Practical Perfect Hashing

G.V. Cormack[1]*, R.N.S. Horspool*[2]*, M. Kaiserswerth*[3]

School of Computer Science
McGill University

ABSTRACT

A practical method is presented that permits retrieval from a table in constant time. The method is suitable for large tables and consumes, in practice, $O(n)$ space for n table elements. In addition, the table and the hashing function can be constructed in $O(n)$ expected time. Variations of the method that offer different compromises between storage usage and update time are presented.

1. Introduction

This paper presents a practical method for building perfect hash tables. A perfect hash table is a table from which any element may be retrieved in constant time. In addition to allowing constant retrieval time, the tables described here can be updated in constant expected time, and can therefore be built incrementally in time proportional to the number of elements, n. The tables carry some extra information, but their overall size is also linear in n. The method presented is suitable for tables of arbitrary size.

Hashing is a popular technique because the expected retrieval time is effectively a constant. Unfortunately, hashing is often avoided in real-time applications because the worst-case retrieval time is proportional to n, implying that some retrieval operations may be unacceptably slow. For these applications, perfect hashing schemes are desirable. Such schemes have, to date, been suitable only for small, constant tables. This is because they require either tremendous computational effort to construct the table, or require much more storage than that required to actually hold the elements in the table.

Recently, Fredman *et al.* [1] have shown that perfect hash tables of linear size exist, but do not give a practical algorithm for constructing them. In addition, they state that the tables may consume many times more storage than is necessary to store the elements. Updating the table is not discussed. The research described here was motivated by this paper.

The following sections present the data structures that represent the hash table, the retrieval algorithm, and the algorithms for insertion and deletion. The speed and storage consumption of the method are discussed. Variations that improve the speed of insertion at the expense of some storage are presented. Finally, empirical results are presented that show the actual construction time for the table to vary little from the expected time.

---

[1] Present address: Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1.
[2] Present address: Department of Computer Science, University of Victoria, Box 1700, Victoria, B.C. V8W 2Y2.
[3] Present address: Universität Erlangen, IMMD IV, Martensstrasse 3, 8520 Erlangen, Federal Republic of Germany.

2. Random Hash Functions

In order to construct the hash tables, we require a family of hash functions, $\{h_i(k,r)\}$, each of which accepts a key value, k, and a range size, r, and returns a value within the range 0 to $r-1$. Each h_i should provide a different, random mapping from the set of possible keys to the set of range values. Knuth [3] discusses various criteria for the selection of appropriate hash functions, and concludes that remainder on division works well, subject to some constraints on the divisor. For the experimental work described here, the following function was found to work well:

$$h_i(k,r)=mod(mod(k,2i+100r+1),r).$$

In the actual implementation, the value of $2i+100r+1$ was manipulated directly, rather than the value i which appears in the following algorithms. It is implicitly assumed here that $k \gg 2i+100r+1$. If this is not the case, the range of key values should be shifted by adding a large number to k.

3. The Hash Table

The hash table consists of two parts: a dense or nearly dense table, D, containing the actual data elements, and a header table, H, which contains information for the calculation of the address of a particular data item in the dense table. The header contains $O(n)$ elements, but does not contain any keys or data. The retrieval algorithm consists of extracting one element from H, based on the value of the key, and then using that information to calculate the exact address in D of the appropriate data item, if it exists. Existence of the data item in the table must be verified by checking the key found at this address.

D contains the data items along with their associated keys, and nothing else. If the data elements are of variable size, they must be stored with one extra level of indirection, thus resulting in one additional storage reference per retrieval.

Each header element contains three fields: an index p into D, and two small integers, i and r. The header size s is chosen such that $n=\rho s$, where ρ is a constant with a value near one; ρ represents a loading factor for H. Choosing an appropriate value for ρ will be discussed in section 6.

4. The Retrieval Algorithm

In order to find the record with key k, we perform the following steps:

Compute $x=h(k,s)$
Extract $<p,i,r>$ from $H[x]$
If $r = 0$ then there is no record with $key=k$ in D
Otherwise:
  Compute $y=p+h_i(k,r)$
  Extract $<key,data>$ from $D[y]$
  If $key = k$ then the record is found
  Otherwise there is no record with $key=k$ in D

Figure 1. Retrieval algorithm.

Some specific details of the algorithm for the actual implementation follow. For the first step, we chose s (the header size) to be a prime number, and used $h(k,s)=mod(k,s)$ to ensure a reasonable distribution. In the fifth step, we used the definition for h_i in section 2. Note that any retrieval requires at most two probes: one to the header and, probably, one to the dense table. Each probe involves retrieving a fixed amount of contiguous storage and perform-

ing a simple calculation. Therefore, the entire retrieval takes constant time. If the tables are on secondary storage, as is likely the case for very large tables, the retrieval requires no more than two physical input operations. This number is likely to be more significant than the amount of computation performed.

5. Building the Table

The hash table can be built incrementally. Each header element is initialized with $p=i=r=0$. A record to be inserted is denoted by $<k,d>$, where k is the key value and d is the associated data. The algorithm for inserting this new record is as follows:

```
Compute x=h(k,s)
Let <p,i,r> be H[x]
If r=0 then
  Let y = the index of any free slot in D
  Store <y,0,1> back into H[x]
  Store <k,d> into D[y]
Otherwise:
  Let <k_j,d_j> be D[p+j]  0≤j<r
  Search to find an m such that the r+1 values
    h_m(k_j,r+1)  0≤j<r,
    and  h_m(k,r+1)
  are all distinct
  Let y= the index of the first of r+1 consecutive free slots in D
  Store <y,m,r+1> in H[x]
  Store <k,d> into D[y+h_m(k,r+1)]
  For 0≤j<r
    Move <k_j,d_j> to D[y+h_m(k_j,r+1)]
  Mark the r slots D[p] .. D[p+r−1] as free
```

Figure 2. Insertion algorithm.

The first hash calculation, $h(k,s)$, determines a group of of records in D. All records that hash to the same value in the calculation of $h(k,s)$ are stored as a contiguous group in D. For each group, the H table entry records the starting index for the group (p), the size of the group (r), and a parameter (i) to select the second hash function. A group may be placed at an arbitrary position in D.

The second hash calculation, $h_i(k,r)$, is required to yield the offset within the group of the desired record. As new records are inserted, the groups grow in size. Each time the size of a group increases, we must find a new hash function, h_m, and we must find new storage in D for the group. Details of this storage management are omitted from the algorithm. In principle, groups could be shifted down to make space where it is needed. However, movement of large numbers of records in secondary storage is liable to be expensive. A better approach is to permit a little space in secondary storage to be wasted and use a storage allocation and de-allocation strategy, such as *first fit* or *best fit* [2].

6. How long does it take?

The insertion algorithm may, potentially, be quite slow. The statement beginning *Search* can test an arbitrary number of hash functions before finding a suitable h_m. Each attempt requires up to $r+1$ trial computations of the hash function before being accepted or rejected because of a collision. There is no upper bound on the amount of time required to do a particular insertion. We can, however, compute the expected time to perform an insertion.

The first key in a group of size $r+1$ will hash to an unoccupied location with probability 1; the second key with probability $r/(r+1)$; the third key with probability $(r-1)/(r+1)$, and so on. Taking the product of these probabilities for all $r+1$ keys, we see that the probability of a particular h_m being appropriate (perfect) is given by:

$$\frac{(r+1)!}{(r+1)^{r+1}}.$$

The average number of h_m that need to be tested is simply the reciprocal of this expression. Clearly, this search can take a long time for a large value of r. It remains to be seen that such large r occur sufficiently infrequently for our hashing scheme to be feasible.

As insertions into the table is made, the sets of keys that hash to the same group of slots in D will tend to grow in size. The expected cost of each insertion will grow as the table fills up. Let us consider just the cost of the last insertion, when the n^{th} record is inserted into a table that already holds $n-1$ records. Now, assuming that all s^{n-1} combinations of $h(k,s)$ values were equally likely for the $n-1$ keys in the table, the probability P_r that a particular group holds exactly r records is given [3, exercise 6.4(34)] by:

$$P_r = \frac{\binom{n-1}{r}(s-1)^{n-r-1}}{s^{n-1}}$$

If the n^{th} key should hash into this group of size r, the expected cost of performing the insertion is, at worst, given by:

$$\frac{(r+1)^{r+1}}{r!}.$$

We say "at worst" because we have assumed that each trial hash function requires $r+1$ computations before it is accepted or rejected. Our cost measure is simply a count of how many trial computations of hash functions are performed. The expected cost of an insertion, C, is therefore limited by:

$$C < \sum_{r=0}^{n-1} \frac{(r+1)^{r+1}}{r!} \frac{\binom{n-1}{r}(s-1)^{n-r-1}}{s^{n-1}}$$

Rearranging and using the fact that $\frac{s-1}{s} < 1$, we have:

$$C < \sum_{r=0}^{n-1} \frac{(r+1)^{r+1}\,(n-1)!}{(r!)^2\,(n-1-r)!\,s^r}$$

Further weakening the bound, and substituting ρ for n/s, we arrive at:

$$C < \sum_{r=0}^{n-1} \frac{(r+1)^{r+1}}{(r!)^2}\rho^r$$

Since we are interested only in a bound on C, we can extend the upper bound on the summation to give:

$$C < \sum_{r=0}^{\infty} \frac{(r+1)^{r+1}}{(r!)^2} \rho^r$$

The ratio between the $k-1^{th}$ and k^{th} terms in this summation is $\rho(k+1)^{k+1}/k^{k+2}$. Since this ratio tends to zero as $k \to \infty$, the summation must converge. In other words, the expected cost of an insertion into the table is constant.

We note, however, that decreasing the ratio n/s would decrease the expected cost. This ratio, ρ, can be used to make a tradeoff between insertion time and storage usage. It should be noted, however, that only the size of H is directly affected by ρ, and that these header elements may well be much smaller than the data elements in D. Thus, the storage overhead associated with using a small value of ρ may, in fact, be negligible. The retrieval time, of course, is unaffected by the choice of ρ.

Finally, we note that the Poisson approximation yields a value of C that is less than our bound by a factor of e^{ρ}. This approximation has been found to be very accurate in predicting the distribution of group sizes and insertion costs.

7. Using memory to decrease insertion time

While the average insertion time is not large, a particular insertion could take a long time, especially if a large group is involved. It is possible to use storage in D to reduce expected insertion time. Previously, we assumed that a group of size r must occupy r consecutive slots in D. If we use more than r slots for this group of r records, there is more chance that we can find a suitable hash function. In fact, by using sparsely populated regions in D for the groups, the average number of h_m tested can be made independent of the group size. The result is that the overall insertion time is decreased, while the expected size of the table is increased.

If we insert r elements into a table of size t using a random hash function, the probability, $\pi(t,r)$, of the hash being collision-free is

$$\pi(t,r) = \prod_{i=1}^{r-1} 1 - \frac{i}{t}.$$

This formula is similar to that derived in the previous section, where $r=t$ held, and is derived in the same way. In order to minimize the search time, we can choose t to be some function of r, $f(r)$ say, where $f(r) \geq r$. The expected amount of space consumed for a group is

$$\sum_{r=0}^{n} \frac{\binom{n}{r} (s-1)^{n-r}}{s^n} f(r)$$

This sum is bounded by some constant, independent of n, for any polynomial $f(r)$. Since there are s groups, the total amount of storage used for groups in D must, therefore, be proportional to s and, hence, proportional to n. Thus, we can choose a polynomial $f(r)$ in order to improve insertion time while still maintaining storage consumption proportional to n.

In order to find an appropriate $f(r)$, let us re-express the probability of a particular hash function being collision-free in terms of its logarithm:

$$log(\pi(t,r)) = \sum_{i=1}^{r-1} log(1 - \frac{i}{t})$$

Using the Taylor series for $log(1-x)$, we have

$$log(\pi(t,r)) = \sum_{i=1}^{r-1} \sum_{j=1}^{\infty} -\frac{(i/t)^j}{j}$$

$$= -\sum_{j=1}^{\infty} \frac{1}{jt^j} \sum_{i=1}^{r-1} i^j .$$

If $t=f(r)$, and we choose $f(r)$ such that $\lim_{r\to\infty} f(r)<r^2$, then $\lim_{r\to\infty} log(\pi(r))=-\infty$, and hence $\lim_{r\to\infty} \pi(r)=0$. (Note: we omit the t parameter from π when t has been bound to a particular function of r.) If $\lim_{r\to\infty} f(r)>r^2$, then $\lim_{r\to\infty} log(\pi(r))=0$, and $\lim_{r\to\infty} \pi(r)=1$. Finally, when $\lim_{r\to\infty} f(r)=r^2$, $\lim_{r\to\infty} log(\pi(r))$ converges to a constant. In particular, if $f(r)=r^2$, $\lim_{r\to\infty} log(\pi(r))=-0.5$, and $\lim_{r\to\infty} \pi(r)=0.606$.

Thus, if we pick $f(r)=r^2$, we have at least a .606 probability of a particular random hash function being appropriate, regardless of group size. If we assume $\rho=1$, the amount of storage consumed in D is doubled by choosing $f(r)=r^2$, as compared with $f(r)=r$.

With this choice of $f(r)=r^2$, the number of attempts to find a suitable hash function for a group is made largely independent of the size of the group. Thus, we can optimize storage consumption by varying ρ.

Other functions, even discontinuous functions, can be used for $f(r)$. One such function is:

$$f(r) = \begin{cases} r & \text{if } r \le c \\ r^2 & \text{if } r > c \end{cases}$$

If $c=2$, we achieve a large reduction in table size with little increase in insertion time. For larger values of c, the trade-off is less pronounced. More time is spent and the storage saving is smaller. Experimental results for various choices of ρ and the c parameter in this $f(r)$ function are presented in section 9.

8. Storage management

During the construction of the table, contiguous groups of slots in D of various sizes are allocated and freed, potentially wasting storage and time due to external fragmentation [2]. However, the number of discrete sizes is small, and the method of keeping a list of available groups of each size works well. Since the number of groups with r members varies inversely with $r!$, groups of any particular size are consumed faster than they are created, as long as $\rho \le 1$. Thus we expect to find very few groups in any availability list. The experimental results presented in the next section include storage wasted due to our use of this memory-management strategy.

When elements are deleted from the table, the availability lists will grow. This freed storage cannot be reclaimed (for another purpose) without compaction. However, the storage is available should the table grow again.

9. Experimental results

The hashing method was implemented in the C language on a VAX 11/750. A dictionary of approximately 25,000 English words was used as the set of keys. The program was run for all combinations of ρ = 0.5, 1, and 2, and c = 1, 2, 3, 4, 5, and ∞, where c is the parameter of the $f(r)$ function suggested in section 7. All combinations performed reasonably well, and we must conclude that the appropriate choice depends on the intended application. The time to construct the table was measured by counting the number of calculations of h_i for each insertion. The amount of storage consumed by D was measured by the number of storage slots allocated. Other samples of various sizes were also used and, in all cases, the time and storage consumption varied from the figures presented in this paper by less than two per cent.

For finite values of c, the following optimization was performed: Whenever an insertion was made into a sparse group (*i.e.* a group that contained unused slots), the existing h_i was evaluated for the new key, and if no collision resulted, the key was inserted into the existing group without expanding its size to $f(r+1)$. This optimization caused the construction time and space to be noticeably less than was predicted in section 7.

Figure 3 shows the average time per key required to construct the table. This cost is presented as a function of c. The three curves represent ρ = 0.5, 1 and 2, from bottom to top. Note that the larger values of c are more costly, but for small values of ρ, the difference is not large.

Figure 3. Time to construct hash table.

Figure 4 shows the number of storage slots (per key) required in D for each strategy. Again, the values of ρ are ascending from bottom to top. There is a clear tradeoff between time and space, as seen by comparing the slopes of the lines in figures 3 and 4. If an optimal choice of c is desired, the relative values of time and storage would have to be evaluated, and the two costs added using suitable weights.

Figure 4. Storage consumed by hash table.

The storage needed for H has not been included in this figure as there is no direct relationship between the lengths of the elements of H and those of D. In many cases, this storage will be insignificant. In other cases, such as when the header is to be stored using a different medium, its length may be of paramount importance.

Another consideration in the selection of the appropriate values is the variance in insertion times. If the table is being constructed incrementally in real time, the fact that the average insertion time is short will be less significant than the fact that there may be occasional, anomalous, transactions that take hundreds of times longer.

Figures 5, 6, and 7 give the distribution of insertion times for $\rho = 0.5$, 1, and 2, respectively. Each figure shows the minimum amount of time required for the most expensive n per cent of the insertions as a function of n. The five curves represent $c = 1$, 2, 3, 4, 5, and ∞, from bottom to top. The best combination is $\rho = 0.5$, $c = 1$ where only one per cent of insertions require more than 7 evaluations of h_i. The worst is $\rho = 2$, $c = \infty$, where one per cent require 250 evaluations.

Figure 5. Distribution of insertion times for ρ=0.5.

Figure 6. Distribution of insertion times for ρ=1.

Figure 7. Distribution of insertion times for ρ=2.

10. Conclusions

The method presented here is practical. The implementation is hardly any more complex than other hashing methods, such as hashing with external chaining. The retrieval cost of two storage probes is competitive with the average cost of other methods. In this case, however, it is the worst case as well. Since the retrieval algorithm has no loops, it is particularly suitable for implementation in hardware.

The appropriate choices for the parameter ρ and the function $f(r)$ depend on the particular application. It appears to be desirable to make ρ as small as possible, whatever is chosen for $f(r)$. When choosing c values for the $f(r)$ function suggested in section 7, small values reduce construction time but increase storage consumption. In addition, large values of c give rise to a small fraction of very expensive insertions. These anomalies are of little consequence when constructing a static table, as was done for these experiments. They could be significant in a real-time application, where insertions and deletions are performed dynamically. We note that it is possible to build the table using one choice for $f(r)$ and to use another when doing further updates.

Finally, the table can be constructed in linear time even if the number of elements is not known in advance. One can pick an arbitrary header size and begin building the table. When ρ exceeds the desired value, the table is rebuilt with double the header size. The total time will not exceed double that to construct the final table.

11. Acknowledgement

Funding for this research was received from the Natural Sciences and Engineering Research Council of Canada under grants A4333 and A5485.

12. References

[1] L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with O(1) worst case access time, *Proc.* 23rd *Symposium on Foundations of Computer Science*, IEEE Computer Society (1982), 165-168.

[2] D. Knuth, *The art of computer programming*, Vol. I, second edition, Addison-Wesley, Reading (1973).

[3] D. Knuth, *The art of computer programming*, Vol. III, Addison-Wesley, Reading (1973).