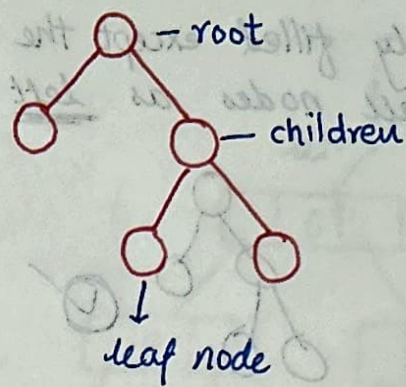
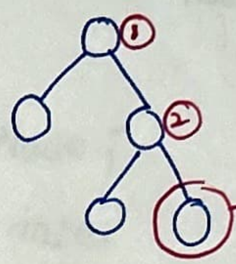
- Introduction to trees :-




Ancestors:-



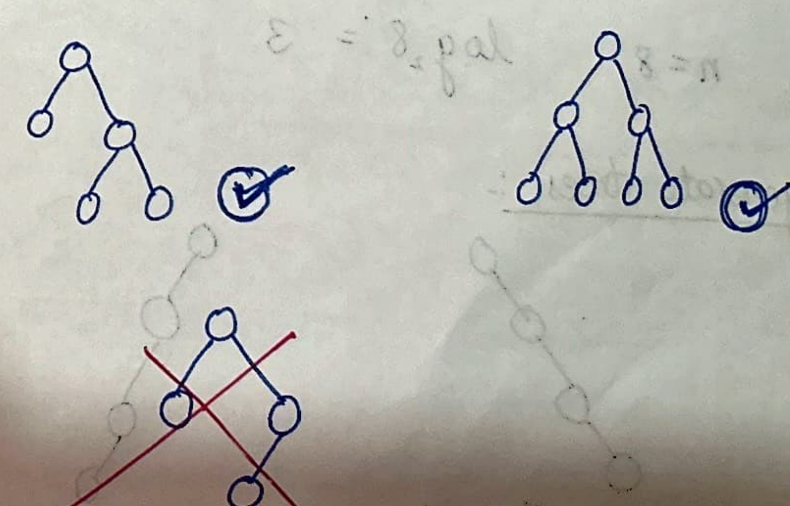
for this node ① & ② are ancestors

# Types of binary trees :-

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

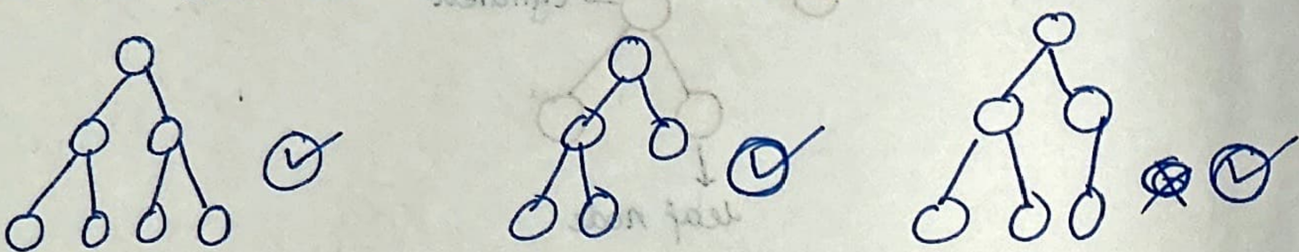
① Full binary tree :-

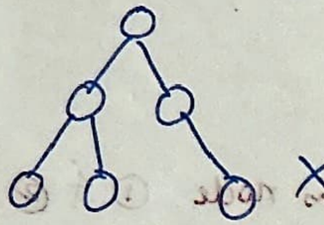
Either have zero or two childrens.



## ② Complete binary tree :-

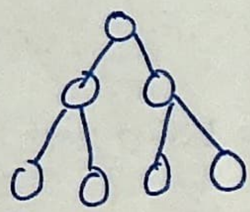
a) All levels are completely filled except the last level.
b) The last level has all nodes as left as possible.





## ③ Perfect binary tree :-

All leaf nodes are at same level

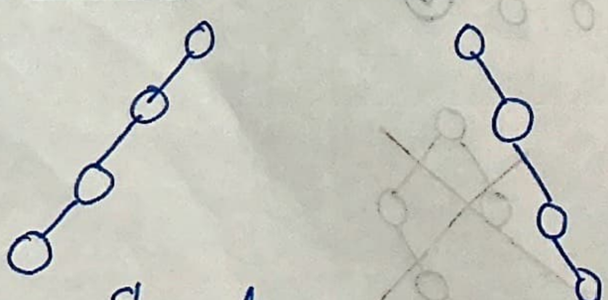


## ④ Balanced binary tree :-

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

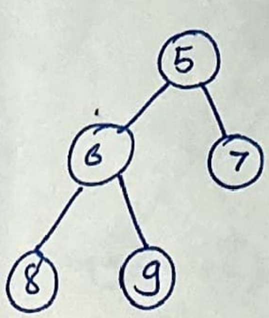
$n = 8 \quad \log_2 8 = 3$

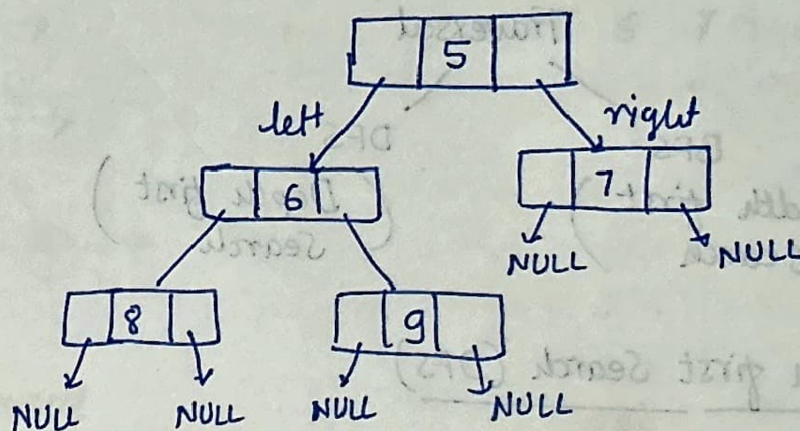
## ⑤ Degenerate trees :-



Skewed

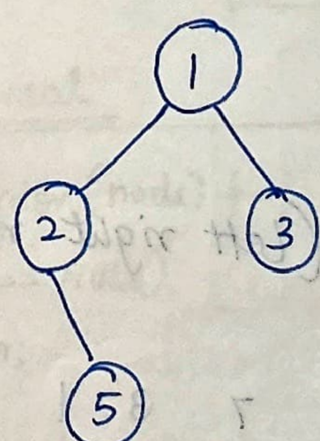
- ## Binary tree representation in C++ :-



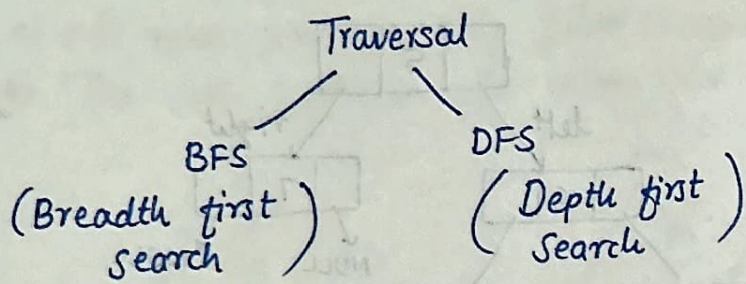



```
struct Node {
  int data;
  struct Node* left;
  struct Node* right;
  Node (int val){
    data = val
    left = NULL;
    right = NULL;
  }
}
```

```
int main() {
  struct Node* root = new Node(1)

  root → left = new Node (2);
  root → right = new Node (3);
  root → left → right = new Node(5);
}
```

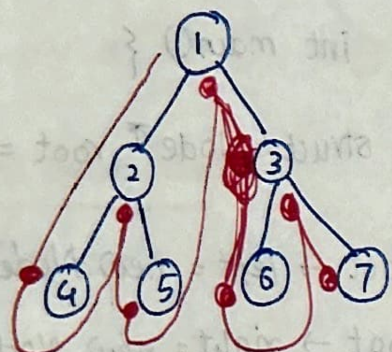


- Traversal Techniques




① Depth first Search (DFS)

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




4 2 5 1 6 3 7

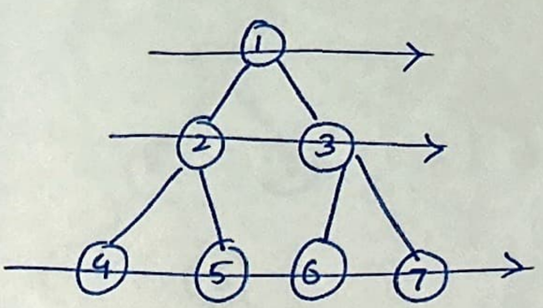
② preorder traversal (root left right)

1 2 4 5 3 6 7

③ post-order traversal (left right root)

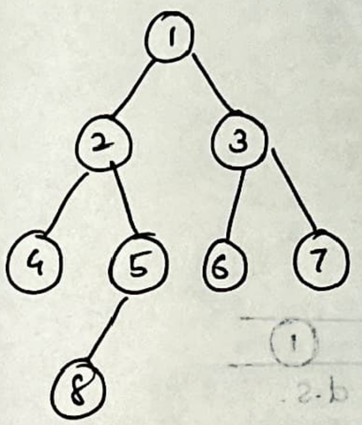
4 5 2 6 7 3 1

## ② Breadth First Search




1 2 3 4 5 6 7

## ① Pre-Order Traversal




root

```
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);
}
```

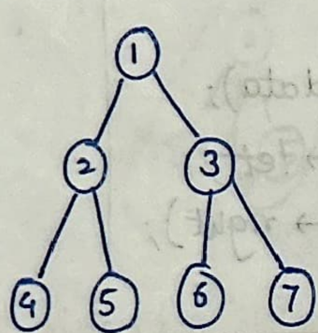
T.C = O(N)

S.C = O(N)

## ③ Post Order Traversal

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

# level-order traversal :-



① 
queue d.s.
(initially having root)

Start

① → Take out ① Now check left side exists for this 1

yes! put in queue

same for right side

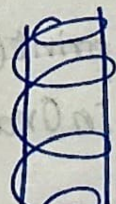
| |
|---|
| 3 |
| 2 |
| ~~1~~ |

Q

Now do same for 2 4 3

for ②

| |
|---|
| 7 |
| 6 |
| 5 |
| 4 |

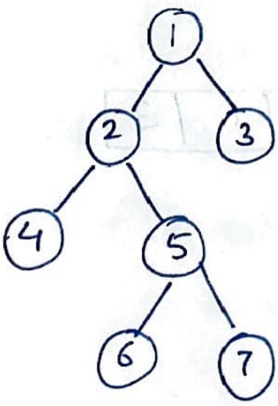


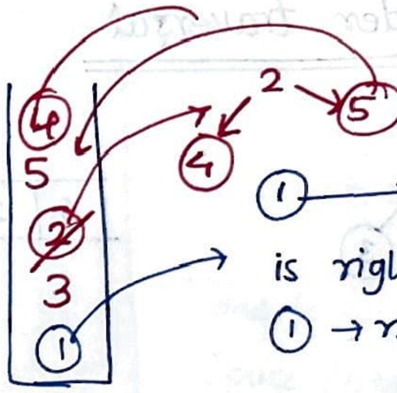
whenever empty → Completed !!

| |
|---|
| [4, 5, 6, 7] |
| [2, 3] |
| [1] |

vector

# Pre-Order traversal (Iterative)





stack

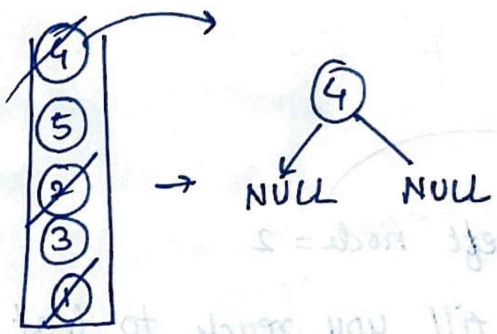
array

① → push [1, 2, 4]

is right present

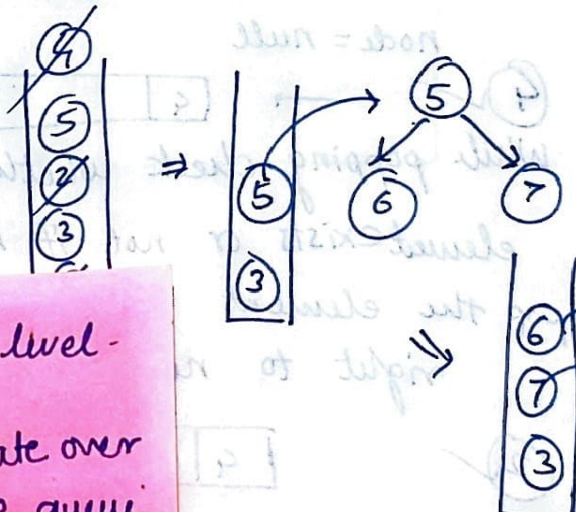
① → right ⇒ 3 ✓
push → stack

① → left ⇒ present → push in stack



→ pop this and push into array

[1, 2, 4]



⇒ push 5 in ans.

[1, 2, 4, 5]

, 6, 7, 3]

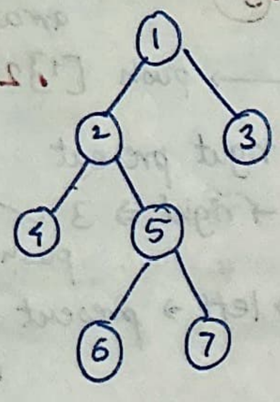
While doing the level-order traversal
→ we have to iterate over the size of the queue and push left and then right

---

But in preorder iterative traversal,
Use stack and push the <u>RIGHT</u> node first and then left node.

```
vector<int> ans;
Stack<TreeNode*> st;
 st.push(root)
while(!st.empty()){
    root = st.top();
    st.pop();
    ans.pb(root → val);
    if (root → right) → push into st.
    if (root → left) → push into st,
}
return ans;
```

# Iterative Inorder traversal



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

node = 1

4
2
1
Stack

take left node = 2

same till you reach to leaf node

node = null

⇒ 4
2
1
Stack

4 → | 4 |

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

2 → | 4 | 2 |

node = 5

6 → 6
5
1

| 4 | 2 | 6 | 5 |

null

null

node = 7

7
1

| 4 | 2 | 6 | 5 | 7 | 1 | 3 |

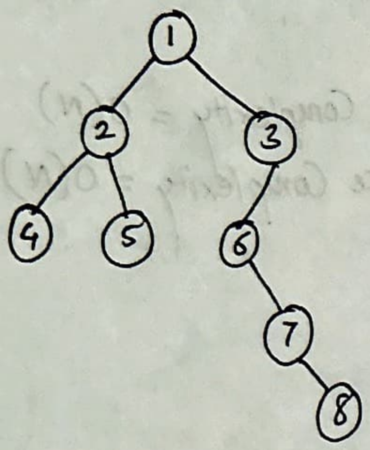
like this.

```
stack <TreeNode*> st;
vector <int> result;
TreeNode* node = root;
while(true) {
    if(node!=NULL) {
        st.push(node);
        node = node -> left;
    }else {
        if (st.empty()) break;
        node = st.top();
        st.pop(); result.pb(node->val);
        node = node -> right;
    }
}
return result;
```

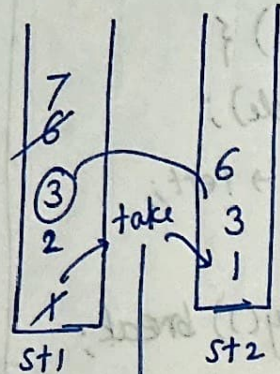
Time Complexity = O(N)
Space Complexity = O(N)

# • Iterative post-order traversal

## ① Using Two-stacks



**Left - Right - Root**

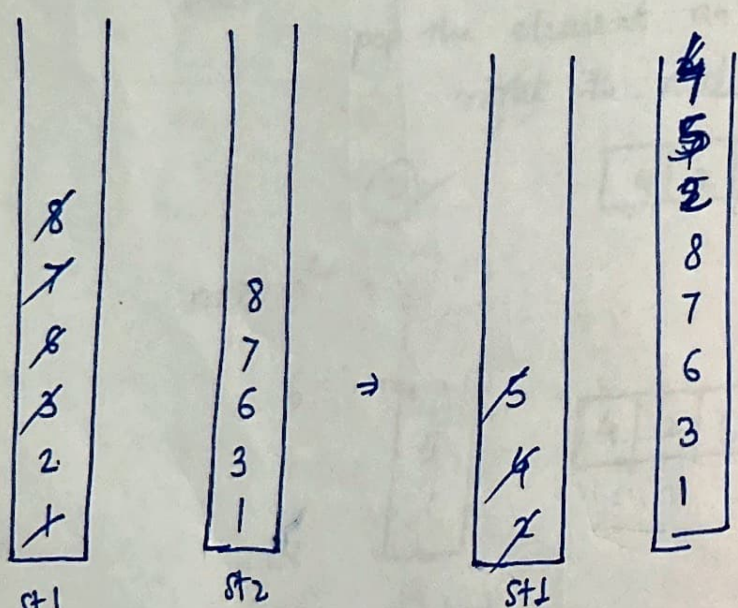



Check if '1' has left → yes push into 'st1'
Check if '1' has right → ———————

⇒ 3 → left → 6 → push in St1

⇒ 6 → right → 7 → push in St1

Perform this

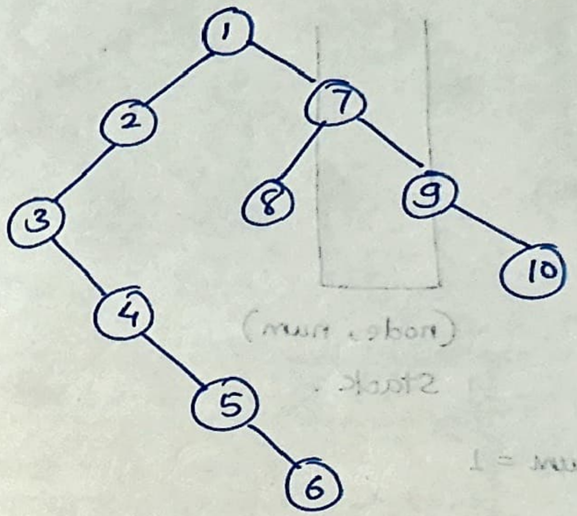



```
vector <int> postorder;
 if(root == NULL) return postorder;
Stack <Node*> St1, St2;
 St1.push (root);
 while (!St1.empty()){
     root = St1.top();
      St1.pop();
      St2.push (root);
   if(root → left)
        St1.push(root→left);
   if(root→ right)
        St1.push (root→right);
  while (!St2.empty())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

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

This is post order

② Using One stack



Go left - left - left → until you reach NULL

then take right

↓

AGAIN Go left - left - left

curr = 1 ← root

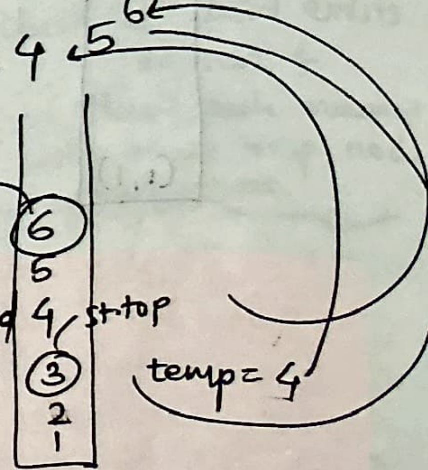
```
While (curr != NULL || !st.isEmpty()) {
    if (curr != NULL) {
        st.push(curr);
        curr = curr → left;
    }
    else {
        temp = st.top() → right;
        if (temp == NULL) {
            temp = st.top();
            st.pop();
            postOrder.push_back(temp → val);
            While (!st.isEmpty() && temp == st.top() → right) {
                temp = st.top();
                st.pop();
                postOrder.push_back(temp → val);
            }
        }
        else
            curr = temp;
    }
}
```

curr = ~~1~~ ~~2~~ ~~3~~ null 4 5 6

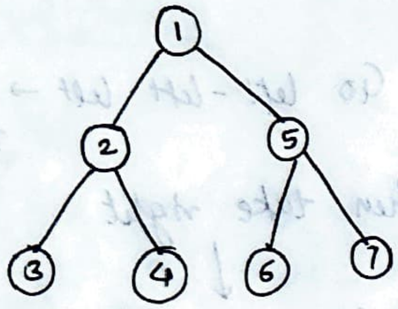
pop

[p.o] = [6, 5]

like this the intution worked



# • PreOrder | InOrder | PostOrder in One traversal



(node, num)

Stack

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

if (num == 3)

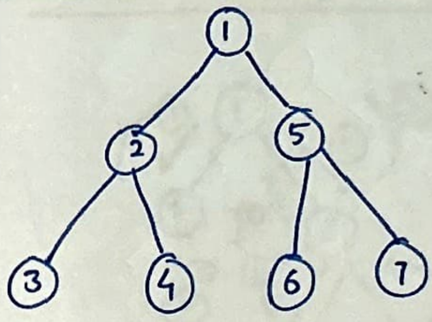
↳ POST-ORDER

While ~~checking~~ 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 <>

## DRYRUN



PreOrder : 1 2 3 4 5 6 7

InOrder : 3 2 4 1 6 5 7

PostOrder : 3 4 2 1 6 ~~7~~ 5

⇒ (1,1)

(node, num)

num = 1 → num ++

reinsert to stack

⇓

~~(5,1)~~ → left

(2,~~1~~)2

(1,~~1~~)2

⇐

(3,~~1~~)2

(2,2)

(1,2)

! pop-it → check num

increament it

check if child exists

or not &

then push num=1

with their resp. node

~~add it to~~

⇓

(3,2)

(2,2)

(1,2)

num = 2

INORER

num++ = 3

Check right

⇓

NO

(3,3)

(2,2)

(1,2)

num = 3

POST ORER

pop the element

& node

⇒

(4,1)

(2,~~2~~)3

(1,2)

INORER : 2

↓ num++ ⇒ 3

check right

↓

(4,1)

⇒

(4,~~1~~)2 → 1 ++

(2,3)

(1,2)

⇒ Now 2

INORER

↓

++

⇒ (4,3)

(4,3)

(2,2)

(1,2)

post orde

likewise

you have

to

perform

(2,3) → post → pop

(1,2) →

⇒

(1,3) → ~~post~~ post → pop

Inorder

⇒

(5,1)

(1,3)

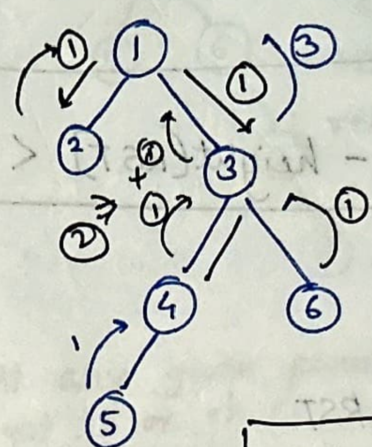
⇒ .....

Time Complexity = O(N)

Space Complexity = O(N)

```
~~Stack<TreeNode*>~~
Stack < pair <TreeNode* , int>> st;
st.push({root, 1});
vector <int> pre;
vector <int> in;
vector <int> post;
if(root == NULL) return;
while( !st.empty()) {
    auto it = st.top();   // ~~TreeNode* top~~
    st.pop();
    ~~if( top~~
    if(it.second ==1){
        pre.push_back(it.first->val);
        it.second ++;
        st.push(it)
        if(it.first->left != NULL){
            st.push({it.first->left, 1});
        }
    }
    else if(it.second ==2){
        in.push_back(it.first->val);
        it.second++;
        st.push(it):
        if(it.first->right != NULL){
            st.push({it.first->right, 1});

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

    }
}
```

# Maximum Depth of Binary Tree



height = 4

Two approaches

① Recursive → Auxiliary Space O(N)

② Iterative
Queue D.S.
S.C. ⇒ O(N)

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

At ④ → 1 + max(1, 0)
⇒ 1+1 = 2 → for 4 → height = 2

for ⑥ ⇒ ① + max(0,0) → 1

for ③ ⇒ 1 + max(left, right)
1 + max(2, 1) ⇒ 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);
```

## • Check for balanced binary Tree :-

> **Balanced binary Tree :-**
>
> height (LST) - height (RST) <= 1

### Naive solution :-

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

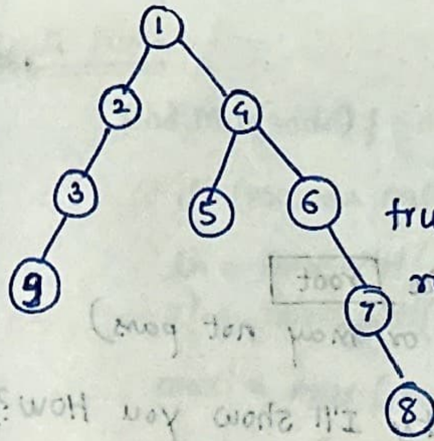
**pseudo code**

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

    Lh = findHLeft(node->left);
    Rh = findRight(node->right);

    if (abs(Lh - Rh) > 1) return false;

    Bool left = check(node->left);
    Bool right = check(node->right);

    if (!left || !right) false;

    return true;

}
```

T.C.

$O(n) \times O(n) \Rightarrow O(n^2)$

we have to check for every node.



## Optimal code

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

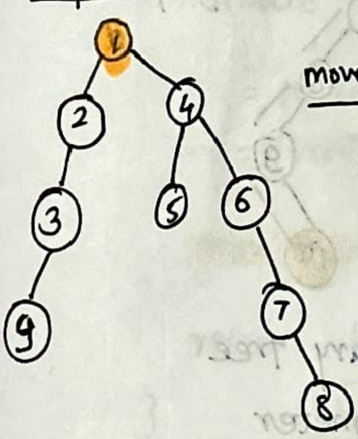
```
int check (node){
  if(node == null) return 0;

  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
```

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

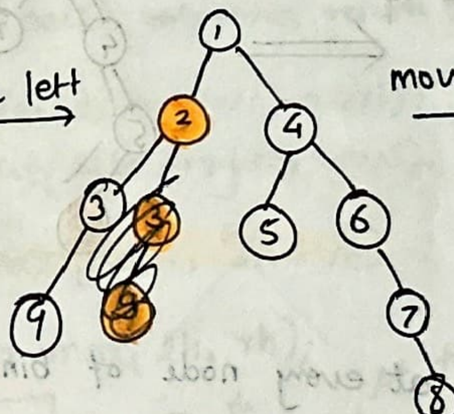
return -1 if absolute diff exceeds 1

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

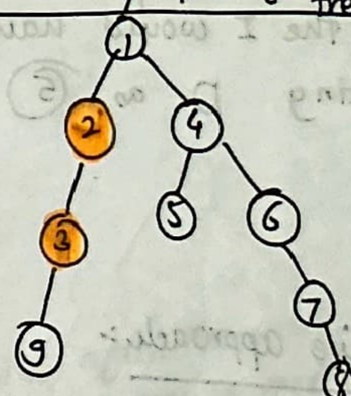
Dry run :-



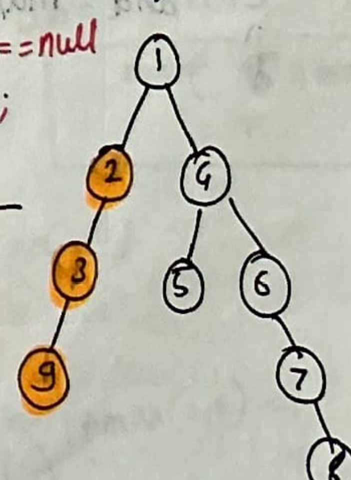
move left →



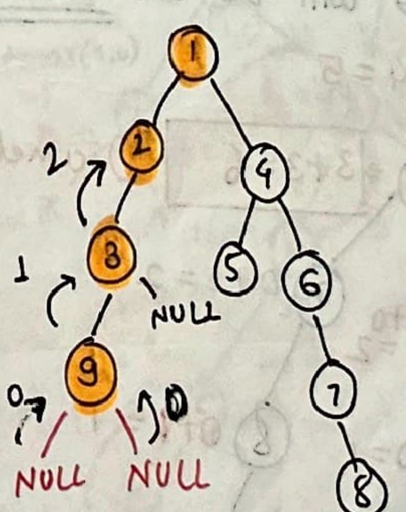
move left →



↓ move left



move left here, node == null return 0;
⇒ lh = 0
⇒ rh = 0
return max(0,0) + 1
= 1




At ③
from left = 1
from right = 0

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

⇓

At ② → the height lh = 2, rh = 0 → if(abs(lh-rh)>1) return -1

This condition is executed!

Tree is not a balanced binary tree

# Diameter of binary tree :

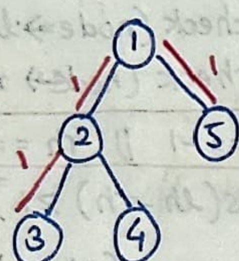
**Diameter :-**

⇒ Longest path between 2 nodes.

⇒ Path does not need to pass via $\boxed{\text{root}}$
optional (It can pass or may not pass)

⇓

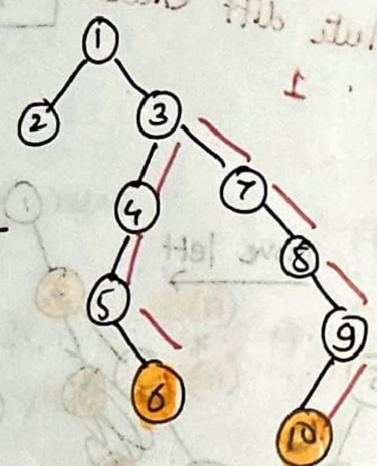
This is tricky I'll show you How?!



⇒ 3

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

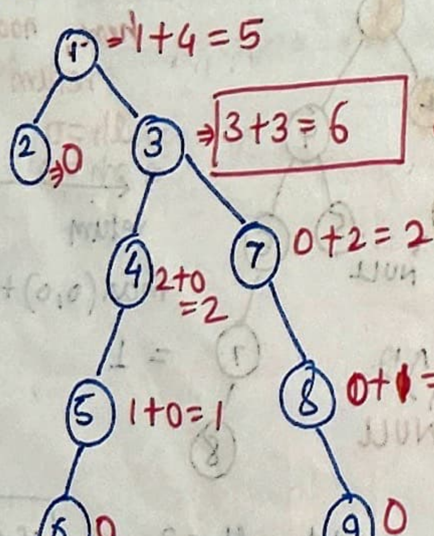
BUT




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

## Naive approach :-

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




Diameter = 6

## Brute Force:-

```
find Max(node) {
    if (root == null) return 0;
    lh = FindLeftH(node->left);
    rh = findRightH(node->right);
    maxi = max(maxi, lh + rh);
    findMax(node -> left);
    find Max (node -> right);
}
```

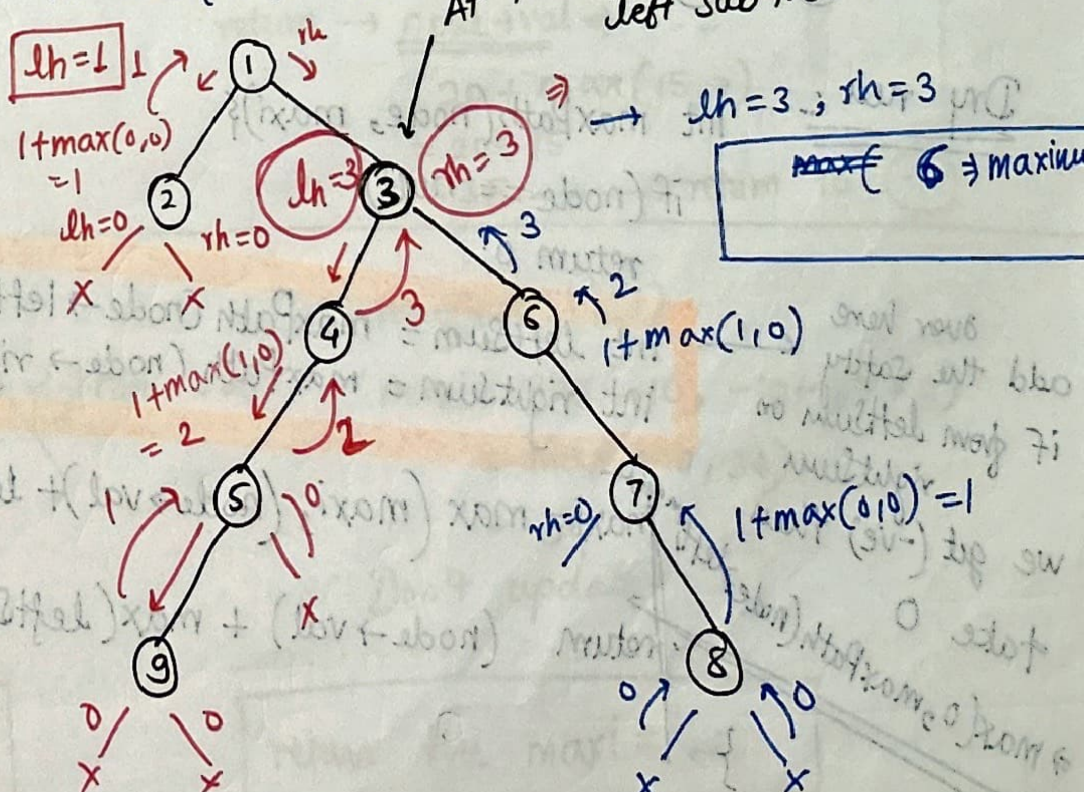
→ Go to left and right and compute the max height EVERYTIME!!

T.C = O (N²)

## Optimal ⇒

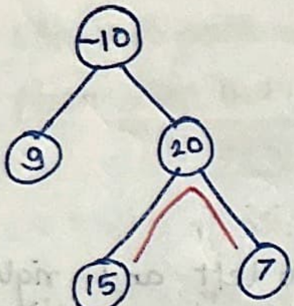
```
int findMax(node, maxi) {
    if (node == null) return 0;
    lh = findMax(node->left, maxi);
    rh = find Max(node->right, maxi);
    maxi = max (maxi, lh + rh);
    return 1+ max(lh, rh);
}
```

At this point after traversal left sub tree ⇒ maxi = 3



- **Maximum Path Sum :-**

Any node, appears once while traversing through tree



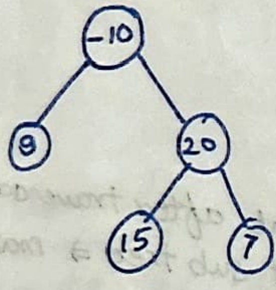
$15 + 20 + 7 = 42$ (Maximum path sum)

**Brute force :-**

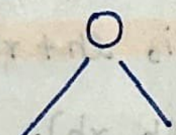
find the path sum ~~of~~ for every pair of nodes which can

cost $> O(n^2)$

prerequiste ⇒ Maximum Height (BST)
Width of BST



for every node, we need to compute the Left Sum and right sum



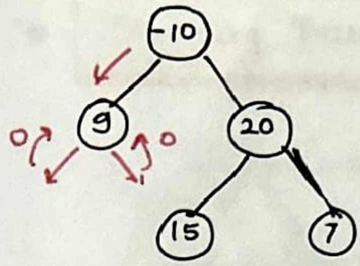
Formula ⇒ (node → val) + (rightSum + left

**Dry run :-**

```
int maxPath(node, maxi){
    if(node == null)
        return 0;
    int leftSum = maxPath(node->left), maxi );
    int rightSum = maxPath(node->right), maxi);
    maxi = max(maxi, (node->val) + leftSum + rightSum);
    return (node->val) + max(leftSum, rightSum);
}
```

over here add the safety if from leftSum or rightSum we get (-ve) then take 0

⇒ max(0, maxPath(node→left), maxi)



At 9

maxi = max(0, 9 + 0 + 0)
= max(0, 9)

maxi = 9

return → 9 + max(0, 0) = 9 → return to -10

---

At -10 → leftSum = 9

right, 20 → go left, 15

0 0

maxi = 9 (old)

maxi = max(9, 15 + 0 + 0)
= max(9, 15)

maxi = 15 → returned 15 to 20

return 15 + max(0, 0)

---

20 → right → 7

0 0

maxi = 15 (old)

maxi = max(15, 7 + 0 + 0)
= max(15, 7)
= 15 → ~~return to 20~~

return → 7 + max(0, 0) → return to 20

---

leftS = 15 (20) rightSum = 7

sum)

maxi = 15

maxi = max(15, 20 + 15 + 7)
= max(15, 42)

maxi = 42

return → $\underline{\text{node→val}}$ + ---

20 + max(15, 7)
= 20 + 15
= 35 → return to -10

---

on -10

leftSum = 9 rightSum = 35

maxi = 35 (old)

maxi = max(-10, -10 + 35 + 9)
= max(-10, 34)
= 34

Don't update

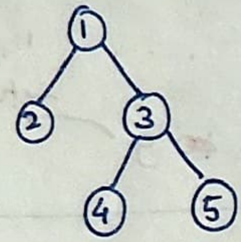
T.C = O(N)

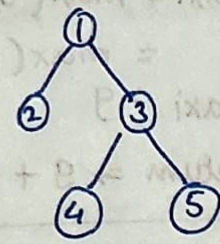
S.C = O(N)

return the maxi = 42

from main function

- Same Tree or not :-

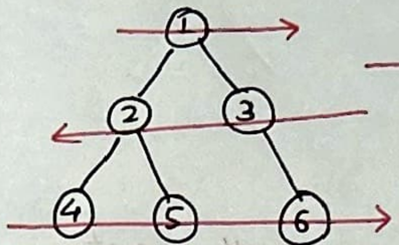




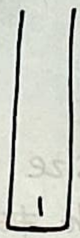
preOrder = Traverse in both the ~~preOrder~~ trees at the same time

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

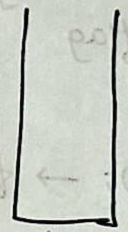
# Zig-Zag Traversal




123456 → This is zig-zag pattern
⇄
⇄
:




queue

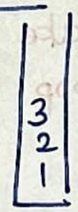


ds
vector<vector<>>

direction = 0

| | |
|---|---|
| "0" → | LEFT ⇒ RIGHT |
| "1" → | RIGHT ⇒ LEFT |

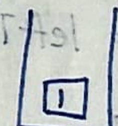
Dry Run :-



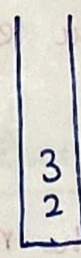

Q

check 1 → left ⇒ 2 ⟶ push into Q
1 → right ⇒ 3 ⟶ push into Q

Once done
remove 1 from Q and put it into ds

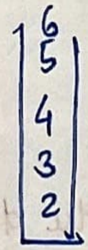



Now reverse the direction → 1

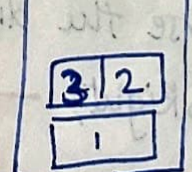



2 ⟹ 2 → left ⇒ 4 ⟶ push → Q
2 → right ⇒ 5 ⟶ push → Q

3 ⇒ 3 → left ⇒ NULL
3 → Right ⇒ 6 ⟶ push → Q




Once done ⟶ from right to left

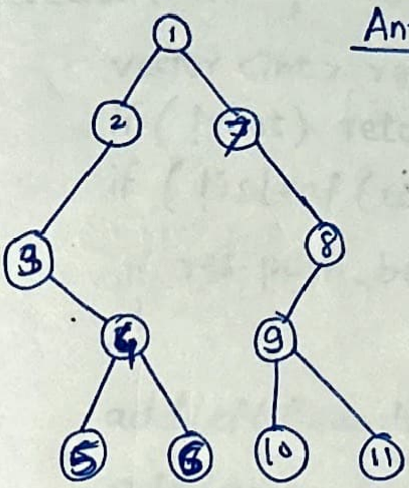



Likewise

```
vector<vector<int>> zigZagLevelOrder(TreeNode* root){
    vector<vector<int>> result;
    if(root == NULL){
        return result;
    }

    queue<TreeNode*> nodesQueue;
    nodesQueue.push(root);          → Initially push the root
    bool leftToRight = true;        → flag

    while(!nodesQueue.empty()){
        int size = nodesQueue.size();   → find the size  E.g. [3 / 2]
                                          All the ~~levels~~ nodes
                                          on which I have
                                          to travel
        vector<int> row(size);

        for(int i=0; i<size; i++){
            TreeNode* node = nodesQueue.front();   → Take front
            nodesQueue.pop();                        & pop

            //find position to fills node value
            int index = leftToRight ? i : (size-i-1);   → Check
                                                          direction
            row[index] = node->val     → push basis on direction
            if(node->left){
                nodesQueue.push(node->left);    ] push the left node
            }
            if(node->right){
                nodesQueue.push(node->right);   ] push the right node
            }
        }
        // after this level, reverse the direction
        leftToRight = !leftToRight;     → Reverse the diretion
        result.push_back(row);          → put the row in result
    }
    return result;
}
```

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

- **Boundry Traversal :-**




Antidockwise

~~1 2 4 5 6 7~~

> Left Boundry (Excluding leafes)
> Leafes (prefer Inorder)
> Right Boundry in the reverse + Exclude leaf nodes

root → | 1 | ds

Start with left boundry
node → left → put it into ds ⇒ | 4 3 2 1 |
check again node → left
If there doesn't exist left → go to right
then check the node → left ⇒ 5 but this is leaf node so SKIP

for leaf nodes (INORDER TRAVERSAL) ⇒ Root left Right ⇒ 5 6 10 11

| 11 10 6 5 4 3 2 1 |

Right Boundry In Reverse

| 9 8 7 | → go to right 8 ⇒ does right exist ??
NO
If right doesn't exist Then move to LEFT
Again check for right node

~~vector~~ st.

Take from last

| 7 8 9 11 10 6 5 4 3 2 1 | → This is the boundry traversal.

```
void addLeftBoundry( 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 addRightBoundry (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);}
}
```

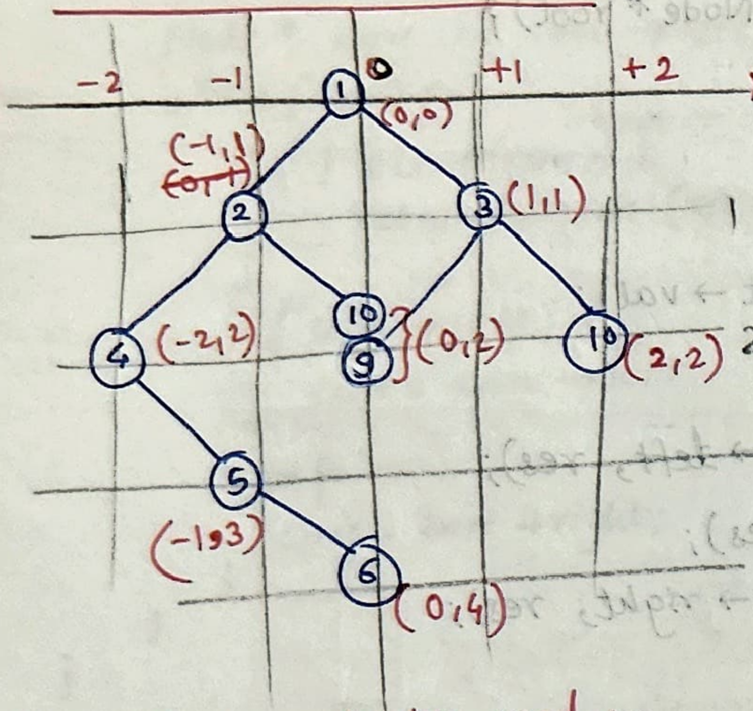
Inorder traversal
Root
Left
Right

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

    vector <int> res;
    if (!root) return res;
    if (!isLeaf (root)) {
        res.push_back (root → val);
    }

    addLeftBoundry (root → left, res);
    addLeaves (root, res);
    addRightBoundry (root → right, res);

    return res;
}
```

$$S.C. = \underbrace{O(H)}_{\text{height of tree}} + \underbrace{O(H)}_{\text{height of tree}} + \underbrace{O(N)}_{\text{Inorder traversal}} \approx O(N)$$

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

- Vertical Order traversal




x-axis [4]

[2, 5]

[1, 9, 10, 6] If overlapping, consider smaller one first

[3]

[10]

Data structures to be used :-

Queue

Q{node, vertical, level}

multi nodes

map < int, ~~vector~~ map < int, multiset<int>>>

vertical ↑ (represent vertical), level ↑

for every vertical (~~level~~) ⟶ why I'm taking map??

so if you see at (0,2) ⟶ 10, 9

At every level there can be multiple nodes I need them into sorted order that is why $\boxed{\text{multiset}}$

Why multiset ?

Because at each level duplicate node values can present.

node = 1

$\boxed{(1,0,0)}$

q (node, vertical, level)

Insert into ds $\boxed{0 \longrightarrow \{0 \rightarrow \{1\}\}}$ ds

map <int, map<>>

This is the form

⇓

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

$\boxed{\cancel{(1,0,0)}, (2,-1,1),}$

q

⇓

move ~~left~~ right
~~node~~
node = 3

$\boxed{\cancel{(2,-1,1)}, (3,1,1)}$

q

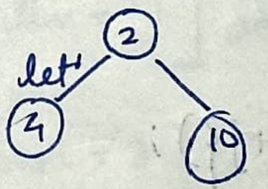
vertical ++;
level ++;

node = 2
vertical = -1
level = 1

→ store into ds

ds

| | | |
|---|---|---|
| -1 → | 1 → | {2} |
| 0 → | 0 → | {1} |

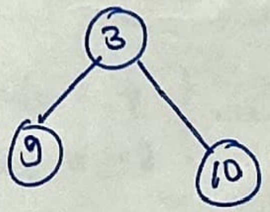



$\boxed{(3,\cancel{1},1), (4,-2,2), (10,0,2)}$

q

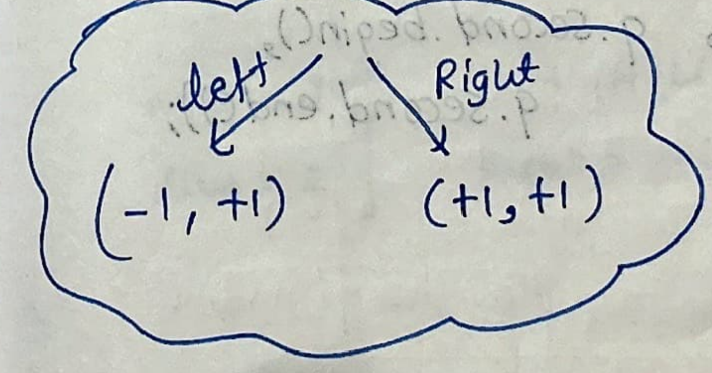
| | | |
|---|---|---|
| 1 → | 1 → | {3} |
| -1 → | 1 → | {2} |
| 0 → | 0 → | {1} |

ds




$\boxed{(4,-2,2), (10,0,2), (9,0,2), (10,2,2)}$

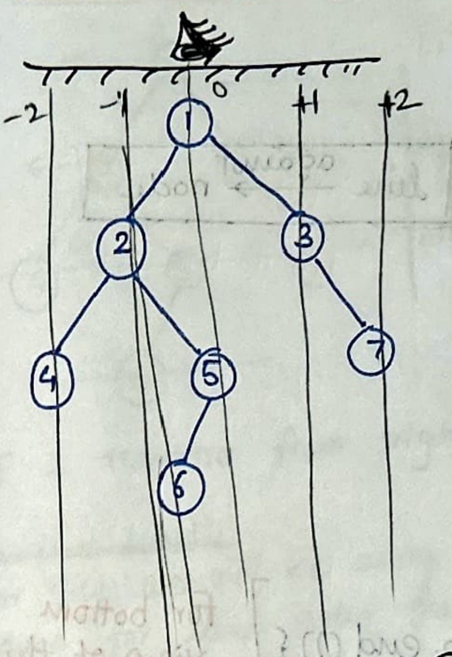
like this



Level order traversal used :

```
vector<vector<int>> verticalTraversal( TreeNode* root) {
  map<int, map<int, multiset<int>>> nodes;
                          node
  queue< pair< TreeNode*, pair<int, int>>> todo;
                                  (vertical, level)
   todo.push({root, {0,0}});
  while (!todo.empty()) {
     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.inser(col.end(), q.second.begin(),
                               q.second.end());
      }
      ans.push_back(col);
  }
  return ans;
}
```

- <u>Top view of Binary tree</u>




4 2 1 3 7

Q: ~~(1,0)~~, (2,-1), (3,1), (4,-2), (5,0), (7,2)

(node, line) map (line, node)

| |
|---|
| 0 → 1 |
| -1 → 2 |
| 1 → 3 |
| -2 → 4 |
| 2 → 7 |

map which is sorted always

| |
|---|
| -2 ⇒ 4 |
| -1 ⇒ 2 |
| 0 ⇒ 1 |
| 1 ⇒ 3 |
| 2 ⇒ 7 |

node = 1
line = 0 ⟶ left ② → present on the -1 line
and on right ③ → +1 to line

node = 2
line = -1 ] → left ④ → do line-- → -1 -1 = -2
and right ⑤ → do line++ → -1 +1 = 0

node = 3
line = 1 ] → left → NULL
right ⟶ ⑦ → do line++ → 1+1 = 2

node = 4
line = -2 ] → left → NULL
right → NULL

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

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

```
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;

}
```

$\boxed{\text{line} \xrightarrow{\text{against}} \text{node}}$

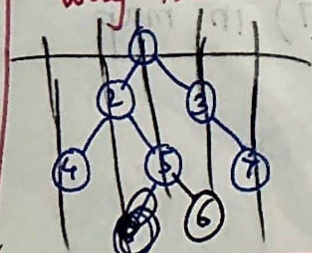
For bottom view of the tree we can remove the if block and override the values.

$\boxed{\begin{array}{l} TC = O(N) \\ SC = O(N) \end{array}}$

**Notes to be discussed**

Why the recursive approach is not used?

why the level-order traversal is used.

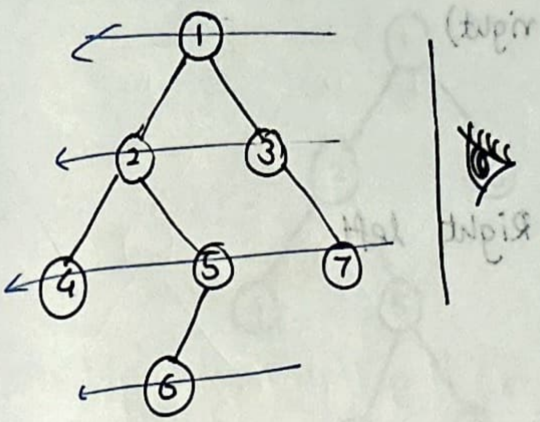



→ If you've used inorder traversal then you would have visited

1 2 4 5 6 → This node is at (+1)

so in map the desired node (i.e. 3) would have not been pushed!!

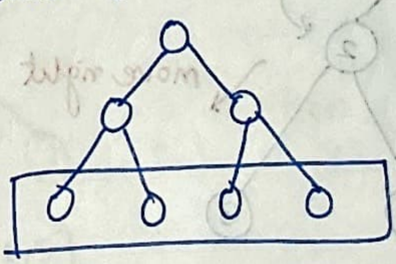
# Right / Left View of Binary Trees :-



⇒ [1 3 7 6]  Right

If I traverse from right to left → 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) → Height of the tree.

preorder → Root, left, right

But for this [Root, right, left]

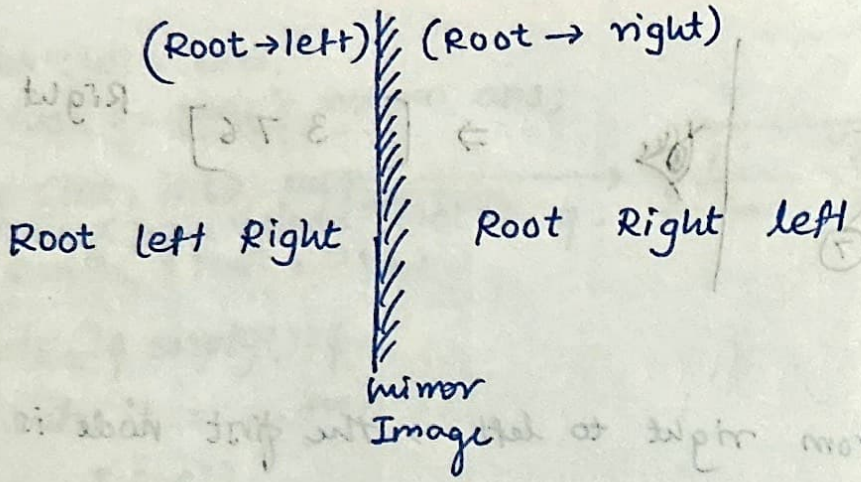
```
root, 0
f(node, level) {
    if (node ==NULL) return;
    if (level == ds.size){
        ds.push-back (node → val);
    f (node → right, level +1);
    f (node → left , level +1);
}
```

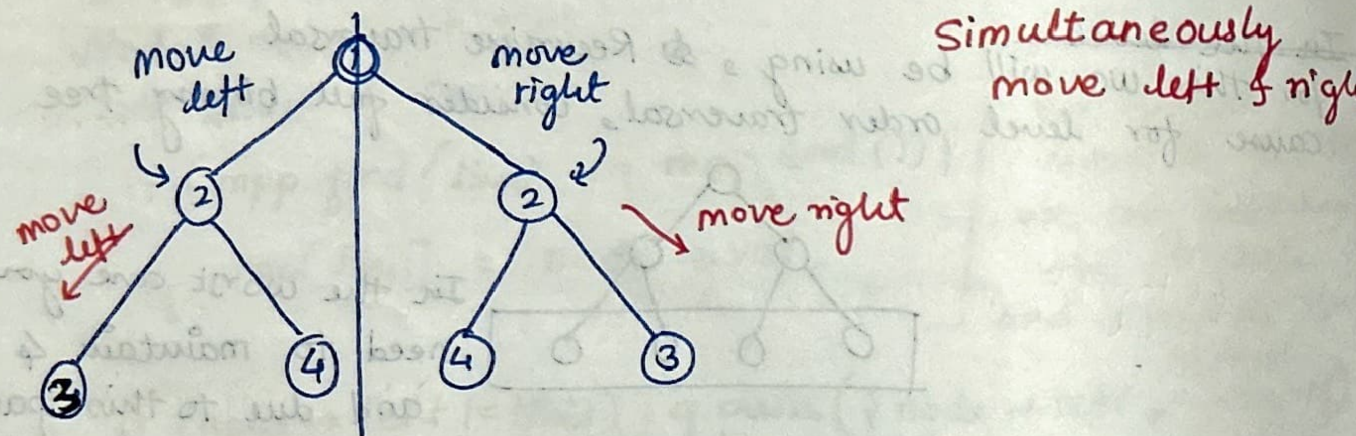
→ Check the ds size at each traversal.

| ds |
|---|
| 6 |
| 7 |
| 3 |
| 1 |

for left view ⇒ Traverse (R L Right)

- Symmetric Binary Tree:-

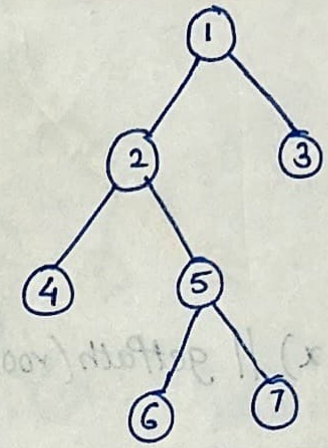



```
bool isSymmetricTree(TreeNode* node){
    return root == NULL || isSymmetricHelper(node→left, node→right);
}

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);

}
```

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

# Print Root to Node path in Binary Tree :-

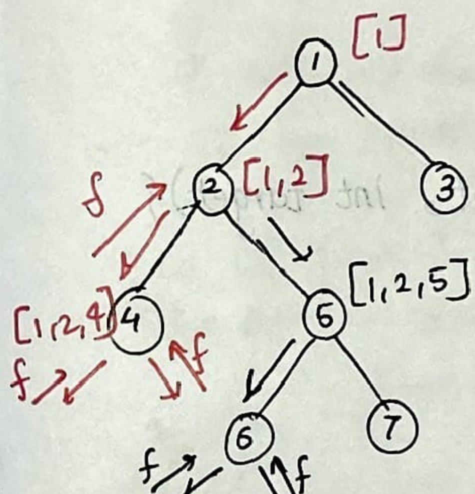


node = 7

path from '1' to '7'

⇒ [1, 2, 5, 7]

Traversal ⇒ Inorder traversal




First move to left & come back

then right and come back

At ② → is 2 == 7 NO! → push [1,2]

move left ④ → is 4 == 7, NO! → push [1,2,4]

move left ⇒ NULL ⇒ Return false

move right ⇒ NULL ⇒ return false

so from node ④ → false is returned hence pop the last element (4) ⇒ [1,2,4] will become [1,2]

move right from ② ⟶ 5 == 7 NO! → push [1,2,5]

move left ⇒ ⑥ ⟶ 6 == 7 NO! → push [1,2,5,6]

move left of ⑥ → NULL → return false

mover right of ⑥ → NULL → return false

⇒ pop the ⑥

Array is [1,2,5]

Now move right of ⑤ ⟶ 7 is 7 == 7 ⇒ True → push [1,2,5,7]

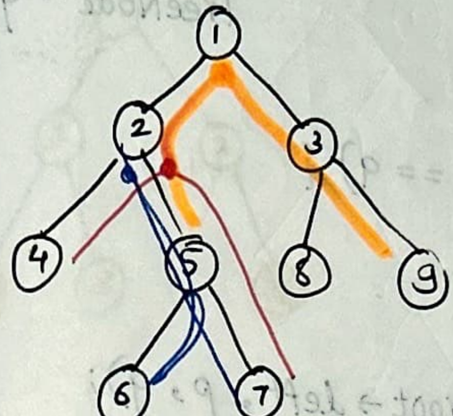
return True.

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;
}

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

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

# Lowest Common Ancestor in Binary Trees :-



lca(4,7) = 2

lca(5,9) = 1

lca(2,6) = 2

node = 4 path (1, 2, 7)

node = 7 Path (1, 2, 5, 7)

matched

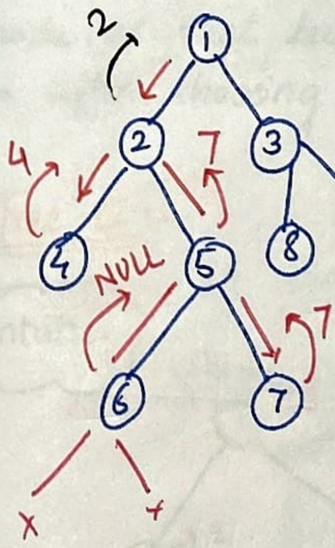
Matched

⇒ 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

lca(4,7)



→ We will recursively traverse

→ go to left ⇒ DFS → and whenever you encouter any of the node (4/7) then you have to return (node ~~val~~) IMPORTANT!

⇒ At 5 from left ↷ NULL

and ↷7 Right

and Now 5 will return 7

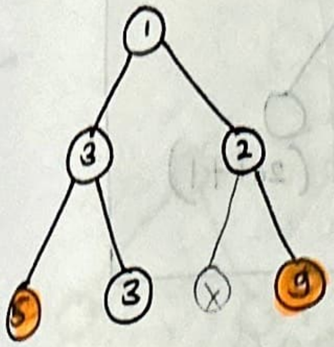
At 2 left ↷4 2 ↷7 Right ⇒ This signifies you have figured out both the nodes

If left != NULL && right != NULL then

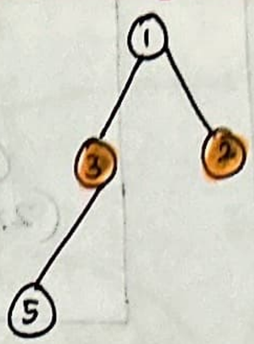
At this point, means whenever you encouter any node whose left & right both have valid values then return the (node ~~val~~) ⇒ And this is LCA.

```
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;

}
```

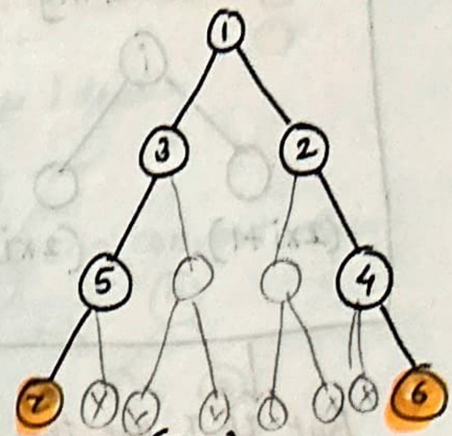
# Maximum width of Binary Tree :-



(I)



(II)



(III)

**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 ~~Tree~~ have the left node then the maximum width of tree ⇒ 4

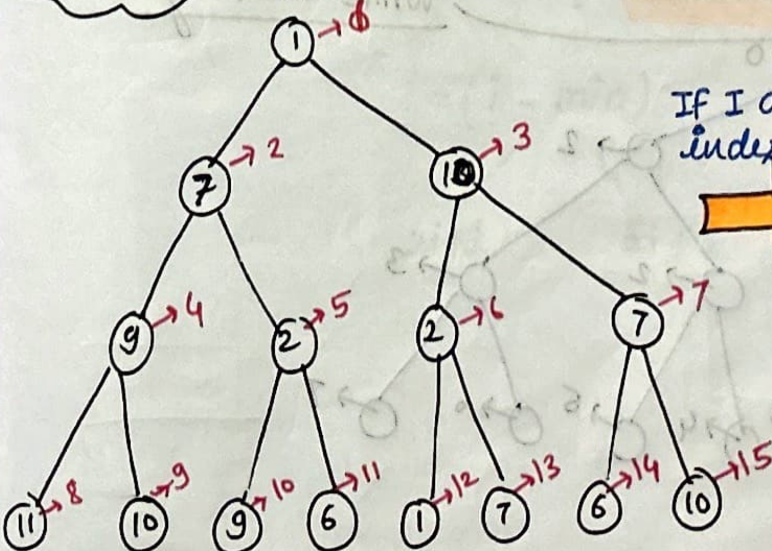
**Tree - II :-**

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

**Tree - III :-**

In third tree ⇒ Maximum width = 8

Intution



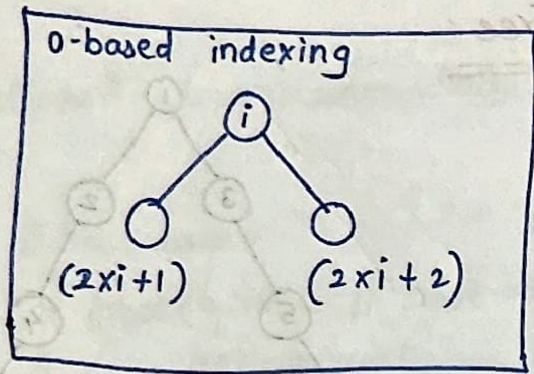
If I assign the index at each level.

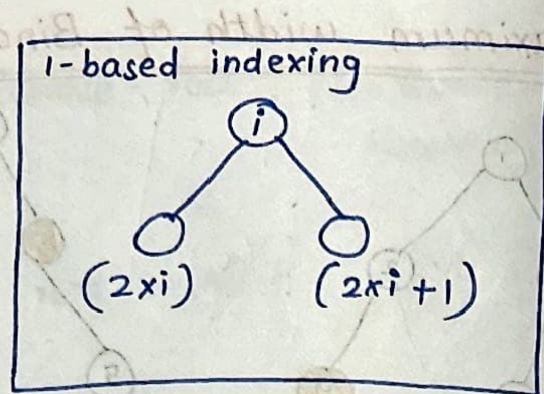
Then at each level I just have to subtract the ( last idx - first idx + 1 )

⇓

I will get the width

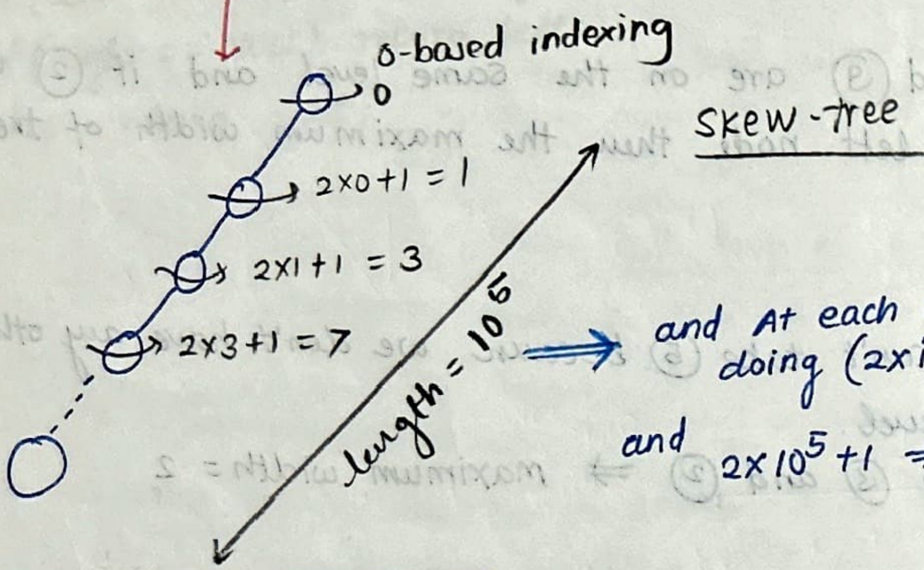
EX : ① 3 - 2 + 1 = 2
② 7 - 4 + 1 = 4
③ 15 - 8 + 1 = 8




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

Why??




and At each step we are doing $(2 \times i + 1)$

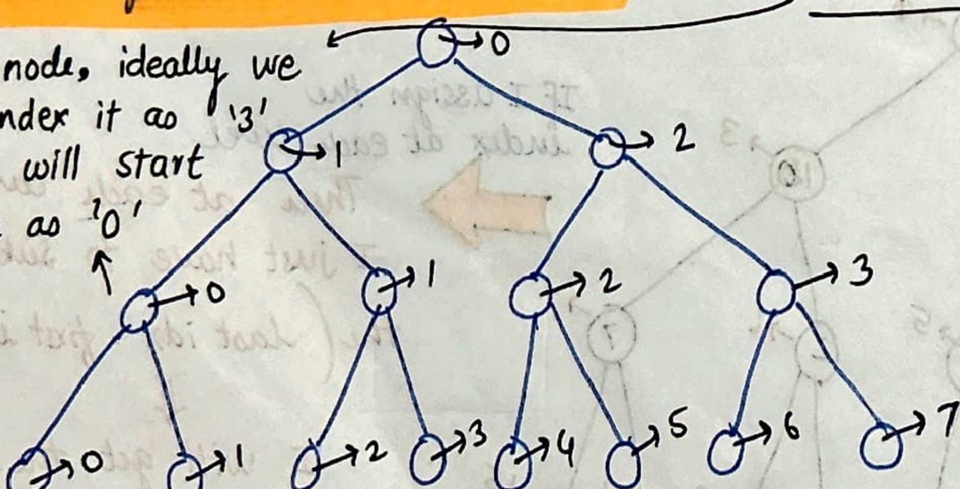
and $2 \times 10^5 + 1 \Rightarrow$ will overflow!!!

Hence this will fail!!

How will you prevent this overflow ⇒

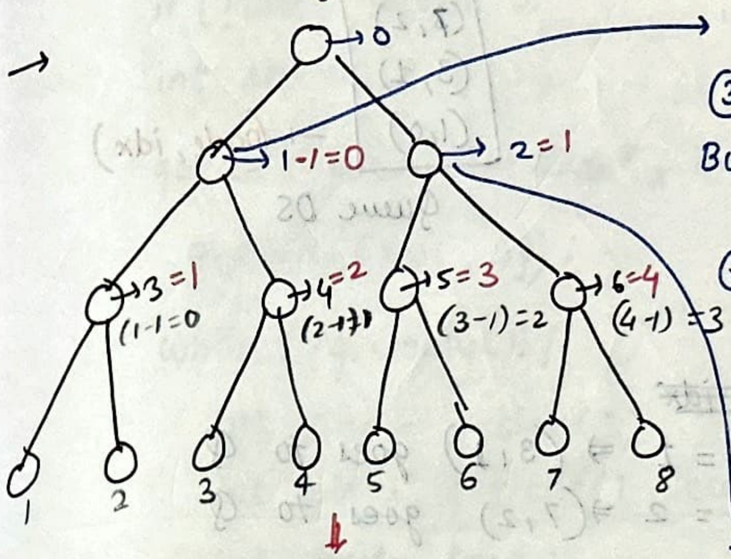
While indexing

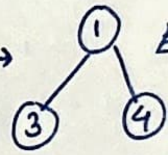
At this node, ideally we should index it as '3' but we will start indexing as '0'




Same here

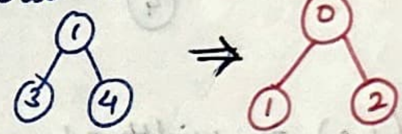
# How will you get those indexes ?




← Ideally

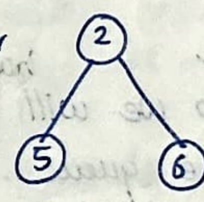
But if I can convert this




by doing 1-1 = 0

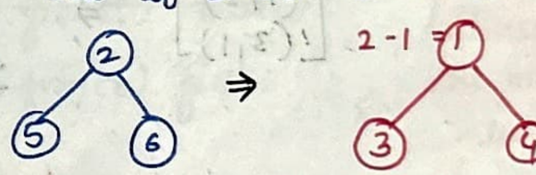
and formula $2 \times i+1 = 1$, $2 \times i+2 = 2$

for




← Ideally

but if I do 2-1 = 1 ⇒ 1




left = $2 \times i+1 = 2 \times 1+1 = 3$

right = $2 \times i+2 = 2 \times 1+2 = 4$

⇓

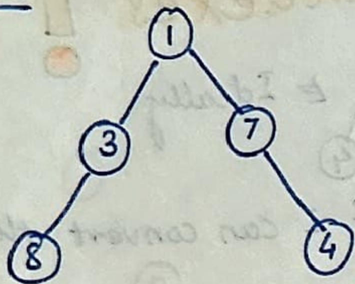
THIS IS HOW IT WILL GET SIMPLIFIED TILL THE DEPTH OF TREE !

This is how it will get prevented from OVERFLOW!!

$$i = (i - min)$$

$2i+1$     $2i+2$

Dry Run



(7, 2)
(3, 1)
(1, 0) → (node, idx)

Queue DS

At (1,0) ⇒ width = 1

Go left ⇒ 3 → ~~2x1+1-2 → idx~~
left index = 2x0+1 = 1 ⇒ (3,1) goes to Q
Right index = 2x0+2 = 2 ⇒ (7,2) goes to Q

(1,0) → is done
After doing this we will do the next iteration on the number of elements in the queue.

[(7,2), (3,1)] ⇒ (3,1) leftmost, (7,2) rightmost

⇒ width = last - first + 1
= 2 - 1 + 1
$\boxed{\text{width} = 2}$

do the index-1 for overflow prevention

for (3, 1) 1-1=0 — left → (8,1), Right → NULL
2x0+1=1

(7, 2) → 2-1 = 1 — left → NULL, Right → (4, 4)
2x1+2=4

push these in Queue

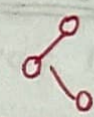
[(4,4), (8,1)] Q →

for (4, 4) — x, x
(8,1) — x, x

W = 4 - 1 + 1
⇒ $\boxed{W = 4}$

```
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 currIndex = q.front().second - min;
```



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

~~TreeNode * node = q.front(). first()~~

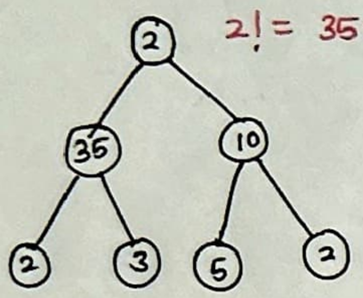
```
            TreeNode * node = q.front(). first;
            q.pop();
            if (i==0) first = currIndex;
            if (i== size -1) last = currIndex;

            if (node ->left) {
                q.push({node -> left, 2*currIndex +1});
            }
            if(node -> right) {
                q.push ({ node -> right, 2* currIndex + 2});
            }
        }
        ans = max (ans, last - first +1);
    }
    return ans
}
```

# Children sum property :-

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

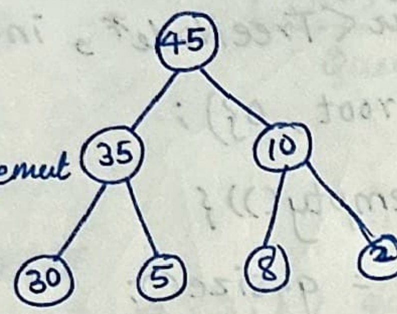
→ If this condition is not satisfying
(1) you can add +1 to any node to any number of times

$2 \neq 35+10$

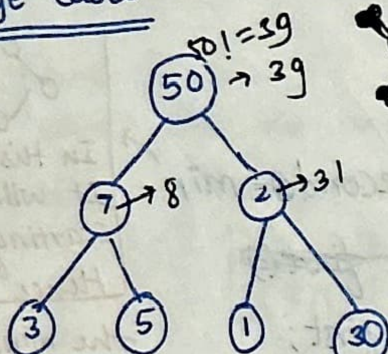



⇒

You can increment any node However you want



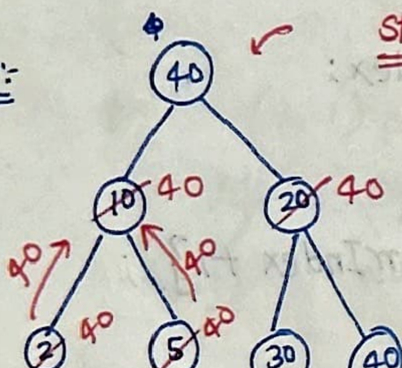

# Edge Cases -




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 !!

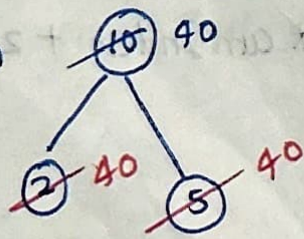
→ We will be using recursive traversal

Ex:-




Start → root → ~~left~~ left = 10
root → 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

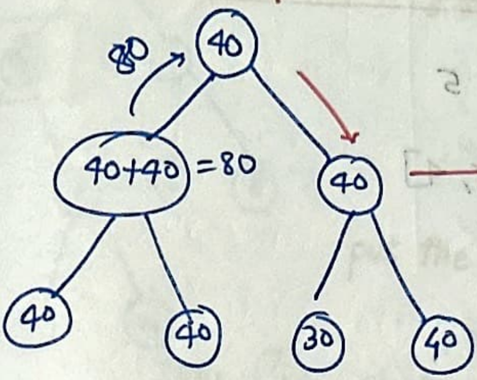
from

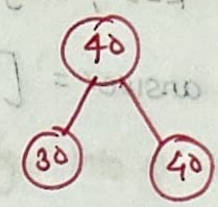



→ $2+5 < 40$
Reassign 2 and 5 to 40

Now from 2 → No other nodes are present hence return 40 from 2 and same, return 40 from 5

→ Add them up




30 + 40 = 70

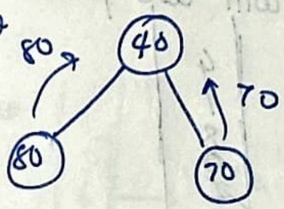
70 > 40

so make the node 40 (parent) as 70.

move left 30 → return 30

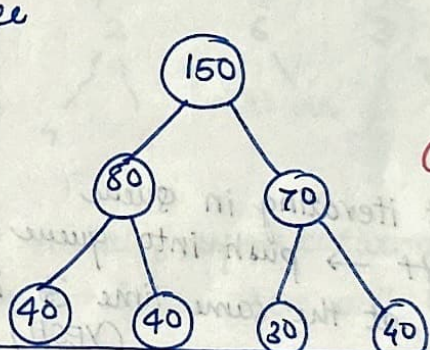
move right 40 → return 40

Now




⇒ Update the root by 80 + 170 = 150

⇒ Hence the tree



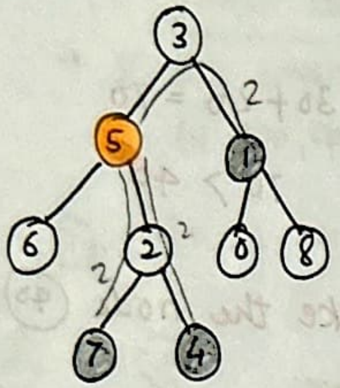

Children Sum property Satisfied

```
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;
        else if(root->right) root->right->data = root->data;
    }
    ChangeTreeBinary(root->left);
    ChangeTreeBinary(root->right);
    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;
```

While coming back to parent

T.C = O(N)

## Print all the Nodes at a distance of "K" in Binary Tree :-

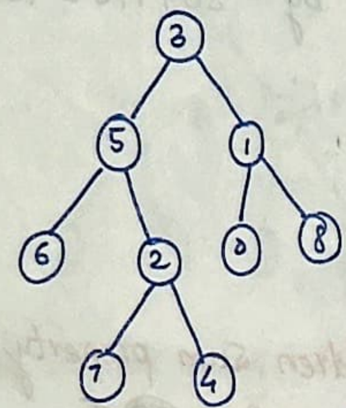



k=2 , target= 5

answer = [1, 7, 4]

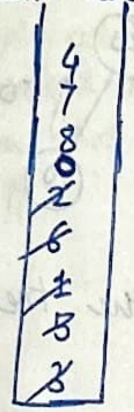
→ If we want to get all the nodes from node 5 at a distance "2" then we have to move downward + upward in both the direction AND This is the main concern.

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




k=2, target=5

(BFS)




Queue

Start iterating in queue

if 3→ left → push into queue (5)

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

Now 3→right ⇒ push into queue (1)

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

This is how we can map the parent nodes.

5→left →6 →push Q

5→right → 2 →push into Q

1→ left → 0 →push into Q

1→right → 8 → push into Q

6 →left ⇒NULL

6→right ⇒NULL

2→left → 7 →push into Q

2→ right→4 → push into Q

like this

parent child map

{ child → parent

5 → 3

1 → 3

6 → 5

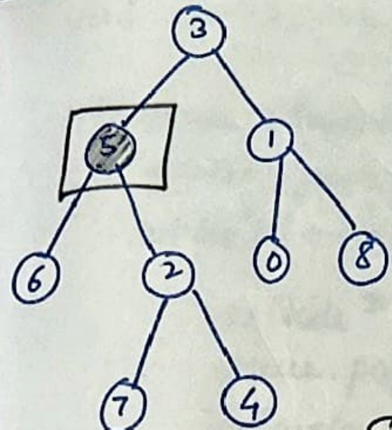
2 → 5

7 → 2

4 → 2

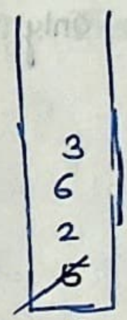
0 → 1

8 → 1

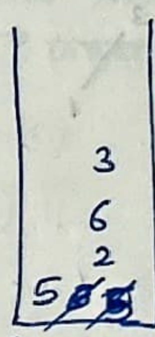



dist = 0

put the target into Q →

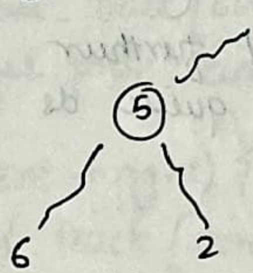



Q



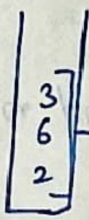

Carry a visited hash

take 5 check upward, left and right

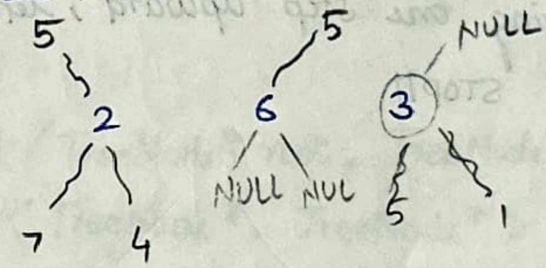



→ push those nodes into Q
dist = 1

Now



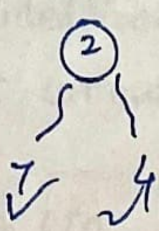

→ Take all these three nodes and try to move radially outwards



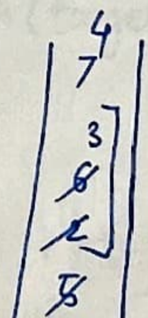

Now think!!

→ If we move radially ~~upwa~~ outwards from 2 we will have 5 which is useless → hence there will help our **visited hashmap**

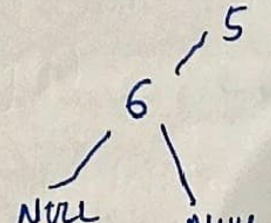
from 2 to 5 → you will not go cause 5 is already visited



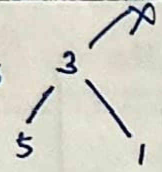
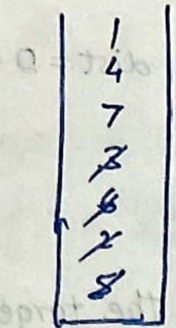

push into Q
(7 and 4)




Now for 6 →




→ Nothing will be pushed into queue

for 3  → only 1 pushed 

Now distance will be updated to 2

and compare ~~with~~ this distance to k (k== distance)

Hence ~~your~~ no need to proceed furthur

and the nodes present in your queue ds is at 'k' distance from target.

→ Summary:-

→ Have parent pointers.

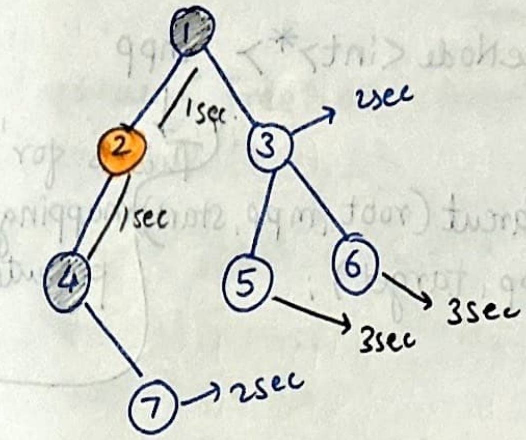
→ keep on moving one step upward, leftward and rightward

→ at dist == k STOP!!

```
void markParents(TreeNode* root, unordered_map <TreeNode*, TreeNode*>
                                     & parent_tract, 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;
    mark Parents (root, parent_track, target); -> mapping done!

    unordered_map< TreeNode*, bool > visitedNodes;
    queue <TreeNode*> queue;
    queue.push(root); 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~~) Not present
↳ it can burn above(1)
It can burn left (4)

Queue: ~~2~~, ~~1~~, 4, 3, 7, 5, 6

visited:
2 → True
1 → true
4 → true
3 → true
7 → true
5 → true
6 → true

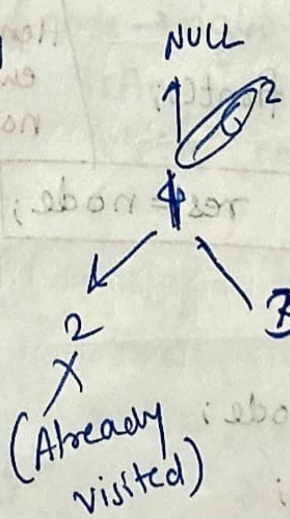
This is just for map (no use)

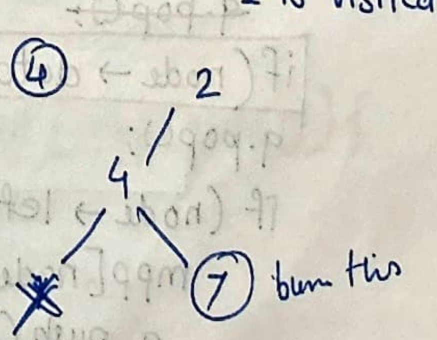
2 is visited

Now pick both the nodes

time = 1

4
1




time = 2

7
3

Now perform the operation on this ⇒ time = 3

⇒

6
5

Hence don't update the time
5 doesn't burns anyone
and so 6 also don't burn

```
int findMinTimeToBurn( TreeNode <int* root , int start){
    map < TreeNode <int>* , TreeNode <int>*> mpp

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

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

~~queue Binary~~
    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;  }
```

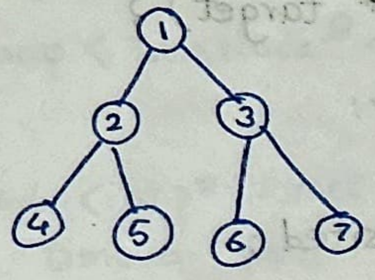
mpp → This is for mapping of parents

int start

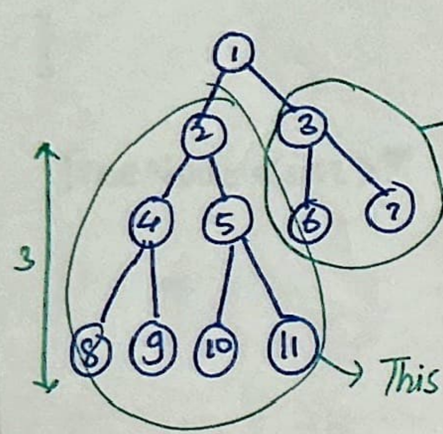
Since in question int start is given they did not given us the address of the node Hence whenever we encounter the node → data == start Store the address (node) and return that;

```
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 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]) {
             isAnyNode Burnt = 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




No. of nodes = $2^3 - 1$




→ This is also a Complete BT $= 2^2 - 1 = 3$

→ This is complete binary tree $= 2^n - 1$
$= 2^3 - 1$
$= 7$

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

$lh = 4$
$rh = 3$
$(lh \,!= rh)$ → hence its not complete binary tree → can't apply $2^n - 1$ formula directly

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

```
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;
}

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

$\log N * \log N$

$= (\log N)^2$

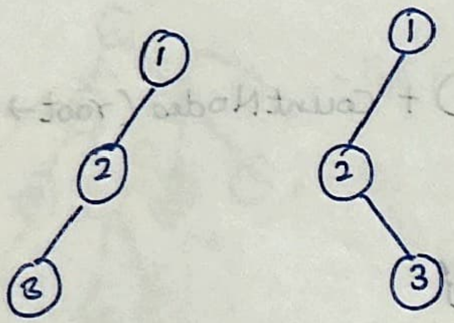
• **Requirements needed to construct a unique binary tree :-**

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

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 $\boxed{\text{UNIQUE}}$ binary tree.

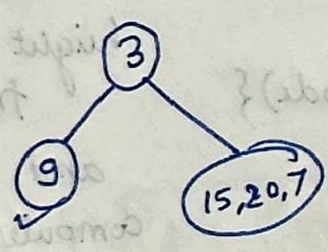
Q.2) What if preOrder & Inorder is given

Left Root Right

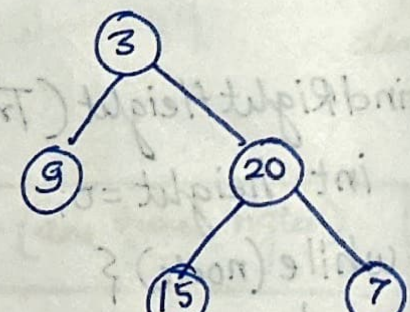
InOrder = [9 3 15 20 7] - (Left - Root - Right)

PreOrder = [3 9 20 15 7] - (Root - Left - Right)

Root




⇒




⇒ And this is unique binary tree

⇒ We can create the unique binary tree only if the

⇒ InOrder + PreOrder

⇒ InOrder + PostOrder

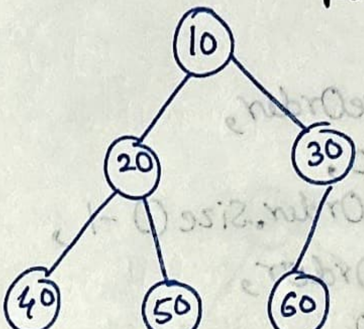
is given

- Create Binary tree from InOrder & PreOrder :-

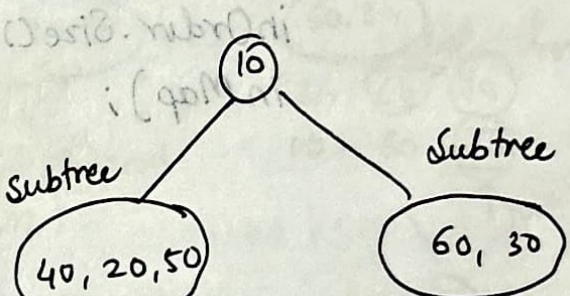
InOrder = [40, 20, 50, 10, 60, 30] → (Left - Root - Right)

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

Root




⇒ This is the tree




find the Inorder & preOrder

Inorder= 40, 20, 50

PreOrdr = 20, 40, 50

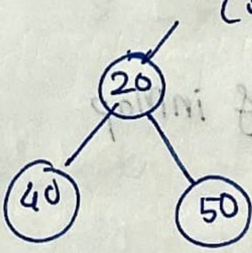
This is a new problem (sub problem)

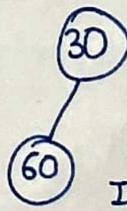
find the Inorder & preOrdr

Inordr = 60, 30

pre Order = 30, 60

This is a new problem (Sub problem)




InOrdr = 40

pre : 40

(New Subproblem)

In : 50

pre : 50

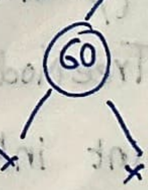
(New Subprob.)

In : 60

pre : 60

(New Subproblem)






if (you find the both child as null then attach the node

```
TreeNode* build (vector<int>& preOrder, vector<int>& inOrder
    map<int, int> inMap;
    ~~for(auto i: inOrder){
        inMap[i]++;
    }~~
    for(int i=0; i< inOrder.size(); i++){
        inMap[inOrder[i]] = i;
    }

    TreeNode* root = buildTree(preOrder,
                        0,
                        preOrder.size()-1,
                        inOrder,
                        0,
                        inOrder.size()-1,
                        inMap);

    return root;

}

TreeNode* buildTree( vector<int>& preOrder,
                    int preStart,
                    int preEnd;
                    vector<int>& inOrder,
                    int inStart,
                    int inEnd,
                    map<int, int>& inMap
                  ){

    ~~if( preEnd~~
    if(preStart > preEnd || inStart > inEnd) return NULL;
    TreeNode* root = new TreeNode(preOrder[preStart]);
    int inRoot = inMap[root->val];
    int numsLeft = inRoot - inStart;
    root->left = buildTree(preOrder, prestart+1, preStart+numsLeft,
                        inOrder, inStart, inRoot-1, inMap);
    root->right = buildTree(preOrder, prestart+numsLeft+1, preEnd,
                        inOrder, inRoot+1, inEnd, inMap);
    return root;
```

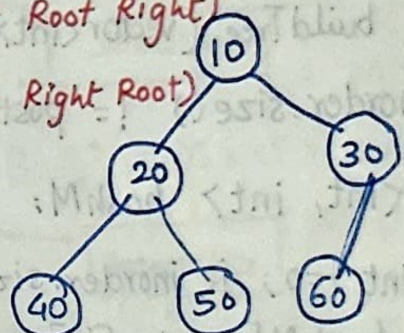
T.C = O(N)
S.C = O(N)

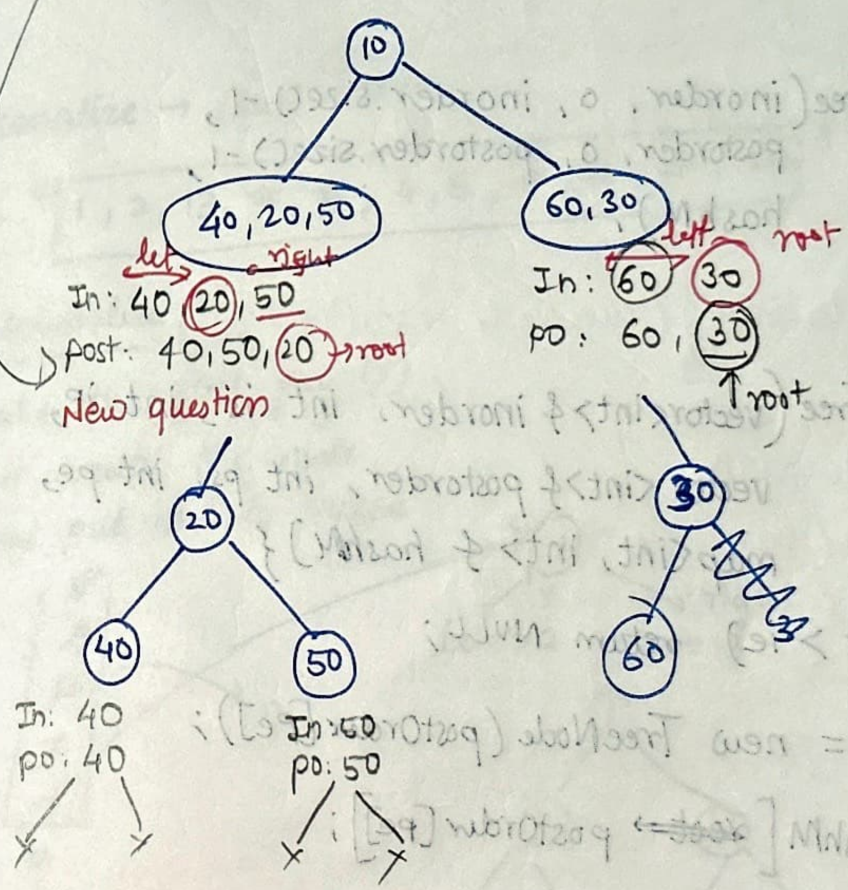
- Construct the binary tree from PostOrder and Inorder :-

Left Subtree ⟵ ⟶ Right Subtree

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

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

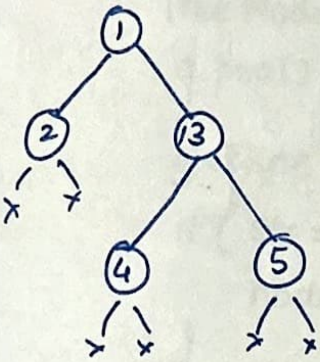


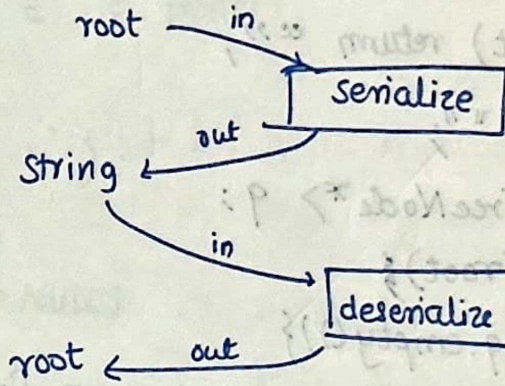



whatever postOrder is given the last element is root.

```
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 :-



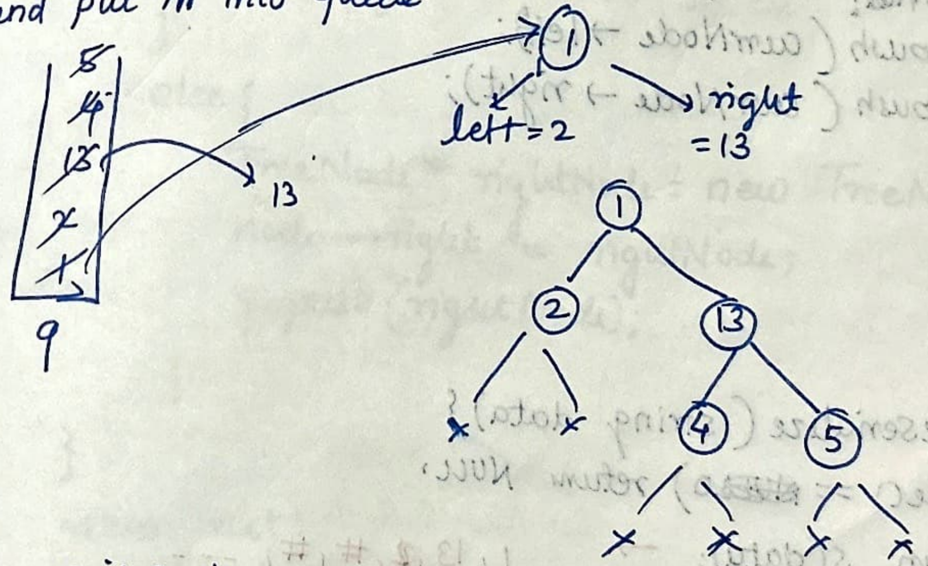



# serialize → Level order

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

# deserialize

take the first guy (1)
and create the node
and put it into queue




q. is empty

```
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);   →   1, 13, 2, #, #, ---
    string str;
    getline(s, str, ',');   →   by using getline, it will automatically picks the element
                 ↳ delimeter
```

by using getline, it will automatic picks the element

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

~~TreeNode* root~~

```
    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 ?

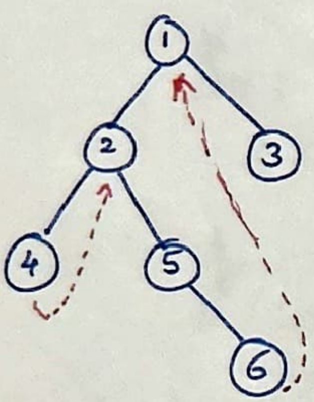
→ In the recursive traversal ⟶ T.C = O(N) & S.C = O(N)

but in morris traversal ⟶ T.C = O(N) & $\boxed{S.C = O(1)}$

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 Sub tree and if you go back ⟹ root

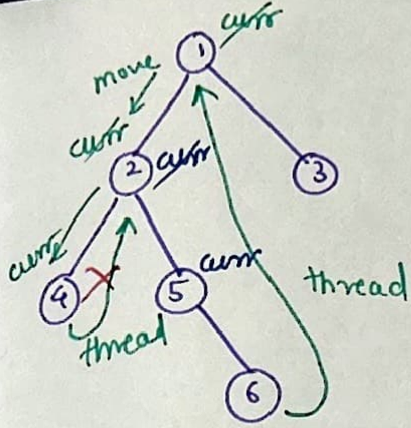
**Case 1:-** If the left → NULL

⟹ print the current node

⟹ move to right

**Case 2:-** Before going to left → whichever the righmost 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 2
print ⇒ 42

Now on 2 curve says
↳ when I go to left → the last guy on left sub tree ~~itself~~ is already pointing to himself (means 2)
so in this case go to the left ⇒ find the rightmost guy and cut off the link take the curve and move right ⇒ 5

5 does not have left → so its already a root → print 425

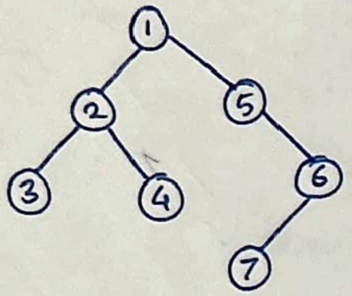
root is printed → go to the right → print → 4256
and through the curve you came back to 1 print ⇒ 42561

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

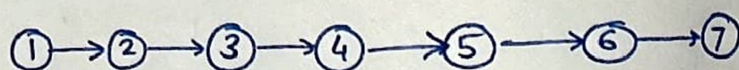
move right → 3 ⇒ print 425613

```
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;
            }

            // case 1 :
            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 :-



⇒



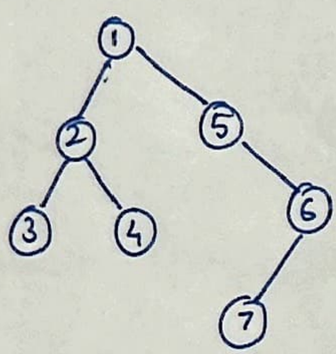
↳ preorder ⇒ 1 2 3 4 5 6 7

Whatever is the preorder → You might think like this
Create a node and
generate LL

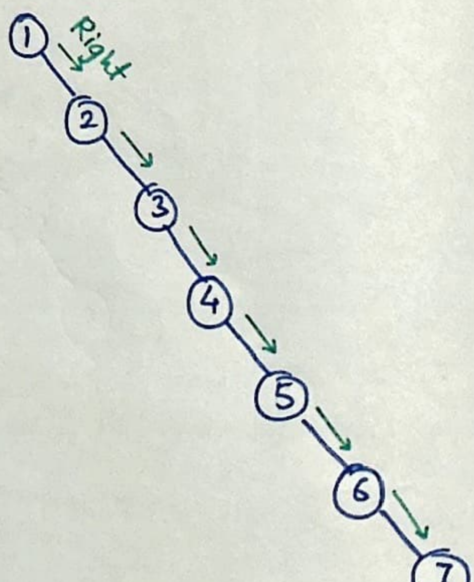
BUT THIS IS NOT ACCEPTABLE!!

Because we have to flatten the BT

EX:-

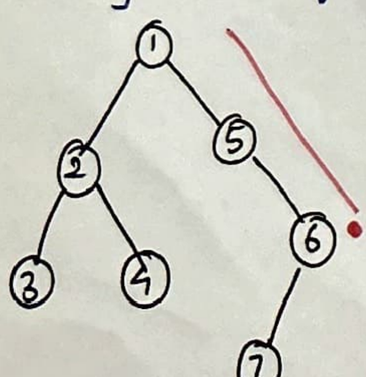


⇒




### Approach : I

basically rearrange the tree

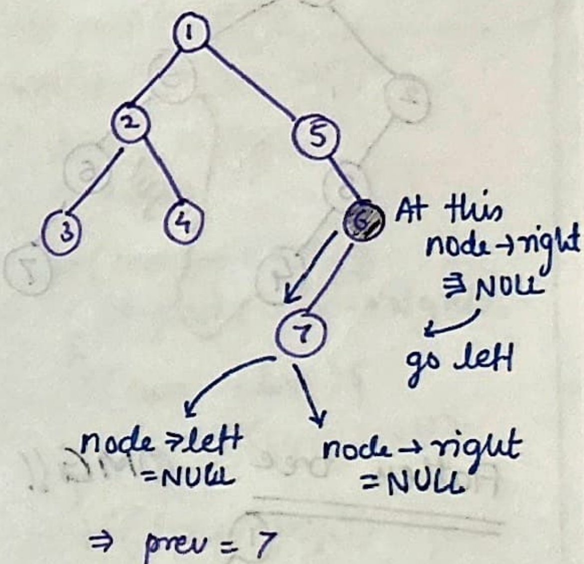


• go till ⑥ then arrange 7
& After that deal with the left Sub tree

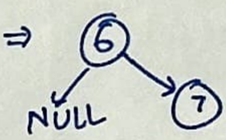
so the traversal will be

$\boxed{\text{Right} - \text{Left} - \text{Root}}$

```
prev = null;
flatten(node){
    if(node == null) return;
    flatten(node→right);
    flatten(node→left);
    node → right = prev;
    node → left = null
    prev = node
}
```

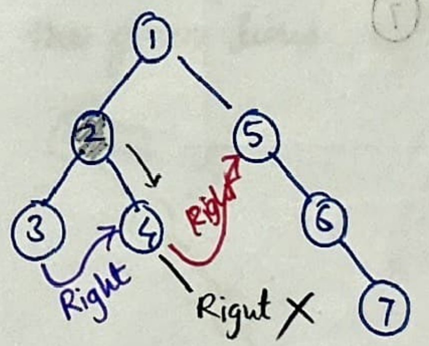



next step → ⑥ ↘Right ⇒ prev

⇒



prev = 6

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



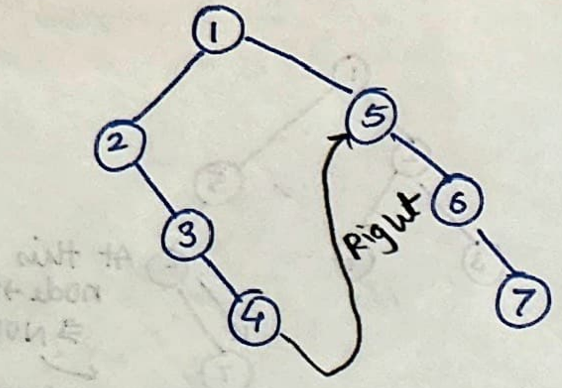

Hence prev = 5 and at ④ node→right = prev
means ④ $\xrightarrow{\text{Right}}$ ⑤

Now prev = 4

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

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

Now comes back to ②, It's right ⇒ DONE, left ⇒ DONE } point ② $\xrightarrow{\text{right}}$ prev
② $\xrightarrow{\text{right}}$ ③




after pointing the ② $\xrightarrow{\text{right}}$ ③

⇒ prev = 2

↓

Come back to 1

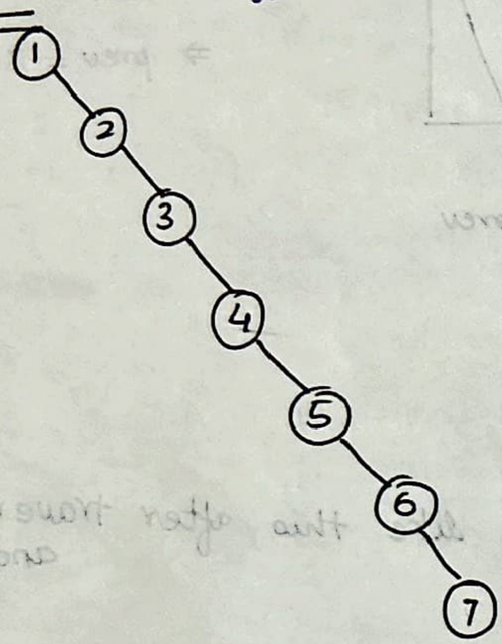
Now its left and right is done!

hence ① $\xrightarrow{\text{Right}}$ prev

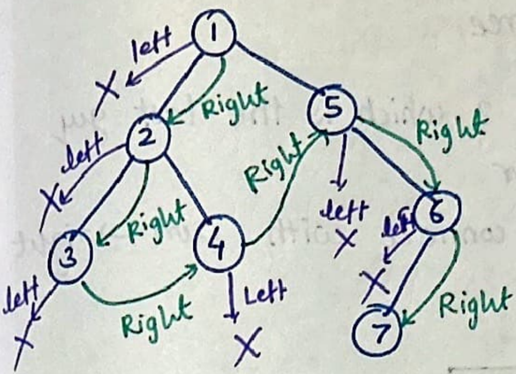
① $\xrightarrow{\text{Right}}$ ②

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

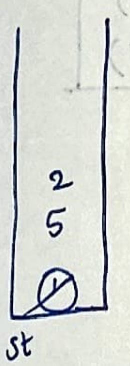
Flatten tree **OMG!!**



# Approach : II    Using stack!




```
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;
}
```




curr = 1

due to this line

1 → Right = st. top()

↓

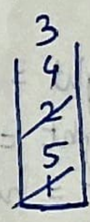
2

1 → left = NULL

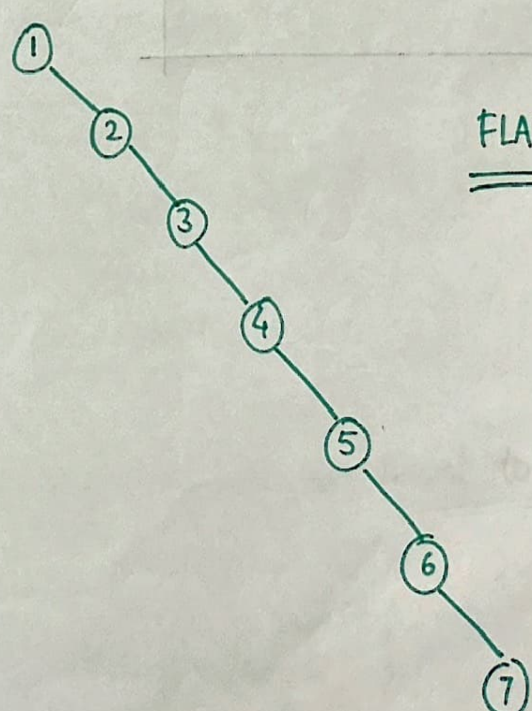
pick 2 ← curr

2 ↙ 3

2 ↘ 4 → put it into st.




Now join the green lines

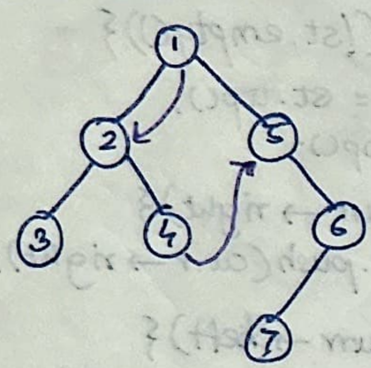



FLATTEN!!

T.C = O(N)

S.C = O(N)

## Approach : III

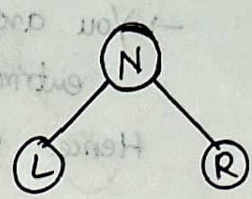


on the left subtree
↓
Try to figure out ? which is the last guy in the preorder
and if found connect with curr → right

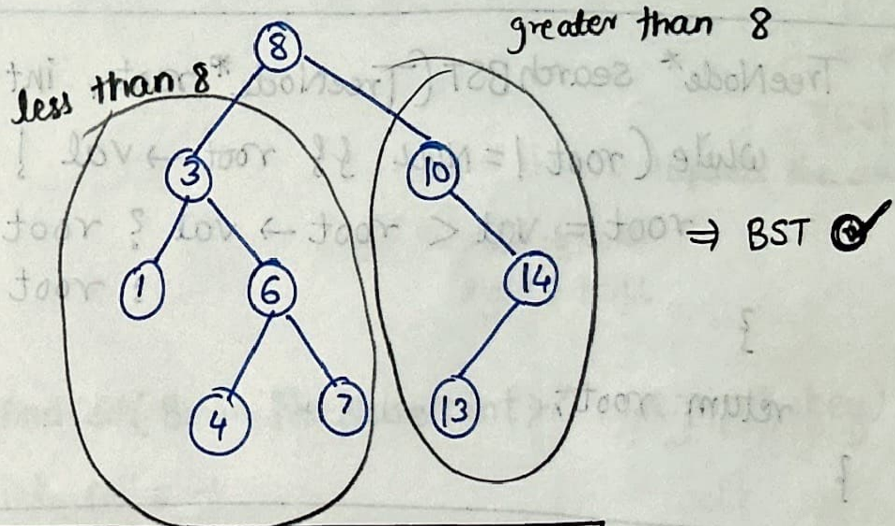
```
curr = root;
while (curr != NULL) {
    if (curr->left != NULL) {
        prev = curr->left;
        while (prev->right) {
            prev = prev->right;
        }
        prev->right = curr->right;
        curr->right = curr->left;
        curr->left = nullptr;
    }
    curr = curr->right;
}
```

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

# Binary Search Tree




BST signifies ⇒ 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.

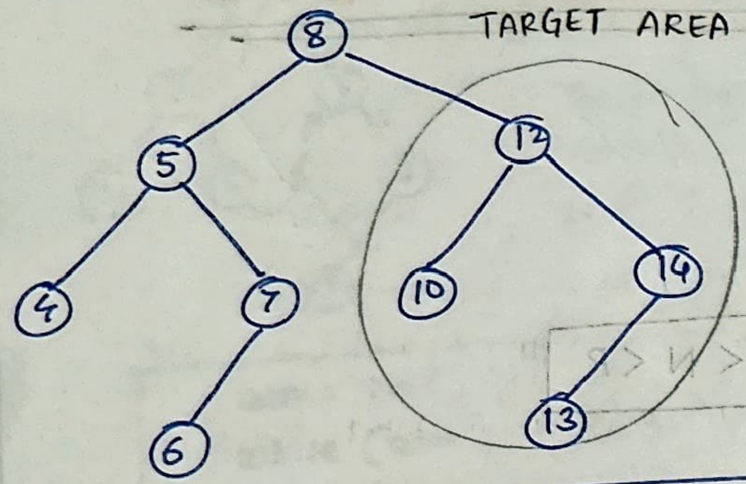
## Why BST ?

In binary tree, if we want to search any node ↳ O(N)

but in BST → $\boxed{O(\log N)}$

* Search in BST :-

node = 10



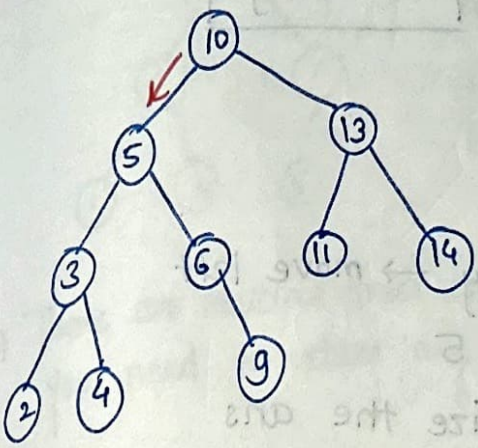

→ You are not traversing the entire tree

Hence ~~of~~ T.C = O(logn)

```
TreeNode* searchBST (TreeNode* root, int val){
    while (root != NULL && root→val != val){
        root = val < root→val ? root→left
                              : root→right;
    }
    return root;
}
```

# Ceil in a BST :-

$\boxed{val >= key}$



key=8

root = 10 ⇒ 10 ≥ 8 ✓
But we need to reduce this as much as possible.

Now key < root → move left

from (5) → go to right ⇒ (6) is 6 possible
NO!

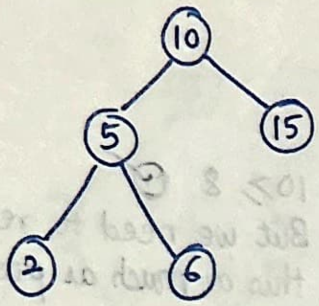
from (6) → go to right ⇒ 9
↓
is 9 possible ?
YES!
update the ans

go to left ⇒ NULL
Right ⇒ 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

$\boxed{val <= key}$

6<7

root = 10 ⟶ root > key → move left

root→data = 5 ⟶ ans = 5

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

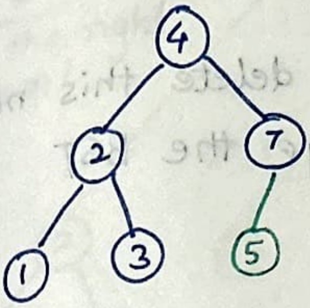
6 ⟶ 6 <= 7 → True   ans = 6

move left ⇒ NULL
move right ⇒ NULL

return 6

```
int findFloor ( BinaryTreeNode<int>* root , int key){
    int floor = -1;
    while(root != NULL) {
        if (root → data == key) return root → data;

        if (root → data < key){
            //move right
            ~~root =~~ floor = root → data;
            root = root → right;
        }
        else {
            root = root → left;
        }
    }
    return floor;
}
```

- Insert a given node in Binary Search Tree :-




node = 5

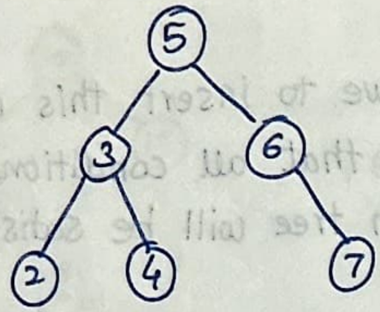
Now you have to insert this node in such a way that all conditions for binary search tree will be satisfied.

→ 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 new TreeNo~~
        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); ← Insertion
                break;
            }
        }
        else{
            if(curr→left != NULL) curr = curr→left;
            else{
                curr→left = new TreeNode(val); ← Insertion
                break;
            }
        }
    } return root;
```

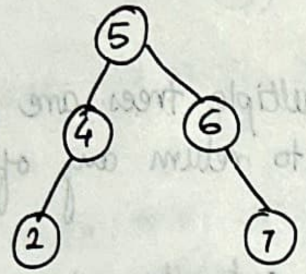
## Delete a Node in Binary Search Tree :-



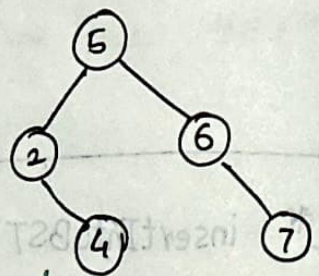
node = 3

You have to delete this node and rearrange the BST

I ⇒



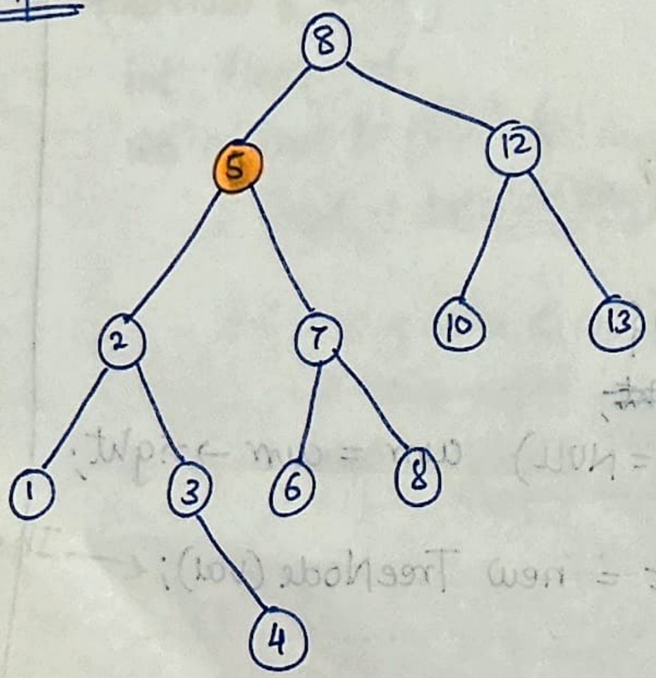
II ⇒



There are multiple solutions present

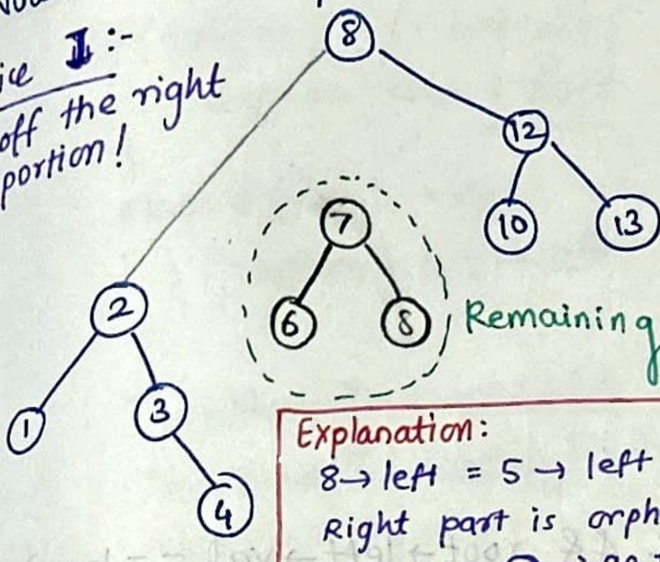
**Example :-**

node = 5

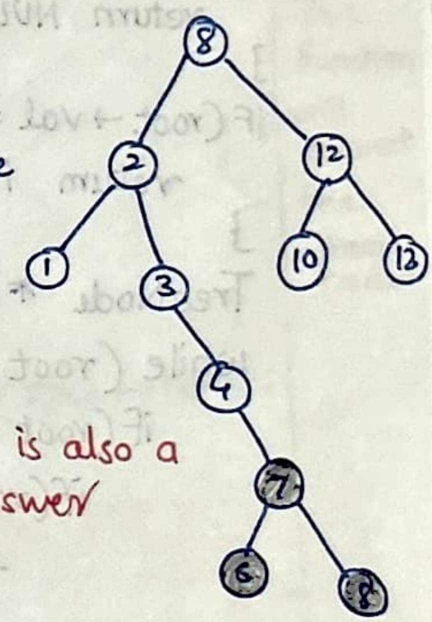


Now if I simply delete the 5

choice I :-
Cut off the right portion !



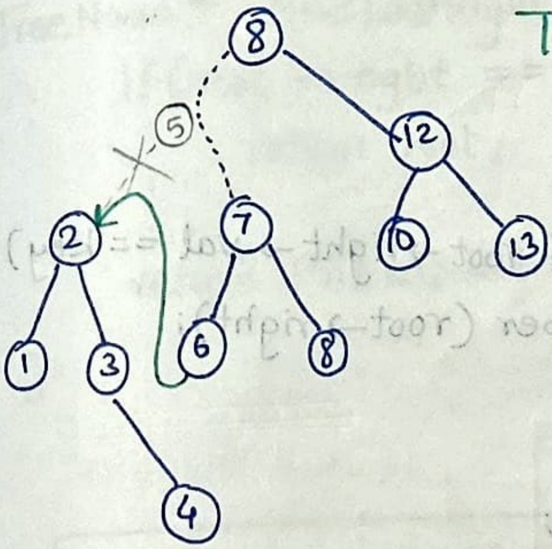

if I do like this →




Explanation:
8→left = 5→left
Right part is orphan
Now from 2 → go to extreme right
make sure
4 →right = 5→right

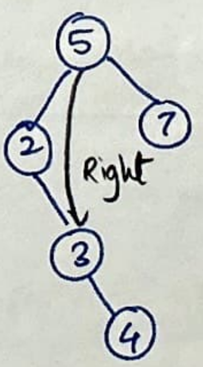
This is also a answer

Choice : II



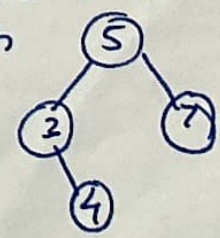

The smallest number on the right subtree = 6
connect this to 2

# EDGE CASE :-




node = 2

So in this there is no left so simply attach



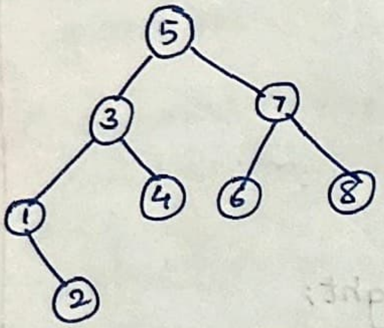
```
TreeNode* deleteNode(TreeNode* root, int key){
    if (root == NULL) {
        return NULL;
    }
    if (root→val == key) {
        return helper(root);
    }
    TreeNode *dummy = root;
    while (root != NULL) {
        if (root→val > key) {
            if(root→left != NULL && root→left→val ==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

}
```

This function will connect the ~~liner~~ nodes

$\boxed{\text{Time Complexity} \Rightarrow O(H) \quad \text{Space Complexity} \Rightarrow O(1)}$

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



$k = 3$

Write down the inorder :-

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

Imp!!

**Inorder of binary Search tree is always in the sorted order**

Maintain the count as "0"
and whenever you encounter the node → count++;

Three Approches :-

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 = ~~O(~~ O(1) → preferred solution

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

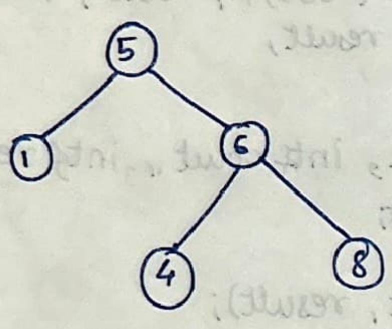
logic ⇒ $k^{th}$ largest = ~~(N - k th) smallest element~~
(N-K+1)th smallest element

```
int kth Smallest (TreeNode* root, int k) {
    int count = 0, result=0;
    ~~traverse (root, k, count);~~   traverse (root, k, count, result);
    ~~return 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 kth Largest (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 ⇒

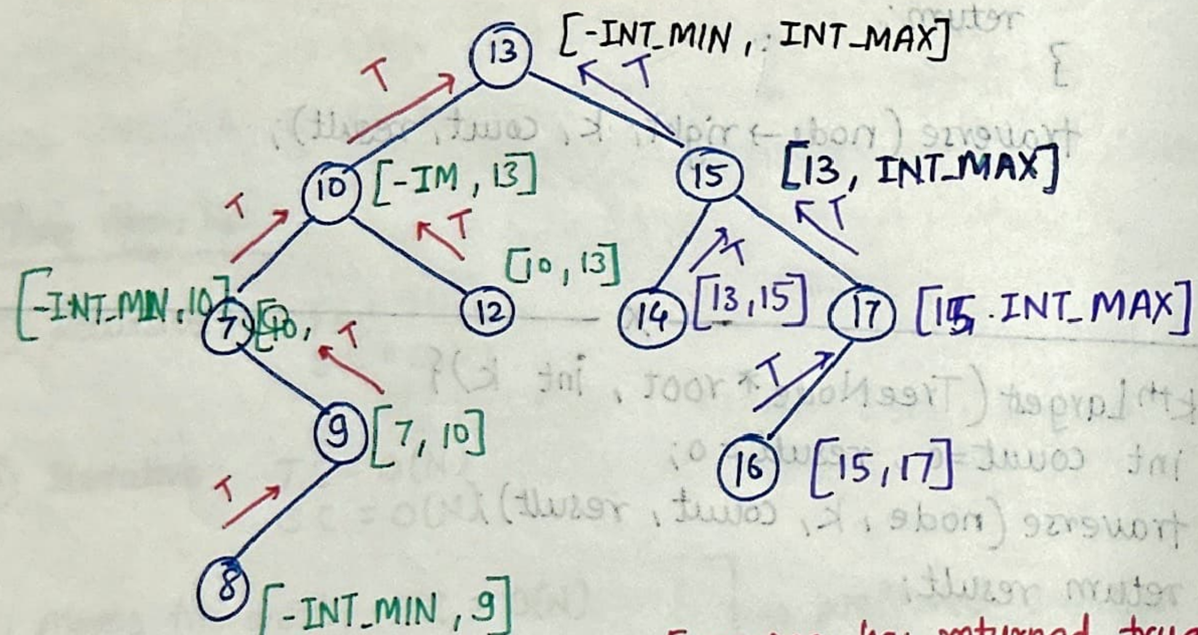
for 4 It should be lesser than 6
AND should be greater than 5

Hence we need to consider the scenario

**Intution :-**

node = [L, H]

Example :- for 4 it should lie betn [5, 6] ⇒ False

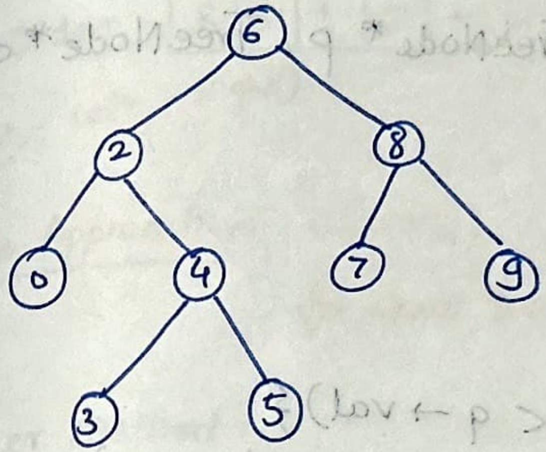



Everyone has returned true
⇒ Hence its a valid BST

$$\boxed{\begin{array}{l} T.C = O(N) \\ S.C = O(1) \end{array}}$$

```
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)
        && isValid(root->right, root->val, maxValue)
}
```

# • Lowest Common Ancestor in Binary Search Tree :-

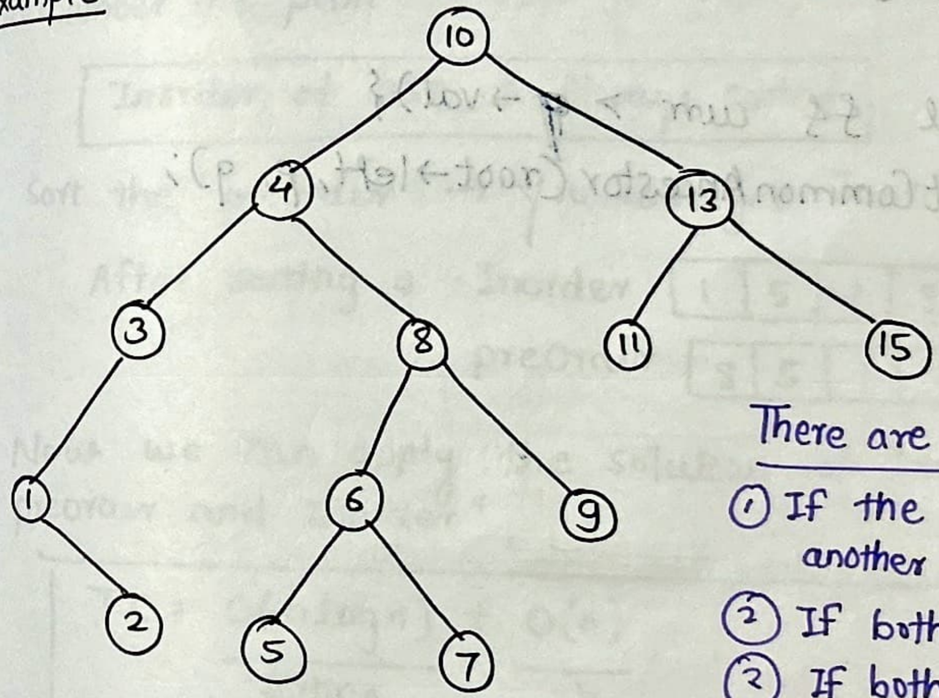


LCA(2,5) = 2

LCA(0,5) = 2

first intersection path (from bottom)

Example :-



There are 4 possibilites

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 ⇒ 4

(4) → 5 & 9 are greater ⟶ move to right ⇒ (8)

on (8) → 5 is on left and 9 is on right ] → This is where the path has been splitted so this is my LCA

$\boxed{T.C = O(H) \quad S.C = O(1)}$

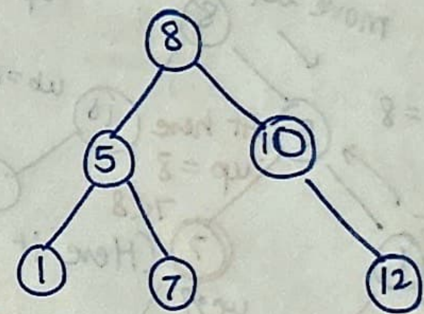
```
TreeNode* lowestCommonAncestor(TreeNode* root,
                               TreeNode* p, TreeNode* q) {

    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;
}
```

## Construct a BST from preorder traversal :-

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

preOrder

(Root - Left - Right)




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

**Better Method** :-

Remember the point

$\boxed{\text{Inorder of BST} \Rightarrow \text{Always sorted}}$

Sort the preorder → you will get Inorder

After sorting ⇒ 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".

$$T.C = \underbrace{O(n\log n)}_{\text{sorting}} + \underbrace{O(n)}_{\text{construct the tree}}$$

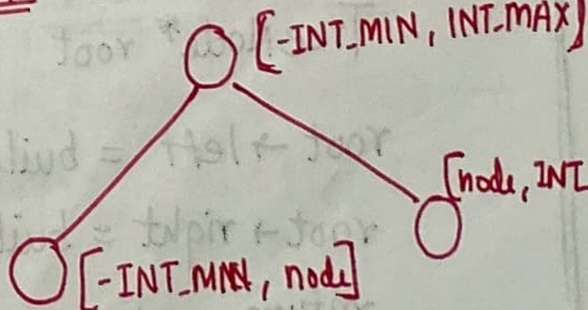
$$S.C = O(N)$$

**Efficient method** :-

preOrder

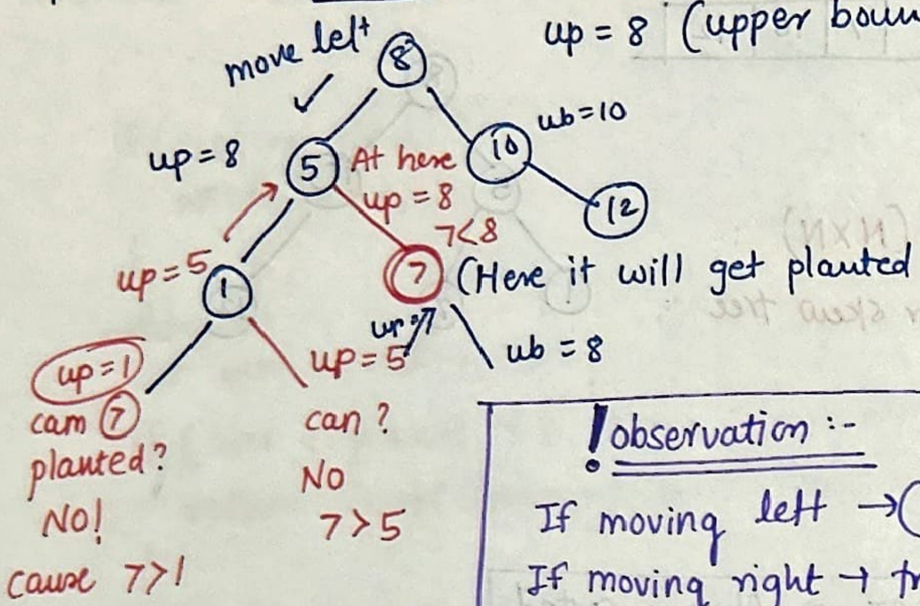
| | | | | | |
|---|---|---|---|---|---|
| 8 | 5 | 1 | 7 | 10 | 12 |

Ref: check if tree is BST or not

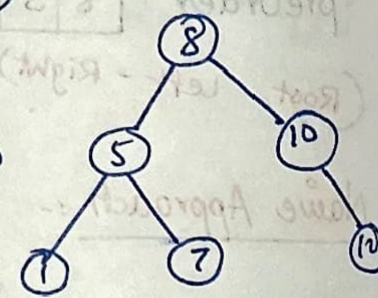


preOrder | 8 | 5 | 1 | 7 | 10 | 12 |

up = 8 (upper bound)




⇒



**! observation :-**

If moving left → (node → val) upper bound

If moving right → transfer the bound

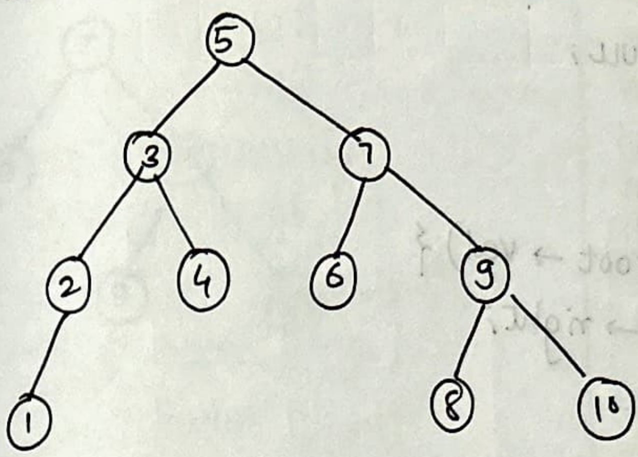
T.C = O(3N)
↓
You are visiting 3 times a single node

$\boxed{T.C \approx O(N)}$

$\boxed{S.C = O(1)}$

```
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 :-



Val = 8

Inorder ⇒ 1 2 3 4 5 6 7 8 9 10

↑
Inorder
Successor

If the no successor is present → return NULL.

## Efficient Approach :-

successor = NULL     val = 8

At root → 5 → I should move to right

Now at 7 → I should move to right

at 9 → Here we can say 9 is the successor

But are we sure ?

~~NOT~~ → 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 $\underline{\underline{> 8}}$ and $\underline{\underline{< 9}}$

move left → 8 found

Now move right we reach NULL.

Hence our successor is $\underline{\underline{9}}$.

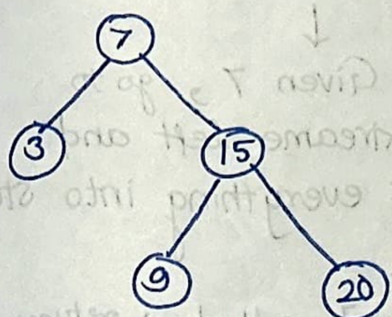
$$\boxed{\begin{array}{l}\text{Time Complexity} = O(H) \\ \text{Space Complexity} = O(1)\end{array}}$$

```
TreeNode* inOrderSuccessor(TreeNode * root , TreeNode* ~~x~~){
    TreeNode* Successor = NULL;
    while (root != NULL) {
        ~~if( val~~
        if(x → val >= root → val) {
            root = root → right;
        }
        else {
            Successor = root;
            root = root → left;
        }
    }
    return successor;
}
```

# predecessor :-

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

• BST Iterator :-



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 ⇒ Left - Root - Right

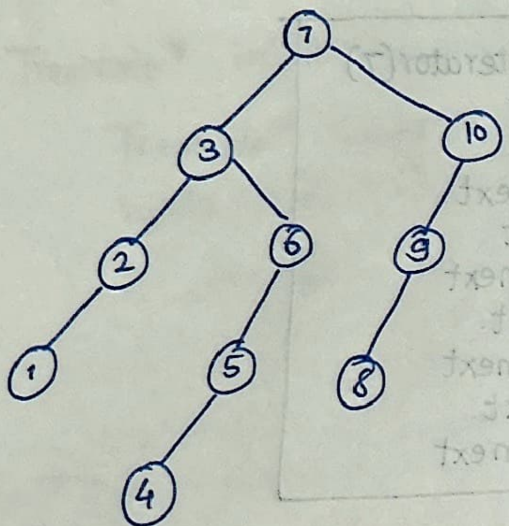
BSTiterator(7)
next
next
hasnext
next
hasnext
next.
hasnext
next.
hasnext

T.C = O(1) for returning the

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 ⇒ O(H) T.C




BST iterator (7)

↓

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

~~1~~
~~2~~
~~3~~
7

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)~~ elements

⇓

~~4~~
5
6
7

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

next → 4 → return 4
If 4 has right → X

next → 5 → return 5

This is how this will work!

pushing n elements into stack → N

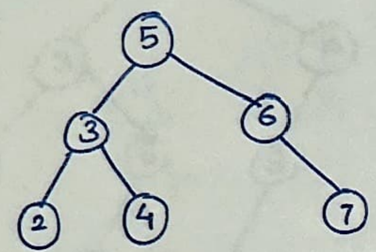
& there might be N [next] calls = T.C ⇒ $O(\cancel{N}/\cancel{N})$

$\boxed{S.C = O(H) \quad T.C = O(1)}$

$\boxed{T.C = O(1)}$

```
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;
    }

private:
    void pushAll(TreeNode* node){
        ~~for( ; node != NULL ;~~
        while (node != NULL){
            mystack.push(node);
            node = node->left;
        }
    }
}
```

- Two Sum IV in BST



$\underline{k=5}$

return true if pair exists, return false

Brute force :-

why inorder have taken ? cause its always sorted !

find the inorder ⇒

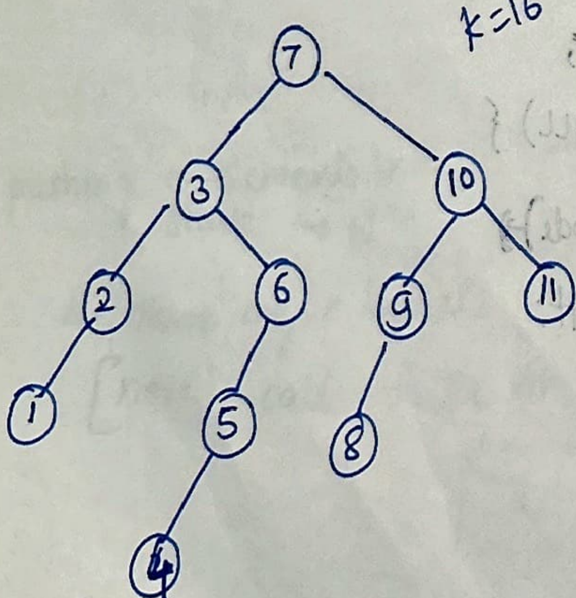
| 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|
| ↑ l | ↑ r → GOT IT!! | | | | ↑ r |

then take $l=0$
$r = \text{size}-1$

if $2+7=k \rightarrow$ No ⇒ $9>5$
then reduce the greater no. index
r--;
$2+6=8>5$
reduce r--
$2+5=7>5$ ⇒ r--;
$2+4=6>5$ ⇒ r--
$2+3=(5=5)$ ⇒ return true

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

$k=16$



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

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

Now If you ~~iterate~~ DFS ⇒ Right - Node - Left
⇓
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 ith pointer stands at i = next()
and jth pointer = j = before()

next will be the smallest element ⇒ i = 1
before will be the largest element ⇒ j = 11

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

2 + 11 = 13 (move upward)

3 + 11 = 14 increase

Now 'i' is moved to 4 ⇒ 4 + 11 ⇒ 15 (increase)
5 + 11 ⇒ 16 MATCHED
↓
return true.

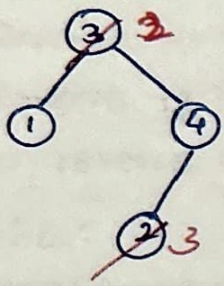
$$\boxed{\begin{array}{l} T.C = O(N) \\ S.C = 2 \times O(H) \end{array}}$$

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();
            }
        }
        return false;
```

## 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 ⇒ Inorder traversal and sort it
and while going to

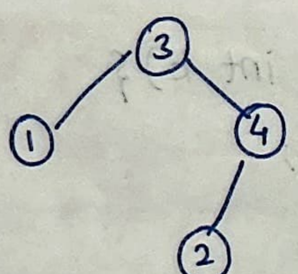
node → left
node
node → right

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

Sorted inorder :-

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

idx = 0




node ⇒ 1 → inorder [idx] = 1 ✓ idx++
and then at 3 inoder [idx] = 2
↓
✗ swap

this is how you can process.

$$\boxed{\begin{array}{l} T.C = 2N + N\log N \\ S.C = O(N) \end{array}}$$

## Better Approach

I) Swapped nodes are not adjacent

3 25 7 8 10 15 20 5    NOT ADJACENT

II) Swapped nodes are adjacent.

~~3 5 7 8 7 10 15 20 25~~

3 5 8 7 10 15 20 25

### Case I)

3 25 7 8 10 15 20 5

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

25 → first
7 → middle

Now from 20 → 5 Again violation and if we found the second violation last = 5

swap (first & last)

### Case :- II)

3 25 7 8 10 15 20 5

first = 7
middle = 8

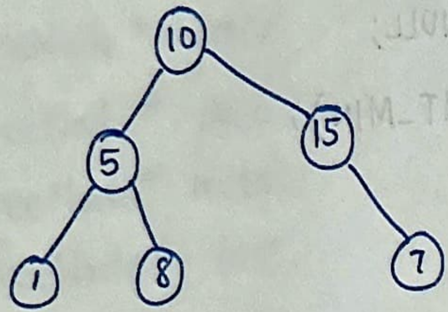
No second violation found ⇒ swap (middle, first)

$\boxed{\begin{array}{c} T.C = O(N) \\ S.C = O(1) \end{array}}$

```
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 vioation, mark this node as last
            else {
                last = root.
            }
        }
        // mark this node as previous
        prev = root;
        inorder(root->right);
    }
```

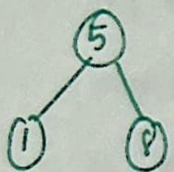
```
public:
    void recoverTree (TreeNode* root){
        first = middlle = last = NULL;
        prev = new TreeNode(INT_MIN);
        inorder( root);
        if ( first && last) {
            swap(first → val, last → val);
        }
        else if ( first && middle) {
            swap( first → val,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

- ## Largest Binary Search tree in a given binary Tree :-



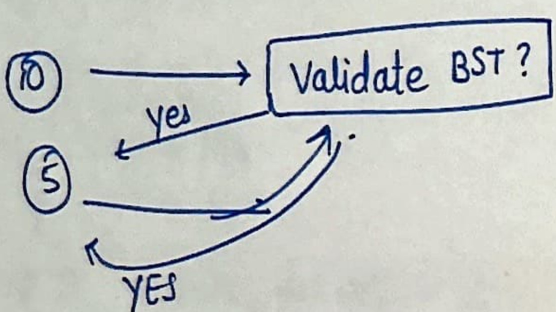
This is not BST

 ⇒ BST

 ⇒ BST

### Brute force :-

**Ref: Validate BST**




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

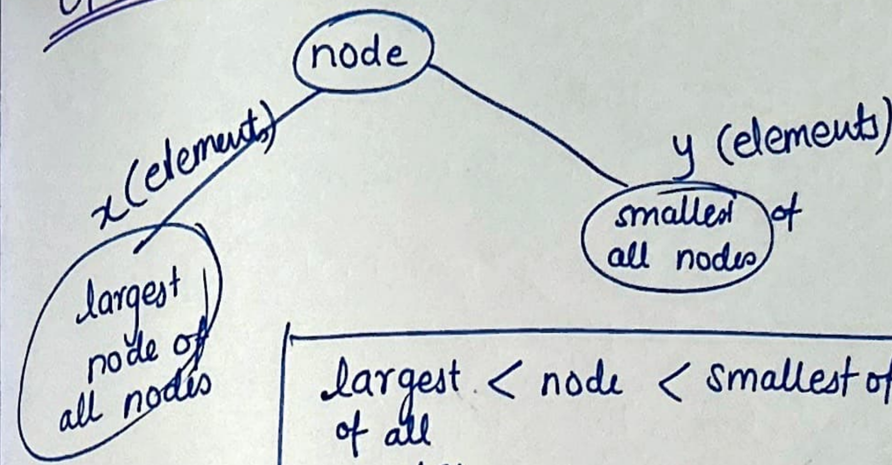
$$T.C = O(N) * O(N)$$

↑ Validate a BST (first $O(N)$)

↑ reading to every node (second $O(N)$)

$$\boxed{T.C = O(N^2)}$$

## Optimal :-




largest of all nodes < node < smallest of all nodes

size = $1 + x + y$