图 4-25 中的程序完成删除的工作，但它的效率并不高，因为它沿该树进行两趟搜索以查找和删除右子树中最小的节点。通过写一个特殊的 removeMin 方法可以容易地改变这种效率不高的缺点，我们这里将它略去只是为了简明。

```
/**
 * Internal method to remove from a subtree.
 * @param x the item to remove.
 * @param t the node that roots the subtree.
 * @return the new root of the subtree.
 */
private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return t;   // Item not found; do nothing

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = remove( x, t.left );
    else if( compareResult > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = remove( t.element, t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}
```

图 4-25　二叉查找树的删除例程

如果删除的次数不多，通常使用的策略是**懒惰删除**(lazy deletion)：当一个元素要被删除时，它仍留在树中，而只是被标记为删除。这特别是在有重复项时很常用，因为此时记录出现频率数的域可以减 1。如果树中的实际节点数和“被删除”的节点数相同，那么树的深度预计只上升一个小的常数(为什么?)，因此，存在一个与懒惰删除相关的非常小的时间损耗。再有，如果被删除的项是重新插入的，那么分配一个新单元的开销就避免了。

4.3.5 平均情况分析

直观上，我们期望前一节所有的操作都花费 $O(\log N)$时间，因为我们用常数时间在树中降低了一层，这样一来，对其进行操作的树大致减小一半左右。因此，所有操作的运行时间都是 $O(d)$，其中 d 是包含所访问的项的节点的深度。

我们在本节要证明，假设所有的插入序列都是等可能的，则树的所有节点的平均深度为 $O(\log N)$。

一棵树的所有节点的深度的和称为**内部路径长**(internal path length)。我们现在将要计算二叉查找树平均内部路径长，其中的平均是对向二叉查找树中所有可能的插入序列进行的。

令 $D(N)$是具有 N 个节点的某棵树 T 的内部路径长，$D(1)=0$。一棵 N 节点树由一棵 i 节点左子树和一棵$(N-i-1)$-节点右子树以及深度 0 处的一个根节点组成，其中 $0\leqslant i<N$，