简介

红黑树被广泛的应用，Java集合中的TreeSet和TreeMap等。

二叉树（BST）查找树存在的瑕疵

最好的情况查找、插入和删除操作的时间复杂度都为O(logn),底数为2
最坏的情况下，一颗斜树，插入删除时间复杂度都为)(longn)，与线性表一致
实际情况: 二叉搜索树的效率应该在O(N)和O(logN)之间，这取决于树的不平衡程度

平衡二叉树存在的瑕疵

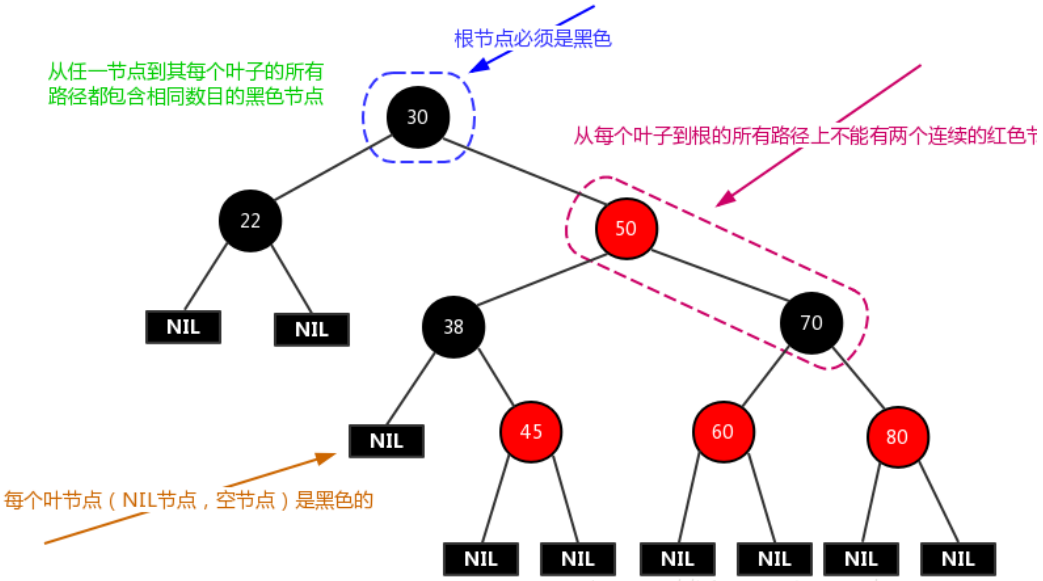
在查找平衡树（AVL）中，查找可以达到Log2(n)，插入可以通过最多两次旋转来保证高效需要，平衡因子改变也是有规律的只需要改变在删除节点时有可能因为失衡，导致需要从删除节点的父节点开始，不断的回溯到根节点，如果这棵平衡二叉树很高的话，那中间就要判断护，而数据量一旦很大，就必然高度很高

$$2^{10} = 1024$$

$$2^{15} = 32768$$

如上：一个 高度为 10的二叉树，节点最多 为 1024个，一个高度为15的二叉树，节点最多为32768个

红黑树的性质

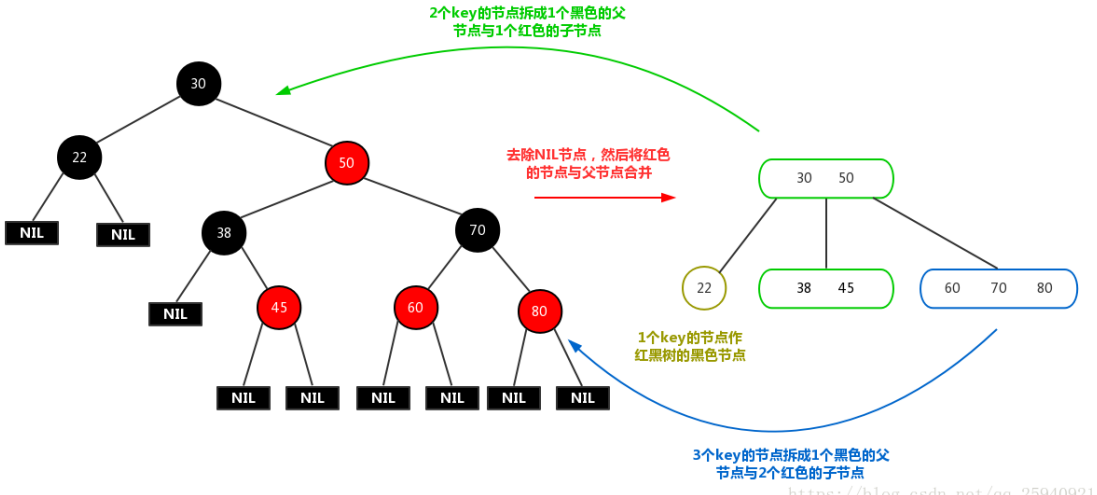


（1）节点是红色或黑色；
（2）根节点是黑色；
（3）如果节点是红色的，则它的子节点必须是黑色的（反之不一定）,(也就是从每个叶子到根的所有路径上不能有两个连续的红色节点
（4）从任一节点到该节点自身的每个叶子的所有路径都包含相同数目的黑色节点
（5）每个叶子节点（Null节点，空节点)是黑色的

红黑树与2-3-4树

红黑树从本质上来说是一棵2-3-4树，但是红黑树的子节点最多只能有两个，在历史上，也是先提出了2-3-4树，后来红黑树由它发展而

红黑树与2-3-4树之间是可以互相转化的，如果一棵树满足红黑树，把红色的节点收缩到其父节点，就变成了2-3-4树，所有红色节点都点，其它节点为1个key的节点。如下图



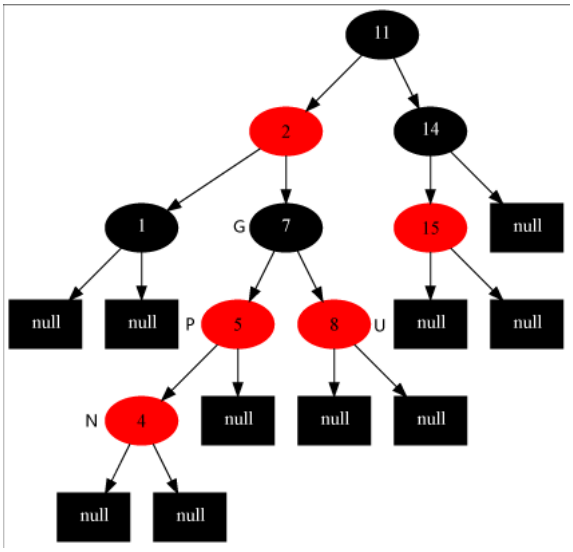
红黑树的构建（插入）过程

1. 首先，新节点直接插入并初始化颜色为**红色**

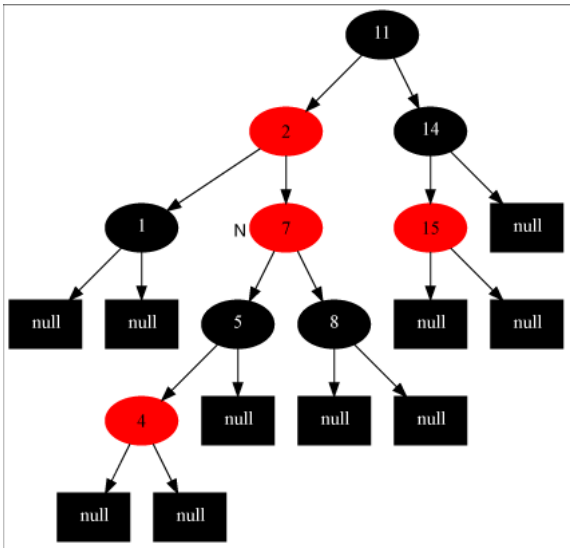
```
因为插入节点是红色的概率更大一些，所以初始化为红色，如果颜色不对，进行更正
```

2. 根据实际情况，判断是否需要对插入节点进行变色
   - 若新插入节点是根节点，违反了性质（2），则将节点颜色改为黑色即可
   - 若新插入节点的父节点为黑色，此时无需调整，满足红黑树
   - 若新插入节点的父节点为红色，违反了性质（3），此时需要对树进行平衡更正，有三种情况
     - 插入节点的父节点和其叔叔节点（祖父节点的另一个子节点）均为红色

       此时，肯定存在祖父节点，但是不知道父节点是其左子节点还是右子节点，但是由于对称性，我们只要讨论出一边的情况虑父节点是其祖父节点的左子节点的情况，如下图所示

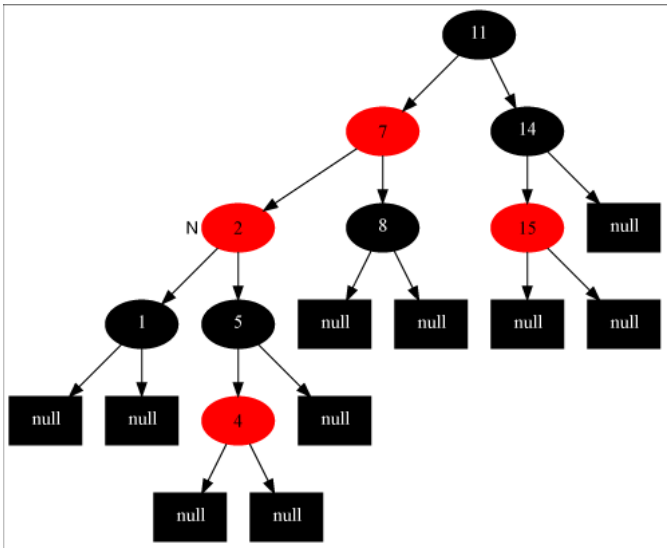


对于这种情况，我们要做的操作有：将当前节点(4) 的父节点(5) 和叔叔节点(8) 涂黑，将祖父节点(7)涂红。再将当前节点前节点（2）开始算法（具体看下面的步骤）。这样下图就变成下面的情况2了（插入节点的父节点是红色的，叔叔节点是黑色的）



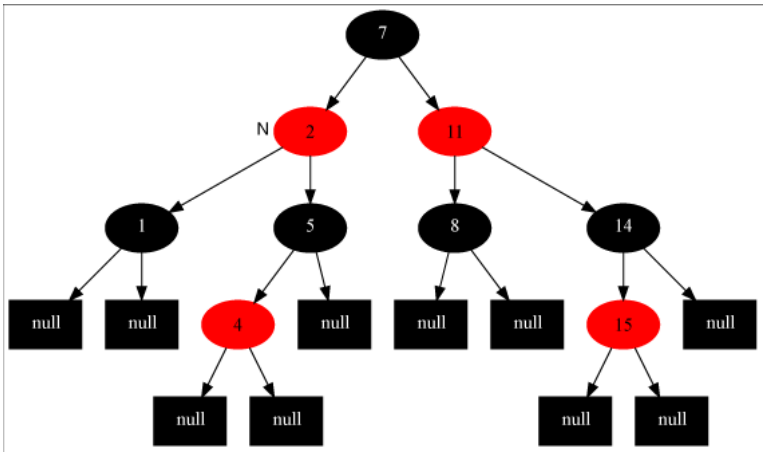
     - 插入节点的父节点是红色的，叔叔节点是黑色的

       要做的操作有：将当前节点(7)的父节点(2)作为新的当前节点，以新的当前节点（2）为支点做左旋操作。完成后如下图所



- 插入节点的父节点是红色，叔叔节点是黑色

要做的操作有：将当前节点的父节点(7)涂黑，将祖父节点(11)涂红，在祖父节点为支点做右旋操作。最后把根节点涂黑，示。至此，插入操作完成



我们可以看出，如果是从情况1开始发生的，必然会走完情况2和3，也就是说这是一整个流程，当然咯，实际中可能不一定生，那再走个情况3即可完成调整，如果直接只要调整情况3，那么前两种情况均不需要调整了。故变色和旋转之间的先后关系可以表示

总结

红黑树能够以 O(log2(N)) 的时间复杂度进行搜索、插入、删除操作。

一般红黑树也被拿出来跟 AVL树 做比较，都说红黑树的综合性能比 AVL树 要好，但是好在哪里呢?

实际上大体来讲 AVL树 的插入，删除还有查询的时间复杂度跟红黑树是一样的，都是 O(log2(N)) 。有人说，红黑树比AVL树性能好的原旋转3次，而 AVL树 删除节点的删除可能会不断的回溯直到根，这点确实是，但不是最主要的原因，因为红黑树删除节点过程中也可能需要么多旋转而已（而且旋转也不会耗费很大的性能，只是将节点的引用关系做下改变而已，可以认为是 O(1) ），红黑树比 AVL树 性能好最主要右子树高度差为1），所以不管是插入节点也好，删除节点也好，都会很容易导致树的不平衡，从而引发调整，而红黑树不是严格的平衡树，进行调整，所以这点才是红黑树综合性能比 AVL树 好的原因