CMSC 420: Lecture 10
Hashing - Basic Concepts and Hash Functions

Hashing: We have seen various data structures (e.g., binary trees, AVL trees, splay trees, skip lists) that can perform the dictionary operations `insert()`, `delete()` and `find()`. We know that these data structures provide $O(\log n)$ time access. It is unreasonable to expect any type of comparison-based structure to do better than this in the worst case. Using binary decisions, there is a lower bound of $\Omega(\log n)$ (and more precisely, $1 + \lfloor \lg n \rfloor$) on the worst case search time.

Remarkably, there is a better method, assuming that we are willing to give up on the idea of using comparisons to locate keys. The best known method is called *hashing*. Hashing and its variants support all the dictionary operations in $O(1)$ (i.e. constant) expected time. Hashing is considered so good, that in contexts where just these operations are being performed, hashing is the *method of choice*. The price we pay is that we cannot perform dictionary operations based on search order, such as range queries (finding the keys x such that $x_1 \leq x \leq x_2$) or nearest-neighbor queries (find the key closest to a given key x).

The idea behind hashing is very simple. We have a table of given size m, called the *table size*. We will assume that m is at least a small constant factor larger n. We select a *hash function* $h(x)$, which is an easily computable function that maps a key x to a "virtually random" index in the range `[0..m-1]`. We then attempt to store x (and its associated value) in index $h(x)$ in the table. Of course, it may be that different keys are mapped to the same location. Such events are called *collisions*, and a key element in the design of a good hashing system how collisions are to be handled. Of course, if the table size is large (relative to the total number of entries) and the hashing function has been well designed, collisions should be relatively rare.

Hashing is quite a versatile technique. One way to think about hashing is as a means of implementing a *content-addressable array*. We know that arrays can be addressed by an integer index. But it is often convenient to have a look-up table in which the elements are addressed by a key value which may be of any discrete type, strings for example or integers that are over such a large range of values that devising an array of this size would be impractical. Note that hashing is not usually used for continuous data, such as floating point values, because similar keys 3.14159 and 3.14158 may be mapped to entirely different locations.

There are two important issues that need to be addressed in the design of any hashing system, the *hash function* and the method of *collision resolution*. Let's discuss each of these in turn.

Hash Functions: A good hashing function should have the following properties:

- It should be *efficiently computable*, say in constant time and using simple arithmetic operations.
- It should produce *few collisions*. Two additional aspects of a hash function implied by this are:
  - It should be a function of *every bit* of the key (otherwise keys that differ only in these bits will collide)
  - It break up (scatter) naturally occuring *clusters* of key values.

As an example of the last rule, observe that in writing programs it is not uncommon to use very similar variables names, "`temp1`", "`temp2`", and "`temp3`". It is important such similar names be mapped to very different locations in the *hash output space*. By the way, the origin of the name "hashing" is from this mixing aspect of hash functions (thinking of "hash" in food preparation as a mixture of things).

We will think of hash functions as being applied to *nonnegative integer keys*. Keys that are not integers will generally need to be converted into this form (e.g., by converting the key into a bit string, such as an ASCII or Unicode representation of a string) and then interpreting the bit string as an integer. Since the hash function's output is the range $[0..m-1]$, an obvious (but not very good) choice for a hash function is:

$$h(x) \;=\; x \bmod m.$$

This is called *division hashing*. It satisfies our first criteria of efficiency, but consecutive keys are mapped to consecutive entries, and this is does not do a good job of breaking up clusters.

Some Common Hash Functions: Many different hash functions have been proposed. The topic is quite deep, and we will not claim to have a definitive answer for the best hash function. Here are three simple, commonly used hash functions:

Multiplicative Hash Function: Uses the hash function

$$h(x) \;=\; (ax) \bmod m,$$

where a is a *large prime number* (or at least, sharing no common factors with m).

Linear Hash Function: Enhances the multiplicative hash function with an added constant term

$$h(x) \;=\; (ax + b) \bmod m.$$

Polynomial Hash Function: We can further extend the linear hash function to a polynomial. This is often handy with keys that consist of a sequence of objects, such as strings or the coordinates of points in a multi-dimensional space.

Suppose that the key being hashed involves a sequence of numbers $x = (c_0, c_1, \ldots, c_{k-1})$. We map them to a single number by computing a polynomial function whose coefficients are these values. For example, if the c_i's are characters of a string, we might convert each to an integer (e.g., using Java's function `Character.getNumericValue(c[i])`, which returns the character's Unicode value as an integer) and then for some fixed value p (which you as the hash function designer pick) compute the polynomial

$$h(x_0, \ldots, x_n) \;=\; \left(\sum_{i=0}^{k-1} c_i p^i\right) \bmod m$$

For example, if $k = 4$ and $p = 37$, the associated polynomial would be $c_0 + c_1 37 + c_2 37^2 + c^3 37^3$.

You might wonder whether we can efficiently compute high-order polynomial functions. A useful algorithm for computing polynomials is called *Horner's rule*. The idea is to compute the polynomial through nested multiplications. To see how it works, observe that the above polynomial could be expressed equivalently as

$$c_0 + c_1 37 + c_2 37^2 + c^3 37^3 \;=\; ((c_3 \cdot 37 + c_2) \cdot 37 + c_1) \cdot 37 + c_0.$$

Using this idea, the polynomial hash function could be expressed in Java as

Polynomial hash function with Horner's rule

```
public int hash(String c, int m) {  // polynomial hash of a string
    final int P = 37;               // replace this with whatever you like
    int hashValue = 0;
    for (int i = c.length()-1; i >= 0; i--) { // Horner's rule
        hashValue = P * hashValue + Character.getNumericValue(c.charAt(i));
    }
    return hashValue % m;           // take the final result mod m
}
```

Randomization and Universal Hashing: Any deterministic hashing scheme runs the risk that we may (very rarely) come across a set keys that behaves badly for this choice. As we have seen before, one way to evade attacks by a clever adversary is to employ *randomization*. Any given hash function might be bad for a particular set of keys. So, it would seem that we can never declare that any one hash function to be "ideal."

One way to approach this conundrum is to flip the question on its head. Rather than trying to determine the chances that a fixed hash function works for a random set of keys, let us instead fix two keys x and y, say, and then select our hash function h at random out of large bag of possible hash functions. Then we ask the question, what is the probability that $h(x) = h(y)$, given that the choice of h is random. Since there are m table entries, a probability of $1/m$ would be the best we could hope for.

This gives rise to the notion of *universal hashing*. A hashing scheme is said to be *universal* if the hash function is selected randomly from a large class of functions, and the probability of a collision between any two fixed keys is $1/m$.

There are many different universal hash functions. Let's consider one simple one (which was first proposed by the inventors of universal hashing, Carter and Wegman). First, let p be any prime number that is chosen to be larger than any input key x to the hash function. Next, select two integers a and b at random where

$$a \in \{1, 2, \ldots, p-1\} \qquad \text{and} \qquad b \in \{0, 1, \ldots, p-1\}.$$

(Note that $a \neq 0$.) Finally, consider the following linear hash function, which depends on the choice of a and b.

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

As a and b vary, this defines a *family* of functions. Let H_p denote the class of hash functions that arise by considering all possible choices of a and b (subject to the above restrictions). The following theorem shows that H_p is a universal hashing system by showing that the probability that two fixed keys collide is $1/m$. The proof is not terribly deep, but it involves some nontrivial modular arithmetic. We present it for the sake of completeness.

Theorem: Consider any two integers x and y, where $0 \leq y < x < p$. Let $h_{a,b}$ be a hash function chosen uniformly at random from H_p. Then the probability that $h_{a,b}(x) = h_{a,b}(y)$ is at most $1/m$.

Proof: (Optional) Let us select a and b randomly as specified above and let $h = h_{a,b}$. Observe that $h(x) = h(y)$ if and only if the two values $(ax + b) \bmod p$ and $(ay + b) \bmod p$ differ from each other by a multiple of m. This is equivalent to saying that there exists an integer i, where $|i| \leq (p-1)/m$, such that:

$$(ax + b) \bmod p = (ay + b) \bmod p + i \cdot m.$$

Because p is prime, we can express this equivalently as

$$ax + b \equiv (ay + b) + i \cdot m \pmod{p},$$

where $0 \leq i \leq (p-1)/m$. Subtracting, we have

$$a(x - y) \equiv i \cdot m \pmod{p}.$$

Since $y < x$, their difference $x - y$ is nonzero and (since p is prime) $x - y$ has an inverse modulo p. That is, there exists a number q such that $(x-y) \cdot q \equiv 1 \pmod{p}$. Multiplying both sides by q, we have

$$a \equiv i \cdot m \cdot q \pmod{p}$$

By definition of our hashing system, are $p - 1$ possible choices for a. By varying i in the allowed range, there are $\lfloor (p-1)/m \rfloor$ possible nonzero values for the right-hand side. Thus, the probability of collision is

$$\frac{\lfloor (p-1)/m \rfloor}{p-1} \leq \frac{(p-1)/m}{p-1} = \frac{1}{m},$$

as desired.

Like the other randomized structures we have seen this year, universal hash functions are both simple and provide good guarantees on the expected-case performance of hashing systems. We will pick this topic up in our next lecture, focusing on methods for collision resolution, under the assumption that our hashing function has a low probability of collisions.