注意测试的顺序。关键的问题是首先要对是否空树进行测试，否则我们就会生成一个企图通过 null 引用访问数据域的 NullPointerException 异常。剩下的测试应该使得最不可能的情况安排在最后进行。还要注意，这里的两个递归调用事实上都是尾递归并且可以用一个 while 循环很容易地代替。尾递归的使用在这里是合理的，因为算法表达式的简明性是以速度的降低为代价的，而这里所使用的栈空间的量也只不过是 $O(\log N)$而已。图 4-19 显示需要使用一个函数对象而不是要求这些项是 Comparable 的。它模仿 1.6 节的风格。

```
1  public class BinarySearchTree<AnyType>
2  {
3      private BinaryNode<AnyType> root;
4      private Comparator<? super AnyType> cmp;
5
6      public BinarySearchTree( )
7        { this( null ); }
8
9      public BinarySearchTree( Comparator<? super AnyType> c )
10       { root = null; cmp = c; }
11
12     private int myCompare( AnyType lhs, AnyType rhs )
13     {
14         if( cmp != null )
15             return cmp.compare( lhs, rhs );
16         else
17             return ((Comparable)lhs).compareTo( rhs );
18     }
19
20     private boolean contains( AnyType x, BinaryNode<AnyType> t )
21     {
22         if( t == null )
23             return false;
24
25         int compareResult = myCompare( x, t.element );
26
27         if( compareResult < 0 )
28             return contains( x, t.left );
29         else if( compareResult > 0 )
30             return contains( x, t.right );
31         else
32             return true;    // Match
33     }
34
35     // Remainder of class is similar with calls to compareTo replaced by myCompare
36 }
```

图 4-19　对使用函数对象实现二叉查找树的注释

4.3.2　findMin 方法和 findMax 方法

这两个 private 例程分别返回树中包含最小元和最大元的节点的引用。为执行 findMin，从根开始并且只要有左儿子就向左进行。终止点就是最小的元素。findMax 例程除分支朝向右儿子外其余过程相同。