Data Structures 2020
Prof. Talal Rahwan – Khalid Mengal
By: Yahia Beethoven Tayel
5th of December 2020

Assignment 3: Text Analyzer System

- In this assignment, I built a text analyzer system which is basically a program that draws from the classes HashNode and HashTable (as shown in the source code). The main function calls on methods of the HashTable which is just an array of linked lists of type HashNode based on the input of the user.
- My default program is the one labeled “assignment3a.cpp”.
- In order for us to be able to deal with the input properly whatever its data type is (string in this case), we have to break it down into k pieces (numbers we can deal with) where k is less than or equal to 32 bits.
    - Since the string is made up of characters and each character takes up 1 byte (8 bits), by casting it into an integer (its ASCII code) we basically fill up the remaining 24 bits with zeros to obtain a 32-bit number. The number is unsigned (always positive) for ease when converting the code to an index of the array.
- The hash code of a single string element is simply the accumulation of each of its individual characters’ integer values that are calculated in the hash function.
- There are numerous ways for us to calculate that hash code and the 3 methods which I chose for this assignment are:
    - Polynomial Hash Code
    - Summation Hash Code
    - Cycle-Shift Hash Code
- For my compression function, I used the modulus division function, where I divide the hash code by the modulus of the capacity of my hash table (n) to squeeze the code within the range of 0 to n-1 so it can fit inside the table.
- The table is just a simple array of linked lists of capacity given in the argument of its constructor. In each cell, there is an STL linked list with nodes of type HashNode which I defined myself to contain a key (string), and a frequency (integer).

Data Structures 2020
Prof. Talal Rahwan – Khalid Mengal
By: Yahia Beethoven Tayel
5th of December 2020

- The capacity of the table is determined by multiplying the total number of words initially in the file ("The_Alchemist.txt" in this case, which is about 39,083 words) by 1.333, and choosing the next prime number greater than the product as the capacity (which was 52,121), which has proven to minimize collisions.
- After building the table of empty lists, all we have to do is insert nodes into the lists in the indices specified for each string by its compression function for the hash code. If a word was already existing, we just increase the frequency of the existing node by 1 and return. Upper cases are ignored, I insert the words as lower case initially and change the input of find_freq() to lower case as well. I also ignore all special characters in find_freq() of the entry of the user.
- The program can be exited at any point (even before loading the file) using the keyword 'exit'.

Data Structures 2020
Prof. Talal Rahwan – Khalid Mengal
By: Yahia Beethoven Tayel
5th of December 2020

In order to effectively count the number of unique words in the table:

- I realized that for any individual string being inserted, it is surely one of two scenarios, either the first word to be inserted in the table, or it is not being inserted for the first time.
- So, instead of iterating through all the cells, then through the nodes inside the list of each cell and have a counter incrementing for each node with frequency 1 to know the number of unique words, I decided to calculate the number of unique words using the converse of the logic:
    - Unique words = total words – repeated words.
- I defined an integer attribute of the class HashTable called wordRepeated.
- So, for every time we insert a word, we iterate through the list at the index given by the compression function of the hash code (since upper case is ignored, any word consisting of exactly the same letters in the same order will definitely end up with the same hash code). If the word indeed existed inside the list (ie. The entry is not the first of its kind), then we increment the variable wordRepeated and the total words counter, then we exit the insertion function, otherwise, we do nothing and just keep iterating.
- Every time a word is inserted, and another same word existed, the counter increases, and by the end we can subtract wordRepeated from the total number of words to get the remaining unique words which never entered the if-condition that another same word existed inside the list.

Data Structures 2020
Prof. Talal Rahwan – Khalid Mengal
By: Yahia Beethoven Tayel
5th of December 2020

Algorithm insert(k):

Input: A key k

Output: nothing

```
index = hashCode(k) % capacity                    //keep hash code within range of size of HashTable
for each position p ∈[L[index].begin(), L[index].end()] do
if p->key = k then                                //if word already exists in list at index
        p->frequency = p->frequency + 1
        wordRepeated = wordRepeated + 1
        totalWords = totalWords + 1
        return  //exit function after incrementing word's frequency, variable of repeated words, and total words
if L[index] is not empty then                     //if for loop was not just exited because list was empty
        collisions = collisions +1
L[index].push_back(HashNode(key, 1))              //add node since word doesn't exist
totalWords = totalWords + 1                       //make sure total words' incremented with every insertion
```

- Line 5 is how we exploited the HashTable in order to effectively count the number of unique words in the entire text.

Pseudo code of the count_unique_words()method:

Algorithm count_unique_words():

Input: nothing

Output: number of unique words in the entire text

return totalWords – wordRepeated

- So the actual method of unique words takes O(1) time and exploits the HashTable efficiently.

Data Structures 2020
Prof. Talal Rahwan – Khalid Mengal
By: Yahia Beethoven Tayel
5th of December 2020

1. For the **polynomial hash function**, I simply looked up the best numbers which when inserted into the polynomial hash code minimize collisions and they were 31, 41, 37, 39, 13, and others. After trying them all out, 41 worked best for my function with only 99 collisions for "The_Alchemist.txt" file which contains initially 39083 words and which I reduced to 38,868 by removing all special characters and sentence delimiters.

- In very simple terms, the polynomial hash function works by dealing with each character individually and storing the length of the string - 1 as the order of the polynomial (n). Going in ascending order, each character's ASCII code (which is obtained by casting the char into an unsigned int) is multiplied by 41 to the power of n. This multiplication is done to take into account the order of each character, so that "ab" is not equal to "ba", for example, to minimize collisions. The product is accumulated to the hash code of the string for each character. Then n is decremented for the next character in turn and so on until all of the characters are dealt with and are accumulated to the code.

Pseudo code of the hash function:

```
Algorithm HashCodePolynomial(k):
    Input: A key k
    Output: The hash code of the matching entry, or 0 if k is an empty string
    order = k.length – 1          //the order of the polynomial is 1 less than the length of string
    code = 0
    if k.length = 0 then          //if empty string, return 0
        return 0
    For each character i in [0, k.length] do
        code += ASCII code of k[i] * 41^order     //accumulate product of ASCII by 41 power order
        order = order -1
    return code
```

- After using this function in my text analyzer system, it created only: **99 collisions** (where I consider a collision only when a different key gets the same index as another existing entry).

2. For the **summation hash function**, I just performed a very simplified version of the polynomial hash function where it is as if the number 'a' which was 41 in the previous method is instead 1, so it's useless. Elaborately:
    - The summation hash function works by dealing with each character individually. Going in ascending order, each character's ASCII code (which is obtained by casting the char into an unsigned int) is accumulated to the hash code of the string without further multiplication.

Pseudo code of the hash function:

```
Algorithm HashCodeSummation(k):
    Input: A key k
    Output: The hash code of the matching entry, or 0 if k is an empty string
    if k.length = 0 then                          //if empty string, return 0
        return 0
    For each character i in [0, k.length] do
        code += ASCII code of k[i]                //accumulate ASCII codes of each character
    return code
```

- After using this function in my text analyzer system, it created: **2,615 collisions** (where I consider a collision only when a different key gets the same index as another existing entry).

3. For the **cycle-shift hash function**, I used the summation function and modified it. I divide the entry string into characters, and I cast them into integers (their ASCII codes), but instead of just adding them all up, I do some shifting of bits.
    - In very simple terms, the function works by dealing with each character. Going in ascending order, I add two characters' ASCII codes (which is obtained by casting the char into an unsigned int), then I shift the sum by a certain number of bits, then I add the following character's code to the sum and shift more bits, and this continues until all characters are dealt with.
    - The bit shifting I used is one that was proven using experiments to minimize collisions: cyclic 5-bit shift. This means that the 5 shifted bits wrap around the 32 bits. Using the bitwise '|' operator, we obtain a shifted number after each addition that makes each entry more unique and hence minimizes collisions.

Pseudo code of the hash function:

```
Algorithm HashCodeCycleShift(k):
    Input: A key k
    Output: The hash code of the matching entry, or 0 if k is an empty string
    if length = 0 then                          //if empty string, return 0
        return 0
    code = 0
    For each character i in [0, k.length] do
        code = (code<<5) | (code>>27)           //cyclic 5-bit shift, wrapping around 32 bits
        code += ASCII code of k[i]
    return code
```

- After using this function in my text analyzer system, it created: **119 collisions** (where I consider a collision only when a different key gets the same index as another existing entry).

Data Structures 2020
Prof. Talal Rahwan – Khalid Mengal
By: Yahia Beethoven Tayel
5th of December 2020

Conclusion:

As can be seen from the results of each hash function, the summation function is quite inefficient and cause numerous collisions that do not exploit the hash table effectively. This is because it does not account for the difference in position of each character, so any words with the same exact letters even if in different order combinations would have the same hash code. But the polynomial and cyclic functions took order into account so performed much better. They are quite close to each other when it comes to collisions but the polynomial was better with a slight a difference. This is why the polynomial hash function was chosen to be my default function for the text analyzer program.

In all of the functions, the total recognizable number of words is 38,868, of which there are 3,321 unique words which comprise about 8.54% of the total.