1. 对 α 的左儿子的左子树进行一次插入。

2. 对 α 的左儿子的右子树进行一次插入。

3. 对 α 的右儿子的左子树进行一次插入。

4. 对 α 的右儿子的右子树进行一次插入。

情形 1 和 4 是关于 α 点的镜像对称，而 2 和 3 是关于 α 点的镜像对称。因此，理论上只有两种情况，当然从编程的角度来看还是四种情形。

第一种情况是插入发生在“外边”的情况(即左－左的情况或右－右的情况)，该情况通过对树的一次**单旋转**(single rotation)而完成调整。第二种情况是插入发生在“内部”的情形(即左－右的情况或右－左的情况)，该情况通过稍微复杂些的**双旋转**(double rotation)来处理。我们将会看到，这些都是对树的基本操作，它们多次用在一些平衡树算法中。本节其余部分将描述这些旋转，证明它们足以保持树的平衡，并顺便给出 AVL 树的一种非正式的实现。第 12 章将描述其他的平衡树方法，这些方法着眼于 AVL 树的更仔细的实现。

4.4.1　单旋转

图 4-31 显示了单旋转如何调整情形 1。旋转前的图在左边，而旋转后的图在右边。让我们来分析具体的做法。节点 k_2 不满足 AVL 平衡性质，因为它的左子树比右子树深 2 层(图中间的几条虚线标示树的各层)。该图所描述的情况只是情形 1 的一种可能的情况，在插入之前 k_2 满足 AVL 性质，但在插入之后这种性质被破坏了。子树 X 已经长出一层，这使得它比子树 Z 深出 2 层。Y 不可能与新 X 在同一水平上，因为那样 k_2 在插入以前就已经失去平衡了；Y 也不可能与 Z 在同一层上，因为那样 k_1 就会是在通向根的路径上破坏 AVL 平衡条件的第一个节点。

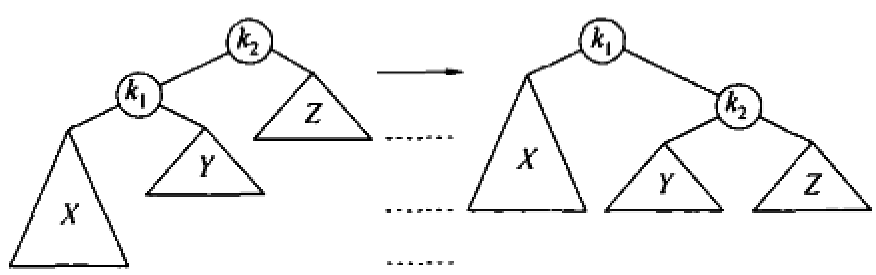

图 4-31　调整情形 1 的单旋转

为使树恢复平衡，我们把 X 上移一层，并把 Z 下移一层。注意，此时实际上超出了 AVL 性质的要求。为此，我们重新安排节点以形成一棵等价的树，如图 4-31 的第二部分所示。抽象地形容就是：把树形象地看成是柔软灵活的，抓住子节点 k_1，闭上你的双眼，使劲摇动它，在重力作用下，k_1 就变成了新的根。二叉查找树的性质告诉我们，在原树中 $k_2 > k_1$，于是在新树中 k_2 变成了 k_1 的右儿子，X 和 Z 仍然分别是 k_1 的左儿子和 k_2 的右儿子。子树 Y 包含原树中介于 k_1 和 k_2 之间的那些节点，可以将它放在新树中 k_2 的左儿子的位置上，这样，所有对顺序的要求都得到满足。

这样的操作只需要一部分的链改变，结果我们得到另外一棵二叉查找树，它是一棵 AVL 树，因为 X 向上移动了一层，Y 停在原来的水平上，而 Z 下移一层。k_2 和 k_1 不仅满足 AVL 要求，而且它们的子树都恰好处在同一高度上。不仅如此，整个树的新高度恰恰与插入前原树的高度相同，而插入操作却使得子树 X 长高了。因此，通向根节点的路径的高度不需要进一步的修正，因而也不需要进一步的旋转。图 4-32 显示了在将 6 插入左边原始的 AVL 树后节点 8 便不再平衡。于是，我们在 7 和 8 之间做一次单旋转，结果得到右边的树。

正如我们较早提到的，情形 4 代表一种对称的情形。图 4-33 指出单旋转如何使用。让我们演示一个更长一些的例子。假设从初始的空 AVL 树开始插入关键字 3、2 和 1，然后依序插入 4～