

# 基本的红黑树数据结构

杨子凡

Jul 19, 2025

红黑树作为一种自平衡二叉搜索树，在计算机科学领域具有重要地位。它广泛应用于高性能库中，例如 C++ STL 的 map 和 set，以及 Java 的 TreeMap。这些应用得益于红黑树能保证最坏情况下的  $O(\log n)$  时间复杂度，包括插入、删除和查找操作。本文旨在深入解析红黑树的原理，并结合手写代码实现来阐明其工作机制。同时，我们将对比其他平衡树如 AVL 树，讨论其适用场景差异，帮助开发者在工程选型时做出明智决策。通过理论与实践的结合，本文力求降低理解门槛，确保读者能突破平衡树难点。

## 1 红黑树核心特性

红黑树的核心特性体现在其五大性质上。节点颜色非红即黑；根节点始终为黑；叶子节点（通常使用 NIL 哨兵节点）也为黑；红色节点的子节点必须为黑，这禁止了连续红节点的出现；任意节点到其叶子路径的黑高（即路径上黑节点数量）相同，这是维持平衡的关键。这些性质共同确保红黑树的平衡性。数学推导证明：设最短路径全由黑节点构成，长度为黑高  $bh$ ；最长路径红黑交替，长度不超过  $2bh$ 。因此，树高差不超过  $bh$ ，树高本身在  $bh$  到  $2bh$  之间，保证了最坏情况下的  $O(\log n)$  性能。这种设计以较少的平衡代价换取高效动态操作。

## 2 核心操作：旋转与颜色调整

旋转操作是红黑树调整平衡的基础，包括左旋和右旋，它们在不破坏二叉搜索树性质的前提下调整子树高度。左旋用于降低右子树高度，而右旋则相反。以下以 Python 代码为例，详细解读左旋操作。

```
1 def left_rotate(node):
2     right_child = node.right
3     # 更新子节点关联：将右子节点的左子树移为当前节点的右子树
4     node.right = right_child.left
5     if right_child.left != NIL:
6         right_child.left.parent = node
7     # 更新父节点关联：将右子节点的父节点设为当前节点的父节点
8     right_child.parent = node.parent
9     # 更新根节点或父节点的子节点指向
10    if node.parent == NIL:
11        root = right_child # 如果当前节点是根，更新根
12    elif node == node.parent.left:
13        node.parent.left = right_child
```

```
15     else:
16         node.parent.right = right_child
17         # 完成旋转：将当前节点设为右子节点的左子树
18         right_child.left = node
19         node.parent = right_child
```

这段代码首先保存当前节点的右子节点，然后更新子树关联：如果右子节点有左子树，则将其父指针指向当前节点。接着处理父节点关联：根据当前节点是左子或右子，更新父节点的指向。最后，建立旋转后的父子关系，确保树结构正确。颜色调整策略则用于解决插入或删除后可能出现的连续红节点冲突，通过重新着色和旋转组合来恢复性质。例如，在插入新节点时，如果出现连续红节点，则根据叔节点颜色决定调整方式。

### 3 插入操作详解

插入操作首先遵循标准二叉搜索树规则：将新节点初始化为红色，并插入到适当位置。之后，修复红黑树性质以防止连续红节点。修复过程分情况讨论：如果叔节点为红，则通过重新着色解决，将父节点和叔节点变黑、祖父节点变红；如果叔节点为黑，则需旋转加着色。具体分为 LL 或 RR 型（单旋操作）以及 LR 或 RL 型（双旋操作）。例如，在 LR 型中，先对父节点进行左旋转换为 LL 型，再对祖父节点右旋，最后重新着色。整个过程通过决策流程图确保逻辑完备，新节点的插入总是从底层向上递归修复，确保黑高一致性和颜色规则。

### 4 删除操作详解

删除操作同样基于标准二叉搜索树：分类处理零个、一个或两个子节点的情况。删除后，修复过程重点关注「双重黑」节点的出现（即被删除节点的位置被视为额外黑色）。修复分三种情况：如果兄弟节点为红，则通过旋转（如左旋或右旋）将其转为黑，并重新着色；如果兄弟为黑且其子节点全黑，则重新着色并将双重黑上移至父节点；如果兄弟为黑且存在红子节点，则通过旋转（如单旋或双旋）和着色修复平衡。例如，在兄弟有右红子节点时，对兄弟节点左旋并调整颜色。删除修复同样以流程图形式确保所有路径覆盖，解决双重黑问题后递归向上检查。

### 5 完整代码实现

完整的红黑树实现包括节点结构设计和树类框架。节点结构定义了键值、颜色和子节点指针。

```
class Node:
2     def __init__(self, key, color='R'):
3         self.key = key
4         self.color = color # 'R' 表示红, 'B' 表示黑
5         self.left = self.right = self.parent = NIL # NIL 为哨兵节点
```

这段代码中，每个节点包含键值 key、颜色属性 color（默认为红色），以及指向左子、右子和父节点的指针，初始化为 NIL 哨兵。哨兵节点统一处理边界条件，提高代码健壮性。红黑树类框架则封装核心方法。

```
1 class RedBlackTree:
```

```
def __init__(self):
    self.NIL = Node(None, 'B') # 哨兵节点为黑
    self.root = self.NIL

def insert(self, key):
    # 标准 BST 插入逻辑
    new_node = Node(key, 'R')
    # ... 插入新节点到适当位置
    self._fix_insert(new_node) # 调用修复方法

def _fix_insert(self, node):
    # 插入修复逻辑, 处理连续红节点
    while node.parent.color == 'R':
        # 分情况处理叔节点颜色
        # Case 1: 叔节点为红, 重新着色
        # Case 2 & 3: 叔节点为黑, 旋转加着色
        ...
```

在 `insert` 方法中, 新节点插入后调用 `_fix_insert` 修复。`_fix_insert` 方法通过循环处理父节点为红的情况, 分情况实现着色和旋转。类似地, `delete` 和 `_fix_delete` 方法处理删除后修复。关键点在于修复逻辑的完备性, 例如在 `_fix_delete` 中循环处理双重黑节点, 直到根节点或问题解决。

## 6 正确性验证与测试

为确保红黑树实现的正确性, 需要验证五大性质。可编写递归工具函数检查黑高一致性: 从根节点到每个叶子路径的黑高应相同; 同时扫描是否存在连续红节点违规。测试用例设计包括顺序插入和删除 (模拟最坏情况, 如顺序插入) 以验证旋转逻辑; 随机操作压力测试 (执行大量随机插入和删除) 验证平衡性和时间复杂度。例如, 顺序插入 1000 个元素后, 树高应保持在  $O(\log n)$  范围内。可视化工具如 Graphviz 可生成树结构图辅助调试, 但本文避免图片, 推荐使用日志输出节点关系。测试中需覆盖所有插入和删除的分支情况, 确保代码健壮。

## 7 红黑树 vs. 其他平衡树

红黑树与 AVL 树的对比凸显其工程优势。红黑树在插入和删除操作上更快, 因为它允许更宽松的平衡 (旋转次数较少), 适合频繁修改的场景; 而 AVL 树维护更严格的平衡, 查询操作更快, 适用于读密集型应用。例如, 在 Linux 内核的进程调度器「Completely Fair Scheduler」中, 红黑树用于高效管理任务队列; 在数据库如 MySQL InnoDB 的辅助索引中, 它支持动态数据更新。这种取舍源于红黑树的设计哲学: 以少量平衡代价换取高效动态操作。开发者应根据应用场景 (高更新频率 vs 高查询频率) 选择合适结构。

红黑树的设计哲学在于平衡效率与动态性, 通过五大性质和旋转操作保证最坏情况性能。实现难点集中在删除操作的修复逻辑, 尤其是双重黑节点的处理, 需要完备的分情况讨论。进阶方向包括并发红黑树 (支持多线程操作) 和磁盘存储优化 (如 B+ 树)。通过本文的解析和代码实现, 读者可深入掌握红黑树原理, 并在实际项目中

应用。完整代码可参考 GitHub 仓库，理论基础推荐《算法导论》和 Linux 内核源码 `rbtree.h`。