Steven Dea

Algorithms for Bioinformatics Spring 2020

Programming Assignment 2 Analysis

What I learned

In doing this project on hash tables, I learned an in-depth look at how to create a hash table using various collision handling methods as well as dealing with tables of different bucket sizes. I had already learned of how hash tables work with insertion, searching, collision handling, etc. when using a bucket size of 1, however I had never had any experience working with hash tables using a bucket size of n (in this case, n = 3). I somewhat understood the theory behind bucket sizes, but the idea of collision handling using different bucket sizes eluded me for a time while working on this project. I think the information that allowed me to really understand buckets is that a bucket is simply a grouping of slots, that when an item is hashed to it, allows for that particular item to be stored in any of the available slots. This reduces upfront hashing collisions because instead of only being able to store a single item in a hash bucket, we can store however many items up to the size of the bucket we are working with. When we have to handle collisions using a linear or quadratic probe, we first need to “search” through the entire bucket to make sure it does not have an open space. If it does have an open space, then we insert there and no collision handling is necessary. If it doesn’t have an open space, then we either linearly or quadratically probe to the next bucket and search that bucket for empty spaces.

What I might do differently next time

If I were to do this project over, I would try to implement my linear and quadratic probing functions in a single function. I am not as happy with my functions merely because they do have some repeated code which violates the DRY principle. I think that in understanding linear and quadratic probing a bit more before I started, I would have been able to implement the two functions in a single function to avoid this code repetitions. Additionally, if I were to do this project again, I would spend more time really understanding how buckets work and how to implement collision handling while using buckets of size n. As it stands, I spent quite a bit of time writing and rewriting the code that went into collision handling with buckets because I did not fully understand what it was that I was trying to do. To be fair to myself, I did **think** that collision handling with different bucket sizes worked a certain way and it wasn’t until I spent some time looking it up to double check my ideas that I realized that I was incorrect.

Design justification

Something that I learned from my previous assignment was to make my code more modular and able to be reused. I decided to split this program into two modules, the first was the main driver program that handled the file I/O and formatting as well as running the actual program that was required by the assignment. My second module consisted of a hashTable class which contained the hash table data structure itself as well as all of the required functionality. I split the functionality of the hash table into a separate method to allow for greater modularity and avoid as many confusing logic errors as possible. For the three shared methods of linearProbe, quadraticProbe, and hashInsert, I designed them to take in variables regarding which type of hash function was being used, which modulo was being used, and which type of collision resolution was being used so I could run them in the main() function with as little manual programming as possible.

Ramifications of different hashing and collision resolution

**Modulo hashing:** When I took data structures, I learned that modulo hashing was generally the best “go-to” basic method of hashing as it limited the resulting hash value by the size the modulo, which is generally the size of the table. Modulo-based hashing is quite straight forward with the key being divided by the modulo and the remainder being returned. This is useful because as I stated earlier, it limits the value of the resulting hash by whatever the modulo in question is. In our case, since our modulos were 120, 113, and 41, our resulting hash values were never greater than the size of our hash table (which is good since if it were, we would end up needing a lot of error handling to deal with array indices out of bounds when checking for empty spots/attempting to insert). When deciding on a modulo, it is generally better to use prime numbers as there is a lower chance of having collisions when our input data contains patterns.

**My hashing:** The hashing method I chose to employ was one of my own design. I first multiplied the input key to be hashed by 7 and then summed up the value of each individual integer within the resulting key (i.e. a key of 12345 would sum up to 15). I then multiplied the resulting key by 2 and subtracted it from 120. The idea behind the various multiplications and subtractions that I added into the hash was to limit the resulting hash to always be less than 120 (our table size). I learned through my output that this method of hashing is quite inefficient as it seems to be prone to clustering, thus increasing the number of collisions present. While my hash function is not particularly efficient, I wanted to try something new that did not involve any sort of division.

**Linear probing:** Linear probing is quite an easy collision resolution handling scheme to implement, but it does have its drawbacks. I noticed that in my hashing where I implemented linear probing, my hash table suffered from clustering of data due to the way linear probing resolves collisions. Since it searches for the next available slot closest to the original collision, it ends up stacking keys in clusters depending on any patterns that occur within our input data sets. Clustering tends to affect the amount of collisions that occur in a hash table when data has patterns to it because as more keys are inserted and certain areas of the table fill up, every key that is attempted to be inserted in the filled locations encounters more and more collisions as it checks the clustered area.

**Quadratic probing:** Quadratic probing is also similarly relatively easy to implement and it tries to mitigate the issues with linear probing where clustering tends to be an issue. It does this by iterating up in a quadratic manner when collisions occur. Issues tend to arise, however, when the table size is not of a prime number and when our table will be more than half full of values. To make sure that a quadratic probe will hit every value in a table when searching for an open slot, it must follow the two rules stated above. Due to the fact that our table size is not a prime number, it violates that rule and as such, our quadratic probe cannot be guaranteed to probe every single location within our hash table when a collision occurs. To mitigate this issue, I implemented a line of code stipulating that once our quadratic probe has run the same amount of times as our table size, then I will revert to linear probing to resolve any issues. This is to avoid an probing loop that ends up with an integer too large and returns a negative value.

**Chaining:** Chaining in our assignment is different from what I understood chaining to be in my data structures class. My implementation of chaining came from an idea put forth by one of my classmates in the discussion section of our class portal. I decided to implement open address chaining by creating a Linked List of overflow values. Every time there was a collision on an initial hash insertion, instead of resolving the collision immediately, I added the key in question to my Linked List at the end. Once I had inserted all of the values that could be immediately inserted into my hash table without collisions, I then filled up my hash table in a linear fashion beginning with the first empty space (starting from 0) and ending with the last empty space pulling key values from my Linked List in a FIFO manner. If there were no more empty spaces, I then stipulated that all of the keys that had not yet been removed from the Linked List were keys that failed to be hashed into the table. I found that this manner of collision handling drastically reduced the amount of collisions that occurred as collisions were only registered on the initial hash. Instead of trying to resolve the collision immediately and potentially registering further collisions as it tried to be inserted, the key is set aside to be resolved at the end of the function.

Issues of efficiency

Hash tables are highly efficient when it comes to searching as they are **supposed** to be O(1). However, design flaws in hash tables and resolution strategies do not always make this so. For example, our table has a size of 120, which is already not ideal as it is not a prime number. Additionally, our table has no ability to resize itself to accommodate additional keys. This, in addition to our resolution handling leads to issues of efficiency where as the table gets more and more full, the efficiency of our insertion and subsequent searching (which was not implemented in this project) goes down. In my example data sets, I found that when using larger data sets that had more keys than 50% of the table size, my collisions increased. This is due to the issues that I have stated above and could be fixed by the ability to resize the table.

How would I address problems encountered

The large problems that I encountered while doing this project revolved around the theory of how bucket implementations worked with collision resolution. I was under the impression at first that collision resolution methods treated buckets of size 3 the same as buckets of size 1 once the initial bucket was full. However, after lots of googling and questions asked, I realized that it actually works by checking the initial bucket, finding it is full, and then probing to the next bucket, and checking the entire bucket, etc.

Another problem that I ran into was making sure that my collision resolution logic took into account the table size to avoid any array index out of bounds exceptions. This was particularly prevalent in my implementation of collision handling using buckets of size 3. The way I designed my methods required me to check the key % 41 and then multiply it by 3 to determine the hash table location as a factor of 120 slots.

What did I learn about load factors

The load factor of a hash table is simply the number of keys stored in the table divided by the total size of the table. We want the load factor of any hash table to remain under 1, otherwise that would mean that the table is full and we cannot insert any more keys into the table. If this is the case, we end up with keys that fail to hash into our table and we lose data. A key part of hash tables that is not included in our assignment is the ability for them to resize themselves when they reach a certain load factor that is chosen by the programmer. This allows hash tables to keep up with an increasing number of keys while also keeping in mind space efficiency. If space were not an issue. We could create a massively large hash table so that we’d never have to resize our table as keys increase, but this is not practically feasible as that would partition off a chunk of space that would remain unused for quite some time. As our load factor approaches 1, more and more collisions will occur and reduce the efficiency of our hash table dramatically.

Ramifications of deleting items from table

A big ramification of deleting an item from a table is that of deletion of a key at a hash location when there have been multiple collisions on that same location. For example, say we have 3 keys: a, b, and c. All three of these keys hash to the same location of 1. For our example, we will say that a is inserted first and b and c are then inserted using linear probing. If we were to delete a and leave an empty location, then we would no longer be able to track down b and c. We would search where b and c **should** have hashed to and return empty, thus not triggering a linear probe to find the actual locations of b and c. One way around this issue is to instead leave some sort of flag in the location of the deletion to tell our program that this slot is available, but has had a value be deleted from this location. In this way, when our search function looks at location 1 to find b or c and instead finds this flag, we know that a key was deleted from this spot and we should continue our linear probe to search for where b and c were inserted. Another less efficient option could be to delete a key and then re hash all of the keys that are currently stored in the hash table to re-insert them into their new correct locations. I do not think that this idea makes sense for every single deletion, but after many deletions a table could get a bit “messy” and might benefit from a rehashing.

Relevancy to Bioinformatics

Hash tables are a widely used data structure for the indexing of genomes. By storing genomic indexes in a hash table, we can search for specific genomic positions or specific oligomers of a given size in a very quick manner through the use of hashing. Due to the massive size of genomes and the large amounts of genomic data that bioinformaticians need to sift through, a hash table is the most efficient data structure when it comes to looking up specific keys. A big example of how this efficiency helps bioinformaticians is in trying to align reads to a reference genome. By using a hash table to store and index the genomic intervals of a reference genome and reads, we can more quickly align these reads to the genome through the high time efficiency of searching using hashing.

Another example could be if we want to find the most common sequence of kmer in a given genome or part of a genome. By using a hash table, we can find all kmers in a genome and then store the all of the locations of a specific kmer in a particular index. In doing this, we can then find every location of a repeat kmer using a hash table and some implemented method of collision resolution, most likely chaining.