Advanced Algorithms

Lecture 14: Hashing

Lecturer: Nathan Klein

1 Hashing

Hashing is a common and powerful primitive. In this lecture, we will overview the basics of hashing and the math that underlies it.

1.1 Introduction to Hashing

Hashing is a way to compress address spaces that are very large. Say you want to build a lookup table for the number of people that have each name. For concreteness let's say we're focused on names consisting of at most 30 English characters.

1. One option is to build a long list of all possible names and index them $0, 1, \dots, n$ in alphabetical order where n is the number of possible names. Then upon seeing each name you can compute the index and then increment the counter at that index. How much space would that take up? Well, there are $27^{30} \approx 2^{140}$ possible names in that list, so, 2^{140} bits at least. That is a huge number: way more than the amount of storage that exists on earth.

2. Another option is to build a binary search tree to allow us to do lookups and increments in $O(\log n)$ time. This is not a bad idea for this example. But, in some cases it's important for the lookup to be very fast.

Hashing is a way to make option 1 practical by "renaming" the address space. To be concrete, suppose we have an address space consisting of 140 bits, but only expect around a billion items, around 2^{30}. Storing 2^{30} items is reasonable: on the order of Gigabytes, while storing 2^{140} items would be mind-bogglingly impossible, approaching the number of atoms on earth.

What's one way to do this? Well: suppose we had a function h that mapped items in our original space (call it U) to the space of 30 bits, so $h : U \to \{0,1\}^{30}$ or equivalently $h : U \to \{0, 1, \dots, 2^{30} - 1\}$. Furthermore, suppose this function *magically* has the property that in our stream of 2^{30} inserts, $h(x) \neq h(y)$ for all insertions x, y with $x \neq y$. Then we could just have a list of size 2^{30} initialized at 0. Upon seeing some x in our original space, we can set $h(x) = h(x) + 1$.

Of course, that's too much to ask for. Inevitably, whatever function you build (which doesn't know all the 2^{30} inserts you're going to make) will have **collisions**: places where $h(x) = h(y)$ even though $x \neq y$. And now hopefully you see the dream: first, let's find hash functions with few collisions. Then, let's find a way to handle the collisions when they do occur.

1. To minimize collisions, we want h to basically map $x \in U$ to a random number in the desired range, here $0, \dots, 2^{30} - 1$. Since h is in fact deterministic, that's not possible. Instead, we want to somehow take x, chop it up, and mix it around in some crazy but deterministic way to get $h(x)$. "To hash" means "to chop into small pieces," and it also means something like "hodgepodge." And that's why (as far as I can tell, anyway) we call h a hash function.

2. To deal with collisions, we will consider two ideas. The first idea is to do nothing. At location i, simply store all items $x \in U$ you see where $h(x) = i$. The second idea is to do a 2-layer hash.

In this lecture we will study building a hash table which supports the basic operations: insertions, deletions, and lookups. The first hash table will allocate k bits of storage in an indexed list and use a hash function $h : U \to \{0, 1, \ldots, k-1\}$. It will store all elements $x \in U$ that appear at index $h(x)$ in a linked list, i.e., it will not do anything interesting with collisions.

So it will be important to understand how long these linked lists are. If our *input* is random, we may be able to do some sort of average case analysis. But that's often not reasonable, like in our running example of storing a list of names. The name "vskjhgskfhj lsjfl" is probably never going to appear. Instead, we will build random hash functions. These will work with high probability against inputs which are arbitrary (so, worst-case) but do not know the random coins used by our hash function. If an adversary knew our hash function exactly, we would be doomed to one big linked list: they would just keep sending all the elements in U which map to a single entry. So it's very important that we do not need to work against adversaries that know our random coins.

1.2 Word RAM Model

Hold on a second. If we expect to see n elements, then we will definitely need at least $\Omega(\log n)$ buckets to prevent tons of collisions. That means our hash function has to *output* about $\log n$ bits. But then isn't its runtime $\Omega(\log n)$: no better than a binary search tree?

The very quick answer is that modern computers can do operations on numbers with 64 bits (or 32 bits, depending on the "word size" of the computer) in just one operation. This is the word RAM model of computation. So if we plan for n to be at most 2^{64}, which is almost surely the case, then we might be able to produce a hash of an input using just a few operations on a number with 64 bits: $O(1)$ time.

When we actually construct the hash functions, we'll check back in about this.

1.3 Random Hash Functions

We want to understand the *expected* number of collisions to get a handle on how long these linked lists will be. Since any particular hash function is deterministic, the only way to do this analysis is to choose a random hash function from a *family* of possible ones.

Let $[k] = \{0, \ldots, k-1\}$ be the range of our hash function. A natural thing to want is that for any $x \in U$ and any $a \in [k]$, $\mathbb{P}[h(x) = a] = \frac{1}{k}$. In other words, any particular element in our universe should map to each element in the domain with equal probability. This is super easy to achieve and unfortunately utterly useless for avoiding collisions. Do you see how to pick a trivial family of functions with this property?

Let $h_0, \ldots, h_{k-1}$ be a set of hash functions where $h_i(x) = i$ for all $i \in [k]$ and all $x \in U$. Then, we will pick $h = h_i$ uniformly at random among all $i \in [k]$. This satisfies this property.

But, the following property makes things interesting: for all $x, y \in U$, we would like that:

$$\mathbb{P}[h(x) = h(y)] \leq \frac{1}{k}$$

This property is called the **2-universal** property. It is quite reasonable to expect as it's implied by pairwise independence. One can attempt achieving even higher order independence, and there are applications for k-wise independent hashing. However, as k grows, the hash function itself becomes more and more expensive, which defeats the purpose. You could, for example, store fresh random bits for every possible element in U. But then what was the point of the hash table? We would be using storage $|U|$ again.

Lemma 1.1. *Let h be selected from a 2-universal family of hash functions. Then the expected number of collisions after n insertions is $\binom{n}{2}\frac{1}{k}$.*

Proof. Let $I \subseteq U$ be our set of unique insertions (having the same element twice will not change the number of collisions on differing elements) with $|I| = n$. For all $x, y \in I$, let C_{xy} indicate a collision, i.e., the event that for our randomly selected h we have $h(x) = h(y)$. So, the number of collisions is $C = \sum_{x,y \in I, x \neq y} C_{xy}$. Note that this linked lists with r elements generate $\binom{r}{2} \approx r^2$ collisions. Using the 2-universal property, we can compute:

$$\mathbb{E}\left[C_{xy}\right] = \mathbb{P}\left[h(x) = h(y)\right] \leq \frac{1}{k}$$

Note that in our bad hash family example, we had $C_{xy} = 1$ with probability 1, so this 2-universal property is crucial. Now, using linearity of expectation:

$$\mathbb{E}\left[C\right] = \sum_{x \neq y} \mathbb{E}\left[C_{xy}\right] = \sum_{x \neq y} \frac{1}{k} = \binom{n}{2}\frac{1}{k}$$ □

A nice corollary is the following:

Corollary 1.2. *Given a 2-universal family of hash functions where we plan to insert at most n elements, if we choose $k = 50n^2$, we get 0 collisions with probability 99%.*

Proof. The expected number of collisions is at most:

$$\binom{n}{2}\frac{1}{10n^2} \leq \frac{n^2}{2} \cdot \frac{1}{50n^2} = \frac{1}{100}$$ □

This is already something interesting, although it is quite a large space overhead. In our example with 2^{30} elements, this makes our overhead around 2^{60}, or on the order of an exabyte. This is within human reach, but it's also completely impractical, while for smaller insertion sizes, say 2^{15}, this isn't so bad. It turns out we can do much better and reduce the space requirement to $O(n)$ by chaining hash functions. But first let's show how to design a 2-universal hash family.

1.4 2-Universal Hash Families

Constructing a 2-universal hash family is not difficult, it turns out. Pick a prime p with $|U| \leq p \leq 2|U|$, which always exists by something called Bertrand's postulate. We can find this with $O(\frac{1}{\log|U|})$ operations in expectation by generating random numbers in this range and testing if they're prime.[1] Given this prime p, we will pick random numbers $a, b \in [p]$ with $a \neq 0$ and let:

$$f_{a,b}(x) = ax + b \pmod{p}$$

[1]This is not hard to do. See the (silly) prime number website I run which can easily generate random primes with 300 digits.

If x is a string or something, just look at its bit representation to find its numeric value to which you'll apply the hash. Then our hash function $h : U \to [k]$ will be:

$$h_{a,b}(x) = f_{a,b} \pmod{k}$$

This simple idea turns out to yield a 2-universal hash family. Better yet, it takes constant time in the word RAM model since it's just multiplying two word-sized integers and adding another. So: let's prove that it's 2-universal. First, we need a lemma:

Lemma 1.3. *Let $x \neq y \in U$ and let $r_1, r_2 \in [p]$ with $r_1 \neq r_2$. Then, there is exactly one set of $a, b \in [p]$ so that both of the following equations hold:*

$$ax + b = r_1 \pmod{p}, \qquad ay + b = r_2 \pmod{p}$$

Proof. We will solve for the values of a, b, showing that they are unique. Subtracting the second equation from the first yields:

$$\begin{aligned} a(x-y) &= r_1 - r_2 \pmod{p} \\ a &= (r_1 - r_2)(x-y)^{-1} \pmod{p} \end{aligned}$$

where we use that since p is prime, the set of integers mod p is a finite field and therefore the non-zero elements have multiplicative inverses (and $x - y \neq 0$ since $x \neq y$). So, a is fixed. But since a is fixed, b must be fixed too, as $ax + b = r_1 \pmod{p}$, so $b = r_1 - ax \pmod{p}$. □

Now we can finish the proof:

Lemma 1.4. *Fix a prime p and a domain k. Let a be picked uniformly at random in the range $1, \ldots, p-1$ and b be picked uniformly at random from $0, \ldots, p-1$. Then, the resulting distribution over hash functions $h_{a,b} = (ax + b \pmod{p}) \pmod{n}$ is 2-universal.*

Proof. Let $x, y \in U$ with $x \neq y$. First, notice that for any choice of $a, b \in [p]$ (with $a \neq 0$) we have $f_{a,b}(x) \neq f_{a,b}(y)$ for $x \neq y$. To see this, suppose not: then $ax + b \neq ay + b \pmod{p}$. But now this implies $ax = ay \pmod{p}$, and after multiplying by a^{-1} this implies $x = y$.

Therefore, the only way we get a collision is if $f_{a,b}(x) \neq f_{a,b}(y)$ but $f_{a,b}(x) = f_{a,b}(y) \pmod{k}$. Writing this another way, it must be that for some $r_1, r_2 \in [p]$ with $r_1 \neq r_2$ and $r_1 = r_2 \pmod{k}$, we have $ax + b = r_1 \pmod{p}$, $ay + b = r_2 \pmod{p}$.

But for *each* such pair r_1, r_2 there is only *one pair* a, b that can be used to achieve $ax + b = r_1 \pmod{p}$ and $ay + b = r_2 \pmod{p}$: that's what Lemma 1.3 says! The probability of picking that particular pair a, b is just $\frac{1}{(p-1) \cdot p}$ because that's the chance we pick the hash function $h_{a,b}$.

How many pairs r_1, r_2 are there, then? For each $r_1 \in [p]$, there are at most $\frac{p}{k} - 1 \leq \frac{p-1}{k}$ choices for r_2 (since we exclude r_1). So:

$$\mathbb{P}\left[h_{a,b}(x) = h_{a,b}(y)\right] \leq (p \cdot \frac{p-1}{k}) \cdot \frac{1}{(p-1)p} = \frac{1}{k}$$

where the first parenthesis is the number of choices for r_1, r_2 and the second is the chance the hash functions pick that particular a, b. Therefore, this family is 2-universal. □

1.5 2-Layer Hashing, or Perfect Hashing

We already said that to get a collision-free ("perfect") hash table with probability 99%, it suffices to choose k to be about n^2, where n is the number of inserted elements. To get to $k \in O(n)$, we need a 2-layer hash table. We will assume for the moment that we're only interested in building a *static* hash table with $O(1)$ look-up time, but there are workarounds for allowing this to happen dynamically as well: see [Die+94].

The idea is simple and due to [FKS84]. First, hash everything, collisions and all. Then for each bucket $b \in [k]$ in the hash table, if there are b_i elements colliding there, we will create a new hash function with $O(b_i^2)$ buckets. As already discussed, we are guaranteed we can find a hash function now that has no collisions over these b_i elements (choosing a random one works with probability 99%).

So the size of our new table is going to be k, our original choice, plus $\sum_{i\in[k]} O(b_i^2)$. But in fact this is the same order as the number of collisions, which is $\sum_{i\in[k]} \binom{b_i}{2}$.

And what's the expected number of collisions? Well we computed this. For n insertions it's $\binom{n}{2}\frac{1}{k} \approx \frac{n^2}{2k}$. Therefore, the expected size of our hash table is going to be $k + O(\frac{n^2}{2k})$, and choosing $k = n$ gives us a hash table of size $O(n)$ **with no collisions**, as desired.

References

[Die+94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. "Dynamic Perfect Hashing: Upper and Lower Bounds". In: *SIAM Journal on Computing* 23.4 (1994), pp. 738–761. DOI: 10.1137/S0097539791194094. eprint: https://doi.org/10.1137/S0097539791194094 (cit. on p. 5).

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. "Storing a Sparse Table with 0(1) Worst Case Access Time". In: *J. ACM* 31.3 (June 1984), 538–544. ISSN: 0004-5411. DOI: 10.1145/828.1884 (cit. on p. 5).