数据结构与算法

Data Structure and Algorithm

极夜酱

目录

0.1 红黑树

0.1.1 红黑树（Red Black Tree）

红黑树是一种自平衡的二叉查找树，除了符合二叉查找树的基本特性外，它还具有如下附加特性：

1. 结点是红色或黑色的。
2. 根结点是黑色的。
3. 叶子结点都是黑色的空结点 NIL。
4. 红色结点的两个子结点都是黑色的，即从叶子到根的所有路径上不能有连续的两个红色结点。
5. 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。

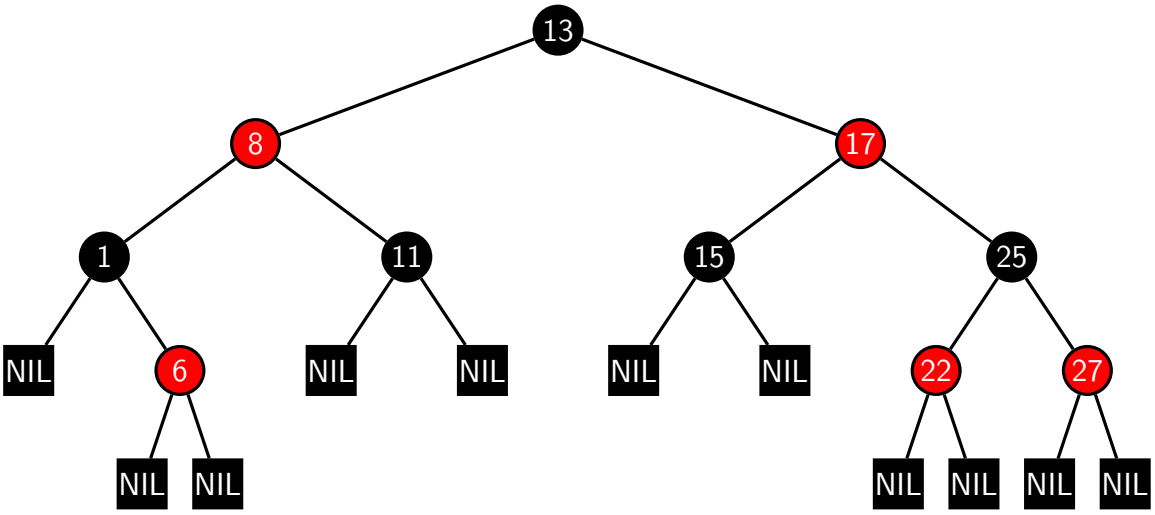


图 1: 红黑树

天呐，这条条框框的太多了吧！

正是因为这些规则限制，才保证了红黑树的自平衡，红黑树从根到叶子的最长路径不会超过最短路径的 2 倍。

红黑树的应用有很多，其中 JDK 的集合类 TreeMap 和 TreeSet 底层就是红黑树实现的。在 Java8 中，连 HashMap 也用到了红黑树。

0.1.2 失衡调整

当插入或删除结点时，红黑树的规则可能被破坏，需要调整使其重新符合规则。

例如向红黑树中插入新结点 14，由于父结点 15 是黑色结点，这种情况不会破坏红黑树的规则，无需做任何调整。

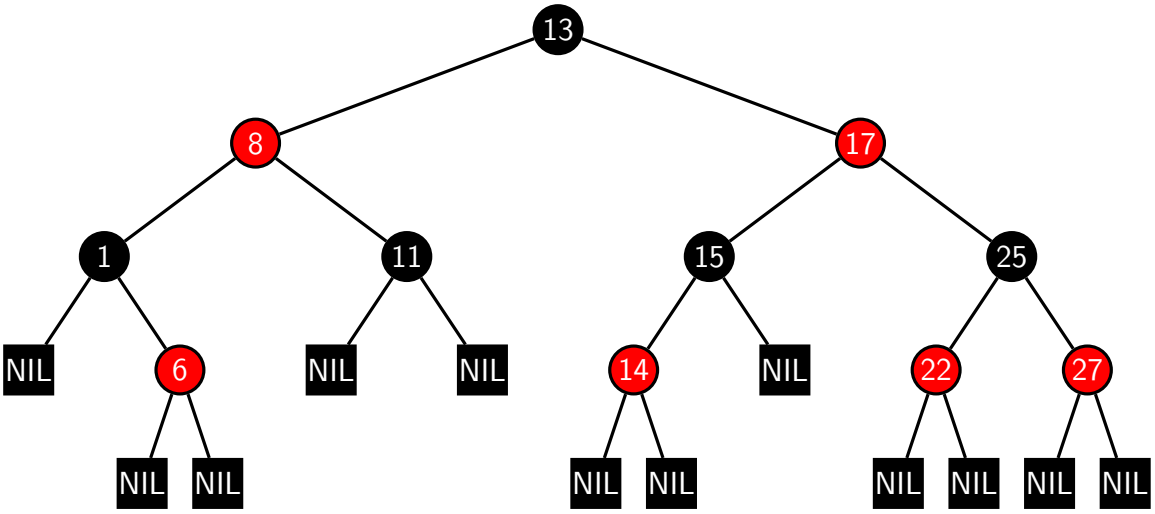


图 2: 插入 14

向红黑树中插入新结点 21，由于父结点 22 是红色结点，违反了红黑树的规则 4（红色结点的两个子结点都是黑色的）。

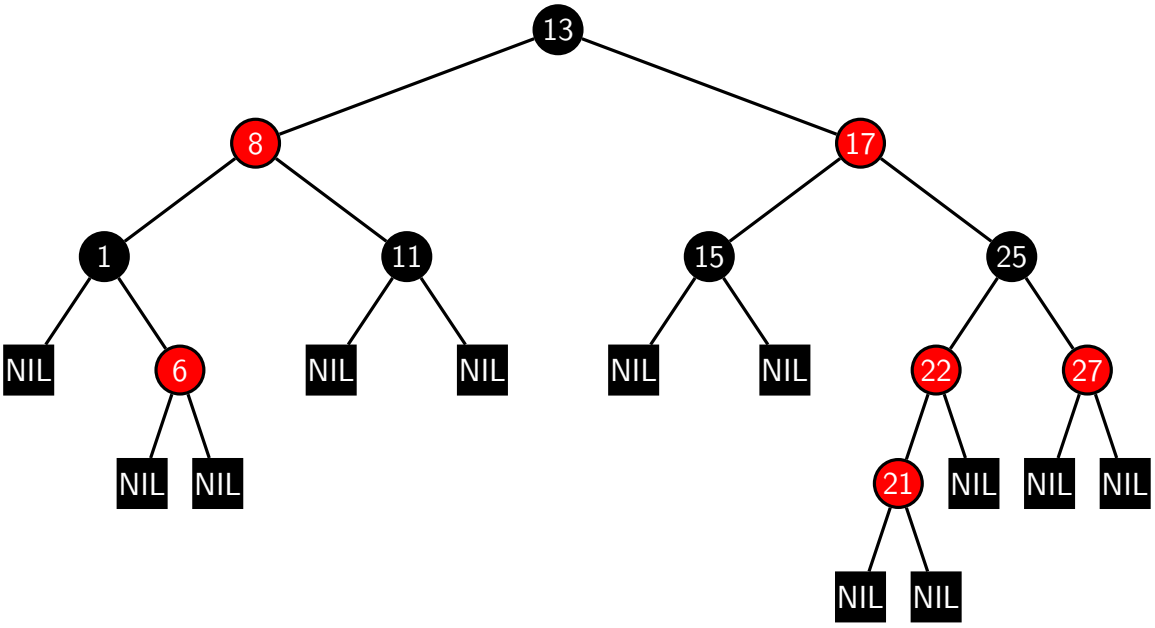


图 3: 插入 21

调整的方法有变色和旋转两种，而旋转又包含左旋转和右旋转两种方式。

为了重新符合红黑树的规则，有时需要把红色结点变为黑色，或是把黑色结点变为红色。

例如对于红黑树的一部分（子树），新插入的结点 Y 是红色结点，它的父结点 X 也是红色结点，不符合规则 4（红色结点的两个子结点都是黑色的），因此可以把结点 X 变为黑色。

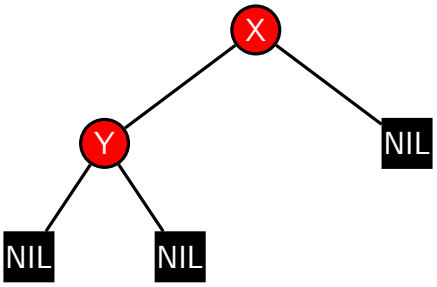


图 4: 违反规则 4

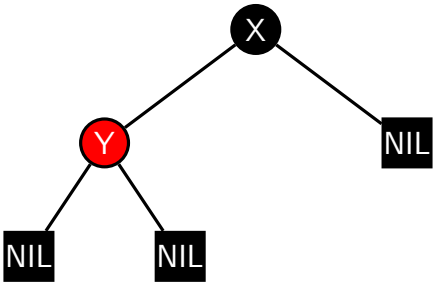


图 5: 变色

但是，如果这是简单的把一个结点变色，会导致相关路径凭空多出一个黑色结点，这样就会打破规则 5（从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点），因此还需要其它的调整策略。

0.1.3 红黑树插入结点

红黑树插入新结点时，可以分为五种不同的局面。每一种局面有不同的调整方法。

局面 1

新结点（A）位于树根，没有父结点。

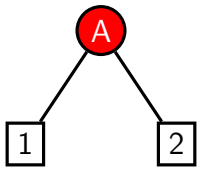
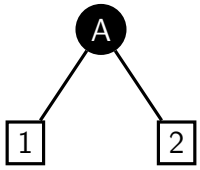


图 6: 局面 1

这种局面，直接让新结点变色为黑色，规则 2（根结点是黑色的）满足。同时黑色的根结点使每条路径上的黑色结点数目都增加了 1，因此并没有打破规则 5（从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点）。



局面 2

新结点（B）的父结点是黑色的。新插入的红色结点 B 并没有打破规则，无需调整。

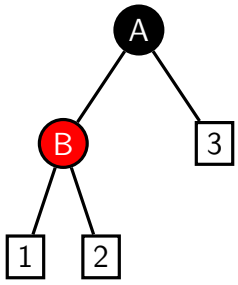


图 7: 局面 2

局面 3

新结点（D）的父结点和叔叔结点都是红色。

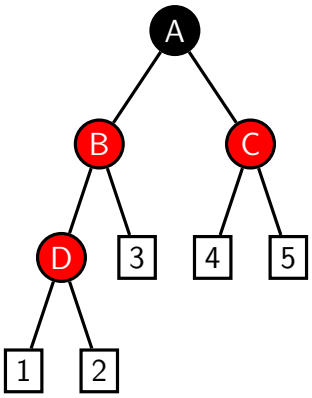
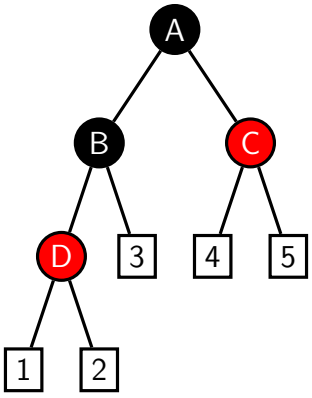
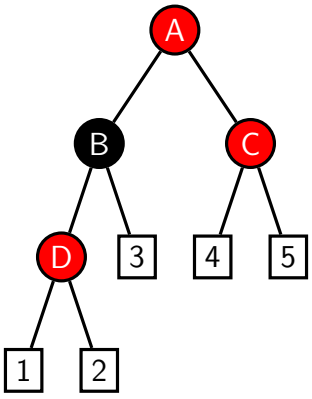


图 8: 局面 3

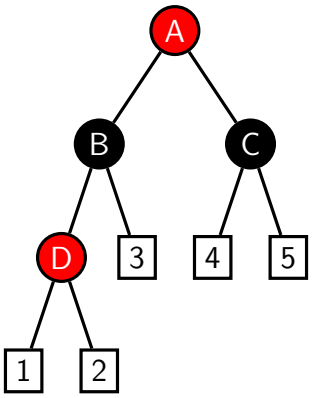
这种局面，两个红色结点 B 和 D 连续，违反了规则 4（红色结点的两个子结点都是黑色的），因此需要先让结点 B 变为黑色。



但是这样一来，结点 B 所在路径凭空多出了一个黑色结点，打破了规则 5（从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点），因此再让结点 A 变为红色。



这时结点 A 和 C 又成为了连续的红色结点，再将结点 C 变为黑色。



局面 4

新结点（D）的父结点是红色，叔叔结点是黑色或者没有叔叔，且新结点是父结点的右孩子，父结点是祖父结点的左孩子。

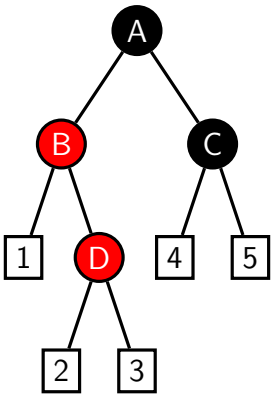
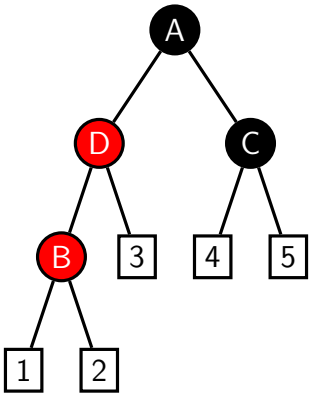


图 9: 局面 4

这个局面可以以结点 B 为轴，做一次左旋转，使得新结点 D 成为父结点，结点 B 成为 D 的左孩子。这样一来进入了局面 5。



局面 5

新结点（D）的父结点是红色，叔叔结点是黑色或者没有叔叔，且新结点是父结点的左孩子，父结点是祖父结点的左孩子。

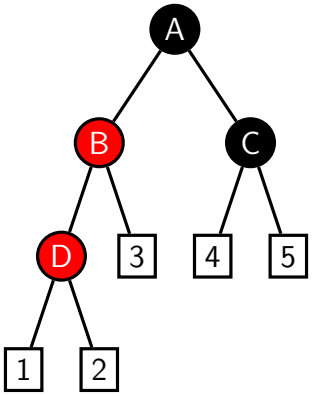
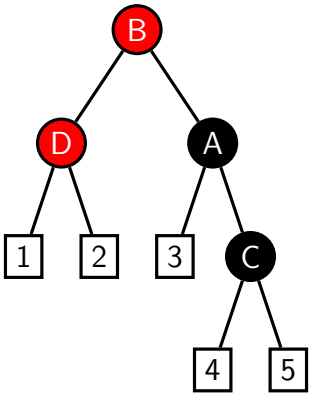
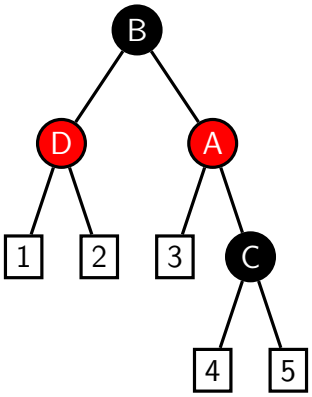


图 10: 局面 5

这一局面可以以结点 A 为轴，做一次右旋转，使得结点 B 成为祖父结点，结点 A 成为 B 的右孩子。

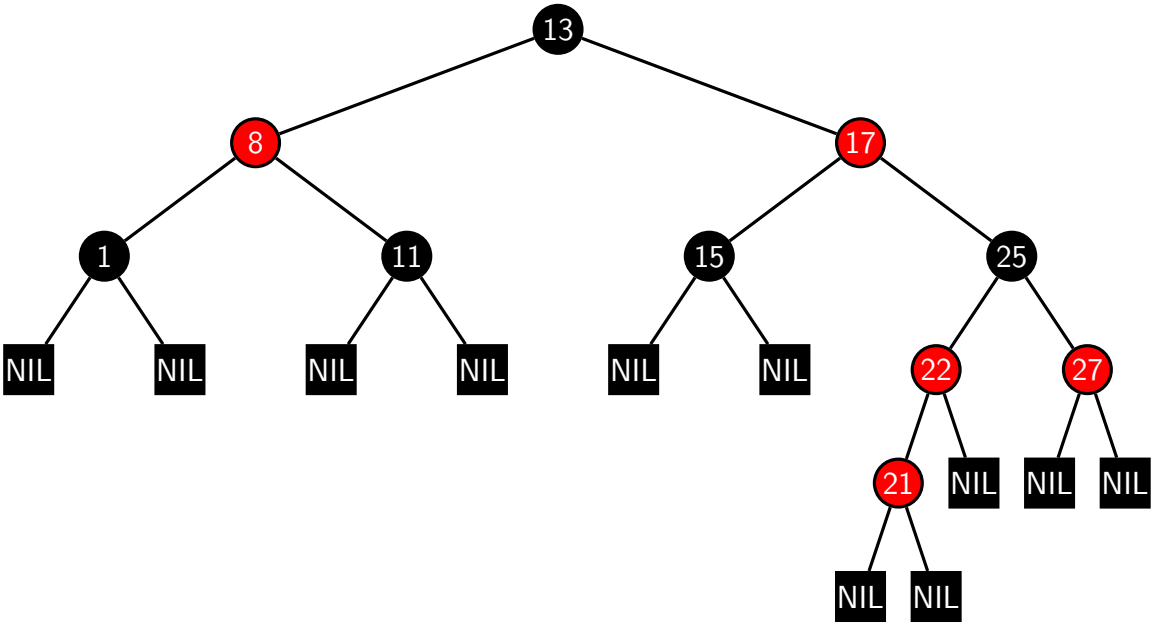


再将结点 B 变为黑色，结点 A 变为红色。

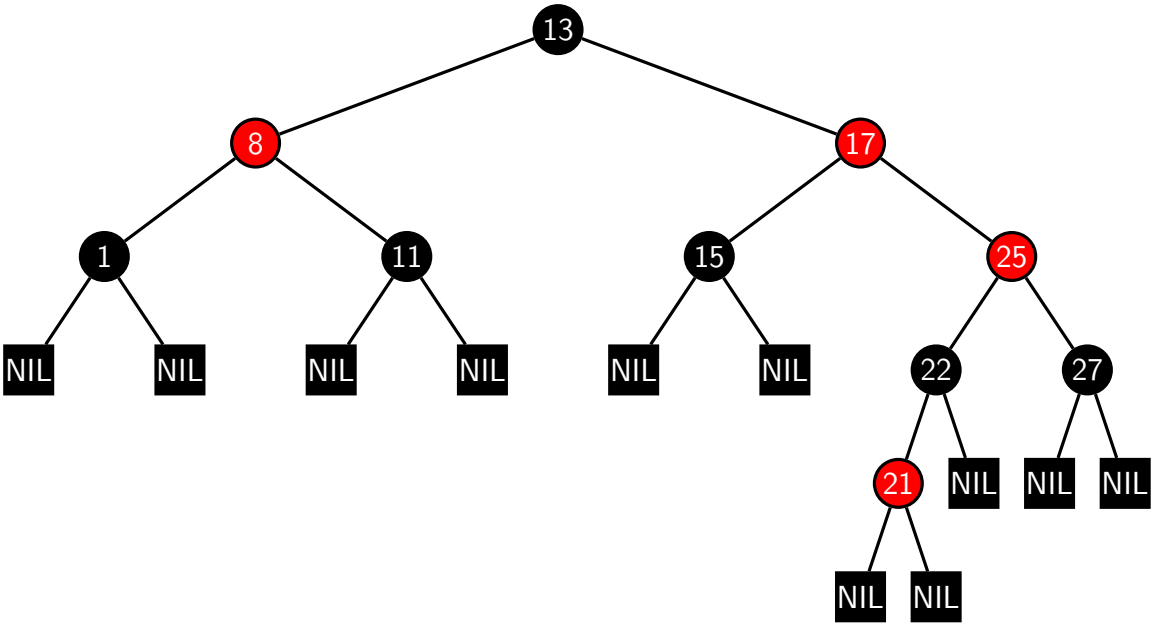


红黑树的插入操作设计到这 5 种局面。如果局面 4 中父结点 B 是右孩子，则成为了局面 5 的镜像，原本的右旋转改为左旋转；如果局面 5 中父结点 B 是右孩子，则成为了局面 4 的镜像，原本的左旋转改为右旋转。

例如在一个红黑树中插入新结点 21，需要根据不同局面进行调整。

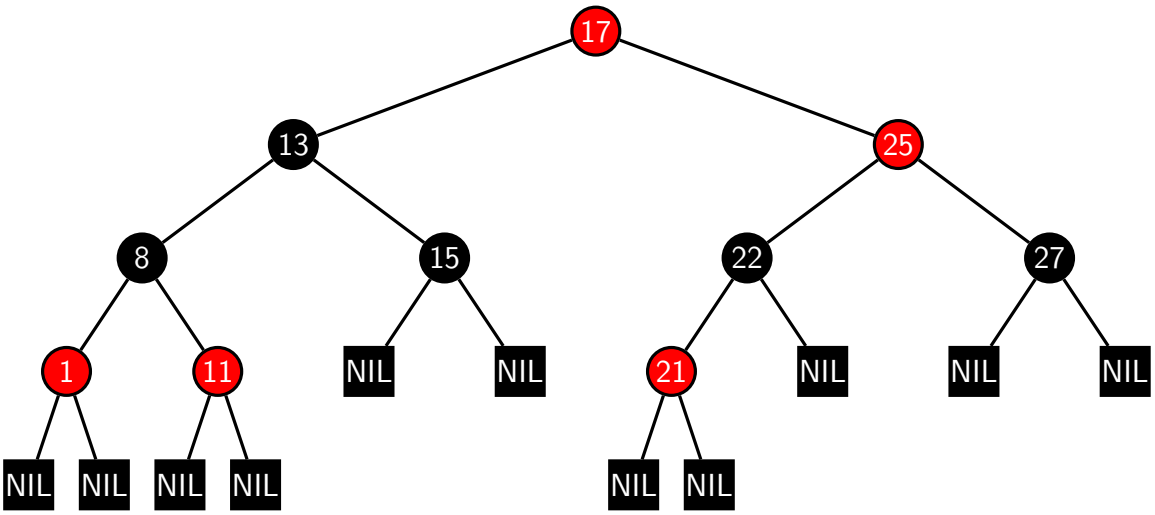


新结点 21 和它的父结点 22 是连续的红色结点，违背了规则 4。当前情况符合局面 3（新结点的父结点和叔叔结点都是红色）。于是经过三次变色（22 变为黑色，25 变为红色，27 变为黑色），将以结点 25 为根的子树符合了红黑树的规则。

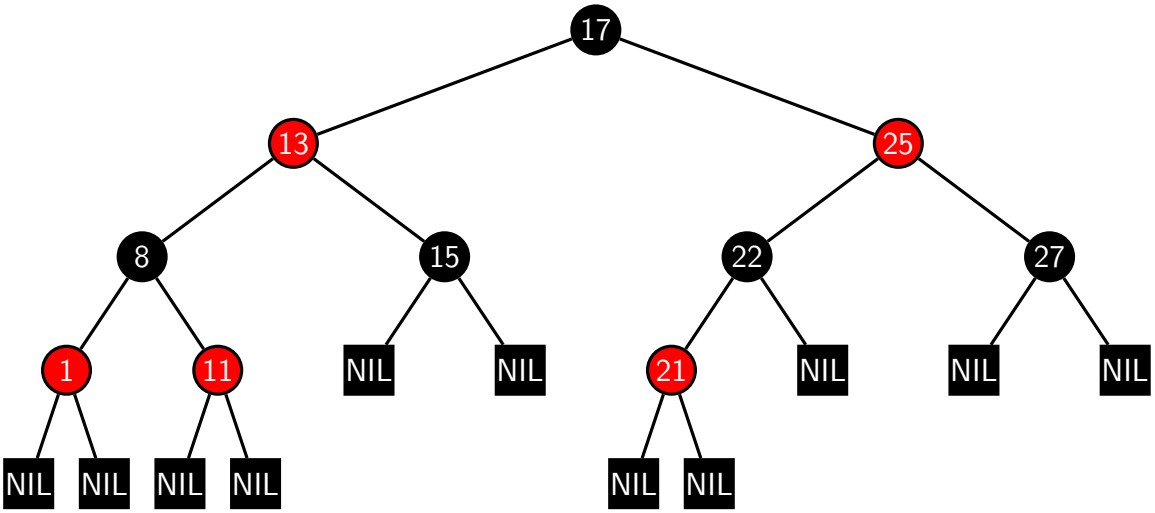


但结点 25 和结点 17 成为了连续的红色结点，违背了规则 4。于是可以将结点 25

看做一个新结点，当前正好符合局面 5 的镜像（新结点的父结点是红色，叔叔是黑色或者没有叔叔，且新结点是父结点的右孩子，父结点是祖父结点的右孩子）。因此可以以根结点 13 为轴进行左旋转，使得结点 17 成为新的根结点。



再让结点 17 变为黑色，13 变为红色，使红黑树重新符合规则。



0.1.4 二叉查找树删除结点

在介绍红黑树的删除操作之前，需要先理解二叉查找树的删除操作。

二叉查找树的删除可分为三种情况：

待删除结点无子结点

如果待删除结点没有子结点，直接删除即可。

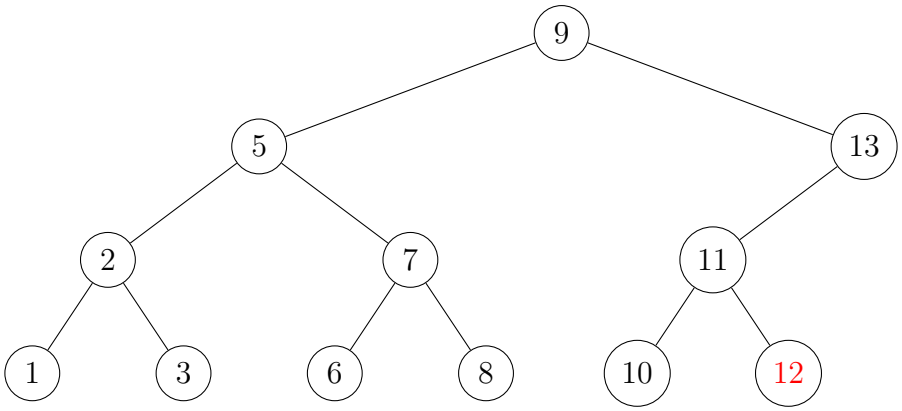


图 11: 删除结点 12

待删除结点只有一个孩子

如果待删除结点只有一个孩子，让孩子取代待被删除结点。

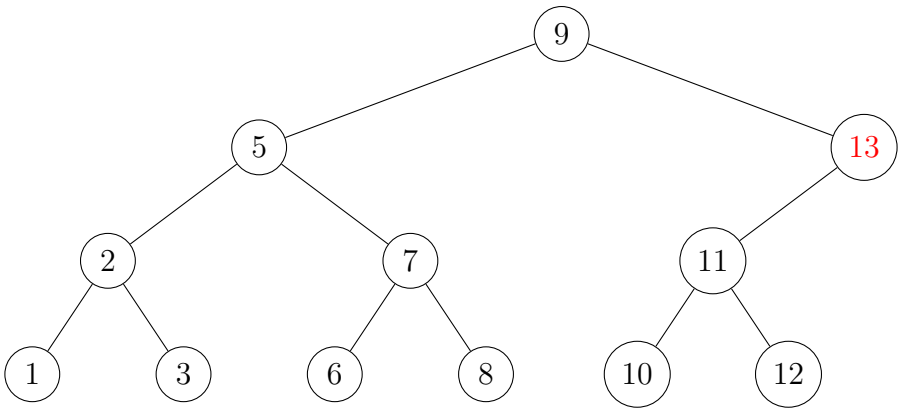
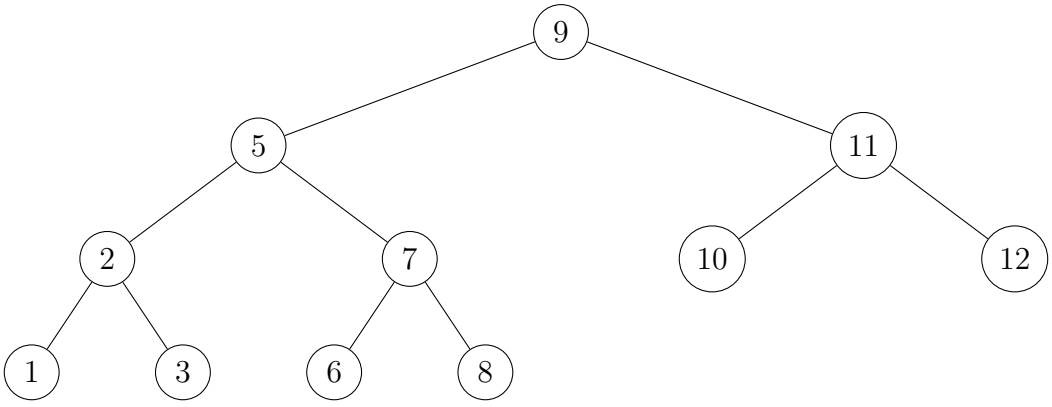


图 12: 删除结点 13



待删除结点有两个孩子

选择仅小于或仅大于待删除结点的结点取代，习惯上更多地会选择仅大于待删除结点的结点。

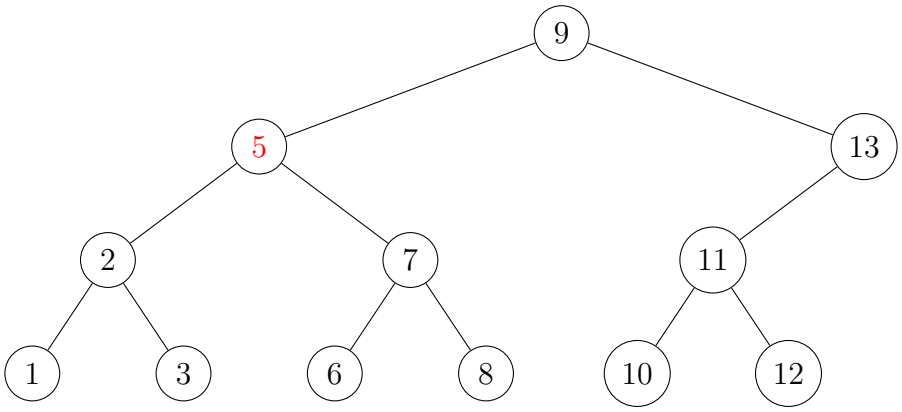
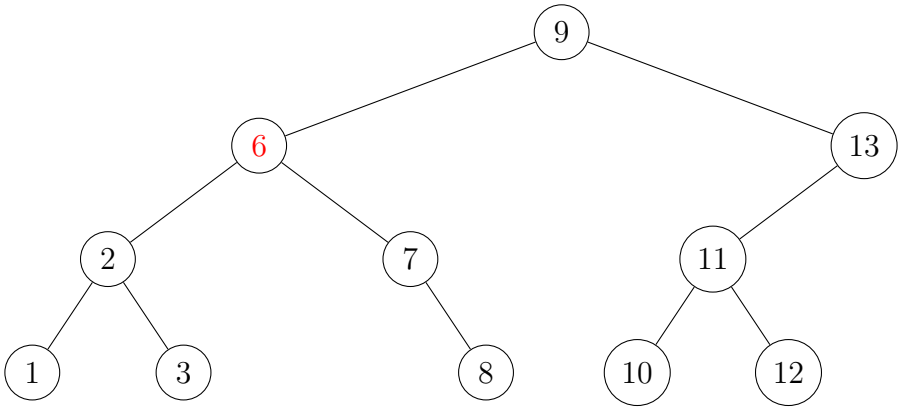


图 13: 删除结点 5



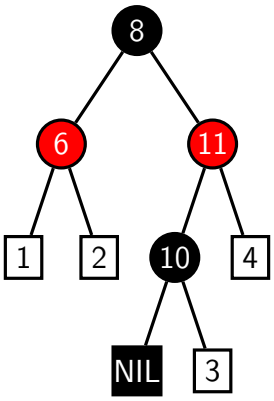
0.1.5 红黑树删除结点

红黑树的删除操作要比插入操作复杂得多。

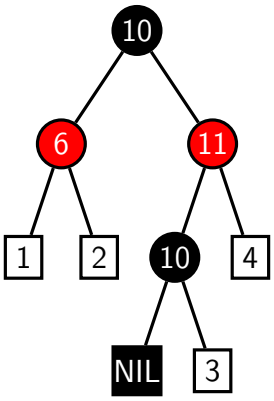
第一步

如果待删除结点有两个非空的孩子结点，转化成待删除结点只有一个孩子（或没有孩子）的情况。

例如删除结点 8:



因为结点 8 有两个孩子，可以选择仅大于 8 的结点 10 复制到 8 的位置，结点颜色变成待删除结点的颜色。

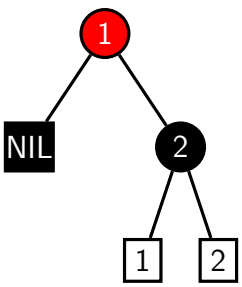


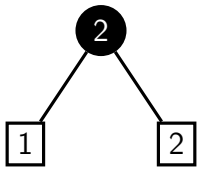
结点 10 能成为仅大于 8 的结点，必定没有左孩子结点，所以问题转换成了待删除结点只有一个右孩子（或者没有孩子）的情况。

第二步

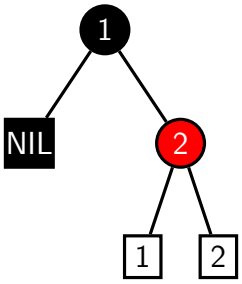
根据待删除结点和其唯一子结点的颜色，分情况处理。

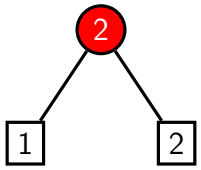
情况 1：自身是红色，子结点是黑色。直接按照二叉查找树的删除操作，删除结点 1 即可。



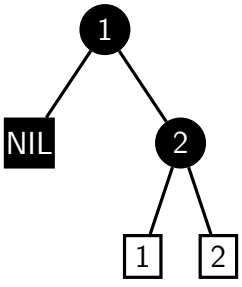


情况 2：自身是黑色，子结点是红色。按照二叉查找树的删除操作，删除结点 1。此时这条路径凭空少了一个黑色结点，因此需要将结点 2 变成黑色即可。





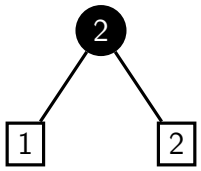
情况 3：自身是黑色，子结点也是黑色，或者子结点是空叶子结点。这种情况最为复杂，涉及到很多变化。首先还是按照二叉查找树的删除操作，删除结点 1。此时这条路径凭空少了一个黑色结点，而且并不能改变结点 2 的颜色来解决问题。这时需要进入第三步，专门解决父子双黑的情况。



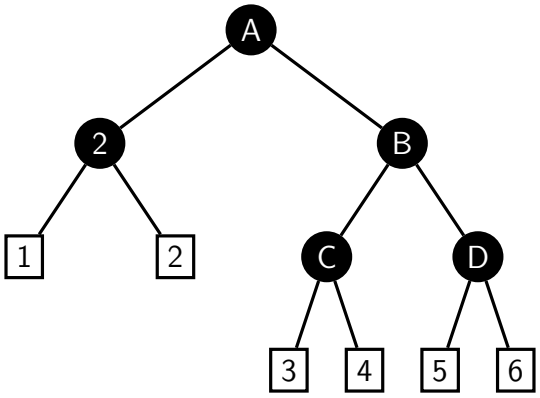
第三步

遇到双黑结点，在子结点顶替父结点后，可分为 6 种情况处理。

情况 1：结点 2 是红黑树的根。此时所有路径都减少了一个黑色结点，并未打破规则，无需调整。



情况 2：结点 2 的父亲、兄弟和侄子结点都是黑色。



直接把结点 2 的兄弟结点变为红色。这样结点 B 所在路径少了一个黑色结点，两边扯平了。