

Java Advanced Data Structures and algorithms(属于盛夏八月的高阶数据结构与算法进阶)

deepwaterooo

February 21, 2023

Contents

1 红黑树 (添加节点看懂了，删除不是很懂，改天还需要再看一下，自己写一遍代码)	1
1.1 左旋和右旋: 红黑树的基本操作 (一) 左旋和右旋	2
1.1.1 左旋	2
1.1.2 右旋	3
1.1.3 区分左旋和右旋	3
1.2 添加: 红黑树的基本操作 (二) 添加 - 关键步骤描述: 假设我们插入的新节点为 X	4
1.2.1 3. 如果 X 的 parent 不是黑色, 同时 X 也不是 root:	5
1.2.2 小例子: 插入 10, 20, 30, 15 到一个空树中	7
1.2.3 灵魂追问	8
1.3 删除: 红黑树的基本操作 (三) 删除	8
1.3.1 1. (Case 1)x 是" 黑 + 黑" 节点, x 的兄弟节点是红色	9
1.3.2 2. (Case 2) x 是" 黑 + 黑" 节点, x 的兄弟节点是黑色, x 的兄弟节点的两个孩子都是黑色	10
1.3.3 3. (Case 3)x 是“黑 + 黑”节点, x 的兄弟节点是黑色; x 的兄弟节点的左孩子是红色, 右孩子是黑色的	11
1.3.4 4. (Case 4)x 是“黑 + 黑”节点, x 的兄弟节点是黑色; x 的兄弟节点的右孩子是红色的, x 的兄弟节点的左孩子任意颜色	11
1.4 删除: 上面的讲法太晦涩了, 不好看懂, 重新来个看图片版本的	12
1.4.1 Case 1: 新节点的兄弟节点为红色, 此时若新节点在左边则做左旋操作, 否则做右旋操作, 之后再将其父节点颜色改变为红色, 兄弟节点	13
1.4.2 Case 2: 新节点的兄弟节点为黑色, 此时可能有如下情况	13
1.4.3 插入源码	15
1.4.4 删除节点:	17
1.5 性能	19
1.6 源码: java	19
1.6.1 红黑树的结点用 Java 表示数据结构:	19
1.6.2 左旋的 Java 实现如下	20
1.6.3 右旋其实就是左旋的逆操作	20
2 java.util.concurrent.ArrayBlockingQueue 原理简要分析	21
2.1 引言	21
2.1.1 它和 LinkedBlockingQueue 存在以下几个不同点:	21
2.2 原理分析	21
2.2.1 1、构造方法	22
2.2.2 2、添加元素	22
2.2.3 3、获取元素	23

3 并发容器简介	24
3.1 1.1 并发场景下的 Map	24
3.2 1.2 并发场景下的 List	25
4 Map	25
4.1 HashMap	25
4.2 ConcurrentHashMap	25
4.3 ConcurrentHashMap 的原理	25
4.3.1 Java 1.7	25
4.3.2 Java 1.8	26
4.3.3 对比 JDK1.7 与 1.8	28
4.3.4 线程安全问题	29
5 List	29
5.1 CopyOnWriteArrayList	34
5.1.1 CopyOnWriteArrayList 原理	34

1 红黑树 (添加节点看懂了，删除不是很懂，改天还需要再看一下，自己写一遍代码)

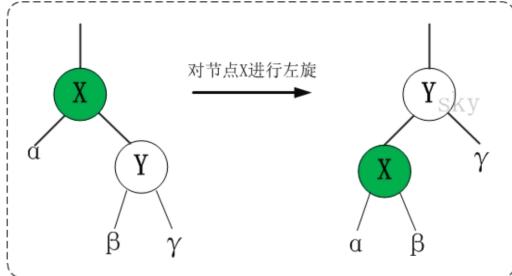
- 红黑树，Red-Black Tree「RBT」是一个自平衡 (不是绝对的平衡) 的二叉查找树 (BST)，树上的每个节点都遵循下面的规则：
 - 每个节点或者是黑色，或者是红色。
 - **树的根始终是黑色的** (黑土地孕育黑树根)
 - 每个叶子节点 (NIL) 是黑色。【注意：这里叶子节点，是指为空 (NIL 或 NULL) 的叶子节点！】
 - 没有两个相邻的红色节点 (红色节点不能有红色父节点或红色子节点，* 并没有说不能出现连续的黑色节点 *) 也有的表述成：* 如果一个节点是红色的，则它的子节点必须是黑色的。*
 - 从节点 (包括根) 到其任何后代 NULL 节点 (叶子结点下方挂的两个空节点，并且认为他们是黑色的) 的每条路径都具有相同数量的黑色节点。确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。
- 红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是 $O(\lg n)$ ，效率非常之高。
 - 例如，Java 集合中的 TreeSet 和 TreeMap，C++ STL 中的 set、map，以及 Linux 虚拟内存的管理，都是通过红黑树去实现的。
- 红黑树有两大操作：
 - recolor (重新标记黑色或红色)
 - rotation (旋转，这是树达到平衡的关键)
- 我们会先尝试 recolor，如果 recolor 不能达到红黑树的 4 点要求，然后我们尝试 rotation，其实红黑树的关键玩法就是弄清楚 recolor 和 rotation 的规则，接下来看看详细的算法公式吧

1.1 左旋和右旋：红黑树的基本操作（一）左旋和右旋

- 红黑树的基本操作是添加、删除。在对红黑树进行添加或删除之后，都会用到旋转方法。为什么呢？道理很简单，添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一颗红黑树了，而是一颗普通的树。而通过旋转，可以使这棵树重新成为红黑树。简单点说，旋转的目的是让树保持红黑树的特性。
- 旋转包括两种：左旋和右旋。下面分别对它们进行介绍。

1.1.1 左旋

1. 左旋

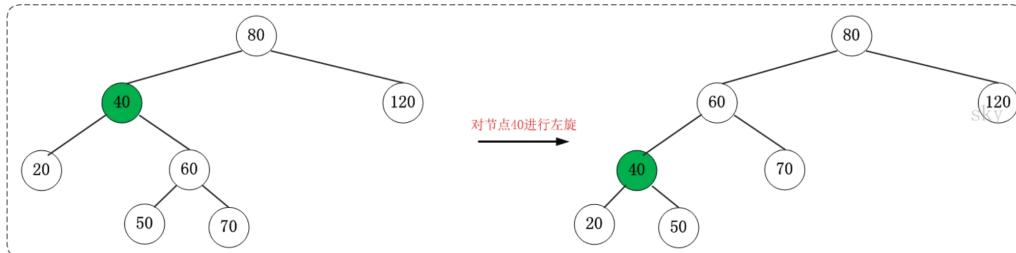


对x进行左旋，意味着“将x变成一个左节点”。

左旋的伪代码《算法导论》：参考上面的示意图和下面的伪代码，理解“红黑树T的节点x进行左旋”是如何进行的。

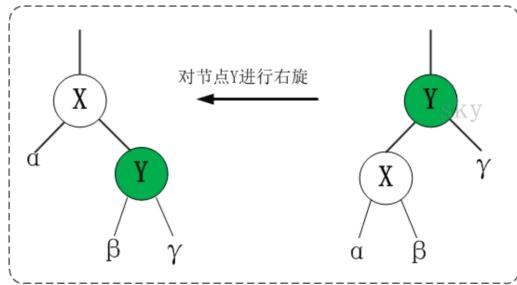
```
LEFT-ROTATE (T, x)
01 y ← right[x]           // 前提：这里假设x的右孩子为y。下面开始正式操作
02 right[x] ← left[y]     // 将“y的左孩子”设为“x的右孩子”，即 将β设为x的右孩子
03 p[left[y]] ← x         // 将“x”设为“y的左孩子的父亲”，即 将β的父亲设为x
04 p[y] ← p[x]             // 将“x的父亲”设为“y的父亲”
05 if p[x] = nil[T]
06 then root[T] ← y       // 情况1：如果“x的父亲”是空节点，则将y设为根节点
07 else if x = left[p[x]]
08     then left[p[x]] ← y // 情况2：如果x是它父节点的左孩子，则将y设为“x的父节点的左孩子”
09     else right[p[x]] ← y // 情况3：(x是它父节点的右孩子) 将y设为“x的父节点的右孩子”
10 left[y] ← x             // 将“x”设为“y的左孩子”
11 p[x] ← y               // 将“x的父节点”设为“y”
```

理解左旋之后，看看下面一个更鲜明的例子。你可以先不看右边的结果，自己尝试一下。



1.1.2 右旋

2. 右旋

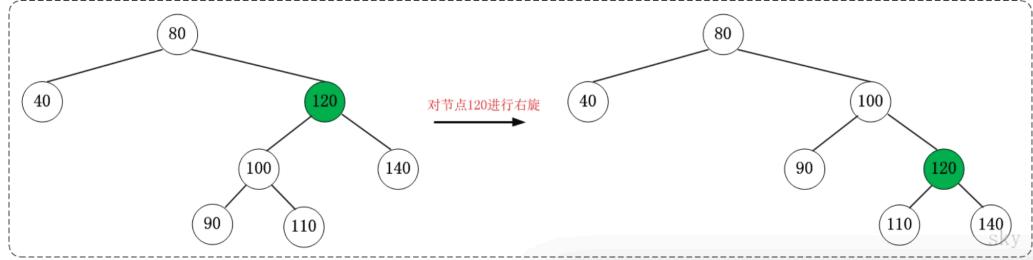


对x进行左旋，意味着“将x变成一个左节点”。

右旋的伪代码《算法导论》：参考上面的示意图和下面的伪代码，理解“红黑树T的节点y进行右旋”是如何进行的。

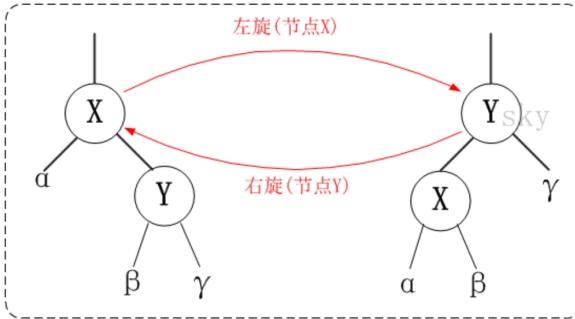
```
RIGHT-ROTATE(T, y)
01 x ← left[y]           // 前提：这里假设y的左孩子为x。下面开始正式操作
02 left[y] ← right[x]    // 将“x的右孩子”设为“y的左孩子”，即 将β设为y的左孩子
03 p[right[x]] ← y      // 将“y”设为“x的右孩子的父亲”，即 将β的父亲设为y
04 p[x] ← p[y]           // 将“y的父亲”设为“x的父亲”
05 if p[y] = nil[T]
06 then root[T] ← x       // 情况1：如果“y的父亲”是空节点，则将x设为根节点
07 else if y = right[p[y]]
08     then right[p[y]] ← x // 情况2：如果y是它父节点的右孩子，则将x设为“y的父节点的左孩子”
09     else left[p[y]] ← x // 情况3：(y是它父节点的左孩子) 将x设为“y的父节点的左孩子”
10 right[x] ← y           // 将“y”设为“x的右孩子”
11 p[y] ← x               // 将“y的父节点”设为“x”
```

理解右旋之后，看看下面一个更鲜明的例子。你可以先不看右边的结果，自己尝试一下。

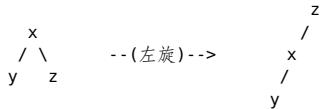


1.1.3 区分左旋和右旋

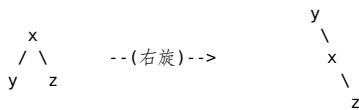
- 仔细观察上面“左旋”和“右旋”的示意图。我们能清晰的发现，它们是对称的。无论是左旋还是右旋，被旋转的树，在旋转前是二叉查找树，并且旋转之后仍然是一颗二叉查找树。



- 左旋示例图 (以 x 为节点进行左旋):



- 对 x 进行左旋，意味着，将“x 的右孩子”设为“x 的父亲节点”；即，将 x 变成了一个左节点 (x 成为了 z 的左孩子)!。因此，左旋中的“左”，意味着“被旋转的节点将变成一个左节点”。
- 右旋示例图 (以 x 为节点进行右旋):



- 对 x 进行右旋，意味着，将“x 的左孩子”设为“x 的父亲节点”；即，将 x 变成了一个右节点 (x 成为了 y 的右孩子)! 因此，* 右旋中的“右”，意味着“被旋转的节点将变成一个右节点”。*

1.2 添加: 红黑树的基本操作 (二) 添加 - 关键步骤描述: 假设我们插入的新节点为 X

- 将一个节点插入到红黑树中，需要执行哪些步骤呢？首先，将红黑树当作一颗二叉查找树，将节点插入；然后，将节点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。详细描述如下：

- 第一步: 将红黑树当作一颗二叉查找树，将节点插入。

- 红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。此外，无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。

- 好吧？那接下来，我们就来想方设法的旋转以及重新着色，使这棵树重新成为红黑树！

- 第二步: 将插入的节点着色为“红色”。

- 为什么着色成红色，而不是黑色呢？为什么呢？在回答之前，我们需要重新温习一下红黑树的特性：
 - * (1) 每个节点或者是黑色，或者是红色。
 - * (2) 根节点是黑色。
 - * (3) 每个叶子节点是黑色。[注意：这里叶子节点，是指为空的叶子节点！]
 - * (4) 如果一个节点是红色的，则它的子节点必须是黑色的。

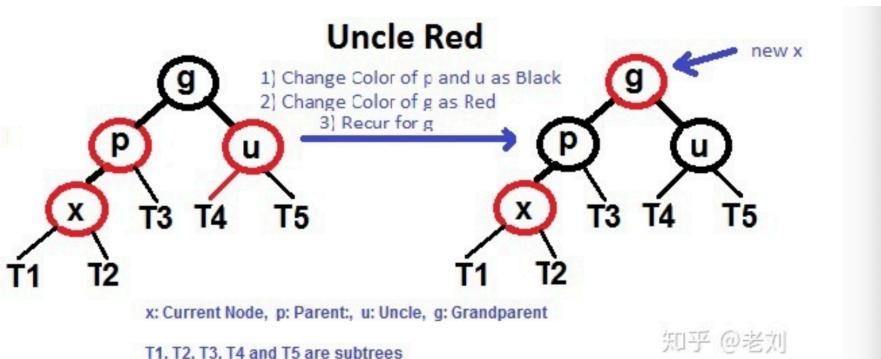
- * (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。
- 将插入的节点着色为红色，不会违背“特性(5)”！少违背一条特性，就意味着我们需要处理的情况越少。接下来，就要努力的让这棵树满足其它性质即可；满足了的话，它又是一颗红黑树了。o(O)...哈哈
- 第三步：通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。
 - 第二步中，将插入节点着色为“红色”之后，不会违背“特性(5)”。那它到底会违背哪些特性呢？
 - * 对于“特性(1)”，显然不会违背了。因为我们已经将它涂成红色了。
 - * 对于“特性(2)”，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉查找树的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。
 - * 对于“特性(3)”，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。
 - * 对于“特性(4)”，是有可能违背的！
 - 那接下来，想办法使之“满足特性(4)”，就可以将树重新构造成红黑树了。

- 用另一种更为简洁的表述法

- 1. 将新插入的节点标记为红色
- 2. 如果 X 是根结点 (root)，则标记为黑色
- 3. 如果 X 的 parent 不是黑色，同时 X 也不是 root:

1.2.1 3. 如果 X 的 parent 不是黑色，同时 X 也不是 root:

1. 3.1 如果 X 的 uncle (叔叔) 是红色
 - 3.1.1 将 parent 和 uncle 标记为黑色
 - 3.1.2 将 grand parent (祖父) 标记为红色
 - 3.1.3 让 X 节点的颜色与 X 祖父的颜色相同，然后重复步骤 2、3



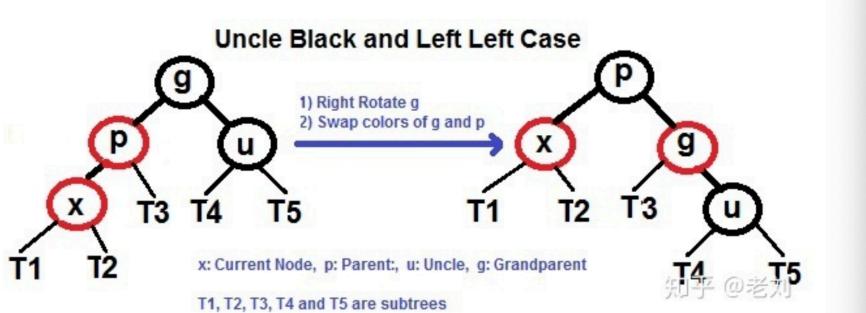
- 跟着上面的公式走：
 - 将新插入的 X 节点标记为红色
 - 发现 X 的 parent (P) 同样为红色，这违反了红黑树的第三条规则「不能有两个连续相邻的红色节点」
 - 发现 X 的 uncle (U) 同样为红色

- 将 P 和 U 标记为黑色
- 将 X 和 X 的 grand parent (G) 标记为相同颜色，即红色，继续重复公式 2、3
- 发现 G 是根结点，标记为黑色
- 结束

2. 3.2 刚刚说了 *X 的 uncle 是 * 红色的情况，接下来要说是 * 黑色 * 的情况

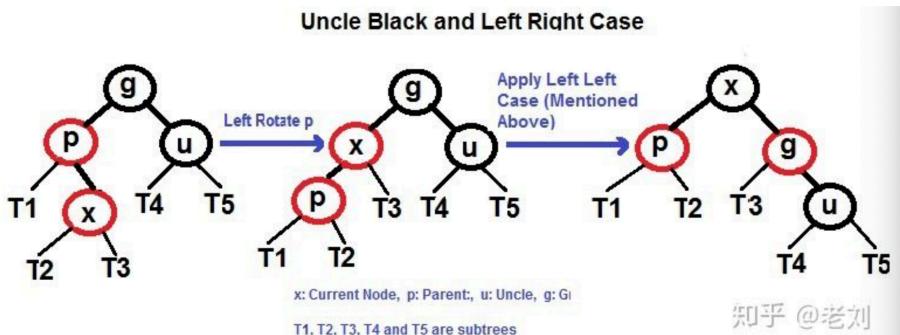
- 1. 将新插入的节点标记为红色
- 2. 如果 X 是根结点 (root)，则标记为黑色
- 3. 如果 X 的 parent 不是黑色，同时 X 也不是 root:
 - 3.1 如果 X 的 uncle (叔叔) 是红色
 - * 3.1.1 将 parent 和 uncle 标记为黑色
 - * 3.1.2 将 grand parent (祖父) 标记为红色
 - * 3.1.3 让 X 节点的颜色与 X 祖父的颜色相同，然后重复步骤 2、3
 - 3.2 如果 X 的 uncle (叔叔) 是黑色，我们要分四种情况处理
 - * 3.2.1 左左 (P 是 G 的左孩子，并且 X 是 P 的左孩子)
 - * 3.2.2 左右 (P 是 G 的左孩子，并且 X 是 P 的右孩子)
 - * 3.2.3 右右 (和 3.2.1 镜像过来，恰好相反)
 - * 3.2.4 右左 (和 3.2.2 镜像过来，恰好相反)
- 当出现 uncle 是黑色的时候我们第一步要考虑的是 旋转，这里先请小伙伴 * 不要关注红黑树的第 4 条规则 *，主要是为了演示如何旋转的，来一点点看，不要看图就慌，有解释的：

(a) 左左情况：很简单，想象这是一根绳子，手提起 P 节点，然后变色即可

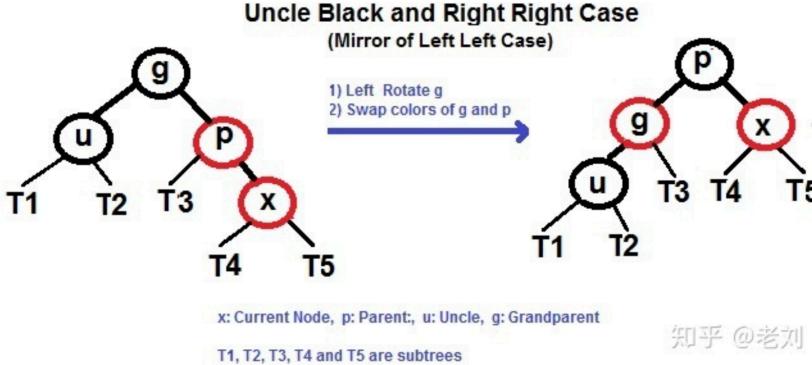


(b) 左右

- 左旋：使 X 的父节点 P 被 X 取代，同时父节点 P 成为 X 的左孩子，然后再应用左左情况



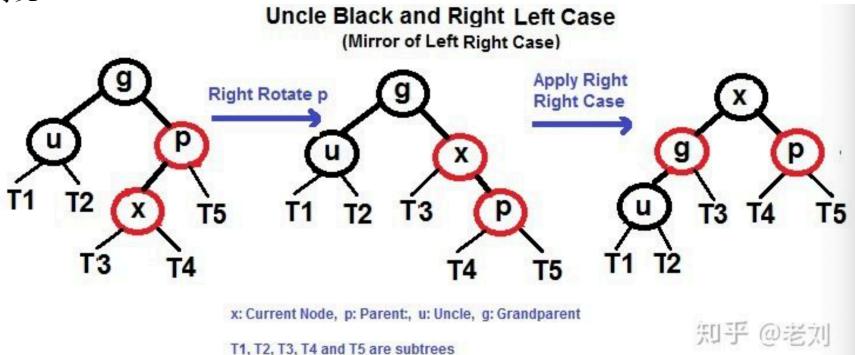
(c) 右右: 与左左情况一样, 想象成一根绳子, 手提起 P 节点, 然后变色即可



知乎 @老刘

(d) 左左

- 右旋: 使 X 的父节点 P 被 X 取代, 同时父节点 P 成为 X 的右孩子, 然后再应用右右情况



知乎 @老刘

1.2.2 小例子: 插入 10, 20, 30, 15 到一个空树中

- 向空树中第一次插入数字 10, 肯定是 root 节点
- root 节点标记成黑色
- 向树中插入新节点 20, 标记为红色
 - $20 > 10$, 并发现 10 没有叶子节点, 将新节点 20 作为 10 的右孩子
- 向树中插入新节点 30, 标记为红色
 - $30 > 10$, 查找 10 的右子树, 找到 20
 - $30 > 20$, 继续查找 20 的右子树, 发现 20 没有叶子节点, 将值插在此处
 - 30 和 20 节点都为红色, 30 为右孩子, 20 也为右孩子, 触发了右右情况
 - 通过一次旋转, 提起 20 节点
 - 20 节点是根结点, 标记为黑色
- 向树中插入新节点 15, 标记为红色
 - 通过比对大小和判断是否有叶子节点, 最终插值为 10 节点的右孩子
 - 15 和 10 节点都为红色, 15 的 uncle 节点 30 也为红色
 - 按照公式, 将 15 的 parent 10 和 uncle 30 更改为黑色
 - 让 15 节点 grand parent 20 的颜色与 15 节点的颜色一样, 变为红色
 - 20 为根结点, 将其改为黑色

1.2.3 灵魂追问

- jdk 1.8 HashMap 中有使用到红黑树，你知道触发条件是什么吗？有读过源码是如何 put 和 remove 的吗？
- 这里讲的是红黑树的 insert, delete 又是什么规则呢？
- 哪些场景可以应用红黑树？
- 你了解各种树的时间复杂度吗？
- 留个小作业，应用工具将 [10 70 32 34 13 56 32 56 21 3 62 4] 逐个插入到树中，理解红黑树 recolor 和 rotation 的转换规则
- 工具链接：<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

1.3 删除：红黑树的基本操作（三）删除

- 将红黑树内的某一个节点删除。需要执行的操作依次是：首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过“旋转和重新着色”等一系列来修正该树，使之重新成为一棵红黑树。详细描述如下：
 - 第一步：将红黑树当作一颗二叉查找树，将节点删除。
 - 这和“删除常规二叉查找树中删除节点的方法是一样的”。分 3 种情况：
 - * 被删除节点没有儿子，即为 * 叶节点。那么，直接将该节点删除就 OK 了。*
 - * 被删除节点 * 只有一个儿子 *。那么，* 直接删除该节点，并用该节点的唯一子节点顶替它的位置。*
 - * * 被删除节点有两个儿子 *。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给“该节点的内容”；之后，删除“它的后继节点”。在这里，后继节点相当于替身，在将后继节点的内容复制给“被删除节点”之后，再将后继节点删除。这样就巧妙的将问题转换为“删除后继节点”的情况了，下面就考虑后继节点。在“被删除节点”有两个非空子节点的情况下，它的后继节点不可能是双子非空。既然“的后继节点”不可能双子都非空，就意味着“该节点的后继节点”要么没有儿子，要么只有一个儿子。若没有儿子，则按“情况”进行处理；若只有一个儿子，则按“情况”进行处理。
 - 第二步：通过“旋转和重新着色”等一系列来修正该树，使之重新成为一棵红黑树。
 - 因为“第一步”中删除节点之后，可能会违背红黑树的特性。所以需要通过“旋转和重新着色”来修正该树，使之重新成为一棵红黑树。
 - 下面对删除函数进行分析。在分析之前，我们再次温习一下红黑树的几个特性：
 - (1) 每个节点或者是黑色，或者是红色。
 - (2) 根节点是黑色。
 - (3) 每个叶子节点是黑色。[注意：这里叶子节点，是指为空的叶子节点！]
 - (4) 如果一个节点是红色的，则它的子节点必须是黑色的。
 - (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。
 - 前面我们将“删除红黑树中的节点”大致分为两步，在第一步中“将红黑树当作一颗二叉查找树，将节点删除”后，* 可能违反“特性 (2)、(4)、(5)”三个特性 *。第二步需要解决上面的三个问题，进而保持红黑树的全部特性。

- 为了便于分析，我们假设“x”包含一个额外的黑色（“x”原本的颜色还存在），这样就不会违反“特性(5)”。为什么呢？
- 通过 RB-DELETE 算法，我们知道：删除节点 y 之后，x 占据了原来节点 y 的位置。既然删除 y（y 是黑色），意味着减少一个黑色节点；那么，再在该位置上增加一个黑色即可。这样，当我们假设“x”包含一个额外的黑色，就正好弥补了“删除 y 所丢失的黑色节点”，也就不会违反“特性(5)”。因此，假设“x”包含一个额外的黑色（“x”原本的颜色还存在），这样就不会违反“特性(5)”。
- 现在，“x”不仅包含它原本的颜色属性，“x”还包含一个额外的黑色。即 x 的颜色属性是“红 + 黑”或“黑 + 黑”，它违反了“特性(1)”。
- 现在，我们面临的问题，由解决“违反了特性(2)、(4)、(5)三个特性”转换成了“解决违反特性(1)、(2)、(4)三个特性”。RB-DELETE-FIXUP 需要做的就是通过算法恢复红黑树的特性(1)、(2)、(4)。RB-DELETE-FIXUP 的思想是：将 x 所包含的额外的黑色不断沿树上移（向根方向移动），直到出现下面的姿态：
 - a) x 指向一个“红 + 黑”节点。此时，将 x 设为一个“黑”节点即可。
 - b) x 指向根。此时，将 x 设为一个“黑”节点即可。
 - c) 非前面两种姿态。
- 将上面的姿态，可以概括为 3 种情况。
 - 情况说明：*x 是“红 + 黑”节点。*
 - * 处理方法：* 直接把 x 设为黑色，结束 *。此时红黑树性质全部恢复。
 - 情况说明：*x 是“黑 + 黑”节点，且 x 是根。*
 - * 处理方法：* 什么都不做，结束 *。此时红黑树性质全部恢复。
 - 情况说明：*x 是“黑 + 黑”节点，且 x 不是根。*
 - * 处理方法：这种情况又可以划分为 4 种子情况。这 4 种子情况如下表所示：*

	现象说明	处理策略
Case 1	x是“黑+黑”节点，x的兄弟节点是红色。（此时x的父节点和x的兄弟节点的子节点都是黑节点）。	(01) 将x的兄弟节点设为“黑色”。 (02) 将x的父节点设为“红色”。 (03) 对x的父节点进行左旋。 (04) 左旋后，重新设置x的兄弟节点。
Case 2	x是“黑+黑”节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。	(01) 将x的兄弟节点设为“红色”。 (02) 设置“x”的父节点为“新的x节点”。
Case 3	x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。	(01) 将x兄弟节点的左孩子设为“黑色”。 (02) 将x兄弟节点设为“红色”。 (03) 对x的兄弟节点进行右旋。 (04) 右旋后，重新设置x的兄弟节点。
Case 4	x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色。	(01) 将x父节点颜色 赋值给 x 的兄弟节点。 (02) 将x父节点设为“黑色”。 (03) 将x兄弟节点的右子节设为“黑色”。 (04) 对x的父节点进行左旋。 (05) 设置“x”为“根节点”。

1.3.1 1. (Case 1)x 是“黑 + 黑”节点，x 的兄弟节点是红色

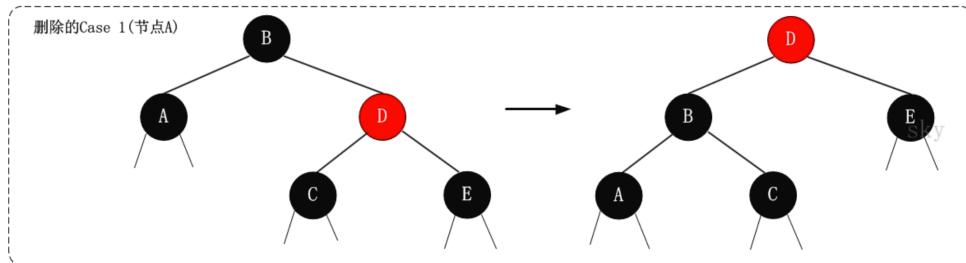
1. 1.1 现象说明

- x 是“黑 + 黑”节点，x 的兄弟节点是红色。（此时 x 的父节点和 x 的兄弟节点的子节点都是黑节点）。

2. 1.2 处理策略

- (01) 将 x 的兄弟节点设为“黑色”。
- (02) 将 x 的父节点设为“红色”。
- (03) 对 x 的父节点进行左旋。
- (04) 左旋后，重新设置 x 的兄弟节点。
- 下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)
- 这样做的目的是将“Case 1”转换为“Case 2”、“Case 3”或“Case 4”，从而进行进一步的处理。对 x 的父节点进行左旋；左旋后，为了保持红黑树特性，就需要在左旋前“将 x 的兄弟节点设为黑色”，同时“将 x 的父节点设为红色”；左旋后，由于 x 的兄弟节点发生了变化，需要更新 x 的兄弟节点，从而进行后续处理。

3. 1.3 示意图



1.3.2 2. (Case 2) x 是“黑 + 黑”节点， x 的兄弟节点是黑色， x 的兄弟节点的两个孩子都是黑色

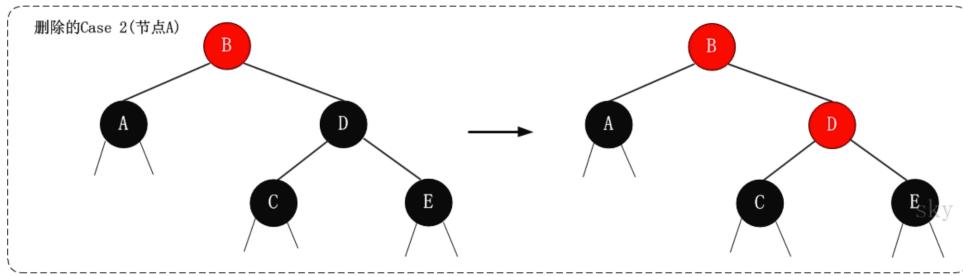
1. 2.1 现象说明

- x 是“黑 + 黑”节点， x 的兄弟节点是黑色， x 的兄弟节点的两个孩子都是黑色。

2. 2.2 处理策略

- (01) 将 x 的兄弟节点设为“红色”。
- (02) 设置“ x 的父节点”为“新的 x 节点”。
- 下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)
- 这个情况的处理思想：是将“ x 中多余的一个黑色属性上移(往根方向移动)”。 x 是“黑 + 黑”节点，我们将 x 由“黑 + 黑”节点变成“黑”节点，多余的一个“黑”属性移到 x 的父节点中，即 x 的父节点多出了一个黑属性(若 x 的父节点原先时“黑”，则此时变成了“黑 + 黑”；若 x 的父节点原先时“红”，则此时变成了“红 + 黑”)。此时，需要注意的是：所有经过 x 的分支中黑节点个数没变化；但是，所有经过 x 的兄弟节点的分支中黑色节点的个数增加了 1(因为 x 的父节点多了一个黑色属性)！为了解决这个问题，我们需要将“所有经过 x 的兄弟节点的分支中黑色节点的个数减 1”即可，那么就可以通过“将 x 的兄弟节点由黑色变成红色”来实现。
- 经过上面的步骤(将 x 的兄弟节点设为红色)，多余的一个颜色属性(黑色)已经跑到 x 的父节点中。我们需要将 x 的父节点设为“新的 x 节点”进行处理。若“新的 x 节点”是“黑 + 红”，直接将“新的 x 节点”设为黑色，即可完全解决该问题；若“新的 x 节点”是“黑 + 黑”，则需要对“新的 x 节点”进行进一步处理。

3. 2.3 示意图



1.3.3 3. (Case 3)x 是“黑 + 黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的左孩子是红色，右孩子是黑色的

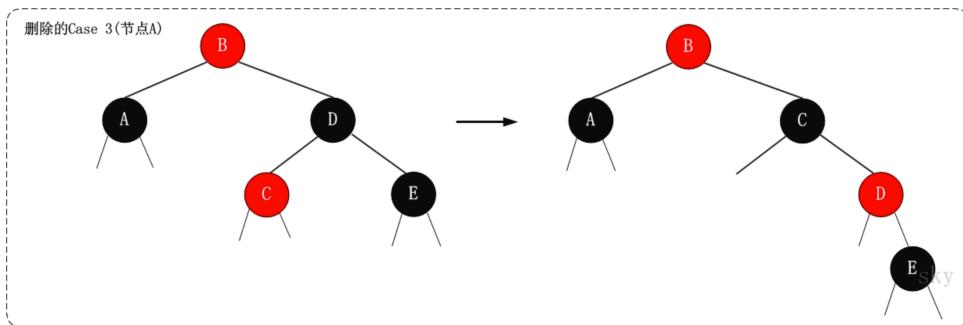
1. 3.1 现象说明

- x 是“黑 + 黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的左孩子是红色，右孩子是黑色的。

2. 3.2 处理策略

- (01) 将 x 兄弟节点的左孩子设为“黑色”。
- (02) 将 x 兄弟节点设为“红色”。
- (03) 对 x 的兄弟节点进行右旋。
- (04) 右旋后，重新设置 x 的兄弟节点。
- 下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)
 - 我们处理“Case 3”的目的是为了将“Case 3”进行转换，转换成“Case 4”，从而进行进一步的处理。转换的方式是对 x 的兄弟节点进行右旋；为了保证右旋后，它仍然是红黑树，就需要在右旋前“将 x 的兄弟节点的左孩子设为黑色”，同时“将 x 的兄弟节点设为红色”；右旋后，由于 x 的兄弟节点发生了变化，需要更新 x 的兄弟节点，从而进行后续处理。

3. 3.3 示意图



1.3.4 4. (Case 4)x 是“黑 + 黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的右孩子是红色的，x 的兄弟节点的左孩子任意颜色

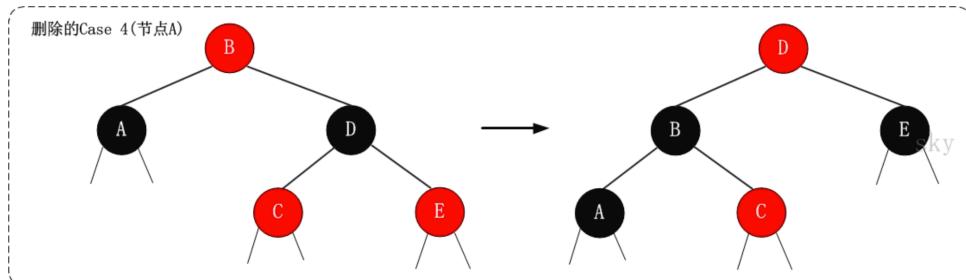
1. 4.1 现象说明

- x 是“黑 + 黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的右孩子是红色的，x 的兄弟节点的左孩子任意颜色。

2. 4.2 处理策略

- (01) 将 x 父节点颜色赋值给 x 的兄弟节点。
- (02) 将 x 父节点设为“黑色”。
- (03) 将 x 兄弟节点的右子节设为“黑色”。
- (04) 对 x 的父节点进行左旋。
- (05) 设置“ x ”为“根节点”。
- 下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)
- 我们处理“Case 4”的目的是：去掉 x 中额外的黑色，将 x 变成单独的黑色。处理的方式是“进行颜色修改，然后对 x 的父节点进行左旋。下面，我们来分析是如何实现的。
- 为了便于说明，我们设置“当前节点”为 S (Original Son)，“兄弟节点”为 B (Brother)，“兄弟节点的左孩子”为 BLS (Brother's Left Son)，“兄弟节点的右孩子”为 BRS (Brother's Right Son)，“父节点”为 F (Father)。
- 我们要对 F 进行左旋。但在左旋前，我们需要调换 F 和 B 的颜色，并设置 BRS 为黑色。为什么需要这里处理呢？因为左旋后， F 和 BLS 是父子关系，而我们已知 BL 是红色，如果 F 是红色，则违背了“特性(4)”；为了解决这一问题，我们将“ F 设置为黑色”。但是， F 设置为黑色之后，为了保证满足“特性(5)”，即为了保证左旋之后：
- 第一，“同时经过根节点和 S 的分支的黑色节点个数不变”。
- 若满足“第一”，只需要 S 丢弃它多余的颜色即可。因为 S 的颜色是“黑 + 黑”，而左旋后“同时经过根节点和 S 的分支的黑色节点个数”增加了 1；现在，只需将 S 由“黑 + 黑”变成单独的“黑”节点，即可满足“第一”。
- 第二，“同时经过根节点和 BLS 的分支的黑色节点数不变”。
- 若满足“第二”，只需要将“ F 的原始颜色”赋值给 B 即可。之前，我们已经将“ F 设置为黑色”(即，将 B 的颜色“黑色”，赋值给了 F)。至此，我们算是调换了 F 和 B 的颜色。
- 第三，“同时经过根节点和 BRS 的分支的黑色节点数不变”。
- 在“第二”已经满足的情况下，若要满足“第三”，只需要将 BRS 设置为“黑色”即可。
- 经过，上面的处理之后。红黑树的特性全部得到的满足！接着，我们将 x 设为根节点，就可以跳出 while 循环(参考伪代码)；即完成了全部处理。
- 至此，我们就完成了 Case 4 的处理。理解 Case 4 的核心，是了解如何“去掉当前节点额外的黑色”。

3. 4.3 示意图

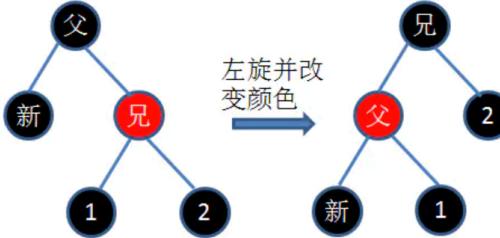


1.4 删除：上面的讲法太晦涩了，不好看懂，重新来个看图片版本的

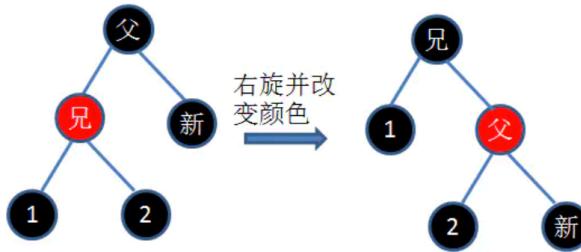
- 参考见页：<https://www.jianshu.com/p/4cd37000f4e3>
- 不知道这个哀家分析问题写代码的能力到底怎么样，看着这成片的代码还是没有看；但不管别人的代码写得好不好，仍然可以作为一个参照来帮助自己理解和改进，写出自己逻辑清晰的自己版本的代码

- 前两种简单的情况一样，不多说（被删除的节点没有子节点；或是被删除的节点只有一个子节点）。这里只说被删除的节点有两个子节点的情况
- Case 3: 被删除的节点是黑色，其子节点也是黑色，将其子节点顶替上来，变成了双黑的问题，此时有以下情况

1.4.1 Case 1: 新节点的兄弟节点为红色，此时若新节点在左边则做左旋操作，否则做右旋操作，之后再将其父节点颜色改变为红色，兄弟节点



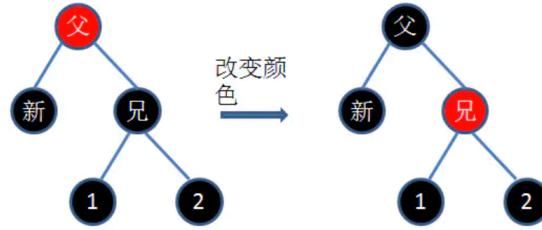
image



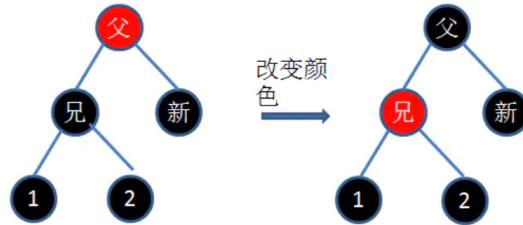
- 从图中可以看出，操作之后红黑树并未达到平衡状态，而是变成的黑兄的情况

1.4.2 Case 2: 新节点的兄弟节点为黑色，此时可能有如下情况

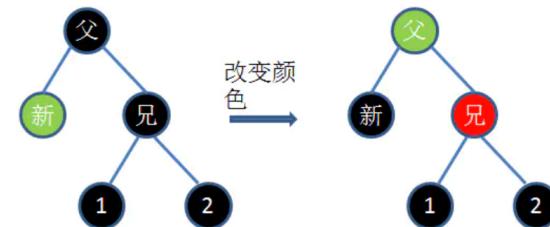
1. 红父二黑侄：将父节点变成黑色，兄弟节点变成红色，新节点变成黑色即可，如下图所示



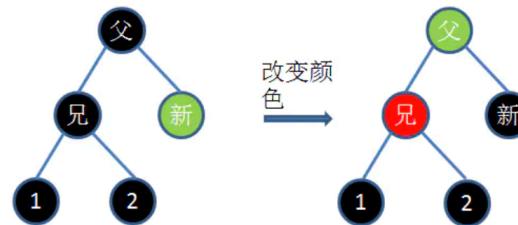
image



2. 黑父二黑侄：将父节点变成新节点的颜色，新节点变成黑色，兄弟节点染成红色，还需要继续以父节点为判断点继续判断，如下图所示

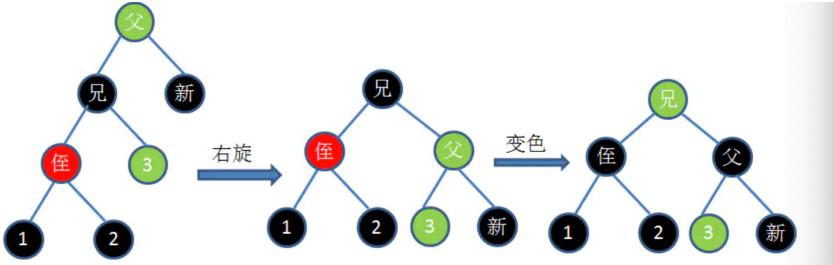


image

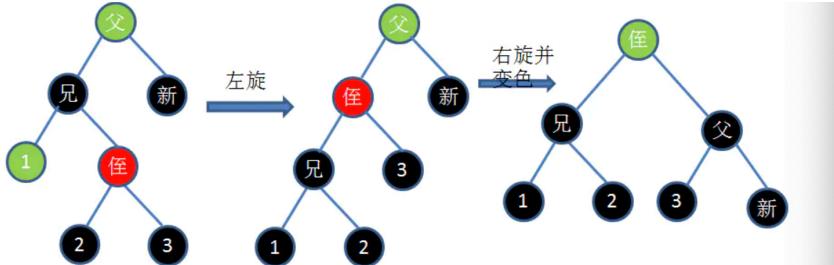


3. 红侄：

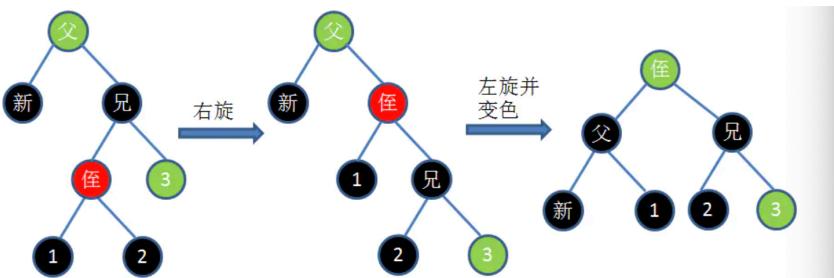
- (a) 情况一：新节点在右子树，红侄在兄弟节点左子树，此时的操作为右旋，并将兄弟节点变为父亲的颜色，父亲节点变为黑色，侄节点变为黑色，如下图所示



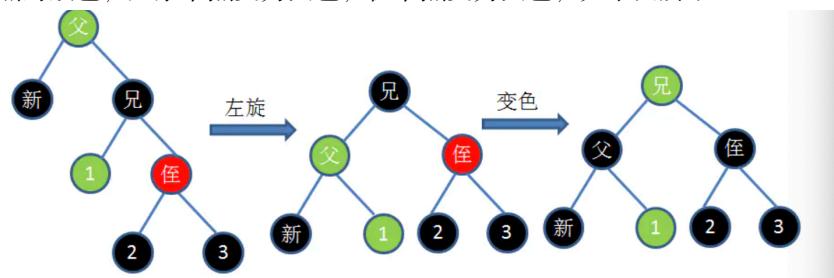
- (b) 情况二: 新节点在右子树, 红侄在兄弟节点右子树, 此时的操作为先左旋, 后右旋并将侄节点变为父亲的颜色, 父节点变为黑色, 如下图所示



- (c) 情况三: 新节点在左子树, 红侄在兄弟节点左子树, 此时的操作为先右旋在左旋并将侄节点变为父亲的颜色, 父亲节点变为黑色, 如下图所示



- (d) 情况四: 新节点在右子树, 红侄在兄弟节点右子树, 此时的操作为左旋, 并将兄弟节点变为父节点的颜色, 父亲节点变为黑色, 侄节点变为黑色, 如下图所示



1.4.3 插入源码

- 插入节点的操作主要分为以下几步:
 - 1. 定位: 即遍历整理红黑树, 确定添加的位置, 如上代码中 `insert` 方法中就是在找到添加的位置
 - 2. 修复: 这也就是前面介绍的, 添加元素后可能会使得红黑树不在满足其特性, 这时候需要通过变色、旋转来调整红黑树, 也就是如上代码中 `insertFixUp` 方法

```

// 插入一个节点
private void insert(RBTreeNode<T> node){
    int cmp;
    RBTreeNode<T> root = this.rootNode;
    RBTreeNode<T> parent = null;
    // 定位节点添加到哪个父节点下
    while(null != root){
        parent = root;
        cmp = node.key.compareTo(root.key);
        if (cmp < 0){
            root = root.left;
        } else {
            root = root.right;
        }
    }
    node.parent = parent;
    // 表示当前没一个节点，那么就当新增的节点为根节点
    if (null == parent){
        this.rootNode = node;
    } else {
        // 找出在当前父节点下新增节点的位置
        cmp = node.key.compareTo(parent.key);
        if (cmp < 0){
            parent.left = node;
        } else {
            parent.right = node;
        }
    }
    // 设置插入节点的颜色为红色
    node.color = COLOR_RED;
    // 修正为红黑树
    insertFixUp(node);
}

// 红黑树插入修正
private void insertFixUp(RBTreeNode<T> node){
    RBTreeNode<T> parent,gparent;
    // 节点的父节点存在并且为红色
    while( ((parent = getParent(node)) != null) && isRed(parent)){
        gparent = getParent(parent);
        // 如果其祖父节点是空怎么处理
        // 若父节点是祖父节点的左孩子
        if(parent == gparent.left){
            RBTreeNode<T> uncle = gparent.right;
            if ((null != uncle) && isRed(uncle)){
                setColorBlack(uncle);
                setColorBlack(parent);
                setColorRed(gparent);
                node = gparent;
                continue;
            }
            if (parent.right == node){
                RBTreeNode<T> tmp;
                leftRotate(parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }
            setColorBlack(parent);
            setColorRed(gparent);
            rightRotate(gparent);
        } else {
            RBTreeNode<T> uncle = gparent.left;
            if ((null != uncle) && isRed(uncle)){
                setColorBlack(uncle);
                setColorBlack(parent);
                setColorRed(gparent);
                node = gparent;
                continue;
            }
            if (parent.left == node){
                RBTreeNode<T> tmp;
                rightRotate(parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }
        }
    }
}

```

```

        }
        setColorBlack(parent);
        setColorRed(gparent);
        leftRotate(gparent);
    }
}
setColorBlack(this.rootNode);
}

```

1.4.4 删除节点：

- 删除节点主要分为几种情况去做对应的处理：
- 1. 删除节点，按照如下三种情况去删除节点
 - 1. 真正删除的节点没有子节点
 - 2. 真正删除的节点有一个子节点
 - 3. 正在删除的节点有两个子节点
- 2. 修复红黑树的特性，如代码中调用 removeFixUp 方法修复红黑树的特性。

```

private void remove(RBTREE<T> node){
    RBTREE<T> child, parent;
    boolean color;
    // 被删除节点左右孩子都不为空的情况
    if ((null != node.left) && (null != node.right)){
        // 获取到被删除节点的后继节点
        RBTREE<T> replace = node;

        replace = replace.right;
        while(null != replace.left){
            replace = replace.left;
        }
        // node 节点不是根节点
        if (null != getParent(node)){
            // node 是左节点
            if (getParent(node).left == node){
                getParent(node).left = replace;
            } else {
                getParent(node).right = replace;
            }
        } else {
            this.rootNode = replace;
        }
        child = replace.right;
        parent = getParent(replace);
        color = getColor(replace);
        if (parent == node){
            parent = replace;
        } else {
            if (null != child){
                setParent(child, parent);
            }
            parent.left = child;

            replace.right = node.right;
            setParent(node.right, replace);
        }
        replace.parent = node.parent;
        replace.color = node.color;
        replace.left = node.left;
        node.left.parent = replace;
        if (color == COLOR_BLACK){
            removeFixUp(child, parent);
        }
        node = null;
        return;
    }
    if (null != node.left){
        child = node.left;
    }
}

```

```

} else {
    child = node.right;
}
parent = node.parent;
color = node.color;
if (null != child){
    child.parent = parent;
}

if (null != parent){
    if (parent.left == node){
        parent.left = child;
    } else {
        parent.right = child;
    }
} else {
    this.rootNode = child;
}

if (color == COLOR_BLACK){
    removeFixUp(child, parent);
}
node = null;
}

// 删除修复
private void removeFixUp(RBTreeNode<T> node, RBTreeNode<T> parent){
    RBTreeNode<T> other;
    // node 不为空且为黑色，并且不为根节点
    while ((null == node || isBlack(node)) && (node != this.rootNode) ){
        // node 是父节点的左孩子
        if (node == parent.left){
            // 获取到其右孩子
            other = parent.right;
            // node 节点的兄弟节点是红色
            if (isRed(other)){
                setColorBlack(other);
                setColorRed(parent);
                leftRotate(parent);
                other = parent.right;
            }
            // node 节点的兄弟节点是黑色，且兄弟节点的两个孩子节点也是黑色
            if ((other.left == null || isBlack(other.left)) &&
                (other.right == null || isBlack(other.right))){
                setColorRed(other);
                node = parent;
                parent = getParent(node);
            } else {
                // node 节点的兄弟节点是黑色，且兄弟节点的右孩子是红色
                if (null == other.right || isBlack(other.right)){
                    setColorBlack(other.left);
                    setColorRed(other);
                    rightRotate(other);
                    other = parent.right;
                }
                // node 节点的兄弟节点是黑色，且兄弟节点的右孩子是红色，左孩子是任意颜色
                setColor(other, getColor(parent));
                setColorBlack(parent);
                setColorBlack(other.right);
                leftRotate(parent);
                node = this.rootNode;
                break;
            }
        } else {
            other = parent.left;
            if (isRed(other)){
                setColorBlack(other);
                setColorRed(parent);
                rightRotate(parent);
                other = parent.left;
            }
            if ((null == other.left || isBlack(other.left)) &&
                (null == other.right || isBlack(other.right))){
                setColorRed(other);
                node = parent;
                parent = getParent(node);
            }
        }
    }
}

```

```

        } else {
            if (null == other.left || isBlack(other.left)){
                setColorBlack(other.right);
                setColorRed(other);
                leftRotate(other);
                other = parent.left;
            }
            setColor(other,getColor(parent));
            setColorBlack(parent);
            setColorBlack(other.left);
            rightRotate(parent);
            node = this.rootNode;
            break;
        }
    }
    if (node!=null)
        setColorBlack(node);
}

```

1.5 性能

- (1) 查找代价:
 - 由于红黑树的性质 (最长路径长度不超过最短路径长度的 2 倍)，可以说明红黑树虽然不像 AVL 一样是严格平衡的，但平衡性能还是要比 BST 要好。其查找代价基本维持在 $O(\log N)$ 左右，但在最差情况下 (最长路径是最短路径的 2 倍少 1)，比 AVL 要略逊色一点。
- (2) 插入代价:
 - RBT 插入结点时，需要旋转操作和变色操作。但由于只需要保证 RBT 基本平衡就可以了。因此插入结点最多只需要 2 次旋转，这一点和 AVL 的插入操作一样。虽然变色操作需要 $O(\log N)$ ，但是变色操作十分简单，代价很小。
- (3) 删除代价:
 - RBT 的删除操作代价要比 AVL 要好的多，删除一个结点最多只需要 3 次旋转操作。
 - 从根到叶子节点的最大路径不能大于最短路径的两倍长，大致上是平衡的，插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例。
- 如果查找、插入、删除频率差不多，那么选择红黑树。

1.6 源码：java

1.6.1 红黑树的结点用 Java 表示数据结构：

```

private static final boolean RED = true;
private static final boolean BLACK = false;
private Node root; // 二叉查找树的根节点

// 结点数据结构
private class Node {
    private Key key; // 键
    private Value value;// 值
    private Node left, right; // 指向子树的链接：左子树和右子树.
    private int N; // 以该节点为根的子树中的结点总数
    boolean color; // 由其父结点指向它的链接的颜色也就是结点颜色.

    public Node(Key key, Value value, int N, boolean color) {
        this.key = key;
        this.value = value;
        this.N = N;
        this.color = color;
    }
}

```

```

}

// 获取整个二叉查找树的大小
public int size(){
    return size(root);
}

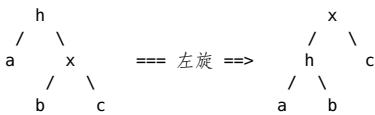
// 获取某一个结点为根结点的二叉查找树的大小
private int size(Node x){
    if (x == null) return 0;
    else return x.N;
}

private boolean isRed(Node x){
    if (x == null) return false;
    return x.color == RED;
}

```

1.6.2 左旋的 Java 实现如下

- 假设的节点如下：



- 代码如下：

```

// 左旋转
private Node rotateLeft(Node h){
    Node x = h.right;
    // 把 x 的左结点赋值给 h 的右结点
    h.right = x.left;
    // 把 h 赋值给 x 的左结点
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1+ size(h.left) + size(h.right);

    return x;
}

```

1.6.3 右旋其实就是左旋的逆操作

- 假设的节点如下：



- 代码如下：

```

// 右旋转
private Node rotateRight(Node h){
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1+ size(h.left) + size(h.right);
    return x;
}

```

2 java.util.concurrent.ArrayBlockingQueue 原理简要分析

- 昨天再网上扫了扫这些原理，感觉基本上是看懂了，ReentrantLock 能够想像出它是如何通过类似 Semaphore 之类的锁原理来达到生产者消费者之间的公平锁，但是如果想要不公平的 ArrayBlockingQueue，我目前还是自己想不出该如何实现的，改天需要所这些锁的基础也再复习巩固一下。
- 不想占用相对宝贵的时间，等改天四周周五周尾脑袋不太清醒的时候再来总结这些文档来提神吧。这两天暂时在心里在概念理论上一再巩固自己的理解就好了。文档改天再补。

2.1 引言

- ArrayBlockingQueue 是一个基于静态数组的阻塞队列，可用于实现生产-消费者模型。

2.1.1 它和 LinkedBlockingQueue 存在以下几个不同点：

1. 1、锁的实现不同

- ArrayBlockingQueue 的入队和出队都是使用的一个锁，意味着只能有一个线程来修改它。
- LinkedBlockingQueue 采用了两把锁：入队锁和出队锁，可以有一个线程负责生产，一个线程负责消费而不阻塞。

2. 2、内部保存对象的方式不同

- ArrayBlockingQueue 在加入元素时，直接将元素添加到数组。
- LinkedBlockingQueue 在加入元素时，需要把对象封装成内部类 Node 并拼接到链表尾部。

3. 3、构造阶段

- ArrayBlockingQueue 在构造阶段必须指定队列最大长度
- LinkedBlockingQueue 在构造阶段无须指定最大长度(默认最大长度为 Integer.MAX_VALUE)

4. 4、锁的公平

- ArrayBlockingQueue 可以实现公平锁，而 LinkedBlockingQueue 则只能使用非公平锁

2.2 原理分析

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    //存储元素的数组
    final Object[] items;
    //下一个出队元素的数组下标
    int takeIndex;
    //下一个入队元素的数组下标
    int putIndex;
    //元素数量
    int count;
    //锁
    final ReentrantLock lock;
    //用来等待、通知尝试获取元素的线程
    private final Condition notEmpty;
    //用来等待、通知尝试添加元素的线程
    private final Condition notFull;
    //迭代器和这个队列更新数据的中间体
    transient Itrs itrs = null;
}
```

- ArrayBlockingQueue 在内部实现了一个静态数组来存储元素，并通过 takeIndex 和 putIndex 来实现元素的快速入队出队。在并发方面，ArrayBlockingQueue 使用了一个重入锁来保证并发安全性，和 LinkedBlockingQueue 一样采用两个 Condition 用来通知入队出队线程。

2.2.1 1、构造方法

- ArrayBlockingQueue 提供了三种 public 构造方法：

构造方法	解释
ArrayBlockingQueue(int capacity)	构造一个最大大小为 capacity 的阻塞队列
ArrayBlockingQueue(int capacity, boolean fair)	同上，若 fair 为 true 则使用公平锁
ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)	同上，构造时默认将集合 c 中所有元素加入到队列中

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    //分配一个 capacity 长度的数组
    this.items = new Object[capacity];
    //创建一个重入锁
    lock = new ReentrantLock(fair);
    //获取这个锁对应的 Condition
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}
```

- 这个构造方法比较简单，主要是完成实例变量的赋值操作

```
public ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c) {
    this(capacity, fair);
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        int i = 0;
        try {
            for (E e : c) {
                checkNotNull(e);
                items[i++] = e;
            }
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new IllegalArgumentException();
        }
        //更新元素数量计数器
        count = i;
        //更新出队指针
        putIndex = (i == capacity) ? 0 : i;
    } finally {
        lock.unlock();
    }
}
```

- 上述构造方法在完成变量的赋值操作后，还会将集合 c 中所有元素加入到队列中。但需要注意的是：1、集合不能有 null 元素，否则会抛出 NullPointerException。2、集合元素的数量不能超过这个队列的最大长度，否则会抛出 IllegalArgumentException。

2.2.2 2、添加元素

- ArrayBlockingQueue 提供了一下 API 来添加元素：

方法	作用
boolean add(E e)	尝试调用 offer 添加元素，添加失败抛出 IllegalStateException
boolean offer(E e)	无阻塞地添加元素，如果队列已满则直接返回 false
boolean offer(E e, long timeout, TimeUnit unit)	阻塞地添加元素，如果队列已满但最多等待 timeout，则返回 false
void put(E e)	阻塞地添加元素，如果队列已满会阻塞到被 interrupt

- 其中，put 方法和 offer(E,long,TimeUnit) 在阻塞过程中可被 interrupt。

1. put 方法分析

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    //加锁，保证线程安全
    lock.lockInterruptibly();
    try {
        //当队列已满时会调用 await 使当前线程等待
        while (count == items.length)
            notFull.await();
        //入队
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

- ArrayBlockingQueue 采用内部方法 enqueue 来完成入队操作：

```
private void enqueue(E x) {
    final Object[] items = this.items;
    //将元素 x 放入 putIndex 位置
    items[putIndex] = x;
    //增加入队下标，若等于入队长度则从 0 开始
    if (++putIndex == items.length)
        putIndex = 0;
    //增加数组元素
    count++;
    //激活一个等待获取元素的线程
    notEmpty.signal();
}
```

- enqueue 方法直接将元素插入到数组的 putIndex 位置，并将 putIndex 加 1（或设为 0），然后激活一个等待元素的线程。

2.2.3 3、获取元素

- ArrayBlockingQueue 提供了一下 API 来获取元素：

方法	作用
E poll()	获取元素并删除队首元素（出队）
E take()	获取元素并删除队首元素（出队），若队列没有元素则阻塞
E poll(long timeout, TimeUnit unit)	获取元素并删除队首元素（出队），若队列没有元素则至多等待 time
E peek()	获取队首元素，如果队列为空返回 null

1. take 方法分析

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        //如果队列为空则让当前线程等待
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

- ArrayBlockingQueue 采用内部方法 enqueue 来完成出队操作：

```

private E dequeue() {
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    //根据 takeIndex 获取元素
    E x = (E) items[takeIndex];
    //删除数组中的 takeIndex 位置的元素
    items[takeIndex] = null;
    //takeIndex 下标加 1
    if (++takeIndex == items.length)
        takeIndex = 0;
    //元素数量计数器减 1
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    //激活一个等待入队的线程
    notFull.signal();
    return x;
}

```

3 并发容器简介

并发容器	对应的普通容器	描述
ConcurrentHashMap	HashMap	Java 1.8 之前采用分段锁机制细化锁粒度，降低阻塞，从而 Java 1.8 之后基于 CAS 实现。
ConcurrentSkipListMap	SortedMap	基于跳表实现的
CopyOnWriteArrayList	ArrayList	基于 CopyOnWriteArrayList 实现。
CopyOnWriteArraySet	Set	基于 ConcurrentSkipListMap 实现。
ConcurrentSkipListSet	SortedSet	线程安全的无界队列。底层采用单链表。支持 FIFO。
ConcurrentLinkedQueue	Queue	线程安全的无界双端队列。底层采用双向链表。支持 FIFO。
ConcurrentLinkedDeque	Deque	数组实现的阻塞队列。
ArrayBlockingQueue	Queue	链表实现的阻塞队列。
LinkedBlockingQueue	Queue	双向链表实现的双端阻塞队列。
LinkedBlockingDeque	Deque	

- **Concurrent***:

- 这类型的锁竞争相对于 CopyOnWrite* 要高一些，但写操作代价要小一些。
- 此外，Concurrent* 往往提供了较低的遍历一致性，即：当利用迭代器遍历时，如果容器发生修改，迭代器仍然可以继续进行遍历。代价就是，在获取容器大小 size()，容器是否为空等方法，不一定完全精确，但这是为了获取并发吞吐量的设计取舍，可以理解。与之相比，如果是使用同步容器，就会出现 fail-fast 问题，即：检测到容器在遍历过程中发生了修改，则抛出 ConcurrentModificationException，不再继续遍历。
- *CopyOnWrite**: 一个线程写，多个线程读。读操作时不加锁，写操作时通过在副本上加锁保证并发安全，空间开销较大。
- *Blocking**: 内部实现一般是基于锁，提供阻塞队列的能力。

3.1 1.1 并发场景下的 Map

- 如果对数据有强一致要求，则需使用 Hashtable；在大部分场景通常都是弱一致性的情况下，使用 ConcurrentHashMap 即可；如果数据量在千万级别，且存在大量增删改操作，则可以考虑使用 ConcurrentSkipListMap

3.2 1.2 并发场景下的 List

- 读多写少用 CopyOnWriteArrayList。
- 写多读少用 ConcurrentLinkedQueue，但由于是无界的，要有容量限制，避免无限膨胀，导致内存溢出。

4 Map

4.1 HashMap

- HashMap 是一个线程不安全的类，不能在多线程下使用
- JDK1.7 结构：数组 + 链表（采用拉链法）
- JDK1.8 结构：数组 + 链表/红黑树（链表长度要大于阈值 8）

4.2 ConcurrentHashMap

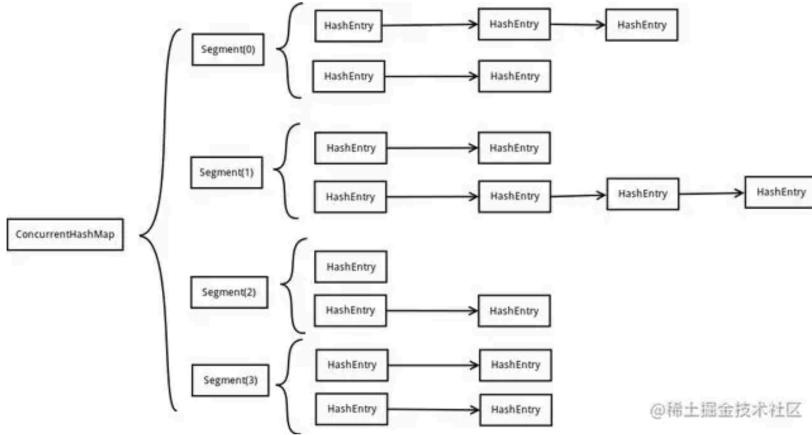
- Map 接口的两个实现是 **ConcurrentHashMap** 和 **ConcurrentSkipListMap**；
- 它们从应用的角度来看，主要区别在于 ConcurrentHashMap 的 key 是无序的，而 ConcurrentSkipListMap 的 key 是有序的。所以如果你需要保证 key 的顺序，就只能使用 ConcurrentSkipListMap。
- 使用 ConcurrentHashMap 和 ConcurrentSkipListMap 需要注意的地方是，它们的 key 和 value 都不能为空，否则会抛出 NullPointerException 这个运行时异常。

4.3 ConcurrentHashMap 的原理

- ConcurrentHashMap 是线程安全的 HashMap，用于替代 Hashtable。

4.3.1 Java 1.7

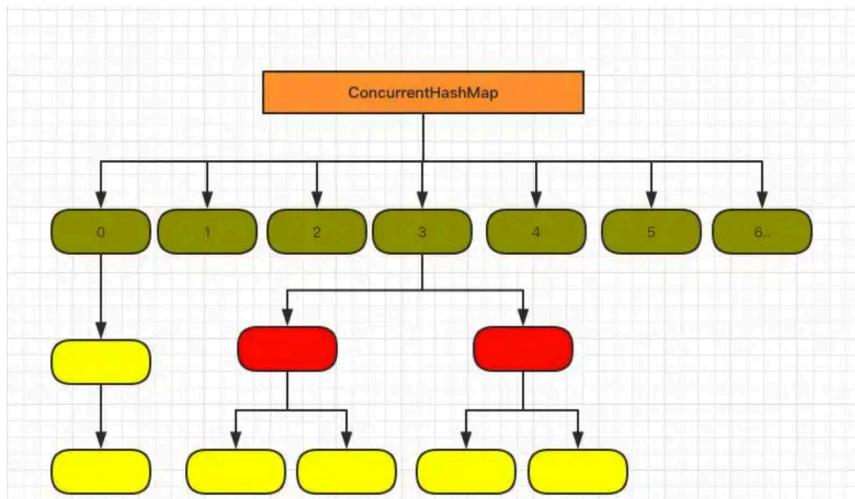
- JDK7 中，ConcurrentHashMap 最外层是多个 segment，每个 segment 的底层数据结构与 HashMap 类似，仍然是数组 + 链表组成的拉链法
- 每个 Segment 独立上 ReentrantLock 锁，每个 Segment 之间互不影响，提高了并发效率（Segment 继承自 ReentrantLock）
- ConcurrentHashMap 默认有 16 个 segment，所以最多支持 16 个线程并发写（操作在不同的 segment 上时）。默认值在初始化的时候可以指定，但是一旦初始化过后，就不可以扩容。但是每个 Segment 内部是可以扩容的



- 数据结构：数组 + 单链表
- 并发机制：采用分段锁机制细化锁粒度，降低阻塞，从而提高并发性。
- 分段锁，是将内部进行分段（Segment），里面是 HashEntry 数组，和 HashMap 类似，哈希相同的条目也是以链表形式存放。HashEntry 内部使用 volatile 的 value 字段来保证可见性，也利用了不可变对象的机制，以改进利用 Unsafe 提供的底层能力，比如 volatile access，去直接完成部分操作，以最优化性能，毕竟 Unsafe 中的很多操作都是 JVM intrinsic 优化过的。

4.3.2 Java 1.8

- 根本没有借鉴 JDK1.7，而是重写了一遍。。。
- JDK1.8 中的 ConcurrentHashMap 结构和 1.8 中的 HashMap 结构是相似的，也是数组 + 链表/红黑树（阈值也是 8，不过还要满足 table.length >= MIN_TREEIFY_CAPACITY 这个值是 64）



- 数据结构：数组 + 单链表 + 红黑树
- 并发机制：取消分段锁，之后基于 CAS + synchronized 实现。
- 数据结构改进：与 HashMap 一样，将原先数组 + 单链表的数据结构，变更为数组 + 单链表 + 红黑树的结构。当出现哈希冲突时，数据会存入数组指定桶的单链表，当链表长度达到 8，则

将其转换为红黑树结构，这样其查询的时间复杂度可以降低到 $O(\log N)$ ，以改进性能（因为链表的查询性能较差，改成红黑树查询效率更高）

- 并发机制改进：

- 取消 segments 字段，直接采用 transient volatile HashEntry<K,V>[] table 保存数据，采用 table 数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。
- 使用 CAS + synchronized 操作，在特定场景进行无锁并发操作。使用 Unsafe、LongAdder 之类底层手段，进行极端情况的优化。现代 JDK 中，synchronized 已经被不断优化，可以不再过分担心性能差异，另外，相比于 ReentrantLock，它可以减少内存消耗，这是一个非常大的优势。

- put 方法：现在不怎么能看得懂，希望改天能够把他们看懂

```
/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // key-value 的值不能为空
    if (key == null || value == null) throw new NullPointerException();
    // 计算 hash
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // table 如果为空，或者长度为零就执行初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 找出节点需要放置的位置如果为空，然后用 CAS 来赋值
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
                break; // no lock when adding to empty bin
        }
        // 如果处于 MOVED 状态 就帮助转换
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        // 如果 table 上要放的位置不为空就执行下列操作
        else {
            V oldVal = null;
            // 锁住当前 table 上的位置
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            //key 相同就替换
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            // 找不到相同的就插入到最尾部
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                               value, null);
                                break;
                            }
                        }
                    }
                    // 如果数组下方的链式结构是红黑树 就按红黑树处理放置
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if (((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                               value)) != null)
                            oldVal = p.val;
                }
            }
        }
    }
}
```

```

        if (!onlyIfAbsent)
            p.val = value;
    }
}
}
// 检查是否满足阈值
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        // 满足时就把链表转成红黑树
        // 注意此方法里面还有一个判断 tab.length 小于 64 的不转化
        treeifyBin(tab, i);
    // 如果老值不为空就返回
    if (oldVal != null)
        return oldVal;
    break;
}
}
addCount(lL, binCount);
return null;
}

```

- get 方法

```

public V get(Object key) {
    Node<K,V>[] tab;
    Node<K,V> e, p;
    int n, eh;
    K ek;
    // 计算 hash 值
    int h = spread(key.hashCode());
    // 排除为空的情况，并找到对应位置
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 如果相等就直接在 table 上取值
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        // 在红黑树中找值
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        // 在链表中找值
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}

```

4.3.3 对比 JDK1.7 与 1.8

- 首先是数据结构上：
 - 1.7 是 segment 数组，+ Segment (类似 HashMap 的结构)
 - 1.8 是数据 + 链表/红黑树与 HashMap 类似
- 并发上：
 - 1.7 是使用 ReentrantLock 锁住每个 Segment
 - 1.8 是使用 CAS + synchronized
- 为什么超过 8 要使用红黑树
 - 首先 * 链表的结构存储要比红黑树存储节省空间 *

- 而 * 链表在查询上又没有红黑树块 *
- 这个时候就需要一个边界，源码作者做了一个泊松分布运算，在链表达到 8 时的概率已经非常小了。而链表长度为 8 时，查找费时也不大。概率只有千万分之几 (这里没有看懂)

4.3.4 线程安全问题

- ConcurrentHashMap 并发下单独操作的确是安全的，但是组合操作就未必了。所以如果在多线程情况下，有多步操作 ConcurrentHashMap 的时候需要额外留心
 - 如：如果要修改一个值：可以使用 boolean replace(key, oldValue, newValue) 来修改，而不是先 get 然后 put，这个方法类似于 CAS 的思想
 - 此外还有 putIfAbsent(key, value)，先判断有没有这个值，如果没有就 put，有就取出来给你

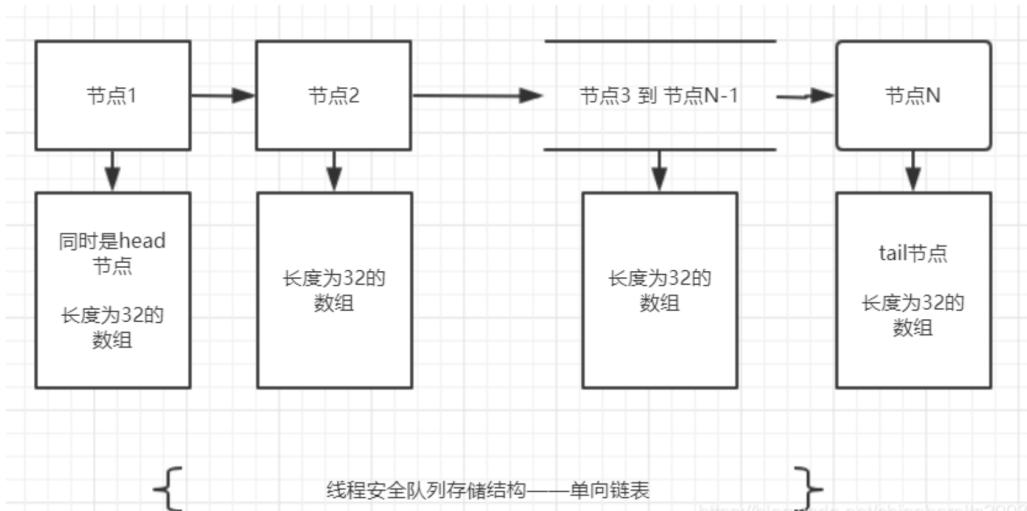
5 List

- 先前总结的比较繁杂，感觉很多的同步数据结构，可能并没有真正理解。现在需要把它们理解得再为深刻彻底一点儿
- 现在我们看下线程安全队列的实现方式：(参考自：.net framework 4.x)，核心代码全部做了注释【这么看的话，感觉对底层结构就熟悉很多】

总的来说，（总结语放到前面，防止代码篇幅太大，同志们没有耐心翻到最底下~）

1、线程安全队列通过SpinWait自旋类来实现等待并行线程完成与Interlocked原子操作类计数实现的。

2、线程安全队列通过单向链表实现的，链的节点为长度32的数组，通过记录链的头节点与尾节点、以及队列的头尾实现队列的存储与入队、出队操作的。



```
public class MyConcurrentQueue<T> : IProducerConsumerCollection<T> {
    [NonSerialized]
    private volatile Segment m_head;
    [NonSerialized]
    private volatile Segment m_tail;

    private T[] m_serializationArray;
    private const int SEGMENT_SIZE = 32; // 每个节点，每个数组有 32 个元素
    [NonSerialized]
```

```

internal volatile int m_numSnapshotTakers = 0;

// 链尾部节点
public MyConcurrentQueue() {
    m_head = m_tail = new Segment(0, this);
}
// 尝试添加
bool IProducerConsumerCollection<T>.TryAdd(T item) {
    Enqueue(item);
    return true;
}
// 尝试从中移除并返回对象
bool IProducerConsumerCollection<T>.TryTake(out T item) {
    return TryDequeue(out item);
}
// 判断当前链是否为空
public bool IsEmpty {
    get {
        Segment head = m_head;
        if (!head.IsEmpty)
            // 如果头不为空，则链非空
            return false;
        else if (head.Next == null)
            // 如果头节点的下一个节点为空，且为链尾,
            return true;
        else {
            // 如果头节点为空且不是最后一个节点，则标识另一个线程正在写入该数组
            // 等待中..
            SpinWait spin = new SpinWait();
            while (head.IsEmpty) {
                // 此时为空
                if (head.Next == null)
                    return true;
                // 否则标识正在有线程占用写入
                // 线程循环一次
                spin.SpinOnce();
                head = m_head;
            }
            return false;
        }
    }
}
// 用来判断链是否在变化
private void GetHeadTailPositions(out Segment head, out Segment tail,
                                   out int headLow, out int tailHigh) {
    head = m_head;
    tail = m_tail;
    headLow = head.Low;
    tailHigh = tail.High;
    SpinWait spin = new SpinWait();
    Console.WriteLine($"head.Low:{head.Low},tail.High: {tail.High}, head.m_index:{head.m_index}, tail.m_index: {tail.m_index}");
    // 通过循环来保证值不再更改（也就是说并行线程操作结束）
    // 保证线程串行核心的判断逻辑
    while (
        // 头尾发生变化
        head != m_head || tail != m_tail
        // 如果队列头、尾索引发生变化
        || headLow != head.Low || tailHigh != tail.High
        || head.m_index > tail.m_index) {
        spin.SpinOnce();
        head = m_head;
        tail = m_tail;
        headLow = head.Low;
        tailHigh = tail.High;
    }
}
// 获取总数
public int Count {
    get {
        Segment head, tail;
        int headLow, tailHigh;
        GetHeadTailPositions(out head, out tail, out headLow, out tailHigh);
        if (head == tail) {
            return tailHigh - headLow + 1;
        }
    }
}

```

```

// 头节点长度
int count = SEGMENT_SIZE - headLow;
// 加上中间其他节点长度
count += SEGMENT_SIZE * ((int)(tail.m_index - head.m_index - 1));
// 加上尾节点长度
count += tailHigh + 1;
return count;
}
}

public object SyncRoot => throw new NotImplementedException();
public bool IsSynchronized => throw new NotImplementedException();
public void CopyTo(T[] array, int index) { }

// 暂未实现
public IEnumerator<T> GetEnumerator() {
    return null;
}

// 添加
public void Enqueue(T item) {
    SpinWait spin = new SpinWait();
    while (true) {
        Segment tail = m_tail;
        if (tail.TryAppend(item))
            return;
        spin.SpinOnce();
    }
}

// 尝试删除节点
public bool TryDequeue(out T result) {
    while (!IsEmpty) {
        Segment head = m_head;
        if (head.TryRemove(out result))
            return true;
    }
    result = default(T);
    return false;
}

// 查看最后一个添加入的元素
public bool TryPeek(out T result) {
    // 原子增加值
    Interlocked.Increment(ref m_numSnapshotTakers);
    while (!IsEmpty) {
        // 首先从头节点看一下第一个节点是否存在
        Segment head = m_head;
        if (head.TryPeek(out result)) {
            Interlocked.Decrement(ref m_numSnapshotTakers);
            return true;
        }
    }
    result = default(T);
    Interlocked.Decrement(ref m_numSnapshotTakers);
    return false;
}

public void CopyTo(Array array, int index) {
    throw new NotImplementedException();
}

IEnumerator IEnumerable.GetEnumerator() {
    throw new NotImplementedException();
}

public T[] ToArray() {
    throw new NotImplementedException();
}

// 为线程安全队列提供一个 单向链表,
// 链表的每个节点存储长度为 32 的数组
private class Segment {
    // 定义一个数组, 用于存储每个节点的内容
    internal volatile T[] m_array;
    // 定义一个结构数组, 用于标识数组中每个节点是否有效 (是否存储内容)
    internal volatile VolatileBool[] m_state;
    // 指针, 指向下一个节点数组
    // 如果是最后一个节点, 则节点为空
    private volatile Segment m_next;
    // 索引, 用来存储链表的长度
    internal readonly long m_index;
    // 用来标识队列头-数组弹出索引
    private volatile int m_low;
}

```

```

// 用来标识队列尾-数组最新存储位置
private volatile int m_high;
// 用来标识队列
private volatile MyConcurrentQueue<T> m_source;
// 实例化链节点
internal Segment(long index, MyConcurrentQueue<T> source) {
    m_array = new T[SEGMENT_SIZE];
    m_state = new VolatileBool[SEGMENT_SIZE]; // all initialized to false
    m_high = -1;
    m_index = index;
    m_source = source;
}
// 链表的下一个节点
internal Segment Next {
    get { return m_next; }
}
// 如果当前节点数组为空返回 true,
internal bool IsEmpty {
    get { return (Low > High); }
}
// 非安全添加方法 (无判断数组长度)
// <param name="value"></param>
internal void UnsafeAdd(T value) {
    m_high++;
    m_array[m_high] = value;
    m_state[m_high].m_value = true;
}

internal Segment UnsafeGrow() {
    Segment newSegment = new Segment(m_index + 1, m_source);
    m_next = newSegment;
    return newSegment;
}
// 如果当前数组满了 >=32, 则链扩展节点。
internal void Grow() {
    // 重新分配数组
    Segment newSegment = new Segment(m_index + 1, m_source);
    // 赋值给 next 指针
    m_next = newSegment;
    // 将节点添加到链
    m_source.m_tail = m_next;
}
// 在末尾添加元素
// <param name="value"> 元素 </param>
// <param name="tail">The tail.</param>
// <returns> 如果附加元素，则为 true；如果当前数组已满，则为 false</returns>
// <remarks> 如果附加指定的元素成功，并且在此之后数组满了，在链上添加新节点（节点为 32 长度数组） </remarks>
internal bool TryAppend(T value) {
    // 如果数组已满则跳出方法
    if (m_high >= SEGMENT_SIZE - 1) {
        return false;
    }
    // 局部变量初始化
    int newhigh = SEGMENT_SIZE;
    try {} finally {
        // 原子递增
        newhigh = Interlocked.Increment(ref m_high);
        if (newhigh <= SEGMENT_SIZE - 1) {
            m_array[newhigh] = value;
            m_state[newhigh].m_value = true;
        }
        // 如果数组满了，则扩展链节点。
        if (newhigh == SEGMENT_SIZE - 1) {
            Grow();
        }
    }
    // 如果 newhigh <= SEGMENT_SIZE-1, 这意味着当前线程成功地占据了一个位置
    return newhigh <= SEGMENT_SIZE - 1;
}
// 尝试从链的头部数组删除节点
// <param name="result"></param>
internal bool TryRemove(out T result) {
    SpinWait spin = new SpinWait();
    int lowLocal = Low, highLocal = High;

```

```

while (lowLocal <= highLocal) {
    // 获取队头索引
    if (Interlocked.CompareExchange(ref m_low, lowLocal + 1, lowLocal) == lowLocal) {
        // 如果要弹出队列的值不可用，说明这个位置被并行线程获取到了权限，但是值还未写入。
        // 通过线程自旋等待值写入
        SpinWait spinLocal = new SpinWait();
        while (!m_state[lowLocal].m_value) {
            spinLocal.SpinOnce();
        }
        // 取出值
        result = m_array[lowLocal];
        // 如果没有其他线程读取 (GetEnumerator ()、ToList ()) 执行删除
        // 如 TryPeek 的时候 m_numSnapshotTakers 会在进入方法体时 ++，在出方法体 --
        // 清空该索引下的值
        if (m_source.m_numSnapshotTakers <= 0)
            m_array[lowLocal] = default(T);
        // 如果说 lowLocal+1 = 32 说明当前链节点的数组已经全部出队
        if (lowLocal + 1 >= SEGMENT_SIZE) {
            // 由于 lowLocal <= highLocal 成立
            // lowLocal + 1 >= SEGMENT_SIZE 如果成立，且 m_next == null 成立，
            // 说明此时有其他线程正在做扩展链结构
            // 那么当前线程需要等待其他线程完成扩展链表，再做出队操作。
            spinLocal = new SpinWait();
            while (m.next == null) {
                spinLocal.SpinOnce();
            }
            m_source.m_head = m.next;
        }
        return true;
    }
    else {
        // 此时说明 当前线程竞争资源失败，做短暂自旋后继续竞争资源
        spin.SpinOnce();
        lowLocal = Low; highLocal = High;
    }
}
// 失败的情况下返回空值
result = default(T);
return false;
}

// 尝试获取队列头节点元素
internal bool TryPeek(out T result) {
    result = default(T);
    int lowLocal = Low;
    // 校验当前队列是否正确
    if (lowLocal > High)
        return false;
    SpinWait spin = new SpinWait();
    // 如果头节点无效，则说明当前节点被其他线程占用，并在做写入操作，
    // 需要等待其他线程写入后再执行读取操作
    while (!m_state[lowLocal].m_value) {
        spin.SpinOnce();
    }
    result = m_array[lowLocal];
    return true;
}
// 返回队列首位置
internal int Low {
    get {
        return Math.Min(m_low, SEGMENT_SIZE);
    }
}
// 获得队列长度
internal int High {
    get {
        // 如果 m_high>SEGMENT_SIZE，则表示超出范围，我们应该返回 SEGMENT_SIZE-1
        return Math.Min(m_high, SEGMENT_SIZE - 1);
    }
}
}

}

```

- <https://blog.csdn.net/chinaherolts2008/article/details/116696405>
- 也应该需要自己快速测试一下。

5.1 CopyOnWriteArrayList

- CopyOnWriteArrayList 是线程安全的 ArrayList。CopyOnWrite 字面意思为写的时候会将共享变量新复制一份出来。复制的好处在于读操作是无锁的（也就是无阻塞）。
- CopyOnWriteArrayList 仅适用于写操作非常少的场景，而且能够容忍读写的短暂不一致。如果读写比例均衡或者有大量写操作的话，使用 CopyOnWriteArrayList 的性能会非常糟糕。

5.1.1 CopyOnWriteArrayList 原理

- CopyOnWriteArrayList 内部维护了一个数组，成员变量 array 就指向这个内部数组，所有的读操作都是基于 array 进行的，如下图所示，迭代器 Iterator 遍历的就是 array 数组。

- lock - 执行写时复制操作，需要使用可重入锁加锁
- array - 对象数组，用于存放元素

```
/** The lock protecting all mutators */
final transient ReentrantLock lock = new ReentrantLock();
/** The array, accessed only via getArray/setArray. */
private transient volatile Object[] array;
```

1. 读操作

- 在 CopyOnWriteArrayList 中，读操作不同步，因为它们在内部数组的快照上工作，所以多个迭代器可以同时遍历而不会相互阻塞。
- CopyOnWriteArrayList 的读操作是不用加锁的，性能很高。

```
public E get(int index) {
    return get(getArray(), index);
}
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

2. 写操作

- 所有的写操作都是同步的。他们在备份数组的副本上工作。写操作完成后，后备阵列将被替换为复制的阵列，并释放锁定。支持数组变得易变，所以替换数组的调用是原子。
- 写操作后创建的迭代器将能够看到修改的结构。
- 写时复制集合返回的迭代器不会抛出 ConcurrentModificationException，因为它们在数组的快照上工作，并且无论后续的修改如何，都会像迭代器创建时那样完全返回元素。

(a) 添加操作：添加的逻辑很简单，先将原容器 copy 一份，然后在新副本上执行写操作，之后再切换引用。当然此过程是要加锁的。

```
public boolean add(E e) {
    //ReentrantLock 加锁，保证线程安全
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        //拷贝原容器，长度为原容器长度加一
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        //在新副本上执行添加操作
        newElements[len] = e;
        //将原容器引用指向新副本
        setArray(newElements);
        return true;
    } finally {
        //解锁
        lock.unlock();
    }
}
```

(b) 删除操作：删除操作同理，将除要删除元素之外的其他元素拷贝到新副本中，然后切换引用，将原容器引用指向新副本。同属写操作，需要加锁。

```
public E remove(int index) {
    //加锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        E oldValue = get(elements, index);
        int numMoved = len - index - 1;
        if (numMoved == 0)
            //如果要删除的是列表末端数据，拷贝前 len-1 个数据到新副本上，再切换引用
            setArray(Arrays.copyOf(elements, len - 1));
        else {
            //否则，将除要删除元素之外的其他元素拷贝到新副本中，并切换引用
            Object[] newElements = new Object[len - 1];
            System.arraycopy(elements, 0, newElements, 0, index);
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            setArray(newElements);
        }
        return oldValue;
    } finally {
        //解锁
        lock.unlock();
    }
}
```

- <https://blog.csdn.net/abc123lzf/article/details/82702123>
- <https://juejin.cn/post/6844903602444582920>
- <https://juejin.cn/post/6844903602448760845>
- <https://juejin.cn/post/6844903602423595015>
- [https://kkewei.github.io/elasticsearch_learning/2017/10/02/Condition%
E5%8E%9F%E7%90%86%E8%A7%A3%E8%AF%BB/](https://kkewei.github.io/elasticsearch_learning/2017/10/02/Condition%E5%8E%9F%E7%90%86%E8%A7%A3%E8%AF%BB/)
- https://blog.csdn.net/mocas_wang/article/details/108476505
- https://blog.csdn.net/weixin_54499878/article/details/117924412