Facultad de Ingeniería
UNER

Trabajo Práctico N° 1

ALUMNOS:

Biasoli, Ana Inés

Taborda, Veronica

PROBLEMA 2:

Se debe implementar un TAD `ListaDobleEnlazada` para almacenar elementos comparables. La lista debe incluir métodos para verificar si está vacía, obtener su longitud, agregar o insertar elementos, extraer elementos (con complejidad O(1) en los extremos), copiar, invertir, concatenar y sumar listas. La clase debe ser eficiente en tiempo y memoria, evitando el uso de estructuras auxiliares como listas de Python. Además, se debe graficar el tiempo de ejecución de los métodos `len`, `copiar` e `invertir` en función de la cantidad de elementos y deducir su complejidad. La implementación debe pasar los tests dados por la cátedra.

Un nodo doblemente enlazado es una estructura que forma parte de una **lista doblemente enlazada**. Cada nodo guarda:

1. **Un dato** (el valor que queremos almacenar, como un número, texto, etc.).
2. **Un puntero (referencia) al nodo anterior**.
3. **Un puntero (referencia) al nodo siguiente**.

Esto le permite a la lista recorrer elementos **hacia adelante y hacia atrás**, a diferencia de una lista simplemente enlazada, que solo puede avanzar.

Módulo nodo
En principio, creamos la clase nodo

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato            # El valor almacenado (int, string, float, etc.)
        self.siguiente = None       # Referencia al nodo anterior
        self.anterior = None        # Referencia al nodo siguiente
```

Luego creamos una clase denominada ListaDobleEnlazada

```
from Nodo import Nodo

class ListaDoblementeEnlazada:
    def __init__(self):
        self.cabeza = None #referencia al primer nodo de la lista (None
al principio, porque la lista está vacía).
        self.cola = None   #referencia al último nodo de la lista
(también None inicialmente).
        self.tamanio = 0   #contador de elementos en la lista (inicia
en 0)
      def lista_vacia(self):
```

```
        return self.tamanio == 0

    def __len__(self):
        return self.tamanio  # Devolvemos el tamaño actual de la lista
```

Se crea un nuevo nodo con el dato que se quiere insertar. Este nodo es una instancia de la clase Nodo.

```
    def agregar__al__inicio(self, dato):
        nuevo_nodo = Nodo(dato)
        if self.cabeza is None:
            self.cabeza = nuevo_nodo
            self.cola = nuevo_nodo  #Si la lista está vacía (cabeza es None), significa que este será el único nodo de la lista. Por lo tanto, tanto la cabeza como la cola apuntan al mismo nodo.

        else:
            nuevo_nodo.siguiente = self.cabeza
            self.cabeza.anterior = nuevo_nodo
            self.cabeza = nuevo_nodo
        self.tamanio += 1
```

Si la lista **ya tiene elementos**, entonces:

- El nuevo nodo apunta hacia el antiguo primer nodo (nuevo_nodo.siguiente = self.cabeza).
- El antiguo primer nodo apunta hacia atrás al nuevo (self.cabeza.anterior = nuevo_nodo).
- Luego, se actualiza la cabeza de la lista para que sea el nuevo nodo.

Finalmente se incrementa el contador de elementos de la lista

```
    def agregar__al__final(self, dato):
        nuevo_nodo = Nodo(dato)
        if self.lista_vacia():
            self.cabeza = nuevo_nodo
            self.cola = nuevo_nodo
        else:
```

```
            nuevo_nodo.anterior = self.cola
            self.cola.siguiente = nuevo_nodo
            self.cola = nuevo_nodo
        self.tamanio += 1


    def insertar(self,dato,posicion): #inserta un nodo en una posicion
especifica
         if posicion < 0 or posicion > self.tamanio:
            raise Exception("Posición inválida") #controla que posicion
este dentro de los limites validos
           if posicion == 0:
            self.agregar__al__inicio(dato)
            return

        elif posicion == self.tamanio:
            self.agregar__al__final(dato)
            return

        nuevo_nodo=Nodo(dato)
        nodo_actual = self.cabeza
        for _ in range(posicion - 1):  # Recorremos hasta la posición
anterior
            nodo_actual = nodo_actual.siguiente

        # Ajuste de punteros para insertar en medio
        nuevo_nodo.siguiente = nodo_actual.siguiente
        nuevo_nodo.anterior = nodo_actual

        if nodo_actual.siguiente:  # Si existe un nodo después, lo
enlazamos correctamente
            nodo_actual.siguiente.anterior = nuevo_nodo
        nodo_actual.siguiente = nuevo_nodo
        self.tamanio += 1  # Aumentamos el tamaño de la lista

    def __iter__(self):

 self._nodo_actual = self.cabeza  # Iniciamos desde la cabeza
        return self

    def __next__(self):
        if self._nodo_actual is None:
```

```
            raise StopIteration  # Detener iteración cuando ya no hay
nodos
        dato = self._nodo_actual.dato  # Devolver el dato, no el nodo
        self._nodo_actual = self._nodo_actual.siguiente  # Avanzar al
siguiente nodo
        return dato


    def __add__(self, otra_lista):
        if not isinstance(otra_lista, ListaDoblementeEnlazada):
            raise TypeError("Se requiere otra instancia de
ListaDoblementeEnlazada")
        nueva_lista = ListaDoblementeEnlazada()
        for lista in (self, otra_lista):
            nodo = lista.cabeza
            while nodo:
                nueva_lista.agregar__al__final(nodo.dato)
                nodo = nodo.siguiente
        return nueva_lista


    def copiar(self):
        lista_copiada = ListaDoblementeEnlazada()
        nodo_actual = self.cabeza
        while nodo_actual:
            lista_copiada.agregar__al__final(nodo_actual.dato)
            nodo_actual = nodo_actual.siguiente
        return lista_copiada


    def concatenar(self, otra_lista):
        if otra_lista.lista_vacia():
            return

        # Si la lista original está vacía, simplemente asignamos el
primer nodo de la otra lista
        if self.lista_vacia():
            self.cabeza = otra_lista.cabeza
            self.cola = otra_lista.cola
        else:
            # Si ambas listas tienen elementos, conectamos el último
nodo de la lista original
            # al primer nodo de la otra lista
```

```
            self.cola.siguiente = otra_lista.cabeza
            otra_lista.cabeza.anterior = self.cola
            self.cola = otra_lista.cola  # Actualizamos el último nodo
de la lista original
        self.tamanio += otra_lista.tamanio  # Actualizamos el tamaño
total de la lista

    # Método para extraer un elemento de una posición dada
    def extraer(self, posicion):
        if self.lista_vacía():
            raise Exception("La lista está vacía")

        if posicion < 0 or posicion >= self.tamanio:
            raise Exception("Posición fuera de rango")
        if posicion == -1:
            posicion = self.tamanio - 1

        # Si la posición es 0 (extraer el primer nodo)
        if posicion == 0:
            dato_extraido = self.cabeza.dato
            if self.cabeza.siguiente:
                self.cabeza = self.cabeza.siguiente
                self.cabeza.anterior = None
            else:
                self.cabeza = None
                self.cola = None
            self.tamanio -= 1
            return dato_extraido

        # Si la posición es la última (extraer el último nodo)
        if posicion == self.tamanio - 1:
            dato_extraido = self.cola.dato
            if self.cola.anterior:
                self.cola = self.cola.anterior
                self.cola.siguiente = None
            else:
                self.cabeza = None
                self.cola = None
            self.tamanio -= 1
            return dato_extraido

        # Extraer un nodo intermedio
        actual = self.cabeza
```

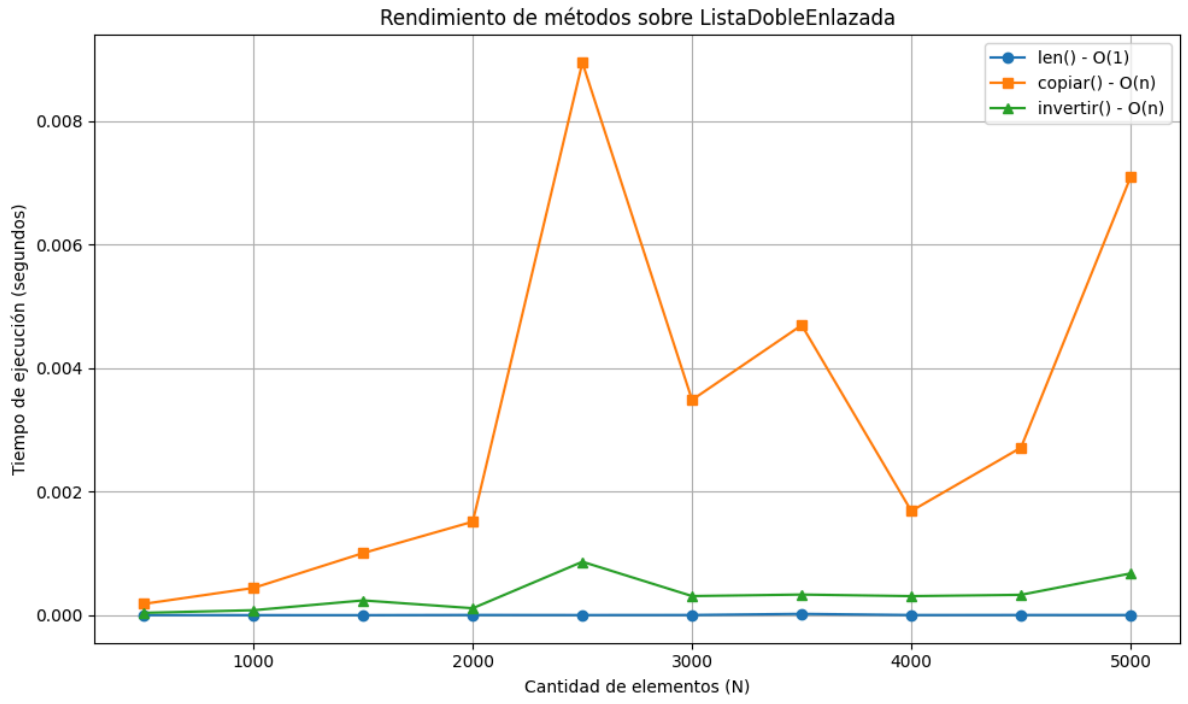
```
        for i in range(posicion):
            actual = actual.siguiente

        dato_extraido = actual.dato
        actual.anterior.siguiente = actual.siguiente
        actual.siguiente.anterior = actual.anterior
        self.tamanio -= 1
        return dato_extraido

    def invertir(self):
        if self.cabeza is None or self.cabeza == self.cola:
            return

        actual = self.cabeza
        while actual:
            temp = actual.siguiente
            actual.siguiente = actual.anterior
            actual.anterior = temp
            actual = temp  # avanzar al siguiente nodo (el original
siguiente)

        self.cabeza, self.cola = self.cola, self.cabeza
```

Las funciones utilizadas y otorgadas por la cátedra son las siguientes:

Método	**¿Qué hace?**
`esta_vacia()`	Dice si está vacía.
`__len__()`	Devuelve cuántos elementos hay (como `len(lista)`).
`agregar_al_inicio(item)`	Agrega un dato al **principio**.
`agregar_al_final(item)`	Agrega un dato al **final**.

`insertar(item, posicion)`	Inserta un dato en una posición dada.
`extraer(posicion)`	Elimina y devuelve el dato en una posición dada.
`copiar()`	Devuelve una **copia nueva** de la lista.
`invertir()`	Invierte el orden de los elementos.
`concatenar(otra_lista)`	Junta otra lista al final de esta.
`__add__(otra_lista)`	Lo mismo que concatenar, pero usando `+`.

La gráfica del problema 2 en una de sus compilaciones es la siguiente:



Observamos que:

- la curva $len() - O(1)$ es casi plana, cercana a cero. Esto podríamos interpretarlo como que el tiempo de ejecución no varía de manera significante con el tamaño de la lista, lo que indica que este método es constante en el tiempo, lo que es típico en casos de que la lista mantenga un contador de elementos actualizado, sin necesidad de recorrerla para contarlos.
- la curva del método $copiar() - O(n)$ aumenta de manera irregular pero general con N, y muestra picos y caídas que podrían estar relacionadas al sistema; lo que se interpretaría como una complejidad lineal $O(n)$, lo que es consistente con tener que recorrer todos los nodos para copiarlos.
- la curva del método $invertir() - O(n)$ también aumenta con N pero de una manera menos "ruidosa" que copiar, por lo que podemos decir que sigue la tendencia general de crecimiento con N. Por lo que podemos concluir que este metodo tambien parece tener complejidad lineal $O(n)$, porque necesita recorrer la lista para invertir punteros

PROBLEMA 3

El juego de cartas "Guerra" es un juego por turnos entre dos jugadores que buscan quedarse con todas las cartas. Cada jugador comienza con 26 cartas tomadas de un mazo común de 52. En cada turno, ambos revelan la primera carta de su mazo; quien tenga la más alta se lleva ambas. Si hay empate, se produce una "guerra": colocan tres cartas boca abajo y una más boca arriba para desempatar; el ganador se lleva todas. El proceso se repite hasta que un jugador gana todas las cartas o el otro se queda sin cartas.

Se proporciona el código del juego, pero falta implementar la clase `Mazo`, que debe usar una lista doblemente enlazada para almacenar cartas y soportar ciertas operaciones. También debe lanzarse la excepción `DequeEmptyError` al intentar extraer una carta de un mazo vacío. Los tests del juego y de la clase deben pasar con una implementación correcta.

```
"""
Simulación del juego de cartas Guerra en un solo archivo.
"""

import random

# ============================
# Clase Carta
# ============================

class Carta:
    valores = ['2','3','4','5','6','7','8','9','10','J','Q','K','A']
    valor_numerico = {v: i for i, v in enumerate(valores, start=2)}
```

```
    def __init__(self, valor, palo):
        self.valor = valor
        self.palo = palo

    def __str__(self):
        return f"{self.valor}{self.palo}"

    def __gt__(self, otra):
        return Carta.valor_numerico[self.valor] >
Carta.valor_numerico[otra.valor]

    def __lt__(self, otra):
        return Carta.valor_numerico[self.valor] <
Carta.valor_numerico[otra.valor]

    def __eq__(self, otra):
        return Carta.valor_numerico[self.valor] ==
Carta.valor_numerico[otra.valor]

# ==========================
# Excepción personalizada
# ==========================

class DequeEmptyError(Exception):
    pass

# ==========================
# Clase Nodo para la Lista Doblemente Enlazada
# ==========================

class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.anterior = None
        self.siguiente = None

# ==========================
# Clase Lista Doblemente Enlazada
# ==========================

class ListaDoblementeEnlazada:
    def __init__(self):
        self.primero = None
```

```
        self.ultimo = None
        self._tamanio = 0

    def insertar_inicio(self, dato):
        nuevo = Nodo(dato)
        if not self.primero:
            self.primero = self.ultimo = nuevo
        else:
            nuevo.siguiente = self.primero
            self.primero.anterior = nuevo
            self.primero = nuevo
        self._tamanio += 1

    def insertar_final(self, dato):
        nuevo = Nodo(dato)
        if not self.ultimo:
            self.primero = self.ultimo = nuevo
        else:
            nuevo.anterior = self.ultimo
            self.ultimo.siguiente = nuevo
            self.ultimo = nuevo
        self._tamanio += 1

    def eliminar_inicio(self):
        if not self.primero:
            raise DequeEmptyError("La lista está vacía")
        dato = self.primero.dato
        self.primero = self.primero.siguiente
        if self.primero:
            self.primero.anterior = None
        else:
            self.ultimo = None
        self._tamanio -= 1
        return dato

    def __len__(self):
        return self._tamanio

    def __str__(self):
        elementos = []
        actual = self.primero
        while actual:
            elementos.append(str(actual.dato))
```

```
            actual = actual.siguiente
        return ' '.join(elementos)

# ========================
# Clase Mazo
# ========================

class Mazo:
    def __init__(self):
        self.cartas = ListaDoblementeEnlazada()

    def poner_carta_arriba(self, carta):
        self.cartas.insertar_inicio(carta)

    def poner_carta_abajo(self, carta):
        self.cartas.insertar_final(carta)

    def sacar_carta_arriba(self, mostrar=False):
        if len(self.cartas) == 0:
            raise DequeEmptyError("El mazo está vacío")
        carta = self.cartas.eliminar_inicio()
        if mostrar:
            print(f"    saca {carta}")
        return carta

    def __len__(self):
        return len(self.cartas)

    def __str__(self):
        return str(self.cartas)

# ========================
# Clase JuegoGuerra
# ========================

N_TURNOS = 10000

class JuegoGuerra:

    valores = ['2','3','4','5','6','7','8','9','10','J','Q','K','A']
    palos = ['♠', '♥', '♦', '♣']

    def __init__(self, random_seed = 0):
```

```
        self._mazo_inicial = Mazo()
        self.mazo_1 = Mazo()
        self.mazo_2 = Mazo()
        self._guerra = False
        self._ganador = ''
        self.empate = False
        self._turno = 0
        self._cartas_en_la_mesa = []
        self._seed = random_seed

    @property
    def turnos_jugados(self):
        if self.empate:
            return N_TURNOS
        return self._turno + 1

    @property
    def ganador(self):
        return self._ganador

    def armar_mazo_inicial(self):
        random.seed(self._seed)
        cartas = [Carta(valor, palo) for valor in JuegoGuerra.valores
                  for palo in JuegoGuerra.palos]
        random.shuffle(cartas)
        for carta in cartas:
            self._mazo_inicial.poner_carta_arriba(carta)
        return self._mazo_inicial

    def repartir_cartas(self):
        while len(self._mazo_inicial):
            carta_1 = self._mazo_inicial.sacar_carta_arriba()
            self.mazo_1.poner_carta_arriba(carta_1)
            carta_2 = self._mazo_inicial.sacar_carta_arriba()
            self.mazo_2.poner_carta_arriba(carta_2)
        return self.mazo_1, self.mazo_2

    def iniciar_juego(self, ver_partida=True):
        self.armar_mazo_inicial()
        self.repartir_cartas()
        self._cartas_en_la_mesa = []
        self._turno = 0
```

```
        while len(self.mazo_1) and len(self.mazo_2) and self._turno !=
N_TURNOS:
            try:
                if self._guerra:
                    for _ in range(3):

self._cartas_en_la_mesa.append(self.mazo_1.sacar_carta_arriba())

self._cartas_en_la_mesa.append(self.mazo_2.sacar_carta_arriba())


self._cartas_en_la_mesa.append(self.mazo_1.sacar_carta_arriba(mostrar=T
rue))

self._cartas_en_la_mesa.append(self.mazo_2.sacar_carta_arriba(mostrar=T
rue))

            except DequeEmptyError:
                if len(self.mazo_1):
                    self._ganador = 'jugador 1'
                else:
                    self._ganador = 'jugador 2'
                self._guerra = False
                if ver_partida:
                    print(f'***** {self._ganador} gana la partida
*****')

            else:
                if ver_partida:
                    self.mostrar_juego()

                if self._cartas_en_la_mesa[-2] >
self._cartas_en_la_mesa[-1]:
                    for carta in self._cartas_en_la_mesa:
                        self.mazo_1.poner_carta_abajo(carta)
                    self._cartas_en_la_mesa = []
                    self._guerra = False
                    if len(self.mazo_2):
                        self._turno += 1
                elif self._cartas_en_la_mesa[-1] >
self._cartas_en_la_mesa[-2]:
                    for carta in self._cartas_en_la_mesa:
                        self.mazo_2.poner_carta_abajo(carta)
```

```
                    self._cartas_en_la_mesa = []
                    self._guerra = False
                    if len(self.mazo_1):
                        self._turno += 1
                else:
                    self._guerra = True
                    if ver_partida:
                        print('**** Guerra!! ****')

            finally:
                if self._turno == N_TURNOS:
                    self.empate = True
                    if ver_partida:
                        print('***** Empate *****')

        if self._turno != N_TURNOS and not self._ganador:
            if len(self.mazo_1):
                self._ganador = 'jugador 1'
            else:
                self._ganador = 'jugador 2'
            if ver_partida:
                print(f'***** {self._ganador} gana la partida *****')

    def mostrar_juego(self):
        print(f"Turno: {self._turno+1}")
        print('jugador 1:')
        print(self.mazo_1)
        print()
        print('               ', end='')
        for carta in self._cartas_en_la_mesa:
            print(carta, end=' ')
        print('\n')
        print('jugador 2:')
        print(self.mazo_2)
        print()
        print('-----------------------------------')
        if self._ganador:
             print(f'***** {self._ganador} gana la partida *****')

# ==========================
# Ejecutar el juego
# ==========================
```

```
if __name__ == "__main__":
    n = random.randint(0, 1000)
    juego = JuegoGuerra(random_seed=n)
    juego.iniciar_juego()
    print(f"Semilla usada: {n}")
```

Aqui hacemos la clase mazo

```
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))
from LDE import ListaDoblementeEnlazada  # LDE.py contiene la clase
ListaDobleEnlazada
from carta import Carta  # Importa la clase Carta

#from modules.LDE import ListaDobleEnlazada  # Importa la clase
ListaDobleEnlazada
import random

class DequeEmptyError(Exception):
    pass

# ========================
# Clase Mazo
# ========================

class Mazo:
    def __init__(self):
        self.cartas = ListaDoblementeEnlazada() #Crea un mazo que
internamente usa una lista doblemente enlazada para guardar las cartas


    def poner_carta_arriba(self, carta):
        self.cartas.insertar_inicio(carta)


    def poner_carta_abajo(self, carta):
        self.cartas.insertar_final(carta)
```

Permite agregar una carta al **inicio** o al **final** del mazo, simulando que la carta se coloca arriba o abajo del mazo.

```
    def sacar_carta_arriba(self, mostrar=False):
        if len(self.cartas) == 0:
            raise DequeEmptyError("El mazo está vacío")
        carta = self.cartas.eliminar_inicio()
        if mostrar:
            print(f"saca {carta}")
        return carta
```

Si el mazo está vacío, lanza un error. Si no, elimina y retorna la carta de arriba. Si `mostrar=True`, imprime qué carta se sacó.

```
    def __len__(self):
        return len(self.cartas)


    def __str__(self):
        return str(self.cartas)
```

`__len__`: permite usar `len(mazo)` para obtener la cantidad de cartas.

`__str__`: devuelve una representación en texto del mazo.

Finalmente la terminal nos muestra lo siguiente:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

------------------------------------
saca -X
saca -X
Turno: 37
jugador 1:
-X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X -X

              -X -X

jugador 2:
-X -X -X

------------------------------------
                      **** Guerra!! ****
saca -X
             ***** jugador 1 gana la partida*****
465
PS C:\Users\Usuario\Desktop\Vero\FACULTAD\Algoritmos\TP1 TAd\AyED2025c1-Biasoli-Taborda>
```

Como podemos observar, al compilar el código se ejecuta correctamente.