4.4.2 双旋转

上面描述的算法有一个问题；如图 4-34 所示，对于情形 2 和 3 上面的做法无效。问题在于子树 Y 太深，单旋转没有减低它的深度。解决这个问题的双旋转在图 4-35 中表出。

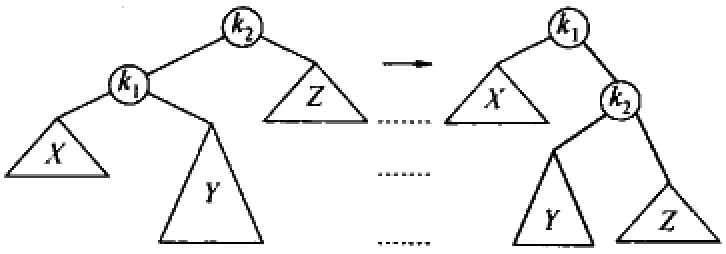


图 4-34 单旋转不能修复情形 2

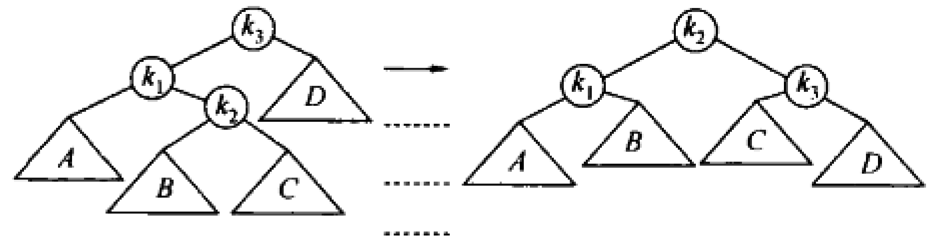


图 4-35 左－右双旋转修复情形 2

在图 4-34 中的子树 Y 已经有一项插入其中，这个事实保证它是非空的。因此，我们可以假设它有一个根和两棵子树。于是，我们可以把整棵树看成是 4 棵子树由 3 个节点连结。如图所示，恰好树 B 或树 C 中有一棵比 D 深两层(除非它们都是空的)，但是我们不能肯定是哪一棵。事实上这并不要紧，在图 4-35 中 B 和 C 都被画成比 D 低 $1\frac{1}{2}$层。

为了重新平衡，我们看到，不能再把 k_3 用作根了，而图 4-34 所示的在 k_3 和 k_1 之间的旋转又解决不了问题，唯一的选择就是把 k_2 用作新的根。这迫使 k_1 是 k_2 的左儿子，k_3 是它的右儿子，从而完全确定了这四棵树的最终位置。容易看出，最后得到的树满足 AVL 树的性质，与单旋转的情形一样，我们也把树的高度恢复到插入以前的水平，这就保证所有的重新平衡和高度更新是完善的。图 4-36 指出，对称情形 3 也可以通过双旋转得以修正。在这两种情形下，其效果与先在 α 的儿子和孙子之间旋转而后再在 α 和它的新儿子之间旋转的效果是相同的。

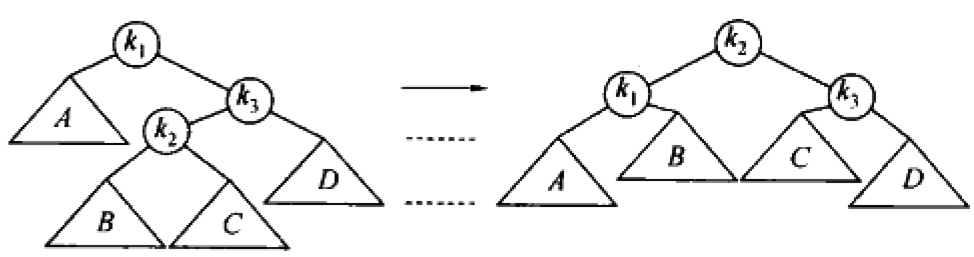
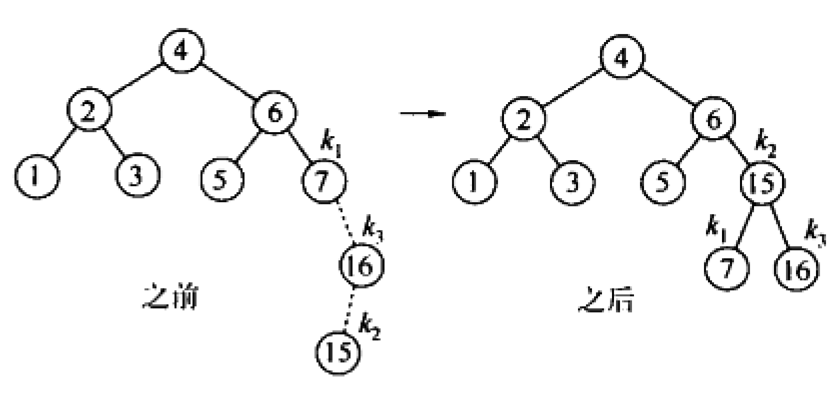


图 4-36 右－左双旋转修复情形 3

我们继续在前面例子的基础上以倒序插入关键字 10～16，接着插入 8，然后再插入 9。插入 16 容易，因为它并不破坏平衡性质，但是插入 15 就会引起在节点 7 处的高度不平衡。这属于情形 3，需要通过一次右－左双旋转来解决。在我们的例子中，这个右－左双旋转将涉及 7、16 和 15。此时，k_1 是含有项 7 的节点，k_3 是含有项 16 的节点，而 k_2 是含有项 15 的节点。子树 A、B、C 和 D 都是空树。



下面我们插入 14，它也需要一个双旋转。此时修复该树的双旋转还是右－左双旋转，它将涉及 6、15 和 7。在这种情况下，k_1 是含有项 6 的节点，k_2 是含有项 7 的节点，而 k_3 是含有项 15