jdk源码解析-TreeMap红黑树

讲师简介

smart哥，互联网悍将，历经从传统软件公司到大型互联网公司的洗礼，入行即在中兴通讯等大型通信公司担任项目leader，后随着互联网的崛起，先后在美团支付等大型互联网公司担任架构师，公派旅美期间曾与并发包大神Doug Lea探讨java多线程等最底层的核心技术。对互联网架构底层技术有相当的研究和独特的见解，在多个领域有着丰富的实战经验。

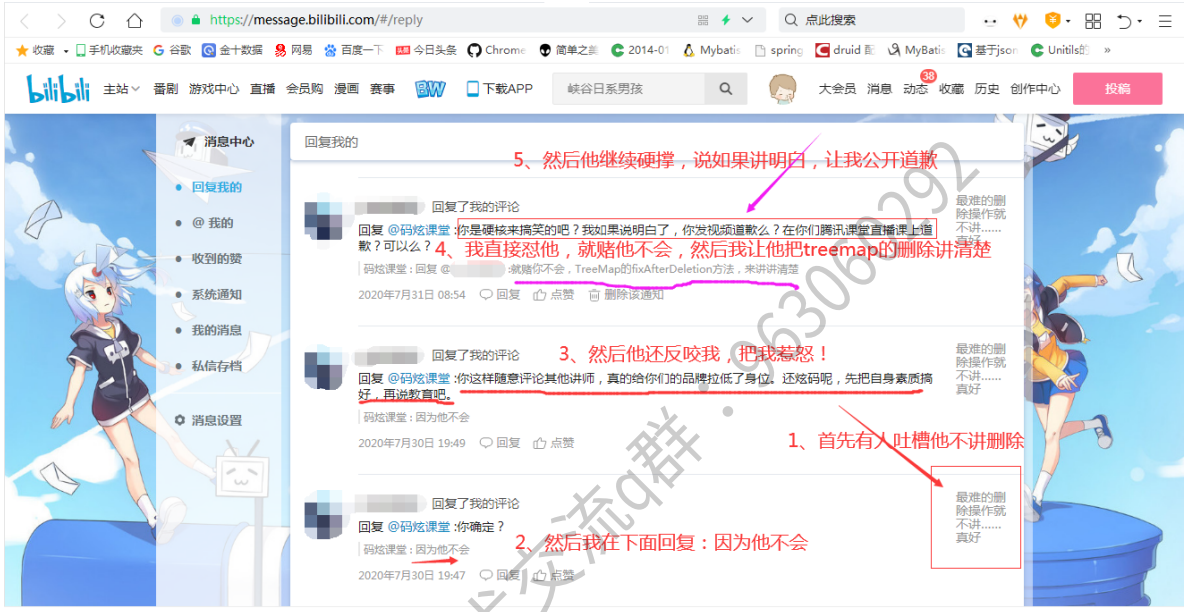
一、前言

红黑树是数据结构中比较复杂的一种，找遍b站几乎没有人能讲清楚，看了很多所谓牛逼的大佬的课程，几乎上来就是红黑树5大性质一丢，再搞个旋转动图，然后开始讲节点插入操作，貌似很牛逼，但是我从没看见几个讲节点删除操作的，更别谈说能讲的透彻明白的，而且就插入节点这种情况而言，它是红黑树操作中相对简单点的，一般照着给定的规则也能看懂源码，但是很多人今天听了貌似懂了，但

是明天就忘记了。

这就很奇怪了，不是听懂了吗？怎么还忘记了？说到底就是讲的人没讲明白，让你死记硬背着色、旋转规则（其实连他自己都不知道为什么要定这样的规则），那听的人自然也就没听明白了，学红黑树不是学语文，靠死记硬背规则。所以那些误人子弟的垃圾讲师他只是告诉你红黑树的变色规则和旋转规则，根本讲不出来为什么要变色？为什么要旋转？说到底就是因为他们自己也不会，所以讲不出来，只会照本宣科，按照规则给你讲一遍，让你们记住规则就行了，这样的结果肯定就是今天听了明天保证忘记。

前两天我怼了b站上一个讲源码的垃圾讲师，他只讲插入，删除不讲，下面评论就有人吐槽为什么删除不讲？我在下面回复说，删除操作他根本不会，然后他@我，说如果他会的话让我公开道歉，我说行，那如果不会你从此退出b站，他顿时傻了。这可是打蛇打到了他的七寸，立马闭嘴，然后把我俩的对话都删除了。



这种垃圾讲师纯粹是来割韭菜的，引诱你去买他的收费课。在这里呼吁所谓"牛逼大佬们"停止这种割韭菜行为，你自己都没搞明白还出来割韭菜，良心何在？？

smart哥实在看不下去，决定把TreeMap的源码免费给大家手撕一遍(TreeMap完全是红黑树实现的，是由**Josh Bloch and Doug Lea**两位并发领域的大师共同完成的这个源码，整个TreeMap源码3000多行，代码量不大，但是TreeMap的红黑树的代码实现比红黑树发明者用到的算法更优越，https://www.cs.usfca.edu/~galles/visualization/RedBlack.html 这个国外大学的网站上大家可以自己动手操作，大家可以看下删除操作它是找的前驱节点替代原删除节点，而TreeMap源码里面**Doug Lea**他们用的是后继节点替代原删除节点，这两种方案实际操作效果是一样的，只不过树的结构不一样，但是对应的红黑树都是平衡的，这个我会在视频讲解中演示，所以大家不要觉得奇怪为什么不一样)，后面我还会免费给大家详细的讲解红黑树原理及源码。

学习本课程你将会收获如下：

- 5大性质的缘由，我会讲解为什么要有这5大性质
- 网络上红黑树的理解方式较为单一，一般是 `双黑、caseN` 法，而插入和删除的情况很多，每种都有对应的处理方式，如果死记硬背的话，再过一段时间再回忆各种情况可能就一头雾水了（**我会教你一种方法不用去死记硬背，保证能够立马记住**）。
- smart哥会讲解插入和删除的每一种情况并给出解释为什么要这么做，解释为什么要定义这样的规则（这里是关键，网上的课程基本都不会讲为什么变色？为什么要旋转？所以让你死记硬背规则，那么后果就是今天记得明天忘记，那么说到底也不能怪他们，因为他们自己也不会，所以只有糊弄小白了）

网络上讲红黑树的实现多来源于《算法导论》一书，直接讲红黑树的实现，需要处理颜色和高度两种属性约束，比较晦涩。本文通过红黑树的等同—— 2-3-4树，避开颜色属性约束，也弱化了高度的影响，以另一种方式去理解红黑树，虽然并不能完全降低它的复杂度，但这种方式更易理解。

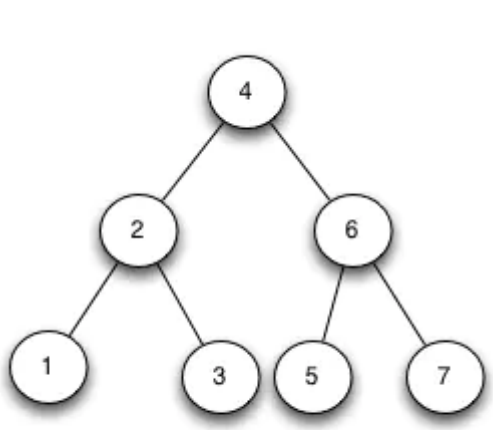
二、二叉查找树（BST）

定义

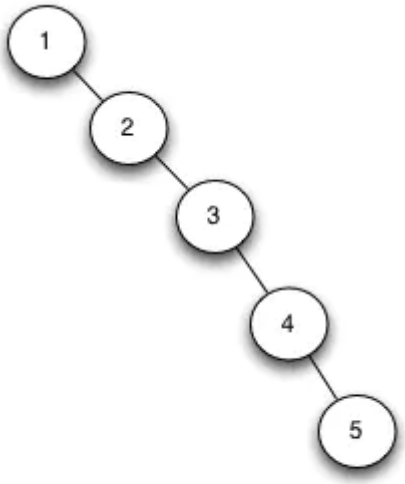
树的概念（作图讲解）

二叉树：每个子节点只有两个节点的树

二叉查找树（二叉搜索树）：就是一颗二叉树，他的左节点比父节点要小，右节点比父节点要大。他的高度决定的查找效率。



平衡二叉查找树



倾斜的二叉查找树

常见操作

查找（红黑树通用）：查找每个节点我们从根节点开始查找

- 查找值比当前值大，则搜索右子树
- 查找值等于当前值，停止查找，返回当前节点
- 查找值比当前值小，则搜索左子树

插入：要插入节点，必须先找到插入节点位置。依然是从根节点开始比较，小于根节点的话就和左子树比较，反之与右子树比较，直到左子树为空或者右子树为空，则插入到相应为空的位置。

遍历（红黑树通用）：根据一定顺序来访问整个树，常见的有前序遍历，中序遍历（用的较多），后序遍历

- 前序遍历：左子树-》根节点-》右子树　123 4 567
- 中序遍历：根节点-》左子树-》右子树　4 213 657
- 后续遍历：左子树-》右子树-》根节点　132 576 4

查找最小值（红黑树通用）：沿着根节点的左子树一路查找，直到最后一个不为空的节点，该节点就是当前这个树的最小节点

查找最大值（红黑树通用）：沿着根节点的右子树一路查找，直到最后一个不为空的节点，该节点就是当前这个树的最大节点

查找前驱节点（红黑树通用）：小于当前节点的最大值

查找后继节点（红黑树通用）：大于当前节点的最小值

删除：本质上是找前驱或者后继节点来替代

- 叶子节点直接删除（没有前驱或后继节点）
- 只有一个子节点的用子节点替代（本质上就是找的前驱节点或者是后继节点，左节点就是前驱节点，右节点就是后继节点）
- 有两个子节点的，需要找到替代节点（替代节点就是前驱节点或者后继节点）

删除操作和红黑树一样，只不过红黑树多了着色和旋转过程

三、BST存在的问题

BST存在的问题是，树在插入的时候会导致倾斜，不同的插入顺序会导致数的高度不一样，而树的高度直接影响了树的查找效率。最坏的情况所有的节点都在一条斜线上，这样树的高度为N。

基于BST存在的问题，平衡查找二叉树（Balanced BST）产生了。平衡树的插入和删除的时候，会通过旋转操作将高度保持在LogN。其中两款具有代表性的平衡术分别为AVL树（**高度平衡树**，具备二叉搜索树的全部特性，而且左右子树高度差不超过1）和红黑树。

AVL树如何实现平衡的？

通过左旋或者右旋（左旋右旋后一定不会破坏二叉搜索树的查找规则，需要演示）

面试题：有了AVL树为什么还要红黑树呢？？（10-1的序列演示）

AVL树由于实现比较复杂，而且插入和删除性能差（为什么？），在实际环境下的应用不如红黑树。

红黑树的实际应用非常广泛，如Java中的HashMap和TreeSet，Java 8中HashMap的实现因为用RBTree代替链表（链表长度>8时），性能有所提升。

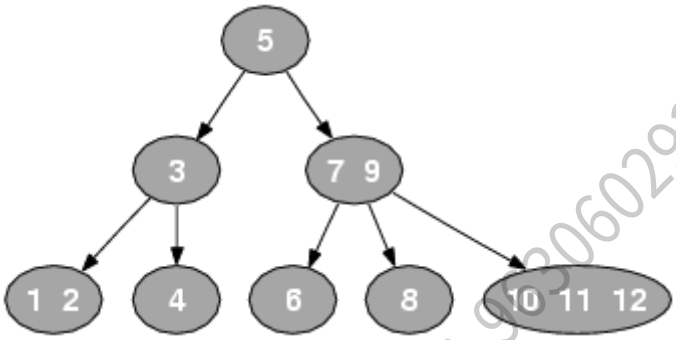
四、2-3-4树

定义

2-3-4树是四阶的 B树(Balance Tree)，他属于一种多路查找树，它的结构有以下限制：

- 所有叶子节点都拥有相同的深度。
- 节点只能是 2-节点、3-节点、4-节点之一。
    - 2-节点：包含 1 个元素的节点，有 2 个子节点；
    - 3-节点：包含 2 个元素的节点，有 3 个子节点；
    - 4-节点：包含 3 个元素的节点，有 4 个子节点；
    - 所有节点必须至少包含1个元素
- 元素始终保持排序顺序，整体上保持二叉查找树的性质，即父结点大于左子结点，小于右子结点；而且结点有多个元素时，每个元素必须大于它左边的和它的左子树中元素。

下图是一个典型的 2-3-4树（来自维基百科）：

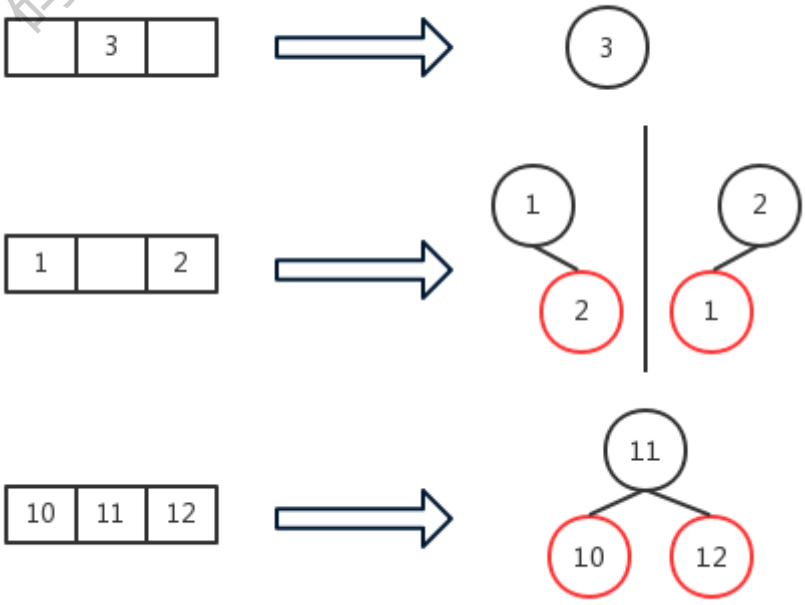


2-3-4树的查询操作像普通的二叉搜索树一样，非常简单，但由于其结点元素数不确定，在一些编程语言中实现起来并不方便，实现一般使用它的等同——红黑树。

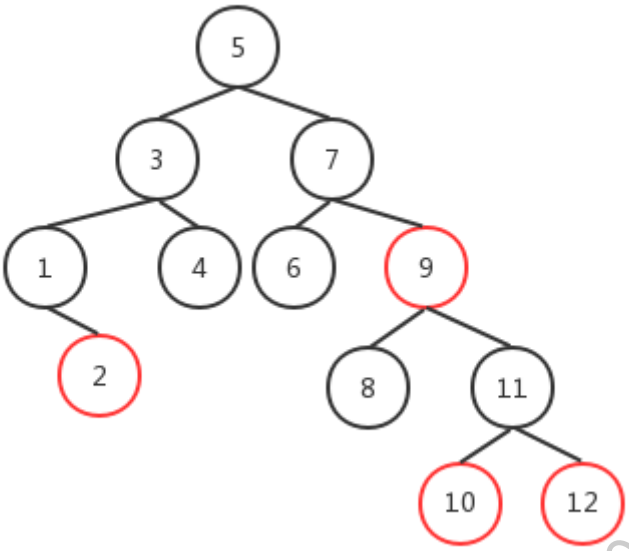
对应红黑树

就跟玩魔方一样，先给你一个结果（清一色6个面），然后你给我转，只要最后达到这个目的就行，那中间的步骤每个人都不一样，但是最后都达到了6个面，不同面都是不同的清一色。

至于为什么说红黑树是 2-3-4树的一种等同呢，这是因为 2-3-4树的每一个结点都对应红黑树的一种结构，所以每一棵 2-3-4树也都对应一棵红黑树，下图是 2-3-4树不同结点与红黑树子树的对应。



而上文中的 2-3-4树也可以转换成一棵红黑树：



由红黑树的性质5，和 2-3-4树的性质1，为了便于理解红黑树和 2-3-4树的对应关系，我们可以 `把红黑树从根结点到叶子结点的黑色结点个数定义为高度` 。

红黑树和 2-3-4树的结点添加和删除都有一个基本规则：避免子树高度变化，因为无论是 2-3-4树还是红黑树，一旦子树高度有变动，势必会影响其他子树进行调整，所以我们在插入和删除结点时尽量通过子树内部调整来达到平衡，2-3-4树实现平衡是通过结点的旋转和结点元素数变化，红黑树是通过结点旋转和变色。

下面来对照着 2-3-4树说一下红黑树结点的添加和删除：

结点插入

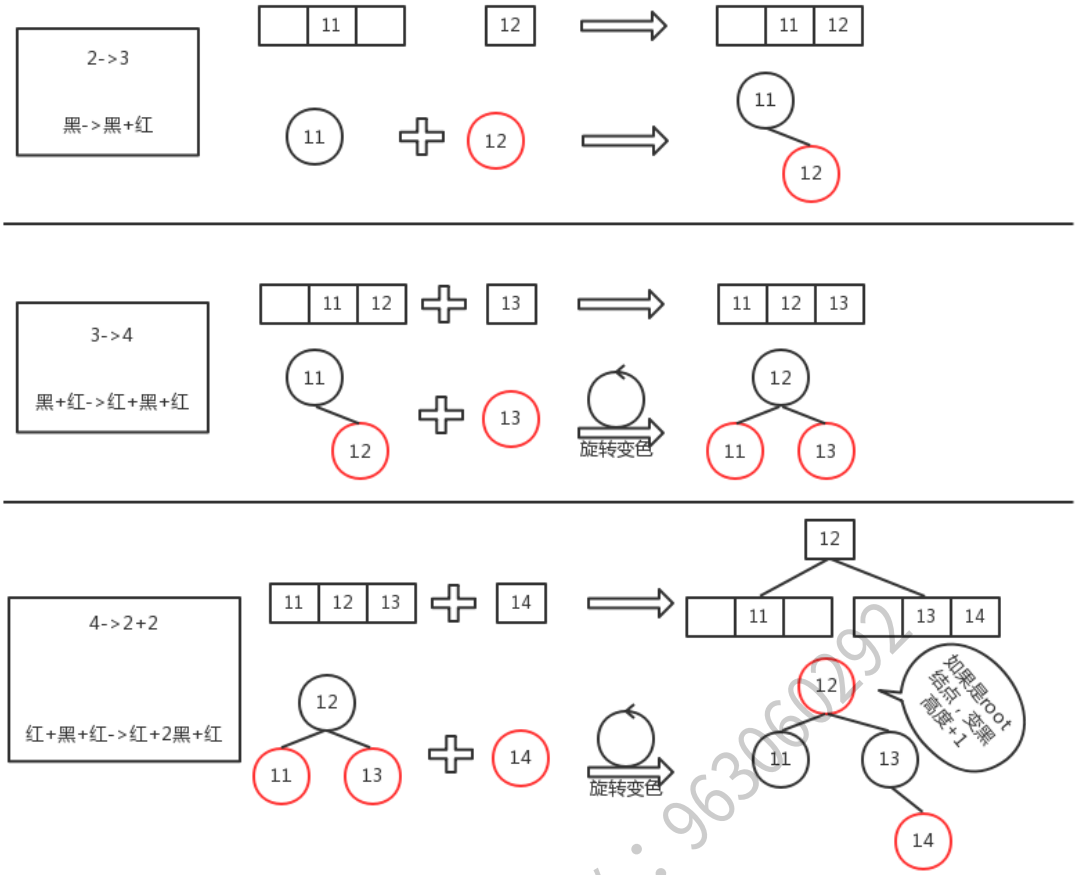
2-3-4树中结点添加需要遵守以下规则：

- 插入都是向最下面一层插入；
- 升元：将插入结点由 2-结点升级成 3-结点，或由 3-结点升级成 4-结点；
- 向 4-结点插入元素后，需要将中间元素提到父结点升元，原结点变成两个 2-结点，再把元素插入 2-结点中，如果父结点也是 4-结点，则递归向上层升元，至到根结点后将树高加1；

而将这些规则对应到红黑树里，就是：

- 新插入的结点颜色为 `红色` ，这样才可能不会对红黑树的高度产生影响。
- 2-结点对应红黑树中的单个黑色结点，插入时直接成功（对应 2-结点升元）。
- 3-结点对应红黑树中的 `黑+红` 子树，插入后将其修复成 `红+黑+红` 子树（对应 3-结点升元）；
- 4-结点对应红黑树中的 `红+黑+红` 子树，插入后将其修复成 `红色祖父+黑色父叔+红色孩子` 子树，然后再把祖父结点当成新插入的红色结点递归向上层修复，直至修复成功或遇到 root 结点；

公式：**红黑树+新增一个节点（红色）=对等的2-3-4树+新增一个节点**



如上图所示，虽然向红黑树中插入了一个新结点，但由于旋转和变色，子树的高度保持不变。

练习（一个长序列生产2-3-4树）

删除结点

2-3-4树的删除可以全部转换为叶子节点的删除，删除原则是先看能不能和下面的叶子节点合并，能合并的直接合并完后删除，不能合并的就要找个元素替换上去，最终都是要保持平衡。

合并==》删除

合并==》替换==》删除

合并==》无法替换==》再合并==》删除

红色节点一定全部都在多元素节点中

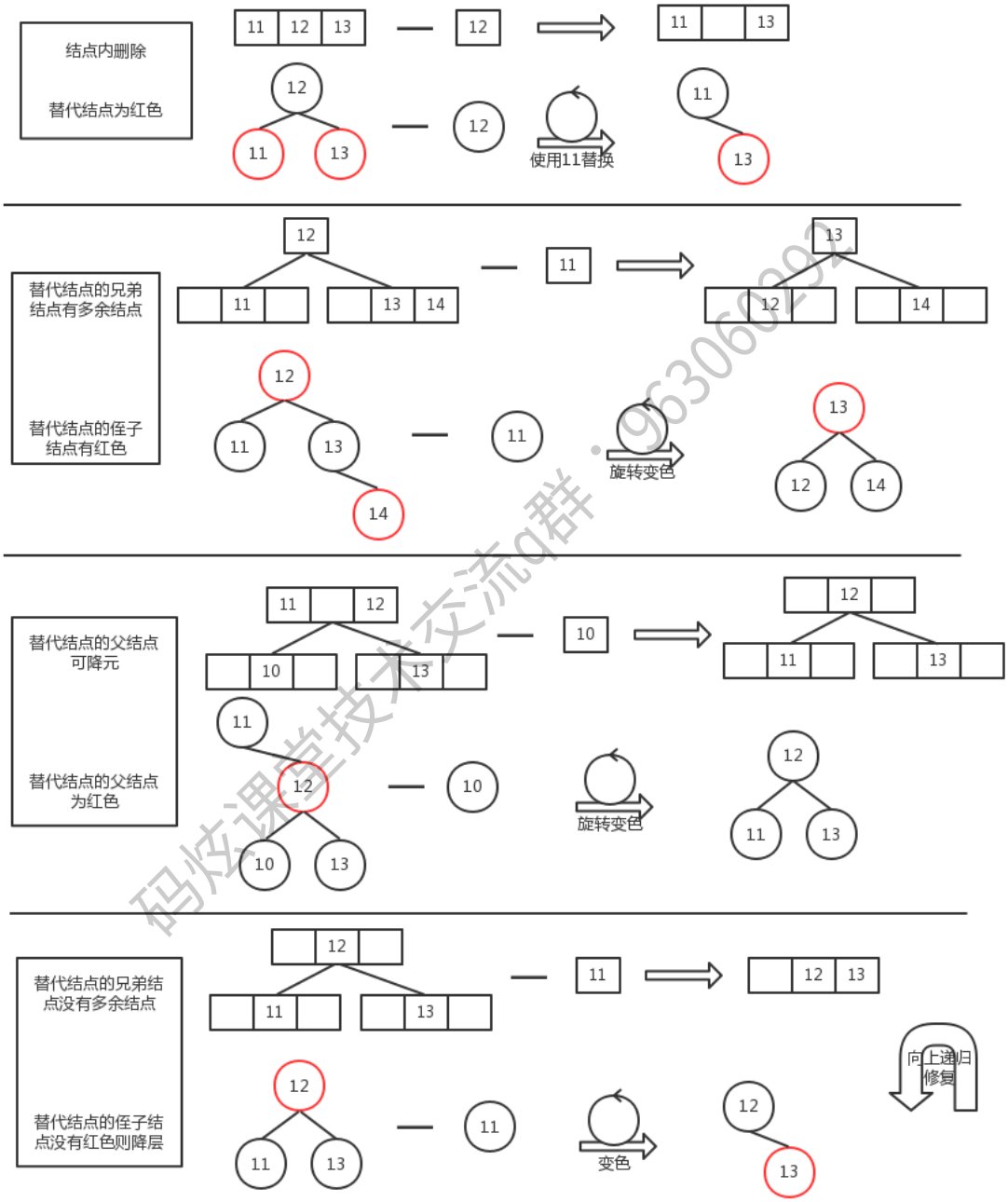
红黑树的删除要比插入要复杂一些，我们还是类比 2-3-4树来讲：

- 查找最近的叶子结点中的元素替代被删除元素，删除替代元素后，从替代元素所处叶子结点开始处理；
- 降元：4-结点变 3-结点，3-结点变 2-结点；
- 2-结点中只有一个元素，所以借兄弟结点中的元素来补充删除后的造成的空结点；
- 当兄弟结点中也没有多个元素可以补充时，尝试将父结点降元，失败时向上递归，至到子树降元成功或到 root 结点树高减1；

将这些规则对应到红黑树中即：

- 查找离当前结点最近的叶子结点作为`替代结点`（左子树的最右结点或右子树的最左结点都能保证替换后保证二叉查找树的结点的排序性质，叶子结点的替代结点是自身）替换掉被删除结点，从替代的叶子结点向上递归修复；

- 替代结点颜色为红色（对应 2-3-4树中 4-结点或 3-结点）时删除子结点直接成功；
- 替代结点为黑色（对应 2-3-4树中 2-结点）时，意味着替代结点所在的子树会降一层，需要依次检验以下三项，以恢复子树高度：
  - 兄弟结点的子结点中有红色结点（兄弟结点对应 3-结点或 4-结点）能够“借用”，旋转过来后修正颜色；
  - 父结点是红色结点（父结点对应 3-结点或 4-结点，可以降元）时，将父结点变黑色，自身和兄弟结点变红色后删除；
  - 父结点和兄弟结点都是黑色时，将子树降一层后把`父结点当作替代结点`递归向上处理。



如上图，删除的要点是`找到替代结点`，如果替代结点是黑色，递归向上依次判断侄子结点、父结点是否可以补充被删除的黑色，整体思想就是将删除一个黑色结点造成的影响局限在子树内处理。

注：倒数第二张图是错误的。

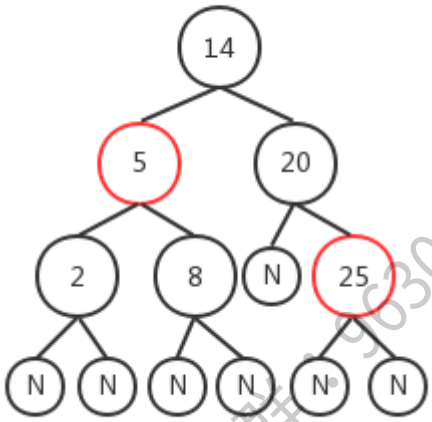
五、红黑树

定义

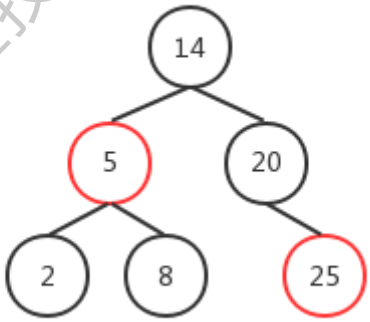
红黑树是一种结点带有颜色属性的二叉查找树，但它在二叉查找树之外，还有以下5大性质：

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是NIL节点，这类节点不可以忽视，否则代码会看不懂）。
4. 每个红色节点必须有两个黑色的子节点。（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。）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点（黑色平衡）。

下图就是一个典型的红黑树：



但实现上我省略了其中的 Nil 结点，一般如下图，大家理解时也可以忽略它们。



常见操作

变色：节点的颜色由黑变红或者由红变黑

左旋：以某个节点作为旋转点，其右子节点变为旋转节点的父节点，右子节点的左子节点变为旋转节点的右子节点，左子节点保持不变。

右旋：以某个节点作为旋转点，其左子节点变为旋转节点的父节点，左子节点的右子节点变为旋转节点的左子节点，右子节点保持不变。

新增：分情况讨论，主要是要找到插入位置，然后自平衡（左旋或者右旋）且插入节点是红色（如果是黑色的话，那么当前分支上就会多出一个黑色节点出来，从而破坏了黑色平衡），以下分析全部以左子树为例子，右子树的情况则相反。

- 如果插入的是第一个节点（根节点），红色变黑色
- 如果父节点为黑色，则直接插入，不需要变色
- 如果父节点为红色，叔叔节点也是红色（此种情况爷爷节点一定是黑色），则父节点和叔叔节点变黑色，爷爷节点变红色（如果爷爷节点是根节点，则再变成黑色），爷爷节点此时需要递归（把爷爷节点当做新插入的节点再次进行比较）
- 如果父节点是红色，没有叔叔节点或者叔叔节点是黑色（此时只能是NIL节点），则以爷爷节点为支点右旋，旋转之后原来的爷爷节点变红色，原来的父节点变黑色。

右子树的情况和左子树类似，请自行研究，不再赘述。

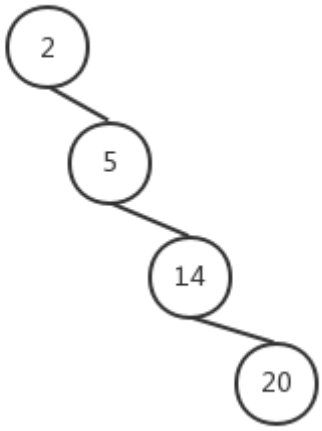
删除（重点）：

通俗点讲就是三句话：**自己能搞定的自己搞定；搞不定的找兄弟和父亲帮忙；父亲和兄弟都帮不了那有福同享，有难同当（父亲和兄弟自损）**

- 自己能搞定的自己搞定
  - 如果删除的节点对应于2-3-4树的3节点或者4节点，则直接删除，不用跟兄弟和父亲借
  - 如果删除的是红色节点，则直接删；如果删除的是黑色节点，则红色节点上来替代，变黑即可
- 搞不定的找兄弟和父亲帮忙
  - 前提是找到“真正”的兄弟节点（如何找见视频讲解）
  - 兄弟节点有的借（此时兄弟节点一定是黑色，如果是红色那说明这个节点不是真正的兄弟节点，需要回到上一步找真正的兄弟节点）
    - 兄弟节点有两个子节点的情况（2个子节点肯定是红色，如果是黑色的话相当于此时兄弟节点对应2-3-4树是2节点，不可能有多余的元素可以借），此时需要旋转变色（具体见视频讲解）
    - 兄弟节点只有一个子节点的情况，此时需要旋转变色（具体见视频讲解）
- 兄弟和父亲节点帮不了忙，于是开始递归自损
  - 前提是找到“真正”的兄弟节点（如何找见视频讲解）
  - 兄弟节点没有多余的元素可借（此时兄弟节点一定为黑色2节点），此时兄弟节点所在分支也要自损一个黑色节点以此达到黑色平衡，最快的方式就是兄弟节点直接变红（相当于就是减少一个黑色节点），此时一父节点为root的子树又达到了平衡（两边都比之前少一个黑色）。但是以祖父节点为root的树依然是不平衡的，此时需要递归处理，具体见视频讲解。

优势和用途

我们知道二叉查找树在不停地添加或删除结点后，可能会导致结点情况如下：



这种情况下，二叉查找树的查找效率最坏会降低为 `O(n)`。

而红黑树由于在插入和删除结点时都会进行变色旋转等操作，在符合红黑树条件的情况下，即使一边子树全是黑色结点，另一边子树全是红黑相间，两子树的高度差也不会超过一半。一棵有 n 个结点的红黑树高度至多为 `2log(n+1)`，查找效率最坏为 `O(log(n))`。

所以红黑树常被用于需求查找效率稳定的场景，如 Linux 中内核使用它管理内存区域对象、Java8 中 HashMap ，TreeMap的实现等，所以了解红黑树也很有意义。

六、小结

红黑树是数据结构中相对比较难理解的一种数据结构，所以很多讲师不太愿意讲这部分内容，其中很大一部分原因就是很多半吊子讲师自己只有半碗水，他怎么能给你传道受业呢？想给别人讲半碗水的内容，起码自己得有一碗水吧。讲红黑树低于10个小时的课程建议大家不要看了，浪费时间！