

17 | 红黑 (R-B) 树：和平衡二叉树有什么不同？

2023-03-22 王健伟 来自北京

《快速上手C++数据结构与算法》



你好，我是王健伟。

上次我和你分享了“平衡二叉树”这个话题，引出了“平衡性调整”的概念。主要目的是让这棵二叉树左右看起来比较“平衡”，不出现左子树很高、右子树很矮，或者左子树很矮、右子树很高的情形。这样，在进行节点的查找、插入、删除等操作时效率会比较高。

但这同时也带来了缺点——在插入和删除节点时，为了调整平衡，必须对树中的节点进行旋转，从而在一定程度上影响程序的执行效率。

平衡二叉树的一条重要性质是“任一节点的左子树和右子树的高度之差不超过 1”。这里引入一个“**非严格平衡二叉树**”的概念。

非严格平衡二叉树指的是**不完全符合**前面平衡二叉树的定义，或者说并不是一种严格意义上的平衡二叉树。但这种二叉树最小高度仍旧在 \log_2^n (n 代表节点数) 附近，或者说，查找操作

的时间复杂度仍旧为 $O(\log_2^n)$ 。所以，我们仍旧可以认为这种二叉树是一种平衡二叉树。这就引出了这节课要讲解的“红黑树”。

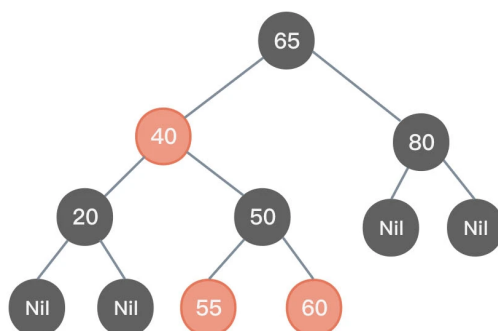
首先一起看一看红黑树的基本概念。

什么是红黑树？

红黑树 (Red-Black Tree)，简称 R-B 树，是一种高效的二叉查找树，由 Rudolf Bayer 在 1972 年发明的，当时被称为“对称 / 平衡二叉 B 树”，后来在 1978 年由 Leo J. Guibas 和 Robert Sedgwick 修改为“红黑树”。

红黑树首先是一棵**二叉查找树**，也是一种典型的非严格平衡二叉树，或者你也可以理解成一种**特殊 / 特化**的 AVL 树，甚至在很多资料中，**提到平衡二叉树指的就是红黑树**。红黑树在**插入和删除**节点时，会通过特定的操作保持二叉查找树的相对平衡，从而获得**比较高的查找效率**。既然是相对平衡，所以任一节点的左子树和右子树的高度之差很可能会超过 1。

观察图 1。



极客时间

shikey.com 转载分享 图1 一棵红黑树

红黑树的节点是有颜色的，颜色分两种——一种是红色，一种是黑色。红黑树有 4 个性质（特点），实现红黑树代码的时候一定要遵循它们，我们一起来看一下。

性质 1：每个节点要么是黑色，要么是红色，而**根节点必须是黑色**的。向后面看，就可以知道，根节点是黑色，其子节点就很灵活——可以是黑色，也可以是红色。

性质 2：每个**叶子节点都是黑色**的空节点（Nil），这一点和其他树不同。也就是说，叶子节点不存储数据。所以，往往绘制红黑树时，**叶子节点会省略不绘制**。如果看到一棵树的底端出现的节点是红色（1 中的节点 55 和节点 60），则这个节点肯定**不是叶子节点**，只是下面的叶子节点省略没有绘制而已。

性质 3：任何相邻的节点，也就是用**线条连接起来的节点**，比如当前节点的父或子节点，**不能同时为红色**（红色节点不能相邻），如果换句话说，可以有下面几种说法，理解任何一种都可以。

1. 红色节点是被黑色节点分隔开的。
2. 红色节点的两个孩子节点或者父亲节点必须是黑色的。
3. 从叶节点到根的所有路径上不可以有两个连续的红色节点。

性质 4：红黑树左右子树的高度差很可能会超过 1 甚至超过更多。红黑树的平衡性指的是对于每个节点，从该节点到达其所能够到达的**叶子节点**的所有路径都包含相同数量的**黑色**节点。这个黑色节点数量叫做“**黑高度**”或者“**黑高**”，用于保证黑色节点的平衡性。

举个例子。

从节点 65 要访问到节点 20 左下叶子节点，需要经过 $65 \rightarrow 40 \rightarrow 20 \rightarrow \text{Nil}$ ，这其中显然包括了 65、20、Nil 这三个黑色节点。

从节点 65 要访问到节点 80 的左下叶子节点，需要经过 $65 \rightarrow 80 \rightarrow \text{Nil}$ ，这三个节点显然都是黑色节点。

从节点 65 要访问到节点 55 的左下叶子节点（图中没有画出但仍旧是存在的），需要经过 $65 \rightarrow 40 \rightarrow 50 \rightarrow 55 \rightarrow \text{Nil}$ ，这其中显然包括了 65、50、Nil 这三个黑色节点。

显然，图 2 的节点组合情形在红黑树中都不应该存在，其中左侧的上下共 2 棵红黑树违背了红黑树的性质 3，右侧的上下共 4 棵红黑树违背了红黑树的性质 4。

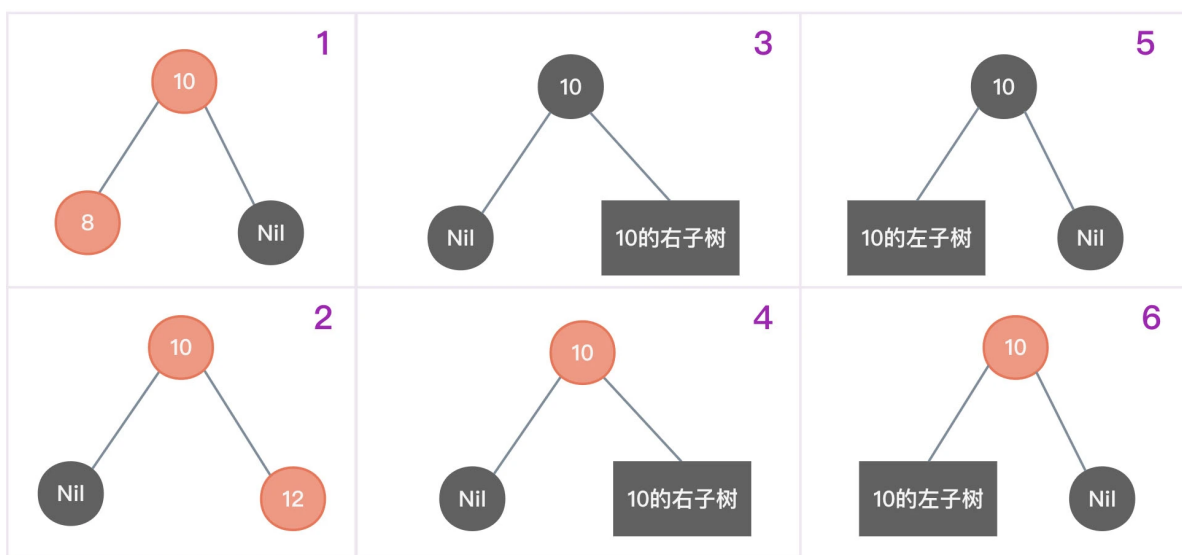


图2 几种红黑树中不可能出现的情形

这里我也把图 2 违背性质的地方列了出来。

编号为 1 的二叉树红色节点 10 和红色节点 8 相邻，违背（红黑树）性质 3。

编号为 2 的二叉树红色节点 10 和红色节点 12 相邻，违背性质 3。

编号为 3 的二叉树黑色节点 10 无左子节点却有黑色的右子节点，违背性质 4。

编号为 4 的二叉树红色节点 10 无左子节点却有黑色的右子节点，违背性质 4。

编号为 5 的二叉树黑色节点 10 有黑色的左子节点却无右子节点，违背性质 4。

编号为 6 的二叉树红色节点 10 有黑色的左子节点却无右子节点，违背性质 4。

总结一下就是，红黑树是：

shike.com 转载分享

1. 黑色的根；
2. 黑色的叶子；
3. 红色节点不相邻，也就是红色节点的父亲和孩子都是黑色节点；
4. 当前节点到其所有叶节点包含的黑色节点数量相同。

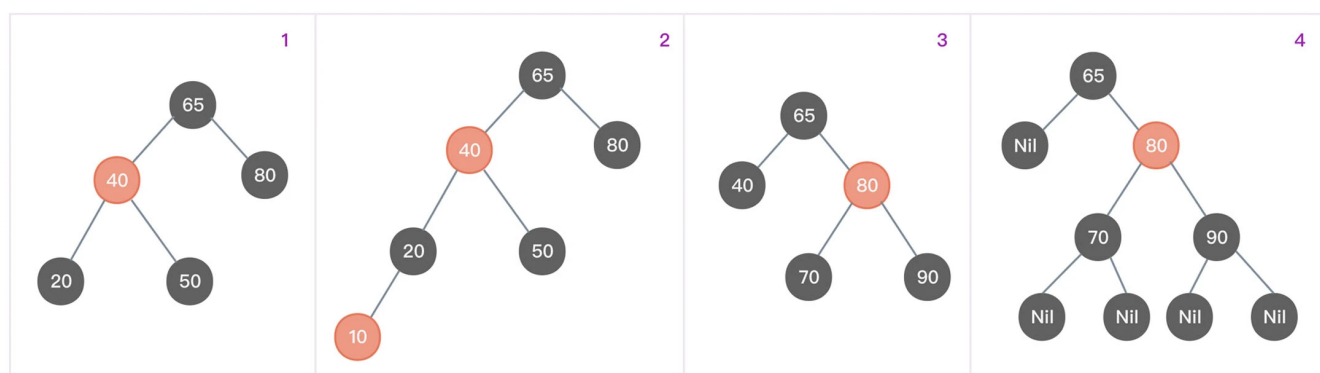
如果你正在思考为什么已经有 AVL 树了还需要引入红黑树，那么简单来说可以这样理解。

如果 AVL 树**很大**，那么在**插入和删除**操作时会进行大量的旋转操作以达到 AVL 树的平衡，尤其是删除节点时，可能要经过若干次的旋转操作，甚至可能需要从最底部的节点一直到根节点都进行平衡性调整。

但在红黑树中，当进行插入和删除操作时，维护红黑树的平衡性成本就比较低——多数情况下只需要旋转两次或者不需要旋转只需要对节点的颜色进行修改，也就是把红色修改为黑色。

所以，虽然 AVL 树比红黑树更加平衡，但针对**插入和删除**操作，红黑树可以保证平均情况时间复杂度更接近 $O(\log_2^n)$ 。换句话说，如果从单纯**搜索**角度来讲，**AVL 树更快**，但如果**频繁进行插入和删除**操作，因为**红黑树**需要更少的旋转，无疑**效率会更高**，而且红黑树的查找、插入、删除操作性能较稳定。

再进一步看一看红黑树左右子树的高度问题以及如何区分是不是一棵合法红黑树的问题，如下图 3。



shikey.com转载分享

极客时间

图3 红黑树的观察左右子树高度观察

图 3 中第 1 个图是一棵红黑树，如果希望为节点 20 增加一个新的左孩子节点 10，节点 10 必须是红色以保证红黑树的性质 4（ $65 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow \text{Nil}$ 包含 3 个黑色节点，到所有其他叶子节点也都包含 3 个黑色节点），参见图 3 第 2 个图，也是一棵红黑树。

如果再为节点 10 增加一个新的左孩子节点，那么这个新节点无论是红色还是黑色，得到的新树都不会再是一棵红黑树，因为如果增加的节点是红色，则不满足红黑树性质 3，如果增加的节点是黑色，则不满足红黑树性质 4。

这意味着，对于图 3 第 2 个图这棵红黑树，从根节点（65）到叶子节点 10 的高度为 4，不可以比 4 再大，否则就不再是一棵红黑树。所以数字 4 在图 3 第 2 个图这棵红黑树中被称为从根节点到叶节点的最长路径（红黑节点间隔存在）。

而显然，从根节点 65 到叶子节点 80 的高度为 2，数字 2 被称为从根节点到叶节点的最短路径（黑节点紧挨在一起）。

因此，不难得到一个结论：红黑树从根节点到叶子的最长路径不超过最短路径的两倍。在红黑树中因为红色节点不能相邻，所以，必须用黑色节点将红色节点隔开。

这也就意味着红黑树中最短路径（最小高度）是不会超过 \log_2^n （二叉树性质五），那么加入红色节点后，最长路径也就不会超过 $2\log_2^n$ ，这意味着红黑树的高度大概为 $2\log_2^n$ 。所以红黑树高度只比 AVL 树高一倍而已，在执行效率上差不太多，而且往往因为上面提到的其他方面的优势，红黑树效率更高，性能更好。

显然，图 3 中第 3 个图也是一棵红黑树。但是如果把这个图中的节点 40 去掉，那么就变成了一棵只有右子树的二叉查找树，这棵二叉查找树就不再是一棵红黑树了，因为不满足红黑树性质 4，很多人看不出来把节点 40 去掉后为什么就不是一棵红黑树了，此时只需要在图中把黑色的 Nil 节点画出来，就很容易看出是不是一棵红黑树了，如图 3 第 4 个图，显然不满足红黑树性质 4。


在具体针对红黑树进行编程时，有几点值得一提。

往往不需要考虑叶子节点，也就是那个黑色的空节点（Nil）不需要考虑。

红黑树的节点查找操作同二叉查找树的节点查找操作（SearchElem 成员函数）完全一样，你可以参考前面已经实现的代码。

红黑树的基础实现代码

下面是红黑树的基础实现代码。

 复制代码

```
1 //红黑树中每个节点的定义
2 template <typename T>    //T代表数据元素的类型
3 struct RBNode
4 {
5     T        data;
6     RBNode* leftChild,    //左子节点指针
7     * rightChild,    //右子节点指针
8     * parentNd;        //父节点指针, 引入方便操作
9
10    bool        isRed;        //是否是红色节点, true: 是, false: 不是 (而是黑色节点)
11 };
12
13 //红黑树的定义
14 template <typename T>
15 class RBTree
16 {
17 public:
18     RBTree()    //构造函数
19     {
20         root = nullptr;
21     }
22     ~RBTree()    //析构函数
23     {
24         ReleaseNode(root);
25     }
26 private:
27     void ReleaseNode(RBNode<T>* pnode)
28     {
29         if (pnode != nullptr)
30         {
31             ReleaseNode(pnode->leftChild);
32             ReleaseNode(pnode->rightChild);
33         }
34         delete pnode;
35     }
36
37 private:
38     RBNode<T>* root;    //树根指针
39 };
```

小结

这节课我带你学习了红黑树，这个用途最广并且面试中最常出现的一种二叉树。

红黑树的基础概念，长什么模样，我们就不再多说。在频繁进行插入和删除操作的场合，红黑树比平衡二叉树具有更高的效率。这也是已经存在了平衡二叉树，但还要引入红黑树的原因。

依据红黑树的几条性质，我们可以判断红黑树的合法性，具体来说有三点。

根节点必须是黑色。

任何相邻的节点不能同时为红色。

对于每个节点，从该节点到达其所能够到达的叶子节点的所有路径都包含相同数量的黑色节点。

这节课，我也给出了红黑树的基础实现代码。有了这些基础实现代码，就可以将研究的重点放在插入数据和删除数据上了。红黑树插入和删除数据所面临的主要问题是平衡性调整问题，这既是重点也是难点，需要好好学习。

下节课，我们就来看看红黑树插入操作后的平衡性调整以及它的实现代码。

课后思考

你以往是否使用过红黑树，用它做过哪方面的工作？你是否还知道更多的关于红黑树的应用场合，能否举一些例子？当然，也可以通过搜索引擎来寻找答案。

欢迎你在留言区分享自己的经验，如果觉得有所收获，也可以把课程分享给更多的朋友一起学习，我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

shikey.com转载分享