ECE 250 — Winter 2024
Electrical and Computer Engineering
University of Waterloo

Due Date: Mar 11, 2024
Professor: Ziqiang Patrick Huang
Lab Instructor: Ahmed Fahmy

Lab 2:

Spell-Checker

1 Overview

The objectives of Lab 2 are:

- Create a spell-checking program.
- Implement and utilize a Trie ADT to store, spell-check and modify words.

2 Trie ADT

A Trie (pronounced "try"), also known as a prefix tree, is a tree-like data structure used to store a dynamic set of strings. It is particularly efficient for tasks such as searching for strings with a common prefix. Trie data structures are commonly employed in tasks requiring fast prefix matching, such as dictionary implementations, auto-complete systems, and spell checkers.

Structure of a Trie

A Trie is a 26-ary tree that consists of a rooted tree structure where each node represents a single character of a string. The path from the root to a node spells out a particular string. Additionally, each node may have links to other nodes, representing the characters that can follow the current character to form valid strings. Let's illustrate the structure of a Trie with an example. Consider the Trie in Figure 1 representing a small set of strings: "cat", "car", "can", and "dog".

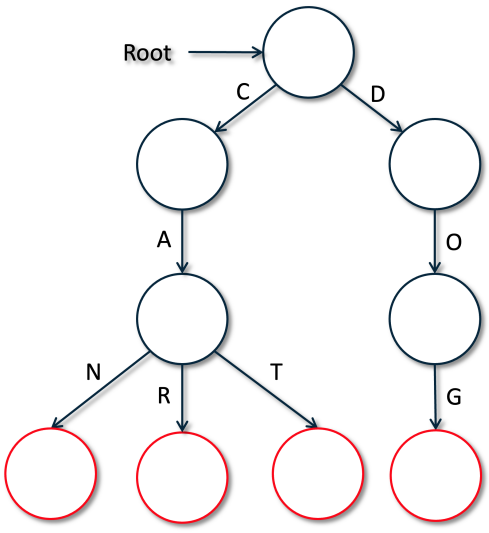

Figure 1: Trie representing a set of strings: "cat", "car", "can", and "dog".

In this Trie:

- Each node represents (but does not store) a character.
- The edges between nodes represent the transition from one character to the next.
- The nodes outlined in red denote the end of a valid word.

Note that edges are labeled with characters only for clarity. The characters are not actually stored in the edges or the nodes. Instead, the position of a node determines the key associated with it. For example, the first child of any node represents 'a', the second child represents 'b' and so on. It is possible to store an additional variable (possibly Boolean) in each node to indicate that a character terminates a word.

Operations on Tries

Insertion

To insert a new string into a Trie, we start from the root and traverse down the tree, creating new nodes as necessary. If a character is not present in the current node's children, a new node is created for that character. Once all characters of the string are inserted, the last node is marked as the end of a word. Let's insert the string "do" into the Trie:

1. Start from the root.
2. Traverse to the node 'd'.
3. Traverse to the node 'o'.
4. Mark the node 'o' as the end of a word.

Now let's insert the string "dot" into the Trie:

1. Start from the root.
2. Traverse to the node 'd'.
3. Traverse to the node 'o'.
4. Create a child node for 't'
5. Traverse to the node 't'.
6. Mark the node 't' as the end of a word.

The updated Trie looks like this:

Search

Searching in a Trie involves traversing the tree from the root, following the path defined by the characters of the target string. If we encounter a node that does not exist in the path, or if the path ends before reaching a node marked as the end of a word, the search fails. Let's search for the string "car" in the Trie:

1. Start from the root.
2. Traverse to the node 'c'.
3. Traverse to the node 'a'.
4. Traverse to the node 'r'.
5. Since 'r' is marked as the end of a word, the search succeeds.

Let's search for the string "ca" in the Trie:

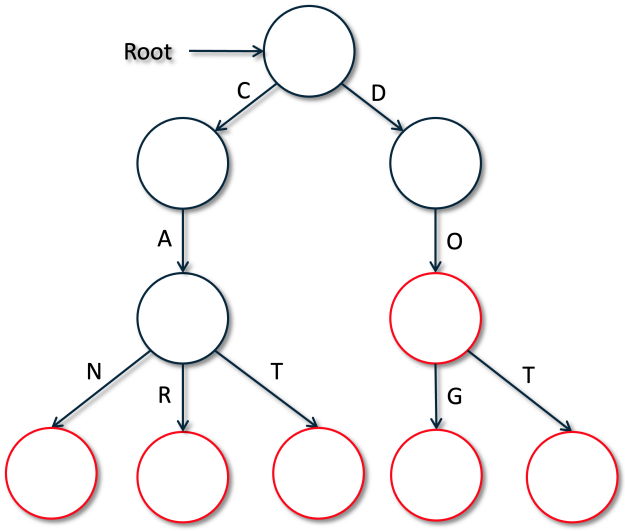

Figure 2: Trie representing a set of strings: “cat”, “car”, “can”, “do”, “dot” and “dog”.

1. Start from the root.
2. Traverse to the node ‘c’.
3. Traverse to the node ‘a’.
4. Since ‘a’ is **not** marked as the end of a word, the search fails.

2.1 Deletion

Deletion in a Trie can be a bit more complex because removing a node may affect other strings. However, the basic idea is similar to insertion: traverse the Trie to locate the node corresponding to the string to be deleted and remove it. If the removal of a node leaves its parent node with no other children, it may also need to be removed recursively. Let’s remove the string “car” from the Trie:

1. Start from the root.
2. Traverse to the node ‘c’.
3. Traverse to the node ‘a’.
4. Traverse to the node ‘r’.
5. Delete the node ‘r’.
6. Recursively delete the parent node if it has no children.
7. Since the parent node ‘a’ has at least one child, the operation is terminated.

Removing “cat” follows exactly the same logic, except Steps 4 and 5 traverse to and remove the node ‘t’, respectively. Now let’s remove the string “can” from the Trie:

1. Start from the root.
2. Traverse to the node ‘c’.
3. Traverse to the node ‘a’.
4. Traverse to the node ‘n’.

5. Delete the node ‘n’.
6. Recursively delete the parent node if it has no children.
7. Delete the parent node ‘a’ as it has no children.
8. Delete the parent node ‘c’ as it has no children.
9. Since the parent node is the root, the operation is terminated.

The updated Trie looks like this:

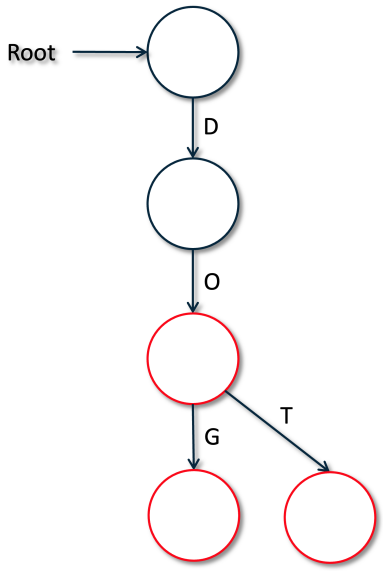

Figure 3: Trie representing a set of strings: “do”, “dot” and “dog”.

3 Project Overview

In this project, you will use implement and utilize a Trie to create a spell-checker. You will be provided with a text file `corpus.txt`, which contains words that you may assume are correctly spelled. Your program should input this data into a Trie. The Trie may not contain duplicate words, so this must be accounted for when are parsing the input file. Your program should accept commands to use, manipulate, and analyze the Trie. The details can be found in the table 1.

4 Program Requirements

The goal of this project is to write a C++ program that implements a spellchecker. You must **implement and use a Trie class** to store and manipulate input strings. **Using the STL library is not allowed**. Every node in the Trie should contain an array of node pointers of size 26 (one child per alphabet letter). You may create more variables for convenience. However, it is not allowed to store strings in nodes to represent input words.

Your program must read commands from standard input (e.g. `cin`) and write to standard output (e.g. `cout`). For each input, the program is expected to execute a designated action with a specified run-time requirement and provide the corresponding output[1] message, as outlined in Table 1. **Note:** You may assume that

[1]The outputs specified in the “Output” column must be displayed as indicated. For example, in the first row, the expected output is the string "success" in lowercase.

all commands are valid. For run-time requirements, we define n as the number of characters in an input word, and denote N as the number of nodes in the Trie. **Note**: In this project, no characters of any kind outside of the 26 upper-case English letters will be considered to be valid. **You may assume the parameters are always valid during evaluation**.

Command	Parameters	Action	Output	Run-time
`load`	N/A	Load the corpus into the Trie	`success`	N/A
`i`	`word`	Insert `word` into the Trie if the word is **not** in the Trie.	`success` OR `failure`	$O(n)$
`c`	`prefix`	Outputs number of words in the Trie that have the given `prefix`.	`count is` num OR `not found`	$O(N)$
`e`	`word`	Erase `word` from the Trie **if it exists** in the Trie.	`success` OR `failure`	$O(n)$
`p`	N/A	Print all words in the Trie in alphabetical order on a single line. No output **if the Trie is empty**.	`word1 word2...` OR (no output)	$O(N)$
`spellcheck`	`word`	If `word` is in the Trie, print `correct`. **If not**, suggest words that have the maximum common prefix. No output **if no** suggestion found. **More details below**.	`correct` OR `word1 word2...` OR (no output)	$O(N)$
`empty`	N/A	Check if the Trie is empty. If empty print `empty 1`, otherwise print `empty 0`.	`empty 1` OR `empty 0`	$O(1)$
`clear`	N/A	Delete all words from the Trie.	`success`	$O(N)$
`size`	N/A	Print the number of words in the Trie.	`number of words is` num	$O(1)$
`exit`	N/A	End the program	(no output)	N/A

Table 1: Input/Output Requirements

Command Spell-check

The `Spell-check` command takes a word (`word`) as a parameter, and spell-checks `word` by checking if it exists in the Trie. If `word` exists then the command outputs `correct`. If not, the command prints all words in the Trie such that these words have the maximum common prefix in the Trie. For example, the corpus document 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.

Provided Files

You are provided with the following files: `driver.cpp` where the `main()` function is implemented, `trie.h` where your classes are *defined*, and `trie.cpp` where your classes are *implemented*. **Do not create, delete or change the name of the provided files**. You will also be provided with the input file `corpus.txt` to be used with the `load` command. In addition, there are some examples of input files along with their corresponding output files. The test files are named `test01.in`, `test02.in`, and so on, with their respective output files named `test01.out`, `test02.out`, etc.

5 GitLab Repository

We are going to use GitLab for code management and submission. The repository is created for you with the URL: `https://git.uwaterloo.ca/ece250-w24/lab2/ece250-w24-lab2-USERID`

6 Evaluation Criteria

Evaluation Focus

Output Accuracy and Performance: The primary focus of the evaluation for Lab 2 will be on the correctness of the output generated by your program and the Performance (Time complexity of the functions designed for different commands). We will use a variety of inputs and edge cases to test the accuracy of your results.

Deductions for Non-Compliance (up to -50 points): Marks will be deducted if your code does not adhere to the specified guidelines, including the prohibition of using C++ STL.

Note: It is imperative that you adhere to the prescribed utilization of a Trie as outlined in the lab manual for your assignment. Failure to comply with this directive will result in a substantial deduction of points from your overall assignment score.

Additional Considerations

Code Inspection: Following the output evaluation, we will review your code to ensure it aligns with the provided notes and guidelines.

Code Comments: Include comments to explain your code logic and design choices.

Testing: Thoroughly test your program with various inputs to ensure its correctness.

No STL: Remember that the use of any C++ STL containers or algorithms is prohibited.

Performance Evaluation: For Lab 2, we will be checking the time complexity of functions you have designed for different commands.

Memory Leaks: We will be checking for memory leaks in Lab 2.

Use of AI Tools: If you utilized any AI tools in the development of your code, please include a comment in your source code indicating their use.

Code Compilation: Ensure that your code compiles without errors on ECE Linux servers (e.g., eceubuntu, ecetesla). We will make and test your code on these servers. If we cannot compile your code, you may receive a score of 0 for the marking.

Object-Oriented Design Principles:

- You must use proper Object-Oriented design principles in the implementation of your code.
- Create a design using classes that have appropriately private/protected member variables and appropriate public member functions.
- It is not acceptable to simply make all member variables public.
- You should consider separating the interface and implementation of class member functions by placing the declarations in a .h (header) file and the definitions in a .cpp (source code) file.
- Implement constructors for initialization and destructors for resource release in classes.