The BXL works like a TIX if the second operand (in this case 3) is odd; the BXH with an even second argument allows the increment and comparand to be in adjacent registers, in this case 4 and 5. The SR subtracts register 5 from register 2. C is a full word comparison which sets the condition code; this example could be modified for a halfword table by changing C to CH and "4" to "2" throughout the instructions above. BC branches on condition code 6, which is 0110 in binary—i.e., first operand low or first operand high.

The timing in this example depends on the model. On the Mod 30, the timings in microseconds of the given instructions are: BXL, 38; BXH, 38; SR, 30; C, 30; BC, 15. The total is 151 microseconds. A *single* multiply instruction on the Mod 30 takes 235 microseconds. On other models the situation is less clear, but in any event at least five instructions are needed: multiply, mask, BXH to TFULL, C, and BC. Note that on this machine we cannot "enclose" a BXH within a comparison because the C instruction does not skip, but merely sets the condition code.

Scatter Storage Techniques

ROBERT MORRIS
Bell Telephone Laboratories, Murray Hill, N.J.

Scatter storage techniques as a method for implementing the symbol tables of assemblers and compilers are reviewed and a number of ways of using them more effectively are presented. Many of the most useful variants of the techniques are documented.

KEY WORDS AND PHRASES: scatter storage, hash addressing, searching, file searching, file addressing, storage layout
CR CATEGORIES: 3.73, 3.74

Introduction

This paper is primarily concerned with the application of scatter storage techniques to internal tables such as compiler and assembler symbol tables. Most of the previous literature on the subject of scatter storage has been aimed at the problems of addressing random access secondary storage. The techniques can be applied profitably to any table or file in which access is to be made to the entries in unpredictable order and the items are identified by some key associated with their contents.

Consider the simple case of an assembler symbol table. Names which appear in location fields during assembly are given a value which is the value of the location counter current at the time the location field is encountered. This value must be consulted whenever the same name appears elsewhere in the program. The location counter values are accessed in unpredictable order and therefore must be searched for at every access; each entry is associated with a name which is used as a key for finding the entry. The table could of course be kept in alphabetical order, but the best search technique ir a sorted table requires at least an average of $\log_2 N$ probes to find an item, where N is the size of the table. Thus, for example, a table with 1024 entries would require an average of at least 10 probes to find an item by searching. One can do much better than this with a scatter storage table.

A more important disadvantage of using a sorted table in this application is that if items need to be looked up before all of the entries are made, then either

—the table must be kept sorted after each entry is made, with the resulting heavy overhead for making each entry,
or
—the items must be looked up in an unsorted table until all of the entries are made, at the cost of a tremendous number of probes to find an item.

Hash Addressing

The fundamental idea behind scatter storage is that the key associated with the desired entry is used to locate the entry in storage. Some transformation is performed on the key (the name, in the example above) to produce an address in the table to hold the key and the entry associated with the key. A good transformation is one that spreads the calculated addresses (sometimes called hash addresses) uniformly across the available addresses. If the calculated address is already filled with some other key and its entry because two keys happened to be transformed into the same calculated address, a method is needed for resolving the collision of keys, and we will discuss several such methods in what follows.

If the keys are names or other objects that fit into a single machine word, a popular method of generating a hash address from the key is to choose some bits from the middle of the square of the key— enough bits to be used as an index to address any item in the table. Since the value of the middle bits of the square depends on all of the bits of the key, we can expect that different keys will give rise to different hash addresses with high probability, more or less independently of whether the keys share some com-

mon feature, say all beginning or all ending with the same bit pattern.

If the keys are multiword items, then some bits from the product of the words making up the key may be satisfactory as long as care is taken that the calculated address does not turn out to be zero most of the time. The most dangerous situation in this respect is when blanks are coded internally as zeros or when partial word items are padded to full word length with zeros.

A third method of computing a hash address is to cut the key up into n-bit sections, where n is the number of bits needed for the hash address, and then to form the sum of all of these sections. The low-order n bits of the sum is used as the hash address. This method can be used for single-word keys as well as for multiword keys. A program using this method appears in Appendix A.

All three of these methods of computing hash addresses have been in use for years with satisfactory results, but in a new application or on a new computing machine, care must still be taken that the computed addresses are spread uniformly over the available table space.

Handling Collisions

Once some entries have been made into a scatter storage table, it becomes possible for the computed addresses of different keys to be the same, causing a collision between the storage locations allocated to each. Some other place must be found for one of the items. We will initially assume that once an item has been entered it is never moved or deleted. So another potential place must be found for the new entry. In general, when the table is nearly full, many collisions may occur while probing the table for an empty slot. Hence some procedure is needed which generates additional calculated addresses until an empty slot is found, probing the entire table if necessary. Of course, the same procedure for generating additional calculated addresses must be used when the item is later looked up.

In practice, when a scatter storage table routine is called, it is not necessary to specify whether an item is being entered or being looked up. What is required of the routine is to determine the address at which the offered key belongs and to report whether the key was already entered. Then the calling routine can make the entry or extract the information, as appropriate. The procedure, then, will be to generate successive hash addresses until encountering either a slot that contains the desired key or an empty slot. In the latter case, the key is entered in the empty slot, if it is entered at all.

The possibility that several keys can generate the same calculated address means that the key *must* be stored in the table along with its associated entry. For example, when numbers are stored in an array, the index in the array uniquely specifies the storage allocated to an item. In an application (such as a sparse matrix) where it is known that very few elements in the array will actually be used, one might advantageously use the subscript combination as a key and compute a hash address from the values of the subscripts. The item can then be accessed through the hash address. But then the subscripts associated with the number must be stored along with the number itself. For example, if $A(10)$ were equal to 1.5, then the corresponding table entry would most likely be a two-word item, one word containing the subscript 10, and the other word containing the number 1.5. This is a powerful but little-known technique for handling sparse arrays. It can greatly reduce the storage requirements at the cost of increased programming complexity and longer access time. Observe that this is a useful method for storing arrays only when array elements are to be accessed in unpredictable order; it would not be a good method if matrix operations such as addition or multiplication were to be performed on the arrays.

Many methods of resolving collisions have been suggested and used, and the particular method to be used in a particular application should be chosen carefully since the method of handling collisions profoundly affects the efficiency of the technique and the difficulty of the programming task.

Table Layout

In almost every application, an entry in a scatter storage table occupies more than one word, typically at least one word for the key and at least one word for the data associated with the key. If an entry occupies k words and space is required for N entries, then altogether kN words of storage are needed. Two straightforward ways of arranging the table are:

—to put each entry into k consecutive words and use only hash addresses that point to the first word of an entry. This can readily be done by multiplying each hash address by k and using the product as an index in the table.

—to divide the table into k sections, each with N words. Then the hash address is used as an index in the first table to find the first word of an entry and as an index in the second table to find the second word of an entry, and so forth.

The only difference between these two methods is one of programming convenience. In either case, the hash address is used as an index in the table and not as an absolute storage address.

Since most hash-addressing schemes produce a number of random bits to be used as an index, it is most convenient to use only values of N which are powers of 2. For instance, if the hash transformation generates l bits, one can use the l bits as an index in a table with space for $N = 2^l$ entries. Other considerations apply to storage that does not have binary addresses.

There must be some way of distinguishing an empty slot in the table from any possible valid entry. For instance, if zero can be excluded as a valid key, then a zero key can be used to mean that an entry is empty. Of course the entire table must be initialized to whatever state is being used to signal empty slots in the table.

Random Probing

An efficient and elegant method of generating successive calculated addressses to handle collisions is as follows:

1. Calculate an address l in the table by using some transformation on the key as an index.
2. If the item is already at this address or if the place is empty, the job is done.
3. If some other key is there, call a pseudorandom number generator for an integer offset ρ. Make the next probe at $l + \rho$ and go to step (2).

The pseudorandom number generator can be of the simplest sort and usually can be written in less than six machine instructions. It must generate every integer from 1 to $N - 1$ (where N is the size of the table) exactly once. When the generator runs out of integers, the table is full and the entry cannot be made.

The following will serve the purpose as a random number generator for tables whose size is $N = 2^n$, a power of two.

Initialize an integer R to be equal to 1 every time the tabling routine is called, and then on each successive call for a random number:

—set $R = R * 5$,
—mask out all but the low-order $n + 2$ bits of the product and place the result in R,
—set $\rho = R/4$ and return.

A scatter storage routine using this technique of handling collisions is given in Appendix A. The important property of the pseudorandom number generator in this application is that for every value of i, the numbers, $\rho_{i+k} - \rho_i$ for $i \leq i + k \leq n - 1$, are all different, where ρ_j is the jth random number which is generated.

The efficiency of the random probing method is best expressed in terms of the average number E of probes necessary to retrieve an item in the table. This happens to be equal to the average number of probes which were required to enter the items originally. The number E depends on the fraction α of the table which is occupied but not on the size of the table. If N is the size of the table, and k items are in the table, then $\alpha = k/N$. The expected number A of probes necessary to enter the $(k + 1)$-st item, including the final probe, is

$$A = 1 + \frac{k}{N} + \frac{k(k-1)}{N(N-1)} + \cdots + \frac{k(k-1)\cdots(1)}{N(N-1)\cdots(N-k+1)} \tag{1}$$

where the jth term in the sum is the probability that j or more probes are needed to enter the item. By induction on k, (1) can be rewritten as

$$A = 1 + \frac{k}{N-k+1} = 1 \Big/ \left(1 - \frac{k}{N+1}\right) \tag{2}$$

For large values of N, we can replace $k/(N + 1)$ by α in (2) and approximate A by

$$A = 1/(1 - \alpha) \tag{3}$$

and then E is equal to the average of A for values of k from 0 to $k - 1$. We can approximate this average by the integral

$$E = \frac{N}{k}\int_0^\alpha \frac{dx}{1-x} = -(1/\alpha)\log(1-\alpha) \tag{4}$$

The approximations that were made insure that the actual expected performance is slightly better than is predicted by (4). Some sample values of E for various values of α are:

Load factor α	E
0.1	1.05
0.5	1.39
0.75	1.83
0.9	2.56

The number of probes necessary for searching a sorted table is greater than this for all but the tiniest tables, but to make a fair comparison, one must include the extra time it takes to compute a hash address. This method of handling collisions is a variant of a method due to Vyssotsky [1].

Deletion of entries made using this scheme is a troublesome process. One cannot simply mark an entry as empty in order to delete it because other entries may have collided at that place and they would become unreachable. The hash addresses for every entry in the table would have to be recomputed and some of them moved in order to close up the gap caused by the deleted entry. A much more convenient method of deletion is to reserve a special signal for a deleted entry (much like the special signal for an empty item). On searching for a key, the search continues if a deleted entry is encountered. A new item can be installed in place of any deleted entry encountered in searching for its proper place. The disadvantage of this method is that the lookup time is not reduced when entries are deleted—only the lost space is reclaimed.

Linear Probing

The first method of generating successive calculated addresses to be suggested in the literature [2] was simply to place colliding entries as near as possible to their nominally allocated position, in the following sense. Upon collision, search forward from the nominal position (the initial calculated address), until either the desired entry is found or an empty space is encountered—searching circularly past the end of the table to the beginning, if necessary. If an empty space is encountered, that space becomes the home for the new entry.

This procedure is the *least* effective strategy in common use for resolving collisions in terms of the average number of probes required to retrieve an item. The reason for its relatively poor efficiency is that after a few collisions have

been resolved in this way, the entries are clumped in such a way that, given that a collision has just occurred at location l, the probability of a collision at location $l + 1$ is higher than the average probability over the whole table.

The efficiency of the linear probing method can be analyzed by techniques similar to those used in [3] to evaluate a related method. The result is that, to within suitable approximation, the average number E of probes necessary to look up an item in the table is

$$E = (1 - \alpha/2)/(1 - \alpha). \quad (5)$$

Sample values of E are:

Load factor α	E
0.1	1.06
0.5	1.50
0.75	2.50
0.9	5.50

For values of α greater than 0.5, the random probing method is distinctly superior. On the other hand, the linear probing method is easier to program, and each probe beyond the first requires less computation.

The problem of deleting entries made in the linear probing method is similar to the problem for random probing. However, in order to close the gap caused by the deleted entry, one need consider only those entries between the deleted entry and the next empty place in the table; no other entries could have collided at that spot. If a special mark is used for a deleted entry, then the lookup time can be reduced as a result of the deletion by looking at the next entry in the table. If that entry is empty, then the deleted entry can also be marked as empty. Moreover, the table can then be scanned backwards from that point and every deleted entry marked as empty until an entry is encountered that is not marked as deleted.

Direct Chaining

At a small penalty in space, another method of resolving collisions, called direct chaining [4], is considerably more efficient in terms of number of probes per entry than either of the preceding methods. In this technique, part of one of the words in each entry is reserved as a pointer to indicate where additional entries with the same calculated address are to be found, if there are any. Thus all of the entries with the same calculated address are to be found on a linked list (or chain) starting at that address. The last entry on each chain must be distinguished in some way (such as having a zero pointer).

When a key is to be looked up, its hash address is computed and then

—if that address is empty, the key has not been entered.

—if that address is occupied, search down the chain hanging from that address; if the key is not encountered, it is not in the table.

When a new entry is to be made, compute its hash address and then

—if that address is empty, install the item there.

—if that address is occupied by the head of a chain, find an unallocated cell in the scatter table by any technique whatever, and place the new entry in the newly allocated cell. Then insert the new entry into the chain hanging from the calculated address.

—if that address is occupied by an entry which is not the head of a chain, i.e., by an entry which is not at its own calculated address, then the old entry must be moved to another slot and the new entry inserted in its place. Moving the old entry requires finding an empty slot for it, placing it there, and updating the chain it is on.

The principal disadvantage of this method is that entries must be moved in storage, with all the programming complexity that this implies. Observe that when a newly entered item is to be placed on a chain, it is usually more profitable to place it near the head of its chain rather than at the end of the chain.

An attractive feature of this method is that when the table fills up, new items can be placed in an overflow area with no change in the strategy of making entries or looking them up. Moreover, as we shall see, the efficiency of the method is still quite good even after overflow has occurred.

The average number E of probes necessary to find an item using this scheme is calculated in [4], and is

$$E = 1 + \alpha/2 \quad (6)$$

where, as usual, $\alpha = k/N$, k = number of entries, and N = number of slots in the table. The formula (6) is still valid when $\alpha > 1$, as will happen when items are placed in an overflow table. The efficiency of lookup depends in no way on how cells are allocated to items on collision. Some sample values of efficiency are:

Load factor α	E
0.1	1.05
0.5	1.25
0.75	1.38
0.9	1.45
1.5	1.75
2.0	2.00

Formula (6) does not supply a fair comparison with other methods for values of α greater than 1, since then space is being used outside of the scatter table, but it is not easy to say how the comparison should be made in this case.

A variation on this method [4] decreases programming complexity at only slight cost in storage efficiency. The variation is to treat every item which collides at a calculated address as an overflow item to be stored in an overflow table rather than in the scatter table itself. This implies that items need never be moved in storage once they are entered, but space is used in the overflow area before the scatter table is itself full. The only inefficiency caused by this variation is storage inefficiency as more space is required in the overflow table. However, since there is no

chaining through the scatter table, the scatter table can be smaller. Usually it is the scatter table which has rigid storage requirements while the overflow table can be gotten from any general purpose storage allocator, which may well be present for other uses.

The expected amount of overflow space used can be calculated as follows: A good approximation [5, p. 110] to the number of occupied slots in the table after k items have been entered is $N(1 - e^{-\alpha})$ where α, k, and N have their previous meanings. But then, since the total of k items have been entered, the number which have overflowed is $k - N(1 - e^{-\alpha})$. Therefore the total (expected) amount of storage that has been used is

$$N(\alpha + e^{-\alpha}). \tag{7}$$

This factor must be taken into account when the two chaining methods are compared.

The technique for deleting items entered by chaining is as follows:

—an entry not stored at its calculated address may be marked empty and its former chain joined around it.
—an entry stored at its calculated address, but with no chain hanging from it, may merely be marked empty.
—an entry stored at its calculated address with a chain hanging from it must either be marked as deleted or one of the items on its chain must be moved to the calculated address and the chain properly fixed up.

Scatter Index Tables

It is possible to extend the idea of treating every colliding item as an overflow item, namely, to place all entries in a separate storage area in which space is allocated for entries only as needed. Then the scatter table consists only of pointers to the entries. We will call such a table a scatter index table to emphasize that the scatter table contains not the entries themselves, but only pointers to the entries. It is possible to combine the idea of a scatter index table with any of the three methods of resolving collisions, but there is seldom reason to use anything but the chaining method. Once the programming arrangements have been made to allocate space as needed from a free storage area, the chaining method is just as easy to program as the other methods, and it is more efficient.

There are a number of advantages to keeping all of the data items in a storage area separate from the scatter table. A data item often occupies many words, whereas a pointer usually occupies only a part of a single word, so the unused portion of the scatter table will consist of single word items rather than the multiword spaces suitable for data items. Because of this, much of the efficiency of the chaining method can be regained without having to move entries around in storage.

Only so much space need be allocated for each entry as is necessary to hold it. If entries are kept in the scatter table, one would have to allocate for each item as much space as is needed for the largest of them. Moreover, it may well not be known at the time a key is first encountered how much space will be required to hold the associated entry.

Deletion of entries is trivial. An item to be deleted is simply removed from the chain it is on and returned to free storage.

Appendix B is a program which uses the method of scatter index tables.

Virtual Scatter Tables

The time spent searching for an item in a table depends not only on the number of probes needed to find the item, but also on the time required to make a single probe. If the keys are at all complex, for instance, if they are character strings of varying length, it may take a considerable amount of time to find out whether or not two keys are the same.

For some applications, an attractive method of decreasing the amount of time needed to make a probe is to compute a hash address suitable for a much larger scatter table than is actually being used and then to place the extra bits in the entry. For example, if a 1024-word scatter table is being used and a 20-bit hash address is computed, then 10 of the bits can be used to address the table entry and the remaining 10 bits placed in the entry. Then the table can be made to act in some respects as if it were a very lightly loaded scatter table with 2^{20} slots.

When an entry is being looked up, compute its hash address as usual and follow whatever collision doctrine has been chosen, but instead of comparing keys, first compare the extra hash bits just computed with the extra hash bits stored in the entries. Only if the extra hash bits are the same, need the keys be compared. In the example above, where twice as many bits were computed as were needed, one could expect more often than not that by the time the table was filled, no two keys were ever compared which turned out to be different. The probability of two keys having the same hash address and the same extra hash bits is precisely the same as the probability that they would collide in the larger virtual table.

The expected total number of probes needed to enter 2^{19} items into a table with 2^{20} slots is equal to $2^{19} + \frac{1}{2}$ because $\alpha = 2^{-1}$ and for very lightly loaded tables each of the collision doctrines gives an expected average number of probes $E = 1 + \alpha/2$.

A second application of the idea of virtual scatter tables is that it permits a scatter table to be increased in size during execution without rehashing all of the keys, indeed, without rehashing any of the keys. In order to double the size of a scatter table, a single pass is made through the old table and entries are placed in the lower or upper half of the new table depending on whether their lowest order extra hash bit is 0 or 1. If collisions were resolved by the linear method or by the random method, then the original hash bits need to be kept in addition to the extra hash bits so that the original calculated address can be recovered. If collisions were resolved by chaining, then this is unnecessary. Of course, when the scatter table is

used as an index, no data items need be moved, only pointers.

A curious possible use of virtual scatter tables arises when a hash address can be computed with more than about three times as many bits as are actually needed for a calculated address. The possibility that two different keys have the same virtual hash address becomes so remote that the keys might not need to be examined at all. If a new key has the same virtual hash address as an existing entry, then the keys could be assumed to be the same. Then, of course, there is no longer any need to keep the keys in the entry; unless they are needed for some other purpose, they can just be thrown away. Typically, years could go by without encountering two keys in the same program with the same virtual hash address. Of course, one would have to be quite certain that the hash addresses were uniformly spread over the available virtual addresses. No one, to the author's knowledge, has ever implemented this idea, and if anyone has, he might well not admit it.

The most important application of virtual scatter tables is discussed in the next section.

Scatter Tables on Paged Machines

On some machines it is possible for a program to address more storage than is actually available to the program in the fast storage of the machine. When an item is referenced by the program and the item is not already in fast storage, the block of data (called a page) which contains the item must be brought into fast storage from secondary storage. Further execution of the program is delayed until the input operation is completed. On such a machine, a scatter table can be defined whose size exceeds the amount of fast storage available to the program, so that every new access to the scatter table might cause an input operation to occur. Slow execution would result.

In such cases, it is most efficient to choose means of accessing entries which ensure that consecutive references to storage are as often as possible in pages that have recently been referenced and thus are likely to be already in fast storage.

If the entries themselves are kept in the scatter table, then the linear probing method becomes more attractive because consecutive probes are highly likely to be on the same page. (Page sizes are most often in the range from 2^6 to 2^{12} words.) It may well turn out that two probes on different pages are more expensive of time than a dozen probes all on the same page. This consideration can be neglected if it is known that all the required pages will remain in fast storage.

For a really large scatter table, where it is unlikely or impossible that the whole table can be held in fast storage, it would almost certainly be most efficient to use a scatter index table and keep extra hash bits along with the pointer in the index table. Also, collisions should be resolved within the index and not by chaining through free storage. Since the index table consists of single-word items, many more of its pages can be kept in fast storage—possibly all of them. Then the program stands the chance of needing a new page only when it tries to access the entry itself. By keeping enough extra hash bits in the index, the probability of accessing an unwanted entry can be reduced as much as desired. Then the risk of bringing a new page into fast storage occurs only once for each item referenced.

It would be wasteful of time to define a very large scatter table on a paged machine and then to use either random probing or chaining through free storage to resolve collisions. Either method would result in the risk of bringing a new page into fast storage for every probe.

REFERENCES

1. McIlroy, M. D. A variant method of file searching. *Comm. ACM 6* (Jan. 1963), 101.
2. Peterson, W. W. Addressing for random-access storage. *IBM J. Res. Dev. 1* (1957), 130–146.
3. Schay, G., and Spruth, W. G. Analysis of a file addressing method. *Comm. ACM 5* (Aug. 1962), 459–462.
4. Johnson, L. R. Indirect chaining method for addressing on secondary keys. *Comm. ACM 4* (May 1961), 218–222.
5. Feller, W. *An Introduction to Probability Theory and Its Applications, Vol. 1*. John Wiley & Sons, New York, 1950.

Appendix A. FORTRAN IV Scatter Storage Program—Random Method

```
C  PROGRAM TO LOOKUP AND ENTER DATA IN A SCATTER TABLE.
C  THE TABLE IS SEARCHED ACCORDING TO A GIVEN KEY.
C  IF SUCCESSFUL, AN ASSOCIATED VALUE IS RETURNED.
C  A LOGICAL VARIABLE IS SET TO .TRUE. OR .FALSE. ACCORDING
C  TO WHETHER THE SEARCH WAS SUCCESSFUL OR NOT.
C  IN EITHER CASE, THE KEY AND AN ASSOCIATED VALUE
C  MAY BE INSTALLED IN THE TABLE BY CALLING INSTAL.
C  A WORD OF ALL ZEROES MAY NOT BE USED AS A KEY.
C  NO PROVISION IS MADE FOR DELETION OF ENTRIES.
C  THE TABLE SIZE MUST BE A POWER OF TWO.

C  MEANING OF SYMBOLS

C       KEYS     = TABLE OF KEYS ENTERED
C       VALUES(J) = VALUE ASSOCIATED WITH KEYS(J)
C       KEY      = KEY FOR CURRENT CALL
C       VALUE    = ASSOCIATED VALUE FOUND OR TO BE ENTERED
C       KEYSAV   = KEY USED IN MOST RECENT CALL TO LOOKUP
C       FOUND    = .TRUE. IF KEY WAS FOUND
C       FIRST    = .TRUE. BEFORE FIRST CALL TO LOOKUP
C       KPLACE   = CURRENT INDEX FOR TABLE ENTRY
C       KRAND/4  = CURRENT PSEUDO-RANDOM OFFSET
C       IHASH    = HASH ADDRESS FOR CURRENT KEY
C       N        = NUMBER OF BITS IN HASH ADDRESS
C       WDSIZE   = NUMBER OF BITS IN MACHINE WORD

      SUBROUTINE LOOKUP(KEY,FOUND,VALUE)
      LOGICAL FIRST, FOUND
      INTEGER WDSIZE
      COMMON KEYS, VALUES, KEYSAV, KPLACE
      DATA FIRST /.TRUE./
      DATA WDSIZE /36/

C  THE FOLLOWING TWO CARDS MUST BE CHANGED
C  TO CHANGE THE TABLE SIZE.

      DIMENSION KEYS(1024), VALUES(1024)
      DATA N /10/

      IF(FIRST) GO TO 91
1     IF(KEY .EQ. 0) GO TO 99
      KEYSAV = KEY

C  USE AS A HASH ADDRESS THE PRODUCT OF THE KEY WITH AN
C  APPROPRIATE MULTIPLIER.

      KRAND = 1
      IHASH = 0
      KEYA = IABS(KEY)
      DO 11 I=1,WDSIZE,N
11    IHASH = IHASH + KEYA/(2**(I-1))
```

(*Please turn the page*)

```
C  LOOK AT THE INDICATED PLACE IN THE TABLE
C  TO FIND OUT IF IT IS
C       - EMPTY
C       - OCCUPIED BY THIS KEY
C       - OCCUPIED BY ANOTHER KEY, SO WE MUST LOOK FURTHER.

21      KPLACE = MOD(IHASH+KRAND/4,2**N) + 1
        IF(KEYS(KPLACE) .EQ. KEY) GO TO 31
        IF(KEYS(KPLACE) .EQ. 0) GO TO 41
        KRAND = MOD(5*KRAND,2**(N+2))
        IF(KRAND .EQ. 1) GO TO 99
        GO TO 21

31      FOUND = .TRUE.
        VALUE = VALUES(KPLACE)
        RETURN

41      FOUND = .FALSE.
        RETURN

91      K = 2**N
        DO 92 I=1,K
92      KEYS(I) = 0
        FIRST = .FALSE.
        GO TO 1

99      CALL ERROR
        STOP

        END
        SUBROUTINE INSTAL(KEY,VALUE)
        COMMON KEYS, VALUES, KEYSAV, KPLACE

C  THE FOLLOWING CARD MUST BE CHANGED TO CHANGE THE
C  TABLE SIZE.

        DIMENSION KEYS(1024), VALUES(1024)

        IF(KEY .NE. KEYSAV) GO TO 99
        KEYS(KPLACE) = KEYSAV
        VALUES(KPLACE) = VALUE
        RETURN

99      CALL ERROR
        STOP

        END
```

Appendix B. FORTRAN IV Scatter Index Table Program

```
PROGRAM TO LOOKUP, ENTER, AND DELETE DATA IN A
SCATTER TABLE.  THE TABLE IS SEARCHED ACCORDING TO
A GIVEN KEY.  IF SUCCESSFUL, AN ASSOCIATED VALUE IS
RETURNED.  A LOGICAL VARIABLE IS SET TO .TRUE. OR
.FALSE. ACCORDING TO WHETHER THE SEARCH WAS
SUCCESSFUL OR NOT.  IN EITHER CASE, THE KEY AND AN
ASSOCIATED VALUE MAY BE INSTALLED IN THE TABLE BY
CALLING INSTAL.  THE KEY MAY BE DELETED FROM THE
TABLE BY CALLING DELETE.

ACCESS TO ENTRIES IS BY A HASH ADDRESS IN A SCATTER
INDEX TABLE.  THE ENTRIES ARE THREE WORD ITEMS
ALLOCATED AS NEEDED FROM A FREE STORAGE LIST.
THE FIRST WORD OF EACH ENTRY IS A POINTER TO
ADDITIONAL ITEMS WITH THE SAME HASH ADDRESS, IF
THERE ARE ANY.  THE FIRST WORD IS OTHERWISE ZERO.
THE SECOND WORD OF EACH ENTRY HOLDS THE KEY AND
THE THIRD WORD HOLDS THE ASSOCIATED VALUE.
THE SIZE OF THE INDEX TABLE MUST BE A POWER OF TWO.

MEANING OF SYMBOLS

    KEY     = KEY FOR CURRENT CALL
    VALUE   = ASSOCIATED VALUE FOUND OR TO BE ENTERED
    FOUND   = .TRUE. IF KEY WAS FOUND
    FIRST   = .TRUE. BEFORE FIRST CALL TO LOOKUP
    KEYSAV  = KEY USED IN MOST RECENT CALL TO LOOKUP
    IHASH   = HASH ADDRESS FOR CURRENT KEY
    WDSIZE  = NUMBER OF BITS IN MACHINE WORD
    TABLE   = INDEX TABLE
    KPLACE  = CURRENT INDEX IN TABLE
    N       = NUMBER OF BITS IN HASH ADDRESS
    FSL     = FREE STORAGE LIST
    FRSIZE  = NUMBER OF WORDS IN FSL
    KFREE   = POINTER TO NEXT AVAILABLE WORD IN FSL
    KPROBE  = POINTER TO CURRENT ITEM IN FSL
    LPROBE  = ITEM JUST PREVIOUS IN CHAIN TO CURRENT ITEM
            = 0 IF NO PREVIOUS ITEM

    SUBROUTINE LOOKUP(KEY,FOUND,VALUE)
    LOGICAL FOUND, FIRST
    INTEGER WDSIZE, FRSIZE, VALUE
    COMMON TABLE, FSL
    DATA WDSIZE /36/
    DATA FIRST /.TRUE./

THE FOLLOWING CARDS MUST BE CHANGED TO CHANGE THE TABLE SIZES.

    INTEGER TABLE(64)
    DATA N /6/
    DATA FRSIZE /4000/
    INTEGER FSL(4000)


    IF(FIRST) GO TO 91
    KEYSAV = KEY

USE AS A HASH ADDRESS THE PRODUCT OF THE
KEY WITH AN APPROPRIATE MULTIPLIER.

    IHASH = 0
    KEYA = IABS(KEY)
    DO 11 I=1,WDSIZE,N
11  IHASH = IHASH + KEYA/(2**(I-1))
    KPLACE = MOD(IHASH,2**N) + 1

DISCOVER WHETHER OR NOT CURRENT SLOT IN INDEX
TABLE IS OCCUPIED.

    LPROBE = 0
    KPROBE = TABLE(KPLACE)
    IF(KPROBE .EQ. 0) GO TO 41

FOLLOW CHAIN THROUGH FREE STORAGE UNTIL THE KEY IS
FOUND OR UNTIL THE END OF THE CHAIN IS REACHED.

21  IF(FSL(KPROBE+1) .EQ. KEY) GO TO 31
    IF(FSL(KPROBE) .EQ. 0) GO TO 41
    LPROBE = KPROBE
    KPROBE = FSL(KPROBE)
    GO TO 21

31  FOUND = .TRUE.
    VALUE = FSL(KPROBE+2)
    RETURN

41  FOUND = .FALSE.
    RETURN

C  INITIALIZE FREE STORAGE LIST

91  KFREE = 0
    K = FRSIZE - 2
    DO 92 I=1,K,3
    FSL(I) = KFREE
92  KFREE = I

    ISIZ = 2**N
    DO 93 I=1,ISIZ
93  TABLE(I) = 0
    FIRST = .FALSE.
    GO TO 1



    ENTRY INSTAL(KEY,VALUE)

C  CHECK FOR PROPER KEY AND CHECK WHETHER KEY
C  IS ALREADY THERE OR NOT.

    IF(KEY .NE. KEYSAV) GO TO 99
    IF(KPROBE .EQ. 0) GO TO 121
    IF(FSL(KPROBE+1) .EQ. KEYSAV) GO TO 131

C  ALLOCATE SPACE FOR THE NEW ENTRY

121 IF(KFREE .EQ. 0) GO TO 99
    KPROBE = KFREE
    KFREE = FSL(KFREE)
    FSL(KPROBE) = TABLE(KPLACE)
    TABLE(KPLACE) = KPROBE

C  MAKE THE ENTRY.

131 FSL(KPROBE+1) = KEYSAV
    FSL(KPROBE+2) = VALUE
    RETURN



    ENTRY DELETE(KEY)

C  CHECK FOR PROPER KEY.

    IF(KEY .NE. KEYSAV) GO TO 99
    IF(KEY .NE. FSL(KPROBE+1)) GO TO 99

C  BREAK THE CHAIN AND RESTORE THE SPACE TO FREE STORAGE.

    IF(LPROBE .EQ. 0) TABLE(KPLACE) = 0
    IF(LPROBE .NE. 0) FSL(LPROBE) = FSL(KPROBE)
    FSL(KPROBE) = KFREE
    KFREE = KPROBE
    RETURN

99  CALL ERROR
    STOP

    END
```