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，因此AVL树的插入和删除操作的时间复杂度也可以看作，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，高度为。

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；

