Lab 10: Stacks and Queues

Goals for this lab:

- Using a LinkedList, implement a Stack
- Using a LinkedList, implement a Queue
- Write and execute Python programs using the given instructions

General Guidelines:

- Use meaningful variable names
- Use appropriate indentation
- Use comments, especially for the header, variables, and blocks of code.

Example Header:

```
# Author: Your name
# Date: Today's date
# Description: A small description in your own words that describes what
the program does. Any additional information required for the program to
execute should also be provided.
```

Example Function Docstring:

```
"""
General function description
:param p1: DESCRIPTION
:type p1: TYPE
:param p2: DESCRIPTION
:type p2: TYPE
:return: DESCRIPTION.
"""
```

1. Set Up

You should use the provided **LinkedList.py** file as a starting point for implementing the requested functionality for the programs in this lab. Note, you should not import the `LinkedList` or `Node` class, from **LinkedList.py**, but rather use them as a reference to implement the requested functionality for the lab. Additionally, you should use the **inputStack.txt** and **inputQueue.txt** files for the associated programs.

2. Stack

The Stack data model organizes data in such a way that the most recently added element is always the first one to be removed, i.e. the LIFO strategy. This typically includes the use of operations like push and pop to add and remove elements, as well as peek to examine what element is current at the top of the Stack. For this portion of the lab, you will need to implement a Stack using a LinkedList to store the data.

To complete this program:

- In a file named **Stack.py**,
    - All member variables should be private
    - Define a class named `Node` which contains:
        - A constructor that takes in a parameter representing the data to be stored in the `Node`
            - Assigns the parameter to a new member variable
            - Defines a new member variable to keep track of the next `Node` in the chain and initialize it to `None`
        - Appropriate getter and setter functions
    - Define a class named `EmptyStackException`, that inherits `Exception`, and contains:
        - A constructor that takes in a parameter representing the action being performed when the exception was generated
            - Defines a new variable to store a message explaining that the stack was empty and thus the specified action cannot be completed
            - Calls the superclass's constructor and passes the message to it
    - Define a class named `Stack` which contains:
        - A constructor that takes in no parameters (other than self) and defines a new variable to store the head `Node` of the LinkedList representing the Stack, initialized to `None`
        - A function named `push()` that takes in one parameter representing the data to be added to the Stack, returns nothing, and should:
            - Store the data in a new `Node`, and prepend the `Node` to the LinkedList
        - A function named `pop()` that takes in no parameters, returns the data at the top of the Stack, and should:
            - Check if the Stack is empty, and if so raise an `EmptyStackException` passing in the function name as a string
            - Otherwise, remove the `Node` from the front of the LinkedList, and return the data in the `Node`
        - A function named `peek()` that takes in no parameters, returns the data at the top of the Stack, and should:

        - Check if the Stack is empty, and if so raise an `EmptyStackException` passing in the function name as a string
        - Otherwise, return the data in the `Node` at the front of the LinkedList
    - A function named `clear()` that takes in no parameters, returns nothing, and removes all `Nodes` from the LinkedList
        - Think about the easiest possible way to do this (it should only take one line of code), and remember that Python is a garbage collected language!
    - A new version of the `__str__()` function, that returns a string containing all of the data in the Stack, delimited by commas
        - If there is no data in the Stack, return an empty string
- In a file named **mainStack.py**,
    - Define a `main()` function that will:
        - Create a new instance of your Stack class and store it in an appropriate variable
        - Open the provided file **inputStack.txt** for reading
        - While there are still more lines in the file:
            - Read in a line containing a Stack operation
            - If the push operation is specified, attempt to push the provided value onto the Stack, using the appropriate Stack function, and output that the value was pushed onto the Stack
            - If the pop operation is specified, attempt to pop a value off of the Stack, using the appropriate Stack function. If this succeeds output that the value was popped off of the Stack, otherwise handle the exception and output that the Stack was empty and that no value could be popped
            - If the peek operation is specified, attempt to peek at the value on top of the Stack, using the appropriate Stack function. If this succeeds output the value at the top of the Stack, otherwise handle the exception and output that the Stack was empty and that no value could be peeked at
            - If the clear operation is specified, reset the Stack to an empty state using the appropriate Stack function
            - Otherwise, print out all of the values in the Stack using the appropriate Stack function
        - Your program should not crash even if Stack related exceptions are thrown

Complete the program, making sure to execute it to verify that it produces the correct results. Note that you will submit the **Stack.py** and **mainStack.py** files to Canvas.

3. Queue

The Queue data model organizes data in such a way that the element that has been in the Queue the longest is always the first one to be removed, i.e. the FIFO strategy. This typically includes the use of operations like enqueue and dequeue to add and remove elements, as well as peek to examine what element is current at the front of the Queue. For this portion of the lab, you will need to implement a Queue using a LinkedList to store the data.

To complete this program:

- In a file named **Queue.py**,
  - All member variables should be private
  - Define a class named `Node` which contains:
    - A constructor that takes in a parameter representing the data to be stored in the `Node`
      - Assigns the parameter to a new member variable
      - Defines a new member variable to keep track of the next `Node` in the chain and initialize it to `None`
    - Appropriate getter and setter functions
  - Define a class named `EmptyQueueException`, that inherits `Exception`, and contains:
    - A constructor that takes in a parameter representing the action being performed when the exception was generated
      - Defines a new variable to store a message explaining that the queue was empty and thus the specified action cannot be completed
      - Calls the superclass's constructor and passes the message to it
  - Define a class named `Queue` which contains:
    - A constructor that takes in no parameters (other than self) and:
      - Defines a new variable to store the head `Node` of the LinkedList representing the Queue, initialized to `None`
      - Defines a new variable to store the tail `Node` of the LinkedList representing the Queue, initialized to `None`
    - A function named `enqueue()` that takes in one parameter representing the data to be added to the Queue, returns nothing, and should:
      - Store the data in a new `Node`, and append the `Node` to the LinkedList
        - Note, pay attention to the case where the LinkedList is empty, versus if it contains at least one element, and how that affects updating the head and tail `Nodes`
    - A function named `dequeue()` that takes in no parameters, returns the data at the top of the Queue, and should:
      - Check if the Queue is empty, and if so raise an `EmptyQueueException` passing in the function name as a string
      - Otherwise, remove the `Node` from the front of the LinkedList, and return the data in the `Node`

        - Note, pay attention to the case where the LinkedList contains exactly one element, versus if it contains more than one element, and how that affects updating the head and tail `Nodes`
    - A function named `peek()` that takes in no parameters, returns the data at the front of the Queue, and should:
        - Check if the Queue is empty, and if so raise an `EmptyQueueException` passing in the function name as a string
        - Otherwise, return the data in the `Node` at the front of the LinkedList
    - A function named `clear()` that takes in no parameters, returns nothing, and removes all `Nodes` from the LinkedList
        - Think about the easiest possible way to do this (it should only take one line of code), and remember that Python is a garbage collected language!
            - Don't forget to update both the head and tail `Nodes`!
    - A new version of the `__str__()` function, that returns a string containing all of the data in the Queue, delimited by commas
        - If there is no data in the Queue, return an empty string
- In a file named **mainQueue.py**,
    - Define a `main()` function that will:
        - Create a new instance of your Queue class and store it in an appropriate variable
        - Open the provided file **inputQueue.txt** for reading
        - While there are still more lines in the file:
            - Read in a line containing a Queue operation
            - If the enqueue operation is specified, attempt to enqueue the provided value into the Queue, using the appropriate Queue function, and output that the value was enqueued into the Queue
            - If the dequeue operation is specified, attempt to dequeue a value from the Queue, using the appropriate Queue function. If this succeeds output that the value was dequeued from the Queue, otherwise handle the exception and output that the Queue was empty and that no value could be dequeued
            - If the peek operation is specified, attempt to peek at the value at the front of the Queue, using the appropriate Queue function. If this succeeds output the value at the front of the Queue, otherwise handle the exception and output that the Queue was empty and that no value could be peeked at
            - If the clear operation is specified, reset the Queue to an empty state using the appropriate Queue function
            - Otherwise, print out all of the values in the Queue using the appropriate Queue function
        - Your program should not crash even if Queue related exceptions are thrown

Complete the program, making sure to execute it to verify that it produces the correct results. Note that you will submit the **Queue.py** and **mainQueue.py** files to Canvas.

Submission

Once you have completed this lab it is time to turn in your work. Please submit the following files to the Lab 10 assignment in Canvas:

- **Stack.py, mainStack.py**
- **Queue.py, mainQueue.py**

Sample Output

Stack.py, mainStack.py

```
Sorry, the stack is empty and we cannot peek!
Pushed 37
Pushed 84
Pushed 24
Popped 24
84, 37
Pushed 5
Pushed 46
Pushed 38
Pushed 98
Popped 98
38, 46, 5, 84, 37
Popped 38
Peeked at 46
Pushed 66
Cleared out the stack.
Pushed 13
13
Pushed 89
Popped 89
Popped 13
Sorry, the stack is empty and we cannot pop!
```

Queue.py, mainQueue.py

```
Sorry, the queue is empty and we cannot peek!
Enqueued Emmeric
Enqueued Borus
Enqueued Nyer
Dequeued Emmeric
Borus, Nyer
Enqueued Bria
Enqueued Jeralia
Enqueued Rye
Enqueued Ari
Dequeued Borus
Nyer, Bria, Jeralia, Rye, Ari
Dequeued Nyer
Dequeued Bria
Dequeued Jeralia
Peeked at Rye
```

```
Enqueued Tarinne
Cleared out the queue.
Sorry, the queue is empty and we cannot dequeue!
Enqueued Iowyn
Iowyn
Enqueued Hadena
Dequeued Iowyn
Dequeued Hadena
Sorry, the queue is empty and we cannot dequeue!
```

Rubric

For each program in this lab, you are expected to make a good faith effort on all requested functionality. Each submitted program will receive a score of either 100, 75, 50, or 0 out of 100 based upon how much of the program you attempted, regardless of whether the final output is correct (See table below). The scores for all of your submitted programs for this lab will be averaged together to calculate your grade for this lab. Keep in mind that labs are practice both for the exams in this course and for coding professionally, so make sure you take the assignments seriously.

Score	Attempted Functionality
100	100% of the functionality was attempted
75	<100% and >=75% of the functionality was attempted
50	<75% and >=50% of the functionality was attempted
0	<50% of the functionality was attempted