Facultad de
UNER Ingeniería

**Barrera, Muchiut, Oliver**
**AyED 2C2025**

# Trabajo Práctico N°1:

## Informe Actividad n°1:

Para resolver esta actividad implementamos una TAD que modela una lista doblemente enlazada o LDE. Esta nos permite eliminar e insertar objetos dentro de la misma en distintas posiciones, recorrerla de principio a fin y viceversa, etc.

Como objetivos, teníamos que verificar que la implementación respete la especificación lógica, y además, realizar gráficas para ver que tan costoso en cuanto a tiempo es realizar las siguientes funciones: __len__ , copiar e invertir.

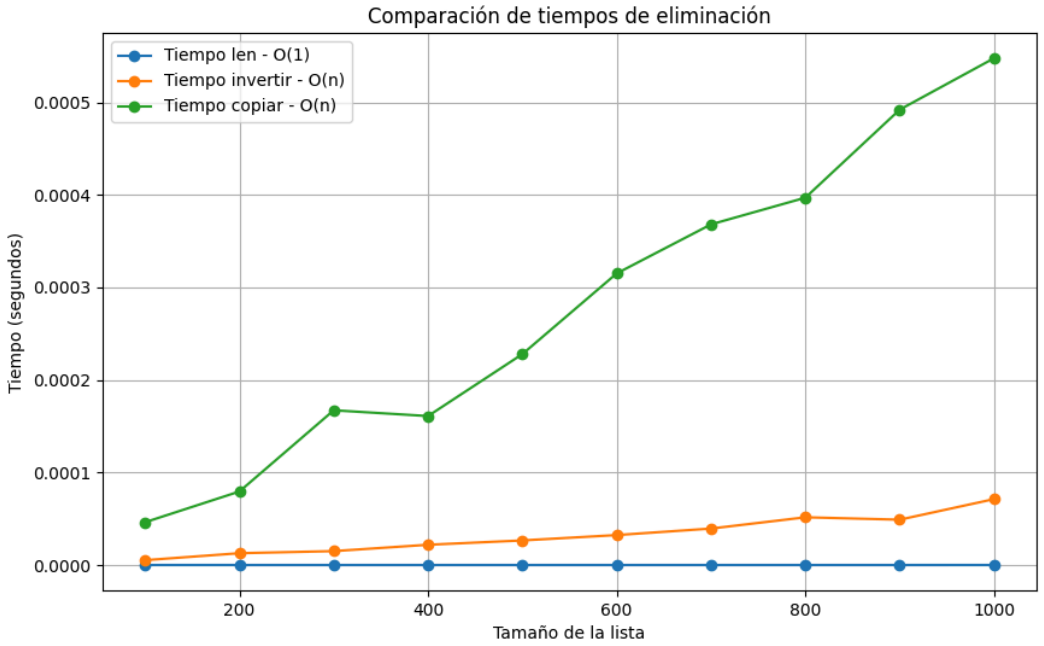
La clase de Lista Doblemente Enlazada implementa las siguientes funciones:

- esta_vacia(): Retorna True si la lista no tiene ningún elemento.
- __len__(): Te indica la cantidad de objetos que hay en la lista. Esta posee un orden de complejidad de O(1).
- agregar_al_inicio(dato): Agrega un dato al inicio de la lista.
- agregar_al_final(dato): Agrega un dato al final de la lista.
- insertar(dato,posición): Agrega un dato en la lista, en una posición especificada anteriormente, siempre y cuando la posición esté dentro del rango, sino saltara error.
- Extraer(posición): Extrae un dato en una posición especificada anteriormente de la lista y te la devuelve, esta puede ser al principio, al final o más adentro. Si no se brinda una posición, este eliminará el último y lo devolverá.
- copiar(): Realiza una copia de la lista elemento por elemento y la devuelve. Esta posee un orden de complejidad de O(n) ya que recorre la lista una vez, así que dependerá de cuantos elementos posee la misma.
- invertir(): Invierte el orden de los elementos de la lista. Mismo caso que con copiar(), el programa recorre la lista una sola vez, por lo que su orden de complejidad es de O(n) también.
- concatenar(lista): Recibe una lista como argumento y retorna la lista principal con los elementos de la lista nueva agregados al final.
- __add__(lista): Su resultado es una nueva lista formada por los elementos de las otras 2 listas sumadas.

- __iter()__: Permite recorrer una lista mediante el uso de un ciclo for.

Además podemos encontrar a la función __init__(), que sirve para crear una nueva lista vacía.
Pasando a la gráfica, realizamos mediciones de tiempo para los métodos __len__, copiar e invertir, utilizando listas con diferentes cantidades de objetos (100, 200, 300, … , 1000). Además, la función tiem.perf_counter() para obtener tiempos sumamente precisos.

Gráfica de tiempo de cada método investigado:




Analizando las gráficas, podemos observar que la función azul (len) realmente tiene un orden de complejidad O(1) al ser una constante, significando también que es la que menos tiempo demora en ejecutarse al 100%.

La función naranja perteneciente al método invertir, posee un carácter lineal creciente, por lo que sí se puede afirmar que su orden de complejidad es de O(n), además de que tarda un poco más que el método anteriormente nombrado.

Finalmente tenemos a la función verde, perteneciente al método copiar, que también posee un comportamiento lineal, más exagerado que el método invertir debido a que tarda más en ejecutarse del todo, eso quizás depende de la cantidad de objetos de la lista. Concluimos que su orden de complejidad es también de O(n).