CSE 30 Spring 2022 Programming Assignment #3 - (Vers 1.5)
Due Saturday May 7, 2022 @ 11:59PM

Remember: 'Start Early, Start Often'

Assignment: In-Memory Database Using a Hash Table

MAKE SURE YOU READ the section titled: IMPORTANT READ THIS!!!! Before editing any files!!!!!

In PA2, part of the test fixture used data extracted from the NYC parking ticket database for the year 2019. In PA3, we will be loading parts of the same data into an in-memory database implemented using a hash table. **For this PA, all the tickets in the database are treated as unpaid parking tickets.**

In this PA you will write the following hash functions:

1. Insert a ticket into the hash table
2. Delete a ticket from the hash table (pay the ticket)
3. Find all the tickets for a specific vehicle (identified by the license plate number and state)
4. Free all all the memory in the hashtable when the program exits

You will reuse your `split_input function()`, (**not `wr_row()`**) from PA2.

The following functions are provided to you in source form:

1. main() in the file parking.c
2. Command line option parsing (getopt) in the file misc.c
3. dropmsg() from PA2 in the file misc.c
4. Routines used to read up the data files (these will call your split_input function) in the file readsubs.c
5. Hash function in the file hashsubs.c
6. String conversion functions for (i) converting a summons id (each ticket has a unique summons id number) string into a number and (ii) converting a date string into a number (hashsubs.c)
7. A simple interactive command front end to interact with the database (a simple test fixture as real databases have a much more complex interface) (commands.c)

The following functions are provided to you in a custom library (no source)

1. Debug Function to dump and print all the tickets in the entire database
2. Debug Function to print all the tickets on a single hash chain
3. Debug Function to perform some checks that the hash table was properly built
4. Function to print the vehicle with the most tickets and the largest total fine

Overview of the Four (4) Functions you will write

The csv file read into the database has the format below. There are no errors in the data file.

```
Summons Number,Plate ID,Registration State,Issue Date,Violation Code
1105232165,GLS6001,NY,07/03/2018,14
1121274900,HXM7361,NY,06/28/2018,46
1130964875,GTR7949,NY,06/08/2018,24
1130964887,HH1842,NC,06/07/2018,24
```

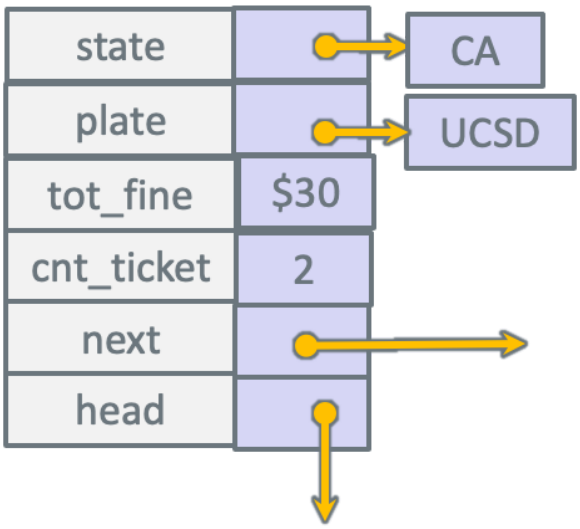
The names of the fields are: Summons Number, Plate ID (the license plate number), Registration State (the state that issued the plate), Issue Date (the date of the ticket), Violation Code (the type of the ticket, like parking next to a fire hydrant, and are encoded with a number from 1 to 99).

The hash table hashes on the license plate number string to specify the hash chain (single linked list) on which the tickets for the vehicle with that license is stored. Each node on a hash chain has the following struct (found in hasdb.h):

```
struct vehicle {
    char *state;            /* state on license plate */
    char *plate;            /* id on license plate the plate number */
    unsigned int tot_fine;  /* summary field; all tickets */
    unsigned int cnt_ticket;/* number of tickets unpaid */
    struct vehicle *next;   /* pointer to next vehicle on hash chain */
    struct ticket *head;    /* pointer to the vehicles first ticket; add tickets to end */
};
```

A vehicle has an entry in the database ONLY if it has at least one unpaid ticket.

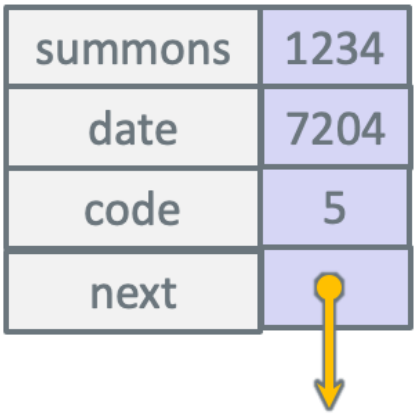
Below we show a visualization of the struct above.



- A vehicle is identified by the values in two member fields, plate number and registration state strings.
- The tot_fine member is the total fine for all tickets this vehicle has (they are all on the linked list pointed at by member head).
- The cnt_tickets is the number of tickets for this vehicle.
- The next member points at the next vehicle (struct vehicle) on the collision chain.
- The head member points at a linked list of unpaid tickets for this vehicle.

Each unpaid summons (ticket) uses the following struct (all tickets in the database are unpaid).

```
struct ticket {
    unsigned long long summons;    /* summons or ticket id */
    time_t date;                   /* date summons was issued */
    unsigned int code;             /* fine code 1-99 */
    struct ticket *next;           /* pointer to next ticket */
};
```



Below we show a visualization of the struct above.

- The summons field is the id of the summons (it is converted from a string to an unsigned long long integer because the value is so large). The starter code includes a function to convert a summons string to an unsigned long long. By storing the unsigned long long in the struct takes less space than storing a string.
- The date member is in a date string “MM/DD/YYYY” format (01/24/2020) that is encoded into Linux time type (called a time_t). The starter code includes a function to convert a date string with this format into the Linux time_t type.
- The next member points at the next ticket for this vehicle in the linked list
- The code member is a number from 1 to 99 identifying the type of the summons (the violation). You are supplied with functions that read up the fine table into a table in memory. The in memory table has a test string description and a fine amount. By just encoding a summons type number in the ticket field saves space you do not have to store the test strings in every ticket.

Here is an extract of the fine codes (found in file tests/in/Fines.csv)

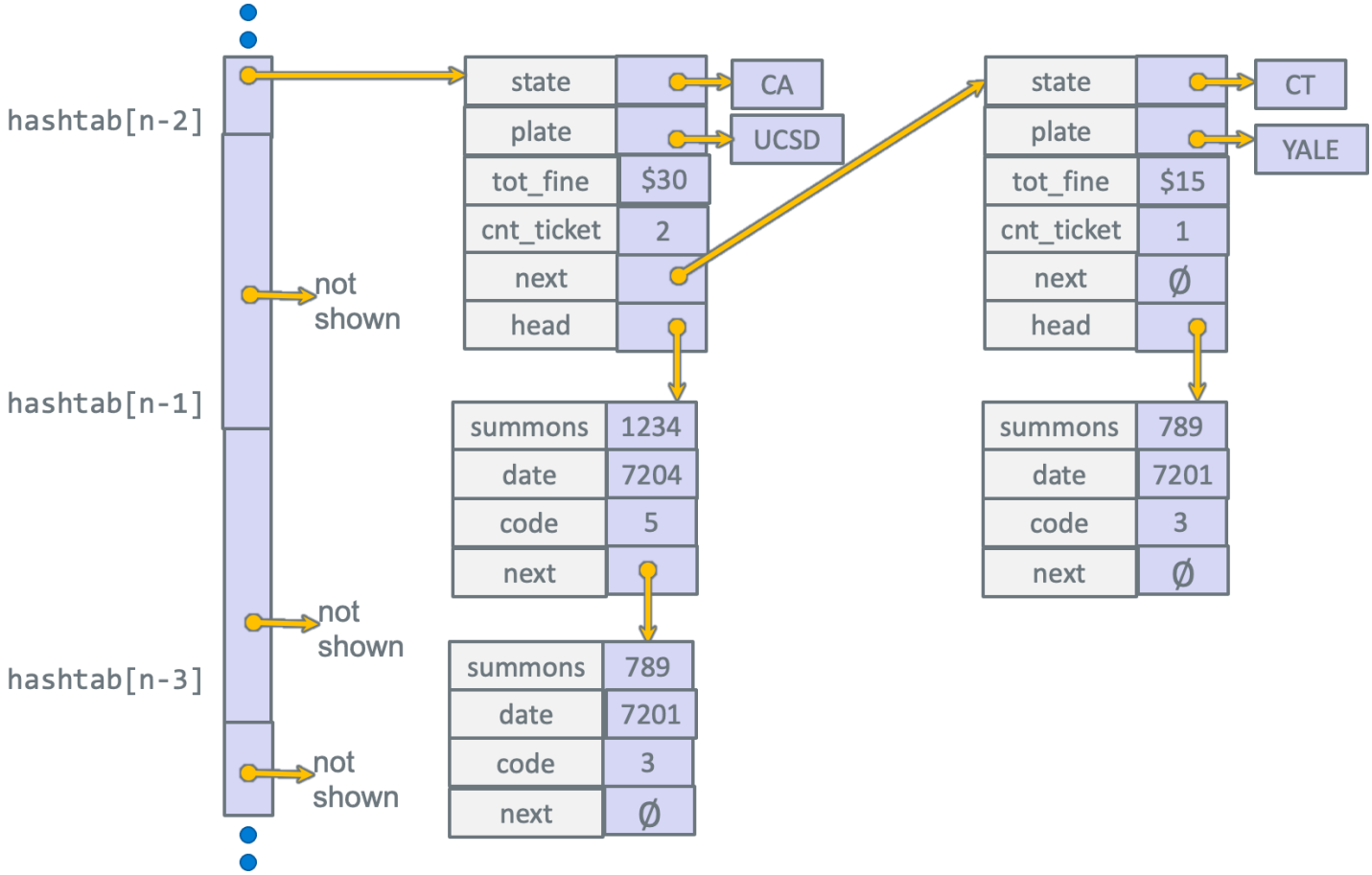
VIOLATION CODE	VIOLATION DESCRIPTION	FINE (DOLLARS)
1	FAILURE TO DISPLAY BUS PERMIT	515
2	NO OPERATOR NAM/ADD/PH DISPLAY	515
3	UNAUTHORIZED PASSENGER PICK-UP	515
4	BUS PARKING IN LOWER MANHATTAN	115
5	BUS LANE VIOLATION	115
6	OVERNIGHT TRACTOR TRAILER PKG	265
7	FAILURE TO STOP AT RED LIGHT	50

The fine table (*fineTable) has the following format, one element for each code. So Code 1 data is in fineTable[1];

```
struct fine {
    char *desc;             /* text description of code */
    unsigned int fine;      /* value of the fine in $ */
};
```

When a ticket is inserted into the database (your function in insert_ticket.c) the following operations are performed: YOU CAN ONLY USE MALLOC. Use of calloc() or zeroing out malloc's space before filling with data (either with code you write or using routines like memset() or bzero()) is poor coding practice and will result in a large number of points being deducted.

1. Hashed the license plate string to determine the hash chain
2. The hash chain is searched to find the vehicle, (strcmp on both state and plate must match). if it is not found a new struct vehicle is malloc()'d for the vehicle, Then the struct vehicle members are filled in with the vehicle information. You can use strdup() to allocate space for the state and plate strings. **The new vehicle struct is then inserted AT THE FRONT OF THE HASH CHAIN.**
3. If the vehicle was found or after a new vehicle node was added, a new struct ticket is malloc()'d and the ticket information is used to fill in the struct ticket members. **The new struct ticket is inserted AT THE END of the linked list of tickets (pointed at by the head member) for this vehicle.**
4. Each time you add a summons to the linked list (starting from member head) you must also update the member tot_fine in the vehicle struct, by adding in the cost of the new fine to the running total of fines in tot_fine. How this is done is described in the comments in insert_ticket.c You must also update the member cnt_ticket by adding 1 to account for the ticket you just added.



Above is a visualization of the hash table. In this example we show just one chain that contains two vehicles, one has two tickets and the other has one ticket. The hash function (unspecified) caused these two vehicles to collide on the same chain.

From the location of the nodes in the hash table and the rules for insertion given above you see that the vehicle YALE,CT was inserted as the first entry in the database with summons number 789. This was followed by vehicle UCSD,CA since new vehicles are added at the front. For UCSD,CA, summons 1234 was the first summons and then summons 789 was added to the ticket linked list since new summons are added at the end of the ticket linked list.

To search for a vehicle (your function in vehicle_lookup.c) you are supplied plate,state:

1. Hashed by the license plate string to get the hash chain
2. The hash chain is searched to find the vehicle, (strcmp on both state and plate must match).
3. Return a pointer to the matched vehicle struct or NULL if not found.

To delete a summons (pay the ticket) (your function in del_ticket.c), you are supplied with plate,state,summon string:

1. Hashed by the license plate string to get the hash chain
2. The hash chain is searched to find the vehicle, (strcmp on both state and plate must match). If not found the function returns a -1.
3. Convert the summons string using strtosumid() - how to do this is described in the comments in del_ticket() to a long long unsigned number
4. Search the ticket linked list to find the entry whose summons member number matches the converted number, If not found return -1
5. If a match for the summons is found, then delete the ticket/summons from the ticket linked list freeing all heap memory used by the struct ticket node
6. Update the members tot_fine and cnt_ticket in the struct vehicle node the ticket was linked to. If the cnt_ticket goes to zero (and the tot_fine should also go to zero, if not you have an error in your code) then there are no more tickets for this vehicle. **When there are no more tickets linked to a vehicle, you delete the struct vehicle from the hash chain and free all the heap memory (including the memory you allocated for state and plate using strdup) used by the struct vehicle node**.

To free up all the memory when the program exits (to look for memory leaks and to pass valgrind) (your function in free_tickets.c):

1. For each hash chain in the table that is not null, go to each struct vehicle on the hash chain and delete all the tickets while freeing up the memory. When the last ticket is deleted, save the pointer to the next struct vehicle on the hash chain and delete the struct vehicle that no longer has any ticket, freeing all the memory associated with that struct vehicle.
2. If the saved pointer is NULL, you have reached the end of the hash chain, go to step 1 for the next chain to process.
3. If the saved pointer is not NULL, Go to the next struct vehicle on the chain (using the saved pointer) and go to step 2.
4. Free the memory allocated for the hash table.

Description of the Command Line Options

`./parking -d Tickets.csv -f Fines.csv [-t size] [-s]`

Flag	Description
`-d`	Specifies the name of the csv file where the ticket/summons are stored. This option is **mandatory**.
`-f`	Specifies the name of the csv file where the Fines information (description of the parking violation and fine for the ticket) for each of the fine codes are stored. This option is **mandatory**.

`-t`	Allows you to override the size of the default hash table size TABSZ found in misc.h. It is often 1helpful to set the table size small when testing to make sure your hash chain code is working. The test harness uses -t 3 a lot as that will force hash collisions on even small datasets. **Default: see TABSZ value in misc.h**
`-s`	Used by the test harness and gradescope to turn off prompting in the command interface. WHen testing interactively you should not use this option. **Default: command prompts are enabled**

Coding and Program Development Requirements

1. You will write your code only in C and it must run in a Linux environment on the pi-cluster.
2. Only edit the following files that you will submit
    a. del_ticket.c
    b. free_tickets.c
    c. insert _ticket.c
    d. vehicle_lookup.c
3. Using an editor (like vim) insert your split_input() function and any helper functions from PA2 into the file split_input.c.
4. Many of the routines are passed argv as an argument. Argv is used when printing error messages to stderr. The Linux/Unix standard is that error messages are identified by the program that produced them. The identification is the string contained in argv[0]. So you would write error messages like:

```
fprintf(stderr, "%s: -your message here -\n", *argv, ....);
```

5. You can only use malloc() in any routines you write. Use of calloc() will cause points to be deducted. Calloc() is used to create the hash table as that is its initial value, all null pointers.
6. You may not use memset(), bzero() or similar functions in your code. Do not zero out malloc()'d memory and then set the values properly. This includes writing loops to zero malloc'd memory. This will slow your code and is poor practice. We will deduct a large number of points for this practice.
7. You must test your program on the pi cluster.
8. You must pass all supplied tests ***before submitting*** to gradescope for the complete set of MVP tests and the Field Trial Tests.

How to Approach This Programming Assignment

The starter files are contained in the `cs30sp22` public folder (`../public` relative to your home directory) in a tarball (a common way to distribute files in Linux). From your home directory, run the following to create the directory PA3 in your home directory:

```
tar -xvf ../public/PA3.tar.gz
```

Inside the directory at the top-level the following files are of interest:

Makefile: The Makefile you will use to develop your program. Please examine the Makefile to understand the various targets (especially the test and demo targets).

DemoRecords: These are the records in the database test/in/Tiny.csv in an easy to read format so you can use them to type in input when testing your code interactively using the make target %make demo

libpa3.a: This is a library contain functions not supplied in source form

vehicle_lookup.c: This is the file where you will write the function vehicle_lookup()

free_tickets.c: This is the file where you will write the function free_tickets()

insert_ticket.c: This is the file where you will write the function insert_ticket()

split_input.c: Your split_input() function from PA2

del_ticket.c: This is the file where you will write the function del_ticket()

hash_subs.c: This file contains helper functions that you will need to call when writing your four functions.

The rest of the files contain supplied sources for you to examine as needed (or are curious how things work). **You should not need to edit any of these supplied sources including any of the header files.**

IMPORTANT READ THIS!!!! Before editing any files!!!!!

BEFORE you edit one of the files where you will place your functions, look in the file, you will see the following lines (using the file free_tickets.c as an example). You will fill in your code between the { }

```
//uncomment the next line when you want to use your routine
//#define MYCODE
#ifndef MYCODE
TODO(USING THE SOLUTION FUNCTION NOT MY CODE)
#else
/*
```

```
 * free_tickets
 *      tears down the database freeing all allocated memory
 * args
 *  hashtab pointer to hashtable (pointer to an array of pointers)
 *  tabsz   number of elements in the hash table
 */
void
free_tickets(struct vehicle **hashtab, uint32_t tabsz)
{
/* insert your code here */
}
#endif
```

Please make sure that when editing the file you do not delete or reposition these lines near the top:

```
//uncomment the next line when you want to use your routine
//#define MYCODE
#ifndef MYCODE
TODO(USING THE SOLUTION FUNCTION NOT MY CODE)
#else
```

Or the LAST line in the file:

```
#endif
```

Now if the line `//#define MYCODE` is commented out, as it is in the example above, when you run make your function written after the `#else` will not be compiled, and the compiler will link in the same function from the solution code. So you will get a properly running function to test with.

You will get a message at compile time alerting you that this is happening. For example:

```
free_tickets.c:14:1: note: ‘#pragma message: TODO - USING THE SOLUTION FUNCTION NOT MY CODE’
   14 | TODO(USING THE SOLUTION FUNCTION NOT MY CODE)
      | ^~~~
```

This is just a message and the program is still being compiled properly. It is just telling you that it is NOT using your code, but is using the solution code.

Make sure you get none of these messages right before you turn in your code!

When you want to compile and test your code, just remove the // in front to get

```
#define MYCODE
```

Then your function implementation will be compiled and used.

It is suggested that the first function to write is vehicle_lookup() it is easy to write and should take less than 40 lines to implement (not including comments). The suggested next function to write is del_tickets(). It is a good way to learn the structure of the hash table by writing code to free up all the vehicle and ticket structures.

All the files that you need to edit contain descriptions on how the functions work and what the parameters passed to them are.

PA3 Demo/Debug - On the pi-cluster First before editing any files

After you extract the starter files and before doing anything else, (before editing any file) **on the pi-cluster** do a

```
% make demo   (or if you like: % make debug)
```

It is suggested that you use make debug to debug your routines before trying to pass the test files.
You will notice the program will compile and run in an interactive mode. This will use solution code stored in the supplied library. You can play with the commands (just input a return to see the options) to get a feel how to use the commands (especially ones that are helpful for debugging your code).
The file **DemoRecords** documents the data that is loaded into the demo database so you can make queries for specific vehicles you know are in the database. The csv file for the demo database is in tests/in/Tiny.csv

It is **important to try the following commands** in debug/demo mode so you know what they do when debugging your code.

D	dumps the entire database so you can see what the hash chains and ticket linked lists look like
C	dumps a specific hash chain (like D above but just one chain) give the index number of the hash table
F	find a vehicle in the database and print it
I	insert a summons for a vehicle, give the plate, state, summons id, date and code
L	list the vehicle with the largest fine and largest number of tickets
V	verify the various member fields in the database, tries to find problems (limited if links are bad)
P	pay a summons for a vehicle, give the plate, the state and the summons id
Q	quit

Press return to see the command list in the demo. Here is an example

```
Input command: D
Chain 0:
   Plate: HLC3177, State: NY, Tickets: 1, Total: $65
      Ticket #0001: 1133401569,07/02/2018,(21) $ 65 NO PARKING-STREET CLEANING
   Plate: GER9006, State: NY, Tickets: 1, Total: $95
      Ticket #0001: 1131610520,07/02/2018,(17) $ 95 NO STANDING-EXC. AUTH. VEHICLE
   Plate: GTR7949, State: NY, Tickets: 1, Total: $65
      Ticket #0001: 1130964875,06/08/2018,(24) $ 65 NO PARKING-EXC. AUTH. VEHICLE
   Plate: HXM7361, State: NY, Tickets: 1, Total: $115
      Ticket #0001: 1121274900,06/28/2018,(46) $115 DOUBLE PARKING
Chain 1:
```

```
   Plate: HZJ8359, State: NY, Tickets: 1, Total: $65
      Ticket #0001: 1133401636,07/02/2018,(21) $ 65 NO PARKING-STREET CLEANING
   Plate: HDG7076, State: NY, Tickets: 1, Total: $95
      Ticket #0001: 1131599342,06/29/2018,(17) $ 95 NO STANDING-EXC. AUTH. VEHICLE
Chain 2:
   Plate: HPC9135, State: NY, Tickets: 1, Total: $65
      Ticket #0001: 1133401594,07/02/2019,(21) $ 65 NO PARKING-STREET CLEANING
   Plate: HZN6473, State: NY, Tickets: 1, Total: $65
      Ticket #0001: 1133401570,07/02/2018,(21) $ 65 NO PARKING-STREET CLEANING
   Plate: GLS6001, State: NY, Tickets: 3, Total: $345
      Ticket #0001: 1105232165,07/03/2018,(14) $115 NO STANDING-DAY/TIME LIMITS
      Ticket #0002: 1105232166,07/04/2018,(14) $115 NO STANDING-DAY/TIME LIMITS
      Ticket #0003: 1105232167,07/05/2018,(14) $115 NO STANDING-DAY/TIME LIMITS
Input command: P GLS6001 NY 1105232165
Plate: GLS6001, State NY, Summons 1105232165 paid
```

After thoroughly testing your program in debug/demo mode, you can run the test harness and will pass all the tests (so you can see what is going on when everything works running the test cases):

```
% make test
```

Submitting your code

Comment and check your code follows the CSE30_C_Style document

To get all the style points, read over the style document and clean up your code by following the guidelines. In PA3, style will be more enforced than in PA2.

Submit to Gradescope under the assignment titled "A2: extract"

After you pass all the MVP tests in the test harness and only then, will you submit to gradescope the following files:

```
del_tickets.c free_tickets.c insert_ticket,c vehicle_lookup.c
```

You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all the files and upload the .zip to the assignment. Ensure that the files you submit are not in a nested folder.

After submitting, the autograder will

1. Check all files were submitted
2. That the program compiles without errors
3. Rerun the public tests and then the private tests

Make sure to check the autograder's output while submitting!

Grading
The assignment will be based on 60 points:

5 points on programming style and comments
35 points on passing the MVP public tests **(stdout data only; stderr output will not be checked)**
20 points on passing the Field Trial private tests

Note: The coding style is expected to be followed and will be evaluated stricter than PA2.

The Test Harness

This test harness is basically the same as previous PA's (it has the same hierarchy of files). The script runtests are a little longer. Feel free to modify it as required.

It is best to start with running the demo with your functions to test and debug before trying to use the test harness. Do not depend on the test harness to test all your cases. There are debug routines in the supplied code that you should use.