有益的——当涉及递归时尤其重要，因此要注意这个例程声明树叶的高度为零，这是正确的。这种一般的遍历顺序叫做*后序遍历*，我们在前面也见到过。因为在每个节点的工作花费常数时间，所以总的运行时间也是 O(N)。

```
1     /**
2      * Print the tree contents in sorted order.
3      */
4     public void printTree( )
5     {
6         if( isEmpty( ) )
7             System.out.println( "Empty tree" );
8         else
9             printTree( root );
10    }
11
12    /**
13     * Internal method to print a subtree in sorted order.
14     * @param t the node that roots the subtree.
15     */
16    private void printTree( BinaryNode<AnyType> t )
17    {
18        if( t != null )
19        {
20            printTree( t.left );
21            System.out.println( t.element );
22            printTree( t.right );
23        }
24    }
```

图 4-56　按顺序打印二叉查找树的例程

```
1     /**
2      * Internal method to compute height of a subtree.
3      * @param t the node that roots the subtree.
4      */
5     private int height( BinaryNode<AnyType> t )
6     {
7         if( t == null )
8             return -1;
9         else
10            return 1 + Math.max( height( t.left ), height( t.right ));
11    }
```

图 4-57　使用后序遍历计算树的高度的例程

我们见过的第三种常用的遍历格式为**先序遍历**(preorder traversal)。这里，当前节点在其儿子节点之前处理。这种遍历是有用的。比如，如果要想用其深度标记每一个节点，那么这种遍历就会用到。

所有这些例程有一个共同的想法，即首先处理 null 的情形，然后才是其余的工作。注意，此处缺少一些附加的变量。这些例程仅仅传递对作为子树的根的节点的引用，并没有声明或是传递任何附加的变量。程序越紧凑，一些愚蠢的错误出现的可能就越少。第四种遍历用得很少，叫做**层序遍历**(level order traversal)，我们以前尚未见到过。在层序遍历中，所有深度为 d 的节点要在深度 d+1 的节点之前进行处理。层序遍历与其他类型的遍历不同的地方在于它不是递归地执