

基本的红黑树（Red-Black Tree）数据结构

杨奇瑞

Aug 11, 2025

1 从理论到代码，掌握高效自平衡二叉搜索树的核心

二叉搜索树（BST）在理想情况下能提供 $O(\log n)$ 的查询效率，但在极端场景下会退化成链表结构，导致操作时间复杂度恶化至 $O(n)$ 。这种不平衡性催生了自平衡二叉树的诞生，其中红黑树以其卓越的工程实践价值脱颖而出。与严格平衡的 AVL 树相比，红黑树通过松散的平衡约束减少了旋转操作次数，特别适合写入频繁的场景。工业级应用中，Linux 内核进程调度器、Java 的 HashMap（JDK8+）以及 C++ STL 的 map/set 都采用了红黑树作为核心数据结构。

2 红黑树核心性质

红黑树通过五大约束条件维持近似平衡：

- 每个节点非红即黑；
- 根节点必为黑色；
- 所有叶子节点（NIL）均为黑色；
- 红色节点的子节点必为黑色（无连续红节点）；
- 从根节点到任意叶子节点的路径包含相同数量的黑色节点（黑高一致）。

这些规则衍生出关键推论：任意路径长度不超过最短路径的两倍。设黑高为 h ，树高 H 满足 $h \leq H \leq 2h$ 。这种设计哲学的本质是用颜色规则替代严格平衡，通过容忍一定的不平衡性来减少旋转操作，从而提升写入性能。当插入或删除破坏规则时，系统通过旋转和变色操作恢复平衡，整个过程最多需要 $O(\log n)$ 次调整。

3 红黑树操作：插入与修复

新节点初始设为红色，以最小化对黑高的破坏。插入后可能触发三种修复场景：

- Case 1:** 若插入节点为根，直接染黑即可；
- Case 2:** 父节点为黑时无需处理；
- Case 3:** 父节点为红时需考察叔节点颜色。

当叔节点为红时（Case 3.1），将父节点和叔节点染黑，祖父节点染红，并递归向上修复祖父节点。若叔节点为黑（Case 3.2），则需根据父子关系进行旋转操作。例如在 LL 型结构中（新节点是祖父左子的左子），执行祖父

右旋后交换父节点与祖父节点颜色：

```

1 def _fix_insertion(self, node):
    while node != self.root and node.parent.color == 'RED':
3         if node.parent == node.parent.parent.left: # 父节点是左子
            uncle = node.parent.parent.right
5             if uncle.color == 'RED': # Case 3.1
                node.parent.color = 'BLACK'
7                 uncle.color = 'BLACK'
                node.parent.parent.color = 'RED'
9                 node = node.parent.parent
            else: # Case 3.2
11                if node == node.parent.right: # LR 型
                    node = node.parent
13                    self._rotate_left(node)
                    node.parent.color = 'BLACK'
15                    node.parent.parent.color = 'RED'
                    self._rotate_right(node.parent.parent)
17 self.root.color = 'BLACK' # 确保根节点为黑

```

代码中 `_rotate_left` 和 `_rotate_right` 实现标准二叉树的旋转操作，同时更新父子指针。LR 型场景先通过左旋转换为 LL 型再处理，这种分阶段转换是修复逻辑的核心技巧。

4 红黑树操作：删除与修复

删除操作首先执行标准 BST 删除：若删除叶子节点直接移除；若删除单子节点用子节点替代；若删除双子节点则用后继节点值替代后删除后继节点。当被删节点为黑色时，会引发「双黑」问题，需根据兄弟节点 S 的颜色进行四类修复：

1. **Case 1**: S 为红时，将父节点染红、 S 染黑后旋转父节点，转为后续场景；
2. **Case 2**: S 为黑且其子节点全黑时， S 染红并将双黑上移至父节点；
3. **Case 3**: S 为黑且近侄子为红、远侄子为黑时，旋转 S 并交换颜色转为 Case 4；
4. **Case 4**: S 为黑且远侄子为红时，交换父节点与 S 颜色，远侄子染黑后旋转父节点结束修复。

```

1 def _fix_deletion(self, node):
    while node != self.root and node.color == 'BLACK':
3         if node == node.parent.left:
            sib = node.parent.right
5             if sib.color == 'RED': # Case 1
                sib.color = 'BLACK'
7                 node.parent.color = 'RED'
                self._rotate_left(node.parent)

```

```
9         sib = node.parent.right
10        if sib.left.color == 'BLACK' and sib.right.color == 'BLACK': # Case 2
11            sib.color = 'RED'
12            node = node.parent
13        else:
14            if sib.right.color == 'BLACK': # Case 3
15                sib.left.color = 'BLACK'
16                sib.color = 'RED'
17                self._rotate_right(sib)
18                sib = node.parent.right
19            sib.color = node.parent.color # Case 4
20            node.parent.color = 'BLACK'
21            sib.right.color = 'BLACK'
22            self._rotate_left(node.parent)
23            node = self.root
24        node.color = 'BLACK'
```

删除修复通过兄弟节点的颜色和子节点状态决定操作序列，Case 3 到 Case 4 的转换体现了状态机式的处理逻辑。

5 手把手实现红黑树

节点结构需包含颜色标记和父指针，使用 NIL 哨兵节点统一处理边界条件：

```
class RedBlackTree:
2     class Node:
3         __slots__ = 'val', 'left', 'right', 'parent', 'color'
4         def __init__(self, val, color='RED'):
5             self.val = val
6             self.left = None
7             self.right = None
8             self.parent = None
9             self.color = color # 新节点默认为红色
10
11     def __init__(self):
12         self.NIL = self.Node(None, 'BLACK') # 统一 NIL 哨兵
13         self.root = self.NIL
14
15     def insert(self, val):
16         new_node = self.Node(val)
17         new_node.left = self.NIL # 初始化子节点为 NIL
```

```
18     new_node.right = self.NIL
    # ... 标准 BST 插入逻辑
20     self._fix_insertion(new_node) # 触发修复

22     def _rotate_left(self, x):
        y = x.right
24         x.right = y.left
        if y.left != self.NIL:
26             y.left.parent = x
        y.parent = x.parent
28         if x.parent == self.NIL:
            self.root = y
30         elif x == x.parent.left:
            x.parent.left = y
32         else:
            x.parent.right = y
34         y.left = x
        x.parent = y
```

实现要点在于：旋转操作需同步更新父子关系；插入后从新节点向上修复；删除后从替代节点开始修复；每次操作后需重置根节点为黑色。NIL 节点的统一处理避免了空指针异常，是工程实现的常见技巧。

6 红黑树性能分析

时间复杂度层面，查询操作稳定在 $O(\log n)$ ，由黑高约束 $h \geq \lceil \log_2(n+1) \rceil$ 保证。插入删除同样为 $O(\log n)$ ，且旋转次数上限为 3 次（插入）和 1 次（删除）。

与 AVL 树对比，红黑树在插入删除时旋转更少，但查询稍慢（树高更高）。空间复杂度为 $O(n)$ ，每个节点需额外存储颜色和父指针。

进阶话题与扩展

左倾红黑树（LLRB）通过强制左倾属性简化实现，可视为 2-3-4 树的二叉树投影。并发场景中，读多写少时可使用读写锁优化。调试时需递归验证五大性质，特别要检查所有路径黑高是否一致。常见陷阱包括：未正确处理 NIL 节点颜色（必须为黑）、旋转后忘记更新父指针、删除后未重置根节点颜色。可视化工具如 Graphviz 能生成树结构图辅助验证。

红黑树的设计精髓在于用颜色规则换取高效平衡，通过精心设计的旋转与变色策略维持 $O(\log n)$ 的操作复杂度。

- ↪ 作复杂度。它特别适合高频写入的关联容器场景，如数据库索引和内存缓存。学习红黑树不仅能掌握经
- ↪ 典数据结构，更能深入理解复杂系统设计中的权衡艺术 (Trade-off) —— 在理论完美性与工程实用
- ↪ 性之间寻找最佳平衡点。