深入理解并实现基本的红黑树（Red-Black Tree）数据结构

杨子凡

Aug 09, 2025

二叉搜索树（Binary Search Tree）是一种基础数据结构，支持高效的查找、插入和删除操作，时间复杂度在理想情况下为 $O(\log n)$。然而，当插入有序数据时，二叉搜索树可能退化为链表结构，导致时间复杂度恶化至 $O(n)$。例如，依次插入序列 $1, 2, 3, \ldots, n$ 会形成一条单链，完全丧失平衡性。为解决这一问题，平衡二叉搜索树应运而生，它通过约束树的结构来维持近似平衡状态，确保操作效率稳定在 $O(\log n)$。红黑树（Red-Black Tree）作为其中一种经典实现，与 AVL 树形成对比；AVL 树追求严格平衡（高度差不超过 1），适用于读多写少的场景，而红黑树采用近似平衡策略，在插入和删除频繁的环境中更具优势，例如 Linux 内核调度器用于进程管理、Java 的 `TreeMap` 和 `TreeSet` 或 C++ 的 `std::map` 容器，以及文件系统如 Ext4 的索引结构。这些应用场景突显了红黑树在工业实践中的核心价值，即通过较少的平衡调整开销换取高效性能。

1　红黑树核心特性解析

红黑树通过五大规则确保近似平衡性，这些规则共同约束节点的颜色（红色或黑色）和结构。规则 1 规定根节点必须为黑色，这为树的统一性提供基础；规则 2 定义叶子节点（NIL 节点）为黑色，作为路径的边界基准；规则 3 要求红色节点的子节点必为黑色，防止连续红节点出现，从而限制路径长度；规则 4 确保任意路径从根到叶的黑色节点数相同（称为黑高度），这是平衡的关键；规则 5 则设定新插入节点默认为红色，以最小化平衡调整的需求。这些规则的本质在于通过颜色标记实现黑高度平衡，数学推导可证明树高的上限。考虑最短路径（全黑节点）和最长路径（红黑交替），设黑高度为 h_b，则最短路径长度为 h_b，最长路径不超过 $2h_b$（因红色节点不连续）。结合节点总数 n 和黑高度关系，可推导树高上限为 $2\log_2(n+1)$，这确保了红黑树始终近似平衡，时间复杂度稳定在 $O(\log n)$。

2　红黑树操作：旋转与变色

旋转操作是红黑树维持平衡的核心机制，分为左旋和右旋两种对称形式。左旋用于调整右子树过高的场景，以节点 x 为支点，其右子节点 y 成为新父节点，过程包括重定位子树和更新父指针。以下 C++ 伪代码展示左旋实现：

```
void left_rotate(Node* x) {
  Node* y = x->right; // 1. 定位 x 的右子节点 y
  x->right = y->left; // 2. y 的左子树成为 x 的右子树
  if (y->left != nil) y->left->parent = x; // 3. 若 y 的左子存在，更新其父指针
  y->parent = x->parent; // 4. 将 y 的父指针指向 x 的原父节点
  // 后续处理父节点链接（省略部分代码）
}
```

代码解读：步骤 1 获取 x 的右子节点 y；步骤 2 将 y 的左子树转移至 x 的右子树位置，保持二叉搜索树性质；步骤 3 更新子树节点的父指针，确保链接正确；步骤 4 开始处理父节点关系，需根据 x 是否为根节点等情况继续完成。右旋操作与之对称，用于左子树过高的场景。变色操作则涉及颜色翻转（Color Flip），例如在插入调整中，当父节点和叔节点均为红色时，通过将父和叔节点变黑、祖父节点变红来局部恢复平衡，避免旋转开销。

3 红黑树插入：全流程拆解

红黑树插入始于标准二叉搜索树（BST）插入：递归或迭代定位插入位置，将新节点（默认为红色）挂载到叶节点。若插入后破坏红黑树规则（如产生连续红节点），则启动修正流程。修正策略基于叔节点（父节点的兄弟节点）颜色和结构，分为三种核心场景。Case 1 中叔节点为红色，此时将父节点和叔节点变黑，祖父节点变红，然后递归向上检查祖父节点；Case 2 为叔节点黑色且形成三角型结构（如新节点是父节点的右子，而父节点是祖父节点的左子），此时通过左旋将结构转为线性；Case 3 为叔节点黑色且线性结构（如新节点和父节点均为左子），此时右旋祖父节点并交换父节点与祖父节点颜色。例如，插入序列 $[3, 21, 32, 15]$ 时，插入 32 后触发 Case 1 变色，插入 15 后进入 Case 2 旋转调整 Case 3 变色，最终恢复平衡。

4 红黑树删除：复杂场景攻克

删除操作先执行标准 BST 删除：若为叶子节点直接移除；单子节点时用子节点替代；双子节点时用后继节点值替换再删除后继节点。删除后可能破坏规则（如减少黑色节点），引入双重黑色（Double Black）概念 —— 一个虚拟的额外黑色权重，需通过修正消除。修正分四种场景：Case 1 兄弟节点为红色时，旋转父节点使兄弟变黑；Case 2 兄弟黑色且兄弟的子节点全黑时，将兄弟变红，双重黑向上传递至父节点；Case 3 兄弟黑色且兄弟的近侄子（与兄弟同侧的子节点）为红时，旋转兄弟节点并变色，转为 Case 4；Case 4 兄弟黑色且兄弟的远侄子（与兄弟异侧的子节点）为红时，旋转父节点，变色解决双重黑。例如，删除根节点时可能触发 Case 4，删除红色叶节点通常无调整，删除黑色叶节点则需处理双重黑。

5 手把手实现红黑树（代码框架）

实现红黑树需设计节点结构，包含值、颜色标记、父指针和 NIL 哨兵节点。以下 Python 伪代码定义节点类：

```
class Node:
  def __init__(self, val):
    self.val = val # 节点存储的值
    self.color = 'RED' # 新节点默认红色（规则 5）
    self.left = NIL # 左子节点指向 NIL 哨兵
    self.right = NIL # 右子节点指向 NIL 哨兵
    self.parent = NIL # 父指针，初始指向 NIL
```

代码解读：`__init__` 方法初始化节点属性，`color` 字段设置为 `'RED'` 遵循插入规则；`left`、`right` 和 `parent` 均初始指向全局 NIL 节点，简化边界处理。关键方法包括 `insert()` 和 `delete()` 入口函数，它们调用 BST 逻辑后触发 `fix_insertion()` 或 `fix_deletion()` 修正函数；旋转函数如 `_left_rotate` 和 `_right_rotate`

实现前述操作逻辑。辅助工具如层级打印函数可视化树结构，黑高度验证函数递归检查从根到叶的路径是否满足规则 4（黑节点数相同）。

6 红黑树实战：测试与验证

正确性测试需覆盖边界和压力场景。插入有序序列 $1, 2, 3, \ldots, 100$ 后，验证树高不超过 $2\log_2(100+1) \approx 14$，确保未退化；随机执行 10^4 次插入和删除操作，实时检查五大规则（如递归遍历验证无连续红节点）。性能对比实验量化优势：红黑树与普通 BST 在插入有序数据时，前者耗时保持 $O(\log n)$，后者恶化至 $O(n)$；红黑树与 AVL 树在随机插入删除中，因旋转次数较少，红黑树效率更高，尤其写操作频繁时。

7 延伸与进阶

红黑树可优化为无父指针实现，如 Linux 内核通过颜色嵌入指针低位节省内存；延迟删除（Lazy Deletion）策略标记节点而非移除，提升批量操作效率。红黑树与 2-3-4 树存在等价性：红节点表示与父节点融合的 3 节点或 4 节点，黑节点表示独立节点，等价性证明涉及结构转换。跳表（Skip List）作为替代方案，以概率平衡换取简单实现。工业级参考如 JDK `TreeMap` 源码，其 `fixAfterInsertion` 方法处理插入修正，逻辑类似前述 Case 分析。

红黑树的设计哲学在于以少量规则（五大特性）换取高效近似平衡，适用于读写均频繁的场景，如内存数据库索引；相较磁盘优化结构如 B 树，红黑树更适内存操作。学习建议强调动手实现：从基础插入删除编码开始，通过调试和可视化工具逐步验证，最终深入工业源码。掌握红黑树不仅深化数据结构理解，更为高性能系统开发奠基。