**Problem Statement**

Seema is tasked with implementing a program that creates a binary tree using an array of integers as input. The program should then print the tree preorder traversal.

Help her with the program.

**Input format :**

The input consists of five integers separated by spaces. These integers represent the values of nodes in the binary tree.

**Output format :**

The first line of output displays "Preorder traversal:"

The second line prints space-separated integers, representing the preorder traversal values.

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

**Code constraints :**

1 ≤ value of nodes ≤ 1000

The input array size is fixed to 5 integers.

**Sample test cases :**

**Input 1 :**

1 2 3 4 5

**Output 1 :**

Preorder traversal:

1 2 4 5 3

**Input 2 :**

5 3 7 2 4

**Output 2 :**

Preorder traversal:

5 3 2 4 7

**Solution:**

import java.util.Scanner;

// TreeNode class represents a node in the binary tree

class TreeNode {

    int val;         // Value stored in the node

    TreeNode left;   // Pointer to the left child

    TreeNode right;  // Pointer to the right child

    // Constructor to initialize a new TreeNode with a value

    TreeNode(int x) {

        val = x;

        left = null;

        right = null;

    }

}

public class BinaryTreePreorder {

    static int index = 0;  // Static variable to keep track of current index in array

    // Function to insert nodes in level order using recursion

    static TreeNode insertLevelOrder(int[] arr, TreeNode root, int i) {

        // Base case for recursion: stop when index exceeds array size or root is null

        if (i < arr.length) {

            TreeNode temp = new TreeNode(arr[i]);  // Create a new node with the value from array

            root = temp;  // Set current root to the newly created node

            // Insert left child: Recursively call insertLevelOrder for left subtree

            root.left = insertLevelOrder(arr, root.left, 2 \* i + 1);

            // Insert right child: Recursively call insertLevelOrder for right subtree

            root.right = insertLevelOrder(arr, root.right, 2 \* i + 2);

        }

        return root;  // Return the root of the modified tree

    }

// Function to perform preorder traversal of the binary tree

    static void preorderTraversal(TreeNode root) {

        if (root != null) {

            System.out.print(root.val + " ");  // Print current node's value

            preorderTraversal(root.left);      // Recursively traverse left subtree

            preorderTraversal(root.right);     // Recursively traverse right subtree

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int[] arr = new int[5];  // Initialize an array to hold 5 integers

        for (int i = 0; i < 5; i++) {

            arr[i] = sc.nextInt();  // Read integers from user input into the array

        }

        TreeNode root = insertLevelOrder(arr, null, 0);  // Construct the binary tree from array

        // Output the preorder traversal of the binary tree

        System.out.println("Preorder traversal:");

        preorderTraversal(root);

    }

}

**Explanation:**

1. **TreeNode Class:**
   * **Represents a node in the binary tree with a constructor to initialize the node's value (val) and pointers to its left and right children (left and right).**
2. **insertLevelOrder Function:**
   * **Recursive function to construct the binary tree using level order insertion.**
   * **Parameters:**
     + **arr: Array of integers to insert into the binary tree.**
     + **root: Current root of the subtree being constructed.**
     + **i: Current index in the array representing the position of the node in the tree.**
   * **Base Case:**
     + **Stops recursion when i exceeds the size of the array.**
   * **Recursive Steps:**
     + **Creates a new TreeNode with the value from arr[i].**
     + **Recursively calls insertLevelOrder for the left and right subtrees (calculated using 2\*i+1 and 2\*i+2 indices).**
3. **preorderTraversal Function:**
   * **Performs a preorder traversal of the binary tree (Root-Left-Right).**
   * **Parameters:**
     + **root: Current root of the subtree being traversed.**
   * **Base Case:**
     + **Stops recursion when root is null.**
   * **Recursive Steps:**
     + **Prints the value of the current node (root.val).**
     + **Recursively calls preorderTraversal for the left and right subtrees.**
4. **main Function:**
   * **Reads 5 integers from user input into the array arr.**
   * **Constructs the binary tree using insertLevelOrder.**
   * **Prints "Preorder traversal:" followed by the result of preorderTraversal to output the preorder traversal of the constructed binary tree.**

**Problem Statement**

You are tasked with implementing a program that creates a binary tree using an array of integers as input. The program should then print the tree in-order traversal.

**Input format :**

The input consists of a single line containing five integers separated by a space.

**Output format :**

The first line prints "Inorder traversal:" .

The second line prints space separated Inorder traversal values.

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

**Code constraints :**

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

1 ≤ Value of each node ≤ 100

**Sample test cases :**

**Input 1 :**

1 2 3 4 5

**Output 1 :**

Inorder traversal:

4 2 5 1 3

**Input 2 :**

5 3 7 2 4

**Output 2 :**

Inorder traversal:

2 3 4 5 7

Solution:

import java.util.Scanner;

// TreeNode class represents a node in the binary tree

class TreeNode {

    int val;         // Value stored in the node

    TreeNode left;   // Pointer to the left child

    TreeNode right;  // Pointer to the right child

    // Constructor to initialize a new TreeNode with a value

    TreeNode(int x) {

        this.val = x;

        left = null;

        right = null;

    }

}

public class BinaryTreePreorder {

    static int index = 0;  // Static variable to keep track of current index in array

    // Function to insert nodes in level order using recursion

    static TreeNode insertLevelOrder(int[] arr, TreeNode root, int i) {

        // Base case for recursion: stop when index exceeds array size or root is null

        if (i < arr.length) {

            TreeNode temp = new TreeNode(arr[i]);  // Create a new node with the value from array

            root = temp;  // Set current root to the newly created node

            // Insert left child: Recursively call insertLevelOrder for left subtree

            root.left = insertLevelOrder(arr, root.left, 2 \* i + 1);

            // Insert right child: Recursively call insertLevelOrder for right subtree

            root.right = insertLevelOrder(arr, root.right, 2 \* i + 2);

        }

        return root;  // Return the root of the modified tree

    }

    // Function to perform preorder traversal of the binary tree

    static void InorderTraversal(TreeNode root) {

        if (root != null) {

            InorderTraversal(root.left);      // Recursively traverse left subtree

            System.out.print(root.val + " ");  // Print current node's value

            InorderTraversal(root.right);     // Recursively traverse right subtree

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int[] arr = new int[5];  // Initialize an array to hold 5 integers

        for (int i = 0; i < 5; i++) {

            arr[i] = sc.nextInt();  // Read integers from user input into the array

        }

        TreeNode root = insertLevelOrder(arr, null, 0);  // Construct the binary tree from array

        // Output the preorder traversal of the binary tree

        System.out.println("Inorder traversal:");

        InorderTraversal(root);

    }

}

**Problem Statement**

Write a program to construct a binary search tree (BST) using a given set of characters, perform a postorder traversal of the constructed BST, and print the result.

**Example:**

**Input:**

5

Z E W T Y

**Output:**

Postorder traversal: T Y W E Z

**Input format :**

The first line of input consists of the number of characters (N).

The second line of input consists of N characters separated by a space.

**Output format :**

The output displays the post-order traversal of the given inputs as space-separated.

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

**Code constraints :**

The given test cases will fall under the following constraints:

1 ≤ size ≤ 100

Each character in the input set is a valid ASCII character.

**Sample test cases :**

**Input 1 :**

5

Z E W T Y

**Output 1 :**

Postorder traversal: T Y W E Z

**Input 2 :**

6

F B G A D C

**Output 2 :**

Postorder traversal: A C D B

**Solution:**

import java.util.Scanner;

// TreeNode class represents a node in the binary search tree (BST)

class TreeNode {

    char val;         // Value stored in the node (character in this case)

    TreeNode left;    // Pointer to the left child

    TreeNode right;   // Pointer to the right child

    // Constructor to initialize a new TreeNode with a value

    TreeNode(char x) {

        this.val = x;

        left = null;

        right = null;

    }

}

class BinarySearchTreePostOrder {

    // Function to insert a node in the BST

    static TreeNode insert(TreeNode root, char val) {

        if (root == null) {

            return new TreeNode(val);  // Create a new node if root is null

        }

        if (val < root.val) {

            root.left = insert(root.left, val);   // Insert into the left subtree recursively

        } else if (val > root.val) {

            root.right = insert(root.right, val); // Insert into the right subtree recursively

        }

        return root;  // Return the modified root of the tree

    }

    // Function to perform postorder traversal of the BST

    static void postOrderTraversal(TreeNode root) {

        if (root != null) {

            postOrderTraversal(root.left);    // Recursively traverse left subtree

            postOrderTraversal(root.right);   // Recursively traverse right subtree

            System.out.print(root.val + " "); // Print current node's value

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Read the number of characters (nodes) to insert into the BST

        int n = sc.nextInt();

        sc.nextLine();  // Consume the newline character after reading integer input

        // Read the characters (node values) as a single line of input and split by space

        String[] chars = sc.nextLine().split(" ");

        TreeNode root = null;  // Initialize the root of the BST as null

        // Construct the BST by inserting each character into the BST

        for (String ch : chars) {

            root = insert(root, ch.charAt(0));  // Convert string character to char and insert into BST

        }

        // Output format

        System.out.print("Postorder traversal:");

        postOrderTraversal(root);  // Perform postorder traversal and print the result

    }

}

**Problem Statement**

Amir needs a program to construct a binary tree from user-provided input and analyze it. The program takes the number of nodes and their values as input, constructs the binary tree, performs a pre-order traversal, and computes the sum of all node values encountered during traversal.

**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 **n** space-separated integers, representing the values of the nodes in the binary tree.

**Output format :**

The first line of output prints "Preorder Traversal: " followed by the pre-order traversal of the constructed binary tree.

The second line prints "Sum: " followed by the sum of all node values in the binary tree.

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

**Code constraints :**

1 ≤ n ≤ 15

1 ≤ node values ≤ 150

**Sample test cases :**

**Input 1 :**

4

15 62 34 27

**Output 1 :**

Preorder Traversal: 15 62 27 34

Sum: 138

**Input 2 :**

6

9 5 3 7 1 2

**Output 2 :**

Preorder Traversal: 9 5 7 1 3 2

Sum: 27

import java.util.Scanner;

// TreeNode class represents a node in the binary tree

class TreeNode {

    int val;           // Value stored in the node

    TreeNode left;     // Pointer to the left child

    TreeNode right;    // Pointer to the right child

    // Constructor to initialize a new TreeNode with a value

    TreeNode(int x) {

        val = x;

        left = null;

        right = null;

    }

}

class BinaryTreePreOrderSum {

    // Function to insert nodes in level order

    static TreeNode insertLevelOrder(int[] arr, TreeNode root, int i) {

        // Base case for recursion

        if (i < arr.length) {

            TreeNode temp = new TreeNode(arr[i]);  // Create a new node with the value from array

            root = temp;  // Set current root to the newly created node

            // Insert left child: Recursively call insertLevelOrder for left subtree

            root.left = insertLevelOrder(arr, root.left, 2 \* i + 1);

            // Insert right child: Recursively call insertLevelOrder for right subtree

            root.right = insertLevelOrder(arr, root.right, 2 \* i + 2);

        }

        return root;  // Return the root of the modified tree

    }

    // Function to perform pre-order traversal of the binary tree and compute sum

    static void preOrderTraversal(TreeNode root, StringBuilder traversal, int[] sum) {

        if (root != null) {

            traversal.append(root.val).append(" ");  // Append current node's value to traversal StringBuilder

            sum[0] += root.val;  // Add current node's value to sum

            // Recursively traverse left subtree

            preOrderTraversal(root.left, traversal, sum);

            // Recursively traverse right subtree

            preOrderTraversal(root.right, traversal, sum);

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Read the number of nodes

        int n = sc.nextInt();

        // Read the node values into an array

        int[] arr = new int[n];

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        // Create the binary tree from the array

        TreeNode root = insertLevelOrder(arr, null, 0);

        // Initialize StringBuilder to store traversal sequence

        StringBuilder traversal = new StringBuilder();

        // Initialize array to store sum of node values (using array to pass by reference)

        int[] sum = {0};

        // Perform pre-order traversal to populate traversal StringBuilder and compute sum

        preOrderTraversal(root, traversal, sum);

        // Output the results

        System.out.println("Preorder Traversal: " + traversal.toString().trim());  // Trim to remove extra space at end

        System.out.println("Sum: " + sum[0]);  // Output the computed sum

    }

}

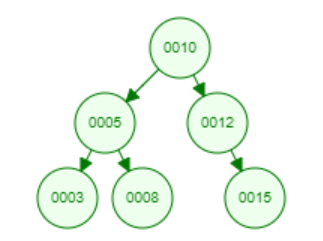
**Explanation with Inline Comments:**

1. **TreeNode Class:**
   * **Represents a node in the binary tree with a value (val) and pointers to left and right children (left, right).**
2. **insertLevelOrder Function:**
   * **Recursively constructs the binary tree using level order insertion.**
   * **Parameters:**
     + **arr: Array of integers containing node values.**
     + **root: Current root of the subtree being constructed.**
     + **i: Current index in the array representing the position of the node in the tree.**
   * **Base Case:**
     + **Stops recursion when i exceeds the size of the array.**
   * **Recursive Steps:**
     + **Creates a new TreeNode with the value from arr[i].**
     + **Recursively calls insertLevelOrder for the left and right subtrees (calculated using 2\*i+1 and 2\*i+2 indices).**
3. **preOrderTraversal Function:**
   * **Performs a preorder traversal (Root-Left-Right) of the binary tree.**
   * **Parameters:**
     + **root: Current root of the subtree being traversed.**
     + **traversal: StringBuilder to store the preorder traversal sequence.**
     + **sum: Array to store the sum of node values encountered during traversal (passed by reference).**
   * **Base Case:**
     + **Stops recursion when root is null.**
   * **Recursive Steps:**
     + **Appends the value of the current node (root.val) to the traversal StringBuilder.**
     + **Adds the current node's value to sum[0].**
     + **Recursively calls preOrderTraversal for the left and right subtrees.**
4. **main Function:**
   * **Reads input using Scanner:**
     + **Reads an integer n which indicates the number of nodes to be inserted into the binary tree.**
     + **Reads the next n integers into the array arr representing the node values.**
   * **Constructs the binary tree by calling insertLevelOrder with null root and starting index 0.**
   * **Initializes a StringBuilder traversal to store preorder traversal sequence.**
   * **Initializes an int array sum with one element (sum[0]) to store the sum of node values.**
   * **Calls preOrderTraversal to perform preorder traversal, updating traversal and sum.**
   * **Outputs the preorder traversal sequence and the computed sum.**

**Problem Statement**

Alex is working on a project involving binary search trees (BSTs) and needs to find the lowest common ancestor (LCA) of two given nodes in a BST.

Help him write a program to find the LCA of two nodes in a BST constructed from user-provided data.



**Example**

**Input:**

6

10 5 12 3 8 15

3 8

**Output:**

5

**Explanation:**

The BST for the nodes is

The nodes for which we need to find the LCA are 3 and 8. The lowest common ancestor of these two nodes is 5, as it is the lowest node in the tree that has both 3 and 8 as descendants.

**Input format :**

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

The second line consists of **n** space-separated integers, representing the values of the nodes to be inserted into the BST.

The third line consists of two integers **n1** and **n2**, representing the nodes for which you need to find the lowest common ancestor.

**Output format :**

The output prints an integer, representing the value of the lowest common ancestor of the two given nodes.

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

**Code constraints :**

1 ≤ n ≤ 15

1 ≤ tree elements ≤ 100

**Sample test cases :**

**Input 1 :**

4

71 90 44 59

44 90

**Output 1 :**

71

**Input 2 :**

6

10 5 12 3 8 15

3 8

**Output 2 :**

5

Sol

import java.util.Scanner;

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

class TreeNode {

    int val;           // Value stored in the node

    TreeNode left;     // Pointer to the left child

    TreeNode right;    // Pointer to the right child

    // Constructor to initialize a new TreeNode with a value

    TreeNode(int x) {

        val = x;

        left = null;

        right = null;

    }

}

class BSTLowestCommonAncestor {

    // Function to insert a node in BST

    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 find the lowest common ancestor (LCA) of two nodes in BST

    static TreeNode lowestCommonAncestor(TreeNode root, int p, int q) {

        if (root == null) {

            return null;

        }

        if (p < root.val && q < root.val) {

            return lowestCommonAncestor(root.left, p, q);

        } else if (p > root.val && q > root.val) {

            return lowestCommonAncestor(root.right, p, q);

        } else {

            return root;

        }

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Read the number of nodes

        int n = sc.nextInt();

        // Read the node values into an array

        int[] arr = new int[n];

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        // Read the two nodes for which we need to find the LCA

        int n1 = sc.nextInt();

        int n2 = sc.nextInt();

        // Create the BST from the array of node values

        TreeNode root = null;

        for (int val : arr) {

            root = insert(root, val);

        }

        // Find the LCA of the two nodes

        TreeNode lca = lowestCommonAncestor(root, n1, n2);

        // Output the value of the LCA

        System.out.println(lca.val);

    }

}

**Explanation with Inline Comments:**

1. **TreeNode Class**:
   * Represents a node in the binary search tree (BST) with a value (val) and pointers to left and right children (left, right).
2. **insert Function**:
   * Inserts a node into the BST recursively based on its value.
   * Parameters:
     + root: Current root of the subtree where insertion is to be performed.
     + val: Value to be inserted into the BST.
   * Base Case:
     + If root is null, a new node with the value val is created and returned.
   * Recursive Steps:
     + If val is less than root.val, recursively call insert on the left subtree.
     + If val is greater than root.val, recursively call insert on the right subtree.
   * Returns the updated root of the subtree.
3. **lowestCommonAncestor Function**:
   * Finds the lowest common ancestor (LCA) of two nodes (p and q) in the BST.
   * Parameters:
     + root: Current root of the subtree being examined.
     + p, q: Values of the two nodes whose LCA is to be found.
   * Base Case:
     + If root is null, returns null.
   * Recursive Steps:
     + If both p and q are less than root.val, recursively call lowestCommonAncestor on the left subtree.
     + If both p and q are greater than root.val, recursively call lowestCommonAncestor on the right subtree.
     + If the above conditions are not met, it means p and q are on different sides of root, hence root is the LCA.
4. **main Function**:
   * Reads input using Scanner:
     + Reads an integer n which indicates the number of nodes to be inserted into the BST.
     + Reads the next n integers into the array arr representing the node values.
     + Reads two integers n1 and n2 which represent the values of the two nodes for which the LCA needs to be found.
   * Constructs the BST by iteratively inserting nodes from arr using insert function.
   * Finds the LCA of nodes n1 and n2 using lowestCommonAncestor function.
   * Outputs the value of the LCA.