Hash functions

Hash tables are one of the most useful data structures ever invented. Unfortunately, they are also one of the most misused. Today we will talk about hash functions. NOTE: you do not need to build your own hash function for PS5!

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. And recall that sometimes one does not know when writing a program what the input will be (hence, rapid prototyping!)

Perfect hashing and minimal perfect hashing

Sometimes you know there is a limited set of inputs. This allows you to do something amazing! Example: OCaml compiler.

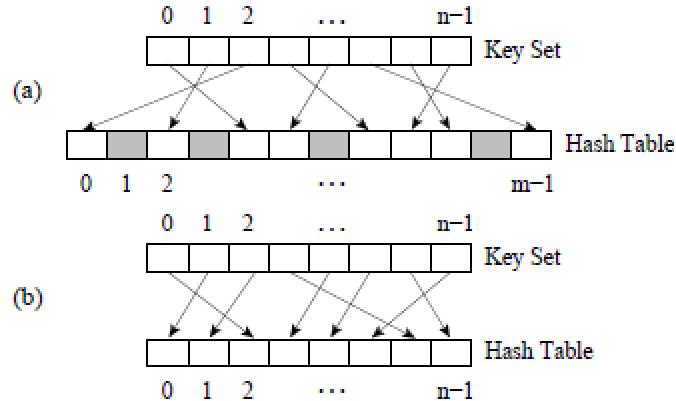


Figure 1.1: (a) Perfect hash function (b) Minimal perfect hash function.

How do you find a perfect hashing function, let alone a minimal one, for a given input set? If the hash table is big enough you can just search (or even just guess!)

There are lower bounds on a general algorithm. The bounds are actually on the SIZE of the hash function (think about this!)

Another interesting variant: preserving ordering in the hash function.

Back to regular hashing

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.

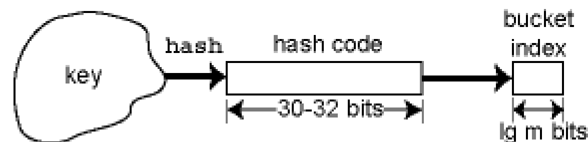
If we imagine writing the bucket index as a binary number, a small change to the key should randomly flip the bits in the bucket index. This is called **information diffusion**. For example, a one-bit change to the key should cause every bit in the index to flip with 1/2 probability.

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.

There are many ways to compute clustering, somewhat related to entropy (spread). Note that we want to know what it does on the real data!

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, but most don't. This is one reason why people often implement their own hash tables, especially in performance-critical applications.

Designing a hash function

For a hash table to work well for all applications, 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 high probability.
- **Diffusion** (stronger than injection, needed for some applications): 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.

Therefore the client-side hash function $h_{client}(k)$ is defined as $(h_{diff} \circ h_{serial})(k) \bmod m$, where$h_{diff}$ implements diffusion.

There are several different good ways to implement diffusion (step 2): multiplicative hashing, modular hashing, cyclic redundancy checks, and secure hash functions such as MD5 and SHA-1. They offer a tradeoff between collision resistance and performance.

Usually, 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 side 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.

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.

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; they are **collision-resistant**. Examples of cryptographic hash functions are MD5 and SHA-1. MD5 is not as strong as once thought, but it is roughly four times faster than SHA-1 and usually still fine for generating hash table indices.