6.4. HASHING

SO FAR WE HAVE CONSIDERED search methods based on comparing the given argument K to the keys in the table, or using its digits to govern a branching process. A third possibility is to avoid all this rummaging around by doing some arithmetical calculation on K, computing a function $f(K)$ that is the location of K and the associated data in the table.

For example, let's consider again the set of 31 English words that we have subjected to various search strategies in Sections 6.2.2 and 6.3. Table 1 shows a short MIX program that transforms each of the 31 keys into a unique number $f(K)$ between -10 and 30. If we compare this method to the MIX programs for the other methods we have considered (for example, binary search, optimal tree search, trie memory, digital tree search), we find that it is superior from the standpoint of both space and speed, except that binary search uses slightly less space. In fact, the average time for a successful search, using the program of Table 1 with the frequency data of Fig. 12, is only about $17.8u$, and only 41 table locations are needed to store the 31 keys.

Unfortunately, such functions $f(K)$ aren't very easy to discover. There are $41^{31} \approx 10^{50}$ possible functions from a 31-element set into a 41-element set, and only $41 \cdot 40 \cdot \ldots \cdot 11 = 41!/10! \approx 10^{43}$ of them will give distinct values for each argument; thus only about one of every 10 million functions will be suitable.

Functions that avoid duplicate values are surprisingly rare, even with a fairly large table. For example, the famous "birthday paradox" asserts that if 23 or more people are present in a room, chances are good that two of them will have the same month and day of birth! In other words, if we select a random function that maps 23 keys into a table of size 365, the probability that no two keys map into the same location is only 0.4927 (less than one-half). Skeptics who doubt this result should try to find the birthday mates at the next large parties they attend. [The birthday paradox was discussed informally by mathematicians in the 1930s, but its origin is obscure; see I. J. Good, *Probability and the Weighing of Evidence* (Griffin, 1950), 38. See also R. von Mises, *İstanbul Üniversitesi Fen Fakültesi Mecmuasi* **4** (1939), 145–163, and W. Feller, *An Introduction to Probability Theory* (New York: Wiley, 1950), Section 2.3.]

On the other hand, the approach used in Table 1 is fairly flexible [see M. Greniewski and W. Turski, *CACM* **6** (1963), 322–323], and for a medium-sized table a suitable function can be found after about a day's work. In fact it is rather amusing to solve a puzzle like this. Suitable techniques have been discussed by many people, including for example R. Sprugnoli, *CACM* **20** (1977), 841–850, **22** (1979), 104, 553; R. J. Cichelli, *CACM* **23** (1980), 17–19; T. J. Sager, *CACM* **28** (1985), 523–532, **29** (1986), 557; B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, *Comp. J.* **39** (1996), 547–554; Czech, Havas, and Majewski, *Theoretical Comp. Sci.* **182** (1997), 1–143. See also the article by J. Körner and K. Marton, *Europ. J. Combinatorics* **9** (1988), 523–530, for theoretical limitations on perfect hash functions.

Of course this method has a serious flaw, since the contents of the table must be known in advance; adding one more key will probably ruin everything,

Table 1

TRANSFORMING A SET OF KEYS INTO UNIQUE ADDRESSES

	Instruction		A	AND	ARE	AS	AT	BE	BUT	BY	FOR	FROM	HAD	HAVE	HE	HER
	LD1N	K(1:1)	−1	−1	−1	−1	−1	−2	−2	−2	−6	−6	−8	−8	−8	−8
	LD2	K(2:2)	−1	−1	−1	−1	−1	−2	−2	−2	−6	−6	−8	−8	−8	−8
	INC1	-8,2	−9	6	10	13	14	−5	14	18	2	5	−15	−15	−11	−11
	J1P	*+2	−9	6	10	13	14	−5	14	18	2	5	−15	−15	−11	−11
	INC1	16,2	7	.	.	.	.	16	.	.	.	.	2	2	10	10
	LD2	K(3,3)	7	6	10	13	14	16	14	18	2	5	2	2	10	10
	J2Z	9F	7	6	10	13	14	16	14	18	2	5	2	2	10	10
	INC1	-28,2	.	−18	−13	.	.	.	9	.	−7	−7	−22	−1	.	−1
	J1P	9F	.	−18	−13	.	.	.	9	.	−7	−7	−22	−1	.	−1
	INC1	11,2	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	LDA	K(4:4)	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	JAZ	9F	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	DEC1	-5,2	.	.	.	.	.	.	.	.	.	9	.	15	.	.
	J1N	9F	.	.	.	.	.	.	.	.	.	9	.	15	.	.
	INC1	10	.	.	.	.	.	.	.	.	.	19	.	25	.	.
9H	LDA	K	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1
	CMPA	TABLE,1	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1
	JNE	FAILURE	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1

making it necessary to start over almost from scratch. We can obtain a much more versatile method if we give up the idea of uniqueness, permitting different keys to yield the same value $f(K)$, and using a special method to resolve any ambiguity after $f(K)$ has been computed.

These considerations lead to a popular class of search methods commonly known as *hashing* or *scatter storage* techniques. The verb "to hash" means to chop something up or to make a mess out of it; the idea in hashing is to scramble some aspects of the key and to use this partial information as the basis for searching. We compute a *hash address* $h(K)$ and begin searching there.

The birthday paradox tells us that there will probably be distinct keys $K_i \neq K_j$ that hash to the same value $h(K_i) = h(K_j)$. Such an occurrence is called a *collision*, and several interesting approaches have been devised to handle the collision problem. In order to use a hash table, programmers must make two almost independent decisions: They must choose a hash function $h(K)$, and they must select a method for collision resolution. We shall now consider these two aspects of the problem in turn.

Hash functions. To make things more explicit, let us assume throughout this section that our hash function h takes on at most M different values, with

$$0 \leq h(K) < M, \tag{1}$$

for all keys K. The keys in actual files that arise in practice usually have a great deal of redundancy; we must be careful to find a hash function that breaks up clusters of almost identical keys, in order to reduce the number of collisions.

HIS	I	IN	IS	IT	NOT	OF	ON	OR	THAT	THE	THIS	TO	WAS	WHICH	WITH	YOU
Contents of rI1 after executing the instruction, given a particular key K																
−8	−9	−9	−9	−9	−15	−16	−16	−16	−23	−23	−23	−23	−26	−26	−26	−28
−8	−9	−9	−9	−9	−15	−16	−16	−16	−23	−23	−23	−23	−26	−26	−26	−28
−7	−17	−2	5	6	−7	−18	−9	−5	−23	−23	−23	−15	−33	−26	−25	−20
−7	−17	−2	5	6	−7	−18	−9	−5	−23	−23	−23	−15	−33	−26	−25	−20
18	−1	29	.	.	25	4	22	30	1	1	1	17	−16	−2	0	12
18	−1	29	5	6	25	4	22	30	1	1	1	17	−16	−2	0	12
18	−1	29	5	6	25	4	22	30	1	1	1	17	−16	−2	0	12
12	.	.	.	.	20	.	.	.	−26	−22	−18	.	−22	−21	−5	8
12	.	.	.	.	20	.	.	.	−26	−22	−18	.	−22	−21	−5	8
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−10	.	−2	.	.	−5	11	.
.	.	.	.	.	.	.	.	.	−10	.	−2	.	.	−5	11	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	21	.
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8

It is theoretically impossible to define a hash function that creates truly random data from the nonrandom data in actual files. But in practice it is not difficult to produce a pretty good imitation of random data, by using simple arithmetic as we have discussed in Chapter 3. And in fact we can often do even better, by exploiting the nonrandom properties of actual data to construct a hash function that leads to fewer collisions than truly random keys would produce.

Consider, for example, the case of 10-digit keys on a decimal computer. One hash function that suggests itself is to let $M = 1000$, say, and to let $h(K)$ be three digits chosen from somewhere near the middle of the 20-digit product $K \times K$. This would seem to yield a fairly good good spread of values between 000 and 999, with low probability of collisions. Experiments with actual data show, in fact, that this "middle square" method isn't bad, provided that the keys do not have a lot of leading or trailing zeros; but it turns out that there are safer and saner ways to proceed, just as we found in Chapter 3 that the middle square method is not an especially good random number generator.

Extensive tests on typical files have shown that two major types of hash functions work quite well. One is based on division, and the other is based on multiplication.

The division method is particularly easy; we simply use the remainder modulo M:

$$h(K) = K \bmod M. \tag{2}$$

In this case, some values of M are obviously much better than others. For example, if M is an even number, $h(K)$ will be even when K is even and odd

when K is odd, and this will lead to a substantial bias in many files. It would be even worse to let M be a power of the radix of the computer, since $K \bmod M$ would then be simply the least significant digits of K (independent of the other digits). Similarly we can argue that M probably shouldn't be a multiple of 3; for if the keys are alphabetic, two keys that differ only by permutation of letters would then differ in numeric value by a multiple of 3. (This occurs because $2^{2n} \bmod 3 = 1$ and $10^n \bmod 3 = 1$.) In general, we want to avoid values of M that divide $r^k \pm a$, where k and a are small numbers and r is the radix of the alphabetic character set (usually $r = 64$, 256, or 100), since a remainder modulo such a value of M tends to be largely a simple superposition of the key digits. Such considerations suggest that we *choose M to be a prime number* such that $r^k \not\equiv \pm a$ (modulo M) for small k and a. This choice has been found to be quite satisfactory in most cases.

For example, on the MIX computer we could choose $M = 1009$, computing $h(K)$ by the sequence

```
LDX   K        rX ← K.
ENTA  0        rA ← 0.                    (3)
DIV   =1009=   rX ← K mod 1009.
```

The multiplicative hashing scheme is equally easy to do, but it is slightly harder to describe because we must imagine ourselves working with fractions instead of with integers. Let w be the word size of the computer, so that w is usually 10^{10} or 2^{30} for MIX; we can regard an integer A as the fraction A/w if we imagine the radix point to be at the left of the word. The method is to choose some integer constant A relatively prime to w, and to let

$$h(K) = \left\lfloor M \left(\left(\frac{A}{w} K \right) \bmod 1 \right) \right\rfloor . \tag{4}$$

In this case we usually let M be a power of 2 on a binary computer, so that $h(K)$ consists of the leading bits of the least significant half of the product AK.

In MIX code, if we let $M = 2^m$ and assume a binary radix, the multiplicative hash function is

```
LDA   K    rA ← K.
MUL   A    rAX ← AK.
ENTA  0    rAX ← AK mod w.              (5)
SLB   m    Shift rAX m bits to the left.
```

Now $h(K)$ appears in register A. Since MIX has rather slow multiplication and shift instructions, this sequence takes exactly as long to compute as (3); but on many machines multiplication is significantly faster than division.

In a sense this method can be regarded as a generalization of (3), since we could for example take A to be an approximation to $w/1009$; multiplying by the reciprocal of a constant is often faster than dividing by that constant. The technique of (5) is almost a "middle square" method, but there is one important difference: We shall see that multiplication by a suitable constant has demonstrably good properties.