Ben Tomlin: 834198

Assignment 2: Dynamic Hash Tables

Load Factor & Collisions

The relationship between load factor and collisions of a linear probing hash table was investigated by plotting data for a tables average load, collision ratio and average probe sequence length. Data was gathered by running the assignment program repeatedly for a variety of inputs. Statistics were generated by the program, with functions added to the linear hash module to print to stdout table statistics in comma separated format. Bash was used to automate running the program with many inputs. Jupyter-Notebook, Python and Matplotlib were used to visualize the resulting data.

The load factor is defined as the number of keys inside the table divided by the key capacity of the hash table. The average load factor was used in place of the load factor because a load factor in itself only represents the current state of the table, not its history. To meaningfully compare the load factor to average probe sequence length, an average load factor that represents the cumulative set of loading states for the table is necessary. A collision factor was used instead of the count of collisions, since data was generated by a non constant insert command set size, it had to be normalized for comparision. These statistics are defined below;

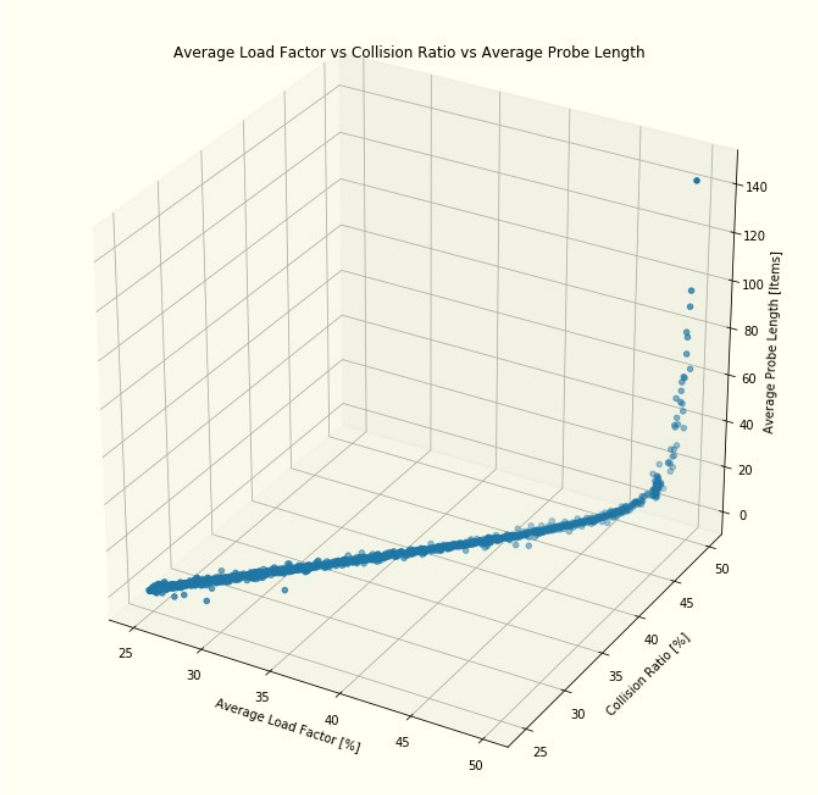
$$average\,load\,factor = \frac{\sum load\,factor\,after\,insert\,i}{number\,of\,inserts}$$

$$collision\,ratio = \frac{collision\,count}{number\,of\,inserts}$$

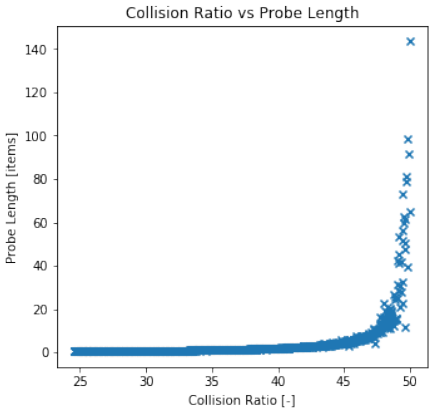
Data generation used as the set of initial table sizes, the magnitude ordered set of all positive integers less than 32 such that any member of the set is not equal to any previous member of that set multiplied by a factor of two raised to some positive integer power. This ensures unique table sizes at each doubling of the hash table. The program was run for each initial table size for a variety of sizes of input command sets.

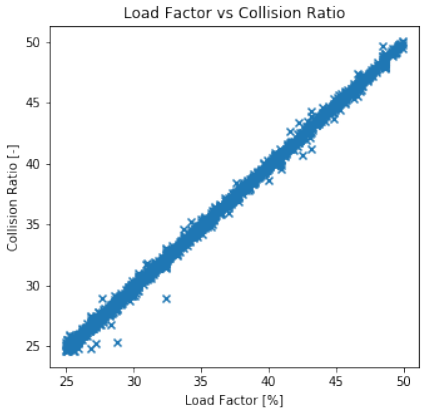
The 3D scatter plot presented on the following page shows that collision ratio and average load factor are linearly related and both average load factor and collision ratio are exponentially related to average probe length.

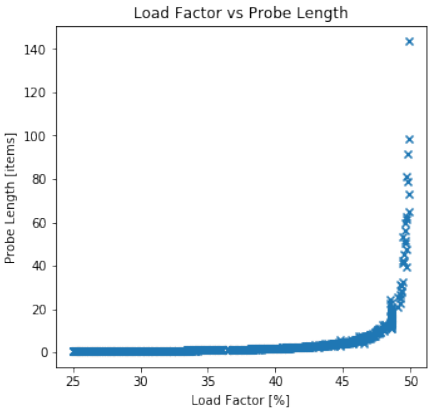
The linear relationship between the collision ratio and average load factor indicates that the hash function has the property of uniformity. If a hash function distrbutes values randomly over the table, then load factor indicates the probability that any slot a key is hashed to will be full. That is, the probability of a collision.



The exponential relationship between average probe length and load factor is consistent with a uniform hash function. Intuitively, for a uniformly sparse table, we will almost always not need to probe. For a uniformly dense table, we almost always need to probe. The probability that any given cell is full is the load factor. Summing all the possible probe length probabilitys weighted by the probe length and dividing by the number of possible lengths could give us a Euler sequence which will converge to an exponential function. Similarly for the collision ratio, which is just an alias for the averaged load factor. Below are presented three more graphs showing more clearly these relationships.







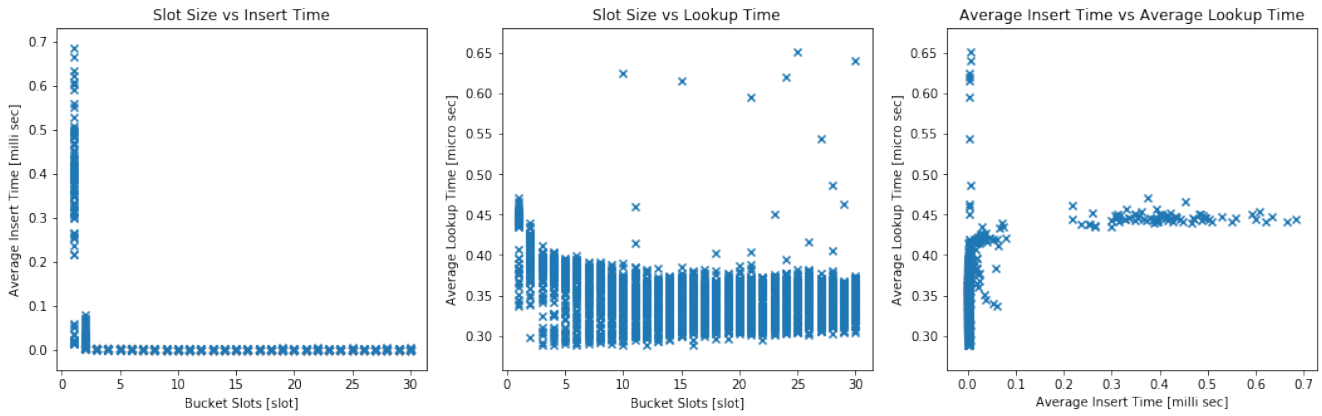
Note that average load factor never exceeds 50%. This can be derived from its formula by placing a lower bound on the number of inserts.

Ben Tomlin: 834198

Keys Per Bucket

Extendable hash table performance was investigated by plotting data gathered for lookup and insert performance for a variety of bucket sizes. Data was gathered by the same methods as per the previous section, but table sizes were replaced with bucket sizes, and the bucket sizes tested were the set of integers [1..30].

Lookup and insert cpu time accumulators were added to the extendable hash table module. An average time required for insertion was calculated by dividing the accumulated insert time statistic by the number of insert operations. The same was done for lookup operations. Below are presented scatter graphs illustrating a constant time requirement for lookup and insert operations, mostly independent from bucket size whenever bucketsize is three or greater. Insert time is the time it takes for a complete insert operation, including all subtasks such as splitting the table or lookups. Lookup time is the time taken for a lookup operation.



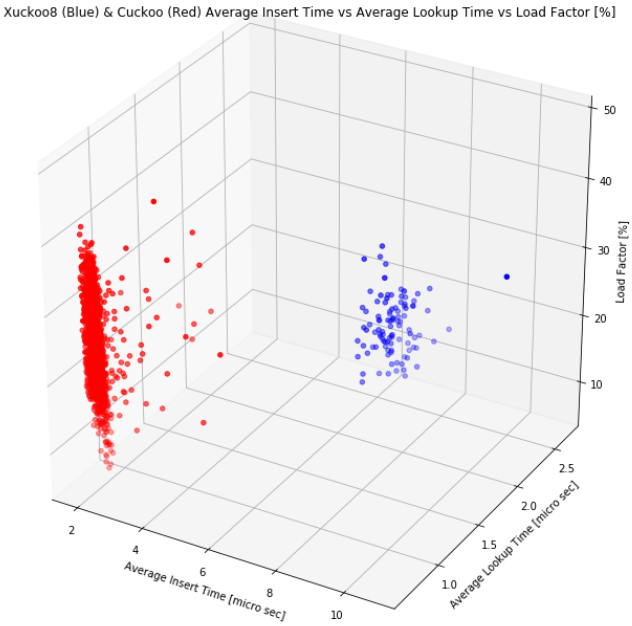
Insert v Lookup - For higher insert times, lookup time is constant. Higher insert times occur only for tables with bucket size of 1 or 2 slots. Lookup time is constant because for single slot tables, insertion is always hashing the value and going to an address in memory.

Slots v Insert - Average insert time for buckets with one or two slots has large variation. This could be because small buckets require more table splits and these are computationally expensive. Larger bucket sizes show a constant insertion time, since table splits are less now a negligible computational component of the insertion operation.

Slots v Lookup – Smaller bucket size tables have higher average lookup times. This could be because smaller bucket sizes increase the likelyhood of table splits. This results in larger tables with more spread out buckets. Thus, more computation and is done and larger blocks of memory must be jumped over. Buckets with many slots show no increase lookup time which

suggests scanning small blocks of memory (bucket slots) takes negligible time. Some noise is apparent possibly due to the micro second time scale increasing sensitivity to external factors.

Extendible Cuckoo Hashing



Cuckoo v XuckooN - Datasets generated by the previous method for cuckoo and 8 bucket hash tables were generated. Average insert time, lookup time and load factor were plotted. Results are shown to the right. We see cuckoo is approximately twice as fast as xuckoon. Perhaps this is because xuckoon requires additional memory fetches. One for a bucket address, one for the bucket itself. The tradeoff is cuckoo requires more storage than xuckoo.

Some improvements on this analysis would be to use average load factor.

XuckooN – Data was generated for xuckoon tables with one to eight bucket slots. Due to the table split condition of the program, small bucket tables have low load factors, and higher bucket tables have higher load factors. A better split condition should be devised that takes into account how many buckets in a table. Then a comparision of performance for tables of same load factor and varying bucketsize could be done for a more meaningful comparision.

We see that low bucket sizes perform poorly in insert time compared to higher bucket sizes. To fix this a better split condition is needed. Lookup time is constant.