ECE250 Project 3: Spell-Checking Using Trie ADT

Due: Friday, March 10th at 11pm to the DropBox on Learn

Overview

In this project, you will implement a ***trie*** data structure which is defined as: "A trie is a 26-ary tree where the root node represents an empty string ("") and if the k^{th} (k going from 0 to 25) subtree is not a null subtree, it represents a string that is the concatenation of the characters represented by the parent and the k^{th} letter of the alphabet (where 'a' is the 0th letter and 'z' is the 25th)". Each node may additionally indicate that it is a terminal entry. This means when a node has no children. In this project, no characters of any kind outside of the ***26 upper-case English letters*** will be considered to be valid.

For example, *presuming we didn't care about case,* consider the sentence "the fable then faded from my thoughts and memory". If we want to put these nine words into a trie, it will look like as depicted in Figure 1 with two important concepts: (1) only the root node shows its 26 subtrees and (2) some of those subtrees are null because the letters they correspond to are not present in the input.

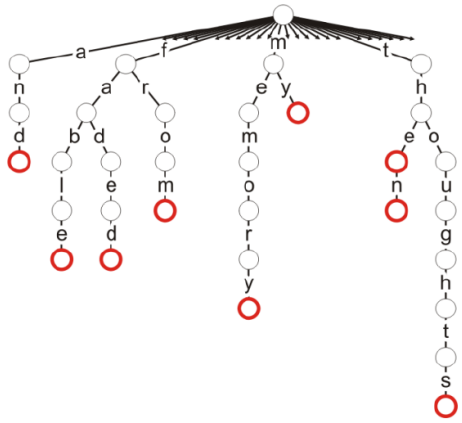

Figure 1: A trie

Note that edges are labeled with letters only for clarity. The letters are not actually stored in the edges or the nodes. Instead, the position of a node determines the key associated with it. The first child of any node represents 'a', the second child represents 'b' and so on. You are free to store an additional symbol to indicate that the letter terminates a word. If you do, this means your trie will be a 27-ary tree. This is not necessary to complete the project.

In this project, you will use a trie to implement a spell-checker. You will be provided with a text document, known as a "corpus", and stored in the file "corpus.txt" and posted on LEARN, which contains words that you may assume are correctly spelled[1]. You will write code to input this data into your trie. Your trie may not contain duplicate words, so this must be accounted for when you are parsing the corpus. You will also implement code to use, manipulate, and analyze the trie. The details can be found in the table of the commands below.

Program Design and Documentation

You must use proper Object-Oriented design principles to implement your solution. You will create a design using classes which have appropriately private/protected data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.** Write a short description of your design (three pages) to be submitted along with your C++ solution files for marking according to the template posted on LEARN. **You may use the vector class, but no others from the STL**.

[1] The corpus comes from a review of the 1991 hit computer game, "Where in Europe is Carmen Sandiego", original available here: http://textfiles.com/games/REVIEWS/careurp.rev. The corpus has been edited to ensure it contains only upper-case English letters.

Input/Output Requirements

In this project, you must create a test program that must read commands from standard input and write the output to standard output. The commands are described below.

Command	Parameters	Description	Expected Runtime	Output
load		Load the corpus into the trie, ensuring that duplicate words are not added. You should do this by opening the file and reading it one word at a time. In the corpus, a "word" is separated from other words by whitespace. You may **not** assume that this command will always start all input files. You do **not** have to worry that some words in the corpus are not valid English words. Pretend that they are[2].	**N/A**	**success** ***Note:*** This command should output "success" no matter what.
i	word	Insert a new word into the trie.	**O(n)** *where n is the number of characters in word*	**success** if the insertion was successful **failure** if the word is already in the trie **illegal argument** (see below the table)
c	prefix	Outputs a count of all words in the trie that have the given prefix. The prefix does not need to be a full word in the trie, although it may be. For example, if the Trie contains the words "CAR", "CARD", and "CARMEN" only, then searching for the prefix CA should return a count of 3, as should searching for the prefix CAR.	**O(N)** *where N is the number of words in the trie*	**count is NUM** where NUM is the number of words with that prefix **not found** if no words with that prefix exist in the trie **illegal argument** (see below the table)
e	word	Erase the word in the trie.	**O(n)** *where n is the number of characters in word*	**success** if the word is in the trie and was erased **failure** if the word is not in the trie, or the trie is empty **illegal argument** (see below the table)
p		Prints all words in the trie in alphabetical order on a single line (use depth-first traversal).	**O(N)** *where N is the number of words in trie*	**word1 word2 word3…** There should be a new line after the last word, but otherwise all words should be printed on a single line. No output is created if the trie is empty.

[2] For instance, the string "II" is in the corpus. It is there on purpose. Why?

Command	Parameters	Description	Expected Runtime	Output
spellcheck	word	Spell-checks the word or offers suggestions if the word is not spelled correctly. For example, assume the trie contains the words: YOU, YOUN, YOUNG, YOUR. The command: *spellcheck YOL* will print a list of all words in the trie starting with "YO". It should print this list alphabetically.	**O(N)** *where N is the number of words in trie*	**correct** if the word is in the trie and terminates with the last letter of the word **word1 word2 word3…** if the word is not in the trie. Start from the first letter with an error For example, the corpus contains the words: YOU, YOUN, YOUNG, YOUR. The commands: *spellcheck YOL* and *spellcheck YO* will print a list of all words in the trie starting with "YO" alphabetically. Therefore, it will output: **YOU YOUN YOUNG YOUR** If a word is provided for which there is no match on the first letter, this command will produce no output other than a newline. For example, the corpus has no words starting with 'Z', so a call to "spellcheck ZEBRA" should not give any suggestions and should output a blank line. ***Note:*** Yes, we know this isn't exactly how spellcheck works
empty		Checks if the trie is empty.	**O(1)**	**empty 1** if the trie is empty **empty 0** if the trie is not empty
clear		Deletes all words from the trie.	**O(N)** *where N is the number of words in trie*	**success** if the trie was already empty, this command should print "success" anyway
size		Prints a message indicating the number of words in the trie.	**O(1)**	**number of words is** *count* where "*count*" is the number of words. Count may be 0 if the trie is empty.
exit		Last command for all input files.		This command does not print any output.

Illegal arguments: For the commands *i, c,* and *e,* you must handle invalid input. If the word contains any characters other than those of the upper-case English alphabet your code must throw an illegal_argument exception, catch it, and output

"illegal argument" (without quotes) if it is caught. Afterward, the code should continue parsing input if any remains. You may assume that the words will not contain spaces. To do this, you will need to:

a. Define a class for this exception, call it **illegal_exception**
b. Throw an instance of this class when the condition is encountered using this line:

```
throw illegal_exception();
```

c. Use a try/catch block to handle this exception and print the desired output of the command

You must analyze the expected runtime of your algorithms in your design document. Prove that your implementation achieves these runtimes.

Valgrind and Memory Leaks

10% of your grade in this project will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. To test your code with Valgrind, presuming you are using an input file test01.in, you would use the following command:

```
valgrind ./a.out < test01.in
```

Test Files

Learn contains some examples input files with the corresponding output. The files are named *test01.in*, *test02.in* and so on with their corresponding output files named *test01.out* etc.

All lines in the input files end with a UNIX newline character (\n), and there are no spaces before this character.

Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you should transfer your files to the eceUbuntu server and test there. We perform automated tested on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect. **A makefile is required for this project since the exact source structure you use will be unique to you. A video has been posted on Learn about how to create it.**

Once you are done your testing you must create a compressed file in the tar.gz format, that contains:

- A typed document, maximum of three pages, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID (for example, I would use my ID "mstachow", not my ID "mstachowsky", even though both are valid UW IDs), and n is the project number. In my case, my file would be mstachow_design_p3.pdf.
- A test program, trietest.cpp, that reads the commands and writes the output.
- Required header files that you created.
- Any additional support files that you created.
- A makefile, named Makefile, with commands to compiler your solution and creates an executable. Do not use the -o output flag in your makefile. The executable's name must be a.out.

The name of your compressed file should be **xxxxxxxx_p3.tar.gz**, where xxxxxxxx is your UW ID as above.