Facultad de Ingeniería
UNER

Trabajo Práctico N°1:
Aplicaciones de TADs

Asignatura: Algoritmos y Estructuras de Datos

2° cuatrimestre - Año 2023

Fecha de entrega: 18/09/2023

Integrantes:

- **Claure Endara, Jorge**
- **Diaz Pinel, Enzo**

Ejercicio 1

Implementar el **TAD Lista doblemente enlazada** utilizando la **estructura de datos (ED)** de nodos doblemente enlazados para almacenar elementos de cualquier tipo. Utilizar un TAD Nodo para gestionar la estructura de datos interna. La lista debe permitir las siguientes operaciones:

- Crear una lista vacía (implementar inicializador).
- Copiar la lista (implementar método “copiar”) para generar una copia profunda. La operación debe tener orden de complejidad O(n).
- Agregar un elemento en cualquier posición de la lista. La posición debe ser válida.
- Eliminar un elemento en cualquier posición de la lista. La posición debe ser válida. Para la eliminación en los extremos de la lista, la operación debe tener orden de complejidad O(1).
- Invertir el orden de los elementos de la lista.
- Iterar sobre la lista.
- Ordenar de “menor a mayor” los elementos de la lista.
- Concatenar dos listas con el operador ‘+’ (suma).

Su clase ListaDobleEnlazada debe pasar el programa de testing provisto por la cátedra.

Aclaraciones:

- No utilice almacenamiento adicional innecesario ni funciones de la biblioteca estándar de python o de terceros en la implementación de los métodos. La implementación debe ser eficiente en relación al uso de la memoria de la computadora. Ejemplo: no se puede copiar el contenido de la Lista doblemente enlazada a una lista de Python, y viceversa, para implementar las operaciones del TAD.
- El algoritmo de ordenamiento del método “ordenar” de la Lista debe tener la eficiencia del algoritmo de ordenamiento por inserción, o mejor.

En el problema 1, agregar al proyecto un análisis del orden de complejidad del algoritmo de ordenamiento implementado y realizar una gráfica del orden de complejidad en función del número de elementos (con valores aleatorios).

Como resultados de este problema, podemos decir que:

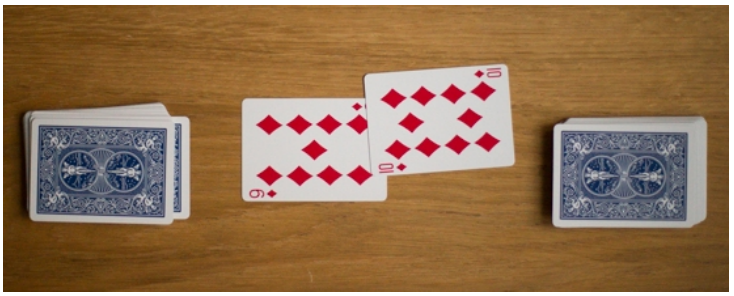
- Pudimos resolver todas las operaciones propuestas.
- Decidimos implementar el algoritmo de ordenamiento por inserción para realizar dicha operación debido a la limitación de la eficiencia.
- Nuestra clase ListaDobleEnlazada pudo pasar satisfactoriamente el testing.
- La gráfica del orden de complejidad quedó de la siguiente manera:

https://colab.research.google.com/drive/13gdqeV90zlXf0yWCcgzKpidiqm494DuV?usp=sharing

Ejercicio 2

El juego de cartas **“Guerra”** es un juego de azar donde el objetivo es ganar todas las cartas. El juego consiste en las siguientes etapas:

- Inicialmente, un mazo de cartas común (52 cartas) se reparte entre 2 jugadores de forma que ambos jugadores tengan 26 cartas cada uno (un mazo de 26 cartas por jugador). Los jugadores no pueden ver sus cartas ni las del oponente.
- El juego se realiza por turnos: en cada turno, ambos jugadores deben colocar sus cartas boca abajo sobre la mesa. El jugador 1 voltea la primera carta del mazo en el centro de la mesa. El jugador 2 hace lo mismo con la primera carta de su mazo.



- El jugador con la carta más alta gana el turno y se queda con ambas cartas para añadirlas al final de su mazo en el mismo orden en las cuales fueron reveladas. El orden de las cartas de menor a mayor es: 2,3,4,5,6,7,8,9,10,J,Q,K,A (no se tiene en cuenta el palo de la baraja).
- Ambos jugadores vuelven a voltear la siguiente carta en su mazo y se repite el proceso. Esto continúa hasta que uno de los jugadores gana todas las cartas.

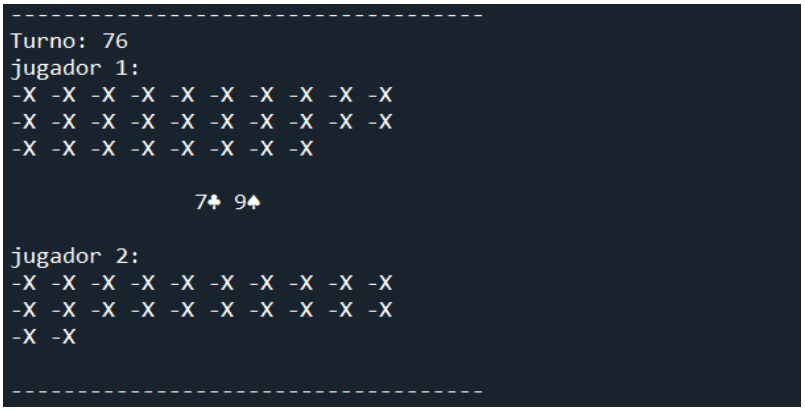
Guerra:
Cuando las cartas volteadas en cualquier turno tienen el mismo número, se entra en “Guerra”. Cuando esto sucede, las cartas jugadas al iniciar el turno permanecen en la mesa a las cuales los jugadores agregan de forma alternada (jugador 1, jugador 2, jugador 1, ….) 3 cartas más de sus respectivos mazos boca abajo. En este momento, quedan 8 cartas “en la mesa”: dos boca arriba (las que iniciaron la guerra) y 6 boca abajo (que representan un botín de guerra). Después, cada jugador agrega una carta más boca arriba; ahora van 10 cartas “en la mesa”. Se comparan estas 2 últimas cartas y el jugador con el mayor valor conserva las 10 cartas de la ronda y las agrega al final de su mazo.
Las cartas añadidas al final del mazo, siempre se agregan en el orden exacto en el que fueron jugadas (jugador 1 primera carta, jugador 2 primera carta, jugador 1 segunda carta, jugador 2 segunda carta, etc.).
Si estando en una “Guerra” se vuelve a entrar en “Guerra”, entonces, cada jugador agrega de forma alternada 3 cartas boca abajo (que se suman a las 10 cartas que ya había previamente en la mesa) y una boca arriba al final. Esta vez el jugador con la carta más alta se quedará con 18 cartas (6 boca arriba y 12 boca abajo). En el caso poco probable de que se vuelva a entrar en “Guerra”, el proceso se repite hasta que uno gane el turno o hasta que uno de los jugadores no tenga suficientes cartas para continuar.

Si un jugador se queda sin cartas para terminar el turno, pierde el juego.

Si se llega a un número determinado de turnos previamente acordado y la partida no ha finalizado, se considera un empate.

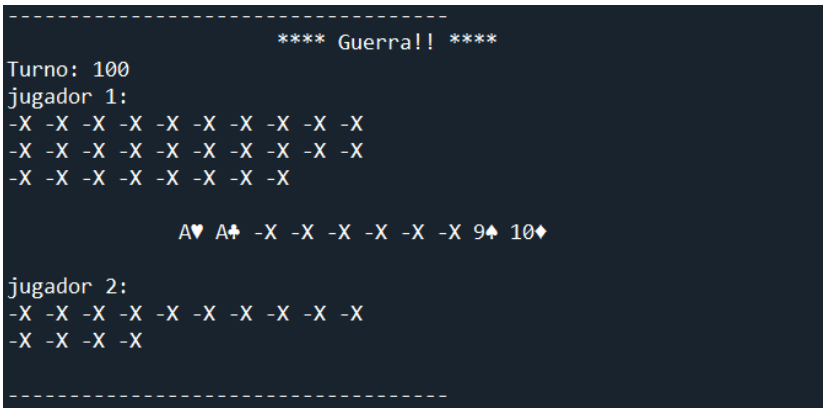
Se desea simular este juego (para 2 jugadores) para predecir en qué turno termina y cual jugador gana la partida (‘jugador 1’ o ‘jugador 2’) dado un estado inicial de los mazos de cartas de cada jugador (determinado por una semilla generadora de números aleatorios).

La simulación debe mostrar la evolución de la partida por consola de la siguiente manera:

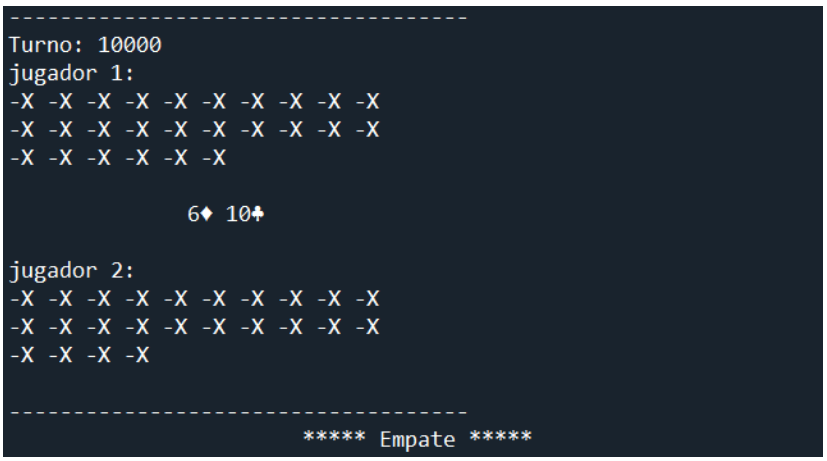


En cada turno debe mostrar las cartas de ambos jugadores. Las cartas en los mazos se deben mostrar *‘boca abajo’,* las cartas en la mesa se deben mostrar *‘boca arriba’* o *‘boca abajo’*, según corresponda.

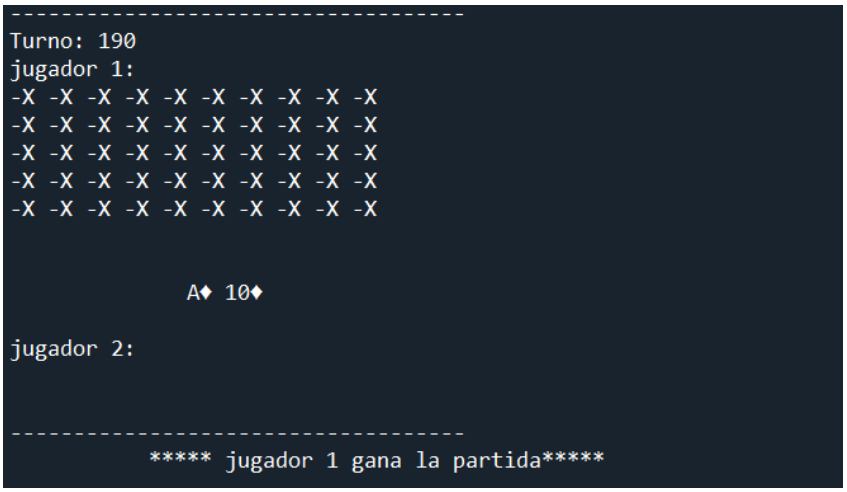
Se debe mostrar un mensaje en cada turno donde se entra en *Guerra,* las cartas que se agregan en la mesa deben mostrarse con sus estados correspondientes *‘boca abajo’* y *‘boca arriba’*



Dependiendo del estado inicial de los mazos de cada jugador, el juego puede volverse infinito, la simulación debe identificar el ganador de la partida en un máximo de 10000 turnos, en caso contrario declare un empate.



Si hubo ganador, mostrar mediante un mensaje al final de la partida.



Implemente una clase ‘JuegoGuerra’ que simule el juego de cartas, su clase debe pasar el programa de testing provisto por la cátedra. Seleccione la estructura de datos adecuada para representar los mazos de los jugadores (pila, cola, cola doble, etc.) y utilice la lista doblemente enlazada del problema anterior para implementar dicha estructura. Implemente las clases adicionales que considere necesarias para representar el juego.

Utilice las siguientes listas para representar los valores y palos de la baraja:

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

Enlace a repo inicial para JuegoDeGuerra (con su código de prueba): enlace

Aclaración:

- **No modifique el programa de prueba de ninguna manera, la firma de los métodos de su clase deben corresponderse con los del test para poder pasarlo.**

Como resultados de este problema, podemos decir que:

- Para guiarnos de una mejor manera a la hora de codificar el programa, armamos un diagrama en bloque disponible aquí: https://drive.google.com/file/d/1bgaBgCbQS6MzbLM73XyXk4hei1t4JCn5/view?usp=sharing
- Pudimos resolver todas las operaciones propuestas.
- Nuestra clase Guerra pasó satisfactoriamente el testing.

Ejercicio 3

Escribir un programa que genere un archivo de texto con un gran número de líneas tal que el archivo tenga un tamaño mayor o igual a 100 megabytes. En cada línea del archivo debe haber un único número entero. Los números deben estar desordenados en este archivo. Ejemplo: con números de 20 cifras y unas 5 millones de líneas se logra un archivo de unos 100 megabytes. Puede utilizarse el código en el siguiente enlace como punto de partida.

Luego, codifique el algoritmo de ordenamiento externo "mezcla directa" tomando bloques de B claves en cada lectura. B es un número entero positivo menor al número de datos a ordenar que representa la cantidad de claves por bloque leído. Típicamente B es un valor tal que el número de claves por bloque leído cabe cómodamente en memoria principal. Los bloques de B claves se ordenan luego de cada lectura de bloque, o se ordenan, antes de iniciar el proceso de mezcla directa, esto es opcional.

Ejemplo de un archivo donde N es 8 y B es 3. Notar que, en este caso, cada bloque ha sido ordenado previo al inicio del proceso de ordenamiento externo, y que el último bloque queda incompleto:

4	58	90	15	22	30	7	9

Posteriormente, aplique el algoritmo indicado al archivo previamente creado para ordenarlo de menor a mayor de manera tal que la primera línea del archivo resultante contenga el menor número entero y la última el mayor.

Finalmente, escribir una prueba que verifique el funcionamiento del algoritmo; esto es, que el archivo resultante posea el mismo tamaño (en bytes) que el archivo original y que los datos en su interior están realmente ordenados de menor a mayor luego de aplicar el algoritmo.

Como resultados de este problema, podemos decir que:

- El programa presenta una observación: a la hora de generar datos aleatorios y luego llamar a la función para su ordenamiento en otro archivo, se debe hacer solamente una vez porque sino se generan varios archivos distintos de generación de datos distintos (se van pisando entre sí) mientras que el de ordenamiento queda tal cual como la primera vez que se compiló. Para su correcto funcionamiento, se deben borrar los dos .txt creados a la hora de reiniciar el proceso.
- El programa cumple con lo solicitado en la consigna.