Program 3 – CSCI 211 Fall 2023

Trie

Introduction:

Trie, pronounced "try", is an efficient information re***Trie***val data structure. Using Trie, search complexities can be brought to an optimal limit key length.[1] This is done by considering the prefix of a String. The tree is built using the minimum required prefix. For example, *predict*, *pretend* and *prefix*, all share the prefix "pre". Therefore, a trie would traverse the same "pre" path for all three words – not until the "d", "t", and "f", respectively, would the path differ.

Read the details of a trie data structure at the following website: https://www.toptal.com/java/the-trie-a-neglected-data-structure Though code is given in this website, we will use a different implementation.

For program 3, you are asked to write the ***createDictionary, build, words*** (2 overloaded methods but you will only write the recursive method), and ***prefixWords*** (2 overloaded methods in which you will write both the non-recursive and recursive methods).

Program Details:

1. Begin by creating an IntelliJ Maven Project, <YourLastName>Program3, replacing <YourLastName> with your actual last name.

   From Blackboard, download, unzip and save all the .java files (except for the test file) to your project's src>main>java folder. Save the unit test file to src>main>test>java. Save all the .txt files to your project's root folder. Rename Trie_Starter.java to Trie.java (this includes the class and constructor names) – be sure to KEEP the ***implements TrieADT***, which is the Abstract Data Type associated with this assignment. Trie.java is the file in which you will implement a few methods whereas the TrieTest_Student is 12 of the 25 JUnit tests that will be used to test your program – I highly recommend writing additional test cases of your own. Note that only these 25 JUnit tests will be used to grade your file (i.e, you will only get credit for those tests that pass even if your code otherwise runs on command line). I've also provided additional helper classes. ***TrieADT*** which is the abstract data type that specifies the methods required in Trie. ***TrieDriver***, which may be used for development before trying to run the unit tests. ***TrieNode*** is the node class used in Trie. ***InvalidTrieException***, which is a customized exception to throw should you encounter an error.

2. Open Trie.java and add the following header to the beginning of your code:

```
// CSCI 211
// Jane Doe (use your name)
// Student ID 12345678 (use your student ID)
// Program 3
// Due

// In keeping with the UM Honor Code, I have neither given nor
// received assistance from anyone other than the instructor/TA/tutor.
// Put a program description here
```

3. Review the code that has been provided:

[1] https://www.geeksforgeeks.org/trie-insert-and-search/

a. three instance variables/ArrayLists: ***dictionary, root,*** and ***filename***
b. a constructor, which instantiates the TrieNode ***root*** and the ArrayList ***dictionary***. Normally, the constructor would also call the ***createDictionary*** and ***build*** methods, but I've left these out since they are tested in the JUnit tests.

c. helper methods (given): ***toString***, (2 overloaded methods in which one is recursive) and ***printDictionary***.

4. Open the TrieNode class to observe the node components of the Trie. When a "new" letter is added to the Trie, you will begin by instantiating a TrieNode object. The letter will be assigned to the ***letter*** data field, the ***next*** ArrayList is instantiated, and ***isWord*** is false. The ***next*** ArrayList is initially empty. Below are details also given in the Trie ***build*** method's TODO stub, but I will repeat here in order to explain the TrieNode methods that you will use in ***build***.

   For example, say I am adding my first word "bum" to the trie. I will use a "current" node to traverse the trie. I begin by assigning "current" to the root. At current's position (in this case, the root), I call the TrieNode ***contains*** method to see if 'b' is already in the current's ***next*** ArrayList. It is not, so I call the ***add*** method which adds 'b' to current's ***next*** ArrayList (i.e., the root will now have the letter 'b' in its ***next*** ArrayList). At this point, I also know that 'b' has an empty ***next*** ArrayList because I just added it. Therefore, I immediately add 'u' to 'b's ***next*** ArrayList (i.e., since I had to add 'b' there is no need to call ***contains*** because I know 'b' does NOT have any letters in its ***next*** ArrayList, including 'u' – in fact, all the remaining letters after 'b' can be added without calling ***contains***). I also ***add*** 'm' to 'u's ***next*** ArrayList. My trie looks like the following:

```
root
 |
 b
 |
 u
 |
 m
```

   NOTE: That to implement the above trie, I will use "current" to traverse as I add letters. To begin, as I mentioned, "current" is at the root. Once, I add 'b', I update "current" to be the same TrieNode as 'b'. When I add 'u', "current" will be 'u', and, finally, when I add 'm', "current" is 'm'. When I'm done adding TrieNodes, my root and each of my internal nodes have just 1 element in their ***next*** ArrayLists (m is a leaf and has an empty ***next*** ArrayList). I have one last action – I need to call TrieNode's ***setWord*** method for 'm'. This will change 'm's isWord data field to true. That is, if I traverse the trie to 'm', I have a word (i.e., bum) that was in my dictionary.

   Let's say the second word in my dictionary is "bump". I begin at the root, assigning "current" to the root, and call the ***contains*** method to see if there is a 'b' – there is so I ***get*** this node (i.e., traverse and assign "current" to it). I then check if 'b' ***contains*** a 'u' – it also does so I ***get*** it (i.e., traverse and assign "current" to it). I check if 'u' ***contains*** an 'm' – it does so I ***get*** it (i.e., traverse and assign "current" to it). Finally, at 'm' I have an empty ***next*** ArrayList so I ***add*** 'p' and assign "current" to it. I also call 'p's ***setWord*** since "bump" is also a word.

5. Now, back in Trie.java, look for the TODO stubs for the methods you are to implement: ***createDictionary***, ***build***, recursive ***words***, non-recursive ***prefixWords***, and recursive ***prefixWords***. Detailed instructions are given in these stubs so I won't repeat here.

6. Before each significant step, provide a comment explaining the step (do not comment every line of code). You can add to the comments I've already provided for you.

7. Submit Trie.java on Blackboard. As stated in the syllabus, twenty percent will be deducted per day (or any part of a day) for late submissions. Late programs must be emailed to me: kdavidso@olemiss.edu