Project 3
Zayne Bonner

| | Hash Table Sizes | # of steps to find a wor | Expected # of steps to f | Total # of Steps | Total # of diff words | Words total | Expected Length of list |
|---|---|---|---|---|---|---|---|
| | 500 | 3.130631427 | 10.933 | 584204 | 10933 | 186609 | 21.866 |
| | 1000 | 2.075301834 | 5.4665 | 387270 | 10933 | 186609 | 10.933 |
| | 2000 | 1.542808761 | 2.73325 | 287902 | 10933 | 186609 | 5.4665 |
| | 5000 | 1.223226104 | 1.0933 | 228265 | 10933 | 186609 | 2.1866 |
| Expir. | 500 | 4.033572872 | 10.933 | 752701 | 10933 | 186609 | 21.866 |
| | 1000 | 3.685529637 | 5.4665 | 687753 | 10933 | 186609 | 10.933 |
| | 2000 | 3.644331195 | 2.73325 | 680065 | 10933 | 186609 | 5.4665 |
| | 5000 | 3.644331195 | 1.0933 | 680065 | 10933 | 186609 | 2.1866 |
| | 10000 | 3.644331195 | 0.54665 | 680065 | 10933 | 186609 | 1.0933 |

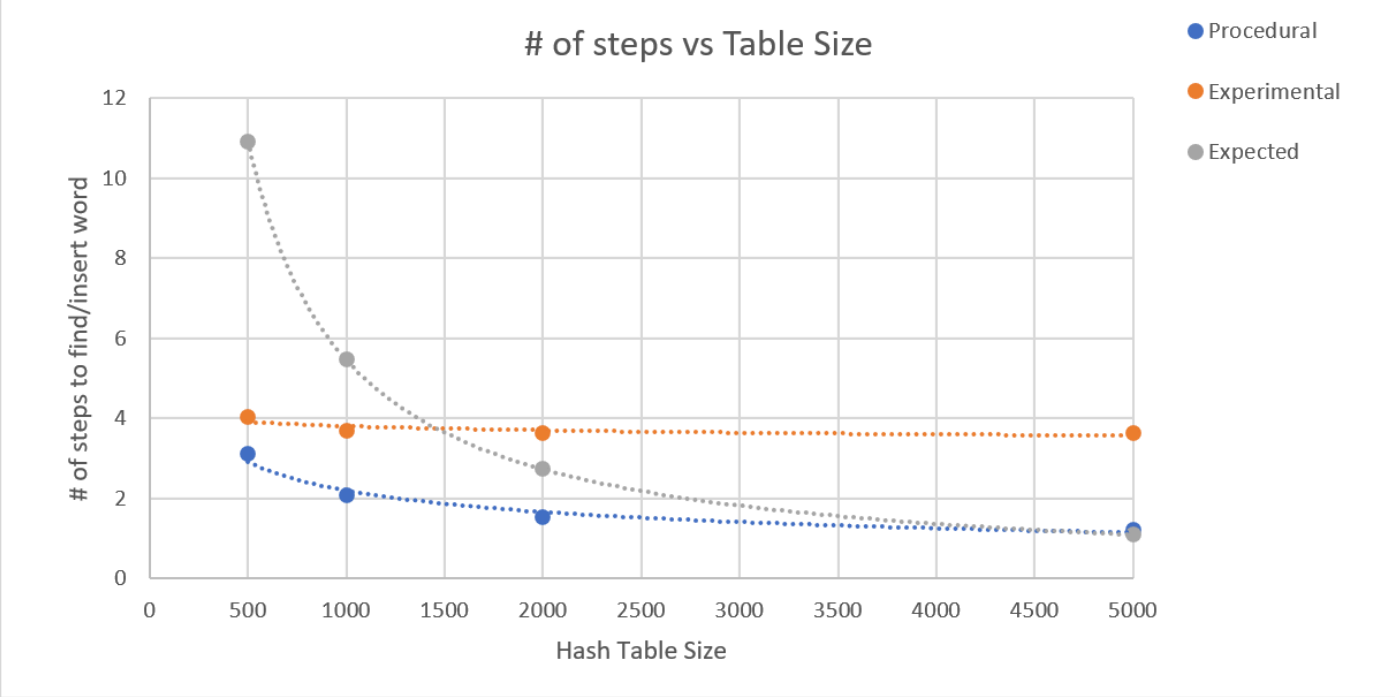
This project asks us to experiment with and analyze hash tables. This is done through hashing a small novel into the hash table via the word. Above is the raw data and results of these tests. The first 4 rows are the procedural experiment that is explicitly asked, and the final 5 are the my experiment to be discussed later. The total number of steps to find or insert a word is given in the second column. We can see that the actual average number of steps at a hash table size of 500 is 3.13, or 3 if we round to the full step. The expected is 10.933 at hash table size 500, so we can see that we are almost 3 times faster in actuality. As our hash table size gets larger, we can see that the difference in the two gets smaller, and then overlaps at a hash table size of 5000. This means that, given our inputted data, our hash function is extremely efficient with smaller hash table sizes, however is less efficient, but still efficient, as the hash table size grows. In the best possible circumstance, it would take 186609 steps to access all of the words. This is because there is that many total words that could be accessed to the O(1) time complexity. This is not feasible, especially in our implementation. We took 584204, 387270, 287902, and 228265 steps to access all words given the 500,1000, 2000, and 5000 sized hash tables respectively. All of this is in the data table above. The data is not what would be expected. Our expected results to find a given word on average is shown in the first 4 data points of column 3. We can see that we are much faster than that by 3 times, which decreases to being slightly slower, or equal to if we round, than the expected values. We are statistically slower than the expected number of steps. This is because each index of our hash table generally is shorter than the average size, with of course some being larger, and even maybe much larger in linked list length. Some other factors are that the more common words are generally used first, so they end up towards the front of the linked list of their respective hash table index, which can decrease the number of steps to find them. Our hash function is also pretty efficient too, our test will provide some comparison when we compare to a different hash function and look at that data later. Lastly, if we add words to the end of our linked list, such as maybe adding another novel to our hash table, it will function just as well. We can determine the order of the linked list by dividing the # of steps by the occurrences, This should give us a linear 1,2,3,4 and so on, then reset back to 1 on the next hash table index. We can continuously add whatever words we want to the linked list. It will always just keep appending them to the linked list. The only effect this has is pushing the new words added after the previous new word back in steps, making it less efficient to find

those later in the list. Of course we want our linked list as short as possible, so traversing it is shorter. The conclusion of this hash table procedure is that it is extremely efficient. We are really fast with small data, and still extremely efficient with bigger data. We would want the hash table to be bigger too, of course, which improves the efficiency by decreases number of nodes in our lists.

For our experiment we will be using a new hash function to hash the words. The results of it are the final 5 rows of the data table up above.

```
int NewHashCode(const string &str) {//experiment usage
    int h = 0;
    for (int i = 0; i < str.size(); i++){
        h += static_cast<int>(str[i]);
    }
    return abs(h % tableSize);
}
```

Above is the new hash function, which adds the ascII codes from each character in a word and then gets the modulus. This is a really cool method for words, which I will talk about soon. This new hash function starts off much quicker than expected, at just over 2.5 times as fast at a table size of 500. It performs at 4.034 steps to find a word on average, which is faster than the expected, which is 10.933. It is,however, slower than the provided hash function. We would statistically expect the two to perform equivalently, considering they are hashing the same words into the same size array, but our new hash function is slower. When we look at the graph below, we can then see that our new hash function seems to slow into an almost linear relationship. It performs equally well at the respective 2000, 5000, and 10,000 hash table sizes. The number of steps stays the same. The coolest part of this property is that when looking at the data printed out at a hash table size of 5000 and 10,000 in a CSV file, we can see that it essentially sorts the words into order by the size of each word, with exceptions. The main, and notable exception being numbers and words consisting of mostly special characters. This happens due to the fact that the ASCII codes for numbers and special characters are less than that of lower-case letters, which all fall within a certain range. This observed property unfortunately does have the side effect of the middle indexes of our hash table having significantly longer linked list lengths on average than the first and last indexes. This is the main factor as to why the average number of steps it takes becomes stable as the hash table size grows. This is a cool property that could have some very specific specialized uses. We were attempting to see if a simpler hash function could be more efficient, but instead we found a hash function that has a more linear fall off, that plateaus to 3.644 average steps per word insert or search as the hash table size grows, likely due to the math of the hash function itself. At the hash table size of 500, we can expect 1 whole additional step on average per word insert. This grows to 1.5 , 2.1, and 3.4 additional steps on average for the 1000, 2000, and 5000 hash table sizes respectively. Our new hash function is not nearly as efficient, but this cool property was interesting to see, and pretty easy to understand why it happens.




Given our data, we can see hashing is extremely efficient. Accessing the data, such as searching for it, and inserting data is extremely quick, especially compared to the usage of a singular linked list instead. We can access the hash table index in O(1) and traverse the linked lists in O(n) time complexity. This is extremely fast. If Amazon needed to Insert a new account, they can do so extremely fast, which is a necessity. If they needed to search for an account, for instance, for logging in, they can do so super fast. Hash tables ar what makes modern life possible. There are some factors that have drastic effect on the efficiency. The size of the hash table and the hash function in use are the main characteristics. If the hash table size is close to or larger than the size of the data within it, the data structure within it is smaller, so traversing, or searching, it takes less time. This is ideal. The hash function also plays an important role in efficiency too. We see that in the experiment and in the plot above. A hash function that equally distributes the data is optimal. Having one linked list extremely long, while the others are empty, makes traversing that full list take more time, which is not ideal. We want n to be small so that we can access it in the most optimal O(n) time. Below is the code, but I will also include the cpp file. The raw data, the data table, and plot will also be included for use if needed. Otherwise, it can be ignored.

```
#include <iostream>
#include <list>
#include <string>
#include <algorithm>
#include <fstream>
#include <cctype>
#include <cstdlib>
```

```
using namespace std;

struct Words{//struct of each word to be inputted.
    string word;
    int steps;
    int occurrences;

    Words(const string& w, int s){ // Constructor
        word = w;
        steps = s;
        occurrences = 1;
    }
};

class HashTable{
    static const int tableSize = 500;//hash table size
    list<Words> table[tableSize];

public:
int HashCode (const string &str){
    int h = 0;
    for (size_t i = 0; i < str.size(); ++i)
    h = h * 31 + static_cast<int>(str[i]);
    return abs(h % tableSize);
}
int NewHashCode(const string &str) {//experiment usage
    int h = 0;
    for (int i = 0; i < str.size(); i++){
        h += static_cast<int>(str[i]);
    }
    return abs(h % tableSize);
}
void Insert(string word){
    for (size_t i = 0; i < static_cast<long int>(word.length()); ++i){
        word[i] = std::tolower(static_cast<unsigned char>(word[i]));//make
words lowercase
    }//make all lowercase
    int index = HashCode(word);
    int positionInList = 0;
    for (auto& entry : table[index]) {
```

```
        if (entry.word == word) {
            entry.occurrences++; //word found!
            entry.steps += (positionInList + 1); //step counter
            return;
        }
        positionInList++;
    }

    table[index].insert(table[index].end(), Words(word, positionInList +
1));//word not found? append to linked list
    }

    void ToCSV(const string& filename) const{
        ofstream file(filename);
        if (!file) {
            cerr << "Error opening file!" << endl;
            return;
        }
        for (int i = 0; i < tableSize; ++i) {//loop through and add each
word to the CSV file
            for (const auto& entry : table[i]) {
                file << entry.word << "," << entry.steps << "," <<
entry.occurrences << "\n";
            }
        }
        file.close();
        cout << "CSV: " << filename << endl;
    }

};


int main(){
    HashTable myTable;
    ifstream
inputFile("C:\\Users\\Zayne\\projects\\LittleWomen.txt");//change as
needed, my environment required this

    if (!inputFile) {
        cerr << "Error opening file!" << endl;
```

```
        return 1; // error ==> exit
    }
    int wordCount = 0;
    string word;
    while (inputFile >> word) { //Reads words
        myTable.Insert(word); //Insert said words
        wordCount++;
    }

    inputFile.close();

    cout << wordCount;//print total number of words, useful for error
checking ==186609
    myTable.ToCSV("output500.CSV");
return 0;
}
```