Implementing AVL Tree Balancing in Java

Objective:

To build on your understanding of binary search trees by implementing a self-balancing binary search tree, specifically an **AVL tree**. In an AVL tree, the heights of two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is performed to restore this property.

This lab will introduce tree rotations and balancing techniques, with a focus on ensuring optimal performance for operations like insertion, deletion, and searching.

This lab complements the previous binary search tree lab and will take approximately 3-4 hours to complete.

Requirements:

- JDK 8 or higher
- Your favourite Java IDE
- Completion of the previous binary search tree lab

Tasks:

---

1. Modify the Node Class to Include Height Information

In this task, you'll modify the Node<T> class to include a height field, which will be used to track the height of each node for rebalancing purposes.

CODE:

```
// Node class with left, right, height, and generic data
public class Node<T> {
   T data;
   Node<T> left, right;

   // Constructor to initialize data, children, and height
   public Node(T data) {
      this.data = data;
      left = right = null;
   }
}
```

---

2. Create the AVL Tree Class

The AVL tree is an extension of the binary search tree that includes balancing after each insertion. You will modify the insert method to include balancing logic.

- Implement the *getHeight* method to return the height of a node.
- Implement the *getBalance* method to calculate the balance factor (difference in height between left and right subtrees).
- Implement the rotation methods (*leftRotate* and *rightRotate*) to balance the tree when it becomes unbalanced.

CODE:

```
// AVL Tree class that extends BinarySearchTree

public class AVLTree<T extends Comparable<T>> {

  Node<T> root;

  // Get the height of the node

  private int getHeight(Node<T> node) {

  }

  // Get the balance factor of the node

  private int getBalance(Node<T> node) {

  }

  // Right rotate around a given node (single rotation)

  private Node<T> rightRotate(Node<T> y) {

  }

  // Left rotate around a given node (single rotation)

  private Node<T> leftRotate(Node<T> x) {

  }


  // Insert method with AVL balancing

  public void insert(T data) {

  }
// Recursive helper function for insertion with balancing
  private Node<T> insertRec(Node<T> node, T data) {


    }


    // Perform the standard BST insertion
    }


    // Update the height of the node


    // Get the balance factor of this node


    // If the node becomes unbalanced, there are 4 cases:


    // Case 1: Left Left


    // Case 2: Right Right



    // Case 3: Left Right


    // Case 4: Right Left


    // Return the (unchanged) node pointer
    return node;
  }
}
```

3. Implement the Delete Method with Balancing

Just like insertion, deletions in AVL trees require rebalancing. After deleting a node, you should ensure that the balance property is maintained.

- Implement the delete method for removing nodes.
- Rebalance the tree after each deletion using the rotation methods.

CODE:

```
// Delete method with AVL balancing

public void delete(T data) {

}


// Recursive helper function for deletion with balancing

private Node<T> deleteRec(Node<T> node, T data) {

  if (node == null) {

  }


  // Perform the standard BST delete

    // Node with only one child or no child

    if ((node.left == null) || (node.right == null)) {

      Node<T> temp = (node.left != null) ? node.left : node.right;

      // No child case

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

      // Copy the inorder successor's data to this node

      // Delete the inorder successor

    }

  }


  // Update height of the current node

  // Get the balance factor of this node

  // Rebalance the tree if necessary (similar to the insert function)

}
```

4. Test the AVL Tree Implementation

Write test code to verify that the AVL tree maintains its balance property after each insertion and deletion.

CODE:

```
public class Main {
  public static void main(String[] args) {
    AVLTree<Integer> avlTree = new AVLTree<>();

    // Insert nodes
    avlTree.insert(10);
    avlTree.insert(20);
    avlTree.insert(30);
    avlTree.insert(40);
    avlTree.insert(50);
    avlTree.insert(25);

    System.out.println("In-order Traversal:");
    avlTree.inOrder(avlTree.root); // Should print a balanced in-order sequence

    // Delete nodes
    avlTree.delete(30);
    avlTree.delete(20);

    System.out.println("\nIn-order Traversal after deletion:");
    avlTree.inOrder(avlTree.root); // Should maintain AVL balance after deletion
  }
}
```

5. Verify AVL Tree Properties

- Test that the AVL tree remains balanced after several insertions and deletions.
- Write additional tests to confirm the balance factor never exceeds 1 or -1.

Deliverables:

- All Java source code files.
- A paragraph describing what you did, any challenges you faced, and what you learned.

Bonus Tasks:

1. **Implement Level-Order Traversal:**
   Add a method to traverse the tree level by level (breadth-first).
2. **Measure Rotation Frequency:**
   Count the number of rotations performed during a series of insertions and deletions.

This lab provides a deep dive into the AVL tree balancing process, including tree rotations, insertion and deletion with balancing, and testing for correctness.