Hashing. The whats, hows, and whys.

Author: doug@neverfear.org

Introduction

Hashing normally makes people think of cryptographic hashing often used to store passwords. Understandable considering this is what users might more familiar with. However, the principles of hashing are used for more than simply preventing retrieval of certain data.

What I'm referring to is hashing for the purposes of efficient data referencing. This is better known to developers rather than users.

Contents

Big-O

Let me explain a bit about how efficiency is measured. Algorithm analysts use what we call Big-O notation. In simple terms it represents the worst case runtime taken to process data of size N and the trends that the algorithm exhibits as N gets larger. It is notated by O(x) where x is the 'order of' the algorithm in terms of the above definition. The order of an algorithm is the type of growth rate over time required to complete execution as the volume of input data increases.

For example, any algorithm which takes the same time regardless of how much data it is required to process we notate this as O(1). Code-wise, an O(1) statement is generally a very simple one, such as:

```
int i = 5;
```

For an algorithm which takes the same time to process per element of data, that is to say that if we say T (time) is directly proportional to N, where N is the size of the input data, we can say it has O(N) efficiency. A code example demonstrating this order is often:

```
for(i = 0; i < N; i++) {
    puts(".");
}
```

As you can see, it executes the puts() function N times and so has a run-time efficiency of O(N).

Let's take one last example.

```
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        puts(".");
    }
}
```

In this example we see that there are two nested loops, and it will call puts() N^2 times, and so is considered $O(N^2)$.

I'm not going to go into any more detail about this, so long as you understand basically what big-O notation is and that the slower the graph of the growth rate (the order) is, the more 'efficient' the code is considered.

Primitive data structures

Now let's consider how we can handle a variable N element list of data, and things we might want to do to that data. There are two commonly useful models of organising. There are

- Arrays (sequential lists)
- Linked-lists (non sequential lists)

Arrays are allocated in one lump in memory and are accessed by the program by memory offsets. Consider this block of memory.

Index	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A
Value	110	117	108	108	100	105	103	105	116	97	108

In C, if you wanted to get the value at offset 5 then set it with another, we simply do the following:

```
int value = offset[5]; // retrieve the value
offset[5] = 111;       // set a new value
```

Simple enough right? This is obviously of the order, O(1) for both read and write operations.

How about if we didn't really know the offset a value we're after is at? Consider an array which contains multiple pieces of data, such as an object or a struct (C structure) might. Let's say we have this structure:

```
struct country_code {
    char * code;
    char * name;
};
```

Say this is our new array:

Memory location	0x00		0x01		0x02		0x03	
Struct fields	key	value	key	value	key	value	key	value
Field values	au	Australia	de	Germany	uk	United Kingdom	fr	France

This can be initialised as follows:

```
struct country_code codes[] = { {"au", "Australia"},
                                {"de", "Germany"},
                                {"uk", "United Kingdom"},
                                {"fr", "France"} };
```

Now imagine you wanted to figure out what country 'de' stood for. This is how you'd need to accomplish it:

```
#define N 4
int i;
for (i = 0; i < N; i++) {
    if (strcmp(codes[i].code, "de") == 0) {
        printf("The country is '%s'\n", codes[i].name);
    }
}
```

You can see here that this algorithm is of O(N). N here is the length of our array which in this case we already knew to be 4.

So to read or to write to any element in our new array based on the country code to locate the structure we have lost efficiency from our original approach of directly accessing memory locations.

Hashing

Can we improve on this at all? Yes! You can improve on this by quite a bit. Finally we bring in the subject of discussion, hashing. Hashing of our key elements we can easily calculate the offsets our elements with specific keys are to be.

A hash function must be chosen to produce positive integer hash codes from a given key. For the purposes of this article our key will always be a string (character array) but it is possible to develop hashing functions for any data type. The algorithm behind the hashing is crucial to a successful hashing implementation but I'll talk more about good qualities in hash functions later. Let's proceed with the notion of a hash function being a blackbox where you push in a key and get out an integer.

Now consider an array with 10 elements. We decide we're going to use hashing to handle organisation of this array. The question now is how do we use our hash code, to translate into an array index?

The answer is to use modulo to break the hash code down into an index within 10 elements. For instance suppose our hash code is 7362138 for a given key then the index within our array is 8. In C this can be expressed as follows:

```
#define N 10
int hash_code = 7362138;
int index     = hash_code % N;
```

The hash code production function is allowed to return negatives if the hash algorithm is designed in such a way to permit that, we should always take the positive form. In C we can use the abs() function to ensure our hash code is positive.

```
int index = abs(hash_code) % N;
```

For the remainder of this article the array we're managing with hashing will be referred to as a hash table and an array element will be referred to as a "bucket".

Collisions

An observant reader will notice straight away, that this is going to be highly prone to collisions for different keys. What if hash code was 326438 or -318? When either of these are integer-divided by 10, returning the remainder they will produce 8.

Hashing algorithms, like password hashing, are often non-injective. Due to constraints of a finite memory and a limited hash table width (the size of the table), it is very likely that you will end up with a "collision" where two different inputs produce the same output. So computer scientists and mathematicians decide not to care too much about creating a true injective hash function.

Collisions become more frequent as the hash table size is reduced and if we blindly ignore the possibility of a collision we can accident overwrite an element of the hash table array. To resolve collisions we take a few steps to make them harder to produce and if we get a collision then we resolve it in an efficient and safe way.

First off we attempt to make our hash() function produce results more uniformly distributed over the output range. This means that for every possible input key the probability of a particular output hash are equal to every other possible input key. So if we knew our keys were only ever going to be n characters in length (such as ID codes for employees, or car registration numbers), then we try to distribute the hash codes of the keys evenly over the output range. Knowing the that the keys will be fixed length makes building a hash function that much easier. Unfortunately most useful hashing needs to make use of variable length keys.

We should also take a prime number as the size of our array, hopefully reducing the number of common factors the hash code has for n although there is some debate over whether we should keep our table width as a prime number or whether we should use table widths that are powers of 2 and use bit logic to calculate keys rather than modulo. This argument comes about because of the speed of modulo can be rather slow when given lots of hash keys to calculate. In a previous example I showed how to calculate a key using the modulus operator this should be used if the width of a hash table is a prime. For completeness if you choose to implement hash tables whose widths are a power of 2 then you can do the following:

```
int hash_code = 7362138;
int width     = 8;
int mask      = width - 1;
int index     = hash_code & mask; // index will become 2
```

This is considerably faster than the modulus method but will result in greater possible frequency of collisions.

Given that our hash function produces codes of equal probability and if we choose to implement our hash table widths as prime numbers then we have reduced the likelihood of collisions, but not prevented them. In fact, its impractical to prevent collisions, instead we design a way to handle them. This takes two forms. **Chained hashing** and **Open Addressing**.

Chained hashing

With chained hashing each bucket of our hash table (the array where our actual data is stored in) we consider to be a linked list. When inserting new data into the hash table we simply look at the resulting index. If it is null (I.e. there is no list here) we create a list item and insert it at that index. If it is not null we iterate through the linked list until we find the end, then link our data value on the end.

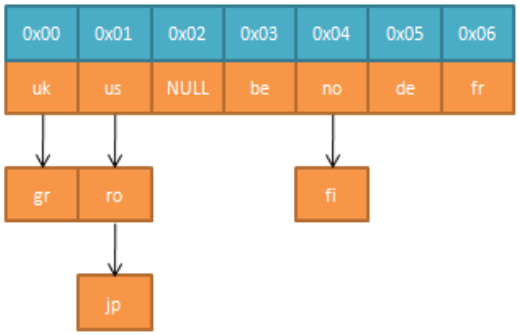
Retrieval from a hash table using this collision resolution technique requires that the original hash keys themselves are stored along with the data values. A retrieval operation will first find the elements index using the hash code, then iterate through the linked list and evaluates whether the stored key within each list item is equal to our search key.

Let M be the length of our hash table and N be the number of elements within the table. If our hash function is uniformly distributed then the average length of our linked lists will be N/M. This is called the load factor. So our search efficiency will become O(N/M). Since we will never know for sure what keys are being used it is improbable but possible for all of the elements to end up in the same bucket attached on the same list. If this occurs then our search efficiency is back to where we started at O(N). This is precisely why the choice of hash function is crucial.

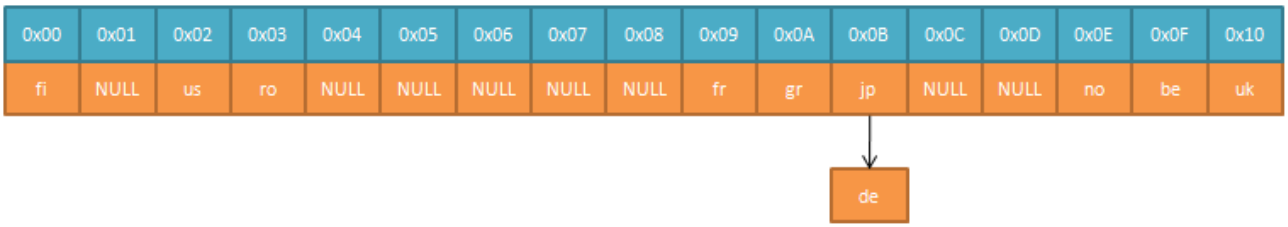
Since we are trying to improve things we introduced the concept of load factors. The load factor of a chained hash table is the theoretical average length of each list. If our hash function produces uniformly distributed hash codes then our load factor should always be N/M. We decide on a reasonable threshold load factor, say 1, and when our load factor is reached we create a new table, approximately twice the size and rehash all the elements. If you are using a prime number as the table width then you will need to generate a new prime number, this adds a small overhead on the resize and rehashing operation. If you are using a table width that is a power of 2 then this will simply be a case of doubling the existing size with no extra overhead. Resizing and rehashing will reduce the current load factor to 0.5 because M has doubled. The factor by which the table size increases by is implementation specific, but should result in the length of the table being at least approximately double.

The load factor of this will always range between our minimal load factor (in the above implementation description this would be approximately 0.5) to our threshold load factor (in the above example this was 1.0). As a rule low load factors result in too much memory not being used (but items being retrieved faster) and high load factors indicate the risk of long bucket chains which reduces the efficiency of collision resolution. A lot of sources recommend an optimal load factor is approximately 0.7.

To demonstrate why this is important let's look at the resulting hash table using a table width of 7 with 10 items. The load factor for this is ~1.429.



As you can see the hash table is overloaded, with all but one index's containing values and 3 of those indexes containing one or more collisions. Now if we set M to be approximately 2N to the nearest prime, let's say 17 our hash table will look more like this.



There are now 14 empty indexes and only one collision. The load is now ~0.588.

Open Addressing

The second way to resolve solutions is called Open Addressing. Instead of appending elements to a list at the index calculated from the hash functions result we simply insert our elements elsewhere if a particular index is already occupied. The way in which we locate where to insert the element depends on some collision resolution algorithm. This algorithm is used to retrieve items that are not found at their calculated index as well. With chained hashing we also monitor the load factor and at a particular threshold we once again create a new table and rehash all the elements.

Linear probing

Linear probing is the simplest open addressing locator algorithm. To insert an item we take our resulting "natural" index (often called the home address) produced by the hashing function scaled down by the table width and if its occupied by an element with a different key we move linearly over the table picking the next bucket to investigate by some fixed offset. For ease of explanation, for now let's just assume we move up by one element at a time starting from the home address. When we reach the end of the array we wrap around and begin at the start. We do this until we find an unoccupied index, then we insert the data element here.

If we never find an unoccupied index then the hash table is full and no further items can be added. In a good implementation of open addressed hashing this should never happen because the table should be resized and rehashed after surpassing a load factor threshold.

The same process is used later to find the original item by beginning at the home address and iterating over the array, checking the keys of every element as it goes. If it reaches an unoccupied index or has visited every element of the table then the item doesn't exist.

To remove an item we overwrite the item in a bucket with a unique identifier to label the element as removed. We do this instead of deleting the item completely because if we did then the linear probing would find the empty index and assume it had found the first available unoccupied index already and that the element it is searching for does not exist when actually there could be elements with the same hash code further down the list after the removed item.

A length of continuous items is called a cluster. When several different keys hash to a near but not identical home address they contribute to the length of the cluster which makes the search time for any element whose home address exists within the cluster longer. To demonstrate clusters let's look at a diagram that contains a combined-cluster, imagine the following hash table. Each item contains its key and for demonstration purposes its home address too. The last bucket there is unoccupied.

Bucket	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
Home	0x00	0x00	0x01	0x00	0x01	0x05	0x03	NULL
Key	One	Two	Three	Four	Five	Six	Seven	NULL

We don't really care about the key. What I wish to show you here is that the home address of an element in a bucket can be mixed. Here we see 4 different home addresses merged into one cluster of items. So if we were looking to find the item with the key "Seven" our home address is calculated to be 0x03 but 0x03 is occupied by an item with the key "Four". So we work rightwards

checking each key until we find “Seven”. As more and more items are added to the table and the load factor increases clustering becomes a serious performance issue.

There is also an example in that diagram of cluster collisions. The element with the key “Six” could have been the first item inserted into the table and the others afterwards. The clusters then over time have merged together (called colliding). If a very long cluster at 0x05 had developed before our cluster starting at 0x00 had became long enough to merge with it then when the two clusters do eventually merge the number of operations required to insert an item with the home address of 0x00 would become the length of both clusters combined together.

Avoiding clustering tends to be handled from two angles. The first is to alter the hashing function to produce hashes at intervals further away from one another. This gives a little room for clusters to expand before becoming a problem but as a result in order to implement this sections of memory are considered redundant buffers between clusters. This leads to greater memory consumption.

Non-Linear probing

The second angle this can be tackled from is called non-linear probing. I’m not going to go into detail on this but the jist is that instead of checking each index consecutively you probe in a larger non-linear offset. For instance you could probe in intervals of 1^2, 2^2, 3^2, etc or using triangular intervals such as 1, 1+2, 1+2+3, etc. These approaches are called quadratic and triangular probing respectively and apply to the insert, search and removal operations in place of a fixed increment. These approaches only work if the hash table width is a prime and the load factor is below 0.5. Otherwise it becomes possible that an unoccupied location will never be found.

This avoids the clustering effect described above but now suffers from a different form of clustering whereby items with the same home address still probe the same sequence of bucket locations. This second form of clustering can be avoided by using a technique called double hashing.

Double hashing uses the hash code in addition to calculating the home address to calculate an offset from the home address. An example of how this might be calculated in C:

```
int hash_code     = 7362138;
int width         = 16;
int offset_factor = 1 + abs(hash_code) % (width - 2);
```

This works because even if two hash codes result in the same home address, it is still unlikely that the offset_factor is the same for two items with the same hash code. The probing sequence then becomes something like 1 * offset_factor, 2 * offset_factor, 3 * offset_factor, etc.

Efficiency of hashing

Some very bright people went ahead and calculated the efficiency of each of these three distinct collision resolution algorithms as load factor increases that I've decided to include here.

Load Factor	0.10	0.25	0.50	0.75	0.90	2.00
INSERT						
Chained hashing	0.10	0.25	0.50	0.75	0.90	2.00
Linear Probing	1.12	1.39	2.50	8.50	50.50	N/A
Double Hashing	1.11	1.33	2.00	4.00	10.00	N/A
FIND/REMOVE						
Chained hashing	1.05	1.12	1.25	1.37	1.45	2.00
Linear Probing	1.06	1.17	1.50	2.50	5.50	N/A
Double Hashing	1.05	1.15	1.39	1.85	2.56	N/A

You'll notice that both probing approaches cannot cope when the load factor is over 1.0. This is obvious when you think about what the load factor means. Especially it means how many items exist in each bucket, since the buckets are finite it is impossible to have a higher load factor than 1.0!

Conclusions

As shown in the table in the previous section I can safely give the following conclusions. Use chained hashing, select an appropriate hash function for your keys and make a suitable choice regarding the width of your table and you shall be the king of efficient hashing.

Remember that probing methodologies require a lower load factor than chained hashing for efficient access and this means a wider table with more redundancy than chained hashing which not only is faster, and not only can carry an infinite load factor (although obviously the efficiency nose dives as the load factor increases) but is operates are higher load factors much better than probing methods.

Finally be aware that although it's not obvious whether a prime table width or a power of 2 width is superior as with most things in engineering identify the advantages and disadvantages of each approach and select the approach that is best for the likely data that will be managed by your hashing implementation.