COP 3502C Programming Assignment#4

Topics covered: Trie

Please check Webcourses for the Due Date

Read all the pages before starting to write your code

Introduction: For this assignment you have to write a c program that will utilize the Trie data structure.

What should you submit?

Write all the code in a single .c file and upload the .c file.

Please include the following lines in the beginning of your code to declare that the code was written by you:

/* COP 3502C Programming Assignment 4

This program is written by: Your Full Name */

Compliance with Rules: UCF Golden rules apply towards this assignment and submission. Assignment rules mentioned in syllabus, are also applied in this submission. The TA and Instructor can call any students for explaining any part of the code in order to better assess your authorship and for further clarification if needed.

Caution!!!

Sharing this assignment description (fully or partly) as well as your code (fully or partly) to anyone/anywhere is a violation of the policy. I may report such incidence to office of student conduct and an investigation can easily trace the student who shared/posted it. Also, getting a part of code from anywhere will be considered as cheating.

Deadline:

See the deadline in Webcourses. **No late submission will be accepted.** The assignment submission will be locked after the deadline. An assignment submitted by email will not be graded and such emails will not be replied according to the course policy.

Additional notes: The TAs and course instructor can call any students to explain their code for grading.

Problem: Text Prediction

Most cellphones are equipped with word completion. If you type in the starting part of the word, suggestions are made to complete the word, based on the frequency with which words are used.

For example, if you've used "computer" 45 times, "complain" 15 times, and "complex" 99 times, if you type "comp", your phone may suggest that you complete the word as "complex", since is the word you've used the most which starts with "comp".

Unfortunately, you often accidentally hit the button to complete the word, and in the instances where this was the word you meant to type, you have lots of erasing to do, potentially.

Thus, you'd like to create your own text completion program which simply suggests the most likely next letter. In the example above, if the three words shown were the only ones that started with "comp", then the most likely next letter would be "l", which appeared 15 + 99 = 114 in the past, whereas "u" only appeared 45 times in the past. It is possible that multiple letters have an equal chance of being the next letter. In this case, you would like your program to list every one of these letters that are "most likely", with equal probability.

Your program will handle two types of commands:

(1) Adding a word to the dictionary of used words a particular number of times.

(2) A query with a prefix, for which your program must produce the most likely next letter, or list of most likely next letters. If the given query is not a prefix of any word previously added to the dictionary, then your program must detect this instead.

The Problem

Given a list of words being used and their frequencies, as well as queries of prefixes at various points in time, answer each of the queries with the most likely next letter (or list of most likely next letters), or answer "unknown word", if there is no word in the dictionary for which this is a proper prefix. (Note: A proper prefix is a prefix of a word that is strictly shorter than it.)

The Input (to be read from in.txt file) - Your Program Will Be Tested on Multiple input files

The first line of the input file contains a single positive integer, *n*, representing the number of commands for your program to execute. Each of the commands follow, one per line, on the following *n* lines.

There are two possible commands.

The first command starts with the integer 1, representing an insert into the dictionary of used words. This is followed by a space and *s*, a string of lowercase letters, representing the word to be inserted. This is followed by a space and *f*, an integer, representing the number of times that word is to be inserted. The command means that the user has used the word *s* an additional *f* times.

(Thus, it's possible that the same word may appear more than once in the set of input commands, representing that the user used that word some, and then used it at a later time some more.)

The second command starts with the integer 2, representing a query. This is followed by a space and *p*, a string of lowercase letters, representing the prefix for the query. Note that a query will NOT make any modifications to the dictionary keeping track of used words.

The total number of letters in all of the input words will not exceed two million. The sum of the frequencies of the added words will not exceed one billion.

The Output (to be printed to out.txt file)

Output will only to be printed for queries. For each query, a single line of output is printed. If the prefix queried is the proper prefix for any word in the dictionary at that point in time, then print out each letter, in alphabetical order, which is the most likely letter to come next. There should be NO SPACES between the letters. If the prefix queried is NOT a proper prefix for any word in the dictionary, print out "unknown word" on a line by itself.

Sample Input

```
1 cap 15
2 ca
2 cap
2 pen
1 cat 20
2 ca
2 c
1 act 10
1 able 10
2 a
1 ace 2
2 a
2 ab
2 ac
2 ace
```

Sample Output

```
p
unknown word
unknown word
t
a
bc
c
l
t
unknown word
```

Implementation Restrictions

You must create a trie to store all of the inputted words and to use to answer each of the queries. Your trie node struct should store the following items of information:

1) The frequency of the node itself (so how many copies the string upto this node has in the dictionary)

2) The sum of frequencies for which this string is a prefix of words in the dictionary, including itself.

3) The current maximum frequency of any child node (thus, if this node represents "ca", and at the current time there are 20 words starting with "cat" and 15 starting with "cap", then 20 should be stored here.

4) An array of size 26 representing pointers to the next possible letters. These pointers should be set to NULL if there are no words in the dictionary that follow these paths. (In most cases, some of the 26 will be set to NULL and some will not.)

Commands of type 1 should be executed using the trie's insert function. Note that since a word can appear more than once in the input, insert should be written in such a way that in some cases, no new nodes are created.

Commands of type 2 should be executed as a function that takes the trie in as a parameter, but that isn't exactly one of the standard trie functions. (This function will look similar to the search function in a regular trie, but with extra code added to execute the desired request.)

Additional Requirement:

You must have to use read data from in.txt file and write the result to out.txt file. The output must have to match with the sample output format. Do not add additional space, extra characters or words with the output as we will use diff command to check whether your result matches with our result. Next, you must have to write a well structure code. ***There will be a deduction of 10% points if the code is not well indented and 5% for not commenting important blocks.***

- As always, all the programming submission will be graded base on the result from **Eustis**. If your code does not work in Eustis we will conclude that your code has an error even if it works in your computer.
- Your code should contain the memory leak detector like programming assignment 1 (otherwise 10% penalty)
- Again, your program must have to compile in Eustis to get points. Note that you can use <math.h> library if you need. In that case you have to use –lm option while compiling your code.

For example: $*gcc filename.c -lm*

Steps to check your output AUTOMATICALLY in Eustis or repl.it:

You can run the following commands to check whether your output is exactly matching with the sample output or not.

Step1: Copy the sample output into sample_out.txt file and move it to the server (you can make your own sample_out.txt file)

Step2: compile and run your code using typical gcc and other commands. Your code should produce out.txt file.

Step3: Run the following command to compare your out.txt file with the sample output file

```
$diff -i out.txt sample_out.txt
```

The command will not produce any output if the files contain exactly same data. Otherwise, it will tell you the lines with mismatches.

Incase if your code does not match, you can use the following command to see the result in side by side:

```
$diff -y out.txt sample_out.txt
```

Rubric:

1) A code not compiling or creating seg fault without producing any result can get zero. There may or may not be any partial credit. But at least they will not get more than 50% even if it is a small reason. Because, we cannot test those code at all against our test cases to see whether it produces the correct result or not.
2) There is no grade for just writing the required functions. However, there will be 20% penalty for not writing insert function, 10% penalty for not writing appropriate search function.
3) Implementing insert operation properly: 25%
4) Implementing prediction properly: 25%
5) Matching test cases : 50%. Your code will be tested against a set of test files and will be grade based on the match.
6) There is no point for well structured, commented and well indented code. ***There will be a deduction of 10% points if the code is not well indented and 5% for not commenting important blocks.***

Hints:

- Start programming assignment as soon as possible.
- Read the entire assignment and get a very good idea what is expected.

- Use the sample input/output to understand it better
- Draw and simulate the sample input/output step by step by drawing the Trie step by step. Update the values of the structure accordingly while drawing.
- Based on that, work on the insert function. You can review the insert function we have discussed in the class. You mainly need to update some extra variables.
- After that, start working on the prediction function. It would be somehow like search with lot of modification. You will do typical search. However, as soon as you are at the end of you search word, you need to make a decision. You might need to return NULL, or you might need to return a string with one or more characters. During this process, you need to check all the children of the current node and their frequency and need to match who has the highest frequency based on the current node's stored max frequency. If multiple children have max frequency, then you can produce a string with those characters and return that string! Don't forget to terminate your string by '\0'