princeton univ. F'23 cos 521: Advanced Algorithm Design

Lecture 2: Hashing

Lecturer: *Huacheng Yu*

1 Hashing: Preliminaries

Hashing can be thought of as a way to *rename* an address space. For instance, a router at the internet backbone may wish to have a searchable database of destination IP addresses of packets that are whizing by. An IP address is 128 bits, so the number of possible IP addresses is 2^{128}, which is too large to let us have a table indexed by IP addresses. Hashing allows us to rename each IP address by fewer bits.

Formally, we want to store a subset S of a large universe U (where $|U| = 2^{128}$ in the above example). And $|S| = m$ is a relatively small subset.

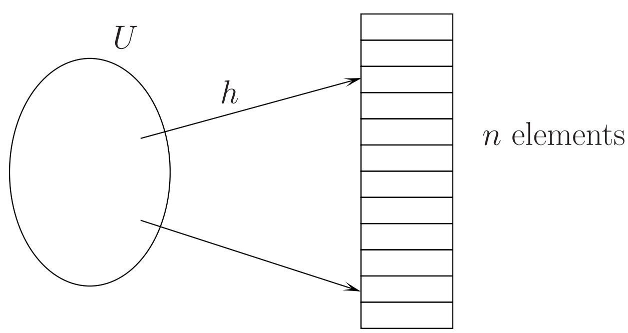

Figure 1: Hash table. x is placed in $T[h(x)]$.

We design a hash function

$$h : U \longrightarrow \{0, 1, \dots, n-1\} \tag{1}$$

such that $x \in U$ is placed in $T[h(x)]$, where T is a table of size n. Typically, we can assume that $m \leq n \ll |U|$.

There are two flexible components in this design: 1. the hash function h; 2. how do we deal with multiple elements that are mapped to the same location in T. In this lecture, we will focus on the former, designing good hash functions. For resolving hash collisions, we use the standard linked list solution – storing all keys mapped to the same location in T using a linked list. If there are t such keys, then it takes $O(t)$ to search through them.

The behavior of the hash function can be analysed under two kinds of assumptions:

1. Assume the input is the random.
2. Assume the input is arbitrary, but the hash function is random.

Assumption 1 may not be valid for many applications.

Hashing is a concrete method towards Assumption 2. We designate a set of hash functions $\mathcal{H}$, and when it is time to hash S, we choose a random function $h \in \mathcal{H}$ and hope that on average we will achieve good performance for S. This is a frequent benefit of a randomized approach: no single hash function works well for every input, but the average hash function may be good enough.

2 Hash Functions

What do we want out of a random hash function? Ideally, we would hope that h "evenly" distributes the elements of S across the hash table. One option would be to map every element in U to a random value in $[n]$. However, constructing such a "fully random" hash function is very expensive: we would need to build a lookup table with $|U|$ rows, each storing $\log_2(n)$ bits to specify the value of $h(x) \in [n]$ for one $x \in U$. At this cost, we might as well have just stored our original data in a $|U|$ length array – it's often simply impossible.

The goal in hashing is to find a *cheaper* function (fast and space efficient) that's still *random enough* to evenly distribute elements of S into our table. For a family of hash functions $\mathcal{H}$, and for each $h \in \mathcal{H}$, $h : U \longrightarrow [n]$[1], what we mean by "random enough".

For any $x_1, x_2, \ldots, x_m \in S$ ($x_i \neq x_j$ when $i \neq j$), and any $a_1, a_2, \ldots, a_m \in [n]$, ideally a random $\mathcal{H}$ should satisfy:

- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1] = \frac{1}{n}$.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2] = \frac{1}{n^2}$. Pairwise independence.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_k) = a_k] = \frac{1}{n^k}$. k-wise independence.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_m) = a_m] = \frac{1}{n^m}$. Full independence (note that $|U| = m$).

Generally speaking, we encounter a tradeoff. The more random $\mathcal{H}$ is, the greater the number of random bits needed to generate a function h from this class, and the higher the cost of computing h. The challenge is to prove that, even when we use few random bits, the hash stable still performs well in terms of insert/delete/query time.

2.1 Goal One: Bound expected number of collisions

As a first step, we want to understand the expected length of a single linked list. Note that this is just the first step towards understanding the runtime of our desired operations. Assume that $\mathcal{H}$ is a pairwise-independent hash family.

Now, we want to count the expected number of collisions. To do this, let the random variable

$$I_{xy} = \begin{cases} 1 & \text{if } h(y) = h(x), \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

[1] We use $[n]$ to denote the set $\{0, 1, \ldots, n-1\}$

Observe that the number of collisions is exactly $\sum_{x \neq y} I_{xy}$. By linearity of expectation, we get:

$$\mathbb{E}[\#\text{ collisions}] = \sum_{x \neq y} \mathbb{E}[I_{xy}] = \sum_{x \neq y} 1/n = \binom{m}{2}/n. \tag{3}$$

Above, the second inequality follows as $h(x) = h(y)$ with probability exactly $1/n$ whenever $\mathcal{H}$ is pairwise independent. Observe that if, for example, we take $n \geq m^2$, then we are likely to have zero collisions. Similarly, observe that for a fixed x, even when $n = 2m$, that x is unlikely to have any collisions.

3 2-Universal Hash Families

Definition 1 (Carter Wegman 1979)**.** *Family $\mathcal{H}$ of hash functions is 2-universal if for any $x \neq y \in U$,*

$$\mathop{\mathbf{Pr}}_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n} \tag{4}$$

Exercise: Convince yourself that this property is weaker than pairwise independence – i.e. that every pairwise independent hash function also satisfies (4).

We can design 2-universal hash families in the following way. Choose a prime $p \in \{|U|, \ldots, 2|U|\}$,[2] and let

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0) \tag{5}$$

Then let

$$h_{a,b}(x) = f_{a,b}(x) \mod n \tag{6}$$

We now make a few observations about $f_{a,b}(\cdot)$, before arguing that the family $\mathcal{H} = \{h_{a,b}(\cdot)\}_{a,b \in [p], a \neq 0}$ is 2-universal.

observation 1. *If $x_1 \neq x_2$, then $f_{a,b}(x_1) \neq f_{a,b}(x_2)$.*

Proof. Assume for contradiction that $f_{a,b}(x_1) = f_{a,b}(x_2) = s$. Then:

$$\begin{aligned} ax_1 + b = s &\mod p \\ ax_2 + b = s &\mod p \\ \Rightarrow a(x_1 - x_2) = 0 &\mod p. \end{aligned}$$

But as p is prime, and $a \neq 0$, this implies that $x_1 = x_2$, a contradiction. □

Of course, it could still very well be the case that $h_{a,b}(x_1) = h_{a,b}(x_2)$. So we have to later analyze the probability of this.

[2] How do we know that such a prime exists? This is due to Bertrand's Postulate, which exactly states that such a prime exists. Second, how do we find such a prime? One option is to guess random numbers between $|U|$ and $2|U|$, check if they're prime, and continue until we find one. The Prime Number Theorem states that each guess is likely to be prime with probability roughly $1/\log(|U|)$. Also, the AKS primality test lets us test whether a number is in fact prime in time $\text{poly}(\log(|U|))$. Alternatively, one could imagine an online pre-computed database of primes that lie in the correct range.

Lemma 1. *For any $x_1 \neq x_2$ and $s \neq t$, the following system*

$$ax_1 + b = s \mod p \tag{7}$$
$$ax_2 + b = t \mod p \tag{8}$$

has exactly one solution (i.e. one set of possible values for a, b). In that solution, $a \neq 0$.

Proof. If you're familiar with modular arithmetic, this is clear. Since p is a prime, the integers $\mod p$ constitute a finite field. This implies that any element in $[p]$ has a multiplicative inverse $\mod p$, so we know that $a = (x_1 - x_2)^{-1}(s - t)$ and $b = s - ax_1$.

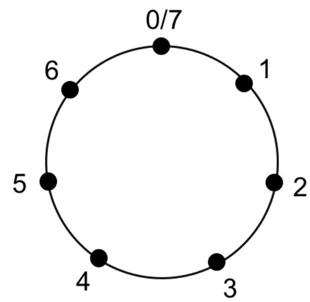

Figure 2: Modular arithmetic for prime $p = 7$.

It's not to hard to see this directly with a little thought. We want to claim that

$$a(x_1 - x_2) = (s - t) \mod p$$

has a unqiue solution a. Without loss of generality, assume that $x_1 > x_2$. When we multiply $(x_1 - x_2)$ by an integer, we're moving around the circle pictured in Figure 2 in increments of $(x_1 - x_2)$. Since p is prime, at each step before the p^{th} step, it better be that we hit a new element of $[p]$ on the circle. Otherwise, we would have found that $(x_1 - x_2)$ (which is $< p$) multiplies by some other number $< p$ to equal a multiple of p. This of course can't be true when p is prime.

So, as we multiply $(x_1 - x_2)$ by integers in $[p]$, we hit $(s - t) \mod p$ exactly once. $\square$

By Lemma 1, since there are $p(p - 1)$ different possible choices of a, b:

$$\mathop{\mathbf{Pr}}_{a,b \leftarrow U(\{1,\ldots,p-1\} \times \{0,\ldots,p-1\})}[f_{ab}(x_1) = s \wedge f_{ab}(x_2) = t] = \frac{1}{p(p-1)} \tag{9}$$

Claim $\mathcal{H} = \{h_{a,b} : a, b \in [p] \wedge a \neq 0\}$ *is 2-universal.*

Proof. For any $x_1 \neq x_2$,

$$\begin{aligned}
&\mathbf{Pr}[h_{a,b}(x_1) = h_{a,b}(x_2)] && (10)\\
=&\sum_{s,t\in[p],s\neq t} \mathbb{1}[s = t \mod n)] \cdot \mathbf{Pr}[f_{a,b}(x_1) = s \wedge f_{a,b}(x_2) = t] && (11)\\
=&\frac{1}{p(p-1)} \sum_{s,t\in[p],s\neq t} \mathbb{1}[s = t \mod n] && (12)\\
\leq&\frac{1}{p(p-1)} \frac{p(p-1)}{n} && (13)\\
=&\frac{1}{n} && (14)
\end{aligned}$$

where $\mathbb{1}$ is an indicator function (that is, $\mathbb{1}[x] = 1$ if statement x is true, and $\mathbb{1}[x] = 0$ otherwise). Equation (13) follows because for each $s \in [p]$, we have at most $\lceil p/n \rceil$ t such that $s = t \mod n$, and one of these is $s = t$ itself. So there are at most $\lceil p/n \rceil - 1 \leq (p-1)/n$ different t such that $s \neq t$ and $s = t \mod n$. □

4 Perfect hashing

Can we design a collision free hash table then? This is usually referred to as perfect hashing.

Solution 1: Collision-free hash table in $O(m^2)$ space.

Say we have m elements, and the hash table is of size n. Since for any $x_1 \neq x_2$, $\mathbf{Pr}_h[h(x_1) = h(x_2)] \leq \frac{1}{n}$, the expected number of total collisions is just

$$\mathbb{E}[\sum_{x_1\neq x_2} h(x_1) = h(x_2)] = \sum_{x_1\neq x_2} \mathbb{E}[h(x_1) = h(x_2)] \leq \binom{m}{2}\frac{1}{n} \tag{15}$$

Let's pick $n \geq m^2$, then

$$\mathbb{E}[\text{number of collisions}] \leq \frac{1}{2} \tag{16}$$

and so by Markov's inequality,

$$\mathop{\mathbf{Pr}}_{h\in H}[\exists \text{ a collision}] \leq \frac{1}{2} \tag{17}$$

So if the size the hash table is large enough, we can easily find a collision free hash function. In particular, if we try a random hash function it will succeed with probability 1/2. If we see a collision when inserting elements of S into the table, we simply draw a new random hash function and try again. The expected function of this proceedure is:

$$\mathbb{E}[\text{time to insert } m \text{ items}] = m + \frac{1}{2}m + \frac{1}{4}m + \ldots = 2m.$$

Solution 2: Collision-free hash table in $O(m)$ space (FKS hashing).

At this point, we have designed a hash table that has no collisions. The drawback is that it is that our table must be large: m^2 to store only m elements. But in reality, such a large table is often unrealistic. We may use a two-layer hash table to avoid this problem.

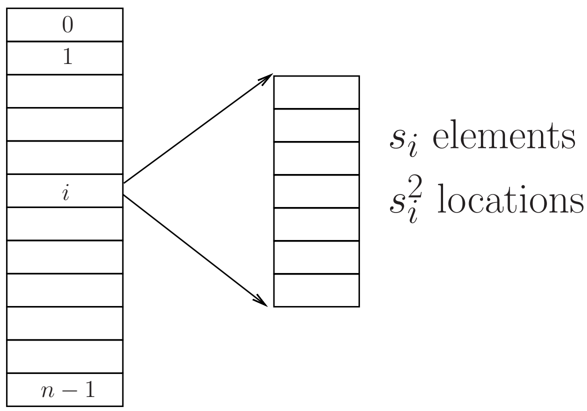

Figure 3: Two layer hash tables.

Specifically, let s_i denote the number of elements at location i. If we can construct a second layer table of size s_i^2, we can easily find a collision-free hash table to store all the s_i elements. Thus the total size of the second-layer hash tables is $\sum_{i=0}^{m-1} s_i^2$.

To bound the expected size of $\sum_{i=0}^{m-1} s_i^2$, we note that this sum is nearly equal to the total number of hash collisions, which we bound in Equation (15)! Specifically,

$$\mathbb{E}[\sum_i s_i^2] = \mathbb{E}[\sum_i s_i(s_i-1)] + \mathbb{E}[\sum_i s_i] = \frac{m(m-1)}{n} + m \leq 2m \tag{18}$$

Note that $s_i(s_i-1)/2$ is exactly the number of *collisions* at location i (because if there are s_i elements at location i, there are $\binom{s_i}{2}$ pairs which collide at i). Therefore, $\mathbb{E}[\sum_i s_i(s_i-1)/2]$ is exactly the expected number of total collisions, which we bounded with $\binom{m}{2}/n$ previously.

To construct the hash function, we will first sample a first-layer function h such that $\sum_i s_i^2 \leq 4m$, then allocate a range of $2s_i^2$ for bucket i for the second layer.

Including the first layer, we have now designed a hash table of size $O(m)$ to store m elements (so some overhead, but much less than before).

FKS hashing is mostly used in the *static* setting, where the set S is given in advance and does not change over time. However, it is also possible to support key insertions and deletions by rebuilding. That is, when we insert some x to the hash table, if it remains collision-free, then we just insert it to the corresponding entry; otherwise, we rebuild the corresponding bucket with possibly larger space (as s_i increases). If $\sum_i s_i^2$ becomes too large after the insertion, we rebuild the whole hash table. It can be shown that the expected insertion time is $O(1)$, and it has the benefit that its lookup time is $O(1)$ *in worst-case.*

Note that for perfect hashing, the hash function used is inevitably dependent on the set S. For the above two-layer construction, it also takes $O(m \log U)$ bits to just encode the hash function, while it still only takes $O(1)$ to compute the hash value of any x. A more careful encoding can improve it to $O(m)$ bits, but it has been shown that one cannot hope to encode a perfect hash function using $\ll m$ bits, if the size of the hash table is linear.