EFFICIENT ORDERING OF HASH TABLES*

GASTON H. GONNET and J. IAN MUNRO†

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 and the minimax problem of ordering the table so as to minimize the length of the longest probe sequence to find any element. Both arranging problems can be translated to assignment problems. A unified algorithm is presented for these, together with 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.

Key words. hashing; open addressing; maximum flow; table searching; assignment problem; analysis of algorithms; asymptotic analysis; simulation

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 [4], Yao and Yao [13]) for interpolation search, or log n for binary search. Recently, Guibas, Knuth and Szemeredi [7,8,9] performed very sophisticated analysis of the behavior of hashing techniques. 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:

(i) 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 $O(n^{1/2})$.

* Received by the editors September 6, 1977.
† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. This research was supported in part by NSERC under grant A8237 and by the University of Waterloo under operating grant 126-7029. A preliminary version of this paper was presented at the 9th Annual ACM Symposium on the Theory of Computing (May 1977).
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, apparently, an expected longest search 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. We find a close correspondence between organizing hashing tables and the assignment problem. For example, some results reported by Kurtzberg [10] correspond to open addressing hashing. The improvements and bounds obtained by Donath [2] correspond to Brent's reorganizing scheme. The Edmonds and Karp [3] algorithm for assignment problems corresponds to our optimal hashing algorithms.

Several simulation results are presented.

2. A reordering scheme. The essence of our 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 colliding keys 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 these locations and subsequent rehashes of the keys encountered until an empty location is found. In the example of Figure 1, 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.

![Diagram](image)

**FIG. 1.** 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 $\alpha$, that is $(1-\alpha)^{j-1}$, where $\alpha=n/m$ is the load factor.

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

$$E(\text{accesses}) = \alpha^{-1}[H_m-H_{m-n+1}] \sim -\alpha^{-1}\ln(1-\alpha),$$

where $H_n$ is the $n^{th}$ harmonic number.

Counting the root of the search tree to be a depth 1, the average depth 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 [5] shows that when $\alpha \to 1^-$ then

$$\alpha D(\alpha) = -\log_2(-\log_2\alpha) + \frac{1}{2} - \frac{\gamma}{\ln 2} + P(\log_2(-\log_2\alpha))$$

$$+ (1-\alpha) + \frac{1}{3} (1-\alpha)^2 + \frac{4}{21} (1-\alpha)^3 + O((1-\alpha)^4)$$

where $P(x)$ is periodic with period 1, and can be disregarded for practical purposes since

$$\left| P(x) \right| < 0.0000032.$$ 

This means that the last element inserted in a complete table increases the total path length by:

$$D \left( \frac{m-1}{m} \right) = \log_2 m + O(1).$$

$D(\alpha)$, then, represents the expected length of the path to 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
Another quantity which may be of interest is the expected number of moves required during insertion. Let \(v(j)\) denote the number of 1's in the binary representation of \(j\). Referring back to Figure 1, we see that the number of elements which are moved from their previous locations, while making an insertion, is precisely \(v(j)-1\), where the \(j^{th}\) location inspected is the first empty one found. An expression for the expected number of moves may be derived as

\[
E(\text{moves}) = \sum_{j=1}^{\infty} (v(j)-1) (1-\alpha) \alpha^{j-1}.
\]

Decomposing \(v(j)-1\) for each of its bit components we have

\[
E(\text{moves}) = \alpha^{-1}(1-\alpha)\left[ (\alpha^3+\alpha^5+\alpha^7+\alpha^9+\ldots) + (\alpha^6+\alpha^7+\alpha^{10}+\alpha^{11}+\alpha^{14}+\ldots) + (\alpha^{12}+\alpha^{13}+\alpha^{14}+\alpha^{15}+\alpha^{20}+\alpha^{21}+\ldots)\ldots \right]
\]

\[
= \alpha^{-1}\left[ (\alpha^3-\alpha^4+\alpha^5-\alpha^6+\ldots) + (\alpha^6-\alpha^8+\alpha^{10}-\alpha^{12}+\ldots) + (\alpha^{12}-\alpha^{16}+\alpha^{20}-\alpha^{24}+\ldots) + \ldots \right]
\]

\[
= \alpha^{-1}\sum_{k=0}^{\infty} \frac{\alpha^{3x2^k}}{(1+\alpha^{2^k})} = M(\alpha).
\]

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 \(\alpha\) is

\[
E(\text{moves}) = \overline{M}(\alpha) = \frac{1}{n_k=0} M(k/m) \]

\[
= \alpha^{-1}\left[ \overline{M}(p) \right. dp + O(m^{-1})
\]

\[
= \alpha^{-1}\sum_{k=0}^{\infty} 2^{-k}\ln(1+\alpha^{2^k}) - 1 + O(m^{-1})
\]

\[
\leq \overline{M}(1) = \ln(4)-1 = 0.386294\ldots
\]

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, that the next \(i\) probe positions, \(h_{s+1}, h_{s+2}, \ldots h_{s+i}\), are occupied. This
last sequence of occupied positions will be called the *chain of K*. An equivalent way of defining $p_i(\alpha)$ (or $p_i$ for short) is by

$$p_i(\alpha) = \frac{E(\text{number of chains of length } \geq i)}{n}.$$

We will now study the behavior of the quantity $\alpha p_i(\alpha)$ which represents the probability of finding a chain of length $i$ starting at any random location.

Inserting one key and studying the growth of chains, we derive the following system of difference equations:

$$(\alpha + 1/m) X_i \alpha(\alpha + 1/m) - \alpha p_i(\alpha) = \frac{\alpha^i}{m} \quad \text{(creation of a new chain)}$$

$$+ \frac{1}{m(1-\alpha)} \sum_{j=0}^{i-1} \alpha^{i-j}[p_j(\alpha) - p_{j+1}(\alpha)] \quad \text{(extension of a chain by the random placement of the new key)}$$

$$+ \frac{1}{m} \sum_{j=0}^{i-1} \alpha^{i-j} Q_j \quad \text{(extension of a chain caused by the binary tree insertion)}.$$

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 $j$. For example we have

$$Q_0 = \alpha^2(1-p_1)[1+\alpha p_1+\alpha^2 p_2+\alpha^3 p_3+\alpha^4 p_4+...];$$

$$Q_1 = \alpha^3(p_1-p_2)p_1[1+\alpha p_2+\alpha^2 p_3+\alpha^3 p_4+...].$$

$$\ldots \ldots$$

We will now explain each of the summands of the right hand side of the difference equations in terms of Figure 1.

The first summand comes from the creation of a new chain, in Figure 1, the chain in positions 10, 9, 3. Note that since $a$ is not yet in the table, the locations composing this chain are still independent.

The second summand appears from the extension of unrelated chains by the filling of an empty table position. In the example, location 4 will be filled, and consequently any previous chain (not necessarily related to the present construction) that was terminated by location 4, will now be extended. Observe that if a chain is extended (one location) by such an insertion, it may be extended several more positions by the random location of other keys. Note that the contributions of the first two summands are a consequence of any open-addressing scheme.

The last summand represents the extension of a chain originating in the binary tree search. In our example location 4 (and consequently the chain starting at $b$) has a higher than random probability of being filled by belonging to the full binary tree 10, 9, 7, 3, 15. The locations in the chain following the empty position are independent.

The crux of our use of the $p_i(\alpha)$ is that they carry information concerning the expected length of a chain which, intuitively, we expect to be larger than for uniform or random probing.
This model is not exact; there are several approximations. The most significant is that after insertion, some keys may be moved forward. This will reduce the length of a particular chain, (b in the example will now be located in position 4) while it increases the length of the new one (a will be guaranteed to have a chain of length 2). The total effect on the average length of a chain cancels out exactly, but it may alter the distribution of the \( p_i(\alpha) \) slightly.

Straightforward manipulation of the above expressions shows that the total increment in the number of accesses is given by

\[
D^*(\alpha) = 1 + \alpha + \alpha^2 p_1 + \alpha^3 p_2 + \alpha^4 p_3 + \alpha^5 p_4 + \alpha^6 p_5 + \alpha^7 p_6 + \cdots
\]

and the average number of accesses is then

\[
\overline{D}^*(\alpha) = \alpha^{-1} \int_0^\infty D^*(t) \, dt
\]

Taking the limit as \( m \to 0 \) in the above equations we derive an infinite system of differential equations. We can find the solution in terms of a power series in \( \alpha \), obtaining

\[
\overline{D}^*(\alpha) = 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + 17\alpha^6 / 105 + 53\alpha^7 / 720 - \cdots
\]

This series does not provide a reasonable method of determining \( \overline{D}^*(\alpha) \) as \( \alpha \) approaches to 1, but we can obtain reasonably good numerical approximations by numerically integrating the system of differential equations.

A similar analysis on the expected number of moves can also be performed. Using the \( p_i(\alpha) \), we define

\[
M^*(\alpha) = \alpha^2 (1 - p_1) \left( 1 + \alpha + 2\alpha p_2 + \alpha^2 p_2 + \cdots \right)
\]

and

\[
\overline{M}^*(\alpha) = \alpha^{-1} \int_0^\infty M^*(t) \, dt.
\]

Table 1 shows \( \overline{D}^*(\alpha) \) and \( \overline{M}^*(\alpha) \) obtained by numerical integration.

We observe that the numerically computed \( \overline{M}^*(\alpha) \) is, in each case, slightly smaller than \( \overline{M}(\alpha) \). 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 our other hashing experiments, use the double hashing scheme noted in the appendix 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. The appendix contains a comparison of the two methods of probe sequence generation for fairly small tables.

Table 2 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 behavior for the non-asymptotic case and see that for all intents and purposes the limiting behavior is achieved with tables of a few hundred elements. For this reason we are able to compare our experimental results with predicted asymptotic behavior. Note that in all cases our theoretical average is well within the confidence interval of the experimental, and furthermore it
is neither consistently higher nor 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. 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 3 indicates the behavior of our scheme on full tables.
Another interesting point is the behavior 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 3 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 the optimal arrangement.

The problem of optimal allocation is, as Rivest [12] has also observed, a special case of an assignment or minimum cost network flow problem. The solution to the assignment problem which we outline is essentially that of Edmonds and Karp [3]. In the terminology of network flows, we can construct a directed network with nodes

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

(ii) the keys $K_i$

(iii) the locations $l_i$

and arcs with cost $\Delta$ at a particular time

$$(s,K_i); \quad \Delta(s,K_i) = 0 \quad \text{for all } K_i \text{ not assigned}$$
$$(l_i,t); \quad \Delta(l_i,t) = 0 \quad \text{for all } l_i \text{ empty}$$
$$(K_i,l_j); \quad \Delta(K_i,l_j) = p \quad \text{if } K_i \text{ is not assigned to } l_j \text{ and } K_i \text{ probes to } l_j \text{ in its } p^{th} \text{ probe}$$
$$(l_j,K_i) = -p \quad \text{if } K_i \text{ is assigned to } l_j \text{ in its } p^{th} \text{ probe}.$$

### Table 3

**Binary Tree hashing with a full table**

<table>
<thead>
<tr>
<th>file size</th>
<th>sample files</th>
<th>average accesses</th>
<th>average max. acc.</th>
<th>average p.q.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>1000</td>
<td>$1.8903 \pm 0.0138$</td>
<td>$5.083 \pm 0.0914$</td>
<td>$106.11 \pm 2.26$</td>
</tr>
<tr>
<td>41</td>
<td>1000</td>
<td>$2.0053 \pm 0.0105$</td>
<td>$6.438 \pm 0.0984$</td>
<td>$331.25 \pm 6.41$</td>
</tr>
<tr>
<td>101</td>
<td>400</td>
<td>$2.0758 \pm 0.0107$</td>
<td>$7.855 \pm 0.156$</td>
<td>$1229.8 \pm 33.1$</td>
</tr>
<tr>
<td>499</td>
<td>100</td>
<td>$2.1358 \pm 0.0104$</td>
<td>$10.78 \pm 0.357$</td>
<td>$12612.4 \pm 462.$</td>
</tr>
<tr>
<td>997</td>
<td>50</td>
<td>$2.13466 \pm 0.00958$</td>
<td>$11.02 \pm 0.443$</td>
<td>$31587.4 \pm 1487.$</td>
</tr>
</tbody>
</table>
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:

```plaintext
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 given by Figure 2.

![Diagram](image)

**FIG. 2. The network resulting from an optimal partial assignment (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.

To implement the optimal arrangement we use Dijkstra's algorithm to find the minimum cost path from $s$ to $t$. Although the network contains negative arcs it can be demonstrated that in our case, this causes no problems. With these considerations, the algorithm can be coded with some redundancies in pseudo Algol 68 as follows
int n; # is the number of keys to locate in the table #
int m; # is the number of table entries #
[1:m+1] int
key, # contains the key number in location i; 0 if not occupied #
cost, # contains number of probes used to locate key in location #
sigma; # used to find a minimum cost path #
[1:m] int path; # 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;

Program Notes:

Probe (Key,p) = l gives the p\textsuperscript{th} probe position of Key. The vector, cost, can be avoided if we are able to compute probe\textsuperscript{-1}(Key,l) = p easily. The vector, path, is needed only to perform a simple and efficient trace-back through the minimal path. Note that the hashing function should not be linear probing, since for that scheme any ordering produces the same average number of accesses (Peterson [11]).

To implement a priority queue we use a heap which stores records of three components. Each record represents a node in the network. The first component identifies the associated key, the second, its next probe position, and the third, the path cost up to the node in question. The third element is 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

\[ \text{heap} \leftarrow \{k, \text{cost}(k) + 1, \sigma(k) - \text{zero} + 1\}, \]

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.

Tables 4 and 5 summarize simulation results performed with the optimal algorithm.

### Table 4

**Simulation results for Optimal Hashing**

*Size of table = 997  Number of sample files = 75*

<table>
<thead>
<tr>
<th>occup. factor</th>
<th>number of records</th>
<th>average accesses</th>
<th>average max. acc.</th>
<th>average p.q.o.</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>798</td>
<td>1.48902±0.00416</td>
<td>4.4±0.112</td>
<td>7456.7±230.</td>
</tr>
<tr>
<td>90%</td>
<td>897</td>
<td>1.61039±0.00432</td>
<td>5.1467±0.0888</td>
<td>27973.6±1431.</td>
</tr>
<tr>
<td>95%</td>
<td>947</td>
<td>1.68918±0.00586</td>
<td>5.68±0.118</td>
<td>79757.4±4052.</td>
</tr>
<tr>
<td>99%</td>
<td>987</td>
<td>1.78514±0.00583</td>
<td>6.77±0.126</td>
<td>223262.6±6931.</td>
</tr>
</tbody>
</table>

### Table 5

**Simulation of Optimal Hashing for full tables.**

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

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 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 [6] has demonstrated that for the minimax allocation the average length of the longest probe sequence is bounded below by \( \ln(n) + O(1) \).

With a small variation in the optimal algorithm of the preceding 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, \ldots 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.

\begin{verbatim}
int n; # is the number of keys to locate in the table #
int m; # is the number of table entries #
[1:m+1] int key, # contains the key number in location i; 0 if not occupied #
cost, # contains number of probes used to locate key in location #
sigma; # used to find a minimum cost path #
[1:m] int path; # 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
  minimax := 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<minimax 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 minimax := minimax+1; goto start fi;
    [j,ppos,] := heap
  od;
\end{verbatim}
while \( k < m+1 \) do

\[
\begin{align*}
  j &:= \text{path}[k]; \\
  \text{key}[k] &:= \text{key}[j]; \\
  \text{cost}[k] &:= \text{cost}[j] + \sigma[k] - \sigma[j]; \\
  k &:= j \\
  \text{od}; \\
\end{align*}
\]

\[
\text{zero} := \text{zero} - m - 1
\]
\[
\text{od};
\]

Tables 6 and 7 report our simulations of minimax hashing.

**Table 6**

*Simulation results for Minimax Optimal Hashing*

<table>
<thead>
<tr>
<th>Occup. factor</th>
<th>Number of records</th>
<th>Average accesses</th>
<th>Average max. acc.</th>
<th>Average p.q.o.</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>399</td>
<td>1.49378±0.00670</td>
<td>3±0</td>
<td>4464.4±198.</td>
</tr>
<tr>
<td>90%</td>
<td>449</td>
<td>1.64829±0.00785</td>
<td>3.05±0.0429</td>
<td>22120.4±1744.</td>
</tr>
<tr>
<td>95%</td>
<td>474</td>
<td>1.69945±0.00704</td>
<td>3.99±0.0196</td>
<td>41644.4±2787.</td>
</tr>
<tr>
<td>99%</td>
<td>494</td>
<td>1.78824±0.00774</td>
<td>5.12±0.0893</td>
<td>77304.4±4815.</td>
</tr>
</tbody>
</table>

**Table 7**

*Simulation of Minimax Optimal Hashing for full tables.*

<table>
<thead>
<tr>
<th>File size</th>
<th>Sample files</th>
<th>Average accesses</th>
<th>Average max. acc.</th>
<th>Average p.q.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>1000</td>
<td>1.74858±0.0111</td>
<td>3.929±0.0622</td>
<td>241.04±7.35</td>
</tr>
<tr>
<td>41</td>
<td>600</td>
<td>1.79638±0.0102</td>
<td>4.665±0.0877</td>
<td>938.2±31.2</td>
</tr>
<tr>
<td>101</td>
<td>250</td>
<td>1.80737±0.0102</td>
<td>5.528±0.140</td>
<td>4851.4±231.</td>
</tr>
<tr>
<td>499</td>
<td>100</td>
<td>1.82998±0.00807</td>
<td>7.38±0.287</td>
<td>91915.4±3396.</td>
</tr>
</tbody>
</table>

6. **Conclusion.** We have examined the problem of arranging elements in a hash table to reduce the average and also the maximum number of probes required to access an element. The main results are summarized in Figure 3.

The thrust of our work is toward the thesis 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.

7. Appendix. Guibas and Szemeredi [8] show that double hashing is equivalent to uniform probing up to a certain load factor. However, all techniques discussed in this paper tend to yield short probe sequences to access elements, even when the table is full. Therefore we claim 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 re-
taining) random probe positions is prohibitive for large tables. Hence all experi-
ments noted in the body of the paper were performed using a double hashing
scheme, suggested by Brent [1], which is very useful in practice. The table size, \( m \), is
chosen to be prime, the table running from position 0 up to \( m - 1 \). The primary
hash location of a key is obtained by taking (the binary number represented by the
bit pattern of) the key modulo \( m \), subsequent locations are determined repeatedly by
adding (modulo \( m \)) the key modulo \( (m - 2) + 1 \). The table below shows the results of
simulations performed with rather small tables using double hashing (d.h.) and ran-
dom 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.

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

Acknowledgment. The authors thank Richard Lipton and Stanley Eisenstat for
many fruitful discussions on the subject of optimal hash assignments and the referee
for his/her very precise comments on an earlier manuscript.

REFERENCES

105-109.
[5] ———, Notes on the derivation of asymptotic expressions from summations, Information Processing
Theoretical Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August
1977, pp. 159-162.


