CIT 594 Final Project Design Document

Dingyi Shen, Jinlu Ma, Yangjia Chen, Yuju Kuo

EXECUTIVE SUMMARY

An autocomplete System

For the final project, our group will implement a simple autocomplete system. The system will make 5 autocomplete suggestions based on a string that the user typed in. The user may type in either a few letters, or a word, or several words separated by a white space. The 5 suggestions, which are phrases or sentences, will be selected based on the frequency that they are used in our training file. The user may continue entering up to 7 words, or end the program.

Our design approach

The data structure we chose is a Trie, which is a tree-like data structure: we will store one letter of a word in each wordTrieNode, and one word of a sentence in each sentenceTrieNode. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree. The way we implement the autocomplete system using TrieNode and Trie interfaces represents a composite design pattern due to the fact that we treat word Trie and sentence Trie in a similar way since they both implements the Trie interface, thus having similar operations.

We will use 4 interfaces to help us maintain a clean and structured design:

- 2 interfaces responsible for the *operations* of our data structure
  - Trie Node: trie nodes are the units that consist of the Trie data structure
  - Trie: a tree data structure
- 2 interface responsible for the *actions* that our program will perform
  - FileParser: used for parsing training files to create the initial Tries
  - IUserInterface: includes the actions to populate Trie and the main method

Besides the 3 interfaces, we will have 4 concrete classes that implement the Trie Node and Trie interfaces (word and sentence respectively), and 1 more concrete class to implement the actions of our program.

Ensuring a smooth implementation in the future

With our design document listing out the clear steps of how the program works, reasoning for our design pattern and data structure choice, as well as a detailed class diagram, we have set up the cornerstone for us to implement the system in the future.

Problem description and usage scenarios

We want to make searching and typing easy - when a user types into a search bar some letters or words, we want to save users time and effort by showing them possible phrases and sentences that they may have in mind directly without them typing in the whole sentence letter by letter.

The purpose of our project is to autocomplete the phrase/sentence that a user types into the console by giving 5 suggestions below the user-typed string. A detailed description of steps/ usage scenarios can be found below:

- Greetings
    - When the user starts running the program, we will print out a greeting and instructions on how to use the system
    - “Welcome! Please type in a sentence or phrase to see autocomplete suggestions”
    - “You may click Enter/return to view suggestions at any time. To end the sequence, please type #”
- First word
    - Assuming the user wants to type in “University of Pennsylvania”, as the user types in letters in “Uni”, he/she has the option to click Enter
    - If the user clicked Enter, a list of 5 suggestions will be given below the typed string. E.g., “University of Pennsylvania”, “Universal Studio”, “Universe” etc.
    - The suggestions will be selected based on the 3 most used words starting with “Uni”, then check the mostly used 5 phrases/sentences that start with the 3 words
        - This will be achieved by first searching through the wordTrie to find words that start with “Uni” and find the 3 highest frequency words
        - Then search through the sentenceTrie to find sentences that start with each of the 3 words
        - For each of the 3 words, put all of the possible sentences by traversing through their subtree in the sentenceTree in a priority queue of non-increasing order based on frequency usage
        - Grab the 5 highest frequency sentences from the priority queue, which will be our suggestions
        - The final suggestions will be the 5 most frequent sentences out of all sentences that start with the 3 mostly used words we found in substep 2
    - The words’ and phrases’ usage initial frequency will be calculated and extracted from a training file we import into our program
    - If the user typed in a complete word already, for example, “university”, then the program will still show the highest frequency phrases starting with the word “university”

        - The final suggestions will be the 5 mostly used sentences out of all sentences starting with the 3 mostly used words we found in substep 2
- Subsequent words
    - To continue, the user can type number 1-5 to select a suggestion, or 0 to not select any
    - If the user types 1-5, the console will show the phrase that the user selected and the user can continue typing after the phrase. Next time when the user wants to see suggestions, the program will treat this new phrase as the prefix to search for suggestions
    - If the user types 0, then the console will show the phrase they entered before so that the user can continue entering to the same phrase and click to see suggestions at any time
- End the sequence
    - When the user has the phrase he/she wants and wants to end the program, he/she can simply type "#"
    - This will increase the final sentence's frequency by 1 in the sentenceTrie
    - This will reset the program and user can now start entering a new phrase to repeat the steps 2-4

In summary, a user can:
- Type in either letters or words
- Click "Enter" to view suggestions
- Type "#" to end tying

The program can show autocomplete phrase suggestions, or end the sequence and start over.

Design pattern

We will use composite Design Pattern, and there are four participants in our Composite Pattern:
- Component
    - Declares the Trie interface for objects in the composition and for accessing and managing its child components – WordTrie and SentenceTrie
    - Implements the default behaviors - add and search from the interface are common to all classes as appropriate
- WordTrie
    - Defines the behaviors for words in the composition. It represents word trie objects in the composition.
- SentenceTrie
    - Stores the sentence components and implements child related operations in the component Trie interface.

- User
    - Manipulates the objects in the composition through the component Trie interface.

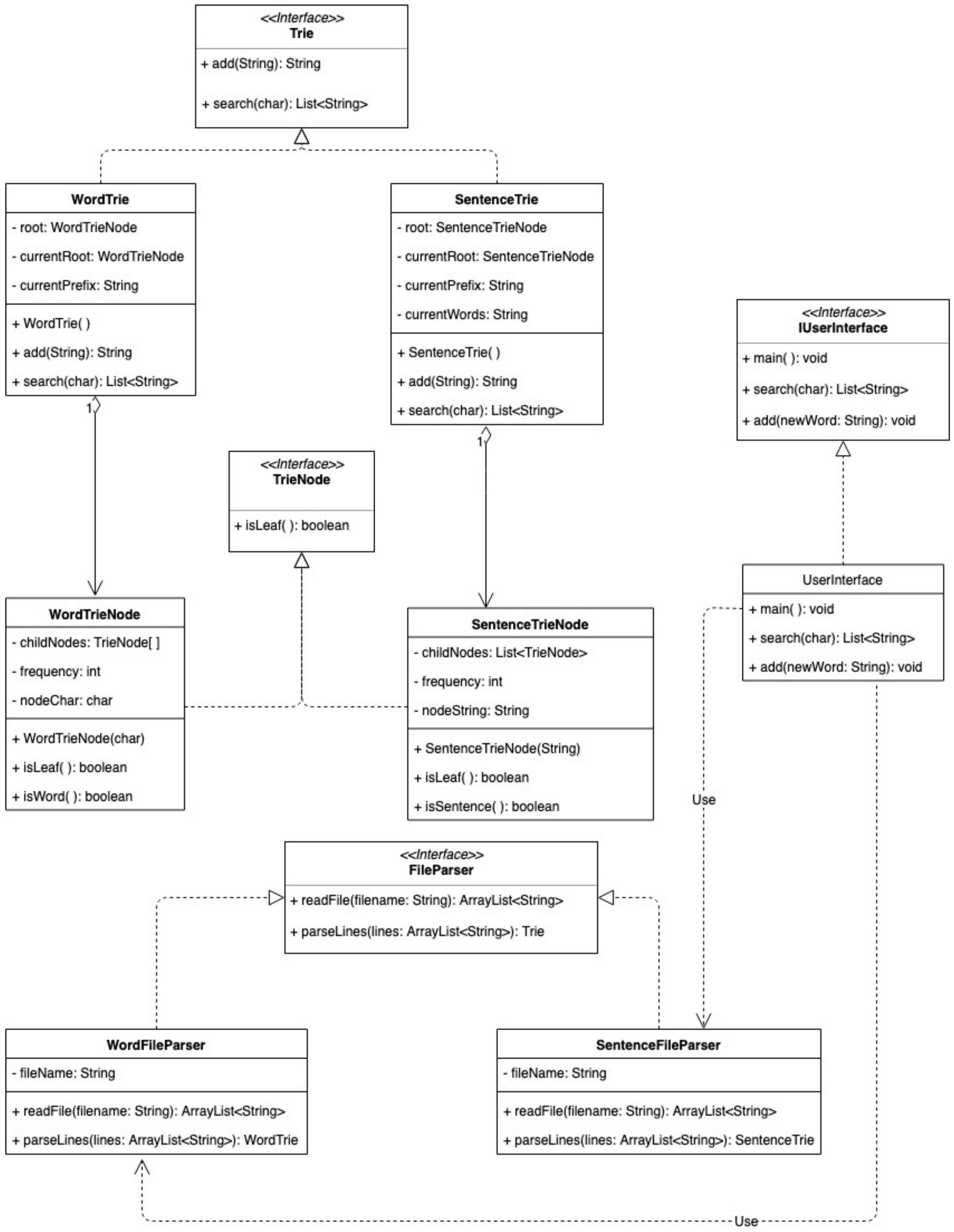
User uses the component class interface Trie to interact with objects in the composition structure. If the recipient is a word, then it forwards the request to its wordTrie components. If it is a sentence, the request is forwarded to sentenceTrie components. Both wordTrie and sentenceTrie have additional variables and operations.

Data Structure

Listed below are the data structures we plan to use in our implementation:

- List
    - We will use a List of TrieNode to store the children characters and words (possible next character/word) of a character or a word in the trie.
- Priority Queue
    - When the program pulls out all the possible next letters/words, the search results will be put in a priority queue sorted by frequency of use. By doing so, the program can retrieve the 5 highest frequency characters/words as suggestions.
- Trie
    - We will have two Tries for words and sentences respectively. Each Trie has an empty root node, and each of its children nodes is a possible next letter/word value.
    - Both Tries will have the “add” and “search” methods. It will also have the current String as an instance variable, which will help us index the current search node.
    - The frequency of a word/sentence will be saved in the node. Once a sentence is selected, its frequency would increase by 1. If a sentence that the user typed in cannot be found in the sentenceTrie, then we will create a new path to the new sentence in the Trie.

Class Diagrams



<<Interface>>
Trie
+ add(String): String
+ search(char): List<String>
WordTrie
- root: WordTrieNode
- currentRoot: WordTrieNode
- currentPrefix: String
+ WordTrie( )
+ add(String): String
+ search(char): List<String>
SentenceTrie
- root: SentenceTrieNode
- currentRoot: SentenceTrieNode
- currentPrefix: String
- currentWords: String
+ SentenceTrie( )
+ add(String): String
+ search(char): List<String>
<<Interface>>
IUserInterface
+ main( ): void
+ search(char): List<String>
+ add(newWord: String): void
1
1
<<Interface>>
TrieNode
+ isLeaf( ): boolean
UserInterface
+ main( ): void
+ search(char): List<String>
+ add(newWord: String): void
WordTrieNode
- childNodes: TrieNode[ ]
- frequency: int
- nodeChar: char
+ WordTrieNode(char)
+ isLeaf( ): boolean
+ isWord( ): boolean
SentenceTrieNode
- childNodes: List<TrieNode>
- frequency: int
- nodeString: String
+ SentenceTrieNode(String)
+ isLeaf( ): boolean
+ isSentence( ): boolean
Use
<<Interface>>
FileParser
+ readFile(filename: String): ArrayList<String>
+ parseLines(lines: ArrayList<String>): Trie
WordFileParser
- fileName: String
+ readFile(filename: String): ArrayList<String>
+ parseLines(lines: ArrayList<String>): WordTrie
SentenceFileParser
- fileName: String
+ readFile(filename: String): ArrayList<String>
+ parseLines(lines: ArrayList<String>): SentenceTrie
Use