红黑树 RB-Tree

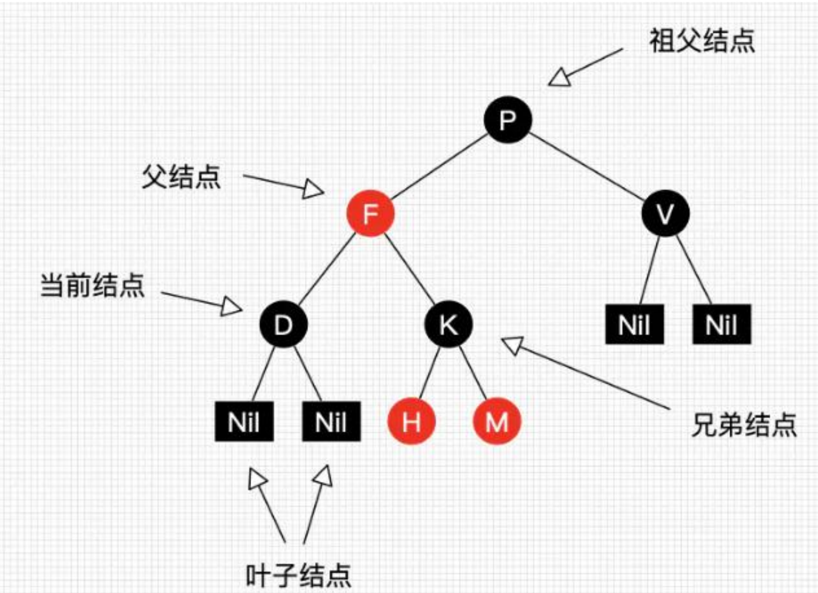
1. 红黑树定义和性质

红黑树是一种含有红黑节点并能自平衡的二叉查找树。它必须满足下面性质：

- 性质 1：每个节点要么是黑色，要么是红色。
- 性质 2：根节点是黑色。
- 性质 3：每个叶子节点（NIL）是黑色。
- 性质 4：每个红色节点的两个子节点一定都是黑色。
- **性质 5：任意一节点到每个叶子节点的路径都包含数量相同的黑节点。**

红黑树并不是一个完美平衡二叉查找树，不过任意一个节点到到每个叶子节点的路径都包含数量相同的黑节点(性质 5)。所以我们叫红黑树这种平衡为黑色完美平衡。

为了后面讲解不至于混淆，我们还需要来约定下红黑树一些节点的叫法，如图 2 所示。



红黑树能自平衡，它靠的是什么？三种操作：左旋、右旋和变色。

1) **左旋**：以某个节点作为支点(旋转节点)，其右子节点变为旋转节点的父节点，右子节点的左子节点变为旋转节点的右子节点，左子节点保持不变。
2) **右旋**：以某个节点作为支点(旋转节点)，其左子节点变为旋转节点的父节点，左子节点的右子节点变为旋转节点的左子节点，右子节点保持不变。
3) **变色**：节点的颜色由红变黑或由黑变红。

上面所说的旋转节点也即旋转的支点，图 3 和图 4 中的 P 节点。我们先忽略颜色，可以看到旋转操作不会影响旋转节点的父节点，父节点以上的节构还是保持不变的。

左旋只影响旋转节点和其右子树的节构，把右子树的节点往左子树挪了。

右旋只影响旋转节点和其左子树的节构，把左子树的节点往右子树挪了。

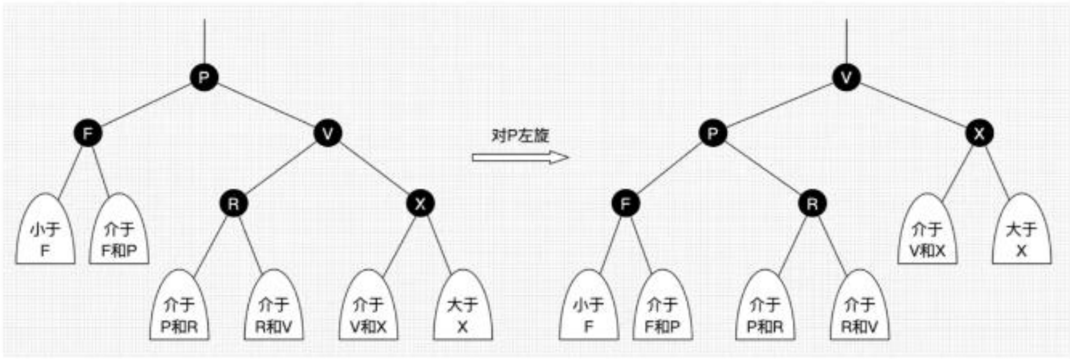


图3 左旋

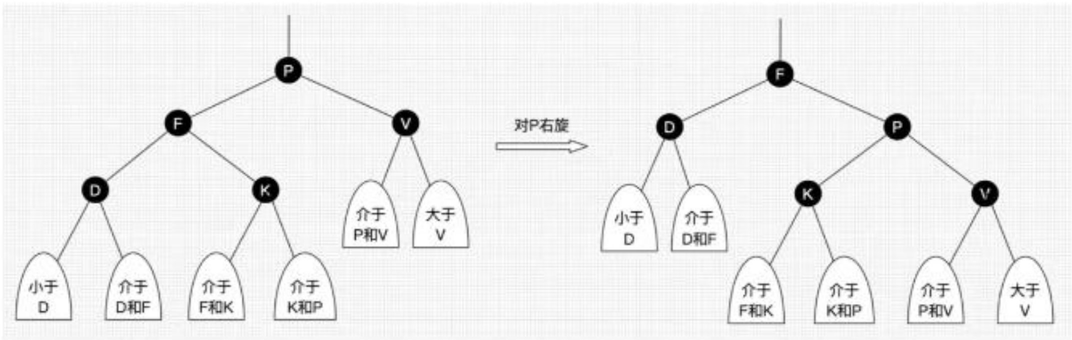


图4 右旋

2. 红黑树查找

因为红黑树是一颗二叉平衡树，并且查找不会破坏树的平衡，所以查找跟二叉平衡树的查找无异：

1) 从根节点开始查找，把根节点设置为当前节点；
2) 若当前节点为空，返回 null；
3) 若当前节点不为空，用当前节点的 key 跟查找 key 作比较；
4) 若当前节点 key 等于查找 key，那么该 key 就是查找目标，返回当前节点；
5) 若当前节点 key 大于查找 key，把当前节点的左子节点设置为当前节点，重复步骤 2；
6) 若当前节点 key 小于查找 key，把当前节点的右子节点设置为当前节点，重复步骤 2；

3. 红黑树插入

3.1. 查找插入的位置

插入操作包括两部分工作：一查找插入的位置；二插入后自平衡。查找插入的父节点很简单，跟查找操作区别不大：

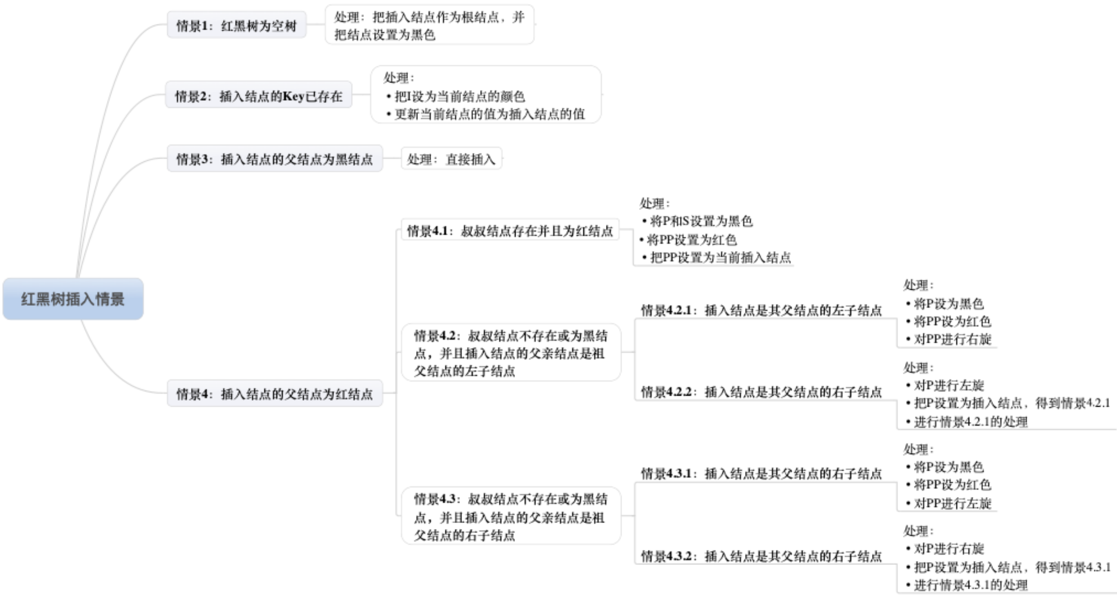
1) 从根节点开始查找；
2) 若根节点为空，那么插入节点作为根节点，节束。
3) 若根节点不为空，那么把根节点作为当前节点；

4) 若当前节点为 null，返回当前节点的父节点，节束。
5) 若当前节点 key 等于查找 key，那么该 key 所在节点就是插入节点，更新节点的值，节束。
6) 若当前节点 key 大于查找 key，把当前节点的左子节点设置为当前节点，重复步骤 4;
7) 若当前节点 key 小于查找 key，把当前节点的右子节点设置为当前节点，重复步骤 4;

3.2. 插入后自平衡

插入位置已经找到，把插入节点放到正确的位置就可以了，但插入节点是应该是什么颜色呢？答案是红色。理由很简单，红色在父节点（如果存在）为黑色节点时，红黑树的黑色平衡没被破坏，不需要做自平衡操作。但如果插入节点是黑色，那么插入位置所在的子树黑色节点总是多 1，必须做自平衡。

所有插入情景如图所示：



插入情景 1：红黑树为空树

最简单的一种情景，直接把插入节点作为根节点就行，但注意，根据红黑树性质 2：根节点是黑色。还需要把插入节点设为黑色。

处理：把插入节点作为根节点，并把节点设置为黑色。

插入情景 2：插入节点的 Key 已存在

插入节点的 Key 已存在，既然红黑树总保持平衡，在插入前红黑树已经是平衡的，那么把插入节点设置为将要替代节点的颜色，再把节点的值更新就完成插入。

处理：（1）把 I 设为当前节点的颜色；（2）更新当前节点的值为插入节点的值。

插入情景 3：插入节点的父节点为黑节点

由于插入的节点是红色的，并不会影响红黑树的平衡，直接插入即可，无需做自平衡。

处理：直接插入。

插入情景 4：插入节点的父节点为红节点

再次回想下红黑树的性质 2：根节点是黑色。**如果插入的父节点为红节点，那么该父节点不可能为根节点，所以插入节点总是存在祖父节点**。这点很重要，因为后续的旋转操作肯定需要祖父节点的参与。

情景 4 又分为很多子情景：

插入情景 4.1：叔叔节点存在并且为红节点

根据红黑树性质 4（每个红色节点的两个子节点一定都是黑色），祖父节点肯定为黑节点，因为不可以同时存在两个相连的红节点。那么此时该插入子树的红黑层数的情况是：黑红红。显然最简单的处理方式是把其改为：红黑红。如图所示。

处理：

（1）将 P 和 S 设置为黑色

（2）将 PP 设置为红色

（3）把 PP 设置为当前插入节点

可以看到，我们把 PP 节点设为红色了，如果 PP 的父节点是黑色，那么无需再做任何处理；但如果 PP 的父节点是红色，根据性质 4，此时红黑树已不平衡了，所以还需要把 PP 当作新的插入节点，继续做插入操作自平衡处理，直到平衡为止。

试想下 PP 刚好为根节点时，那么根据性质 2，我们必须把 PP 重新设为黑色，那么树的红黑节构变为：黑黑红。换句话说，从根节点到叶子节点的路径中，黑色节点增加了。这也是唯一一种会增加红黑树黑色节点层数的插入情景。

插入情景 4.2：叔叔节点不存在或为黑节点，并且插入节点的父亲节点是祖父节点的左子节点

单纯从插入前来看，也即不算情景 4.1 自底向上处理时的情况，叔叔节点非红即为叶子节点(Nil)。因为如果叔叔节点为黑节点，而父节点为红节点，那么叔叔节点所在的子树的黑色节点就比父节点所在子树的多了，这不满足红黑树的性质 5。后续情景同样如此，不再多做说明了。

前文说了，需要旋转操作时，肯定一边子树的节点多了或少了，需要租或借给另一边。插入显然是多的情况，那么把多的节点租给另一边子树就可以了。

插入情景 4.2.1：插入节点是其父节点的左子节点

处理：

（1）将 **P** 设为黑色
（2）将 **PP** 设为红色
（3）对 **PP** 进行右旋

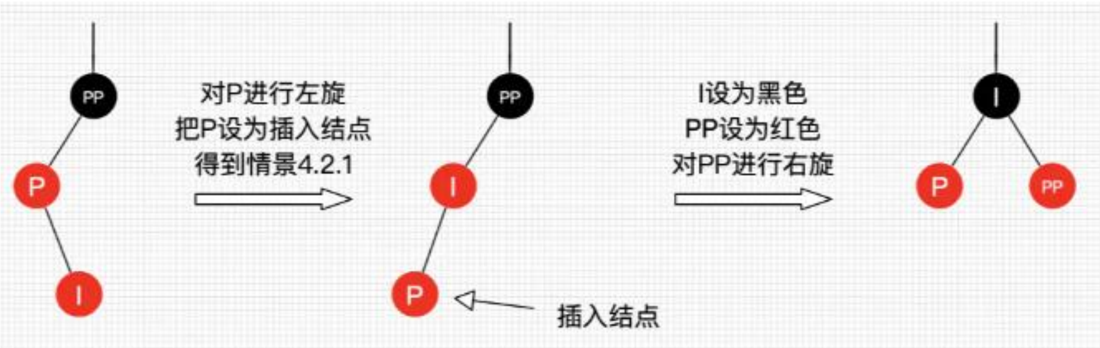
由上图可得，左边两个红节点，右边不存在，那么一边一个刚刚好，并且因为为红色，肯定不会破坏树的平衡。

插入情景 4.2.2：插入节点是其父节点的右子节点

这种情景显然可以转换为情景 4.2.1，如下图所示，不做过多说明了。

处理：

（1）对 **P** 进行左旋
（2）把 **P** 设置为插入节点，得到情景 **4.2.1**
（3）进行情景 **4.2.1** 的处理



插入情景 4.3：叔叔节点不存在或为黑节点，并且插入节点的父亲节点是祖父节点的右子节点

该情景对应情景 4.2，只是方向反转，不做过多说明了，直接看图。

插入情景 4.3.1：插入节点是其父节点的右子节点

处理：

（1）将 P 设为黑色
（2）将 PP 设为**红色**
（3）对 PP 进行左旋

插入情景 4.3.2：插入节点是其父节点的左子节点

处理：

（1）对 P 进行右旋

（2）把 P 设置为插入节点，得到情景 4.3.1

（3）进行情景 4.3.1 的处理

4. 红黑树删除

红黑树的删除操作也包括两部分工作：一查找目标节点；二删除后自平衡。查找目标节点显然可以复用查找操作，当不存在目标节点时，忽略本次操作；当存在目标节点时，删除后就得做自平衡处理了。删除了节点后我们还需要找节点来替代删除节点的位置，不然子树跟父辈节点断开了，除非删除节点刚好没子节点，那么就不需要替代。

二叉树删除节点找替代节点有 3 种情情景：

1) 情景 1：若删除节点无子节点，直接删除
2) 情景 2：若删除节点只有一个子节点，用子节点替换删除节点
3) 情景 3：若删除节点有两个子节点，用后继节点（大于删除节点的最小节点）替换删除节点

补充说明下，情景 3 的后继节点是大于删除节点的最小节点，也是删除节点的右子树种最左节点。那么可以拿前继节点（删除节点的左子树最右节点）替代吗？可以的。但习惯上大多都是拿后继节点来替代，后文的讲解也是用后继节点来替代。

接下来，讲一个重要的思路：**删除结点被替代后，在不考虑结点的键值的情况下，对于树来说，可以认为删除的是替代结点！**如下图，在不看键值对的情况下，图中的红黑树最终结果是删除了 Q 所在位置的结点！

基于此，上面所说的 3 种二叉树的删除情景可以相互转换并且最终都是转换为情景 1！

- 情景 2：删除结点用其唯一的子结点替换，子结点替换为删除结点后，可以认为删除的是子结点，若子结点又有两个子结点，那么相当于转换为情景 3，一直自顶向下转换，总是能转换为情景 1。（对于红黑树来说，根据性质 5.1，只存在一个子结点的结点肯定在树末了）
- 情景 3：删除结点用后继结点（肯定不存在左结点），如果后继结点有右子结点，那么相当于转换为情景 2，否则转为为情景 1。

所有删除的场景：

跟插入操作一样，存在左右对称的情景，只是方向变了，没有本质区别。同样的，我们还是来约定下，如图所示：