W. W. Peterson

Addressing for Random-Access Storage*

Abstract: Estimates are made of the amount of searching required for the exact location of a record in several types of storage systems, including the index-table method of addressing and the sorted-file method. Detailed data and formulas for access time are given for an "open" system which offers high flexibility and speed of access. Experimental results are given for actual record files.

Introduction

It is sometimes impossible to store information in an information storage system in such a way that the exact location of a record, or unit of information, can be determined completely from the identification for that record. This is usually the case, for example, when names are used for identification in business files.[1] Under these circumstances, given only the identification for a record, some searching will be required to locate the record in the storage system. This paper provides estimates of the amount of searching required for several storage systems, including detailed data on one system which often excels commonly used systems both in flexibility and in speed of access.

Although this study was motivated by an interest in the use of random-access storage systems of very large capacity for business application, the results apply to many other situations. The dictionary for language translation by a computer, the symbol table for an assembly program or compiler, and many other problems which are essentially table look-up require a system like those described in this paper. Likewise, the results are not restricted to storage systems of a certain size or speed, or even to electronic systems. No attempt was made, however, to extend the work to include cross-referencing; each record was assumed to have only one identification.

The next section explains the addressing problem in more detail. Following that are the descriptions of several addressing systems with methods for estimating length of search. Finally the application of these data is illustrated with several sample problems.

*Portions of this paper were presented at the AIEE Winter General Meeting at New York, January 25, 1957.

The addressing problem

In large files of records, a portion of each record is generally used as identification. Usually this identification is a number, different for each record, such as the man number in some payroll and personnel files, the part number in an inventory file, or the policy number in an insurance file. The identification may consist partly or entirely of alphabetic characters, for example, names in a business file or words in a dictionary. Any identification, however, can be converted to numbers. One simple scheme for accomplishing this conversion would be to list in sequence all the permitted identification groups of characters and to number them starting with "one." The numbers could then be used to replace the corresponding groups of characters as identification.*

Although identification numbers may be chosen in a sequence in which every acceptable number is used, as in the case of the serial numbers on checks, more frequently only a small fraction of acceptable identification numbers are used, and these are chosen in some mathematically erratic way. For example, in a file in which names are used as identification, sixteen character positions might be allowed for the name, and any of 27 characters (the alphabet plus a blank) might appear in any column. There are $(27)^{16}$, or about 8×10^{22} possible 20-character sequences, but in an actual file at most a few million would appear. More typically, the ratio of the number of acceptable identifications to the number actually used ranges from about two to perhaps a million.

*This is essentially the same as considering the sequence of characters as a representation of a number with the base equal to the number of characters permitted in each position. The conversion procedure described above is conceptually simple, but impracticable, although formulas or conversion schemes are not difficult to derive.

Whenever a file of records is stored in a data-processing system, some procedure must be devised for deciding where to store each record and for locating a stored record, given its identification number. Such a procedure will be called an *addressing system*. The addressing system should make the average access time, i.e., the average time required for obtaining a record, as small as possible. At the same time, the system should be economical, and in particular the storage space in the random-access memory should be used efficiently.

When all possible identification numbers are assigned to records, the identification numbers, perhaps with slight modification, can be used as address numbers for the random-access storage.* One and only one record will be placed in each storage section, and the access time will be that inherent in the memory device.

Addressing becomes difficult when only a small fraction of all possible identification numbers is used and these are chosen in some erratic way. If a memory section is assigned to each possible identification number, the access time would be that inherent in the memory device, but only a small fraction of the memory would contain records. (It would certainly not be economical to have a memory a thousand times larger in capacity than necessary to store the records.) The memory can be used efficiently by simply storing the records in the memory without trying to make memory addresses and identification numbers correspond. Then it would be necessary to search for the record, and the effective access time would be much greater than the inherent access time for obtaining a record of known memory address.

Practical schemes are compromises between these two extremes; they set up a partial correspondence between identification numbers and memory addresses and also require some searching. Several such schemes are considered in this paper.

Use of an index table

An often-suggested system for addressing is to store somewhere in the data-processing system a table which lists for each identification number corresponding to a record, the actual address in the random-access memory where the record is stored. But how should the table look-up be organized?

Upon closer analysis, it becomes clear that *the problem of table look-up is precisely the same as the problem of addressing for the random-access memory*. The table entries can be considered to be records consisting of an identification number and the address at which the corresponding real record can be found. These table entry "records" may be considerably shorter than the real records, however, and this may make the access time for the table look-up, and even the over-all access time, less than would be possible without the table look-up.

It is simple to evaluate such a system if the memory requirements and access time for the table look-up system are known. The average access time is the table look-up time plus the time required to read one record from the random-access memory. Likewise the memory space required is the space required for the file plus the space required for the table.

Given an addressing system, then, it may be used directly on the record file, or for the table look-up operation in an index file. The systems will be discussed in the remainder of this paper as applying directly to the file, although they could be used alternatively as table look-up systems in an index file.

Addressing systems using a sorted file

The most commonly used addressing systems depend upon having the file sorted so that the records are stored with the identification numbers in proper sequence.

One common way of finding a record in a sorted file, given its identification number, is to proceed as follows. The identification number of the desired record is compared with that of the middle record of the file. This comparison tells which half of the file contains the desired record. The next comparison is made with the middle record of the proper half and it determines which quarter of the file contains the record. The procedure continues, narrowing the search by a factor of two until the record is found. For a file of N records, this "binary search" requires about $\log_2 N$ comparisons and as many accesses to the file.

If the identification numbers for the file run from 0000 to 9999 and record No. 1000 is desired, it would seem more reasonable to look first at a point about one-tenth of the way through the file. If it turns out that the record at that point actually has identification 1100, then the next place to check would be 1/11 of the way back toward the beginning of the file. In general, at each stage, one might estimate by linear interpolation or some other simple scheme where the desired record is and then look for it there.

The effectiveness of such a system depends upon the statistical characteristics of the file. If the identification numbers are chosen with perfect uniformity, this system will find the record on the first access. For the case of randomly chosen identification numbers, a lower bound can be determined for the number of accesses to the file required by this modified binary search. This bound can be established by an argument which involves estimating the amount of information required to locate a record.

The optimum point at which to enter the file at each stage is the point which divides the file into two parts of nearly equal probability. For large files, this probability will be very close to one-half. Entry at this point gives the most information regarding the record, very nearly one bit.[2] Thus the uncertainty, or entropy, of the location of the record, provides an estimate of the number of accesses required.

*Minor adaptations might be necessary in certain applications. For example, a file of 3000 100-character records with identification numbers assigned in succession could be stored in a memory which has 1000 300-character sections by placing the first three records in the first memory section, the next three records in the second memory section, and so forth. The calculation of memory address from identification number would consist of dividing the identification number by three and using the quotient as the memory address.

Consider a file of N records. It can be assumed without loss of generality that the identification numbers are between "zero" and "one," for if they were not, they could be normalized. It will be assumed that the set of N numbers are independent random numbers with a uniform probability distribution over the interval from "zero" to "one." This approximates the situation in which only a small fraction of a large number of acceptable identification numbers are chosen randomly.

For a search of the file, one of these N numbers would be known, and the required information is its position in the file. This is equivalent to knowing how many numbers are below it. Thus there are N possible events: no numbers below p, one number below p, . . . $N-1$ numbers below p.

Since the numbers are assumed to have a uniform probability distribution in the interval from "zero" to "one," the probability that any one number falls below p is equal to p. The numbers are independent, and therefore the probability that k of the $N-1$ other numbers falls below p is given by the binomial distribution,[3]

$$P_k = b\,(k; N-1, p) = \binom{N-1}{k} p^k q^{N-k-1}, \qquad (1)$$

where $q = 1 - p$.

When this expression for P_k is substituted in the formula for uncertainty or entropy,

$$H\,(p) = \sum_{k=0}^{N-1} P_k \log_2 P_k, \qquad (2)$$

the resulting formula is useful for calculations only for very small values of $N-1$. A few values, calculated on the IBM 704, are given in Table 1.

For large values of $N-1$, an approximate formula for H can be found by approximating the binomial distribution by the normal distribution and approximating the sum by an integral. The resulting expression, derived in Appendix I, is

$$H \approx \tfrac{1}{2} \log_2 (N-1) + \tfrac{1}{2} \log_2 pqe. \qquad (3)$$

Unfortunately, it does not converge rapidly. Some comparisons between the values computed directly from (2) and the approximation (3) are given in Table 2. For large values of $N-1$, the second term in (3) becomes negligible, and H is approximately $0.5 \log_2 N$.

This quantity H represents the average amount of uncertainty as to the location of the record, or the amount of information required to find a record on the average. Since for large files each access to the file provides very nearly one bit of information (and on the average no more than one bit), the modified binary search requires on the average about $0.5 \log_2 N$ (and on the average no fewer) accesses to the file, or about half as many as required by the ordinary binary search.

Another way in which addressing with a sorted file may be modified, often to advantage, is to store, separate from the file, a table of key entries, like the thumb index on a dictionary. For example, with a file of 10,000 records, the identification and location of every 100th record might be stored in a separate table. A search of the table would narrow the search in the large file to a section of 100 records.

If the ordinary binary search is used in both the search of the key table and the indicated section of the main file, for a file of N records with a key table of k entries, the search of the key table requires roughly $\log_2 k$ comparisons and the search in the main file $\log_2 N/k$ comparisons. The total is then roughly $\log_2 N$, the same as without the key table. This indicates that there is no theoretical advantage in having the key table. There may be practical advantages, however. For example, it may be possible to keep the entire key table in the fast memory of the

Table 1 **Entropy of the binomial distribution — *(Base Two)***

$P =$	0.0001	0.001	0.01	0.1	0.5
N					
10	0.0114	0.0809	0.4809	1.8436	2.7064
20	0.0208	0.1419	0.7714	2.4052	3.2077
30	0.0295	0.1955	0.9957	2.7245	3.5004
40	0.0376	0.2442	1.1799	2.9445	3.7080
60	0.0530	0.3319	1.4717	3.2475	4.0005
80	0.0673	0.4100	1.7971	3.4595	4.2080
100	0.0809	0.4813	1.8790	3.6229	4.3690
120	0.0940	0.5472	2.0303	3.7561	
140	0.1065	0.6086	2.1590	3.8684	
160	0.1187	0.6663	2.2702	3.9655	

Table 2 **Entropy of the binomial distribution — *Comparison of approximate formula and calculated values.***

N	P	*Calculated*	*Formula* (3)
10	0.1	1.8436	0.5694
	0.5	2.7064	1.3064
40	0.01	1.1799	0.0349
	0.1	2.9445	1.6271
	0.5	3.7080	2.3641
100	0.5	4.3690	3.0361
175	0.01	2.3446	1.1137
	0.1	4.0307	2.7059

machine, making the key table search much faster than the search in the main file.

The addressing systems which use a sorted file have several drawbacks. In the first place, the file must be sorted before storing, and sorting is a time consuming process. Secondly, they cannot accommodate the insertion of new records and the deletion of records no longer required, without resorting. With an "open" type system it is possible to overcome these drawbacks and gain other advantages.

"Open" type addressing systems

The typical open type addressing system is organized as follows: there is a set of rules which determines, for each acceptable identification number, a list of possible memory positions in which the record might be stored. Initially the record is normally stored in the first position on the list. If that is already occupied, the second listed position would be used. If that also is full, the third would be tried, etc., until an empty position is found. When a search is made for a record, the search starts with the first memory position listed and proceeds down the list until the record is found.

Of all the possible variations to this procedure, the simple system to be described is superior to anything else which I have considered. It is the only open system which will be considered in any detail in this paper.

This open system can best be described by example. Consider a file of 8,000 records. The memory is divided into 1000 "buckets." From the identification number of each record a three-digit number would be derived, perhaps by using a certain three digits of the original identification number. This three-digit number designates the bucket of memory in which the record should be stored. The average number of records per bucket of the memory is 8,000/1,000 = 8. The buckets should have somewhat larger capacity, perhaps 10, in order to take care of most of the upward deviations from the average.

In those cases when a bucket is filled, additional records intended for that bucket are stored in vacancies in succeeding buckets. For example, suppose that the three-digit number derived from the identification of a particular record is 680. Then that record should be stored in bucket number 680. If that bucket is full, it would be stored in 681. If 681 is also filled, an attempt would be made to store it in 682. This procedure would be carried on until a space was found for the record.

In searching for a record, a similar procedure would be used. For example, if the three-digit number derived from the identification of a record were 997, then the search for the record would begin at bucket 997 and proceed to 998, 999, 000, 001, 002, etc. until the record was found.

The important parameters of this system are the bucket size, the total memory size, and the total number of records in the storage. Rather than to refer to the total number of records stored, it is usually more convenient to refer to the ratio of the number of records stored to the total capacity of the memory, i.e., the percentage of the memory being used.

As long as there is a space anywhere in the memory, another record can be stored with this system. The search for a place to store the record will start from the point designated by the number derived from the identification and it will continue until a space is found. The access time for the last few records is so great, however, that it generally makes this type of operation impractical. Therefore, in evaluating this system, the most important characteristic is the average number of buckets through which one must search to find any given record, as a function of the bucket size and the percentage of the memory which contains records.

It turns out to be very difficult to calculate the average length of search for this system. Therefore, it was simulated on an IBM 704 data-processing system, using random numbers in place of record identification numbers, and also using several actual record files. The method of simulation is outlined in Appendix II.

The results from a single simulation run with random identification numbers are presented in Table 3.

Table 3 **Length of search for a record** *in a random-access memory with the open addressing system and random identification numbers.*
(Bucket capacity 20 records.
Memory capacity 10,000 records, 90% full.)

Length of Search	*No. of Records*	*Length × No.*
1	8418	8418
2	336	672
3	111	333
4	70	280
5	26	130
6	9	54
7	14	98
8	7	56
9	5	45
10	1	10
11	1	11
12	0	0
13	1	13
14	1	14
	9000	10134

For this file 8418 of the 9000 records would be found in the first bucket searched, while in 336 cases the search would continue to the next bucket and end there, etc. The average length of search is found by calculating the total number of buckets which would have to be searched to find every record in the file and dividing by the total number of records. For this case it is 10,134/9000 = 1.126.

Table 4 shows the average length of search at various stages during the loading of the random-access memory.

The table gives data for four separate runs with the simulated memory loaded with different random numbers on each run. There is a rather wide variation in the results, especially when the memory was nearly full. This was typical of all the data obtained.

Extensive data on the average length of search for a record are presented in Tables 5A and 5B. The data are also shown graphically in Fig. 1. These data are also results of the simulation of the 704 data-processing system, as described in Appendix II. Because of the large variations in results, as observed in Table 4, several runs were made for each case and the table entries are the averages of the results from the several runs.

All the data obtained by simulation for a particular bucket size were also made with the same memory capacity. Except when the memory is almost full, the average length of search will not be greatly affected by the memory size. This is because the maximum length of search is very unlikely to be comparable with the memory size. When the memory is full or lacks only a few records of being full, however, the last few records inserted are likely to be placed so far from the first address tried that the length of search for these records is an appreciable fraction of the memory size, and the average length of search does depend strongly on the memory size.

Figure 1 **Average length of search for a record** *in random-access storage with open addressing system (simulated system with random identification numbers).*

Table 4 **Average length of search for a record** *in random-access memory with the open addressing system. (Bucket capacity 20 records. Memory capacity 10,000 records. Random identification numbers. Individual data from four runs.)*

% Full	1st Run	2nd Run	3rd Run	4th Run
40	1.000	1.000	1.000	1.000
60	1.001	1.002	1.002	1.003
70	1.008	1.013	1.009	1.010
80	1.026	1.043	1.029	1.035
85	1.064	1.073	1.062	1.067
90	1.134	1.126	1.138	1.137
95	1.321	1.284	1.331	1.392
97	1.623	1.477	1.512	1.797
99	2.944	2.112	2.085	2.857
100	4.735	3.319	3.830	4.279

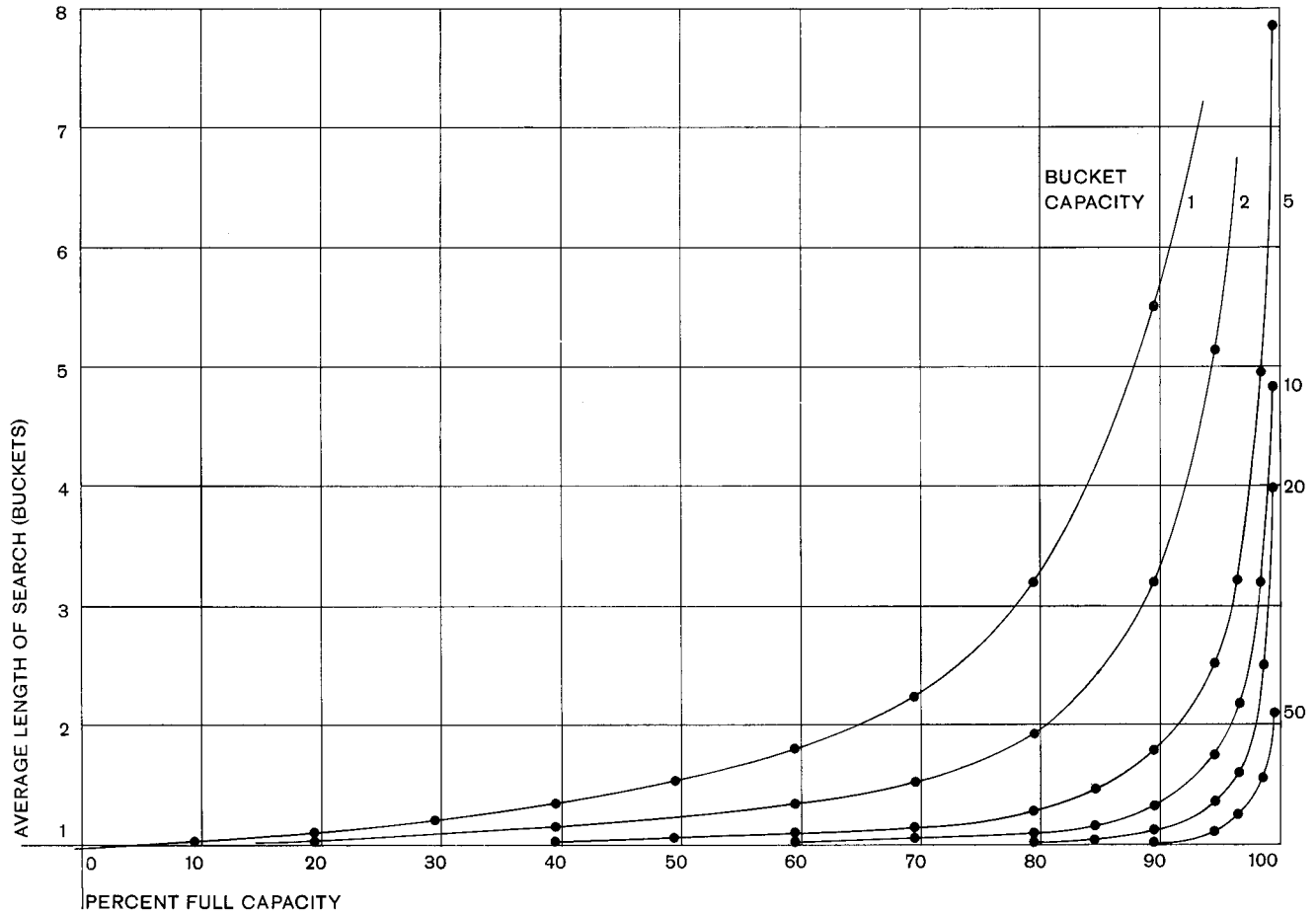


Table 5A **Average length of search for a record**
in random-access memory with 704 addressing system (simulated memory with random identification numbers).

Bucket Capacity (Records)	5	10	20	30	40	50
Memory Capacity (Records)	2,500	5,000	10,000	10,000	10,000	10,000
Number of Runs	8	7	4	4	3	3
% Full						
40	1.015	1.001	1.000	1.000	1.000	1.000
60	1.072	1.016	1.002	1.001	1.000	1.000
70	1.131	1.042	1.010	1.003	1.001	1.000
80	1.280	1.111	1.033	1.017	1.011	1.005
85	1.443	1.172	1.066	1.038	1.028	1.015
90	1.762	1.330	1.134	1.082	1.071	1.034
95	2.467	1.755	1.334	1.231	1.185	1.110
97	3.154	2.187	1.602	1.374	1.399	1.228
99	4.950	3.212	2.499	1.852	2.007	1.585
100	6.870	4.889	4.041	2.718	2.844	2.102

A rough quantitative estimate can be made as follows for bucket size 1: Consider a memory of size M in which there are R records. It can be shown that if the records are randomly placed in the memory, and if one enters the file at any given point and looks there and at successive memory positions for an unused memory position, the length of search will be, on the average,

$$S_R = \frac{M+1}{M+1-R}. \tag{4}$$

(Note that this agrees with the obvious answer $S_R = 1$ for the first record, i.e. $R = 0$, and $S_R = (M+1)/2$ for the last record, i.e. $R = M - 1$.)

Table 5B **Average length of search for a record**
in random-access memory with 704 addressing system (simulated memory with random identification numbers).

Bucket Capacity (Records)	1		2
Memory Capacity (Records)	500		1000
Number of Runs	9		9
% Full		*% Full*	
10	1.053	20	1.034
20	1.137	40	1.113
30	1.230	60	1.325
40	1.366	70	1.517
50	1.541	80	1.927
60	1.823	90	3.148
70	2.260	95	5.112
80	3.223	100	11.389
90	5.526		
100	16.914		

Now, storing random numbers using the open addressing system is not quite the same as storing records in a completely random manner, and it turns out that the open system would require a somewhat longer search than this. This is close enough, however, to give an indication of the behavior to be expected.

If formula (4) is used for the length of search for the $(R+1)$st record inserted into the file, then the average length of search for a record in a file with R_0 records in it would be:

$$S = \frac{1}{R_0} \sum_{R=0}^{R_0-1} \frac{M+1}{M+1-R}. \tag{5}$$

For files having several thousand records, it would be difficult to calculate this sum. It can be approximated by using the formula[4]

$$\sum_{K=1}^{n} \frac{1}{K} \approx \log_e n + 0.5772157. \tag{6}$$

Formula (5) can be rewritten

$$S = \frac{M+1}{R_0}\left[\sum_{K=1}^{M+1}\frac{1}{K} - \sum_{K=1}^{M-R_0+1}\frac{1}{K}\right], \tag{7}$$

and substituting from (6) gives

$$S = \frac{M+1}{R_0}\left[\log_e(M+1) - \log_e(M+1-R_0)\right], \tag{8}$$

provided $M + 1 - R_0$ is not too small.

Dropping the 1's in the terms $M + 1$, and denoting the percentage full by

$$P = \frac{R_0}{M}, \tag{9}$$

the approximation becomes

$$S \approx -\frac{\log(1-P)}{P}. \tag{10}$$

For a full memory, the second sum in (7) consists of only one term, equal to one, and the formula becomes

$$S_{\text{full memory}} \approx \frac{M+1}{M}\left[\log_e (M+1) + 0.5772157 - 1\right]$$

$$\approx \log_e(M+1) - 0.4227843. \qquad (11)$$

Note that this approximation for a memory not full is independent of the memory size, but the formula for the full memory gives a result which increases as $\log_e(M+1)$. Even for the full memory, making it $e(=2.718)$ times as large only increases the length of search by one record.

Numerical calculations from these formulas are compared with the observed results for memory size 500 in Table 6.

The results in Table 6 agree well only for 10% full. However, they do have the right order of magnitude, and, while this is not a sufficiently good approximation to use in engineering calculations, they indicate the type of dependence on memory size.

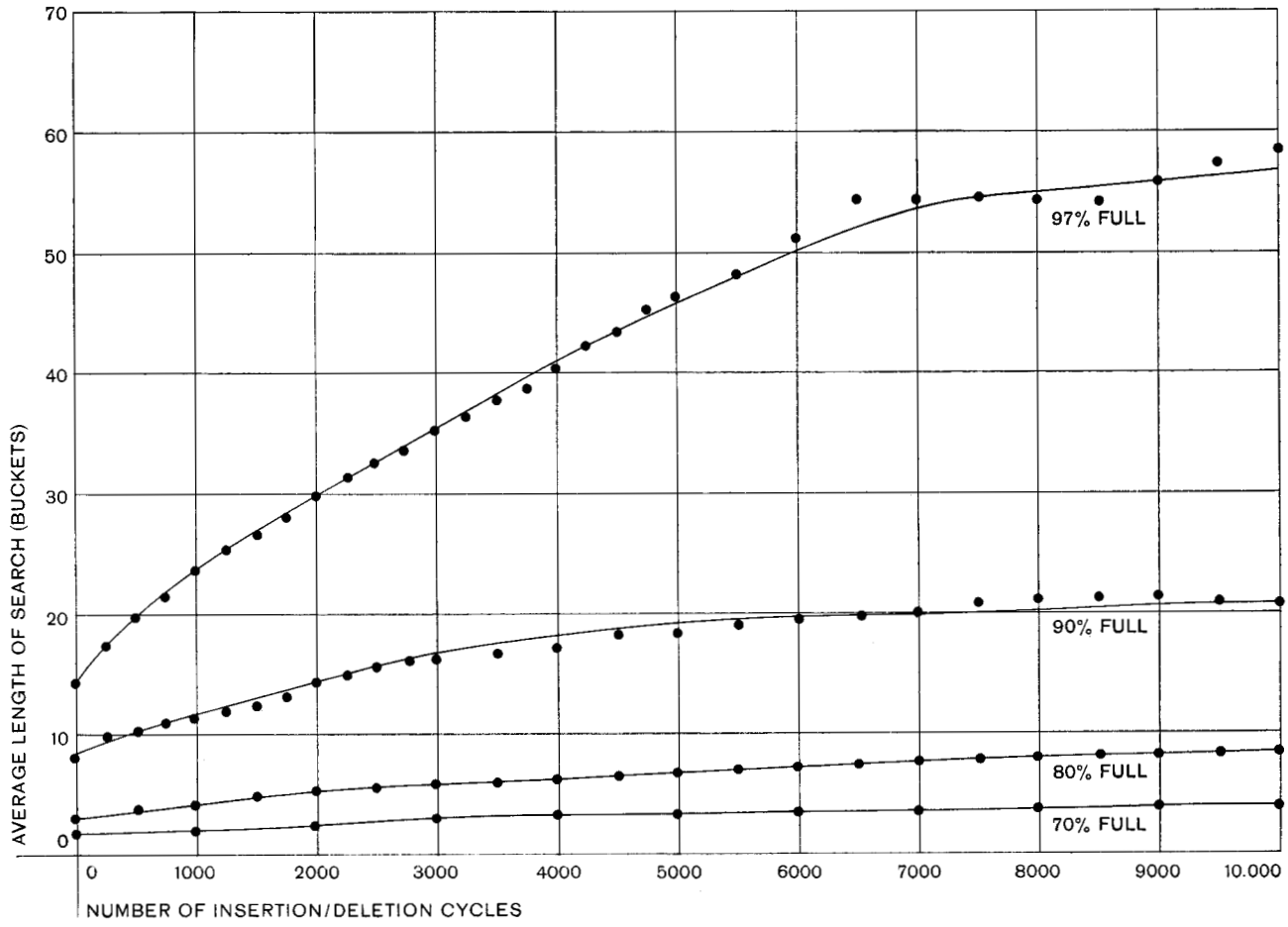
It is clearly possible with the open system to insert new records, or to delete records no longer required, at any time without disturbing the rest of the file. This feature is important, since insertions and deletions are necessary in many applications, particularly in business data processing.

Table 6 **Comparison of simulation results** *and calculations from formulas (10) and (11).*

	Average Length of Search	
Percent Full	*Calculated from (10) and (11)*	*Simulation on 704*
10	1.054	1.053
20	1.116	1.137
30	1.189	1.230
40	1.277	1.366
50	1.386	1.541
60	1.527	1.823
70	1.720	2.260
80	2.012	3.223
90	2.558	5.526
100	5.792	16.914

The effect of insertions and deletions on average access time was studied by simulation on the IBM 704, as described in Appendix II. During the run, first the memory was loaded to a certain fraction of its capacity, and the average length of search was recorded. Then a random

Figure 2(a) **Average length of search** *for open addressing system with insertions and deletions. Bucket size 1.*



number was inserted and one deleted, one inserted and one deleted, etc., until a certain number of these insertion-deletion cycles had taken place. Again the average length of search was printed. Another batch of insertion-deletion cycles was made, and another line printed, etc., until the file appeared to reach an equilibrium.

Some of these data are presented in Figs. 2(a) through (f) for memories 70%, 80%, 90% and 97% full, respectively. There was, in every case, a rather large increase in average length of search. The increase took place gradually and rather steadily as the processing continued, and it seemed to approach an equilibrium when the number of insertion-deletion cycles was in the order of several times the memory capacity.

Effects of rearranging records before storing them

An interesting fact about the open addressing system is that when there are no deletions, and when records are all used with equal frequency on the average, the average number of accesses required to find a record is independent of the order in which the records are stored. The proof will now be shown.

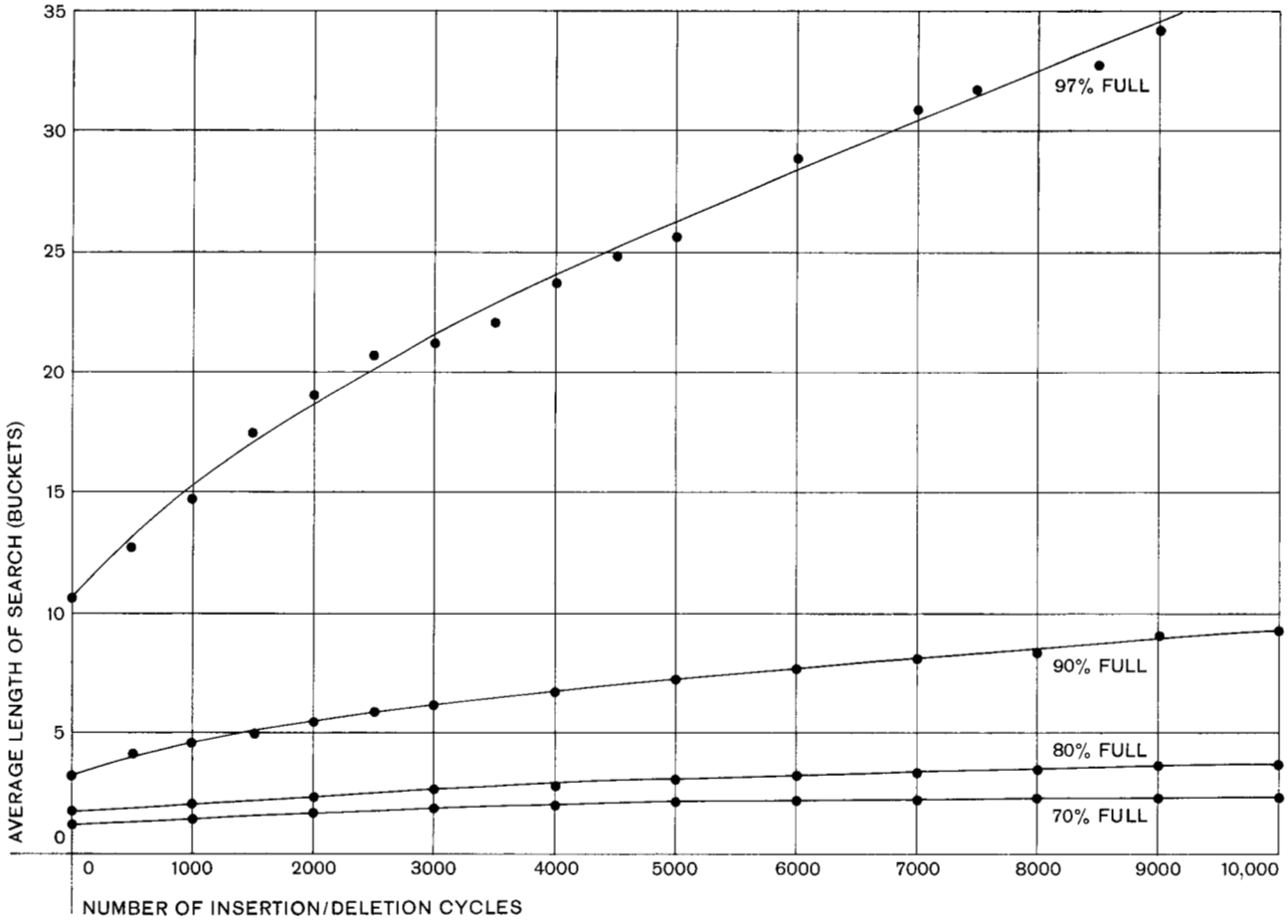
Figure 2(b) **Average length of search**
for open addressing system with insertions and deletions. Bucket size 2.

Suppose that a and b are two records in the file and that b is the next record following a in the process of storing the file in the random-access memory. Consider the effect of storing b and then a. There will be no effect if the records go into the same places as they would have gone in their normal order. This will be the case, unless b goes into the space occupied by a in the previous case. But then a will clearly go into the space previously occupied by b. The length of search for b is reduced, but the length of search for a is increased by an equal amount, and hence the average is unchanged.

Any rearrangement of the file can be achieved as a combination of (perhaps many) transpositions of two adjacent records, as described in the previous paragraph. Since each transposition leaves the average the same, so would the entire rearrangement.

It is interesting to note also that if the file were sorted initially, the records in each bucket will be in sequence after they are stored.

If some records in the file are used more frequently than others, and if the relative frequencies are known, then this knowledge can be used to reduce the average access time with the open addressing system. This could be done simply by storing the records in order by frequency of use, with the ones most frequently called for stored first. For example, consider the open addressing



system with bucket capacity 10, and capacity 10,000 records. Referring to Table 5, we find the average length of search when the memory is 40% full, i.e., contains 4000 records, is 1.001. This would also be the average length of search for the first 4000 records, no matter how many other records are stored in the memory, because storing additional records will not affect the locations of records already stored. Thus, if almost all transactions involve only the first 4000 records stored and very infrequent accesses are made to other records, the average length of search would be very little more than 1.001 buckets.

The average length of search for the next 2000 records can be calculated as follows. The average length of search for the first 6000 records is given as 1.016; since this is the average of access times for the first 4000 and the next 2000, the average for the next 2000 could be found from the formula $4000\ (1.001) + 2000X = 6000\ (1.016)$, which gives $X = 1.046$ as the average for the 4001st through the 6000th records. Similarly, the average length of search can be calculated for each stage of the storing process, and a new average weighted according to frequency of use can be calculated.

Figure 2(c) **Average length of search** *for open addressing system with insertions and deletions. Bucket size 5.*

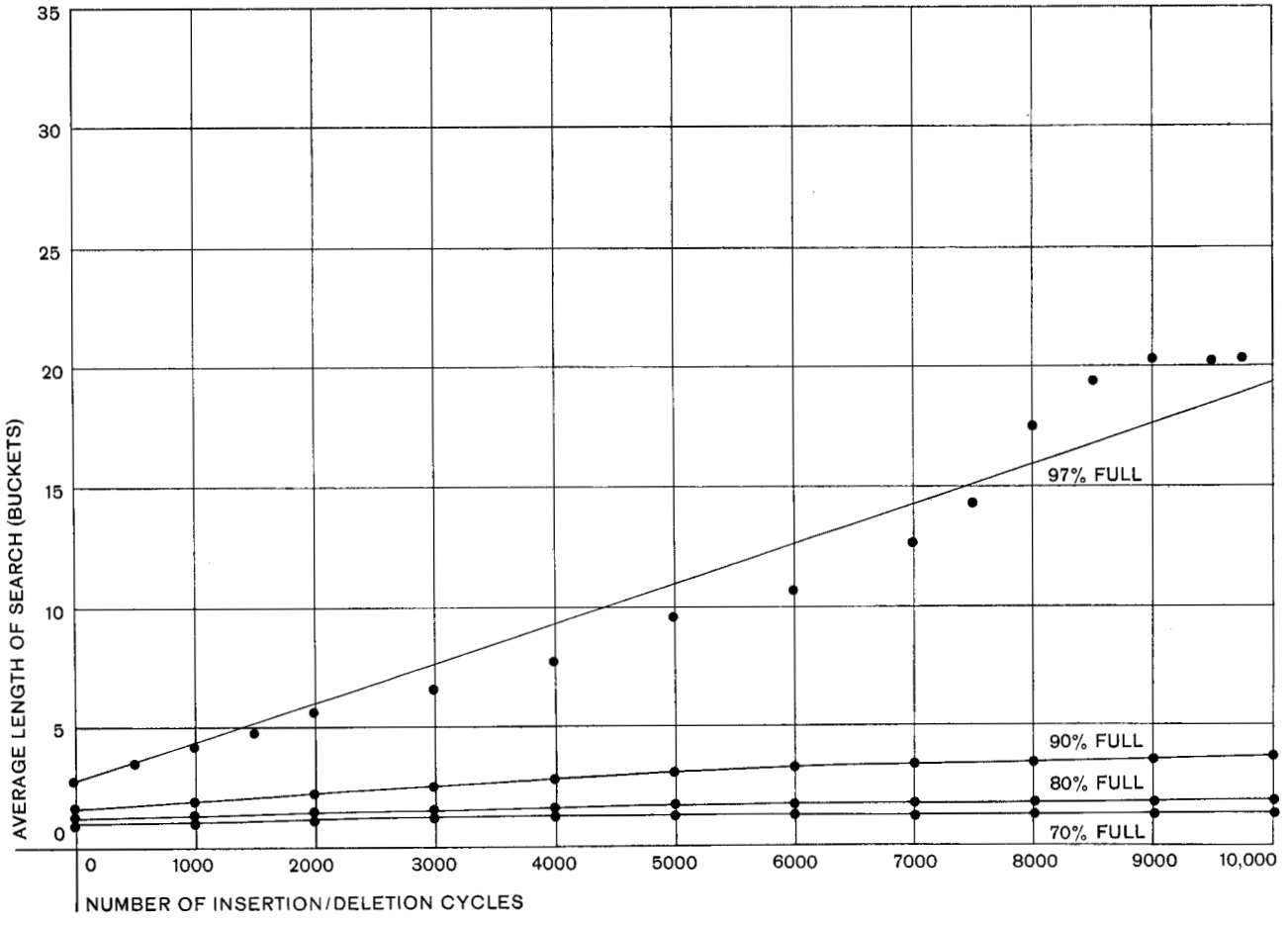
Table 7 **Average table length of search** *for the open system.**

Bucket Size	*Length of Search (Buckets)*	*Length of Search (Records)*
1	5.526	5.526
2	3.148	5.796
5	1.762	6.810
10	1.330	8.830
20	1.134	13.180
30	1.082	17.96
40	1.071	23.34
50	1.034	27.20

*The data in the first and second columns were taken from Table 5. In calculating the last column, it was assumed that the search involved the record which was found and half the capacity of the rest of the last bucket as well as all of the previous buckets. The formula used is

$$L_R = (L_B - 1)\ B + \frac{B-1}{2} + 1$$

where L_R = length of search in records
L_B = length of search in buckets
B = bucket capacity in records



Choice of bucket size

In many random-access memories, there is a natural unit of memory such as a track on a drum or on a disc, and the access is always made to the beginning of one of these units. In such a case, there appears to be no advantage in making a bucket any smaller than this.

On the other hand, in general there will be no advantage in using bucket size greater than one track (or one record, if a record is larger than a track). The advantage of small bucket size is shown clearly in Table 7, where average length of search is expressed in "records" as contrasted to "buckets" in the previous data. Exceptions to this rule might occur when there is considerable advantage in shortening the index number, or when the statistical properties of the identification numbers are such that access time would be shorter with a larger bucket size.

Other addressing systems

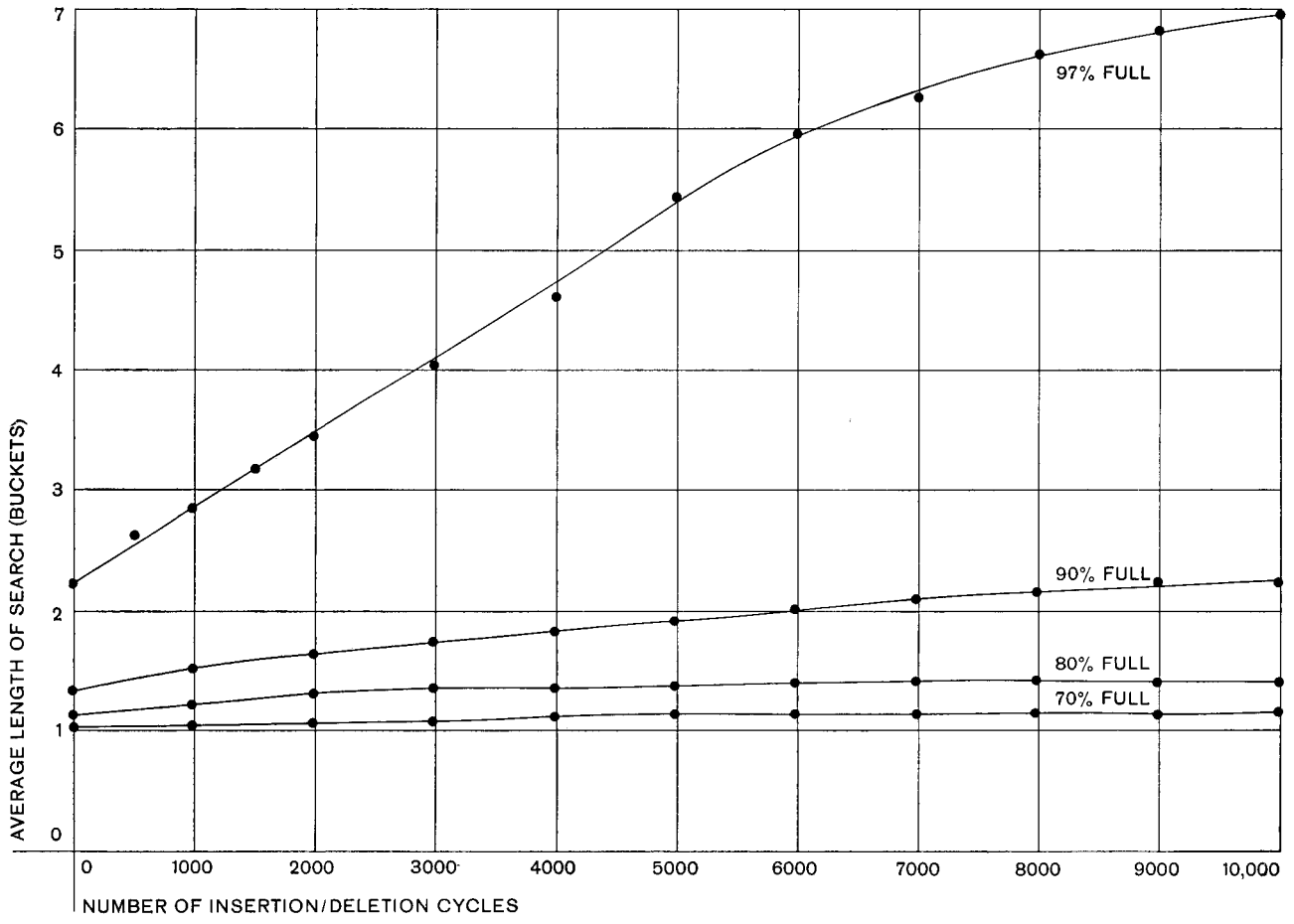
As an alternative to the open system described here, the memory might be divided into separate main store and overflow parts. A record would normally be stored in a main store bucket, but after a main store bucket is full any further records designated for that bucket would go to an overflow bucket according to some system.

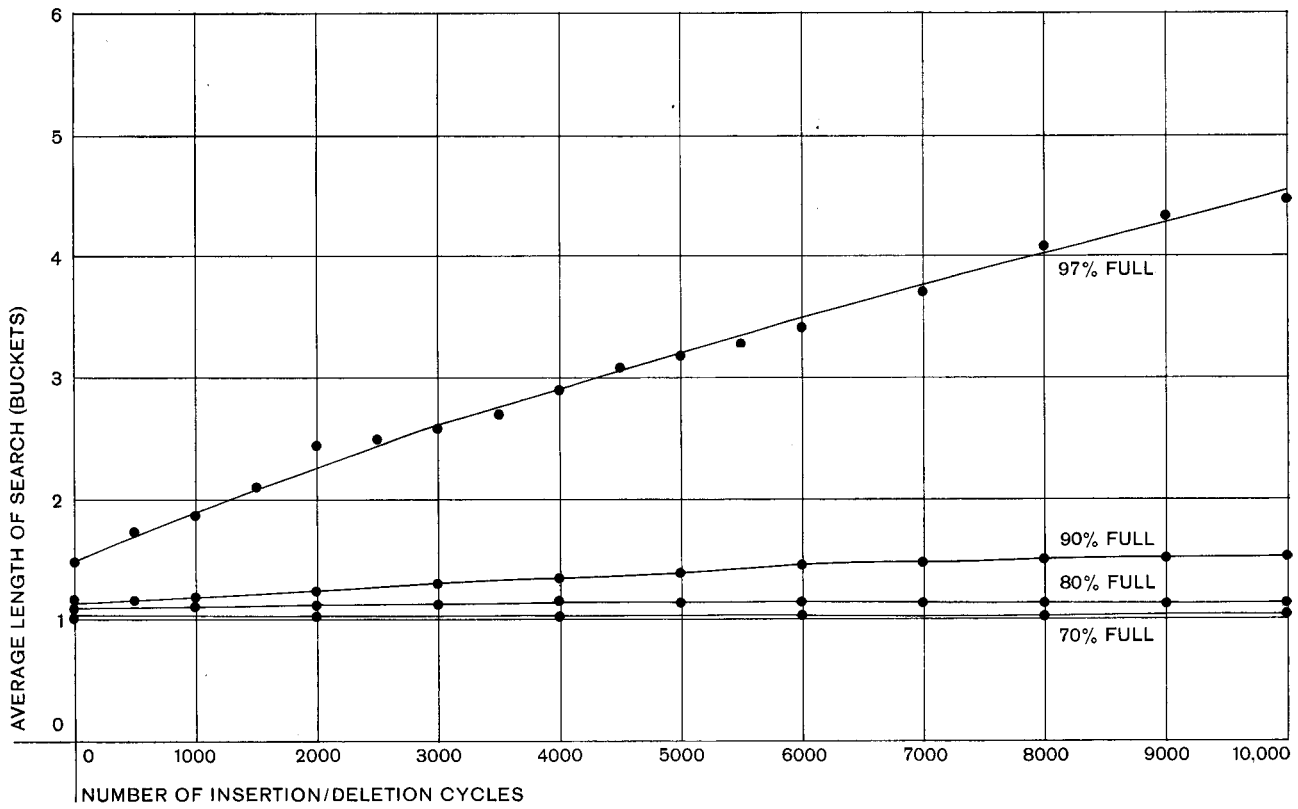
The difficulty with a system of this sort is that it provides no way of using the space in buckets in the main store which happen to have received few records. The overflow sections may, and indeed almost certainly will, be filled while there are still unused and unusable spaces in the main store. One such system was simulated on the 704. It was never superior to the open system described here. Although it was almost as good when the bucket size was large and the memory not too full, with small bucket size and a nearly full memory, the open system described here was far superior.

There is a system often used in business applications with insertions and deletions, which has features of both the open addressing system and addressing with a sorted file. A typical variation might be organized as follows.

First the file would be sorted. The storage would be divided into buckets. Let the number of buckets be N. Then the sorted file would be divided into N nearly equal parts. The first part would be stored in the first bucket, the second part in the second bucket, etc. A table of the identification numbers of the first record in each bucket

Figure 2(d) **Average length of search** *for open addressing system with insertions and deletions. Bucket size 10.*




AVERAGE LENGTH OF SEARCH (BUCKETS)
0
1
2
3
4
5
6
97% FULL
90% FULL
80% FULL
70% FULL
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10,000
NUMBER OF INSERTION/DELETION CYCLES

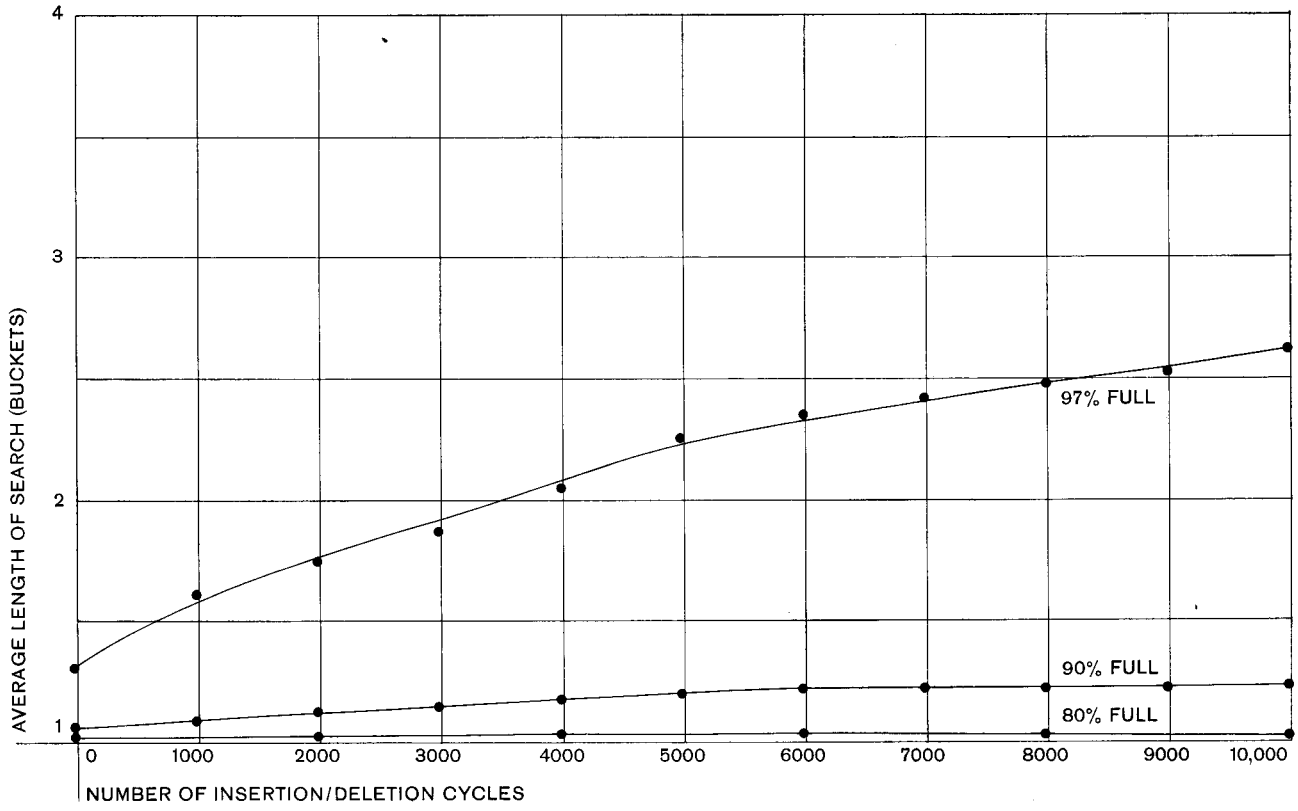

AVERAGE LENGTH OF SEARCH (BUCKETS)
1
2
3
4
97% FULL
90% FULL
80% FULL
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10,000
NUMBER OF INSERTION/DELETION CYCLES

Figure 2(e) **Average length of search** *for open addressing system with insertions and deletions. Bucket size 20.*

would be kept, preferably in the fast memory of the data processing machine. So far this system is just a variation of addressing with a sorted file.

In order to make handling of insertions and deletions possible, the buckets are made larger than necessary. When a record is to be inserted, the bucket in which it logically belongs is found from the table of identification numbers for the first record in each bucket. The record is stored in that bucket if there is space for it there. If that bucket is already full, an attempt is made to store it in the next bucket, etc. In other words, insertions are handled just as with the open addressing system.

With this system the inserted records would not generally be placed in the file in sequence. They would, however, be near their proper place, and this fact could be used to reduce the time required for re-sorting the file.

The average access time for this system depends upon the statistical properties of the identification of the records being inserted. It seems reasonable that in some situations, an inserted record is as likely to go in any one bucket as any other. On the other hand, buckets in a sparsely used region of identification numbers span a longer range of identification numbers than buckets in a densely used region, and it might be more reasonable to assume the probability is proportional to the number of identification numbers associated with a bucket. In still other situations, certain regions will receive more insertions than others because of peculiarities of the file. For example, the section of a parts file associated with a new machine would receive more than its share of insertions. The most favorable of the above mentioned possibilities is the first. The other two would have a greater tendency to overflow certain buckets.

This addressing system with insertions and deletions was also simulated on the 704. All buckets were assumed equally likely to receive each inserted record. This is the first and most favorable of the possibilities mentioned in the preceding paragraph. The data taken are exactly analogous to that taken for the open addressing system. The results are presented in Fig. 3.

The length of search for this system, *excluding sorting time and the table look-up,* is less than for the open addressing system with the same bucket size and percent full when the number of insertions is small. But, when the number of insertions reaches the size of the file, the length of search for the two systems is roughly the same. With the assumption that all buckets have the same probability of receiving each inserted record, this system is mathematically the same as the open addressing system during the insertion-deletion processing except for the initial condition of the file. Therefore, agreement would be expected after so many insertions and deletions that the files have reached equilibrium. This can be seen for the 70% full files, but for the others, the system with the initially sorted file still shows some advantage at the end of the runs.

Figure 2(f) **Average length of search** *for open addressing system with insertions and deletions. Bucket size 40.*

Experimental results with the open addressing system and actual record files

The identification numbers for four business record files were stored in the simulated random-access storage with the open addressing system. Though the experiments were limited in scope, they brought out some important points regarding the use of the open addressing system.

The method of simulation is described in Appendix II. In all of the experiments a bucket size of 10 numbers was used. The memory size was 5,000 or 10,000 records, depending upon the size of the file being stored. Some of

Table 8 **Performance of the open addressing system** *with actual files (bucket capacity 10).*

File Description and Method of Indexing	*Capacity Memory*	*Percent Full*	*Average Length of Search*
Random Numbers	5,000	80	1.111
		90	1.330
		100	4.889
IBM Parts Numbers			
Columns 7-6-5	10,000	90	1.227
		100	4.749
Columns 5-6-7	10,000	90	1.570
		100	13.776
First Account Numbers File			
Columns 2-3-4	5,000	80	1.118
Columns 4-2-3	5,000	80	1.075
Columns 3-2-4	5,000	80	1.086
Second Account Numbers File			
Columns 2-3-4	5,000	90	1.780
		100	30.965
Columns 4-3-5	5,000	90	1.780
		100	5.320
With Random Permutation			
Columns 2-3-4	5,000	100	4.206
Columns 4-3-5	5,000	90	1.172
		100	4.578
Names File			
Method A			
(See Appendix II)	10,000	80	1.872
		90	2.960
		100	14.945
Method B	10,000	80	1.888
		90	2.559
		100	10.080
Names File (Edited)			
Method A	10,000	80	1.241
		90	1.647
Method B	10,000	80	1.279
		90	1.792

the results are presented in Table 8. The average length of search for random identification numbers is included for comparison.

In the description of the open addressing system it was stated that an index number designating the bucket to be tried first should be derived from the identification number. The experiments described here consisted essentially of comparing average length of search for various methods of deriving the index numbers.

The first file consisted of the first 10,000 records of the IBM parts file. The identification was a seven-digit decimal number. The last three digits were each found to have at least as uniform a distribution as would be expected if they were randomly chosen. When they were used in reverse order as the index numbers, the average length of search was less than that found for random identification numbers. Note, however, that these same digits in their normal order resulted in considerably larger average length of search.

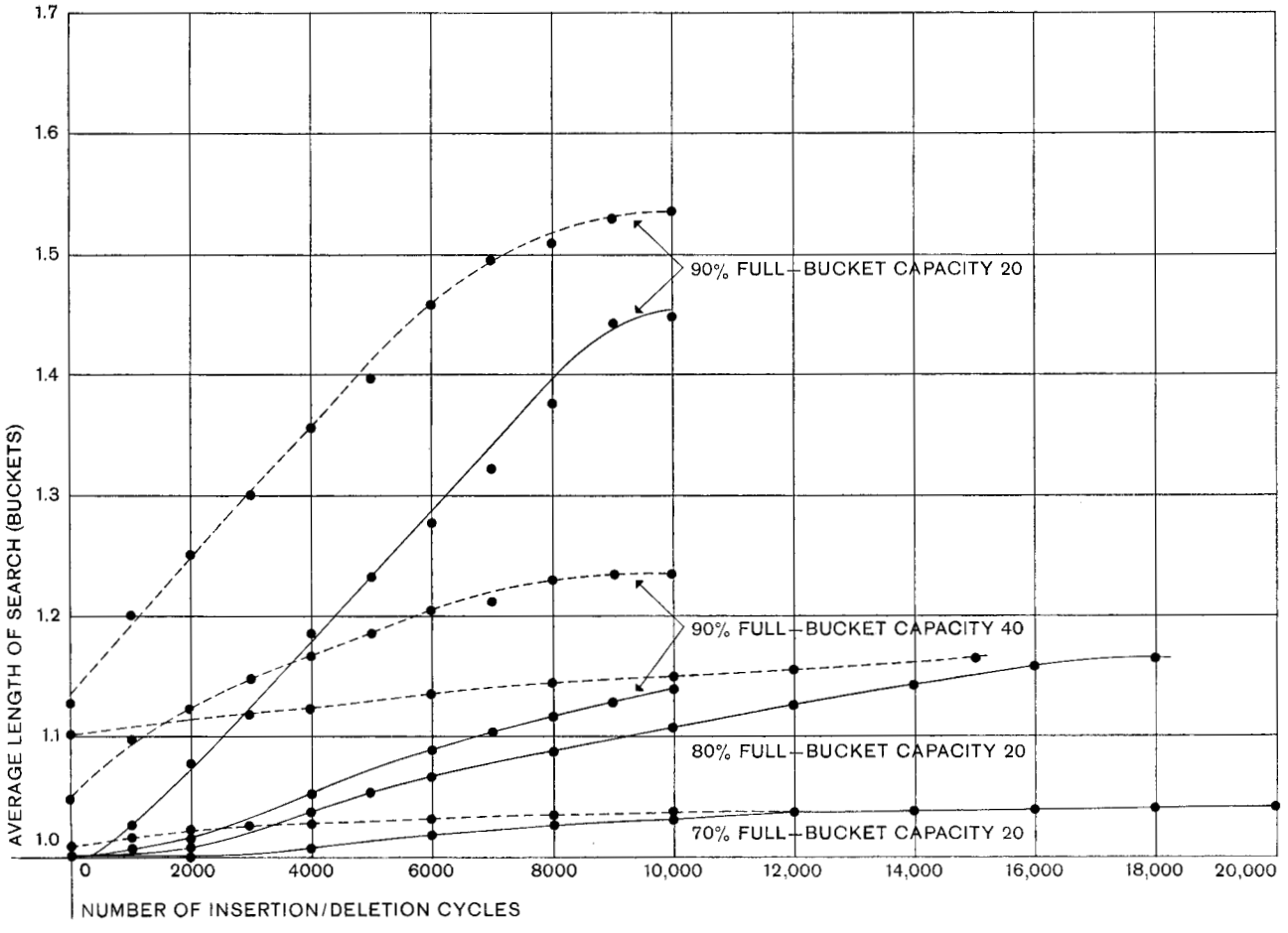
The second file consisted of a file of 4000 account numbers from an insurance agency. The identification consisted of five-digit numbers. The middle three digits were found most nearly uniform: the index number was found by halving a three-digit number found from the middle three digits. Of the three arrangements tried, the normal order for the digits gave the poorest results, only slightly poorer than those for random numbers.

The third file was also a file of account numbers from an insurance agency. All digits except the first were found to have at least as uniform distribution as would be expected with random numbers. Yet of about eight arrangements of these digits, none resulted in as short a length of search as was found with random numbers. A further modification of the identification numbers, consisting of applying a fixed randomly-chosen permutation to the 1000 possible three-digit numbers, resulted in marked improvement, giving results better than those found with random identification numbers.

Neither the rearrangement of digits nor the random permutation could have been successful if the digits had not already been quite uniformly distributed, as was the case with all three of these files. If certain digits predominate in each digit position, some more drastic modification of the identification numbers would be required. Schemes such as adding digits (dropping the carry), or using digits from the center of the square of the identification number have been tried with some success.

The fourth file consisted of about 10,000 last names

Figure 3 **Addressing with sorted file** *(solid curves)* **Open addressing systems** *(dashed curves)* *Memory capacity 10,000 records.*



of IBM employees. The index numbers were derived from the names by doing mathematical operations on the binary numbers which represented these names as they entered the IBM 704 during the simulation. A number of schemes such as squaring the number and using the center digits were tried. Those corresponding to the data in the table are described in Appendix II. Experiments were tried also with an edited version of this file in which nearly all duplications of names were eliminated. With this edited file, length of search was somewhat shorter.

In no case with this file was the length of search as short as with random identification numbers. I know no reason why these names should result in longer access times. In fact, it would seem that with non-random identification numbers one always ought to be able to use the non-randomness to his advantage to reduce length of search to a value less than that for random numbers. At the very least, one would expect to be able to "randomize" the identification numbers by some mathematical procedure to approach the length of search found for random identification numbers. The facts of the matter are, however, that though I tried diligently with this file, I did not succeed in finding a way of achieving results as good as were obtained with random numbers.

In summary, it appears that with some files it is possible to achieve, quite simply, an average length of search as good as or better than that found for random numbers. In other cases this can be achieved only after trying many schemes for deriving index numbers from identification numbers, or it may not be possible at all. The scheme which is good for one file is not necessarily good for another. Even though it may not be possible to achieve results as good as with random numbers, the open addressing system will frequently perform more efficiently than systems using a sorted file.

Applications

In this section the application of the data of previous sections is illustrated by the calculation of average access time for two problems. For simplicity, neither the problem nor the characteristics of the data-processing system will be considered in detail, and hence the results will not be precise. The average length of search found for random identification numbers will be used in these calculations. The data presented in the previous section indicate that the average length of search for actual files may deviate either up or down from average length of search for random identification numbers.

For the first problem, suppose that 18,000 250-character records are to be stored in an IBM magnetic disc random-access storage unit. What type of addressing system should be used, and what would be the average access time?

The following characteristics will be assumed for the disc unit:

50	discs
200	tracks per disc
500	characters per track
50 msec.	to read a track
100 msec.	access to adjacent track
200 msec.	access track to track on same disc
600 msec.	access track to track on different discs

These are roughly the maximum access times, but they will be used as average access times in the calculations. Computer instruction times relating to addressing will not be considered. In general, they will probably be small in comparison with the actual access times for the disc unit, and probably could be overlapped with disc unit access times in many cases. In a thorough analysis of an application, they would certainly have to be accounted for.

Consider first the open addressing system. As has been pointed out, the smaller the buckets are, down to a single track, the lower the access time will be. This indicates a bucket of size 1 track, or 2 records. For this problem, there will be 10,000 buckets, 90% full. Table 5 gives 3.148 buckets as the length of search, with no insertions or deletions. This corresponds to reading on the average 3.148 successive tracks.

The initial access is generally to a new disc and requires 600 msec. Occasionally it is to a new track on the same disc and requires only 200 msec. For simplicity the average will be considered to be 600 msec.; the corrected average is about two percent smaller. Likewise going from one track to the succeeding one generally requires 100 msec. Occasionally, however, the succeeding track is on a different disc face, and requires more access time. This fact will also be neglected; it results in an increase in the average of about 3%.

The total time for a search of length n tracks is the sum of the initial access time to the other $(n-1)$ discs, and the reading time for the n discs. The total is

$$T = 600 + 100 \cdot (n-1) + 50n$$
$$= 500 + 150 \cdot n$$

For average length of search, 3.148 successive tracks, the time would be

$$T = 500 + 150 \cdot 3.148 = 972 \text{ msec.}$$

When there are insertions and deletions, the average length of search increases gradually, finally reaching an equilibrium when the number of insertions and deletions is one or two times the number of records in the file. For this problem the equilibrium value, as seen from Fig. 2, is about 10. Thus after many insertions and deletions the average access time would be about

$$T = 500 + 150 \cdot 10 = 2000 \text{ msec.}$$

If the file is sorted and the binary search is used, the number of accesses to the file would be the next integer larger than $\log_2 18{,}000$, or 15. Each access would require an average of 650 msec. Clearly the time required would be many times that of the open addressing system.

If the file is divided into "buckets" and a table is kept in the fast memory of the machine which gives the iden-

tification number of the first record in each bucket, the number of the bucket which contains any desired record can be found by searching this table. Then the contents of this bucket can be brought into the main memory and searched for the desired record.

When the contents of a bucket are brought into the main memory, they could be searched one track at a time. If the desired record is in the first track, the search could end there. If it is not, the second track would be searched, then the third, etc. until the whole bucket had been searched. If each bucket contains enough records to fill m tracks, then a fraction $1/m$ of the records are in the first, likewise in the second, the third, or say the ith. The records in the ith track require that i tracks be read. Hence the average number of tracks which have to be read is

$$n = \frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m}\cdot\frac{m(m+1)}{2} = \frac{m+1}{2}.$$

It follows that the average access time is

$$T = 500 + 150\cdot\frac{m+1}{2}. \tag{11}$$

This, of course, does not include time required for table look-up. This system certainly could not match the open system if this time, exclusive of table look-up, is greater than 972 msec., the average time for the open system. The critical value of m can be found by solving

$$972 = 500 + 150\cdot\frac{m+1}{2},$$

and m is found to be 5.293 tracks. Thus for this system to excel the open system, the bucket size would have to be five or fewer tracks, i.e., ten or fewer records. It would require a table of 1800 entries or more, which would probably be prohibitive with present memories.

Now consider the possibility of using an index table. The records for this problem require 4,500,000 of the 5,000,000 characters of storage, leaving 500,000 characters for an index table. The table entries would require, let us assume, 8 digits for the identification number and 4 digits for the track number, or 12 digits per record. This makes 18,000 x 12 = 216,000 characters. Therefore the table requires less than half the remaining space. Each track could hold as many as 40 table entries, and this would make a good bucket size. With the table organized according to the open system of addressing and less than 50% full, the probability that a record would not be found in the first track searched would be so small that it would have a negligible effect on access time. The access time to the table would be

$$T = 500 + 150\cdot 1 = 650 \text{ msec.}$$

and a like amount of time would be required to get the record after the exact location of the record is found. Thus the total access time would be

$$T = 1300 \text{ msec.}$$

This would hold even with insertions and deletions. Therefore this system would be somewhat poorer than the open system with no insertions and deletions, but would be better for a problem in which many insertions and deletions occur. More uniform distribution of identification numbers would favor the open system, while this system would be much less sensitive to poorly distributed identification numbers.

Similar calculations made assuming 180,000 25-character records and 1800 2500-character records clearly indicate the use of the open system for the former, and an index table using the open system for table look-up for the latter case.

As a second example, consider a table look-up operation completely in the core memory of a 704 computer such as might be required for an assembly program or compiler. Assume that 2048 words are available for storage, and that each item or record requires one word of storage. An estimate of average access time is required for the binary search and for the search using the open addressing system described in this paper, with bucket size 1.

I found it possible to do the binary search on the IBM 704 with a program requiring $C_{\text{binary}} = 4 + 3n$ instruction executions, where n is the number of comparisons required, i.e. the smallest integer as $\log_2 N$ for an N record file. For the open addressing system, $C_{\text{Open}} = 7 + 2n$ instruction executions. This time n is the average length of search for bucket size one, from table 5B, assuming that the records in this problem store as efficiently as random numbers. It was assumed in both cases that the comparison is made on the entire word—to mask part of the word would require about $2n$ additional instructions in either case. In the case of the open system three instructions were allowed for the formation of the nine-bit index number which designates the first word to be compared: this is probably a bare minimum. On the other hand, no account is taken of the fact that the binary search requires the file to be sorted, while the open addressing system does not.

For the example being considered, for a file of anywhere from 1025 to 2048 records, n for the binary search is nine, and the search requires thirty-one instructions per search on the average. For the open addressing system, the results are summarized as follows:

Table 9 **Average length of table search** *for sample problem (open addressing system—704 computer).*

No. Items	*Percent Full*	*No. of Comparisons per Search*	*No. of Instructions Executions per Search*
1024	50	1.2	9.4
1229	60	1.325	9.65
1434	70	1.517	10.03
1638	80	1.927	10.85
1843	90	3.148	13.30
1946	95	5.112	17.22

The third column in the table was taken directly from Table 5B. At 100% full, the average access time for the open system would probably exceed that for the binary search, but not by a great amount. At 70% full the open addressing system is three times as fast as the binary search.

Conclusion

Several systems of addressing for random-access storage have been described in this paper. For each, data and formulas have been presented which enable one to estimate average access time for records in the storage. Two simplified examples of the calculation of average access time for specific problems have been included.

While the best system to use will depend in general upon the problem, the "open" addressing system described in this paper does seem to offer advantages in flexibility, simplicity, and speed over commonly used systems based upon a sorted file or index.

Acknowledgment

To the best of my knowledge, the open type addressing system described in this paper was devised in 1954 by Dr. A. L. Samuel, Dr. G. M. Amdahl, and Miss Elaine Boehme for use in the table look-up process in an assembly program for the IBM 701. The system is so natural, that it very likely may have been conceived independently by others either before or since that time.

I have received many ideas and suggestions from a number of engineers at IBM both from conversations and from reports. Dr. A. L. Samuel, A. J. Critchlow, and N. Rochester have been particularly helpful.

Appendix I

In this appendix, an estimate of the entropy of the binomial distribution is found by using the normal approximation[3]:

$$P_k = b\,(k; N-1, p) = \binom{N-1}{k} p^k q^{n-k-1} \approx h\,\phi\,(X_k),$$

where

$$h = [(N-1)\,pq]^{-\frac{1}{2}},$$

$$X_k = h\,[k - (N-1)\,p],$$

and $\phi\,(X) = (2\pi)^{-\frac{1}{2}} \exp\left[\frac{-X^2}{2}\right]$

Then,

$$H\,(p)$$

$$= -\sum_{k=0}^{N-1} p_k \log_2 p_k \approx \sum_{k=0}^{N-1} -h\,\phi\,(X_k) \log_2 [h\,\phi\,(X_k)]$$

$$= -\sum_{k=0}^{N-1} h\,\phi\,(X_k) \log_2 h - \sum_{k=0}^{N-1} h\,\phi\,(X_k) \log_2 \phi\,(X_k)$$

$$= -\log_2 h$$

$$-\sum_{k=0}^{N-1} h\,\phi\,[hk - h\,(N-1)\,p] \log_2 \phi\,[hk - h\,(N-1)\,p]$$

The last sum can be approximated by an integral,

$$H\,(p) \approx -\log_2 h - \int_{-h(N-1)\,p}^{h(N-1\,(1-p)} \phi\,(X) \log_2 \phi\,(X)\,dx.$$

$$\approx -\log_2 h - \log_2 e \int_{-\infty}^{\infty} \phi\,(X)\,\log_e \phi\,(X)\,dx.$$

The last step is possible for large N, since the integrand becomes very small for large positive or negative values of X.

The integral can be found in many tables of integrals; its value $-\frac{1}{2}$. Thus

$$H\,(p) \approx -\log_2 h + \tfrac{1}{2}\log_2 e = \tfrac{1}{2}\log_2 (N-1) + \tfrac{1}{2}\log_2 pqe$$

The average value of $H\,(p)$ for all values of p is

$$H = \int_0^1 H\,(p)\,d\,p$$

$$= \tfrac{1}{2}\log_2 (N-1) - \tfrac{1}{2}\log_2 e\,.$$

The maximum of $H\,(p)$ occurs when $p = \frac{1}{2}$, and then

$$H(\tfrac{1}{2}) = \tfrac{1}{2}\log_2 (N-1) + \tfrac{1}{2}\log_2 e - 1.$$

Appendix II

In this appendix the method of simulating the open addressing system on an IBM 704 data-processing system is described.

Nine bits of memory (one-fourth of a 704 word) were used for each record position in the simulated random-access memory. The program was set up to allow any memory size up to 10,000 records. This could be divided into buckets of any size, provided the memory was an integral multiple of the bucket size. A "zero" in any record position (9 bits of memory) indicated that no record was stored in that position. In storing a record, the machine would calculate the proper bucket number from the record identification, go to the first record position in that bucket, and start from there to search each record position until a position containing a zero was found. At the same time count was kept of the *number of buckets* which had to be searched. When an empty record position was found, this number was stored, to indicate that a record was stored there and to store the access time for that record. Note that while this model is very similar to the open addressing system, only the necessary information is kept. Neither the actual records nor even the record identification is of interest. The only important data are the presence of a record and its access time.

There was, in some cases, a possibility that a record would be more than 511 buckets from where it was entered. Whenever this occurred, the number 511 was entered, since no larger number could be stored with 9 bits. If this happened more than a few times, it would affect the accuracy of the calculation of average access time. Therefore, the number of 511's was checked in cases where they were at all likely to occur, and if they occurred, the run was repeated with smaller memory size.

Deletions were handled as follows: a random number between zero and the random-access memory capacity was generated, and the corresponding nine-bit record position was inspected. If it was not zero, it was made zero, i.e. this record was deleted. If it was already zero, another random number was generated and another position inspected, etc., until a record was deleted.

The average length of search is obviously the sum of all the 9-bit numbers, divided by the number of records in the file.

The random-number generator used was the currently popular one, successive powers of 5^{13} mod 2^{35} (Refs. 5, 6). The right-hand 18 bits of each 35 bit "random number" were dropped by shifting the number to the right 18 places in the computer. A number between zero and $M-1$ was required, where M was the number of buckets in the memory for an insertion, and the capacity of the memory in the records for a deletion. This number was obtained by dividing the remaining 17 bits of the "random number" by M, and using the remainder. The remainder, which was in the accumulator, was used as the random number.

The first three actual record files were stored on tape and read by a subroutine which translated any specified three or four columns into a binary number. This subroutine simply replaced the random-number generator subroutine.

Simple permutation of the digits did not sufficiently randomize the identification numbers in all cases. A more thorough randomizing was accomplished as follows: one thousand cards were numbered with punches from 000 to 999, and were punched with random numbers in fifteen other columns. Then they were sorted on six columns of random numbers in order to put them in random order. Then they were stored in the 704 memory in random order. When an identification number was read from the tape, it was converted by table look-up in this table before being stored in the random-access memory.

The names file and the edited-names file were also stored on tape. The names consisted of at most twelve characters, which were brought into the IBM 704 in a binary code in two 36-bit 704 words. They were read and processed by a subroutine which replaced the random-number generator.

A number of methods for transforming this identification into an index number were tried. The two best methods correspond to the data in Table 8. Of those, method A consisted of adding the two binary words comprising the identification, squaring, shifting to obtain the center 36 bits of the 72-bit square, and finally, dividing the center by 1000 and using the remainder. Method B consisted of dividing each of the two 36-bit words into two 18-bit words, adding these four 18-bit words, dividing by 1000, and using the remainder.

References

1. A. I. Dumey, "Indexing for Rapid Random-Access Memory," *Computers and Automation* **5**, No. 12, 6-8 (Dec. 1956).
2. C. E. Shannon and W. W. Weaver, *Mathematical Theory of Communication,* University of Illinois Press, 1949.
3. Feller, *An Introduction to Probability Theory and its Applications,* Wiley, New York (1950), pp. 104-106.
4. Whitaker and Watson, *Modern Analysis,* Cambridge University Press, Cambridge, England (1927) p. 235. The constant is known as Euler's constant. The formula is within about 2.2% for $n = 10$ and improves as n increases.
5. D. H. Lehmer, "Mathematical Methods on Large Scale Computing Units," Harvard Computation Laboratory *Annals,* **26**, 141-146, (1951).
6. O. Taussky and J. Todd, "Generation and Testing of Pseudo-Random Numbers," *Symposium on Monte Carlo Methods,* edited by H. A. Meyer, Wiley, New York (1956) pp. 15-28.

Received October 1, 1956