Design of Algorithms Assessment 2 Report

Part 4: Load Factor and Collisions in a Linear Hash Table

To create these results, I used **`cmdgen`** to generate 20 sets of sample inputs, and analysed the relationship between the number of collisions and the average length of a probe sequence with load factor using a linear hash table. To experiment on these results, I loaded a csv containing the results into a Jupyter Notebook for analysis.

First, I tested inserting 20 sets of values into the table, incrementing by 100 each time. The 20 sample inputs showed that the inserted items had a nearly linear correlation with collisions (**Figure 1**), resulting in a Pearson Correlation r ≈ 0.99. Working off this, I then analysed Collisions vs Load Factor (**Figure 2**). The results seemed to show a linear trend that reset

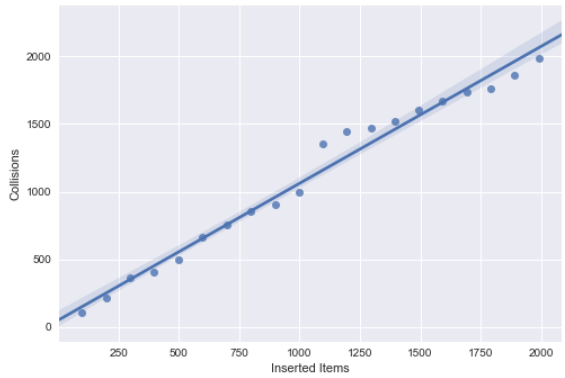

Figure 1

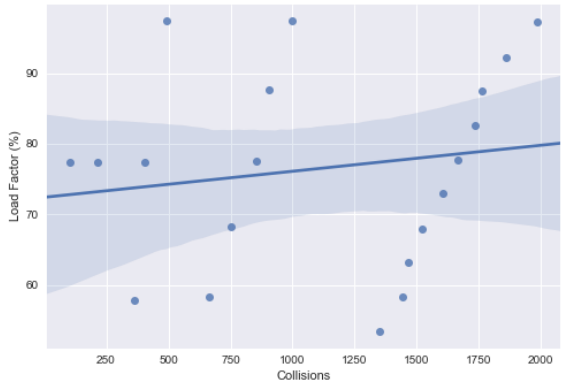

Figure 2

every number of collisions (looking at the ranges of 250-500, 600-1000 and 1250-2000). The load factor % seemed to change depending on the number of insertions, so to check on this, I restricted the number of insertions to the ranges (500-1200) and (1200-2000).

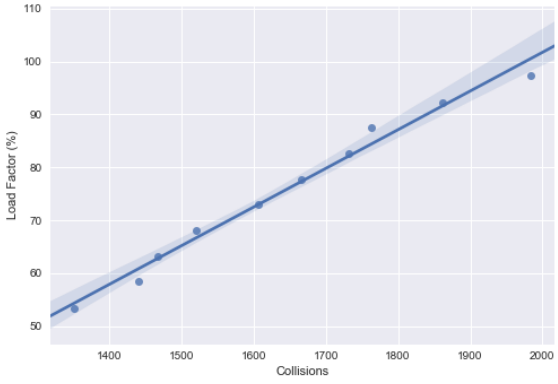

Figure 3

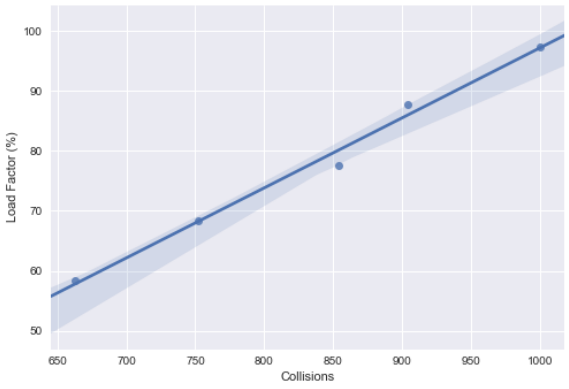

Figure 4

As you can see from **Figure 3 and 4**, there is a nearly perfect linear correlation – both graphs with r ≈ 0.99, suggesting that the number of collisions correlates linearly with load factor in a hash table, *depending on the number of slots allocated relative* to the number of inserted items. The two values at the beginning (<250 collisions) of the graph also support this, as the number of slots doubles twice within these points, and as such there is a single 'point' for the trend line.

I then plotted Average Probe distance vs Load factor (**Figure 5**). It seemed as there was no trends as with collisions (r ≈ 0.006). However, looking at the raw data, it seemed to be that when there were more than 1024 slots, as the load factor approached ≈ 77% load, the average probe would reach a local minimum within that set (**Figure 6**). This trend seems to exhibit a wave like behaviour; in the 1024-2048 slot bracket (**Figure 7**), the average probe rate starts at 19, goes up to 21, up to 27, before starting to fall to reach the local minimum of 17.456. This trend repeats again afterward, increasing to 20.618 again.

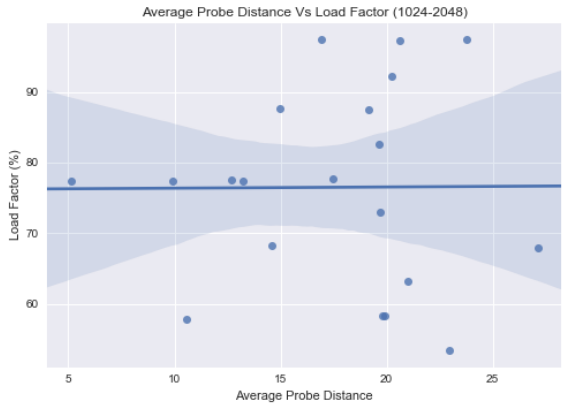

Figure 5

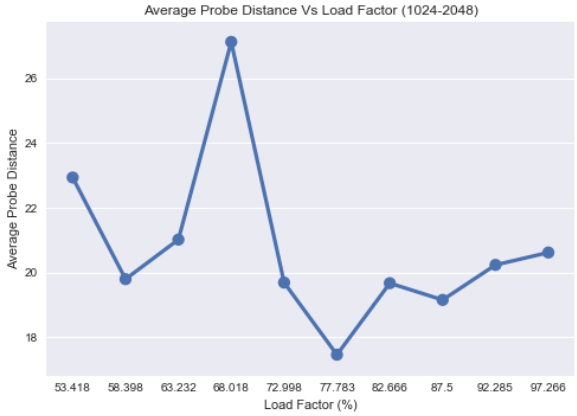

Figure 7

Collisions	Average Probes	Load Factor	Loaded Items	Total Slots
106	5.113	77.344	99	128
215	9.902	77.344	198	256
363	10.565	57.812	296	512
403	13.208	77.344	396	512
495	16.915	97.461	499	512
663	19.929	58.301	597	1024
752	14.608	68.359	700	1024
854	12.685	77.637	795	1024
904	14.949	87.695	898	1024
1000	23.756	97.363	997	1024
1352	22.939	53.418	1094	2048
1441	19.795	58.398	1196	2048
1467	21.018	63.232	1295	2048
1520	27.134	68.018	1393	2048
1606	19.689	72.998	1495	2048
1666	17.456	77.783	1593	2048
1732	19.67	82.666	1693	2048
1763	19.152	87.5	1792	2048
1861	20.235	92.285	1890	2048
1984	20.618	97.266	1992	2048

Figure 6

Part 5: Keys Per Bucket

To experiment with this, and generate meaningful CPU time for the `xtndbln` hash table, I used `cmdgen` to create another set of inputs. This time I just used one sample, containing 100,000 inserts and 100,000 lookups. I would then alter the -s flag to experiment with the CPU time.

I then plotted the results the same way I did in part 4 (**Figures 8, 9**) analysing bucket size vs number of buckets and CPU time.

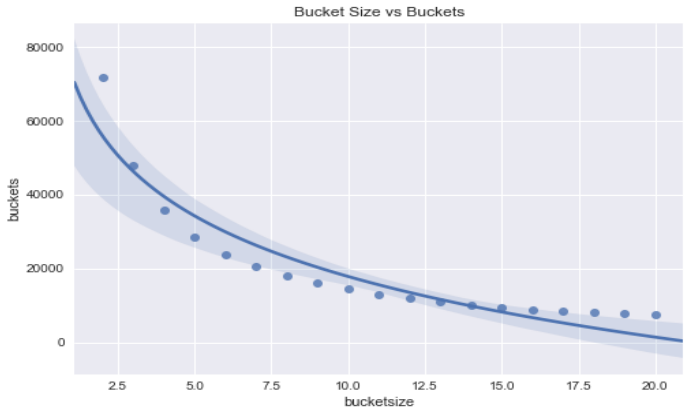

Figure 8

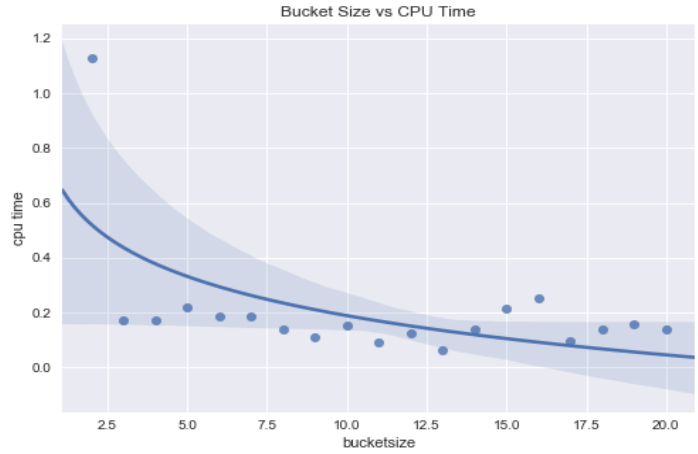

Figure 9

Looking at the trend, there seems to be an asymptotic trend starting from 1 bucket. In my testing, I was unable to test a `bucketsize` of 1 as I went over the memory limit. But from the samples it's fairly clear that the number of buckets scales asymptotically with bucket size, as does CPU time. In my testing the closer CPU times (3 buckets and onward) were more inaccurate as my computer was running under several conditions (other operations in the background), but there is still a clear difference from bucket size of 2 to 4. Strangely, a bucket size of 13 seems to be the fastest for 100,000 inserts and 100,000 lookups. I tried this with a variety of data and it this seemed to hold true. This is because as bucket size increases, then the lookup function needs more time to iterate through a bucket to do a search (and also to reinsert all keys in an old bucket). These small differences will add up over time, and the advantage of having to split less eventually cancels out with the size of the bucket, hence why the scaling is pseudo asymptotic.

Part 6: Multi-key Extendible Cuckoo Hashing (Bonus Question)

In my testing, I run the 4 different types of hash table against a few sample text inputs. These were all, again, using `cmdgen` to generate 10,000 inserts and lookups. Cuckoo was the most consistent, averaging ≈ 0.09 seconds, as growing the table is always consistent. `Xuckoon` and `Xtndbln` were the also fairly consistent, averaging ≈ 0.04 seconds, depending on how the table was grown. `Xuckoo` was by far the most inconsistent, as growing the table was more often and random in comparison to `Xuckoon` (**Figure 10**). This is also reflected in the load factor, as the consistency of `Xuckoon`, `Xtndbln` and Cuckoo were all fairly consistent at ≈ 30% load, while again `Xuckoo` was more different (**Figure 11**). This may be a result of how I've decided to split the bucket – instead of splitting until there is space my algorithm attempts to insert again before splitting again, as the insert function is a recursive function call.

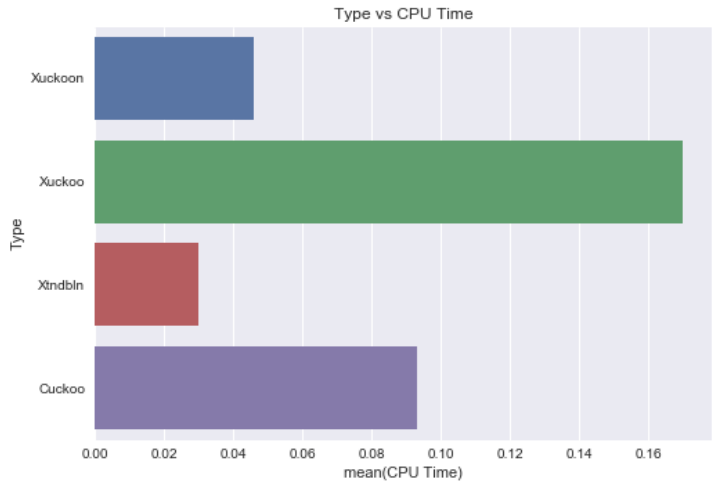

Figure 10

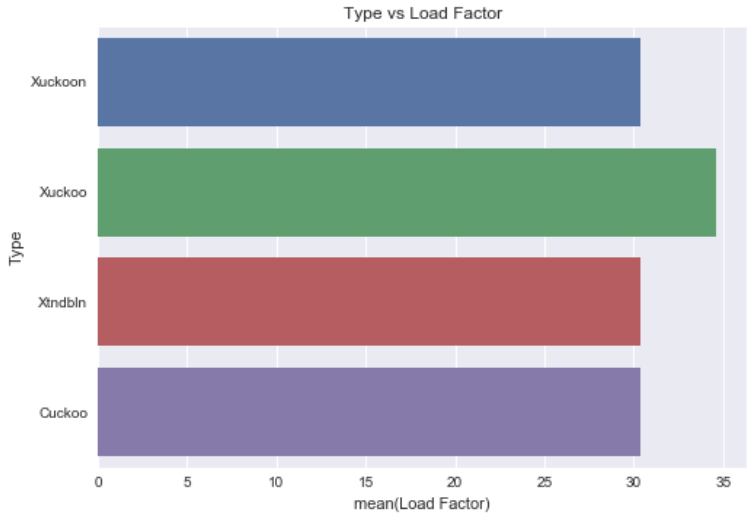

Figure 11