Hash Tables – A Simple Introduction

Richard Lynch
123022021
28th October 2016

Abstract

A simple double hash function is presented which takes a string and returns an index for the given string. This hash function calculates an index using the key and the table size, and if this index is filled will perform a second hash to calculate an increment away from the current index to probe. It will return the index if an empty slot is found or if the key is found.

The key is "hashed" by calculating the sum of each character put to the power of its position in the string. The modulo(mod) of this number is found using the size of the table as the remainder operator. The increment is the first character value mod the size of the table plus one.

Testing hardware: MacBook Pro(Late 2013), 4258U Intel i5, 8GB of RAM,
Hash table details: size = 22943, Filled = 21303, Load = ~90% load
Results: Max probes = 115(0.046 s),
Entire operation entries = 21303(97.989ms)

1 Introduction

For many applications it is critical that data be inserted, retrieved and deleted quickly from some large set of data. This can often be the case for time sensitive queries such as websites and databases which may have very large sets of data, but are required to query the database and return the data in near real time.

Retrieving data from a linked list or unsorted array would involve probing each node or index until the key is found. This may be a viable option if the number of items (n) is small, but will not scale well, with $\mathcal{O}$(n). Similarly, a pre-sorted array can be searched using a binary search which scales with $\mathcal{O}(\log_2 n)$. This is an improvement on a linear probing search, but is still dependant on n, so as n becomes large, the search time will increase.

Hash tables are one solution to this issue as a well-designed hash table can approach $\mathcal{O}(1)$. A hash table functions by using some unique value of the data(the key), to generate a unique address in which to store or retrieve the data. This will often be in the form of an index of an array. The function which performs this task is referred to as the "hashing function" and will map the key on to some data structure.

The major strength of the hash table is the fact that in most cases, a hashing function can be performed on the key and the data found in a single probe of the array. This will not always be the case, if the address probed contains another value, a collision has occurred. Several solutions exist to deal with collisions, broadly in two categories "Open Addressing" and "Closed Addressing" [1]

An "Open Addressing" function is one where the address of the data is not determined solely by the key. Upon collision, another address will be used.

1. The key is "hashed"; Simply, some operation is performed on the key to return a "home address", which is the first address which data could be stored.
2. The "home address" is probed; If this address is found to be filled by another value, it is known as a collision.
3. Iterate the address; If a collision occurs, change the address and probe some new address, according to the "probe sequence". Repeat this step until no collision occurs.
4. Store(or return) the data at the address; The current address is the correct one and will be returned by the hashing function.

A "Closed Addressing" function, commonly known as "Separate Chaining", treats each address as a "bucket" which contains some other data structure behind it. In this example a linked list is behind each bucket, and the bucket array contains only pointers.

1. The key is "hashed"; The "home address" is returned.
2. The "home address" is probed; The data is retrieved from/added to the linked list whose head pointer is stored in the home address.

These two methods have many varieties but serve to illustrate the strengths and weaknesses of hash tables.

2 "My Pet Hash Function"

2.1 Implementation

My implementation of the hash function used open addressing with double hashing to store names in a hash table. A class called HashTable was created which can construct a hash table according to given parameters. A single method(function) was used to store and return the index of a stored key. This method also prints the number of probes required to find the key. This class presumes the table size is a prime number, but this is not required.

2.2 Hash Function

The "hash_function" first puts each character of the key to it's the power of its position within the key. That is to say;

$$index \mathrel{+}= string[n]^n;$$

This yields a large number which is quite unique to the key. This index may be outside of the array, so the mod operation is performed with the size of the hash table as the remainder operator. This operation repeatedly divides the large index by the size of the array, and returns the remainder of the division, which is guaranteed to be from 0 to the size of the array. For optimum performance, the table size should be a prime number, as this will result in much better coverage. For testing prime numbers close to the desired array size were used.

2.3 Table Load and Coverage

This is one of the most important metrics to consider when designing a hash function, as different hash functions will perform better or worse at a given load. Table load was calculated by a method in this class by dividing the filled cells by the size of the array. Both of these values are members of this class. This hash function was tested at 20%, 50% and 90% load, with table sizes of 17, 5101 and 22943, which are all prime numbers.

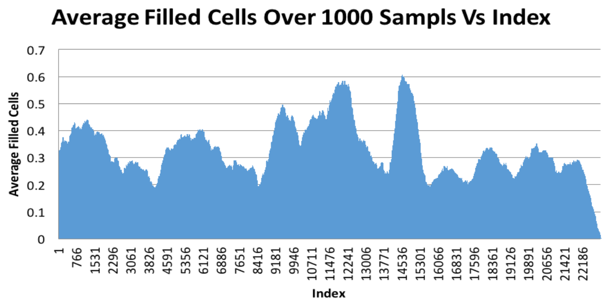

Figure 1: Table of size 22943 at 50% load

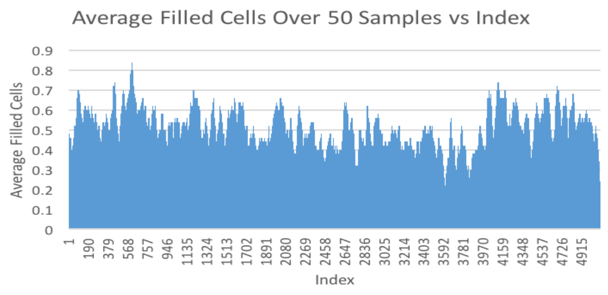

Figure 2:Table Size of 5101 with 50% load

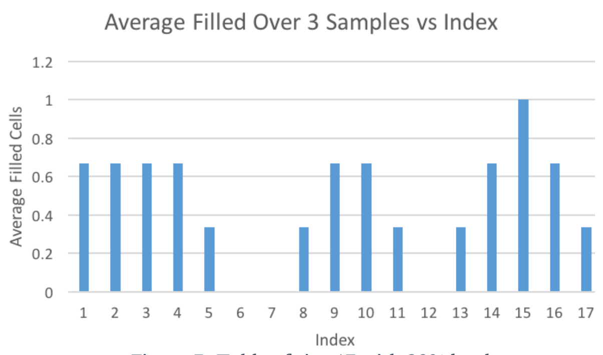

Figure 3: Table of size 17 with 90% load

In Figure 1, 2 and 3 the average number of filled cells per is charted on the y-axis and the index on the x-axis for the 3 test table sizes

2.4 Collisions

Collisions occur when an index being probed is occupied by another key. A small numbers of collisions is acceptable, but if too many occur the hash function will begin to approach a linear probing search, which has O(n).

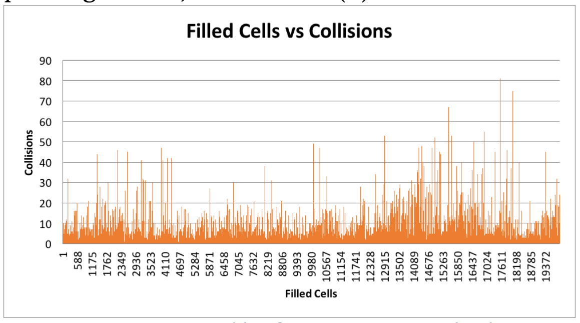

Figure 4: Table of size 22943 at 90% load

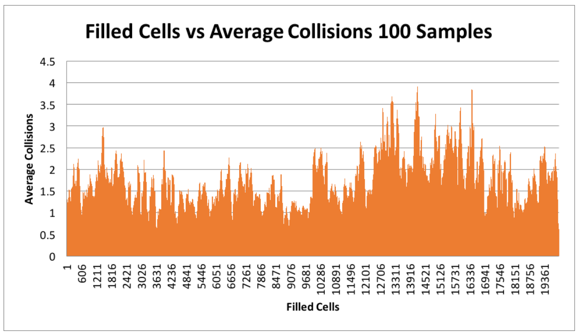

Figure 5: Table of size 22943 at 90% load

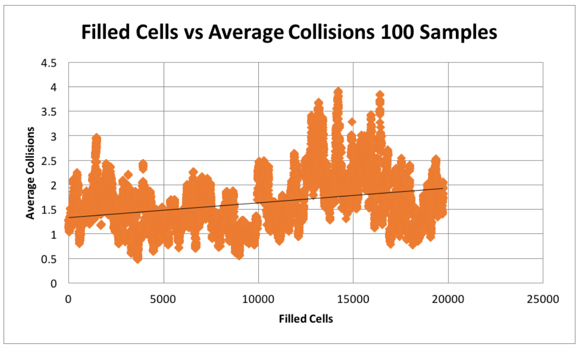

Figure 6: Table of size 22943 at 90% load

Figure 4, 5 and 6 show the number of collisions (y-axis) as the number of filled cells (x-axis) increases. In figure 6, the black trend line shows that the average number of collisions increases with the number of filled cells, $\text{slope} = 3\times10^{-5}$. This is very low, which demonstrates that the hash function is well designed and spreads the keys out to the entire map in few collisions.

3 Research and Review – A Cuckoo Solution

3.1 Introduction

The above implementation resolves collisions using double hashing. Another example of an open addressing system is Cuckoo hashing [2] [3] [4] [5], which deals with collisions in a very different way. When a collision occurs, the pair occupying the home address may be forced into another address. This document will discuss an example cuckoo algorithm sourced from GitHub. [4]

3.2 Implementation

This implementation uses a class "cuckoo" which stores and manages a hash table using cuckoo hashing. The implementation will be discussed in terms of the three activities; construct, lookup and insert, as lookup is very similar to delete.

Construct

When the class is constructed, it initialises several values;
"p" and "n": integer primes from two different arrays, p being the size of the array, and n being used in the hashing functions.
"a1" and "a2": sudo random integers from 1 to p, used in hash function, acting as a randomising force in the hash functions.
"max_loop": equal to p, it is the maximum number of times that the insert function will be allowed to loop attempting to find a home location for a pair before calling the "rehash()" function.
"t1" and "t2": two tables stored as "vectors" [6] of "pairs" [7] initialised to "NULL_PAIR", which is a pair defined at the top of the "cuckoo.cpp".

Lookup

The "lookup()" method takes a string as a key, and returns the key-data pair. To achieve this, the first hash function("hash()") is performed on the key, and if this matches the key stored in the first table("t1"), the pair is returned. Otherwise, the second hash function("hash2()") is performed on the key, and if this matches the key stored in the second table("t2"), the pair is returned. Otherwise, the "NULL_PAIR" is returned.

Insert

The "insert()" method is the most complicated of the functions within this system.

First, the "lookup()" method is called to see if the key is already stored in either of the tables. If not, the function will enter a for loop which will loop "max_loop" times, which is equal to the first value of "p".

On each loop, the first index is found by hashing the key. The current pair("p") is swapped for the pair stored in the hash index of the first table. If this pair is the "NULL_PAIR", the function returns.

If this pair isn't the "NULL_PAIR", the index was occupied. This new pair was displaced and needs to be stored. The second index is found by hashing the key stored in the new pair using the second hash function. It is now swapped for the pair in the stored in the second index of the second table. If this pair is the "NULL_PAIR", the function returns.

This works because any pair will always be in either its first or second hash index, so can be looked up with just two probes.

If max loop is reached, the table may be too small to hold any more values. At this point, the "rehash()" method is called.

This method will run through every index in the two tables, and store all of the pairs in a vector called "pairs" using the "pushback" [8] method. This is to allow us to destroy and rebuild the current tables.

The method then checks is there are more total entries than there are buckets in each table. If this is the case, the "grow()" method is called. This method sets "p" and "n" to the next primes in the array, and then calls the "vector.reserve" [9] method on the two tables. This ensures they are at least large enough to contain the new bucket size(equal to the new "p" value). The tables are now filled with "NULL_PAIRS" to the size of "p", using "push_back". At this point the program will return to "rehash()".

"a1" and "a2" are now given new random values using "rand". [10] The pairs stored in "pairs" are then all reinserted using "insert", at which point we return to "insert". Finally, the "insert" function is then called for the pair which was stored in "p" the current pair.
This is demonstrated visually in Figures 7 and 8.

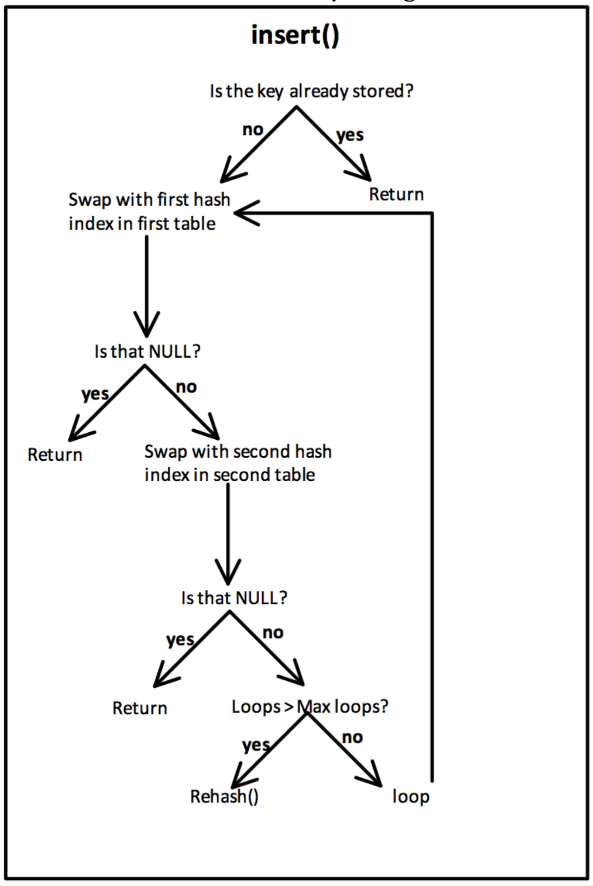
insert()
Is the key already stored?
no
yes
Swap with first hash index in first table
Return
Is that NULL?
yes
no
Return
Swap with second hash index in second table
Is that NULL?
yes
no
Return
Loops > Max loops?
yes
no
Rehash()
loop

Figure 7: insert() function

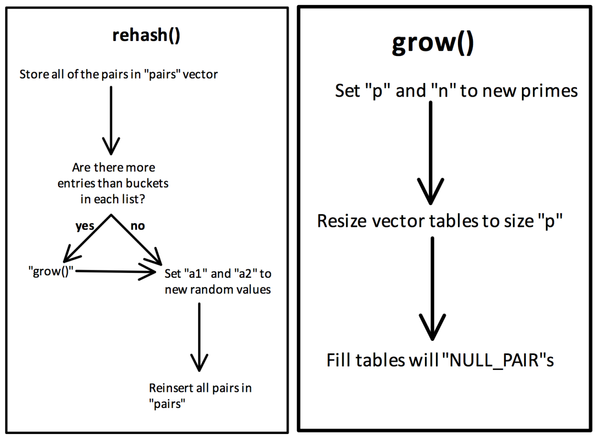
rehash()
Store all of the pairs in "pairs" vector
Are there more entries than buckets in each list?
yes
no
"grow()"
Set "a1" and "a2" to new random values
Reinsert all pairs in "pairs"
grow()
Set "p" and "n" to new primes
Resize vector tables to size "p"
Fill tables will "NULL_PAIR"s

Figure 8: rehash() and grow() functions

3.3 Review

Overall this is a clever, well-designed algorithm. It performs reasonably in terms of coverage. The "rehash()" function is particularly well designed. Although it is computationally intensive to reinsert the entire table, if the loop called by "insert()" generates a closed loop (a small number of keys could be looped over the max number of times), the "grow()" function will only allocate more memory if it needs it to fit the current amount of data, otherwise it will just reinsert all of the pairs again, using the new "a1" and "a2" values which will result in the keys being stored in new indexes and should avoid another closed loop being created.
A negative of this algorithm is that it uses two tables of the same size to store the pairs, but when the tables have a low load, the second table is rarely used, and more memory will be allocated (if the rehash function is called) if the total number of entries is equal to size of just one array, meaning that load may not go much higher than 50%.
It's also conceivable that another type of closed loop could be created, where the "rehash()" function is called continuously, but the "grow()" function is never called as the list is not full. This seems unlikely, or completely impossible with the use of "rand()" for the generation of "a1" and "a2" but it is worth consideration.

3.4 Improvements

1. Simply adding more hash tables and more salt values("a1" and "a2", values used to randomise a hash), it may be possible to reduce the necessary number of calls of "rehash()" and "grow()". It would be sensible in this case to change the size required for the "grow()" function to be called to something higher than the size of one table, to avoid the situation described above where the max load is inversely proportional to the number of tables.
2. The addition of more dynamic a scaling element to the system may be beneficial. A system could be designed such that instead of always having two tables of an identical size, when the "grow()" function is called, it creates a new table which has a size one prime index higher than the current highest table and creates a new value "a(n+1)" to hash it with. This table now becomes the "top" table, which is checked first. All other tables would be checked in descending order of size, as each table down would be smaller, but also have a lower table load. It may be sensible to limit the number of tables to some small number, to prevent an unnecessary memory requirement. The "bottom" table could be culled by the grow function.

Bibliography

[1] A. M. Tenenbaum, Y. Langsam and M. J. Augenstein, Data Structures Using C., Englewood Cliffs, N.J.: Prentice Hall. , 1990.

[2] D. G. A. M. K. Bin Fan, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, 2013.

[3] Efficient Computing at Carnegie Mellon, "efficient/libcuckoo," September 2016. [Online]. Available: https://github.com/efficient/libcuckoo. [Accessed October 2016].

[4] A. A. Kvernmo, "cuckoo/src/," 2014. [Online]. Available: https://github.com/augustak/cuckoo/tree/master/src. [Accessed 2016].

[5] D. G. A. M. K. M. J. F. Xiaozhou Li, "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing," in *EuroSys*, 2014.

[6] CPlusPlus, "std::vector," [Online]. Available: http://www.cplusplus.com/reference/vector/vector/. [Accessed 2016].

[7] CppRefence, "std::pair," [Online]. Available: http://en.cppreference.com/w/cpp/utility/pair. [Accessed 2016].

[8] CPlusPlus, "Vector::push_back," [Online]. Available: http://www.cplusplus.com/reference/vector/vector/push_back/. [Accessed 2016].

[9] CPlusPlus, "Vector::reserve," [Online]. Available: http://www.cplusplus.com/reference/vector/vector/reserve/. [Accessed 2016].

[10] CPlusPlus, "cstdlib/rand," [Online]. Available: http://www.cplusplus.com/reference/cstdlib/rand/?kw=rand. [Accessed 2016].