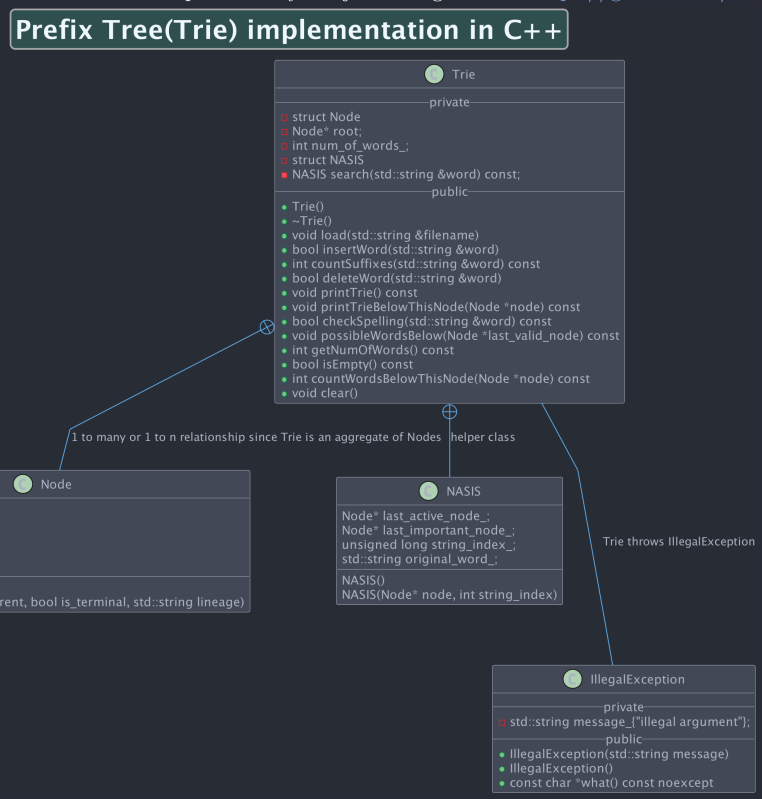
Abstract

This is the design document of Chaitanya Sharma for Project 3 for ECE 250's Winter 2023 offering.

1 Introduction

My program consists of a major class Trie, and two nested classes inside that ,namely Node and NASIS. While the Trie class is an aggregate of the Node class, the NASIS class is a helper **information** class which allows me to create a single search function for insertion and deletion of nodes. Also, I've created a trietest.cpp file to contain the driver code which runs a while loop to parse inputs. There is also an excetion class called IllegalException which I've inherited from the std::exception class. I attribute the latex template which I've used for this design document to myself as I've used it in the past for my other projects in this course.
I've tested all edge test cases possible and also used the test cases provided by the jekelautograder on https://github.com/JZJisawesome/ece250-testcases.



2 Class Structure

2.1 class IllegalException

I'm using a default constructor and a parameterized constructor for this class. Though I'm not using the parameterized constructor in my code, I've included it for future use. This is a simple exception class which I've inherited from the std::exception class. It has a private member variable message_ which is a string which stores the message to be displayed when the exception is thrown. It has a constructor which takes a string as an argument and assigns it to the message_ variable. It also has a default constructor which assigns the default message to the message_ variable. It has a what() function which returns the message_ variable as a const char*.

2.2 class Trie

The constructor for this class just initializes the root_ pointer and the num_of_words_ variable to nullptr and 0 respectively. The destructor for this class just deletes the root_ pointer, which indirectly deletes all the nodes in the trie. This is the major and the only top level class in my whole code. The most important member object of this class is a root_ of type Node which is a pointer to the root node of the trie. And the only other member object is a num_of_words_ which is an integer which stores the number of words in the trie.

2.2.1 class Node (private nested class)

This is a private nested class inside Trie class. A simple class which has a static array of 26 pointers to other nodes i.e. its children, initialized to nullptr, has a parent pointer, an inherent character value, a lineage which contains a string of all nodes from root node to itself, and a boolean variable which tells whether the node is a word ending node or not. It's destructor is created such that it deletes all the nodes which are children of itself, and then deletes itself.

2.2.2 class NASIS (stands for Node And String Index for Searching) (private nested class)
★ a special feature of my code ★

This is a quintessential but convenient helper class which I've created to help me with the search function. It has 4 member objects, namely last_active_node_, last_important_node_, string_index_ and original_word_. The reason I implemented this is because traversal in trees is extremely common, as well as expensive, hence it should make sense to make a single function for traversing the prefix trie, I had the same idea for the Linked List project as well as for the Hash Table Project, but they were both linear structures(in the sense of the base structure), but this project is not, hence we need more information to be returned, since I would always use a word to find the appropriate node for mutation and accession, I have the string index, which returns the character at the last valid node when traversing in the path of the word. last_important_node_ which either has more than one child, or is a word ending node before traversing to the possible last_active_node_, which is the last possible node with the same lineage_ as the word, and the original_word_ which is the word itself. More info in the search function subsubsection.

2.2.3 NASIS search(std::string word) const

This function is aimed at reducing redundant traversals in my code, and handles the search for a word operation in the Trie. It takes in a std::string word, and from the root, traverses the trie parallel to what character appears in an increasing index value in the word. Along the way, if it observes any node with more than one child, or a word ending node, it rewrites the pointer node in last_important_node_. If it sees that there is no child node for the current character, it returns the NASIS object with the last_active_node_ as the last node which has the same lineage_ as the word, and the string_index_ as the index of the last character in the word which is valid. The information aims to make a single function for traversing the trie, and then using the information to perform the required operation. This function runs in O(n) where n is the number of characters in the word.

2.2.4 void load(std::string filename)

This function uses the C++ standard library std::ifstream class to read the file and load the words into the trie using the insert function repeatedly on the appearing words. This function outputs success no matter what. This function runs in O(n) where N where N is the number of words in the file though it is directly dependant on the number of words, but since it callse the insert function which runs in O(n) where n is the number of characters in the word, the time complexity is the same. "load"

2.2.5 bool insert(std::string word)

This function inserts a word into the trie, it uses the search function to get the appropriate information from the NASIS object, then, if the string_index_ is equal to the length of the word, it checks if the last_active_node_ is a word ending node, if it is, it returns false, otherwise, it sets the is_word_here_ boolean to true, and increments the number of words in the trie, and returns true. If the string_index_ is not equal to the length of the word, it gets the substring of the word from the string_index_ to the end, and then traverses the trie from the last_active_node_ and creates new nodes for each character in the substring, and then sets the is_word_here_ boolean to true, and increments the number of words in the trie, and returns true. This function runs in O(n) where n is the number of characters in the word. Also, this function, if comes across a character outside the range of capital english alphabets, it throws an exception of type IllegalException. "i"

2.2.6 int countSuffixes(std::string word) const And int countWordsBelowThisNode(Node *node) const

The function countSuffixes counts the number of suffixes of a word in the trie, first it traverses the trie parallel to the word, and if it finds a node with no child for the current character, it returns false. Otherwise it completes the loop, and then uses the countWordsBelowThisNode function to count the number of words below the last active node, which is the last node with the same lineage as the word. This function runs in O(N) where N is the number of characters in the word. The reason this function runs in O(N) and not O(n) (where n is the number of characters in the word) is because the average word in the English language has between 4-7 characters, and even if we assume that the word has 1000 characters, which situation is very unlikely, O(1000N) is still O(N). Also, it can be safely assumed that M, the maximum characters in a word, $M << N$ and that the number of characters in the trie are a constant multiple of the number of words. The function countWordsBelowThisNode counts the number of words below a node, it uses recursion to traverse the trie, and if it finds a word ending node, it increments the count, and then returns the count. This function runs in O(N) where N is the number of words in the trie. Also, this function, if comes across a character outside the range of capital english alphabets, it throws an exception of type IllegalException. "c"

2.2.7 bool deleteWord(std::string word)

This function removes a word from the trie, by considering all three cases (1) the word doesn't exist, (2) the word exists but has other suffixes, and (3) the word exists and has no other suffixes. First, it uses the search function to get the appropriate information from the NASIS object, then it checks if the string_index_ is equal to the length of the word, if it is, it checks if the last_active_node_ is a word ending node and that it has children, if it is, it sets the is_word_here_ boolean to false, and decrements the number of words in the trie, and returns true. If the string_index_ is not equal to the length of the word, it returns false. If the last_active_node_ is a word ending node, it checks if all children of the last_active_node_ are null, if they are, it deletes the children, and then traverses the trie from the last_active_node_ to the last_important_node_ , and deletes the nodes that have no children, and returns true. Otherwise, it returns true. This function runs in O(n) where n is the number of characters in the word, since it is directly proportional to the number of characters in the word. Also, this function, if comes across a character outside the range of capital english alphabets, it throws an exception of type IllegalException. "e"

2.2.8 void printTrieBelowThisNode(Node *node) const

This function prints the trie below a node, it uses recursion to traverse the trie, and if it finds a word ending node, it prints the lineage of the node, and then returns. This function runs in O(N) where N is the number of words in the trie. It traverses the trie in a Depth First manner, and prints the words in the trie in alphabetical order. "print"

2.2.9 void printTrie()

This funciton just calls the printTrieBelowThisNode function, and prints a new line if the trie is not empty. This function runs in O(N) where N is the number of words in the trie, since it just calls the printTrieBelowThisNode function. "print"

2.2.10 bool checkSpelling(std::string word) const

This function checks if a word is spelled correctly, it uses the search function to get the appropriate information from the NASIS object, then it checks if the string_index_ There are three cases to consider, (1) The word exists fully, (2) the word exists partially i.e. a subsequence exists but is not a word. (3) the word does not exist at all i.e. even the root doesn't have the first character of the word. In case (1) it returns true simply, in case(2) it gets the last valid node, and then checks if the last valid node has more than one child, if it does, it calls the possibleWordsBelow function, and returns false. In case(3) it simply returns false. This function runs in O(N) since it is directly dependant on the length of the word which as we proved before has is at most N. "spellcheck"

2.2.11 void possibleWordsBelow(Node *node) const

This function prints all the possible words below a node, it uses recursion to traverse the trie, and if it finds a word ending node, it prints the lineage of the node, and then returns. This function just calls the printTrieBelowThisNode function on all the children of the given node. This function runs in O(N) where N is the number of words in the trie, since it just calls the printTrieBelowThisNode function. "spellcheck"

2.2.12 void clear()

This function clears the trie, it deletes the root node, and creates a new root node, and sets the number of words to 0. This function runs in O(N) since it just deletes the root node which subsequently deletes all the nodes in the trie. "clear"

2.2.13 bool isEmpty() const

This function returns true if the trie is empty, and false otherwise. This function runs in O(1) since it just checks a single variable. "empty"

2.2.14 int getNumOfWords() const

This function returns the number of words in the trie. This function runs in O(1) since it just returns a single variable. "size"