Assignment 3 -- Indexer

Ian Lozinski, Edward Zaneski CS 214

The goal of the assignment was to implement an "indexer" which would take a filename or a directory as its argument, read the file(s), and index every word the it comes across. Before settling on one, we analyzed the runtimes of a few different data structures. The naive approach would be to keep a sorted linked list, and keeping a second linked list for each file associated with every word. Obviously, the worst case runtime of this would be O(n^2) and thus, very inefficient. We quickly discarded that idea and considered using a hashtable. It seemed like a good idea at first because of it's O(n) insertion time, but in order to get everything in order, we'd need to throw every tuple, (word, filename(s), count), into an array and then sort it using either mergesort or quicksort, yielding O(nlogn). We still weren't satisfied with this. Finally, we decided to implement a prefix-tree. Every prefix-tree node has a 36-index array of pointers to other nodes. Index 0 through 9 represent digits, and index 10-35 hold letters. Each word of length w added yields an insertion time of O(w), which is practically constant.

This is what happens when the word "cat" is read in:

- Starting at the root of the prefix-tree, we see 'c'.
- Since 'c' is the third character in the alphabet, we check index 12.
  - If index 12 has a pointer stored, we follow it to the next node
  - Otherwise, we create a new node, store the pointer to it, then follow it.
- The next character 'a' is read in and we do the same thing, but starting at the node we just traversed to.
- Same thing for 't'.
- Now that we are out of character, we know that we have complete word stored in the tree.
- The filename is also stored in a prefix tree in the exact same way, except we additionally have an associated count at leaf nodes.

In order to print out every word in lexicographical order, all that is required is a pre-order traversal. No sorting necessary. This is what makes the prefix-tree really shine. The algorithm we used, recursively goes to each node, from left to right. Using a buffer string, we add characters to the string when an edge is traversed downward. When we see that a filename is associated with a node, we know that it is a complete word and we can print it. Similarly, when we traverse back up the tree, characters are removed from the buffer string.

Complexity analysis:

Inserting n words to the tree takes O(n) time. Traversing through the tree is also O(n).