Red Black Tree

红黑树

描述：

红黑树（Red Black Tree）是目前最广泛使用的一种自平衡二叉查找树。AVL 树在插入和删除操作中，从根节点向下找到指定节点完成插入或删除操作后，需要再从底向上回到根节点，对途中的所有节点判断是否需要 LL、RR、LR、RL 四个自平衡操作，从底向上的时间复杂度为$O(log_2 N)$，因此 AVL 树的插入和删除操作的时间复杂度也可以看作$O(2\times log_2 N)$，而红黑树在从底向上的过程中只需要 2-3 次的自平衡操作，不需要遍历$O(log_2 N)$数量的节点，因此是目前速度最快的一种自平衡二叉查找树。红黑树的查找、插入、删除操作的平均时间复杂度都是$O(log_2 N)$，高度为$O(log_2 N)$。

红黑树是 2-3-4 树的一种等同，很多算法教科书介绍 2-3-4 树是因为它是理解红黑树算法的重要工具，但 2-3-4 树本身在实践中并不经常使用。

除了二叉查找树的基本性质，红黑树还具有以下 5 点性质：

(1) 节点是红色或黑色的；

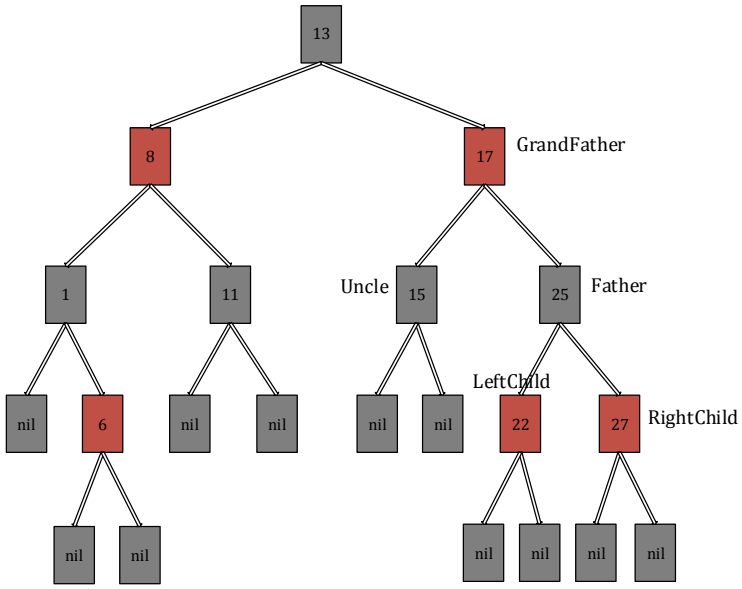
(2) 根节点是黑色的；

(3) 所有空节点（nil 节点）是黑色的；

(4) 每个红色节点必然有两个黑色子节点，或者说，任意从根节点向下到叶子结点的路径上不能有两个连续的红色节点；

(5) 从任意节点向下到叶子结点的所有路径上都包含相同数量的黑色节点；

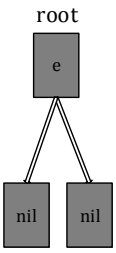
如图：



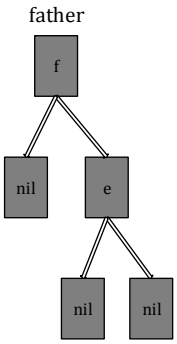
与 AVL 树类似，在红黑树的插入和删除操作中，也需要对树进行染色和旋转等操作来保证红黑树的上面 5 条性质。基本的插入和删除操作，都是二叉搜索，找到对应的插入和删除位置。

插入操作，首先按照二分法查找到新节点的位置，将新节点插入，默认情况下新节点 e 染色为 Red，由于新节点会破坏红黑树的性质，因此需要从新节点 e 开始向上调整这棵红黑树，调整的操作如下：

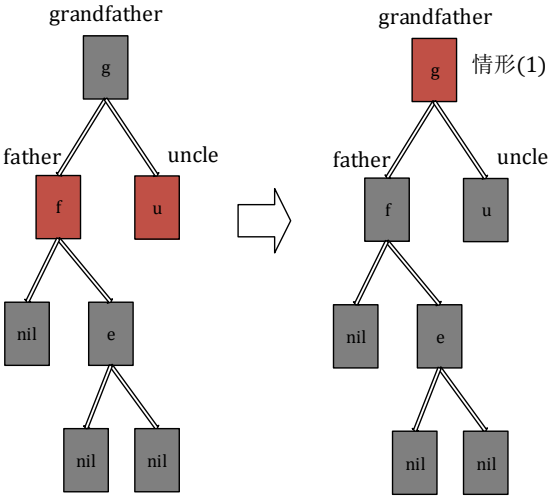
(插入操作 1) 若新节点 e 的插入位置为根节点，没有 Father，则 e 染色为 Black 以满足性质 2，算法结束。否则进入(插入操作 2)；



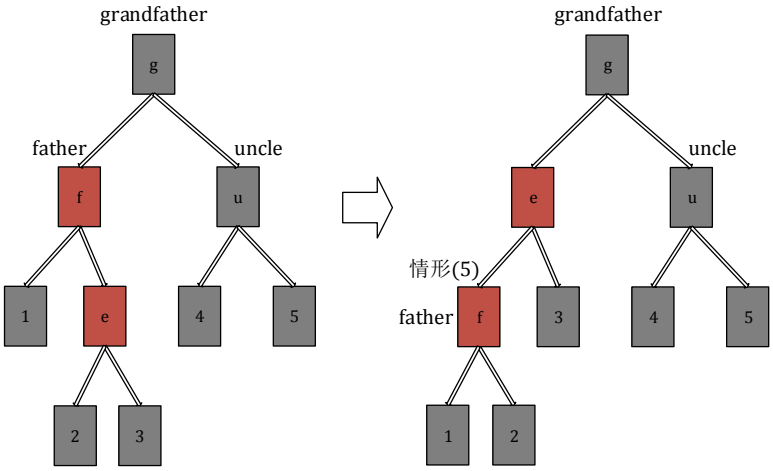
(插入操作 2) 若新节点 e 的插入位置的父节点 Father 为 Black，不需任何改变就满足性质 4，算法结束。否则进入(插入操作 3)；



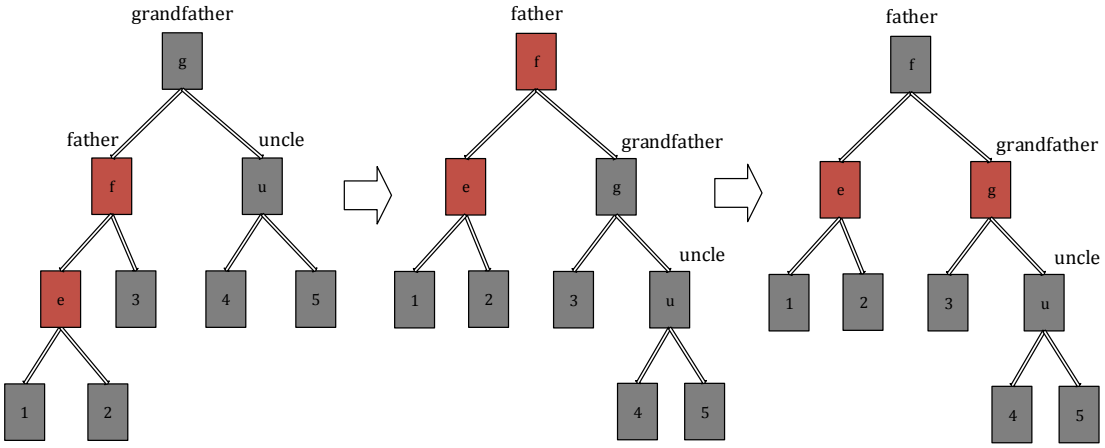
(插入操作 3) 若新节点 e 的插入位置的父节点 Father 和叔父节点 Uncle 都是 Red，则将 Father 和 Uncle 染色为 Black，并将祖父节点 GrandFather 染色为 Red，又由于 GrandFather 可能是根节点，则把 GrandFather 当作新插入的节点递归的从(插入操作 1)开始操作；



(插入操作 4) 若新节点 e 的插入位置的父节点 Father 是 Red，而叔父节点 Uncle 是 Black 或缺少，并且节点 e 是 Father 的右孩子节点，Father 是 GrandFather 的左孩子节点。这时进行一次左旋转（和 AVL 树的旋转操作类似，但不一样）来调整 e 和 Father 的位置，然后对 Father 按照(插入操作 5)进行处理，来解决性质 4 的失效问题。注意因为节点 e 和 Father 都是 Red，所以其他节点的性质 5 仍然有效，算法结束；



(插入操作 5) 若新节点 e 的插入位置的父节点 Father 是 Red，叔父节点 Uncle 是 Black 或缺少，并且新节点 e 是 Father 的左孩子节点，父节点 Father 是 GrandFather 的左孩子节点。这时对 GrandFather 进行一次右旋转，旋转后的 e 和 Father 节点是相邻的，并且都是 Red，因此再交换 GrandFather 和 Father 的颜色，算法结束；



删除操作：

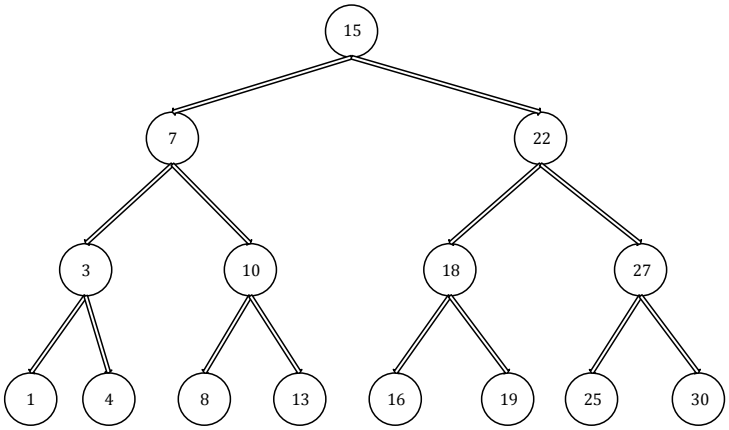
(删除操作 0) 首先按照二叉平衡树的方式将节点 e 删除，这个过程分 3 种情况：

(删除操作 0-a)节点 e 没有孩子节点，即 e 为叶子节点，直接删除节点 e，红黑树的性质不会改变，不需要调整；

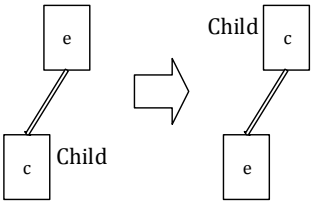
(删除操作 0-b)节点 e 只有 1 个孩子节点，直接删除节点 e，并用其孩子节点替代 e 的位置，红黑树的性质可能会改变，需要从 e 的孩子节点开始向上进行调整；

(删除操作 0-c)节点 e 有 2 个孩子节点。首先找出他的后继节点 next，将 next 的值（内容）复制到节点 e，然后再删除后继节点 next。这样就将问题转化为了删除节点 next 的情况。但节点 next 只可能有(a)没有孩子节点，(b)只有 1 个孩子节点这 2 种情况，因此用(删除操作 0-a)和(删除操作 0-b)进行处理，因为(删除操作 0-a)直接删除 next 即可，因此接下来只考虑(删除操作 0-b)，后面我们仍然称需要被删除的节点 next 为 e；

注：平衡二叉树中的后继节点是指按照节点顺序排列的下一个节点，一个拥有 2 个孩子节点的节点，其后继节点必然是一个叶子节点或叶子结点上一层的节点。在下图中，对于所有拥有 2 个孩子节点的节点来说，3 的后继节点是 4，7 的后继节点是 8，10 的后继节点是 13，15 的后继节点是 16，18 的后继节点是 19，22 的后继节点是 25，27 的后继节点是 30：



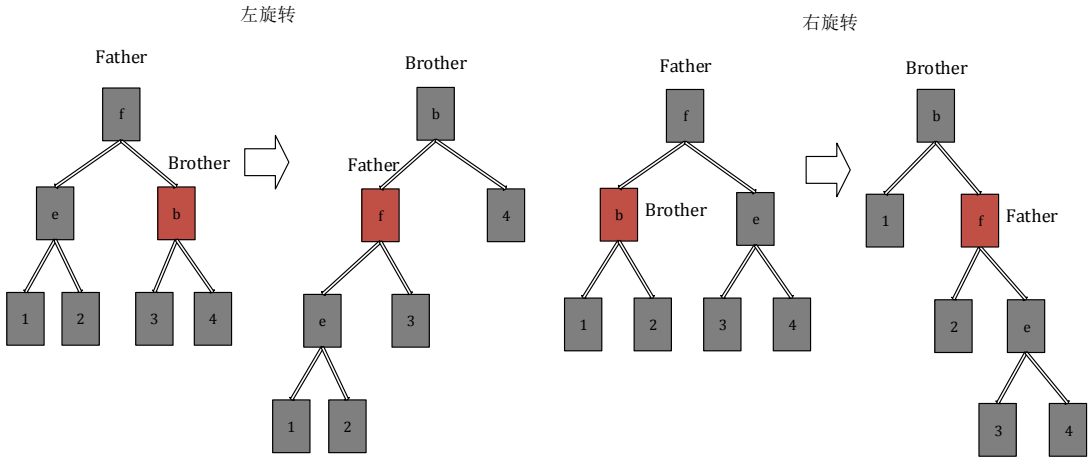
根据(删除操作 0)可以看出，最终需要删除的节点 e（节点 e 是(删除操作 0-b)中的节点 e，或(删除操作 0-c)中的节点 next），它有且只有 1 个孩子节点，即节点 e 是叶子节点的父节点，位于红黑树的倒数第二层。



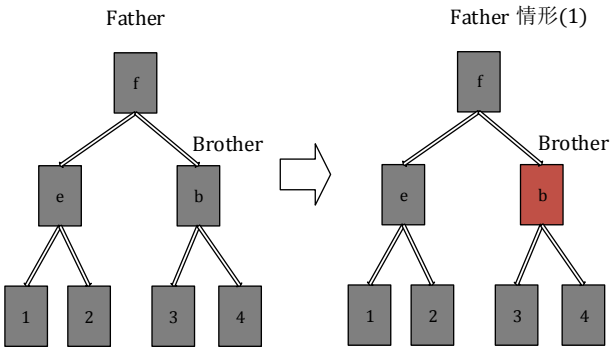
交换 e 和 Child 后，对 Child 调整的操作如下：

(删除操作 1) 若节点 e（此处节点 e 为上面的 Child 节点，后面也一样）为根节点，算法结束。否则通过(删除操作 2)对 e 进行操作；

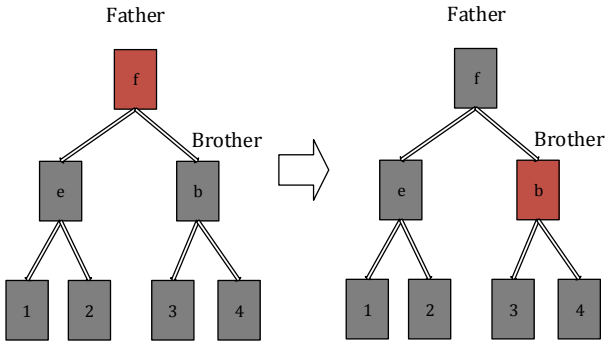
(删除操作 2) 若节点 e 的兄弟节点 Brother 为 Red，并且 e 是 Father 的左孩子节点，则进行左旋转操作，然后再交换 Father 和 Brother 的颜色。若节点 e 的兄弟节点 Brother 为 Red，并且 e 为 Father 的右孩子节点，则进行右旋转操作，然后再交换 Father 和 Brother 的颜色，算法结束；



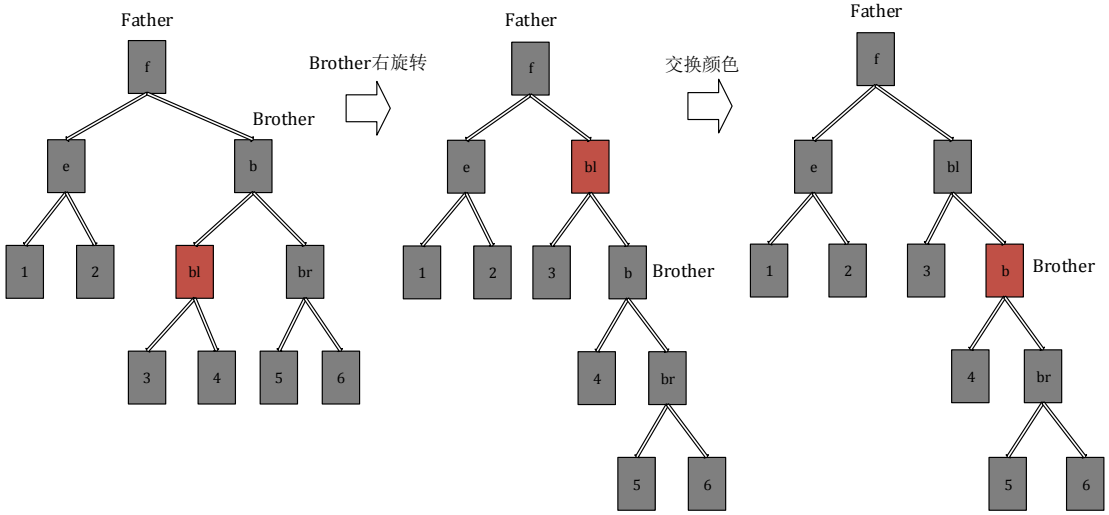
(删除操作 3) 若节点 e 的 Father 和 Brother 都是 Black，并且 Brother 的左右孩子节点都是 Black，将 Brother 染色为 Red，并对 Father 重新从(删除操作 1)进行操作，因为 Brother 更改颜色后 Father 这一条路上的黑色节点减少了不再满足性质 5；



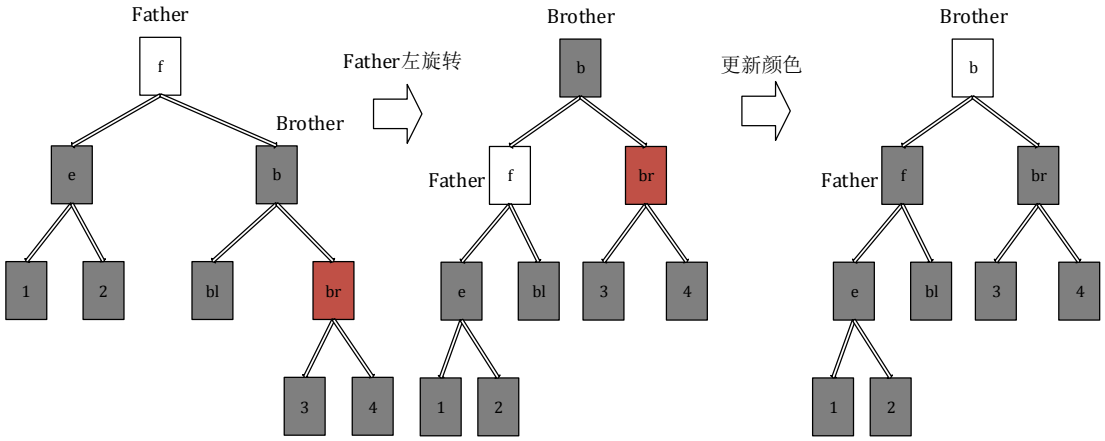
(删除操作 4) 若节点 e 和 Brother 是 Black，Brother 的孩子节点也是 Black，而 Father 是 Red，交换 Father 和 Brother 的颜色，算法结束；



(删除操作 5) 若节点 e 是 Father 的左孩子节点，Brother 节点是 Black，并且 Brother 的左孩子节点是 Red，右孩子节点是 Black，对 Brother 节点进行右旋转操作，然后交换 Brother 和它的父节点（该父节点是 Brother 的原左孩子节点）的颜色，接着对节点 e 进行(删除操作 6)（删除操作 5 和删除操作 6 是连起来的）；



(删除操作 6) 若节点 e 是 Father 的左孩子节点，Brother 是 Black，并且 Brother 的右孩子节点是 Red，对 Father 节点进行左旋转，然后交换 Father 和它的父节点的颜色，再将 Brother 的右孩子节点染色为 Black，算法结束；


Father
f
e
b
Brother
1
2
bl
br
3
4
Father左旋转
Brother
b
Father
f
br
e
bl
3
4
1
2
更新颜色
Brother
b
Father
f
br
e
bl
3
4
1
2