Assignment 3: Student Enrollment System using linked chains and a priority queue.
Due date: April 12, 2021 at 23:55 (Edmonton time)
Percentage overall grade: 7%
Penalties: No late assignments allowed
Maximum mark: 100

Assignment Specification:

Part 1:

In this first part of the assignment, you will implement an enrollment table to store the data for students registering in a course. The maximum capacity for the course is 50 students. The maximum number of slots in the table is 51 but no more students than 50 will be added to the table at any time.

You will implement and test the following two classes to register the first 50 students in the course by adding their records to the enrollment table. These classes are: StudentNode class and EnrollTable class that you will put in *enrollStudent.py*.

StudentNode class

The StudentNode class should have the following method:

`StudentNode(id, faculty, first, last)` – creates a new student node instance. The __init__ method takes four strings as input parameters: the student id, faculty, first name, and last name.

You will also implement setter and getter methods as follows –

`setID(id)` – sets the student id to the given id.

`setFac(faculty)` – sets the faculty abbreviation to the given faculty.

`setFirstName(first)` – sets the student's first name to the given first name.

`setLastName(last)` – sets the student's last name to the given last name.

`setNext(next)` – sets next as the next node.

`setPrev(previous)` – sets previous as the previous node.

`getID()` – returns the student id as a string.

`getFac()` – returns the faculty abbreviation as a string.

`getFirstName()` – returns a string representing the student's first name.

`getLastName()` – returns a string representing the student's last name.

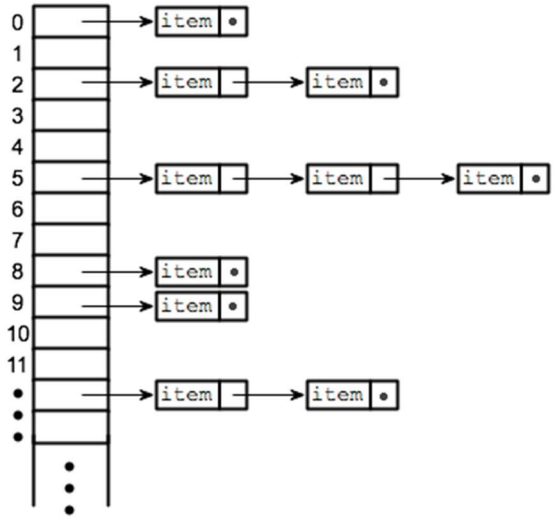
`getNext()` – returns the next node.

`getPrev()` – returns the previous node.

EnrollTable class

The enrollment table will be implemented as a python list with references to a singly linked chain off each of its slots. Its maximum capacity is 51. The enrollment table is initially empty with the references to the singly linked chains initially set to None when no students are enrolled yet in the course.

An example of how this data structure looks like is provided in the picture below except that the nodes of the singly linked chains are not showing the student node data but only showing an item inside them; they show however the links to the next nodes. Links to previous nodes in StudentNode instances should be set to None. Also, the maximum capacity of the table in the below example is not shown while the data structure that you will implement has a maximum capacity of 51.



`EnrollTable(capacity)` – creates a new empty enrollment table, basically a python list with a number of slots equal to the given capacity, and each slot is initialized to None. The table size should be set to 0.

To compute the index of the table slot in which a new student node object will be inserted, you will implement the following method:

`cmputIndex(studentID)` – this method takes a given student id string as a parameter and returns an integer that will represent the index in which a new student node will be inserted in the enrollment table. This method will split the student id string such that each two characters in the string represent one number made up of two digits. Since each id is made up of six digits, there will be three numbers which the method will sum up after squaring the third number. For example, if the student id is the string "123456", the method will split the string into 3 two-digit numbers, raise the last number to the power of 2, and then sum up these three values as follows: $12 + 34 + (56)^2$. The sum of these is 3182. The method then returns the remainder of the division of the sum by the maximum table capacity, i.e., sum % capacity. In this example with student id "123456", the method will return 3182 % 51 = 20. So, 20 will be the index in which the new student node will be inserted in the enrollment table.

Note: it is possible that the way this index is computed can result in a scenario in which many student nodes will be inserted at the same index in the enrollment table while some slots in the table might remain empty with a reference to None, as shown in the table example above in the picture.

`insert(item)` – this method inserts a given item, a student node object, in the enrollment table. This method does not return anything. To determine where to insert a new student in the table, this method will first call the cmputIndex() method to compute the index of the table slot in which the given item can be inserted. The insert() method will then traverse the singly linked chain referenced at the table slot with this index and insert the new student node in the singly linked chain in ascending order by student id starting from the head of the singly linked chain, if the linked chain is not empty. But if the chain is empty, this new student node will be added as the first node in the chain. Once the new student node is inserted in the table, the table size is incremented by 1.

`remove(studentID)` – this method removes from the enrollment table the node for the relevant student corresponding with the given student id. This method is called when a student drops out from the course. This method should first call the cmputIndex() method to compute the index of the table slot where the given student data may be found. This method will then traverse the singly linked chain referenced from this table slot at this index to find and remove the node for the given student id. This method returns True if the student has been successfully dropped from the course or False otherwise. Once the student node with the given studentID has been removed from the table, the table size is decremented by 1.

`isEnrolled(studentID)` – this method searches the enrollment table given a student id, and returns True if the corresponding student is found in the table, and False otherwise. This method should first call the cmputIndex() method to compute the index of the table in which the student with the given id may be found. If no singly linked chain is referenced off this table index, the method should return False. Otherwise, the method should traverse the singly linked chain referenced by the table slot at this index. Considering that the chains are sorted by ascending order of student id, the linear search should stop once this student id is found or once a student id larger than the given id is found in the chain. In the latter scenario, it means that the student with the given id is not found and the method will return False.

`size()` – this method returns the size of the enrollment table.

`isEmpty()` – this method returns True if the table is empty, i.e., size is 0, and False otherwise.

`__str__()` – returns the string representation of the enrollment table. The printout of the table contents should be between square brackets, separated by commas, and includes the index in which a record is stored and the student id, faculty and first and last names. Multiple records at the same index in the enrollment table can be printed following this index and separated by commas.

Part 2: PriorityQueue class

In this second part of the assignment, you will implement and test a priority queue for students that want to register in the course after the course has reached its maximum enrollment capacity. These students will be placed in a priority queue based on the faculty they are registered in.

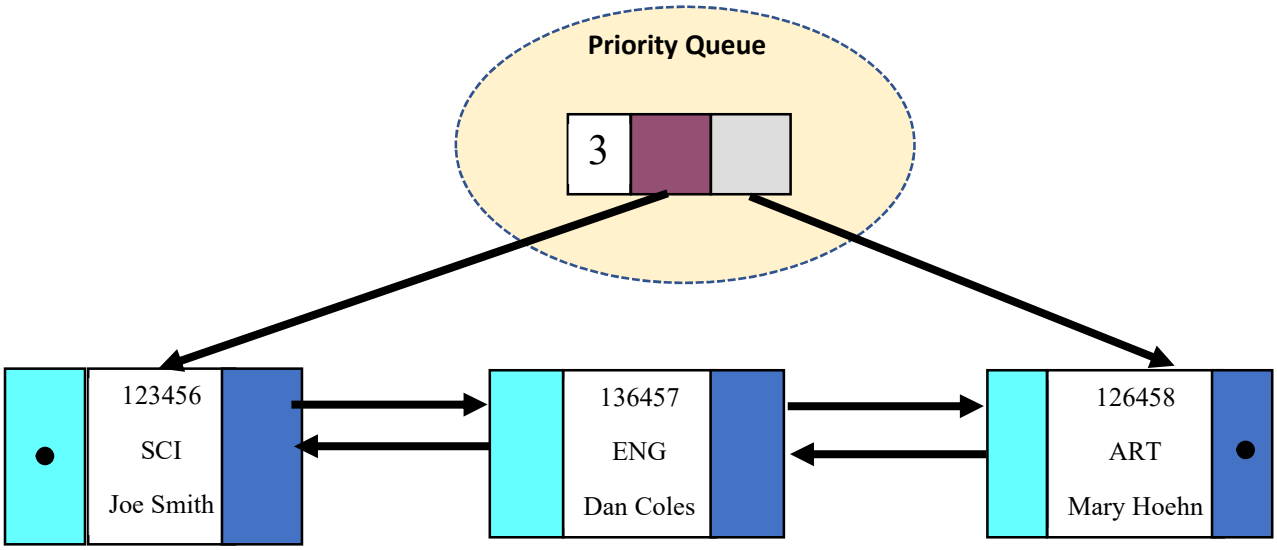
Create and test a PriorityQueue class in *enrollStudent.py*. You will implement this priority queue using a doubly linked chain data structure. The highest priority nodes will be dequeued from the front of the queue and the lowest priority nodes will be enqueued to the rear of the queue.

Priority is based on the student's faculty. Students with the highest priority are dequeued from the front of the queue. Faculty names and their corresponding priority values can be stored as key-value pairs in a Python dictionary. The priority is represented with an integer number and the faculty abbreviation as a string in the dictionary. The priority can be looked up based on the faculty abbreviation and ranges from 0 to 4, with 4 being the highest priority and 0 being the lowest priority.

The faculties are SCI, ENG, BUS, ART, EDU and their priority values are 4, 3, 2, 1, and 0, respectively.

For example, if the priority queue is initially empty and a student needs to be enqueued, this student can be added to the empty queue regardless of their faculty priority. If a second student from SCI needs to be enqueued, this second student is enqueued at the front of the queue if the first student was not from SCI. The second student is enqueued to the rear of the queue if the first student was also from SCI. Then, if a third student needs to be enqueued, the position in which this third student is inserted into the priority queue depends on their faculty priority. If their faculty has lower priority than any other student in the queue, then the third student is enqueued at the rear of the queue. If their faculty priority is lower than that of the student at the front of the queue and higher than that of the student at the rear of the queue, then the third student node is inserted between the two already existing nodes in the chain. However, if all students that need to be enqueued have the same faculty priority (e.g., all students are from ENG), then they are all enqueued based on a first come first serve order.

The following illustration of the priority queue with a doubly linked chain representation shows three enqueued students. The highest priority student "Joe" who is from SCI is at the front of the queue and will be dequeued first, when space becomes available in the course, to be then added to the enrollment table. The lowest priority student "Mary" who is from ART, is at the head of the rear of the queue and will be dequeued last if space becomes available in the course. The student from ENG, "Dan", has a lower priority than Joe but a higher priority than Mary. So, the node containing Dan's data is inserted between the two already existing nodes in the chain such that you will have to linearly traverse the chain that represents your priority queue to determine the position in which a new student node will be inserted based on their faculty priority.



Methods that you will implement in the PriorityQueue class:

`PriorityQueue()` – the __init__ method creates an empty doubly linked chain with both head and tail having references to None, and the size of the priority queue should be set to 0.

`enqueue(item)` – enqueues a new student node to the rear of the queue or traverses the queue to determine the position in which the new node will be inserted based on the faculty priority of the given item, i.e., the student node. This method updates the priority queue size by incrementing it by 1. This method does not return anything.

`dequeue()` – dequeues and returns the highest priority student node from the front of the queue and updates the size of the priority queue by decrementing it by 1. An Exception is raised (with an appropriate argument) if dequeueing from an empty queue.

`size()` – returns the number of students in the priority queue.

`isEmpty()` – returns True if the priority queue is empty, if its size is 0, or False otherwise.

`__str__()` – returns the string representation of the priority queue. The printout of the queue contents should be between square brackets, separated by commas, with the highest priority students' records printed out first including their student id, faculty and first and last names.

Test your PriorityQueue class thoroughly. You may want to use assertions to verify the behavior of your class implementation. Place these tests under if __name__ == "__main__": in enrollStudent.py for the marker to see.

Part 3: The main program

Write a main program in *assignment3.py* to do the following:

Prompt the user to either register or drop students from the course. The user can enter either 'R' or 'D' or 'Q' for registering or dropping or to exit the program, respectively. Any other responses are invalid, and should cause your program to re-prompt for a valid entry. Your program will prompt the user to enter the file name containing the input student data, and if the file cannot be read for any reason, continue to re-prompt until a valid filename is provided. Your program will then read the input students data from the file. Each line in the file represents one student record. Each student record includes the student id, faculty, and the student's first and last names separated by white space. You will need to read each line in the input file, i.e., each student record, as a string and split it into four different string tokens to separately obtain the student id, faculty, and first and last names. If a line in the text file contains invalid data or formatting, the entire program should exit gracefully with a warning message indicating which line of the text file that the program failed on.

If the user enters 'R' at the prompt, your program will prompt for the filename of the file containing the students to be registered and will read this file, create a student node for each record in the file, and populate the enrollment table accordingly. If the file contains more than 50 records, i.e., the number of records exceeds the capacity of the enrollment table, the records after line 50 will be placed in the priority queue.

Once all the students have been registered, the program will then print out the contents of the enrollment table between square brackets and separated by commas. The printout should include the index in which

each record is stored followed by the student record. The output should be written to a file named "*enrolled.txt*". This output file shows the current status of enrollment at any point in time but not the history of enrollment from previous program runs, i.e., you do not need to append to this file.

The program should also print out the contents of the priority queue or empty square brackets if the queue is empty. The output should be appended to a file named "*waitlist.txt*" to keep track of the history of the waiting list, i.e., the priority queue as it gets shorter or longer.

If the user enters 'D' at the prompt, your program will prompt for the filename of the file containing the records of the students to be dropped from the course. Your program will read this file and search the enrollment table for student nodes given the student ids provided in this file. Each time a student is dropped from the course, i.e., a spot becomes available in the enrollment table, your program should then dequeue the highest priority student from the priority queue and enroll this student in the course by adding this student node to the enrollment table. If the file contains a valid record for a student who is not currently enrolled in the course, a warning message should be displayed (see sample output below) and this student should be ignored.

After dropping students and registering new students from the priority queue, the program will then print the contents of the updated priority queue or empty square brackets if the queue is empty. The output is appended to the file named "*waitlist.txt*".

If the user enters 'Q' at the prompt, your program will finish after displaying a goodbye message.

Sample outputs for registering a subset of six students:

```
Would you like to register or drop students [R/D]: R

Please enter a filename for student records: sample_reg.txt
```

Sample outputs in *enrolled.txt*:

```
[0: 129051 ART Paul Johnston,
1:  124912 SCI Ahmed Salem,
4:  124592 ENG Jeanne Desjardins,
20: 123456 SCI Mary Soleiman,
37: 126013 SCI Faruk Hosney,
42: 124830 EDU Jane Osborne]
```

Sample outputs in *waitlist.txt*:

```
[]
```

Sample outputs for registering two students at same table index:

```
Would you like to register or drop students [R/D]: R

Please enter a filename for student records: sample_reg_sameIndex.txt
```

Sample outputs in *enrolled.txt*:

```
[0: 129051 SCI Jake Sun, 188451 ART Charlotte Williams]
```

Sample outputs in *waitlist.txt*:

```
[]
```

Sample outputs after registering fifty students from the input.txt file:

```
Would you like to register or drop students [R/D]: D

Please enter a filename for student records: sample_drop.txt

WARNING: Christie Fee (ID: 168258) is not currently enrolled and cannot be
dropped.
```

Sample outputs in *waitlist.txt*:

```
[177829 ENG Benjamin Schneider,
 181844 ENG Antoine Ledophin,
 183286 ENG Marina Alfred,
 186798 ENG Louis Windsor,
 176812 BUS John Deacon,
 185792 BUS Maha Bushara,
 173812 ART Nevine Salib,
 176681 ART Aimee Sauver,
 187987 ART Samuel Nathaneal,
 183476 EDU Jonathan Beck]
```

Assessment:

In addition to making sure that your code runs properly, we will also check that you follow good programming practices. For example, divide the problem into smaller sub-problems, and write functions/methods to solve those sub-problems so that each function/method has a single purpose; use the most appropriate data structures for your algorithm; use concise but descriptive variable names; define constants instead of hardcoding literal values throughout your code; include meaningful comments to document your code, as well as docstrings for all methods and functions; and be sure to acknowledge any collaborators/references in a header comment at the top of your Python files. Be sure to assert that all inputs to all methods are valid.

Restrictions for this assignment are that you cannot use break/continue, and you cannot import any modules other than your `enrollStudent` module. Doing so will result in deductions.

Rubric:

- Code quality and adherence to specifications: 20%
- StudentNode class and tests: 15%
- EnrollTable class and tests: 25%
- PriorityQueue class and tests: 25%
- Main program: 15%

Submission Instructions:

- All of your code should be contained in **TWO** Python files: **enrollStudent.py** and **assignment3.py**.
- Make sure that you include your name (as author) in a header comment at the top of all files, along with an acknowledgement of any collaborators/references.
- Please submit your TWO python files via eClass before the due date/time.
- Do not include any other files in your submission.
- Note that late submissions will not be accepted. You can make as many submissions as you would like before the deadline – only your last submission will be marked. So submit early, and submit often.

REMINDER: Plagiarism will be checked for

Just a reminder that, as with all submitted assessments in this course, we use automated tools to search for plagiarism. In case there is any doubt, you **CANNOT** post this assignment (in whole or in part) on a website like Chegg, Coursehero, StackOverflow or something similar and ask for someone else to solve this problem (in whole or in part) for you. Similarly, you cannot search for and copy answers that you find already posted on the Internet. You cannot copy someone else's solution, regardless of whether you found that solution online, or if it was provided to you by a person you know. **YOU MUST SUBMIT YOUR OWN WORK**.