CSCI 2270 – Data Structures - Section 100

Instructors: Shayon Gupta, Ashutosh Trivedi, Maciej Zagrodzki

Assignment 7 - Hash Tables

OBJECTIVES

1. **Use a hash function**
2. **Store data in a chained hash table (Separate Chaining)**
3. **Search for data in a hash table**

Overview

This assignment will recreate assignment 2, but using hash tables instead of arrays. You are welcome to use assignment 2 code wherever possible. There are 3 files on moodle

1. HarryPotter.txt - contains text to be read and analysed
2. ignoreWords.txt - 50 most common words in english language
3. HashTable.hpp - header file

You must implement the functions declared in the header file on Moodle: HashTable.hpp. **Do not** modify this header file. You will also need to write a main function. We will assume your main function is written in a separate file while autograding. If you would like to write all of your code in one file, you will have to split it up when submitting it.

Your program must take **4** command-line arguments in the following order

1. The number of most common words to print out
2. The name of the text file to process
3. The name of the stop words file
4. The size of your hash table

For example:

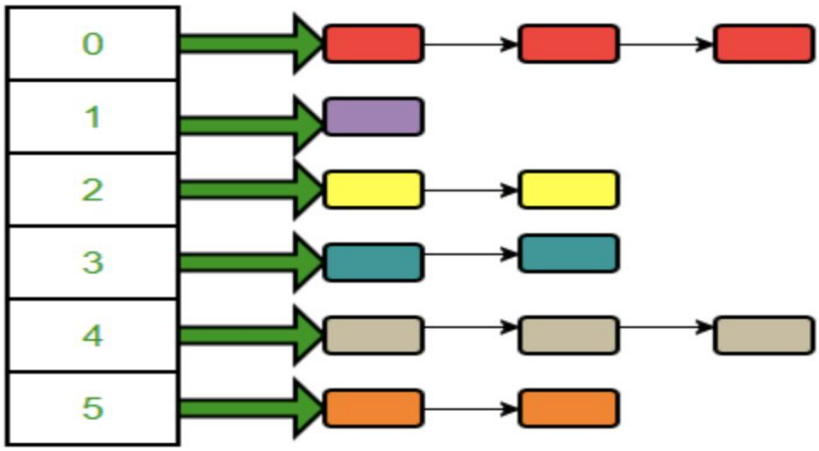
```
./Assignment7 15 HarryPotter.txt ignoreWords.txt 500
```

Your program should do the following:

- Read in a list of *stop words* from the file ***(50 stop words***) specified by the *third* command line argument ***(ignoreWords.txt)***. Store the stopwords in a ***stopwords hash table.*** Use the hash function detailed below in the description of the **getHash function**

- Build another hash table of size N, to store all the unique words, where N is the *fourth* command line argument. Use the hash function detailed below in the description of the ***getHash*** function

- Your hash table should be stored in the private ***hashTable*** variable, which should be a dynamically allocated array of pointers to ***wordItem*** structs. Each of those *wordItem* structs stores a word and how many times that words has appeared, as well as a pointer to the next *wordItem* struct with the same hash, in the event of a hash collision. A diagram of the hash table layout is shown below:



- Read in every word from the file specified by the *second* command line argument ***(HarryPotter.txt)***. Store all *non-stop words* in this hash table. Do not store the same word multiple times - instead, each word is stored with a *count* variable that indicates how many times it appears
  - If the word is already present in the hash table, then increment the count by calling ***incrementCount*** method
  - Else, call ***addWord*** method described below to add the word to the hashtable

- Print out the top N words, where N is the *first* command line argument, along with some other information (detailed below)

For example running your program with the below command

```
./Assignment7 15 HarryPotter.txt ignoreWords.txt 500
```

will give the below output:

```
0.0241 - harry
0.0236 - was
```

```
0.0158 - said
0.0139 - had
0.0100 - him
0.0081 - ron
0.0068 - were
0.0067 - hagrid
0.0065 - them
0.0052 - back
0.0051 - hermione
0.0047 - its
0.0044 - into
0.0042 - been
0.0040 - off
#
Number of collisions: 5485
#
Unique non-stop words: 5985
#
Total non-stop words: 50331
```

HashTable Class

HashTable(int hashTableSize)

- ➔ Parameterized constructor: using the class variable **hashTable**, allocate a dynamic array of **wordItem** pointers with size **hashTableSize**. *Hint: you may want to initialize all the pointers in the array to* **nullptr**. Initialize all other class variables to default values.

~HashTable()

- ➔ Deallocate all memory that was dynamically allocated

void addWord(std::string word)

- ➔ Inserts word into the hash table with a count of 1. If the word collides with other existing words in the hash table then, add the new word to the beginning of the linked list chain and update the **numCollisions** class variable accordingly

bool isInTable(std::string word)

- ➔ Using your **searchTable** function, search for the **wordItem** containing **word**. Return true if it is found, otherwise return false.

void incrementCount(std::string word);

➔ Using your **searchTable** function, search for the **wordItem** containing **word**. Then increment its count field.

void printTopN(int n);

➔ Print the top **n** most frequent words in descending order of frequency and the probability of occurrence **(upto 4 decimal places)** for each word. Use the following format:

```
/* for each wordItem, w, in the top n most frequent words
   totalNumberofWords - total number of non-stop words */

cout << (float)w->count/totalNumberofWords << " - " << w->word <<
endl;
```

int getNumCollisions();

➔ Return the class variable **numCollisions**.

int getNumItems();

➔ Returns the number of unique words in the hash table. Return the class variable **numItems** which needs to be updated every time a unique word gets added to the hash table.

int getTotalNumWords();

➔ Return the sum of **count**'s for every **wordItem** in the hash table.

unsigned int getHash(std::string word);

➔ We will be using DJB2 function to compute the hash value for each **word**. The complete code is provided for you:

```
unsigned int hashValue = 5381;
int length = word.length();
for (int i=0;i<length;i++)
{
    hashValue=((hashValue<<5)+hashValue) + word[i];
}
hashValue %= hashTableSize;

return hashValue;
```

wordItem* searchTable(std::string word)

➔ Search the hash table for the **wordItem** containing **word** and return a pointer pointing to it

CSCI 2270 – Data Structures - Section 100

Instructors: Shayon Gupta, Ashutosh Trivedi, Maciej Zagrodzki

Submitting your code:

Log onto Moodle and go to the Assignment 7 Submit link. It's set up in the quiz format. Follow the instructions on each question to submit all or parts of each assignment question.