Programming Assignment #2

Brendan Foley

12 April 2023

1 Description

In this project, we analyzed the performance of various hashing schemes and divisors. We have defined 11 cases for testing our implementations of hash functions, division and a user defined one, for which I chose multiplication. The cases are defined in Table 1.
My code runs through two main methods: hash and handle_collision. There are several other functions that I defined to aid in the hashing, formatting output, and reading/writing files, but those are auxiliary to the main effort.
Hash takes in the input keys as a list, along with other variables that define the case (modulus, bucket size, etc.), and returns the hash table along with other statistical tracking variables for post-analysis. For each key we have, it sees if the location it is hashed to is occupied, if it is then it calls handle_collision, which returns the index we should use (calculated based on the user-specified collision_scheme - linear, quadratic, or chaining). If the key cannot be inserted at all, then it is added to a list to track unadded keys.

2 Performance

My algorithm, while is not perfectly efficient, has relative time and space efficiency. The average time complexity of storing items in a hash table is O(1), while the storage complexity is O(n). Likewise, the average lookup and deletion efficiency of a hash table is O(1). This can change depending on the collision resolution you use, as using chaining can, in worst case, result in a single slot in the hash table being a linked list, which gives O(n) lookup and deletion complexities. Figure 1 and Figure 2 show the trends of number of comparisons against the number of keys to insert (Figure 2 eliminates the worst performing cases in order to show smaller differences between trends that may not be appreciable from the zoomed out view of Figure 1).
In my testing, I included varying numbers of keys, along with a couple of edge cases. The number of keys used for insertion were: 35, 36, 60, 84, 108, 120, and 121. The edge cases I tested were: a key of 0s, all matching keys, and the case where we have more keys than slots.

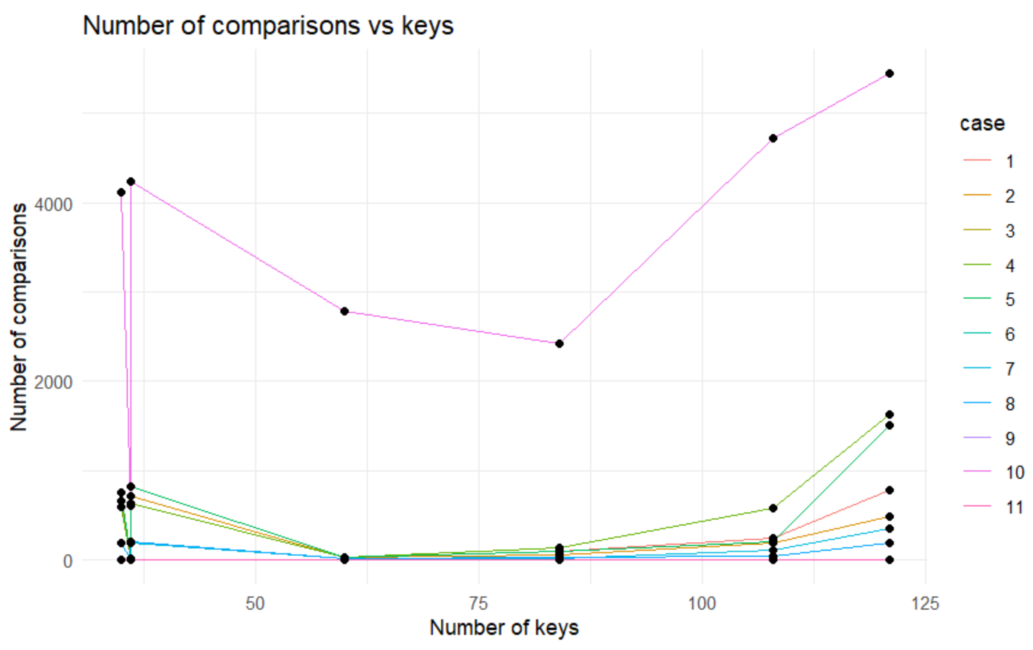

Figure 1: The number of comparisons required for datasets containing x keys.

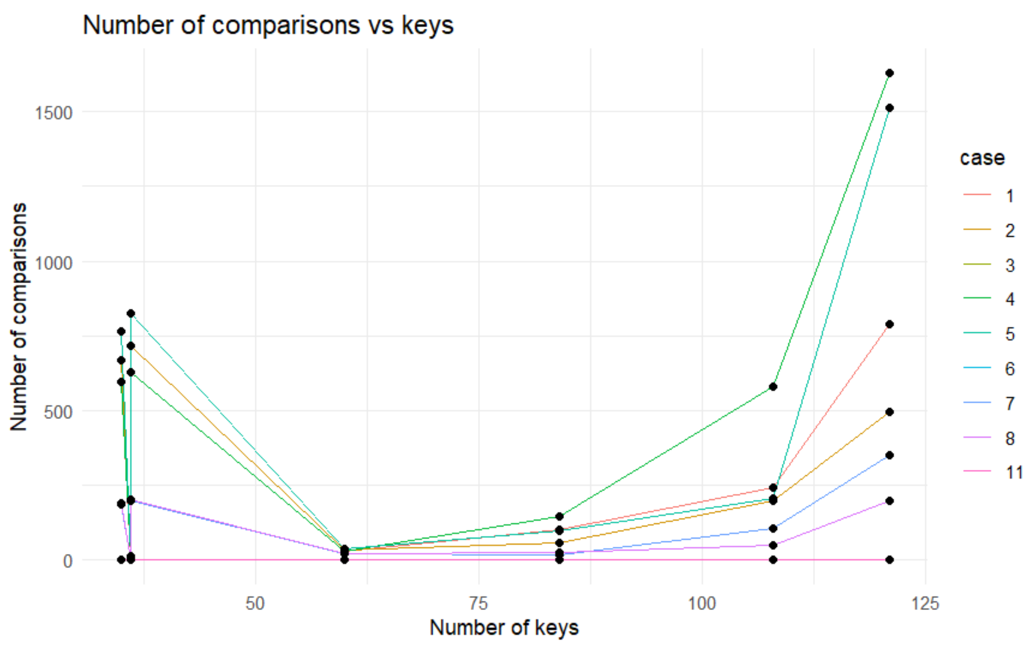

Figure 2: The number of comparisons required for datasets containing x keys, without the worst performing cases.

I also created a figure to visualize the number of keys not inserted in each case as the number of input keys grows, shown in Figure 3.

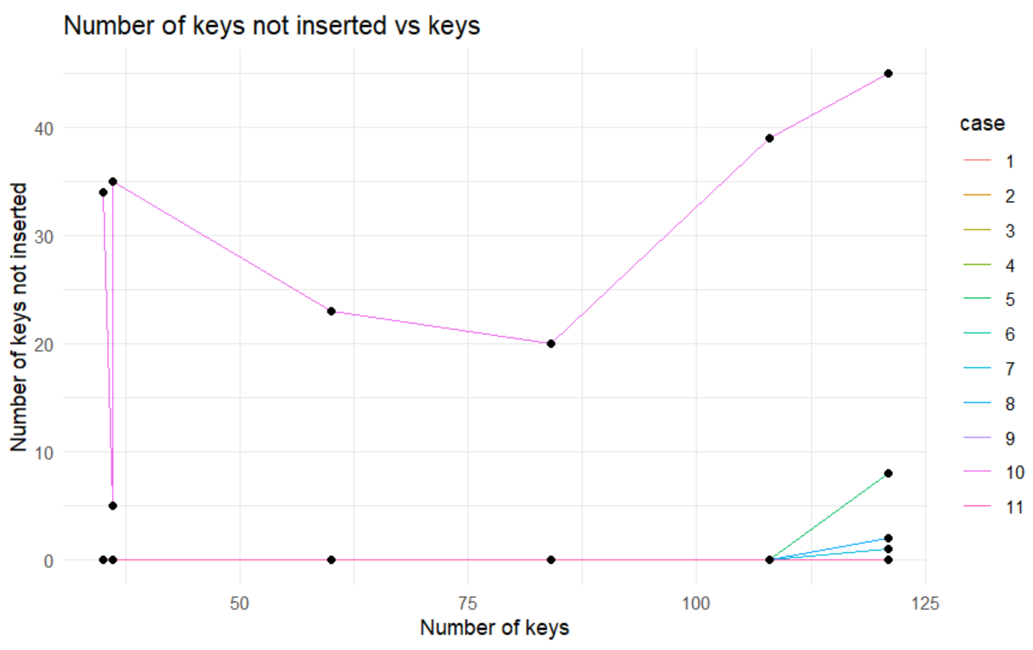

Figure 3: The number of keys not inserted when inserting x keys, by case.

primary_collisions	**secondary_collisions**	**num_comparisons**	**num_not_inserted**	**case**	**num_keys**
4	1	5	0	1	36
4	3	7	0	2	36
0	0	0	0	3	36
9	1	11	0	4	36
9	4	14	0	5	36
0	0	0	0	6	36
1	0	1	0	7	36
1	0	1	0	8	36
5	5	605	5	9	36
5	5	605	5	10	36
0	0	0	0	11	36
35	34	630	0	1	36
35	34	715	0	2	36
0	0	0	0	3	36
35	34	630	0	4	36
35	34	826	0	5	36
0	0	0	0	6	36
33	30	198	0	7	36
33	30	204	0	8	36
35	35	4235	35	9	36
35	35	4235	35	10	36
0	0	0	0	11	36
34	33	595	0	1	35
34	33	668	0	2	35
0	0	0	0	3	35
34	33	595	0	4	35
34	33	763	0	5	35
0	0	0	0	6	35

Continued on next page

Table 1 – continued from previous page

primary_collisions	secondary_collisions	num_comparisons	num_not_inserted	case	num_keys
32	29	187	0	7	35
32	29	192	0	8	35
34	34	4114	34	9	35
34	34	4114	34	10	35
0	0	0	0	11	35
34	18	102	0	1	84
30	15	58	0	2	84
0	0	0	0	3	84
30	17	144	0	4	84
29	20	96	0	5	84
0	0	0	0	6	84
12	4	19	0	7	84
12	6	25	0	8	84
20	20	2420	20	9	84
20	20	2420	20	10	84
0	0	0	0	11	84
47	32	242	0	1	108
54	37	198	0	2	108
0	0	0	0	3	108
53	37	581	0	4	108
47	36	207	0	5	108
0	0	0	0	6	108
28	19	107	0	7	108
25	11	49	0	8	108
39	39	4719	39	9	108
39	39	4719	39	10	108
0	0	0	0	11	108
51	38	789	1	1	121
52	40	494	2	2	121
0	0	0	0	3	121
64	50	1629	8	4	121
64	46	1514	8	5	121
0	0	0	0	6	121
32	23	349	1	7	121
29	20	198	2	8	121
45	45	5445	45	9	121
45	45	5445	45	10	121
0	0	0	0	11	121
19	6	30	0	1	60
20	8	33	0	2	60
0	0	0	0	3	60
17	5	30	0	4	60
19	5	36	0	5	60
0	0	0	0	6	60
16	4	21	0	7	60
15	4	23	0	8	60
23	23	2783	23	9	60

Continued on next page

Table 1 – continued from previous page

primary_collisions	secondary_collisions	num_comparisons	num_not_inserted	case	num_keys
23	23	2783	23	10	60
0	0	0	0	11	60

3 Analysis

The performance of the hashing algorithms appears to be in sync for the lower number of keys (with the exception of case 10), however, over time the asymptotic differences begin to get teased out. For the moment, we will eliminate case 10 from our analysis and will revisit that case below. By choosing a different hashing algorithm, we begin to see an increasing number of comparisons around the 100 key mark, as well as the number of keys not being inserted. In real life settings, we will often be dealing with thousands, if not millions, of keys to deal with. If we are beginning to see differences in the number of comparisons at 100 keys, then these asymptotic differences in efficiency will be massive later on. Granted, the number of keys not inserted is also related to the size of the hash table.
When choosing a scheme, it is important to not only consider asymptotic run time, but the collision resolution. Linear collision resolution will lead to clustering of keys that share common hashes, so in a system that continuously spits out keys that hash to the same value we may see a drop in performance. This is where quadratic hashing can play a role and break up the clustering seen with linear resolution, however, this can lead to not every location in the table being visited and an increased number of of keys being rejected. This can be seen in Figure 3, with case 10 and case 5. Lastly, chaining can be beneficial as each key will be stored, however, it will lead to longer lookup times as you may have to traverse a list to find the item you need.
The load factor is calculated by: $\frac{\texttt{\# of items in table}}{\texttt{\# number of slots in table}}$. This can grow to be larger than 1 in a table that uses chaining to resolve collisions, as each slot can be made up of a linked list with several elements in it. Also, as the load factor grows, retrieval becomes slower, and insertion may become slower as well due to the increased risk of collision.
When we delete a key from the table, we must be aware of how the lookup process works. If we hash a key to its ”normal spot”, but that spot is taken up we proceed to collision resolution. In this case, let us assume it is linear. We then insert the key at the next available spot. If we delete this key and do not insert a ”tombstone”, then when we go to lookup any keys that may be after that that hashed to the same original place, we will prematurely abort the lookup process. Thus, when we delete a key, we need to insert a marker to represent a deleted, but not empty, slot.

4 Application

Hash tables have applications in any computer-related field, especially bioinformatics, which typically requires large, in-memory, storage structures with fast lookup and insertion time. This is where hash tables excel. The 2013 paper by Rizk et al. is an amazing example of this, as they use hash tables to keep k-mers of DNA/RNA sequences in memory and count occurances. Hash tables also have applications in sequence alignment, genome and transcriptome assembly, and error correction.

5 Improvements

The only improvement I made was allowing for various bucket sizes.
The largest problem I encountered was in defining my own hash. I arbitrarily chose multiplication hash. I also arbitrarily chose the fraction multiple to use. Specifically with that, I wish to understand further how to select and optimal A value in my hash calculation. I also wish to further understand how the quadratic collision resolution degrades the multiplication hashing strategy, as that has a large discrepancy between any other case, including ones using the same hashing algorithm.

6 Appendix

6.1 References

G. Rizk, D. Lavenier, R. Chikhi, DSK: k-mer counting with very low memory usage, Bioinformatics (2013)