Exhibit 7

SORTING AND SEARCHING

THE ART OF COMPUTER PROGRAMMING

VOL. 3

ADDISON WESLEY

culty is to look more closely at the structure underlying Theorem S; for short progressions of keys, only the first few partial quotients of the continued fraction representation of θ are relevant, and small partial quotients correspond to good distribution properties. Therefore we find that the best values of θ lie in the ranges

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

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

$$A = \begin{array}{|c|c|c|c|c|c|}\hline + & 61 & 25 & 42 & 33 & 71 \\ \hline\end{array}\ . \tag{9}$$

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

A good hash function should satisfy two requirements:

a) Its computation should be very fast.

b) It should minimize collisions.

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

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

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

Of all the other hash methods that have been tried, perhaps the most interesting is a technique based on algebraic coding theory; the idea is analogous to the division method above, but we divide by a polynomial modulo 2 instead of dividing by an integer. (As observed in Section 4.6, this operation is analogous to division, just as addition is analogous to exclusive-or.) For this method, M

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

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

using polynomial arithmetic modulo 2; then $h(K) = (h_{m-1} \ldots h_1h_0)_2$. If $P(x)$ is chosen properly, this hash function can be guaranteed to avoid collisions between nearly-equal keys. For example if $n = 15$, $m = 10$, and

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

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

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

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

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

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

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

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

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

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

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

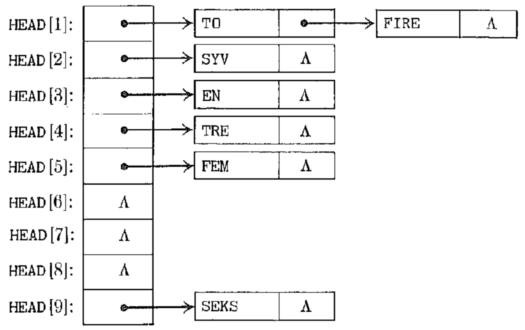

Fig. 38. Separate chaining.

to make the lists ascending, the TO and FIRE nodes of Fig. 38 would be interchanged, and all the Λ links would be replaced by pointers to a dummy record whose key is ∞. (Cf. Algorithm 6.1T.) Alternatively we can make use of the "self-organizing file" concept discussed in Section 6.1; instead of keeping the lists in order by key, they may be kept in order according to the time of most recent occurrence.

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

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

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

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

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

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

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

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

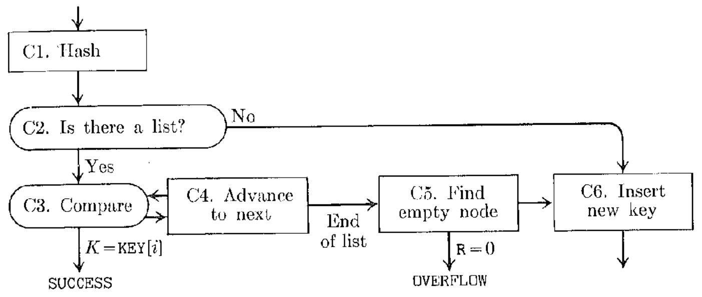

Fig. 39. Chained scatter table search and insertion.

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

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

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

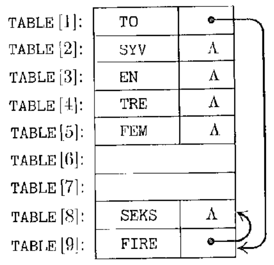

Fig. 40. Coalesced chaining.

DEF00006596

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

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

empty	−	1	0	0	0	0
occupied	+	LINK		KEY		

(13)

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

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

The running time of this program depends on

C = number of table entries probed while searching;

A = 1 if the initial probe found an occupied node;

S = 1 if successful, 0 if unsuccessful;

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

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

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

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

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

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

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

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

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

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

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

At first glance it may appear that step C5 is inefficient, since it has to search sequentially for an empty position. But actually the total number of table probes made in step C5 as a table is being built will never exceed the number of items in the table; so we make an average of at most one of these probes per insertion! Exercise 41 proves that T is approximately αe^{α} in a random unsuccessful search.

It would be possible to modify Algorithm C so that no two lists coalesce, but then it would become necessary to move records around. For example, consider the situation in Fig. 40 just before we wanted to insert SEKS into position 9; in order to keep the lists separate, it would be necessary to move FIRE, and for this purpose it would be necessary to discover which node points to FIRE. We could solve this problem without providing two-way linkage by using circular lists, as suggested by Allen Newell in 1962, since the lists are short; but that would probably slow down the main search loop because step C4 would be more complicated. Exercise 34 shows that the average number of probes, when lists aren't coalesced, is reduced to

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

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

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

On the other hand, Butler Lampson has observed that most of the space actually needed for links can actually be saved in the chaining method, if we avoid coalescing the lists. This leads to an interesting algorithm which is discussed in exercise 13.

Note that chaining can be used when $N > M$, so overflow is not a serious problem. If separate lists are used, formulas (18) and (19) are valid for $\alpha > 1$. When the lists coalesce as in Algorithm C, we can link extra items into an auxiliary storage pool; L. Guibas has proved that the average number of probes to insert the $(M+L+1)$st item is then $(L/2M+\frac{1}{4})((1+2/M)^M-1)+\frac{1}{2}$.

Collision resolution by "open addressing." Another way to resolve the problem of collisions is to do away with the links entirely, simply looking at various entries of the table one by one until either finding the key K or finding an empty position. The idea is to formulate some rule by which every key K determines a "probe sequence," namely a sequence of table positions which are to be inspected whenever K is inserted or looked up. If we encounter an open position while searching for K, using the probe sequence determined by K, we can conclude that K is not in the table, since the same sequence of probes will be made every time K is processed. This general class of methods was named *open addressing* by W. W. Peterson [*IBM J. Research & Development* **1** (1957), 130–146].

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

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

as in the following algorithm.

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

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

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

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

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

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

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

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

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

Fig. 41. Linear open addressing.

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

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