Computer Science 1 – Program 6

KnightsRegistrar Rehashed (aka KnightsRegistrar Part Deux)

Assigned: 4/6/12

Due: 4/20/12 (Friday) at 11:55pm (WebCourses time)

Late submissions are NOT allowed. This is the FINAL due date!

Objective

Learn how to implement Priority Queues (Heaps) and Hash Tables.

The Problem

As the name suggests, this is a new, or rehashed, take on Program 4, KnightsRegistrar. Although students ultimately did get registered with the proposed environment in Program 4, there was a problem of fairness. The laptop check-out line was previously based on the "first come first serve" paradigm. Meaning, the first student in line would be the first one to receive a laptop. The first major change for this program is to more realistically model "real-life" registration, err, um, at least how it was done in the 90's! Instead of first come first serve, students will be placed into the laptop check-out line based on their class level. Seniors now have priority over juniors, juniors over sophomores, and sophomores over freshmen. So if the laptop check-out line has only twenty students, all Junior level or below, and then a Senior comes to register, that Senior will go to the front of the line, as they have preference. Instead of using a queue for the laptop check-out line, as in Program 4, you will now use a Priority Queue (Heap), where priority is based on class level (which, in turn, is based on credit hours).

Additionally, the KnightsRegistrar is also trying out new database software that permanently stores ALL student registrations in a hash table, utilizing the separate chaining method. As with Program 4, all successful registrations, on a given day, are saved into a linked-list, which is to be ordered alphabetically (by last name and then by first name). Since this list is used to print the daily summary report, this linked list must be reset (deleted/freed) on a daily basis. As the nodes of this list are deleted, they are to be inserted into the permanent hash table, named `knightsRegistrations`, which is maintained over all of the days (it will NOT be reset at each day).

In summary, for all intents and purposes, this assignment is effectively a copy of Program 4, following all details, rules and structures as specifically stated in the Program 4 write-up. You should simply modify your Program 4, making the following two changes:

1. The laptop check-out line must now be a priority queue (max heap)
   - This priority queue is based on the class level, which, in turn, is based on credit hours
   - Each student's credit hours are read in from the file (see output)
   - Input and output slightly change (see sample files)

2. Registrations must be "permanently" stored into a hash table
   - By permanent, we mean that this hash table will NOT reset over each day's simulation.
   - The daily summary, which is printed from the linked list, does get reset after each day
   - The nodes of linked list then get placed into master hash table, `knightsRegistrations`

Your Assignment is to write a simulation that models the aforementioned Registration over *n* number of days, where the simulation runs over each minute of every day, from 12 PM to 5 PM.

Implementation

You must use the following structs, **as shown**; meaning, do **NOT** change the names or even capitalization of ANY of the struct members below. To avoid errors, simply COPY and PASTE these structs, AS-IS, into your program. You may, at your own discretion, add struct members as you see fit. **HOWEVER**, the current struct members (as well as the name and typedef name of the struct) must NOT change at all, because if they do change, it could cause problems with the grading program, which will result in points being needlessly lost.

```
typdef struct ucfClass {
      char ID[10];      // class ID, ex: "COP-3502"
      char days[6];     // a combination of "MTWRF" (days)
      char time[20];    // class time, ex: "10:30 AM - 12:00 PM"
} class;

typedef struct ucfStudent {
      char lastName[21];      // student last name
      char firstName[21];     // student first name
      int ID;                 // student ID
      int creditHours;        // credit hours already taken
      int laptopID;           // serial number of the laptop the student picks up
      int errorCode;          // flag to determine if they will make mistakes
      int numClasses;         // number of classes the student will register
      class *classes;         // array of said classes (2 be dynamically allocated)
      int enterTime;          // time student arrived, in minutes, after 12:00 PM
      int timeLeft;           // countdown timer to measure the 5 min. reg. process
      int timeRegistered;     // Time student finished reg. and left Registrar
      int eCounter;           // Resolves the order that the student came in, if
                     //they entered at the same time and have the same class level.
      struct ucfStudent *next;  // pointer to next student in queue
} student;
```

*NOTE: In addition to these structures, you must use/make stack and queue structures as shown on the stack and queue code on the course website.

Laptop Waiting Line:

As mentioned, as soon as a student enters the Registrar's office, they must stand in line to check out a laptop. You will implement this line as a **priority queue**. The priority queue must be implemented as a MAX-heap containing students (which is ultimately just implemented as an array). The priority is based on students' class level which is derived from students' credit hours as follows:

Freshmen:	0+ credit hours
Sophomores:	30+ credit hours
Juniors:	60+ credit hours
Seniors:	90+ credit hours

All students of the same class level have equal priority. For example, two seniors, who have 95 and 100 credit hours, respectively, will have the same priority in the queue. But when two students are the same class level, we must have a way of choosing one over the other.

As such, we have the following rule: when two students have are the same class level, the one who arrives at the registrar first will get a laptop first. Additionally, if two students have the same class level and same arrival time, the first one to get a laptop will be the first one shown in the input file.

From this rule, you can understand the following. When removing students from the front of the priority queue, you will move the last node, in the heap, to the root position and then, most likely, will have to percolate down to a suitable position (this is just the rule of heap deletion), constantly swapping with the GREATER (higher priority) of the two children. When doing so, you should first compare students by class level. If the compared-to student has the same class level, then compare the students based on entry time to the registrar. If the student percolating downwards arrived earlier than BOTH of the students below them, then you can stop swapping, because the earlier arriving student has a HIGHER priority (assuming class level is the same). Else, if they arrived later, then you must swap with the higher priority of the two children. Finally, if the compared-to student has the same class level AND the same arrival time, there must be some sort of tiebreaker: the student who came first, in the input file, has a slight edge in priority (when class level and arrival time are the same).

And now the nuts and bolts:

- As with Program 4, students can make errors on their registrations. Follow the specs just like in program 4. However, this results in a rare situation that you must take care of: TWO students could possibly try to enter the laptop return line at the EXACT same minute. One of those students is finishing the first-time registration, and the other student is finishing the second registration (after error). So when these two students try to enter the laptop return line at the SAME minute, the question becomes, WHO enters first? Answer: the student that has the earlier arrival time will enter first.

Completed Registrations:

All successful registrations, on a given day, are saved into a linked-list, which is to be ordered alphabetically (by last name and then by first name). Although students may have the same last name or the same first name, it is guaranteed that all students have a unique first and last name combination. Since this list is used to print the daily summary report, this linked list must be reset (deleted/freed) on a daily basis. (this is just like Program 4)

As the nodes of this list are deleted, they are to be inserted into the permanent hash table, **`knightsRegistrations`**, which is maintained over all of the days (it will NOT be reset at each day). The "master UCF database", which was beyond the scope of Program 4, is now represented by this Hash Table that you will create. This hash table MUST utilize the separate chaining method. Your hash table must be an array of pointers, **of size 100**, where each pointer points to the head of that linked list. Here's how to make the hash table:

```
// Makes Hash Table:
// Initializes an array of 100 student pointers that all point to NULL.
student **knightsRegistrations = (student**) calloc(100, sizeof(student*));
```

You can use ANY hash function you like; actually, coming up with a quality hash function is one of the goals of this assignment. **However**, not only must your hash function must "work", but it must work effectively, meaning, it should distribute the items across the majority of the hash indices.

In order to check the integrity (accuracy and effectiveness) of your hash table, at the END of the simulation of ALL of the days, the final thing you should do is **call a function that we use to GRADE the final Hash Table**. The details of this function will NOT be given to you. All you need to know is HOW to call the function and what input to give the function.

This grading function is part of a header file, **`TAgrade.h`**, which **you MUST include** at the top of your program. NOTE: you MUST put the following include statement, in your program, **AFTER your student `struct` declaration**:

```
#include "TAgrade.h"
```

Again, at the END of the simulation of ALL days, you must then call this grading function EXACTLY as follows:

```
gradeMyHashTable(UCFregistrations);
```

The one, and only, input to the function is your hash table of all registrations, `UCFregistrations`. The function does not return anything (has a void return type). So you simply call the function EXACTLY as shown above. This function will take in your hash table of registrations and will grade it as follows:

1. ACCURACY of Hash Table: It will make sure that all of the expected students (from the correct output) are indeed saved in your hash table. So if the final (and correct) hash table has 42,568 students (as an example), this function will confirm that all of those 42,568 students are in your hash table (no more and no less).

2. EFFECTIVENESS of Hash Table: You will also be graded on the effectiveness of your hash function (as described previously in the write-up). Ideally, your hash function should distribute the student registrations amongst ALL of the 100 entries into your hash table. As discussed in class, the worst possibly hash function would result in ALL of the student registrations being stored at one, singular index of the hash table (epic fail!). You will get FULL credit, for the "effectiveness" portion of the hash table grade, as long as the indices of your hash table are 80% (or more) utilized. Meaning, out of 100 indices in the hash table, once all students have been added to the hash table, at most 20 of said indices can point to NULL in order to get full credit for “effectiveness.”

Everything Else:

Input and output files are to be named ***KnightsRegistrar2.in*** and ***KnightsRegistrar2.out*** respectively. Beyond this, refer to the Program 4 writeup! In all, this assignment will be exactly like assignment 4 with the exception of the changes mentioned in this write-up. Outputs from assignment 4 will not match outputs for this assignment, so refer to the new sample input/outputs provided.

<u>*WARNING***</u>**
Your program MUST adhere to the EXACT format (spacing capitalization, use of colons, periods, punctuation, etc) shown in the sample output file. The graders will use very large input files, resulting in very large output files. As such, the graders will use text comparison programs to compare your output to the correct output. If, for example, you have two spaces between in the output when there should be only one space, this will show up as an error even through you may have the program correct. You WILL get points off if this is the case, which is why this is being explained in detail. Minimum deduction will be 10% of the grade, as the graders will be forced to go to text editing of your program in order to give you an accurate grade.

<u>Included with this Write-up</u>
As usual, a sample input, with corresponding output, has been uploaded. In addition to these files, we've uploaded a sample `TAgrade.h` header file as well as a very basic `KnightsRegistrar2.c` file. Do NOT touch (modify) the `TAgrade.h` header file. Simply save it in the SAME directory as your `KnightsRegistrar2.c` file and call the function, `gradeMyHashTable`, as described above. Finally, the only purpose of our uploading the `KnightsRegistrar2.c` file is for you to see precisely WHERE the `TAgrade.h` file should be included (AFTER the student struct declaration).

<u>Grading Details</u>
Your program will be graded upon the following criteria:
1) Adhering to the implementation specifications listed on this write-up.
2) Your algorithmic design.
3) Correctness.
4) **<u>Use of heaps and hash tables. If your program does not use heaps and hash tables, you will NOT get credit for the assignment. Period.</u>**
5) The frequency and utility of the comments in the code, as well as the use of white space for easy readability. (We're not kidding here. If your code is poorly commented and spaced and works perfectly, you could earn as low as 85-90% on it.)
6) Compatibility to CodeBlocks. (If your program does not compile in CodeBlocks, you will get a sizable deduction from your grade.)
7) Your output MUST adhere to the EXACT output format shown in the sample output file.

<u>Restrictions</u>
Name the file you create and turn in KnightsRegistrar2.c. Although you may use other compilers, your program must compile and run using CodeBlocks. Your program should include a header comment with the following information: your name, course number, section number, assignment title, and date. You should also include comments throughout your code, when appropriate.

<u>Deliverables</u>
A single source file named KnightsRegistrar2.c turned in through WebCourses.