15-451/651: Design & Analysis of Algorithms September 11, 2018
Lecture #5: Hashing last changed: August 30, 2018

Hashing is a great practical tool, with an interesting and subtle theory too. In addition to its use as a dictionary data structure, hashing also comes up in many different areas, including cryptography and complexity theory. In this lecture we describe two important notions: *universal hashing* (also known as *universal hash function families*) and *perfect hashing*.

Material covered in this lecture includes:

- The formal setting and general idea of hashing.
- Universal hashing.
- Perfect hashing.

1 Maintaining a Dictionary

While describing the desired properties, let us keep one application in mind. We want to maintain a dictionary. We have a large universe of "keys" (say the set of all strings of length at most 80 using the Roman alphabet), denoted by U. The actual dictionary (say the set of all English words) is some subset S of this universe. S is typically much smaller than U. The operations we want to implement are:

- `add(x)`: add the key x to S.
- `query(q)`: is the key $q \in S$?
- `delete(x)`: remove the key x from S.

In some cases, we don't care about adding and removing keys, we just care about fast query times—e.g., the actual English dictionary does not change (or changes very gradually). This is called the *static case*. Another special case is when we just add keys: the *insertion-only case*. The general case is called the *dynamic case*.

For the static problem we could use a sorted array with binary search for lookups. For the dynamic we could use a balanced search tree. However, hashing gives an alternative approach that is often the fastest and most convenient way to solve these problems. For example, suppose you are writing an AI-search program, and you want to store situations that you've already solved (board positions or elements of state-space) so that you don't redo the same computation when you encounter them again. Hashing provides a simple way of storing such information. There are also many other uses in cryptography, networks, complexity theory.

2 Hashing basics

The formal setup for hashing is as follows.

- Keys come from some large universe U. (E.g, think of U as the set of all strings of at most 80 ASCII characters.)

- There is some set S in U of keys we actually care about (which may be static or dynamic). Let $N = |S|$. Think of N as much smaller than the size of U. For instance, perhaps S is the set of names of students in this class, which is much smaller than 128^{80}.

- We will perform inserts and lookups by having an array A of some size M, and a **hash function** $h : U \to \{0, \ldots, M-1\}$. Given an element x, the idea of hashing is we want to store it in $A[h(x)]$. Note that if U was small (like 2-character strings) then you could just store x in $A[x]$ like in bucketsort. The problem is that U is big: that is why we need the hash function.
- We need a method for resolving collisions. A *collision* is when $h(x) = h(y)$ for two different keys x and y. For this lecture, we will handle collisions by having each entry in A be a linked list. There are a number of other methods, but for the issues we will be focusing on here, this is the cleanest. This method is called *separate chaining.* To insert an element, we just put it at the top of the list. If h is a good hash function, then our hope is that the lists will be small.

One great property of hashing is that all the dictionary operations are incredibly easy to implement. To perform a lookup of a key x, simply compute the index $i = h(x)$ and then walk down the list at $A[i]$ until you find it (or walk off the list). To insert, just place the new element at the top of its list. To delete, one simply has to perform a delete operation on the associated linked list. The question we now turn to is: what do we need for a hashing scheme to achieve good performance?

Desired properties: The main desired properties for a good hashing scheme are:

1. The keys are nicely spread out so that we do not have too many collisions, since collisions affect the time to perform lookups and deletes.
2. $M = O(N)$: in particular, we would like our scheme to achieve property (1) without needing the table size M to be much larger than the number of elements N.
3. The function h is fast to compute. In our analysis today we will be viewing the time to compute $h(x)$ as a constant. However, it is worth remembering in the back of our heads that h shouldn't be too complicated, because that affects the overall runtime.

Given this, the time to lookup an item x is $O(\text{length of list } A[h(x)])$. The same is true for deletes. Inserts take time $O(1)$ no matter what the lengths of the lists. So, we want to be able to analyze how big these lists get.

Basic intuition: One way to spread elements out nicely is to spread them *randomly.* Unfortunately, we can't just use a random number generator to decide where the next element goes because then we would never be able to find it again. So, we want h to be something "pseudorandom" in some formal sense.

We now present some bad news, and then some good news.

Claim 1 (Bad news) *For any hash function h, if $|U| \geq (N-1)M + 1$, there exists a set S of N elements that all hash to the same location.*

Proof: by the pigeon-hole principle. In particular, to consider the contrapositive, if every location had at most $N-1$ elements of U hashing to it, then U could have size at most $M(N-1)$. ■

So, this is partly why hashing seems so mysterious — how can one claim hashing is good if for any hash function you can come up with ways of foiling it? One answer is that there are a lot of simple hash functions that work well in practice for typical sets S. But what if we want to have a good *worst-case* guarantee?

2.1 A Key Idea

Here is a key idea: let's use randomization in our *construction* of h, in analogy to randomized quicksort. (The function h itself will be a deterministic function, of course). What we will show is that for *any* sequence of insert and lookup operations (we won't need to assume the set S of elements inserted is random), if we pick h in this probabilistic way, the performance of h on this sequence will be good in expectation. So, this is the same kind of guarantee as in randomized quicksort or treaps. In particular, this is idea of **universal hashing**.

Once we develop this idea, we will use it for an especially nice application called "perfect hashing".

3 Universal Hashing

Definition 2 *A randomized algorithm H for constructing hash functions $h : U \to \{0, \ldots, M-1\}$ is* **universal** *if for all $x \neq y$ in U, we have*

$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M. \qquad (1)$$

We also say that a set H of hash functions is a **universal hash function family** *if the procedure "choose $h \in H$ at random" is universal. (Here we are identifying the set of functions with the uniform distribution over the set.)*

Make sure you understand the definition! This condition must hold for *every pair* of distinct keys $x \neq y$, and the randomness is over the choice of the actual hash function h from the set H.

Here's an equivalent way of looking at this. First, count the number of hash functions in H that cause a and b to collide. This is

$$|\{h \in H \mid h(a) = h(b)\}|.$$

Divide this number by $|H|$, the number of hash functions. This is the probability on the left hand side of (1). So, to show universality you want

$$\frac{|\{h \in H \mid h(a) = h(b)\}|}{|H|} \leq \frac{1}{M}$$

for every $a \neq b \in U$.

Here are some examples and exercises to help you become comfortable with the definition.

Example 1: The following three hash families with hash functions mapping the set $\{a, b\}$ to $\{0, 1\}$ are universal, because at most $1/M$ of the hash functions in them cause a and b to collide, where $M = |\{0, 1\}|$.

	a	b
h_1	0	0
h_2	0	1

	a	b
h_1	0	1
h_2	1	0

	a	b
h_1	0	0
h_2	1	0
h_3	0	1

On the other hand, this next two hash families are not, since a and b collide with probability more than $1/M = 1/2$.

	a	b
h_1	0	0
h_3	1	1

	a	b	c
h_1	0	0	1
h_2	1	1	0
h_3	1	0	1

Here's another example of hash function family that *is* universal, with $M = |\{0, 1, 2\}|$:

	a	b	c	
h_0	0	0	0	$\leftarrow$ Note!
h_1	0	1	2	
h_2	1	2	0	
h_3	2	0	1	

Note that the universal family can have some bad hash functions in it (here h_0) so long as the family is large enough that the probability of picking those bad functions is small.

3.1 Using Universal Hashing

Theorem 3 *If H is universal, then for any set $S \subseteq U$ of size N, for any $x \in U$ (e.g., that we might want to lookup), if we construct h at random according to H, the* **expected** *number of collisions between x and other elements in S is at most N/M.*

Proof: Each $y \in S$ $(y \neq x)$ has at most a $1/M$ chance of colliding with x by the definition of "universal". So,

- Let the random variable $C_{xy} = 1$ if x and y collide and 0 otherwise.
- Let C_x be the r.v. denoting the total number of collisions for x. So, $C_x = \sum_{y \in S, y \neq x} C_{xy}$.
- We know $\mathbb{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M$.
- So, by linearity of expectation, $\mathbb{E}[C_x] = \sum_y \mathbb{E}[C_{xy}] < N/M$. ■

We now immediately get the following corollary.

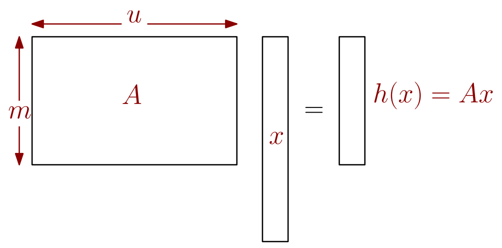
Corollary 4 *If H is universal then for any* **sequence** *of L insert, lookup, and delete operations in which there are at most M elements in the system at any one time, the expected total cost of the L operations for a random $h \in H$ is only $O(L)$ (viewing the time to compute h as constant).*

Proof: For any given operation in the sequence, its expected cost is constant by Theorem 3, so the expected total cost of the L operations is $O(L)$ by linearity of expectation. ■

3.2 Constructing a universal hash family: the matrix method

Can we actually construct a universal H? If not, this this is all pretty vacuous. Luckily, the answer is yes.

Let's say keys are u-bits long. Say the table size M is power of 2, so an index is m-bits long with $M = 2^m$. What we will do is pick A to be a random m-by-u 0/1 matrix, and define $h(x) = Ax$, where we do addition mod 2. These matrices are short and fat. For instance:



Claim 5 *For $x \neq y$, $\Pr_h[h(x) = h(y)] = 1/M = 1/2^m$.*

Proof: First of all, what does it mean to multiply A by x? We can think of it as adding some of the columns of A (doing vector addition mod 2) where the 1 bits in x indicate which ones to add.

Now, take an arbitrary pair of keys x, y such that $x \neq y$. They must differ someplace, so say they differ in the ith coordinate and for concreteness say $x_i = 0$ and $y_i = 1$. Imagine we first choose all of A but the ith column. Over the remaining choices of ith column, $h(x)$ is unchanged (since $x_i = 0$). However, each of the 2^m different settings of the ith column gives a different value of $h(y)$ (in particular, every time we flip a bit in that column, we flip the corresponding bit in $h(y)$). So there is exactly a $1/2^m$ chance that $h(x) = h(y)$. ■

There are other methods to construct universal hash families based on multiplication modulo primes as well (see Section 3.3).

One last note: there is a closely-related concept called an "ℓ-universal" or "ℓ-wise-independent" hash function. A hash function family is ℓ-universal if for every set of ℓ distinct keys $x_1, x_2, \ldots, x_\ell$ and every set of ℓ values $v_1, v_2, \ldots, v_\ell \in \{0, \ldots, M-1\}$, we have

$$\Pr_{h \leftarrow H}[h(x_1) = v_1 \text{ AND } h(x_2) = v_2 \text{ AND } \ldots \text{ AND } h(x_\ell) = v_\ell] = 1/M^\ell.$$

It is easy to see that if H is 2-universal then it is universal. Note that the above matrix-based hash family was not 2-universal since the hash functions all mapped 0 to 0.

Exercise 1: Show that if we choose $A \in \{0,1\}^{m \times u}$ and $b \in \{0,1\}^m$ independently and uniformly at random, then the hash family $h(x) = Ax + b$ is 2-universal.

3.3 Another method for universal hashing

Here is another method for constructing universal hash functions that is a bit more efficient than the matrix method given earlier.

In the matrix method, we viewed the key as a vector of bits. In this method, we will instead view the key x as a vector of integers $[x_1, x_2, \ldots, x_k]$ with the only requirement being that each x_i is in the range $\{0, 1, \ldots, M-1\}$. For example, if we are hashing strings of length k, then x_i could be the ith character (assuming our table size is at least 256) or the ith pair of characters (assuming our table size is at least 65536). Furthermore, we will require our table size M to be a prime number. To select a hash function h we choose k random numbers $r_1, r_2, \ldots, r_k$ from $\{0, 1, \ldots, M-1\}$ and define:

$$h(x) = r_1 x_1 + r_2 x_2 + \ldots + r_k x_k \mod M.$$

The proof that this method is universal follows the exact same lines as the proof for the matrix method. Let x and y be two distinct keys. We want to show that $\Pr_h[h(x) = h(y)] \leq 1/M$. Since

$x \neq y$, it must be the case that there exists some index i such that $x_i \neq y_i$. Now imagine choosing all the random numbers r_j for $j \neq i$ first. Let $h'(x) = \sum_{j \neq i} r_j x_j$. So, once we pick r_i we will have $h(x) = h'(x) + r_i x_i$. This means that we have a collision between x and y exactly when $h'(x) + r_i x_i = h'(y) + r_i y_i \mod M$, or equivalently when

$$r_i(x_i - y_i) = h'(y) - h'(x) \mod M.$$

Since M is prime, division by a non-zero value mod M is legal (every integer between 1 and $M-1$ has a multiplicative inverse modulo M), which means there is exactly one value of r_i modulo M for which the above equation holds true, namely $r_i = (h'(y) - h'(x))/(x_i - y_i) \mod M$. So, the probability of this occuring is exactly $1/M$.

4 Perfect Hashing

The next question we consider is: if we fix the set S (the dictionary), can we find a hash function h such that *all* lookups are constant time (not in expectation)? The answer is *yes*, and this leads to the topic of *perfect hashing*. We say a hash function is **perfect** for S if all lookups involve $O(1)$ work. Here are now two methods for constructing perfect hash functions for a given set S.

4.1 Method 1: an $O(N^2)$-space solution

Say we are willing to have a table whose size is quadratic in the size N of our dictionary S. Then, here is an easy method for constructing a perfect hash function. Let H be universal and $M = N^2$. Then just pick a random h from H and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim 6 *If H is universal and $M = N^2$, then* $\Pr_{h \leftarrow H}(\textit{no collisions in } S) \geq 1/2$.

Proof:

- How many pairs (x, y) in S are there? **Answer:** $\binom{N}{2}$
- For each pair, the chance they collide is $\leq 1/M$ by definition of "universal".
- So, Pr(exists a collision) $\leq \binom{N}{2}/M < 1/2$. ■

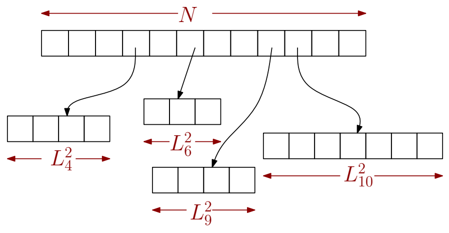
This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from H, and if we got any collisions, we just pick a new h. On average, we will only need to do this twice. Now, what if we want to use just $O(N)$ space?

4.2 Method 2: an $O(N)$-space solution

The question of whether one could achieve perfect hashing in $O(N)$ space was a big open question for some time, posed as "should tables be sorted?" That is, for a fixed set, can you get constant lookup time with only linear space? There was a series of more and more complicated attempts, until finally it was solved using the nice idea of universal hash functions in 2-level scheme.

The method is as follows. We will first hash into a table of size N using universal hashing. This will produce some collisions (unless we are extraordinarily lucky). However, we will then rehash each bin using Method 1, squaring the size of the bin to get zero collisions. So, the way to think of this scheme is that we have a first-level hash function h and first-level table A, and then N second-level hash functions $h_1, \ldots, h_N$ and N second-level tables $A_1, \ldots, A_N$. To lookup an element x, we first compute $i = h(x)$ and then find the element in $A_i[h_i(x)]$. (If you were doing this in practice, you might set a flag so that you only do the second step if there actually were collisions at index i, and otherwise just put x itself into $A[i]$, but let's not worry about that here.)



Say hash function h hashes L_i elements of S to location i. We already argued (in analyzing Method 1) that we can find $h_1, \ldots, h_N$ so that the total space used in the secondary tables is $\sum_i (L_i)^2$. What remains is to show that we can find a first-level function h such that $\sum_i (L_i)^2 = O(N)$. In fact, we will show the following:

Theorem 7 *If we pick the initial h from a universal set H, then*

$$\Pr\left[\sum_i (L_i)^2 > 4N\right] < 1/2.$$

Proof: We will prove this by showing that $\mathbb{E}[\sum_i (L_i)^2] < 2N$. This implies what we want by Markov's inequality. (If there was even a 1/2 chance that the sum could be larger than $4N$ then that fact by itself would imply that the expectation had to be larger than $2N$. So, if the expectation is less than $2N$, the failure probability must be less than 1/2.)

Now, the neat trick is that one way to count this quantity is to count the number of ordered pairs that collide, including an element colliding with itself. E.g, if a bucket has {`d,e,f`}, then `d` collides with each of {`d,e,f`}, `e` collides with each of {`d,e,f`}, and `f` collides with each of {`d,e,f`}, so we get 9. So, we have:

$$\begin{aligned}
\mathbb{E}\left[\sum_i (L_i)^2\right] &= \mathbb{E}\left[\sum_x \sum_y C_{xy}\right] \qquad (C_{xy} = 1 \text{ if } x \text{ and } y \text{ collide, else } C_{xy} = 0) \\
&= N + \sum_x \sum_{y \neq x} \mathbb{E}[C_{xy}] \\
&\leq N + N(N-1)/M \qquad (\text{where the } 1/M \text{ comes from the definition of universal}) \\
&< 2N. \qquad (\text{since } M = N) \blacksquare
\end{aligned}$$

So, we simply try random h from H until we find one such that $\sum_i L_i^2 < 4N$, and then fixing that function h we find the N secondary hash functions $h_1, \ldots, h_N$ as in method 1.