Assignment 6: Population Lookup

CSE30: Computer Organization and Systems Fall 2021

Instructors: Bryan Chin and Keith Muller

Due: Monday, November 8th, 2021 @ 5:00PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. REMEMBER: Everyone procrastinates but it is important to know that you are procrastinating and still leave yourself enough time to finish. **Start early**. **You MUST run the assignment on the pi cluster. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.**

NOTE: Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on Edstem.

Table of Contents

Learning Goals

- Pointers
- Parsing input files
- Dynamic memory allocation and management
- Implementation of dynamic memory structures (e.g. hash tables, linked lists)
- Use of structs

Assignment Overview

The villagers of Minecraft want to learn about the world outside your computer, and they need your help. They want to know how many people live in your world, and they want to know how many people live in the cities and states of the U.S.

For this assignment, you shall be parsing the population data of cities in the U.S. provided in the form of a CSV (comma separated value) file. Your job is to build a system that stores this population data and is able to query it by city or state name in near constant time. You shall do this by implementing a hash table-based in-memory database using single linked chains for collision resolution. The database will contain the city names, state names, and population numbers of all the cities in the U.S.

You will load the CSV file into the database and then make a query of either a city name or a state name. If the city or state is not found in the database, your program will respond with a “not found” message. Otherwise, the program will print out the minimum, maximum, and average populations of all cities by that name, or all cities in that state, depending on your initial query. There can be multiple city entries for any one city name (e.g. Marysville,CA and Marysville,OH).
Figure 1 shows a high level representation of a hash table once it has been loaded with data from the CSV file. For a reminder of how hashed chaining works, watch this 5 minute video from Prof. Leo Porter about hash tables. [Video on Google Drive]

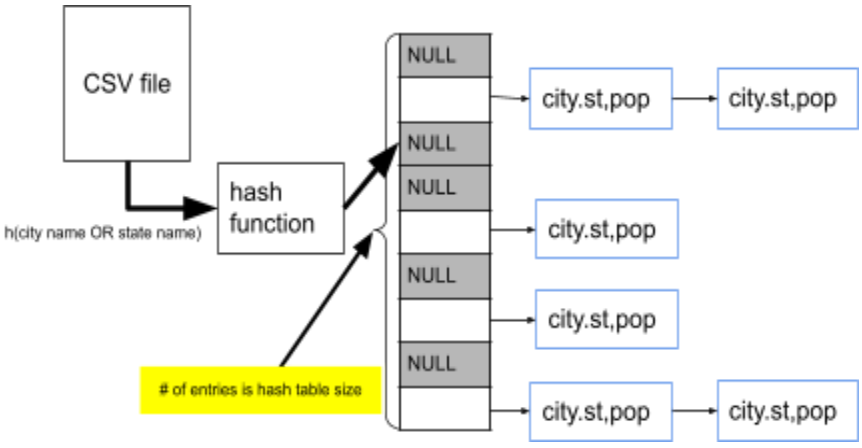

Figure 1: Hash table with chaining

Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**. To access the server, you will need your `cs30fa21cxx` student account, and know how to SSH into the cluster.

Need help or instructions? See Edstem FAQ or watch CSE30 Development Tools Tutorial
(Do NOT wait until Friday to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following GitHub repository.

1. Clone/download the GitHub repository.[1]
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and begin working.

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, we've placed the reference solution binary at:
`/home/linux/ieng6/cs30fa21c/public/bin/poplookup-a6-ref`.
You can directly use `poplookup-a6-ref` as a command.

Makefile: The `Makefile` provided will create a `poplookup` executable from the source files provided. Compile it by typing `make` into the terminal. By default, it will run with warnings turned on. To compile without warnings, run `make WARN=` instead. Run `make clean` to remove files generated by `make`.

[1] If you use your own Github repo, make sure it is private. Your code should never be pushed to a public Github repository. **A public Github repo is an AI violation.**

How the Program Works

The program shall take the city or state name to query, and the filename of the CSV as arguments. It also takes some optional arguments: a flag to indicate printing of hash table metadata, and a hash table size.

Inputs:

- City name OR state name
- Filename of the CSV
- Optional: Print metadata flag
- Optional: Hash table size

Outputs:

- The minimum, maximum, and average population of the query results, OR an error message if the city/state name couldn't be found
- Optional: Hash table metadata

Population Lookup Arguments

Format for calling this executable with arguments (the flags cannot be grouped with each other, e.g. `-itc`):

```
./poplookup [-i] [-t tablesize] <[-c city]/[-s state]> <filename>
```

Argument(s)	Description
`-i` (User-optional flag)	Prints descriptive metadata about the hash table and its linked lists chains before the query.
`-t tablesize` (User-optional flag)	`tablesize` (hash table size) is an integer specifying the size to be used to create the hash table. If `-t` is not specified, the default is the constant `TABLE_SIZE`. If the argument to `-t` is smaller than the constant `MIN_TABLE_SIZE`, an error message is printed with the usage message and the program quits.
`-c city`	`city` is a string specifying the name of the city to query. **Exactly 1 of `-c city` OR `-s state` must be provided, but not both.**
`-s state`	`state` is a string specifying the name of the state to query. **Exactly 1 of `-c city` OR `-s state` must be provided, but not both.**
`filename`	This is the path to the input CSV. It **must** have three columns, in this order: `city,state,population`

	We will **not** give malformed input CSVs to your program. The input path will always be valid and be a properly-formatted CSV.

You **do not** have to write the option parsing: It is done for you in the provided function `parse_opts()`. Do not edit `parse_opts()`!
If the mandatory arguments are not provided, their error messages are printed, followed by the usage message. You can assume that a valid CSV will be included in any test command.

Output Examples

```
$ ./poplookup -c Houston us2021census.csv
Minimum: 143    Maximum: 2304580        Average: 289390
```

```
$ ./poplookup -c "Los Angeles" us2021census.csv
Minimum: 3898747        Maximum: 3898747        Average: 3898747
```

```
$ ./poplookup -c Housty us2021census.csv
Unable to find any cities by the name of Housty.
```

```
$ ./poplookup -c "Lossy Angelossy" us2021census.csv
Unable to find any cities by the name of Lossy Angelossy.
```

```
$ ./poplookup -s TX us2021census.csv
Minimum: 0      Maximum: 2304580        Average: 17250
```

```
$ ./poplookup -s AZ us2021census.csv
Minimum: 296    Maximum: 1608139        Average: 62954
```

```
$ ./poplookup -s testin us2021census.csv
Unable to find any states by the name of testin.
```

```
$ ./poplookup -s "Steve Vill" us2021census.csv
Unable to find any states by the name of Steve Vill.
```

```
$ ./poplookup -s Houston us2021census.csv
Unable to find any states by the name of Houston.
```

```
$ ./poplookup -i -c Boston us2021census.csv
Table size: 1873
Total entries: 21397
Longest chain: 48
Shortest chain: 1
Empty buckets: 1
Minimum: 150    Maximum: 675647 Average: 225668
```

Error Examples

```
$ ./poplookup
./poplookup: filename is required
./poplookup: -c city or -s state is required
Usage: ./poplookup [-i] [-t tablesize] [-c city]/[-s state]
filename
```

```
$ ./poplookup -t 1 -c Houston us2021census.csv
./poplookup: -t value must be equal or larger than 3
Usage: ./poplookup [-i] [-t tablesize] [-c city]/[-s state]
filename
```

```
$ ./poplookup -s TX -c Houston us2021census.csv
./poplookup: Cannot query both a city and a state
Usage: ./poplookup [-i] [-t tablesize] [-c city]/[-s state]
filename
```

Program Implementation

Functions and Behaviors to Implement

When allocating strings, we recommend *against* the use of `strdup()` as that will result in warning messages from the C compiler. `strdup()` is not part of the C11 standard. Instead, you may use a combination of `malloc()` and `strcpy()`.

`node* node_lookup()`

Arguments:

`node* front`	→ The current head of the linked list chain
`char* city`	→ name of city
`char* state`	→ name of state
`int pop`	→ population size

Operation:

- Searches the linked list chain for a node that matches the city name `city`, the state name `state`, and the population size `pop`.
- Returns the pointer to the node with matching data, otherwise, return `NULL`.

`node* add_front()`

Arguments:

`node* front`	→ The current head of the linked list chain
`char* city`	→ name of city
`char* state`	→ name of state
`int pop`	→ population size

Operation:

- Creates a new node with the city, state, and population data.[2]
- If this creation fails, returns `NULL`.
- Otherwise, inserts this new node at the head of the chain.
- Returns the new head of the chain.

`void print_population()`

Arguments:

`node** table`	→ pointer to hash table
`char* str`	→ name of city or state

[2] You will want to implement a deep copy and dynamic memory allocation..

`unsigned long size` → hash table size
`int hash_by_city` → if non-zero, hash data by city name, else hash by state name

Operation:

- Hashes the given string to get an index into the hash table.
- Finds every record that matches the city or state name.
- Uses the data in each record to calculate the minimum, maximum and average population for the query using this format string:

  `Minimum: %d\tMaximum: %d\tAverage: %d\n`

  Note: the average is calculated by dividing two **integers**, i.e. two `int`s.
- If no matches are found, prints, to `stdout`, using the format string

  `Unable to find any %s by the name of %s.\n`

  where the first string is either "cities" or "states", and the second string is the given parameter `str`.

`int load_table()`

Arguments:

`node** table` → pointer to hash table
`unsigned long size` → hash table size
`char *filename` → name of the census data CSV file
`int hash_by_city` → if non-zero, hash data by city name, else hash by state name

Operation:

- Opens the filename file for "r", reading.
  - If it is unable to open the file for reading, print an error message by using this line of code, and return 1:
    `perror("load_table filename open");`
- Reads a line of data from the CSV. This line will be no longer than `LINE_SIZE-1` characters. If it is unable to allocate space for this line buffer, print an error message using this line of code, and return 1:
  `perror("load_table malloc");`
- Parses the line into separate strings for city, state, and population. Population is string of digits 0-9 representing an integer less than 2^{31}.
- Hashes the city name or state name.
- Calls `node_lookup()` to see if the entry is already in the table. An entry is a duplicate if it has the exact same city name, state name, and population.

- If a duplicate is found, `fprintf()` to `stderr` and skips to the next entry. Use this format string:

  `"load_table duplicate entry: %s %s %d\n"`
- Otherwise, if there is no duplicate entry, calls `add_front()` to insert a new record (city, state, population) into the table. If `add_front()` fails (returns `NULL`), print an error message to `stderr` using this format string:
  `"load_table could not add %s\n"`
  where the argument is the line of data we are currently reading.
  The program should then continue to the next line of data.
- Returns 0 if there were no memory errors, 1 otherwise. Duplicate entries are not errors.

Implementation Hint: Parsing an Input Buffer

If you read one line of the file at a time, you will get an array like this.

L	a		J	o	l	l	a	,	C	A	,	1	0	'\n'	'\0'

What you want is this:

L	a		J	o	l	l	a	**'\0'**	C	A	**'\0'**	1	0	**'\0'**	'\0'

Now you can set the pointers to `city` and `state` to their appropriate substrings as we have effectively split the string into 3 strings. Ask yourself, where should `city` point, where should `state` point? Remember, this is not a deep copy!

`void print_info()`

Arguments:

`node** table` → pointer to hash table
`unsigned long size` → hash table size

Operation:

- Walk the hash table chain by chain.
- Output the following:
  - The size of the table (Table size)
  - The total number of nodes in the table (Total entries)
  - The length of longest and shortest non-empty chains (Longest chain, Shortest chain)
  - The number of empty buckets in the table (Empty buckets)
- Prints all this information to `stdout` using the following format strings, in this order:
  `"Table size: %lu\n"`

```
"Total entries: %lu\n"
"Longest chain: %lu\n"
"Shortest chain: %lu\n"
"Empty buckets: %lu\n"
```

`void delete_table()`

Arguments:

`node** table` → pointer to hash table

`unsigned long size` → hash table size

Operation:

- Frees all of the memory allocated while creating linked lists for each entry in the hash table.
- Frees the memory allocated for the hash table itself.

Run Valgrind to check for memory leaks and memory errors. Valgrind is a suite of tools that do memory error analysis, among other things. Use `man valgrind` for more information. When you run `valgrind ./poplookup <args>`, it should report “All heap blocks were freed -- no leaks are possible” and “ERROR SUMMARY: 0 errors from 0 contexts”.

Suggested Debugging/Development Tips

This program is bigger than prior ones. We suggest you break this program into smaller segments.

1. START EARLY!
2. Read the instructions carefully.
3. Check the FAQ often and search Edstem for previous questions, if possible.
4. Outline your program in pseudocode or a flow chart. Tutors may ask to see this!
5. Draw your data structures.
6. Debug the input file parsing first, making sure you can cleanly parse each input line.
7. Implement `add_front()`. You may want to test `add_front()` on its own by building an independent linked list, outside the hash table, using `add_front()`.
8. Once you have this basic functionality, you can then go on to implement the rest of the functionality (`print_info()`, `print_population()`, `node_lookup()`).
9. Implement `delete_table()` and debug memory leak or dangling pointer issues.
10. If you are working on a bug, use GDB. You can set breakpoints at interesting points (e.g. `break add_front()` and step through your code to watch it add a node). You can make better use of tutor lab hours if you have a debug session to show them. If you have a segmentation fault, run GDB before going to a tutor. Otherwise, they may not have time to help you, especially when the queue gets busy.
11. TESTING: **List** all the test cases you can think of (re-read this document).

For example, do you handle the case where a duplicate entry is being added? How would you test this?
How about different table sizes (small, medium, and big)? Does your program produce the same statistics as the reference binary?
Do you handle different databases?

The public autograder will only test some features. DO NOT RELY ON THE AUTOGRADER (many CSE30 students have been burned by this). Test your code using your own test cases!!!

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled “A6: Population Lookup”. You will submit the following files:

   ```
   poplookup.c
   README.md
   ```

   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 of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
   a. Check that all required files were submitted.
   b. Check that `poplookup.c` compiles without warnings.
   c. Runs some tests on the resulting `poplookup` executable.

Grading Breakdown [2 + 48 pts]

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests.***

To encourage you to write your own tests, we are not providing any public tests that have not already been detailed in this writeup.

The assignment will be graded out of 50 points and will be allocated as follows:

- **Midpoint writeup:** 2 points. **This part of the assignment is due earlier than the full assignment, on Friday 11/5 at 5pm.** Submit a 1-2 page (no more than 2 pages) document describing your algorithm. This can be pseudocode, a flow chart, or some other easily understandable description of how your program will work. You should have some picture or description of how your data structures work, similar to the diagram above but with more detail on the actual data nodes. This is a planning document and does not need to reflect your final implementation, although you are encouraged to keep it up to date. Submit this writeup to the Gradescope assignment titled "A6: Midpoint".
- **No warnings when compiling**.
- **No errors from Valgrind at any point**.
- **The public tests as detailed in this writeup**.
- A variety of hidden tests that will test many different cases.
- In progress… Check back later!

NOTE: The end to end tests expect an EXACT output match with the reference binary. There will be NO partial credit for any differences in the output. Test your code - DO NOT RELY ON THE AUTOGRADER FOR PROGRAM VALIDATION (READ THIS SENTENCE AGAIN).

Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster without warnings. **Any assignment that does not compile will receive 0 credit.**

Checking For Exact Output Match

A common technique is to redirect the outputs to files and compare against the reference solution[3]:

```
./your-program args > output; our-reference args > ref

diff output ref
```

If the second command outputs nothing, there is no difference. Otherwise, it will output lines that differ with a < or > in front to tell which file the line came from.

END OF INSTRUCTIONS, PLEASE RE-READ!

[3] You might want to check out `vimdiff` on the pi-cluster (https://www.tutorialspoint.com/vim/vim_diff.htm).

Writeup Version

1.0 Initial release.
1.0.1 Clarifications on what counts as an error in `load_table`.
1.0.2 Fixed sentence ordering in `load_table` error message from when `add_front` fails.
1.0.3 Fixed example output for "`./poplookup -i -c Boston us2021census.csv`".