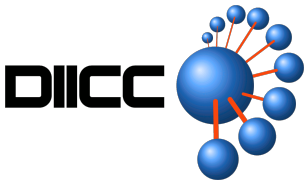


FUNDAMENTOS DE ESTRUCTURAS DE DATOS
Y ANÁLISIS DE ALGORITMOS

INFORME N°3

"Diccionarios, Tablas Hash vs Árboles de búsqueda"

Junio 2023

ERICH GERMÁN GRÜTTNER DÍAZ

Índice

1. Introducción

Este informe tiene como objetivo presentar un análisis sobre las diferencias de funcionamiento, rendimiento e implementación de árboles de búsqueda versus el uso de tablas hash con hashing abierto y cerrado.

En la primera parte se describen las estructuras a utilizar y algunos aspectos relevantes en el proceso de construcción del software, tales como el uso de librerías y generación de gráficos. En particular para el caso de Árbol binario de búsqueda se utilizará la librería AVLTree que promete una complejidad de tiempo del orden de O(Log n).

Luego, se describen los diferentes datasets utilizados tanto de input como de output. El contexto del presente proyecto es el análisis de información de usuarios de Twitter que siguen a universidades chilenas. Se tomó como base inicial un archivo que contiene esa información, y luego se parceló en 5 archivos más pequeños de 1000, 5000, 10000, 15000 y 20000 registros.

En el apartado de resultados experimentales, se muestran datos y gráficos resultantes de la ejecución de las diversas tareas en el software. Para ello se utilizó la librería “Chrono” de C++. En tanto que los gráficos se obtuvieron a través de Python usando la librería “Matplotlib”.

Otro punto importante en el análisis de este informe es el uso de diferentes tipos de clave en la inserción y búsqueda de información en las distintas estructuras de datos. En particular se realizarán pruebas con una clave de tipo long (User_Id) y de tipo string (User_Name). Se verá el impacto en algunas tareas analizadas.

Finalmente se presentan las conclusiones. Estas abarcan desde temas puntuales de las estructuras (Ej: rendimiento de claves User_Id vs User_Name) como también comparaciones de rendimiento entre estructuras (Ej: Inserción usando clave User_Id para árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado).

Este informe se complementa con un repositorio Github en donde se puede encontrar tanto el código utilizado en los diferentes experimentos, como los diferentes archivos de salida y gráficos de análisis correspondientes.

2. Descripción de las estructuras de datos

2.1 Árbol binario de búsqueda

Un árbol de búsqueda binario (BST, Binary Tree Search), es una estructura de datos jerárquica y ordenada que se utiliza para almacenar y organizar datos de manera eficiente, permitiendo una búsqueda rápida. [1]

En un árbol binario de búsqueda, cada nodo contiene un valor único y se divide en un subárbol izquierdo y uno derecho. La clave característica de estas estructuras es que, para cada nodo, todos los valores en el subárbol izquierdo son menores que el valor del nodo, y todos los valores en el subárbol derecho son mayores. Esta propiedad permite realizar búsquedas de manera eficiente.

La estructura organizada de un Árbol binario de búsqueda se basa en el principio de "dividir para conquistar". Al realizar una búsqueda, se compara el valor buscado con el valor del nodo actual. Si es igual, se ha encontrado el elemento buscado. Si es menor, se sigue el subárbol izquierdo. Si es mayor, se sigue el subárbol derecho. Este proceso continúa hasta encontrar el valor deseado o llegar a una hoja, donde se determina que el valor no existe en el árbol.

La principal ventaja de los BST es que permiten búsquedas eficientes en tiempo logarítmico, lo que significa que el tiempo necesario para encontrar un elemento aumenta de manera gradual a medida que el tamaño del árbol crece. Además, los BST también permiten otras operaciones útiles como inserción, eliminación y recorrido ordenado de los elementos.

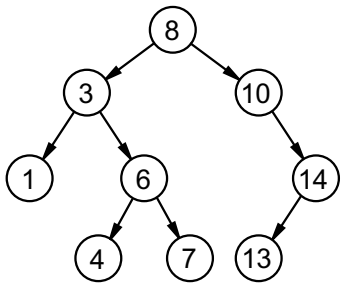


Figura 1 - Ejemplo de árbol binario de búsqueda

Sin embargo, es importante destacar que la eficiencia de un BST depende de su estructura y puede degradarse si el árbol está desequilibrado. En el peor de los casos, un árbol desequilibrado puede tener un rendimiento similar a una lista enlazada, lo que anula las ventajas de la búsqueda rápida. Por lo tanto, es necesario mantener el equilibrio del árbol para asegurar un rendimiento óptimo, para lo cual existen técnicas de balanceo, como el árbol AVL y el árbol rojo-negro. [2] [3]

2.2 Tabla Hash

Es es una estructura de datos que implementa el tipo de dato abstracto llamado **diccionario**. Esta asocia **llaves** o **claves** con **valores**. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos almacenados a partir de una clave generada. Funciona transformando la clave con una función hash en un hash, un número que identifica la posición (casilla o bucket) donde la tabla hash localiza el valor deseado. [4]

Comparada con otras estructuras de arreglos asociadas, las tablas hash son más útiles cuando se almacenan grandes cantidades de información.

Las funciones hash más utilizadas son "hash de división" y "hash de multiplicación".

Si dos llaves generan un hash apuntando al mismo índice, los registros correspondientes no podrán ser almacenados en la misma posición. En estos casos, cuando una casilla ya está ocupada, se debe encontrar otra ubicación donde almacenar el nuevo registro, y hacerlo de tal manera que pueda ser encontrado cuando se requiera.

En este informe se utilizarán las técnicas de hashing abierto y cerrado, que serán descritas a continuación:

2.2.1 Tabla Hash con hashing abierto

En la técnica más simple de encadenamiento, cada casilla en el arreglo referencia a una lista con los registros insertados que colisionan en dicha casilla. La inserción consiste en encontrar la casilla correcta y agregar al final de la lista correspondiente. El borrado consiste en buscar y quitar de la lista.

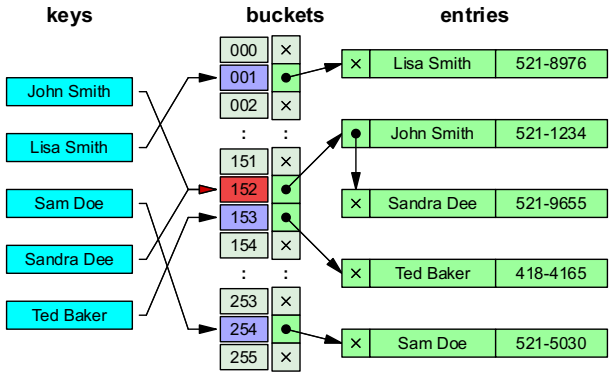


Figura 2-Ejemplo Tabla Hash con hashing abierto

2.2.2 Tabla Hash con hashing cerrado

Las tablas hash de hashing abierto pueden almacenar los registros directamente en el array. Las colisiones se resuelven mediante un **sondeo** del arreglo, en el que se buscan diferentes posiciones dentro del arreglo (secuencia de sondeo) hasta que el registro es encontrado o se llega a una casilla vacía, indicando que no existe esa llave en la tabla.

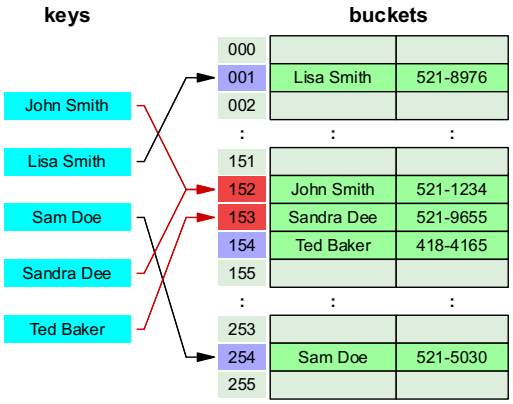


Figura 3-Ejemplo Tabla Hash con hashing cerrado

Las secuencias de sondeo más utilizadas son:

1. **Sondeo lineal:** El intervalo entre sondeos es fijo (usualmente 1)
2. **Sondeo cuadrático:** El intervalo entre sondeos se incrementa agregando las sucesivas salidas de un polinomio cuadrático al valor dado por la función hash original
3. **Doble hasheo:** El intervalo entre sondeos es calculado por otra función hash

3. Código y documentación

El código de la aplicación construida se encuentra en el repositorio [5]

3.1 Implementación de árbol binario

Se utilizó la librería “AVL-Tree”, desarrollada por Kadir Emre Oto en 2018 [6]

Según su autor, el AVL Tree que implementa esta librería corresponde a un árbol binario auto-balanceado, que garantiza un complejidad de tiempo O(Log N) para la inserción, borrado y búsqueda. Para mostrar el árbol y limpiarlo, en cambio, se garantiza una complejidad de tiempo de O(n).

Permite almacenar cualquier tipo de dato que permita la operación “mayor que (>)”. En particular para este informe se utilizarán long y string.

3.2 Implementación de tabla hash con hashing abierto

Se utilizó un código base generado por ChatGPT y luego modificado y adaptado a las necesidades de la tarea.

Fueron creadas clases separadas para manejar User_Id y User_Name, con sus tipos de datos propios (long y string, respectivamente).

Para el caso de string, la función transforma cada elemento del dato de entrada a int multiplicando por 31 (número primo). Con el resultado se realiza la división entera contra el tamaño de la tabla. Lo que es considerado un Hash de división.

Se utilizó una tabla de tamaño 10000.

3.3 Implementación de tabla hash con hashing cerrado

Se utilizó un código base generado por ChatGPT y luego modificado y adaptado a las necesidades de la tarea.

Fueron creadas clases separadas para manejar User_Id y User_Name, con sus tipos de datos propios (long y string, respectivamente).

Para el caso de string, toma cada elemento del dato, lo convierte a int y lo suma. Con el resultado se realiza la división entera contra el tamaño de la tabla. Lo que es considerado un Hash de división.

Para el manejo de colisiones se utilizó **“Sondeo Lineal”.**

Se utilizó una tabla de tamaño 30000.

3.4 Medición de tiempos de ejecución

Se utilizó la librería “Chrono” de STL, para medir la ejecución del código.
Cada tarea ejecutada fue medida en base a un promedio de ejecución de 10 intentos.
Cada ejecución se almacena en un archivo CSV con los tiempos para cada tamaño de dataset.

3.5 Construcción del código

Se realizó en base a **make**, lo que permite realizar ejecuciones programáticas en base a parámetros de entrada del programa.
Adicionalmente, se incorporó un archivo de tipo Bash para ejecutar otras tareas como limpieza y carga del programa.

3.6 Generación de gráficos

Finalmente, con los datos obtenidos en formato CSV, se utilizó un script en lenguaje Python, que utiliza la librería **MatPlotLib** [7] y que permite generar gráficos customizados de acuerdo a las pruebas requeridas por el proyecto.

4. Datasets y diseño experimental

4.1 Archivo base

El proyecto se basa en procesar datos sobre el dominio de usuarios de Twitter. Para ello se dispuso del archivo **"universities_followers_2022.csv.zip"** el cual consiste en 29.245 usuarios de Twitter que son seguidores de la cuenta oficial de alguna universidad chilena en la red social.

4.2 Archivos de input – Inserción

Para realizar pruebas parciales, se crearon copias del archivo base de distinto tamaño y peso [8]

Archivo	Cantidad de registros	Peso
Input1.csv	1000	85KB
Input2.csv	5000	418KB
Input3.csv	10000	816KB
Input4.csv	15000	1,2MB
Input5.csv	20000	1,7MB

Inputx.csv

university;user_id;user_name;number_tweets;friends_count;followers_count;created_at
Dato 1
...
Dato n

4.3 Archivos para realizar búsquedas

Archivo	Cantidad de registros	Peso
Input1.csv	100	8KB
Input2.csv	200	17KB
Input3.csv	300	25KB
Input4.csv	400	34KB
Input5.csv	500	42KB

Inputx.csv

university;user_id;user_name;number_tweets;friends_count;followers_count;created_at
Dato 1
...
Dato n

Para el caso de datos no “encontrables” se aplicó el siguiente algoritomo

- Si se procesa User_Id, se toma el dato desde el registro de datos “encontrables” y se le aumenta un número de su valor
- Si se procesa User_Name, se toma el dato desde el registro datos “encontrables” y se le concatena una letra “X” para que no aparezca en el resultado

4.4 Archivos para graficar

Para facilitar la generación de gráficos para la observación del rendimiento de los algoritmos, la ejecución del programa genera archivos del tipo CSV con el siguiente formato:

tarea_results.csv

Tamaño muestra, Tiempo[ms]
valor 1, tiempo 1
...
valor n, tiempo n

Estos archivos se pueden acceder en el repositorio en [9]

4.5 Diseño experimental

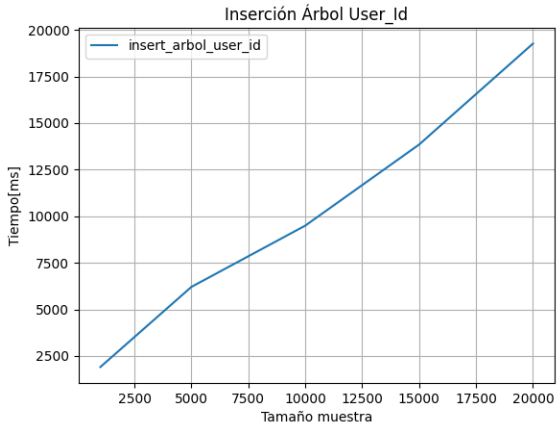
Para la realización de las pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria. El chip M1 tiene 8 núcleos (4 de alta eficiencia a 3.2 GHz + 4 de alto rendimiento a 2.0 GHz) y una velocidad de transferencia de 50Gb por segundo.

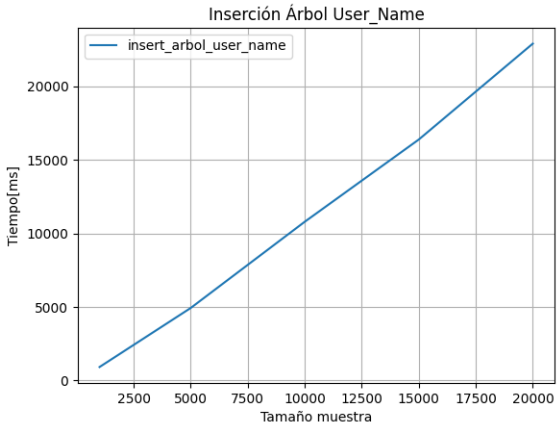
5. Resultados experimentales

5.1 Mediciones de rendimiento usando Árbol de búsqueda

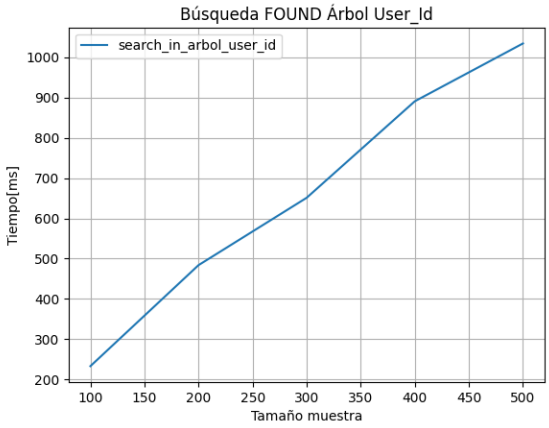
Se realizaron ejecuciones separadas en base al índice utilizado. A nivel de código implicó instanciar dos tipos de árboles, uno que gestiona datos de tipo **long (User_Id)**, y el otro que gestiona datos de tipo **string (User_Name)**.

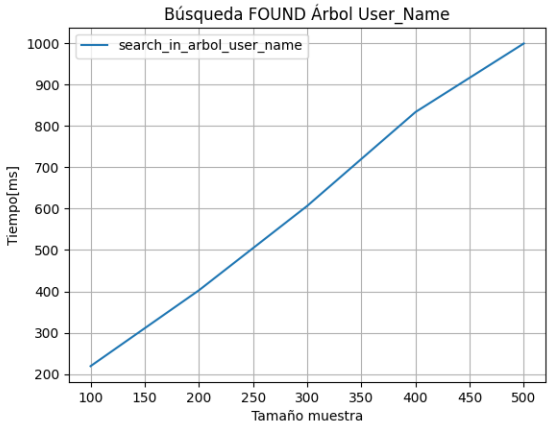
5.1.1 Árbol - Inserción usando User_Id – User_Name



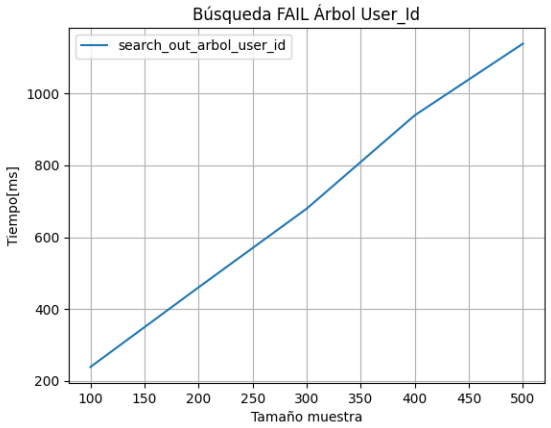


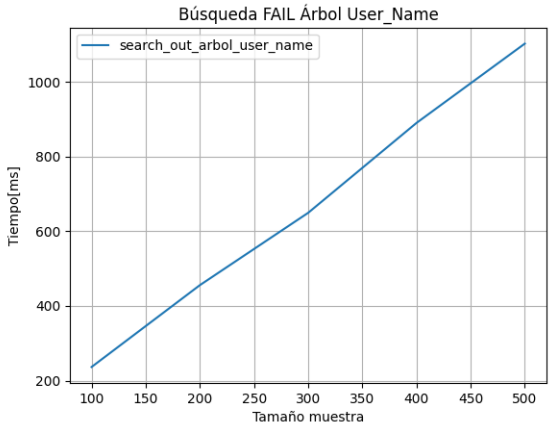
5.1.2 Árbol - Búsqueda de datos cargados, usando User_Id – User_Name





5.1.3 Árbol - Búsqueda de datos no cargados, usando User_Id - User_Name

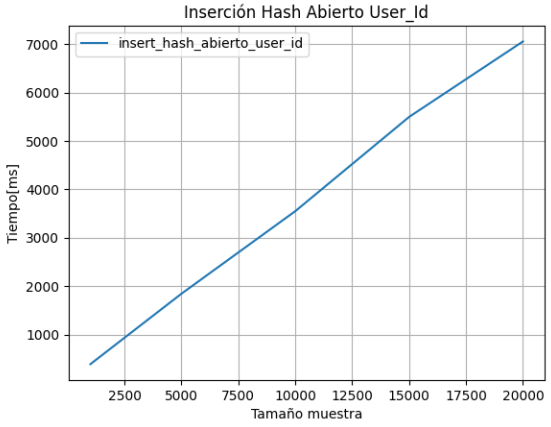


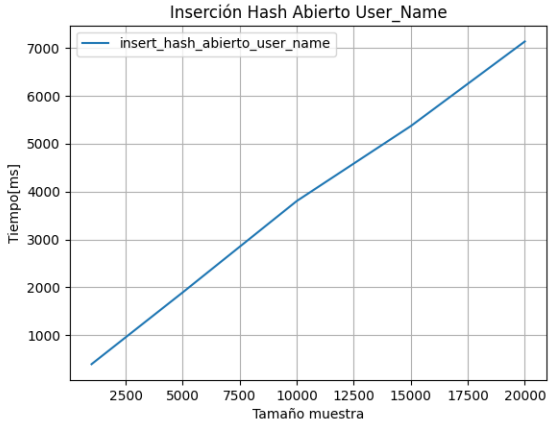


5.2 Mediciones de rendimiento usando Tabla Hash con hashing abierto

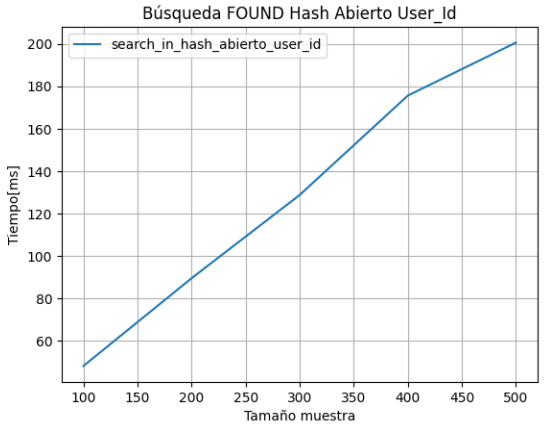
Se realizaron ejecuciones separadas en base al índice utilizado. A nivel de código implicó instanciar dos tipos de tablas, uno que gestiona datos de tipo **long (User_Id)**, y el otro que gestiona datos de tipo **string (User_Name)**.

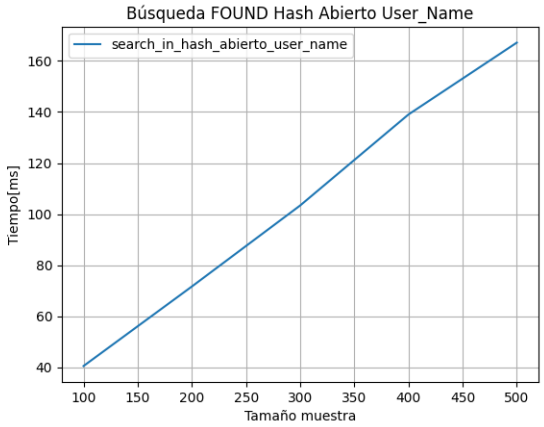
5.2.1 Hashing abierto – Inserción usando User_Id – User_Name



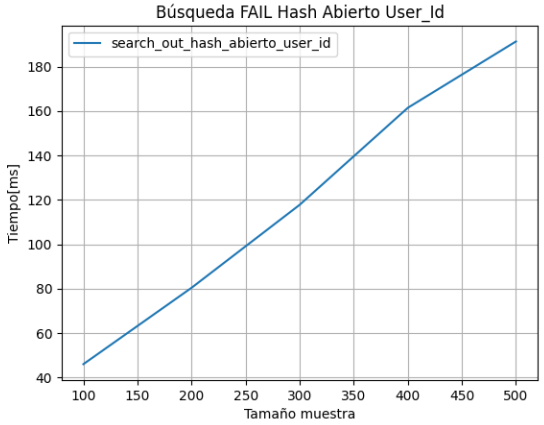


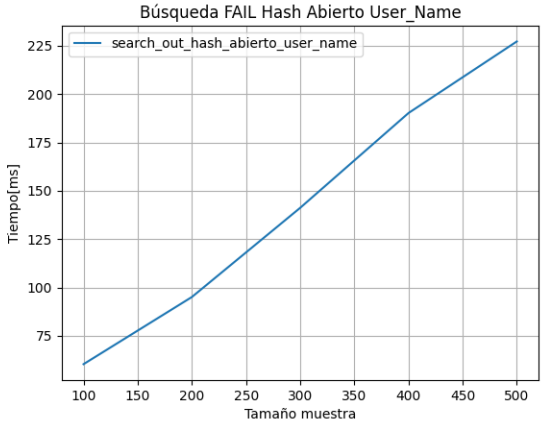
5.2.2 Hashing abierto - Búsqueda cargados, usando User_Id – User_Name





5.2.3 Hashing abierto - Búsqueda no cargados, usando User_Id – User_Name

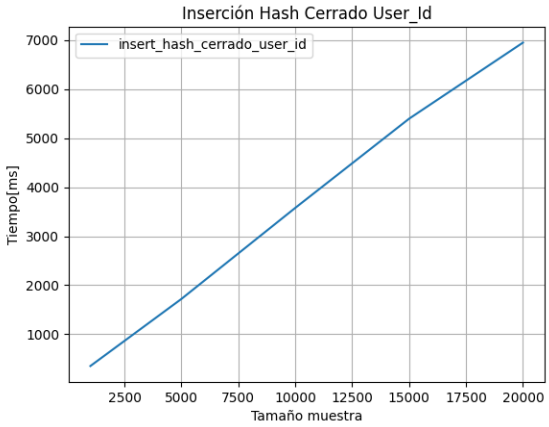


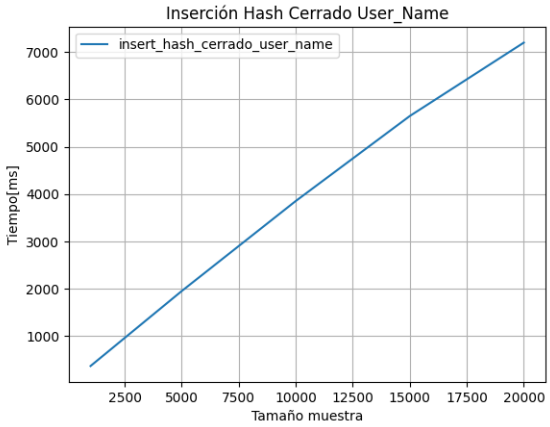


5.3 Mediciones de rendimiento usando Tabla Hash con hashing cerrado

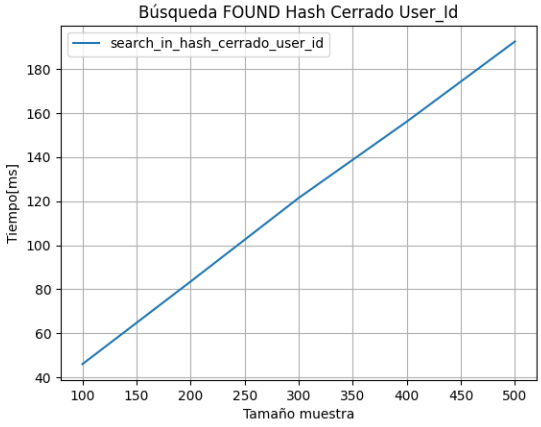
Se realizaron ejecuciones separadas en base al índice utilizado. A nivel de código implicó instanciar dos tipos de tablas, uno que gestiona datos de tipo **long (User_Id)**, y el otro que gestiona datos de tipo **string (User_Name)**.

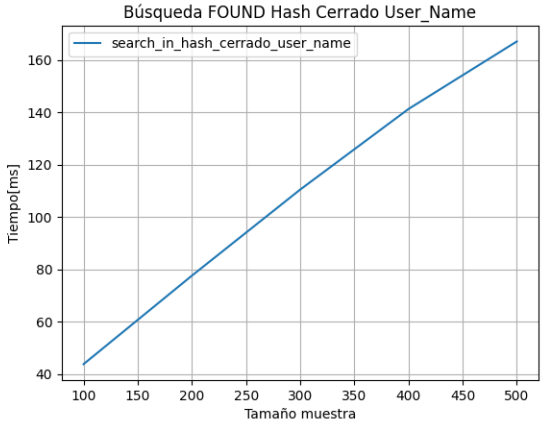
5.3.1 Hashing cerrado – Inserción usando User_Id – User_Name



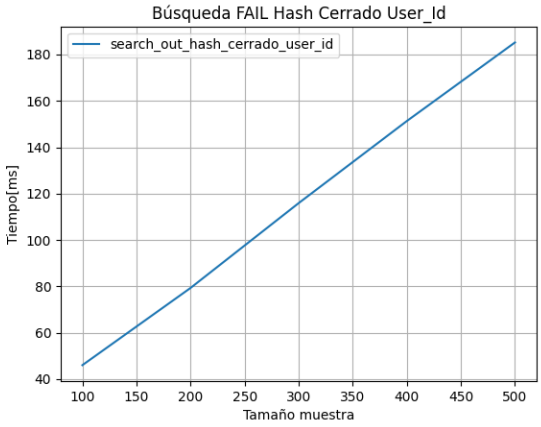


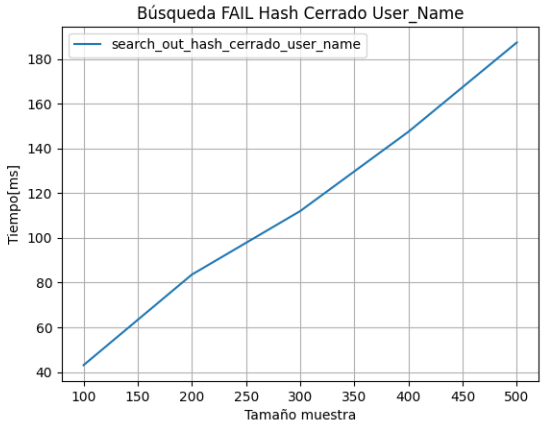
5.3.2 Hashing cerrado - Búsqueda cargados, usando User_Id – User_Name





5.3.3 Hashing cerrado - Búsqueda no cargados, usando User_Id – User_Name





6. Conclusiones

En base a las hipótesis planteadas en el comienzo, y a las que se fueron generando en el transcurso del desarrollo del presente informe, se describen conclusiones en varios aspectos relevantes de la experimentación.

Para complementar el análisis de complejidad de tiempo, se adjunta la siguiente figura [10]

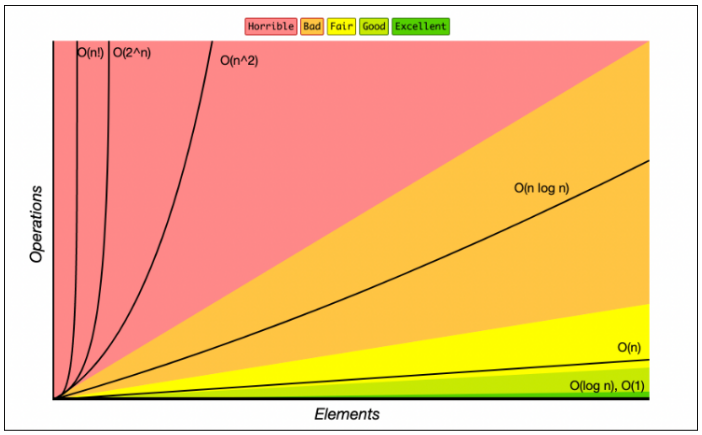
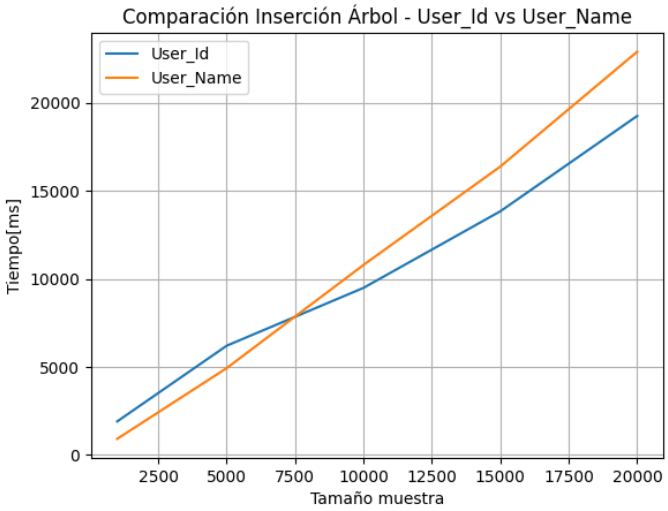


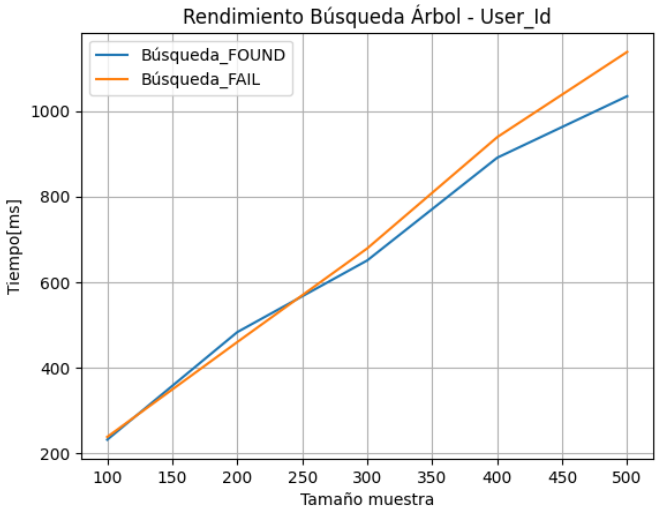
Figura 4-Gráfico que ilustra complejidad de tiempo en notación Big O

6.1 Uso de clave User_Id vs User_Name para inserción en árbol



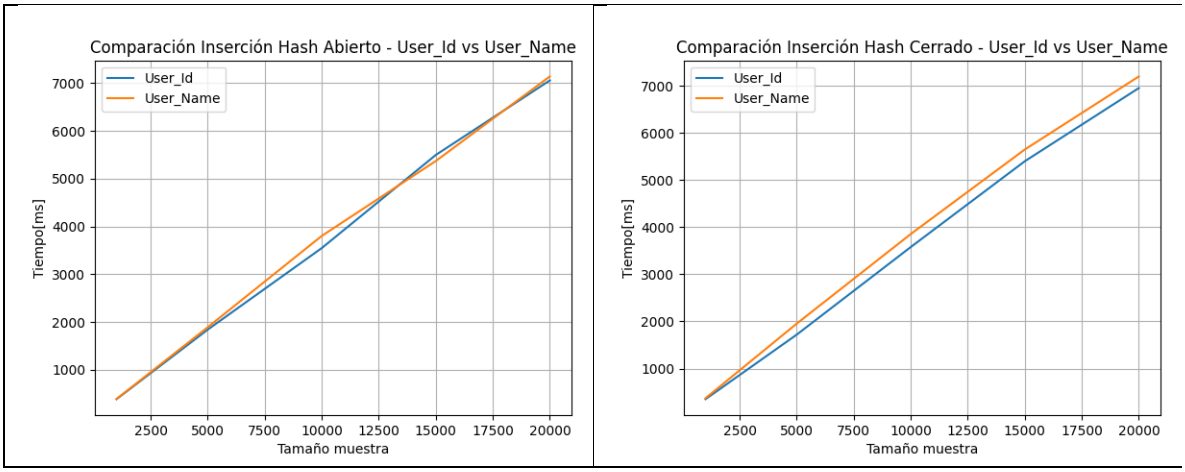
- Se observa que para muestras pequeñas (hasta los 7500 registros) el uso de clave User_Name resulta ligeramente superior a la clave User_Id. Pero para el resto de las muestras, User_Id es bastante superior. Se puede atribuir este comportamiento a la operación adicional de convertir String a Int.

6.2 Complejidad de tiempo esperada O(Log n) en búsquedas para árbol



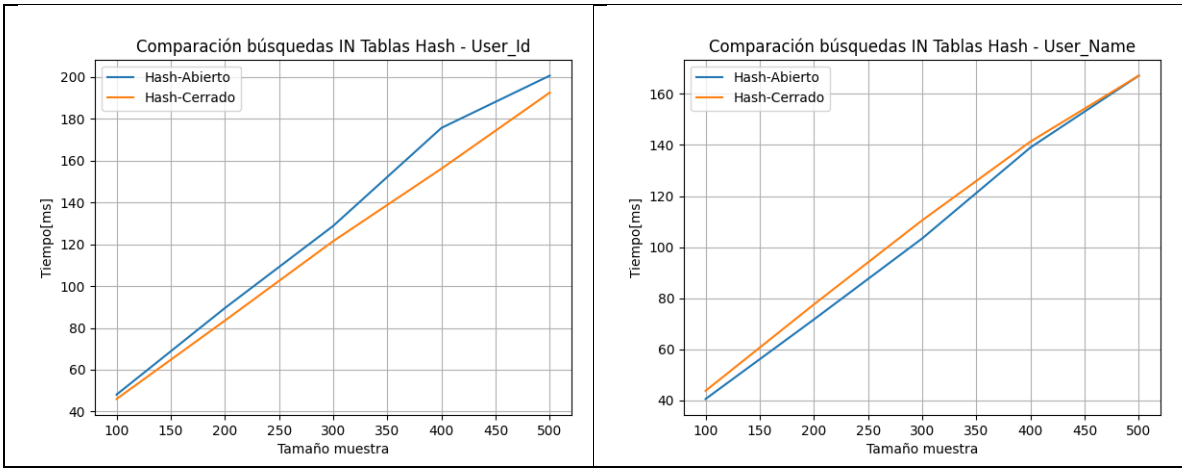
- Si bien la librería utilizada (AVLTree) prometía un rendimiento aproximado de O(Log n), se observa que, de acuerdo a las pruebas, se obtiene un rendimiento más cercano a O(n Log n).

6.3 Uso de clave User_Id vs User_Name para inserción en Tabla Hash Abierta



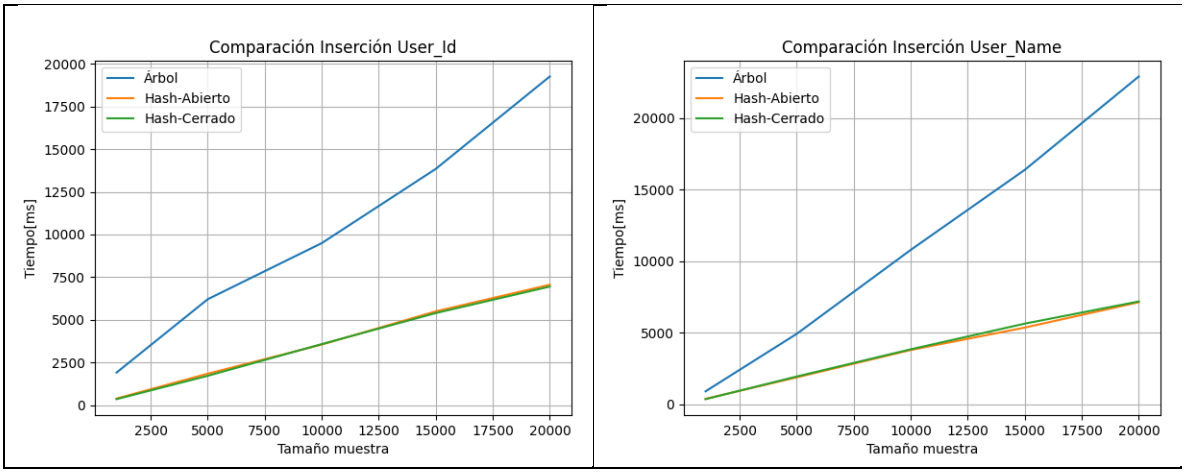
- No se observa mayor diferencia en el uso de diferentes claves. Ambas claves son almacenadas en la lista sin mayor costo adicional.
- Debido a la implementación, el uso de claves de tipos de datos diferentes solamente afecta al proceso de inserción del árbol binario de búsqueda.

6.4 Complejidad de tiempo en búsquedas en Tablas Hash, usando User_Id



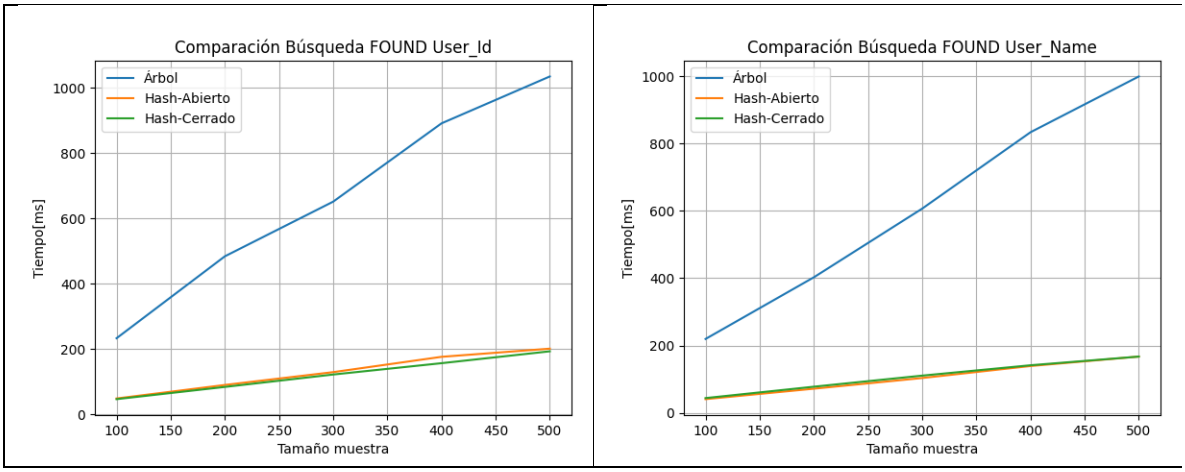
- Se observan rendimientos similares, sin embargo el uso de claves numéricas (User_ID) favorecen al hashing cerrado, mientras que las claves alfanumércias (User_Name) ofrecen mejor rendimiento con hashing abierto
- Por supuesto este resultado se basa en la implementación realizada en el código para este caso en particular

6.5 Comparación de inserción



- Se observa que el rendimiento en inserción es mucho mejor para el caso de Tablas Hash con hashing abierto y cerrado. La complejidad de tiempo de inserción para el árbol binario se acerca más a O(n Log n), mientras que para las Tablas Hash se acerca más a O(n)

6.6 Comparación de búsquedas



- Se aprecia una diferencia de rendimiento entre árbol y tabla hash (con hashing abierto y cerrado). Entre la búsqueda de un dataset pequeño y uno grande hay un costo casi 4 veces superior.

6.7 Conclusiones generales

- El impacto del uso de una clave de tipo long (User_Id) vs string (User_Name) depende exclusivamente de la implementación en el código. En las pruebas realizadas solamente se observó un efecto en la inserción del árbol binario de búsqueda.
- El proceso de inserción y de búsqueda en Tablas Hash presenta un mejor rendimiento que en un árbol binario de búsqueda. La diferencia se hace mayor utilizando muestras más grandes de datos.
- Para la implementación utilizada, no se ven grandes diferencias de rendimiento entre Tabla Hash con hashing abierto y hashing cerrado, salvo, por supuesto, a nivel de complejidad de espacio, donde la primera necesita memoria adicional para las listas enlazadas, mientras que la segunda opción requiere una tabla de al menos el tamaño de la muestra a probar.
- Existen más variaciones que se podrían probar a futuro como diferentes funciones de hashing, otro tipo de árbol u otras funciones de sondeo.

7. Referencias

[1] Wikipedia, Árbol binario. [En línea]. Disponible:
https://es.wikipedia.org/wiki/Árbol_binario

[2] Wikipedia, AVL Tree. [En línea]. Disponible:
https://en.wikipedia.org/wiki/AVL_tree

[3] Wikipedia, Árbol rojo-negro. [En línea]. Disponible:
https://es.wikipedia.org/wiki/Árbol_rojo-negro

[4] Wikipedia, Tabla Hash. [En línea]. Disponible:
https://es.wikipedia.org/wiki/Tabla_hash

[5] Github, Código fuente. [En línea]. Disponible:
https://github.com/egruttner/FEDA-Informe3/code

[6] Github, AVL-Tree. [En línea]. Disponible:
https://github.com/KadirEmreOto/AVL-Tree

[7] Matplotlib. [En línea]. Disponible: https://matplotlib.org

[8] Github, Datasets. [En línea]. Disponible:
https://github.com/egruttner/FEDA-Informe3/code/datasets

[9] Github, Archivos CSV. [En línea]. Disponible:
https://github.com/egruttner/FEDA-Informe3/code/csv

[10] Freecodecamp, TimeComplexityChart. [En línea]. Disponible:
https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/