最后的例程是插入。正如分离链接散列方法那样，若x已经存在，则我们就什么也不做。有些工作也只是简单的修正。否则，我们就把要插入的元素放在findPos例程指出的地方。程序在图5-17中显示。如果装填因子超过0.5，则表是满的，需要将该散列表放大。这称为再散列(rehashing)，我们将在5.5节进行讨论。

```
1       /**
2        * Insert into the hash table. If the item is
3        * already present, do nothing.
4        * @param x the item to insert.
5        */
6       public void insert( AnyType x )
7       {
8               // Insert x as active
9           int currentPos = findPos( x );
10          if( isActive( currentPos ) )
11              return;
12
13          array[ currentPos ] = new HashEntry<AnyType>( x, true );
14
15              // Rehash; see Section 5.5
16          if( ++currentSize > array.length / 2 )
17              rehash( );
18      }
19
20      /**
21       * Remove from the hash table.
22       * @param x the item to remove.
23       */
24      public void remove( AnyType x )
25      {
26          int currentPos = findPos( x );
27          if( isActive( currentPos ) )
28              array[ currentPos ].isActive = false;
29      }
```

图5-17 使用平方探测散列表的insert例程

虽然平方探测排除了一次聚集，但是散列到同一位置上的那些元素将探测相同的备选单元。这叫做**二次聚集**(secondary clustering)。二次聚集是理论上的一个小缺憾。模拟结果指出，对每次查找，它一般要引起另外的少于一半的探测。下面的技术将会排除这个缺憾，不过这要付出计算一个附加的散列函数的代价。

5.4.3 双散列

我们将要考察的最后一个冲突解决方法是双散列(double hashing)。对于双散列，一种流行的选择是$f(i)=i\cdot hash_2(x)$。这个公式是说，我们将第二个散列函数应用到x并在距离$hash_2(x)$，$2hash_2(x)$，…等处探测。$hash_2(x)$选择得不好将会是灾难性的。例如，若把99插入到前面例子中的输入中去，则通常的选择$hash_2(x)=x \bmod 9$将不起作用。因此，函数一定不要算得0值。另外，保证所有的单元都能被探测到也是很重要的(但在下面的例子中这是不可能的，因为表的大小不是素数)。诸如$hash_2(x)=R-(x \bmod R)$这样的函数将起到良好的作用，其中R为