CSE 222 Programming Assignment #3

January 30, 2020

1 Introduction

In this assignment, you're going to write a program that can hash a database of license plates and names. After reading in a file containing plate/name pairs, your program will let the user enter a license plate, and it will display the corresponding name. The user can also enter commands instead of a license plate, in order to examine the internal data structure of your hash table.

2 Behavior

You should run /tmp/plate (you can use /tmp/database.txt as a sample database) to observe the following behaviors. Your program's output and other behavior should match that sample program as closely as possible. Minor variations ("PLATE NOT FOUND" vs "Plate not found" for example) are fine, but anything other than minor cosmetic differences will cost points.

2.1 Start-up

Your executable program must be named "plate" and can be run in one of two ways:

- plate database_name

which will read the plate information from database_name and save it in a hash table of size 100; or

- plate database_name hash_table_size

which lets you specify the size of the hash table (must be an integer $\geq$1)

Your program should report an ERROR if it is not run in one of these two ways (or if hash_table_size is not an integer $\geq$1) and then return(1).

2.2 Reading the database

Once the program begins, it should open the database for reading and ingest data, one line at a time. The data looks like this:

```
PTN-673 RICHELLE MUELLER
QFJ-481 LUDIVINA WOODWARD
RRW-885 VITO HUBBARD
```

and so on. Each line contains a license plate ID (no spaces, but make no other assumptions about the format), followed by a space, a first name, a space, a last name, and a newline (\n). If the database cannot be read, the program should report an ERROR and return(1).

2.3 User Interaction

The program then prompts the user to enter a license plate or a command. If a plate is entered, it should search the hashtable and either print the name associated with the plate, or print the message "Plate not found." If the user enters a command, it should execute the command. In any case, the program will repeatedly prompt the user and process their commands.

2.3.1 Legal Commands

There are thee commands the user can enter:

***LOAD**

This will display the number of plates stored in each cell of the hash table. This is the length of the corresponding linked list (not counting the sentinel node)

***DUMP**

This will display the *contents* of each cell of the hash table

***DUMP n**

(where **n** is an integer) will display the contents of cell **n** of the hash table (if n is not a legal, in-range integer, print an error message)

2.3.2 Output Details

Normal plate information (in response to a plate look-up) should be printed as follows:

```
Enter command or plate: BLD-947
First name: ELIZ
Last name: FRANKLIN
Enter the plate:
```

The response to the *LOAD command should be presented as follows:

```
Enter the plate: *LOAD
Entry 0: length=10
Entry 1: length=7
Entry 2: length=12
Entry 3: length=3
```

```
Entry 4: length=10
Entry 5: length=10
Enter the plate:
```

The *DUMP command should show the contents of a cell or cells in the following format:

```
Enter the plate: *DUMP 3
Contents of cell 3
Plate: <AOG-380>  Name: YATES, DORETTA
Plate: <UVJ-347>  Name: FINCH, SHELTON
Plate: <MOQ-693>  Name: MYERS, INDIRA
---------------------------------
Enter the plate: *DUMP 5
Contents of cell 5
---------------------------------
Enter the plate:
```

2.4 Exiting

The user exits the program by entering EOF (CTRL-D at the beginning of a line). This is the only way your program should exit: it should not exit in response to errors, and it should not seg fault. Before exiting, the program should free all memory that it allocated.

3 Implementation

You will implement a pair of data structures:

- a linked list, comprised of the following nodes:

```
struct node{
  char *plate;
  char *first;
  char *last;
  struct node *next;
};
```

and including a sentinel node; and

- a hash table comprised of an array of struct node* (pointers to sentinel nodes)

If you use

```
typedef struct node* hashEntry; // each hash entry is a sentinel node,
                                // pointing to a struct node
```

then you would declare your hash table as an array of hashEntry's, i.e.

```
hashEntry* hashTable
```

Note that you cannot assume a pre-define hash table size, i.e. you cannot declare hashEntry table[100]. Also note that you cannot assume pre-defined sizes of the struct node's strings: plate, first and last could be 2 characters, 20 characters, or anything else. Therefore, you will be using malloc to allocate memory for the hash table, the nodes of the linked list, and the contents of those nodes.

Remember to free all malloc'd memory before exiting.

4 DETAILS

4.1 Hash Function

Use the following hash function:

$$hash(plate) = \left(\sum_{i=0}^{strlen(plate)-1} (i+5) * plate[i] \right) \% hashtablesize$$

4.2 Data Structures

The structure for your linked list nodes is described above. Your hash table should be an array of pointers to sentinel nodes (hashtablesize of them), each initialized to point to a different sentinel node of an empty linked list.

4.3 Required Functions

Implement the following **hash** functions, for use from your main program:

```
hashEntry *hashInit(int hashsize); // construct the initial (empty) hash table

void hashAdd(hashEntry *hashTable,
             char *plate,
             char *first,
             char *last); // make a new entry in the hash table for this plate/name

int hashFind(hashEntry *hashTable,
             char *plate,
             char *first,
             char *last); // search for the given plate in the hash table
                          // If found, load the matching first and last name
                          //    into first and last, and return 1
                          // If not found, return 0

void hashLoad(hashEntry *hashTable); // print the load in each cell
```

```
void hashDump(hashEntry *hashTable, int cellNum); // print contents (list) of cell cellNum

void hashFree(hashEntry *hashTable); // free all memory associated with this hash
```

Implement the following **list** functions, for use from your hash code:

```
struct node *listInit(); // create empty list (just a sentinel)

void listAdd(struct node *sent,
             char *plate,
             char *firstname,
             char *lastname); // add data to the *beginning* of the list

int listFind(struct node *sent,
             char *plate,
             char *firstname,
             char *lastname); // search for the given plate in the linked list
                              // If found, load the matching first and last name
                              //    into firstname and lastname, and return 1
                              // If not found, return 0

int listLen(struct node *sent); // count # of (data) nodes and return as function value

void listPrint(struct node *sent); // print the list's contents (as described above)

void listFree(struct node *sent); // release all memory associated with the list
                                  // (including the sentinel itself)
```

5 SUGGESTIONS

Start early (like, today!) but don't start coding right away. make a plan for how you will structure this. Look over the above function prototypes for the list and the hash, and think about how they will fir together. You already have linked list code from PA2, which can be modified slightly to work here.

The hash is the main new piece of this assignment. Write a hash function. You can test it by running the sample code on the server and seeing where different plates hashed to, and comparing to the output of your own hash function. Then think about initializing the hash, and then adding entries to it. You can print the contents of the hash for debugging purposes, and compare to the *DUMP output from the sample code.

Adding to the hash is really just using the list's add function. The only hash-related task is finding *which* list to add to. Once you understand that, you should be able to search the hash in the same way: determine which list

you're interested in, and then use the list's search function. Same for printing the contents of a cell in the hash.

hashLoad and hashDump are different versions of list printing: hashLoad requires you to determine the length of each list; hashDump is printing the contents of one or more hash cells.

The next piece is freeing the memory, which will take some thinking about but is really freeing each list (which you're doing in PA2) and then freeing the array you initially malloc'd for the hash table.

The final piece is the main program. This is mainly a bunch of calls to hash functions, but you need to parse user input (as a plate search request or as a command), and also deal with the different command line options, while doing error checking along the way. There's nothing to this that you didn't do in CSE 224, but there are a lot of details. As usual, you can start simpler (minimal options, assume no input errors, etc.) and then add complexity as you develop the program further.

Ask me questions if you are getting stuck, or aren't sure how to proceed. Take things in small steps, start with something you know how to do, and build it up into larger pieces. And have fun!

6 SUBMISSION

Submit via GITLab to a repository named "PlatePA3" Make sure I am added as a reporter. Include all source code, ".h" files and your Makefile. **Please do not include .o files or your executable file**.

Before you consider this assignment finished, *I strongly recommend* cloning your GIT repository into a new directory, saying "make" and then trying to run your program. If that doesn't work for you, then it probably will not work for me, which will cost you at least 25 points for late submission. This is basically an acid test for GIT, make, etc.