CS 302 – Assignment #07

Purpose: Learn concepts regarding hash tables.
Due: Tuesday (3/26) → Must be submitted on-line before class.
Points: Part A → 75 pts, Part B → 50 pts

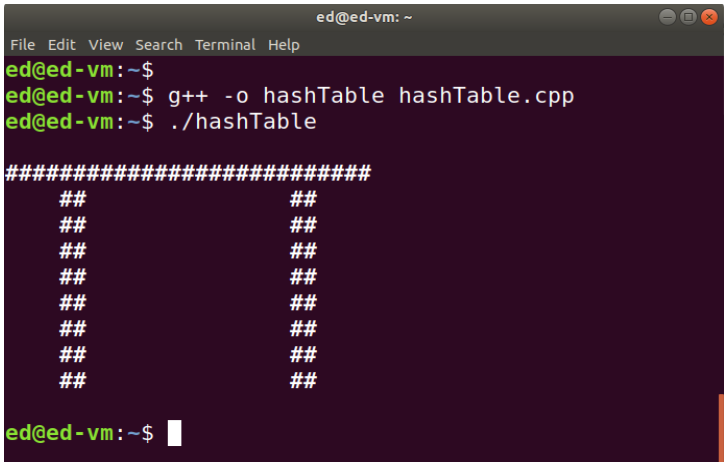
Assignment:

Part A:

Design and implement a C++ class, ***wordHash***, to implement a ***hash table***[1] data structure. The ***wordHash*** class will implement a hash table for storing and counting words for a concordance. A main will be provided that performs a series of tests. Refer to the UML descriptions for implementation details.

Part B:

Create and submit a brief write-up including the following:



- Name, Assignment, Section.
- Summary of the ***hash table*** data structure.
- Compare the hash table data structure to using a balanced binary search tree. Include the associated trade-offs.
- Explain why the print word count results were not in alphabetical order. How could they be displayed in alphabetical order?
- For the hashing, explain what occurs when the load factor is reached (which may occur multiple times). Note why the initial table size was set to a prime. Explain why the hash function used is good or poor for this application.
- Big-O for the various hash operations (insert(), incCount(), rehash(), getMaxWord()).
- Time the program using the words7.txt test file (note the hash statistics). Change the hash function to a simple hash the adds each character of the word modulo the table size. Time the program using this new hash. Explain the results in terms of the statistical data and the difference in execution time (by percentage).

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 10:00 AM (before class).
- Submit a copy of the write-up (PDF format).

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

1 For more information, refer to: http://en.wikipedia.org/wiki/Hash_table

Make File:
You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executable.

Class Descriptions

- Word Hash Class
  The word hash class will implement functions specified below.

wordHash
-hashSize: unsigned int
-reSizeCount: unsigned int
-collisionCount: unsigned int
-uniqueWordCount: unsigned int
-*wordList: string
-*wordCounts: unsigned int
-loadFactor=0.65: static const double
-initialHashSize=30011: static const int
+wordHash()
+~wordHash()
+insert(string): void
+getUniqueWordCount() const: unsigned int
+getWordCount(string) const: unsigned int
+incCount(string): bool
+getMaxNode(unsigned int &, string &) const: void
+printHash() const: void
+showHashStats() const: void
-insert(string, unsigned int): void
-hash(string) const: unsigned int
-next(unsigned int, unsigned int) const: unsigned int
-rehash(): void

Function Descriptions

- The *wordHash()* constructor should initialize the hash table to an empty state.
- The *~wordHash()* destructor should recover all dynamically allocated memory.
- The *insert()* function should increment the word count if the item is already in the hash table and if not, use the private function to insert the passed item into the hash table.
- The private *insert()* function should insert the passed item into the hash table. If the hash table entries exceeds the load factor (count/tableSize), the table must be rehashed via the private *rehash()* function. The *hash()* function must be used determine the table location. If a collision occurs, the collision count should be incremented and the next() function should be called to implement quadratic probing. The appropriate counts should be updated accordingly which may be called repeatedly. The collision count should be incremented appropriately.

- The *getUniqueWordCount()* function should return the current total unique word count.
- The *getWordCount()* function should return the current word count for the passed word.
- The *incCount()* function should search the hash table for the passed string and, if found, increment the word count associated with that string and return true. If the passed string is not found, the function should return false.
- The private *next()* function should be called when there is a collision and should implement a quadratic probing approach.
- The *hash()* function should return a hash from the passed string. The hash should be created with the Jenkins One-At-A-Time Hash function[2]. The final returned hash must be mod'ed with the current hash size.
- The *getMaxNode()* function should search for the word string associated with the largest count and return, via reference, the word string and count.
- The *rehash()* function should create a new hash table approximately twice the size of the existing hash table (based on the provided table below), extract all entries from the current hash table, insert them into the new table, and delete the old hash table. The hash table resize counter should be updated. The entries should be placed with the private *insert()* function into the new hash table (which may place them at a different location). The next largest hash size should be obtained from the below table.

```
static      const unsigned int      hashSizes[12]=
      {30011, 60013, 120017, 240089, 480043, 960017, 1920013,
      3840037, 7680103, 30720299, 15360161, 61440629};
```

  If the the hash size needs to exceed the last entry and error message should be displayed.
- The *printHash()* function should print all non-empty entries in the hash table.
- The *showHashStats()* function is a utility function to print the current hash size, current hash table resize count, and the collision count.

- Word Count Class

  The word count class will implement functions specified below.

WordCount: public wordHash
-totalWordCount: unsigned int
-docFileName: string
+wordCount()
+readDocument(const string): bool
+showDocumentFileName() const: void
+showTotalWordCount() const: void
+showUniqueWordCount() const: void
+showStats() const: void
+getMaxWord(unsigned int &, string &): void
+printWordCounts() const: void
+getWordCount(string) const: unsigned int

2 For more information, refer to: http://en.wikipedia.org/wiki/Jenkins_hash_function

Function Descriptions

- The *wordCount()* constructor should perform applicable initializations.
- The *readDocument()* function should attempt to open the passed file name, and if successful read the document one word. Each word should be added to to the hash table which will create new entries for unique words and update the count for duplicate words. The function should remove any brackets, numbers, or punctuation from the words (e.g., '(', '{', '[', ')', '}', ']', '!', '?', '.', ':', '$', '#', etc.). Additionally, numbers should be filtered out. For example, '**`Hello!`**' would be stored as '**`hello`**'. All words should be converted to lower case. This will count '**`Hello`**' and '**`hello`**' as the same word. Additionally, the word '**`1234`**', '**`p@asw0rd`**', '**`re-did`**', and '**`count12er`**' should be ignored (i.e. not stored in the tree). The function should update the total word count (including duplicate words). See the sample output for examples. If the open and read are successful, the function should return true and false otherwise. The function should close the file.
- The *showDocumentFileName()* function should display the formatted document file name. See the sample output for formatting.
- The *showTotalWordCount()* function should display the formatted total word count. See the sample output for formatting.
- The *showUniqueWordCount()* function should display the formatted unique word count by calling using the word hash *countNodes()* function. See the sample output for formatting.
- The function *showStats()* should call the *showHashStats()* function.
- The *getMaxWord()* function should return the node with the largest word count, word count and word, by reference.
- The *printWordCounts()* function should call the print hash function.
- The *getWordCount()* function should return the count of the passed word (0 if not found) by calling the base class function. *Note*, since the words are all lower case, must search for only lower case words.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Example Execution:

Below is an example output for the test script and a program execution for the main.

```
ed-vm%
ed-vm% ./concordance -i words1.txt
------------------------------------------------------------
CS 302 - Assignment #7

Document Information:
---------------------
Document File Name: words1.txt

Total Word Count: 22496
Unique Word Count: 1777
Hash Stats
   Current Hash Size: 9973
   Hash Resize Operations: 0
   Hash Collisions: 287

Most Frequent Word is: 'the' occurring 2228 times.

------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
ed-vm%
```

```
ed-vm%
ed-vm%
ed-vm% ./concordance -i words5.txt
------------------------------------------------------------
CS 302 - Assignment #7

Document Information:
---------------------
Document File Name: words5.txt

Total Word Count: 262140
Unique Word Count: 2
Hash Stats
   Current Hash Size: 9973
   Hash Resize Operations: 0
   Hash Collisions: 0

Most Frequent Word is: 'number' occurring 131070 times.

------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
ed-vm%
ed-vm%
ed-vm%
```