MINIMAL PERFECT (ORDERED) HASH FUNCTIONS

hash function is to take $h(x) = x \bmod m$ for some value $m > n/\alpha$, where α is the *loading*, the ratio of records to available addresses, and m is usually chosen to be a prime number. Thus, if asked to provide a hash function for 1,000 integer keys, a programmer might suggest something like $h(x) = x \bmod 1{,}399$ to give a load factor of $\alpha = 0.7$ in a table declared to have 1,399 locations.

The smaller the value of α, the less likely it is that two of the keys *collide* at the same hash value. Nevertheless, collisions are almost impossible to avoid. This fact is somewhat surprising when first encountered and is demonstrated by the well-known *birthday paradox*, which asks, "Given that there are 365 days in the year, how many people must be collected together before the probability that two people share a birthday exceeds 0.5?" In other words, given 365 hash slots, how many keys can be randomly assigned before the probability of collision exceeds 0.5? The initial reaction is usually to say that lots of people are needed. In fact, the answer is just 23, and the chance that a hash function of realistic size is collision-free is insignificant.[1] For example, with 1,000 keys and 1,399 randomly selected slots, the probability of there being no collisions at all is 2.35×10^{-217}. (The derivation of this probability can be found in the next subsection: it is given by Equation 4.1 with $m = 1{,}399$ and $n = 1{,}000$.) The inevitability of collisions has led to a large body of literature on how best to handle them. Here, however, we seek instead those one-in-a-million hash functions that do manage to avoid all collisions.

If the hash function has the additional property that, for x_i and x_j in L, $h(x_i) = h(x_j)$ if and only if $i = j$, it is a *perfect* hash function. In this case, no collisions arise when hashing the set of keys L.

If a hash function h is both perfect and maps into the range $m = n$, each of the n keys hashes to a unique integer between 1 and n and the table loading is $\alpha = 1.0$. Then h is a *minimal perfect hash function*, or MPHF. An MPHF provides guaranteed one-probe access to a set of keys, and the table contains no unused slots.

Finally, if a hash function has the property that if $x_i < x_j$ then $h(x_i) < h(x_j)$, it is *order preserving*. Given an order-preserving minimal perfect hash function (abbreviated OPMPHF and pronounced "oomph!"), keys are located in constant time without any space overhead and can be processed in sorted order should that be necessary. An OPMPHF simply returns the sequence number of a key directly.

Of course, an MPHF or OPMPHF h for one set L will not be perfect for another set of keys, and so it is nothing more than a precalculated lookup function for a single set. Nevertheless, there are occasions when the precalculation is warranted, and the space saving can be great.

1 It is always interesting to try this experiment with groups of people. Having tried this experiment many times with students while teaching them about hash tables, there is one important tip that we would like to pass on: the participants should be asked to write down their birthday (or any other date) *before* the collation process is commenced, so that the temptation for mysterious negative feedback is eliminated. In our experience, unless this is done, it can sometimes take 366 students before the first collision. Perhaps there is a psychology paradox here too.

Table 4.3 Tables for a minimal perfect hash function: (a) terms and hash functions; (b) function g.

(a)

Term t	$h_1(t)$	$h_2(t)$	$h(t)$
jezebel	5	9	0
jezer	5	7	1
jezerit	10	12	2
jeziah	6	10	3
jeziel	13	7	4
jezliah	13	11	5
jezoar	4	2	6
jezrahiah	0	3	7
jezreel	6	3	8
jezreelites	8	4	9
jibsam	9	14	10
jidlaph	3	1	11

(b)

x	$g(x)$
0	0
1	4
2	0
3	7
4	6
5	0
6	1
7	1
8	3
9	0
10	2
11	2
12	0
13	3
14	10

As an example, Table 4.3 gives an OPMPHF for the same set of 12 keys that was used earlier. The methodology leading to this hash function is described in the next section. The construction presumes the existence of two normal hash functions $h_1(t)$ and $h_2(t)$ that map strings into integers in the range $0 \ldots m-1$ for some value $m \geq n$, with duplicates permitted. One way to define these is to take the numeric value for each character of a string radix 36, as before, and compute a weighted sum for some set of weights w_j,

$$h_j(t) = \left(\sum_{i=1}^{|t|} t[i] \times w_j[i] \right) \bmod m,$$

where $t[i]$ is the radix-36 value of the ith character of term t and $|t|$ is the length in characters of term t. Then two different sets of weights $w_1[i]$ and $w_2[i]$ for $1 \leq i \leq |t|$ yield two different functions $h_1(t)$ and $h_2(t)$. As well as these two functions, a rather special array g is needed that maps numbers $0 \ldots m-1$ into the range $0 \ldots n-1$; this is shown in Table 4.3b.

To evaluate the OPMPHF $h(t)$ for some string t, calculate

$$h(t) = g(h_1(t)) +_n g(h_2(t)),$$

where $+_n$ means addition modulo n—that is, add the two numbers together, and take the remainder on division of the sum by n. (For example, $4 +_9 7 = 2$.) In other words, evaluate the two nonperfect hash functions, convert the resulting values using the mapping g, and then add them modulo n. The result of this calculation for the example lexicon is shown in the fourth column of Table 4.3a. As if by magic, the final hash values are exactly the ordinal positions within the list of strings.

To make this work, the array g must meet some very special constraints. A detailed description of how it can be obtained appears shortly. Let us simply accept that for any given set of strings it is possible to construct functions h_1, h_2, and g so that $h(t) = g(h_1(t)) +_n g(h_2(t))$ is the ordinal number of string t.

Suppose that an OPMPHF h is calculated for the set of index terms in a given lexicon. There is no need to store the strings or the string pointers—all that is required is for f_t and the inverted file address for term t to be stored in the $h(t)$th positions of their respective arrays.

There is a catch, which is the space required for the description of the hash function h. It has been shown that at least $1.44n$ bits of storage are required by *any* MPHF (Fox, Heath, et al. 1992), and more typically, easily calculable MPHFs for large values of n require from 4 to 20 bits per key. The specification of OPMPHFs is even more lengthy and requires at least $n \log n$ bits of storage (Fox et al. 1991). In the OPMPHF described, the two functions h_1 and h_2 are determined by small tables of weights w_1 and w_2, so they require negligible space. On the other hand, array g is m items long and occupies $m \log n$ bits even when stored as compactly as possible. The method detailed in the next section operates with $m \approx 1.25n$, and this is why, in the example of Table 4.3, the $n = 12$ strings were handled using $m = 15$ entries in the array g. This means that array g occupies at least 25 bits per string, or, in practical terms with each entry stored as a four-byte integer, $1.25 \times 4 \times 1{,}000{,}000 = 5$ Mbytes for the hypothetical lexicon of $n = 1{,}000{,}000$ words. Another 8 Mbytes is still required for the disk pointers and term frequencies. In total, if an OPMPHF of the type described here is used, this lexicon can be reduced to 13 Mbytes, compared with 15.5 Mbytes for a 3-in-4 front-coded representation.

Design of a minimal perfect hash function

To set the scene for the development of an algorithm for finding minimal perfect hash functions, let us first calculate the probability for the birthday paradox. Suppose that n items are to be hashed into m slots. The first item can be placed anywhere without risk of collision. The second item will avoid collision with probability $(m-1)/m$ since one slot is now occupied; the third, with probability $(m-2)/m$; and so on. The probability of inserting n consecutive items without collision is the product of these probabilities:

$$\prod_{i=1}^{n} \frac{m-i+1}{m} = \frac{m!}{(m-n)!\,m^n}. \tag{4.1}$$

When $m = 365$ and $n = 22$, the probability is 0.524, and when $n = 23$, the probability decreases to 0.493, so if there are 23 people in a room, it is more likely than not that at least two of them will have the same birthday.

Now let us turn to the construction of the array g that is the secret of the MPHF shown in the example in Table 4.3. Recall that

$$h(t) = g(h_1(t)) +_n g(h_2(t))$$

and that

$$h_1(t) = \left(\sum_{i=1}^{|t|} t[i] \times w_1[i]\right) \bmod m,$$

$$h_2(t) = \left(\sum_{i=1}^{|t|} t[i] \times w_2[i]\right) \bmod m,$$

where $t[i]$ is the ith character of the string being hashed. The first step in developing an OPMPHF is to choose mappings randomly for the functions h_1 and h_2. There are several ways to do this, the easiest of which is to generate random integers into the two arrays w_1 and w_2 used in their definition. Once this is done, the search for a function g can be commenced.

One way to visualize the situation is as an m-vertex graph, with vertices labeled $0 \ldots m-1$ and edges defined by $(h_1(t), h_2(t))$ for each of the terms t. Each term in the lexicon corresponds to one edge of the graph, and the values of the two hash functions define to which vertices that edge is incident. Finally, suppose that each edge is labeled with a value $h(t)$, where $h(t)$ is the desired value of the hash function for term t. The graph corresponding to the functions h_1 and h_2 in Table 4.3a is shown in Figure 4.4. It has $m = 15$ vertices and $n = 12$ edges. Graph algorithms are normally described with n as the number of vertices and m the number of edges, but consistency has required the opposite convention for this discussion.

What is needed now is a mapping g from vertices to integers $0 \ldots n-1$ such that, for each edge $(h_1(t), h_2(t))$, the mapping yields $g(h_1(t)) +_n g(h_2(t)) = h(t)$, the label on the edge.

For a general graph, finding such a labeling, if it exists, is difficult. But suppose that the graph is known to be *acyclic*; that is, it has no closed cycles of edges. For example, the graph of Figure 4.4 is acyclic, but if there were an edge from vertex 2 to vertex 8, then a cycle 2–4–8–2 would be formed, and it would no longer meet this requirement.

The desired function g for an acyclic graph is easily derived. Any unprocessed vertex v is chosen and assigned $g(v) = 0$. The edges out of that vertex are then traced, and the destinations of those edges are labeled with the h value of the edge used. In the next step, a second generation of vertices is labeled, this time with the *difference* between the tag on the edge used and the label of the vertex that is the source of the edge. If there are unlabeled vertices, another root is chosen and the process repeats. Work continues until all vertices are labeled, at which point the mapping g is complete.

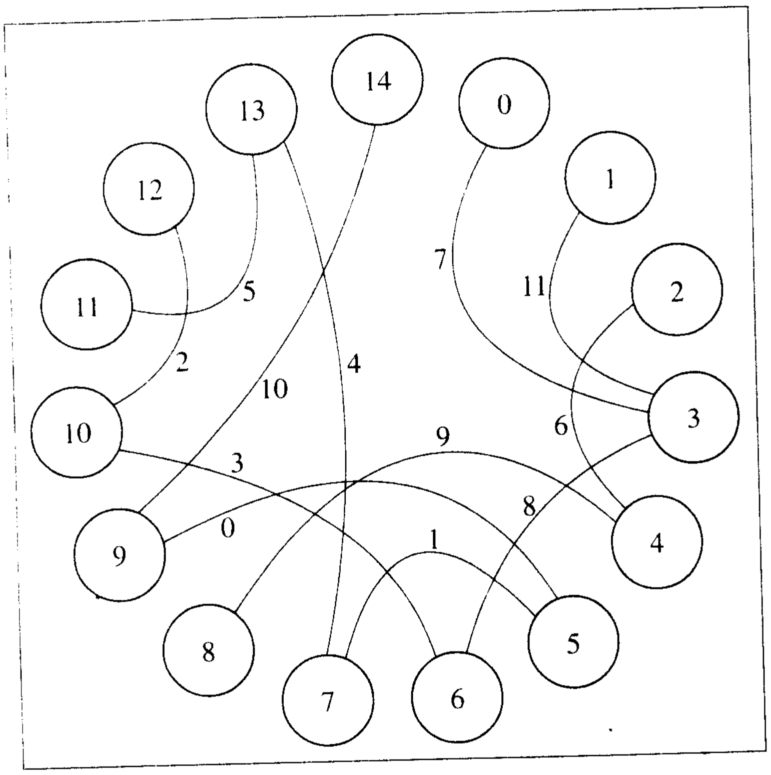

Figure 4.4 Graph corresponding to hash function of Table 4.3.

For example, suppose that vertex 0 in Figure 4.4 is chosen as the root of one of these connected components, and the assignment $g[0] = 0$ is made. Then $g[3]$ can be set to 7, which in turn means that $g[6]$ can be set to 1 and $g[1]$ can be set to 4. But if $g[6] = 1$, then $g[10]$ must be 2, and if $g[10] = 2$, then $g[12]$ must be 0. This is the end of the component rooted at vertex 0, and the next unlabeled vertex—in this case vertex 2—is selected as the root of a new component, giving $g[2] = 0$, $g[4] = 6$, and $g[8] = 3$. Finally, vertex 5 is taken as a root, and the remaining vertices are assigned values for g during the processing of that component.

If the graph were not acyclic, this labeling process might trace around a cycle and insist on relabeling some already-processed vertex with a different label than the one that has already been assigned to it. On an acyclic graph this cannot happen, and labeling is always possible. Because of this, the test for acyclicity can be built into the labeling process. Figure 4.5 describes this process for an arbitrary undirected graph $G = (V, E)$. It is assumed that *adjacent*(v) is a list of vertices that share an edge with vertex v and that $h((v, u))$ is the label associated with the edge joining v and u.

This mechanism requires a linear number of steps to either fully assign the mapping g or to report that the graph is not acyclic, and the function *LabelFrom* will be called at most $2m$ times. Figure 4.6 describes an iterative process that generates and

```
To label an acyclic graph,
  1. For $v \in V$,
         Set $g[v] \leftarrow unknown$.
  2. For $v \in V$,
         If $g[v] = unknown$ then
             $LabelFrom(v, 0)$.

where the function $LabelFrom(v, c)$ is defined by
  1. If $g[v] \neq unknown$ then
         If $g[v] \neq c$ then
             return with failure—the graph is not acyclic,
         else
             return—this vertex has already been visited.
  2. Set $g[v] \leftarrow c$.
  3. For $u \in adjacent(v)$,
         $LabelFrom(u, h((v, u)) - g[v])$.
```

Figure 4.5 Checking for acyclicity and assigning a mapping.

```
To generate a perfect hash function,
  1. Choose a value for $m$.
  2. Randomly choose weights $w_1[i]$ and $w_2[i]$ for $1 \leq i \leq \max_{t \in L} |t|$, where $L$
     is the set of strings to be hashed and $|t|$ is the length of string $t$.
  3. Generate the graph $G = (V, E)$, where
         $V = \{1, \ldots m\}$ and
         $E = \{(h_1(t), h_2(t)) \mid t \in L\}$.
  4. Use the algorithm of Figure 4.5 to attempt to calculate the mapping $g$.
  5. If the labeling algorithm returns with failure, go back to step 2.
  6. Return the arrays $w_1$, $w_2$, and $g$.
```

Figure 4.6 Generating a perfect hash function.

tests hash functions. The essence of this algorithm is simple: new functions h_1 and h_2 are generated until an acyclic graph results.

The single remaining question is crucial to the usefulness of this technique: how likely is it that the graph G produced at steps 2 and 3 is acyclic? The answer to this question depends upon the value of m that is chosen in the first step. Clearly, the larger the value of m, the sparser the graph G and the more likely it is to be acyclic.

Analysis based upon the theory of random graphs shows that for $m \leq 2n$, the probability of generating an acyclic graph tends toward zero as n grows—the edges are too dense, and it becomes inevitable that a cycle is formed somewhere in the graph (Czech, Havas, and Majewski 1992). On the other hand, when $m > 2n$, the probability of a random graph of m vertices and n edges being acyclic is approximately

$$\sqrt{\frac{m-2n}{m}}.$$

Hence the expected number of graphs generated until the first acyclic graph is found is

$$\sqrt{\frac{m}{m-2n}}.$$

For example, if $m = 3n$, then on average $\sqrt{3} \approx 1.7$ graphs will be tested before one suitable for use in the hash function is generated. In total, the time taken to generate the hash function is proportional to the number of items in the set L, provided that $m > 2n$.

The drawback of using $m = 3n$ is the space required by the array g. If this array takes three four-byte integers per term, then it is no cheaper than storing the terms themselves. On the other hand, reducing m/n below 2 means that many graphs must be generated before an acyclic one is found. This is tolerable only if the set of keys is small and the time taken to generate the mapping is of no concern.

There is, however, another way to reduce the ratio m/n. The example given in Table 4.3 and Figure 4.4 assumes the use of a 2-graph, where each edge connects two vertices. Suppose a third random hash function h_3 is introduced, and a 3-graph is formed on m vertices, where each edge is a triple of the form $(h_1(t), h_2(t), h_3(t))$, and the hash function is given by

$$h(t) = g(h_1(t)) +_n g(h_2(t)) +_n g(h_3(t)).$$

The requirement for the existence of a mapping function g becomes somewhat more complex, but the logic is the same: a graph can be used if there is some sequence of edge deletions such that each edge deleted has at least one vertex of degree one, and the sequence removes all the edges. Another way to state this requirement is to ask that no subgraph contain only vertices of degree two or greater.

Experiment and analysis have shown that with 3-graphs, the critical ratio of m to n is about 1.23 (Havas et al. 1993; Majewski et al. 1996). That is, if $m > 1.23n$, then a graph with the desired property will be constructed on average after a constant number of trials. All the other steps continue to take time proportional to the size of the graph and in practice are extremely fast. For example, it takes less than 1 minute of processor time to build a minimal perfect hash function for the more than 500,000 terms in the *TREC* lexicon.

Finally, we should admit to a small inconsistency. To avoid confusion, the example used in Table 4.3 and Figure 4.4 shows $n = 12$ and $m = 15$ in the ratio 1.25 but

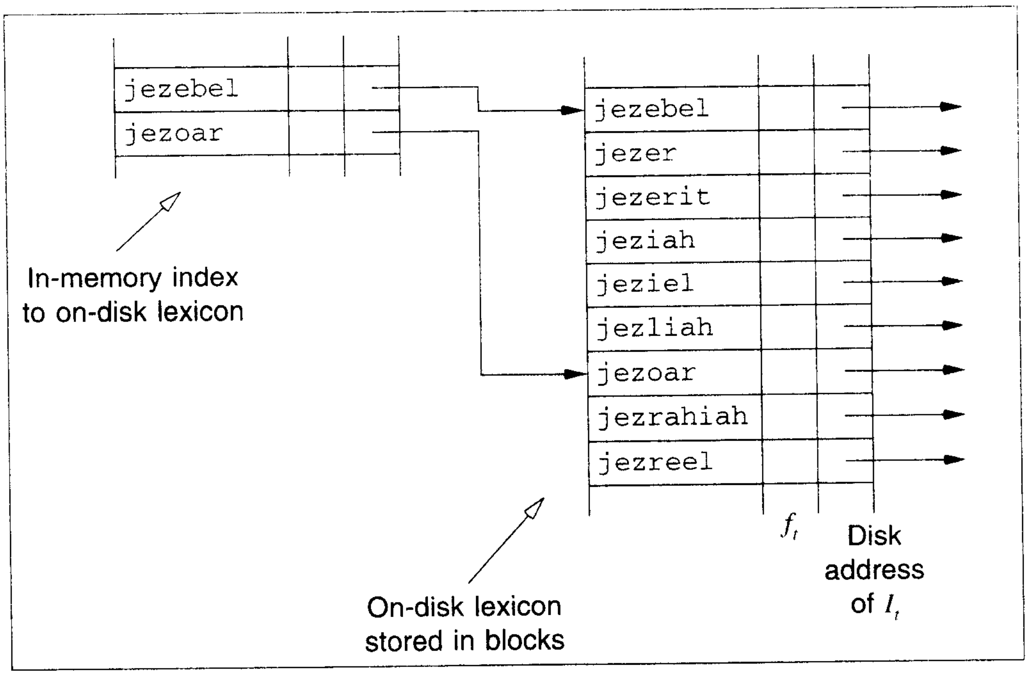

Figure 4.7 Disk-based lexicon storage.

employs two hash functions rather than three. This worked because the example was carefully chosen.

Disk-based lexicon storage

There is a much simpler way to reduce the amount of primary memory required by the lexicon: put it on disk, with just enough information retained in primary memory to identify the disk block corresponding to each term t. Using 4 Kbyte disk blocks, the 20 Mbyte example lexicon of strings, inverted file pointers, and f_t values together occupy about 5,000 disk blocks. Provided that the 5,000 first-in-block terms are held in memory in a searchable array, as little as 100 Kbytes are sufficient to store this memory-resident index table. Figure 4.7 shows how the example strings from *Bible* might be stored.

To locate the information corresponding to a given term, the in-memory index is searched to determine a block number; that block is read into a buffer, and the search is continued within the block. More generally, a B-tree or other dynamic index structure can be used, with leaf pages on disk and a single root page in memory.

This approach is attractive because it is simple and requires a minimal amount of primary memory. However, a disk-based lexicon is many times slower to access than a memory-based one. One disk access per lookup is required—perhaps 10 milliseconds even on a fast machine, compared with the few microseconds required by an in-memory binary search. This extra time is tolerable when just a few terms are