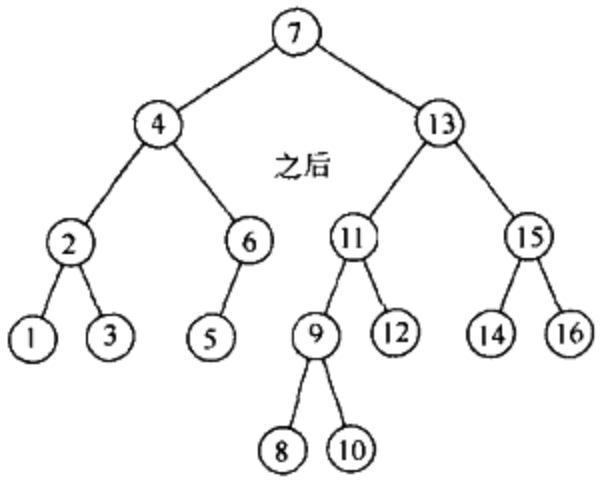

现在让我们对上面的讨论做个总结。除几种情形外，编程的细节是相当简单的。为将项是 X 的一个新节点插入到一棵 AVL 树 T 中去，我们递归地将 X 插入到 T 的相应的子树(称为 T_{LR})中。如果 T_{LR}的高度不变，那么插入完成。否则，如果在 T 中出现高度不平衡，则根据 X 以及 T 和 T_{LR}中的项做适当的单旋转或双旋转，更新这些高度(并解决好与树的其余部分的链接)，从而完成插入。由于一次旋转总能足以解决问题，因此仔细地编写非递归的程序一般说来要比编写递归程序快得多。然而，要想把非递归程序编写正确是相当困难的，因此许多编程人员还是用递归的方法实现 AVL 树。

另一个效率问题涉及到高度信息的存储。由于真正需要的实际上就是子树高度的差，应该保证它很小。如果我们真的尝试这种方法，可用两个二进制位(代表 +1、0、-1)表示这个差。这样将避免平衡因子的重复计算，但是却丧失某些简明性。最后的程序多多少少要比在每一个节点存储高度复杂。如果编写递归程序，那么速度恐怕不是主要考虑的问题。此时，通过存储平衡因子所得到的些微速度优势很难抵消清晰和相对简明性的损失。不仅如此，由于大部分机器存储的最小单位是 8 个二进制位，因此所用的空间量不可能有任何差别。一个 8 位的字节使我们存储高达 127 的绝对高度。既然树是平衡的，因此空间是足够的(见练习)。

有了上面的讨论，现在准备编写 AVL 树的一些例程。不过，这里我们只想做一部分工作，其余的联机提供。首先，我们需要 AvlNode 类，它在图 4-37 中给出。我们还需要一个快速的方法来返回节点的高度，这个方法必须处理 null 引用的麻烦情形。该程序在图 4-38 中给出。基本的插入例程写起来很容易，因为它主要由一些方法调用组成(见图 4-39)。

```
1     private static class AvlNode<AnyType>
2     {
3             // Constructors
4         AvlNode( AnyType theElement )
5           { this( theElement, null, null ); }
6
7         AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt )
8           { element  = theElement; left = lt; right = rt; height   = 0; }
9
10        AnyType           element;      // The data in the node
11        AvlNode<AnyType>  left;         // Left child
12        AvlNode<AnyType>  right;        //       child
13        int               height;       // Height
14    }
```

图 4-37　AVL 树的节点声明