Reporte del Funcionamiento del Proyecto

Curso: Análisis de Algoritmos y Estructuras de Datos
Lenguaje de implementación: Python
Integrantes:

- Adrián Arias Vargas – C30749
- Jimena Bejarano Sánchez – C31074

---

1. Introducción

El presente proyecto tiene como objetivo analizar el rendimiento de diferentes estructuras de datos utilizadas para la implementación de un modelo tipo diccionario. El análisis se realizó en el lenguaje Python, desarrollando un conjunto de pruebas que permitieron comparar el comportamiento temporal de las operaciones básicas sobre distintas estructuras.

El proyecto se desarrolló en tres etapas progresivas:

1. Implementación de las estructuras base (listas y tablas hash).
2. Inclusión de estructuras más complejas (árbol binario de búsqueda y trie).
3. Medición y comparación del rendimiento de todas las implementaciones del modelo diccionario.

2. Descripción general del sistema

El sistema cuenta con un menú principal que permite ejecutar las pruebas de rendimiento de manera automatizada o probar manualmente el funcionamiento de cada estructura por separado.

A través del menú, el usuario puede inicializar las estructuras, insertar datos, realizar búsquedas, eliminar elementos y vaciar el contenido. Además, el programa permite ejecutar una batería de pruebas que miden el tiempo de ejecución promedio de las operaciones Init, Insert, Member, Delete, Print y Done en cada estructura, generando como salida un archivo .csv con los resultados y una serie de gráficos comparativos.

Las pruebas se realizaron utilizando conjuntos de datos de distintos tamaños, con 10 corridas por cada tamaño, lo que permitió obtener tiempos promedio más representativos a diferencia de haber hecho solamente una.

3. Estructuras de datos implementadas

3.1. Lista ordenada (por arreglos y punteros)

Se implementaron dos versiones de la lista ordenada: una basada en arreglos y otra basada en punteros. Ambas mantienen los elementos en orden ascendente en todo momento.

- Lista por arreglos: utiliza un arreglo de tamaño fijo. Las inserciones y eliminaciones implican mover elementos, lo cual genera un costo promedio de O(n).

- Lista por punteros: implementada mediante nodos enlazados dinámicamente. Ofrece mayor flexibilidad en tamaño y reduce el desperdicio de memoria, aunque las operaciones también tienen costo promedio O(n).

3.2. Tabla Hash abierta

La tabla hash implementada utiliza listas enlazadas para manejar colisiones. Cada clave se transforma en un índice mediante una función hash que combina los valores ASCII de los caracteres multiplicados por números primos.
Las operaciones de inserción, búsqueda y eliminación presentan un tiempo promedio O(1), aunque en el peor de los casos pueden alcanzar O(n) si se producen muchas colisiones.

3.3. Árbol Binario de Búsqueda (ABB)

Se desarrollaron dos versiones:

- ABB por punteros: donde cada nodo tiene referencias a sus hijos izquierdo y derecho. Permite operaciones dinámicas sin preocuparse por el tamaño inicial del árbol.

- ABB por vector: mantiene la estructura en un arreglo, aprovechando el índice de cada elemento para representar las relaciones jerárquicas.

Ambas versiones permiten búsquedas, inserciones y eliminaciones con complejidad O(log n) en promedio, aunque pueden llegar a O(n) en árboles desbalanceados.

3.4. Trie

Se implementó también el Trie en dos versiones:

- Trie por punteros: cada nodo contiene un arreglo de referencias a sus hijos (uno por cada letra) y un marcador booleano que indica el fin de una palabra.

- Trie por arreglos: utiliza arreglos de enteros para representar las conexiones entre nodos, con un valor de -1 para indicar ausencia de hijos.

El Trie permite realizar búsquedas y verificaciones de prefijos de manera muy eficiente, en tiempo proporcional a la longitud de la palabra (O(m)).

4. Funcionamiento de las pruebas de rendimiento

El módulo principal (main.py) permite seleccionar la opción de pruebas automáticas, donde se generan palabras aleatorias que actúan como claves del diccionario.

Las funciones principales del sistema son:

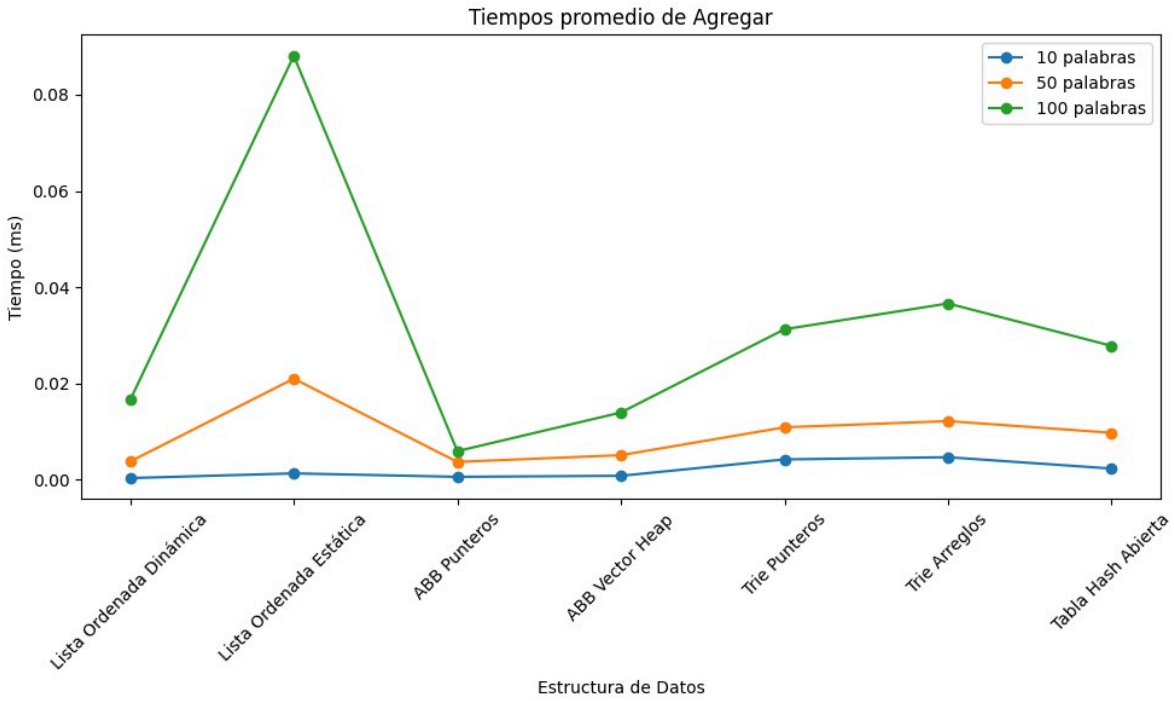
- generar_palabras(): crea cadenas aleatorias de hasta 20 caracteres en minúsculas.

- medir_tiempo(): calcula el tiempo en milisegundos que tarda cada operación en completarse.

- Insert, Delete, Member, Print, Clear, Done: ejecutan las operaciones sobre las distintas estructuras y reportan sus tiempos.

El programa organiza los resultados en una tabla mediante la librería pandas, y los exporta en formato .csv. Posteriormente, utiliza matplotlib para generar gráficos comparativos (uno por operación) donde se observa la relación entre el tamaño de los datos y el tiempo promedio de ejecución.

5. Análisis de gráficas

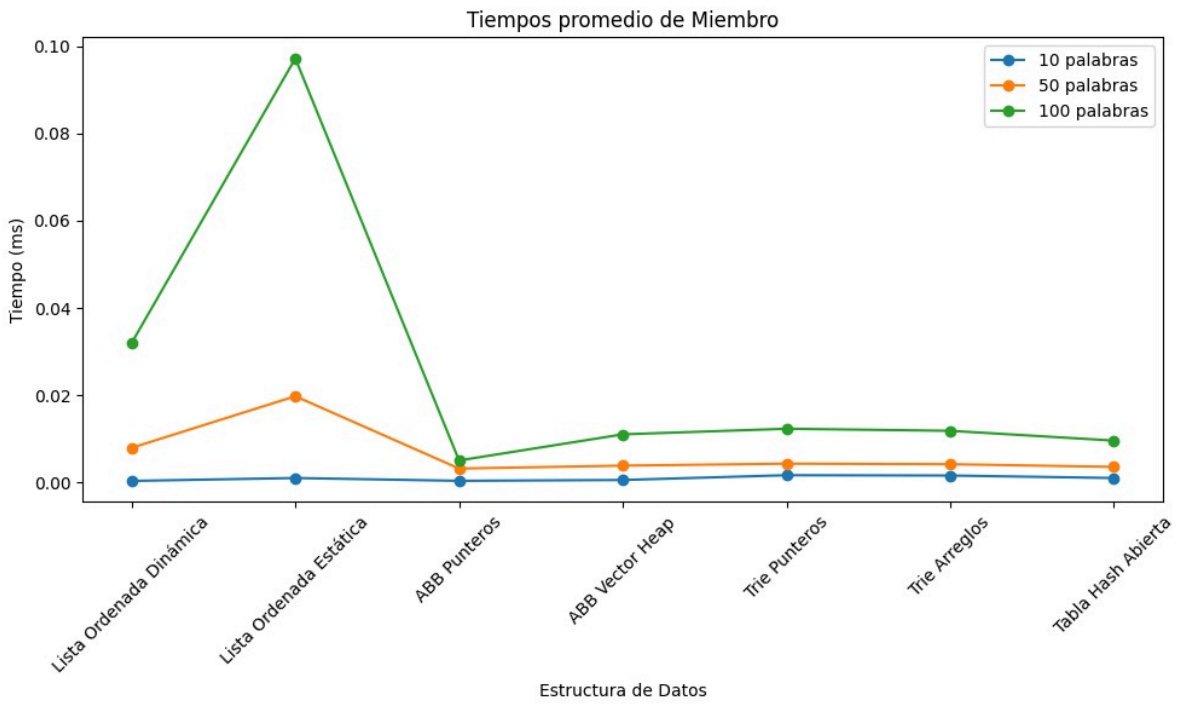


En esta gráfica se observa cómo el tiempo promedio de inserción aumenta a medida que crece el tamaño del diccionario.

- Para diccionarios pequeños (10), todas las estructuras muestran tiempos muy bajos, pero la Lista Ordenada Dinámica y la Tabla Hash presentan las mejores inserciones.

- Al crecer a 50 y 100 elementos, la Tabla Hash Abierta mantiene un crecimiento moderado gracias a su complejidad promedio O(1), mientras que las listas ordenadas y los árboles binarios muestran aumentos más pronunciados por sus costos O(n) y O(log n) respectivamente.

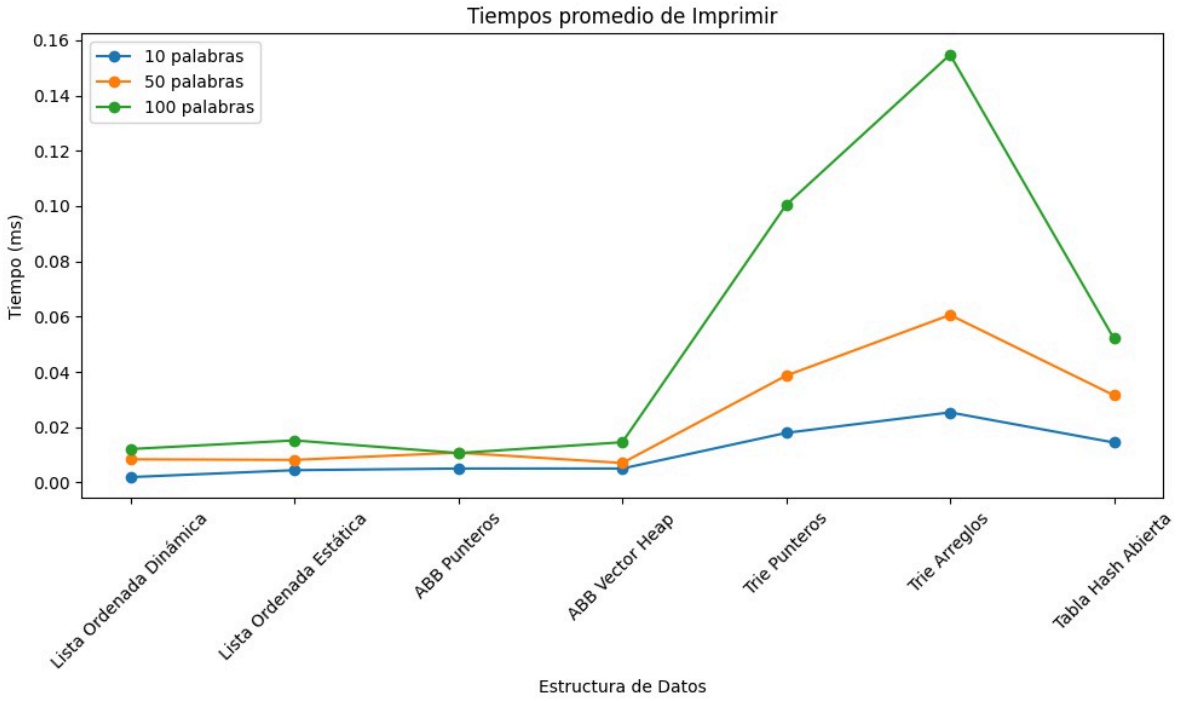
- El Trie (por punteros y arreglos) muestra un rendimiento intermedio, afectado por el recorrido carácter a carácter, pero sigue siendo eficiente en estructuras con muchas claves distintas.



- Esta métrica muestra la eficiencia en encontrar un elemento en el diccionario.
- En tamaños pequeños (10) las diferencias son casi imperceptibles, pero al aumentar a 50 y 100, se nota que las Tablas Hash y los Tries se mantienen con tiempos bajos y estables, como se espera por su búsqueda promedio O(1) y O(L) (siendo L la longitud de la palabra).

- Los árboles binarios presentan buen comportamiento con tiempos en torno a O(log n), aunque si el árbol está desbalanceado, el rendimiento puede degradarse.

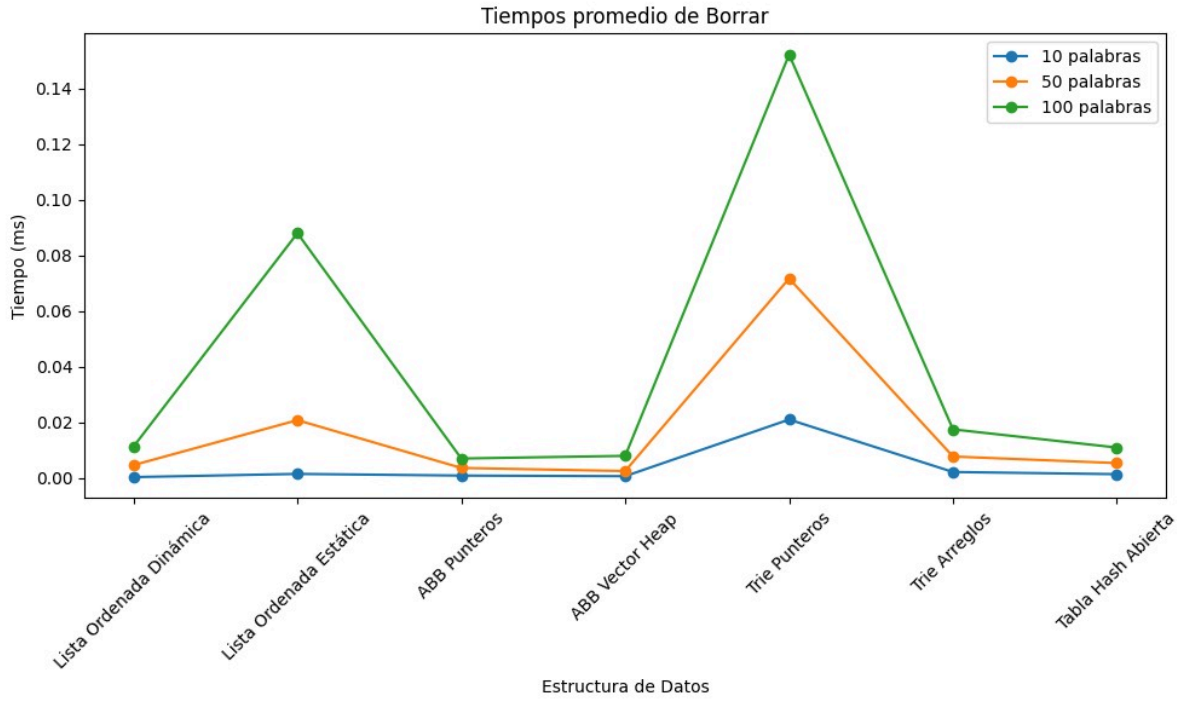
- En cambio, las Listas Ordenadas (dinámica y estática) tienen los tiempos más altos porque requieren recorrido secuencial (O(n)).



Esta prueba mide el tiempo de recorrer y mostrar todos los elementos.

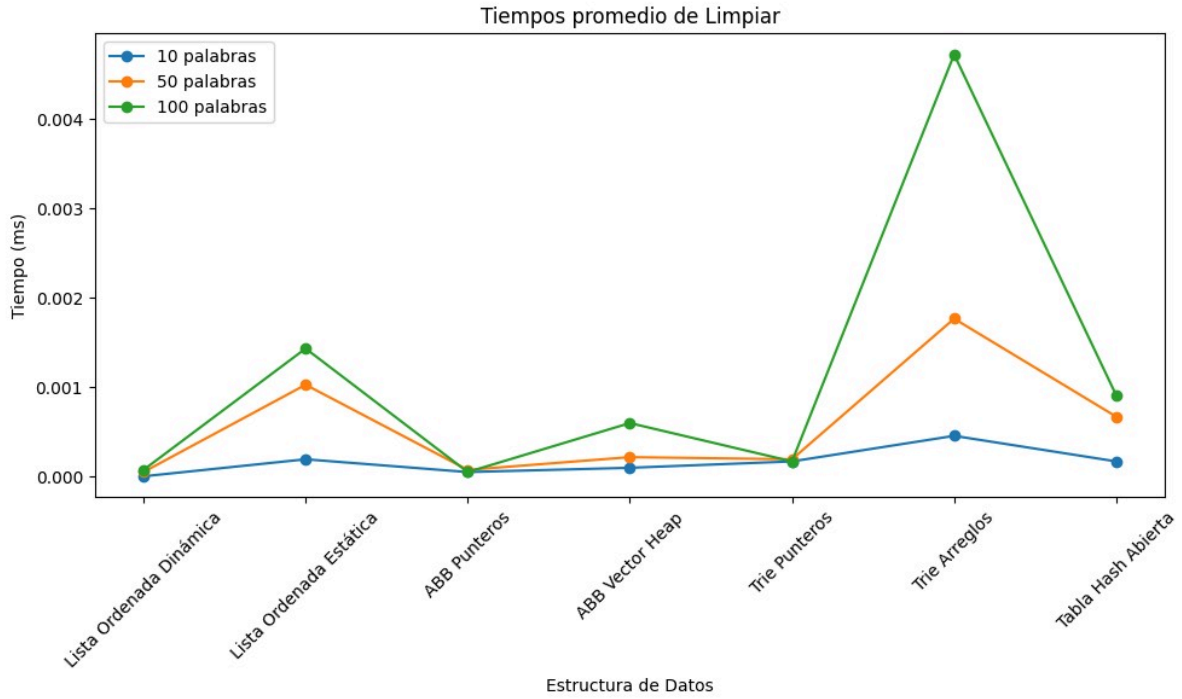
- Dado que todas las estructuras deben recorrer el conjunto completo, los tiempos dependen directamente del número de elementos almacenados (O(n) para todas).

- En diccionarios pequeños, las diferencias son mínimas. Pero con 50 y 100 elementos, las Listas y Tries muestran tiempos ligeramente mayores por el orden y la construcción de las cadenas de salida.



Esta gráfica refleja la complejidad de eliminar un elemento.

- Para tamaños pequeños, las diferencias son mínimas.
- A medida que crece el tamaño del diccionario, los tiempos aumentan notablemente en las Listas Ordenadas, ya que necesitan recorrer para encontrar y ajustar punteros o índices (O(n)).
- Los árboles binarios mantienen tiempos moderados (O(log n) promedio), aunque varían según el balance.
- La Tabla Hash presenta los mejores tiempos en general, al eliminar directamente del bucket correspondiente.
- Los Tries mantienen un comportamiento estable, aunque un poco más costoso, porque deben recorrer cada letra y liberar nodos solo si es seguro.



Esta métrica mide cuánto tarda en vaciar toda la estructura.

- En todos los casos, los tiempos son muy bajos, cercanos a cero, ya que se limitan a reinicializar referencias o listas.

- Las diferencias son tan pequeñas que dependen más de la implementación del lenguaje que de la estructura en sí.

- Las listas y árboles muestran tiempos similares, mientras que los tries y la tabla hash pueden tardar un poco más si deben reconstruir estructuras internas o listas vacías.

Como conclusión:

- La Tabla Hash Abierta se posiciona como la estructura más eficiente en general para operaciones de inserción, búsqueda y eliminación, gracias a su acceso promedio constante O(1).
- Árboles Binarios (ABB) son buenos candidatos cuando se requiere ordenamiento, con desempeño O(log n) en la mayoría de operaciones.
- Tries son ideales cuando las claves comparten prefijos o hay muchas cadenas similares, manteniendo tiempos estables O(L) sin depender del número total de palabras.
- Las Listas Ordenadas, tanto dinámica como estática, son simples pero ineficientes a medida que el tamaño crece, ya que la mayoría de sus operaciones

son O(n).

En cuanto a imprimir y limpiar, todas las estructuras tienen tiempos comparables y adecuados, con crecimiento lineal controlado.

6. Conclusiones generales

El desarrollo del proyecto permitió comprender de manera práctica el funcionamiento interno y la eficiencia de las estructuras de datos más utilizadas en la implementación de diccionarios.

El proceso evidenció que la elección de la estructura depende directamente del tipo de operaciones que se realicen con mayor frecuencia. Estructuras como la tabla hash resultan ideales para búsquedas rápidas, mientras que los árboles binarios y tries destacan en operaciones de ordenamiento o manejo de claves jerárquicas.

Además, se comprobó la importancia de medir el rendimiento real de los algoritmos, más allá de su complejidad teórica, ya que factores como la gestión de memoria o la implementación interna del lenguaje pueden influir significativamente en los resultados.

7. Dificultades y aspectos por mejorar

Durante el desarrollo se presentaron varios retos técnicos y oportunidades de mejora:

- Se requirió la instalación de varias librerías adicionales como venv, uv, matplotlib y pandas, necesarias para la ejecución, análisis y visualización de resultados.

- La interfaz de usuario podría mejorarse para ser más atractiva y amigable visualmente, facilitando la interacción con el programa.

- A pesar de que el sistema manejó hasta 50,000 elementos, sería deseable optimizar el manejo de memoria y procesamiento para trabajar con volúmenes aún mayores de manera más fluida.

- Algunas estructuras podrían mejorarse para reducir tiempos de inserción y eliminar posibles cuellos de botella durante las pruebas masivas.