Data Structure:

- For this assignment, I utilized a Trie, which is an efficient retrieval data structure, to insert and search words/strings, as well as to update their counts and occurrences. Information was stored into "TrieNode" structures, which consisted of the following fields:
    - **occurrence** – An integer that stores the number of times a given/specific word in the data file appears in the dictionary file.
    - **superwords** – An integer that keeps count of how many times a given/specific dictionary file word is a prefix of a data file word.
    - **pref** – An integer that keeps count of how many times a given/specific data file word is a prefix of a dictionary file word.
        - used **only** in 'second.c'
        - proper prefix count is updated recursively when printing output and freeing nodes (in printAndFree method)
    - **isLeaf** – A boolean that keeps track of whether or not a TrieNode is a leaf node (a node with no children).
    - **letter** – A character that stores the TrieNode's alphabetical letter.
    - **string** – A pointer to a character that stores the full word, or string, that was added to the Trie. For non-leaf nodes, the default value is set to NULL, while for leaf nodes, the field value is set to the added word/string.
    - **child** – A pointer to a TrieNode that consists of an array of 26 child TrieNodes. When a TrieNode is created, all of its children are originally set to NULL until more TrieNodes are created to continue the creation of the word (the 26 children correspond to the 26 alphabetical letters in which 0=a, 1=b, 2=c, etc.).
        - The children are implemented in a linked list style, since each child (representing a letter) is connected to its children (representing other letters), which eventually form complete words/strings.

    *** Note**: The leaf nodes contain the major information, such as the full word/string, the number of occurrences, etc., while the other nodes contain mostly default field values.

Algorithm/Methods:

- **Main**: Takes the input (mapping file) and obtains the names of the dictionary and data files by using string tokenizer. Line by line, these dictionary (first b/c we have to build the Trie) and data files (second b/c we have to search the Trie) are sent into the readDict method so that they can be read/scanned for possible strings/words.
- **readDict**: Scans the given dictionary or data file character by character and creates a word or string by storing these letters into a character array as long as they are alphabetical letters. Once the scanning reaches a non-alphabetical letter, a string terminator is added to the end of the current word/string and is sent into the matchStr method to be either added to or searched in the Trie (depending on if the file is a dictionary or data file).
- **matchStr**: Scans each letter of the given word/string (calls updateField at end of method).
    - If the given string is from the **dictionary file**, then a node is created for each letter and is stored in the node's children array in an alphabetical manner (0=a, 1=b, 2=c, etc. -- a character field called "letter" also stores its actual alphabetical letter).

    - If the given string is from a **data file**, then each letter of the word is searched in the Trie. If a letter from the word is not found, then it does not exist in the Trie, so the match fails and the counts/occurrences remain at the default value (zero).
- **assignNodeValues**: Assigns the default values to each field of a newly created TrieNode.
- **markNull**: Sets all of the newly created TrieNode's 26 children to null to aid in a search or traversal.
- **updateField**: Does one of the following:
    - If **dictionary file**, then copies the given string to an allocated character array, sets pointer's string field to the copied string, and marks the current TrieNode as a leaf node.
    - If **data file**, then the searched string was found in the Trie, so increment the pointer's/leaf node's occurrence field.
- **printResult**: Creates and opens an output file in write mode and passes that file into the printAndFree method in order to print the output in the desired format in the newly created file.
- **printAndFree**: Recursively traverses through the Trie until reaching a leaf node, where it prints out all of the leaf node's information, such as the current word, its occurrence, and its superwords count (if 'first.c') or its proper prefix count (if 'second.c'). After printing the output for that specific leaf node, it frees the allocated string that was stored inside the leaf node and as the recursion continues to print the output, the nodes are also being freed as well.

Big-O Analysis:

- m = maximum number of words in either the dictionary or data files
- k = maximum length of a word
- n = number of unique words in the dictionary file

Since the first and second part utilized the same algorithm/code, the Big-O analysis will be the same for the two.

- Running Time: O(m*k)
    - Reading Dictionary File: O(n*k) b/c we have to obtain every word, which can be max length k, from the dictionary file
    - Reading Data File: O(m*k) b/c we have to obtain every word, which can be max length k, from the data file.
    - Insertion: O(n*k) b/c we have to create a Trie that consists of nodes for all letters of every unique word, which can be max length k, from the dictionary file.
    - Search: O(m*k) b/c we have to search the Trie for all the words, which can be max length k, from the data file.
    - Setting Node Field Values: O(1) b/c we are only assigning values to node fields and setting a fixed 26-character array to null.

- Recursively Printing Output and Freeing Trie: O(n*k) b/c we have to recursively traverse through every letter (node) of every unique word, which can be max length k, in the Trie in order to reach the leaf node and print out the desired output, as well as to free the allocated strings and nodes.

- Space Complexity: O(n*k) b/c the Trie is created only once in O(n*k) time (searching for one word is convenient because it only takes O(k) time, since the children indices are structured like a linked list and can be accessed based on their alphabetical positioning, but overall, searching for all data file words in the Trie takes O(m*k) time).

Challenges Faced:

- When first starting this assignment, I found that obtaining the dictionary and data file names was somewhat confusing, but after realizing that the process was very similar to the matrix problems in assignment 1, I was able to figure out that I could use the string tokenizer to determine the file names, so that I could move on to scanning the files for words. In addition, the binary search trees (BST's) from assignment 1 allowed me to feel more comfortable with recursion and traversing through trees, especially since I was able to code a method that recursively traverses through the Trie to print the desired output and at the same time, free the allocated strings and nodes as the recursion continues. Creating the Trie was the most difficult part of this assignment because we had to create nodes for each letter of every word to be inserted, which led to constant node creations in other nodes' children arrays. We also had to utilize an efficient way to store and print out specific information, such as the line numbers and the unique words/strings, as well as their prefix counts and occurrences. Maintaining efficiency and making sure to free/deallocate all the allocated space created was an important part of this assignment (opposed to the first assignment) because some of the dictionary and data files consisted of several possible strings to a few hundred thousand possible strings, which could affect our code's performance if not handled properly.