CS 1501 Project 1

Released: Wednesday, January 22

Due: Tuesday, February 4, 11:59 PM

Goal

To gain a better understanding of search algorithms and symbol tables by implementing an autocompletion engine.

Background

In order to make typing on a mobile device quick and easy, an autocomplete feature is commonly implemented to try to guess what word a user wishes to type before they are finished. Such a feature requires extensive use of search algorithms.

Specifications

- Your Box folder contains a copy of the English dictionary that you will use for this project, named `dictionary.txt`
- You must implement a De La Briandais (DLB) trie data structure (as described in lecture) to use in your project. This data structure should be entirely your own code, and should not use code from the Java class library or elsewhere (though you are welcome to use `TrieST.java` for ideas).
- When your program is run, it should first create a new DLB trie and add all of the words in `dictionary.txt` to that trie (this will be the dictionary trie).
- Once the dictionary trie is established, you should prompt the user to start typing a word. For this project, you will be writing only a simple program to demonstrate autocomplete functionality. Due to complexities with gathering single character input in Java, you should accept a single character at a time, each followed by `Enter`. After each character, present the user with the list of predictions of the word that they are trying to type.

  If the user types a number (1-5), you should consider the user to have selected the corresponding prediction, and restart the process for a new word. Consider `$` to be a terminator character; if the user

enters a `$`, consider the characters input so far to be the full word as intended (regardless of suggestions). Finally, if the user enters a `!` at any point, your program should exit.

- To generate the list of predictions, your program should not only consult the dictionary trie, but also keep track of what words the user has entered in the past. If the user has previously entered the same sequence of characters as a prefix to a word, you should prioritize the words that most frequently resulted from this sequence previously. If the user has never entered the current sequence before, or has entered fewer than 5 words with the current sequence as a prefix (i.e., not enough words to complete the list of 5 predictions), your program should suggest words from `dictionary.txt` that have the current sequence as a prefix.
- You program should propose at most 5 suggestions at a time. If there are fewer than 5 suggestions available from the user's history and the dictionary trie combined, then only print out the available suggestions.
- If the current sequence of characters has not been entered by the user before and does not appear in `dictionary.txt`, you should display a message to the user stating that no predicions were found, and allow the user to continue entering characters one at a time. Once the user enters a `$`, you should consider the word to be finished and add it to the user's history so that you can predict it in the future.
- Your program should run in a case sensitive manner in order to allow for easier prediction of proper nouns and acronyms.
- The design of the data structure that keeps track of a user's previously entered words is entirely up to you. You must create a file named `approach.txt` that both describes your approach to implementing this symbol table and justifies your decision to take this approach. Note that this file does not need to be extensive, just a few lines so the TA is aware of what to look for in your code and why you chose this approach.
- The history of the user's entered words should persist across runs of your program. To enable this, your program should save a representation of this data structure to the file `user_history.txt` before exiting.
    - Do not add this file to your Box folder! It should be generated by your program if it does not exist.
- Each time the user enters a character, you should use Java's `System.nanoTime()` to calculate how long your program takes to find the predictions. You should display this time along with the list of predictions.
- After the user enters `!`, your program should output the average time that was required to produce a list of predictions.

An example run of the program would proceed as follows:

```
Enter your first character: t

(0.000251 s)
Predictions:
(1) t    (2) ta    (3) tab    (4) tab's    (5) tabbed

Enter the next character:  h

(0.000159 s)
Predictions:
(1) thalami    (2) thalamus    (3) thalamus's    (4) thalidomide    (5) thalidomide's

Enter the next character:  e

(0.000052 s)
Predictions:
(1) the    (2) theater    (3) theater's    (4) theatergoer    (5) theatergoer's

Enter the next character:  r

(0.000225 s)
Predictions:
(1) therapeutic    (2) therapeutically    (3) therapeutics    (4) therapeutics's    (
5) therapies

Enter the next character:  e

(0.000182 s)
Predictions:
(1) there    (2) there's    (3) thereabout    (4) thereabouts    (5) thereafter

Enter the next character:  3


 WORD COMPLETED:  thereabout

Enter first character of the next word: t

(0.000128 s)
Predictions:
(1) thereabout    (2) t    (3) ta    (4) tab    (5) tab's

Enter the next character:  h
```

```
(0.000094 s)
Predictions:
(1) thereabout    (2) thalami    (3) thalamus    (4) thalamus's    (5) thalidomide

Enter the next character:  e

(0.000085 s)
Predictions:
(1) thereabout    (2) the    (3) theater    (4) theater's    (5) theatergoer

Enter the next character:  r

(0.000145 s)
Predictions:
(1) thereabout    (2) therapeutic    (3) therapeutically    (4) therapeutics    (5) t
herapeutics's

Enter the next character:  e

(0.000130 s)
Predictions:
(1) thereabout    (2) there    (3) there's    (4) thereabouts    (5) thereafter

Enter the next character:  !


Average time:  0.000145 s
Bye!
```

Submission Guidelines

- Upload your submission to the provided Box folder named `cs1501-p1-abc123`, where `abc123` is your Pitt username.
- **DO NOT** upload `user_history.txt` to your Box folder; it must be generated by your program.
- **DO NOT** upload any IDE package files.
- You must name your main program file `ac_test.java`.
- You must be able to compile your program by running `javac ac_test.java`.
- You must be able to run your program by running `java ac_test`.
- You must fill out `info_sheet.txt`.
- The project is due at the precise date and time stated above. Upload your progress to Box frequently, even far in advance of this deadline. **No late assignments will be accepted.** At the deadline, your Box

folder will automatically be changed to read-only, and no more changes will be accepted. Whatever is present in your Box folder at that time will be considered your submission for this assignment—no other submissions will be considered.

Additional Notes

- You are free to use any data structures written by you, the textbook authors, or in the Java standard library to implement the user history symbol table. However, if you use code that you do not write yourself, you must research the efficiency of that particular implementation and discuss that in your `approach.txt`.
- Note that if your user history predictions contain a word that is also contained in the dictionary predictions, this word should not be presented as a suggestion to the user twice in the same prompt (e.g., the final list of predictions in the example above).
- You do not need to implement any sort of autocorrect. You can assume that each character entered by the user is intentional.
- You can assume that the user will not try to enter any numerical characters aside from selecting a prediction.

Grading Rubric

Feature	Points
DLB trie implemented as described in class	20
Dictionary predictions are correctly populated	25
User history predictions are correctly populated	30
Sufficient justification of user history approach	10
User interaction works as specified	5
Timing data presented	5
Assignment info sheet/submission	5