## 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 Mecmuası* **4** (1939), 145–163, and W. Feller, *An Introduction to Probability Theory* (New York: Wiley, 1950), Section II.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

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

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 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.

One of the nice features of the multiplicative scheme is that no information is lost when we blank out the A register in (5); we could determine $K$ again, given only the contents of rAX after (5) has finished. The reason is that $A$ is relatively prime to $w$, so Euclid's algorithm can be used to find a constant $A'$ with $AA' \bmod w = 1$; this implies that $K = \big(A'(AK \bmod w)\big) \bmod w$. In other words, if $f(K)$ denotes the contents of register $X$ just before the `SLB` instruction in (5), then

$$K_1 \neq K_2 \qquad \text{implies} \qquad f(K_1) \neq f(K_2). \tag{6}$$

Of course $f(K)$ takes on values in the range 0 to $w-1$, so it isn't any good as a hash function, but it can be very useful as a *scrambling function*, namely a function satisfying (6) that tends to randomize the keys. Such a function can be very useful in connection with the tree search algorithms of Section 6.2.2, if the order of keys is unimportant, since it removes the danger of degeneracy when keys enter the tree in increasing order. (See exercise 6.2.2–10.) A scrambling function is also useful in connection with the digital tree search algorithm of Section 6.3, if the bits of the actual keys are biased.

Another feature of the multiplicative hash method is that it makes good use of the nonrandomness found in many files. Actual sets of keys often have a preponderance of arithmetic progressions, where $\{K, K+d, K+2d, \ldots, K+td\}$ all appear in the file; for example, consider alphabetic names like {`PART1`, `PART2`, `PART3`} or {`TYPEA`, `TYPEB`, `TYPEC`}. The multiplicative hash method converts an arithmetic progression into an approximate arithmetic progression $h(K)$, $h(K+d)$, $h(K+2d)$, ... of distinct hash values, reducing the number of collisions from what we would expect in a random situation. The division method has this same property.

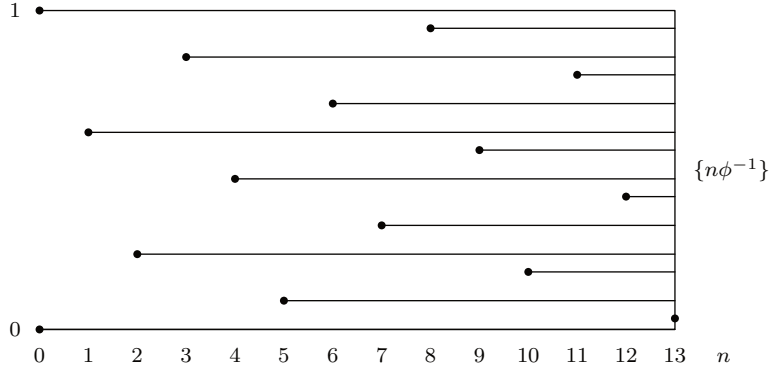



**Fig. 37.** Fibonacci hashing.

Figure 37 illustrates this aspect of multiplicative hashing in a particularly interesting case. Suppose that $A/w$ is approximately the golden ratio $\phi^{-1} = \big(\sqrt{5}-1\big)/2 \approx 0.6180339887$; then the successive values $h(K)$, $h(K+1)$, $h(K+2)$, ... have essentially the same behavior as the successive hash values $h(0)$, $h(1)$, $h(2)$, ..., so the following experiment suggests itself: Starting with the line

segment $[0\,..\,1]$, we successively mark off the points $\{\phi^{-1}\}$, $\{2\phi^{-1}\}$, $\{3\phi^{-1}\}$, ..., where $\{x\}$ denotes the fractional part of $x$ (namely $x-\lfloor x\rfloor$, or $x \bmod 1$). As shown in Fig. 37, these points stay very well separated from each other; in fact, each newly added point falls into one of the largest remaining intervals, and divides it in the golden ratio! [This phenomenon was observed long ago by botanists Louis and Auguste Bravais, *Annales des Sciences Naturelles* **7** (1837), 42–110, who gave an illustration equivalent to Fig. 37 and related it to the Fibonacci sequence. See also S. Świerczkowski, *Fundamenta Math.* **46** (1958), 187–189.]

The remarkable scattering property of the golden ratio is actually just a special case of a very general result, originally conjectured by Hugo Steinhaus and first proved by Vera Turán Sós [*Acta Math. Acad. Sci. Hung.* **8** (1957), 461–471; *Ann. Univ. Sci. Budapest. Eötvös Sect. Math.* **1** (1958), 127–134]:

**Theorem S.** *Let $\theta$ be any irrational number. When the points $\{\theta\}$, $\{2\theta\}$, ..., $\{n\theta\}$ are placed in the line segment $[0\,..\,1]$, the $n+1$ line segments formed have at most three different lengths. Moreover, the next point $\{(n+1)\theta\}$ will fall in one of the largest existing segments.* ▮

Thus, the points $\{\theta\}, \{2\theta\}, \ldots, \{n\theta\}$ are spread out very evenly between 0 and 1. If $\theta$ is rational, the same theorem holds if we give a suitable interpretation to the segments of length 0 that appear when $n$ is greater than or equal to the denominator of $\theta$. A proof of Theorem S, together with a detailed analysis of the underlying structure of the situation, appears in exercise 8; it turns out that the segments of a given length are created and destroyed in a first-in-first-out manner. Of course, some $\theta$'s are better than others, since for example a value that is near 0 or 1 will start out with many small segments and one large segment. Exercise 9 shows that the two numbers $\phi^{-1}$ and $\phi^{-2} = 1-\phi^{-1}$ lead to the "most uniformly distributed" sequences, among all numbers $\theta$ between 0 and 1.

The theory above suggests *Fibonacci hashing*, where we choose the constant $A$ to be the nearest integer to $\phi^{-1}w$ that is relatively prime to $w$. For example if `MIX` were a decimal computer we would take

$$A = \boxed{+}\boxed{61}\boxed{80}\boxed{33}\boxed{98}\boxed{87}\ . \tag{7}$$

This multiplier will spread out alphabetic keys like `LIST1`, `LIST2`, `LIST3` very nicely. But notice what happens when we have an arithmetic series in the fourth character position, as in the keys `SUM1`␣, `SUM2`␣, `SUM3`␣: The effect is as if Theorem S were being used with $\theta = \{100A/w\} = .80339887$ instead of $\theta = .6180339887 = A/w$. The resulting behavior is still all right, in spite of the fact that this value of $\theta$ is not quite as good as $\phi^{-1}$. On the other hand, if the progression occurs in the second character position, as in `A1`␣␣␣, `A2`␣␣␣, `A3`␣␣␣, the effective $\theta$ is .9887, and this is probably too close to 1.

Therefore we might do better with a multiplier like

$$A = \boxed{+}\boxed{61}\boxed{61}\boxed{61}\boxed{61}\boxed{61}$$

in place of (7); such a multiplier will separate out consecutive sequences of keys that differ in *any* character position. Unfortunately this choice suffers from

another problem analogous to the difficulty of dividing by $r^k \pm 1$: Keys such as `XY` and `YX` will tend to hash to the same location! One way out of this difficulty is to look more closely at the structure underlying Theorem S. For short progressions of keys, only the first few partial quotients of the continued fraction representation of $\theta$ are relevant, and small partial quotients correspond to good distribution properties. Therefore we find that the best values of $\theta$ lie in the ranges

$$\tfrac{1}{4} < \theta < \tfrac{3}{10}, \qquad \tfrac{1}{3} < \theta < \tfrac{3}{7}, \qquad \tfrac{4}{7} < \theta < \tfrac{2}{3}, \qquad \tfrac{7}{10} < \theta < \tfrac{3}{4}.$$

A value of $A$ can be found so that each of its bytes lies in a good range and is not too close to the values of the other bytes or their complements, for example

$$A = \boxed{+}\boxed{61}\boxed{25}\boxed{42}\boxed{33}\boxed{71}\ . \tag{8}$$

Such a multiplier can be recommended. (These ideas about multiplicative hashing are due largely to R. W. Floyd.)

A good hash function should satisfy two requirements:

a) Its computation should be very fast.
b) It should minimize collisions.

Property (a) is machine-dependent, and property (b) is data-dependent. If the keys were truly random, we could simply extract a few bits from them and use those bits for the hash function; but in practice we nearly always need to have a hash function that depends on all bits of the key in order to satisfy (b).

So far we have considered how to hash one-word keys. Multiword or variable-length keys can be handled by multiple-precision extensions of the methods above, but it is generally adequate to speed things up by combining the individual words together into a single word, then doing a single multiplication or division as above. The combination can be done by addition mod $w$, or by exclusive-or on a binary computer; both of these operations have the advantage that they are invertible, namely that they depend on all bits of both arguments, and exclusive-or is sometimes preferable because it avoids arithmetic overflow. However, both of these operations are commutative, hence $(X, Y)$ and $(Y, X)$ will hash to the same address; G. D. Knott has suggested avoiding this problem by doing a cyclic shift just before adding or exclusive-oring.

An even better way to hash $l$-character or $l$-word keys $K = x_1 x_2 \ldots x_l$ is to compute

$$h(K) = \big(h_1(x_1) + h_2(x_2) + \cdots + h_l(x_l)\big) \bmod M, \tag{9}$$

where each $h_j$ is an independent hash function. This idea, introduced by J. L. Carter and M. N. Wegman in 1977, is especially efficient when each $x_j$ is a single character, because we can then use a precomputed array for each $h_j$. Such arrays make multiplication unnecessary. If $M$ is a power of 2, we can avoid the division in (9) by substituting exclusive-or for addition; this gives a different, but equally good, hash function. Therefore (9) certainly satisfies property (a). Moreover, Carter and Wegman proved that if the $h_j$ are chosen at random, property (b) will hold *regardless of the input data.* (See exercise 72.)

Many more methods for hashing have been suggested, but none of them have proved to be superior to the simple methods described above. For a survey of several approaches together with detailed statistics on their performance with actual files, see the article by V. Y. Lum, P. S. T. Yuen, and M. Dodd, *CACM* **14** (1971), 228–239.

Of all the other hash methods that have been tried, perhaps the most interesting is a technique based on algebraic coding theory; the idea is analogous to the division method above, but we divide by a polynomial modulo 2 instead of dividing by an integer. (As observed in Section 4.6, this operation is analogous to division, just as addition is analogous to exclusive-or.) For this method, $M$ should be a power of 2, say $M = 2^m$, and we make use of an $m$th degree polynomial $P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_0$. An $n$-digit binary key $K = (k_{n-1}\ldots k_1 k_0)_2$ can be regarded as the polynomial $K(x) = k_{n-1}x^{n-1} + \cdots + k_1 x + k_0$, and we compute the remainder

$$K(x) \bmod P(x) = h_{m-1}x^{m-1} + \cdots + h_1 x + h_0$$

using polynomial arithmetic modulo 2; then $h(K) = (h_{m-1}\ldots h_1 h_0)_2$. If $P(x)$ is chosen properly, this hash function can be guaranteed to avoid collisions between nearly equal keys. For example if $n = 15$, $m = 10$, and

$$P(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1, \tag{10}$$

it can be shown that $h(K_1)$ will be unequal to $h(K_2)$ whenever $K_1$ and $K_2$ are distinct keys that differ in fewer than seven bit positions. (See exercise 7 for further information about this scheme; it is, of course, more suitable for hardware or microprogramming implementation than for software.)

It is often convenient to use the constant hash function $h(K) = 0$ when debugging a program, since all keys will be stored together; an efficient $h(K)$ can be substituted later.

**Collision resolution by "chaining."** We have observed that some hash addresses will probably be burdened with more than their share of keys. Perhaps the most obvious way to solve this problem is to maintain $M$ linked lists, one for each possible hash code. A `LINK` field should be included in each record, and there will also be $M$ list heads, numbered say from 1 through $M$. After hashing the key, we simply do a sequential search in list number $h(K) + 1$. (See exercise 6.1–2. The situation is very similar to multiple-list-insertion sorting, Program 5.2.1M.)

Figure 38 illustrates this simple chaining scheme when $M = 9$, for the sequence of seven keys

$$K = \texttt{EN}, \texttt{TO}, \texttt{TRE}, \texttt{FIRE}, \texttt{FEM}, \texttt{SEKS}, \texttt{SYV} \tag{11}$$

(the numbers 1 through 7 in Norwegian), having the respective hash codes

$$h(K) + 1 = 3,\ 1,\ 4,\ 1,\ 5,\ 9,\ 2. \tag{12}$$

The first list has two elements, and three of the lists are empty.

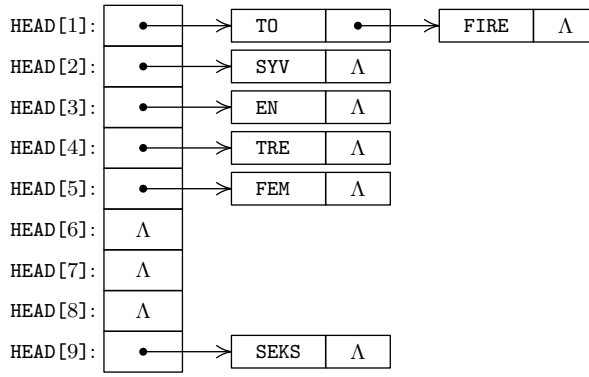



**Fig. 38.** Separate chaining.

Chaining is quite fast, because the lists are short. If 365 people are gathered together in one room, there will probably be many pairs having the same birthday, but the average number of people with any given birthday will be only 1! In general, if there are $N$ keys and $M$ lists, the average list size is $N/M$; thus hashing decreases the average amount of work needed for sequential searching by roughly a factor of $M$. (A precise formula is worked out in exercise 34.)

This method is a straightforward combination of techniques we have discussed before, so we do not need to formulate a detailed algorithm for chained hash tables. It is often a good idea to keep the individual lists in order by key, so that unsuccessful searches — which must precede insertions — go faster. Thus if we choose to make the lists ascending, the `TO` and `FIRE` nodes of Fig. 38 would be interchanged, and all the $\Lambda$ links would be replaced by pointers to a dummy record whose key is $\infty$. (See Algorithm 6.1T.) Alternatively we could make use of the "self-organizing" concept discussed in Section 6.1; instead of keeping the lists in order by key, they may be kept in order according to the time of most recent occurrence.

For the sake of speed we would like to make $M$ rather large. But when $M$ is large, many of the lists will be empty and much of the space for the $M$ list heads will be wasted. This suggests another approach, when the records are small: We can overlap the record storage with the list heads, making room for a total of $M$ records and $M$ links instead of for $N$ records and $M + N$ links. Sometimes it is possible to make one pass over all the data to find out which list heads will be used, then to make another pass inserting all the "overflow" records into the empty slots. But this is often impractical or impossible, and we'd rather have a technique that processes each record only once when it first enters the system. The following algorithm, due to F. A. Williams [*CACM* **2**, 6 (June 1959), 21–24], is a convenient way to solve the problem.

**Algorithm C** (*Chained hash table search and insertion*). This algorithm looks for a given key $K$ in an $M$-node table. If $K$ is not in the table and the table is not full, $K$ is inserted.

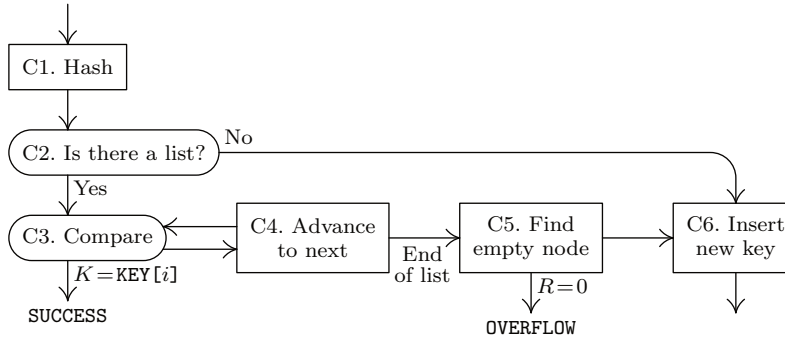



**Fig. 39.** Chained hash table search and insertion.

The nodes of the table are denoted by `TABLE[`$i$`]`, for $0 \le i \le M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key field `KEY[`$i$`]`, a link field `LINK[`$i$`]`, and possibly other fields.

The algorithm makes use of a hash function $h(K)$. An auxiliary variable $R$ is also used, to help find empty spaces; when the table is empty, we have $R = M + 1$, and as insertions are made it will always be true that `TABLE[`$j$`]` is occupied for all $j$ in the range $R \le j \le M$. By convention, `TABLE[0]` will always be empty.

**C1.** [Hash.] Set $i \leftarrow h(K) + 1$. (Now $1 \le i \le M$.)

**C2.** [Is there a list?] If `TABLE[`$i$`]` is empty, go to C6. (Otherwise `TABLE[`$i$`]` is occupied; we will look at the list of occupied nodes that starts here.)

**C3.** [Compare.] If $K =$ `KEY[`$i$`]`, the algorithm terminates successfully.

**C4.** [Advance to next.] If `LINK[`$i$`]` $\neq 0$, set $i \leftarrow$ `LINK[`$i$`]` and go back to step C3.

**C5.** [Find empty node.] (The search was unsuccessful, and we want to find an empty position in the table.) Decrease $R$ one or more times until finding a value such that `TABLE[`$R$`]` is empty. If $R = 0$, the algorithm terminates with overflow (there are no empty nodes left); otherwise set `LINK[`$i$`]` $\leftarrow R$, $i \leftarrow R$.

**C6.** [Insert new key.] Mark `TABLE[`$i$`]` as an occupied node, with `KEY[`$i$`]` $\leftarrow K$ and `LINK[`$i$`]` $\leftarrow 0$. ∎

This algorithm allows several lists to coalesce, so that records need not be moved after they have been inserted into the table. For example, see Fig. 40, where `SEKS` appears in the list containing `TO` and `FIRE` since the latter had already been inserted into position 9.

In order to see how Algorithm C compares with others in this chapter, we can write the following `MIX` program. The analysis worked out below indicates that the lists of occupied cells tend to be short, and the program has been designed with this fact in mind.

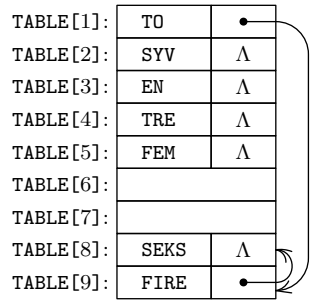
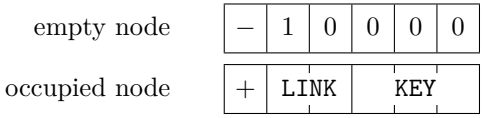



**Fig. 40.** Coalesced chaining.

**Program C** (*Chained hash table search and insertion*). For convenience, the keys are assumed to be only three bytes long, and nodes are represented as follows:




(13)

The table size $M$ is assumed to be prime; `TABLE[`$i$`]` is stored in location `TABLE`$+i$. rI1 $\equiv i$, rA $\equiv K$; rI2 $\equiv$ `LINK[`$i$`]` and/or $R$.

```
01  KEY    EQU  3:5
02  LINK   EQU  0:2
03  START  LDX  K                  1          C1. Hash.
04         ENTA 0                  1
05         DIV  =M=                1
06         STX  *+1(0:2)           1
07         ENT1 *                  1          i ← h(K)
08         INC1 1                  1                  + 1.
09         LDA  K                  1
10         LD2  TABLE,1(LINK)      1          C2. Is there a list?
11         J2N  6F                 1          To C6 if TABLE[i] empty.
12         CMPA TABLE,1(KEY)       A          C3. Compare.
13         JE   SUCCESS            A          Exit if K = KEY[i].
14         J2Z  5F               A − S1       To C5 if LINK[i] = 0.
15  4H     ENT1 0,2              C − 1        C4. Advance to next.
16         CMPA TABLE,1(KEY)     C − 1        C3. Compare.
17         JE   SUCCESS          C − 1        Exit if K = KEY[i].
18         LD2  TABLE,1(LINK)  C − 1 − S2
19         J2NZ 4B             C − 1 − S2     Advance if LINK[i] ≠ 0.
20  5H     LD2  R                A − S        C5. Find empty node.
21         DEC2 1                  T          R ← R − 1.
22         LDX  TABLE,2            T
23         JXNN *-2                T          Repeat until TABLE[R] empty.
24         J2Z  OVERFLOW         A − S        Exit if no empty nodes left.
25         ST2  TABLE,1(LINK)    A − S        LINK[i] ← R.
26         ENT1 0,2              A − S        i ← R.
```

```
27          ST2  R                    A − S    Update R in memory.
28    6H    STZ  TABLE,1(LINK)        1 − S    C6. Insert new key. LINK[i] ← 0.
29          STA  TABLE,1(KEY)         1 − S    KEY[i] ← K.  ∎
```

The running time of this program depends on

$C$ = number of table entries probed while searching;

$A$ = [initial probe found an occupied node];

$S$ = [search was successful];

$T$ = number of table entries probed while looking for an empty space.

Here $S = S1 + S2$, where $S1 = 1$ if successful on the first try. The total running time for the searching phase of Program C is $(7C + 4A + 17 - 3S + 2S1)u$, and the insertion of a new key when $S = 0$ takes an additional $(8A + 4T + 4)u$.

Suppose there are $N$ keys in the table at the start of this program, and let

$$\alpha = N/M = \text{load factor of the table.} \tag{14}$$

Then the average value of $A$ in an unsuccessful search is obviously $\alpha$, if the hash function is random; and exercise 39 proves that the average value of $C$ in an unsuccessful search is

$$C'_N = 1 + \frac{1}{4}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) \approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}. \tag{15}$$

Thus when the table is half full, the average number of probes made in an unsuccessful search is about $\frac{1}{4}(e + 2) \approx 1.18$; and even when the table gets completely full, the average number of probes made just before inserting the final item will be only about $\frac{1}{4}(e^2 + 1) \approx 2.10$. The standard deviation is also small, as shown in exercise 40. These statistics prove that *the lists stay short even though the algorithm occasionally allows them to coalesce*, when the hash function is random. Of course $C$ can be as high as $N$, if the hash function is bad or if we are extremely unlucky.

In a successful search, we always have $A = 1$. The average number of probes during a successful search may be computed by summing the quantity $C + A$ over the first $N$ unsuccessful searches and dividing by $N$, if we assume that each key is equally likely. Thus we obtain

$$C_N = \frac{1}{N}\sum_{0\le k<N}\left(C'_k + \frac{k}{M}\right) = 1 + \frac{1}{8}\frac{M}{N}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) + \frac{1}{4}\frac{N-1}{M}$$

$$\approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{8\alpha} + \frac{\alpha}{4} \tag{16}$$

as the average number of probes in a random successful search. Even a full table will require only about 1.80 probes, on the average, to find an item! Similarly (see exercise 42), the average value of $S1$ turns out to be

$$S1_N = 1 - \tfrac{1}{2}\big((N-1)/M\big) \approx 1 - \tfrac{1}{2}\alpha. \tag{17}$$

At first glance it may appear that step C5 is inefficient, since it has to search sequentially for an empty position. But actually the total number of table probes

made in step C5 as a table is being built will never exceed the number of items in the table; so we make an average of at most one of these probes per insertion. Exercise 41 proves that $T$ is approximately $\alpha e^{\alpha}$ in a random unsuccessful search.

It would be possible to modify Algorithm C so that no two lists coalesce, but then it would become necessary to move records around. For example, consider the situation in Fig. 40 just before we wanted to insert SEKS into position 9; in order to keep the lists separate, it would be necessary to move FIRE, and for this purpose it would be necessary to discover which node points to FIRE. We could solve this problem without providing two-way linkage by hashing FIRE and searching down its list, as suggested by D. E. Ferguson, since the lists are short. Exercise 34 shows that the average number of probes, when lists aren't coalesced, is reduced to

$$C'_N = 1 + \frac{N(N-1)}{2M^2} \approx 1 + \frac{\alpha^2}{2} \quad \text{(unsuccessful search)}, \tag{18}$$

$$C_N = 1 + \frac{N-1}{2M} \approx 1 + \frac{\alpha}{2} \quad \text{(successful search)}. \tag{19}$$

This is not enough of an improvement over (15) and (16) to warrant changing the algorithm.

On the other hand, Butler Lampson has observed that most of the space that is occupied by links can actually be saved in the chaining method, if we avoid coalescing the lists. This leads to an interesting algorithm that is discussed in exercise 13. Lampson's method introduces a tag bit in each entry, and causes the average number of probes needed in an unsuccessful search to decrease slightly, from (18) to

$$\left(1 - \frac{1}{M}\right)^N + \frac{N}{M} \approx e^{-\alpha} + \alpha. \tag{18'}$$

Separate chaining as in Fig. 38 can be used when $N > M$, so overflow is not a serious problem in that case. When the lists coalesce as in Fig. 40 and Algorithm C, we can link extra items into an auxiliary storage pool; L. Guibas has proved that the average number of probes to insert the $(M + L + 1)$st item is then $\left(L/2M + \frac{1}{4}\right)\left((1 + 2/M)^M - 1\right) + \frac{1}{2}$. However, it is usually preferable to use an alternative scheme that puts the first colliding elements into an auxiliary storage area, allowing lists to coalesce only when this auxiliary area has filled up; see exercise 43.

**Collision resolution by "open addressing."** Another way to resolve the problem of collisions is to do away with links entirely, simply looking at various entries of the table one by one until either finding the key $K$ or finding an empty position. The idea is to formulate some rule by which every key $K$ determines a "probe sequence," namely a sequence of table positions that are to be inspected whenever $K$ is inserted or looked up. If we encounter an empty position while searching for $K$, using the probe sequence determined by $K$, we can conclude that $K$ is not in the table, since the same sequence of probes will be made every

time $K$ is processed. This general class of methods was named *open addressing* by W. W. Peterson [*IBM J. Research & Development* **1** (1957), 130–146].

The simplest open addressing scheme, known as *linear probing*, uses the cyclic probe sequence

$$h(K),\ h(K)-1,\ \ldots,\ 0,\ M-1,\ M-2,\ \ldots,\ h(K)+1 \tag{20}$$

as in the following algorithm.

**Algorithm L** (*Linear probing and insertion*). This algorithm searches an $M$-node table, looking for a given key $K$. If $K$ is not in the table and the table is not full, $K$ is inserted.

The nodes of the table are denoted by `TABLE[`$i$`]`, for $0 \le i < M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key, called `KEY[`$i$`]`, and possibly other fields. An auxiliary variable $N$ is used to keep track of how many nodes are occupied; this variable is considered to be part of the table, and it is increased by 1 whenever a new key is inserted.

This algorithm makes use of a hash function $h(K)$, and it uses the linear probing sequence (20) to address the table. Modifications of that sequence are discussed below.

**L1.** [Hash.] Set $i \leftarrow h(K)$. (Now $0 \le i < M$.)

**L2.** [Compare.] If `TABLE[`$i$`]` is empty, go to step L4. Otherwise if `KEY[`$i$`]` $= K$, the algorithm terminates successfully.

**L3.** [Advance to next.] Set $i \leftarrow i-1$; if now $i < 0$, set $i \leftarrow i+M$. Go back to step L2.

**L4.** [Insert.] (The search was unsuccessful.) If $N = M-1$, the algorithm terminates with overflow. (This algorithm considers the table to be full when $N = M-1$, not when $N = M$; see exercise 15.) Otherwise set $N \leftarrow N+1$, mark `TABLE[`$i$`]` occupied, and set `KEY[`$i$`]` $\leftarrow K$. ∎

Figure 41 shows what happens when the seven example keys (11) are inserted by Algorithm L, using the respective hash codes 2, 7, 1, 8, 2, 8, 1: The last three keys, `FEM`, `SEKS`, and `SYV`, have been displaced from their initial locations $h(K)$.

| | |
|---|---|
| 0 | FEM |
| 1 | TRE |
| 2 | EN |
| 3 | |
| 4 | |
| 5 | SYV |
| 6 | SEKS |
| 7 | TO |
| 8 | FIRE |

**Fig. 41.** Linear open addressing.

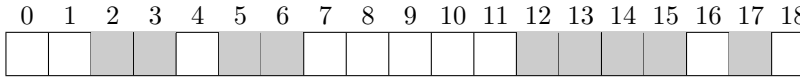
**Program L** (*Linear probing and insertion*). This program deals with full-word keys; but a key of 0 is not allowed, since 0 is used to signal an empty position in the table. (Alternatively, we could require the keys to be nonnegative, letting empty positions contain $-1$.) The table size $M$ is assumed to be prime, and `TABLE[`$i$`]` is stored in location `TABLE` $+ i$ for $0 \le i < M$. For speed in the inner loop, location `TABLE`$-1$ is assumed to contain 0. Location `VACANCIES` is assumed to contain the value $M - 1 - N$; and rA $\equiv K$, rI1 $\equiv i$.

In order to speed up the inner loop of this program, the test "$i < 0$" has been removed from the loop so that only the essential parts of steps L2 and L3 remain. The total running time for the searching phase comes to $(7C + 9E + 21 - 4S)u$, and the insertion after an unsuccessful search adds an extra $8u$.

```
01  START LDX  K              1          L1. Hash.
02        ENTA 0              1
03        DIV  =M=            1
04        STX  *+1(0:2)       1
05        ENT1 *              1          i ← h(K).
06        LDA  K              1
07        JMP  2F             1
08  8H    INC1 M+1            E          L3. Advance to next.
09  3H    DEC1 1          C+E-1          i ← i − 1.
10  2H    CMPA TABLE,1      C+E          L2. Compare.
11        JE   SUCCESS      C+E          Exit if K = KEY[i].
12        LDX  TABLE,1    C+E-S
13        JXNZ 3B         C+E-S          To L3 if TABLE[i] nonempty.
14        J1N  8B         E+1-S          To L3 with i ← M if i = −1.
15  4H    LDX  VACANCIES    1-S          L4. Insert.
16        JXZ  OVERFLOW     1-S          Exit with overflow if N = M − 1.
17        DECX 1            1-S
18        STX  VACANCIES    1-S          Increase N by 1.
19        STA  TABLE,1      1-S          TABLE[i] ← K.  ∎
```

As in Program C, the variable $C$ denotes the number of probes, and $S$ tells whether or not the search was successful. We may ignore the variable $E$, which is 1 only if a spurious probe of `TABLE[`$-1$`]` has been made, since its average value is $(C - 1)/M$.

Experience with linear probing shows that the algorithm works fine until the table begins to get full; but eventually the process slows down, with long drawn-out searches becoming increasingly frequent. The reason for this behavior can be understood by considering the following hypothetical hash table in which $M = 19$ and $N = 9$:




(21)

Shaded squares represent occupied positions. The next key $K$ to be inserted into the table will go into one of the ten empty spaces, but these are not equally likely; in fact, $K$ will be inserted into position 11 if $11 \le h(K) \le 15$, while it

will fall into position 8 only if $h(K) = 8$. Therefore position 11 is five times as likely as position 8; long lists tend to grow even longer.

This phenomenon isn't enough by itself to account for the relatively poor behavior of linear probing, since a similar thing occurs in Algorithm C. (A list of length 4 is four times as likely to grow in Algorithm C as a list of length 1.) The real problem occurs when a cell like 4 or 16 becomes occupied in (21); then two separate lists are combined, while the lists in Algorithm C never grow by more than one step at a time. Consequently the performance of linear probing degrades rapidly when $N$ approaches $M$.

We shall prove later in this section that the average number of probes needed by Algorithm L is approximately

$$C'_N \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) \quad \text{(unsuccessful search)}, \tag{22}$$

$$C_N \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \quad \text{(successful search)}, \tag{23}$$

where $\alpha = N/M$ is the load factor of the table. Therefore Program L is almost as fast as Program C, when the table is less than 75 percent full, in spite of the fact that Program C deals with unrealistically short keys. On the other hand, when $\alpha$ approaches 1 the best thing we can say about Program L is that it works, slowly but surely. In fact, when $N = M-1$, there is only one vacant space in the table, so the average number of probes in an unsuccessful search is $(M+1)/2$; we shall also prove that the average number of probes in a successful search is approximately $\sqrt{\pi M/8}$ when the table is full.

The pileup phenomenon that makes linear probing costly on a nearly full table is aggravated by the use of division hashing, if consecutive key values $\{K, K+1, K+2, \ldots\}$ are likely to occur, since these keys will have consecutive hash codes. Multiplicative hashing will break up these clusters satisfactorily.

Another way to protect against the consecutive hash code problem is to set $i \leftarrow i - c$ in step L3, instead of $i \leftarrow i - 1$. Any positive value of $c$ will do, so long as it is *relatively prime* to $M$, since the probe sequence will still examine every position of the table in this case. Such a change would make Program L a bit slower, because of the test for $i < 0$. Decreasing by $c$ instead of by 1 won't alter the pileup phenomenon, since groups of $c$-apart records will still be formed; equations (22) and (23) will still apply. But the appearance of consecutive keys $\{K, K+1, K+2, \ldots\}$ will now actually be a help instead of a hindrance.

Although a fixed value of $c$ does not reduce the pileup phenomenon, we can improve the situation nicely by letting $c$ depend on $K$. This idea leads to an important modification of Algorithm L, first introduced by Guy de Balbine [Ph.D. thesis, Calif. Inst. of Technology (1968), 149–150]:

**Algorithm D** (*Open addressing with double hashing*). This algorithm is almost identical to Algorithm L, but it probes the table in a slightly different fashion by making use of two hash functions $h_1(K)$ and $h_2(K)$. As usual $h_1(K)$ produces a value between 0 and $M-1$, inclusive; but $h_2(K)$ must produce a value between

1 and $M-1$ that is *relatively prime* to $M$. (For example, if $M$ is prime, $h_2(K)$ can be any value between 1 and $M-1$ inclusive; or if $M = 2^m$, $h_2(K)$ can be any *odd* value between 1 and $2^m - 1$.)

**D1.** [First hash.] Set $i \leftarrow h_1(K)$.

**D2.** [First probe.] If `TABLE[`$i$`]` is empty, go to D6. Otherwise if `KEY[`$i$`]` $= K$, the algorithm terminates successfully.

**D3.** [Second hash.] Set $c \leftarrow h_2(K)$.

**D4.** [Advance to next.] Set $i \leftarrow i - c$; if now $i < 0$, set $i \leftarrow i + M$.

**D5.** [Compare.] If `TABLE[`$i$`]` is empty, go to D6. Otherwise if `KEY[`$i$`]` $= K$, the algorithm terminates successfully. Otherwise go back to D4.

**D6.** [Insert.] If $N = M - 1$, the algorithm terminates with overflow. Otherwise set $N \leftarrow N + 1$, mark `TABLE[`$i$`]` occupied, and set `KEY[`$i$`]` $\leftarrow K$. ∎

Several possibilities have been suggested for computing $h_2(K)$. If $M$ is prime and $h_1(K) = K \bmod M$, we might let $h_2(K) = 1 + \big(K \bmod (M-1)\big)$; but since $M-1$ is even, it would be better to let $h_2(K) = 1 + \big(K \bmod (M-2)\big)$. This suggests choosing $M$ so that $M$ and $M-2$ are "twin primes" like 1021 and 1019. Alternatively, we could set $h_2(K) = 1 + \big(\lfloor K/M \rfloor \bmod (M-2)\big)$, since the quotient $\lfloor K/M \rfloor$ might be available in a register as a by-product of the computation of $h_1(K)$.

If $M = 2^m$ and we are using multiplicative hashing, $h_2(K)$ can be computed simply by shifting left $m$ more bits and "oring in" a 1, so that the coding sequence in (5) would be followed by

```
ENTA  0     Clear rA.
SLB   m     Shift rAX m bits left.
OR    =1=   rA ← rA | 1.
```
(24)

This is faster than the division method.

In each of the techniques suggested above, $h_1(K)$ and $h_2(K)$ are essentially independent, in the sense that different keys will yield the same values for both $h_1$ and $h_2$ with probability approximately proportional to $1/M^2$ instead of to $1/M$. Empirical tests show that the behavior of Algorithm D with independent hash functions is essentially indistinguishable from the number of probes that would be required if the keys were inserted at random into the table; there is practically no "piling up" or "clustering" as in Algorithm L.

It is also possible to let $h_2(K)$ depend on $h_1(K)$, as suggested by Gary Knott in 1968; for example, if $M$ is prime we could let

$$h_2(K) = \begin{cases} 1, & \text{if } h_1(K) = 0; \\ M - h_1(K), & \text{if } h_1(K) > 0. \end{cases} \tag{25}$$

This would be faster than doing another division, but we shall see that it does cause a certain amount of *secondary clustering*, requiring slightly more probes because of the increased chance that two or more keys will follow the same path. The formulas derived below can be used to determine whether the gain in hashing time outweighs the loss of probing time.

Algorithms L and D are very similar, yet there are enough differences that it is instructive to compare the running time of the corresponding MIX programs.

**Program D** (*Open addressing with double hashing*). Since this program is substantially like Program L, it is presented without comments. rI2 $\equiv c - 1$.

```
01  START LDX  K            1          15  3H DEC1 1,2          C-1
02        ENTA 0            1          16     J1NN *+2          C-1
03        DIV  =M=          1          17     INC1 M              B
04        STX  *+1(0:2)     1          18     CMPA TABLE,1      C-1
05        ENT1 *            1          19     JE   SUCCESS      C-1
06        LDX  TABLE,1      1          20     LDX  TABLE,1    C-1-S2
07        CMPX K            1          21     JXNZ 3B         C-1-S2
08        JE   SUCCESS      1          22  4H LDX  VACANCIES     1-S
09        JXZ  4F         1-S1         23     JXZ  OVERFLOW      1-S
10        SRAX 5          A-S1         24     DECX 1             1-S
11        DIV  =M-2=      A-S1         25     STX  VACANCIES     1-S
12        STX  *+1(0:2)   A-S1         26     LDA  K             1-S
13        ENT2 *          A-S1         27     STA  TABLE,1       1-S
14        LDA  K          A-S1
```

The frequency counts $A$, $C$, $S1$, $S2$ in this program have a similar interpretation to those in Program C above. The other variable $B$ will be about $(C-1)/2$ on the average. (If we restricted the range of $h_2(K)$ to, say, $1 \le h_2(K) \le M/2$, $B$ would be only about $(C-1)/4$; this increase of speed will probably *not* be offset by a noticeable increase in the number of probes.) When there are $N = \alpha M$ keys in the table, the average value of $A$ is, of course, $\alpha$ in an unsuccessful search, and $A = 1$ in a successful search. As in Algorithm C, the average value of $S1$ in a successful search is $1 - \frac{1}{2}\big((N-1)/M\big) \approx 1 - \frac{1}{2}\alpha$. The average number of probes is difficult to determine exactly, but empirical tests show good agreement with formulas derived below for "uniform probing," namely

$$C'_N = \frac{M+1}{M+1-N} \approx (1-\alpha)^{-1} \quad \text{(unsuccessful search)}, \tag{26}$$

$$C_N = \frac{M+1}{N}(H_{M+1} - H_{M+1-N}) \approx -\alpha^{-1}\ln(1-\alpha) \quad \text{(successful search)}, \tag{27}$$

when $h_1(K)$ and $h_2(K)$ are independent. When $h_2(K)$ depends on $h_1(K)$ as in (25), the secondary clustering causes (26) and (27) to be increased to

$$C'_N = \frac{M+1}{M+1-N} - \frac{N}{M+1} + H_{M+1} - H_{M+1-N} + O(M^{-1}) \approx (1-\alpha)^{-1} - \alpha - \ln(1-\alpha); \tag{28}$$

$$C_N = 1 + H_{M+1} - H_{M+1-N} - \frac{N}{2(M+1)} - (H_{M+1} - H_{M+1-N})/N + O(N^{-1}) \approx 1 - \ln(1-\alpha) - \tfrac{1}{2}\alpha. \tag{29}$$

(See exercise 44.) Note that as the table gets full, these values of $C_N$ approach $H_{M+1} - 1$ and $H_{M+1} - \frac{1}{2}$, respectively, when $N = M$; this is much better than we observed in Algorithm L, but not as good as in the chaining methods.

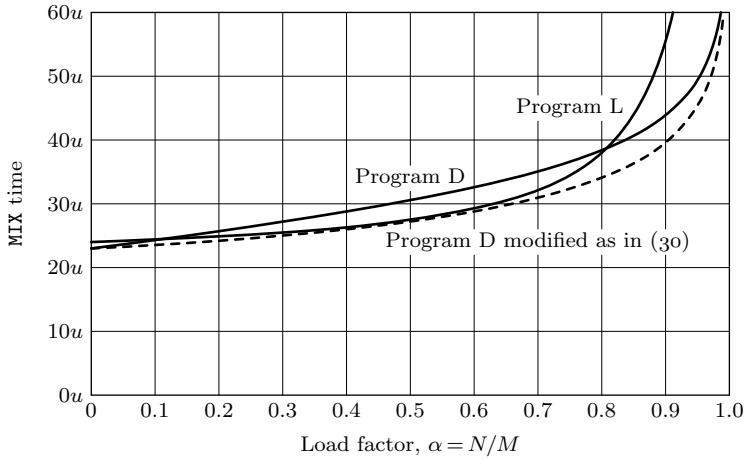



**Fig. 42.** The running time for successful searching by three open addressing schemes.

Since each probe takes slightly less time in Algorithm L, double hashing is advantageous only when the table gets full. Figure 42 compares the average running time of Program L, Program D, and a modified Program D that involves secondary clustering, replacing the rather slow calculation of $h_2(K)$ in lines 10–13 by the following three instructions:

```
ENN2 1-M,1     c ← M − i.
J1NZ *+2
ENT2 0         If i = 0, c ← 1.                    (30)
```

Program D takes a total of $8C + 19A + B + 26 - 13S - 17S1$ units of time; modification (30) saves about $15(A - S1) \approx 7.5\alpha$ of these in a successful search. In this case, secondary clustering is preferable to independent double hashing.

On a binary computer, we could speed up the computation of $h_2(K)$ in another way, if $M$ is prime greater than, say, 512, replacing lines 10–13 by

```
AND  =511=     rA ← rA mod 512.
STA  *+1(0:2)
ENT2 *         c ← rA + 1.                         (31)
```

This idea (suggested by Bell and Kaman, *CACM* **13** (1970), 675–677, who discovered Algorithm D independently) avoids secondary clustering without the expense of another division.

Many other probe sequences have been proposed as improvements on Algorithm L, but none seem to be superior to Algorithm D except possibly the method described in exercise 20.

By using the relative order of keys we can reduce the average running time for unsuccessful searches by Algorithms L or D to the average running time for successful search; see exercise 66. This technique can be important in applications for which unsuccessful searches are common; for example, TEX uses such an algorithm when looking for exceptions to its hyphenation rules.

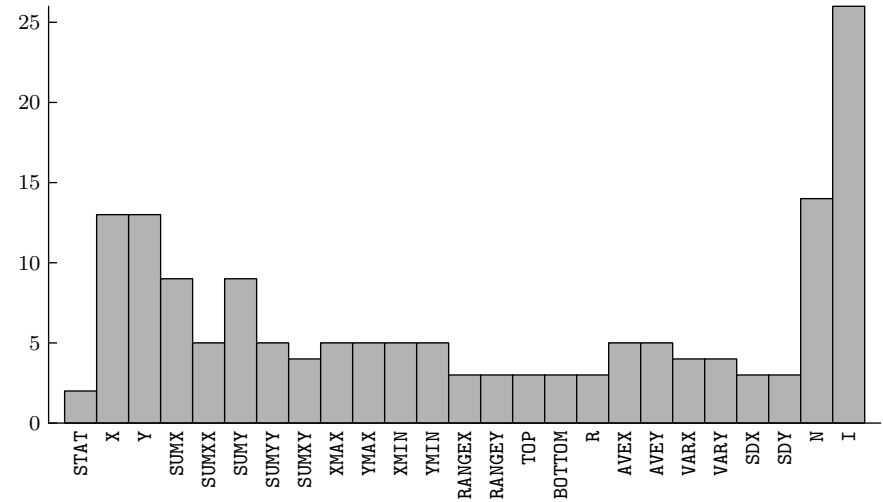



**Fig. 43.** The number of times a compiler typically searches for variable names. The names are listed from left to right in order of their first appearance.

**Brent's Variation.** Richard P. Brent has discovered a way to modify Algorithm D so that the average successful search time remains bounded as the table gets full. His method [*CACM* **16** (1973), 105–109] is based on the fact that successful searches are much more common than insertions, in many applications; therefore he proposes doing more work when inserting an item, moving records in order to reduce the expected retrieval time.

For example, Fig. 43 shows the number of times each identifier was actually found to appear, in a typical PL/I procedure. This data indicates that a PL/I compiler that uses a hash table to keep track of variable names will be looking up many of the names five or more times but inserting them only once. Similarly, Bell and Kaman found that a COBOL compiler used its symbol table algorithm 10988 times while compiling a program, but made only 735 insertions into the table; this is an average of about 14 successful searches per unsuccessful search. Sometimes a table is actually created only once (for example, a table of symbolic opcodes in an assembler), and it is used thereafter purely for retrieval.

Brent's idea is to change the insertion process in Algorithm D as follows. Suppose an unsuccessful search has probed locations $p_0$, $p_1$, $\ldots$, $p_{t-1}$, $p_t$, where $p_j = \big(h_1(K) - jh_2(K)\big) \bmod M$ and `TABLE[`$p_t$`]` is empty. If $t \le 1$, we insert $K$ in position $p_t$ as usual; but if $t \ge 2$, we compute $c_0 = h_2(K_0)$, where $K_0 =$ `KEY[`$p_0$`]`, and see if `TABLE[`$(p_0 - c_0) \bmod M$`]` is empty. If it is, we set it to `TABLE[`$p_0$`]` and then insert $K$ in position $p_0$. This increases the retrieval time for $K_0$ by one step, but it decreases the retrieval time for $K$ by $t \ge 2$ steps, so it results in a net improvement. Similarly, if `TABLE[`$(p_0 - c_0) \bmod M$`]` is occupied and $t \ge 3$, we try `TABLE[`$(p_0 - 2c_0) \bmod M$`]`; if that is full too, we compute $c_1 = h_2($`KEY[`$p_1$`]`$)$ and try `TABLE[`$(p_1 - c_1) \bmod M$`]`; etc. In general, let $c_j = h_2($`KEY[`$p_j$`]`$)$ and

$p_{j,k} = (p_j - kc_j) \bmod M$; if we have found TABLE[$p_{j,k}$] occupied for all indices $j$ and $k$ such that $j+k < r$, and if $t \geq r+1$, we look at TABLE[$p_{0,r}$], TABLE[$p_{1,r-1}$], ..., TABLE[$p_{r-1,1}$]. If the first empty space occurs at position $p_{j,r-j}$ we set TABLE[$p_{j,r-j}$] $\leftarrow$ TABLE[$p_j$] and insert $K$ in position $p_j$.

Brent's analysis indicates that the average number of probes per successful search is reduced to the levels shown in Fig. 44, on page 545, with a maximum value of about 2.49.

The number $t+1$ of probes in an unsuccessful search is not reduced by Brent's variation; it remains at the level indicated by Eq. (26), approaching $\frac{1}{2}(M+1)$ as the table gets full. The average number of times $h_2$ needs to be computed per insertion is $\alpha^2 + \alpha^5 + \frac{1}{3}\alpha^6 + \cdots$, according to Brent's analysis, eventually approaching $\Theta(\sqrt{M})$; and the number of additional table positions probed while deciding how to make the insertion is about $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^6 + \cdots$.

E. G. Mallach [*Comp. J.* **20** (1977), 137–140] has experimented with refinements of Brent's variation, and further results have been obtained by Gaston H. Gonnet and J. Ian Munro [*SICOMP* **8** (1979), 463–478].

**Deletions.** Many computer programmers have great faith in algorithms, and they are surprised to find that *the obvious way to delete records from a hash table doesn't work*. For example, if we try to delete the key EN from Fig. 41, we can't simply mark that table position empty, because another key FEM would suddenly be forgotten! (Recall that EN and FEM both hashed to the same location. When looking up FEM, we would find an empty place, indicating an unsuccessful search.) A similar problem occurs with Algorithm C, due to the coalescing of lists; imagine the deletion of both TO and FIRE from Fig. 40.

In general, we can handle deletions by putting a special code value in the corresponding cell, so that there are three kinds of table entries: empty, occupied, and deleted. When searching for a key, we should skip over deleted cells, as if they were occupied. If the search is unsuccessful, the key can be inserted in place of the first deleted or empty position that was encountered.

But this idea is workable only when deletions are very rare, because the entries of the table never become empty again once they have been occupied. After a long sequence of repeated insertions and deletions, all of the empty spaces will eventually disappear, and every unsuccessful search will take $M$ probes! Furthermore the time per probe will be increased, since we will have to test whether $i$ has returned to its starting value in step D4; and the number of probes in a successful search will drift upward from $C_N$ to $C'_N$.

When linear probing is being used (Algorithm L), we can make deletions in a way that avoids such a sorry state of affairs, if we are willing to do some extra work for the deletion.

**Algorithm R** (*Deletion with linear probing*). Assuming that an open hash table has been constructed by Algorithm L, this algorithm deletes the record from a given position TABLE[$i$].

**R1.** [Empty a cell.] Mark TABLE[$i$] empty, and set $j \leftarrow i$.

**R2.** [Decrease $i$.] Set $i \leftarrow i - 1$, and if this makes $i$ negative set $i \leftarrow i + M$.

**R3.** [Inspect `TABLE[`$i$`]`.] If `TABLE[`$i$`]` is empty, the algorithm terminates. Otherwise set $r \leftarrow h($`KEY[`$i$`]`$)$, the original hash address of the key now stored at position $i$. If $i \le r < j$ or if $r < j < i$ or $j < i \le r$ (in other words, if $r$ lies cyclically between $i$ and $j$), go back to R2.

**R4.** [Move a record.] Set `TABLE[`$j$`]` $\leftarrow$ `TABLE[`$i$`]`, and return to step R1. ∎

Exercise 22 shows that this algorithm causes no degradation in performance; in other words, the average number of probes predicted in Eqs. (22) and (23) will remain the same. (A weaker result for tree insertion was proved in Theorem 6.2.2H.) But the validity of Algorithm R depends heavily on the fact that linear probing is involved, and no analogous deletion procedure for use with Algorithm D is possible. The average running time of Algorithm R is analyzed in exercise 64.

Of course when chaining is used with separate lists for each possible hash value, deletion causes no problems since it is simply a deletion from a linked linear list. Deletion with Algorithm C is discussed in exercise 23.

Algorithm R may move some of the table entries, and this is undesirable if they are being pointed to from elsewhere. Another approach to deletions is possible by adapting some of the ideas used in garbage collection (see Section 2.3.5): We might keep a reference count with each key telling how many other keys collide with it; then it is possible to convert unoccupied cells to empty status when their reference drops to zero. Alternatively we might go through the entire table whenever too many deleted entries have accumulated, changing all the unoccupied positions to empty and then looking up all remaining keys, in order to see which unoccupied positions still require "deleted" status. These procedures, which avoid relocation and work with any hash technique, were originally suggested by T. Gunji and E. Goto [*J. Information Proc.* **3** (1980), 1–12].

***Analysis of the algorithms.** It is especially important to know the average behavior of a hashing method, because we are committed to trusting in the laws of probability whenever we hash. The worst case of these algorithms is almost unthinkably bad, so we need to be reassured that the average behavior is very good.

Before we get into the analysis of linear probing, etc., let us consider an approximate model of the situation, called *uniform probing*. In this model, which was suggested by W. W. Peterson [*IBM J. Research & Devel.* **1** (1957), 135–136], we assume that each key is placed in a completely random location of the table, so that each of the $\binom{M}{N}$ possible configurations of $N$ occupied cells and $M-N$ empty cells is equally likely. This model ignores any effect of primary or secondary clustering; the occupancy of each cell in the table is essentially independent of all the others. Then the probability that any permutation of table positions needs exactly $r$ probes to insert the $(N+1)$st item is the number of configurations in which $r-1$ given cells are occupied and another is empty, divided by $\binom{M}{N}$, namely

$$P_r = \binom{M-r}{N-r+1} \Big/ \binom{M}{N};$$

therefore the average number of probes for uniform probing is

$$\begin{aligned} C'_N &= \sum_{r=1}^{M} rP_r = M+1-\sum_{r=1}^{M}(M+1-r)P_r \\ &= M+1-\sum_{r=1}^{M}(M+1-r)\binom{M-r}{M-N-1}\Big/\binom{M}{N} \\ &= M+1-\sum_{r=1}^{M}(M-N)\binom{M+1-r}{M-N}\Big/\binom{M}{N} \\ &= M+1-(M-N)\binom{M+1}{M-N+1}\Big/\binom{M}{N} \\ &= M+1-(M-N)\frac{M+1}{M-N+1} = \frac{M+1}{M-N+1}, \quad \text{for } 1 \le N < M. \end{aligned} \tag{32}$$

(We have already solved essentially the same problem in connection with random sampling, in exercise 3.4.2–5.) Setting $\alpha = N/M$, this exact formula for $C'_N$ is approximately equal to

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots, \tag{33}$$

a series that has a rough intuitive interpretation: With probability $\alpha$ we need more than one probe, with probability $\alpha^2$ we need more than two, etc. The corresponding average number of probes for a successful search is

$$\begin{aligned} C_N = \frac{1}{N}\sum_{k=0}^{N-1} C'_k &= \frac{M+1}{N}\Big(\frac{1}{M+1} + \frac{1}{M} + \cdots + \frac{1}{M-N+2}\Big) \\ &= \frac{M+1}{N}\big(H_{M+1} - H_{M-N+1}\big) \approx \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \end{aligned} \tag{34}$$

As remarked above, extensive tests show that Algorithm D with two independent hash functions behaves essentially like uniform probing, for all practical purposes. In fact, double hashing is asymptotically equivalent to uniform probing, in the limit as $M \to \infty$ (see exercise 70).

This completes our analysis of uniform probing. In order to study linear probing and other types of collision resolution, we need to set up the theory in a different, more realistic way. The probabilistic model we shall use for this purpose assumes that each of the $M^N$ possible "hash sequences"

$$a_1\, a_2 \ldots a_N, \qquad 0 \le a_j < M, \tag{35}$$

is equally likely, where $a_j$ denotes the initial hash address of the $j$th key inserted into the table. The average number of probes in a successful search, given any particular searching algorithm, will be denoted by $C_N$ as above; this is assumed to be the average number of probes needed to find the $k$th key, averaged over $1 \le k \le N$ with each key equally likely, and averaged over all hash sequences (35) with each sequence equally likely. Similarly, the average number of probes needed

when the $N$th key is inserted, considering all sequences (35) to be equally likely, will be denoted by $C'_{N-1}$; this is the average number of probes in an unsuccessful search starting with $N-1$ elements in the table. When open addressing is used,

$$C_N = \frac{1}{N}\sum_{k=0}^{N-1} C'_k, \tag{36}$$

so that we can deduce one quantity from the other as we have done in (34).

Strictly speaking, there are two defects even in this more accurate model. In the first place, the different hash sequences aren't all equally probable, because the keys themselves are distinct. This makes the probability that $a_1 = a_2$ slightly less than $1/M$; but the difference is usually negligible since the set of all possible keys is typically very large compared to $M$. (See exercise 24.) Furthermore a good hash function will exploit the nonrandomness of typical data, making it even less likely that $a_1 = a_2$; as a result, our estimates for the number of probes will be pessimistic. Another inaccuracy in the model is indicated in Fig. 43: Keys that occur earlier are (with some exceptions) more likely to be looked up than keys that occur later. Therefore our estimate of $C_N$ tends to be doubly pessimistic, and the algorithms should perform slightly better in practice than our analysis predicts.

With these precautions, we are ready to make an "exact" analysis of linear probing.* Let $f(M,N)$ be the number of hash sequences (35) such that position 0 of the table will be empty after the keys have been inserted by Algorithm L. The circular symmetry of linear probing implies that position 0 is empty just as often as any other position, so it is empty with probability $1 - N/M$; in other words

$$f(M,N) = \left(1 - \frac{N}{M}\right) M^N. \tag{37}$$

By convention we also set $f(0,0) = 1$. Now let $g(M,N,k)$ be the number of hash sequences (35) such that the algorithm leaves position 0 empty, positions 1 through $k$ occupied, and position $k+1$ empty. We have

$$g(M,N,k) = \binom{N}{k} f(k{+}1,k)\, f(M{-}k{-}1, N{-}k), \tag{38}$$

because all such hash sequences are composed of two subsequences, one (containing $k$ elements $a_i \le k$) that leaves position 0 empty and positions 1 through $k$ occupied and one (containing $N-k$ elements $a_j \ge k+1$) that leaves position $k+1$ empty; there are $f(k{+}1,k)$ subsequences of the former type and $f(M{-}k{-}1, N{-}k)$ of the latter type, and there are $\binom{N}{k}$ ways to intersperse two such subsequences. Finally let $P_k$ be the probability that exactly $k+1$ probes will be needed when the $(N+1)$st key is inserted; it follows (see exercise 25)

* The author cannot resist inserting a biographical note at this point: I first formulated the following derivation in 1962, shortly after beginning work on *The Art of Computer Programming*. Since this was the first nontrivial algorithm I had ever analyzed satisfactorily, it had a strong influence on the structure of these books. Ever since that day, the analysis of algorithms has in fact been one of the major themes of my life.

that

$$P_k = M^{-N}\big(g(M,N,k) + g(M,N,k+1) + \cdots + g(M,N,N)\big). \tag{39}$$

Now $C'_N = \sum_{k=0}^{N}(k+1)P_k$; putting this equation together with (36)–(39) and simplifying yields the following result.

**Theorem K.** *The average number of probes needed by Algorithm L, assuming that all $M^N$ hash sequences (35) are equally likely, is*

$$C_N = \tfrac{1}{2}\big(1 + Q_0(M,N-1)\big) \quad \text{(successful search)}, \tag{40}$$

$$C'_N = \tfrac{1}{2}\big(1 + Q_1(M,N)\big) \quad \text{(unsuccessful search)}, \tag{41}$$

*where*

$$\begin{aligned} Q_r(M,N) &= \binom{r}{0} + \binom{r+1}{1}\frac{N}{M} + \binom{r+2}{2}\frac{N(N-1)}{M^2} + \cdots \\ &= \sum_{k\ge 0}\binom{r+k}{k}\frac{N}{M}\frac{N-1}{M}\cdots\frac{N-k+1}{M}. \end{aligned} \tag{42}$$

*Proof.* Details of the calculation are worked out in exercise 27. (For the variance, see exercises 28, 67, and 68.) ∎

The rather strange-looking function $Q_r(M,N)$ that appears in this theorem is really not hard to deal with. We have

$$N^k - \binom{k}{2}N^{k-1} \le N(N-1)\ldots(N-k+1) \le N^k;$$

hence if $N/M = \alpha$,

$$\sum_{k\ge 0}\binom{r+k}{k}\left(N^k - \binom{k}{2}N^{k-1}\right)\Big/M^k \le Q_r(M,N) \le \sum_{k\ge 0}\binom{r+k}{k}N^k/M^k,$$

$$\sum_{k\ge 0}\binom{r+k}{k}\alpha^k - \frac{\alpha}{M}\sum_{k\ge 0}\binom{r+k}{k}\binom{k}{2}\alpha^{k-2} \le Q_r(M,\alpha M) \le \sum_{k\ge 0}\binom{r+k}{k}\alpha^k;$$

that is,

$$\frac{1}{(1-\alpha)^{r+1}} - \frac{1}{M}\binom{r+2}{2}\frac{\alpha}{(1-\alpha)^{r+3}} \le Q_r(M,\alpha M) \le \frac{1}{(1-\alpha)^{r+1}}. \tag{43}$$

This relation gives us a good estimate of $Q_r(M,N)$ when $M$ is large and $\alpha$ is not too close to 1. (The lower bound is a better approximation than the upper bound.) When $\alpha$ approaches 1, these formulas become useless, but fortunately $Q_0(M,M-1)$ is the function $Q(M)$ whose asymptotic behavior was studied in great detail in Section 1.2.11.3; and $Q_1(M,M-1)$ is simply equal to $M$ (see exercise 50). In terms of the standard notation for hypergeometric functions, Eq. 1.2.6–(39), we have $Q_r(M,N) = F(r+1,-N;;-1/M) = F\big(\begin{smallmatrix} r+1,-N,1 \\ 1 \end{smallmatrix}\big|-\frac{1}{M}\big)$.

Another approach to the analysis of linear probing was taken in the early days by G. Schay, Jr. and W. G. Spruth [*CACM* **5** (1962), 459–462]. Although their method yielded only an approximation to the exact formulas in Theorem K, it sheds further light on the algorithm, so we shall sketch it briefly here. First let us consider a surprising property of linear probing that was first noticed by W. W. Peterson in 1957:

**Theorem P.** *The average number of probes in a successful search by Algorithm L is independent of the order in which the keys were inserted; it depends only on the number of keys that hash to each address.*

In other words, any rearrangement of a hash sequence $a_1\, a_2 \ldots a_N$ yields a hash sequence with the same average displacement of keys from their hash addresses. (We are assuming, as stated earlier, that all keys in the table have equal importance. If some keys are more frequently accessed than others, the proof can be extended to show that an optimal arrangement occurs if we insert them in decreasing order of frequency, using the method of Theorem 6.1S.)

*Proof.* It suffices to show that the total number of probes needed to insert keys for the hash sequence $a_1\, a_2 \ldots a_N$ is the same as the total number needed for $a_1 \ldots a_{i-1}\, a_{i+1}\, a_i\, a_{i+2} \ldots a_N$, $1 \le i < N$. There is clearly no difference unless the $(i+1)$st key in the second sequence falls into the position occupied by the $i$th in the first sequence. But then the $i$th and $(i+1)$st merely exchange places, so the number of probes for the $(i+1)$st is decreased by the same amount that the number for the $i$th is increased. ∎

Theorem P tells us that the average search length for a hash sequence $a_1\, a_2 \ldots a_N$ can be determined from the numbers $b_0\, b_1 \ldots b_{M-1}$, where $b_j$ is the number of $a$'s that equal $j$. From this sequence we can determine the "carry sequence" $c_0\, c_1 \ldots c_{M-1}$, where $c_j$ is the number of keys for which both locations $j$ and $j-1$ are probed as the key is inserted. This sequence is determined by the rule

$$c_j = \begin{cases} 0, & \text{if } b_j = c_{(j+1) \bmod M} = 0; \\ b_j + c_{(j+1) \bmod M} - 1, & \text{otherwise.} \end{cases} \tag{44}$$

For example, let $M = 10$, $N = 8$, and $b_0 \ldots b_9 = 0\ 3\ 2\ 0\ 1\ 0\ 0\ 0\ 0\ 2$; then $c_0 \ldots c_9 = 2\ 3\ 1\ 0\ 0\ 0\ 0\ 1\ 2\ 3$, since one key needs to be "carried over" from position 2 to position 1, three from position 1 to position 0, two of these from position 0 to position 9, etc. We have $b_0 + b_1 + \cdots + b_{M-1} = N$, and the average number of probes needed for retrieval of the $N$ keys is

$$1 + (c_0 + c_1 + \cdots + c_{M-1})/N. \tag{45}$$

Rule (44) seems to be a circular definition of the $c$'s in terms of themselves, but actually there is a unique solution to the stated equations whenever $N < M$ (see exercise 32).

Schay and Spruth used this idea to determine the probability $q_k$ that $c_j = k$, in terms of the probability $p_k$ that $b_j = k$. (These probabilities are independent

of $j$.) Thus

$$\begin{aligned} q_0 &= p_0q_0 + p_1q_0 + p_0q_1, \\ q_1 &= p_2q_0 + p_1q_1 + p_0q_2, \\ q_2 &= p_3q_0 + p_2q_1 + p_1q_2 + p_0q_3, \end{aligned} \tag{46}$$

etc., since, for example, the probability that $c_j = 2$ is the probability that $b_j + c_{(j+1) \bmod M} = 3$. Let $B(z) = \sum p_k z^k$ and $C(z) = \sum q_k z^k$ be the generating functions for these probability distributions; the equations (46) are equivalent to

$$B(z)C(z) = p_0q_0 + (q_0 - p_0q_0)z + q_1z^2 + \cdots = p_0q_0(1 - z) + zC(z).$$

Since $B(1) = 1$, we may write $B(z) = 1 + (z - 1)D(z)$, and it follows that

$$C(z) = \frac{p_0q_0}{1 - D(z)} = \frac{1 - D(1)}{1 - D(z)}, \tag{47}$$

since $C(1) = 1$. The average number of probes needed for retrieval, according to (45), will therefore be

$$1 + \frac{M}{N}C'(1) = 1 + \frac{M}{N}\frac{D'(1)}{1 - D(1)} = 1 + \frac{M}{2N}\frac{B''(1)}{1 - B'(1)}. \tag{48}$$

Since we are assuming that each hash sequence $a_1 \ldots a_N$ is equally likely, we have

$$\begin{aligned} p_k &= \Pr(\text{exactly } k \text{ of the } a_i \text{ are equal to } j, \text{ for fixed } j) \\ &= \binom{N}{k}\left(\frac{1}{M}\right)^k\left(1 - \frac{1}{M}\right)^{N-k}; \end{aligned} \tag{49}$$

hence

$$B(z) = \left(1 + \frac{z - 1}{M}\right)^N, \qquad B'(1) = \frac{N}{M}, \qquad B''(1) = \frac{N(N - 1)}{M^2}, \tag{50}$$

and the average number of probes according to (48) will be

$$C_N = \frac{1}{2}\left(1 + \frac{M - 1}{M - N}\right). \tag{51}$$

Can the reader spot the incorrect reasoning that has caused this answer to be different from the correct result in Theorem K? (See exercise 33.)

***Optimality considerations.** We have seen several examples of probe sequences for open addressing, and it is natural to ask for one that can be proved *best possible* in some meaningful sense. This problem has been set up in the following interesting way by J. D. Ullman [*JACM* **19** (1972), 569–575]: Instead of computing a hash address $h(K)$, we map each key $K$ into an entire permutation of $\{0, 1, \ldots, M-1\}$, which represents the probe sequence to use for $K$. Each of the $M!$ permutations is assigned a probability, and the generalized hash function is supposed to select each permutation with that probability. The question is, "What assignment of probabilities to permutations gives the best performance,

in the sense that the corresponding average number of probes $C_N$ or $C'_N$ is minimized?"

For example, if we assign the probability $1/M!$ to each permutation, it is easy to see that we have exactly the behavior of *uniform probing* that we have analyzed above in (32) and (34). However, Ullman found an example with $M = 4$ and $N = 2$ for which $C'_N$ is smaller than the value $\frac{5}{3}$ obtained with uniform probing. His construction assigns zero probability to all but the following six permutations:

| Permutation | Probability | Permutation | Probability |
|---|---|---|---|
| 0 1 2 3 | $(1+2\epsilon)/6$ | 1 0 3 2 | $(1+2\epsilon)/6$ |
| 2 0 1 3 | $(1-\epsilon)/6$ | 2 1 0 3 | $(1-\epsilon)/6$ |
| 3 0 1 2 | $(1-\epsilon)/6$ | 3 1 0 2 | $(1-\epsilon)/6$ |

(52)

Roughly speaking, the first probe tends to be either 2 or 3, but the second probe is always 0 or 1. The average number of probes needed to insert the third item, $C'_2$, turns out to be $\frac{5}{3} - \frac{1}{9}\epsilon + O(\epsilon^2)$, so we can improve on uniform probing by taking $\epsilon$ to be a small positive value.

However, the corresponding value of $C'_1$ for these probabilities is $\frac{23}{18} + O(\epsilon)$, which is larger than $\frac{5}{4}$ (the uniform probing value). Ullman proved that any assignment of probabilities such that $C'_N < (M+1)/(M+1-N)$ for some $N$ always implies that $C'_n > (M+1)/(M+1-n)$ for some $n < N$; you can't win all the time over uniform probing.

Actually the number of probes $C_N$ for a *successful* search is a better measure than $C'_N$. The permutations in (52) do not lead to an improved value of $C_N$ for any $N$, and indeed Ullman conjectured that no assignment of probabilities will be able to make $C_N$ less than the uniform value $\big((M+1)/N\big)(H_{M+1} - H_{M+1-N})$. Andrew Yao proved an asymptotic form of this conjecture by showing that the limiting cost when $N = \alpha M$ and $M \to \infty$ is always $\geq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ [*JACM* **32** (1985), 687–693].

The strong form of Ullman's conjecture appears to be very difficult to prove, especially because there are many ways to assign probabilities to achieve the effect of uniform probing; we do not need to assign $1/M!$ to each permutation. For example, the following assignment for $M = 4$ is equivalent to uniform probing:

| Permutation | Probability | Permutation | Probability |
|---|---|---|---|
| 0 1 2 3 | 1/6 | 0 2 1 3 | 1/12 |
| 1 2 3 0 | 1/6 | 1 3 2 0 | 1/12 |
| 2 3 0 1 | 1/6 | 2 0 3 1 | 1/12 |
| 3 0 1 2 | 1/6 | 3 1 0 2 | 1/12 |

(53)

with zero probability assigned to the other 16 permutations.

The following theorem characterizes *all* assignments that produce the behavior of uniform probing.

**Theorem U.** *An assignment of probabilities to permutations will make each of the* $\binom{M}{N}$ *configurations of empty and occupied cells equally likely after* $N$

*insertions, for $0 < N < M$, if and only if the sum of probabilities assigned to all permutations whose first $N$ elements are the members of a given $N$-element set is $1/\binom{M}{N}$, for all $N$ and for all $N$-element sets.*

For example, the sum of probabilities assigned to each of the $3!(M-3)!$ permutations beginning with the numbers $\{0,1,2\}$ in some order must be $1/\binom{M}{3} = 3!(M-3)!/M!$. Observe that the condition of this theorem holds in (53), because $1/6 + 1/12 = 1/4$.

*Proof.* Let $A \subseteq \{0,1,\ldots,M-1\}$, and let $\Pi(A)$ be the set of all permutations whose first $|A|$ elements are members of $A$; also let $S(A)$ be the sum of the probabilities assigned to those permutations. Let $P_k(A)$ be the probability that the first $|A|$ insertions of the open addressing procedure occupy the locations specified by $A$, and that the last insertion required exactly $k$ probes. Finally, let $P(A) = P_1(A) + P_2(A) + \cdots$. The proof is by induction on $N \ge 1$, assuming that

$$P(A) = S(A) = 1\Big/\binom{M}{n}$$

for all sets $A$ with $|A| = n < N$. Let $B$ be any $N$-element set. Then

$$P_k(B) = \sum_{\substack{A\subseteq B\\ |A|=k}} \sum_{\pi\in\Pi(A)} \Pr(\pi) P\big(B \setminus \{\pi_k\}\big),$$

where $\Pr(\pi)$ is the probability assigned to permutation $\pi$ and $\pi_k$ is its $k$th element. By induction

$$P_k(B) = \sum_{\substack{A\subseteq B\\ |A|=k}} \frac{1}{\binom{M}{N-1}} \sum_{\pi\in\Pi(A)} \Pr(\pi),$$

which equals

$$\binom{N}{k}\Big/\binom{M}{N-1}\binom{M}{k}, \qquad \text{if } k < N;$$

hence

$$P(B) = \frac{1}{\binom{M}{N-1}}\left(S(B) + \sum_{k=1}^{N-1} \frac{\binom{N}{k}}{\binom{M}{k}}\right),$$

and this can be equal to $1/\binom{M}{N}$ if and only if $S(B)$ has the correct value. ∎

**External searching.** Hashing techniques lend themselves well to external searching on direct-access storage devices like disks or drums. For such applications, as in Section 6.2.4, we want to minimize the number of accesses to the file, and this has two major effects on the choice of algorithms:

1) It is reasonable to spend more time computing the hash function, since the penalty for bad hashing is much greater than the cost of the extra time needed to do a careful job.
2) The records are usually grouped into pages or *buckets*, so that several records are fetched from the external memory each time.

The file is divided into $M$ buckets containing $b$ records each. Collisions now cause no problem unless more than $b$ keys have the same hash address. The following three approaches to collision resolution seem to be best:

A) *Chaining with separate lists.* If more than $b$ records fall into the same bucket, a link to an overflow record can be inserted at the end of the first bucket. These overflow records are kept in a special overflow area. There is usually no advantage in having buckets in the overflow area, since comparatively few overflows occur; thus, the extra records are usually linked together so that the $(b + k)$th record of a list requires $1 + k$ accesses. It is usually a good idea to leave some room for overflows on each cylinder of a disk file, so that most accesses are to the same cylinder.

Although this method of handling overflows seems inefficient, the number of overflows is statistically small enough that the average search time is very good. See Tables 2 and 3, which show the average number of accesses required as a function of the load factor

$$\alpha = N/Mb, \tag{54}$$

for fixed $\alpha$ as $M$, $N \to \infty$. Curiously when $\alpha = 1$ the asymptotic number of accesses for an unsuccessful search increases with increasing $b$.

**Table 2**

AVERAGE ACCESSES IN AN UNSUCCESSFUL SEARCH BY SEPARATE CHAINING

| Bucket size, $b$ | Load factor, $\alpha$ 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 95% |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.0048 | 1.0187 | 1.0408 | 1.0703 | 1.1065 | 1.1488 | 1.197 | 1.249 | 1.307 | 1.34 |
| 2 | 1.0012 | 1.0088 | 1.0269 | 1.0581 | 1.1036 | 1.1638 | 1.238 | 1.327 | 1.428 | 1.48 |
| 3 | 1.0003 | 1.0038 | 1.0162 | 1.0433 | 1.0898 | 1.1588 | 1.252 | 1.369 | 1.509 | 1.59 |
| 4 | 1.0001 | 1.0016 | 1.0095 | 1.0314 | 1.0751 | 1.1476 | 1.253 | 1.394 | 1.571 | 1.67 |
| 5 | 1.0000 | 1.0007 | 1.0056 | 1.0225 | 1.0619 | 1.1346 | 1.249 | 1.410 | 1.620 | 1.74 |
| 10 | 1.0000 | 1.0000 | 1.0004 | 1.0041 | 1.0222 | 1.0773 | 1.201 | 1.426 | 1.773 | 2.00 |
| 20 | 1.0000 | 1.0000 | 1.0000 | 1.0001 | 1.0028 | 1.0234 | 1.113 | 1.367 | 1.898 | 2.29 |
| 50 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0007 | 1.018 | 1.182 | 1.920 | 2.70 |

**Table 3**

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

| Bucket size, $b$ | Load factor, $\alpha$ 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 95% |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.0500 | 1.1000 | 1.1500 | 1.2000 | 1.2500 | 1.3000 | 1.350 | 1.400 | 1.450 | 1.48 |
| 2 | 1.0063 | 1.0242 | 1.0520 | 1.0883 | 1.1321 | 1.1823 | 1.238 | 1.299 | 1.364 | 1.40 |
| 3 | 1.0010 | 1.0071 | 1.0215 | 1.0458 | 1.0806 | 1.1259 | 1.181 | 1.246 | 1.319 | 1.36 |
| 4 | 1.0002 | 1.0023 | 1.0097 | 1.0257 | 1.0527 | 1.0922 | 1.145 | 1.211 | 1.290 | 1.33 |
| 5 | 1.0000 | 1.0008 | 1.0046 | 1.0151 | 1.0358 | 1.0699 | 1.119 | 1.186 | 1.268 | 1.32 |
| 10 | 1.0000 | 1.0000 | 1.0002 | 1.0015 | 1.0070 | 1.0226 | 1.056 | 1.115 | 1.206 | 1.27 |
| 20 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0005 | 1.0038 | 1.018 | 1.059 | 1.150 | 1.22 |
| 50 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.001 | 1.015 | 1.083 | 1.16 |

B) *Chaining with coalescing lists.* Instead of providing a separate overflow area, we can adapt Algorithm C to external files. A doubly linked list of available space can be maintained for each cylinder, linking together each bucket that is not yet full. Under this scheme, every bucket contains a count of how many record positions are empty, and the bucket is removed from the doubly linked list only when its count becomes zero. A "roving pointer" can be used to distribute overflows (see exercise 2.5–6), so that different chains tend to use different overflow buckets. This method has not yet been analyzed, but it might prove to be quite useful.

C) *Open addressing.* We can also do without links, using an "open" method. Linear probing is probably better than random probing when we consider external searching, because the increment $c$ can often be chosen so that it minimizes latency delays between consecutive accesses. The approximate theoretical model of linear probing that was worked out above can be generalized to account for the influence of buckets, and it shows that linear probing is indeed satisfactory unless the table has gotten very full. For example, see Table 4; when the load factor is 90 percent and the bucket size is 50, the average number of accesses in a successful search is only 1.04. This is actually *better* than the 1.08 accesses required by the chaining method (A) with the same bucket size!

**Table 4**

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY LINEAR PROBING

| Bucket size, $b$ | Load factor, $\alpha$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 95% |
| 1 | 1.0556 | 1.1250 | 1.2143 | 1.3333 | 1.5000 | 1.7500 | 2.167 | 3.000 | 5.500 | 10.50 |
| 2 | 1.0062 | 1.0242 | 1.0553 | 1.1033 | 1.1767 | 1.2930 | 1.494 | 1.903 | 3.147 | 5.64 |
| 3 | 1.0009 | 1.0066 | 1.0201 | 1.0450 | 1.0872 | 1.1584 | 1.286 | 1.554 | 2.378 | 4.04 |
| 4 | 1.0001 | 1.0021 | 1.0085 | 1.0227 | 1.0497 | 1.0984 | 1.190 | 1.386 | 2.000 | 3.24 |
| 5 | 1.0000 | 1.0007 | 1.0039 | 1.0124 | 1.0307 | 1.0661 | 1.136 | 1.289 | 1.777 | 2.77 |
| 10 | 1.0000 | 1.0000 | 1.0001 | 1.0011 | 1.0047 | 1.0154 | 1.042 | 1.110 | 1.345 | 1.84 |
| 20 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0003 | 1.0020 | 1.010 | 1.036 | 1.144 | 1.39 |
| 50 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.001 | 1.005 | 1.040 | 1.13 |

The analysis of methods (A) and (C) involves some very interesting mathematics; we shall merely summarize the results here, since the details are worked out in exercises 49 and 55. The formulas involve two functions strongly related to the $Q$-functions of Theorem K, namely

$$R(\alpha,n)=\frac{n}{n+1}+\frac{n^2\alpha}{(n+1)(n+2)}+\frac{n^3\alpha^2}{(n+1)(n+2)(n+3)}+\cdots, \tag{55}$$

and

$$\begin{aligned} t_n(\alpha) &= e^{-n\alpha}\left(\frac{(\alpha n)^n}{(n+1)!}+2\,\frac{(\alpha n)^{n+1}}{(n+2)!}+3\,\frac{(\alpha n)^{n+2}}{(n+3)!}+\cdots\right)\\ &= \frac{e^{-n\alpha}n^n\alpha^n}{n!}\big(1-(1-\alpha)R(\alpha,n)\big). \end{aligned} \tag{56}$$

In terms of these functions, the average number of accesses made by the chaining method (A) in an unsuccessful search is

$$C'_N = 1 + \alpha b t_b(\alpha) + O\Big(\frac{1}{M}\Big) \tag{57}$$

as $M$, $N \to \infty$, and the corresponding number in a successful search is

$$C_N = 1 + \frac{e^{-b\alpha} b^b \alpha^b}{2b!}\big(2 + (\alpha-1)b + \big(\alpha^2 + (\alpha-1)^2(b-1)\big)R(\alpha, b)\big) + O\Big(\frac{1}{M}\Big). \tag{58}$$

The limiting values of these formulas are the quantities shown in Tables 2 and 3.

Since chaining method (A) requires a separate overflow area, we need to estimate how many overflows will occur. The average number of overflows will be $M(C'_N - 1) = N t_b(\alpha)$, since $C'_N - 1$ is the average number of overflows in any given list. Therefore Table 2 can be used to deduce the amount of overflow space required. For fixed $\alpha$, the standard deviation of the total number of overflows will be roughly proportional to $\sqrt{M}$ as $M \to \infty$.

Asymptotic values for $C'_N$ and $C_N$ appear in exercise 53, but the approximations aren't very good when $b$ is small or $\alpha$ is large; fortunately the series for $R(\alpha, n)$ converges rather rapidly even when $\alpha$ is large, so the formulas can be evaluated to any desired precision without much difficulty. The maximum values occur for $\alpha = 1$, when

$$\max C'_N = 1 + \frac{e^{-b} b^{b+1}}{b!} = \sqrt{\frac{b}{2\pi}} + 1 + O(b^{-1/2}), \tag{59}$$

$$\max C_N = 1 + \frac{e^{-b} b^b}{2b!}\big(R(b) + 1\big) = \frac{5}{4} + \sqrt{\frac{2}{9\pi b}} + O(b^{-1}), \tag{60}$$

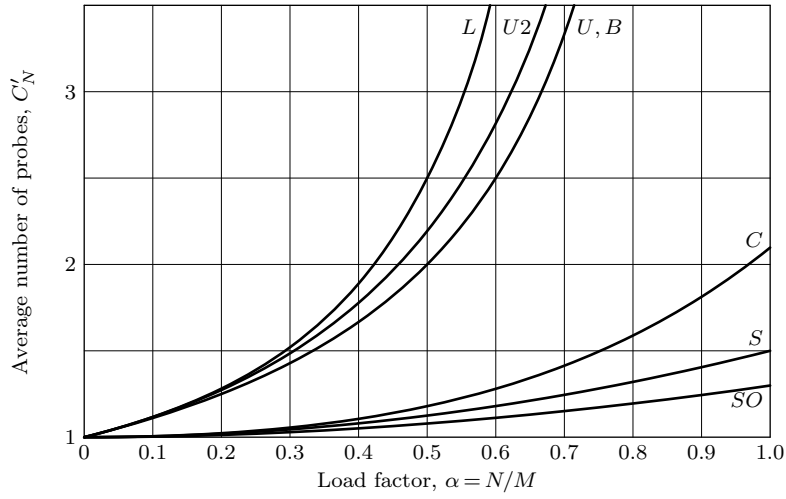
as $b \to \infty$, by Stirling's approximation and the analysis of the function $R(n) = R(1, n) - 1$ in Section 1.2.11.3.

The average number of accesses in a successful external search with *linear* probing has the remarkably simple expression

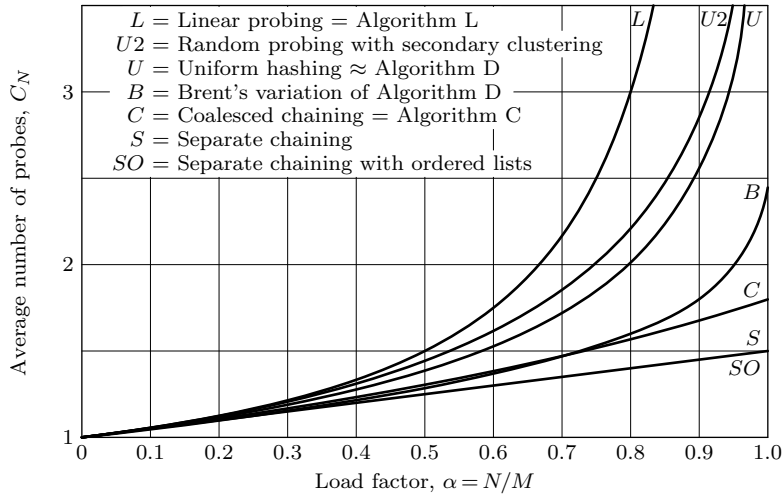
$$C_N \approx 1 + t_b(\alpha) + t_{2b}(\alpha) + t_{3b}(\alpha) + \cdots, \tag{61}$$

which can be understood as follows: The average total number of accesses to look up all $N$ keys is $NC_N$, and this is $N + T_1 + T_2 + \cdots$, where $T_k$ is the average number of keys that require more than $k$ accesses. Theorem P says that we can enter the keys in any order without affecting $C_N$, and it follows that $T_k$ is the average number of overflow records that would occur in the chaining method if we had $M/k$ buckets of size $kb$, namely $N t_{kb}(\alpha)$ by what we said above. Further justification of Eq. (61) appears in exercise 55.

An excellent early discussion of practical considerations involved in the design of external hash tables was given by Charles A. Olson, *Proc. ACM Nat. Conf.* **24** (1969), 539–549. He included several worked examples and pointed out that the number of overflow records will increase substantially if the file is subject to frequent insertion/deletion activity without relocating records. He also presented an analysis of this situation that was obtained jointly with J. A. de Peyster.




(a) Unsuccessful search




(b) Successful search

**Fig. 44.** Comparison of collision resolution methods: limiting values of the average number of probes as $M \to \infty$.

**Comparison of the methods.** We have now studied a large number of techniques for searching; how can we select the right one for a given application? It is difficult to summarize in a few words all the relevant details of the trade-offs involved in the choice of a search method, but the following things seem to be of primary importance with respect to the speed of searching and the requisite storage space.

Figure 44 summarizes the analyses of this section, showing that the various methods for collision resolution lead to different numbers of probes. But probe

counting does not tell the whole story, since the time per probe varies in different methods, and the latter variation has a noticeable effect on the running time (as we have seen in Fig. 42). Linear probing accesses the table more frequently than the other methods shown in Fig. 44, but it has the advantage of simplicity. Furthermore, even linear probing isn't terribly bad: When the table is 90 percent full, Algorithm L requires fewer than 5.5 probes, on the average, to locate a random item in the table. (However, a 90-percent-full table does require about 50.5 probes for every *new* item inserted by Algorithm L.)

Figure 44 shows that the chaining methods are quite economical with respect to the number of probes, but the extra memory space needed for link fields sometimes makes open addressing more attractive for small records. For example, if we have to choose between a chained hash table of capacity 500 and an open hash table of capacity 1000, the latter is clearly preferable, since it allows efficient searching when 500 records are present and it is capable of absorbing twice as much data. On the other hand, sometimes the record size and format will allow space for link fields at virtually no extra cost. (See exercise 65.)

How do hash methods compare with the other search strategies we have studied in this chapter? From the standpoint of speed we can argue that they are better, when the number of records is large, because the average search time for a hash method stays bounded as $N \to \infty$ if we stipulate that the table never gets too full. For example, Program L will take only about 55 units of time for a successful search when the table is 90 percent full; this beats the fastest `MIX` binary search routine we have seen (exercise 6.2.1–24) when $N$ is greater than 600 or so, at the cost of only 11 percent in storage space. Moreover the binary search is suitable only for fixed tables, while a hash table allows efficient insertions.

We can also compare Program L to the tree-oriented search methods that allow dynamic insertions. Program L with a 90-percent-full table is faster than Program 6.2.2T when $N$ is greater than about 90, and faster than Program 6.3D (exercise 6.3–9) when $N$ is greater than about 75.

Only one search method in this chapter is efficient for successful searching with virtually no storage overhead, namely Brent's variation of Algorithm D. His method allows us to put $N$ records into a table of size $M = N + 1$, and to find any record in about 2.5 probes on the average. No extra space for link fields or tag bits is needed; however, an unsuccessful search will be very slow, requiring about $N/2$ probes.

Thus hashing has several advantages. On the other hand, there are three important respects in which hash table searching is inferior to other methods:

a) After an unsuccessful search in a hash table, we know only that the desired key is not present. Search methods based on comparisons always yield more information; they allow us to find the largest key $\le K$ and/or the smallest key $\ge K$. This is important in many applications; for example, it allows us to interpolate function values from a stored table. We can also use comparison-based algorithms to locate all keys that lie *between* two given values $K$ and $K'$. Furthermore the tree search algorithms of Section 6.2 make it easy to traverse the contents of a table in ascending order, without sorting it separately.

b) The storage allocation for hash tables is often somewhat difficult; we have to dedicate a certain area of the memory for use as the hash table, and it may not be obvious how much space should be allotted. If we provide too much memory, we may be wasting storage at the expense of other lists or other computer users; but if we don't provide enough room, the table will overflow. By contrast, the tree search and insertion algorithms deal with trees that grow no larger than necessary. In a virtual memory environment we can keep memory accesses localized if we use tree search or digital tree search, instead of creating a large hash table that requires the operating system to access a new page nearly every time we hash a key.

c) Finally, we need a great deal of faith in probability theory when we use hashing methods, since they are efficient only on the average, while their worst case is terrible! As in the case of random number generators, we can never be completely sure that a hash function will perform properly when it is applied to a new set of data. Therefore hash tables are inappropriate for certain real-time applications such as air traffic control, where people's lives are at stake; the balanced tree algorithms of Sections 6.2.3 and 6.2.4 are much safer, since they provide guaranteed upper bounds on the search time.

**History.** The idea of hashing appears to have been originated by H. P. Luhn, who wrote an internal IBM memorandum in January 1953 that suggested the use of chaining; in fact, his suggestion was one of the first applications of linked linear lists. He pointed out the desirability of using buckets that contain more than one element, for external searching. Shortly afterwards, A. D. Lin carried Luhn's analysis further, and suggested a technique for handling overflows that used "degenerative addresses"; for example, the overflows from primary bucket 2748 were put in secondary bucket 274; overflows from that bucket went to tertiary bucket 27, and so on, assuming the presence of 10000 primary buckets, 1000 secondary buckets, 100 tertiary buckets, etc. The hash functions originally suggested by Luhn were digital in nature; for example, he combined adjacent pairs of key digits by adding them mod 10, so that 31415926 would be compressed to 4548.

At about the same time the idea of hashing occurred independently to another group of IBMers: Gene M. Amdahl, Elaine M. Boehm, N. Rochester, and Arthur L. Samuel, who were building an assembly program for the IBM 701. In order to handle the collision problem, Amdahl originated the idea of open addressing with linear probing. [See also Derr and Luke, *JACM* **3** (1956), 303.]

Hash coding was first described in the open literature by Arnold I. Dumey, *Computers and Automation* **5**, 12 (December 1956), 6–9. He was the first to mention the idea of dividing by a prime number and using the remainder as the hash address. Dumey's interesting article mentions chaining but not open addressing. A. P. Ershov of Russia independently discovered linear open addressing in 1957 [*Doklady Akad. Nauk SSSR* **118** (1958), 427–430]; he published empirical results about the number of probes, conjecturing correctly that the average number of probes per successful search is $< 2$ when $N/M < 2/3$.

A classic article by W. W. Peterson, *IBM J. Research & Development* **1** (1957), 130–146, was the first major paper dealing with the problem of searching in large files. Peterson defined open addressing in general, analyzed the performance of uniform probing, and gave numerous empirical statistics about the behavior of linear open addressing with various bucket sizes, noting the degradation in performance that occurred when items were deleted. Another comprehensive survey of the subject was published six years later by Werner Buchholz [*IBM Systems J.* **2** (1963), 86–111], who gave an especially good discussion of hash functions. Correct analyses of Algorithm L were first published by A. G. Konheim and B. Weiss, *SIAM J. Appl. Math.* **14** (1966), 1266–1274; V. Podderjugin, *Wissenschaftliche Zeitschrift der Technischen Universität Dresden* **17** (1968), 1087–1089.

Up to this time linear probing was the only type of open addressing scheme that had appeared in the literature, but another scheme based on repeated random probing by independent hash functions had independently been developed by several people (see exercise 48). During the next few years hashing became very widely used, but hardly anything more was published about it. Then Robert Morris wrote a very influential survey of the subject [*CACM* **11** (1968), 38–44], in which he introduced the idea of random probing with secondary clustering. Morris's paper touched off a flurry of activity that culminated in Algorithm D and its refinements.

It is interesting to note that the word "hashing" apparently never appeared in print, with its present meaning, until the late 1960s, although it had already become common jargon in several parts of the world by that time. The first published appearance of the word seems to have been in H. Hellerman's book *Digital Computer System Principles* (New York: McGraw–Hill, 1967), 152; the only previous occurrence among approximately 60 relevant documents studied by the author as this section was being written was in an unpublished memorandum written by W. W. Peterson in 1961. Somehow the verb "to hash" magically became standard terminology for key transformation during the mid-1960s, yet nobody was rash enough to use such an undignified word in print until 1967!

**Later developments.** Many advances in the theory and practice of hashing have been made since the author first prepared this chapter in 1972, although the basic ideas discussed above still remain useful for ordinary applications. For example, the book *Design and Analysis of Coalesced Hashing* by J. S. Vitter and W.-C. Chen (New York: Oxford Univ. Press, 1987) discusses and analyzes several instructive variants of Algorithm C.

From a practical standpoint, the most important hash technique invented in the late 1970s is probably the method that Witold Litwin called *linear hashing* [*Proc. 6th International Conf. on Very Large Databases* (1980), 212–223]. Linear hashing — which incidentally has nothing to do with the classical technique of linear probing — allows the number of hash addresses to grow and/or contract gracefully as items are inserted and/or deleted. An excellent discussion of linear hashing, including comparisons with other methods for internal searching, has

been given by Per-Åke Larson in *CACM* **31** (1988), 446–457; see also W. G. Griswold and G. M. Townsend, *Software Practice & Exp.* **23** (1993), 351–367, for improvements when many large and/or small tables are present simultaneously. Linear hashing can also be used for huge databases that are distributed between many different sites on a network [see Litwin, Neimat, and Schneider, *ACM Trans. Database Syst.* **21** (1996), 480–525]. An alternative scheme called *extendible hashing*, which has the property that at most two references to external pages are needed to retrieve any record, was proposed at about the same time by R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong [*ACM Trans. Database Syst.* **4** (1979), 315–344]; related ideas had been explored by G. D. Knott, *Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control* (1971), 187–206. Both linear hashing and extendible hashing are preferable to the $B$-trees of Section 6.2.4, when the order of keys is unimportant.

In the theoretical realm, more complicated methods have been devised by which it is possible to guarantee $O(1)$ maximum time per access, with $O(1)$ average amortized time per insertion and deletion, regardless of the keys being examined; moreover, the total storage used at any time is bounded by a constant times the number of items currently present, plus another additive constant. This result, which builds on ideas of Fredman, Komlós, and Szemerédi [*JACM* **31** (1984), 538–544], is due to Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, and Tarjan [*SICOMP* **23** (1994), 738–761].

## EXERCISES

**1.** [*20*] When the instruction `9H` in Table 1 is reached, how small and how large can the contents of rI1 possibly be, assuming that bytes 1, 2, 3 of $K$ each contain alphabetic character codes less than 30?

**2.** [*20*] Find a reasonably common English word not in Table 1 that could be added to that table without changing the program.

**3.** [*23*] Explain why no program beginning with the five instructions

```
LD1  K(1:1)   or  LD1N  K(1:1)
LD2  K(2:2)   or  LD2N  K(2:2)
INC1 a,2
LD2  K(3:3)
J2Z  9F
```

could be used in place of the more complicated program in Table 1, for any constant $a$, since unique addresses would not be produced for the given keys.

**4.** [*M30*] How many people should be invited to a party in order to make it likely that there are *three* with the same birthday?

**5.** [*15*] Mr. B. C. Dull was writing a FORTRAN compiler using a decimal `MIX` computer, and he needed a symbol table to keep track of the names of variables in the FORTRAN program being compiled. These names were restricted to be at most ten characters in length. He decided to use a hash table with $M = 100$, and to use the fast hash function $h(K) =$ leftmost byte of $K$. Was this a good idea?

**6.** [*15*] Would it be wise to change the first two instructions of (3) to `LDA K`; `ENTX 0`?

**7.** [*HM30*] (*Polynomial hashing.*) The purpose of this exercise is to consider the construction of polynomials $P(x)$ such as (10), which convert $n$-bit keys into $m$-bit addresses, in such a way that distinct keys differing in $t$ or fewer bits will hash to different addresses. Given $n$ and $t \le n$, and given an integer $k$ such that $n$ divides $2^k - 1$, we shall construct a polynomial whose degree $m$ is a function of $n$, $t$, and $k$. (Usually $n$ is increased, if necessary, so that $k$ can be chosen to be reasonably small.)

Let $S$ be the smallest set of integers such that $\{1, 2, \ldots, t\} \subseteq S$ and $(2j) \bmod n \in S$ for all $j \in S$. For example, when $n = 15$, $k = 4$, and $t = 6$, we have $S = \{1, 2, 3, 4, 5, 6, 8, 10, 12, 9\}$. We now define the polynomial $P(x) = \prod_{j \in S}(x - \alpha^j)$, where $\alpha$ is an element of order $n$ in the finite field GF$(2^k)$, and where the coefficients of $P(x)$ are computed in this field. The degree $m$ of $P(x)$ is the number of elements of $S$. Since $\alpha^{2j}$ is a root of $P(x)$ whenever $\alpha^j$ is a root, it follows that the coefficients $p_i$ of $P(x)$ satisfy $p_i^2 = p_i$, so they are 0 or 1.

Prove that if $R(x) = r_{n-1}x^{n-1} + \cdots + r_1 x + r_0$ is any nonzero polynomial modulo 2, with at most $t$ nonzero coefficients, then $R(x)$ is not a multiple of $P(x)$ modulo 2. [It follows that the corresponding hash function behaves as advertised.]

**8.** [*M34*] (*The three-distance theorem.*) Let $\theta$ be an irrational number between 0 and 1, whose regular continued fraction representation in the notation of Section 4.5.3 is $\theta = //a_1, a_2, a_3, \ldots //$. Let $q_0 = 0$, $p_0 = 1$, $q_1 = 1$, $p_1 = 0$, and $q_{k+1} = a_k q_k + q_{k-1}$, $p_{k+1} = a_k p_k + p_{k-1}$ for $k \ge 1$. Let $\{x\}$ denote $x \bmod 1 = x - \lfloor x \rfloor$, and let $\{x\}^+$ denote $x - \lceil x \rceil + 1$. As the points $\{\theta\}, \{2\theta\}, \{3\theta\}, \ldots$ are successively inserted into the interval $[0\,..\,1]$, let the line segments be numbered as they appear in such a way that the first segment of a given length is number 0, the next is number 1, etc. Prove that the following statements are all true: Interval number $s$ of length $\{t\theta\}$, where $t = rq_k + q_{k-1}$ and $0 \le r < a_k$ and $k$ is even and $0 \le s < q_k$, has left endpoint $\{s\theta\}$ and right endpoint $\{(s+t)\theta\}^+$. Interval number $s$ of length $1 - \{t\theta\}$, where $t = rq_k + q_{k-1}$ and $0 \le r < a_k$ and $k$ is odd and $0 \le s < q_k$, has left endpoint $\{(s+t)\theta\}$ and right endpoint $\{s\theta\}^+$. Every positive integer $n$ can be uniquely represented as $n = rq_k + q_{k-1} + s$ for some $k \ge 1$, $1 \le r \le a_k$, and $0 \le s < q_k$. In terms of this representation, just before the point $\{n\theta\}$ is inserted the $n$ intervals present are

the first $s$ intervals (numbered $0, \ldots, s-1$) of length $\{(-1)^k(rq_k + q_{k-1})\theta\}$;
the first $n - q_k$ intervals (numbered $0, \ldots, n - q_k - 1$) of length $\{(-1)^{k+1}q_k\theta\}$;
the last $q_k - s$ intervals (numbered $s, \ldots, q_k - 1$) of length $\{(-1)^k((r-1)q_k + q_{k-1})\theta\}^+$.

The operation of inserting $\{n\theta\}$ removes interval number $s$ of the third type and converts it into interval number $s$ of the first type, number $n - q_k$ of the second type.

**9.** [*M30*] When we successively insert the points $\{\theta\}$, $\{2\theta\}$, ... into the interval $[0\,..\,1]$, Theorem S asserts that each new point always breaks up one of the largest remaining intervals. If the interval $[a\,..\,c]$ is thereby broken into two parts $[a\,..\,b]$, $[b\,..\,c]$, we may call it a *bad break* if one of these parts is more than twice as long as the other, namely if $b - a > 2(c - b)$ or $c - b > 2(b - a)$.

Prove that bad breaks will occur for some $\{n\theta\}$ unless $\theta \bmod 1 = \phi^{-1}$ or $\phi^{-2}$; and the latter values of $\theta$ *never* produce bad breaks.

**10.** [*M38*] (R. L. Graham.) If $\theta, \alpha_1, \ldots, \alpha_d$ are real numbers with $\alpha_1 = 0$, and if $n_1, \ldots, n_d$ are positive integers, and if the points $\{n\theta + \alpha_j\}$ are inserted into the interval $[0\,..\,1]$ for $0 \le n < n_j$ and $1 \le j \le d$, prove that the resulting $n_1 + \cdots + n_d$ (possibly empty) intervals have at most $3d$ different lengths.

**11.** [*16*] Successful searches are often more frequent than unsuccessful ones. Would it therefore be a good idea to interchange lines 12–13 of Program C with lines 10–11?

▸ **12.** [*21*] Show that Program C can be rewritten so that there is only one conditional jump instruction in the inner loop. Compare the running time of the modified program with the original.

▸ **13.** [*24*] (*Abbreviated keys.*) Let $h(K)$ be a hash function, and let $q(K)$ be a function of $K$ such that $K$ can be determined once $h(K)$ and $q(K)$ are given. For example, in division hashing we may let $h(K) = K \bmod M$ and $q(K) = \lfloor K/M \rfloor$; in multiplicative hashing we may let $h(K)$ be the leading bits of $(AK/w) \bmod 1$, and $q(K)$ can be the other bits.

Show that when chaining is used without overlapping lists, we need only store $q(K)$ instead of $K$ in each record. (This almost saves the space needed for the link fields.) Modify Algorithm C so that it allows such abbreviated keys by avoiding overlapping lists, yet uses no auxiliary storage locations for overflow records.

**14.** [*24*] (E. W. Elcock.) Show that it is possible to let a large hash table *share memory* with any number of other linked lists. Let every word of the list area have a 2-bit `TAG` field and two link fields called `LINK` and `AUX`, with the following interpretation:

`TAG(P)` $= 0$ indicates a word in the list of available space; `LINK(P)` points to the next entry in this list, and `AUX(P)` is unused.

`TAG(P)` $= 1$ indicates a word in use where `P` is not the hash address of any key in the hash table; the other fields of the word in location `P` may have any desired format.

`TAG(P)` $= 2$ indicates that `P` is the hash address of at least one key; `AUX(P)` points to a linked list specifying all such keys, and `LINK(P)` points to another word in the list memory. Whenever a word with `TAG(P)` $= 2$ is accessed during the processing of any list, we set `P` $\leftarrow$ `LINK(P)` repeatedly until reaching a word with `TAG(P)` $\leq 1$. (For efficiency we might also then change prior links so that it will not be necessary to skip over the same entries again and again.)

Define suitable algorithms for inserting and retrieving keys in such a hash table.

**15.** [*16*] Why is it a good idea for Algorithm L and Algorithm D to signal overflow when $N = M - 1$ instead of when $N = M$?

**16.** [*10*] Program L says that $K$ should not be zero. But doesn't it actually work even when $K$ is zero?

**17.** [*15*] Why not simply define $h_2(K) = h_1(K)$ in (25), when $h_1(K) \neq 0$?

▸ **18.** [*21*] Is (31) better or worse than (30), as a substitute for lines 10–13 of Program D? Give your answer on the basis of the average values of $A$, $S1$, and $C$.

**19.** [*40*] Empirically test the effect of restricting the range of $h_2(K)$ in Algorithm D, so that (a) $1 \leq h_2(K) \leq r$ for $r = 1, 2, 3, \ldots, 10$; (b) $1 \leq h_2(K) \leq \rho M$ for $\rho = \frac{1}{10}, \frac{2}{10}, \ldots, \frac{9}{10}$.

**20.** [*M25*] (R. Krutar.) Change Algorithm D as follows, avoiding the hash function $h_2(K)$: In step D3, set $c \leftarrow 0$; and at the beginning of step D4, set $c \leftarrow c + 1$. Prove that if $M = 2^m$, the corresponding probe sequence $h_1(K)$, $\big(h_1(K) - 1\big) \bmod M$, $\ldots$, $\big(h_1(K) - \binom{M}{2}\big) \bmod M$ will be a permutation of $\{0, 1, \ldots, M-1\}$. When this "quadratic probing" method is programmed for `MIX`, how does it compare with the three programs considered in Fig. 42, assuming that the algorithm behaves like random probing with secondary clustering?

▸ **21.** [*20*] Suppose that we wish to delete a record from a table constructed by Algorithm D, marking it "deleted" as suggested in the text. Should we also decrease the variable $N$ that is used to govern Algorithm D?

**22.** [*27*] Prove that Algorithm R leaves the table exactly as it would have been if `KEY[`$i$`]` had never been inserted in the first place.

▸ **23.** [*33*] Design an algorithm analogous to Algorithm R, for deleting entries from a chained hash table that has been constructed by Algorithm C.

**24.** [*M20*] Suppose that the set of all possible keys that can occur has $MP$ elements, where exactly $P$ keys hash to any given address. (In practical cases, $P$ is very large; for example, if the keys are arbitrary 10-digit numbers and if $M = 10^3$, we have $P = 10^7$.) Assume that $M \geq 7$ and $N = 7$. If seven distinct keys are selected at random from the set of all possible keys, what is the exact probability that the hash sequence 1 2 6 2 1 6 1 will be obtained (namely that $h(K_1) = 1$, $h(K_2) = 2$, ..., $h(K_7) = 1$), as a function of $M$ and $P$?

**25.** [*M19*] Explain why Eq. (39) is true.

**26.** [*M20*] How many hash sequences $a_1\, a_2 \ldots a_9$ yield the pattern of occupied cells (21), using linear probing?

**27.** [*M27*] Complete the proof of Theorem K. [*Hint:* Let

$$s(n, x, y) = \sum_k \binom{n}{k} (x + k)^{k+1} (y - k)^{n-k-1} (y - n);$$

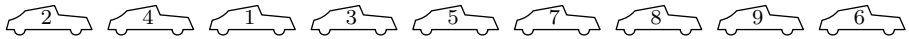
use Abel's binomial theorem, Eq. 1.2.6–(16), to prove that $s(n, x, y) = x(x + y)^n + ns(n-1, x+1, y-1)$.]

**28.** [*M30*] In the old days when computers were much slower than they are now, it was possible to watch the lights flashing and see how fast Algorithm L was running. When the table began to fill up, some entries would be processed very quickly, while others took a great deal of time.

This experience suggests that the standard deviation of the number of probes in an unsuccessful search is rather high, when linear probing is used. Find a formula that expresses the variance in terms of the $Q_r$ functions defined in Theorem K, and estimate the variance when $N = \alpha M$ as $M \to \infty$.

**29.** [*M21*] (*The parking problem.*) A certain one-way street has $m$ parking spaces in a row, numbered 1 through $m$. A man and his dozing wife drive by, and suddenly she wakes up and orders him to park immediately. He dutifully parks at the first available space; but if there are no places left that he can get to without backing up (that is, if his wife awoke when the car approached space $k$, but spaces $k$, $k + 1$, ..., $m$ are all full), he expresses his regrets and drives on.

Suppose, in fact, that this happens for $n$ different cars, where the $j$th wife wakes up just in time to park at space $a_j$. In how many of the sequences $a_1 \ldots a_n$ will all of the cars get safely parked, assuming that the street is initially empty and that nobody leaves after parking? For example, when $m = n = 9$ and $a_1 \ldots a_9 = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5$, the cars get parked as follows:




[*Hint:* Use the analysis of linear probing.]

**30.** [*M38*] When $n = m$ in the parking problem of exercise 29, show that all cars get parked if and only if there exists a permutation $p_1\, p_2 \ldots p_n$ of $\{1, 2, \ldots, n\}$ such that $a_j \le p_j$ for all $j$.

**31.** [*M40*] When $n = m$ in the parking problem of exercise 29, the number of solutions turns out to be $(n+1)^{n-1}$; and from exercise 2.3.4.4–22 we know that this is the same as the number of free trees on $n+1$ labeled vertices! Find an interesting connection between parking sequences and trees.

**32.** [*M27*] Prove that the system of equations (44) has a unique solution $(c_0, c_1, \ldots, c_{M-1})$, whenever $b_0, b_1, \ldots, b_{M-1}$ are nonnegative integers whose sum is less than $M$. Design an algorithm to find that solution.

▸ **33.** [*M23*] Explain why (51) is only an approximation to the true average number of probes made by Algorithm L. What was there about the derivation of (51) that wasn't rigorously exact?

▸ **34.** [*M23*] The purpose of this exercise is to investigate the average number of probes in a chained hash table when the lists are kept separate as in Fig. 38.

a) What is $P_{Nk}$, the probability that a given list has length $k$, when the $M^N$ hash sequences (35) are equally likely?

b) Find the generating function $P_n(z) = \sum_{k\ge 0} P_{Nk} z^k$.

c) Express the average number of probes for a successful search in terms of this generating function.

d) Deduce the average number of probes in an *unsuccessful* search, considering variants of the data structure in which the following conventions are used: (i) hashing is always to a list head (see Fig. 38); (ii) hashing is to a table position (see Fig. 40), but all keys except the first of a list go into a separate overflow area; (iii) hashing is to a table position and all entries appear in the hash table.

**35.** [*M24*] Continuing exercise 34, what is the average number of probes in an unsuccessful search when the individual lists are kept in order by their key values? Consider data structures (i), (ii), and (iii).

**36.** [*M23*] Continuing exercise 34(d), find the *variance* of the number of probes when the search is unsuccessful, using data structures (i) and (ii).

▸ **37.** [*M29*] Equation (19) gives the average number of probes in separate chaining when the search is successful; what is the *variance* of that number of probes?

**38.** [*M32*] (*Tree hashing.*) A clever programmer might try to use binary search trees instead of linear lists in the chaining method, thereby combining Algorithm 6.2.2T with hashing. Analyze the average number of probes that would be required by this compound algorithm, for both successful and unsuccessful searches. [*Hint:* See Eq. 5.2.1–(15).]

**39.** [*M28*] Let $c_N(k)$ be the total number of lists of length $k$ formed when Algorithm C is applied to all $M^N$ hash sequences (35). Find a recurrence relation on the numbers $c_N(k)$ that makes it possible to determine a simple formula for the sum

$$S_N = \sum_k \binom{k}{2} c_N(k).$$

How is $S_N$ related to the number of probes in an unsuccessful search by Algorithm C?

**40.** [*M33*] Equation (15) gives the average number of probes used by Algorithm C in an unsuccessful search; what is the *variance* of that number of probes?

**41.** [*M40*] Analyze $T_N$, the average number of times the index $R$ is decreased by 1 when the $(N+1)$st item is being inserted by Algorithm C.

▸ **42.** [*M20*] Derive (17), the probability that Algorithm C succeeds immediately.

**43.** [*HM44*] Analyze a modification of Algorithm C that uses a table of size $M' \geq M$. Only the first $M$ locations are used for hashing, so the first $M'-M$ empty nodes found in step C5 will be in the extra locations of the table. For fixed $M'$, what choice of $M$ in the range $1 \leq M \leq M'$ leads to the best performance?

**44.** [*M43*] (*Random probing with secondary clustering.*) The object of this exercise is to determine the expected number of probes in the open addressing scheme with probe sequence

$$h(K), \quad (h(K)+p_1) \bmod M, \quad (h(K)+p_2) \bmod M, \quad \ldots, \quad (h(K)+p_{M-1}) \bmod M,$$

where $p_1\, p_2 \ldots p_{M-1}$ is a randomly chosen permutation of $\{1, 2, \ldots, M-1\}$ that depends on $h(K)$. In other words, all keys with the same value of $h(K)$ follow the same probe sequence, and the $(M-1)!^M$ possible choices of $M$ probe sequences with this property are equally likely.

This situation can be modeled accurately by the following experimental procedure performed on an initially empty linear array of size $m$. Do the following operation $n$ times: "With probability $p$, occupy the leftmost empty position. Otherwise (that is, with probability $q = 1-p$), select any table position except the one at the extreme left, with each of these $m-1$ positions equally likely. If the selected position is empty, occupy it; otherwise select *any* empty position (including the leftmost) and occupy it, considering each of the empty positions equally likely."

For example, when $m = 5$ and $n = 3$, the array configuration after such an experiment will be (occupied, occupied, empty, occupied, empty) with probability

$$\tfrac{7}{192}qqq + \tfrac{1}{6}pqq + \tfrac{1}{6}qpq + \tfrac{11}{64}qqp + \tfrac{1}{3}ppq + \tfrac{1}{4}pqp + \tfrac{1}{4}qpp.$$

(This procedure corresponds to random probing with secondary clustering, when $p = 1/m$, since we can renumber the table entries so that a particular probe sequence is 0, 1, 2, ... and all the others are random.)

Find a formula for the average number of occupied positions at the left of the array (namely 2 in the example above). Also find the asymptotic value of this quantity when $p = 1/m$, $n = \alpha(m+1)$, and $m \to \infty$.

**45.** [*M43*] Solve the analog of exercise 44 with *tertiary clustering*, when the probe sequence begins $h_1(K)$, $\big((h_1(K)+h_2(K)\big) \bmod M$, and the succeeding probes are randomly chosen depending only on $h_1(K)$ and $h_2(K)$. (Thus the $(M-2)!^{M(M-1)}$ possible choices of $M(M-1)$ probe sequences with this property are considered to be equally likely.) Is this procedure asymptotically equivalent to uniform probing?

**46.** [*M42*] Determine $C'_N$ and $C_N$ for the open addressing method that uses the probe sequence

$$h(K),\ 0,\ 1,\ \ldots,\ h(K)-1,\ h(K)+1,\ \ldots,\ M-1.$$

**47.** [*M25*] Find the average number of probes needed by open addressing when the probe sequence is

$$h(K),\ h(K)-1,\ h(K)+1,\ h(K)-2,\ h(K)+2,\ \ldots.$$

This probe sequence was once suggested because all the distances between consecutive probes are distinct when $M$ is even. [*Hint:* Find the trick and this problem is easy.]

▸ **48.** [*M21*] Analyze the open addressing method that probes locations $h_1(K)$, $h_2(K)$, $h_3(K), \ldots$, given an infinite sequence of mutually independent random hash functions $\langle h_n(K)\rangle$. In this setup it is possible to probe the same location twice, for example if $h_1(K) = h_2(K)$, but such coincidences are rather unlikely until the table gets full.

**49.** [*HM24*] Generalizing exercise 34 to the case of $b$ records per bucket, determine the average number of probes (external memory accesses) $C_N$ and $C'_N$, for chaining with separate lists, assuming that a list containing $k$ elements requires $\max(1, k-b+1)$ probes in an unsuccessful search. Instead of using the exact probability $P_{Nk}$ as in exercise 34, use the *Poisson approximation*

$$\binom{N}{k}\Big(\frac{1}{M}\Big)^k\Big(1-\frac{1}{M}\Big)^{N-k} = \frac{N}{M}\frac{N-1}{M}\cdots\frac{N-k+1}{M}\Big(1-\frac{1}{M}\Big)^N\Big(1-\frac{1}{M}\Big)^{-k}\frac{1}{k!}$$
$$= \frac{e^{-\rho}\rho^k}{k!}(1+O(k^2/M)),$$

which is valid for $N = \rho M$ and $k \le \sqrt{M}$ as $M \to \infty$; derive formulas (57) and (58).

**50.** [*M20*] Show that $Q_1(M,N) = M-(M-N-1)Q_0(M,N)$, in the notation of (42). [*Hint:* Prove first that $Q_1(M,N) = (N+1)Q_0(M,N) - NQ_0(M,N-1)$.]

**51.** [*HM17*] Express the function $R(\alpha, n)$ defined in (55) in terms of the function $Q_0$ defined in (42).

**52.** [*HM20*] Prove that $Q_0(M,N) = \int_0^\infty e^{-t}(1+t/M)^N\,dt$.

**53.** [*HM20*] Prove that the function $R(\alpha, n)$ can be expressed in terms of the incomplete gamma function, and use the result of exercise 1.2.11.3–9 to find the asymptotic value of $R(\alpha, n)$ to $O(n^{-2})$ as $n \to \infty$, for fixed $\alpha < 1$.

**54.** [*HM28*] Show that when $b = 1$, Eq. (61) is equivalent to Eq. (23). *Hint:* We have

$$t_n(\alpha) = \frac{(-1)^{n-1}}{n!\,\alpha}\sum_{m>n}\frac{(-n\alpha)^m}{m(m-1)(m-n-1)!}.$$

**55.** [*HM43*] Generalize the Schay–Spruth model, discussed after Theorem P, to the case of $M$ buckets of size $b$. Prove that $C(z)$ is equal to $Q(z)/\big(B(z)-z^b\big)$, where $Q(z)$ is a polynomial of degree $b$ and $Q(1) = 0$. Show that the average number of probes is

$$1+\frac{M}{N}C'(1) = 1+\frac{1}{b}\left(\frac{1}{1-q_1}+\cdots+\frac{1}{1-q_{b-1}}-\frac{1}{2}\frac{B''(1)-b(b-1)}{B'(1)-b}\right),$$

where $q_1, \ldots, q_{b-1}$ are the roots of $Q(z)/(z-1)$. Replacing the binomial probability distribution $B(z)$ by the Poisson approximation $P(z) = e^{b\alpha(z-1)}$, where $\alpha = N/Mb$, and using Lagrange's inversion formula (see Eq. 2.3.4.4–(21) and exercise 4.7–8), reduce your answer to Eq. (61).

**56.** [*HM43*] Generalize Theorem K, obtaining an exact analysis of linear probing with buckets of size $b$. What is the asymptotic number of probes in a successful search when the table is full ($N = Mb$)?

**57.** [*M47*] Does the uniform assignment of probabilities to probe sequences give the minimum value of $C_N$, over all open addressing methods?

**58.** [*M21*] (S. C. Johnson.) Find ten permutations on $\{0,1,2,3,4\}$ that are equivalent to uniform probing in the sense of Theorem U.

**59.** [*M25*] Prove that if an assignment of probabilities to permutations is equivalent to uniform probing, in the sense of Theorem U, the number of permutations with nonzero probabilities exceeds $M^a$ for any fixed exponent $a$, when $M$ is sufficiently large.

**60.** [*M47*] Let us say that an open addressing scheme involves *single hashing* if it uses exactly $M$ probe sequences, one beginning with each possible value of $h(K)$, each of which occurs with probability $1/M$.

Are the best single-hashing schemes (in the sense of minimum $C_N$) asymptotically better than the random ones described by (29)? In particular, is $C_{\alpha M} \geq 1 + \frac{1}{2}\alpha + \frac{1}{2}\alpha^2 + O(\alpha^3)$ as $M \to \infty$?

**61.** [*M46*] Is the method analyzed in exercise 46 the worst possible single-hashing scheme, in the sense of exercise 60?

**62.** [*M49*] A single hashing scheme is called *cyclic* if the increments $p_1\, p_2 \ldots p_{M-1}$ in the notation of exercise 44 are fixed for all $K$. (Examples of such methods are linear probing and the sequences considered in exercises 20 and 47.) An *optimum* single hashing scheme is one for which $C_M$ is minimum, over all $(M-1)!^M$ single hashing schemes for a given $M$. When $M \leq 5$ the best single hashing schemes are cyclic. Is this true for all $M$?

**63.** [*M25*] If repeated random insertions and deletions are made in a hash table, how many independent insertions are needed on the average before all $M$ locations have become occupied at one time or another? (This is the mean time to failure of the deletion method that simply marks cells "deleted.")

**64.** [*M41*] Analyze the expected behavior of Algorithm R (deletion with linear probing). How many times will step R4 be performed, on the average?

▸ **65.** [*20*] (*Variable-length keys.*) Many applications of hash tables deal with keys that can be any number of characters long. In such cases we can't simply store the key in the table as in the programs of this section. What would be a good way to deal with variable-length keys in a hash table on the `MIX` computer?

▸ **66.** [*25*] (Ole Amble, 1973.) Is it possible to insert keys into an open hash table making use also of their numerical or alphabetic order, so that a search with Algorithm L or Algorithm D is known to be unsuccessful whenever a key *smaller* than the search argument is encountered?

**67.** [*M41*] If Algorithm L inserts $N$ keys with respective hash addresses $a_1\, a_2 \ldots a_N$, let $d_j$ be the displacement of the $j$th key from its home address $a_j$; then $C_N = 1 + (d_1 + d_2 + \cdots + d_N)/N$. Theorem P tells us that permutation of the $a$'s has no effect on the sum $d_1 + d_2 + \cdots + d_N$. However, such permutation might drastically change the sum $d_1^2 + d_2^2 + \cdots + d_N^2$. For example, the hash sequence $1\ 2\ \ldots\ N{-}1\ N{-}1$ makes $d_1\ d_2\ \ldots\ d_{N-1}\ d_N = 0\ 0\ \ldots\ 0\ N{-}1$ and $\sum d_j^2 = (N-1)^2$, while its reflection $N{-}1\ N{-}1\ \ldots\ 2\ 1$ leads to much more civilized displacements $0\ 1\ \ldots\ 1\ 1$ for which $\sum d_j^2 = N - 1$.

a) Which rearrangement of $a_1\, a_2 \ldots a_N$ minimizes $\sum d_j^2$?

b) Explain how to modify Algorithm L so that it maintains a least-variance set of displacements after every insertion.

c) Determine the average value of $\sum d_j^2$ with and without this modification.

**68.** [*M41*] What is the variance of the average number of probes in a successful search by Algorithm L? In particular, what is the average of $(d_1+d_2+\cdots+d_N)^2$ in the notation of exercise 67?

**69.** [*M25*] (Andrew Yao.) Prove that all cyclic single hashing schemes in the sense of exercise 62 satisfy the inequality $C'_{\alpha M} \geq \frac{1}{2}(1 + 1/(1-\alpha))$. [*Hint:* Show that an unsuccessful search takes exactly $k$ probes with probability $p_k \leq (M-N)/M$.]

**70.** [*HM43*] Prove that the expected number of probes that are needed to insert the $(\alpha M+1)$st item with double hashing is at most the expected number needed to insert the $(\alpha M + \sqrt{O(\log M)/M}\,)$th item with uniform probing.

**71.** [*40*] Experiment with the behavior of Algorithm C when it has been adapted to external searching as described in the text.

▸ **72.** [*M28*] (*Universal hashing.*) Imagine a gigantic matrix $H$ that has one column for every possible key $K$. The entries of $H$ are numbers between 0 and $M-1$; the rows of $H$ represent hash functions. We say that $H$ defines a *universal family of hash functions* if any two columns agree in at most $R/M$ rows, where $R$ is the total number of rows.

a) Prove that if $H$ is universal in this sense, and if we select a hash function $h$ by choosing a row of $H$ at random, then the expected size of the list containing any given key $K$ in the method of separate chaining (Fig. 38) will be $\leq 1+N/M$, after we have inserted any set of $N$ distinct keys $K_1$, $K_2$, ..., $K_N$.

b) Suppose each $h_j$ in (9) is a randomly chosen mapping from the set of all characters to the set $\{0, 1, \ldots, M-1\}$. Show that this corresponds to a universal family of hash functions.

c) Would the result of (b) still be true if $h_j(0) = 0$ for all $j$, but $h_j(x)$ is random for $x \neq 0$?

**73.** [*M26*] (Carter and Wegman.) Show that part (b) of the previous exercise holds even when the $h_j$ are not completely random functions, but they have either of the following special forms: (i) Let $x_j$ be the binary number $(b_{j(n-1)} \ldots b_{j1} b_{j0})_2$. Then $h_j(x_j) = (a_{j(n-1)} b_{j(n-1)} + \cdots + a_{j1} b_{j1} + a_{j0} b_{j0}) \bmod M$, where each $a_{jk}$ is chosen randomly modulo $M$. (ii) Let $M$ be prime and assume that $0 \leq x_j < M$. Then $h_j(x_j) = (a_j x_j + b_j) \bmod M$, where $a_j$ and $b_j$ are chosen randomly modulo $M$.

**74.** [*M29*] Let $H$ define a universal family of hash functions. Prove or disprove: Given any $N$ distinct columns, and any row chosen at random, the expected number of zeros in those columns is $O(1) + O(N/M)$. [Thus, every list in the method of separate chaining will have this expected size.]

**75.** [*M26*] Prove or disprove the following statements about the hash function $h$ of (9), when the $h_j$ are independent random functions:

a) The probability that $h(K) = m$ is $1/M$, for all $0 \leq m < M$.

b) If $K \neq K'$, the probability that $h(K) = m$ and $h(K') = m'$ is $1/M^2$, for all $0 \leq m, m' < M$.

c) If $K$, $K'$, and $K''$ are distinct, the probability that $h(K) = m$, $h(K') = m'$, and $h(K'') = m''$ is $1/M^3$, for all $0 \leq m, m', m'' < M$.

d) If $K$, $K'$, $K''$, and $K'''$ are distinct, the probability that $h(K) = m$, $h(K') = m'$, $h(K'') = m''$, and $h(K''') = m'''$ is $1/M^4$, for all $0 \leq m, m', m'', m''' < M$.

▸ **76.** [*M21*] Suggest a way to modify (9) for keys with variable length, preserving the properties of universal hashing.

**77.** [*M22*] Let $H$ define a universal family of hash functions from 32-bit keys to 16-bit keys. (Thus $H$ has $2^{32}$ columns, and $M = 2^{16}$, in the notation of exercise 72.) A 256-bit key can be regarded as the concatenation of eight 32-bit parts $x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8$; we

can map it into a 16-bit address with the hash function

$$h_4\big(h_3\big(h_2(h_1(x_1)h_1(x_2))h_2(h_1(x_3)h_1(x_4))\big)h_3\big(h_2(h_1(x_5)h_1(x_6))h_2(h_1(x_7)h_1(x_8))\big)\big),$$

where $h_1$, $h_2$, $h_3$, and $h_4$ are randomly and independently chosen rows of $H$. (Here, for example, $h_1(x_1)h_1(x_2)$ stands for the 32-bit number obtained by concatenating $h_1(x_1)$ with $h_1(x_2)$.) Prove that the probability is less than $2^{-14}$ that two distinct keys hash to the same address. [This scheme requires substantially fewer random choices than (9).]

▸ **78.** [*M26*] (P. Woelfel.) If $0 \le x < 2^n$, let $h_{a,b}(x) = \lfloor (ax + b)/2^k \rfloor \bmod 2^{n-k}$. Show that the set $\{h_{a,b} \mid 0 < a < 2^n,\ a \text{ odd, and } 0 \le b < 2^k\}$ is a universal family of hash functions from $n$-bit keys to $(n-k)$-bit keys. (These functions are particularly easy to implement on a binary computer.)

*She made a hash of the proper names, to be sure.*
— GRANT ALLEN, *The Tents of Shem* (1889)

*HASH, x. There is no definition*
*for this word —*
*nobody knows what hash is.*
— AMBROSE BIERCE, *The Devil's Dictionary* (1906)