Daniel Chen
CS214
Assignment 0

Introduction

This assignment was to create a CSV sorter that would sort a CSV file of movie details with a given category title such as director_name, movie_title, etc. The sorter.c file contains the main, getCat(), and the buildString() functions. The mergesort.c file contains the functions merge() and merge_sort(). The header file sorter.h contains the function definitions as well as the definition for the struct we used Record.

How to Use

The user reads out a CSV file through use of cat and piping it into the compiled function from sorter.c and mergesort.c. The user specifies the first argument to sort by category, -c, before specifying the second argument of the category name. It would look something like below:

cat [filename.csv] | CSVSorter [-c] [category]

Ex: *cat movie_metadata.csv | CSVSorter -c director_name*

If the program finds the category within the available categories, it will sort for that category and print out a sorted CSV file.

Design

The overarching design splits the work up into three sections. The first section is the input through stdin in the main(). This splits the stdin input up line by line to be parsed by “index”. This index is determined by the category titles given in the first line. Once the target substring has been found, the index is saved for parsing the subsequent lines from the CSV file. The next part is parsing each line for the target data. This is done through the helper functions getCat() and stringBuilder(), which manually search for the delimiters to tokenize by. These parsed substrings are then saved into a struct for modularity, and the struct is saved into an array that is passed to the final part, mergesort. Mergesort splits up the array in place based on the mergesort function. The main() then prints out the sorted CSV file line by line.

Assumptions and Difficulties

Some assumptions made are that the first row will always have the category titles, that the user will specify a valid category title, and the length of the row will not exceed 1024 characters. This also assumes that the category titles do not have quotations, and that all rows have the same number of categories (but do not have to be filled). It was also assumed that the data could be sorted through strcmp through merge sort. However, this led to an issue because strcmp as it turns out does sorts numbers as individual chars, resulting in wrongly sorted data. I tried to fix this by going back and changing the way data was handled by using void pointers instead of

char pointers. However, this led to a breakdown in code due to the ambiguity of what data type the tokenized substring will be. I believe a correct method would be to use unions to allow for data to be parsed and stored, and later sorted through a set of specific flags to signify whether the substring is an int, float, or char. One more difficulty I encountered was that the last row movie_facebook_likes seems to disappear when it's sorted. The data exists when it's passed to mergesort, but not after. Other people in the Piazza group also encountered this issue it appears.

Extra Credit: Generalizing Any CSV

I set up the code to generalize inputs by not hard coding any category titles. Instead, I parse the initial input through index numbers based on the user-provided argument of the category title. This too can be generalized even more so that the user could potentially just specify the column they would like to sort. The struct array data is stored in is also dynamically sized so that any size of the CSV can be input. Some additional parameters to consider would be a maximum size of the data in order to know if we'd max out our memory heap.

Files and Functions

sorter.c

sorter.c is the main body of the code as it has the main function which handles stdin input from the CSV file we're passing into it.

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

Arguments: CSV through stdin, column (-c) flag, category title
Returns: A sorted CSV file

When we pass the CSV through stdin, we're able to differentiate the arguments from the user through argv, where the specified category to be sorted is passed in as argv[2]. We get the first line of input through the function fgets(), which stops at a newline. We parse this string by index using strsep() using a comma delimiter to find out how many indices there are within our first line. This allows for modularity for any number of categories given in the data. If the category is not found however, it exits after printing an error message. The rest of the CSV lines are attained line by line through fgets() in a while loop that dynamically stores data into an array of structs called recordList, of type Record. Data is parsed through the getCat() function and the buildString() function to get the substring to sort by. The struct tmpList stores the sorted substring as a char pointer field_data, and the original string to original_row. The array recordList stores structs tmpList's and then passes the array to mergesort.c to sort by field_data. The returned sorted array is printed out as a sorted CSV.

*char *getCat(char *line, int catIndex)*

Input: Char pointer to line from CSV, index number of category data to sort
Returns: The searched for substring based on category index

Function getCat takes in the char pointer for a line of data from the CSV as well as the category index we got from comparing the first line of category titles and user input from the main function. The way the function operates is by using two char pointers to keep track of the beginning and end of a string based on the given delimiters. The end pointer iterates until it finds the ending delimiter. When a delimiter is found, the start and end pointers are passed to the stringBuilder function to be tokenized. The tokenized substring is returned back and is returned back to the call.

*char *stringBuilder(char *start, char *end)*

Input: Start and end char pointers pointing to beginning and end of the substring
Returns: The tokenized substring

This helper function calculates the size to allocate for the resulting substring through malloc and uses memcpy() to copy the string. It then returns the resulting substring.

mergesort.c

This file has two functions merge() and merge_sort(). It follows the well known method of sorting mergesort which recursively splits the row of data until it reaches its elements before sorting them correctly in the return with a runtime of O(nlogn).

*void *merge_sort(Record **a, int low, int high)*

Input: Array to be sorted, low index, high index

Output: None (void)

This function takes in the array passed in from main() and recursively divides by index value. Once the array has been reduced to its individual elements, it passes it to merge() to be recombined in sorted order.

*void *merge(Record **a, int low, int mid, int high)*

Input: Array to be sorted, low index, mid index, high index

Output: None (void)

This function takes in each array element and recombines based on its value using strcmp. The final result is a sorted array that's done in place.

sorter.h

The header file holds the functions as well as the struct definitions. It defines the struct Record to hold two char pointers, field_data and original_row. The sorter program stores the data as an array of Record structs, where the field_data holds the category data to be sorted, and the original_row holds the original string from the input CSV file. This allows for fast and modular sorting in an efficient method.