Notes on Hashing

Cardcaptor

December 11, 2019

1 Hashing

- The *dictionary* ADT supports the following operations:
  - $insert(x)$: Insert x into the dictionary.
  - $remove(x)$: Remove x from the dictionary.
  - $find(x)$: Check whether x is in the dictionary.
- A *hash table* is an implementation of the dictionary ADT where the items to manipulate are integers or can be mapped to the integers. Let us suppose that the items comes from the set $[U] = \{0, 1, \ldots, U-1\}$ for some positive integer U.

  A hash table consists of an array a with M slots, and a *hash functions* $h : [U] \to [m]$. The array a starts empty. The three operations can be implemented as follows:
  - $insert(x)$: Compute $h(x)$ and put x in $a[h(x)]$.
  - $remove(x)$: Compute $h(x)$ and remove x from $a[h(x)]$ if it is there.
  - $find(x)$: Compute $h(x)$ and check whether x is in $a[h(x)]$.
- The simple implementation above has the problem of *collision*: there may be $x_1, x_2 \in [U]$ such that $x_1 \neq x_2$ and $h(x_1) \neq h(x_2)$. Collision is unaviodable if $U > m$.
- There are many ways to resolve collision. The most simple way is called *chaining*: instead of each slot storing only one item, it stores a linked list of items placed there. It is clear that the performance of hasing with chaining depends on the distribution of items into slots. Any operation involving items that go into a particular slot takes time linear to the number of items already in the slot.
- We are interested in the problem of building *static hash tables*. A fixed set of n items $\{x_1, x_2, \ldots, x_n\} \subseteq [U]$ is given to us. We would like to build a data structure that can perform $find(x)$ — i.e., testing whether x belongs to the set — very fast. An example of an application of static hash table is the English language dictionary.

  We describe a scheme invented by Fredman, Komlós, Szemerédi, which builds a static hash table for n items in $O(n)$ expected time using $O(n)$ space and each $find(x)$ operation takes $O(1)$ worst case time.
- The mathematical tool used in the scheme is the *universal family of hash functions*.

  Definition 1. *A set of hash functions $\mathcal{H}$ is called a* universal family of hash function *if, for all $x, y \in [U]$ such that $x \neq y$,*

$$\Pr_{h \leftarrow \mathcal{H}}[h(x) = h(y)] = \frac{O(1)}{m}.$$

  In other words, if we pick a random hash function out of h, the probability that any two fixed items collide is a constant over the size of the hash table.

- Is it easy to construct a universal family of hash function? Yes. We first pick a prime number $p > U$, and let

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in \{1, 2, \ldots, p-1\}, b \in \{0, 1, \ldots, p-1\}\},$$

where

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

Proposition 2. *$\mathcal{H}_{p,m}$ is a universal family of hash function.*

We prove the following lemma first:

Lemma 3. *Let $x, y, z, w \in [p]$ be such that $x \neq y$. Then,*

$$\Pr_{(a,b) \leftarrow [p]^2}[(ax + b) \bmod p = z \ \wedge \ (ay + b) \bmod p = w] = \frac{1}{p^2}.$$

Proof. The fact that $(ax + b) \bmod p = z$ and $(ay + b) \bmod p = w$ is equivalent to the following system of equations:

$$\begin{aligned} ax + b &\equiv z \pmod{p} \\ ay + b &\equiv w \pmod{p} \end{aligned}$$

which is equivalent to the following matrix equation:

$$\begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \equiv \begin{bmatrix} z \\ w \end{bmatrix} \pmod{p}.$$

The two-by-two matrix has determinant $x - y$, which is not zero. This means that there is one and only one solution to the system of equation. Since we pick a and b uniformly at random, the probability that a and b make up the solution for the system is $1/p^2$. □

Proof. (Proposition) Fix $x, y \in U$ such that $x \neq y$. We have that $h_{a,b}(x) = h_{a,b}(y)$ if and only if $(ax + b) \bmod p = z$ for some pair of z and w such that $z \equiv w \pmod{m}$.

We claim that there are at most $4p^2/m$ such (z, w) pairs. To see why, consider the set $0, 1, \ldots, p-1$. It can be partitioned into m subsets based on the value of each element modulo m. Each subset has at most $2p/m$ elements. Any pair of elements in each subset are congruent modulo m, so there are $4p^2/m^2$ pairs per subset. Since there are m subsets, it follows that there are at most $4p^2/m$ pairs.

By the union bound,

$$\begin{aligned} \Pr_{h_{a,b} \leftarrow \mathcal{H}_{p,m}}[h_{a,b}(x) = h_{a,b}(y)] &\leq \sum_{z \equiv w (\bmod m)} \Pr_{(a,b) \leftarrow [p]^2}[(ax + b) \bmod p = z \ \wedge \ (ay + b) \bmod p = w] \\ &= \sum_{z \equiv w (\bmod m)} \frac{1}{p^2} \leq \frac{4p^2}{m}\left(\frac{1}{p^2}\right) = \frac{4}{m} \end{aligned}$$

as desired. □

- We now describe the construction of FKS dictionary. We pick $m > cn$ for some constant c and let $\mathcal{H}$ be a universal family of hash functions. We try pick h from $\mathcal{H}$ at random, and hash $x_1, x_2, \ldots, x_n$ into the table. We stop if there are no more than n collisions. Otherwise, we pick a new hash function an try again.

We show that the above process takes $O(n)$ expected time. Let I_{ij} be an indicator random variable such that $I_{ij} = 1$ if and only if $x_i = x_j$. Then,

$$E[\#\text{collisions}] = E\Bigg[\sum_{1\le i<j<n} I_{ij}\Bigg] = \sum_{1\le i<j<n} \mathbf{E}[I_{ij}] = \sum_{1\le i<j<n} \frac{O(1)}{cn} < \frac{n^2}{2}\frac{O(1)}{cn} = \frac{O(1)}{2c}n.$$

We can use c large enough such that the expected number of collisions is less than $n/2$. By Markov's inequality, the probability that the number of collisions is greater than n is less than $1/2$. Thus, the expected number of trials is constant. Since each trial takes linear time, the process takes expected linear time.

Now, we look to the m slots of the hash table produced above. Let there be n_i elements in slot $a[i]$. For each slot, we create a *collision-free* hash table of size cn_i^2 by the process above: we pick a hash function from h, hash the elements, and try again if there's a collision. We claim that this process takes $O(n_i^2)$ expected time because

$$E[\#\text{collisions}] \le \frac{n_i^2}{2}\frac{O(1)}{cn_i^2} = \frac{O(1)}{2c}.$$

Summing the time and space used for all slots together, the total time and space taken is $O(n_0^2) + O(n_1^2) + \ldots + O(n_{m-1}^2) = O(n)$ because the total number of collisions is less than $n_0^2 + n_1^2 + \cdots + n_{p-1}^2$.