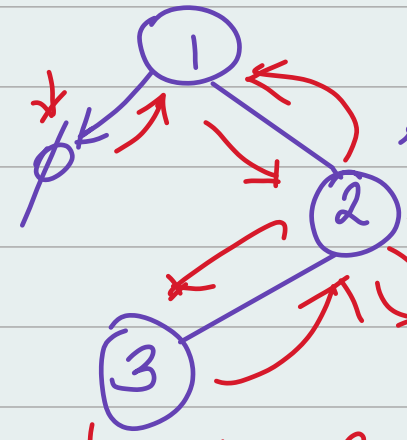
81) Binary Tree In Order Traversal :-

In Order Traversal :- (Sort the numbers in BST)

Left - currentNode - Right



Input :
root = [1, null, 2, 3]

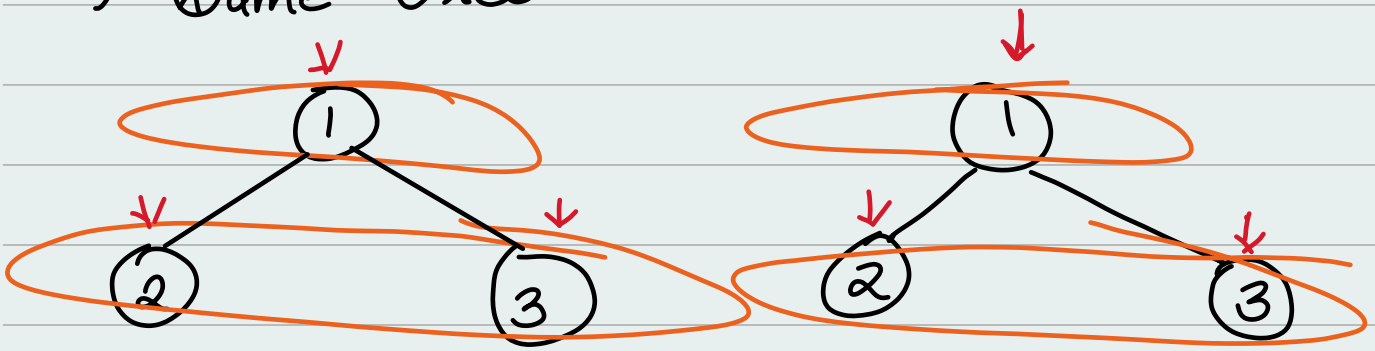
output : [1, 3, 2]

output 1, 3, 2 Ans

```
class Solution{

    public List<Integers> inorderTraversal(
                                TreeNode root){
        List<Integer> ans = new Arraylist<>();
        inOrder(root, ans);
        return ans;
    }
    private void inOrder(TreeNode node,
                         List<Integer> ans){
        if(node == null){
            return;
    }
```

```
        inOrder ( node.left, ans );
        ans.add ( node.val );
        inOrder ( node.right, ans );
    }
}
```

Q2) Same Tree



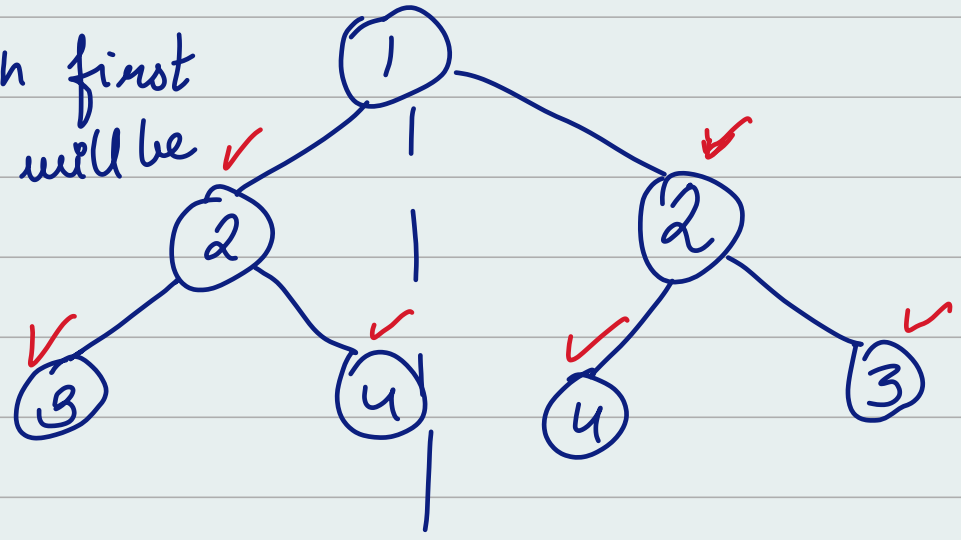
1 == 1 && 2 == 2 && 3 == 3

```
public class SameTree {

    public boolean isSameTree (TreeNode p,
                               TreeNode q ) {
        return helper (p, q);
    }

    private boolean helper (TreeNode p,
                            TreeNode q ) {
```

```
if (p == null && q == null){
    return true;
}
if (p == null || q == null){
    return false;
}
if (p.val != q.val){
    return false;
}
return helper(p.left, q.left) &&
       helper(p.right, q.right);
}
}
```

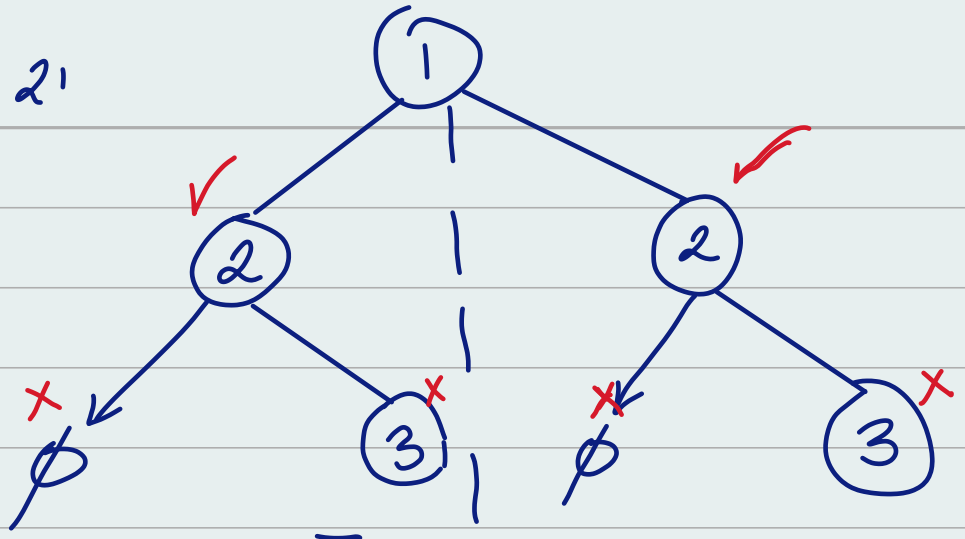
Q3) Symmetric Tree :-

Breadth first search will be used.



Input : root = [1, 2, 2, 3, 4, 4, 3]
output : true

Example 2:

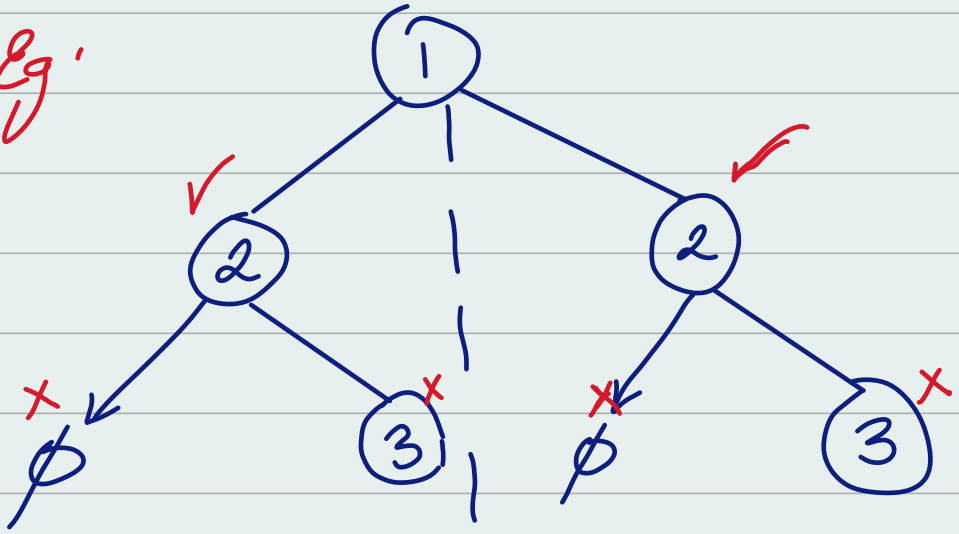


Input: root [1, 2, 2, null, 3, null, 3]
output: false

[1, 2, 3, 4, 4, 3]

To solve this null get left. left first and right right to get elements in order then left right first and right. left first. We won't skip null this time because of assymetric mirror case

Eg:



[l.l
r.r
l.r
r.l]

(i) Use Queue

(ii) Do not add root cause its a single element and it's a mirror of itself so Queue.add(root.left);
and Queue.add(root.right);

(iii) Create a TreeNode to poll left and right.

(iv) check for null values

(v) check for mirror.

(vi) Now add in Queue the l.l, r.r, r.l, l.r in a order

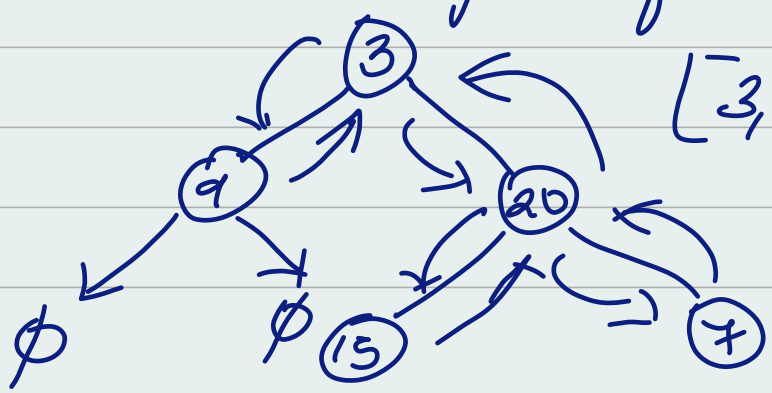
(vii) In the end return true.

Code:

```
public boolean isSymmetric(TreeNode root){
    Queue<TreeNode> queue = new LinkedList<>();

    queue.add(root.left);
    queue.add(root.right);

    while(!queue.isEmpty()){
        TreeNode left = queue.poll();
```

```
TreeNode right = queue.poll();
if ((left == null) && (right == null)){
    continue;
}
if (left == null || right == null){
    return false;
}
if (left.val != right.val){
    return false;
}
queue.add(left.left);
queue.add(right.right);
queue.add(left.right);
queue.add(right.left);
return true;
}
}
```

Q.4) Maximum depth of a binary tree.



[3, 9, 20, ∅, ∅, 15, 7]

```
class DepthBT{
    public int maxDepth(TreeNode root){
        if(root == null){
            return 0;
        }
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);

        int maxDepth = Math.max(left, right) + 1;
        return maxDepth;
    }
}
```

85) Convert Sorted Array to Binary Search Tree

nums = [-10, -3, 0, 5, 9]
(i-2, i-1, i, i+1, i+2)

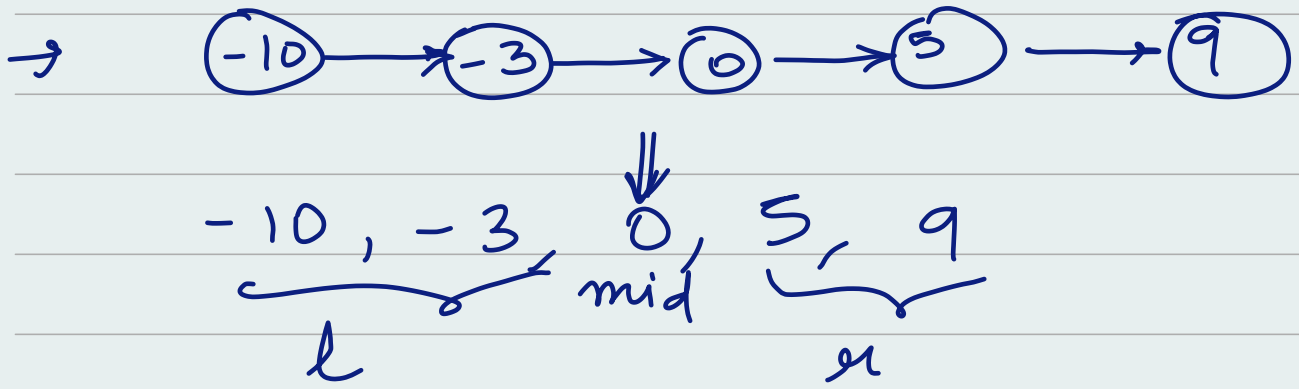
output = [0, -3, 9, -10, ∅, 5, ∅]

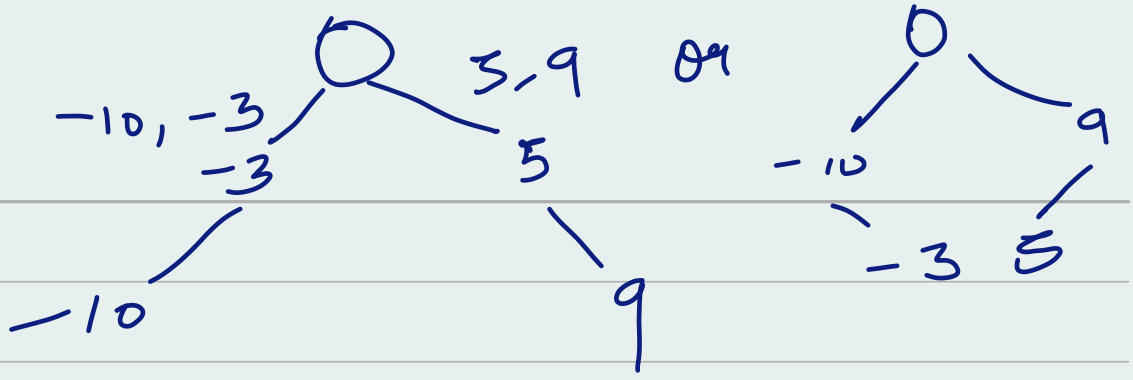
print in sorted order start from 0 till nums.length
// use the binary search method
now create a helper function.
then if start >= end then there is no node to create return null

Now take a middle element that is 0 and insert it creating a new TreeNode.

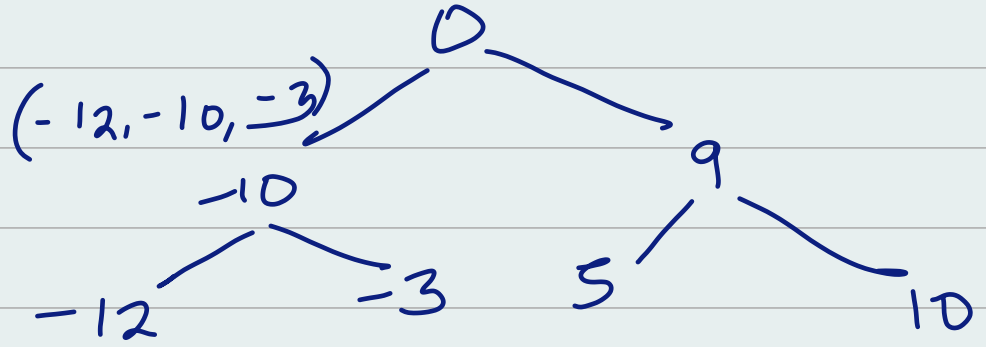
```
int mid = (start + end) / 2;
TreeNode node = new TreeNode
                (nums[mid]);

node.left = sortedArrayToBST(
            nums, start,
            mid);
node.right = sortedArrayToBST(nums, mid+1,
             end);
return node;
}
}
```

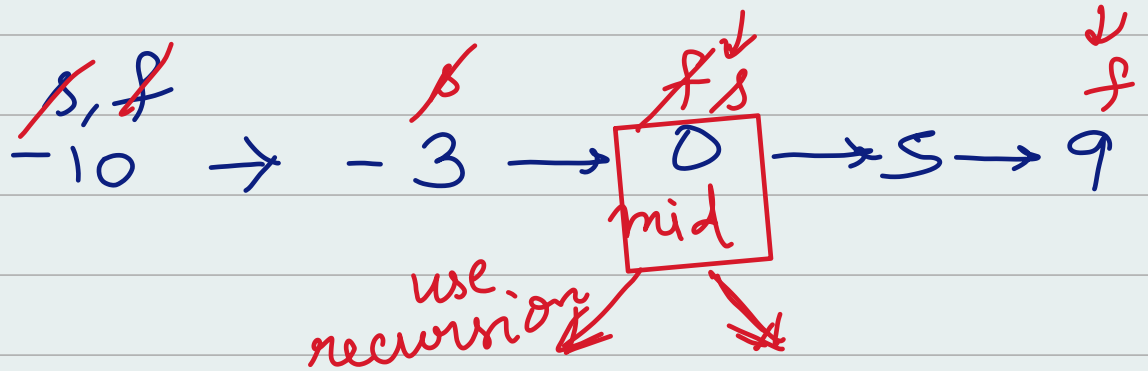
06) Convert Sorted LinkedList to Binary Search Tree (Medium)





Similarly for $-12, -10, -3, 0, 5, 9, 10$





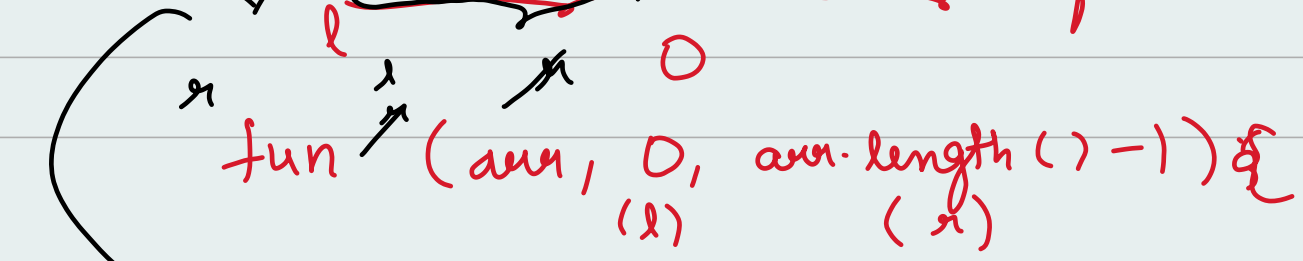
① → ϕ head.next == null
return head.val.

or

-10 → -3 → 0 → 5 → 9

i) first convert the linked list into list;

-10, -3, 0, 5, 9 ; often this



node.val = arr[mid]; → if (l > r) return null

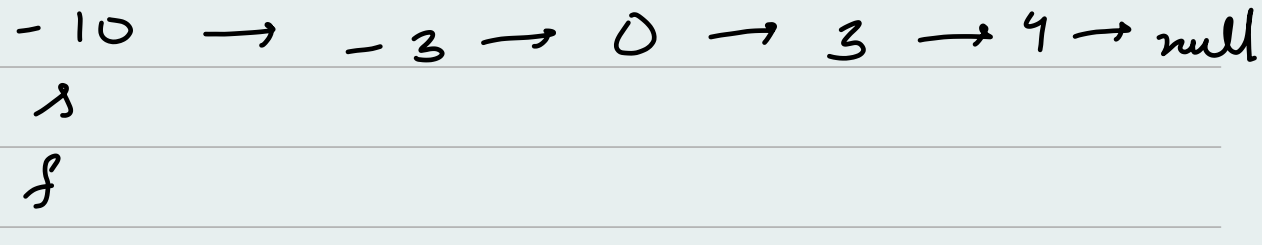
```
node.left = fun(arr, 0, mid-1);
node.right = fun(arr, mid+1, r)
return node;
```

ii) Then store the linked list values into list

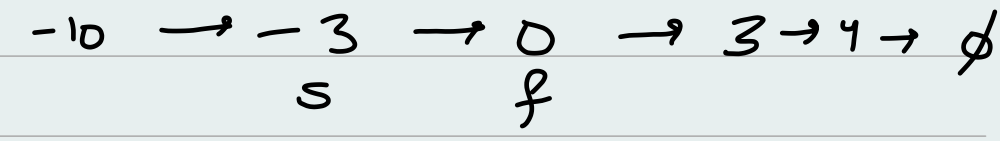
```
while (head != null){
    list.add(head.val);
    head = head.next;
}
```

---

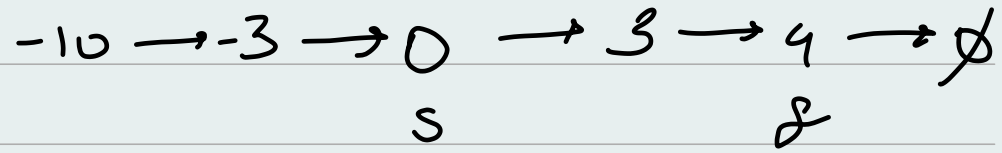
or



1st iteration:



2nd iteration:



Here condition fast != null and fast.next != null

Middle is at zero so create a middle node with value 0

Now the recursive calls will construct the right and left subtrees

left subtree call: sortedListToBSTHelper (head = -10, tail = 0);

-10 → -3 → 0 (stop here because zero is the tail & tail won't be included.)
s
f

1st iteration: -10 → -3 → 0
s f

middle is at slow pointer
create a node that is with value -3

Similarly Right subtree call:

sortedListToBSTHelper (head = 3, tail = null)

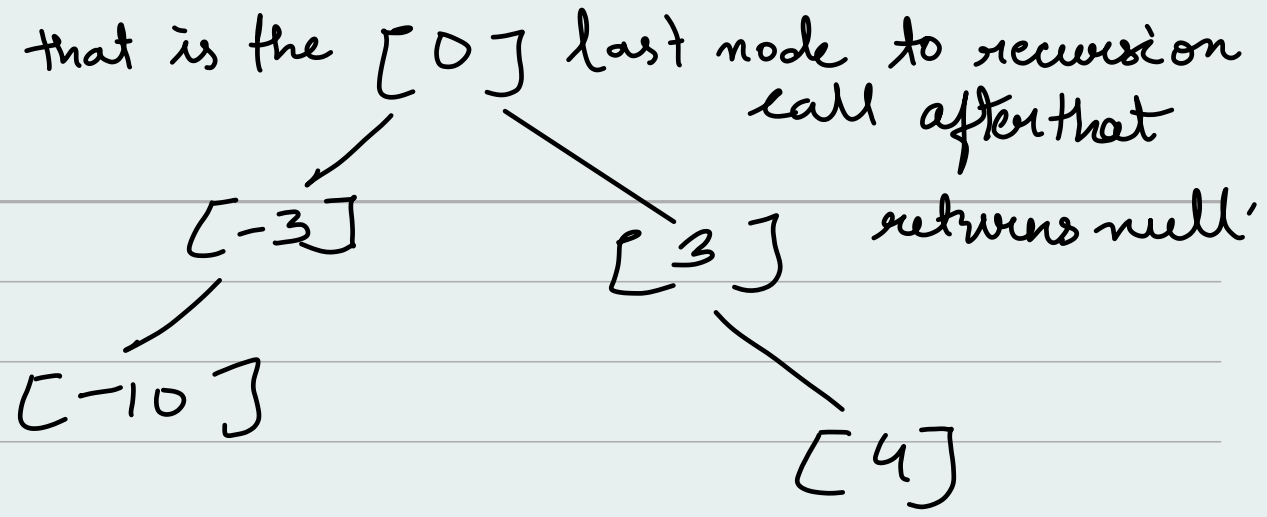
3 → 4 → null
s
f

1st iteration: 3 → 4 → null
s s f f

condition violated fast.next != null
so middle would be 3 and fast pointer would be at 4. Take the middle and create the node and recursion again

Here the base condition would be head equals

to tail that is the [0] last node to recursion call after that returns null.



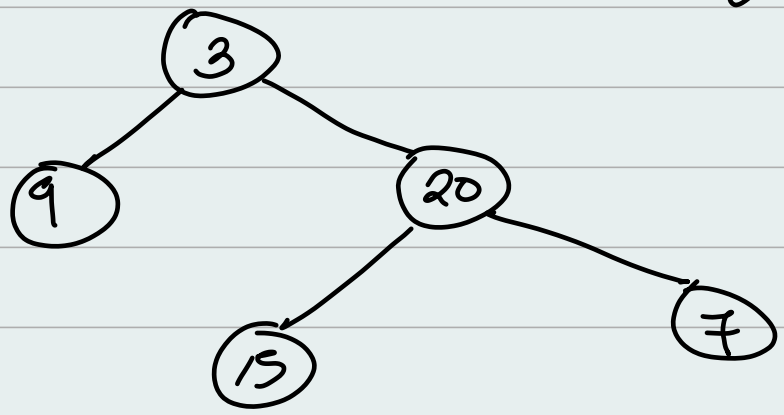
Code!

```
public TreeNode sortedListToBST(ListNode head){
    if (head == null){
        return null;
    }
    return sortedListToBSTHelper(head, null);
} // here head is the start and end is the null. Head is inclusive and tail is exclusive.

private TreeNode sortedListToBSTHelper(ListNode head, ListNode tail){
    if (head == tail){
        return null;
    }
    ListNode slow = head;
    ListNode fast = head;

    while (fast != null && fast.next != null){
        slow = slow.next;
        fast = fast.next.next;
    }
```

```
// at this point, slow is the middle of
    the list.

TreeNode node = new TreeNode (slow.val);

// recursively call for the left side of the
   list

node.left = sortedListToBSTHelper(head,
                                  slow)
node.right = sortedListToBSTHelper(slow.next,
                                   tail);
    return node;
  }
}
```

Q7) Balanced Binary Tree: Determine if it is height-balanced binary tree:

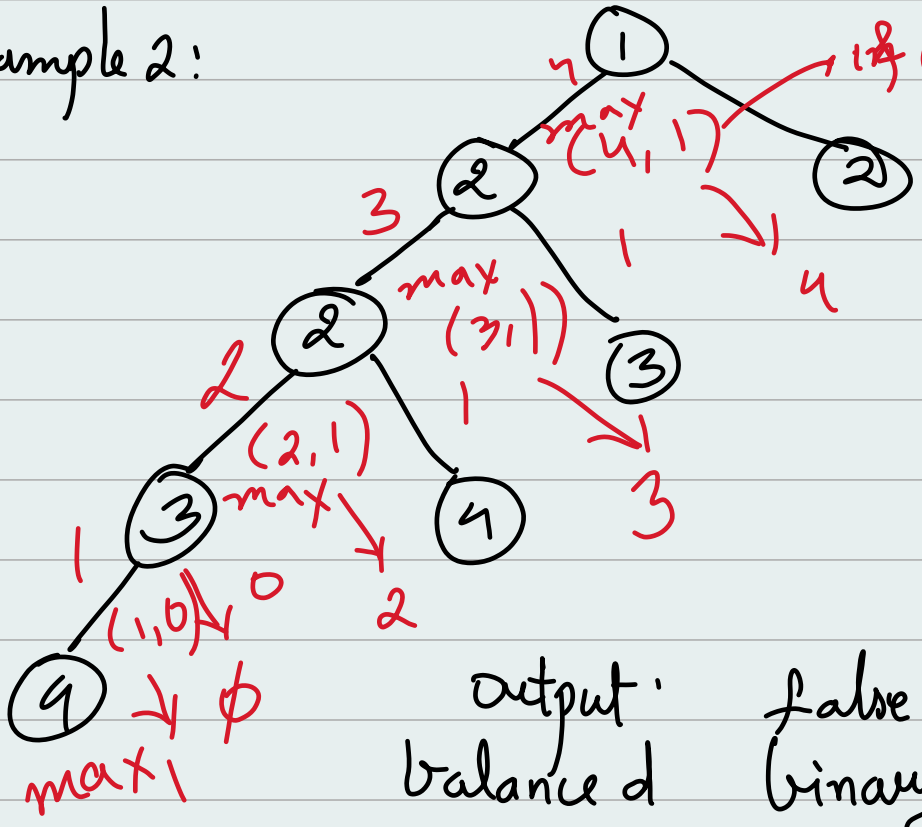
Example 1:



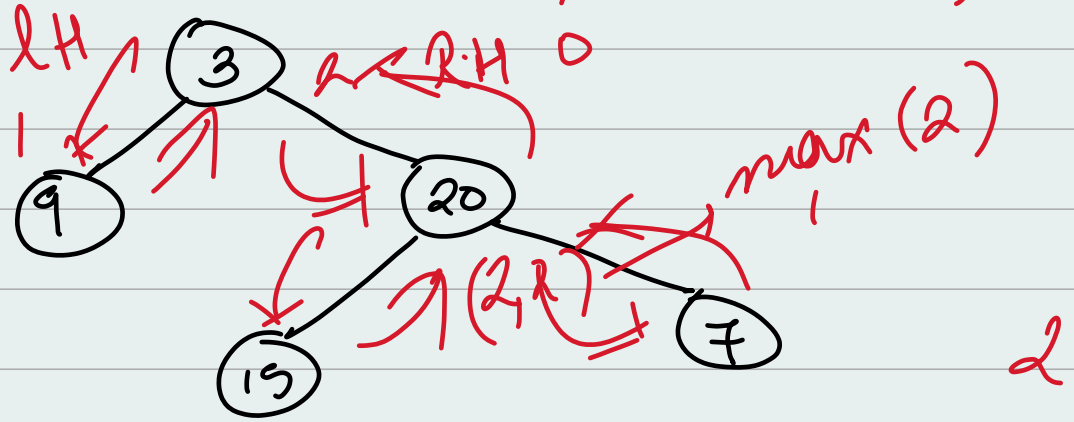
Input : root = [3, 9, 20, null, null, 15, 7]

output = true

Example 2:



output: false, not a balanced binary tree

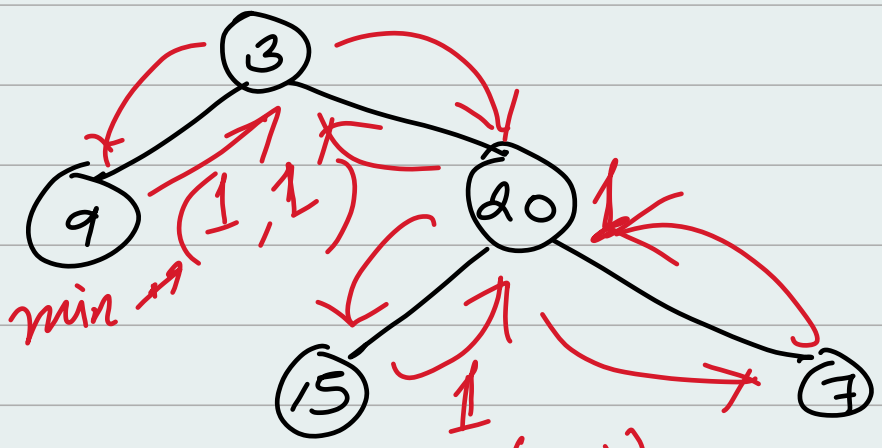



```
boolean func(){
  if (root == null){
      return true;

  return Height(root) != -1 ;
} Here if the Height is -1 then it is not balanced
```

```
int fun ( ){
int leftHeight =  Height ( root.left);
int rightHeight=  Height ( root.right);
if (leftHeight == -1 or rightHeight == -1 ||
        Math.abs (leftHeight - rightHeight)>1)
            return -1;
  Math.max ( leftHeight, rightHeight) +1;
}
```

88) Maximum Depth of a Binary Tree

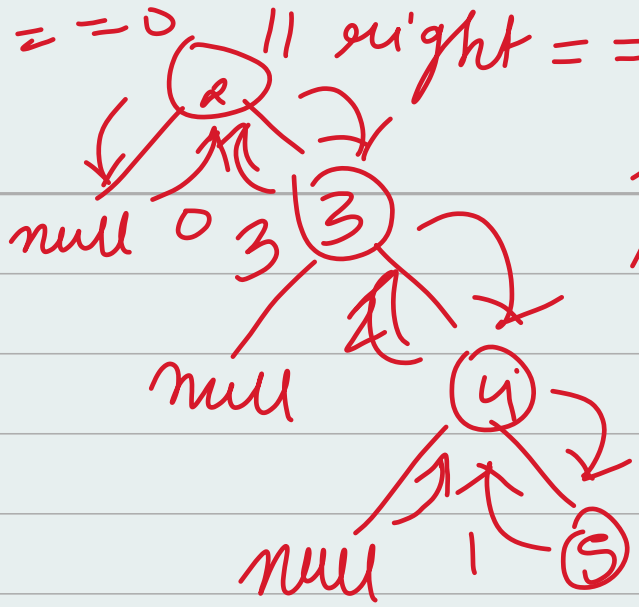
Not so optimal solution using DFS.



min(1,1)

=1+1 =2

if (left == 0 || right == 0) {



return Math.max(left, right)+1

max(1, 0)

max(2, 0)

max(3, 0)

max(3, 0) ⇒ 3 + 1 = 4

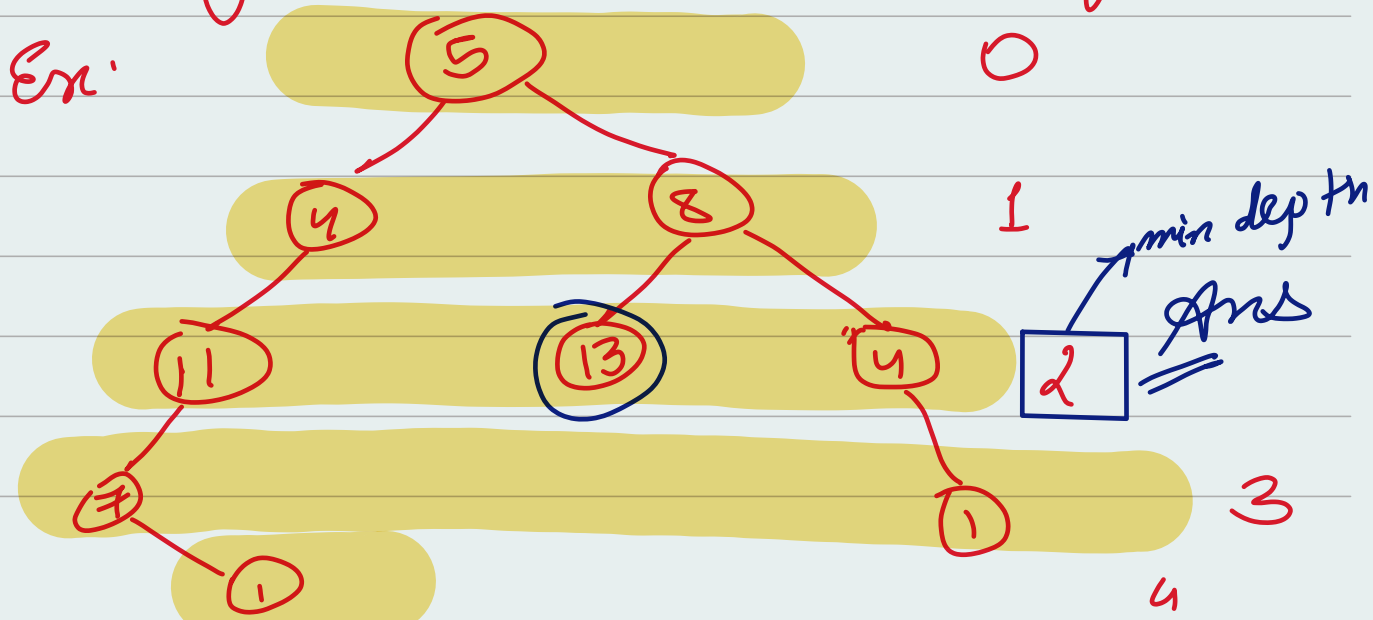
else {return min(left, right)+1

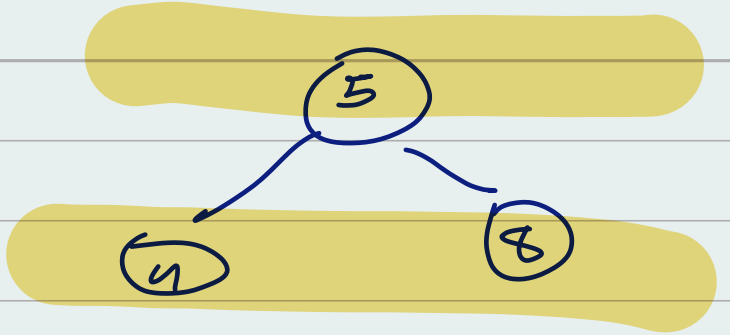
}

---

finding it's minimum Depth

Ex.



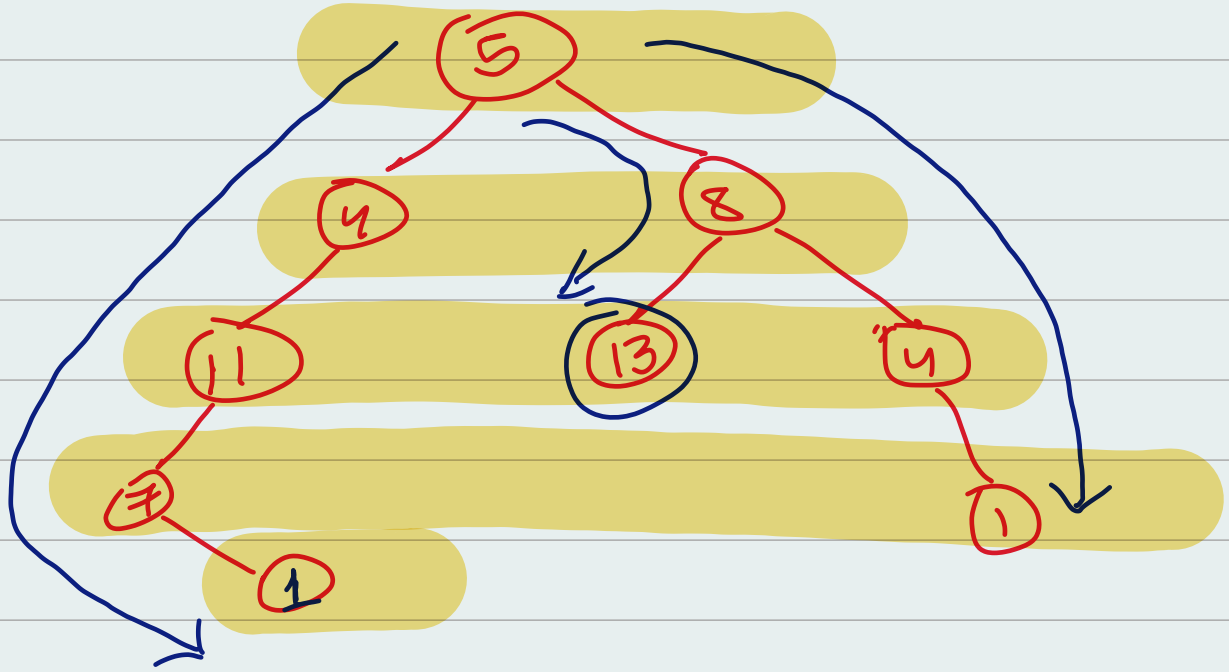


0

1

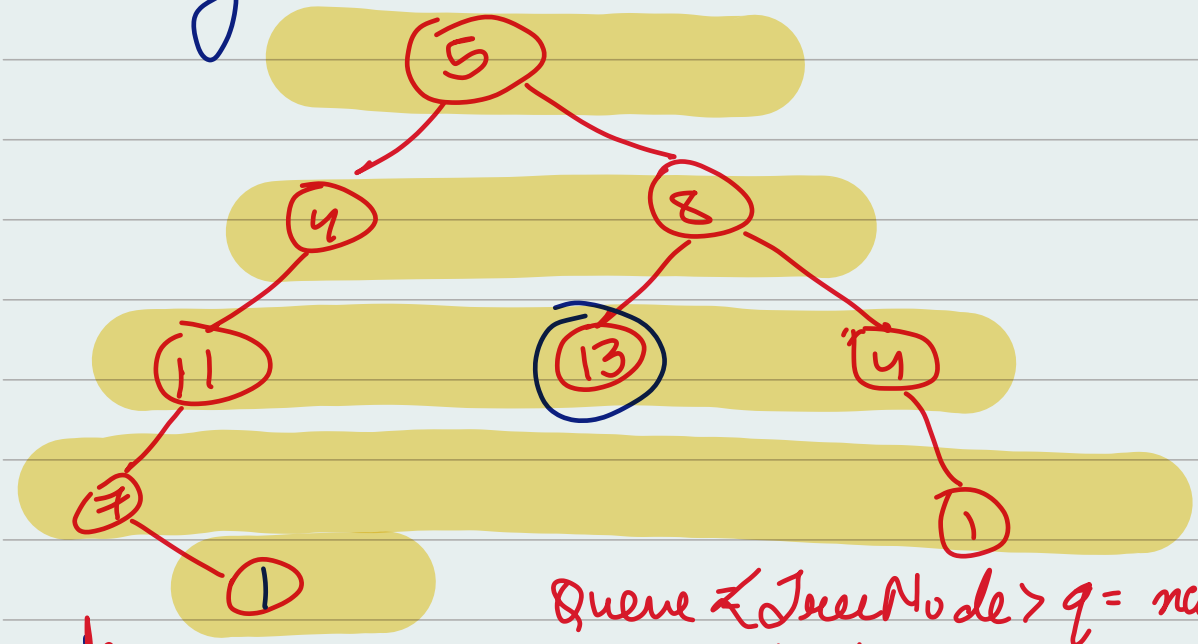
Ans

Brute force solution.



node	depth
1	4
13	2 → Ans
9	3

Using level order Traversal



Queue<TreeNode> q = new LinkedList<>();
int depth = 1; start from root

make a Queue

[5, 4, 8, 11, 13, 4, 7, 1, 1]
leaf node

// level order traversal

while (!queue.isEmpty()){

if a leaf node is found just return the depth.

int size = queue.size();

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

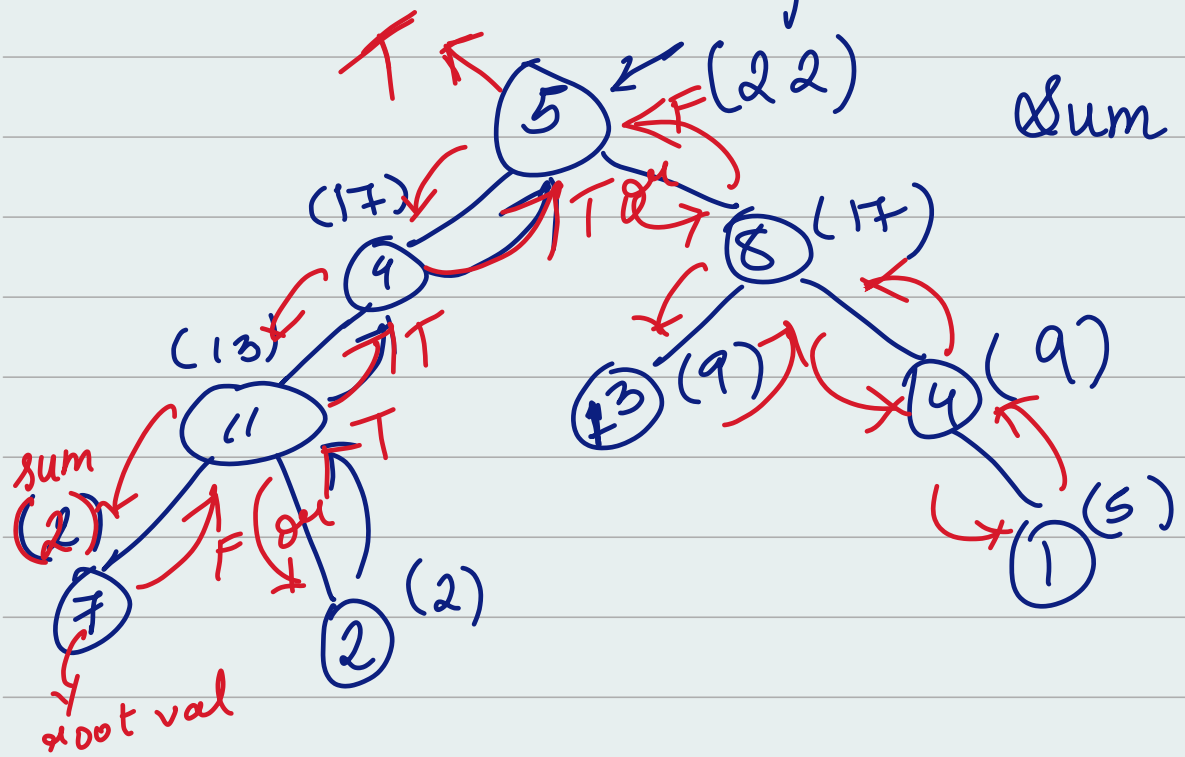
TreeNode node = queue.poll();

if (node.left == null && node.right == null){

return depth;

```
if ( left is not null ){
    queue.offer(node.left);
}
if ( right is not null ){
    queue.offer(node.right);
}
depth++;
return depth;
```

Q9) Path Sum (Depth First Search)

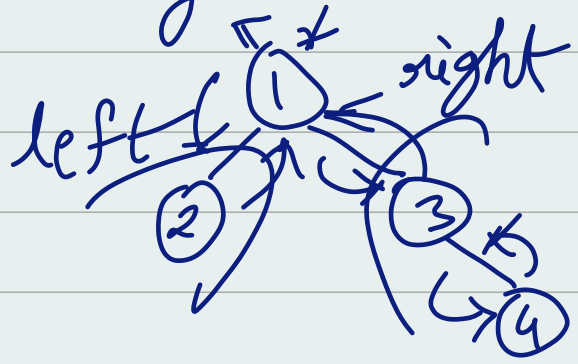


Code:

```
public boolean hasPathSum(TreeNode root,
                          int targetSum){
```

```
if (root == null){
    return false;
}
if (root.val == targetSum &&
    root.left == null && root.right == null){
    return true;
}
return hasPathSum(root.left, targetSum -
root.val) || hasPathSum(root.right, targetSum -
    root.val)
}
}
```

Q10) Binary Tree PreOrder Traversal



PreOrder :
Current Node — left — right

1 → 2 → 3 → 4

root.left root.right

[1, 2, 3, null, null, null, 4]

root — root left — root right

We have to return the preOrder Traversal, this list as the ans

1) first create a new ArrayList. then pass the helper function by root and list

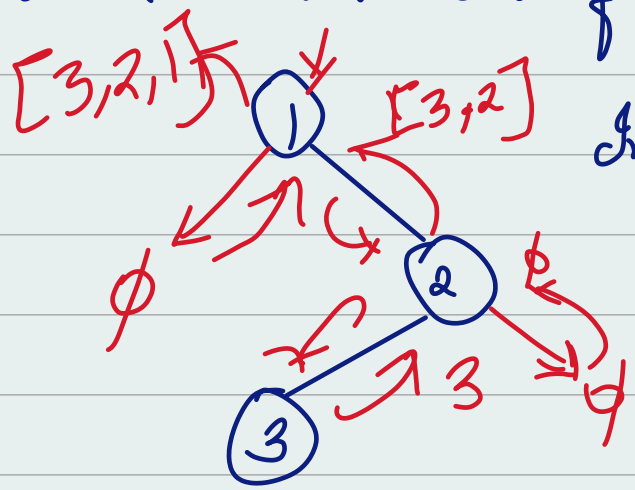
```
List<Integer> list = new ArrayList<>();
    helper(root, list);

    return list;
}
private void helper(TreeNode root,
                    List<Integer> list){
    if(root == null){
        return;
    }
    list.add(root.val);
    helper(root.left, list);
    helper(root.right, list);
}
}
```

---

811) Post order Traversal

→ Given the root of a binary tree return the

post order traversal of it's nodes values.



Input root = [1, null, 2, 3]

output : [3, 2, 1]

postOrderTraversal :

Left - right - current node

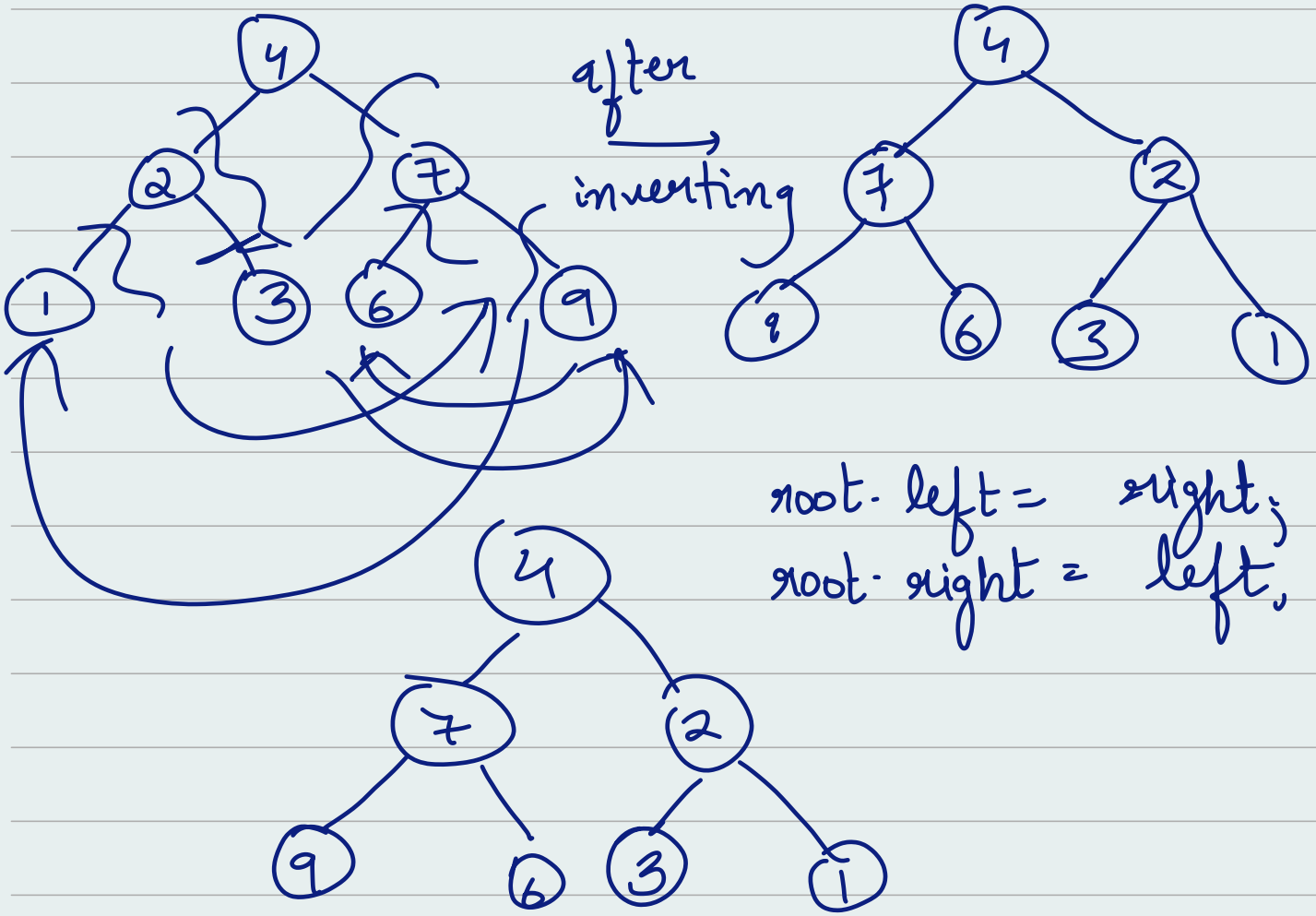
Code :-

```
class Solution {
    public List<Integer> postOrderTraversal(
                                TreeNode root){
        List<Integer> list = new ArrayList<>();
        postOrder(root, list);
        return list;
    }

    private void postOrder(TreeNode root,
                           List<Integer> list){
        if(root == null){
            return;
        }
```

```
postOrder(root.left, list);
postOrder(root.right, list);
list.add(root.val);
}
}
```

Q 12) Invert a binary tree.

Given the root of a binary tree, invert the tree and return its root.




```
root.left = right;
root.right = left;
```


Here call TreeNode left = invertTree
Because swap the internal nodes first.(root.left);

```
        TreeNode right = invertTree
                         (root.right);
    }
class Solution {
    public TreeNode invertTree (TreeNode
                                root) {
        if (root == null) {
            return null;
        }
        TreeNode left = invertTree (root.left);
        TreeNode right = invertTree (root.right);

        // Swap the nodes.

        root.left = right;
        root.right = left;
        return root;
    }
}
```

---

13) Binary Tree Paths (DFS best solution)

Given the root of a binary tree return root-to-leafs paths in any order

Input root = [1, 2, 3, null, 5]

Output = [1 → 2 → 5 , 1 → 3]

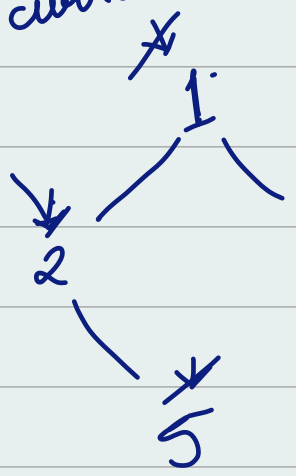
create a list of string

List < String > result = new ArrayList <> ();

if root == null return null;

create (1) → root ["1 → 2 → 5", "1 → 3"]

current_path



"1 → 2 → 5"

string current_path =
Integer.toString(root.val);
// 1 is added to the current path

// Now check for left & right null value if it has it means it is root node.

```
if (root.left == null && root.right == null){
    result.add (current_path);
}
if (root.left != null){
    dfs(root.left, current_path,
                        result);
}
if (root.right != null){
    dfs(root.right, current_path, result);
}
return result;
}
```

```
public void dfs(TreeNode node,
                String current_path,
                List<String> result){

    current_path = current_path + "->" +
                   node.val );

    if( node.left==null && node.right==null){
        result.add ( current_path);
        return;
    }
    if (node.left!=null)
        dfs( node.left , current_path,
                                result);

    }
    if(node.right!=null){
        dfs(node.right, current_path, result);
    }

}

// More optimized code

depthFirstSearch (node = 1, path = "", ans = [])
path = "1"
```

DFS (node = 2, path = "1 → ", ans [ ])

path = "1 → 2"

DFS (node = 5, path = "1 → 2 → ", ans [ ])

path = "1 → 2 → 5"

(leaf node, add "1 → 2 → 5" to ans)

Backtrack: path = "1 → 2"

BackTrack: path = "1"

DFS (node = 3, path = "1 → ", ans = ["1 → 2 → 5"])

path = "1 → 3"

(leaf node, add "1 → 3" to ans)

Backtrack: path = "1"

Code:

```
class Solution {
    public List<String> answer = new
                                ArrayList<>();
```

```
DFS( root, new StringBuilder(), answer),

    return result;

}

private void DFS(TreeNode node, StringBuilder
                        path, List<String> answer){

    if(node == null){
        return;
    }
    int length = path.length;
    if(length > 0){
        path.append("->");
    }
    path.append(node.val);

    if(node.left == null && node.right ==
                                    null){
        answere.add(path.toString());
    }
    else{
        dfs(node.left, path, answer);
        dfs(node.right, path, answer);
    }
```

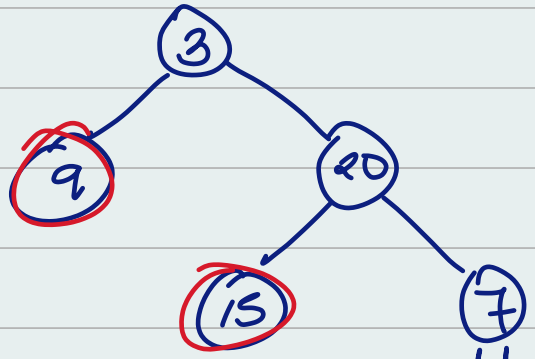
path.setLength(length);//

} Backtrack by resetting the StringBuilder

Q14) Sum of left leaves

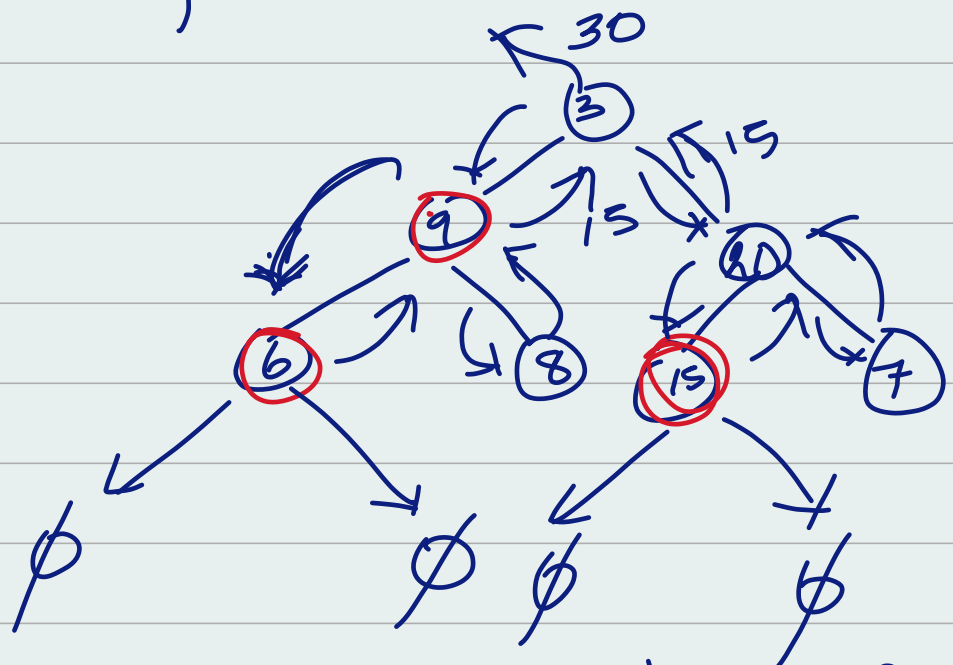
Given the root of a binary tree, return the sum of all left leaves

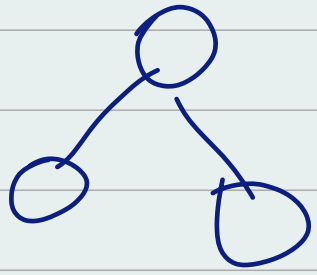
Example:



root = [3, 9, 20, null, null, 15, 7]

output = 24





left - root - right

By using stack & By using queue

Code:

```
class Solution {

    public int sumOfLeftLeaves (TreeNode root) {

        if (root == null) {
            return 0;
        }

        int sum_of_left_leaves = 0;

        Stack <TreeNode > stack = new
                                  Stack<>();

        stack.add (root);

        while (! stack.isEmpty()) {
            TreeNode node = stack.pop ();

            if (node.left != null ) {
                if (node.left.left == null &&
                    node.left.right == null ) {
                sum_of_left_leaves += node.left.val;
                } else {
                    stack.add ( node.left );
                }
```

```
            if(node.right != null){
                if(node.right.left != null ||
                    node.right.right != null){
                    stack.add(node.right);
                }
            }
        }
        return sum_of_left_leaves;
    }

// More efficient solution using queue
    public int sumOfLeftLeaves(TreeNode root){
        if(root == null){
            return 0;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int sumLeftLeaves = 0;
        while(!queue.isEmpty()){
```

```
TreeNode node = queue.poll();
if(node.left != null){
if(node.left.left == null &&
   node.left.right == null){
    sumLeftLeaves += node.left.val;
}
else{
    queue.offer(node.left);
}
}
if(node.right != null){
    queue.offer(node.right);
}
}
return sumLeftLeaves;
}
```

// Now by using Recursion.

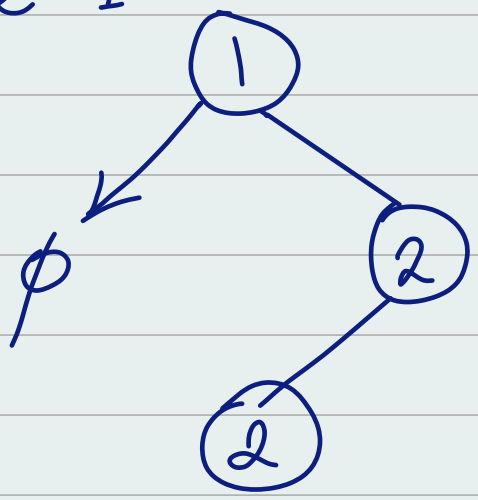
```
public int sumOfLeftLeaves(TreeNode
                           root){
```

```
if (root == null){
    return 0;
}
int sum_of_left_leaves = 0;

if (root.left != null){
    if (root.left.left == null &&
        root.left.right == null){
        sum_of_left_leaves += root.left
                              .val;
    }else{
        sum_of_left_leaves +=
            sumOfLeftLeaves
            (root.left);
    }
}
if (root.right != null){
    if (root.right.left != null ||
        root.right.right != null){
        sum_of_left_leaves +=
            sumOfLeftLeaves(
                root.right);
    }
}
return sum_of_left_leaves;
}
}
```

Q15) Find mode in Binary Search Tree

→ Given the root of a Binary Search tree (BST) with duplicates, return all the modes (i.e frequently occured element) in it.

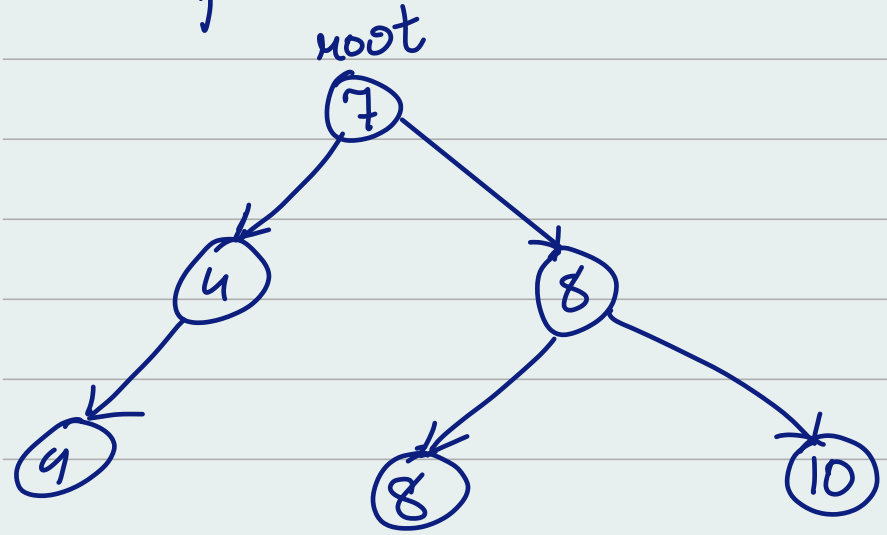
Example 1:



root = [1, null, 2, 2]

output = 2

Example 2:



currentNum = 0
currentFreq = 0
maxFreq = 0
result = { }

currentNum = 4
currentFreq = 1
maxFreq = 1
result = { 4 }

{4, 4, 7, 8, 8, 10}

if current frequency is equal to max frequency then update the result with the currentNum = 4

Here currentfreq is greater than maxfreq update the maxfreq to 2. reset the result.

currentNum = 4
currentfreq = ~~1~~ 2
maxfreq = ~~1~~ 2
result = {4}

currentNum = is currentNum equal to num, no, update (7)
currentfreq = 1
maxfreq = 2 {Here currentfreq is less than maxfreq no updation}
result = {4} → no updation as maxfreq is not updated.

Here currentfreq is not equal to maxfrequency than do not update the result

currentNum = 8
currentfreq = 1
maxfreq = 2
result = {4}

currentNum = 8
currentfreq = 2
maxfreq = 2
result = {4, 8}

Here current freq is equal to maxfreq update the result with 8.

currentNum = 10
currentfreq = 1
maxfreq = 2
result = [4, 8]

Ans

Here Main three conditions

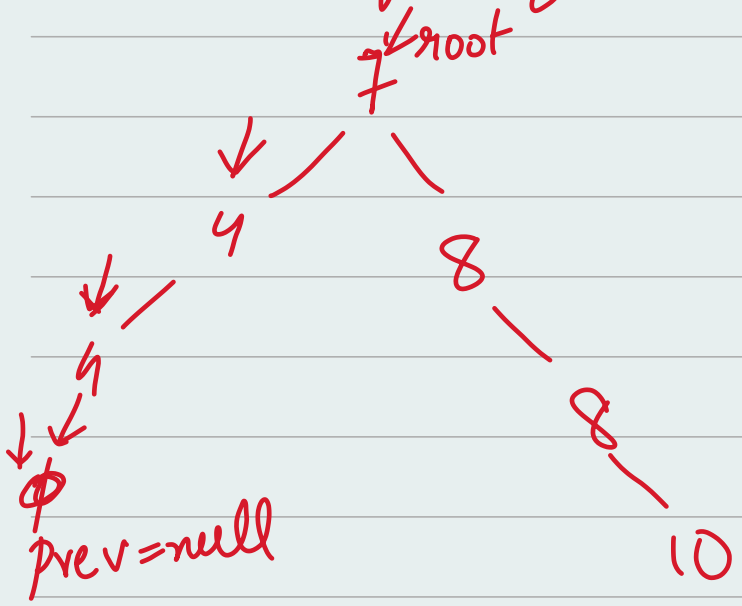
① Is the number in the root/list equal to currentNum than update the current frequency if not currentNum update with new value & update freq i

② if currentfrequency > maxfrequency update the maxfreq to currfreq reset the result

③ if currentfrequency == maxfreq update the result with currentNum

if currentfreq >= maxfreq
update the maxfreq
and update the result

// More optimized code



TreeNode prev = null // to track the previous node in-order

int maxFreq = 0;
int currentFreq = 0;
int modeCount = 0; // to keep track of how many mode we have.
int[] modes;

→ Start at the root (7), since it has a left child so move the left child (4).

→ At left child 4, there is another left child 4. So move it to 4

→ At node 4 the prev is null.

So process it - first node is 4 so current Freq is 1 MaxFrequency is updated to 1.

```
Tree Node prev = null;
int currentFreq = 1
int maxFreq = 1
```

So move up to parent 4, it's the same value so currentFrequency is incremented to 2 maxfreq is updated to 2.

```
Tree Node prev = 4;
int currentFreq = 2
int maxFreq = 2.
```

equals modeCount increment to 1

prev

4, 4, 7, 8, 8, 10

```
int currentFreq = 1   (reset for 7)
int maxFreq = 2
```

```
int currentFreq = 1   (reset for 8)
int maxfreq = 2
```

int currentFreq = 2, (again &)
int maxFreq = 2 } equals

So mode count is incremented to 2.

int currentFreq = 1 (reset to 1 0)
int maxFreq = 2

After the first traversal we found that the maxFreq is 2 and there are 2 numbers that occurs with this frequency ('mode count' is 2).

Resetting for Second Traversal.

Now we reset prev, currentFreq and mode count to prepare for the second traversal

{ 4, 4, 7, 8, 8, 10 }
↑ prev
int currentFreq = ✗ 2
int maxFreq = ✓ 2

Since currentFreq is equal to maxFreq we add 4 to the mode array.

int currentFreq = 1 [reset at 7]
int maxFreq = 2

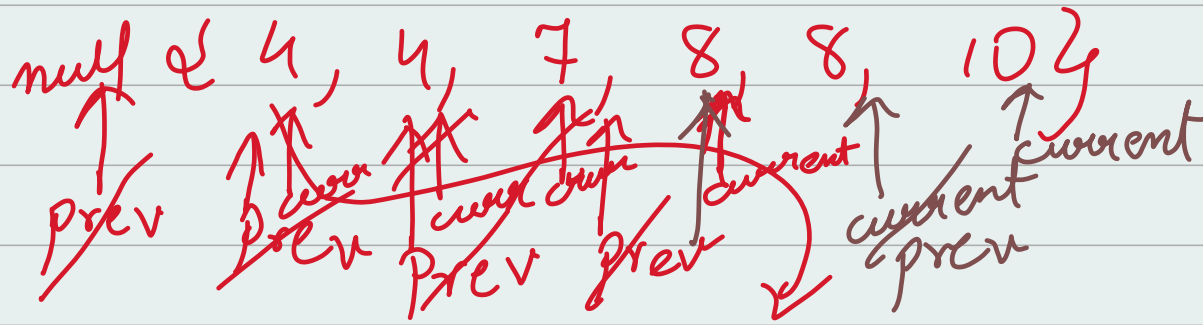
Int currentFreq = 2 [at second 8]
int maxFreq = 2 [equals

add 8 to the mode array

int currentFreq = 1 [reset at 10]
int maxFreq = 2

at the end of the second traversal our mode array would have [4, 8].

→ Here prev pointer working



int currentFreq = 1 [at 4]
int maxFreq = Zero 1

Since prev. val is same as current-val incrementing the currentFreq to 2. As currentFreq > maxFreq the maxFreq = currentFreq So maxFreq would also be 2.

As prev pointing to 4 and current 7 reset the currentFreq to 1 again but maxFreq will remain same.

Here as current and prev val again got same at 8 so int currentFreq is 2 and maxFrequency = 2.

It will again reset to 1 after the last iteration cause prev will point to 8 and current at 10.

When found a new maxFrequency (i.e when currentFreq > maxFreq) update

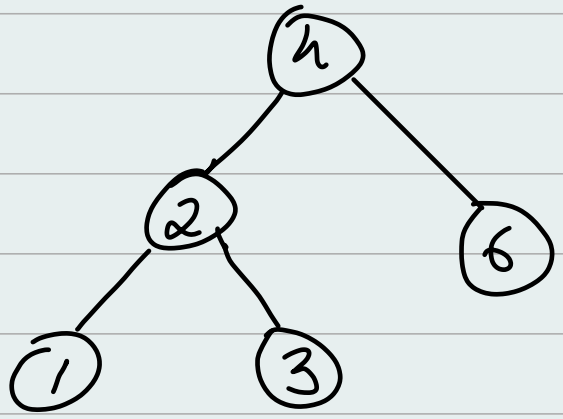
the maxFreq and reset the mode count to 1.

```
class Solution{

    private TreeNode prev = null,

    private TreeNode currentFreq = 0;

    private TreeNode maxFreq = 0,

    private int modeCount = 0;

    private int[] modes,

    public int[] findModes(TreeNode root){

        //first inOrder traversal for modeCount.

        inOrder(root, true); // true means
          we are counting the mode count

        modes = new int[modeCount];

        modeCount = 0,
```

```
        currentFreq = 0;
        prev = null

    // Second inOrder traversal to find modes
    inOrder(root, false) // false means we are finding the mode
    return nodes;
}

private void inOrder(TreeNode node,
                     boolean findMaxFreq){
    if (node == null){
        return;
    }

    inOrder(node.left, findMaxFreq);

    if (prev != null && prev.val == node.val){
        currentFreq++;
    } else{
        currentFreq = 1;
    }
```

```
if(currentFreq > maxFreq){
        maxFreq = currentFreq;
        modeCount = 1;
} else if (currentFreq == maxFreq){
// if currentFreq is eql to maxFreq then add the element to findMaxFreq.
        if (findMaxFreq){
            modeCount++;
        } else{
            modes[modeCount++] = nodeval;
        }
}

prev = node; //update the prev to current node before moving to right subtree.

inOrder(node.right, findMaxFreq);

}
```

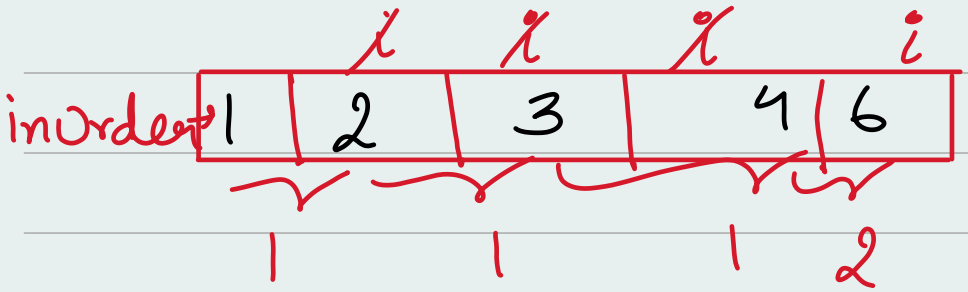
816) Minimum absolute difference in a BST.



root = [4, 2, 6, 1, 3]

output = 1.

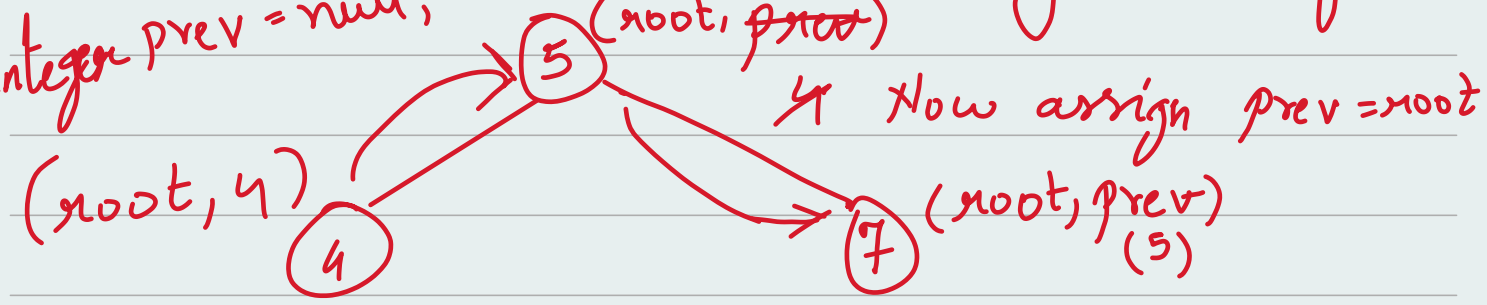
BST → inOrder → sorted



Here the arrays are sorted so adjacent element will give the smallest abs difference.

It can be done without using extra space.

Integer prev = null;



↳ Now assign prev = root

(In root right it's previous is root so update prev = root.val;

Assign a global min difference as Integer.MAX_VALUE

and also assign Integer prev = null.

so that if any prev is changed it should be reflected.

Code:

```
class Solution{
    int minDifference = Integer.MAX_VALUE;
    Integer prev = null;
    public int getMinDiff(TreeNode root){
        if(root == null){
            return minDifference;
        }
        getMinDiff(root.left);
        if(prev != null){
            minDifference = Math.min(minDifference,
                                     root.val - prev);
```

```
        }
            prev = root.val;
            getminDiff(root.right);
            return minDifference;
        }
    }
```

Q17) Diameter of a Binary Tree.