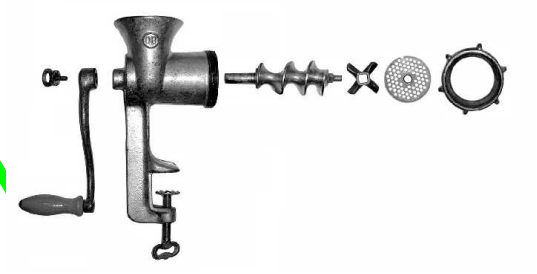
4

Hash Tables and Associative Arrays



If you want to get a book from the central library of the University of Karlsruhe, you have to order the book in advance. The library personnel fetch the book from the stacks and deliver it to a room with 100 shelves. You find your book on a shelf numbered with the last *two digits of your library card. Why the last digits and not the leading digits? Probably because this distributes the books more evenly among the shelves. The library cards are numbered consecutively as students sign up, and the University of Karlsruhe was founded in 1825. Therefore, the students enrolled at the same time are likely to have the same leading digits in their card number, and only a few shelves would be in use if the leading digits were used.*

The subject of this chapter is the robust and efficient implementation of the above "delivery shelf data structure". In computer science, this data structure is known as a *hash*[1] *table*. Hash tables are one implementation of *associative arrays*, or *dictionaries*. The other implementation is the tree data structures which we shall study in Chap. 7. An associative array is an array with a potentially infinite or at least very large index set, out of which only a small number of indices are actually in use. For example, the potential indices may be all strings, and the indices in use may be all identifiers used in a particular C++ program. Or the potential indices may be all ways of placing chess pieces on a chess board, and the indices in use may be the placements required in the analysis of a particular game. Associative arrays are versatile data structures. Compilers use them for their *symbol table*, which associates identifiers with information about them. Combinatorial search programs often use them for detecting whether a situation has already been looked at. For example, chess programs have to deal with the fact that board positions can be reached by different sequences of moves. However, each position needs to be evaluated only once. The solution is to store positions in an associative array. One of the most widely used implementations of the *join* operation in relational databases temporarily stores one of the participating relations in an associative array. Scripting languages such as AWK

[1] Photograph of the mincer above by Kku, Rainer Zenz (Wikipedia), Licence CC-by-SA 2.5.

[7] and `Perl` [203] use associative arrays as their *main* data structure. In all of the examples above, the associative array is usually implemented as a hash table. The exercises in this section ask you to develop some further uses of associative arrays.

Formally, an associative array S stores a set of elements. Each element e has an associated key $key(e) \in Key$. We assume keys to be unique, i.e., distinct elements have distinct keys. Associative arrays support the following operations:

- $S.insert(e : Element)$: $S := S \cup \{e\}$.
- $S.remove(k : Key)$: $S := S \setminus \{e\}$, where e is the unique element with $key(e) = k$.
- $S.find(k : Key)$: If there is an $e \in S$ with $key(e) = k$, return e; otherwise, return $\perp$.

In addition, we assume a mechanism that allows us to retrieve all elements in S. Since this *forall* operation is usually easy to implement, we discuss it only in the exercises. Observe that the *find* operation is essentially the random access operator for an array; hence the name "associative array". *Key* is the set of potential array indices, and the elements of S are the indices in use at any particular time. Throughout this chapter, we use n to denote the size of S, and N to denote the size of *Key*. In a typical application of associative arrays, N is humongous and hence the use of an array of size N is out of the question. We are aiming for solutions which use space $O(n)$.

In the library example, *Key* is the set of all library card numbers, and elements are the book orders. Another precomputer example is provided by an English–German dictionary. The keys are English words, and an element is an English word together with its German translations.

The basic idea behind the hash table implementation of associative arrays is simple. We use a *hash function* h to map the set *Key* of potential array indices to a small range $0..m-1$ of integers. We also have an array t with index set $0..m-1$, the *hash table*. In order to keep the space requirement low, we want m to be about the number of elements in S. The hash function associates with each element e a *hash value* $h(key(e))$. In order to simplify the notation, we write $h(e)$ instead of $h(key(e))$ for the hash value of e. In the library example, h maps each library card number to its last two digits. Ideally, we would like to store element e in the table entry $t[h(e)]$. If this works, we obtain constant execution time[2] for our three operations *insert*, *remove*, and *find*.

Unfortunately, storing e in $t[h(e)]$ will not always work, as several elements might *collide*, i.e., map to the same table entry. The library example suggests a fix: allow several book orders to go to the same shelf. The entire shelf then has to be searched to find a particular order. A generalization of this fix leads to *hashing with chaining*. We store a set of elements in each table entry, and implement the set using singly linked lists. Section 4.1 analyzes hashing with chaining using some rather optimistic (and hence unrealistic) assumptions about the properties of the hash function. In this model, we achieve constant expected time for all three dictionary operations.

In Sect. 4.2, we drop the unrealistic assumptions and construct hash functions that come with (probabilistic) performance guarantees. Even our simple examples show

[2] Strictly speaking, we have to add additional terms for evaluating the hash function and for moving elements around. To simplify the notation, we assume in this chapter that all of this takes constant time.

that finding good hash functions is nontrivial. For example, if we apply the least-significant-digit idea from the library example to an English–German dictionary, we might come up with a hash function based on the last four letters of a word. But then we would have many collisions for words ending in "tion", "able", etc.

We can simplify hash tables (but not their analysis) by returning to the original idea of storing all elements in the table itself. When a newly inserted element e finds the entry $t[h(x)]$ occupied, it scans the table until a free entry is found. In the library example, assume that shelves can hold exactly one book. The librarians would then use adjacent shelves to store books that map to the same delivery shelf. Section 4.3 elaborates on this idea, which is known as *hashing with open addressing and linear probing*.

Why are hash tables called hash tables? The dictionary defines "to hash" as "to chop up, as of potatoes". This is exactly what hash functions usually do. For example, if keys are strings, the hash function may chop up the string into pieces of fixed size, interpret each fixed-size piece as a number, and then compute a single number from the sequence of numbers. A good hash function creates disorder and, in this way, avoids collisions.

Exercise 4.1. Assume you are given a set M of pairs of integers. M defines a binary relation R_M. Use an associative array to check whether R_M is symmetric. A relation is symmetric if $\forall(a,b) \in M : (b,a) \in M$.

Exercise 4.2. Write a program that reads a text file and outputs the 100 most frequent words in the text.

Exercise 4.3 (a billing system). Assume you have a large file consisting of triples (transaction, price, customer ID). Explain how to compute the total payment due for each customer. Your algorithm should run in linear time.

Exercise 4.4 (scanning a hash table). Show how to realize the *forall* operation for hashing with chaining and for hashing with open addressing and linear probing. What is the running time of your solution?

4.1 Hashing with Chaining

Hashing with chaining maintains an array t of linear lists (see Fig. 4.1). The associative-array operations are easy to implement. To insert an element e, we insert it somewhere in the sequence $t[h(e)]$. To remove an element with key k, we scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, we remove it and return. To find the element with key k, we also scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, we return it. Otherwise, we return $\perp$.

Insertions take constant time. The space consumption is $O(n+m)$. To remove or find a key k, we have to scan the sequence $t[h(k)]$. In the worst case, for example if *find* looks for an element that is not there, the entire list has to be scanned. If we are

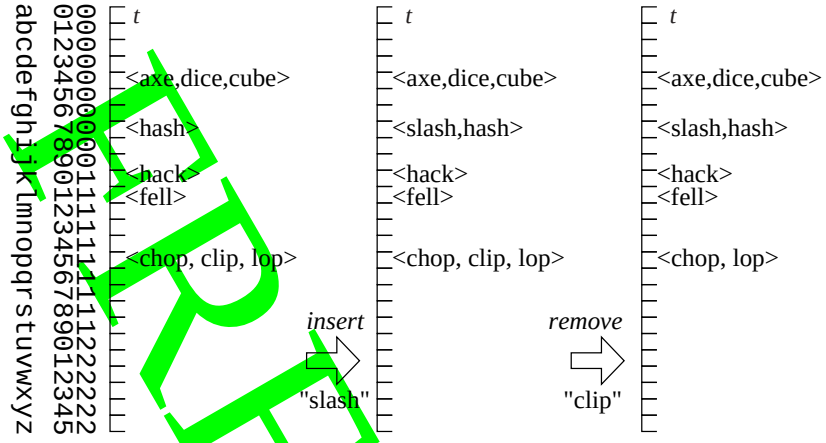


Fig. 4.1. Hashing with chaining. We have a table t of sequences. The figure shows an example where a set of words (short synonyms of "hash") is stored using a hash function that maps the last character to the integers 0..25. We see that this hash function is not very good

unlucky, all elements are mapped to the same table entry and the execution time is $\Theta(n)$. So, in the worst case, hashing with chaining is no better than linear lists.

Are there hash functions that guarantee that all sequences are short? The answer is clearly no. A hash function maps the set of keys to the range $0..m-1$, and hence for every hash function there is always a set of N/m keys that all map to the same table entry. In most applications, $n < N/m$ and hence hashing can always deteriorate to a linear search. We shall study three approaches to dealing with the worst-case behavior. The first approach is average-case analysis. We shall study this approach in this section. The second approach is to use randomization, and to choose the hash function at random from a collection of hash functions. We shall study this approach in this section and the next. The third approach is to change the algorithm. For example, we could make the hash function depend on the set of keys in actual use. We shall investigate this approach in Sect. 4.5 and shall show that it leads to good worst-case behavior.

Let H be the set of all functions from *Key* to $0..m-1$. We assume that the hash function h is chosen randomly[3] from H and shall show that for any fixed set S of n keys, the expected execution time of *remove* or *find* will be $O(1+n/m)$.

Theorem 4.1. *If n elements are stored in a hash table with m entries and a random hash function is used, the expected execution time of remove or find is $O(1+n/m)$.*

[3] This assumption is completely unrealistic. There are m^N functions in H, and hence it requires $N \log m$ bits to specify a function in H. This defeats the goal of reducing the space requirement from N to n.

Proof. The proof requires the probabilistic concepts of random variables, their expectation, and the linearity of expectations as described in Sect. A.3. Consider the execution time of *remove* or *find* for a fixed key k. Both need constant time plus the time for scanning the sequence $t[h(k)]$. Hence the expected execution time is $O(1+E[X])$, where the random variable X stands for the length of the sequence $t[h(k)]$. Let S be the set of n elements stored in the hash table. For each $e \in S$, let X_e be the *indicator* variable which tells us whether e hashes to the same location as k, i.e., $X_e = 1$ if $h(e) = h(k)$ and $X_e = 0$ otherwise. In shorthand, $X_e = [h(e) = h(k)]$. We have $X = \sum_{e \in S} X_e$. Using the linearity of expectations, we obtain

$$E[X] = E[\sum_{e \in S} X_e] = \sum_{e \in S} E[X_e] = \sum_{e \in S} \text{prob}(X_i = 1) \ .$$

A random hash function maps e to all m table entries with the same probability, independent of $h(k)$. Hence, $\text{prob}(X_e = 1) = 1/m$ and therefore $E[X] = n/m$. Thus, the expected execution time of *find* and *remove* is $O(1+n/m)$. □

We can achieve a linear space requirement and a constant expected execution time of all three operations by guaranteeing that $m = \Theta(n)$ at all times. Adaptive reallocation, as described for unbounded arrays in Sect. 3.2, is the appropriate technique.

Exercise 4.5 (unbounded hash tables). Explain how to guarantee $m = \Theta(n)$ in hashing with chaining. You may assume the existence of a hash function $h' : \mathit{Key} \to \mathbb{N}$. Set $h(k) = h'(k) \bmod m$ and use adaptive reallocation.

Exercise 4.6 (waste of space). The waste of space in hashing with chaining is due to empty table entries. Assuming a random hash function, compute the expected number of empty table entries as a function of m and n. Hint: define indicator random variables $Y_0, \ldots, Y_{m-1}$, where $Y_i = 1$ if $t[i]$ is empty.

Exercise 4.7 (average-case behavior). Assume that the hash function distributes the set of potential keys evenly over the table, i.e., for each i, $0 \le i \le m-1$, we have $|\{k \in \mathit{Key} : h(k) = i\}| \le \lceil N/m \rceil$. Assume also that a random set S of n keys is stored in the table, i.e., S is a random subset of *Key* of size n. Show that for any table position i, the expected number of elements in S that hash to i is at most $\lceil N/m \rceil \cdot n/N \approx n/m$.

4.2 Universal Hashing

Theorem 4.1 is unsatisfactory, as it presupposes that the hash function is chosen randomly from the set of all functions[4] from keys to table positions. The class of all such functions is much too big to be useful. We shall show in this section that the same performance can be obtained with much smaller classes of hash functions. The families presented in this section are so small that a member can be specified in constant space. Moreover, the functions are easy to evaluate.

[4] We shall usually talk about a *class* of functions or a *family* of functions in this chapter, and reserve the word "set" for the set of keys stored in the hash table.

Definition 4.2. *Let c be a positive constant. A family H of functions from Key to $0..m-1$ is called* c-universal *if any two distinct keys collide with a probability of at most c/m, i.e., for all x, y in Key with $x \neq y$,*

$$|\{h \in H : h(x) = h(y)\}| \leq \frac{c}{m}|H| .$$

In other words, for random $h \in H$,

$$\text{prob}(h(x) = h(y)) \leq \frac{c}{m} .$$

This definition has been constructed such that the proof of Theorem 4.1 can be extended.

Theorem 4.3. *If n elements are stored in a hash table with m entries using hashing with chaining and a random hash function from a c-universal family is used, the expected execution time of remove or find is $O(1+cn/m)$.*

Proof. We can reuse the proof of Theorem 4.1 almost word for word. Consider the execution time of *remove* or *find* for a fixed key k. Both need constant time plus the time for scanning the sequence $t[h(k)]$. Hence the expected execution time is $O(1+E[X])$, where the random variable X stands for the length of the sequence $t[h(k)]$. Let S be the set of n elements stored in the hash table. For each $e \in S$, let X_e be the indicator variable which tells us whether e hashes to the same location as k, i.e., $X_e = 1$ if $h(e) = h(k)$ and $X_e = 0$ otherwise. In shorthand, $X_e = [h(e) = h(k)]$. We have $X = \sum_{e \in S} X_e$. Using the linearity of expectations, we obtain

$$E[X] = E[\sum_{e \in S} X_e] = \sum_{e \in S} E[X_e] = \sum_{e \in S} \text{prob}(X_i = 1) .$$

Since h is chosen uniformly from a c-universal class, we have $\text{prob}(X_e = 1) \leq c/m$, and hence $E[X] = cn/m$. Thus, the expected execution time of *find* and *remove* is $O(1+cn/m)$. □

Now it remains to find c-universal families of hash functions that are easy to construct and easy to evaluate. We shall describe a simple and quite practical 1-universal family in detail and give further examples in the exercises. We assume that our keys are bit strings of a certain fixed length; in the exercises, we discuss how the fixed-length assumption can be overcome. We also assume that the table size m is a prime number. Why a prime number? Because arithmetic modulo a prime is particularly nice; in particular, the set $\mathbb{Z}_m = \{0,\ldots,m-1\}$ of numbers modulo m form a field.[5] Let $w = \lfloor \log m \rfloor$. We subdivide the keys into pieces of w bits each, say k pieces. We interpret each piece as an integer in the range $0..2^w - 1$ and keys as k-tuples of such integers. For a key $\mathbf{x}$, we write $\mathbf{x} = (x_1,\ldots,x_k)$ to denote its partition

[5] A field is a set with special elements 0 and 1 and with addition and multiplication operations. Addition and multiplication satisfy the usual laws known for the field of rational numbers.

into pieces. Each x_i lies in $0..2^w-1$. We can now define our class of hash functions. For each $\mathbf{a}=(a_1,\ldots,a_k)\in\{0..m-1\}^k$, we define a function $h_{\mathbf{a}}$ from *Key* to $0..m-1$ as follows. Let $\mathbf{x}=(x_1,\ldots,x_k)$ be a key and let $\mathbf{a}\cdot\mathbf{x}=\sum_{i=1}^k a_i x_i$ denote the scalar product of $\mathbf{a}$ and $\mathbf{x}$. Then

$$h_{\mathbf{a}}(x)=\mathbf{a}\cdot\mathbf{x} \bmod m\ .$$

It is time for an example. Let $m=17$ and $k=4$. Then $w=4$ and we view keys as 4-tuples of integers in the range $0..15$, for example $\mathbf{x}=(11,7,4,3)$. A hash function is specified by a 4-tuple of integers in the range $0..16$, for example $\mathbf{a}=(2,4,7,16)$. Then $h_{\mathbf{a}}(\mathbf{x})=(2\cdot 11+4\cdot 7+7\cdot 4+16\cdot 3) \bmod 17=7$.

Theorem 4.4.

$$H=\left\{h_{\mathbf{a}}:\mathbf{a}\in\{0..m-1\}^k\right\}$$

is a 1-universal family of hash functions if m is prime.

In other words, the scalar product between a tuple representation of a key and a random vector modulo m defines a good hash function.

Proof. Consider two distinct keys $\mathbf{x}=(x_1,\ldots,x_k)$ and $\mathbf{y}=(y_1,\ldots,y_k)$. To determine $\text{prob}(h_{\mathbf{a}}(x)=h_{\mathbf{a}}(\mathbf{y}))$, we count the number of choices for $\mathbf{a}$ such that $h_{\mathbf{a}}(\mathbf{x})=h_{\mathbf{a}}(\mathbf{y})$. Fix an index j such that $x_j\neq y_j$. Then $(x_j-y_j)\not\equiv 0 (\text{mod } m)$, and hence any equation of the form $a_j(x_j-y_j)\equiv b (\text{mod } m)$, where $b\in\mathbb{Z}_m$, has a unique solution in a_j, namely $a_j\equiv(x_j-y_j)^{-1}b (\text{mod } m)$. Here $(x_j-y_j)^{-1}$ denotes the *multiplicative inverse*[6] of (x_j-y_j).

We claim that for each choice of the a_i's with $i\neq j$, there is exactly one choice of a_j such that $h_{\mathbf{a}}(\mathbf{x})=h_{\mathbf{a}}(\mathbf{y})$. Indeed,

$$\begin{aligned} h_{\mathbf{a}}(\mathbf{x})=h_{\mathbf{a}}(\mathbf{y}) &\Leftrightarrow \sum_{1\le i\le k} a_i x_i \equiv \sum_{1\le i\le k} a_i y_i && (\text{mod } m)\\ &\Leftrightarrow a_j(x_j-y_j)\equiv \sum_{i\neq j} a_i(y_i-x_i) && (\text{mod } m)\\ &\Leftrightarrow \quad a_j\equiv (y_j-x_j)^{-1}\sum_{i\neq j} a_i(x_i-y_i) && (\text{mod } m)\ . \end{aligned}$$

There are m^{k-1} ways to choose the a_i with $i\neq j$, and for each such choice there is a unique choice for a_j. Since the total number of choices for $\mathbf{a}$ is m^k, we obtain

$$\text{prob}(h_{\mathbf{a}}(x)=h_{\mathbf{a}}(\mathbf{y}))=\frac{m^{k-1}}{m^k}=\frac{1}{m}\ .$$

□

Is it a serious restriction that we need prime table sizes? At first glance, yes. We certainly cannot burden users with the task of providing appropriate primes. Also, when we adaptively grow or shrink an array, it is not clear how to obtain prime

[6] In a field, any element $z\neq 0$ has a unique multiplicative inverse, i.e., there is a unique element z^{-1} such that $z^{-1}\cdot z=1$. Multiplicative inverses allow one to solve linear equations of the form $zx=b$, where $z\neq 0$. The solution is $x=z^{-1}b$.

numbers for the new value of m. A closer look shows that the problem is easy to resolve. The easiest solution is to consult a table of primes. An analytical solution is not much harder to obtain. First, number theory [82] tells us that primes are abundant. More precisely, for any integer k there is a prime in the interval $[k^3,(k+1)^3]$. So, if we are aiming for a table size of about m, we determine k such that $k^3 \le m \le (k+1)^3$ and then search for a prime in this interval. How does this search work? Any nonprime in the interval must have a divisor which is at most $\sqrt{(k+1)^3} = (k+1)^{3/2}$. We therefore iterate over the numbers from 1 to $(k+1)^{3/2}$, and for each such j remove its multiples in $[k^3,(k+1)^3]$. For each fixed j, this takes time $((k+1)^3 - k^3)/j = \mathrm{O}\bigl(k^2/j\bigr)$. The total time required is

$$\sum_{j \le (k+1)^{3/2}} \mathrm{O}\left(\frac{k^2}{j}\right) = k^2 \sum_{j \le (k+1)^{3/2}} \mathrm{O}\left(\frac{1}{j}\right)$$
$$= \mathrm{O}\Bigl(k^2 \ln\bigl((k+1)^{3/2}\bigr)\Bigr) = \mathrm{O}\bigl(k^2 \ln k\bigr) = o(m)$$

and hence is negligible compared with the cost of initializing a table of size m. The second equality in the equation above uses the harmonic sum (A.12).

Exercise 4.8 (strings as keys). Implement the universal family $H^{\cdot}$ for strings. Assume that each character requires eight bits (= a byte). You may assume that the table size is at least $m = 257$. The time for evaluating a hash function should be proportional to the length of the string being processed. Input strings may have arbitrary lengths not known in advance. Hint: compute the random vector **a** lazily, extending it only when needed.

Exercise 4.9 (hashing using bit matrix multiplication). For this exercise, keys are bit strings of length k, i.e., $Key = \{0,1\}^k$, and the table size m is a power of two, say $m = 2^w$. Each $w \times k$ matrix M with entries in $\{0,1\}$ defines a hash function h_M. For $x \in \{0,1\}^k$, let $h_M(x) = Mx \bmod 2$, i.e., $h_M(x)$ is a matrix–vector product computed modulo 2. The resulting w-bit vector is interpreted as a number in $[0\,..\,m-1]$. Let

$$H^{\mathrm{lin}} = \left\{ h_M : M \in \{0,1\}^{w \times k} \right\}.$$

For $M = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$ and $x = (1,0,0,1)^T$, we have $Mx \bmod 2 = (0,1)^T$. Note that multiplication modulo two is the logical AND operation, and that addition modulo two is the logical exclusive-OR operation $\oplus$.

(a) Explain how $h_M(x)$ can be evaluated using k bit-parallel exclusive-OR operations. Hint: the ones in x select columns of M. Add the selected columns.
(b) Explain how $h_M(x)$ can be evaluated using w bit-parallel AND operations and w parity operations. Many machines provide an instruction $parity(y)$ that returns one if the number of ones in y is odd, and zero otherwise. Hint: multiply each row of M by x.

(c) We now want to show that H^{lin} is 1-universal. (1) Show that for any two keys $x \neq y$, any bit position j, where x and y differ, and any choice of the columns M_i of the matrix with $i \neq j$, there is exactly one choice of a column M_j such that $h_M(x) = h_M(y)$. (2) Count the number of ways to choose $k-1$ columns of M. (3) Count the total number of ways to choose M. (4) Compute the probability $\text{prob}(h_M(x) = h_M(y))$ for $x \neq y$ if M is chosen randomly.

***Exercise 4.10 (more matrix multiplication).** Define a class of hash functions

$$H^{\times} = \left\{ h_M : M \in \{0..p-1\}^{w \times k} \right\}$$

that generalizes the class H^{lin} by using arithmetic modulo p for some prime number p. Show that $H^{\times}$ is 1-universal. Explain how $H^{\cdot}$ is a special case of $H^{\times}$.

Exercise 4.11 (simple linear hash functions). Assume that $Key = 0..p-1 = \mathbb{Z}_p$ for some prime number p. For $a, b \in \mathbb{Z}_p$, let $h_{(a,b)}(x) = ((ax+b) \bmod p) \bmod m$, and $m \leq p$. For example, if $p = 97$ and $m = 8$, we have $h_{(23,73)}(2) = ((23 \cdot 2 + 73) \bmod 97) \bmod 8 = 22 \bmod 8 = 6$. Let

$$H^{*} = \left\{ h_{(a,b)} : a, b \in 0..p-1 \right\} .$$

Show that this family is $(\lceil p/m \rceil / (p/m))^2$-universal.

Exercise 4.12 (continuation). Show that the following holds for the class H^{*} defined in the previous exercise. For any pair of distinct keys x and y and any i and j in $0..m-1$, $\text{prob}(h_{(a,b)}(x) = i \text{ and } h_{(a,b)}(y) = j) \leq c/m^2$ for some constant c.

Exercise 4.13 (a counterexample). Let $Key = 0..p-1$, and consider the set of hash functions

$$H^{\text{fool}} = \left\{ h_{(a,b)} : a, b \in 0..p-1 \right\}$$

with $h_{(a,b)}(x) = (ax+b) \bmod m$. Show that there is a set S of $\lceil p/m \rceil$ keys such that for any two keys x and y in S, all functions in H^{fool} map x and y to the same value. Hint: let $S = \{0, m, 2m, \ldots, \lfloor p/m \rfloor m\}$.

Exercise 4.14 (table size 2^{ℓ}). Let $Key = 0..2^k - 1$. Show that the family of hash functions

$$H^{\gg} = \left\{ h_a : 0 < a < 2^k \wedge a \text{ is odd} \right\}$$

with $h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell}$ is 2-universal. Hint: see [53].

Exercise 4.15 (table lookup). Let $m = 2^w$, and view keys as $k+1$-tuples, where the zeroth element is a w-bit number and the remaining elements are a-bit numbers for some small constant a. A hash function is defined by tables t_1 to t_k, each having a size $s = 2^a$ and storing bit strings of length w. We then have

$$h_{\oplus(t_1,\ldots,t_k)}((x_0,x_1,\ldots,x_k)) = x_0 \oplus \bigoplus_{i=1}^{k} t_i[x_i] \ ,$$

i.e., x_i selects an element in table t_i, and then the bitwise exclusive-OR of x_0 and the $t_i[x_i]$ is formed. Show that

$$H^{\oplus[]} = \left\{h_{(t_1,\ldots,t_k)} : t_i \in \{0..m-1\}^s\right\}$$

is 1-universal.

4.3 Hashing with Linear Probing

Hashing with chaining is categorized as a *closed* hashing approach because each table entry has to cope with all elements hashing to it. In contrast, *open* hashing schemes open up other table entries to take the overflow from overloaded fellow entries. This added flexibility allows us to do away with secondary data structures such as linked lists – all elements are stored directly in table entries. Many ways of organizing open hashing have been investigated [153]. We shall explore only the simplest scheme. Unused entries are filled with a special element $\perp$. An element e is stored in the entry $t[h(e)]$ or further to the right. But we only go away from the index $h(e)$ with good reason: if e is stored in $t[i]$ with $i > h(e)$, then the positions $h(e)$ to $i-1$ are occupied by other elements.

The implementations of *insert* and *find* are trivial. To insert an element e, we linearly scan the table starting at $t[h(e)]$, until a free entry is found, where e is then stored. Figure 4.2 gives an example. Similarly, to find an element e, we scan the table, starting at $t[h(e)]$, until the element is found. The search is aborted when an empty table entry is encountered. So far, this sounds easy enough, but we have to deal with one complication. What happens if we reach the end of the table during an insertion? We choose a very simple fix by allocating m' table entries to the right of the largest index produced by the hash function h. For "benign" hash functions, it should be sufficient to choose m' much smaller than m in order to avoid table overflows. Alternatively, one may treat the table as a cyclic array; see Exercise 4.16 and Sect. 3.4. This alternative is more robust but slightly slower.

The implementation of *remove* is nontrivial. Simply overwriting the element with $\perp$ does not suffice, as it may destroy the invariant. Assume that $h(x) = h(z)$, $h(y) = h(x)+1$, and x, y, and z are inserted in this order. Then z is stored at position $h(x)+2$. Overwriting y with $\perp$ will make z inaccessible. There are three solutions. First, we can disallow removals. Second, we can mark y but not actually remove it. Searches are allowed to stop at $\perp$, but not at marked elements. The problem with this approach is that the number of nonempty cells (occupied or marked) keeps increasing, so that searches eventually become slow. This can be mitigated only by introducing the additional complication of periodic reorganizations of the table. Third, we can actively restore the invariant. Assume that we want to remove the element at i. We overwrite it with $\perp$ leaving a "hole". We then scan the entries to the right

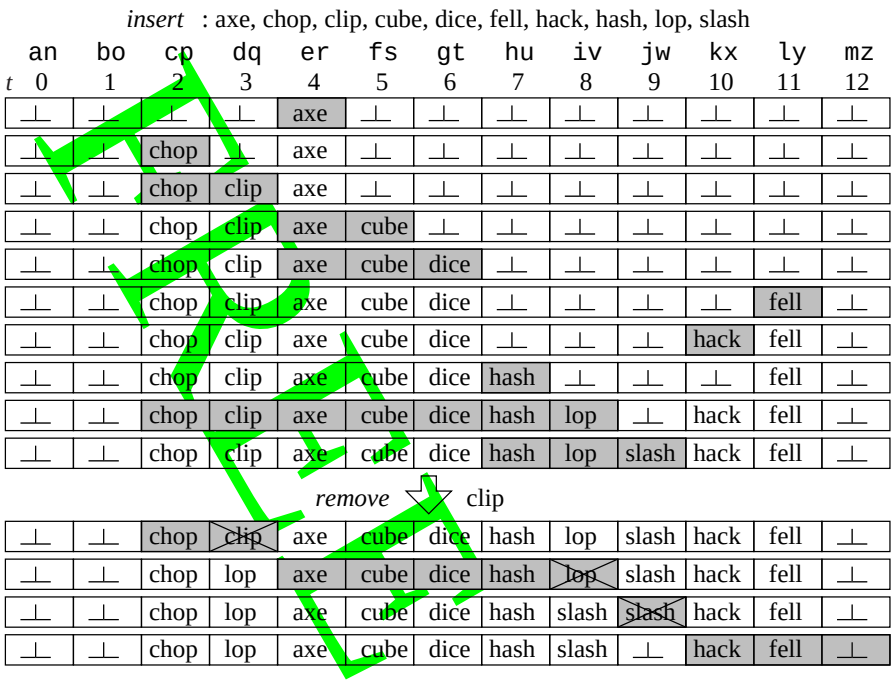


Fig. 4.2. Hashing with linear probing. We have a table t with 13 entries storing synonyms of "(to) hash". The hash function maps the last character of the word to the integers 0..12 as indicated above the table: a and n are mapped to 0, b and o are mapped to 1, and so on. First, the words are inserted in alphabetical order. Then "clip" is removed. The figure shows the state changes of the table. Gray areas show the range that is scanned between the state changes

of i to check for violations of the invariant. We set j to $i+1$. If $t[j] = \bot$, we are finished. Otherwise, let f be the element stored in $t[j]$. If $h(f) > i$, there is nothing to do and we increment j. If $h(f) \leq i$, leaving the hole would violate the invariant, and f would not be found anymore. We therefore move f to $t[i]$ and write $\bot$ into $t[j]$. In other words, we swap f and the hole. We set the hole position i to its new position j and continue with $j := j+1$. Figure 4.2 gives an example.

Exercise 4.16 (cyclic linear probing). Implement a variant of linear probing, where the table size is m rather than $m+m'$. To avoid overflow at the right-hand end of the array, make probing wrap around. (1) Adapt *insert* and *remove* by replacing increments with $i := i+1 \bmod m$. (2) Specify a predicate $between(i,j,k)$ that is true if and only if i is cyclically between j and k. (3) Reformulate the invariant using *between*. (4) Adapt *remove*.

Exercise 4.17 (unbounded linear probing). Implement unbounded hash tables using linear probing and universal hash functions. Pick a new random hash function whenever the table is reallocated. Let α, β, and γ denote constants with $1 < \gamma < \beta <$

α that we are free to choose. Keep track of the number of stored elements n. Expand the table to $m = \beta n$ if $n > m/\gamma$. Shrink the table to $m = \beta n$ if $n < m/\alpha$. If you do not use cyclic probing as in Exercise 4.16, set $m' = \delta m$ for some $\delta < 1$ and reallocate the table if the right-hand end should overflow.

4.4 Chaining Versus Linear Probing

We have seen two different approaches to hash tables, chaining and linear probing. Which one is better? This question is beyond theoretical analysis, as the answer depends on the intended use and many technical parameters. We shall therefore discuss some qualitative issues and report on some experiments performed by us.

An advantage of chaining is referential integrity. Subsequent find operations for the same element will return the same location in memory, and hence references to the results of find operations can be established. In contrast, linear probing moves elements during element removal and hence invalidates references to them.

An advantage of linear probing is that each table access touches a contiguous piece of memory. The memory subsystems of modern processors are optimized for this kind of access pattern, whereas they are quite slow at chasing pointers when the data does not fit into cache memory. A disadvantage of linear probing is that search times become high when the number of elements approaches the table size. For chaining, the expected access time remains small. On the other hand, chaining wastes space on pointers that linear probing could use for a larger table. A fair comparison must be based on space consumption and not just on table size.

We have implemented both approaches and performed extensive experiments. The outcome was that both techniques performed almost equally well when they were given the same amount of memory. The differences were so small that details of the implementation, compiler, operating system, and machine used could reverse the picture. Hence we do not report exact figures.

However, we found chaining harder to implement. Only the optimizations discussed in Sect. 4.6 made it competitive with linear probing. Chaining is much slower if the implementation is sloppy or memory management is not implemented well.

4.5 *Perfect Hashing

The hashing schemes discussed so far guarantee only *expected* constant time for the operations *find*, *insert*, and *remove*. This makes them unsuitable for real-time applications that require a worst-case guarantee. In this section, we shall study *perfect hashing*, which guarantees constant worst-case time for *find*. To keep things simple, we shall restrict ourselves to the *static* case, where we consider a fixed set S of n elements with keys k_1 to k_n.

In this section, we use H_m to denote a family of c-universal hash functions with range $0..m-1$. In Exercise 4.11, it is shown that 2-universal classes exist for every

m. For $h \in H_m$, we use $C(h)$ to denote the number of collisions produced by h, i.e., the number of pairs of distinct keys in S which are mapped to the same position:

$$C(h) = \{(x,y) : x,y \in S,\ x \neq y \text{ and } h(x) = h(y)\} \ .$$

As a first step, we derive a bound on the expectation of $C(h)$.

Lemma 4.5. *$\mathrm{E}[C(h)] \leq cn(n-1)/m$. Also, for at least half of the functions $h \in H_m$, we have $C(h) \leq 2cn(n-1)/m$.*

Proof. We define $n(n-1)$ indicator random variables $X_{ij}(h)$. For $i \neq j$, let $X_{ij}(h) = 1$ iff $h(k_i) = h(k_j)$. Then $C(h) = \sum_{ij} X_{ij}(h)$, and hence

$$\mathrm{E}[C] = \mathrm{E}[\sum_{ij} X_{ij}] = \sum_{ij} \mathrm{E}[X_{ij}] = \sum_{ij} \mathrm{prob}(X_{ij} = 1) \leq n(n-1) \cdot c/m \ ,$$

where the second equality follows from the linearity of expectations (see (A.2)) and the last equality follows from the universality of H_m. The second claim follows from Markov's inequality (A.4). □

If we are willing to work with a quadratic-size table, our problem is solved.

Lemma 4.6. *If $m \geq cn(n-1)+1$, at least half of the functions $h \in H_m$ operate injectively on S.*

Proof. By Lemma 4.5, we have $C(h) < 2$ for half of the functions in H_m. Since $C(h)$ is even, $C(h) < 2$ implies $C(h) = 0$, and so h operates injectively on S. □

So we choose a random $h \in H_m$ with $m \geq cn(n-1)+1$ and check whether it is injective on S. If not, we repeat the exercise. After an average of two trials, we are successful.

In the remainder of this section, we show how to bring the table size down to linear. The idea is to use a two-stage mapping of keys (see Fig. 4.3). The first stage maps keys to buckets of constant average size. The second stage uses a quadratic amount of space for each bucket. We use the information about $C(h)$ to bound the number of keys hashing to any table location. For $\ell \in 0..m-1$ and $h \in H_m$, let B_ℓ^h be the elements in S that are mapped to ℓ by h and let b_ℓ^h be the cardinality of B_ℓ^h.

Lemma 4.7. *$C(h) = \sum_\ell b_\ell^h(b_\ell^h - 1)$.*

Proof. For any ℓ, the keys in B_ℓ^h give rise to $b_\ell^h(b_\ell^h - 1)$ pairs of keys mapping to the same location. Summation over ℓ completes the proof. □

The construction of the perfect hash function is now as follows. Let α be a constant, which we shall fix later. We choose a hash function $h \in H_{\lceil \alpha n \rceil}$ to split S into subsets B_ℓ. Of course, we choose h to be in the good half of $H_{\lceil \alpha n \rceil}$, i.e., we choose $h \in H_{\lceil \alpha n \rceil}$ with $C(h) \leq 2cn(n-1)/\lceil \alpha n \rceil \leq 2cn/\alpha$. For each ℓ, let B_ℓ be the elements in S mapped to ℓ and let $b_\ell = |B_\ell|$.

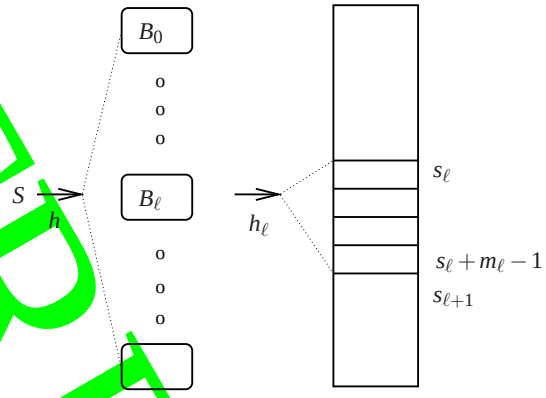


Fig. 4.3. Perfect hashing. The top-level hash function h splits S into subsets $B_0, \ldots, B_\ell, \ldots$. Let $b_\ell = |B_\ell|$ and $m_\ell = cb_\ell(b_\ell - 1) + 1$. The function h_ℓ maps B_ℓ injectively into a table of size m_ℓ. We arrange the subtables into a single table. The subtable for B_ℓ then starts at position $s_\ell = m_0 + \ldots + m_{\ell-1}$ and ends at position $s_\ell + m_\ell - 1$

Now consider any B_ℓ. Let $m_\ell = cb_\ell(b_\ell - 1) + 1$. We choose a function $h_\ell \in H_{m_\ell}$ which maps B_ℓ injectively into $0..m_\ell - 1$. Half of the functions in H_{m_ℓ} have this property by Lemma 4.6 applied to B_ℓ. In other words, h_ℓ maps B_ℓ injectively into a table of size m_ℓ. We stack the various tables on top of each other to obtain one large table of size $\sum_\ell m_\ell$. In this large table, the subtable for B_ℓ starts at position $s_\ell = m_0 + m_1 + \ldots + m_{\ell-1}$. Then

$\ell := h(x)$; **return** $s_\ell + h_\ell(x)$

computes an injective function on S. This function is bounded by

$$\begin{aligned}\sum_\ell m_\ell &\le \lceil \alpha n \rceil + c \cdot \sum_\ell b_\ell(b_\ell - 1) \\ &\le 1 + \alpha n + c \cdot C(h) \\ &\le 1 + \alpha n + c \cdot 2cn/\alpha \\ &\le 1 + (\alpha + 2c^2/\alpha)n\ ,\end{aligned}$$

and hence we have constructed a perfect hash function that maps S into a linearly sized range, namely $0..(\alpha + 2c^2/\alpha)n$. In the derivation above, the first inequality uses the definition of the m_ℓ's, the second inequality uses Lemma 4.7, and the third inequality uses $C(h) \le 2cn/\alpha$. The choice $\alpha = \sqrt{2}c$ minimizes the size of the range. For $c = 1$, the size of the range is $2\sqrt{2}n$.

Theorem 4.8. *For any set of n keys, a perfect hash function with range $0..2\sqrt{2}n$ can be constructed in linear expected time.*

Constructions with smaller ranges are known. Also, it is possible to support insertions and deletions.

Exercise 4.18 (dynamization). We outline a scheme for "dynamization" here. Consider a fixed S, and choose $h \in H_{2\lceil \alpha n \rceil}$. For any ℓ, let $m_\ell = 2cb_\ell(b_\ell - 1) + 1$, i.e., all m's are chosen to be twice as large as in the static scheme. Construct a perfect hash function as above. Insertion of a new x is handled as follows. Assume that h maps x onto ℓ. If h_ℓ is no longer injective, choose a new h_ℓ. If b_ℓ becomes so large that $m_\ell = cb_\ell(b_\ell - 1) + 1$, choose a new h.

4.6 Implementation Notes

Although hashing is an algorithmically simple concept, a clean, efficient, robust implementation can be surprisingly nontrivial. Less surprisingly, the hash functions are the most important issue. Most applications seem to use simple, very fast hash functions based on exclusive-OR, shifting, and table lookup rather than universal hash functions; see, for example, `www.burtleburtle.net/bob/hash/doobs.html` or search for "hash table" on the Internet. Although these functions seem to work well in practice, we believe that the universal families of hash functions described in Sect. 4.2 are competitive. Unfortunately, there is no implementation study covering all of the fastest families. Thorup [191] implemented a fast family with additional properties. In particular, the family $H^{\oplus[]}$ considered in Exercise 4.15 should be suitable for integer keys, and Exercise 4.8 formulates a good function for strings. It might be possible to implement the latter function to run particularly fast using the SIMD instructions of modern processors that allow the parallel execution of several operations.

Hashing with chaining uses only very specialized operations on sequences, for which singly linked lists are ideally suited. Since these lists are extremely short, some deviations from the implementation scheme described in Sect. 3.1 are in order. In particular, it would be wasteful to store a dummy item with each list. Instead, one should use a single shared dummy item to mark the ends of all lists. This item can then be used as a sentinel element for *find* and *remove*, as in the function *findNext* in Sect. 3.1.1. This trick not only saves space, but also makes it likely that the dummy item will reside in the cache memory.

With respect to the first element of the lists, there are two alternatives. One can either use a table of pointers and store the first element outside the table, or store the first element of each list directly in the table. We refer to these alternatives as *slim tables* and *fat tables*, respectively. Fat tables are usually faster and more space-efficient. Slim tables are superior when the elements are very large. Observe that a slim table wastes the space occupied by m pointers and that a fat table wastes the space of the unoccupied table positions (see Exercise 4.6). Slim tables also have the advantage of referential integrity even when tables are reallocated. We have already observed this complication for unbounded arrays in Sect. 3.6.

Comparing the space consumption of hashing with chaining and hashing with linear probing is even more subtle than is outlined in Sect. 4.4. On the one hand, linked lists burden the memory management with many small pieces of allocated memory; see Sect. 3.1.1 for a discussion of memory management for linked lists.

On the other hand, implementations of unbounded hash tables based on chaining can avoid occupying two tables during reallocation by using the following method. First, concatenate all lists into a single list L. Deallocate the old table. Only then, allocate the new table. Finally, scan L, moving the elements to the new table.

Exercise 4.19. Implement hashing with chaining and hashing with linear probing on your own machine using your favorite programming language. Compare their performance experimentally. Also, compare your implementations with hash tables available in software libraries. Use elements of size eight bytes.

Exercise 4.20 (large elements). Repeat the above measurements with element sizes of 32 and 128. Also, add an implementation of *slim chaining*, where table entries only store pointers to the first list element.

Exercise 4.21 (large keys). Discuss the impact of large keys on the relative merits of chaining versus linear probing. Which variant will profit? Why?

Exercise 4.22. Implement a hash table data type for very large tables stored in a file. Should you use chaining or linear probing? Why?

4.6.1 C++

The C++ standard library does not (yet) define a hash table data type. However, the popular implementation by SGI (`http://www.sgi.com/tech/stl/`) offers several variants: *hash_set*, *hash_map*, *hash_multiset*, and *hash_multimap*.[7] Here "set" stands for the kind of interface used in this chapter, whereas a "map" is an associative array indexed by keys. The prefix "multi" indicates data types that allow multiple elements with the same key. Hash functions are implemented as *function objects*, i.e., the class *hash*<*T*> overloads the operator "`( )`" so that an object can be used like a function. The reason for this approach is that it allows the hash function to store internal state such as random coefficients.

LEDA [118] offers several hashing-based implementations of dictionaries. The class $h\_array\langle Key, T\rangle$ offers associative arrays for storing objects of type T. This class requires a user-defined hash function *int Hash*(*Key*&) that returns an integer value which is then mapped to a table index by LEDA. The implementation uses hashing with chaining and adapts the table size to the number of elements stored. The class *map* is similar but uses a built-in hash function.

Exercise 4.23 (associative arrays). Implement a C++ class for associative arrays. Support *operator*`[]` for any index type that supports a hash function. Make sure that `H[x]=...` works as expected if x is the key of a new element.

4.6.2 Java

The class *java.util.HashMap* implements unbounded hash tables using the function *hashCode* defined in the class *Object* as a hash function.

[7] Future versions of the standard will have these data types using the word "*unordered*" instead of the word "*hash*".

4.7 Historical Notes and Further Findings

Hashing with chaining and hashing with linear probing were used as early as the 1950s [153]. The analysis of hashing began soon after. In the 1960s and 1970s, average-case analysis in the spirit of Theorem 4.1 and Exercise 4.7 prevailed. Various schemes for random sets of keys or random hash functions were analyzed. An early survey paper was written by Morris [143]. The book [112] contains a wealth of material. For example, it analyzes linear probing assuming random hash functions. Let n denote the number of elements stored, let m denote the size of the table and set $\alpha = n/m$. The expected number T_{fail} of table accesses for an unsuccessful search and the number T_{success} for a successful search are about

$$T_{\text{fail}} \approx \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right) \text{ and } T_{\text{success}} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) ,$$

respectively. Note that these numbers become very large when n approaches m, i.e., it is not a good idea to fill a linear-probing table almost completely.

Universal hash functions were introduced by Carter and Wegman [34]. The original paper proved Theorem 4.3 and introduced the universal classes discussed in Exercise 4.11. More on universal hashing can be found in [10].

Perfect hashing was a black art until Fredman, Komlos, and Szemeredi [66] introduced the construction shown in Theorem 4.8. Dynamization is due to Dietzfelbinger et al. [54]. Cuckoo hashing [152] is an alternative approach to perfect hashing.

A *minimal perfect hash function* bijectively maps a set $S \subseteq 0..U-1$ to the range $0..n-1$, where $n = |S|$. The art is to do this in constant time and with very little space – $\Omega(n)$ bits is a lower bound. There are now practicable schemes that achieve this bound [29]. One variant assumes three truly random hash functions[8] $h_i : 0..U-1 \to im/3..(i+1)m/3-1$ for $i \in 0..2$ and $m \approx 1.23n$. In a first *mapping step*, a key $k \in 0..U-1$ is mapped to

$$p(k) = h_i(k), \text{ where } i = g(h_0(k)) \oplus g(h_1(k)) \oplus g(h_2(k)) \bmod 3 ,$$

and $g : 0..\alpha n \to \{0,1,2\}$ is a lookup table that is precomputed using some simple greedy algorithm. In a second *ranking step*, the set $0..\alpha n$ is mapped to $0..n-1$, i.e., $h(k) = \mathit{rank}(p(k))$, where $\mathit{rank}(i) = |\{k \in S : p(k) \le i\}|$. This ranking problem is a standard problem in the field of *succinct data structures* and can be supported in constant time using $O(n)$ bits of space.

Universal hashing bounds the probability of any two keys colliding. A more general notion is k-way independence, where k is a positive integer. A family H of hash functions is *k-way independent* if for some constant c, any k distinct keys x_1 to x_k, and any k hash values a_1 to a_k, $\text{prob}(h(x_1) = a_1 \wedge \cdots \wedge h(x_k) = a_k) \le c/m^k$. The polynomials of degree $k-1$ with random coefficients are a simple k-wise independent family of hash functions [34] (see Exercise 4.12).

[8] Actually implementing such hash functions would require $\Omega(n \log n)$ bits. However, this problem can be circumvented by first splitting S into many small *buckets*. We can then use the same set of fully random hash functions for all the buckets [55].

Cryptographic hash functions need stronger properties than what we need for hash tables. Roughly, for a value x, it should be difficult to come up with a value x' such that $h(x') = h(x)$.