**1. 简介**

红黑树是一种**自平衡二叉查找树**。它的统计性能要好于平衡二叉树（AVL树），因此，红黑树在很多地方都有应用。在C++ STL中，很多部分(目前包括**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)。**它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在O(log n)时间内做查找，插入和删除等操作。**

本文介绍了红黑树的基本性质和基本操作。

**2. 红黑树的性质**

红黑树，顾名思义，通过**红黑两种颜色域保证树的高度近似平衡(左右子节点的高度差不大于1)。**它的每个节点是一个**五元组**：**color（颜色），key（数据），left（左孩子），right（右孩子）和p（父节点）。**

**二叉搜索树的规则是：任何节点的键值一定大于其左子树的每一个节点值，并小于右子树的每一个节点值。**

红黑树的定义也是它的性质，有以下五条：

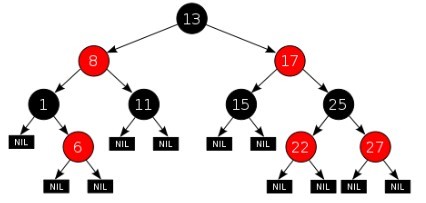
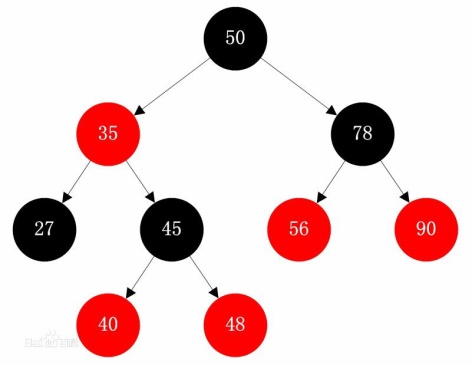
**性质1. 节点是红色或黑色**

**性质2. 根是黑色**

**性质3. 所有叶子都是黑色（叶子是NIL节点）**

**性质4. 如果一个节点是红的，则它的两个儿子都是黑的**

**性质5. 从任一节点到其叶子的所有简单路径都包含相同数目的黑色节点。**

[](http://dongxicheng.org/wp-content/uploads/2011/08/red-black-tree.jpg)

这五个性质强制了红黑树的关键性质: **从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。**为什么呢？性质4暗示着任何**一个简单路径上不能有两个毗连的红色节点**，这样，**最短的可能路径全是黑色节点，最长的可能路径有交替的红色和黑色节点。**同时根据性质5知道：**所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。**

**一、红黑树的介绍**

先来看下算法导论对R-B Tree的介绍：  
红黑树，**一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是Red或Black。**通过对任何一条从根到叶子的路径上各个结点着色方式的限制，**红黑树确保没有一条路径会比其他路径长出俩倍，因而是接近平衡的。**

红黑树，作为一棵二叉查找树，满足二叉查找树的一般性质。下面，来了解下 二叉查找树的一般性质。

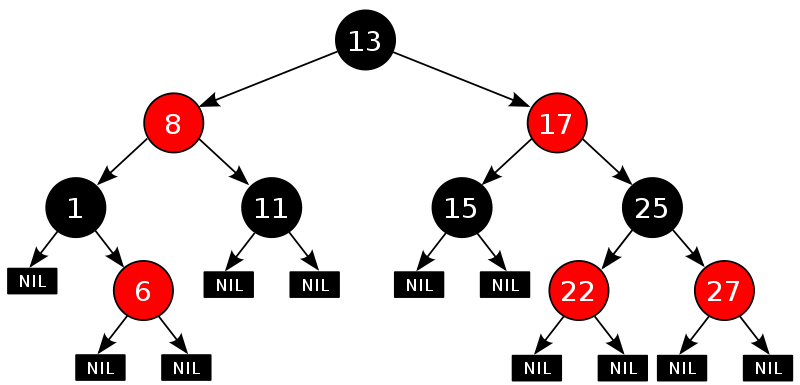
**二叉查找树**

二叉查找树，也称有序二叉树（ordered binary tree），或已排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

* **若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；**
* **若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；**
* **任意节点的左、右子树也分别为二叉查找树。**
* **没有键值相等的节点（no duplicate nodes）。**

因为一棵由n个结点随机构造的二叉查找树的高度为lgn，所以顺理成章，二叉查找树的一般操作的执行时间为O(lgn)。但**二叉查找树若退化成了一棵具有n个结点的线性链后，则这些操作最坏情况运行时间为O(n)。**

红黑树虽然本质上是一棵二叉查找树，但它在二叉查找树的基础上增加了**着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。**



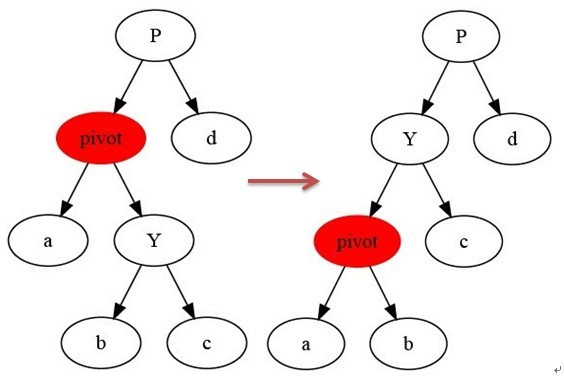
此图忽略了叶子和根部的父结点。同时，上文中我们所说的 "叶结点" 或"NULL结点"，如上图所示，它不包含数据而只充当树在此结束的指示，这些节点在绘图中经常被省略，望看到此文后的读者朋友注意。

**二、树的旋转知识**

    当在对红黑树进行插入和删除等操作时，对树做了修改可能会破坏红黑树的性质。为了继续保持红黑树的性质，可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即通过修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后继续保持它的性质或平衡的目的。

    树的旋转分为左旋和右旋，下面借助图来介绍一下左旋和右旋这两种操作。

1.左旋



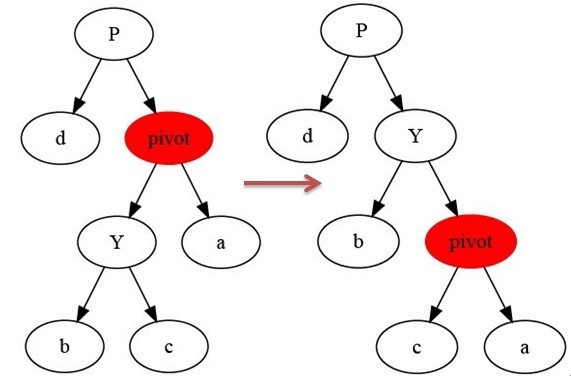
如上图所示，当在某个结点pivot上，做左旋操作时，我们假设它的右孩子y不是NIL[T]，pivot可以为任何不是NIL[T]的左子结点。左旋以pivot到Y之间的链为“支轴”进行，它使Y成为该子树的新根，而Y的左孩子b则成为pivot的右孩子。

**[cpp]** [view plain](https://blog.csdn.net/v_july_v/article/details/6105630) [copy](https://blog.csdn.net/v_july_v/article/details/6105630)

1. LeftRoate(T, x)
2. y ← x.right                    //定义y：y是x的右孩子
3. x.right ← y.left                //y的左孩子成为x的右孩子
4. **if** y.left ≠ T.nil
5. y.left.p ← x
6. y.p ← x.p                      //x的父结点成为y的父结点
7. **if** x.p = T.nil
8. then T.root ← y
9. **else** **if** x = x.p.left
10. then x.p.left ← y
11. **else** x.p.right ← y
12. y.left ← x                       //x作为y的左孩子
13. x.p ← y

2.右旋

右旋与左旋差不多，再此不做详细介绍。



树在经过左旋右旋之后，**树的搜索性质保持不变，但树的红黑性质则被破坏了，所以，红黑树插入和删除数据后，需要利用旋转与颜色重涂来重新恢复树的红黑性质。**

    至于有些书如《STL源码剖析》有对双旋的描述，其实双旋只是单旋的两次应用，并无新的内容，因此这里就不再介绍了，而且左右旋也是相互对称的，只要理解其中一种旋转就可以了。

**3. 红黑树的基本操作**

因为红黑树也是**二叉查找树**，因此红黑树上的查找操作与普通二叉查找树上的查找操作相同。然而，红黑树上的**插入操作和删除操作会导致不再符合红黑树的性质**。恢复红黑树的性质需要**少量(O(log n))的颜色变更(实际是非常快速的)和不超过三次树旋转(对于插入操作是两次)**。虽然**插入和删除很复杂，但操作时间仍可以保持为 O(log n) 次。**

**3.1 插入操作**

插入操作可以概括为以下几个步骤：

**(1) 查找要插入的位置，时间复杂度为：O(N)**

**(2) 将新节点的color赋为红色**

**(3) 自下而上重新调整该树为红黑树**

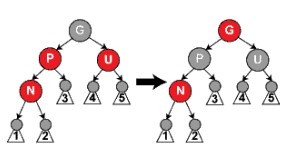
其中，第(1)步的查找方法跟普通二叉查找树一样，第(2)步之所以将新插入的节点的颜色赋为红色，是因为：**如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点（性质5：从任一节点到其叶子的所有简单路径都包含相同数目的黑色节点。），这个是很难调整的。**但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整，这样简单多了。下面讨论步骤(3)的一些细节：

设要插入的节点为N，其父节点为P，其父亲G的兄弟节点为U（即P和U是同一个节点的两个子节点）。

[1] 如果P是黑色的，则整棵树**不必调整便是红黑树**。

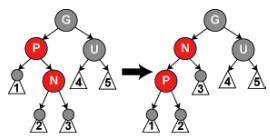
[2] 如果P是红色的（可知，其父节点G一定是黑色的），则插入z后，违背了性质4**（不能红红）**，需要进行调整。**调整时分以下3种情况**：

**（a）N的叔叔U是红色的**

[](http://dongxicheng.org/wp-content/uploads/2011/08/red-black-tree-s1.jpg)

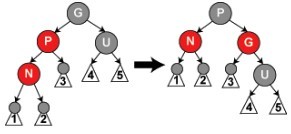
如上图所示，我们将**P和U重绘为黑色**并重绘节点**G为红色**(用来保持性质5)。现在新节点N有了一个黑色的父节点P，因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点G的父节点也有可能是红色的，这就违反了性质4。为了解决这个问题，**我们在祖父节点G上递归调整颜色。**

**（b）N的叔叔U是黑色的，且N是右孩子**



如上图所示，我们对**P进行一次左旋转调换新节点和其父节点的角色;** 接着，**按情形(c)处理以前的父节点P以解决仍然失效的性质4。**

**（c）N的叔叔U是黑色的，且N是左孩子**



如上图所示，**对祖父节点G 的一次右旋转; 在旋转产生的树中，**以前的**父节点P现在是新节点N和以前的祖父节点G的父节点，然后交换以前的父节点P和祖父节点G的颜色，**结果的树满足性质4，同时性质5[4]也仍然保持满足。（再递归上去即可）

**3.2 删除操作**

删除操作可以概括为以下几个步骤：

(1) 查找要删除位置，时间复杂度为：O(N)

(2) 用删除节点后继或者节点替换该节点（只进行数据替换即可，不必调整指针，后继节点是中序遍历中紧挨着该节点的节点，即：右孩子的最左孩子节点）

**(3) 如果删除节点的替换节点为黑色，则需重新调整该树为红黑树**

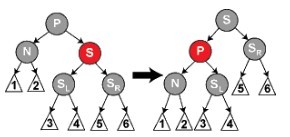
其中，第(1)步的查找方法跟普通二叉查找树一样，第(2)步之所以用后继节点替换删除节点，是**因为这样可以保证该后继节点之上仍是一个红黑树，**而后继节点可能是一个叶节点或者只有右子树的节点，**这样只需用有节点替换后继节点即可达到删除的目的**。如果需要删除的节点有两个儿子，那么问题可以**被转化成删除另一个只有一个儿子的节点的问题**。（没看懂？？？可参考：http://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91 ）在第(3)步中，**如果，如果删除节点为红色节点，则他的父亲和孩子全为黑节点，这样直接删除该节点即可，**不必进行任何调整。如果**删除节点是黑节点，分四种情况：**

设要删除的节点为N，其父节点为P，其兄弟节点为S。

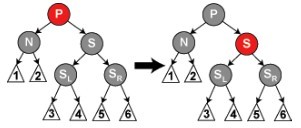
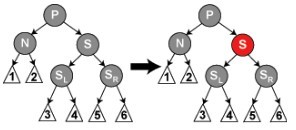
**由于N是黑色的，则P可能是黑色的，也可能是红色的，S也可能是黑色的或者红色的**

**（1）S是红色的**

此时P肯定是**黑色**的。我们对N的父节点进行左旋转，然后把红色兄弟转换成N的祖父。我们接着对调 N 的父亲和祖父的颜色。尽管所有的路径仍然有相同数目的黑色节点，现在 **N 有了一个黑色的兄弟和一个红色的父亲，**所以我们可以接下去按 (2)、(3)或(4)情况来处理。

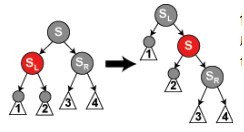


**（2）S和S的孩子全是黑色的**

在这种情况下，**P可能是黑色的或者红色的，**我们简单的重绘S 为红色。结果是通过S的所有路径，它们就是**以前不通过 N 的那些路径，都少了一个黑色节点。**因为删除 N 的初始的父亲使通过 N 的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过 P 的所有路径现在比不通过 P 的路径少了一个黑色节点。接下来，要调整以P作为N递归调整树。  


**（3）S是黑色的，S的左孩子是红色，右孩子是黑色**

这种情况下我们在 S 上做右旋转，这样 S 的左儿子成为 S 的父亲和 N 的新兄弟。我们接着交换 S 和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在 N 有了一个右儿子是红色的黑色兄弟，所以我们进入了情况（4）。N 和它的父亲都不受这个变换的影响。



**（4）S是黑色的，S的右孩子是红色**

在这种情况下我们在 N 的父亲上做左旋转，这样 S 成为 N 的父亲和 S 的右儿子的父亲。我们接着交换 N 的父亲和 S 的颜色，并使 S 的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以属性 3 没有被违反。但是，N 现在增加了一个黑色祖先: 要么 N 的父亲变成黑色，要么它是黑色而 S 被增加为一个黑色祖父。所以，通过 N 的路径都增加了一个黑色节点。

