扩展：CAS无锁算法

2018年9月8日 20:43

锁（lock）的代价

锁是用来做并发最简单的方式，当然其代价也是最高的。内核态的锁的时候需要操作系统进行一次上下文切换，加锁、释放锁会导致比较多的上下文切换和调度延时，等待锁的线程会被挂起直至锁释放。在上下文切换的时候，cpu之前缓存的指令和数据都将失效，对性能有很大的损失。用户态的锁虽然避免了这些问题，但是其实它们只是在没有真实的竞争时才有效。

Java在JDK1.5之前都是靠synchronized关键字保证同步的，这种通过使用一致的锁定协议来协调对共享状态的访问，可以确保无论哪个线程持有守护变量的锁，都采用**独占**的方式来访问这些变量，如果出现多个线程同时访问锁，那未抢到锁的线程将被挂起，当线程恢复执行时，必须等待其它线程执行完他们的时间片以后才能被调度执行，**在挂起和恢复执行过程中存在着很大的开销。**锁还存在着其它一些缺点，当一个线程正在等待锁时，它不能做任何事。如果一个线程在持有锁的情况下被延迟执行，那么所有需要这个锁的线程都无法执行下去。如果被阻塞的线程优先级高，而持有锁的线程优先级低，将会导致优先级反转(Priority Inversion)。

乐观锁与悲观锁

独占锁是一种悲观锁，synchronized就是一种独占锁，它假设最坏的情况，并且只有在确保其它线程不会造成干扰的情况下执行，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。**所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**

CAS无锁算法

要实现无锁（lock-free）的非阻塞算法有多种实现方法，其中 CAS（比较与交换，Compare and swap） 是一种有名的无锁算法。CAS, CPU指令，在大多数处理器架构中都支持调用。**CAS的语义是“我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少”。**

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。**CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。**CAS无锁算法的C实现如下：

```
int compare_and_swap (int* reg, int oldval, int newval) {
ATOMIC(); int old_reg_val = *reg;
 if (old_reg_val == oldval) *reg = newval;
END_ATOMIC();
return old_reg_val;
}
```

以上翻译过来就是指当两者进行比较时，如果相等，则证明共享数据没有被修改，替换成新值，然后继续往下运行；如果不相等，说明共享数据已经被修改，放弃已经所做的操作，然后重新执行刚才的操作。

扩展：红黑树

2017年12月21日 9:15

概述

红黑树(Red Black Tree) 是一种**自平衡二叉查找树**，是在计算机科学中用到的一种数据结构，它是在1972年由Rudolf Bayer发明的，当时被称为平衡二叉B树(symmetric binary B-trees)。后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的"红黑树"。

红黑树在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。

它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的: 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。

它的统计性能要好于平衡二叉树。因此，红黑树在很多地方都有应用。在C++ STL中，很多部分(包括set, multiset, map, multimap)应用了红黑树的变体(SGI STL中的红黑树有一些变化，这些修改提供了更好的性能，以及对set操作的支持)。其他平衡树还有:AVL，SBT，伸展树，TREAP 等等。

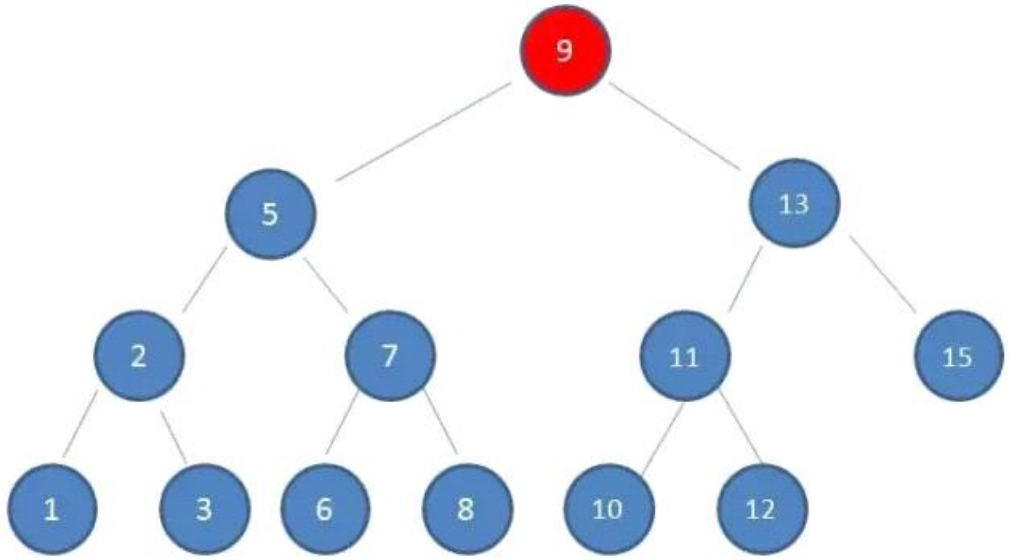
二叉查找树（BST）具备的特性呢

1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉排序树。

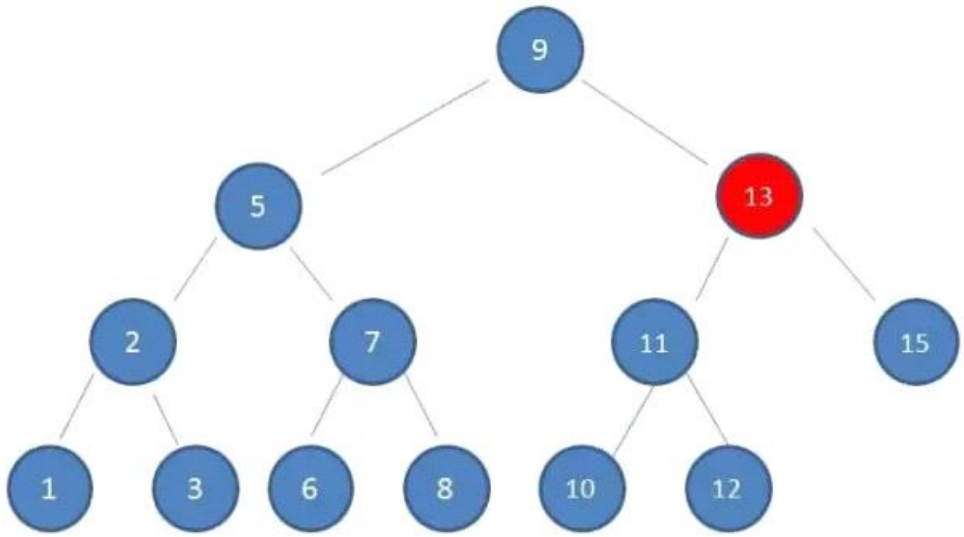
下图中这棵树，就是一颗典型的二叉查找树：



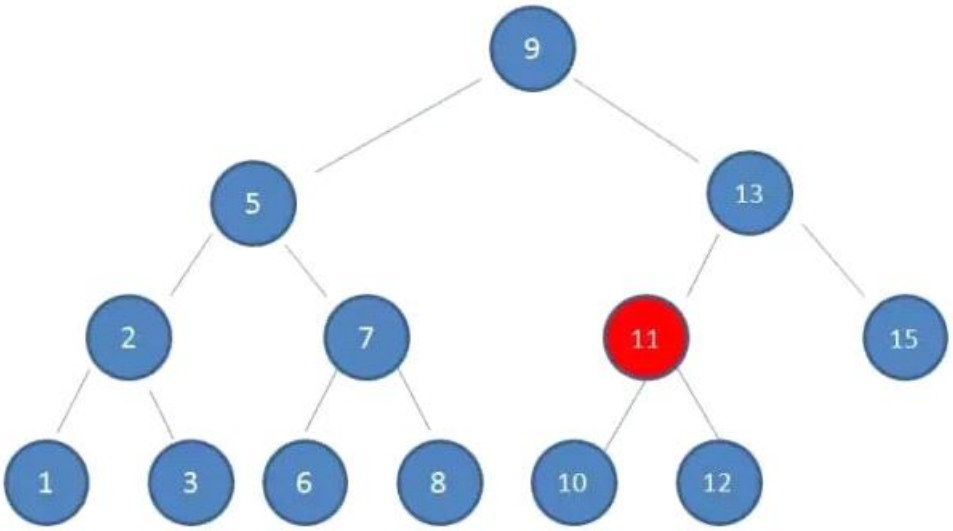
找10的过程：

1.查看根节点9：

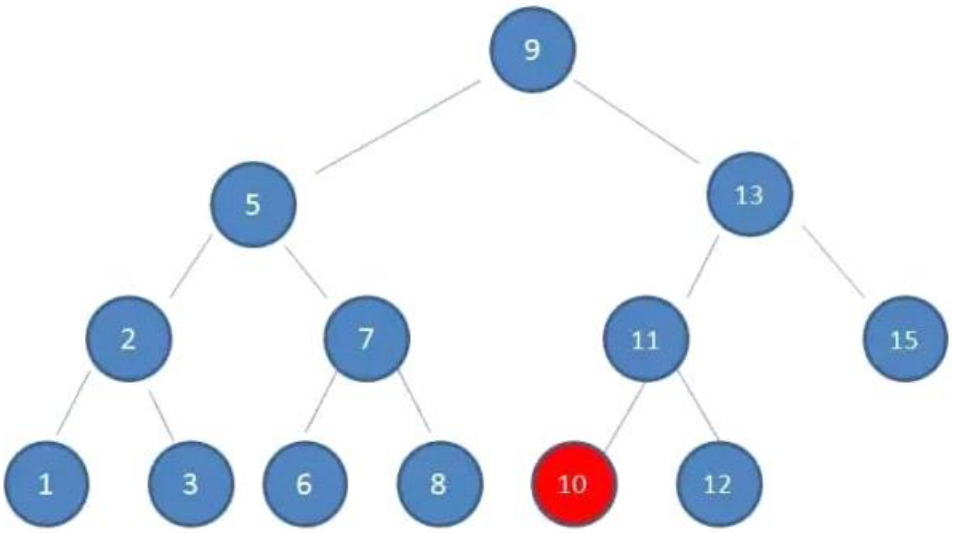
2.由于10 > 9，因此查看右孩子13：



3.由于10 < 13，因此查看左孩子11：



4.由于10 < 11，因此查看左孩子10，发现10正是要查找的节点：

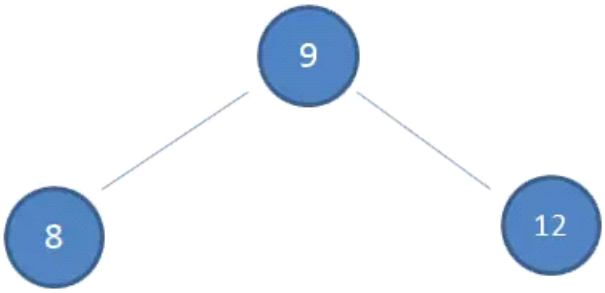


这种方式正式二分查找的思想，查找所需的最大次数等同于二叉查找树的高度

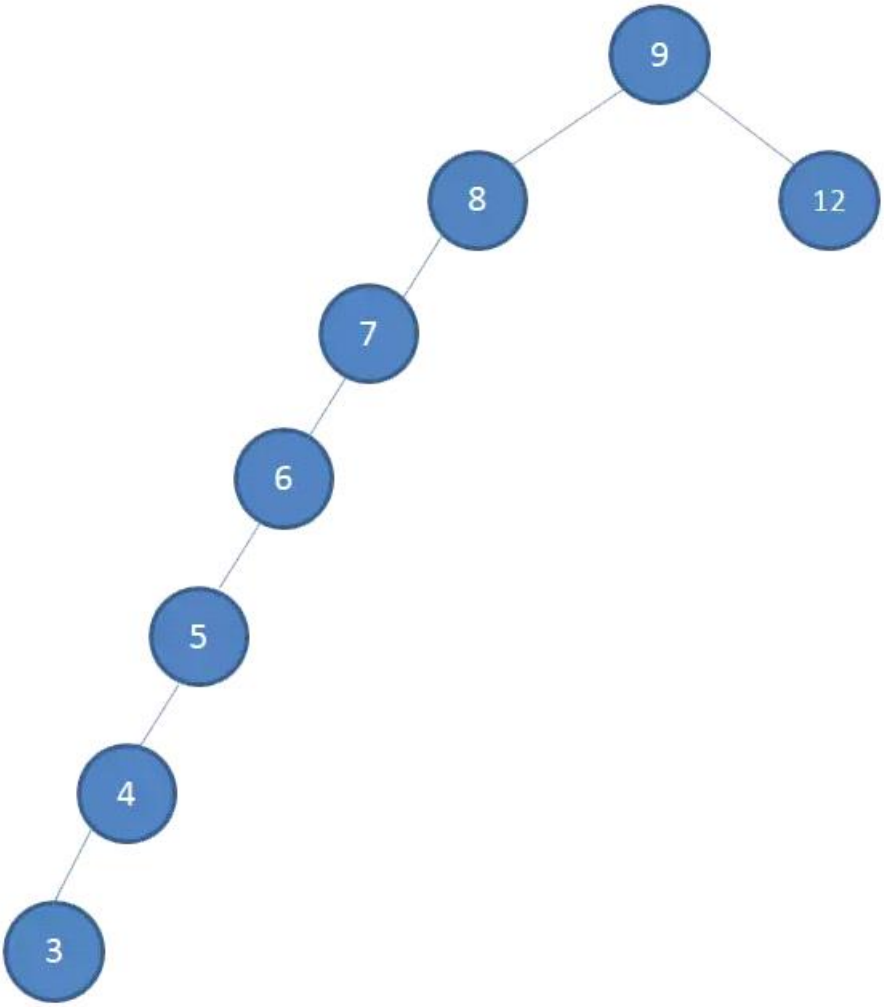
在插入节点的时候也是利用类似的方法，通过一层一层比较大小，找到新节点适合插入的位置。

但二叉查找树仍然存在缺陷，体现在插入数据时。

假设初始的二叉查找树只有三个节点，根节点值为9，左孩子值为8，右孩子值为12：



接下来我们依次插入如下五个节点：7,6,5,4,3。依照二叉查找树的特性，结果会变成什么样呢？



可以看到，二叉树变成了瘸子，查找几乎变成了线性查找。

红黑树

红黑树是一种自平衡的二叉查找树，除了符合二叉树的基本特性之外，它还具有附加特性：

1.节点是红色或黑色。

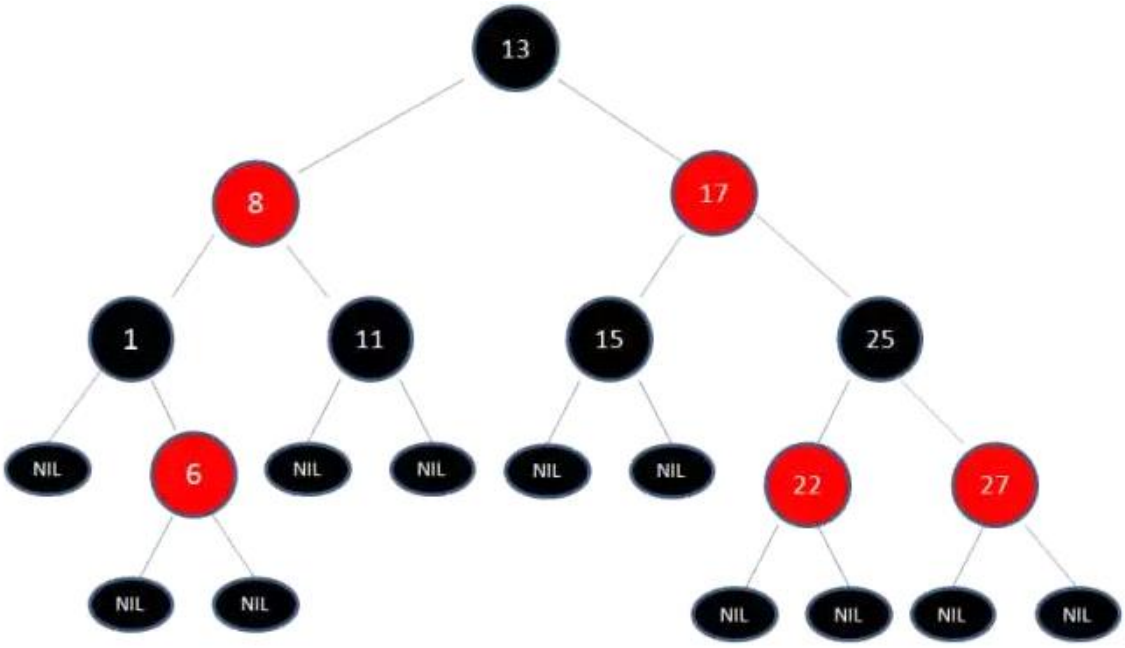
2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

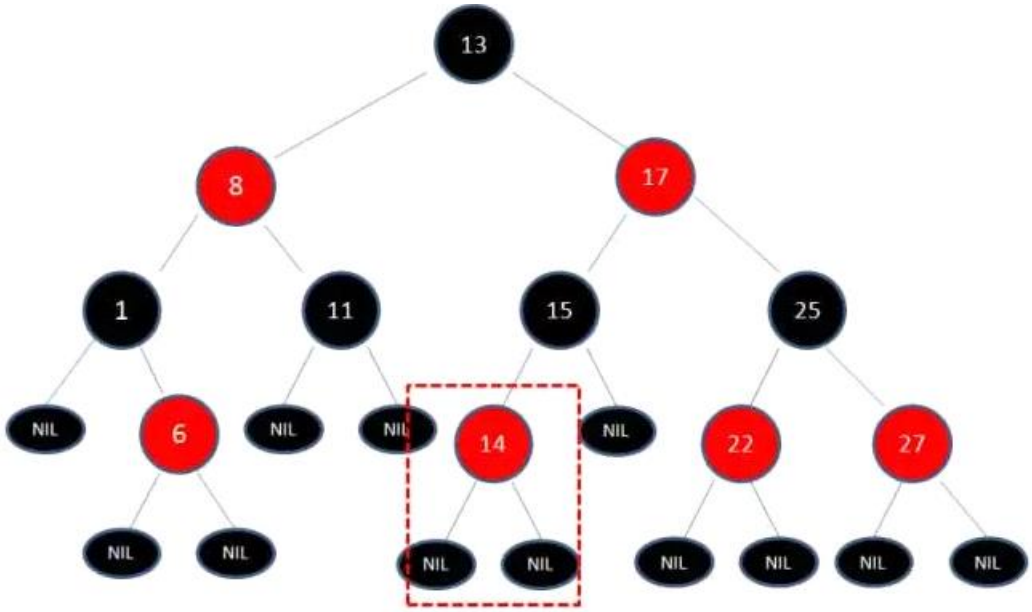
4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上**不能有两个连续的红色节点**)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

下图中这棵树，就是一颗典型的红黑树：

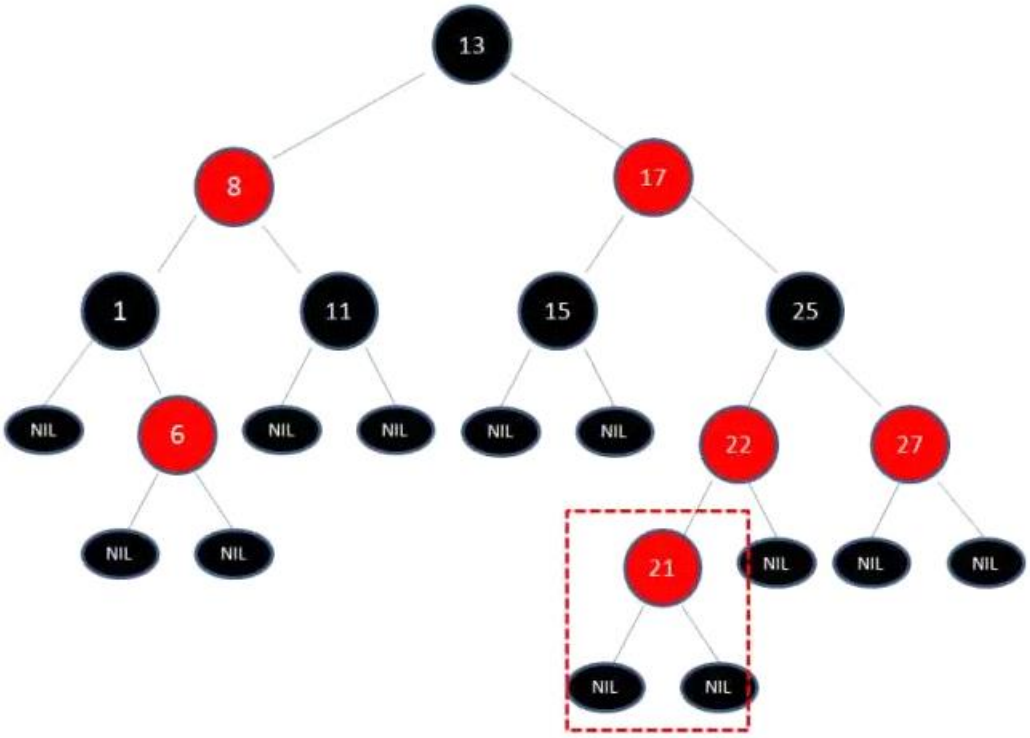


比如插入14数值后，会变成这样：



但很多情况下，向红黑树中插入数据会打破红黑树的5个特征，所以需要红黑树通过旋转来保证。

比如插入21之后的树：



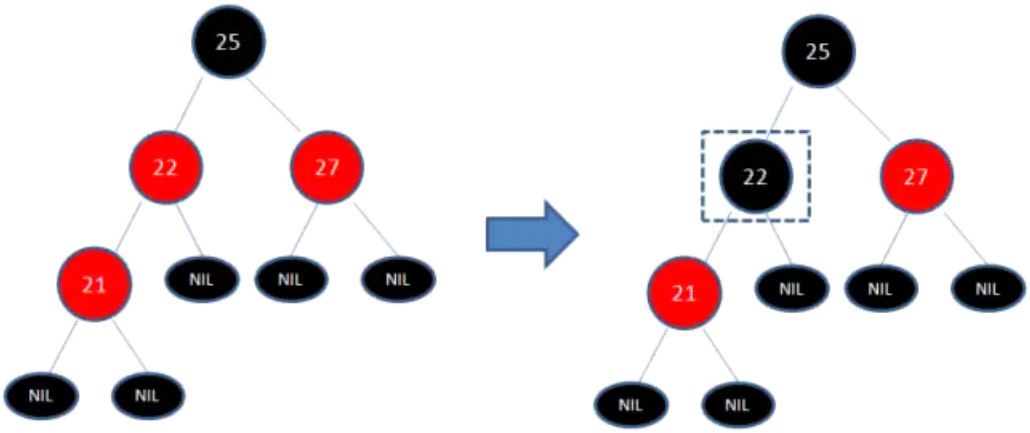
由于父节点22是红色节点，因此这种情况打破了红黑树的规则4（每个红色节点的两个子节点都是黑色），必须进行调整，使之重新符合红黑树的规则。

红黑树的变色与旋转

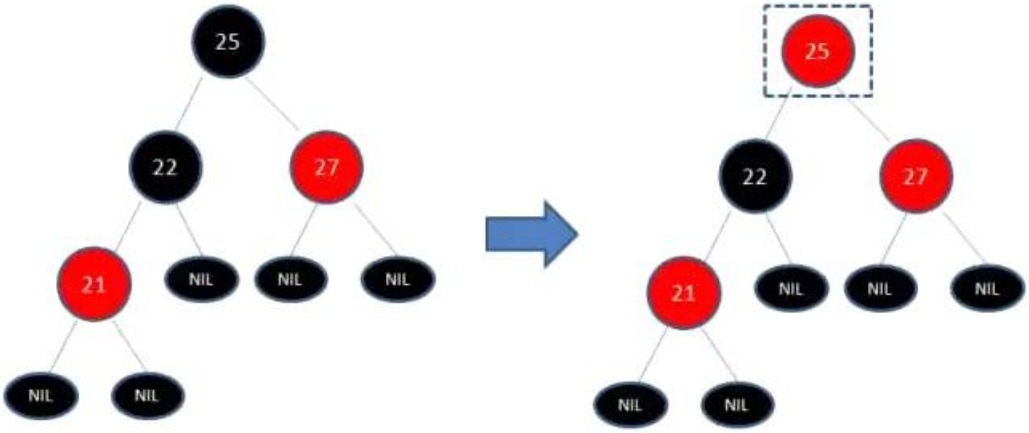
调整的方式是通过变色和旋转

变色是为了重新符合红黑树的规则，尝试把红色节点变为黑色，或者把黑色节点变为红色。

下图所表示的是红黑树的一部分，需要注意节点25并非根节点。因为节点21和节点22连续出现了红色，不符合规则4，所以把节点22从红色变成黑色：



但这样并不算完，因为凭空多出的黑色节点打破了规则5，所以发生连锁反应，需要继续把节点25从黑色变成红色：



此时仍然没有结束，因为节点25和节点27又形成了两个连续的红色节点，需要继续把节点27从红色变成黑色：