

CSC 378 Lecture 6

June 21, 2001

For another reference, see chapter 12 of the textbook (CLR).

1 Examples of Hash Functions

Recall the definition of simple uniform hashing: if A_i is the event (subset of U) $\{k \in U \mid h(k) = i\}$, then

$$\Pr(A_i) = \sum_{k \in A_i} \Pr(k) = 1/m.$$

Basically, the hash table gets evenly used for whatever distribution of keys we are dealing with. The problem is that we often don't know the distribution of keys before we see them. So how can we choose a good hash function?

For uniformly distributed keys in the range 1 through K (for large K), the following methods come close to simple, uniform hashing:

The division method: First choose a natural number m . Then, the hash function is just

$$h(k) = k \bmod m.$$

One advantage here is that computing $h(k)$ is very fast (just one division operation). But m has to be chosen with some care. If $m = 2^p$, then $h(k)$ is just the p lowest bits of k (see example 1). Instead, m is usually chosen to be a prime not close to any power of 2.

- **Example:** Most compilers or interpreters of computer programs construct a symbol table to keep track of the identifiers used in the input program. A hash table is a good data structure for a symbol table: identifiers need to be inserted and searched for quickly. We would like to use the division method for hashing, but first we need to turn the identifiers (strings of text) into positive integers. We can do this by considering each string to be a number in base 128 (if there are 128 text characters). Each character x can be represented by a number from 1 through 128 denoted $num(x)$. Then, a string of characters $x_n x_{n-1} \dots x_1$ can be represented uniquely by the number $\sum_{i=1}^n num(x_i)(128)^{i-1}$. For our choice of m here, we definitely want to avoid powers of 2, especially powers of 128. If m is 128^3 , for instance, then any two identifiers that share the same last three letters will hash to the same entry in the table. If the program is computing a lot of maximum values, for instance, then many of the variable names may end in "max" and they will all collide in the hash table, causing longer search times if we use chaining.

The multiplication method: Another way to hash natural numbers is just to scale them to something between 0 and $m - 1$. Here we choose m (often a power of 2 in this case) and a real

number A (often the fractional part of a common irrational number, such as the golden ratio: $(\sqrt{5} - 1)/2$). We then compute

$$h(k) = \lfloor m \times \text{fract}(kA) \rfloor,$$

where $\text{fract}(x)$ is the fractional part of a real number x .

2 Adversaries and Universal Hash Functions

Whenever we limit ourselves to one specific hash function, we are susceptible to attack from an *adversary*. An adversary is the most malicious and knowledgeable user of the program we are designing. He or she always uses the worst input for our algorithms. In this setting, that means the adversary would try to create a lot of collisions, since that increases the search time for many elements.

For instance, if we use the hash function $h(k) = k \bmod p$, for some prime p , and our universe of keys are the numbers $\{1, \dots, K\}$, where $K > np$, then the adversary might choose elements with keys $p, 2p, 3p, \dots, np$. When we try to hash these elements, they all get inserted in the 0 entry of the hash table, instead of being distributed across the table. The situation is analogous to Quicksort when we chose the first element as the pivot. While Quicksort handled the average list quickly, it was easy to find a list (the sorted list, for example), that would slow it down.

Similarly to the Quicksort situation, however, we can introduce some randomness into our program so that no single input will definitely be bad for the hash table. We will start with a class of hash functions \mathcal{H} . On each execution of our program, after the user chooses the elements he or she wants stored in our hash table, we will choose a hash function h randomly from \mathcal{H} and use it for the whole execution (we assume that the hash table isn't *persistent*: that is, it doesn't survive from one execution of the program to another). Now the adversary doesn't know which hash function we are going to use, and we will see that he or she can't design a bad input.

The class \mathcal{H} must have some special properties, however. Let's say the elements of \mathcal{H} are functions from the universe U to $\{0, \dots, m - 1\}$. Then \mathcal{H} is called *universal* if, for any two $x, y \in U$ such that $x \neq y$,

$$\Pr_{h \in \mathcal{H}}(h(x) = h(y)) = 1/m.$$

In words: take two elements with different keys and hash them with a random hash function from \mathcal{H} . Then the probability (over random choices of the hash function) that they collide is $1/m$. This is the lowest probability you could hope for since, if you just chose two random numbers from $\{0, \dots, m - 1\}$, the probability that they would be equal is $1/m$.

Now we show that using a universal \mathcal{H} makes for a good expected running time for SEARCH. Let x_1, \dots, x_n be the key-values of the elements we want to store. We choose h randomly from \mathcal{H} and then hash all the elements with h . Now we want to search for some arbitrary key x in the hash table.

Let $C(h)$ be the number of unsuccessful comparisons performed when searching for x (this is basically the running time of SEARCH(S, x) as a function of which random h we choose). Let $C_i(h)$ be 1 if $x \neq x_i$ and $h(x) = h(x_i)$ (that is, if x collides with x_i). Otherwise, let $C_i(h)$ be 0. Notice that if $x = x_i$, then the expected value of C_i , $E(C_i)$ is 0; in fact, $C_i(h) = 0$ everywhere. If $x \neq x_i$, then $E(C_i) = 1 \times \Pr(h(x) = h(x_i)) = 1/m$, by the definition of universal. Finally, notice that $C(h) \leq \sum_{i=1}^n C_i(h)$, since $\sum C_i(h)$ is basically the number of elements in the same entry as x in the hash table, and at worst we will have to search each of them to find x . Now we can calculate

$E(C)$, the expected running time:

$$E_{h \in \mathcal{H}}(C) \leq E\left(\sum_{i=1}^n C_i\right) \tag{1}$$

$$= \sum_{i=1}^n E(C_i) \tag{2}$$

$$\leq \sum_{i=1}^n 1/m \tag{3}$$

$$= n/m \tag{4}$$

$$= a \tag{5}$$

So the expected running time is at most the a , the load factor of the hash table, regardless of the input.

3 Constructing a Universal Class of Hash Functions

We saw that to get this input-independent fast expected running time, we need to have a universal class of hash functions. We will show one way to construct them:

1. Choose a prime number m .
2. Choose a “word size” $w \leq m$. Let’s say $w = 2^c$ for some value c . We will be partitioning the key-values into words. For example, for a key x , let x_0 be the number represented by the lowest c bits of x , let x_1 be the number represented by the next c bits of x , etc.
3. Let $r + 1$ be the maximum number of words needed to represent any x in the universe U .

Now, to choose a random hash function, we choose a_0, a_1, \dots, a_r randomly and independently from the set $\{0, 1, \dots, m - 1\}$. Let a be the collection of a_0, \dots, a_r . We then compute the hash function:

$$h_a(x) = (a_0x_0 + a_1x_1 + \dots + a_rx_r) \bmod m.$$

The class \mathcal{H} is the set of h_a for all possible a ’s. Since a consists of $r + 1$ elements chosen from a set of size m , there are m^{r+1} possible a ’s.

We want to show that \mathcal{H} is universal. Let $x, y \in U$ be two different key-values. We want to show that $\Pr_{h \in \mathcal{H}}(h(x) = h(y)) = 1/m$. If $x \neq y$, then they must differ in at least one particular word: say $x_j \neq y_j$. Then, for random h_a in \mathcal{H} , $h_a(x) = h_a(y)$ if and only if

$$\left(\sum_{i=0}^r a_i x_i\right) \bmod m = \left(\sum_{i=0}^r a_i y_i\right) \bmod m.$$

This is the same as saying

$$\left(\sum_{i=0}^r a_i (x_i - y_i)\right) \bmod m = 0.$$

If we move every term except the j th to the right side, then we get

$$a_j(x_j - y_j) \bmod m = \left(-\sum_{i \neq j} a_i(x_i - y_i)\right) \bmod m.$$

Technically, since m is prime, there is a unique solution for a_j . That is, given $a_0, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_r$, there is only one specific value for a_j in $\{0, \dots, m-1\}$ that will make this equation true. But the a_i 's were all chosen randomly and independently of each other. So the probability that a_j was chosen to be this one magic value is $1/m$. So the probability that $h_a(x) = h_a(y)$ is $1/m$.