深入理解并实现基本的平衡二叉树（Balanced Binary Tree）数据结构

黄京

Sep 16, 2025

本文将深入探讨平衡二叉树的核心概念，重点解析为何需要它、它是如何通过旋转操作维持平衡的。我们将以最经典的 AVL 树为例，逐步拆解其四大旋转操作，并最终用代码（C++）完整实现一个具备插入、删除和查询功能的 AVL 树数据结构。无论你是正在学习数据结构的学生，还是希望巩固基础的开发者，这篇文章都将为你提供清晰的指引。

二叉搜索树（BST）是一种常见的数据结构，它具有高效的搜索、插入和删除操作，理想情况下的时间复杂度为 $O(h)$，其中 h 是树的高度。BST 的定义基于每个节点的值大于其左子树的所有值且小于其右子树的所有值。然而，BST 有一个致命的缺陷：当插入的数据是有序的，例如序列 1, 2, 3, 4, 5，BST 会退化成一条链表，此时高度 h 等于节点数 n，操作时间复杂度降至 $O(n)$，效率急剧下降。

为了解决这个问题，平衡二叉树应运而生。平衡二叉树的核心思想是在 BST 的基础上，通过某些策略控制树的高度，使其尽可能保持「矮胖」的形状，从而保证操作效率始终维持在 $O(\log n)$。常见的平衡树类型包括 AVL 树、红黑树和 Treap 等。本文将聚焦于最基础且易于理解的 AVL 树，因为它提供了严格的平衡保证和相对简单的实现方式。

1　AVL 树：定义与核心概念

AVL 树是一种自平衡的二叉搜索树，由 Adelson-Velsky 和 Landis 在 1962 年提出。它首先满足 BST 的所有性质，但额外要求每个节点的平衡因子（Balance Factor, BF）必须保持在 -1、0 或 1 的范围内。平衡因子定义为该节点左子树的高度减去右子树的高度，即 $BF = h_{\text{left}} - h_{\text{right}}$。如果任何节点的平衡因子超出这个范围，树就被认为是不平衡的，需要通过旋转操作来调整。

计算节点高度是 AVL 树操作的基础。通常，我们约定空节点的高度为 -1（这便于计算），而非空节点的高度为其左右子树高度的最大值加一。例如，一个叶子节点的高度为 0，因为它没有子节点。在代码中，我们会频繁调用一个辅助函数来获取或更新节点的高度，以确保平衡因子的正确计算。

2　维持平衡的魔法：AVL 旋转

当插入或删除节点导致平衡因子超出允许范围时，AVL 树通过旋转操作来恢复平衡。旋转的目的是在不破坏 BST 排序性质的前提下，对局部子树进行重组，以降低高度。旋转操作分为四种基本类型，对应不同的不平衡情况。

第一种情况是左左（LL）情况。这发生在节点的平衡因子大于 1（即左子树比右子树高太多），且其左孩子的平衡因子大于或等于 0（表示不平衡源于左孩子的左子树）。解决方案是执行右旋操作：以该节点为轴，将其左孩

子提升为新的根节点，原节点成为新根节点的右孩子，同时调整子树指针以维持 BST 性质。
第二种情况是右右（RR）情况。这发生在节点的平衡因子小于 -1（即右子树比左子树高太多），且其右孩子的平衡因子小于或等于 0（表示不平衡源于右孩子的右子树）。解决方案是执行左旋操作：以该节点为轴，将其右孩子提升为新的根节点，原节点成为新根节点的左孩子，并调整子树指针。
第三种情况是左右（LR）情况。这发生在节点的平衡因子大于 1，但其左孩子的平衡因子小于 0（表示不平衡源于左孩子的右子树）。解决方案需要两步：先对左孩子执行左旋操作（将其转化为 LL 情况），再对原节点执行右旋操作。
第四种情况是右左（RL）情况。这发生在节点的平衡因子小于 -1，但其右孩子的平衡因子大于 0（表示不平衡源于右孩子的左子树）。解决方案同样需要两步：先对右孩子执行右旋操作（将其转化为 RR 情况），再对原节点执行左旋操作。
理解这些旋转的关键在于可视化子树的重组过程，但本文避免使用图片，因此建议读者在脑海中模拟指针的调整。旋转操作完成后，树的高度会减少，平衡因子恢复正常，从而确保整体效率。

3 代码实现：手把手构建 AVL 树

我们将使用 C++ 语言实现 AVL 树。代码实现分为多个步骤，从定义节点类到实现核心操作函数。每个代码块后会有详细解读，以帮助理解。
首先，定义树节点类。节点包含整数值 `val`、左右子节点指针 `left` 和 `right`，以及高度 `height`。我们选择存储高度而非平衡因子，因为平衡因子可以通过高度计算得出。

```
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    int height;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr), height(0) {}
};
```

这段代码定义了一个简单的节点类，构造函数初始化值、指针和高度。高度初始化为 0，但后续会通过更新函数调整。
接下来，实现辅助函数。`getHeight` 函数用于获取节点高度，处理空节点情况；`updateHeight` 函数更新节点高度基于其子节点高度；`getBalanceFactor` 函数计算平衡因子。

```
int getHeight(TreeNode* node) {
    if (node == nullptr) return -1;
    return node->height;
}

void updateHeight(TreeNode* node) {
    if (node == nullptr) return;
    node->height = 1 + std::max(getHeight(node->left), getHeight(node->right));
```

```
}

int getBalanceFactor(TreeNode* node) {
   if (node == nullptr) return 0;
   return getHeight(node->left) - getHeight(node->right);
}
```

getHeight 函数检查节点是否为空，如果是则返回 -1，否则返回存储的高度。updateHeight 函数重新计算节点高度为左右子树高度的最大值加一。getBalanceFactor 函数返回左子树高度减右子树高度，直接使用高度值计算。

现在，实现旋转函数。左旋和右旋是基本操作，它们返回调整后的新根节点。

```
TreeNode* rotateLeft(TreeNode* y) {
   TreeNode* x = y->right;
   TreeNode* T2 = x->left;
   x->left = y;
   y->right = T2;
   updateHeight(y);
   updateHeight(x);
   return x;
}

TreeNode* rotateRight(TreeNode* x) {
   TreeNode* y = x->left;
   TreeNode* T2 = y->right;
   y->right = x;
   x->left = T2;
   updateHeight(x);
   updateHeight(y);
   return y;
}
```

在 rotateLeft 函数中，y 是原根节点，x 是其右孩子。操作将 x 的左子树（T2） attached 到 y 的右边，然后更新 y 和 x 的高度。右旋类似，但方向相反。旋转后，BST 性质保持不变，因为值的相对顺序没有改变。

基于旋转函数，实现平衡函数 balance。它检查节点的平衡因子，并根据四种情况调用相应的旋转。

```
TreeNode* balance(TreeNode* node) {
   if (node == nullptr) return nullptr;
   int bf = getBalanceFactor(node);
   if (bf > 1) {
      if (getBalanceFactor(node->left) >= 0) {
         return rotateRight(node); // LL case
```

```
        } else {
            node->left = rotateLeft(node->left); // LR case: first left rotate left child
            return rotateRight(node);
        }
    }
    if (bf < -1) {
        if (getBalanceFactor(node->right) <= 0) {
            return rotateLeft(node); // RR case
        } else {
            node->right = rotateRight(node->right); // RL case: first right rotate right
                ↪ child
            return rotateLeft(node);
        }
    }
    return node; // no need to balance
}
```

这个函数首先计算平衡因子。如果平衡因子大于 1，检查左孩子的平衡因子以区分 LL 或 LR 情况，并执行相应旋转。类似地处理平衡因子小于 -1 的情况。旋转后返回新根节点。

实现插入操作。插入是递归的，先执行标准 BST 插入，然后更新高度并平衡节点。

```
TreeNode* insert(TreeNode* node, int key) {
    if (node == nullptr) {
        return new TreeNode(key);
    }
    if (key < node->val) {
        node->left = insert(node->left, key);
    } else if (key > node->val) {
        node->right = insert(node->right, key);
    } else {
        return node; // duplicate keys not allowed
    }
    updateHeight(node);
    return balance(node);
}
```

插入函数递归地找到合适位置插入新节点。插入后，更新当前节点高度并调用 balance 来恢复平衡。返回调整后的根节点。

实现删除操作。删除同样递归，处理三种情况：无子节点、有一个子节点、有两个子节点。删除后更新高度并平衡。

```
TreeNode* remove(TreeNode* node, int key) {
```

```
    if (node == nullptr) return nullptr;
    if (key < node->val) {
        node->left = remove(node->left, key);
    } else if (key > node->val) {
        node->right = remove(node->right, key);
    } else {
        if (node->left == nullptr || node->right == nullptr) {
            TreeNode* temp = node->left ? node->left : node->right;
            delete node;
            return temp;
        } else {
            TreeNode* temp = node->right;
            while (temp->left != nullptr) {
                temp = temp->left;
            }
            node->val = temp->val;
            node->right = remove(node->right, temp->val);
        }
    }
    updateHeight(node);
    return balance(node);
}
```

删除函数首先找到要删除的节点。如果节点有一个或无子节点，直接删除并返回子节点。如果有两个子节点，找到中序遍历后继（右子树的最小值），复制值到当前节点，并递归删除后继节点。最后更新高度并平衡。

查找操作与普通 BST 相同，无需修改。

```
bool search(TreeNode* node, int key) {
    if (node == nullptr) return false;
    if (key == node->val) return true;
    if (key < node->val) return search(node->left, key);
    return search(node->right, key);
}
```

查找函数递归搜索值，返回是否存在。

4 测试与验证

为了验证 AVL 树的正确性，我们编写一个简单的测试程序。测试数据使用有序序列 [10, 20, 30, 40, 50, 25] 来演示 AVL 树如何避免退化。

```
#include <iostream>
```

```
#include <algorithm>
void inOrder(TreeNode* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    std::cout << node->val << " ";
    inOrder(node->right);
}
int main() {
    TreeNode* root = nullptr;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
    std::cout << "In-order traversal: ";
    inOrder(root); // should output sorted values
    std::cout << "\nHeight of tree: " << getHeight(root) << std::endl; // should be
        ↪ small
    // Test deletion if implemented
    root = remove(root, 30);
    std::cout << "After deletion, in-order: ";
    inOrder(root);
    std::cout << std::endl;
    return 0;
}
```

中序遍历应输出有序序列（10, 20, 25, 30, 40, 50），证明 BST 性质维持。树高度应远小于节点数，表示平衡。删除操作后，树应保持平衡和有序。

AVL 树通过严格的平衡条件确保了操作的时间复杂度为 $O(\log n)$，非常适合读多写少的场景。然而，其缺点在于插入和删除时需要频繁旋转，可能导致性能开销。相比之下，红黑树采用近似平衡，旋转次数较少，广泛应用于标准库如 C++ STL 的 map 和 set。

对于进一步学习，建议探索红黑树、B 树和 Splay 树等其他平衡数据结构。这些结构在不同场景下各有优势，例如 B 树用于数据库和文件系统。

5 附录 & 互动

完整代码可参考 GitHub 仓库示例。欢迎在评论区提问或分享您的实现经验。参考文献包括经典算法书籍和在线资源，如 Introduction to Algorithms by Cormen et al.。