Lecture 10

Universal and Perfect Hashing

10.1 Overview

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.

10.2 Introduction

We will be looking at the basic dictionary problem we have been discussing so far and will consider two versions, static and dynamic:

- Static: Given a set S of items, we want to store them so that we can do lookups quickly. E.g., a fixed dictionary.
- Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests. We want to do these all efficiently.

For the first problem we could use a sorted array with binary search for lookups. For the second 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.

10.3 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 purposes of 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 hopefully the lists will be small.

Nice properties of hashing. One nice 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. To insert, just place the new element at the top of its list, and to delete, one simply has to perform a delete operation on the linked list.

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. 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 something crazy.

Given this, the time to lookup an item x is $O(\text{length of list at } h(x))$. 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 things out nicely is to spread them *randomly.* Unfortunately, we can't just use a random number generator to decide where the next thing goes because then we would never be able to find it again. So, we want h to be something "pseudorandom" in some sense.

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

Claim 10.1 (Bad news) *For any hash function h, if $|U| \geq (N-1)M+1$, there exists a set S of size N that all hash to the same location.*

Proof: by the pigeon-hole principle. ∎

So, this is partly why hashing seems so mysterious — how can you 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. But what if we want to have a guarantee for *every* set S?

Here is a nice idea: let's use randomization in analogy to randomized quicksort. In particular, we will use some random numbers in our *construction* of h. (h itself will be a deterministic function, of course). What we will show is that for *any* set S (we won't need to assume S is random), if we pick h in this random way, things will be good in expectation. So, this is the same kind of guarantee as in randomized quicksort. This is idea of **universal hashing**.

Once we develop this idea, we will use it for a really nice practical application called "perfect hashing".

10.4 Universal Hashing

Definition 10.1 *A probability distribution H over hash functions from U to $\{1, \ldots, M\}$ is* **universal** *if for all $x \neq y$ in U, we have*

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

Theorem 10.2 *If H is universal, then for any set $S \subseteq U$, for any $x \in U$ (that we might want to insert or lookup), for a random h taken from 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 $C_{xy} = 1$ if x and y collide and 0 otherwise.
- Let C_x denote the total number of collisions for x. So, $C_x = \sum_{y \in S, y \neq x} C_{xy}$.
- We know $\mathbf{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M$.
- So, by linearity of expectation, $\mathbf{E}[C_x] = \sum_y \mathbf{E}[C_{xy}] < N/M$. ∎

As a corollary, we immediately get that the expected time for any given lookup is $O(1 + N/M)$, since the lookup time is proportional to the number of collisions plus the time to compute h (which we are viewing as constant). So, overall, this is constant time if $M = N$.

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

Terminology: If H is a uniform distribution over a set of hash functions $\{h_1, h_2, \ldots\}$, then that set is called a **universal hash family**. We will use "H" for both the *set* and the *probability distribution*. Either way, we think of H as a probabilistic way of constructing a hash function.

10.4.1 Constructing a universal hash family: the matrix method

Let's say keys are u-bits long. Say the table size M is power of 2, so an index is b-bits long with $M = 2^b$.

What we'll do is pick h to be a random b-by-u 0/1 matrix, and define $h(x) = hx$, where we do addition mod 2. These matrices are short and fat. For instance:

$$\begin{array}{ccc} \text{h} & \text{x} & \text{h(x)} \\ \boxed{\begin{matrix}1&0&0&0\\0&1&1&1\\1&1&1&0\end{matrix}} & \boxed{\begin{matrix}1\\0\\1\\0\end{matrix}} = & \boxed{\begin{matrix}1\\1\\0\end{matrix}} \end{array}$$

Claim 10.3 *For $x \neq y$,* $\Pr[h(x) = h(y)] = 1/M = 1/2^b$.

Proof: First of all, what does it mean to multiply h by x? We can think of it as adding some of the columns of h (mod 2) where the 1 bits in x tell you which ones to add. (e.g., we added the 1st and 3rd columns of h above)

Now, take arbitrary 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 h but the ith column. Over the remaining choices of ith column, $h(x)$ is fixed. But, each of the 2^b 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 $1/2^b$ chance that $h(x) = h(y)$. ■

There are other methods to construct universal hash families based on multiplication modulo primes as well.

The next question we consider is: if we fix the set S, can we find a hash function h such that *all* lookups are constant-time? The answer is *yes*, and this leads to the next topic of *perfect hashing*.

10.5 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.

10.5.1 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. Let H be universal and $M = N^2$. Pick a random h from H and try it out, hashing everything in S.

Claim 10.4 $\Pr($*no collisions*$) \geq 1/2$.

So, we just try it, and if we got any collisions, we just try a new h. On average, we will only need to do this twice.

Proof of Claim:

- 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, $\Pr(\text{exists a collision}) \leq \binom{N}{2}/M < 1/2$. ■

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.

What if we want to use just $O(N)$ space?

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

This 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 n_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 $\sum_i (n_i)^2$. What remains is to show that we can find a first-level function h such that $\sum_i (n_i)^2 = O(N)$. In fact, we will show the following:

Theorem 10.5 *If we pick the initial h from a universal set H, then*

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

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

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

10.6 Further discussion: other uses of hashing

Suppose we have a long sequence of items and we want to see how many *different* items are in the list. What is a good way of doing that?

One way is we can create a hash table, and then make a single pass through our sequence, for each element doing a lookup and then inserting if it is not in the table already. The number of distinct elements is just the number of inserts.

Now what if the list is really huge, so we don't have space to store them all, but we are OK with just an approximate answer. E.g., imagine we are a router and watching a lot of packets go by, and we want to see (roughly) how many different source IP addresses there are.

Here is a neat idea: say we have a hash function h that behaves like a random function, and let's think of $h(x)$ as a real number between 0 and 1. One thing we can do is just keep track of the *minimum* hash value produced so far (so we won't have a table at all). E.g., if keys are 3,10,3,3,12,10,12 and $h(3) = 0.4, h(10) = 0.2, h(12) = 0.7$, then we get 0.2.

The point is: if we pick N random numbers in $[0, 1]$, the expected value of the minimum is $1/(N+1)$. Furthermore, there's a good chance it is fairly close (we can improve our estimate by running several hash functions and taking the median of the minimums).

Question: why use a hash function rather than just picking a random number each time? That is because we care about the number of *different* items, not just the total number of items (that problem is a lot easier: just keep a counter...).