Brian Bauman

Homework 5

**Q1:** What is the algorithmic complexity of creating/populating a hash table of size N?

Testing methodology:

In order to determine the algorithmic complexity, we will time the insertion of N keys into a hash table (of either the linear probing or separate chaining variety). We isolate the put() calls and remove I/O noise by loading the entire input file into memory before beginning the timer. Also,

for small values of N, these total times can be quite small. As such, we perform each set of insertions over 10 repetitions and use the average value as our resulting time for that value of N.

| N | elapsed (linear probing) (s) | elapsed (separate chaining (s) |
| --- | --- | --- |
| **0** | 0 | 0 |
| **50000** | 0.1707 | 0.0714 |
| **100000** | 0.5087 | 0.1913 |
| **150000** | 0.7568 | 0.2607 |
| **200000** | 0.9771 | 0.3385 |
| **250000** | 1.1605 | 0.3727 |
| **300000** | 1.2693 | 0.4355 |
| **350000** | 1.4765 | 0.5846 |
| **400000** | 1.5609 | 0.6221 |

Data:

Conclusions:

In conclusion, it seems as though both linear probing an separate chaining have an algorithmic complexity of O(N) when inserting N keys into an empty table. In the case studied, separate chaining performed about three times as fast.

Possible limitations:

Even after removing I/O costs from the timed data, there is still more processing happening within our timing block than the puts. We must access each key from the temporary array, which happens once at the begging of the for loop that drives the put() calls. The number of times we must access memory in this way also grows with N, but is unlikely to dwarf the processing responsible for our put() calls. As such, the resulting data should be enough to justify the algorithmic complexity.

**Q2:** What is the algorithmic complexity of inserting the N+1st key into a hash table of size N?

Testing methodology:

For this experiment, we needed to simulate an action that is, by nature, extremely quick. As such, we copied two useful tricks from the first question — removing the I/O overhead in our time calculations and using repetitions to get a clearer view into the actual amount of time that a single put takes. Due to the low fidelity with which the Stopwatch class measures very small increments of time, a concession that was made was to put 10 keys rather than a single key. The thought here is that the cost of putting the N+1st, N+2nd, N+3rd, …, N+10th keys are all relatively the same for large N. As such, this sacrifice allows us better comparison with the Stopwatch timer across different values of N.

| N | elapsed (linear probing) (100,000 puts) (s) | elapsed (separate chaining) (100,000 puts) (s) |
| --- | --- | --- |
| **0** | 0.0270 | 0.0170 |
| **50000** | 0.0850 | 0.0670 |
| **100000** | 0.2410 | 0.0880 |
| **150000** | 0.0020 | 0.0050 |
| **200000** | 0.8490 | 0.1960 |
| **250000** | 1.1570 | 0.0040 |
| **300000** | 0.0120 | 0.0070 |
| **350000** | 0.5920 | 0.3040 |
| **400000** | 0.3660 | 0.1130 |

Data:

This data shows the amount of time it took to insert 10 keys 10,000 times into a ST of size N, restoring the ST to size N between each set of 10 inserts.

Conclusions:

It’s apparent from the data that a call to put() runs in constant time, regardless of the size of the symbol table. While the trendlines do so a correlation with N, it is very weak and almost surely explained by noise. One interesting thing to note in the linear probing example is the data point at N = 250,000, which took much longer than the others. It is very possible that an array resizing took place in the 10 inserts at this table size, adding processing time that would otherwise not be necessary. Again, the separate chaining implementation seemed to perform better, in general.

Possible limitations:

The obvious limitations here have to do with the fact that we were not putting a single key, as this operation happens so fast that it’s difficult to obtain an accurate, direct measure with the tools available. As such, inserting 10 keys in a row (10,000 repeated times) was a close approximation. Another limitation has to do with the fact that, at a particular size of N, a single put() call will require the entire ST to be rebuilt. With the procedures used in this example, it is difficult to avoid or isolate this cost.

| N | elapsed (linear probing) (100K gets) (s) | elapsed (separate chaining) (100K gets) (s) |
| --- | --- | --- |
| **0** | 0.0030 | 0.0010 |
| **50000** | 0.0190 | 0.0620 |
| **100000** | 0.0180 | 0.0280 |
| **150000** | 0.0080 | 0.0330 |
| **200000** | 0.0130 | 0.0200 |
| **250000** | 0.0260 | 0.0150 |
| **300000** | 0.0460 | 0.0200 |
| **350000** | 0.0060 | 0.0310 |
| **400000** | 0.0160 | 0.0150 |

**Q3:** What is the algorithmic complexity of searching for a key in a hash table of size N?

Testing methodology:

For this question, we set up symbol tables of various sizes and performed 10 gets on the symbol table, repeated 10,000 times. We chose random keys as we were building each iteration of the symbol table, remembering them as the keys to gather once the table was built. By performing 10,000 repetitions, we are left with times that can be easily compared.

Data:

This data shows the amount of time required to get 10 keys, repeated 10,000 times, from a symbol table of size N

Conclusions:

Based on the data we see here, it’s clear that the get() call also has algorithmic complexity of constant time. The trendiness displayed have slopes due only to noise in the data. In this example, it is difficult to tell whether linear probing or separate chaining is preferred.

Possible limitations:

Possible limitations with this procedure are largely mitigated by reading data into memory first and performing many repetitions. However, we are still measuring aggregate time to perform 10 gets, which could potentially muddy the results, though it doesn’t seem likely here.

**Q4:** Which hash implementation performs better on the following scenario:

- a mixture of a large number (but near equal) number of puts and gets.

Testing methodology:

In order to test this scenario, we create a symbol table by randomly interspersing 100,000 puts with 100,000 gets, recording the total time required between a linear probing implementation and a separate chaining implementation. We will perform each test 10 times and use the average time taken for comparison.

Data:

LinearProbingHashST: 0.7008 seconds

SeparateChainingHashST: 0.2125 seconds

Conclusions:

Based on this analysis, the SeparateChainingHashST implementation is clearly a superior choice, almost 4 times as fast.

Possible limitations:

In order to decide whether to put or get the next key, we call a standard library that generates random numbers. If this call required processing time that dwarfed the symbol table calls, the resulting times would be meaningless for answering our question. Also, for each repetition of the experiment, we process the keys in the same order, but randomly decide whether to get or put them. In a perfect world, even the order with which we visit these keys would be randomized.