Data Structures Used:

- A trie is used to store dictionary and counters.
- The structure of a set of nodes represent 26 characters is as follows:

```
typedef struct LetterNodes {
        unsigned char letterChildIndex          [ALPHABET_COUNT];
        unsigned char childCount;
        unsigned char letterLeafIndex           [ALPHABET_COUNT];
        unsigned char leafCount;
        unsigned int    dataWordCount           [ALPHABET_COUNT];
        int*      pCounters;
} LetterNodes;
```

  - letterChildIndex has 26 bytes and it indicates if a letter is not present or present but with no child or present with child. It is also the index of the pointer to the child nodes.
  - childCount is how many children these 26 letters have.
  - letterLeafIndex has 26 bytes and it indicates if a letter is a leaf or not. If it is a leaf, it also indicates the location of its counters in an integer array created when processing the data file.
  - leafCount is how many leaves are there among these 26 letters.
  - dataWordCount is the number of words in the data file at this node. The count will be propagated down the trie as the prefix of the dictionary words.
  - pCounters point to an array of integers that serve as counters for all the leaves of these 26 letters.

Big O Analysis:

1. Loading the dictionary

    a. Space requirements:

        i. Only unique words are stored in the trie.

        ii. Let the tree depth be k (max word length) and the number of unique words is n.

        iii. Memory requirement is k*n for the whole trie

        iv. O(n*k)

    b. Computing requirements:

        i. Unit time to process each word is k (tree depth)

        ii. Assuming m words to process, big O is O(m*k)

        iii. O(m*k)

2. Processing the data file
    a. Space requirements:
        i. Assuming n unique words and max word length is k
        ii. Memory required to store counters is O(n*k). Duplicate words do not increase memory requirements.
        iii. O(n*k)
    b. Computing requirements:
        i. Unit time to process each word is k (tree depth)
        ii. Assuming m words to process, big O is O(m*k)
        iii. O(m*k)
3. Total Big O:
    a. Memory:
        i. O(n*k)
    b. Computing:
        i. O(m*k)

Challenges:

1. The challenge is trying to find the best compromise between CPU time and space requirements.
2. Because each set of nodes has 26 letters, if 26 units of memory are allocated, it would be a big waste if only 1 is used.
3. If a linked list is used to store only the ones needed, it would require more CPU processing time to search the list.
4. I ended up using a byte array of 26 letters to represent a set of nodes and allocated pointers only when needed. In this way, there is no need to perform searching as everything is indexed. No memory is wasted creating a full set of nodes.