AVL Tree(C++)

This C++ file provides a complete and robust implementation of an **AVL Tree** encapsulated within a C++ class. An AVL tree is a self-balancing binary search tree, and this file is a masterclass in demonstrating the mechanisms that keep it perpetually balanced, guaranteeing efficient $O(\log n)$ performance for all major operations.

The core of this implementation lies in the recursive rInsert and Delete methods. Unlike a standard BST, after every insertion or deletion, these methods travel back up the tree during the recursive unwinding. At each ancestor node, they perform two critical actions:

1. **Update Height:** They recalculate the node's height.
2. **Check Balance:** They check the node's **balance factor**.

If an imbalance is detected (a balance factor of +2 or -2), the code triggers one of four powerful **rotations** (LL, RR, LR, RL) to restructure that part of the tree and restore the AVL property. The Delete method is particularly sophisticated; when removing a node with two children, it intelligently chooses a replacement from the taller subtree to help maintain the tree's balance.

By encapsulating this complex logic within an AVL class, the file provides a clean, safe, and reusable data structure.

**Class Definition and Helper Methods**

This section defines the AVL class and the essential helper functions that act as the tree's "sensors," constantly monitoring its height and balance.

* class Node { ... int height; }; The standard Node class is extended to include an integer height. Storing the height directly in each node is a key optimization that avoids costly recalculations.
* class AVL{ public: Node\* root; ... }; This defines the AVL class. The root pointer is the only data member, and all the logic for manipulating the tree is encapsulated within its public methods.
* int AVL::NodeHeight(Node \*p) { ... } This function calculates the height of a given node p. It's defined as 1 + the height of its taller child. The ternary expressions (p && p->lchild ? p->lchild->height : 0) are a concise way to safely get a child's height, returning 0 if the child is nullptr.
* int AVL::BalanceFactor(Node \*p) { ... } This is the core function for the AVL property. It calculates the **balance factor** by finding the difference between the left subtree's height and the right subtree's height (hl - hr). The tree is balanced at this node if the result is -1, 0, or 1.

**Rotation Functions**

These are the mechanical functions that physically restructure the tree to fix an imbalance. Each rotation is a precise sequence of pointer reassignments.

* Node\* AVL::LLRotation(Node \*p) { ... } The **Left-Left Rotation** is performed when a node p becomes imbalanced due to an insertion into the **left** subtree of its **left** child. The left child (pl) becomes the new root, and the original root (p) becomes its right child.
* Node\* AVL::RRRotation(Node \*p) { ... } The **Right-Right Rotation** is the mirror image of the LL rotation, used when an imbalance is caused by an insertion into the **right** subtree of the **right** child.
* Node\* AVL::LRRotation(Node \*p) { ... } The **Left-Right Rotation** is a double rotation. It's used when the imbalance is caused by an insertion into the **right** subtree of the **left** child. It's more complex, involving three nodes (p, pl, and plr) being rearranged.
* Node\* AVL::RLRotation(Node \*p) { ... } The **Right-Left Rotation** is the mirror image of the LR rotation, used for an imbalance caused by an insertion into the **left** subtree of the **right** child.

**Recursive Insert (rInsert)**

This is the main insertion method. It elegantly combines a standard recursive BST insertion with the AVL tree's self-balancing logic.

* if (p == nullptr){ ... t->height = 1; ... } This is the **base case**. When an empty spot is found, a new node t is created. Its height is initialized to 1, as a new leaf node always has a height of 1.
* p->lchild = rInsert(p->lchild, key); This is the standard **recursive insertion**. The function calls itself on the left or right subtree to find the correct insertion point.
* p->height = NodeHeight(p); This is the first crucial AVL step. As the recursion unwinds (returns back up the tree), the height of each ancestor node p is **recalculated**, as it may have changed due to the insertion below it.
* if (BalanceFactor(p) == 2 && BalanceFactor(p->lchild) == 1) { return LLRotation(p); } This is the **rebalancing trigger**. Immediately after updating the height, the BalanceFactor is checked. If it's 2 or -2, the node is imbalanced. The code then checks the balance factor of the appropriate child to determine which of the four rotations (LL, LR, RR, or RL) is needed to fix the imbalance.

**Recursive Delete (Delete)**

This method handles the removal of a node, which is more complex than insertion because it can also cause imbalances that need fixing.

* if (key < p->data){ p->lchild = Delete(p->lchild, key); } This is the standard **recursive search** for the node to be deleted. The function calls itself on the appropriate subtree.
* else { if (NodeHeight(p->lchild) > NodeHeight(p->rchild)){ ... } else { ... } } This is the logic for when the node to delete has **two children**. It's a key optimization for maintaining balance.
  1. It first checks which subtree is **taller**.
  2. It then finds a replacement (either the **In-order Predecessor** from the left or the **In-order Successor** from the right) from the taller subtree.
  3. It copies the replacement's data into the current node p.
  4. Finally, it makes a recursive call to Delete to remove the original predecessor/successor node.
* if (BalanceFactor(p) == 2 && BalanceFactor(p->lchild) == 0){ return LLRotation(p); } Similar to insertion, as the recursion unwinds after a deletion, it updates heights and checks balance factors at each level. If an imbalance is found, it performs the necessary rotation. Deletion introduces new cases, such as the **L0 and R0 rotations**, where the child's balance factor is 0, which are handled here.