605.420 Foundations of Algorithms

Anam Ahsan

Project 2 Analysis

Due Date: November 8th, 2022

Dated Turned In: November 8th, 2022

Description and justification of your data structures, implementation choices, and design decisions

The main data structure used throughout is an array of varying bucket size as per the assignment requirements. Another data structure implemented was a stack, this was used to keep track of free space available in an array in one of the hashing schemes without iterating through the array.

In order to clearly parse input file into correct hash table, separate hash tables were created within the main program. Although, this could have been a method or a separate class as well. Likewise, in the main are three sub-sections that run through a given hashing scheme, depending on the user input of modulo and bucket size. These all occur in the 'main' and calls are made to the following methods that are outside of 'main': 'LinearProbe', 'QuadraticProbe', 'deleteStackspot', 'printList', 'printBList', and 'printClist'.

The 'LinearProbe' method produces and return a new index using the linear probe equation. It also checks whether the index is within the array range. The 'QuadraticProbe' method produces and returns a new index using the quadratic probe equation. It also checks whether the index is within the array range. The 'deleteStackspot' method, deletes used space in the array from the stack, so that a used space is not erroneously given during a collision. Finally, 'printList', 'printBList', and 'printClist' print the arrays according to bucket size. The 'printList' method prints bucket size 1 arrays that only had linear and quadratic probing. The 'printBList' method prints bucket size 3 arrays. Lastly, the 'printClist' method prints the arrays that have open address chaining. These separate methods were made to accommodate the unique requirements of each array and scheme.

Efficiency with respect to both time and space

Initially, when looking at the table and graph, it seems easy to assume that chaining is the most efficient form of collision handling, and that division with modulo 113 is the best hashing scheme. This is because across all collisional handling schemes, chaining regardless of the hashing scheme has the least collisions. Likewise, in terms of the hashing schemes, division with modulo 113 yielded the lowest number of collisions, this is likely due to it being a prime number which allows for more uniformity. In contrast, division with modulo 41 easily yielded the most collisions, this is a prime number also, but the bucket size used with his hashing scheme negated its advantage.

Chaining, with open addressing has less collisions to deal with due to the stack that is implemented with it that assures free space is available. This aspect significantly reduces collisions from occurring as one is not iterating through the array to find free space, as is the case with linear and quadratic probing. Another positive aspect of chaining is that the array does not have to be significantly larger than the input due to the chaining scheme which only provides space that is open. The downside of chaining is that maintaining the stack while filling up the array is expensive in terms of time.

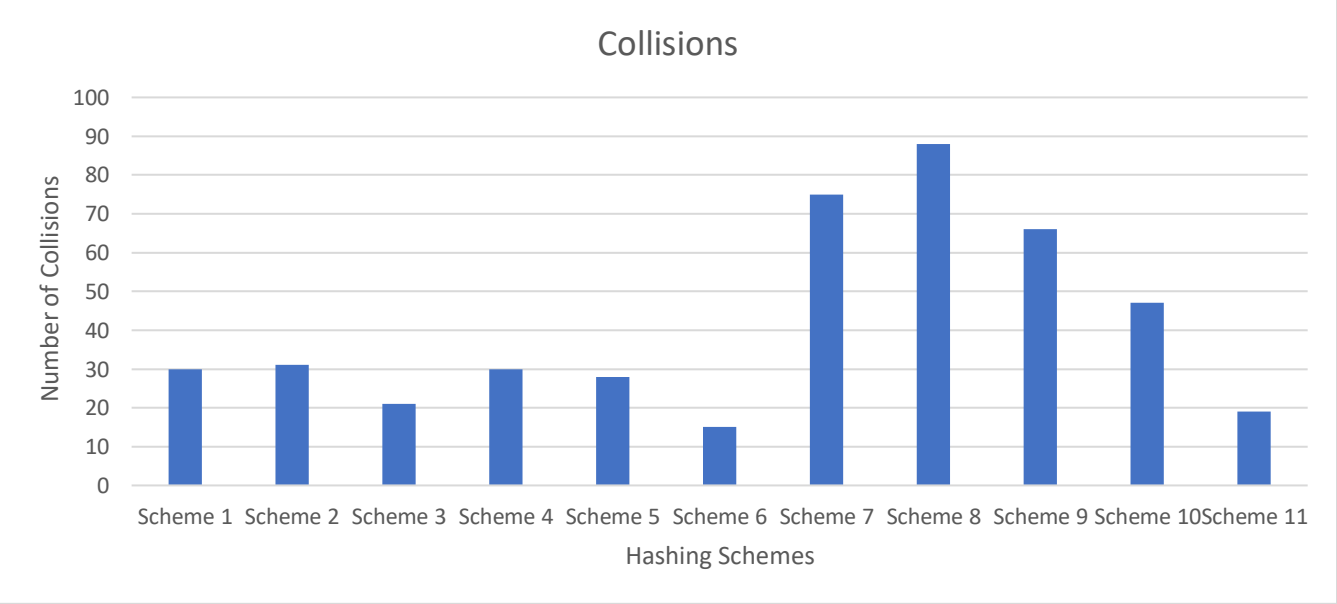
In comparison, linear and quadratic probing significantly rely on the array and bucket size. Interestingly, the difference between linear probing and quadratic probing is not as dramatic in terms of collisions, though it could be argued that quadratic probing is slightly better than linear probing given the hashing scheme. In example, with the multiplication hashing scheme quadratic probing performs significantly better than linear probing in terms of collisions. The biggest ramification of using linear and quadratic

probing is that as the table becomes more full clustering is more likely which in turn impacts the load factor. To address this short-coming if we could increase the array size while taking into consideration the load factor. This way we can avoid clustering and we could avoid more collisions. The important aspect to consider with load factors is that although more space is helpful, we must remain cognizant of the lookup time. This is especially apparent with the bucket size three array, because despite having the same number of slots it had a much more expensive lookup.

Deleting an item from a hashing scheme has large ramifications. In the chaining scheme, it would require updating the pointers, and repopulating the stack that tracks free space. In linear and quadratic probing, it would require marking the deleted space so that they are not overwritten (tombstone deletion) or shifting elements backwards to cover the new gaps (robin hood hashing). This would be quite expensive to accommodate, but again the least difficulty in deleting an item is in the chaining scheme as only the select pointers will need to be changed, rather than the entire array.

A table and a graph summarizing your observations with respect to asymptotic efficiency.

Hashing	Hashing Scheme	Collision Scheme	Bucket Size	Collisions	Comparisons
Scheme 1	**Division modulo 120**	**Linear Probing**	**1**	**30**	**156**
Scheme 2	**Division modulo 120**	**Quadratic Probing**	**1**	**31**	**138**
Scheme 3	**Division modulo 120**	**Chaining**	**1**	**21**	**81**
Scheme 4	Division modulo 113	Linear Probing	1	30	149
Scheme 5	Division modulo 113	Quadratic Probing	1	28	134
Scheme 6	Division modulo 113	Chaining	1	15	75
Scheme 7	**Division modulo 41**	**Linear Probing**	**3**	**75**	**117**
Scheme 8	**Division modulo 41**	Quadratic Probing	**3**	**88**	**125**
Scheme 9	Multiplication	Linear Probing	1	66	156
Scheme 10	Multiplication	Quadratic Probing	1	47	138
Scheme 11	Multiplication	Chaining	1	19	79



What you learned / What you might do differently next time

During this project I learned a lot about efficient storage of data. I was always aware of needing a good hashing scheme to come up with indexes for storing values, but this project really helped highlight the importance of space and collision schemes. In terms of space, my initial thought was that more buckets would make things easier, and result in less collisions, but in practice I realized that was not the case. Although it allowed data to be more uniformly spread, it was more expensive in terms of collisions. Every time we reached an occupied index, we then had to iterate through the buckets to make sure there was indeed no more space which was complex. This really put into perspective that the load factor is not necessarily indicative of performance. Likewise, with collision schemes, I did not expect there to be as big of a difference between chaining versus probing. Chaining is quite efficient, and it was interesting to see it implemented as open addressing, I do think it could have been made more efficient with a different implementation.

There are a number of things I would do differently with this program. The main design changes would be to have a separate method to create the necessary hash tables. This would not only have saved a lot of space in my main method, but it would entirely change the conditions I had to have in place to accommodate the bucket sizes of the various arrays. In modifying these two things, the way that my output is printed would be a lot more efficient. As it is now, it requires to be run 5 times to get all the schemes, and two of runs yield redundant information. Also, this time I implemented the chaining method by using a 2D array, where the pointers were the indices themselves that were stored in the array. Next time, I would create a class of nodes that I put in the array that hold the key value. The nodes would hold pointers to the next chained key rather than the array.

Applicability to Bioinformatics

This is especially useful and relevant to bioinformatics because what immediately comes to mind is sequencing unknown DNA or aligning genomes. When working with unknown DNA it is cut up into very small pieces and sequenced to then reassemble with a reference for gene alignment. In this process you can have thousands of unique pieces of DNA fragments that needs to be stored efficiently, but separately. This is very similar to this project where these ‘fragments’ of numbers have to be stored efficiently into the array. Chaining is especially relevant because you would want DNA ‘fragments’ that perhaps fall into a similar category to be chained together. Using these scheme, one can find a homologous gene. In a more general sense hashing is becoming more important in the field of bioinformatics as more data is available for us to store. Efficient storage of this data, along with search and retrieval is essential.