Algorithms for Bioinformatics
Project 2
Shuting Chen

Introduction

This project is doing hash table implementations. An ideal hash table will hash values to keys which are equally likely distributed. To analyze the performance with customized table size, bucket size, and input size, we can implement a hash table and count physically how many collisions happened within the build-up process, and analyze asymptotically what the running time would be. Additionally, the ramifications of duplicate inputs and deletions will be discussed as well.

Code Explanation

a. Basic Idea

Referring to the documentation of the project, my code is implementing the functionalities of the hash table, and reporting the related statistics for further analysis. To store a value into a hash table, the key of that value should be calculated. If the slot of that key is occupied, then update that key by a collision handling scheme: linear probing, quadratic probing, or chaining. Repeat this step until an available slot is found, otherwise the value will be marked as failing to insert into the hash table. In addition, the collision will be counted in the whole insertion process in terms of primary and secondary collisions.

b. Data Structure and Implementation

I implemented two classes for this project: HashTable and TableNode.
TableNode is the basic unit for saving slots on a hash table. It has 4 properties which have a default value of -1:

- key: the index of the slot to save the value
- value: the value stored in the slot
- next: the next node (only used on chaining strategy)
- primary_insertion: a flag to help count primary and secondary collisions

HashTable is an overall structure containing all the information it may need, such as collision schemes, the sizes of the table and the bucket, counters for primary and secondary collisions, and failed values of insertions. The most important property of HashTable is an array of TableNode objects, which is a dynamic array storing the key-value pairs for each slot on the hash table. Besides these properties, functions such as insertions and collision handling strategies are implemented in HashTable class as well.
Another data structure I used in this project is a stack for keeping track of the free spaces in the hash table. This stack is constructed with all the available indices and randomized to make sure the insertion will not be clustered.

Efficiency Analysis

This part will focus on the running time of data insertion, deletion, and searching, also concerning different collision handling strategies: linear probing, quadratic probing, and chaining. Load factor, $\alpha = n/m,$ is used as an important value in this running time analysis, where n is the number of occupied slots and m is the table size. Since it's open addressing, $n \leq m$, meaning that $\alpha = n/m \leq 1$.

Insertion:
The average running time to insert a value into a hash table is constant. This step is composed of two sub-steps: 1. find an available key; 2. insert into the hash table. Both of these sub-steps are in $O(1)$ time, making the overall running time $O(1)$.

Step 1: find an available key

Assuming we are using uniform hashing, each key between 0 to m is equally likely to be hashed. Based on that, we can know the probability of accessing an occupied j-th value is (n - j + 1)/(m - j +1). When n approaches to m ($n \leq m$), that probability approaches 1/(m - n +1).

With further transformations by C.24 and A.6, that probability would be 1/(1-α). Therefore, assuming we have a constant α, then the running time for an unsuccessful search will be $O(1)$. For example, when α = 0.8, the number of unsuccessful search would be 1/0.2 = 5, which means it needs 5 unsuccessful searches on average to find a empty slot for a hash table with α = 0.8.

Step 2: insert into the hash table

This step only needs constant time since we have already got the key available to save the value. By using hash_table[key] to access that node object, we should save the value on that slot in $O(1)$ time for linear and quadratic probing. For chaining, we have to update the pointer if there is a collision at the original key. But that would not affect the overall running time because it can be done in constant time as well.

Searching:
Searching can be unsuccessful, similar the insertion step, we need 1/(1-α) running time to find the right slot for a given value. Assuming α is constant, this searching step would cost constant time.

Deletion:
The running time for deletion is constant. For linear and quadratic probing, all I need to remove a value from a hash table is searching and clearing all the information in that slot to -1 (default value). As proved above, searching by a given value needs 1/(1-α) time, which is $O(1)$. Clearing the node back to -1 will only take constant time as well. Hence, the overall deletion cost for linear and quadratic probing is $O(1)$. For chaining, the searching step also need 1/(1-α) running time to find the node. And its clearing step will need to mark that node as deleted and redirect the pointering of that node's parents to its next node. In addition, push the key back into the stack of free space. Even there are more operations in the chaining process, all these will be done in constant time, meaning that the overall running time is still asymptotically 1/(1-α), which is $O(1)$.

Real case cost:

modulo	linear	quadratic	chaining
41	64 (0.9)	65 (0.9)	
113	402 (0.9)	233 (0.9)	40 (0.9)
120	246 (0.9)	144 (0.9)	44 (0.9)

Table 1. Number of collisions for 3 collision handling mechanisms.
* table size = 120, bucket size = 1, modulo = 108
* load factor α = 0.9 for all the test cases

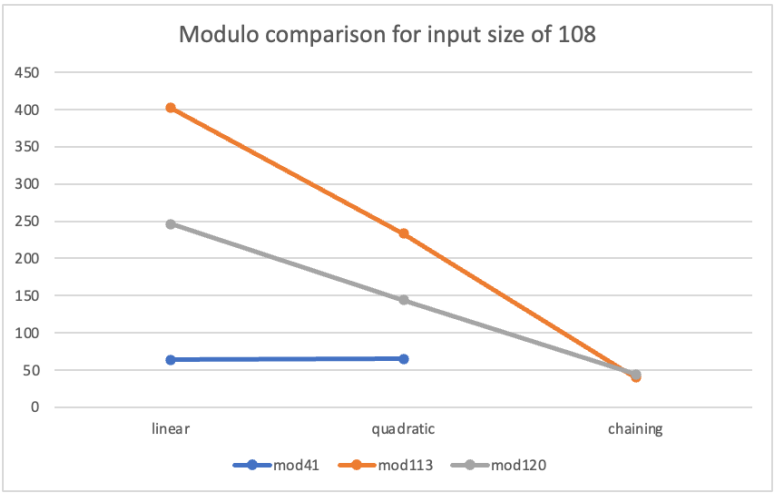
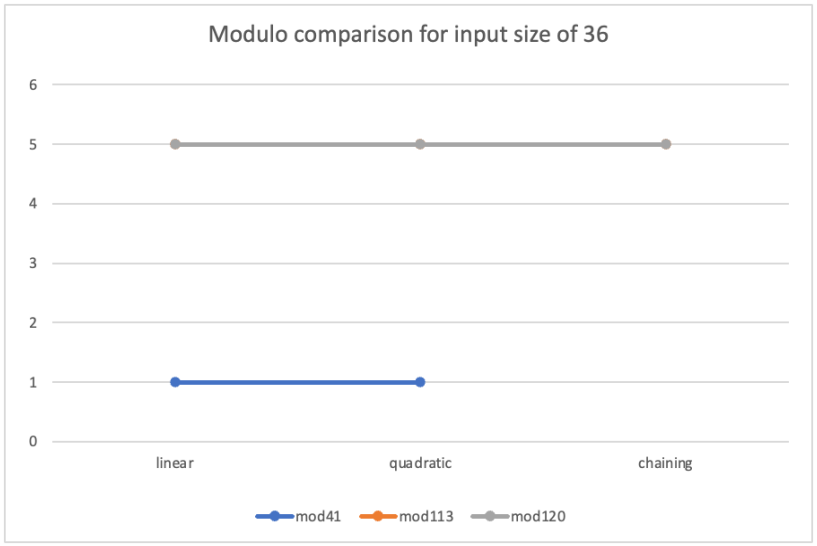


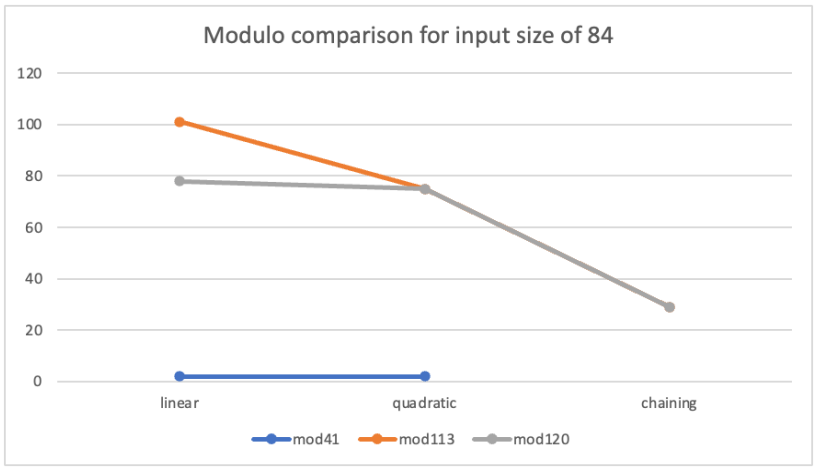
Figure 1. Number of collisions for 3 collision handling mechanisms.
* table size = 120, bucket size = 1, modulo = 108

From Table 1 and Figure 1, we can see the real number of collisions happened in 8 test cases. When $\alpha = 0.9$, the average collision counts for each insertion will be $1/(1-0.9) = 10$. That is the average number of collisions happened when $\alpha = 0.9$, which is when the last value is inserted. Before that, α is approaching from 0 to 0.9. Since there are 108 input values, theoretically the expected collisions happened would be $(0 + 10*108)/2 = 540$. That does not apply to cases with bucket size of 3 (module value = 41) since in this scenario, each key will have 3 available slots.
From the real case statistics above, I think that the upper bound of cost is in accordance with the real counts of collision. In addition, I found that quadratic probing performs better than linear probing, and chaining performs better than quadratic probing. More details are provided in the following paragraphs.

Collision Comparison

1. the effect of modulo values
   At the beginning of the collision analysis, I compared the modulo values of 41, 113, and 120 on three input cases (input size of 36, 84, and 108).





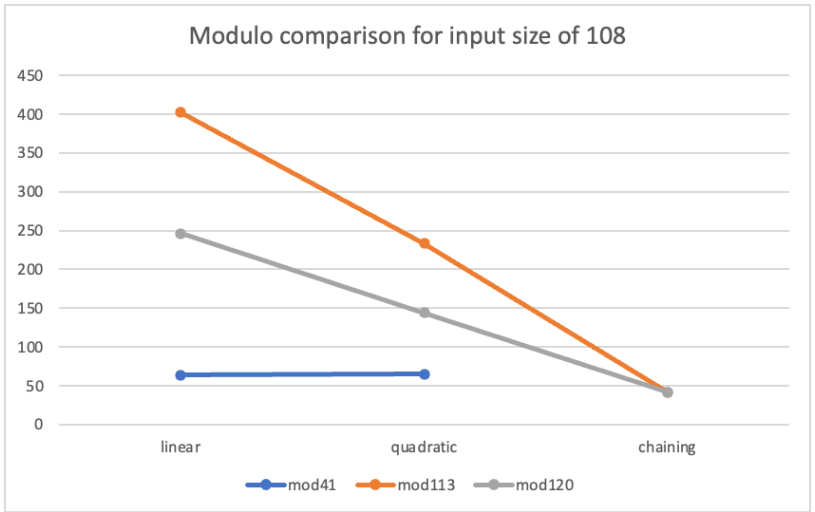


Figure 1. collision comparisons for different modulo values.
*Note: for input size of 36, the lines of “mod113” and “mod120” overlap.

Based on Figure 1 above, we can see modulo value 41 always gives a better performance than 113 and 120. That is the case in which bucket size is 3 and each key contains 3 slots. For a given key, as long as on the hash table there is a free slot at the location of that key, insertion will be successful without collision. That is a trade off with the search operation,

because for each key, there should be 3 comparisons on that key, instead of only once when bucket size is 1.
Another observation of Figure 1 is that chaining always performs better than linear and quadratic probing, except that chaining is not used when bucket size is 3 (see lines for mod41). I will compare these three collision handling strategies for further details. Since the number of collisions increases with the number of input, I will use input size of 120 to maximize the difference for a better resolution.

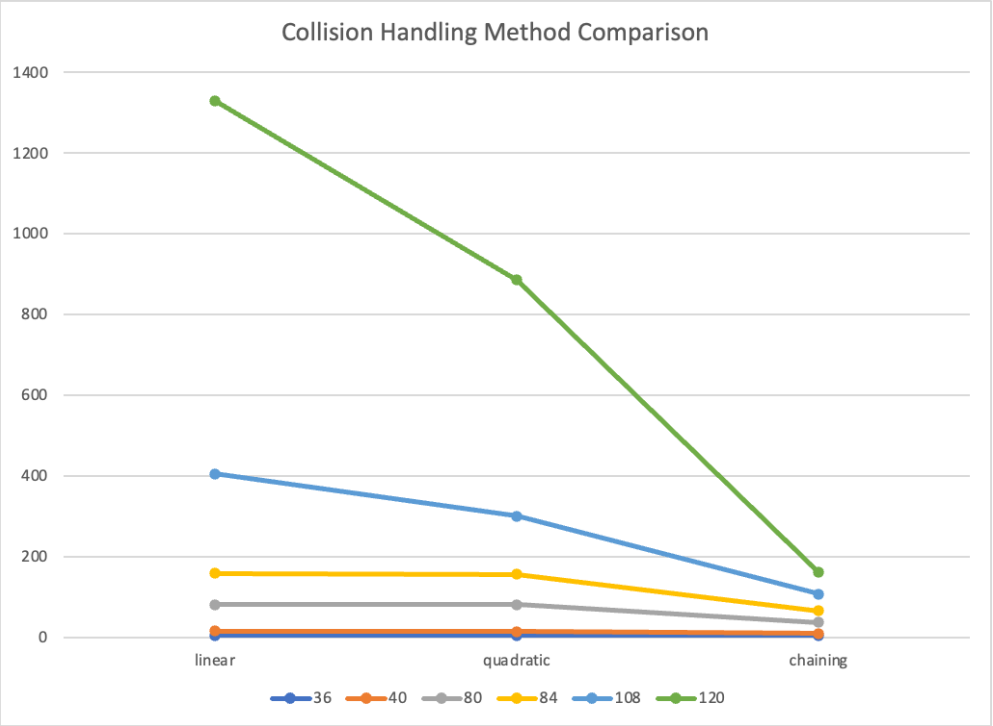


Figure 2. Number of collisions for different input sizes by three collision handling methods (modulo = 120): linear probing, quadratic probing, and chaining. There are 6 lines plotted, each line represents one input size of 36, 40, 80, 84, 108, and 120.

From Figure 2 above, there is an obvious decrease of collisions for chaining compared to linear and quadratic probing. This is because the implementation of free space and pointers for chaining helps to avoid collisions brought by calculating new keys from probing strategies. As a tradeoff, chaining needs more memory to store the free space stack and an extra pointer for each slot. But for large data inputs, such as the green line above representing input size of 120, the number of collisions can reduced by a half by chaining than linear probing.
Another reason that chaining offers a better performance is about its randomized keys on the free space stack. If the stack stores keys in sequential sequence [1, 2, 3,..., m], then the last-in-first-out(LIFO) property of stack will always pop the top element as the key of the next available slot. This will make the slots clustered together. Assuming uniform hashing will generate the hash value equally like among the hash table keys, a cluster of occupied slots will trigger more collisions for next insertions. That's the same reason why quadratic probing performs better than linear probing. Quadratic probing will generate keys farther apart which prevents clumping at one specific place.

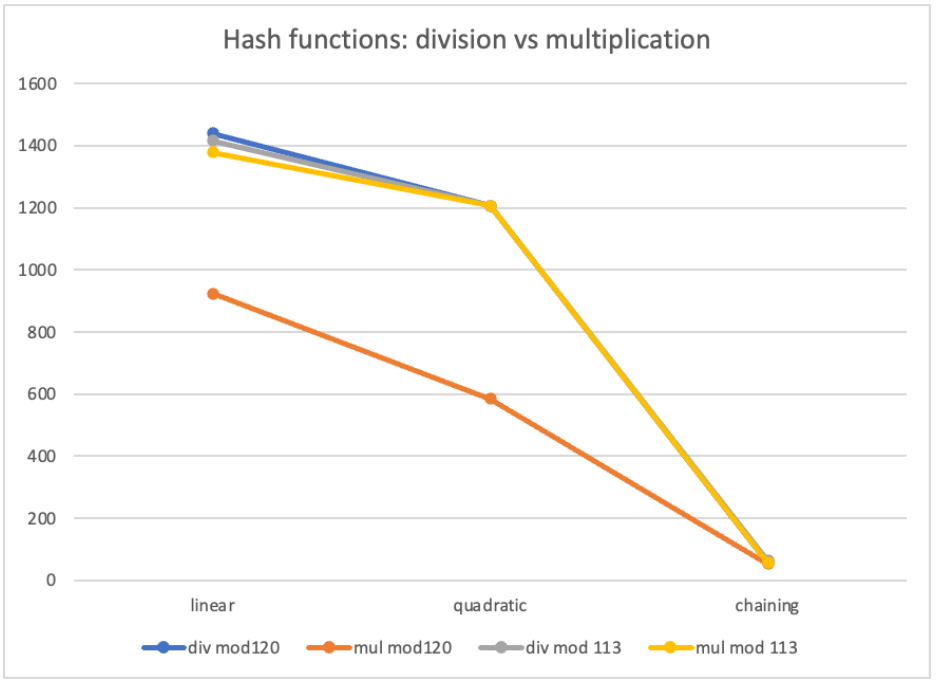


Figure 3. Number of collisions generated by hash function of division and multiplication.

In addition, I tested my hash scheme with the classic division scheme. I used the multiplication hash function h(k) = m(kA mod1), where m=120 and k=0.618. I compared these 2 hash functions under table size of 120 and input size of 120 with modulo value 120 and 113. Each case performed similarly, except the multiplication function performed better at modulo value of 120. I think that is because as the textbook suggested, the k value of 0.618 is the classical preferred value used for multiplication. And it does return better hash values more uniformly to the table.

Extra Work

In previous analysis, quadratic probing has default values of c1 = 0.5 and c2 = 0.5. Then I tried different c1 and c2 values for quadratic probing.
When c1 = 0.5, different c2 values (0.4 ~ 0.9) bring different collision counts:

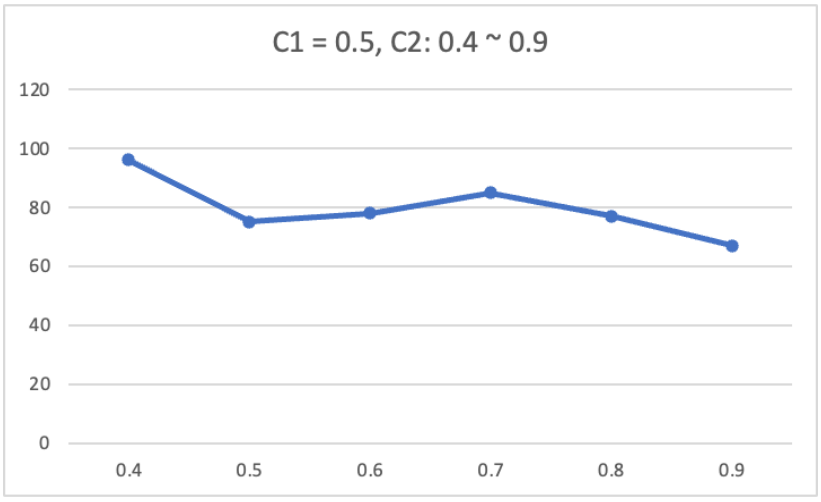


Figure 4. Number of collisions vs different c2 values (c1 = 0.5).

As shown by Figure 4 above, the least number of collisions is done by c1 = 0.5, and c2 = 0.9. This is because c2 is constant for i^2 and c1 is for i . A larger c2 value will make the probing key to go further on the hash table, which brings less collisions and less clusters.

Using the same strategy, I analyzed the different values of c1 (0.4 ~ 0.9) as c2 = 0.5.

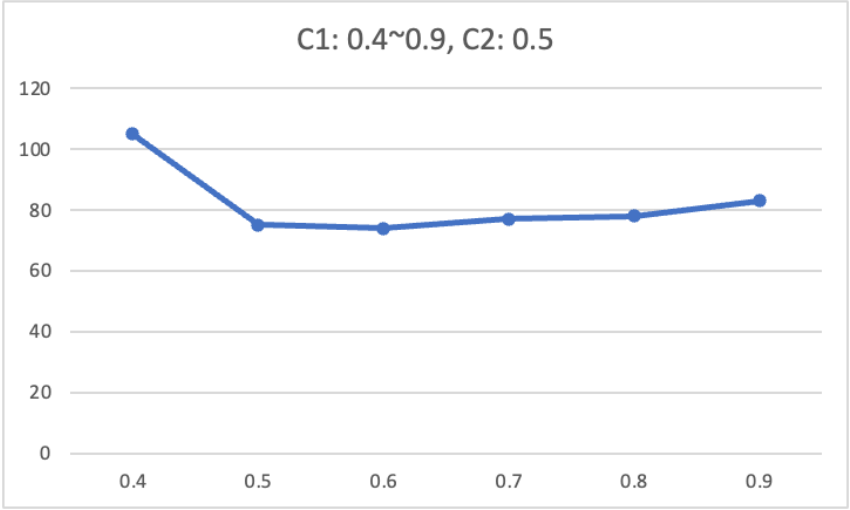


Figure 5. Number of collisions vs different c1 values (c2 = 0.5).
As shown on Figure 5 above, as long as c1 is not too small, c1 does not affect the performance too much. That also makes sense that c1 is the constant for i and it will not affect the probing value greatly.

Discussion

a. About this project
I learned several things from this project:
1. The performance of a hash table highly depends on the quality of the hash function. Also, a good table size and a bucket size are influencing the performance as well. But all these parameters need case-by-case concerns, which there is no golden rule and it needs the algorithm developer to fine-tune the parameters.
2. For an open addressing hash table, the load factor can be 1. But in practice, be careful to choose the proper table size to hold the inputs. Choosing a table size which makes load factor near 1 is dangerous to bring collisions and insertion failures. Hence, it's better to choose a table size greater (but not too much greater) than the input size.
3. Comparison among strategies might not have obvious differences on a small dataset, therefore we need to test under datasets with different sizes to evaluate the ramifications.
4. Memory sometimes will be a tradeoff with computation. I need to evaluate the algorithm by both concerning the memory and the time cost.

b. About the Bioinformatics
The first impressive thing I learned about hash tables is their quick access to data. That would be great for Bioinformatic studies because a lot of data about genetics is

cold data over gigabytes. If we can develop a preferable hash function, then a hash table would be a great tool to access data in constant time. For example, there are over 20,000 genes for each person. Usually except genome-wide studies, clinical research is more focused on a part of these genes, such as to find potential targets or to diagnose diseases. Therefore, if we could save genomes into a system based on hash tables, then for the studies with hundreds or thousands of clinical trials, accessing data will be much quicker. What we need to figure out is how to find a good hash function to map values (gene ids or gene names) evenly into the hash table. Like the figure below, by saving a pointer at each slot pointing to the corresponding genetic sequence, we can get that sequence quickly by the key calculated from the hash function.

key	slot	sequence
0		
1	gene 3	ATGGTCTTGA…
2		
3	gene 2	TCGGAAGAGCCC…
4	gene 1	CAAATATACCTTC…

c. Future thoughts

Next time, I would like to see if the free space stack can be utilized to help probing strategies when bucket size is greater than 1. In this project, I implemented and tested hash tables with bucket size of 3. Everytime I insert a given key into that table by a given key, I have to iterate through the 3 values on that key. If bucket size increases, iterating for every insertion and searching operation will be expensive. Therefore, it might be helpful to repurpose the free memory stack to deal with hash tables with a big bucket size. Only pop the key out when all the slots at that key are occupied, which means that key is full and never being considered as a free space.