605.620: Algorithms for Bioinformatics

Thomas Muamar

Project 2: Hashing

Due Date: November 9, 2021

Dated Turned In: November 9, 2021

Project 1:

Justification for data structures/reasons for choices:

Within my program I decided to separate each of functions into there own classes. So, what is located within the folder is 6 different classes. To begin with we have the hashTable class which initializes the hash table into an array to allow values to be stored within. In this class also contains the free space that is required for the chaining method. Lastly there is a method to print the tables in the right format for either bucket size 1 or 3. The way this is done is when the value of printR is set to 4 the bucket size would be of size 1 making it print 5 in one row. Basically buckets 1-5 would be in one row and next row would be buckets 6-10. Then any other value assigned would give that one slot in the table a bucket size of 3 meaning it would have 3 slots for when it would check for empty spots. So, the overall reasoning for this was to create the hash table and be able to display the values as indicated in the instructions.

Then there is the next class which is the hash entry. This class was made to categorize the buckets and create the nodes for the linked list for chaining. This method would check whether the buckets were full or not. Class was basically to initializes the variables and a bunch of setters and getters for the collisions.

Separating the different collision methods allowed for a much easier time in correcting errors and not having a clutter in one huge class. In linear probing it had to different methods. The reasoning for this is we needed to have linear probing produce one for a bucket size 1 and another for bucket size 3. The bucket size 3 needed an extra step in there that would check all three buckets to see whether they would be full or not. The reason I had an else statement that would add on to the number of collisions is because I would need to count the secondary collisions that would occur. Then for quadratic probing does the same thing with collisions but the way I set up the occurrence was when the collision happened it would end up calculating a new hash function. This hash function is produced for the amount of times

the collision occurs. So basically, every time a collision occurs it would calculate a new hash function and that function gets larger the more the collisions occur.

With chaining this was done using the free space array that would be looked into for a free spot when a collision occurs. If the collision would occur repeatedly then the add function in the hash entry class would be called which would add a new entry to the end of the chain.

Lastly the hashing contained the different schemes that would be produced. In this class also contained a way to read the file through two arguments the input and output file. The scanner would read the input file by using nextInt function for each value. This would allow to not read the empty spaces within the text file and only read the integers. The hashing schemes except mine would use a division modulo and when a collision occurred it would go to one of the other classes to handle the specific collision assigned to that. Also in this class I figured out the system.Setout was a great way in being able to get everything into the output file. It sets the standard output stream with the print stream object that was specified.

What I might do differently next time:

Something I definitely could have done differently is combining the quadratic and linear probing into one function. The reason I would say this is because a lot of what the functions had were basically repeated code. The reason I ended up not combining these two functions is because I couldn't figure it out. Each time I would try it would produce a different result than the original that I had. One more thing that I was trying to do was figure out execution time as one of the stats. The problem with the execution time right now it seems to only be printing the time it takes for just one collision. If I didn't get stuck on the chaining I would have tried and figured out the execution time.

Ramifications of different hashing and collision resolutions:

One of method used was the division method was used for mostly all the results. The way this is done is through the equation h(k) = k mod n. The problem with using this number is that n which is the

hash table needs to be a prime number if you want the distribution to be more uniformed. Using a prime number will usually give good results if it is unrelated to any patterns in the distribution keys. The modulo values that were used for this problem was 120,113, and 41. The modulo not being greater than the size of the hash table would not give us a value that was greater than the array size. From the results in the below table it seems the modulo that was equal to the table size produced less collisions in comparison to the modulo that had prime numbers. This could be due to the linear probing methods or due to the input data not having any patterns.

	Division Modulo 120 Collisions	Division Modulo 113 Collisions	Division modulo 41 Collisions	My hashing scheme collisions
Linear probing	30	90		535
Quadratic probing	25	88		586
Chaining	18	16		48
Linear probing (Bucket size = 3)			3	
Quadratic Probing(Bucket size = 3)			3	

The hashing method that I decided to use was the multiplication method. The multiplication was taking the size of the table times the constant % 1. The constant can be a value from $0 < a < 1$ so the value that I choose to use was 0.3. When looking at multiplication as a hashing function it seems to cause a lot more collisions than the division modulo. The value of the A can end up causing the difference in the amount of collisions that occurs. This would make multiplication quiet inefficient

because of these increase in the number of collisions. So far from these two hashing schemes it seems that division modulo will produce the best efficiency over all.

Linear probing was one of the alogrithms that was probably easier to implement than the other two collision handling schemes. The way that linear probing works is it when the collision occurs from the hash function that was calculated it will start by searching for the next available spot that is closest. It seems that with division modulo 113 it ended up causing a lot of clustering. Clustering is when the collision scheme ends up creating a long runs of filled slots near the hash position of keys. 113 modulo ended up showing 90 collisions which seems like the hash function produced ended up having a lot of filled positions near it. So, it basically ended up searching the positions near it and as those positions got filled up due to them being inserted the more collisions occurred.

Quadratic probing was the next collision handling method. The way this collision scheme works here is it will calculate a new hash function based on the number of times the collisions occur. Quadratic probing tries not to run into the problem that linear probing has with clustering. The reason it doesn't happen with quadratic probing as much is because it is able to iterate the function to a higher value when the collision occurs. So, every time a collision occurs the new hash function becomes larger. If the table size is not a prime number, then it ends up running into problems because it will have trouble finding the open slots when the table ends up becoming full.

Last probing handling method was chaining. This method ended up being the hardest one for me to implement at first it was giving me all null values and not storing anything in the table. Then started figuring out the implementation from a little bit of reading the text and online searching. The chaining method was basically done by searching through the free space array for a location that is available to store the key. A linked list will end up being created when a collision ends up occurring repeatedly at the same spot. When this occurs, it will then add it to the end of the list as a new item.

Efficiency issues:

As expected hash tables are usually efficient when it comes to searching. The time complexity of a hash table when it comes to searching is around O(1). The problem with this design is that the table size is not a prime number. The table size is a fixed number at 120. The table size not being a prime number ends up causing more collisions which in turn will give us a worse efficiency time. So as the table become more full the more the number of collisions would increase. So if we had the same table size of 120 and increased the amount of values needed to be stored to 70 we would end up seeing a larger amount of collisions for all of the hashing schemes. As noticed with output 2 we saw a increase in collisions when I added an extra set of 10 values to the input which is in the input2.txt file.

What would you do to address some of the problems encountered?

Some of the problems encountered with this was trying to get a bucket of size 3 to work. At first, I implemented it completely wrong by having the buckets separated from each other. Then I just realized I can have the implementation just check whether each key is empty or not and just have it insert it in that position. Another issue that occurred was trying to get chaining to work correctly. At first the results would only produce a null value for everything. The reason for that was the hash table wasn't set up correctly. Getting the free space to work correctly so it was able to store the values that were repeatedly colliding.

What Did you learn about load factors?

The load factor is basically the number of keys that are going to be stored and dividing that by the total size of the table. From what I noticed the higher the load factor that was shown the more collisions would end up occurring. The reasoning for that would be due to the number of keys being increased not the total size of the table. The loading factor needs to remain under 1 because then the table becomes full and we cant insert any more keys. Something that can be done to not occur a load factor issue would be to end up resizing the hash table size. This would allow the hash table to increase the amount of space and help with efficiency by not having trouble finding a open slot.

What are the ramifications of deleting items from your hash table?

The problem with deleting an item from a hash table is that we need to make sure that the newly emptied slot is able to reach the records of the other slots that are after that deleted value. The slot that has also been emptied cannot just be left as empty because this will affect the other probes down the sequence.

Relevancy to Bioinformatics:

In bioinformatics the hash tables can be used to represent a genome sequence as multiple lists of genomic positions. So in genomics it basically a simple look up table that will point to the different possible k-mers. Another importance is that "Hash tables are used by various programs, including blast [2], patternhunter [3], shrimp [4], blat [5], nextgenmap [6], and gmap [7], to identify short oligomer (or seed) matches between a read and the genome" (Wu). The hash tables can also be used to also "align reads that can have mismatchses or indels relative to a genome"(Wu). Storing these values in a hash table allows for a quick read through because the time complexity of searching through a has h table is in constant time.

Reference:

1. Wu, Thomas D. "Bitpacking techniques for indexing genomes: I. Hash tables." Algorithms for molecular biology : AMB vol. 11 5. 18 Apr. 2016, doi:10.1186/s13015-016-0069-5