Project Title: Interesting Binary Tree and Binary Search Tree Methods

Overview

In this assessment/exam, you are provided with starter code (see Solution.java for Java and Solution.py for Python) that defines the structure of your project. The starter code includes class definitions for a basic Node, a BinaryTree class with four methods, and a BinarySearchTree class that extends BinaryTree with three additional methods. Each method currently contains "Todo" comments. Your task is to implement these 7 methods.

You may choose to write your solution in Java or Python.

Provided Starter Code

The starter code includes the following structure:

- **Node Class:**
  A basic class representing a tree node, which includes a value and pointers/references to the left and right children.
- **BinaryTree Class:**
  Contains the following methods (each marked with a "Todo" comment):
  i. **levelOrder:** Perform a level order (breadth-first) traversal.
  ii. **isComplete:** Check whether the binary tree is complete.
  iii. **countLeaves:** Count the number of leaf nodes.
  iv. **pathSum:** Determine if there exists a root-to-leaf path with a specified sum.
- **BinarySearchTree Class:**
  Extends BinaryTree and includes these additional methods:
  v. **validateBST:** Check whether the tree satisfies the Binary Search Tree (BST) properties.
  vi. **rangeSearch:** Retrieve all node values within a given inclusive range.
  vii. **balance:** Rebuild the BST so that it is balanced (minimizes its height).
- **Main/Test Suite:**
  A main method (or equivalent in Python) is provided with extensive test cases (minimum eight per method) using if/else conditions. These tests compare expected results to the output of your implementations. You must ensure that all tests pass by correctly implementing the "Todo" sections.

Detailed Method Descriptions

1. Level Order Traversal (`levelOrder`)

- **Objective:**
  Implement a method to traverse the tree level by level (breadth-first) and return the node values grouped by level.
- **Description:**
  Your method should return a collection (for example, a list of lists in Python or a List of List in Java) where each inner list contains the node values found at that level. Use an iterative approach with a queue to traverse the tree.
- **Key Points:**
  - Handle trees of any shape, including those with missing nodes.
  - Make sure the output correctly groups values by level.

2. Check Completeness (`isComplete`)

- **Objective:**
  Create a method to determine if a binary tree is complete.
- **Description:**
  A complete binary tree is one in which every level except possibly the last is fully filled and all nodes in the last level are as far left as possible. The method should return a boolean indicating the tree's completeness.
- **Key Points:**
  - Use a level order traversal to detect missing nodes.
  - Once a missing (null) node is encountered, all subsequent nodes must also be null.
  - Consider edge cases such as empty trees and skewed trees.

3. Count Leaf Nodes (`countLeaves`)

- **Objective:**
  Implement a method that counts the number of leaf nodes in the binary tree.
- **Description:**
  A leaf node has no children. Traverse the tree and count all nodes that do not have any children.
- **Key Points:**

- Differentiate this method from those that return the actual list of leaves.
- Ensure proper handling of trees with various structures, including skewed trees.

4. Root-to-Leaf Path Sum (`pathSum`)

- **Objective:**
  Write a method to determine if there exists any root-to-leaf path such that the sum of the node values equals a given target.
- **Description:**
  Starting at the root and traversing to any leaf, the method should check whether there is a path whose node values add up to the target sum. Return a boolean value.
- **Key Points:**
  - Use recursion (or an iterative method) to explore each path.
  - Correctly handle both positive and negative integers.
  - Include test cases for single-node trees and deeper trees.

5. Validate Binary Search Tree (`validateBST`)

- **Objective:**
  Implement a method to verify if a binary tree satisfies the properties of a BST.
- **Description:**
  For a valid BST, every node's left subtree must contain values less than the node's value, and every node's right subtree must contain values greater than the node's value. Your method should return a boolean indicating if the tree is a valid BST.
- **Key Points:**
  - Use recursion with boundary values to validate each node.
  - Consider edge cases such as duplicate values (if applicable) and empty trees.

6. Range Search in BST (`rangeSearch`)

- **Objective:**
  Create a method to retrieve all node values within a specified range.
- **Description:**
  Given two boundary values (low and high), the method should return a sorted collection of all node values in the BST that fall within this inclusive range.
- **Key Points:**

- Leverage BST properties to optimize the search.
- Use in-order traversal to efficiently collect values.
- Ensure the final result is sorted.

7. Balance the BST (`balance`)

- **Objective:**
  Implement a method to balance an unbalanced BST.
- **Description:**
  One approach is to perform an in-order traversal to gather the node values into a sorted list/array, then rebuild the tree by recursively choosing the median as the root. The goal is to minimize the height of the tree.
- **Key Points:**
  - The balanced tree must still satisfy BST properties.
  - Ensure that the method works efficiently for trees of all sizes.
  - Handle edge cases such as empty trees and trees that are already balanced.

Happy coding, and good luck with your implementation!