# COMP20007 Assignment 2: Dynamic Hash Tables
# Written Tasks

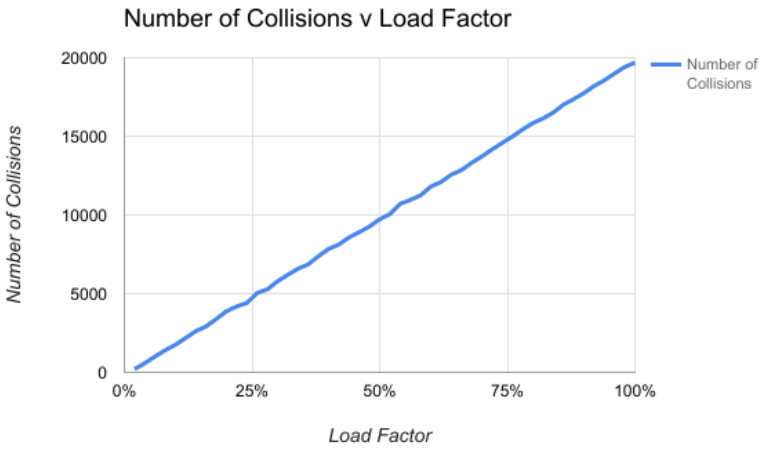
Liam Aharon
Student ID 835596

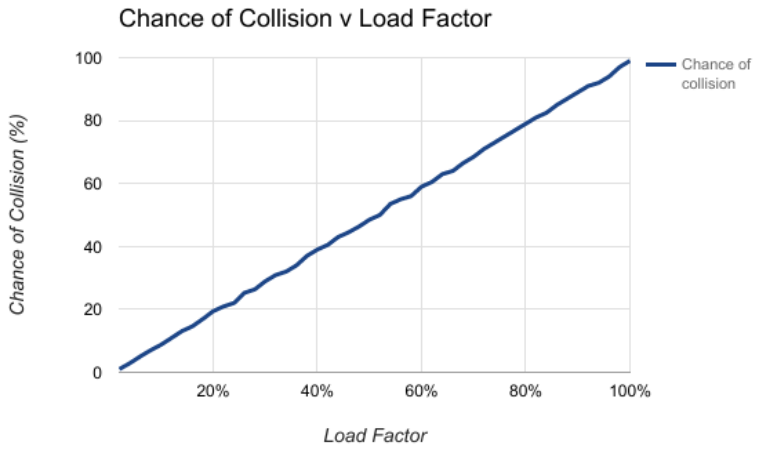
---

## Part 4: Load Factor and Collisions

To get data to answer this question, linear.c was modified to collect the information about each insert, grouping information by what the load factor was during the insert. Just under one million random keys were then generated using 'cmdgen' then run under the linear hash tables program[1]. A visualisation and analysis of the results can be found below.

### Number of Collisions



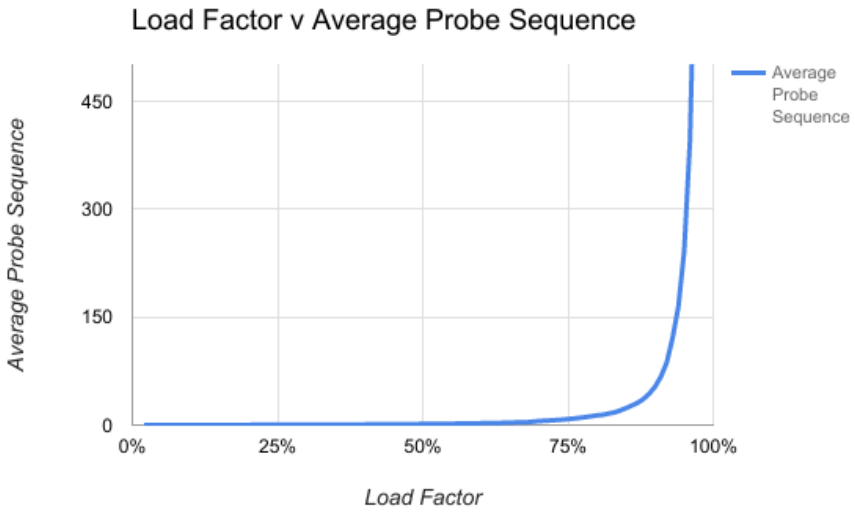




A clear linear relationship between load factor and chance of collision and an almost identical relationship was found between load factor and number of collisions. This result was expected, as due to the address of each key being hashed each key should be inserted into a random position. Therefore the chance of a collision occurring should be equal to the load factor of the table. As the load factor increases, there is more chance of a collision occurring therefore should be more collisions occurring as the load factor increases.

---

[1] To avoid discrepancies during these calculations, the table must not double. To achieve this, the starting size of the table must be set to be the same as the input. Matt confirmed this is OK on the discussion forum.

### Length of average probe sequence

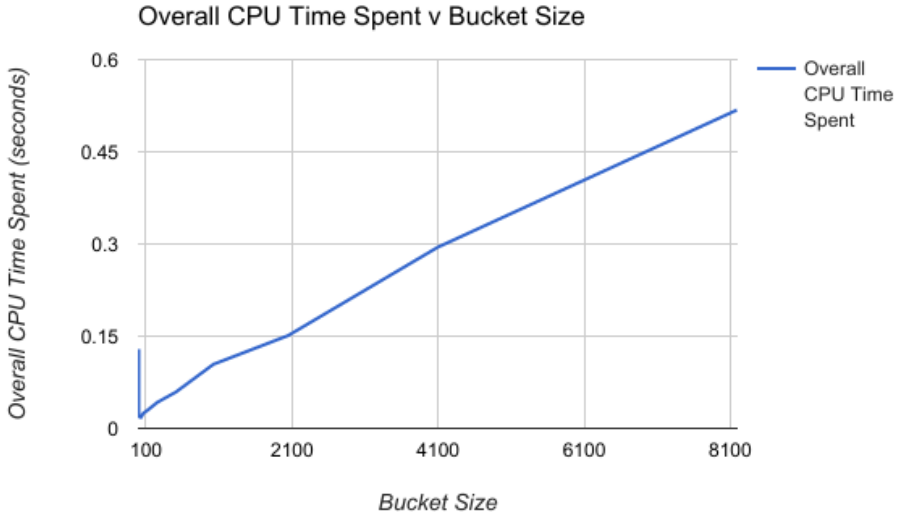



A clear asymptotic relationship exists between load factor and the average probe sequence during insertion. The probe sequence is acceptable for load factors < 75~%, however once it passes this point it quickly tends to infinity. The average probe sequence for a 99% load factor (does not fit on the table) is 51966.42! This result was expected, because for each probe the probability for the next slot to be empty is $1-l$. (where $l$ is load factor). We can probe $n$ times to form a probe sequence of length $n$, making the probability to find an empty slot in such a sequence $(1-l)^n$. It's easy to understand from this equation that for load factors < 75~% the value of $n$ does not need to be large for the probability of encountering an empty slot quickly become very large, while for larger load factors > 90~% the value of $n$ required for a reasonable probability grows to infinity. The graph is a very valuable visualisation, as it shows in an easy to understand way the penalties for allowing a load factor of a linear hash table to grow too large.

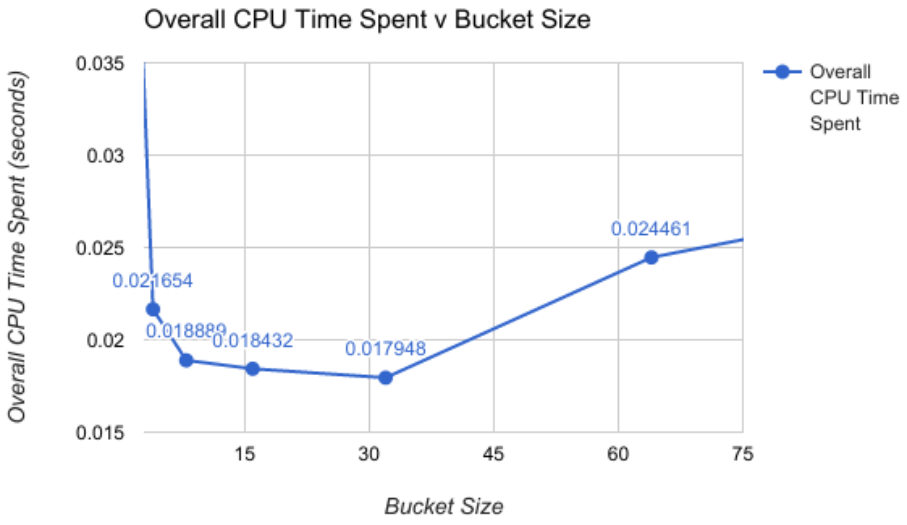
## Part 5: Keys per Bucket

To get data to answer this question, xtndbln.c was modified to record overall time spent inserting and looking up keys. 10,000 random keys were generated using 'cmdgen' and input into the multi-key extendible hashing program multiple times, using different bucket sizes that are 2 to the power of n in size. Each test was run 10 times, and the average taken. A visualisation and analysis of the results can be found below.

From the first graph generated showing the run time for bucket sizes all the way from $2^0$ to $2^{13}$ (1 - 8192) that there appears to be a sharp drop sometime between a bucket size 0 and 90, and then a continuous linear increase as the bucket size tends towards infinity. In order to identify the minimum point, we will need to reduce the bounds



of the x-axis to closely examine this minimum.




After some experimentation it was found that reducing the x-axis to show values from 3 - 75 was an effective way to visualise the minimum point, that could not be accurately discerned on the graph with a larger x-axis range. We can clearly see from this restricted graph that the program is able to run the fastest when the bucket size is $2^5$ (32). Tests were ran to make sure the true minimum was not somewhere in between $2^5$ and $2^6$. CPU time spent was measured for every bucket sizes in between 32 to 64, and at no time did the average of the 10 runs used come out lower than 0.017948 seconds.

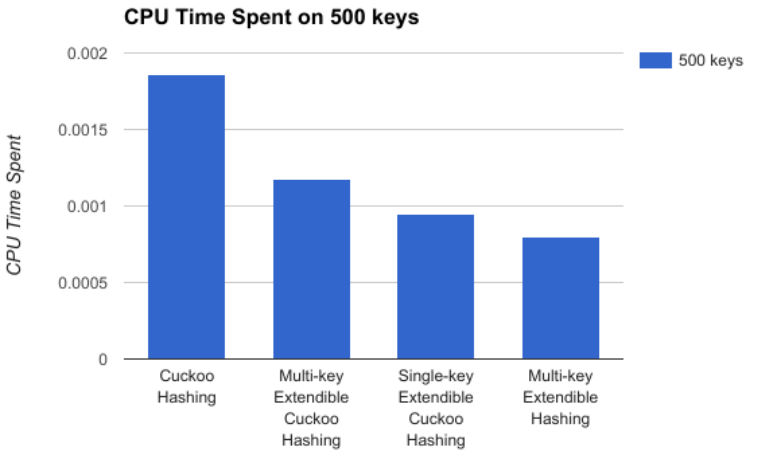
Results like this were expected, as if the bucket size is too low the table will spend too much time doubling, and reinserting keys, decreasing performance. If the bucket size is too high the table will spend too much time linearly searching through each bucket. A happy medium between must be found where the benefits of an O(1) lookup time is not ruined by a large bucket size causing long linear scans, but the table doesn't need to spend lots of CPU time doubling in size so often. This the perfect bucket size happens to be 32 from the findings of this investigation.

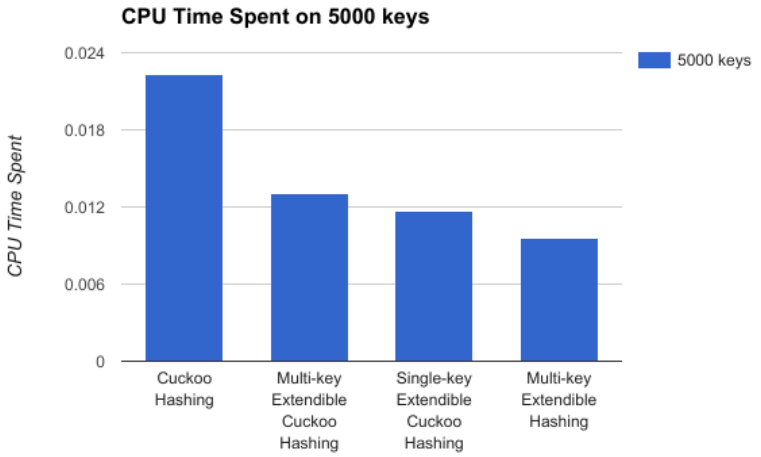
## Bonus Challenge

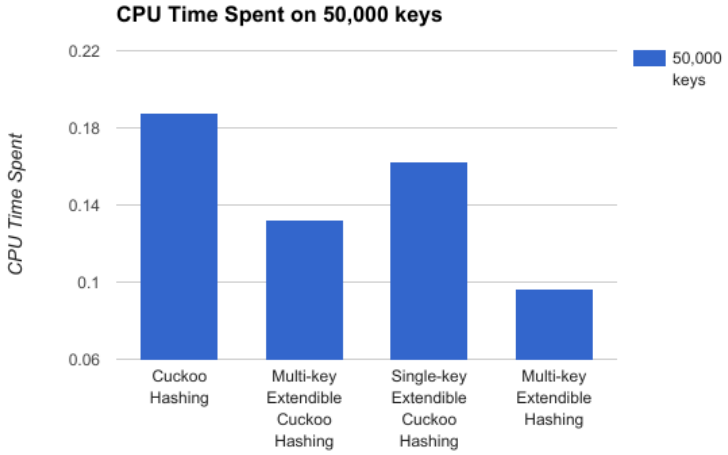
For this test, I generated 3 test inputs. 500 keys, 5000 keys, 50,000 keys and 100,000 keys. I chose not to use any smaller test sizes as the CPU time was too inconsistent between runs. I ran the tests on each of Multi-key Extendible Cuckoo Hashing[2], Single-key Extendible Cuckoo Hashing (Part 3), Multi-key Extendible Hashing (Part 2)[2], and Cuckoo Hashing (Part 1), and averaged the CPU time spent over 10 runs.

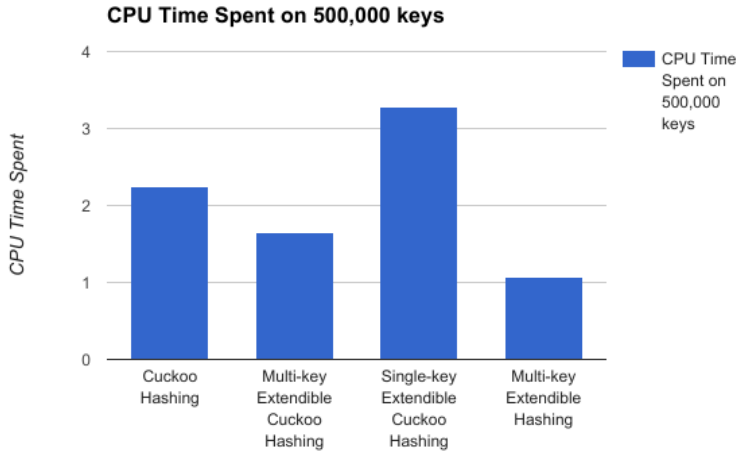
| | Cuckoo Hashing | Multi-key Extendible Cuckoo Hashing | Single-key Extendible Cuckoo Hashing | Multi-key Extendible Hashing |
|---|---|---|---|---|
| 500 keys | 0.00186 | 0.001177 | 0.000946 | 0.000795 |
| 5000 keys | 0.022275 | 0.013089 | 0.011693 | 0.00959 |
| 50,000 keys | 0.187629 | 0.13258 | 0.162781 | 0.0968 |
| 500,000 keys | 2.24139 | 1.652151 | 3.289329 | 1.068423 |

[2] Using bucket size of 32




## Analysis

Multi-key Extendible Hashing is the fastest hashing algorithm of all being compared, however due to its nature uses lots of space so there is a trade off here. Single-key Extendible Hashing is fast for smaller inputs but does not scale well, performing 2nd best for 500 keys, but worst during the 500,000 keys test. Multi-key Extendible Cuckoo Hashing performs the opposite of Single-key Extendible Hashing, starting off slow but performing better for larger inputs, still using a lot of memory. Cuckoo Hashing performs the poorly however it is still is able to keep up with the two multi-key extendible hashing algorithms. It is worth pointing out that although it is not the best choice for time complexity, cuckoo hashing uses far less memory than the other algorithms so it should be absolutely be considered if memory is a constraint in the system running the algorithm.