Aaron Kau and Kelvin Liu

README.pdf

This program, which parses a CSV file, takes a column as input, and sorts the file by the inputted column, makes use of various functions and data structures to make the sorting process quick and efficient.

Usage

To run the code, first compile by running the *all* script on the supplied makefile.

1. *make*

Then, run the program by executing the following command.

2. cat [input CSV file] | ./sorter > [output CSV file] –c [column to sort by]

where [input CSV file] is the location of the input CSV file, [output CSV file] is the desired location of the output CSV file, and [column to sort by] is the name of the column to sort the CSV file by.

As an example, the *run* script (Called by running *make run*) in the make file will sort the movie_metadata.csv file (assuming it is in the same directory as sorter) by the column "director_name" and store the resultant CSV file in results.csv in the same directory.
This is equivalent to typing: *cat movie_metadata.csv | ./sorter > results.csv –c director_name*

Assumptions

This program assumes that the header file (Sorter.h) will contain the correct info about the CSV file's columns. The stringValues, intValues, and doubleValues arrays should contain the names of the columns that contain strings, integers, and doubles respectively (and stringValuesSize, intValuesSize, and doubleValuesSize should contain the length of these arrays).

The program will also initially allocate 5000 bytes of space for each string and space for 10,000 rows in the CSV file (although it will automatically increase these global variables if they are exceeded).

The program will also assume the following about any null entries:

If the column stores strings, it is the empty string "".
If the column stores integers or doubles, it is the value zero (0 or 0.0).

Lastly, the program will always assume the CSV file is correctly formatted and that the inputted column to sort by is a valid column in the CSV file.

Design

To parse the CSV file, we created several structs that contain information about the file including the data stored, data types for each of the columns, column names, as well as the number of entries being stored. An enum called *type* was used for each of the possible data types that could be stored in the CSV (string, number, and decimal). The types for each of the columns in the CSV file were set beforehand using the F.A.Q. from the CSV file, although a CSV file with other column names and data types could be parsed by modifying the appropriate constants in the Sorter.h file (stringValues, intValues, doubleValues, and columns).

Each row in the CSV file was stored as an array of values. Since values could have differing data types depending on the column it resides in, value is saved as a union, which could be either an integer value, a double value, or a pointer to a string. The corresponding data type is retrieved from csv->columnTypes[columnNumber] to determine the type of that value.

Mergesort

A mergesort algorithm was used to sort the CSV file by the specified column. The array of entries was split in half and the mergesort algorithm was recursively called on these subarrays. After these subarrays were sorted, they were merged together to form the fully sorted array using the algorithm to merge 2 sorted lists (Traverse through both lists with 2 pointers and keep adding the smallest value). Mergesort has a run-time of O(n log(n)) with n inputs, where n is the number of rows for the CSV file.

Testing Procedures

Our program was tested by sorting the movie_metadata.csv file by various different columns. We made sure to sort by each data type and we manually checked the output to ensure that in general, the resultant rows appeared to be in the correct order.

We also ran the program on CSV files already sorted to make sure that the CSV parser could take its own output as input and sort by another column.

To check for CSV files of varying sizes, we also cut out varying number of rows to from the list to make sure that the CSV file could parse smaller CSV files, and we also appended the CSV file to itself (doubling the size of the CSV file) to see if the program could parse larger files.

Lastly, we changed several names of columns (and reflected such changes in our header file), and even changed the data types of many integer and double types to string and made sure the program was outputting correct CSV files based on these new column names and data types.

Challenges

The largest challenge for this program was working in groups for the first time. Having to familiarize ourselves with using Git, Github, and the overall notion of version control was a bit challenging at first, especially when we both were making changes to the code simultaneously and had to merge our changes together.

Another challenge was in parsing the CSV files in general. It was difficult to determine exactly how much space to allocate for the CSV file, since the size is not known beforehand until the CSV is fully scanned and parsed. I addressed this by estimating an upper limit for the number of rows in the CSV file (Currently 10,000), and adding a check to automatically allocate more space if this value is exceeded (using realloc).

Extra Credit 1:

For the first extra credit assignment, we decided to discover some interesting statistics about movie durations from the movies in the CSV file. After sorting the CSV file, we were able to find the longest movie. The longest movie in the CSV file is “Trapped”, with a total duration of 511 minutes! (That’s almost 9 hours long!)

We also calculated the average duration of a movie based on the values given in the CSV file (ignoring movies with null values for duration). We found that on average, movies are 107 minutes long (out of 5028 movies with non-null durations).

The code used to calculate these durations can be found in SorterEC1.c (and SorterEC1.h is the header file used).

This code can be compiled and run using the following commands:

```
make ec1
make runec1
```

These commands compile the code and execute the program, respectively (the run script assumes the movie_metadata.csv is in the same directory as the program and the makefile).

Extra Credit 2:

For the second extra credit assignment, we created a way to parse any type of CSV file. Additional information is needed about the data types of each of the columns however, which we require to be stored in the second row of the CSV file.

In the second CSV row, we require that each column contain either the value “string”, “integer”, or “decimal” (without quotes), to specify the type of data that will be stored in that column. The file movie_metadata_ec2.csv is provided as an example.

With these data types provided, the new Sorter will have enough information to sort any CSV file by any column name. The code for this can be found at SorterEC2.c (and SorterEC2.h is the header file used).

This code can be compiled and run using the following commands:

```
make ec2
make runec2
```

These commands compile the code and execute the program, respectively (the run script currently uses movie_metadata_ec2.csv as input, sorts by director_name, and outputs to results_ec2.csv.

To sort any CSV file by any column, you can use the following command after compiling (same as for the original sorter):

```
cat [input CSV file] | ./sorterec2 > [output CSV file] –c [column to sort by]
```