**3.3 平衡查找树**

**3.3.1 2-3查找树**

我们将一棵标准的二叉查找树中的结点称为2-结点（含有一个键和两条链接），而现在我们引入3-结点，它含有两个键和三条链接。

一棵2-3查找树或为一棵空树，或由以下结点组成：

2-结点，含有一个键和两条链接，左链接指向2-3树中的键都小于该结点，右链接指向的键都大于该结点。

3-结点，含有两个键和三条链接，左连接和右链接同上，中链接指向2-3树中的键都位于该结点的两个键之间。

一棵完美平衡的2-3查找树中的所有空链接到根节点的距离都应该是相同的。简洁起见，这里我们用2-3树指代一棵完美平衡的2-3查找树。

**插入新键：**

如果未命中的查找结束于一个2-结点，我们把2-结点替换为3-结点，将要插入的键保存在其中即可。

向一棵只含有一个3-结点的树中插入新键：

1，假设我们需要向一棵只含有一个3-结点的树中参入一个新键，我们先临时创建一个4-结点，然后转换为一棵由3个2-结点组成的2-3树；

2，假设未命中的查找结束于一个3-结点，而它的父结点是一个2-结点。我们先临时创建一个4-结点，然后将中键移动到原来的父结点，父结点变成3-结点

3，假设未命中的查找结束于一个父结点为3-结点的结点。我们一直向上不断分解临时的4-结点，直到遇到一个2-结点并将它替换为一个不需要继续分解的3-结点，或者到达3-根节点。

将一个4-结点分解为一棵2-3树可能有6种情况，这个4-结点可能是根节点，可能是一个2-结点的左子节点或右子节点，也可能是一个3-结点的左，中，右子节点。

在一棵大小为N的2-3树中，查找和插入操作访问的结点必然不超过lgN个。

含有10亿个结点的一棵2-3树的高度仅在19到30之间，我们最多只需要访问30个结点就能够在10亿个键中进行任意查找和插入操作，这是相当惊人的。

尽管我们可以用不同的数据类型表示2-结点和3-结点并写出变换所需的代码，但用这种直白的表示方法实现大多数操作并不方便，因为需要处理的情况太多了。

**3.3.2 红黑二叉查找树**

红黑二叉查找树背后的基本思想是用标准的二叉查找树和一些额外的信息替换3-结点来表示2-3树。

我们将树中的链接分为两种类型：红链接将两个2-结点连接起来构成一个3-结点，黑链接则是2-3树中的普通链接。

确切的说，我们将3-结点表示为由一条左斜的红色链接（两个2-结点其中之一是另一个的左子节点）相连的两个2-结点。

这种表示法的一个优点是，我们无需修改就可以直接使用标准二叉查找树的get()方法。

对于任意2-3树，只要对结点进行转换，我们都可以立即派生出一棵对应的二叉查找树。

我们将用这种方式表示2-3树的二叉查找树称为红黑二叉查找树，简称红黑树。

红黑树的另一种定义是含有红黑链接并满足下列条件的二叉查找树：

1，红链接均为左链接；

2，没有任何一个结点同时和两条红链接相连；

3，该树是完美黑色平衡的；

满足这样定义的红黑树和相应的2-3树是一一对应的。

如果我们将一棵红黑树中的红链接画平，那么所有的空链接到根节点的距离都是相同的。

红黑树能够将两个算法的优点结合起来：二叉查找树中简洁高效的查找方法和2-3树中高效的平衡插入算法。

当我们提到一个结点的颜色时，我们指的是该结点的链接的颜色。

在我们实现的某些操作中可能会出现红色右链接或者两条连续的红链接，但在操作完成前这些情况都会被小心地旋转并修复。

假设我们有一条红色的右链接需要被转化为左链接，这个操作叫做左旋转。

无论左旋转还是右旋转，旋转操作都会返回一条链接。我们总是会用rotateRight()或rotateLeft()的返回值重置父节点（或根节点）中相应的链接。

返回的链接可能是红色也可能是黑色，这可能会产生两条连续的红链接，但我们的算法会继续用旋转操作修正这种情况。

在插入新的键时，我们可以使用旋转操作帮助我们保证2-3树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：有序性和完美平衡性。

1，向单个2-结点中插入新键

一棵只含有一个键的红黑树只含有一个2-结点。插入另一个键，如果新键小于老键，我们只需要新增一个红色结点即可。

如果新键大于老键，那么新增的红色结点将会产生一条红色的右链接，我们需要将其旋转为红色左链接并修正根节点的链接，插入操作才算完成。

2，向树底部的2-结点插入新键

向一棵红黑树中插入一个新键会在树的底部新增一个结点，但总是用红链接将新节点和它的父结点相连。

如果它的父结点是一个2-结点，那么刚才的两种处理方法仍然适用。如果指向新节点的是父结点的左链接，那么父结点就直接成为了一个3-结点。

如果指向新节点的是父结点的右链接，这就是一个错误的3-结点，但一次左旋转就能够修正它。

3，向一个3-结点中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。

每种情况都会产生一个同时连接到两条红链接的结点，而我们的目标就是修正这一点。

1，新键大于原树中的两键，因此它被链接到3-结点的右链接。此时，有两条红链分别和较大和较小的结点相连。我们将两条链接的颜色由红变黑即可。

2，如果新键小于原树中的两个键，它会被连接到最左边的空链接，这样就产生了两条连续的红链接。此时我们只需要将上层的红链接右旋转即可得到第一种情况。

3，如果新键介于原树中的两个键之间，这会产生两条连续的红链接，一条红色左连接接一条红色右链接。我们需要将下层红链接左旋转即可得到第二种情况。

总的来说，我们通过0次，1次和2次旋转以及颜色的变化得到了期望的结果。

颜色转换：除了将子节点的颜色由红变黑之外，我们同时还要将父结点的颜色由黑变红。不会影响整棵树的黑色平衡性。

根节点总是黑色：颜色转换会使根节点变为红色。

严格地说，红色的根节点说明根节点是一个3-结点的一部分，但实际情况并不是这样。我们在每次插入后都会将根节点设为黑色。

每当根节点由红变黑时树的黑链接高度就会加1。

每次必要的旋转之后，我们都会进行颜色转换，这使得中结点变红。在父结点看来，处理这样一个红色结点的方式和处理一个新插入的红色结点完全相同，即继续把红链接转移到中结点上去。

总之，只要谨慎地使用左旋转，右旋转和颜色转换，我们就能够保证插入操作后红黑树和2-3树的一一对应关系。

在沿着插入点到根节点的路径向上移动时在所经过的每个结点中顺序完成以下操作，我们就能完成插入操作：

1，如果右子节点是红色的而左子节点是黑色的，进行左旋转；

2，如果左子节点是红色的且它的左子节点也是红色的，进行右旋转；

3，如果左右子节点均为红色，进行颜色转换。

**3.3.3 实现**

编程：红黑树的插入算法

**3.3.4 删除操作**

put()方法是本书中最复杂的实现之一，而红黑树的deleteMin(),deleteMax()和delete()的实现更麻烦。（待续）