红黑树的插入、删除操作之为什么

tseesing

January 14, 2019

红黑树是一种近似平衡的二叉搜索树[1]，因其“平衡”（尽管是近似的），其搜索效率比普通二叉搜索树高。既然是“平衡”的，插入、删除节点时，不可像普通搜索二叉树一般，插入或删除后就撒手不管，红黑树要多做一些工作，以维持“平衡”。

这个数据结构学习得太痛苦，得作下笔记。难点就在核心的插入、删除节点后为恢复红黑树性质而作的调整。

红黑树性质 [1]（红黑树操作起来复杂，全是为了让整棵树维持着以下这些性质）：

1. 每个节点要么是红色，要么是黑色
2. 根节点是黑色
3. 每个叶节点（NIL）是黑色
4. 如果一个节点是红色的，它的两个孩子必须是黑色 [2]
5. 任意一个节点，从该节点到其所有后代节点的各个简单路径（simple path），黑色节点的数目相同（即黑高相同）。

1 节点变化时为什么要调整？

为了平衡！做法就是：将有变化的子树通过旋转、变色操作，使其重新符合红黑树的 5 个性质。

2 说的“平衡”是什么？

红黑树性质5，黑色节点（黑高）相同。每一个节点为根的子树，其左右子树黑高相同：是为“平衡”。

3 插入操作相关的为什么

以下是书本 [1] 抄出的伪代码（第 9 行 else if 我擅自分成了两行，方便理解。另外要吐槽下中译版 [1]，将伪代码从 11 行开始分割印到两页上，读者要拿尺出来量缩进才知道下一页开始时的代码属于哪个 block！）

[1] 见算法导论第 13 章

[2] 自注：窃以为可理解为父子不能同红

```
RB-INSERT-FIXUP(T, z)

while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK       // case 1
            y.color = BLACK         // case 1
            z.p.p.color = RED       // case 1
            z = z.p.p               // case 1
        else
            if z == z.p.right
                z = z.p             // case 2
                LEFT-ROTATE(T, z)   // case 2
            z.p.color = BLACK       // case 3
            z.p.p.color = RED       // case 3
            RIGHT-ROTATE(T, z.p.p)  // case 3

    else
        （与 z.p 为 z.p.p 的左孩子的情况对称）

T.root.color = BLACK
```

Figure 1: 插入节点后恢复红黑树性质的伪代码

3.1 为什么新插入的节点设置成红色？

《算法导论（第 3 版）》[1] 也问这个问题，我懒，当然不确定自己的想法是否正确，为了不误导人，直接查到了书中*练习题 13.3-1* 的答案 [2]. 原因就是为了

> 保持红黑树的性质5 — 黑高数目不变！

如果加入的是黑色节点，性质4确实不违反；但明显性质会5违反。直接不平衡！如果加入的是红色节点，性质4可能违反，但性质5却明显不违反！—— 一个是有可能违反，一个明显违反！

3.2 插入调整的伪代码是怎么完成调整的？

新插入的节点 Z 与父节点 P 是否同为红色？同就要调整，使其符合性质4。(对照 “插入修正过程2” 时，请将图中的 N 看作 Z)。
否则，看叔节点的颜色了，

- 如果叔节点也是红的，就将叔变黑，父 z.p 也变黑，但 z.p.p 要变红（相当于

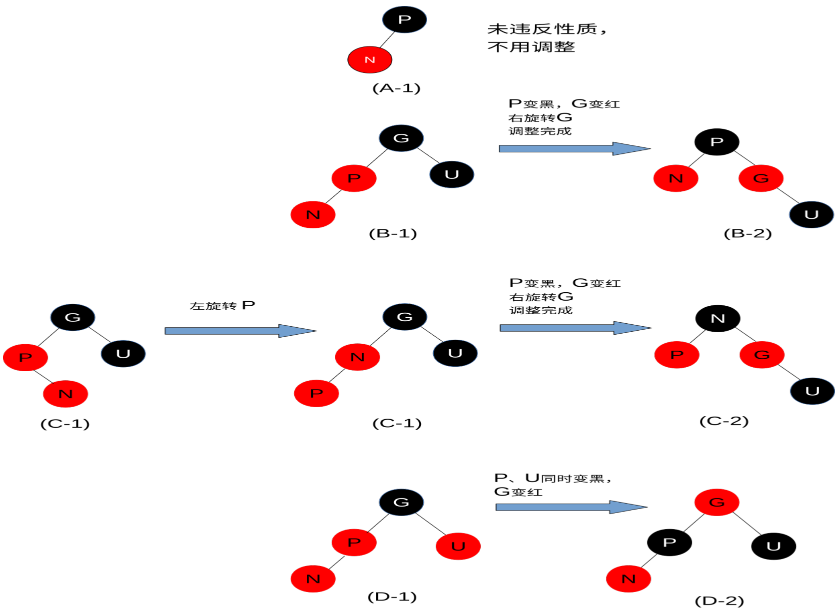


Figure 2: 插入修正过程（N 节点为新插入的节点）

z.p.p 这个黑高改由父、叔子树各自持有）。此时，z.p 子树恢复了性质4。叔节点为根的子树的性质也没被破坏。

可是爷 z.p.p 变红了，万一存在 z.p.p.p，而且也是红的，就会违反性质4；不存在 z.p.p.p 时（即 z.p.p 已是树根），也会违反性质2。所以，执行 $z = z.p.p$，令 z 提升到爷节点那层，while 循环以爷节点为新的起点，重复调整过程。

- 如果叔节点是黑的，由于 while 循环的条件已经明确了 $z.p.color == RED$, 所以，根据性质4, 知道 $z.p.p.color == BLACK$, 这种情况下，首先确保节点 z 处于 z.p 的左子树上（L9, L10, L11）[3]

  随后，将爷节点 z.p.p 变红、父节点 z.p 变黑；节点 z.p.p 右旋转一下，这样，原来叔的子叔没有增加黑高，父的子树也恢复了性质4。调整完成。

（本次操作不会将 NIL 的颜色变红。z.p.p 要变颜色的情况，父节点是红，就不可能是树根，那父节点本身肯定还有它的父节点，即 z.p.p 不会是 NIL；另外，只有叔是红节点的情况下才有可能需要叔节点变色，红色节点注定不会是 NIL3。所以不用担心变色操作到 NIL。）

总结：插入时，没有破坏性质就不必调整。破坏性质就在 z.p.p 子树内看看能不能将性质恢复，不能，则先调整好 z.p.p 子树以满足所有性质，然后从 z.p.p 作为新起点，向“上级”寻求调整。

[3]这是因为：如果 z 是 z.p 的右孩子，接下来的旋转操作，会使 z 这个红色节点成为原来 z.p.p 的左孩子，仍然会出现父子同红的情况。这里可参照图2辅助理解

3.2.1 为什么要到达 z.p.p 级来找子树调整？

请画图！画出符合红黑树性质的图来，就很容易明白原因：一旦需要调整就问候叔叔节点的情况，是因为有红黑树性质4、性质5限制，要进入 while 循环，已经明确无误父节点是红色的，父是红色时，非 NIL 的兄弟根本不存在。

对于**新插入的节点**，如果需要调整，肯定是没有非 NIL 的兄弟的 (父子不能同红，若兄弟是黑，则兄弟所在子树就会多一个黑高，不符合红黑树性质！既然红或黑的节点都不符合性质，只能不存在了)，所以，在 z.p 这一级调不了啊！只能通过 z.p.p 这一级了。

注：不需要调整的情况下，可能有一个兄弟（并且只可能是红色的）

3.3 为什么 while 的循环条件是 z.p.color == RED ？

z.p 不红就不需要调整啊！得看了上边3.2.1的阐述才好理解。新插入的节点，是红色的，z.p 也是红色时的才会导致性质破坏，才需要进入该循环内调整！另外，如果是 while 第二次循环之后进入的，也是因为是红色节点有可能破坏了红黑树性质, 同样的理由。

上述分析是 `if z.p == z.p.p.left` 的情形, 与 `z.p == z.p.p.right` 的情形是对称的，理解通一个，另一个自然明白。

最后的 *T.root.color = BLACK*，只是强制保证性质2 —— 根节点是黑的。

4 插入节点后恢复性质操作总结

如果插入节点破坏了红黑树性质，则可以在 z.p.p 子树内部通过与叔、父、爷三节点协作变色或旋转，可恢复完成；但若是将 z.p.p 变红了，需要以 z.p.p 为新起点，重复调整过程。

5 删除操作相关的为什么

以下是从《算法导论 (第 3 版)》[1] 抄来的删除操作的调整的伪代码（稍有变动，原本 12 行的 else if 分成两行，方便理解）。[4]

[4]这次机械工业出版社没将伪代码分页了！但特么的，本来原书中第 9 行 if 与第 12 行的 else 是配对的，缩进是同样的，可中译版中两者的缩进并不同样，要是没能看到原书的读者，一脸懵逼：这几个语句块，谁跟谁是配对的？

```
RB-DELETE(T, z)

…

x = y.right
if y.p == z
    x.p = y

…

if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)


RB-DELETE-FIXUP(T, x)

while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color == BLACK                // case 1
            x.p.color == RED                // case 1
            LEFT-ROTATE(T, x.p)             // case 1
            w = x.p.right                   // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                   // case 2
            x = x.p                         // case 2
        else
            if w.right.color == BLACK
                w.left.color = BLACK        // case 3
                w.color = RED               // case 3
                RIGHT-ROTATE(T, w)          // case 3
                w = x.p.right               // case 3
            w.color = x.p.color             // case 4
            x.p.color = BLACK               // case 4
            w.right.color = BLACK           // case 4
            LEFT-ROTATE(T, x.p)             // case 4
            x = T.root                      // case 4
    else
        (与上边伪代码操作对称)
x.color ==  BLACK
```

先说明下：要删除一个节点[5] *，可以找它的前驱或后继节点来补上空出的位置，但本文*

[5]网络好多资料会说只需将后继（或前驱）节点的数据，替换掉欲删除的节点的数据，然后让后继（或前驱）作为真正被删除的节点，话虽如此，但实际开发时务必注意，相信采用像 C 语言来实现时，多数是从

找的是后继节点

这里没贴出书本中完整的 `RB-DELETE(T, z)`，在理解前，需明白这两段伪代码的 y-original-color 是什么，以及形参 x 是哪个节点！

- 如果待删除的 D 节点*有两个非 NIL 的孩子*，y 记为 D 的后继节点。y-original-color 就是 y 的颜色；x 就是 y 的右孩子（注：x 有可能是 NIL）；— *这种情况下，y 被抽去顶替被删除的节点了，y 的位置则 x 节点顶替）*
- 否则，y 就是被删除的节点，y-original-color 就是 y 的颜色；x 就是被删除节点的孩子

5.1 RB-DELETE 中为什么 y-original-color == BLACK 时才调用修正函数？

如果真正被除掉的节点（y）是红色的，删了后红黑树的性质仍然得到保持。只有删除掉黑色的节点才会破坏红黑树性质 — 黑高少了一！子树不平衡了，需调整！

5.2 为什么删除后调整函数的 while 循环是判断 x 黑色？

调用本函数，就意味着 y 节点是黑色的，如果 x 是红的，直接将 x 置黑（伪代码第 23 行），将少了的一个黑高补上，就完成了调整。但如果 x 是黑色，无法这样做，只好令其进入循环里调整了。

5.3 被删除的节点 y 一定有非 NIL 的兄弟节点？伪代码中似乎不用考虑这情况？

不一定（按照红黑树性质动手画个图，一目了然）。如果 y 是树根显然就没有。

但只要 y 不是 NIL 节点并且是黑色的，就肯定有。因为删除 y 前，整棵树是平衡的，即 y.p 左右子树的黑高得相同，y 充当了一个黑高，兄弟子树也得带有黑色的节点，以跟 y 平衡。

同理，也是为了平衡，如果 y 是红色的，也肯定没有！

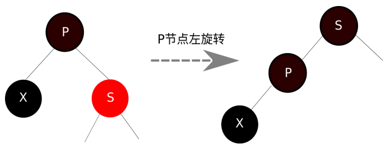
另外，伪代码其实有考虑 NIL 节点的情况的，`RB-DELETE(T, Z)` 的第 13 行，已经考虑了 x 是 NIL 节点的情况。（这行代码书中也有讲述）

5.4 删除后调整的伪代码是怎么完成调整的？

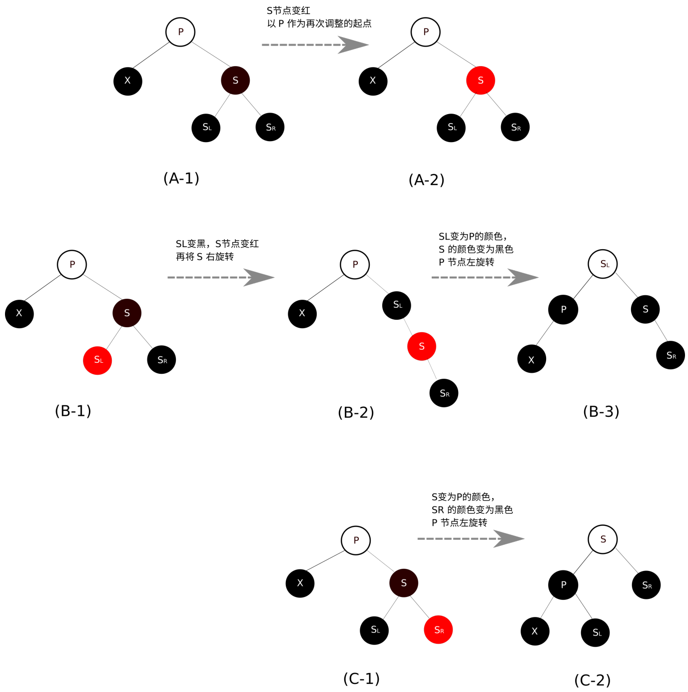
首先明确：调用 `RB-DELETE-FIXUP(T, x)` 前，出现的问题就是黑色节点 y 被摘除了，黑高少了一！

理解删除后恢复红黑树性质的操作，费了不少力气，算法导论中讲述说，弄个虚拟的额外的黑色给 x ···，看得我更难理解 ······ 整理学习笔记时，我以为我完全弄明白了。但我发觉无法按伪代码来逐段描述，一描述就陷进去，乱得不清楚伪代码是要做什

堆区分配节点内存的，FIXUP 函数收到的参数是指向待删除节点的指针，若是像所说的那样换个值就了事，使用本函数的用户又不知情，以为函数返回后，所指向的节点，已经从树上摘下了，free 掉，就坏事了！



(a) 兄弟节点为红色



(b) 兄弟节点为黑色

Figure 3: 删除后恢复性质过程

么了。后来在写实现代码时，换了个方式来想，这些伪代码段要在处理什么样的情况，才逐渐明朗，最终绘成了图片3a与 3b（为了便于跟踪节点是如何旋转调整的，节点的字母我没变动）：

图3a 的情况是兄弟是红的，由性质4、性质2推出父肯定存在，而且是黑色的。所要做的就是挪父节点 P 去补回缺失的黑高：红兄弟 S 变黑，右旋转 P 节点，使得 P 补回缺失的黑高去了，S 又刚好顶替了 P 的位置。重新平衡！完工。

图3b 的情况是兄弟是黑的，由于被删除的 y 节点也是黑的，P 节点的颜色无法确定，但依然无法阻挡红黑树的发明者（等你弄明白了红黑树结构，想必也会惊叹发明者思想的精妙）。此情况下，要看看 S 的子节点有没有可以利用的红节点，

1. A 行的图，表明 S 没有红色子节点，没办法在 P 子树内自行恢复红黑树性质，只好 S 变红，将 S 子树也自减一个黑高，现在 P 左右是平衡了，但 P 子树仍然少一个黑高！所以，以 P 为新的起点，再循环，好向上层寻求调整。就会出现以下三种可能：

   - 如果 P 是黑的，以 P 为新的调整起点，循环重复调整过程。

- 如果 P 已经是树根，此时，树根左右已经是平衡的了，循环结束。
- 如果 P 是红的，循环结束，FIXUP 函数返回时将 P 变黑（第 23 行），平衡了！

2. B 行的图，S 的左孩子 S_L 是红的, 则将 S 变红，S_L 变黑, 右旋转 S，将 S_L 顶替 S 的位置。再将 S_R 使用 S_L 的颜色（黑色），S_L 使用 P 的颜色，P 变黑，左旋转 P，现在 P 补回了被删除的黑高，原来的兄弟子树性质也没变！完工。
3. C 行的图，S 的右孩子 S_R 是红的, 去除图中节点标记的字母，只看颜色，是不是跟上述 B 行的右旋转 S 之后的情况一样了？[6]

5.5 删除操作一系列旋转、变色，究竟有没有动到了 NIL 叶子节点？

可能有！但也仅是变更其指针指向某个节点。从始至终，颜色是肯定没有被改成红色的。

[6] 等我画完3b的图，我算是意识到了为什么看别人的资料那么难，包括那本书，是因为有些资料将几种情况试图最终汇集成一种情况，所以，就看到了那些标记节点的字母在各个节点间飞来飞去。那本算法书上还介绍说假定 X 节点还有额外的一重黑色，以抵上被删除的黑高，然后目的就成了找一个节点来抵销这个 "额外" 的一重黑色，在刚开始学、没理解这个算法前，像我这样的理解能力，觉得这样的描述真是加重了复杂程度！更难理解。

References

[1] 机械工业出版社：算法导论（第 3 版）, 中译版，2015

[2] http://www2.imm.dtu.dk/ phbi/files/teaching/solution.pdf

[3] 可视化的红黑树操作（注意它的删除操作是找前继的） https://www.cs.usfca.edu/ galles/visualization/RedBlack.html