \rightarrow \text{return to } ⑯$$

$$\text{return } \rightarrow 7 + \max(0, 0) \rightarrow \text{return to } ⑯$$

leftSum = 15  
rightSum = 7

rightSum = 7

$$\text{maxi} = 15$$

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

$$= \max(15, 42)$$

$$\text{maxi} = 42$$

return → node + val → ---

$$20 + \max(15, 7)$$

$$= 20 + 15$$

$$= 35 \rightarrow \text{return to } (-10)$$

on  
leftSum = 9  
rightSum = 35

$$\text{maxi} = 35 \text{ (old)}$$

$$\text{maxi} = \max(-10, -10 + 35 + 9)$$

$$= \max(-10, 34)$$

$$= 34$$

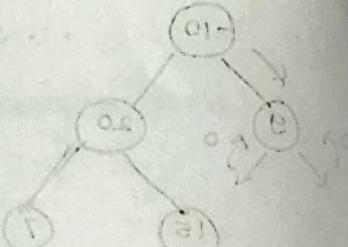
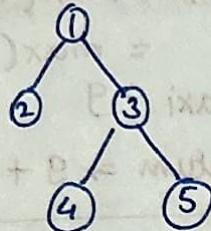
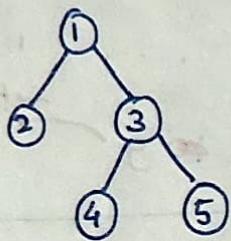
Don't update

T.C = O(N)  
S.C = O(N)

return the maxi = 42

from main function

## Same Tree or not :



preOrder = Traverse in both the ~~preOrder~~ trees at the same time

```

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

(b)  $2E = IXOM$

(e + 2E + 0, 0)  $= IXOM$

(4E, 0)  $=$

$pE =$

stibgu + no(

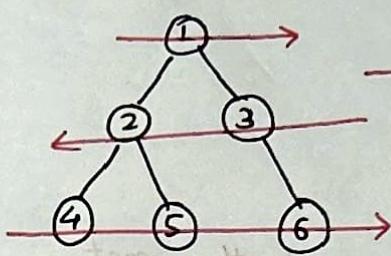
$SP = IXOM$  ent water

nothing more

(H)O = 3T

(H)O = 3B

## Zig-Zag Traversal

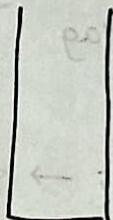
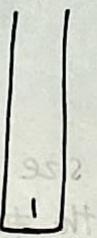


1 2 3 4 5 6 → This is zig-zag pattern



:

new node → (root) → (left) → (right) → (left) → (right) → ...



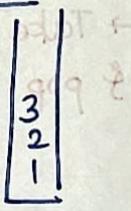
ds vector<vector>

direction = 0

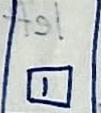
{ "0" → LEFT → RIGHT  
"1" → RIGHT → LEFT }

Dry Run ::

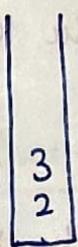
1 → left → 2 → push into Q  
1 → right → 3 → push into Q



Once done  
remove 1 from Q and put it into ds



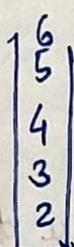
Now reverse the direction → 1



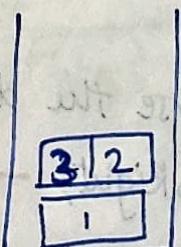
2 → 2 → left → 4 → push → Q  
2 → right → 5 → push → Q

3 ⇒ 3 → left → NULL

3 → right → 6 → push → Q



Once done → from right to left



Likewise

$$(1) O = 3T$$

$$(2) O = 3Z$$

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

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

```
    if (root == NULL) {  
        return result;  
    }
```

```
    queue<TreeNode*> nodesQueue;
```

nodesQueue.push(root);  $\rightarrow$  Initially push the root

```
    bool leftToRight = true;  $\rightarrow$  flag
```

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

```
        int size = nodesQueue.size();  $\rightarrow$  find the size E.g.
```

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

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

```
            TreeNode* node = nodesQueue.front();  $\rightarrow$  Take front
```

```
            nodesQueue.pop();
```

ab

// find position to fills node value

int index = leftToRight ? i : (size - i - 1);  $\rightarrow$  Check direction

row[index] = node  $\rightarrow$  val  $\rightarrow$  push basic on direction

```
if (node  $\rightarrow$  left) {
```

nodesQueue.push(node  $\rightarrow$  left);  $\boxed{}$  push the left node

```
}
```

```
if (node  $\rightarrow$  right) {
```

nodesQueue.push(node  $\rightarrow$  right);  $\boxed{}$  push the right node

```
}
```

// after this level, reverse the direction

leftToRight = !leftToRight;  $\rightarrow$  Reverse the direction

result.push\_back(row);  $\rightarrow$  put the row in result

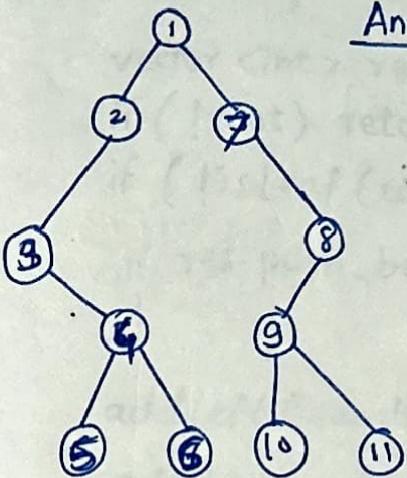
```
}
```

```
return result;
```

3

T.C = O(N)  
S.C = O(N)

## • Boundary Traversal :-



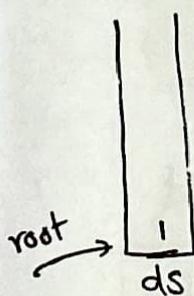
Anticlockwise

~~+ 2 4 5 6 7~~

Left Boundary (Excluding leaves)

Leaves (prefer Inorder)

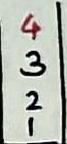
Right Boundary in the reverse + Exclude leaf nodes



Start with left boundary

node → left → put it into ds ⇒

check again node → left



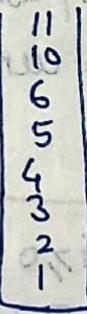
If there doesn't exist left → go to right

then check the node → left ⇒ 5 but this

is leaf nod  
so SKIP

for leaf nodes  
(INORDER TRAVERSAL)

Root left Right ⇒ 5 6 10 11



## Right Boundary In Reverse



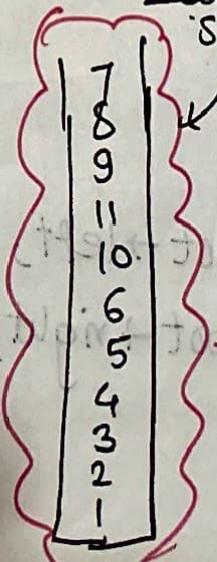
go to right 8 ⇒ does right exist ??

No

If right doesn't exist

Then move to LEFT

Again check for right node



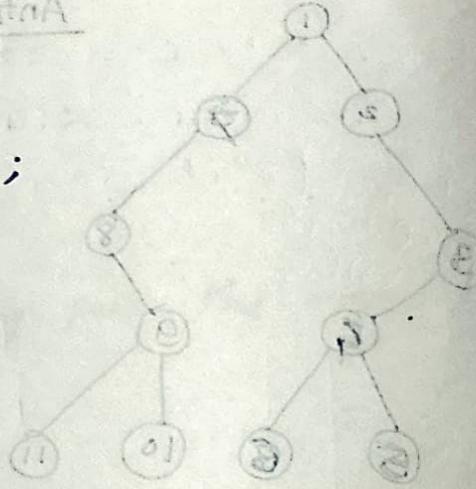
Take from last

This is the boundary traversal.

```

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

```



```
void addRightBoundary(Node* root, vector<int>& res) {
```

```
    Node* curr = root->right;
```

```
    vector<int> tmp;
```

```
    while (curr) {
```

```
        if (!isLeaf(curr)) tmp.push_back(curr->val);
        if (curr->right) curr = curr->right;
        else curr = curr->left;
    }
```

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

Reversing

```
void addLeaves(Node* root, vector<int>& res) {
```

```
    if (isLeaf(root)) {
```

```
        res.push_back(root->val);
        return;
    }
```

```
}
```

```
    if (root->left) { addLeaves(root->left, res); }
```

```
    if (root->right) { addLeaves(root->right, res); }
```

Inorder traversal

Root

Left

Right

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

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

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

```
    if (!isLeaf (root)) {
```

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

```
}
```

```
    addLeftBoundary (root->left, res);
```

```
    addLeaves (root->right, res);
```

```
    addRightBoundary (root->right, res);
```

```
    return res;
```

```
}
```

$$T.C. = O(H) + O(H) + O(N) \approx O(N)$$

height of tree      height of tree      Inorder traversal

$$S.C = O(N)$$



01  $\rightarrow$  (0,0) to see why it is

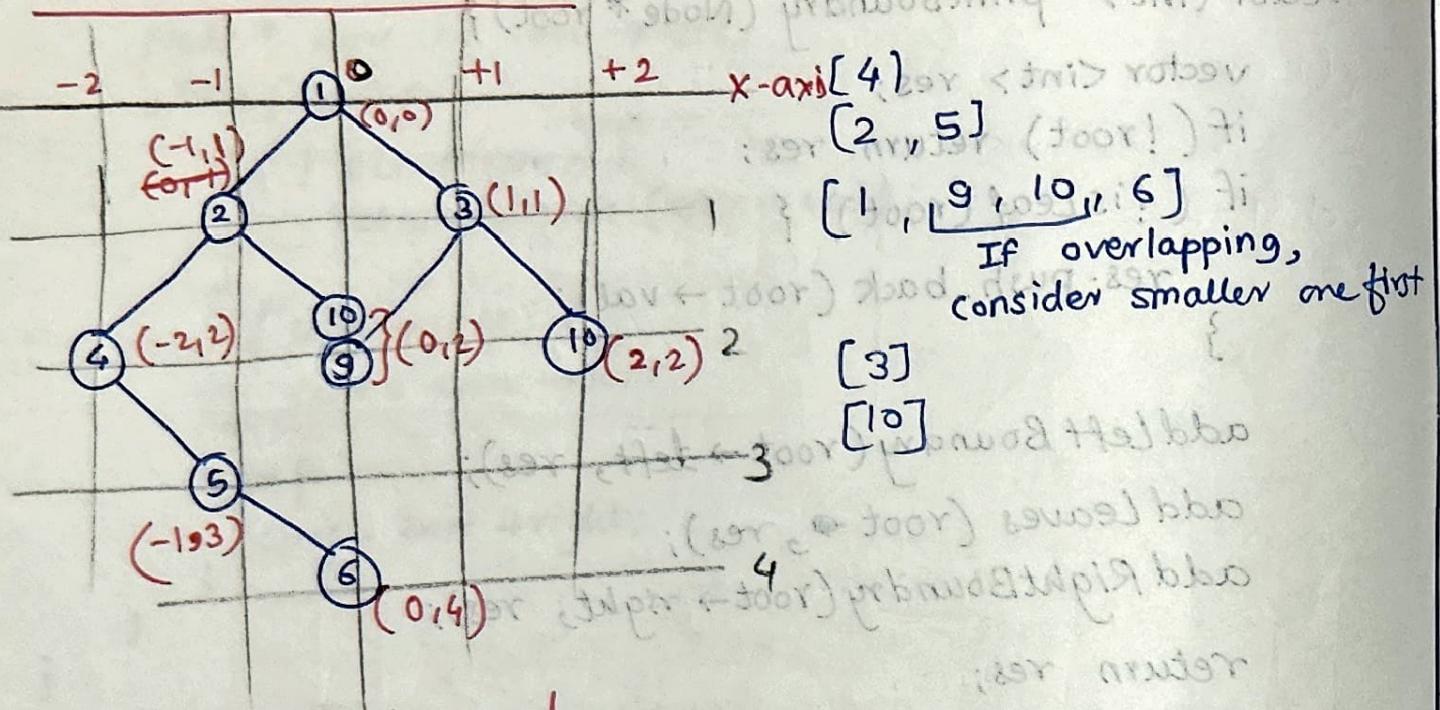
leftmost child  
child is left

leftmost child

because it comes first

no other son exists

## • Vertical Order traversal



Data Structures to be used :-

Queue

Q<node, vertical, level>

map<int, vector<int>>

represent vertical

for every vertical → Why I'm taking  
map!!

so if you see at (0,2) →



At every level there can be multiple nodes I need them into sorted order that is why multiset

Why multiset

Because at each level duplicate node values can present.

node = 1

(1, 0, 0)

Q (node, vertical, level)

ds

map<int, map<>>

Insert into ds

0 → {0} → {1}

This is the form

↓

node = 2 (moved left) → vertical -;  
level + 1;

(node, level)

(1, 0, 0), (2, -1, 1),

Q

↓

move ~~left~~ right

~~node~~

node = 3

(2, -1, 1), (3, 1, 1)

Q

vertical ++;

level ++;

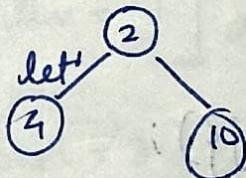
node = 2

vertical = -1

level = 1

→ store into ds

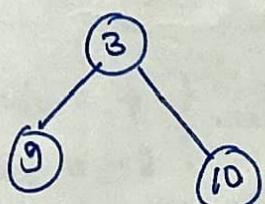
-1 → 1 → {2}  
0 → 0 → {1}



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

Q

-1 → 1 → {3}  
-1 → 1 → {2}  
0 → 0 → {1}



(4, -2, 2), (10, 0, 2), (9, 0, 2), (10, 2, 2)

; like this

left

Right

(-1, +1)

(+1, +1)

## Level order traversal used :

```
vector<vector<int>> verticalTraversal(TreeNode* root) {
    map<int, map<int, multiset<int>> nodes;
```

queue < pair<TreeNode\*, pair<int, int>> todo;  
 todo.push({root, {0, 0}});  
 while (!todo.empty()) {  
 (vertical, level)

auto p = todo.front();

todo.pop();

TreeNode\* node = p.first;

int x = p.second.first; → vertical

int y = p.second.second; → level

nodes[x][y].insert(node->val);

if (node->left) {

todo.push({node->left, {x-1, y+1}});

}

if (node->right) {

todo.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());

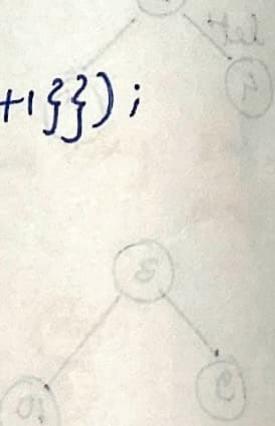
}

ans.push-back(col);

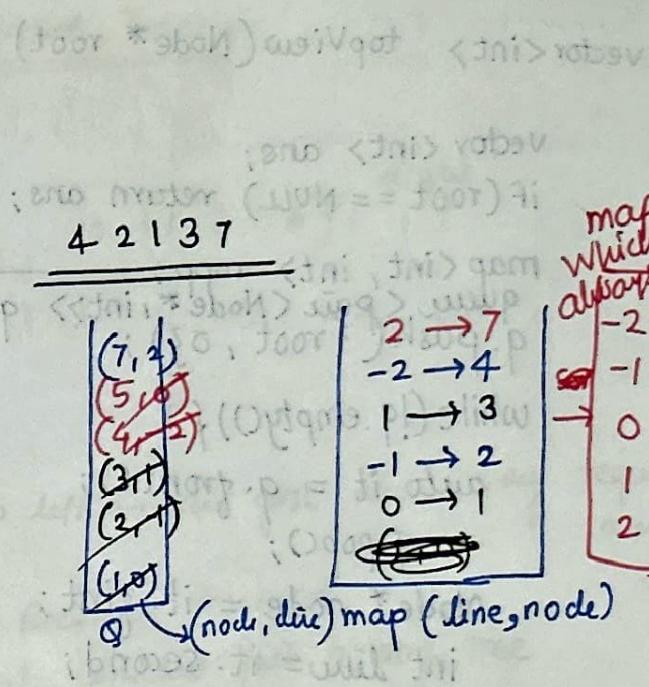
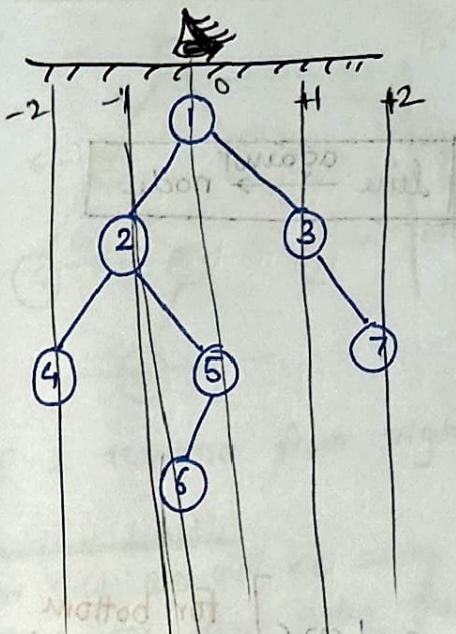
}

return ans;

}



## Top view of Binary tree



node = 1      left ② → present on the -1 line

line = 0      and on right ③ → +1 to line

node = 2      left ④ → do line -1 → -1 -1 = -2

line = -1      and right ⑤ → do line +1 → -1 +1 = 0

node = 3      left → NULL

line = 1      right → ⑦ → do line ++ → 1 + 1 = 2

node = 4      left → NULL

line = -2      right → NULL

node = 5      Now on line = 0 → (0 → 1) "1" is already present hence don't consider  
 line = 0      (0 → 5) ×

node = 7      left = NULL → push (2 → 7) in map

line = 2      Right = NULL

(2, 1) above level 2 from 1 to 2

1 level need to visit below

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

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

```
    while (!q.empty()) {  
        auto it = q.front();  
        q.pop();  
        Node* node = it.first;  
        int line = it.second;  
        if (mpp.find(line) == mpp.end()) {  
            mpp[line] = node->val;  
        }
```

```
        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)

Notes to be discussed

Why the recursive approach is not used?

Why the level-order traversal is used.

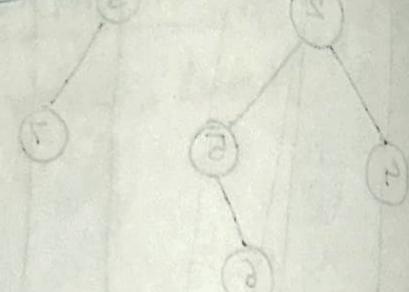
→ If you've used inorder traversal  
then you would have visited

1 2 4 5 6 → This node is at (+1)

so in map the desired node (i.e. 3)

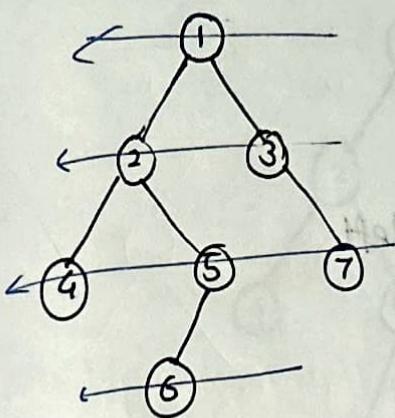
would have not been pushed!!

line against node



For bottom view of the tree we can remove the if block and override the values.

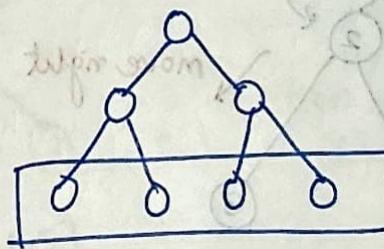
## Right / Left View of Binary Trees :-



(Right view)  $\rightarrow [1, 3, 7]$  (Left view)  $\rightarrow [1, 3, 6]$

If I traverse from right to left  $\rightarrow$  the first node is my required ans.

In the level for this we will be using  $\&$  Recursive traversal cause for level order traversal, consider full binary tree



In the worst case you will need to maintain 4 nodes and due to this space complexity can be very much.

Hence we will be using "RECURSIVE TRAVERSAL".

but in recursive the T.C =  $O(N)$  but S.C =  $O(H)$   $\rightarrow$  Height of the tree.

preOrder  $\rightarrow$  Root, left, right

But for this  $\rightarrow$  [Root, right, left]

```

root, o
f(node, level) {

```

```

    if (node == NULL) return;

```

```

    if (level == ds.size) {

```

```

        ds.push_back(node->val);
    }

```

```

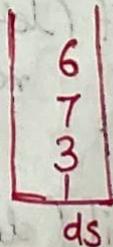
    if (node->right, level+1);

```

```

    if (node->left, level+1);
}

```



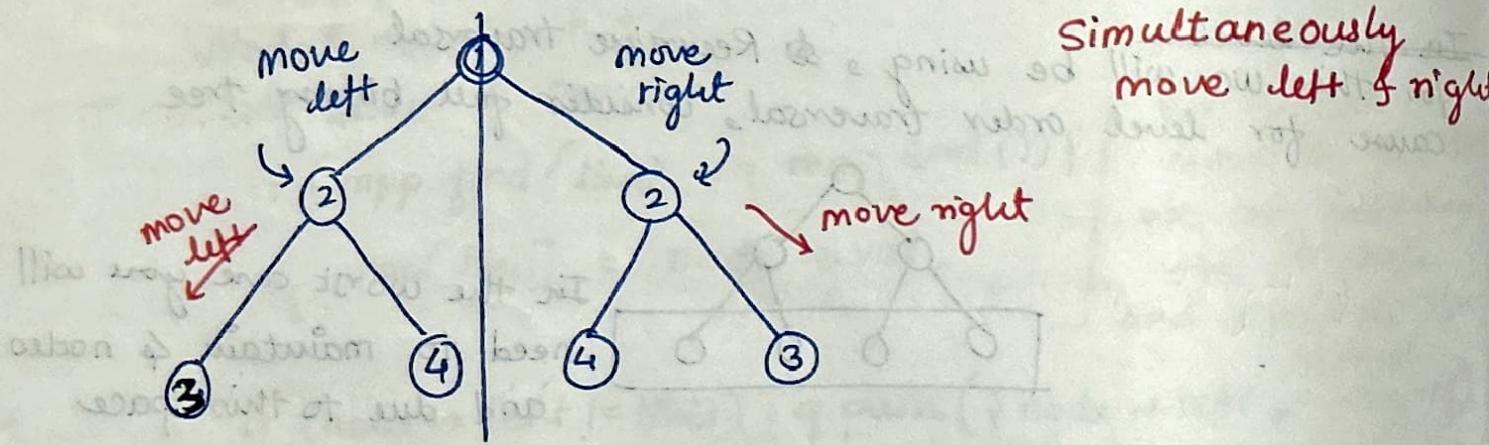
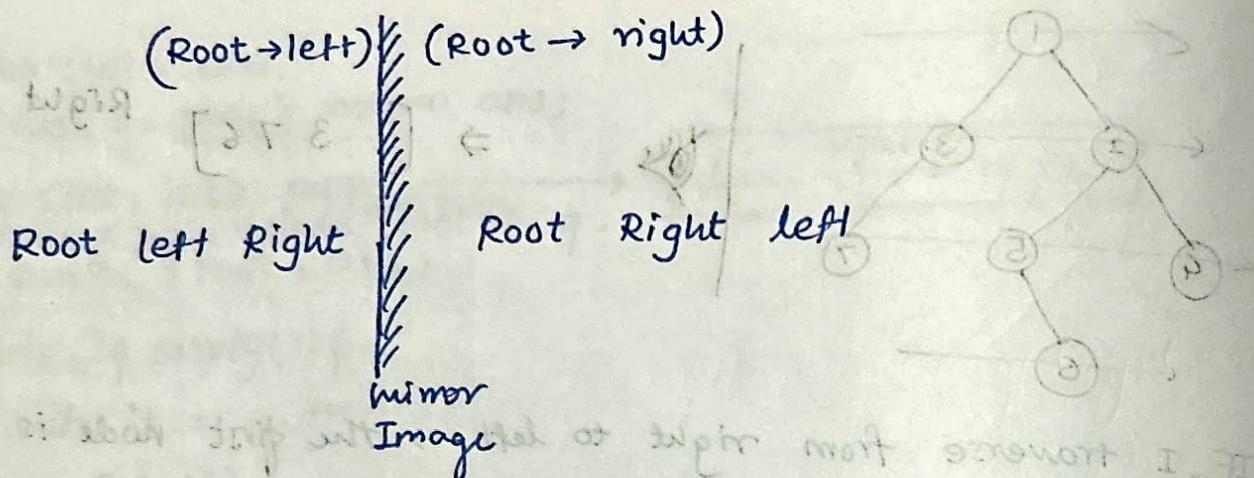
Check the ds size at each traversal.

(H.O = 3.7)
(L.O = 3.2)

for left view  $\Rightarrow$  Traverse

(R L Right)

## Symmetric Binary Tree:-



```
bool isSymmetricTree(TreeNode* node){
```

```
    return root == NULL || isSymmetricHelper(node → left, node → right);
```

```
}
```

```
bool isSymmetricHelper(TreeNode* left, TreeNode* right){
```

```
if (left == NULL || right == NULL){
```

```
    return left == right;
```

```
}
```

```
if (left → val != right → val) return false;
```

```
return isSymmetricHelper(left → left, right → right);
```

```
&&
```

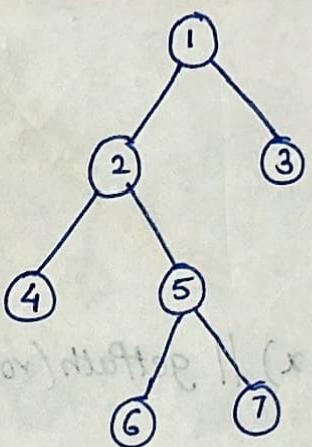
```
isSymmetricHelper(left → right, right → left);
```

```
}
```

T.C = O(N)
------------

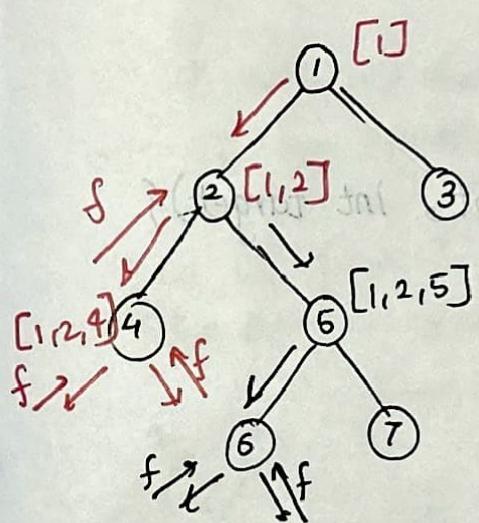
S.C = O(N)
------------

• Print Root to Node path in Binary Tree:



node = 7  
path from '1' to '7'  
 $\Rightarrow [1, 2, 5, 7]$

Traversal  $\Rightarrow$  Inorder traversal



first move to left & come back

then right and come back

At ②  $\rightarrow$  is 2 == 7 No!  $\rightarrow$  push [1, 2]

move left ④  $\rightarrow$  is 4 == 7 No!  $\rightarrow$  push [1, 2, 4]

move left  $\Rightarrow$  NULL  $\Rightarrow$  return false

move right  $\Rightarrow$  NULL  $\Rightarrow$  return false

so from node ④  $\rightarrow$  false is returned hence pop the last element (4)  $\Rightarrow$  [1, 2, 4] will become [1, 2]

move right from ②  $\rightarrow$  5 == 7 No!  $\rightarrow$  push [1, 2, 5]

move left  $\Rightarrow$  ⑥  $\rightarrow$  6 == 7 No!  $\rightarrow$  push [1, 2, 5, 6]

move left of ⑥  $\rightarrow$  NULL  $\rightarrow$  return false.

move right of ⑥  $\rightarrow$  NULL  $\rightarrow$  return false.  $\Rightarrow$  pop the ⑥

Array is [1, 2, 5]

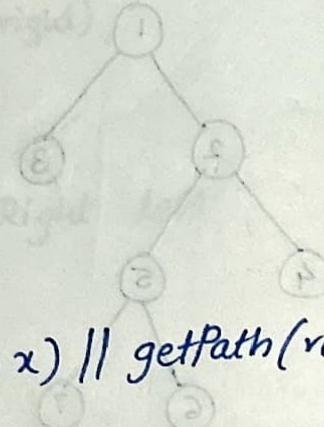
Now move right of ⑤  $\rightarrow$  7 is 7 == 7  $\Rightarrow$  True  $\rightarrow$  push [1, 2, 5, 7]  
return True.

The moment you get true from any of the left call or right call you've got the answer.

```

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;
}

```

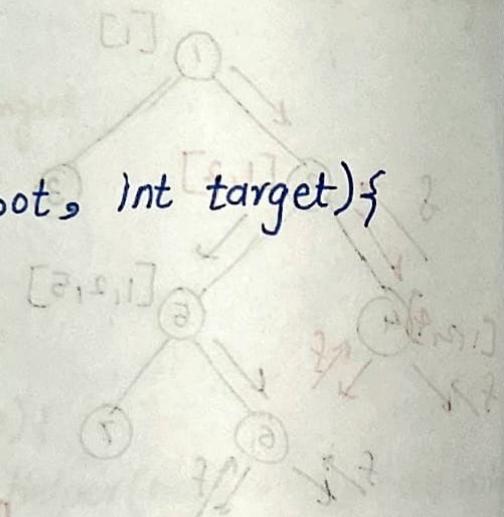


last visit reborn ← last visit

```

vector<int> pathToNode(TreeNode* root, int target) {
    vector<int> arr;
    if (root == NULL)
        return arr;
    getPath(root, arr, target);
    return arr;
}

```



$$\boxed{\begin{array}{l} T.C = O(N) \\ S.C = O(H) \end{array}}$$

(2) sort qop

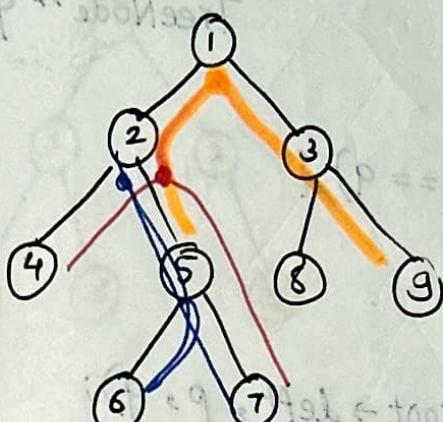
[2,5,1] si ponA

[2,5,1] swap ← sort ← T == S & S ← @ to tulpin evom woh

sort mptor

nos habi att fo pro sort sort top way trunom att  
wanted att top way no tulpin n

## Lowest Common Ancestor in Binary Trees



$$\text{dca}(4,7) = 2$$

$$\text{lca}(5, 9) = 1$$

$$\text{lca}(2,6) = 2$$

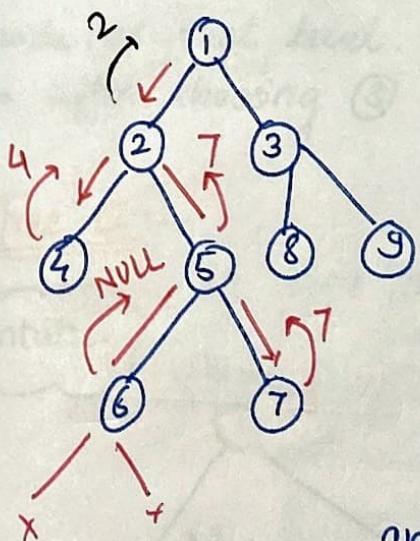
node = 4      path (1, 2, 7)  
 node = 7      Path (1, 2, 5, 7)  
 ↑               ↑  
 Matched          Matched

⇒ Whatever the last element is matched in path that is our lowest common ancestor.

T.C = O(N) → To find the path

S.C =  $O(N)$  → Extra use of time complexity

$\text{lca}(4,7)$



- We will recursively traverse
- go to left  $\Rightarrow$  DFS → and whenever you encounter any of the node (4 / 7) then you have to return (node ~~marked~~) IMPORTANT!

⇒ At ⑤ from left → NULL  
and ↑  
Right

and now ⑤ will return 7

At  $\textcircled{2}$   Right  $\rightarrow$  This signifies you have figured out both the nodes

If left != NULL & right != NULL then

At this point, means whenever you encounter any node whose left & right both have valid values then return the (node ~~→~~)  $\Rightarrow$  And this is LCA.

`TreeNode* LowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q);`

// base case :-

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

`TreeNode* left = lowestCommonAncestor (root → left, p, q);`  
`TreeNode* right = lowestCommonAncestor (root → right, p, q);`

if (left == NULL) return right;

```
if (right == NULL) return left;
```

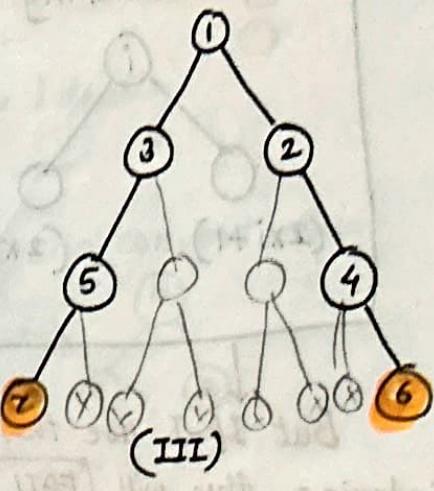
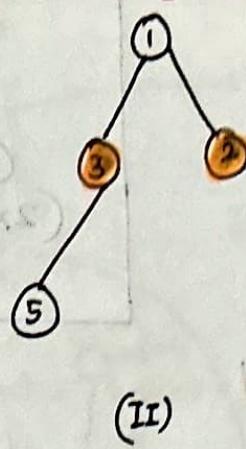
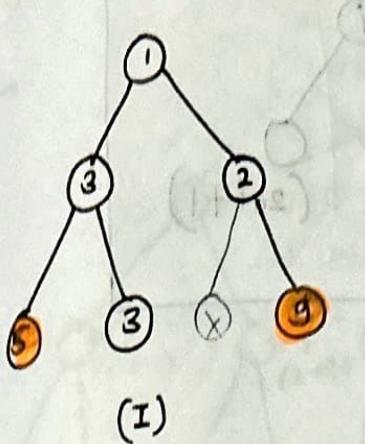
else return root; returns normal tree if

3

I wonder how (2) you know

ADJ: si  $a \in B$   $\in (\text{Folge ab})$  mit  $a_n \rightarrow a$

## Maximum width of Binary Tree :-



Width of Binary tree :- No. of nodes in a level betn any two nodes.

(I) Tree :-

⑤ and ⑨ are on the same level and if ② would have the left node then the maximum width of tree  $\geq 4$

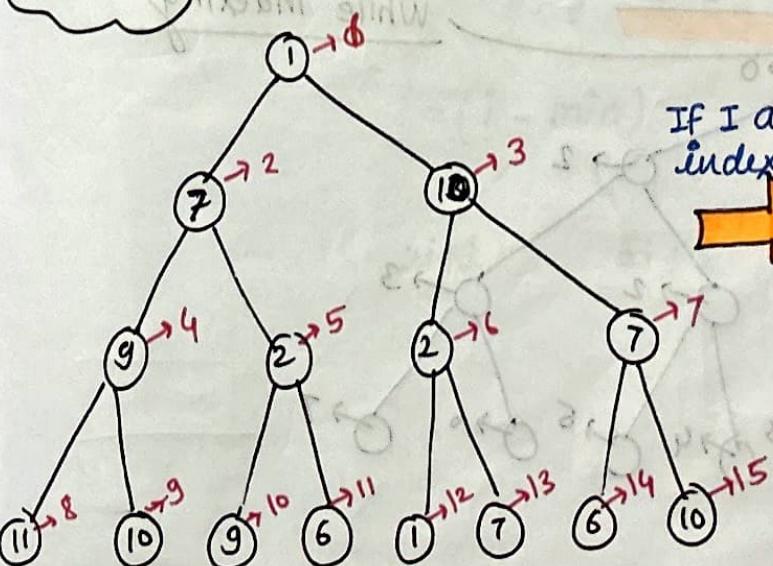
Tree - II :-

we can't take ⑤ because we don't have any other node at that level.  
so after choosing ③ and ②  $\Rightarrow$  maximum width = 2

Tree - III :-

In third tree  $\Rightarrow$  Maximum width = 8

Intuition



If I assign the index at each level.

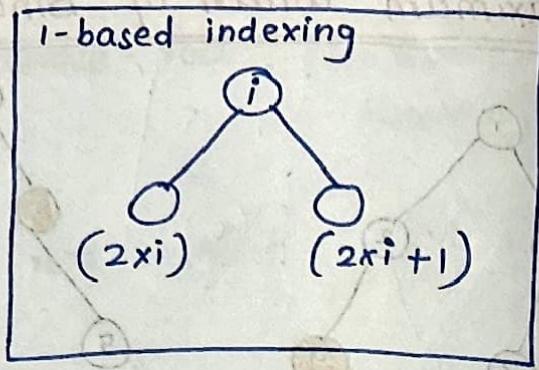
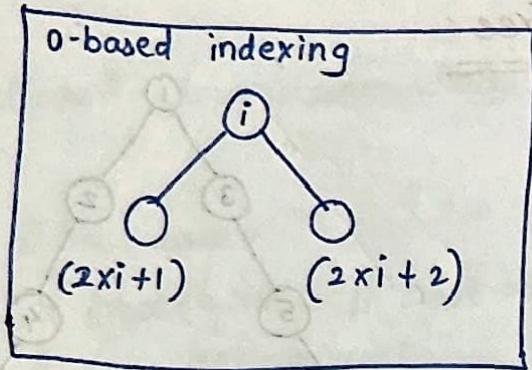
Then at each level  
I just have to subtract  
the (last idx - first idx + 1)

I will get the width

$$\text{EX: } ① 3 - 2 + 1 = \frac{2}{2}$$

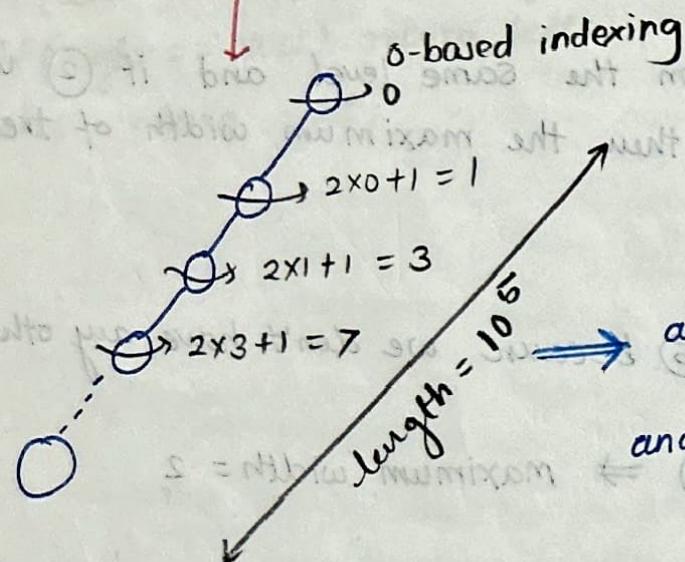
$$② 7 - 4 + 1 = \frac{4}{4}$$

$$③ 15 - 8 + 1 = \frac{8}{8}$$



But If I use this 0-based indexing this will FAIL

Why??



skew-tree

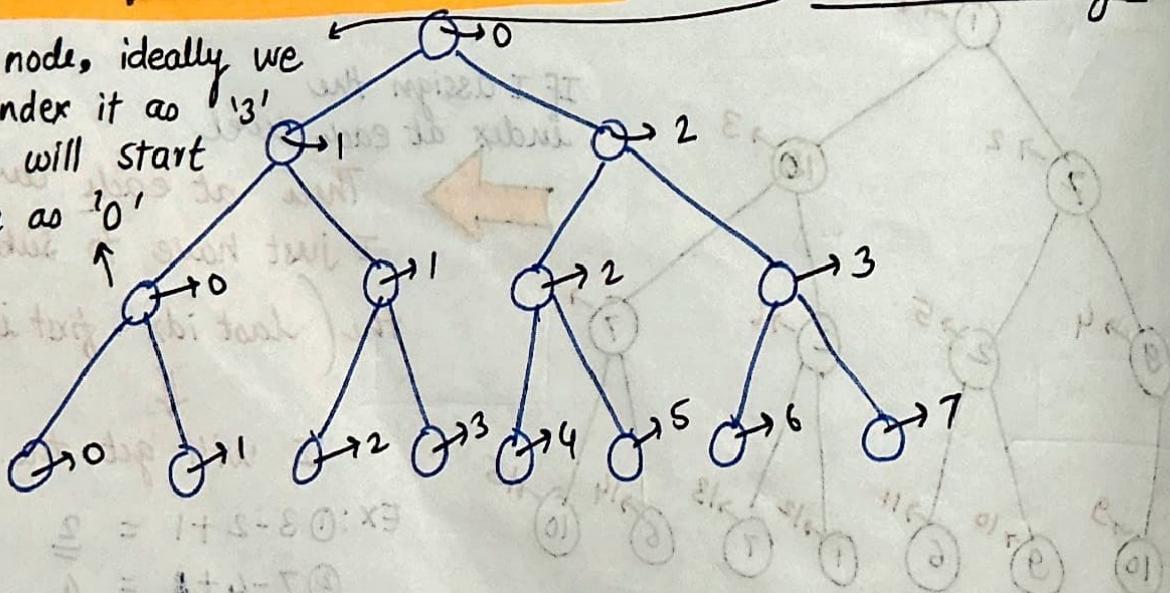
and At each step we are doing  $(2xi+1)$   
and  $2 \times 10^5 + 1 \Rightarrow$  will overflow!!!

Hence this will fail!!

How will you prevent this overflow  $\Rightarrow$

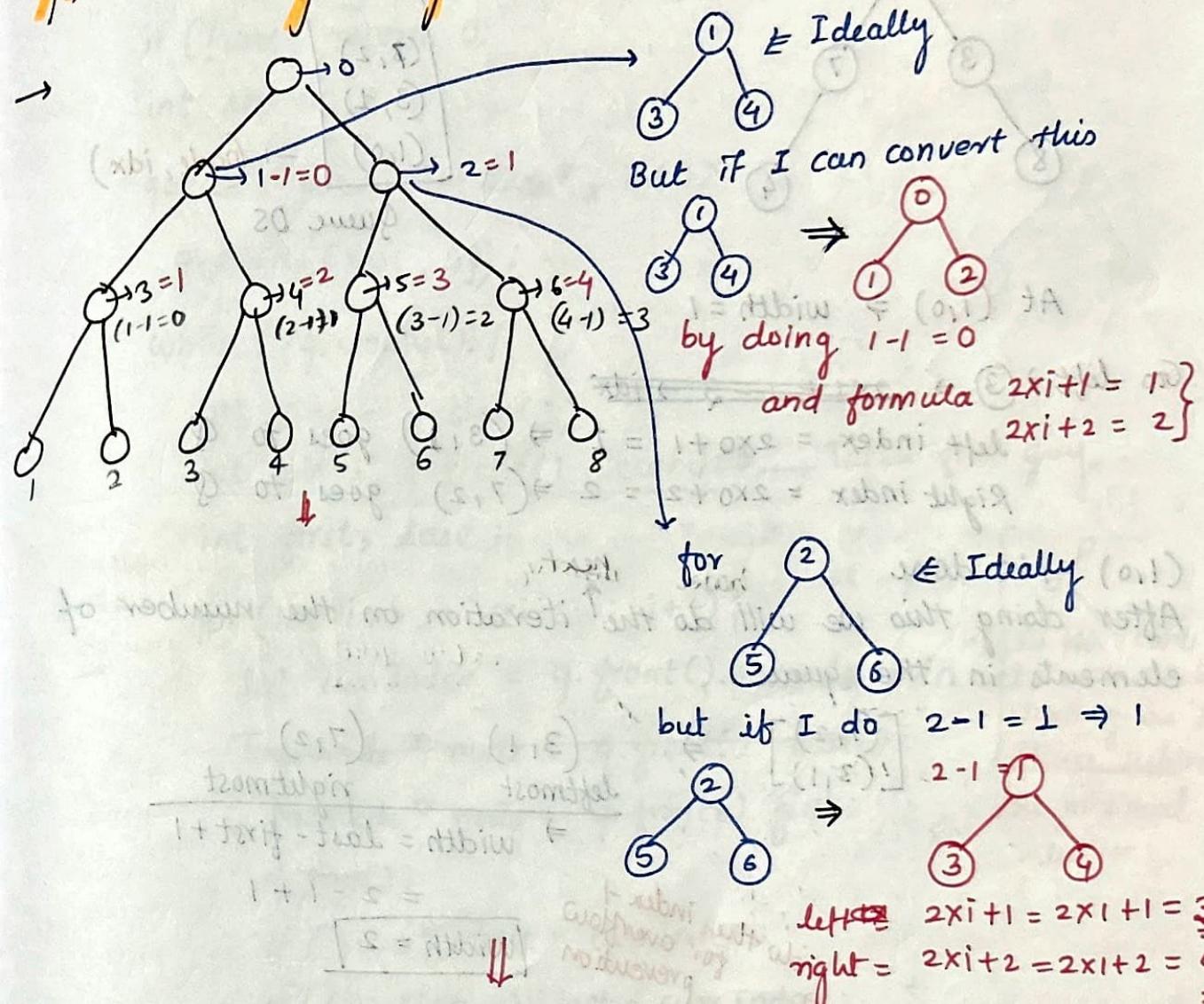
At this node, ideally we should index it as '3'  
but we will start indexing as '0'

while indexing



Same here

# How will you get those indexes?



THIS IS HOW IT WILL GET SIMPLIFIED TILL THE DEPTH OF TREE!

This is how it will get prevented from OVERFLOW!!

$$i = (i - \min)$$

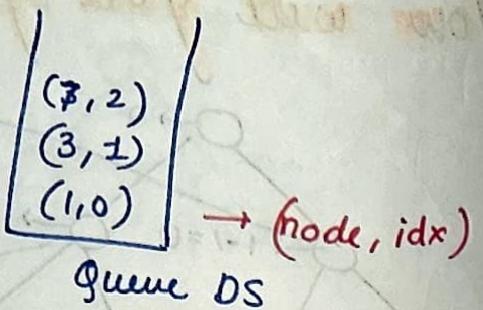
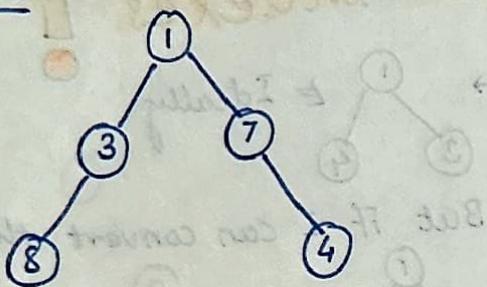
$$2i+1$$

$$2i+2$$

$$\begin{array}{|c|} \hline i+1-\Delta = w \\ p = w \\ \hline \end{array}$$

(1, 8)

## Dry Run



At  $(1, 0) \Rightarrow \text{width} = 1$

Go left  $\Rightarrow 3 \Rightarrow$  ~~2x0 + 2 > idx~~  
 left index  $= 2 \times 0 + 1 = 1 \Rightarrow (3, 1)$  goes to Q  
 Right index  $= 2 \times 0 + 2 = 2 \Rightarrow (7, 2)$  goes to Q

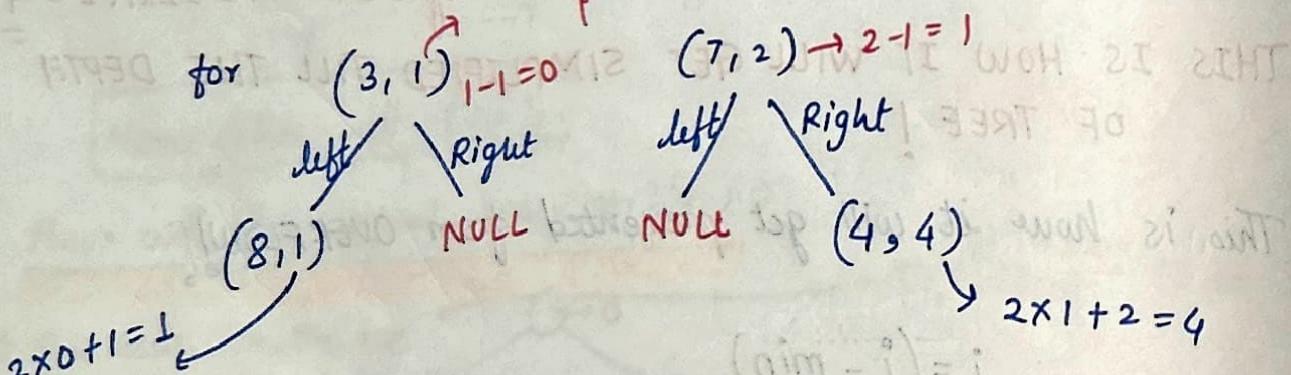
$(1, 0) \rightarrow$  is done  
 After doing this we will do the  $\uparrow$  iteration on the number of elements in the queue.

$$\begin{bmatrix} (7, 2) \\ (3, 1) \end{bmatrix} \Rightarrow \begin{array}{c} (3, 1) \\ \text{leftmost} \\ \hline (7, 2) \\ \text{rightmost} \end{array} \Rightarrow \text{width} = \text{last} - \text{first} + 1$$

$$= 2 - 1 + 1$$

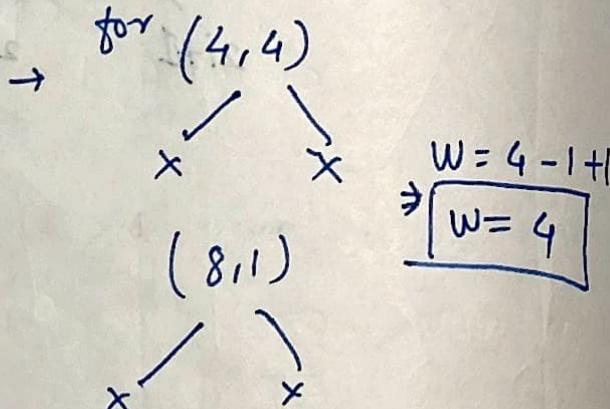
do the index-1  
for overflow  
prevention

$$\boxed{\text{width} = 2}$$



push these in Queue

$$\begin{bmatrix} (4, 4) \\ (8, 1) \end{bmatrix} \quad Q$$



```

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

```

In this case  
It will not be  
starting as '1'  
Hence subtract  
the minimal  
index

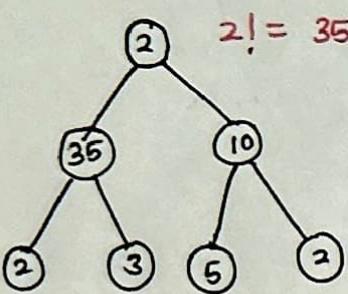
<img alt="A binary tree diagram with root 1. Root 1 has left child 2 and right child 3. Node 2 has left child 4 and right child 5. Node 3 has left child 6 and right child 7. Node 4 has left child 8 and right child 9. Node 5 has left child 10 and right child 11. Node 6 has left child 12 and right child 13. Node 7 has left child 14 and right child 15. Node 8 has left child 16 and right child 17. Node 9 has left child 18 and right child 19. Node 10 has left child 20 and right child 21. Node 11 has left child 22 and right child 23. Node 12 has left child 24 and right child 25. Node 13 has left child 26 and right child 27. Node 14 has left child 28 and right child 29. Node 15 has left child 30 and right child 31. Node 16 has left child 32 and right child 33. Node 17 has left child 34 and right child 35. Node 18 has left child 36 and right child 37. Node 19 has left child 38 and right child 39. Node 20 has left child 40 and right child 41. Node 21 has left child 42 and right child 43. Node 22 has left child 44 and right child 45. Node 23 has left child 46 and right child 47. Node 24 has left child 48 and right child 49. Node 25 has left child 50 and right child 51. Node 26 has left child 52 and right child 53. Node 27 has left child 54 and right child 55. Node 28 has left child 56 and right child 57. Node 29 has left child 58 and right child 59. Node 30 has left child 60 and right child 61. Node 31 has left child 62 and right child 63. Node 32 has left child 64 and right child 65. Node 33 has left child 66 and right child 67. Node 34 has left child 68 and right child 69. Node 35 has left child 70 and right child 71. Node 36 has left child 72 and right child 73. Node 37 has left child 74 and right child 75. Node 38 has left child 76 and right child 77. Node 39 has left child 78 and right child 79. Node 40 has left child 80 and right child 81. Node 41 has left child 82 and right child 83. Node 42 has left child 84 and right child 85. Node 43 has left child 86 and right child 87. Node 44 has left child 88 and right child 89. Node 45 has left child 90 and right child 91. Node 46 has left child 92 and right child 93. Node 47 has left child 94 and right child 95. Node 48 has left child 96 and right child 97. Node 49 has left child 98 and right child 99. Node 50 has left child 100 and right child 101. Node 51 has left child 102 and right child 103. Node 52 has left child 104 and right child 105. Node 53 has left child 106 and right child 107. Node 54 has left child 108 and right child 109. Node 55 has left child 110 and right child 111. Node 56 has left child 112 and right child 113. Node 57 has left child 114 and right child 115. Node 58 has left child 116 and right child 117. Node 59 has left child 118 and right child 119. Node 60 has left child 120 and right child 121. Node 61 has left child 122 and right child 123. Node 62 has left child 124 and right child 125. Node 63 has left child 126 and right child 127. Node 64 has left child 128 and right child 129. Node 65 has left child 130 and right child 131. Node 66 has left child 132 and right child 133. Node 67 has left child 134 and right child 135. Node 68 has left child 136 and right child 137. Node 69 has left child 138 and right child 139. Node 70 has left child 140 and right child 141. Node 71 has left child 142 and right child 143. Node 72 has left child 144 and right child 145. Node 73 has left child 146 and right child 147. Node 74 has left child 148 and right child 149. Node 75 has left child 150 and right child 151. Node 76 has left child 152 and right child 153. Node 77 has left child 154 and right child 155. Node 78 has left child 156 and right child 157. Node 79 has left child 158 and right child 159. Node 80 has left child 160 and right child 161. Node 81 has left child 162 and right child 163. Node 82 has left child 164 and right child 165. Node 83 has left child 166 and right child 167. Node 84 has left child 168 and right child 169. Node 85 has left child 170 and right child 171. Node 86 has left child 172 and right child 173. Node 87 has left child 174 and right child 175. Node 88 has left child 176 and right child 177. Node 89 has left child 178 and right child 179. Node 90 has left child 180 and right child 181. Node 91 has left child 182 and right child 183. Node 92 has left child 184 and right child 185. Node 93 has left child 186 and right child 187. Node 94 has left child 188 and right child 189. Node 95 has left child 190 and right child 191. Node 96 has left child 192 and right child 193. Node 97 has left child 194 and right child 195. Node 98 has left child 196 and right child 197. Node 99 has left child 198 and right child 199. Node 100 has left child 200 and right child 201. Node 101 has left child 202 and right child 203. Node 102 has left child 204 and right child 205. Node 103 has left child 206 and right child 207. Node 104 has left child 208 and right child 209. Node 105 has left child 210 and right child 211. Node 106 has left child 212 and right child 213. Node 107 has left child 214 and right child 215. Node 108 has left child 216 and right child 217. Node 109 has left child 218 and right child 219. Node 110 has left child 220 and right child 221. Node 111 has left child 222 and right child 223. Node 112 has left child 224 and right child 225. Node 113 has left child 226 and right child 227. Node 114 has left child 228 and right child 229. Node 115 has left child 230 and right child 231. Node 116 has left child 232 and right child 233. Node 117 has left child 234 and right child 235. Node 118 has left child 236 and right child 237. Node 119 has left child 238 and right child 239. Node 120 has left child 240 and right child 241. Node 121 has left child 242 and right child 243. Node 122 has left child 244 and right child 245. Node 123 has left child 246 and right child 247. Node 124 has left child 248 and right child 249. Node 125 has left child 250 and right child 251. Node 126 has left child 252 and right child 253. Node 127 has left child 254 and right child 255. Node 128 has left child 256 and right child 257. Node 129 has left child 258 and right child 259. Node 130 has left child 260 and right child 261. Node 131 has left child 262 and right child 263. Node 132 has left child 264 and right child 265. Node 133 has left child 266 and right child 267. Node 134 has left child 268 and right child 269. Node 135 has left child 270 and right child 271. Node 136 has left child 272 and right child 273. Node 137 has left child 274 and right child 275. Node 138 has left child 276 and right child 277. Node 139 has left child 278 and right child 279. Node 140 has left child 280 and right child 281. Node 141 has left child 282 and right child 283. Node 142 has left child 284 and right child 285. Node 143 has left child 286 and right child 287. Node 144 has left child 288 and right child 289. Node 145 has left child 290 and right child 291. Node 146 has left child 292 and right child 293. Node 147 has left child 294 and right child 295. Node 148 has left child 296 and right child 297. Node 149 has left child 298 and right child 299. Node 150 has left child 300 and right child 301. Node 151 has left child 302 and right child 303. Node 152 has left child 304 and right child 305. Node 153 has left child 306 and right child 307. Node 154 has left child 308 and right child 309. Node 155 has left child 310 and right child 311. Node 156 has left child 312 and right child 313. Node 157 has left child 314 and right child 315. Node 158 has left child 316 and right child 317. Node 159 has left child 318 and right child 319. Node 160 has left child 320 and right child 321. Node 161 has left child 322 and right child 323. Node 162 has left child 324 and right child 325. Node 163 has left child 326 and right child 327. Node 164 has left child 328 and right child 329. Node 165 has left child 330 and right child 331. Node 166 has left child 332 and right child 333. Node 167 has left child 334 and right child 335. Node 168 has left child 336 and right child 337. Node 169 has left child 338 and right child 339. Node 170 has left child 340 and right child 341. Node 171 has left child 342 and right child 343. Node 172 has left child 344 and right child 345. Node 173 has left child 346 and right child 347. Node 174 has left child 348 and right child 349. Node 175 has left child 350 and right child 351. Node 176 has left child 352 and right child 353. Node 177 has left child 354 and right child 355. Node 178 has left child 356 and right child 357. Node 179 has left child 358 and right child 359. Node 180 has left child 360 and right child 361. Node 181 has left child 362 and right child 363. Node 182 has left child 364 and right child 365. Node 183 has left child 366 and right child 367. Node 184 has left child 368 and right child 369. Node 185 has left child 370 and right child 371. Node 186 has left child 372 and right child 373. Node 187 has left child 374 and right child 375. Node 188 has left child 376 and right child 377. Node 189 has left child 378 and right child 379. Node 190 has left child 380 and right child 381. Node 191 has left child 382 and right child 383. Node 192 has left child 384 and right child 385. Node 193 has left child 386 and right child 387. Node 194 has left child 388 and right child 389. Node 195 has left child 390 and right child 391. Node 196 has left child 392 and right child 393. Node 197 has left child 394 and right child 395. Node 198 has left child 396 and right child 397. Node 199 has left child 398 and right child 399. Node 200 has left child 400 and right child 401. Node 201 has left child 402 and right child 403. Node 202 has left child 404 and right child 405. Node 203 has left child 406 and right child 407. Node 204 has left child 408 and right child 409. Node 205 has left child 410 and right child 411. Node 206 has left child 412 and right child 413. Node 207 has left child 414 and right child 415. Node 208 has left child 416 and right child 417. Node 209 has left child 418 and right child 419. Node 210 has left child 420 and right child 421. Node 211 has left child 422 and right child 423. Node 212 has left child 424 and right child 425. Node 213 has left child 426 and right child 427. Node 214 has left child 428 and right child 429. Node 215 has left child 430 and right child 431. Node 216 has left child 432 and right child 433. Node 217 has left child 434 and right child 435. Node 218 has left child 436 and right child 437. Node 219 has left child 438 and right child 439. Node 220 has left child 440 and right child 441. Node 221 has left child 442 and right child 443. Node 222 has left child 444 and right child 445. Node 223 has left child 446 and right child 447. Node 224 has left child 448 and right child 449. Node 225 has left child 450 and right child 451. Node 226 has left child 452 and right child 453. Node 227 has left child 454 and right child 455. Node 228 has left child 456 and right child 457. Node 229 has left child 458 and right child 459. Node 230 has left child 460 and right child 461. Node 231 has left child 462 and right child 463. Node 232 has left child 464 and right child 465. Node 233 has left child 466 and right child 467. Node 234 has left child 468 and right child 469. Node 235 has left child 470 and right child 471. Node 236 has left child 472 and right child 473. Node 237 has left child 474 and right child 475. Node 238 has left child 476 and right child 477. Node 239 has left child 478 and right child 479. Node 240 has left child 480 and right child 481. Node 241 has left child 482 and right child 483. Node 242 has left child 484 and right child 485. Node 243 has left child 486 and right child 487. Node 244 has left child 488 and right child 489. Node 245 has left child 490 and right child 491. Node 246 has left child 492 and right child 493. Node 247 has left child 494 and right child 495. Node 248 has left child 496 and right child 497. Node 249 has left child 498 and right child 499. Node 250 has left child 500 and right child 501. Node 251 has left child 502 and right child 503. Node 252 has left child 504 and right child 505. Node 253 has left child 506 and right child 507. Node 254 has left child 508 and right child 509. Node 255 has left child 510 and right child 511. Node 256 has left child 512 and right child 513. Node 257 has left child 514 and right child 515. Node 258 has left child 516 and right child 517. Node 259 has left child 518 and right child 519. Node 260 has left child 520 and right child 521. Node 261 has left child 522 and right child 523. Node 262 has left child 524 and right child 525. Node 263 has left child 526 and right child 527. Node 264 has left child 528 and right child 529. Node 265 has left child 530 and right child 531. Node 266 has left child 532 and right child 533. Node 267 has left child 534 and right child 535. Node 268 has left child 536 and right child 537. Node 269 has left child 538 and right child 539. Node 270 has left child 540 and right child 541. Node 271 has left child 542 and right child 543. Node 272 has left child 544 and right child 545. Node 273 has left child 546 and right child 547. Node 274 has left child 548 and right child 549. Node 275 has left child 550 and right child 551. Node 276 has left child 552 and right child 553. Node 277 has left child 554 and right child 555. Node 278 has left child 556 and right child 557. Node 279 has left child 558 and right child 559. Node 280 has left child 560 and right child 561. Node 281 has left child 562 and right child 563. Node 282 has left child 564 and right child 565. Node 283 has left child 566 and right child 567. Node 284 has left child 568 and right child 569. Node 285 has left child 570 and right child 571. Node 286 has left child 572 and right child 573. Node 287 has left child 574 and right child 575. Node 288 has left child 576 and right child 577. Node 289 has left child 578 and right child 579. Node 290 has left child 580 and right child 581. Node 291 has left child 582 and right child 583. Node 292 has left child 584 and right child 585. Node 293 has left child 586 and right child 587. Node 294 has left child 588 and right child 589. Node 295 has left child 590 and right child 591. Node 296 has left child 592 and right child 593. Node 297 has left child 594 and right child 595. Node 298 has left child 596 and right child 597. Node 299 has left child 598 and right child 599. Node 300 has left child 600 and right child 601. Node 301 has left child 602 and right child 603. Node 302 has left child 604 and right child 605. Node 303 has left child 606 and right child 607. Node 304 has left child 608 and right child 609. Node 305 has left child 610 and right child 611. Node 306 has left child 612 and right child 613. Node 307 has left child 614 and right child 615. Node 308 has left child 616 and right child 617. Node 309 has left child 618 and right child 619. Node 310 has left child 620 and right child 621. Node 311 has left child 622 and right child 623. Node 312 has left child 624 and right child 625. Node 313 has left child 626 and right child 627. Node 314 has left child 628 and right child 629. Node 315 has left child 630 and right child 631. Node 316 has left child 632 and right child 633. Node 317 has left child 634 and right child 635. Node 318 has left child 636 and right child 637. Node 319 has left child 638 and right child 639. Node 320 has left child 640 and right child 641. Node 321 has left child 642 and right child 643. Node 322 has left child 644 and right child 645. Node 323 has left child 646 and right child 647. Node 324 has left child 648 and right child 649. Node 325 has left child 650 and right child 651. Node 326 has left child 652 and right child 653. Node 327 has left child 654 and right child 655. Node 328 has left child 656 and right child 657. Node 329 has left child 658 and right child 659. Node 330 has left child 660 and right child 661. Node 331 has left child 662 and right child 663. Node 332 has left child 664 and right child 665. Node 333 has left child 666 and right child 667. Node 334 has left child 668 and right child 669. Node 335 has left child 670 and right child 671. Node 336 has left child 672 and right child 673. Node 337 has left child 674 and right child 675. Node 338 has left child 676 and right child 677. Node 339 has left child 678 and right child 679. Node 340 has left child 680 and right child 681. Node 341 has left child 682 and right child 683. Node 342 has left child 684 and right child 685. Node 343 has left child 686 and right child 687. Node 344 has left child 688 and right child 689. Node 345 has left child 690 and right child 691. Node 346 has left child 692 and right child 693. Node 347 has left child 694 and right child 695. Node 348 has left child 696 and right child 697. Node 349 has left child 698 and right child 699. Node 350 has left child 700 and right child 701. Node 351 has left child 702 and right child 703. Node 352 has left child 704 and right child 705. Node 353 has left child 706 and right child 707. Node 354 has left child 708 and right child 709. Node 355 has left child 710 and right child 711. Node 356 has left child 712 and right child 713. Node 357 has left child 714 and right child 715. Node 358 has left child 716 and right child 717. Node 359 has left child 718 and right child 719. Node 360 has left child 720 and right child 721. Node 361 has left child 722 and right child 723. Node 362 has left child 724 and right child 725. Node 363 has left child 726 and right child 727. Node 364 has left child 728 and right child 729. Node 365 has left child 730 and right child 731. Node 366 has left child 732 and right child 733. Node 367 has left child 734 and right child 735. Node 368 has left child 736 and right child 737. Node 369 has left child 738 and right child 739. Node 370 has left child 740 and right child 741. Node 371 has left child 742 and right child 743. Node 372 has left child 744 and right child 745. Node 373 has left child 746 and right child 747. Node 374 has left child 748 and right child 749. Node 375 has left child 750 and right child 751. Node 376 has left child 752 and right child 753. Node 377 has left child 754 and right child 755. Node 378 has left child 756 and right child 757. Node 379 has left child 758 and right child 759. Node 380 has left child 760 and right child 761. Node 381 has left child 762 and right child 763. Node 382 has left child 764 and right child 765. Node 383 has left child 766 and right child 767. Node 384 has left child 768 and right child 769. Node 385 has left child 770 and right child 771. Node 386 has left child 772 and right child 773. Node 387 has left child 774 and right child 775. Node 388 has left child 776 and right child 777. Node 389 has left child 778 and right child 779. Node 390 has left child 780 and right child 781. Node 391 has left child 782 and right child 783. Node 392 has left child 784 and right child 785. Node 393 has left child 786 and right child 787. Node 394 has left child 788 and right child 789. Node 395 has left child 790 and right child 791. Node 396 has left child 792 and right child 793. Node 397 has left child 794 and right child 795. Node 398 has left child 796 and right child 797. Node 399 has left child 798 and right child 799. Node 400 has left child 800 and right child 801. Node 401 has left child 802 and right child 803. Node 402 has left child 804 and right child 805. Node 403 has left child 806 and right child 807. Node 404 has left child 808 and right child 809. Node 405 has left child 810 and right child 811. Node 406 has left child 812 and right child 813. Node 407 has left child 814 and right child 815. Node 408 has left child 816 and right child 817. Node 409 has left child 818 and right child 819. Node 410 has left child 820 and right child 821. Node 411 has left child 822 and right child 823. Node 412 has left child 824 and right child 825. Node 413 has left child 826 and right child 827. Node 414 has left child 828 and right child 829. Node 415 has left child 830 and right child 831. Node 416 has left child 832 and right child 833. Node 417 has left child 834 and right child 835. Node 418 has left child 836 and right child 837. Node 419 has left child 838 and right child 839. Node 420 has left child 840 and right child 841. Node 421 has left child 842 and right child 843. Node 422 has left child 844 and right child 845. Node 423 has left child 846 and right child 847. Node 424 has left child 848 and right child 849. Node 425 has left child 850 and right child 851. Node 426 has left child 852 and right child 853. Node 427 has left child 854 and right child 855. Node 428 has left child 856 and right child 857. Node 429 has left child 858 and right child 859. Node 430 has left child 860 and right child 861. Node 431 has left child 862 and right child 863. Node 432 has left child 864 and right child 865. Node 433 has left child 866 and right child 867. Node 434 has left child 868 and right child 869. Node 435 has left child 870 and right child 871. Node 436 has left child 872 and right child 873. Node 437 has left child 874 and right child 875. Node 438 has left child 876 and right child 877. Node 439 has left child 878 and right child 879. Node 440 has left child 880 and right child 881. Node 441 has left child 882 and right child 883. Node 442 has left child 884 and right child 885. Node 443 has left child 886 and right child 887. Node 444 has left child 888 and right child 889. Node 445 has left child 890 and right child 891. Node 446 has left child 892 and right child 893. Node 447 has left child 894 and right child 895. Node 448 has left child 896 and right child 897. Node 449 has left child 898 and right child 899. Node 450 has left child 900 and right child 901. Node 451 has left child 902 and right child 903. Node 452 has left child 904 and right child 905. Node 453 has left child 906 and right child 907. Node 454 has left child 908 and right child 909. Node 455 has left child 910 and right child 911. Node 456 has left child 912 and right child 913. Node 457 has left child 914 and right child 915. Node 458 has left child 916 and right child 917. Node 459 has left child 918 and right child 919. Node 460 has left child 920 and right child 921. Node 461 has left child 922 and right child 923. Node 462 has left child 924 and right child 925. Node 463 has left child 926 and right child 927. Node 464 has left child 928 and right child 929. Node 465 has left child 930 and right child 931. Node 466 has left child 932 and right child 933. Node 467 has left child 934 and right child 935. Node 468 has left child 936 and right child 937. Node 469 has left child 938 and right child 939. Node 470 has left child 940 and right child 941. Node 471 has left child 942 and right child 943. Node 472 has left child 944 and right child 945. Node 473 has left child 946 and right child 947. Node 474 has left child 948 and right child 949. Node 475 has left child 950 and right child 951. Node 476 has left child 952 and right child 953. Node 477 has left child 954 and right child 955. Node 478 has left child 956 and right child 957. Node 479 has left child 958 and right child 959. Node 480 has left child 960 and right child 961. Node 481 has left child 962 and right child 963. Node 482 has left child 964 and right child 965. Node 483 has left child 966 and right child 967. Node 484 has left child 968 and right child 969. Node 485 has left child 970 and right child 971. Node 486 has left child 972 and right child 973. Node 487 has left child 974 and right child 975. Node 488 has left child 976 and right child 977. Node 489 has left child 978 and right child 979. Node 490 has left child 980 and right child 981. Node 491 has left child 982 and right child 983. Node 492 has left child 984 and right child 985. Node 493 has left child 986 and right child 987. Node 494 has left child 988 and right child 989. Node 495 has left child 990 and right child 991. Node 496 has left child 992 and right child 993. Node 497 has left child 994 and right child 995. Node 498 has left child 996 and right child 997. Node 499 has left child 998 and right child 999. Node 500 has left child 1000 and right child 1001.
</p>

## Children sum property :-

$\text{node} = \text{left node} + \text{right node}$   $\Rightarrow$  children sum property

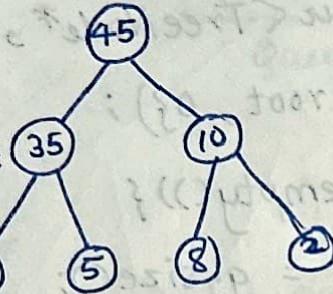
→ If this condition is not satisfying

- you can add +1 to any node to any number of times

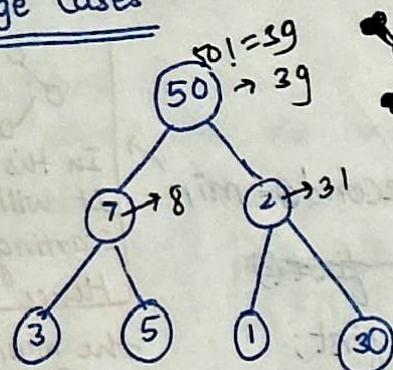


$$2 = 35 + 10$$

You can increment any node. However you want.



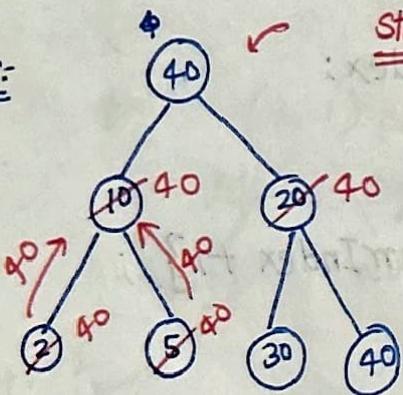
## # Edge Cases -



In this problem, ~~the~~ you can simply add the nodes and put up the sum into parent node cause it won't guarantee you that as you go up you will get that sum !!

→ We will be using recursive traversal

Ex:-



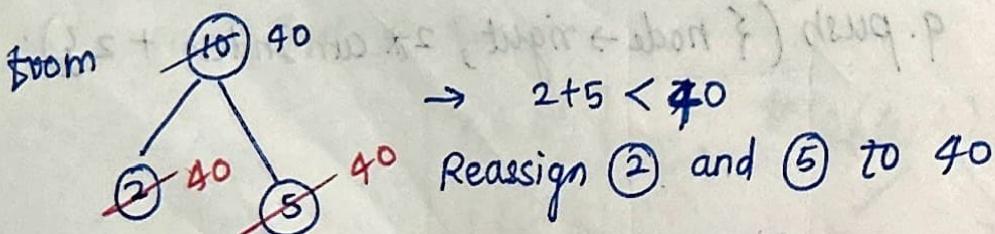
Start →

$\text{root} \rightarrow \text{left} = 10$

$\text{root} \rightarrow \text{right} = 20$

$$10 + 20 = 30 < 40$$

Got the sum less than 40 then I'll assign the left node as 40 and right node as 40



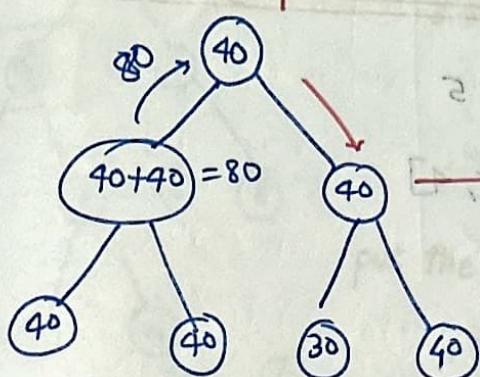
$$\rightarrow 2 + 5 < 40$$

Reassign ② and ⑤ to 40

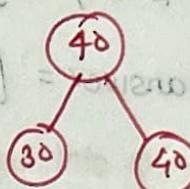
Now from ② → No other nodes are present here

return 40 from ② and same, return 40 from 5

→ Add them up



$2 = \text{parent}, 2 = 3$



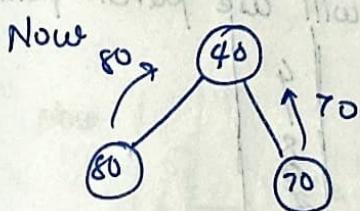
$$30 + 40 = 70$$

$$70 > 40$$

so make the node 40 (parent) as 70.

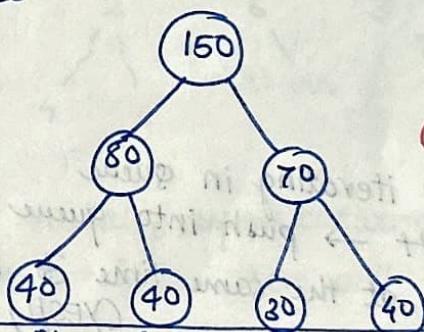
move left  $\rightarrow$  return 30

move right  $\rightarrow$  return 40



Now update the root by  $80 + 170 = 150$

∴ Hence the tree



Children sum property Satisfied

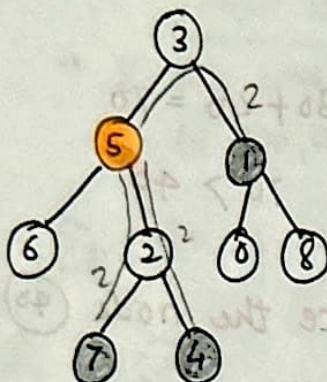
```
void ChangeTreeBinary(BinaryTreeNode<int>* root) {
    if (root == NULL) return;
    int child = 0;
    if (root->left) child += root->left->data;
    if (root->right) child += root->right->data;
    if (child >= root->data) root->data = child;
    else {
        if (root->left) root->left->data = root->data;
        if (root->right) root->right->data = root->data;
    }
    ChangeTreeBinary(root->left);
    ChangeTreeBinary(root->right);
}
```

T.C = O(N)

While coming back to parent

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

- Print all the Nodes at a distance of "k" in Binary Tree

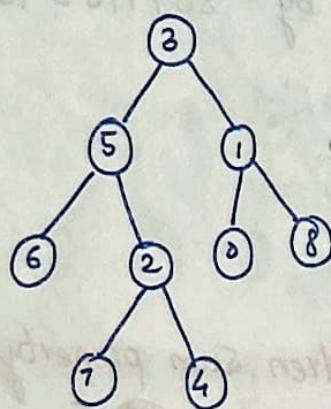


$k=2$ , target = 5

answer = [1, 7, 4]

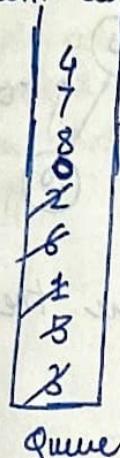
→ IF we want to get all the nodes from node 5 at a distance "2" then we have to move downward + upward in both the direction AND This is the main concern.

→ In order to move in upward direction, we will use parent pointer



$k=2$ , target = 5

(BFS)



Start iterating in queue

if  $3 \rightarrow \text{left} \rightarrow$  push into queue (5)

and at the same time 3 is a parent of 5  
(YES!!)

Now  $3 \rightarrow \text{right} \Rightarrow$  push into queue (1)

and say hey! ① your parent is 3 → YES!!

This is how we can map the parent nodes.

$5 \rightarrow \text{left} \rightarrow 6 \rightarrow$  push q

$5 \rightarrow \text{right} \rightarrow 2 \rightarrow$  push into q

$1 \rightarrow \text{left} \rightarrow 0 \rightarrow$  push into q

$1 \rightarrow \text{right} \rightarrow 8 \rightarrow$  push into q

$6 \rightarrow \text{left} \Rightarrow \text{NULL}$

$6 \rightarrow \text{right} \Rightarrow \text{NULL}$

$2 \rightarrow \text{left} \rightarrow 7 \rightarrow$  push into q

$2 \rightarrow \text{right} \rightarrow 4 \rightarrow$  push into q

like this

parent child map  
↳ child → parent

5 → 3

1 → 3

6 → 5

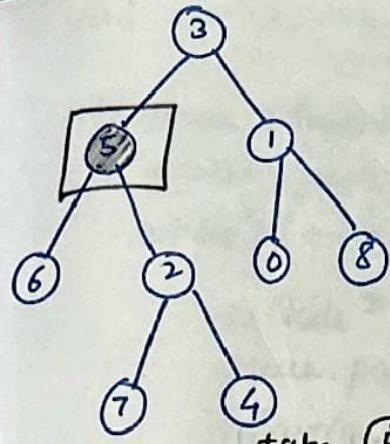
2 → 5

7 → 2

4 → 2

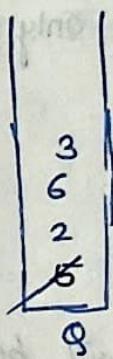
0 → 1

8 → 4



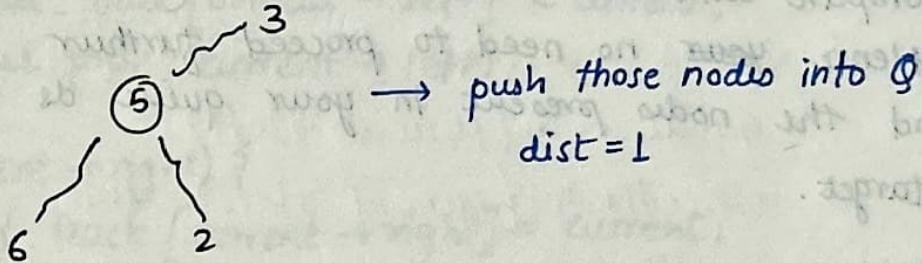
$\text{dist} = 0$

put the target into  $\emptyset \rightarrow$

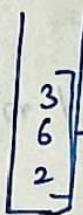


Carry a visited hash

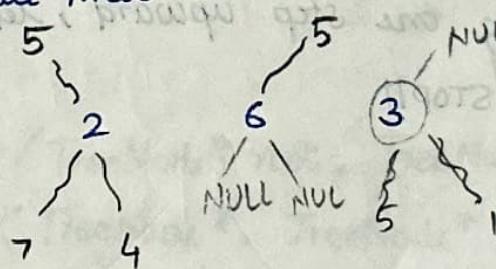
take 5 check upward, left and right



Now

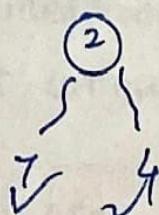


Take all these three nodes and try to move radially outwards

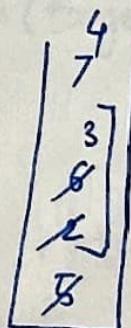


Now think!!

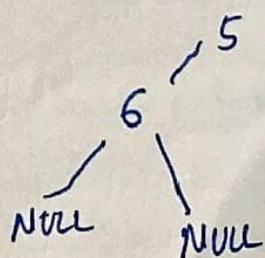
→ If we move radially outwards from 2 we will have 5 which is visited → hence there will help our visited hashmap  
from 2 to 5 → you will not go cause 5 is already visited



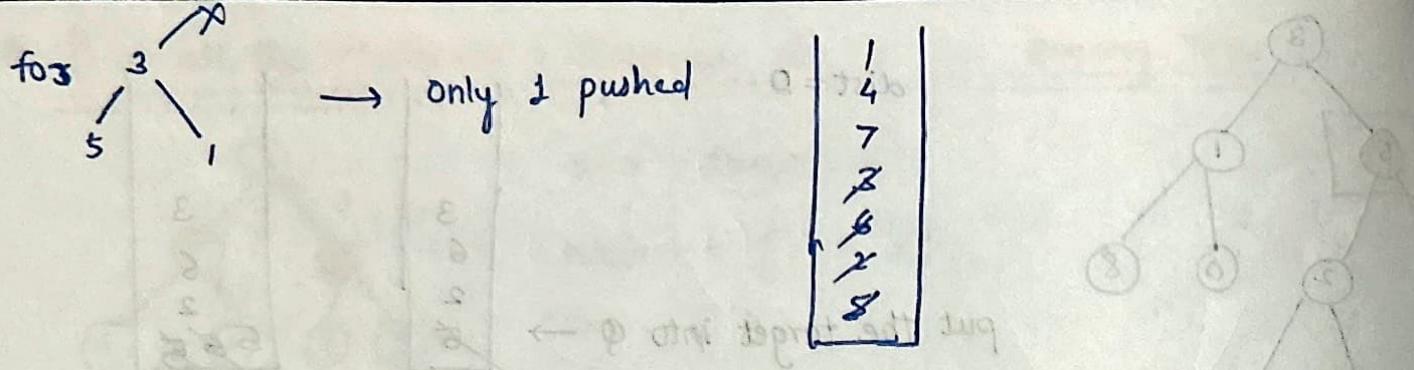
push into  $\emptyset$   
(7 and 4)



Now for 6 →



Nothing will be pushed into Queue



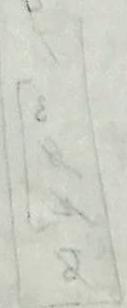
Now distance will be updated to 2  
and compare with this distance to  $k$  ( $k == \text{distance}$ )

Hence ~~your~~ no need to proceed further  
and the nodes present in your queue de is at ' $k$ ' distance  
from target.

→ Summary:-

- Have parent pointers.
- keep on moving one step upward, leftward and rightward
- at  $\text{dist} == k$  STOP!!

② ~~start from the most distant edge number sum of it + common between two edges start with 0 below it~~  
~~parent pointers is a cause of too many ways - ② or ③ wrong~~



Q. other ways  
(P, K, R)



bottom sd the problem  
and then

→ 2 not work

```

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

vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
    unordered_map<TreeNode*, TreeNode*> parent_track;
    markParents(root, parent_track, target); → mapping done!
    unordered_map<TreeNode*, bool> visitedNodes;
    queue<TreeNode*> queue;
    queue.push(target);
    visitedNodes[target] = true;
    int currentLevel = 0;
}

```

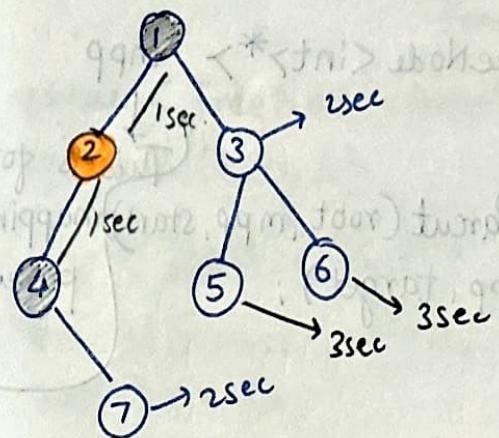
// → PTO.

```

while (!queue.empty()) {
    int size = queue.size();
    if (currentLevel++ == k) break; → reached k distance.
    for (int i=0 ; i<size ; i++) {
        TreeNode* current = queue.front();
        queue.pop();
        if (current->left && !visitedNodes[current->left]) {
            queue.push(current->left);
            visitedNodes[current->left] = true;
        }
        if (current->right && !visitedNodes[current->right]) {
            queue.push(current->right);
            visitedNodes[current->right] = true;
        }
        if (parent_track[current] && !visitedNodes[parent_track[current]]) {
            queue.push(parent_track[current]);
            visitedNodes[parent_track[current]] = true;
        }
    }
    vector<int> result;
    while (!queue.empty()) {
        TreeNode* x = queue.front();
        queue.pop();
        result.push_back(x->val);
    }
    return result;
}

```

• Minimum time taken to burn the binary tree from a node:-



node = 2

Total Time = 3 seconds

BFS Traversal :-

Step 1:- Have the parents pointers ready.

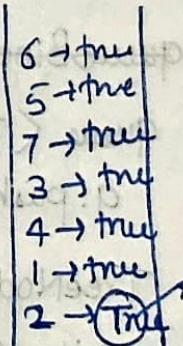
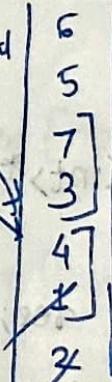
(Refer the code for markParents function of last question)

time = 0

→ node = 2 (above, left, right)

↳ it can burn above(1)

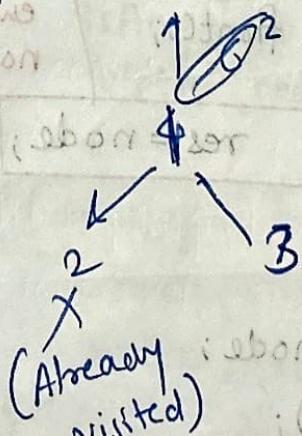
↳ It can burn left (4)



Now pick both the nodes

time = 1

NULL



Now perform the operation on this

⇒ time = 3

time = 2

{ (More - abon) }

abon = [1001 < abon] qm

Here don't update  
5 the time

doesn't burn anyone  
and so 6 also don't burn

```

int findMinTimeToBurn(TreeNode<int>* root, int start) {
    map<TreeNode<int>*, TreeNode<int>*> mpp
    This is for
    mapping of
    parents
}

TreeNode<int>* target = bfsToMapParent(root, mpp, start);
int time = findTimeToBurn(mpp, target);
return time;
}

```

```

TreeNode<int>* bfsToMapParent(
    TreeNode<int>* root,
    map<TreeNode<int>*, TreeNode<int>*> mpp,
    int start);

```

~~queueBinary~~

```
queue<TreeNode<int>> q;
```

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

```
TreeNode<int>* res;
```

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

```
TreeNode<int>* node = q.front();
```

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

```
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; }
```

Since in question  
is given they did  
not give us the  
address of the node

Hence whenever we  
encounter the  
node->data == star

store the addm  
(node)  
and return  
that;

```
int findTimeToBurn (map<TreeNode<int>*, &TreeNode<int>> mpp,
```

```
    TreeNode<int>* target) {
```

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

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

```
map<TreeNode<int>*, bool> visited;
```

```
visited[target] = true;
```

```
int time = 0;
```

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

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

```
    bool bool isAnyNodeBurnt = false;
```

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

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

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

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

```
            isAnyNodeBurnt = true;
```

```
            visited[node->left] = true;
```

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

```
            isAnyNodeBurnt = true;
```

```
            visited[node->right] = true;
```

```
}
```

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

```
            isAnyNodeBurnt = true;
```

```
            visited[mpp[node]] = true;
```

```
}
```

```
}
```

```
        if (isAnyNodeBurnt) {
```

```
            time++;
```

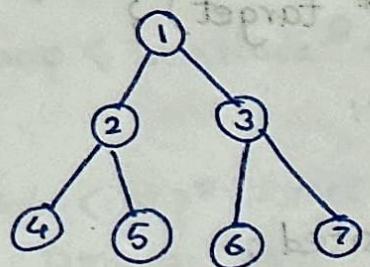
```
}
```

```
}
```

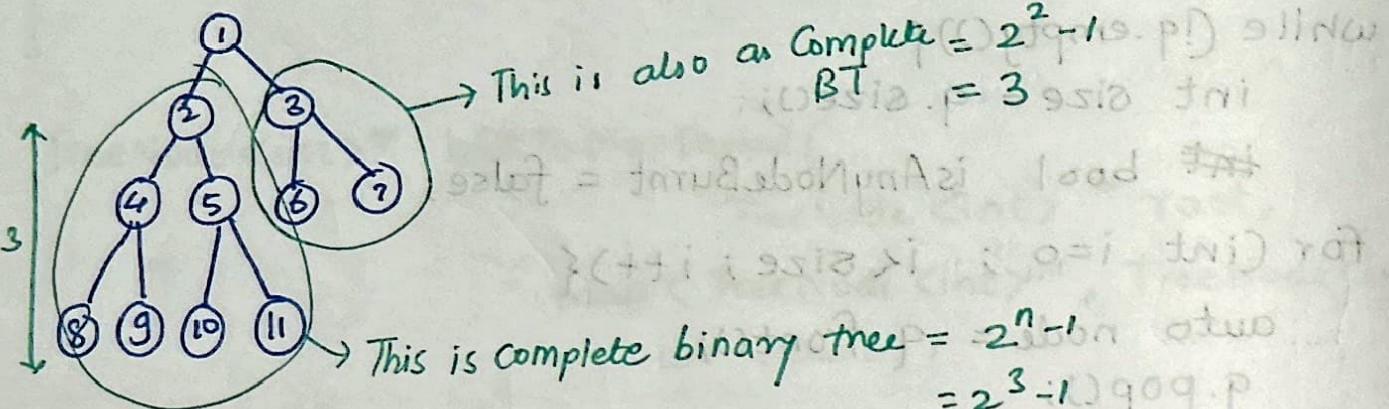
```
return time;
```

```
}
```

Count total number of nodes in a complete binary tree



$$\text{No. of nodes} = 2^3 - 1$$



$$\text{This is complete binary tree} = 2^n - 1$$

$$= 2^3 - 1$$

$$= 7$$

([Height]  $\leftarrow$  [left + right])

$$7 + 3 + 1 = \underline{\underline{11}}$$

$lh = 4$  ( $lh \neq rh$ )  $\rightarrow$  hence it's not complete binary tree  $\rightarrow$  can't apply  $2^{n-1}$  formula directly

$$rh = 3$$

$1 + (\text{from left}) + (\text{ans from right})$

([Left + Right]  $\leftarrow$  [left + right])

```

int countNodes(TreeNode* root) {
    if (root == NULL) return 0;
    int lh = findLeftHeight(root->left);
    int rh = findRightHeight(root->right);
    if (lh == rh) {
        return (pow(2, lh) - 1);
    }
    return 1 + countNodes(root->left) + countNodes(root->right);
}

```

```

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

```

$$S.C = O(\log N)$$

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

At the worst case you will end up traversing the height of the tree  $\Rightarrow \log N$

and every time compute the height

$$\log N * \log N = (\log N)^2$$

```

int findRightHeight(TreeNode* node) {
    int height = 0;
    while (node) {
        height++;
        node = node->right;
    }
}

```

```

return height;
}

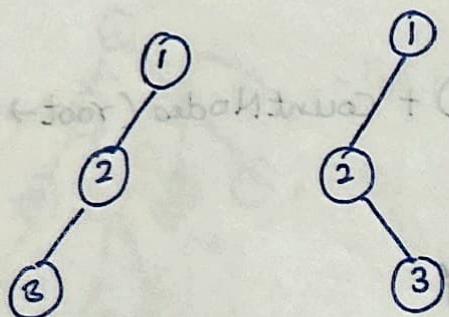
```

- Requirements needed to construct a unique binary tree :-

Q.1) You will be given a preOrder and post-order traversal and you have to create a unique binary tree? Can you?

Example :-

① preOrder :- 1 2 3 (Root - Left - Right)  
postOrder :- 3 2 1 (Left - Right - Root)

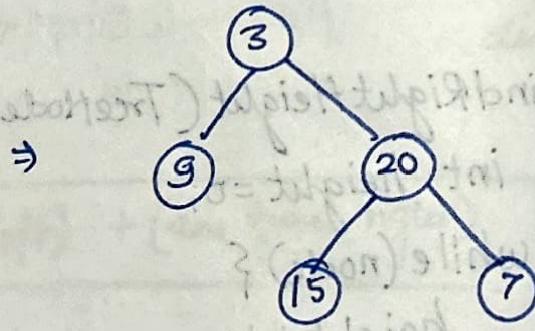
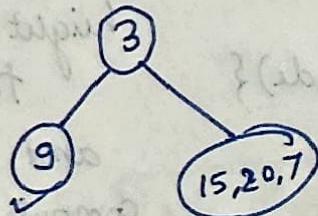


so if you check in this example, we have constructed two binary trees  
Hence with preOrder & postOrder we can't create **UNIQUE** binary tree.

Q.2) What if preOrder & InOrder is given

InOrder = [<sup>Left</sup> 9 <sup>Root</sup> 3 <sup>Right</sup> 15 20 7] - (Left - Root - Right)

PreOrder = [3 <sup>Root</sup> 9 20 15 7] - (Root - Left - Right)

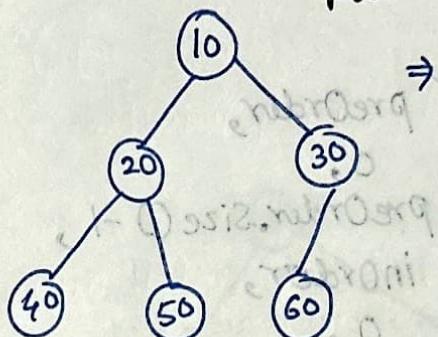


⇒ And this is unique binary tree!

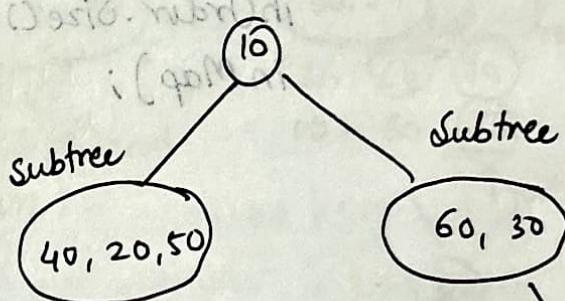
⇒ We can create the unique binary tree only if the  
 ⇒ InOrder + PreOrder  
 ⇒ InOrder + PostOrder  
 is given

Create Binary tree from InOrder & preOrder :-

InOrder = [40, 20, 50, 10, 60, 30] → (Left - Root - Right)  
 preOrder = [10, 20, 40, 50, 30, 60] → (Root - Left - Right)



⇒ This is the tree



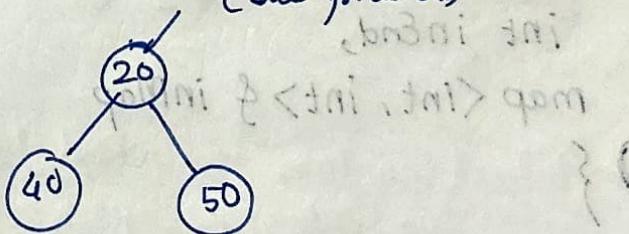
find the InOrder & preOrder

InOrder = 40, 20, 50

preOrder = 20, 40, 50

This is a new problem

(sub problem)



InOrder = 40

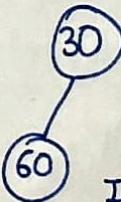
pre: 40

In: 50

pre: go

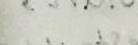
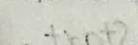
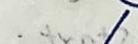
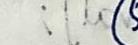
(New Subproblem)

(New Subprob.)



In: 60  
pre: 60

(New Subproblem)



~~TreeNode\* buildTree(vector<int>& preOrder, vector<int>& inOrder)~~

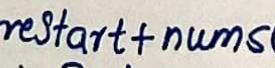
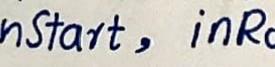
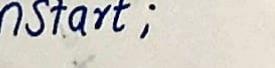
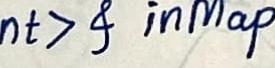
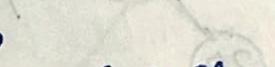
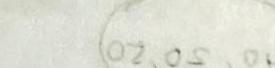
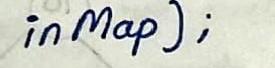
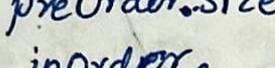
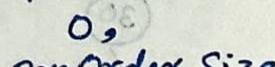
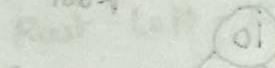
~~map<int, int> inMap;~~

~~for(auto i: inOrder){ for(int i=0; i<inOrder.size(); i++) { inMap[inOrder[i]] = i; }~~

~~3~~ inMap[i]++;

~~3~~

~~TreeNode\* root = buildTree(~~



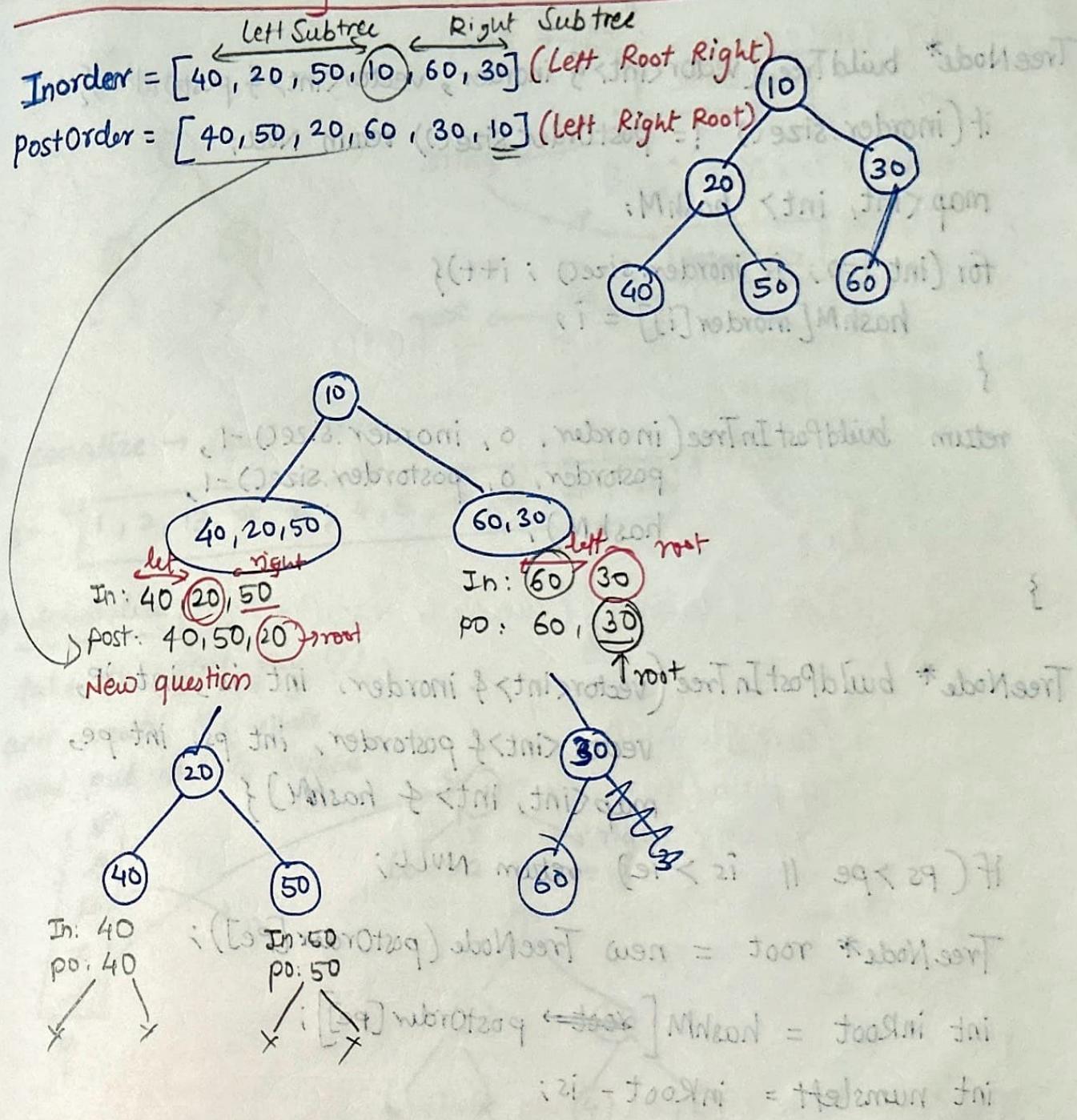
- Construct the binary tree from PostOrder and Inorder :-

Inorder = [40, 20, 50, 10, 60, 30] (Left Root Right)

PostOrder = [40, 50, 20, 60, 30, 10] (Left Right Root)

```

graph TD
    10((10)) --> 20((20))
    10 --> 30((30))
    20 --> 40((40))
    20 --> 50((50))
  
```



(1-foorhri, 21 rebron) seit achterblind = Heit-foor  
@ Heit + Heitwunn + 29, 29 rebrof20q

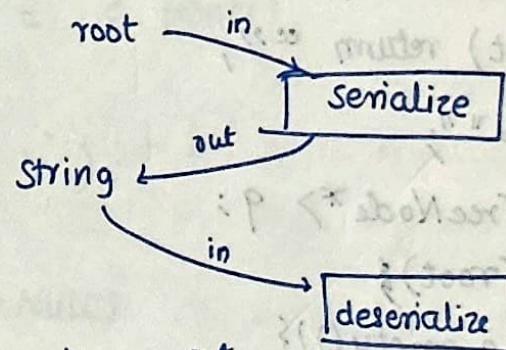
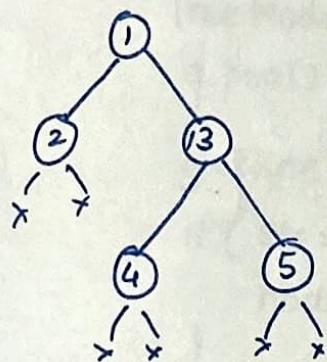
(Mildred 1-29, Helen 29, rebron 1-29)

Whatever postOrder is given the last element is root.

```
TreeNode* buildTree (vector<int>& inorder, vector<int>& postOrder){  
    if (inorder.size() != postOrder.size()) return NULL;  
  
    map<int, int> hashM;  
    for (int i=0; i < inorder.size(); i++) {  
        hashM[inorder[i]] = i;  
    }  
  
    return buildPostInTree(inorder, 0, inorder.size() - 1,  
                           postOrder, 0, postOrder.size() - 1,  
                           hashM);  
}
```

TreeNode\* buildPostInTree (vector<int>& inorder, int is, int ie,  
 vector<int>& postorder, int ps, int pe,  
 map<int, int> &hashM) {  
 if (ps > pe || is > ie) return NULL;  
  
 TreeNode\* root = new TreeNode(postOrder[pe]);  
 int inRoot = hashM[postOrder[pe]];  
 int numsleft = inRoot - is;  
  
 root->left = buildPostInTree(inorder, is, inRoot - 1,  
 postOrder, ps, ps + numsleft - 1, hashM);  
  
 root->right = buildPostInTree(inorder, inRoot + 1, ie,  
 postOrder, ps + numsleft, pe - 1, hashM);  
}

## Serialize and deserialize binary Tree



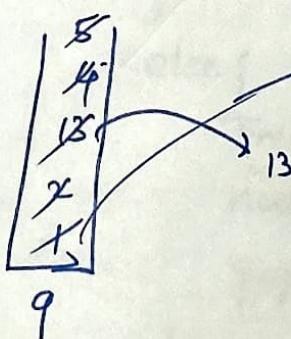
# serialize → Level order

s = "1, 2, 13, #, #, 4, 5, #, #, #, #"

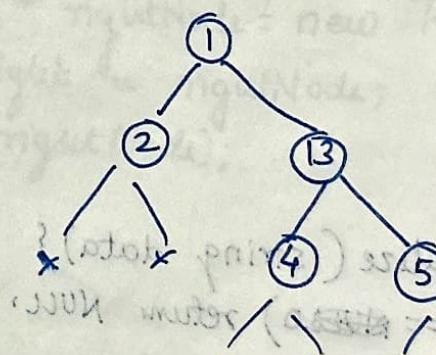
# deserialize

take the first guy (1)

and create the node  
and put it into queue



left = 2  
right = 13



q.is empty

to normal int dsg[]  
int top prev sd  
int (Unfilled ab var maxval) ntmid  
abnode cur int dsg[]

i ((rtz).dat) ablnsort cur = doo \* ablnsort

if p < \*ablnsort) swap

(doo) dsg.p

```
string serialize(TreeNode* root) {
```

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

```
    string s = "";
```

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

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

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

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

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

```
        if (currNode == NULL) s += "#";
```

```
        else {
```

```
            s += (to_string(currNode->val) + ',');
```

```
}
```

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

```
            q.push(currNode->left);
```

```
            q.push(currNode->right);
```

```
}
```

```
    }
```

```
    return s;
```

```
}
```

```
TreeNode* deserialize(string data) {
```

```
    if (data.size() == 0) return NULL;
```

```
    stringstream s(data);
```

```
    string str;
```

```
    getline(s, str, ',');
```

```
    TreeNode* root
```

1, 13, 2, #, #, ---

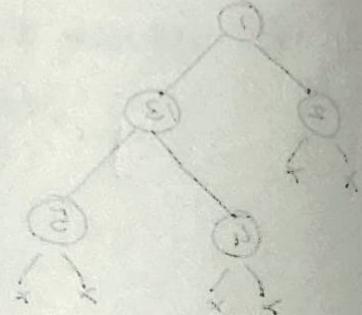
by using `getline` it will automatically  
pick the element `\n`

Whenever you do `getline()` → it  
will pick the ~~one~~ elements

```
TreeNode* root = new TreeNode(stoi(str));
```

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

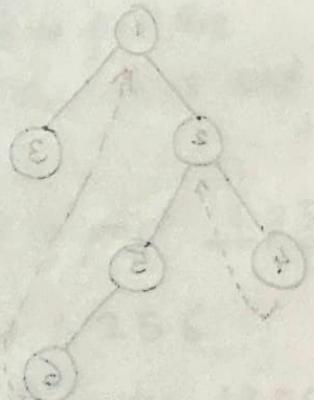
```
q.push(root);
```



```

while (!q.empty()) {
    TreeNode* node = q.front();
    q.pop();
    getline(s, str, ',');
    if (str == "#") {
        node->left = NULL;
    } else {
        TreeNode* leftNode = new TreeNode(stoi(str));
        node->left = leftNode;
        q.push(leftNode);
    }
    getline(s, str, ',');
    if (str == "#") {
        node->right = NULL;
    } else {
        TreeNode* rightNode = new TreeNode(stoi(str));
        node->right = rightNode;
        q.push(rightNode);
    }
}
return root;
}

```



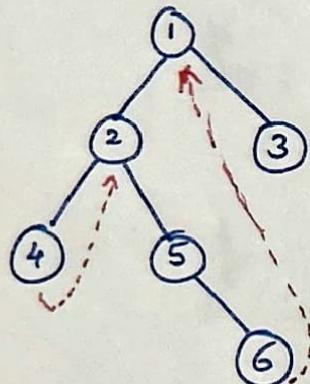
## Morris Traversal

Why morris traversal ?

→ In the recursive traversal  $T.C = O(N)$  &  $S.C = O(N)$   
but in morris traversal  $T.C = O(N)$  &  $S.C = O(1)$

It Only takes  $O(1)$  space for traversal  
cause morris traversal uses

### Threaded binary tree



Inorder = 4 2 5 6 1 3

last node of any  
subtree and if you go back  $\Rightarrow$  root

#### Case 1:-

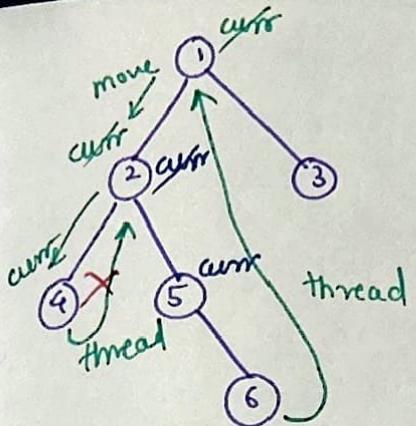
If the left  $\rightarrow$  NULL  
 $\Rightarrow$  print the current node  
 $\Rightarrow$  move to right

#### Case 2:-

Before going to left  $\rightarrow$  whichever the rightmost guy  
of the left sub tree  
will be connected to the current

#### Case 3:-

If the rightmost guy is already point to current  
then remove the thread.



At ④ → if you consider this as subtree  
This will never have left so consider this  
as root and print 4

4 Now use the thread and go to point  $\Rightarrow$  42

Now on ② curve says

When I go to left  $\rightarrow$  the last guy on left sub tree ~~itself~~ is already pointing

so in this case go to the left  $\Rightarrow$  find the rightmost guy and cut off the link take the curve and move right  $\Rightarrow$  ⑤

move right  $\Rightarrow$  ⑤  
⑤ does not have left  $\rightarrow$  so it's already a root  $\rightarrow$  print 425  
root is printed  $\rightarrow$  go to the right  $\rightarrow$  print  $\rightarrow$  4256  
and through the curve you came back to ① print  $\Rightarrow$  4256

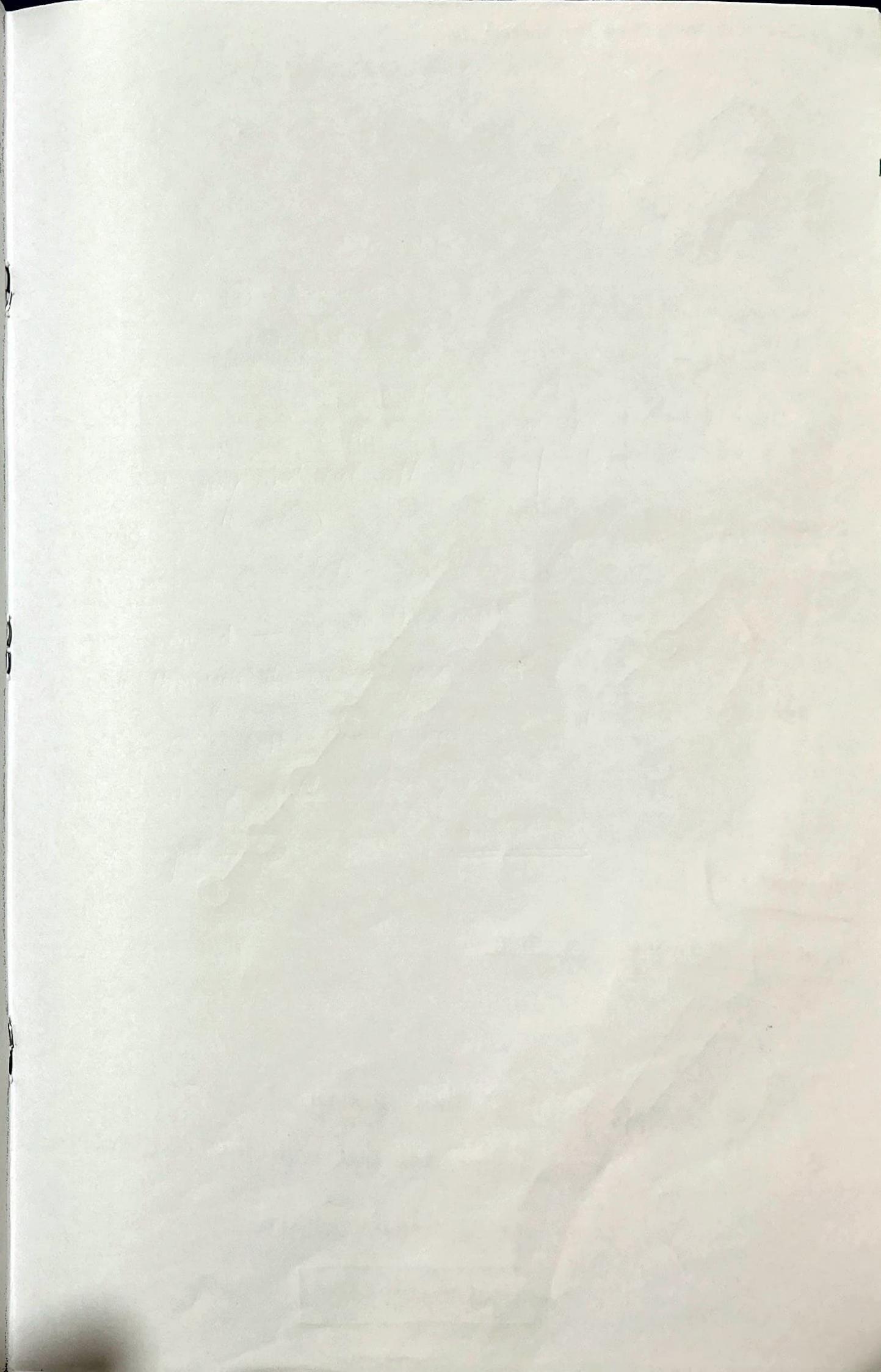
Now on ①'s left already has a thread which is already visited the left arena.

move right  $\rightarrow$  ③  $\Rightarrow$  print 425613

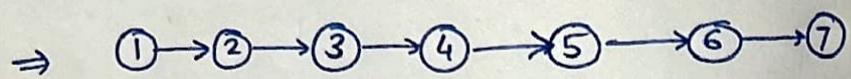
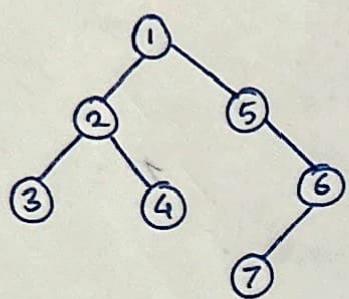
```

vector<int> getInorder(TreeNode* root) {
    vector<int> inorder;
    TreeNode* curr = root;
    while (curr != NULL) {
        if (curr->left != NULL) {
            inorder.push_back(curr->val);
            curr = curr->right;
        } else {
            TreeNode* prev = curr->left;
            while (prev->right && prev->right != curr) {
                prev = prev->right;
            }
            if (prev->right == NULL) {
                prev->right = curr;
                curr = curr->left;
            } else {
                prev->right = NULL;
                inorder.push_back(curr->val);
                curr = curr->right;
            }
        }
    }
    return inorder;
}

```



## Flatten a binary tree to Linked List :-



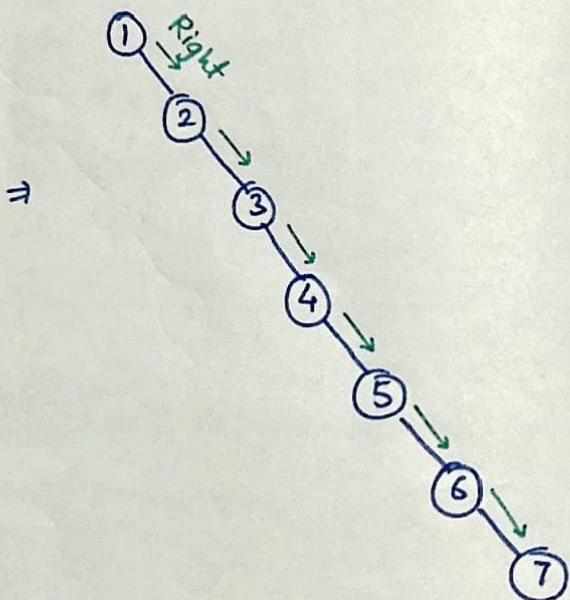
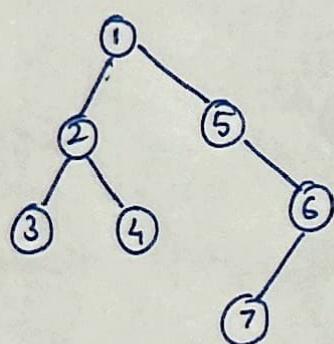
preorder  $\Rightarrow 1 2 3 4 5 6 7$

Whatever is the preorder  $\rightarrow$  you might think like this  
create a node and  
generate LL

BUT THIS IS NOT ACCEPTABLE!!

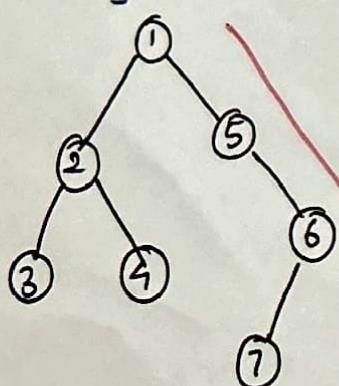
Because we have to flatten the BT

EX:-



**Approach : I**

basically rearrange the tree



go till ⑥ then arrange 7

& After that deal with the left subtree

so the traversal will be

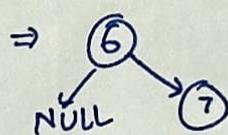
**Right - Left - Root**

```

prev = null;
flatten(node) {
    if(node == null) return;
    flatten(node->right);
    flatten(node->left);
    node->right = prev;
    node->left = null;
    prev = node
}

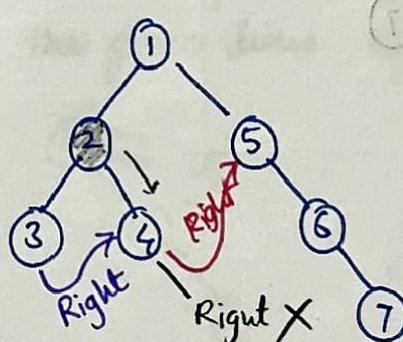
```

next step → ⑥ Right → prev



prev = 6

Now the tree is looking like this after traversal till ① and prev=5



Hence prev=5 and at ④

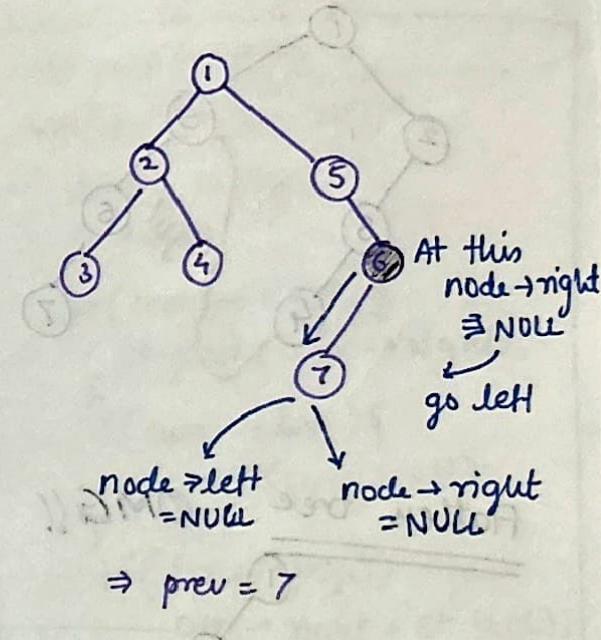
$\text{node} \rightarrow \text{right} = \text{prev}$   
means ④ Right ⑤

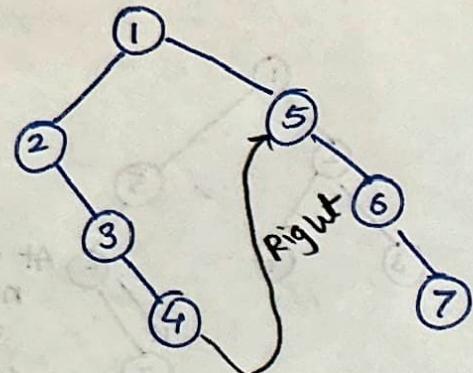
Now prev = 4

come to 2, and then come to ③ Now at ③ Right X

and ③  $\text{node} \rightarrow \text{right} = \text{prev}$   
means ③ Right ④  
prev = 3

Now comes back to ②, It's right → DONE }  
left → DONE } point ② right prev  
② right ③





after pointing the  $\overset{\text{right}}{\underset{\text{(about) next}}{\rightarrow}}$   $\overset{\text{right}}{\underset{\text{(about) next}}{\rightarrow}}$   $\overset{\text{right}}{\underset{\text{(about) next}}{\rightarrow}}$

$\Rightarrow \text{prev} = 2$

Come back to 1

Now its left and right is done!

hence  $\overset{\text{Right}}{\underset{\text{prev}}{\rightarrow}}$

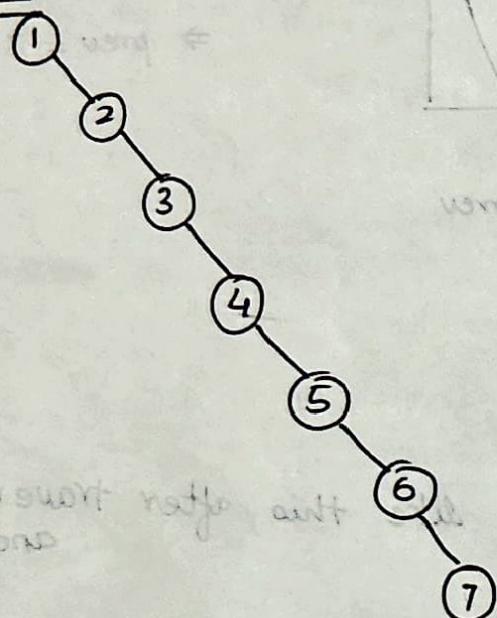
$\overset{\text{Right}}{\underset{\text{about}}{\rightarrow}}$   $\overset{\text{Right}}{\underset{\text{about}}{\rightarrow}}$

$\text{about} = \text{prev} \leftarrow \text{about}$

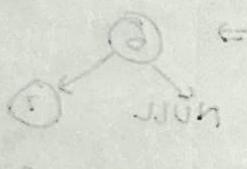
$\text{about} = \text{very}$

$T.C = O(N)$   
 $S.C = O(N)$

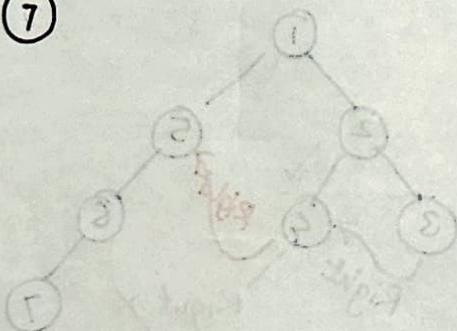
Flatten tree **OMG!!**



① No least recent node until prev is set to null



$a = \text{very}$



$\text{using} = \text{tuplr} \leftarrow \text{about}$

②  $\text{tuplr}$

$p = \text{very}$   $\text{about}$

③ to bns  $\text{p} = \text{very}$   $\text{about}$

X  $\text{tuplr}$  ④ to work ⑤ at some next bns  $\text{p} = \text{very}$   $\text{about}$

$\text{very} = \text{tuplr} \leftarrow \text{about}$  ⑥ bns

⑦  $\text{tuplr}$

⑧  $\text{wrong}$

$\text{p} = \text{very}$

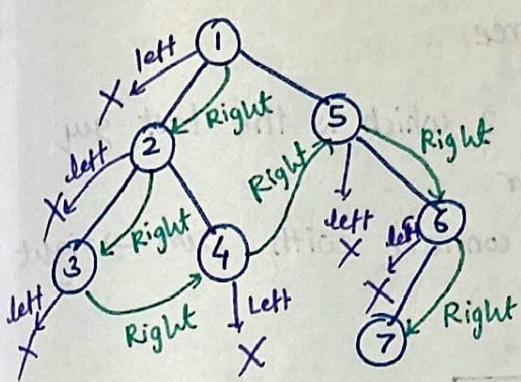
How come p does not change? ⑨ at about some next bns

⑩  $\text{tuplr}$

{  $\text{about} \in \text{tuplr}$  } {  $\text{about} \in \text{tuplr}$  }

## Approach : II

Using stack!



(1) o = 0  
(1) o = 0.2

due to this line

1 → Right = st. top()

1 → leftt = NULL

st  
2  
5  
()

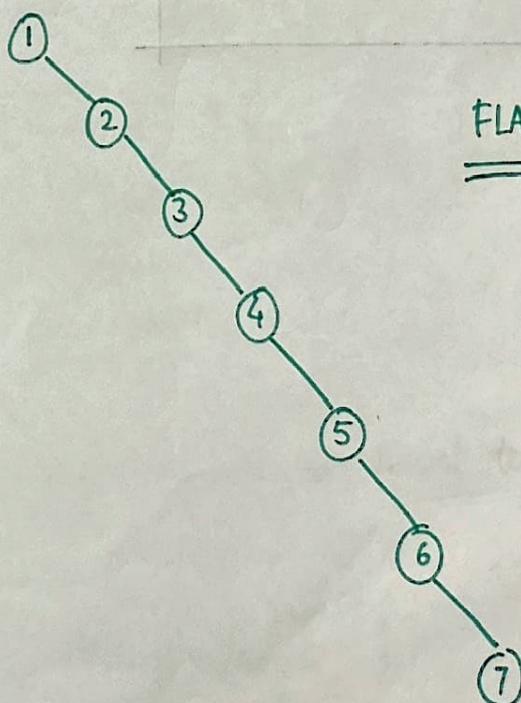
pick  
2  
3  
4

curr

4 → put it into st.

3  
4  
2  
5

Now join the green lines



FLATTEN!!

T.C = O(N)  
S.C = O(N)

st.push(root);  
while (!st.empty()) {

curr = st.top();

st.pop();

if (curr → right) {

st.push(curr → right);

}

if (curr → left) {

st.push(curr → left);

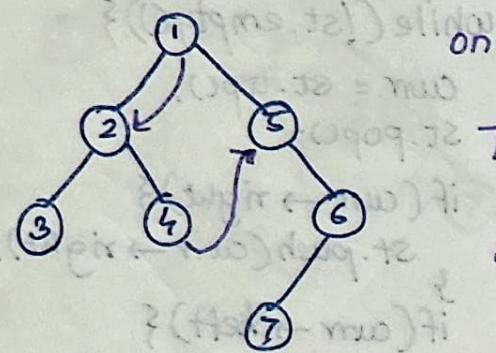
}

if (!st.empty())

curr → right = st.top();

curr → left = NULL;

### Approach : III



on the left subtree

Try to figure out ? which is the last guy  
in the preorder  
and if found connect with curr → right

```

curr = root;
while (curr != NULL) {
    if (curr->left != NULL) {
        prev = curr->left;
        while (prev->right) {
            prev = prev->right;
        }
        prev->right = curr->right;
        curr->right = curr->left;
        curr->left = nullptr;
    }
    curr = curr->right;
}
  
```

T.C = O(N)  
S.C = O(1)

(4) O = O.T  
(4) O = O.B

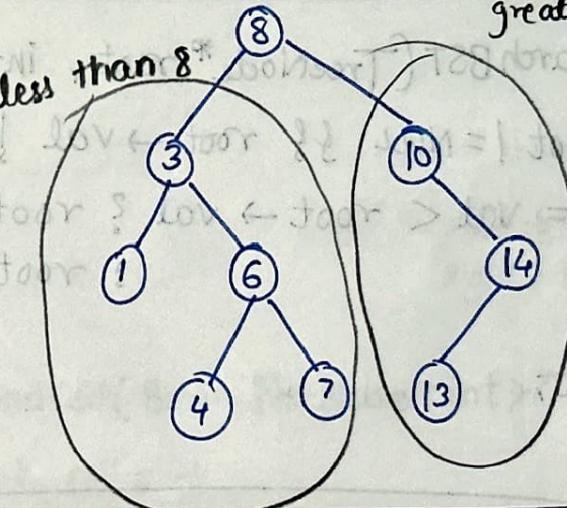
# Binary Search Tree

alt prntwrd for bnd nodes  
sort strings

(mgd) O = O.T

~~→ O(1)~~

BST signifies  $\Rightarrow L < N < R$



- (1) Left Subtree should be BST
- (2) Right Subtree should be BST

Are duplicates allowed?

→ it depends and if so then  $L < N < R$  will be modified.

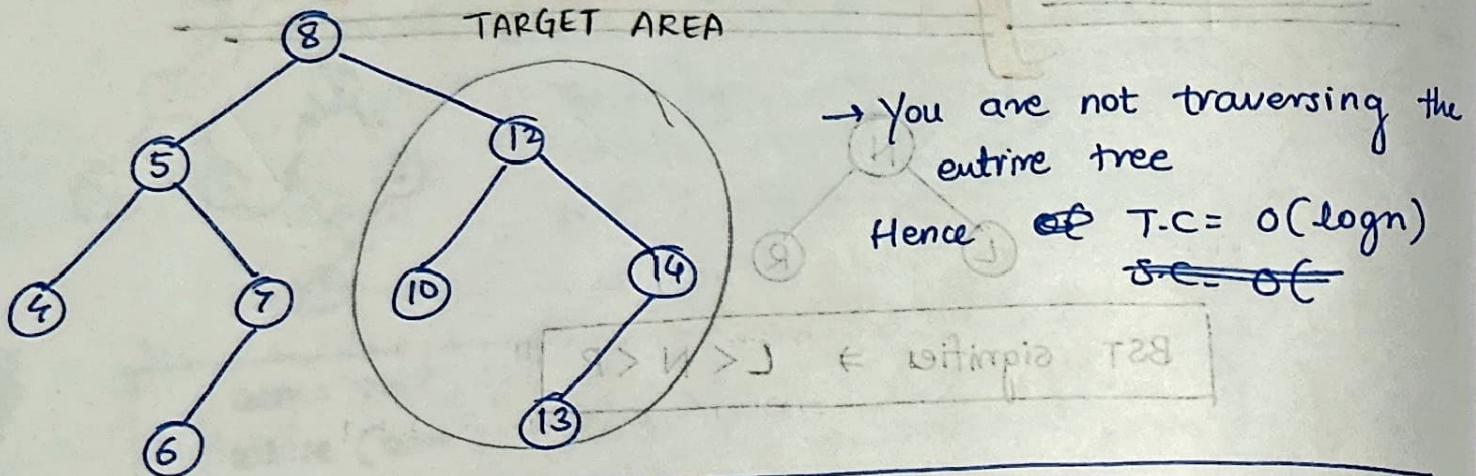
Why BST?

In binary tree, if we want to search any node

$\hookrightarrow O(N)$

but in BST  $\rightarrow O(\log N)$

## • Search in BST :-



```

TreeNode* searchBST(TreeNode* root, int val) {
    while (root != NULL && root->val != val) {
        if (root->val < val) root = root->left;
        else root = root->right;
    }
    return root;
}

```

T2B se bawde sortduz Haf (1)  
T2B se bawde sortduz Trig (2)

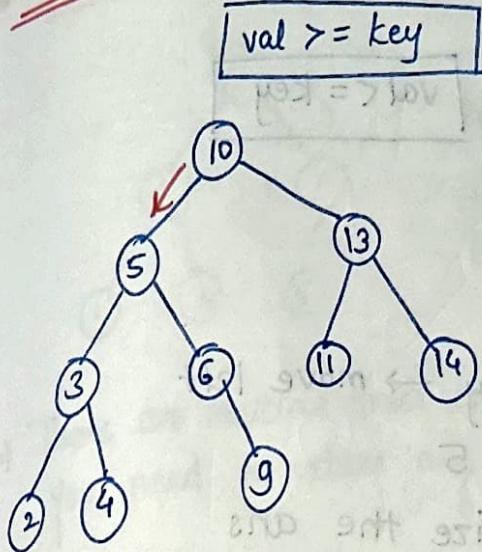
↑ parrallo ato sildub gta  
et hia g>N>J nift & 77 bawde abngets fir  
. bsifibom

? T2B jntw

abon jntw abnget of thow sw fi sort monid ni  
(1) 0 ✓

(Input) o f T2B ni jnd

• Ceil in a BST :-



val  $\geq$  key

key = 8

root = 10  $\Rightarrow$

10  $\geq$  8

But we need to reduce this as much as possible.

Now, key  $<$  root  $\rightarrow$  move left

from 5  $\rightarrow$  go to right  $\Rightarrow$  6 is possible

from 6  $\rightarrow$  go to right  $\Rightarrow$  9

↓

is 9 possible?

YES!

update the ans

go to left  $\Rightarrow$  NULL

Right  $\Rightarrow$  NULL

```
int findCeil(BinaryTreeNode<int>* root, int key) {
```

int ceil = -1;

while (root) {

if (root->data == key) {

return root->data;

}

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

else {

ceil = root->data;

root = root->left;

}

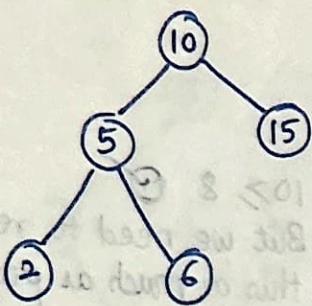
}

return ceil;

}

T.C = O(N)

- Floor in Binary Search Tree



key = 7

$\text{val} \leq \text{key}$

6 < 7

$\text{root} = 10 \xrightarrow{\text{---}} \text{root} > \text{key} \rightarrow \text{move left}$

$$\text{root} \rightarrow \text{data} = 5 \rightarrow \text{ans} = 5$$

Now you have to maximize the ans  
so you have to move right

$$⑥ \quad 6 <= 7 \rightarrow \text{True} \quad \text{ans} = 6$$

move left  $\Rightarrow$  NULL

move right  $\Rightarrow$  NULL

return 6 - 164 dupis

```
int findFloor (BinaryTreeNode<int>* root, int key){
```

```
int floor = -1;
```

```
while (root != NULL) {
```

if ( $\text{root} \rightarrow \text{data} == \text{key}$ ) return  $\text{root} \rightarrow \text{data}$ ;

if ( $\text{root} \rightarrow \text{data} < \text{key}$ ) {

//move right

~~root~~ = floor = root  $\rightarrow$  data;

$\text{root} = \text{root} \rightarrow \text{right};$

}  
else s

else {

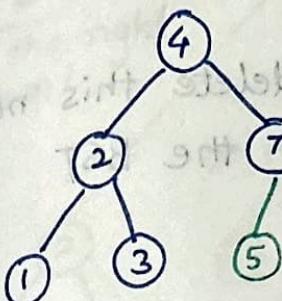
$\text{root} = \text{root} \rightarrow \text{left};$

}

$$\{ \text{H}_2\text{O} = -T$$

return floor;

## Insert a given node in Binary Search Tree



node = 5

Now you have to insert this node in such a way that all conditions for binary search tree will be satisfied.

⇒ There are multiple trees are possible.  
you need to return any of them.

Try to find out the place for the node.

Find where it can be inserted  
and its always on the leaf

For simplicity.

```

TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (root == NULL) { return newTree
        return new TreeNode(val);
    }
}
  
```

while(true) {

TreeNode\* curr = root;

while(true){

if(curr->val < val) {

curr = curr->right;

if(curr->right != NULL) curr = curr->right;

else{

curr->right = new TreeNode(val);

break;

}

}

else{

if(curr->left != NULL) curr = curr->left;

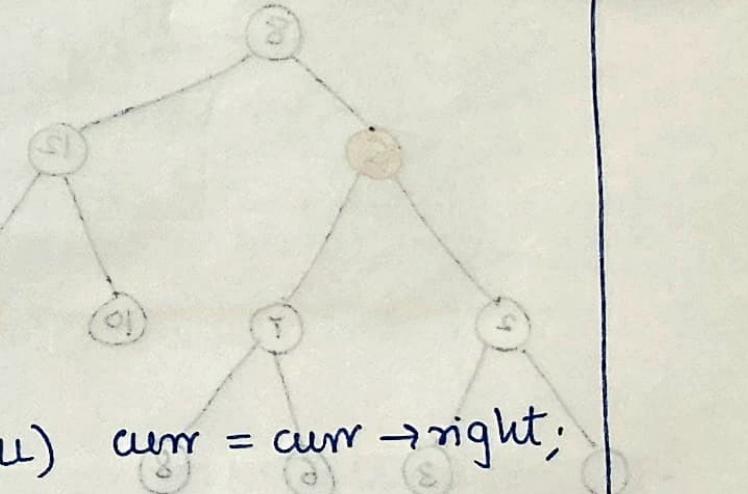
else{

curr->left = new TreeNode(val);

break;

}

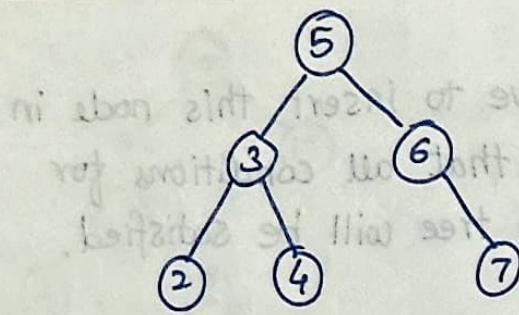
return root;



Insertion

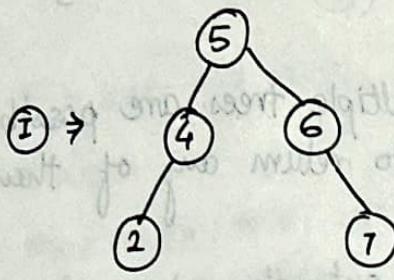
Insertion

## • Delete a Node in Binary Search Tree:-

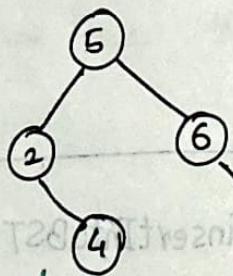


node = 3

You have to delete this node  
and rearrange the BST



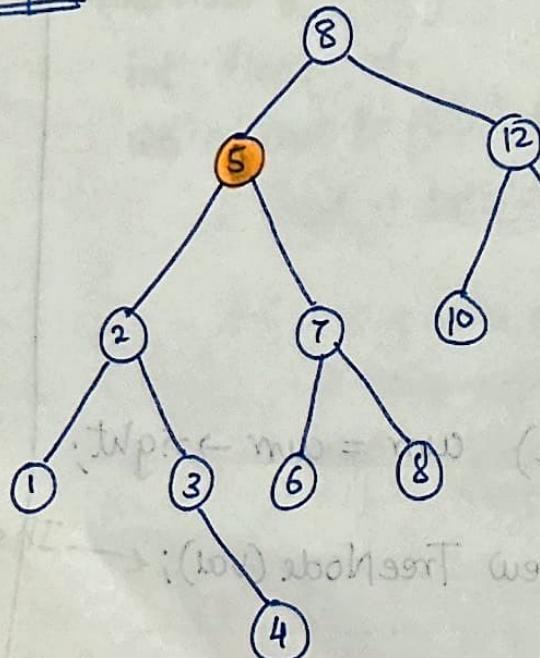
① =>



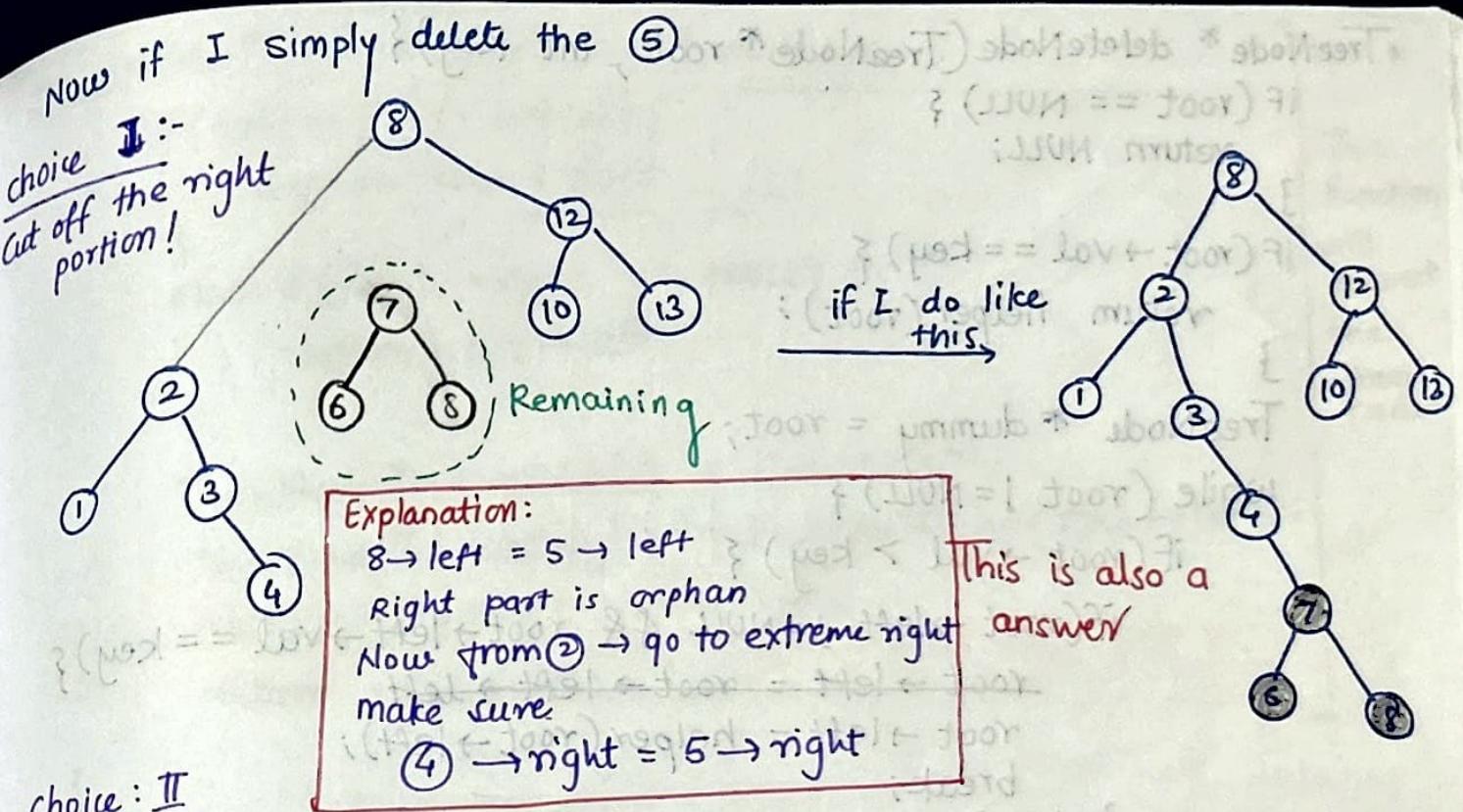
② =>

There are multiple solutions present

Example :-

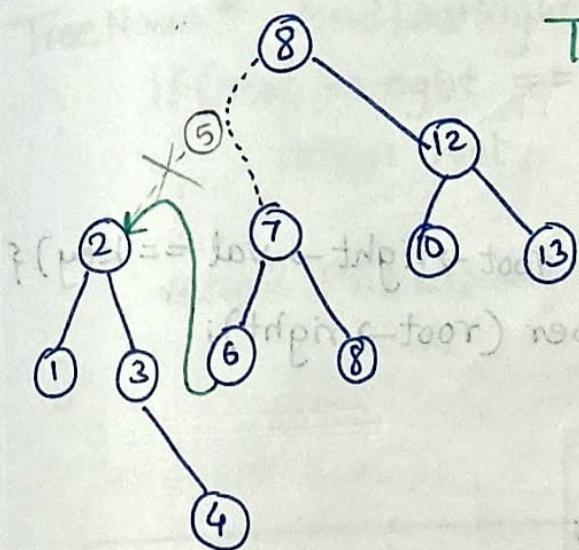


node = 5

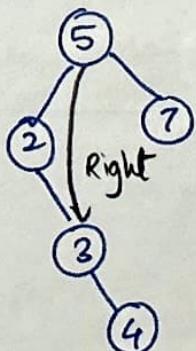


choice II

The smallest number on the right subtree = 6  
connect to this to 2

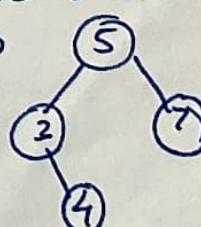


## EDGE CASE



node = 2

So in this there is no left so  
simply attach



```
TreeNode* deleteNode(TreeNode* root, int key) {
    if (root == NULL) {
        return NULL;
    }
    if (root->val == key) {
        return helper(root);
    }
    TreeNode *dummy = root;
    while (root != NULL) {
        if (root->val > key) {
            if (root->left != NULL && root->left->val == key) {
                root->left = root->left->left
                root->left = helper(root->left);
                break;
            } else {
                root = root->left;
            }
        } else {
            if (root->right != NULL && root->right->val == key) {
                root->right = helper(root->right);
                break;
            } else {
                root = root->right;
            }
        }
    }
    return dummy;
}
```

```

TreeNode* helper(TreeNode* root) {
    if (root->left == NULL) {
        return root->right;
    }
    else if (root->right == NULL) {
        return root->left;
    }
}

```

This function will connect the linear nodes

```

TreeNode* rightChild = root->right;
TreeNode* lastRight = findLastRight(root->left);
lastRight->right = rightChild;
return root->left;
}

```

```

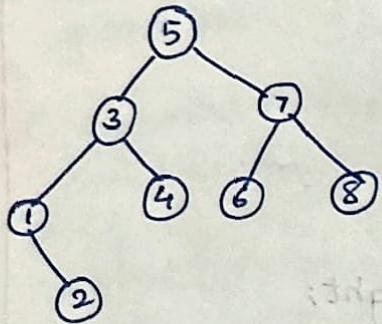
TreeNode* findLastRight(TreeNode* root) {
    if (root->right == NULL) {
        return root;
    }
    return findLastRight(root->right);
}
}

```

Time Complexity  $\Rightarrow O(H)$   
Space Complexity  $\Rightarrow O(1)$

$$\text{Time Complexity} (H) = O(H)$$

- $K^{th}$  smallest element in BST :-



Write down the inorder :-

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

Impl!!

Inorder of binary Search tree is always in the sorted order

Maintain the count as "0"

and whenever you encounter the node  $\rightarrow$  count++;

Three Approaches :-

① Recursive :-  $T.C = O(N)$   
 $S.C = O(N)$

② Iterative :  $T.C = O(N)$   
 $S.C = O(N)$

③ Morris traversal :  $T.C = O(N)$   
 $S.C = \cancel{O(N)} O(1)$

→ preferred solution

- $k^{th}$  largest element in BST

logic  $\Rightarrow$   $k^{th}$  largest =  $(N - k + 1)^{th}$  smallest element  
 $(N - k + 1)^{th}$  smallest element

```

int KthSmallest(TreeNode* root, int k) {
    int count = 0, result = 0;
    traverse(root, k, count, result);
    return result;
}

void traverse(TreeNode* node, int k, int& count, int& result) {
    if (node == NULL) return; result,
    traverse(node->left, k, count, result);
    count++;
    if (count == k) {
        result = node->val;
        return;
    }
    traverse(node->right, k, count, result);
}

```

---

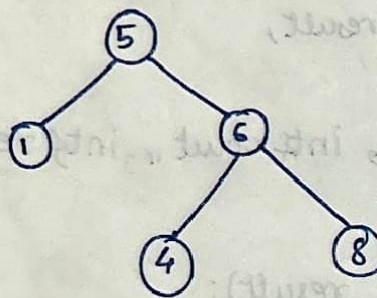
```

int KthLargest(TreeNode* root, int k) {
    int count = 0, result = 0;
    traverse(node, k, count, result);
    return result;
}

void traverse(TreeNode* node, int k, int& count, int& result) {
    if (node == NULL) return;
    traverse(node->right, k, count, result); Opposite Traversal
    count++;
    if (count == k) {
        result = node->val;
        return;
    }
    traverse(node->left, k, count, result);
}

```

- Check if tree is a BST or BT
- for satisfying the condition of BST  $\Rightarrow$



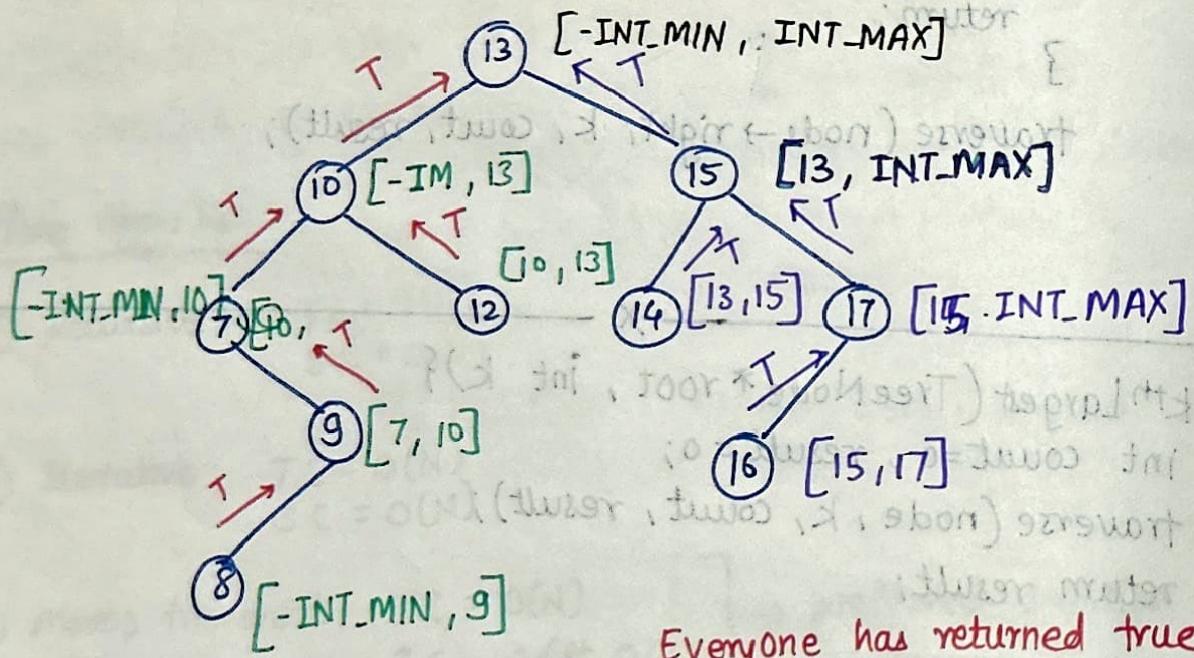
for ④ It should be lesser than ⑥  
AND should be greater than ⑤

Hence we need to consider the scenario

Intuition:-

$$\text{node} = [L, H]$$

Example :- for ④ it should lie b/w  $[5, 6]$   $\Rightarrow$  false



Everyone has returned true

$\Rightarrow$  Hence its a valid BST

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

$$S.C = O(1)$$

bool isValidBST(TreeNode\* root) {

    return isValid(root, INT\_MIN, INT\_MAX);

}

bool isValid(TreeNode\* root, long minValue, long maxValue) {

    if (root == NULL) return true;

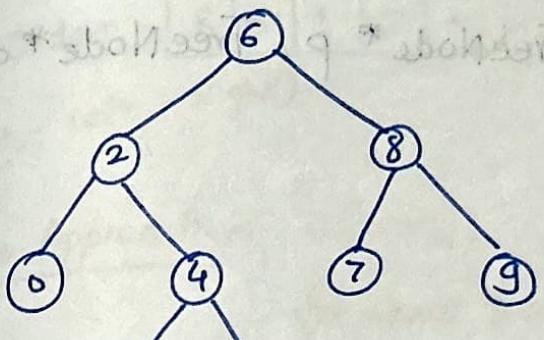
    if (root->val >= maxValue || root->val <= minValue) return false;

    return isValid(root->left, minValue, root->val);

    if (isValid(root->right, root->val, maxValue))

        return true;

## Lowest Common Ancestor in Binary Search Tree :-

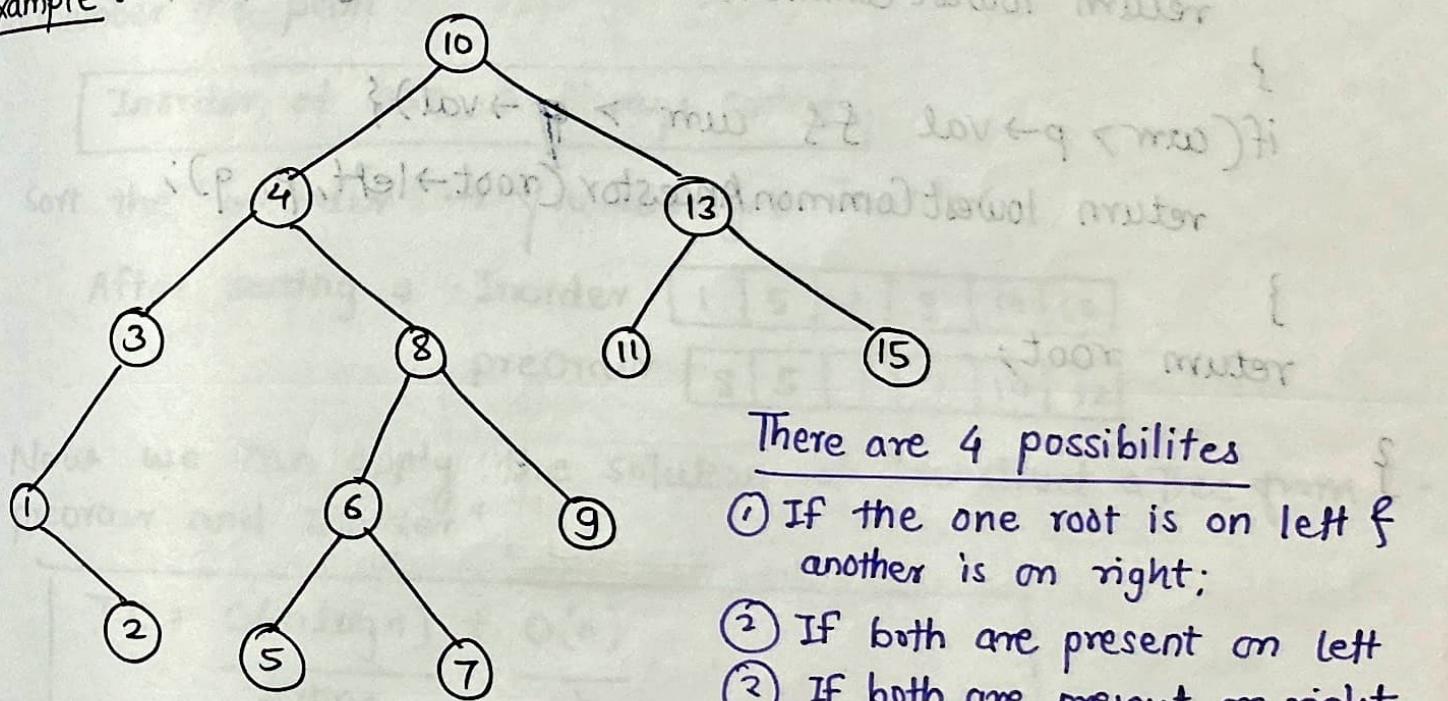


$$LCA(2, 5) = 2$$

$$LCA(0, 5) = 2$$

first intersection path (from bottom)

Example :-



There are 4 possibilities

- ① If the one root is on left & another is on right;
- ② If both are present on left
- ③ If both are present on right
- ④ If there is a root node from one amongst them.

let's discuss  $L(5, 9)$

Now 5 & 9 both are lesser than 10. move to left  $\Rightarrow$  4

④  $\rightarrow$  5 & 9 are greater  $\rightarrow$  move to right  $\Rightarrow$  8

on ⑧  $\rightarrow$  5 is on left and 9 is on right  $\rightarrow$  This is where the path has been split so this is my LCA

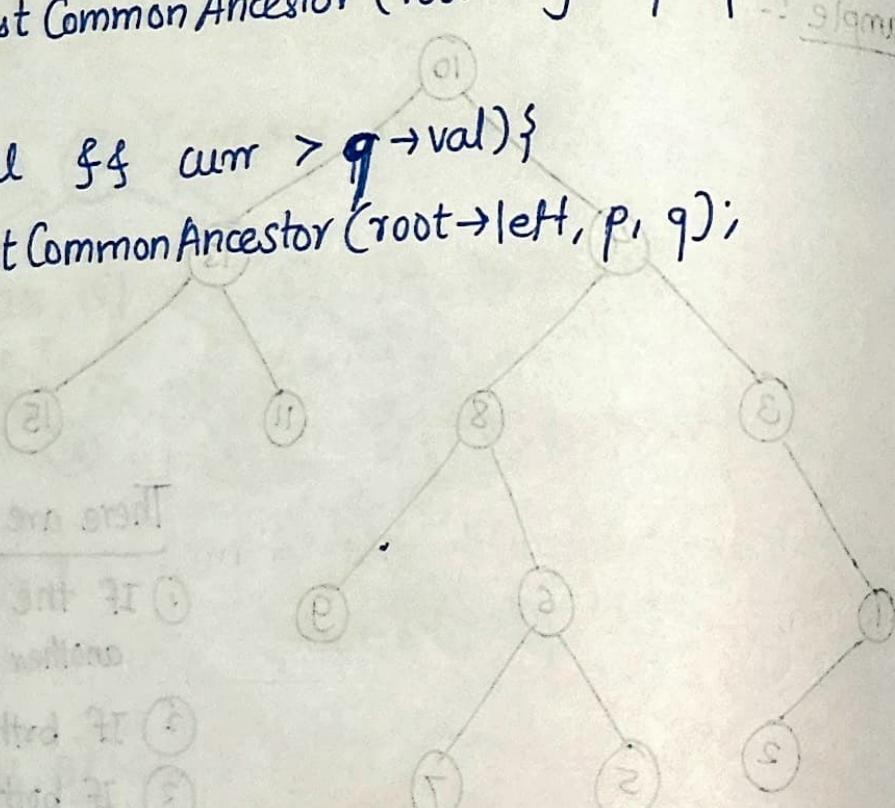
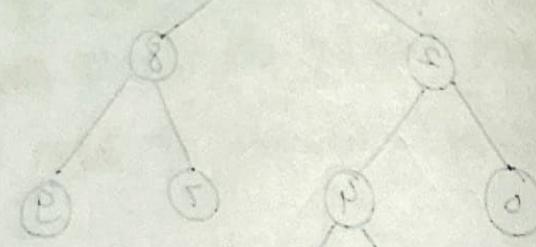
$$T.C = O(H)$$

$$S.C = O(1)$$

```

TreeNode* LowestCommonAncestor(TreeNode* root,
                                TreeNode* p, TreeNode* q) {
    if (root == NULL) {
        return NULL;
    }
    int curr = root->val;
    if (curr < p->val && curr < q->val) {
        return lowestCommonAncestor(root->right, p, q);
    }
    if (curr > p->val && curr > q->val) {
        return lowestCommonAncestor(root->left, p, q);
    }
    return root;
}

```



(p, q) = lowest common ancestor

p ∈ left of 8 and q ∈ right of 8

8 ∈ right of 8 and p ∈ left of 8

need to check if p is in right of 8

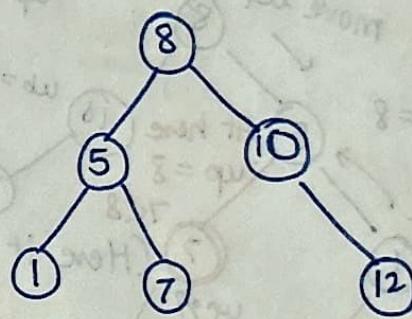
ADJ

$$(H)O = OT$$

Construct a BST from preOrder traversal :-

preOrder  
(Root - Left - Right)

8	5	1	7	10	12
---	---	---	---	----	----



Naive Approach :-  $O(N \times N)$   
for skew tree

Better Method :-

Remember the point

Inorder of BST  $\Rightarrow$  Always sorted

sort the preorder  $\rightarrow$  you will get Inorder

After sorting  $\Rightarrow$  Inorder 

1	5	7	8	10	12
---	---	---	---	----	----

preOrder 

8	5	1	7	10	12
---	---	---	---	----	----

Now we can apply the solution of "construct a Tree from preOrder and Inorder".

$$T.C = \frac{O(n \log n)}{\text{sorting}} + O(n) \quad \downarrow \text{construct the tree}$$

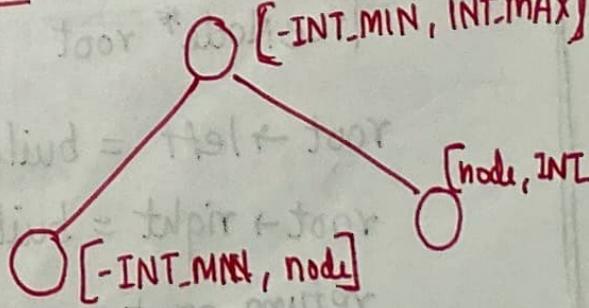
$$S.C = O(n)$$

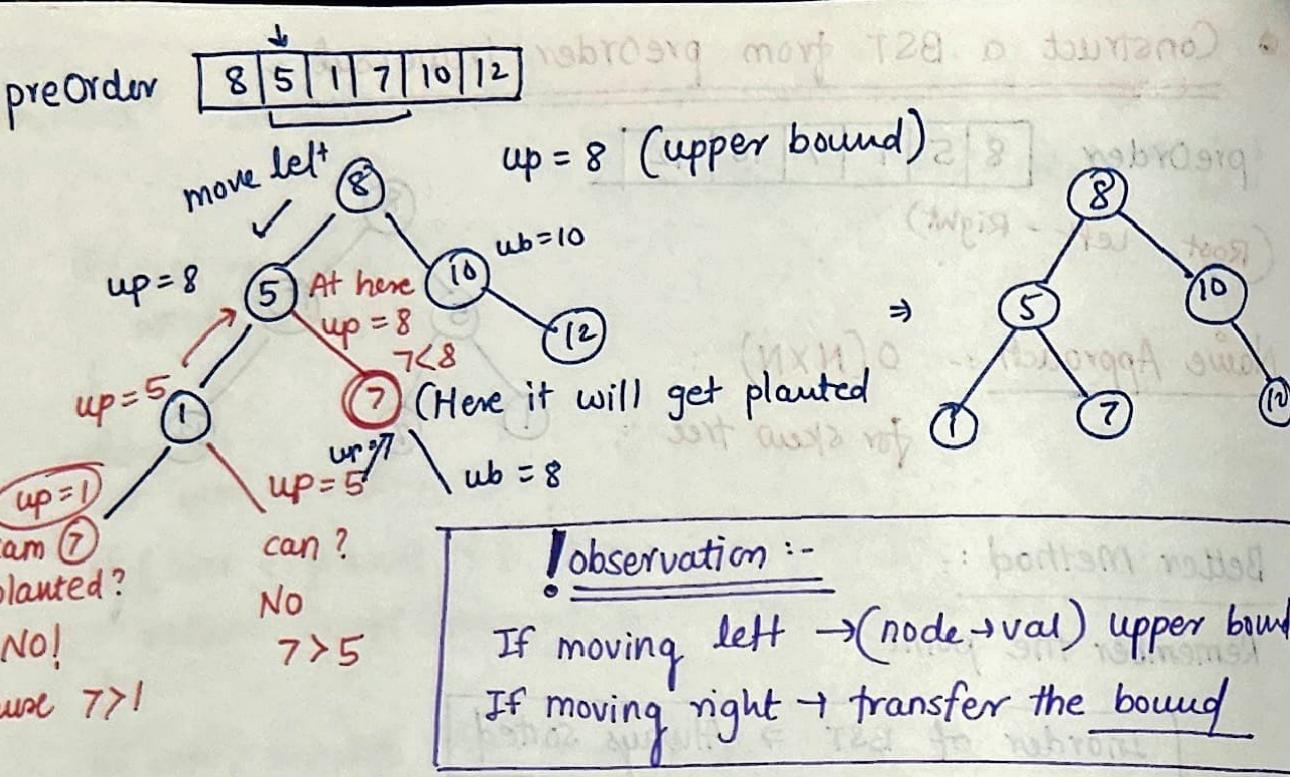
Efficient Method :-

preOrder 

8	5	1	7	10	12
---	---	---	---	----	----

Ref: check if tree is BST or not





$$T.C = O(3N)$$

You are visiting 3 times a single node

$$T.C \approx O(N)$$

$$S.C = O(1)$$

TreeNode\* bstFromPreOrder(~~TreeNode\*~~ vector<int>& A) {

    int i = 0;

    return build(A, i, INT\_MAX);

}

TreeNode\* build(vector<int>& A, int i, int bound) {

    if (i == A.size() || A[i] > bound) return NULL;

    TreeNode\* root = new TreeNode(A[i++]);

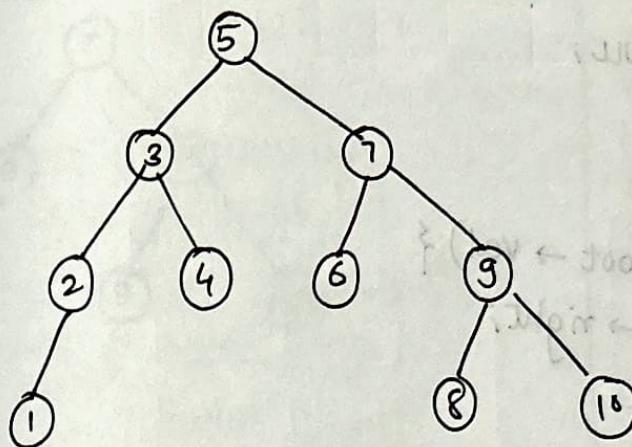
    root->left = build(A, i, root->val);

    root->right = build(A, i, bound);

    return root;

}

## Inorder Successor in Binary Search Tree :-



Inorder  $\Rightarrow 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10$

$\uparrow$   
Inorder Successor

If the no successor is present  $\rightarrow$  return NULL.

### Efficient Approach :-

successor = NULL      val = 8

At root  $\rightarrow 5 \rightarrow$  I should move to right

Now at 7  $\rightarrow$  I should move to right

at 9  $\rightarrow$  Here we can say 9 is the successor  
But are we sure?

NOT  $\rightarrow$  cause there might be a possibility  
that the successor (min) would  
present at left

I know 9  $\xrightarrow{\text{Right}}$  10 will not be my successor

Hence I need someone who is  $> 8$  and  $\leq 9$

move left  $\rightarrow$  8 found

Now move right we reach NULL.

Hence our successor is 9.

Time Complexity =  $O(H)$

Space Complexity =  $O(1)$

```

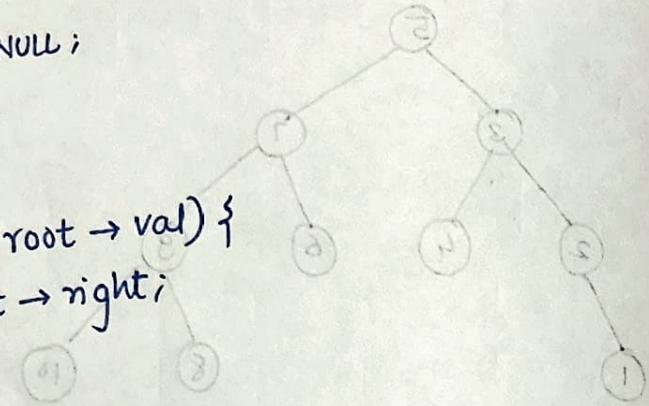
TreeNode* inOrderSuccessor(TreeNode* root, TreeNode* x) {
    TreeNode* successor = NULL;
    while (root != NULL) {
        if (x->val <= root->val) {
            root = root->right;
        } else {
            successor = root;
            root = root->left;
        }
    }
    return successor;
}

```

```

graph TD
    C((C)) --> B((B))
    C((C)) --> E((E))
    B((B)) --> A((A))
    B((B)) --> D((D))
    E((E)) --> F((F))
    E((E)) --> G((G))
    G((G)) --> H((H))
    G((G)) --> I((I))

```



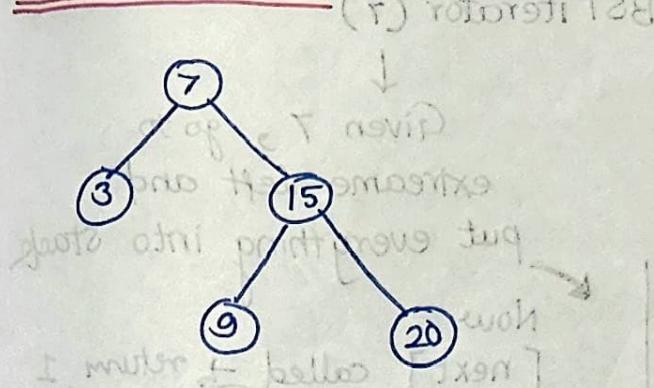
# predecessor :-

```

TreeNode* inOrderPredecessor(TreeNode* root, TreeNode* x) {
    TreeNode* predecessor = NULL;
    while (root != NULL) {
        if (x->val >= root->val) {
            predecessor = root;
            root = root->left;
        } else {
            predecessor = root;
            root = root->right;
        }
    }
    return predecessor;
}

```

## BST Iterator :-



BST iterator(7)

next  
next  
hasnext  
next  
hasnext  
next.  
hasnext  
next.  
hasnext

Inorder 

3	7	9	15	20
---	---	---	----	----

Initially pointer will be pointing to a NULL

- next → return 3
- next → return 7
- hasnext → return True
- next → 9
- hasnext → return True
- next → 15
- hasnext → True
- Next → 20
- hasnext → false.

Inorder  $\Rightarrow$  Left  $\leftrightarrow$  Root - Right

$$\begin{aligned} (4) O &= O.H \\ (1) O &= O.T \end{aligned}$$

$$(x)(x) O \leftarrow O.T = \text{also } O.H$$

$$(1) O = O.T$$

for returning the

T.C = O(1)

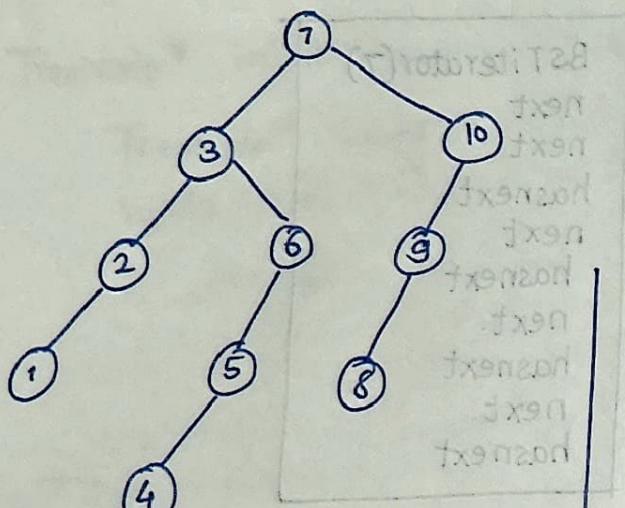
S.C = O(N)

for storing the inorder

This question is easy if we are allowed to store the inorder but if they don't then its a tricky question

The constraint  $\Rightarrow$  O(H)

O.H  
T.C



## BST iterator (7)

Given 7, go to  
extreme left and  
put everything into stack

- Now [next] called → return 1  
check for right
- [next] called → return 2  
check for right x
- [next] called → return 3  
check for right ⇒ yes!

Now go to right & put everything into stack (~~left~~)

$\epsilon$  minor  $\in$   $\text{tex}_n$

[hasnext] → if stack empty?  
→ No → return true

next → 4 → return 4  
If 4 has right → x

next → 5 → return 5

$\{$   $o_1 \in \text{task} \leftarrow$   
 $o_2 \in \text{task} \leftarrow$

This is how this will work!

pushing  $n$  elements into stack  $\rightarrow N$

& there might be N

$$[\text{next}] \text{ calls} = T.C \Rightarrow O(A/x)$$

$$S.C = O(H)$$
$$T.C = O(1)$$

$$T.C = O(1)$$

```
class BSTIterator {
```

```
    stack<TreeNode*> myStack;
```

```
public:
```

```
    BSTIterator(TreeNode* root) {
```

```
        myStack.push(
```

```
            pushAll(root);
```

```
}
```

```
    bool hasNext() {
```

```
        return !myStack.empty();
```

```
}
```

```
    int next() {
```

```
        TreeNode* tmpNode = myStack.top();
```

```
        myStack.pop();
```

```
        pushAll(tmpNode->right);
```

```
        return tmpNode->val;
```

```
;--x } = 2 <= i < s
```

```
private:
```

```
    void pushAll(TreeNode* node) {
```

```
        for(; node != NULL;
```

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

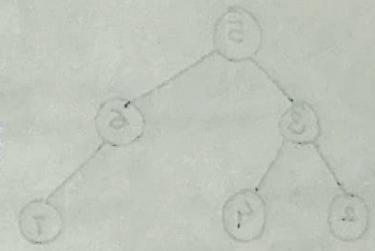
```
                myStack.push(node);
```

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

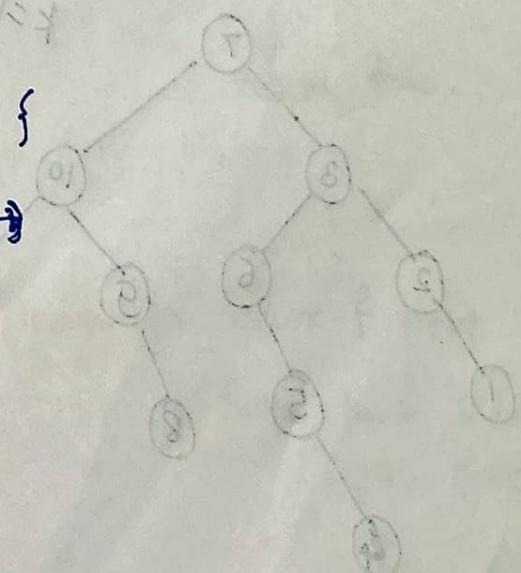
```
}
```

```
}
```

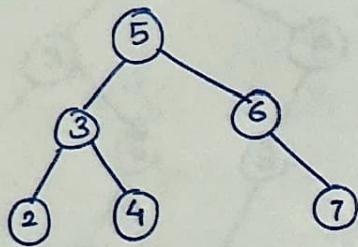
```
}
```



$$(n)O + (n)O = O.T$$
$$(n)O = O.2$$



## Two Sum IV in BST



return true if pair exists, return false

Brute force :-

why inorder have taken? cause its always sorted!

find the inorder  $\Rightarrow$ 

2	3	4	5	6	7
↑	↑				↑
l	r				r

 GOT IT!! r

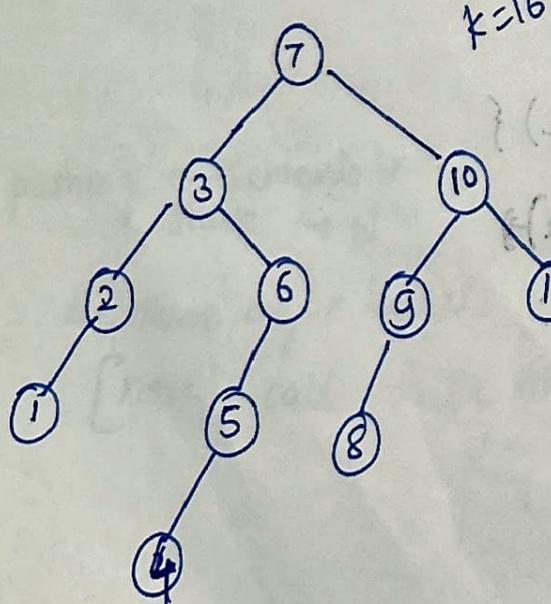
then take  $l=0$   
 $r = \text{size} - 1$       if  $2 + 7 = k \rightarrow \text{No} \Rightarrow 9 > 5$   
 then reduce the greater no. index

$T.C = O(N) + O(N)$   
 $S.C = O(N)$

$2 + 5 = 7 > 5 \Rightarrow r--;$

$2 + 4 = 6 > 5 \Rightarrow r--$

$2 + 3 = (5 = 5) \Rightarrow \text{return true}$



$k=16$

We will try to reduce the space complexity, cause in the worst case we need to travel each node.

Now you can take a help from the concept BST Iterator  
for before concept

Now If you iterate DFS  $\Rightarrow$  Right - Node - Left

$\Downarrow$   
resulting in descending  
order sort

In next() concept we pushed everything which is present on left  
but In before() concept we push everything which is present on RIGHT

Initially my  $i^{th}$  pointer stands at  $i = \text{next}()$   
and  $j^{th}$  pointer  $= j = \text{before}()$

next will be the smallest element  $\Rightarrow i=1$

before will be the largest element  $\Rightarrow j=11$

$1+11 = 12 \rightarrow$  less than  $k=16 \rightarrow$  increase the  $i$   
(move upward)

$2+11 = 13$  (move upward)

$3+11 = 14$  increase

Now  $i$  is moved to ④  $\Rightarrow 4+11 = 15$  (increase)  
 $5+11 = 16$  MATCHED

$$\boxed{\begin{aligned} T.C &= O(N) \\ S.C &= 2 \times O(H) \end{aligned}}$$

$\downarrow$   
return true.

because storing for before & next  
hence twice is the  
space complexity

```

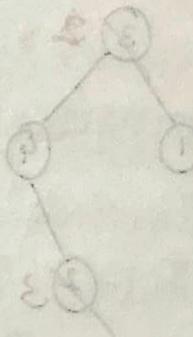
class BSTIterator {
    stack<TreeNode*> myst;
    // Reverse → true ⇒ before
    // Reverse → false ⇒ next
    bool reverse = true;
public:
    BSTIterator(TreeNode* root, bool isReverse) {
        reverse = isReverse;
        pushAll(root);
    }
    // Returns whether we have a next smallest number
    bool hasNext() {
        return !myst.empty();
    }
    // Returns the next smallest number
    int next() {
        TreeNode* tmpNode = myst.top();
        myst.pop();
        if (!reverse) {
            pushAll(tmpNode->right);
        } else {
            pushAll(tmpNode->left);
        }
        return tmpNode->val;
    }
}

```

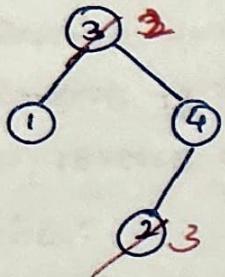
private:  
 void pushAll(TreeNode\* node){  
 while (node != NULL) {  
 mySt.push(node);  
 if (reverse == true) {  
 node = node->right;  
 } else {  
 node = node->left;  
 }
 }

class Solution {

public:  
 bool findTarget(TreeNode\* root, int k) {  
 if (root == NULL) return false;  
 BSTIterator left(root, false);  
 BSTIterator right(root, true);  
  
 int i = left.next();  
 int j = right.next();  
  
 while (i < j) {  
 if (i + j == k) return true;  
 else if (i + j < k) i = left.next();  
 else {  
 j = right.next();  
 }
 }
 }



## • Recover BST :-



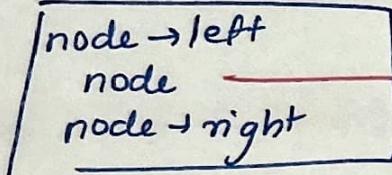
In the given tree, two nodes are swapped  
you have to recover the BST

If you swap 2 & 3 then the BST will  
be recovered.

Brute :-

get  $\Rightarrow$  Inorder traversal and sort it

and while going to

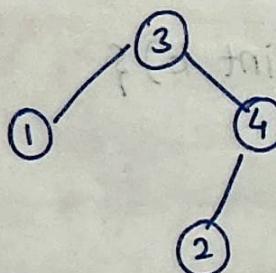


when it comes to  
"node" check  
if the node is in  
inorder(idx)

sorted inorder:-

1	2	3	4
---	---	---	---

idx = 0



node  $\Rightarrow$  1  $\rightarrow$  inorder[idx] = 1 ✓ idx++  
and then at 3  $\rightarrow$  inorder[idx] = 2

swap

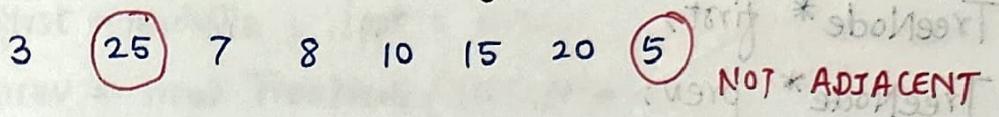
this is how you can process.

$$T.C = 2N + N \log N$$

$$S.C = O(N)$$

## Better Approach

i) Swapped nodes are not adjacent



ii) Swapped nodes are adjacent.

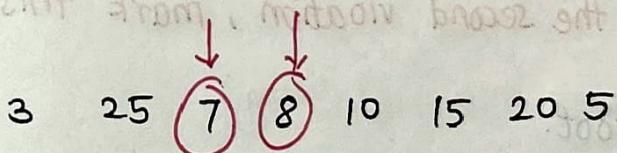


### Case I)

while traversal from 25 if we move to 7 It decreased  
(Inorder)  $\rightarrow$  Hence this is the first  
violation

Now from 20  $\rightarrow$  5 the Again violation second violation  
 $\rightarrow$  swap(first & last)

### Case II)



first = 7

middle = 8

No second violation found  $\Rightarrow$  swap(middle, first)

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

$$S.C = O(1)$$

# Class Solution;

private:

TreeNode\* first;

TreeNode\* prev;

TreeNode\* middle;

TreeNode\* last;

private:

void inorder(TreeNode\* root) {

if (root == NULL) return;

inorder(root->left);

if (prev != NULL && (root->val < prev->val)) {

// If this is the first violation, mark these two nodes

// as 'first' and 'middle'

if (first == NULL) {

first = prev;

middle = root;

}

// If this is the second violation, mark this node as

else {

last = root.

}

}

// mark this node as previous

prev = root;

inorder(root->right);

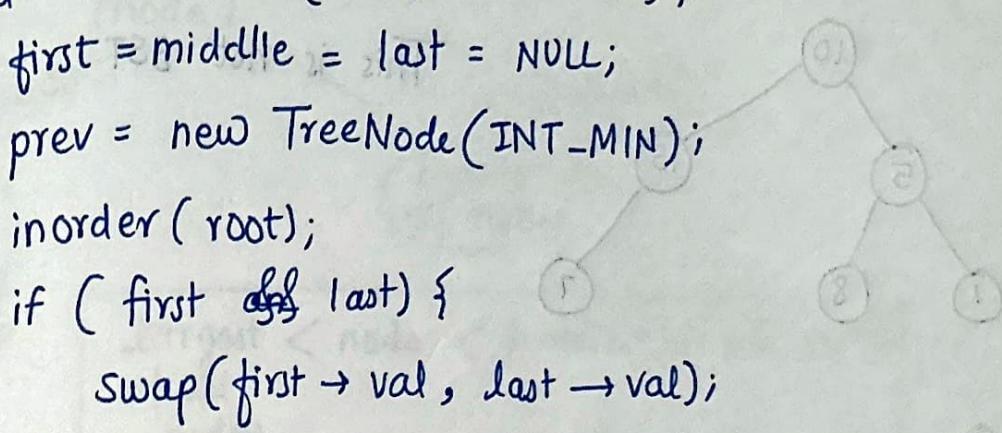
}

(1) O = O.T  
(2) O = O.Z

```

public:
    void recoverTree(TreeNode* root) {
        first = middle = last = NULL;
        prev = new TreeNode(INT_MIN);
        inorder(root);
        if (first && last) {
            swap(first->val, last->val);
        } else if (first && middle) {
            swap(first->val, middle->val);
        }
    }
};

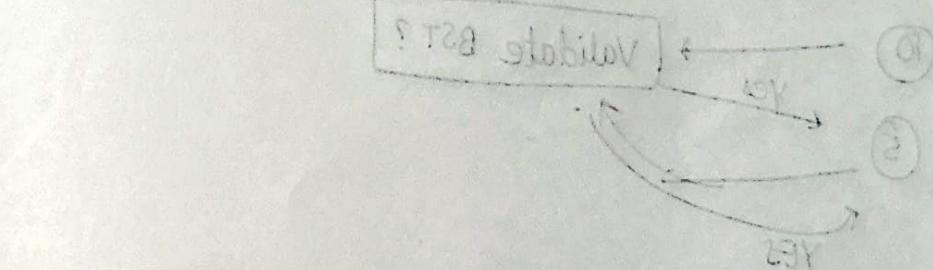
```



```

    } else if (first && middle) {
        swap(first->val, middle->val);
    }
}

```

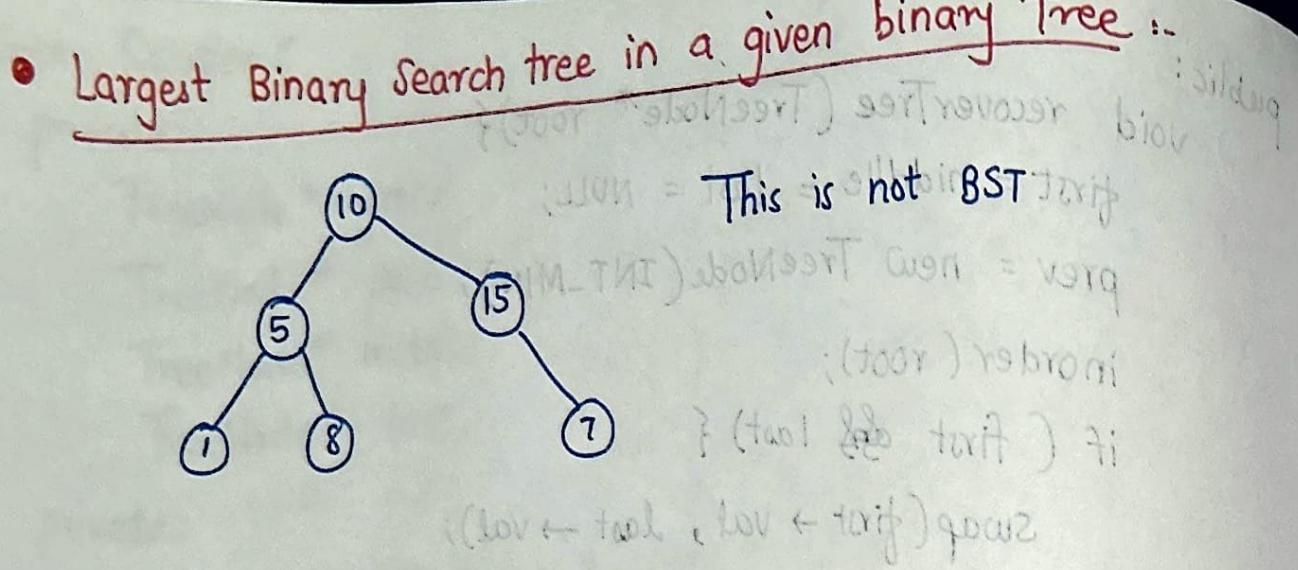


(pre) lernwerk ~~aus~~ nicht ~~aus~~ ist

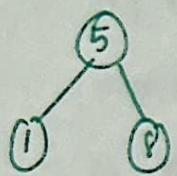
$$(n)O * (n)O = O(n^2)$$

↑                      ↑  
more of problem      solution  
space

$(\log n)O = O(n \log n)$



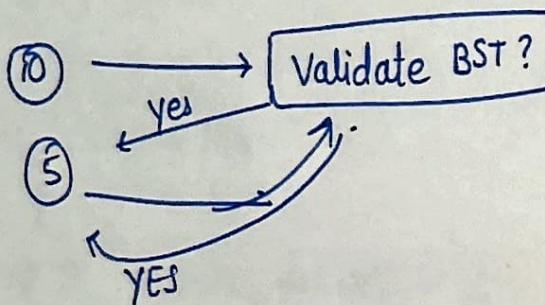
$\textcircled{7} \Rightarrow \text{BST}$



$\Rightarrow \text{BST}$

Brute force :-

Ref: Validate BST



If YES then ~~any~~ traversal (any)

$$T.C = O(N) * O(N)$$

$\uparrow$   
Validate  
a BST

$\uparrow$   
reading to every  
Node

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

Optimal :-

