**Implementation**

package DSA.BST;

import java.util.\*;

// Node class represents a node in the binary search tree

class Node{

    int data;

    Node left;

    Node right;

    Node(int data){

        this.data=data;

        left=null;

        right=null;

    }

}

// BuildTree class contains methods to build, manipulate, and traverse the binary search tree

class BuildTree{

    Node root;

    // Method to insert a new node in the binary search tree

    Node insert(Node root, int data) {

        if(root==null) {

            root = new Node(data);

            return root;

        }

        if(data<root.data) {

            root.left = insert(root.left,data);

        }

        else if(data>root.data){

            root.right = insert(root.right, data);

        }

        return root;

    }

    // Method to delete a node from the binary search tree

    Node deleteFromBst(Node root,int key) {

        if(root==null) {

            return root;

        }

        if(root.data == key) {

            // 0 child

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

                return null;

            }

            // 1 child

            // left child

            if(root.left==null && root.right!=null) {

                Node temp = root.right;

                root.right = null;

                return temp;

            }

            // right child

            if(root.left!=null && root.right==null) {

                Node temp = root.left;

                root.left = null;

                return temp;

            }

            // 2 child

            if(root.left!=null && root.right!=null) {

                int mini = findMin(root.right);

                root.data = mini;

                root.right = deleteFromBst(root.right,mini);

                return root;

            }

        }

        else if(root.data > key) {

            root.left = deleteFromBst(root.left,key);

            return root;

        }

        else {

            root.right = deleteFromBst(root.right,key);

            return root;

        }

        return root;

    }

    // Method to perform level order traversal of the binary search tree

    void levelOrder(Node root) {

        if (root == null) return;

        Queue<Node> q = new LinkedList<>();

        q.add(root); // Add the root node to the queue

        while (!q.isEmpty()) { // Continue the loop until the queue is empty

            Node temp = q.poll(); // Remove and retrieve the front node from the queue

            System.out.print(temp.data + "->"); // Print the data of the current node

            // Add the left child to the queue if it exists

            if (temp.left != null) {

                q.add(temp.left);

            }

            // Add the right child to the queue if it exists

            if (temp.right != null) {

                q.add(temp.right);

            }

        }

    }

    // Method to search for a key in the binary search tree

    Node search(Node root, int key) {

        // Base Cases: root is null or key is present at root

        if (root == null || root.data == key)

            return root;

        // Key is greater than root's key

        if (root.data < key)

            return search(root.right, key);

        // Key is smaller than root's key

        return search(root.left, key);

    }

    // Method to find the minimum value in the binary search tree

    int findMin(Node root) {

        if(root==null) {

            return root.data;

        }

        Node temp = root;

        while(temp.left!=null) {

            temp=temp.left;

        }

        return temp.data;

    }

    // Method to find the maximum value in the binary search tree

    int findMax(Node root) {

        if(root==null) {

            return root.data;

        }

        Node temp = root;

        while(temp.right!=null) {

            temp=temp.right;

        }

        return temp.data;

    }

}

// Class containing the main method to test the binary search tree operations

public class BSTDriver {

    public static void main(String[] args) {

        BuildTree tr = new BuildTree();

        // Inserting elements into the binary search tree

        tr.root = tr.insert(tr.root, 50);

        tr.insert(tr.root, 20);

        tr.insert(tr.root, 70);

        tr.insert(tr.root, 10);

        tr.insert(tr.root, 30);

        tr.insert(tr.root, 90);

        tr.insert(tr.root, 110);

        // Performing level order traversal of the binary search tree

        tr.levelOrder(tr.root);

        System.out.println("\n");

        // Key to be found

        int key = 6;

        // Searching for a key in the binary search tree

        if (tr.search(tr.root, key) == null)

            System.out.println(key + " not found");

        else

            System.out.println(key + " found");

        key = 60;

        // Searching for a key in the binary search tree

        if (tr.search(tr.root, key) == null)

            System.out.println(key + " not found");

        else

            System.out.println(key + " found");

        System.out.println("\n Min : "+tr.findMin(tr.root)+" Max : "+tr.findMax(tr.root));

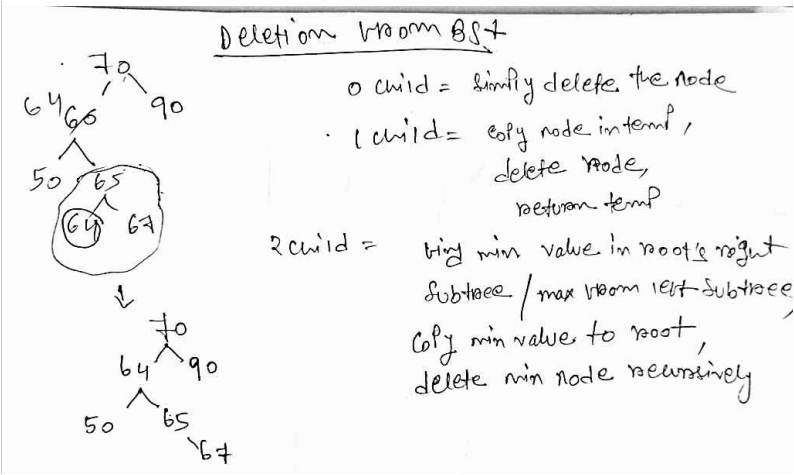
        System.out.println("After deleting : ");

        Node newRoot = tr.deleteFromBst(tr.root, 50);

        tr.levelOrder(newRoot);

    }

}



**218\_Check if a tree is a BST or not**

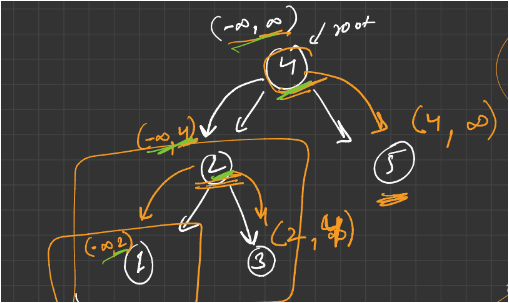
**Ap 1 : finding Inorder**

Inorder of BST is always be sorted. So, frind inorder and check if it is sorted not.

**TC : O( Nlog(N) )**

**Ap 2 : by checking range**

1. Check if data for every node falls in a range or not



**TC : O( N )**

**SC : O( H )**

boolean solve(Node root, int min, int max){

    // Base case: If the current node is null, it's a valid BST node.

    if(root==null){

        return true;

    }

    // Check if the current node's value is within the valid range (min, max).

    if(root.data>min && root.data<max){

        // Recursively check the left subtree with the new max value as the current node's value,

        // and the right subtree with the new min value as the current node's value.

        boolean left = solve(root.left, min, root.data);

        boolean right = solve(root.right, root.data, max);

        // Return true only if both left and right subtrees are valid BSTs.

        return left && right;

    }

    // If the current node's value is not within the valid range, it's not a valid BST node.

    return false;

}

//Function to check whether a Binary Tree is BST or not.

boolean isBST(Node root)

{

    // Start the recursive check with the initial range set to the minimum and maximum possible values.

    return solve(root,Integer.MIN\_VALUE, Integer.MAX\_VALUE);

}

**225\_Find Kth largest element in a BST**

public int kthLargest(Node root, int k) {

    // ArrayList to store the inorder traversal of the BST

    ArrayList<Integer> inTraversal = new ArrayList<Integer>();

    // Perform inorder traversal to populate the ArrayList with node values

    inorder(root, inTraversal);

    // Get the size of the ArrayList

    int n = inTraversal.size();

    // Return the (n - k)th element from the end of the ArrayList, which represents the kth largest element in the BST

    return inTraversal.get(n - k);

}

public static void inorder(Node root, ArrayList<Integer> inTraversal) {

    // Base case: If the current node is null, return

    if (root == null) return;

    // Recursively traverse the left subtree

    inorder(root.left, inTraversal);

    // Add the data of the current node to the ArrayList

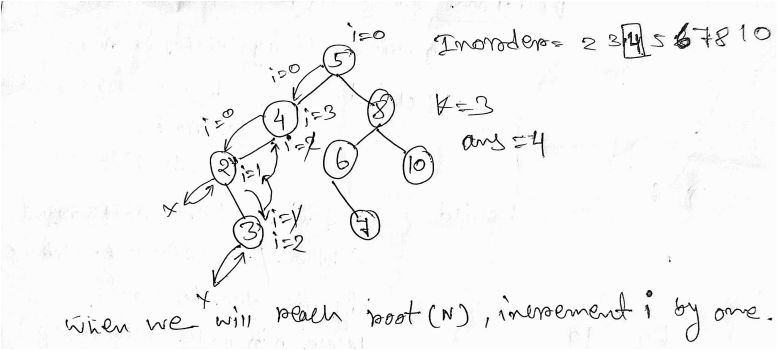
    inTraversal.add(root.data);

    // Recursively traverse the right subtree

    inorder(root.right, inTraversal);

}

**226\_Find Kth smallest element in a BST**



**TC : O( N )**

**SC : O( H )**

class Solution {

    // Function to find the Kth smallest element in the given BST

    int solve(Node root, int K, int[] i){

        // Base case: If the current node is null, return -1 indicating not found.

        if(root == null){

            return -1;

        }

        // L: Recursively search the left subtree for the Kth smallest element

        int left = solve(root.left, K, i);

        // If Kth smallest element is found in the left subtree, return it

        if(left != -1){

            return left;

        }

        // N: Increment the counter as we encounter each node

        i[0]++;

        // Check if the current node is the Kth smallest element

        if(i[0] == K){

            return root.data; // Return the data of the current node

        }

        // R: If the Kth smallest element is not found in the left subtree or the current node,

        // recursively search in the right subtree

        return solve(root.right, K, i);

    }

    // Public method to call the recursive function to find the Kth smallest element

    public int KthSmallestElement(Node root, int K) {

        int[] i = {0}; // Counter to keep track of the current position

        // Call the recursive function to find the Kth smallest element

        int ans = solve(root, K, i);

        return ans; // Return the Kth smallest element

    }

}

**By using morris traversal, SC can be optimizted to O( 1 ) ->**

public int KthSmallestElement(Node root, int K) {

    if (root == null || K <= 0) {

        return -1; // If the BST is empty or K is invalid, return -1

    }

    int count = 0; // Initialize a counter to keep track of the number of nodes visited

    Node current = root; // Start traversal from the root node

    while (current != null) {

        // If the left child of the current node is null, visit the current node

        if (current.left == null) {

            count++; // Increment the counter

            if (count == K) {

                return current.data; // If the counter reaches K, return the data of the current node

            }

            current = current.right; // Move to the right child

        } else {

            // Find the inorder predecessor of the current node

            Node predecessor = current.left;

            while (predecessor.right != null && predecessor.right != current) {

                predecessor = predecessor.right;

            }

            if (predecessor.right == null) {

                // Make the current node the right child of its inorder predecessor

                predecessor.right = current;

                current = current.left; // Move to the left child

            } else {

                // Revert the changes made to restore the original tree structure

                predecessor.right = null;

                count++; // Increment the counter

                if (count == K) {

                    return current.data; // If the counter reaches K, return the data of the current node

                }

                current = current.right; // Move to the right child

            }

        }

    }

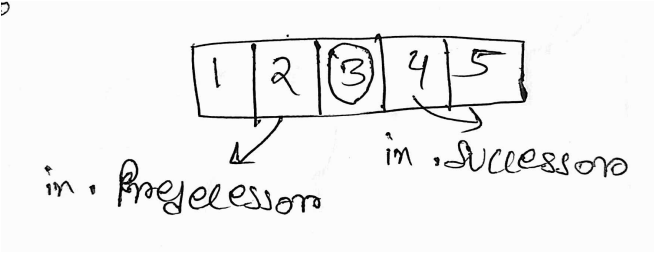
    return -1; // If K is greater than the number of nodes in the BST, return -1

}

**217\_Inorder successor and predecessor**

**Ap 1 :**

1. Find inorder and store in a ArrayList
2. Get successor and predecessor from that list



**Ap 2 : iterative method**

**TC : O( N )**

**SC : O( 1 )**

public static List<Integer> predecessorSuccessor(TreeNode root, int key) {

    // Initialize variables to store predecessor and successor

    int pred = -1;

    int succ = -1;

    // Start from the root of the tree

    TreeNode temp = root;

    // Traverse the tree until either the key is found or a leaf node is reached

    while (temp != null && temp.data != key) {

        // If the current node's data is greater than the key, update successor and move to the left subtree

        if (temp.data > key) {

            succ = temp.data;

            temp = temp.left;

        }

        // If the current node's data is less than or equal to the key, update predecessor and move to the right subtree

        else {

            pred = temp.data;

            temp = temp.right;

        }

    }

    // If the key is found, update predecessor and successor

    if (temp != null) {

        // Calculate predecessor

        if (temp.left != null) {

            TreeNode tempLeft = temp.left;

            // Traverse right until reaching the rightmost node of the left subtree

            while (tempLeft.right != null) {

                tempLeft = tempLeft.right;

            }

            pred = tempLeft.data;

        }

        // Calculate successor

        if (temp.right != null) {

            TreeNode tempRight = temp.right;

            // Traverse left until reaching the leftmost node of the right subtree

            while (tempRight.left != null) {

                tempRight = tempRight.left;

            }

            succ = tempRight.data;

        }

    }

    // Store the predecessor and successor in a list and return it

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

    result.add(pred);

    result.add(succ);

    return result;

}

**220\_LCA of 2 nodes in a BST**

// Function to find the lowest common ancestor in a BST.

Node LCA(Node root, int n1, int n2)

{

    // Traverse the BST starting from the root node

    while (root != null) {

        // If both n1 and n2 are smaller than the current node's value, move to the left subtree

        if (root.data > n1 && root.data > n2) {

            root = root.left;

        }

        // If both n1 and n2 are greater than the current node's value, move to the right subtree

        else if (root.data < n1 && root.data < n2) {

            root = root.right;

        }

        // If one of n1 or n2 is smaller and the other is greater than the current node's value,

        // or if one of n1 or n2 matches the current node's value, then this node is the lowest common ancestor

        else {

            return root;

        }

    }

    // If no common ancestor is found, return null

    return root;

}

**2Sum in BST**

**Ap 1 : brute force**

The idea is to traverse the BST in a level-order manner and let’s denote the value of the current node as A. For each node, check whether the node with value (target - A) is present in the BST or not. We can search this node in the given tree using the BST property, i.e. at any node we go to left subtree or right subtree by comparing the node value to (target - A), and if the value is found in the BST, return true. If the value is not found for any node, return false.

**TC : O( N\*H )**

**SC : O( N )**

**Ap 2 : finding inorder**

1. Find inorder and store in arraylist
2. Apply two pointer approach

**TC : O( N )**

**SC : O( N )**

static void inorder(BinaryTreeNode<Integer> root, ArrayList<Integer> inorderVal){

    if(root==null){

        return; // If the current node is null, return to backtrack

    }

    inorder(root.left, inorderVal); // Traverse left subtree recursively

    inorderVal.add(root.data); // Add current node's value to the inorder traversal list

    inorder(root.right, inorderVal); // Traverse right subtree recursively

}

public static boolean twoSumInBST(BinaryTreeNode<Integer> root, int target) {

    ArrayList<Integer> inorderVal = new ArrayList<>();

    inorder(root, inorderVal); // Generate inorder traversal of the BST

    int s = 0, e = inorderVal.size()-1;

    while(s<e){

        int sum = inorderVal.get(s)+inorderVal.get(e); // Calculate sum of current pair

        if(sum==target){

            return true; // If sum matches target, return true

        }

        else if(sum < target){

            s++; // If sum is less than target, move to next pair by increasing left pointer

        }

        else{

            e--; // If sum is greater than target, move to previous pair by decreasing right pointer

        }

    }

    return false; // If no such pair found, return false

}

**235\_Flatten BST to sorted list**

1. Find inorder and store
2. From that create sorted list

static void inorder(Node root, ArrayList<Integer> inorderVal) {

    if (root == null) {

        return;

    }

    inorder(root.left, inorderVal);

    inorderVal.add(root.data);

    inorder(root.right, inorderVal);

}

public Node flattenBST(Node root) {

    // List to store the inorder

    ArrayList<Integer> inorderVal = new ArrayList<>();

    // Perform inorder traversal

    inorder(root, inorderVal);

    // If the BST is empty, return null

    if (inorderVal.isEmpty()) {

        return null;

    }

    // Create a new root node with the first value in the inorder traversal list

    Node newRoot = new Node(inorderVal.get(0));

    // Initialize a pointer to the current node in the linked list

    Node cur = newRoot;

    // Iterate through the remaining values in the inorder traversal list

    for (int i = 1; i < inorderVal.size(); i++) {

        // Create a new node with the current value

        Node temp = new Node(inorderVal.get(i));

        // Disconnect the left child of the current node

        cur.left = null;

        // Set the right child of the current node to the new node

        cur.right = temp;

        // Move the pointer to the new node

        cur = temp;

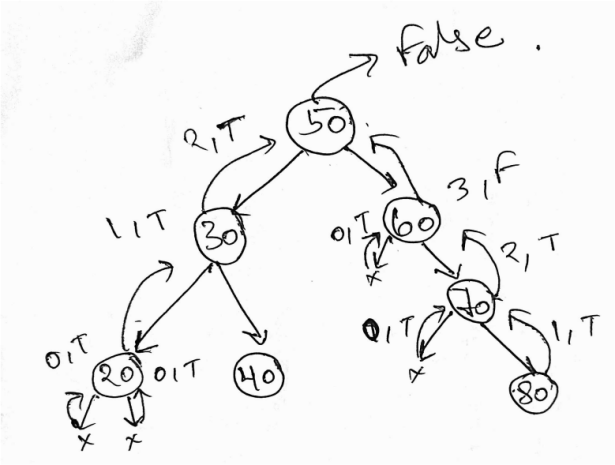
    }

    return newRoot;

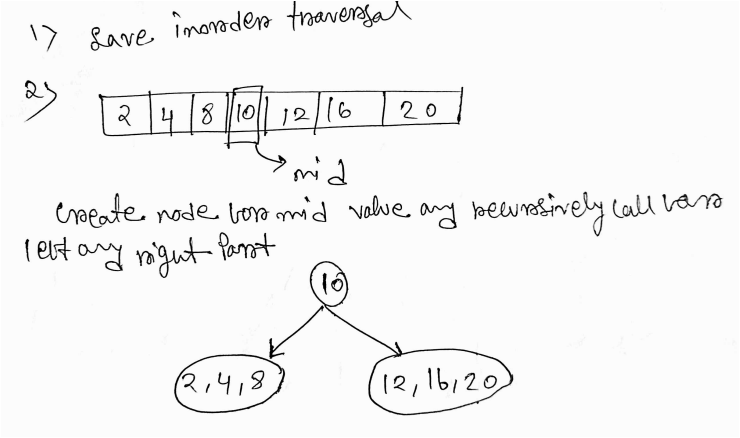
}

**Normal BST to Balanced BST**

For every node : abs( height(leftSubTree) - (height(rigthSubTree) ) <=1



**Approach :**



static void inorder(Node root, ArrayList<Integer> inorderVal) {

        if (root == null) {

            return;

        }

        inorder(root.left, inorderVal);

        inorderVal.add(root.data);

        inorder(root.right, inorderVal);

    }

    Node buildTree(int s, int e, ArrayList<Integer> inorderVal) {

        // Base case: if start index is greater than end index, return null

        if (s > e) {

            return null;

        }

        int mid = (s + e) / 2;

        // Create a new node with the value at the middle index

        Node root = new Node(inorderVal.get(mid));

        // Recursively build the left subtree using elements from the left portion of the array

        root.left = buildTree(s, mid - 1, inorderVal);

        // Recursively build the right subtree using elements from the right portion of the array

        root.right = buildTree(mid + 1, e, inorderVal);

        // Return the root of the constructed tree

        return root;

    }

    Node buildBalancedTree(Node root) {

        // ArrayList to store the inorder traversal

        ArrayList<Integer> inorderVal = new ArrayList<>();

        inorder(root, inorderVal);

        // Build a balanced binary search tree from the inorder traversal list

        return buildTree(0, inorderVal.size() - 1, inorderVal);

    }

**221\_Construct BST from preorder traversal**

**Ap 1 :** simply construct bst as already done in bst implementation

**TC : O( N^2 )**

**Ap 2 :**

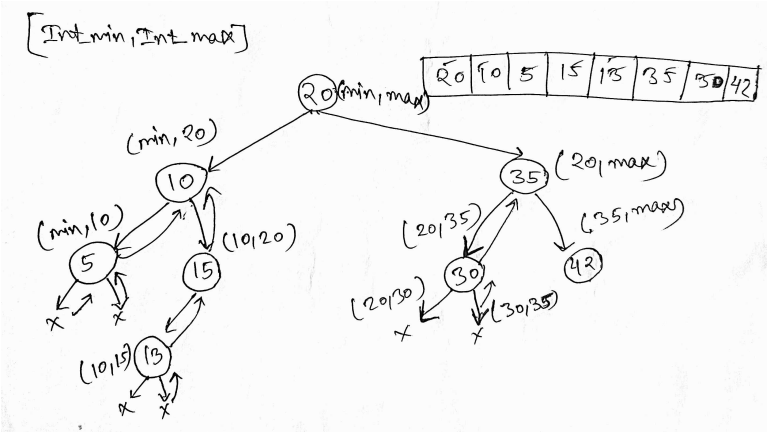
1. Find inorder from pre order by sorting preorder
2. Build tree using inorder and preorder as already done in binary tree

**TC : O( Nlog(N) )**

**Ap 3 :**

By using the concept of range checking as done in isBST question.

Checking range for every input data and inserting accordingly.



**TC : O( N )**

**SC : O( N )**

static TreeNode solve(int[] preOrder, int[] index, int min, int max){

    // Base case: If index exceeds the length of the pre-order array, return null

    if(index[0] >= preOrder.length){

        return null;

    }

    // If the current element is not within the range defined by min and max, return null

    if(preOrder[index[0]] < min || preOrder[index[0]] > max){

        return null;

    }

    // Create a new TreeNode with the current element

    TreeNode root = new TreeNode(preOrder[index[0]]);

    // Increment the index to move to the next element in the pre-order array

    index[0]++;

    // Recursively construct the left subtree with elements within the range [min, root.data)

    root.left = solve(preOrder, index, min, root.data);

    // Recursively construct the right subtree with elements within the range (root.data, max]

    root.right = solve(preOrder, index, root.data, max);

    return root;

}

public static TreeNode preOrderTree(int[] preOrder) {

    if(preOrder == null || preOrder.length == 0){

        return null;

    }

    int min = Integer.MIN\_VALUE;

    int max = Integer.MAX\_VALUE;

    // Initialize index to keep track of the current position in the pre-order array

    int[] index = {0};

    // Call the solve method to construct the binary search tree recursively

    return solve(preOrder, index, min, max);

}

**224\_Merge two BST [ V.V.V>IMP ]**

**Ap 1 :**

1. Find inorder for two bst
2. Merge two inorder list
3. Construct bst from merged inorder list

**TC : O( N+M )**

**SC : O( N+M )**

static void inorder(Node root, ArrayList<Integer> in){

    // If the root is null, return

    if(root == null){

        return;

    }

    inorder(root.left, in);

    in.add(root.data);

    inorder(root.right, in);

}

List<Integer> merge(ArrayList<Integer> a, ArrayList<Integer> b){

    // Initialize an ArrayList to store the merged result.

    List<Integer> ans = new ArrayList<>();

    int i = 0, j = 0, k = 0;

    while(i < a.size() && j < b.size()){

        if(a.get(i) <= b.get(j)){

            ans.add(a.get(i));

            i++;

            k++;

        }

        else{

            ans.add(b.get(j));

            j++;

            k++;

        }

    }

    // Add any remaining elements from the first ArrayList.

    while(i < a.size()){

        ans.add(a.get(i));

        i++;

        k++;

    }

    // Add any remaining elements from the second ArrayList.

    while(j < b.size()){

        ans.add(b.get(j));

        j++;

        k++;

    }

    return ans;

}

public List<Integer> merge(Node root1, Node root2){

    // Perform inorder traversal of both BSTs to get sorted lists of node values.

    ArrayList<Integer> in1 = new ArrayList<>();

    ArrayList<Integer> in2 = new ArrayList<>();

    inorder(root1, in1);

    inorder(root2, in2);

    // Merge the sorted lists obtained from both BSTs.

    return merge(in1, in2);

}

**Ap 2 :**

1. Convert two bst in doubly linked list( as we are changing just pointer, so SC would be O( H ) which is less than Ap 1 )
2. Merge two doubly linked list

How to convert into DLL? ( Don’t worry, recursion will take care of it)

1. convert right subtree into DLL which will return a head, say rightHead
2. Root.right = rightHead
3. Head.left = root
4. Head = root
5. Convert leftsubtree into DLL

Use this approach when the root of the new bst is said to be returned in he question.

// Function to convert a Binary Search Tree (BST) to a doubly linked list (DLL) in descending order.

void convertToDLL(TreeNode root, TreeNode head){

    // Base case: If the root is null, return as there's nothing to process.

    if(root == null){

        return;

    }

    // Convert the right subtree to DLL recursively.

    convertToDLL(root.right, head);

    // Set the right pointer of the current node to the head of the DLL.

    root.right = head;

    // If the head is not null, set the left pointer of the head to the current node.

    if(head != null){

        head.left = root;

    }

    // Update the head to the current node.

    head = root;

    // Convert the left subtree to DLL recursively.

    convertToDLL(root.left, head);

}

// Function to merge two sorted doubly linked lists (DLLs) into one sorted DLL.

TreeNode mergeDll(TreeNode head1, TreeNode head2){

    // Initialize pointers for the merged DLL and its tail.

    TreeNode head = null;

    TreeNode tail = null;

    // Merge the two DLLs while both are not empty.

    while(head1 != null && head2 != null){

        // If the value in the first DLL is smaller, append it to the merged DLL.

        if(head1.data < head2.data){

            if(head == null){

                head = head1;

                tail = head1;

                head1 = head1.right;

            }

            else{

                tail.right = head1;

                head1.left = tail;

                tail = head1;

                head1 = head1.right;

            }

        }

        // If the value in the second DLL is smaller, append it to the merged DLL.

        else{

            if(head == null){

                head = head2;

                tail = head2;

                head2 = head2.right;

            }

            else{

                tail.right = head2;

                head2.left = tail;

                tail = head2;

                head2 = head2.right;

            }

        }

    }

    // Append any remaining nodes from the first DLL.

    while(head1 != null){

        tail.right = head1;

        head1.left = tail;

        tail = head1;

        head1 = head1.right;

    }

    // Append any remaining nodes from the second DLL.

    while(head2 != null){

        tail.right = head2;

        head2.left = tail;

        tail = head2;

        head2 = head2.right;

    }

    // Return the head of the merged DLL.

    return head;

}

// Function to merge two Binary Search Trees (BSTs) and return the merged list of node values in sorted order.

public static List<Integer> mergeBST(TreeNode root1, TreeNode root2) {

    // Convert the BSTs into doubly linked lists (DLLs).

    TreeNode head1 = null;

    convertToDLL(root1, head1);

    head1.left = null; // Set the left pointer of the head to null.

    TreeNode head2 = null;

    convertToDLL(root2, head2);

    head2.left = null; // Set the left pointer of the head to null.

    // Merge the two DLLs.

    TreeNode head = mergeDll(head1, head2);

    return head;

}

**234\_Largest BST in a Binary Tree [ V.V.V.V.V IMP ]**

**Ap 1 :**

For every node call isBst function as already done before.

**TC : O( n^2 )**

**Ap 2 :**

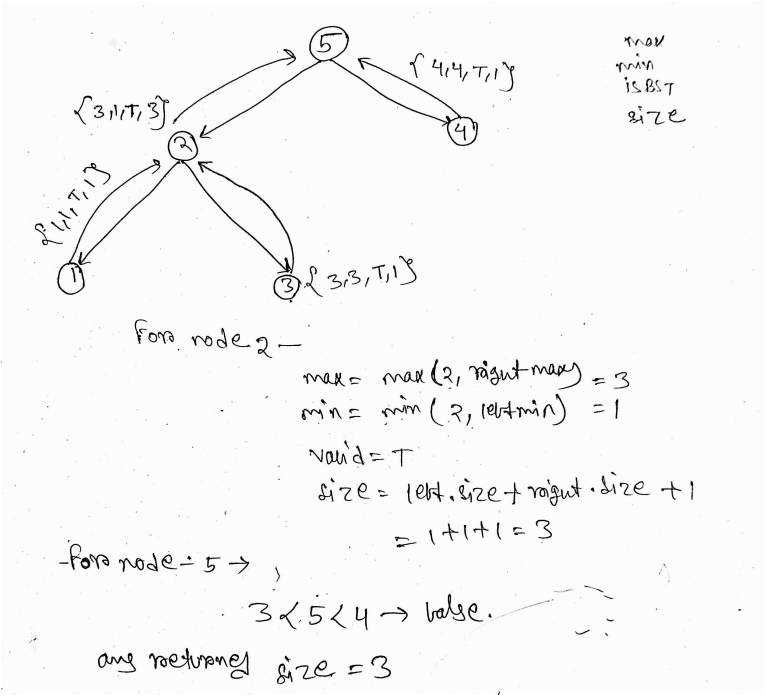
We will store information of proccessed nodes. So that we don’t need to apply same logic multiple times for each and every node.

Node -> isBst?

1. Left subtree valid Bst O( 1 )
2. Right subtree valid Bst O( 1 )
3. Max\_left<root.data<min\_right

Make a class for Information needed :

1. Max(int)
2. Min(int)
3. isBst(boolean)
4. Size(int)



class Info {

    int max;

    int min;

    boolean isBst;

    int size;

    Info() { }

    Info(int max, int min, boolean isBst, int size) {

        this.max = max;

        this.min = min;

        this.isBst = isBst;

        this.size = size;

    }

}

class Solution {

    // Method to recursively solve for each node

    static Info solve(Node root, int[] maxSize) {

        // Base case: if root is null, return Info object representing empty tree

        if(root == null) {

            return new Info(Integer.MIN\_VALUE, Integer.MAX\_VALUE, true, 0);

        }

        // Recursively solve for left and right subtrees

        Info left = solve(root.left, maxSize);

        Info right = solve(root.right, maxSize);

        // Create Info object for current node

        Info cur = new Info();

        // Update max and min values for current node

        cur.max = Math.max(root.data, right.max);

        cur.min = Math.min(root.data, left.min);

        // Calculate size of current subtree

        cur.size = left.size + right.size + 1;

        // Check if current subtree is a BST based on conditions

        if(left.isBst && right.isBst && (root.data > left.max && root.data < right.min)) {

            cur.isBst = true;

        } else {

            cur.isBst = false;

        }

        // Update maxSize if current subtree is a BST

        if(cur.isBst == true) {

            maxSize[0] = Math.max(maxSize[0], cur.size);

        }

        // Return Info object for current node

        return cur;

    }

    static int largestBst(Node root) {

        // Array to store maxSize

        int[] maxSize = {0};

        Info temp = solve(root, maxSize);

        return maxSize[0];

    }

}