Chapter 35

Hashing

By Sariel Har-Peled, April 26, 2022[1]

"I tried to read this book, Huckleberry Finn, to my grandchildren, but I couldn't get past page six because the book is fraught with the 'n-word.' And although they are the deepest-thinking, combat-ready eight- and ten-year-olds I know, I knew my babies weren't ready to comprehend Huckleberry Finn on its own merits. That's why I took the liberty to rewrite Mark Twain's masterpiece. Where the repugnant 'n-word' occurs, I replaced it with 'warrior' and the word 'slave' with 'dark-skinned volunteer."'

Paul Beatty, The Sellout

35.1. Introduction

We are interested here in dictionary data structure. The settings for such a data-structure:
(A) $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
(B) Data structure to store a subset $S \subseteq \mathcal{U}$
(C) **Operations:**
    (A) **search**/**lookup**: given $x \in \mathcal{U}$ is $x \in S$?
    (B) **insert**: given $x \notin S$ add x to S.
    (C) **delete**: given $x \in S$ delete x from S
(D) ***Static*** structure: S given in advance or changes very infrequently, main operations are lookups.
(E) ***Dynamic*** structure: S changes rapidly so inserts and deletes as important as lookups.

Common constructions for such data-structures, include using a static sorted array, where the lookup is a binary search. Alternatively, one might use a *balanced* search tree (i.e., red-black tree). The time to perform an operation like lookup, insert, delete take $O(\log |S|)$ time (comparisons).

Naturally, the above are potently an "overkill", in the sense that sorting is unnecessary. In particular, the universe $\mathcal{U}$ may not be (naturally) totally ordered. The keys correspond to large objects (images, graphs etc) for which comparisons are expensive. Finally, we would like to improve "average" performance of lookups to $O(1)$ time, even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Hashing and Hash Tables. The hash-table data structure has an associated (hash) table/array T of size m (the table ***size***). A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$. An item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.

Given a set $S \subseteq \mathcal{U}$, in a perfect ideal situation, each element $x \in S$ hashes to a distinct slot in T, and we store x in the slot $h(x)$. The **Lookup** for an item $y \in \mathcal{U}$, is to check if $T[h(y)] = y$. This takes constant time.

Unfortunately, *collisions* are unavoidable, and several different techniques to handle them. Formally, two items $x \neq y$ ***collide*** if $h(x) = h(y)$.

[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

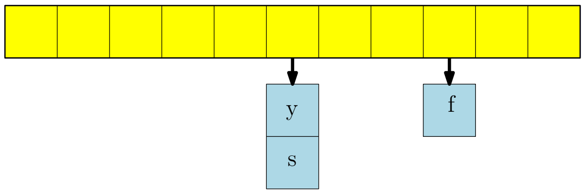

Figure 35.1: Open hashing.

A standard technique to handle collisions is to use ***chaining*** (aka *open hashing*). Here, we handle collisions as follows:

(A) For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list.

(B) **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.

Other techniques for handling collisions include associating a list of locations where an element can be (in certain order), and check these locations in this order. Another useful technique is ***cuckoo hashing*** which we will discuss later on: Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick out the stored value to its other location. Repeat till stable. if no stability then rebuild table.

The relevant questions when designing a hashing scheme, include: (I) Does hashing give $O(1)$ time per operation for dictionaries? (II) Complexity of evaluating h on a given element? (III) Relative sizes of the universe $\mathcal{U}$ and the set to be stored S. (IV) Size of table relative to size of S. (V) Worst-case vs average-case vs randomized (expected) time? (VI) How do we choose h?

The ***load factor*** of the array T is the ratio n/t where $n = |S|$ is the number of elements being stored and $m = |T|$ is the size of the array being used. Typically n/t is a small constant smaller than 1.

In the following, we assume that $\mathcal{U}$ (the universe the keys are taken from) is large – specifically, $N = |\mathcal{U}| \gg m^2$, where m is the size of the table. Consider a hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$. If hash N items to the m slots, then by the pigeon hole principle, there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i. In particular, this implies that there is set $S \subseteq \mathcal{U}$, where $|S| = m$ such that all of S hashes to same slot. Oops.

Namely, for every hash function there is a *bad set* with many collisions.

Observation 35.1.1. *Let $\mathcal{H}$ be the set of all functions from $\mathcal{U} = \{1, \ldots, U\}$ to $\{1, \ldots, m\}$. The number of functions in $\mathcal{H}$ is m^U. As such, specifying a function in $\mathcal{H}$ would require* $\log_2 |\mathcal{H}| = O(U \log m)$.

As such, picking a truely random hash function requires many random bits, and furthermore, it is not even clear how to evaluate it efficiently (which is the whole point of hashing).

Picking a hash function. Picking a good hash function in practice is a dark art involving many non-trivial considerations and ideas. For parameters $N = |\mathcal{U}|$, $m = |T|$, and $n = |S|$, we require the following:

(A) $\mathcal{H}$ is a ***family*** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).

(B) h is chosen *randomly* from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.

(C) Require that for any *fixed* set $S \subseteq \mathcal{U}$, of size m, the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

35.2. Universal Hashing

We would like the hash function to have the following property – For any element $x \in \mathcal{U}$, and a random $h \in \mathcal{H}$, then $h(x)$ should have a uniform distribution. That is $\Pr[h(x) = i] = 1/m$, for every $0 \le i < m$. A somewhat stronger property is that for any two distinct elements $x, y \in \mathcal{U}$, for a random $h \in \mathcal{H}$, the probability of a collision between x and y should be at most $1/m$. $\mathbb{P}[h(x) = h(y)] = 1/m$.

Definition 35.2.1. A family $\mathcal{H}$ of hash functions is ***2-universal*** if for all distinct $x, y \in \mathcal{U}$, we have $\mathbb{P}[h(x) = h(y)] \le 1/m$.

Applying a 2-universal family hash function to a set of distinct numbers, results in a 2-wise independent sequence of numbers.

Lemma 35.2.2. *Let S be a set of n elements stored using open hashing in a hash table of size m, using open hashing, where the hash function is picked from a 2-universal family. Then, the expected lookup time, for any element $x \in \mathcal{U}$ is $O(n/m)$.*

Proof: The number of elements colliding with x is $\ell(x) = \sum_{y \in S} D_y$, where $D_y = 1 \iff x$ and y collide under the hash function h. As such, we have

$$\mathbb{E}[\ell(x)] = \sum_{y \in S} \mathbb{E}\big[D_y\big] = \sum_{y \in S} \mathbb{P}[h(x) = h(y)] = \sum_{y \in S} \frac{1}{m} = |S|/m = n/m. \qquad \blacksquare$$

Remark 35.2.3. The above analysis holds even if we perform a sequence of $O(n)$ insertions/deletions operations. Indeed, just repeat the analysis with the set of elements being all elements encountered during these operations.

The worst-case bound is of course much worse – it is not hard to show that in the worst case, the load of a single hash table entry might be $\Omega(\log n / \log \log n)$ (as we seen in the occupancy problem).

Rehashing, amortization, etc. The above assumed that the set S is fixed. If items are inserted and deleted, then the hash table might become much worse. In particular, $|S|$ grows to more than cm, for some constant c, then hash table performance start degrading. Furthermore, if many insertions and deletions happen then the initial random hash function is no longer random enough, and the above analysis no longer holds.

A standard solution is to rebuild the hash table periodically. We choose a new table size based on current number of elements in table, and a new random hash function, and rehash the elements. And then discard the old table and hash function. In particular, if $|S|$ grows to more than twice current table size, then rebuild new hash table (choose a new random hash function) with double the current number of elements. One can do a similar shrinking operation if the set size falls below quarter the current hash table size.

If the working $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say 10), rebuild.

The ***amortize*** cost of rebuilding to previously performed operations. Rebuilding ensures $O(1)$ expected analysis holds even when S changes. Hence $O(1)$ expected look up/insert/delete time *dynamic* data dictionary data structure!

35.2.1. How to build a 2-universal family

35.2.1.1. A quick reminder on working modulo prime

Definition 35.2.4. For a number p, let $\mathbb{Z}_n = \{0, \ldots, n-1\}$.

For two integer numbers x and y, the ***quotient*** of x/y is $x \operatorname{div} y = \lfloor x/y \rfloor$. The ***remainder*** of x/y is $x \bmod y = x - y \lfloor x/y \rfloor$. If the $x \bmod y = 0$, than y ***divides*** x, denoted by $y \mid x$. We use $\alpha \equiv \beta \pmod{p}$ or $\alpha \equiv_p \beta$ to denote that α and β are ***congruent modulo*** p; that is $\alpha \bmod p = \beta \bmod p$ – equivalently, $p \mid (\alpha - \beta)$.

Remark 35.2.5. A quick review of what we already know. Let p be a prime number.

(A) Lemma 35.6.1: For any $\alpha, \beta \in \{1, \ldots, p-1\}$, we have that $\alpha\beta \not\equiv 0 \pmod{p}$.
(B) Lemma 35.6.1: For any $\alpha, \beta, i \in \{1, \ldots, p-1\}$, such that $\alpha \neq \beta$, we have that $\alpha i \not\equiv \beta i \pmod{p}$.
(C) Lemma 35.6.1: For any $x \in \{1, \ldots, p-1\}$ there exists a unique y such that $xy \equiv 1 \pmod{p}$. The number y is the ***inverse*** of x, and is denoted by x^{-1} or $1/x$.
(D) Lemma 35.6.3: For any numbers $x, y \in \mathbb{Z}_p$. If $x \neq y$ then, for any $a, b \in \mathbb{Z}_p$, such that $a \neq 0$, we have $ax + b \not\equiv ay + b \pmod{p}$.
(E) Lemma 35.6.2: For any numbers $x, y \in \mathbb{Z}_p$. If $x \neq y$ then, for each pair of numbers $r, s \in \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, such that $r \neq s$, there is exactly one unique choice of numbers $a, b \in \mathbb{Z}_p$ such that $ax + b \pmod{p} = r$ and $ay + b \pmod{p} = s$.

35.2.1.2. Constructing a family of 2-universal hash functions

For parameters $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$. Choose a ***prime*** number $p \geq N$. Let

$$\mathcal{H} = \left\{ h_{a,b} \;\middle|\; a, b \in \mathbb{Z}_p \text{ and } a \neq 0 \right\},$$

where $h_{a,b}(x) = ((ax + b) \pmod{p}) \pmod{m}$. Note that $|\mathcal{H}| = p(p-1)$.

35.2.1.3. Analysis

Once we fix a and b, and we are given a value x, we compute the hash value of x in two stages:
(A) ***Compute***: $r \leftarrow (ax + b) \pmod{p}$.
(B) ***Fold***: $r' \leftarrow r \pmod{m}$

Lemma 35.2.6. *Assume that p is a prime, and $1 < m < p$. The number of pairs $(r, s) \in \mathbb{Z}_p \times \mathbb{Z}_p$, such that $r \neq s$, that are folded to the same number is $\leq p(p-1)/m$. Formally, the set of bad pairs*

$$B = \left\{ (r, s) \in \mathbb{Z}_p \times \mathbb{Z}_p \;\middle|\; r \equiv_m s \right\}$$

is of size at most $p(p-1)/m$.

Proof: Consider a pair $(x, y) \in \{0, 1, \ldots, p-1\}^2$, such that $x \neq y$. For a fixed x, there are at most $\lceil p/m \rceil$ values of y that fold into x. Indeed, $x \equiv_m y$ if and only if

$$y \in L(x) = \{x + im \mid i \text{ is an integer}\} \cap \mathbb{Z}_p.$$

The size of $L(x)$ is maximized when $x = 0$, The number of such elements is at most $\lceil p/m \rceil$ (note, that since p is a prime, p/m is fractional). One of the numbers in $O(x)$ is x itself. As such, we have that

$$|B| \leq p(|L(x)| - 1) \leq p(\lceil p/m \rceil - 1) \leq p(p-1)/m,$$

since $\lceil p/m \rceil - 1 \leq (p-1)/m \iff m \lceil p/m \rceil - m \leq p - 1 \iff m \lfloor p/m \rfloor \leq p - 1 \iff m \lfloor p/m \rfloor < p$, which is true since p is a prime, and $1 < m < p$. ■

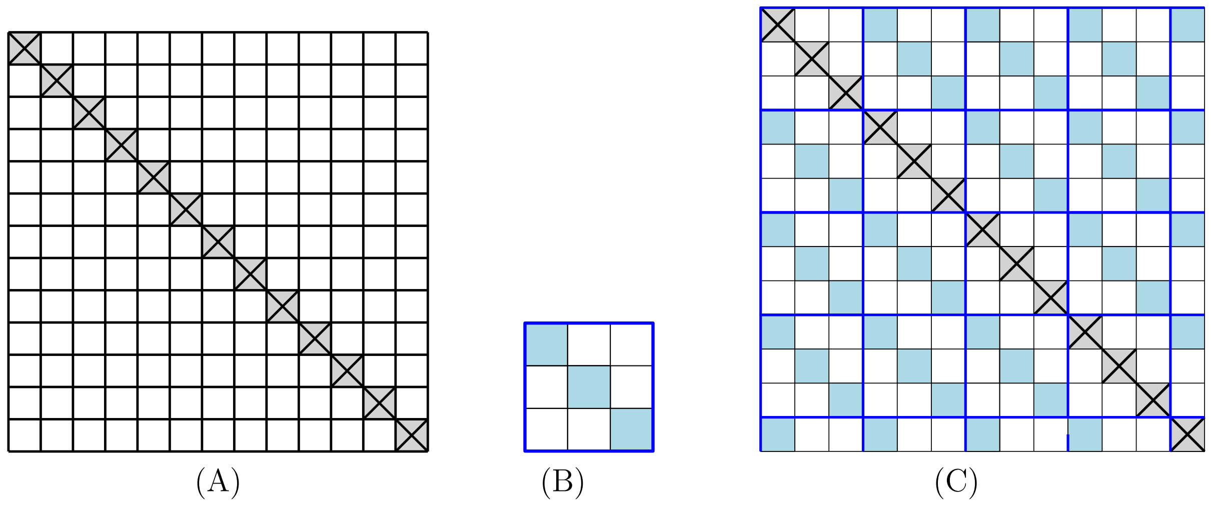

Figure 35.2: Explanation of the hashing scheme via figures.

Claim 35.2.7. *For two distinct numbers $x, y \in \mathcal{U}$, a pair a, b is* **bad** *if $h_{a,b}(x) = h_{a,b}(y)$. The number of bad pairs is $\leq p(p-1)/m$.*

Proof: Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$. Let

$$r = (ax + b) \bmod p \qquad \text{and} \qquad s = (ay + b) \bmod p.$$

By Lemma 35.6.3, we have that $r \neq s$. As such, a collision happens if $r \equiv s \pmod{m}$. By Lemma 35.2.6, the number of such pairs (r, s) is at most $p(p-1)/m$. By Lemma 35.6.2, for each such pair (r, s), there is a unique choice of a, b that maps x and y to r and s, respectively. As such, there are at most $p(p-1)/m$ bad pairs. ■

Theorem 35.2.8. *The hash family $\mathcal{H}$ is a 2-universal hash family.*

Proof: Fix two distinct numbers $x, y \in \mathcal{U}$. We are interested in the probability they collide if h is picked randomly from $\mathcal{H}$. By Claim 35.2.7 there are $M \leq p(p-1)/m$ bad pairs that causes such a collision, and since $\mathcal{H}$ contains $N = p(p-1)$ functions, it follows the probability for collision is $M/N \leq 1/m$, which implies that $\mathcal{H}$ is 2-universal. ■

35.2.1.4. Explanation via pictures

Consider a pair $(x, y) \in \mathbb{Z}_p^2$, such that $x \neq y$. This pair (x, y) corresponds to a cell in the natural "grid" $\mathbb{Z}_p^2$ that is off the main diagonal. See Figure 35.2

The mapping $f_{a,b}(x) = (ax + b) \bmod p$, takes the pair (x, y), and maps it randomly and uniformly, to some other pair $x' = f_{a,b}(x)$ and $y' = f_{a,b}(y)$ (where x', y' are again off the main diagonal).

Now consider the smaller grid $\mathbb{Z}_m \times \mathbb{Z}_m$. The main diagonal of this subgrid is bad – it corresponds to a collision. One can think about the last step, of computing $h_{a,b}(x) = f_{a,b}(x) \bmod m$, as tiling the larger grid, by the smaller grid. in the natural way. Any diagonal that is in distance mi from the main diagonal get marked as bad. At most $1/m$ fraction of the off diagonal cells get marked as bad. See Figure 35.2.

As such, the random mapping of (x, y) to (x', y') causes a collision only if we map the pair to a badly marked pair, and the probability for that $\leq 1/m$.

35.3. Perfect hashing

An interesting special case of hashing is the static case – given a set S of elements, we want to hash S so that we can answer membership queries efficiently (i.e., dictionary data-structures with no insertions). it is easy to come up with a hashing scheme that is optimal as far as space.

35.3.1. Some easy calculations

The first observation is that if the hash table is quadraticly large, then there is a good (constant) probability to have no collisions (this is also the threshold for the birthday paradox).

Lemma 35.3.1. *Let $S \subseteq \mathcal{U}$ be a set of n elements, and let $\mathcal{H}$ be a 2-universal family of hash functions, into a table of size $m \geq n^2$. Then with probability $\leq 1/2$, there is a pair of elements of S that collide under a random hash function $h \in \mathcal{H}$.*

Proof: For a pair $x, y \in S$, the probability they collide is at most $\leq 1/m$, by definition. As such, by the union bound, the probability of any collusion is $\binom{n}{2}/m = n(n-1)/2m \leq 1/2$. ■

We now need a second moment bound on the sizes of the buckets.

Lemma 35.3.2. *Let $S \subseteq \mathcal{U}$ be a set of n elements, and let $\mathcal{H}$ be a 2-universal family of hash functions, into a table of size $m \geq cn$, where c is an arbitrary constant. Let $h \in \mathcal{H}$ be a random hash function, and let X_i be the number of elements of S mapped to the ith bucket by h, for $i = 0, \ldots, m-1$. Then, we have $\mathbb{E}\left[\sum_{j=0}^{m-1} X_j^2\right] \leq (1 + 1/c)n$.*

Proof: Let $s_1, \ldots, s_n$ be the n items in S, and let $Z_{i,j} = 1$ if $h(s_i) = h(s_j)$, for $i < j$. Observe that $\mathbb{E}\left[Z_{i,j}\right] = \mathbb{P}\left[h(s_i) = h(s_j)\right] \leq 1/m$ (this is the only place we use the property that $\mathcal{H}$ is 2-universal). In particular, let $\mathcal{Z}(\alpha)$ be all the variables $Z_{i,j}$, for $i < j$, such that $Z_{i,j} = 1$ and $h(s_i) = h(s_j) = \alpha$.

If for some α we have that $X_\alpha = k$, then there are k indices $\ell_1 < \ell_2 < \ldots < \ell_k$, such that $h(s_{\ell_1}) = \cdots = h(s_{\ell_k}) = i$. As such, $z(\alpha) = |\mathcal{Z}(\alpha)| = \binom{k}{2}$. In particular, we have

$$X_\alpha^2 = k^2 = 2\binom{k}{2} + k = 2z(\alpha) + X_\alpha$$

This implies that

$$\sum_{\alpha=0}^{m-1} X_\alpha^2 = \sum_{\alpha=0}^{m-1}\left(2z(\alpha) + X_\alpha\right) = 2\sum_{\alpha=0}^{m-1} z(\alpha) + \sum_{\alpha=0}^{m-1} X_\alpha = n + 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Z_{ij}$$

Now, by linearity of expectations, we have

$$\mathbb{E}\left[\sum_{\alpha=0}^{m-1} X_\alpha^2\right] = \mathbb{E}\left[n + 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Z_{ij}\right] = n + 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathbb{E}\left[Z_{ij}\right] \leq n + 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{m}$$
$$= n + \frac{2}{m}\binom{n}{2} = n + \frac{2n(n-1)}{2m} \leq n\left(1 + \frac{n-1}{m}\right) \leq n\left(1 + \frac{1}{c}\right)$$

since $m \geq cn$. ■

35.3.2. Construction of perfect hashing

Given a set S of n elements, we build a open hash table T of size, say, $2n$. We use a random hash function h that is 2-universal for this hash table, see Theorem 35.2.8. Next, we map the elements of S into the hash table. Let S_j be the list of all the elements of S mapped to the jth bucket, and let $X_j = |L_j|$, for $j = 0, \ldots, n-1$.

We compute $Y = \sum_{i=1} X_j^2$. If $Y > 6n$, then we reject h, and resample a hash function h. We repeat this process till success.

In the second stage, we build secondary hash tables for each bucket. Specifically, for $j = 0, \ldots, 2n-1$, if the jth bucket contains $X_j > 0$ elements, then we construct a secondary hash table H_j to store the elements of S_j, and this secondary hash table has size X_j^2, and again we use a random 2-universal hash function h_j for the hashing of S_j into H_j. If any pair of elements of S_j collide under h_j, then we resample the hash function h_j, and try again till success.

35.3.2.1. Analysis

Theorem 35.3.3. *Given a (static) set $S \subseteq \mathcal{U}$ of n elements, the above scheme, constructs, in expected linear time, a two level hash-table that can perform search queries in $O(1)$ time. The resulting data-structure uses $O(n)$ space.*

Proof: Given an element $x \in \mathcal{U}$, we first compute $j = h(x)$, and then $k = h_j(x)$, and we can check whether the element stored in the secondary hash table H_j at the entry k is indeed x. As such, the search time is $O(1)$.

The more interesting issue is the construction time. Let X_j be the number of elements mapped to the jth bucket, and let $Y = \sum_{i=1}^{n} X_i^2$. Observe, that $\mathbb{E}[Y] \leq (1 + 1/2)n = (3/2)n$, by Lemma 35.3.2 (here, $m = 2n$ and as such $c = 2$). As such, by Markov's inequality, $\mathbb{P}[X > 6n] = \frac{(3/2)n}{6n} \leq 1/4$. In particular, picking a good top level hash function requires in expectation at most $1/(3/4) = 4/3 \leq 2$ iterations. Thus the first stage takes $O(n)$ time, in expectation.

For the jth bucket, with X_j entries, by Lemma 35.3.1, the construction succeeds with probability $\geq 1/2$. As before, the expected number of iterations till success is at most 2. As such, the expected construction time of the secondary hash table for the jth bucket is $O(X_j^2)$.

We conclude that the overall expected construction time is $O(n + \sum_j X_j^2) = O(n)$.

As for the space used, observe that it is $O(n + \sum_j X_j^2) = O(n)$. ■

35.4. Bloom filters

Consider an application where we have a set $S \subseteq \mathcal{U}$ of n elements, and we want to be able to decide for a query $x \in \mathcal{U}$, whether or not $x \in S$. Naturally, we can use hashing. However, here we are interested in more efficient data-structure as far as space. We allow the data-structure to make a mistake (i.e., say that an element is in, when it is not in).

First try. So, let start silly. Let $B[0 \ldots, m]$ be an array of bits, and pick a random hash function $h : \mathcal{U} \to \mathbb{Z}_m$. Initialize B to 0. Next, for every element $s \in S$, set $B[h(s)]$ to 1. Now, given a query, return $B[h(x)]$ as an answer whether or not $x \in S$. Note, that B is an array of bits, and as such it can be bit-packed and stored efficiently.

For the sake of simplicity of exposition, assume that the hash functions picked is truly random. As such, we have that the probability for a false positive (i.e., a mistake) for a fixed $x \in \mathcal{U}$ is n/m. Since we want the size of the table m to be close to n, this is not satisfying.

Using k hash functions. Instead of using a single hash function, let us use k independent hash functions $h_1, \ldots h_k$. For an element $s \in S$, we set $B[h_i(s)]$ to 1, for $i = 1, \ldots, k$. Given an query $x \in \mathcal{U}$, if $B[h_i(x)]$ is zero, for any $i = 1, \ldots, k$, then $x \notin S$. Otherwise, if all these k bits are on, the data-structure returns that x is in S.

Clearly, if the data-structure returns that x is not in S, then it is correct. The data-structure might make a mistake (i.e., a false positive), if it returns that x is in S (when is not in S).

We interpret the storing of the elements of S in B, as an experiment of throwing kn balls into m bins. The probability of a bin to be empty is

$$p = p(m, n) = (1 - 1/m)^{kn} \approx \exp(-k(n/m)).$$

Since the number of empty bins is a martingale, we know the number of empty bins is strongly concentrated around the expectation pm, and we can treat p as the true probability of a bin to be empty.

The probability of a mistake is

$$f(k, m, n) = (1 - p)^k.$$

In particular, for $k = (m/n) \ln n$, we have that $p = p(m, n) \approx 1/2$, and $f(k, m, n) \approx 1/2^{(m/n) \ln 2} \approx 0.618^{m/n}$.

Example 35.4.1. Of course, the above is fictional, as k has to be an integer. But motivated by these calculations, let $m = 3n$, and $k = 4$. We get that $p(m, n) = \exp(-4/3) \approx 0.26359$, and $f(4, 3n, n) \approx (1 - 0.265)^4 \approx 0.294078$. This is better than the naive $k = 1$ scheme, where the probability of false positive is $1/3$.

Note, that this scheme gets exponentially better over the naive scheme as m/n grows.

Example 35.4.2. Consider the setting $m = 8n$ – this is when we allocate a byte for each element stored (the element of course might be significantly bigger). The above implies we should take $k = \lceil (m/n) \ln 2 \rceil = 6$. We then get $p(8n, n) = \exp(-6/8) \approx 0.5352$, and $f(6, 8n, n) \approx 0.0215$. Here, the naive scheme with $k = 1$, would give probability of false positive of $1/8 = 0.125$. So this is a significant improvement.

Remark 35.4.3. It is important to remember that Bloom filters are competing with direct hashing of the whole elements. Even if one allocates 8 bits per item, as in the example above, the space it uses is significantly smaller than regular hashing. A situation when such a Bloom filter makes sense is for a cache – we might want to decide if an element is in a slow external cache (say SSD drive). Retrieving item from the cache is slow, but not so slow we are not willing to have a small overhead because of false positives.

35.5. Bibliographical notes

Practical Issues Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement it for other objects, one needs to map objects in some fashion to integers.

- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Recent important paper bridging theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing

35.6. From previous lectures

Lemma 35.6.1. *Let p be a prime number.*

(A) For any $\alpha, \beta \in \{1, \ldots, p-1\}$, we have that $\alpha\beta \not\equiv 0 \pmod{p}$.
(B) For any $\alpha, \beta, i \in \{1, \ldots, p-1\}$, such that $\alpha \neq \beta$, we have that $\alpha i \not\equiv \beta i \pmod{p}$.
(C) For any $x \in \{1, \ldots, p-1\}$ there exists a unique y such that $xy \equiv 1 \pmod{p}$. The number y is the **inverse** *of x, and is denoted by x^{-1} or $1/x$.*

Lemma 35.6.2. *Consider a prime p, and any numbers $x, y \in \mathbb{Z}_p$. If $x \neq y$ then, for each pair of numbers $r, s \in \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, such that $r \neq s$, there is exactly one unique choice of numbers $a, b \in \mathbb{Z}_p$ such that $ax + b \pmod{p} = r$ and $ay + b \pmod{p} = s$.*

Lemma 35.6.3. *Consider a prime p, and any numbers $x, y \in \mathbb{Z}_p$. If $x \neq y$ then, for any $a, b \in \mathbb{Z}_p$, such that $a \neq 0$, we have $ax + b \not\equiv ay + b \pmod{p}$.*

References

[MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge, UK: Cambridge University Press, 1995.