**Problem Statement**

Arun is exploring operations on binary search trees (BST). He wants to write a program with an unsorted distinct integer array that represents the BST keys and construct a height-balanced BST from it.

After constructing, he wants to perform the following operations that can alter the structure of the tree and traverse them using a level-order traversal:

1. Insertion
2. Deletion

Your task is to assist Arun in completing the program without any errors.

**Input format :**

The first line of input consists of an integer**N,** representing the number of initial keys in the BST.

The second line consists of**N**space-separated integers, representing the initial keys.

The third line consists of an integer **X**, representing the new key to be inserted into the BST.

The fourth line consists of an integer **Y**, representing the key to be deleted from the BST.

**Output format :**

The first line of output prints "Initial BST: " followed by a space-separated list of keys in the initial BST after constructing it.

The second line prints "BST after inserting a new node [X]: " followed by a space-separated list of keys in the BST after inserting X.

The third line prints "BST after deleting node [Y]: " followed by a space-separated list of keys in the BST after deleting Y.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ N ≤ 15

1 ≤ initial keys ≤ 1000

1 ≤ X, Y ≤ 500

**Sample test cases :**

**Input 1 :**

5

25 14 56 28 12

34

12

**Output 1 :**

Initial BST: 25 14 56 12 28

BST after inserting a new node 34: 25 14 56 12 28 34

BST after deleting node 12: 25 14 56 28 34

**Input 2 :**

6

5 1 4 6 9 8

7

9

**Output 2 :**

Initial BST: 5 1 6 4 9 8

BST after inserting a new node 7: 5 1 6 4 9 8 7

BST after deleting node 9: 5 1 6 4 8 7

// You are using Java

import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;

// TreeNode class representing a node in the Binary Search Tree

class TreeNode {

    int val;

    TreeNode left, right;

    // Constructor to initialize a node with a given value

    public TreeNode(int x) {

        val = x;

        left = null;

        right = null;

    }

}

// BSTOperations class for handling BST operations

class BSTOperations {

    // Function to insert a node into the BST

    public static TreeNode insert(TreeNode root, int val) {

        // If the tree is empty, create a new node as the root

        if (root == null) {

            return new TreeNode(val);

        }

        // Insert recursively into the left subtree if the value is less than the current node's value

        if (val < root.val) {

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

        }

        // Insert recursively into the right subtree if the value is greater than the current node's value

        else if (val > root.val) {

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

        }

        // Return the updated root node

        return root;

    }

    // Function to delete a node from the BST

    public static TreeNode deleteNode(TreeNode root, int key) {

        // If the tree is empty, return null

        if (root == null) {

            return root;

        }

        // If the key is less than the root's value, delete from the left subtree

        if (key < root.val) {

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

        }

        // If the key is greater than the root's value, delete from the right subtree

        else if (key > root.val) {

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

        }

        // If the key is found

        else {

            // Node with only one child or no child

            if (root.left == null) {

                return root.right;

            } else if (root.right == null) {

                return root.left;

            }

            // Node with two children: Get the inorder successor (smallest node in the right subtree)

            root.val = minValue(root.right);

            // Delete the inorder successor

            root.right = deleteNode(root.right, root.val);

        }

        // Return the updated root node

        return root;

    }

    // Function to find the minimum value node in a BST (leftmost node)

    public static int minValue(TreeNode root) {

        int minValue = root.val;

        while (root.left != null) {

            minValue = root.left.val;

            root = root.left;

        }

        return minValue;

    }

    // Function to perform level order traversal of the BST

    public static void levelOrderTraversal(TreeNode root) {

        if (root == null) {

            return;

        }

        // Create a queue to store nodes for level order traversal

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

        queue.offer(root); // Enqueue the root node

        while (!queue.isEmpty()) {

            TreeNode current = queue.poll(); // Dequeue the front node

            System.out.print(current.val + " "); // Print the value of the dequeued node

            // Enqueue the left child if it exists

            if (current.left != null) {

                queue.offer(current.left);

            }

            // Enqueue the right child if it exists

            if (current.right != null) {

                queue.offer(current.right);

            }

        }

    }

    // Main method for testing the BST operations

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the number of initial keys in the BST

        int n = scanner.nextInt();

        TreeNode root = null;

        // Constructing the initial BST from input keys

        for (int i = 0; i < n; ++i) {

            int val = scanner.nextInt();

            root = insert(root, val); // Insert each key into the BST

        }

        // Print initial BST using level order traversal

        System.out.print("Initial BST: ");

        levelOrderTraversal(root);

        System.out.println();

        // Inserting a new node into the BST

        int x = scanner.nextInt();

        root = insert(root, x); // Insert the new node

        System.out.print("BST after inserting a new node " + x + ": ");

        levelOrderTraversal(root);

        System.out.println();

        // Deleting a node from the BST

        int y = scanner.nextInt();

        root = deleteNode(root, y); // Delete the specified node

        System.out.print("BST after deleting node " + y + ": ");

        levelOrderTraversal(root);

        System.out.println();

        scanner.close(); // Close the scanner

    }

}

**Explanation of the Code:**

1. **TreeNode Class**: Represents a node in the BST with three fields:
   * val: Value of the node.
   * left: Reference to the left child node.
   * right: Reference to the right child node. The constructor initializes these fields when a new node is created.
2. **BSTOperations Class**:
   * **insert Method**: Inserts a new node into the BST recursively. It compares the value to be inserted with the current node's value and inserts it into the left subtree if it's smaller, or the right subtree if it's larger.
   * **deleteNode Method**: Deletes a node with a specified key from the BST. It handles cases where the node to be deleted has zero, one, or two children. For nodes with two children, it finds the inorder successor (smallest node in the right subtree) to maintain the BST properties.
   * **minValue Method**: Finds and returns the minimum value node in a subtree, which is the leftmost node starting from a given node.
   * **levelOrderTraversal Method**: Performs a level order traversal (Breadth-First Search) of the BST using a queue. It starts from the root and prints nodes level by level, ensuring nodes at the same level are processed before moving on to the next level.
   * **main Method**: Handles user input and demonstrates the BST operations:
     + Constructs the initial BST from a set of keys provided by the user.
     + Prints the initial BST using level order traversal.
     + Inserts a new node into the BST based on user input and prints the updated BST.
     + Deletes a node from the BST based on user input and prints the updated BST.

**Key Concepts:**

* **BST (Binary Search Tree)**: A binary tree where for each node, the left subtree contains only nodes with values less than the node's value, and the right subtree contains only nodes with values greater than the node's value.
* **Insertion**: Adding a new node while maintaining the BST properties.
* **Deletion**: Removing a node while maintaining the BST properties, which may involve replacing the node with its inorder successor.
* **Level Order Traversal**: Visiting nodes level by level, starting from the root, using a queue to manage node processing order.

**Problem Statement**

Emily is studying binary search trees (BST). She wants to write a program that inserts values(either a character or an integer) into a BST and then finds and prints the minimum and maximum values.

Guide her with the program.

**Input format :**

The first line of input consists of an integer **N,** representing the number of values to be inserted into the BST.

The second line consists of **N** space-separated values.

**Output format :**

The first line of output prints "Minimum value: " followed by the minimum value of the given inputs.

The second line prints "Maximum value: " followed by the maximum value of the given inputs.

**Refer to the sample outputs for formatting specifications.**

**Code constraints :**

The input values will either be an upper-case character or an integer.

**Sample test cases :**

**Input 1 :**

5

Z E W T Y

**Output 1 :**

Minimum value: E

Maximum value: Z

**Input 2 :**

7

5 6 7 1 2 8 9

**Output 2 :**

Minimum value: 1

Maximum value: 9

import java.util.Scanner;

// TreeNode class representing a node in the Binary Search Tree

class TreeNode {

    Comparable val;

    TreeNode left, right;

    // Constructor to initialize a node with a given value

    public TreeNode(Comparable x) {

        val = x;

        left = null;

        right = null;

    }

}

// BSTOperations class for handling BST operations

class BSTOperations {

    // Function to insert a node into the BST

    public static TreeNode insert(TreeNode root, Comparable val) {

        // If the tree is empty, create a new node as the root

        if (root == null) {

            return new TreeNode(val);

        }

        // Insert recursively into the left subtree if the value is less than the current node's value

        if (val.compareTo(root.val) < 0) {

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

        }

        // Insert recursively into the right subtree if the value is greater than the current node's value

        else if (val.compareTo(root.val) > 0) {

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

        }

        // Return the updated root node

        return root;

    }

    // Function to find minimum value node in a BST

    public static Comparable minValue(TreeNode root) {

        if (root == null) {

            return null;

        }

        // Traverse left until the leftmost leaf node to find the minimum value

        while (root.left != null) {

            root = root.left;

        }

        return root.val;

    }

    // Function to find maximum value node in a BST

    public static Comparable maxValue(TreeNode root) {

        if (root == null) {

            return null;

        }

        // Traverse right until the rightmost leaf node to find the maximum value

        while (root.right != null) {

            root = root.right;

        }

        return root.val;

    }

}

public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read number of values to be inserted into the BST

        int n = scanner.nextInt();

        scanner.nextLine(); // Consume newline after integer input

        // Read values to be inserted into the BST

        String[] inputValues = scanner.nextLine().split("\\s+");

        TreeNode root = null;

        // Constructing the BST from input values

        for (String value : inputValues) {

            if (Character.isDigit(value.charAt(0))) {

                root = BSTOperations.insert(root, Integer.parseInt(value));

            } else {

                root = BSTOperations.insert(root, value);

            }

        }

        // Find minimum and maximum values in the BST

        Comparable minValue = BSTOperations.minValue(root);

        Comparable maxValue = BSTOperations.maxValue(root);

        // Print the minimum and maximum values

        System.out.println("Minimum value: " + minValue);

        System.out.println("Maximum value: " + maxValue);

        scanner.close();

    }

}

**Explanation:**

1. **TreeNode Class**: Represents a node in the BST with fields val, left, and right. The constructor initializes a node with a given value.
2. **BSTOperations Class**:
   * **insert Method**: Inserts a new node into the BST based on whether the value is less than or greater than the current node's value.
   * **minValue Method**: Finds the minimum value in the BST by traversing left from the root until the leftmost leaf node.
   * **maxValue Method**: Finds the maximum value in the BST by traversing right from the root until the rightmost leaf node.
3. **Main Method**:
   * Reads input values from standard input.
   * Constructs the BST by inserting each value into it.
   * Uses BSTOperations.minValue and BSTOperations.maxValue to find and print the minimum and maximum values in the BST.

**Key Concepts:**

* **Binary Search Tree (BST)**: A binary tree where each node has at most two children, with the left child having a value less than the node's value and the right child having a value greater than the node's value.
* **Insertion**: Adding nodes to the BST while maintaining the order property.
* **Traversal**: Finding the minimum and maximum values by navigating through the leftmost and rightmost nodes of the BST, respectively.

This implementation covers scenarios where the input values can be either integers or characters, ensuring flexibility in handling different types of data in the BST.

In Java and many other programming languages, \\s+ is a regular expression pattern that represents a sequence of one or more whitespace characters. Let's break down what each component means:

**Integer.parseInt** is a Java method used to parse a string representation of an integer into its int equivalent.

**Purpose of Comparable**

The Comparable interface is used to define a single method compareTo(T o) that allows objects to be compared with each other. This interface is typically implemented by classes whose instances can be ordered. By implementing Comparable, a class indicates that its instances have a natural ordering, which can be used for sorting purposes or other operations where ordering is required.

**Explanation:**

1. **Enhanced for-loop (for-each loop)**:
   * for (String value : inputValues) iterates over each element value in the inputValues array (or collection).
2. **Condition (if-else block)**:
   * if (Character.isDigit(value.charAt(0))): Checks if the first character of value is a digit.
     + Character.isDigit(value.charAt(0)) returns true if the first character of value is a digit ('0'-'9').
3. **Insertion into BST (BSTOperations.insert)**:
   * If value starts with a digit:
     + Integer.parseInt(value): Converts value from a String representation of an integer to an int.
     + root = BSTOperations.insert(root, Integer.parseInt(value)): Inserts the integer value into the BST root using the insert method from BSTOperations.
   * Else (if value does not start with a digit):
     + root = BSTOperations.insert(root, value): Inserts the String value directly into the BST root using the insert method from BSTOperations.

**Summary:**

* The loop iterates through each element (value) in the inputValues array.
* It checks whether each value starts with a digit using Character.isDigit(value.charAt(0)).
* Depending on whether value is numeric or not, it inserts either the integer value (if numeric) or the string value (if non-numeric) into the BST root.
* After each insertion, root is updated to reflect the modified BST with the newly inserted node.

**Problem Statement**

Write a program to find the kth largest element in a binary search tree (BST).

**Input format :**

The first line of input consists of integer **n,** the number of nodes in the BST.

The second line consists of the tree elements, separated by space.

The third line consists of the value of**k.**

**Output format :**

The output prints the kth largest element in the binary search tree.

For invalid inputs, print "Invalid value of k".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 15

1 ≤ BST elements ≤ 500﻿

**Sample test cases :**

**Input 1 :**

7

8 4 12 2 6 10 14

1

**Output 1 :**

14

**Input 2 :**

4

1 7 8 5

5

**Output 2 :**

Invalid value of k

**Explanation:**

1. **TreeNode Class**:
   * Represents a node in the BST with val (value), left (left child), and right (right child).
2. **BSTOperations Class**:
   * Contains methods for inserting nodes, performing reverse in-order traversal, and finding the kth largest element.
3. **insert Method**:
   * Inserts a node with a given value into the BST, maintaining the BST properties.
4. **kthLargestUtil Method**:
   * Performs a reverse in-order traversal to find the kth largest element. It uses two helper arrays:
     + count[0] to keep track of the number of nodes visited.
     + result[0] to store the kth largest element.
5. **findKthLargest Method**:
   * Validates the value of k and then calls kthLargestUtil to find the kth largest element.
   * If k is invalid (not between 1 and n), it returns Integer.MIN\_VALUE.
6. **main Method**:
   * Reads input values, constructs the BST, and prints the kth largest element or an error message for invalid k values.

This solution should handle the input constraints effectively and provide the correct output for the given problem statement.

import java.util.Scanner;

// Structure for BST node

class TreeNode {

    int val;

    TreeNode left, right;

    public TreeNode(int x) {

        val = x;

        left = null;

        right = null;

    }

}

class BSTOperations {

    // Function to insert a node in BST

    public static TreeNode insert(TreeNode root, int val) {

        if (root == null) {

            return new TreeNode(val);

        }

        if (val < root.val) {

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

        } else if (val > root.val) {

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

        }

        return root;

    }

    // Helper function to perform reverse in-order traversal and find the kth largest element

    public static void kthLargestUtil(TreeNode root, int k, int[] count, int[] result) {

        if (root == null || count[0] >= k) {

            return;

        }

        // Traverse the right subtree first (reverse in-order)

        kthLargestUtil(root.right, k, count, result);

        // Increment count of visited nodes

        count[0]++;

        // If count becomes k, store the result

        if (count[0] == k) {

            result[0] = root.val;

            return;

        }

        // Traverse the left subtree

        kthLargestUtil(root.left, k, count, result);

    }

    // Function to find the kth largest element in BST

    public static int findKthLargest(TreeNode root, int k, int n) {

        if (k <= 0 || k > n) {

            return Integer.MIN\_VALUE; // Indicates an invalid value of k

        }

        int[] count = {0}; // To keep track of count of nodes visited

        int[] result = {Integer.MIN\_VALUE}; // To store the kth largest element

        kthLargestUtil(root, k, count, result);

        return result[0];

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Reading number of nodes in BST

        int n = scanner.nextInt();

        TreeNode root = null;

        // Reading the tree elements and constructing the BST

        for (int i = 0; i < n; ++i) {

            int val = scanner.nextInt();

            root = insert(root, val);

        }

        // Reading the value of k

        int k = scanner.nextInt();

        // Finding the kth largest element

        int kthLargest = findKthLargest(root, k, n);

        if (kthLargest == Integer.MIN\_VALUE) {

            System.out.println("Invalid value of k");

        } else {

            System.out.println(kthLargest);

        }

        scanner.close();

    }

}

The function kthLargestUtil performs a reverse in-order traversal of a binary search tree (BST) to find the kth largest element. Here’s a detailed explanation of how it works:

**Reverse In-Order Traversal**

A reverse in-order traversal visits nodes in the following order for a BST:

1. Right subtree
2. Root node
3. Left subtree

By doing this, the nodes are visited in descending order. This is useful for finding the kth largest element, as we can count the nodes visited and stop once we reach the kth largest.

**Function Parameters**

* TreeNode root: The root node of the BST.
* int k: The position (kth largest) we want to find.
* int[] count: An array used to keep track of the number of nodes visited so far. This is necessary because Java doesn't support pass-by-reference for primitive types. Using an array allows us to maintain and update the count across recursive calls.
* int[] result: An array to store the result once the kth largest element is found. Similar to count, this allows us to maintain and update the result across recursive calls.

**Function Logic**

1. **Base Case**:

java

Copy code

if (root == null || count[0] >= k) {

return;

}

If the root is null (meaning we've reached a leaf node) or we've already visited k nodes (count[0] >= k), we return immediately. This stops further unnecessary traversal once we've found the kth largest element.

1. **Traverse the Right Subtree**:

java

Copy code

kthLargestUtil(root.right, k, count, result);

We start by traversing the right subtree because we want to visit the largest elements first.

1. **Visit the Current Node**:

java

Copy code

count[0]++;

if (count[0] == k) {

result[0] = root.val;

return;

}

After returning from the right subtree, we increment the count of visited nodes. If the count equals k, we store the value of the current node in result and return. This ensures that once we've found the kth largest element, we stop further traversal.

1. **Traverse the Left Subtree**:

java

Copy code

kthLargestUtil(root.left, k, count, result);

If we haven't yet found the kth largest element, we proceed to traverse the left subtree.

**Overall Flow**

The reverse in-order traversal ensures that we visit nodes in descending order. The count array keeps track of how many nodes we've visited, and once we reach the kth node, we store the value in result and stop further traversal.

Let's break down the findKthLargest function in detail. This function aims to find the kth largest element in a binary search tree (BST). It uses a helper function kthLargestUtil to perform a reverse in-order traversal and keep track of the nodes visited.

**Function Signature**

java

Copy code

public static int findKthLargest(TreeNode root, int k, int n)

* **Parameters**:
  + TreeNode root: The root node of the BST.
  + int k: The position of the largest element to find (1st largest, 2nd largest, etc.).
  + int n: The total number of nodes in the BST.
* **Return**: An integer representing the kth largest element in the BST. If k is invalid, it returns Integer.MIN\_VALUE.

**Steps of the Function**

1. **Input Validation**:

java

Copy code

if (k <= 0 || k > n) {

return Integer.MIN\_VALUE; // Indicates an invalid value of k

}

* + Checks if k is within the valid range (1 to n). If k is less than or equal to 0 or greater than n, it returns Integer.MIN\_VALUE to indicate an invalid value of k.

1. **Initialize Helper Arrays**:

java

Copy code

int[] count = {0}; // To keep track of count of nodes visited

int[] result = {Integer.MIN\_VALUE}; // To store the kth largest element

* + count: An array with a single element initialized to 0. It keeps track of how many nodes have been visited so far during the traversal.
  + result: An array with a single element initialized to Integer.MIN\_VALUE. It will store the kth largest element once it is found.

1. **Call Helper Function**:

java

Copy code

kthLargestUtil(root, k, count, result);

* + Calls the kthLargestUtil function to perform the reverse in-order traversal. The function updates count and result as it traverses the tree.

1. **Return the Result**:

java

Copy code

return result[0];

* + Returns the value stored in result[0], which is the kth largest element in the BST.

**Problem Statement**

Suppose you are developing a software system for a company that manages inventory for two different warehouses. Each warehouse's inventory is represented as a Binary Search Tree (BST) containing unique product IDs.

Your task is to create a program that checks if the inventory in both warehouses is identical. The program should compare the two BSTs and determine whether they contain the same product IDs or not.

**Input format :**

The first line of input consists of the space-separated product IDs for the first warehouse's inventory, terminated by -1.

The second line consists of the space-separated product IDs for the second warehouse's inventory, terminated by -1.

**Output format :**

If the inventories of both warehouses are identical, print "Both BSTs are identical".

Otherwise, print "BSTs are not identical".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

1 ≤ product IDs ≤ 200

**Sample test cases :**

**Input 1 :**

5 3 8 2 4 -1

5 3 8 2 4 -1

**Output 1 :**

Both BSTs are identical

**Input 2 :**

10 5 15 18 32 -1

10 5 18 15 32 -1

**Output 2 :**

BSTs are not identical

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class TreeNode {

    int val;

    TreeNode left, right;

    public TreeNode(int val) {

        this.val = val;

        this.left = null;

        this.right = null;

    }

}

public class WarehouseInventory {

    // Function to insert a node into the BST

    public static TreeNode insert(TreeNode root, int val) {

        if (root == null) {

            return new TreeNode(val);

        }

        if (val < root.val) {

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

        } else if (val > root.val) {

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

        }

        return root;

    }

    // Function to perform in-order traversal of BST and collect the nodes' values

    public static void inOrderTraversal(TreeNode root, List<Integer> result) {

        if (root != null) {

            inOrderTraversal(root.left, result);

            result.add(root.val);

            inOrderTraversal(root.right, result);

        }

    }

    // Function to compare two BSTs

    public static boolean areIdenticalBSTs(TreeNode root1, TreeNode root2) {

        List<Integer> list1 = new ArrayList<>();

        List<Integer> list2 = new ArrayList<>();

        inOrderTraversal(root1, list1);

        inOrderTraversal(root2, list2);

        return list1.equals(list2);

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        TreeNode root1 = null;

        TreeNode root2 = null;

        // Read and construct the first BST

        while (true) {

            int val = scanner.nextInt();

            if (val == -1) break;

            root1 = insert(root1, val);

        }

        // Read and construct the second BST

        while (true) {

            int val = scanner.nextInt();

            if (val == -1) break;

            root2 = insert(root2, val);

        }

        // Compare the two BSTs

        if (areIdenticalBSTs(root1, root2)) {

            System.out.println("Both BSTs are identical");

        } else {

            System.out.println("BSTs are not identical");

        }

        scanner.close();

    }

}

**Input Format and Reading Process**

The input consists of space-separated product IDs for two warehouses' inventories, each list terminated by -1. This means that each while loop will keep reading integers until it encounters -1, which signifies the end of the list for that particular warehouse.

**Detailed Explanation**

**Constructing the First BST**

java

Copy code

TreeNode root1 = null; // Initialize the root of the first BST to null

// Read and construct the first BST

while (true) {

int val = scanner.nextInt(); // Read the next integer from the input

if (val == -1) break; // If the value is -1, break out of the loop

root1 = insert(root1, val); // Insert the value into the first BST

}

* TreeNode root1 = null;: This initializes the root of the first BST to null. This is necessary because we need a starting point to begin inserting nodes into the BST.
* while (true): This creates an infinite loop that will keep running until we explicitly break out of it.
* int val = scanner.nextInt();: This reads the next integer value from the input using the Scanner object.
* if (val == -1) break;: This checks if the read value is -1. If it is, the loop breaks, indicating the end of input for the first warehouse's inventory.
* root1 = insert(root1, val);: If the value is not -1, it calls the insert method to insert the value into the first BST. The insert method ensures that the value is placed in the correct position according to BST properties. The updated root of the BST is then stored back in root1.

**Constructing the Second BST**

The process for constructing the second BST is identical to the first BST.

java

Copy code

TreeNode root2 = null; // Initialize the root of the second BST to null

// Read and construct the second BST

while (true) {

int val = scanner.nextInt(); // Read the next integer from the input

if (val == -1) break; // If the value is -1, break out of the loop

root2 = insert(root2, val); // Insert the value into the second BST

}

* TreeNode root2 = null;: This initializes the root of the second BST to null.
* while (true): This creates an infinite loop that will keep running until we explicitly break out of it.
* int val = scanner.nextInt();: This reads the next integer value from the input using the Scanner object.
* if (val == -1) break;: This checks if the read value is -1. If it is, the loop breaks, indicating the end of input for the second warehouse's inventory.
* root2 = insert(root2, val);: If the value is not -1, it calls the insert method to insert the value into the second BST. The updated root of the BST is then stored back in root2.

**Why Use These Loops?**

* **Flexibility**: These loops allow for reading an unspecified number of integers until a sentinel value (-1) is encountered. This is useful because it makes the input format flexible and clear.
* **Dynamic Construction**: The insert method dynamically constructs the BST as values are read. This ensures that each value is placed in the correct position according to the BST properties.

**Overall Flow**

1. Initialize the root of the first BST to null.
2. Enter an infinite loop to read integers for the first BST.
3. Read an integer from the input.
4. If the integer is -1, break the loop.
5. Otherwise, insert the integer into the first BST.
6. Repeat steps 1-5 for the second BST.

**Use of List<Integer> and ArrayList<Integer>**

1. **Dynamic Size Management**:
   * **List Interface**: List<Integer> provides a flexible way to store a collection of integers. It allows you to add elements dynamically without specifying the size beforehand. This is beneficial because you often do not know how many integers you will read until you encounter the termination marker (-1).
2. **Ease of Iteration and Insertion**:
   * **ArrayList Implementation**: ArrayList<Integer> is chosen because it provides constant-time complexity (O(1)) for adding elements to the end of the list and efficient iteration through its elements (O(n) complexity). This makes it suitable for scenarios where you need to read a sequence of integers and then process them sequentially.
3. **Reading Input Until Termination**:
   * By using a List<Integer> (specifically an ArrayList<Integer>), you can continuously read integers from the input until you encounter the termination marker (-1). This allows you to handle varying input sizes gracefully without having to manage array resizing or fixed-size arrays.

**Detailed Explanation**

* **List Interface**: The List<Integer> interface in Java represents an ordered collection (sequence) of elements where each element is of type Integer. It provides methods to add, remove, and access elements based on their index. ArrayList<Integer> is one of the implementations of this interface and is backed by an array.
* **ArrayList Implementation**:
  + **Dynamic Sizing**: ArrayList<Integer> automatically grows its internal array as more elements are added beyond its current capacity, ensuring that you can add elements without worrying about running out of space.
  + **Efficient Access**: It provides O(1) time complexity for adding elements to the end of the list (amortized constant time).
  + **Iterating through Elements**: You can iterate through an ArrayList efficiently using a simple for loop or using an enhanced for-loop (for-each loop), which simplifies processing each element in sequence.

**Problem Statement**

You are given a binary tree, and your task is to determine whether the tree contains any duplicate values or not.

**Input format :**

The first line of input consists of an integer representing the value of the root node.

For each node in the tree, there are two integers,

Left child data: an integer representing the value of the left child node. Use -1 to indicate no left child.

Right child data: an integer representing the value of the right child node. Use -1 to indicate no right child.

**Output format :**

Print "Yes" if the binary tree contains duplicate values.

Else, print "No".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ tree elements ≤ 100

**Sample test cases :**

**Input 1 :**

1

2

3

-1

2

-1

-1

-1

-1

**Output 1 :**

Yes

**Input 2 :**

1

20

3

-1

4

-1

-1

-1

-1

**Output 2 :**

No

import java.util.\*;

// TreeNode class representing each node in the binary tree

class TreeNode {

    int val;

    TreeNode left, right;

    TreeNode(int val) {

        this.val = val;

        this.left = null;

        this.right = null;

    }

}

public class BinaryTreeDuplicates {

    // Function to check if the binary tree contains any duplicate values

    public static boolean containsDuplicate(TreeNode root) {

        if (root == null) {

            return false;

        }

        Set<Integer> seenValues = new HashSet<>();

        return containsDuplicateHelper(root, seenValues);

    }

    // Helper function to recursively check for duplicates

    private static boolean containsDuplicateHelper(TreeNode node, Set<Integer> seenValues) {

        if (node == null) {

            return false;

        }

        // If current node's value is already seen, return true (duplicate found)

        if (seenValues.contains(node.val)) {

            return true;

        }

        // Add current node's value to the set of seen values

        seenValues.add(node.val);

        // Recursively check left and right subtrees

        return containsDuplicateHelper(node.left, seenValues) || containsDuplicateHelper(node.right, seenValues);

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the root node value

        int rootValue = scanner.nextInt();

        // Construct the binary tree from input

        TreeNode root = new TreeNode(rootValue);

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

        queue.offer(root);

        while (!queue.isEmpty()) {

            TreeNode current = queue.poll();

            // Read left child value

            int leftValue = scanner.nextInt();

            if (leftValue != -1) {

                current.left = new TreeNode(leftValue);

                queue.offer(current.left);

            }

            // Read right child value

            int rightValue = scanner.nextInt();

            if (rightValue != -1) {

                current.right = new TreeNode(rightValue);

                queue.offer(current.right);

            }

        }

        scanner.close();

        // Check if the binary tree contains any duplicate values

        if (containsDuplicate(root)) {

            System.out.println("Yes");

        } else {

            System.out.println("No");

        }

    }

}

**Problem Statement**

You are given an unordered binary tree, and your task is to determine the minimum number of swaps required to convert it into a binary search tree (BST).

A binary search tree is a binary tree in which, for each node:

1. All nodes in its left subtree have values less than the node's value.
2. All nodes in its right subtree have values greater than the node's value.

Your task is to find the minimum number of swaps of the nodes in the given binary tree to transform it into a valid BST.

**Input format :**

The first line of input consists of an integer **N,** representing the number of nodes in the binary tree.

The second line consists of the values of the **N** nodes separated by space, representing the unordered binary tree.

The input is terminated by entering -1.

**Output format :**

The output prints an integer representing the minimum number of swaps required to convert the given binary tree into a BST.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ N ≤10

**Sample test cases :**

**Input 1 :**

7

5 6 7 8 9 10 11 -1

**Output 1 :**

3

**Input 2 :**

7

50 60 70 80 90 100 110 -1

**Output 2 :**

3

**Input 3 :**

4

3 1 4 2 -1

**Output 3 :**

3

import java.util.\*;

class TreeNode {

    int val;

    TreeNode left, right;

    TreeNode(int val) {

        this.val = val;

        this.left = null;

        this.right = null;

    }

}

public class BinaryTreeToBST {

    // Function to perform inorder traversal of the binary tree

    public static void inorderTraversal(TreeNode root, List<Integer> inorderList) {

        if (root == null) return;

        inorderTraversal(root.left, inorderList);

        inorderList.add(root.val);

        inorderTraversal(root.right, inorderList);

    }

    // Function to count the minimum number of swaps to sort an array

    public static int minSwapsToSort(int[] arr) {

        int n = arr.length;

        int[] temp = Arrays.copyOf(arr, n);

        Arrays.sort(temp);

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

        for (int i = 0; i < n; i++) {

            map.put(arr[i], i);

        }

        boolean[] visited = new boolean[n];

        int swaps = 0;

        for (int i = 0; i < n; i++) {

            if (visited[i] || temp[i] == arr[i]) {

                continue;

            }

            int cycle\_size = 0;

            int j = i;

            while (!visited[j]) {

                visited[j] = true;

                j = map.get(temp[j]);

                cycle\_size++;

            }

            if (cycle\_size > 0) {

                swaps += (cycle\_size - 1);

            }

        }

        return swaps;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read number of nodes

        int n = scanner.nextInt();

        // Read the tree node values

        int[] nodes = new int[n];

        for (int i = 0; i < n; i++) {

            nodes[i] = scanner.nextInt();

        }

        scanner.close();

        // Convert the array into a binary tree (assuming level-order input)

        TreeNode root = createTree(nodes);

        // Get the inorder traversal of the binary tree

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

        inorderTraversal(root, inorderList);

        // Convert the list to an array for processing

        int[] inorderArray = inorderList.stream().mapToInt(i -> i).toArray();

        // Find the minimum number of swaps to sort the inorder array

        int minSwaps = minSwapsToSort(inorderArray);

        // Output the result

        System.out.println(minSwaps);

    }

    // Helper function to create a binary tree from level-order input

    public static TreeNode createTree(int[] nodes) {

        if (nodes.length == 0) return null;

        TreeNode root = new TreeNode(nodes[0]);

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

        queue.offer(root);

        int i = 1;

        while (i < nodes.length) {

            TreeNode current = queue.poll();

            if (nodes[i] != -1) {

                current.left = new TreeNode(nodes[i]);

                queue.offer(current.left);

            }

            i++;

            if (i < nodes.length && nodes[i] != -1) {

                current.right = new TreeNode(nodes[i]);

                queue.offer(current.right);

            }

            i++;

        }

        return root;

    }

}

**Problem Statement**

You are given a binary tree, and your task is to determine whether it is a binary search tree (BST) or not.

A binary search tree is defined as follows:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.

Write a program to check if the given binary tree is a BST or not based on these criteria.

**Input format :**

The first line of input consists of the root node's value as an integer.

For each non-null node, there will be two inputs: the value of the left child (if it exists) -1 if there is no left child, and the value of the right child (if it exists) or -1 if there is no right child.

**Output format :**

The output prints "The given binary tree is a BST" if the input binary tree satisfies the BST properties.

Otherwise, print "The given binary tree is not a BST".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

0 ≤ node values ≤ 100

**Sample test cases :**

**Input 1 :**

4 2 1 -1 -1 3 -1 -1 5 -1 -1

**Output 1 :**

The given binary tree is a BST

**Input 2 :**

3 2 1 -1 -1 4 -1 -1 5 -1 -1

**Output 2 :**

The given binary tree is not a BS

import java.util.Scanner;

class Node {

    int data;

    Node left, right;

    // Constructor to initialize the node with given data

    Node(int data) {

        this.data = data;

        this.left = this.right = null;

    }

}

class BinaryTreeCheckBST {

    // Utility function to check if the tree is a BST

    static boolean isBSTUtil(Node node, int min, int max) {

        // Base case: an empty tree is a BST

        if (node == null) {

            return true;

        }

        // If the node's data is outside the valid range, return false

        if (node.data < min || node.data > max) {

            return false;

        }

        // Recursively check the left and right subtrees with updated ranges

        return isBSTUtil(node.left, min, node.data - 1) &&

               isBSTUtil(node.right, node.data + 1, max);

    }

    // Function to check if the tree is a BST

    static boolean isBST(Node node) {

        // Initialize the range with the minimum and maximum possible values

        return isBSTUtil(node, Integer.MIN\_VALUE, Integer.MAX\_VALUE);

    }

    // Function to build the binary tree from user input

    static Node buildTree(Scanner sc) {

        int data = sc.nextInt();

        // Base case: if data is -1, it indicates a null node

        if (data == -1) {

            return null;

        }

        // Create the root node with the given data

        Node root = new Node(data);

        // Recursively build the left subtree

        root.left = buildTree(sc);

        // Recursively build the right subtree

        root.right = buildTree(sc);

        return root;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Build the binary tree from the user input

        Node root = buildTree(sc);

        // Check if the constructed tree is a BST and print the result

        if (isBST(root)) {

            System.out.println("The given binary tree is a BST");

        } else {

            System.out.println("The given binary tree is not a BST");

        }

        sc.close();

    }

}