Tanvi Mankame & Dhruv Duggal
Systems Programming
Section 04

Files in our project :

- SimpleCSVsorter reads standard input from the terminal, reads the file line by line using getline() and stores it into a linked list. Each node in the linked list has two attributes, one being the entire line of data from each row in the file and the other being the specific column value that we are looking to sort. After the sorter file inserts each line of data (and each individual column value) into each node in the linked list, we call on the mergesort file which is linked through a “#include” hashtag.

- Mergesort has three helper methods for the main mergesort method. It has a Split method that finds the middle node in the linked list every time it is called. The actual sorting method has three subcategories: sorting integers, sorting strings and sorting doubles. It takes two halves of a list, compares it to each other and recursively calls the method again and again until the entire list is sorted either alphabetically or numerically. The last helper method is a simple print method that allows us to see the output after mergesort, split, and sorting methods are called.

- SimpleCSVsorter.h holds all the method we use in our project, summarized by a short blurb of what each function is supposed to do.

- Makefile compiles both mergesort and simpleCSVsorter at the same time and allows a more efficient way to test our code. In order to compile, just type ‘make’ into the terminal.

Design elements:

- As mentioned above, we decided to have two unique attributes for each of our nodes. One being the typical char*data which would hold the entire row and the other being the specific column value as defined by the user.

- We used strsep to search for the column value to store in each indiviual node because strsep accounts for NULL values while strtok does not.

- We used strcmp to compare strings in the SortingStrings method since the function returns numerical values based on whether the strings are the same, less than or more than the other in terms of alphabets.

- We used atoi and atof to convert the string values we got from the file in each SortingInts and SortingDoubles so that we could compare the numerical value of the two items.

- We also used a lot of recursive calls in our mergesort file to help decrease the number of lines of code and also make the method more efficient to read and run.

Assumptions:

- We tried to use fgets before using getline() thinking that fgets mallocs automatically. We used getline() instead because it mallocs the buffer itself.

Difficulties:

- We spent a lot of time trying to figure out how to sort the data and how to look for the specific string in the entire line in every single node. Our main problem with the sorting was how to implement a method that would sort no matter what data type and how to specifically look for the column that the user inputted. We decided to write a small bundle of code that found the column data for each row for us in the simpleCSVsorter and stored it inside a node pointer. As for the sorting, we brainstormed a basic algorithm for sorting numerical values and decided to modify it for the two other helper methods each specifically targeting the three possible data types we could have in this project: integers, doubles and strings.

- Another thing that we had a difficulty with was how to pick out the specific string value (column value) from the entire line that we copied from the file. We solved this problem by writing a couple lines of code in simpleCSVsorter.c that first counted what column number the user inputted string was located at (lets assume column n) and then traversing the line we read from the csv file n number of times using strsep() and then storing that value inside node→val_sort;

Testing:

- We tested our code using a smaller set of data that worked the same way as the original CSV file. We did this because it made it easier to keep track of the data that we were storing and sorting without the possiblity of overlooking silly mistakes generally associated with large amounts of data(human error).

How to run the program:

1. Compile the program using the Makefile already provided in the folder. Type “make” into the terminal window.
   The basic compilation command an example test.c file is: gcc -Wall -Werror -fsanitize=address test -o test.c
2. The next command simultaneously opens the csv file and runs the program given a user input of whatver column they want to sort by and then outputs the sorted list into another csv file named sortedmovies.csv.
   The command you type is:
   cat movie_metadata.csv | ./simpleCSVsorter -c (column_name) > sortedmovies.csv
   Example: cat movie_metadata.csv | ./simpleCSVsorter -c duration > sortedmovies.csv

EXTRA CREDIT:
Part 1:

movie_title	budget
Avatar	2.37E+08
Pirates of the Caribbean: At World's End	3E+08
Spectre	2.45E+08
The Dark Knight Rises	2.5E+08
Star Wars: Episode VII - The Force Awakens	
John Carter	2.64E+08
Spider-Man 3	2.58E+08
Tangled	2.6E+08
Avengers: Age of Ultron	2.5E+08
Harry Potter and the Half-Blood Prince	2.5E+08
Batman v Superman: Dawn of Justice	2.5E+08
Superman Returns	2.09E+08
Quantum of Solace	2E+08
Pirates of the Caribbean: Dead Man's Chest	2.25E+08
The Lone Ranger	2.15E+08
Man of Steel	2.25E+08
The Chronicles of Narnia: Prince Caspian	2.25E+08
The Avengers	2.2E+08
Pirates of the Caribbean: On Stranger Tides	2.5E+08
Men in Black 3	2.25E+08
The Hobbit: The Battle of the Five Armies	2.5E+08
The Amazing Spider-Man	2.3E+08
Robin Hood	2E+08
The Hobbit: The Desolation of Smaug	2.25E+08

We noticed that most of the movies in the list that allocated a huge budget for their production costs also required the most amount of CGI. This shows a directly correlation between the budget and CGI that most movie directors/producers use in their media. This shows that most movie producers are willing to spend more on computer graphics to increase the aesthetic if they have a bigger budget to work off of. This helps them rake in more revenue to help balance off the excessive use of computerized special effects.

Part 2:

```
char *input = argv[2];
read = getline(&line, &len, stdin);
char* cols = line;
char *new = strdup(cols);
int column_num = 0;
char *tok = new;
while(strcmp(tok, input)!=0)
  {

    tok = strsep(&new,",");
    column_num++;
  }
```

This following snippet of our code (copied from our simpleCSVsorter.c line 25-44) creates a small program that is able to scan the user input in argv[2], read the first line which is just the column headings, and then traverse through the columns saved under variable “read” to find the column number in which the column value is located.

For example, if we were looking for where the column heading “imdb_score” was located in the csv file, we would first scan the column headings and store it into read. Then we would traverse the column headings using strsep() using the commas as the delimeter. It scans whether each token in the line equals the column heading that we are trying to find. If it matches with what the user inputted, we get the column number by finding out how many times the counter was incremented by.

This can be generalized and used for any CSV file with any number of rows and columns as long as the user inputs a valid column name, it will be able to search it up into the column headings and return a column number so that we know where it is located in each node’s data.