School of Computing Sciences

Module: CMP-5014Y Data Structures and Algorithms
Assignment: Coursework Assignment 2: Word Auto Completion with Tries

Set by: Anthony Bagnall (ajb@uea.ac.uk)
Checked by: Jason Lines (j.lines@uea.ac.uk)
Date set: Tuesday 30th January 2020
Value: 35%

Date due: Wednesday 29th April 2020 3pm **(Week 10)**
Returned by: *date missing*
Submission: PDF prepared with PASS submitted electronically on e:vision

Learning outcomes

Outcomes we aim for are: Increased experience of problem-solving and algorithm design and analysis; further experience of programming in Java; increased awareness of the importance of algorithm complexity.

Specification

Overview

Your assignment involves developing a simplified system for auto-completion of words. The assignment is in three stages:

1. read in a document of words, find the set of unique words to form a dictionary, count the occurrence of each word, then save the words and counts to file;
2. define a trie data structure for storing a prefix tree and design and implement algorithms to manipulate the trie; and
3. make a word completion system using the dictionary from part 1) and the data structure from part 2).

Your submission should include a document called report.pdf containing the required psuedo code and code listing.
I am obviously aware that there are trie implementations available online. I have no problem with you looking at them to help understand how to solve the problem, but clearly I do not want you copying code without understanding. Hence, if you do not adopt the required structure or your pseudocode does not match your code, you will lose marks. Do not add more than is asked for. If we suspect that you do not actually understand your implementation, we reserve the right to require you to walk us through your code and explain how it works.

Description

Part 1: Form a Dictionary and Word Frequency Count (20%)

Design an algorithm that takes as input a list of words and returns a dictionary of words and the frequency count of each word. Implement your algorithm as part of a class `DictionaryFinder` that fulfills the following process:

1. read text document into a list of strings;
2. form a set of words that exist in the document and count the number of times each word occurs in a method called `FormDictionary`;
3. sort the words alphabetically; and
4. write the words and associated frequency to file.

In your algorithm design, you may assume standard data structures are available and that you do not have to give pseudocode for operations such as `add`, `remove` and `contains`. For your implementation, you may use any of the built in data structures and algorithms available in Java. You should make the operation `formDictionary` as efficient as possible. The skeleton `DictionaryMaker` class on blackboard includes a method to read the words from a file into an `ArrayList` and an example of outputting a collection to file. You simply need to implement the methods `formDictionary` and `saveToFile`. A small example input and output text file (`testDocument.csv` and `testDictionary.csv`) are also on blackboard. Include a main method demonstrating the usage of your class. I am not interested in testing your text parsing skills, so you may assume the input contains only lower case comma separated words with no punctuation, hyphenation etc. Describe your algorithm in psuedocode in a section of report and perform an algorithm analysis. Don't over think this, it should be pretty simple to do this.

Part 2: Implement a Trie Data Structure (50%)

You need to design and implement a trie to hold string keys, including methods to manipulate the trie. You should assume that all the characters of any word input to the trie are lower case and restricted to contain the characters 'a' to 'z'. Figure 1 gives and example dictionary and trie.

For each requirement, you should give a pseudo code algorithm in your report that is directly comparable to your implementation. You do not need to give algorithmic descriptions of any secondary data structures you employ.

1. Define a `TrieNode` data structure and class that contains a list of offspring and a flag to indicate whether the node represents a complete word or not. Offspring should be stored in an array of fixed size 26 and the char values of the characters in the trie used as the index. So, for example, the letter 'a' is represented by the position 0 in the offspring array. Hence, the root node for the trie shown in Figure 1 would contain a `TrieNode` array of size 26 with all null values except in positions 1 ('b') and 2 ('c').
2. Define a `Trie` data structure and class with a `TrieNode` as a root.

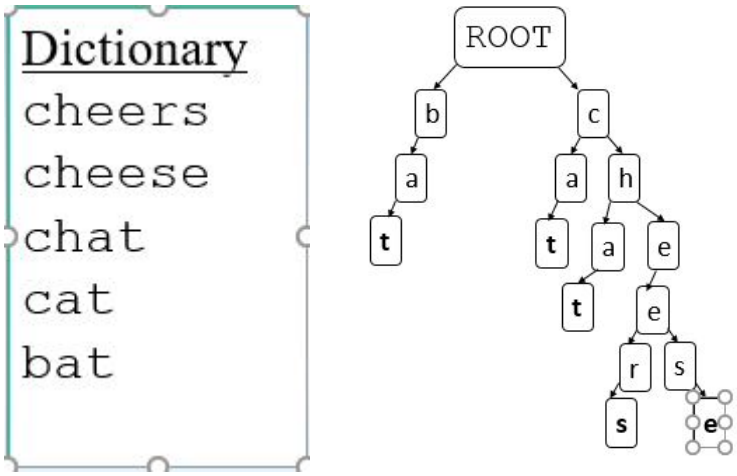

Figure 1: Example Dictionary and Resulting Trie. Complete keys are indicated in bold.

Design and implement the following algorithms to manipulate the trie.

1. `boolean add(String key)`: adds a key to the trie, creating any nodes required and returns true if add was successful (i.e. returns false if key is already in the trie, true otherwise).

2. `boolean contains(String key)`: returns true if the word passed is in the trie as a whole word, not just as a prefix. So, for the example in Figure 1,
`contains("chee")`, `contains("afc")` and `contains("ba")`
would all return false, but
`contains("cheese")` and `contains("bat")`
would return true.

3. `String outputBreadthFirstSearch()`: returns a string representing a breadth first traversal. So, for example, for the tree shown in Figure 1, the method should output the string `"bcaahttaetersse"`

4. `String outputDepthFirstSearch()`: returns a string representing a pre order depth first traversal. So, for example, for the tree shown in Figure 1, the method should output the string `"batcathateersse"`

5. `Trie getSubTrie(String prefix)`: returns a new trie rooted at the prefix, or null if the prefix is not present in this trie. So, for example, calling `getSubTrie("ch")` on the trie in Figure 1 would return the trie shown in Figure 2. Note the word flags are retained in the new `Trie`.

6. `List getAllWords()`: returns a list containing all words in the trie. So, for example, the trie shown in Figure 1 would return `bat, cat, cheers, cheese, chat`, whereas calling on the `Trie` in Figure 2 would return `eers,eese,at`. The order the words are returned in is unimportant, but you should make your algorithm `getAllWords` as efficient as possible.

I want you to follow the exact structure described above. You may add the odd small helper method and add extra attributes, but do not make substantial changes. I am not interested in testing your understanding of programming 2 here, so do not make it generic, iterable etc. Again, I realise there are trie implementations available, so engineering you code will just evoke suspicion. `TrieNode` and `Trie` should be separate classes (i.e. not nested). You may use any of

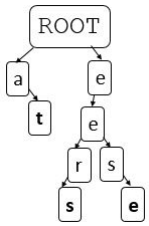

Figure 2: Trie resulting from calling `getSubTrie("ch")` on the trie shown in Figure 1

the data structures provided in the standard Java release. Include a main method in class `Trie` that demonstrates the functionality of your data structure and algorithms.

Part 3: Word Auto Completion Application (30%)

Your task is to adapt you your solutions to parts 1 and 2 to use as a basis for a prototype word completion program. The challenge is, given a partially complete word, find the most likely complete words based on the word frequencies in the text file lotr.csv. You do not need to develop an interface or anything complicated. It should be a simple program that loads a set of searches, finds the best matches, displays the results to the standard output stream and also saves them to file. The basic use case is:

1. Load all the queries file called queries.csv from the project directory.
2. For each query, find the best three matches (at most) with the most likely first, and with associated estimated probability of correctness. If words have equal probability, choose the first occurring word as determined by a breadth first search. Probabilities should be calculated from the frequencies (see example below).
3. Write the results into a file called matches.csv in exactly the specified format.

An test input and output file using the example in Figure 3 are provided.

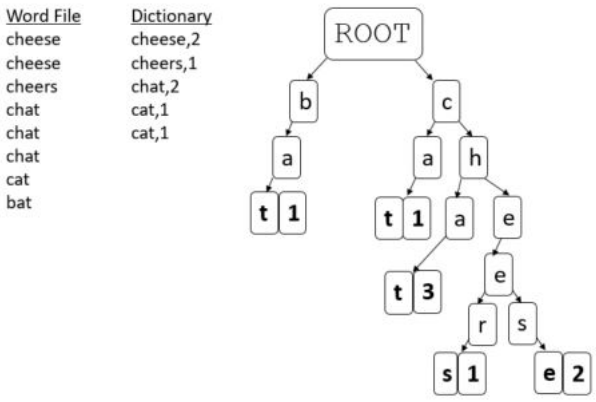

Figure 3: Example dictionary and resulting trie, including frequency counts. Complete keys are indicated in bold.

To accomplish this task you need to:

1. form a dictionary file of words and counts from the file lotr.csv (if you cannot complete part 1, you may use the file gollum.csv instead without penalty).
2. construct a trie from the dictionary using your solution from part 2 (if you cannot complete part 2, you may attempt a solution based on built in Java data structures, although this will attract a penalty).
3. load the prefixs from lotrQueries.csv
4. for each prefix query
   4.1. Recover all words matching the prefix from the trie.
   4.2. Choose the three most frequent words and display to standard output.
   4.3. Write the results to lotrMatches.csv.

You may have to adapt your solution to part 2 for this question to include the frequency counts. If you do so, clone the code and call the class `AutoCompletionTrie`. Probabilities are based on relative frequencies of matches. So, for example, suppose we have the prefix `hi` and we are using the lotr dictionary. We find the following matches and frequencies:

Word	Frequency	Probability
hidden	2	0.025
high	5	0.0625
higher	1	0.0125
hill	9	0.1125
hills	1	0.0125
hillside	2	0.025
him	21	0.2625
himself	5	0.0625
his	33	0.4125
hiss	1	0.0125

The sum of frequencies for words matching “hi” is 80, so the probabilities are found by dividing the frequency by 80. E.g. the probability of ”his” is 33/80. You application should display the following to standard output
his (probability 0.4125)
him (probability 0.2625)
hill (probability 0.1125)
and output the following line to file hi,his,0.4125,him,0.2625,hill,0.1125
If the prefix is also a complete word in the trie it should be included in the calculation and may be displayed. So, for the example above, the query “hill” should match hill, hills and hillside. Include a pseudo code description of your algorithm in your report and implement your solution in a class AutoCompletion.java. Include the output in your report and generate lotrMatches.csv in the project directory.

Relationship to formative work

Please see the following lectures and lab exercises for background on each of the specified tasks:

- **Describing algorithms**: the lectures in week 2 introduced informal and formal descriptions of algorithms. Coursework 1 required algorithm design, description and analysis.
- **Dictionaries**: Lecture 9 covered hash tables and dictionaries. There is a relevant lab sheet.
- **Trees and Tries**: Lectures 10 and 11 covered trees and tries. There are relevant lab sheets.

Deliverables

You must submit a `.zip` file to BlackBoard using the submission point located in the `Summative Assessment` section. The `.zip` must contain the source files of your program, preferably in a Intellij project. The `.zip` file must also contain a `.pdf` file for your report (description of solutions to part 1-3 and a code listing). Please follow this link for a template report
`https://www.overleaf.com/read/wdrrnxknztkr`
Copy this project and use it as a template. There is help on overleaf on programming 2 blackboard.

Plagiarism and collusion: Stackoverflow and other similar forums (such as the module discussion board) are valuable resources for learning and completing formative work. However, for assessed work such as this assignment, **you may not post questions relating to coursework solutions in such forums**. Using code from other sources/written by others without acknowledgement is plagiarism, which is not allowed (General Regulation 18).

Resources

- **Previous exercises**: the lab exercises for hash tables, trees and tries will help, and the teaching assistants can offer direct support for these.
- **Discussion board**: if you have clarification questions about what is required then please ask these questions on the Blackboard discussion board. This will enable other students to also benefit from the question/answer. Please check that your question has not been asked previously before starting a new thread.
- **Course text:** Goodrich, M. T., Tamassia, R. (2005) *Data Structures and Algorithms in Java*, 4th edition, especially chapter 13.

Marking Scheme

1. **Form a Dictionary and Word Frequency Count** (20 marks, 7% of the overall module marks)
2. **Implement a Trie Data Structure** (50 marks, 17.5% of the overall module marks)
3. **Word Auto Completion Application** (30 marks, 10.5% of the overall module marks)

Total: 100 Marks, 35% of the overall module marks