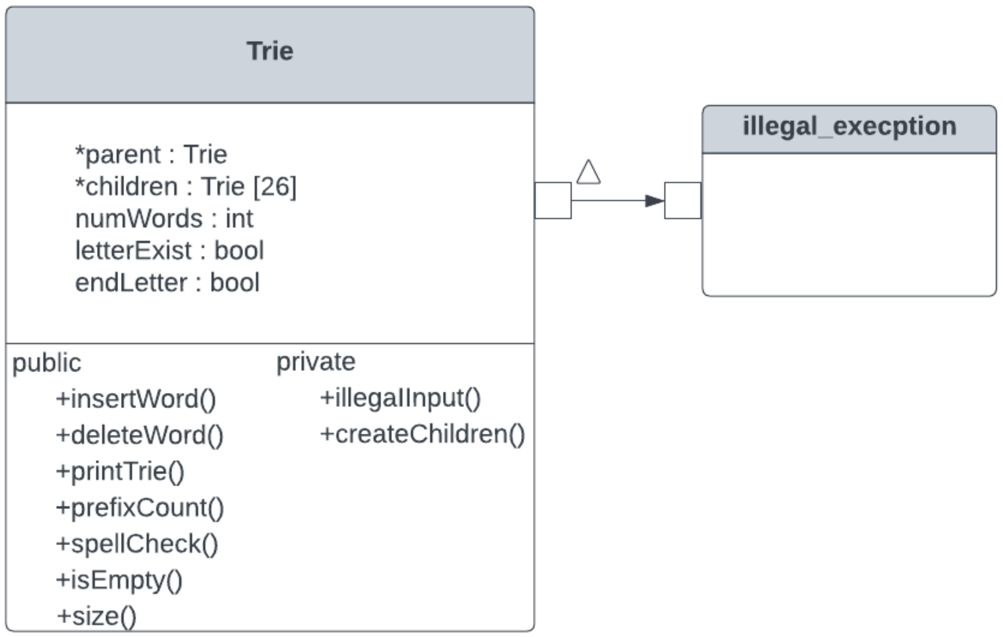
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 250 – Project 3
Trie Design Document

Hossein Molavi : hmolavi

Structure - UML Diagram



Annotation style taken from https://www.omg.org/spec/UML/2.4/Superstructure/PDF Figure 12.122 (page 416) - Exception pin notations.

Overview of Classes

Trie Class

Constructor and Destructor

The constructor is empty. The destructor is simply delete[] children pointer (*children*).

Member variables:

Each Trie consists of a *letterExist* (type boolean) defaulted to false holding true only if that letter exists. An *endLetter* boolean variable which determines if the letter is at the end of a word, an integer, *numWords*, holding the number of words after the letter which includes if word ends at current *letter*, a *parent* pointer to the parent Trie and a *children* pointer which points to an array of 26 Trie holding the child Tries all initially set to null. Please note that for root Trie the *parent* is always nullptr.

Member functions: Time Complexity

int insertWord (std::string n) O(N)

Inserts words into the Trie and returns a 1 or 0 if procedure was successful or not. Recursively call the *insertWord* function of its children N times where N is the number of valid letters in the given word. Base case runs if word length is 1 and not called by the root Trie; base case sets the *letterExist* variable to true if only it was previously false, or sets *endLetter* to true if the letter already exits. Checks to see if the input is valid if parent pointer is root and creates children node if needed by use of *createChildren*. Finally adds to the *numWords* variable if the insertion was successful.

int deleteWord (std::string n) O(N)

Searches for and deletes a set of characters (a word) from root Trie. Return conditions are same as *insertWord*. Function recursively calls the Childrens' *deleteWord* function for every letter in the given word (valid letters only otherwise returns zero). Base case sets *endLetter* to false only if the word exists in the Trie and the word is a prefix of another word (eg HAND and HANDY) or the word connects halfway to another, otherwise the function will cut off the unnecessary Trie by deleting that Tries' *children* and setting that *letterExist* to false. An example would be, assuming the words HEAD and HEEL and we would like to delete the first, function will travel to the connecting letter 'E' and delete the 'A's *children* and setting 'A's *letterExist* to false. Runtime is O(N) where N is the number of valid characters in n.

int prefixCount (std::string n) O(N)

Given a string, returns the number of words which start from it, including the endLetter. Achieves this by traveling down the Trie, to the end of given string, and returning the *numWords* of that last letter, looping itself N times where N is the number of valid characters in given string (if input is not valid returns zero).

void spellCheck (std::string recursionInput, std::string constantInput) O(N)

Given a string, prints the of words which start from it if the word is not already in the Trie. Achieves this calling itself with one less character. Both the recursionInput and constantInput are initially the same value, the recursionInput will decrease by one character every function call but the constantInput will be the same. If the next letter does not exits in the Trie it will then subtract the constantInput and recursionInput to get the prefix and it will call the *printTire*. Having to call the function with two same values is not ideal but it is needed as the return parameter does not exits. Function runs on a time complexity of N where N is the number of valid characters in the input (ie characters which already exits in the Trie).

void printTrie (std::string prefix, int& wordCount) O(M)

Given the characters from the Trie to the root (prefix) and the number of words in that Trie it will travel to all of the *children* which have a letter (ie *letterExist* is true) and print them. The function achieves this by recursively calling itself with one extra letter corresponding to the index in the array and printing the *prefix* whenever the *endLetter* variable is true. The wordCount is needed to not print extra space at the end of the line. Runtime is O(M) where M is the number of words in that Trie, O(1) if the Trie is empty.

int size () O(1)

Returns the *numWords* of the Trie it is called from. Constant runtime.

bool isEmpty () O(1)

Returns true if the *children* of that Trie is nullptr, false otherwise. Constant runtime.

private : bool illegalInput (std::string n) O(N)

Called at the beginning of the insertWord, deleteWord and prefixCount. Iterates through the characters and returns true if the input is illegal is not valid, false otherwise. Runtime is O(N) where N is the number of valid characters in n.

private : void createChildren () O(1)

Sets the *children* pointer to an array of 26 Trie. Constant runtime.

Illegal_excpetion Class

Created and used by the *illegalInput* function from Trie class for try and catch. No variables and functions, no destructor and an empty constructor.

Why one class (Trie) ?

By creating one class, all of the main functionalities can be done by recursion. The constructor and destructors are much easier to read/readable (one line each) and more importantly, all of the private variables will be in one class which means there is no need for setters.

Why bother counting for printTrie ?

I have a belief whenever I am programming and I stick to it as I learned it in the 'Software Engineering at Google' book. The belief is to always think of the future and how to create maintainable code which is easy to change if required days/weeks/months or even years later. My implementation is perfect at the moment and does not print an extra white space at the end of the final word in the Trie, however if for whatever reason there is an issue where the numWords is not valid we can simply change the function by removing three of its eighteen lines and change the two times the program is called from spellcheck to match the appropriate function call. The present solution is the perfect and maintainable solution, the other approach is satisfactory but not maintainable, hence I chose this method.

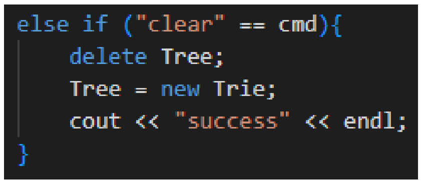
Performance

All of the time complexities are as required in the project description, I used different letters as it makes more sense at least to me, but they are essentially the same.

Sources & Inspirations

None, it took many many many trials and errors to get here.

**Please note that a clear function does not exist, the clear command simply deletes the tree and replaces it by a new one.



```
else if ("clear" == cmd){
    delete Tree;
    Tree = new Trie;
    cout << "success" << endl;
}
```

Delete will call the destructor which calls delete[] children, a repeating loop until all nodes are deleted. Running time of O(n) where n is the number of existing Trie (ie size of Trie).