Esau Hervert
eah170630

Assignment 6

CSE 5343
August 3, 2019

Problem

Implement hash table.

your hash table will consist of English language words - 8 characters or less. Start with a table size of 19 and insert words one at a time. When you have inserted 10 words, your program must give a message "load factor > .5". Then the program automatically increases the table size to double the current size and round up to the next higher prime number. Insert a few more words after that.

You may select a library hash function. But you must implement your own collision resolution scheme (quadratic probing).

Keep track of number of collisions for a given word. So if you try to insert a word and there is a collision then you select the next location based on quadratic probing. If this second location also gives a collision then you have 2 collisions and so on. Keep track of the highest collision for a given word and print it.

Run your program to insert words and then show a screen shot of its execution to

1. Find a word that has been added.
2. Try to find a word that is not in the table - your program should print a message saying it is not in the dictionary.

Approach

What I did is I considered a mapping from words of eight characters or less to the unit interval. Then I mapped the unit interval to the indexes of 1 to 19 (0 to 18 in the program). The map took two things into consideration:

1. Length of Word
2. Starting Character

I did not differentiate from upper case and lower case.

Code

Code 1: Hash_Functions.h

```
#ifndef Hash_Functions
#define Hash_Functions

#include <iostream>

// Since we are using words, we need this library.
#include <string>

// Since it is easier to change the size of vectors than arrays,
// I used vectors.
#include <vector>

// I used this to hold words while i change the size of the Hash Table.
#include <queue>

// For isblank.
#include <ctype.h>

using namespace std;

// This function will determine whether a number is prime or not.
bool Prime(int n) {
        int m = 2;
        while (m < n / 2) {
                if (n % m == 0) {
                        return false;
                }
                m++;
        }
        return true;
}

// Given a number n, this function will find the next highest prime
// number, if n is prime, then it will return n.
int Next_Prime(int n) {
        if (Prime(n)) {
                return n;
        }
        else return Next_Prime(n + 1);
}

// This function will take a string of at most 8 characters and
// assign a number between 0 and 1 based on two things:
// 1. The size of the string
// 2. The starting character
// Note that we do not differentiate between lower case and
```

```
// upper case.
double String_Value(string s) {
        double value = 0;

        // l is the raw size of the string.
        // tl is the true size of the string, which ommits blank spaces.
        int l = s.size();
        int tl = 0;
        int m = 0;
        for (int i = 0; i < l; i++) {
                if (!isblank(s[i])) {
                        tl++;
                }
        }

        // Assing the first letter of the string a value.
        // Note that I do not differentiate between lower
        // case and upper case.
        if (!isblank(s[0])) {
                m = (int)s[0];
                if (m > 64 && m < 91) {
                        m = m - 64;
                }
                else if (m > 96 && m < 123) {
                        m = m - 96;
                }
        }

        // Here we calculate the value of the word based on two things:
        // 1. Length of word
        // 2. Starting letter
        double dm = (double)m / 26;
        double dtl = (double)tl / 8;
        value = (dm + dtl)/2;
        return value;
}

// Here we will map the value of the word to an index from 0 to
// size - 1.
int Hash_Map(int size, double val) {
        double m = val * (double)size;
        return (int)m - 1;
}

// Here we have the Hash Class containing the Hash Table.
class Hash {
public:
        int size;
        double load;
        vector<string> Hash_Table;
```

```
        vector<int> Collisions;
        void Change_Size(int);
        void add(string);
        void find(string);
        void Print();
};

//This function changes the size of the Hash Table.
void Hash::Change_Size(int n) {
        Hash_Table.clear();
        Collisions.clear();
        size = n;
        for (int i = 0; i < size; i++) {
                Hash_Table.push_back(" ");
                Collisions.push_back(0);
        }
}

// This function will add words to the hash table.
void Hash::add(string s) {
        cout << "The Word added is: " << s << endl;
        cout << "Word Value is: " << String_Value(s) << endl;

        if (load > .5) {
                cout << "Load Factor -> .5" << endl << endl;
                int n = Next_Prime(Hash_Table.size() * 2);
                cout << "Changing Hash Table size to " << n << "."
                        <<endl << endl;

                // Save the current words in the Hash Table to a temporary
                // queue while we change the size of the Table.
                queue<string> temp;
                for (int i = 0; i < size; i++) {
                        if (String_Value(Hash_Table[i]) != 0) {
                                temp.push(Hash_Table[i]);
                        }
                }
                cout << "Now adding all the words back in including the "
                        << "new word: " << endl << endl;
                // Now lets add the new element to this temp vector.
                temp.push(s);
                Change_Size(n);
                load = 0;
                int sizetemp = temp.size();
                for (int i = 0; i < sizetemp; i++) {
                        add(temp.front());
                        temp.pop();
                }
        }
        else {
```

```
                // We calculae the position that the word should be in.
                double value = String_Value(s);
                int i = Hash_Map(size, value);
                string temp = Hash_Table[i];

                // If the position is empty, then we place it there.
                if (String_Value(temp) == 0) {
                        Hash_Table[i] = s;
                        load = load + 1 / ((double)size);
                        cout << "The Load is: " << load << endl;
                }
                // Otherwise we have collisions and must act accordingly.
                else
                {
                        Collisions[i]++;
                        int p = 1;
                        int I = (i + p) % size;
                        temp = Hash_Table[I];
                        while (String_Value(temp) != 0) {
                                Collisions[I]++;
                                p = p * 2;
                                I = (i + p) % size;
                                temp = Hash_Table[I];
                        }
                        Hash_Table[I] = s;
                        load = load + 1 / ((double)size);
                        cout << "The Load is: " << load << endl;
                }
        }

        cout << endl;
}

// This function will compare two strings to see if they are the same or not.
bool Compare(string a, string b) {
        int la = a.size();
        int lb = b.size();
        if (la != lb) {
                return false;
        }
        for (int i = 0; i < la; i++) {
                if (a[i] != b[i]) {
                        return false;
                }
        }
        return true;
}

// This function will take a string and see if it is in the Hash Table.
void Hash::find(string w) {
```

```
        cout << "Finding the word: " << w << endl;

        // Calculating the position of the word in the Hash Table.
        double val = String_Value(w);
        int i = Hash_Map(size, val);
        int I = 0;
        string s = Hash_Table[i];

        // If we find it immediatly, then we found it.
        if (Compare(w, s)) {
                cout << "The word was found at index " << i + 1 << endl << endl;
                return;
        }

        // If it is not in its correct spot but there are collisions in this
        // spot, then we will traverse through the collisions until we either
        // find the word or run out of collisions.
        if (Collisions[i] > 0) {
                int p = 1;
                I = (i + p) % size;
                s = Hash_Table[I];
                while (!Compare(w, s) && (Collisions[I] > 0)) {
                        p = p * 2;
                        I = (i + p) % size;
                        s = Hash_Table[I];
                }
        }

        // Once we go through all the collisions, we check and see if the
        // word is there.
        if (Compare(w, s)) {
                cout << "The word was found at index " << I + 1 << endl << endl;
                return;
        }
        else {
                cout << "The word was not found." << endl << endl;
                return;
        }
}

// For formatting the table.
void Word_Print(string s) {
        int l = s.size();
        for (int i = 0; i < 8; i++) {
                if (i < l) {
                        cout << s[i];
                }
                else {
                        cout << " ";
                }
```

```
        }
}

// This function will print the Hash Table.
void Hash::Print() {
        for (int i = 0; i < size; i++) {
                string w = Hash_Table[i];
                cout << i + 1 << ": ";
                Word_Print(w);
                cout << "                Collisions: " << Collisions[i] << endl;
        }
        cout << endl;
}
#endif Hash_Functions
```

Code 2: Assignment_6_Hash_Table.cpp

```
#include "Hash_Functions.h"

using namespace std;

int main() {
        // Initialise the Hash Table and make it to be size 19.
        Hash Words;
        Words.Change_Size(19);

        // Add the following words.
        Words.add("It");
        Words.add("Would");
        Words.add("Be");
        Words.add("Nice");
        Words.add("If");
        Words.add("You");
        Words.add("Give");
        Words.add("Me");
        Words.add("An");
        Words.add("A");

        // Print the current table.
        Words.Print();

        // Find the following words.
        Words.find("Nice");
        Words.find("Me");
        Words.find("Triangle");

        // Now let us add more words but here we pass the load threshold.
        Words.add("In");
        Words.add("This");
        Words.add("Class");

        // Print the final table.
        Words.Print();

        // Find these following words in the final table.
        Words.find("In");
        Words.find("Nice");
        Words.find("Me");
        Words.find("Banana");
}
```

Results

The following is the run I did.

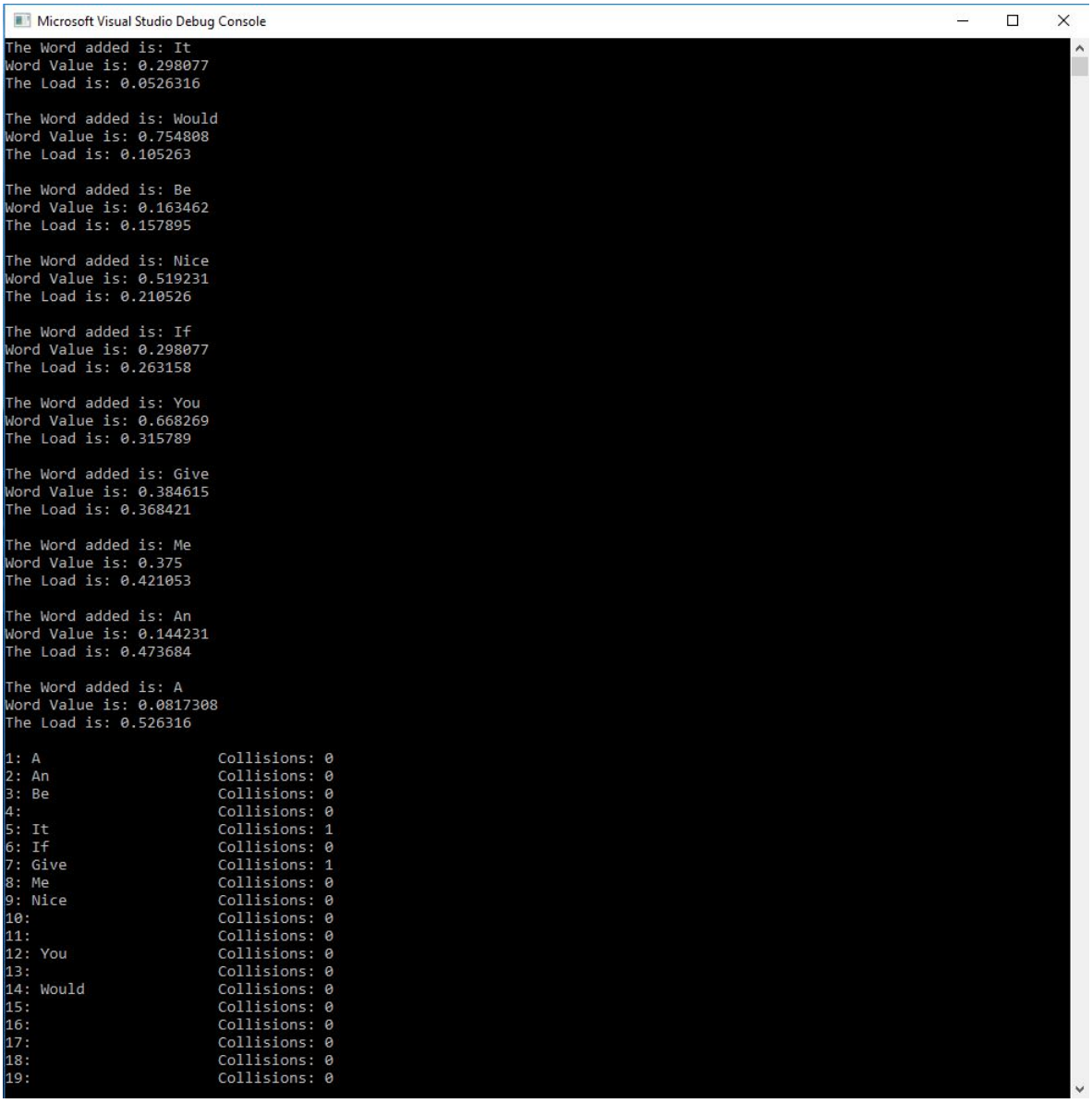

Figure 1: Results

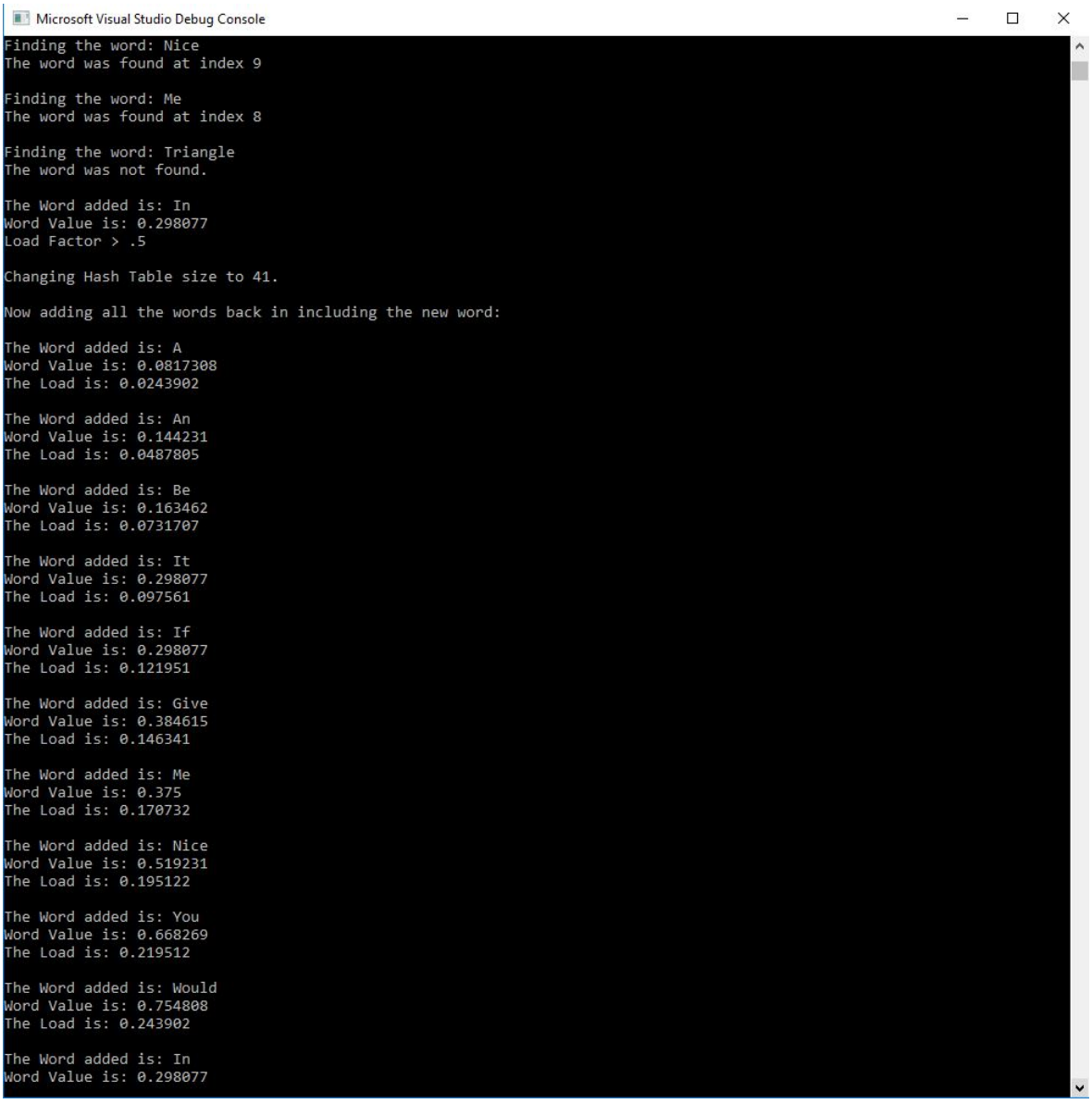

Figure 2: Results

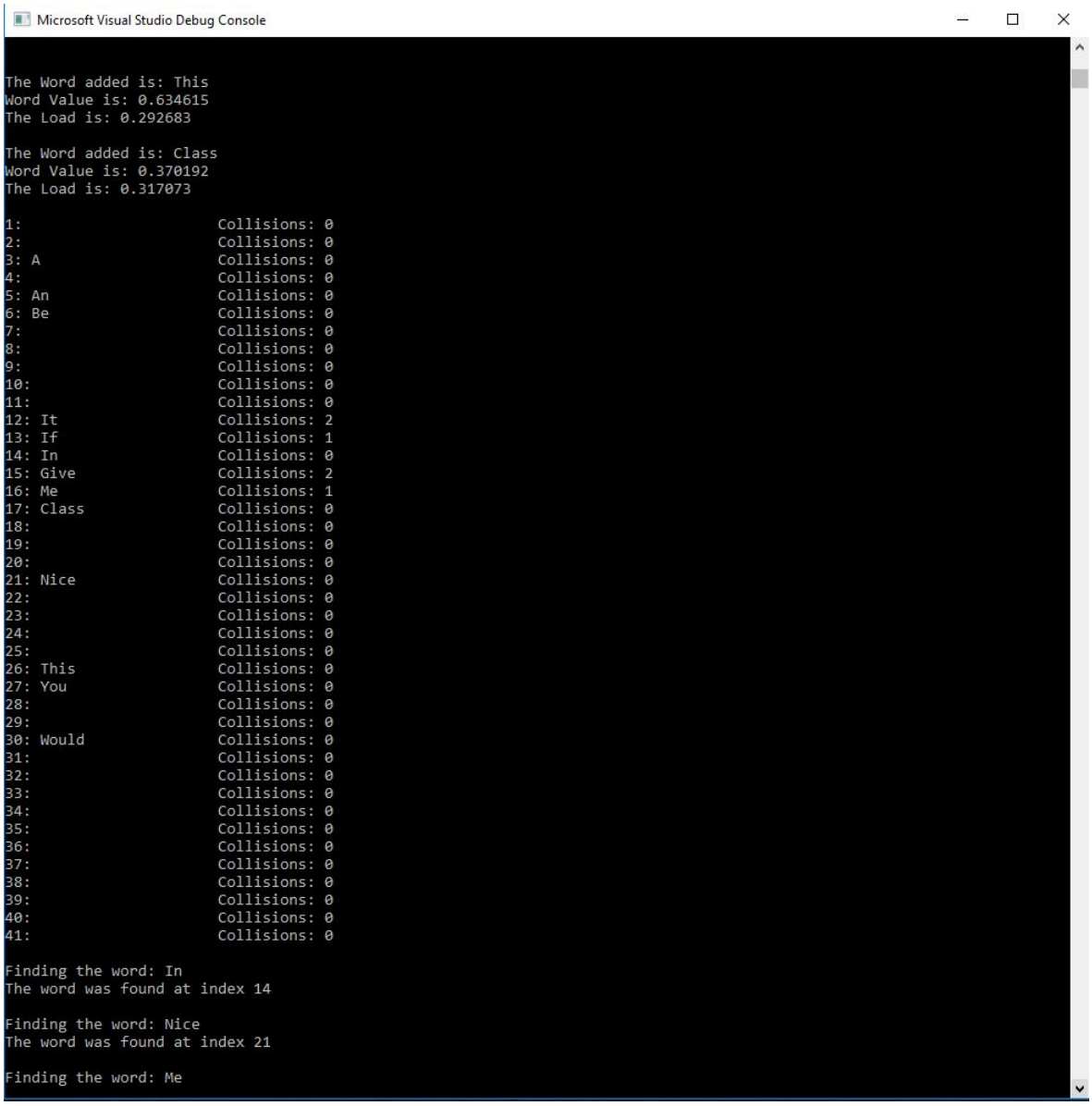

Figure 3: Results

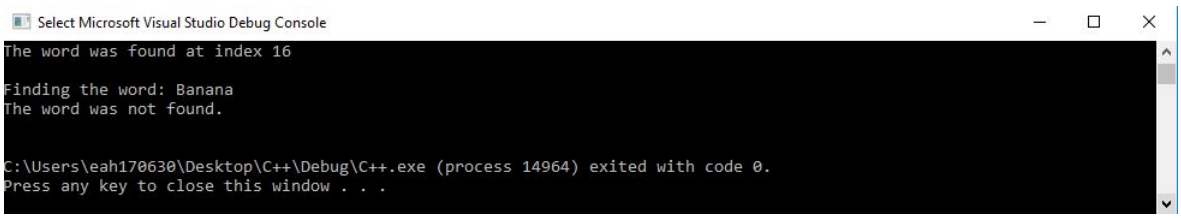

Figure 4: Results