15-451/651: Design & Analysis of Algorithms September 13, 2022
Lecture #5: Hashing last changed: September 12, 2022

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* and *perfect hashing*.

Objectives of this lecture

In this lecture, we want to:

- Understand the formal definition and general idea of hashing
- Define and analyze *universal hashing* and its properties
- Analyze an algorithm for *perfect hashing*

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 11, Hash Tables
- DPV, *Algorithms*, Chapter 1.5, Universal 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** $h : U \to \{0, \dots, M-1\}$. 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. (This is called "separate chaining".) 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 the lengths of the lists. So, we want to be able to analyze how big these lists get.

Cost model: When analyzing hash functions, we will assume that standard arithmetic operations on integers from our universe are constant time. When analyzing algorithms that use hashing, we will assume that we can compute the hash function in constant time (even though not all hash functions are computable in constant time, including some in this lecture).

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: 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 pigeonhole 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

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. We can come up with different kinds of hashing schemes depending on what we mean by "good" in expectation. Essentially, the goal is to make the hash appear *as if it was a totally random function*, even though it isn't.

We will first develop the idea of *universal hashing.* Then, we will use it for an especially nice application called "perfect hashing".

3 Universal Hashing

Definition: Universal Hashing

A set of hash functions $\mathcal{H}$ where each $h \in \mathcal{H}$ maps $U \to \{0, \ldots, M-1\}$ is called **universal** (or is called a *universal family*) if for all $x \neq y$ in U, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq 1/M. \tag{1}$$

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 $\mathcal{H}$. Here's an equivalent way of looking at this. First, count the number of hash functions in $\mathcal{H}$ that cause x and y to collide. This is

$$|\{h \in \mathcal{H} | h(x) = h(y)\}|.$$

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

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

for every $x \neq y \in U$. Here are some examples to help you become comfortable with the definition.

Example

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, were $M = |\{0, 1\}|$.

	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, these 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

3.1 Using Universal Hashing

Theorem 1: Universal hashing

If $\mathcal{H}$ 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 h is drawn randomly from $\mathcal{H}$, the **expected** number of collisions between x and other elements in S is less than 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 random variable denoting the total number of collisions for x. So,

$$C_x = \sum_{\substack{y \in S \\ y \neq x}} C_{xy}.$$

- We know $\mathbb{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M$.
- So, by linearity of expectation,

$$\mathbb{E}[C_x] = \sum_{\substack{y \in S \\ y \neq x}} \mathbb{E}[C_{xy}] \leq \frac{|S|-1}{M} = \frac{N-1}{M},$$

which is less than N/M.

□

We now immediately get the following corollary.

Corollary

If $\mathcal{H}$ is universal then for any **sequence** of L insert, lookup, and delete operations in which there are at most M keys in the data structure at any one time, the expected total cost of the L operations for a random $h \in \mathcal{H}$ 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 1, so the expected total cost of the L operations is $O(L)$ by linearity of expectation. □

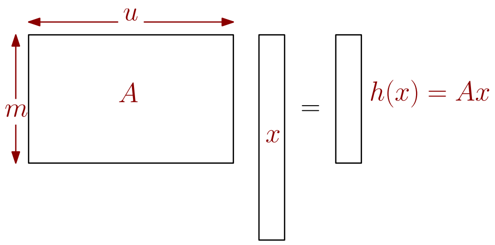
Can we actually construct a universal $\mathcal{H}$? If not, this this is all pretty vacuous. Luckily, the answer is yes.

3.2 Constructing a universal hash family: the matrix method

Definition: The matrix method for universal hashing

Let's say keys are u-bits long, so $u = \lceil \log |U| \rceil$. We require that the table size M is a power of 2, so an index is m-bits long with $M = 2^m$. We pick a random m-by-u 0/1 matrix A, and define $h(x) = Ax$, where we do addition mod 2. Here, x is interpreted as a 0/1 vector of length u, and $h(x)$ is a 0/1 vector of length m, denoting the bits of the result.

These matrices are short and fat. For instance:



Claim: The matrix method is universal

Let $\mathcal{H}$ be the hash family generated by the matrix method. For all $x \neq y$ from U, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{2^m} = \frac{1}{M}$$

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. E.g., if $x = (1010\cdots)^\intercal$, Ax is the sum of the 1st and 3rd columns of A.

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 the entries of A but those in the ith column. Over the remaining choices of ith column, $h(x) = Ax$ is fixed, since $x_i = 0$ and so Ax does not depend on the ith column of A. 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 $h(x) = h(y)$.

More verbosely, let $y' = y$ but with the ith entry set to zero. So $Ay = Ay' +$ the ith column of A. Now Ay' is also fixed now since $y'_i = 0$. Now if we choose the entries of the ith column of A, we get $Ax = Ay$ exactly when the ith column of A equal $A(x - y')$, which has been fixed by the choices of all-but-the-ith-column. Each of the m random bits in this ith column must come out right, which happens with probability $(1/2)$ each. These are independent choices, so we get probability $(1/2)^m$. □

Okay great, so its universal! But how efficient is it? Well, if we manually compute the matrix product, since it is an $m \times u$ matrix, this will take us $O(mu) = O(\lg |U| \lg M)$ time, which is not amazing, since this is actually worse than using a binary search tree. However, this is assuming that we compute the result bit-by-bit. If we take advantage of the fact that the key and rows are $\lg |U|$-bit integers, we can compute each row-vector product in constant time, and improve the performance to $O(\lg M)$ time, which is about the same as a balanced binary search tree.

3.3 Another method for universal hashing: the dot-product method

Definition: The dot-product method for universal hashing

We will first require M to be a prime number. 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 requirement being that each x_i is in the range $\{0, 1, \ldots, M-1\}$ and hence $k = \log_M |U|$. There is a very natural interpretation of this. Just think of the key being written in base M.

To select a hash function, we choose k random numbers $r_1, r_2, \ldots, r_k$ in $\{0, 1, \ldots, M-1\}$ and define:

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

Note that choosing $[r_1, r_2, \ldots, r_k]$ here is equivalent to just picking a single value r in the universe and then writing it in base M as well. The proof that this method is universal follows the exact same lines as the proof for the matrix method.

Claim: The dot-product method is universal

Let $\mathcal{H}$ be the hash family generated by the dot-product method. For all $x \neq y$ from U, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{M}$$

Proof. 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 occurring is exactly $1/M$. □

Is this more efficient than the matrix method? Well, our keys consist of k elements, where $k = \log_M |U|$, so computing the hash function takes $O(k) = O(\log_M |U|)$ time, which is faster than the matrix method when M is large, but slower when M is small.

4 More powerful hash families

Recall that our overarching goal with universal hashing was to produce a hash function that behaved *as if it was totally random.* We can try to be more specific about what we mean. In the case of universal hashing, if we took any two distinct keys x, y from our universe, and then hashed them using our hash function from a universal family, then the probability of collision was at most $1/M$, which is the probability that we would get if the hash function was totally random! We can therefore think of universal hashing as hashing that appears to behave totally randomly if all we care about is pairwise collisions.

In some cases (for some algorithms), though, this is not good enough. Although universal hashing looks good if all we care about are collisions, there are scenarios where universal hashes appear totally not random. Lets consider an example. Suppose we are maintaining a hash table of size $M = 2$, and an evil adversary would like to cause a collision by inserting just two items. If our hash was totally random, then the adversary would have a 50/50 chance of success just by pure chance. Suppose that we use the following universal family for our hash table.

	a	b	c
h_1	0	0	1
h_2	1	0	1

In this case, the evil adversary can just first insert a, and now we are in trouble. If a goes into slot 0, then the adversary knows we have h_1 and can hence select b to insert next, causing a guaranteed collision. Otherwise, if a goes into slot 1, then the adversary can select c and cause a guaranteed collision. So, even though we used a universal hash family, it wasn't as good as a totally random hash, because the adversary was able to figure out which hash function had been selected by just knowing the hash of one element. The problem at a high level was that although this family makes collisions unlikely, it doesn't do anything to prevent the hashes of different elements from correlating. In this family, the adversary can deduce the hash values of b and c by just knowing the hash of a.

To fix this, there is a closely-related concept called pairwise independence, or "2-universality".

Definition: Pairwise independence

A hash family $\mathcal{H}$ is *pairwise independent* (a.k.a. 2-universal) if for all pairs of distinct keys $x_1, x_2 \in U$ and every pair of values $v_1, v_2 \in \{0, \ldots, M-1\}$, we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2] = \frac{1}{M^2}$$

Intuitively, pairwise independence guarantees that if we only ever look at pairs of keys in our universe, then their hash values appear to behave totally randomly! In other words, if the adversary ever learns the hash value of one key, it can not deduce any information about the hash values of the other keys, they appear totally random. Of course, it is possible that by learning the hash values of *two* elements, the adversary may be able to deduce information about other elements. To improve this, we can generalize the definition of pairwise independence to arbitrary-size sets of keys.

Definition: ℓ-wise independence

A hash family $\mathcal{H}$ is *ℓ-wise independent* (a.k.a. ℓ-universal) if for all 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

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2 \text{ and } \ldots \text{ and } h(x_\ell) = v_\ell] = \frac{1}{M^\ell}$$

Intuitively, if a hash family is ℓ-wise independent, then the hash values of sets of ℓ elements appear totally random, or, if an adversary learns the hash values of $\ell - 1$ elements, it can not deduce any information about the hash values of any other elements.

Exercise

Show that any pairwise independent (2-universal) hash family is also a universal hash family.

Exercise

Show that the matrix method as defined above, which was universal, is **not** pairwise independent.

5 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? 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 $O(1)$ deterministic work-case cost (though lookup must be deterministic, randomization is still needed to actually construct the hash function). Here are now two methods for constructing perfect hash functions for a given set S.

5.1 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 $\mathcal{H}$ be universal and $M = N^2$. Then just pick a random h from $\mathcal{H}$ and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim

If $\mathcal{H}$ is universal and $M = N^2$, then

$$\Pr_{h \in \mathcal{H}}(\text{no collisions in } S) \geq 1/2.$$

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. Therefore,

$$\Pr(\text{exists a collision}) \leq \frac{\binom{N}{2}}{M} = \frac{N(N-1)}{2M} \leq \frac{N^2}{2N^2} = \frac{1}{2}$$

□

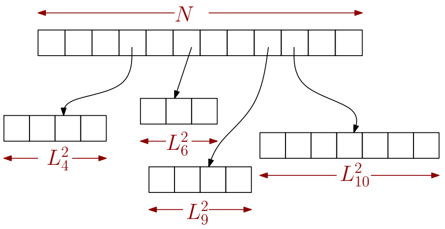
This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from $\mathcal{H}$, 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?

5.2 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 a 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

If we pick the initial h from a universal family $\mathcal{H}$, then

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

Proof. We will prove this by showing that $\mathbb{E}\left[\sum_i (L_i)^2\right] < 2N$. 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:

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

□

So, we simply try random h from $\mathcal{H}$ 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.