13

The Dictionary Problem

In this lecture we present *randomized* and simple, yet smart, data structures that solve efficiently the classic **Dictionary Problem**. These solutions will allow us to propose *algorithmic fixes* to some *issues* that are typically left untouched or only addressed via "hand waiving" in basic courses on algorithms.

Problem. *Let $\mathcal{D}$ be a set of n objects, called the* dictionary*, uniquely identified by* `keys` *drawn from a* universe *U. The dictionary problem consists of designing a data structure that efficiently supports the following three basic operations:*

- `Search(`k`):` *Check whether $\mathcal{D}$ contains an object o with key k =* `key[`o`]`*, and then return* `true` *or* `false`*, accordingly. In some cases, we will ask to return the object associated to this key, if any, otherwise return* `null`*.*
- `Insert(`x`):` *Insert in $\mathcal{D}$ the object x indexed by the key k =* `key[`x`]`*. Typically it is assumed that no object in $\mathcal{D}$ has key k, before the insertion takes place; condition which may easily be checked by executing a preliminary query* `Search(`k`)`*.*
- `Delete(`k`):` *Delete from $\mathcal{D}$ the object indexed by the key k, if any.*

In the case that all three operations have to be supported, the problem and the data structure are named *dynamic*; otherwise, if only the query operation has to be supported, the problem and the data structure are named *static*.

We point out that in several applications the structure of an object x typically consists of a pair $\langle k, d \rangle$, where $k \in U$ is the key indexing x in $\mathcal{D}$, and d is the so called *satellite data* featuring x. For the sake of presentation, in the rest of this chapter, we will drop the satellite data and the notation $\mathcal{D}$ in favor just of the key set $S \subseteq U$ which consists of all keys indexing objects in $\mathcal{D}$. This way we will simplify the discussion by considering dictionary search and update operations only on those

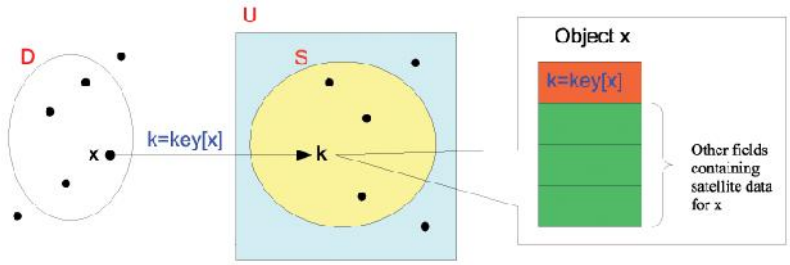

FIGURE 13.1: Illustrative example for U, S, $\mathcal{D}$ and an object x.

keys rather than on (full) objects. But if the context will require also satellite data, we will again talk about objects and their implementing pairs. See Figure 13 for a graphical representation.

Without any loss of generality, we can assume that keys are non-negative integers: $U = \{0, 1, 2, ...\}$. In fact keys are represented in our computer as binary strings, which can thus be interpreted as natural numbers.

In the following sections, we will analyze three main data structures: direct-address tables (or arrays), hash tables (and some of their sophisticated variants) and the Bloom Filter. The former are introduced for teaching purposes, because several times the dictionary problem can be solved very efficiently without resorting involved data structures. The subsequent discussion on hash tables will allow us, first, to fix some issues concerning with the design of a *good* hash function (typically flied over in basic algorithm courses), then, to design the so called *perfect* hash tables, that address *optimally and in the worst case* the static dictionary problem, and then move to the elegant *cuckoo* hash tables, that manage dictionary updates efficiently, still guaranteing constant query time in the *worst case*. The chapter concludes with the *Bloom Filter*, one of the most used data structures in the context of large dictionaries and Web/Networking applications. Its surprising feature is to guarantee query and update operations in constant time, and, more surprisingly, to take space depending on the number of keys n, but not on their lengths. The reason for this impressive *"compression"* is that keys are dropped and only a *fingerprint of few bits* for each of them is stored; the incurred cost is a *one-side error* when executing `Search(`k`)`: namely, the data structure answers in a correct way when $k \in S$, but it may answer un-correctly if k is not in the dictionary, by returning answer *true* (a so called *false positive*). Despite that, we will show that the probability of this error can be mathematically bounded by a function which exponentially decreases with the space m reserved to the Bloom Filter or, equivalently, with the number of bits allocated per each key (i.e. its fingerprint). The nice thing of this formula is that it is enough to take m a constant-factor slightly more than n and reach a negligible probability of error. This makes the Bloom Filter much appealing in several interesting applications: crawlers in search engines, storage systems, P2P systems, etc..

13.1 Direct-address tables

The simplest data structure to support all dictionary operations is the one based on a binary table T, of size $u = |U|$ bits. There is a one-to-one mapping between keys and table's entries, so that entry $T[k]$ is set to 1 *iff* the key $k \in S$. If some satellite data for k has to be stored, then T is implemented as a table of *pointers to* these satellite data. In this case we have that $T[k] \neq$ `NULL` *iff* $k \in S$ and it points to the memory location where the satellite data for k are stored.

Dictionary operations are trivially implemented on T and can be performed in *constant (optimal) time* in the worst case. The main issue with this solution is that table's occupancy depends on the

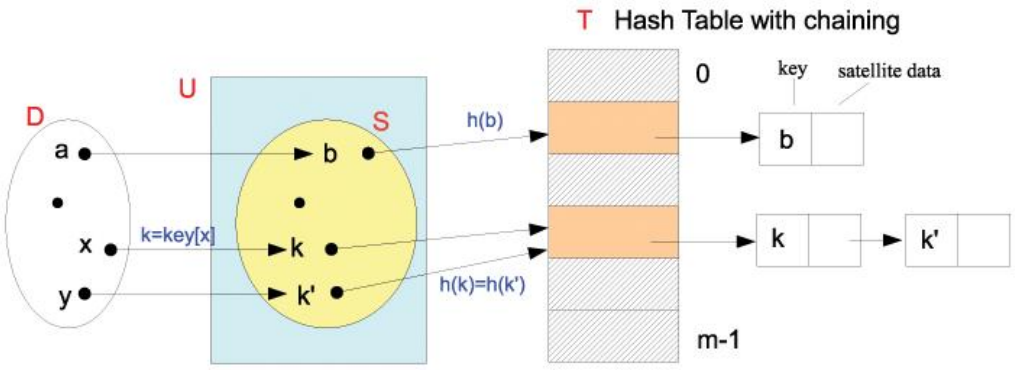

FIGURE 13.2: Hash table with chaining.

universe size u; so if $n = \Theta(u)$, then the approach is optimal. But if the dictionary is small compared to the universe, the approach wastes a lot of space and becomes unacceptable. Take the case of a university which stores the data of its students indexed by their IDs: there can be even million of students but if the IDs are encoded with integers (hence, 4 bytes) then the universe size is 2^32, and thus of the order of billions. Smarter solutions have been therefore designed to reduce the sparseness of the table still guaranteeing the efficiency of the dictionary operations: among all proposals, hash tables and their many variations provide an excellent choice!

13.2 Hash Tables

The simplest data structure for implementing a dictionary are arrays and lists. The former data structure offers constant-time access to its entries but linear-time updates; the latter offers opposite performance, namely linear-time to access its elements but constant-time updates whenever the position where they have to occur is given. Hash tables combine the best of these two approaches, their simplest implementation is the so called *hashing with chaining* which consists of an *array of lists*. The idea is pretty simple, the hash table consists of an array T of size m, whose entries are either `NULL` or they point to lists of dictionary items. The mapping of items to array entries is implemented via an *hash function* $h : U \to \{0, 1, 2 \ldots, m-1\}$. An item with key k is appended to the list pointed to by the entry $T[h(k)]$. Figure 13.2 shows a graphical example of an hash table with chaining; as mentioned above we will hereafter interchange the role of items and their indexing keys, to simplify the presentation, and imply the existence of some satellite data.

Forget for a moment the implementation of the function h, and assume just that its computation takes constant time. We will dedicate to this issue a significant part of this chapter, because the overall efficiency of the proposed scheme strongly depends on the efficiency and efficacy of h to *distribute* items evenly among the table slots.

Given a *good* hash function, dictionary operations are easy to implement over the hash table because they are just turned into operations on the array T and on the lists which spur out from its entries. Searching for an item with key k boils down to a search for this key in the list $T[h(k)]$. Inserting an item x consists of appending it at the front of the list pointed to by $T[h(\texttt{key}[x])]$. Deleting an item with key k consists of first searching for it in the list $T[h(k)]$, and then removing the corresponding object from that list. The running time of dictionary operations is constant for `Insert(`x`)`, provided that the computation of $h(k)$ takes constant time, and it is linear in the length of the list pointed to by $T[h(k)]$ for both the other operations, namely `Search(`k`)` and `Delete(`k`)`.

Therefore, the efficiency of hashing with chaining depends on the ability of the hash function h to *evenly distribute* the dictionary items among the m entries of table T, the more evenly distributed they are the shorter is the list to scan. The worst situation is when all dictionary items are hashed to the same entry of T, thus creating a list of length n. In this case, the cost of searching is $\Theta(n)$ because, actually, the hash table boils down to a single linked list!

This is the reason why we are interested in *good* hash functions, namely ones that distribute items among table slots uniformly at random (aka *simple uniform hashing*). This means that, for such hash functions, every key $k \in S$ is *equally likely* to be hashed to everyone of the m slots in T, *independently* of where other keys are hashed. If h is such, then the following result can be easily proved.

THEOREM 13.1 *Under the hypotheses of simple uniform hashing, there exists a hash table with chaining, of size m, in which the operation* `Search(k)` *over a dictionary of n items takes* $\Theta(1 + n/m)$ *time on average. The value* $\alpha = n/m$ *is often called the* load factor *of the hash table.*

Proof In case of unsuccessful search (i.e. $k \notin S$), the average time for operation `Search(k)` equals the time to perform a full scan of the list $T[h(k)]$, and thus it equals its length. Given the uniform distribution of the dictionary items by h, the average length of a generic list $T[i]$ is $\sum_{x \in S} p(h(\texttt{key}[x]) = i) = |S| \times \frac{1}{m} = n/m = \alpha$. The "plus 1" in the time complexity comes from the constant-time computation of $h(k)$.

In case of successful search (i.e. $k \in S$), the proof is less straightforward. Assume x is the i-th item inserted in T, and let the insertion be executed at the tail of the list $\mathcal{L}(x) = T[h(\texttt{key}[x])]$; we need just one additional pointer per list keep track of it. The number of elements examined during `Search(key[x])` equals the number of items which were present in $\mathcal{L}(x)$ plus 1, i.e. x itself. The average length of $\mathcal{L}(x)$ can be estimated as $n_i = \frac{i-1}{m}$ (given that x is the i-th item to be inserted), so the cost of a successful search is obtained by averaging $n_i + 1$ over all n dictionary items. Namely,

$$\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \frac{\alpha}{2} - \frac{1}{2m}$$

Therefore, the total time is $O(2 + \frac{\alpha}{2} - \frac{1}{2m}) = O(1 + \alpha)$. ∎

The space taken by the hash table can be estimated very easily by observing that list pointers take $O(\log n)$ bits, because they have to index one out of n items, and the item keys take $O(\log u)$ bits, because they are drawn from a universe U of size u. It is interesting to note that the key storage can dominate the overall space occupancy of the table as the universe size increases (think e.g. to URL as keys). It might take even more space than what it is required by the list pointers and the table T (aka, the indexing part of the hash table). This is a simple but subtle observation which will be exploited when designing the Bloom Filter in Section 13.7. To be precise on the space occupancy, we state the following corollary.

COROLLARY 13.1 Hash table with chaining occupies $(m + n) \log_2 n + n \log_2 u$ bits.

It is evident that if the dictionary size n is known, the table can be designed to consists of $m = \Theta(n)$ cells, and thus obtain a constant-time performance over all dictionary operations, on average. If n is unknown, one can resize the table whenever the dictionary gets too small (many deletions), or too large (many insertions). The idea is to start with a table size $m = 2n_0$, where n_0 is the initial number of dictionary items. Then, we keep track of the current number n of dictionary items present in T.

If the dictionary gets too small, i.e. $n < n_0/2$, then we halve the table size and rebuild it; if the dictionary gets too large, i.e. $n > 2n_0$, then we double the table size and rebuild it. This scheme guarantees that, at any time, the table size m is proportional to the dictionary size n by a factor 2, thus implying that $\alpha = m/n = O(1)$. Table rebuilding consists of inserting the current dictionary items in a new table of proper size, and drop the old one. Since insertion takes $O(1)$ time per item, and the rebuilding affects $\Theta(n)$ items to be deleted and $\Theta(n)$ items to be inserted, the total rebuilding cost is $\Theta(n)$. But this cost is paid at least every $n_0/2 = \Omega(n)$ operations, the worst case being the one in which these operations consist of all insertions or all deletions; so the rebuilding cost can be spread over the operations of this sequence, thus adding a $O(1 + m/n) = O(1)$ *amortized cost* at the actual cost of each operation. Overall this means that

COROLLARY 13.2 Under the hypothesis of simple uniform hashing, there exists a *dynamic* hash table with chaining which takes constant time, expected and amortized, for all three dictionary operations, and uses $O(n)$ space.

13.2.1 How do we design a "good" hash function ?

Simple uniform hashing is difficult to guarantee, because one rarely knows the probability distribution according to which the keys are drawn and, in addition, it could be the case that the keys are not drawn independently. Let us dig into this latter feature. Since h maps keys from a universe of size u to a integer-range of size m, it induces a partition of those keys in m subsets $U_i = \{k \in U \ : \ h(k) = i\}$. By the *pigeon principle* it does exist at least one of these subsets whose size is larger than the average load factor u/m. Now, if we reasonably assume that the universe is sufficiently large to guarantee that $u/m = \Omega(n)$, then we can choose the dictionary S as that subset of keys and thus force the hash table to offer its worst behavior, by boiling down to a single linked list of length $\Omega(n)$.

This argument is independent of the hash function h, so we can conclude that no hash function is robust enough to guarantee *always* a "good" behavior. In practice heuristics are used to create hash functions that perform well sufficiently often: The design principle is to compute the hash value in a way that it is expected to be independent of any regular pattern that might exist among the keys in S. The two most famous and practical hashing schemes are based on division and multiplication, and are briefly recalled below (for more details we refer to any classic text in Algorithms, such as [3].

Hashing by division. The hash value is computed as the remainder of the division of k by the table size m, that is: $h(k) = k \bmod m$. This is quite fast and behaves well as long as $h(k)$ does not depend on few bits of k. So power-of-two values for m should be avoided, whereas prime numbers not too much close to a power-of-two should be chosen. For the selection of large prime numbers do exist either simple, but slow (exponential time) algorithms (such as the famous Sieve of Eratosthenes method); or fast algorithms based on some (randomized or deterministic) *primality test*.[1] In general, the cost of prime selection is $o(m)$; and thus turns out to be negligible with respect to the cost of table allocation.

Hashing by multiplication. The hash value is computed in two steps: First, the key k is multiplied by a constant A, with $0 < A < 1$; then, the fractionary part of kA is multiplied by m and the integral part of the result is taken as index into the hash table T. In formula: $h(k) = \lfloor m \, \mathtt{frac}(kA) \rfloor$. An

[1] The most famous, and randomized, primality test is the one by Miller and Rabin; more recently, a deterministic test has been proposed which allowed to prove that this problem is in $\mathcal{P}$. For some more details look at `http://en.wikipedia.org/wiki/Prime_number`.

advantage of this method is that the choice of m is not critical, and indeed it is usually chosen as a power of 2, thus simplifying the multiplication step. For the value of A, it is often suggested to take $A = (\sqrt{5} - 1)/2 \cong 0.618$.

It goes without saying that none of these practical hashing schemes surpasses the problem stated above: it is always possible to select a bad set of keys which makes the table T to boil down to a single linked list, e.g., just take multiples of m to disrupt the hashing-by-division method. In the next section, we propose an hashing scheme that is robust enough to guarantee a "good" behavior on average, whichever is the input dictionary.

13.3 Universal hashing

Let us first argue by a counting argument why the uniformity property, we required to *good* hash functions, is computationally hard to guarantee. Recall that we are interested in hash functions which map keys in U to integers in $\{0, 1, ..., m-1\}$. The total number of such hash functions is $m^{|U|}$, given that each key among the $|U|$ ones can be mapped into m slots of the hash table. In order to guarantee uniform distribution of the keys and independence among them, our hash function should be anyone of those ones. But, in this case, its representation would need $\Omega(\log_2 m^{|U|}) = \Omega(|U| \log_2 m)$ bits, which is really too much in terms of space occupancy and in the terms of computing time (i.e. it would take at least $\Omega(\frac{|U| \log_2 m}{\log_2 |U|})$ time to just read the hash encoding).

Practical hash functions, on the other hand, suffer of several *weaknesses* we mentioned above. In this section we introduce the powerful Universal Hashing scheme which overcomes these drawbacks by means of *randomization* proceeding similarly to what was done to make *more robust* the pivot selection in the Quicksort procedure (see Chapter 5). There, instead of taking the pivot from a fixed position, it was chosen *uniformly at random* from the underlying array to be sorted. This way no input was bad for the pivot-selection strategy, which being unfixed and randomized, allowed to spread the risk over the many pivot choices guaranteeing that most of them led to a good-balanced partitioning.

Universal hashing mimics this algorithmic approach into the context of hash functions. Informally, we do not set the hash function in advance (cfr. fix the pivot position), but we will choose the hash function *uniformly at random* from a *properly defined* set of hash functions (cfr. random pivot selection) which is defined in a way that it is very probable to pick a *good* hash for the current input set of keys S (cfr. the partitioning is balanced). Good function means one that minimizes the number of *collisions* among the keys in S, and can be computed in constant time. Because of the randomization, even if S is fixed, the algorithm will behave differently on various executions, but the nice property of Universal Hashing will be that, on average, the performance will be the expected one. It is now time to formalize these ideas.

DEFINITION 13.1

Let $\mathcal{H}$ be a finite collection of hash functions which map a given universe U of keys into integers in $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be **universal** if, and only if, for all pairs of distinct keys $x, y \in U$ it is:

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

In other words, the class $\mathcal{H}$ is defined in such a way that a randomly-chosen hash function h from this set has a chance to make the distinct keys x and y to collide *no more than* $\frac{1}{m}$. This is exactly the basic property that we deployed when designing hashing with chaining (see the proof of Theorem 13.1). Figure 13.3 pictorially shows this concept.

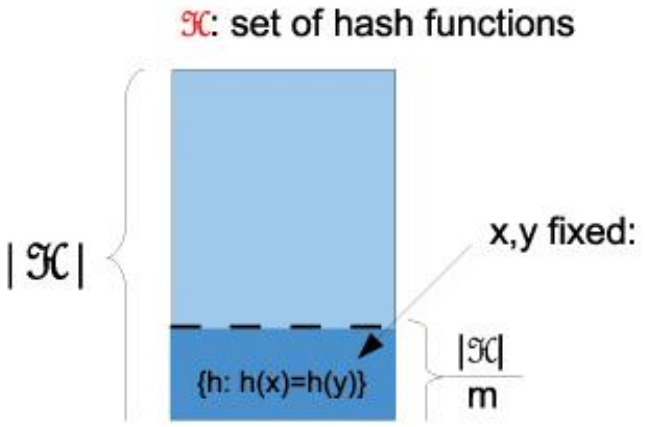

FIGURE 13.3: A schematic figure to illustrate the Universal Hash property.

It is interesting to observe that the definition of Universal Hashing can be extended with some slackness into the guarantee of probability of collision.

DEFINITION 13.2 Let c be a positive constant and $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into integers in $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be c**-universal** if, and only if, for all pairs of distinct keys $x, y \in U$ it is:

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

That is, for each pair of distinct keys, the number of hash functions for which there is a collision between this keys-pair is c times larger than what is guaranteed by universal hashing. The following theorem shows that we can use a universal class of hash functions to design a good hash table with chaining. This specifically means that Theorem 13.1 and its Corollaries 13.1–13.2 can be obtained by substituting the ideal Simple Uniform Hashing with Universal Hashing. This change will be effective in that, in the next section, we will define a real Universal class $\mathcal{H}$, thus making concrete all these mathematical ruminations.

THEOREM 13.2 *Let $T[0, m-1]$ be an hash table with chaining, and suppose that the hash function h is picked at random from a universal class $\mathcal{H}$. The expected length of the chaining lists in T, whichever is the input dictionary of keys S, is still no more than $1 + \alpha$, where α is the load factor n/m of the table T.*

Proof We note that the expectation here is over the choices of h in $\mathcal{H}$, and it does not depend on the distribution of the keys in S. For each pair of keys $x, y \in S$, define the indicator random variable I_{xy} which is 1 if these two keys collide according to a given h, namely $h(x) = h(y)$, otherwise it assumes the value 0. By definition of universal class, given the random choice of h, it is $P(I_{xy} = 1) = P(h(x) = h(y)) \leq 1/m$. Therefore we can derive $E[I_{xy}] = 1 \times P(I_{xy} = 1) + 0 \times P(I_{xy} = 0) = P(I_{xy} = 1) \leq 1/m$, where the average is computed over h's random choices.

Now we define, for each key $x \in S$, the random variable N_x that counts the number of keys other than x that hash to the slot $h(x)$, and thus collide with x. We can write N_x as $\sum_{\substack{y \in S \\ y \neq x}} I_{xy}$. By averaging,

and applying the linearity of the expectation, we get $\sum_{\substack{y \in S \\ y \neq x}} E[I_{xy}] = (n-1)/m < \alpha$. By adding 1, because of x, the theorem follows. ∎

We point out that the time bounds given for hashing with chaining are in expectation. This means that the average length of the lists in T is small, namely $O(\alpha)$, but there could be one or few lists which might be very long, possibly containing up to $\Theta(n)$ items. This satisfies Theorem 13.2 but is of course not a nice situation because it might occur sadly that the *distribution of the searches* privileges keys which belong to the very long lists, thus taking significantly more that the "average" time bound! In order to circumvent this problem, one should guarantee also a small upper bound on the length of the *longest list* in T. This can be achieved by putting some care when inserting items in the hash table.

THEOREM 13.3 *Let T be an hash table with chaining formed by m slots and picking an hash function from a universal class $\mathcal{H}$. Assume that we insert in T a dictionary S of $n = \Theta(m)$ keys, the expected length of the longest chain is $O(\frac{\log n}{\log \log n})$.*

Proof Let h be an hash function picked uniformly at random from $\mathcal{H}$, and let $Q(k)$ be the probability that exactly k keys of S are hashed by h to a particular slot of the table T. Given the universality of h, the probability that a key is assigned to a fixed slot is $\leq \frac{1}{m}$. There are $\binom{n}{k}$ ways to choose k keys from S, so the probability that a slot gets k keys is:

$$Q(k) = \binom{n}{k}\left(\frac{1}{m}\right)^k \left(\frac{m-1}{m}\right)^{n-k} < \frac{e^k}{k^k}$$

where the last inequality derives from Stirling's formula $k! > (k/e)^k$. We observe that there exists a constant $c < 1$ such that, fixed $m \geq 3$ and $k_0 = c \log m / \log \log m$, it holds $Q(k_0) < 1/m^3$.

Let us now introduce M as the length of the longest chain in T, and the random variable $N(i)$ denoting the number of keys that hash to slot i. Then we can write

$$\begin{aligned} P(M = k) &= P(\exists i : N(i) = k \text{ and } N(j) \leq k \text{ for } j \neq i) \\ &\leq P(\exists i : N(i) = k) \ \leq \ m\, Q(k) \end{aligned}$$

where the two last inequalities come, the first one, from the fact that probabilities are ≤ 1, and the second one, from the union bound applied to the m possible slots in T.

If $k \leq k_0$ we have $P(M = k_0) \leq mQ(k_0) < m\frac{1}{m^3} \leq 1/m^2$. If $k > k_0$, we can pick c large enough such that $k_0 > 3 > e$. In this case $e/k < 1$, and so $(e/k)^k$ decreases as k increases, tending to 0. Thus, we have $Q(k) < (e/k)^k \leq (e/k_0)^{k_0} < 1/m^3$, which implies again that $P(M = k) < 1/m^2$.
We are ready to evaluate the expectation of M:

$$E[M] = \sum_{k=0}^{n} k \times P(M = k) = \sum_{k=0}^{k_0} k \times P(M = k) + \sum_{k=k_0+1}^{n} k \times P(M = k) \tag{13.1}$$

$$\leq \sum_{k=0}^{k_0} k \times P(M = k) + \sum_{k=k_0+1}^{n} n \times P(M = k) \tag{13.2}$$

$$\leq k_0 \sum_{k=0}^{k_0} P(M = k) + n \sum_{k=k_0+1}^{n} P(M = k) \tag{13.3}$$

$$= k_0 \times P(M \leq k_0) + n \times P(M > k_0) \tag{13.4}$$

We note that $P(M \leq k_0) \leq 1$ and

$$Pr(M > k_0) = \sum_{k=k_0+1}^{n} P(M = k) < \sum_{k=k_0+1}^{n} (1/m^2) < n(1/n^2) = 1/n.$$

By combining together all these inequalities we can conclude that $E[M] \leq k_0 + n(1/n) = k_0 + 1 = O(\log m / \log\log m)$, which is the thesis since we assumed $m = \Theta(n)$. ∎

Two observations are in order at this point. The condition on $m = \Theta(n)$ can be easily guaranteed by applying the *doubling method* to the table T, as we showed in Theorem 13.2. The bound on the maximum chain length is on average, but it can be turned into worst case via a simple argument. We start by picking a random hash function $h \in \mathcal{H}$, hash every key of S into T, and see whether the condition on the length of the longest chain is at most twice the expected length $\log m / \log\log m$. If so, we use T for the subsequent search operations, otherwise we pick a new function h and re-insert all items in T. A constant number of trials suffice to satisfy that bound[2], thus taking $O(n)$ construction time in expectation.

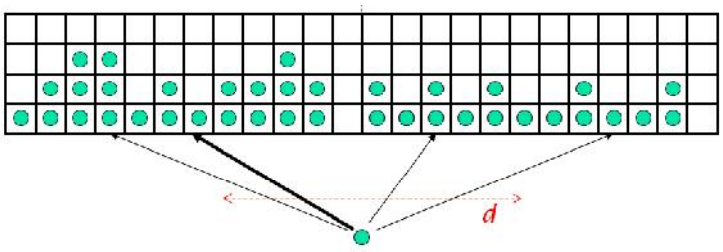

FIGURE 13.4: Example of d-left hashing with four subtables, four hash functions, and each table entry consisting of a bucket of a 4 slots.

Surprisingly enough, this result can be further improved by using two or more, say d, hash functions and d sub-tables $T_1, T_2, ..., T_d$ of the same size m/d, for a total space occupancy equal to the classic single hash table T. Each table T_i is indexed by a different hash function h_i ranging in $\{0, 1, \ldots, m/d - 1\}$. The specialty of this scheme resides in the implementation of the procedure `Insert(`k`)`: it tests the loading of the d slots $T_i[h_i(k)]$, and inserts k in the sparsest one. In the case of a tie, about slots' loading, the algorithm chooses the leftmost table, i.e. the one with minimum index i. For this reason this algorithmic scheme is also known as *d-left hashing*. The implementation of `Search(`k`)` follows consequently, we need to search all d lists $T_i[h_i(k)]$ because we do not know which were their loading when k was inserted, if any. The time cost for `Insert(`k`)` is $O(d)$ time, the time cost of `Search(`k`)` is given by the total length of the d searched lists. We can upper bound this length by d times the length of the longest list in T which, surprisingly, can be proved to be $O(\log\log n)$, when $d = 2$, and it is $\frac{\log\log n}{\log d} + O(1)$ for larger $d > 2$. This result has to be compared against the bound $O(\frac{\log n}{\log\log n})$ obtained for the case of a single hash function in Theorem 13.3. So, by just using one more hash function, we can get an exponential reduction in the search time: this sur-

[2]Just use the Markov bound to state that the longest list longer than twice the average may occur with probability $\leq 1/2$.

prising result is also known in the literature as *the power of two choices*, exactly because choosing between two slots the sparsest one allows to reduce exponentially the longest list.

As a corollary we notice that this result can be used to design a better hash table which does not use chaining-lists to manage collisions, thus saving the space of pointers and increasing locality of reference (hence less cache/IO misses). The idea is to allocate *small and fixed-size* buckets per each slot, as it is illustrated in Figure 13.4. We can use two hash functions and buckets of size $c \log\log n$, for some small $c > 1$. The important implication of this result is that even for just two hash functions there is a large reduction in the maximum list length, and thus search time.

13.3.1 Do universal hash functions exist?

The answer is positive and, surprisingly enough, universal hash functions can be easily constructed as we will show in this section for three classes of them. We assume, without loss of generality, that the table size m is a prime number and keys are integers represented as bit strings of $\log_2 |U|$ bits.[3] We let $r = \frac{\log_2 |U|}{\log_2 m}$ and assume that this is an integer. We decompose each key k in r parts, of $\log_2 m$ bits each, so $k = [k_0, k_1, ...k_{r-1}]$. Clearly each part k_i is an integer smaller than m, because it is represented in $\log_2 m$ bits. We do the same for a generic integer $a = [a_0, a_1, ..., a_{r-1}] \in [1, |U| - 1]$ used as the parameter that defines the universal class of hash functions $\mathcal{H}$ as follows: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \mod m$. The size of $\mathcal{H}$ is $m^r = |U| - 1$, because we have one function per positive value of a.

THEOREM 13.4 *The class $\mathcal{H}$ that contains the following hash functions: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \mod m$, where m is prime and a is a positive integer smaller than $|U|$, is universal.*

Proof Suppose that x and y are two distinct keys which differ, hence, on at least one bit. For simplicity of exposition, we assume that a differing bit falls into the first part, so $x_0 \neq y_0$. According to Definition 13.1, we need to count how many hash functions make these two keys collide; or equivalently, how many a do exist for which $h_a(x) = h_a(y)$. Since $x_0 \neq y_0$, and we operate in arithmetic modulo a prime (i.e. m), the inverse $(x_0 - y_0)^{-1}$ must exist and it is an integer in the range $[1, |U| - 1]$, and so we can write:

$$
\begin{aligned}
h_a(x) = h_a(y) &\Leftrightarrow \sum_{i=0}^{r-1} a_i x_i \equiv \sum_{i=0}^{r-1} a_i y_i \quad (\mod m) \\
&\Leftrightarrow a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r-1} a_i (x_i - y_i) \quad (\mod m) \\
&\Leftrightarrow a_0 \equiv \left(-\sum_{i=1}^{r-1} a_i (x_i - y_i) \right) (x_0 - y_0)^{-1} \quad (\mod m)
\end{aligned}
$$

The last equation actually shows that, whatever is the choice for $[a_1, a_2, ..., a_{r-1}]$, there exists only one choice for a_0 (the one specified in the last line above) which causes x and y to collide. As a consequence, there are $m^{r-1} = (|U| - 1)/m$ choices for $[a_1, a_2, ..., a_{r-1}]$ that cause x and y to collide. These are $< |U|/m$ so the Definition 13.1 of Universal Hash class is satisfied. ∎

[3] This is possible by pre-padding the key with 0, thus preserving its integer value.

It is possible to turn the previous definition holding for any table size m, thus not only for prime values. The key idea is to make a *double modular computation* by means of a large prime $p > |U|$, and a generic integer $m \ll |U|$ equal to the size of the hash table we wish to set up. We can then define an hash function parameterized in two values $a \geq 1$ and $b \geq 0$: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$, and then define the family $\mathcal{H}_{p,m} = \bigcup_{a>0,b\geq 0}\{h_{a,b}\}$. It can be shown that $\mathcal{H}_{p,m}$ is a universal class of hash functions.

The above two definitions require r multiplications and r modulo operations. There are indeed other universal hashing which are faster to be computed because they rely only on operations involving power-of-two integers. As an example take $|U| = 2^h$, $m = 2^l < |U|$ and a be an odd integer smaller than $|U|$. Define the class $\mathcal{H}_{h,l}$ that contains the following hash functions: $h_a(k) = (ak \bmod 2^h)$ `div` 2^{h-l}. This class contains 2^{h-1} distinct functions because a is odd and smaller than $|U| = 2^h$. The following theorem presents the most important property of this class:

THEOREM 13.5 *The class $\mathcal{H}_{h,l} = \{h_a(k) = (ak \mod 2^h)\,\texttt{div}\,2^{h-l}\}$, with $|U| = 2^h$ and $m = 2^l < |U|$ and a odd integer smaller than 2^h, is 2-universal because for any two distinct keys x and y, it is $P(h_a(x) = h_a(y)) \leq \dfrac{1}{2^{l-1}} = \dfrac{2}{m}$.*

Proof Without loss of generality let $x > y$ and define A as the set of possible values for a (i.e. a odd integer smaller than $2^h = |U|$). If there is a collision $h_a(x) = h_a(y)$, then we have:

$$\begin{gathered} ax \mod 2^h \,\texttt{div}\, 2^{h-l} - ay \mod 2^h \,\texttt{div}\, 2^{h-l} = 0 \\ |ax \mod 2^h \,\texttt{div}\, 2^{h-l} - ay \mod 2^h \,\texttt{div}\, 2^{h-l}| < 1 \\ |ax \mod 2^h - ay \mod 2^h| < 2^{h-l} \\ |a(x-y) \mod 2^h| < 2^{h-l} \end{gathered}$$

Set $z = x - y > 0$ (given that keys are distinct and $x > y$) and $z < |U| = 2^h$, it is $z \not\equiv 0(\mod 2^h)$ and $az \not\equiv 0(\mod 2^h)$ because a is odd, so we can write:

$$az \mod 2^h \in \{1, ..., 2^{h-l} - 1\} \cup \{2^h - 2^{h-l} + 1, ..., 2^h - 1\} \tag{13.5}$$

In order to estimate the number of $a \in A$ that satisfy this condition, we write z as $z'2^s$ with z' odd and $0 \leq s < h$. The odd numbers $a = 1, 3, 7, ..., 2^h - 1$ create a mapping $a \mapsto az' \mod 2^h$ that is a permutation of A because z' is odd. So, if we have the set $\{a2^s \mid a \in A\}$, a possible permutation is so defined: $a2^s \mapsto az'2^s \mod 2^h = az \mod 2^h$. Thus, the number of $a \in A$ that satisfy Eqn. 13.5 is the same as the number of $a \in A$ that satisfy:

$$a2^s \mod 2^h \in \{1, ..., 2^{h-l} - 1\} \cup \{2^h - 2^{h-l} + 1, ..., 2^h - 1\}$$

Now, $a2^s \mod 2^h$ is the number represented by the $h - s$ least significant bits of a, followed by s zeros. For example:

- If we take $a = 7$, $s = 1$ and $h = 3$:
  $7 * 2^1 \mod 2^3$ is in binary $111_2 * 10_2 \mod 1000_2$ that is equal to $1110_2 \mod 1000_2 = 110_2$. The result is represented by the $h - s = 3 - 1 = 2$ least significant bits of a, followed by $s = 1$ zeros.
- If we take $a = 7$, $s = 2$ and $h = 3$:
  $7 * 2^2 \mod 2^3$ is in binary $111_2 * 100_2 \mod 1000_2$ that is equal to $11100_2 \mod 1000_2 = 100_2$. The result is represented by the $h - s = 3 - 2 = 1$ least significant bits of a, followed by $s = 2$ zeros.

- If we take $a = 5$, $s = 2$ and $h = 3$:
  $5{*}2^2 \mod 2^3$ is in binary $101_2{*}100_2 \mod 1000_2$ that is equal to $10100_2 \mod 1000_2 = 100_2$. The result is represented by the $h-s = 3-2 = 1$ least significant bits of a, followed by $s = 2$ zeros.

So if $s \geq h - l$ there are no values of a that satisfy Eqn. 13.5, while for smaller s, the number of $a \in A$ satisfying that equation is at most 2^{h-l}. Consequently the probability of randomly choosing such a is at most $2^{h-l}/2^{h-1} = 1/2^{l-1}$. Finally, the universality of $\mathcal{H}_{h,l}$ follows immediately because $\frac{1}{2^{l-1}} < \frac{1}{m}$. ∎

13.4 Perfect hashing, minimal, ordered!

The most known algorithmic scheme in the context of hashing is probably that of hashing with chaining, which sets $m = \Theta(n)$ in order to guarantee an average constant-time search; but that optimal time-bound is on average, as most students forget. This forgetfulness is not erroneous in absolute terms, because do indeed exist variants of hash tables that offer a constant-time *worst-case* bound, thus making hashing a competitive alternative to *tree-based* data structures and the *de facto* choice in practice. A crucial concept in this respect is *perfect hashing*, namely, a hash function which avoids collisions among the keys to be hashed (i.e. the keys of the dictionary to be indexed). Formally speaking,

DEFINITION 13.3 A hash function $h : U \to \{0, 1, \ldots, m-1\}$ is said to be *perfect* with respect to a dictionary S of keys if, and only if, for any pair of distinct keys $k', k'' \in S$, it is $h(k') \neq h(k'')$.

An obvious counting argument shows that it must be $m \geq |S| = n$ in order to make perfect-hashing possible. In the case that $m = n$, i.e. the minimum possible value, the perfect hash function is named *minimal* (shortly `MPHF`). A hash table T using an `MPHF` h guarantees $O(1)$ worst-case search time as well as no waste of storage space, because it has the size of the dictionary S (i.e. $m = n$) and keys can be directly stored in the table slots. Perfect hashing is thus a sort of "perfect" variant of direct-address tables (see Section 13.1), in the sense that it achieves constant search time (like those tables), but optimal linear space (unlike those tables).

A (minimal) perfect hash function is said to be *order preserving* (shortly `OP(MP)HF`) iff, $\forall k_i < k_j \in S$, it is $h(k_i) < h(k_j)$. Clearly, if h is also minimal, and thus $m = n$; then $h(k)$ returns the *rank* of the key in the ordered dictionary S. It goes without saying that, the property `OP(MP)HF` strictly depends onto the dictionary S upon which h has been built: by changing S we could destroy this property, so it is difficult, even if not impossible, to maintain this property under a *dynamic* scenario. In the rest of this section we will confine ourselves to the case of *static* dictionaries, and thus a fixed dictionary S.

The design of h is based upon three auxiliary functions h_1, h_2 and g, which are defined as follows:

- h_1 and h_2 are two universal hash functions from strings to $\{0, 1, \ldots, m' - 1\}$ picked randomly from a universal class (see Section 13.3). They are not necessarily perfect, and so they might induce collisions among S's keys. The choice of m' impacts onto the efficiency of the construction, typically it is taken $m' = cn$, where $c > 1$, so spanning a range which is larger than the number of keys.
- g is a function that maps integers in the range $\{0, \ldots, m' - 1\}$ to integers in the range $\{0, \ldots, n - 1\}$. This mapping cannot be perfect, given that $m' \geq n$, and so some output

(a)	Term t	$h_1(t)$	$h_2(t)$	$h(t)$	(b)	x	$g(x)$
	body	1	6	0		0	0
	cat	7	2	1		1	5
	dog	5	7	2		2	0
	flower	4	6	3		3	7
	house	1	10	4		4	8
	mouse	0	1	5		5	1
	sun	8	11	6		6	4
	tree	11	9	7		7	1
	zoo	5	3	8		8	0
						9	1
						10	8
						11	6

TABLE 13.1 An example of an `OPMPHF` for a dictionary S of $n = 9$ strings which are in alphabetic order. Column $h(t)$ reports the lexicographic rank of each key; h is minimal because its values are in $\{0, \ldots, n-1\}$ and is built upon three functions: (a) two random hash functions $h_1(t)$ and $h_2(t)$, for $t \in S$; (b) a properly derived function $g(x)$, for $x \in \{0, 1, \ldots, m'\}$. Here $m' = 11 > 9 = n$.

values could be repeated. The function g is designed in a way that it *properly combines* the values of h_1 and h_2 in order to derive the `OP(MP)HF` h:

$$h(t) = (g(h_1(t)) + g(h_2(t))) \bmod n$$

The construction of g is obtained via an elegant randomized algorithm which deploys *paths in acyclic random graphs* (see below).

Examples for these three functions are given in the corresponding columns of Table 13.1. Although the values of h_1 and h_2 are randomly generated, the values of g are derived by a proper algorithm whose goal is to guarantee that the formula for $h(t)$ maps a string $t \in S$ to its lexicographic rank in S. It is clear that the evaluation of $h(t)$ takes constant time: we need to perform accesses to arrays h_1 and h_2 and g, plus two sums and one modular operation. The total required space is $O(m' + n) = O(n)$ whenever $m' = cn$. It remains to discuss the choice of c, which impacts onto the efficiency of the randomized construction of g and onto the overall space occupancy. It is suggested to take $c > 2$, which leads to obtain a successful construction in $\sqrt{\frac{m}{m-2n}}$ trials. This means about two trials by setting $c = 3$ (see [4]).

THEOREM 13.6 *An* `OPMPHF` *for a dictionary S of n keys can be constructed in $O(n)$ average time. The hash function can be evaluated in $O(1)$ time and uses $O(n)$ space (i.e. $O(n \log n)$ bits); both time and space bounds are worst case and optimal.*

Before digging into the technical details of this solution, let us make an important observation which highlights the power of OPMPHF. It is clear that we could assign the rank to each string by deploying a trie data structure (see Theorem 7.7), but this would incur two main limitations: (i) rank assignment would need a trie search operation which would incur $\Theta(s)$ I/Os, rather than $O(1)$; (ii) space occupancy would be linear in the total dictionary length, rather than in the total dictionary cardinality. The reason is that, the OPMPHF's machinery does not need the string storage thus allowing to pay space proportional to the number of strings rather than their length; but on the other hand, OPMPHF does not allow to compute the rank of strings *not* in the dictionary, an operation which is instead supported by tries via the so called *lexicographic search* (see Chapter 7).

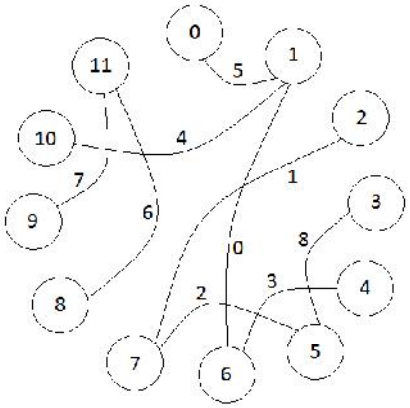

FIGURE 13.5: Graph corresponding to the dictionary of strings S and to the two functions h_1 and h_2 of Table 13.1.

We are left with detailing the algorithm that computes the function g, which will be actually implemented as an array of m' positions storing values bounded by n (see above). So g-array will occupy a total of $O(m' \log_2 n)$ bits. This array must have a quite peculiar feature: its entries have to be defined in a way that the computation

$$h(t) = (g(h_1(t)) + g(h_2(t))) \bmod n$$

returns the rank of term t in S. Surprisingly enough, the computation of g is quite simple and consists of building an undirected graph $G = (V, E)$ with m' nodes labeled $\{0, 1, \ldots, m' - 1\}$ (the same range as the co-domain of h_1 and h_2, and the domain of g) and n edges (as many as the keys in the dictionary) defined according to $(h_1(t), h_2(t))$ labeled with the *desired* value $h(t)$, for each dictionary string $t \in S$. It is evident that the topology of G depends only on h_1 and h_2, and it is exactly what it is called a *random graph*.[4]

Figure 13.5 shows an example of graph G constructed according to the setting of Table 13.1. Take the string $t =$ `body` for which $h_1(t) = 1$ and $h_2(t) = 6$. The lexicographic rank of `body` in the dictionary S is $h(t) = 0$ (it is the first string), which is then the value labeling edge $(1, 6)$. We have to derive $g(t)$ so that 0 must be turn to be equal to $(g(1) + g(6)) \bmod 9$. Of course there are some correct values for entries $g(1)$ and $g(6)$ (e.g. $g(1) = 0$ and $g(6) = 0$), but these values should be correct for all terms t whose edges are incident onto the nodes 1 and 6. Because these edges refer to terms whose g's computation depends on $g(1)$ or $g(6)$. In this respect, the structure of G is useful to drive the instantiation of g's values, as detailed by the algorithm `LabelAcyclicGraph`(G).

The key algorithmic idea is that, if the graph originated by h_1 and h_2 is acyclic, then it can be decomposed into paths. If we assume that the first node, say v_0, of every path takes value 0, as indeed we execute `LabelFrom`$(v_0, 0)$ in `LabelAcyclicGraph`(G), then all other nodes v_i subsequently traversed in this path will have value `undef` for $g(v_i)$ and thus this entry can be easily set by solving the following equation with respect to $g(v_i)$:

$$h(v_{i-1}, v_i) = (g(v_{i-1}) + g(v_i)) \bmod n$$

[4]The reader should be careful that the role of letters n and m' is exchanged here with respect to the traditional graph notation in which n refers to number of nodes and m' refers to number of edges.

Algorithm 13.1 Procedure `LabelAcyclicGraph`(G)

```
1: for v ∈ V do
2:     g[v] = undef
3: end for
4: for v ∈ V do
5:     if g[v] = undef then
6:         LabelFrom(v, 0)
7:     end if
8: end for
```

which actually means to compute:

$$g(v_i) = (\, h(v_{i-1}, v_i) - g(v_{i-1}) \,) \bmod n$$

Algorithm 13.2 Procedure `LabelFrom`(v, c)

```
1: if g[v] ≠ undef then
2:     if g[v] ≠ c then
3:         return - the graph is not acyclic
4:     else
5:         return - the graph is cyclic
6:     end if
7: end if
8: g[v] = c
9: for u ∈ Adj[v] do
10:    LabelFrom(u, h(v, u) − g[v])
11: end for
```

This is exactly what the algorithm `LabelFrom`(v, c) does. It is natural at this point to ask whether these algorithms always find a good assignment to function g or not. It is not difficult to convince ourselves that they return a solution only if the input graph G is acyclic, otherwise they stop. In this latter case, we have to rebuild the graph G, and thus the hash functions h_1 and h_2 by drawing them from the universal class.

What is the likelihood of building an acyclic graph ? This question is quite relevant since the technique is useful only if this *probability of success* is large enough to need few rebuilding steps. According to Random Graph Theory [4], if $m' \leq 2n$ this probability is almost equal to 0; otherwise, if $m' > 2n$ then it is about $\sqrt{\frac{m'-2n}{m'}}$, as we mentioned above. This means that the average number of graphs we will need to build before finding an acyclic one is: $\sqrt{\frac{m'}{m'-2n}}$; which is a constant number if we take $m' = \Theta(n)$. So, on average, the algorithm `LabelAcyclicGraph`(G) builds $\Theta(1)$ graphs of $m' + n = \Theta(n)$ nodes and edges, and thus it takes $\Theta(n)$ time to execute those computations. Space usage is $m' = \Theta(n)$.

In order to reduce the space requirements, we could resort multi-graphs (i.e. graphs with multiple edges between a pair of nodes) and thus use three hash functions, instead of two:

$$h\,(t) = (g\,(h_1\,(t)) + g\,(h_2\,(t)) + g\,(h_3\,(t))) \bmod n$$

We conclude this section by a running example that executes `LabelAcyclicGraph`(G) on the graph in Figure 13.5.

1. Select node 0, set $g(0) = 0$ and start to visit its neighbors, which in this case is just the node 1.
2. Set $g(1) = (h(0,1) - g(0)) \texttt{mod} 9 = (5 - 0) \texttt{mod} 9 = 5$.
3. Take the unvisited neighbors of 1: 6 and 10, and visit them recursively.
4. Select node 6, set $g(6) = (h(1,6) - g(1)) \texttt{mod} 9 = (0 - 5) \texttt{mod} 9 = 4$.
5. Take the unvisited neighbors of 6: 4 and 10, the latter is already in the list of nodes to be explored.
6. Select node 10, set $g(10) = (h(1,10) - g(1)) \texttt{mod} 9 = (4 - 5) \texttt{mod} 9 = 8$.
7. No unvisited neighbors of 10 do exist.
8. Select node 4, set $g(4) = (h(4,6) - g(6)) \texttt{mod} 9 = (3 - 4) \texttt{mod} 9 = 8$.
9. No unvisited neighbors of 4 do exist.
10. No node is left in the list of nodes to be explored.
11. Select a new starting node, for example 2, set $g(2) = 0$, and select its unvisited neighbor 7.
12. Set $g(7) = (h(2,7) - g(2)) \texttt{mod} 9 = (1 - 0) \texttt{mod} 9 = 1$, and select its unvisited neighbor 5.
13. Set $g(5) = (h(7,5) - g(7)) \texttt{mod} 9 = (2 - 1) \texttt{mod} 9 = 1$, and select its unvisited neighbor 3.
14. Set $g(3) = (h(3,5) - g(5)) \texttt{mod} 9 = (8 - 1) \texttt{mod} 9 = 7$.
15. No node is left in the list of nodes to be explored.
16. Select a new starting node, for example 8, set $g(8) = 0$, and select its unvisited neighbor 11.
17. Set $g(11) = (h(8,11) - g(8)) \texttt{mod} 9 = (6 - 0) \texttt{mod} 9 = 6$, and select its unvisited neighbor 9.
18. Set $g(9) = (h(11,9) - g(11)) \texttt{mod} 9 = (7 - 6) \texttt{mod} 9 = 1$.
19. No node is left in the list of nodes to be explored.
20. Since all other nodes are isolated, their g's value is set to 0;

It goes without saying that, if S undergoes some insertions, we possibly have to rebuild $h(t)$. Therefore, all of this works for a static dictionary S.

13.5 A simple perfect hash table

If ordering and minimality (i.e. $h(t) < n$) is not required, then the design of a (static) perfect hash function is simpler. The key idea is to use a *two-level hashing scheme* with universal hashing functions at each level. The first level is essentially hashing with chaining, where the n keys are hashed into m slots using a universal hash function h; but, unlike chaining, every entry $T[j]$ points to a **secondary hash table** T_j which is addressed by another specific universal hash function h_j. By choosing h_j carefully, we can guarantee that there are no collisions at this secondary level. This way, the search for a key k will consist of two table accesses: one at T according to h, and one at some T_i according to h_i, given that $i = h(k)$. For a total of $O(1)$ time complexity in the worst case.

The question now is to guarantee that: (i) hash functions h_j are perfect and thus elicit no collisions among the keys mapped by h into $T[j]$, and (ii) the total space required by table T and all sub-tables T_j is the optimal $O(n)$. The following theorem is crucial for the following arguments.

THEOREM 13.7 *If we store q keys in a hash table of size $w = q^2$ using a universal hash function, then the probability of having a collision among those keys is less than $1/2$.*

Proof In a set of q elements there are $\binom{q}{2} < q^2/2$ pairs of keys that may collide; if we choose the function h from a universal class, we have that each pair collides with probability $1/w$. If we set $w = q^2$ the expected number of collisions is $\binom{q}{2}\frac{1}{w} < q^2/(2q^2) < \frac{1}{2}$. ∎

We use this theorem in two ways: we will guarantee (i) above by setting the size m_j of hash table T_j as n_j^2 (the square of the number of keys hashed to $T[j]$); we will guarantee (ii) above by setting $m = n$ for the size of table T. The former setting ensures that every hash function h_j is perfect, by just two re-samples on average; the latter setting ensures that the total space required by the sub-tables is $O(n)$ as the following theorem formally proves.

THEOREM 13.8 *If we store n keys in a hash table of size $m = n$ using a universal hash function h, then the expected size of all sub-tables T_j is less than $2n$: in formula, $E[\sum_{j=0}^{m-1} n_j^2] < 2n$ where n_j is the number of keys hashing to slot j and the average is over the choices of h in the universal class.*

Proof Let us consider the following identity: $a^2 = a + 2\binom{a}{2}$ which is true for any integer $a > 0$. We have:

$$
\begin{aligned}
E[\sum_{j=0}^{m-1} n_j^2] &= E[\sum_{j=0}^{m-1}(n_j + 2\binom{n_j}{2})] \\
&= E[\sum_{j=0}^{m-1} n_j] + 2E[\sum_{j=0}^{m-1}\binom{n_j}{2}] \\
&= n + 2E[\sum_{j=0}^{m-1}\binom{n_j}{2}]
\end{aligned}
$$

The former term comes from the fact that $\sum_{j=0}^{m-1} n_j$ equals the total number of items hashed in the secondary level, and thus it is the total number n of dictionary keys. For the latter term we notice that $\binom{n_j}{2}$ accounts for the number of collisions among the n_j keys mapped to $T[j]$, so that $\sum_{j=0}^{m-1}\binom{n_j}{2}$ equals the number of collisions induced by the primary-level hash function h. By repeating the argument adopted in the proof of Theorem 13.7 and using $m = n$, the expected value of this sum is at most $\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}$. Summing these two terms we derive that the total space required by this two-level hashing scheme is bounded by $n + 2\frac{n-1}{2} = 2n - 1 \leq 2n$. ∎

It is important to observe that every hash function h_j is independent on the others, so that if it generates some collisions among the n_j keys mapped to $T[j]$, it can be re-generated without influencing the other mappings. Analogously if at the first level h generates more than zn collisions, for some large constant z, then we can re-generate h. These theorems ensure that the average number of re-generations is a small constant per hash function. In the following subsections we detail insertion and searching with perfect hashing.

Inserting keys with perfect hashing

In order to insert n keys of a dictionary S into a perfect-hashing scheme we proceed as follows:

1. Choose a universal hash function h from the family $H_{p,m}$;
2. Compute the address $h(k)$ of all keys in S, and count in n_j the number of keys that hash to slot j of table T;
3. If there are no collisions, i.e. $n_j = 1$ for all j, then the function h is perfect and we can stop (no secondary level is needed).
4. Otherwise, compute $L = \sum_{j=0}^{m-1} n_j^2$;
5. If $L \geq 2n$, choose a new function h in the family $H_{p,m}$, and return to step 2;
6. Else ($L < 2n$), construct tables T_j of size $m_j = n_j^2$, and define a universal hash function h_j of class H_{p,m_j};
7. For all $j = 0, \ldots, m-1$, store the keys mapped in $T[j]$ by h with the local hash h_j and sub-table T_j;
8. If there are some collisions, take a new hash h_j from class H_{p,m_j} and return to step 7;
9. At this point there are no collisions in any T_j, so the process is finished.

Theorem 13.8 ensures that the probability to extract a good function from the family $H_{p,m}$ is $1/2$, so at the first level we have an average number of extractions equal to 2 to succeed in guaranteeing $L < 2n$. At the second level Theorem 13.7 and the setting $m_j = n_j^2$ ensure that the probability to have a collision by h_j is very low ($< 1/2$): in other words we need on average two extractions per j.

Consider for example the dictionary $S = \{98, 19, 14, 50, 1, 72, 79, 3, 69\}$, with $n = 9$. The hash function of the first level $h(k) = ((ak + b) \bmod p) \bmod m$ is set as $m = 11$, $p = 101$, $a = 2$ and $b = 42$. This gives the structure in Figure 13.6, where at the first level $L = 1 + 1 + 4 + 4 + 4 + 1 = 15 < 2 \times 9$, and at the second level there is no collision.

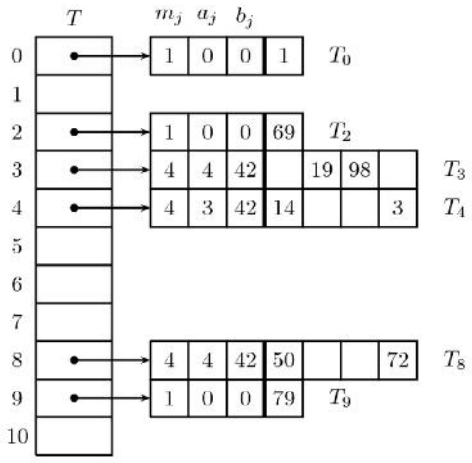

FIGURE 13.6: Inserting and searching keys with perfect hashing.

Searching keys with perfect hashing

The following algorithm describes the search for a given key k within T.

1. Compute $h(k) = ((ak + b) \bmod p) \bmod m$, using the function h defined in the insertion algorithm;
2. Say $j = h(k)$, then $T[j]$ is the pointer to sub-table T_j of size m_j which might contain k;
3. If T_j is empty, then $k \notin S$ and the algorithm stops.
4. Else compute $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$ as the candidate position in T_j for k;
5. Check whether $k = T_j[h_j(k)]$, if they are equal we have found the key, otherwise we can conclude that $k \notin S$.

With reference to Figure 13.6, for a successful search, let us consider the key $98 \in S$; the h function gives:

$$h(98) = ((2 \times 98 + 42) \bmod 101) \bmod 11 = 3.$$

Since $T(3)$ points to T_3 with size $m_3 = 4$, we apply the second level hash function:

$$h_3(98) = ((4 \times 98 + 42) \bmod 101) \bmod 4 = 2.$$

Given that $T_3(2) = 98$, so we have found the key.

For an unsuccessful search, let us consider $k = 8 \notin S$. At the first level we have:

$$h(8) = ((2 \times 8 + 42) \bmod 101) \bmod 11 = 3.$$

Again, we have to look at the table T_3 at the second level:

$$h(8) = ((4 \times 8 + 42) \bmod 101) \bmod 4 = 2.$$

Given that $T_3(2) = 19$, we conclude that the key $k = 8$ is not in the dictionary.

13.6 Cuckoo hashing

When the dictionary is dynamic a different hashing scheme has to be devised, an efficient and elegant solution is the so called **cuckoo hashing**: it achieves constant time in updates, on average, and constant time in searches, in the worst case. The only drawback of this approach is that it makes use of two hash functions that are $O(\log n)$-independent (new results in the literature have significantly relaxed this requirement [1] but we stick on the original scheme for its simplicity). In pills, cuckoo hashing combines the multiple-choice approach of d-left hashing with the ability to move elements. In its simplest form, cuckoo hashing consists of two hash functions h_1 and h_2 and one table T of size m. Any key k is stored either at $T[h_1(k)]$ or at $T[h_2(k)]$, so that searching and deleting operations are trivial: we need to look for k in both those entries, and eventually remove it. Inserting a key is a little bit more tricky in that it can trigger a *cascade* of key moves in the table. Suppose a new key k has to be inserted in the dictionary, according to the Cuckoo scheme it has to be inserted either at position $h_1(k)$ or at position $h_2(k)$. If one of these locations in T is empty (if both are, $h_1(k)$ is chosen), the key is stored at that position and the insertion process is completed. Otherwise, both entries are occupied by other keys, so that we have to create room for k by evicting one of the two keys stored in those two table entries. Typically, the key y stored in $T[h_1(k)]$ is evicted and replaced with k. Then, y plays the role of k and the insertion process is repeated.

There is a warning to take into account at this point. The key y was stored in $T[h_1(k)]$, so that $T[h_i(y)] = T[h_1(k)]$ for either $i = 1$ or $i = 2$. This means that if both positions $T[h_1(y)]$ and $T[h_2(y)]$ are occupied, the key to be evicted cannot be chosen from the entry that was storing $T[h_1(k)]$ because it is k, so this would induce a trivial infinite cycle of evictions over this entry between keys k and y. The algorithm therefore is careful to always avoid to evict the previously inserted key. Nevertheless cycles may arise (e.g. consider the trivial case in which $\{h_1(k), h_2(k)\} = \{h_1(y), h_2(y)\}$) and they can

be of arbitrary length, so that the algorithm must be careful in defining an *efficient escape condition* which detects those situations, in which case it re-sample the two hash functions and re-hash all dictionary keys. The key property, proved in Theorem 13.9, will be to show that cycles occurs with bounded probability, so that the $O(n)$ cost of re-hashing can be amortized, by charging $O(1)$ time per insertion (Corollary 13.4).

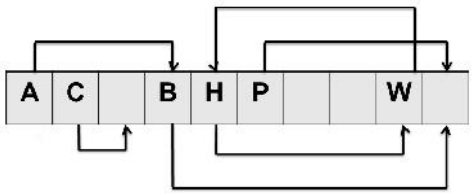

FIGURE 13.7: Graphical representation of cuckoo hashing.

In order to analyze this situation it is useful to introduce the so called *cuckoo graph* (see Figure 13.7), whose nodes are entries of table T and edges represent dictionary keys by connecting the two table entries where these keys can be stored. Edges are directed to keep into account where a key is stored (source), and where a key could be alternatively stored (destination). This way the cascade of evictions triggered by key k traverses the nodes (table entries) laying on a directed path that starts from either node (entry) $h_1(k)$ or node $h_2(k)$. Let us call this path the *bucket* of k. The bucket reminds the list associated to entry $T[h(k)]$ in hashing with chaining (Section 13.2), but it might have a more complicated structure because the cuckoo graph can have cycles, and thus this path can form loops as it occurs for the cycle formed by keys `W` and `H`.

For a more detailed example of insertion process, let us consider again the cuckoo graph depicted in Figure 13.7. Suppose to insert key `D` into our table, and assume that $h_1(D) = 4$ and $h_2(D) = 1$, so that D evicts either `A` or `B` (Figure 13.8). We put `D` in table entry 1, thereby evicting `A` which tries to be stored in entry 4 (according to the directed edge). In turn, `A` evicts `B`, stored in 4, which is moved to the last location of the table as its possible destination. Since such a location is free, `B` goes there and the insertion process is successful and completed. Let us now consider the insertion of key `F`, and assume two cases: $h_1(F) = 2$ and $h_2(F) = 5$ (Figure 13.9.a), or $h_1(F) = 4$ and $h_2(F) = 5$ (Figure 13.9.b). In the former case the insertion is successful: `F` causes `C` to be evicted, which in turn finds the third location empty. It is interesting to observe that, even if we check first $h_2(F)$, then the insertion is still successful: `F` causes `H` to be evicted, which causes `W` to be evicted, which in turn causes again `F` to be evicted. We found a cycle which nevertheless does not induces an infinite loop, in fact in this case the eviction of `F` leads to check its second possible location, namely $h_1(F) = 2$, which evicts `C` that is then stored in the third location (currently empty). Consequently the existence of a cycle does not imply an unsuccessful search; said this, in the following, we will compute the probability of the existence of a cycle as an upper bound to the probability of an unsuccessful search. Conversely, the case of an unsuccessful insertion occurs in Figure 13.9.b where there are two cycles `F`-`H` and `A`-`B` which gets glued together by the insertion of the key `F`. In this case the keys flow from one cycle to the other without stopping. As a consequence, the insertion algorithm must be designed in order to check whether the traversal of the cuckoo graph ended up in an infinite loop: this is done *approximately* by bounding the number of eviction steps.

13.6.1 An analysis

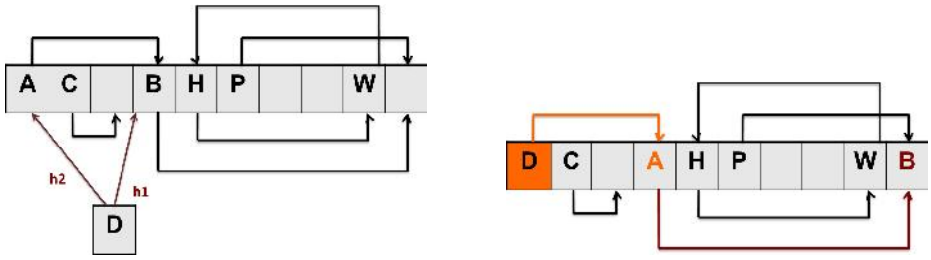

FIGURE 13.8: Inserting the key D: (left) the two entry options, (right) the final configuration.

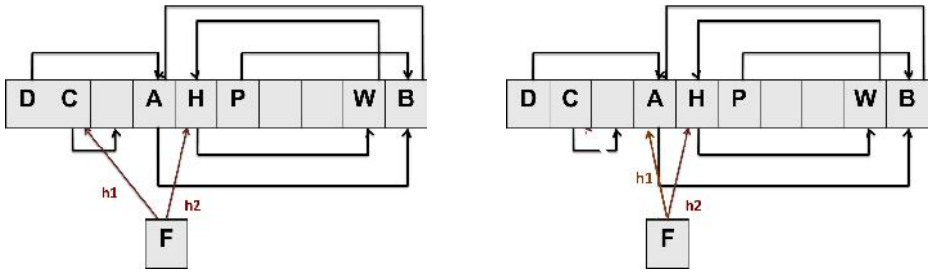

FIGURE 13.9: Inserting the key F: (left) successful insertion, (right) unsuccessful insertion.

Querying and deleting a key k takes constant time, only two table entries have to be checked. The case of insertion is more complicated because it has necessarily to take into account the formation of paths in the (random) cuckoo graph. In the following we consider an *undirected* cuckoo graph, namely one in which edges are not oriented, and observe that a key y is in the bucket of another key x only if there is a path between one of the two positions (nodes) of x and one of the two positions (nodes) of y in the cuckoo graph. This relaxation allows to easy the bounding of the probability of the existence of these paths (recall that m is the table size and n is the dictionary size):

THEOREM 13.9 *For any entries i and j and any constant $c > 1$, if $m \geq 2cn$, then the probability that in the undirected cuckoo graph there exists a shortest path from i to j of length $L \geq 1$ is at most c^{-L}/m.*

Proof We proceed by induction on the path length L. The base case $L = 1$ corresponds to the existence of the undirected edge (i, j); now, every key can generate that edge with probability no more than $\leq 2/m^2$, because edge is undirected and $h_1(k)$ and $h_2(k)$ are uniformly random choices among the m table entries. Summing over all n dictionary keys, and recalling that $m \geq 2cn$, we get the bound $\sum_{k \in S} 2/m^2 = 2n/m^2 \leq c^{-1}/m$.

For the inductive step we must bound the probability that there exists a path of length $L > 1$, but no path of length less than L connects i to j (or vice versa). This occurs only if, for some table entry h, the following two conditions hold:

- there is a shortest path of length $L - 1$ from i to z (that clearly does not go through j);
- there is an edge from z to j.

By the inductive hypothesis, the probability that the first condition is true is bounded by $c^{-(L-1)}/m = c^{-L+1}/m$. The probability of the second condition has been already computed and it is at most $c^{-1}/m = 1/cm$. So the probability that there exists such a path (passing through z) is $(1/cm) * (c^{-L+1}/m) = c^{-L}/m^2$. Summing over all m possibilities for the table entry z, we get that the probability of a path of length L between i and j is at most c^{-L}/m. ∎

In other words, this Theorem states that if the number m of nodes in the cuckoo graph is sufficiently large compared to the number n of edges (i.e. $m \geq 2cn$), there is a low probability that any two nodes i and j are connected by a path, thus fall in the same bucket, and hence participate in a cascade of evictions. Very significant is the case of a constant-length path $L = O(1)$, for which the probability of occurrence is $O(1/m)$. This means that, even for this restricted case, the probability of a large bucket is small and thus the probability of a not-constant number of evictions is small. We can related this probability to the *collision probability* in hashing with chaining. We have therefore proved the following:

THEOREM 13.10 *For any two distinct keys x and y, the probability that x hashes to the same bucket of y is $O(1/m)$.*

Proof If x and y are in the same bucket, then there is a path of some length L between one node in $\{h_1(x), h_2(x)\}$ and one node in $\{h_1(y), h_2(y)\}$. By Theorem 13.9, this occurs with probability at most $4\sum_{L=1}^{\infty} c^{-L}/m = \frac{4}{c-1}/m = O(1/m)$, as desired. ∎

What about rehashing? How often do we have to rebuild table T? Let us consider a sequence of operations involving ϵn insertions, where ϵ is a small constant, e.g. $\epsilon = 0.1$, and assume that the table size is sufficiently large to satisfy the conditions imposed in the previous theorems, namely $m \geq 2cn + 2c(\epsilon n) = 2cn(1+\epsilon)$. Let S' be the final dictionary in which all ϵn insertions have been performed. Clearly, there is a re-hashing of T only if some key insertion induced an infinite loop in the cuckoo graph. In order to bound this probability we consider the final graph in which all keys S' have been inserted, and thus all cycles induced by their insertions are present. This graph consists of m nodes and $n(1+\epsilon)$ keys. Since we assumed $m \geq 2cn(1+\epsilon)$, according to the Theorem 13.9, any given position (node) is involved in a cycle (of any length) if it is involved in a path (of any length) that starts and end at that position: the probability is at most $\sum_{L=1}^{\infty} c^{-L}/m$. Thus, the probability that there is a cycle of any length involving any table entry can be bounded by summing over all m table entries: namely, $m\sum_{L=1}^{\infty} c^{-L}/m = \frac{1}{c-1}$. As we observed previously, this is an upper bound to the probability of an unsuccessful insertion given that the presence of a cycle does not necessarily imply an infinite loop for an insertion.

COROLLARY 13.3 By setting $c = 3$, and taking a cuckoo table of size $m \geq 6n(1+\epsilon)$, the probability for the existence of a cycle in the cuckoo graph of the final dictionary S' is at most $1/2$.

Therefore a constant number of re-hashes are enough to ensure the insertion of ϵn keys in a dictionary of size n. Given that the time for one rehashing is $O(n)$ (we just need to compute two hashes per key), the expected time for all rehashing is $O(n)$, which is $O(1/\epsilon)$ per insertion.

COROLLARY 13.4 By setting $c > 2$, and taking a cuckoo table of size $m \geq 2cn(1+\epsilon)$, the cost for inserting ϵn keys in a dictionary of size n by cuckoo hashing is constant expected amortized. Namely, expected with respect to the random selection of the two universal hash functions driving the cuckoo hashing, and amortised over the $\Theta(n)$ insertions.

In order to make the algorithm works for every n and ϵ, we can adopt the same idea sketched for hashing with chaining and called *global rebuilding technique*. Whenever the size of the dictionary becomes too small compared to the size of the hash table, a new, smaller hash table is created; conversely, if the hash table fills up to its capacity, a new, larger hash table is created. To make this work efficiently, the size of the hash table is increased or decreased by a constant factor (larger than 1), e.g. doubled or halved.

The cost of rehashing can be further reduced by using a very small amount (i.e. constant) of extra-space, called a *stash*. Once a failure situation is detected during the insertion of a key k (i.e. k incurs in a loop), then this key is stored in the stash (without rehashing). This reduces the rehashing probability to $\Theta(1/n^{s+1})$, where s is the size of the stash. The choice of parameter s is related to some structural properties of the cuckoo graph and of the universal hash functions, which are too involved to be commented here (for details see [1] and refs therein).

13.7 Bloom filters

There are situations in which the universe of keys is very large and thus every key is long enough to take a lot of space to be stored. Sometimes it could even be the case that the storage of table pointers, taking $(n+m)\log n$ bits, is much smaller than the storage of the keys, taking $n \log_2 |U|$ bits. An example is given by the dictionary of URLs managed by crawlers in search engines; maintaining this dictionary in internal memory is crucial to ensure the fast operations over those URLs required by crawlers. However URLs are thousands of characters long, so that the size of the indexable dictionary in internal memory could be pretty much limited if whole URLs should have to be stored. And, in fact, crawlers do not use neither cuckoo hashing nor hashing with chaining but, rather, employ a simple and randomised, yet efficient, data structure named *Bloom filter*. The crucial property of Bloom filters is that keys are *not explicitly stored*, only a small fingerprint of them is, and this induces the data structure to make a *one-side error* in its answers to membership queries whenever the queried key is not in the currently indexed dictionary. The elegant solution proposed by Bloom filters is that those errors can be controlled, and indeed their probability *decreases exponentially* with the size of the fingerprints of the dictionary keys. Practically speaking tens of bits (hence, few bytes) per fingerprint are enough to guarantee tiny error probabilities[5] and succinct space occupancy, thus making this solution much appealing in a big-data context. It is useful at this point recall the *Bloom filter principle*: "Wherever a list or set is used, and space is a consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives".

Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n keys and B a bit vector of length m. Initially, all bits in B are set to 0. Suppose we have r universal hash functions $h_i : U \longrightarrow \{0, ..., m-1\}$, for $i = 1, ..., r$. As anticipated above, every key k is not represented explicitly in B but, rather, it is fingerprinted by setting r bits of B to 1 as follows: $B[h_i(k)] = 1, \forall\, 1 \leq i \leq r$. Therefore, inserting a key in a Bloom filter requires $O(r)$ time, and sets at most r bits (possibly some hashes may collide). For searching, we claim that a key y is in S if $B[h_i(y)] = 1, \forall\, 1 \leq i \leq r$. Searching costs $O(r)$, as well as inserting. In the example of Figure 13.10, we can assert that $y \notin S$, since three bits are set to 1 but the last checked bit $B[h_4(y)]$ is zero.

Clearly, if $y \in S$ the Bloom filter correctly detects this; but it might be the case that $y \notin S$ and nonetheless all r bits checked are 1 because of the setting due to other hashes and keys. This is called *false positive* error, because it induces the Bloom filter to return a positive but erroneous answer to

[5] One could object that, errors anyway might occur. But programmers counteract by admitting that these errors can be made smallers than hardware/network errors in data centers or PCs. So they can be neglected!

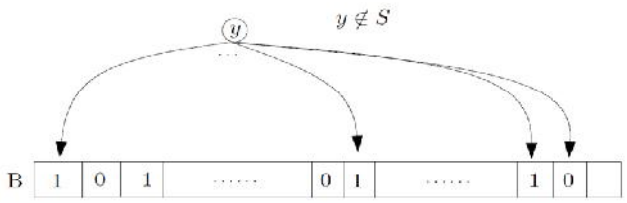

FIGURE 13.10: Searching key y in a Bloom filter.

a membership query. It is therefore natural to ask for the probability of a false-positive error, which can be proved to be bounded above by a surprisingly simple formula.

The probability that the insertion of a key $k \in S$ has left null a bit-entry $B[j]$ equals the probability that the r independent hash functions $h_i(k)$ returned an entry different of j, which is $\left(\frac{m-1}{m}\right)^r \approx e^{-\frac{r}{m}}$. After the insertion of all n dictionary keys, the probability that $B[j]$ is still null can be then bounded by $p_0 \approx \left(e^{-\frac{r}{m}}\right)^n = e^{-\frac{rn}{m}}$ by assuming independencies among those hash functions.[6] Hence the probability of a false-positive error (or, equivalently, the false positive rate) is the probability that all r bits checked for a key not in the current dictionary are set to 1, that is:

$$p_{err} = (1 - p_0)^r \approx \left(1 - e^{-\frac{rn}{m}}\right)^r$$

Not surprisingly the error probability depends on the three parameters that define the Bloom filter's structure: the number r of hash functions, the number n of dictionary keys, and the number m of bits in the binary array B. It's interesting to notice that the fraction $f = m/n$ can be read as the average number of bits per dictionary key allocated in B, hence the fingerprint size f. The larger is f the smaller is the error probability p_{err}, but the larger is the space allocated for B. We can optimize p_{err} according to m and n, by computing the first-order derivative and equalling it to zero: this gets $r = \frac{m}{n} \ln 2$. It is interesting to observe that for this value of r the probability a bit in B gets null value is $p_0 = 1/2$; which actually means that the array is half filled by 1s and half by 0s. And indeed this result could not be different: a larger r induces more 1s in B and thus a larger probability of positive errors, a lower r induces more 0s in B and thus a larger probability of correct answers: the correct choice of r falls in the middle! For this value of $r = \frac{m}{n} \ln 2$, we have $p_{err} = (0.6185)^{m/n}$ which decreases exponentially with the fingerprint size $f = m/n$. Figure 13.11 reports the false positive rate as a function of the number r of hashes for a Bloom-filter designed to use $m = 32n$ bits of space, hence a fingerprint of $f = 32$ bits per key. By using 22 hash functions we can minimize the false positive rate to less than $1.E-6$. However, we also note that adding one more hash function does not significantly decreases the error rate when $r \geq 10$.

It is natural now to derive the size m of the B-array whenever r is fixed to its optimal value $\frac{m}{n} \ln 2$, and n is the number of current keys in the dictionary. We obtain different values of p_{err} depending on the choice of m. For example, if $m = n$ then $p_{err} = 0.6185$, if $m = 2n$ then $p_{err} = 0.38$, and if $m = 5n$ we have $p_{err} = 0.09$. In practice $m = cn$ is a good choice, and for $c > 10$ the error rate is interestingly small and useful for practical applications. Figure 13.12 compares the performance of hashing (with chaining) to that of Bloom filters, assuming that the number r of used hash functions is the one which minimizes the false-positive error rate. In hashing with chaining,

[6] A more precise analysis is possible, but much involved and without changing the final result, so that we prefer to stick on this simpler approach.

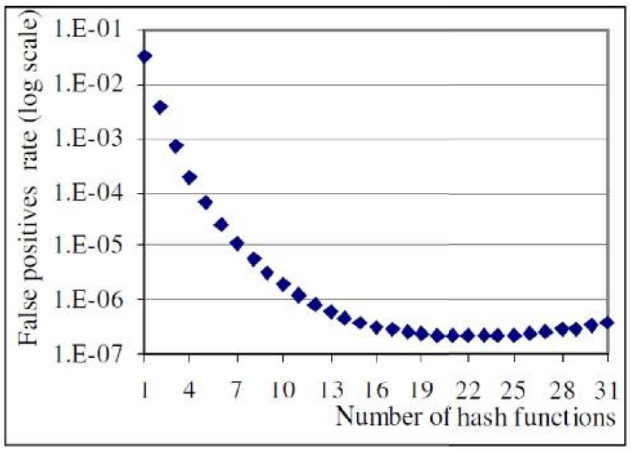

FIGURE 13.11: Plotting p_{err} as a function of the number of hashes.

we need $\Theta(n(\log n + \log u))$ bits to store the pointers in the chaining lists, and $\Theta(m \log n$ is the space occupancy of the table, in bits. Conversely the Bloom filter does not need to store keys and thus it incurs only in the cost of storing the bit array $m = fn$, where f is pretty small in practice as we observed above.

$\diamond$	Hash Tables	Bloom Filters
build time	$\Theta(n)$	$\Theta(n)$
space needed	$\Theta((m+n)\log n + n \log \lvert U \rvert)$	$\Theta(m)$
search time	$O(1)$	$(m/n)\ln 2$
ε value	0	$(0.6185)^{m/n}$

FIGURE 13.12: Hashing vs Bloom filter

13.7.1 A lower bound on space

The question is how small can be the bit array B in order to guarantee an given error rate ϵ for a dictionary of n keys drawn from a universe U of size u. This lower bound will allow us to prove that space-wise Bloom filters are within a factor of $\log_2 e \approx 1.44$ of the asymptotic lower bound.

The proof proceeds as follows. Let us represent any data structure solving the membership query on a dictionary $X \subseteq U$ with those time/error bounds with a m-bit string $F(X)$. This data structure must work for every possible subset of n elements of U, they are $\binom{u}{n}$. We say that an m-bit string s accepts a key x if $s = F(X)$ for some X containing x, otherwise we say that s rejects x. Now, let us consider a specific dictionary X of n elements. Any string s that is used to represent X must accept each one of the n elements of X, since no false negatives are admitted, but it may also accept at most $\epsilon(u-n)$ other elements of the universe, thus guaranteeing a false positive rate smaller than ϵ. Each string s therefore accepts at most $n + \epsilon(u-n)$ elements, and can thus be used to represent any of the

$\binom{n+\epsilon(u-n)}{n}$ subsets of size n of these elements, but it cannot be used to represent any other set. Since we are interested into data structures of m bits, they are 2^m, and we are asking ourselves whether they can represent all the $\binom{u}{n}$ possible dictionaries of U of n keys. Hence, we must have:

$$2^m \times \binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

or, equivalently:

$$m \geq \log_2 \binom{u}{n} / \binom{n+\epsilon(u-n)}{n} \;\geq\; \log_2 \binom{u}{n} / \binom{\epsilon u}{n} \;\geq\; \log_2 \epsilon^{-n} = n \log_2(1/\epsilon)$$

where we used the inequalities $(\frac{a}{b})^b \leq \binom{a}{b} \leq (\frac{ae}{b})^b$ and the fact that in pratice it is $u \gg n$. If we consider a Bloom filter with the same configuration— namely, error rate ϵ and space occupancy m bits—, then we have $\epsilon = (1/2)^r \geq (1/2)^{m \ln 2/n}$ by setting r to the optimal number of hash functions. After some algebraic manipulations, we find that:

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} \approx 1.44\, n\, \log_2(1/\epsilon)$$

This means that Bloom filters are asymptotically optimal in space, the constant factor is 1.44 more than the minimum possible.

13.7.2 Compressed Bloom filters

In many Web applications, the Bloom filter is not just an object that resides in memory, but it is a data structure that must be transferred between proxies. In this context it is worth to investigate whether Bloom filters can be *compressed* to save bandwidth and transfer time [3]. Suppose that we optimize the false positive rate of the Bloom filter under the constraint that the number of bits to be sent after compression is $z \leq m$. As compression tool we can use Arithmetic coding (see Chapter 10), which well approximates the entropy of the string to be compressed: here simply expressed as $-(p(0)\log_2 p(0) + p(1)\log_2 p(1))$ where $p(b)$ is the frequency of bit b in the input string. Surprisingly enough it turns out that using a larger, but sparser, Bloom filter can yield the same false positive rate with a smaller number of transmitted bits. Said in other words, one can transmit the same number of bits but reduce the false positive rate. An example is given in Table 13.2, where the goal is to obtain small false positive rates by using less than 16 transmitted bits per element. Without compression, the optimal number of hash functions is 11, and the false positive rate is 0.000459. By making a sparse Bloom filter using 48 bits per element but only 3 hash functions, one can compress the result down to less than 16 bits per item (with high probability) and decrease the false positive rate by roughly a factor of 2.

Array bits per element	m/n	16	28	48
Transmission bits per element	z/n	16	15,846	15,829
Hash functions	k	11	4	3
False positive probability	f	0,000459	0,000314	0,000222

TABLE 13.2 Using at most sixteen bits per element after compression, a bigger but sparser Bloom filter can reduce the false positive rate.

Compressing a Bloom filter has benefits: (i) it uses a smaller number of hash functions, so that the lookups are more efficient; (ii) it may reduce the false positive rate for a desired compressed size, or reduce the transmited size for a fixed false positive rate. However the size m of the uncompressed

Bloom filter increases the memory usage at running time, and comes at the computational cost of compressing/decompressing it. Nevertheless, some sophisticate approaches are possible which allow to access directly the compressed data without incurring in their decompression. An example was given by the FM-index in Chapter 12, which could built over the bit-array B.

13.7.3 Spectral Bloom filters

A *spectral bloom filter* (SBF) is an extension of the original Bloom filter to multi-sets, thus allowing the storage of multiplicities of elements, provided that they are below a given threshold (a *spectrum* indeed). SBF supports queries on the multiplicities of a key with a small error probability over its estimate, using memory only slightly larger than that of the original Bloom filter. SBF supports also insertions and deletions over the data set.

Let S be a multi-set consisting of n distinct keys from U and let f_x be the multiplicity of the element $x \in S$. In a SBF the bit vector B is replaced by an array of counters $C[0, m-1]$, where $C[i]$ is the sum of f_x-values for those elements $x \in S$ mapping to position i. For every element x, we add the value f_x to the counters $C[h_1(x)], C[h_2(x)], ..., C[h_r(x)]$. Due to possible conflicts among hashes for different elements, the $C[i]$s provide approximated values, specifically upper bounds (given that $f_x > 0$.

The multi-set S can be dynamic. When inserting a new item s, its frequency f_s increases by one, so that we increase by one the counters $C[h_1(s)], C[h_2(s)], \ldots, C[h_r(s)]$; deletion consists symmetrically in decreasing those counters. In order to search for the frequency of element x, we simply return the minimum value $m_x = \min_i C[h_i(x)]$. Of course, m_x is a biased estimator for f_x. In particular, since all $f_x \leq C[h_i]$ for all i, the case in which the estimate is wrong (i.e. $m_x < f_x$) corresponds to the event "all counters $C[h_i(x)]$ have a collision", which in turn corresponds to a "false positive" event in the classical Bloom filter. So, the probability of error in a SBF is the error rate probability for a Bloom filter with the same set of parameters m, n, r.

13.7.4 A simple application

Bloom filters can be used to approximate the intersection of two sets, say A and B, stored in two machines M_A and M_B. We wish to compute $A \cap B$ distributively, by exchanging a small number of bits. Typical applications of the problem are data replication check and distributed search engines. The problem can be efficiently solved by using Bloom filters $BF(A)$ and $BF(B)$ stored in M_A and M_B, respectively. The algorithmic idea to compute $A \cap B$ is as follows:

1. M_A sends $BF(A)$ to M_B, using $r_{opt} = (m_A ln2)/|A|$ hash functions and a bit-array $m_A = \Theta(|A|)$;
2. M_B checks the existence of elements B into A by deploying $BF(A)$ and sends back explicitly the set of found elements, say Q. Note that, in general, $Q \supseteq A \cap B$ because of false positives;
3. M_A computes $Q \cap A$, and returns it.

Since Q contains $|A \cap B|$ keys plus the number of false positives (elements belonging only to A), we can conclude that $|Q| = |A \cap B| + |B|\epsilon$ where $\epsilon = 0.6185^{m_A/|A|}$ is the error rate for that design of $BF(A)$. Since we need $\log |U|$ bits to represent each key, the total number of exchanged bits is $\Theta(|A|) + (|A \cap B| + |B|0.6185^{m_A/|A|}) \log |U|$ which is much smaller than $|A| \log |U|$ the number of bits to be exchanged by using a plain algorithm that sends the whole A's set to M_B.

References

[1] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash. In Proceedings of European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 7501, Springer, 108–120, 2012.

[2] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4): 485-509, 2003.

[3] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networks*, 10(5): 604-612, 2002.

[4] Ian H. Witten, Alistair Moffat and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann, 1999.