Ana María Durán Burgos G2

CONTROL DE LECTURA

1. Considere una pila *S* y una cola *Q*, ambas vacías en un principio. Ilustre el paso a paso de las siguientes operaciones:
   - $S.push(2) \rightarrow S.push(8) \rightarrow S.push(11) \rightarrow S.pop() \rightarrow S.push(-3) \rightarrow S.push(7) \rightarrow S.pop() \rightarrow S.pop()$

s = [ ] s = [2] s = [2, 8] s = [2, 8, 11] s = [2, 8] s = [2, 8, -3] s = [2, 8, -3, 7] s = [2, 8]

- $Q.enqueue(4) \rightarrow Q.enqueue(17) \rightarrow Q.enqueue(20) \rightarrow Q.enqueue(6) \rightarrow Q.dequeue(\square) \rightarrow Q.dequeue(\square) \rightarrow Q.enqueue(-5) \rightarrow Q.dequeue(\square)$

Q = [ ] Q = [4] Q = [4, 17] Q = [4, 17, 20] Q = [4, 17, 20, 6] Q = [17, 20, 6] Q = [20, 6]

Q = [20, 6, -5] Q = [6, -5]

2. Desarrolle un algoritmo que identifique si una cadena de texto contiene una lista de paréntesis correctamente anidados y balanceados, por ejemplo:
   - Correcto: [{((())())()}]{[]}
   - Correcto: (({}))[]
   - Incorrecto:)([])(
   - Incorrecto: (){[]([]

parentesis.py

```
def verificarParentesis(cadena):
    lista = []
    for i in cadena:
        if i in abrir_lista:
            lista.append(i)
        elif i in cerrar_lista:
            pos = cerrar_lista.index(i)
            if ((len(lista) > 0) and
                    (abrir_lista[pos] == lista[len(lista) - 1])):
                lista.pop()
            else:
                return "Incorrecto"
    if len(lista) == 0:
        return "Correcto"
    else:
        return "Incorrecto"


cadena = "[{((())())}]{[]}"
print(cadena, "-", verificarParentesis(cadena))

cadena = "(({}))[]"
print(cadena, "-", verificarParentesis(cadena))

cadena = ")([])("
```

```
Run: parentesis
C:\Users\pc\PycharmProjects\AYED\venv\Scripts\python.exe C:/Users/pc/PycharmProjects/AYED/Contr
[{((())())}]{[]} - Correcto
(({}))[] - Correcto
)([])( - Incorrecto
){[]([] - Incorrecto

Process finished with exit code 0
```

3. Diseñe una función para invertir la dirección de una lista enlazada simple, es decir, una función que invierta todos los punteros entre los elementos de la lista. El algoritmo debe tener complejidad lineal $O(n)$

Ana María Durán Burgos G2

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:

    def __init__(self):
        self.head = None

    def reverse(self):
        prev = None
        current = self.head
        while current is not None:
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev

    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def printList(self):
        temp = self.head
        while (temp):
            print(temp.data, end=" ")
            temp = temp.next
```

```python
llist = LinkedList()
llist.push(1)
llist.push(2)
llist.push(3)
llist.push(4)
llist.push(5)
llist.push(6)
llist.push(7)
llist.push(8)

print("Lista Enlazada")
llist.printList()
llist.reverse()
print("\nLista Enlazada Invertida")
llist.printList()
```

```
Lista Enlazada
8 7 6 5 4 3 2 1
Lista Enlazada Invertida
1 2 3 4 5 6 7 8
Process finished with exit code 0
```

4. Modifique el código de la lista enlazada para que sea una doble lista enlazada, e implemente la siguientes funciones:
   - Insertar un nuevo elemento
   - Eliminar un elemento dado su valor (considere el caso de borrar la cabeza de la lista)
   - Eliminar los elementos duplicados
   - Unir dos listas

```
def append(self, value):
    # Insertar un nuevo elemento
    new_node = Node(value)
    if len(self) == 0:
        self.head = new_node
        self.setTail(new_node)
    else:
        current_tail = self.tail
        current_tail.setNext(new_node)
        new_node.setPrev(current_tail)
        self.setTail(new_node)
    self.len = self.len + 1
```

```
def merge(self, list_b):
    # Unir dos listas
    if self.isEmpty():
        return list_b
    if list_b.isEmpty():
        return self
    self.tail.setNext(list_b.getHead())
    self.setTail(list_b.getTail())
```

```
def delete(self, value):
    # Eliminar un elemento dado su valor
    value_node = self.search(value)
    if value_node is not None:
        if len(self) == 1: # Soy el único                #Si es
            self.head, self.tail = None, None
        else:
            if value_node == self.getHead():            #Si el
                self.head = value_node.getNext()
                self.head.setPrev(None)
            else:
                #Buscar el previo a value_node
                prev = value_node.getPrev()
                if value_node == self.getTail():        #Si el
                    self.setTail(prev)
                else:
                    nxt = value_node.getNext()
                    prev.setNext(nxt)  #Si el valor es cualqui
                    if nxt is not None:
                        nxt.setPrev(prev)               # El a

        value_node.clear()                              #borra
        self.len -= 1                                   #cambi
    else:
        raise Exception("Element not found.")
```

```
def deleteDuplicates(self):
    # Eliminar elementos duplicados
    s = []
    node = self.getHead()
    while node is not None:
        if node.getValue() not in s:
            s.append(node.getValue())
            node = node.getNext()
        else:
            self.delete(node.getValue())
            node = node.getNext()
    return self
```