David Tian (dt474) & Rahul Boppana (rsb172)

[Extra Credit] **Our program is built to be a general CSV sorter**
[Extra Credit] **Interesting Observation:**
About 57% of the countries listed in the database entered the film industry after 2000. And about 28% have entered after 2010. This is possibly a result of the further development of many countries in recent years and considerable economic growth.
-The excel spreadsheet is attached to the submission with the statistics. The data was calculated with built-in counting formulas set with the conditions “>2000” & “>2010”.

Design:

Our first design issue was to decide how to properly store the data from the CSV file. We initially decided to create a specific struct for the provided data columns from IMDB. In addition, since we wouldn't know how many rows were present in the CSV and would only get a single chance to parse through the data, we decided to store the data in a linked list and copy it over to an array at a later time. We assumed that the CSV files would contain only numerical or alphanumerical or null characters to sort for at least this assignment.

After some consideration with this design we realized that we would encounter trouble with dynamically accessing the desired value from the struct, as well as being unable to complete the extra credit. To address this we changed our design to instead store each row as an array of unions, each of which contains either an integer, a double, or a pointer to a string.

This design also seemed complicated to implement during the mergesort stage so we revised it and finalized with an array of structs called "Listings" containing only 2 fields: the first being a string of the a given row in the CSV, and the second also a string but of the Column-Of-Interest in that row. (The column value that we would be sorting on.)

Reading in these values required the use of a few helper functions to properly initialize the strings and checking commas and quotes when traverse the CSV. Since we would need to store strings of unknown length, we decided to implement a function to read characters from a file pointer and reallocate the string to a larger char array whenever necessary.

To find the Column-Of-Interest, we counted the indexes until we reached a the column in the header row(first row of the CSV) that matched the input parameter; we used string tokenizer to traverse the header row.

Since we did not know the number of rows in the CSV, we stored them in a LinkedList and then transferred them over to an array. This LinkedList is freed properly when we transfer it to the array.

And to establish the type of data in the Column-Of-Interest, we set an int flag called "columnType" that would default to 0 and when an alphabet character is detected in any of the values of that column in the Listings, "columnType" is set to 1. 0 instructs mergesort to compare values numerically and 1 instructs mergesort to compare lexicographically using strcmp().

Once everything was read in properly and the Listing array called "data" was populated correctly, we also initizalized "indexArray" which would simply store the array indexes of the "data" array 0 to n-1, n being the number of rows. For simplicity's sake, and to preserve the original data for testing purposes, the mergesort algorithm only sorts the values in the index array and at the end, the "data" array is printed out to a file in the order presented in "indexArray".

The mergesort algorithm uses a special comparator function that can handle both numerical and string values. The inputs are both strings and if the columnType = 0 and signified to sort numerically, it converts the strings to numbers and returns a value based on the comparison. For strings, it uses strcmp() as mentioned. This algorithm returns -1 if the first input is less than the second, 1 if greater, and 0 if equal. This function also accurately checks for null values.

Implementation Basic Overview:

The program proceeds as follows:

1. Read in header column and store values in a string array
2. Search for target column and store its index in the array
3. For each row:
    a. Read until the (column index) comma and store the value in the row struct
    b. Store the entire line in the row struct as well
    c. Add the struct to the linked list
4. Copy the linked list to an array of row structs
5. Set the type of the column to be sorted in a global int
6. Create an array of integers to represent the indexes of the struct array
7. Mergesort the integer array by comparing the column data in the corresponding indexes
8. Print the row string to a file in the order of the indexes

Headerfile:

The simpleCSVsorter.h file contains the Listing struct, the Node struct, and the front node of the LinkedList. The data array of Listing pointers, the indexArray, and the global columnType integer are also present. The rest of the declared functions are written out in the mergesort.c file. InsertNode inserts a node to the front of the list. findHeader checks the presence of a header row and selects that value for processing and populateListing creates and sets the fields of a listing struct that gets added to the LinkedList. readLine reads a line of the CSV file to be processed later and numChars is a smaller helper method to count occurrences of a specified character in a string. removeWhitespace simply does as it says to the value stored in the Column-Of-Interest field of each struct to prepare for sorting while preserving the original data. The Mergesort implementation consists of three functions: mergeSort, merge, and comparator.

The mergeSort and merge functions mutate the indexArray directly and comparator is as described previously. The remaining print methods are for printing different things such as the LinkedList or a Listing. These are mostly for testing purposes however, the printData method prints the final result to a file as requested by the assignment. The methods freeLL, freeArray, and freeListing simply free the respective structures in their titles.

Testing Process:

We created a small test.csv with a few entries to initially test the program. At first there were plenty of small typos that prevented the code form compiling and then a few pointer type errors which simply required a bit of re-evaluation. After making those fixes, the code compiled fine and worked with our small test.csv. With the large movie database csv, we would get an address sanitizer heap overflow error. We found that this happened because our mergesort and other string manipulation functions were accessing arrays outside of their index bounds. To tackle this, we restructured mergesort to sort directly on the original indexArray and keep track of the necessary indexes, instead of creating new arrays. The program ran well after making this edit and we were able to ensure it handled some error cases well. Situations like a missing header row will direct the program to return to STDERR and print a message explaining the issue. We also created a cars.csv and a modified food.csv to test.

Instructions for Running:

Please follow the same instructions outline in the Ass0 assignment description. It specifies to use the “cat input.file | ./simpleCSVsorter -c movie_title” format to run the file once compiled.