Data Structures Used:

- During this project, I used the data structure known as a Trie. The Trie consists of Trie Nodes. Each Trie node has five fields (word, occurrence, super-word/prefix, letter, and children). As the program progresses, it creates a Trie tree for every dictionary file it encounters.

Methods Used:

Main:

- Used to read in dictionary and data file names. Calls the following methods in order:

1) readDict

- Reads a file and creates a Trie for it. The method completes this by reading each word of the dictionary and only inserting if the word is not already in the tree.
  Big O Analysis:

  Since you read all the words, you will read m words from dictionary. However you traverse down the length of the tree(log(k)) for each word so your big o is O(mlog(k))

2) leafNodeDater / idkdd

- leafNodeDater takes the data file. From this data file it creates strings by adding and clearing out of an array. Once the method creates a string, it sends the method idkdd the string name. Here is where the magic happens. The idkdd method searches for the string, and updates the prefix/super word and occurrence count.
  Big O Anaylsis:
  Worst Case. Each word from data file goes to the bottom of the tree. Then you will have m traversals at the length of the tree log(k).
  O(mlog(k))

3) printResult/printHelper

- prints out all the unique words. printResult accomplishes this by using preorder traversal.
  Big O Anaylsis:
  Visit every element of the tree. O(nk)

4) De-Allocate

- Frees the space allocated from malloc for Trie. To do this, the method traverses to the bottom nodes and free them by using recursion. More specifically, it uses post-order traversal. The Big O of this function is:

  Assume Worst Case: all unique words (n) are max length (k).
  O(nk) because it traverses through all letters of each word.

Thus the big O is O(nk + mlog(k))

The memory requirements are O(26*K*N)