Double Hashing

Double hashing attempts to combine the best thing about of linear probing (each probing sequence contains all addresses) with the strong point of quadratic probing (reduced primary clustering). The technique is simple: we include a second hash function h"(k), and define

$$h(k,i) = (h'(k) + i * h''(k)) \mod m$$

Double hashing is effectively a generalization of linear probing, except that instead of having a fixed "step size" that determines how far we jump forward in the hash table on each iteration (in linear probing, the step size is 1), we use the key itself to determine the step size. Since the key is used in two different hash functions to determine the initial address in the probing sequence *and* the step size, the probability that two keys will have exactly the same probing sequence is greatly reduced. This reduces both primary and secondary clustering.

Example:

Let m = 10, let $h'(k)$ = "sum of the digits of k", and let $h"(k) = k^2$

Consider the probing sequence for k_1 = 13

i	h(13,i)
0	4
1	3
2	2
3	1
...	...

Now consider the probing sequence for k_2 = 22

i	h(22,i)
0	4
1	8
2	2
3	6
...	...

Here we see that even though the probing sequences for k_1 and k_2 start in the same address (4) they are completely dissimilar after that.

Note that we now have potentially m^2 different probing sequences, as opposed to the m probing sequences we get with quadratic probing. It is easy to see that the use of $h''(k)$ reduces primary clustering (unless $h''(k) == 1$ for all k, which would be quite pointless). But what about the problem of missing empty addresses? We can see that in the example above, the probing sequence for $k_2 = 22$ will only try the even-numbered addresses in T. There is an even worse possibility: if $h''(k) == 0$ for some k then the probing sequence for that key will never move from the initial location given by $h'(k)$.

However, if we can ensure that $h''(k)$ and m are **relatively prime** for all k, each probing sequence will include all addresses (this is easy to prove based on material we studied in CISC-203) ... and this is quite easy to achieve: for example **if m is a power of 2, and $h''(k)$ is odd for all k**, then we satisfy the requirement.

Making sure $h''(k)$ is odd is also easy:

```
h''(k):
        x = "some computation based on k, such as k^2, or (k + p1)^p2 where
              p1 and p2 are primes, etc."
        if x % 2 == 1:
                return x
        else if x == m-1:   # this test ensures that we never return m
                return x-1
        else:
                return x+1
```

There are of course infinitely many other ways to ensure that $h''(k)$ and m are always relatively prime.

Double hashing is considered to be one of the most effective collision resolution methods in use.

Many discussions of hashing suggest that the table size m should always be prime because that reduces some of the potential for clustering when we do the "mod m" step of the hashing. However, as we have just seen, there are sometimes very good reasons for letting $m = 2^x$ for some integer x. Here is another good reason: if we can perform bit-level

operations on integers (which is easy in the "C" family of languages, as well as others), then computing $t \mod 2^x$ (where t is some integer) is trivial: we simply throw away all but the last x bits of t. This can accelerate the real time efficiency of our hashing function.