AVL Trees

This C++ file provides a brilliant, clear implementation of an **AVL Tree**, a special type of self-balancing Binary Search Tree. The primary goal of an AVL tree is to automatically maintain a balanced structure, which guarantees that operations like search, insert, and delete are always highly efficient (with a time complexity of $O(\log n)$), even in the worst case.

The magic of the AVL tree lies in a property called the **balance factor**. For every node, the difference between the height of its left subtree and the height of its right subtree must be -1, 0, or 1. If an insertion or deletion violates this rule, the tree becomes imbalanced.

This file masterfully demonstrates the rebalancing mechanism through a set of **rotations**. When an imbalance is detected, the tree performs one of four possible rotations (LL, RR, LR, or RL) to restore its balance. The core of the file is the RInsert function, which combines a standard recursive BST insertion with the crucial logic to check the balance factor and trigger the correct rotation after each insertion.

**Helper Functions (NodeHeight and balanceFactor)**

These are two essential helper functions that provide the information needed to keep the tree balanced. They are the "eyes and ears" of the AVL tree.

* struct Node{ ... int height; ... }; The Node structure is extended from a standard BST node to include an integer height. This field is critical, as it stores the height of the subtree rooted at this node, avoiding the need to recalculate it constantly.
* int NodeHeight(struct Node \*p){ ... } This function calculates the height of a given node p. The height of a node is defined as 1 + the height of its taller child. The logic p && p->lchild ? p->lchild->height : 0; is a concise way to handle the case where a child is NULL, correctly treating its height as 0.
* int balanceFactor(struct Node \*p){ ... } This function is the heart of the AVL property. It calculates the **balance factor** of a node by subtracting the right subtree's height from the left subtree's height (height(L) - height(R)). If the result is not in the set {-1, 0, 1}, the node is considered imbalanced.

**Rotation Functions (LL, LR, RR, RL)**

These four functions are the "mechanisms" that physically restructure the tree to restore balance after an insertion has caused an imbalance.

* struct Node \*LLRotation(struct Node \*p){ ... } This is the **Left-Left Rotation**. It's performed when an imbalance at node p is caused by an insertion into the **left** child of p's **left** subtree.
  + struct Node \*pl = p->lchild; pl is the left child of the imbalanced node p.
  + pl->rchild = p; p->lchild = plr; This is the core rotation. The left child pl becomes the new root of this subtree. The original root p becomes the right child of pl. The original right child of pl (plr) is moved to become the new left child of p.
  + p->height = NodeHeight(p); pl->height = NodeHeight(pl); After the pointers are rewired, the heights of the affected nodes (p and pl) must be recalculated.
* struct Node \*RRRotation(struct Node \*p){ ... } This is the **Right-Right Rotation**, which is the mirror image of the LL Rotation. It's used when an imbalance is caused by an insertion into the **right** child of the **right** subtree. The logic is symmetrical to the LL rotation.
* struct Node \*LRRotation(struct Node \*p){ ... } This is the **Left-Right Rotation**. It handles an imbalance caused by an insertion into the **right** child of the **left** subtree. It's a two-step rotation.
  + struct Node \*pl = p->lchild; struct Node \*plr = pl->rchild; pl is the left child of p, and plr is the right child of pl.
  + plr->lchild = pl; plr->rchild = p; The node plr becomes the new root of this subtree, with pl and p becoming its new left and right children, respectively. This complex re-wiring effectively performs two simple rotations in one function.
* struct Node \*RLRotation(struct Node \*p){ ... } This is the **Right-Left Rotation**, the mirror image of the LR Rotation. It's used when an imbalance is caused by an insertion into the **left** child of the **right** subtree.

**Recursive Insert (RInsert)**

This is the main function that drives the AVL tree. It combines a standard recursive BST insertion with the rebalancing logic.

* if(p==NULL){ ... t->height = 1; ... } This is the **base case**. When an empty spot is found, a new node is created. Crucially, its height is initialized to 1 because a new leaf node always has a height of 1.
* p->lchild = RInsert(p->lchild, key); This is the standard **recursive BST insertion**. The function calls itself on the left or right subtree to find the correct insertion point.
* p->height = NodeHeight(p); **After** a node has been inserted somewhere in a subtree below p, the recursion unwinds. On its way back up, it's essential to **recalculate the height** of each ancestor node, as it may have changed.
* if(balanceFactor(p) == 2 && balanceFactor(p->lchild) == 1) return LLRotation(p); This is the **rebalancing trigger**. After updating the height, the function immediately checks the balanceFactor of the current node p.
  + If the balance factor is 2, it means the left subtree is too tall. It then checks the balance factor of the left child (p->lchild) to determine if it's an LL (1) or LR (-1) case.
  + If the balance factor is -2, the right subtree is too tall, and it similarly checks the right child to determine if it's an RR (-1) or RL (1) case. The function then calls the appropriate rotation, which returns the new root of the balanced subtree.