Morales, Moreira Yanz, Zampedri
AyED 1C2025

Facultad de Ingeniería
UNER

Trabajo Práctico N°1

Informe Problema 2

Para la resolución de este problema implementamos un TAD que modela una lista doblemente enlazada (LDE), esta estructura nos permite realizar inserciones y eliminaciones de manera eficiente tanto al principio como al final de la lista, así como recorrerla en ambos sentidos.

Nuestro objetivo fue no solo verificar que la implementación respete la especificación lógica propuesta en la consigna, sino también analizar empíricamente la eficiencia de algunos de sus métodos fundamentales: __len__, copiar e invertir.

Nuestra clase implementa todos los métodos solicitados por la especificación lógica:

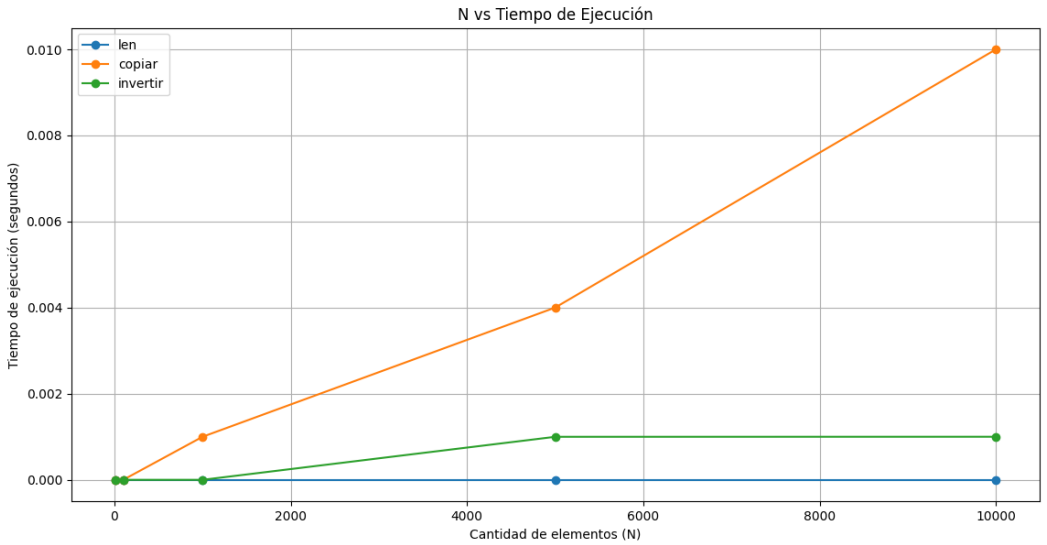
- esta_vacia(): Retorna True si la lista está vacía.
- __len__(): Retorna la cantidad de elementos. Utiliza un contador interno actualizado en cada inserción o extracción, lo que garantiza una complejidad **O(1)**.
- agregar_al_inicio(item) y agregar_al_final(item): Insertan elementos en los extremos en **tiempo constante**, actualizando los punteros primero y ultimo.
- insertar(item, posicion): Inserta un elemento en la posición especificada. Lanza una excepción IndexError si la posición es inválida.
- extraer(posicion=None): Elimina y retorna el elemento en la posición indicada. Si no se especifica, elimina el último. Las extracciones en los extremos se realizan en **O(1)**, cumpliendo con la restricción de eficiencia.
- copiar(): Crea una nueva lista con los mismos elementos en el mismo orden, recorriendo la lista una única vez. Su complejidad es **O(n)**.
- invertir(): Invierte el orden de los elementos modificando directamente los punteros de los nodos, sin usar estructuras auxiliares.
- concatenar(lista): Añade todos los elementos de otra lista doblemente enlazada al final de la actual, actualizando los punteros correspondientes.
- __iter__(): Implementa un iterador para recorrer los elementos de la lista desde el primero hasta el último. Requiere recorrer toda la lista, por lo que su complejidad es O(n).
- __add__(lista): Devuelve una nueva lista resultante de concatenar dos listas, reutilizando el método concatenar para evitar código duplicado.

El método __init__ crea una lista inicialmente vacía.

Realizamos mediciones de tiempo de ejecución para los métodos __len__, copiar e invertir, utilizando listas de distintos tamaños: 10, 100, 1000, 5000 y 10000 elementos.

Utilizamos la función time.perf_counter() para obtener mediciones de alta precisión, y la biblioteca matplotlib para visualizar los resultados.

Gráfica de N vs tiempo de ejecución para los métodos antes mencionados:



Observamos que:

El tiempo de ejecución del método __len__() se mantuvo prácticamente constante a medida que aumentaba la cantidad de elementos en la lista, lo cual es coherente con su complejidad O(1), ya que simplemente retorna el valor almacenado del tamaño sin necesidad de recorrer la estructura.

En el caso de copiar(), se observa un crecimiento claramente lineal del tiempo de ejecución en función de la cantidad de elementos, lo que es consistente con una complejidad O(n), dado que se recorren todos los nodos para crear una nueva lista con copias de los elementos.

Por su parte, invertir() también presenta un crecimiento que se **aproxima** al lineal, aunque con una pendiente más suave. Esto es esperable, ya que si bien también recorre toda la lista, la operación de reacomodar los punteros podría tener una carga ligeramente menor en comparación con la duplicación de nodos que realiza copiar(). Teniendo un orden de complejidad O(n).

Aunque los métodos copiar e invertir tienen complejidad lineal, no todas las operaciones constantes pesan lo mismo, razón por la cual creemos a la que se deben las diferencias en la gráfica.

Sabemos que invertir recorre la lista y solo intercambia punteros (siguiente y anterior) y actualiza un nodo: operaciones baratas en memoria. Mientras que copiar crea un nuevo nodo para cada elemento (asignación de memoria). Llama a agregar_al_final, que aunque es O(1), realiza varias operaciones por nodo: creación del nodo, enlace de punteros, asignaciones múltiples, incremento de tamaño, etc. En total, hay más trabajo por iteración; entonces: O(n) vs. O(n), pero con distintas “constantes ocultas”.