Word Games

Implementing a Minimal Perfect Hash Function

Project Due: 06/18/20 @ 2:30pm

1.0 Objective

This project is meant to help you develop skills at working with hash tables and writing entire programs from scratch. In this project you will only be provided with a project description and list of program requirements. No code will be provided at all. This means you will need to learn to design and implement an entire program on your own.

2.0 Project Description

2.1 Basic Program

For this project you will design a program that searches a text file (**HW#1-MLKSpeech.txt**) looking for key words, keeps statistics on the results of the search, and print out these statistics.

The first part of this assignment is for you to use the contents of one data file to create a hash table. The second part of this assignment is for you to open a second data file and compare all of the words in that file to your hash table using a hash table look up function. When you have done this, you will then print out statistics showing how many of the words in your hash table matched words that were in the second data file.

To make this work, you will need to read all the key words from a file (**HW1-MLKSpeechP1.txt**), build a minimal perfect hash table, and then start reading the contents of a second text file and look for these key words. The following sections examine each of these steps.

2.2 Reading from a File

To read from a file you must first use the Python *infile1 = open(filename,"r")* command. At this point you can start reading from the file using *variable = infile1.readline()*. Do be careful, if your program can't find the file that it is looking for it will crash.

2.3 The Hash Function

For this project you will be using *Cichelli's Method* to build a minimal perfect hash table. How to build this table is discussed in the next section. Here we describe the hash function itself.

h(word) = (length(word) + g*(firstletter(word)) + g*(lastletter(word))) % size

Note that there is a very good video on YouTube that shows how the Cichelli Method works. You can view it here:
https://www.youtube.com/watch?v=naKfUA42llw

For this implementation of Cichelli's Method, we'll be using g=11 (see the video for an explanation).

The following table is a description of each term in this equation:

Term	Description
h(word)	This will be the index into the hash table for the word.
length(word)	The number of characters in the word.
g	g is the value associated with a given letter. The value ranges between 0 and some maximum value. The max value is one of the inputs to the *Cichelli Algorithm*. For this homework, the maximum value of g will be 50% of the number of unique words that will be placed in your hash table = 11.
firstletter(word)	The character that is in the first position of the word.
lastletter(word)	The character that is at the end of the word.
size	The total number of key words. It is also the number of elements in the hash table.

To compute the hash function for a word a number of data structures are going to be required. Obviously, you must have the array that represents the hash table. This array will be filled with the actual key words. Each key word should appear only once in the hash table. Secondly, you will need an array to store the *g value* of each letter that appears in the beginning or end of a word.

2.4 Building a Minimal Perfect Hash Table

This is one of the major components of your project. You will need to open the key word file provided by the user and read each of the words into a list. Once you have them all read, you can start to build your hash table and define your hash function. You will accomplish this by using the *Cichelli Method*. The list that all the key words were initially read into will be the initial word list for the algorithm.

2.5 Counting Key Words

Once this minimal perfect hash table is constructed, you are ready to begin reading the text file and counting key words. This should be a fairly simple process. You will read a line from the text file, break the line into tokens (one token for each word in the line), and then examine each

token. To examine a token, simply pass it through the hash function discussed in Section 2.3 and see if the word is currently in the hash table.

A couple of things to watch for. First of all, you should make all of your comparisons case insensitive. In fact, I suggest that wherever you store all of your strings in all lower case (use the Python *.lower* function). This will make your life a bit simpler.

2.6 Statistics

Another major part of this project is recording statistics. A summary of all the statistics you must keep are presented in the following table:

Statistic	Description
lines	This is the total number of lines read by your tester program. Blank lines should not be counted in this total. Only lines that actually have at least one word on them should be counted.
words	This is the total number of words checked (key words plus non- key words). A word does not have to be in the dictionary. A word is considered to be any string of consecutive characters with no white space in between them. In other words both "hello" and "sadflk" are considered to be words by this program.
keyWords	This is the total number of key words found in the text. Words should only be counted if they are an exact match to the keyword given in the file: "the" should not match to "then".

Time	You need to record the total time of your program. This time should include the time to read the key words, create the hash table, read the text file, and record all the statistics. It is suggested that you start a timer as one of the first things you do in the program and stop it immediately before printing out the statistics. The following is a little sample code that shows how you can time events.
	
	`import time`
	
	`start = time.time()`
	`...`
	`end = time.time()`
	`print(end - start)`

When the program is finished reading the text file, it should print out a list of statistics. The output should look exactly like the following:

```
**********************
***** Statistics *****
**********************
Total Lines Read: xxx
Total Words Read: xxx
Break Down by Key Word
     xx : key1
     xx : key2
     xx : key3
       .
       .
       .
Total Key Words:  xxx
Total Time of Program: xxxxx
```

The x's should be replaced with actual numbers and key1,2,3,... should be replaced with the actual key words. The number following the key word is to indicate how many times that key word appeared in the analyzed text.

3.0 Program Design Tips

The number one rule about writing a program from scratch is not to write a single line of code until you have developed a good design. Developing this good design is probably the hardest thing to learn how to do in programming. Here are a few of my suggestions on how you should go about designing and writing this project.

1. Read the program through once to get an idea of what is expected. Then, read it again. The second time through you may want to take some notes. At this point, you should have a thorough understanding of what is expected of you. If you need to read it a third time, do so.

2. Do a very rough design. This basically means figuring out what you are going to want to create and what the purpose of each part of the program will be. For this project, I can think of at least three parts you should build. A part that contains your main program. A function for the Cichelli hash table. And another function to hold, calculate, and print the statistics. You may also want additional parts for parsing files or for some other kind of work. It is completely up to you; however, be sure you know why you need each part and

what its purpose will be for.

3. Once you have a list of your parts, go through each part and determine what functions it will need and what each of these functions will do. This should include information about what parameters each function will take and what type of value it will return.

4. Then go through each function and write psuedo-code describing how the function will flow. This is a very critical step because if it is done properly, your code writing will be much simpler. A good piece of advice at this stage is when you are examining a function, assume all the other functions already work - even if you have not written the pseudo-code for a function. If you did Step 3 properly, you will know how each function should work, what arguments it will take, and what values it will return.

5. The very last step is to go ahead and write your project up. Do not write all the code at once. Write functions one or two at a time, test them and make sure they work as you expect and then go on to the next function or two. It is always best to start with the lowest level functions first and work your way up. In other words, if you have some function that does not call any other of your functions, then that is the one you want to start with. You can then test it and make sure it works (this often requires a little creativity). Once you have all of these functions written, you can then move up the chain to functions that only call these lowest level functions. Once you finish all of these, move up another level. Continue to do this until all of your functions are written.

One of the things you will notice about this design is that Step 4 calls for a top-down approach (describe all of the higher level functions assuming the lower level ones are done and then move down to the next level). Whereas, Step 5 calls for a bottom-up approach (write and test all of the low level functions before moving on to the higher level functions). I think if you follow this approach, you should find your programs (for this or any other class) get written much more quickly and with far fewer bugs.

4.0 Handing in Your Project

Be sure to comment your code well so that it can be easily browsed to see what is going on. Code that is excessively difficult to read will be penalized.

Turn in a single file called "**COP 4530 – HW#1.py**" to the assignment on Canvas.

7.0 Grading

The project is due at 2:30pm on June 18, 2020. The project will be graded out of 100 points and be graded on the following criteria:

- Non-trivial program compilation
- Correct Output
- Performance (running time)

We will grade your code by running it against a series of test files and checking the output of the program against a known, correct implementation. Any differences in your output versus that of the correct implementation will result in a point deduction. Without a doubt, much more in depth tests will be run than the initial one provided. This means you should write some of your own test files and check for any errors.

We will also examine parts of your code to make sure that you are implementing the data structures as required. If you are getting the correct results but not implementing the proper data structure, your score will suffer severely. You must implement the proper data structures – it is a data structures course after all!