深入理解并实现基本的 AVL 树数据结构

杨子凡

Jun 14, 2025

二叉搜索树（BST）是一种基础的数据结构，但其存在显著局限性。当插入有序数据时，BST 可能退化成链表，导致搜索、插入和删除操作的时间复杂度降至 $O(n)$，严重降低效率。这一缺陷凸显了引入平衡二叉搜索树的必要性，它能动态维持树高平衡，确保操作复杂度稳定在 $O(\log n)$。AVL 树正是为此而生，由 Adelson-Velsky 和 Landis 在 1962 年提出，其核心目标是通过旋转操作动态调整树结构，保证任意节点高度差不超过 1。本文旨在深入解析 AVL 树的工作原理，手把手实现插入、删除和旋转等基础操作，并对比其他平衡树如红黑树的适用场景，帮助读者在工程实践中做出合理选择。

1　AVL 树核心概念

AVL 树的平衡性依赖于平衡因子（Balance Factor）这一关键指标。平衡因子定义为节点左子树高度减去右子树高度，数学表示为 $\text{BF}(node) = \text{height}(\text{left\_subtree}) - \text{height}(\text{right\_subtree})$。一个 AVL 树平衡的条件是任意节点的平衡因子属于集合 $\{-1, 0, 1\}$。树高维护是动态过程：叶子节点高度为 0，空树高度为 -1，每次插入或删除后需递归更新高度。失衡类型有四种：LL 型（左子树更高且左子树的左子树更高）、RR 型（右子树更高且右子树的右子树更高）、LR 型（左子树更高但左子树的右子树更高）和 RL 型（右子树更高但右子树的左子树更高）。这些失衡类型决定了后续旋转策略的选择。

2　旋转操作：AVL 树的平衡基石

旋转操作是维持 AVL 树平衡的核心机制。右旋（RR Rotation）用于解决 LL 型失衡：以节点链 A → B → C 为例（其中 B 是 A 的左子节点，C 是 B 的左子节点），旋转时 B 成为新根节点，A 变为 B 的右子节点，C 保持为 B 的左子节点，同时更新相关节点高度。左旋（LL Rotation）针对 RR 型失衡：节点链 A → B → C（B 是 A 的右子节点，C 是 B 的右子节点），旋转后 B 成为新根节点，A 变为 B 的左子节点，C 保持为 B 的右子节点。组合旋转处理更复杂失衡：LR 旋转先对失衡节点的左子节点执行左旋，再对自身执行右旋；RL 旋转先对右子节点执行右旋，再对自身执行左旋。旋转后必须立即更新节点高度，确保平衡因子计算准确。

3　AVL 树的操作实现

节点结构设计是 AVL 树实现的基础。以下 Python 代码定义了一个 AVL 节点类，包含键值、左右子节点指针和高度属性：

```
class AVLNode:
    def __init__(self, key):
```

```
        self.key = key
        self.left = None
        self.right = None
        self.height = 0 # 当前节点高度
```

此代码中，key 存储节点值，left 和 right 分别指向左子树和右子树，height 记录节点高度（初始化为 0）。辅助函数简化操作：get_height(node) 处理空节点情况，返回 -1；update_height(node) 计算节点高度为 $1 + \max(\text{get\_height}(node.left), \text{get\_height}(node.right))$；get_balance(node) 返回平衡因子 $\text{BF}(node)$。插入操作遵循标准 BST 插入逻辑，但插入后需回溯更新高度并检查平衡因子：若失衡则触发旋转。删除操作类似，处理 BST 删除的三种情况（无子节点、单子节点或双子节点），删除后同样回溯更新高度和平衡。搜索操作与 BST 一致，时间复杂度为 $O(\log n)$。

4 关键代码实现详解

旋转函数的实现是 AVL 树的核心。以下以左旋函数为例，详细解释其逻辑：

```
def left_rotate(z):
    y = z.right
    T2 = y.left
    # 旋转
    y.left = z
    z.right = T2
    # 更新高度（先更新 z 再更新 y）
    update_height(z)
    update_height(y)
    return y # 返回新的子树根
```

此函数解决 RR 型失衡：参数 z 是失衡节点。第一步，y 指向 z 的右子节点，T2 指向 y 的左子树。第二步，执行旋转：y.left 指向 z（使 z 成为 y 的左子节点），z.right 指向 T2（将 y 的原左子树挂载到 z 的右侧）。第三步，更新高度：先更新 z 的高度（因其子树可能变化），再更新 y 的高度。最后返回 y 作为新子树的根节点。平衡调整逻辑在插入后调用，以下代码处理四种失衡类型：

```
def balance(node):
    balance_factor = get_balance(node)
    # LL 型
    if balance_factor > 1 and get_balance(node.left) >= 0:
        return right_rotate(node)
    # RR 型
    if balance_factor < -1 and get_balance(node.right) <= 0:
        return left_rotate(node)
    # LR 型
    if balance_factor > 1 and get_balance(node.left) < 0:
```

```
        node.left = left_rotate(node.left)
        return right_rotate(node)
    # RL 型
    if balance_factor < -1 and get_balance(node.right) > 0:
        node.right = right_rotate(node.right)
        return left_rotate(node)
    return node # 无需旋转
```

此函数首先获取当前节点的平衡因子。对于 LL 型失衡（平衡因子大于 1 且左子节点平衡因子非负），直接右旋；对于 RR 型（平衡因子小于 -1 且右子节点平衡因子非正），直接左旋；对于 LR 型（平衡因子大于 1 但左子节点平衡因子为负），先对左子节点左旋转换为 LL 型，再对自身右旋；RL 型类似，先右旋右子节点再左旋自身。无需旋转时返回原节点。

5 复杂度分析与正确性验证

AVL 树的时间复杂度源于其严格平衡性。树高 h 满足 $h = O(\log n)$，可通过斐波那契数列证明：最小高度树对应斐波那契树，节点数 n 满足 $n \geq F_{h+2} - 1$，其中 F 是斐波那契数列，推导出 $h \leq 1.44 \log_2(n+1)$。因此，插入和删除操作时间复杂度为 $O(\log n)$（旋转操作本身是 $O(1)$），搜索操作同样为 $O(\log n)$。正确性验证需结合两种方法：中序遍历应输出有序序列（验证 BST 属性）；递归检查每个节点平衡因子是否在 $[-1, 0, 1]$ 内（验证平衡性）。测试用例设计包括有序插入、随机插入和混合操作（插入后删除），覆盖边界情况如空树或单节点树。

6 AVL 树 vs. 其他平衡树

AVL 树与红黑树的对比至关重要。AVL 树平衡更严格（高度差不超过 1），带来更优的查找效率（树高更低），但插入或删除时可能需更多旋转操作。红黑树平衡相对宽松（高度差可至 2），旋转次数较少，适合写密集型场景。适用场景上，AVL 树优先用于读密集型应用如数据库索引（PostgreSQL 部分实现使用 AVL），红黑树适用于写频繁的内存存储如 C++ STL 的 `std::map`。与 B/B+ 树相比，B 树专为磁盘设计（减少 I/O 次数），AVL 树更适用于内存操作。

7 实际应用场景

AVL 树在多个领域发挥重要作用。数据库索引是其典型应用，如 PostgreSQL 利用 AVL 树实现高效查询；内存中的有序数据结构（如某些语言的标准库）可选 AVL 作为底层实现；游戏开发中，AVL 树用于空间分区加速碰撞检测或对象查询，确保实时性能。

AVL 树的核心价值在于其强平衡性保证高效查找（时间复杂度 $O(\log n)$）。实现关键点包括动态维护高度和四种旋转策略（右旋、左旋、LR 和 RL）。适用建议上，读多写少场景（如缓存系统）优先考虑 AVL 树，写频繁场景（如实时日志处理）则推荐红黑树。延伸学习可尝试实现红黑树或伸展树（Splay Tree），深化对平衡树的理解。