9

Hash Tables

Pat Morin
Carleton University

9.1 Introduction

A *set abstract data type (set ADT)* is an abstract data type that maintains a set S under the following three operations:

1. INSERT(x): Add the key x to the set.
2. DELETE(x): Remove the key x from the set.
3. SEARCH(x): Determine if x is contained in the set, and if so, return a pointer to x.

One of the most practical and widely used methods of implementing the set ADT is with *hash tables*.

Note that the three set ADT operations can easily be implemented to run in $O(\log n)$ time per operation using balanced binary search trees (See Chapter 10). If we assume that the input data are integers in the set $U = \{0, \ldots, u-1\}$ then they can even be implemented to run in sub-logarithmic time using data structures for integer searching (Chapter 39). However, these data structures actually do more than the three basic operations we require. In particular if we search for an element x that is not present in S then these data structures can report the smallest item in S that is larger than x (the *successor* of x) and/or the largest item in S that is smaller than x (the *predecessor* of x).

Hash tables do away with this extra functionality of finding predecessors and successors and only perform exact searches. If we search for an element x in a hash table and x is not present then the only information we obtain is that $x \notin S$. By dropping this extra functionality hash tables can give better performance bounds. Indeed, any reasonable hash table implementation performs each of the three set ADT operations in $O(1)$ expected time.

The main idea behind all hash table implementations discussed in this chapter is to store a set of $n = |S|$ elements in an array (the hash table) A of length $m \geq n$. In doing this, we require a function that maps any element x to an array location. This function is called a *hash function* h and the value $h(x)$ is called the *hash value* of x. That is, the element x gets stored at the array location $A[h(x)]$. The *occupancy* of a hash table is the ratio $\alpha = n/m$ of stored elements to the length of A.

The study of hash tables follows two very different lines. Many implementations of hash tables are based on the *integer universe assumption*: All elements stored in the hash table come from the universe $U = \{0, \ldots, u-1\}$. In this case, the goal is to design a hash function $h : U \to \{0, \ldots, m-1\}$ so that for each $i \in \{0, \ldots, m-1\}$, the number of elements $x \in S$ such that $h(x) = i$ is as small as possible. Ideally, the hash function h would be such that each element of S is mapped to a unique value in $\{0, \ldots, m-1\}$. Most of the hash functions designed under the integer universe assumption are number-theoretic constructions. Several of these are described in Section 9.2.

Historically, the integer universe assumption seems to have been justified by the fact that any data item in a computer is represented as a sequence of bits that can be interpreted as a binary number. However, many complicated data items require a large (or variable) number of bits to represent and this make u the size of the universe very large. In many applications u is much larger than the largest integer that can fit into a single word of computer memory. In this case, the computations performed in number-theoretic hash functions become inefficient.

This motivates the second major line of research into hash tables. This research work is based on the *random probing assumption*random probing assumption: Each element x that is inserted into a hash table is a black box that comes with an infinite random *probe sequence* $x_0, x_1, x_2, \ldots$ where each of the x_i is independently and uniformly distributed in $\{0, \ldots, m-1\}$. Hash table implementations based on the random probing assumption are described in Section 9.3.

Both the integer universe assumption and the random probing assumption have their place in practice. When there is an easily computing mapping of data elements onto machine word sized integers then hash tables for integer universes are the method of choice. When such a mapping is not so easy to compute (variable length strings are an example) it might be better to use the bits of the input items to build a good pseudorandom sequence and use this sequence as the probe sequence for some random probing data structure.

To guarantee good performance, many hash table implementations require that the occupancy α be a constant strictly less than 1. Since the number of elements in a hash table changes over time, this requires that the array A be resized periodically. This is easily done, without increasing the amortized cost of hash table operations by choosing three constants $0 < \alpha_1 < \alpha_2 < \alpha_3 < 1$ so that, whenever n/m is not the interval (α_1, α_3) the array A is resized so that its size is n/α_2. A simple amortization argument (Chapter 1) shows that the amortized cost of this resizing is $O(1)$ per update (Insert/Delete) operation.

9.2 Hash Tables for Integer Keys

In this section we consider hash tables under the integer universe assumption, in which the key values x come from the universe $U = \{0, \ldots, u-1\}$. A *hash function* h is a function whose domain is U and whose range is the set $\{0, \ldots, m-1\}$, $m \leq u$. A hash function h is said to be a *perfect hash function* for a set $S \subseteq U$ if, for every $x \in S$, $h(x)$ is unique. A perfect hash function h for S is *minimal* if $m = |S|$, i.e., h is a bijection between S and $\{0, \ldots, m-1\}$. Obviously a minimal perfect hash function for S is desirable since it

allows us to store all the elements of S in a single array of length n. Unfortunately, perfect hash functions are rare, even for m much larger than n. If each element of S is mapped independently and uniformly to a random element of $\{0, \ldots, m-1\}$ then the birthday paradox (See, for example, Feller [27]) states that, if m is much less than n^2 then there will almost surely exist two elements of S that have the same hash value.

We begin our discussion with two commonly used hashing schemes that are heuristic in nature. That is, we can not make any non-trivial statements about the performance of these schemes when storing an arbitrary set S. We then discuss several schemes that have provably good performance.

9.2.1 Hashing by Division

In *hashing by division*, we use the hash function

$$h(x) = x \bmod m \ .$$

To use this hash function in a data structure, we maintain an array $A[0], \ldots, A[m-1]$ where each element of this array is a pointer to the head of a linked list (Chapter 2). The linked list L_i pointed to by the array element $A[i]$ contains all the elements x such that $h(x) = i$. This technique of maintaining an array of lists is called *hashing with chaining*.

In such a hash table, inserting an element x takes $O(1)$ time; we compute $i = h(x)$ and append (or prepend) x to the list L_i. However, searching for and/or deleting an element x is not so easy. We have to compute $i = h(x)$ and then traverse the list L_i until we either find x or reach the end of the list. The cost of this is proportional to the length of L_i. Obviously, if our set S consists of the elements $0, m, 2m, 3m, \ldots, nm$ then all elements are stored in the list L_0 and searches and deletions take linear time.

However, one hopes that such pathological cases do not occur in practice. For example, if the elements of S are uniformly and independently distributed in U and u is a multiple of m then the expected size of any list L_i is only n/m. In this case, searches and deletions take $O(1+\alpha)$ expected time. To help avoid pathological cases, the choice of m is important. In particular, m a power of 2 is usually avoided since, in a binary computer, taking the remainder modulo a power of 2 means simply discarding some high-order bits. Taking m to be a prime not too close to a power of 2 is recommended [37].

9.2.2 Hashing by Multiplication

The implementation of a hash table using *hashing by multiplication* is exactly the same as that of hashing by division except that the hash function

$$h(x) = \lfloor mxA \rfloor \bmod m$$

is used. Here A is a real-valued constant whose choice we discuss below. The advantage of the multiplication method is that the value of m is not critical. We can take m to be a power of 2, which makes it convenient for use on binary computers.

Although any value of A gives a hash function, some values of A are better than others. (Setting $A = 0$ is clearly not a good idea.)

Knuth [37] suggests using the *golden ratio* for A, i.e., setting

$$A = (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

This choice of A is motivated by a theorem, first conjectured by Oderfeld and later proven by Sẃierczkowski [59]. This theorem states that the sequence

$$mA \bmod m,\ 2mA \bmod m,\ 3mA \bmod m,\ \ldots,\ nmA \bmod m$$

partitions the interval $(0, m)$ into $n + 1$ intervals having only three distinct lengths. Furthermore, the next element $(n + 1)mA \bmod m$ in the sequence is always contained in one of the largest intervals.[1]

Of course, no matter what value of A we select, the pigeonhole principle implies that for $u \geq nm$ then there will always exist some hash value i and some $S \subseteq U$ of size n such that $h(x) = i$ for all $x \in S$. In other words, we can always find a set S all of whose elements get stored in the same list L_i. Thus, the worst case of hashing by multiplication is as bad as hashing by division.

9.2.3 Universal Hashing

The argument used at the end of the previous section applies equally well to any hash function h. That is, if the table size m is much smaller than the universe size u then for any hash function there is some large (of size at least $\lceil u/m \rceil$) subset of U that has the same hash value. To get around this difficulty we need a collection of hash functions from which we can choose one that works well for S. Even better would be a collection of hash functions such that, for any given S, most of the hash functions work well for S. Then we could simply pick one of the functions at random and have a good chance of it working well.

Let $\mathcal{H}$ be a collection of hash functions, i.e., functions from U onto $\{0, \ldots, m-1\}$. We say that $\mathcal{H}$ is *universal* if, for each $x, y \in U$ the number of $h \in \mathcal{H}$ such that $h(x) = h(y)$ is at most $|\mathcal{H}|/m$. Consider any $S \subseteq U$ of size n and suppose we choose a random hash function h from a universal collection of hash functions. Consider some value $x \in U$. The probability that any key $y \in S$ has the same hash value as x is only $1/m$. Therefore, the expected number of keys in S, not equal to x, that have the same hash value as x is only

$$n_{h(x)} = \begin{cases} (n-1)/m & \text{if } x \in S \\ n/m & \text{if } x \notin S \end{cases}$$

Therefore, if we store S in a hash table using the hash function h then the expected time to search for, or delete, x is $O(1 + \alpha)$.

From the preceding discussion, it seems that a universal collection of hash functions from which we could quickly select one at random would be very handy indeed. With such a collection at our disposal we get an implementation of the set ADT that has $O(1)$ insertion time and $O(1)$ expected search and deletion time.

Carter and Wegman [8] describe three different collections of universal hash functions. If the universe size u is a prime number[2] then

$$\mathcal{H} = \{h_{k_1,k_2,m}(x) = ((k_1 x + k_2) \bmod u)) \bmod m : 1 \leq k_1 < u, 0 \leq k_2 < u\}$$

[1] In fact, any irrational number has this property [57]. The golden ratio is especially good because it is not too close to a whole number.

[2] This is not a major restriction since, for any $u > 1$, there always exists a prime number in the set $\{u, u + 1, \ldots, 2u\}$. Thus we can enforce this assumption by increasing the value of u by a constant factor.

is a collection of universal hash functions. Clearly, choosing a function uniformly at random from $\mathcal{H}$ can be done easily by choosing two random values $k_1 \in \{1, \ldots, u-1\}$ and $k_2 \in \{0, \ldots, u-1\}$. Thus, we have an implementation of the set ADT with $O(1)$ expected time per operation.

9.2.4 Static Perfect Hashing

The result of Carter and Wegman on universal hashing is very strong, and from a practical point of view, it is probably the strongest result most people will ever need. The only thing that could be improved about their result is to make it deterministic, so that the running times of all operations are $O(1)$ *worst-case*. Unfortunately, this is not possible, as shown by Dietzfelbinger et al. [23].

Since there is no hope of getting $O(1)$ worst-case time for all three set ADT operations, the next best thing would be to have searches that take $O(1)$ worst-case time. In this section we describe the method of Fredman, Komlós and Szemerédi [28]. This is a static data structure that takes as input a set $S \subseteq U$ and builds a data structure of size $O(n)$ that can test if an element x is in S in $O(1)$ worst-case time. Like the universal hash functions from the previous section, this method also requires that u be a prime number. This scheme uses hash functions of the form

$$h_{k,m}(x) = (kx \bmod u)) \bmod m \ .^3$$

Let $B_{k,m}(S,i)$ be the number of elements $x \in S$ such that $h_{k,m}(x) = i$, i.e., the number of elements of S that have hash value i when using the hash function $h_{k,m}$. The function $B_{k,m}$ gives complete information about the distribution of hash values of S. The main lemma used by Fredman et al. is that, if we choose $k \in U$ uniformly at random then

$$\mathrm{E}\left[\sum_{i=0}^{m-1} \binom{B_{k,m}(S,i)}{2}\right] < \frac{n^2}{m} \ . \tag{9.1}$$

There are two important special cases of this result.

In the *sparse case* we take $m = n^2/\alpha$, for some constant $0 < \alpha < 1$. In this case, the expectation in (9.1) is less than α. Therefore, by Markov's inequality, the probability that this sum is greater than or equal to 1 is at most α. But, since this sum is a non-negative integer, then with probability at least $1-\alpha$ it must be equal to 0. In other words, with probability at least $1-\alpha$, $B_{k,m}(S,i) \leq 1$ for all $0 \leq i \leq m-1$, i.e., the hash function $h_{k,m}$ is perfect for S. Of course this implies that we can find a perfect hash function very quickly by trying a small number of random elements $k \in U$ and testing if they result in perfect hash functions. (The expected number of elements that we will have to try is only $1/(1-\alpha)$.) Thus, if we are willing to use quadratic space then we can perform searches in $O(1)$ worst-case time.

In the *dense case* we assume that m is close to n and discover that, for many values of k, the hash values are distributed fairly evenly among the set $1, \ldots, m$. More precisely, if we use a table of size $m = n$, then

$$\mathrm{E}\left[\sum_{i=0}^{m-1} B_{k,m}(S,i)^2\right] \leq 3n \ .$$

[3] Actually, it turns out that any universal hash function also works in the FKS scheme [16, Section 11.5].

By Markov's inequality this means that

$$\Pr\left\{\sum_{i=0}^{m-1} B_{k,m}(S,i)^2 \leq 3n/\alpha\right\} \geq 1-\alpha \ . \tag{9.2}$$

Again, we can quickly find a value of k satisfying (9.2) by testing a few randomly chosen values of k.

These two properties are enough to build a two-level data structure that uses linear space and executes searches in worst-case constant time. We call the following data structure the FKS-α data structure, after its inventors Fredman, Komlós and Szemerédi. At the top level, the data structure consists of an array $A[0],\ldots,A[m-1]$ where $m=n$. The elements of this array are pointers to other arrays $A_0,\ldots,A_{m-1}$, respectively. To decide what will be stored in these other arrays, we build a hash function $h_{k,m}$ that satisfies the conditions of (9.2). This gives us the top-level hash function $h_{k,m}(x) = (kx \bmod u) \bmod m$. Each element $x \in S$ gets stored in the array pointed to by $A[h_{k,m}(x)]$.

What remains is to describe how we use the arrays $A_0,\ldots,A_{m-1}$. Let S_i denote the set of elements $x \in S$ such that $h_{k,m}(s) = i$. The elements of S_i will be stored in A_i. The size of S_i is $n_i = B_{k,m}(S,i)$. To store the elements of S_i we set the size of A_i to $m_i = {n_i}^2/\alpha = B_{k,n}(S,i)^2/\alpha$. Observe that, by (9.2), all the A_i's take up a total space of $O(n)$, i.e., $\sum_{i=0}^{m-1} m_i = O(n)$. Furthermore, by trying a few randomly selected integers we can quickly find a value k_i such that the hash function h_{k_i,m_i} is perfect for S_i. Therefore, we store the element $x \in S_i$ at position $A_i[h_{k_i,m_i}(x)]$ and x is the unique element stored at that location. With this scheme we can search for any value $x \in U$ by computing two hash values $i = h_{k,m}(x)$ and $j = h_{k_i,m_i}(x)$ and checking if x is stored in $A_i[j]$.

Building the array A and computing the values of $n_0,\ldots,n_{m-1}$ takes $O(n)$ expected time since for a given value k we can easily do this in $O(n)$ time and the expected number of values of k that we must try before finding one that satisfies (9.2) is $O(1)$. Similarly, building each subarray A_i takes $O({n_i}^2)$ expected time, resulting in an overall expected running time of $O(n)$. Thus, for any constant $0 < \alpha < 1$, an FKS-α data structure can be constructed in $O(n)$ expected time and this data structure can execute a search for any $x \in U$ in $O(1)$ worst-case time.

9.2.5 Dynamic Perfect Hashing

The FKS-α data structure is nice in that it allows for searches in $O(1)$ time, in the worst case. Unfortunately, it is only static; it does not support insertions or deletions of elements. In this section we describe a result of Dietzfelbinger et al. [23] that shows how the FKS-α data structure can be made dynamic with some judicious use of partial rebuilding (Chapter 10).

The main idea behind the scheme is simple: be lazy at both the upper and lower levels of the FKS-α data structure. That is, rebuild parts of the data structure only when things go wrong. At the top level, we relax the condition that the size m of the upper array A is exactly n and allow A to have size anywhere between n and $2n$. Similarly, at the lower level we allow the array A_i to have a size m_i anywhere between ${n_i}^2/\alpha$ and $2{n_i}^2/\alpha$.

Periodically, we will perform a *global rebuilding* operation in which we remove all n elements from the hash table. Some elements which have previously been marked as deleted will be discarded, thereby reducing the value of n. We put the remaining elements in a list, and recompute a whole new FKS-($\alpha/2$) data structure for the elements in the list. This data structure is identical to the standard FKS-($\alpha/2$) data structure except that, at the top level we use an array of size $m = 2n$.

Searching in this data structure is exactly the same as for the static data structure. To search for an element x we compute $i = h_{k,m}(x)$ and $j = h_{k_i,m_i}(x)$ and look for x at location $A_i[j]$. Thus, searches take $O(1)$ worst-case time.

Deleting in this data structure is done in the laziest manner possible. To delete an element we only search for it and then mark it as deleted. We will use the convention that this type of deletion does not change the value of n since it does not change the number of elements actually stored in the data structure. While doing this, we also keep track of the number of elements that are marked as deleted. When this number exceeds $n/2$ we perform a global rebuilding operation. The global rebuilding operation takes $O(n)$ expected time, but only occurs during one out of every $n/2$ deletions. Therefore, the amortized cost of this operation is $O(1)$ per deletion.

The most complicated part of the data structure is the insertion algorithm and its analysis. To insert a key x we know, because of how the search algorithm works, that we must ultimately store x at location $A_i[j]$ where $i = h_{k,m}(x)$ and $j = h_{k_i,m_i}(x)$. However, several things can go wrong during the insertion of x:

1. The value of n increases by 1, so it may be that n now exceeds m. In this case we perform a global rebuilding operation and we are done.
2. We compute $i = h_{k,m}(x)$ and discover that $\sum_{i=0}^{m-1} {n_i}^2 > 3n/\alpha$. In this case, the hash function $h_{k,m}$ used at the top level is no longer any good since it is producing an overall hash table that is too large. In this case we perform a global rebuilding operation and we are done.
3. We compute $i = h_{k,m}(x)$ and discover that, since the value of n_i just increased by one, ${n_i}^2/\alpha > m_i$. In this case, the array A_i is too small to guarantee that we can quickly find a perfect hash function. To handle this, we copy the elements of A_i into a list L and allocate a new array A_i with the new size $m_i = 2{n_i}^2/\alpha$. We then find a new value k_i such that h_{k_i,m_i} is a perfect hash function for the elements of L and we are done.
4. The array location $A_i[j]$ is already occupied by some other element y. But in this case, we know that A_i is large enough to hold all the elements (otherwise we would already be done after Case 3), but the value k_i being used in the hash function h_{k_i,m_i} is the wrong one since it doesn't give a perfect hash function for S_i. Therefore we simply try new values for k_i until we find a find a value k_i that yields a perfect hash function and we are done.

If none of the preceding 4 cases occurs then we can simply place x at location $A_i[j]$ and we are done.

Handling Case 1 takes $O(n)$ expected time since it involves a global rebuild of the entire data structure. However, Case 1 only happens during one out of every $\Theta(n)$ insertions, so the amortized cost of all occurrences of Case 1 is only $O(1)$ per insertion.

Handling Case 2 also takes $O(n)$ expected time. The question is: How often does Case 2 occur? To answer this question, consider the *phase* that occurs between two consecutive occurrences of Case 1. During this phase, the data structure holds at most m distinct elements. Call this set of elements S. With probability at least $(1 - \alpha)$ the hash function $h_{k,m}$ selected at the beginning of the phase satisfies (9.2) so that Case 2 never occurs during the phase. Similarly, the probability that Case 2 occurs exactly once during the phase is at most $\alpha(1 - \alpha)$. In general, the probability that Case 2 occurs exactly i times during a phase is at most $\alpha^i(1 - \alpha)$. Thus, the expected cost of handling all occurrences of Case 2

during the entire phase is at most

$$\sum_{i=0}^{\infty} \alpha^i (1-\alpha) i \times O(n) = O(n) \ .$$

But since a phase involves $\Theta(n)$ insertions this means that the amortized expected cost of handling Case 2 is $O(1)$ per insertion.

Next we analyze the total cost of handling Case 3. Define a *subphase* as the period of time between two global rebuilding operations triggered either as a result of a deletion, Case 1 or Case 2. We will show that the total cost of handling all occurrences of Case 3 during a subphase is $O(n)$ and since a subphase takes $\Theta(n)$ time anyway this does not contribute to the cost of a subphase by more than a constant factor. When Case 3 occurs at the array A_i it takes $O(m_i)$ time. However, while handling Case 3, m_i increases by a constant factor, so the total cost of handling Case 3 for A_i is dominated by the value of m_i at the end of the subphase. But we maintain the invariant that $\sum_{i=0}^{m-1} m_i = O(n)$ during the entire subphase. Thus, handling all occurrences of Case 3 during a subphase only requires $O(n)$ time.

Finally, we consider the cost of handling Case 4. For a particular array A_i, consider the *subsubphase* between which two occurrences of Case 3 cause A_i to be rebuilt or a global rebuilding operation takes place. During this subsubphase the number of distinct elements that occupy A_i is at most $\alpha\sqrt{m_i}$. Therefore, with probability at least $1-\alpha$ any randomly chosen value of $k_i \in U$ is a perfect hash function for this set. Just as in the analysis of Case 2, this implies that the expected cost of handling all occurrences of Case 3 at A_i during a subsubphase is only $O(m_i)$. Since a subsubphase ends with rebuilding all of A_i or a global rebuilding, at a cost of $\Omega(m_i)$ all the occurrences of Case 4 during a subsubphase do not contribute to the expected cost of the subsubphase by more than a constant factor.

To summarize, we have shown that the expected cost of handling all occurrences of Case 4 is only a constant factor times the cost of handling all occurrences of Case 3. The cost of handling all occurrences of Case 3 is no more than a constant factor times the expected cost of all global rebuilds. The cost of handling all the global rebuilds that occur as a result of Case 2 is no more than a constant factor times the cost of handling all occurrences of global rebuilds that occur as a consequence of Case 1. And finally, the cost of all global rebuilds that occur as a result of Case 1 or of deletions is $O(n)$ for a sequence of n update operations. Therefore, the total expected cost of n update operation is $O(n)$.

9.3 Random Probing

Next we consider hash table implementations under the random probing assumption: Each element x stored in the hash table comes with a random sequence $x_0, x_1, x_2, \ldots$ where each of the x_i is independently and uniformly distributed in $\{1, \ldots, m\}$.[4] We begin with a discussion of the two basic paradigms: hashing with chaining and open addressing. Both these paradigms attempt to store the key x at array position $A[x_0]$. The difference between these two algorithms is their *collision resolution strategy*, i.e., what the algorithms do when a user inserts the key value x but array position $A[x_0]$ already contains some other key.

[4] A variant of the random probing assumption, referred to as the *uniform hashing* assumption, assumes that $x_0, \ldots, x_{m-1}$ is a random permutation of $0, \ldots, m-1$.

9.3.1 Hashing with Chaining

In *hashing with chaining*, a collision is resolved by allowing more than one element to live at each position in the table. Each entry in the array A is a pointer to the head of a linked list. To insert the value x, we simply append it to the list $A[x_0]$. To search for the element x, we perform a linear search in the list $A[x_0]$. To delete the element x, we search for x in the list $A[x_0]$ and splice it out.

It is clear that insertions take $O(1)$ time, even in the worst case. For searching and deletion, the running time is proportional to a constant plus the length of the list stored at $A[x_0]$. Notice that each of the at most n elements not equal to x is stored in $A[x_0]$ with probability $1/m$, so the expected length of $A[x_0]$ is either $\alpha = n/m$ (if x is not contained in the table) or $1 + (n-1)/m$ (if x is contained in the table). Thus, the expected cost of searching for or deleting an element is $O(1+\alpha)$.

The above analysis shows us that hashing with chaining supports the three set ADT operations in $O(1)$ expected time per operation, as long as the occupancy, α, is a constant. It is worth noting that this does not require that the value of α be less than 1.

If we would like more detailed information about the cost of searching, we might also ask about the *worst-case search time* defined as

$$W = \max\{\text{length of the list stored at } A[i] : 0 \leq i \leq m-1\} \ .$$

It is very easy to prove something quite strong about W using only the fact that the length of each list $A[i]$ is a binomial$(n, 1/m)$ random variable. Using Chernoff's bounds on the tail of the binomial distribution [13], this immediately implies that

$$\Pr\{\text{length of } A[i] \geq \alpha c \ln n\} \leq n^{-\Omega(c)} \ .$$

Combining this with Boole's inequality ($\Pr\{A \text{ or } B\} \leq \Pr\{A\} + \Pr\{B\}$) we obtain

$$\Pr\{W \geq \alpha c \ln n\} \leq n \times n^{-\Omega(c)} = n^{-\Omega(c)} \ .$$

Thus, with very high probability, the worst-case search time is logarithmic in n. This also implies that $\mathrm{E}[W] = O(\log n)$. The distribution of W has been carefully studied and it is known that, *with high probability*, i.e., with probability $1 - o(1)$, $W = (1+o(1)) \ln n / \ln \ln n$ [33, 38].[5] Gonnet has proven a more accurate result that $W = \Gamma^{-1}(n) - 3/2 + o(1)$ with high probability. Devroye [18] shows that similar results hold even when the distribution of x_0 is not uniform.

9.3.2 Hashing with Open Addressing

Hashing with open addressing differs from hashing with chaining in that each table position $A[i]$ is allowed to store only one value. When a collision occurs at table position i, one of the two elements involved in the collision must move on to the next element in its probe sequence. In order to implement this efficiently and correctly we require a method of marking elements as deleted. This method could be an auxiliary array that contains one bit for each element of A, but usually the same result can be achieved by using a special key value **del** that does not correspond to any valid key.

[5] Here, and throughout this chapter, if an asymptotic notation does not contain a variable then the variable that tends to infinity is implicitly n. Thus, for example, $o(1)$ is the set of non-negative functions of n that tend to 0 as $n \to \infty$.

To search for an element x in the hash table we look for x at positions $A[x_0]$, $A[x_1]$, $A[x_2]$, and so on until we either (1) find x, in which case we are done or (2) find an empty table position $A[x_i]$ that is not marked as deleted, in which case we can be sure that x is not stored in the table (otherwise it would be stored at position x_i). To delete an element x from the hash table we first search for x. If we find x at table location $A[x_i]$ we then simply mark $A[x_i]$ as deleted. To insert a value x into the hash table we examine table positions $A[x_0]$, $A[x_1]$, $A[x_2]$, and so on until we find a table position $A[x_i]$ that is either empty or marked as deleted and we store the value x in $A[x_i]$.

Consider the cost of inserting an element x using this method. Let i_x denote the smallest value i such that x_{i_x} is either empty or marked as deleted when we insert x. Thus, the cost of inserting x is a constant plus i_x. The probability that the table position x_0 is occupied is at most α so, with probability at least $1-\alpha$, $i_x = 0$. Using the same reasoning, the probability that we store x at position x_i is at most

$$\Pr\{i_x = i\} \leq \alpha^i(1-\alpha) \tag{9.3}$$

since the table locations $x_0, \ldots, x_{i-1}$ must be occupied, the table location x_i must not be occupied and the x_i are independent. Thus, the expected number of steps taken by the insertion algorithm is

$$\sum_{i=1}^{\infty} i \Pr\{i_x = i\} = (1-\alpha)\sum_{i=1}^{\infty} i\alpha^{i-1} = 1/(1-\alpha)$$

for any constant $0 < \alpha < 1$. The cost of searching for x and deleting x are both proportional to the cost of inserting x, so the expected cost of each of these operations is $O(1/(1-\alpha))$.[6]

We should compare this with the cost of hashing with chaining. In hashing with chaining,the occupancy α has very little effect on the cost of operations. Indeed, any constant α, even greater than 1 results in $O(1)$ time per operation. In contrast, open addressing is very dependent on the value of α. If we take $\alpha > 1$ then the expected cost of insertion using open addressing is infinite since the insertion algorithm never finds an empty table position. Of course, the advantage of hashing with chaining is that it does not require lists at each of the $A[i]$. Therefore, the overhead of list pointers is saved and this extra space can be used instead to maintain the invariant that the occupancy α is a constant strictly less than 1.

Next we consider the worst case search time of hashing with open addressing. That is, we study the value $W = \max\{i_x : x \text{ is stored in the table at location } i_x\}$. Using (9.3) and Boole's inequality it follows almost immediately that

$$\Pr\{W > c\log n\} \leq n^{-\Omega(c)}.$$

Thus, with very high probability, W, the worst case search time, is $O(\log n)$. Tighter bounds on W are known when the probe sequences $x_0, \ldots, x_{m-1}$ are random permutations of $0, \ldots, m-1$. In this case, Gonnet[29] shows that

$$\mathrm{E}[W] = \log_{1/\alpha} n - \log_{1/\alpha}(\log_{1/\alpha} n) + O(1).$$

[6]Note that the expected cost of searching for or deleting an element x is proportional to the value of α *at the time* x *was inserted.* If many deletions have taken place, this may be quite different than the current value of α.

Open addressing under the random probing assumption has many nice theoretical properties and is easy to analyze. Unfortunately, it is often criticized as being an unrealistic model because it requires a long random sequences $x_0, x_1, x_2, \ldots$ for each element x that is to be stored or searched for. Several variants of open addressing discussed in the next few sections try to overcome this problem by using only a few random values.

9.3.3 Linear Probing

Linear probing is a variant of open addressing that requires less randomness. To obtain the probe sequence $x_0, x_1, x_2, \ldots$ we start with a random element $x_0 \in \{0, \ldots, m-1\}$. The element x_i, $i > 0$ is given by $x_i = (i + x_0) \bmod m$. That is, one first tries to find x at location x_0 and if that fails then one looks at $(x_0 + 1) \bmod m$, $(x_0 + 2) \bmod m$ and so on.

The performance of linear probing is discussed by Knuth [37] who shows that the expected number of probes performed during an unsuccessful search is at most

$$(1 + 1/(1 - \alpha)^2)/2$$

and the expected number of probes performed during a successful search is at most

$$(1 + 1/(1 - \alpha))/2 \ .$$

This is not quite as good as for standard hashing with open addressing, especially in the unsuccessful case.

Linear probing suffers from the problem of *primary clustering*. If j consecutive array entries are occupied then a newly inserted element will have probability j/m of hashing to one of these entries. This results in $j + 1$ consecutive array entries being occupied and increases the probability (to $(j + 1)/m$) of another newly inserted element landing in this cluster. Thus, large clusters of consecutive elements have a tendency to grow larger.

9.3.4 Quadratic Probing

Quadratic probing is similar to linear probing; an element x determines its entire probe sequence based on a single random choice, x_0. Quadratic probing uses the probe sequence $x_0, (x_0 + k_1 + k_2) \bmod m, (x_0 + 2k_1 + 2^2 k_2) \bmod m, \ldots$. In general, the ith element in the probe sequence is $x_i = (x_0 + ik_1 + i^2 k_2) \bmod m$. Thus, the final location of an element depends quadratically on how many steps were required to insert it. This method seems to work much better in practice than linear probing, but requires a careful choice of m, k_1 and k_2 so that the probe sequence contains every element of $\{0, \ldots, m-1\}$.

The improved performance of quadratic probing is due to the fact that if there are two elements x and y such that $x_i = y_j$ then it is not necessarily true (as it is with linear probing) that $x_{i+1} = y_{j+1}$. However, if $x_0 = y_0$ then x and y will have exactly the same probe sequence. This lesser phenomenon is called *secondary clustering*. Note that this secondary clustering phenomenon implies that neither linear nor quadratic probing can hope to perform any better than hashing with chaining. This is because all the elements that have the same initial hash x_0 are contained in an implicit chain. In the case of linear probing, this chain is defined by the sequence $x_0, x_0 + 1, x_0 + 2, \ldots$ while for quadratic probing it is defined by the sequence $x_0, x_0 + k_1 + k_2, x_0 + 2k_1 + 4k_2, \ldots$

9.3.5 Double Hashing

Double hashing is another method of open addressing that uses two hash values x_0 and x_1. Here x_0 is in the set $\{0, \ldots, m-1\}$ and x_1 is in the subset of $\{1, \ldots, m-1\}$ that is

relatively prime to m. With double hashing, the probe sequence for element x becomes $x_0, (x_0 + x_1) \bmod m, (x_0 + 2x_1) \bmod m, \ldots$. In general, $x_i = (x_0 + ix_1) \bmod m$, for $i > 0$. The expected number of probes required by double hashing seems difficult to determine exactly. Guibas has proven that, asymptotically, and for occupancy $\alpha \leq .31$, the performance of double hashing is asymptotically equivalent to that of uniform hashing. Empirically, the performance of double hashing matches that of open addressing with random probing regardless of the occupancy α [37].

9.3.6 Brent's Method

Brent's method [5] is a heuristic that attempts to minimize the average time for a successful search in a hash table with open addressing. Although originally described in the context of double hashing (Section 9.3.5) Brent's method applies to any open addressing scheme. The *age* of an element x stored in an open addressing hash table is the minimum value i such that x is stored at $A[x_i]$. In other words, the age is one less than the number of locations we will probe when searching for x.

Brent's method attempts to minimize the total age of all elements in the hash table. To insert the element x we proceed as follows: We find the smallest value i such that $A[x_i]$ is empty; this is where standard open-addressing would insert x. Consider the element y stored at location $A[x_{i-2}]$. This element is stored there because $y_j = x_{i-2}$, for some $j \geq 0$. We check if the array location $A[y_{j+1}]$ is empty and, if so, we move y to location $A[y_{j+1}]$ and store x at location $A[x_{i-2}]$. Note that, compared to standard open addressing, this decreases the total age by 1. In general, Brent's method checks, for each $2 \leq k \leq i$ the array entry $A[x_{i-k}]$ to see if the element y stored there can be moved to any of $A[y_{j+1}], A[y_{j+2}], \ldots, A[y_{j+k-1}]$ to make room for x. If so, this represents a decrease in the total age of all elements in the table and is performed.

Although Brent's method seems to work well in practice, it is difficult to analyze theoretically. Some theoretical analysis of Brent's method applied to double hashing is given by Gonnet and Munro [31]. Lyon [44], Munro and Celis [49] and Poblete [52] describe some variants of Brent's method.

9.3.7 Multiple-Choice Hashing

It is worth stepping back at this point and revisiting the comparison between hash tables and binary search trees. For balanced binary search trees, the average cost of searching for an element is $O(\log n)$. Indeed, it easy to see that for at least $n/2$ of the elements, the cost of searching for those elements is $\Omega(\log n)$. In comparison, for both the random probing schemes discussed so far, the expected cost of search for an element is $O(1)$. However, there are a handful of elements whose search cost is $\Theta(\log n / \log \log n)$ or $\Theta(\log n)$ depending on whether hashing with chaining or open addressing is used, respectively. Thus there is an inversion: Most operations on a binary search tree cost $\Theta(\log n)$ but a few elements (close to the root) can be accessed in $O(1)$ time. Most operations on a hash table take $O(1)$ time but a few elements (in long chains or with long probe sequences) require $\Theta(\log n / \log \log n)$ or $\Theta(\log n)$ time to access. In the next few sections we consider variations on hashing with chaining and open addressing that attempt to reduce the worst-case search time W.

Multiple-choice hashing is hashing with chaining in which, during insertion, the element x has the choice of $d \geq 2$ different lists in which it can be stored. In particular, when we insert x we look at the lengths of the lists pointed to by $A[x_0], \ldots, A[x_{d-1}]$ and append x to $A[x_i]$, $0 \leq i < d$ such that the length of the list pointed to by $A[x_i]$ is minimum. When searching for x, we search for x in each of the lists $A[x_0], \ldots, A[x_{d-1}]$ *in parallel*. That is, we

look at the first elements of each list, then the second elements of each list, and so on until we find x. As before, to delete x we first search for it and then delete it from whichever list we find it in.

It is easy to see that the expected cost of searching for an element x is $O(d)$ since the expected length of each the d lists is $O(1)$. More interestingly, the worst case search time is bounded by $O(dW)$ where W is the length of the longest list. Azar et al. [3] show that

$$\mathrm{E}[W] = \frac{\ln \ln n}{\ln d} + O(1) \ . \tag{9.4}$$

Thus, the expected worst case search time for multiple-choice hashing is $O(\log \log n)$ for any constant $d \geq 2$.

9.3.8 Asymmetric Hashing

Asymmetric hashing is a variant of multiple-choice hashing in which the hash table is split into d blocks, each of size n/d. (Assume, for simplicity, that n is a multiple of d.) The probe value x_i, $0 \leq i < d$ is drawn uniformly from $\{in/d, \ldots, (i+1)n/d - 1\}$. As with multiple-choice hashing, to insert x the algorithm examines the lengths of the lists $A[x_0], A[x_1], \ldots, A[x_{d-1}]$ and appends x to the shortest of these lists. In the case of ties, it appends x to the list with smallest index. Searching and deletion are done exactly as in multiple-choice hashing.

Vöcking [64] shows that, with asymmetric hashing the expected length of the longest list is

$$\mathrm{E}[W] \leq \frac{\ln \ln n}{d \ln \phi_d} + O(1) \ .$$

The function ϕ_d is a generalization of the *golden ratio*, so that $\phi_2 = (1 + \sqrt{5})/2$. Note that this improves significantly on standard multiple-choice hashing (9.4) for larger values of d.

9.3.9 LCFS Hashing

LCFS hashing is a form of open addressing that changes the collision resolution strategy.[7] Reviewing the algorithm for hashing with open addressing reveals that when two elements collide, priority is given to the first element inserted into the hash table and subsequent elements must move on. Thus, hashing with open addressing could also be referred to as *FCFS (first-come first-served) hashing*.

With LCFS (last-come first-served) hashing, collision resolution is done in exactly the opposite way. When we insert an element x, we always place it at location x_0. If position x_0 is already occupied by some element y because $y_j = x_0$ then we place y at location y_{j+1}, possibly displacing some element z, and so on.

Poblete and Munro [53] show that, after inserting n elements into an initially empty table, the expected worst case search time is bounded above by

$$\mathrm{E}[W] \leq 1 + \Gamma^{-1}(\alpha n) \left(1 + \frac{\ln \ln(1/(1-\alpha))}{\ln \Gamma^{-1}(\alpha n)} + O\left(\frac{1}{\ln^2 \Gamma^{-1}(\alpha n)} \right) \right) \ ,$$

[7] Amble and Knuth [1] were the first to suggest that, with open addressing, any collision resolution strategy could be used.

where Γ is the gamma function and

$$\Gamma^{-1}(\alpha n) = \frac{\ln n}{\ln \ln n}\left(1 + \frac{\ln \ln \ln n}{\ln \ln n} + O\left(\frac{1}{\ln \ln n}\right)\right) \quad .$$

Historically, LCFS hashing is the first version of open addressing that was shown to have an expected worst-case search time that is $o(\log n)$.

9.3.10 Robin-Hood Hashing

Robin-Hood hashing [9, 10, 61] is a form of open addressing that attempts to equalize the search times of elements by using a fairer collision resolution strategy. During insertion, if we are trying to place element x at position x_i and there is already an element y stored at position $y_j = x_i$ then the "younger" of the two elements must move on. More precisely, if $i \leq j$ then we will try to insert x at position x_{i+1}, x_{i+2} and so on. Otherwise, we will store x at position x_i and try to to insert y at positions y_{j+1}, y_{j+2} and so on.

Devroye et al. [20] show that, after performing n insertions on an initially empty table of size $m = \alpha n$ using the Robin-Hood insertion algorithm, the worst case search time has expected value

$$\mathrm{E}[W] = \Theta(\log \log n)$$

and this bound is tight. Thus, Robin-Hood hashing is a form of open addressing that has doubly-logarithmic worst-case search time. This makes it competitive with the multiple-choice hashing method of Section 9.3.7.

9.3.11 Cuckoo Hashing

Cuckoo hashing [50] is a form of multiple choice hashing in which each element x lives in one of two tables A or B, each of size $m = n/\alpha$. The element x will either be stored at location $A[x_A]$ or $B[x_B]$. There are no other options. This makes searching for x an $O(1)$ time operation since we need only check two array locations.

The insertion algorithm for cuckoo hashing proceeds as follows:[8] Store x at location $A[x_A]$. If $A[x_A]$ was previously occupied by some element y then store y at location $B[y_B]$. If $B[y_B]$ was previously occupied by some element z then store z at location $A[z_A]$, and so on. This process ends when we place an element into a previously empty table slot or when it has gone on for more than $c \log n$ steps. In the former case, the insertion of x completes successfully. In the latter case the insertion is considered a failure, and the entire hash table is reconstructed from scratch using a new probe sequence for each element in the table. That is, if this reconstruction process has happened i times then the two hash values we use for an element x are $x_A = x_{2i}$ and $x_B = x_{2i+1}$.

Pagh and Rodler [50] (see also Devroye and Morin [19]) show that, during the insertion of n elements, the probability of requiring a reconstruction is $O(1/n)$. This, combined with the fact that the expected insertion time is $O(1)$ shows that the expected cost of n insertions in a Cuckoo hashing table is $O(n)$. Thus, Cuckoo hashing offers a somewhat simpler alternative to the dynamic perfect hashing algorithms of Section 9.2.5.

[8] The algorithm takes its name from the large but lazy cuckoo bird which, rather than building its own nest, steals the nest of another bird forcing the other bird to move on.

9.4 Historical Notes

In this section we present some of the history of hash tables. The idea of hashing seems to have been discovered simultaneously by two groups of researchers. Knuth [37] cites an internal IBM memorandum in January 1953 by H. P. Luhn that suggested the use of hashing with chaining. Building on Luhn's work, A. D. Linh suggested a method of open addressing that assigns the probe sequence $x_0, \lfloor x_0/10 \rfloor, \lfloor x_0/100 \rfloor, \lfloor x_0/1000 \rfloor, \ldots$ to the element x.

At approximately the same time, another group of researchers at IBM: G. M. Amdahl, E. M. Boehme, N. Rochester and A. L. Samuel implemented hashing in an assembly program for the IBM 701 computer. Amdahl is credited with the idea of open addressing with linear probing.

The first published work on hash tables was by A. I. Dumey [24], who described hashing with chaining and discussed the idea of using remainder modulo a prime as a hash function. Ershov [25], working in Russia and independently of Amdahl, described open addressing with linear probing.

Peterson [51] wrote the first major article discussing the problem of searching in large files and coined the term "open addressing." Buchholz [7] also gave a survey of the searching problem with a very good discussion of hashing techniques at the time. Theoretical analyses of linear probing were first presented by Konheim and Weiss [39] and Podderjugin. Another, very influential, survey of hashing was given by Morris [47]. Morris' survey is the first published use of the word "hashing" although it was already in common use by practitioners at that time.

9.5 Other Developments

The study of hash tables has a long history and many researchers have proposed methods of implementing hash tables. Because of this, the current chapter is necessarily incomplete. (At the time of writing, the hash.bib bibliography on hashing contains over 800 entries.) We have summarized only a handful of the major results on hash tables in internal memory. In this section we provide a few references to the literature for some of the other results. For more information on hashing, Knuth [37], Vitter and Flajolet [63], Vitter and Chen [62], and Gonnet and Baeza-Yates [30] are useful references.

Brent's method (Section 9.3.6) is a collision resolution strategy for open addressing that reduces the expected search time for a successful search in a hash table with open addressing. Several other methods exist that either reduce the expected or worst-case search time. These include *binary tree hashing* [31, 45], *optimal hashing* [31, 54, 55], Robin-Hood hashing (Section 9.3.10), and *min-max hashing* [9, 29]. One interesting method, due to Celis [9], applies to any open addressing scheme. The idea is to study the distribution of the ages of elements in the hash table, i.e., the distribution give by

$$D_i = \Pr\{x \text{ is stored at position } x_i\}$$

and start searching for x at the locations at which we are most likely to find it, rather than searching the table positions $x_0, x_1, x_2 \ldots$ in order.

Perfect hash functions seem to have been first studied by Sprugnoli [58] who gave some heuristic number theoretic constructions of minimal perfect hash functions for small data sets. Sprugnoli is responsible for the terms "perfect hash function" and "minimal perfect hash function." A number of other researchers have presented algorithms for discovering minimal and near-minimal perfect hash functions. Examples include Anderson and Anderson [2], Cichelli [14, 15], Chang [11, 12], Gori and Soda [32], and Sager [56]. Berman et al. [4]

and Körner and Marton [40] discuss the theoretical limitations of perfect hash functions. A comprehensive, and recent, survey of perfect hashing and minimal perfect hashing is given by Czech et al. [17].

Tarjan and Yao [60] describe a set ADT implementation that gives $O(\log u/\log n)$ worst-case access time. It is obtained by combining a trie (Chapter 28) of degree n with a compression scheme for arrays of size n^2 that contain only n non-zero elements. (The trie has $O(n)$ nodes each of which has n pointers to children, but there are only a total of $O(n)$ children.) Although their result is superseded by the results of Fredman et al. [28] discussed in Section 9.2.4, they are the first theoretical results on worst-case search time for hash tables.

Dynamic perfect hashing (Section 9.2.5) and cuckoo hashing (Section 9.3.11) are methods of achieving $O(1)$ worst case search time in a dynamic setting. Several other methods have been proposed [6, 21, 22].

Yao [65] studies the *membership problem*. Given a set $S \subseteq U$, devise a data structure that can determine for any $x \in U$ whether x is contained in S. Yao shows how, under various conditions, this problem can be solved using a very small number of memory accesses per query. However, Yao's algorithms sometimes derive the fact that an element x is in S without actually finding x. Thus, they don't solve the set ADT problem discussed at the beginning of this chapter since they can not recover a pointer to x.

The "power of two random choices," as used in multiple-choice hashing, (Section 9.3.7) has many applications in computer science. Karp, Luby and Meyer auf der Heide [34, 35] were the first to use this paradigm for simulating PRAM computers on computers with fewer processors. The book chapter by Mitzenmacher et al. [46] surveys results and applications of this technique.

A number of table implementations have been proposed that are suitable for managing hash tables in external memory. Here, the goal is to reduce the number of disk blocks that must be accessed during an operation, where a disk block can typically hold a large number of elements. These schemes include *linear hashing* [43], *dynamic hashing* [41], *virtual hashing* [42], *extendible hashing* [26], *cascade hashing* [36], and *spiral storage* [48]. In terms of hashing, the main difference between internal memory and external memory is that, in internal memory, an array is allocated at a specific size and this can not be changed later. In contrast, an external memory file may be appended to or be truncated to increase or decrease its size, respectively. Thus, hash table implementations for external memory can avoid the periodic global rebuilding operations used in internal memory hash table implementations.

Acknowledgment

The author is supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.

[2] M. R. Anderson and M. G. Anderson. Comments on perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 22(2):104, 1979.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[4] F. Berman, M. E. Bock, E. Dittert, M. J. O'Donnell, and D. Plank. Collections of functions for perfect hashing. *SIAM Journal on Computing*, 15(2):604–618, 1986.

[5] R. P. Brent. Reducing the storage time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

[6] A. Brodnik and J. I. Munro. Membership in constant time and almost minimum space. *SIAM Journal on Computing*, 28:1627–1640, 1999.

[7] W. Buchholz. File organization and addressing. *IBM Systems Journal*, 2(1):86–111, 1963.

[8] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[9] P. Celis. Robin Hood hashing. Technical Report CS-86-14, Computer Science Department, University of Waterloo, 1986.

[10] P. Celis, P.-Å. Larson, and J. I. Munro. Robin Hood hashing. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS'85)*, pages 281–288. IEEE Press, 1985.

[11] C. C. Chang. An ordered minimal perfect hashing scheme based upon Euler's theorem. *Information Sciences*, 32(3):165–172, 1984.

[12] C. C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM*, 27(4):384–387, 1984.

[13] H. Chernoff. A measure of the asymptotic efficient of tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[14] R. J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[15] R. J. Cichelli. On Cichelli's minimal perfect hash functions method. *Communications of the ACM*, 23(12):728–729, 1980.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachussetts, 2nd edition, 2001.

[17] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

[18] L. Devroye. The expected length of the longest probe sequence when the distribution is not uniform. *Journal of Algorithms*, 6:1–9, 1985.

[19] L. Devroye and P. Morin. Cuckoo hashing: Further analysis. *Information Processing Letters*, 86(4):215–219, 2002.

[20] L. Devroye, P. Morin, and A. Viola. On worst case Robin-Hood hashing. *SIAM Journal on Computing*. To appear.

[21] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming (ICALP'90)*, pages 6–19, 1990.

[22] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP'92)*, pages 235–246, 1992.

[23] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[24] A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[25] A. P. Ershov. On programming of arithmetic operations. *Doklady Akademii Nauk SSSR*, 118(3):427–430, 1958.

[26] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[27] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, New York, 1968.

[28] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[29] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, pages 289–304, 1981.

[30] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1991.

[31] G. H. Gonnet and J. I. Munro. Efficient ordering of hash tables. *SIAM Journal on Computing*, 8(3):463–478, 1979.

[32] M. Gori and G. Soda. An algebraic approach to Cichelli's perfect hashing. *Bit*, 29(1):2–13, 1989.

[33] N. L. Johnson and S. Kotz. *Urn Models and Their Applications*. John Wiley & Sons, New York, 1977.

[34] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. Technical Report TR-93-040, International Computer Science Institute, Berkeley, CA, USA, 1993.

[35] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on the Theory of Computing (STOC'92)*, pages 318–326. ACM Press, 1992.

[36] P. Kjellberg and T. U. Zahle. Cascade hashing. In *Proceedings of the 10th International Conference on Very Large Data Bases (VLDB'80)*, pages 481–492. Morgan Kaufmann, 1984.

[37] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1997.

[38] V. F. Kolchin, B. A. Sevastyanov, and V. P. Chistyakov. *Random Allocations*. John Wiley & Sons, New York, 1978.

[39] A. G. Konheim and B. Weiss. An occupancy discipline and its applications. *SIAM Journal of Applied Mathematics*, 14:1266–1274, 1966.

[40] J. Körner and K. Marton. New bounds for perfect hashing via information theory. *European Journal of Combinatorics*, 9(6):523–530, 1988.

[41] P.-Å. Larson. Dynamic hashing. *Bit*, 18(2):184–201, 1978.

[42] W. Litwin. Virtual hashing: A dynamically changing hashing. In *Proceedings of the 4th International Conference on Very Large Data Bases (VLDB'80)*, pages 517–523. IEEE Computer Society, 1978.

[43] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80)*, pages 212–223. IEEE Computer Society, 1980.

[44] G. E. Lyon. Packed scatter tables. *Communications of the ACM*, 21(10):857–865, 1978.

[45] E. G. Mallach. Scatter storage techniques: A unifying viewpoint and a method for reducing retrieval times. *The Computer Journal*, 20(2):137–140, 1977.

[46] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In P. Pardalos, S. Rajasekaran, and J. Rolim, editors, *Handbook of Randomized Computing*, volume 1, chapter 9. Kluwer, 2001.

[47] R. Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, 1968.

[48] J. K. Mullin. Spiral storage: Efficient dynamic hashing with constant performance. *The Computer Journal*, 28(3):330–334, 1985.

[49] J. I. Munro and P. Celis. Techniques for collision resolution in hash tables with open addressing. In *Proceedings of 1986 Fall Joint Computer Conference*, pages 601–610. ACM Press, 1999.

[50] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer-Verlag, 2001.

[51] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

[52] P. V. Poblete. Studies on hash coding with open addressing. M. Math Essay, University of Waterloo, 1977.

[53] P. V. Poblete and J. Ian Munro. Last-come-first-served hashing. *Journal of Algorithms*, 10:228–248, 1989.

[54] G. Poonan. Optimal placement of entries in hash tables. In *ACM Computer Science Conference (Abstract Only)*, volume 25, 1976. (Also DEC Internal Tech. Rept. LRD-1, Digital Equipment Corp. Maynard Mass).

[55] R. L. Rivest. Optimal arrangement of keys in a hash table. *Journal of the ACM*, 25(2):200–209, 1978.

[56] T. J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM*, 28(5):523–532, 1985.

[57] V. T. Sós. On the theory of diophantine approximations. i. *Acta Mathematica Budapest*, 8:461–471, 1957.

[58] R. Sprugnoli. Perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, 1977.

[59] S. Świerczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Mathematica*, 46:187–189, 1958.

[60] R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

[61] A. Viola and P. V. Poblete. Analysis of linear probing hashing with buckets. *Algorithmica*, 21:37–71, 1998.

[62] J. S. Vitter and W.-C. Chen. *The Design and Analysis of Coalesced Hashing*. Oxford University Press, Oxford, UK, 1987.

[63] J. S. Vitter and P. Flajolet. Analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. North Holland, 1990.

[64] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 131–140. IEEE Press, 1999.

[65] A. C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.