Name: Douglas Rudolph
Course: Systems Programming
Professor: Dr. Francisco
Assignment: Assign. 0

Description:

The design methodology behind my *pointersorter.c* application was to minimize the amount of helper functions needed to get the correct output. as well as implement a solution using the least amount of lines of code. Knowing this, my program can be split up into five major steps: filtering out non-alphabetic characters, store all words in nodes, insert all nodes into its proper lexicographic location within a list, and print out the list, destroy the list. Generally speaking, these steps define the overall algorithm for my program. Building on top of this, the functions I implemented within my program are a named directly after each step:: *main(filter strings), create_node, print_list,* and *insert*, *destroy.*

Implementation & Features

1. **Main**
    a. Implementation: Inside my main method is the algorithm for filtering out non-alphabetic characters. The way I went about solving this problem was by trying to figure out a way to force the algorithm to allow for the *strtok* function to be used. In my case, the way I went about doing that was by first looping through the strings given at the command line and converting all non-alphabetic characters to a space character. After this, I then would call *strtok*, used the space character as a delimiter, and would then grab all the proper words that we had to filter into.
    b. Features: One of the most important features that was added into *main* was to ensure that exactly one input is read from the command line. I also made sure that the other cases for when no string is entered and when too many strings were entered were complete.
2. **insert**
    a. Implementation: After filtering out all the characters, I would call the *insert* function and pass the strings I wanted to add into a Linked List. It's important to note that when passing the string from *main* into *insert,* the string was not in the form of a Node struct type. Instead, I would create an instance of Node inside of my *insert* function, and then insert the instance of Node once the Node instance had memory allocated.
    b. Why I chose a Linked List: The reason I chose a Linked List as my data structure was due to the fact that it is very easy to insert a string into its properly sorted location - also, a Linked List was the fastest implementation I could think of as it

offered a clean solution that used a very minimal amount of lines. When thinking about using a Linked List in contrast to other data structures, the only other data structure I truly thought of implementing was a Hash Table that mapped each string to a bin based on the first letter of the string. The problem I had with with this kind of Hash Table was when mapping data to a bin, there is a chance that every string entered would be mapped to the same bin. Thus, because of this, the worst case scenario is still *O(n^2)*. While in general, a Hash Table is faster, the Linked List offered the cleanest solution.

3. **create_node**
    a. Implementation: The *create_node* function was simply used to take in a string, and return a struct Node pointer that stored that string that was passed in. *Create_node* was called inside of my *insert* function.
4. **print_list**
    a. Implementation: The *print_list* function was quite simple to understand. It's purpose to was to loop through the entire list, and print out the string that was stored at each index in the sorted Linked List.
5. **destroy**
    a. Implementation: The *destroy* functions purpose is to loop through all values in the Linked List, free the string in each node, as well as free the node itself. I did this by creating a loop that would store the reference to *ptr->next* in a temporary variable, free the data in node as well as the node itself, and then set the *ptr* variable equal to the temporary variable storing the reference to *ptr->next*. This process would then be completed until all strings were destroyed.