Part 1 Justification:

Part 1 required a Trie to be programmed, importing a dictionary of words and placing them into a tree. By using this new Trie, the GUI could then make informed decisions on what the end of a word would most likely be.
All participants were provided a barebones template of a Trie Class, with set tests, some methods already generated and a semi-functional GUI. This GUI merely appended “bogus” to the end of every word a user was typing, but would later turn into the predictive text that was assigned.
The following subsections go into detail on the different methods created for each class and justifications for any necessary decisions made.

Trie.java

This class contained the majority of the challenges that arose throughout this part of the assessment, such as *getMostFrequentWordWithPrefix()* and *getAlphabeticalListWithPrefix()*, the details for which will be explained later. The other classes aren’t broken up into methods like this one, but due to the number of methods in this class, it was formatted to break up the information. Note that some methods aren’t mentioned, that is due to their simplicity. Some of the methods were relatively straightforward and didn’t warrant justifying any decisions.

findWords()

This method iterates through every word to identify any terminal words and returns a List of discovered words. This method is given a few variables;

- A list of discovered words
- A working prefix
- The letter that is currently being used
- And a pointer

Using these variables, the method moves the pointer to the letter, appends the letter to the prefix, and identifies every child node to the pointer, calling the *findWords()* method to the children. Once a list is returned, the function then proceeds to check to see if the current word is terminal, and if it hasn’t been added yet, appends it to the list of found words. Only then does it return the list of discovered words.
Justification:
A choice had to be made on how the discovered words would be sorted. The options were: Hashmap, Binary Search Tree or List. A Hashmap didn’t seem appropriate, as to store the options there needed to be another data structure anyway (to store the keys). In terms of a BST or List, both were viable options, as both had a default sorting time complexity of O(n log n), however Lists insertion time (to the end of the list) is O(1), whereas the BST has an insertion time of O(h) with h being the height of the BST. Another viable option would be to insert the words in a sorted state.

The largest issue with this method is the amount of input variables. If enough time was given, these input variables could be reduced, removing the current letter and the working prefix, however this seemed to be the best course of action to reduce time complexity.

getAlphabeticalListWithPrefix()

Consider this method as the initializer for *findWords().* This method creates the list for the discovered words and maneuvers the pointer to the end of the prefix. From there, the list of children (if any) are created and the pointer iterates its way through each of the children, calling the *findWords()* method for each child of the anchor. Once all of the words have been found, the list is sorted alphabetically and then returned.

Justification:

The only reason there had to be two methods for this task (*findWords() and getAlphabeticalListWithPrefix())* is that there are some tasks in the initializer that don't need to be part of the recursive method. If the recursive method had to start from the root each time, the time complexity of the method would be astronomically high. Not only that, it would increase the programming difficulty as well. By referring to a second "recursive" method, the layout is a lot neater than what it would've been.

That doesn't mean that it is perfect. As can be noticed, this method does contain a lot of code that is incredibly similar to *findWords()* particularly everything inside the "if pointer is null" statement. This could be mitigated if the reformatting of both the "initializer" and "recursive" methods were modified correctly, but due to time constraints, there wasn't enough time to reconfigure it.

findMostFrequent()

Similar to *findWords(), findMostFrequent()* is a recursive method for *getMostFrequentWordWithPrefix().* The method sets the pointer as the temporary "most frequent word", comparing these words against all of the children's terminal words. Once compared to the children (if there are any), the method returns the most frequent word from below itself, in *frequencyData* form.

Justification:

The *frequent* variable had the option to be a TrieNode or TrieData. The reason TrieData was chosen was due to its size, or lack thereof. The TrieNode has quite a few variables to track that are unnecessary for this method. For example, there is no need to keep track of any children, as the *frequent* variable would have no children. The same can be said for whether it's terminal and its number of children. Instead, TrieData objects only store two variables, frequency and word, which is exactly what needs to be stored for this method.

getMostFrequentWordWithPrefix()

The initializer for findMostFrequent(). The method checks whether the prefix is valid (exists), then parses the pointer to the recursive *findMostFrequent()* method.

Justification:

There is a chance a user could enter a wrong prefix, or in the case for the GUI, there would be a new word that hasn't been entered into the dictionary. To mitigate this, it is worth covering for these instances, returning null, and dealing with it in any other methods.

readInDictionary()

readInDictionary() is a pretty simple method. It scans a file, creates a new dictionaryTrie, separates each word by a whitespace, and parses each word through the *insert()* method mentioned above.

Justification:

When splitting the words, originally, only a " " was checked for. The problem is that this didn't account for any other whitespace characters, or any time there were multiple whitespace characters in a row. Instead, the regex "/s+" matched at least one whitespace, grouping multiple whitespace characters together, removing them. This avoided the loop attempting to enter empty strings or whitespace strings to the dictionary.

TrieNode.java

There wasn't much change to this class, however there needed to be a storage of children, or nodes lower than the node selected. The choice between Data Structures varied, but HashMaps seemed to fit the situation perfectly. They have an access and insert time of O(1), and the fact that HashMaps can't be sorted is irrelevant, as the children don't need to be in any order. Having a tree of children overcomplicates the solution, and almost duplicates the issue to begin with. A linked list could be a viable option, but having an access time complexity of O(n) means it is less efficient than the HashMap. The only other method that was added was the *getChildren()* method which provides a list of the keys for the HashMap. This means that Trie.java can iterate through all of the nodes children.

TrieData.java

The biggest modification to this class was the storage of a word. The constructor changed so that it could take a string which would be the word itself that the frequency correlates to. This means that some of the Trie.java methods would be able to get the string itself from any terminal nodes, instead of having to work its way back up the trie. It was an easy trade from time to space. Instead of spending more time to get the word, spending more space to store it seemed better.

Part 2 Justification:

Part 2 involved creating a Suffix Trie that could be cast upon a txt file with any format needed. For example, for every word in a text file, every suffix possible (including the whole word) would be available in Trie form. At first glance it would seem quite similar to Part 1, and for some areas it would be. The following sections discuss justification for any choices made throughout the assignment, plus a breakdown for any thought processes for the methods. SuffixTrieDriver isn't included, as there was nothing changed and was kept the same.

SuffixTrie.java:

insert()

This *insert()* method differs from Part 1's as this method needed to add **every** possible suffix to the Trie. Using nested for loops, the method can start with the entire length of the word, then trim off the first letter, adding that suffix, removing another letter and repeating until there is nothing but a white space is left. The loop then retrieves the next word, resetting the pointer to the root Node. This process repeats until all of the words in the sentence are done.

Justification:

The strings were brought to lowercase to avoid double-ups of words. If the words were case sensitive, there could be multiple instances of the same word, resulting in an unnecessary increase in Suffix Trie size.

readInFromFile()

This method is quite similar to *readInDictionary()*, as both takes in a file name, and parses it through a scanner. The scanner splits the file up into strings / sentences, a sentence being defined as either a punctuation mark (one of . ? or !) followed by a white space, or a newLine character. The resulting string is stripped of its trailing whitespace characters, and if by that point there are still characters in the processed string, is parsed through the *insert() function.*

Justification:

The delimiter is a little confusing, as it seems very specific. That is due to the fact that it was based around passing the Testing B Coderunner page. The specifics are explained in the testing phase, but ultimately the resulted strings always have at least one non-whitespace character in them and are separated according to Test B's specifications.

The reason that Test B was chosen over Test A was that it was closest to the code that was written before testing had begun, and

Testing:

Two major CodeRunner tests were provided, both had different delimiter requirements. The following explains what requirements would need to be met to pass these tests and any issues that occurred during the testing phase.

Option A:

The specification for these test cases are as follows:

Empty lines are treated as sentences, i.e. they contribute to the sentence count (increase the sentence index), even if they contain no text.
If a sentence starts with or ends with spaces, they are not ignored, and these spaces increase the character index that will be returned by searching.

These requirements are able to be achieved by splitting the entire file with nextLine(), splitting it even further using the split() method, using the delimiter “[.?!]”. This means a sentence doesn’t extend past a newLine (\n) nor one of the punctuation marks mentioned in the delimiter. When viewing the codeRunner results below however, a few errors still appear. The sentence indexes are accurate for each test, however some lines have incorrect character indexes. This is because the text uses some characters that cause issues. For example — looks like a hyphen, however it is actually an em dash. CodeRunner expects the program to register em dashes as three index iterations, however this code doesn’t take them into consideration, and merely adds them as is into the SuffixTrie. The same is done with the “æ” in daemon.txt (inserted as one character instead of three). This could be used by checking for any em or non-regular characters that might appear in these texts and adding extra nodes to compensate, however it was decided that this was a bug in the CodeRunner rather than a feature that needed to be implemented.

Test Case 1
Input:
Frank02.txt and , the
Output: (Colors have same meaning as codeRunner)
[and]: [1.27, 1.87, 3.34, 3.153, 4.15, 6.70, 7.66, 7.148, 8.88, 9.96100, 9.1204, 9.1737, 9.199203, 10.17, 12.67, 12.91, 13.97, 13.114, 14.27, 14.90] [the]: [0.58, 1.116, 3.51, 3.96, 5.39, 7.36, 7.48, 7.127, 8.1, 8.18, 8.76, 9.1, 9.859, 9.2328, 9.24953, 10.57, 10.74, 12.7, 12.22, 12.56, 12.158, 13.42, 13.65, 13.144, 14.1, 14.31, 14.146, 15.19, 15.69, 15.133, 15.184, 15.288, 15.302] [, the]: [8.16]

Test Case 2 isn’t included due to its size and to improve readability, however can be seen through CodeRunner.

Option B:

The test case specifications are as follows:

Empty lines are ignored, i.e. they do not contribute to the sentence count (increase the sentence index).

If a sentence starts with or ends with spaces, these spaces are stripped off first, and they do not contribute to the character index.

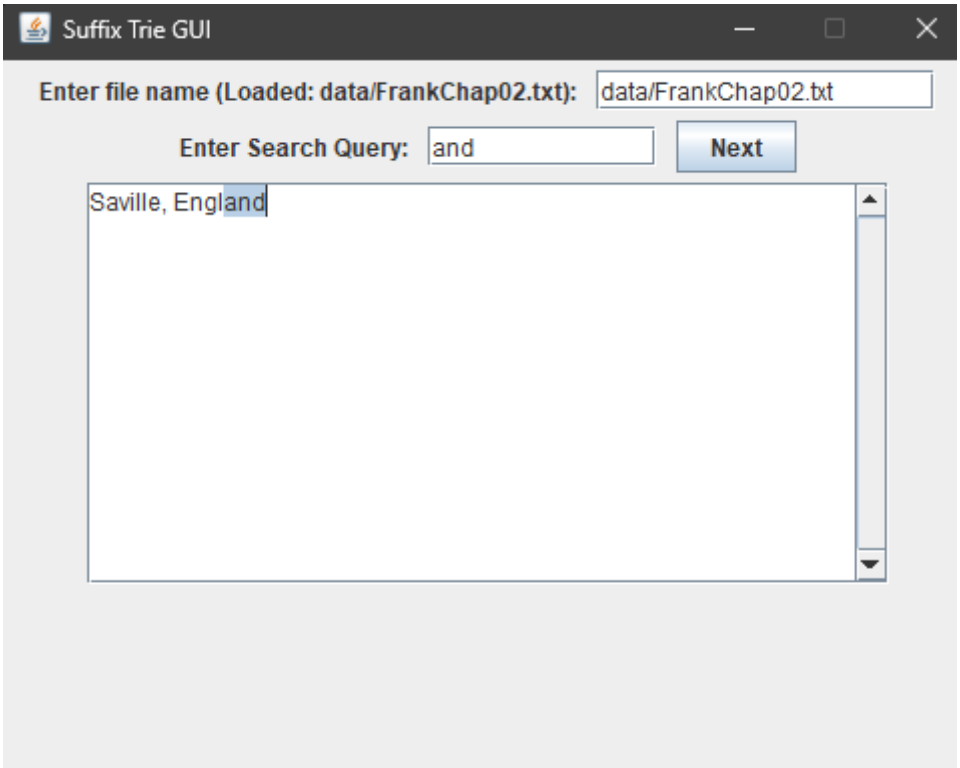
The previous example used NextLine(), which splits by line rather than a sentence. Instead a delimiter was set that allowed the Next() function to either use the delimiter “[.?!]\\s” or act as nextLine(). This is due to the fact that this test case considered the end of a line as a sentence (rightfully so). This resulted in almost perfect test cases (as can be seen in codeRunner), however Test Case 2 had one error. When querying the suffix “and”, all entries with sentence index 131+ were offset by one. If the error was on the program’s behalf, it occurs between line 131 and 138 (inclusive).However, after hours of troubleshooting no errors had been found, so this “error” was marked down as a coderunner error.

Note:

Unfortunately, due to time constraints, an in-depth analysis into the run times and space complexity of the program couldn’t be completed, however it is worth mentioning that Frankenstein.txt file is too large for the program to process, however if the Suffix Trie was translated to a Suffix Tree instead, there may be a chance for it to fit. The program’s responsiveness is adequately fast, with a small wait time when loading a new file, but more investigation should be conducted on whether or not the time complexity could be reduced further (perhaps taking another look at the *readInFromFile()* method).

Extension:

There were a few opportunities available to improve the classes. Ultimately, a GUI implementation of a “find” feature seemed like the best case. The plan was to implement a GUI that can iterate through each of the lines of an input file, presenting the sentence that contains the suffix, and highlighting the position of that phrase in the sentence. This task could be broken down further into two major sections, the GUI itself and the context-referencing. Both tasks had their own challenges, and the code isn’t exactly the most optimal, but for a first crack at creating a GUI and implementing a “relatively” difficult task to it, It turned out nominally well.



Conclusion:

Both the Dictionary Trie and Suffix Trie were lengthy challenges, requiring many hours of troubleshooting, programming, and screaming at the monitor. Due to time constraints, some errors still appear, but given enough time, can be ironed out. Similarly, the testing phase did provide great insight on the direction what the formatting of the “sentences” should be, despite being cut short, and should definitely be revisited to improve the stability of the software. The GUI, while very simple, provided a great opportunity to learn different aspects about generating GUIs and using JFrames in this type of setting.