

## CSCI 104L Lecture 17: Hash Tables

**Question 1.** A company has assigned a unique 3-digit ID to each of its 1000 employees. We want to design a data structure so that you can input an employee ID and quickly bring up their employee record. How should we implement this?

**Question 2.** Does something similar work if we want to do this from USC student ID to student records? Why or why not?

**Question 3.** What if we want to store the English dictionary as a set of strings, so we can quickly look up if something is a word or not. Can we do something like this for non-integer keys?

A **hash function** takes a valid input (in the case of the last question, a word in the English language) and outputs the entry in the array to store it. To be a good hash function it must:

- be efficient to calculate
- distribute the inputs well
- be consistent

**Question 4.** Is  $h(k) = 0$  a good hash function? Why or why not?

**Question 5.** Is  $h(k) = k \% m$ , where  $m$  is the size of the table, a good hash function? Why or why not?

**Question 6.** Is  $h(k) =$  a random integer between 0 and  $m - 1$ , where  $m$  is the size of the table, a good hash function? Why or why not?

The goal is to design a hash function where the probability of collision is  $\leq \frac{1}{m}$ . Any “good” hash function that satisfies this is called a “Universal Hash Function.”

**Question 7.** What do we mean when we say that a good hash function is pseudo-random?

**Question 8.** How can a good hash function be used to store passwords?

**Question 9.** Suppose we make a hash table to implement the set or map ADT. What is the *worst case* running time for create, insert, delete, contains?

**Question 10.** Suppose we make a hash table to implement the set or map ADT, and use a Universal Hash Function. What is the *expected* running time for create, insert, delete, contains?

Of course, for that to be useful, we need a Universal Hash Function. Here you go: assume that all inputs are base  $p$ , for some prime  $p$ .  $p$  will be the size of our Hash Table. Now, English words are base 26, but we can translate them to base  $p$  quite easily. So an English word will be  $w = w_1w_2\dots w_x, 0 \leq w_i < p$ .

Suppose that the longest english word has length  $x$ . Then we will choose a random number (base  $p$ )  $a = a_1a_2\dots a_x$ . We choose this random number ONCE when we create our hash table, and then keep that random number until we delete the hash table. The hash function is then this:

$$h(w) = (\sum_i a_i w_i) \bmod p.$$

## Collision Resolution

Apart from *chaining*, another way to handle collisions is *probing*, wherein we just have an array of either key/value pairs (no linked lists). There are many variants.

In *linear probing*, if  $h(k) = i$  and  $A[i]$  is taken, we try  $A[i + 1]$  and then  $A[i + 2]$  and so on. If you reach the end of the array, you loop back to the beginning.

That is,  $h(k, i) = (h(k) + i) \% m$ , where  $i$  is the number of failed inserts, and  $m$  is the size of our hash table.

**Question 11.** What problems can you see arising when using linear probing?

## Quadratic Probing

As before, if  $h(k) = i$  and  $A[i]$  is taken, try somewhere else. Except now,  $A[i + 1]$  is followed by an attempt at  $A[i + 4]$ , then  $A[i + 9]$  and so on. As before, loop back to the beginning if you hit the end of the array.

That is,  $h(k, i) = (h(k) + i^2) \% m$ .

Two different items mapped to  $i$  and  $i + 1$  respectively will follow very different paths. We don't get *primary clumping* as we did for Linear Probing.

*Secondary clumping* will only occur amongst all of the items which were originally mapped to index  $i$ . Note that chaining has this form of clumping as well.