Tarea programada 1, Segundo Semestre 2025.

Estudiantes: Paula Sofia Grell Araya C22977, Melissa Garita Chacón C23186.

Resumen. En este trabajo se desarrolló la primera etapa del Proyecto de Algoritmos y Estructuras de Datos, centrado en la implementación del Modelo Diccionario y en el análisis de sus primeras estructuras de datos. El objetivo fue comprender en profundidad la organización, funcionamiento y rendimiento de distintos enfoques de almacenamiento y búsqueda de información.

Se definió el Modelo Diccionario con las operaciones fundamentales: Init, Done, Clear, Insert, Delete, Member y Print considerando cadenas de caracteres de hasta 20 letras en el rango 'a'..'z' . A partir de este modelo se implementaron tres estructuras concretas: Lista Ordenada Dinámica (por punteros), basada en listas enlazadas simples, que permite inserciones ordenadas y un crecimiento flexible en memoria.

Lista Ordenada Estática (por arreglos), que mantiene los elementos en un array ordenado, con acceso directo y un manejo más eficiente en lectura.

Tabla Hash Abierta, con resolución de colisiones mediante listas de buckets, una función hash definida a partir de la suma de códigos ASCII y un mecanismo de redistribución (rehash) que se activa al superar un factor de carga

Palabras clave: *estructuras de datos, diccionario, lista ordenada, lista enlazada, arreglos, tabla hash, inserción, búsqueda, eliminación, complejidad temporal, complejidad espacial.*

.

I. INTRODUCCIÓN

Las estructuras de datos constituyen un componente esencial en el diseño de algoritmos, pues permiten organizar y manipular información de manera eficiente. En este contexto, el Modelo Diccionario ofrece un marco uniforme de operaciones básicas (Init, Done, Clear, Insert, Delete, Member, Print) aplicadas a cadenas de caracteres de hasta 20 letras en el rango 'a'..'z'.

En la primera etapa se desarrollaron las estructuras Lista Ordenada por punteros, Lista Ordenada por arreglos y Tabla Hash abierta, analizando la uniformidad de la función hash y el costo del rehash. En la segunda etapa se incorporaron estructuras más avanzadas: el Árbol de Búsqueda Binaria (ABB) por punteros y por vector heap y el Trie por punteros y por arreglos, ampliando la comparación entre enfoques jerárquicos y de prefijos.

Finalmente, la tercera etapa se enfocó en el análisis de rendimiento y uso de espacio de las siete estructuras, midiendo tiempos de inserción, búsqueda y borrado para distintos tamaños de diccionario. Este estudio permitió contrastar los resultados experimentales con las complejidades teóricas y evaluar la eficiencia de cada estructura bajo un mismo marco de operaciones.

II. METODOLOGÍA

El proyecto se implementó en Python, utilizando un enfoque orientado a objetos y apoyándose en librerías estándar como time y statistics para medir tiempos de ejecución y evaluar la distribución de la función hash.

Se definió un Modelo Diccionario abstracto con las operaciones Init, Done, Clear, Insert, Delete, Member y Print. A partir de él se desarrollaron tres estructuras concretas:

- Lista Ordenada por punteros, basada en listas enlazadas.
- Lista Ordenada por arreglos, utilizando un vector ordenado.
- Tabla Hash abierta, con colisiones resueltas mediante listas y redistribución dinámica (*rehash*).

- Árbol de Búsqueda Binaria (ABB) por punteros y por vector heap, que permiten comparar enfoques jerárquicos de almacenamiento.
- Trie por punteros y Trie por arreglos, diseñados para almacenar cadenas compartiendo prefijos comunes.

Las implementaciones fueron integradas en un programa interactivo que permite seleccionar la estructura y ejecutar operaciones básicas del diccionario.

En la tercera etapa, se añadió un módulo de análisis experimental para medir los tiempos de inserción, búsqueda y eliminación en cada estructura, utilizando diferentes tamaños de diccionario. (N = 1000, 5000 y 10000). Esto permitió evaluar su rendimiento y uso de espacio, contrastando los resultados obtenidos con la complejidad teórica esperada.

III. DISCUSIÓN

El Modelo Diccionario constituye una abstracción fundamental dentro de las estructuras de datos, ya que define un conjunto uniforme de operaciones aplicables sin importar la implementación concreta utilizada. En el contexto de este proyecto, el diccionario se concibe como un tipo de dato abstracto cuyo propósito es almacenar cadenas de caracteres formadas únicamente por letras en el rango 'a'..'z', con una longitud máxima de veinte caracteres.

Un Abstract Data Type (ADT) es un modelo conceptual que define un conjunto de operaciones y su comportamiento esperado, sin especificar cómo se implementan internamente esas operaciones ni cómo se organiza la memoria. En otras palabras, el ADT se enfoca en el qué puede hacerse —las operaciones permitidas y sus efectos—, dejando al implementador la libertad de decidir el cómo. Esta definición se conoce también como abstracción, pues oculta los detalles internos de la representación de datos. [1]

El modelo fue implementado en Python como una clase abstracta que actúa como interfaz común, utilizando el módulo abc para garantizar que todas las estructuras hereden e implementen las operaciones definidas. De esta manera, se asegura que la interacción con el diccionario sea consistente sin importar si se utiliza una lista enlazada, un arreglo o una tabla hash como soporte interno.

Las operaciones que conforman el modelo son las siguientes. Init, que inicializa el diccionario vacío y se corresponde con el constructor __init__ de las implementaciones. Done, que libera los recursos utilizados y marca la estructura como destruida. Clear, encargada de

eliminar todos los elementos pero manteniendo el diccionario inicializado. Insert, que inserta un nuevo elemento aun si este ya existe en la colección. Delete, que busca un elemento y lo elimina si se encuentra, devolviendo un valor booleano que indica el éxito de la operación. Member, que verifica si un elemento pertenece al diccionario. Print, que muestra en pantalla todos los elementos almacenados siguiendo el orden interno de cada estructura. Finalmente, se define un método __str__ para devolver una representación textual útil en procesos de depuración.

Gracias a este diseño, el Modelo Diccionario mantiene independencia entre la interfaz y la implementación. Esto permite utilizar indistintamente distintas estructuras de datos —como la Lista Ordenada por punteros, la Lista Ordenada por arreglos y la Tabla Hash abierta—, mientras se asegura que todas cumplen con el mismo conjunto de operaciones definidas. Este enfoque, además de facilitar la validación y depuración, posibilita la comparación experimental entre diferentes técnicas de almacenamiento y búsqueda.

1. Lista Ordenada

La Lista Ordenada es una estructura de datos lineal que guarda los elementos siguiendo un orden específico, por ejemplo, alfabético o numérico. Cada vez que se inserta un nuevo dato, este se coloca en la posición que le corresponde para mantener el orden de la lista. Gracias a esto, es posible recorrer los elementos de manera secuencial y realizar operaciones como insertar, eliminar o buscar de forma más organizada. Su principal ventaja es que las búsquedas pueden hacerse más rápido porque los elementos ya están ordenados, aunque insertar o eliminar puede requerir mover datos o ajustar enlaces para conservar ese orden. [2]

La Lista Ordenada por punteros simplemente enlazada es una estructura donde cada elemento, llamado nodo, contiene dos partes: el dato en sí y una referencia o puntero al siguiente nodo de la lista. A diferencia de una lista estática, los nodos no ocupan posiciones contiguas en memoria, sino que se enlazan entre sí mediante estos punteros. En una versión ordenada, cada nodo se inserta en el lugar que mantiene el orden definido (por ejemplo, alfabético o numérico). Esto hace que las operaciones de inserción o eliminación sean más flexibles, ya que solo se necesita ajustar los punteros sin mover todos los elementos, aunque acceder a un nodo específico puede ser más lento porque se debe recorrer la lista desde el inicio. [3]

La Lista Ordenada por arreglos almacena sus elementos dentro de un bloque de memoria continua, como un arreglo o vector. En este tipo de lista, los datos se mantienen en orden y cada nuevo elemento se inserta en la posición correcta desplazando los elementos posteriores si es necesario. Esto permite un acceso rápido a cualquier posición, ya que los índices del arreglo facilitan el recorrido y la búsqueda secuencial o binaria. Sin embargo, las operaciones de inserción y eliminación pueden ser más costosas, ya que implican mover varios elementos para mantener el orden. Es una estructura adecuada cuando se necesita trabajar con un número fijo o limitado de elementos y se requiere un acceso rápido por posición, manteniendo al mismo tiempo la lista ordenada.[4]

2. Tabla Hash

La Tabla Hash es una estructura de datos que permite almacenar y acceder a

los elementos de forma muy rápida, utilizando una función especial llamada función hash. Esta función convierte cada clave (por ejemplo, una palabra o número) en un valor numérico que indica la posición donde se guardará el elemento dentro de la tabla. De esta manera, se puede acceder directamente al dato sin tener que recorrer toda la estructura. Cuando dos claves diferentes generan la misma posición (colisión), se aplican métodos como el encadenamiento o la dirección abierta para resolver el conflicto. Su principal ventaja es la eficiencia: las operaciones de inserción, búsqueda y eliminación suelen tener un tiempo casi constante. Por eso, las tablas hash son una de las estructuras más utilizadas cuando se necesita manejar grandes volúmenes de información de forma rápida y organizada.

La Tabla Hash abierta es una variación de la tabla hash que maneja las colisiones usando listas enlazadas en lugar de intentar encontrar otra posición libre dentro del arreglo principal. Cada posición o bucket de la tabla contiene una lista (normalmente una lista enlazada) con todos los elementos que comparten el mismo valor hash. Así, cuando ocurre una colisión, el nuevo elemento simplemente se agrega al final de la lista correspondiente. Este enfoque mantiene la eficiencia del acceso directo de las tablas hash, mientras permite manejar múltiples elementos en la misma posición sin sobrescribir datos. Aunque puede requerir más memoria al usar estructuras dinámicas, su comportamiento es más estable incluso cuando el factor de carga aumenta. Es un método sencillo, flexible y muy utilizado para implementar diccionarios o mapas en los que se necesita una gestión eficiente de colisiones. [4]

La función hash es la parte más importante de una tabla hash, ya que se encarga de transformar una clave (por ejemplo, una cadena de texto) en un número entero que indica la posición del elemento dentro de la tabla. Su objetivo principal es distribuir las claves de forma uniforme entre las diferentes posiciones o buckets, evitando que se concentren demasiados valores en un mismo lugar. Una buena función hash debe ser rápida de calcular y producir resultados aparentemente aleatorios, incluso cuando las claves tienen patrones similares. Para evaluar su “aleatoriedad” o uniformidad, se pueden generar muchas claves de prueba y analizar cuántas caen en cada bucket. Si la distribución es equilibrada, se considera que la función hash es eficiente. Esto reduce el número de colisiones y mejora el rendimiento general en las operaciones de inserción, búsqueda y eliminación. [4]

El proceso de redistribución o rehashing ocurre cuando la tabla hash alcanza un alto factor de carga, es decir, cuando la cantidad de elementos almacenados se acerca al tamaño máximo de la tabla. En ese momento, se crea una nueva tabla con una capacidad mayor y se vuelven a insertar todos los elementos antiguos, recalculando su posición con la misma función hash. Este proceso permite mantener una buena eficiencia y minimizar las colisiones, pero tiene un costo temporal, ya que requiere recorrer y reubicar todos los elementos existentes. Para evaluar su tiempo de duración, se mide cuánto tarda en completarse el rehash dependiendo del número de elementos. [4]

3. Árbol de Búsqueda Binaria

El Árbol de Búsqueda Binaria (ABB) es una estructura de datos jerárquica en la que cada nodo contiene un valor y tiene, como máximo, dos hijos: uno izquierdo y uno derecho. Su característica principal es que los valores

almacenados en el subárbol izquierdo son menores que el valor del nodo, mientras que los del subárbol derecho son mayores. Esta propiedad permite realizar operaciones de búsqueda, inserción y eliminación de manera eficiente, ya que en promedio pueden ejecutarse en tiempo O(log n). Sin embargo, si el árbol se desbalancea (por ejemplo, cuando los elementos se insertan en orden creciente o decreciente), su rendimiento puede degradarse hasta O(n). Los ABB son ampliamente utilizados por su capacidad para mantener datos ordenados dinámicamente y servir como base para estructuras más avanzadas, como los árboles balanceados (AVL o Red-Black Trees). [5]

El Árbol de Búsqueda Binaria por punteros implementa el modelo genérico del ABB utilizando nodos enlazados dinámicamente, donde cada nodo contiene un valor y dos referencias o punteros: uno hacia su hijo izquierdo y otro hacia su hijo derecho. Esta representación permite crear y manipular el árbol de forma flexible en memoria, sin necesidad de un tamaño predefinido. Las operaciones fundamentales —inserción, búsqueda y eliminación— se realizan recorriendo los punteros a partir de la raíz, comparando valores y descendiendo hacia el hijo correspondiente según el criterio de orden. Su principal ventaja radica en que permite una organización dinámica de los datos y un uso eficiente del espacio, ajustándose al crecimiento del árbol en tiempo de ejecución. Sin embargo, su rendimiento depende del grado de balanceo del árbol, ya que un ABB muy desbalanceado puede perder eficiencia y comportarse de manera similar a una lista enlazada.

El Árbol de Búsqueda Binaria por vector heap representa la estructura del ABB utilizando un arreglo lineal en lugar de nodos enlazados. En esta implementación, cada elemento del árbol se almacena en una posición del vector, donde el índice del hijo izquierdo se calcula como 2i + 1 y el del hijo derecho como 2i + 2, siendo i la posición del nodo padre. Este enfoque elimina el uso de punteros y aprovecha las posiciones del arreglo para mantener la jerarquía del árbol. Las operaciones de inserción, búsqueda y borrado se realizan siguiendo las mismas reglas de orden de un ABB, pero navegando mediante índices en lugar de referencias dinámicas. Esta representación ofrece acceso directo y eficiente a los nodos, simplificando la gestión de memoria y evitando la sobrecarga de asignaciones dinámicas. Sin embargo, el costo es un uso menos flexible de la memoria, ya que el vector puede contener espacios vacíos y requerir redimensionamiento si el árbol crece más allá de su capacidad inicial.

4. Trie

El Trie (también conocido como árbol de prefijos) es una estructura de datos jerárquica utilizada para almacenar y buscar cadenas de texto de manera eficiente, especialmente cuando comparten prefijos comunes. Cada nodo del Trie representa un carácter, y las rutas desde la raíz hasta un nodo terminal conforman una palabra completa. Esta organización permite realizar búsquedas, inserciones y verificaciones de existencia en tiempo proporcional a la longitud de la palabra, en lugar del número total de elementos almacenados. A diferencia de otras estructuras de búsqueda, el Trie no ordena los elementos según un valor numérico, sino según la secuencia de caracteres. Su principal ventaja es la optimización del almacenamiento de cadenas con prefijos compartidos, lo que reduce la redundancia de información y mejora el rendimiento en

aplicaciones como autocompletado, diccionarios digitales o compresión de texto. [7]

El Trie por punteros implementa el modelo genérico del Trie utilizando nodos enlazados dinámicamente, donde cada nodo mantiene un conjunto de punteros o referencias directas a sus hijos. Cada puntero representa una posible transición hacia el siguiente carácter dentro de una palabra. Esta representación permite un manejo flexible y dinámico de la estructura, ya que los nodos se crean únicamente cuando son necesarios, optimizando el uso de memoria. Las operaciones de inserción, búsqueda y eliminación recorren los punteros desde la raíz según los caracteres de la palabra, y marcan los nodos terminales para indicar el fin de una cadena. Gracias a este enfoque, la estructura puede adaptarse fácilmente a conjuntos de palabras de diferentes longitudes y es ideal para aplicaciones donde las palabras comparten muchos prefijos, como sistemas de autocompletado, correctores ortográficos o motores de búsqueda. [7]

El Trie por arreglos representa la estructura del Trie utilizando arreglos indexados en lugar de punteros o referencias dinámicas. En esta implementación, cada nodo contiene un arreglo de tamaño fijo (generalmente de 26 posiciones, una por cada letra del alfabeto), donde cada índice apunta al siguiente nodo correspondiente a un carácter. Este enfoque permite acceder directamente a los hijos de un nodo mediante operaciones de índice, logrando un acceso constante y predecible en tiempo O(1). Aunque esta representación tiende a consumir más memoria —ya que reserva espacio para todas las posibles letras en cada nodo—, resulta más eficiente en términos de velocidad y simplicidad para búsquedas y recorridos. El Trie por arreglos es especialmente útil cuando se trabaja con alfabetos limitados y conjuntos grandes de palabras, como en aplicaciones de procesamiento de texto, filtrado de palabras o autocompletado rápido, donde la prioridad es la velocidad de acceso sobre el ahorro de espacio. [7]

IV. LIMITACIONES DEL ANÁLISIS.

1. **Hardware y entorno:** los tiempos dependen del equipo, carga de CPU y memoria disponibles durante la ejecución.

2. **Medición de memoria:** sys.getsizeof( ) sólo mide el objeto principal, no el tamaño real de las estructuras anidadas.

3. **Datos aleatorios:** los tiempos pueden variar ligeramente entre ejecuciones debido al orden aleatorio de inserción.

4. **Escalabilidad:** el análisis se realizó hasta N = 10 000; tamaños mayores podrían requerir optimización o estructuras balanceadas.

5. **ABB sin balanceo explícito:** el ABB por punteros depende del orden aleatorio para mantenerse eficiente (sin AVL/Red-Black Tree).

V. RANGOS DE N UTILIZADOS

Rango de N	Descripción	Propósito
1000	Tamaño pequeño	Verificar comportamiento inicial y estabilidad
5000	Tamaño medio	Observar crecimiento temporal de operaciones
10000	Tamaño grande	Evaluar escalabilidad y eficiencia general

VI. CUADRO DE RESULTADOS EXPERIMENTALES

Cuadro 1. Tiempos promedio (en segundos) por estructura y tamaño (N).

Estructura	N=1000 Inserción	N=1000 Búsqueda	N=1000 Borrado
Lista Ordenada Dinámica	0.0228	0.0386	0.0212
Lista Ordenada Estática	0.0094	0.0118	0.0068
Tabla Hash Abierta	0.0094	0.0015	0.0016
ABB Punteros	0.0012	0.0012	0.0015
ABB Vector Heap	0.0109	0.0018	0.0032
Trie Punteros	0.0150	0.0015	0.0087
Trie Arreglos	0.0182	0.0071	0.0383

N=5000 Inserción	N=5000 Búsqueda	N=5000 Borrado
0.4338	0.7794	0.4422
0.1282	0.1819	0.2043
0.0344	0.0118	0.0136
0.0100	0.0090	0.0137
6.9246	0.0123	0.0187
1.8158	0.0084	0.0401
0.0995	0.0197	0.1861

N=10000 Inserción	N=10000 Búsqueda	N=10000 Borrado
2.0175	3.5783	1.9273
0.5053	0.8519	0.8161
0.0697	0.0263	0.0247
0.0282	0.0221	0.0260
21.6319	0.0378	0.0461
3,7111	0.0209	0.0732
3.1323	0.0478	0.4412

VII. GRÁFICOS

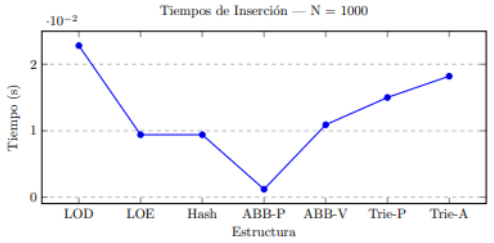

Figure 1: Comparación de inserción para N = 1000.

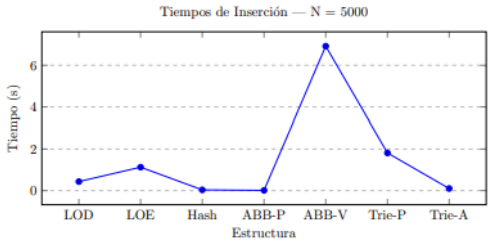

Figure 2: Comparación de inserción para N = 5000.

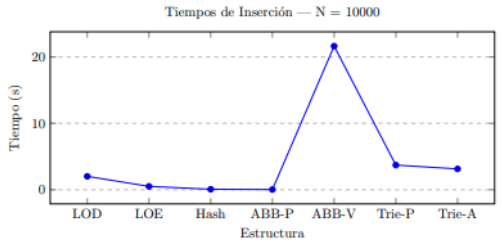

Figure 3: Comparación de inserción para N = 10000.

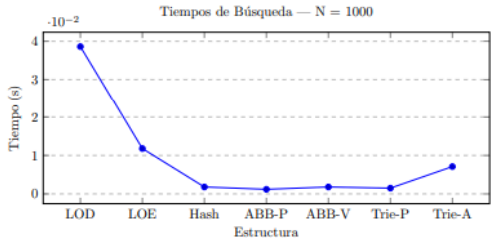

Figure 4: Comparación de búsqueda para N = 1000.

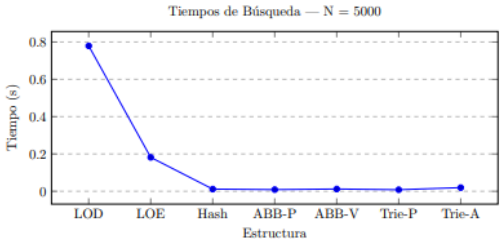

Figure 5: Comparación de búsqueda para N = 5000.

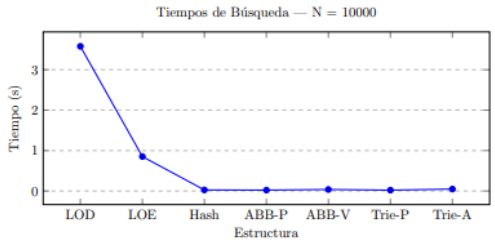

Figure 6: Comparación de búsqueda para N = 10000.

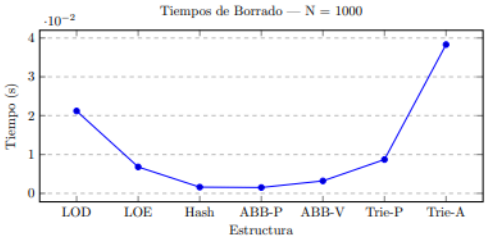

Figure 7: Comparación de borrado para N = 1000.

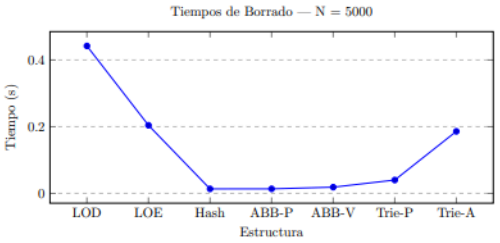

Figure 8: Comparación de borrado para N = 5000.

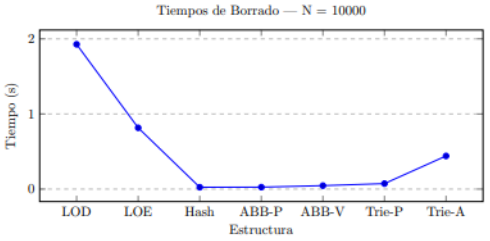

Figure 9: Comparación de borrado para N = 10000.

VIII. RESULTADOS Y ANÁLISIS

- **Inserción**

  Los resultados de inserción muestran diferencias notables en la eficiencia de las estructuras analizadas. Para tamaños pequeños (N = 1000), la mayoría de estructuras mantienen tiempos bajos, pero conforme aumenta el tamaño de los datos, las diferencias se amplifican. Se observa que la **Tabla Hash Abierta** y el **ABB con punteros** presentan los mejores tiempos de inserción, manteniéndose prácticamente estables incluso para N = 10000, gracias a sus complejidades promedio O(1) y O(logn), respectivamente. Por el contrario,

el **ABB con vector heap** evidencia un crecimiento exponencial en el tiempo de inserción, alcanzando el peor rendimiento entre todas las estructuras debido al costo de reorganización del vector. Los **tries** también muestran un incremento considerable en los tiempos, lo que sugiere un alto consumo de recursos conforme crece el número de elementos.

- **Búsqueda**

  En los gráficos de búsqueda se aprecia un comportamiento más estable en comparación con la inserción, aunque las diferencias de eficiencia siguen siendo evidentes. La **Tabla Hash Abierta** nuevamente presenta los menores tiempos, confirmando su rendimiento constante en operaciones de acceso, mientras que el **ABB con punteros** logra mantener un desempeño competitivo. Las **listas ordenadas**, tanto dinámica como estática, exhiben tiempos crecientes a medida que aumenta N, reflejando la necesidad de recorridos secuenciales o búsquedas binarias. Los **tries** se comportan adecuadamente para volúmenes moderados, pero su rendimiento tiende a degradarse en conjuntos de datos más grandes por la profundidad del árbol y el número de nodos requeridos. En general, la búsqueda es más eficiente en estructuras con índices jerárquicos o dispersión controlada.

- **Borrado**

  El análisis de la operación de borrado reafirma las tendencias observadas en las operaciones anteriores. El ABB con punteros y la Tabla Hash Abierta destacan por su consistencia y bajos tiempos, demostrando ser las estructuras más equilibradas para operaciones de actualización. En contraste, las **listas ordenadas** presentan un incremento notable en los tiempos de eliminación, ya que requieren desplazamientos o reasignaciones para mantener el orden. Los **tries** también evidencian una pérdida de eficiencia al eliminar nodos, especialmente en los casos con N mayores, debido al costo adicional de recorrer y limpiar las ramas del árbol. Por su parte, el **ABB con vector heap** continúa mostrando los peores resultados, confirmando su escasa adecuación para aplicaciones con altas tasas de borrado.

En conjunto, los resultados obtenidos permiten concluir que la **Tabla Hash Abierta** y el **Árbol Binario de Búsqueda (ABB) implementado con punteros** son las estructuras que presentan un rendimiento más eficiente y equilibrado en las tres operaciones evaluadas: inserción, búsqueda y borrado. Ambas mantienen tiempos bajos y una buena escalabilidad al aumentar el tamaño de los datos, validando sus complejidades promedio O(1) y O(logn). En contraste, estructuras como el **ABB con vector heap** y los **tries** mostraron un crecimiento desproporcionado en sus tiempos de ejecución, lo que refleja un alto costo computacional y menor eficiencia práctica. Finalmente, las **listas ordenadas**, aunque sencillas de implementar, resultaron menos adecuadas para volúmenes grandes de información debido a su comportamiento lineal en la mayoría de las operaciones. De esta forma, los resultados experimentales

confirman las predicciones teóricas sobre el desempeño esperado de cada estructura y evidencian la importancia de seleccionar el modelo de datos adecuado según el tipo de operación predominante y la magnitud del conjunto de datos.

IX. CONCLUSIONES

A lo largo del desarrollo de las tres etapas del proyecto, se implementaron, probaron y analizaron distintas estructuras de datos aplicadas al Modelo Diccionario, con el objetivo de comprender su comportamiento, eficiencia y aplicabilidad según las operaciones básicas de inserción, búsqueda y eliminación.

En la primera etapa, se abordaron las estructuras lineales, Lista Ordenada Dinámica, Lista Ordenada Estática y Tabla Hash Abierta, verificando la correcta ejecución de las operaciones del modelo y evaluando la uniformidad y desempeño de la función hash. Los resultados mostraron que las listas presentan un crecimiento lineal en tiempo de ejecución, mientras que la Tabla Hash Abierta ofrece un rendimiento más eficiente gracias a su acceso directo promedio O(1).

Durante la segunda etapa, el proyecto se amplió incorporando estructuras jerárquicas y de prefijos: el Árbol Binario de Búsqueda (ABB) en sus variantes por punteros y por vector heap, y el Trie en sus versiones por punteros y por arreglos. Estas implementaciones permiten comparar distintos enfoques de almacenamiento y organización de datos, demostrando que los árboles mejoran considerablemente los tiempos de búsqueda y borrado en comparación con las estructuras lineales, aunque con un mayor costo de implementación y uso de memoria.

Finalmente, en la tercera etapa, se desarrolló un programa de análisis experimental para medir los tiempos de ejecución y uso de espacio en diferentes tamaños de diccionario (N = 1000, 5000 y 10000). Los resultados empíricos confirman las expectativas teóricas:

- La Tabla Hash Abierta fue la más eficiente en todas las operaciones, manteniendo tiempos estables incluso con incrementos en N.
- El ABB por punteros mostró un rendimiento logarítmico, siendo una alternativa robusta y equilibrada.
- Las listas ordenadas perdieron eficiencia conforme aumentó el tamaño del conjunto de datos.
- Los Tries, aunque menos eficientes en tiempo, resultan ventajosos para búsquedas por prefijo o coincidencias parciales.

En términos globales, el proyecto evidenció la importancia de elegir adecuadamente la estructura de datos en función del tipo de operación y del volumen de información. Asimismo, permitió fortalecer el entendimiento práctico del análisis algorítmico, la complejidad temporal y espacial, y el impacto que la representación de los datos tiene sobre la eficiencia de los algoritmos.

X. REFERENCIAS

[1] G. “Abstract Data Types,” GeeksforGeeks, 28-Mar-2025. [En línea]. Disponible en: https://www.geeksforgeeks.org/dsa/abstract-data-types/.

[2] Google Sites, “La estructura Lista,” *Programación II - Unidad 3: Estructuras de datos comunes y colecciones*, [En línea]. Disponible en: https://sites.google.com/site/programacioniiuno/temario/unidad-3---estructuras-de-datos-comunes-y-colecciones/la-estructura-lista.

[3] WsCube Tech, “Singly Linked List Data Structure,” *Data Structures and Algorithms Resources*, [En línea]. Disponible en: https://www.wscubetech.com/resources/dsa/singly-linked-list-data-structure.

[4] The Data Structures Handbook, “Hash Tables,” *The DSA Handbook*, [En línea]. Disponible en: https://www.thedshandbook.com/hash-tables/.

[4] The Data Structures Handbook, “Hash Tables,” *The DSA Handbook*, [En línea]. Disponible en: https://www.thedshandbook.com/hash-tables/.

[5] Pulkit. *Basics of a Binary Search Tree Data Structure*. Medium. [En línea]. Disponible en:

https://medium.com/@goelpulkit43/b-is-for-binary-search-trees-3e1f82dd6064

[6] *What is Heap Data Structure? Types, Examples, Full Guide*. (s. f.). WsCube Tech, [En línea]. Disponible en: https://www.wscubetech.com/resources/dsa/heap-data-structure

[7] *TrIEs | Aprende programación competitiva*. (s. f.). , [En línea]. Disponible en: https://aprende.olimpiada-informatica.org/algoritmia-tries

Enlace del repositorio:

https://github.com/Meli290404/Proyecto1_Algoritmos.git