红黑树学习

1.红黑树的介绍

红黑树（Red-Black Tree，简称R-B Tree），它是一种特殊的二叉查找树。

红黑树是特殊的二叉查找树，意味着它满足二叉查找树的特征：任意一个结点所包含的键值，大于等于左孩子的键值，小于等于右孩子的键值。除了具备该特性之外，红黑树还包括许多额外信息。红黑树的每个结点上都有存储位表示结点的颜色，颜色是红或黑。

红黑树的特性：

（1）每个结点或者是黑色，或者是红色；

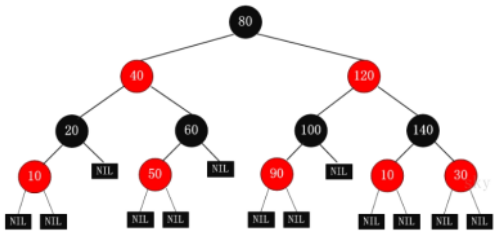
（2）根结点是黑色；

（3）每个叶子结点是黑色，注意：这里叶子结点，是指为空的叶子结点；

（4）如果一个结点是红色的，则它的子结点必须是黑色的；

（5）从一个结点到该结点的子孙结点的所有路径上包含相同数目的黑结点。

注意：特性3中的叶子结点，是只为空的结点；特性5，确保没有一条路径会比其他路径长出两倍。因而，红黑树是接近平衡的二叉树。



2.红黑树的应用

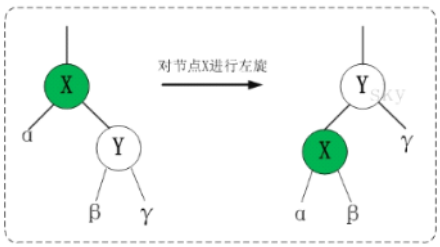
红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率很高。例如，Java集合中的TreeSet和TreeMap，C++STL中的set、map以及Linux虚拟内存的管理，都是通过红黑树去实现的。

2.红黑树的基本操作

（1）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结点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一棵红黑树，而是一棵普通的树。而通过旋转，可以使这棵树重新成为红黑树。简单地说，旋转的目的是让树保持红黑树的特性。

- 左旋

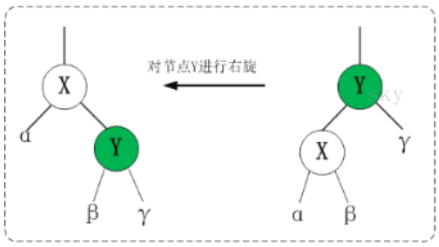


对x进行左旋，意味着将x变成一个左结点。

伪代码：

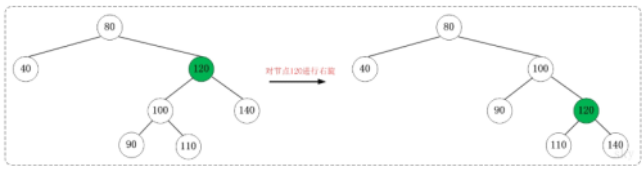
```
LEFT-ROTATE(T, x)
01  y ← right[x]            // 前提：这里假设x的右孩子为y。下面开始正式操作
02  right[x] ← left[y]      // 将 “y的左孩子” 设为 “x的右孩子”，即 将β设为x的右孩子
03  p[left[y]] ← x          // 将 “x” 设为 “y的左孩子的父亲”，即 将β的父亲设为x
04  p[y] ← p[x]             // 将 “x的父亲” 设为 “y的父亲”
05  if p[x] = nil[T]
06  then root[T] ← y                // 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”
```

- 右旋



伪代码：

```
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”
```



（2）添加

将一个结点插入到红黑树中，需要执行哪些步骤？首先，将红黑树当作一棵二叉查找树，将结点插入；然后，将结点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一棵红黑树。详细描述如下：

- 第一步：将红黑树当作是一棵二叉查找树，将结点插入。

红黑树本身是一棵二叉查找树，将结点插入后，该树仍然是一棵二叉查找树。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棵二叉查找树的事实。

接下来，就是想方设法的通过旋转和重新着色操作，使这棵树重新成为红黑树。

- 第二步：将插入的结点着色为“红色”。

为什么着色成红色，而不是黑色呢？

根据红黑树的特性，将插入的结点着色为红色，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他性质即可；满足了的话，它就又是一棵红黑树。

- 第三步：通过一系列的旋转或着色等操作，使之重新成为一棵红黑树。

第二步中，将插入结点着色为“红色”之后，不会违背特性5，那会违背哪些特性呢？对于特性1，显然不会违背，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，然后执行插入操作。而根据二叉查找树的特点，插入操作不会改变根结点。所以，根结点仍然是黑色；对于特性3，显然不会违背，这里的叶子结点指的是空叶子结点，插入非空结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，就可以重新构造红黑树。

（3）删除