Reducing the Retrieval Time of Scatter Storage Techniques

Richard P. Brent,
IBM Thomas J. Watson Research Center

A new method for entering and retrieving information in a hash table is described. The method is intended to be efficient if most entries are looked up several times. The expected number of probes to look up an entry, predicted theoretically and verified by Monte Carlo experiments, is considerably less than for other comparable methods if the table is nearly full. An example of a possible Fortran implementation is given.

Key Words and Phrases: address calculation, content addressing, file searching, hash addressing, hash code, linear probing, linear quotient method, scatter storage, searching, symbol table

CR Categories: 3.7, 3.73, 3.74, 4.1, 4.9

1. Introduction

Scatter storage (hash coding) techniques are used to minimize the time required to enter and retrieve information in tables. Rather similar techniques can be used for internal tables, such as the symbol tables of compilers and assemblers, and large files which are stored on random-access devices such as disks or drums. Some of these techniques are described in an excellent survey paper [5] and more recently in [1, 2, and 6].

Our aim is to describe a method for entering information so that subsequent retrievals are very efficient. Suppose that each item consists of an identifying name or *key*, which may be regarded as an integer, and an associated *value*. If m keys $k_1, \cdots, k_m$ are stored at addresses $a(k_1), \cdots, a(k_m)$ in a table T of length $n \geq m$ (i.e. $T(a(k_i)) = k_i$ for $i = 1, \cdots, m$) and a key k is given, the problem is to determine efficiently whether k is in T, and if so, to find $a(k)$. In order to compare the efficiency of different algorithms, we count the number of fetches of elements of T, *i.e. probes*, that they require.

In practical applications it usually happens that most entries in the table are looked up several times. Bell and Kaman [2] found that their hashing routine was entered 10,988 times, but with only 735 different keys, when a typical COBOL program was compiled. As a more extreme example, a table of opcode mnemonics or reserved words may be built up once and thereafter used purely for retrieval [1]. Thus it is very important to minimize the number of probes required to look up keys which are already in the table. The number of probes required to look up (and perhaps insert) keys which are not already there is not so important.

The idea of our method, which is described in detail in Section 2, is to take more care than usual when keys are inserted, in an attempt to reduce the number of probes required for subsequent lookups. Although we present the method as a modification of the "linear quotient" method of [2], the same idea could be used to modify some other methods, e.g. the "quadratic quotient" method of [1].

In Section 3 we consider the number of probes required to insert and look up entries with our method, and compare our method with other methods. The results of Monte Carlo experiments are described in Section 4. Some theoretical results are derived in Section 5, which could be skipped by the casual reader. In

Author's present address: Computer Centre, Australian National University, P.O. Box 4, Canberra, ACT 2600, Australia.

Section 6 we draw some conclusions and mention a few practical considerations. Finally, an example of a possible FORTRAN implementation is given in the Appendix.

2. The Method

Let $n \geq 3$ be a prime number, and let T be a table (i.e. an array) of length n containing m nonzero keys $k_1, \cdots, k_m$. We shall describe how keys are looked up and inserted in T. In the FORTRAN subroutine given in the Appendix, subscripts run from 1 to n, but here it is simpler to assume that they run from 0 to $n - 1$. Note that whenever keys are added to T, the values associated with the keys must also be added to another table of length n, and if keys are moved in T, the associated values must be moved appropriately.

Let k be a nonzero integer key ($k = 0$ is not allowed because 0 is reserved to denote an empty space in T). As in the linear quotient method [2], integers $r = R(k)$ and $q = Q(k)$, satisfying $0 \leq r < n$ and $0 < q < n$, are computed. Any pseudorandom functions R and Q may be used: a good choice on a machine with reasonably fast division is

$$R(k) = k \bmod n \tag{1}$$

and

$$Q(k) = (k \bmod (n - 2)) + 1. \tag{2}$$

(We divide by $n - 2$ rather than by $n - 1$ because n is odd, and as noted in [4], the parity of $k \bmod (n - 1)$ is the same as the parity of k.)

The algorithm for looking up k in T is the same as for the linear quotient method: if

$$h_s = (r + sq) \bmod n \tag{3}$$

for $s \geq 0$, then $T(h_0)$, $T(h_1)$, ... are inspected until for some s, either (a) $T(h_s) = k$, so k is found (after $p(k) = s + 1$ probes); or (b) $T(h_s) = 0$ (or $s > 0$ and $h_s = h_0$), so k is not in T. (If $s > 0$ and $h_s = h_0$, then because n is prime and $0 < q < n$, the whole table has been searched.)

Suppose that T is not full, k is not in T, and we wish to insert k. On looking up k, the above algorithm terminates with $s \geq 0$ such that $T(h_0) \neq 0, \ldots, T(h_{s-1}) \neq 0$, and $T(h_s) = 0$.

Define

$$q_i = Q(T(h_i)) \quad \text{for} \quad i \geq 0, \tag{4}$$

and

$$h_{i,j} = (h_i + jq_i) \bmod n \quad \text{for} \quad j \geq 1. \tag{5}$$

Among all i and j such that $T(h_{i,j}) = 0$, choose i and j to minimize $i + j$, and in case of a tie, to minimize i. There are two possibilities:

1. $i + j \geq s$: Insert k by setting $T(h_s) \leftarrow k$ (as in the linear quotient method).
2. $i + j < s$: Insert k by setting $T(h_{i,j}) \leftarrow T(h_i)$ and $T(h_i) \leftarrow k$, i.e. the key at h_i is moved to $h_{i,j}$ to make room for k at h_i.

Note that, once s is known, no more than $\frac{1}{2}s(s - 1)$ probes (at locations $h_{0,1}, \ldots, h_{0,s-1}, h_{1,1}, \ldots, h_{1,s-2}, \ldots, h_{s-2,1}$) are needed to determine how to insert k. Two examples, showing the relevant entries in T, are illustrated in Figure 1.

We shall describe the reason for moving the entry at h_i to $h_{i,j}$ in case 2 above. Let

$$c = \sum_{i=1}^{m} p(k_i) \tag{6}$$

be the total number of probes required to look up all of the keys $k_1, \ldots, k_m$ in T. If each entry has the same probability of being looked up, then c should be kept as low as possible, because c/m is just the expected number of probes to look up an entry in T. Thus we should add a new key k so that the resultant increase, Δ, in c is minimized. In case 1 above, $p(k)$ becomes $s + 1$, so $\Delta = s + 1$. In case 2, $p(k)$ becomes $i + 1$ and $p(T(h_i))$ increases by j, so $\Delta = i + j + 1$. Thus, to keep Δ as low as possible, we should move the entry at h_i if $i + j < s$.

3. Comparison with Other Methods

Several scatter storage methods have been proposed; see [5]. Among the best are the linear quotient method [2] and the quadratic quotient method [1]. Since the linear and quadratic quotient methods perform similarly, we shall compare our method with the linear quotient method.

If a table of length n contains m entries, then the *load factor* α is defined by $\alpha = m/(n + 1)$. If terms of order $1/n$ are neglected, then for the methods under consideration, the expected number of probes to make an entry or perform a lookup is a function of α alone. We assume that the functions Q and R are "good," i.e. that all possible pairs (q, r) occur independently and with equal probability. We are interested in $A(\alpha)$, the expected number of probes to look up an entry which is in the table, and to a lesser extent in $B(\alpha)$, the expected number of probes to look up a nonentry.

For the linear quotient method [1, 3, 5],

$$B(\alpha) = 1/(1 - \alpha) \tag{7}$$

and

$$\begin{aligned} A(\alpha) &= (1/\alpha) \int_0^{\alpha} B(\beta)\, d\beta \\ &= (1/\alpha) \log(1/(1 - \alpha)). \end{aligned} \tag{8}$$

For our method, $B(\alpha) = 1/(1 - \alpha)$ is the same as for the linear quotient method. It is more difficult to derive

Fig. 1. Two possible cases.

CASE 1. ($s = 3$)

$T(h_0) \neq 0 \rightarrow T(h_{0,1}) \neq 0 \rightarrow T(h_{0,2}) \neq 0$
$\downarrow$
$T(h_1) \neq 0 \rightarrow T(h_{1,1}) \neq 0$
$\downarrow$
$T(h_2) \neq 0$
$\downarrow$
$T(h_3) = 0$

CASE 2. ($s = 3$, $i = 0, j = 2$)

$T(h_0) \neq 0 \rightarrow T(h_{0,1}) \neq 0 \rightarrow T(h_{0,2}) = 0$
$\downarrow$
$T(h_1) \neq 0$
$\downarrow$
$T(h_2) \neq 0$
$\downarrow$
$T(h_3) = 0$

Fig. 2. The expected number of probes to look up an entry.

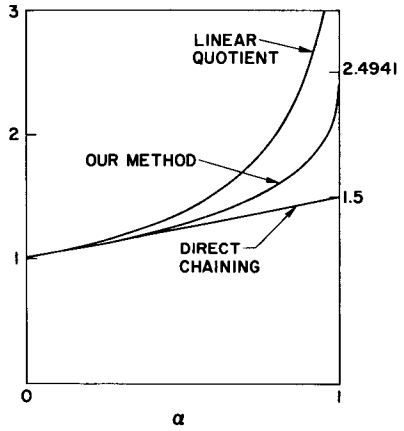
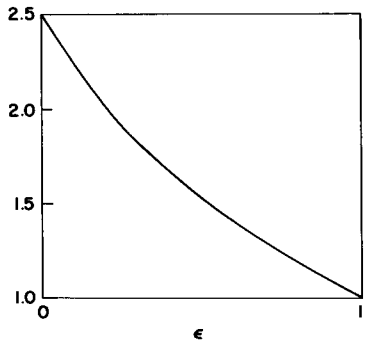


Fig. 3. The expected number of probes as a function of $\epsilon = 1 - \alpha)^{\frac{1}{2}}$.



an expression for $A(\alpha)$. If a key is inserted when the load factor is β, then the increase Δ in c will satisfy

$$\Delta > d \tag{9}$$

provided that

$$T(h_0) \neq 0,\ T(h_{0,1}) \neq 0,\ \ldots,\ T(h_{0,d-1}) \neq 0,$$
$$T(h_1) \neq 0,\ \ldots,\ T(h_{1,d-2}) \neq 0,\ \ldots,\ T(h_{d-1}) \neq 0.$$

Since the probability that any $T(h_{i,j}) \neq 0$ is β, we might conclude that (9) holds with probability $\beta^{d(d+1)/2}$. Then the expected value of Δ would be $\sum_{d=0}^{\infty} \beta^{d(d+1)/2}$, giving

$$\begin{aligned} A(\alpha) &\simeq \alpha^{-1} \int_0^{\alpha} \sum_{d=0}^{\infty} \beta^{d(d+1)/2}\, d\beta \\ &= \sum_{d=0}^{\infty} \alpha^{d(d+1)/2}/(1 + d(d+1)/2) \\ &= 1 + \alpha/2 + \alpha^3/4 + \alpha^6/7 + \cdots. \end{aligned} \tag{10}$$

However, the approximation (10) to $A(\alpha)$ is not quite correct because the probabilities $P(T(h_{i,j}) \neq 0)$ are not independent. The lack of independence is caused by our rule for inserting keys in case 2 above. In Section 5 we show that

$$\begin{aligned} A(\alpha) = 1 &+ \alpha/2 + \alpha^3/4 + \alpha^4/15 \\ &- \alpha^5/18 + 2\alpha^6/15 + \cdots, \end{aligned} \tag{11}$$

and computation shows that the approximation (10) may underestimate $A(\alpha)$ by up to 5 percent.

Figure 2 shows $A(\alpha)$ for our method, the linear quotient method, and the direct chaining method [3, 5]. It is clear that the different methods are approximately equally efficient if α is small, but our method is appreciably more efficient than the linear quotient method if α is close to one. Direct chaining has different areas of application and is not really comparable to the other two methods. With direct chaining some space is taken up by links, but no links are required for our method. The space gained by not needing links may be used to increase the table size, reducing α and $A(\alpha)$. Thus, in applications where either method may be used, our method will be more efficient than direct chaining if there is only a small amount of information associated with each key.

Our method is certainly preferable to the linear quotient method if the table is nearly full; then $A(\alpha)$ is of order $\log n$ for the linear quotient method, whereas for our method (see Section 5),

$$A(\alpha) < 2.5. \tag{12}$$

4. Some Monte Carlo Experiments

To test the theoretical results, we filled a table of length $n = 4{,}999$, using pseudorandom keys. As the table was being filled, we kept track of the total number of probes made, so the average number required to make an entry was easily computed. We also kept track of the total number of probes which would be required to

look up each entry once, so the average number needed to look up an entry could be found.

The experiment was repeated 1,000 times, and the results are summarized in Table I. $A(\alpha)$ is the expected number of probes to look up an entry for our method (as predicted in Section 5); $\hat{A}(\alpha)$ is the observed mean number of probes to look up an entry; $\lambda(\alpha) = (1/\alpha) \log (1/(1 - \alpha))$ is the expected number of probes to look up an entry for the linear quotient method; $\hat{E}(\alpha)$ is the observed mean number of probes to make an entry for our method; and

$$\hat{v}(\alpha) = (\hat{E}(\alpha) - \lambda(\alpha))/(\lambda(\alpha) - \hat{A}(\alpha)) \tag{13}$$

is the minimal number of lookups per entry required to make our method preferable to the linear quotient method (considering the expected number of probes required to insert $n\alpha$ keys and then make $\hat{v}n\alpha$ lookups).

It is clear from the table that $\hat{A}(\alpha)$ agrees very well with $A(\alpha)$. From the last column of the table, we see that our method should be more efficient than the linear quotient method if an entry is looked up three or more times on the average, and this is true in most practical applications. Monte Carlo experiments with $n = 257$ and $n = 997$ gave similar results. ($\hat{A}(\alpha)$ was slightly less than $A(\alpha)$ for the smaller values of n.)

5. Theoretical Results

In Section 3 we derived the approximation (10) for $A(\alpha)$. Here we briefly describe how a much better approximation may be found. Suppose that n is large, so terms of order $1/n$ may be neglected. In the notation of Section 2, let k be a key at position h_s in the table, and let $p_v(\alpha)$ be the probability that $T(h_{s+1}) \neq 0, \ldots, T(h_{s+v}) \neq 0$ for $v \geq 0$. To derive (10) we assumed that $p_v(\alpha) = \alpha^v$, but this is not quite correct for $v \geq 1$. If δ is small and δn new keys are inserted in the table according to the algorithm described in Section 2, then

$$\begin{aligned}(\alpha+\delta)\, p_v(\alpha+\delta) - \alpha p_v(\alpha) &= \delta\alpha^v + (\delta/(1-\alpha)) \sum_{i=0}^{v-1} \alpha^{v-i} (p_i(\alpha) - p_{i+1}(\alpha)) \\ &\quad + \delta \sum_{i=0}^{\infty} \sum_{j=1}^{v} \alpha^{v-j} P_{i,j}(\alpha) + O(\delta^2),\end{aligned} \tag{14}$$

where

$$\begin{aligned}P_{i,j} &= \alpha^{i+j+1} p_{i+j} p_{i+j-1} \\ &\quad \cdots p_{j+1}(p_{j-1} - p_j) p_{j-2} \cdots p_0\end{aligned} \tag{15}$$

is the probability that an entry is made after moving the key at h_i to $h_{i,j}$ (see Section 2, case 2). Dividing both sides of (14) by δ, and taking the limit as $\delta \to 0$, we have

$$\begin{aligned}(d/d\alpha)(\alpha p_v) &= (\alpha^v - \alpha p_v)/(1-\alpha) + \sum_{i=0}^{v-1} \alpha^{v-i} p_i \\ &\quad + \sum_{j=1}^{v} \alpha^{v-j} \sum_{i=0}^{\infty} P_{i,j}\end{aligned} \tag{16}$$

Table I. Results of Monte Carlo Experiments

α	$A(\alpha)$	$\hat{A}(\alpha)$	$\lambda(\alpha)$	$\hat{E}(\alpha)$	$\hat{v}(\alpha)$
0.20	1.1021	1.1021	1.1157	1.1548	2.85
0.40	1.2178	1.2175	1.2771	1.4337	2.62
0.60	1.3672	1.3668	1.5272	1.9248	2.48
0.80	1.5994	1.5991	2.0118	2.9713	2.32
0.90	1.8023	1.8020	2.5584	4.2740	2.26
0.95	1.9724	1.9725	3.1534	5.8382	2.26
0.99	2.2421	2.2422	4.6517	10.3922	2.36

for $v = 1, 2, \ldots$. With the initial conditions $p_v(0) = 0$ for $v = 1, 2, \ldots$, this infinite system of differential equations defines the functions $p_v(\alpha)$ (except for $p_0(\alpha) = 1$). If the sum involving $P_{i,j}$ is omitted, then the system of equations has the solution $p_v(\alpha) = \alpha^v$, which is correct for the linear quotient method.

We want to find

$$A(\alpha) = (1/\alpha) \int_0^{\alpha} F(\beta)\, d\beta, \tag{17}$$

where

$$F(\alpha) = 1 + \alpha + \alpha^2 p_1 + \alpha^3 p_1 p_2 + \cdots \tag{18}$$

is the expected increase Δ in c when a new key is inserted.

If the system of differential equations (16) is solved by numerical integration, then it is convenient to write (17) as

$$(d/d\alpha)A(\alpha) = (F(\alpha) - A(\alpha))/\alpha \tag{19}$$

and append (19) to the system of differential equations (with the initial condition $A(0) = 1$).

The system of differential equations (16) and (19) may be solved by a formal power series expansion, which gives

$$\begin{aligned}A(\alpha) &= 1 + \alpha/2 + \alpha^3/4 + \alpha^4/15 - \alpha^5/18 \\ &\quad + 2\alpha^6/15 + 9\alpha^7/80 - 293\alpha^8/5670 \\ &\quad - 319\alpha^9/5600 + \cdots.\end{aligned} \tag{20}$$

This gives a satisfactory approximation to $A(\alpha)$ unless α is close to 1. If α is close to 1, a large number of terms must be taken in (20), so it appears better to integrate the system numerically. It is worthwhile to make the change of variable

$$\alpha = 1 - \epsilon^2 \tag{21}$$

to avoid numerical difficulties because of the vertical tangent of $A(\alpha)$ at $\alpha = 1$. The function $A_e(1 - \epsilon^2)$ is quite well behaved for $\epsilon \in [0, 1]$; see Figure 3. By numerical extrapolation to $\epsilon = 0$ we find that

$$\lim_{\alpha \to 1^-} A(\alpha) \simeq 2.4941, \tag{22}$$

so the inequality (12) certainly holds for all α.

6. Conclusion

We have shown that our method compares favorably with the linear quotient method, and the difference is

considerable if the hash table is nearly full, provided that most entries are looked up several times.

So far we have not mentioned how entries may be deleted, and the theoretical results derived above are not valid if they are. Care must be taken when deleting an entry in order to ensure that other entries are still accessible. The simplest solution is to reserve a special key to denote a deleted entry, although checking for this special key increases the cost per probe. The number of probes required will also increase if the table contains many deleted entries.

Finally, we note that the algorithm described above is suitable for use with a table stored in a computer's high-speed, random-access memory. If the table is stored on a device such as a disk or drum, some modifications may be desirable. For example, suppose that several keys and their associated values can be stored on one disk track. After a probe has been made on a certain track, another probe on that track may be cheaper than a probe on a different track. Thus if a collision occurs when we are attempting to make a new entry on some track, it may be worthwhile to try to make the entry somewhere on that track before making a probe on some other track. Such a strategy would increase the expected number of probes but would decrease the number expected on different tracks. Similar considerations apply if a computer with a paged memory is used.

Appendix. A Fortran Subroutine

```
C     THIS ROUTINE WILL LOOK UP AND INSERT OR DELETE INTEGER KEYS
C     IN THE TABLE KEYTAB.  VALUES ASSOCIATED WITH THE KEYS MAY BE
C     STORED IN THE TABLE KVTAB.  THE ROUTINE IS DESIGNED TO BE
C     EFFICIENT, EVEN IF THE TABLE IS NEARLY FULL, PROVIDED MOST
C     ENTRIES ARE LOOKED UP SEVERAL TIMES.
C     GLOBAL VARIABLES:
C       KEYTAB:  AN ARRAY OF LENGTH LEN FOR THE KEYS.  IT MUST BE
C                INITIALIZED TO ALL ZEROS BEFORE THE FIRST CALL OF
C                HASH.  LEN AND LEN2 = LEN - 2 ARE SET IN A DATA
C                STATEMENT BELOW.  LEN MUST BE AN ODD PRIME.
C       KVTAB:   AN ARRAY OF LENGTH LEN FOR THE VALUES ASSOCIATED WITH
C                THE KEYS.  KEYTAB AND KVTAB ARE IN COMMON /HTABS/.
C     PARAMETERS:
C       KEY:     AN INTEGER KEY (NOT 0 OR -1, AS THESE VALUES ARE
C                RESERVED FOR EMPTY SPACES AND DELETED ENTRIES).
C       KA:      RETURNED AS THE ADDRESS OF THE KEY IN KEYTAB, OR ZERO.
C       FOUND:   A FLAG RETURNED TRUE IFF THE KEY WAS ALREADY IN THE
C                TABLE.
C       MODE:    AN INDICATOR:
C         LOOKUP ONLY (MODE = 1)
C                IF THE KEY IS IN THE TABLE, THE ADDRESS KA IS
C                RETURNED (THE ASSOCIATED VALUE IS AT KVTAB(KA)).
C                OTHERWISE KA = 0 IS RETURNED.
C         LOOKUP AND ENTER (MODE = 2)
C                IF THE KEY IS IN THE TABLE, THE ADDRESS KA IS
C                RETURNED. OTHERWISE THE KEY IS ENTERED AT KEYTAB(KA).
C                THE CALLING PROGRAM MUST ENTER THE ASSOCIATED VALUE
C                AT KVTAB(KA).  IF AN ENTRY CAN NOT BE MADE BECAUSE
C                THE TABLE IS FULL (OR BECAUSE KEY = 0 OR -1),
C                KA = 0 IS RETURNED.
C         LOOKUP AND DELETE (MODE = 3)
C                IF THE KEY IS IN THE TABLE, IT IS DELETED AND ITS
C                FORMER ADDRESS KA IS RETURNED.  IF THE KEY IS NOT
C                THERE, KA = 0 IS RETURNED.  (THE SUBROUTINE CAN BE
C                SIMPLIFIED CONSIDERABLY IF KEYS ARE NEVER DELETED.)
      SUBROUTINE HASH (KEY, MODE, KA, FOUND)
      LOGICAL FOUND, DEL
C     THE NEXT 3 CARDS MUST BE CHANGED IF THE TABLE LENGTH IS NOT 4999.
      COMMON /HTABS/ KEYTAB(4999), KVTAB(4999)
      DATA LEN /4999/
      DATA LEN2 /4997/
      IC = -1
C     COMPUTE ADDRESS OF FIRST PROBE (IR) AND INCREMENT (IQ).
C     ANY INDEPENDENT PSEUDO-RANDOM FUNCTIONS OF THE KEY MAY BE USED,
C     PROVIDED  0 < IQ < LEN  AND  0 < IR <= LEN
      IQ = MOD(IABS(KEY), LEN2) + 1
      IR = MOD(IABS(KEY), LEN)  + 1
      KA = IR
C     LOOK IN THE TABLE.
   20 KT = KEYTAB(KA)
C     CHECK FOR AN EMPTY SPACE, A DELETED ENTRY, OR A MATCH.
      IF (KT.EQ.0) GO TO 30
      IF (KT.EQ.-1) GO TO 40
      IF (KT.EQ.KEY) GO TO 60
      IC = IC + 1
C     COMPUTE ADDRESS OF NEXT PROBE.
      KA = KA + IQ
      IF (KA.GT.LEN) KA = KA - LEN
C     SEE IF WHOLE TABLE HAS BEEN SEARCHED.
      IF (KA.NE.IR) GO TO 20
C     THE KEY IS NOT IN THE TABLE.
   30 FOUND = .FALSE.
C     RETURN WITH KA = 0 UNLESS AN ENTRY HAS TO BE MADE.
      IF ((MODE.EQ.2).AND.(IC.LE.LEN2).AND.(KEY.NE.0).AND.(KEY.NE.-1))
     -  GO TO 70
      KA = 0
      RETURN
C     A DELETED ENTRY HAS BEEN FOUND.
   40 IA = KA
C     COMPUTE ADDRESS OF NEXT PROBE.
   50 IA = IA + IQ
      IF (IA.GT.LEN) IA = IA - LEN
      IS = KEYTAB(IA)
C     CHECK FOR AN EMPTY SPACE OR A COMPLETE SCAN OF THE TABLE.
      IF ((IS.EQ.0).OR.(IA.EQ.IR)) GO TO 30
C     CHECK FOR A MISMATCH OR DELETED ENTRY.
      IF ((IS.NE.KEY).OR.(IS.EQ.-1)) GO TO 50
C     KEY FOUND.  MOVE IT AND THE ASSOCIATED VALUE TO SAVE PROBES
C     ON THE NEXT SEARCH FOR THE SAME KEY.
      KVTAB(KA) = KVTAB(IA)
      KEYTAB(KA) = IS
      KEYTAB(IA) = -1
C     THE KEY IS IN THE TABLE.
   60 FOUND = .TRUE.
C     DELETE IT IF MODE = 3.
      IF (MODE.EQ.3) KEYTAB(KA) = -1
      RETURN
C     LOOK FOR THE BEST WAY TO MAKE AN ENTRY.
   70 IF (IC.LE.0) GO TO 120
C     SET DEL IF A DELETED ENTRY HAS BEEN FOUND.
      DEL = KT.NE.0
      IA = KA
      IS = 0
C     COMPUTE THE MAXIMUM LENGTH TO SEARCH ALONG CURRENT CHAIN.
   80 IX = IC - IS
C     COMPUTE INCREMENT JQ FOR CURRENT CHAIN.
      JQ = MOD(IABS(KEYTAB(IR)), LEN2) + 1
      JR = IR
C     LOOK ALONG THE CHAIN.
   90 JR = JR + JQ
      IF (JR.GT.LEN) JR = JR - LEN
      KT = KEYTAB(JR)
C     CHECK FOR A HOLE (AN EMPTY SPACE OR A DELETED ENTRY).
      IF ((KT.EQ.0).OR.(KT.EQ.-1)) GO TO 100
      IX = IX - 1
      IF (IX.GT.0) GO TO 90
      GO TO 110
C     SKIP IF THIS IS AN EMPTY SPACE AND A DELETED ENTRY HAS
C     ALREADY BEEN FOUND.
  100 IF (DEL.AND.(KT.EQ.0)) GO TO 110
C     CHECK FOR A DELETED ENTRY.
      IF (KT.NE.0) DEL = .TRUE.
C     SAVE LOCATION OF HOLE.
      IA = JR
      KA = IR
      IC = IC - IX
C     MOVE DOWN TO THE NEXT CHAIN.
  110 IS = IS + 1
      IR = IR + IQ
      IF (IR.GT.LEN) IR = IR - LEN
C     GO BACK IF A BETTER HOLE MIGHT STILL BE FOUND.
      IF (IC.GT.IS) GO TO 80
C     SKIP IF THERE IS NOTHING TO MOVE.
      IF (IA.EQ.KA) GO TO 120
C     MOVE AN OLD ENTRY AND ITS ASSOCIATED VALUE TO MAKE ROOM FOR
C     THE NEW ENTRY.
      KVTAB(IA) = KVTAB(KA)
      KEYTAB(IA) = KEYTAB(KA)
C     ENTER THE NEW KEY, BUT NOT ITS ASSOCIATED VALUE.
  120 KEYTAB(KA) = KEY
      RETURN
      END
```

Received November 1971; revised January 1972

References

1. Bell, J.R. The quadratic quotient method: a hash code eliminating secondary clustering. *Comm. ACM 13*, 2 (Feb. 1970), 107–109.
2. Bell, J.R., and Kaman, C.H. The linear quotient hash code. *Comm. ACM 13*, 11 (Nov. 1970), 675–677.
3. Johnson, L.R. An indirect chaining method for addressing on secondary keys. *Comm. ACM 4*, 5 (May 1961), 218–222.
4. Maurer, W.D. An improved hash code for scatter storage. *Comm. ACM 11*, 1 (Jan. 1968), 35–38.
5. Morris, R. Scatter storage techniques. *Comm. ACM 11*, 1 (Jan. 1968), 38–44.
6. Radke, C.E. The use of quadratic residue research. *Comm. ACM 13*, 2 (Feb. 1970), 103–105.