第 11 章

Introduction to Algorithms, Third Edition

散　列　表

许多应用都需要一种动态集合结构，它至少要支持 INSERT、SEARCH 和 DELETE 字典操作。例如，用于程序语言编译的编译器维护了一个符号表，其中元素的关键字为任意字符串，它与程序中的标识符相对应。散列表(hash table)是实现字典操作的一种有效数据结构。尽管最坏情况下，散列表中查找一个元素的时间与链表中查找的时间相同，达到了 $\Theta(n)$。然而在实际应用中，散列查找的性能是极好的。在一些合理的假设下，在散列表中查找一个元素的平均时间是 $O(1)$。

散列表是普通数组概念的推广。由于对普通数组可以直接寻址，使得能在 $O(1)$时间内访问数组中的任意位置。11.1 节将更详细地讨论直接寻址。如果存储空间允许，我们可以提供一个数组，为每个可能的关键字保留一个位置，以利用直接寻址技术的优势。

当实际存储的关键字数目比全部的可能关键字总数要小时，采用散列表就成为直接数组寻址的一种有效替代，因为散列表使用一个长度与实际存储的关键字数目成比例的数组来存储。在散列表中，不是直接把关键字作为数组的下标，而是根据关键字计算出相应的下标。11.2 节介绍这种技术的主要思想，着重介绍通过“链接”(chaining)方法解决“冲突”(collision)。所谓冲突，就是指多个关键字映射到数组的同一个下标。11.3 节介绍如何利用散列函数根据关键字计算出数组的下标。另外，还将介绍和分析散列技术的几种变形。11.4 节介绍“开放寻址法”(open addressing)，它是处理冲突的另一种方法。散列是一种极其有效和实用的技术：基本的字典操作平均只需要 $O(1)$的时间。11.5 节介绍当关键字集合是静态存储(即关键字集合一旦存入后就不再改变)时，“完全散列”(perfect hashing)如何能够在 $O(1)$的最坏情况时间内完成关键字查找。

253

11.1　直接寻址表

当关键字的全域 U 比较小时，直接寻址是一种简单而有效的技术。假设某应用要用到一个动态集合，其中每个元素都是取自于全域 $U=\{0, 1, \cdots, m-1\}$中的一个关键字，这里 m 不是一个很大的数。另外，假设没有两个元素具有相同的关键字。

为表示动态集合，我们用一个数组，或称为**直接寻址表**(direct-address table)，记为 $T[0..m-1]$。其中每个位置，或称为**槽**(slot)，对应全域 U 中的一个关键字。图 11-1 描绘了该方法。槽 k 指向集合中一个关键字为 k 的元素。如果该集合中没有关键字为 k 的元素，则 $T[k]=$NIL。

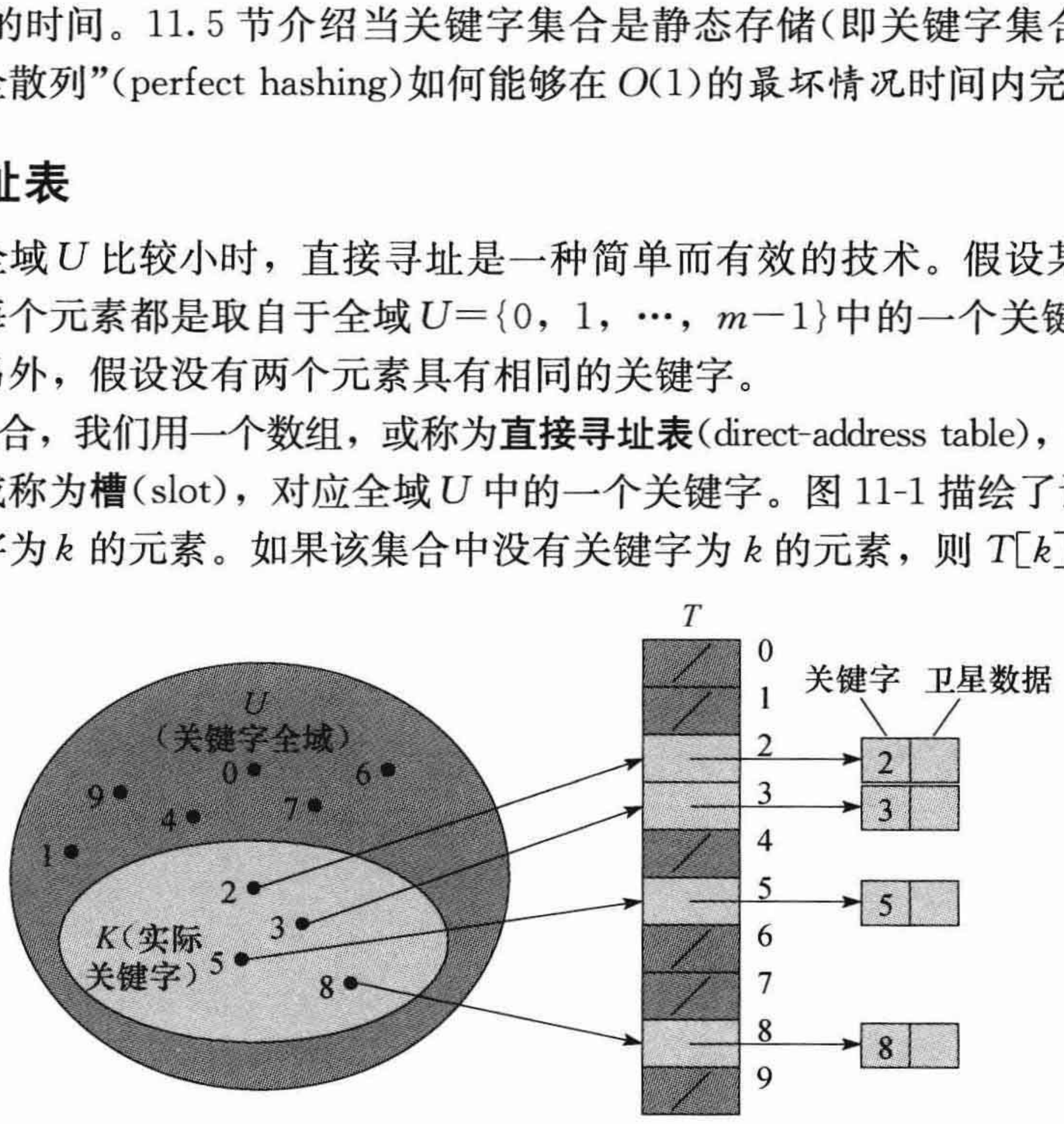

图 11-1　如何用一个直接寻址表 T 来实现动态集合。全域 $U=\{0, 1, \cdots, 9\}$中的每个关键字都对应于表中的一个下标值。由实际关键字构成的集合 $K=\{2, 3, 5, 8\}$决定表中的一些槽，这些槽包含指向元素的指针。而另一些槽包含 NIL，用深阴影表示

几个字典操作实现起来比较简单：

```
DIRECT-ADDRESS-SEARCH(T,k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T,x)
1  T[x.key]=x

DIRECT-ADDRESS-DELETE(T,x)
1  T[x.key]=NIL
```

上述的每一个操作都只需 $O(1)$ 时间。 254

对于某些应用，直接寻址表本身就可以存放动态集合中的元素。也就是说，并不把每个元素的关键字及其卫星数据都放在直接寻址表外部的一个对象中，再由表中某个槽的指针指向该对象，而是直接把该对象存放在表的槽中，从而节省了空间。我们使用对象内的一个特殊关键字来表明该槽为空槽。而且，通常不必存储该对象的关键字属性，因为如果知道一个对象在表中的下标，就可以得到它的关键字。然而，如果不存储关键字，我们就必须有某种方法来确定某个槽是否为空。

练习

11.1-1 假设一动态集合 S 用一个长度为 m 的直接寻址表 T 来表示。请给出一个查找 S 中最大元素的过程。你所给的过程在最坏情况下的运行时间是多少？

11.1-2 **位向量**(bit vector)是一个仅包含 0 和 1 的数组。长度为 m 的位向量所占空间要比包含 m 个指针的数组少得多。请说明如何用一个位向量来表示一个包含不同元素(无卫星数据)的动态集合。字典操作的运行时间应为 $O(1)$。

11.1-3 试说明如何实现一个直接寻址表，表中各元素的关键字不必都不相同，且各元素可以有卫星数据。所有三种字典操作(INSERT、DELETE 和 SEARCH)的运行时间应为 $O(1)$。(不要忘记 DELETE 要处理的是被删除对象的指针变量，而不是关键字。)

***11.1-4** 我们希望在一个非常大的数组上，通过利用直接寻址的方式来实现一个字典。开始时，该数组中可能包含一些无用信息，但要对整个数组进行初始化是不太实际的，因为该数组的规模太大。请给出在大数组上实现直接寻址字典的方案。每个存储对象占用 $O(1)$ 空间；SEARCH、INSERT 和 DELETE 操作的时间均为 $O(1)$；并且对数据结构初始化的时间为 $O(1)$。(提示：可以利用一个附加数组，处理方式类似于栈，其大小等于实际存储在字典中的关键字数目，以帮助确定大数组中某个给定的项是否有效。) 255

11.2 散列表

直接寻址技术的缺点是非常明显的：如果全域 U 很大，则在一台标准的计算机可用内存容量中，要存储大小为 $|U|$ 的一张表 T 也许不太实际，甚至是不可能的。还有，实际存储的关键字集合 K 相对 U 来说可能很小，使得分配给 T 的大部分空间都将浪费掉。

当存储在字典中的关键字集合 K 比所有可能的关键字的全域 U 要小许多时，散列表需要的存储空间要比直接寻址表少得多。特别地，我们能将散列表的存储需求降至 $\Theta(|K|)$，同时散列表中查找一个元素的优势仍得到保持，只需要 $O(1)$ 的时间。问题是这个界是针对平均情况时间的，而对直接寻址来说，它是适用于最坏情况时间的。

在直接寻址方式下，具有关键字 k 的元素被存放在槽 k 中。在散列方式下，该元素存放在槽 $h(k)$ 中；即利用**散列函数**(hash function) h，由关键字 k 计算出槽的位置。这里，函数 h 将关键字

的全域U映射到**散列表**(hash table)$T[0..m-1]$的槽位上：

$$h:U \to \{0,1,\cdots,m-1\}$$

这里散列表的大小m一般要比$|U|$小得多。我们可以说一个具有关键字k的元素被**散列**到槽$h(k)$上，也可以说$h(k)$是关键字k的**散列值**。图11-2描述了这个基本方法。散列函数缩小了数组下标的范围，即减小了数组的大小，使其由$|U|$减小为m。

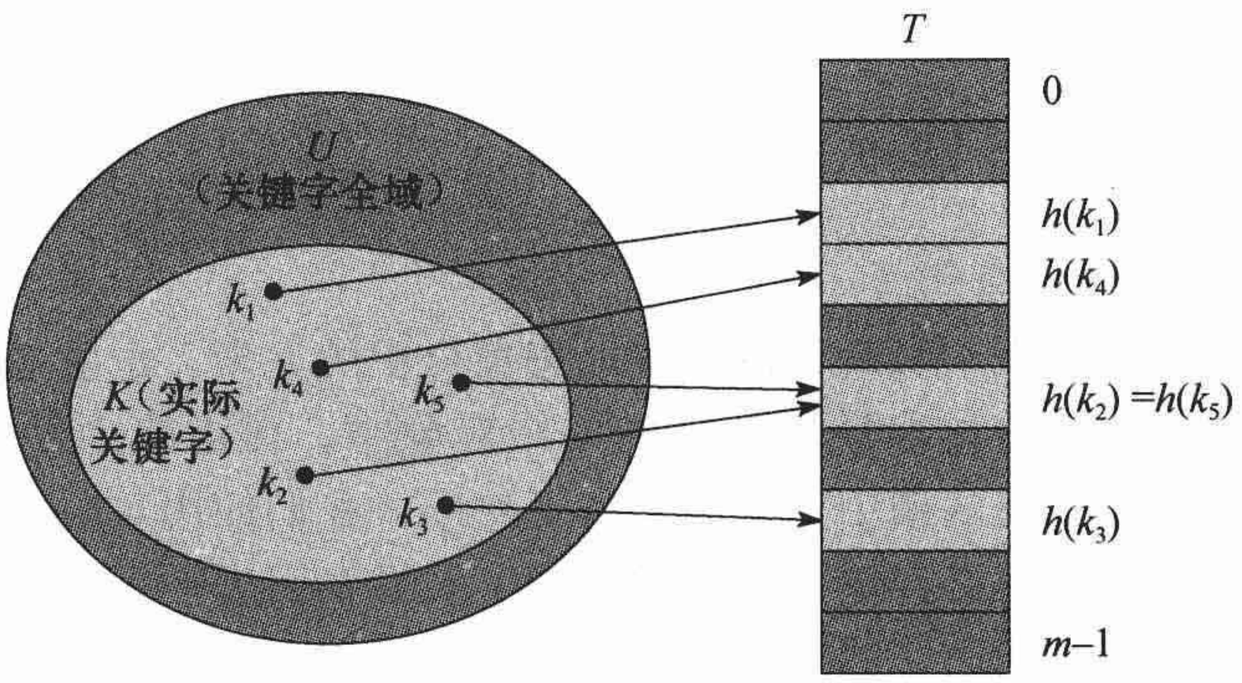

图11-2 用一个散列函数h将关键字映射到散列表的槽中，关键字k_2和k_5映射到同一个槽中，因而产生了冲突

256

这里存在一个问题：两个关键字可能映射到同一个槽中。我们称这种情形为**冲突**(collision)。幸运的是，我们能找到有效的方法来解决冲突。

当然，理想的解决方法是避免所有的冲突。我们可以试图选择一个合适的散列函数h来做到这一点。一个想法就是使h尽可能的“随机”，从而避免冲突或者使冲突的次数最小化。实际上，术语“散列”原意就是随机混杂和拼凑，即体现了这种思想。(当然，一个散列函数h必须是确定的，因为某一个给定的输入k应始终产生相同的结果$h(k)$。)但是，由于$|U|>m$，故至少有两个关键字其散列值相同，所以要想完全避免冲突是不可能的。因此，我们一方面可以通过精心设计的散列函数来尽量减少冲突的次数，另一方面仍需要有解决可能出现冲突的办法。

本节余下的部分要介绍一种最简单的冲突解决方法，称为**链接法**(chaining)。11.4节还要介绍另一种冲突解决方法，称为**开放寻址法**(open addressing)。

通过链接法解决冲突

在链接法中，把散列到同一槽中的所有元素都放在一个链表中，如图11-3所示。槽j中有一个指针，它指向存储所有散列到j的元素的链表的表头；如果不存在这样的元素，则槽j中为NIL。

在采用链接法解决冲突后，散列表T上的字典操作就很容易实现。

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

插入操作的最坏情况运行时间为$O(1)$。插入过程在某种程度上要快一些，因为假设待插入的元素x没有出现在表中；如果需要，可以在插入前执行一个搜索来检查这个假设(需付出额外代价)。查找操作的最坏情况运行时间与表的长度成正比。下面还将对此操作进行更详细的分析。

如图 11-3 所示，如果散列表中的链表是双向链接的，则删除一个元素 x 的操作可以在 $O(1)$时间内完成。(注意到，CHAINED-HASH-DELETE 以元素 x 而不是它的关键字 k 作为输入，所以无需先搜索 x。如果散列表支持删除操作，则为了能够更快地删除某一元素，应该将其链表设计为双向链接的。如果表是单链接的，则为了删除元素 x，我们首先必须在表 $T[h(x.key)]$中找到元素 x，然后通过更改 x 前驱元素的 $next$ 属性，把 x 从链表中删除。在单链表情况下，删除和查找操作的渐近运行时间相同。)

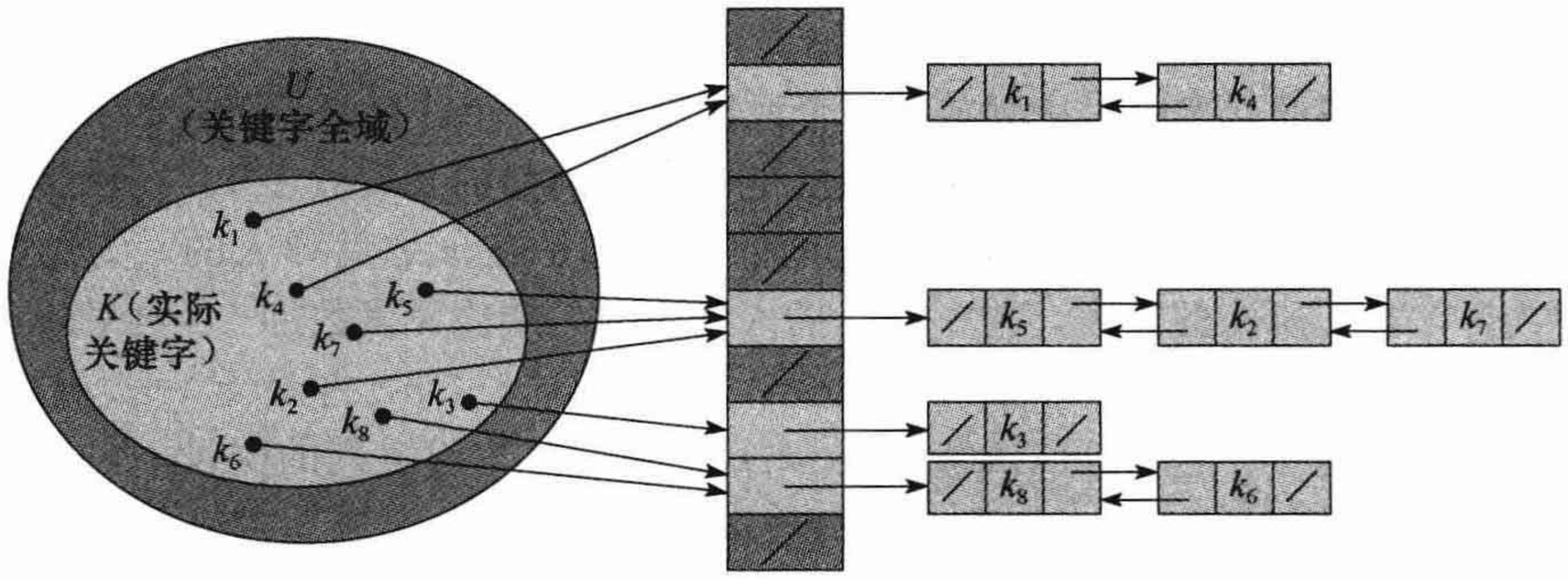

图 11-3　通过链接法解决冲突。每个散列表槽 $T[j]$都包含一个链表，其中所有关键字的散列值均为 j。例如，$h(k_1)=h(k_4)$，还有 $h(k_5)=h(k_7)=h(k_2)$。这个链表可能是单链表，也可能是双向链表；图中链表画为双链，因为删除操作比较快

链接法散列的分析

采用链接法后散列的性能怎么样呢？特别地，要查找一个具有给定关键字的元素需要多长时间呢？

给定一个能存放 n 个元素的、具有 m 个槽位的散列表 T，定义 T 的**装载因子**(load factor)α 为 n/m，即一个链的平均存储元素数。我们的分析将借助 α 来说明，α 可以小于、等于或大于 1。

用链接法散列的最坏情况性能很差：所有的 n 个关键字都散列到同一个槽中，从而产生出一个长度为 n 的链表。这时，最坏情况下查找的时间为 $\Theta(n)$，再加上计算散列函数的时间，如此就和用一个链表来链接所有的元素差不多了。显然，并不是因为散列表的最坏情况性能差，就不使用它。(11.5 节中介绍的完全散列能够在关键字集合为静态时，提供比较好的最坏情况性能。)

257
~
258

散列方法的平均性能依赖于所选取的散列函数 h，将所有的关键字集合分布在 m 个槽位上的均匀程度。11.3 节将讨论这些问题，现在我们先假定任何一个给定元素等可能地散列到 m 个槽中的任何一个，且与其他元素被散列到什么位置上无关。我们称这个假设为**简单均匀散列**(simple uniform hashing)。

对于 $j=0, 1, \cdots, m-1$，列表 $T[j]$的长度用 n_j 表示，于是有

$$n = n_0 + n_1 + \cdots + n_{m-1} \tag{11.1}$$

并且 n_j 的期望值为 $E[n_j]=\alpha=n/m$。

假定可以在 $O(1)$时间内计算出散列值 $h(k)$，从而查找关键字为 k 的元素的时间线性地依赖于表 $T[h(k)]$的长度 $n_{h(k)}$。先不考虑计算散列函数和访问槽 $h(k)$的 $O(1)$时间，我们来看看查找算法查找元素的期望数，即为比较元素的关键字是否为 k 而检查的表 $T[h(k)]$中的元素数。分两种情况来考虑。在第一种情况中，查找不成功：表中没有一个元素的关键字为 k。在第二种情况中，成功地查找到关键字为 k 的元素。

定理 11.1　*在简单均匀散列的假设下，对于用链接法解决冲突的散列表，一次不成功查找的平均时间为 $\Theta(1+\alpha)$。*

证明　在简单均匀散列的假设下，任何尚未被存储在表中的关键字 k 都等可能地被散列到 m

个槽中的任何一个。因而，当查找一个关键字 k 时，在不成功的情况下，查找的期望时间就是查找至链表 $T[h(k)]$ 末尾的期望时间，这一时间的期望长度为 $\mathrm{E}[n_{h(k)}]=\alpha$。于是，一次不成功的查找平均要检查 α 个元素，并且所需要的总时间(包括计算 $h(k)$ 的时间)为 $\Theta(1+\alpha)$。■

对于成功的查找来说，情况略有不同，这是因为每个链表并不是等可能地被查找到的。替代的是，某个链表被查找到的概率与它所包含的元素数成正比。然而，期望的查找时间仍然是 $\Theta(1+\alpha)$。

定理 11.2 在简单均匀散列的假设下，对于用链接法解决冲突的散列表，一次成功查找所需的平均时间为 $\Theta(1+\alpha)$。

证明 假定要查找的元素是表中存放的 n 个元素中任何一个，且是等可能的。在对元素 x 的
259 一次成功查找中，所检查的元素数就是 x 所在的链表中 x 前面的元素数多 1。在该链表中，因为新的元素都是在表头插入的，所以出现在 x 之前的元素都是在 x 之后插入的。为了确定所检查元素的期望数目，对 x 所在的链表，在 x 之后插入到表中的期望元素数加 1，再对表中的 n 个元素 x 取平均。设 x_i 表示插入到表中的第 i 个元素，$i=1, 2, \cdots, n$，并设 $k_i=x_i.key$。对关键字 k_i 和 k_j，定义指示器随机变量 $X_{ij}=\mathrm{I}\{h(k_i)=h(k_j)\}$。在简单均匀散列的假设下，有 $\Pr\{h(k_i)=h(k_j)\}=1/m$，从而根据引理 5.1，有 $\mathrm{E}[X_{ij}]=1/m$。于是，在一次成功的查找中，所检查元素的期望数目为

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \qquad \text{(由期望的线性性)}$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \qquad \text{(由等式(A.1))}$$

$$=1+\frac{n-1}{2m}=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

因此，一次成功的查找所需要的全部时间(包括计算散列函数的时间)为 $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$。■

上面的分析意味着什么呢？如果散列表中槽数至少与表中的元素数成正比，则有 $n=O(m)$，从而 $\alpha=n/m=O(m)/m=O(1)$。所以，查找操作平均需要常数时间。当链表采用双向链接时，插入操作在最坏情况下需要 $O(1)$ 时间，删除操作最坏情况下也需要 $O(1)$ 时间，因而，全部的字
260 典操作平均情况下都可以在 $O(1)$ 时间内完成。

练习

11.2-1 假设用一个散列函数 h 将 n 个不同的关键字散列到一个长度为 m 的数组 T 中。假设采用的是简单均匀散列，那么期望的冲突数是多少？更准确地，集合 $\{\{k, l\}: k\neq l$，且 $h(k)=h(l)\}$ 基的期望值是多少？

11.2-2 对于一个用链接法解决冲突的散列表，说明将关键字 5,28,19,15,20,33,12,17,10 插入到该表中的过程。设该表中有 9 个槽位，并设其散列函数为 $h(k)=k \bmod 9$。

11.2-3 Marley 教授做了这样一个假设，即如果将链模式改动一下，使得每个链表都能保持已排好序的顺序，散列的性能就可以有较大的提高。Marley 教授的改动对成功查找、不成功查找、插入和删除操作的运行时间有何影响？

11.2-4 说明在散列表内部，如何通过将所有未占用的槽位链接成一个自由链表，来分配和释放元素所占的存储空间。假定一个槽位可以存储一个标志、一个元素加上一个或两个指针。所有的字典和自由链表操作均应具有 $O(1)$ 的期望运行时间。该自由链表需要是双

向链表吗？或者，是不是单链表就足够了呢？

11.2-5　假设将一个具有 n 个关键字的集合存储到一个大小为 m 的散列表中。试说明如果这些关键字均源于全域 U，且 $|U|>nm$，则 U 中还有一个大小为 n 的子集，其由散列到同一槽位中的所有关键字构成，使得链接法散列的查找时间最坏情况下为 $\Theta(n)$。

11.2-6　假设将 n 个关键字存储到一个大小为 m 且通过链接法解决冲突的散列表中，同时已知每条链的长度，包括其中最长链的长度 L，请描述从散列表的所有关键字中均匀随机地选择某一元素并在 $O(L\cdot(1+1/\alpha))$ 的期望时间内返回该关键字的过程。 261

11.3　散列函数

本节将讨论一些关于如何设计好的散列函数的问题，并介绍三种具体方法。其中的两种方法(用除法进行散列和用乘法进行散列)本质上属于启发式方法，而第三种方法(全域散列)则利用了随机技术来提供可证明的良好性能。

好的散列函数的特点

一个好的散列函数应(近似地)满足简单均匀散列假设：每个关键字都被等可能地散列到 m 个槽位中的任何一个，并与其他关键字已散列到哪个槽位无关。遗憾的是，一般无法检查这一条件是否成立，因为很少能知道关键字散列所满足的概率分布，而且各关键字可能并不是完全独立的。

有时，我们知道关键字的概率分布。例如，如果各关键字都是随机的实数 k，它们独立均匀地分布于 $0\leqslant k<1$ 范围中，那么散列函数

$$h(k)=\lfloor km\rfloor$$

就能满足简单均匀散列的假设条件。

在实际应用中，常常可以运用启发式方法来构造性能好的散列函数。设计过程中，可以利用关键字分布的有用信息。例如，在一个编译器的符号表中，关键字都是字符串，表示程序中的标识符。一些很相近的符号经常会出现在同一个程序中，如 pt 和 pts。好的散列函数应能将这些相近符号散列到相同槽中的可能性最小化。

一种好的方法导出的散列值，在某种程度上应独立于数据可能存在的任何模式。例如，“除法散列”(11.3.1 节中要介绍)用一个特定的素数来除所给的关键字，所得的余数即为该关键字的散列值。假定所选择的素数与关键字分布中的任何模式都是无关的，这种方法常常可以给出好的结果。

最后，注意到散列函数的某些应用可能会要求比简单均匀散列更强的性质。例如，可能希望某些很近似的关键字具有截然不同的散列值(使用 11.4 节中定义的线性探查技术时，这一性质特别有用)。11.3.3 节中将介绍的全域散列(universal hashing)通常能够提供这些性质。 262

将关键字转换为自然数

多数散列函数都假定关键字的全域为自然数集 $\mathbf{N}=\{0,1,2,\cdots\}$。因此，如果所给关键字不是自然数，就需要找到一种方法来将它们转换为自然数。例如，一个字符串可以被转换为按适当的基数符号表示的整数。这样，就可以将标识符 pt 转换为十进制整数对(112，116)，这是因为在 ASCII 字符集中，p=112，t=116。然后，以 128 为基数来表示，pt 即为(112×128)+116=14 452。在一特定的应用场合，通常还能设计出其他类似的方法，将每个关键字转换为一个(可能是很大的)自然数。在后面的内容中，假定所给的关键字都是自然数。

11.3.1　除法散列法

在用来设计散列函数的除法散列法中，通过取 k 除以 m 的余数，将关键字 k 映射到 m 个槽

中的某一个上，即散列函数为：

$$h(k) = k \bmod m$$

例如，如果散列表的大小为 $m=12$，所给关键字 $k=100$，则 $h(k)=4$。由于只需做一次除法操作，所以除法散列法是非常快的。

当应用除法散列法时，要避免选择 m 的某些值。例如，m 不应为 2 的幂，因为如果 $m=2^p$，则 $h(k)$就是 k 的 p 个最低位数字。除非已知各种最低 p 位的排列形式为等可能的，否则在设计散列函数时，最好考虑关键字的所有位。练习 11.3-3 要求读者证明，当 k 是一个按基数 2^p 表示的字符串时，选 $m=2^p-1$ 可能是一个糟糕的选择，因为排列 k 的各字符并不会改变其散列值。

一个不太接近 2 的整数幂的素数，常常是 m 的一个较好的选择。例如，假定我们要分配一张散列表并用链接法解决冲突，表中大约要存放 $n=2\,000$ 个字符串，其中每个字符有 8 位。如果我们不介意一次不成功的查找需要平均检查 3 个元素，这样分配散列表的大小为 $m=701$。选择 701 这个数的原因是，它是一个接近 2 000/3 但又不接近 2 的任何次幂的素数。把每个关键字 k 视为一个整数，则散列函数如下：

$$h(k) = k \bmod 701$$

11.3.2 乘法散列法

263

构造散列函数的乘法散列法包含两个步骤。第一步，用关键字 k 乘上常数 $A(0<A<1)$，并提取 kA 的小数部分。第二步，用 m 乘以这个值，再向下取整。总之，散列函数为：

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

这里“$kA \bmod 1$”是取 kA 的小数部分，即 $kA-\lfloor kA \rfloor$。

乘法散列法的一个优点是对 m 的选择不是特别关键，一般选择它为 2 的某个幂次($m=2^p$，p 为某个整数)，这是因为我们可以在大多数计算机上，按下面所示方法较容易地实现散列函数。假设某计算机的字长为 w 位，而 k 正好可用一个单字表示。限制 A 为形如 $s/2^w$ 的一个分数，其中 s 是一个取自 $0<s<2^w$ 的整数。参见图 11-4，先用 w 位整数 $s=A\cdot 2^w$ 乘上 k，其结果是一个 $2w$ 位的值 $r_1 2^w + r_0$，这里 r_1 为乘积的高位字，r_0 为乘积的低位字。所求的 p 位散列值中，包含了 r_0 的 p 个最高有效位。

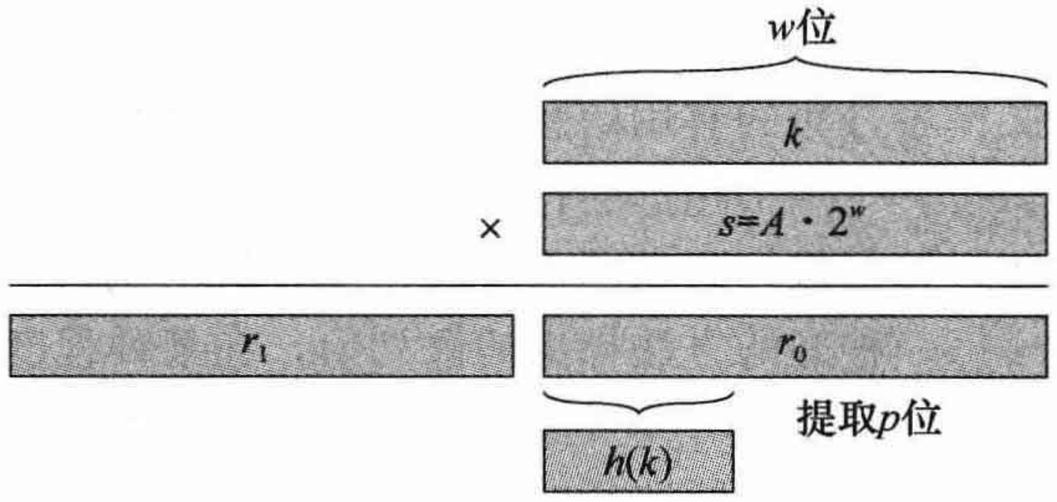

图 11-4 散列的乘法方法。关键字 k 的 w 位表示乘上 $s=A\cdot 2^w$ 的 w 位值。在乘积的低 w 位中，p 个最高位构成了所需的散列值 $h(k)$

虽然这个方法对任何的 A 值都适用，但对某些值效果更好。最佳的选择与待散列的数据的特征有关。Knuth[211]认为

$$A \approx (\sqrt{5}-1)/2 = 0.618\,033\,988\,7\cdots \tag{11.2}$$

是个比较理想的值。

作为一个例子，假设 $k=123\,456$，$p=14$，$m=2^{14}=16\,384$，且 $w=32$。依据 Knuth 的建议，取 A 为形如 $s/2^{32}$ 的分数，它与 $(\sqrt{5}-1)/2$ 最为接近，于是 $A=2\,654\,435\,769/2^{32}$。那么，$k\times s=327\,706\,022\,297\,664=(76\,300\times 2^{32})+17\,612\,864$，从而有 $r_1=76\,300$ 和 $r_0=17\,612\,864$。r_0 的 14

264 个最高有效位产生了散列值 $h(k)=67$。

* 11.3.3 全域散列法

如果让一个恶意的对手来针对某个特定的散列函数选择要散列的关键字，那么他会将 n 个关

$$= (p-1)/m$$

当模 m 进行归约时，s 与 r 发生冲突的概率至多为 $((p-1)/m)/(p-1)=1/m$。

所以，对于任何不同的数对 k，$l \in \mathbf{Z}_p$，有

$$\Pr\{h_{ab}(k) = h_{ab}(l)\} \leqslant 1/m$$

于是，$\mathcal{H}_{pm}$ 的确是全域的。 ■

练习

11.3-1 假设我们希望查找一个长度为 n 的链表，其中每一个元素都包含一个关键字 k 并具有散列值 $h(k)$。每一个关键字都是长字符串。那么在表中查找具有给定关键字的元素时，如何利用各元素的散列值呢？

11.3-2 假设将一个长度为 r 的字符串散列到 m 个槽中，并将其视为一个以 128 为基数的数，要求应用除法散列法。我们可以很容易地把数 m 表示为一个 32 位的机器字，但对长度为 r 的字符串，由于它被当做以 128 为基数的数来处理，就要占用若干个机器字。假设应用除法散列法来计算一个字符串的散列值，那么如何才能在除了该串本身占用的空间外，只利用常数个机器字？ 268

11.3-3 考虑除法散列法的另一种版本，其中 $h(k)=k \bmod m$，$m=2^p-1$，k 为按基数 2^p 表示的字符串。试证明：如果串 x 可由串 y 通过其自身的字符置换排列导出，则 x 和 y 具有相同的散列值。给出一个应用的例子，其中这一特性在散列函数中是不希望出现的。

11.3-4 考虑一个大小为 $m=1\,000$ 的散列表和一个对应的散列函数 $h(k)=\lfloor m(kA \bmod 1)\rfloor$，其中 $A=(\sqrt{5}-1)/2$，试计算关键字 61、62、63、64 和 65 被映射到的位置。

***11.3-5** 定义一个从有限集合 U 到有限集合 B 上的散列函数簇 $\mathcal{H}$ 为 ε 全域的，如果对 U 中所有的不同元素对 k 和 l，都有

$$\Pr\{h(k) = h(l)\} \leqslant \varepsilon$$

其中概率是相对从函数簇 $\mathcal{H}$ 中随机抽取的散列函数 h 而言的。试证明：一个 ε 全域的散列函数簇必定满足：

$$\varepsilon \geqslant \frac{1}{|B|} - \frac{1}{|U|}$$

***11.3-6** 设 U 为由取自 $\mathbf{Z}_p$ 中的值构成的 n 元组集合，并设 $B=\mathbf{Z}_p$，其中 p 为素数。对于一个取自 U 的输入 n 元组 $\langle a_0, a_1, \cdots, a_{n-1}\rangle$，定义其上的散列函数 $h_b: U \to B(b \in \mathbf{Z}_p)$ 为：

$$h_b(\langle a_0, a_1, \cdots, a_{n-1}\rangle) = \left(\sum_{j=0}^{n-1} a_j b^j\right) \bmod p$$

并且设 $\mathcal{H}=\{h_b: b \in \mathbf{Z}_p\}$。根据练习 11.3-5 中 ε 全域的定义，证明 $\mathcal{H}$ 是 $((n-1)/p)$ 全域的。（提示：见练习 31.4-4。）

11.4 开放寻址法

在**开放寻址法**(open addressing)中，所有的元素都存放在散列表里。也就是说，每个表项或包含动态集合的一个元素，或包含 NIL。当查找某个元素时，要系统地检查所有的表项，直到找到所需的元素，或者最终查明该元素不在表中。不像链接法，这里既没有链表，也没有元素存放在散列表外。因此在开放寻址法中，散列表可能会被填满，以至于不能插入任何新的元素。该方法导致的一个结果便是装载因子 α 绝对不会超过 1。 269

当然，也可以将用作链接的链表存放在散列表未用的槽中(见练习 11.2-4)，但开放寻址法的好处就在于它不用指针，而是计算出要存取的槽序列。于是，不用存储指针而节省的空间，使得可以用同样的空间来提供更多的槽，潜在地减少了冲突，提高了检索速度。

为了使用开放寻址法插入一个元素，需要连续地检查散列表，或称为**探查**(probe)，直到找到一个空槽来放置待插入的关键字为止。检查的顺序不一定是0，1，…，$m-1$(这种顺序下的查找时间为$\Theta(n)$)，而是要依赖于待插入的关键字。为了确定要探查哪些槽，我们将散列函数加以扩充，使之包含探查号(从0开始)以作为其第二个输入参数。这样，散列函数就变为：

$$h:U\times\{0,1,\cdots,m-1\}\rightarrow\{0,1,\cdots,m-1\}$$

对每一个关键字k，使用开放寻址法的**探查序列**(probe sequence)

$$\langle h(k,0),h(k,1),\cdots,h(k,m-1)\rangle$$

是$\langle 0,1,\cdots,m-1\rangle$的一个排列，使得当散列表逐渐填满时，每一个表位最终都可以被考虑为用来插入新关键字的槽。在下面的伪代码中，假设散列表T中的元素为无卫星数据的关键字；关键字k等同于包含关键字k的元素。每个槽或包含一个关键字，或包含NIL(如果该槽为空)。HASH-INSERT过程以一个散列表T和一个关键字k为输入，其要么返回关键字k的存储槽位，要么因为散列表已满而返回出错标志。

```
HASH-INSERT(T,k)
1  i=0
2  repeat
3      j=h(k,i)
4      if T[j]==NIL
5          T[j]=k
6          return j
7      else i=i+1
8  until i==m
9  error "hash table overflow"
```

270 查找关键字k的算法的探查序列与将k插入时的算法一样。因此，查找过程中碰到一个空槽时，查找算法就(非成功地)停止，因为如果k在表中，它就应该在此处，而不会在探查序列随后的位置上(之所以这样说，是假定了关键字不会从散列表中删除)。过程HASH-SEARCH的输入为一个散列表T和一个关键字k，如果槽j中包含了关键字k，则返回j；如果k不在表T中，则返回NIL。

```
HASH-SEARCH(T,k)
1  i=0
2  repeat
3      j=h(k,j)
4      if T[j]==k
5          return j
6      i=i+1
7  until T[j]==NIL or i==m
8  return NIL
```

从开放寻址法的散列表中删除操作元素比较困难。当我们从槽i中删除关键字时，不能仅将NIL置于其中来标识它为空。如果这样做，就会有问题：在插入关键字k时，发现槽i被占用了，则k就被插入到后面的位置上；此时将槽i中的关键字删除后，就无法检索到关键字k了。有一个解决办法，就是在槽i中置一个特定的值DELETED替代NIL来标记该槽。这样就要对过程HASH-INSERT做相应的修改，将这样的一个槽当做空槽，使得在此仍然可以插入新的关键字。对HASH-SEARCH无需做什么改动，因为它在搜索时会绕过DELETED标识。但是，当我们使用特殊的值DELETED时，查找时间就不再依赖于装载因子α了。为此，在必须删除关键字的应用中，更常见的做法是采用链接法来解决冲突。

在我们的分析中，做一个**均匀散列**(uniform hashing)的假设：每个关键字的探查序列等可能地为$\langle 0, 1, \cdots, m-1\rangle$的$m!$种排列中的任一种。均匀散列将前面定义过的简单均匀散列的概念加以了一般化，推广到散列函数的结果不只是一个数，而是一个完整的探查序列。然而，真正的均匀散列是难以实现的，在实际应用中，常常采用它的一些近似方法(如下面定义的双重散列等)。

有三种技术常用来计算开放寻址法中的探查序列：线性探查、二次探查和双重探查。这几种技术都能保证对每个关键字k，$\langle h(k, 0), h(k, 1), \cdots, h(k, m-1)\rangle$都是$\langle 0, 1, \cdots, m-1\rangle$的一个排列。但是，这些技术都不能满足均匀散列的假设，因为它们能产生的不同探查序列数都不超过m^2个(均匀散列要求有$m!$个探查序列)。在三种技术中，双重散列产生的探查序列数最多，似乎能给出最好的结果。 271

线性探查

给定一个普通的散列函数h'：$U\rightarrow\{0, 1, \cdots, m-1\}$，称之为**辅助散列函数**(auxiliary hash function)，**线性探查**(linear probing)方法采用的散列函数为：

$$h(k,i) = (h'(k) + i) \bmod m, \quad i = 0,1,\cdots,m-1$$

给定一个关键字k，首先探查槽$T[h'(k)]$，即由辅助散列函数所给出的槽位。再探查槽$T[h'(k)+1]$，依此类推，直至槽$T[m-1]$。然后，又绕到槽$T[0]$，$T[1]$，…，直到最后探查到槽$T[h'(k)-1]$。在线性探查方法中，初始探查位置决定了整个序列，故只有m种不同的探查序列。

线性探查方法比较容易实现，但它存在着一个问题，称为**一次群集**(primary clustering)。随着连续被占用的槽不断增加，平均查找时间也随之不断增加。群集现象很容易出现，这是因为当一个空槽前有i个满的槽时，该空槽为下一个将被占用的概率是$(i+1)/m$。连续被占用的槽就会变得越来越长，因而平均查找时间也会越来越大。

二次探查

二次探查(quadratic probing)采用如下形式的散列函数：

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \tag{11.5}$$

其中h'是一个辅助散列函数，c_1和c_2为正的辅助常数，$i=0$，1，…，$m-1$。初始的探查位置为$T[h'(k)]$，后续的探查位置要加上一个偏移量，该偏移量以二次的方式依赖于探查序号i。这种探查方法的效果要比线性探查好得多，但是，为了能够充分利用散列表，c_1、c_2和m的值要受到限制。思考题11-3给出了一种选择这几个参数的方法。此外，如果两个关键字的初始探查位置相同，那么它们的探查序列也是相同的，这是因为$h(k_1, 0)=h(k_2, 0)$蕴涵着$h(k_1, i)=h(k_2, i)$。这一性质可导致一种轻度的群集，称为**二次群集**(secondary clustering)。像在线性探查中一样，初始探查位置决定了整个序列，这样也仅有m个不同的探查序列被用到。

双重散列

双重散列(double hashing)是用于开放寻址法的最好方法之一，因为它所产生的排列具有随机选择排列的许多特性。双重散列采用如下形式的散列函数：

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$$

其中h_1和h_2均为辅助散列函数。初始探查位置为$T[h_1(k)]$，后续的探查位置是前一个位置加上偏移量$h_2(k)$模m。因此，不像线性探查或二次探查，这里的探查序列以两种不同方式依赖于关键字k，因为初始探查位置、偏移量或者二者都可能发生变化。图11-5给出了一个使用双重散列法进行插入的例子。 272

为了能查找整个散列表，值$h_2(k)$必须要与表的大小m互素(见练习11.4-4)。有一种简便的方法确保这个条件成立，就是取m为2的幂，并设计一个总产生奇数的h_2。另一种方法是取m为素数，并设计一个总是返回较m小的正整数的函数h_2。例如，我们可以取m为素数，并取

$$h_1(k) = k \bmod m, h_2(k) = 1 + (k \bmod m')$$

其中 m' 略小于 m（比如，$m-1$）。例如，如果 $k=123\,456$，$m=701$，$m'=700$，则有 $h_1(k)=80$，$h_2(k)=257$，可知我们的第一个探查位置为 80，然后检查每第 257 个槽（模 m），直到找到该关键字，或者遍历了所有的槽。

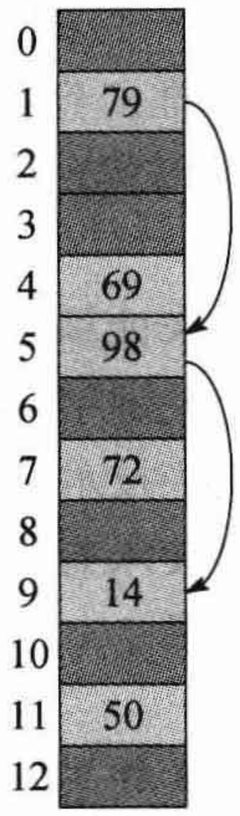

图 11-5 双重散列法的插入。此处，散列表的大小为 13，$h_1(k)=k \bmod 13$，$h_2(k)=1+(k \bmod 11)$。因为 $14 \equiv 1 \pmod{13}$，且 $14 \equiv 3 \pmod{11}$，故在探查了槽 1 和槽 5，并发现它们被占用后，关键字 14 被插入到槽 9 中

当 m 为素数或者 2 的幂时，双重散列法中用到了 $\Theta(m^2)$ 种探查序列，而线性探查或二次探查中用了 $\Theta(m)$ 种，故前者是后两种方法的一种改进。因为每一对可能的 $(h_1(k), h_2(k))$ 都会产生一个不同的探查序列。因此，对于 m 的每一种可能取值，双重散列的性能看起来就非常接近“理想的”均匀散列的性能。

273

尽管除素数和 2 的幂以外的 m 值在理论上也能用于双重散列中，但是在实际中，要高效地产生 $h_2(k)$ 确保使其与 m 互素，将变得更加困难。部分原因是这些数的相对密度 $\phi(m)/m$ 可能较小（见公式(31.24)）。

开放寻址散列的分析

像在链接法中的分析一样，开放寻址法的分析也是以散列表的装载因子 $\alpha=n/m$ 来表达的。当然，使用开放寻址法，每个槽中至多只有一个元素，因而 $n \leqslant m$，也就意味着 $\alpha \leqslant 1$。

假设采用的是均匀散列。在这种理想的方法中，用于插入或查找每一个关键字 k 的探查序列 $\langle h(k,0), h(k,1), \cdots, h(k,m-1)\rangle$ 等可能地为 $\langle 0, 1, \cdots, m-1\rangle$ 的任意一种排列。当然，每一个给定的关键字有其相应的唯一固定的探查序列。我们这里想说的是，考虑到关键字空间上的概率分布及散列函数施于这些关键字上的操作，每一种探查序列都是等可能的。

现在就来分析在均匀散列的假设下，用开放寻址法来进行散列时探查的期望次数。先来分析一次不成功查找时的探查次数。

定理 11.6 给定一个装载因子为 $\alpha=n/m<1$ 的开放寻址散列表，并假设是均匀散列的，则对于一次不成功的查找，其期望的探查次数至多为 $1/(1-\alpha)$。

证明 在一次不成功的查找中，除了最后一次探查，每一次探查都要检查一个被占用但并不包含所求关键字的槽，最后检查的槽是空的。先定义随机变量 X 为一次不成功查找的探查次数，再定义事件 $A_i(i=1, 2, \cdots)$ 为第 i 次探查且探查到的是一个已经被占用的槽。那么，事件 $\{X \geqslant i\}$ 即为事件 $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$ 的交集。下面通过给出 $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$ 的界来得到 $\Pr\{X \geqslant i\}$ 的界。根据练习 C.2-5，有

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$$

由于有 n 个元素和 m 个槽，所以 $\Pr\{A_1\}=n/m$。对于 $j>1$，在前 $j-1$ 次探查到的都是已占用槽

274

的前提下，第 j 次探查且探查到的仍是已占用槽的概率是 $(n-j+1)/(m-j+1)$。这是因为要在 $(m-(j-1))$ 个未探查的槽中，查找余下的 $(n-(j-1))$ 个元素中的某一个。由均匀散列的假设知，这一概率为这两个量的比值。注意到 $n<m$，对于所有 $j(0 \leqslant j<m)$，就有 $(n-j)/(m-j) \leqslant n/m$。于是，对所有 $i(1 \leqslant i \leqslant m)$，有

$$\Pr\{X \geqslant i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leqslant \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

现在，再利用公式(C.25)来得出探查期望数的界：

$$\mathrm{E}[X]=\sum_{i=1}^{\infty}\Pr\{X\geqslant i\}\leqslant\sum_{i=1}^{\infty}\alpha^{i-1}=\sum_{i=0}^{\infty}\alpha^{i}=\frac{1}{1-\alpha}$$ ■

$1/(1-\alpha)=1+\alpha+\alpha^2+\alpha^3+\cdots$的这个界有一个直观的解释。无论如何，总要进行第一次探查。第一次探查发现的是一个已占用的槽时，必须要进行第二次探查，进行第二次探查的概率大约为α。前两次探查所发现的槽均是已占用时，需要进行第三次探查，进行第三次探查的概率大约为α^2，等等。

如果α是一个常数，由定理11.6可知，一次不成功查找的运行时间为$O(1)$。例如，如果散列表一半是满的，一次不成功查找的平均探查数至多是$1/(1-0.5)=2$。如果散列表是90%满的，则平均探查数至多为$1/(1-0.9)=10$。

根据定理11.6，几乎直接可以得到HASH-INSERT过程的性能。

推论11.7 假设采用的是均匀散列，平均情况下，向一个装载因子为α的开放寻址散列表中插入一个元素至多需要做$1/(1-\alpha)$次探查。 275

证明 只有当表中有空槽时，才可以插入新元素，故$\alpha<1$。插入一个关键字要先做一次不成功的查找，然后将该关键字置入第一个遇到的空槽中。所以，期望的探查次数至多为$1/(1-\alpha)$。■

对于一次成功的查找，需要稍做一些工作来得到探查的期望次数。

定理11.8 对于一个装载因子为$\alpha<1$的开放寻址散列表，一次成功查找中的探查期望数至多为

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

假设采用均匀散列，且表中的每个关键字被查找的可能性是相同的。

证明 查找关键字k的探查序列与插入关键字为k的元素的探查序列是相同的。根据推论11.7，如果k是第$(i+1)$个被插入表中的关键字，则对k的一次查找中，探查的期望次数至多为$1/(1-i/m)=m/(m-i)$。对散列表中所有n个关键字求平均，则得到一次成功查找的探查期望次数为：

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\leqslant\frac{1}{\alpha}\int_{m-n}^{m}(1/x)\mathrm{d}x\qquad\text{(由不等式(A.12))}$$

$$=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$ ■

如果散列表是半满的，则一次成功的查找中，探查的期望数小于1.387。如果散列表为90%满的，则探查的期望数小于2.559。 276

练习

11.4-1 考虑用开放寻址法将关键字10、22、31、4、15、28、17、88、59插入到一长度为$m=11$的散列表中，辅助散列函数为$h'(k)=k$。试说明分别用线性探查、二次探查($c_1=1$，$c_2=3$)和双重散列($h_1(k)=k$，$h_2(k)=1+(k \bmod (m-1))$)将这些关键字插入散列表的过程。

11.4-2 试写出HASH-DELETE的伪代码；修改HASH-INSERT，使之能处理特殊值DELETED。

11.4-3 考虑一个采用均匀散列的开放寻址散列表。当装载因子为3/4和7/8时，试分别给出一次不成功查找和一次成功查找的探查期望数上界。

***11.4-4** 假设采用双重散列来解决冲突，即所用的散列函数为$h(k,i)=(h_1(k)+ih_2(k)) \bmod m$。试证明：如果对某个关键字$k$，$m$和$h_2(k)$有最大公约数$d\geqslant 1$，则在对关键字$k$的一次