Erick Galvez 189005643
Eric Zhang 172005801
Professor Francisco

csvSorter Documentation

A big difficulty we faced in our experience working on this project was our inexperience with C. Our initial general approach was to somehow input the CSV file, read each line, trim out non-alpha-numeric characters, then store the lines it into a 2-D array, with X being the column/header name, and Y being row lines. From there, we could easily loop through the array and sort/output the sorted order. However, we realized that we could not implement 2-D arrays as easily as we could in Java. From there, we re-approached our method, and decided on the following general process: Input the CSV File, read each line, trim out non-alpha-numeric characters, make a struct storing relevant information for each row of data, then link the structs together into a linked list, use merge sort on the linked list, then finally traverse through the now-sorted linked list and output the row data into a .csv file using stdout. To achieve this, we planned out the way our 3 submitted files, csvSorter.c, csvSorter.h, and mergesort.c. In csvSorter.h, we initialized the struct variable rowData to be linked together in a linked list, and made several helper methods to help us organize our main in a more efficient way. These helper methods were:

Stringtokenizer:

This code was used as an alternative to the strtok() method stored in the #include <string> library. This is because when we were looking through the data in the movie_metadata.csv file, we spotted several movie names that included commas in them, such as The Chronicles of Narnia: The Lion, the Witch and the WardrobeÂ. If we were to call the unedited strtok() method, the tokenizer would create additional incorrect tokens from the commas in the title of the movie. To avoid this, we created our own stringtokenizer method that would identify the quotation marks around these movies with commas in the raw/non-excel version of the csv file (for example using: nano movie_metadata.csv would output the csv in text form, showing us these quotation marks i.e.”The Chronicles of Narnia: The Lion, the Witch and the WardrobeÂ”). Then, we looked for new line characters and white spaces to eliminate from the line of data.

columnNum:

If we image the csv file movie_metadata.csv as a 2-D array, then in the first row which stores the column header, each different column header would be assigned a number, and each row would also be assigned a number. columnNum as a method was used to traverse an inputted line, and then search for an inputted column header name using strcmp and running our stringtokenizer method mentioned above and comparing the tokens. Using a loop, we implemented a counter every time the loop repeated while comparing the inputted header name and the next token. If strcmp ==1, or there was a match, we would return the counter to obtain which column we would be sorting on.

getField:

Going back to the 2-D array from the movie_metadata.csv file described above in the columnNum method, once we had the column number, we then looked for all of the data in each row pertaining to that specific column number. For example, if our column number is 2, which in

the movie_metadata.csv is director_name, the names of the director in a specific row would be what would be returned from getField.

Now, using these, we would be able to fill out our struct nodes. In each rowData struct/node, we store the column name, value (charValue/intValue, explained below) from getField, the entire read line from the csv file, and a pointer to the next struct. A while loop was then implemented to read each line of the data and terminate when the line was empty indicating that there were no more lines left to read. For each line of data, a struct was filled out and linked accordingly into a linked list. In this while loop, we hard coded cases in which the column name would return numeric values and when the column name would return char values, as we knew from the instructions that the first row of column names wouldn't be changed to different names, and store it into charValue, or intValue, accordingly. At this point, we would have a linked list holding data for all rows of the csv file. Now, all that was left to do was implement merge sort.

mergesort:

Basically following the merge sort algorithm for arrays, we first needed to find a way to split the linked lists into a first half and second half. We did this by implementing two pointers- one would traverse the linked list one at a time, and the other would traverse the linked list two at a time. Then, when the pointer traversing two linked lists at a time reached the end, the one that only traversed one at a time would be at the middle point of the linked list. Then, both halves were stored into different arrays. This was all done and stored into a method FrontBackSplitChar or FrontBackSplit Int (depending on the data type obtained from getField) and called recursively throughout the mergesort.c file. Then, we followed the algorithm and compared the first values of each array and sorted accordingly. Ultimately, there were two different mergesorts, one that compared chars and one that compared ints (depending on the data type obtained from getField).

To output, after calling either mergesorts (char/int), we just made a new pointer pointing to the first node of the linked list, and using a while (new pointer !=NULL) traversed through the linked list, printf'ing each row of data, which would then, using stdout, stored into a new csv file.