

1 Hashtable

1.1 Summary

You will design and implement a hashtable Structure. The application of the structure will include spell checking and word counting. Words in an inputFile will be sequentially investigated and compared to a comprehensive list of correctly spelled words – a lexicon. The lexicon will be stored 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. Although some tips are provided below (and have been provided in class), the determination of *how* to make the structure efficient is largely your charge.

1.2 Programming Languages

I encourage you to submit your projects using C++. You may choose a different programming language with prior approval from me. However there are caveats as not all programming languages have the same characteristics. Also note, if choosing a language other than C++, you may by chance choose a language which the TAs are not familiar, thus limiting the amount of assistance they might provide. Note 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** pre-existing such as vectors, stacks, lists, trees etc. To help facilitate io and timing, you may use ctime, string, ifstream, ostream, and stringstream.

Note: There are many versions of C++. You must use the version that is currently running on the course server.

Please note that complex data structures (non-elemental constructs) are “built-into” some programming languages. If this is the case, you cannot use the 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. If you are using Python, you will not receive credit when using these 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.

1.3 Input Requirements

The program will take three command line arguments (**Do not prompt the user**).

```
./p5 [dictionaryFile] [inputFile]
```

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

1.4 Structural and Operational Requirements

1. Hashtable Implementation

- Collision resolution or mitigation should be considered. **Hint (do this):** Implement a linear-space, perfect hash by creating a hash table of hash tables. Note: 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:
 - (a) 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".
 - (b) Hash the lexicon into the outer table and count the number of collisions in each bucket.
 - (c) 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.

2. Application: Spell Checking

- The program sequentially investigates each word in `inputFile` and confirm that 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, I have provided 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: I suggest 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. (More info in Output Section.)

1.5 Output Requirements

The main method should read in the inputFile and dictionary

The program should then execute the following operations, provide the following output, and then exit normally:

1. Store lexicon in a hashtable that provides for **efficient** lookup. Perform spell check of inputFile. Keep track of the number of misspelled words.
 - **Print to the screen, the total number of misspelled words in inputFile.**
 - **Print to the screen, the total time needed to “create” the perfect hash, and the total runtime during the spell checking. (How – see notes below.)**

No other outputs should be observed.

1.6 Submission and Compilation Requirements

Please feel free to provide a discussion or explanation of your implementation within Canvas Comments. Submission Deadline – See BB. Budget your time well. Include significant time for design / planning and testing / debugging. Please submit early and often! Your last submission (before the end of the grace period) will be graded.

Your code must compile and run on the class server.

To standardize submissions, you will submit a makefile, which will contain the necessary compilation commands for your code. The target executable will be named p5.

Thus, the following steps should run your code on the course server (CHECK IT).

```
make p5
./p5 [dictionaryFile] [inputFile]
```

If the program does not compile (following your instructions) or the program does not run, the submission will not be accepted.

1.7 Testing and Debugging (Optional)

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.*

1.8 Rubric

Efficiency should be considered.

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	1.0