Lecture 11 (October 2nd)

Hashing

How do we design hash functions that work well?

We have a universe $\mathcal{U}$ of objects that we want to store in a hash table
We will take $\mathcal{U}$ to be the set of w-bit words, i.e.,

$$\mathcal{U} = \{0, 1, \ldots, 2^w - 1\} \triangleq [2^w] \qquad w\text{-bit words}$$

We will also have a table $T[0, \ldots, m-1]$ where $m = 2^\ell$. These correspond to ℓ-bit labels we want to hash to.

← Not a big assumption in practice

Then we have a hash function that maps elements of $\mathcal{U}$ to ℓ-bit labels

$$h: \mathcal{U} \longrightarrow [m]$$

What we would like is that if we apply the hash function to two different objects in $\mathcal{U}$, we get different labels but if we apply the function to the same value we get the same label

This is impossible! So, collisions are inevitable!

There are some standard things that people suggest. For example,

$$h(x) = \lfloor \phi x \rfloor \bmod m \quad \text{where } \phi \text{ is the golden ratio}$$

This is used as an example of a good hash function in Knuth's art of computer programming book. The argument he gives is the following: suppose we start hashing $0, 1, 2, 3, \ldots, s$ where $s \leq m-1$, then the numbers are put in the table as far as possible, i.e., the smallest gap between them is as large as it could be. This is the property of the golden ratio.

But if we are just hashing integers $\leq m-1$, we can just use $h(x)=x$, i.e. an array. We don't need hashing! So, this is not a good hash function!

In fact, it is easy for someone to design an input for this hash function that all map to zero to slow down your algorithm This is not just a theoretical exercise. Most of 21st century hashing was developed at AT&T. Why does AT&T care about hashing? It turns out that one of the things that AT&T does is that it maintains routers for the backbone of the internet. Packets come in these routers, they have addresses in them and the router needs to very quickly look up that address

and figure out where to send the packet next. There is a hash table in there especially because the information about where to send things changes over time, so it's actually a dynamically changing hash table. This decision is made billions of time a second, so the hash function has to be fast. On the other hand, there are malicious actors out there who want to bring the network to its knees. So, one of the attacks that AT&T has to defend against is people sending packets through the network looking for delays, literally, looking for collisions in the hash table. So, the routers are using complex sophisticated hash functions that we will see today. These guarantee mathematically that calculations happen in microseconds but also it is resistant to these kinds of denial of service attacks. You should just let security experts implement these functions because it really has an effect on the world, and it is quite complicated.

Here, we just focus on one aspect of hashing that we can explain in this class, which is that hash function has to behave randomly. You might have seen people saying this and writing down a deterministic algorithm. So, the assumption here is that the data we are hashing is somewhat random. This is a dangerous assumption. Data is not random. It could be chosen by a malicious party that wants to attack your system. Also, to figure out how your data is distributed even when it is random is not easy. So, we can't assume anything about the data. That means if we want things to behave randomly, we have to supply the randomness

So, the right way to think about hashing is the following:

Imagine we have a family $\mathcal{H}$ of hash functions. This is fixed in the code. When we initialize the hash table we choose a particular function from this set at random. And then we use this for the lifetime of the data structure.

So, the data is not random but the hash function is!

The usual way to do this is via a two parameter hash function $h(s, x)$ where x is the object we want to hash and s (called the salt) choses a random hash function in the family

So, what properties do we want from our hash function (family)

- Uniform

$$\mathbb{P}_{h \in \mathcal{H}}\left[h(x) = i\right] = \frac{1}{m} \quad \text{for all } x \in \mathcal{U} \text{ and for all } i \in [m]$$

This is fairly intuitive as the hash function should distribute the objects fairly uniformly, but this is not what we want as $h(s,x) = s$ satisfies this but it is not really "hashing"

This is not enough!

- Universal — This is much more useful property. We actually want the probability of collision to be small.

$$\mathbb{P}_{h \in H}\left[h(x) = h(y) \right] \leq \frac{1}{m} \quad \text{for all } x \neq y$$

Sometimes we want a stronger variant called 2-uniform

$$\mathbb{P}_{h \in H}\left[h(x) = i \text{ and } h(x) = j \right] = \frac{1}{m^2} \quad \text{for all } x \neq y \text{ for all } i \& j$$

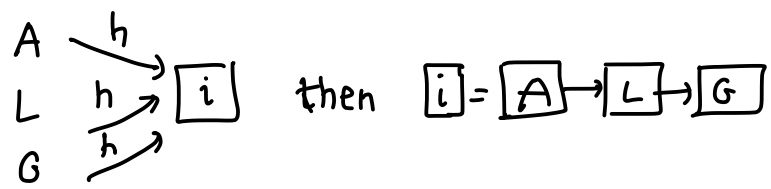
Here every pair of hash values are uniformly distributed.
You can generalize it to 3-uniform, 4-uniform, etc.

The limit is the ideal case where all hash values are uniformly and independently distributed, which is the same as k-uniform for all k.

This is what we would like but is not very practical since complete independence is not efficient to achieve. So, we need limited notions of independence so we can compute it quickly.

Again there are going to be collisions even if we pick a hash function randomly.
So, what can we do?

- We can use chained hashing, where $T[i]$ = linked list of items with hash value i



But we can use any secondary data structure as well. Assuming it is a linked list,

$$\text{Expected time to search for } x = O\left(\mathbb{E}\left[\text{length}\left(T(h(x))\right]\right)\right)$$

So, let's compute this. Let's suppose our hash table stores $y_1 \ldots y_n$ and x is not in the table yet. To compute the expected time, we are going to compute the above expectation via indicator variables.

$$\mathbb{E}\left[\text{len}\left(T(h(x))\right)\right] = \sum_{i=1}^{n} \mathbb{E}\left[\mathbb{1}\left[h(x) = h(y_i)\right]\right] = \sum_{i=1}^{n} \mathbb{P}\left[h(x) = h(y_i)\right]$$

$$\leq \frac{n}{m} \quad \text{(universal hashing)}$$

This means that $\mathbb{E}[\text{time for unsuccessful search}] = O\left(\frac{n}{m} + 1\right)$

↑ Load factor of hash table

So, if load factor $= O(1)$ & h is universal, then $\mathbb{E}[\text{time}] = O(1)$

The last thing we need to guarantee is a universal hash family.
We are going to state some examples of universal hash families (proof in the notes)

- $h(x) = (ax + b \mod p) \mod m$

  a, b: salt, $0 \le a, b \le p-1$

  p: prime # $> m$

  This is weakly universal

  $$\mathbb{P}[h(x) = h(y)] \le \frac{2}{m}$$

- Random matrix hashing

  We want a function from $[2^w]$ to $[2^\ell]$
  We can think of it as a function from $\{0,1\}^w$ to $\{0,1\}^\ell$

  Then hash function family is

  $$h_M = Mx \mod 2$$ where $M \in \{0,1\}^{\ell \times w}$ is a matrix chosen at random

  $$= \begin{bmatrix} \langle M_1, x \rangle \mod 2 \\ \\ \\ \langle M_\ell, x \rangle \mod 2 \end{bmatrix}$$

  where M_i is the i^{th} row of M

  One can show that $\mathbb{P}[h(x) = h(x)] = \frac{1}{m}$ for all $x \neq y \neq 0$

  (Proof left as exercise)

Stronger Guarantees

What we saw above gives that if $m = \theta(n)$, $\mathbb{E}\left[\begin{matrix}\text{Time to search} \\ \text{for } x\end{matrix}\right] = \theta(1)$ for all x

But we would like to say something stronger ideally

$$\mathbb{E}\left[\max_x \underbrace{\text{Time to search for } x}_{:= T(x)}\right] = O(1)$$

But infact, this does not work as above even if we assume ideal hash functions. If $n = m$,

$$\mathbb{E}\left[\max \text{ chain length}\right] = \Theta\left(\frac{\log n}{\log\log n}\right)$$

$\equiv$ items being hashed $\quad$ $\equiv$ cells of the hash table

Lemma If we toss n balls uniformly into n bins, then whp the fullest bin contains

$$O\left(\frac{\log n}{\log\log n}\right) \text{ balls}$$

← One can also prove that fullest bin contains atleast

$\Omega\left(\frac{\log n}{\log\log n}\right)$ balls but it is more complicated

Proof Let X_j = number of balls in bin j

$$\mathbb{E}[X_j] = 1 \quad \text{for all } j$$

What is the probability that bin j contains $\geq k$ balls

$$\mathbb{P}[X_j \geq k] \leq \binom{n}{k}\left(\frac{1}{n}\right)^k$$

→ $\mathbb{P}$ of each choice

↳ # choices of k balls

$$\leq \frac{n^k}{k!} \cdot \frac{1}{n^k} \leq \frac{1}{k!}$$

Taking $k = \frac{2c\log n}{\log\log n}$, $k! \geq k^{k/2} = e^{\frac{k}{2}\log k}$

$$\geq e^{\frac{2c\log n}{\log\log n}\left(\frac{\log\log n}{2}\right)}$$

$$= n^c$$

So, $\mathbb{P}\left[X_j \geq \frac{2c\log n}{\log\log n} \right] < \frac{1}{n^c}$ for all bins j

By union bound,

$$\mathbb{P}\left[\max_j X_j \geq \frac{2c\log n}{\log\log n} \right] = \mathbb{P}\left[X_j \geq \frac{2c\log n}{\log\log n} \text{ for some } j \right]$$

$$\leq \sum_{j=1}^{n} \mathbb{P}\left[X_j \geq \frac{2c\log n}{\log\log n} \right] < \frac{1}{n^{c-1}} \quad \blacksquare$$

So, what do we do? The technique is called Perfect Hashing.

Perfect Hashing — due to Komlos-Szemeredi

The idea is to replace linked lists with secondary hash tables

Each item i points to a secondary hash table of size $m_i = n_i^2$ where $n_i = \{\#\, x \in T \mid h(x) = i\}$ i.e. number of items with hash value i

Why quadratic size?

Lemma If $m = n^2$, then $\mathbb{E}[\#\text{ collisions}] < 1$ assuming universal hashing.

Proof $\mathbb{E}[\#\text{ collisions}] = \sum_{x \neq y} \mathbb{P}[h(x) = h(y)]$

$$\leq \binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2} \quad \square$$

So, Markov's inequality implies that $\mathbb{P}[>1 \text{ collision}] < 1/2$

Most of the times there will not be any collision.

So, after two tries there are no collisions! This is why we choose a secondary hash table. We first look in the first hash table for x then the second, either x is there or not but no collisions

Lookup(x)

```
i ← h(x)
j ← h_i(x)
If   T_i[j] = x
        return TRUE
else
        return FALSE
```

Primary Hash Function

Secondary Hash Function associated with T[i]

The obvious disadvantage is that this seems to be using a lot more space, quadratic in n. But n_i depends on # items hashed to i which is random, so the question is :

What is the expected size of this table ?

It turns out that it is not too bad.

$$\mathbb{E}[\text{Total Space}] = \mathbb{E}\left[\sum_i n_i^2\right] = \sum_i \mathbb{E}[n_i^2]$$

What is n_i ? $\quad n_i = \sum_x \mathbb{1}[h(x) = i]$

So,

$$\mathbb{E}[n_i^2] = \sum_{x,y} \mathbb{E}\left[\mathbb{1}[h(x)=i] \cdot \mathbb{1}[h(y)=i]\right]$$

$$= \sum_x \mathbb{E}[\mathbb{1}[h(x)=i]] + \sum_{x \neq y} \mathbb{P}[h(x)=i \text{ AND } h(y)=i]$$

So, $\mathbb{E}[\text{Total Space}] = \underbrace{\sum_i \mathbb{E}[\mathbb{1}[h(x)=i]]}_{= n \text{ (because we are hashing } n \text{ items)}} + 2\sum_i \sum_{x<y} \mathbb{P}[h(x)=i, h(y)=i]$

$$= n + 2\underbrace{\sum_{x<y}\left(\sum_i \mathbb{P}[h(x)=i, h(y)=i]\right)}_{\mathbb{E}[\#\text{ collisions in primary table}]}$$

$$= n + 2\binom{n}{2}\frac{1}{n} = 2n - 1 = O(n)$$

↑ Linear Space