Lab 4 Report
EECS 560
2/25/19
Evan Brown

Organization of Experimental Profiling

There were a number of moving parts that had to be coordinated in order to measure the performance of each hash table. First, of course, were the hash tables themselves, which must be working perfectly in order to even think about measure how *well* they work. The random number generator used to populate the tables also must be calibrated to make sure that the data being supplied to the hash tables in order to check their performance is as random as possible and does not skew the results by holding to some sort of pattern. Finally, the timing functions used to directly measure the performance of a single operation must also be implemented correctly, to make sure that valid time readings are obtained.

Generating Random Input Data

For this lab, I wrote a small wrapper class for managing random number generation. It contains a *seed* function, which takes an unsigned integer as a seed for the random generator; a *seedTime* function, which seeds the generator using the current CPU time; a *get* function, which returns an unbounded random integer; a *getFromZeroTo* function, which returns a random integer from zero to a given maximum; a *getFromOneTo* function, which returns a random integer from one to a given maximum; and a *getRange* function, which returns a random integer between an upper and lower bound. I also included functionality for specifying whether the return ranges should be inclusive or exclusive of their bounds.

Seeding in random number generation is the most useful way to counteract the intrinsic determinism of computers. Since computers are only pseudo-random and generate random numbers in the same (extremely large) sequence every time, specifying where to start in the sequence by seeding the random number generator can ensure that the generated numbers are as random as possible. In this lab, before generating a set of random numbers to insert into the hash tables, a call to *seedTime* was made to make sure that unique numbers were generated for every test of the hash table.

CPU Time Recording in C++

I also wrote a simple wrapper for timing functions in C++. It has basic *start*, *stop*, *reset*, and *get* functionality, as well as a *getMS* function. The timing library measures times in CPU cycles, so in order to obtain durations that make sense to humans, the raw time values are multiplied by 1000 and divided by the constant CLOCKS_PER_SEC in order to obtain timing values in milliseconds.

Data Recording and Analysis

For each type of hash table, the following process was performed:

1. Create a hash table of size m (in this case m = 1,000,003).
2. Start the timer.
3. Insert a random number between 1 and 5m into the hash table.
4. Stop the timer.
5. If the insert was successful, add the time taken to the total build time. Otherwise, discard it.
6. Repeat steps #2 to #5 0.1*m times.
7. Generate another random number between 1 and 5m.
8. Start the timer.
9. Find the number in the hash table.
10. Stop the timer.
11. If the number was successfully found, add the time taken to the total 'found' time. Otherwise, add it to the total 'not found' time.
12. Repeat steps #7 to #11 0.01*m times.
13. Repeat steps #1 to #12 4 more times, and record the average total build, found, and not found times for the 5 trials.
14. Repeat steps #1 to #13 4 more times, but replace 0.1m in step #6 and 0.01m in step #12 with 0.2m & 0.02m, 0.3m & 0.03m, 0.4m & 0.04m, and 0.5m and 0.05m.

Performance Comparison & Summary

The recorded data was as follows:

Open Hashing with Separate Chaining:

# elements	**0.1m**	**0.2m**	**0.3m**	**0.4m**	**0.5m**
Build Time	110.131 ms	219.012 ms	324.622 ms	427.413 ms	515.562 ms
Found Time	0.187 ms	0.7212 ms	1.6176 ms	2.912 ms	4.395 ms
Not Found	9.0126 ms	17.6116 ms	25.954 ms	34.0492 ms	41.1948 ms

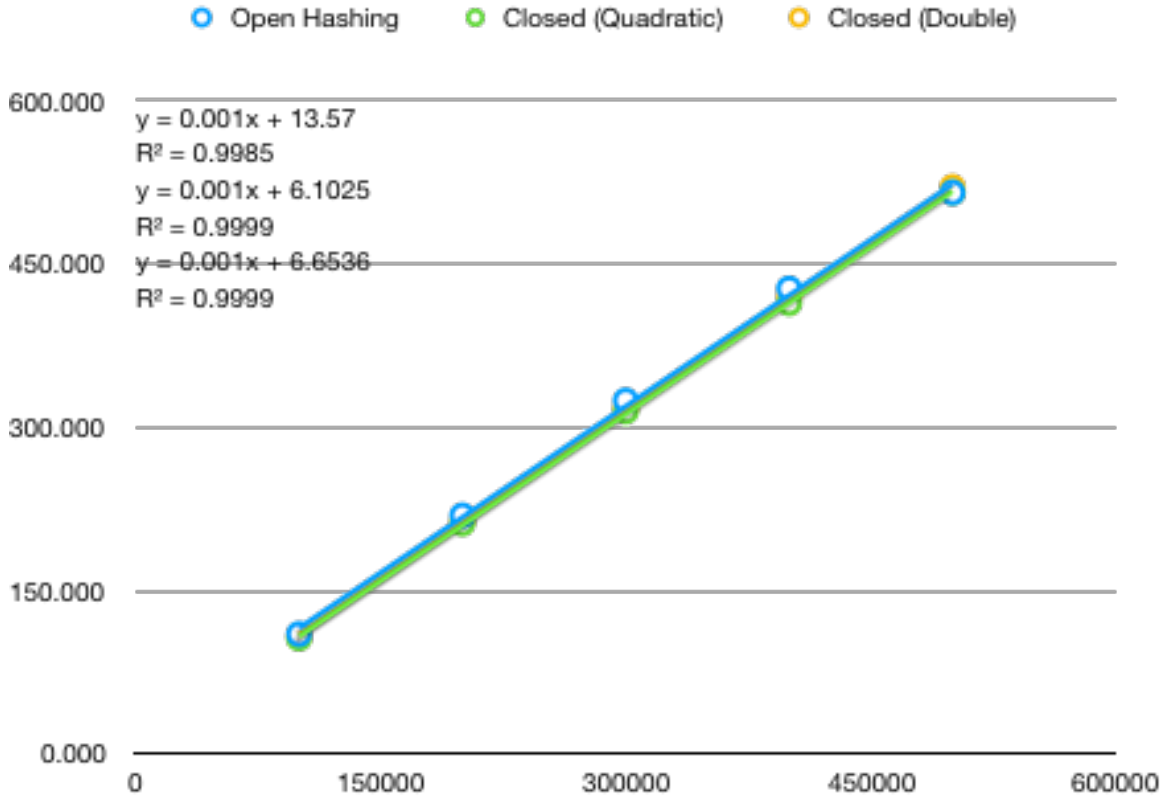
Closed Hashing with Quadratic Probing:

# elements	**0.1m**	**0.2m**	**0.3m**	**0.4m**	**0.5m**
Build Time	106.271 ms	211.743 ms	314.725 ms	414.073 ms	515.81 ms
Found Time	0.1808 ms	0.7356 ms	1.5532 ms	2.8284 ms	4.3876 ms
Not Found	8.6376 ms	17.033 ms	25.287 ms	33.8436 ms	40.968 ms

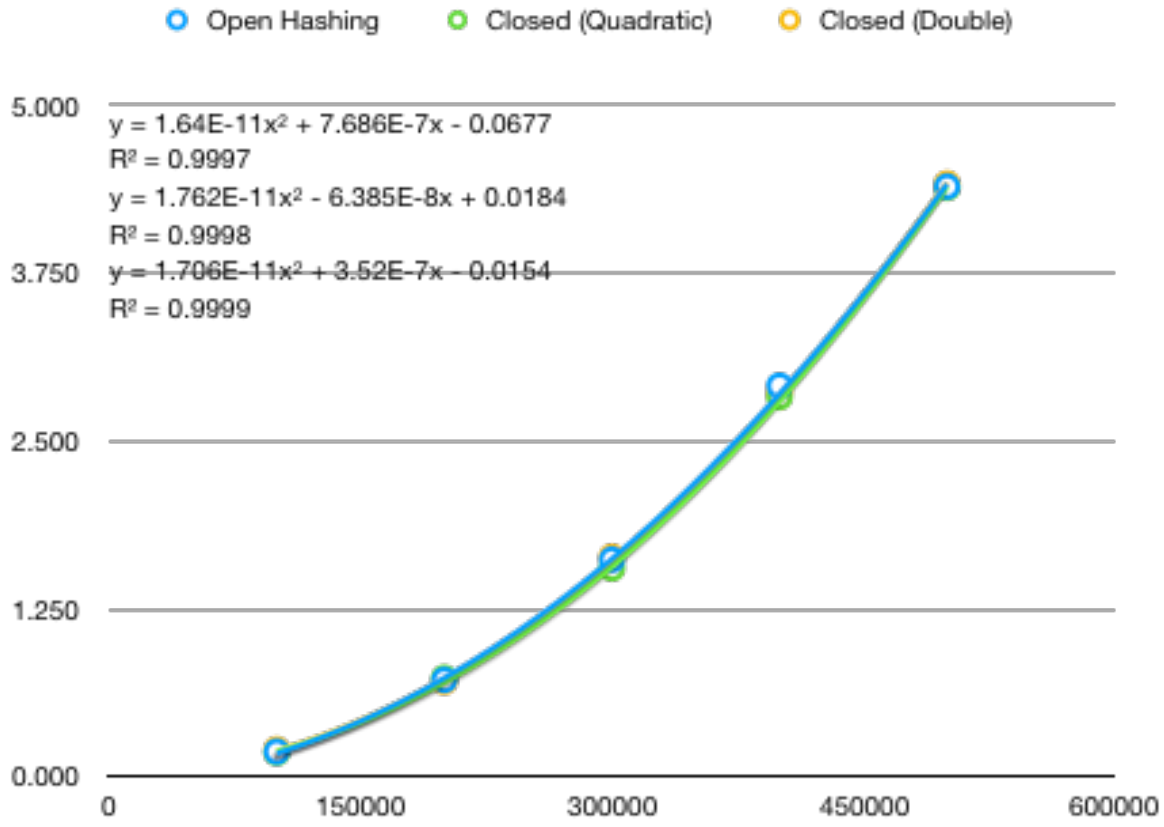
Closed Hashing with Double Hashing:

# elements	**0.1m**	**0.2m**	**0.3m**	**0.4m**	**0.5m**
Build Time	108.076 ms	213.937 ms	317.491 ms	420.128 ms	520.654 ms
Found Time	0.2026 ms	0.7088 ms	1.6374 ms	2.8666 ms	4.4164 ms
Not Found	8.8454 ms	17.4662 ms	25.9532 ms	34.2318 ms	42.4254 ms

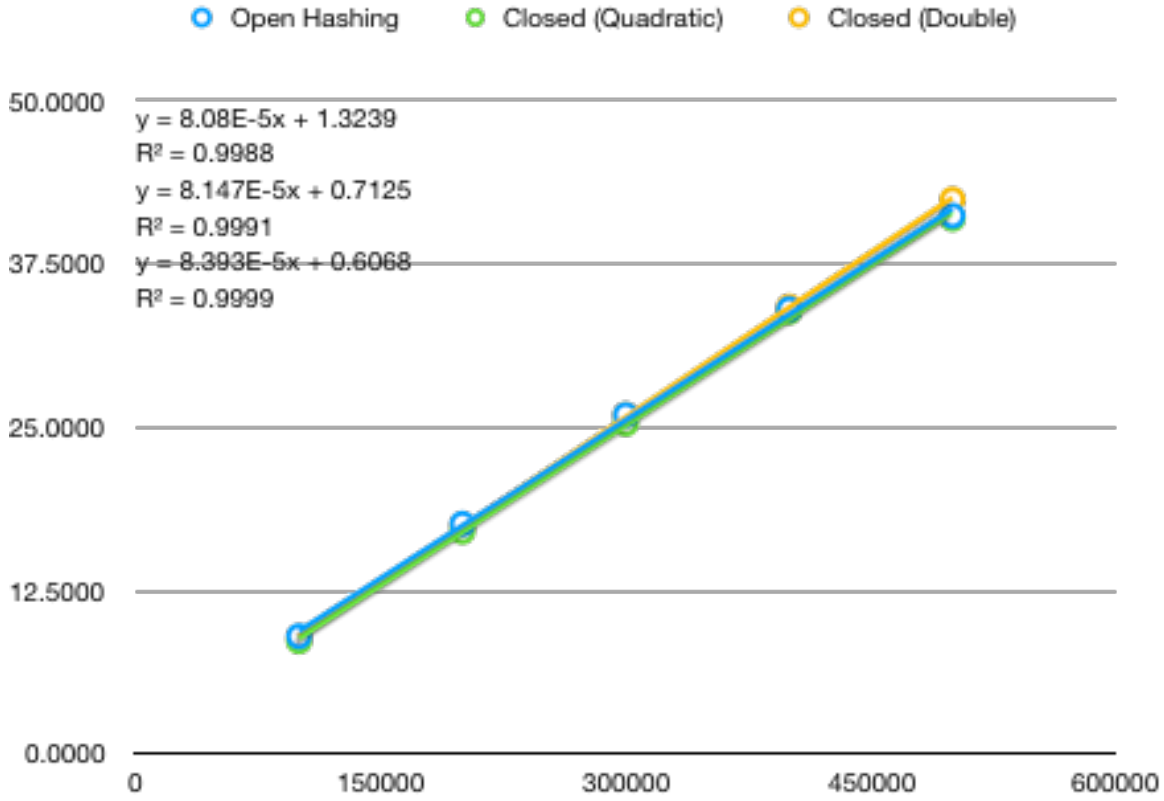
Build Times:



Found Times:



Not Found Times:



As can be seen, the timing data for all data points was virtually identical between the three types of hash table, suggesting that if there is a difference in performance between the three, the sizes of hash tables used was not large enough for one to be measured.

The data suggests that build times and not found times increase linearly with more elements, and found times increase with a higher complexity—there are not enough data points to tell more than that it is nonlinear.

The time required to compute a hash function is some constant C, which is O(1). Since our data is approaching random, we can assume it is randomly distributed throughout the table. For an open hash table, on average a bucket will contain x/m elements, where x is the number of elements in table and m is the table size. When inserting, every element in the bucket must be checked to avoid duplicates. Therefore an insertion is O(x/m) complexity. For a closed hash table, the chance of a collision in a bucket is also x/m, and the chance of no collision is 1-(x/m), so the time required to insert will be:

$$0\left(1-\frac{x}{m}\right)+1\left(\frac{x}{m}\right)\left(1-\frac{x}{m}\right)+2\left(\frac{x}{m}\right)^2\left(1-\frac{x}{m}\right)\ldots=\sum_{i=0}^{\infty} i\left(\frac{x}{m}\right)^i\left(1-\frac{x}{m}\right)=\frac{x}{m-x}$$

$$\approx \frac{x}{m}\ when\ x \ll m$$

For the combined insertion of y items, we must find the combined complexity, which will be as follows:

$$\sum_{i=1}^{y} \frac{i-1}{m} = \frac{1}{m}\left(\sum_{i=1}^{y} i - \sum_{i=1}^{y} 1\right) = \frac{1}{m}\left(\frac{y(y+1)}{2} - y\right) = \frac{y^2 - y}{2m}$$

This result indicates that we should expect quadratic complexity from build times. The discrepancy between the expected complexity and the observed is due to duplicates. As the table fills up, the chance that an item is a duplicate will increase, and the less overall time the build will take.

For a filled out hash table with x entries, a find will take the same amount of time as an insertion. The probability that an element is found in a table with y elements is y/5m. We're doing 0.1y finds in a table with y elements, so the complexity will be as such:

$$\sum_{i=1}^{0.1y} \frac{y}{m} * \frac{y}{5m} = \frac{0.1y^3}{5m^2}$$

This seems to be in line with the experimental data for found times. The complexity for not found times will be very similar, except the probability is 1-(y/5m):

$$\sum_{i=1}^{0.1y} \frac{y}{m} * \frac{5m - y}{5m} = \frac{0.5my^2 - y^3}{5m^2}$$

In this case, the offsetting y^3 and y^2 values are what gives a linear complexity for smaller values of y, as shown in the experiment.

Conclusion

All of the components for this lab were relatively easy to implement, and the results showed the usefulness of hash tables for inserting and finding large amounts of data. The experimental results seemed to agree somewhat with the theoretical complexities for the algorithms tested, but it is difficult to draw many conclusions from the experiment, and the complexities were complicated by the possible insertion of duplicates into the table. If I were to redesign this experiment, I would include a much greater number of data points collected than just 5; such a small sample size was not enough to reliably gauge the complexity of each algorithm, or whether the complexities differed between hash tables.