EECS 560: Lab 4 Report

Hash Tables Experimental Profiling

Emilia Paz Ojeda

KUID: 2889275

Date submitted: 02/27/2019

1. Organization of experimental profiling.

The experimental profiling analyzes the build and find value time for hash table with the following implementations:

a. Hash table with open chaining
b. Hash table with closed chaining with quadratic probing
c. Hash table with closed chaining with double hashing

The experimental profiling follows the next steps:

- Each table was built with 0.1m random unique numbers from 1 to 5m. The building time only takes into account the successful insertions. If a random number is a duplicate, time is reset and a new random number is generated until it is unique.
- Then, new 0.01m random numbers from 1 to 5m are generated and find in each of the hash tables. The time it takes will be added to time found or time not found, depending on the outcome. m is set to 1000003 for our testing.
- In order to get a closer approximation, we repeat this approach 5 times and take the average of the times.
- Lastly, we repeat the same procedure but by inserting 0.2m, 0.3m, 0.4m and 0.5m, each also by generating 5 random set of numbers and taking the average. This let us look how the implementation changes when the number of keys increase.

2. Input data generated using the random number generator.

The input data is generated using the random number generator, which follows the next steps:

- First, to prevent sequence repetition between runs, we place `srand(time(NULL))` at the beginning of the program. This initializes the random seed in order for every time we run the program it will generate new random numbers.
- Then `rand()`returns a random positive integer in the range from 0 to 32,767. Because we need a specific range, we use `rand() % l + f`, where l is the last number and f is the first number in the range.
- This functions use the libraries `stdlib.h` and `time.h`

3. CPU time recording in C++.

Time in C++ can be recorded using the library `time.h`, under the following steps:

- `clock()` returns the processor time consumed by the program. It returns a type `clock_t` value that is the number of clock ticks elapsed since the last execution. `clock_t` is a type defined in `<ctime>`
- Therefore, in order to collect the time it takes for a specific of the program to run, we need to start the clock just before it, and then call it again. Where we call it again we subtract the old clock time, thus giving the difference which is the time we want. It follows this: `t = clock() - t;`
- In this implementation, the time is recorded for each procedure in each hash table. This allows to recollect all the data to analyze it.

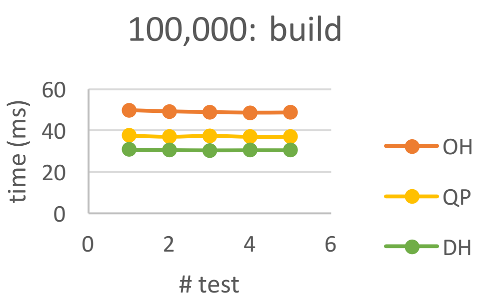
4. Data recording and analysis.

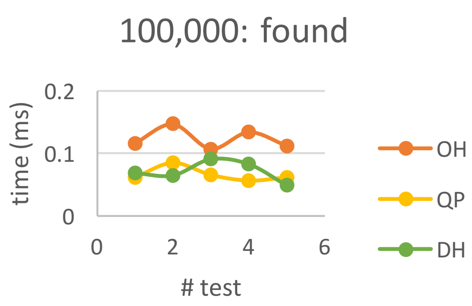
The data is recorded in the following way:

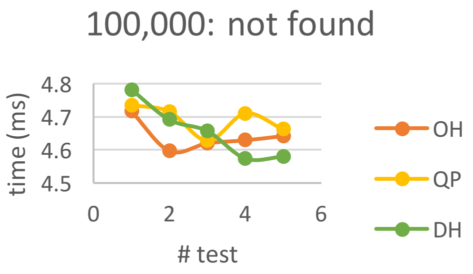
- Every time a value is inserted in a hash table, the time it takes is added to the corresponding time variable `(t_o, t_qp or t_dh)`. Time is not added for duplicate numbers. If a duplicate number is found, the function will loop until finding a unique one and the it records the time.
- Every time a value is searched in a hash table, the time it takes to be found is added to the corresponded time variable according to the hash table and its success `(t_o_find_s, t_o_find_u, t_qp_find_s, t_qp_find_u, t_dh_find_s, t_dh_find_u)`
- After inserting and searching for one test, the times are added to their respective grids. First each time is divided by times per second to get the value in seconds, and multiplied by 1000 to change it to miliseconds. Then it is sum to the grid at their respective position. It adds every iteration, that way it takes into account the 5 repetititons.
- The same data procedure is repeated for the 5 repetitions for each of the 5 number sets. After 1 number set (0.1m,0.2m,..) is finished, the times in the grid are divided by 5 in order to get the average. Until this point the grid was holding the total time of the 5 iterations.

The following charts and graphs show the recorded times in ms. for each of the tests. They are divided by the 5 random numbers generated for the hash tables. Each table shows the time it took for build, found and not found for each table in each of the 5 trials. Then, the graphs show the time for build, found and not found. There we can see that the time are similar, but not always. This is good, because it is the reason why we take 5 tests and take the average, because of that margin of error. Also, we see that it is consistent which of the hash tables is faster for each of the functions.

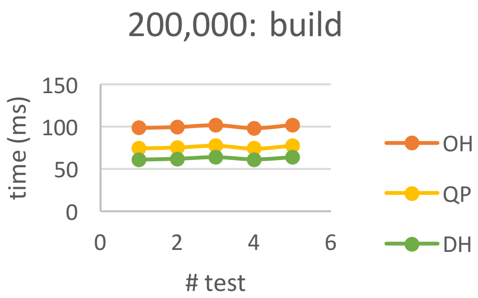
			#1	#2	#3	#4	#5	Average
100000	build	OH	49.786	49.17	48.813	48.496	48.593	*48.9716*
		QP	37.495	37.066	37.44	37.036	37.06	*37.2194*
		DH	30.685	30.44	30.286	30.351	30.344	*30.4212*
	found	OH	0.116	0.147	0.107	0.134	0.112	*0.1232*
		QP	0.062	0.085	0.066	0.057	0.062	*0.0664*
		DH	0.069	0.065	0.091	0.083	0.049	*0.0714*
	not found	OH	4.717	4.596	4.62	4.629	4.642	*4.6408*
		QP	4.735	4.715	4.628	4.71	4.662	*4.69*
		DH	4.782	4.691	4.657	4.574	4.58	*4.6568*

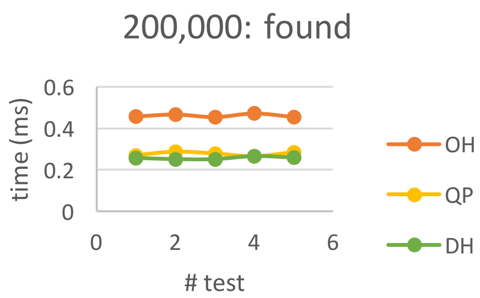


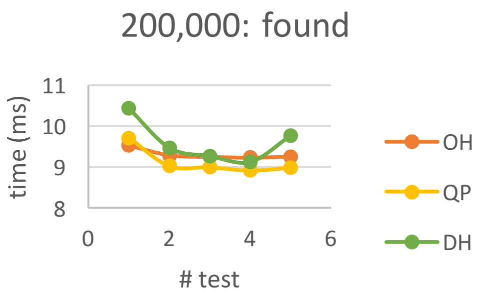




			#1	#2	#3	#4	#5	Average
20000	build	OH	98.49	99.608	101.9	98.187	102.166	*100.0702*
		QP	74.584	75.552	77.633	74.172	77.731	*75.9344*
		DH	60.868	61.957	63.853	61.178	63.776	*62.3264*
	found	OH	0.457	0.467	0.454	0.472	0.455	*0.461*
		QP	0.27	0.287	0.278	0.265	0.284	*0.2768*
		DH	0.257	0.251	0.251	0.266	0.259	*0.2568*
	not found	OH	9.532	9.288	9.247	9.23	9.252	*9.3098*
		QP	9.709	9.027	8.999	8.921	8.988	*9.1288*
		DH	10.434	9.463	9.26	9.12	9.773	*9.61*

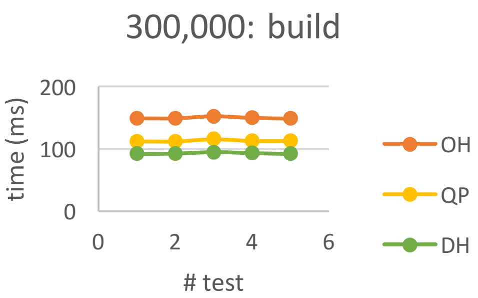


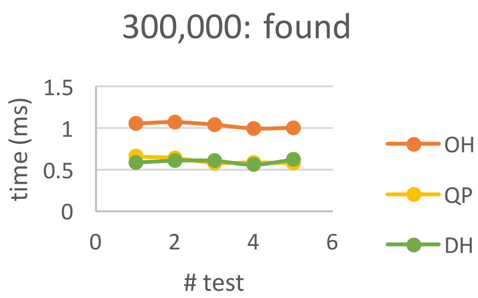


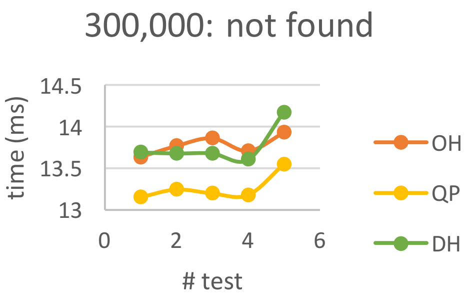


			#1	#2	#3	#4	#5	Average
30000	build	OH	149.182	149.22	152.457	149.805	148.79	*149.8908*
		QP	112.323	112.291	115.887	112.949	112.931	*113.2762*
		DH	91.942	92.452	94.577	93.075	91.861	*92.7814*
	found	OH	1.055	1.072	1.041	0.996	1.003	*1.0334*
		QP	0.658	0.639	0.579	0.585	0.576	*0.6074*

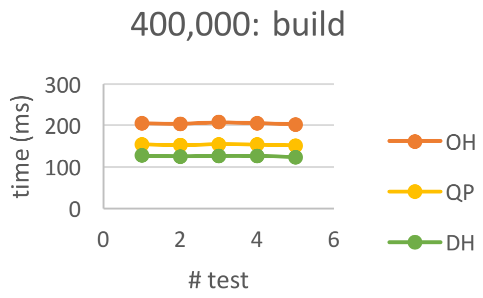
		DH	0.588	0.61	0.609	0.563	0.621	*0.5982*
	not found	OH	13.633	13.766	13.86	13.705	13.928	*13.7784*
		QP	13.15	13.243	13.196	13.176	13.548	*13.2626*
		DH	13.691	13.678	13.679	13.603	14.174	*13.765*

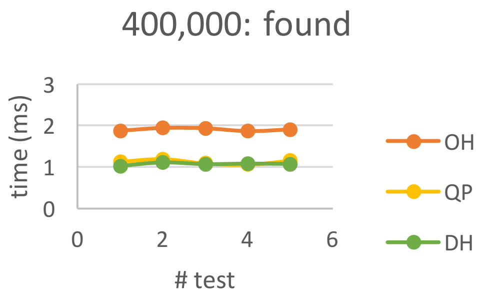


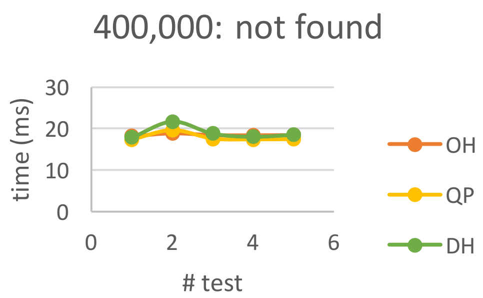




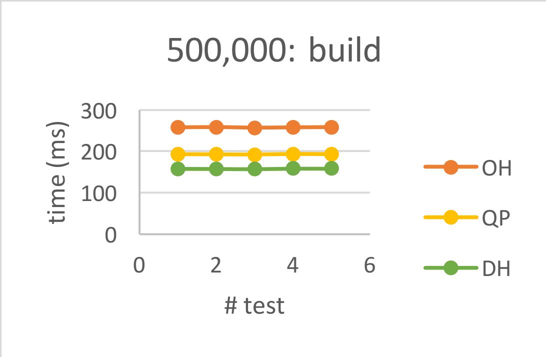
			#1	**#2**	**#3**	**#4**	**#5**	**Average**
40000	**build**	**OH**	205.082	204.012	207.634	205.581	202.298	*204.9214*
		QP	154.654	152.775	154.886	153.923	151.858	*153.6192*
		DH	127.45	125.6	127.054	126.575	123.723	*126.0804*
	found	**OH**	1.874	1.939	1.932	1.866	1.904	*1.903*
		QP	1.13	1.19	1.088	1.059	1.153	*1.124*
		DH	1.021	1.104	1.064	1.076	1.07	*1.067*
	not found	**OH**	18.132	18.774	18.374	18.354	18.394	*18.4056*
		QP	17.274	19.58	17.488	17.315	17.371	*17.8056*
		DH	17.871	21.651	18.775	18.061	18.546	*18.9808*

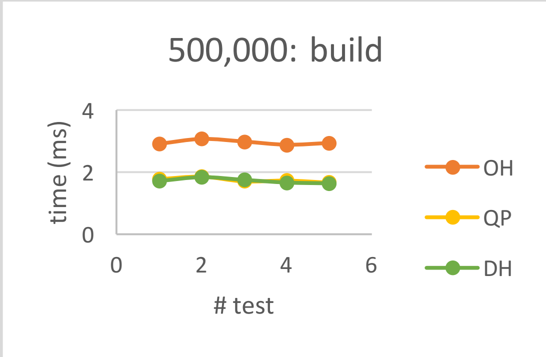


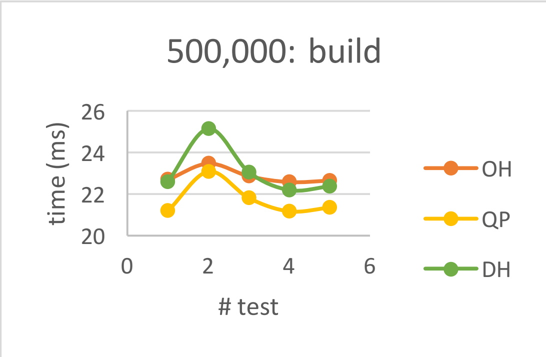




			#1	#2	#3	#4	#5	Average
50000	**build**	**OH**	257.539	258.129	256.908	257.608	258.056	*257.648*
		QP	192.704	192.323	191.96	193.218	192.682	*192.5774*
		DH	158.083	157.763	157.354	158.494	158.236	*157.986*
	found	**OH**	2.898	3.057	2.972	2.87	2.926	*2.9446*
		QP	1.778	1.858	1.706	1.732	1.664	*1.7476*
		DH	1.702	1.827	1.74	1.654	1.628	*1.7102*
	not found	**OH**	22.695	23.478	22.864	22.58	22.652	*22.8538*
		QP	21.193	23.069	21.8	21.17	21.352	*21.7168*
		DH	22.588	25.151	23.038	22.172	22.358	*23.0614*







5- Performance comparison, observations, and summary. (Experimentally determine the complexity of functions for each hash table, and compare them to their theoretical complexities. If they are different, you need to explain why.)

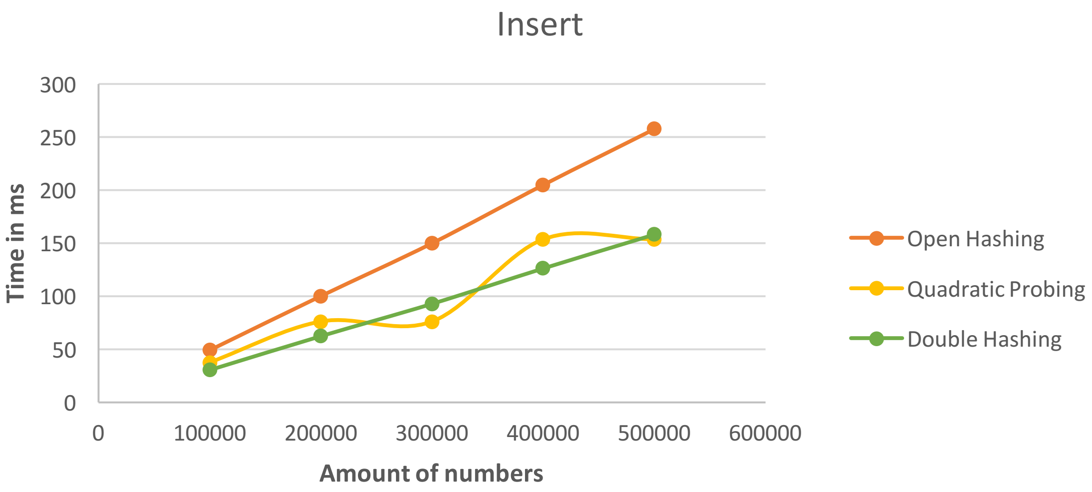
The final chart gives us the average times in ms:

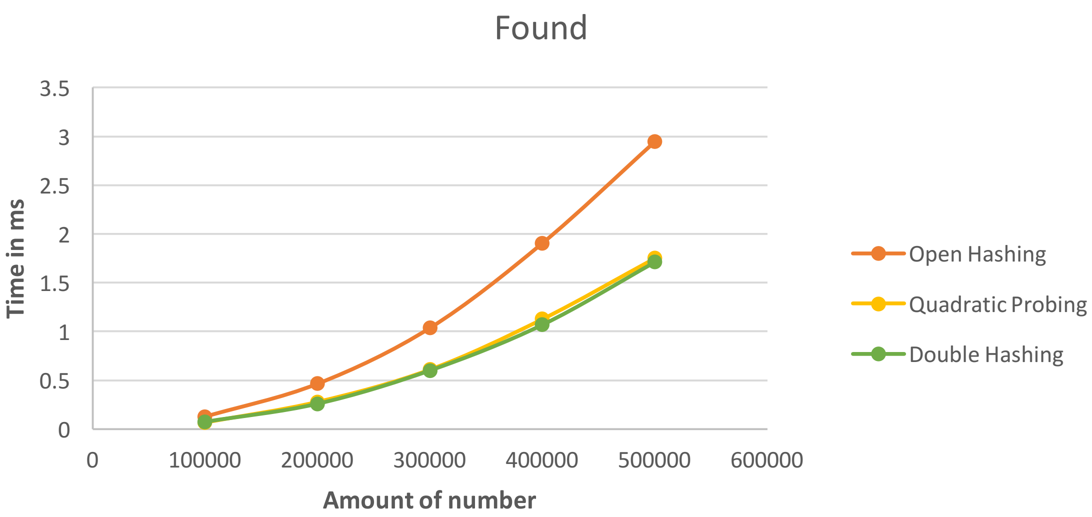
		100000	**200000**	**300000**	**400000**	**500000**
OH	**build**	48.9716	100.0702	149.8908	204.9214	257.648
	found	0.1232	0.461	1.0334	1.903	2.9446
	not found	4.6408	9.3098	13.7784	18.4056	22.8538

		100000	**200000**	**300000**	**400000**	**500000**
QP	**build**	37.2194	75.9344	75.9344	153.6192	153.6192
	found	0.0664	0.2768	0.6074	1.124	1.7476
	not found	4.69	9.1288	13.2626	17.8056	21.7168

		100000	**200000**	**300000**	**400000**	**500000**
DH	**build**	30.4212	62.3264	92.7814	126.0804	157.986
	found	0.0714	0.2568	0.5982	1.067	1.067
	not found	4.6568	9.61	13.765	18.9808	23.0614

From these tables, we can build charts for build, found and not found that allow us to look better at the results:


Insert
300
250
200
150
100
50
0
Time in ms
0
100000
200000
300000
400000
500000
600000
Amount of numbers
Open Hashing
Quadratic Probing
Double Hashing


Found
3.5
3
2.5
2
1.5
1
0.5
0
Time in ms
0
100000
200000
300000
400000
500000
600000
Amount of number
Open Hashing
Quadratic Probing
Double Hashing


Not Found
25
20
15
10
5
0
Time in ms
0
100000
200000
300000
400000
500000
600000
Amount of numbers
Open Hashing
Quadratic Probing
Double Hashing

By looking at the numbers and the charts, we can see that:

a. Insert
   - The insert is slower for open hashing and increases linearly. Insertion in open hashing is done by hashing the key O(1), and then looping through the linked list in that slot. Therefore, worst case can be of O(n). The theoretical complexity average is O(1) and the worst is O(n), which matches our results.
   - The quadratic probing increase is not consistent and looks exponential, but still faster than open hashing. Quadratic probing insertion is done by hashing the key O(1) and if it is not successful it will loop a max of k times until finding the right place to insert O(k). The theoretical complexity average is O(1) and the worst is O(k), which doesn't completely match our results since we got an up and down curve.
   - For double hashing it also increases linearly but a much slower rate. Double hashing it will loop a max of k times until finding the right place to insert O(k). The reason why it is faster than quadratic probing is that it applies a second hashing that tends to have less collisions. The theoretical complexity average is O(1) and the worst is O(n), which matches our results.

b. Found
   - Open hashing takes the longest time out of the three hast tables to find a value. In order to find a values it hashes the key O(1) and if it is not already found, it searches for it on the linked list O(n). The theoretical complexity if O(1)(n/m) = O(n), which doesn't match our exponential growth. In the graph we see an exponential growth, which means we must have various chains and therefore it has to loop through the linked list.
   - Quadratic probing is faster than hashing because it has a limit amount of places where it can search. It hashes the key O(1) and if it is not already found it looks at the other possible places where it could have been placed, which is O(k). In the graph we see it slightly exponential, which we could infer the find is taking various steps until it find the key. The theoretical complexity is O(n), which matches our graph.
   - Double hashing is really similar to quadratic probing because it hashes the key O(1) and if it is not already found it looks at the other possible places where it could have been placed, which is O(k). It is slightly faster because the probability to find the key is higher since they are more spread out. In the graph we see it slightly exponential, since we are generating random numbers in a big range it takes longer time to find them. The theoretical complexity is O(n), which matches our graph.

c. Not found
   - Open hashing follows the same procedure as found, but for the worst case since the key is not found. This means it hashes the key O(1) and searches through the linked list to the end O(n). The graph shows a linear increase. The theoretical complexity is O(k), which matches our graph.
   - Quadratic probing follows the same procedure as found, but for the worst case since the key is not found. This means it hashes the key O(1) and tries to look for it k more times O(k), which k is the limit of the times it can search for a key. The graph shows a linear increase. The theoretical complexity is O(k), which matches our graph.
   - Double hashing follows the same procedure as found, but for the worst case since the key is not found. This means it hashes the key O(1) and tries to look for it k more times O(k), which k is the limit of the times it can search for a key. The graph shows a linear increase. The theoretical complexity is O(k), which matches our graph.

6. Conclusions

- Open chaining is slower because it needs to follow pointers to data.
- Open chaining deals easier with collisions by inserting the key to the linked list.
- Closed chaining insertion is faster than open chaining. However; insertion may fail even though the table is not empty.
- Open chaining search is problematic to implement, but the run time is faster than closed chaining.
- Double hashing is the most efficient when clustering is avoided
- Quadratic probing is the most efficient when the keys to be stores are not greater than half of the table.
- Quadratic probing and double chaining is usually faster than open chaining when the load factor is low because you don't have to follow pointers between list nodes. However, it will get very slow as the load factor gets closer to the number of buckets, because finding that empty bucket will take longer and longer.
- Note that in these implementations we have not rehashed the values, which could decrease the amount of time for future insert or find.