CS 312 Lecture 21
Hash functions

Hash functions

Hash tables are one of the most useful data structures ever invented. Unfortunately, they are also one of the most misused. Code built using hash tables often falls far short of achievable performance. There are two reasons for this:

- Clients choose poor hash functions that do not act like random number generators, invalidating the simple uniform hashing assumption.
- Hash table abstractions do not adequately specify what is required of the hash function, or make it difficult to provide a good hash function.

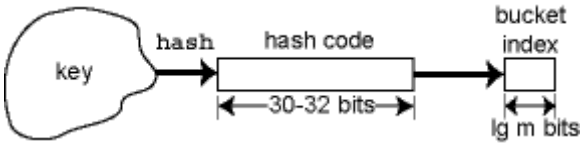
Clearly, a bad hash function can destroy our attempts at a constant running time. A lot of obvious hash function choices are bad. For example, if we're mapping names to phone numbers, then hashing each name to its length would be a very poor function, as would a hash function that used only the first name, or only the last name. We want our hash function to use all of the information in the key. This is a bit of an art. While hash tables are extremely effective when used well, all too often poor hash functions are used that sabotage performance.

Recall that hash tables work well when the hash function satisfies the simple uniform hashing assumption -- that the hash function should look random. If it is to look random, this means that any change to a key, even a small one, should change the bucket index in an apparently random way. If we imagine writing the bucket index as a binary number, a small change to the key should randomly flip the bits in the bucket index. This is called **information diffusion**. For example, a one-bit change to the key should cause every bit in the index to flip with 1/2 probability.

Client vs. implementer

As we've described it, the hash function is a single function that maps from the key type to a bucket index. In practice, the hash function is the composition of *two* functions, one provided by the client and one by the implementer. This is because the implementer doesn't understand the element type, the client doesn't know how many buckets there are, and the implementer probably doesn't trust the client to achieve diffusion.

The client function h_{client} first converts the key into an integer hash code, and the implementation function h_{impl} converts the hash code into a bucket index. The actual hash function is the composition of these two functions, $h_{client} \circ h_{impl}$:



To see what goes wrong, suppose our hash code function on objects is the memory address of the objects, as in Java. This is the usual choice. And suppose that our implementation hash function is like the one in SML/NJ; it takes the hash code modulo the number of buckets, where the number of buckets is always a power of two. This is also the usual implementation-side choice. But memory addresses are typically equal to zero modulo 16, so at most 1/16 of the buckets will be used, and the performance of the hash table will be 16 times slower than one might expect.

Measuring clustering

When the distribution of keys into buckets is not random, we say that the hash table exhibits **clustering**. It's a good idea to test your function to make sure it does not exhibit clustering with the data. With any hash function, it is possible to generate data that cause it to behave poorly, but a good hash function will make this unlikely.

A good way to determine whether your hash function is working well is to measure clustering. If bucket i contains x_i elements, then a good measure of clustering is $(\sum_i(x_i^2)/n)$ - α. A uniform hash function produces clustering near 1.0 with high probability. A clustering measure of $c > 1$ greater than one means that the performance of the hash table is slowed down by clustering. For example, if all elements are hashed into one bucket, the clustering measure will be n^2/n - α = n-α. If the clustering measure is less than 1.0, the hash function is spreading elements out more evenly than a random hash function would; not something you want to count on!

Unfortunately most hash table implementations do not give the client a way to measure clustering. This means the client can't directly tell whether the hash function is performing well or not. Hash table designers should provide some clustering estimation as part of the interface. Note that it's not necessary to compute the sum of squares of all bucket lengths; picking a few at random is cheaper and usually good enough.

The reason the clustering measure works is because it is based on an estimate of the **variance** of the distribution of bucket sizes. If clustering is occurring, some buckets will have more elements than they should, and some will have fewer. So there will be a wider range of bucket sizes than one would expect from a random hash function.

For those who have taken some probability theory: Consider bucket i containing x_i elements. For each of the n elements, we can imagine a random variable e_j, whose value is 1 if the element lands in bucket i (with probability $1/m$), and 0 otherwise. The bucket size x_i is a random variable that is the sum of all these random variables:

$$x_i = \sum_{j \in 1..n} e_j$$

Let's write $\langle x \rangle$ for the **expected value** of variable x, and $\mathrm{Var}(x)$ for the **variance** of x, which is equal to $\langle (x - \langle x \rangle)^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2$. Then we have:

$$\langle e_j \rangle = 1/m$$
$$\langle e_j^2 \rangle = 1/m$$
$$\mathrm{Var}(e_j) = 1/m - 1/m^2$$
$$\langle x_i \rangle = n\langle e_j \rangle = \alpha$$

The variance of the sum of independent random variables is the sum of their variances. If we assume that the e_j are independent random variables, then:

$$\mathrm{Var}(x_i) = n\ \mathrm{Var}(e_j) = \alpha - \alpha/m = \langle x_i^2 \rangle - \langle x_i \rangle^2$$
$$\langle x_i^2 \rangle = \mathrm{Var}(x_i) + \langle x_i \rangle^2$$
$$= \alpha(1 - 1/m) + \alpha^2$$

Now, if we sum up all m of the variables x_i, and divide by n, as in the formula, we should effectively divide this by α:

$$(1/n)\ \langle \sum x_i^2 \rangle = (1/\alpha)\langle x_i^2 \rangle = 1 - 1/m + \alpha$$

Subtracting α, we get $1 - 1/m$, which is close to 1 if m is large, regardless of n or α.

Now, suppose instead we had a hash function that hit only one of every c buckets. In this case, for the non-empty buckets, we'd have

$$\langle e_j \rangle = \langle e_j^2 \rangle = c/m$$
$$\langle x_i \rangle = \alpha c$$
$$(1/n)\ \langle \sum x_i^2 \rangle - \alpha = (1/n)(m/c)(\mathrm{Var}(x_i) + \langle x_i \rangle^2) = 1 - c/m + \alpha c$$
$$= 1 - c/m + \alpha(c\text{-}1)$$

If the clustering measure gives a value significantly greater than one, it is like having a hash function that misses a substantial fraction of buckets.

Designing a hash function

For a hash table to work well, we want the hash function to have two properties:

- **Injection**: for two keys $k_1 \neq k_2$, the hash function should give different results $h(k_1) \neq h(k_2)$, with probability m-1/m.
- **Diffusion** (stronger than injection): if $k_1 \neq k_2$, knowing $h(k_1)$ gives *no information* about $h(k_2)$. For example, if k_2 is exactly the same as k_1, except for one bit, then every bit in $h(k_2)$ should change with 1/2 probability compared to $h(k_1)$. Knowing the bits of $h(k_1)$ does not give any information about the bits of $h(k_2)$.

As a hash table designer, you need to figure out which of the client hash function and the implementation hash function is going to provide diffusion. For example, Java hash tables provide (somewhat weak) information diffusion, allowing the client hashcode computation to just aim for the injection property. In SML/NJ hash tables, the implementation provide only the injection property. Regardless, the hash table specification should say whether the client is expected to provide a hash code with good diffusion (unfortunately, few do).

If clients are sufficiently savvy, it makes sense to push the diffusion onto them, leaving the hash table implementation as simple and fast as possible. The easy way to accomplish this is to break the computation of the bucket index into three steps.

1. Serialization: Transform the key into a stream of bytes that contains all of the information in the original key. Two equal keys must result in the same byte stream. Two byte streams should be equal only if the keys are actually equal. How to do this depends on the form of the key. If the key is a string, then the stream of bytes would simply be the characters of the string.
2. Diffusion: Map the stream of bytes into a large integer x in a way that causes every change in the stream to affect the bits of x apparently randomly. There are a number of good off-the-shelf ways to accomplish this, with a tradeoff in performance versus randomness (and security).
3. Compute the hash bucket index as $x \bmod m$. This is particularly cheap if m is a power of two, but see the caveats below.

There are several different good ways to accomplish step 2: multiplicative hashing, modular hashing, cyclic redundancy checks, and secure hash functions such as MD5 and SHA-1.

Frequently, hash tables are designed in a way that doesn't let the client fully control the hash function. Instead, the client is expected to implement steps 1 and 2 to produce an integer **hash code**, as in Java. The implementation then uses the hash code and the value of m (usually not exposed to the client, unfortunately) to compute the bucket index.

Some hash table implementations expect the hash code to look completely random, because they directly use the low-order bits of the hash code as a bucket index, throwing away the information in the

high-order bits. Other hash table implementations take a hash code and put it through an additional step of applying an **integer hash function** that provides additional diffusion. With these implementations, the client doesn't have to be as careful to produce a good hash code,

Any hash table interface should specify whether the hash function is expected to look random. If the client can't tell from the interface whether this is the case, the safest thing is to compute a high-quality hash code by hashing into the space of all integers. This may duplicate work done on the implementation side, but it's better than having a lot of collisions.

Modular hashing

With **modular hashing**, the hash function is simply $h(k) = k \bmod m$ for some m (usually, the number of buckets). The value k is an integer hash code generated from the key. If m is a power of two (i.e., $m=2^p$), then $h(k)$ is just the p lowest-order bits of k. The SML/NJ implementation of hash tables does modular hashing with m equal to a power of two. This is very fast but the the client needs to design the hash function carefully.

The Java `Hashmap` class is a little friendlier but also slower: it uses modular hashing with m equal to a prime number. Modulo operations can be accelerated by precomputing $1/m$ as a fixed-point number, e.g. $(2^{31}/m)$. A precomputed table of various primes and their fixed-point reciprocals is therefore useful with this approach, because the implementation can then use multiplication instead of division to implement the mod operation.

Multiplicative hashing

A faster but often misused alternative is **multiplicative hashing**, in which the hash index is computed as $\lfloor m * \mathrm{frac}(ka) \rfloor$. Here k is again an integer hash code, a is a real number and frac is the function that returns the fractional part of a real number. Multiplicative hashing sets the hash index from the fractional part of multiplying k by a large real number. It's faster if this computation is done using fixed point rather than floating point, which is accomplished by computing $(ka/2^q) \bmod m$ for appropriately chosen integer values of a, m, and q. So q determines the number of bits of precision in the fractional part of a.

Here is an example of multiplicative hashing code, written assuming a word size of 32 bits:

```
val multiplier: Word.word = 0wx678DDE6F (* a recommendation by Knuth *)
  fun findBucket({arr, nelem}, e) (f:bucket array*int*bucket*elem->'a) =
    let
      val n = Word.fromInt(Array.length(arr))
      val d = (0wxFFFFFFF div n)+0w1
      val i = Word.toInt(Word.fromInt(Hash.hash(e)) * multiplier div d)
      val b = Array.sub(arr, i)
    in
      f(arr, i, b, e)
    end
```

Multiplicative hashing works well for the same reason that linear congruential multipliers generate apparently random numbers—it's like generating a pseudo-random number with the hashcode as the seed. The multiplier a should be large and its binary representation should be a "random" mix of 1's and 0's. Multiplicative hashing is cheaper than modular hashing because multiplication is usually considerably faster than division (or mod). It also works well with a bucket array of size $m=2^p$, which is convenient.

In the fixed-point version, The division by 2^q is crucial. The common mistake when doing multiplicative hashing is to forget to do it, and in fact you can find web pages highly ranked by Google that explain multiplicative hashing without this step. Without this division, there is little point to multiplying by a, because $ka \bmod m = (k \bmod m) * (a \bmod m) \bmod m$. This is no better than modular hashing with a modulus of m, and quite possibly worse.

Cyclic redundancy checks (CRCs)

For a longer stream of serialized key data, a cyclic redundancy check (CRC) makes a good, reasonably fast hash function. A CRC of a data stream is the remainder after performing a long division of the data (treated as a large binary number), but using exclusive or instead of subtraction at each long division step. This corresponds to computing a remainder in the field of polynomials with binary coefficients. CRCs can be computed very quickly in specialized hardware. Fast software CRC algorithms rely on accessing precomputed tables of data.

Cryptographic hash functions

Sometimes software systems are used by adversaries who might try to pick keys that collide in the hash function, thereby making the system have poor performance. **Cryptographic hash functions** are hash functions that try to make it computationally infeasible to invert them: if you know $h(x)$, there is no way to compute x that is asymptotically faster than just trying all possible values and see which one hashes to the right result. Usually these functions also try to make it hard to find different values of *x* that cause collisions. Examples of cryptographic hash functions are MD5 and SHA-1. Some attacks are known on MD5, but it is faster than SHA-1 and still fine for use in generating hash table indices.

Precomputing hash codes

High-quality hash functions can be expensive. If the same values are being hashed repeatedly, one trick is to precompute their hash codes and store them with the value. Hash tables can also store the full hash codes of values, which makes scanning down one bucket fast. In fact, if the hash code is long and the hash function is high-quality (e.g., 64+ bits of a properly constructed MD5 digest), two keys with the same hash code are almost certainly the same value. Your computer is then more likely to get a wrong answer from a cosmic ray hitting it than from a hash code collision.