CS-202
Data Structures

Assignment 1 (100 points)

Topics: Linked Lists, Stacks, and Queues

Due Date: 15-2-2022 11:55pm

Learning Outcomes

In this assignment, you will:

- Practice creating various functions of Linked Lists.
- Create Stacks and Queues using Linked Lists.
- Apply concepts of stacks and queues in real life scenarios.
- See how inheritance can be applied.

Pre-Requisites

For this assignment, you will require:

- Clear understanding of Linked Lists, Stacks, and Queues.
- Familiarity with Python syntax.
- Concepts of Object-oriented programming in Python.

Structure

This assignment has three parts. **Part 1** is on Linked Lists. **Part 2** is on creating Stacks and Queues using Linked Lists. **Part 3** is on applications of stacks and queues in real life scenarios.

Note: Generally it is a good idea to follow standards of Python programming (especially the naming conventions) as mentioned here.

The deadline for this assignment is strict. No extension will be provided. You can use your grace days if you are unable to finish your assignment on time.

Refer to the end of this manual (**Page 11**) for the Grading Scheme and instructions for automated testing.

Plagiarism Policy

The course policy about plagiarism is as follows:

1. Students must not share the actual program code with other students.

2. Students must be prepared to explain any program code they submit.

3. Students cannot copy code from the Internet.

4. Students must indicate any assistance they received.

5. All submissions are subject to automated plagiarism detection. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Best of Luck!

Part 1: Singly Linked Lists

Expected Time: 3-4 hours

In this part, you will be applying what you learned in class to implement different functionalities of a linked list efficiently. The basic layout of a linked list is given to you in Part1.py file.

It contains a class Node that has two attributes: *data* and reference to the *next* node.

Task:

Your task is to complete all functions of the Linked List. *Do not modify any function definitions or add/remove any parameters that each function receives.*

It is necessary to complete all functions within this part before moving to the next. Else, there is no guarantee that subsequent parts would work.

Note: When implementing functions, pay special attention to corner cases such as deleting from an empty list. There should be no hard coding. Your functions should be reusable for future parts as well.

Member functions:

Write implementation for the following methods as described here.

Note: Return *None* (unless asked otherwise) wherever the linked list has no elements (corner cases).

- get_head(self): **(2 points)**
  - o Return the data value of the head of the Linked List. Return None if the linked list is empty.
- get_tail(self): **(2 points)**
  - o Return the data value of the tail of the Linked List. Return None if the linked list is empty.
- is_empty(self): **(1 points)**
  - o Return *True* if Linked List is empty. Else return *False*.
- insert_at_head(self, data): **(4 points)**
  - o Insert a new node at the head of the Linked List. Data will typically be a *string*.
- insert_at_tail(self, data): **(4 points)**
  - o Insert a new node at the tail of the Linked List. Data will typically be a *string*.
- insert_in_between(self, data, after, before): **(5 points)**
  - o Insert a new node at the position specified by *after* and *before*. These are two *strings*. For simplicity, you can assume both are present in the Linked List one after the other. Therefore, no error handling is required.
- delete_head(self): **(2 points)**
  - o Delete the head of the Linked List.
- delete_tail(self): **(2 points)**

  - Delete the tail of the Linked List.
- delete_any(self, data): **(5 points)**
  - Delete the first instance of node with the given *data*. Data will typically be a *string*.
- get_length(self): **(3 points)**
  - Get the length of the Linked List. Return 0 if the list is empty, else return the length.
- get_element(self, data): **(4 points)**
  - This is sort of a search function. It returns *False* if the *data* does not exist within the list. Else, it returns the data itself.
- reverse_list(self): **(6 points)**
  - Reverse the order of elements of the Linked List.
    E.g. [‘a’, ‘b’, ‘c’] becomes [‘c’, ‘b’, ‘a’]
    Note: You are not allowed to create another Linked List for this function.

In addition to implementing the operations above, answer the following questions:

1. What is the time complexity of each operation given above? (write at the start of each function as a comment)
2. How can the time complexity of operations of the Linked List that you have implemented be improved? (write at the end of Part 1 file as a comment)

Part 2: Stacks and Queues

Expected Time: 1-2 hours

In this part, you will be using the Linked List you created in Part 1 to implement stacks and queues. The basic layout of stacks and queues is given to you in Part2.py file.

It imports the class definitions for *Node* and *Linked List* from Part 1. In addition to that, it contains a class *Stack* and *Queue* that have one attribute of its own: *capacity*.

Note: *Stack* and *Queue* are child classes of Linked List class. They inherit all the functions and attributes of the Linked List.

Note: Typically, in Linked List-based implementations of Stacks and Queues, an upper limit on number of elements might not exist; however, in this Part, you have to make sure that each data structure does not exceed its capacity.

Task:

Your task is to complete all functions of the *Stack* and *Queue* class. *Do not modify any function definitions or add/remove any parameters that each function receives.*

It is necessary to complete all functions within this part before moving to the next. Else, there is no guarantee that subsequent parts would work.

Note: When implementing functions, pay special attention to corner cases such as deleting from an empty stack or queue. Moreover, do not code each function from scratch! You have written **all the functions** that you need in Part 1. Smartly apply relevant conditions and reuse those functions. Failure to do so might result in a deduction of marks.

Stack

Member functions:

Write implementation for the following methods as described here.

- push(self, data): **(6 points)**
    - Insert a node in the stack. Return *None* if the stack is full.
- pop(self): **(2 points)**
    - Remove a node from the stack. Return *None* if the stack is empty.
- top(self): **(2 points)**
    - Return the top of the stack. Return *None* if the stack is empty.

Queue

Member functions:

Write implementation for the following methods as described here.

- enqueue(self, data): **(3 points)**
    - Insert a node in the queue. Return *None* if the queue is full.
- dequeue(self): **(3 points)**
    - Remove a node from the queue. Return *None* if the queue is empty.
- get_front(self): **(2 points)**
    - Return the front of the queue. Return *None* if the queue is empty.
- get_rear(self): **(2 points)**
    - Return the rear of the queue. Return *None* if the queue is empty.

Part 3: Application of Linked Lists, Stacks, and Queues

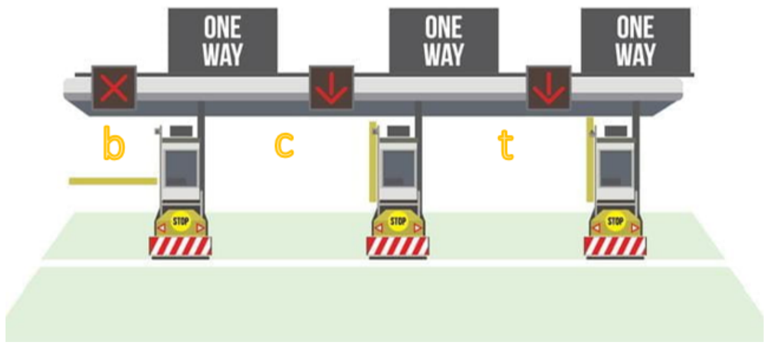
Expected Time: 5-6 hours

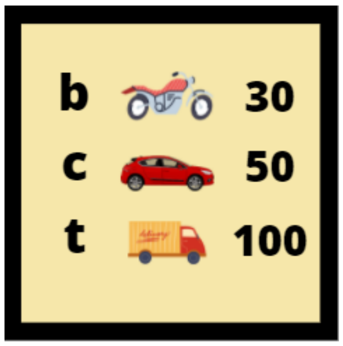
Note: Read the entire question carefully before attempting!

In this part, you will be using the Linked List, Stack, and Queue that you created in Part 1 and Part 2 to apply these data structures in a real life scenario. The skeleton code is given to you in Part3.py file.

It imports the class definitions for *Node, Linked List, Stack, and Queue* from Part 1 and Part 2.

Consider the following scenario **carefully**:





b	30
c	50
t	100

There are three types of toll plazas: *b, c and t.*

Toll plaza b is for bikes only. Toll place c is for cars only. Toll plaza *t* is for trucks only.

The image above shows the tax for each type of vehicle.

Each toll plaza has a **max capacity** of **5** vehicles of its own type only.

Each vehicle is recognized by its *type identifier* and *ID*. For example, b_1 is a bike with type identifier b and ID 1, c_7 is a car with type identifier c and ID 7 and t_10 is a truck with a type identifier t and ID 10. It does not matter what the ID of any vehicle is. They can be in random order as well. Focus on the type of the vehicle mainly.

Task:

Your task is to:

- Manage incoming and outgoing traffic to the toll plazas. Decide upon a data structure that should hold the vehicles. Then direct each type of vehicle to its respective plaza.
- Calculate the total toll collected on a given day.
- Handle vehicles that need to wait when one of the toll plazas is completely full.

To handle vehicles that need to wait, we will use a single Linked List. Keep adding all extra vehicles into the Linked List. When a particular type of vehicle leaves the toll plaza, check if there is the same type of vehicle waiting in the List. If so, remove it from there and insert into its respective toll plaza’s stack or queue. Make sure that you maintain the order of vehicles accurately! For instance, if b_10 comes into the List first and b_12 comes later, b_10 should reach the toll plaza first.

Hint: Think of how you should insert nodes into the Linked List for this purpose.

Functions:

There are a total of 4 functions in Part3.py. The following function has been implemented for you:

- find_vehicle_in_list(type_of_vehicle, waiting):
  - This function is used to find a particular type of vehicle in the waiting linked list. It takes two parameters: *type_of_vehicle* and *waiting*.
    type_of_vehicle is a string that can take the value of “bike”, “car”, or “truck”.
    waiting is the linked list that stores waiting vehicles.
    The function returns the vehicle name (type identifier_ID e.g. b_10) from the waiting list in order of their arrival whenever required.
    Go over the function in order to understand how it works.

Write implementation for the following methods as described here.

- incoming_traffic(data, DS1, DS2, DS3, waiting, tax_collection):
  - This function is called whenever a new vehicle comes to one of the toll plazas.
    data refers to the vehicle name (e.g. b_19).
    DS1, DS2, and DS3 are the three data structures (one for each plaza). DS1 holds bikes, DS2 holds cars, and DS3 holds trucks. **Keep this order consistent throughout!**
    waiting refers to the Linked List that store waiting vehicles.
    tax_collection is a dictionary that stores the total tax collected for each type of vehicle.

    Cater to each type of incoming vehicle and direct it to its relevant plaza (DS1, DS2, or DS3). Be mindful of the capacity of each plaza. If exceeded, make use of the *waiting* linked list. Also, update the dictionary that stores total tax collected for each type of vehicle.
    Expected # of lines of code for each vehicle: 6

- outgoing_traffic(type_of_vehicle, DS1, DS2, DS3, waiting, tax_collection):
  - This function is called whenever a particular type of vehicle leaves one of the toll plazas.
    type_of_vehicle is a string that can take the value of “bike”, “car”, or “truck”.
    DS1, DS2, and DS3 are the three data structures (one for each plaza). DS1 holds bikes, DS2 holds cars, and DS3 holds trucks.
    waiting refers to the Linked List that stores waiting vehicles.
    tax_collection is a dictionary that stores the total tax collected for each type of vehicle.

Cater to each type of vehicle leaving the plaza and remove it from your stack/queue. In addition to that, also check if you need to search for the particular type of vehicle in the waiting list or not (apply relevant condition). If you need to search for the vehicle, use the *find_vehicle_in_list* function here.

Note: Do not call this function with every vehicle leaving a toll plaza! Logically think when you need to call it else it will raise an error.
If you do need to search for the vehicle within the waiting list, handle all cases appropriately using functions of the Linked List that you made in Part 1. Send the vehicle back to its respective toll plaza.
Expected # of lines of code for each vehicle: 9

- total_tax(tax_collection):
    - Return the total tax collected
      Expected # of lines of code for each vehicle: 1

- tax_collection dictionary is defined as follow:
    - tax_collection = {"bike_total": 0, "car_total": 0, "truck_total": 0}

Grading Scheme

There is a single file that will be used to test all parts: *test.py*

To test your implementation, run this file on the terminal in the same folder as your other files. Each function across all three parts is tested independently of each other. The test will show marks for each individual function, along with your total marks. Overall breakdown is as below:

Part 1: 40 marks

Part 2: 20 marks

Part 3: 40 marks

Each function carries individual marks. The points breakdown is written in front of each function. Part 3 is graded as follows:

Task	**Points**
Directing incoming traffic	10
Storing in waiting list	10
Managing outgoing traffic	5
Managing removal from waiting list into toll plaza again	10
Calculating total tax	5
Total	40

There will be no partial marking between each function. Scores assigned by the grader will be final. Pay special emphasis to corner cases.

Note: You can create an object of the particular class within each part and use the functions that you have written in order to test your implementation. However, please make sure to comment out this portion before you run your tests as it can mix up with the test cases.

Best of Luck.