```
import java.util.Queue;
import java.util.ArrayDeque;
import java.util.Stack;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.LinkedList;
import java.util.List;

public class CompleteBinarySearchTree {

    /**
     * static BufferedReader to take input - output from the Console
     * Window.
     */
    static BufferedReader input = new BufferedReader(new InputStreamReader(System.
in));


    /**
     * method to initialise List by user given Values
     * @param keys
     * @throws NumberFormatException
     * @throws IOException
     */
    private static void Input(List<Integer> keys) throws NumberFormatException,
IOException {
        boolean flag = true;
        while (flag) {
            keys.add(Integer.parseInt(input.readLine()));
            System.out.println("\nAdd more Values (1/0) :");
            int choice = Integer.parseInt(input.readLine());
            if (choice == 0)
                flag = false;
        }
    }

    private static boolean IsBst(Node root) {
        if (util_function(root, Integer.MIN_VALUE, Integer.MAX_VALUE))
            return true;
        return false;
    }


    /**
     * this will return true if the given tree is BST else return false
     * @param root
     * @param min
     * @param max
```

```
     * @return
     */
    private static boolean util_function(Node root, int min, int max) {
        if (root == null)
            return true;
        if (root.item < min || root.item > max)
            return false;
        if (util_function(root.left, min, root.item - 1))
            if (util_function(root.right, root.item + 1, max))
                return true;
        return false;
    }

    /**
     * this method will takes two nodes and return List in LvlOdr
     * of the values present in the two BinarySearchTree
     * @param root1
     * @param root2
     * @return
     */
    private static List<Integer> merge2BST(Node root1, Node root2) {
        List<Integer> values = new LinkedList<>();
        Node replica_ofRoot1 = root1, replica_ofRoot2 = root2;
        Queue<Node> q = new ArrayDeque<>();
        q.add(replica_ofRoot1);

        while (!q.isEmpty()) {
            replica_ofRoot1 = q.poll();
            values.add(replica_ofRoot1.item);
            if (replica_ofRoot1.left != null)
                q.add(replica_ofRoot1.left);
            if (replica_ofRoot1.right != null)
                q.add(replica_ofRoot1.right);
        }
        q.clear();
        q.add(replica_ofRoot2);

        while (!q.isEmpty()) {
            replica_ofRoot2 = q.poll();
            values.add(replica_ofRoot2.item);
            if (replica_ofRoot2.left != null)
                q.add(replica_ofRoot2.left);
            if (replica_ofRoot2.right != null)
                q.add(replica_ofRoot2.right);
        }
        return values;
    }

    /**
```

```
     * this functions will return true if the requested  items is persent
     * in BinarySearchTree
     * @param root
     * @param item
     * @return
     */
    private static boolean IsPresent(Node root, int item) {
        if (root == null)
            return false;
        if (root.item == item)
            return true;
        else if (root.item > item)
            return IsPresent(root.left, item);
        else if (root.item < item)
            return IsPresent(root.right, item);
        return false;
    }

    /**
     * this functions will return the requested element from
     * the BinarySearchTree
     * @param root
     * @param key
     * @return
     */
    private static Node delete(Node root, int key) {
        if (root == null)
            return root;
        else if (root.item > key)
            root.left = delete(root.left, key);
        else if (root.item < key)
            root.right = delete(root.right, key);
        else {
            if (root.left == null)
                return root.right;
            else if (root.right == null)
                return root.left;
            root.item = getMin(root.right);
            root.right = delete(root.right, root.item);
        }
        return root;
    }


    /**
     * this will print the RightView of BinarySearchTree
     * @param root
     */
    private static void print_right_view(Node root) {
```

```
    if (root == null)
        return;
    Queue<Node> q = new ArrayDeque<>();
    q.add(root);
    while (!q.isEmpty()) {
        int size = q.size();
        for (int i = 1; i <= size; i++) {
            Node tmp = q.poll();
            if (i == 1)
                System.out.print(tmp.item + " ");
            if (tmp.right != null)
                q.add(tmp.right);
            if (tmp.left != null)
                q.add(tmp.left);
        }
    }
}

private static boolean IsMirrorBst(Node root, Node __root) {
    if ((root == null && __root != null) || (root != null && __root == null))
        return false;
    if (root == null && __root == null)
        return true;
    return ((root.item == __root.item) && IsMirrorBst(root.left, __root.right)
            && IsMirrorBst(root.right, __root.left));
}


/**
 * this will print the LeftView of BinarySearchTree
 * @param root
 */
private static void print_left_view(Node root) {
    if (root == null)
        return;
    Queue<Node> q = new ArrayDeque<>();
    q.add(root);
    while (!q.isEmpty()) {
        int size = q.size();
        for (int i = 1; i <= size; i++) {
            Node tmp = q.poll();
            if (i == 1)
                System.out.print(tmp.item + " ");
            if (tmp.left != null)
                q.add(tmp.left);
            if (tmp.right != null)
                q.add(tmp.right);
        }
    }
```

```
}

/**
 * this function will return the Height of BinarySearchTree
 * @param root
 * @return
 */
private static int getHeight(Node root) {
    if (root == null)
        return 0;
    int left = getHeight(root.left);
    int right = getHeight(root.right);
    return Math.max(left, right) + 1;
}


/**
 * This will return true is both BinarySearchTrees are same,
 * else return false.
 * @param root
 * @param __root
 * @return
 */
private static boolean IsIdentical(Node root, Node __root) {
    if ((root == null && __root != null) || (root != null) && (__root == null))
        return false;
    if (root == null && __root == null)
        return true;
    return (root.item == __root.item) && IsIdentical(root.left, __root.left)
            && IsIdentical(root.right, __root.right);
}


/**
 * Driver Code main method of BinarySearchTree
 * @param args
 * @throws IOException
 */
public static void main(String[] args) throws IOException {
    System.out.println("Enter values for bst :");
    List<Integer> keys = new LinkedList<>();
    Input(keys);
    Node root = null;

    for (Integer key : keys)
        root = add_items(root, key);
    print_in_order(root);
    System.out.println();
    print_pre_order(root);
```

```
        System.out.println();
        print_post_order(root);
        System.out.println();
        pattern(root);
        System.out.println();
        pattern1(root);
        System.out.println();
        pattern2(root);
        System.out.println("\nMAX from ROOT :" + getMax(root) + "\nMIN from ROOT :" +
getMin(root)
                + "\nCOUNT of LEAF NODES :" + getLeafCount(root) + "\nHeight from
ROOT IS :" + getHeight(root));
        Node root1 = null;
        System.out.println("\nEnter values for sec. tree :");
        keys.clear();
        Input(keys);

        for (Integer key : keys)
            root1 = add_items(root, key);
        List<Integer> items = merge2BST(root, root1);
        Node mergedBstRoot = null;
        for (Integer item : items)
            mergedBstRoot = add_items(mergedBstRoot, item);
        print_in_order(mergedBstRoot);
        System.out.println("\nBoth Trees are :" + IsIdentical(root, root1));
        System.out.println("\nLeft View :");
        print_left_view(root);
        System.out.println("\nRight View :");
        print_right_view(root);
        System.out.println("\nBST STATUS :" + IsBst(root) + "\nIsMirror STATUS :" +
IsMirrorBst(root, root1));
        System.out.println("\nTree after Deletion :" + delete(root, Integer.parseInt(input.
readLine())));
        System.out.println("\nEnter Search Value " + IsPresent(root, Integer.parseInt(input.
readLine())));
        System.out.println("\nConverting BinarySearchTree to DoublyLinkedList :");
        PrintList(bstToDll(root));

        input.close();
    }

    /**
     * this will return the count of leaf Nodes
     * @param root
     * @return
     */
    private static int getLeafCount(Node root) {
        if (root == null)
            return 0;
```

```
    if (root.left == null && root.right == null)
        return 1;
    return getLeafCount(root.left) + getLeafCount(root.right);
}


/**
 * this will return the min value in the BinarySearchTree from
 * the given root node
 * @param root
 * @return
 */
private static int getMin(Node root) {
    if (root == null)
        return Integer.MAX_VALUE;
    Node replica = root;
    int ans = replica.item;
    while (replica != null) {
        ans = Integer.max(ans, replica.item);
    }
    return ans;
}


/**
 * this will return the max value in the BinarySearchTree from
 * the given root node
 * @param root
 * @return
 */
private static int getMax(Node root) {
    if (root == null)
        return Integer.MIN_VALUE;
    Node replica = root;
    int ans = root.item;
    while (replica != null) {
        ans = Integer.max(ans, replica.item);
    }
    return ans;
}


/**
 * this will print the LevelOrderTraversal pattern of the
 * BinarySearchTree
 * @param root
 */
private static void pattern1(Node root) {
    if (root != null) {
```

```
        Queue<Node> q = new ArrayDeque<>();
        q.add(root);

        while (!q.isEmpty()) {
            Node tmp = q.poll();
            System.out.print(tmp.item + " ");
            if (tmp.left != null)
                q.add(tmp.left);
            if (tmp.right != null)
                q.add(tmp.right);
        }
    }
}


/**
 * this will print the reverse LevelOrderTraversal pattern of the BinarySearchTree
 * @param root
 */
private static void pattern(Node root) {
    if (root != null) {
        Queue<Node> q = new ArrayDeque<>();
        Stack<Integer> stk = new Stack<>();
        q.add(root);

        while (!q.isEmpty()) {
            Node tmp = q.poll();
            stk.push(tmp.item);
            if (tmp.left != null)
                q.add(tmp.left);
            if (tmp.right != null)
                q.add(tmp.right);
        }

        while (!stk.isEmpty()) {
            System.out.print(stk.pop() + " ");
        }
    }
}


/**
 * method to print the random pattern of BinarySearchTree
 * @param root
 */
private static void pattern2(Node root) {
    if (root != null) {
        Stack<Node> stk = new Stack<>();
        stk.push(root);
```

```
        while (!stk.isEmpty()) {
            Node tmp = stk.pop();
            System.out.print(tmp.item + " ");
            if (tmp.left != null)
                stk.push(tmp.left);
            if (tmp.right != null)
                stk.push(tmp.right);
        }
    }
}


/**
 * method to print the InOrder Traversal of Binary Search Tree
 * @param root
 */
private static void print_in_order(Node root) {
    if (root != null) {
        print_in_order(root.left);
        System.out.print(root.item + " ");
        print_in_order(root.right);
    }
}


/**
 * method to print the PostOrder Traversal of Binary Search Tree
 * @param root
 */
private static void print_post_order(Node root) {
    if (root != null) {
        print_post_order(root.left);
        print_post_order(root.right);
        System.out.print(root.item + " ");
    }
}


/**
 * method to print the PreOrder Traversal of Binary Search Tree
 * @param root
 */
private static void print_pre_order(Node root) {
    if (root != null) {
        System.out.print(root.item + " ");
        print_pre_order(root.left);
        print_pre_order(root.right);
    }
```

```
}

/**
 * method to add items in the Binary Search Tree
 * @param root
 * @param item
 * @return
 */
private static Node add_items(Node root, int item) {
    if (root == null)
        return new Node(item);
    else if (root.item > item)
        root.left = add_items(root.left, item);
    else if (root.item < item)
        root.right = add_items(root.right, item);
    else
        return root;
    return root;
}

/**
 * this function will print the given DoublyLinkedList of
 * BinarySearchTree
 * @param head
 */
private static void PrintList(ListNode head) {
    if (head != null) {
        System.out.print(head.item + " ");
        PrintList(head.next);
    }
}

/**
 * this function will convert the given BinarySearchTree to a
 * DoublyLinkedList and return a Node to it
 * Time Complexity : O(N)
 * Space Complexity : O(N)
 */
private static ListNode bstToDll(Node root) {
    if (root == null)
        return null;
    else if (root.left == null && root.right == null)
        return new ListNode(root.item);
    Queue<Node> items = new LinkedList<>();
    items.add(root);
    List<Integer> keys = new LinkedList<>();
    while (!items.isEmpty()) {
        Node tmp = items.poll();
        keys.add(tmp.item);
```

```
        if (tmp.left != null)
            items.add(tmp.left);
        if (tmp.right != null)
            items.add(tmp.right);
    }
    ListNode dll = null;
    for (int key : keys) {
        if (dll == null) {
            dll = new ListNode(key);
        } else {
            ListNode tmp = dll;
            while (tmp.next != null)
                tmp = tmp.next;
            ListNode __new__ = new ListNode(key);
            tmp.next = __new__;
            __new__.prev = tmp;
        }
    }
    return dll;
}

/**
 * static Node of DoublyLinkedList
 */
static class ListNode {
    int item;
    ListNode next, prev;

    // Constructor of ListNode
    public ListNode(int item) {
        this.item = item;
        prev =  next = null;
    }
}

/**
 * static Node of Binary Search Tree
 */
static class Node {
    int item;
    Node left, right;

    // Constructor of BinarySearchTree
    public Node(int item) {
        this.item = item;
        left = right = null;
    }
}
```

```
}
```