的树增加了某些附加空间，但是，却比将重复信息放到树中要好(它将使树的深度变得很大)。当然，如果 compareTo 方法使用的关键字只是一个更大结构的一部分，那么这种方法行不通，此时我们可以把具有相同关键字的所有结构保留在一个辅助数据结构中，如表或是另一棵查找树。

图 4-22 显示插入例程的代码。由于 t 引用该树的根，而根又在第一次插入时变化，因此 insert 被写成一个返回对新树根的引用的方法。第 15 行和第 17 行递归地插入 x 到适当的子树中。

```
1     /**
2      * Internal method to insert into a subtree.
3      * @param x the item to insert.
4      * @param t the node that roots the subtree.
5      * @return the new root of the subtree.
6      */
7     private BinaryNode<AnyType> insert( AnyType x,BinaryNode<AnyType> t )
8     {
9         if( t == null )
10            return new BinaryNode<AnyType>( x, null, null );
11
12        int compareResult = x.compareTo( t.element );
13
14        if( compareResult < 0 )
15            t.left = insert( x, t.left );
16        else if( compareResult > 0 )
17            t.right = insert( x, t.right );
18        else
19            ;  // Duplicate; do nothing
20        return t;
21    }
```

图 4-22　将元素插入到二叉查找树的例程

4.3.4　remove 方法

正如许多数据结构一样，最困难的操作是 remove(删除)。一旦我们发现要被删除的节点，就需要考虑几种可能的情况。

如果节点是一片树叶，那么它可以被立即删除。如果节点有一个儿子，则该节点可以在其父节点调整自己的链以绕过该节点后被删除(为了清楚起见，我们将明确地画出链的指向)，见图 4-23。

复杂的情况是处理具有两个儿子的节点。一般的删除策略是用其右子树的最小的数据(很容易找到)代替该节点的数据并递归地删除那个节点(现在它是空的)。因为右子树中的最小的节点不可能有左儿子，所以第二次 remove 要容易。图 4-24 显示一棵初始的树及其中一个节点被删除后的结果。要被删除的节点是根的左儿子；其关键字是 2。它被其右子树中的最小数据 3 所代替，然后关键字是 3 的原节点如前例那样被删除。

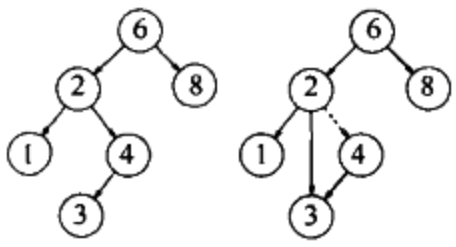

图 4-23　具有一个儿子的节点 4 删除前后的情况

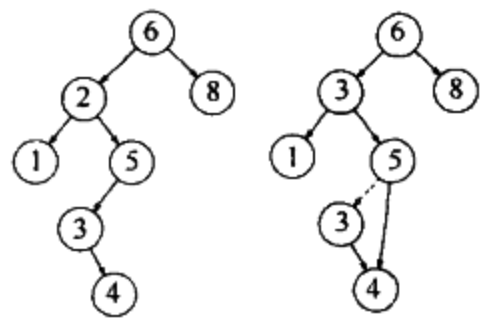

图 4-24　删除具有两个儿子的节点 2 前后的情况