**Binary Trees**

At most 2 children.

1) Full Binary Tree: Every node has 0 or 2 children.

2) Complete Binary Tree: All levels are completely filled and the last level has nodes as left as possible.

3) Perfect Binary Tree: All leaf nodes are at the same level.

4) Balanced Binary Tree: Height can be at a maximum of logn, where n is the number of nodes (done to improve time complexity while searching or other operations).

5) Degenerate Tree: Skew form of tree. (Each node has a single child)

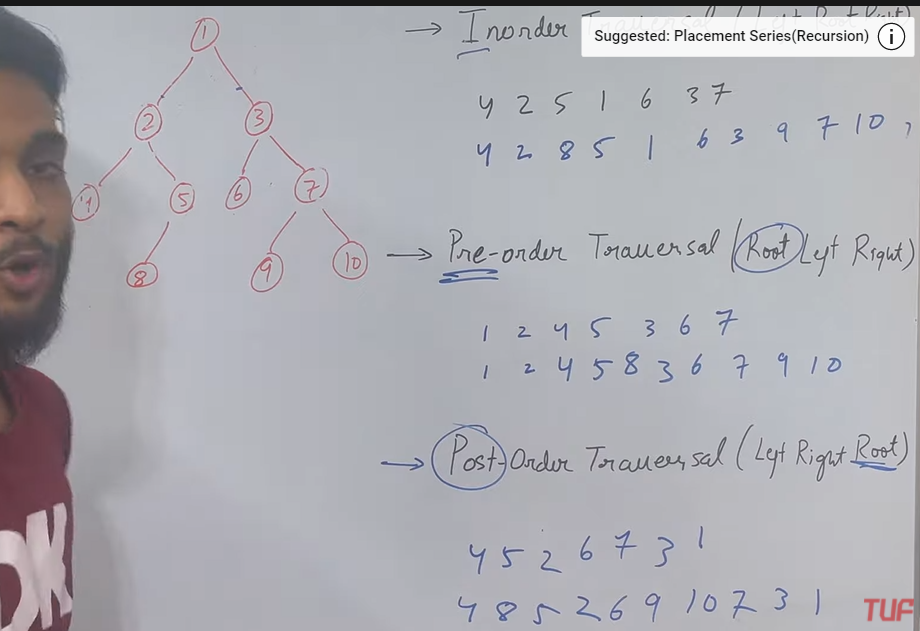
**Tree Traversals**

There are 3 DFS techniques as follows:

1) Pre-order traversal: Root, left, right => print root then go to the left most subtree and perform root, left, right. Then go to the right sub tree (of the main root) and perform root, left, right.

2) In-order traversal: Left, root, right. Go to the leftmost subtree and perform left root right, then print main root and go to the right most subtree and perform left, root, right.

3) Post-order traversal: Left, right, root. Go the leftmost subtree and perform left, right, root. Then go to the rightmost subtree and perform left, right, root. Finally, print the main root.



Breadth first search is performed normally. (Level order traversal)

To solve all dfs together: <https://www.youtube.com/watch?v=ySp2epYvgTE&list=PLgUwDviBIf0q8Hkd7bK2Bpryj2xVJk8Vk&index=14>

**Maximum Depth of a Binary Tree**

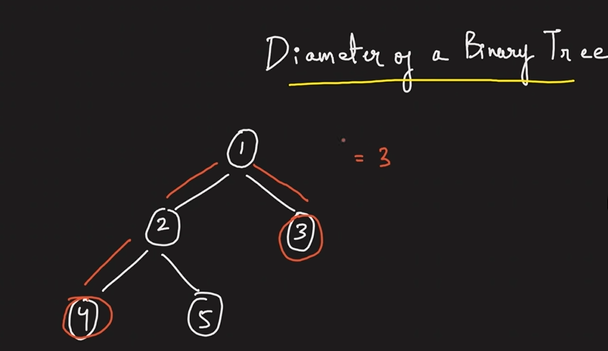
We can use BFS which has space complexity of O(N) maximum for N nodes (if the tree has levels filled upto a good amount).

However, if we use recursion (unless it’s a skewed tree for which space is again O(N)), it uses less auxiliary space.

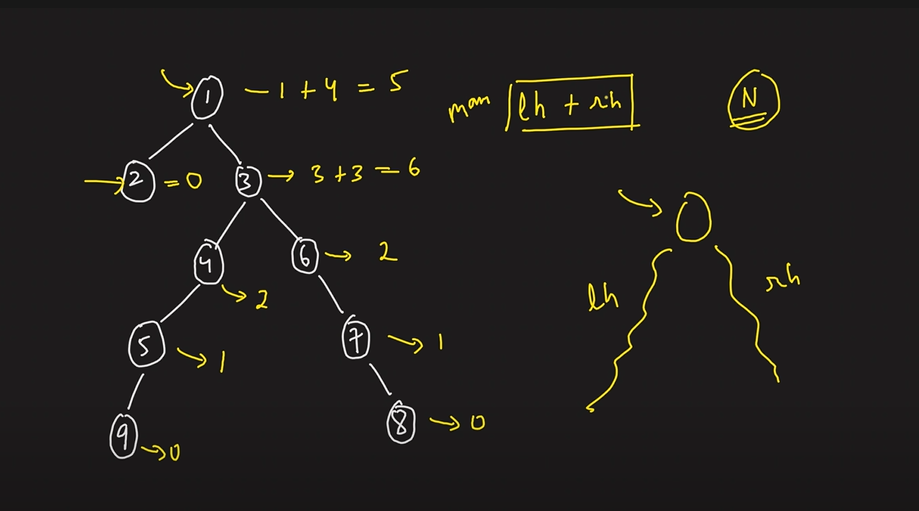
**Check for a Balanced Binary Tree**

**Diameter of a Binary Tree**

Diameter is the longest path between any two nodes and passing through the root is not necessary.

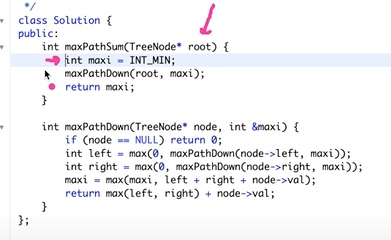


Brute force: Sum of max heights. However, this is N^2 as we traverse all nodes and traverse for all.



**Maximum Path Sum in Binary Tree**

<https://www.youtube.com/watch?v=WszrfSwMz58&list=PLgUwDviBIf0q8Hkd7bK2Bpryj2xVJk8Vk&index=18>



Basically we consider each node and calculate the largest sum possible including it which we store in the maxi variable.

However, we return only either the left or right subtree of that node to its parent node because a path can contain only 1 of the left/right nodes.

(Also we have written max(0) in int left and int right because if the child nodes are negative then do not consider them while backtracking).

**Identical Root**

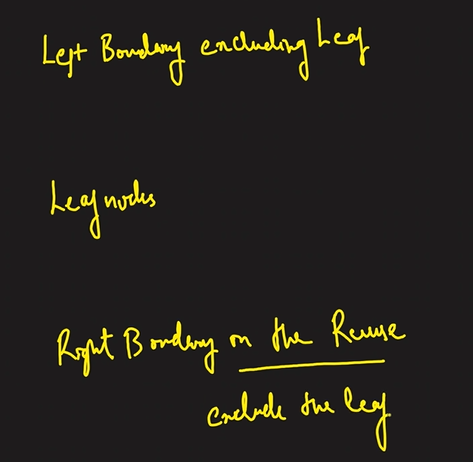
Check traversals and compare.

**Zig-Zag Traversal**

Just use flag with normal bfs.

**Boundary Traversal**

****

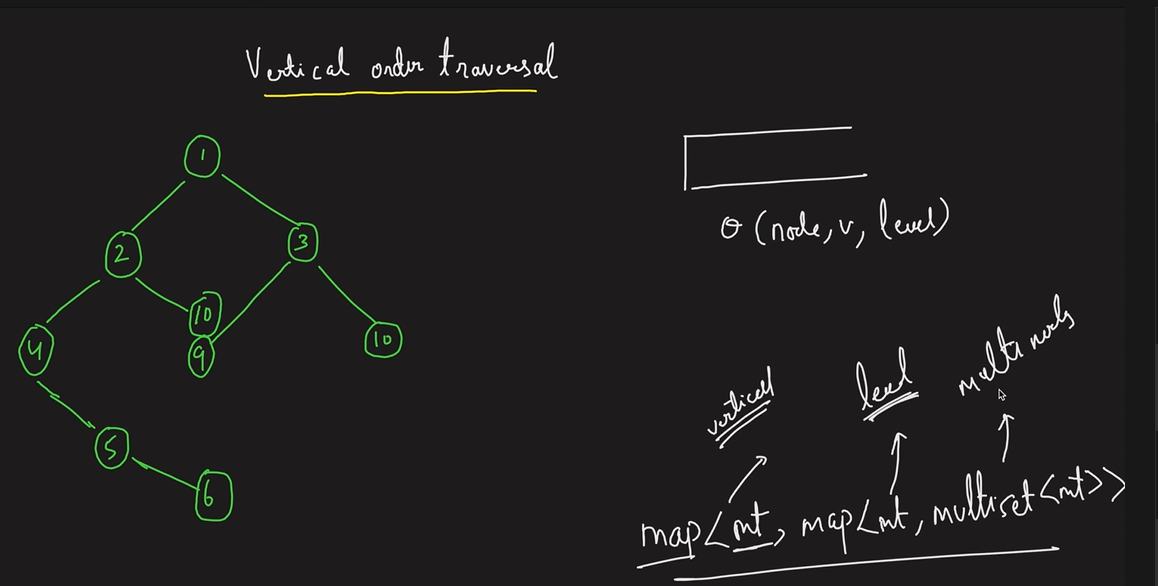
****

1) Left boundary excluding leaf: keep going left from the root unless there isn’t a left possible and then go right.

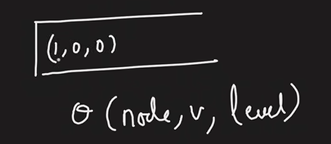
2) Leaf nodes: perform in-order traversal to get the leaf nodes in the order.

3) Right boundary: keep going right from the root unless there isn’t a right and then go left. However, we need to reverse this order and then place it in the answer.

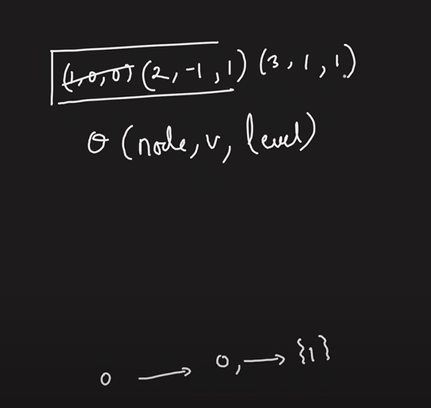
**Vertical Order Traversal**



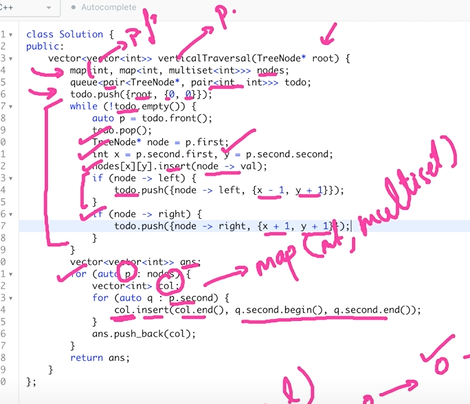
Using multiset instead of set because at the same level and vertical we might have 2 nodes with the same value.



We store this element in our data structure (map) and pop it from the queue. Now, we perform bfs on it and if we go left, we decrease vertical by 1 and increase level by 1 whereas, if we go right we increase level and vertical both by 1.

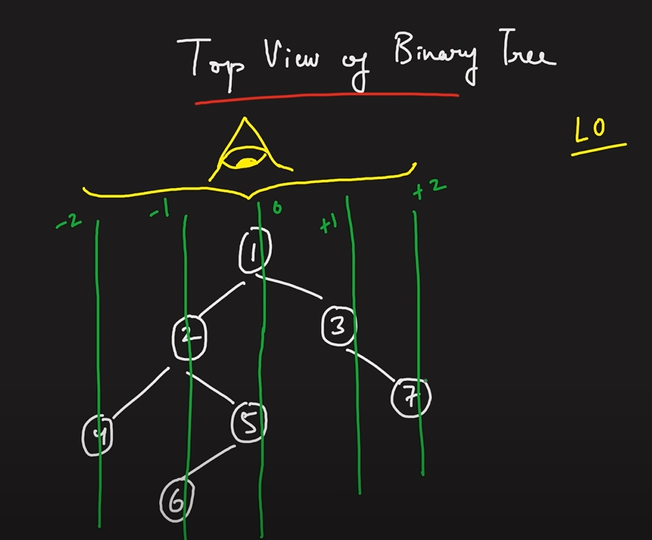


0->vertical, 0->level, 1->value



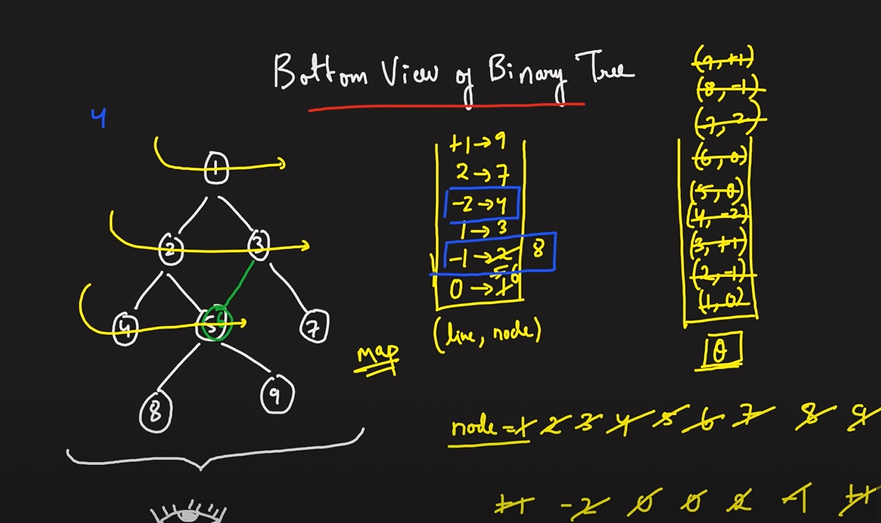
In the first for loop we are iterating over the verticals and within the internal for loop we are iterating over the levels. Hence, we are successfully able to add all the nodes.

**Top view of a Binary Tree**



The first node in every level is my top view.

**Bottom view of a Binary Tree**



The last node in every level is my top view.

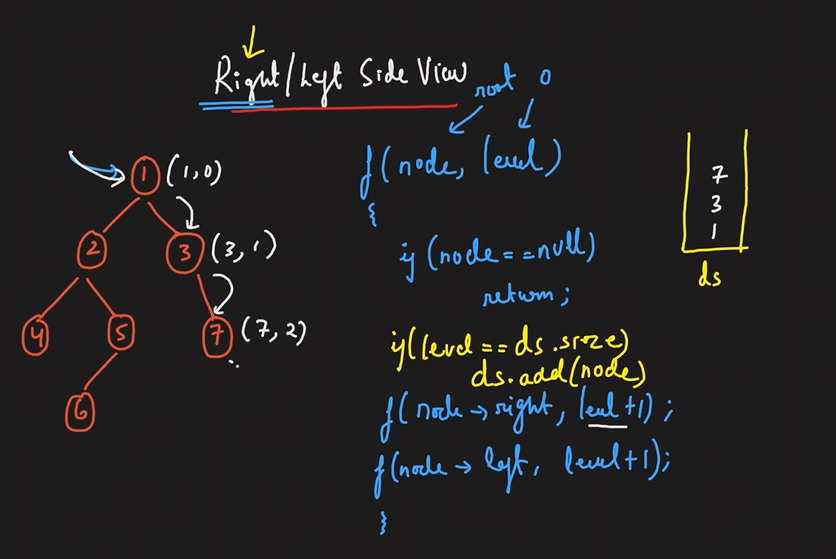
Since a map stores in sorted order, -2 will be the first key.

Note here that since the last node is needed for every level, hence when we insert in the map, the older value gets overwritten by the later one.

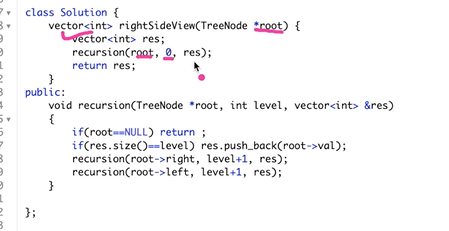
**Right view of a Binary Tree**

We can use bfs and get the last node for each level. However, the space complexity will be worst when the tree is complete and the queue has to store n/2 nodes of the last level. Hence, we use dfs so that we can get an auxiliary space complexity of O(height).

We use reverse pre-order: Root,Right,Left (right first because we need to fill our data structure with the right nodes first)



When we come to elements on the left, the stack/queue is already filled with size greater than their levels and hence we don’t add them to our data structure. We add 6 because its level = 3.



**Print root to node path**

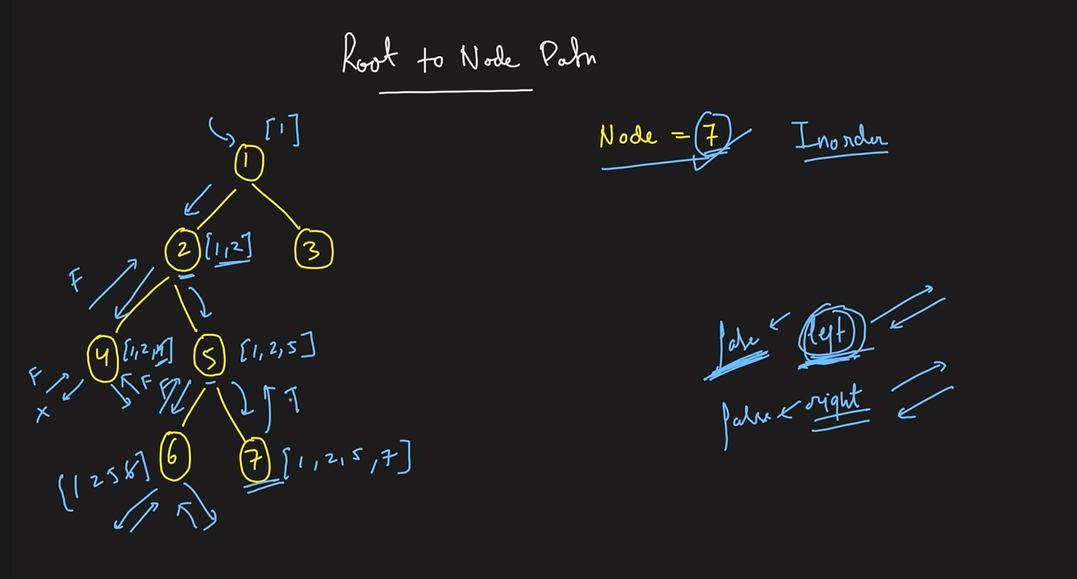
We create a single vector to store the result.

Perform inorder traversal.

We add the starting node to our vector and go left and add that node as well. Then we check for its left and right and we do this recursively and if both the left and right subtree return false, i.e., they do not contain the node to which we want to traverse, then we pop that element from our vector.

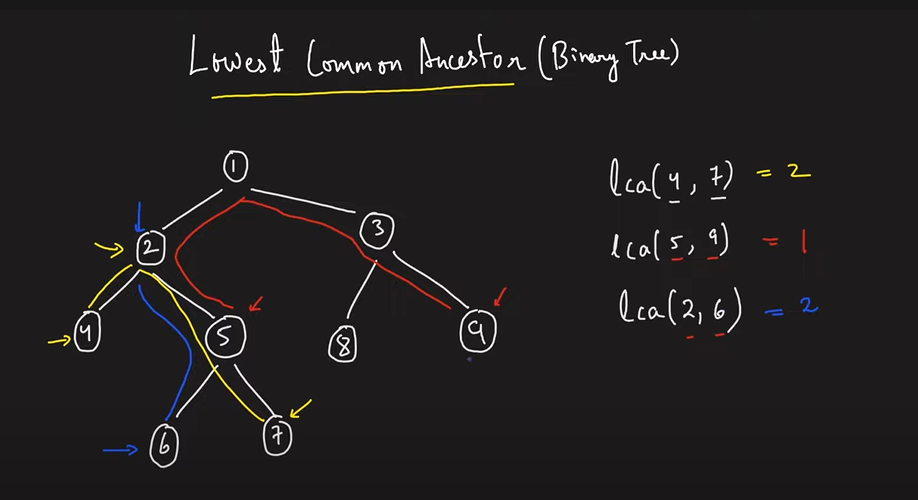


Since the node 4 gets false from both its left and right side, we pop 4 and move to 5.

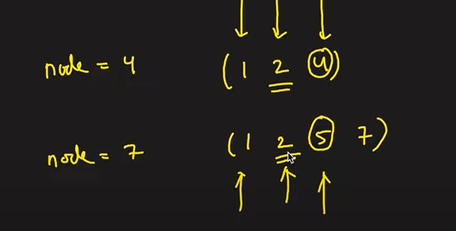


As soon as the right subtree of 5 returns true, we just return and keep retuning true till we reach the root node.

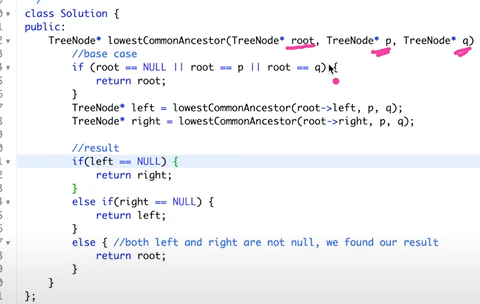
**Lower Common Ancestor**



Brute force approach is that find the path to both the nodes and get the last common node. However, time complexity is O(N) to find the path and space complexity of O(N) to store the data structure. However, we can save this space.



However, we can use the approach wherein we return a Boolean and when both left and right return true we can return the node.



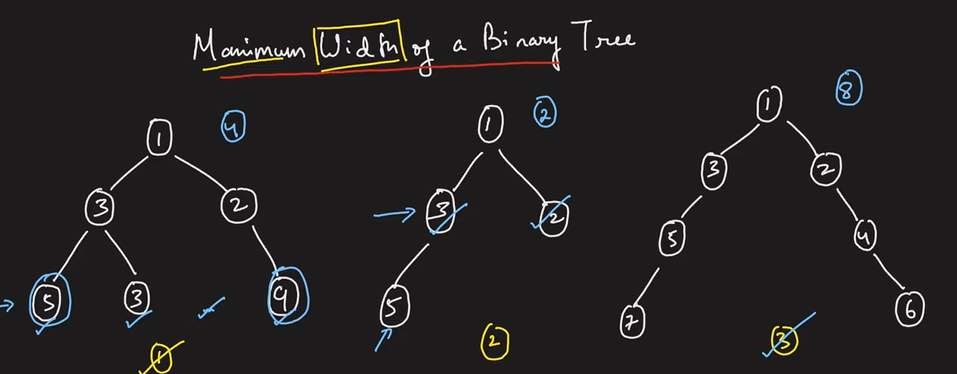
**Maximum Width of Binary Tree**

Width = maximum number of nodes between (including the two nodes) any two nodes in a given particular level.

Figure 1: width = 4 between first and last node of the last level.

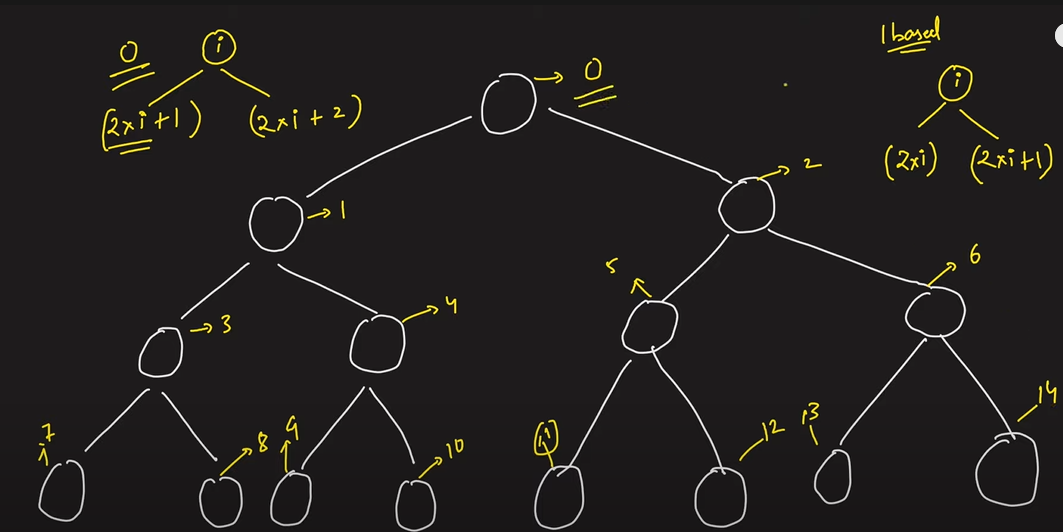
Figure 2: width = 2 between first and last node of the second level. We cannot take the last level because we only have a single node there.

Figure 3: width = 8 between first and last node of the last level. (because we have a first and last node, so we can have some imaginary nodes in between)



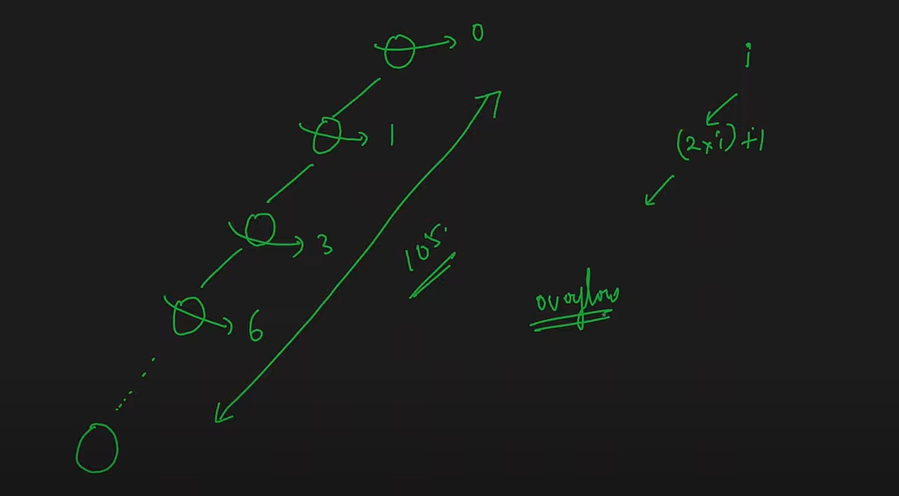
The approach is using bfs. However, if we can somehow make the indexing proper, then we only need to get the first and last node in a particular level and perform **last – first + 1** to get the answer.

Hence, we try to perform this indexing.

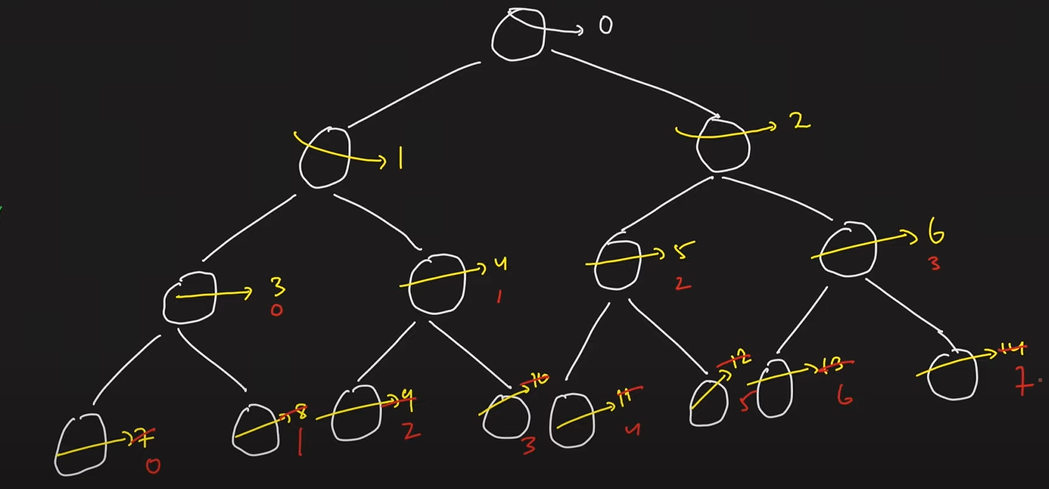


If we use the above approach then even if some nodes are missing in between the start and final, the indexing will be done perfectly.

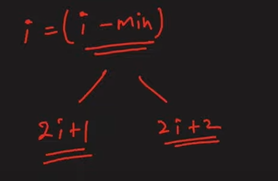
However, we cant directly use the **2\*i+1 and 2\*i+2** approach because for the given example below if we perform this, then the memory overflows. (210^50)

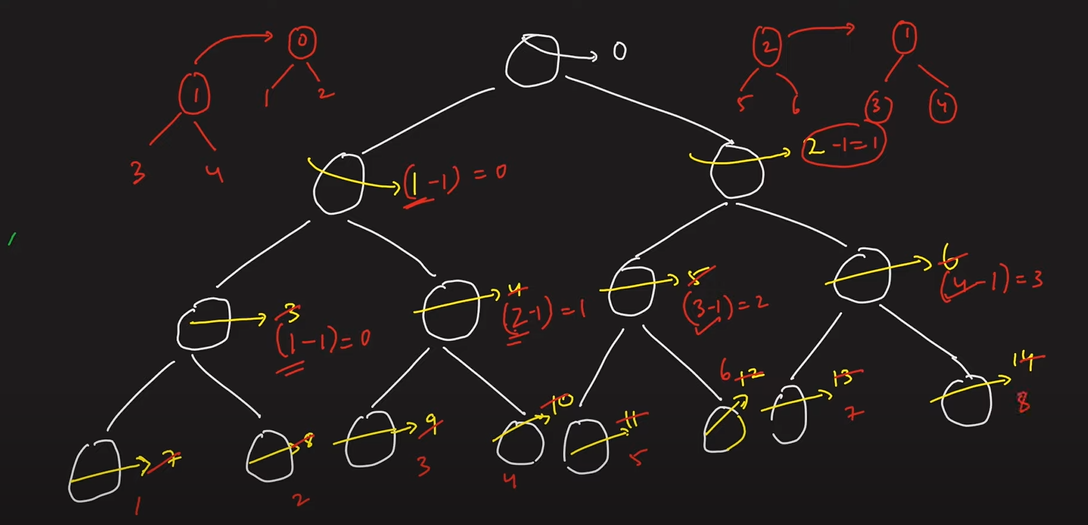


Hence, to reduce this overflow, we try to do indexing in the following way:

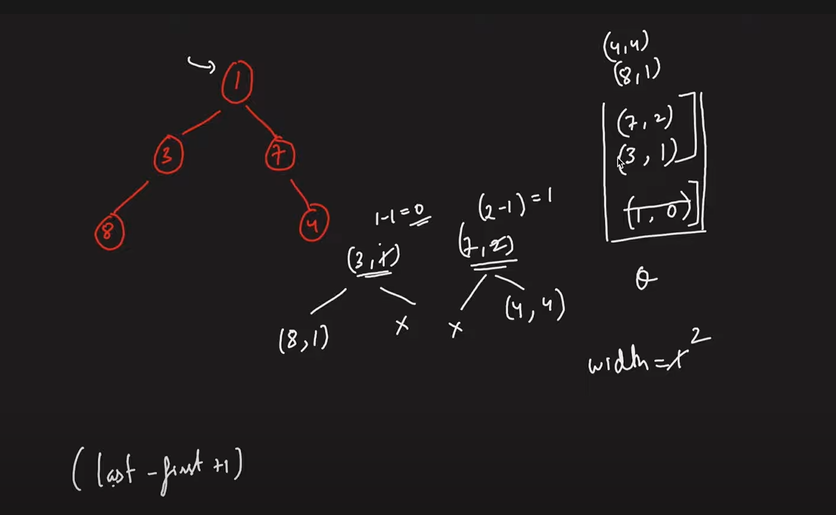


We subtract the minimum value of node of a particular level from all nodes of that level.

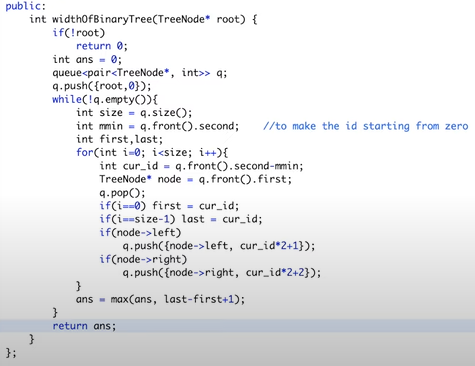




Hence, we perform bfs and take a queue and store a **pair<root,index>.**



We have indexed 8 as 1 and 4 as 4 to avoid overflowing. (else we would have done 4 and 7 respectively which will eventually cause memory overflow after sometime).

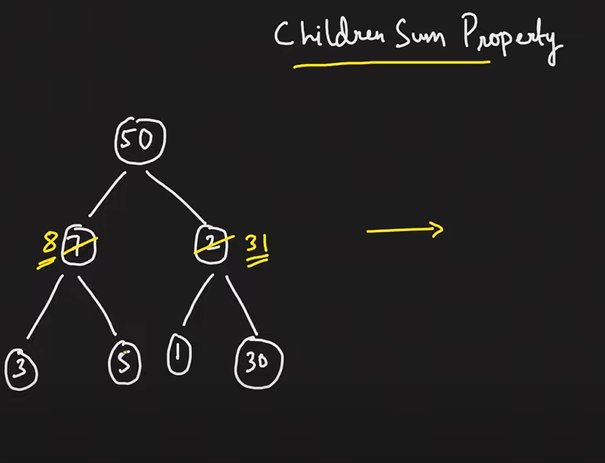


**Children Sum Property**

Sum of a node should be equal to its 2 children.

However, its not a simple problem as we can see from the following figure (direct addition of 3+5 and 1+30 will give us 8+31 != 50).

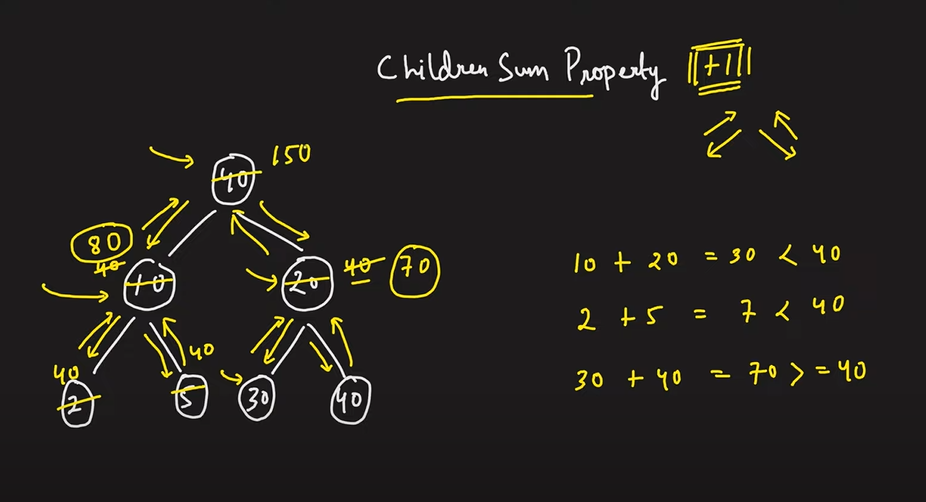
We can add 1.



We make use of left and right traversal. Go left then come upwards and then go down rightwards and come back.

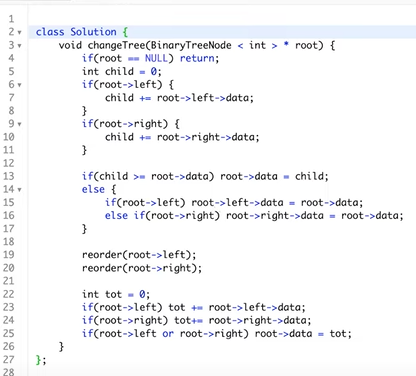
While going downwards, if the sum of the children’s value is lesser than the parent value, we make each of it equal to its parent. If the sum of the children’s value is greater than their parent, we make parent equal to their sum and continue to explore below the children.

While coming up we just add the left and right child value to get the parent value.



**Hence, we see that earlier when we add 8 and 31 to get 39 which was less than 50, we couldn’t subtract from 50. Therefore, its better to increase the child values atleast equal to their parent value while going down so that when we traverse upwards and add them, they are not less than the parent and if more, we can simple add +1 any number of times to the parent.**

**Hence, we increased values of child nodes while going down to atleast their parent value (if their sum was less) so that their sum is atleast equal to their parent on traversing upwards (because a greater sum can be covered by adding +1’s to the parent).**

****

**Print all the Nodes at a distance of K in Binary Tree**

Let us say we need to find all the nodes at a distance of 2 from 5 in the given binary tree.

****

Step1: To get the nodes in the upward direction as well, we need to have pointer from the child to the parent node aswell so that we can traverse upwards. Hence, we perform BFS using a queue and mark parent node for the child node. (we can use hashmap to store parent of every nodes/mark parent pointers).

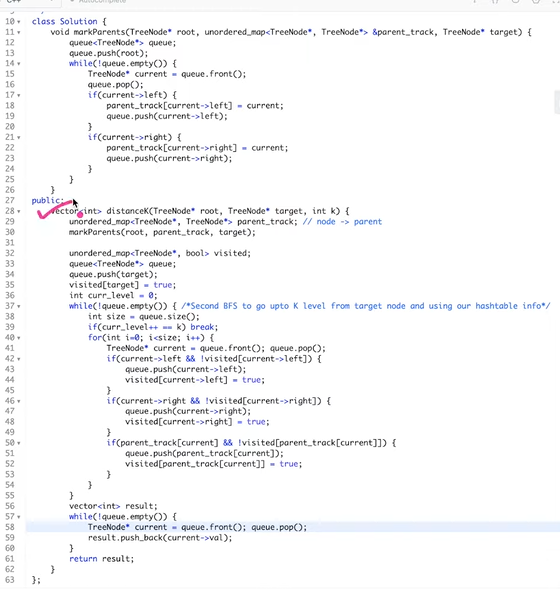
Step2: Now we can move both upwards and downwards from a particular node. Find nodes at a distance of k. We do a bfs traversal on the given node and traverse both upwards and backwards radially (means moving together, eg in the below figure we move to 2,3,6 together) and calculate distance while making the traversal and when the distance = k, we are at all the nodes which are at a distance of k from the given node.

We use a queue to perform the bfs and a hash to store the visited nodes.



In the next step we take all these 3 nodes (2,3,6) together and move radially outwards from these nodes (like we did from 5 earlier). This is where the visited hash comes into use so that we don’t visit a node again (because we might visit 5 by going upwards from 2 again which we do not want). **We take 2, visit 4&7 and add them to the queue, then we take 3 and visit 1 and it to the queue, finally we take 6 and don’t add anything to queue(since no new neighbours except 5 itself. We also add these new nodes to the visited hash). When we are done with radially traversing the nodes for a particular level of nodes (in the queue using bfs), we pop them from the queue. Now, when distance 2 is reached, we stop and just pop all elements from queue which will be our answer (since we popped some elements from queue earlier when we bfs traversed from them).**

**Therefore, the nodes finally present in our queue are the answer.**

****

**Minimum time to burn a Binary Tree**

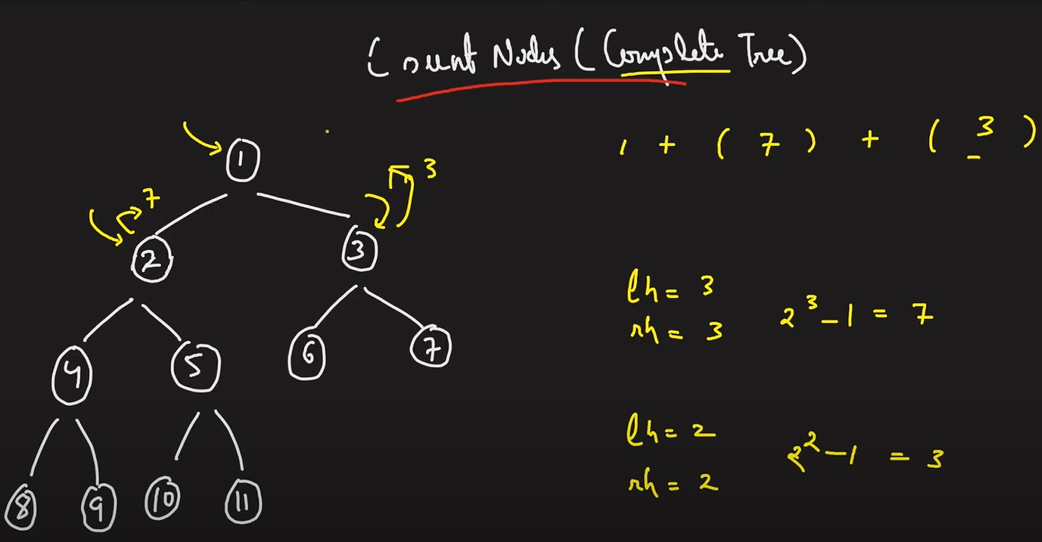
We can traverse radially outwards from a node and the maximum distance node = maximum time.

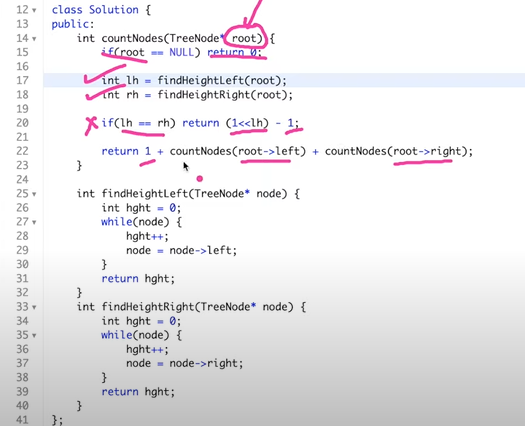
We don’t use dfs because nodes at the same distance are burnt simultaneously.

**Count total Nodes in a COMPLETE Binary Tree | O(Log^2 N) Approach**

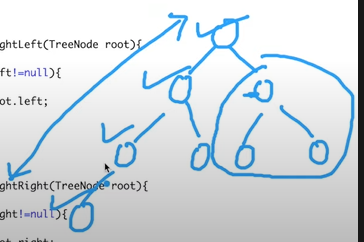
We can use normal bfs but it takes O(N) approach.

For each node we compute its left and right height. If lh = rh, then we can say it’s a complete binary sub-tree and apply the formula of nodes as 2\*h-1. If they are not equal, we recursively traverse them until lh == rh or the nodes end and while doing so, we apply 1 + left tree nodes + right tree nodes as the formula.



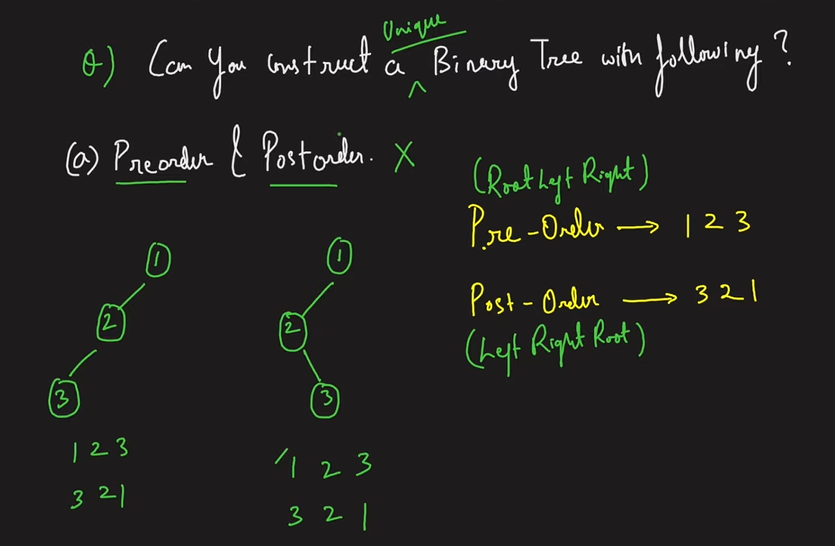


**Time complexity = O(logN)^2 because it takes logN time to calculate the height (because height in a COMPLETE BINARY TREE will not be more than logN) and we are finding/traversing the height for NOT N but RATHER logN nodes.**

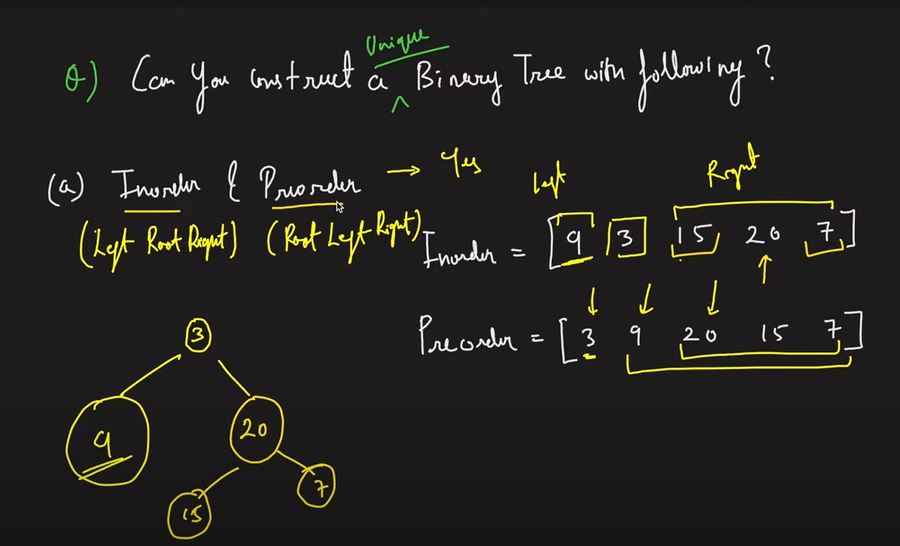
****

**The above is the worst case complete tree scenario. Here, we can see that we are traversing for all left nodes and for the right nodes we stop at level 2 itself. Hence, we are traversing the heights for LogN nodes and since height in a complete binary tree cannot be more than LogN, it takes LogN time to find the height for a particular node.**

**Constructing a unique Binary Tree**

****

Using in-order traversal is necessary as we need to know what’s on the left and right side of a root (which we are not able to verify in the above figure).

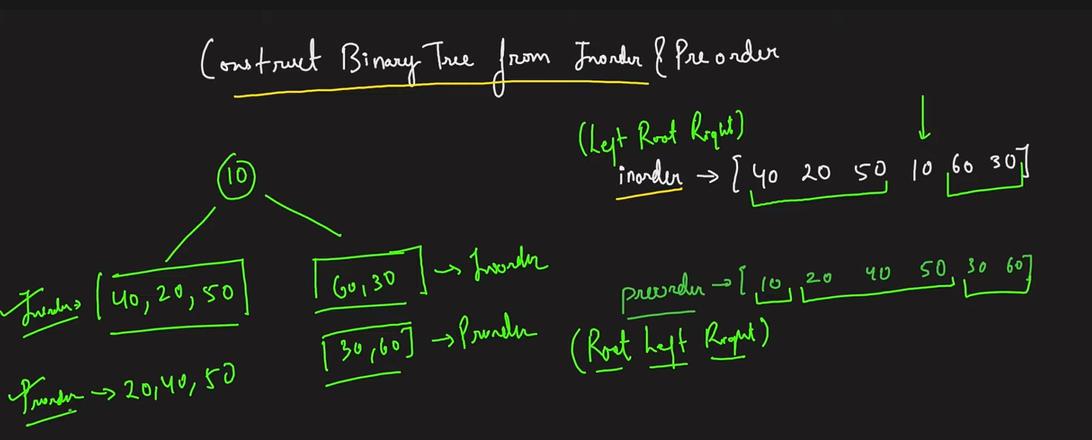
****

We get the root using pre-order and then in-order tell what’s in the left and right subtree of the root. Finally, when we use pre-order again, we get to know the subsequent sub-roots.

(After 3, 9 is considered as the left-subtree root and since we know from in-order that 9 is the only node on the left we move ahead. Now 20 is considered as the root and since using in-order we know 15,20,7 are on the right and using pre-order we see 20 as the root and 15 as the left and 7 as the right child respectively)

**We need the in-order traversal to make unique binary trees because we need to know which nodes exist on the left and right side of the root.**

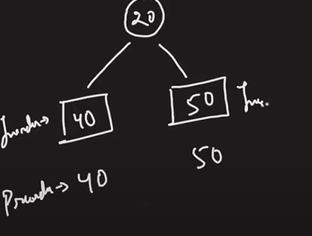
**Construct a Binary Tree from Preorder and Inorder Traversal**

****

Since pre-order starts from root, we see 10 is the root. Now using in-order we figure out the left and right subtrees. Hence, 40 20 50 belong to left and 60 30 belong to right subtree.

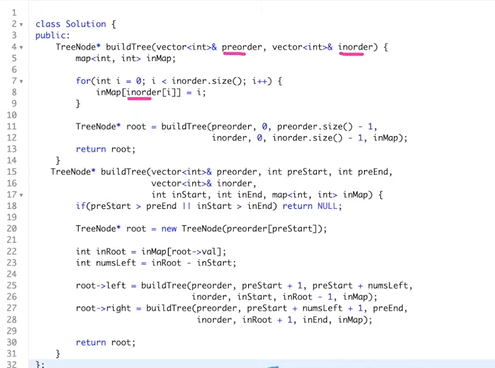
Now we have to kind of repeat what we did with the full tree. We now have to use in-order and pre-order traversals again on the left and right subtrees to generate the unique binary tree.

In the left subtree, using pre-order we see 20 is the root and now using in-order we see 40 is left and 50 is right. Hence, we generate the left subtree.

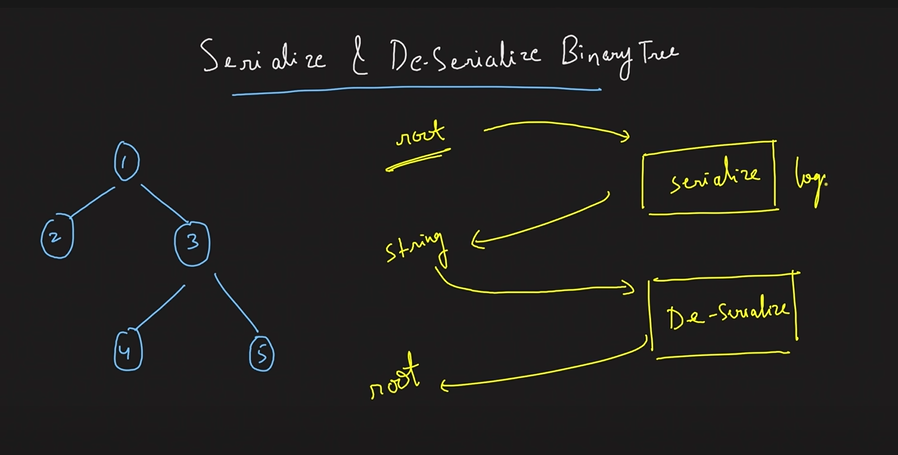


When we perform same operations on 40 and 50, we see that they are the roots in their own sub-trees and since the in-order list contains nothing except 40 and 50 now, it means they have nothing on their left and right and hence return null.



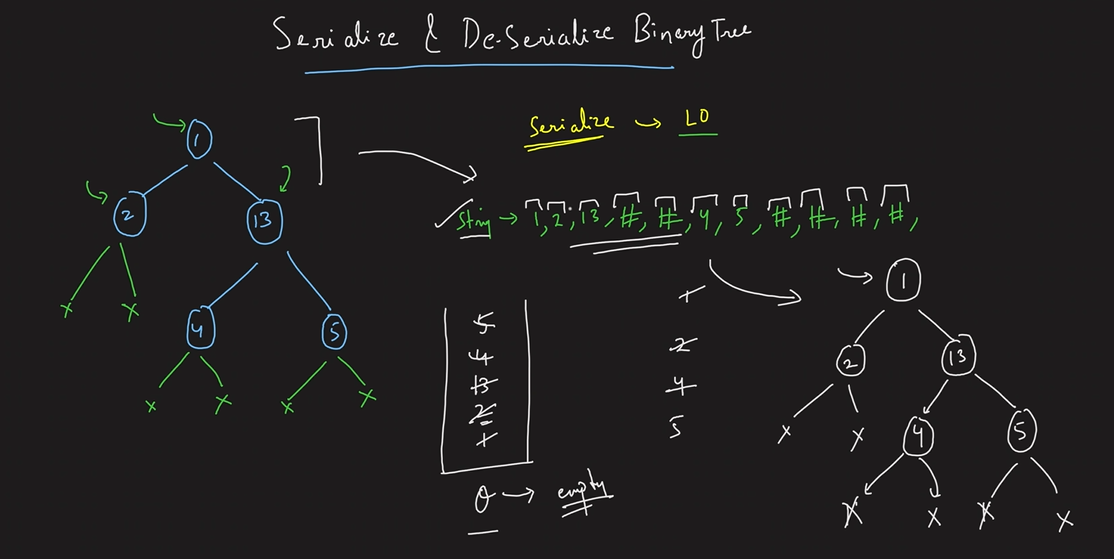


**Serialize and De-serialize Binary Tree**

****

Step 1: Using bfs (level order traversal) we make a string (if a node has no left or right child we mark it as ‘#’).

Step 2: Using the obtained string, we take the first node and add it to the queue. Next, we pop it and take 2 items from the string and mark them as the left and right child of this node. **Note that if they are ‘#’ we just mark them as left/right child else, we also add them to the queue.** Next we take the next element and continue to traverse the string further (taking 2 nodes at a time for a particular popped element from the queue since 2 nodes means left and right child (there will always be 2 children as we marked null as #)).

****

class Codec {

public:

    // Encodes a tree to a single string.

    string serialize(TreeNode\* root) {

        if(!root) return "";

        string s ="";

        queue<TreeNode\*> q;

        q.push(root);

        while(!q.empty()) {

           TreeNode\* curNode = q.front();

           q.pop();

           if(curNode==NULL) s.append("#,");

           else s.append(to\_string(curNode->val)+',');

           if(curNode != NULL){

               q.push(curNode->left);

               q.push(curNode->right);

           }

        }

        return s;

    }

    // Decodes your encoded data to tree.

    TreeNode\* deserialize(string data) {

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

        stringstream s(data);

        string str;

        getline(s, str, ',');

        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;

    }

};

**Binary Search Tree**

Left < root < right.

Left and right subtrees should themselves be bst.

If a value appears more than once, we can allows left <= root < right.

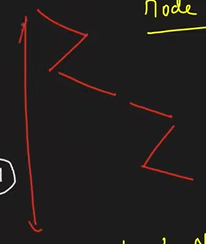
Sometimes binary trees can be degenerate (i.e., height goes upto n). However, bst’s are generally of height log(n). The lesser height helps in lowering the time complexity of search.

Search in binary tree = O(N) for inorder, or any traversal. We need to traverse each node until we find the required node.

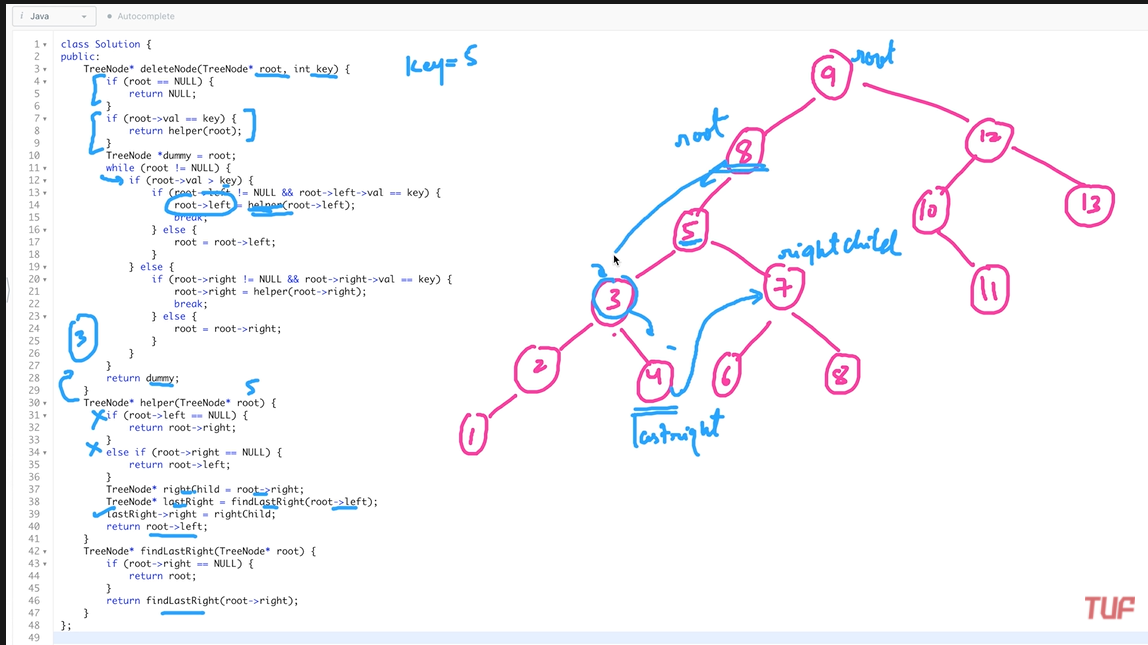
Search in bst = O(logn) as we use comparison like binary search (either left or right). Hence, we reach the node we want to find earlier as comparisons are made and direction chosen is always correct (no need to visit all nodes).

**Search in a BST**

We are not traversing each node (zig-zag) but rather making comparisons in a straight line which is equal to the height of the tree or nearly log(N).



**Delete a given node in a BST**

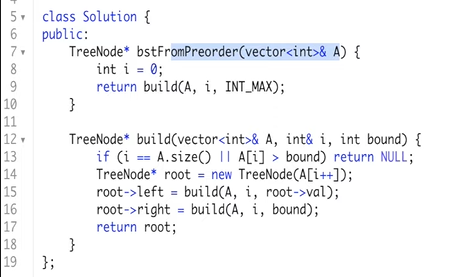


**Construct a BST from a preorder traversal**

We can generate inorder by using sorting of preorder and then make a BST. However, this will take NlogN as sorting takes NlogN.

Hence, we use the following approach wherein we take an element and if the next element is smaller we add it to its left with the upper bound as the node to which we just attached it. However, if its greater, then we check for right and take upper bound as the node above it (for example if we have to insert something on the right of 1, its upper bound will be 5 because it has to be less than 5 and it can be greater than 1 upto 5 any number).





Line 13, if array finished or value > bound we return null.

Line 15, adjusting bound for left.

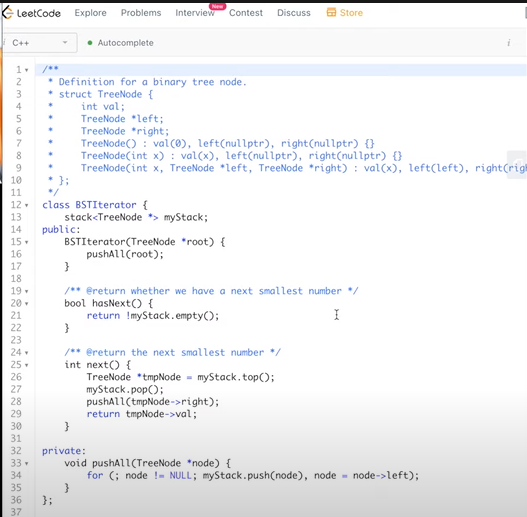
Line 16, if we go right, the bound remains unchanged.

**BST iterator**

<https://www.youtube.com/watch?v=D2jMcmxU4bs&list=PLgUwDviBIf0q8Hkd7bK2Bpryj2xVJk8Vk&index=51>

If we use inorder array we take T(N) = O(1) and S(N) = O(N). If we use stack approach we take T(N) = O(1) and S(N) = O(H) since we store H elements in a stack

Time complexity is O(1) in both cases because that’s the time it will take to fetch the required element from our data structure. To create that stack or vector the time complexity still remains O(N)



Push all is pushing all the nodes towards the left (line 28 is when we pop an element from the stack, we push all elements to right of this node (and it’s left) to the stack).

**Recover BST with 2 nodes swapped**

Using inorder traversal fetch the nodes then swap.

**Largest BST in Binary Tree**

Very important question

<https://www.youtube.com/watch?v=X0oXMdtUDwo&list=PLgUwDviBIf0q8Hkd7bK2Bpryj2xVJk8Vk&index=54>

