ALS HASHMAP

By Jose Aaron Lopez Garcia – joseaaronlopez@gmail.com

Index

Introduction

HashMaps are one of the most important areas of study within data structures and algorithms.
A HashMap is a collection of key-value pairs where we can insert, update, lookup or delete a value given its key with amortized O(1) complexity and general constant runtime.
The practical uses for such a structure are limitless; refraction in object oriented code, dynamic data types, dictionaries, index cache in databases and much more. This paper however does not entail into much detail into the use cases for a HashMap, they are already well known.

In the following document I will present, with as much detail as possible, the inner workings of all HashMap implementations most known and used and introduce a new type of HashMap with a different collision resolution approach.

First things first: What's a HashMap?
In general terms a HashMap consists of a table (an array or buffer) of size T (it can hold up to T elements). We want to insert the value V at a position withing this array by using the key K.
It is obvious we cannot use the key to index so we need to convert it to an actual index within this table.
This is achieved by generating a hash of the key using a hash function H(K) that generates a well distributed redundant hash of the key. Since the hash can be any number and does not work as an index, we do the modulo operation of the hash with the table size to obtain a valid index.

$$I = H(K)\%T$$

However we have that the hash value is theoretically infinite while our table size is limited, meaning that more than one key will end up hashing to the same slot within the table, this is called a collision and must be handled to allow for both keys to be stored within our table.

This is what this paper will dig deep into and attempt to provide a solution that is efficient in both time and memory consumption.

Chaining

The first collision resolution that was designed is chaining. The idea is that all slots in your table contain a linked list. As a key gets mapped to a slot you simply insert (or update) the key-value pair in the list. Since the keys will always fall into their hashed slot the technique is called Closed Addressing.
The following C code shows a simple design of such a structure.

```
struct MapNode{
    Pair pair; // the <key - value> pair
    struct MapNode* next; // simple linked list for collision
};

struct HashMap{
    MapNode* table; // the buffer
    size_t T; // the size of buffer
};
```

This implementation is certainly simple, insertion and deletion inside linked lists is very well known and easy to implement.
If run time is crucial you can use a self-balancing tree such as the Red Black Tree for O(logn) operations at the expense of increasing the overhead of pointers needed by it.

This implementation is however inefficient, allocating memory dynamically has an extra cost and the items within linked list suffer from cache misses, unlike items that are in sequential order.
You also end up with a lot of empty space if the hash function does not have a uniform distribution.

This implementation becomes extremely slow as the load factor increases as it means the data structures are getting bigger and the operations on them are expensive as explained above.

A better approach that solves the problem with cache misses, dynamic memory overhead and unused table space is probing.

Probing

Probing is a technique that solves collision by searching for an empty spot within the table to store the colliding key.
It does not use any auxiliary data structure as the place to store the pair is found within the table itself. This overhead and cache misses are alleviated as the table is by itself a sequential array of elements and the only extra memory overhead is a bit on each slot to know if it's used or free.
Since a given key does not necessarily end up in its hashed slot the technique is called Open Addressing.
A simple C code illustrating the structure of such collections can be the following:

```
struct MapNode{
    Pair pair; // the <key - value> pair
    unsigned char used; // mark if bucket is used
};

struct HashMap{
    MapNode* table; // the buffer
    size_t T; // number of elements
};
```

There are many algorithms used to find an empty slot on the table but they can all be grouped into the following:
- Linear
An empty spot is found by linearly iterating the table starting from the hashed slot.
-Quadratic
Each iteration is done by multiplying the index to a constant factor.
- Double Hashing
A second hash function is applied to either the key or the first hash. Some implementations use more than two hash functions.

To enhance future operations on the newly inserted key some implementations store a pointer on each slot that specifies where the next collided item has been reallocated to, this way instead of doing more probes for that item you can follow a linked list that traverses the table.

This implementation has a few disadvantages that chaining doesn't: it can't store more elements than the table's size and the load factor increases dramatically faster than with chaining.

Whichever implementation you choose the disadvantages are clear: the load factor cannot exceed 75% and you either loose space or you loose time. Resizing is also another inconvenience that has to be avoided at all costs as the operation is undeniably O(N).

In this document I present a design of my own for an algorithm that attempts to alleviate these problems that affect current HashMap implementations by being efficient in both memory and time usage.

But first lets take a look at some interesting techniques used to improve some of the problems with probing that introduce a nice new idea; we can reorganize elements to enhance the runtime of future probes.

Cuckoo Hashing

This technique is a variation of double hashing where there's a second hash function that maps each key to a different slot, there is a difference with regular double hashing techniques though; a newly inserted item may displace an already inserted one.

When a key is inserted and there is a collision, we insert the new key into the slot it has been hashed too, moving the contained key to another slot by using the second hash function. This in turn may displace another element contained in the slot the second hash function has calculated.

This proves specially beneficial as the new item will always have O(1) complexity, but the displaced item may actually require more iterations to achieve; O(logn). If we consider using simple probing (linear or otherwise), both items will then end up with O(logn) complexity, while cuckoo allows us to have at least one O(1) element.

The general idea is: when a new item collides, we reinsert the contained item and insert the new one: this will always be better as you will be guaranteed a much more preferable O(1) operation.

Hopscotch Hashing

A rather interesting technique that is often overlooked when probing hashmaps are concerned. Hopscotch hashing works on a very interesting idea that serves as one of the basis of the ALS HashMap; probing should at all times be limited to a certain amount of iterations to avoid doing more probes than necessary.

This is achieved by defining what is called the Hopscotch region; an area that limits the number of buckets that can be between a key's original hashed entry and its actual entry as found by the collision resolution approach (in this case linear probing).

The procedure is as follows: when inserting item V and a collision occurs, find an empty spot E to store the item using linear probing, if this slot is away from the original hashed slot I by more than H (the Hopscotch region) then a third already stored item S is to be found so that that its position is within the Hopscotch region of the to-be inserted item and the item contained within S can be reallocated to E without breaking its own Hopscotch constraint.
In a (much simplified) pseudocode the algorithm would resolve collisions this way:

```
if (index(E)-index(V) > H){
    S = ... // item such as: index(S)-index(E) < H
    pair(E) = pair(S)
    pair(S) = pair(V) // collision resolved
}
```

There is however a big problem with Hopscotch that allows for O(n) operations to occur on insertion, even if they are technically forbidden for lookup and deletion; the algorithm is only really useful if you find an empty spot that falls out of the Hopscotch region, meaning this empty spot could very well be far enough to require an O(n) operation to find it.

To fix this, for the ALS HashMap all probing operations are limited to the Hopscotch region only, if our probing algorithm has to search deeper than H then it has to stop and consider probing as failure, requiring another technique to be used: chaining.

Thus the ALS HashMap is a hybrid between probing and chaining that uses probing as its first methodology, using cuckoo hashing to allow for the displacement of elements benefiting future operations and the Hopscotch region to forbid any probe operation to go above O(logn) iterations to complete.

ALS HashMap

The ALSHashMap was designed as a way to improve how the HashTable operates by combining both probing and chaining using an algorithm that correctly decides when to use one and when to use the other to make sure the table's space is correctly used and that more elements can be stored than the table's size to avoid resizing.
It works by limiting the probe size and remembering elements that have been inserted by probing or are within an auxiliary collection.
Probing is limited to guarantee constant operations and is recommended to be equal to the logarithm in base 2 of the table's size or close. To improve usage of memory and cache, probing is done on both directions starting from the hashed slot and linearly. This limitation to probing is called the Probe Range, and the area of probing for a particular slot is called Probe Area, and are given by the following formula:

Probe Range R = log(T)
Hopscotch Region H = 2R

Each slot in the table requires space for:
- 2 bits to mark the type of the slot.
- enough space to allocate either a key-value pair or a collection of your choice.

There are 4 different types that can be assigned to a given bucket depending on its condition:
- E: the bucket is empty.
- A: the bucket contains an auxiliary collection.
- L: the bucket contains an item that belongs here → H(Ki)%T = i
- S: the bucket contains an item that was reallocated by probing → H(Ki)%T != i

The general structure of the map can be seen in the following C code:

```
enum {E, A, L, S}; // types of buckets

struct ALSBucket{
    unsigned char type; // the type of this bucket
    union{ // data is stored in either a pair or a collection
        Pair pair;
        Collection col;
    } data;
};

struct ALSHashMap{
    ALSBucket* table; // our table of ALS buckets
    size_t T; // the size of the table
    size_t R; // the maximum probing range: log(T)
};
```

Initializing such table is as simple as setting each node to empty and a correct probe region R.

To operate on this table we do the normal calculation to obtain an index:

$$I = H(K)\%T$$

However this leads us to a bucket that can be of different types and we must operate accordingly in order to maintain the rule of constant limited spacing.

Insertion

When an item is inserted the following operations must be done depending on type:

- E: empty bucket, insert key-value pair and mark bucket as type L.

- A: bucket contains auxiliary data structure, insert key-value pair into it.

- L: bucket contains pair that belongs here. If the key matches insertion key, update the value. If the key collides try using probing. If probing finds an empty slot within the probe area then allocate pair to the new slot and mark it as type S. If probing fails we must resort to chaining: we create an auxiliary collection in that bucket and insert into it all the pairs that belong here (including pairs reallocated by probing), marking the bucket as type A.

- S: this scenario is where cuckoo hashing takes place. Cuckoo hashing tells us that if an item has been hashed to a slot and it collides, we must solve the collision by moving the already contained item, not the new one, thus guaranteeing that at least one of the items will have O(1) complexity. In this case the bucket contains an item that has been allocated here by probing, so we know applying the cuckoo technique here will without a doubt allow us to have an O(1) item.
Since the new key has been hashed to this bucket, it has priority over the probed pair; we insert the new pair here (marking it as type L) and do a recursive insertion on the old pair.

The idea behind limited and marked probing is to allow pairs to “borrow” nearby empty buckets until the actual key that belongs there requests it, and in the the case it does then it will attempt to find another nearby empty slot or resort to using an auxiliary data structure.

Lookup

Finding a value given a key follows a similar behavior: we hash the key and find the slot it belongs to, then depending on its type we follow different approaches:

- E: the slot is empty, no key has ever hashed to this slot so we must generate the appropriate error.

- A: we search for the value in the contained collection.

- L: the pair allocated in this slot belongs here, if the key does not match then do probing, if not found then key not found.

- S: the key contained has been allocated by probing; key not found.

Deletion

Deletion is the same; calculate table index and act depending on type:

- E: key not found.

- A: search in collection.

- L: if contained key matches then probe for nearby pairs that belong here and move them, if none then mark as E. If the key contained does not match then do probe, if probe works mark that bucket as E, otherwise key not found.

- S: key not found.

Resize

Resizing is done by iterating over the table, if a slot is of type L or S then insert the pair contained, if it’s of type A you have to iterate over the collection and insert each contained pair.
This is an expensive operation with O(N) complexity and should be avoided at all costs.
This over ALS algorithm already tries to improve on time and space consumption and manages to be stable with extremely high load factors, but there are times when more table space will be better for future operations.

For the ALS HashMap there are three basic stats that are monitored with no extra overhead on the operations:
- The collision resolution counter: when a data is to be stored in an auxiliary collection we check if a future table resize might help avoid this collision. This is done by hashing the key with the next possible table size and checking if it resolves to a different index. We use this value to calculate the collision factor:

$$CF = CRC/N$$

Where N is the number of elements in the table. This factor must not exceed a certain amount.

- Maximum recorded collection: whenever a pair is inserted into an auxiliary collection we check if such collection we check its size and record the maximum ever reached. We use this value to calculate the iteration factor:

$$IF = MA/R$$

Which means the maximum known collection must not exceed the probe range multiplied by a constant factor.

- Number of auxiliary collections: counter incremented for each auxiliary collection created. We use it to calculate the auxiliary factor:

$$AF = NA/T$$

We use this to keep auxiliary collections to a minimum.

The idea is to keep these values at a minimum for better performance but also keep resizing low. For that we define a maximum allowed constant for each of the factors. These constants are called the table's growth policy.

Example Insertion

Given a table of size T=8, probe range R=4 and pairs of values to be inserted:
<449, 26>, <48, 2>, <487, 15>, <521, 45>, <52, 14>, <977, 30>, <865, 26>, <409, 25>, <926, 49>, <103, 38>, <847, 6>, <255, 22>, <738, 41>, <538, 32>, <505, 36>, <243, 5>, <414, 41>, <557, 2>, <906, 28>, <47, 11>.
With the following growth policy:
- CF: 0.5
- IF: 1.5
- AF: 0.5

And growing functions:
T' = 2T // table size duplicates
R' = R+1 // probe range increases by 1
The table is initially empty.
Show a step by step of the insertion of the elements (omit reinsertions caused by resize).

Step 1.
Table is initially empty
[, , , , , , ,]
CRC: 0
MA: 0
T: 8
R: 4
NE: 8
NA: 0 -> 0.000000
SA: 0
Size: 0
Load Factor: 0.000000

Step 2.
Setting <449, 26>
449%8 = 1
Slot 1 is empty so we insert the pair here and change to type L
[, L, , , , , ,]
CRC: 0
MA: 0
T: 8
R: 4
NE: 7
NA: 0 -> 0.000000
SA: 0
Size: 1
Load Factor: 0.125000

Step 3.
Setting <48, 2>
48%8 = 0
Slot 0 is empty so we insert there and change to type L.
[L, L, , , , , ,]
CRC: 0
MA: 0
T: 8
R: 4
NE: 6
NA: 0 -> 0.000000
SA: 0
Size: 2
Load Factor: 0.250000

Step 4.

Setting <487, 15>
487%8 = 7
Slot 7 is empty so we insert there and change to type L.
[L, L, , , , , , L]
CRC: 0
MA: 0
T: 8
R: 4
NE: 5
NA: 0 -> 0.000000
SA: 0
Size: 3
Load Factor: 0.375000

Step 5.
Setting <521, 45>
521%8 = 1
Slot 1 is occupied, we probe for nearby empty buckets. One is is found at slot 2, where we insert the pair and mark as type S.
[L, L, S, , , , , L]
CRC: 0
MA: 0
T: 8
R: 4
NE: 4
NA: 0 -> 0.000000
SA: 0
Size: 4
Load Factor: 0.500000

Step 6.
Setting <52, 14>
52%8 = 4
Slot 4 is empty so pair is inserted directly and marked as L.
[L, L, S, , L, , , L]
CRC: 0
MA: 0
T: 8
R: 4
NE: 3
NA: 0 -> 0.000000
SA: 0
Size: 5
Load Factor: 0.625000

Step 7.
Setting <977, 30>
977%8 = 1
Slot 1 is occupied so we probe for a nearby empty bucket. One is found at slot 3, which is marked as S.
[L, L, S, S, L, , , L]
CRC: 0
MA: 0
T: 8
R: 4
NE: 2
NA: 0 -> 0.000000
SA: 0
Size: 6
Load Factor: 0.750000

Step 8.
Setting <865, 26>
865%8 = 1
Slot 1 is occupied so a nearby empty bucket is probed. One is found at slot 5.
[L, L, S, S, L, S, , L]
CRC: 0
MA: 0
N: 8
R: 4
NE: 1
NA: 0 -> 0.000000
SA: 0
Size: 7
Load Factor: 0.875000

Step 9.
Setting <409, 25>
409%8 = 1
Slot 1 is occupied. We probe for nearby empty buckets but none is found (the only empty slot is out of probe range), so we create an auxiliary data structure in the conflicting slot and insert all colliding elements into it.
[L, 5, , , L, , , L]
CRC: 0
MA: 5
T: 8
R: 4
NE: 4
NA: 1 -> 0.125000
SA: 5

Size: 8
Load Factor: 0.500000

Step 10.
Setting <926, 49>
926%8 = 6
Slot 6 is empty, we insert pair into it.
[L, 5, , , L, , L, L]
CRC: 0
MA: 5
T: 8
R: 4
NE: 3
NA: 1 -> 0.125000
SA: 5
Size: 9
Load Factor: 0.625000

Step 11.
Setting <103, 38>
103%8 = 7
Slot 7 is occupied so we probe for a nearby empty bucket; one is found at slot 5.
[L, 5, , , L, S, L, L]
CRC: 0
MA: 5
T: 8
R: 4
NE: 2
NA: 1 -> 0.125000
SA: 5
Size: 10
Load Factor: 0.750000

Step 12.
Setting <847, 6>
847%8 = 7
Slot 7 is used so we probe for a nearby empty bucket, which is found at slot 3.
[L, 5, , S, L, S, L, L]
CRC: 0
MA: 5
T: 8
R: 4
NE: 1
NA: 1 -> 0.125000
SA: 5

Size: 11
Load Factor: 0.875000

Step 13.
Setting <255, 22>
255%8 = 7
Slot 7 is occupied. Probing does not find any empty bucket so we create an auxiliary collection in the bucket, inserting all colliding elements into it.
[L, 5, , , L, , L, 4]
CRC: 0
MA: 5
T: 8
R: 4
NE: 3
NA: 2 -> 0.250000
SA: 9
Size: 12
Load Factor: 0.625000

Step 14.
Setting <738, 41>
738%8 = 2
Slot 2 is empty so pair is inserted here.
[L, 5, L, , L, , L, 4]
CRC: 0
MA: 5
T: 8
R: 4
NE: 2
NA: 2 -> 0.250000
SA: 9
Size: 13
Load Factor: 0.750000

Step 15.
Setting <538, 32>
538%8 = 2
Slot 2 is used so we probe for a nearby bucket, which is found at slot 3.
[L, 5, L, S, L, , L, 4]
CRC: 0
MA: 5
T: 8
R: 4
NE: 1
NA: 2 -> 0.250000

SA: 9
Size: 14
Load Factor: 0.875000

Step 16.
Setting <505, 36>
505%8 = 1
Slot 1 contains an auxiliary data structure so we insert the element there.
However there is a restriction on the size of auxiliary data structures: they have to be bellow 1.5 times the probe region as imposed by the growth policy.
In this case the size of the collection in slot 1 is 6 and 1.5*R = 6, so the table needs a resize.
After a resize operation all elements are reinserted in the new table using the same algorithm. The table ends like this.
[L, L, L, S, L, S, S, L, S, L, L, S, , S, L, L]
CRC: 0
MA: 0
T: 16
R: 5
NE: 1
NA: 0 -> 0.000000
SA: 0
Size: 15
Load Factor: 0.937500

Step 17.
Setting <243, 5>
243%16 = 3.
Slot 3 contains a pair of type S. We insert the new pair into this position as it owns it and we reinsert the old pair, which ends up creating a new collection in slot 1 as it can’t find any other empty bucket.
[L, 3, L, L, L, , S, L, S, L, L, S, , S, L, L]
CRC: 0
MA: 3
T: 16
R: 5
NE: 2
NA: 1 -> 0.062500
SA: 3
Size: 16
Load Factor: 0.875000

Step 18.
Setting <414, 41>

414%16 = 14
Slot 14 is occupied by pair of type L, we probe for nearby bucket and find one at slot 12.
[L, 3, L, L, L, , S, L, S, L, L, S, S, S, L, L]
CRC: 0
MA: 3
T: 16
R: 5
NE: 1
NA: 1 -> 0.062500
SA: 3
Size: 17
Load Factor: 0.937500

Step 19.
Setting <557, 2>
557%18 = 13
Slot 13 is occupied by type S pair, we insert new pair here and reinsert the old pair, which can’t find any new empty buckets so it creates a collection at slot 15.
[L, 3, L, L, L, , S, L, S, L, L, S, S, L, L, 2]
CRC: 0
MA: 3
T: 16
R: 5
NE: 1
NA: 2 -> 0.125000
SA: 5
Size: 18
Load Factor: 0.937500

Step 20.
Setting <906, 28>
906%16 = 10
Slot 10 contains pair of type L, we probe for nearby buckets and one is found at slot 5.
[L, 3, L, L, L, S, S, L, S, L, L, S, S, L, L, 2]
CRC: 0
MA: 3
T: 16
R: 5
NE: 0
NA: 2 -> 0.125000
SA: 5
Size: 19
Load Factor: 1.000000

Step 21.
Setting <47, 11>
47%16 = 15
The slot contains data collection, we insert the pair into it.
[L, 3, L, L, L, S, S, L, S, L, L, S, S, L, L, 3]
CRC: 0
MA: 3
T: 16
R: 5
NE: 0
NA: 2 -> 0.125000
SA: 6
Size: 20
Load Factor: 1.000000

The final table ends up being the following:
[L, 3, L, L, L, S, S, L, S, L, L, S, S, L, L, 3]
With 20 inserted elements.
Containing:
- 2 auxiliary data structures of size 3 each. Operations on these items are logarithmic: log2(20) = 4.
- 5 reallocated pairs. Operations on these items are logarithmic, since the maximum distance from a reallocated pair to its original value is R = log2(16)+1
- 9 correctly allocated pairs. Operations on these items will be constant O(1) as their key hashes to the slot the are stored in.

Tests

I have conducted a test on my initial implementation of the ALSHashMap and gathered the following data.
The test consisted on generating 100,000, 300,000 and 500,000 random key-value pairs on a table with initial size of 8 buckets and conducted a series of stats on the resulting table. The code has been implemented to gather statistical data about the general structure of the table and print it on screen, the results are shown in the following table.

Stat	100,000 Elements	300,000 Elements	500,000 Elements
Collision Resolution Counter	2,833	1,572	267
Maximum Size of Auxiliary	9	9	9

Collection			
Table Size	65,536	262,144	524,288
Probe Range	17	19	20
Number of empty Buckets	770	5,947	43,524
Number of auxiliary collections	19,185	25,271	11,196
Number of L	32,182	153,328	310,944
Number of elements in the table	99,999	299,975	499,943
Load Factor	0.988251	0.977314	0.9169858

After inserting the elements, deletion and lookup operations conducted show no performance issue and appear to be fully operational.

Conclusion

In summary, the ALSHashMap works by attempting to use already available space whenever possible to avoid auxiliary collections, but the inevitable use of auxiliary structures allow for more items to be stored than the table's occupation to a certain degree avoiding expensive resize operations.
This allows for the HashMap to operate correctly regardless of the load factor.

The high load factor not only does not slow down or affect the performance of the table operations, it is actually a desired result that the algorithm attempts to achieve all the time. The more the table is used the better for both storage and performance, but keeping this possible without doing a lot of table resizes or suffering from scanning big tables using non-limited probing can only be achieved through the use of auxiliary data structures, always limiting its use to not affect the overall performance.

In theory if an area within a table becomes full and probing within that area becomes slower and slower, we have to avoid it by moving all conflicting pairs to an auxiliary collection, freeing up the space occupied and allow for more items to be inserted by probing in that area.

In this sense auxiliary data collections act as a congestion avoidance system, relieving the stress within areas of the table with too many collisions and allowing for future operations to be stable in performance and allow for more probing to occur.

With this in mind the ALS HashMap attempts to maintain performance by both imposing a limitations to both probing and chaining to guarantee worst case logarithmic time.

For the 500,000 elements test we got a probe range R=20 and a maximum size of collection MA=9. Both of these numbers, which are responsible for the limitations to probing and chaining, prove the O(logn) worst-case scenario:

$$\log(N) = \text{ceil}(\log(500{,}000)/\log(2)) = 19$$
$$R = \log(N)+1 = 20$$
$$MA <= \log(N) \rightarrow 9 <= 19$$

In all cases the load factor was above 0.9, meaning table space was extremely well used while still maintaining the above mentioned performance.
And as shown in the sample insertion exercise there are times when the load factor can reach 1 and the table can still manage to perform well and even act effectively on these cases to make sure we can continue to operate correctly without a table resize.