匹配的节点再加上0个或更多其他的节点。在 N 个元素的散列表以及 M 个链表中"其他节点"的期望个数为 $(N-1)/M = \lambda - 1/M$，它基本上就是 λ，因为假设 M 是大的。平均看来，一半的"其他"节点被搜索到，再结合匹配节点，我们得到 $1 + \lambda/2$ 个节点的平均查找代价。这个分析指出，散列表的大小实际上并不重要，而装填因子才是重要的。分离链接散列法的一般法则是使得表的大小与预料的元素个数大致相等(换句话说，让 $\lambda \approx 1$)。在图5-10的程序中，如果装填因子超过1，那么我们通过调用在26行上的rehash函数扩大散列表的大小。rehash将在5.5节讨论。正如前面提到的，使表的大小是素数以保证一个好的分布，这个想法很好。

5.4 不用链表的散列表

分离链接散列算法的缺点是使用一些链表。由于给新单元分配地址需要时间(特别是在其他语言中)，因此这就导致算法的速度有些减慢，同时算法实际上还要求对第二种数据结构的实现。另有一种不用链表解决冲突的方法是尝试另外一些单元，直到找出空的单元为止。更常见的是，单元 $h_0(x)$，$h_1(x)$，$h_2(x)$，…相继被试选，其中 $h_i(x) = (hash(x) + f(i)) \bmod TableSize$，且 $f(0) = 0$。函数 f 是冲突解决方法。因为所有的数据都要置入表内，所以这种解决方案所需要的表要比分离链接散列的表大。一般说来，对于不使用分离链接的散列表来说，其装填因子应该低于 $\lambda = 0.5$。我们把这样的表叫做**探测散列表**(probing hash table)。现在我们就来考察三种通常的冲突解决方案。

5.4.1 线性探测法

在线性探测法中，函数 f 是 i 的线性函数，典型情形是 $f(i) = i$。这相当于相继探测逐个单元(必要时可以回绕)以查找出一个空单元。图5-11显示使用与前面相同的散列函数将各个关键字{89，18，49，58，69}插入到一个散列表中的情况，而此时的冲突解决方法就是 $f(i) = i$。

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

图5-11 每次插入后使用线性探测得到的散列表

第一个冲突在插入关键字49时产生；它被放入下一个空闲地址，即地址0，该地址是开放的。关键字58先与18冲突，再与89冲突，然后又和49冲突，试选三次之后才找到一个空单元。对69的冲突用类似的方法处理。只要表足够大，总能够找到一个自由单元，但是如此花费的时间是相当多的。更糟的是，即使表相对较空，这样占据的单元也会开始形成一些区块，其结果称为**一次聚集**(primary clustering)，就是说，散列到区块中的任何关键字都需要多次试选单元才能够解决冲突，然后该关键字被添加到相应的区块中。

虽然我们不在这里进行具体计算，但是可以证明，使用线性探测的预期探测次数对于插入和不成功的查找来说大约为 $\frac{1}{2}(1 + 1/(1-\lambda)^2)$，而对于成功的查找来说则是 $\frac{1}{2}(1 + 1/(1-\lambda))$。相关的一些计算多少有些复杂。从程序中容易看出，插入和不成功查找需要相同次数的探测。略加思考不难得出，成功查找应该比不成功查找平均花费较少的时间。