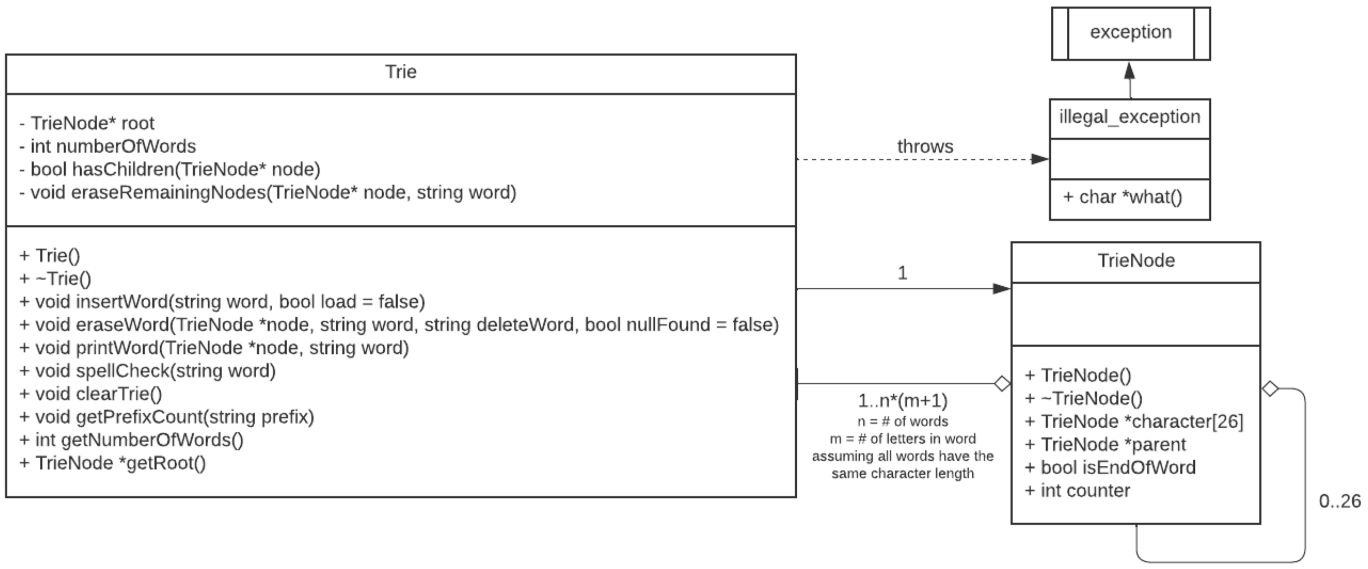
ECE250 Project 3 Design Document

Brandon Vo
March 17, 2023



Trie
- TrieNode* root
- int numberOfWords
- bool hasChildren(TrieNode* node)
- void eraseRemainingNodes(TrieNode* node, string word)
+ Trie()
+ ~Trie()
+ void insertWord(string word, bool load = false)
+ void eraseWord(TrieNode *node, string word, string deleteWord, bool nullFound = false)
+ void printWord(TrieNode *node, string word)
+ void spellCheck(string word)
+ void clearTrie()
+ void getPrefixCount(string prefix)
+ int getNumberOfWords()
+ TrieNode *getRoot()
throws
exception
illegal_exception
+ char *what()
1
TrieNode
+ TrieNode()
+ ~TrieNode()
+ TrieNode *character[26]
+ TrieNode *parent
+ bool isEndOfWord
+ int counter
1..n*(m+1)
n = # of words
m = # of letters in word
assuming all words have the same character length
0..26

TrieNode Class

The TrieNode class is used to hold the characters for building the word in the Trie. Each node contains a pointer to an array of size 26 for characters A to Z, a pointer to the parent node, a boolean flag to mark the end of a word in a Trie, and a counter to keep track of the number of words that have the prefix up to that node. The number of TrieNode objects in this program range from 1, which is the root, to n * (m + 1), where n is the number of words and m is the length of the longest word. In the constructor of this class, it initializes each character index and the parent to nullptr. The end of word flag is also set to false. The destructor iterates through each character index from 0 to 25 and deletes that index if it exists.

Trie Class

The Trie class is used to build the prefix tree using the TrieNode object as a private variable defined as the root. To keep track of the number of words in the Trie, an integer counter is used. There are also two private utility functions for helping erase remaining nodes from the Trie (*eraseRemainingNodes*), and checking if a node has children (*hasChildren*). The public member functions are used when executing the required commands for this project. These consist of insertWord, eraseWord, printWord, spellCheck, clearTrie, and getPrefixCount, which are all void functions since they do not return a value. There are also *getNumberOfWords* and *getRoot* functions to get these corresponding values from Trie.

illegal_exception Class

The illegal_exception class is thrown when a non-uppercase English alphabet letter is inputted. This class inherits from the exception library and uses exception::what() to return a null terminating string. The class is used in the i, c, and e commands by using a try/catch statement to attempt to run the command, then throwing this class which calls the exception::what() function and prints “illegal argument” if an error has been caught.

Command Analysis

i: To insert a word into the Trie, we use the *insertWord* function which takes in a string word input. There is also a boolean parameter called load, which default value is set to false, to determine if we are using *insertWord* for loading words from the corpus text file. This is because we only want to print a single “success” from using the load command, rather than success being printed from every word from the text file inserted. This function is handled in a try/catch block to ensure the input is a valid uppercase English letter.

The *insertWord* function works by first assigning the root node to a pointer node called current. First we loop through the length of the word and check each character to see if it is not a valid input by using the !*isupper* function. If it is not valid, illegal_exception will be thrown. In this loop, we also traverse through the path of the characters in word if possible. This is so we can determine if the word already exists in the Trie. If we find that one of the nodes in our path don't exist, we know that this is not a duplicate insert. In the second loop, we go through the length of the word again and define an index from 0 to 25 by subtracting the current character of the word by 'A', which has an ASCII code value of 65. We use this index for searching the current node's character value at our defined index and checking if it is equal to nullptr. If it is, we know to create a new node at this index and define the parent as our current node. After this check, we traverse the current node to the next node at the correspond letter, increase a counter to keep track of how many times this node has been used in other words, and loop through the rest of the word. Once we are finished looping through the word, we check if our current node already has the isEndOfWord flag. If it does, the word already exists in the Trie so we print "failure". Otherwise, we set the isEndOfWord flag to true and increase the numberOfWords counter.

The time complexity of this command is O(2n)=O(n), where n is the length of the word being inputted. The *insertWord* function uses two loop that iterates through the entire word length, where n is the number of characters in the word. We use two loops because we need to ensure that the input is valid before inserting any new nodes into the Trie. If we used one loop to check each character and immediately insert that node, we may run into a case where an input is valid at the start of the word, but contains invalid characters later in the word. This would cause the Trie to have dangling nodes that are not associated to any words. We also check if the word already exists to properly increase our counter in the next loop.

c: To count the number of words with a given prefix, we call the *countPrefix* function wrapped within a try/catch block to handle inputs that are not English uppercase letters. This function requires one string parameter for the prefix we want to count the words of. The function iterates through the length of the prefix input and determines if it is a valid input similarly to how we checked it in *insertWord*. We also use the same method of finding the index by subtracting the current character by the ASCII value of 'A', then checking if the current node at that index doesn't exist. If it doesn't, we know that the given prefix is not in the Trie. Otherwise, we continue traversing through the entire prefix length. Once we exit the loop, we use the counter variable node property we kept track of in the *insertWord* function to get the number of times this node has been used in other words, and print it.

With the assumption that there is a maximum length, n, for a prefix word, and n<<N, this function would have O(n)=O(N) runtime. Any O(n) operation is automatically O(N) with this assumption, since big O notation expresses the upper bound of the function's growth rate. With n<<N, the growth rate is determined by N since it is the dominant term.

e: The erase command is wrapped in a try/catch block to ensure the input is valid. It uses the same process as the i and c commands. The command calls a function called *eraseWord* which starts at the root and gets the index of the character from 0-25 for searching the nodes. We then slice the first character of the word using the *substr* function. If there are still words left in the string, we traverse the node pointer and recursively call the *eraseWord* function with our newly sliced string. While iterating, if we find that the current node is a nullptr, we mark a boolean called nullFound to true and prevent any more node traversing. Once this is marked, we continue iterating through the rest of the word to see if illegal_exception should be thrown instead. Since exceptions have higher priorities than failures, we use the nullFound boolean to make sure we return a print failure if the input ends up being valid.

Once we iterate through the entire word, we check if the current node has the end of word flag. If it does, we unmark this flag, decrement the number of words and node counter, and check if the node does not have children by using the *hasChildren* function. This function has constant runtime, as it is iterating through a defined interval of 0 to 25 to check for all indexes in the character array, and returning true if we find a node. Otherwise, the function returns false. If we find that the final node does not have children, we delete it. If our current node is not the root, we redefine the node to the parent, and call the *eraseRemainingNodes* function which is used to recursively traverse back up the tree and delete the nodes with our word input that do not have children. For this, we use a similar process by instead using the *pop_back* function to remove the last letter from our word input, defining the character index in the node with our word input by reading it in reverse order, and checking that node with the *hasChildren* function. If it does not have children, we delete it. Then we traverse to the parent and recursively call again until we either run out of letters to pop back.

The runtime of the erase command is O(n), where n is the number of characters in the word. To find the end node in the given word, it takes O(n) time. This recursively calls *eraseWord* on each character of the word. If it reaches the end, it recursively calls *eraseRemainingNodes* with the same word input, which also takes O(n) time to traverse back up the n nodes. Therefore, the overall time complexity is O(n) + O(n) = O(n).

p: The print command uses the *printWord* function to print all words in the Trie. it takes in a TrieNode parameter and a string for the word to be printed. When this command is ran, it uses the *getNumberOfWords* getter from the Trie object. If there are 0 words, we will just continue. Otherwise, there are words in the Trie and we call *printWord* with the root of the Trie using *getRoot,* and an empty string. In this function, we loop through all indexes of each node from 0 to 25 until we find a character. Once a character is found, we add the ASCII value of 'A' to the current loop index and add this to a string with our current word. Then we recursively call the function with the current node and the string we are building to perform depth-first traversal. Once we find that our current node has the endOfWord flag, we know to print the word we have been appending to. This process repeats for the rest of the the words.

Using the assumption that there is a limit MAX_CHAR, for which no word can be larger than, the time complexity of the print command is O(MAX_CHAR*N) = O(N), where N is the number of words in the Trie. With each node containing a character in a word, and each word having the MAX_CHAR limit, the maximum number of nodes that can be visited in a recursive traversal is that limit. This is because we need to traverse through each node in the Trie exactly once. The function concatenates characters and recursively calls itself, which it's time complexity is proportional to the maximum length of a word. Since we are only considering printing a single word, the runtime would be O(N).

spellcheck: The spellcheck command uses the *spellCheck* function to either determine if a word is spelled correctly, or offer suggestions if the word is spelled incorrectly. This function only takes one parameter, which is a string value of the word we are spellchecking. For this function, we define a current pointer to the root and an empty string to build our suggestion. We then iterate through the length of the word and get the index 0-25 through the same methods as all previous commands. Next, we check if the the current node doesn't exist. If it doesn't we call the *printWord* function to print all of the words at our current node and string suggestion we have built. Otherwise, we traverse our current pointer to the next node and build the suggestion string with the current character. Once we exit the loop, we will find that if our current node has the endOfWord flag, it means the input word was spelled correctly, so we print "correct". Otherwise, we call the *printWord* function with our current node and the suggestion string we have built up.

With our assumption that a word has a maximum length, the runtime of using the spellcheck command is O(N), where N is the number of words in the Trie. As mentioned in the print command, the time complexity of *printWord* is O(N). Since we are assuming that there is a maximum length of a word, this operation is constant. Overall, the spellcheck runtime would be O(N+1) = O(N), where N is the number of words in the Trie.

empty: The empty command makes use of the *getNunberOfWords* function in our Trie class. In our trietest.cpp test driver, we simply output "empty " + the boolean value of if *getNumberOfWords* is equal to 0 (outputs 1 if true or 0 if not true). The runtime of this command is constant O(1) since we are only using a getter function to get a value.

clear: The clear command deletes all nodes in the Trie by using the *clearTrie* function. This uses a loop that iterates from 0 to 25 and checks if the root contains a character. If it does, it will delete the root and call the destructor. The destructor essentially does the same thing, but to it's child nodes. Every time it deletes a character in the Trie, it recursively calls the destructor to delete the remaining nodes. The runtime of using this command is O(N), where N is the number of words in the Trie. This is because it needs to visit each node, search for all 26 characters in that node (single operation and independent from N), and delete the rest of the nodes that form the word.

size: The size command also makes use of the *getNumberOfWords* function. In our trietest.cpp file, we simply print "number of words is " + the value of *getNumberOfWords*. The runtime of this command is constant O(1) since we are only using a getter function to get a value.