Hashing - The Greatest Idea In Programming

Written by Mike James

Article Index

Although it is a matter of opinion, you can't help but admire the idea of the hash function. It not only solves one of the basic problems of computing - finding something that you have stored somewhere - but it helps with detecting file tampering, password security and more.

One of the basic problems of computing is finding something that you have stored somewhere.

For example, if you are keeping a table of names and telephone numbers how can you organise the list so that you can find the phone number given the name?

The most obvious solution is to keep the list sorted into alphabetical order and use some sort of search, usually binary, to locate the name. This is efficient but only as long as the list is kept in strict order.

If you only have to add names and number infrequently then you could use an overflow table that isn't sorted into order. Then to find a name you would use a binary search on the sorted table and if you didn't find the name there a linear search on the overflow table would either find the name or confirm that the name wasn't known.

Using an overflow table in this way works as long as the overflow table is keep short. If you are adding a lot of names then the time to perform the linear search in the overflow table will quickly become too much. In this case it is time to think of hashing.

Hashing is one of the great ideas of computing and every programmer should know something about it. The basic idea is remarkably simple, in fact it is so simple you can spend time worrying about where the magic comes from.

Suppose you have a function,

```
hash(name)
```

that will compute a number in the range 1 to N depending in some way on the value of *name* then why not store the name, and phone number at hash(*name*) in the table.

Notice that the hash function is a bit strange. It has to be completely deterministic in the sense that you give it a name and it always gives you the same number. However it also has to provide you with a different number for each name and as we will see this isn't always possible.

Using this scheme finding a name in the table is just a matter of computing hash(*name*) and looking at this location in the table to see if *name* is stored there!

Yes, that's all there is to it. Well there are a few practical details but using hash(*name*) to give the location where *name* is stored is the key idea.

An Example

To make sure that there can be no confusion about how simple hashing really is let's look at a small example.

If the hashing function is the sum of the character codes of the first two letters of the name minus 128 i.e.

```
hash(ABCDE)=ASC(A)+ASC(B)-128
```

where ASC returns the character code, then

```
hash("JAMES")
```

would be stored at

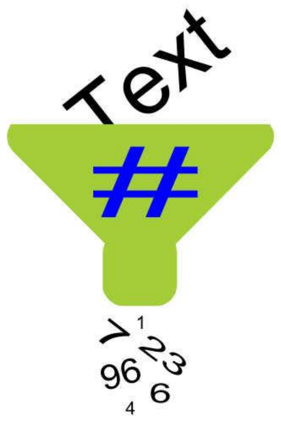
```
ASC("J")+ASC("A")-128
```

or, in plain English, at location 11 in the table.

If you want to know if you have a phone number for JAMES you simply compute hash("JAMES") and look in location 11 to see if it's there.

No sorting and no searching required. When you compute the hash function you know where to store the data and you know where to find the data.

The key idea is that a hash function takes in text or any sort of data and outputs a set of numbers based on that data.



This, or something similar, is the way most computer languages implement advanced data structures such as dictionaries are implemented using hashing. For example in Python a dictionary object is implemented using a hash table. So even if you don't explicitly build a hash table you could well be using one.

Collisions

Now that we have a concrete example to look at you might begin to see what the practical problems referred to earlier are. In particular where does our hashing function suggest that we store the phone number for JAMESON?

Yes you're absolutely right in exactly the same place that we store the phone number for JAMES. Clearly how you design the hashing function is all important. Indeed it isn't difficult to show that no matter how much effort you put into a hashing function collisions such as the one typified by JAMES and JAMESON will happen.

A good hashing function will tend to scatter the data values over the storage table as much as possible - hence another name for the technique is scatter storage - but collisions will still occur.

There are two main approaches to dealing with collisions - chaining and open addressing.

Chaining is simply an adaptation of the overflow method described earlier. If you compute the hash function and find the location already occupied then the second item to get there is stored in an overflow table and a pointer to it is stored along with the item in the main table. If more collisions occur the extra data items are stored in the overflow table and pointers are set to form a chain of items that all have the same hash value.

Now when you look for, or store, an item you may have to perform a linear search of a chained list but as long there aren't too many collisions this shouldn't take much time. The real problem with chaining is the extra storage needed to implement the pointers.

This brings us to the alternative - open addressing. The simplest form of open addressing is the interestingly named `linear probe'. If you go to store an item in the table and find that the location indicated by the hash function is already occupied then search down the table for the first free location and store the item there. This results in data items with the same hash value being stored close to each other in the table.

When you are looking for a name using this scheme you now compute the hash function and inspect sequential locations in the table until you either find what you are looking for or the first blank entry. The table is regarded as circular, i.e. the next location after the last is the first.

Obviously if the table is getting near full then hashing with a linear probe gets to look very like a simple linear search of an unsorted table! In practice linear probing works very well as long as the table is less than 75% full.

An alternative to linear probing is double hashing. In linear probing the locations searched are given by hash(*name*)+*n* where *n* is 0,1,2,3.. The problem with linear probing is that the initial hashing function `scatters' the data into the table but after that the linear stepping through the table groups the data together. It more like `blobbing' the data into the table rather than scattering it!

A better method would be to use two hash functions - one for the initial location in the table and the second to scatter the collisions a little. That is, use hash(*name*)+*n* hash2(*name*) where *n* is 0,1,2,3.. and hash2 is a second hashing function. With a careful choice of the second hash function you can make savings in the time taken to find a data item.

At this point I could start into a description of what makes a good second hash function, but I have to admit that unless memory is in very short supply it is usually simpler to make the table bigger using a linear probe than use double hashing. The point is that the efficiency of hashing is most affected by how full the table is and as long as the table is only around 50% used then there isn't much to be gained by using the more complicated double hashing.

What Makes A Good Hash Function

Most good hashing functions work by computing the remainder after dividing by the table size N.

This always gives a value between 0 and N-1 so it suitable but if N is a prime number then it is also excellent at scattering the data round the table. Of course, if you have a text value that you want to hash you first have to convert it into a suitable numeric value and a simple scheme like the one in the example will not do.

You need to produce a different numeric value for every possible text value and adding together the ASCII codes of the first two letters clearly doesn't work. A better method is to weight each of the ASCII codes by the position of the letter by multiplying by 1 for the first character, 10 for the second, 100 for the third and so on.. before adding them up to give a single value.

In general building a really good hash function is difficult and in most cases you need to find one that has good properties and has been well tested.

This brings us to the question of what makes a good hash function?

In general the set of things that you want to apply a hash function to is big, very big, but the set of results that you want to get back from the hash function is much smaller. That is, in hash(x)=y the range of x is usually large and the range of y is much smaller. For example, you might have x in the set of all possible 10-character words and y in the range 0 to 1023. The idea is that not all of the very large number of 10-character words will actually occur - usually less than the 1024 storage locations you might have set aside. You want the hash function to map the input words onto the 1024 possible values as evenly as possible - this minimizes the probability of a collision.

So a good hash function maps its input to its output in a way that is as unpredictable as possible i.e. in simple terms it is a repeatable random looking function.

Digest Hashing

The original purpose of computing a hash was to find a way to store and retrieve data, but hashes have many other uses. A hash serves as a label as well as an address for an item of data.

For example, suppose I send you a file of important data and I also tell you that

```
hash(data)
```

is 1234 where 1234 is standing in for a very large number in practice. Now suppose you received the file and stored it as data'. Then if you compute the same hash function:

```
hash(data')
```

and discover that it is 2342 you know that the data has been changed after it was sent.

So a hash can be used to detect changes in the data. Now consider for a moment what you can conclude if the hash of the data had worked out to 1234?

You can only conclude that there is a high probability that nothing has changed in the data. The reason is that the hash value acts as a digest of the data and more than one set of data will give you the same digest - i.e. there will be collisions. So getting the same digest simply means that the data is OK apart from any changes that might give you the same digest value. The key issue here is what is the probability that changing the data is going to result in the same digest value and the answer is usually that it is very small.

In other words, you are very likely to detect any change to the data.

This is the reason you often see an MD5 hash value quoted for downloads. MD5 is a cryptographic, i.e. very high quality, hash function which is used to compute a digest which you can use to check for errors.

Interestingly MD5 is not a good cryptographic hash function. There are ways of finding collisions fairly quickly and this could be used to make changes that leave the digest unchanged.

With a little modification, the idea of a digest hash can be made secure so that both the data and the digest can be securely sent to someone, who can in turn use the digest to verify the data and who the data was sent by. This is the basis of a digital signature.

The Password Mechanism

The fact that a hash can be used to label data is also the basis of most password systems in use. The basic idea is that the user invents a password and the system stores a hash of the password. When the user logs on the system applies the hash function to the password that the user has supplied and if it matches the stored hash then the user is allowed to proceed.

Notice that we have the usual problem with collisions - if the user gives the wrong password that just happens to give the same hash then they get in anyway.

The big advantage of this scheme is that even if an attacker gets the hash value they cannot use it to log in. They need the password that generates the hash and this is difficult to find. The reason is that while it is easy to generate the hash from a password, it is hard to generate a password from the hash. In other words, hash(password) is easy to compute, but hash^{-1}(value) is hard.

Just because it is hard doesn't mean it is impossible and so called password crackers use big computers and GPUs to compute the inverse.

Proof Of Work

One of the more unusual uses of the hash idea is in Bitcoin's proof of work algorithm. In this case a block of data is published and before a server can claim to have validated the transaction it has to find an item of data that when added to the block gives a hash with a given number of zeros. As computing the inverse hash function isn't easy the only way to do this is to add trial data and compute the hash until you find the result with the required number of zeros. This takes time and means that there is a high probability that just one server will find the answer first - and so avoid the problem of multiple servers validating the block.

A nice touch it that the algorithm anticipated the fact that hardware would get better. It includes feedback by measuring the average time taken to solve the hash problem and adds additional zeros to the requirement. You can prove that the time taken to find some data to add to the block to give n zeros in the hash goes up exponentially with n. In other words the difficulty of the problem can be varied to keep the time to solve about the same.

Once the problem is solved and the block declared valid the data is appended to the block so that the data it contains cannot be changed from that point on without invalidating the hash making the transaction tamper proof.

Hash Functions In Algorithms

There are also lots of uses for hash functions to speed up algorithms. For example, if you need to weed out duplicate records in a set of size N you might end up comparing every possible pair, i.e. roughly $N^2/2$ comparisons, or better sorting the table. An alternative is to compute hash values for all the records and store pointers to each record in a hash table. Duplicate records are then just collisions in the table.

You can also use hash functions with additional properties to implement matching algorithms. If you can invent a hash function that maps data that you consider to be similar to hash values that are close, you can use hashing to search not just for duplicates, i.e. identical values, but values that are close. This approach is important in string matching, acoustic matching and even pattern recognition.

There are lots of very clever algorithms that make use of the quick and easy way you can check for inequality between data. The basic principle is always that if two objects have different hash values then they can't be the same thing.

http://www.i-programmer.info/babbages-bag/479-hashing.html?start=1