CS 245 - Assignment 1 Fall, 2019

Assignment 1 - Spell Checker

The goal of this assignment is to demonstrate your mastery of trees and similar structures by generating a spell checker using external data.

Background

All commonly-used word processors, smartphones and other computing platforms have a built in spell check function allowing typists to check their spelling in real time. In this assignment, the task is to create two spell checkers based on a dictionary, one using a search tree; the other using a trie data structure.

We have reviewed the tree data structure for storing data. One similar data structure of this is called a "trie," pronounced either like "tree" or (improperly but frequently) like "try" in order to distinguish it from its more popular related data structure. A trie differs from a tree in that, rather than storing data or a key directly in each node, a trie stores a prefix in a path toward a key, as is shown in conceptually Figure 1 (from Wikipedia).

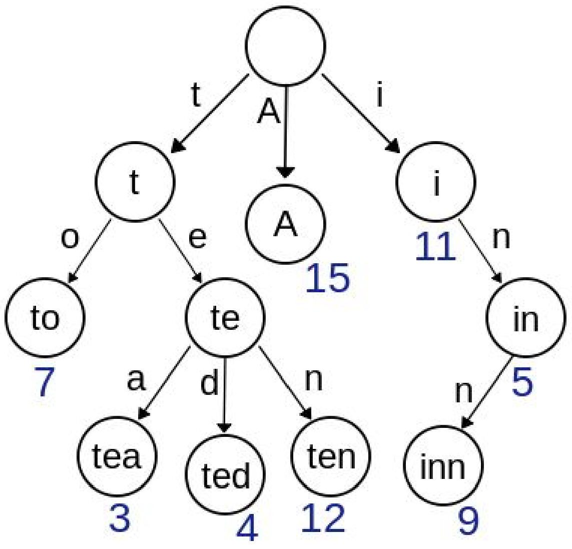

Figure 1: A trie showing words "a", "in", "inn", "to", "tea", "ted" and "ten"

The trie will be your main data structure for one part of this assignment.

Requirements

There are four implementation requirements, each detailed below. Overall, your implementation must run as follows:

```
        java CS245A1 input.txt output.txt
```

… where:

- `CS245A1` is the name of your main class
- `input.txt` is the name of the input file
- `output.txt` is the name of the output file

For input and output files, see Implementation Requirement 4. Your main class may be named something else, but document this change upon submission of your assignment. In addition to the implementation requirements, there is one analysis requirement, also listed in this section.

Implementation requirement 1: Read Jazzy Spell Checker file
Your implementation must read and persist (store) all the words in the Jazzy English language dictionary, hereinafter referred to as “english.0”.. The location of the source file for the Jazzy English dictionary is https://github.com/magsilva/jazzy/blob/master/resource/dict/english.0. No alterations are permitted to this file. This file is assumed to be in the same directory (“folder”) as your implementation’s main class file. This dictionary must be read once only, at the time your implementation is constructed. Note that this file may be read differently depending on the platform (Mac, Unix/Linux or Windows) where it is executed. Your implementation is responsible for reading the file correctly regardless of platform.

Implementation requirement 2: Accept configuration for storage
Your implementation must be prepared to read a property (configuration) file which contains a key “storage” and a value of either “tree” or “trie” in plain text after a “=” symbol. For example:

```
        storage=trie
```

If present, the contents of the file indicate which data structure will store the contents of the english.0 file. This file will be named “`a1properties.txt`” and will exist in the same directory where your compiled implementation is executed. If the properties file is missing, your implementation must default to the use of a trie to store the contents of the english.0 file. Your implementation may read the properties file with the `java.util.Properties` class.

Implementation requirement 3: Find misspellings and suggest alternatives
Using the trie constructed above from english.0 data, create one or more functions to determine whether a word is a misspelling. This function must take at least one argument/parameter: the proposed (possibly incorrect) spelling of a word. If the argument represents the correct spelling of a word, your implementation must indicate this. Where the argument represents a misspelling, your implementation must output at least one and up to three proposed spellings from the english.0 data. The details of how you do this are left to you.

Implementation requirement 4: Process input file to produce output file
Your implementation must read a file, the name of which is provided on command line, process the contents of the input file according to Requirement 3 above and produce an output file based on the contents of the input. The input file has the form of one word per line, some lines containing entries from the english.0 file, some with entries not contained there. For each entry (line) in the input file, your implementation must produce one line in an output file. You may assume this output file needs to be

created upon your implementation's construction. Where the input is contained in the english.0 file, your implementation's corresponding output must mimic the input. Where an entry is not found in the english.0 file, one or more suggestions should appear in the output. For example, the input file may be as follows:

```
trie
basket
```

… in which case the output file must be similar to the following:

```
tree glee
basket
```

… with the last entry (representing a correct spelling) being an exact duplicate of the input.

Analysis requirement
You are required to determine which overall strategy (using the search tree or the trie) is faster and the reason why it is faster. Justify your answer using big-Θ notation and other techniques from this course.

Extra Credit

This assignment is graded on adherence to the requirements above. Your grade for this assignment will be based exclusively on the implementation and analysis contained in the Requirements section. There are three opportunities for extra credit, each with a different weight, listed below. You may attempt any single extra credit opportunity or any combination of them. Any extra credit points will be added to your Midterm 1 grade.

Extra Credit 1 (3 points): Read directly from github
Your implementation is required to read the english.0 file from the same directory where it is run. Instead, your implementation may read the file directly from the github source listed in Implementation Requirement 1. While this file has not changed in years, in theory, this allows continual updates to the spell checker when new entries are submitted.

Extra Credit 2 (10 points): Good spell checking
While there is a requirement that the spell checker produce suggestions for misspellings, there is no requirement that those suggestions be any good. https://www.macmillandictionary.com/us/misspells.html contains many common misspellings and their corrected form.

One technique for finding an alternative to a misspelling is Levenshtein Distance, which is the number of edits to change one string to another. While it is nearly ideal for determining choosing among a limited set of alternatives, it is computationally inefficient on a large number of entries, such as is found in english.0 or any natural language. This extra credit opportunity would have you produce a good spell checker, one which provide useful suggestions such as might be found with a Levenshtein Distance algorithm. However, your implementation should be computationally efficient.

Extra Credit 3 (7 points): Active spell checking
Many current spell checkers perform their spelling function while the user is actively engaged in typing. Such a spell checker may provide suggestions after each letter is typed. This portion of the active spell checker will not operate on an input or output files. Your implementation must be written in a different (main) class in order to support active spelling. Suggestions may appear in any manner of your choice.

Submission

Submit the following on Canvas:

1. The Java source code for your implementation conforming to the Implementation Requirements. This must include your search tree and trie classes and a main file.
2. Your analysis of the running time for the Analysis Requirement in a README file (text, PDF or Word document).
3. Your optional comments in a README file (text, PDF or Word document) to help the grader understand or execute your implementation. Describe any extra credit submission(s) here as well.
4. Your optional extra credit implementations.

Grading

Your grade for this assignment will be determined as follows:

- 70% = Implementation: your implementation must run successfully with the source files and expected data (english.0). It must produce the expected results, a sample of which appears in the Requirements section above. Any deviation from the expected results may produce no credit for implementation.
- 10% = Decomposition: in the eyes of the grader, your solution follow the suggestions above or otherwise must represent a reasonable object-oriented and procedural decomposition to this problem.
- 15% = Efficiency: within the constraints in the “requirements” above, your implementation must use the minimum amount of computational time — O(m), where “m” is the length of the word being checked — and minimum space (amount of memory).
- 5% = Style: your code must be readable to the point of being self-documenting; in other words, it must have consistent comments describing the purpose of each class and the purpose of each function within a class. Names for variables and functions must be descriptive, and any code which is not straightforward or is in any way difficult to understand must be described with comments.