Assignment 6
Due: Nov 9, 2023 at 11:59PM

1 Assignment 6 Specification

1.1 Goals

1. Practice converting visualizations of data structure operations to Java code
2. Practice building and traversing the trie data structure

1.2 Implementation

The skeleton code at `https://classroom.github.com/a/9Pbopl2a` is the start of a program to perform spell checking and suggestion. Given a dictionary, the program constructs the requested data structure to hold the dictionary. Then, the program is given a word to check. The program either reports that the word is in the dictionary or gives a list of possible corrected spellings – words that differ by one or two letter substitutions. Currently the program has three techniques implemented: an array dictionary, a binary tree dictionary, and a hash table dictionary. All three approaches use a brute force approach to generate and check all possible edited versions of an incorrect word.

Improve lookup speeds by implementing a trie data structure and accompanying methods. When looking up incorrect words, design a search that is limited by which parts of the trie exist rather than generating every possible edit. For words that differ by one or two letters, only consider substitutions. Do not consider character insertions or deletions for this assignment.

The skeleton code has an abstract **Speller** class. This class has methods to insert a word, check if a word exists, and get suggestions if the word does not. There are three classes already implemented:

Linear Implements an array data structure

BinTree Implements a balanced binary tree data structure

Hash Implements a hash table data structure

For this assignment, you will implement the methods in the remaining class: **Trie.java**. The `getSuggestions` method in your class **should not** generate every possible edit word like the other three classes do. It should instead recursively traverse existing nodes in the trie, stopping early at dead ends as was discussed in lecture and lab. It should return its array of suggested strings in sorted order. You will get **no credit** for `getSuggestions` if you generate all the edit words in an efficient way like the other classes do. The two dead-end cases are when there have been more than the allowed number of edits in the current path from the root, or when the path is deeper in the tree than the query word's length.

If you want to create a nested class or separate class to represent trie nodes, you can do that. You can also use `ArrayList` objects and/or Java's built-in sorting method in your internal code if you find them helpful. All inserted and tested words will consist of solely lower-case English letters. If you add any additional files to the project, make sure that you stage/add/commit/push them in git to submit them.

1.3 Testing

JUnit tests are provided in the **edu.wit.cs.comp2350.tests** package. The tests check if insertion of various sizes works, and checks if `contains` and `getSuggestions` work. For grading this assignment, I will use a larger dictionary in addition to the ones supplied. You can change the dictionary that **A6.java** uses by changing the file path on **Line 58** to a different file. The tests are set to timeout quickly for this assignment. If your code takes too long to run, even if your answers are correct, you will only get partial credit. The ***Fast** methods test if the results are fast and correct, whereas the ***Correct** tests have a slower timeout speed, and just check for correctness.

A **ChartMaker** class is included, which will create a chart of runtimes of each data structure class based on the number of string lookups. You can use this chart to verify that your trie's runtime is what you expect.

2 Grading

Trie contains: 75%

Trie suggestions: 25%