Data Structures Term Project
Dead Line: 3-July-2022
Team of 3 members
Final Grade Weight 20% (10 project + 10 CP)

Implement a console-based social networking tool that focuses on UCP campus events. It will run in the console/terminal at the command prompt, and the database of users and their data will be stored in text file. (You might think of this project a skeleton for a program/app that will eventually have a web-based GUI and database in your upcoming projects.

Each user will hold the following pieces of data:
Name
Login ID (email address) ← *key*
Password
Class Year
List of upcoming events that the user plans to attend
List of their own "timeline" posts
List of friends
For example, one entry in the text file might look like:

```
Rana Waqas
ranawaqas@ucp.edu.pk
Camels2014
BSSE 2014
"12 03 2014 16 30 ACM Meeting in 3s46 ",...
"Keeping calm...", "Go Camels!", "Out raising money for the college",...
gparker@umt.edu, aross@umt.edu,
```

In the text file, each line is a different field of data for the student, and a blank line in-between student records can delimit them.

Every time the program is run, the text file data should be read in, and every time it is exited/shut down, the updated data should be written back out.

Student records should be read from the file and stored in a hash table, keyed by student email address.

Student record objects are comprised of many fields, including (among others):

- **List of timeline posts**
- **List of friends**
- **List of campus events the student will be attending,...**

The data structures you choose to use for these fields are up to you, except that the list of friends should be an AVL tree, keyed by the friend's name. Note that keying by name will allow the list of friends to be printed in alphabetical order using an in-order traversal of the tree

While the program is running, all campus events that all users plan to attend will be kept on a heap-based priority queue, where the highest priority event is the next one to take place on campus. Once the event date passes, the event is removed from the heap. (You can use built-in tools for getting the current local date/time as well as for comparing two date/times to see

which comes first.) You should update the heap to reflect the events for the new current date/time every time someone logs in.

From the main menu prompt, the user can either:

- log in (if she is an existing user),
- create a new account, or
- exit the entire program.

The user-interaction is described below, but make sure your interface is very readable.

1. If the user chooses to create a new account, she will be prompted for her info and a new record will be created for her.

2. If she is an existing user, after logging in, she sees her "home" screen, which displays:

    1. the next campus event that is scheduled to take place,
    2. her most recent timeline entries (last 3-5 or so?)
    3. a listing of the campus events she plans to attend.

    She is then prompted with the option of:

    - posting to her timeline,
    - adding an event to her list that she plans to attend,
    - viewing a listing of her friends,
    - adding/removing a friend, or
    - logging out.

    1. If she chooses to post to her timeline, she is prompted to input text that will become the latest item in her timeline.
    2. If she chooses to add an event, she is prompted with a numbered list of upcoming events. She can choose an existing event from this list or choose to enter a new event. If she chooses to enter a new event, she will be asked for the event info, including date and time, and it should get written to her record. (When you eventually write this info back to the text file you will want to follow some standard format that will allow you to easily extract the time and date of the event, so consider this when you are prompting the user and storing the data.)
    3. If she chooses to list her friends, her friends are displayed along with their own latest timeline post. She can then choose a friend to see what campus events they plan to attend or to post a message to their timeline. (A timeline message posted by someone else should be labeled clearly with the poster's name.)
    4. If she chooses to add a new friend, she is then prompted for the email address of the friend she wishes to add. If the email address matches an existing user, that friend gets added to her list of friends (and she gets added to the list of that user). If not, then she is informed that the user does not exist in the system. You should devise a similarly easy way for the user to remove a friend.

5. The user is returned to her own “home screen” after completing any of these options.
6. If she chooses to log out, the main menu reappears for the next user.

The object-oriented design of the project is up to you and your development team.
You will need to do a lot of planning and design work as a group initially. This will allow you to break the project up into pieces and assign tasks to each member of your team. You should do this within the next few days so everyone can get started on implementing their part asap.
Coming up with a very clear specification for each method of each class during the design phase is important. This can change as you go (as long as all changes are communicated to the relevant team members), but you must have a clear place to start from.
It will be a good idea for everyone to work and meet together as much as possible so you can easily discuss/approve small tweaks in the design/specs with each other on the fly as you are implementing.
You will want to have very regular group meetings.

*A **tentative and approximate** point distribution....*

Overall design and documentation (including a diagram) of entire project	10%
Implementation/coding style	10%
Student Record (for each student)	15%
Hash Table (for the records)	20%
AVL Tree (for a user’s friends) • note: fewer points here because it’s *bigger* and *more* challenging to get right, not easier/smaller	5%
Priority Queue (for the campus events) • expired events get removed upon each login • (note that many users can log in and out before the program is exited/shut-down)	20%
User Interface (combining all the pieces together) • meets specs, satisfies description of functionality • output: friendly, readable • user input: friendly, easy-to-use • reasonably robust to bad user-input	20%