Implementación de una Lista Doblemente Enlazada

Introducción

En este documento, se presenta una implementación en Python de una lista doblemente enlazada. Una lista doblemente enlazada es una estructura de datos que consta de nodos, donde cada nodo contiene un elemento de datos y dos punteros, uno que apunta al nodo siguiente y otro que apunta al nodo anterior en la lista. Esta implementación proporciona una serie de operaciones básicas que se pueden realizar en una lista doblemente enlazada, cómo agregar elementos al principio o al final, insertar elementos en una posición específica, extraer elementos de la lista, invertir la lista, ordenarla y concatenarla con otra lista.

Desarrollo

La implementación consta de dos clases principales:

1. `Nodo_DobleEnlazado`: Esta clase representa un nodo en la lista doblemente enlazada. Cada nodo contiene un campo de dato, así como punteros al nodo siguiente y al nodo anterior.

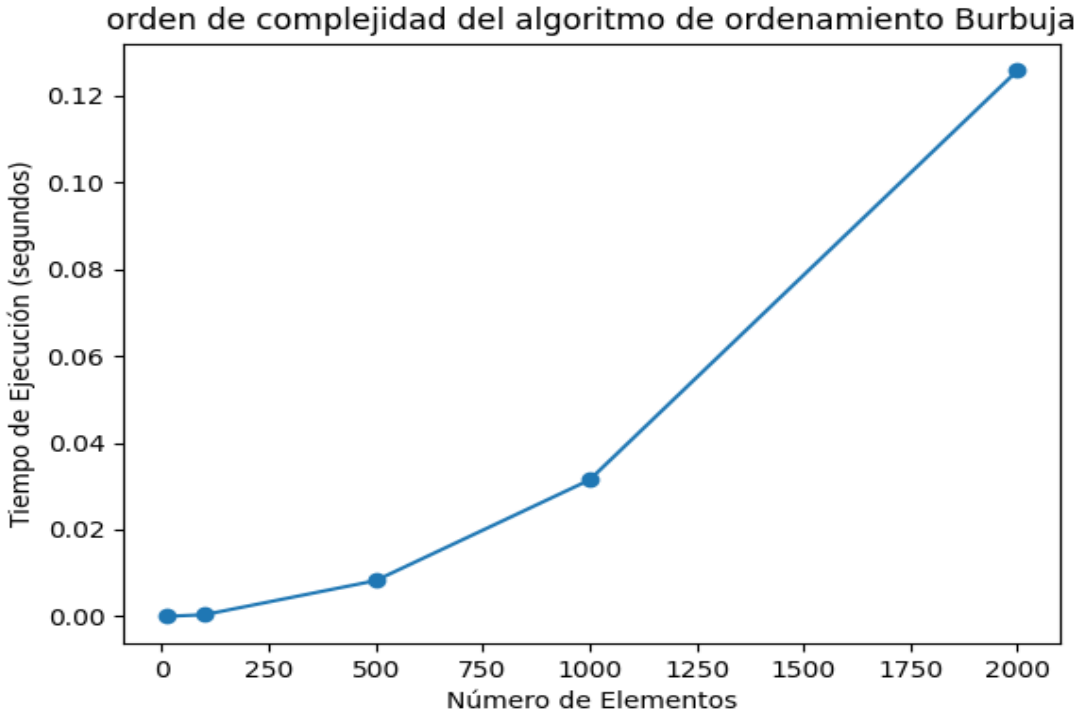
2. `ListaDobleEnlazada`: Esta clase representa la lista doblemente enlazada en sí. Contiene un puntero a la cabeza (primer nodo), un puntero a la cola (último nodo) y un contador de tamaño para rastrear el número de elementos en la lista. La clase `ListaDobleEnlazada` proporciona diversas operaciones para gestionar la lista, cómo agregar elementos, insertar elementos en una posición específica, extraer elementos, invertir la lista, ordenarla y concatenarla con otra lista.

Operaciones Principales

- Agregar al principio (`agregar_al_inicio`) y al final (`agregar_al_final`): Estas operaciones permiten agregar elementos al principio o al final de la lista, respectivamente.
- Insertar en una posición específica (`insertar`): Esta operación permite insertar un elemento en una posición específica de la lista.
- Extraer (`extraer`): Esta operación permite extraer un elemento de la lista, ya sea desde el principio, el final o una posición específica.
- Invertir (`invertir`): Esta operación invierte el orden de los elementos en la lista.
- Ordenar (`ordenar`): Esta operación ordena los elementos de la lista en orden ascendente.

- Concatenar (`concatenar`) y operación de suma (`__add__`): Estas operaciones permiten concatenar la lista actual con otra lista.

Análisis de complejidad:



El tiempo de ejecución aumenta a medida que el tamaño de la lista crece, por lo que el algoritmo tiene una complejidad $O(n^2)$, donde n es el número de elementos en la lista. Esto se debe a que se utiliza el algoritmo de ordenación de inserción, que tiene una complejidad $O(n^2)$.

Conclusiones

Lo más destacable es que en el código se proporcionan una amplia gama de operaciones comunes de lista, como: agregar elementos al principio y al final, insertar en una posición específica, extraer elementos, copiar, invertir y ordenar la lista. Además, la capacidad de concatenar dos listas es una característica útil que facilita la manipulación de datos.

Un aspecto clave del que nos dimos cuenta es la gestión adecuada de la memoria, asegurándonos de crear y liberar nodos de manera correcta ya que previene problemas de pérdida de memoria o fragmentación.

Sin embargo, nos pareció importante destacar que, la inserción en una posición específica puede volverse costosa si esa posición está cerca del final de la lista, ya que se requiere recorrer la lista desde el principio hasta la posición deseada.

Por último, cuando medimos el rendimiento del algoritmo de ordenamiento de burbuja en listas de diferentes tamaños, al observar la gráfica, para analizar cómo varía el tiempo de ejecución con el tamaño de la lista, identificamos cómo el tiempo de ejecución aumenta a medida que el tamaño de la lista crece, lo que hace a nuestro algoritmo ser de complejidad

$O(n^2)$, por lo que, aunque Bubble Sort es fácil de entender e implementar, es muy ineficiente para listas grandes debido a su complejidad cuadrática ($O(n^2)$).