Chapter 1

Hashing

Today we briefly study hashing, both because it is such a basic data structure, and because it is a good setting to develop some fluency in probability calculations.

1.1 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. Furthermore, this renaming is done probabilistically, and the renaming scheme is decided in advance before we have seen the actual addresses. In other words, the scheme is *oblivious* to the actual addresses.

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. For each $x \in U$, we want to support 3 operations:

- $insert(x)$. Insert x into S.
- $delete(x)$. Delete x from S.
- $query(x)$. Check whether $x \in S$.

A hash table can support all these 3 operations. We design a hash function

$$h : U \longrightarrow \{0, 1, \ldots, n-1\} \tag{1.1}$$

such that $x \in U$ is placed in $T[h(x)]$, where T is a table of size n.

Since $|U| \gg n$, multiple elements can be mapped into the same location in T, and we deal with these collisions by constructing a linked list at each location in the table.

One natural question to ask is: how long is the linked list at each location?

This can be analysed under two kinds of assumptions:

1. Assume the input is the random.

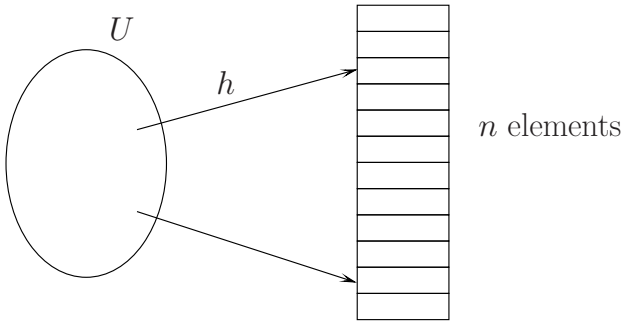

Figure 1.1: Hash table. x is placed in $T[h(x)]$.

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.

1.2 Hash Functions

Suppose we have a family of hash functions $\mathcal{H}$, and for each $h \in \mathcal{H}$, $h : U \longrightarrow [n]$, where $[n]$ denote the set $\{0, 1, \ldots, n-1\}$. What does it mean to say these functions are random?

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 trade-off. 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.

For example, if $\mathcal{H}$ is a fully random family, there are n^m possible h, since each of the m elements at S have n possible locations they can hash to. So we need $\log |\mathcal{H}| = m \log n$ bits to represent each hash function. Since m is usually very large, this is not practical.

But the advantage of a random hash function is that it ensures very few collisions with high probability. Let L_x be the length of the linked list containing x; this is just the number

of elements with the same hash value as x. Let random variable

$$I_y = \begin{cases} 1 & \text{if } h(y) = h(x), \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

So $L_x = 1 + \sum_{y \in S; y \neq x} I_y$. Furthermore, remember that expectation is a linear operator: the expectation of the sum of random variables is the sum of their expectations. (This simple fact saves the day in many probabilistic calculations.)

$$E[L_x] = 1 + \sum_{y \in S; y \neq x} E[I_y] = 1 + \frac{m-1}{n} \tag{1.3}$$

Usually we choose $n > m$, so this expected length is less than 2. Later we will analyse this in more detail, asking how likely is L_x to exceed say 100.

The expectation calculation above doesn't need full independence; pairwise independence would actually suffice. This motivates the next idea.

1.3 2-Universal Hash Families

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

$$\mathop{\mathbf{Pr}}_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n} \tag{1.4}$$

Sometimes this definition is relaxed to allow $2/n$ (or $3/n$) instead of $1/n$, since that doesn't greatly affect the bucket sizes needed to handle collisions.

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

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0) \tag{1.5}$$

And let

$$h_{a,b}(x) = f_{a,b}(x) \mod n \tag{1.6}$$

The reason this construction works is that the integers modulo p form a field when p is prime, meaning that it is possible to define addition, multiplicition and division (except division by 0, which is undefined) among them.

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

$$ax_1 + b = s \mod p \tag{1.7}$$
$$ax_2 + b = t \mod p \tag{1.8}$$

has exactly one solution.

Since $[p]$ constitutes a finite field, we have that $a = (x_1 - x_2)^{-1}(s - t)$ and $b = s - ax_1$. Since we have $p(p-1)$ different hash functions in $\mathcal{H}$ in this case,

$$\mathop{\mathbf{Pr}}_{h\in\mathcal{H}}[h(x_1) = s \wedge h(x_2) = t] = \frac{1}{p(p-1)} \tag{1.9}$$

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

PROOF: For any $x_1 \neq x_2$,

$$\begin{aligned}
&\mathbf{Pr}[h_{a,b}(x_1) = h_{a,b}(x_2)] && (1.10)\\
=&\sum_{s,t\in[p],s\neq t} \delta_{(s=t \mod n)}\, \mathbf{Pr}[f_{a,b}(x_1) = s \wedge f_{a,b}(x_2) = t] && (1.11)\\
=&\frac{1}{p(p-1)} \sum_{s,t\in[p],s\neq t} \delta_{(s=t \mod n)} && (1.12)\\
\leq&\frac{1}{p(p-1)} \frac{p(p-1)}{n} && (1.13)\\
=&\frac{1}{n} && (1.14)
\end{aligned}$$

where δ is the Dirac delta function. Equation (1.13) follows because for each $s \in [p]$, we have at most $(p-1)/n$ different t such that $s \neq t$ and $s = t \mod n$. □

Can we design a hash function for the data set that has no collisions? We show this is possible if $n \geq m^2$, where m is the size of the set being hashed. 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

$$E[\sum_{x_1\neq x_2} h(x_1) = h(x_2)] = \sum_{x_1\neq x_2} E[h(x_1) = h(x_2)] \leq \binom{m}{2}\frac{1}{n} \tag{1.15}$$

Since $n \geq m^2$, we have

$$E[\text{number of collisions}] \leq \frac{1}{2} \tag{1.16}$$

and so

$$\mathop{\mathbf{Pr}}_{h\in H}[\exists \text{ a collision}] \leq \frac{1}{2} \tag{1.17}$$

So if the size the hash table is large enough, there exists a collision-free hash function, and in fact a random hash function from the above family is collision-free with probability at least 1/2. But in reality, such a large table is often unrealistic.

A more practical method to deal with collisions is to allow a linked list (also called bucket) at each location, which can be used to store additional elements in case of collisions. Sometimes this is also called a 2-layer hash.

Specifically, let s_i denote the number of collisions 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 $\sum_{i=0}^{m-1} s_i^2$.

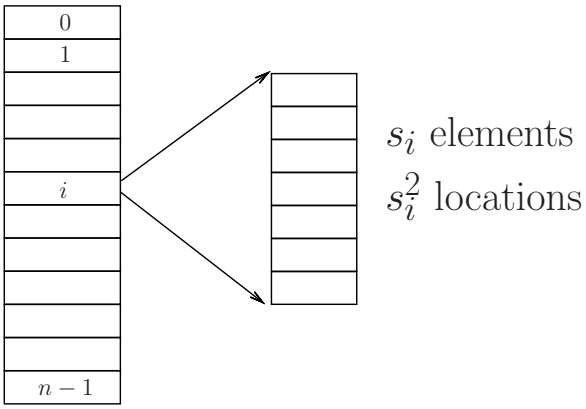

Figure 1.2: Two layer hash tables.

Note that $\sum_{i=0}^{m-1} s_i(s_i-1)$ is just the number of collisions calculated in Equation (1.15), so

$$E[\sum_i s_i^2] = E[\sum_i s_i(s_i-1)] + E[\sum_i s_i] = \frac{m(m-1)}{n} + m \leq 2m \tag{1.18}$$

Do we need to know the size of the set? The above calculation shows that if n, the size of the hash table, is roughly the same as m, the size of the set being hashed, then the expected bucket size at each hash location (and hence the expected lookup time) is at most $2m/n$, which is $O(1)$. This leads to the question: *Is it necessary to know m before we pick the size of the hash table?* The answer is no. Instead it suffices to adaptively increase the size of the hash table. Suppose the current hash table has size 2^i. As elements of the set arrive, keep hashing them until the expected bucket size rises above 2. This means the set must be now of size about 2^i. Now *rebuild* the hash table to one of size 2^{i+1} using a new hash function. The total time spent in rebuilding is only $O(2^{i+1})$, which is still a constant factor times the size of the set we already have, which is 2^i. Thus the entire hashing still takes $O(1)$ time per element.

1.3.1 Pair-wise independence

The above construction of random hash functions is a special case of a phenomenon called *pairwise independence*. A set of random variables $X_1, X_2, \ldots,$ is called *pairwise independent* if for all values s, t, and for all indices $i \neq j$

$$\Pr[X_i = s \wedge X_j = t] = \Pr[X_i = s] \Pr[X_j = t].$$

The above discussion actually shows that if we pick a random hash functions as above, picking $h(x) = ax + b \bmod p$ where a, b are random integers mod p, then the places where the keys get hashed to are pairwise independent: The set of random variables $h(x)$ are pairwise independent (over the choice of h).

We return to this property later and also in the homeworks.

1.4 Load Balancing

Now we think a bit about how large the linked lists (ie number of collisions) can get. Let us think for simplicity about hashing n keys in a hash table of size n. This is the famous balls-and-bins calculation, also called load balance problem. We have n balls and n bins. For simplicity, assume that each balls is assigned to a random bin. (In other words, the hash function is a completely random function instead of just pairwise independent as above.) Clearly, the expected number of balls in each bin is 1. But the maximum can be a fair bit higher.

For a given i,

$$\mathbf{Pr}[\text{bin}_i \text{ gets more than } k \text{ elements}] \leq \binom{n}{k} \cdot \frac{1}{n^k} \leq \frac{1}{k!} \tag{1.19}$$

(This uses the union bound, that the probability that any of the $\binom{n}{k}$ events happen is at most the sum of the their individual probabilities.) By Stirling's formula,

$$k! \sim \sqrt{2nk}(\frac{k}{e})^k \tag{1.20}$$

If we choose $k = O(\frac{\log n}{\log \log n})$, we can let $\frac{1}{k!} \leq \frac{1}{n^2}$. Then

$$\mathbf{Pr}[\exists \text{ a bin} \geq k \text{ balls}] \leq n \cdot \frac{1}{n^2} = \frac{1}{n} \tag{1.21}$$

So with probability larger than $1 - \frac{1}{n}$,

$$\max \text{load} \leq O(\frac{\log n}{\log \log n}) \tag{1.22}$$

By changing the parameters a little, this success probability can be improved to $1 - \frac{1}{n^c}$ for any constant c.

Exercise: Show that with high probability the max load is indeed $\Omega(\log n / \log \log n)$.

1.4.1 Improved load balancing: Power of Two Choices

The above load balancing is not bad; no more than $O(\frac{\log n}{\log \log n})$ balls in a bin with high probability. Can we modify the method of throwing balls into bins to improve the load balancing? How about the method you use at the supermarket checkout: instead of going to a random checkout counter you try to go to the counter with the shortest queue? In the load balancing case (especially in distributed settings) this is computationally too expensive: one has to check all n queues. A much simpler version is the following: when the ball comes in, pick 2 random bins, and place the ball in the one that has fewer balls. Turns out this modified rule ensures that the maximal load drops to $O(\log \log n)$, which is a huge improvement. This called the *power of two choices*. The intuition why this helps is that even though the max load is $O(\log n / \log \log n)$, most bins have very few balls. For instance, at most 1/10th of the bins will have more than 10 balls. Thus when we pick two bins randomly, the chance is good that the ball goes to a bin with constant number of balls. Let us give a proof sketch.

For a ball b let us define

$$height(b) = \text{load of its bin when } b \text{ was placed in it.}$$

Let $\rho(k,t)$ be the fraction of bins with at least k balls in it at time t.

Then the probability that the $t+1$'th ball has height $k+1$ is at most $\rho(k,t)^2$, since both of its choices must have had at least k balls when it arrived.

Noting that $\rho(2,t) \leq 1/2$ since the total number of balls is n, we use the above logic above to obtain:

$$E[\text{fraction of balls in } B \text{ w/ } height(b) \geq i] \leq \left(\frac{1}{2}\right)^{2^{i-2}}.$$

In order to bound the size of largest the bin, we want to find the value of i such that the probability on the right is $\frac{1}{n}$. For this we need,

$$\begin{aligned} 2^i &\sim \Omega(\log n) \\ \Rightarrow i &\sim \Omega(\log \log n). \end{aligned}$$

This argument is hand-wavy and not 100% precise; for a full proof please see either various lecture notes around the web, or the Mitzenmacher et al. survey linked below.

1.5 Cuckoo Hashing

Can we do the ultimate load balancing, and obtain a hashing scheme with $O(1)$ lookup time? Yes, provided we are willing to take a bit of a hit on insert operations.

A simple and practical way to do this is *cuckoo hashing*, invented by Pagh and Rodler in 2001. The name refers to the cuckoo's habit of putting its eggs in crows' nests —shifting its parental duties to others.

The idea is that we pick two hash functions h_1, h_2 instead of one. Thus each key x has two possible designated spots $h_1(x), h_2(x)$ to go to. When key x arrives, it randomly picks one of these two, say $h_1(x)$. If $h_1(x)$ happens to be occupied by some other key y, then y gets kicked out and x takes this spot. Now y has to go to its *other* designated location. If that happens to be occupied, then its occupant is kicked out and and forced to go to *its* other designated location. And so on. Thus insert can take an unbounded amount of time, or even deadlock if this chain of kickings turns around and returns to x! One way to react in case of deadlock is to rebuild the hash table in place using two new hash functions. It can be shown that the deadlock is a very rare event (ie happens with probability less than $1/n$) and so in the expectation this rebuilding cost is not a huge overhead.

It is possible to prove probabilistic bounds on the running time, as we will explore in the homeworks.

Notice however that look up takes $O(1)$ time: just check both of the designated locations, and if the key is not in either, return `fail`.

This basic cuckoo hashing idea has (inevitably) many more variants, as a web search will show.