**COMPSCI 330: Design and Analysis of Algorithms** September 28, 2017

# Lecture 10: Hashing

*Lecturer: Rong Ge* *Scribe: Yumin Zhang*

## 1 Hashing

Hashing is a technique to implement the map and set data-structures. The operations of a map includes

1. **Insert(key, value)** Store the (key, value) pair in the data-structure.
2. **Find(key)** If there was a pair (key, value) stored, return value, otherwise return NULL.
3. **Delete(key)** Remove the (key, value) pair associated with the key.

We can think of the map data-structure as a dictionary, where key will be words in the vocabulary, and value will be the explanations.

Our goal in this lecture is to design a data-structure for map, such that all the operations take $O(1)$ expected time, and the amount of space used is proportional to the number of (key, value) pairs stored.

### 1.1 Naïve implementations of map

There are two simple ways to implement a map.

**Linked List.** It is possible to store all the (key, value) pairs in the linked list. If there are $n$ pairs, the space required is $O(n)$, which is optimal. However, the problem of a linked list is that **Find** operation will take $O(n)$ time.

**Large Array.** If the set of keys is integers, we can also just allocate a very large array $a[]$, and directly store $a[key] = value$. This way the running times of all operations are $O(1)$. However, this array might be much larger than we need.

### 1.2 Hashing

One way to implement hashing is to combine the two naïve implementations, and get the best of both worlds.

We will allocate an array of size $m$ (which is usually a constant times $n$, the number of pairs). We will also have a function $f$ that maps keys to $\{0, 1, 2, ..., m-1\}$. We call this function the hash function, and it should be as "random" as possible.

Each entry in the array is going to be a pointer to a linked list. Initially all the linked lists are empty.

Now, to Insert a (key, value) pair, we insert the (key, value) pair to the linked list at $a[f(key)]$. To find a (key, value) pair, we look for the key value in the linked list at $a[f(key)]$. Ideally, because we have $m$ linked lists, none of these linked lists are very long, which makes all operations efficient.

**Collisions.** In the ideal case, for all pairs of keys $x, y$, we have $f(x) \neq f(y)$, then all the linked lists have at most one (key, value) pair. However, this is impossible, because the hash function maps a large set of keys to a small range $\{0, ..., m-1\}$, and by pigeon-hole principle there must be many repetitions. When $f(x) = f(y)$ we say that is a collision. Collisions are bad for hash tables - when there are many collisions some of the linked lists can be very long and the operations will take more time. Next we will see what we can do to minimize the number of collisions.

### 1.3 Attempt: Fixed Hash Function

For an array $a$, denote as a[m], whose size is m. A fixed hash function would be $f(i) = i \ mod \ m$. If the set is $S =\{$ 5,m+5,2m+5, ..., nm+5 $\}$ $for\ all\ i \in S$,. In this case, we can see all elements in the set has mapped to the same key which will cause collision. The running time of the data-structure will be $O(n)$ per operation.

In fact, for all fixed hash functions, it is possible to construct such a worst-case example, so in general we do not want to use a fixed hash function.

### 1.4 Solution: Random Hash Functions

As we see the problem from the fixed hash function, we could use a family of random function.

**When do we make the random choice.** The random hash function $f$ is chosen at the *beginning* of the algorithm. After that it is fixed for all the operations. That is, after the function is chosen, if we call $f(5)$ twice, it should return the same result. However, if we run the entire algorithm again on the same data-set, the function will be chosen randomly, and $f(5)$ can be of a different value.

**How to choose a random hash function.** For every integer i, we choose $f(i)$ independently at random from $\{0, 1, 2, \ldots, m-1\}$ Note that function is chosen at the beginning, and fixed for later use. Ideally, we want to choose a totally random function. However, we cannot store all the $f(i)$ values for a totally random function.

### 1.5 Hash Function by Modular Arithmetic

Our goal is to construct a hash function that is random enough to avoid collision but does not take that much of space. In other words, construct a family of hash function $f$ such that for $x \neq y$, we have

$$Pr_{f \sim F}[f(x) = f(y)] = \frac{1}{m}$$

**Recap the modular arithmetic.** For a prime number p, we only consider numbers $\{0, 1, 2, \ldots, p-1\}$. We can do addition, subtraction, and multiplication in the usual way and take $mod\ p$ at the end.

**Inverse.** For any integer $0 < x < p$, there is an integer $0 < y < p$ such that $xy \equiv 1 (mod\ p)$. An example would be p = 7, x = 2, then y = 4. We call $y = x^{-1}$. Note inverse can be computed efficiently.

**Problem.** Pick a prime number p, construct a hash family with $p^2$ functions. For every a,b in $\{0, 1, 2, \ldots, p-1\}$, we have $f_{a,b}(x) = ax + b \ (mod\ p)$

**Claim.** For every x, y ($x \neq y$), any two numbers u, v in $\{0, 1, 2, \ldots, p-1\}$ , we have

$$Pr_{a,b}[f_{a,b}(x) = u, f_{a,b}(y) = v] = \frac{1}{p^2}$$

**Proof.** In order to have $f_{a,b}(x) = u, f_{a,b}(y) = v$, we have the following

$$\begin{cases} ax + b \equiv u \ (mod\ p) \\ ay + b \equiv v \ (mod\ p) \end{cases}$$

Since x,y,u,v are known to us, all we need to solve for is a, and b. With two unknowns and two equations, the system of equation above has a unique solution, as

$$a = \frac{u - v}{x - y},\ b = u - ax = v - ay$$

Since there are $p^2$ choices of a, b, and 1 unique solution of the equation,

$$Pr_{a,b}[f_{a,b}(x) = u, f_{a,b}(y) = v] = \frac{1}{p^2}$$