Data Structure:

The main data structure to store the ‘strings’ or words from the Shakespeare.txt file was a Java Trie (or a variation of a trie/search tree). Storing (potentially) millions or 100k+ strings in trie would, in comparison to an array, map, table, or a set, would save memory if the keys are long and/or share many common prefixes. It would also be easier to access/search words based on the prefixes using a trie as it would go down (DFS) an existing character followed by the starting character.

In my assignment, I’ve implemented a Trie like data structure that holds all the entries or strings from the Shakespeare.txt. In inserting a word or creating a new TrieNode, I kept track of the word occurrences – which in this case, would be the line and column number of the words in order they are inserted in the trie. The trie itself would not be used to find duplicate words for methods such as wordcount(), therefore by keeping track of the occurrences of that string, we can count exactly the line number and the column number of the word as it appears on the actual text document. Every new node that is created or a child of the node that is set should take in a character, line number, column number, as well as a Boolean to keep track of the occurrences of duplicate and non-duplicate entries. There is also a custom implemented HashMap that is like the java.util framework. It is used to represent the children of TrieNode with a character as the key and more TrieNodes as the value in order to store subsequent characters of a string.

Algorithms:

1. Counting the number of occurrences of a word
   - Aside from saving memory, storing a large text file in a Trie while maintaining track of its occurrences allowed quick lookup of word counts based on a single string. The justification for using a Trie is that searching for data in a trie would be quicker assuming the worst case O(n) time where n is the length of the string searched, compared to another option – a hashtable. Aside from hashtables potentially mapping different keys to same positions in the table, it was difficult to implement a brand new hash map from scratch as we were limited to creating our own data structures and not importing other frameworks/libraries.
2. Search for a Single Word
   - As mentioned above, tries will have no key collisions or instances where the hash function maps different keys to the same position in a HashTable. Assuming no errors were made in storing the data into a trie, searching for a single word in a trie would be faster in the worst case than a hashtable or other structures such as a hashmap, set, etc.
3. Search for Prefix of a word
   - In addition to comments made above, all the descendants or children of a node have a common prefix of the word that is being associated with the node and the empty root node. Instead of having to iterate/traverse different keys to find a word associated with a prefix, you would traverse down given a matching character of the word.
4. Search for phrase (contiguous)
   - A better method for phrases would have been an implementation of a radix tree or a search tree that constructs associative arrays with string keys. My current implementation of a trie relied solely on word occurrences (line and column number) which were directly compared with the occurrences of the following word. The algorithms would require O(n^3) or lower efficiency methods/for loops to obtain, compare, and put line numbers and column numbers for each word in the parameter.
5. Boolean Search Logic

- For same reasons as #4, a better method for Boolean search logics would have been a radix tree or a similar that uses arrays or strings keys. Current implementations are also heavily dependent on occurrence comparisons using for loops.