Informe de TAD Lista Doblemente Enlazada

UNIVERSIDAD NACIONAL DE ENTRE RÍOS

Facultad de Ingeniería

Carrera: T.U.P.E.D

Materia: Fundamentos de Algoritmos y Estructuras de Datos

Estudiante: Franco Tomiozzo

Profesor: Dr. Javier E. Diaz Zamboni

Bioing. Jordán F. Insfrán

Esp. Bioing. Francisco Rizzato

Comisión: 1

Fecha: Septiembre 2025

Introducción

El trabajo tiene como objetivo implementar y analizar un Tipo Abstracto de Datos (TAD) usando una Lista Doblemente Enlazada, e analizando los tiempos de función de tres métodos fundamentales: **len()**, **copiar()** e **invertir()**.

Desarrollo

La implementación incluye de varias optimizaciones:

- **Contador de tamaño:** Permite que el método **len()** sea **O(1)** en ves de **O(n)**
- **Referencias a cola:** Facilita el ingreso y extracción de **O(1)** al final
- **Búsqueda bidireccional:** El método (**_obtener_nodo()**) optimiza la búsqueda comenzando desde el extremo más cercano

Método len() - O(1)

El método **len() solo**e retorna el valor del atributo (**self.tamanio**), que se mantiene actualizado en cada operación de agregado o de eliminación. Entonces su complejidad temporal es constante.

```
def __len__(self):
    """Devuelve el número de ítems - O(1)"""
    return self.tamanio
```

Método copiar() - O(n)

El método **copiar()** recorre toda la lista original y crear nuevos nodos para una lista copia. El número de operaciones proporciona con el número de elementos que hay.

```
def copiar(self):
    """Realiza una copia de la lista - O(n)"""
    nueva_lista = ListaDobleEnlazada()
    actual = self.cabeza
    while actual is not None:
        nueva_lista.agregar_al_final(actual.dato)
        actual = actual.siguiente
    return nueva_lista
```

Justificación: El bucle **while** se ejecuta exactamente **n** iteraciones, donde **n** es el número de elementos en la lista.

Método invertir() - O(n)

El método **invertir()** cambia los punteros siguiente y anterior de cada uno del nodo, requiriendo que recorra completamente la lista.

```
def invertir(self):
        """Invierte el orden de los elementos - O(n)"""
        if self.esta_vacia() or self.tamanio == 1:
                return
        actual = self.cabeza
        while actual is not None:
                # Intercambiar punteros
                actual.siguiente, actual.anterior = actual.anterior,
actual.siguiente
                actual = actual.anterior # Moverse al siguiente
        # Intercambiar cabeza y cola
        self.cabeza, self.cola = self.cola, self.cabeza
```

Experimento

Método de Medición

Se realizaron mediciones experimentales para saber cuánto tiempo lleva ejecutar cada método con las siguientes características:

Tamaños de la prueba: 100, 500, 1000, 2000, 3000, 4000, 5000, 7500, 10000, 15000 elementos

Repeticiones de la medición: 1000 repeticiones para **len()**, 10 para **copiar()** e **invertir()**

Herramienta de medición: **time.perf_counter()** para máxima precisión

Tabla de promedio: Se calcula la media aritmética de todas las repeticiones y se agrega en una tabla y un gráfico por cada método.

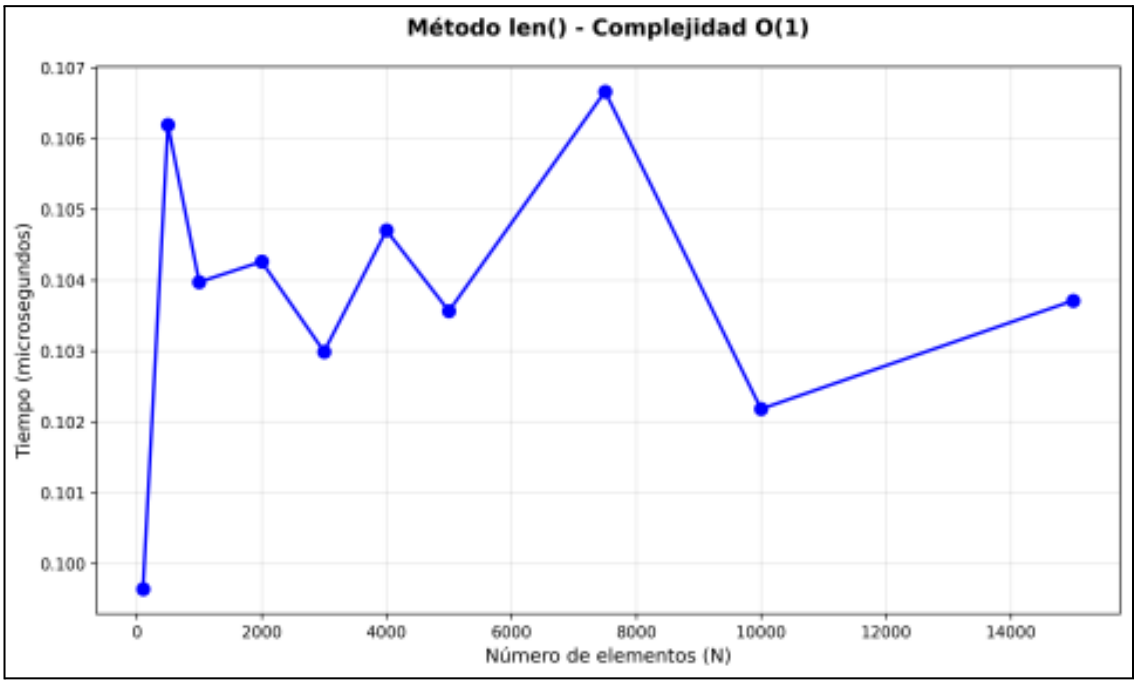
Resultados Obtenidos

Tamaño (N)	len() (µs)	copiar() (ms)	invertir() (µs)
100	0.10	0.035	7.3
500	0.11	0.183	12.8
1000	0.10	0.372	24.3
2000	0.10	0.640	48.5
3000	0.10	2.351	72.7
4000	0.10	2.665	90.6
5000	0.10	3.079	119.2

7500	0.11	4.001	176.3
10000	0.10	5.097	233.3
15000	0.10	8.945	344.3

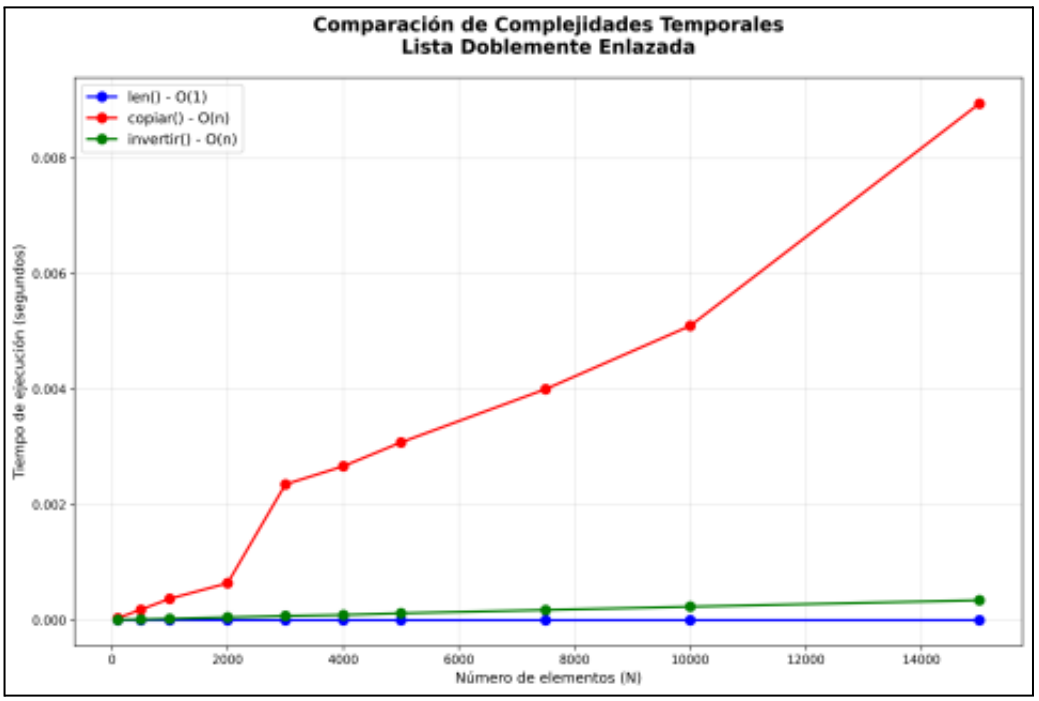
Gráficas de Rendimiento

Figura 1: Método len() - Complejidad O(1)



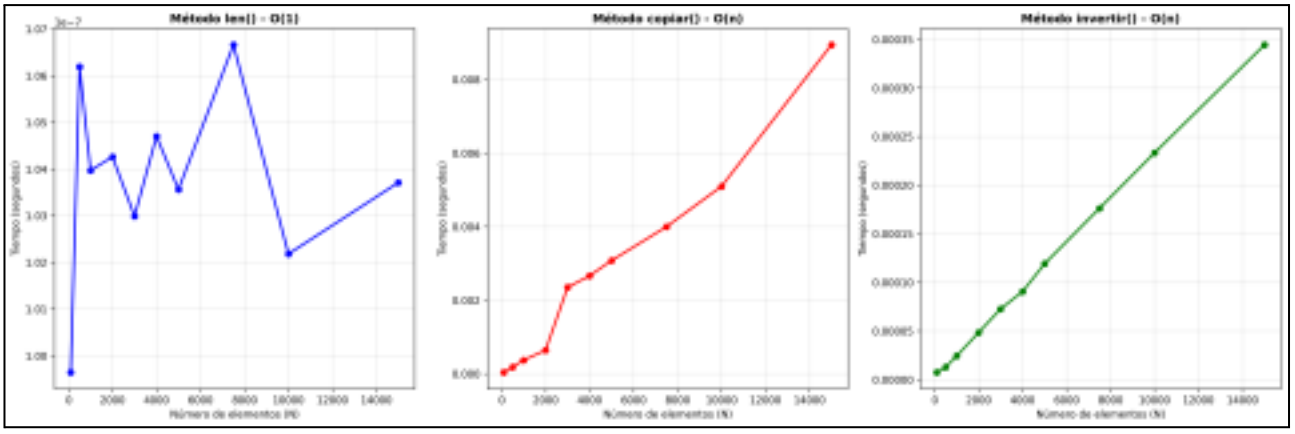
La observación es que el tiempo permanece prácticamente constante independientemente del tamaño de la lista

Figura 2: Comparación de Complejidades Temporales



La gráfica muestra una diferencia entre operaciones O(1) y O(n) y los métodos copiar() e invertir() muestran crecimiento lineal, mientras que len() permanece constante.

Figura 3: Análisis Individual de Métodos



Pseudocódigo de Métodos Clave

Pseudocódigo del método len()

FUNCIÓN len() RETORNAR self.tamanio FIN FUNCIÓN Complejidad: O(1)
Justificación: Una sola operación de lectura

Pseudocódigo del método copiar()

FUNCIÓN copiar() nueva_lista ← nueva ListaDobleEnlazada() actual ←
self.cabeza MIENTRAS actual ≠ NULL HACER
nueva_lista.agregar_al_final(actual.dato) actual ← actual.siguiente
FIN MIENTRAS RETORNAR nueva_lista FIN FUNCIÓN Complejidad: O(n)
Justificación: Un bucle que itera n veces (n = número de elementos)

Pseudocódigo del método invertir()

FUNCIÓN invertir() SI self.esta_vacia() O self.tamanio = 1 ENTONCES
RETORNAR FIN SI actual ← self.cabeza MIENTRAS actual ≠ NULL HACER //
Intercambiar punteros siguiente y anterior temp ← actual.siguiente
actual.siguiente ← actual.anterior actual.anterior ← temp actual ←
actual.anterior // Moverse al siguiente nodo FIN MIENTRAS //
Intercambiar cabeza y cola temp ← self.cabeza self.cabeza ← self.cola
self.cola ← temp FIN FUNCIÓN Complejidad: O(n) Justificación: Un
bucle que visita cada nodo exactamente una vez

Conclusión

Los resultados experimentales dan un definición más detallada de los métodos:

El método **len() - O(1):** Los tiempos permanecen prácticamente constantes con (~0.10 µs), independientemente del tamaño de la lista, demuestra que es más eficiente mantener un contador de tamaño.

El método **copiar() - O(n):** Con una relación lineal clara entre el tamaño de la lista y el tiempo de ejecución y con un factor de proporcional aproximado de un 0.6 µs por cada elemento.

El método **invertir() - O(n):** También muestra un comportamiento lineal, pero es un poco más eficiente que el método copiar() por ~23 ns por elemento, debido a que solo modifica punteros sin crear nuevos nodos.