Computer Programming 3 – Assignment 1 Development and Testing

J. Francis

# INTRODUCTION

T

HIS report will detail the development, testing and functionality of the “SuffixTrie” program noting any errors, complications and design decisions made throughout this process.

# DEVELOPMENT

The basic structure of this program was taken from LAB2 task B. The first step in developing the SuffixTrie program was to construct the “**SuffixTrie**” class and implement the “**insert**”, “**getNode**” and “**readInFromFile**” methods. This was done by copying over the code from the Trie class from LAB2 and modifying it as necessary. First implemented was the “**readInFromFile**” method.

### **SuffixTrie** Class

#### “**readInFromFile**” Method

The “**readInFromFile**” method, for the most part, was already complete in terms of actually obtaining input from a text file. The method of parsing the input data was modified such that each line of text was split wherever a full stop, exclamation mark or question mark occurred and stored in an array of strings representing individual sentences. Each individual sentence within the array was then inserted into the SuffixTrie via the “**insert**” method. Additionally, the “**readInFromFile**” method needed to pass the sentence number to the “**insert**” method such that searches could be performed when implementing later functionality.

#### “**insert**” and “**insertHelper**” Methods.

The next method to be implemented was the “**insert**” method. The structure for this method was, again, taken from the Trie class and modified as necessary. The “**insert**” method was first modified to accept a sentence number as a parameter. The sentence, passed as a string, was then converted to a list due to unforeseen errors that will be discussed later. This newly created list was then passed to the “**insertHelper**” method for the recursive insertion. The code from the “**insert**” method from the Trie class was moved into the “**insertHelper**” method and modified to operate with a list instead of an array. The first major change was that when the sentence was inserted, the first character needed to be removed and the method called again for the new suffix. This was done via removing the first item of the list and recursively calling the “**insertHelper**” method with the new list as a parameter. Next added to the “**insertHelper**” method was the startIndex variable. This variable was needed in order to keep track of where each character within the sentence occurred within the text, with respect to the start of the text. This variable was declared within the scope of the class such that it was accessible to all methods within the class and so that it did not get reset upon each successive call of the “**insertHelper**” method. For each SuffixTrieNode that was inserted, the sentence number and character index was first inserted into that node, then the node into the Trie. For each iteration of the “**insertHelper**” method, a check is performed to determine if the required node already existed. If not, it was inserted with the appropriate data. If it already existed, the data within the node was updated such that it could be determined if the node was part of multiple sentences and which ones it was. A problem arose in the future with this method which will be discussed later.

#### “**getNode**” Method.

The implementation of the “**getNode**” method was quite trivial as the method taken directly from the Trie class worked after implementing the “**SuffixTrieNode**” class.

#### “**getSentence**” and “**getSentenceHelper**” Methods.

In order to search the SuffixTrie for a particular sentence, referenced by number, the “**getSentence**” method was created. To begin development of this method, a variable, foundWord, was declared within the scope of the SuffixTrie class such that is could be used for other methods. This variable emptied before calling the “**getSentenceHelper**” method from within the “**getSentence**” method. The “**getSentenceHelper**” method takes 3 parameters, a map of children of the current node, a target sentence number and a SuffixTrieNode. A loop was written to iterate through all the children of the current node, via the map, and to get a list of all the SentenceData objects of said node. Each SentenceData objects sentence number was compared against the target sentence number in order to find the nodes corresponding to the target sentence. When one was found, the key of that node was added to the foundWord string and the “**getSentenceHelper**” method was recursively called with an updated node and child map. This approach resulted in partial sentences or no sentence at all being returned. This was because it would occasionally find the suffix of the sentence before the whole sentence and return that. From this it was evident that a check was needed to determine if the node was part of the original sentence or not. The first attempt at rectifying this problem was to implement a flag within the SuffixTrieData object to signify if the node was part of the original sentence. This resulted in further errors when testing with text file Frank.txt. The problem was fixed by linking the flag with each individual sentence number within the “**SentenceData**” class. The check then consisted of a comparison of the nodes sentence numbers to the target sentence number and if that particular sentence number within the node was part of the original sentence. 1 final check was needed to determine if the sentence had been found. If so, a flag is set and checked in order to prevent saving the sentence to the foundWord variable multiple times.

#### “**getSentenceAtIndex**” and “**getSentenceAtIndexHelper**” Methods.

The final function related method to be implemented within the SuffixTrie class was the “**getSentenceAtIndex**” and “**getSentenceAtIndexHelper**” methods**.** This method is responsible for returning the sentence starting at a user provided index and ending at the next full stop, question mark or exclamation mark. The “**getSentenceAtIndex**” method utilised the foundWord variable previously declared. The foundWord variable is cleared before calling the “**getSentenceAtIndexHelper**” method. This method takes 3 parameters, a map of all children of the current node, a user specified character index and a SuffixTrieNode. A loop was written to iterate through all children of the current node storing all character indexes of said node in a list of integers. Another loop was written to iterate through all character indexes of the current node and compare them to the target character index. It there was a match, the key of the node was added to the foundWord variable and the “**getSentenceAtIndexHelper**” was recursively called with an updated child map and current node. This method worked beautifully, first time.

### **SuffixTrieNode** Class

To begin, the code from the TrieNode class was copied over. I decided to keep the same choice of map (TreeMap) for storing the children of the node as it stores keys in an alphabetical order, which was nice for displaying the structure of the SuffixTrie during testing and, iterating through all children of a node in alphabetical order may prove to be useful as the more common letters used in the English language occur towards the start of the alphabet. Therefore, it may be more efficient when searching for children. As a result of this, the “**addChild**” and “**getChild**” methods did not need to be modified. The functionality regarding terminal nodes and number of children was removed as they were not needed. The “**toString**” method was overridden such that when a node is converted to a string, it displays which sentences the character of the node occurs in and the character index of the node in a pleasing comma separated format. Next the data getter and setter methods, “**addSuffixData**”, “**appendSentenceData**”, “**appendStartIndex**”, “**hasData**”, “**getSentenceDataList**” and the “**getStartIndexList**” methods were implemented.

#### “**addSuffixData**” Method.

The implementation of this method was also quite trivial, I simple rename was all that was needed.

#### “**appendSentenceData**” Method.

This method merely consisted of a call to another method within the **SuffixTrieData** class, “**addSentenceData**” passing on the sentence number and the state of a flag used to determine if this sentenceData object was part of the whole sentence of part of a suffix of the sentence. This will be discussed later.

#### “**appendStartIndex**” Method.

This method takes the character index of the current letter within the node and passes it on by calling another method “**addStartIndex**” located within the **SuffixTrieData** class.

#### “**hasData**” Method.

This method simply returns a true or false value depending on if the node has data of not.

#### “**getSentenceDataList**” Method.

This method returns a list of SentenceData objects by calling “**getSentenceDataList**” from within the **SuffixTrieData** class.

#### “**getStartIndexList**” Method.

This method simply returns a list of integers containing the character indexes of the node. (Everywhere this character occurred within the text with respect to the start of the text).

#### “**getChildren**” Method.

The final method implemented within this class returns the map of all the children of the node for external use within the **SuffixTrie** class.

### **SuffixTrieData** Class.

This object is responsible for storing all the node related data and was constructed fresh. The **SuffixTrieData** object contains 2 arrays, 1 containing the character indexes and the other containing the SentenceData objects. The constructor takes 3 parameters, **startIndex**, **sentenceNumber** and a flag **wholeSentence**, used to determine if the sentence related data (sentence number) is part of the whole sentence or just a suffix of the sentence. 5 simple methods were then conjured into existance.

#### “**addStartIndex**” Method.

This method just appends the supplied character index to the array of start indexes.

#### “**addSentenceData**” Method.

In similar fashion to the previous method, this method simply creates a new SentenceData object with the supplied sentence number and flag state then add it to the array of SentenceData objects.

#### “**getStartIndexes**” Method.

This method just returns the array of start indexes.

#### “**getSentenceDataList**” Method.

Returns the list of SentenceData objects.

#### “**getSentenceNumbers**” Method.

This method iterates through the list of SentenceData objects extracting the sentence numbers and storing them in a list of integers which is then returned. This was just for displaying all the sentences a particular character occurred in from within the “**toString**” method in the **SuffixTrieNode** Class.

### **SentenceData** Class.

This class was deemed necessary when implementing the “**getSentence**” method for the **SuffixTrie** class. When implementing said method, searching for which node had the correct sentence number would not always return the correct sentence. This is because a suffix of the sentence would occasionally be found first, which still occurred in the 1st sentence for example. Upon discovering this issue when testing, it was evident that some sort of marker was needed to signify that any particular node was part of the original sentence. Further explanation and justification will occur later within this report. The **SentenceData** object contains 2 variables, the sentence number and a flag to determine if the node containing this sentence number was part of the first whole sentence or not. A constructor was then written and 4 methods, “**getSentenceNumber**”, “**setSentenceNumber**”, “**isWholeSentence**” and “**setIsWholeSentence**” were implemented.

#### “**getSentenceNumber**” Method.

This method simply returns the sentence number of this particular SentenceData object.

#### “**setSentenceNumber**” Method.

This method is used to set the sentence number of this particular SentenceData object.

#### “**isWholeSentence**” Method.

This method returns a true of false value depending on if this SentenceData object exists in a node that is part of the whole sentence and not a suffix.

#### “**setIsWholeSentence**” Method.

This method is used to set the state of the flag.

### **GUI** Class.

The final class developed was the graphical user interface. To begin, a new JFrame was constructed. Utilizing NetBeans JFrame design capabilities made designing the frame itself relatively simple, aside from changing or working around automatically generated code. Textarea’s, Labels, Buttons etc were all placed from the palette. All the text for the labels and buttons were initialized within the constructor. The next thing written were the actions for each of the buttons. The load button within the JMenu utilizes a file chooser in order to allow the user to load a text file into the SuffixTrie with a file browsing window. The quit button implementation was quite trivial. The search sentence button takes the last line of user input from the 2nd textarea, converts in to an integer and calls the “**getSentence**” method. The resultant string is then appended to the 2nd text area in a fancy format. Similar actions were performed for the search character index and search word buttons. Small things were added for handling edge cases, to make the user experience more pleasant i.e. handling incorrect input, shortcuts, scroll bars and basic instructions on how to use the program.

### 

# TESTING

The code was constantly tested throughout development and several issues were identified/rectified along the way. The testing regime was as follows.

### Testing **readInFromFile**

This method was tested using a simple system output statement to output the sentence being passed to the “**insert**” method to ensure that the input text was being split into sentences correctly.

### Testing **insert**

This method was tested via another system output statement that output the sentence to be inserted on each iteration, the associated sentence number and character indexes. The result of this was hard to read so a simple loop was written to output the structure of the SuffixTrie after the complete insertion of each sentence. This greatly helped when troubleshooting the “**getSentence**” bug.

### Testing **getNode**

Testing this method was not necessary as it was taken directly from a program previously written that was known to work.

### Testing **getSentence** and **getSentenceHelper**

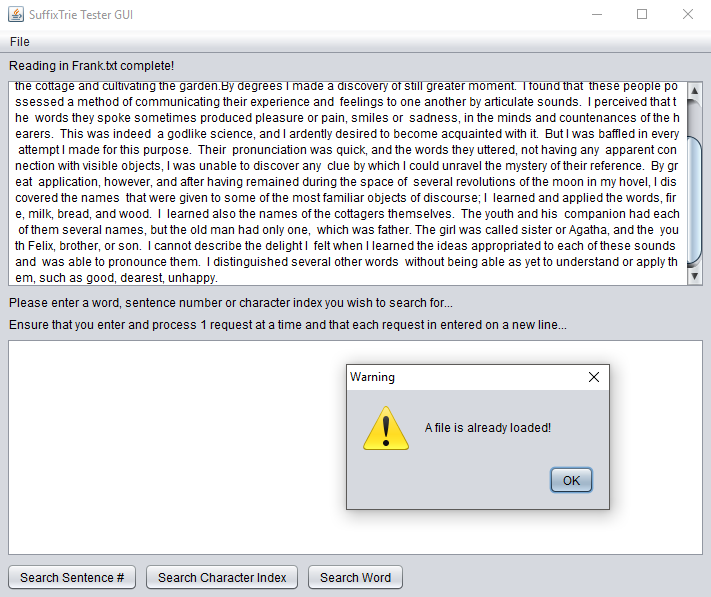
This method was a bit more difficult to test than others but testing and troubleshooting essentially occurred in the same fashion as other methods, with system output statements. Upon first writing this method, it was evident that the sentence returned was not correct. Using various system output statements, I was able to determine where and why particular errors occurred, this will be discussed later.

### Testing **getSentenceAtIndex** and **getSentenceAtIndexHelper**

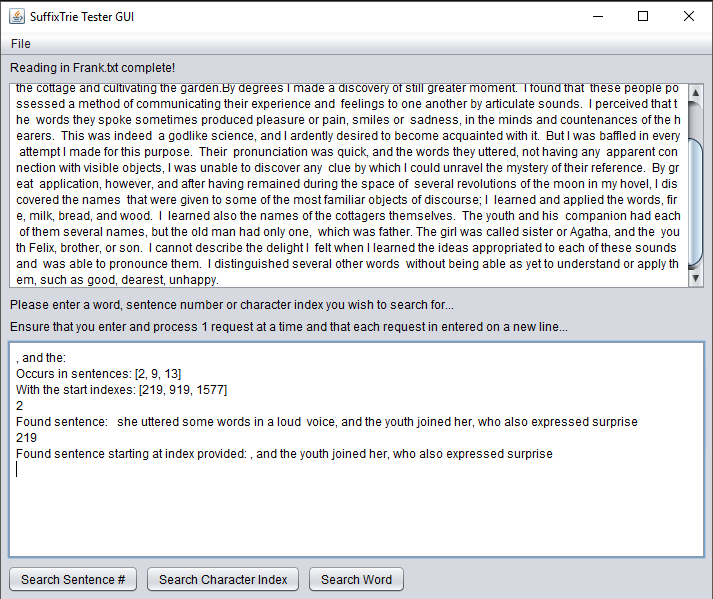
Luckily, or skillfully, this method worked correctly first time and no testing or troubleshooting was necessary. If this were not the case, id image using system output statements would be a fool proof method of determining what could be going wrong.

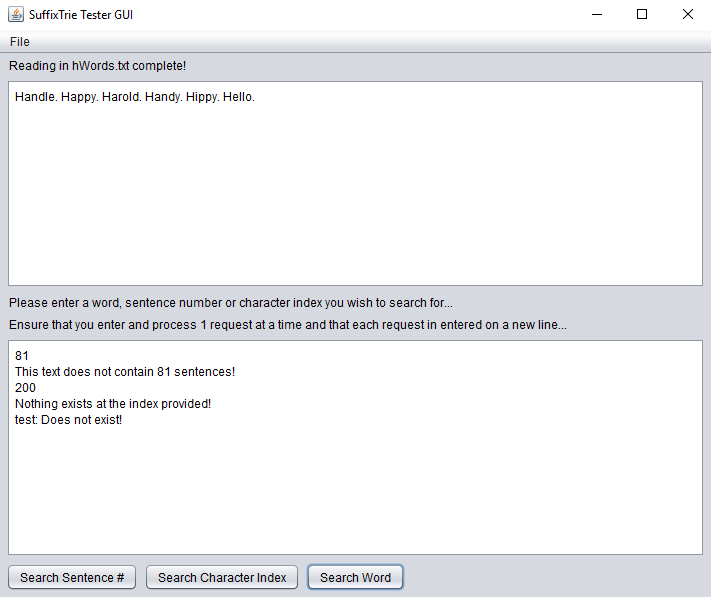
### Testing the **GUI**

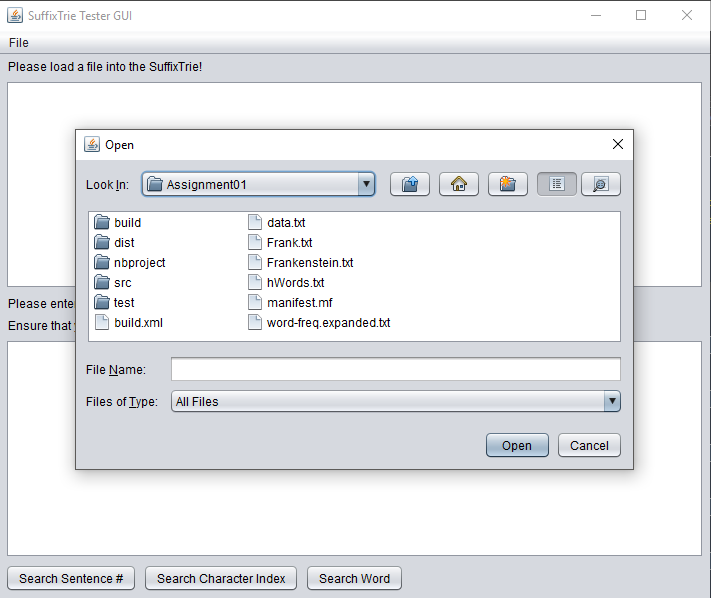
This is where the majority of the testing occurred. From the graphical user interface, I was able to extensively test all functionality related methods by providing small and large text documents, incorrect data i.e. using characters as target sentence numbers etc. Rather than implement incorrect type detection within the methods, I checked for inconsistencies at the GUI level to avoid a lot of potential run time errors. Additional measures were taken for various edge cases like trying to load multiple files into the Trie etc. Some screenshots of testing can be seen below.



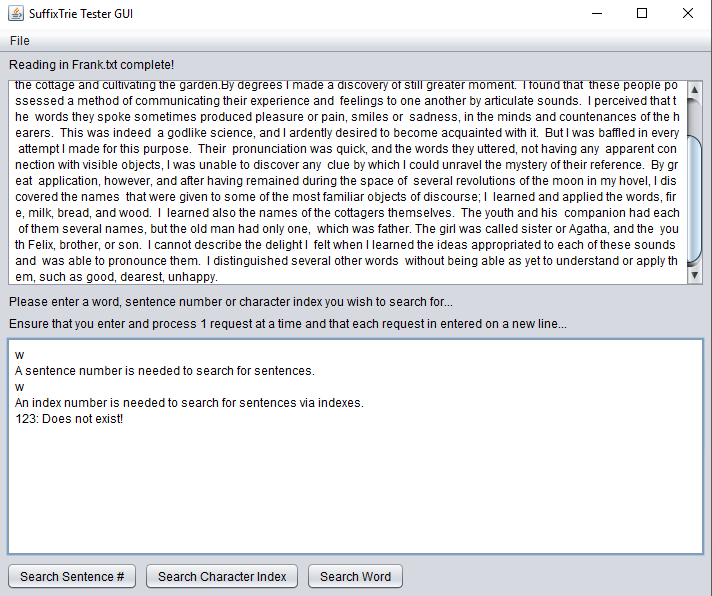
*Fig. 1. Testing result of loading multiple files.*



*Fig. 2. Testing Search Sentence #, Search Character Index and Search Word functionality. Fig. 3. Testing cases of non-existence.*



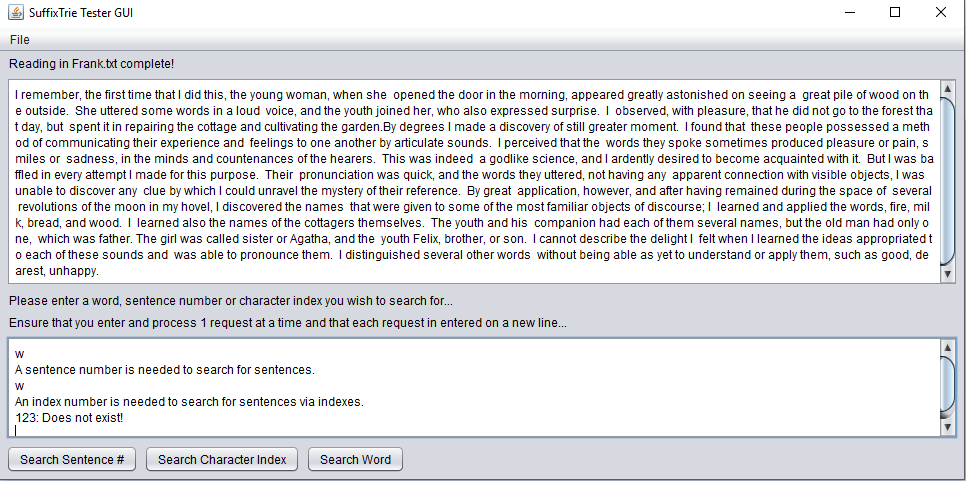
*Fig. 4. Testing default load directory.*



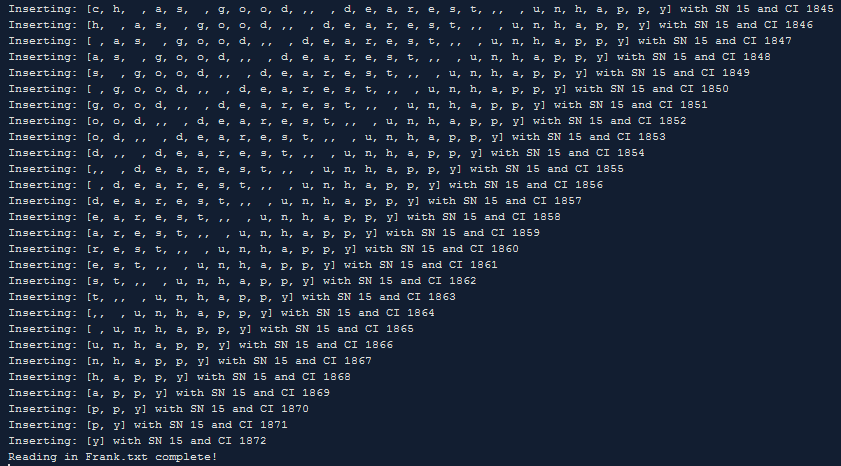
*Fig. 5. Testing incorrect data types.*



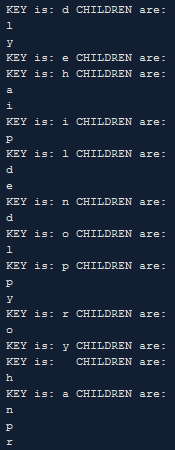
*Fig. 6. Testing ability to load large files, out of memory error.*



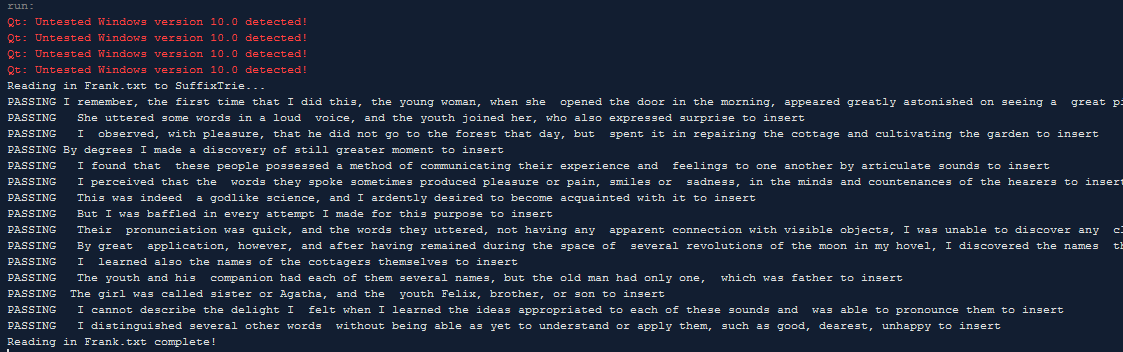
*Fig. 7. Testing dynamic resize ability.*

**

*Fig. 8. Testing insert method.*

**

*Fig. 9. Testing structure.*

**

*Fig. 10. Testing readInFromFile.*

# COMPLICATIONS & DESIGN DECISIONS

Various complications were encountered throughout the development of this program, some of which were minor although some influenced the restructure of methods thus influencing design decisions.

### “**readInFromFile**” method complications.

When developing the “**readInFromFile**” method, I discovered that the split method was not working correctly. After doing a bit of research, I discovered that a full stops and question marks meant something different as a regular expression and they needed to be escaped. I found an example of escaping multiple characters using square brackets so I implemented which solved the problem.

### “**insert**” and “**insertHelper**” method complications.

When first developing the “**insert**” and “**insertHelper**” methods, I attempted to use an array of characters rather than a list of characters. I thought this would be more efficient due to not having to convert the array to a list before calling “**insertHelper**” but troubles arose when trying to delete the first character. Passing Array.remove(0) to the recursive call of “**insert**” would remove the first character but not delete the last character when there was only 1 character left. Upon discovering that its much easier to delete an item from a list, I modified the “**insertHelper**” method to use a list and converted the array to a list within the “**insert**” method. This served to fix the issue.

The next issue with the “**insertHelper**” method was updating the character index after every recursive call. This was a simple thing to fix as I just moved the startIndex variable outside of the method to avoid resetting it.

### “**getSentence**” and “**getSentenceHelper**” method complications.

The largest issue faced was when I was developing the “**getSentence**” and “**getSentenceHelper**” methods. The first problem encountered was that after writing the methods, when trying to search for a sentence, either a suffix or an empty string would be returned. After testing it step by step, both in code and on paper, I realized that I was only checking to see if the sentence number of the current node matched the target sentence number. This resulted in the return of suffixes of the target sentence because that was the first node found and it matched the target sentence number. Even though the suffix is still the first sentence, it is not the original sentence or the WHOLE sentence. To solve this problem I modified the SufffixTrieData object to contain a Boolean value that would be true for nodes that occurred in the first sentence, the whole sentence. After making the necessary changes and testing it on a small file I made myself. Everything appeared to work, until, I tested the Frank.txt file. Upon testing the Frank.txt file, I discovered that it would return some correct sentences and some incorrect sentences. Looking at the code, nothing appeared to be wrong so decided to tackle the problem by hand. That’s when I discovered the underlying problem, that a node can be labelled as being part of a particular sentence with a false identifier signifying the node was not part of the first sentence, but that value can be changed but another sentence that begins with that particular letter. Thus, when inserting another sentence that begins with the same letter as the suffix of another sentence, the flag would be set to true saying that that suffix is the first whole sentence inserted. After finally figuring this out, contemplation on how to solve it began. My solution was to make a new object, SentenceData, that contained the sentence number but also the associated flag identifying whether it was inserted as the first sentence or a suffix. This way there can be multiple children of a node with the same sentence number but only 1 will be identified as being part of the whole sentence and it cannot be overwritten. So after implementing the new class, adding the appropriate methods, with unpleasing levels of abstraction, to other classes and heavily modifying both the “**insert**” and “**getSentence**” methods within the SuffixTrie class, it almost worked. Now I was getting the sentence stored twice within the foundWord string. I was able to fix this with yet another flag that gets set after the deepest layer of recursion rolls out such that the “**insertHelper**” method would not further modify the foundWord string. Final testing of the “**getSentenceHelper**” method yielded happy days!

### **GUI** complications.

When implementing the ability to display the text from the text file within the 1st textarea, I discovered there seemed to be no way to alter the automatically generated text from the graphical GUI construction tool within NetBeans. After a bit of research, I found a work around which allowed me the make the declaration of the 1st textarea static such that I could append text to it from within the SuffixTrie class without having to instantiate a GUI object. Problem solved.

### Design Decisions.

* Chose to use a TreeMap for mapping the children of a node due to it’s ability to search keys alphabetically, possibly increasing efficiency of searching the children of a node.
* The implementation of “**getSentence**” influenced the use of a secondary data object, SentenceData, to store sentence numbers.
* Chose to make a GUI as I had no experience making GUI’s and wanted to learn how.
* Chose to use recursion wherever possible, because it looks nicer in code and its more satisfying when it works.
* Chose not to implement SuffixTrie and SuffixTrieNode as subclasses of TrieNode and TrieData as I wanted the data structure to be a standalone data structure i.e. don’t need Trie files in project folder.

# FUNCTIONALITY

The final product is a relevantly clean, dynamic graphical user interface that serves to demonstrate the functionality of the SuffixTrie data structure. It possesses the ability to load text files into the data structure, refer to *Figure 4*, the ability to search for the occurrences of specific words and substrings within the text, refer to *Figure 2,* the ability to find sentences by number, refer to *Figure 2,* and the ability to locate the remainder of a sentence starting at a user specified index, refer to *Figure 2.*

Limitations worth noting:

* Large files cannot be loaded into the SuffixTrie due to memory limitations. This is possibly due to poor memory usage. This could be fixed by implementing compression functionality.
* Cannot search for substrings that span over full stops, question marks and exclamation marks.
* Multiple files cannot be loaded concurrently within 1 instance of the program.