ESTRUCTURA DE DATOS Y ALGORITMOS

LABORATORIO N° 06

COLAS AVANZADAS

Alumno	Nota
Mamani Sayco Gian Franco	
Grupo	**3C24-C**
Fecha de Entrega	**03/05/2023**
Docente	**Renato Usnayo Cáceres**

DISEÑO Y DESARROLLO DE SOFTWARE
PROGRAMA DE FORMACIÓN REGULAR

OBJETIVOS:

- Definición, gestión de colas, aplicaciones, implementaciones, array circular simple.
- Implementar búsqueda secuencial y binaria.

SEGURIDAD:

	**Advertencia:** **En este laboratorio está prohibida la manipulación del hardware, conexiones eléctricas o de red; así como la ingestión de alimentos o bebidas.**

FUNDAMENTO TEÓRICO:

- Revisar el texto guía que está en el campus Virtual.

NORMAS EMPLEADAS:

- No aplica

RECURSOS:

- En este laboratorio cada alumno trabajará con un equipo con Windows 10.

METODOLOGÍA PARA EL DESARROLLO DE LA TAREA:

- El desarrollo del laboratorio es individual.

PROCEDIMIENTO:

Nota:

Las secciones en cursivas son demostrativas, pero sirven para que usted pueda instalar las herramientas de desarrollo en un equipo externo.

EJERCICIO DE APLICACIÓN

1.- Cola Dinámica Circular

```
class QueueDynamicCircularArray:

    def __init__(self, limit = 2):
        self.que = []
        self.limit = limit
        self.front = None
        self.rear = None
        self.size = 0
```

```
def isEmpty(self):
    return self.size <= 0

def enQueue(self, item):
    if self.size >= self.limit :
        self.resize()
    self.que.append(item)

    if self.front is None:
        self.front = self.rear = 0
    else:
        self.rear = self.size

    self.size += 1
    print('Queue after enQueue',self.que)

def deQueue(self):
    if self.size <= 0:
        print('Queue Underflow....!')
        return 0
    else:
        self.que.pop(0)
        self.size -= 1
        if self.size == 0:
            self.front = self.rear = None
        else:
            self.rear = self.size - 1
        print('Queue after deQueue', self.que)

def queueRear(self):
    if self.rear is None:
        print('Sorry, the queue is empty..!')
        raise IndexError
    return self.que[self.rear]

def queueFront(self):
    if self.front is None:
        print('Sorry, the queue is empty')
        raise IndexError
    return self.que[self.front]

def getSize(self):
    return self.size

def resize(self):
    newQue = list(self.que)
    self.limit = 2 * self.limit
    self.que = newQue
```

```
    def getQue(self):
        return self.que

    def getLimit(self):
        return self.limit


# Execution
que = QueueDynamicCircularArray()

data = ["first","second","third","fourth",
        "fifth","sixth","seventh","eighth"]

for item in data:
    print("-----------------------------")
    que.enQueue(item)
    print("Que   --> ", que.getQue())
    print("Front --> ", que.queueFront())
    print("Rear  --> ", que.queueRear())
    print("Limit --> ", que.getLimit())
    print("Size  --> ", que.getSize())

print("-----------------------------")
que.deQueue()
print("Que   --> ", que.getQue())
print("Front --> ", que.queueFront())
print("Rear  --> ", que.queueRear())
print("Limit --> ", que.getLimit())
print("Size  --> ", que.getSize())
```

Pregunta 1 : crear una cola que soporte 5 elementos , agregar 6 elementos ¿ Qué pasa ?

```
Queue after enQueue ['1']
Queue after enQueue ['1', '2']
Queue after enQueue ['1', '2', '3']
Queue after enQueue ['1', '2', '3', '4']
Queue after enQueue ['1', '2', '3', '4', '5']
Queue after enQueue ['1', '2', '3', '4', '5', '6']
```

Al ser una cola dinámica, no hay problema alguno en almacenar datos más allá de su límite, pues el tamaño de la cola se ira modificando conforme se agreguen elementos, ósea no sucede nada si agregamos 6 elementos o 1000.

Pregunta 2: crear una cola que soporte 10 elementos, agregar 5 elementos y retirar 6 elementos ¿Qué pasa

```
Queue after enQueue ['1']
Queue after enQueue ['1', '2']
Queue after enQueue ['1', '2', '3']
Queue after enQueue ['1', '2', '3', '4']
Queue after enQueue ['1', '2', '3', '4', '5']
Queue after deQueue ['2', '3', '4', '5']
Queue after deQueue ['3', '4', '5']
Queue after deQueue ['4', '5']
Queue after deQueue ['5']
Queue after deQueue []
Queue Underflow....!
```

Si se eliminaran los 5 elementos y un sexto que no existe, luego, nos mandara un mensaje de que cola esta con valores en 0 o negativos, pero no mostrara mensaje de error.

Ejercicio 1.1:

```
# Ejercicio  - Crear un cola dinamica , donde vas
# ingresar 5 valores en forma secuencial : "A","E","I","O","U"
#             ["A","E","I","O","U"]
# recuperar 2 valores de la cola
#             ["I","O","U"]
# ingresar 3 valores : "1","2","3"
#             ["I","O","U","1","2","3"]
# recuperar 1 valor de la cola
#             ["O","U","1","2","3"]
```

```
uno.enQueue("A")
uno.enQueue("E")
uno.enQueue("I")
uno.enQueue("O")
uno.enQueue("U")
uno.deQueue()
uno.deQueue()
uno.enQueue("1")
uno.enQueue("2")
uno.enQueue("3")
uno.deQueue()
```

```
Queue after enQueue ['A']
Queue after enQueue ['A', 'E']
Queue after enQueue ['A', 'E', 'I']
Queue after enQueue ['A', 'E', 'I', 'O']
Queue after enQueue ['A', 'E', 'I', 'O', 'U']
Queue after deQueue ['E', 'I', 'O', 'U']
Queue after deQueue ['I', 'O', 'U']
Queue after enQueue ['I', 'O', 'U', '1']
Queue after enQueue ['I', 'O', 'U', '1', '2']
Queue after enQueue ['I', 'O', 'U', '1', '2', '3']
Queue after deQueue ['O', 'U', '1', '2', '3']
```

Ejercicio 1.2 :

```
# Ejercicio  : obtener el valor desencolado
#              al llamar al metodo deQueue()
# rd = que.deQueue()
#
#  rd deberia almacenar el valor desencolado
```

```
Queue after enQueue ['A']
Queue after enQueue ['A', 'E']
Queue after enQueue ['A', 'E', 'I']
Queue after enQueue ['A', 'E', 'I', 'O']
Queue after enQueue ['A', 'E', 'I', 'O', 'U']
Queue after enQueue ['I', 'O', 'U', '1']
Queue after enQueue ['I', 'O', 'U', '1', '2']
Queue after enQueue ['I', 'O', 'U', '1', '2', '3']
['O', 'U', '1', '2', '3']
Elementos desencolados:
 ['A', 'E', 'I']
```

Ejercicio 1.3 :

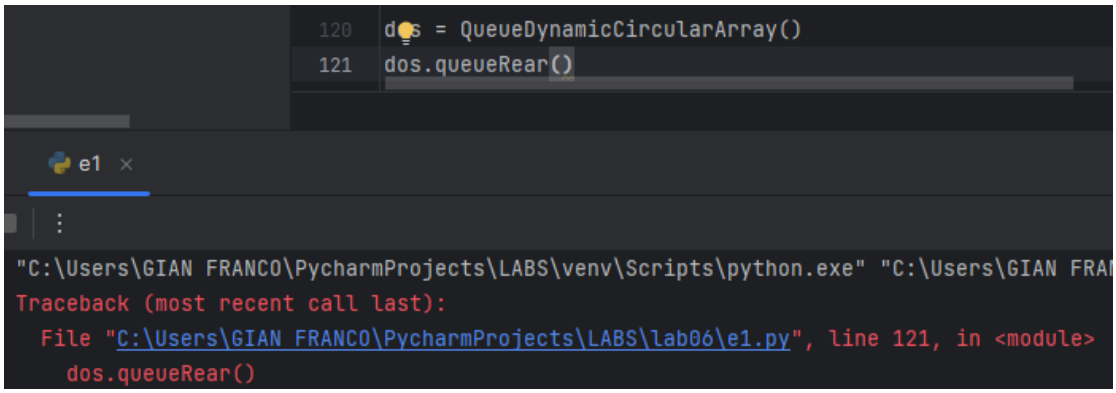
```
'''

- Verificar que al desencolar una cola vacia sale
  un error....!

- Verificar que al momento de encolar una cola llena NO sale
  un error al sobrepasar el limite de la cola !

'''
```

ERROR AL ELIMINAR ELEMENTOS DE UNA COLA VACIA



LIMITE DE LA COLA

```
def __init__(self, limit = 5):
    self.que = []
    self.limit = limit
    self.front = None
    self.rear = None
    self.size = 0
    self.deleted_items = []
```

AGREGAMOS 6 ELEMENTOS Y NO DA ERROR

```
Queue after enQueue ['1']
Queue after enQueue ['1', '2']
Queue after enQueue ['1', '2', '3']
Queue after enQueue ['1', '2', '3', '4']
Queue after enQueue ['1', '2', '3', '4', '5']
Queue after enQueue ['1', '2', '3', '4', '5', '6']
```

2.- Linked List Implementation

```
# Node of a Single Linked List
class Node:

    # Constructor
    def __init__(self, data=None):
        self.data = data
        self.next = None

    # Method for setting the data
    def setData(self, data):
        self.data = data
```

```
    # Method for getting the data
    def getData(self):
        return self.data

    # Method for setting the next
    def setNext(self, next):
        self.next = next

    # Method for getting the next
    def getNext(self):
        return self.next

    # return true if thenode point to another node
    def hasNext(self):
        return self.next != None

class QueueLinkedListsCircular:

    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0

    def enQueue(self, data):
        self.lastNode = self.rear
        self.rear = Node(data)

        if self.lastNode:
            self.lastNode.setNext(self.rear)

        if self.front is None:
            self.front = self.rear

        self.size +=1

    def deQueue(self):
        if self.front is None:
            print('Sorry, the queue is empty..!')
            raise IndexError
        result = self.front.getData()
        self.front = self.front.getNext()
        self.size -=1
        return result

    def queueRear(self):
        if self.rear is None:
            print('Sorry, the queue is empty..!')
            raise IndexError
```

```
        return self.rear.getData()

    def queueFront(self):
        if self.front is None:
            print('Sorry, the queue is empty')
            raise IndexError
        return self.front.getData()

    def getSize(self):
        return self.size

    def print( self ):
        node = self.front
        while node != None:
            print(node.getData(), end =" => ")
            node = node.getNext()
        print("NULL")

# Execution
que = QueueLinkedListsCircular()

data = ["first","second","third","fourth",
        "fifth","sixth","seventh","eighth"]

for item in data:
    print("-----------------------------")
    que.enQueue(item)
    #print("Que   --> " , que.getQue())
    que.print()
    print("Front --> " , que.queueFront())
    print("Rear  --> " , que.queueRear())
    print("Size  --> " , que.getSize())

print("-----------------------------")
que.deQueue()
#print("Que   --> " , que.getQue())
que.print()
print("Front --> " , que.queueFront())
print("Rear  --> " , que.queueRear())
print("Size  --> " , que.getSize())
```

Pregunta 1 : crear una cola que soporte 5 elementos , agregar 6 elementos ¿ Qué pasa ?

No podemos especificar concretamente el límite de la cola, por ende, no hay un máximo que defina el tamaño de la cola. Además, cuando agregamos elementos, estos se apilan en la cola, uno detrás de otro, además, la cola actualiza su tamaño cada vez que agregamos un elemento

```
1 => 2 => 3 => 4 => 5 => 6 => NULL
```

Pregunta 2: crear una cola que soporte 10 elementos, agregar 5 elementos y retirar 6 elementos ¿Qué pasa

Elimina los 5 elementos que insertamos a la cola, pero al llegar el momento de eliminar el sexto elemento de la cola, el cual no existe, nos mostrara el mensaje de que la cola ha alcanzado el valor de 0 elementos y otro mensaje de error.

```
Sorry, the queue is empty..!
```

Ejercicio 2.1

```
'''
Ejercicio  - Crear un cola con una lista enlazada , donde :

ingresar 5 valores en forma secuencial : "A","E","I","O","U"
        ["A","E","I","O","U"]
recuperar 2 valores de la cola
        ["I","O","U"]
ingresar 3 valores : "1","2","3"
        ["I","O","U","1","2","3"]
recuperar 1 valor de la cola
        ["O","U","1","2","3"]

'''
```

```
ele=QueueLinkedListsCircular()
ele.enQueue("A")
ele.enQueue("E")
ele.enQueue("I")
ele.enQueue("O")
ele.enQueue("U")
ele.print()
ele.deQueue()
ele.deQueue()
ele.print()
ele.enQueue("1")
ele.enQueue("2")
ele.enQueue("3")
ele.print()
ele.deQueue()
ele.print()
```

```
A => E => I => O => U => NULL
I => O => U => NULL
I => O => U => 1 => 2 => 3 => NULL
O => U => 1 => 2 => 3 => NULL
```

Ejercicio 2.2

```
'''
Ejercicio: Verificar si se requiere hacer modificaciones
en el método deQueue() en la clase QueueLinkedListsCircular
para poder obtener el valor desencolado.

rd = que.deQueue()

rd deberia almacenar el valor desencolado
'''
```

```
ele=QueueLinkedListsCircular()
ele.enQueue("1")
ele.enQueue("2")
ele.enQueue("3")
ele.print()
rd=ele.deQueue()
print("Valor desencolado:", rd)
```

```
1 => 2 => 3 => NULL
Valor desencolado: 1
```

Ejercicio 2.3

```
'''
- Verificar que al desencolar una cola vacia sale
  un error....!

- Verificar que al momento de encolar una cola llena NO sale
  un error al momento de incrementar los datos en la cola !

'''
```

ERROR AL DESENCOLAR COLA VACIA

```
Traceback (most recent call last):
  File "C:\Users\GIAN FRANCO\PycharmProjects\LABS\lab06\e2.py", line 105, in <module>
    ele.deQueue()
  File "C:\Users\GIAN FRANCO\PycharmProjects\LABS\lab06\e2.py", line 51, in deQueue
    raise IndexError
IndexError
```

NO ERROR AL AGREGAR ELEMENTOS A UNA COLA LLENA

```
1 => 2 => 3 => 4 => NULL
1 => 2 => 3 => 4 => 5 => NULL
```

Implementación de una cola de prioridad utilizando la clase QueueDynamicCircularArray.

La idea es asignar una prioridad a cada elemento que se inserte en la cola y asegurarse de que los elementos se eliminen de la cola en orden de prioridad, de modo que los elementos con mayor prioridad se eliminen primero. Para ello, se podrían implementar las funciones enQueue() y deQueue() de manera que tengan en cuenta la prioridad de los elementos.

Ejecuta el siguiente código

```
class PriorityQueue:
    def __init__(self, limit=10):
        self.que = []
        self.limit = limit

    def isEmpty(self):
        return len(self.que) == 0

    def enQueue(self, item, priority):
        if len(self.que) == self.limit:
            print("Queue Overflow!")
        else:
            self.que.append((item, priority))

    def deQueue(self):
        if self.isEmpty():
            print("Queue Underflow!")
            return None
        else:
            highest = 0
            for i in range(len(self.que)):
                if self.que[i][1] > self.que[highest][1]:
                    highest = i
            return self.que.pop(highest)[0]

    def queueFront(self):
        if self.isEmpty():
            return None
        else:
            highest = 0
            for i in range(len(self.que)):
                if self.que[i][1] > self.que[highest][1]:
                    highest = i
            return self.que[highest][0]
```

```
    def queueRear(self):
        if self.isEmpty():
            return None
        else:
            return self.que[-1][0]

    def getQueue(self):
        return self.que


# Ejemplo de uso

tasks = [("task1", 3), ("task2", 2), ("task3", 1), ("task4", 3), ("task5", 2)]

# Creamos una cola de prioridad
queue = PriorityQueue()

# Insertamos las tareas en la cola
for task in tasks:
    item, priority = task
    queue.enQueue(item, priority)

# Imprimimos el contenido de la cola
print("Contenido de la cola:", queue.getQueue())

# Vamos eliminando las tareas de mayor prioridad
while not queue.isEmpty():
    print("Tarea a realizar:", queue.queueFront())
    queue.deQueue()
```

Muestre una captura de su funcionamiento

```
C:\Users\Tecsup\PycharmProjects\lab06\venv\Scripts\python.exe C:\Users\Tecsup\PycharmProjects\lab06\priority.py
Contenido de la cola: [('task1', 3), ('task2', 2), ('task3', 1), ('task4', 3), ('task5', 2)]
Tarea a realizar: task1
Tarea a realizar: task4
Tarea a realizar: task2
Tarea a realizar: task5
Tarea a realizar: task3

Process finished with exit code 0
```

Prueben la cola de prioridad con diferentes conjuntos de datos para verificar si funciona correctamente y si se eliminan los elementos en el orden correcto según su prioridad. Adjunte una captura de la prueba.

```
Contenido:  [('task1', 5), ('task2', 5), ('task3', 1), ('task4', 2), ('task5', 4), ('taskó', 5), ('task7', 3), ('task8', 1), ('task9', 1), ('task10', 4)]
Tarea realizada: task1
Tarea realizada: task2
Tarea realizada: taskó
Tarea realizada: task5
Tarea realizada: task10
Tarea realizada: task7
Tarea realizada: task4
Tarea realizada: task3
Tarea realizada: task8
Tarea realizada: task9
```

Indique que hace cada una de las funciones en este código

La función enQueue() : **Método que inserta un elemento en la cola con prioridad. El primer parámetro item es el elemento que se quiere insertar y el segundo parámetro priority es su valor de prioridad.**

La función deQueue() : **Método que remueve el elemento con la mayor prioridad en la cola (el que tiene el mayor priority) y lo retorna. ¡Si la cola está vacía, el método imprime "Queue Underflow!"**

La función isEmpty() : **Método que retorna True si la cola está vacía y False en caso contrario.**

Tarea

Cree un menú dirigido hacia una empresa (cine, parque de atracciones, banco, etc), en la cual se pueda ingresar elementos a la cola, se pueda mostrar el tamaño de la cola y eliminar elementos de la cola, en caso venga una persona discapacitada, esta tendrá prioridad en su atención. (Muestre capturas de su código y de su ejecución) (De ser necesario adjunte el código en un .zip)

```python
class PriorityQueue:
    def __init__(self, limit=10):
        self.que = []
        self.limit = limit

    def isEmpty(self):
        return len(self.que) == 0

    def enQueue(self, item, priority):
        if len(self.que) == self.limit:
            print("Queue Overflow!")
        else:
            self.que.append((item, priority))

    def deQueue(self):
        if self.isEmpty():
            print("Queue Underflow!")
            return None
        else:
            highest = 0
```

```
            for i in range(len(self.que)):
                if self.que[i][1] > self.que[highest][1]:
                    highest = i
            return self.que.pop(highest)[0]

    def queueFront(self):
        if self.isEmpty():
            return None
        else:
            highest = 0
            for i in range(len(self.que)):
                if self.que[i][1] > self.que[highest][1]:
                    highest = i
            return self.que[highest][0]

    def queueRear(self):
        if self.isEmpty():
            return None
        else:
            return self.que[-1][0]

    def getQueue(self):
        return self.que

menu = PriorityQueue()

while True:
    print(" 1. Ingresar cliente a la cola")
    print(" 2. Tamaño de la cola")
    print(" 3. Atender cliente")
    print(" 4. Mostrar cola ")
    print(" 5. Salir")

    opcion = input("Ingrese una opcion: ")

    if opcion == "1":
        print("     1. Cliente normal")
        print("     2. Cliente discapacitado")
        op = input(" Seleccione una opcion (1 o 2):")

        if op == "1":
            val = input("\nElemento a agregar: ")
            menu.enQueue(val, 1)
            print("\nElemento agregado!\n")

        elif op == "2":
            val = input("\nElemento a agregar: ")
            menu.enQueue(val, 2)
            print("\nElemento agregado!\n")

        elif op >= "3":
            print("\nopcion invalida\n")

    elif opcion == "2":
        print("\nEl tamaño de la cola es de", len(menu.getQueue()),
"elementos.\n")
```

```
    elif opcion == "3":
            print("Cliente atendido: ", menu.queueFront())
            menu.deQueue()

    elif opcion == "4":
        print("Lista:\n", menu.getQueue())

    elif opcion == "5":
        print("Gracias :)")
        break

    else:
        print("\nElija una opcion disponible :)\n")
```

OBSERVACIONES:

- La implementación de una cola de prioridad puede resultar más compleja en comparación con las colas simples debido a la necesidad de utilizar una estructura de datos más avanzada, como un montículo binario.
- A diferencia de los arreglos, las listas enlazadas no permiten un acceso directo a los elementos, lo que puede resultar en una lectura más lenta de los datos.
- Para implementar una cola mediante una lista enlazada, es importante tener en cuenta cómo manejar los punteros para asegurarse de mantener la estructura de datos coherente y evitar problemas de memoria.
- Al utilizar una cola circular dinámica es importante tener en cuenta la posición de los elementos, ya que deberán moverse circularmente una vez que se llegue al último índice de la cola.
- Si no se implementa correctamente, la cola circular dinámica puede tener problemas de contención o sobreescritura de datos.

CONCLUSIONES:

- La estructura de datos de cola circular dinámica es muy útil para casos en los que se requiere una cola de tamaño finito con una capacidad determinada y que no se permite que deje de funcionar.
- La implementación de la clase de cola circular dinámica puede ser más eficiente en términos de uso de memoria en comparación con una cola mediante una matriz.
- La estructura de datos de cola circular dinámica es muy útil para casos en los que se requiere una cola de tamaño finito con una capacidad determinada y que no se permite que deje de funcionar.
- La implementación de la clase de cola circular dinámica puede ser más eficiente en términos de uso de memoria en comparación con una cola mediante una matriz.
- La estructura de datos de cola circular dinámica es muy útil para casos en los que se requiere una cola de tamaño finito con una capacidad determinada y que no se permite que deje de funcionar.
- La implementación de la clase de cola circular dinámica puede ser más eficiente en términos de uso de memoria en comparación con una cola mediante una matriz.