The goal of this lab is to practice writing code using Exceptions, Bounded Queues, Circular Queues and Single Linked List.

**Q1** In this program you must write three functions. Here is the definition of the three functions:

```
def main()
def readAccounts(infile)
def processAccounts(accounts)
```

in the main function the user is asked to enter the name of the file. If the file does not exist, the main function should raise an exception. If raised, the main function should be able to catch the IOError exception. Upon catching the exception, the main function prints an error message and exits the program. Here is a sample run that shows what happens when the filename entered by the user does not exist:

```
Enter filename >doesnotexit.txt
IOError.Input file does not exist
```

If the file exists, the main function opens the file in read mode and proceeds to call a function readAccounts. The main function passes the TextIOWrapper object (the file handler) as an argument to this function.

Each line in the file is a record of a person's name followed by a colon symbol followed by a number that represents the amount the person has in their account. The function `readAccount` reads the file one line at a time and stores the `name:account` as a key:value pair inside the dictionary. If the amount value associated to a name is not a valid number the function should raise an exception and that `name:amount` with the invalid amount is not added to the dictionary. If an exception is raised, the following message is printed by the function:

```
ValueError. Account for Smith not added: illegal value for
balance
```

After raising and catching the exception the program should continue to the next record in the file.

Once all records with valid amounts have been added into the dictionary, the `readAccounts` function should return this dictionary.

The main function should now call the `processAccounts` function and pass the dictionary that was returned by `readAccounts` as an argument to this function. This function should ask the user to enter the name of a person. If the name does not exist in the dictionary a KeyValue Error is raised and caught by the program. The following message is displayed in this case:

```
KeyError.Account does not Exist.Transaction cancelled.
```

After catching the exception, the function asks the user to enter name again. If the name exists in the dictionary, the function asks the user to enter the amount value that would be added to the existing amount associated to the name in the dictionary. If the user enters a value that is not a float or integer, the function raises and catches a ValueError exception and the following message is displayed:

```
Value Error.Incorrect Amount.Transaction cancelled.
```

After catching the exception, the program allows the user to continue entering name of the account holder and the transaction amount until the user enters Stop. If the name exists in the accounts dictionary and the amount is a valid integer or float value, the transaction goes through and the function reports the amount in the account has been updated. Please note you don't have to update or write to accounts.txt file. You just need to update the amount in the dictionary.
**Here is a complete sample run of the program:**

```
Enter filename >accounts.txt
ValueError. Account for Smith not added: illegal value for balance
ValueError. Account for George not added: illegal value for balance
Enter account name, or 'Stop' to exit: bob
KeyError.Account does not Exist.Transaction cancelled
Enter account name, or 'Stop' to exit:Bob
Enter transaction amount for Bob:30
New balance for account Bob: 264.7
Enter account name, or 'Stop' to exit:Zoya
Enter transaction amount for Zoya:hello
Value Error.Incorrect Amount.Transaction canceled
Enter account name, or 'Stop' to exit:Johnson
Enter transaction amount for Johnson:-20
New balance for account Johnson: 771.56
Enter account name, or 'Stop' to exit:Stop
```

**Q2** Implement the Bounded Queue and Circular Queue ADT as seen in class. The python implementations of these data structures are in the lecture slides. Make sure you raise exceptions when – peek()/dequeue() if the queue is empty. – enqueue() if the queue is full .

Compare the runtime of dequeue() methods in Bounded Queue and Circular Queue. In bounded queue, once you dequeue an item, the remaining items in the list will be shifted left. This happen because we are using the pop(index) method

in the list class to dequeue. Therefore, the dequeue in Bounded Queue is O(n). However in Circular Queue, the front most item is accessed in the list by using the subscription operator and the head is incremented by 1. Therefore, the dequeue in a Circular Queue is O(1).

So in theory, dequeue in Circular Queue should be much faster than the dequeue in a Bounded Queue. Write your code to justify whether this hypothesis is true or false.

Here are the steps for this experiment:

1. Create a Circular Queue object and enqueue 100000 items in it.
2. Create a Bounded Queue object and enqueue 100000 items in it
3. Compute the time required to dequeue all items from the Circular Queue.
4. Compute the time required to dequeue all items from the Bounded Queue.

Here is how to use the time module :

```
import time
start = time.time()
# The statement(s) that you want to test
end = time.time()
time_interval = end - start
```

5. Print the dequeue() runtime for each queue

   **Please note** : We need to enqueue many items in the Circular and Bounded Queue in order to get a reasonable run time for both queues.

**Here is the sample output:**

```
For Bounded Queue, the total runtime of dequeing 100000 is:
2.2983570098876953
For Circular Queue, the total runtime of dequeing 100000 is:
0.11895871162414551
```

**Q3** You are given the file **LinkedList.py**. There are two classes in the file – the SLinkedListNode class and the SLinkedList class. Some of the methods in the SLinkedList class have been done in the lectures in the class. Your task is to complete the implementation of the **insert method** in SLinkedList class. Once you have completed the method, you can use the code in the main function to test the **insert method**. Here is the output that you should get if the insert method has been implemented correctly.

```
Original List
A->4->2->77->6->Z->

After inserting the word start at position 0
start->A->4->2->77->6->Z->

After inserting the word end at position 7
start->A->4->2->77->6->Z->end->

After inserting middle at position 4
start->A->4->2->middle->77->6->Z->end->
```