EXHIBIT 5

PART 2 OF 6

called a *collision*, and several interesting approaches have been devised to handle the collision problem. In order to use a scatter table, a programmer must make two almost independent decisions: He must choose a hash function $h(K)$, and he 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.

It is theoretically impossible to define a hash function that creates 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 of these 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 either; for if the keys are alphabetic, two keys which differ from each other only by permutation of letters would then differ in numeric value by a multiple of 3. (This occurs because $10^n \bmod 3 = 4^n \bmod 3 = 1$.) In general, we want to avoid values of M which 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 modu
position of the key dig
a prime number such tl
has been found to be q
For example, on tl
$h(K)$ by the sequence

```
LDX
ENTA
DIV
```

The multiplicative
harder to describe bec
instead of with integer
usually 10^{10} or 2^{30} fo
if we imagine the radi
choose some integer co

In this case we usually
$h(K)$ consists of the lea
In MIX code, if we l
hash function is

```
LDA
MUL
ENTA
SLB
```

Now $h(K)$ appears in
shift instructions, this
on many machines mul
In a sense this met
could for example take
the reciprocal of a cons
that (5) is almost a "m
ence: We shall see that
good properties.
One of the nice fea
was lost in (5); we co
after (5) has finished.
algorithm can be used
that $K = (A'(AK$ mo
tents of register X just

K_1

aches have been devised to scatter table, a programmer must choose a hash function resolution. We shall now con-

let us assume throughout this M different values, with

(1)

in practice usually have a great hash function that breaks up ce the number of collisions. function that creates random ut in practice it is not difficult ta, by using simple arithmetic we can often do even better, ual data to construct a hash random keys would produce. keys on a decimal computer. $= 1000$, say, and to let $h(K)$ middle of the 20-digit product spread of values between 000 riments with actual data show, bad, provided that the keys but it turns out that there are d in Chapter 3 that the middle number generator. that two major types of hash d on division, and the other is

we simply use the remainder

(2)

much better than others. For even when K is even and odd al bias in many files. It would f the computer, since K mod M of K (independent of the other lly shouldn't be a multiple of 3 which differ from each other in numeric value by a multiple d $3 = 1$.) In general, we want k and a are small numbers and (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 $r^k \equiv \pm a$ (modulo M) for small k and a. This choice has been found to be quite satisfactory in virtually all 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. Note that (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 was lost 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 $K = (A'(AK \bmod w)) \bmod w$. In other words, if $f(K)$ denotes the contents of register X just before the SLB instruction in (5), then

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

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) and tending 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. 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.

Figure 37 illustrates this aspect of multiplicative hashing in a particularly interesting case. Suppose that A/w is approximately the golden ratio $\phi^{-1} = (\sqrt{5}-1)/2 = 0.6180339887$; then the behavior of successive values $h(K)$, $h(K+1)$, $h(K+2)$, ... can be studied by considering the behavior of the successive values $h(0)$, $h(1)$, $h(2)$, ... This suggests the following experiment: 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 $x - \lfloor x \rfloor = x \bmod 1$). 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

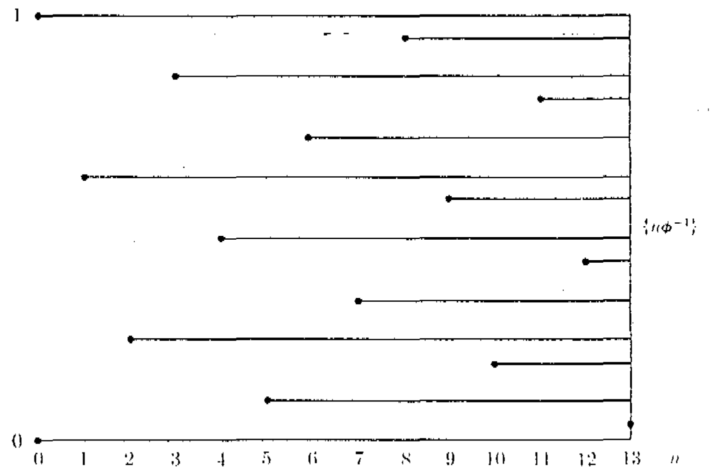


Fig. 37. Fibonacci hashing.

first conjectured by J. Oderfeld a
Math. **46** (1958), 187–189. Fibo
proof.]

This remarkable property of
of a very general result, origina
proved by Vera Turán Sós [*Acta*
Ann. Univ. Sci. Budapest. Eötvös

Theorem S. *Let θ be any irratio*
$\{n\theta\}$ are placed in the line segmen
most three different lengths. More
of the largest existing segments. ∎

Thus, the points $\{\theta\}$, $\{2\theta\}$, ...,
and 1. If θ is rational, the same th
to the segments of length 0 that
denominator of θ. A proof of Th
the underlying structure of the
that the segments of a given leng
out manner. Of course, some θ's a
that is near 0 or 1 will start out
ment. Exercise 9 shows that the
the "most uniformly distributed
0 and 1.

The above theory suggests *Fi*
A to be the nearest integer to ϕ^-
if MIX were a decimal computer w

$$A = \boxed{+}$$

This multiplier will spread out a
nicely. But notice what happen
fourth character position, as in t
if Theorem S were being used
$\theta = .6180339887 = A/w$. The r
the fact that this value of θ is no
the progression occurs in the se
A3␣␣␣, the effective θ is .9887, an
Therefore we might do bette

$$A = \boxed{+}$$

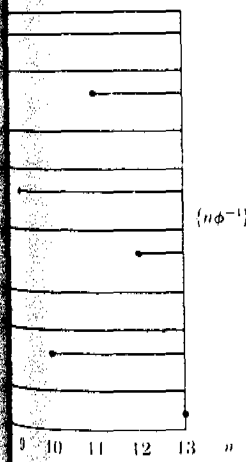
in place of (7); such a multipli
keys differing in *any* character p
another problem analogous to th
as XY and YX will tend to hash to

BTEX0000220

$\phi - 1$, so it isn't any good as a
scrambling function, namely a
the keys. Such a function
rch algorithms of Section 6.2.2,
oves the danger of degeneracy
A scrambling function is also
algorithm of Section 6.3, if the

method is that it makes good
Actual sets of keys often have
here $\{K, K+d, K+2d, \ldots,$
onsider alphabetic names like
The multiplicative hash
to an approximate arithmetic
stinct hash values, reducing the
s in a random situation. The

ative hashing in a particularly
nately the golden ratio $\phi^{-1} =$
or of successive values $h(K)$,
nsidering the behavior of the
gests the following experiment:
ely mark off the points $\{\phi^{-1}\}$,
fractional part of x (namely
points stay very well separated
nt falls into one of the largest
ratio! [This phenomenon was



hing.

first conjectured by J. Oderfeld and proved by S. Świerczkowski, *Fundamenta Math.* **46** (1958), 187–189. Fibonacci numbers play an important rôle in the proof.]

This remarkable 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 θ be any irrational number. When the points $\{\theta\}$, $\{2\theta\}$, $\ldots$, $\{n\theta\}$ 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 θ 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 θ. 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 θ'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 ϕ^{-1} and $\phi^{-2} = 1 - \phi^{-1}$ lead to the "most uniformly distributed" sequences, among all numbers θ between 0 and 1.

The above theory 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

$$A = \boxed{+}\boxed{61}\boxed{80}\boxed{33}\boxed{98}\boxed{87}\,. \qquad (7)$$

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 $\theta = \{100A/w\} = .8033988 7$ instead of $\theta = .6180339887 = A/w$. The resulting behavior is still all right, in spite of the fact that this value of θ is not quite as good as ϕ^{-1}. On the other hand, if the progression occurs in the second character position, as in `A1␣␣␣`, `A2␣␣␣`, `A3␣␣␣`, the effective θ is .9887, and this is probably too close to 1.

Therefore we might do better with a multiplier like

$$A = \boxed{+}\boxed{61}\boxed{61}\boxed{61}\boxed{61}\boxed{61} \qquad (8)$$

in place of (7); such a multiplier will separate out consecutive sequences of keys differing 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 θ are relevant, and small partial quotients correspond to good distribution properties. Therefore we find that the best values of θ 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, e.g.,

$$A = \boxed{+}\boxed{61}\boxed{25}\boxed{42}\boxed{33}\boxed{71}\,. \tag{9}$$

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 somewhat 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 these 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 above methods, 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, i.e., that they depend on all bits of both arguments, and exclusive-or is sometimes preferable because it avoids arithmetic overflow. Note that both of these operations are commutative, so that (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.

Many more methods for hashing have been suggested, but none of these has proved to be superior to the simple division and multiplication methods described above. For a survey of some other methods 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 A
nomial $P(x) = x^m + p_{m-1}x^{\prime\prime}$
$(k_{n-1} \ldots k_1k_0)_2$ can be regar
$k_1x + k_0$, and we compute th

$K(x) \bmod P(x$

using polynomial arithmetic r
is chosen properly, this hash
between nearly-equal keys. F

$P(x) = x^{10}$

it can be shown that $h(K_1)$ w
distinct keys that differ in fe
further information about th
hardware or microprogrammin

It has been found conven
when debugging a program, s
$h(K)$ can be substituted later.

Collision resolution by "chain
will probably be burdened w
most obvious way to solve th
each possible hash code. A l
there will also be M list heads
the key, we simply do a sequ
cise 6.1–2. The situation is
Program 5.2.1M.)

Figure 38 illustrates this
sequence of seven keys

$K =$ E

(i.e., the numbers 1 through 7

$h(K) + 1 =$

The first list has two elements
Chaining is quite fast, bec
together in one room, there wi
day, but the average number
In general, if there are N key
hashing decreases the amou
roughly a factor of M.

This method is a straig
discussed before, so we do not
scatter tables. It is often a g
key, so that insertions and m

underlying Theorem S; for short
quotients of the continued frac-
ll partial quotients correspond to
d that the best values of θ lie in

$\theta < \frac{2}{3}$, $\quad \frac{7}{10} < \theta < \frac{3}{4}$.

bytes lies in a good range and is
or their complements, e.g.,

$\boxed{33}\boxed{71}$. (9)

these ideas about multiplicative

quirements:

and property (b) is data-depen-
d simply extract a few bits from
but in practice we nearly always
n all bits of the key in order to

-word keys. Multiword or vari-
precision extensions of the above
ed things up by combining the
then doing a single multiplication
done by addition mod w, or by
of these operations have the
they depend on all bits of both
ferable because it avoids arith-
ations are commutative, so that
ess; G. D. Knott has suggested
just before adding or exclusive-

en suggested, but none of these
sion and multiplication methods
methods together with detailed
s, see the article by V. Y. Lum,
), 228–239.
en tried, perhaps the most inter-
theory; the idea is analogous to
polynomial modulo 2 instead of
4.6, this operation is analogous
clusive-or.) For this method, M

should be a power of 2, say $M = 2^m$, and we make use of an mth degree polynomial $P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_0$. An n-digit binary key $K = (k_{n-1} \ldots k_1k_0)_2$ can be regarded as the polynomial $K(x) = k_{n-1}x^{n-1} + \cdots + k_1x + k_0$, and we compute the remainder

$$K(x) \bmod P(x) = h_{m-1}x^{m-1} + \cdots + h_1x + h_0$$

using polynomial arithmetic modulo 2; then $h(K) = (h_{m-1} \ldots h_1h_0)_2$. 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

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

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 has been found 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 $h(K) + 1$. (Cf. 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

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

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

$$h(K) + 1 = 3, \quad 1, \quad 4, \quad 1, \quad 5, \quad 9, \quad 2. \qquad (12)$$

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

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 amount of work needed for sequential searching by roughly a factor of M.

This method is a straightforward combination of techniques we have discussed before, so we do not need to formulate a detailed algorithm for chained scatter tables. It is often a good idea to keep the individual lists in order by key, so that insertions and unsuccessful searches go faster. Thus if we choose

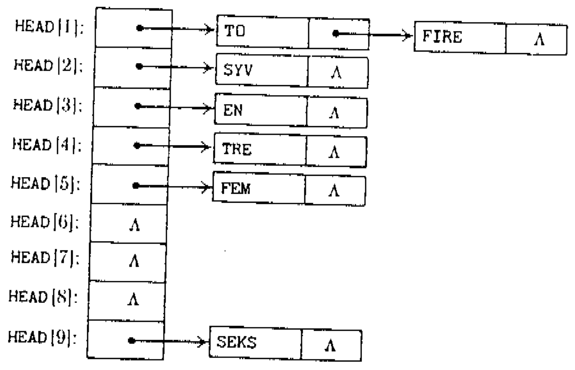


Fig. 38. Separate chaining.

to make the lists ascending, the TO and FIRE nodes of Fig. 38 would be interchanged, and all the Λ links would be replaced by pointers to a dummy record whose key is ∞. (*Cf.* Algorithm 6.1T.) Alternatively we can make use of the "self-organizing file" 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 would 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 scatter table search 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 $0 \le i \le M$, 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 $i \leftarrow h(K)$

C2. [Is there a list?] If T
occupied; we will look

C3. [Compare.] If $K =$ KE

C4. [Advance to next.] If

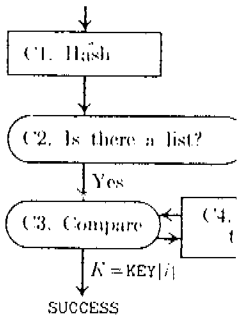
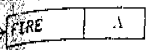


Fig. 39. Ch

C5. [Find empty node.] C
empty position in the
a value such that TAB
with overflow (there a
$i \leftarrow$ R.

C6. [Insert new key.] Mar
LINK[i] ← 0. ▮

This algorithm allows
moved after they have be
where SEKS appears in th
already been inserted into



ing.

...es of Fig. 38 would be inter-
...y pointers to a dummy record
...tively we can make use of the
...n 6.1; instead of keeping the
...according to the time of most

...M rather large. But when M
...ch of the space for the M list
...proach, when the records are
...e list heads, making room for
...N records and M + N links.
...all the data to find out which
...s inserting all the "overflow"
...impractical or impossible, and
...each record only once when it
...due to F. A. Williams [CACM
...olve the problem.

...tion). This algorithm searches
...K is not in the table and the

...i], for 0 ≤ i ≤ M, and they
...d. An occupied node contains
...y other fields.
...h(K). An auxiliary variable
...the table is empty, we have
...ways be true that TABLE[j] is
...vention, TABLE[0] will always

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 which starts here.)

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

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

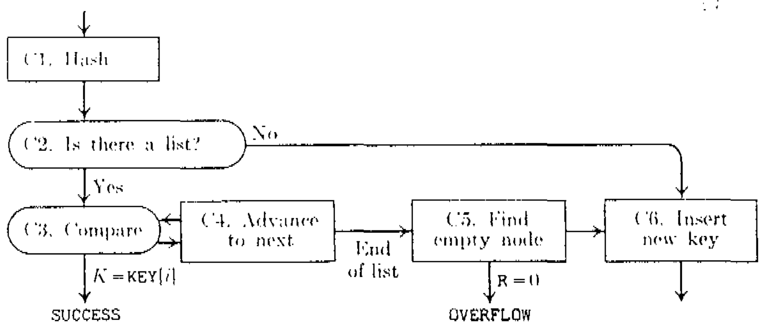


Fig. 39. Chained scatter table search and insertion.

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.

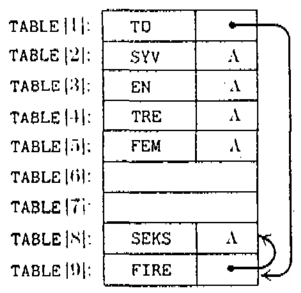


Fig. 40. Coalesced chaining.

BTEX0000225

In order to see how this algorithm 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.

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

empty	−	1	0	0	0	0
occupied	+	LINK		KEY		

(13)

The table size M is assumed to be prime; TABLE[i] is stored in location TABLE + i. rI1 ≡ i, rA ≡ K.

```
KEY    EQU   3:5
LINK   EQU   0:2
START  LDX   K                   1        C1. Hash.
       ENTA  0                   1
       DIV   =M=                 1
       STX   *+1(0:2)            1
       ENT1  *                   1        i ← h(K)
       INC1  1                   1              + 1.
       LDA   K                   1
       LD2   TABLE,1(LINK)       1        C2. Is there a list?
       J2N   6F                  1        To C6 if TABLE[i] empty.
       CMPA  TABLE,1(KEY)        A        C3. Compare.
       JE    SUCCESS             A        Exit if K = KEY[i].
       J2Z   5F              A − S1       To C5 if LINK[i] = 0.
4H     ENT1  0,2              C − 1       C4. Advance to next.
       CMPA  TABLE,1(KEY)     C − 1       C3. Compare.
       JE    SUCCESS          C − 1       Exit if K = KEY[i].
       LD2   TABLE,1(LINK) C − 1 − S2
       J2NZ  4B            C − 1 − S2     Advance if LINK[i] ≠ 0.
5H     LD2   R                A − S       C5. Find empty node.
       DEC2  1                  T         R ← R − 1.
       LDX   TABLE,2            T
       JXNN  *-2                T         Repeat until TABLE[R] empty.
       J2Z   OVERFLOW         A − S       Exit if no empty nodes left.
       ST2   TABLE,1(LINK)    A − S       LINK[i] ← R.
       ENT1  0,2              A − S       i ← R.
       ST2   R                A − S       Update R in memory.
6H     STZ   TABLE,1(LINK)    1 − S       C6. Insert new key.
       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 = 1 if the initial probe found an occupied node;

S = 1 if successful, 0

T = number of table

Here $S = S1 + S2$, where
time for the searching ph
and the insertion of a new
Suppose there are N k

$$\alpha =$$

Then the average value of
hash function is random;
in an unsuccessful search is

$$C'_N = 1 + \frac{1}{4}\Bigg(\Bigg(1 +$$

Thus when the table is half
successful search is about
completely full, the averag
final item will be only abou
small, as shown in exercise
even though the algorithm o
function is random. Of cou
bad or if we are extremely u

In a successful search,
probes during a successful s
$C + A$ over the first N unsu
that each key is equally likel

$$C_N = \frac{1}{N}\sum_{0\le k<N}\Big(C'_k + \frac{k}{M}\Big) =$$

$$\approx$$

as the average number of pr
table will require only abou
Similarly (see exercise 42), th

$$S1_N = 1 -$$

At first glance it may appe
sequentially for an empty po
probes made in step C5 as a t
of items in the table; so we ma
insertion! Exercise 41 proves
cessful search.

It would be possible to m
but then it would become ne

$S = 1$ if successful, 0 if unsuccessful;

$T =$ 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

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

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

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

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

$$C_N = \frac{1}{N}\sum_{0\le k<N}\left(C'_k + \frac{k}{M}\right) = 1 + \frac{1}{8}\frac{M}{N}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) + \frac{1}{4}\frac{N-1}{M}$$
$$\approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \tfrac{1}{4}\alpha \qquad (16)$$

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

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

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 αe^α 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 using circular lists, as suggested by Allen Newell in 1962, since the lists are short; but that would probably slow down the main search loop because step C4 would be more complicated. Exercise 34 shows that the average number of probes, when lists aren't coalesced, is reduced to

$$C'_N = \left(1 - \frac{1}{M}\right)^N + \frac{N}{M} \approx e^{-\alpha} + \alpha \qquad \text{(unsuccessful search)}; \qquad (18)$$

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

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 actually needed for links can actually be saved in the chaining method, if we avoid coalescing the lists. This leads to an interesting algorithm which is discussed in exercise 13.

Note that chaining can be used when $N > M$, so overflow is not a serious problem. If separate lists are used, formulas (18) and (19) are valid for $\alpha > 1$. When the lists coalesce as in 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 $(L/2M + \frac{1}{4})((1 + 2/M)^M - 1) + \frac{1}{2}$.

Collision resolution by "open addressing." Another way to resolve the problem of collisions is to do away with the 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 which are to be inspected whenever K is inserted or looked up. If we encounter an open 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

$$h(K), h(K) - 1, \ldots, 0, M - 1, M - 2, \ldots, h(K) + 1 \qquad (20)$$

as in the following algorithm.

Algorithm L (*Open scatter table search 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 tabl
are of two distinguishable
tains a key, called KEY[i],
used to keep track of how
to be part of the table, an
This algorithm makes
probing sequence (20) to
are discussed below.

L1. [Hash.] Set $i \leftarrow h(K)$.
L2. [Compare.] If KEY[i] =
if TABLE[i] is empty, g
L3. [Advance to next.] Se
to step L2.
L4. [Insert.] (The search
terminates with overf
when N = $M - 1$, n
N ← N + 1, mark TABL

Figure 41 shows wha
inserted by Algorithm L,
last three keys, FEM, SEKS.
tions $h(K)$.

F

Program L (*Open scatter t*
full-word keys; but a key
position in the table. (A
negative, letting empty p
to be prime, and TABLE[i]
speed in the inner loop, l
VACANCIES is assumed to c
In order to speed up
been removed from the lo
remain. The total runnin
21 − 4S)u, and the inse

wanted to insert SEKS into
would be necessary to move
discover which node points
viding two-way linkage by
in 1962, since the lists are
search loop because step C4
that the average number of

successful search); (18)

ccessful search). (19)

d (16) to warrant changing

ed that most of the space
the chaining method, if we
esting algorithm which is

so overflow is not a serious
d (19) are valid for $\alpha > 1$.
k extra items into an aux-
rage number of probes to
$(1+2/M)^M - 1) + \frac{1}{2}$.

way to resolve the problem
simply looking at various
key K or finding an empty
h every key K determines
tions which are to be in-
ncounter an open position
termined by K, we can
quence of probes will be
of methods was named
& *Development* 1 (1957),

linear probing, uses the

$\ldots, h(K)+1$ (20)

This algorithm searches
not in the table and the

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 KEY[i] $= K$, the algorithm terminates successfully. Otherwise if TABLE[i] is empty, go to L4.

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 $h(K)$.

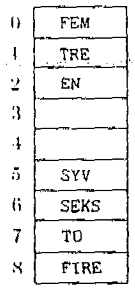


Fig. 41. Linear open addressing.

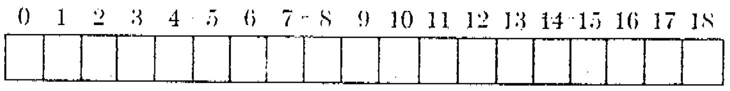
Program L (*Open scatter table search and insertion*). This program deals with full-word keys; but a key of 0 is not allowed, since 0 is used to signal an empty position in the table. (Alternatively, we could require the keys to be nonnegative, letting empty positions contain -1.) The table size M is 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 $9u$.

```
START  LDX   K              1         L1. Hash.
       ENTA  0              1
       DIV   =M=            1
       STX   *+1(0:2)       1
       ENT1  *              1         i ← h(K).
       LDA   K              1
       JMP   2F             1
8H     INC1  M+1            E         L3. Advance to next.
3H     DEC1  1              C+E-1     i ← i - 1.
2H     CMPA  TABLE,1        C+E       L2. Compare.
       JE    SUCCESS        C+E       Exit if K = KEY[i].
       LDX   TABLE,1        C+E-S
       JXNZ  3B             C+E-S     To L3 if TABLE[i] empty.
       J1N   8B             E+1-S     To L3 with i ← M if i = -1.
4H     LDX   VACANCIES      1-S       L4. Insert.
       JXZ   OVERFLOW       1-S       Exit with overflow if N = M - 1.
       DECX  1              1-S
       STX   VACANCIES      1-S       Increase N by 1.
       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 scatter table with $M = 19$, $N = 9$:



(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 is filled 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

6.4

$C'_N \approx \frac{1}{2}\Big(1$

$C_N \approx \frac{1}{2}\Big(1$

where $\alpha = N/M$ is the l
fast as Program C, when
fact that Program C deal
when α approaches 1 th
works, slowly but surely.
space in the table, so th
is $(M+1)/2$; we shall al
cessful search is approxim

The pileup phenomen
table is aggravated by t
$\{K, K+1, K+2, \ldots\}$ a
tive hash codes. Multiplic
Note that multiplicative
since we generally woul
$M = 2^m$ in order to disti
bit position is no problem

Another way to prote
$i \leftarrow i - c$ in step L3, ins
long as it is *relatively pri*
every position of the tab
somewhat slower. It does
records will still be form
appearance of consecutiv
help instead of a hindran

Although a fixed valu
improve the situation ni
important modification c
[Ph.D. thesis, Calif. Inst.

Algorithm D (*Open addr*
identical to Algorithm L,
by making use of two h
duces a value between 0 a
between 1 and $M-1$ t
prime, $h_2(K)$ can be *any*
$h_2(K)$ can be any *odd* val

D1. [First hash.] Set $i \leftarrow$

D2. [First probe.] If TABL
algorithm terminates

flash.

(K).

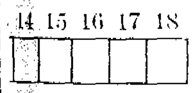
Advance to next.
i ← 1.
Compare.
if *K* = KEY[*i*].

3 if TABLE[*i*] empty.
3 with *i* ← *M* if *i* = −1.
Insert.
with overflow if N = *M* − 1.

ase N by 1.
E[i] ← *K*. ▮

ber of probes, and *S* tells
more the variable *E*, which
ade, since its average value

algorithm works fine until
cess slows down, with long
. The reason for this be-
wing hypothetical scatter



(21)

next key *K* to be inserted
, but these are not equally
11 ≤ *h*(*K*) ≤ 15, while it
position 11 is five times as
er.
nt for the relatively poor
rs in Algorithm C. (A list
im C as a list of length 1.)
l in (21); then two separate
er grow by more than one
f linear probing degrades

verage number of probes

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

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

where $\alpha = N/M$ is the load factor of the table. So 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 α 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 which 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. Note that multiplicative hashing is slightly awkward for the chained method, since we generally would want to use at least $(m+1)$-bit link fields when $M = 2^m$ in order to distinguish rapidly between Λ and a valid link. This wasted bit position is no problem with open addressing since there are no links.

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 will make Program L somewhat slower. It doesn'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 discovered 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 $h_2(K) = 1 + (K \bmod (M-1))$; but since $M - 1$ is even, it would be better to let $h_2(K) = 1 + (K \bmod (M-2))$. This suggests choosing M so that M and $M - 2$ are "twin primes" like 1021 and 1019. Alternatively, we could set $h_2(K) = 1 + (\lfloor K/M \rfloor \bmod (M-2))$, 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 (5) would be followed by

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

This is faster than the division method.

In each of the techniques suggested above, $h_1(K)$ and $h_2(K)$ are "independent," in the sense that different keys will have the same value for both h_1 and h_2 with probability $O(1/M^2)$ instead of $O(1/M)$. Empirical tests show that the behavior of Algorithm D with independent hash functions is essentially indistinguishable from the number of probes which 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

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

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
02          ENTA  0
03          DIV   =M=
04          STX   *+1(0:
05          ENT1  *
06          LDX   TABLE,
07          CMPX  K
       15   3H
       16
       17
       18
       19
       20
       21
       22   4H
       23
       24
       25
       26
       27
```

The frequency counts $A, C,$
to those in Program C above
the average. (If we restrict
B would be only about $\frac{1}{4}(C$
offset by a noticeable increase
keys in the table, the aver
search, are $A = 1$ in a succ
or $S1$ in a successful searc
number of probes is difficu
good agreement with formul

$$C'_N = \frac{M+1}{M+1-N}$$

$$C_N = \frac{M+1}{N}(H_{M+1} - H_M$$

when $h_1(K)$ and $h_2(K)$ are i
(25), the secondary clusterin

$$C'_N = \frac{M+1}{M+1-N} - \frac{N}{M+}$$

$$C_N = 1 + H_{M+1} - H_{M+1-}$$

set $i \leftarrow i + M$.
herwise if KEY[i] = K, the
back to D4.
tes with overflow. Other-
set KEY[i] ← K. ▌

omputing $h_2(K)$. If M is
$= 1 + (K \bmod (M-1))$;
$= 1 + (K \bmod (M-2))$.
e "twin primes" like 1021
$+ (\lfloor K/M \rfloor \bmod (M-2))$,
egister as a by-product of

shing, $h_2(K)$ can be com-
in" a-1, so that the coding

ts left. (24)

K) and $h_2(K)$ are "inde-
the same value for both
M). Empirical tests show
sh functions is essentially
would be required if the
practically no "piling up"

suggested by Gary Knott

K) = 0;
K) > 0. (25)

t we shall see that it does
ring slightly more probes
will follow the same path.
ine whether the gain in

re enough differences that
esponding MIX programs.

ince this program is sub-
ents. rI2 ≡ c − 1.

```
      START  LDX   K              1
             ENTA  0              1
             DIV   =M=            1
             STX   *+1(0:2)       1
             ENT1  *              1
             LDX   TABLE,1        1
             CMPX  K              1
             JE    SUCCESS        1
             JXZ   4F             1 − S1
             SRAX  5              A − S1
             DIV   =M−2=          A − S1
             STX   *+1(0:2)       A − S1
             ENT2  *              A − S1
             LDA   K              A − S1
      3H     DEC1  1,2            C − 1
             J1NN  *+2            C − 1
             INC1  M              B
             CMPA  TABLE,1        C − 1
             JE    SUCCESS        C − 1
             LDX   TABLE,1        C − 1 − S2
             JXNZ  3B             C − 1 − S2
      4H     LDX   VACANCIES      1 − S
             JXZ   OVERFLOW       1 − S
             DECX  1              1 − S
             STX   VACANCIES      1 − S
             LDA   K              1 − S
             STA   TABLE,1        1 − S  ▌
```

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 $\frac{1}{2}(C-1)$ on the average. (If we restricted the range of $h_2(K)$ to, say, $1 \le h_2(K) \le \frac{1}{2}M$, B would be only about $\frac{1}{4}(C-1)$; 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, α 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 $1 - \frac{1}{2}((N-1)/M) \approx 1 - \frac{1}{2}\alpha$. The average number of probes is difficult to determine exactly, but empirical tests show good agreement with formulas derived below for "uniform probing," namely

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

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

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 these formulas to be increased to

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

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

BTEX0000233

(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.

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 which involves secondary clustering, replacing the rather slow calculation of $h_2(K)$ in lines 10–13 by the three instructions

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

In this case, secondary clustering is preferable to independent double hashing.

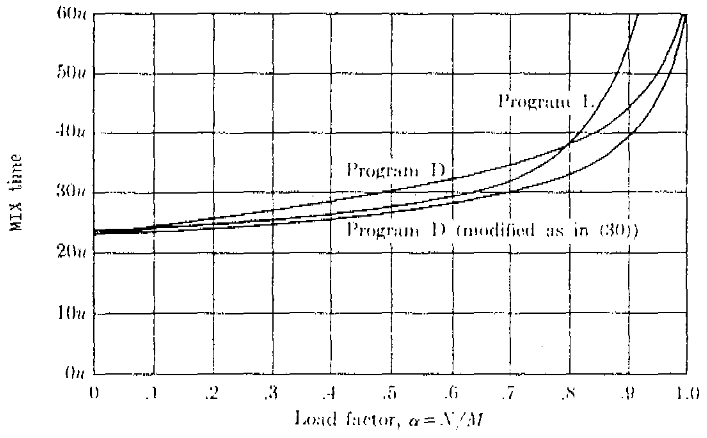


Fig. 42. The running time for successful searching by three open addressing schemes.

On a binary computer, we could speed up the computation of $h_2(K)$ in another way, replacing lines 10–13 by, say,

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

if M is a prime greater than 512. 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.

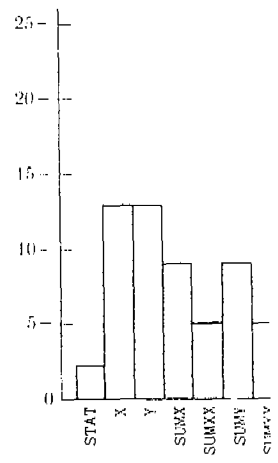


Fig. 43. The number of tin
names are listed from left t

Brent's Variation. Rich
rithm D so that the av
table gets full. His met
much more common than
doing more work when i
the expected retrieval tin

For example, Fig. 43
ally found to appear, in
PL/I compiler which use
looking up many of the n
Similarly, Bell and Kan
table algorithm 10988 ti
insertions into the table;
unsuccessful search. So
example, a table of symb
purely for retrieval.

Brent's idea is to ch
Suppose an unsuccessful

where $p_j = (h_1(K) - ji$
insert K in position p_j as
K_0 = KEY[p_0], and see if

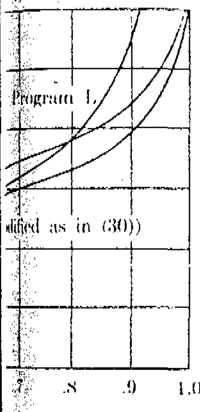
these values of C_N approach
$N = M$; this is much better
in the chaining methods.
orithm L, double hashing is
re 42 compares the average
modified Program D which
er slow calculation of $h_2(K)$

$- i.$

(30)

$0, c \leftarrow 1.$

independent double hashing.



three open addressing schemes.

he computation of $h_2(K)$ in

mod 512.

(31)

$+1.$

ggested by Bell and Kaman,
thm D independently) avoids
er division.
posed as improvements on
orithm D except possibly the

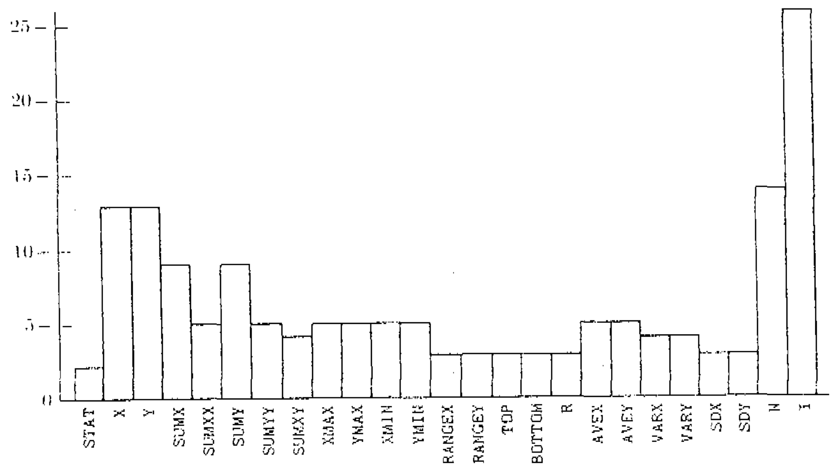


Fig. 43. The number of times a compiler typically searches for variable names. The names are listed from left to right in order of their first appearance.

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 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. [*CACM* 16 (1973), 105–109.]

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 which 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 $p_j = (h_1(K) - jh_2(K)) \bmod M$ 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 $t \geq 2$ steps, so it results in a net improvement. Similarly, if TABLE[$(p_0 - c_0) \bmod M$] is occupied and $t \geq 3$, we try TABLE[$(p_0 - 2c_0) \bmod M$]; if that is full too, we compute $c_1 = h_2(\text{KEY}[p_1])$ and try TABLE[$(p_1 - c_1) \bmod M$]; etc. In general, let $c_j = h_2(\text{KEY}[p_j])$ and $p_{j,k} = (p_j - kc_j) \bmod M$; if we have found TABLE[$p_{j,k}$] occupied for all indices j, 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 as shown in Fig. 44, on page 539, 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 the order of $\sqrt{M}$; and the number of additional table positions probed while deciding how to make the insertion is about $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^6 + \cdots$.

Deletions. Many computer programmers have great faith in algorithms, and they are surprised to find that *the obvious way to delete records from a scatter 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 (i.e., Algorithm L), it is possible to do 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*
table has been constructed
from a given position TABLE[

R1. [Empty a cell.] Mark T

R2. [Decrease i.] Set $i \leftarrow i$

R3. [Inspect TABLE[i].] If TA
wise set $r \leftarrow h(\text{KEY}[i])$, t
position i. If $i \leq r <$
lies cyclically between i

R4. [Move a record.] Set TA

Exercise 22 shows that t
i.e., the average number of
same. (A similar result for t
the validity of Algorithm R
involved, and no analogous
possible.

Of course when chainin
value, deletion causes no p
linear list. Deletion with A

***Analysis of the algorithms**
behavior of a hashing meth
laws of probability whenev
almost unthinkably bad, so
is very good.

Before we get into the
very approximate model of
(cf. W. W. Peterson, *IBM*
this model, we assume that
that each of the $\binom{M}{N}$ possib
empty cells is equally likely.
dary clustering; the occupan
of the others. For this mod
to insert the $(N + 1)$st ite
given cells are occupied and

P_r

therefore the average numb

$$C'_N = \sum_{1 \leq r \leq M} rP_r = M + 1$$

$$= M + 1 - \sum_{1 \leq r \leq M} (M$$

increases the retrieval time
time for K by $t \geq 2$ steps,
TABLE[$(p_0 - c_0) \bmod M$] is
M]; if that is full too, we
mod M]; etc. In general,
if we have found TABLE[$p_{j,k}$]
and if $t \geq r + 1$, we look at
the first empty space occurs
and insert K in position p_j.
umber of probes per successful
.39, with a maximum value of

sful search is not reduced by
ted by Eq. (26), approaching
number of times h_2 needs to
$+ \cdots$, according to Brent's
and the number of additional
make the insertion is about

great faith in algorithms, and
to delete records from a scatter
te the key EN from Fig. 41, we
cause another key FEM would
both hashed to the same loca-
mpty place, indicating an un-
ith Algorithm C, due to the
D and FIRE from Fig. 40.
ing a special code value in the
table entries: empty, occupied,
ld skip over deleted cells, as if
ul, the key can be inserted in
was encountered.
ns are very rare, because the
nce they have been occupied.
ld deletions, all of the empty
successful search will take M
increased, since we will have to
in step D4; and the number of
om C_N to C'_N.
orithm L), it is possible to do
of affairs, if we are willing to

Algorithm R (*Deletion with linear probing*). Assuming that an open scatter 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 $r \leftarrow h(\text{KEY}[i])$, the original hash address of the key now stored at position i. If $i \leq r < j$ or if $r < j < i$ or $j < i \leq r$ (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, i.e., the average number of probes predicted in Eqs. (22) and (23) remains the same. (A similar 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.

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.

***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 a very approximate model of the situation, which may be called *uniform hashing* (cf. W. W. Peterson, *IBM J. Research & Development* 1 (1957), 135–136). In this model, we assume that the keys go into random locations 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 the others. For this model the probability that exactly r probes are needed 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 hashing is

$$C'_N = \sum_{1 \leq r \leq M} rP_r = M + 1 - \sum_{1 \leq r \leq M} (M + 1 - r)P_r$$
$$= M + 1 - \sum_{1 \leq r \leq M} (M + 1 - r)\binom{M-r}{M-N-1} \Big/ \binom{M}{N}$$

$$= M + 1 - \sum_{1 \le r \le M} (M - N)\binom{M+1-r}{M-N} \Big/ \binom{M}{N} \qquad (32)$$

$$= M + 1 - (M - N)\binom{M+1}{M-N+1} \Big/ \binom{M}{N}$$

$$= M + 1 - (M - N)\frac{M+1}{M-N+1} = \frac{M+1}{M-N+1}, \quad \text{for} \quad 1 \le N < M.$$

(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

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

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

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

$$= \frac{M+1}{N}\,(H_{M+1} - H_{M-N+1}) \approx \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \qquad (34)$$

As remarked above, extensive tests show that Algorithm D with two independent hash functions behaves essentially like uniform hashing, for all practical purposes.

This completes the analysis of uniform hashing. 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"

$$a_1\, a_2 \ldots a_N, \qquad 0 \le a_j < M, \qquad (35)$$

is equally likely, where a_j denotes the initial hash address of the jth 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 kth 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 Nth 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,

$$C_N = \frac{1}{N}\sum_{0 \le k < N} C'_k, \qquad (36)$$

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

Strictly speaking, thei
In the first place, the di
because the keys themsel
$a_1 = a_2$ slightly less than
set of all possible keys is t
Furthermore a good hash
data, making it even less l
number of probes will b
indicated in Fig. 43: Keys
likely to be looked up th
C_N tends to be doubly pe
better in practice than ou

With these precaution
probing.* Let $f(M, N)$ be
0 of the table will be emp
The circular symmetry of
as often as any other pos
other words

Now let $g(M, N, k)$ be th
rithm leaves position 0 e
$k + 1$ empty. We have

$g(M, N, k) =$

because all such hash seq
taining k elements $a_i \le k$
k occupied and one (contai
$k + 1$ empty; there are
$f(M - k - 1, N - k)$ of
such subsequences. Finall
will be needed when the
that

$P_k = M^{-N}(g(M,$

Now $C'_N = \sum_{0 \le k \le N} (k +$
and simplifying yields the

* The author cannot resist
the following derivation in 19
gramming. Since this was the
has had a strong influence on
ten years would go by before

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

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

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, 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) that

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

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

* 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 has had a strong influence on the structure of these books. Little did I know that more than ten years would go by before this derivation got into print!

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

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

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

where

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

Proof. Details of the calculation are worked out in exercise 27. ∎

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

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

hence if $N/M = \alpha$,

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

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

i.e.,

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

This relation gives us a good estimate of $Q_r(M, N)$ when M is large and α is not too close to 1. (The lower bound is a better approximation than the upper bound.) When α 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).

Another approach to the analysis of linear probing has been taken by G. Schay, Jr. and W. G. Spruth [*CACM* 5 (1962), 459–462]. Although their method yields 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 which 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 which hash to each address.*

6.4

In other words, any r
a hash sequence with the
addresses. (We are assum
equal importance. If some
proof can be extended to s
them in decreasing order o

Proof. It suffices to show t
for the hash sequence $a_1\ a$
$a_1 \ldots a_{i-1}a_{i+1}a_ia_{i+2} \ldots a$
the $(i+1)$st key in the se
ith in the first sequence. B
so the number of probes fo
the number for the ith is i

Theorem P tells us t
$a_1\ a_2 \ldots a_N$ can be detern
the number of a's that equ
sequence" $c_0\ c_1 \ldots c_{M-1}$,
tions j and $j-1$ are probe
by the rule

$$c_j = \begin{cases} 0 \\ b_j + c_{(j+1)} \end{cases}$$

For example, let $M = 10$,
$c_0 \ldots c_9 = 2\ 3\ 1\ 0\ 0\ 0\ 0$
from position 2 to position
from position 0 to position
the average number of prob

1 +

Rule (44) seems to be a circ
actually there is a unique
(see exercise 32).

Schay and Spruth use
$c_j = k$, in terms of the pr
independent of j.) Thus

$q_0 =$
$q_1 =$
$q_2 =$

etc., since, for example, t
$b_j + c_{(j+1)\bmod M} = 3$. Let
functions for these probabili

$B(z)C(z) = p_0q_0 + (q_0$

1

Algorithm L, assuming that

ful search), (40)

cessful search), (41)

$\frac{1)}{} + \cdots$

$\frac{1}{} \cdots \frac{N-k+1}{M}.$ (42)

exercise 27. ∎

which appears in this theorem

$k+1) \le N^k;$

$\le \sum_{k\ge 0} \binom{r+k}{k} N^k/M^k,$

$(M, \alpha M) \le \sum_{k\ge 0} \binom{r+k}{k} \alpha^k,$

$\alpha M) \le \frac{1}{(1-\alpha)^{r+1}}.$ (43)

when M is large and α is not
proximation than the upper
ome useless, but fortunately
ptotic behavior was studied
1) is simply equal to M

robing has been taken by
459–462]. Although their
formulas in Theorem K, it
sketch it briefly here. First
g which was first noticed

ssful search by Algorithm L
erted; it depends only on the

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, by 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 ith in the first sequence. But then the ith 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 ith 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

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

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

$$C(z) = \frac{p_0 q_0}{1 - D(z)} = \frac{1 - D(1)}{1 - D(z)}, \tag{47}$$

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 see why this answer is different from the result in Theorem K? (Cf. 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 hashing* which we have analyzed above in (32), (34). However, Ullman has found an example with $M = 4$, $N = 2$ for which C'_N is smaller than the value $\frac{5}{3}$ obtained with uniform hashing. His construction assigns zero probability to all but the following six permutations:

6.4

Permutation	Probabi
0 1 2 3	$(1 + 2\epsilon$
2 0 1 3	$(1 - \epsilon)$,
3 0 1 2	$(1 - \epsilon)$,

Roughly speaking, the first c
or 1. The average number of
to be $\frac{5}{3} - \frac{1}{9}\epsilon + O(\epsilon^2)$, so we c
a small positive value.

However, the correspondi
which is larger than $\frac{5}{4}$ (the u
any assignment of probabilit
some N always implies that
you can't win all the time ove

Actually the number of pr
than C'_N. The permutations i
any N, and indeed it seems
probabilities will be able to ma
$(H_{M+1} - H_{M+1-N})$.

This conjecture appears t
there are many ways to assig
hashing; we do not need to a
the following assignment for M

Permutation	Probabil
0 1 2 3	1/6
1 2 3 0	1/6
2 3 0 1	1/6
3 0 1 2	1/6

with zero probability assigned
The following theorem c
behavior of uniform hashing.

Theorem U. *An assignment of*
$\binom{M}{N}$ configurations of empty and
$0 < N < M$, if and only if th
whose first N elements are the m
and all N-element sets.

For example, the sum of p
permutations beginning with
$1/\binom{M}{3} = 3!(M-3)!/M!$. Ob
(53).

D(z), and it follows that

(1) / D(z), (47)

ded for retrieval, according

$+\frac{M}{2N}\,\frac{B''(1)}{1-B'(1)}$. (48)

$a_1 \ldots a_N$ is equally likely we,

to j, for fixed j)

(49)

$B''(1) = \frac{N(N-1)}{M^2}$, (50)

(48) will be

(51)

from the result in Theorem K?

examples of probe sequences
one that can be proved "best
has been set up in the follow-
(1972), 569–575]: Instead of
key K into an entire permuta-
probe sequence to use for K.
bability, and the generalized
tation with that probability.
ties to permutations gives the
ding average number of probes

$M!$ to each permutation, it is
uniform hashing which we have
has found an example with
value $\frac{5}{3}$ obtained with uniform
ity to all but the following six

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 choice favors 2 and 3, but the second choice is 0 or 1. The average number of probes needed to insert the third item turns out to be $\frac{5}{3} - \frac{1}{9}\epsilon + O(\epsilon^2)$, so we can improve on uniform hashing by taking ϵ 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 hashing value). Ullman has 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 hashing.

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 it seems reasonable to conjecture that no assignment of probabilities will be able to make C_N less than the uniform value $((M+1)/N) \times (H_{M+1} - H_{M+1-N})$.

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

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 hashing.

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 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).

Proof. Let $A \subseteq \{0, 1, \ldots, M-1\}$, and let $\prod(A)$ be the set of all permutations whose first $\|A\|$ elements are members of A, and let $S(A)$ be the sum of the probabilities assigned to these 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; and let $P(A) = P_1(A) + P_2(A) + \cdots$. The proof is by induction on $N \geq 1$, 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

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

where $\Pr(\pi)$ is the probability assigned to permutation π and π_k is its kth 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

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

hence

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

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 *buckets*, so that several records are fetched from the external memory each time.

The file is usually 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*
a link to an "overflow" re
These overflow records are
advantage in having buck
overflows occur; thus, the
$(b+k)$th record of a list
leave some room for overf
accesses are to the same cy
Although this method
overflows is statistically s
good. See Tables 2 and 3,
as a function of the load f

for fixed α as $M, N \to \infty$,
accesses for an unsuccessfu

AVERAGE ACCESSES IN A

Bucket size, b	10%	20%	30
1	1.0048	1.0187	1.04
2	1.0012	1.0088	1.02
3	1.0003	1.0038	1.01
4	1.0001	1.0016	1.0
5	1.0000	1.0007	1.00
10	1.0000	1.0000	1.00
20	1.0000	1.0000	1.0
50	1.0000	1.0000	1.00

AVERAGE ACCESSES IN

Bucket size, b	10%	20%	30
1	1.0500	1.1000	1.15
2	1.0063	1.0242	1.05
3	1.0010	1.0071	1.02
4	1.0002	1.0023	1.00
5	1.0000	1.0008	1.00
10	1.0000	1.0000	1.00
20	1.0000	1.0000	1.00
50	1.0000	1.0000	1.00

be the set of all permuta-
nd let $S(A)$ be the sum of
$P_k(A)$ be the probability
procedure occupy the loca-
ired exactly k probes; and
by induction on $N \geq 1$,

M)
n)

N-element set. Then

(π_k)),

utation π and π_k is its kth

$\Pr(\pi)$,

$k < N$;

$\binom{N}{k} \Big/ \binom{M}{k}$),

the correct value. ∎

lves well to external search-
ms. For such applications,
of accesses to the file, and

the hash function, since the
the cost of the extra time

so that several records are

ning b records each. Colli-
ve the same hash address.
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 α 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	Load factor, α									
size, b	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	1.3
2	1.0012	1.0088	1.0269	1.0581	1.1036	1.1638	1.238	1.327	1.428	1.5
3	1.0003	1.0038	1.0162	1.0433	1.0898	1.1588	1.252	1.369	1.509	1.6
4	1.0001	1.0016	1.0095	1.0314	1.0751	1.1476	1.253	1.394	1.571	1.7
5	1.0000	1.0007	1.0056	1.0225	1.0619	1.1346	1.249	1.410	1.620	1.7
10	1.0000	1.0000	1.0004	1.0041	1.0222	1.0773	1.201	1.426	1.773	2.0
20	1.0000	1.0000	1.0000	1.0001	1.0028	1.0234	1.113	1.367	1.898	2.3
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0007	1.018	1.182	1.920	2.7

Table 3

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket	Load factor, α									
size, b	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	1.5
2	1.0063	1.0242	1.0520	1.0883	1.1321	1.1823	1.238	1.299	1.364	1.4
3	1.0010	1.0071	1.0216	1.0458	1.0806	1.1259	1.181	1.246	1.319	1.4
4	1.0002	1.0023	1.0097	1.0257	1.0527	1.0922	1.145	1.211	1.290	1.3
5	1.0000	1.0008	1.0046	1.0151	1.0358	1.0699	1.119	1.186	1.286	1.3
10	1.0000	1.0000	1.0002	1.0015	1.0070	1.0226	1.056	1.115	1.206	1.3
20	1.0000	1.0000	1.0000	1.0000	1.0005	1.0038	1.018	1.059	1.150	1.2
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.015	1.083	1.2

BTEX0000245

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 used which links 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 this count becomes zero. A 'roving pointer' can be used to distribute overflows (cf. exercise 2.5–6), so that different lists tend to use different overflow buckets. It may be a good idea to have separate available-space lists for the buckets on each cylinder of a disk file.

This method has not yet been analyzed, but it should 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 which 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

Table 4

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY LINEAR PROBING

Bucket	Load factor, α									
size, b	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.[illegible]
2	1.0062	1.0242	1.0553	1.1033	1.1767	1.2930	1.494	1.903	3.147	5.6
3	1.0009	1.0066	1.0201	1.0450	1.0872	1.1584	1.286	1.554	2.378	4.0
4	1.0001	1.0021	1.0085	1.0227	1.0497	1.0984	1.190	1.386	2.000	3.2
5	1.0000	1.0007	1.0039	1.0124	1.0307	1.0661	1.136	1.289	1.777	2.7
10	1.0000	1.0000	1.0001	1.0011	1.0047	1.0154	1.042	1.110	1.345	1.8
20	1.0000	1.0000	1.0000	1.0000	1.0003	1.0020	1.010	1.036	1.144	1.4
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.005	1.040	1.1

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!

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

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

iding a separate overflow
doubly linked list of avail-
ucket that is not yet full.
f how many record posi-
he doubly linked list only
can be used to distribute
s tend to use different over-
rate available-space lists for

it should prove to be quite

ks, using an "open" method.
probing when we consider
often be chosen so that it
esses. The approximate theo-
out above can be generalized
hows that linear probing is
very full. For example, see
bucket size is 50, the average

CH BY LINEAR PROBING

%	70%	80%	90%	95%
500	2.167	3.000	5.500	10.5
930	1.494	1.903	3.147	5.6
584	1.286	1.554	2.378	4.0
084	1.190	1.386	2.000	3.2
661	1.136	1.289	1.777	2.7
154	1.042	1.110	1.345	1.8
020	1.010	1.036	1.144	1.4
000	1.001	1.005	1.040	1.1

1.04. This is actually *better*
method (A) with the same

some very interesting mathe-
since the details are worked
two functions strongly related

$$\frac{n^3\alpha^2}{(n+2)(n+3)} + \cdots, \quad (55)$$

and

$$t_n(\alpha) = e^{-n\alpha}\left(\frac{(\alpha n)^n}{(n+1)!} + 2\,\frac{(\alpha n)^{n+1}}{(n+2)!} + 3\,\frac{(\alpha n)^{n+2}}{(n+3)!} + \cdots\right)$$
$$= \frac{e^{-n\alpha}n^n\alpha^n}{n!}\bigl(1 - (1-\alpha)\,R(\alpha, n)\bigr). \tag{56}$$

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

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

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

$$C_N = 1 + \bigl(1 - \tfrac{1}{2}b(1-\alpha)\bigr)t_b(\alpha) + \frac{e^{-b\alpha}b^b\alpha^b}{2b!}\,R(\alpha, b) + O\left(\frac{1}{M}\right). \tag{58}$$

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 $M(C'_N - 1) = Nt_b(\alpha)$, 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 α, 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 α is large; fortunately the series for $R(\alpha, n)$ converges rather rapidly even when α is large, so the formulas can be evaluated exactly without much difficulty. The maximum values occur for $\alpha = 1$, when

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

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

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 access 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 which 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

BTEX0000247

chaining method if we had M/k buckets of size kb, namely $Nl_{kb}(\alpha)$ by what we said above. Further justification of Eq. (61) appears in exercise 55.

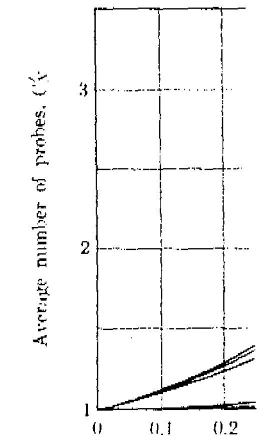
An excellent discussion of practical considerations involved in the design of external scatter tables has been given by Charles A. Olson, *Proc. ACM Nat'l Conf.* 24 (1969), 539–549. He includes several worked examples and points out that the number of overflow records will increase substantially if the file is subject to frequent insertion/deletion activity without relocating records; and he presents an analysis of this situation which was obtained jointly with J. A. de Peyster.

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 this 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 an average of less than 5.5 probes to locate a random item in the table. (However, the average number of probes needed to insert a *new* item with Algorithm L is 50.5, with a 90-percent-full table.)

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 scatter table of capacity 500 and an open scatter 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. (Cf. 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 scatter table allows efficient insertions.



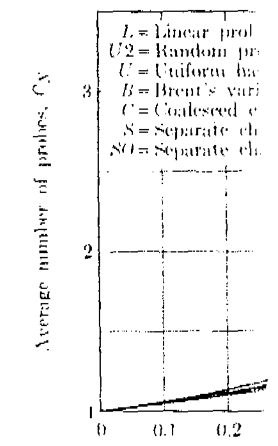


Fig. 44. Comparison of coll
number of probes as $M \to \infty$

We can also compare
allow dynamic insertions.
Program 6.2.2T when N is
(exercise 6.3–9) when N is

Only one search meth
with virtually no storage
His method allows us to

b), namely $Nt_{kb}(\alpha)$ by what we
ars in exercise 55.
ations involved in the design
es A. Olson, *Proc. ACM Nat'l*
rked examples and points out
se substantially if the file is
thout relocating records; and
ich was obtained jointly with

died a large number of tech-
ht one for a given application?
relevant details of the "trade-
but the following things seem
the speed of searching and the

ction, showing that the various
t numbers of probes. But this
obe varies in different methods,
the running time (as we have
ble more frequently than the
antage of simplicity. Further-
en the table is 90 percent full,
probes to locate a random item
probes needed to insert a *new*
-full table.)
ds are quite economical with
memory space needed for link
ractive for small records. For
d scatter table of capacity 500
atter is clearly preferable, since
re present and it is capable of
nd, sometimes the record size
ually no extra cost. (Cf. exer-

ther search strategies we have
speed we can argue that they
because the average search
if we stipulate that the table
ll take only about 55 units of
90 percent full; this beats the
(exercise 6.2.1–24) when N is
cent in storage space. More-
tables, while a scatter table

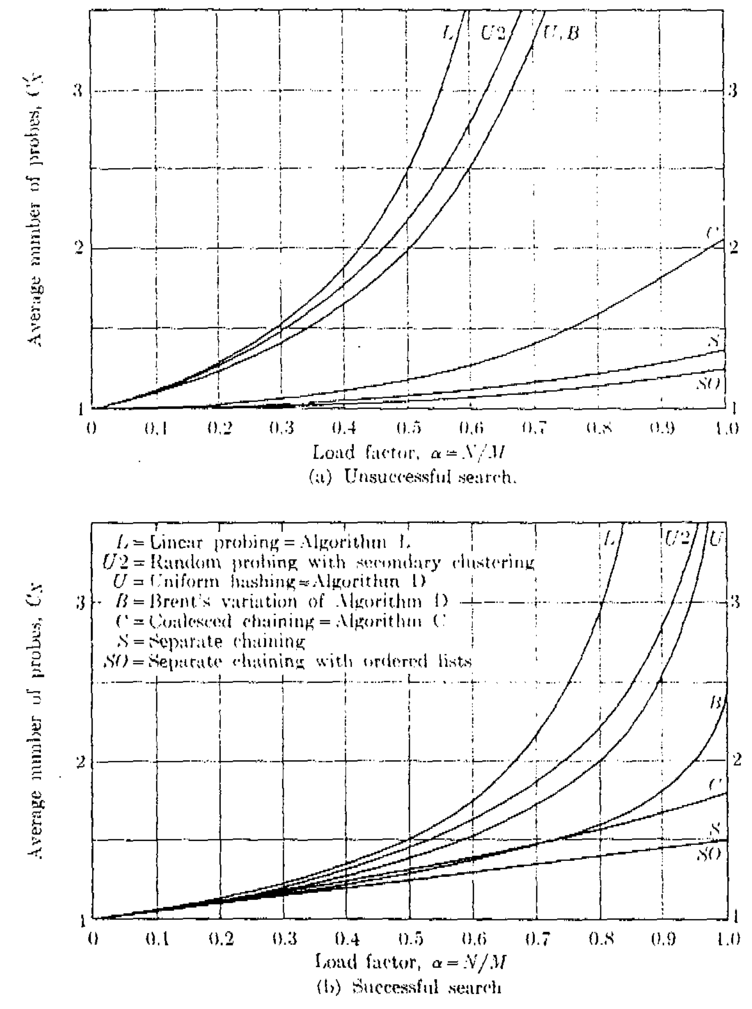


Fig. 44. Comparison of collision resolution methods: Limiting values of the average number of probes as $M \to \infty$.

We can also compare Program L to the tree-oriented search methods which 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 $2\frac{1}{2}$ probes on the average. No extra space for link fields, etc., is needed; however, an unsuccessful search will be very slow, requiring about $\frac{1}{2}N$ probes.

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

a) After an unsuccessful search in a scatter table, we know only that the desired key is not present. Search methods based on comparisons always yield more information, making it possible to find the largest key $\leq K$ and/or the smallest key $\geq K$; this is important in many applications (e.g., for interpolation of function values from a stored table). It is also possible to use comparison-based algorithms to locate all keys which 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, and this is occasionally desirable.

b) The storage allocation for scatter 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. When a scatter table overflows, it is probably best to "rehash" it, i.e., to allocate a larger space and to change the hash function, reinserting every record into the larger table. F. R. A. Hopgood [*Comp. Bulletin* 11 (1968), 297–300] has suggested rehashing the table when it becomes α_0 percent full, replacing M by $d_0 M$; suitable choices of these parameters α_0 and d_0 can be made by using the analyses above and characteristics of the data, so that the critical point at which it becomes cheaper to rehash can be determined. (Note that the method of chaining does not lead to any troublesome overflows, so it requires no rehashing; but the search time is proportional to N when M is fixed and N gets large.) By contrast, the tree search and insertion algorithms require no such painful rehashing; the trees grow no larger than necessary. In a virtual memory environment we probably ought to use tree search or digital tree search, instead of creating a large scatter table that requires bringing in 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 are never completely sure that a hash function will perform properly when it is applied to a new set of data. Therefore scatter storage would be 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 suggesting the use of chaining, in

January 1953; this was al
He pointed out the desir
element, for external searc
analysis a little further, an
used "degenerative address
put in secondary bucket 27
27, etc., assuming the pr
buckets, 100 tertiary buck
Luhn were digital in charac
so that 31415926 would be

At about the same ti
another group of IBMers:
and Arthur L. Samuel, wh
701. In order to handle t
open addressing with linea

Hash coding was first d
Computers and Automation
mention the idea of dividi
the hash address. Dumey'
addressing. A. P. Ersho
addressing in 1957 [*Doklad*
lished empirical results abo
the average number of pro

A classic article by W
(1957), 130–146, was the fir
in large files. Peterson de
formance of uniform hashi
behavior of linear open add
dation in performance that
prehensive survey of the
Buchholz [*IBM Systems J.*
cussion of hash functions.

Up to this time linear p
that had appeared in the
random probing by indepe
veloped by several people (
became very widely used,
Then Robert Morris wrote
(1968), 38–43, in which he
dary clustering). Morris's p
in Algorithm D and its refi

It is interesting to note
in print, with its present m
although it had already be

ge. No extra space for link
ch will be very slow, requiring

other hand, there are three
g is inferior to other methods

able, we know only that the
on comparisons always yield
largest key $\leq K$ and/or the
cations (e.g., for interpolation
o possible to use comparison-
en two given values K and K'.
6.2 make it easy to traverse
out sorting it separately, and

often somewhat difficult; we
use as the hash table, and it
allotted. If we provide too
expense of other lists or other
room, the table will overflow.
to "rehash" it, i.e., to allocate
inserting every record into the
11 (1968), 297–300] has sug-
percent full, replacing M by
d_0 can be made by using the
so that the critical point at
ined. (Note that the method
lows, so it requires no rehash-
M is fixed and N gets large.)
thms require no such painful
In a virtual memory environ-
digital tree search, instead of
ng in a new page nearly every

obability theory when we use
the average, while their worst
ber generators, we are never
properly when it is applied
would be inappropriate for
control, where people's lives
ons 6.2.3 and 6.2.4 are much
on the search time.

originated by H. P. Luhn,
sting the use of chaining, in

January 1953; this was also among the first applications of linked linear lists. He pointed out the desirability of using buckets containing more than one element, for external searching. Shortly afterwards, A. D. Lin carried Luhn's analysis a little further, and suggested a technique for handling overflows that used "degenerative addresses"; e.g., the overflows from primary bucket 2748 are put in secondary bucket 274; overflows from that bucket go to tertiary bucket 27, etc., assuming the presence of 10000 primary buckets, 1000 secondary buckets, 100 tertiary buckets, etc. The hash functions originally suggested by Luhn were digital in character, e.g., adding adjacent pairs of key digits 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. Boehme, 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.

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 hashing, 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.

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 which 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 Morris's article was published in 1968, although it had already become common jargon in several parts of the world

BTEX0000251

by that time. The only previous occurrence of the word 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-1960's, yet nobody was rash enough to use such an undignified word publicly until 1968!

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 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 ten or less characters in length. He decided to use a scatter 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, such that distinct keys differing in t or fewer bits will hash to different addresses. Given n and $t \leq n$, 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 $n = 15$, $k = 4$, $t = 6$, 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 α 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 α^{2j} is a root of $P(x)$ whenever α^j is a root, it follows that the coefficients p_i of $P(x)$ satisfy $p_i^2 = p_i$, so they are all 0 or 1.

Prove that if $R(x) = r_{n-1}$
modulo 2, with at most t nonzer
modulo 2. [It follows that the co

8. [*M34*] (*The three-distance the*
1, whose regular continued fracti
$\theta = /a_1, a_2, a_3, \ldots/$. Let $q_0 = 0$
$p_{k+1} = a_kp_k + p_{k-1}$, for $k \geq 1$.
denote $x - \lceil x \rceil + 1$. As the poi
the interval [0, 1], let the line se
that the first segment of a given
that the following statements ar
$t = rq_k + q_{k-1}$ and $0 \leq r < a_k$
and right endpoint $\{(s + t)\theta\}^+$.
$rq_k + q_{k-1}$ and $0 \leq r < a_k$ and k
and right endpoint $\{s\theta\}^+$. Ever
$n = rq_k + q_{k-1} + s$ for some k
representation, just before the po

the first s intervals (number
the first $n - q_k$ intervals (n
the last $q_k - s$ intervals (nu
$\{(-1)^k((r - 1)q_k + q_{k-1})$

The operation of inserting $\{n\theta\}$
converts it into interval number

9. [*M30*] When we successivel
[0, 1], Theorem S asserts that e
remaining intervals. If the inter
we may call it a *bad break* if one
i.e. if $b - a > 2(c - b)$ or $c -$
Prove that bad breaks will
and the latter values of θ *never* p

10. [*M48*] (R. L. Graham.) Pr
if $\theta, \alpha_1, \ldots, \alpha_d$ are real numbers
and if the points $\{n\theta + \alpha_i\}$ ar
$1 \leq i \leq d$, the resulting $n_1 + \cdots$
different lengths.

11. [*16*] Successful searches are
it therefore be a good idea to int

▸ **12.** [*21*] Show that Program C
jump instruction in the inner loo
with the original.

▸ **13.** [*24*] (*Abbreviated keys.*) Let
of K such that K can be determ
division hashing we may let $h(K)$
hashing we may let $h(K)$ be the
other bits.

Prove that if $R(x) = r_{n-1}x^{n-1} + \cdots + r_1x + r_0$ 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 θ 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 $q_{k+1} = a_kq_k + q_{k-1}$, $p_{k+1} = a_kp_k + p_{k-1}$, for $k \geq 1$. 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 $0 \leq r < a_k$ and k is even and $0 \leq s < q_k$, 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 $0 \leq r < a_k$ and k is odd and $0 \leq s < q_k$, 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 $k \geq 1$, $1 \leq r \leq a_k$, $0 \leq s < q_k$. 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 $\{(-1)^kq_k\theta\}$;
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 latter 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 $\{\theta\}$, $\{2\theta\}$, $\ldots$ 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, i.e. 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 ϕ^{-2}; and the latter values of θ *never* produce bad breaks.

10. [*M48*] (R. L. Graham.) Prove or disprove the following *3d distance conjecture:* 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 $\{n\theta + \alpha_i\}$ are inserted into the interval [0, 1] for $0 \leq n < n_i$, $1 \leq i \leq d$, the resulting $n_1 + \cdots + n_d$ (possibly empty) intervals have at most $3d$ different lengths.

11. [*16*] Successful searches are usually 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 scatter table *share memory* with any number of other linked lists. Let every word of the list area have a 2-bit TAG field, with the following interpretation:

> TAG(P) = 0 indicates a word in the list of available space; LINK(P) points to the next available word.
>
> TAG(P) = 1 indicates any word in use that is not part of the scatter table; the other fields of this word may have any desired format.
>
> TAG(P) = 2 indicates a word in the scatter table; LINK(P) points to another word. Whenever we are processing a list that is not part of the scatter table and we access a word with TAG(P) = 2, we are supposed to set P ← LINK(P) until reaching a word with TAG(P) ≤ 1. (For efficiency we might also then change one of the prior links so that it will not be necessary to skip over the same scatter table entries again and again.)

Show how to define suitable algorithms for inserting and retrieving keys from such a combined table, assuming that words with TAG(P) = 2 also have another link field AUX(P).

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) $1 \le h_2(K) \le r$ for $r = 1, 2, 3, \ldots, 10$; (b) $1 \le h_2(K) \le \rho M$ 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 $h_1(K)$, $(h_1(K) - 1) \bmod M, \ldots, (h_1(K) - \binom{M}{2}) \bmod M$ will be a permutation of $\{0, 1, \ldots, M - 1\}$. When this method is programmed for MIX, how does it compare with the three programs considered in Fig. 42, assuming that the behavior is 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 which 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.** [*23*] Design an algorithm analogous to Algorithm R, for deleting entries from a chained scatter table that has been constructed by Algorithm C.

6.4

24. [*M20*] Suppose that t
where exactly P keys hash
if the keys are arbitrary 10-
that $M \ge 7$ and $N = 7$,
of all possible keys, what is
will be obtained (i.e., that
M and P?

25. [*M19*] Explain why E

26. [*M20*] How many has
(21), using linear probing?

27. [*M27*] Complete the p

$s(n, x, y) =$

use Abel's binomial theor
$ns(n - 1, x + 1, y - 1)$.]

28. [*M30*] In the old day
was possible to watch the
When the table began to fi
others took a great deal of

This experience sugge
linear probing is used. Fi
Q_r functions defined in T
$M \to \infty$.

29. [*M21*] (*The parking p*
in a row, numbered 1 thro
she wakes up and orders
available space; but if ther
(i.e., if his wife awoke whe
are all full), he expresses hi

Suppose, in fact, that
up just in time to park at
of the cars get safely park
nobody leaves after park
3 1 4 1 5 9 2 6 5, the cars g

[*Hint:* Use the analysis of

30. [*M28*] (John Riordan.
show that all cars get park
$\{1, 2, \ldots, n\}$ such that a_j

31. [*M40*] When $n = m$ i
tions turns out to be $(n + 1$
as the number of free trees
between parking sequences

24. [*M20*] Suppose that the set of all possible keys that can occur has MP elements, where exactly P keys hash to a given address. (In practical cases, P is very large; e.g. 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 (i.e., that $h(K_1) = 1$, $h(K_2) = 2, \ldots, h(K_7) = 1$), 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

$$s(n, x, y) = \sum_{k \geq 0} \binom{n}{k} (x + k)^{k+1} (y - k)^{n-k-1} (y - n);$$

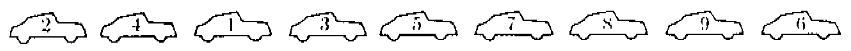
use Abel's-binomial theorem, Eq. 1.2.6–16, to prove that $s(n, x, y) = x(x + y)^n + ns(n - 1, x + 1, y - 1)$.]

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 C'_N is rather high, when linear probing is used. Find a formula which 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 (i.e., if his wife awoke when the car approached space k, but spaces $k, k + 1, \ldots, m$ are all full), he expresses his regrets and drives on.

Suppose, in fact, that this happens for n different cars, where the jth 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. [*M28*] (John Riordan.) 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 $a_j \leq p_j$ 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 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. [*M26*] 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 which finds 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.** [*M22*] The purpose of this exercise is to investigate the average number of probes in a chained scatter 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 $P_N(z) = \sum_{k\geq 0} P_{Nk}z^k$. (c) Express the average number of probes for successful and unsuccessful search in terms of this generating function. (Assume that an unsuccessful search in a list of length k requires $k + \delta_{k0}$ probes.)

35. [*M21*] 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?

36. [*M22*] Find the variance of (18), the number of probes in separate chaining when the search is unsuccessful.

▸ **37.** [*M29*] Find the variance of (19), the number of probes in separate chaining when the search is successful.

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:* Cf. Eq. 5.2.1–11.]

39. [*M30*] The purpose of this exercise is to analyze the average number of probes in Algorithm C (chaining with coalescing lists). Let $c(k_1, k_2, k_3, \ldots)$ be the number of hash sequences (35) that cause Algorithm C to form exactly k_1 lists of length 1, k_2 of length 2, etc., when $k_1 + 2k_2 + 3k_3 + \cdots = N$. Find a recurrence relation which defines these numbers $c(k_1, k_2, k_3, \ldots)$, and use it to determine a simple formula for the sum

$$S_N = \sum_{\substack{j\geq 1\\ k_1+2k_2+\cdots=N}} \binom{j}{2} k_j c(k_1, k_2, \ldots).$$

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

40. [*M33*] Find the variance of (15), the number of probes used by Algorithm C in an unsuccessful search.

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

▸ **42.** [*M20*] Derive (17).

43. [*M42*] Analyze a modification of Algorithm C that uses a table of size $M' > M$. 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 prob*
to determine the expect
probe sequence

$h(K)$, $(h(K) + p_1)$ mod

where $p_1\, p_2 \ldots p_{M-1}$ is
depends on $h(K)$. In oth
same probe sequence, and
this property are equally
This situation can be
performed on an initially
n times:

With probability p,
probability $q = 1$ -
left, with each of th
empty, occupy it; o
and occupy it, consi

For example when $m =$
ment will be (occupied, o

$\frac{7}{192}qqq + \frac{1}{6}$

(This procedure corresp
$p = 1/m$, since we can re
is 0, 1, 2, ... and all the
Find a formula for
array (i.e., z in the abov
when $p = 1/m$, $n = \alpha(m$

45. [*M48*] Solve the an
sequence begins $h_1(K)$.
randomly chosen dependi
possible choices of $M(M$
be equally likely.) Is this

46. [*M4*[illegible]] Determine C'
probe sequence

$h(K)$, 0, 1.

47. [*M25*] Find the aver
probe sequence is

$h(K)$, h

This probe sequence was
probes are distinct when

▸ **48.** [*M21*] Analyze the
$h(K)$), ..., given an infin

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 accurately modeled 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 (i.e., 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 the above 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 (i.e., 2 in the above example). Also find the asymptotic value of this quantity when $p = 1/m$, $n = \alpha(m+1)$, and $m \to \infty$.

45. [*M48*] Solve the analog of exercise 44 with *tertiary clustering*, when the probe sequence begins $h_1(K)$, $((h_1(K)+h_2(K)) \bmod M$, 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 which 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 $h_1(K)$, $h_2(K)$, $h_3(K)$, ..., given an infinite sequence of mutually independent random hash functions

$h_n(K)$. Note that it is possible to probe the same location twice, e.g. if $h_1(K) = h_2(K)$, but this is rather unlikely.

49. [*HM24*] Generalizing exercise 34 to the case of b records per bucket, determine the average number of probes (i.e., file 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*

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

which is valid for $N = \rho M$ and $k \leq \sqrt{M}$ as $M \to \infty$.

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. [*HM16*] 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. [*40*] Experiment with the behavior of Algorithm C when it has been adapted to external searching as described in the text.

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 $Q(z)/(B(z) - z^b)$, where $Q(z)$ is 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 (cf. Eq. 2.3.4.4–9 and exercise 4.7–8), reduce your answer to Eq. (61).

56. [*M48*] Generalize Theorem K, obtaining an exact analysis of linear probing with buckets of size b.

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 hashing in the sense of Theorem U.

59. [*M25*] Prove that if an assignment of probabilities to permutations is equivalent to uniform hashing, 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. [*M48*] Let us say th
uses exactly M probe sequ
of which occurs with prob
Are the best single-ha
better than the random on

61. [*M46*] Is the method
scheme? (Cf. exercise 60.)

62. [*M49*] How good ca
$p_1\,p_2 \ldots p_{M-1}$ in the not
methods are linear probing

63. [*M25*] If repeated ra
how many independent in
have become occupied at

64. [*M46*] Analyze the e
step R4 be performed, on

▶ **65.** [*20*] (*Variable-length*
that can be any number
key in the table as in the
deal with variable-length

66. [*25*] (Ole Amble.) Is
also of their numerical o
Algorithm D is known to
argument is encountered?

60. [*M48*] 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 analyzed in exercise 44?

61. [*M46*] Is the method analyzed in exercise 46 the worst possible single-hashing scheme? (Cf. exercise 60.)

62. [*M49*] How good can a single-hashing scheme be when 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.)

63. [*M25*] If repeated random insertions and deletions are made in a scatter table, how many independent insertions are needed on the average before all M locations have become occupied at one time or another?

64. [*M46*] Analyze the expected behavior of Algorithm R. How many times will step R4 be performed, on the average?

► **65.** [*20*] (*Variable-length keys.*) Many applications of scatter 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 scatter table on the MIX computer?

66. [*25*] (Ole Amble.) 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?

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

SECOND EDITION

Data Structures and Program Design

Robert L. Kruse

St. Mary's University
Halifax, Nova Scotia

Prentice-Hall, Inc.
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

KRUSE, ROBERT LEROY (date)
Data structures and program design.

Bibliography: p.
Includes index.
1. Electronic digital computers—Programming.
2. Data structures (Computer science) 3. PASCAL
(Computer program language) I. Title.
QA76.6.K77 1987 005.1 86-17081
ISBN 0-13-195884-4

Prentice-Hall Software Series
Brian W. Kernighan, adviser

Editorial/production supervision: Joan L. Stone
Interior design and page layout: Kenny Beck
Manufacturing buyer: Ed O'Dougherty

Printed in the United States of America

10 9 8 7 6 5 4

ISBN 0-13-195884-4 025

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice Hall of India Private Limited, *New Delhi*
Prentice Hall of Japan, Inc., *Tokyo*
Prentice Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*