Hashing

Out of all the data structures, the array has one unbeatable property: it allows $\Theta(1)$ time (i.e., constant) access and update. Unfortunately, it only allows small integers to be used as indices.

Wouldn't it be great to index an array with strings, like this:

```
update( arr, "John", 1141 );
...
val john's_room = sub( arr, "John" )
```

Hashing is a method that allows us to do just that, using just about any kind of data (strings, reals, records) as the index (or "key").

But strings *are* numbers!

They may not look like it, but strings are numbers too. Each character in a string can take on one of 256 different values. For example, the character `#"J"` has the value 74. An `#"o"` has the value 111, `#"h"` is 104, and `#"n"` is 110.

So, we could think of the string `"John"` as a number in base 256. Just as the number 123 in base 10 represents $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$, the string `"John"` can represent $74 \times 256^3 + 111 \times 256^2 + 104 \times 256^1 + 110 \times 256^0$, which we would write as 1.248.815.214 in base 10.

But what's the point of that?

Well, if we had an array with at least 1.248.815.214 elements, we could index into it with the string `"John"`!

But that would be a terrible waste, since most of the 1.248.815.214 or more entries in the array would be empty. If we want to store n strings, then we would like to use a structure of size $\Theta(n)$.

So, you can think of hashing as a technique for storing a subset K from a universe U in $\Theta(|K|)$ space rather than $|U|$ space, while still retaining $\Theta(1)$ access and update time.

Computing an index

Ok, say we want to store about 1000 strings in an array with about 1000 entries. We can't use the strings (as numbers) to index into the array, but we could try computing a suitable index from a string and use that instead. I.e., we want a function `hash: string->int` that returns a suitable index for a given string.

Given such a function, we could then write code like this:

```
update( arr, hash("John"), 1141 );
...
val john's_room = sub( arr, hash("John") )
```

This is called "perfect hashing".

Problems

The function `hash` has a tough job. It could be passed any string, and must try to find a different index to use for each one. Oh, and it has to run in $\Theta(1)$ time, or we may as well give up now.

Theorem: Such a `hash` function cannot be written.

Proof: Pretend `hash` has filled up 999 of the 1000 entries in the array. It must return the one remaining index for the next string it is given. But that string could be anything. If it succeeds, we try again with a different string. One way or the other, `hash` loses.

Such a `hash` function can only succeed if it is written with explicit knowledge of the strings that are to be passed in. That's asking too much.

A statistical solution

Forget about perfect hashing. We want a solution that works within $\Theta(1)$ bounds, and that gives us some room to move.

What happens if the strings we pass to `hash` are "random". I.e., when considered as numbers, they come from a uniformly distributed random variable.

`hash` can simply compute $n \bmod$ `arraySize`.

But it looks like we have replaced one problem with another. There is no longer any guarantee that `hash` will return a unique index for each string!

But that doesn't matter

We solve this problem by making each array entry (or "bin") hold a list of values (or "chain"). The `update` operation can insert the new value into the list, and `sub` can search the list.

This works just fine, *provided* the lists don't get too long.

They don't. Here's the statistical analysis.

The probability that k out of n strings fall into a particular slot is

$$\left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} \approx \frac{1}{ek!}$$

Let's have a look at the numbers

For 1000 elements, we have

k	prob. k collisions
0	37%
1	37%
2	18%
3	6%
4	1.5%
5	0.3%

I.e., if we place 1000 random numbers into 1000 bins, about 37% of the bins will be empty, 37% will have exactly one number, 18% will contain two numbers, and we would consider ourselves unlucky if there was a single bin with as many as 6 numbers.

Summary so far

If we have a random collection of strings, *then* the following code efficiently implements a hash table:

```
type 'a HashTable = (string * 'a) list Array.array
local
  open Array
  fun hash s = List.foldl (fn (c,h)=>h*256+ord c) 0 (String.explode s)
  fun matches s (k,_) = (k = s)
in
  fun hsub(ht, s) = List.find (matches s) (sub(ht, hash s mod length ht))
  fun hupdate(ht, s, v) = let
        val h     = (hash s) mod (length ht)
        val chain = sub(ht,h)
    in
      update(ht, h, (s,v)::List.filter (not o matches s) chain)
    end
end
```

Load factor

37% empty space and a maximum chain length of 5 suggests that we can get away with rather fewer bins if we wish. Or, to put it another way, there is no reason to stop adding elements once we reach 1000. The average chain length will grow in proportion, but it turns out the *maximum* chain length grows much more slowly.

Definition: When a hash table contains n bins and m elements, the *load factor* α is defined by

$$\alpha = \frac{m}{n}$$

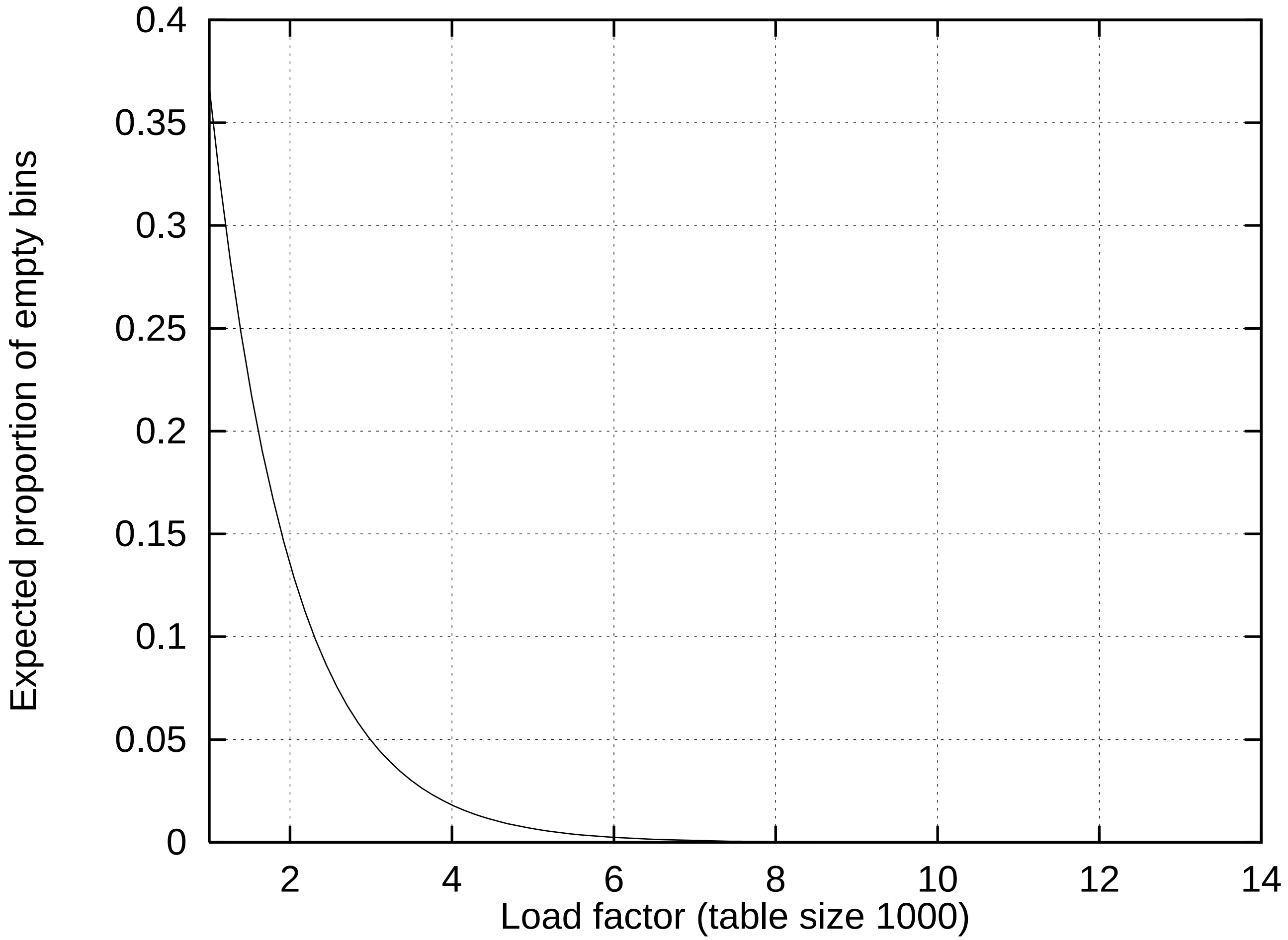
.

Behaviour as load factor increases

The first thing to go as the load factor increases is the empty bins. The probability that any particular bin will be empty after m elements have been added to a table of n bins is

$$\begin{aligned}\left(1 - \frac{1}{m}\right)^n &= \left(1 - \frac{\alpha}{n}\right)^n \\ &\approx e^{-\alpha}\end{aligned}$$

When $\alpha = 2$ (a normal load factor), about 13% of the bins will be empty. When $\alpha = 5$ (still not a high factor), around 0.7% of the bins will be left empty.


0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Expected proportion of empty bins
2
4
6
8
10
12
14
Load factor (table size 1000)

Chain length under load

The probability that k out of m strings fall into a particular slot in a table of n elements is

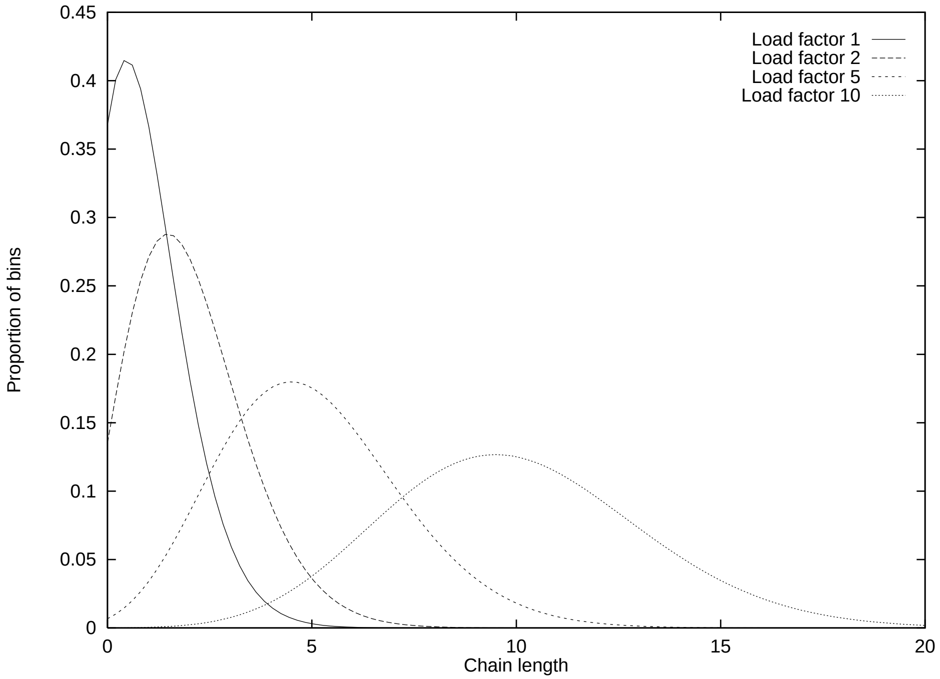
$$\left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{m-k} \binom{m}{k} \approx \frac{\alpha^k}{e^\alpha k!}$$

A look at the numbers

For 1000 elements load factor 2 (i.e., $m = 2000$), we have

k	prob. k collisions
0	13%
1	27%
2	27%
3	18%
4	9%
5	3%
6	1%
7	0.3%

Now, about 13% of the bins will be empty, 95% have 4 or fewer numbers, and we would consider ourselves unlucky if there was a single bin with as many as 8 numbers.



This graph shows how the expected distribution of chain lengths changes as the load factor increases. Observe that, while the average chain length is equal to the load factor, tail of the distribution moves more slowly, growing from around 5 to around 20 as the load factor increases ten-fold.

Average probe length (CLR, p225)

For a given load factor, α, the expected number of nodes examined in a successful search is approximately $1 + \frac{\alpha}{2}$. For an unsuccessful search, the number is $\frac{\alpha}{2}$.

These formulae can be used to predict the performance and design of a hash table. For example, if you expect about two thousand keys, and allocate a hash table with 400 bins, then you will be working with a load factor of around 5. The expected performance is around 2.5 probes (for successful or unsuccessful search) and around 3 unused hash table spaces (less than 1%).

Observe how “perfect” hashing would not make a very large improvement—half a probe at best.

What if the strings are not random?

Well, of course they're not, and the method we have just described *collapses* as soon as it is given anything other than a randomly distributed input.

Consider: people's names are sequences of letters taken from the alphabet. That makes for 29 possibilities for each letter, not 256. For names of up to 10 letters, only 29^{10} of the 256^{10} possible character strings can actually appear. That's about 0.0000003 percent. No distribution that is restricted to such a tiny range can possibly be considered random.

With a skewed distribution, chain lengths shoot right through the roof. Does this mean hashing is a lost cause?

Oh yeah?

Consider a deck of cards. Being finite, a deck holds only a finite amount of information. This information is the sequence of cards within the deck. There are 52 cards and therefore 52! different sequences that it can have, or around $8 \cdot 10^{67}$.

Shuffling a deck adds "noise" to it, or randomises it. A perfect cut-and-shuffle will create one of 2^{52} different equally likely sequences; this is much less information than the deck can hold, since $8 \cdot 10^{67}$ is around 2^{226}. So, after a single cut-and-shuffle, much of the original input sequence remains. A perfect shuffle injects (at most) 52 "bits" worth of "noise" into the deck. But after about 5 perfect shuffles more noise has been added to the deck than the deck can hold, and none of the original sequence can be said to remain.

What's that got to do with hashing?

In a hash function, the output value has a finite capacity to hold information, just like a deck of cards. So, one can "shuffle" the hash input sufficiently to saturate the information capacity of the output.

How to do this is another question.

The magic of exclusive-or

The "exclusive-or" (xor) is the difference between two bits. We talk of bits rather than numbers or characters, since any number or character (or any other data, for that matter) can be expressed as a sequence of bits.

Xor has the property that if you take a random bit stream and a non-random bit stream, their result when xor'd is a random bit stream! You can observe a similar phenomenon when a radio program is superimposed with white noise of equal intensity: the result is white noise.

Xor can be used to make an unbreakable method of encryption. Given a message to encrypt, we choose a random key of the same length and combine them using xor. The result will be left with no vestige of the original. Amazingly, the original message can be retrieved from the encryption by xor'ing it with the same key.

Requirements for a general hash function

Recall that we are trying to write a function whose output appears to be a uniformly distributed random variable, when given an arbitrary, non-random input distribution (such as a set of people's names).

To put that another way, the hash function must ensure that: (1) every bit of the input key affects every bit of the output hash value; and (2) each bit of the output hash is equally likely.

Let's look at how we can write such a function.

Some technical considerations

We have talked only about hashing strings, but there is nothing to prevent the hash function being used on arbitrary data. All data is just bits and bytes underneath, and all the hash function cares about is being given a sequence of bytes to index its table of random numbers.

If you change your notion of equality on keys, you will have to adjust the hash function to match. That is to say, you must ensure that any two keys that compare equal will hash to the same value. So, if you decide to use a case-insensitive string comparison, you had better modify your hash function to be case-insensitive as well.

Worst case hashing

Much is made of an argument that, although hashing is "usually" $\Theta(1)$, in the worst case it can be $\Theta(n)$. The worst case arises when all keys hash to the same value, in which case the hash table contains $n-1$ empty bins and one bin containing a chain of length n. A successful lookup reduces to linear search in a linked list, which takes $\Theta(n)$ time.

To construct the worse case, you will need to find n keys that hash to the same value. For a given table size, m, you could find such a set of keys by hashing at most nm strings; i.e., keep adding strings to the hash table the length of the longest chain exceeds n.

Can the worst case arise in practice?

If a good hash function is applied to n keys, it will uniformly hash them into m bins. The probability that all keys fall into any particular bin is

$$\frac{1}{m^n}$$

For $m = n = 1000$ (both small values), this is an *extremely* small number. You should stay awake worrying about a meteor crashing through the roof before you lose any sleep over a good hash function hitting its worse case.

The fact that n keys exist that hash to the same value does not mean there is any likelihood of encountering them in a dark alley at night.