

CSE 30 Fall 2022 Programming Assignment -#6 - (Vers 1.11)

Due Friday November 4 , 2022 @ 11:59PM

Remember: 'Start Early'

Grading (50 points total) What You Need To Do

You must complete all of the following items by the due date to get all points.

The last day to turn in is Monday Nov 7, 2022 at 11:59 PM.

There are **50 points available** for this PA that are distributed as follows:

- ☐ Up to 2 points for following the C style guidelines.
https://docs.google.com/document/d/1jltOTaolKNfF7QTnCNJQ9KxgkcllDJ_rD86u7nHnHZ4/edit?usp=sharing
- ☐ Up to 8 points (at our discretion) if the following files compile without warnings. This has to be a meaningful attempt to write the code that meets the specifications of the program. Random C statements or empty files will not be awarded any points. **These are the only files that are part of PA6:**
 - freetickets.c (2 points)
 - largest.c (2 points)
 - sumlookup.c (2 points)
 - vehlookup.c (2 points)
- ☐ Up to 25 points for passing the tests in the included test harness
- ☐ Up to 10 points for passing the gradescope tests (these will be run after the late deadline).
- ☐ Up to 5 points canvas checkpoint quiz (due Tuesday 11/1). No slip days or late submissions.

Assignment – Pointers and Data Structures – a Memory based Database

PA6 is the first part of a two part project where you will be working on an in-memory parking ticket database. The data we are using is actual data summons (ticket) data extracted from the New York City summons database for the fiscal year 2018 (~July 2018 to ~June 2019). In PA6 you will be writing four routines to replace ones provided with the starter code. In PA7 you will be writing two more.

1. Practice in writing data structures in C using pointers combined with self-referential structures
2. Basic memory runtime memory allocation and deallocation with malloc() and free().
3. Using the linux tool valgrind to test if your use of malloc() and free() is correct.
4. Working with hash table chains and 2-dimensional linked lists.
5. Writing code to traverse every record in a database to find a specific entry
6. Writing code to use a hash function to find a specific entry in the database

7. Practice doing incremental code development in the context of a fully operational program

(something you will do a whole lot of when you graduate), by writing replacement routines for existing ones.

Both PA6 and PA7 will use the same starter code. The starter code includes a fully operational database that will pass all the test harness tests without any additional code development. The database program for PA6 and PA7 that is created by make is called *parking*. Included in the database program **parking** is an interactive (but very simplistic) command interface that allows you to run both a number of debugging commands (to help debug your code) as well as several basic database query and update commands.

When working on an existing software project as part of a development team in industry, one of the most common tasks is to replace some of the current code with a new version. There are several reasons for this such as: to fix bugs, improve performance or add new features. In this PA you will practice working in that situation by writing replacement software for a fully functional database program called parking.

The best overall approach for the PA is to focus on incremental development by writing one routine at a time to replace the one provided in the starter code. To use your replacement version you must recompile the database code by typing make and then you can test your code.

You can test your code two ways: (1) interactively (type make demodb to try it) and (2) use the test harness. The test harness is the same one as used in PA5, but it just runs different commands (see in/cmd files which is where this is done), data files and expected output files. The tests are designed to stress the routines that you write. You will be able to switch back and forth for each function you write between your version of the code and the one in the starter file by editing a SELVERS.h (described below) and then doing a recompile with make. This will allow you to test just one replacement function at a time (and any combination of replacement functions as you want).

We have included the source code for several of the starter file routines that you do not have to write replacement versions for, to help you understand the program and assist in debugging. You are strongly urged to examine these source code files, especially main.c, before starting this programming assignment.

When reading through the source code that loads the database from a csv file (this is in the file loaddb.c) you will observe that it calls token(). This is the same token() function that you wrote in PA5 part 1. You do not have to provide a token.c from PA5 as a fully operational one is included with the starter code (but not in source code format). Data extraction functions like token() are routinely used in database systems.

Modifying SELVERS.h to Control the Versions in the Compile

In the starter file there is a file called SELVERS.h whose contents are shown below:

```
/*
 * Each line below specifies which version to use
 * Uncomment a line to use YOUR replacement version
 */
```

```

/*
 * The following functions are for PA6
 */
#define MYFREETICKETS    // when defined will use your freetickets.c
#define MYLARGEST        // when defined will use your largest.c
#define MYSUMLOOKUP      // when defined will use your sumlookup.c
#define MYVEHLOOKUP      // when defined will use your vehlookup.c

/*
 * The following functions are for PA7
 */
#define MYINSTICKET      // when defined will use your insticket.c
#define MYDELTICKET      // when defined will use your delticket.c

```

As you can observe above, the file has two sections, one for PA6 and one for PA7. In PA6 you will be writing replacements for the functions in `freetickets.c`, `largest.c`, `sumlookup.c` and `vehlookup.c`. In PA7 you will be writing replacements for `insticket.c` and `delticket.c`. You should only work on the PA6 files now and get them finished and submitted before starting on PA7.

Each of the `#define` lines is what we will call a *selection* line for a specific replacement file (as described in the comment in the line). For example the following line from the file `SELVERS.h`:

```

#define MYLARGEST        // when defined will use your largest.c

```

The line above, controls whether the solution version or your version of the code found in the file `largest.c` is used, ***the next time make is used to recompile parking***. You *must recompile for the change to have an effect!*

In order to use your replacement version(s) in the program you must:

1. Remove the `//` from in-front of the line that lists the file you want to replace. Like any source file, use a text editor like `vim`, then write the file out. When a *selection* line is ***commented out*** in `SELVERS.h`, you are telling the compiler to use the solution code and ignore the contents of the replacement version. You may uncomment any combination of lines as you want. If you want to use the starter code version, just make sure that the appropriate line in `SELVERS.h` is commented out.

For example say you want to use just your version of `freetickets.c`. The file `SELVERS.h` would look like

```

/*
 * The following functions are for PA6
 */
#define MYFREETICKETS    // when defined will use your freetickets.c

```

```

// #define MYLARGEST          // when defined will use your largest.c
// #define MYSUMLOOKUP       // when defined will use your sumlookup.c
// #define MYVEHLOOKUP       // when defined will use your vehlookup.c

/*
 * the following functions are for PA7
 */
// #define MYINSTICKET        // when defined will use your insticket.c
// #define MYDELTICKET        // when defined will use your delticket.c

```

- run make, it will recompile the database program parking using either (1) your replacement version (if the corresponding line in SELVERS.h **is not commented out**) or (2) the starter code version in libpas6.a (if the corresponding line in SELVERS.h **is commented out**).

Overview: A database of Unpaid Tickets (Summons)

There are two datafiles that are read up at the start of program execution. The start code includes routines to read these files for you (you will not have to write any of this code). Both data files are in CSV format. One datafile lists the various parking fines by a code number, the amount of the fine (in US \$) and a brief description of the fine. The other datafiles contain extracts from the NYC summons database for 2019. The only difference in the various summons datasets are in the number of records contained in each. Having different length datasets enables you to test basic functionality on small datasets first and then work up to larger ones. All the summons dataset have properly formed records with no errors. One of the datasets was modified (in/Dups.csv) and contains several duplicated records (used to check database insert functionality).

The summons datasets (parking tickets) all have the following structure (the first 5 records are shown and the first record is always a header record).

```

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 (in order):

Summons Number:	ticket/summons number (a unique value)
Plate ID (the license plate id):	License plate id (often called a number, but is a string of chars)
Registration State:	Two letters for the US state that issued the plate
Issue Date:	date of the ticket in the format mm/dd/year (month/day/year)
Violation Code number:	Number (1-99) that specifies the violation type

There are 99 types of parking violations, each assigned a number from 1 to 99. By using a short index number you can record the violation while consuming a small amount of space in the database (versus storing the full information with every summons). The violation number is then used as an index into another table that has expanded details on each type of violation. This avoids repeating all the summon detail information in every entry in the database. It also allows details of the fines to be easily changed, like increasing the fine without having to update every record in the ticket database.

READ THIS: The summons database does NOT have any corrupt records in it. A summons is associated with just one vehicle! In the test harness, testD checks code correctness using whole record dups using a table (in/Dups.csv that was made with an editor by hand) that has copies of entire records. Duplicate records still have the consistency of one vehicle per summons.

The various summons datasets included with the starter code repository are:

```
in/Empty.csv    // a dataset with just a header entry, 0 (none) summons
in/Tiny.csv     // a dataset with 11 summons
in/Small.csv    // a dataset with 99 summons
in/Medium.csv   // a dataset with 1001 summons
in/Large.csv    // a dataset with 50000 summons
in/Dups.csv     // a dataset with 15 records (same as Tiny but with dups)
```

The fine table is one of the datasets read up into a dynamically allocated array at the start of execution (before the summons dataset is read (see main() in main.c). The fines dataset is stored in a three field CSV file with the following format. The three data fields are as follows (in order):

Fine code:	code number recorded in the summons dataset
Fine description:	text string describing the fine
Fine amount:	amount of the fine in US dollars.

Here are the first few records in that file (in/Fines.csv). The first record is a header file.

```
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
```

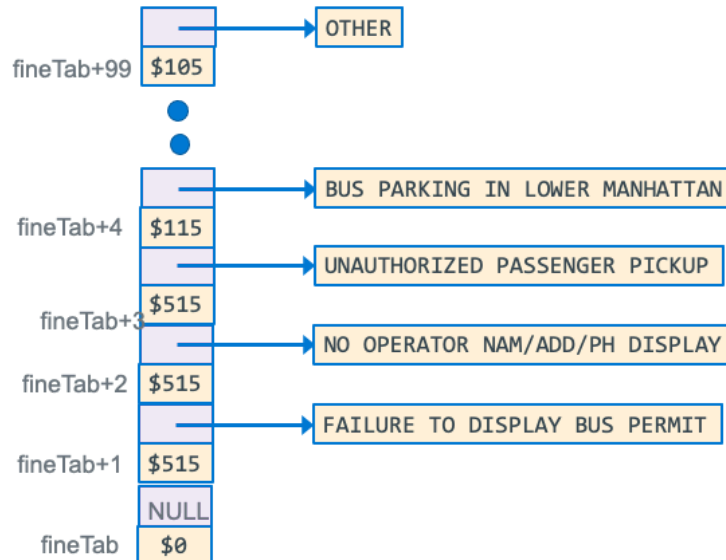
By just encoding a summons type number in the ticket field saves space as all you need to store with each summons is the code number, you do not have to store the description text string or fine amount in every ticket.

The fine table (*fineTable) has the following in-memory format for each element in the array, one element for each code. The specification for this is found in the file parking.h

```

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

```



When the fine table is built by the supplied starter code at program start up, an empty element is inserted at index 0 into the table. Since ticket code numbers range from 1 to 99, you can use the code number directly to find the entry for the code rather than always having to subtract 1 to get the correct element. For example if you wanted the fine (variable `finecost` below) for a ticket with the code number 3, “unauthorized pickup” you would look at

```
finecost = fineTab[3].fine;
```

```
finecost = fineTab[ticket->code].fine; // ticket is pointer to struct ticket
```

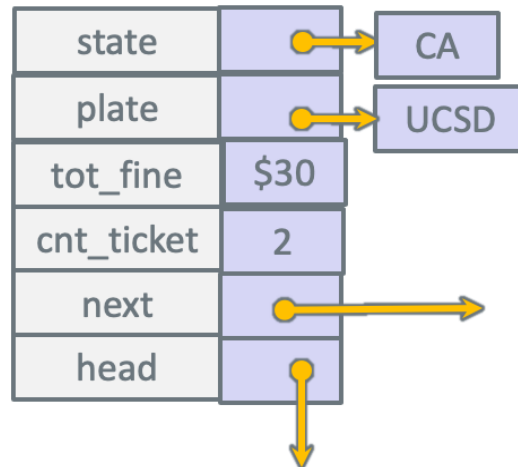
The database uses a hash table that is hashed on the license plate name string (the license plate is the primary key in database jargon). The license plate string is hashed to one of the hash chains (single linked list). Each record on this hash chain describes a vehicle that has at least one unpaid ticket (it may have more). Vehicles that do not have an unpaid summons (ticket) are not in the database (it is a database of unpaid tickets).

Each node on a hash chain has the following struct (found in the file `parking.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 add vehicles at front */
    struct ticket *head;   /* pointer to the vehicles first ticket; add tickets to end */
};

```



Above we show a visualization of a struct vehicle record on the hashchain.

The database is optimized (by hashing) for finding tickets quickly given a license plate number and the two letter code for the state where the license plate was issued. In this single hash table design, other queries, like finding the vehicle with a specific summons, requires searching the entire database (called a full file search in database jargon). In a real database there would be multiple hash tables (called index tables) that allow rapid access for frequently made queries (like find the vehicle given a summons number). **Remember, a vehicle has an entry in the database ONLY if it has at least one unpaid ticket.**


- A vehicle is **uniquely identified by two member strings**: **plate** number string and **state** string. These two strings are both allocated memory **using strdup()**.
- The **tot_fine member** is a calculated (not in the dataset read up) summary field that contains the total fine for all tickets this vehicle has (the dual tickets are all on the linked list pointed at by member **head** in the struct). The **tot_fine** member is increased in value by the fine associated with each new ticket inserted into the database for this vehicle. The **tot_fine** member is decreased in value by the fine associated with a ticket when the fine is paid (the ticket is deleted from the database)
- The **cnt_tickets** member is a calculated value based on the current number of tickets for this vehicle. The **cnt_tickets** member increases when a ticket is added to the vehicle and is decremented when a ticket is paid.
- The **next** member points at the next vehicle (struct vehicle) on the hash collision chain (linked list from the hash table).
- The **head** member points at a linked list of unpaid tickets (struct ticket) 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 */
};
```

Struct member descriptions are:

- **Summons:** id of the summons **converted** from the **summons dataset character string** to an **unsigned long long integer** (the summons id is a big number). **Each summons id is unique**. The starter code includes a function to convert a summons string to an unsigned long long (it is located in the file subs.c and is called **strtosumid()**). By storing the unsigned long long in the struct takes less space (bytes) than storing a string. When working with this **summons** member, remember that when given a character string containing a summons number you first have to convert it to an unsigned long long. **Once converted, you can directly compare summons when stored as unsigned long long in a struct ticket instance.**
- **Date:** date stored using the **Linux time type**, called a **time_t** date. This value is derived from the date character string in the “MM/DD/YYYY” format (01/24/2020) from the summons dataset. The starter code (in subs.c) includes a function **strtoDate()** to convert a date string with this format (and used in the NYC dataset) into the Linux **time_t** type.
- **Next:** a pointer to the next ticket for this vehicle in the ticket linked list
- **Code:** integer number from 1 to 99 identifying which summons (the violation) this is.

summons	1234
date	7204
code	5
next	

There are four global variables defined in main.c that are used by most of the routines.

Since almost all the routines need to access these variables, passing them as arguments to each function actually slows down the code and makes for excessively long and hard to read function parameter lists. Since these are global variables, they can be used in any function directly. While many folks assume that all global variables are wrong to use, this is a situation where one can demonstrate they have merit. However, be careful to not accidentally change the contents of global variables in any routine (only main() should do that).

Here is the list of global variables (these are declared in parking.h):

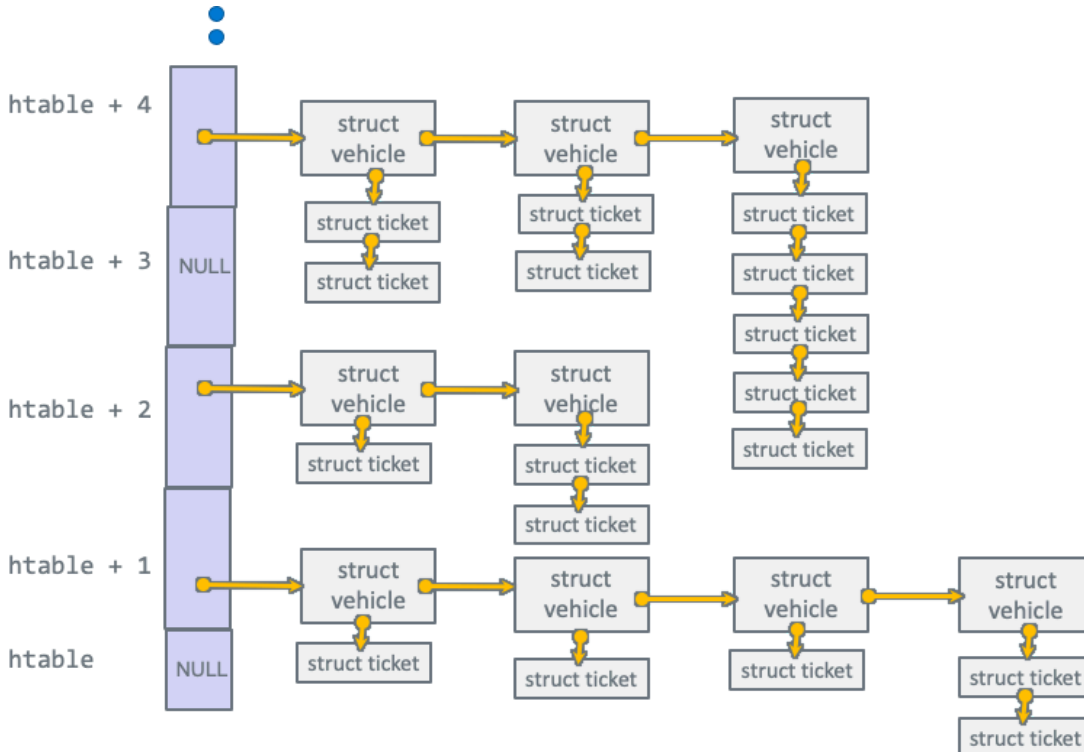
```
char *argv0;           /* pointer to argv0 for error messages */
struct vehicle **htable; /* pointer to hash table */
struct fine *fineTab;   /* table of fines by code 1-99 */
uint32_t tabsz = TABSZ; /* hash table size */
```


1. **argv0** is used for printing the name of the program in error messages
2. **htable** is a pointer to the hash table that is allocated in main() during startup using calloc9) so all the element entries are NULL pointers, indicating an empty database.
3. **fineTab** is a pointer to the fines lookup table described above.
4. **tabsz** is the number of elements in htable, it is the value you use with the hash value to find the hash chain. The function hash() returns a uint32_t. Here is an example how to do this:

```
uint32_t hashval;
struct vehicle *chain;
hashval = hash(plate) % tabsz
chain = *(htable + hashval);
```

Overall Database structure

The database has two types of linked lists. The first type is a linked list of vehicles whose plate id strings hash to the same hash table index. The second type of linked list is a list of tickets for a specific vehicle. The overall layout of the database in memory consists of a hashtable (htable) of pointers to the type struct vehicle. If a vehicle has at least one ticket it has a single struct vehicle to describe it. The vehicles are placed into the hash table by hashing the plate string (hash(plate)). Each vehicle has a linked list of tickets (summons) that were issued to the vehicle. Each vehicle has at least one of these ticket linked lists attached to it. The picture below shows this overall structure (note: null pointers at the end of the linked lists are not shown here).



In the picture above you can see the linked lists of struct vehicles (one for each car which is uniquely identified by the combination of its plate string and state string). Vertically you see the linked list of tickets that were

issued for the same vehicle. Each time a ticket is inserted into the database, the ticket is linked to the struct vehicle for which the ticket was issued. If the ticket was for a vehicle that is not already in the hash table, the vehicle is first added to the hash table, then the ticket is linked to the vehicle.

You should assume that a summons is associated with only one vehicle (plate,state as the unique id for a vehicle) in the database. For these PA's (6 and 7), you do not have to consider cases of corrupt data where a summons is associated with more than one vehicle. To use a data science term the data has been "fully cleaned/scrubbed".

insertticket() (file: insticket.c) - Part of PA7

To insert a ticket (summons) into the database, the function insertticket() is passed four strings: summons, plate, state, date, and an integer value: summons code number.

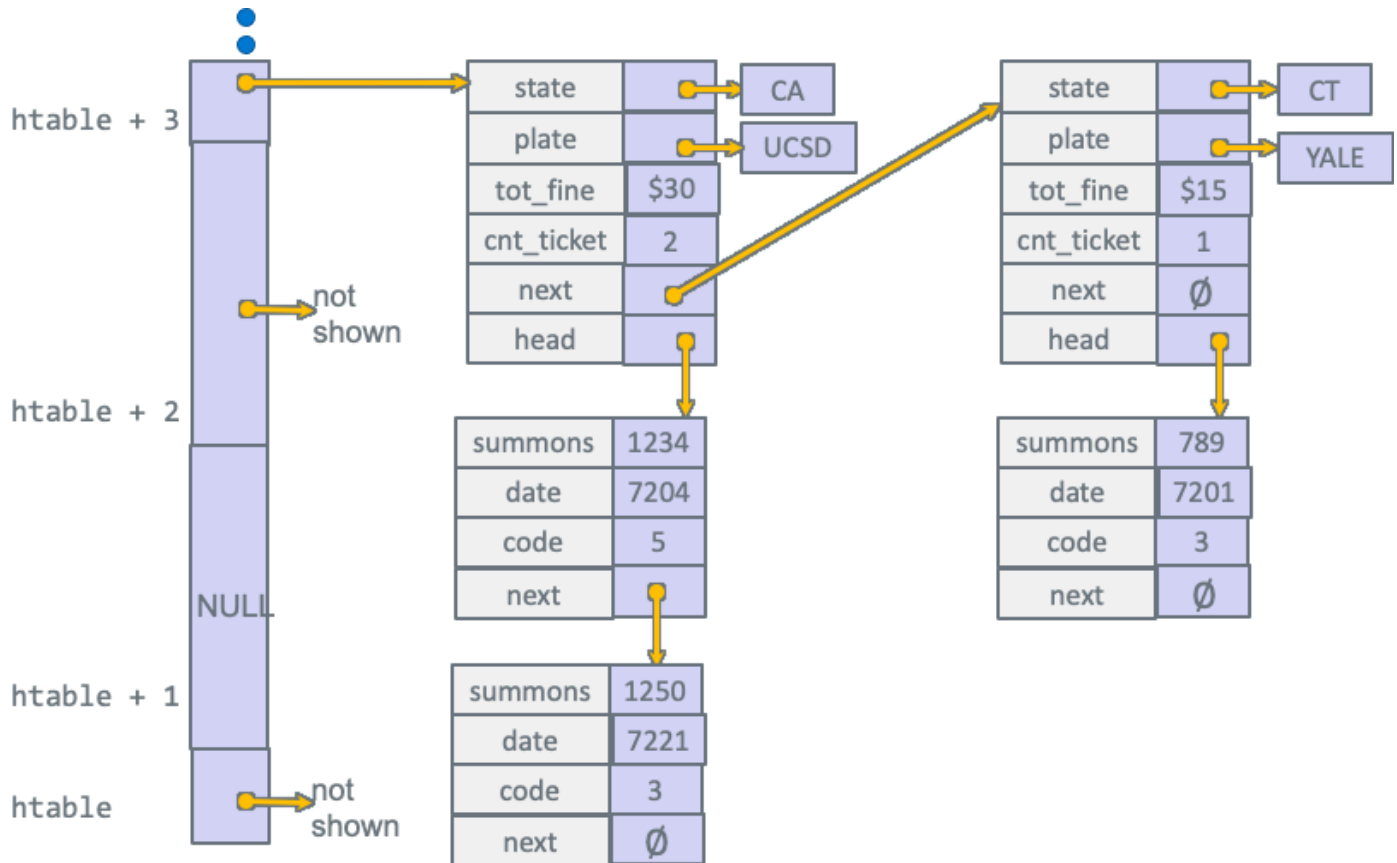
Make a call to vehiclelookup() to do step 1 and step 2 below. Vehiclelookup() is described later.

1. The license plate string is hashed (as above) to determine the hash chain
2. The hash chain is searched to find the vehicle, (strcmp() on **both state and plate must match**)
3. If the vehicle was not found, a new **struct vehicle** is malloc()'d for the vehicle and each of the struct vehicle members are filled in with the vehicle information. **You cannot use calloc() for this! 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 (the struct vehicle linked list).**
4. If the vehicle was already in the database or after a new vehicle node was added, a new struct ticket is malloc()'d (**You cannot use calloc() for this!**) and the ticket information is used to fill in the struct ticket members. Remember to use the converted values for summit and dateval when filling out a ticket's summons and date 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. If the summons is a duplicate (it is already in the ticket list for this vehicle - remember for the PA you should assume that a summons record is only associated with the same vehicle), it is not inserted again and an error message is printed (details on the error message in the starter file).** Since the dataset guarantees that a summons is only associated with one vehicle, if there is a duplicate summons it will be on the vehicle's ticket chain (just check the ticket chain as you traverse to the end). **Do not call sumlookup() to find duplicate summons.**
5. 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.
6. You use fineTab to get the fine. Fine = fineTab[code].fine and add this to the tot_fine member in the vehicle structure.
7. You must also update the member cnt_ticket in the vehicle structure by adding 1 to account for the ticket you just added.

Below 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 (linked list).

From the location of the nodes in the hash table and based on 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 1250 was added to the end of the ticket linked list since new summons are added at the end of the ticket linked list.



vehiclelookup() (file: vehlookup.c) - Part of PA6

To search for a vehicle, `vehiclelookup()` is passed two strings: plate and state.

1. Hash 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). **This is important**, the `hash()` function is passed the plate id string only, so you must check that both the plate id and the state match. You can have two cars with the same plate id, but are issued by different US states (so they are different vehicles).
3. Return a pointer to the matched vehicle struct or NULL if not found.

sumlookup() (file sumlookup.c) - Part of PA6

To search for the vehicle that has a specific summons, `sumlookup()` is passed a string to the summons number. Since the database is hashed by license plate number there is no easy way to directly find the summons with hashing (so you cannot use `vehiclelookup()` here). In a real database there would be another hash table that was based on hashing summons numbers. So to find the summons in our database, you have to do an iterative search of the database one hash chain at a time (this is called a full table scan).

1. For each hash table entry that has a chain (where the element is NOT NULL), you have to search every vehicle entry (struct vehicle).
2. First convert the summons string to an unsigned long long `sumid` (this is done for you in the starter code).
3. Then for each vehicle on the hash chain, search the vehicles ticket chain (starting from the vehicles head member) and compare the summons id in the ticket to `sumid` that you are looking for. If you find a match, return a pointer to the vehicle structure.
4. If you reach the end of the ticket chain for the vehicle, go to the next vehicle on the chain and search its ticket chain. Continue until either you find the summons or reach the end of the chain.
5. If you reach the end of the chain, go to the next hash table entry that is not NULL and go to step 3.
6. After you have searched all of the hash chains in `htable` and have not found the summons on a ticket list for any vehicle, return NULL (as the summons is not in the database).

largest() (file largest.c) - Part of PA6

To print to stdout the largest number of tickets for any vehicle and the largest total fine for any vehicle, the function `largest(void)` is called. This requires you to do an iterative search of the database one hash chain at a time (just like `sumlookup()` above and `freetickets()` below) (a full table scan).

1. `largest()` must visit every vehicle entry in the database (full table scan) (so you cannot use `vehiclelookup()` here).
2. Define and initialize two unsigned int variables to keep track of the largest fine and the greatest number of tickets.
3. For each hash table entry that has a chain (where the element pointer is NOT NULL), you must search every vehicle entry (struct vehicle).
4. For each vehicle on the hash chain:
 - a. compare the `tot_fine` member in the struct vehicle to the largest fine seen for a vehicle so far, and if the `tot_fine` is the **same or larger**, update the variable pointer `fine` to point at that vehicle.
 - b. compare the `cnt_ticket` member in the struct vehicle to the largest number of tickets seen for a vehicle so far, and if the `cnt_ticket` is the **same or larger**, update the variable pointer `count` to point at that vehicle.
5. When the end of the chain is reached, go to the next hash table entry that has an entry (where the element pointer is NOT NULL) and go to step 4.
6. After all the entries in the hash table have been examined, the `printf()` statements in the starter code will print the details of the vehicle with the largest total fine and the vehicle with the largest number of tickets and then return.

freetickets() (file freetickets.c) - Part of PA6

To free up all the memory before the program exits (to look for memory leaks and to pass valgrind) and to delete the database the function `freetickets()` is called (it has no arguments). **Do not use `delticket()` (that is part of PA7) when writing `freetickets()`.** `Delticket` deletes a specific ticket after a lookup operation. This requires you to do an iterative search of the database one hash chain at a time (just like `sumlookup()`) and `largest()` above) (a full table scan).

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 allocated to the struct ticket. Keep count of the number of tickets that are freed (deleted).
2. 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 (including the memory allocated for the strings state and plate).
3. 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.
4. 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.
5. When you have processed all the hash chains, the code at the bottom of the function (supplied in the starter code) will print "Empty Database" if the database did not contain any vehicles. It will then print the "Total number of tickets freed".

delticket() (file delticket.c) - Part of PA7

To delete a specific summons (pay the ticket). `delticket()` is passed three strings: plate, state, and summons. It then uses `vehiclelookup()` to find the vehicle. It does not use `sumlookup`, as that is a full table scan and `vehiclelookup()` is a hash table lookup so it is much faster. You should assume that the summons appears for only one vehicle (the dataset is clean) and the database is not corrupt. **Do not use `delticket()` to implement `freetickets()` it would be too slow!**

You may use a call to `vehiclelookup()` to do step 1 and step 2 if you like, though it is not recommended. It is better to do steps 1 and step 2 in this function.

1. Hash 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. If the vehicle is not found the function returns a -1. (use `vehiclelookup()` here)
4. Convert the summons string using `strtosumid()` - how to do this is described in the comments in `delticket()` to a long long unsigned number and starter code to accomplish this is given.
5. Search the ticket linked list to find the entry whose summons member number matches the converted summons string number, If not found return -1
6. 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
7. 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.

8. 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 allocated for state and plate using strdup()) used by the struct vehicle node.
9. Return a 0 to the caller.

Overall Operation

At program startup, main first parses the command line arguments using getopt() (see below) to get the name of the fines dataset file name, the summons dataset file name and an optional argument to over-ride the default hash table size (element count of pointers in the hash table). Main() uses calloc() to create the hash table and the fines table.

main() calls routines in the file loaddb.c (supplied starter code) to load the data files into memory. The fines table is processed first and if it has errors the program will not run. Next the summons dataset is loaded and any bad records are rejected by token() (the same token from PA5). There are no dataset files in the starter code test harness with bad records.

Main calls the command interpreter commands() (source is supplied in the file commands.c). Commands will allow you to interact with the database and includes commands to help you debug your code and commands that do actual queries. command() has a very simple interface that is typical of a prototype test interface. It contains a mix of debug functions and actual query functions. You leave the database by entering q on the keyboard (on typing cntrl-d) to return to main().

The last part of the part of main() frees up all the memory that was allocated on the heap. It calls a routine freefines() to free up the fines table entries and the fine table itself. main() calls freeticketc() to free up all the entries in the database, and then frees up the fine table itself and 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 parking.h to a specified value. It is often helpful 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 parking.h

-s	<p>This option is only used by the test harness and gradescope to turn off prompting in the command interface. When testing interactively you should not use this option.</p> <p>Default: command prompts are enabled</p>
----	--

Get the PA6 Starter files from Github and the setup

Step 1: You need to get the starter files for PA. Login into your account on the pi-cluster and run the following command. You will develop the code on the pi-cluster. **You must do the git clone on the pi-cluster.** If you run this on windows and copy the files over, windows may corrupt the files.

```
cs30fa22@pi-cluster-153:~ % git clone https://github.com/cse30-fa22/PA6_starter.git PA6
```

```
cs30fa22@pi-cluster-153:~ % cd PA6
```

In the repository you will find the following files:

1. **Makefile** for compiling your program. This file is complete and will not need to be modified. You should examine the file, to see that there are several targets in the Makefile
 - a. Test harness test groups: alltest, testA, testB and testC, testD testV1, testV2. The target alltest will run the test groups testA, testV1, testB, testC, testD and testV2 in order.
 - b. emptydb will start the database with an empty database (no entries)
 - c. demodb will run a small subset of the NYC summons database in interactive mode. This is a good way to run simple early tests and learn how the various commands work.
 - d. testP: does a performance run of the database. It generates an error file that contains how much time it took to run a set of commands. The error file is not an indication it failed, but is just used to report the time consumed by your program. This is not a functional test, but a performance measurement for working code only.
2. **main.c** is a starter file that is supplied in source code format. You will not need to edit this file.
3. **token.h** is a starter file which contains the function prototype for the token() and some constants used by main to call token(). This is the same file as in PA5. You will not need to edit this file (it is not turned in nor is your version of token() used).
4. **commands.c** is a starter file that is supplied in source code format. You will not need to edit this file.
5. **commands.h** is a starter file that contains the function prototypes for command.c
6. **freetickets.c** is a starter file where you will write your replacement freetickets() function for PA6
7. **delticket.c** is a starter file where you will write your replacement delticket() function for PA7.
8. **hashdb.h** is a starter file for the function prototypes for the database functions.
9. **insticket.c** is a starter file where you will write your replacement insertticket() function for PA7.
10. **largest.c** is a starter file where you will write your replacement largest() function for PA6.
11. **lbpa6.a** is a starter file that contains solution code that you will write replacements for as well as some printing and debug functions that are used that are not supplied in source form.
12. **loaddb.c** is a starter file that is supplied in source format that has the routines for loading the fines dataset and the summons dataset into memory
13. **loaddb.h** is a starter file that contains the function prototypes for the functions in loaddb.c

14. **parking.h** is a starter file that contains all the struct specifications, program constants, and global variable declarations.
15. **SELVERS.h** is a starter file where you specify if you replacement functions are used (described in a previous section)
16. **subs.c** is a starter file with three helper functions in source form, hash(), strtosumid() and strtodate()
17. **sumlookup.c** is a starter file where you will write your replacement sumlookup() function for PA6.
18. **vehlookup.c** is a starter file where you will write your replacement vehiclelookup() function for PA6.
19. **runtest** is a shell program for testing your program.
20. **exp** is a directory where the expected files for the output checking (both standard out and standard error) are located. The starter files include the expected output files for both standard output and standard error.
21. **in** is a directory where the test input files are located.
22. **out** is a directory where your program output (omit) is put by the bash program runtest.

Step 2: Set the execute bits on the shell program runtest so it can be executed directly.

```
cs30fa22@pi-cluster-153:~/PA6 % chmod 0755 runtest
```

Step 3: Set the execute bits on the shell programming the in directory so they can be executed directly.

```
cs30fa22@pi-cluster-153:~/PA6 % chmod 0755 in/cmd*
```

Step 4: Compile the program and run the demo.

```
cs30fa22@pi-cluster-153:~/PA6 % make
```

```
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o main.o main.c
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o loaddb.o loaddb.c
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o commands.o commands.c
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o insticket.o insticket.c
insticket.c:13:1: note: '#pragma message: TODO - USING THE SOLUTION insticket.c NOT MY CODE'
13 | TODO(USING THE SOLUTION insticket.c NOT MY CODE)
    | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o delticket.o delticket.c
delticket.c:13:1: note: '#pragma message: TODO - USING THE SOLUTION delticket.c NOT MY CODE'
13 | TODO(USING THE SOLUTION delticket.c NOT MY CODE)
    | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o vehlookup.o vehlookup.c
vehlookup.c:13:1: note: '#pragma message: TODO - USING THE SOLUTION vehlookup.c NOT MY CODE'
13 | TODO(USING THE SOLUTION vehlookup.c NOT MY CODE)
    | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o freetickets.o freetickets.c
freetickets.c:13:1: note: '#pragma message: TODO - USING THE SOLUTION freetickets.c NOT MY CODE'
13 | TODO(USING THE SOLUTION freetickets.c NOT MY CODE)
    | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o subs.o subs.c
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o largest.o largest.c
largest.c:12:1: note: '#pragma message: TODO - USING THE SOLUTION largest.c NOT MY CODE'
12 | TODO(USING THE SOLUTION largest.c NOT MY CODE)
```



```

| ^^^~
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o sumlookup.o sumlookup.c
sumlookup.c:13:1: note: '#pragma message: TODO - USING THE SOLUTION sumlookup.c NOT MY CODE'
13 | TODO(USING THE SOLUTION sumlookup.c NOT MY CODE)
| ^^^~
gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h main.o loaddb.o commands.o insticket.o
delticket.o vehlookup.o freetickets.o subs.o largest.o sumlookup.o -L ./ libpa6.a -o parking

```

It is important to observe the lines during compile that look like this:

```

gcc -I. -ggdb -Wall -Wextra -Werror -include SELVERS.h -c -o insticket.o insticket.c
insticket.c:13:1: note: '#pragma message: TODO - USING THE SOLUTION insticket.c NOT MY CODE'
13 | TODO(USING THE SOLUTION insticket.c NOT MY CODE)
| ^^^~

```

This code is emitted by the compiler and is not an error, it is a notice. It is saying that when it is compiling insticket.c it is NOT using your replacement version, but the version supplied in the starter code (in libpa6.a). If you examine SEVERS.h you will see that the line below is commented out.

```
//#define MYINSTICKET // when defined will use your insticket.c
```

When one of the “selection lines” in SELVERS.h is commented out, you will get a warning from the compiler stating that it is compiling in the solution version and NOT your replacement. If you wanted to use your replacement version for insticket.c you would edit SLEVERS.h line for that file to look like without the // in front of the #define.

```
#define MYINSTICKET // when defined will use your insticket.c
```

Testing the Database in interactive mode

Step 5: Run the database program with a small demo dataset. Notice how the database is being run.

```
cs30fa22@pi-cluster-153:~/PA6 % make demodb
```

```
./parking -t3 -f in/Fines.csv -d in/Tiny.csv
```

```
***** Command Summary *****
```

Debug commands:

Q	Quit exit and free up memory
V	Verify database
D	Dump (print) entire database
C index	Print the vehicles on a hash chain[index]
E	Erase the vehicle database (delete database contents)
R	Reload the vehicle database (load database contents)

Query commands:

L	Print largest fine and largest tickets count
F PLATE STATE	Print the tickets for a vehicle with id PLATE STATE
S SUMMONS	Print vehicle information with the SUMMONS
P PLATE STATE SUMMONS	Pay SUMMONS number for vehicle with id PLATE STATE

I SUMMONS PLATE STATE DATE CODE# Insert a SUMMONS (date format: mm/dd/yyyy)
Input command:

What you see above is the program being run with the line
`./parking -t3 -f in/Fines.csv -d in/Tiny.csv`

The -t3 changes the hash table to be only 3 in size. A smaller hash table size causes a lot of vehicles to end up on the same hash chain. This increases the average length of each hash chain significantly. A longer hash chain helps when testing insertion and deletion routines as your code has to manage longer chains

The datasets are loaded and the command interpreter is waiting for a command to be typed on the keyboard. In the section below we will perform the following operations (in order):

1. dump the database with a D command
2. pay a summons with the P command
3. run a built-in verify program (v command) that checks that internal database structures are correct (this command cannot detect all errors so be warned)
4. print out all the vehicles on the hash chain where the vehicle that we paid the ticket was located.

Input command: d

Chain 0:

```
Plate: HLC3177, State: NY, Tickets: 1, Total: $65
Ticket 0001 Summons #: 1133401569 Date: 07/02/2018 Fine Code: 21 Description: $ 65 NO PARKING-STREET CLEANING
Plate: GER9006, State: NY, Tickets: 1, Total: $95
Ticket 0001 Summons #: 1131610520 Date: 07/02/2018 Fine Code: 17 Description: $ 95 NO STANDING-EXC. AUTH. VEHICLE
Plate: GTR7949, State: NY, Tickets: 1, Total: $65
Ticket 0001 Summons #: 1130964875 Date: 06/08/2018 Fine Code: 24 Description: $ 65 NO PARKING-EXC. AUTH. VEHICLE
Plate: HXM7361, State: NY, Tickets: 1, Total: $115
Ticket 0001 Summons #: 1121274900 Date: 06/28/2018 Fine Code: 46 Description: $115 DOUBLE PARKING
```

Chain 1:

```
Plate: HZJ8359, State: NY, Tickets: 1, Total: $65
Ticket 0001 Summons #: 1133401636 Date: 07/02/2018 Fine Code: 21 Description: $ 65 NO PARKING-STREET CLEANING
Plate: HDG7076, State: NY, Tickets: 1, Total: $95
Ticket 0001 Summons #: 1131599342 Date: 06/29/2018 Fine Code: 17 Description: $ 95 NO STANDING-EXC. AUTH. VEHICLE
```

Chain 2:

```
Plate: HPC9135, State: NY, Tickets: 1, Total: $65
Ticket 0001 Summons #: 1133401594 Date: 07/02/2019 Fine Code: 21 Description: $ 65 NO PARKING-STREET CLEANING
Plate: HZN6473, State: NY, Tickets: 1, Total: $65
Ticket 0001 Summons #: 1133401570 Date: 07/02/2018 Fine Code: 21 Description: $ 65 NO PARKING-STREET CLEANING
Plate: GLS6001, State: NY, Tickets: 3, Total: $345
Ticket 0001 Summons #: 1105232165 Date: 07/03/2018 Fine Code: 14 Description: $115 NO STANDING-DAY/TIME LIMITS
Ticket 0002 Summons #: 1105232166 Date: 07/04/2018 Fine Code: 14 Description: $115 NO STANDING-DAY/TIME LIMITS
Ticket 0003 Summons #: 1105232167 Date: 07/05/2018 Fine Code: 14 Description: $115 NO STANDING-DAY/TIME LIMITS
```

Input command: P HDG7076 NY 1131599342

Plate: HDG7076, State NY, Summons 1131599342 paid

Input command: V

Total vehicles in database: 8
Total tickets in database: 10
Longest hash chain in database index number: 0 Length: 4
Shortest hash chain in database index number: 1 Length: 1
Total of all unpaid fines in the database: \$880

Input command: c 1

Chain 1:

Plate: HZJ8359, State: NY, Tickets: 1, Total: \$65

Ticket 0001 Summons #: 1133401636 Date: 07/02/2018 Fine Code: 21 Description: \$ 65 NO PARKING-STREET CLEANING
index: 1 vehicles: 1 tickets: 1

Next we list the largest total fines and largest number of tickets

Input command: L

Most tickets Plate: GLS6001, State: NY tickets: 3, total fine: \$345

Largest fine Plate: GLS6001, State: NY tickets: 3, total fine: \$345

Now we erase the entire database and reload it twice (the second time will show it rejecting all the entries as they are already in the database).

Input command: e

Total tickets freed: 10

Input command: r

Total tickets loaded: 11

Input command: r

```
./parking: duplicate summons 1105232165
./parking: ticket file record 2 summons:1105232165 insertticket failed
./parking: duplicate summons 1105232166
./parking: ticket file record 3 summons:1105232166 insertticket failed
./parking: duplicate summons 1105232167
./parking: ticket file record 4 summons:1105232167 insertticket failed
./parking: duplicate summons 1121274900
./parking: ticket file record 5 summons:1121274900 insertticket failed
./parking: duplicate summons 1130964875
./parking: ticket file record 6 summons:1130964875 insertticket failed
./parking: duplicate summons 1131599342
./parking: ticket file record 7 summons:1131599342 insertticket failed
./parking: duplicate summons 1131610520
./parking: ticket file record 8 summons:1131610520 insertticket failed
./parking: duplicate summons 1133401569
./parking: ticket file record 9 summons:1133401569 insertticket failed
./parking: duplicate summons 1133401570
./parking: ticket file record 10 summons:1133401570 insertticket failed
./parking: duplicate summons 1133401594
./parking: ticket file record 11 summons:1133401594 insertticket failed
./parking: duplicate summons 1133401636
./parking: ticket file record 12 summons:1133401636 insertticket failed
Total tickets loaded: 0
```

Next we look up a vehicle in the database by plate and state, followed by a lookup for a vehicle with a specific summons id.

Input command: f GLS6001 NY

Plate: GLS6001, State: NY, Tickets: 3, Total: \$345

Ticket 0001 Summons #: 1105232165 Date: 07/03/2018 Fine Code: 14 Description: \$115 NO
STANDING-DAY/TIME LIMITS

Ticket 0002 Summons #: 1105232166 Date: 07/04/2018 Fine Code: 14 Description: \$115 NO
STANDING-DAY/TIME LIMITS

Ticket 0003 Summons #: 1105232167 Date: 07/05/2018 Fine Code: 14 Description: \$115 NO
STANDING-DAY/TIME LIMITS

Input command: S 1105232166

Plate: GLS6001, State: NY, Tickets: 3, Total: \$345

Ticket 0001 Summons #: 1105232165 Date: 07/03/2018 Fine Code: 14 Description: \$115 NO
STANDING-DAY/TIME LIMITS

Ticket 0002 Summons #: 1105232166 Date: 07/04/2018 Fine Code: 14 Description: \$115 NO
STANDING-DAY/TIME LIMITS

Ticket 0003 Summons #: 1105232167 Date: 07/05/2018 Fine Code: 14 Description: \$115 NO
STANDING-DAY/TIME LIMITS

Input command:

Last we insert a new summons in the database, print it out and exit.

Input command: I 7777 UCSD CA 10/29/2022 33

Plate: UCSD, State CA, Summons 7777 inserted

Input command: S 7777

Plate: UCSD, State: CA, Tickets: 1, Total: \$65

Ticket 0001 Summons #: 7777 Date: 10/29/2022 Fine Code: 33 Description: \$ 65 FEEDING METER

Input command: s 777

Summons: 777

Not found!

Input command: q

Total tickets freed: 12

Writing the replacement routines

Step 6:

1. Write **ONLY** the following replacement functions for PA6 in any order you like.

freetickets() in file: freetickets.c

largest() in file: largest.c

sumlookup() in file: sumlookup.c

vehiclelookup() in file: vehlookup.c

2. These are the only files that you should modify!

3. You may add helper functions if you like, but they need to be located in the same file as the function that uses them as we are testing the files individually.

4. Make sure that you remove the comment for the correct function in SELVERS.h

Suggested approach: Develop and test each function by itself with the solution versions for all the other functions. When you get all four replacement functions fully working independently, then test them all together.

Testing the database with the test harness

Part 7: test your replacement functions.

Each test in the alltest list of tests is progressively harder. The dataset size in testA is small and gets progressively larger with each test group. There is a specific test for handling duplicate summons in testA and the testsV1 and testV2 are valgrind tests on small and large datasets.

In PA6 the test harness bash shell program runtest is unchanged from PA5. In the makefile there are the following test groups. The Makefile target runs all these test groups.

```
LISTA    = 1 2 DUP
LISTB    = 3 4
LISTC    = 5
LISTD    = 6
LISTV1   = V1
LISTV2   = V2
LISTP    = P
```

```
%make alltest # runs all tests (except testP - see the appendix about testP)
```

```
%make testA   # runs test group LISTA
```

```
%make testB   # runs test group LISTB
```

And so on.

You can run individual test(s) by calling runtest directly with one or more test id's as arguments

```
% ./runtest 1 3 6 V1 # runs test 1, 3, 6, V1 in order
```

The structure of the test harness is exactly the same as PA5. The actual command run for a test has the same id value as the test but starts with cmd. So looking in the directory in, for test1 the command that is run is in the file cmd1, for testV2 the command that is run is in cmdV1 and so on.

Coding Requirements

1. You will write your code only in C and it must run in a Linux environment on the pi-cluster.

2. In the code you write you cannot use `calloc()`. The only use of `calloc()` in the program is the ones in the starter file `main.c`
3. Do not use recursive solutions in your code.
4. **You cannot use any downloaded software** (other than what is in the github repository **for this PA**). **If it is not already installed on the pi-cluster by ITS, you cannot use it.**
5. Make sure you pass all the tests before submitting to gradescope.

Turning in Files For Your Grade

Before submitting your program to gradescope, make sure at a minimum that your program passes the supplied tests in the test harness.

Submit to gradescope under the **assignment titled PA6**

ONLY submit the following files (total of four (4) files):

```
freetickets.c
largest.c
sumlookup.c
vehlookup.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.

Appendix 1 - Optional — What is this testP?

testP is a really simplistic performance metric on the database. Now the code is not being compiled with the optimizers on, but this still gives you an indication of how fast your code runs. The supplied solution code has a lot of additional tests to try to help you debug your code, so your routines should be as least as fast. The timings are written to stderr so they report as errors by the test harness even though they are not errors!

Here is a sample run:

```
cs30fa22@pi-cluster-153:~/PA6 % make testP
```

```
***** starting testP *****
```

```
./runtest P
```

```
----- Starting test number P -----
```

```
Running in/cmdP < in/testP > out/outP 2> out/errP
```

```
cmdP is: time ./parking -s -t409 -f in/Fines.csv -d in/Large.csv
```

```

./parking returned EXIT_SUCCESS
    Comparing exp/outP to out/outP
**** Standard out    on test number P passed ****
    Comparing exp/errP to out/errP

**** Standard error on test number P failed ****

*** exp/errP 2022-10-23 17:40:41.397440848 -0700
--- out/errP 2022-10-29 01:37:18.990930463 -0700
*****
*** 0 ****
--- 1,4 ----
+
+ real0m37.104s
+ user0m34.310s
+ sys 0m2.711s
----- Ending    test number P -----

***** All Done *****

```

Ignore the line [Standard error on test number P failed](#) that is normal for this test (it is not failing). The important numbers are the user and sys lines (smaller is better), real time is not a good performance guide. These numbers were run on a PI4B with a much faster filesystem than on the pi-cluster, so the sys numbers will be a lot better than you will get. A number to look at are user values only.

Appendix 2 – Testing everything: make alltest

Here is the output from run make alltest

```

cs30fa22@pi-cluster-153:~/PA6 % make alltest

***** starting testA *****
./runtest 1 2 DUP

----- Starting test number 1 -----
Running in/cmd1 < in/test1 > out/out1 2> out/err1
cmd1 is: ./parking -s -t3 -f in/Fines.csv -d in/Tiny.csv
./parking returned EXIT_SUCCESS
    Comparing exp/out1 to out/out1
**** Standard out    on test number 1 passed ****
    Comparing exp/err1 to out/err1
**** Standard error on test number 1 passed ****
----- Ending    test number 1 -----

----- Starting test number 2 -----
Running in/cmd2 < in/test2 > out/out2 2> out/err2

```

```

cmd2 is: ./parking -s -t3 -f in/Fines.csv -d in/Tiny.csv
./parking returned EXIT_SUCCESS
    Comparing exp/out2 to out/out2
**** Standard out    on test number 2 passed ****
    Comparing exp/err2 to out/err2
**** Standard error on test number 2 passed ****
----- Ending    test number 2 -----

----- Starting test number DUP -----
Running in/cmdDUP < in/testDUP > out/outDUP 2> out/errDUP
cmdDUP is: ./parking -s -t3 -f in/Fines.csv -d in/Dups.csv
./parking returned EXIT_SUCCESS
    Comparing exp/outDUP to out/outDUP
**** Standard out    on test number DUP passed ****
    Comparing exp/errDUP to out/errDUP
**** Standard error on test number DUP passed ****
----- Ending    test number DUP -----

***** All Done *****

***** starting testV1 *****
./runtest V1

----- Starting test number V1 -----
Running in/cmdV1 < in/testV1 > out/outV1 2> out/errV1
cmdV1 is: valgrind -q --leak-check=full --leak-resolution=med -s ./parking -s -t3 -f in/Fines.csv
-d in/Tiny.csv 2>&1 1>/dev/null | cut -f2- -d' ' 1>&2 2>/dev/null
./parking returned EXIT_SUCCESS
    Comparing exp/outV1 to out/outV1
**** Standard out    on test number V1 passed ****
    Comparing exp/errV1 to out/errV1
**** Standard error on test number V1 passed ****
----- Ending    test number V1 -----

***** All Done *****

***** starting testB *****
./runtest 3 4

----- Starting test number 3 -----
Running in/cmd3 < in/test3 > out/out3 2> out/err3
cmd3 is: ./parking -s -t3 -f in/Fines.csv -d in/Small.csv
./parking returned EXIT_SUCCESS
    Comparing exp/out3 to out/out3
**** Standard out    on test number 3 passed ****
    Comparing exp/err3 to out/err3
**** Standard error on test number 3 passed ****
----- Ending    test number 3 -----

```



```
----- Starting test number 4 -----
Running in/cmd4 < in/test4 > out/out4 2> out/err4
cmd4 is: ./parking -s -t3 -f in/Fines.csv -d in/Small.csv
./parking returned EXIT_SUCCESS
    Comparing exp/out4 to out/out4
**** Standard out    on test number 4 passed ****
    Comparing exp/err4 to out/err4
**** Standard error on test number 4 passed ****
----- Ending    test number 4 -----
```

***** All Done *****

```
***** starting testC *****
./runtest 5
```

```
----- Starting test number 5 -----
Running in/cmd5 < in/test5 > out/out5 2> out/err5
cmd5 is: ./parking -s -t409 -f in/Fines.csv -d in/Medium.csv
./parking returned EXIT_SUCCESS
    Comparing exp/out5 to out/out5
**** Standard out    on test number 5 passed ****
    Comparing exp/err5 to out/err5
**** Standard error on test number 5 passed ****
----- Ending    test number 5 -----
```

***** All Done *****

```
***** starting testD *****
./runtest 6
```

```
----- Starting test number 6 -----
Running in/cmd6 < in/test6 > out/out6 2> out/err6
cmd6 is: ./parking -s -f in/Fines.csv -d in/Large.csv
./parking returned EXIT_SUCCESS
    Comparing exp/out6 to out/out6
**** Standard out    on test number 6 passed ****
    Comparing exp/err6 to out/err6
**** Standard error on test number 6 passed ****
----- Ending    test number 6 -----
```

***** All Done *****

```
***** starting testV2 *****
./runtest V2
```

```
----- Starting test number V2 -----
Running in/cmdV2 < in/testV2 > out/outV2 2> out/errV2
```

```
cmdV2 is: valgrind -q --leak-check=full --leak-resolution=med -s ./parking -s -t409 -f
in/Fines.csv -d in/Medium.csv 2>&1 1>/dev/null | cut -f2- -d' ' 1>&2 2>/dev/null
./parking returned EXIT_SUCCESS
    Comparing exp/outV2 to out/outV2
**** Standard out    on test number V2 passed ****
    Comparing exp/errV2 to out/errV2
**** Standard error on test number V2 passed ****
----- Ending    test number V2 -----

***** All Done *****
```

Copyright 2022 Keith Muller. All rights reserved.