ECE 361 Fall 2020
Homework #4

THIS ASSIGNMENT SHOULD BE SUBMITTED TO D2L BY 10:00 PM ON WED, 25-NOV-2020. THE ASSIGNMENT WILL BE GRADED AND IS WORTH 100 POINTS. IT IS EASIEST TO GRADE, AND I BELIEVE EASIEST FOR YOU SUBMIT YOUR SOURCE CODE FILES AS TEXT FILES INSTEAD OF TRYING TO CUT/PASTE YOUR CODE INTO A `.doc` OR `.docx` FILE.

WE WILL BE USING GITHUB CLASSROOM FOR THIS ASSIGNMENT SO PLEASE PUSH YOUR FINAL CODE AND TRANSCRIPT TO YOUR HOMEWORK #4 REPOSITORY. WE'D ALSO LIKE YOU TO SUBMIT AN ARCHIVE OF THE REPOSITORY TO YOUR D2L HOMEWORK #4 DROPBOX.

NOTE: This assignment owes much of its concept to Homework #3. As with Homework #3, I am providing less detail on the assignments, leaving more of the design and implementation up to you. I imagine this makes some (hopefully less of you) of you nervous, but you have successfully completed Homework #3 and I have even more confidence in you all.

Problem 1 (65 pts): Hashing and Hash tables

We will be leveraging some of the code we developed for our MLS app from Homework #3 for this problem. The goal is to implement an application that asks the user to select a conference and a Nickname for an MLS team. We will use a hash table-based database of MLS teams and retrieve the team's information (name, conference, points, PPG, game, wins-losses-draws). As with Homework #3, the team information will be read from a file. Each team has an entry in the `hw4_MLS2020.txt` file. The file has the same format as `MLS2020_final.txt`, except for the addition of the "nickname" field which is a character string (`%s`) that has been added to the end of each team's line in the file. The key for each element in the hash table will be team's nickname concatenated with the conference the team plays in (ex: "PTFCWEST" for the Portland Timbers who play in the Western Conference).

Each element in the hash table has the following structure (defined in `hash_table.h`):

```
typedef struct ht_item {
    char* key;
    void* value;
} ht_item;
```

Note that the information for a team is not stored in the hash table – the `value` is a `void*` pointer. This makes the hash table ADT more flexible since it can store a pointer to any `struct`, variable, or array.

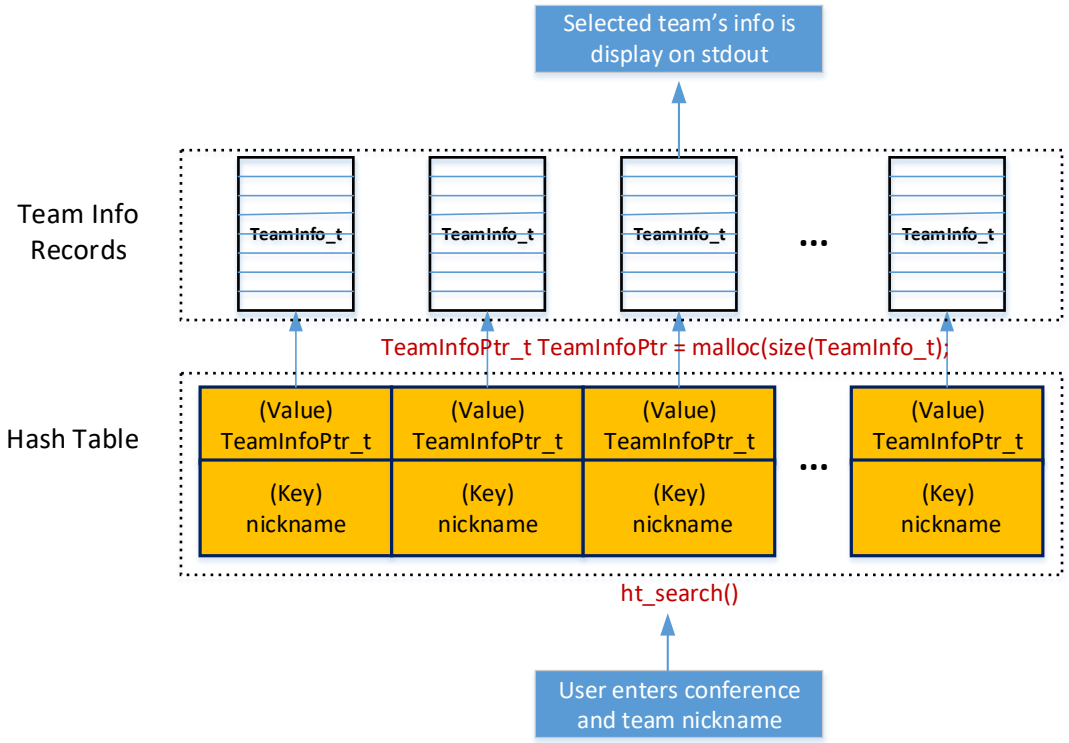
I opted to revise and refactor the code from Jame Routley's hash table tutorial (link is: https://github.com/jamesroutley/write-a-hash-table) for you instead of asking you to create your own hash table ADT. My intent, of course, is that `*value` is a pointer to a `TeamInfo_t` record. Since the hash table ADT is meant to be flexible and reusable it uses a generic (`void*`) pointer so there may be cases where you need to cast `*value` to a `TeamInfoPtr_t`. My revised code does not include the resizing code from the tutorial; instead I simply made the hash table size large enough to support the number of team records in the `input` file.

So what do you need to do? You need to create a new program that uses `stdin` (the terminal window) to query the user to enter a conference and a team nickname, looks up the record in the hash table and displays the information from that Team Info record in a readable format on `stdout` (the terminal window). The program should loop, prompting the user for additional records to look up until the user enters an empty line (just the `Enter` key) or '`q`' for the conference.

My application also checks for '?'. It the user enters '?' for the conference the program does an `ht_dump()` to display the keys and address for all of the entries in the hash table. Since the key is a concatenation of the nickname and conference all of the valid nicknames are listed. You don't need to do implement this functionality, but some of the nicknames are long and hard to remember so being able to refresh the list makes it easier than scrolling up and down in the terminal window to enter valid (and invalid) nicknames. What happens if the user enters an

invalid conference and/or nickname? The application should notify the user that there is no team information for their selected team in that conference.

My intended architecture for this app is shown below:



This is a new application, however you may find that it's easier to refactor (make changes to) functions that you wrote, or made use of in Homework #3 (ex: `MLSapp_Helpers.c`) .

Helpful hints:

- As stated above, the hash table does not store a `TeamInfo_t` record. All it stores is a pointer to a record. You will need to `malloc()` space for each `TeamInfo_t` record and pass the pointer to the `ht_insert()` function in the Hash Table ADT. My application creates an array of `TeamInfoPtr_t` that is large enough to hold pointers for all of the teams (26 for MLS 2020). The application allocates space and stores the pointer in the array as my `buildTable()` function adds entries in the hash table.
- Modify either the `parseTeamInfo()` function in `MLSapp_Helpers.c` or your version of it to use `fscan()` to scan for the additional string (`%s`) containing the nicknamc.
- Add additional functions to your refactored `MLSapp_Helpers.c`. I added two additional functions – one to generate a key and the other to convert a string to uppercase. Using the same function to generate keys for the records you are adding to the hash table and for generating a search key from the user input will make it more likely that a true match will occur. Why a function to convert as string to uppercase? It makes your application less susceptible to user preferences for entering a nickname. For example,

Timbers, TIMBERS, timbers*. If you look at my test results you see that I always enter nicknames in lower case but they are converted to upper case before being added as a key to the hash table.

*Actually none of those will work because the nickname for the Timbers is PTFC (**P**ortland **T**imbers **F**ootball **C**lub). As a side node – Portland's professional Women's soccer team also goes by (in some circles) PTFC – how efficient, we fans can share cheers. The NWSL PTFC is the **P**ortland **T**horns **F**ootball **C**lub.

- Watch out for aborts and abnormal exits. Check your results thoroughly and try numerous test cases – I thought I had everything in my application working and then, out of seemingly nowhere (more than once), I got a segmentation fault. Incidentally, the most common cause of a segmentation fault is either a NULL pointer or a pointer that wasn't initialized correctly (ex: `malloc()` failed or you didn't request enough bytes).
- Use conditional compilation and the `_VERBOSE_` flag to add debug messages to your application. I got stuck and almost got frustrated enough to start debugging w/ C-Lion (Jetbrain's C programming IDE) instead of the command line. I worked my way out of it and found the problem by judiciously adding diagnostic `printf()` statements which I then turned off by setting `_VERBOSE_` to 0 to make the output less cluttered.
- Use/modify the `test_hashtable.c` program I provided after you have made any changes to the `MLSapp_helpers` functions and the hash table ADT to isolate/fix problems before you try to bring-up and debug your final application. I found/fixed a nasty bug in my refactored `MLSapp_helpers.c` using the test program and I'm convinced it saved me debug time on my main application. My error was subtle and it would have been hard to ferret out if I had waited until I was debugging `hw4.c`.
- Write your `make` file early. I recompiled my application at least 30 times. Once you have the make file working rename it `Makefile` (no extension). Then you can just type `make` or `make clean` into the terminal.
- This should go without saying. Start early and work diligently. It took me a full day and night (I finished debugging at 11:30 PM on Saturday) to get my application working…and I was starting with a similar working application from last year's HW #4. Note also, that there is a 2nd programming problem in this assignment. Don't neglect it until the last minute because you are racing to get this application finished.

Grading rubric:

1. (10 pts) Refactor (make modifications to, or rewrite) the functions in `MLS_Helpers.c` or the functions you wrote to do similar functions to accommodate the "nickname" field. The updated `TeamInfo_t struct` is in `hash_table.h`. Be sure to describe your changes in the refactored file and include any files you change in your deliverables.
2. (30 pts) Write `main()`and helper functions to implement the functionality called out in the specification.
3. (5 pts) Make sure your code is organized and documented. Code should be written in a way that helps others (particularly Rishitosh and me) follow and understand your code. Use `doxygen` style comments for the functions you write. `Doxygen` comments help document your code and your API's. Neatness and good coding practice counts for this assignment.

4. (5 pts) Write, and use, a `Makefile` to automate the build process of your application.
5. (15 pts) Build your application using `make`. Execute your program. Include a transcript that shows your program being compiled and executed. The transcript should identify who you are and demonstrate that your program is working correctly.

Push all of your source code (`.c` and `.h`) files and transcripts to your GitHub repository for the assignment

Problem 2 (35 pts): Searching and sorting using the C standard library

Back in the early days of the personal computing era (circa 1981) when monitors were text-only (often with green or orange pixels) IBM moved the nascent industry forward by introducing the

CGA graphics standard (https://en.wikipedia.org/wiki/Color_Graphics_Adapter). CGA supported several resolutions, the highest being of 640 x 200. A color depth of 4-bits, supported 16 colors ($2^4 = 16$). Why such a low resolution and number of colors? CGA was designed with modes that supported TV's as well as computer monitors.

Each of the 16 colors was encoded into a 24-bit value and then converted to the 4-bit value that was sent directly to the monitor (`Red x 1, Green x 1, Blue x 1, Intensity x 1`). The full CGA 16 color pallet is shown in the table below:

Full CGA 16-color palette			
0	black #000000	8	dark gray #555555
1	blue #0000AA	9	light blue #5555FF
2	green #00AA00	10	light green #55FF55
3	cyan #00AAAA	11	light cyan #55FFFF
4	red #AA0000	12	light red #FF5555
5	magenta #AA00AA	13	light magenta #FF55FF
6	brown #AA5500	14	yellow #FFFF55
7	light gray #AAAAAA	15	white #FFFFFF

values (ex: #AA00AA) are given in Hex

Write an application that that returns the 24-bit color value for the named color. That is, `brown` is the key and `#AA5500` is the "value" for CGA color 6. Make use of the `qsort()` and `bsearch()` C standard library functions. The program should query the user for a color and return the 24-bit value for the color. As with the application in Part 1, the program should loop, prompting the user for additional records to look up, until the user enters an empty line (just the Enter key).

Grading rubric:

- 20 pts – C source code for the application
- 10 pts – Transcript showing that your application works correctly
- 5 pts – Quality of the code (i.e. organization, using meaningful variable names, comments as appropriate, readability (e.g. ease of understanding your code)

Push your source file(s) and transcripts to your GitHub repository for the assignment