这种递归是如此容易以至于许多程序设计员不厌其烦地使用它。我们用两种方法编写这两个例程，用递归编写 findMin 而用非递归编写 findMax(见图 4-20)。

```
1   /**
2    * Internal method to find the smallest item in a subtree.
3    * @param t the node that roots the subtree.
4    * @return node containing the smallest item.
5    */
6   private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
7   {
8       if( t == null )
9           return null;
10      else if( t.left == null )
11          return t;
12      return findMin( t.left );
13  }
14
15  /**
16   * Internal method to find the largest item in a subtree.
17   * @param t the node that roots the subtree.
18   * @return node containing the largest item.
19   */
20  private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
21  {
22      if( t != null )
23          while( t.right != null )
24              t = t.right;
25
26      return t;
27  }
```

图 4-20 对二叉查找树的 findMin 的递归实现和 findMax 的非递归实现

注意，我们是如何小心地处理空树的退化情况的。虽然这样做总是重要的，但是特别在递归程序中它尤其重要。此外，还要注意，在 findMax 中对 t 的改变是安全的，因为我们只用到引用的拷贝来进行工作。不管怎么说，还是应该随时特别小心，因为诸如 t.right = t.right .right 这样的语句将会产生一些变化。

4.3.3 insert 方法

进行插入操作的例程在概念上是简单的。为了将 X 插入到树 T 中，你可以像用 contains 那样沿着树查找。如果找到 X，则什么也不用做(或做一些“更新”)。否则，将 X 插入到遍历的路径上的最后一点上。图 4-21 显示实际的插入情况。为了插入 5，我们遍历该树就好像在运行 contains。在具有关键字 4 的节点处，我们需要向右行进，但右边不存在子树，因此 5 不在这棵树上，从而这个位置就是所要插入的位置。

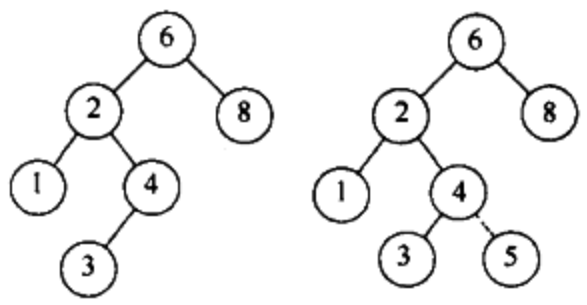

图 4-21 在插入 5 以前和以后的二叉查找树

重复元的插入可以通过在节点记录中保留一个附加域以指示发生的频率来处理。这对整个