Trees

If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**

Tree Traversals

1. Inorder

2. Preorder

3. Postorder

https://www.interviewbit.com/snippet/b79072e1cb4741cce7da/

Main Logic: To implement the tree traversal (preorder, inorder, postorder) using a simple recursive approach, follow these steps:

> **Base Case:** Check if the current node is null. If it is, return immediately.
> **Preorder:** Print or add node.val to the result list to achieve a preorder traversal.
> Traverse Left Subtree: Recursively call the same function on the left child of the current node, passing node.left.
> **Inorder:** Print or add node.val to the result list to achieve an inorder traversal.
> Traverse Right Subtree: Recursively call the same function on the right child of the current node, passing node.right.
> **Postorder:** Print or add node.val to the result list to achieve a postorder traversal.

By following this recursive logic, you can perform preorder, inorder, and postorder traversals of a binary tree.

4. Lavel order Traversals (BFS)

https://www.interviewbit.com/snippet/d86e9904184ac63d41eb/

Main Logic ->

```
// Base logic we will use BFS alog here
    -   Add node in qu
Step1: Qusize while ->
        - remove first node
        - print or add node
        - for loop till qu size
        - add left node of removed node
        - add right node of remved node
```

Steps as follows

Initialize an empty ArrayList of ArrayLists list to store the nodes at each level of the binary tree.

Initialize a Deque (double-ended queue) qu as an ArrayDeque. This queue will be used to perform the level-order traversal.

Add the root node of the binary tree to the queue qu using qu.add(node).

Enter a while loop that continues as long as the queue qu is not empty.

Within each iteration of the loop:

- Create an empty ArrayList listItem to store the nodes at the current level.
- Get the current size of the queue qu using int size = qu.size(). This represents the number of nodes at the current level.

Iterate from i = 0 to i < size to process all nodes at the current level:

- Remove the first node from the queue qu using TreeNode remNode = qu.remove(). This node is removed from the front of the queue.
- Add the value of the removed node (remNode.val) to the listItem ArrayList.
- If the left child of remNode exists (remNode.left != null), add it to the queue qu to process later.

- If the right child of remNode exists (remNode.right != null), add it to the queue qu to process later.

After processing all nodes at the current level, add the listItem ArrayList to the list ArrayList. This represents all nodes at the current level.

Repeat steps 5-7 until all levels of the binary tree have been processed.

Once the while loop finishes, return the list, which contains the nodes of the binary tree arranged by levels.

In summary, the code performs a breadth-first traversal (level-order traversal) of the binary tree using a queue. It collects nodes at each level and stores them in an ArrayList of ArrayLists (list).

5. Right View Traversals

6. Left View Traversals

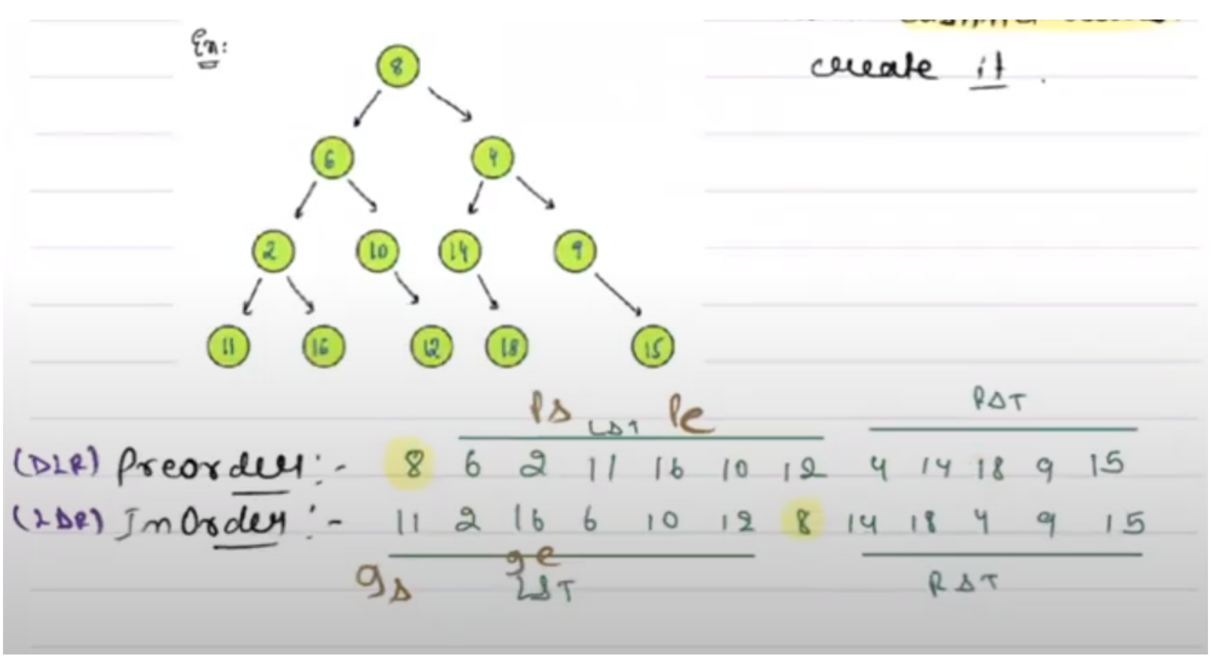
https://www.interviewbit.com/snippet/21720e19ff0648564313/

Main Logic: To perform both **right view** and **left view** traversals of a binary tree, we can follow the same breadth-first search (BFS) logic with just two modifications:

1. Replace the main ArrayList of ArrayLists with a single ArrayList.
2. Instead of inserting the entire ArrayList into the queue while loop, for the right view, take the last element from the listItem and add it to the result list. For the left view, take the first element of the listItem and add it to the result list.

By applying these changes, we can efficiently obtain both the right view and left view of the binary tree during traversal.

- ## Given Preorder & Inorder of a Binary tree with distinct values create it.
  https://www.interviewbit.com/snippet/bd1cc5d314a6f28fa121/



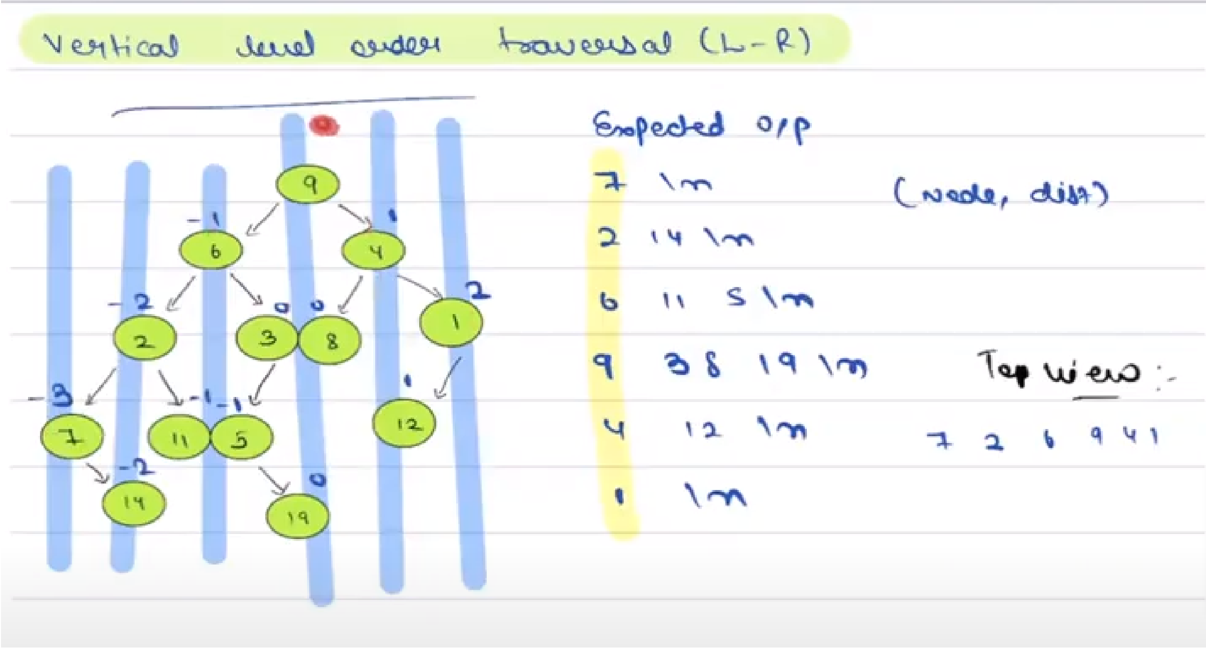
Ex:
create it.
8
6
4
2
10
14
9
11
16
12
18
15
ps
LST
pe
RST
(DLR) Preorder :- 8 6 2 11 16 10 12 4 14 18 9 15
(LDR) InOrder :- 11 2 16 6 10 12 8 14 18 4 9 15
is
ie
LST
RST

- **Main Logic:**
  Naming convention ->
  preorder start (**pst**) and end (**ped**)
  inorder start (**ist**) and end (**ied**)

- To create a binary tree from given preorder and inorder traversal arrays, follow these steps:
- **Base Case:** Check if the given segments (preorder and inorder) of the arrays are empty or out of bounds. If they are, return null as there are no nodes to create. This can be determined by checking if ped > pst and ist > ied. If either condition is met, return null.
- **Select Root Node:** Create a new TreeNode with the value from the first element of the preorder (pst) traversal. This represents the root node.
- Find Root in Inorder: Iterate through the inorder traversal array to find the index where the root value occurs. This index divides the inorder array into left and right subtrees.
- **Calculate Left Subtree Size:** Determine the size of the left subtree by subtracting the starting index of the inorder (ist) array from the root index. You can calculate it as count = nodeIndex - ist.
- **Create Left Subtree:** Recursively call the createTree function for the left subtree by adjusting the indices accordingly. Pass these variables to the recursive call: pre[], in[], pst + 1, pst + count, ist, nodeIndex - 1.
- **Create Right Subtree:** Recursively call the createTree function for the right subtree by adjusting the indices accordingly. Pass these variables to the recursive call: pre[], in[], pst + count + 1, ped, nodeIndex + 1, ied.
- **Return Root Node:** Return the root node of the tree.

This logical breakdown and naming convention should make it clearer how the binary tree is constructed from the preorder and inorder traversals.

Vertical Lable order traversal (LR)



Vertical level order traversal (L-R)

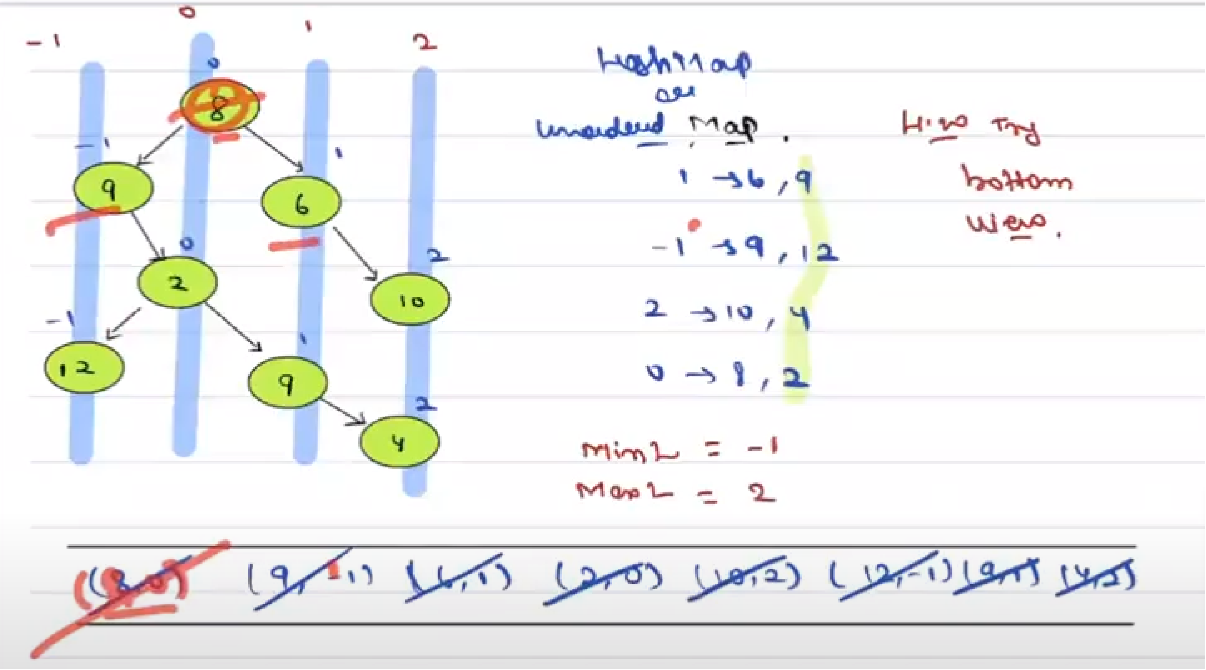
Expected o/p

```
7 \n
2 14 \n
6 11 5 \n
9 3 8 19 \n
4 12 \n
1 \n
```

(node, dist)

Top view :-

7 2 6 9 4 1

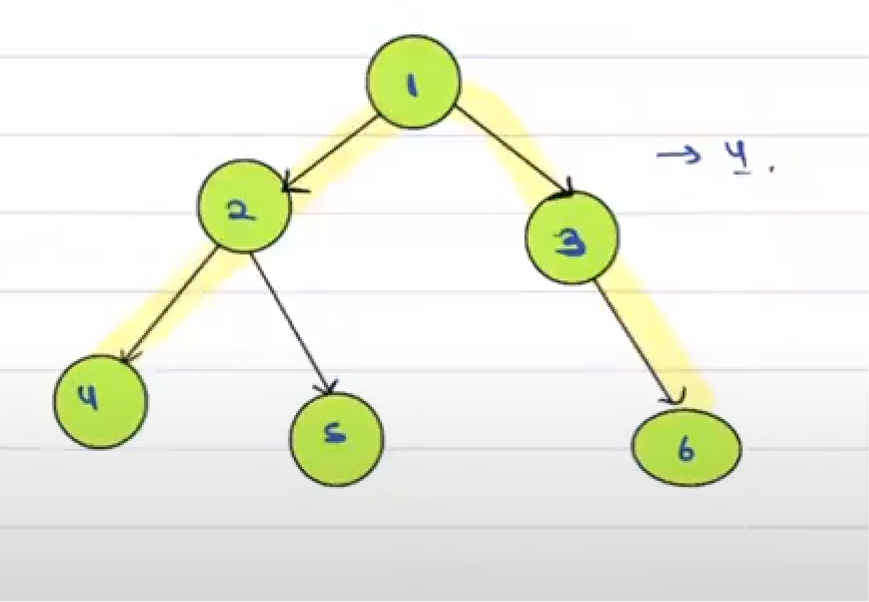


Top view traversal

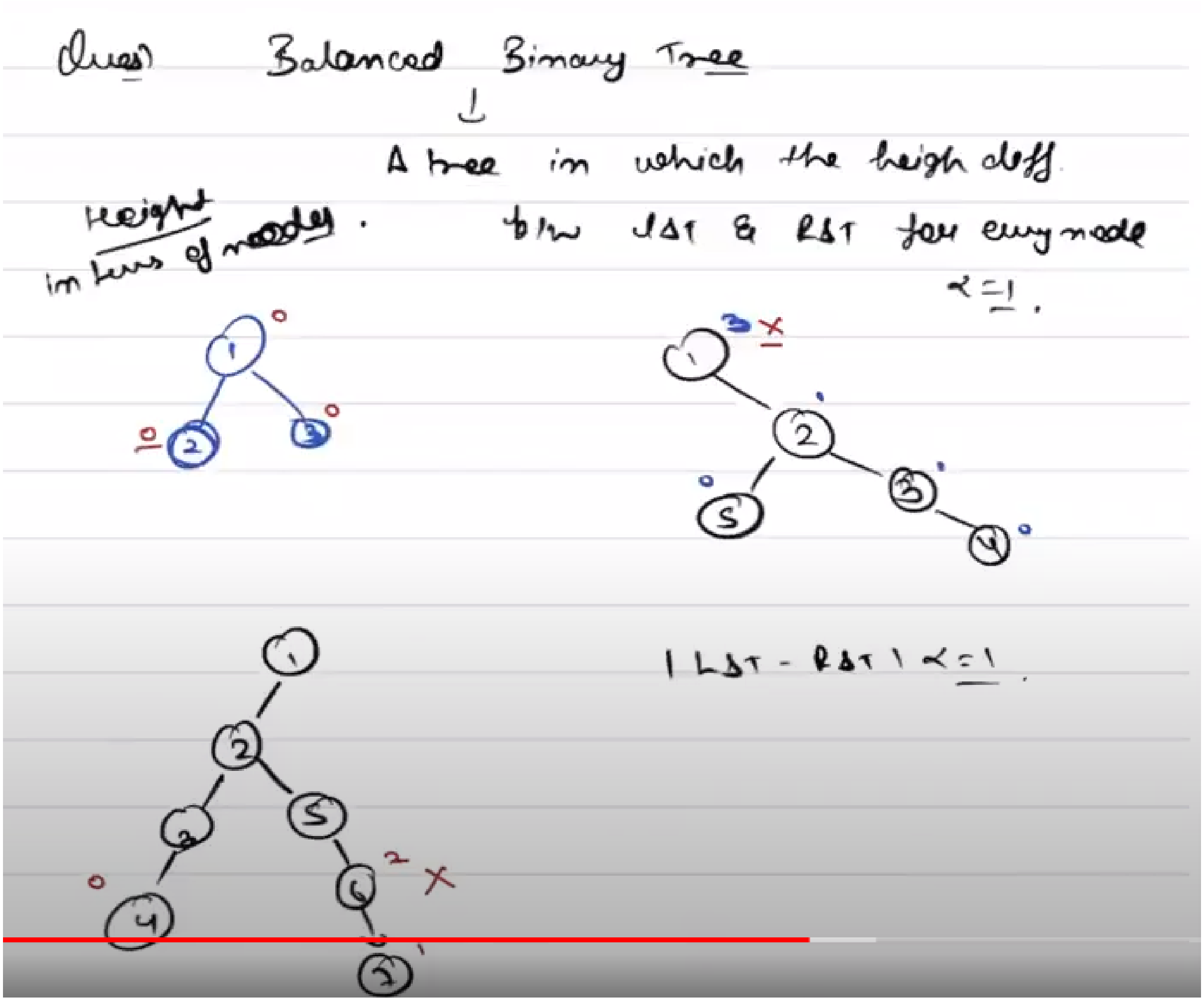
Type of binary tree

- Proper binary tree which has 0 or 2 children no odd numbers children.
- Complete binary tree -> Every lavel is completely filled except the last lavel but nodes in last lavel are filled left to right.
- Perfect binary tree -> all the label filed completely

Diameter (size) of tree -> It is number of edges on longest path b/w nodes in a binary tree.



Balanced Binary tree



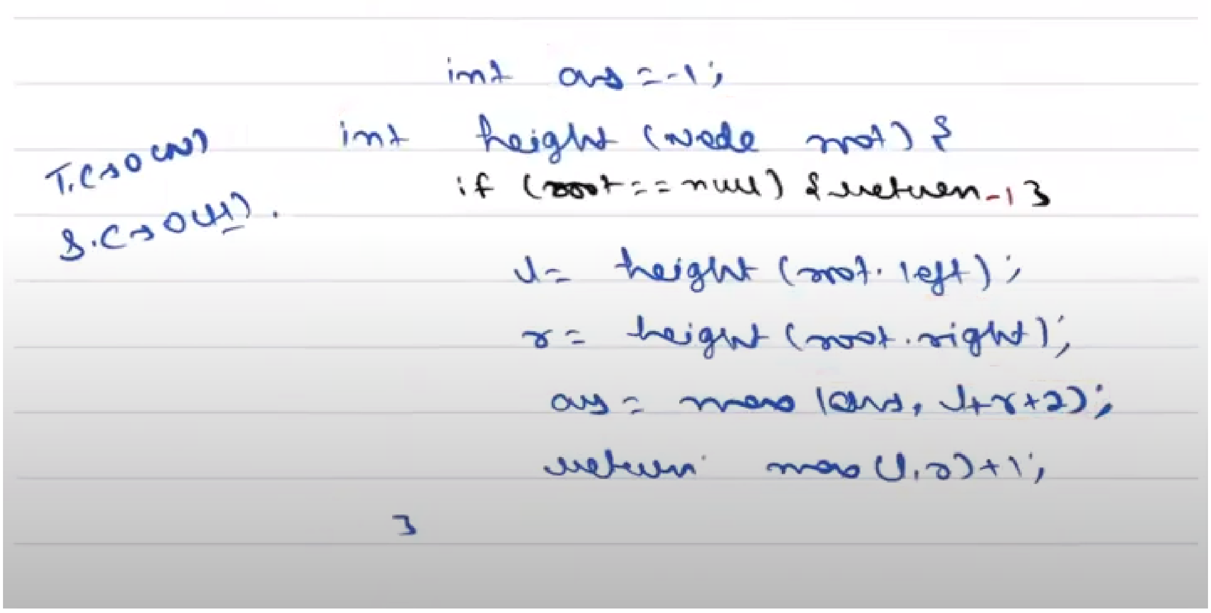
Ques) Balanced Binary Tree

↓

A tree in which the heigh diff. b/w LST & RST for every node <= 1.

Height in terms of nodes.

|LST - RST| <= 1.



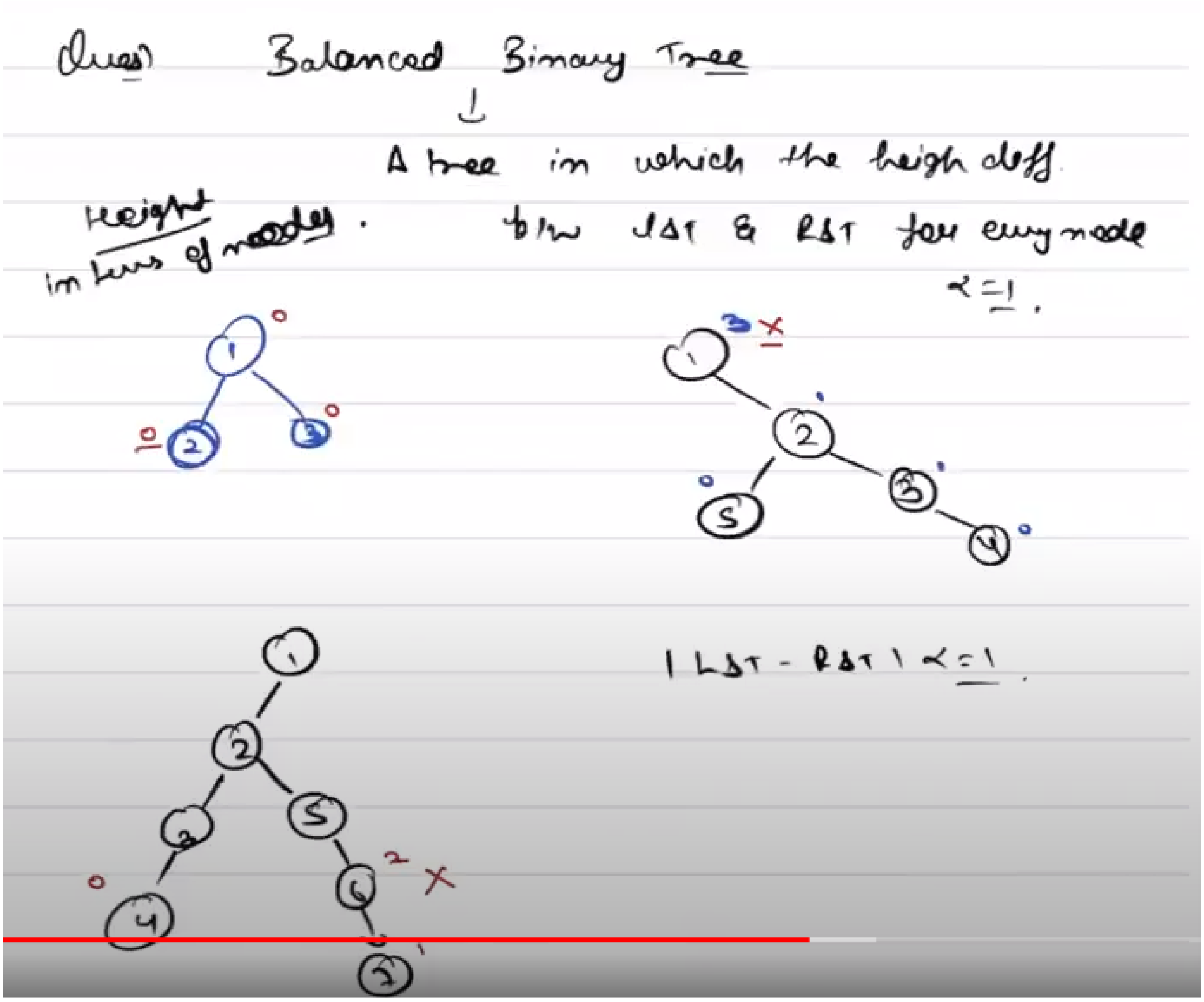
T.C -> O(n)
S.C -> O(H).

```
int ans = -1;
int height (node root) {
    if (root == null) { return -1 }

    l = height (root.left);
    r = height (root.right);
    ans = max (ans, l+r+2);
    return max (l, r)+1;

}
```

Binary tree - 3

Binary Search Tree (BST) -> A Binary tree is a BST if

for all Nodes - All elements in left sub tree < node < All elements in right sub tree



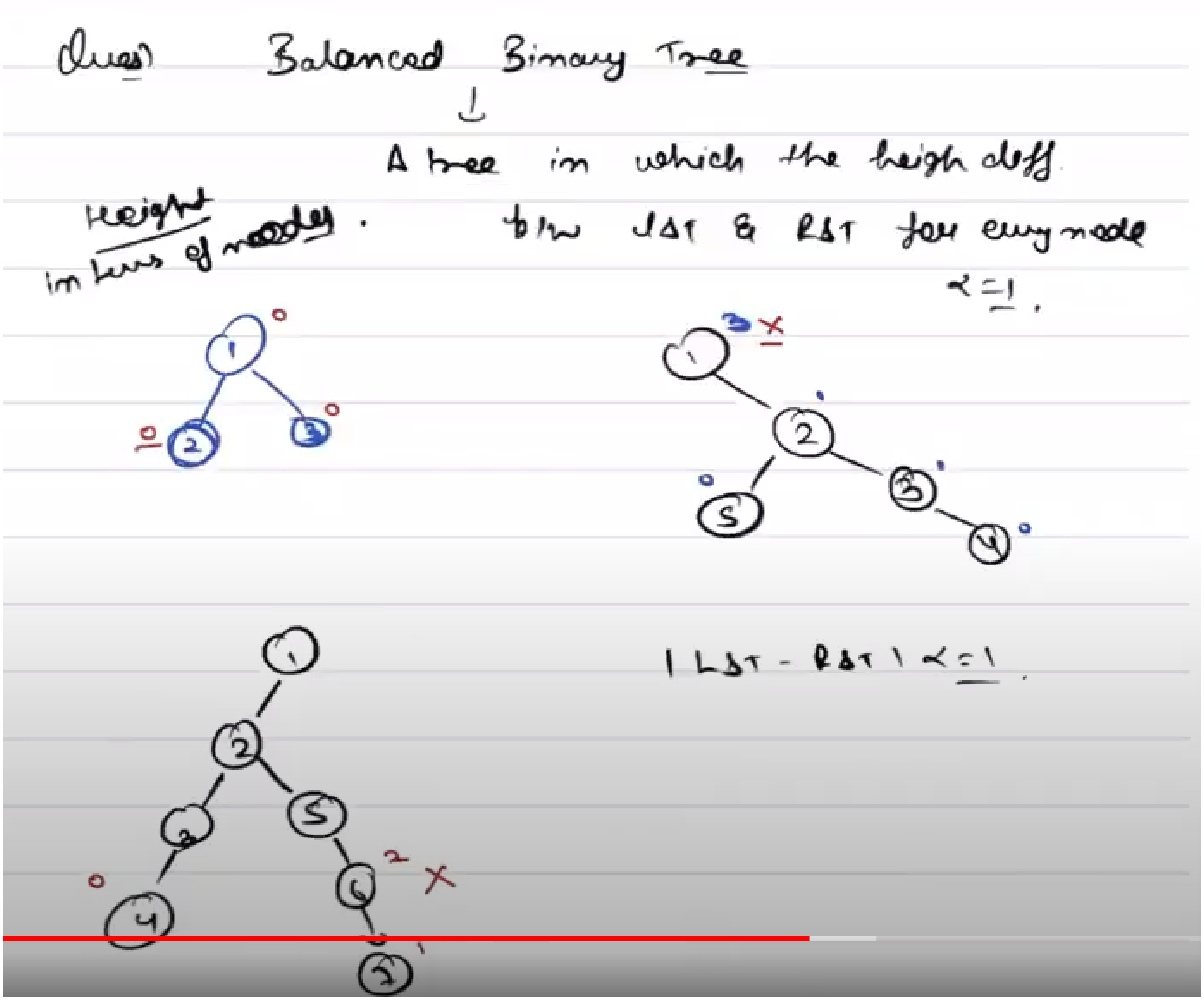
Ques) Balanced Binary Tree

↓

A tree in which the heigh diff. b/w LST & RST for evry node <= 1.

Height in terms of nodes.

| LST - RST | <= 1.



Ques) Balanced Binary Tree

↓

A tree in which the heigh diff. b/w LST & RST for evry node <= 1.

Height in terms of nodes.

| LST - RST | <= 1.

Searching in BST -> Either element equal to node, grater then node or less then node. if it is greater then search in right tree if smaller go to the left and search it out.

Problem1: Given root node of a BST, search if K exist or not.

https://www.interviewbit.com/snippet/ef45c8ab596b40ffbfdb/

Main logic ->

There is simple logic if K is grater then traverse tree to left or right and if node.value == k return 1 elase base case return 0.

- Write a function searchBST
- Write a base case condition node == null return 0
- write a main condition
  - If node.val == k return 1.
  - if node.val > k return searchBST pass node.left;
  - else return searchBST pass node.right;

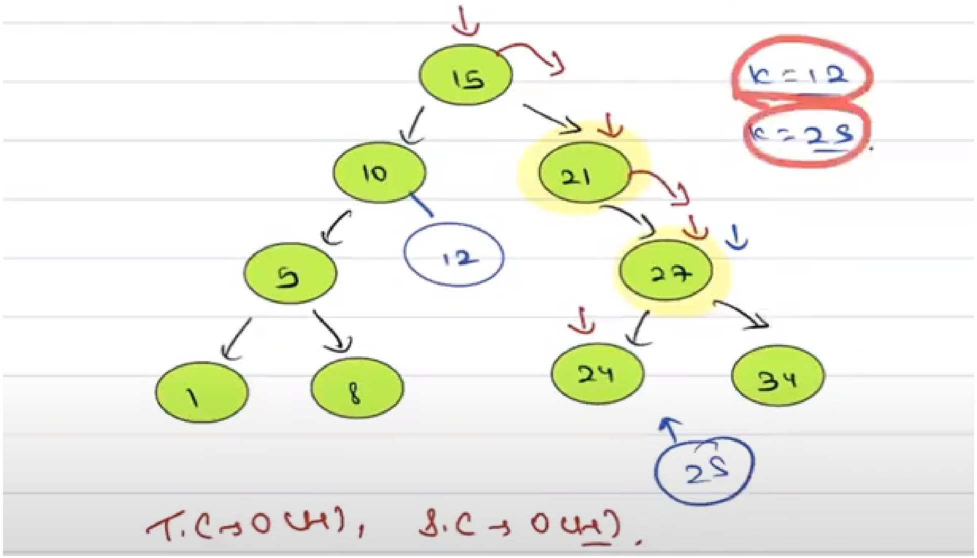
Insert an element in BST ->

Problem: Given root of BST, insert node with data k in the BST. Inserting should be done without suffling the existing node.

https://www.interviewbit.com/snippet/203a71c71aa7c58c119b/

Main Logic ->

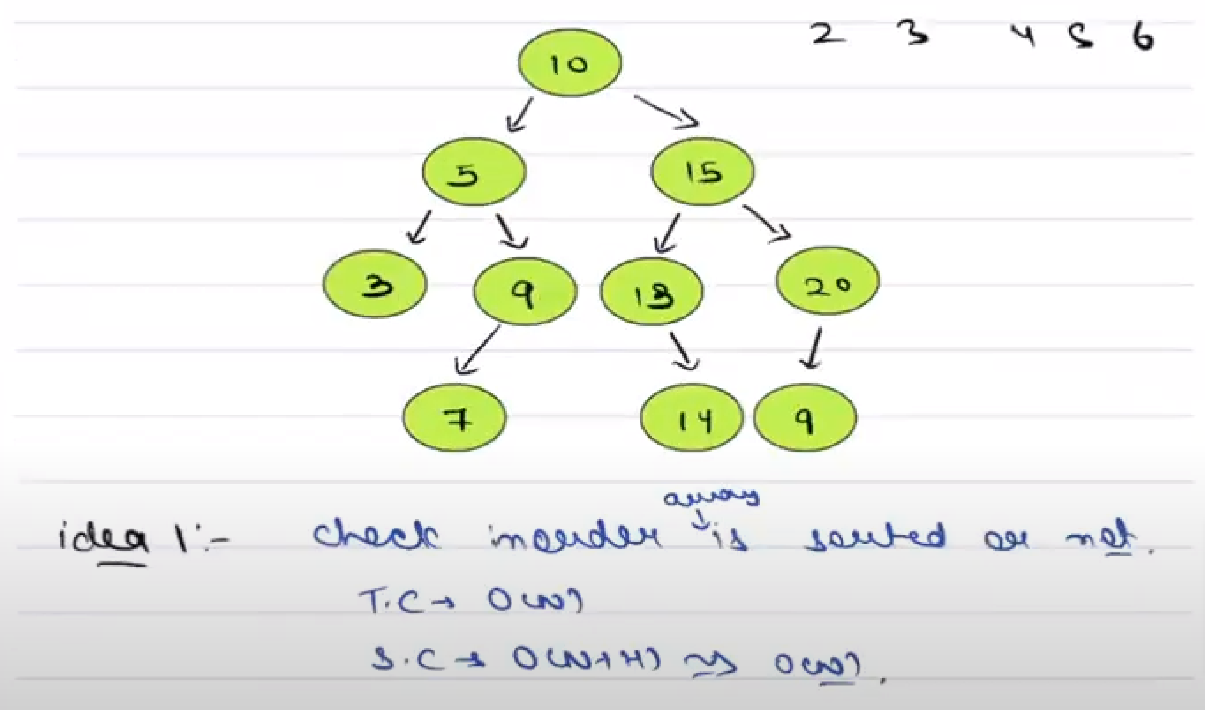
- Create a function called insertKInBST.
- Write the base condition: If the node is null, create a new node with the value 'k', assign it to 'B', and return it.
- Write the main condition:
  - If the value of the current node is equal to 'k', return the node.
  - Otherwise, if the value of the current node is greater than 'k', create a variable 'lAns' and assign the result of calling the 'insertKInBST' function with the arguments 'node.left' and 'k'. Then, assign 'lAns' to 'node.left' and return the node.
  - Otherwise, create a variable 'rAns' and assign the result of calling the 'insertKInBST' function with the arguments 'node.right' and 'k'. Then, assign 'rAns' to 'node.right' and return the node.



Problem3: Given root of a binary tree, check if it is BST or not?
https://www.interviewbit.com/snippet/a37c3d832295740a7c76/

Main Logic ->

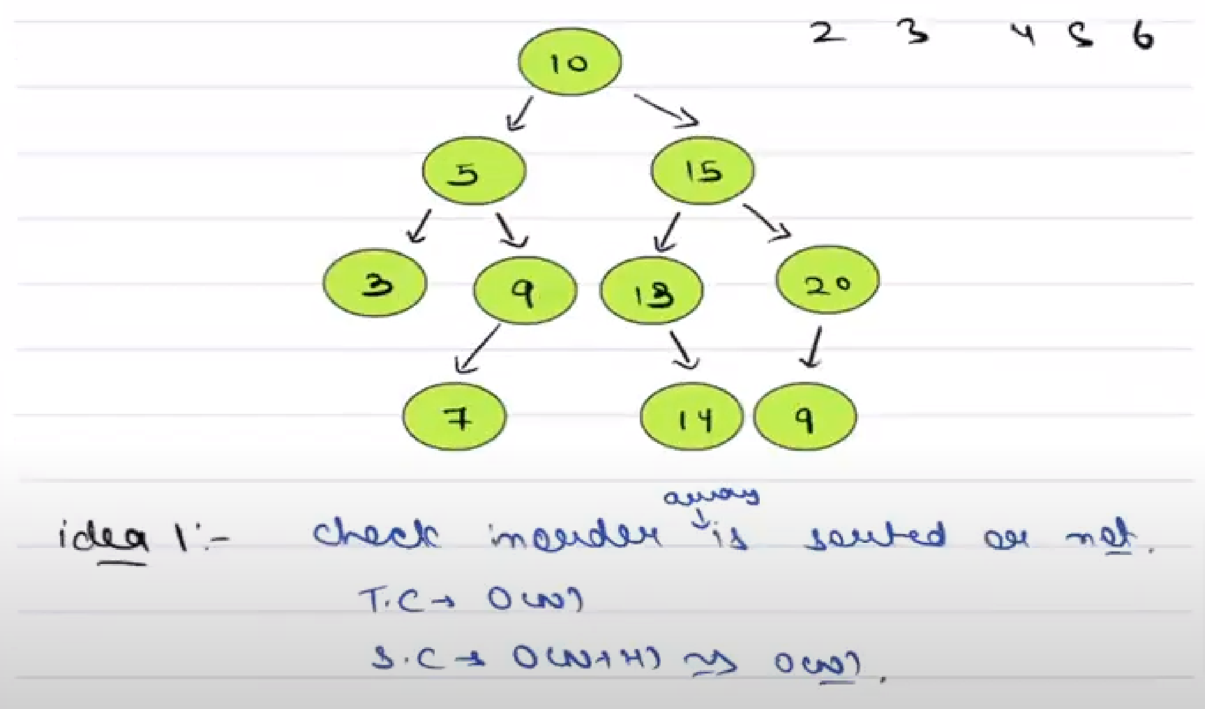
1. Start by setting up a private integer variable named "prevValue."
2. During the initial recursive call, execute an in-order traversal of the tree (within both functions).
3. Introduce a conditional check: if the value of the current node, denoted as "node.value," is less than the "prevValue," terminate the process with a return value of 0.
4. Following the evaluation of the condition, replace the "prevValue" with the value of "node.value."
5. In the subsequent recursive call, finalize the operation by returning a value of 1.



idea 1:- check inorder array is sorted or not.

T.C → O(N)

S.C → O(N+H) ≈ O(N).



idea 1:- check inorder array is sorted or not.

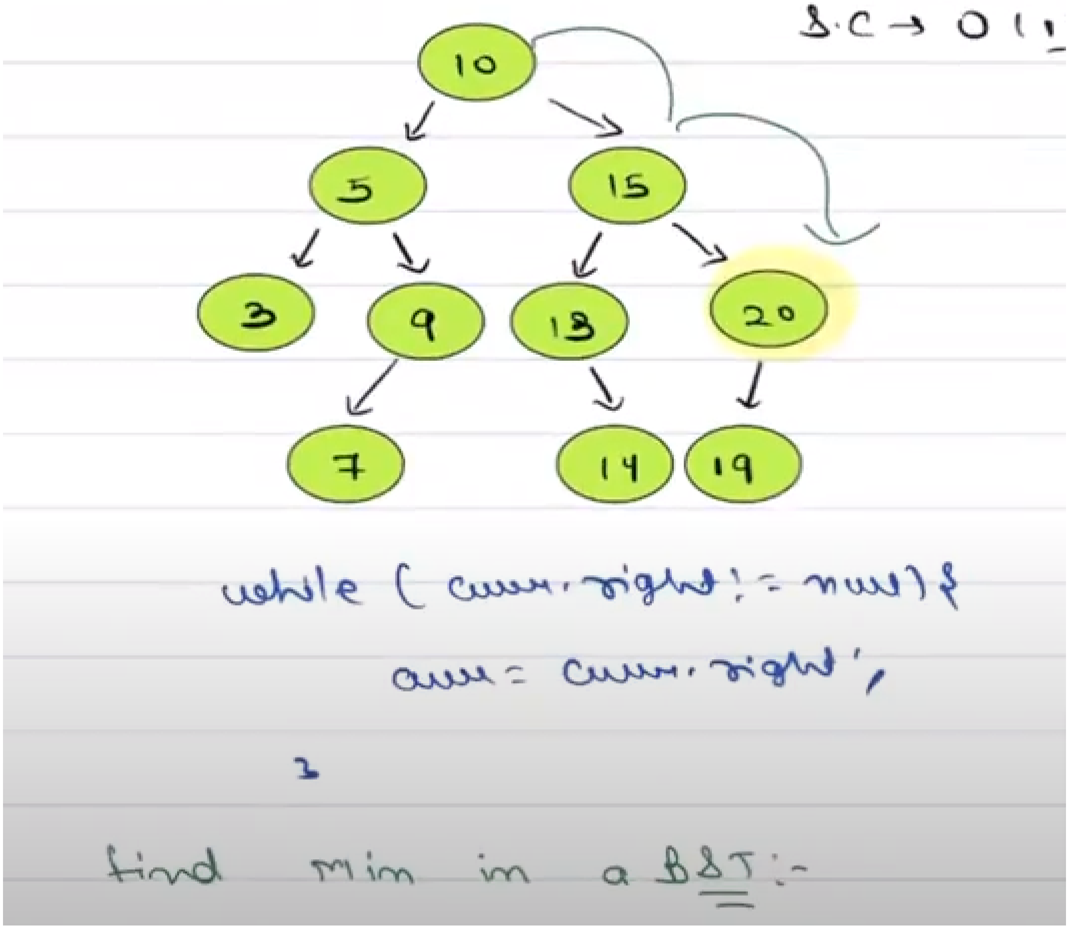
T.C → O(N)

S.C → O(N+H) ≈ O(N).

Max in the BST -> It is always right most child
https://www.interviewbit.com/snippet/b781afd01a8622798cc7/

Main logic ->

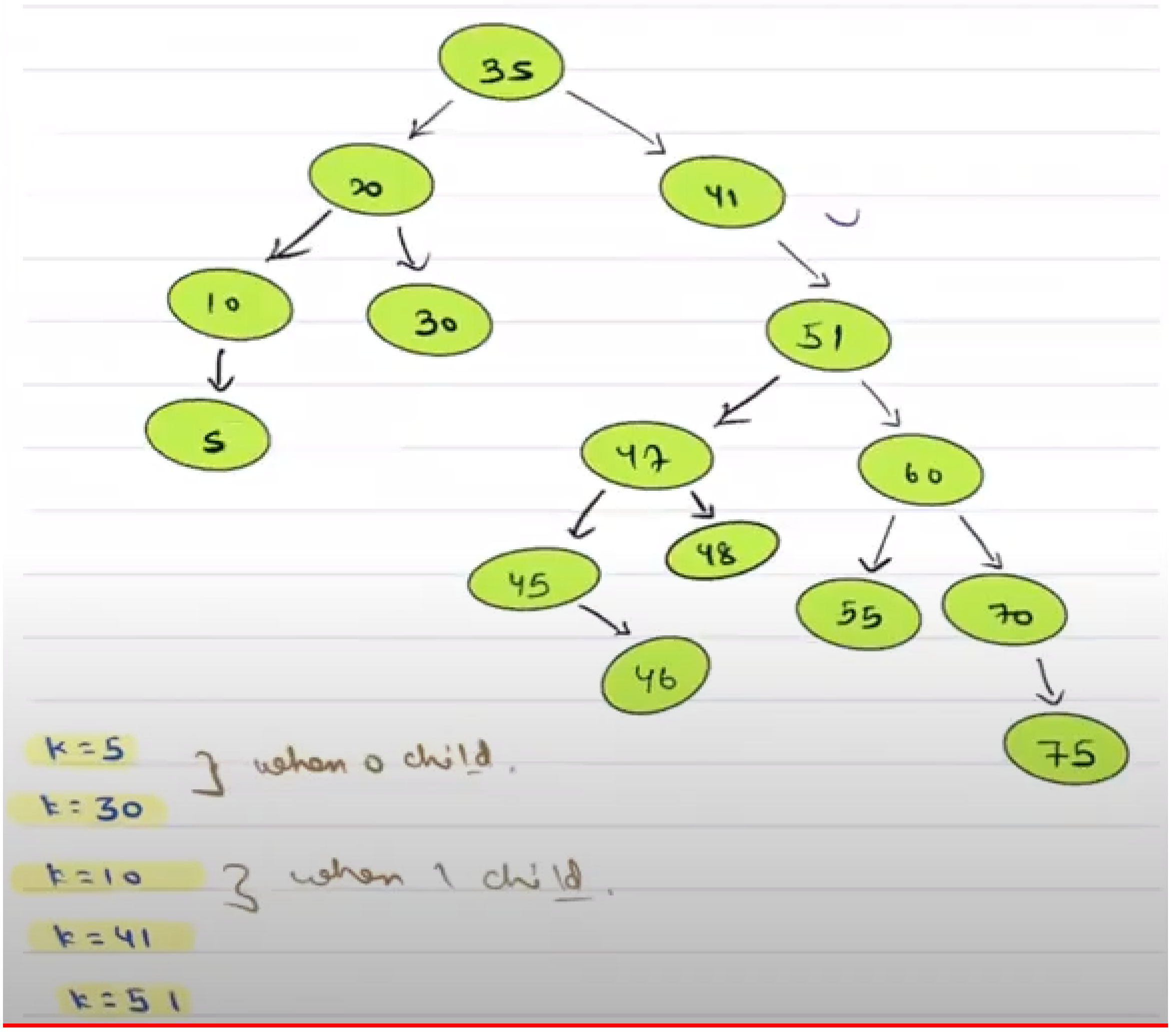
1. Define a function called maxInBST that accepts one parameter, node, and returns an integer.
2. Check a condition: If node is equal to null, return Integer.MIN_VALUE as a sentinel value to indicate that there is no valid maximum.
3. Create an integer variable called maxVal.
4. Recursively call the maxInBST function on the right subtree of the current node (node.right) and assign the result to maxVal. This is because in a Binary Search Tree (BST), all greater values are found in the right subtree.
5. Calculate the maximum value between maxVal (the maximum value from the right subtree) and the value of the current node (node.val) using Math.max.
6. Return the calculated maximum value as the result.



```
while (curr.right != null){
        curr = curr.right;
    }
```

find min in a BST :-

Delete node in BST



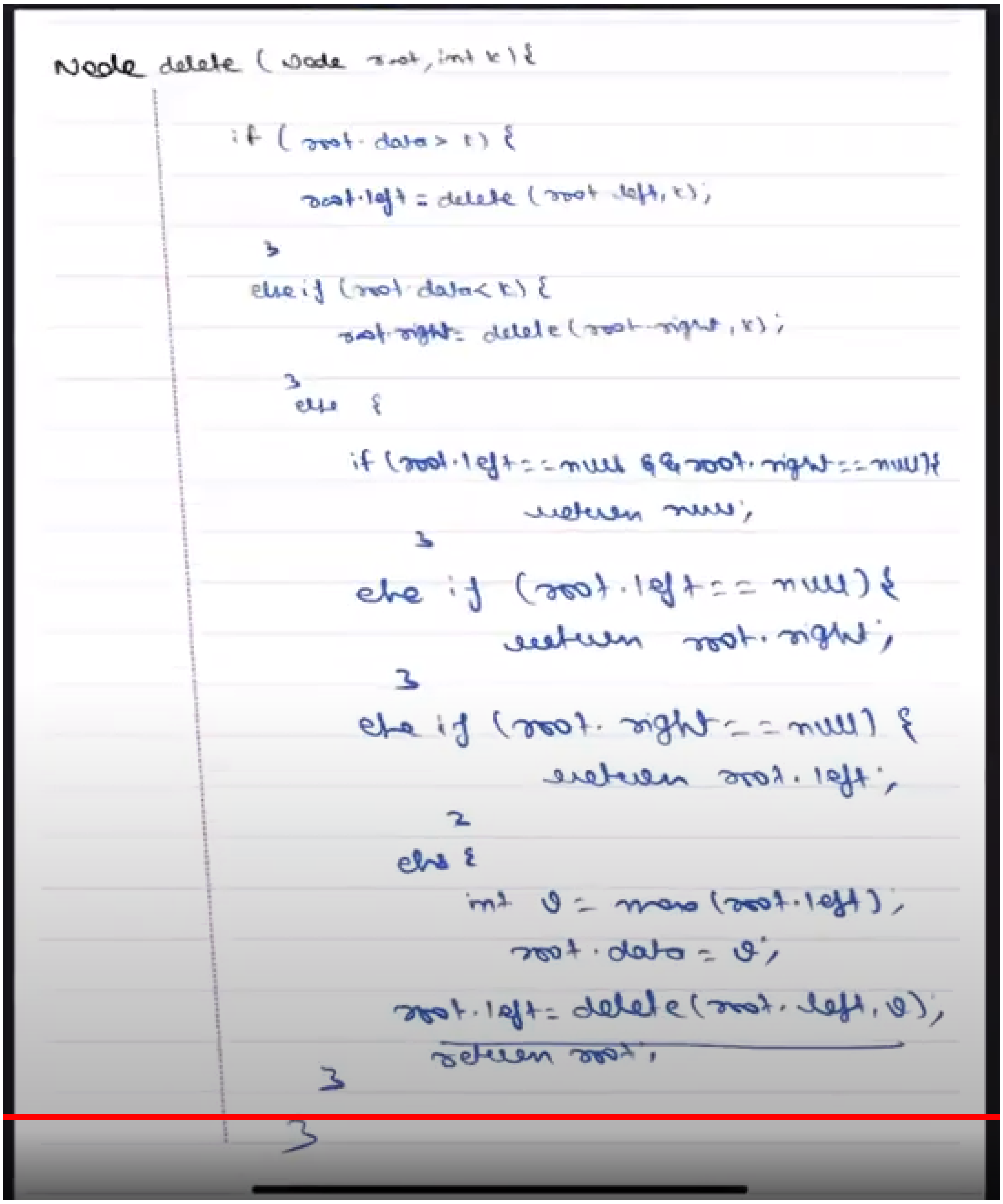
k = 5

k = 30 } when 0 child.

k = 10 } when 1 child.

k = 41

k = 5 1

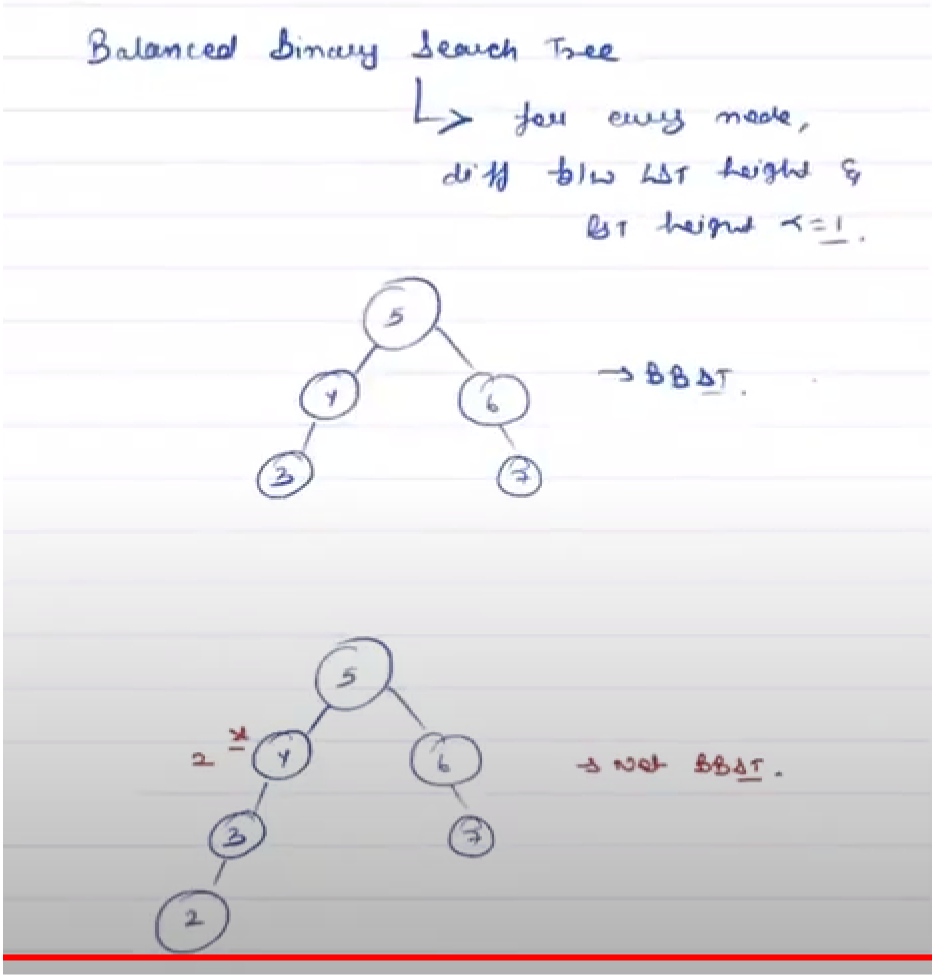


```
Node delete ( Node root, int k) {

    if ( root.data > k) {
        root.left = delete (root.left, k);
    }
    else if (root.data< k) {
        root.right= delete(root.right, k);
    }
    else {
        if (root.left==null && root.right==null){
            return null;
        }
        else if (root.left== null) {
            return root.right;
        }
        else if (root.right==null) {
            return root.left;
        }
        else {
            int v = max (root.left);
            root.data = v;
            root.left= delete(root.left, v);
            return root;
        }
    }
}
```

Given a sorted array. Construct balance BST using this array and return its root node.
https://www.interviewbit.com/snippet/c99ec3c9803330f6da81/

Main logic ->

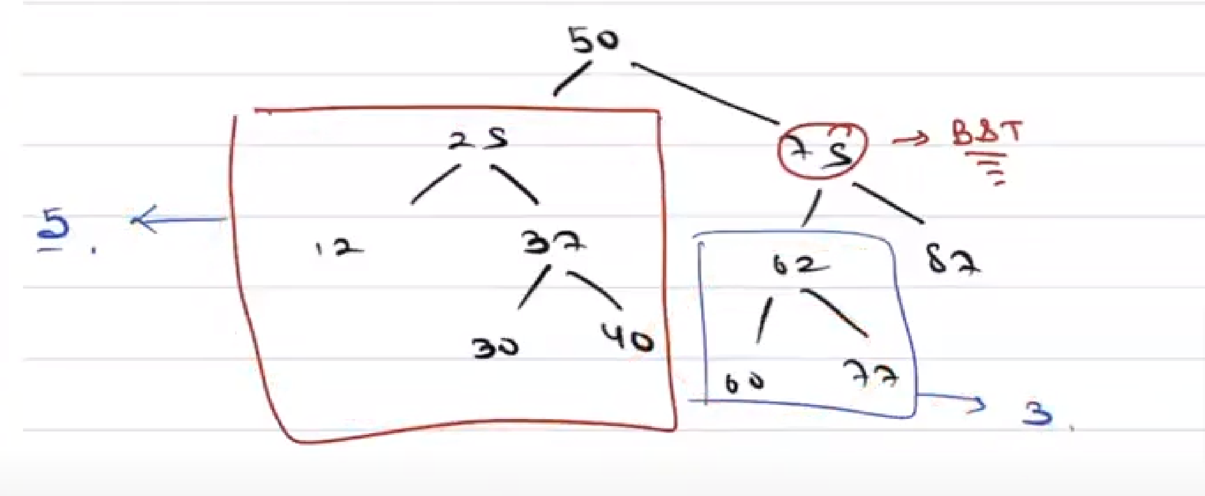
1. Begin by designing a constructor function that takes three parameters: a sorted array called "arr," and two indices, "lo" and "hi."
2. Establish a base condition: if "lo" is greater than "hi," return null.
3. Compute the midpoint and assign it to a variable named "mid" using the formula (lo + hi) / 2.
4. Create a new node labeled "nn" using the value from "arr[mid]."
5. Invoke the recursive function to traverse the left side, passing "arr," "lo," and "mid-1" as arguments.
6. Invoke the recursive function to traverse the right side, passing "arr," "mid+1," and "hi" as arguments.
7. Finally, return the "nn" node to complete the operation.

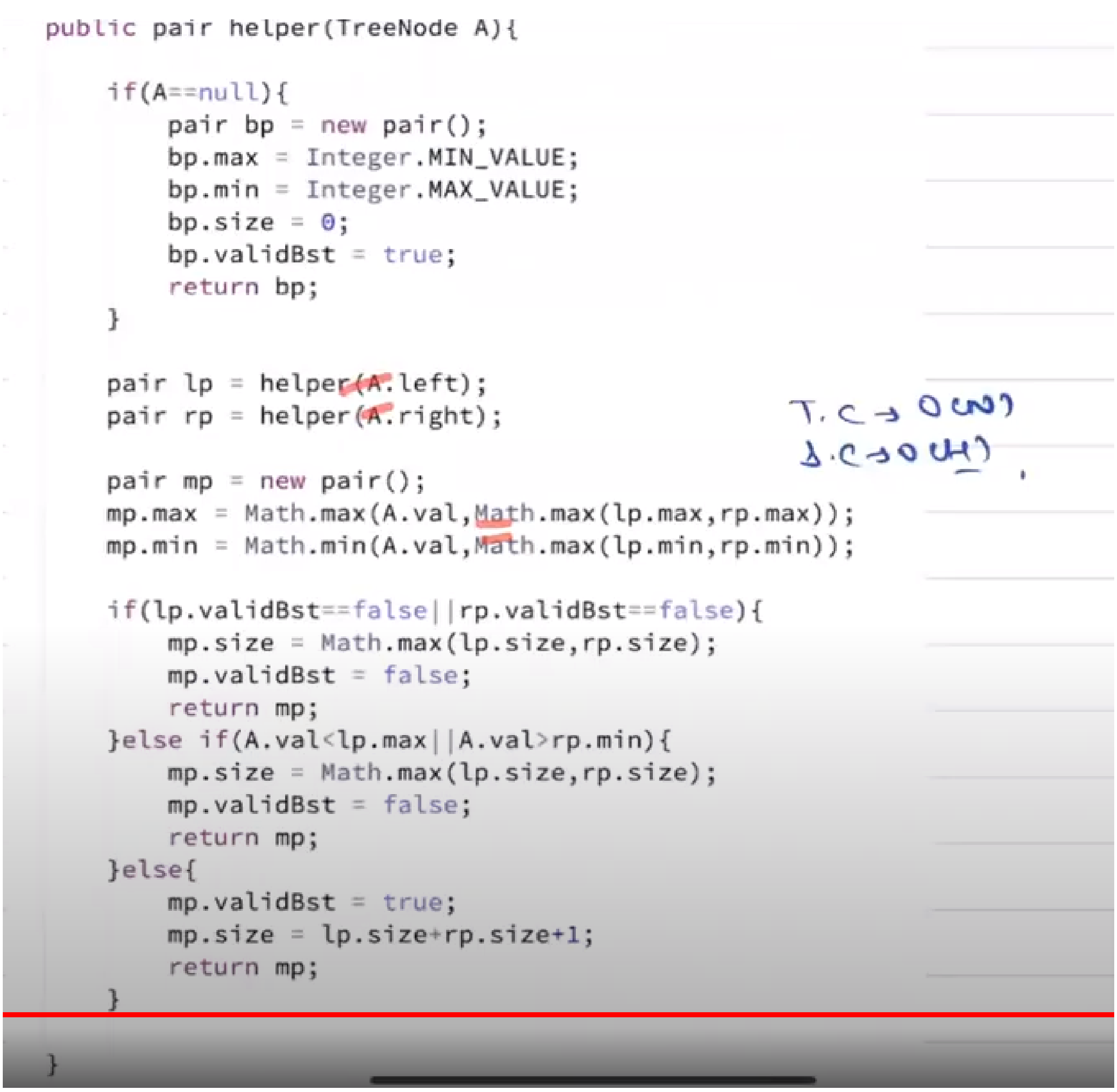


Balanced Binary Search Tree

↳ for every node, diff b/w LST height & RST height <= 1.

Largest BST Subtree





```
public pair helper(TreeNode A){

    if(A==null){
        pair bp = new pair();
        bp.max = Integer.MIN_VALUE;
        bp.min = Integer.MAX_VALUE;
        bp.size = 0;
        bp.validBst = true;
        return bp;
    }

    pair lp = helper(A.left);
    pair rp = helper(A.right);

    pair mp = new pair();
    mp.max = Math.max(A.val,Math.max(lp.max,rp.max));
    mp.min = Math.min(A.val,Math.max(lp.min,rp.min));

    if(lp.validBst==false||rp.validBst==false){
        mp.size = Math.max(lp.size,rp.size);
        mp.validBst = false;
        return mp;
    }else if(A.val<lp.max||A.val>rp.min){
        mp.size = Math.max(lp.size,rp.size);
        mp.validBst = false;
        return mp;
    }else{
        mp.validBst = true;
        mp.size = lp.size+rp.size+1;
        return mp;
    }

}
```

T.C → O(N)

S.C → O(H)