CSE 241 Algorithms and Data Structures 9/30/2015

Choosing Hash Functions

There are two criteria to consider when choosing hash functions for your hash table.

1. You want functions that approximate the assumptions of Simple Uniform Hashing, i.e. that map roughly equal numbers of keys to each slot in the table and are unlikely to deviate from this behavior for keys encountered in practice.

2. If your hash table is open-addressed, you want to ensure that your slot sequence for every key is of maximum length, eventually touching every slot in the table.

1 Multiplicative Hashing

A practical strategy for generating efficient, fairly uniform hash functions is *multiplicative hashing*. The form of a multiplicative hash function is

$$h(k) = \lfloor m(\phi \cdot k - \lfloor \phi \cdot k \rfloor) \rfloor \,,$$

where m is the table size, and ϕ is a real number between 0 and 1. Note that the inner parenthesized expression computes the fractional part of $\phi \cdot k$, which is then multiplied by m to yield a value in the half-open interval $[0, m)$. Taking the floor yields an integer between 0 and $m - 1$.

This strategy works well if multiplication by ϕ does a good job of "scrambling" the bits of k. To ensure good scrambing, ϕ should have a substantial number of 1 bits but should not exhibit a highly regular bit pattern; irrational values are best. Of course, ϕ should also not be too small – very small values will map a lot of keys to low-numbered slots, especially slot 0. I recommend that ϕ be at least $1/2$.

There are uncountably many irrational numbers between $1/2$ and 1. Knuth recommends that you use $\phi = (\sqrt{5} - 1)/2$, the Golden Ratio. Other easy-to-compute values that might work include $\sqrt{2}/2$, $\sqrt{3} - 1$, and so forth.

2 Ensuring Maximum-Length Slot Sequences

Let h_1 and h_2 be base and step hash functions for a table of size m. As we said in class, the ith slot may be computed as

$$s_i(k) = [s_{i-1}(k) + h_2(k)] \bmod m,$$

where $s_0(k) = h_1(k) \bmod m$. How long can this sequence of slot numbers go without repeating?

There are only m possible values for the slot number, so if we keep computing successive slots, we will repeat a slot after *no more* than m steps. Note that repetition is fatal to our efforts to find new slots for a key! Because each slot number depends only on its predecessor, if (say) $s_j(k) = s_i(k)$, then $s_{j+t}(k) = s_{i+t}(k)$ for all $t > 0$. In other words, once we repeat a slot, we will cycle through previously computed slot values in the same order forever, without looking at any new slots.

For poor choices of $h_2(k)$ and m, the slot sequence may be much shorter than m; in the extreme case, if $h_2(k) = \ell m$ for any $\ell \geq 0$, the slot sequence has length one! How can we avoid such bad cases? Let's look more carefully at the circumstances under which a slot repeats.

By definition of the slot sequence, $s_j(k) = s_i(k)$ precisely when

$$h_1(k) + jh_2(k) = h_1(k) + ih_2(k) \pmod{m}.$$

This equality holds iff

$$(j - i)h_2(k) = 0 \pmod{m},$$

or equivalently iff

$$(j - i)h_2(k) = \ell m$$

for some $\ell \geq 0$.

Assume that $h_2(k)$ is *relatively prime* to m; that is, it has no prime factors in common with m. Then the last equality holds iff $j - i$ is itself a multiple of m. In other words, a slot number cannot repeat until a full m steps after it first occurs. Conclude that

> If the step hash function h_2 always produces values that are relatively prime to the table size m, then the slot sequence for any key is guaranteed to have length m before its first repetition.

How can you ensure that $h_2(k)$ is relatively prime to m? Here are two strategies.

1. Make m a prime number, and ensure that $h_2(k)$ always returns a nonzero value that is not a multiple of m. You can do this by reducing $h_2(k)$ to the range $[0, m - 1]$ (by taking its value modulo m), then explicitly checking if it is zero and setting it to some nonzero value if so.

2. Make m a power of two, and ensure that $h_2(k)$'s result is always an odd number. You can do this by checking if $h_2(k)$ is even and adding one if so.

The second approach allows greater efficiency in computing hash values, since computing a mod b is a much more expensive operation on modern processors when b is arbitrary than when it is a power of 2. However, it limits you to only $m/2$ possible step sizes, which increases the likelihood of a collision. In contrast, the first approach, while slower to compute, permits the full range of $m - 1$ possible step sizes. Which choice is more efficient overall should probably be determined empirically for your application.