**1. Binary Search Tree**

**Code :**

class Node {

int key;

Node left, right;

public Node(int key) {

this.key = key;

left = right = null;

}

}

class BinarySearchTree {

Node root;

// Insert a new key

void insert(int key) {

root = insertRec(root, key);

}

private Node insertRec(Node root, int key) {

if (root == null) {

return new Node(key);

}

if (key < root.key) {

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

} else if (key > root.key) {

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

}

return root;

}

void inorder() {

inorderRec(root);

System.out.println();

}

private void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.key + " ");

inorderRec(root.right);

}

}

}

public class CustomBST {

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

bst.insert(50);

bst.insert(30);

bst.insert(20);

bst.insert(40);

bst.insert(70);

bst.insert(60);

bst.insert(80);

bst.inorder();

}

}

Output   
20 30 40 50 60 70 80

Time Complexity :

|  |
| --- |
| Best Case : O(logn) |

|  |
| --- |
| Worst Case : O(n) |

**2. Validate a Binary Search Tree**

**Code**

class Solution {

    public boolean isValidBST(TreeNode root) {

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

            return true;

        }

        return isvalid(root , Long.MIN\_VALUE ,Long.MAX\_VALUE);

    }

    public boolean isvalid(TreeNode root , long min , long max){

        if(root == null ){

            return true;

        }

        if(root.val >=max || root.val <= min){

            return false;

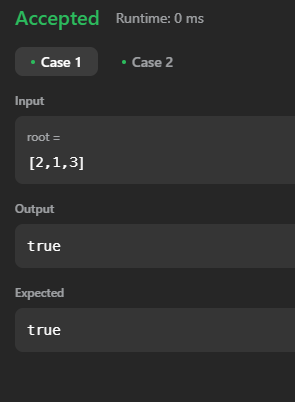
        }

        return isvalid(root.left , min , root.val) && isvalid(root.right , root.val , max);

    }

}

**Output**

****

**Time complexity:O(n)**

**3. validate given tree binary search tree or no if not make it bst**

**Code**

import java.util.ArrayList;

import java.util.Collections;

class TreeNode {

int val;

TreeNode left, right;

public TreeNode(int val) {

this.val = val;

left = right = null;

}

}

public class ValidateAndCorrectBST {

public boolean isValidBST(TreeNode root) {

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

return true;

}

return isvalid(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

public boolean isvalid(TreeNode root, long min, long max) {

if (root == null) {

return true;

}

if (root.val >= max || root.val <= min) {

return false;

}

return isvalid(root.left, min, root.val) && isvalid(root.right, root.val, max);

}

public void correctTree(TreeNode root) {

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

inorderTraversal(root, values);

Collections.sort(values);

int[] index = {0};

rebuildTree(root, values, index);

}

private void inorderTraversal(TreeNode node, ArrayList<Integer> values) {

if (node == null) return;

inorderTraversal(node.left, values);

values.add(node.val);

inorderTraversal(node.right, values);

}

private void rebuildTree(TreeNode node, ArrayList<Integer> values, int[] index) {

if (node == null) return;

rebuildTree(node.left, values, index);

node.val = values.get(index[0]++);

rebuildTree(node.right, values, index);

}

public void inorder(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

public static void main(String[] args) {

ValidateAndCorrectBST tree = new ValidateAndCorrectBST();

TreeNode root = new TreeNode(10);

root.left = new TreeNode(5);

root.right = new TreeNode(8);

root.left.left = new TreeNode(2);

root.left.right = new TreeNode(20);

tree.inorder(root);

System.out.println();

if (!tree.isValidBST(root)) {

tree.correctTree(root);

}

tree.inorder(root);

}

}

**Output**

2 5 20 10 8

2 5 8 10 20

Time Complexity : O(nlogn)

**4. Top View, Bottom View, Left View, and Right View of a Binary Search Tree (BST)**

**Code**

import java.util.\*;

class TreeNode {

int val;

TreeNode left, right;

public TreeNode(int val) {

this.val = val;

left = right = null;

}

}

public class TreeViews {

static class Pair {

TreeNode node;

int hd;

Pair(TreeNode node, int hd) {

this.node = node;

this.hd = hd;

}

}

public void topView(TreeNode root) {

if (root == null) return;

Map<Integer, Integer> map = new TreeMap<>();

Queue<Pair> queue = new LinkedList<>();

queue.add(new Pair(root, 0));

while (!queue.isEmpty()) {

Pair current = queue.poll();

int hd = current.hd;

TreeNode node = current.node;

if (!map.containsKey(hd)) {

map.put(hd, node.val);

}

if (node.left != null) {

queue.add(new Pair(node.left, hd - 1));

}

if (node.right != null) {

queue.add(new Pair(node.right, hd + 1));

}

}

for (int val : map.values()) {

System.out.print(val + " ");

}

System.out.println();

}

public void bottomView(TreeNode root) {

if (root == null) return;

Map<Integer, Integer> map = new TreeMap<>();

Queue<Pair> queue = new LinkedList<>();

queue.add(new Pair(root, 0));

while (!queue.isEmpty()) {

Pair current = queue.poll();

int hd = current.hd;

TreeNode node = current.node;

map.put(hd, node.val);

if (node.left != null) {

queue.add(new Pair(node.left, hd - 1));

}

if (node.right != null) {

queue.add(new Pair(node.right, hd + 1));

}

}

for (int val : map.values()) {

System.out.print(val + " ");

}

System.out.println();

}

public void leftView(TreeNode root) {

if (root == null) return;

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

int n = queue.size();

for (int i = 0; i < n; i++) {

TreeNode node = queue.poll();

if (i == 0) {

System.out.print(node.val + " ");

}

if (node.left != null) {

queue.add(node.left);

}

if (node.right != null) {

queue.add(node.right);

}

}

}

System.out.println();

}

public void rightView(TreeNode root) {

if (root == null) return;

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

int n = queue.size();

for (int i = 0; i < n; i++) {

TreeNode node = queue.poll();

if (i == n - 1) {

System.out.print(node.val + " ");

}

if (node.left != null) {

queue.add(node.left);

}

if (node.right != null) {

queue.add(node.right);

}

}

}

System.out.println();

}

public static void main(String[] args) {

TreeViews tree = new TreeViews();

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.left.left = new TreeNode(4);

root.left.right = new TreeNode(5);

root.right.left = new TreeNode(6);

root.right.right = new TreeNode(7);

System.out.print("Top View: ");

tree.topView(root);

System.out.print("Bottom View: ");

tree.bottomView(root);

System.out.print("Left View: ");

tree.leftView(root);

System.out.print("Right View: ");

tree.rightView(root);

}

}

**Output**

Top View: 4 2 1 3 7

Bottom View: 4 5 6 7

Left View: 1 2 4

Right View: 1 3 7

Time Complexity : O(n)

**5. Traversal**

**Code**import java.util.\*;

class TreeNode {

int val;

TreeNode left, right;

public TreeNode(int val) {

this.val = val;

left = right = null;

}

}

public class TreeTraversal {

public void inOrder(TreeNode root) {

if (root == null) return;

inOrder(root.left);

System.out.print(root.val + " ");

inOrder(root.right);

}

public void preOrder(TreeNode root) {

if (root == null) return;

System.out.print(root.val + " ");

preOrder(root.left);

preOrder(root.right);

}

public void postOrder(TreeNode root) {

if (root == null) return;

postOrder(root.left);

postOrder(root.right);

System.out.print(root.val + " ");

}

public void levelOrder(TreeNode root) {

if (root == null) return;

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

TreeNode node = queue.poll();

System.out.print(node.val + " ");

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

}

public static void main(String[] args) {

TreeTraversal tree = new TreeTraversal();

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.left.left = new TreeNode(4);

root.left.right = new TreeNode(5);

root.right.left = new TreeNode(6);

root.right.right = new TreeNode(7);

System.out.print("In-order Traversal: ");

tree.inOrder(root);

System.out.println();

System.out.print("Pre-order Traversal: ");

tree.preOrder(root);

System.out.println();

System.out.print("Post-order Traversal: ");

tree.postOrder(root);

System.out.println();

System.out.print("Level-order Traversal: ");

tree.levelOrder(root);

System.out.println();

}

}

**Output**

In-order Traversal: 4 2 5 1 6 3 7

Pre-order Traversal: 1 2 4 5 3 6 7

Post-order Traversal: 4 5 2 6 7 3 1

Level-order Traversal: 1 2 3 4 5 6 7

Time Complexity : O(n)