EECS 560: Lab 4 Report

Due on Wednesday, February 27

Lei Wang R 2:30-4:20

Damian Vu

Organization of experimental profiling

My experiment was organized in a way such that I would reduce the size of "critical" sections in my code. This essentially means that all timing errors would be minimized, e.g., failed inserts do not count toward cumulative insert times.

Other than timing considerations, the organization of the profiling was done in such a way that testing my hash table would take the least amount of code possible. Then, looking at my code, each table does the same types of operations at the same time, while the timing of their operations is done separately.

Input data generated using the random number generator

We assume that the time taken to generate a random number is constant time. All of my calculations on this random number were also done in constant time (*rand() % 5m*) + 1, which gives a number between 1 and 5m inclusive. The only problem with this sort of calculation, is that generating our numbers in such a way is biased towards slightly smaller numbers. Since our find operations use numbers generated in the same way, this bias should be minimized when many tests are ran.

CPU time recording in C++

Since CPU time recording in C++ is done by calling the clock, running operations, then calling the clock again and subtracting the start time. This means that most operations in my hash tables need to be optimized and made sure that they only do one thing.

Data recording and analysis

I ran my profiling a total of ten times on the lab machines and then took the averages from the tables. Here are the final tables:

Open hashing:

Size	**0.1m**	**0.2m**	**0.3m**	**0.4m**	**0.5m**
Build	90ms	123.5ms	159.6ms	201.4ms	244ms
Found	0.1ms	0.15ms	0.23ms	0.33ms	0.42ms
Not Found	3.4ms	3.1ms	3.07ms	3.12ms	3.15ms

Closed hashing with quadratic probing:

Size	**0.1m**	**0.2m**	**0.3m**	**0.4m**	**0.5m**
Build	29.13ms	52ms	75.35ms	101.5ms	128.8ms
Found	0.05ms	0.09ms	0.14ms	0.19ms	0.25ms
Not Found	2.5ms	2.22ms	2.17ms	2.20ms	2.22ms

Closed hashing with double hashing:

Size	**0.1m**	**0.2m**	**0.3m**	**0.4m**	**0.5m**
Build	31.55ms	56.5ms	82.9ms	113.7ms	147.2ms
Found	0.05ms	0.10ms	0.14ms	0.20ms	0.25ms
Not Found	2.65ms	2.44ms	2.44ms	2.53ms	2.61ms

Data recording and analysis continued on next page…

Analysis: The build times of the quad and double hashing tables were always very similar, but the open hashing table build time was always longer than the two closed hashing tables. This is probably due to open hashing requiring more memory to be allocated in total, since the open hashing table needs to build long chains of nodes containing information.

In every one of the tables, the cumulative find time was always very small. After running some tests on my code, I found that when searching for $0.01m$ random entries (about 10,000), only about 1,000 were found, and about 9,000 did not exist in the hash table. This was the same case no matter the size of the table initially. This essentially means that my implementation of each tables is able to find existing entries in constant time. The same goes for the not found.

Performance comparison, observations, and summary

Between the three tables, the two closed hashing tables always performed better than the open hashing table. This is to be expected, as we never went over a load factor of $\frac{1}{2}$. Overall, the quadratic probing performed better than the double hashing at all stages. This is probably due to the fact that our collision resolution for double hashing takes a few more operations than our simple quadratic probing does.

Overall, we can see that each table increased in build time by a very similar amount for each $0.1m$ increase in table size. This would imply that our insert function is $O(1)$. This is to be expected, as that is how our hash tables should be theoretically.

Since the found and not found times do not differ much for their respective tables, we can also assume that our find takes $O(1)$ time. Our not found also will take $O(1)$ time, but will be longer on average due to the not found criteria being $O(k) = O(1)$ for the closed hashing tables. For this lab we had $k = 10$.

Experimentally, we were able to confirm that our hash table performs how it theoretically should perform. This is due to cleverly identifying critical areas of the code when recording times. Minimizing these sections of code will result in the least amount of error generated. Some of the methods I used include throwing out time elapsed for a failed insertion, and also the time used to generate a random number was never taken into account.

Conclusions

Overall, this lab was fun to do, and gave great instant feedback on whether my design of my tables in C++ was sufficient to represent a theoretical hash table. It required a lot of thinking about minimizing the critical sections of code so that our timing may be as accurate as possible.

Overall, if we are dealing with a large input, we should use a closed hash table with quadratic probing and rehash when the load factor approaches $\frac{1}{2}$.