Tarea programada 1, Segundo Semestre.

Segundo Avance. 2025.

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

Resumen. *En esta segunda etapa del proyecto se amplió el Modelo Diccionario incorporando nuevas estructuras de datos jerárquicos y de prefijos, con el objetivo de analizar diferentes enfoques para las operaciones de inserción, búsqueda y eliminación. Se implementaron el Árbol de Búsqueda Binaria (ABB) en sus versiones por punteros y por vector heap, así como el Trie en sus variantes por punteros y por arreglos. Todas las estructuras fueron integradas al programa interactivo desarrollado en la primera etapa, permitiendo su validación funcional y comparativa. Este avance nos permitió sentar las bases para el análisis experimental de eficiencia y rendimiento que se abordará en la siguiente fase del proyecto.*

Palabras clave: *estructuras de datos, diccionario, lista ordenada, lista enlazada, arreglos, tabla hash, inserción, búsqueda, eliminación, complejidad temporal, complejidad espacial, Estructuras de datos, modelo diccionario, lista ordenada, lista enlazada, arreglos, tabla hash, árbol de búsqueda binaria (ABB), árbol binario por punteros, árbol binario por vector heap, trie, trie por punteros, trie por arreglos, inserción, búsqueda, eliminación, rehash, uniformidad, eficiencia, 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 este segundo avance se amplía el estudio del Modelo Diccionario incorporando estructuras jerárquicas y de prefijos, específicamente el Árbol de Búsqueda Binaria (ABB) —en sus versiones por punteros y por vector heap— y el Trie —tanto por punteros como por arreglos—. Estas implementaciones permiten comparar distintos enfoques de almacenamiento y acceso a la información, analizando su comportamiento en operaciones de inserción, búsqueda, borrado y recorrido.

Asimismo, se reutiliza el mismo programa de prueba desarrollado en la primera etapa, el cual facilita la validación y depuración de las nuevas estructuras bajo una interfaz interactiva. Con ello, se consolida una base sólida para la tercera etapa, enfocada en el análisis de rendimiento y la comparación experimental de las estructuras implementadas.

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*).

Para esta segunda etapa, se ampliaron las implementaciones con estructura como:

- Árbol de Búsqueda Binaria (ABB) por punteros, con nodos enlazados para inserciones, búsquedas y eliminaciones eficientes.
- ABB por vector heap, que representa el árbol en un arreglo lineal mediante relaciones de índice.
- Trie por punteros y Trie por arreglos, diseñados para almacenar cadenas de caracteres compartiendo prefijos comunes.

Todas las estructuras fueron integradas en un programa interactivo que permite seleccionar la implementación deseada y ejecutar las operaciones del diccionario bajo una interfaz visual desarrollada con la librería rich.

IV. DISCUSIÓN

En esta segunda etapa, el Modelo Diccionario se consolida como una abstracción flexible que permite incorporar nuevas estructuras jerárquicas y de prefijos, manteniendo un conjunto uniforme de operaciones. El objetivo fue extender la funcionalidad del modelo para representar y analizar estructuras más complejas, como el Árbol de Búsqueda Binaria (ABB) y el Trie, cada una con variantes que exploran distintos enfoques de almacenamiento y acceso a la información.

El modelo abstracto, implementado con Python mediante la clase base Diccionario, garantiza que todas las estructuras hereden e implementen las operaciones fundamentales como: Init, Done, Clear, Insert, Delete, Member y Print bajo una misma interfaz. Esto permite que el programa interactivo ejecute las mismas pruebas independientemente de la implementación interna utilizada.

En esta fase, se desarrollaron dos versiones del ABB: una por punteros, basada en nodos enlazados que facilitan el manejo dinámico de la memoria, y otra por vector heap, que representa el árbol dentro de un arreglo mediante relaciones de índice. De igual forma, se implementaron dos variantes del Trie, estructura orientada al almacenamiento eficiente de cadenas: el Trie por punteros, que crea nodos dinámicos a demanda, y el Trie por arreglos, que utiliza índices fijos para un acceso constante.

Estas nuevas implementaciones mantienen la independencia entre interfaz

y representación, permitiendo que el diccionario conserve su uniformidad funcional. Además, su integración en el mismo programa de prueba interactivo facilita la validación y comparación experimental, evidenciando las ventajas y limitaciones de cada enfoque en términos de eficiencia, complejidad temporal y uso de memoria.

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]

VI. REFERENCIAS

[1] G. “Abstract Data Types,” GeeksforGeeks, 28-Mar-2025. [En línea]. Disponible en: https://www.geeksforgeeks.org/dsa/abstrac t-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/programacioni iuno/temario/unidad-3---estructuras-de-dat os-comunes-y-colecciones/la-estructura-lis ta.

[3] WsCube Tech, “Singly Linked List Data Structure,” *Data Structures and Algorithms Resources*, [En línea]. Disponible en: https://www.wscubetech.com/resources/ds a/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/.

[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