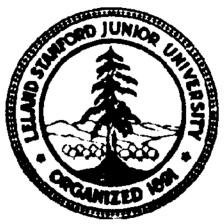
ORDERED HASH TABLES

by

OLE AMBLE
DONALD E. KNUTH

STAN-CS-73-367
June 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)
Stanford University
Dept. of Computer Science
Stanford, California 94305

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3 REPORT TITLE
Ordered Hash Tables

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)
technical report May 1973

5 AUTHOR(S) (First name, middle initial, last name)
Ole Amble and Donald E. Knuth

6 REPORT DATE
May 1973

7a. TOTAL NO. OF PAGES
approx. ~~34~~ 36

7b. NO OF REFS

8a CONTRACT OR GRANT NO
~~ONR~~ N00014-67-A-0112-0057 NR 044-402

b PROJECT NO
~~ONR N-0057~~

c

d

9a. ORIGINATOR'S REPORT NUMBER(S)
STAN-CS-73-367

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10 DISTRIBUTION STATEMENT
Releasable without limitations on dissemination

11 SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

13 ABSTRACT

Some variants of the traditional hash method, making use of the numerical or alphabetical order of the keys, lead to faster searching at the expense of a little extra work when items are inserted. ~~This~~ paper presents the new algorithms and analyzes their average running time.

DD FORM 1 NOV 1473 (PAGE 1)

Unclassified

Ordered Hash Tables

by Ole Amble and Donald E. Knuth

University of Oslo and Stanford University

Abstract: Some variants of the traditional hash method, making use of the numerical or alphabetical order of the keys, lead to faster searching at the expense of a little extra work when items are inserted. This paper presents the new algorithms and analyzes their average running time.

Keywords and phrases: Searching, hash tables, analysis of algorithms, address calculation

CR categories: 3.74, 5.3

The preparation of this paper was supported in part by Norges Almenvitenskapelige Forskningsråd, and in part by the U. S. Office of Naval Research under grant number ONR 00014-67-A-0112-0057 NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Ordered Hash Tables

Traditional methods of search are usually based either on the numerical or alphabetical ordering of keys (e.g. binary search), or on the keys' arithmetical properties (e.g. hashing). By combining these two approaches it is possible to obtain methods which are often superior to the traditional algorithms.

In this paper we shall discuss a new class of search procedures which use both the idea of ordering and the idea of "open" hash addressing. A mathematical analysis of the expected running time is also given.

Definitions

Given a file or table of data containing N distinct keys $K_1, K_2, \ldots, K_N$, the search problem consists of taking a given argument K and determining whether or not $K = K_i$ for some i. In practice the key K_i is part of a larger record of information, R_i, which is being retrieved via its key; but for the purposes of our discussion we may concentrate solely on the keys themselves, since they are the only things which significantly enter into the search algorithms. If the search argument K is not in the table, we sometimes want to put it in; therefore we are generally interested in two algorithms, one for searching and one for insertion. The recent book by Knuth (1973) contains an extensive account of the algorithms which are commonly used for searching and insertion.

One of the important families of search algorithms is the so-called method of "open addressing with double hashing", which works as follows. The table is stored in a larger array of M positions, numbered 0

through $M-1$. If U is the universe of all possible keys that might ever be sought (e.g., U might be all n-bit numbers or all n-character identifiers, for some n), we define two functions for each K in U, namely

$h(K)$ = the "hash address" of K,

$i(K)$ = the "hash increment" of K.

These functions are constrained so that $0 \le h(K) < M$ and $1 \le i(K) < M$ and $i(K)$ is relatively prime to M, for all K. Thus if $M = 2^m$, $i(K)$ is allowed to be any odd positive number less than M; alternatively if M is prime, $i(K)$ is allowed to be any positive number less than M. For best results these functions are usually chosen to be efficiently computable, yet with the property that distinct keys will tend to have different hash addresses.

Some of the M positions of the hash table are unoccupied, while N of the positions contain keys. For convenience we shall assume that all keys have a strictly positive numeric value. The entries of the hash table will be denoted by $T_0, T_1, \ldots, T_{M-1}$, where $T_j = 0$ if that position is empty and $T_j > 0$ if T_j is the key stored in position j.

Algorithms

Using these definitions, it is possible to describe the conventional algorithm for open addressing with double hashing as follows.

Algorithm A. Let K be the search argument.

Step A1. Set $j \leftarrow h(K)$.

Step A2. If $T_j = K$, the algorithm terminates 'successfully'.

Step A3. If $T_j = 0$, the algorithm terminates 'unsuccessfully'.

Step A4. Set $j \leftarrow j-i(K)$. If now $j < 0$, set $j \leftarrow j+M$.

Return to step A2. □

The search is said to be 'successful' or 'unsuccessful' according as K has been found or not. After a successful search, it is possible to fetch the entire record having the given key.

A new record may be inserted into such a table by first searching for its key K; when the algorithm terminates unsuccessfully in step A3, the new record may be placed into the j-th position of the table. Subsequent searches for this key will follow the same path to position j.

The fact that $i(K)$ is relatively prime to M ensures that no part of the table is examined twice, until all M locations have been probed. Since we assume that there is at least one empty position, the search must terminate if K is not present.

The above algorithm includes several noteworthy special cases. If $i(K)$ is identically 1 for all K, it is the well-known method of linear probing. If $i(K) = 1$ and $h(K) = M-1$ for all K, it reduces to the straightforward method of sequential scanning. If $i(K) = f(h(K))$ where f is a more-or-less random function, the algorithm is called double hashing with secondary clustering. On the other hand, if the probability that $h(K) = h(K')$ and $i(K) = i(K')$, for distinct keys K and K' in U, is $1/M\varphi(M)$, i.e., if each of the possible values of the pair $(h(K),i(K))$ is equally likely, the method is called independent double hashing.

Algorithm A makes decisions only by testing for equality vs. inequality. By using the numerical order of keys we obtain a new algorithm which is almost identical to the other:

Algorithm B. (Searching in an ordered hash table.)

Step B1. Set $j \leftarrow h(K)$.

Step B2. If $T_j = K$, the algorithm terminates 'successfully'.

Step B3. If $T_j < K$, the algorithm terminates 'unsuccessfully'.

Step B4. Set $j \leftarrow j-i(K)$. If now $j < 0$, set $j \leftarrow j+M$.

Return to step B2. □

Only step B3 has changed, and in a trivial way. Unsuccessful searches will now be faster.

Of course we cannot use Algorithm B unless the positions of the hash table have been filled in a suitable way. If the keys have been inserted in decreasing order by the ordinary method (i.e., if we start with an empty table, then insert the largest key, then the second-largest, etc.), it is easy to see that Algorithm B will work properly. This proves that there is always an arrangement of keys such that Algorithm B is valid.

Of course in practice we need to be able to insert keys in arbitrary order, as they arrive "on line". The following method can be used:

Algorithm C. (Insertion into an ordered hash table.)

Assume that $K \neq T_j$ for $0 \leq j < M$, and that $N \leq M-2$.

Step C1. Set $j \leftarrow h(K)$.

Step C2. If $T_j = 0$, set $T_j \leftarrow K$ and terminate.

Step C3. If $T_j < K$, interchange the values of $T_j \leftrightarrow K$.

Step C4. Set $j \leftarrow j-i(K)$. If now $j < 0$, set $j \leftarrow j+M$.

Return to step C2. □

During this algorithm, the variable K takes on a decreasing sequence of values, and the increments in step C4 will vary (in general). This is a

rather peculiar state of affairs, in spite of the innocuous appearance of Algorithm C, so it is helpful to look at an example.

Suppose that $M = 11$ and that there are $N = 8$ keys

145, 293, 397, 458, 553, 626, 841, 931,

where the middle digit is the h-value and the rightmost digit is the i-value; thus, $h(293) = 9$ and $i(293) = 3$. Then the keys may be distributed in the T table as follows:

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
0	0	626	931	841	553	293	0	458	397	145

The reader may verify that Algorithm B will indeed retrieve each of these keys properly. Now if we wish to insert the new key 759, Algorithm C first replaces T_5 by 759 and sets $K \leftarrow 553$; after examining $T_2 = 626$, it sets $T_{10} \leftarrow 553$, $K \leftarrow 145$; and eventually $T_0 \leftarrow 145$. The table for all nine keys is therefore

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
145	0	626	931	841	759	293	0	458	397	553

To verify that Algorithm C is correct, consider the <u>path</u> corresponding to key K, namely the sequence of table position numbers

$$h(K),\ h(K)-i(K),\ h(K)-2i(K),\ \ldots,\ h(K)-(M-1)i(K) \pmod M .$$

Since $i(K)$ is relatively prime to M, this sequence consists of the numbers $0,1,\ldots,M-1$ in some order. Algorithm B works properly if and only if, for every key $K = T_j$ in the table, we do not have $K > T_{j'}$ for some j' which appears <u>earlier</u> than j in the path corresponding to K. (This is the essential "invariant" which is relevant to formal proofs of Algorithm B.) Since Algorithm C never decreases the value of any table position, it preserves this condition.

Analyses

Now let us attempt to determine how much faster (if at all) the new algorithms will go. The following uniqueness theorem is very helpful in this regard.

Theorem. A set of N keys $K_1,\ldots,K_N$ can be arranged in a table $T_0,T_1,\ldots,T_{M-1}$ of $M > N$ positions in one and only one way such that Algorithm B is valid.

Proof. We have observed that at least one arrangement is possible. Suppose that there are at least two, and let K_j be the largest key which appears in different positions in two different arrangements. Thus, all keys larger than K_j occupy fixed positions in all possible arrangements. If we look at the path corresponding to K_j, as defined above, the positions of keys larger than K_j are predetermined; and all keys smaller than K_j must occur later than K_j. Therefore K_j must occupy the first vacant place in its path, after the larger keys, contradicting the assumption that K_j can appear in different places. □

In order to know the behavior of these search algorithms, we want to know the corresponding average number of iterations or probes in the table, i.e., the average number of times steps A2, B2, or C2 are performed respectively. (Only the average number is generally considered in discussions of hashing, since the worst case is too horrible to contemplate.)

The classical Algorithm A has been extensively investigated (see Knuth (1973) for a review of the literature), and the results can be summarized as follows. Let $\alpha = N/M$ be the 'load factor' of the hash table. Let A_N be the average number of times step A2 is performed in a random

successful search, and let A'_N be the corresponding number in a random unsuccessful search. By 'random' and 'average' we mean that the hash addresses of the keys are assumed to be independent and uniformly distributed in the range 0 through M-1, and that each of the N keys of the table is equally likely in a successful search. Then the following approximate formulas have been derived, as M and N approach infinity:

Increment method	A_N	A'_N
linear probing	$\frac{1}{2}(1 + (1-\alpha)^{-1})$	$\frac{1}{2}(1 + (1-\alpha)^{-2})$
secondary clustering	$1 - \ln(1-\alpha) - \frac{1}{2}\alpha$	$(1-\alpha)^{-1} - \ln(1-\alpha) - \alpha$
independent double hashing	$-\alpha^{-1} \ln(1-\alpha)$	$(1-\alpha)^{-1}$

Since the number of probes needed to retrieve an item with Algorithm A is the same as the number needed to insert it, the average number of probes needed to find the k-th item inserted is A'_{k-1}. It follows that

$$A_N = (A'_0 + A'_1 + \ldots + A'_{N-1}) / N \quad . \tag{1}$$

Now let us consider the performance of Algorithm B. We shall assume that there is no significant correlation between the hash addresses and the numerical ordering of the keys. Since the position of any fixed set of keys in the table is unique, we may as well assume that they have been inserted in decreasing order. Then the insertion algorithm is identical to that used with Algorithm A, and the average number of probes needed to find the k-th largest item is A'_{k-1}. It follows that

$$B_N = (A'_0 + A'_1 + \ldots + A'_{N-1}) / N = A_N \quad . \tag{2}$$

In other words, Algorithm B is equivalent to Algorithm A with respect to successful searching, on the average.

In an unsuccessful search with Algorithm B, the number of probes is the same as would be required in a successful search if the keys were $\{K_1, K_2, \ldots, K_N, K\}$ instead of $\{K_1, K_2, \ldots, K_N\}$. Therefore

$$B'_N = B_{N+1} = A_{N+1} \quad . \tag{3}$$

The above formulas for A_N and A'_N show that this is indeed an improvement. For example, when $\alpha = .90$ (i.e., when the table is 90 percent full), the quantities for unsuccessful search are

increment method	A'_N	B'_N
linear probing	50.5	5.500
secondary clustering	11.4	2.853
independent double hashing	10.0	2.558

As $\alpha \to 1$, the ratio B'_N/A'_N approaches 0.

Finally let us investigate the new cost of insertion with Algorithm C. Let C_N be the average number of times step C2 is performed when inserting the N-th item. Each time we execute step C2, we increase by one the total number of probes needed to find one of the keys. Thus, if we sum over N insertions, we must have

$$C_1 + \ldots + C_N = NA_N \quad .$$

This equation together with (1) implies that

$$C_N = A'_{N-1} \quad . \tag{4}$$

In other words, the average number of probes needed to insert a new item is exactly the same as it was with Algorithm A.

It is worth noting that the probability distribution of C_K is not in general the same as that of A'_{N-1}, although the average value is the same. In fact, a single insertion with Algorithm C might take up to order N^2 iterations (although such an event is extremely rare). Consider again the case of three-digit keys whose middle digit is the h-value and whose rightmost digit is the i-value; and let $M = 10$. Then the insertion of 941 into the table

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
101	311	521	731	849	659	469	279	0	0

is amazingly slow, as the reader may verify. In general, the table might contain n keys in "organ-pipe order",

$$0 = T_n < T_0 < T_{n-1} < T_1 < \cdots < T_{\lfloor n/2 \rfloor} ,$$

and we might have

$$i(T_j) = \begin{cases} +1 , & \text{for } 0 \le j < \lceil n/2 \rceil , \\ M-1 , & \text{for } \lceil n/2 \rceil \le j < n ; \end{cases}$$

then the insertion of a new largest key whose hash address is $\lfloor n/2 \rfloor$ will take maximum time, namely $(n+1)n/2+1$ iterations of step C2.

We have now analyzed the average number of iterations in both Algorithms B and C. The analysis isn't complete, however, because we have not determined the average number of <u>interchanges</u> performed in step C3. This is an important consideration, since it is the number of times we need to compute an increment $i(K)$; with Algorithm A, the increment needs to be computed only once. Therefore let D_N be the average number of times the operation $T_j \leftarrow K$ is performed in step C3 while inserting the N-th item.

Unfortunately the analysis of D_N is complicated, and we must defer the calculations to Appendix 1. It turns out that D_N is approximately $(1-\alpha)^{-1} + \alpha^{-1} \ln(1-\alpha)$ for linear probing, and approximately equal to $A_N - 1$ for independent double hashing.

Further Development

The above algorithms can be extended in various ways, to gain further improvements. For example, it is easy to see that the ideas can immediately be generalized to the case of external searching, where each of the M table positions is a "bucket" containing b or less keys for some given b.

Another type of extension will make unsuccessful searching still faster, at the expense of M more bits of memory. Let $B_0, B_1, \ldots, B_{M-1}$ be a vector of bits with all B_j initially 0. Suppose that we set $B_j \leftarrow 1$ in step C3 of the insertion algorithm, so that $B_j = 1$ if and only if some successful search "passes through" position j. Then if the search algorithm ever gets to step B3 and finds $B_j = 0$, the search must be unsuccessful.

This extra-bit approach applies, of course, to unordered hash tables as well as ordered ones, but it is especially attractive in the ordered case because the extra testing can be done with almost no cost. We can combine the bit test with the ordinary test if we assume that each bit B_j appears at the left of T_j as a new significant bit. Then Algorithm B can be rewritten as follows.

Step B1. Set $j \leftarrow h(K)$.

Step B2. If $(B_j,T_j) < (1,K)$, then the algorithm terminates successfully or unsuccessfully according as $T_j = K$ or not.

Step B3. If $(B_j,T_j) = (1,K)$, then the algorithm terminates successfully.

Step B4. Set $j \leftarrow j-i(K)$. If now $j < 0$, set $j \leftarrow j+M$. Return to step B2. □

Only steps B2 and B3 have changed, and the change is such that the computer time per iteration is the same as before; there is just a little more calculation at the end of a successful search, plus the cost of attaching a 1 at the left of the input argument K when the search begins.

The average number of probes per unsuccessful search with this modified algorithm appears to be difficult to analyze, but the empirical data in Table 1 at the end of this paper shows that the idea can be worthwhile. Of course the number of probes per successful search is unaffected by the extra bits.

So far none of the ideas mentioned have been of any use in the case of successful search. One possibility which suggests itself is to start searching one place ahead (i.e., to start at position $h(K)-i(K)$), because this will save one probe if K is not at its hash address, and because we will be able to test whether K is in position $h(K)$ if the first search is unsuccessful. Since we have greatly improved the ability to detect unsuccessful searches, we can perhaps use some of this capability in connection with successful searches.

Unfortunately, a more careful analysis shows that such an idea is unsound; it actually <u>increases</u> the average number of probes for both successful and unsuccessful searching. (See Appendix 2.) There is,

however, a case in which it does work, namely if we force $h(K)$ to be correlated with the magnitude of the table entry for K. Suppose we have a hash function such that

$$K \le K' \quad \text{implies} \quad h(K) \le h(K') \ ,$$

and suppose further that we are using linear probing (i.e., that $i(K)$ is identically 1). Then it is not hard to see that the correlation causes the number of probes for successful search in an ordered hash table to have a much smaller variance; there will be fewer keys requiring very small or very large numbers of probes, although the average number will remain unchanged. Appendix 2 shows that this "start one ahead" approach will lead to less probes per successful search when the table is more than about 64.38 percent full. (The limiting value $\alpha = 0.643797758$, where the one-ahead method begins to excel, is the root of $2(1-\alpha)(e^{\alpha}-1) = \alpha$.)

An obvious problem arises, however, if we want the hash function to correlate with the keys in this way. Our options for the choice of hash function will be so drastically reduced that it will probably be impossible to find an efficiently computable $h(K)$ that works well with typical sets of keys. A solution to this dilemma is achieved if we store transformed keys in the T table, instead of the keys themselves. Thus, let $t(K)$ be any function which scrambles keys without loss of information:

$$t(K) = t(K') \quad \text{implies that} \quad K = K' \ .$$

Then we can store $t(K_1), t(K_2), \ldots$ in the table, and search for $t(K)$ instead of K. We can now achieve the desired correlation between $h(K)$ and $t(K)$ by letting $h(K)$ be the leading bits of $t(K)$.

For example, if M is a prime number and if $h(K) = K \bmod M$, we can let $t(K)$ be a packed binary number whose leftmost bits are $h(K)$

and whose rightmost bits represent the quotient $\lfloor K/M \rfloor$. This transformed key $t(K)$ is one bit larger than the original key. Alternatively if $M = 2^m$ is a power of 2 , we may let $t(K) = (aK) \bmod 2^w$, where w is the key length and a is any odd number; then $h(K)$ may be chosen as the leading m bits of $t(K)$.

The reader may justifiably feel at this point that the method is getting "baroque". The last few paragraphs have discussed detailed refinements which are mildly interesting, but they can obviously never save more than one probe per search. Therefore the reader may wonder why we are going on and on, "beating a dead horse". The answer is that it was precisely the above train of thought, together with hand simulations on random numbers, which led us to consider another algorithm which <u>does</u> offer a substantial improvement. We shall now discuss this improved algorithm, which uses the correlation between hash addresses and table entries in a somewhat different fashion.

<u>Bidirectional Linear Probing</u>

Let $t(K)$ be any one-to-one transformation of keys:

$$t(K) = t(K') \quad \text{implies} \quad K = K' \ .$$

Furthermore let $h(K)$ be a hash function such that

$$t(K) \le t(K') \quad \text{implies} \quad h(K) \le h(K') \ .$$

We have already discussed practical ways of finding such functions; and it is natural to assume that a hash method using such transformations would keep the nonempty positions of the hash table in sorted order:

$$T_i \neq 0 \ \text{ and } \ T_j \neq 0 \ \text{ and } \ i < j \quad \text{implies} \quad T_i < T_j \ .$$

Consider now the following straightforward search procedure:

Algorithm X. (Bidirectional linear probing.)

Step X1. Set $j \leftarrow h(K)$, and set $K \leftarrow t(K)$.

Step X2. If $T_j = K$, the algorithm terminates 'successfully'. If $T_j > K$, go to step X5 (downward search). If $T_j = 0$, the algorithm terminates 'unsuccessfully'. Otherwise go to step X3 (upward search).

Step X3. (At this point, $0 < T_j < K$.) Set $j \leftarrow j+1$.

Step X4. If $T_j = K$, the algorithm terminates 'successfully'. If $T_j = 0$ or $T_j > K$, the algorithm terminates 'unsuccessfully'. Otherwise return to step X3.

Step X5. (At this point, $T_j > K$.) Set $j \leftarrow j-1$.

Step X6. If $T_j = K$, the algorithm terminates 'successfully'. If $T_j < K$, the algorithm terminates 'unsuccessfully'. Otherwise return to step X5.

This algorithm searches either up or down depending on the result of the first comparison. Its validity depends on having a table T_j whose nonempty entries are ordered as stated above, having the additional property that no empty space occurs between the location of any transformed key and its hash address. Furthermore there must be empty positions at the ends of the table; we can take care of this by extending the boundaries so that $T_{-1} = T_M = 0$.

In this case there are, in general, many configurations of the T's which will guarantee correct retrieval. For example, suppose that $M = 10$ and consider the transformed keys 614, 621, 637, 641, 647, 698, 841, where $h(K)$ is the leading digit. (It is not typical to have so many keys with the same hash address, but our intent is to give a small example

which exhibits some of the more interesting things that can happen.)
If we use the ordinary method of linear probing (Algorithm B), the table is filled thus:

j =	0	1	2	3	4	5	6	7	8	9
T_j =	0	614	621	637	641	647	698	0	841	0
probes		6	5	4	3	2	1		1	

The bottom line shows how many table entries are examined when searching for T_j ; i.e., it takes 4 probes to find '637', since we start at T_6 . Algorithm X allows us to rearrange the T_j's so that many of the keys will be found sooner:

j =	0	1	2	3	4	5	6	7	8	9
T_j =	0	0	0	614	621	637	641	647	698	841
probes				4	3	2	1	2	3	2

The search for '841' goes upwards now, but we save two probes when searching for '614'. The average number of probes per successful search is reduced from $(6+5+4+3+2+1+1)/7 = 22/7$ to $(4+3+2+1+2+3+2)/7 = 17/7$.

Appendix 5 shows how to characterize the optimum arrangements of the T_j's , for any given set of keys, i.e., those arrangements which minimize the average number of probes per successful search by Algorithm X. As a consequence of the theory developed there, we may use the following algorithm to insert into a bidirectional hash table, maintaining optimum arrangements at all times.

Algorithm Y. (Optimum insertion for bidirectional linear probing.) In this algorithm, let $h'(T_j)$ be $h(t^{-1}(T_j))$; thus if $T_j = t(K_j)$ then $h'(T_j) = h(K_j)$.

Step Y1. Set $j \leftarrow h(K)$, $K \leftarrow t(K)$.

Step Y2. If $T_j = 0$, set $T_j \leftarrow K$ and terminate the algorithm.

Step Y3. Set p to the largest index $< j$ such that $T_p = 0$. Set q to the smallest index $> j$ such that $T_q = 0$.

Step Y4. Set $j \leftarrow q$. Then if $T_{j-1} > K$, repeatedly set $T_j \leftarrow T_{j-1}$ and $j \leftarrow j-1$, until $T_{j-1} < K$. Finally set $T_j \leftarrow K$. (Thus, K has been sorted into the proper place with respect to the other transformed keys.)

Step Y5. Set $d \leftarrow 0$. Then for $j \leftarrow p+1, p+2, \ldots, q$ (in this order), repeatedly set

$$d \leftarrow d+1 \quad \text{if} \quad h'(T_j) \geq j \ ,$$

$$d \leftarrow d-1 \quad \text{if} \quad h'(T_j) < j \ .$$

If at any time during this process d becomes negative, go immediately to step Y6 without finishing the loop. But if d remains ≥ 0 throughout the entire loop, terminate the algorithm.

Step Y6. Set $T_j \leftarrow T_{j+1}$ for $p \leq j < q$, and set $T_q \leftarrow 0$.

Algorithm Y finds the smallest block of consecutive nonempty locations containing position $h(K)$, and inserts $t(K)$ into this block by shifting the transformed keys which are larger. Then step Y5 is used to decide whether or not it would have been better to shift the transformed keys which are smaller; if so, step Y6 moves the whole block down. (Empirical tests show that step Y6 is required only about $1/4$ as often as step Y5.)

In order to use this algorithm, a dozen or so extra table positions T_j should be included for $j < 0$ and for $j \geq M$, to avoid end effects. (There are several ways to make the algorithm cyclically symmetric modulo M, but these are more complicated and time-consuming than simply to provide extra "breathing space" at both ends. The optimum arrangement rarely spills over very far; in our experiments with $M = 4096$ and tables 95 percent full, no more than five locations were needed at either end.)

The theory of linear probing shows that this insertion method isn't extremely slow; the average size $q-p$ of the block of keys considered when the $(N+1)$-st key is being inserted will be $2A'_N-2 \approx (1-\alpha)^{-2}-1$ when $N/M = \alpha$. (Cf. Knuth (1973), exercise 6.4-47.) When this size is averaged over N insertions, it reduces to $2A_N-2 \approx \alpha/(1-\alpha)$. Thus, insertion by Algorithm Y is only four or five times slower than insertion by the classical linear probing algorithms. On the other hand, empirical results (see Table 1) show that retrieval by Algorithm X is significantly better than classical linear probing.

Conclusions

Traditional hash methods are comparatively slow with respect to unsuccessful search. By extending them to make use of the inherent ordering of keys, we have shown that the time for unsuccessful search can be significantly reduced.

Two main algorithms have been presented in this paper. First we discussed Algorithm B, and the corresponding Algorithm C for insertion. This method reduced the time for unsuccessful search to the time for successful search, without significantly increasing the cost per insertion. Therefore it is attractive for applications in which unsuccessful searches

are common. A refinement, adding "pass bits", makes unsuccessful search even faster. However, the method is never useful in typical compiler or assembler applications, where unsuccessful searches are almost always followed by insertions.

The second method we have discussed is Algorithm X, together with the corresponding Algorithm Y for insertion. Here both successful and unsuccessful search times are reduced, at the expense of greater insertion time and slightly more complex programs. (The method may be compared with a scheme recently published by Brent (1973); his method requires less probes than ours on successful searches, but it does not reduce the unsuccessful search time.)

Table 1 presents the behavioral characteristics of the algorithms discussed here, assuming random hash functions. Some of the results have been derived by theoretical analyses; these are shown to three decimal places. The other results, for which only one decimal place of accuracy appears in the table, have not yet been verified theoretically. Every entry in Table 1 is the number of probes per search, i.e., the number of T_j entries examined. This information can be used to predict the behavior of each algorithm; but it should be emphasized that the time per probe and the setup time will vary from one method to another. For example, linear probing and Algorithm X will have faster inner loops than independent double hashing, while the latter (especially with "pass bits") involves fewer probes. Thus the number of probes is not an absolute measure of goodness, the entire algorithm must be considered when making comparisons.

Method	Successful search							Unsuccessful search						
100 α (percent full)	25	50	75	80	85	90	95	25	50	75	80	85	90	95
Alg. A, linear probing	1.167	1.500	2.500	3.000	3.833	5.500	10.500	1.389	2.500	8.500	13.000	22.722	50.500	400.500
Alg. A, secondary clustering	1.163	1.443	2.011	2.209	2.472	2.853	3.521	1.371	2.193	4.636	5.810	7.715	11.402	22.045
Alg. A, indep. double hashing	1.151	1.386	1.848	2.012	2.252	2.558	3.153	1.333	2.000	4.000	5.000	6.667	10.000	20.000
Alg. B, linear probing	1.167	1.500	2.500	3.000	3.833	5.500	10.500	1.167	1.500	2.500	3.000	3.833	5.500	10.500
Alg. B, secondary clustering	1.163	1.443	2.011	2.209	2.472	2.853	3.521	1.163	1.443	2.011	2.209	2.472	2.853	3.521
Alg. B, indep. double hashing	1.151	1.386	1.848	2.012	2.252	2.558	3.153	1.151	1.386	1.848	2.012	2.252	2.558	3.153
Alg. B, linear, with pass bits	1.167	1.500	2.500	3.000	3.833	5.500	10.500	1.0	1.2	2.0	2.4	3.6	5.4	10.3
Alg. B, indep., with pass bits	1.151	1.386	1.848	2.012	2.252	2.558	3.153	1.0	1.1	1.3	1.4	1.6	1.7	2.2
Alg. B, linear, correlated, one ahead	1.871	1.797	2.245	2.613	3.306	4.825	9.667	2.0	2.2	2.9	3.3	4.0	5.8	11.0
Alg. X, bidirectional linear	1.1	1.3	1.7	2.0	2.3	2.9	4.2	1.3	1.5	2.1	2.3	2.6	3.1	4.4

Table 1. Average number of probes required by the algorithms, as a function of the load factor $\alpha = N/M$.

Appendix 1. Analysis of step C3.

In order to analyze the quantity D_N defined in the text, let us assume that the keys are $K_1 < K_2 < \ldots < K_N$. Let D'_N be the average number of times, during N random insertions, that the variable K is set to the smallest key K_1 at some time during the insertion process. In other words, D'_N is the average number of times K_1 is "moved". Then D'_{N-1} is the average number of times K_2 is moved, since the behavior of the algorithm on $\{K_2,\ldots,K_N\}$ is essentially independent of K_1. Similarly, D'_{N+1-i} is the average number of times K_i is moved. Therefore

$$D_1 + \ldots + D_N = (D'_1-1) + \ldots + (D'_1-1)$$

for all N, and we have

$$D_N = D'_N-1 \quad .$$

Consider now the case of independent double hashing. Experience shows (but it has not been rigorously proved) that this case is satisfactorily approximated by uniform hashing, where each key's path is a random permutation of $\{0,1,\ldots,M-1\}$, independent of all other keys. Under this assumption, which has been tacitly made in the text, the analysis of hashing algorithms usually becomes quite easy. However, the "organ pipe" example of the text indicates some of the complexities of Algorithm C, and a rather indirect approach to the analysis of D_N (or D'_N) seems to be necessary.

Let D''_N be the probability that the smallest key K_1 is moved during the insertion of the N-th key. It follows that

$$D'_N = \frac{D''_1 + 2D''_2 + \ldots + ND''_N}{N} \quad ,$$

since the probability that K_1 is moved on the j-th insertion is D''_j times j/N, the probability that K_1 appears among the first j keys inserted.

Consider the entire sequence of actions which occur when the keys $\{K_1,\ldots,K_N\}$ are inserted into the table in decreasing order. This sequence of actions states for example that, when K_j was inserted, a certain sequence of larger keys were encountered before an empty place was found. We shall call the elements of the latter sequence the dominators of K_j. Knowing all the sequences of dominators, in the decreasing-order case, we can deduce what actions will occur when the keys are inserted in any other specified order. Define the function p on the indices $\{2,\ldots,N\}$ such that, if K_j is the last of $\{K_2,\ldots,K_N\}$ to be inserted, then $K_{p(j)}$ will be the last of $\{K_2,\ldots,K_N\}$ to be moved. Now the very last insertion moves K_1 if and only if either (i) K_1 was the last element inserted, or (ii) K_j was the last element inserted, for some $j \geq 2$, and $K_{p(j)}$ is one of the dominators of K_1.

For example, suppose $N = 3$, so that $K_3 > K_2 > K_1$. If K_3 is a dominator of K_2, we have $p(3) = p(2) = 2$, and K_1 is moved on the third insertion if and only if it is the last to be inserted or it is dominated by K_2. On the other hand if K_3 does not dominate K_2, then $p(3) = 3$ and $p(2) = 2$; hence K_1 is moved on the third insertion if and only if it is either the last to be inserted, or it is dominated by the last to be inserted.

For any fixed choice of dominator sequences on $\{K_2,\ldots,K_N\}$, and for fixed $j \geq 2$, the probability that $K_{p(j)}$ dominates K_1 is a function only

of M and N, independent of j and the given actions, because of the assumptions of uniform hashing. This probability may be expressed as

$$\frac{1}{N-1} \sum_{r \geq 1} (r-1)\, P_r \quad ,$$

where P_r is the probability that K_1 has $r-1$ dominators, since exactly $\binom{N-2}{r-2} \Big/ \binom{N-1}{r-1} = (r-1)/(N-1)$ of the possible choices of $r-1$ dominators include the given key $K_{p(j)}$. Since P_r is also the probability that r probes are needed to insert the N-th item by Algorithm A, we have

$$\frac{1}{N-1} \sum_{r \geq 1} (r-1)\, P_r = \frac{1}{N-1} (A'_{N-1} - 1) \quad .$$

The probability that K_j is inserted last is $1/N$; summing for $2 \leq j \leq N$, and adding $1/N$ for the case that K_1 comes last, gives

$$D''_N = \frac{N-1}{N} \left(\frac{1}{N-1} (A'_{N-1} - 1) \right) + \frac{1}{N} = \frac{1}{N} A'_{N-1} \quad .$$

The above formulas now yield the desired answer,

$$D_N = A_N - 1 \quad .$$

Such a simple result deserves a simpler proof; however, it is surprisingly easy to derive this formula by plausible but fallacious arguments, and the above approach is the only reliable one for this analysis that is known to the authors.

We come finally to the case of linear probing. This is much more complicated, and the derivation will only be sketched here. Consider the M^n 'hash sequences' $a_1 \ldots a_n$ to be equally likely, where the k-th key inserted has $h(K) = a_k$. Then the probability that the $(n+1)$-st key inserted moves K_1 and is not itself K_1 is

$$\frac{1}{N} \sum_{\substack{1 \le k \le n \\ 1 \le i \le n}} k a(M,n,k,i) M^{-n} \quad , \qquad (*)$$

where $a(M,n,k,i)$ denotes the number of hash sequences $a_1 \ldots a_n$ which cause T_0 through T_{k-1} to be occupied, T_k to be empty, and $T_0 = K_1$ if the smallest key K_1 is the i-th to be inserted. Let $g(M,n,k)$ be the number of hash sequences which cause T_0 through T_{k-1} to be occupied and $T_{M-1} = T_k = 0$, and let $f(M,n)$ be the number of hash sequences which cause $T_{M-1} = 0$. Then the formulas

$$f(m,n) = (m-n)m^{n-1} \quad , \quad g(m,n,k) = \binom{n}{k} f(k+1,k) f(m-k-1,n-k)$$

can be derived by simple arguments (see Knuth (1973), p. 529). Also let b_n be the number of hash sequences $a_1 \ldots a_n$ which cause $T_0, \ldots, T_{n-1}$ to be occupied, and for which the "pass bit" B_j is set to 1 for $0 < j \le n-1$. From the relation

$$f(n+1,n) = \sum_{0 \le k < n} \binom{n}{k} f(k+1,k) b_{n-k}$$

and Abel's binomial formula, we deduce that $b_n = (n-1)^{n-1}$. Now the value of (*) may be expressed as

$$\frac{1}{N} \sum_{1 \le j \le n} \binom{n}{j} j b_j \sum_{\ell \ge 0} (j+\ell) g(M-j+1,n-j,\ell) M^{-n} \qquad (**)$$

because we obtain each sequence enumerated by $a(M,n,k,i)$ by piecing together, in $\binom{n}{j}$ ways, a sequence enumerated by b_j and a sequence enumerated by $g(M-j+1,n-j,\ell)$, where $1 \le i \le j$ and $k = j+\ell$. The sum (**) can be evaluated as described in Knuth (1973), page 691, exercise 27; the result is

$$\frac{1}{N}\left(\frac{n}{M} + 2\,\frac{n(n-1)}{M^2} + 3\,\frac{n(n-1)(n-2)}{M^3} + \cdots\right) ,$$

essentially an incomplete gamma function. Summing for $0 \leq n < N$, and adding 1 for when K_1 is inserted, yields the desired result

$$D'_N = 1 + \frac{1}{2}\,\frac{N-1}{M} + \frac{2}{3}\,\frac{(N-1)(N-2)}{M^2} + \frac{3}{4}\,\frac{(N-1)(N-2)(N-3)}{M^3} + \cdots .$$

Appendix 2. Starting one place ahead.

Consider the case of linear probing in an ordered hash table, when $h(K)$ is uncorrelated with the magnitude of K. Let P_r be the probability that exactly r probes are needed to find the $(n+1)$-st largest key, for some fixed value of n. Then P_r is the probability that the positions occupied by the n largest keys include $h-1,h-2,\ldots,h-r+1$, but not position $h-r$, given any h; and P_{r+1} is the probability that $h,h-1,\ldots,h-r+1$ (but not $h-r$) are included. Hence P_r-P_{r+1} is the probability that $h-1,\ldots,h-r+1$ are occupied, but neither h nor $h-r$, for any given h. It follows that the expected number of probes needed to locate the $(n+1)$-st largest key K, if we begin searching at location $h(K)-1$ instead of $h(K)$, is

$$\sum_{r\ge 2}(r-1)P_r + \sum_{r\ge 1}(r+1)(P_r-P_{r+1}) = P_1 + \sum_{r\ge 1} r P_r \quad .$$

This always exceeds $\sum r P_r$, which is the corresponding average if we begin searching at $h(K)$.

Essentially the same argument applies to uniform hashing. So we may conclude that it is not a good idea to start probing at location $h(K)-i(K)$.

However, the situation is considerably different when $h(K)$ is correlated with K, so that $h(K) \le h(K')$ whenever $K < K'$, since then T_{h-1} is almost always less than T_h. In order to analyze this situation, let us look first at the case that j never goes from 0 to $M-1$ during a successful search. (In other words, the "pass bit" B_0 is 0.) Then the nonzero T_j's are sorted; hence if we start a search at $h(K)-1$, we will lose only one probe when K is in its "home position" $h(K)$ while we save one probe whenever K is not. It

follows that the one-ahead method is favorable, for successful searching, whenever the number of keys in home position is less than $\frac{1}{2}N$.

An assumption which greatly simplifies the analysis when $B_0 \neq 0$ is to restore cyclic symmetry, by assuming that keys which passed from position 0 to position M-1 behave subsequently as if they are larger than keys which haven't. Under this assumption we shall prove below that the average number of keys in home position is exactly

$$(M-N)\left(\left(1+\frac{1}{M}\right)^{N-1}-1\right)+\left(1+\frac{1}{M}\right)^{N-1} \quad .$$

For $N/M = \alpha$ as $M \to \infty$, this number is approximately

$$(1-\alpha)\,\frac{e^{\alpha}-1}{\alpha}\,N \quad ;$$

curiously as $N \to M$ it drops to approximately e .

Without the above symmetry assumption, the number of keys in home position might be drastically different. For example, when $M = 10$ and $N = 8$, the hash sequence 9 8 7 6 5 1 1 1 leaves only one element in home position under cyclic symmetry, but there will be six keys in home position in the true ordered hash table. However, the average effect of this correction is bounded by the length of search A'_N for the smallest element, and for $N/M = \alpha < 1$ the correction is asymptotically negligible. Similarly we may ignore the fact that the search for a key in home position T_0 might be adversely affected by the presence of larger keys in T_{M-1} , T_{M-2} , etc.

To prove the formula for keys in home position under cyclic symmetry, we can observe that the number of hash sequences $a_1 \ldots a_N$ which leave an element in home position M-1 is exactly

$$f(M+1,N) - f(M,N) \quad ,$$

in the notation of Appendix 1. For if we add the $f(M,N)$ hash sequences which leave T_{M-1} empty, we obtain all the $f(M+1,N)$ hash sequences which would leave T_M empty in a linearly-probed hash table of size $M+1$. Therefore the average total number of elements in home position, under cyclic symmetry, is

$$M(f(M+1,N) - f(M,N)) / M^N \quad .$$

The formula for f, given in Appendix 1, completes the proof.

It is interesting to study the cyclically symmetric algorithm further, to find the average number of elements displaced exactly d locations from their home position when $h(K)$ correlates with K. Let $h(M,n,k)$ be the number of hash sequences $a_1 \ldots a_n$ for which $\le k$ elements pass from position 0 to $M-1$ when they are inserted. Then, by considering the number of such sequences containing exactly j zeroes, we obtain the recurrence

$$h(m,n,k) = \sum_j h(m-1,n-j,k+1-j)$$

for all $m,n,k \ge 0$. Furthermore we have the initial conditions

$$h(m,n,0) = f(m+1,n) = (m+1-n)(m+1)^{n-1} \quad ,$$

from which it is possible to derive the general formula

$$h(m,n,k) = (m+1+k-n) \sum_{0 \le r \le k} \binom{n}{r} (m+k+1-r)^{n-1-r} (r-k-1)^r$$

for all $m,n,k \ge 0$. (Abel's binomial identity shows that this sum equals m^n whenever $k \ge n$.)

The hash sequence $a_1 \ldots a_N$ produces a key with home address 0 and displacement d if and only if it is a sequence with $\ge d$ keys passing from 0 to $M-1$ but $\le d$ passing from 1 to 0. The number

of such hash sequences, when $d > 0$, is

$$h(M,N,d) - h(M,N,d-1) \quad ,$$

because $h(M,N,d)$ is the number of hash sequences with $\leq d$ keys passing from 1 to 0, while $h(M,N,d-1)$ is the number with $< d$ passing from 0 to M-1 and (consequently) $\leq d$ from 1 to 0. It follows that the average total number of keys with displacement $d > 0$ is

$$M(h(M,N,d) - h(M,N,d-1)) / M^N \quad .$$

It would be interesting to obtain asymptotic data about this probability distribution. When $M = N$, the same formulas arise in connection with the classical Kolmogorov-Smirnov tests for random numbers: the quantity $h(n,n,k-1)/n^n$ is the probability that the so-called statistic K_n^+ is $\leq k/\sqrt{n}$. According to a theorem of N. V. Smirnov in 1939, we have

$$\lim_{n\to\infty} \frac{h(n,n,s\sqrt{n})}{n^n} = 1 - e^{-2s^2} \quad ;$$

cf. Knuth (1969), p. 51.

Appendix 5. Optimum bidirectional linear probing.

Given N keys $K_1 < \ldots < K_N$ and corresponding hash addresses $0 \le h(K_1) \le \ldots \le h(K_N) < M$, we wish to place them into table positions so that K_j appears in $T_{p(j)}$ for $1 \le j \le N$, where $p(1) < \ldots < p(N)$. Writing $h_j = h(K_j)$, we wish to find a placement which is optimum, in the sense that the sum

$$\sum_{1 \le j \le N} |h_j - p(j)|$$

is minimized. We shall call this sum the "cost" of the placement. For convenience in exposition, we shall allow the positions $p(j)$ to be negative or greater than M, although the proofs could easily be extended to characterize the optimum arrangements subject to $p(1) \ge x$ and $p(N) \le y$, for any desired bounds $x \le 0$ and $y \ge M-1$. Algorithm X requires a placement such that all positions between h_j and $p(j)$ are occupied, for each j; however, we may ignore this condition, because all optimum placements automatically satisfy it.

Given any placement $p(1) < \ldots < p(N)$, we shall say that a block $[a,b]$ is a set of consecutive positions which are occupied by K_a through K_b (i.e., $p(j+1) = p(j)+1$ for $a \le j < b$). An up-block is a block followed by an empty position, which would lead to less cost if it were shifted one place higher; in other words, it is a block $[a,b]$ such that the shifted placement p' has less cost, where

$$p'(j) = \begin{cases} p(j)+1, & \text{if } a \le j \le b; \\ p(j), & \text{otherwise.} \end{cases}$$

By the definition of cost, we find that $[a,b]$ is an up-block if and only if

(a) either $b = N$ or $p(b+1) > p(b)+1$;

and

(b) the number of j in the range $a \le j \le b$, for which

$h_j > p(j)$,

exceeds the number for which $h_j \le p(j)$.

Thus it is easy to test a given placement for the presence of up-blocks. A down-block is defined similarly.

An optimum placement will, of course, contain neither up-blocks nor down-blocks. Conversely, this condition of local optimality is sufficient for global optimality:

Theorem. Given N hash addresses $h_1 \le \dots \le h_N$, a placement $p(1) < \dots < p(N)$ is optimum if and only if it contains no up-blocks and no down-blocks.

Proof. Let p be an arbitrary placement; we want to prove that p either contains an up-block, or a down-block, or is optimum. Let p' be an optimum placement. If $p(j) = p'(j)$ for all j , we are done. Otherwise suppose that $p(j_0) \ne p'(j_0)$; by symmetry we may assume that $p(j_0) < p'(j_0)$.

It would be nice if we could prove that $p(j_0)$ is part of an up-block, under these hypotheses. However, the following example shows that the argument cannot be quite so trivial:

k =	1	2	3	4	5	6	7	8
h_k =	4	4	4	4	4	6	6	6
$p(k)$ =	0	1	2	3	4	6	7	8
$p'(k)$ =	2	3	4	5	6	7	8	9

If $j_0 = 6$, we have $p(j_0) < p'(j_0)$, but $p(j_0)$ is actually part of a down-block.

We can circumvent such difficulties by arguing as follows. Let a' be minimal so that $[a',j_0]$ is a block in placement p'. Then let b be maximal so that $[a',b]$ is a block in placement p. Then let a be minimal so that $[a,b]$ is a block in placement p'. (In the above example, when $j_0 = 6$, we will have $a' = 1$ and $b = 5$ and $a = 1$.) In general we will always have $p(a') < p'(a')$, $p(b) < p'(b)$, and $[a,b]$ will always be a block in both placements; thus, $p'(j) - p(j)$ has a constant value $t \geq 1$ for $a \leq j \leq b$. Furthermore, position $p(b)+1$ is empty in placement p, while $p'(a)-1$ is empty in placement p'.

Let d_+ be the number of displacements $h_j - p(j)$ in block $[a,b]$ that are positive for placement p; also let d_- be the number of negative displacements in the block, and let d_k be the number of displacements which equal k. Define d'_+, d'_-, and d'_k similarly for placement p'. It follows that $d_k = d'_{k-t}$ for all k.

Now $[a,b]$ is an up-block for p if and only if $d_+ > d_0 + d_-$, and it is a down-block for p' if and only if $d'_- > d'_0 + d'_+$. Our proof would be complete if $[a,b]$ were an up-block for p, hence we may assume that

$$d_+ \leq d_0 + d_- \quad .$$

The optimality of p' implies that

$$d'_- \leq d'_0 + d'_+ \quad .$$

Now the latter inequality is equivalent to

$$d_- + d_0 + d_1 + \ldots + d_{t-1} \leq d_+ - (d_1 + \ldots + d_{t-1}) \quad ,$$

hence we have

$$d_+ \le d_0 + d_- \le d_+ - 2(d_1 + \ldots + d_{t-1}) \quad .$$

This can be true only if $d_+ = d_0 + d_-$ and $d'_- = d'_0 + d'_+$. If we shift block $[a,b]$ one position down from where it was in p', we obtain a new placement p'' of cost equal to p', hence p'' is optimum. Furthermore p'' is <u>closer</u> to the given placement, in an obvious sense, so the proof will eventually terminate.

It is interesting to note that the above proof does not use the hypothesis $h_1 \le \ldots \le h_N$; it characterizes the optimum placements for arbitrary (even non-integral) h_j. If $N = 2$, with $h_1 = 100$ and $h_2 = 1$, there are actually one hundred optimum placements, namely $p_k(j) = k+j$ for $-1 \le k \le 99$. The additional hypothesis $h_1 \le \ldots \le h_N$ leads to a slightly stronger theorem, showing that the optimum placements are more constrained: When $h_j \le h_{j+1}$, we have $h_j - p(j) \le h_{j+1} - p(j) = h_{j+1} - p(j+1) + 1$, for j and $j+1$ in the same block. The above proof can now be strengthened to show that $d_1 > 0$ (and hence $t \ge 1$) whenever p has no up-blocks. Thus, two optimum placements p and p' must have $|p(j) - p'(j)| \le 1$ for all j, whenever the h_j form a nondecreasing sequence.

References

Brent, Richard P. (1973). "Reducing the retrieval time of scatter storage techniques," Communications of the ACM, Vol. 16, No. 2, February 1973.

Knuth, Donald E. (1969). Seminumerical Algorithms; The Art of Computer Programming, Vol. 2, Addison-Wesley Publishing Company.

Knuth, Donald E. (1973). Sorting and Searching; The Art of Computer Programming, Vol. 3, Addison-Wesley Publishing Company.