Assignment 1 - Wordstat

Kaitlin Poskaitis

Algorithm

This program works by using 2 main data structures: a prefix tree and a binary tree. The input file is read character by character, and these characters are inserted into the prefix tree case-insensitively if they are part of a word. For example, if the word "foo" was read, starting from the root node (a generic space), first a pointer to 'f' is created. Then from this 'f' node, a pointer to 'o' is created, and so on. This is done by storing an array of TrieNodes of length 36 (26 letters followed by 10 digits) within each TrieNode. By default, these are all null, but a new TrieNode is created when there should be a character there. Going back to the "foo" example, the root node would have the pointer to a new TrieNode containing 'f' at index 5 of the array of its children. This 'f' TrieNode would then have a pointer to a new 'o' TrieNode in the appropriate location in its children array, and so on. Once the end of a word is reached, a pointer to a WordNode is defined for that TrieNode, which contains relevant data about that word. Checking whether this pointer is NULL determines whether the current word is a new word. I chose this data structure to store the case-insensitive words because of its good efficiency for inserting and printing the words (described in the next section).

The prefix tree I implemented is only used to store case-insensitive words, so there needed to be another way to keep track of the variations. I handled this by containing a pointer to a binary tree within each word node. This binary tree starts with a root NumNode with a value of -1, as this node is not relevant. From this node, if a variation has been found with the respective character in upper-case, the NumNode would branch to the right with a value of 1 in the new NumNode. Otherwise (lower case or number), it will branch to the right with a value of 0. This way, it is easy to check if a certain variation has already been seen by simply traversing the tree and seeing if the path already exists. For example, "Foo1" would produce -1-¿1-¿0-¿0-¿0, while "FOo1" would produce -1-¿1-¿1-¿0-¿0. I chose this data structure because it is a fast way to store variations and check if variations have already been seen.

In order to print out the data obtained by this program, a preorder traversal of the prefix tree is done to ensure alphabetical order. The data required is all stored within the WordNodes, so it is easy to access it once the location of the WordNode is determined.

Big O Analysis

Time complexity: O(n)

The time complexity is on the order of n, where n is the length of the file, because of the prefix tree. For each word, it will take the same number of comparisons