CSCE 221 Cover Page
Programming Assignment #5
Bonus Due Date: November 27th, 11:59pm
Final Due Date: December 2nd, 11:59pm

First Name: Tien Last Name: Tavu UIN: 128002442

Any assignment turned in without a fully completed cover page will receive ZERO POINTS.
Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students	Other People	Printed Material	Web Material (URL)	Other
1.	1.	1.	1.	1.
2.	2.	2.	2.	2.
3.	3.	3.	3.	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today's Date: November 27th, 2019

Printed Name (in lieu of a signature): Tien Tavu

Theoretical Statement

The chaining hash table is one of the three available hash table implementations. This implementation is a simple and efficient hash table where collisions are kept in the same bucket, and in the case of C++, an STL list. The table is stored in memory as a vector containing STL lists, where the key is a hashed key. The STL list contains elements of pair objects where the first and second member values correspond to their respective key and value pairs. The `insert` operation checks if there is an existing key value pair object in the STL list through a for loop – if there is, the function would increment the value by one. If the dictionary word that is being appended does not exist in the STL list, then it would be added with a value of one. The operation has a worse-case running-time complexity of $O(n)$, which would only occur if all the hashed key values are mapped to the same index of an STL list; otherwise, the average-case running-time complexity of this operation is $O(1)$. Similarly, the `remove` operation goes through a for loop that determines the respective key value pair to remove from the table and undergoes an average-case and worse-case running-time complexity of $O(1)$ and $O(n)$, respectively.

The linear probing hash table is the next available hash table implementation. The linear probing implementation, along with the double hashing implementation, would utilize an "open addressing" technique where collisions are kept in a different bucket. In the case of linear probing, the next bucket is found by an equation expressed as $(h_1 + i)\ \%\ N$, where h_1 is the hashing function utilized for all three implementations, and i and N correspond to the current index iteration and the capacity of the buckets, respectively. The table is a vector of pair objects where the first and second values correspond to their respective key and value pairs. The `insert` operation is a while loop that continually tries to find an open bucket to append the

dictionary word or an existing bucket that contains the correct key to increment the value of the pair object. The operation has a worse-case running-time complexity of $O(n)$, which would only occur if all the hashed key values are mapped to the same index of the table vector, as the program would probe over `n` elements; otherwise, the average-case running-time complexity of this operation is $O(1)$. Similarly, the `remove` operation goes through the same while loop that determines if the respective key value pair is correct to remove from the table – this operation undergoes an average-case and worse-case running-time complexity of $O(1)$ and $O(n)$, respectively, with a similar explanation to the `insert` operation.

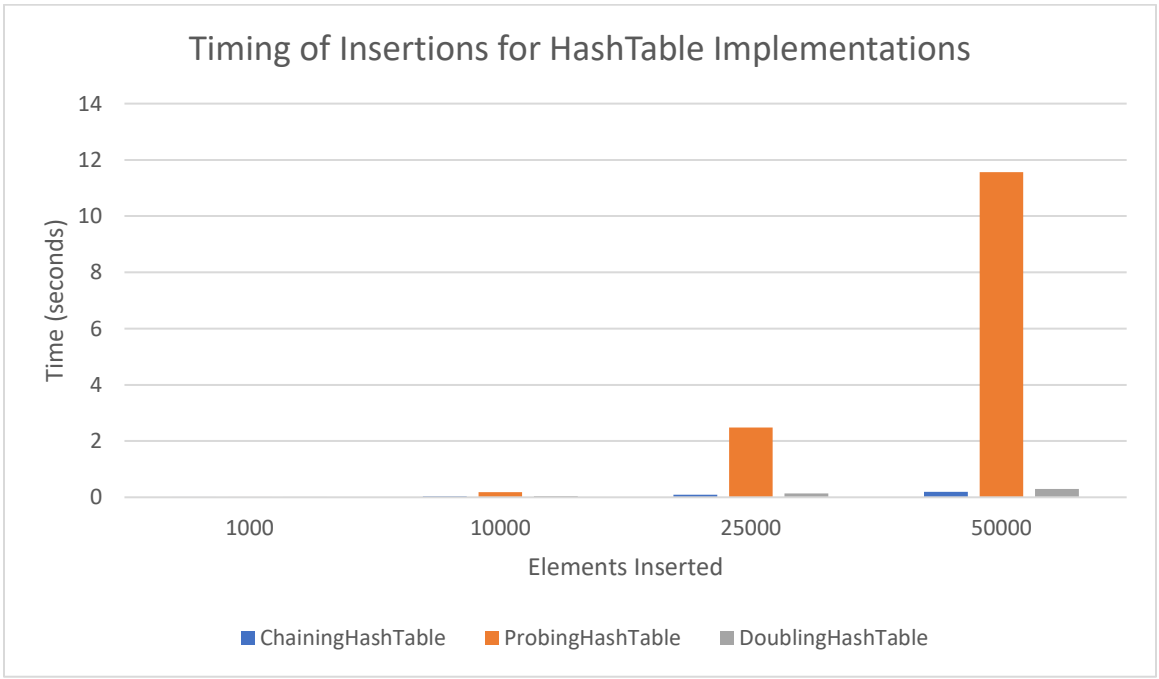
The last hash table implementation is the double hash table, which undergoes an "open addressing" technique as expressed earlier with the probing implementation. The main distinction is how the double hash table will calculate the next bucket's index – it can be expressed as $(h_1(k) + h_2(k) \times i)\ \%\ N$, where h_1 is the original hashing function utilized for all three implementations, h_2 is a second hashing function designed for the double hash table which utilizes the polynomial hash code technique, and i and N correspond to the current index iteration and the capacity of the buckets, respectively. The implementations for the `insert` and `remove` operations are all the same as the probing implementation but the calculation for an open bucket. This leads to both operations having an average-case and worse-case running-time complexity of $O(1)$ and $O(n)$.

Overall, there are three different hash table implementations and even with operations that are specialized for their respective implementation, the `insert` and `remove` operations witness an average-case and worse-case running time complexity of $O(1)$ and $O(n)$. However, after closer observation with the algorithms for each implementation's operations, the chaining hash table would be the most optimized implementation as collisions are stored in buckets,

which wouldn't need to create unnecessary iterations like the "open addressing" technique in order to determine an open bucket. The double hash table would seem to be the next best solution as it creates more "randomness" in determining an open bucket with a second hashing function, which would lead to the probing hash table being the worst as an open bucket would have to be determined from a great number iterations.

Experimental Analysis

A total of three hash table implementations were tested utilizing Texas A&M's CSE servers with the use of PuTTY to both compile and execute the project code. The PuTTY application would be running on a Dell XPPS 13 2-in-1 laptop that contains 16 GB of RAM and an Intel i7 core processor. Each implementation was tested with intervals ranging from the first 1,000, 10,000, 25,000, and 50,000 words in the provided dictionary file as they were inserted in each hash table. Three trials were conducted for each interval and implementation – an average was calculated to be graphed, which can be seen below.



The chaining hash table was the best available implementation in terms of running time to insert elements, no matter the amount. The double hash table would be the second implementation in terms of the least amount of time to insert elements, with the probing hash table being the worst. Again, the number of elements does not seem to impact the rankings of the running time for each implementation.

Discussion

The results were expected after a closer look towards the algorithms for each operation and implementation in the theoretical statement. The chaining implementation separates collisions into buckets, which seems to be the most optimal solution when dealing with such issue as there are less elements to deal with. With the probing implementation, collisions are dealt with by iterating towards the next open bucket, which can be a timely operation as seen in the graphs. The double hash implementation solves the timely operation by adding more "randomness" towards searching an open bucket, which drastically reduces the running time of the insertion operation, but the chaining implementation still prevails.

The experimental data deviated from the theoretical runtimes for the probing implementation. The average and worst-case running time complexities for the `insert` and `remove` operations were calculated to be $O(1)$ and $O(n)$, just like the two other implementations. However, the timings and graphs doesn't show it to be anywhere close towards the other two implementations, and it can be seen by having a numerous number of collisions when inserting, creating an unnecessary number of iterations as the program tries to find an open bucket.

Overall, the best performing implementation would be the chaining implementation, as it performed the best no matter the capacity and the number of elements being inserted. This can be

attributed by the implementation storing collisions in separate buckets and dealing with less elements later, as opposed to the “open addressing” technique utilized by the probing and double hash implementations which goes through numerous buckets to determine an open one for insertion and an existing one for removal. The probing and double hash implementations also performed on par of its expected performance regardless of the number of elements being inserted towards the hash table.