Chapter 9

Hashing

Up to this point, all the data structures we have seen in this course assume nothing but comparability about the input keys they are expected to manage. Hash functions, in general, are functions[1] from the key space into a lower dimensional space, which can be thought of as our memory storage.

Example 9.0.1. For example, assume the keys are taken from the integers, and the hash function $h : \mathbb{N} \to [10]$ sends numbers to their residue modulo 10, namely $h(x) = x \mod 10$. Now, one might use h and a 10-length array A to store numbers. Any time he would like to insert a number x into the structure, he would set $A[h(x)] \leftarrow x$.

However, this is not a "good" method, since two different keys with the same residue modulo 10 are mapped into the same cell in A, which we count as a collision:

Definition 9.0.1 (collision.)**.** Given a function $h : U \to \star$, we name any pair of different keys $x \neq y \in U$ that are mapped by h to the same value, namely $h(x) = h(y)$, a collision.

Clearly, if we assume that no collisions are going to occur, i.e. all the given keys through the running of the program $x_1, x_2, .., x_n$ satisfy that for any $x_i \neq x_j$, it follows that $h(x_i) \neq h(x_j)$, then the method in Example 9.0.1 provides a data structure that[2] supports access, insertion, and deletion in constant time. This gives us the intuition that the complexity of a general data structure which uses a hash function depends on the way it resolves collisions. In this recitation, we will present ways to handle collisions. The first is a heuristic called **open addressing** (or **closed hashing**). The second, **open hashing**, will also be analyzed, and we will show a characterization that, if satisfied, then we get a good running time in expectation. Finally, we will also show examples of function families that satisfy the characterization and examples of families that do not.

[1] projections.

[2] Again, relative to assumptions on the input.

9.1 Open Addressing (Closed Hashing).

9.2 Universal Hashing.

Definition 9.2.1. Let $\mathcal{H} = \{h_i : U \to [m]\}$ be a family of function from the domain U into $[m] = \{0, 1, ..m - 1\}$. $\mathcal{H}$ will be said universal if for any $x \neq y \in U$:

$$\mathbf{Pr}_{h \sim \mathcal{H}} [h(x) = h(y)] \leq \frac{1}{m}$$

Question. For $x = y$ what is the probability that $h(x) = h(y)$?

Example 9.2.1. $\mathcal{H}$ is the set of all function from $U \to [m]$.

Example 9.2.2. $\mathcal{H}$ is the set of all binary matrices from $U = \mathbb{F}_2^n \to \mathbb{F}_2^k$.

Example 9.2.3. $\mathcal{H}$ is the set of all function from $U \to [m]$.

Exercise 9.2.1. *U is the set of all matrices $\mathbb{F}_2^{n \times n}$ and $h_x(A) = x^\top Ax$.*

9.3 Perfect Hashing.

In the past week, we have seen how to store keys in hash tables so that the number of mapped keys in a specific cell is $O(1)$ in expectation. The table is constructed using a hashing function h : key space $\to m$cells, randomly chosen from a universal hash function family. This function maps keys to cells, and in each cell, the keys are stored using a linked list. The cost of supported subroutines depends on the length of the list. We named any pair of different keys $x \neq y$ that are mapped to the same cell in the table, namely $h(x) = h(y)$, a collision.

Perfect hashing is a method to ensure that no collision occurs, it works only if all keys are given in advance and they are unique, meaning that the table doesn't support insertion. The idea is as follows, we sample an hash function, and then check if, for all x, y in the input, it holds that $h(x) \neq h(y)$. If so then we continue. Otherwise we repeat.

Question. What is the probability of choosing h with no collisions on the first trial? Notice that the answer depends on m. (To see this, imagine the case where $m = 1$. In this case, there must be collisions.) Therefore, the correct question is: for what values of m do we succeed in finding a hash function with no collisions on the first trial? Let $X_{x,y}$ be the indicator of the event $h(x) = h(y)$. The expected number of collisions is then:

$$\mathbf{E}\left[\sum_{x \neq y} X_{x,y}\right] = \sum_{x \neq y} \mathbf{E}[X_{x,y}] = \binom{n}{2}\frac{1}{m}$$

Now, we would like to answer for what value of m there is no collision. Therefore, if we take $m = n^2$, then the expected number of collisions is less than $1/2$. By the Markov inequality, the probability of having more than one collision is less than:

$$P\left(\sum_{x \neq y} X_{x,y} > 1\right) \leq \mathbf{E}\left[\sum_{x \neq y} X_{x,y}\right] = \frac{1}{2}$$

```
let collision ← True
while collision do
    collision ← False
    let T be array at length m
    h ← sample uniformly random from universal hash family H
    for x ∈ x_1, x_2..x_n do
        if T_{h(x)} is not empty then
            collision ← True
            break the for-loop
        end
        else
            T_{h(x)} ← x
        end
    end
end
return T, h
```

Algorithm 1: perfect-hashing($x_1, x_2, ..x_n$)

And therefore the expected number of rounds is less than:

$$\mathbf{E}\left[\text{ rounds }\right] = \sum_{t=0}^{\infty} tP(t \text{ rounds }) \leq \sum_{t=0}^{\infty} t\frac{1}{2^{t-1}} = O(1)$$

Question. What is the space complexities? We have to allocate an array at length m which is $\Theta(n^2)$ memory. Is that good? So remember that in standard hash tables, the expected number of elements that were hashed into the same cell as the key x is

$$1 + \frac{n-1}{m}$$

Taking $m = \Theta(n)$ is enough to ensure that the expected running time of in insertion/deletion/access is $O(1)$. This raises the question of whether the space complexity of perfect hashing can be reduced to linear.

9.3.1 Perfect Hashing in Linear Space.

The idea is as follows: we will use a two-stage hashing process. In the first stage, keys will be mapped to hash tables instead of cells. Each hash table will be constructed using perfect hashing and may require a space that is quadratic in the number of elements stored in it (which were mapped to it in the first stage). Therefore, if we denote by n_i the number of elements mapped to the ith hash table, the space cost will be $\sum_i n_i^2$. Instead of starting over when a collision occurs, we will do so when $\sum_i n_i^2 > 4n$. So, now it's left to show that we expect $\sum_i n_i^2$ to be linear, which implies that the expected number of rounds is constant.

$$n_i^2 = 2\binom{n_i}{2} + n_i$$

```
let toomanycollisions ← True
while toomanycollisions do
    toomanycollisions ← False
    let T be array at length m
    initialize any T_i to be an empty linked list.
    h ← sample uniformly random from universal hash family \mathcal{H}
    for x ∈ x_1, x_2..x_n do
        T_{h(x)}.insert(x)
        T_{h(x)}.size = T_{h(x)}.size +1
    end
    if \sum_i T_{h(i)}.size^2 ≥ μ then
        toomanycollisions ← True
    end
end
let H be an array at length m
for i ∈ [m] do
    T_i, h_i ← hash the elements in T_i using
        perfect hashing.
end
return T, h
```

Algorithm 2: perfect-hashing-linear-space($x_1, x_2, ..x_n$)

On the other hand, $\sum_i \binom{n_i}{2}$ is exactly the number of collisions, as for any i, $\binom{n_i}{2}$ counts the number of distinct pairs in the ith table, which is equivalent to counting the number of $x \neq y$ such that $h(x) = h(y) = i$. Thus,

$$\mathbf{E}\left[\sum_i n_i^2\right] = \mathbf{E}\left[\sum_i 2\binom{n_i}{2} + n_i\right] = 2 \cdot \mathbf{E}\left[\text{collisions}\right] + \mathbf{E}\left[\overbrace{\sum_i n_i}^{n}\right]$$

$$= 2 \cdot \binom{n}{2}\frac{1}{m} + n$$

Therefore, by choosing $m = 4n$ for the first stage, the probability of failing to choose a proper hash function is less than $1/2$.