红黑树讲解

资料整理：刘冬煜

红黑树，同样是一种自平衡二叉搜索树，由 Rudolf Bayer 最先提出，当时被称为平衡二叉 B 树（其实红黑树本质就是一棵 B-tree），后来被 Leo J. Guibas 和 Robert Sedgewick 修改为"红黑树"。

红黑树具有如下性质：

1.红黑树是一棵平衡二叉搜索树，其中序遍历单调不减。

2.节点是红色或黑色。

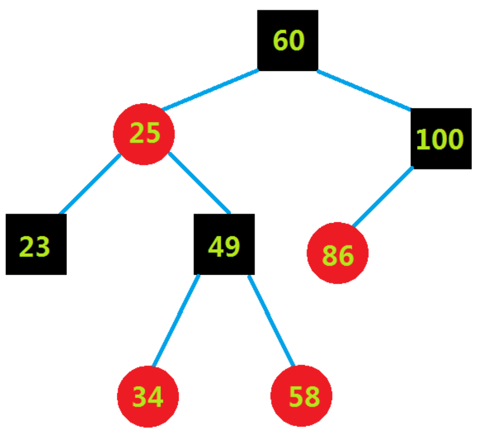
3.根节点是黑色。

4.每个叶节点(也有称外部节点的，目的是将红黑树变为真二叉树，即 NULL 节点，空节点)是黑色的。

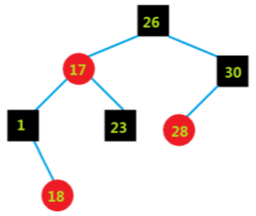
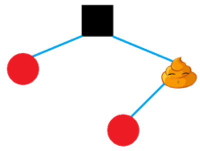
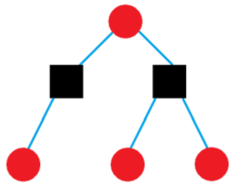
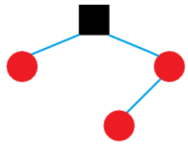
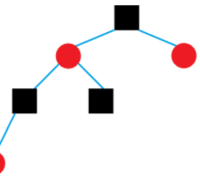
5.每个红色节点的两个子节点都是黑色。(换句话说，从每个叶子到根的所有路径上不能有两个连续的红色节点)

6.从根节点到每个叶子的所有路径都包含相同数目的黑色节点（这个数值叫做黑高度）。

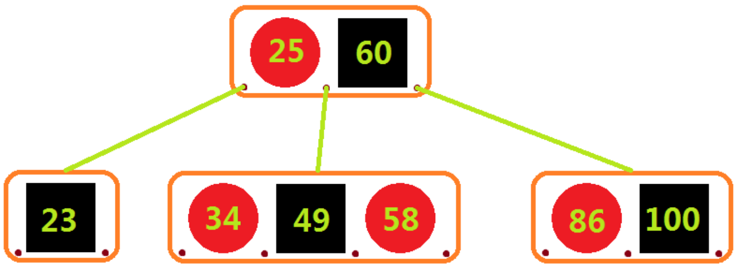
如下面一棵树就是红黑树（请自行脑补外部节点）：



而这几棵树就不是：

					
违反性质编号	1	2	3	5	6

事实上，每一颗红黑树都有等价的 B-树，如上图的红黑树对应的等价 B-树（2,3,4 树）如下：



因此，学好红黑树的诀窍就是，**心中有 B−树**。

红黑树有几个变种，如 AA 树等，今天介绍的将是最常见的**标准红黑树模板**。

节点

RB_Node 结构体，维护信息、左右儿子、父亲、前驱后继函数、真后继函数。

迭代器

iterator 结构体，各种重载。

搜索

private : find(T)和 public : lower_bound(T)、public : upper_bound(T)、public : search(T)的实现，学过其他自平衡 BST 的朋友应该比较容易写出来。

插入与双红现象

为了尽可能维护性质 6，每一次插入，都要将节点作为红色节点插入。BST 的插入应该比较容易做到，但是问题在于，性质 5 可能因此被破坏，即如果被插入的节点的父亲是红色，就会出现**双红现象**。

比如，对于上面的红黑树，插入 1 不会引起双红现象，但插入 59 或 80 都会引起双红现象。

和其他自平衡二叉搜索树一样，红黑树只有在出现缺陷时才进行修正。而双红缺陷就是红黑树可能出现的两大缺陷之一。

双红修正

对于双红现象，我们将会分以下三种情况修正：

RR-0　（没有双红现象）

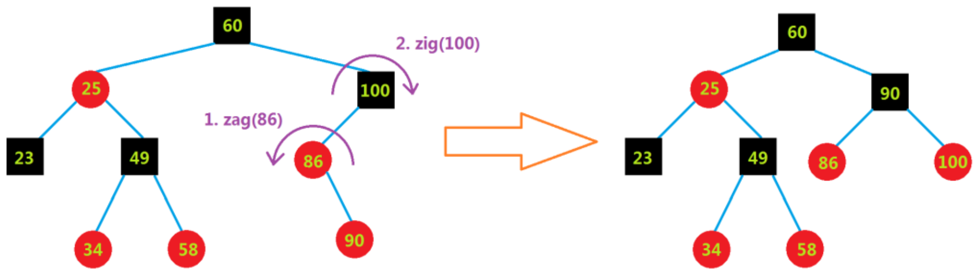
如果正在修正的节点的父亲是黑色，那么修正就已经结束了。（RR-2 的递归基）

RR-1　（叔叔是黑色）（插入到等价 B-树的三节点中）

如果正在修正的节点的父亲是红色（那么祖父一定是黑色），但是叔叔是黑色，那么只需要做 1 或 2 次旋转，再进行 2 次染色就可以解决。（同时也是 RR-2 的递归基）

如对于上面的红黑树，插入 80 之后，就会触发 RR-1 修正，此时只需要做一次旋转，两次染色即可：（右旋祖父 100，再染红 100，染黑 86）

同样，如果插入 90 之后，也会触发 RR-1 修正，此时只需要做两次旋转，两次染色即可：（先左旋父亲 86，再右旋 100，最后染红 100，染黑 90）

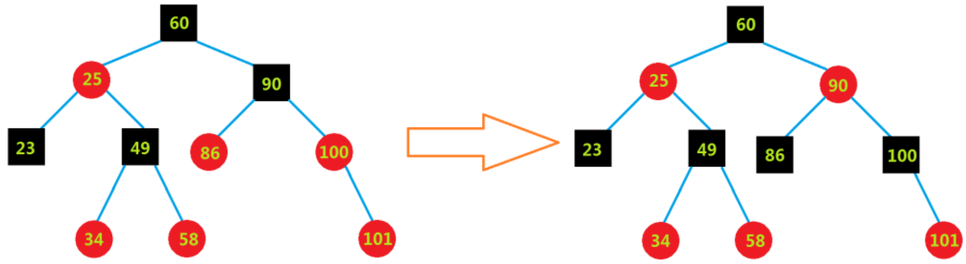


对称的情况也可以对称处理。

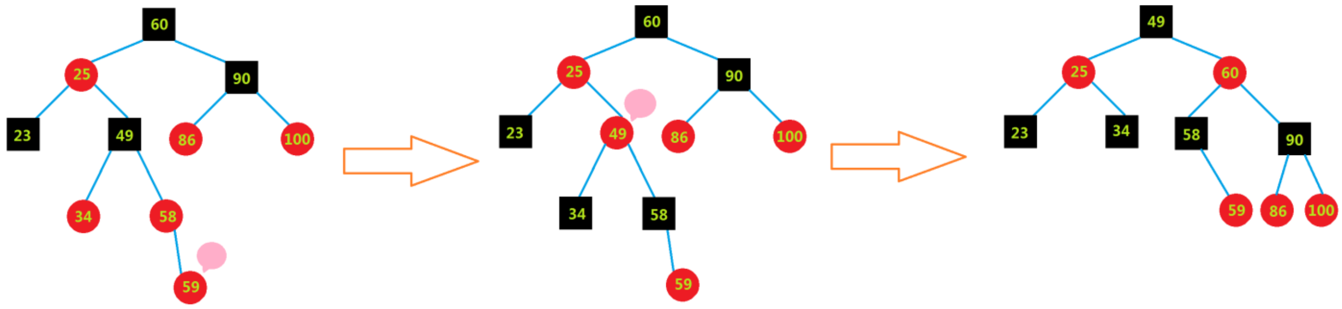
RR-2　（叔叔是红色）（插入到等价 B-树的四节点中）

如果正在修正的节点的父亲是红色（那么祖父一定是黑色），而且叔叔也是红色，那么递归就开始了。

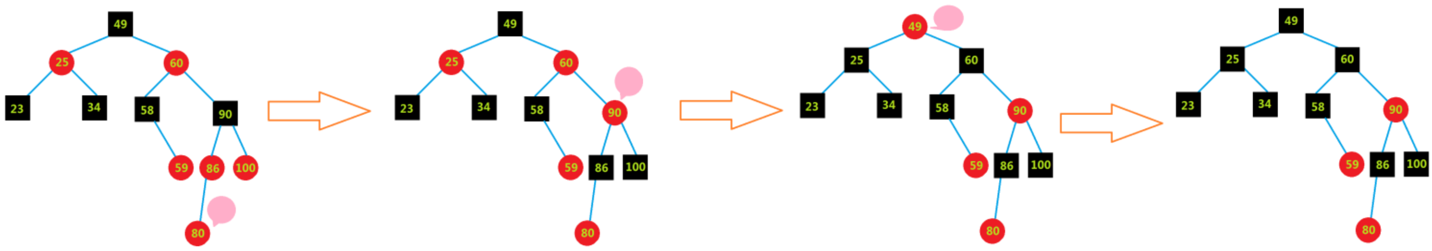
如对于刚刚插入 90 的红黑树，再插入 101 之后，就会触发 RR-2 递归。不过幸运的是，这次递归深度只有 1，因为修正之后就是 RR-0：（染黑 101 的父亲 100、叔叔 86，然后染红祖父 90）



而如果插入的是 59 而不是 101，会触发 RR-2 递归，可是深度依然为 1，因为 RR-2 修正之后就是 RR-1，这就是说 RR-1 也是 RR-2 的递归基的原因：（RR-2 过程：染黑父亲 58、叔叔 34，然后染红祖父 49，转化为 RR-1。RR-1 过程：左旋 25，右旋 60，然后染红 60，染黑 49。这时根节点也发生了改变，注意维护根节点）



那如果继续插入 80 呢？这将是一个递归深度为 3 的 RR-2 修正，最终将触发 RR-2 的第三个递归基——递归达到根节点。这时只需要染黑根节点即可，同时全树黑高度+1：（两次 RR-2 和最后一次染黑根节点的过程，在这张图中也比较明确了，也不再赘述）



可以看出，虽然 RR-2 需要递归解决缺陷，但是递归一定可以结束，因为每次缺陷必会**上升两层**，直到最终达到根节点更新全树黑高度，或者中途触发 RR-0 或 RR-1 结束。

同时，和其他平衡二叉搜索树不同，但和 B-树相同，红黑树不是向下通过枝叶生长的，而是向上通过根生长。

其他接口

双红现象及其修正讲完了，讲完其他接口之后呢，红黑树这个数据结构就已经讲完一半了！那另一半呢？当然是——删除、失黑现象及失黑修正，红黑树的第二大修正。因为红黑树的删除过于麻烦，清华大学教材《数据结构》对此叙述十分混乱，《STL 源码剖析》更是对此闭口不谈。其实红黑树的删除还是比较容易写出的，只是思维上难度较大。

作为热场，在这里我会先讲红黑树的其他接口：

begin() （迭代器起始）

当我们封装好红黑树后，用户依然可以用 iterator 来按元素从小到大的顺序遍历整棵树。而其实迭代器就是 begin(void)。

非常好写有木有！

end()　（迭代器结束、search(T)失败的返回值）

不就是 iterator(NULL)吗！

size()　（包含元素数量）

我们维护着呢！

empty()　（判空）

不就是!_size 吗！

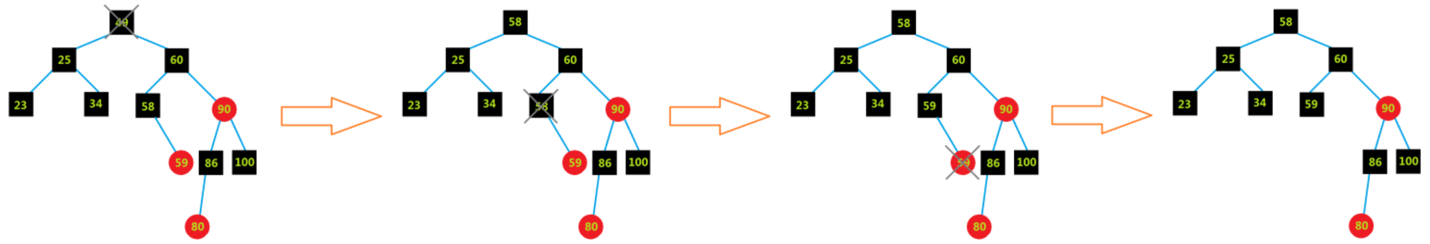
clear()　（移除整棵树）

后序 dfs 遍历一次就 ok 了。（其实 bfs 也是可以的，但为了好写，我选择 dfs）

删除与失黑现象

千呼万唤始出来，犹抱琵琶半遮面。事实上，删除操作才是红黑树真正的难点。第一次写红黑树数据结构，我用了 1 个小时写完了其他操作并调试完毕，然后又用了 4~5 个小时才写完删除，并且调试还花了我 2 天。因为删除需要分五种情况，除去没有失黑现象之外，单纯失黑修正就要占四种情况，而且转化关系也不是很好理解。

首先先介绍删除的思路：一路找到被删除节点的真后继，然后回来覆盖原节点，如此反复。如上面我们刚刚插入过 80 的红黑树，如果我们想删除根节点 49，那我们就要进行如下过程：（找到 49 的真后继 58，覆盖 49；然后找到 58 的真后继 59，覆盖 58；然后删除 59 即可）



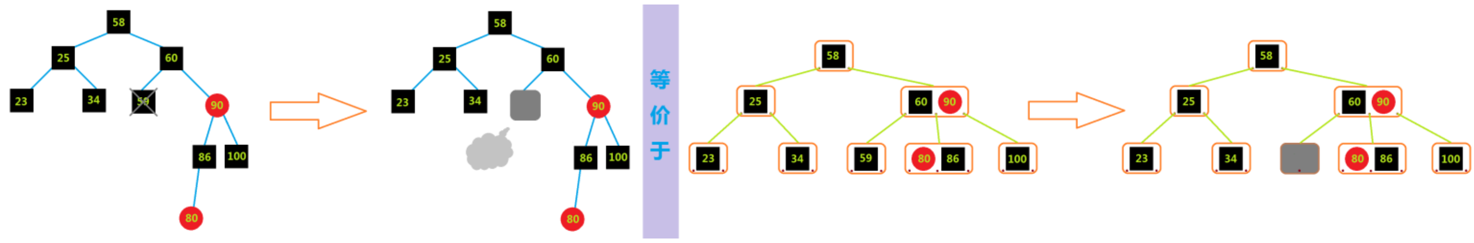
这个过程，我们最后删除的是红节点，所以并没有涉及失黑修正。如果我们最后删除的是黑节点（等价 B-树的二节点），那么麻烦就来了：性质 6 就会被破坏。

失黑修正

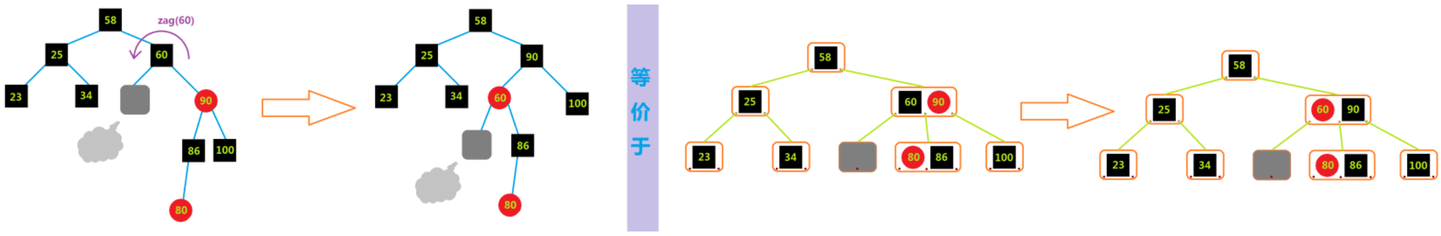
正如前面讲到的，失黑修正有四种情况（根据父亲和兄弟颜色划分），而且失黑节点不像双红节点一样可以比较容易地看出，所以这时就需要我们**心中有 B–树**了。为了方便，我们将需要递归的放在前两种讲解。

LB-1　（父亲为黑色，兄弟为红色）

对于上面的红黑树，如果我们删除 59（其实删除 58 效果也一样，这里为了方便直接删除最终真后继 59），那么就会出现破坏性质 6 的缺陷：

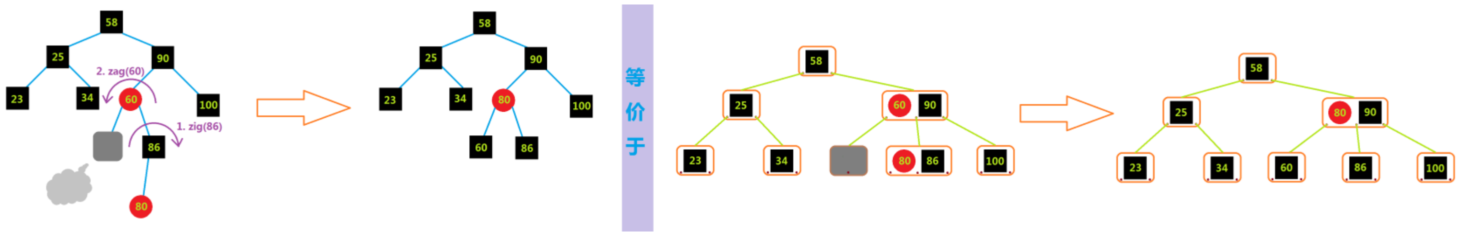


对于 LB-1，我们不能直接解决，但是我们可以利用一次旋转，触发递归深度为 1 的递归，将它转化为不需要递归的 LB-2R 或者 LB-3：（这里转化成了 LB-3，方法是左旋父亲 60）

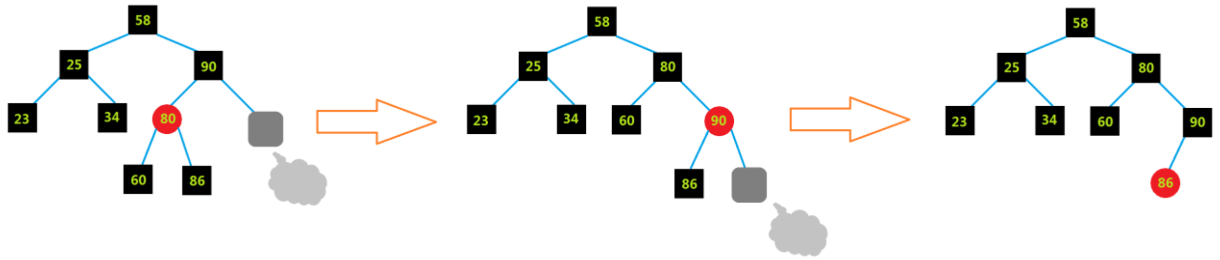


和 RR-2、LB-2 不同，这里的缺陷并没有上升，但是显然我们不需要进一步递归了，因为父亲变红了，不会触发 LB-1、LB-2B 两个需要递归的修正了。

这里我们利用 LB-3 把这棵树修正好：（右旋兄弟 86、再左旋父亲 60，然后染黑 60，这个修正在后面会讲到）



如果删除 100 呢？同样会触发 LB-1 修正，只不过递归后会触发 LB-2R，此时按照 LB-2R 进行修正：（LB-1 过程：右旋 90，染红 90，染黑 80，转化为 LB-2R。LB-2R 过程：染红 86，染黑 90）



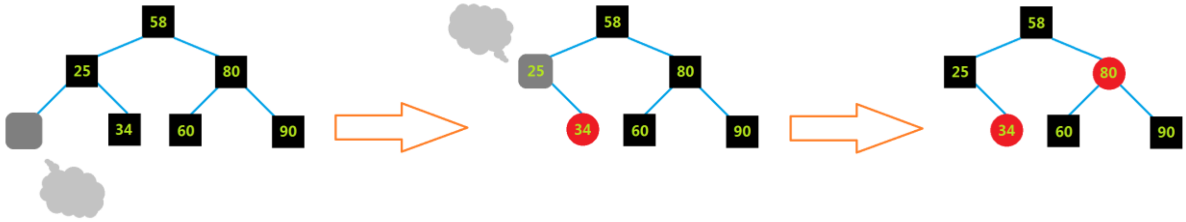
这时全树唯一的红节点挂在了最下面，独占一层，~~惹怒了强迫症患者我不负责任。~~

算了，我还是把 86 删掉吧，毕竟我也是强迫症患者……

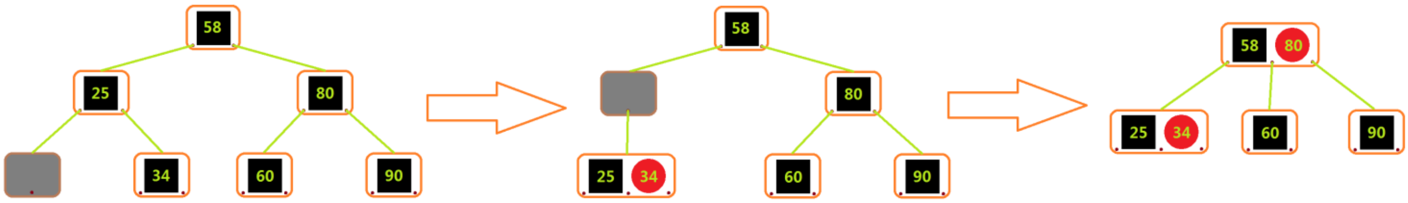
LB-2B　（没有红色侄子，且父亲为黑色，兄弟为黑色）

显而易见，这时自己、父亲、兄弟都独占二节点。这是红黑树失黑修正过程中唯一可能出现对数递归的情况。这时要染红兄弟，然后递归修正父亲。

比如，上面的红黑树（删除了 86 的节点全黑红黑树），我们想再删除 23，就会触发递归深度为 3 的修正，全树黑高度-1：（第一次 LB-2B 过程：染红 34，转化为 LB-2B。第二次 LB-2B 过程：染红 80。第三次 LB-2B 过程：递归到根节点 58，结束修正）



等价 B-树的变化如下：

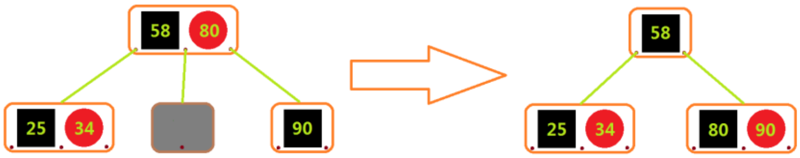


至此，递归的情况已经讲完了，接下来就是非递归的两种情况。

LB-2R　（没有红色侄子，且父亲为红色，兄弟为黑色）

再讲 LB-1 的时候已经提到 LB-2R 的修正方法了，只需要对父亲和兄弟各做一次重染色即可。
例如对于上面的红黑树，我们删去 60，就触发了 LB-2R 的修正：（染红兄弟 90，染黑父亲 80）

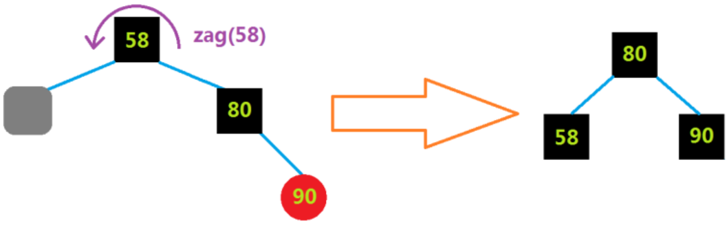
很简单对吧？可是它的等价 B-树并不很赞同：



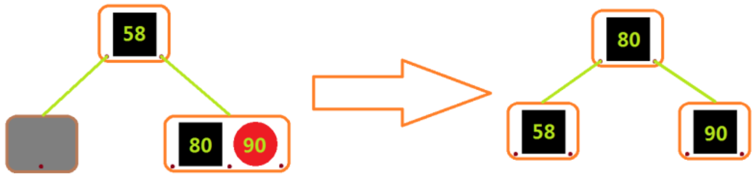
不管怎样至少比较对称了……
但是对称就不好玩了，于是我决定删掉 34。

LB-3　（有红色侄子）

LB-3 的情况在前面也有提到，修正方法是 1 或 2 次旋转，然后 1 或 2 次重染色。
例如，对于上面的红黑树，我又丧心病狂地删除了 25，就会触发 LB-3 修正：（左旋 58，染黑 90，同时要注意维护根节点）



有人问：不染黑 90 而是染红 58 不也可以吗？这时我只能这么回答：**你的心中没有 B-树**。



关于 B-树的强调贯穿了整个删除的讲解，因为对于理解失黑修正，**等价 B-树太重要了**。

两大修正的一些总结

不管是双红修正还是失黑修正，均涉及到重染色的操作，而其中 RR-1、LB-1、LB-3 均需要旋转，注意维护根节点。

RR-2、LB-2B 分别对应着 B-树的上溢和下溢，均需要递归。

RR-2 能递归到 RR-0、RR-1、RR-2 三种双红修正情况中的任何一种，LB-2B 同样也会递归到 LB-1、LB-2B、LB-2R、LB-3 四种失黑修正情况中的任何一种。而同样是递归，LB-1 只能转移到 LB-2R 和 LB-3。

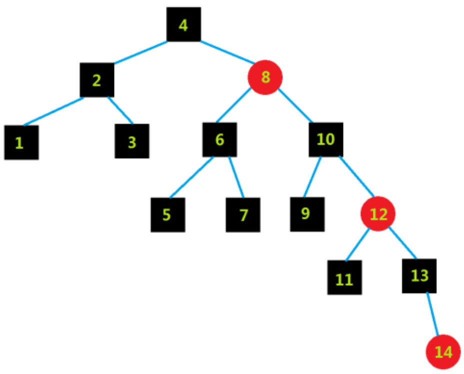
等价 B-树对于我们的理解也很重要，不要忽视。

用势能分析法可以证明双红修正、失黑修正的时间复杂度分摊 O(1)，有兴趣的朋友可以百度。

其他推论

1.按照元素单调的顺序插入到红黑树可以得到偏红黑树（所有红节点在同一枝上）或者其它根节点左右子树域之差较大的红黑树。

比如按顺序 1~14 插入，得到如下的右偏红黑树：



2.黑高度为 h 的偏红黑树，内部节点数最多为$2^{h+1}-2$，此时树高为 2h，从根出发的最短树链长度为 h，最长树链长度为 2h，根节点的左右子树域之

差的绝对值为2^h-1，高度之差为 h。

上图的红黑树符合了这一性质，实际上这个推论是可以用归纳法证明的。

3.内部节点数为 n 的红黑树，其黑高度最小为$\lceil\log_4(n+1)\rceil$，其黑高度最大为$1+\left\lfloor\log_2(\frac{n+1}{2})\right\rfloor$；树高最小为$\lfloor\log_2 n\rfloor$，树高最大为$\max\{2\cdot(\lfloor\log_2(n+2)\rfloor-1),\ 2\cdot\left\lfloor\log_2\left(\frac{n+2}{3}\right)\right\rfloor+1\}$。

证明过程比较麻烦，有兴趣的朋友可以证明一下

写在最后

红黑树不是邪教，我自认为红黑树比线段树写起来方便简单。

红黑树稳定，应用广，**重点是快啊**！

指针也不是邪教，就算它是邪教，教主也不是我。

别看了，都结束了
该干啥干啥去吧

如果觉得本篇有帮助，不如请作者喝一杯 82 年的 java（手动滑稽）

微信支付