COSC-160-01
Fall 2018
Project 3 Description

Hashtable

1. Summary

   You will design and implement a hashtable structure. The application of the structure will include spell checking and word counting. Words in an input text file (**inputFile**) will be sequentially investigated and compared to a comprehensive list of correctly spelled words – a lexicon (**dictionaryFile**). Your program will store the lexicon in a hash table structure to provide for an efficient search time. Major factors to consider for the design should be theoretical efficiency (both time and space), practical efficiency, and appropriate memory management.

2. Submission and Compilation Requirements

   To standardize submissions, you will submit a makefile, which will contain the necessary compilation commands for your code. The target executable will be named **p3**.

   Thus, the graders should be able to compile your program by accessing it on the course server and writing the following command:

   make p3

   Check to make sure that it works on the course server. If the program does not compile and run on the course server, the submission will not be accepted. Please feel free to provide an explanation of your implementation and/or compilation instructions within a comment on Canvas.

3. Input Requirements

   Executing the following line will run your program:

   ./p3 [dictionaryFile] [inputFile]

   Argument 1 is ***dictionaryFile***: the name of a text file that contains all correctly spelled words (a lexicon). Argument 2 is ***inputFile***: the name of a text file to be spell-checked.

4. Output Requirements

   After the main method reads in the dictionaryFile and the inputFile, the program should execute the following operations and then exit normally:

   1. Store lexicon in a hashtable that provides for efficient lookup.
   2. Perform spell check of inputFile. Keep track of the number of misspelled words.
   3. Print out the total number of misspelled words in inputFile.
   4. Print out the total time needed to “create” the perfect hash.
   5. Print out the total runtime during the spell checking.

   No other outputs should be observed.

5. Structural and Operational Requirements
   A. Hashtable Implementation
      - Collision resolution or mitigation should be considered. (Hint: You should implement a linear-space, perfect hash by creating a hash table of hash tables. Note that this will require the use of a random number generator and universal hashing.)
      - Initialize the outer hash to a "reasonable" size. Consider the following: the interior hashes will be quadratic in the size of the number of collisions per bucket, thus you should initialize the outer hash to a size large enough to keep the number of collisions low, e.g. the size of the lexicon should be a good guess here.
      - Constructing the hash table should be done as follows:
        - Initialize the outer hashtable and randomly draw a hash function for use with the outer table. This first draw is kept usually without question or confirmation of "goodness".
        - Hash the lexicon into the outer table and count the number of collisions in each bucket.
        - For each bucket, use the quadratic-space method discussed in class: initialize each inner table to square size of the number of collisions for that bucket. Keep randomly drawing a hash function for each bucket until no inner collisions.
   B. Application: Spell Checking
      - The program sequentially investigates each word in inputFile, confirming if this word exists in the lexicon. If so, it is assumed the word is spelled correctly; if not, the word is spelled incorrectly.
      - The program will keep a count of the number of misspelled words in inputFile.

   Notes and implementation details:

   - Although this document provides a general scheme or framework for implementation, there are many details of design left to you. For example, you must design a family of hash functions that can be randomly drawn, that map the set of strings to a set of indices. (Hint: consider using some variation of the random vector method in combination with folding.)
   - To earn full marks it is expected that you will make the hashtable very efficient (in space and time). Goal: implement a linear-space, perfect hashtable.
   - You will likely need to use C++ ctime or some other timing construct.

6. Rubric

List of Requirements	**Percentage**
Hashtable implementation	0.30
Design of universal hash family	0.10
Hash selection scheme	0.30
Spell checking (traversal of input and lexicon)	0.10
Speed of hash retrieval and construction	0.10
Main method conforms to specs	0.10
TOTAL	**1.00**

7. Programming Languages **[Boilerplate Text]**

   I encourage you to submit your projects using C++. Take note that there are many versions of C++; you must use the version that is currently running on the course server. With prior approval from me, you may choose a different programming language (I have already approved Java for students). Keep in mind that one of the main goals of our class projects is for you to learn how to construct various data structures from the most elemental programming constructs. Thus you will not receive credit when using any pre-existing structures from programming libraries (or code that has been created or designed by others). For example in C++ you *cannot* use the pre-existing vectors, stacks, lists, etc. For some programming languages, complex data structures (non-elemental constructs) are "built-into" the language. You cannot use any built-in structure. For example, in Python, both list and "array" structures are fairly complex and not elemental programming constructs, e.g., they can change size dynamically. For this reason and others, I am unlikely to approve the use of Python for projects. If you hypothetically did use Python, however, you would not receive credit when using those built-in structures. If you have any questions as to what structures are permitted (and which are not permitted), given your language of choice, please ask me.

8. Planning and Design **[Boilerplate Text]**

   Before implementation, you should plan and design your project using standard approaches, e.g. UML class diagrams, flow diagrams, etc. If you have questions pertaining to your project, I will first ask to see your designs. I will not look at your code without first viewing your design documents. You will be faced with many design decisions during this project. It is best to spend the requisite time during the design stages to assure an appropriate and efficient implementation is built. Consider your options, perform a theoretical complexity analysis of the different options, and base your decision on the results of your analysis.

9. Testing and Debugging (Optional) **[Boilerplate Text]**

   You may wish to construct an interactive interface to test the functionality of your structure at intermediate stages of development. This would likely be most efficient with an interactive interface that allowed you to interactively test various functionalities of your structure given different inputs. If you do implement a testing interface, please be sure to comment it (so that it does NOT execute) before submission.

10. Version Control (Not submitted, but encouraged) **[Boilerplate Text]**

    I strongly recommend that you back-up your work periodically throughout the development process. This can mitigate a disaster scenario where you might accidentally delete your program files. I also recommend employing a version control strategy which records your development at different stages (versions). If you have time, I encourage you to investigate GitHub to facilitate version control. Otherwise you can make use of a more simplistic naming scheme: each time you save a file, change the filename to indicate a version: filename v1.cpp, filename v2.cpp, ...