

Project #05: Chicago Crime Lookup

Complete By: Monday, April 3rd @ 11:59pm

Assignment: C program to perform Chicago Crime lookup

Policy: Individual work only, late work **is** accepted (see “Policy” section on last page for more details)

Submission: electronic to Blackboard

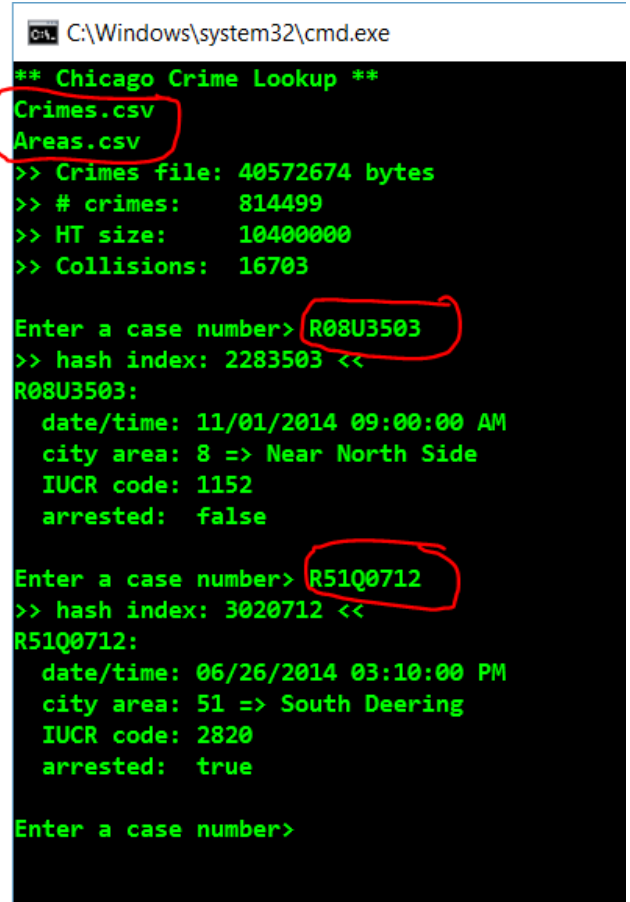
Assignment

The city of Chicago, like many major cities, makes crime data available for analysis. This can be found on Chicago’s data portal: <https://data.cityofchicago.org/>. In this assignment we’re going to use hashing to quickly lookup crimes by their case number. You will be inputting hundreds of thousands (if not millions) of crimes in CSV format, building a hash table, and then allowing the user to lookup a crime.

Here’s a screenshot. This is processing the smaller “Crimes.csv” file, which contains crime data for the city of Chicago during the years 2013..2015. There is a total of 814,499 reported crimes; in response my program created a hash table of size 10,400,000, and produced 16,703 collisions during the creation of the table. Once the hash table is constructed, the program prompts the user for a case number, and then uses the hash table to lookup the case # and output crime information if found. This is repeated until the user enters the empty string, in which case the program stops.

There is a secondary input file “Areas.csv” that maps the numeric location of the crime to the name of the community area. The numeric values fall in the range 0..77, inclusive. As shown to the right, 8 denotes the “Near North Side” area, and 51 denotes “South Deering”.

The format of your output should match the screenshot on the right; user input is circled in red. Notice the internal hash index is output so we can see how your hash function is working.



```
C:\Windows\system32\cmd.exe
** Chicago Crime Lookup **
Crimes.csv
Areas.csv
>> Crimes file: 40572674 bytes
>> # crimes: 814499
>> HT size: 10400000
>> Collisions: 16703

Enter a case number> R08U3503
>> hash index: 2283503 <<
R08U3503:
    date/time: 11/01/2014 09:00:00 AM
    city area: 8 => Near North Side
    IUCR code: 1152
    arrested: false

Enter a case number> R51Q0712
>> hash index: 3020712 <<
R51Q0712:
    date/time: 06/26/2014 03:10:00 PM
    city area: 51 => South Deering
    IUCR code: 2820
    arrested: true

Enter a case number>
```

Input Files

The input file “Crimes.csv” contains info about each reported crime in Chicago, in no particular order. The file format is as follows:

```
Case Number,IUCR,DateTime,Beat,District,Ward,Arrested
R43P6372,0281,01/01/2015 12:00:00 AM,334,3,7,F
R27H7172,0820,11/24/2015 05:30:00 PM,1124,11,28,F
R06Q6148,0560,05/19/2015 01:12:00 AM,1933,19,44,T
.
.
.
```

The 1st line contains column headers, and should be ignored. The data starts on line 2, and each data line contains 7 values:

- Case number: 8-character string, in the format RXXYZZZZ, where XX is the area of the city where the crime occurred, Y is a single letter ‘A’-‘Z’, and ZZZZ is a 4-digit number.
- IUCR code (Illinois Uniform Crime Reporting) — this is **not** an integer, but a 4-character string
- Date and time: character string
- Beat: police beat, integer
- District: voting district, integer
- Ward: city ward, integer
- Was an arrest made? A single character, either T for true or F for false

For this assignment, you can ignore the beat, district, and ward. Note that this is just one possible input file, we may use a different file when grading. The # of reviews in the file is unknown, do not assume a maximum size. A larger version, named “Crimes-2.csv”, is also provided to test the validity & quality of your hashing approach.

The second input file, “Areas.csv”, associates a name with each numeric area of Chicago. The file has the following format:

```
Number,Community
0,Unknown
1,Rogers Park
2,West Ridge
.
.
.
76,Ohare
77,Edgewater
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each line contains 2 values: the numeric area, and the name of that area. You may assume there are 78 areas (0..77), and that the file is in ascending order by numeric area. The contents of this file will not change (other than to

potentially modify the name of a given area / correct a spelling mistake).

These input files are available on the course web page under “Projects”, then “[project05-files](#)”. The files are being made available for each platform (Linux, Mac, and PC) so you may work on your platform of choice. Even though the files end in the file extension “.csv”, these are text files that can be opened in a text editor (e.g. Notepad) or a spreadsheet program (e.g. Excel) — note that Excel may not be able to open the larger “Crimes-2.csv” file. **Note that** the best way to download the files is **not** to click on them directly in dropbox, but instead use the “download” button in the upper-right corner of the dropbox web page.

Requirements

The main focus of this assignment is hashing. For this reason, the main requirement is pretty simple:

- Write your program in C, do not use C++. Your program will be compiled using gcc.
- Use a hash table to store and lookup crime data by case number. Collisions are bound to occur, you may use chaining or linear probing to resolve. The goal however is to use smart design to reduce the # of collisions, without wasting huge amounts of space.
- Dynamically adapt your hashing approach to the size of the input. First, obtain the size of the crimes file using the following function:

```
long getFileSizeInBytes(char *filename)
{
    FILE *file = fopen(filename, "r");
    if (file == NULL)
        return -1;

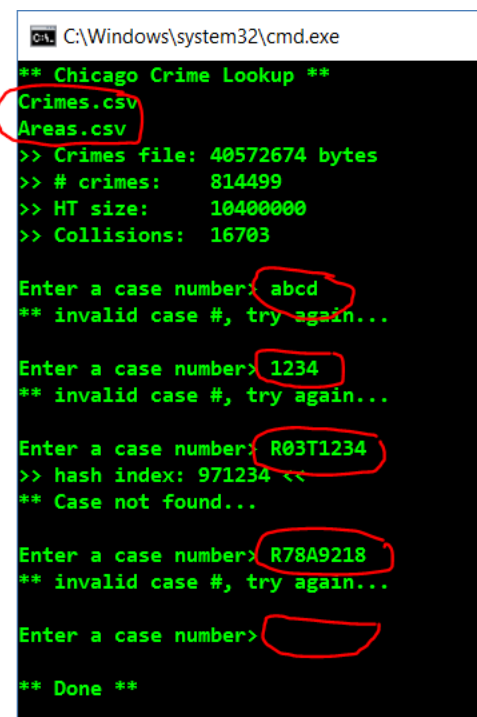
    fseek(file, 0, SEEK_END);
    long size = ftell(file);

    fclose(file);

    return size;
}
```

Each crime report is roughly 50 characters, so divide by 50 and you’ll know roughly how many crimes need to be stored in your hash table. To reduce collisions, you’ll need to make the size of your hash table \gg # of expected crimes. Target a load factor of 20-25% (i.e. 75-80% of the hash table is unused). You are encouraged to experiment with different approaches / sizes.

- As shown in the screenshot, instrument your program to (a)



```
C:\Windows\system32\cmd.exe
** Chicago Crime Lookup **
Crimes.csv
Areas.csv
>> Crimes file: 40572674 bytes
>> # crimes:      814499
>> HT size:      10400000
>> Collisions:    16703

Enter a case number> abcd
** invalid case #, try again...

Enter a case number> 1234
** invalid case #, try again...

Enter a case number> R03T1234
>> hash index: 971234 <<
** Case not found...

Enter a case number> R78A9218
** invalid case #, try again...

Enter a case number>
** Done **
```

output the crimes file size in bytes, (b) output the # of crimes input, (c) output the size of your hash table, and (d) report the # of collisions. You may use global variables to collect this data — the hash table itself *cannot* be a global variable, but you may e.g. have a global variable that keeps track of the # of collisions when building the table. [*Debug and instrumentation-related code is often global, but not the underlying primary data structures and code.*]

You are not required to build a separate hash table ADT in this assignment, but it's a good idea if you want to. When interacting with the user, note that he/she may enter invalid case numbers, or "valid-looking case numbers" for which no crime exists. Your program should handle this without crashing. Examples: "abcd", "1234", "R03T1234", and "R78A9218". See the screenshot on the previous page for how to handle each case.

The final requirements are just good programming:

- Use functions. The main program should be relatively short (no more than 50 lines), and likewise your functions.
- When compiling, use the compiler options `-std=c99` and `-Wall`. Treat all warnings as errors and fix.
- No global variables other than what is allowed for the purposes of instrumentation.

We are serious about these requirements. Use global variables instead of passing parameters? Score of 0. No functions? 0. No hash table? 0. Building but not actually using the hash table? 0.

Hashing...

With regards to hashing, it's all about the case numbers. As noted earlier, the case number is an 8-character string:

- In the format RXXYZZZZ, where XX is the area of the city where the crime occurred, Y is a single letter 'A'-'Z', and ZZZZ is a 4-digit number.

Since you are free to design your own hashing approach, you need to make sure the underlying platform supports the requested hash table size (especially since you are going to dynamically adapt based on the size of the crimes file). **Therefore**, whenever you call `malloc`, check the return value to make sure the memory was actually allocated:

```
ELEMENT_TYPE **hashtable = (ELEMENT_TYPE **)malloc(N * sizeof(ELEMENT_TYPE *));

if (hashtable == NULL)
{
    printf("*** Error: malloc failed to allocate hash table (%d elements).\n", N);
    exit(-1);
}
```

Get in the habit of doing this whenever you call `malloc`, even for small chunks of memory. In this assignment the odds are good you may run out of memory if you aren't careful in your design.

Electronic Submission

Before you submit, the top of each file should contain a brief overview, along with your name and other relevant info. Example:

```
//  
// Chicago Crime Lookup via hashing.  
//  
// <<YOUR NAME>>  
// <<PLATFORM and PROGRAMMING ENVIRONMENT USED>>  
// U. of Illinois, Chicago  
// CS251, Spring 2017  
// Project #05  
//
```

Your program should be readable with proper indentation and naming conventions; commenting is expected where appropriate. You can expect 90% of your grade to be based on correctness, and 10% on style.

Using Blackboard (<http://uic.blackboard.com/>), open “Assignments”, then “Projects”, and submit via “Project05” — attach all files need to build your program (.c, .h, and makefiles). We will be testing your program using gcc; we will not test your programs using Xcode, Visual Studio, nor Eclipse. The simplest way to submit multiple files is to .zip into a single archive. For example, if you are working on Windows, do the following: create a folder, copy your files into this folder, right-click on the folder, select “**Send to**”, then select the sub-option “Compress (zipped) folder”. Afterwards, submit the resulting .zip file.

You may submit as many times as you want before the due date, but we grade the last version submitted. This implies that if you submit a version before the due date and then another after the due date, we will grade the version submitted after the due date — we will **not** grade both and then give you the better grade. We grade the last one submitted. In general, do not submit after the due date unless you had a non-working program before the due date.

Policy

Late work **is** accepted. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you **cannot** work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is described here:

<http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf> .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty

include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at <http://www.uic.edu/depts/dos/studentconductprocess.shtml> .