深入理解并实现基本的二叉平衡树（Balanced Binary Tree）数据结构

叶家炜

Jul 28, 2025

二叉搜索树（Binary Search Tree，简称 BST）是一种常见的数据结构，它支持高效的查找、插入和删除操作，理想情况下时间复杂度为 $O(\log n)$。然而，BST 存在一个重大缺陷：当数据以特定顺序插入时，例如升序或降序序列，树可能退化为链表结构，导致操作时间复杂度恶化至 $O(n)$。这种退化风险在动态数据集中尤为显著。为解决这一问题，平衡二叉树被提出，其核心思想是通过动态调整树结构来保持高度平衡，确保所有操作在 $O(\log n)$ 时间内完成。平衡二叉树在数据库索引、高效查找系统等场景中应用广泛。本文将以 AVL 树为例，深入解析其原理并手写实现，帮助读者掌握这一关键数据结构。

1　2. 平衡二叉树核心概念

平衡二叉树的定义基于平衡因子（Balance Factor）这一核心指标。平衡因子用于量化节点的失衡程度，其计算公式为：

$$\text{BF(node)} = \text{height(left_subtree)} - \text{height(right_subtree)}$$

其中，height 表示子树的高度，定义为从根节点到最深叶节点的边数。AVL 树作为平衡二叉树的经典实现，要求所有节点的平衡因子绝对值不超过 1，即 $|\text{BF}| \leq 1$。这一条件确保树高度始终维持在 $O(\log n)$ 级别。例如，对于一个包含 n 个节点的 AVL 树，其最大高度为 $1.44 \log_2(n+1)$，远优于退化 BST 的线性高度。树高的动态维护是实现平衡的基础，每次插入或删除操作后需重新计算高度值，以便检测失衡。

2　3. 失衡与旋转操作（核心重点）

当 AVL 树的节点平衡因子绝对值大于 1 时，树进入失衡状态，需通过旋转操作修复。失衡分为四种类型：LL 型（左左失衡）、RR 型（右右失衡）、LR 型（左右失衡）和 RL 型（右左失衡）。LL 型失衡发生在节点左子树高于右子树且新节点插入左子树的左侧时，需执行右旋操作。右旋通过将失衡节点降为右子节点，并提升其左子节点为新根来重组子树结构。例如，节点 Z 失衡后，其左子节点 Y 成为新根，Y 的右子树 T3 成为 Z 的左子树，从而降低高度差。RR 型失衡则需左旋操作，其原理与右旋对称。

LR 型失衡更复杂，发生在节点左子树高于右子树但新节点插入左子树的右侧时。修复需分两步：先对失衡节点的左子节点执行左旋，将其转换为 LL 型，再对原节点执行右旋。RL 型失衡与之对称，需先右旋后左旋。旋转操作的本质是调整指针指向，重组子树以恢复平衡，每次旋转后必须更新相关节点的高度值。这些操作时间复杂度为 $O(1)$，仅涉及常数次指针赋值。

3 4. AVL 树节点设计

AVL 树的节点设计需包含键值、左右子节点指针及高度属性。以下 Python 代码展示了节点类的实现：

```
class AVLNode:
    def __init__(self, key):
        self.key = key # 节点存储的键值
        self.left = None # 左子节点指针
        self.right = None # 右子节点指针
        self.height = 1 # 节点高度，初始为 1（叶子节点高度为 1）
```

该代码定义了一个 AVLNode 类，其中 key 存储数据值，left 和 right 分别指向左右子树。height 属性记录节点高度，初始化时为 1，因为叶子节点无子树。高度维护是 AVL 树的核心，需在插入或删除后更新。例如，当节点为叶子时，高度保持为 1；若有子节点，高度基于子树最大值加 1 计算。

4 5. 关键操作实现

实现 AVL 树需先定义辅助函数。get_height(node) 处理空节点情况，返回 0；update_height(node) 根据左右子树高度更新节点高度；get_balance(node) 计算平衡因子。以下为旋转函数示例：

```
def right_rotate(z):
    y = z.left # y 是 z 的左子节点
    T3 = y.right # 保存 y 的右子树 T3
    y.right = z # 将 z 设为 y 的右子节点
    z.left = T3 # 将 T3 设为 z 的左子树
    update_height(z) # 更新 z 的高度
    update_height(y) # 更新 y 的高度
    return y # 返回新子树的根节点
```

这段代码实现了右旋操作。输入为失衡节点 z，其左子节点 y 被提升为新根。T3 临时存储 y 的右子树，以避免指针丢失。旋转后，z 成为 y 的右子节点，T3 附加到 z 左侧。最后调用 update_height 更新高度并返回新根 y。左旋函数与此对称。

插入操作遵循递归逻辑：先在 BST 中插入节点，再回溯更新高度并检查平衡。关键代码如下：

```
def insert(node, key):
    if not node:
        return AVLNode(key) # 空树时创建新节点
    if key < node.key:
        node.left = insert(node.left, key) # 递归插入左子树
    else:
        node.right = insert(node.right, key) # 递归插入右子树
    update_height(node) # 更新当前节点高度
```

```
    balance = get_balance(node) # 计算平衡因子
    if balance > 1 and key < node.left.key: # LL 型失衡
        return right_rotate(node)
    if balance > 1 and key > node.left.key: # LR 型失衡
        node.left = left_rotate(node.left) # 先左旋左子节点
        return right_rotate(node) # 再右旋当前节点
    # RR 和 RL 型处理类似（对称操作）
    return node # 返回调整后的节点
```

该代码首先递归插入节点，类似标准 BST。插入后调用 update_height 更新高度，并通过 get_balance 计算平衡因子。若检测到 LL 型失衡（平衡因子大于 1 且新键小于左子键），执行右旋。对于 LR 型（平衡因子大于 1 但新键大于左子键），先对左子节点左旋，再对当前节点右旋。RR 和 RL 型处理对称。

删除操作更复杂：先递归删除节点（处理零、一或二个子节点情况），然后更新高度并检查平衡。删除可能引发连锁失衡，需从叶节点回溯至根节点执行旋转。例如，删除节点后若其父节点失衡，需应用旋转修复。查找操作与 BST 相同，利用树平衡性确保 $O(\log n)$ 时间。

5 6. 复杂度分析

AVL 树的时间复杂度是其核心优势。查找、插入和删除操作均保证 $O(\log n)$ 最坏情况时间复杂度，因为树高度严格控制在 $\log n$ 量级。旋转操作本身为 $O(1)$，仅涉及指针调整。空间复杂度为 $O(n)$，用于存储节点和高度信息。与红黑树相比，AVL 树平衡更严格，查找性能更优（红黑树高度上限为 $2\log n$），但插入删除操作更频繁触发旋转，效率略低。红黑树通过放宽平衡条件（如允许部分失衡），减少旋转次数，适用于写操作频繁场景。

6 7. 完整代码实现

以下 Python 代码整合了 AVL 树的核心功能，包括节点类、旋转操作及插入删除逻辑。

```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

def get_height(node):
    return node.height if node else 0

def update_height(node):
    node.height = 1 + max(get_height(node.left), get_height(node.right))
```

```
def get_balance(node):
    return get_height(node.left) - get_height(node.right) if node else 0

def left_rotate(z):
    y = z.right
    T2 = y.left
    y.left = z
    z.right = T2
    update_height(z)
    update_height(y)
    return y

def right_rotate(z):
    y = z.left
    T3 = y.right
    y.right = z
    z.left = T3
    update_height(z)
    update_height(y)
    return y

def insert(node, key):
    if not node:
        return AVLNode(key)
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    update_height(node)
    balance = get_balance(node)
    if balance > 1 and key < node.left.key: # LL
        return right_rotate(node)
    if balance < -1 and key > node.right.key: # RR
        return left_rotate(node)
    if balance > 1 and key > node.left.key: # LR
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if balance < -1 and key < node.right.key: # RL
        node.right = right_rotate(node.right)
        return left_rotate(node)
```

```
    return node

# 删除操作类似，需处理子树重组和连锁旋转
```

此代码提供了可运行基础。`insert` 函数实现递归插入与平衡修复，支持所有四种失衡类型。测试时，建议设计序列验证：连续插入升序数据触发 RR 型失衡，降序数据触发 LL 型，乱序数据可能触发 LR 或 RL 型。删除操作需额外处理子树重组（例如，当节点有两个子节点时，用后继节点替换），并在回溯时检查连锁失衡。

7 8. 平衡树的其他变种

除 AVL 树外，平衡树有多种变种。红黑树（Red-Black Tree）放宽平衡条件，允许部分节点失衡，减少旋转次数，适用于高频写入场景，如 C++ STL 的 `map` 和 `set`。伸展树（Splay Tree）基于局部性原理，将最近访问节点移至根节点，提升缓存效率，常用于网络路由。B 树和 B+ 树是多路平衡树，专为磁盘存储优化，通过增加分支因子减少 I/O 操作，广泛应用于数据库索引（如 MySQL InnoDB）。这些变种在不同场景下权衡平衡严格性与操作开销。

8 9. 实际应用场景

平衡二叉树在现实系统中扮演关键角色。数据库引擎如 MySQL 的 InnoDB 使用 B+ 树实现索引，支持高效范围查询。语言标准库中，C++ 的 `std::map` 和 Java 的 `TreeMap` 基于红黑树，提供有序键值存储。游戏开发中，平衡树用于空间分区数据结构（如 KD-Tree），加速碰撞检测。其他场景包括文件系统索引、编译器符号表和实时数据处理系统，其共同需求是保障最坏情况性能。

平衡二叉树的核心价值在于以额外空间（高度存储）换取时间效率，确保所有操作在 $O(\log n)$ 最坏时间复杂度内完成。AVL 树的实现关键包括高度动态维护和四种旋转策略（LL、RR、LR、RL），这些机制能有效修复失衡。进阶方向可探索 B 树在磁盘存储中的应用，或并发平衡树设计以支持多线程环境。读者可通过可视化工具（如在线 AVL Tree Visualizer）深化理解，并尝试习题：给定序列绘制 AVL 树形成过程，实现非递归插入，或统计旋转次数。