Micole Finehart
netID: *mcf155*

Sridhar Sriram
netID: *sss293*

Description:

NOTE: During the description of the assignment, the term *"delimiter"* is defined as any character that is NOT Alphabetic - i.e. 1234,!?\], etc.

The files "readme.pf", "pointersorter.c", and "testcases.txt" for this assignment are intended to satisfy the requirements of Assignment 0: String Sorting for CS214: Systems Programming. You are currently in the "readme.pdf" file. The "pointersorter.c" file contains the working code for this assignment and the "testcases.txt" document provides a file containing all of the cases that were used to ensure that the code that was developed was indeed "robust code".

Essentially, Assignment 0 asked the students to read a user inputted string and separate the string at every delimiter. Then, Assignment 0 asked for the program to return the different "chunks" - parts of the string separated by the delimiters - in alphabetical order, noting that uppercase letters took precedence over lowercase letters. To illustrate, if the user were to input "Hello Zebra/thing;blarg'stuff" the program would return:

Hello
Zebra
blarg
stuff
thing

Overview:

Our program was separated into 4 functions with the function signatures as follows:

```
int main(int argc, char** argv)

Node* createNode(int start, int end, char * user_inputted_String)

void printList(Node * start)

void sort(Node * passed_in_node){
```

GLOBAL DECLARATIONS:

Struct _Node contains two variables:

1. A char* that points to the word stored in the Node
2. A _Node* that points to the node immediately following the current Node

Node * "head_of_list"

1. Initialized to null
2. This pointer is supposed to point to the head of the "master" list
3. Once head_of_list refers to a Node, the rest of the "master list" can be accessed

MAIN:

Within the main function, there were several components that connected to the rest of our program. Initially, the main method checks for an error in input: if there are less than or more than 2 arguments passed in, then an error message is printed, and the program subsequently ends. After this check is made, the main function runs in the following format:

- A char pointer named "user_inputted_String" is created and initialized to the user's input
- Two int variables - iterator and start - are created and initialized to 0, as they will be used to keep track of indices within the string
- A Node pointer "new_node"is first allocated for memory, and then used with the intention of holding the "new" word after each pair of delimiters
- A while loop is entered, that proceeds until the end of the string (indicated by '\0') is found
  - Each character in the string (at index iterator) is checked to see if the character is alphabetic:
    - If the character IS alphabetic, then iterator is incremented, and the loop continues
    - If the character is NOT alphabetic, then new_node calls the createNode function, passing in start (the start of the given word), iterator (the place at which a delimiter is found), and the original String ("user_inputted_String")
      - A check is then made to see if new_node is NOT NULL
        - If so, then we sort directly
        - Otherwise, iterator is incremented
      - A check is then done after these two checks to see if the index of the updated iterator is now at the end of the String (indicated by '\0')
      - If so, then the same steps as if the character was a delimiter are taken, followed by a break outside of the loop

- After the while loop looping through the user_inputted_String is terminated, then a call to the printList function is made
- Finally, the “new_node” is freed, as well as all of the nodes stored within the Linked List
    - Essentially, a temp Node is created (and allocated for) in every iteration of a while loop
    - The temp is then set to point to the head_of_the_list
    - The head_of_list points to the next node in the LL
    - And then both the temp->word and the temp node are freed
- The function returns a 0, and the program is subsequently terminated

CREATENODE:

The createNode function is simple: the function creates a node using the the indices passed in as indicators of what portion in the original string needs to be copied into the newly created node.

- Space is allocated for a Node * “created_node”
- Space is also allocated for created_node-> word
- Then, using memcpy, the word is copied into the created_node->word
- A check is done to see if the created word is null - if so, then NULL is returned
- created_node->next is set to NULL, as at this time we do not know the location of where the newly created node should be placed into

PRINTLIST:

The printList function accepts the “head_of_list” pointer, named “start” in this function, and uses this as the starting point. Afterwards, a while loop is entered, printing the word at each node until start is NULL.

SORT:

The sort function is, as expected, where the majority of the work was done. The sort function only accepts a single parameter: the newly created node. We dubbed it a “sort as you go” function because every time a new node is created, we pass it into the list to be sorted. From there, the following steps are taken:

- Space is allocated for prev and curr pointers, that are set to NULL and head_of_list respectively, as they will help keep track of the current place in the linked list
- A check is made to see if head_of_list == NULL, if so, then head_of_list points to passed_in_node and the function returns
- A while loop is entered that runs as long as curr is NOT NULL
    - The first couple of steps are taken to figure out which word - the word of the passed in node or the word of the current node is shorter - as this will give us a limit during our call to strncmp in order to avoid accessing out-of-bounds information
    - In our call to strncmp we have three possibilities:

- If strncmp returns a number greater than 0, then that means that the passed_in_node comes AFTER the current node, so there is no reason to insert passed_in_node at that location in the LL
    - A follow-up check is made to see if curr->next is null
        - If so, then this means that the end of the linked list has been reached and passed_in_node needs to be inserted at that location
    - At the end of this conditional, the function returns
- If strncmp returns 0, then that indicates that the words compared are equal UP UNTIL the indicated length
    - If this step is reached, then the shorter word comes BEFORE the greater length word
        - If prev is NOT equal to NULL, then this means that we are not at the head of the list, and prev-> next = passed_in_node
        - If prev is EQUAL to NULL, then this means that we are at the head of the list, and head_of_list is now pointing to passed_in_node
    - At the end of this conditional, the function returns
- If strncmp returns a number less than 0, then this means that curr holds a word coming AFTER the word in passed_in_node, meaning that we should insert passed_in_node BEFORE curr
    - A check is done to see if prev == NULL
        - If this is the case, then we are at the head of the of the list
            - Then, passed_in_node ->next points to curr and head_of_list points to passed_in_node
        - Otherwise, passed_in_node -> next points to curr and prev-> next points passed_in_node
    - At the end of this conditional, the function returns

Other Information:

The work for this project was evenly distributed. It is hard to distinguish who worked on which aspect of the assignment, as the two of us had our hands on all parts of the project.

CHALLENGES:

Arguably the most significant challenge with this project was, coincidentally, the goal of the projects in CS214: working together. As neither of us have taken on a coding assignment as part of a team, this was a very interesting experience, in that we both had to find ways to both understand and work with the other's coding styles. Nevertheless, we were able to quickly overcome these differences and put together a rather robust assignment for Asst0.