ECE250: Lab Project 3

Due Date: Saturday, November 19, 2016 – 11:00PM

1 Project Description

In this project you will implement a trie data structure using recursive programming. 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) sub-tree is not a null sub-tree, it represents a string that is the concatenation of string represented by the parent and the k^{th} letter of the alphabet (where a is the 0^{th} letter, b is the 1^{st} letter, and so on). Each node may or may not indicate that it is a terminal entry for a word. While a trie could be used to store hyphenated and capitalized words together with those with apostrophes, we will restrict ourselves to words made up of the twenty-six letters of the English alphabet.

For example, consider the sentence "The fable then faded from my thoughts and memory". If we wanted to put these nine words into a Trie, it would look like Figure 1. Only the root node explicitly shows its 26 sub-trees, each sub-tree being associated with a letter of the alphabet. In this example, 22 of the sub-trees of the root node are null, as there are no words that begin with, for example, 'b'. Four of the children of the root are not null: those representing the letters 'a', 'f', 'm', and't'. For the other nodes, only the non-null sub-trees are shown. These sub-trees represent words starting with these letters, respectively. Terminal nodes are represented as red circles.

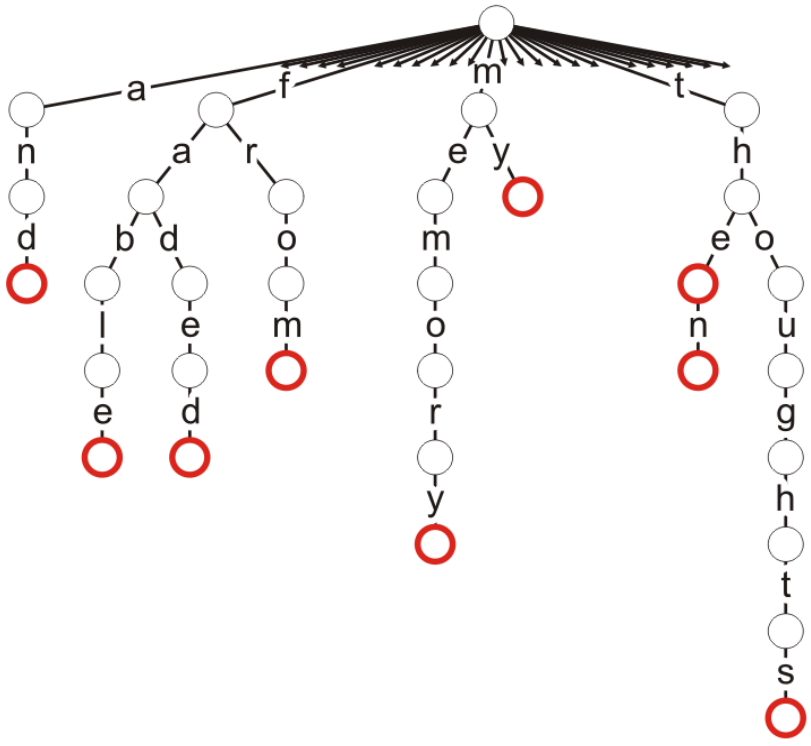

Figure 1. A trie containing the words in the sentence "The fable then faded from my thoughts and memory"

2 Characters and Strings

In this project you need to work with *characters* and *strings*. A string is a list of characters. In C, a string of n characters is an array of type char of size n + 1 or greater where the $(n + 1)^{th}$ character is the null character '\0'. Fortunately, in C++, there is a string class in the standard library. You can include this library with: *#include <string.h>*. As this is in the standard template library, the class is defined within the std namespace: *std::string*. To access the k^{th} character, use the index (similar to arrays). You can also use the *cctype library* to work with characters.

3 How to Test Your Program

We use drivers and tester classes for automated marking, and provide them for you to use while you build your solution. We also provide you with basic test cases, which can serve as a sample for you to create more comprehensive test cases. You can find the testing files on the course website.

4 How to Submit Your Program

Once you have completed your solution, and tested it comprehensively, you need to build a compressed file, in tar.gz format, which should contain the file:

- Trie.h
- Trie_node.h

Build your tar file using the UNIX tar command as given below:

- *tar –cvzf xxxxxxxx_pn.tar.gz* Trie.h Trie_node.h

where *xxxxxxxx* is your UW user id (i.e., jsmith), and *n* is the project number which is 3 for this project. All characters in the file name must be lowercase. Submit your tar.gz file using LEARN, in the drop box corresponding to this project

5 Class Specifications

In this project, you will implement two classes: *Trie.h* that contains the root node, from where every other node in the tree can be reached, and *Trie_node.h* that represents each node in the tree.

5.1 Trie.h

This is the class that implements a trie. In the following, the variable *n* represents length of a string; that is, the number of characters making up the string. This trie tree is not case sensitive; that is, the words “Hello”, “hello” and “HELLO” all represent the same word. In these examples, we converted all letters into lower-case; however, you may chose to convert all letters into upper-case—it works either way. The expected running time of each member function is specified in parentheses at the end of the function description (n indicates the number of characters in the string and N indicates the number of strings).

Member Variables

The *Trie* class has two member variables:

- A pointer to the root node, and
- An integer variable storing the number of words in the tree (that is, the size).

Accessors

This class has four accessors:

- *bool empty() const* - Return true if the trie is empty (the size is 0). (**O**(1))
- *int size() const* - Returns the number of words in the trie. (**O**(1))
- *Trie_node *root() const* - Returns a pointer to the root node. (**O**(1))
- *bool member(std::String str) const* - Return true if the word represented by the string is in the Trie and false otherwise. If the string contains any characters other than those of the English alphabet (‘A’ through ‘Z’ or ‘a’ through ‘z’), throw an illegal_argument exception. (**O**(n))

Mutators

This class has three mutators:

- *bool insert(std::string str)* - Insert the word represented by str into the tree. If the string contains any characters other than those of the English alphabet ('A' through 'Z' or 'a' through 'z'), throw an illegal_argument exception; otherwise if the string is already in the tree, return false; otherwise, return true (the insertion was successful). This is done by calling insert on the root, and if the root node is null, it will be necessary create an instance of the Trie_node class and assign it to the root first. (**O**(n))
- *bool erase(std::string str)* - Erase the word represented by str from the tree. If the string contains any characters other than those of the English alphabet ('A' through 'Z' or 'a' through 'z'), throw an illegal_argument exception; otherwise if the string is not in the tree, return false; otherwise, return true (the erase was successful). If the tree is empty, return false, otherwise this function calls erase on the root. If the word erased is the last one in the tree, delete the root node. (**O**(n))
- *void clear()* - Delete all the nodes in the tree. Again, if the tree is not empty, it should just call clear on the root and set the appropriate member variables. (**O**(N) where N is the number of words in the tree)

Please note, you are allowed to any additional member functions you wish; however, they should be private to the *Trie* class as they are not specified in this interface.

5.2 Trie_node.h

This class implements the nodes of trie. The expected running time of each member function is specified in parentheses at the end of the function description.

Member Variables

This *Trie_node* has two member variables:

- A pointer to an array of pointers to trie nodes, and
- A Boolean flag indicating whether a node represents a terminal node for a word.

There is also a constant static member variable CHARACTERS that is assigned the value 26.

Constructor

The constructor sets the pointer to children to null and sets the Boolean flag to false.

Accessors

This Trie_node class has two accessors:

- *Trie_node *child(int n) const* - Return a pointer to the n-th child. If the children array is empty, return nullptr; otherwise, just return children[i]. This member function will never be called with a value outside 0 to 25. (**O**(1))
- *bool member(std::string const &str, int depth) const* - The string being searched for is being passed recursively; however, as we go deeper into the tree, we must have access to the character corresponding to the depth. **You can assume that the characters are all alphabetical, as the Trie class should have checked for invalid characters.** The trie tree is not case sensitive, so you must map letters to the range 0 to 25. If we are at the end of the word, the is_terminal member variable will determine the appropriate return value; otherwise, we need to call member recursively or return false, as appropriate. You must return the appropriate values in all cases, including: when children are null, when the appropriate child pointer is assigned null, and otherwise. (**O**(n)).

Mutators

This class has three mutators:

- *bool insert(std::string const &str, int depth)* - Like member accessor function, we need to recurse into the tree. If we reach a node and we are at the end of the word we are attempting to insert, we need only check and possibly modify the member variable is_terminal to determine the appropriate return value. If we are not yet at the end of the word, we must recurse through the appropriate sub-tree. This may require first assigning an array of 26 pointers to Trie nodes to children in some cases, and it may require assigning a new Trie node to the k^{th} sub-tree of this array. (**O**(n))
- *bool erase(std::string const &, int depth, B, Trie_node *&ptr_to_this)* - Like member accessor function, we need to recurse into the tree. If we reach the end of branch of the tree before we get to the end of the word, it is clear the word is not stored in this tree and thus cannot be erased. If we get to a node when we are the end of the word, we must choose the appropriate course of action and the appropriate return value based on the value of is_terminal. If this node is a leaf node (children is assigned null), we should delete this node. While we are recursing back, if the children array became entirely unassigned as a result of our erase, the current node must also be erased. For example, in Figure 1, if the word "thoughts" was erased, then the nodes containing “o”, “u”, “g”, “h”, ”t”, and “s” must be deleted and the appropriate sub-child of the node containing “h” must be set to null. (**O**(n))
- *void clear()* - Calls clear on all sub-trees and deletes this node.