Fundamentos de Estructuras de Datos y Algoritmos **Semestre 2025-1**

Tarea n°3

Profesor: José Fuentes *Alumno: Fabián Ortiz*

1 Introducción

1.1 Introducción

El presente informe tiene como objetivo comparar el rendimiento de distintas estructuras de datos para operaciones de almacenamiento y búsqueda sobre un conjunto de 40.000 usuarios de Twitter, extraídos del dataset "TwitterFriends" disponible en Kaggle. Para ello, se implementaron tres estructuras: un Árbol Binario de Búsqueda (BST) sin balanceo, una Tabla Hash con colisiones resueltas mediante hashing abierto (con lista), y una Tabla Hash con hashing cerrado utilizando linear probing.

Las pruebas consideran dos tipos de claves para acceder a los usuarios: el identificador numérico "id" y el nombre de usuario "screenName". Todas las estructuras fueron desarrolladas en lenguaje C++, utilizando bibliotecas estándar.

Para evaluar el rendimiento, se midieron tiempos de inserción y búsqueda (exitosa y fallida) de usuarios, considerando distintos tamaños de entrada (5.000, 10.000, 15.000, 20.000,...,40.000). Además, se exportaron los resultados a archivos .csv para su posterior análisis gráfico. Las búsquedas se ejecutaron sobre subconjuntos válidos del dataset, previamente filtrados y limpiados mediante procesos de parsing y validación.

Los experimentos se realizaron en un entorno controlado, midiendo los tiempos con precisión de nanosegundos usando la biblioteca "chrono". Como resultado, este informe entrega evidencia empírica que permite contrastar el comportamiento de las estructuras en distintos escenarios, presentando ejemplos que aporten a la elección de estructuras de datos.

2 Estructuras de datos

2.1 Árbol Binario de Búsqueda (BST)

El árbol binario de búsqueda implementado almacena objetos del tipo Usuario, y utiliza el campo "id" o "screenName" como clave de ordenamiento. La inserción y búsqueda se realizan de forma recursiva, comparando la clave del nuevo elemento con la clave del nodo actual, desplazándose hacia la izquierda o derecha según corresponda. No se implementa ningún tipo de balanceo, por lo que el rendimiento puede llegar a O(n) en el peor caso. Aunque el árbol binario de búsqueda no implementa ningún mecanismo de balanceo, se espera que en promedio su rendimiento para inserción y búsqueda sea del orden O(log n) cuando los datos están suficientemente aleatorizados y el árbol no se degrada en una estructura lineal. En este proyecto, los datos utilizados presentan una distribución variada tanto en "id" como en "screenName", por lo que se asume que el árbol tiende a mantener una forma razonablemente equilibrada. La estructura fue desarrollada en C++ y se implementaron dos variantes: una por "id" y otra por "screenName".

2.2 Tabla Hash (hashing abierto)

La tabla hash con hashing abierto fue implementada utilizando encadenamiento con listas enlazadas. Cada posición del arreglo principal contiene una lista de usuarios. La función hash utilizada es la estándar de C++ (`std::hash`) aplicada sobre el campo "id" o "screenName", según la clave usada. La complejidad promedio de inserción y búsqueda es O(1), aunque puede aumentar a O(n) en casos de muchas colisiones. Se utilizó un tamaño de tabla suficientemente grande (55.001) para mantener bajo el factor de carga y minimizar colisiones.

2.3 Tabla Hash (hashing cerrado)

La tabla hash con hashing cerrado fue implementada utilizando la técnica de linear probing para resolver colisiones. Cada posición del arreglo contiene opcionalmente un usuario, y si la posición está ocupada, se avanza secuencialmente hasta encontrar una libre o la clave buscada. Al igual que en la versión abierta, se utilizó la función estándar de C++ (`std::hash`) sobre "id" o "screenName". El tamaño de la tabla fue de 80.021, permitiendo mantener un factor de carga bajo (menor al 0.5) y prevenir ciclos infinitos en la exploración. La complejidad promedio es O(1), pero puede aumentar si el factor de carga es alto.

3 Dataset y experimentos

El dataset utilizado para este informe es "TwitterFriends", disponible públicamente en la plataforma Kaggle. Contiene 40.000 registros de usuarios de Twitter con atributos como id, screenName, cantidad de seguidores, amigos, idioma, último tweet, entre otros. Para este trabajo, se utilizaron exclusivamente los campos "id" y "screenName" como claves para las estructuras de datos, aunque se conservaron los demás atributos como parte del objeto Usuario.

La lectura del archivo se realiza desde un archivo CSV llamado "data.csv", el cual se parsea línea a línea utilizando funciones personalizadas. Estas rutinas se encargan de limpiar campos vacíos, convertir tipos de datos y validar que los registros sean utilizables. Sólo se almacenan usuarios que tienen un id válido (mayor a cero) y un nombre de usuario no vacío. El proceso de lectura filtra además el número máximo de usuarios según el parámetro definido en tiempo de ejecución (hasta 40.000).

Los experimentos fueron ejecutados en un computador Apple MacBook Air con chip Apple M1 (8 núcleos), acompañado de 8 GB de memoria RAM unificada. El sistema operativo utilizado fue macOS Ventura 13.6, y el código fue compilado mediante el compilador estándar de C++ con soporte para C++17. Para las mediciones de tiempo se utilizó la biblioteca "chrono", incluida en el estándar de C++, que permite medir con precisión a nivel de nanosegundos. No se usaron herramientas de paralelización ni bibliotecas externas.

4 Resultados

A continuación se presentan los resultados obtenidos en los experimentos de inserción y búsqueda para cada estructura de datos, utilizando las claves "id" y "screenName". Se comparan tanto los tiempos de construcción de las estructuras como el rendimiento en búsquedas exitosas y fallidas.

4.1 Inserción

Árbol Binario de Búsqueda (BST)

El gráfico muestra que la inserción con la clave "screenName" es consistentemente más costosa que con "id". Esto se debe a que las comparaciones de cadenas de texto son más lentas que las comparaciones numéricas. Ambas curvas son crecientes y regulares, lo cual refleja la naturaleza no balanceada del árbol, pero sin degradación extrema en este caso.

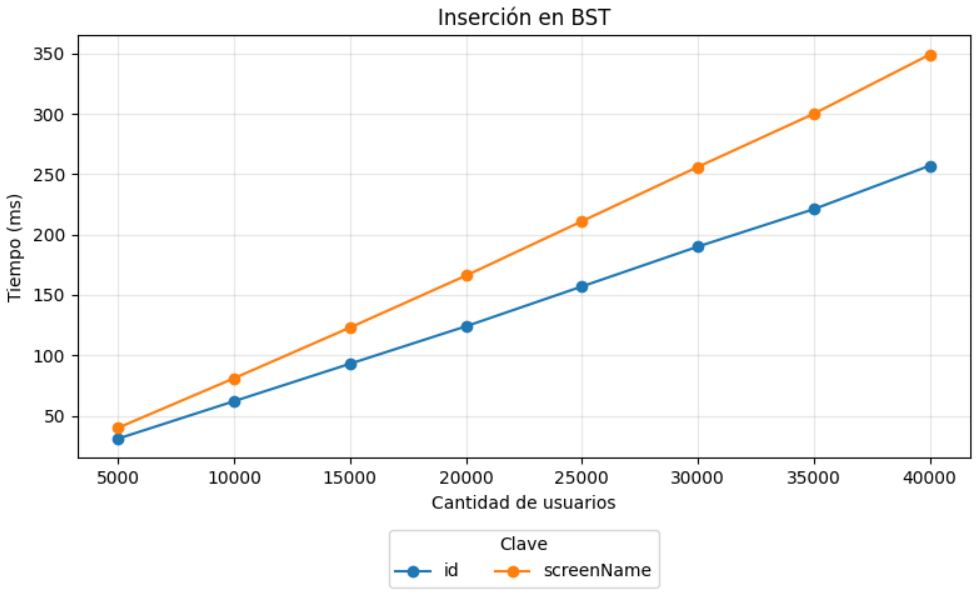


Tabla Hash (Hashing Abierto)

Se observa que los tiempos de inserción para ambas claves son muy similares y crecen de forma lineal. Esto sugiere una distribución uniforme de las claves y un bajo nivel de colisiones gracias al tamaño adecuado de la tabla. La diferencia entre "id" y "screenName" es mínima.

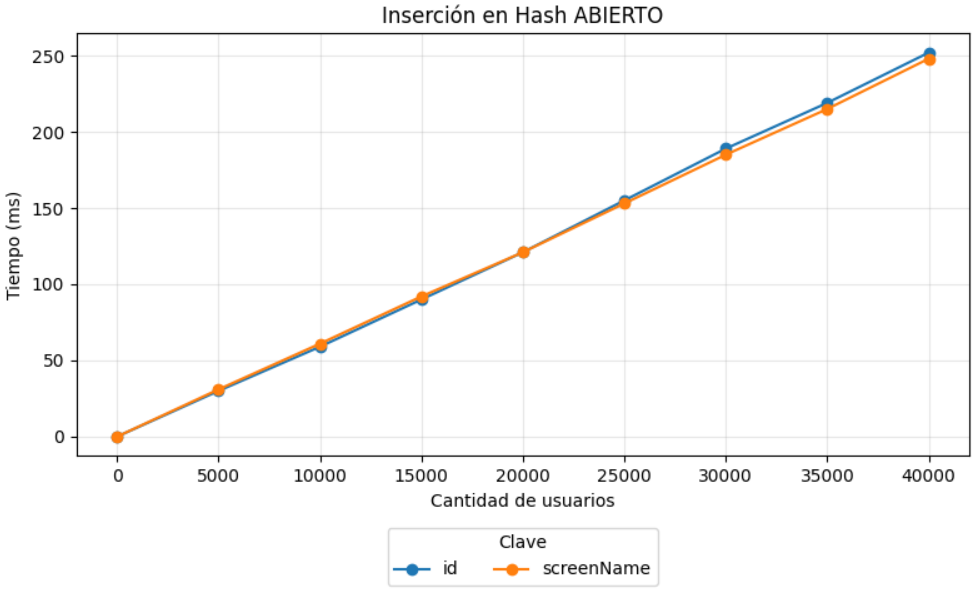
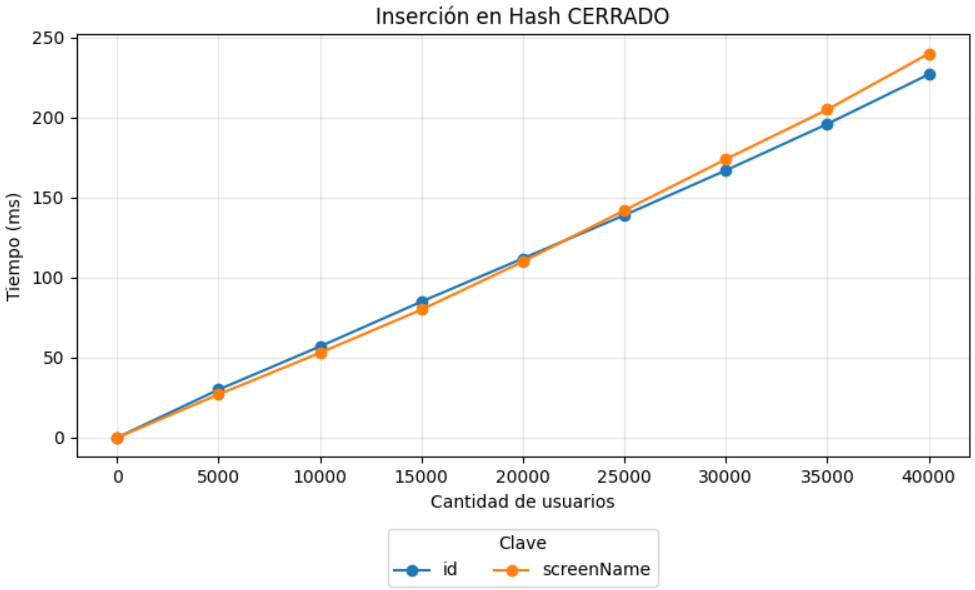


Tabla Hash (Hashing Cerrado)

En este caso, los tiempos de inserción también son similares entre claves, aunque "screenName" presenta un leve aumento respecto a "id". Ambos tiempos escalan linealmente, lo que indica un buen control del factor de carga y una resolución eficiente mediante linear probing.



4.2 Búsqueda

Árbol Binario de Búsqueda (BST)

Las búsquedas exitosas con "id" son más rápidas que con "screenName", mientras que las búsquedas fallidas con "id" se mantienen consistentemente bajas. Las búsquedas fallidas con "screenName" son estables, aunque más lentas que las exitosas con "id". Esto confirma que las comparaciones textuales tienen mayor costo.

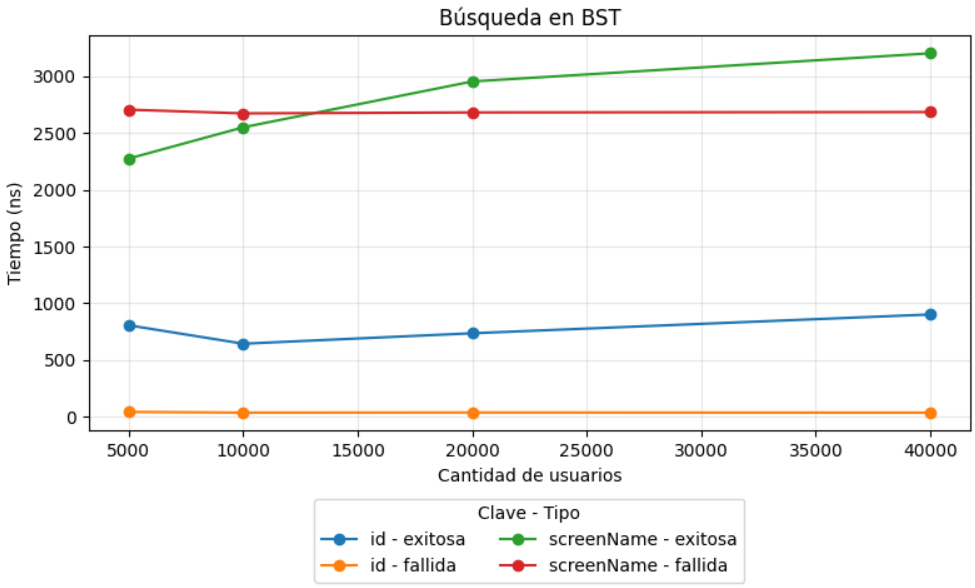


Tabla Hash (Hashing Abierto)

Las búsquedas exitosas con "id" presentan los mejores tiempos. En cambio, las búsquedas fallidas con "screenName" son más costosas, lo cual sugiere mayor cantidad de colisiones o cadenas más largas en las listas de encadenamiento. Aún así, los tiempos se mantienen en rangos aceptables.

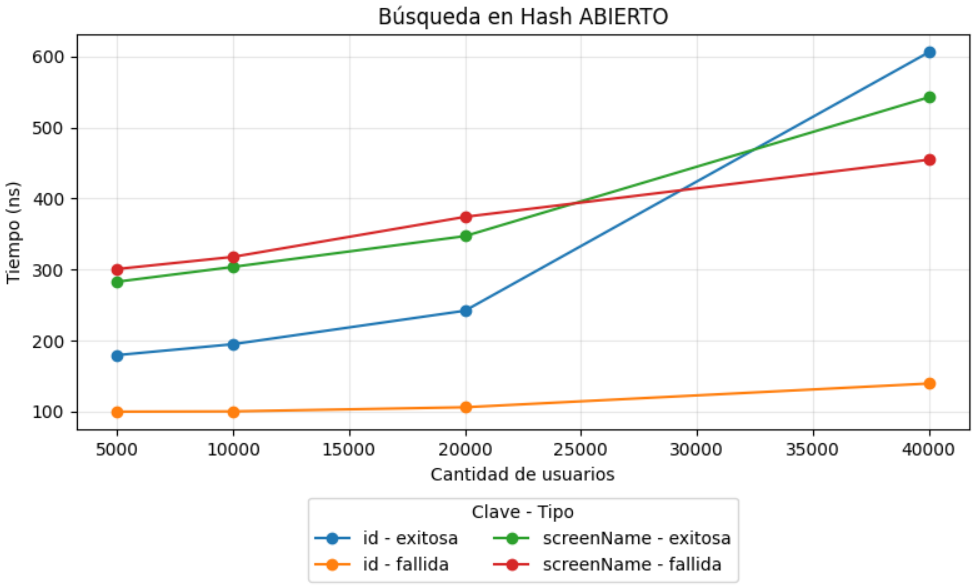
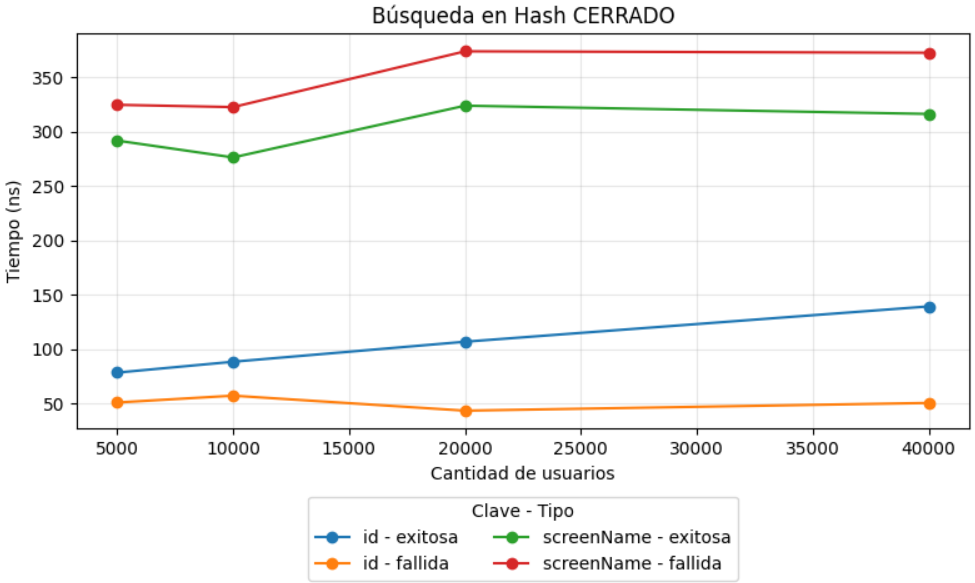


Tabla Hash (Hashing Cerrado)

El rendimiento es más estable que en el caso anterior. "id" sigue siendo más eficiente que "screenName", especialmente en búsquedas exitosas. Las búsquedas fallidas con "screenName" son las más costosas, debido a la exploración lineal de posiciones consecutivas en caso de colisión.



5 Notas sobre el desarrollo

Durante el desarrollo de este proyecto, se recurrió en algunas instancias al apoyo de un modelo de lenguaje (GPT-4) para resolver problemas puntuales derivados de limitaciones de tiempo y reestructuraciones necesarias del proyecto.

En particular, se utilizó asistencia generada por GPT-4 para:

- Crear estructuras base para los archivos `utilitarios.cpp`, `parseo.cpp` y `graficos generar.py`.
- Reorganizar la arquitectura general del proyecto, luego de que el crecimiento no planificado del mismo derivara en una estructura difícil de mantener.
- Resolver errores de compilación y advertencias asociadas al uso de bibliotecas, manejo de dependencias, y compatibilidades de hardware (especialmente en entornos macOS y Linux).

Cabe señalar que todas las ideas, pruebas, modificaciones, análisis experimentales y conclusiones fueron realizadas de manera autónoma, asegurando el entendimiento completo de cada módulo y del comportamiento general del sistema.

Esta decisión se tomó en aras de mantener el foco del trabajo en el diseño de estructuras de datos, el análisis empírico de su rendimiento, y en el cumplimiento de los objetivos planteados en el informe.

6 Conclusiones

En los tiempos actuales, donde el acceso y procesamiento de datos es parte fundamental del desarrollo tecnológico, elegir bien la estructura de datos que se va a utilizar no es solo un detalle técnico, sino una decisión que puede marcar la diferencia en el rendimiento y escalabilidad de un sistema. Este trabajo se enfocó en comparar tres estructuras: un árbol binario de búsqueda (sin balanceo), una tabla hash con hashing abierto y otra con hashing cerrado, aplicadas sobre un conjunto real de usuarios, utilizando como claves tanto el "id" como el "screenName".

Los resultados muestran con claridad que las tablas hash, en la mayoría de los casos, superan al árbol binario en cuanto a eficiencia, especialmente cuando se utiliza "id" como clave. Esto tiene sentido si consideramos que las funciones hash sobre números enteros tienden a ser más rápidas y menos propensas a colisiones que aquellas aplicadas sobre strings. Además, al no tener balanceo, el árbol puede degradarse en rendimiento si los datos no están bien distribuidos (lo que no siempre es controlable en datos reales).

Entre las dos versiones de tablas hash, la abierta se comportó de forma más estable, mientras que la cerrada (aunque rápida en muchos casos) mostró pequeñas caídas de rendimiento cuando se realizaron búsquedas fallidas con "screenName". Esto probablemente se deba al tipo de resolución de colisiones que utiliza (linear probing), que puede llevar a explorar varias posiciones si hay muchos elementos similares.

Desde el punto de vista práctico, este tipo de análisis permite tomar decisiones más informadas. Usar "id" como clave no solo es más eficiente, sino también más seguro en términos de rendimiento constante. Y optar por una tabla hash abierta (al menos en este escenario) parece ser una excelente alternativa cuando se quiere velocidad sin perder simplicidad en la implementación.

En conclusión, la estructura más eficiente en este estudio fue la tabla hash abierta utilizando "id" como clave. Este tipo de evidencia es valiosa no solo para entender cómo funcionan estas estructuras en teoría, sino también para ver cómo se comportan en la práctica, cuando hay que trabajar con datos reales y decisiones concretas.