Binary Search Tree P

Basic to Intermediate Questions:
Find the Kth Smallest/Largest Element in a BST: Use inorder traversal to solve this.
Floor and Ceil of a Given Number in BST: Implement methods to find the floor and ceil values for a given number.
Find the Lowest Common Ancestor (LCA) in a BST: Given two nodes, find their LCA.
Inorder Predecessor and Successor for a Given Node in BST: Find both for any given node.
Convert a BST to a Doubly Linked List: Perform in-place conversion.
Count Pairs from Two BSTs Whose Sum is Equal to a Given Value: Use a two-pointer technique.
Check if a BST Contains a Dead End: A dead end is where no further node can be inserted.
Merge Two Balanced BSTs into a Balanced BST: Without converting to arrays, merge the BSTs.
Find the Distance Between Two Nodes in a BST: Find the shortest path between two nodes.
BST Iterator: Design a class that iterates through the BST in increasing order using O(h) memory.
explain problem with example with input and output in simple way and give optimized and simple java code with its time complexity

Memory updated

Let's go through each problem one by one with examples, explanations, and optimized Java code.

1. Find the Kth Smallest/Largest Element in a BST

Problem Explanation:

In a Binary Search Tree (BST), the inorder traversal of the tree results in nodes being visited in sorted (ascending) order. To find the Kth smallest or largest element, you can perform an inorder traversal and keep count of the nodes visited.

Example:

Input: BST with nodes `[5, 3, 7, 2, 4, 6, 8]`, `k = 3` (Kth smallest) Output: `4` (the 3rd smallest element in the BST)

Java Code:

```java
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) {
        this.val = val;
        left = right = null;
    }
}
```

```
class BSTOperations {
    int count = 0, result = -1;

    // Kth smallest using inorder traversal
    public int kthSmallest(TreeNode root, int k) {
        inorder(root, k);
        return result;
    }

    private void inorder(TreeNode node, int k) {
        if (node == null) return;

        inorder(node.left, k);
        count++;
        if (count == k) {
            result = node.val;
            return;
        }
        inorder(node.right, k);
    }
}
```

Time Complexity:

- **Time Complexity:** O(N) where N is the number of nodes.
- **Space Complexity:** O(H) where H is the height of the tree (due to recursion).

---

2. Floor and Ceil of a Given Number in BST

Problem Explanation:

- **Floor:** Largest value in the BST smaller than or equal to the given number.
- **Ceil:** Smallest value in the BST greater than or equal to the given number.

Example:

Input: BST with nodes ``` `[5, 3, 7, 2, 4, 6, 8]` ```, ``` `key = 6` ``` Output: ``` `Floor = 6` ```, ``` `Ceil = 6` ```

Java Code:

java

```
class FloorCeil {
    public int floor(TreeNode root, int key) {
        int floor = -1;
        while (root != null) {
            if (root.val == key) {
                return root.val;
            } else if (key > root.val) {
                floor = root.val;
                root = root.right;
            } else {
                root = root.left;
            }
        }
        return floor;
    }

    public int ceil(TreeNode root, int key) {
        int ceil = -1;
        while (root != null) {
```

```
            if (root.val == key) {
                return root.val;
            } else if (key < root.val) {
                ceil = root.val;
                root = root.left;
            } else {
                root = root.right;
            }
        }
        return ceil;
    }
}
```

Time Complexity:

- **Time Complexity:** O(H), where H is the height of the tree.
- **Space Complexity:** O(1), iterative approach.

---

3. Find the Lowest Common Ancestor (LCA) in a BST

Problem Explanation:

In a BST, the LCA of two nodes is the deepest node that is the ancestor of both nodes.

Example:

Input: BST with nodes `` `[5, 3, 7, 2, 4, 6, 8]` ``, `` `node1 = 4` ``, `` `node2 = 8` `` Output: `` `LCA = 5` ``

Java Code:

```java
class LCAInBST {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return null;

        if (root.val > p.val && root.val > q.val) {
            return lowestCommonAncestor(root.left, p, q);
        } else if (root.val < p.val && root.val < q.val) {
            return lowestCommonAncestor(root.right, p, q);
        } else {
            return root;
        }
    }
}
```

Time Complexity:

- **Time Complexity:** O(H), where H is the height of the tree.
- **Space Complexity:** O(H), due to recursion stack.

---

4. Inorder Predecessor and Successor for a Given Node in BST

Problem Explanation:

The predecessor is the largest value smaller than the given node, and the successor is the smallest value larger than the given node.

Example:

Input: BST with nodes `[5, 3, 7, 2, 4, 6, 8]`, `key = 5` Output: `Predecessor = 4`, `Successor = 6`

Java Code:

```java
class PreSucInBST {
    TreeNode predecessor = null, successor = null;

    public void findPreSuc(TreeNode root, TreeNode key) {
        if (root == null) return;

        if (root.val == key.val) {
            if (root.left != null) {
                TreeNode temp = root.left;
                while (temp.right != null)
                    temp = temp.right;
                predecessor = temp;
            }
            if (root.right != null) {
                TreeNode temp = root.right;
                while (temp.left != null)
                    temp = temp.left;
                successor = temp;
            }
        } else if (root.val > key.val) {
            successor = root;
            findPreSuc(root.left, key);
        } else {
            predecessor = root;
            findPreSuc(root.right, key);
        }
    }
}
```

Time Complexity:

- **Time Complexity:** O(H), where H is the height of the tree.
- **Space Complexity:** O(1), iterative approach.

---

5. Convert a BST to a Doubly Linked List

Problem Explanation:

Convert the BST into a doubly linked list, such that the nodes are in the same order as an inorder traversal.

Example:

Input: BST with nodes `[5, 3, 7, 2, 4, 6, 8]` Output: Doubly Linked List `2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8`

Java Code:

```java
class BSTToDLL {
    TreeNode head = null, prev = null;

    public TreeNode bstToDoublyList(TreeNode root) {
        if (root == null) return null;

        inorder(root);
        return head;
    }

    private void inorder(TreeNode node) {
        if (node == null) return;

        inorder(node.left);

        if (prev == null) {
            head = node;
        } else {
            prev.right = node;
            node.left = prev;
        }
        prev = node;

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

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), due to recursion stack.

The rest of the problems can follow similar explanations and implementations. Would you like to continue with the remaining ones?

give remaining

Let's continue with the remaining problems in BST.

---

6. Count Pairs from Two BSTs Whose Sum is Equal to a Given Value

Problem Explanation:

Given two BSTs and a target sum, you need to find all pairs (one from each BST) whose sum equals the target.

Example:

Input:

- BST1 with nodes `` `[5, 3, 7, 2, 4]` ``
- BST2 with nodes `` `[8, 6, 10]` ``
- Target = `` `10` `` Output: `` `[(4, 6)]` ``

Approach:

1. Perform inorder traversal of both trees to get two sorted lists.
2. Use a two-pointer technique to find pairs with the given sum.

Java Code:

```java
class PairSumInBSTs {
    // Store inorder traversal in a list
    private void inorder(TreeNode root, List<Integer> list) {
        if (root == null) return;
        inorder(root.left, list);
        list.add(root.val);
        inorder(root.right, list);
    }

    public List<int[]> findPairs(TreeNode root1, TreeNode root2, int target) {
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();

        // Get inorder traversal of both BSTs
        inorder(root1, list1);
        inorder(root2, list2);

        List<int[]> result = new ArrayList<>();
        int i = 0, j = list2.size() - 1;

        // Two-pointer technique
        while (i < list1.size() && j >= 0) {
            int sum = list1.get(i) + list2.get(j);
            if (sum == target) {
                result.add(new int[]{list1.get(i), list2.get(j)});
                i++;
                j--;
            } else if (sum < target) {
                i++;
            } else {
                j--;
            }
        }
        return result;
    }
}
```

Time Complexity:

- **Time Complexity:** O(N + M), where N and M are the number of nodes in the two BSTs.
- **Space Complexity:** O(N + M), for storing the inorder traversals.

---

7. Check if a BST Contains a Dead End

Problem Explanation:

A dead end in a BST is a node that cannot have any children (i.e., you cannot insert any more nodes). This happens when a node has no possible values for new nodes to fit between its parent and itself.

Example:

Input: BST with nodes `[8, 5, 9, 2, 7, 1, 3]` Output: `True` (because the node `3` is a dead end)

Java Code:

```java
class DeadEndInBST {
    public boolean isDeadEnd(TreeNode root) {
        return checkDeadEnd(root, 1, Integer.MAX_VALUE);
    }

    private boolean checkDeadEnd(TreeNode node, int min, int max) {
        if (node == null) return false;

        // Dead end condition
        if (min == max) return true;

        return checkDeadEnd(node.left, min, node.val - 1) ||
               checkDeadEnd(node.right, node.val + 1, max);
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), due to recursion stack.

---

8. Merge Two Balanced BSTs into a Balanced BST

Problem Explanation:

Given two balanced BSTs, merge them into a single balanced BST without converting them to arrays.

Approach:

1. Perform inorder traversal of both trees to get sorted lists.
2. Merge the two sorted lists.
3. Construct a balanced BST from the merged list.

Example:

Input:

- BST1 with nodes `[1, 3, 5]`
- BST2 with nodes `[2, 4, 6]` Output: Merged balanced BST with nodes `[1, 2, 3, 4, 5, 6]`

Java Code:

java

```
class MergeBSTs {
    // Store inorder traversal in a list
    private void inorder(TreeNode root, List<Integer> list) {
        if (root == null) return;
        inorder(root.left, list);
        list.add(root.val);
        inorder(root.right, list);
    }

    // Merge two sorted lists
    private List<Integer> merge(List<Integer> list1, List<Integer> list2) {
        List<Integer> mergedList = new ArrayList<>();
        int i = 0, j = 0;

        while (i < list1.size() && j < list2.size()) {
            if (list1.get(i) < list2.get(j)) {
                mergedList.add(list1.get(i++));
            } else {
                mergedList.add(list2.get(j++));
            }
        }
        while (i < list1.size()) mergedList.add(list1.get(i++));
        while (j < list2.size()) mergedList.add(list2.get(j++));

        return mergedList;
    }

    // Build a balanced BST from sorted list
    private TreeNode buildBST(List<Integer> list, int start, int end) {
        if (start > end) return null;

        int mid = (start + end) / 2;
        TreeNode node = new TreeNode(list.get(mid));
        node.left = buildBST(list, start, mid - 1);
        node.right = buildBST(list, mid + 1, end);

        return node;
    }

    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();

        // Get inorder traversal of both BSTs
        inorder(root1, list1);
        inorder(root2, list2);

        // Merge the two lists
        List<Integer> mergedList = merge(list1, list2);

        // Build balanced BST from merged list
        return buildBST(mergedList, 0, mergedList.size() - 1);
    }
}
```

Time Complexity:

- **Time Complexity:** O(N + M), where N and M are the number of nodes in the two BSTs.
- **Space Complexity:** O(N + M), for storing the inorder traversals.

---

9. Find the Distance Between Two Nodes in a BST

Problem Explanation:

Given two nodes in a BST, find the shortest distance between them.

Approach:

1. Find the Lowest Common Ancestor (LCA).
2. Find the distance from the LCA to each node and sum the distances.

Example:

Input: BST with nodes `[5, 3, 7, 2, 4, 6, 8]`, `node1 = 2`, `node2 = 6` Output: `Distance = 4`

Java Code:

```java
class DistanceBetweenNodes {
    private TreeNode lca(TreeNode root, int n1, int n2) {
        if (root == null) return null;

        if (root.val > n1 && root.val > n2) return lca(root.left, n1, n2);
        if (root.val < n1 && root.val < n2) return lca(root.right, n1, n2);

        return root;
    }

    private int distanceFromLCA(TreeNode root, int val) {
        int distance = 0;
        while (root != null) {
            if (root.val == val) break;
            else if (root.val > val) root = root.left;
            else root = root.right;
            distance++;
        }
        return distance;
    }

    public int findDistance(TreeNode root, int n1, int n2) {
        TreeNode lcaNode = lca(root, n1, n2);
        int d1 = distanceFromLCA(lcaNode, n1);
        int d2 = distanceFromLCA(lcaNode, n2);
        return d1 + d2;
    }
}
```

Time Complexity:

- **Time Complexity:** O(H), where H is the height of the tree.
- **Space Complexity:** O(H), due to recursion stack.

---

10. BST Iterator: O(h) Memory

Problem Explanation:

Design a class that simulates an iterator for a BST, which returns the next smallest number using O(h) memory, where h is the height of the tree.

Example:

Input: BST with nodes `[7, 3, 15, 9, 20]` Iterator sequence: `3 -> 7 -> 9 -> 15 -> 20`

Java Code:

```java
class BSTIterator {
    private Stack<TreeNode> stack;

    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        pushAll(root);
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode node = stack.pop();
        pushAll(node.right);
        return node.val;
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    private void pushAll(TreeNode node) {
        while (node != null) {
            stack.push(node);
            node = node.left;
        }
    }
}
```

Time Complexity:

- **Time Complexity:** O(1) for `next()` and O(h) for the initial push.
- **Space Complexity:** O(h), where h is the height of the tree (for stack storage).

These are all the remaining BST problems. Let me know if you need any further explanations or additional problems!

Advanced Questions:
Recover a BST Where Two Nodes Are Swapped by Mistake: Restore the BST by fixing the swapped nodes.
Largest BST Subtree in a Binary Tree: Return the largest BST subtree rooted at any node in a binary tree.
Find Pairs in BST Whose Sum is Equal to Target: Without extra space, find if two nodes sum up to a target.
Convert a BST into a Min Heap: Do this without using any extra space.
Serialize and Deserialize a BST: Encode and decode a BST into a string and reconstruct it back.
Range Sum of BST: Given two values, return the sum of all nodes within the range.
Find the Kth Smallest Element in a Binary Tree (Not BST): Convert the binary tree into a BST and then solve.
Count BST Subtrees That Lie in a Given Range: Count all the BST subtrees that have values within

a specified range.
Construct BST from Preorder Traversal: Given preorder, reconstruct the BST efficiently.
All Possible Unique BSTs for Keys from 1 to N: Generate all structurally unique BSTs for given N nodes.

Memory updated
Let's go through the advanced BST problems one by one with explanations, examples, and Java code.

---

1. Recover a BST Where Two Nodes Are Swapped by Mistake

Problem Explanation:

Two nodes of a BST are swapped by mistake. Your task is to restore the tree without changing its structure.

Example:

Input: BST where `3` and `7` are swapped.

markdown

```
      6
     / \
    3   8
   / \
  1   7
```

Output: Corrected BST:

markdown

```
      6
     / \
    7   8
   / \
  1   3
```

Approach:

- Perform an inorder traversal to find the two swapped nodes.
- Swap them back to restore the BST.

Java Code:

java

```java
class RecoverBST {
    TreeNode first, second, prev;
    public void recoverTree(TreeNode root) {
        inorder(root);
        // Swap the values of the two nodes
        int temp = first.val;
        first.val = second.val;
        second.val = temp;
    }

    private void inorder(TreeNode node) {
        if (node == null) return;

        inorder(node.left);

        // Detect swapped nodes
        if (prev != null && node.val < prev.val) {
            if (first == null) first = prev;
            second = node;
        }

        prev = node;

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

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), where H is the height of the tree.

---

2. Largest BST Subtree in a Binary Tree

Problem Explanation:

Find the largest subtree in a binary tree that is a valid BST.

Example:

Input:

```markdown
      10
     /  \
    5   15
   / \    \
  1   8    7
```

Output: Subtree rooted at `5` is the largest BST.

Java Code:

```java
class LargestBSTSubtree {
    class SubtreeInfo {
        int size;
        int min;
        int max;
        boolean isBST;

        SubtreeInfo(int size, int min, int max, boolean isBST) {
            this.size = size;
            this.min = min;
            this.max = max;
            this.isBST = isBST;
        }
    }

    private int maxSize = 0;

    public int largestBSTSubtree(TreeNode root) {
        largestBST(root);
        return maxSize;
    }

    private SubtreeInfo largestBST(TreeNode node) {
        if (node == null) return new SubtreeInfo(0, Integer.MAX_VALUE, Integer.MIN_VALUE,
true);

        SubtreeInfo left = largestBST(node.left);
        SubtreeInfo right = largestBST(node.right);

        if (left.isBST && right.isBST && node.val > left.max && node.val < right.min) {
            int size = left.size + right.size + 1;
            maxSize = Math.max(maxSize, size);
            return new SubtreeInfo(size, Math.min(node.val, left.min), Math.max(node.val,
right.max), true);
        } else {
            return new SubtreeInfo(0, 0, 0, false);
        }
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), where H is the height of the tree.

---

3. Find Pairs in BST Whose Sum is Equal to Target Without Extra Space

Problem Explanation:

Find if there exist two nodes in a BST such that their sum is equal to a given target, without using extra space.

Example:

Input: BST with nodes `[5, 3, 7, 2, 4, 6, 8]`, Target = `9` Output: True (because `5 + 4 = 9`)

Approach:

- Use an inorder traversal (for increasing order) and a reverse inorder traversal (for decreasing order) simultaneously, like a two-pointer technique.

Java Code:

```java
class FindPairsInBST {
    public boolean findTarget(TreeNode root, int target) {
        Stack<TreeNode> leftStack = new Stack<>();
        Stack<TreeNode> rightStack = new Stack<>();

        TreeNode left = root, right = root;

        // Initialize the two pointers
        while (left != null) {
            leftStack.push(left);
            left = left.left;
        }

        while (right != null) {
            rightStack.push(right);
            right = right.right;
        }

        // Two-pointer technique
        while (!leftStack.isEmpty() && !rightStack.isEmpty()) {
            TreeNode leftNode = leftStack.peek();
            TreeNode rightNode = rightStack.peek();

            if (leftNode == rightNode) break;

            int sum = leftNode.val + rightNode.val;
            if (sum == target) return true;

            if (sum < target) {
                leftNode = leftStack.pop();
                leftNode = leftNode.right;
                while (leftNode != null) {
                    leftStack.push(leftNode);
                    leftNode = leftNode.left;
                }
            } else {
                rightNode = rightStack.pop();
                rightNode = rightNode.left;
                while (rightNode != null) {
                    rightStack.push(rightNode);
                    rightNode = rightNode.right;
                }
            }
        }
        return false;
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), where H is the height of the tree (for the stack).

---

4. Convert a BST into a Min Heap

Problem Explanation:

Convert a BST into a Min Heap, where each node’s value is smaller than its children. Do this in-place without using any extra space.

Example:

Input: BST with nodes `` `[4, 2, 6, 1, 3, 5, 7]` `` Output: Min Heap:

```markdown
    1
   / \
  3   2
 / \ / \
7  5 6  4
```

Approach:

- First, perform inorder traversal to store values in sorted order.
- Then, modify the tree to follow Min Heap properties using the stored values.

Java Code:

```java
class BSTToMinHeap {
    List<Integer> values = new ArrayList<>();
    int index = 0;

    public void convertToMinHeap(TreeNode root) {
        inorder(root); // Store nodes in inorder
        buildMinHeap(root); // Convert BST to Min Heap
    }

    // Inorder traversal to collect values in sorted order
    private void inorder(TreeNode root) {
        if (root == null) return;
        inorder(root.left);
        values.add(root.val);
        inorder(root.right);
    }

    // Modify the tree using Min Heap properties
    private void buildMinHeap(TreeNode root) {
        if (root == null) return;
        root.val = values.get(index++);
        buildMinHeap(root.left);
        buildMinHeap(root.right);
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(N), due to the list storing values.

I'll continue with the remaining advanced BST questions in the next response.

give remaining

Here are the remaining advanced BST problems with explanations, examples, and Java code.

---

5. Serialize and Deserialize a BST

Problem Explanation:

Serialize a BST to a string and deserialize it back to a BST. The structure of the BST should remain the same.

Example:

Input:

```markdown
       5
      / \
     3   7
    / \ / \
   2  4 6  8
```

Serialized output: `"5,3,2,#,#,4,#,#,7,6,#,#,8,#,#"` Deserialized output: Same BST.

Approach:

- Serialize using preorder traversal and handle null nodes using a special character (e.g., `#`).
- Deserialize using a recursive approach that mimics the preorder construction.

Java Code:

```java
class SerializeDeserializeBST {
    // Serialize the BST
    public String serialize(TreeNode root) {
        if (root == null) return "#";
        return root.val + "," + serialize(root.left) + "," + serialize(root.right);
    }

    // Deserialize the BST
    public TreeNode deserialize(String data) {
        Queue<String> nodes = new LinkedList<>(Arrays.asList(data.split(",")));
        return buildTree(nodes);
    }

    private TreeNode buildTree(Queue<String> nodes) {
        String val = nodes.poll();
        if (val.equals("#")) return null;
        TreeNode root = new TreeNode(Integer.parseInt(val));
        root.left = buildTree(nodes);
        root.right = buildTree(nodes);
        return root;
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(N), for storing the serialized string and deserialized nodes.

---

6. Range Sum of BST

Problem Explanation:

Given a BST and a range `[L, R]`, return the sum of all nodes with values within this range.

Example:

Input:

```markdown
      10
     /  \
    5   15
   / \    \
  3   7   18
```

Range: `[7, 15]` Output: `32` (sum of 7, 10, and 15)

Java Code:

```java
class RangeSumBST {
    public int rangeSumBST(TreeNode root, int L, int R) {
        if (root == null) return 0;
        int sum = 0;
        if (root.val >= L && root.val <= R) sum += root.val;
        if (root.val > L) sum += rangeSumBST(root.left, L, R);
        if (root.val < R) sum += rangeSumBST(root.right, L, R);
        return sum;
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), where H is the height of the tree.

---

7. Find the Kth Smallest Element in a Binary Tree (Not BST)

Problem Explanation:

Find the Kth smallest element in a binary tree by converting the binary tree into a BST and then finding the element.

Example:

Input:

```markdown
        5
       / \
      1   7
     /   / \
    3   6   8
```

K = 3, Output: `5`

Approach:

- Perform an inorder traversal to collect values.
- Sort the values and then return the Kth element.

Java Code:

```java
class KthSmallestInBinaryTree {
    List<Integer> nodes = new ArrayList<>();

    public int kthSmallest(TreeNode root, int k) {
        inorder(root);
        Collections.sort(nodes); // Since it's not a BST, we need to sort
        return nodes.get(k - 1);
    }

    private void inorder(TreeNode node) {
        if (node == null) return;
        inorder(node.left);
        nodes.add(node.val);
        inorder(node.right);
    }
}
```

Time Complexity:

- **Time Complexity:** O(N log N), where N is the number of nodes (due to sorting).
- **Space Complexity:** O(N), for storing node values.

---

8. Count BST Subtrees That Lie in a Given Range

Problem Explanation:

Count all the BST subtrees within a binary tree that have all values in a specified range `[L, R]`.

Example:

Input:

```markdown
      10
     /  \
    5    15
   / \   / \
  1   8 12  18
```

Range: `` `[5, 15]` `` Output: `` `3` ``

Java Code:

```java
class CountBSTSubtrees {
    public int countBSTSubtrees(TreeNode root, int L, int R) {
        int[] count = new int[1];
        countBST(root, L, R, count);
        return count[0];
    }

    private boolean countBST(TreeNode node, int L, int R, int[] count) {
        if (node == null) return true;

        boolean left = countBST(node.left, L, R, count);
        boolean right = countBST(node.right, L, R, count);

        if (left && right && node.val >= L && node.val <= R) {
            count[0]++;
            return true;
        }
        return false;
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), where H is the height of the tree.

---

9. Construct BST from Preorder Traversal

Problem Explanation:

Given a preorder traversal of a BST, reconstruct the BST.

Example:

Input: Preorder = `` `[8, 5, 1, 7, 10, 12]` `` Output:

markdown

```markdown
        8
       / \
      5  10
     / \   \
    1   7   12
```

Java Code:

java

```java
class ConstructBSTFromPreorder {
    public TreeNode bstFromPreorder(int[] preorder) {
        return buildTree(preorder, Integer.MAX_VALUE, new int[]{0});
    }

    private TreeNode buildTree(int[] preorder, int bound, int[] index) {
        if (index[0] == preorder.length || preorder[index[0]] > bound) return null;
        TreeNode root = new TreeNode(preorder[index[0]++]);
        root.left = buildTree(preorder, root.val, index);
        root.right = buildTree(preorder, bound, index);
        return root;
    }
}
```

Time Complexity:

- **Time Complexity:** O(N), where N is the number of nodes.
- **Space Complexity:** O(H), where H is the height of the tree.

---

10. All Possible Unique BSTs for Keys from 1 to N

Problem Explanation:

Generate all structurally unique BSTs that store values from **`1`** to **`N`**.

Example:

Input: **`N = 3`** Output: 5 unique BSTs.

Java Code:

java

```java
class UniqueBSTs {
    public List<TreeNode> generateTrees(int n) {
        if (n == 0) return new ArrayList<>();
        return generate(1, n);
    }

    private List<TreeNode> generate(int start, int end) {
        List<TreeNode> result = new ArrayList<>();
        if (start > end) {
            result.add(null);
            return result;
        }

        for (int i = start; i <= end; i++) {
            List<TreeNode> leftTrees = generate(start, i - 1);
            List<TreeNode> rightTrees = generate(i + 1, end);

            for (TreeNode left : leftTrees) {
                for (TreeNode right : rightTrees) {
                    TreeNode root = new TreeNode(i);
                    root.left = left;
                    root.right = right;
                    result.add(root);
                }
            }
        }
        return result;
    }
}
```

Time Complexity:

- **Time Complexity:** O(2^N), because of the recursive structure.
- **Space Complexity:** O(N), for recursion stack and storing the results.

Let me know if you need further details on any of these problems!

ChatGPT can make mistakes. Check important info.