The Analysis of an Improved Hashing Technique

Gaston H. Gonnet
J. Ian Munro

Department of Computer Science
University of Waterloo

Research Report CS-77-01

Abstract: We discuss the problem of hashing in a full or nearly full table using open addressing. A scheme for reordering the table as new elements are added is presented. Under the assumption of having a reasonable hash function sequence, it is shown that even with a full table only about 2.13 probes will be required, on the average, to access an element. This scheme has the advantage that the expected time for adding a new element is proportional to that required to determine that an element is not in the table. Attention is then turned to the optimal reordering scheme (which is a maximum flow problem) and the minimax problem of arranging the table so as to minimize the length of the longest probe sequence to find any element. A unified algorithm is presented for both of these as well as the first method suggested. A number of simulation results are reported, the most interesting being an indication that the optimal reordering scheme will lead to an average of about 1.83 probes per search in a full table.

1. - Introduction

Hash coding techniques are commonly used to quickly enter and retrieve information from tables. Indeed, they provide the possibility of retrieving data from an n entry table in a number of probes bounded (on the average) by a constant rather than log log n (all logarithms are to base 2 unless otherwise noted) (Gonnet (3), Yao and Yao (10)) for interpolation search, or log n for binary search. Recently, very sophisticated analyses of the behaviour of hashing techniques have been performed (Guibas(4), Guibas and Szemeredi(5), Knuth (6), Paterson (7)). The thrust of this work has, however, not been to provide new and better techniques, but as noted, a more sophisticated analysis of fairly standard methods. The state of the art of hashing remains essentially as follows:

If chaining (i.e., the additional storage of a pointer as part of each record) is permitted, then the search for an element which has been hashed to a full table can be conducted in an average of 1.5 probes.

The permanent retention of pointers in the table is very often unacceptable. We will be concerned with the situation in which no such auxiliary pointers are allowed, but extra storage may be used to determine the appropriate insertions to be made. For many applications this is precisely what is required.

(ii) The usual technique (when chaining is not allowed) of entering an element by rehashing until an empty location is found (simple open addressing) is quite acceptable until the table begins to fill. The average search time in a full table is, however, $ln(n)+O(1)$, and the expected worst case is $O(n)$ (i.e., it will probably take $O(n)$ probes to find some element, in particular n/2 for the last one inserted).

(iii) Brent (1) has suggested a method of reordering the table slightly as new elements are inserted. This leads to about 2.49 probes on the average for a retrieval from a full table, and an expected worst case of $O(n^{1/2})$.

(iv) An open addressing (i.e., rehashing) scheme will require at least an average of $(m+1)/(m+1-n)$ probes to ascertain that a particular element is not in a table of size m with n occupied locations.

The contribution of this paper is a new reordering scheme which is still practical and leads to an average of roughly 2.13 probes for retrieval from a full table and an apparent worst case of $O(\log n)$ probes. Furthermore we examine the problem of finding the arrangements to minimize the average retrieval time and to minimize the worst probe sequence. Simulation results are also presented.

In the analysis of our techniques (as well as the others) it is assumed that a pseudorandom number generator is used to hash a key into a permutation of the integers 0 through m-1 (where m denotes the table size). This permutation indicates the order in which table locations are to be searched for the specified key. In actual practice, a good method is that of choosing the table size, m, to be prime and making the primary hash location the key (binary number represented by the bit pattern of the key) module m. Subsequent locations are determined by repeatedly adding (modulo m) the key (module m-1)+1. As shown by Guibas and Szemeredi (5), this can cause some problems in the simple open addressing scheme when the table starts to fill, we argue (and Brent's and our experiments attest) that under his scheme, and ours, the search paths do not become long enough for this to be a noticeable factor.

2. - A Reordering Scheme

The essence of the algorithm is that when a key to be added to the table hashes to a location already occupied it is "essentially irrelevant" which of the two is located there and which is moved to its next "choice." Hence, if only one of them hashes next to a free location, it is placed there, while the other retains the original spot. Extending this idea another step, if both of these secondary locations are also occupied, there are (in general) 4 locations at the next "level" to check. Carrying the idea to its logical conclusion, we perform a breadth first search of the "binary tree" generated by the locations and subsequent rehashes of the keys encountered until an empty location is found. In the example below, the element to be inserted, *a*, hashes to a location currently occupied by *b* (*b* may or may not be in its "primary" location). The secondary location for *a* is occupied by *c*, and the next location for *b* by *d*. At the third level, however, we see that *b* hashes into an empty spot, and so *b* is moved there and *a* is placed in its primary location. The effect of adding *a* on the average retrieval time for all the elements in the system is equivalent to that of being able to insert *a* in its third location.

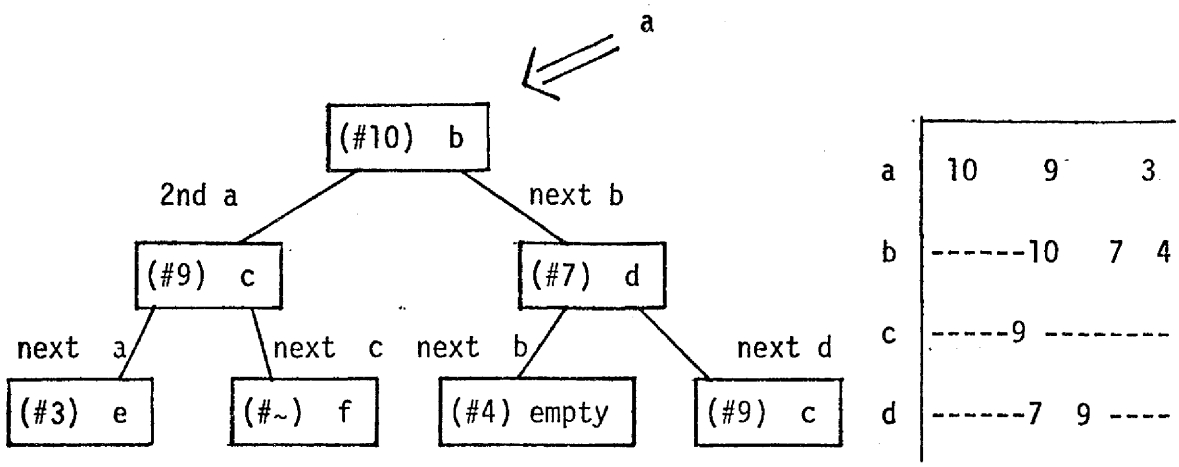


Figure I
The search conducted in adding a to the table and the relevant segment of the hashing function.

Note that rather than searching for an empty location by sampling without replacement (simply rehashing on a) at a "cost" of 1 more probe per sample whenever a search is performed for a, we are essentially sampling with replacement (note the probing of location 9 on two paths), but at an effective cost of the logarithm of the number of locations sampled when searches are performed. The fact that an element hashes into a permutation of the table locations (i.e., a path from any node in the tree of Figure 1 which always takes the left branch has no repetitions) does not significantly help us. We note that an elegant implementation of the search is achieved by representing the search tree as an array with the "2i, 2i+1 - heap style" technique of determining the left and right sons of a node. The binary representation of the heap position of the first empty location indicates the way in which the table is to be rearranged.

3. - Analyses of the Average Number of Probes Required

For purposes of the following preliminary analysis, we assume, that the sequence of probe positions is random and independent. Under this assumption, the number of probes, j, needed to find the first empty position in a table with m locations containing n elements has a geometric distribution with parameter α, that is $(1-\alpha)\alpha^{j-1}$, where $\alpha = n/m$ is the load factor.

This gives, during the search, an expected number of probes necessary of $1/(1-\alpha)$, and an overall average for the first n insertions of

$$\alpha^{-1}ln(1-\alpha).$$

The average depth in the search tree at which the first empty slot is found is

$$\sum_{j=1}^{\infty}(1-\alpha)\alpha^{j-1}(\lfloor\log_2 j\rfloor+1) = \sum_{k=0}^{\infty}\alpha^{2^k-1} = D(\alpha).$$

There is no known closed form for $D(\alpha)$, although the series, being doubly exponential, converges very rapidly for $\alpha<1$. Furthermore an asymptotic analysis shows that $D(1-\epsilon) = -(1-\epsilon)^{-1}\log\epsilon + O(\epsilon)$.

$D(\alpha)$, then, represents the expected length of the path of locate the new element, plus the increase in length of paths to previously located elements. From the point of view of determining the average path length, it is the effective contribution of adding the new element. We conclude, then, that the expected average path length when n elements have been inserted is

$$1/n\sum_{k=0}^{n-1}D(k/n) < 1/\alpha\int_0^{\alpha}D(p)dp =$$
$$\sum_{k=0}^{\infty}2^{-k}\alpha^{2^k-1} = \bar{D}(\alpha) \leqslant \bar{D}(1) = 2$$

Another quantity which may be of interest is the expected number of extra moves required during insertion. Let v(j) denote the number of 1's in the binary representation of j. Then, by examining a heap implementation of the scheme, we see that the average number of moves required when inserting a new element is

$$\sum_{j=1}^{\infty}(v(j)-1)(1-\alpha)\alpha^{j-1}$$
$$= \alpha^{-1}\sum_{k=0}^{\infty}\alpha^{3\times 2^k}/(1+\alpha^{2^k}) = M(\alpha)$$

Again, we know of no closed form for $M(\alpha)$, but it converges rapidly for $\alpha<1$. The expected number of moves of elements already in the table per insertion to fill a table up to a load factor of α (for large n) is

$$\bar{M}(\alpha) = 1/\alpha\int_0^{\alpha}M(p)dp = 1/\alpha\sum_{k=0}^{\infty}2^{-k}ln(1+\alpha^{2^k})-1 = \bar{M}(\alpha)$$
$$\leqslant \bar{M}(1) = ln(4)-1 = 0.386294...$$

This indicates, of course, that complicated sequences of moves happen very rarely.

The approximation of the distribution of the number of probes needed to make an insertion as geometric is rather good for a load factor of .8 or less. Indeed if it held for $\alpha=1$ we could expect to be able to access information from a full table in an average of 2 probes. Unfortunately this approximation leads to an error of a few percent as the table becomes very full. A flaw in the model is that it does not take into account the fact that short chains of probe positions tend to grow more quickly than "at random". Following an approach similar to Brent (1), we define $p_i(\alpha)$ to be the probability that given that a key, K, is in h_s, the s^{th} position of its hash sequence, and the next i probe positions, h_{s+1}, h_{s+2}, ..., h_{s+i}, are occupied.

If we insert δn keys into the file and ignore $O(\delta^2)$ contributions, we derive the following equation

$$(\alpha+\delta)\, p_i(\alpha+\delta) - \alpha p_i(\alpha) =$$

$$\delta\, \alpha^i \quad \text{(creation of a new chain)}$$

$$+ \delta/1-\alpha \sum_{j=0}^{i-1} \alpha^{i-j}(p_j(\alpha) - p_{j+1}(\alpha))$$

(extension of a chain by random placement)

$$+\sum_{j=0}^{i-1} \alpha^{i-j-1} Q_j.$$

Here Q_j denotes the sum of the probabilities of all binary trees for which a breadth first search for a free location ends in a chain of length j. (In terms of Figure 1 this means a right branch followed by j left branches). For example we have

$$Q_0 = \alpha^2(1-p_1(\alpha))\{1+\alpha\, p_1(\alpha) + \alpha\, p_1(\alpha)p_2(\alpha) + \ldots.\}$$

The total increment in the number of accesses is given by

$$D^*(\alpha) = 1+\alpha+\alpha^2 p_1(\alpha) + \alpha^3 p_1^2(\alpha)p_2(\alpha) + \ldots .$$

and the average number of accesses is then

$$\overline{D}^*(\alpha) = \int_0^\alpha D^*(t)dt$$

Taking the limit as $\delta \to 0$ in the above equations we derive an infinite system of differential equations. For small α we can find the solution in terms of a power series in α, obtaining

$$\bar{D}^*(\alpha) = 1+\alpha/2+\alpha^3/4+\alpha^4/15-\alpha^5/18+17\alpha^6/105+53\alpha^7/720-\ldots\ .$$

This series does not provide a reasonable method of determining $\bar{D}^*(\alpha)$ for α near to 1, but we can obtain reasonably good numerical approximations by numerically integrating the system of differential equations. Table 1 shows $\bar{D}^*(\alpha)$ and $\bar{M}^*(\alpha)$ obtained by numerical integration.

Average number of accesses and moves for
Binary Tree hashing

α	$\bar{D}^*(\alpha)$	$\bar{M}^*(\alpha)$
0.20	1.10209	0.01159
0.40	1.21746	0.04192
0.60	1.36362	0.09124
0.80	1.57886	0.17255
0.85	1.65554	0.20264
0.90	1.75084	0.24042
0.95	1.88038	0.29200
0.96	1.91376	0.30526
0.97	1.95143	0.32015
0.98	1.99525	0.33733
0.99	2.04938	0.35819
1.00	2.13414	0.38521

Table I

We observe that the numerically computed $\bar{M}^*(\alpha)$ is, in each case, slightly smaller than $\bar{M}(\alpha)$. This may appear inconsistent, but is explained by the fact that long chains do not require more moves.

A number of simulations were performed in order to test the accuracy of our analysis*.

* These, and all the other hashing experiments performed, used the double hashing scheme noted in Section 1 to generate the hash probe sequences. This was done in order to make extensive testing feasible. We claim that for all the insertion schemes that we use, there will be no noticeable difference between this scheme and that of random probe sequences. Appendix 1 contains a comparison of the two methods of probe sequence generation for fairly small tables.

Table II shows a typical experiment on a table of size 997 with various load factors (the ± terms indicate 95% central confidence limits). It is tedious, but not difficult, to rework our predictions of average behaviour for the non-asymptotic case and see that for all intents and purposes the limiting behaviour is achieved with tables of a few hundred elements. For this reason we are able to compare our experimental results with predicted asymptotic behaviour. Note that in all cases our theoretical average is well within the confidence interval of the experimental, and furthermore it is not consistently higher or lower. The average p.q.o. (priority queue operations in the implementation) column is a good measure of the cumulative time required for all the insertions. Another point of interest is that our preliminary analysis predicts an average of about 1.56 probes for a large table with $\alpha = .8$. We note this is not far off our improved and experimental results.

Simulation of "Binary tree" hashing

Size of table = 997 number of files (sample size) = 250

occup. factor	number of records	theor. average	average accesses	average max. acc.	average p.q.o.
80%	798	1.5789	1.58061±0.00302	6.184±0.114	2563.1±15.0
90%	897	1.7508	1.74778±0.00381	7.272±0.128	4206.3±31.6
95%	947	1.8804	1.87867±0.00433	8.316±0.152	6365.1±68.4
99%	987	2.0494	2.04991±0.00431	9.692±0.161	14250.±242.

Table II

Above this load, however, the difference becomes more significant reaching roughly .05 at 90% (the estimated average is roughly 1.70) and .13 at 100%, since our preliminary analysis predicts an average of 2.

Table III indicates the behaviour of our scheme on full tables.

Binary Tree hashing with a full table

file size	sample files	average accesses	average max. acc.	average p.q.o
19	1000	1.8903±0.0138	5.083±0.0914	106.11±2.26
41	1000	2.0053±0.0105	6.438±0.0984	331.25±6.41
101	400	2.0758±0.0107	7.855±0.156	1229.8±33.1
499	100	2.1358±0.0104	10.78±0.357	12612.±462.
997	50	2.13466±0.00958	11.02±0.443	31587.±1487.

Table III

Another interesting observation is the behaviour of the average of the maximum number of probes needed to access any element in a full table. From the analyses of the insertion scheme we see that as the table becomes full the depth of search required for an insertion will become $\sim$ log n on the average. Based on this we can expect the length of the longest probe sequence required to access an element to be $O(\log n)$ as well. Our experimental results in Table III suggest that may well be very close to $\log_2(n) + c$ (where c is roughly 1).

An efficient implementation of this algorithm, which was used to obtain the simulation results, is described with the optimal allocation algorithm.

4. - The Optimal Arrangement

It is not difficult to construct examples in which our ordering scheme does not provide the best possible arrangement of a set of keys, given their hash sequences. This is a result of the fact that a key tentatively assigned to the i^{th} location in its probe sequence can never be moved to an earlier one, regardless of the new keys added to the table. However, one might wonder how far from the cost of the optimal arrangement the one outlined above tends to be. Before making a comparison we briefly discuss the problem of determining to optimal arrangement.

The problem of optimal allocation is, as Rivest(9) has also observed, a special case of an assignment or minimum cost network flow problem (Edmonds & Karp(2)). In the terminology of network flows, we can construct a directed networks with nodes

(i) a source, s, and terminal node t

(ii) the keys K_i

(iii) the locations l_i and arcs with cost Δ at a particular time

(s,K_i); $\Delta(s,K_i) = 0$ $\forall$ K_i not assigned

(l_i,t) ; $\Delta(l_i,t) = 0$ $\forall$ l_i empty

(K_i,l_j); $\Delta(K_i,l_j) = p$
if K_i is not assigned to l_j and K_i probes to l_j in its p^{th} probe

$(l_j,K_i) = -p$ if K_i is assigned to l_j in its p^{th} probe.

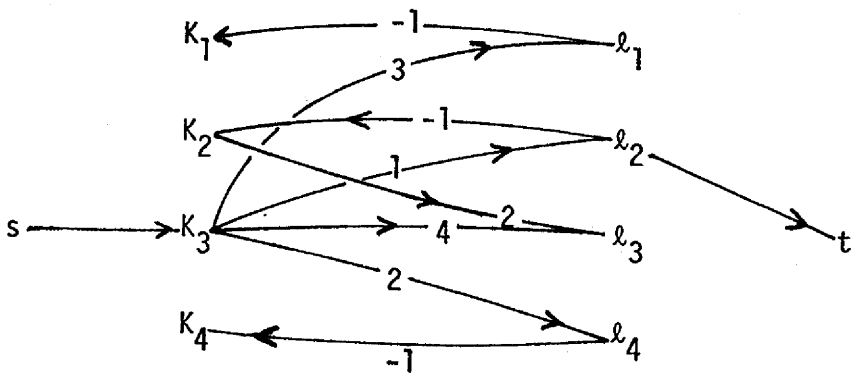
The assignment of a new key is translated to an augmentation of the flow from s to t. This is done by finding a minimum cost path from s to t.

In hashing terms this is equivalent to finding a minimum cost path (way of rearranging) from an unassigned key to an empty table location. For example consider the probe sequences for the keys K_1 to K_4 indicated below

probe positions

$K_1 \rightarrow 1, 4, 3, 2$
$K_2 \rightarrow 2, 3, 4, 1$
$K_3 \rightarrow 2, 4, 1, 3$
$K_4 \rightarrow 4, 2, 1, 3$

After we assign $K_1 \rightarrow 1$; $K_2 \rightarrow 2$ and $K_4 \rightarrow 4$ (that is an optimal partial assignment) the resulting network is



(some arcs are omitted for clarity)

Now if we are to insert K_3 we discover that a minimum cost path is s $\rightarrow$ K_3 $\rightarrow$ l_2 $\rightarrow K_2 \rightarrow l_3 \rightarrow$t. The cost of this path is 2 and the final assignment is

$K_1 \rightarrow l_1$

$K_2 \rightarrow l_3$

$K_3 \rightarrow l_2$

$K_4 \rightarrow l_4$

which is optimal.

In finding the minimum cost path from s to t, if (a) we keep in a priority queue the nodes visited with their respective partial cost, (b) we inspect the minimum-partial-cost location first, and (c) we start from an optimal allocation, then the first empty table location found will lead to a minimum cost path. (a path from l_j to l_i, where l_i is empty has cost $\geqslant 0$).

With these considerations, an algorithm that performs the optimal assignment can be coded in pseudo Algol 68 (with some redundancies) as follows

```
/*
n  is the number of keys to locate in the table,
m  is the number of table entries,
key(1:m+1)  contains the key number in location i; 0 if not occupied,
cost(1:m+1)  contains number of probes used to locate key in location,
sigma(1:m+1)  is used to find a minimum cost path,
path(1:m)  is used to record a minimum cost path.
                                        */
```

```
for i to m+1 do key(i) := 0;  cost(i) := 0;  sigma(i) := 0 od;
zero := -m-1;
for p to n do
        sigma(m+1) := zero;
        key(m+1) := source_key(p);
        clear heap;
        j := m+1;   ppos := 1;
        while true do
                heap ← {j,ppos+1,sigma(j)-zero-cost(j)+ppos+1};
                k := probe(key(j),ppos);
                if sigma(j)-cost(j)+ppos < sigma(k) then
                        sigma(k) := sigma(j)-cost(j)+ppos;
                        path(k) := j;
                        if key(k) = 0 then break while fi;
                        heap ← {k,1,sigma(k)-zero-cost(k)+1} fi;
                {j,ppos,} ← heap
                od;
        while k < m+1 do
                j := path(k);
                key(k) := key(j);
                cost(k) := cost(j)+sigma(k)-sigma(j);
                k := j;
                od;
        zero := zero-m-1;
        od;
```

Probe (Key,p) = l gives the p^{th} probe position of Key. The vector, cost, can be avoided if we are able to easily compute $probe^{-1}$(Key,l) = p. The vector, path, is only needed to perform a simple and efficient trace-back through the minimal path. Note that the function should not be linear probing, since for that scheme any ordering produces the same average number of accesses (Peterson (8)).

To implement a priority queue we use a heap which stores records of three components, the last element representing the ordering value for the priority queue. The use of the variable, zero, is to avoid the initialization of the partial cost vector, sigma, for each key. It is worth noting that if we change the statement

```
heap ←{K,1,sigma(K)-zero-cost(K)+1}
```

to

heap ←{K,cost(K)+1,sigma(K)–zero+1},

in the code above, we obtain an algorithm that only searches for an optimum by moving keys forward in their probe sequence. This is, except for the order in which a level of the binary tree is inspected, our previous algorithm.

The following tables summarize simulation results performed with the optimal algorithm

Simulation results for Optimal Hashing
Size of table = 997

occup. factor	number of records	average accesses	average max. acc.	average p.q.o.
80%	798	1.48902±0.00416	4.4±0.112	7456.±230.
90%	897	1.61039±0.00432	5.1467±0.0888	27973.±1431.
95%	947	1.68918±0.00586	5.68±0.118	79757.±4052.
99%	987	1.78514±0.00583	6.773±0.126	223262.±6931.

Table IV

Simulation of Optimal Hashing for full tables.

file size	sample files	average accesses	average max. acc.	average p.q.o
19	1000	1.72895±0.0107	4.385±0.0710	224.13±5.46
41	500	1.78283±0.0111	5.296±0.105	888.3±26.2
101	200	1.79837±0.0105	6.3±0.175	4611.±157.
499	50	1.82381±0.0110	7.92±0.358	89937.±4334.
997	50	1.82794±0.00639	8.98±0.382	332365.±12373.

Table V

5. - Minimax Arrangements

Another natural problem is that of arranging a set of keys in a table such that the length of the longest probe sequence to access any element is minimized. Among all possible minimax configurations we would, of course, like to find the o one which produces the minimum average number of accesses. The simulations reported in Section 1 indicate that our original scheme produces an average worst case of about log n in a full table. Gonnet (3) has demonstrated that for the minimax allocation the average length of the longest probe sequence is bounded below by $ln(n)+1.07...+o(1)$.

With a small variation in the optimal algorithm of the preceeding section we can derive a minimax allocation. The change is simply not to insert a record in the heap when its probe position exceeds the current minimax. Since, in the creation phase, we do not know the value of the minimax we try the procedure for minimax values of 1, 2, ... until it does not fail (i.e. the heap never empties before finding an empty table position). The bound noted above suggests that the run time will be multiplied by $ln(n)$. As a practical approach, we can improve this by finding the smallest value for the minimax such that at least n different table locations appear in the first minimax probes of the n keys.

The following algorithm constructs a minimax optimal hashing table based upon the above remarks.

```
/*
n  is the number of keys to locate in the table,
m  is the number of table entries,
key(1:m+1)  contains the key number in location i; 0 if not occupied,
cost(1:m+1)  contains number of probes used to locate key in location,
sigma(1:m+1)  is used to find a minimum cost path,
path(1:m)  is used to record a minimum cost path.
                                        */
```

```
uniq := 0;
for i to m do key(i) := 0 od;
for col to m while uniq<n do
        minmax := col;
        for p to n do
                k := probe(source_key(p),col);
                if key(k) = 0 then
                        key(k) := 1;   uniq := uniq+1 fi
                od
        od;
start:
for i to m+1 do key(i) := 0;  cost(i) := 0;  sigma(i) := 0 od;
zero := -m-1;
for p to n do
        sigma(m+1) := zero;
        key(m+1) := source_key(p);
        clear heap;
        j := m+1;   ppos := 1;
        while true do
                if ppos<minmax then
                        heap ← {j,ppos+1,sigma(j)-zero-cost(j)+ppos+1} fi;
                k := probe(key(j),ppos);
                if sigma(j)-cost(j)+ppos < sigma(k) then
                        sigma(k) := sigma(j)-cost(j)+ppos;
                        path(k) := j;
                        if key(k) = 0 then break while fi;
                        heap ← {k,1,sigma(k)-zero-cost(k)+1} fi;
                if empty_heap then minmax := minmax+1; go to start fi;
                {j,ppos,} ← heap
                od;
        while k < m+1 do
                j := path(k);
                key(k) := key(j);
                cost(k) := cost(j)+sigma(k)-sigma(j);
                k := j;
                od;
        zero := zero-m-1;
        od;
```

The tables below report our simulations of minimax hashing.

Simulation results for Minimax Optimal Hashing
Size of table = 499 Sample size = 100

occup. factor	number of records	average accesses	average max. acc.	average p.q.o.
80%	399	1.49378±0.00670	3±0	4464.±198.
90%	449	1.64829±0.00785	3.05±0.0429	22120.±1744.
95%	474	1.69945±0.00704	3.99±0.0196	41644.±2787.
99%	494	1.78824±0.00774	5.12±0.0893	77304.±4815.

Table VI

Simulation of Minimax Optimal Hashing for full tables.

file size	sample files	average accesses	average max. acc.	average p.q.o
19	1000	1.74858±0.0111	3.929±0.0622	241.04±7.35
41	600	1.79638±0.0102	4.665±0.0877	938.2±31.2
101	250	1.80737±0.0102	5.528±0.140	4851.±231.
499	100	1.82998±0.00807	7.38±0.287	91915.±3396.

Table VII

6. - Conclusion

We have examined the problem of arranging elements in a hash table to reduce both the average and expected value of the maximum number of probes required to access an element. The main results are summarized in Figure 2.

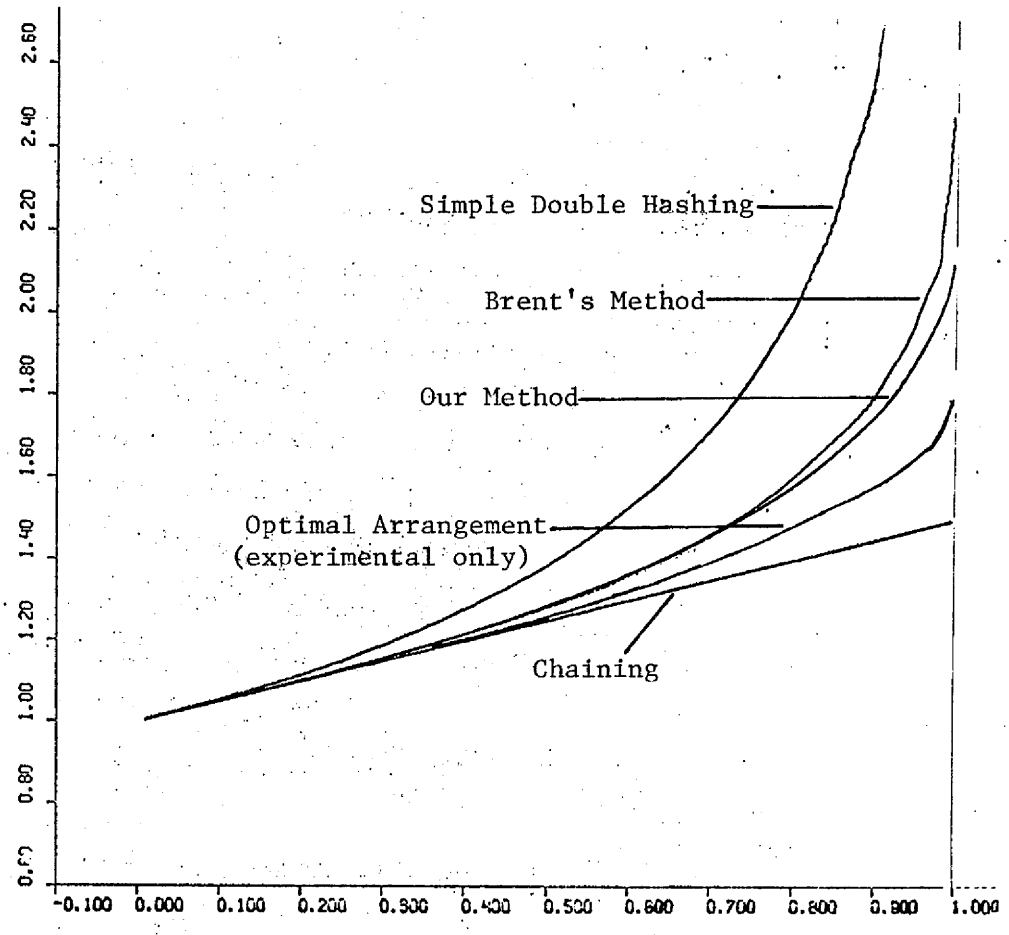


Figure II

The thrust of our work is definitely towards the concept that rather full hash tables using open addressing can still be extremely efficient structures and competitive with chaining techniques. The principal method discussed has the advantages of fast retrieval and insertion (on the average) even when the table is almost completely full. In terms of both the expected number of probes to access an element and the potential overhead in making an insertion, it lies halfway between Brent's limited search for an "insertion route" and the optimal assignment. In practice, it seems quite a reasonable scheme. If, however, the table is more than 80% full and to be referenced an extremely large number of times, it is probably worthwhile finding the optimal assignment

There are a number of interesting problems still open. Clearly the most interesting would be a proof that 1.83 or so probes are required, on the average for retrieval from a full but optimally arranged table.

Tight analyses of the expected maximum probe sequence for an access under our scheme or the optimal average strategy are also of interest.

References

1. Brent, R.P., "Reducing the Retrieval Time of Scatter Storage Techniques", CACM 16, 2(Feb. 1972), pp. 105–109.

2. Edmonds, J. and R.M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", JACM, Vol. 9, No. 2, April 1972, pp. 248–264.

3. Gonnet, G.H.,*Interpolation and Interpolation Hash Searching*, University of Waterloo, Computer Science Dept. Research Report 76–02.

4. Guibas, L.J., "The Analysis of Hashing Algorithms that Exhibit k–ary Clustering,"Proc. 17th Annual IEEE–FOCS Symp., Houston, Texas, Oct. 1976, pp. 183–196.

5. Guibas, L.J., and E. Szemeredi, "The Analysis of Double Hashing," Proc. 8th ACM Symp. on Theory of Computing, Hershey, Pennsylvania, May 1976, pp. 187–191.

6. Knuth, D.E., *The Art of Computer Programming*, Vol.III, Sorting and Searching, Addison–Wesley, Don Mills (1973)

7. Paterson, M.S., "On the Analysis of Hashing Schemes,"CMU Conference on Algorithm Analysis, April 1976.

8. Peterson, W.W., "Addressing for Random–Access Storage", IBM Journal of Research and Development, 1,4 (April 1957), pp. 130–146.

9. Rivest, R.L., "Optimal Arrangement of Keys in a Hash Table", to appear JACM.

10. Yao, A.C., and F.F. Yao, "The Complexity of Searching an Ordered Random Table", Proc. 17th Annual IEEE–FOCS Symp., Houston, Texas, Oct. 1976, pp. 173–177.

Acknowledgement

The authors thank Richard Lipton and Stanley Eisenstat for many fruitful discussions on the subject of optimal hash assignments.

Appendix I

From the work of Guibas (4) and Guibas & Szemeredi (5) it is known that double hashing can be noticeably worse than the use of "random" probe sequences in full tables if the naive insertion scheme is used. However, all techniques discussed in this paper tend to yield short probe sequences to access elements, even when the table is full. Hence, we claim, and note the mathematical details can be worked out to justify, that for our purposes there is no significant difference between random permutations and double hashing, except from the point of view of overhead. In actually using a hash table, the cost of generating (and retaining) random probe positions is prohibitive for large tables. Hence all experiments noted in the body of the paper were performed using double hashing. The table below shows the results of simulations performed with rather small tables using double hashing (d.h.) and random permutations (r.p.). Note that not only do the average number of probes and average maximum number of probes agree to within their confidence limits in all cases, but also that in some cases the average for double hashing just happened to be lower than for random permutations.

Comparison of "Binary tree" hashing using
double hashing vs. random permutations.

model	file size	sample files	average accesses	average max. acc.	average p.q.o
d.h.	19	1000	1.8903±0.0137	5.083±0.0914	106.11±2.26
r.p.	19	1000	1.9006±0.0142	5.06±0.0904	107.2±2.26
d.h.	41	1000	2.0053±0.0105	6.438±0.0984	331.25±6.41
r.p.	41	1000	1.9997±0.0101	6.406±0.0942	333.21±6.23
d.h.	101	400	2.0758±0.0107	7.855±0.156	1229.8±33.1
r.p.	101	400	2.0861±0.0108	7.978±0.179	1292.7±36.7

Table VIII