**CS 251 Project 5 - Word Autocomplete Search**

**Introduction**

Autocomplete is a feature where in an application predicts the rest of a word based on what the user is typing. It is commonly used in search engines, source code editors, and database query tools. In this lab, you will implement a word autocomplete search engine by using compressed trie and BFS. **Your goal is to correctly build the trie and also answer the query in an output file.**

Download P5 folder as a whole, then extract the file. Open the folder as a project in intelliJ, then click File- Project structure. Set project name as P5; project SDK is 1.8; Porject language level is SDK default; Project compiler output is the …(your path)…./files/out folder. After these, click “Modules” on the left bar. Mark files as Resources; mark src as Sources; mark tests as Tests. Click OK. Go to TrieTest.java, import junit, and run the testcases to check if your project is setup.

Before writing the project, you should first try to read TrieTest.java to see how TAs are going to test it. Basically, we will check the correctness of your output files by comparing yours with expected output. Note that we will use buildTrie and autoComplete methods in Trie class to run testcases so you shouldn’t change the signature of these functions. You may add additional classes/methods/variables to help you accomplish the required functionality. Two examples (sample input & expected output) have been provided inside the ‘files’ folder within the main project directory.

**Input Format**

The input file will be a word list. The **first line** of the input file is the **query** which is the string that a user already typed in the prompt in your search engine. From **the second line to the end of file**, you will be given all the words that should be used to build the trie structure. Note that the first word is query, which should not be inserted into the compressed trie.

For example, in input1.txt,

the first word seg is your query,

From the second line sea

segmentationfault

segment

segmental

segmentation

segv

are what should be built in the compressed trie.

You may assume that there is only one word each line. You may assume that the words will be unique and only contain lowercase alphabetic characters.

**Implementation**

A compressed trie should be built with the word lists given in the input file. The words should be inserted into the trie. There are several cases to consider when inserting a word. You can refer to the insert() operation on this site: <http://theoryofprogramming.com/2016/11/15/compressed-trie-tree/>.  **Each node has at most 26 children.** The precedence of a node’s children should be sorted in **lexicographical** order.

A search method should also be implemented in order to answer the query, i.e. provide the auto-complete recommendations to the query string. **If the string we want to search does not exist, nothing should be printed in output file.** For example, searching for “owl” when the trie only has the word “crow”. **Otherwise, the string we want to search for either exists as a prefix or is fully matched.** For example, searching for “face” when the words “face”, “facebook”, and “facetime” are present in the trie must include all three of them in the query results.

input0.txt

face

face

facebook

facetime

It is required to use BFS with a FIFO queue to **find all the words in the order that they are enqueued**. Furthermore, you will need to include the **height** of the word after each word you found in your output files.

In the example input0.txt, your output file should look like:

face 1

facebook 2

facetime 2

Detailed explanation: Build trie: The first word in input is “face”, so “face” is inserted as a child of root. Then the second word is “facebook”, you should insert it by adding a new node-- “book” under “face”. Now face has 1 child. Then you insert “facetime” by adding a new node “time”. Since **the children list should be in lexicographical order**, you arrange the children list of “face” as [“book”, “face”]. Then the trie is done. Now search for the query “face”. You look for children of root. It has only one child, “face”. And it matches the query, luckily. So you enqueue “face” on a queue. The first word you found is “face”. Then, you dequeue “face”. And enqueue all the children of “face” so “book” and “time” are enqueued. These are two more words you found. Then you dequeue “book”. It has no child. Then you dequeue “time”. It has no child. The queue is now empty, so you finish searching.

Root is of height 0, so “face” is of height 1. Similarly, “facebook” and “facetime” both have height 2.