散列表第三种常见的用途是在游戏程序中。当程序搜索游戏不同的行时，它跟踪通过计算基于位置的散列函数而看到的一些位置(并把对于该位置的移动存储起来)。如果同样的位置再出现，程序通常通过移动的简单变换来避免昂贵的重复计算。所有游戏程序的这种一般特点叫做**变换表**(tranposition table)。

散列的另一个用途是在线拼写检验程序。如果错拼检测(与正确性相比)重要，那么整个词典可以预先被散列，单词则可以常数时间被检测。散列表很适合这项工作，因为以字母顺序排列单词并不重要；而以它们在文件中出现的顺序显示出错误拼写当然是可接受的。

我们以第一章的字谜问题来结束本章。如果使用第一章中描述的第二个算法，并且假设最大单词的大小是某个小常数，那么读入包含 W 个单词的词典并把它放入散列表的时间是 $O(W)$。这个时间很可能由磁盘 I/O 而不是由那些散列例程起支配作用。算法的其余部分将对每一个四元组(行，列，方向，字符数)测试一个单词是否出现。由于每次查询时间为 $O(1)$，而只存在常数个数的方向(8)和每个单词的字符，因此这一阶段的运行时间为 $O(R \cdot C)$。总的运行时间是 $O(R \cdot C + W)$，它是对原始 $O(R \cdot C \cdot W)$的明显的改进。我们还可以做进一步的优化，它能够降低实际的运行时间；这些将在练习中描述。

练习

5.1　给定输入{4 371，1 323，6 173，4 199，4 344，9 679，1 989}和散列函数 $h(x) = x \pmod{10}$，指出下列结果：

a. 分离链接散列表。

b. 使用线性探测的散列表。

c. 使用平方探测的散列表。

d. 第二个散列函数为 $h_2(x) = 7 - x \pmod 7$的散列表。

5.2　指出将练习 5.1 中的散列表再散列的结果。

5.3　编写一个程序，计算使用线性探测、平方探测、双散列时的长随机插入序列中所需的冲突次数。

5.4　在分离链接散列表中进行大量的删除可能造成表非常稀疏，浪费空间。在这种情况下，可以再散列一个表，为原表的一半大。设当存在相当于表的大小的二倍的元素的时候，我们再散列到一个更大的表。该表应该有多么稀疏才能再散列到一个更小的表?

5.5　平方探测的 `isEmpty` 例程还没有写出。你能通过返回表达式 `currentSize == 0` 实现它吗?

5.6　在平方探测散列表中，设我们把一个新项插入到搜索路径上第一个非活动的单元而不是把它插入到由 `findPos` 指定的位置(这样，能够重新声明一个标记"deleted"的单元，潜在地节省了空间)。

a. 使用上述分析重新编写插入算法。通过使用一个附加变量让 `findPos` 保留它遇到的第一个非活动单元的位置来完成重写的工作。

b. 解释使得重写的算法快于原来算法的环境。重写的算法可能会更慢吗?

5.7　图 5-4 中的散列函数在 `for` 循环中对 `key.length()`进行重复调用。每次进入循环以前对它进行一次计算值得吗?

5.8　各种冲突解决方案的优点和缺点是什么?

5.9　假设为了减轻二次聚集的影响，我们使用函数 $f(i) = i \cdot r(hash(x))$作为冲突解决方案，