Course: Data Structures, Algorithms, and Their Applications – CPSC 319
Assignment 5 Questions

Instructor: Leonard Manzara
T.A.: Xi Wang
Tutorial Section: T04

Student Name: Danielle Jourdain
UCID: 30114566

Question 1

My hash function is based on the Jenkins one-at-a-time hash (source: https://en.wikipedia.org/wiki/Jenkins_hash_function). This hash function takes each character of a String and shifts it bitwise as well as applying XOR operations. All of this is added to the current hash value until all characters of the String have been processed. Afterwards, the hash value is shifted bitwise a few more times, and a final XOR operation is applied. Finally, there are a few simple arithmetic operations that are applied to ensure the hash function returns a value in a valid range. All the manipulation of the String's individual characters on a bitwise level allows for a pretty random hash function. Out of the many functions I tested, this one gave the best results.

Question 2

Although this hash function was the best out of the ones I tested, there is still a lot of room for improvement. This function only gives approximately 32% hashing efficiency. To improve it, the hash function needs to randomly distribute the values more than it currently does. It also needs to decrease the longest chain of reads. Currently this value is 54 reads. This is a long chain of words to read through when looking for one specific word. One way to improve this potentially would be quadratic probing. This would stop the issue of primary clustering which is one of the biggest drawbacks of linear probing. Another potential change that could improve performance would be using separate chaining. This means each element in the table could hold many table entries. Overall, this would decrease the issues that both linear and quadratic probing have. This method could cause problems as well, since the LinkedLists of each row can get long causing the program to slow down when searching for an element.

Question 3

My hash function is very far from ideal. On average it takes 2.2 reads to find a result. That means, on average it takes at least 2 reads to find the desired result. This means that usually, there was a collision when trying to insert the element in the spot indicated by the hash function, and probing was required to find a new spot. The probing worked well in this case, since this value indicates that on average it found a spot after probing just one time. This case sounds quite reasonable, but when looking at other statistics it shows that the longest chain of reads is 54 reads. Unfortunately, there is at least one element that had to probe 54 times to find a spot in the table. Ideally, there would be no collisions, so the average number of reads per record would be 1. Assuming the capacity and therefore the load factor is kept the same, that means an ideal hash function would have 71% efficiency for this implementation. This means my hash function is about 39% short of being ideal.

Question 4

After some very simple tests, I found that numerical strings performed slightly better (about 1%) than alphabetical ones. Strings of length N made from alphabetic characters can have at least 52^N possible choices (the entire alphabet in both upper and lower case, as well as any special characters. Meanwhile numeric strings of length N can have 10^N possible combinations.

At first having more possible string combinations seems like it should produce a better hash result. However, my hash function is not ideal. This means that some distinct strings hash to the same table index.

Assuming there is an infinite table size, and we are hashing all possible string combinations, the alphabetical strings will produce many more collisions since there are many more possible strings. The numerical strings will still cause collisions, but since there are fewer possible strings overall, there will be fewer collisions when hashing all of them. Even though the numerical strings will perform better, the two possibilities will behave very similarly. From the simple tests, there was only about a 1% difference in hashing efficiency. This means there will be a ratio of hashes that cause collisions to ones that don't, but that the numerical strings perform a bit better.

Another potential reason the numerical strings performed slightly better would be due to the ASCII codes of numbers. They are generally lower than the codes for letters. This would help the hash function perform better, since it would be able to avoid overflow better. Since there are many addition, shifting, and XOR operations in my function, it is very easy to encounter arithmetic overflow. There is a small segment of code in the function to correct this, but it causes the function to be much less efficient. Using smaller ASCII codes means arithmetic overflow is much less likely, so the hash function will be slightly more efficient with numerical strings.