# 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 $T$ to store numbers. Any time he would like to insert a number $x$ into the structure, he would set $T[h(x)] \leftarrow x$.

However, this is not a "good" method, since two different keys with the same residue modulo 10 are mapped into the same cell in $T$, 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.

## 9.1 Open Addressing (Closed Hashing).

Open addressing uses a static $m$-length array $T$, hash function $h$, and handles collisions by probing cells in a specific order. For example, linear probing will first

[1] projections.

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

check if the cell at position $h(x)$ is not occupied or already contains $x$. If it is, then $h(x)$ is considered the correct place to store $x$. Otherwise, the next cell is probed, namely the cell at position $h(x)+1$, and so on.

```
1 let i ← 0
2 while T[h(x) + i] ≠ Null do
3 |   i ← i + 1
4 end
5 return h(x) + i
```

**Algorithm 1:** linear probing - access

## 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 universial if for any $x \neq y \in U$:

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

**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 $U \to [m]$.

Picking a function randomly from the set of all functions is equivalent to picking the targets $h(x)$, for each $x$, independently. So the probability that $h(x) = h(y)$ is as exactly as the probability, in the balls to bins experiment, that two specific balls ($x$ and $y$ in our case) will be thrown into the same bin. As we computed in the last recitation, that probability equals $1/m$.

*Example* 9.2.2. $\mathcal{H}$ is all the inner products by a $n$-length binary vector. Namely,

$$\mathcal{H} = \{h_a : \mathbb{F}_2^n \to \mathbb{F}_2, h_a(x) = \langle a, x\rangle \,|a \in \mathbb{F}_2^n\}$$

Observe that in $\mathbb{F}_2$, the equivalence $h(x) = h(y) \Leftrightarrow h(x) + h(y) = 0$ holds. If $x \neq y$, then there is at least one coordinate $i$ such that $x_i \neq y_i$. In other words, $x_i + y_i = 1$. Thus:

$$\begin{aligned} h_a(x) + h_a(y) = \langle a, x + y\rangle &= a_i(x_i + y_i) + \sum_{j\neq i} a_j(x_j + y_j) \\ &= a_i + \sum_{j\neq i} a_j(x_j + y_j) \end{aligned}$$

Thus:

$$\begin{aligned}
\mathbf{Pr}\left[h_a(x) = h_a(y)\right] &= \sum_{z\in\mathbb{F}_2} \mathbf{Pr}\left[a_i = z \cap \{\sum_{j\neq i} a_j(x_j + y_j) = z\}\right] \\
&= \sum_{z\in\mathbb{F}_2} \mathbf{Pr}\left[a_i = z\right] \cdot \mathbf{Pr}\left[\{\sum_{j\neq i} a_j(x_j + y_j) = z\}\right] \\
&= \sum_{z\in\mathbb{F}_2} \frac{1}{2} \cdot \mathbf{Pr}\left[\{\sum_{j\neq i} a_j(x_j + y_j) = z\}\right] \\
&= \frac{1}{2} \overbrace{\sum_{z\in\mathbb{F}_2} \mathbf{Pr}\left[\{\sum_{j\neq i} a_j(x_j + y_j) = z\}\right]}^{1} = \frac{1}{2}
\end{aligned}$$

When in the second pass we used independence, and in the last, we used that probabilities sum to 1.

*Example* 9.2.3. $\mathcal{H}$ is the set of all binary matrices : $\mathbb{F}_2^n \to \mathbb{F}_2^k$.

Observe that the equivalence $h(x) = h(y)$ implies an equivalence in any coordinate. Since the $i$th coordinate of $h(x)$, namely $h(x)_i$, is given by an inner product of the $i$th row of $h$, which is picked randomly from the $n$-length binary vectors, we get by Example 9.2.2 that the probability of equivalence in the $i$th coordinate is $\frac{1}{2}$, and since the rows are picked independently, we get:

$$\mathbf{Pr}\left[h(x) = h(y)\right] = \mathbf{Pr}\left[\bigcap_i^k h(x)_i = h(y)_i\right] = \left(\frac{1}{2}\right)^k$$

*Example* 9.2.4. $\mathcal{H}$ 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 $h_X(A) = \mathbf{Tr}\left(XA\right)$.*

*Solution* 9.2.1.

$$h_X(A) = \mathbf{Tr}\left(XA\right) = \sum_i^n (XA)_{ii} = \sum_i^n X_{ij}A_{ji}$$

So, the hashing is equivalence to flattening $A$ into an $n \times n$ binary vector and then multiplying it by a randomly chosen vector. Which was proved in Example 9.2.2 to be a 2-universal.

**Exercise 9.2.2.** *$U$ is the set of all binary vectors $\mathbb{F}_2^n$ containing more $1$'s than $0$'s. And $h_a(x) = \langle a, x\rangle$ for any $a$ such $\langle a, 1\rangle = 0$.*

**Exercise 9.2.3.** *$U$ is the set of all matrices $\mathbb{F}_2^{n\times n}$, and $h_x(A) = x^\top Ax$. Show that $\mathcal{H}$ is not 2-universal.*

Equipped with 2-universal hash functions, we can now say intelligent 'things' about the expected number of collisions. Specifically, denote by $h \sim \mathcal{H}$ a sampled hash function. Let us denote by $I$ the input: $I = \{x_1, x_2, \ldots, x_n\} \subset U$. We

can define for each $x \in I$ the random variable $C_x$, which counts the number of collisions in $I$ involving $x$. Namely:

$$C_x = |\{y \in I : h(x) = h(y)\}|$$

*Note, How to recall: at least 1 element for $x$ itself, and $\frac{n-1}{m}$ is what we would guess to have by throwing $n-1$ balls to $m$ bins.*

**Claim 9.2.1.** *If $\mathcal{H}$ is 2-universal hash family, then $\mathbf{E}[C_x] = 1 + \frac{n-1}{m}$.*

*Proof.* For any $x, y \in I$, let $X_{x,y}$ be the indicator of the event $h(x) = h(y)$. Observe that $C_x = \sum_{y \in I} X_{x,y}$. Therefore, by the linearity of expectation and $\mathcal{H}$ being a 2-universal hash family, we have:

$$\begin{aligned}\mathbf{E}[C_x] &= \mathbf{E}\left[\sum_{y \in I} X_{x,y}\right] = \sum_{y \in I} \mathbf{E}[X_{x,y}] = \sum_{y = x \in I} \mathbf{E}[X_{x,y}] + \sum_{y \neq x \in I} \mathbf{E}[X_{x,y}] \\ &= 1 + \sum_{y \neq x \in I} \mathbf{Pr}[h(x) = h(y)] \leq 1 + \frac{n-1}{m}\end{aligned}$$

□

### 9.2.1 Universal Hashing Table. (Open Hashing)

Universal Hashing Table also defined by an $m$-array (table), and an hash function $h$ which index keys $x \in U$ to $T[h(x)]$ cells, Yet in contrast to open addressing each cell might contains multiple keys by maintaining a linked list. Accessing to a stored key, or checking if it is in $T$, is done by iterating the linked list at suited cell. Appending a key, is done by first checking the key is not already stored in the table and then appending it to the tail of the suited linked list. **??** 2 exhibits the access routine, you are encouraged to complete the others.

**Question.** Fix a table, and consider $x \in U$. What will be the running time of each of the operation: Insert $x$, Access $x$, Delete $x$?. (Linear at the number of keys in the list as $T[h(x)]$).

**Question.** In hashing terms, what is the length of $T[h(x)]$ equal to? The number of collision with $x$.

Hence, we infer that the expected running time is bounded by the expected number of collision with $x$. If $h$ is sampled uniformly at random from a 2-universal

```
1 linked ← T[h(x)]
2 for node ∈ linked do
3 |   if node.value = x then
4 |   |   return node
5 |   end
6 end
7 return last node
```

**Algorithm 2:** Universal Hashing Table - access

family $\mathcal{H}$, then by Claim 9.2.1, the expected running time is $1 + \frac{n-1}{m} = O(1)$.

### 9.2.2 How Hashing is Implemented in Python?

In most modern programming languages, the Dictionary data structure is implemented by a variation of Cuckoo hashing, which works as follows: We sample two functions from $\mathcal{H}$, denoted by $h_1$ and $h_2$. Checking if an element $x$ is in the table is done by checking if it is in either $T[h_1(x)]$ or $T[h_2(x)]$. Inserting it into the table is done by the following:[3]

```
if either T[h_1(x)] or T[h_2(x)] are empty then
    Pick an empty spot and set x there.
    return
end
else
    i ← flip a coin.
    z ← h_i(x)
    set T[h_i(x)] ← x
    Call Insert(z).
end
```

**Algorithm 3:** Cuckoo - Insert

Roughly speaking, the algorithm fails when we close a cycle, namely after several recursive calls, back to the point where we were called to insert $x$. The idea of the analysis is that if the functions $h_1, h_2$ were picked randomly, or very similar to randomly (i.e., from $k$-universal families), that event is unlikely, and the number of scanned spots is constant.

## 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:

[3] Here we lie, a slightly clever (but not too mach) rule is used to pick $i$.

```
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 4:** perfect-hashing($x_1, x_2, ..x_n$)

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

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:

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

And therefore the expected number of rounds is less than:

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

**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 $i$th 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

```
1  let toomanycollisions ← True
2  while toomanycollisions do
3  |   toomanycollisions ← False
4  |   let T be array at length m
5  |   initialize any T_i to be an empty linked list.
6  |   h ← sample uniformly random from universal hash family H
7  |   for x ∈ x_1, x_2..x_n do
8  |   |   T_{h(x)}.insert(x)
9  |   |   T_{h(x)}.size = T_{h(x)}.size +1
10 |   end
11 |   if Σ_i T_{h(i)}.size^2 ≥ μ then
12 |   |   toomanycollisions ← True
13 |   end
14 end
15 let H be an array at length m
16 for i ∈ [m] do
17 |   T_i, h_i ← hash the elements in T_i using
18 |       perfect hashing.
19 end
20 return T, h
```

**Algorithm 5:** perfect-hashing-linear-space($x_1, x_2, ..x_n$)

linear, which implies that the expected number of rounds is constant.

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

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 $i$th table, which is equivalent to counting the number of $x \neq y$ such that $h(x) = h(y) = i$. Thus,

$$\begin{aligned}\mathbf{E}\left[\sum_i n_i^2\right] &= \mathbf{E}\left[\sum_i 2\binom{n_i}{2} + n_i\right] = 2 \cdot \mathbf{E}\left[\text{collisions}\right] + \mathbf{E}\left[\overbrace{\sum_i n_i}^{n}\right] \\ &= 2 \cdot \binom{n}{2}\frac{1}{m} + n\end{aligned}$$

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