Binary Search Tree (*BST*)

1. AVL Tree

```java
public class AvlTree {
    static class Node {
        int data, height;
        Node left, right;

        Node(int data) {
            this.data = data;
            height = 1;
        }
    }

    public static Node root;

    public static int height(Node root) {
        if (root == null){
            return 0;
        }
        return root.height;
    }

    // Right rotate subtree rooted with y
    public static Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;
        // rotation using 3 nodes
        x.right = y;
        y.left = T2;
        // update heights
        y.height = Math.max(height(y.left), height(y.right)) + 1;
        x.height = Math.max(height(x.left), height(x.right)) + 1;
        // x is new root
        return x;
    }

    // Left rotate subtree rooted with x
    public static Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;
        // rotation using 3 nodes
        y.left = x;
        x.right = T2;
        // update heights
        x.height = Math.max(height(x.left), height(x.right)) + 1;
        y.height = Math.max(height(y.left), height(y.right)) + 1;
        // y is new root
        return y;
    }

    // Get Balance factor of node
    public static int getBalance(Node root) {
        if (root == null){
            return 0;
        }
```

```
        return height(root.left) - height(root.right);
    }

    public static Node insert(Node root, int key) {
        if (root == null){
            return new Node(key);
        }
        if (key < root.data){
            root.left = insert(root.left, key);
        }
        else if (key > root.data){
            root.right = insert(root.right, key);
        }
        else{
            return root; // Duplicate keys not allowed
        }
        // Update root height
        root.height = 1 + Math.max(height(root.left), height(root.right));
        // Get root's balance factor
        int bf = getBalance(root);
        // Left Left Case
        if (bf > 1 && key < root.left.data){
            return rightRotate(root);
        }
        // Right Right Case
        if (bf < -1 && key > root.right.data){
            return leftRotate(root);
        }
        // Left Right Case
        if (bf > 1 && key > root.left.data) {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }
        // Right Left Case
        if (bf < -1 && key < root.right.data) {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }
        return root; // returned if AVL balanced
    }

    public static void preorder(Node root) {
        if (root == null) {
            return;
        }
        System.out.print(root.data + " ");
        preorder(root.left);
        preorder(root.right);
    }

    public static void main(String[] args) {
        root = insert(root, 10);
        root = insert(root, 20);
        root = insert(root, 30);
        root = insert(root, 40);
        root = insert(root, 50);
        root = insert(root, 25);
```

```
        /*
         * AVL Tree
         * 30
         * / \
         * 20 40
         * / \ \
         * 10 25 50
         */
        System.out.println("PreOrder Traversal of AVL Tree: ");
        preorder(root);
    }
}
```

2. BST to Balanced BST

```
import java.util.ArrayList;

public class BSTtoBalancedBST {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static void preorder(Node root){
        if (root == null) {
            return;
        }
        System.out.print(root.data + " ");
        preorder(root.left);
        preorder(root.right);
    }
    public static Node balanceBST(Node root){
        // InOrder Sequence
        ArrayList <Integer> inorder = new ArrayList<>();
        getInOrder(root, inorder);
        // Sorted inOrder -> Balanced BST
        root = createBST(inorder, 0, inorder.size()-1);
        return root;
    }
    public static Node createBST(ArrayList <Integer> inorder, int st, int end){
        if (st > end) {
            return null;
        }
        int mid = (st + end) / 2;
        Node root = new Node(inorder.get(mid));
        root.left = createBST(inorder, st, mid - 1);
        root.right = createBST(inorder, mid+1, end);
        return root;
    }
    public static void getInOrder(Node root, ArrayList<Integer>inorder){
        if (root == null) {
            return;
        }
```

```
        getInOrder(root.left, inorder);
        inorder.add(root.data);
        getInOrder(root.right, inorder);
    }
    public static void main(String[] args) {
        Node root = new Node(8);
        root.left = new Node(6);
        root.left.left = new Node(5);
        root.left.left.left = new Node(3);
        root.right = new Node(10);
        root.right.right = new Node(11);
        root.right.right.right = new Node(12);
        System.out.print("Original BST: ");
        preorder(root);
        System.out.println();
        root = balanceBST(root);
        System.out.print("Balanced BST: ");
        preorder(root);
    }
}
```

3. Building BST

```
public class BuildingBST {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
        if (root.data > val) {
            // left subtree
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void inOrder(Node root) {
        if (root == null) {
            return;
        }
        inOrder(root.left);
        System.out.print(root.data+" ");
        inOrder(root.right);
    }
    public static void main(String[] args) {
        int val[] = {5, 1, 3, 4, 2, 7};
```

```
        Node root = null;
        for (int i = 0; i < val.length; i++) {
            root = insert(root, val[i]);
        }
        System.out.print("Inorder Traversal of BST: ");
        inOrder(root);
    }
}
```

4. Closing Element in BST

```
public class ClosestElementInBST {
    static int min_diff, min_diff_key;

    static class Node {
        int key;
        Node left, right;
    }

    static Node newNode(int key) {
        Node node = new Node();
        node.key = key;
        node.left = node.right = null;
        return node;
    }

    static void maxDiffUtil(Node ptr, int k) {
        if (ptr == null) {
            return;
        }

        if (ptr.key == k) {
            min_diff_key = k;
            return;
        }

        if (min_diff > Math.abs(ptr.key - k)) {
            min_diff = Math.abs(ptr.key - k);
            min_diff_key = ptr.key;
        }

        if (k < ptr.key) {
            maxDiffUtil(ptr.left, k);
        } else {
            maxDiffUtil(ptr.right, k);
        }
    }

    static int maxDiff(Node root, int k) {
        min_diff = Integer.MAX_VALUE; // Use Integer.MAX_VALUE for readability
        min_diff_key = -1;
        maxDiffUtil(root, k); // Call the utility function here
        return min_diff_key;
    }

    public static void main(String[] args) {
```

```
        Node root = newNode(9);
        root.right = newNode(17);
        root.left = newNode(4);
        root.left.left = newNode(3);
        root.left.right = newNode(6);
        root.left.right.left = newNode(5);
        root.left.right.right = newNode(7);
        root.right.right = newNode(22);
        root.right.right.left = newNode(20);

        int k = 18;
        System.out.println("Closest value to " + k + " is: " + maxDiff(root, k));
    }
}
```

5. Delete Node

```
public class DeleteNode {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
        if (root.data > val) {
            // left subtree
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void inOrder(Node root) {
        if (root == null) {
            return;
        }
        inOrder(root.left);
        System.out.print(root.data+" ");
        inOrder(root.right);
    }
    public static Node findInorderSuccessor(Node root){
        while (root.left != null) {
            root = root.left;
        }
        return root;
    }
    public static Node delete(Node root, int val) {
```

```
        if (root == null) {
            return null;
        }
        if (root.data < val) {
            root.right = delete(root.right, val);
        } else if (root.data > val) {
            root.left = delete(root.left, val);
        } else {
            // C1: Leaf Node
            if (root.left == null && root.right == null) {
                return null;
            }
            // C2: Single Child
            if (root.left == null) {
                return root.right;  // Should return right child when left is null
            }
            if (root.right == null) {
                return root.left;  // Should return left child when right is null
            }
            // C3: Both Children
            Node IS = findInorderSuccessor(root.right);
            root.data = IS.data;
            root.right = delete(root.right, IS.data);
        }
        return root;
    }

    public static void main(String[] args) {
        int val[] = {5, 1, 3, 4, 2, 7};
        Node root = null;
        for (int i = 0; i < val.length; i++) {
            root = insert(root, val[i]);
        }
        System.out.print("Inorder Traversal of BST: ");
        inOrder(root);
        System.out.println();
        root = delete(root, 1);
        System.out.print("Inorder Traversal of BST: ");
        inOrder(root);
    }
}
```

6. Kth Smallest Element in BST

```
public class KthSmallestElementinBST {
    static class Node {
        int data;
        Node left, right;
        Node(int x) {
            data = x;
            left = right = null;
        }
    }

    static int count = 0;
```

```java
    public static Node insert(Node root, int x) {
        if (root == null) {
            return new Node(x);
        }
        if (x < root.data) {
            root.left = insert(root.left, x);
        } else {
            root.right = insert(root.right, x);
        }
        return root;
    }

    public static Node kthSmallest(Node root, int k) {
        if (root == null) {
            return null;
        }

        Node left = kthSmallest(root.left, k);
        if (left != null) {
            return left;
        }

        count++;
        if (count == k) {
            return root;
        }

        return kthSmallest(root.right, k);
    }

    public static void printkthSmallest(Node root, int k) {
        count = 0; // Reset count before each search
        Node res = kthSmallest(root, k);
        if (res == null) {
            System.out.println("There are less than " + k + " nodes in the BST.");
        } else {
            System.out.println(k + "rd Smallest Element is: " + res.data);
        }
    }

    public static void main(String[] args) {
        Node root = null;
        int keys[] = {20, 8, 22, 4, 12, 10, 14};
        for (int x : keys) {
            root = insert(root, x);
        }
        int k = 3;
        printkthSmallest(root, k);
    }
}
```

7. Max Sum BST in BT

```java
public class MaxSumBSTinBT {
    static class Node {
        Node left;
        Node right;
        int data;

        Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    static class Info {
        int max;
        int min;
        boolean isBST;
        int sum;
        int currmax;

        Info(int m, int mi, boolean is, int su, int cur) {
            max = m;
            min = m;
            isBST = is;
            sum = su;
            currmax = cur;
        }

        Info() {
        }
    }

    static class INT {
        int a;
    }

    static Info MaxSumBSTUtil(Node root, INT maxsum) {
        if (root == null) {
            return new Info(Integer.MIN_VALUE, Integer.MAX_VALUE, true, 0, 0);
        }
        if (root.left == null && root.right == null) {
            maxsum.a = Math.max(maxsum.a, root.data);
            return new Info(root.data, root.data, true, root.data, maxsum.a);
        }
        Info L = MaxSumBSTUtil(root.left, maxsum);
        Info R = MaxSumBSTUtil(root.right, maxsum);
        Info BST = new Info();
        if (L.isBST && R.isBST && L.max < root.data && R.min > root.data) {
            BST.max = Math.max(root.data, Math.max(L.max, R.max));
            BST.min = Math.min(root.data, Math.min(L.min, R.min));
            maxsum.a = Math.max(maxsum.a, R.sum + L.sum + root.data);
            BST.sum = R.sum + root.data + L.sum;
            BST.currmax = maxsum.a;
            BST.isBST = true;
            return BST;
```

```
        }
        BST.isBST = false;
        BST.currmax = maxsum.a;
        BST.sum = R.sum + root.data + L.sum;
        return BST;
    }

    static int MaxSumBST(Node root) {
        INT maxsum = new INT();
        maxsum.a = Integer.MIN_VALUE;
        return MaxSumBSTUtil(root, maxsum).currmax;
    }

    public static void main(String[] args) {
/*
        5
       / \
     14   3
     /     \
    6       7
   / \
  9   1
*/
        Node root = new Node(5);
        root.left = new Node(14);
        root.right = new Node(3);
        root.left.left = new Node(6);
        root.right.right = new Node(7);
        root.left.left.left = new Node(9);
        root.left.left.right = new Node(1);
        System.out.println(MaxSumBST(root));
    }
}
```

8. Merge Two BSTs

```
import java.util.ArrayList;

public class Merge2BST {
    static class Node{
        int data;
        Node left;
        Node right;
        public Node(int data){
            this.data = data;
            this.left = this.right = null;
        }
    }
    public static void getInOrder(Node root, ArrayList<Integer> arr){
        if (root == null) {
            return;
        }
        getInOrder(root.left, arr);
        arr.add(root.data);
        getInOrder(root.right, arr);
    }
```

```java
public static Node createBST(ArrayList <Integer> arr, int st, int end){
    if (st > end) {
        return null;
    }
    int mid = (st+end)/2;
    Node root = new Node(arr.get(mid));
    root.left = createBST(arr, st, mid - 1);
    root.right = createBST(arr, mid + 1, end);
    return root;
}
public static void preorder(Node root){
    if (root == null) {
        return;
    }
    System.out.print(root.data + " ");
    preorder(root.left);
    preorder(root.right);
}
public static Node mergeBSTs(Node root1, Node root2){
    // S1: Inorder Sequence = Sorted_1
    ArrayList <Integer> arr1 = new ArrayList<>();
    getInOrder(root1, arr1);

    // S2: Inorder Sequence = Sorted_2
    ArrayList <Integer> arr2 = new ArrayList<>();
    getInOrder(root2, arr2);

    // Merge
    int i=0, j=0;
    ArrayList <Integer> finalArr = new ArrayList<>();
    while (i < arr1.size() && j < arr2.size()) {
        if (arr1.get(i) <= arr2.get(j)) {
            finalArr.add(arr1.get(i));
            i++;
        } else{
            finalArr.add(arr2.get(j));
            j++;
        }
    }
    while (i < arr1.size()) {
        finalArr.add(arr1.get(i));
        i++;
    }
    while (j < arr2.size()) {
        finalArr.add(arr2.get(j));
        j++;
    }
    // Sorted ArrayList -> Balanced BST
    return createBST(finalArr, 0, finalArr.size()-1);
}
public static void main(String[] args) {
    // BST-1
    Node root1 = new Node(2);
    root1.left = new Node(1);
    root1.right = new Node(4);
    System.out.print("BST 1: ");
    preorder(root1);
```

```
        System.out.println();
        // BST-2
        Node root2 = new Node(9);
        root2.left = new Node(3);
        root2.right = new Node(12);
        System.out.print("BST 2: ");
        preorder(root2);
        System.out.println();
        // Merging
        Node root = mergeBSTs(root1, root2);
        System.out.print("Merging Both BSTs: ");
        preorder(root);
    }
}
```

9. Mirroring BST

```
public class MirroringBST {
    static class Node{
        int data;
        Node left;
        Node right;
        public Node(int data){
            this.data = data;
            this.left = this.right = null;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
        if (root.data > val) {
            // left subtree
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void preOrder(Node root) {
        if (root == null) {
            return;
        }
        System.out.print(root.data+" ");
        preOrder(root.left);
        preOrder(root.right);
    }
    public static Node createMirror(Node root){
        if (root == null) {
            return null;
        }
        Node leftMirror = createMirror(root.left);
        Node rightMirror = createMirror(root.right);
        root.left = rightMirror;
```

```
        root.right = leftMirror;
        return root;
    }
    public static void main(String[] args) {
        Node root = new Node(8);
        root.left = new Node(5);
        root.right = new Node(10);
        root.left.left = new Node(3);
        root.left.right = new Node(6);
        root.right.right = new Node(11);
        System.out.print("Original BST: ");
        preOrder(root);
        System.out.println();
        root = createMirror(root);
        System.out.print("Mirroring BST: ");
        preOrder(root);
    }
}
```

10. Print in Range

```
public class PrintInRange {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
        if (root.data > val) {
            // left subtree
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void inOrder(Node root) {
        if (root == null) {
            return;
        }
        inOrder(root.left);
        System.out.print(root.data+" ");
        inOrder(root.right);
    }
    public static void printInrange(Node root, int k1, int k2) {
        if (root == null) {
            return;
        }
```

```
        // Only recurse left if the root's data is greater than k1
        if (root.data > k1) {
            printInrange(root.left, k1, k2);
        }
        // Print the root's data if it is within the range
        if (root.data >= k1 && root.data <= k2) {
            System.out.print(root.data + " ");
        }
        // Only recurse right if the root's data is less than k2
        if (root.data < k2) {
            printInrange(root.right, k1, k2);
        }
    }

    public static void main(String[] args) {
        int val[] = {5, 1, 3, 4, 2, 7};
        Node root = null;
        for (int i = 0; i < val.length; i++) {
            root = insert(root, val[i]);
        }
        System.out.print("Inorder Traversal of BST: ");
        inOrder(root);
        System.out.println();
        System.out.print("Printing in Given Range: ");
        printInrange(root, 3, 5);
    }
}
```

11. Range Sum of BST

```
import java.util.LinkedList;
import java.util.Queue;

public class RangeSumofBST {
    static class Node{
        int val;
        Node left, right;
    }
    static Node newNode(int item){
        Node temp = new Node();
        temp.val = item;
        temp.left = temp.right = null;
        return temp;
    }
    static int sum = 0;
    static int rangeSumBST(Node root, int low, int high){
        if (root == null) {
            return 0;
        }
        Queue <Node> q = new LinkedList<Node>();
        q.add(root);
        while (q.isEmpty() == false) {
            Node curr = q.peek();
            q.remove();
            if (curr.val >= low && curr.val <= high) {
                sum += curr.val;
```

```
            }
            if (curr.left != null && curr.val > low) {
                q.add(curr.left);
            }
            if (curr.right != null && curr.val < high) {
                q.add(curr.right);
            }
        }
        return sum;
    }
    static Node insert(Node node, int data){
        if (node == null) {
            return newNode(data);
        }
        if (data <= node.val) {
            node.left = insert(node.left, data);
        } else{
            node.right = insert(node.right, data);
        }
        return node;
    }
    public static void main(String[] args) {
        Node root = null;
        root = insert(root, 10);
        insert(root, 5);
        insert(root, 15);
        insert(root, 3);
        insert(root, 7);
        insert(root, 18);
        int L=7, R=15;
        System.out.print("Range between Nodes " + L + " & " + R + " Sum is: " + rangeSumBST(root,
L, R));
    }
}
```

12. Red Black Tree

```
public class RedBlackTree {
    public Node root; // Root node

    public RedBlackTree() {
        root = null;
    }

    class Node {
        int data;
        Node left, right, parent;
        char colour; // 'R' or 'B'

        Node(int data) {
            this.data = data;
            this.left = this.right = this.parent = null;
            this.colour = 'R';
        }
    }
```

```
// Flags to track rotation cases
boolean ll = false, rr = false, lr = false, rl = false;

// Function to perform left rotation
Node rotateLeft(Node node) {
    Node x = node.right;
    Node y = x.left;
    x.left = node;
    node.right = y;
    node.parent = x;
    if (y != null) y.parent = node;
    return x;
}

// Function to perform right rotation
Node rotateRight(Node node) {
    Node x = node.left;
    Node y = x.right;
    x.right = node;
    node.left = y;
    node.parent = x;
    if (y != null) y.parent = node;
    return x;
}

// Helper function for insertion
Node insertHelp(Node root, int data) {
    boolean redConflict = false;

    if (root == null) return new Node(data);

    if (data < root.data) {
        root.left = insertHelp(root.left, data);
        root.left.parent = root;

        if (root != this.root && root.colour == 'R' && root.left.colour == 'R') {
            redConflict = true;
        }
    } else {
        root.right = insertHelp(root.right, data);
        root.right.parent = root;

        if (root != this.root && root.colour == 'R' && root.right.colour == 'R') {
            redConflict = true;
        }
    }

    // Rotations and recolouring
    if (ll) {
        root = rotateLeft(root);
        root.colour = 'B';
        root.left.colour = 'R';
        ll = false;
    } else if (rr) {
        root = rotateRight(root);
        root.colour = 'B';
        root.right.colour = 'R';
```

```
            rr = false;
        } else if (rl) {
            root.right = rotateRight(root.right);
            root.right.parent = root;
            root = rotateLeft(root);
            root.colour = 'B';
            root.left.colour = 'R';
            rl = false;
        } else if (lr) {
            root.left = rotateLeft(root.left);
            root.left.parent = root;
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            lr = false;
        }

        // Handle red conflict
        if (redConflict) {
            if (root.parent.right == root) {
                if (root.parent.left == null || root.parent.left.colour == 'B') {
                    if (root.left != null && root.left.colour == 'R') rl = true;
                    else if (root.right != null && root.right.colour == 'R') ll = true;
                } else {
                    root.parent.left.colour = 'B';
                    root.colour = 'B';
                    if (root.parent != this.root) root.parent.colour = 'R';
                }
            } else {
                if (root.parent.right == null || root.parent.right.colour == 'B') {
                    if (root.left != null && root.left.colour == 'R') rr = true;
                    else if (root.right != null && root.right.colour == 'R') lr = true;
                } else {
                    root.parent.right.colour = 'B';
                    root.colour = 'B';
                    if (root.parent != this.root) root.parent.colour = 'R';
                }
            }
        }
        return root;
    }

    // Function to insert data into the tree
    public void insert(int data) {
        if (root == null) {
            root = new Node(data);
            root.colour = 'B';
        } else {
            root = insertHelp(root, data);
        }
    }

    // Helper function to perform inorder traversal
    void inorderTraversalHelper(Node node) {
        if (node != null) {
            inorderTraversalHelper(node.left);
            System.out.print(node.data + " ");
```

```
            inorderTraversalHelper(node.right);
        }
    }

    // Function to perform inorder traversal
    public void inorderTraversal() {
        inorderTraversalHelper(root);
        System.out.println();
    }

    // Helper function to print the tree
    void printTreeHelper(Node root, int space) {
        if (root != null) {
            space += 10;
            printTreeHelper(root.right, space);
            System.out.println();
            for (int i = 10; i < space; i++) System.out.print(" ");
            System.out.println(root.data);
            printTreeHelper(root.left, space);
        }
    }

    // Function to print the tree
    public void printTree() {
        printTreeHelper(root, 0);
    }

    public static void main(String[] args) {
        RedBlackTree tree = new RedBlackTree();
        int[] arr = {1, 4, 6, 3, 5, 7, 8, 2, 9};
        for (int data : arr) {
            tree.insert(data);
            tree.inorderTraversal();
        }
        tree.printTree();
    }
}
```

13. Root to Leaf Path

```
import java.util.ArrayList;

public class RootToLeafPath {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
```

```
        if (root.data > val) {
            // left subtree
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void printRoot2leaf(Node root, ArrayList < Integer> path) {
        if (root == null) {
            return;
        }
        path.add(root.data);
        if (root.left == null && root.right == null) {
            printPath(path);
        }
        printRoot2leaf(root.left, path);
        printRoot2leaf(root.right, path);
        path.remove(path.size()-1);
    }
    public static void printPath(ArrayList<Integer> path){
        for(int i=0; i < path.size(); i++){
            System.out.print(path.get(i) + "->");
        }
        System.out.println("null");
    }
    public static void main(String[] args) {
        int val[] = {5, 1, 3, 4, 2, 7};
        Node root = null;
        for (int i = 0; i < val.length; i++) {
            root = insert(root, val[i]);
        }
        System.out.println("Path from Root to all Leaf nodes: ");
        printRoot2leaf(root, new ArrayList<>());
    }
}
```

14. Search in BST

```
public class SearchInBST {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
        if (root.data > val) {
            // left subtree
```

```
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void inOrder(Node root) {
        if (root == null) {
            return;
        }
        inOrder(root.left);
        System.out.print(root.data+" ");
        inOrder(root.right);
    }
    public static boolean search(Node root, int key){
        if (root == null) {
            return false;
        }
        if (root.data == key) {
            return true;
        }
        if (root.data > key) {
            return search(root.left, key);
        } else {
            return search(root.right, key);
        }
    }
    public static void main(String[] args) {
        int val[] = {5, 1, 3, 4, 2, 7};
        Node root = null;
        for (int i = 0; i < val.length; i++) {
            root = insert(root, val[i]);
        }
        System.out.print("Inorder Traversal of BST: ");
        inOrder(root);
        System.out.println();
        if (search(root, 7)) {
            System.out.println("KEY FOUND");
        } else {
            System.out.println("KEY NOT FOUND");
        }
    }
}
```

15. Size of Largest BST in BT

```
public class SizeOfLargestBSTinBT {
    static class Node {
        int data;
        Node left;
        Node right;
        Node(int data) {
            this.data = data;
        }
    }
```

```java
    static class Info {
        boolean isBST;
        int size;
        int min;
        int max;
        public Info(boolean isBST, int size, int min, int max) {
            this.isBST = isBST;
            this.size = size;
            this.max = max;
            this.min = min;
        }
    }

    public static int maxBST = 0;

    public static Info largestBST(Node root) {
        if (root == null) {
            return new Info(true, 0, Integer.MAX_VALUE, Integer.MIN_VALUE);
        }

        Info leftInfo = largestBST(root.left);
        Info rightInfo = largestBST(root.right);

        int size = leftInfo.size + rightInfo.size + 1;
        int min = Math.min(root.data, Math.min(leftInfo.min, rightInfo.min));
        int max = Math.max(root.data, Math.max(leftInfo.max, rightInfo.max));

        if (leftInfo.isBST && rightInfo.isBST && root.data > leftInfo.max && root.data <
rightInfo.min) {
            maxBST = Math.max(maxBST, size);
            return new Info(true, size, min, max);  // Return the correct Info object here
        }

        return new Info(false, size, min, max);
    }

    public static void main(String[] args) {
        Node root = new Node(50);
        root.left = new Node(30);
        root.left.left = new Node(5);
        root.left.right = new Node(20);
        root.right = new Node(60);
        root.right.left = new Node(45);
        root.right.right = new Node(70);
        root.right.right.left = new Node(65);
        root.right.right.right = new Node(80);

        largestBST(root);
        System.out.println("Largest BST size: " + maxBST);
    }
}
```

16. Sorted Array to Balanced BST

```java
public class SortedArrayToBalancedBST {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static void preorder(Node root){
        if (root == null) {
            return;
        }
        System.out.print(root.data + " ");
        preorder(root.left);
        preorder(root.right);
    }
    public static Node createBST(int arr[], int st, int end){
        if (st > end) {
            return null;
        }
        int mid = (st+end)/2;
        Node root = new Node(arr[mid]);
        root.left = createBST(arr, st, mid - 1);
        root.right = createBST(arr, mid + 1, end);
        return root;
    }
    public static void main(String[] args) {
        int arr[] = {3, 5, 6, 8, 10, 11, 12};
        /*          8
                  /       \
                 5         11
               /  \       /  \
              3    6     10  12
        EXPECTED BST
         */
        Node root = createBST(arr, 0, arr.length-1);
        System.out.print("PreOrder Traversal of BST: ");
        preorder(root);
    }
}
```

17. Two Sum BSTs

```java
import java.util.Stack;

public class TwoSumBSTs {
    static class Node {
        int data;
        Node left, right;

        public Node(int data) {
            this.data = data;
            left = right = null;
        }
```

```
}

static Node root1;
static Node root2;

static int countPairs(Node root1, Node root2, int x) {
    if (root1 == null || root2 == null) {
        return 0;
    }

    Stack<Node> st1 = new Stack<>();
    Stack<Node> st2 = new Stack<>();
    Node top1, top2;
    int count = 0;

    while (true) {
        // Traverse the first BST in inorder
        while (root1 != null) {
            st1.push(root1);
            root1 = root1.left;
        }

        // Traverse the second BST in reverse inorder
        while (root2 != null) {
            st2.push(root2);
            root2 = root2.right;
        }

        // If either stack is empty, break
        if (st1.empty() || st2.empty()) {
            break;
        }

        top1 = st1.peek();
        top2 = st2.peek();

        // Check if the current pair's sum equals x
        if ((top1.data + top2.data) == x) {
            count++;
            st1.pop();
            st2.pop();
            root1 = top1.right;
            root2 = top2.left;
        } else if ((top1.data + top2.data) < x) {
            // If sum is less than x, move to the next larger value in root1
            st1.pop();
            root1 = top1.right;
        } else {
            // If sum is greater than x, move to the next smaller value in root2
            st2.pop();
            root2 = top2.left;
        }
    }

    return count;
}
```

```
    public static void main(String[] args) {
        root1 = new Node(5);
        root1.left = new Node(3);
        root1.right = new Node(7);
        root1.left.left = new Node(2);
        root1.left.right = new Node(4);
        root1.right.left = new Node(6);
        root1.right.right = new Node(8);

        root2 = new Node(10);
        root2.right = new Node(15);
        root2.left = new Node(6);
        root2.left.left = new Node(3);
        root2.left.right = new Node(8);
        root2.right.left = new Node(11);
        root2.right.right = new Node(18);

        int x = 16;
        System.out.println("Pairs: " + countPairs(root1, root2, x));
    }
}
```

18. Validate BST

```
public class ValidateBST {
    static class Node{
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
        }
    }
    public static Node insert(Node root, int val){
        if (root == null) {
            root = new Node(val);
            return root;
        }
        if (root.data > val) {
            // left subtree
            root.left = insert(root.left, val);
        } else {
            // right subtree
            root.right = insert(root.right, val);
        }
        return root;
    }
    public static void inOrder(Node root) {
        if (root == null) {
            return;
        }
        inOrder(root.left);
        System.out.print(root.data+" ");
        inOrder(root.right);
    }
```

```java
    public static boolean isValidBST(Node root, Node min, Node max){
        if (root == null) {
            return true;
        }
        if (min != null && root.data <= min.data) {
            return false;
        }
        else if (max != null && root.data >= max.data) {
            return false;
        }
        return isValidBST(root.left, min, root) && isValidBST(root.right, root, max);
    }
    public static void main(String[] args) {
        int val[] = {1, 1, 1};
        Node root = null;
        for(int i = 0; i < val.length; i++){
            root = insert(root, val[i]);
        }
        inOrder(root);
        System.out.println();
        // Checking valid BST
        if (isValidBST(root, null, null)) {
            System.out.println("Valid");
        } else {
            System.out.println("Not Valid");
        }
    }
}
```