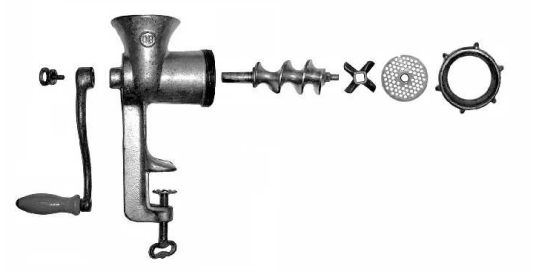
# 4

# Hash Tables and Associative Arrays



*If you want to get a book from the central library of the Karlsruhe Institute of Technology (KIT), 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 register. The University of Karlsruhe, the predecessor of KIT, was founded in 1825. Therefore, the students who 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

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

of the participating relations in an associative array. Scripting languages such as `AWK` [8] and `Perl` [325] 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 in $S$ have distinct keys. Frequently, elements are key-value pairs, i.e., $Element = Key \times Value$. Associative arrays support the following operations:

- $S.build(\{e_1,\ldots,e_n\})$: $S := \{e_1,\ldots,e_n\}$.
- $S.insert(e : Element)$: If $S$ contains no element with key $key(e)$, $S := S \cup \{e\}$. Otherwise, nothing is done.[2]
- $S.remove(x : Key)$: if there is an $e \in S$ with $key(e) = x$: $S := S \setminus \{e\}$.
- $S.find(x : Key)$: if there is an $e \in S$ with $key(e) = x$, return $e$, otherwise return $\perp$.

If only operations *build* and *find* are used, the data structure is called *static*. Otherwise, it is called a *dynamic* data structure. The operation $build(\{e_1,\ldots,e_n\})$ requires that the keys of the elements $e_1$ to $e_n$ are pairwise distinct. If operation $find(x)$ returns a reference to an element it can also be *updated* subsequently by replacing the value associated with key $x$. The operations *find*, *insert*, and *remove* essentially correspond to reading from or writing to an array at an arbitrary position (random access). This explains the name "associative array".

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 defer its discussion to the exercises.

The set *Key* is the set of potential array indices, whereas the set $\{key(s) : e \in S\}$ comprises 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 $O(n)$.

On a parallel machine, we also need atomic operations for finding and updating a hash table element. Equally useful can be an operation *insertOrUpdate* that inserts an element if it is not yet present and updates it otherwise. See Sect. 4.6 for more details. We can also consider bulk operations – inserting, removing and updating many elements in a batched fashion.

In the library example, *Key* is the set of all library card numbers, and the elements are 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*

[2] An alternative implementation replaces the old element with key $key(e)$ by $e$.

*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[3] 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*. In each table entry, we store a *set* of elements 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 good (probabilistic) performance guarantees. Our simple examples already show that finding good hash functions is nontrivial. For example, if we were to 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 $t[h(e)]$ 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*. After comparing the two approaches in Sect. 4.4, we turn to parallel hashing in Sect. 4.6. The main issue here is to avoid or mitigate the effects of multiple PEs trying to access the same entry of the hash table.

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. A good hash function should distribute every subset of the key space about evenly over the hash table. Hash tables are frequently used in time-critical parts of computer programs.

[3] 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.

**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?

**Exercise 4.5 ((database) hash join).** Consider two relations $R \subseteq A \times B$ and $Q \subseteq B \times C$ with $A \neq C$. The (natural) *join* of $R$ and $Q$ is

$$R \bowtie Q := \{(a,b,c) \subseteq A \times B \times C : (a,b) \in R \wedge (b,c) \in Q\}.$$

Give an algorithm for computing $R \bowtie Q$ in expected time $O(|R| + |Q| + |R \bowtie Q|)$ assuming that elements of $B$ can be hashed in constant time. Hint: The hash table entries may have to store sets of elements.

## 4.1 Hashing with Chaining

Hashing with chaining maintains an array $t$ of linear lists (see Fig. 4.1); the linear list $t[k]$ contains all elements $e \in S$ with $key(e) = k$. The associative-array operations are easy to implement. To find an element with key $k$, we scan through $t[h(k)]$. If an element $e$ with $key(e) = k$ is encountered, we return it. Otherwise, we return $\perp$. To remove an element with key $k$, we scan through $t[h(k)]$. If an element $e$ with $key(e) = k$ is encountered, we remove it and return. To insert an element $e$, we also scan through the sequence $t[h(k)]$. If an element $e'$ with $key(e') = k$ is encountered, we do nothing, otherwise, we add $e$ to the sequence. The operation $build(\{e_1, \ldots, e_n\})$ is realized by $n$ insert operations. Since the precondition of the operation guarantees that the elements have distinct keys, there is no need to check whether there is already an element with the same key and hence every element $e$ can be inserted at the beginning of the list $t[h(e)]$. Therefore, the running time is $O(n)$.

The space consumption of the data structure is $O(n+m)$. To remove, find or insert an element with 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 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

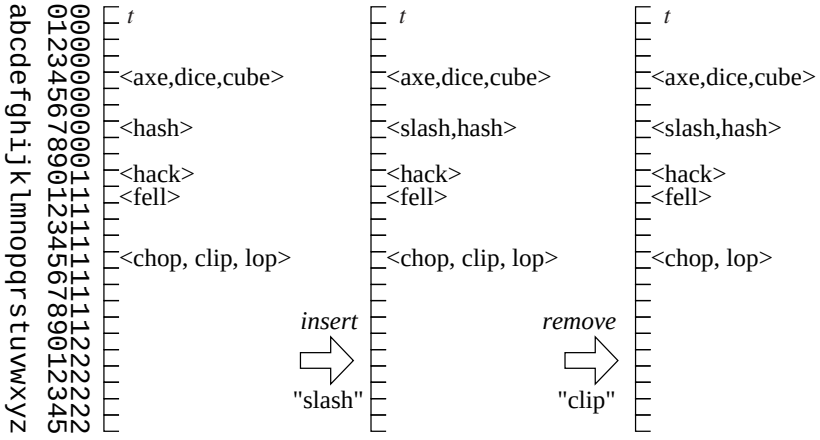



**Fig. 4.1.** Hashing with chaining. We have a table $t$ of sequences. This figure shows an example where a set of words (short synonyms of "hash") is stored using a hash function that maps the last character to the integers 0..25. We see that this hash function is not very good.

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 linear search. We shall study three approaches to dealing with the worst-case. The first approach is average-case analysis, where we average either over all possible hash functions (Theorem 4.1) or over the possible inputs (Exercise 4.8). The second approach is to use randomization, and to choose the hash function at random from a collection of hash functions. This is equivalent to average case analysis where we average over the possible 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[4] from $H$ and shall show that for any fixed set $S$ of $n$ keys, the expected execution time of *insert*, *remove*, and *find* is $O(1+n/m)$. Why do we prove a theorem based on unrealistic assumptions? It shows us what might be possible. We shall later obtain the same time bounds with realistic assumptions.

**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 insert, remove, and find is* $O(1+n/m)$.

*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 ex-

[4] 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 $n$.

ecution time of *insert*, *remove*, and *find* for a fixed key $k$. These operations need constant time for evaluating the hash function and accessing the list $t[h(k)]$ 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, $X_e = [h(e) = h(k)]$. There are two cases. If there is no entry in $S$ with key $k$, then $X = \sum_{e \in S} X_e$. If there is an entry $e_0$ in $S$ with $key(e_0) = k$, then $X = 1 + \sum_{e \in S \setminus \{e_0\}} X_e$. Using the linearity of expectations, we obtain in the first case

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

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 $E[X] = n/m$. In the second case (key $k$ occurs in $S$), we obtain analogously $X \leq 1 + (n-1)/m \leq 1 + n/m$. Thus, the expected execution time of *insert*, *find*, and *remove* is $O(1 + n/m)$. □

We can achieve a linear space requirement and a constant expected execution time for all three operations by guaranteeing that $m = \Theta(n)$ at all times. Adaptive reallocation, as described for unbounded arrays in Sect. 3.4, is the appropriate technique.

**Exercise 4.6 (unbounded hash tables).** Explain how to guarantee $m = \Theta(n)$ in hashing with chaining. You may assume the existence of a hash function $h' : Key \to \mathbb{N}$. Set $h(k) = h'(k) \bmod m$ and use adaptive reallocation.

**Exercise 4.7 (waste of space).** In part, 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 4.8 (average-case behavior).** Assume that the hash function distributes the set of potential keys evenly over the table, i.e., for each $i$, $0 \leq i \leq m-1$, we have $|\{k \in Key : h(k) = i\}| \leq \lceil N/m \rceil$. 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[5] 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

[5] In the context of hashing, one usually talks about a *class* or a *family* of functions and reserves the word "set" for the set of keys or elements stored in the table.

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.

**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 formulated so as to guarantee that the proof of Theorem 4.1 continues to work.

**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 insert, remove and find is $O(1+cn/m)$.*

*Proof.* We can reuse the proof of Theorem 4.1 almost literally. Observe that we have changed the probability space. We are now choosing the hash function $h$ from a $c$-universal class. Nevertheless, the wording of the argument does not basically change. Consider the execution time of *insert*, *remove*, or *find* for a fixed key $k$. They 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, $X_e = [h(e) = h(k)]$. There are two cases. If there is no entry in $S$ with key $k$, then $X = \sum_{e \in S} X_e$. If there is an entry $e_0$ in $S$ with $key(e_0) = k$, then $X = 1 + \sum_{e \in S \setminus \{e_0\}} X_e$. Using the linearity of expectations, we obtain in the first case

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

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

It now 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 $m$ forms a field.[6] 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 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 (over $\mathbb{Z}$) of $\mathbf{a}$ and $\mathbf{x}$. Then

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

This is the scalar product of $a$ and $\mathbf{x}$ over $\mathbb{Z}_m$. 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.** *Let $m$ be a prime. Then*

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

*is a 1-universal family of hash functions.*

In other words, the scalar product of the representation of a key as a tuple of numbers in $\{0..m-1\}$ and a random vector modulo $m$ defines a good hash function if the product is computed modulo a prime number.

*Proof.* Consider two distinct keys $\mathbf{x} = (x_1,\ldots,x_k)$ and $\mathbf{y} = (y_1,\ldots,y_k)$. To determine $\text{prob}(h_\mathbf{a}(\mathbf{x}) = h_\mathbf{a}(\mathbf{y}))$, we count the number of choices for $\mathbf{a}$ such that $h_\mathbf{a}(\mathbf{x}) = h_\mathbf{a}(\mathbf{y})$. Choose an index $j$ such that $x_j \neq y_j$. Then $(x_j - y_j) \not\equiv 0 \pmod m$, and hence any equation of the form $a_j(x_j - y_j) \equiv b \pmod m$, where $b \in \mathbb{Z}_m$, has a unique solution for $a_j$, namely $a_j \equiv (x_j - y_j)^{-1} b \pmod m$. Here $(x_j - y_j)^{-1}$ denotes the *multiplicative inverse*[7] 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,

[6] 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.

[7] In a field, any element $z \neq 0$ has a unique multiplicative inverse, i.e., there is a unique element $z^{-1}$ such that $z^{-1}\cdot z = 1$. For example, in $\mathbb{Z}_7$, we have $1^{-1} = 1$, $2^{-1} = 4$, $3^{-1} = 5$, $4^{-1} = 2$, and $5^{-1} = 3$. Multiplicative inverses allow one to solve linear equations of the form $zx = b$, where $z \neq 0$. The solution is $x = z^{-1}b$.

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

There are $m^{k-1}$ ways to choose the $a_i$ with $i \ne j$, 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

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

Is it a serious restriction that table sizes need to be prime? At first glance, yes. We certainly cannot burden users with the task of providing appropriate primes. Also, when we grow or shrink an array adaptively, it is not clear how to find a prime in the vicinity of the desired new table size. A closer look, however, shows that the problem is easy to resolve.

Number theory tells us that primes are abundant. More precisely, it is an easy consequence of the prime number theorem [138, p. 264] that for every fixed $\alpha > 1$ and every sufficiently large $m$, the interval $[m, \alpha m]$ contains about $(\alpha - 1)m/\ln m$ prime numbers. The easiest solution is then to precompute a table which contains, for example, for each interval $2^\ell..2^{\ell+1} - 1$, a prime number in this interval. Such tables are also available on the internet.

If one does not want to use a table, the required prime numbers can also be computed on the fly. We make use of the following statement ($A_k$), where $k \ge 1$ is an integer:

$$\text{The interval } k^3..(k+1)^3 \text{ contains at least one prime.} \qquad (A_k)$$

It is known that ($A_k$) holds for $k \le 8 \cdot 10^7$ and for $k > e^{e^{15}}$. For "small" $k$, the statement was established by computation; we shall tell you more about this computation in Sect. 4.8. For "large" $k$, the statement was established by mathematical proof [69]. For "intermediate" $k$, the question of whether ($A_k$) holds is open. Fortunately, the "small $k$" result suffices for our purposes. If we want to use a hash table of size approximately $m$, we determine a $k$ with $k^3 \le m \le (k+1)^3$ and then search for a prime in the interval $k^3..(k+1)^3$. The search is guaranteed to succeed for $m \le 64 \cdot 10^{21}$; it may succeed also for larger $m$.

How does this search work? We use a variant of the "sieve of Eratosthenes" (cf. Exercise 2.5). Any nonprime number 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 2 to $\lfloor (k+1)^{3/2} \rfloor$ and, for each such $j$, remove its multiples in $k^3..(k+1)^3$. For each fixed $j$, this takes time $((k+1)^3 - k^3)/j = O(k^2/j)$. The total time required is

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

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.13).

**Exercise 4.9 (strings as keys).** Adapt the class $H^{\cdot}$ to strings of arbitrary length. 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: Use "lazy evaluation" for choosing the random vector $\mathbf{a}$, i.e., fix only the components that have already been in use and extend if necessary. You may assume at first that strings do not start with the character 0 (whose byte representation consists of eight 0's); note that the strings $x$ and $0x$ are different but have the same hash value for every function $h_{\mathbf{a}}$. Once you have solved this restricted case, show how to remove the restriction.

**Exercise 4.10 (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

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

For $M = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$ and $x = (1,0,0,1)^T$, we have $Mx \bmod 2 = (0,1)^T$. This represents the number $0\cdot 2^1 + 1\cdot 2^0 = 1$. Note that multiplication modulo two is the logical AND operation, and that addition modulo two is the XOR 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 1 if the number of ones in $y$ is odd, and 0 otherwise. Hint: Multiply each row of $M$ by $x$.
(c) We now want to show that $H^{\text{lin}}$ 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 4.11 (more on hashing using matrix multiplication).** Let $p$ be a prime number and assume that *Key* is the set of $k$-tuples with elements in $0..p-1$. Let $w \geq 1$ be an integer. Generalize the class $H^{\text{lin}}$ to a class of hash functions

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

that map keys to $(0..p-1)^w$. The matrix multiplication is now performed modulo $p$. Show that $H^{\times}$ is 1-universal. Explain how $H^{\cdot}$ is a special case of $H^{\times}$.

**Exercise 4.12 (simple linear hash functions).** Assume that $Key \subseteq 0..p-1 = \mathbb{Z}_p$ for some prime number $p$. Assume also that $m \leq p$, where $m$ is the table size. For $a, b \in \mathbb{Z}_p$, let $h_{(a,b)}(x) = ((ax+b) \bmod p) \bmod m$. 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 4.13 (continuation of Exercise 4.12).** Show that the following holds for the class $H^{*}$ defined in the previous exercise. If $x$ and $y$ are distinct keys, $i$ and $j$ in $0..m-1$ are arbitrary, and $h_{(a,b)}$ is chosen randomly in $H^{*}$ then

$$\text{prob}(h_{(a,b)}(x) = i \text{ and } h_{(a,b)}(y) = j) \leq c/m^2$$

for some constant $c$.

**Exercise 4.14 (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 $H^{\text{fool}}$ map $x$ and $y$ to the same value. Hint: Let $S = \{0, m, 2m, \ldots, \lfloor p/m \rfloor m\}$.

**Exercise 4.15 (table size $2^{\ell}$).** 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. Note that the binary representation of $ax$ consists of at most $2k$ bits. The hash function select the first $\ell$ bits of the last $k$ bits. Hint: See [93].

**Exercise 4.16 (tabulation hashing, [334]).** 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 methods are categorized as being either open or closed. Hashing with chaining is categorized as an *open* hashing approach as it uses space outside the hash table to store elements. In contrast, *closed* hashing schemes store all elements in the table, but not necessarily at the table position given by the hash value. Closed schemes have no need for secondary data structures such as linked lists; this comes at the expense of more complex insertion and deletion algorithms. Closed hashing schemes are also known under the name *open addressing*, the adjective "open" indicating that elements are not necessarily stored at their hash value. Similarly, hashing with chaining is also referred to as *closed addressing*, the adjective "closed" indicating that elements are stored at their hash value. This terminology is confusing, but standard. Many ways of organizing closed hashing have been investigated [251]; see also [131, Ch. 3.3]. We shall explore only the simplest scheme. It goes under the name of *hashing with linear probing* and is based on the following principles. 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. This invariant is maintained by the implementations of the dictionary operations.

The implementations of *insert* and *find* are trivial. To insert an element $e$, we scan the table linearly starting at $t[h(e)]$, until either an entry storing an element $e'$ with $key(e') = key(e)$ or a free entry is found. In the former case, we do nothing, in the latter case, we store $e$ in the free entry. Figure 4.2 gives an example. Similarly, to find an element $e$, we scan the table, starting at $t[h(e)]$, until that 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 discuss two solutions. A simple fix is to allocate $m'$ additional 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 4.17 and Sect. 3.6. This alternative is more robust but slightly slower.

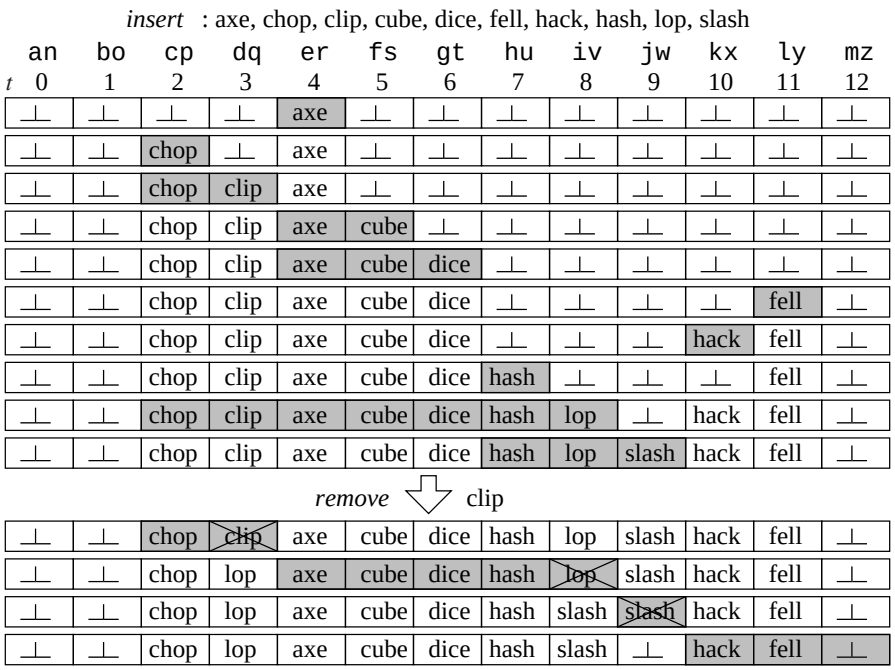



**Fig. 4.2.** Hashing with linear probing. We have a table $t$ with 13 entries storing synonyms of "(to) hash". The hash function maps the last character of the word to the integers 0..12 as indicated above the table: a and n are mapped to 0, b and o are mapped to 1, and so on. First, the words are inserted 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.

The implementation of *remove* is nontrivial. Simply overwriting the element with $\perp$ does not suffice, as it may destroy the invariant. The following example illustrates this point. Assume that $h(x) = h(z)$ and $h(y) = h(x) + 1$. Now, $x$, $y$, and $z$ are inserted in that order. Then $z$ is stored at position $h(x) + 2$. Assume that we next want to delete $y$. Simply overwriting $y$ with $\perp$ is not a solution as it 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 cleanup of the table. A third and much better approach is to 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 of $i$ to check for violations of the invariant. We set $j$ to $i+1$. If $t[j] = \perp$, we are finished. Otherwise, let $f$ be the element stored in $t[j]$. If $h(f) > i$, there is no hole between $h(f)$ and $j$ and we increment $j$. If $h(f) \leq i$, leaving the hole would violate the invariant for $f$, and $f$ would not be found anymore. We

therefore move $f$ to $t[i]$ and write $\perp$ 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.

The analysis of linear probing is beyond the scope of this book. We only mention that we need stronger properties of the hash function than guaranteed by universal hash functions. See also Sect. 4.8.

**Exercise 4.17 (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 $between(i,j,k)$, where $i,j,k \in 1..m-1$, that is true if and only if $i$ is cyclically strictly between $j$ and $k$. (3) Reformulate the invariant using *between*. (4) Adapt *remove*. (5) Can one allow the table to become completely full, i.e., store $m$ elements? Consider a search for an element that is not in the table.

**Exercise 4.18 (unbounded linear probing).** Implement unbounded hash tables using linear probing and universal hash functions. Pick a new hash function whenever the table is reallocated. Let $\alpha$, $\beta$, and $\gamma$ denote constants with $1 < \gamma < \beta < \alpha$ 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 4.17, set $m' = \delta m$ for some $\delta < 1$ and choose a new hash function (without changing $m$ and $m'$) whenever the right-hand end of the table overflows.

## 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. 4.7 made it competitive with linear probing. Chaining is much slower if the implementation is sloppy or memory management is not implemented well.

In Theorem 4.3, we showed that the combination of hashing with chaining and $c$-universal classes of hash functions guarantees good expected behavior. A similar result does *not* hold for the combination of hashing with linear probing and $c$-universal hash functions. For a guarantee of expected constant search time, linear probing needs hash families with stronger randomness properties or the assumption of full randomness. We come back to this point in Sect. 4.8.

## 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. For simplicity, we identify elements and their keys, i.e., $S = \{x_1, \ldots, x_n\}$ is the set of keys occuring.

In this section, we use $H_m$ to denote a family of $c$-universal hash functions with range $0..m-1$. In Exercise 4.12, it was 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 (ordered) pairs of distinct keys in $S$ which are mapped to the same position:

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

If $h$ is chosen randomly in $H_m$, $C$ is a random variable. As a first step, we derive a bound on the expectation of $C$.

**Lemma 4.5.** *$\mathrm{E}[C] \leq cn(n-1)/m$. 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 $X_{ij}(h) = 1$ if $h(x_i) = h(x_j)$ and $X_{ij} = 0$ otherwise. Then $C(h) = \sum_{ij} X_{ij}(h)$, and hence

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

where the second equality follows from the linearity of expectations (see (A.3)) and the inequality follows from the universality of $H_m$. The second claim follows from Markov's inequality (A.5). □

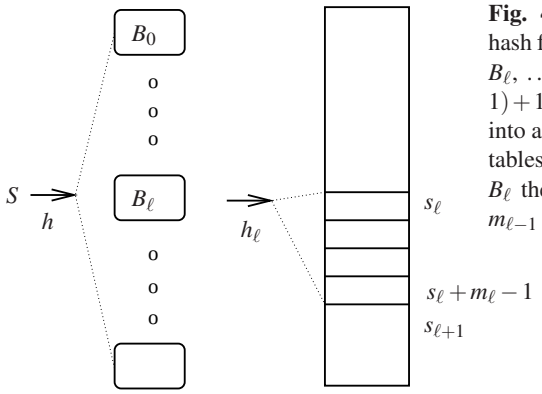



**Fig. 4.3.** Perfect hashing. The top-level hash function $h$ splits $S$ into subsets $B_0, \ldots, B_\ell, \ldots$. Let $b_\ell = |B_\ell|$ and $m_\ell = cb_\ell(b_\ell - 1) + 1$. The function $h_\ell$ maps $B_\ell$ injectively into a table of size $m_\ell$. We arrange the subtables into a single table.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

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 at least half of the functions in $H_m$. Since $C(h)$ is even (recall that it counts ordered pairs), $C(h) < 2$ implies $C(h) = 0$, and so $h$ operates injectively on $S$. □

So we fix a $m$ with $m \geq cn(n-1)+1$, choose a random $h \in H_m$, and check whether or not it is injective on $S$. If not, we iterate until we have found an injective $h$. After an average of two trials, we shall be 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 such that the sum of the squared bucket sizes is linear in $n$. The second stage uses an amount of space for each bucket that is quadratic in the number of elements contained in the bucket. For $\ell \in 0..m-1$ and $h \in H_m$, let $B_\ell^h$ be the elements in $S$ that are mapped to $\ell$ by $h$ and let $b_\ell^h$ be the cardinality of $B_\ell^h$.

**Lemma 4.7.** *For every $h \in H_m$, $C(h) = \sum_\ell b_\ell^h(b_\ell^h - 1)$.*

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

We are now ready for the construction of a perfect hash function. Let $\alpha$ 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 $B_\ell$. 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 $\ell$, let $B_\ell$ be the elements in $S$ mapped to $\ell$ and let $b_\ell = |B_\ell|$.

Now consider any $B_\ell$. Let $m_\ell = cb_\ell(b_\ell - 1) + 1$. We choose a function $h_\ell \in H_{m_\ell}$ which maps $B_\ell$ injectively into $0..m_\ell - 1$. At least half of the functions in $H_{m_\ell}$ have this property, by Lemma 4.6 applied to $B_\ell$. In other words, $h_\ell$ maps $B_\ell$ injectively into a table of size $m_\ell$. 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 $B_\ell$ 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 $S$. The values of this function are bounded by

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

and hence we have constructed a perfect hash function that maps $S$ into a linearly sized range, namely $0..\lfloor(\alpha + 2c^2/\alpha)n\rfloor$. In the derivation above, the first inequality uses the definition of the $m_\ell$'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 $2\sqrt{2}n$. Besides the table, we need space for storing the representation of the hash function. This space is essentially determined by the space needed for storing the parameters of the functions $h_\ell$ and the starting value $s_\ell$ of the $\ell$th subtable, $\ell \in 0..\lceil \alpha n \rceil - 1$. We need to store three numbers for each $\ell$ and hence the space needed for the representation of the function is linear. The expected time for finding the function $h$ is $O(n)$ and the expected time for finding $h_\ell$ is $O(b_\ell)$. Thus the total construction time is linear.

**Theorem 4.8.** *For any set of $n$ keys, a perfect hash function with range $0..\lfloor 2\sqrt{2}n\rfloor$ can be constructed in linear expected time. The space needed to store the function is linear.*

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

**Exercise 4.19 (dynamization).** We outline a scheme for "dynamization" of perfect hashing, i.e., a method that supports insertions and deletions and guarantees constant access time. Consider a fixed $S$ of size $n$ and choose $h \in H_m$, where $m = 2\lceil \alpha n \rceil$. For each $\ell$, let $m_\ell = \lceil 2cb_\ell(b_\ell - 1)\rceil$, i.e., all $m_\ell$'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 $\ell$. Increment $b_\ell$. If $h_\ell$ is not injective on $B_\ell \cup \{x\}$ and $m_\ell \ge \lceil cb_\ell(b_\ell - 1)\rceil$, we choose a new $h_\ell$. Repeat until the hash function is injective. Once $m_\ell < \lceil cb_\ell(b_\ell - 1)\rceil$, we allocate a new table for $B_\ell$ of size $m_\ell = \lceil 2cb_\ell(b_\ell - 1)\rceil$. We also keep track of $n = |S|$ and $C(h)$. Once $n$ exceeds $m/\alpha$, we set $m = 2\lceil \alpha n \rceil$, choose a new function $h$ for the first level, and move $S$ to a new table of size $m$. If $n \le m/\alpha$ but $C(h)$ exceeds $2cn/\alpha$, we keep $m$ fixed and choose a new first-level function $h$. We move $S$ to a new table of size $m$. Work out the details of the algorithm and of the analysis. Hint: See [94].

## 4.6 Parallel Hashing

A shared hash table is a powerful way for multiple PEs to share information in a fine-grained way. However, a shared hash table raises a number of nontrivial issues with respect to correctness, debugging, and performance. What happens when several PEs want to access the same element or the same position in the table? Let us first define what we want to happen, i.e., the programming interface of a shared hash table.

The operation *build* has the same meaning as before, except that each PE might contribute to the initial table content.

*Insertions* are as before. When several PEs attempt to insert an element with the same key concurrently, only one of those elements will be inserted. The application program should not make any assumptions about which of these concurrent operations succeeds. Similarly, when several PEs attempt to *remove* an element, it will be removed only once.

The operation $find(x)$ is almost as before. It returns $\perp$ if $x$ is not part of the table. Otherwise, rather than a reference, it should return a *copy* of the value currently stored with key $x$.[8] On a shared-memory machine, many concurrent read accesses to the same element should be possible without performance penalty.

*Concurrent updates* have to be performed atomically in order to avoid chaos. In many situations, the value written depends on the previously stored value $v'$. We therefore encapsulate this behavior in an atomic update operation $update(x, v, f)$ that is passed not only a value $v$ but also a function $f : Value \times Value \to Value$. This update function stores the value $f(v', v)$. For example, in order to increment a counter associated with key $x$, one could call $update(x, 1, +)$. Sometimes we also need a combined operation *insertOrUpdate*. If an element with the given key $x$ is not in the table yet, then the key value pair $(x, v)$ is inserted. Otherwise, if $(x, v')$ is already in the table, then $v'$ is replaced by $f(v', v)$.

An important alternative to a shared hash table is the use of local hash tables. Additional code will then be needed to coordinate the PEs. This is often more efficient than a shared hash table, and may be easier to debug because the PEs work independently most of the time.

Let us consider both approaches for a concrete example. Suppose we have a multiset $M$ of objects and want to count how often each element of $M$ occurs. The following sequential pseudocode builds a hash table $T$ that contains the counts of all elements of $M$:

```
Class Entry(key : Element, val : ℕ)
T : HashTable of Entry
forall m ∈ M do e := T.find(m);   if e = ⊥ then T.insert((m, 1)) else e.val++
```

A simple parallelization in shared memory is to make $T$ a shared hash table and to make the loop a parallel loop. The only complication is that the sequential code

[8] Returning a reference – with the implication that the element could be updated using this reference – would put the responsibility of ensuring consistent updates on the user. This would be a source of hard to trace errors.

contains a *find* operation, a conditional insertion, and a possible write access. Our concurrent operation *insertOrUpdate* nicely covers what we actually want. We can simply write

**forall** $m \in M$ **do**$\| \; T.insertOrUpdate(m, 1, +)$

When several PEs call this operation concurrently with the same key, this will be translated into several atomic increment operations (see also Sect. 2.4.1). If key $m$ was not present before, one of the concurrent calls will initialize the counter to 1 and the others will perform atomic increment operations.

Having avoided possible correctness pitfalls, there is still a potential performance problem. Suppose most elements of $M$ in our example have the same key. Then the above parallel loop would perform a huge number of atomic increment operations on a single table entry. As already discussed in Sect. 2.4.1, this leads to high contention and consequently low performance.

We next discuss the use of local hash tables. In our example, each element of $M$ will then be counted in some local table. At the end, we need additional code to merge the local tables into a global table. If the number of elements in $M$ is much larger than $p$ times the number of different keys occuring, then this merging operation will have negligible cost and we shall get good parallel performance. Note that this covers, in particular, the high-contention case, where the shared table was bad. A potential performance problem arises when the number of different keys is large. Not only will merging then be expensive, but the total memory footprint of the parallel program may also be up to $p$ times larger than that of the sequential program. Thus, it might happen that the sequential program runs in cache while the parallel program makes many random accesses to main memory. In such a situation, just filling the local tables may take longer than solving the overall problem sequentially. There are more sophisticated algorithms for such *aggregation problems* that interpolate between hashing-based and sorting-based algorithms (e.g., [233]). We refer our readers to the literature for a detailed discussion.

### 4.6.1 Distributed-Memory Hashing

We start with the obvious way of realizing a distributed hash table. Each processor is made responsible for a fraction $1/p$ of the table and handles all requests to this part of the table. More precisely, we distribute the hash table $t[0..m-1]$ to $p$ PEs by assigning the part $t[i \cdot m/p..(i+1) \cdot m/p]$ to PE $i \in 0..p-1$. The operation $find(x)$ is translated into a find request message to PE $h(x) \operatorname{div} m/p$. Each PE provides a hash table server thread which processes requests affecting its part. The operations *insert*, *remove*, and *update* are handled analogously. This approach works for any representation of the local part, for example hashing with chaining or hashing with linear probing. In the latter case, wraparound for collision resolution should be local, i.e., within the part itself, to avoid unnecessary communication. Each PE knows the hash function and hence can evaluate it efficiently.

The distributed hash table just outlined is simple and elegant but leads to fine-grained messages. On most machines, sending messages is much more expensive

than hash table accesses to local memory. Furthermore, in the worst case, there could be considerable contention due to many accesses to the same entry of the hash table. We shall next describe a scheme that overcomes these shortcomings by using globally synchronized processing of batches of operations. It provides high performance for large batches even in the presence of contention, and also simplifies unbounded hash tables.

Each PE has a local batch $O$ containing up to $o$ operations. The batches are combined as follows. Each PE first sorts its $O$ by the ID of the PE responsible for handling an operation. This can be done in time $O(o+p)$ using bucket sort (see Sect. 5.10). On average, we expect a PE to access at most $o/p$ *distinct* locations in any other PE. In parallel computing, the maximum bucket size is important. It will be part of the analysis to show that the probability that some PE accesses more than $2o/p$ distinct locations in some other PE is small.

There is another problem that we have to deal with. If a PE accesses the same location many times, this may lead to buckets with many more than $2o/p$ operations. Buckets whose size "significantly" exceeds the expected bucket size $o/p$ might contain many operations addressing the same key.[9] Therefore, after bucket sorting, the operations in each bucket are inserted into a temporary local hash table in order to condense them: An element inserted/removed/searched several times needs only to be inserted/removed/searched once. We assume here that update operations to the same location can be combined into a single update operation. For example, several counter increment operations can be combined into a single add operation.

Using an all-to-all data exchange (see Sect. 13.6), the condensed operations are delivered to the responsible PEs. Each PE then performs the received operations on its local part. The results of the *find* operations are returned to their initiators using a further all-to-all operation. Figure 4.4 gives an example.

**Theorem 4.9.** *Assuming $o = \Omega(p \log p)$ and that the hash function behaves like a truly random hash function, a batch of hash table operations can be performed in expected time $O(T_{\text{all}\to\text{all}}(o/p))$, where $T_{\text{all}\to\text{all}}(x)$ is the execution time for a regular all-to-all data exchange with messages of size at most $x$ (see Sect. 13.6).*

*Proof.* Recall our assumption that we resolve *all* locally duplicated keys by first building local hash tables. Preparing the messages to be delivered takes time $O(o)$, as explained above. Since $T_{\text{all}\to\text{all}}(o/p) \geq p \cdot (o/p) \cdot \beta = \Omega(o)$, the $O(o)$ term is covered by the bound stated in the theorem.

Each PE sends a message to every other PE. More precisely, the message sent by PE $i$ to PE $j$ contains all operations originating from PE $i$ to elements stored in PE $j$. We have to show that the expected maximum message size is $O(o/p)$. Since duplicate keys have been resolved, at most $o$ keys originate from any processor. Since we assume a random hash function, these keys behave like balls thrown uniformly

[9] There are several ways here to define what "significantly" means. For the analysis, we assume that all common keys will be condensed. In practice, this step is often completely skipped. The truth for a robust but practically efficient solution lies somewhere in the middle. A simple threshold such as $e2 \cdot o/p$ should be a good compromise.

at random into $p$ bins. Hence, we can employ the Chernoff tail bound (A.7): The probability that a message is larger than twice its expectation $o/p$ is bounded by

$$\left(\frac{e^1}{2^2}\right)^{o/p} = \left(\frac{e}{4}\right)^{o/p}.$$

This probability is bounded by $1/p^3$ if $o \geq 3\log(4/e) \cdot p \log p = \Theta(p \log p)$. In this situation, the probability that any of the $p^2$ buckets ($p$ buckets in each PE) is larger then $2o/p$ is at most $p^2/p^3 = 1/p$. Furthermore, no bucket can be larger then $o$. Combining both bounds we obtain an upper bound of

$$\left(1 - \frac{1}{p}\right) \cdot 2\frac{o}{p} + \frac{1}{p}o \leq 2\frac{o}{p} + \frac{o}{p} = 3\frac{o}{p}$$

on the expectation of the maximum bucket size.

A similar argument can be used to bound the expected number of operations any PE has to execute locally – only with probability at most $1/p$ will any message contain more than $2o/p$ operations and, even in the worst case, the total number of received operations cannot exceed $p \cdot o$. Hence, the expected work on any PE is bounded by

$$\left(1 - \frac{1}{p}\right) \cdot 2\frac{o}{p} \cdot p + \frac{1}{p}o \cdot p \leq 2o + o = 3o.$$

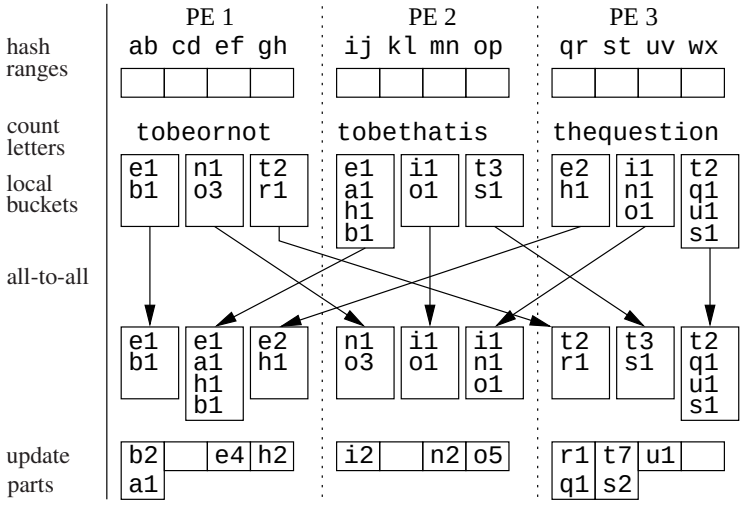



**Fig. 4.4.** Counting letters on three PEs using a distributed hash table. PE 1 is responsible for the key range `a..h`, PE 2 for the range `i..p`, and PE 3 for the range `q..r`. Each PE first processes its part of the text and prepares appropriate messages. For example, the text of PE 1 contains three occurrences of the character `o` and hence PE 1 prepares the message "`o3`" to be sent to PE 2. After the messages are delivered, each PE updates its part. The local hash tables have buckets of size 2.

To summarize, we have expected time $O(o) = O(T_{\text{all}\to\text{all}}(o/p))$ for local preprocessing and final processing and time $O(T_{\text{all}\to\text{all}}(o/p))$ for message exchange when $o \geq 3\log(4/e) \cdot p \log p$. For smaller values of $o \in \Theta(p \log p)$, the running time cannot be larger, and remains in $O(T_{\text{all}\to\text{all}}(\log p)) = O(T_{\text{all}\to\text{all}}(o/p))$. □

***Exercise 4.20.** Redo the proof of Theorem 4.9 for the case where an all-to-all data exchange with variable message sizes is used (see Sect. 13.6.3). Show that in this case time $O\big(T^*_{\text{all}\to\text{all}}(o)\big)$ is already achieved when $o = \Omega(p)$.

In sequential hashing, theorems that have been proved for random hash functions can frequently be transferred to universal hash functions. This is *not* true for parallel hashing. Theorem 4.9 does not transfer if we replace random hash functions by universal hash functions. We need stronger properties of the hash function employed in order to prove that it is unlikely that any message will become too long. The tabulation hash function in Exercise 4.16 can be shown to have the required properties [258].

To make the hash table unbounded, we determine the *maximum* number of elements $n_{\max}$ in any local table using an all-reduce operation (see Sect. 13.2). We choose the local table size based on $n_{\max}$. This way, decisions about resizing the table are made identically on all PEs.

**Exercise 4.21 (MapReduce programming model).** [86] A widely used pattern for processing large data sets is as follows: Consider the (key) sets $K$ and $K'$ and the value sets $V$ and $V'$. The input is a set of key-value pairs $P \subseteq K \times V$. There are two-user defined functions, $map : K \times V \to 2^{K' \times V'}$ and $reduce : K' \times V'^* \to K' \times V'$, where $V'^*$ denotes the set of sequences consisting of elements of $V'$. The resulting MapReduce computation first *maps* each pair $(k,v) \in P$ to a new set of key-value pairs. Consider the union $P' \subseteq K' \times V'$ of all these sets. Now, pairs in $P'$ with the same key are collected together, i.e.,

$$P'' := \left\{ (k', s) \in K' \times V'^* : s = \left\langle v' : (k', v') \in P' \right\rangle \wedge |s| > 0 \right\}.$$

Finally, the *reduce* function is applied to each element of $P''$.

Assuming that the elements of $P$ and $P''$ are distributed over the PEs of a distributed-memory parallel computer, mapping and reducing are local computations. However, the collection step requires communication. Explain how you can implement it using batched distributed hashing. Analyze its performance. Under what circumstances is a speedup $\Omega(p)$ possible?

Explain how the billing system discussed in Exercise 4.3 can be viewed as an instantiation of the MapReduce pattern.

### 4.6.2 Shared-Memory Hashing

The distributed-memory hash table from Sect. 4.6.1 is readily transformed into a shared-memory setting: The hash table is split into $p$ pieces of equal size, with one

PE responsible for all operations on one of the pieces. This approach works efficiently only for large batches of operations. Moreover, the need for synchronization reduces flexibility.

We shall next work out how a single hash table can be accessed concurrently and asynchronously. We start with fast, simple, and more specialized solutions, going to more general solutions in several steps. This approach reflects the fact that concurrent hash tables exhibit a trade-off between simplicity and efficiency on the one side and generality on the other side.[10]

*Only finds.* The most simple case is that only (read-only) *find* operations are processed asynchronously. In this case, we can simply adopt the sequential implementation of the operation *find* – parallel executions of several calls do not interfere with each other. However, before processing a batch of update/insert/remove operations, all PEs need to synchronize. In particular, while a batch is being processed, no thread can execute *find* operations.

The above data structure also allows concurrent asynchronous update operations under the following circumstances: Updates have to be performed in an atomic way, for example using a CAS instruction, and *find* operations also have to read the updateable part of an element atomically.

**Exercise 4.22.** Work out an example where a *find* operation returns an inconsistent value when an *update* operation modifies a table entry in several steps.

*Insertions.* Similarly, we can support concurrent asynchronous insertions of an element $e$ using atomic operations on table entries. We shall discuss this for hashing with linear probing (see also [303, 205]). The operation looks for a free table entry $i$ as in sequential linear probing and attempts to write $e$. This write operation has to be done atomically, for example using a CAS instruction. Without atomic writing, concurrent *find* operations could return inconsistent, partially constructed table entries. If the CAS instruction fails because another concurrent *insert* got there first, the insertion operation continues to look for a free entry starting at position $i$. Entry $i$ has to be reinspected, since another PE might have inserted an element with the same key as $e$, in which case the operation is terminated immediately. Note that reinspecting the same position cannot lead to an infinite loop, since the succeeding CAS will fill the position and our implementation never reverts a filled entry to free.

*Unbounded Hash Tables.* How about adaptive growing? In the sequential case, we simply remarked that this can be handled as with unbounded arrays – reallocation and copying of the table content. In the concurrent case the same approach works, but is quite difficult to implement correctly – we have to make sure that all PEs switch from one version of the table to the next in a consistent fashion. We are only aware of a single paper showing that this can be done efficiently [205].

[10] An interesting related paper by Shun and Blelloch [294] achieves a good compromise between simplicity and generality by requiring global synchronizations between phases with different operation mixes.

*Removals.* We can support removals by marking the corresponding table entry as deleted. This requires that *find* operations read the entire table entry atomically. In Sect. 4.3 we have already pointed out that removal by marking has considerable disadvantages over properly removing the element, since we can never free any space. Hence, for most uses of deletion, we should also have a mechanism for growing the table or cleaning it up (see [205] for details).

*Performance Analysis.* Let us now analyze the performance of the above asynchronous data structure and compare it with batched hashing (see Sect. 4.6.1). The asynchronous implementation works well as long as the application does not access particular keys very frequently. In this situation, we have low contention for memory accesses and we get constant expected execution time for all supported operations. For such workloads, the asynchronous implementation is more flexible than the distributed one and will often be faster. The situation is different when particular keys are used very often. The most problematic operations here are updates, since they actually modify the elements. This leads to massive write contention. Here, the distributed implementation has big advantages, since it resolves the contention locally. For example, when all PEs update the same element $\Theta(p \log p)$ times, this takes expected time $O(p \log p)$ in the distributed implementation by Theorem 4.9, whereas the asynchronous implementation needs time $\Omega\left(p^2 \log p\right)$. The situation is less severe for *find* operations since modern shared-memory machines accelerate concurrent reading via caching. The aCRQW-PRAM model introduced in Sect. 2.4.1 reflects this difference by predicting constant expected running time for *find* even in the case of high contention. The situation is somewhere in between for insertions and removals. Concurrent execution of $p$ operations affecting the same key is guaranteed to make only a single modification to the memory. However, if all these operations are executed at the same time, all PEs might initiate a write instruction. In the aCRQW-PRAM model, this would take time $\Theta(p)$. However, concurrent attempts to *insert* the same element multiple times will be fast if the element was originally inserted sufficiently long ago.

*Shared-Memory Hashing with Chaining.* In hashing with chaining, we can lock individual buckets to allow for full support of *insert*, *remove* and *update*, including support for complex objects that cannot be written atomically. However, even *find* operations then have to lock the bucket they want to access. What sounds like a triviality at first glance can be a severe performance bottleneck. In particular, when many *find* operations address the same table entry, we could have a lot of contention for writing the lock variable, even though we actually only want to read the value (which, by itself, is fast in the aCRQW-PRAM-model).

**Exercise 4.23 (parallel join).** Consider two relations $R \subseteq A \times B$ and $Q \subseteq (B \times C)$. Refine your algorithm obtained in Exercise 4.5 for computing

$$R \bowtie Q = \{(a,b,c) \subseteq A \times B \times C : (a,b) \in R \wedge (b,c) \in Q\}$$

to work on an aCREW PRAM. Each PE should perform only $O(|Q|/p)$ *find* operations and $O(|R|/p)$ *insertOrUpdate* operations. Now assume that each value of $B$ ap-

pears only once in $R$. Show how to achieve running time $O((|R|+|Q|+|R \bowtie Q|)/p)$. Discuss what can go wrong if this assumption is lifted.

### 4.6.3 Implementation of Shared-Memory Hashing

We now discuss an implementation of shared-memory linear probing. We aim for simplicity and efficiency rather than generality and portability. Since linear probing requires atomic operations, there is a strong dependency on the processor instruction set and also on its level of support by the compiler (see also Sects. B.4 and C.4). We give an implementation of a hash table supporting atomic *insert*, *update*, and *find* for the Linux *gcc* compiler on the x86 architecture, which supports 16-byte atomic CAS and 8-byte atomic loads. If the key and the data fit together into 8 bytes, a similar implementation would work for a wider range of architectures. We could support longer key or data fields by introducing additional indirections – the table entries would then contain pointers to the actual data. Transactional synchronization instructions (i.e., the Intel Transactional Synchronization Extensions (Intel TSX) described in Sect. B.5) can provide larger atomic transactions but often require a fallback implementation based on ordinary atomic instructions. We come back to this point at the end of this section. We assume that the key and the data of an element require 8 bytes each, that an empty table entry is indicated by a special key value (in our case the largest representable key), and that a find returns an element. If the find is unsuccessful, the key of the element returned is the special key value. Under these assumptions, we can work with 16-byte reads implemented as two atomic 8-byte reads. Consider the possible cases for the execution of a *find* operation:

- Table entry $t[i]$ is empty: The first 8-byte read copies the special key into the element to be returned. It is irrelevant what the second 8-byte read copies into the element returned; it may copy data that was written by an insert that started after the find, but completed before it. In any case, the returned element has a key indicating an empty table entry. The *find* operation returns an outdated but consistent result.
- Table entry $t[i]$ contains a nonempty element: The first 8-byte read copies a valid *key* and the second 8-byte read copies the latest value written by an *update* before the second read. A valid element is returned, because updates do not change the *key*. Recall that there are no deletions.

With this reasoning, we can use a single `movups` x86 instruction on 16-byte data that issues two 8-byte loads which are guaranteed to be atomic if the data is 8-byte aligned. On most compilers, this instruction can be generated by calling the `_mm_loadu_ps` intrinsic.

The class `MyElement` below encapsulates an element data type, including most architecture-specific issues. Here, we use 64-bit integers for both key and data. Empty table entries are encoded as the largest representable key. Other representations are possible as long as the public methods are implemented in an atomic way.

```
class MyElement
{
public:
  typedef long long int Key; //64-bit key
  typedef long long int Data; //64-bit data or a pointer
private:
  Key key;
  Data data;
  template <class T> T & asWord() { // a helper cast
    return *reinterpret_cast<T *> (this);
  }
  template <class T> const T & asWord() const {
    return *reinterpret_cast<const T *> (this);
  }
public:
  MyElement() {}
  MyElement(const Key & k, const Data & d):key(k),data(d){}
  Key getKey() const { return key; }
  static MyElement getEmptyValue() {
    return MyElement(numeric_limits<Key>::max(), 0);
  }
  bool isEmpty() const {
    return key == numeric_limits<Key>::max();
  }
  bool CAS(MyElement & expected, const MyElement & desired) {
    return __sync_bool_compare_and_swap_16(&asWord<__int128>(),
      expected.asWord<__int128>(), desired.asWord<__int128>());
  }
  MyElement(const MyElement & e) {
    asWord<__m128i>() = _mm_loadu_si128((__m128i*)&e);
  }
  MyElement & operator = (const MyElement & e) {
    asWord<__m128i>() = _mm_loadu_si128((__m128i*)&e);
    return *this;
  }
  void update(const MyElement & e) { data = e.data; }
};//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation
```

Given the class `MyElement` above, the implementation of the hash table is mostly straightforward. The constructor allocates the table array using the instruction `_aligned_malloc` available in Linux[11] in order to have table entries start at multiples of 16. This is required in order to use the 16-byte x86 CAS instructions. The hash function is taken from the C++standard library. To upper-bound the lookup time for densely populated tables, we limit the maximum length of the scan to a large enough *maxDist*. If a scan reaches this limit, the table needs to be enlarged to avoid bad performance. We implement cycling probing (Sect. 4.3) and avoid modulo operations by using only powers of two for the table capacity. If the table capacity $m$ is

[11] In Windows, one would use `memalign`.

equal to $2^\ell$, $x \bmod 2^\ell = x \wedge (2^l - 1)$, where $\wedge$ is a bitwise AND operation, which is much faster than a general modulo operation.

```
template <class Element>                                                    1
class HashTable                                                             2
{                                                                           3
  typedef typename Element::Key Key;                                        4
  size_t h(const Key & k) const { return hash(k) & mask; }                  5
  enum { maxDist = 100 };                                                   6
public:                                                                     7
  HashTable(size_t logSize = 24) : mask((1ULL << logSize) -1){              8
    t = (Element *)_aligned_malloc((mask + 1)*sizeof(Element), 16);         9
    if (t == NULL) std::bad_alloc();                                        10
    std::fill(t, t + mask + 1, Element::getEmptyValue());                   11
  }                                                                         12
  virtual ~HashTable() { if (t) _aligned_free(t); }                         13
  bool insert(const Element & e) {                                          14
    const Key k = e.getKey();                                               15
    const size_t H = h(k), end = H + maxDist;                               16
    for (size_t i = H; i < end; ++i) {                                      17
      /* copy the element guaranteeing that a concurrent update            18
         of the source will not result in an inconsistent state */          19
      Element current(t[i&mask]);                                           20
      if (current.getKey() == k) return false; // key already exists        21
      if (current.isEmpty()) { // found free space                          22
        if (t[i&mask].CAS(current, e)) return true;                         23
        // potentially collided with another insert                         24
        --i; // need to reinspect position i;                               25
      }                                                                     26
    }                                                                       27
    throw bad_alloc(); // no space found for the element                    28
    return false;                                                           29
  }                                                                         30
  Element find(const Key & k) {                                             31
    const size_t H = h(k), end = H + maxDist;                               32
    for (size_t i = H; i < end; ++i) {                                      33
      const Element e(t[i&mask]);                                           34
      if (e.isEmpty() || (e.getKey() == k)) return e;                       35
    }                                                                       36
    return Element::getEmptyValue();                                        37
  }                                                                         38
private:                                                                    39
  Element * t;                                                              40
  std::hash<Key> hash;                                                      41
  const size_t mask;                                                        42
};//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation 43
```

We implement a powerful update function *insertOrUpdate* that inserts an element $e$ if the key of $e$ is not already present in the table. Otherwise, it updates the existing

table entry using a function $f(c,e)$ of the old element $c$ and the new element $e$. This function guarantees an atomic update of the element.

```
template <class F>
bool insertOrUpdate(const Element & e, F f = F()) {
  const Key k = e.getKey();
  const size_t H = h(k), end = H + maxDist;
  for (size_t i = H; i < end; ++i) {
    Element current(t[i&mask]);
    if (current.getKey() == k) { // key already exists
      while (!t[i&mask].atomicUpdate(current, e, f)) {
        // potentially collided with another update
        current = t[i&mask]; // need to reinspect position i
      }
      return false;
    }
    if (current.isEmpty()) { // found free space
      if (t[i&mask].CAS(current, e)) return true;
        // potentially collided with another insert
        --i; // need to reinspect position i
      }
    }
  }
  throw bad_alloc(); // no space found for the element
  return false;
}
template <class F>
bool MyElement::atomicUpdate(MyElement & expected,
            const MyElement & desired, F f) {
  return __sync_bool_compare_and_swap(&data,
      expected.data, f(expected, desired).data);
}//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation
```

The update function object $f$ can be also specified using the C++11 lambda notation. The software design allows us to take advantage of specialized update function objects for insert-or-increment and insert-or-decrement that use more efficient fetch-and-increment/fetch-and-decrement processor instructions instead of general CAS instructions. Additionally, overwriting the data for existing keys can be specialized without using the heavy CAS instruction.

```
struct Overwrite {};
struct Increment {};
struct Decrement {};
// atomic update by overwriting
bool atomicUpdate(MyElement & expected,
    const MyElement & desired, Overwrite f) {
  update(desired);
  return true;
}
```

```
// atomic update by increment
bool atomicUpdate(MyElement & expected,
    const MyElement & desired, Increment f) {
  __sync_fetch_and_add(&data, 1);
  return true;
}
// atomic update by decrement
bool atomicUpdate(MyElement & expected,
    const MyElement & desired, Decrement f) {
  __sync_fetch_and_sub(&data, 1);
  return true;
}//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation
```

We finally give an implementation of *insertOrUpdate* that wraps a simpler sequential code into a memory (Intel TSX) transaction (see Sect. B.5). If the transaction fails, the version using an atomic CAS operation is used as a fallback.

```
bool insertOrUpdateTSX(const Element & e, F f = F()) {
  if(_xbegin() == _XBEGIN_STARTED) // successful transaction start
  {
    const Key k = e.getKey();
    const size_t H = h(k), end = H + maxDist;
    for (size_t i = H; i < end; ++i) {
      Element & current = t[i&mask];
      if (current.getKey() == k) { //key already exists
        current.update(e,f);
        _xend();
        return true;
      }
      if (current.isEmpty()) { //found free space
        current = e;
        _xend();
        return true;
      }
    }_xend();
    //no space found for the element, use a table with a larger capacity
    throw bad_alloc();
    }
  // transaction fall-back using CAS
  return insertOrUpdate(e,f);
}//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation
```

### 4.6.4 Experiments

We conducted a series of experiments to evaluate the scalability of the implementation using the machine described in Appendix B. We compared our implementation with two hash table implementations from the Intel Threading Building Blocks (TBB) Library, both based on chaining with buckets. We used a table with a capacity of $2^{28}$ elements which was initially populated with $2^{26}$ elements.

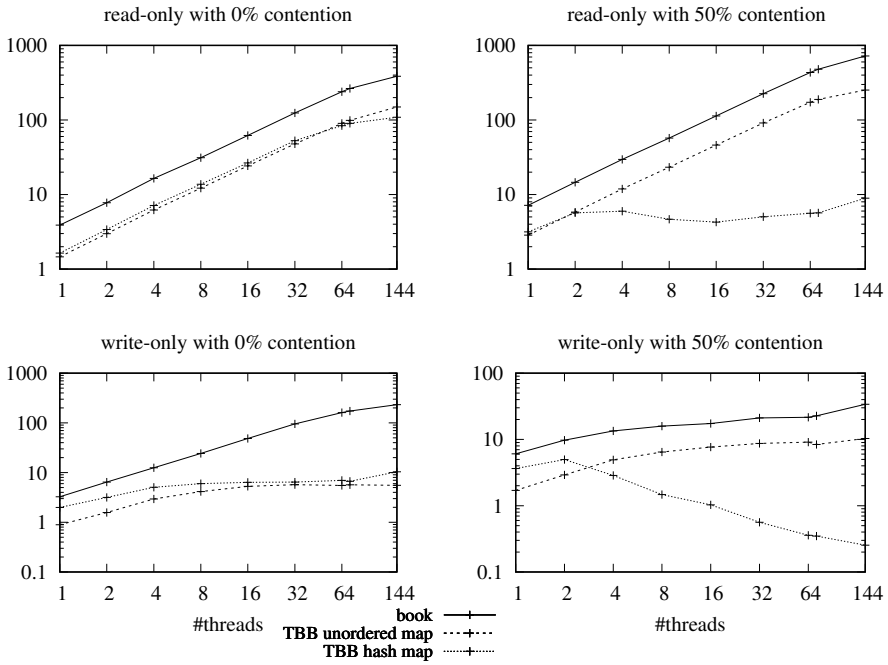



**Fig. 4.5.** Throughput (million operations/second) of concurrent hash table implementations. The horizontal axis indicates the number of threads, and the vertical axis indicates the throughput. Higher is better.

To demonstrate the effects of contention, we conducted the following simple experiment. The measurement performed $2^{28}$ queries with $c\%$ operations using a particular "hot" key value and $(100-c)\%$ operations using a random key. The operations were distributed over $p$ threads. Figure 4.5 shows the resulting performance for a read-only workload (*find* operations) and write-only workload (increment operations). Our implementation used *insertOrUpdate* specialized to a fetch-and-add operation. The Intel TBB hash map returns an accessor object as a result of the insert operation, which locks the existing element. We performed the increment operation under this lock. For the *find* operation on the TBB hash map, we used the *const* accessor, which implies a read lock. The TBB unordered map returns iterators (pointers) to the elements in the hash table without any implicit locks. To implement the update function, we had to use an atomic fetch-and-add instruction on the data referenced by the returned iterator.

Without contention, the read-only workload is easy for all three implementations. However, the book implementation is more than twice as fast as the TBB implementations. Since hyper-threading hides memory latencies, we observed a significant speedup when using 144 threads on the 72-core machine.

When there is read contention, the TBB hash maps do not scale well, because of the contention on bucket and element read locks (the locks have to perform atomic

writes to change their internal state). The book implementation, in contrast, profits from read contention, since it does not modify any locks or data and the hot element can be held in the L1-cache.

For write-only workloads without contention, the book implementation demonstrates very good scaling, whereas the TBB implementations show only a relative speedup of 6 using all cores. Both TBB implementations suffer from contended atomic operations in their internal structures.

With write contention, the scaling reduces, being dependent on the speed of the contended atomic increment. We also tried to use generic 8-byte CAS on the data part of the element instead of an atomic fetch-and-add instruction. This resulted in worse scaling. This experiment confirms that the specialized hardware implementation is preferable when applicable.

Our work on this book motivated us to write a scientific paper on hash tables [205]. This paper describes several generalizations of the data structure described above, in particular, how automatic adaptive growing and shrinking of the table can be implemented. There is also an extensive comparison with other concurrent hash tables.

## 4.7 Implementation Notes

Although hashing is an algorithmically simple concept, a clean, efficient, and robust implementation can be surprisingly difficult. Less surprisingly, the hash functions used are an 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`, `github.com/aappleby/smhasher/wiki/SMHasher`, `cyan4973.github.io/xxHash/`, `github.com/minio/highwayhash`. 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. The Wikipedia pages on "*Universal hashing*" and "*List of hash functions*" mention universal hash functions also.

Unfortunately, there is no implementation study covering all of the fastest families. Thorup [311] implemented a fast 1-universal family with additional properties. We suggest using the family $H^{\oplus[]}$ considered in Exercise 4.16 for integer keys, and the functions in Exercise 4.9 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 we are dealing with many short lists, some deviations from the implementation scheme described in Sect. 3.2 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. 3.2.1. 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. In comparison, a slim table wastes the space occupied by $m$ pointers and a fat table wastes the space of the unoccupied table positions (see Exercise 4.7). 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. 3.9.

Comparing the space consumption of hashing with chaining and hashing with linear probing is even more subtle than what 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. 3.2.1 for a discussion of memory management for linked lists. On the other hand, the slim table implementations of unbounded hash tables based on chaining can avoid occupying two tables during reallocation by use of 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. For fat tables and for hashing with linear probing, the use of two tables during reallocation seems necessary at first sight. However, even for them reallocation can be avoided. The results are hash tables that *never* consume significantly more space than what is needed anyway just to store the elements [204].

**Exercise 4.24.** 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 8 bytes.

**Exercise 4.25 (large elements).** Repeat the above measurements with element sizes of 32 and 128. Also, add an implementation of *slim chaining*, where table entries store only pointers to the first list element.

**Exercise 4.26 (large keys).** Discuss the impact of large keys on the relative merits of chaining versus linear probing. Which variant will profit? Why?

**Exercise 4.27.** Implement a hash table data type for very large tables stored on disk. Should you use chaining or linear probing? Why?

### 4.7.1 C++

The C++ standard library did not define a hash table data type until 2011. The new standard, C++11 introduced, such a data type. It offers several variants that are all realized by hashing with chaining: *unordered_set*, *unordered_map*, *unordered_multiset*, and *unordered_multimap*. 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 that multiple elements with the same key are allowed. 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. This approach allows the hash function to store internal state such as random coefficients.

Unfortunately, the current implementations of these data structures are in many situations considerably slower than what is possible. For example, consider the case of bounded tables with small elements and suppose that space is not too much at a premium. Here, a specialized implementation of linear probing can be several times faster than library implementations that pay a high price for generality.

LEDA [194] offers several hashing-based implementations of associative arrays. The class *h_array*$\langle Key, T\rangle$ offers associative arrays for storing objects of type $T$ with keys of type *Key*. 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 4.28 (associative arrays).** Implement a C++ class for associative arrays. The implementation should offer the *operator*`[]` for any index type that supports a hash function. Overload the assignment operator such that the assignment $H[x] = \cdots$ works as expected if $x$ is the key of a new element.

*Concurrent hash tables.* Our implementations [205] are available at `github.com/TooBiased` and provide good scalability in many situations. More general functionality is provided by the concurrent hash table in the Intel TBB library. However, the price paid for this generality is quite high, for this and other implementations. Currently, one should use such libraries only if the hash table accesses do not consume a significant fraction of the overall work [205].

### 4.7.2 Java

The class *java.util.HashMap* implements unbounded hash tables using the function *hashCode* as a hash function. The function *hashCode* must be defined for the objects stored in the hash table. A concurrent hash table is available as *java.util.concurrent.ConcurrentHashMap*. The caveat about performance mentioned in the preceding paragraph also applies to this implementation [205].

## 4.8 Historical Notes and Further Findings

Hashing with chaining and hashing with linear probing were used as early as the 1950s [251]. The analysis of hashing began soon after. In the 1960s and 1970s, average-case analysis in the spirit of Theorem 4.1 and Exercise 4.8 prevailed. Various schemes for random sets of keys or random hash functions were analyzed. An early survey paper was written by Morris [231]. The book [184] 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 $T_{\text{fail}}$ of table accesses for an unsuccessful search and the number $T_{\text{success}}$ 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 [61]. The original paper proved Theorem 4.3 and introduced the universal classes discussed in Exercise 4.12. More on universal hashing can be found in [15].

In Sect. 4.2, we described a method for finding a prime number in an interval $k^3..(k+1)^3$. Of course, the method will only be successful if the interval actually contains a prime, i.e., the statement $(\mathrm{A}_k)$ holds. For $k > e^{e^k}$ [69] the statement has been proven. If the *Riemann Hypothesis*, one of the most famous, yet unproven conjectures in number theory, is true, $(\mathrm{A}_k)$ holds for all $k \geq 1$ [60]. For the application to hashing, the range $k \leq 8 \cdot 10^7$ is more than sufficient: $(\mathrm{A}_k)$ holds for all such $k$. On the internet, one can find tables of primes and also tables of gaps between primes. The web page `primes.utm.edu/notes/GapsTable.html` lists *largest gaps* between primes for the range up to $4 \cdot 10^{17}$. One can conclude from this table that $(\mathrm{A}_k)$ holds for $k \leq 3000$. It is also known [261] that, for all primes $p > 11 \cdot 10^9$, the distance from $p$ to the next larger prime is at most $p/(2.8 \cdot 10^7)$. A simple computation shows that this implies $(\mathrm{A}_k)$ for all $k$ between 3000 and $8.4 \cdot 10^7$.

The sieve of Eratosthenes is, by far, not the most efficient method of finding primes, and should only be used for finding primes that are less than a few billion. A better method of finding larger primes is based on the fact that primes are quite abundant. Say we want to find a prime in the vicinity of a given number $m$. We repeatedly choose a random number from the interval $[m, 2m]$ (in this interval, there will be $\Omega(m/\ln m)$ primes) and test whether the number is a prime using an efficient randomized primality test [138, p. 254]. Such an algorithm will require storage space $\mathrm{O}(\log p)$ and even a naive implementation will run in expected time $\mathrm{O}\big((\log p)^3\big)$. With such an algorithm, one can find primes with thousands of decimal digits.

Perfect hashing was a black art until Fredman, Komlós, and Szemeredi [113] introduced the construction shown in Theorem 4.8. Dynamization is due to Dietzfelbinger et al. [94]. Cuckoo hashing [247] is an alternative approach to dynamic perfect hashing, where each element can be stored in two [247] or more [109] places of a table. If these places consist of buckets with several slots, we obtain a highly space efficient data structure [95, 204].

A *minimal perfect hash function* bijectively maps a set $S \subseteq 0..U-1$ to the range $0..n-1$, where $n = |S|$. The goal is to find a function that can be evaluated in constant time and requires little space for its representation – $\Omega(n)$ bits is a lower bound. For a few years now, there have been practicable schemes that achieve this bound [37, 52, 234]. One variant assumes three truly random hash functions[12] $h_i : 0..U-1 \to im/3..(i+1)m/3-1$ for $i \in 0..2$ and $m = \alpha n$, where $\alpha \approx 1.23n$. In a first step, called

[12] 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 [95].

the *mapping step*, one searches for an injective function $p\colon S \to 0..m-1$ such that $p(x) \in \{h_0(x), h_1(x), h_2(x)\}$ for all $x \in S$. It can be shown that such a $p$ can be found in linear time with high probability. Next one determines a function $g\colon 0..m-1 \to \{0,1,2\}$ such that

$$p(x) = h_i(x), \text{ where } i = g(h_0(x)) \oplus g(h_1(x)) \oplus g(h_2(x)) \bmod 3 \quad \text{for all } x \in S.$$

The function $g$ is not hard to find by a greedy algorithm. It is stored as a table of $O(n)$ bits. In a second step, called the *ranking step*, the set $0..m$ is mapped to $0..n-1$ via a function $rank(i) = |\{k \in S : p(k) \leq i\}|$. Then $h(x) = rank(p(x))$ is an injective function from $S$ to $0..n-1$. The task of computing a representation of *rank* that uses only $O(n)$ bits and can be evaluated in constant time is a standard problem in the field of *succinct data structures* [237].

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 $c$-independent*, where $c \geq 1$ is a constant, if for any $k$ distinct keys $x_1$ to $x_k$, and any $k$ hash values $a_1$ to $a_k$, $\text{prob}(h(x_1) = a_1 \wedge \cdots \wedge h(x_k) = a_k) \leq c/m^k$. The polynomials of degree at most $k-1$ with random coefficients in a prime field $\mathbb{Z}_p$ and evaluated in $\mathbb{Z}_p$ are a simple $k$-way 1-independent family of hash functions [61] (see Exercise 4.13 for the case $k = 2$).

The combination of linear probing with simple universal classes such as $H^*$ and $H^{\gg}$ may lead to nonconstant insertion and search time [246, 248]. Only 5-independent classes guarantee constant insertion and search time [246]. Also, tabulation hashing [334] (Exercise 4.16 with $w = 0$) makes linear probing provably efficient [258].

*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 $h(x') = h(x)$.