7.5 AVL 树 *

在“二叉搜索树”章节中我们提到，在多次插入和删除操作后，二叉搜索树可能退化为链表。在这种情况下，所有操作的时间复杂度将从 $O(\log n)$ 劣化为 $O(n)$ 。

如图 7-24 所示，经过两次删除节点操作，这棵二叉搜索树便会退化为链表。

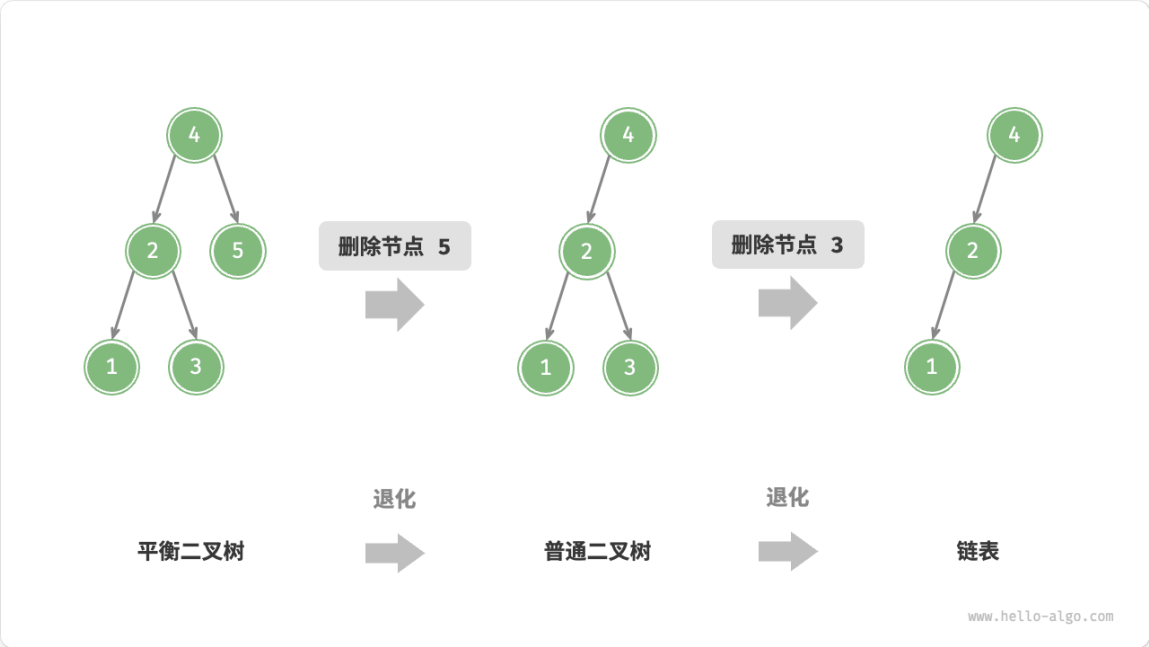

图 7-24 AVL 树在删除节点后发生退化

再例如，在图 7-25 所示的完美二叉树中插入两个节点后，树将严重向左倾斜，查找操作的时间复杂度也随之劣化。

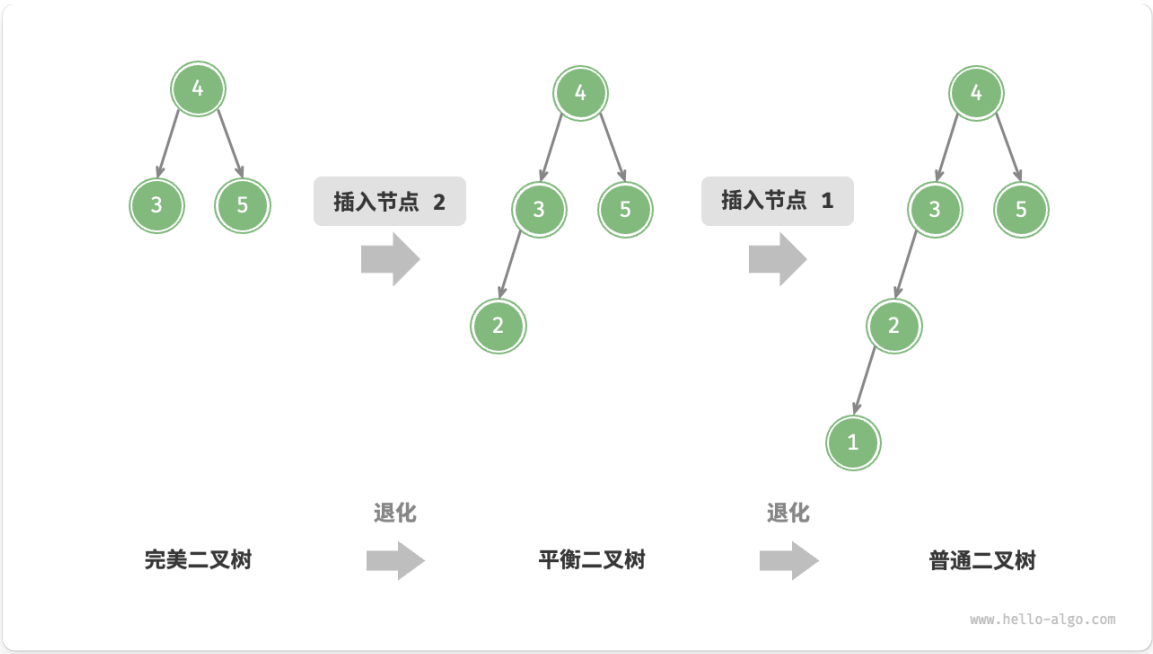

图 7-25 AVL 树在插入节点后发生退化

1962 年 G. M. Adelson-Velsky 和 E. M. Landis 在论文“An algorithm for the organization of information”中提出了 AVL 树。论文中详细描述了一系列操作，确保在持续添加和删除节点后，AVL 树不会退化，从而使得各种操作的时间复杂度保持在 $O(\log n)$ 级别。换句话说，在需要频繁进行增删查改操作的场景中，AVL 树能始终保持高效的数据操作性能，具有很好的应用价值。

7.5.1 AVL 树常见术语

AVL 树既是二叉搜索树，也是平衡二叉树，同时满足这两类二叉树的所有性质，因此是一种平衡二叉搜索树（balanced binary search tree）。

1. 节点高度

由于 AVL 树的相关操作需要获取节点高度，因此我们需要为节点类添加 `height` 变量：

Python

```python
class TreeNode:
    """AVL 树节点类"""
    def __init__(self, val: int):
        self.val: int = val                # 节点值
        self.height: int = 0               # 节点高度
        self.left: TreeNode | None = None  # 左子节点引用
        self.right: TreeNode | None = None # 右子节点引用
```

“节点高度”是指从该节点到它的最远叶节点的距离，即所经过的“边”的数量。需要特别注意的是，叶节点的高度为 0 ，而空节点的高度为 -1 。我们将创建两个工具函数，分别用于获取和更新节点的高度：

Python

avl_tree.py

```
def height(self, node: TreeNode | None) -> int:
    """获取节点高度"""
    # 空节点高度为 -1 ，叶节点高度为 0
    if node is not None:
        return node.height
    return -1

def update_height(self, node: TreeNode | None):
    """更新节点高度"""
    # 节点高度等于最高子树高度 + 1
    node.height = max([self.height(node.left), self.height(node.right)]) +
1
```

2. 节点平衡因子

节点的平衡因子（balance factor）定义为节点左子树的高度减去右子树的高度，同时规定空节点的平衡因子为 0 。我们同样将获取节点平衡因子的功能封装成函数，方便后续使用：

Python

avl_tree.py

```
def balance_factor(self, node: TreeNode | None) -> int:
    """获取平衡因子"""
    # 空节点平衡因子为 0
    if node is None:
        return 0
    # 节点平衡因子 = 左子树高度 - 右子树高度
    return self.height(node.left) - self.height(node.right)
```

Tip

设平衡因子为 f ，则一棵 AVL 树的任意节点的平衡因子皆满足 $-1 \le f \le 1$ 。

7.5.2 AVL 树旋转

AVL 树的特点在于“旋转”操作，它能够在不影响二叉树的中序遍历序列的前提下，使失衡节点重新恢复平衡。换句话说，**旋转操作既能保持“二叉搜索树”的性质，也能使树重新变为“平衡二叉树”**。

我们将平衡因子绝对值 > 1 的节点称为“失衡节点”。根据节点失衡情况的不同，旋转操作分为四种：右旋、左旋、先右旋后左旋、先左旋后右旋。下面详细介绍这些旋转操作。

1. 右旋

如图 7-26 所示，节点下方为平衡因子。从底至顶看，二叉树中首个失衡节点是“节点 3”。我们关注以该失衡节点为根节点的子树，将该节点记为 `node` ，其左子节点记为 `child` ，执行“右旋”操作。完成右旋后，子树恢复平衡，并且仍然保持二叉搜索树的性质。

<1>

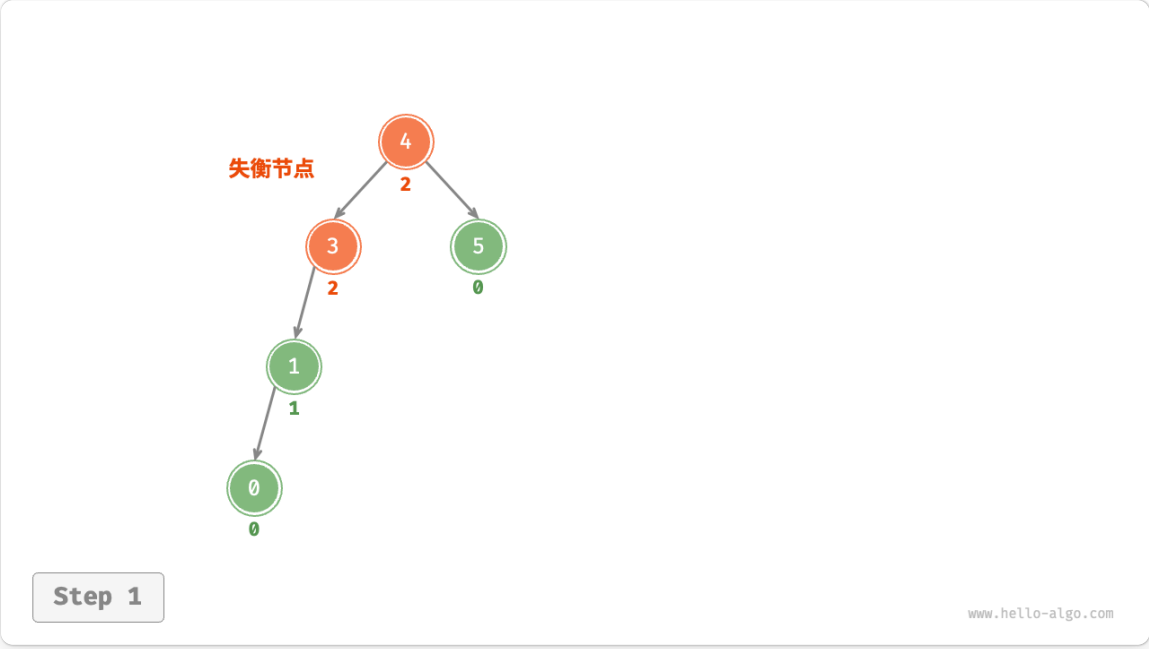

图 7-26 右旋操作步骤

如图 7-27 所示，当节点 `child` 有右子节点（记为 `grand_child` ）时，需要在右旋中添加一步：将 `grand_child` 作为 `node` 的左子节点。

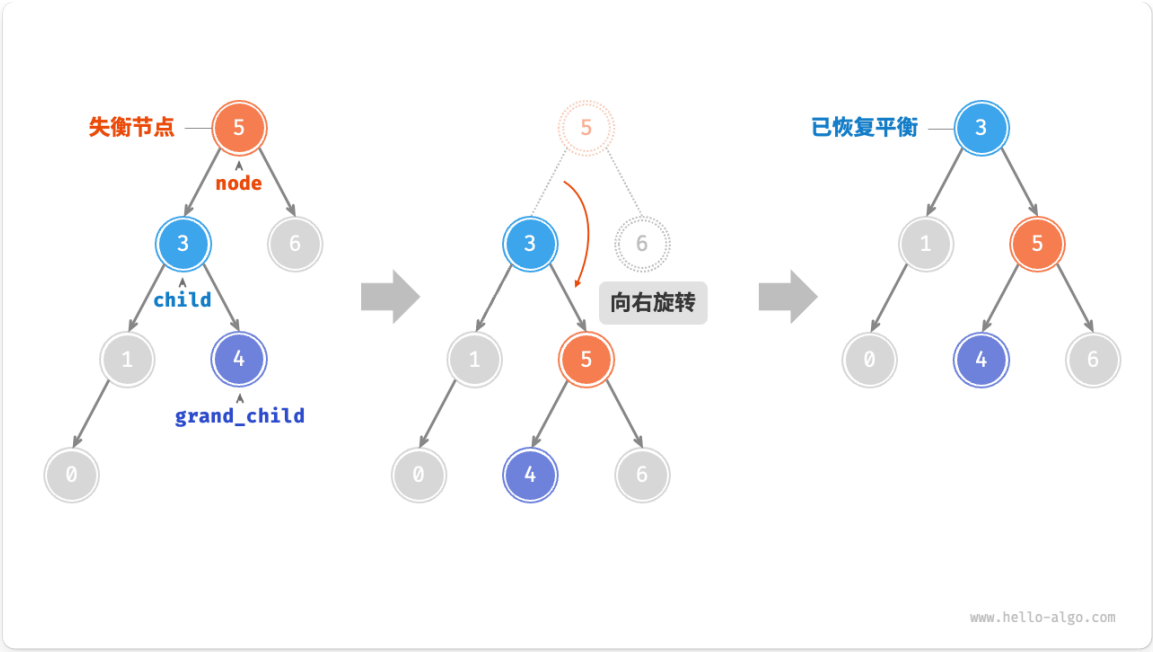

图 7-27 有 grand_child 的右旋操作

“向右旋转”是一种形象化的说法，实际上需要通过修改节点指针来实现，代码如下所示：

Python

avl_tree.py

```
def right_rotate(self, node: TreeNode | None) -> TreeNode | None:
    """右旋操作"""
    child = node.left
    grand_child = child.right
    # 以 child 为原点，将 node 向右旋转
    child.right = node
    node.left = grand_child
    # 更新节点高度
    self.update_height(node)
    self.update_height(child)
    # 返回旋转后子树的根节点
    return child
```

2. 左旋

相应地，如果考虑上述失衡二叉树的“镜像”，则需要执行图 7-28 所示的“左旋”操作。

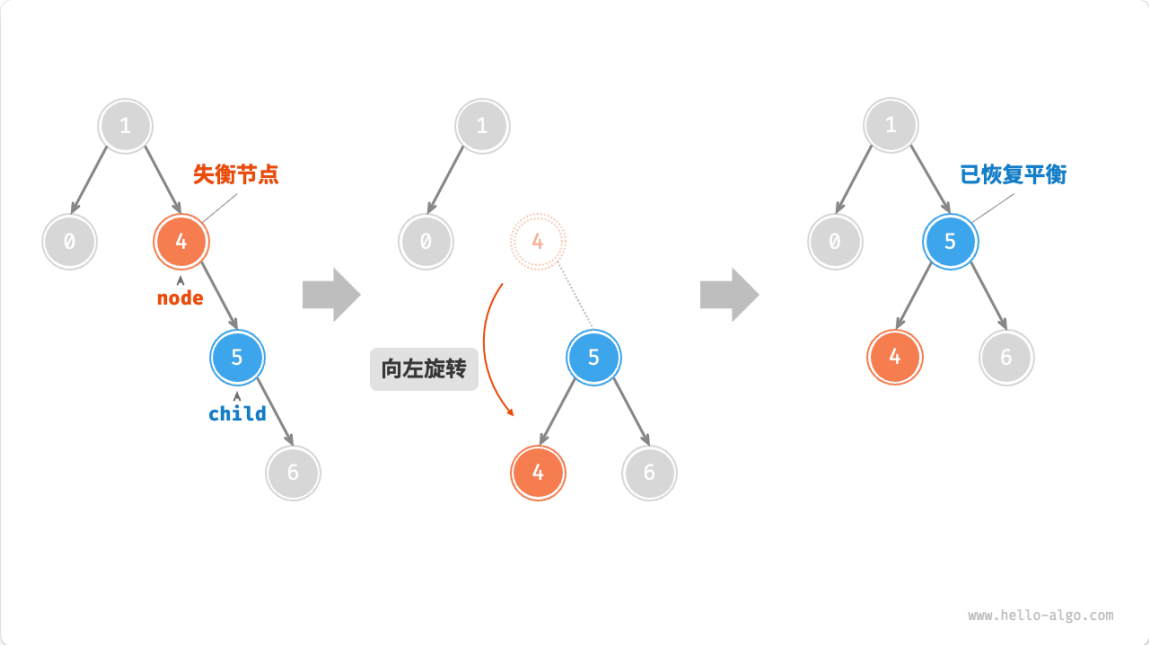

图 7-28 左旋操作

同理，如图 7-29 所示，当节点 `child` 有左子节点（记为 `grand_child` ）时，需要在左旋中添加一步：将 `grand_child` 作为 `node` 的右子节点。

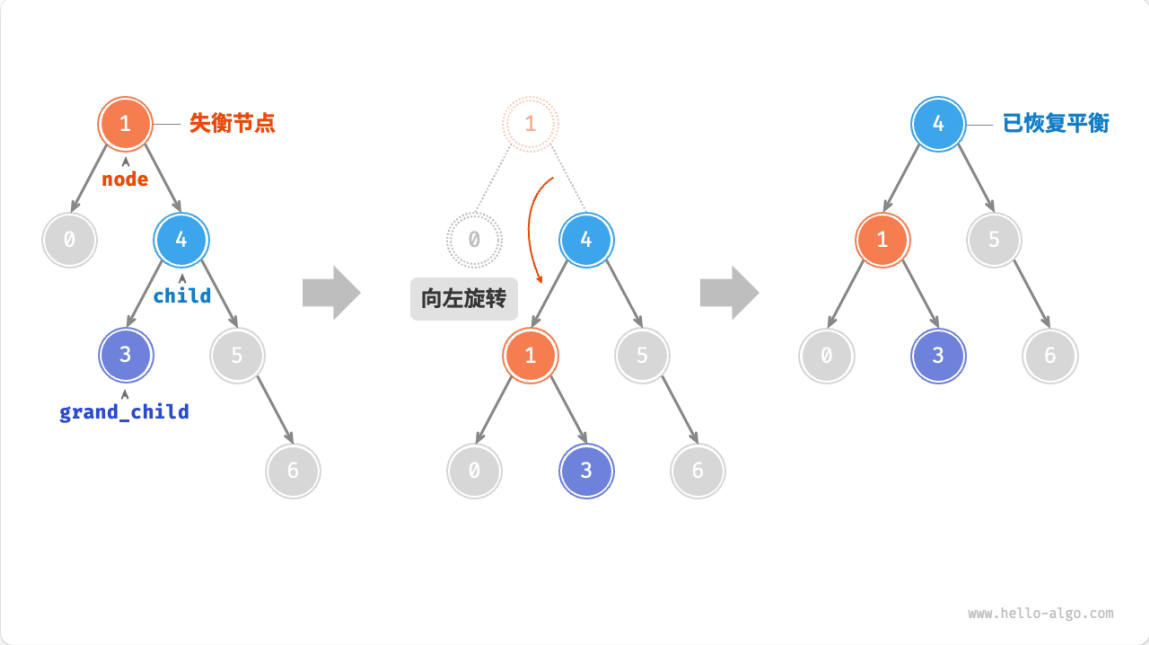

图 7-29 有 grand_child 的左旋操作

可以观察到，**右旋和左旋操作在逻辑上是镜像对称的，它们分别解决的两种失衡情况也是对称的**。基于对称性，我们只需将右旋的实现代码中的所有的 `left` 替换为 `right` ，将所有的 `right` 替换为 `left` ，即可得到左旋的实现代码：

Python

avl_tree.py

```
def left_rotate(self, node: TreeNode | None) -> TreeNode | None:
    """左旋操作"""
    child = node.right
    grand_child = child.left
    # 以 child 为原点，将 node 向左旋转
    child.left = node
    node.right = grand_child
    # 更新节点高度
    self.update_height(node)
    self.update_height(child)
    # 返回旋转后子树的根节点
    return child
```

3. 先左旋后右旋

对于图 7-30 中的失衡节点 3 ，仅使用左旋或右旋都无法使子树恢复平衡。此时需要先对 `child` 执行“左旋”，再对 `node` 执行“右旋”。

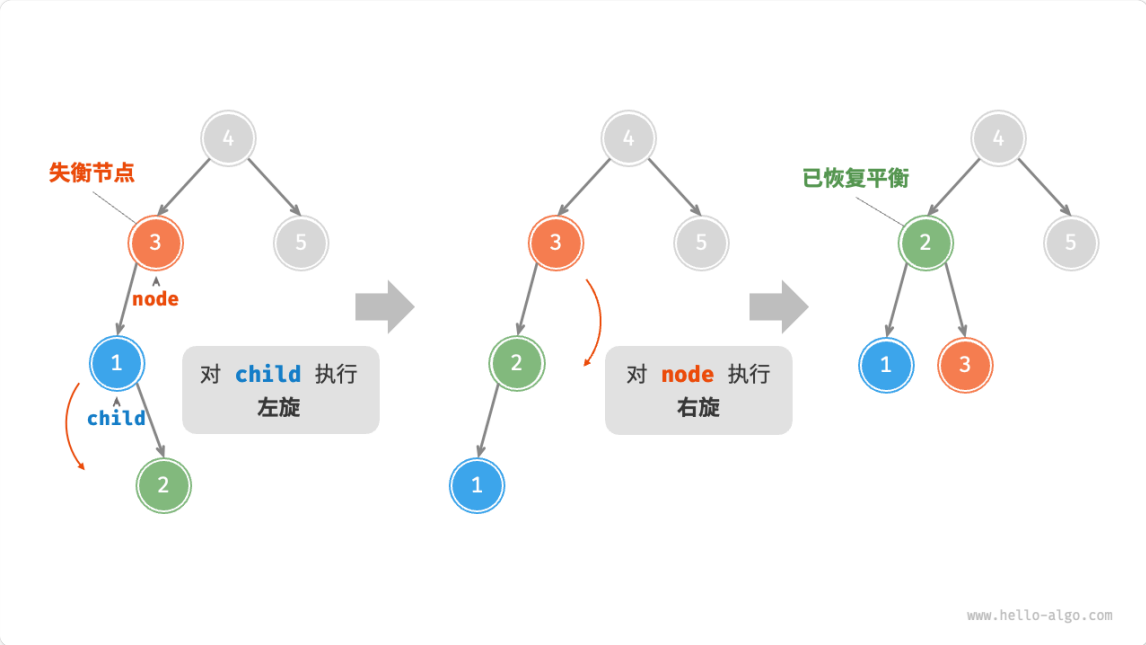

图 7-30 先左旋后右旋

4. 先右旋后左旋

如图 7-31 所示，对于上述失衡二叉树的镜像情况，需要先对 `child` 执行“右旋”，再对 `node` 执行“左旋”。

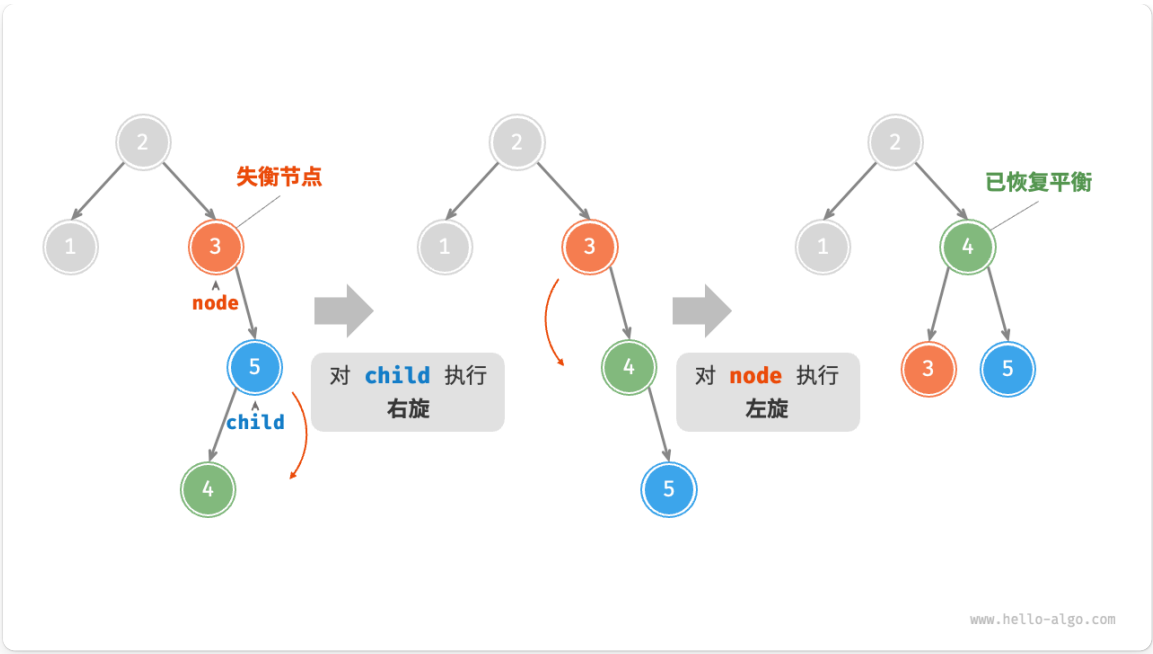

图 7-31　先右旋后左旋

5. 旋转的选择

图 7-32 展示的四种失衡情况与上述案例逐个对应，分别需要采用右旋、先左旋后右旋、先右旋后左旋、左旋的操作。

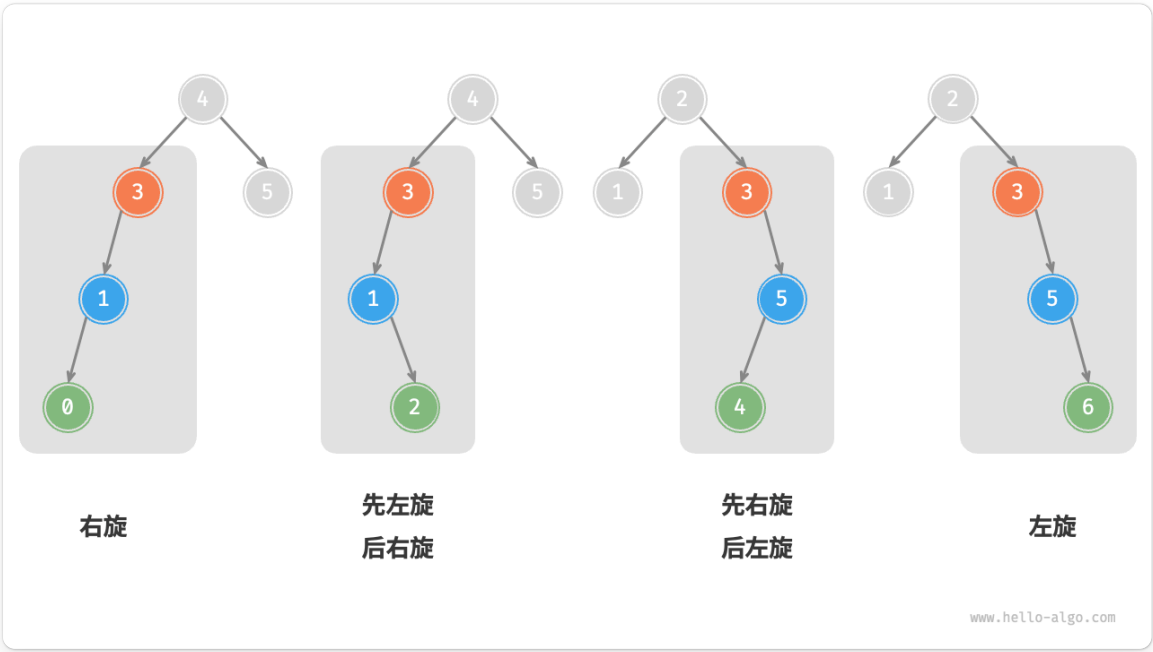

图 7-32　AVL 树的四种旋转情况

如下表所示，我们通过判断失衡节点的平衡因子以及较高一侧子节点的平衡因子的正负号，来确定失衡节点属于图 7-32 中的哪种情况。

表 7-3 四种旋转情况的选择条件

失衡节点的平衡因子	子节点的平衡因子	应采用的旋转方法
> 1 （左偏树）	≥ 0	右旋
> 1 （左偏树）	< 0	先左旋后右旋
< -1 （右偏树）	≤ 0	左旋
< -1 （右偏树）	> 0	先右旋后左旋

为了便于使用，我们将旋转操作封装成一个函数。**有了这个函数，我们就能对各种失衡情况进行旋转，使失衡节点重新恢复平衡**。代码如下所示：

Python

avl_tree.py

```
def rotate(self, node: TreeNode | None) -> TreeNode | None:
    """执行旋转操作，使该子树重新恢复平衡"""
    # 获取节点 node 的平衡因子
    balance_factor = self.balance_factor(node)
    # 左偏树
    if balance_factor > 1:
        if self.balance_factor(node.left) >= 0:
            # 右旋
            return self.right_rotate(node)
        else:
            # 先左旋后右旋
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)
    # 右偏树
    elif balance_factor < -1:
        if self.balance_factor(node.right) <= 0:
            # 左旋
            return self.left_rotate(node)
        else:
            # 先右旋后左旋
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)
    # 平衡树，无须旋转，直接返回
    return node
```

7.5.3 AVL 树常用操作

1. 插入节点

AVL 树的节点插入操作与二叉搜索树在主体上类似。唯一的区别在于，在 AVL 树中插入节点后，从该节点到根节点的路径上可能会出现一系列失衡节点。因此，**我们需要从这个节点开始，自底向上执行旋转操作，使所有失衡节点恢复平衡**。代码如下所示：

Python

avl_tree.py

```python
def insert(self, val):
    """插入节点"""
    self._root = self.insert_helper(self._root, val)

def insert_helper(self, node: TreeNode | None, val: int) -> TreeNode:
    """递归插入节点（辅助方法）"""
    if node is None:
        return TreeNode(val)
    # 1. 查找插入位置并插入节点
    if val < node.val:
        node.left = self.insert_helper(node.left, val)
    elif val > node.val:
        node.right = self.insert_helper(node.right, val)
    else:
        # 重复节点不插入，直接返回
        return node
    # 更新节点高度
    self.update_height(node)
    # 2. 执行旋转操作，使该子树重新恢复平衡
    return self.rotate(node)
```

2. 删除节点

类似地，在二叉搜索树的删除节点方法的基础上，需要从底至顶执行旋转操作，使所有失衡节点恢复平衡。代码如下所示：

Python

avl_tree.py

```python
def remove(self, val: int):
    """删除节点"""
    self._root = self.remove_helper(self._root, val)

def remove_helper(self, node: TreeNode | None, val: int) -> TreeNode |
None:
    """递归删除节点（辅助方法）"""
    if node is None:
        return None
```

```
        # 1. 查找节点并删除
        if val < node.val:
            node.left = self.remove_helper(node.left, val)
        elif val > node.val:
            node.right = self.remove_helper(node.right, val)
        else:
            if node.left is None or node.right is None:
                child = node.left or node.right
                # 子节点数量 = 0 ，直接删除 node 并返回
                if child is None:
                    return None
                # 子节点数量 = 1 ，直接删除 node
                else:
                    node = child
            else:
                # 子节点数量 = 2 ，则将中序遍历的下个节点删除，并用该节点替换当前节点
                temp = node.right
                while temp.left is not None:
                    temp = temp.left
                node.right = self.remove_helper(node.right, temp.val)
                node.val = temp.val
        # 更新节点高度
        self.update_height(node)
        # 2. 执行旋转操作，使该子树重新恢复平衡
        return self.rotate(node)
```

3. 查找节点

AVL 树的节点查找操作与二叉搜索树一致，在此不再赘述。

7.5.4 AVL 树典型应用

- 组织和存储大型数据，适用于高频查找、低频增删的场景。
- 用于构建数据库中的索引系统。
- 红黑树也是一种常见的平衡二叉搜索树。相较于 AVL 树，红黑树的平衡条件更宽松，插入与删除节点所需的旋转操作更少，节点增删操作的平均效率更高。

上一页 ← 7.4 二叉搜索树

下一页 7.6 小结 →

欢迎在评论区留下你的见解、问题或建议