其中 $hash(x)$为 32 比特散列值(尚未化成适当的数组下标)，而 $r(y) = |48\,271\ y(\mathrm{mod}\ (2^{31}-1))|\mathrm{mod}\ TableSize$。(10.4.1 节描述一种执行这种计算而不溢出的方法，不过，在这种情况下溢出是不可能的)。解释为什么这种方法趋向于避免二次聚集，并将这种方法与双散列及平方探测进行比较。

5.10 再散列要求对散列表中的所有项重新计算散列函数。由于计算散列函数开销巨大，因此设对象提供它们自己的散列成员函数，而每个对象在散列函数第 1 次被计算时都把结果存入一个附加的数据成员中。指出这种方案如何用于图 5-8 中的 Employee 类，并解释在什么情况下这些所记忆的散列值在每个 Employee 中仍然有效。

5.11 编写一个程序，实现下面的方案，将大小分别为 M 和 N 的两个稀疏多项式(sparse polynomial)P_1 和 P_2 相乘。每个多项式表示成为对象的一个链表，这些对象由系数和幂组成(练习 3.12)。我们用 P_2 的项乘以 P_1 的每一项，总数为 MN 次运算。一种方法是将这些项排序并合并同类项，但是，这需要排序 MN 个记录，代价可能很高，特别是在小内存环境下。另一种方案，我们可在多项式的项进行计算时将它们合并，然后将结果排序。

a. 编写一个程序实现第二种方案。

b. 如果输出多项式大约有 $O(M+N)$项，两种方法的运行时间各是多少?

*5.12 描述一个避免初始化散列表的过程(以内存消耗为代价)。

5.13 设欲找出在长输入串 $A_1A_2\cdots A_N$ 中串 $P_1P_2\cdots P_k$ 的第一次出现。我们可以通过散列模式串(pattern string)得到一个散列值 H_p，并将该值与从 $A_1A_2\cdots A_k$，$A_2A_3\cdots A_{k+1}$，$A_3A_4\cdots A_{k+2}$，等等直到 $A_{N-k+1}A_{N-k+2}\cdots A_N$ 形成的散列值比较来解决这个问题。如果得到散列值的一个匹配，那么再逐个字符地对串进行比较以检验这个匹配。如果串实际上确实匹配，那么返回其(在 A 中的)位置，而在匹配失败这种不大可能的情况下继续进行查找。

* a. 证明如果 $A_{i+1}\cdots A_{i+k-1}$的散列值已知，那么 $A_{i+1}A_{i+2}\cdots A_{i+k}$的散列值可以以常数时间算出。

b. 证明运行时间为 $O(k+N)$加上排除错误匹配所耗费的时间。

* c. 证明错误匹配的期望次数是微不足道的。

d. 编写一个程序实现该算法。

** e. 描述一个算法，其最坏情形的运行时间为 $O(k+N)$。

** f. 描述一个算法，其平均运行时间为 $O(N/k)$。

5.14 一个(老式的)BASIC 程序由一系列按递增顺序编号的语句组成。控制是通过使用 goto 或 gosub 和一个语句编号实现的。编写一个程序读进合法的 BASIC 程序并给语句重新编号，使得第一句在序号 F 处开始并且每一个语句的序号比前一语句高 D。可以假设 N 条语句的一个上限，但是在输入中语句序号可以大到 32 比特的整数。所编的程序必须以线性时间运行。

5.15 a. 利用本章末尾描述的算法实现字谜程序。

b. 通过存储每一个单词 W 以及 W 的所有前缀，可以大大加快运行速度(如果 W 的一个前缀刚好是词典中的一个单词，那么就把它作为实际的单词来储存)。虽然这看起来极大地增加了散列表的大小，但实际上并非如此，因为许多单词有相同的前缀。当以某个特定的方向执行一次扫描的时候，如果被查找的单词甚至作为前缀都不在散列表中，那么在这个方向上的扫描可以及早终止。利用这种想法编写一个改进的程序来解决字谜游戏问题。