|  |  |  |
| --- | --- | --- |
|  |  |  |

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 4](#_Toc138023844)

[2. Descripción de las estructuras de datos 5](#_Toc138023845)

[2.1 Árbol binario de búsqueda 5](#_Toc138023846)

[2.2 Tabla Hash con hashing abierto 6](#_Toc138023847)

[2.3 Tabla Hash con hashing cerrado 7](#_Toc138023848)

[3. Código y documentación 9](#_Toc138023849)

[3.1 Implementación de árbol binario 9](#_Toc138023850)

[3.2 Implementación de tabla hash con hashing abierto 9](#_Toc138023851)

[3.3 Implementación de tabla hash con hashing cerrado 9](#_Toc138023852)

[3.4 Medición de tiempos de ejecución 10](#_Toc138023853)

[3.5 Construcción del código 10](#_Toc138023854)

[3.6 Generación de gráficos 10](#_Toc138023855)

[4. Datasets y diseño experimental 11](#_Toc138023856)

[4.1 Archivo base 11](#_Toc138023857)

[4.2 Archivos de input - Inserción 11](#_Toc138023858)

[4.3 Archivos para realizar búsquedas 11](#_Toc138023859)

[4.4 Archivos para graficar 12](#_Toc138023860)

[4.5 Diseño experimental 12](#_Toc138023861)

[5. Resultados experimentales 13](#_Toc138023862)

[5.1 Mediciones de rendimiento usando Árbol de búsqueda 13](#_Toc138023863)

[5.1.1 Árbol - Inserción usando User\_Id 13](#_Toc138023864)

[5.1.2 Árbol - Inserción usando User\_Name 13](#_Toc138023865)

[5.1.3 Árbol - Búsqueda de datos cargados, usando User\_Id 14](#_Toc138023866)

[5.1.4 Árbol - Búsqueda de datos cargados, usando User\_Name 14](#_Toc138023867)

[5.1.5 Árbol - Búsqueda de datos no cargados, usando User\_Id 14](#_Toc138023868)

[5.1.6 Árbol - Búsqueda de datos no cargados, usando User\_Name 15](#_Toc138023869)

[5.2 Mediciones de rendimiento usando Tabla Hash con hashing abierto 15](#_Toc138023870)

[5.2.1 Hashing abierto - Usando User\_Id 15](#_Toc138023871)

[5.2.2 Hashing abierto - Usando User\_Name 16](#_Toc138023872)

[5.2.3 Hashing abierto - Búsqueda de datos cargados, usando User\_Id 16](#_Toc138023873)

[5.2.4 Hashing abierto - Búsqueda de datos cargados, usando User\_Name 16](#_Toc138023874)

[5.2.5 Hashing abierto - Búsqueda de datos no cargados, usando User\_Id 17](#_Toc138023875)

[5.2.6 Hashing abierto - Búsqueda de datos no cargados, usando User\_Name 17](#_Toc138023876)

[5.3 Medición de rendimiento de inserción en Tabla Hash con Hashing cerrado 17](#_Toc138023877)

[5.3.1 Hashing cerrado - Usando User\_Id 18](#_Toc138023878)

[5.3.2 Hashing cerrado - Usando User\_Name 18](#_Toc138023879)

[5.3.3 Hashing cerrado - Búsqueda de datos cargados, usando User\_Id 18](#_Toc138023880)

[5.3.4 Hashing cerrado - Búsqueda de datos cargados, usando User\_Name 19](#_Toc138023881)

[5.3.5 Hashing cerrado - Búsqueda de datos no cargados, usando User\_Id 19](#_Toc138023882)

[5.3.6 Hashing cerrado - Búsqueda de datos no cargados, usando User\_Name 19](#_Toc138023883)

[6. Conclusiones 20](#_Toc138023884)

[6.1 Árbol de búsqueda – AVL Tree 20](#_Toc138023885)

[6.2 Tabla Hash con hashing cerrado 20](#_Toc138023886)

[6.3 Tabla Hash con hashing abierto 20](#_Toc138023887)

[6.4 Uso de claves User\_Id vs User\_Name en Árbol 21](#_Toc138023888)

[6.5 Uso de claves User\_Id vs User\_Name en Tabla Hash con hashing abierto 21](#_Toc138023889)

[6.6 Uso de claves User\_Id vs User\_Name en Tabla Hash con hashing cerrado 21](#_Toc138023890)

[6.7 Comparación de inserción de datos usando clave User\_Id en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado 21](#_Toc138023891)

[6.8 Comparación de inserción de datos usando clave User\_Name en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado 21](#_Toc138023892)

[6.9 Comparación de búsqueda de datos usando clave User\_Id en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado 21](#_Toc138023893)

[6.10 Comparación de búsqueda de datos usando clave User\_Name en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado 21](#_Toc138023894)

[7. Referencias 21](#_Toc138023895)

# 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 describirán las estructuras a utilizar: Árbol binario de

búsqueda, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado.

A continuación, serán descritos 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 utilizó la librería AVLTree que promete una complejidad de tiempo esperada del orden del 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 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 dsfsdfsdf.

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 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.

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.

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.

https://es.wikipedia.org/wiki/Árbol\_binario\_de\_búsqueda

https://es.wikipedia.org/wiki/Árbol\_AVL

# 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.

Comparada con otras estructuras de arrays 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”.

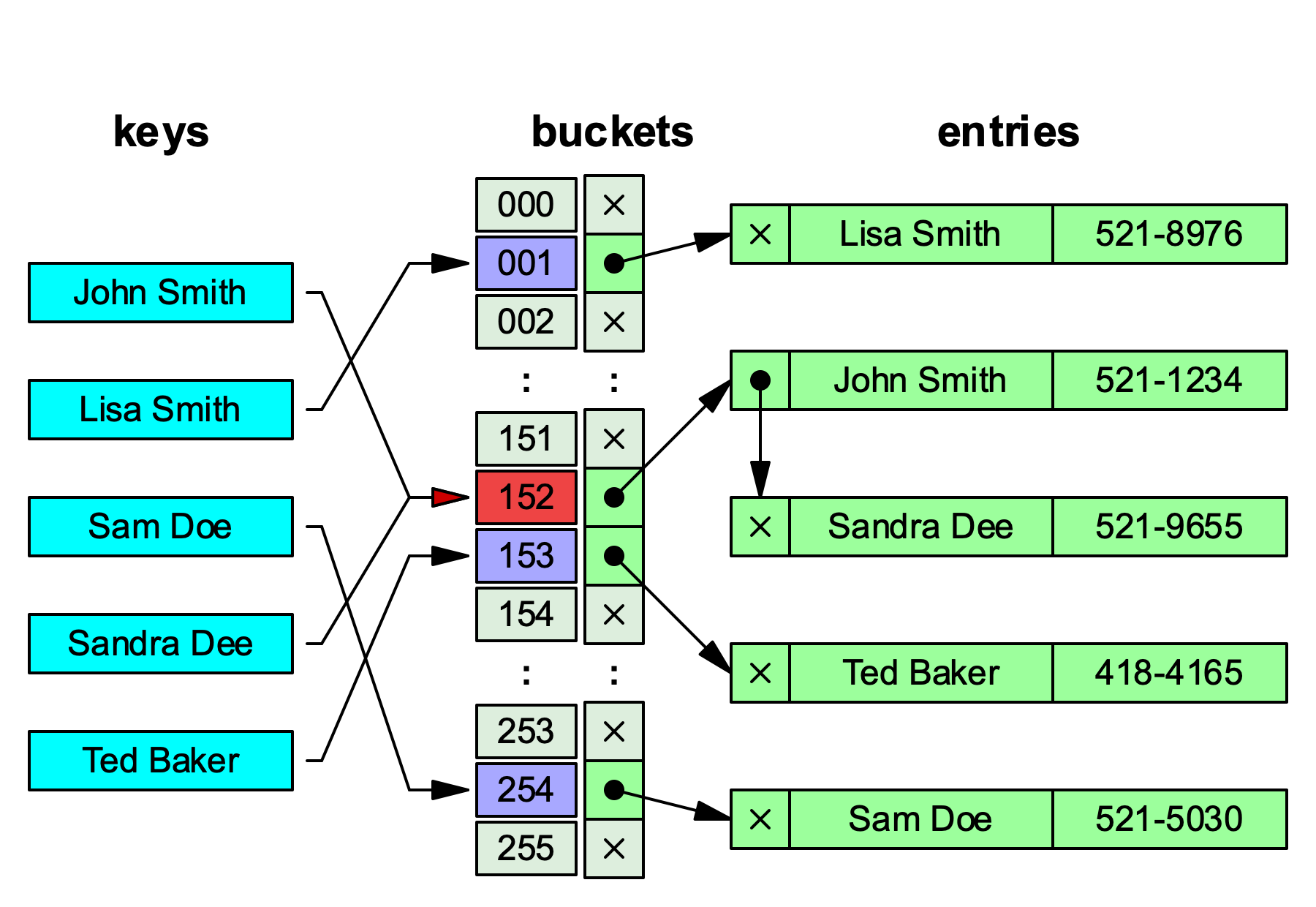
Si dos llaves generan un hash apuntando al mismo índice, los registros correspondientes no podrán ser almacenados en la misma posición. En estos casos, cuando una casilla ya está ocupada, se debe encontrar otra ubicación donde almacenar el nuevo registro, y hacerlo de tal manera que pueda ser encontrado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

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

https://es.wikipedia.org/wiki/Tabla\_hash

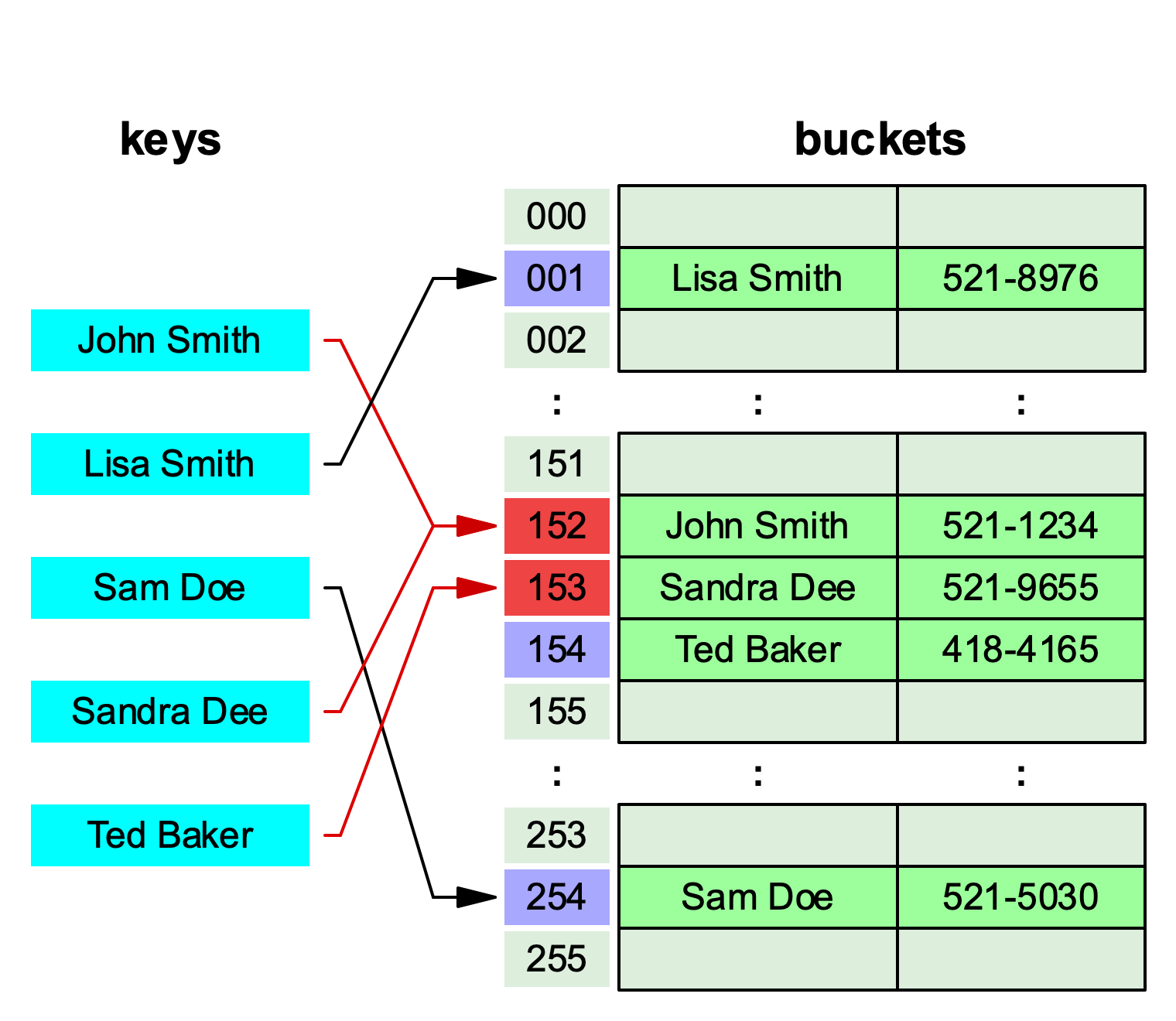
## 2.2.1 Tabla Hash con hashing abierto

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.



## 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.

****

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: <https://github.com/egruttner/FEDA-Informe3>

## 3.1 Implementación de árbol binario

Se utilizó la librería “AVL-Tree”, desarrollada por Kadir Emre Oto en 2018, <https://github.com/KadirEmreOto/AVL-Tree>

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.

**Wikipedia sondeo**

https://es.wikipedia.org/wiki/Tabla\_hash

## 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** 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:

|  |  |  |
| --- | --- | --- |
| 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 [[11](#once)]

## 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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 1000 | 1376.8 | | 5000 | 5042 | | 10000 | 9140.7 | | 15000 | 14282.2 | | 20000 | 20312.2 | |  |

## 5.1.2 Árbol - Inserción usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 1000 | 916.7 | | 5000 | 4960.9 | | 10000 | 10612.1 | | 15000 | 17048.8 | | 20000 | 23350.8 | |  |

## 5.1.3 Árbol - Búsqueda de datos cargados, usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 253.2 | | 200 | 443.2 | | 300 | 673.6 | | 400 | 840.2 | | 500 | 1077 | |  |

## 5.1.4 Árbol - Búsqueda de datos cargados, usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 223.7 | | 200 | 431.5 | | 300 | 616 | | 400 | 818.9 | | 500 | 1009.7 | |  |

## 5.1.5 Árbol - Búsqueda de datos no cargados, usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 258.4 | | 200 | 446.1 | | 300 | 676.6 | | 400 | 993.1 | | 500 | 1167.3 | |  |

## 5.1.6 Árbol - Búsqueda de datos no cargados, usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 246.1 | | 200 | 448.9 | | 300 | 709.7 | | 400 | 905.6 | | 500 | 1119.5 | |  |

## 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 - Usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 1000 | 372 | | 5000 | 1830.3 | | 10000 | 3606.3 | | 15000 | 5633.7 | | 20000 | 7358.5 | |  |

## 5.2.2 Hashing abierto - Usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 1000 | 388.5 | | 5000 | 1912.6 | | 10000 | 3789.4 | | 15000 | 5632.6 | | 20000 | 7256.5 | |  |

## 5.2.3 Hashing abierto - Búsqueda de datos cargados, usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 246.7 | | 200 | 448.2 | | 300 | 666 | | 400 | 896.7 | | 500 | 1104.3 | |  |

## 5.2.4 Hashing abierto - Búsqueda de datos cargados, usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 223.4 | | 200 | 425.9 | | 300 | 642.3 | | 400 | 818.7 | | 500 | 1021.7 | |  |

## 5.2.5 Hashing abierto - Búsqueda de datos no cargados, usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 259.7 | | 200 | 452.9 | | 300 | 663.8 | | 400 | 1047.2 | | 500 | 1103.2 | |  |

## 5.2.6 Hashing abierto - Búsqueda de datos no cargados, usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 283.5 | | 200 | 475.7 | | 300 | 713 | | 400 | 922.5 | | 500 | 1139.3 | |  |

## 5.2 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 - Usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 1000 | 368.4 | | 5000 | 1753.8 | | 10000 | 3459.5 | | 15000 | 5242.6 | | 20000 | 7403 | |  |

## 5.3.2 Hashing cerrado - Usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 1000 | 437.2 | | 5000 | 2480.2 | | 10000 | 4254.8 | | 15000 | 6109.4 | | 20000 | 7517.5 | |  |

## 5.3.3 Hashing cerrado - Búsqueda de datos cargados, usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 240.1 | | 200 | 444.6 | | 300 | 647.6 | | 400 | 871.1 | | 500 | 1124.9 | |  |

## 5.3.4 Hashing cerrado - Búsqueda de datos cargados, usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 222.7 | | 200 | 425.1 | | 300 | 607.7 | | 400 | 815.9 | | 500 | 1016.7 | |  |

## 5.3.5 Hashing cerrado - Búsqueda de datos no cargados, usando User\_Id

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 241.5 | | 200 | 444.2 | | 300 | 625.3 | | 400 | 816.3 | | 500 | 1142.6 | |  |

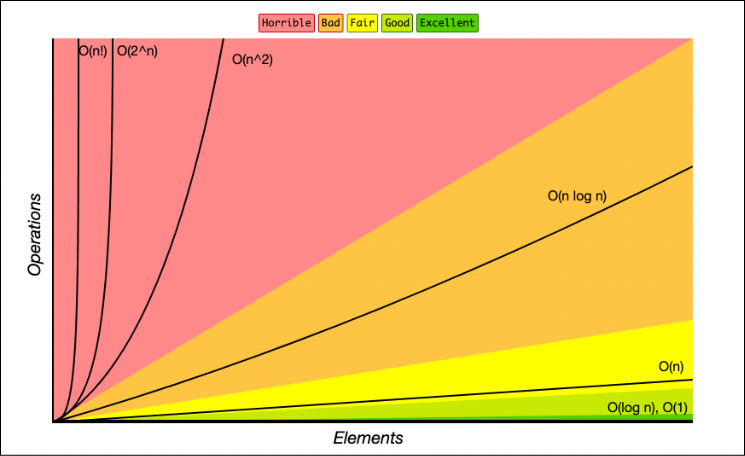
## 5.3.6 Hashing cerrado - Búsqueda de datos no cargados, usando User\_Name

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Tamaño muestra | Tiempo [ms] | | 100 | 261.8 | | 200 | 467.8 | | 300 | 631.8 | | 400 | 814.3 | | 500 | 1107.9 | |  |

# 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.

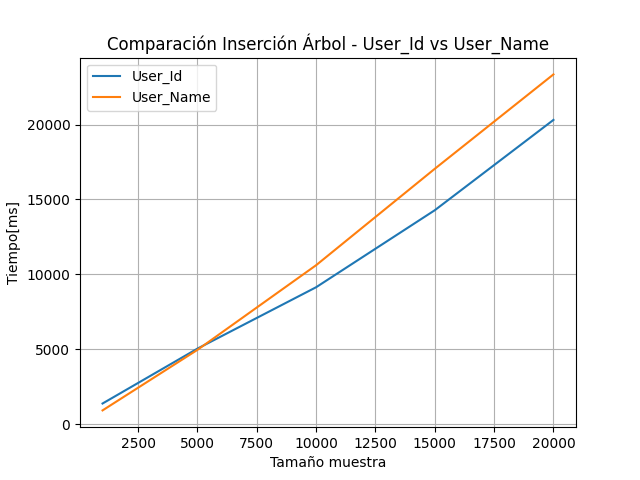
Para complementar el análisis de complejidad de tiempo, se adjunta la siguiente figura.

****

https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/

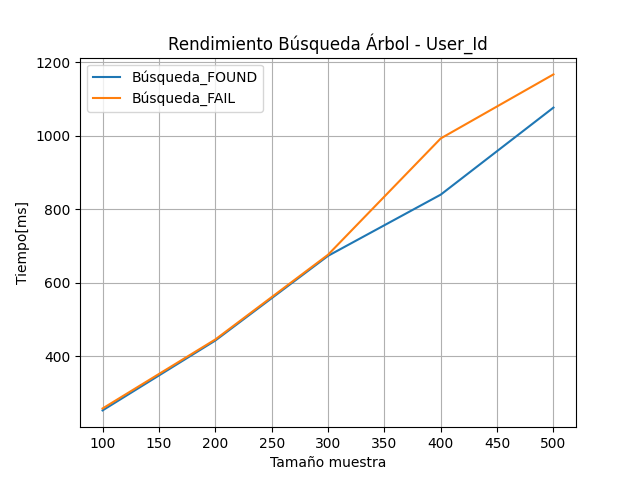
## 6.1 Árbol de búsqueda

**6.1.1 Uso de clave User\_Id vs User\_Name**

****

* Se observa que para muestras pequeñas (hasta los 5000 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 de adicional de convertir String a Int.

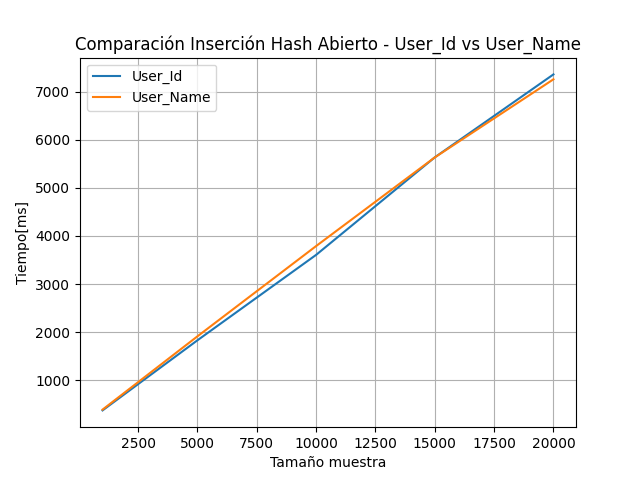
**6.1.2 Complejidad de tiempo esperada O(Log n) en búsquedas**

****

* 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).

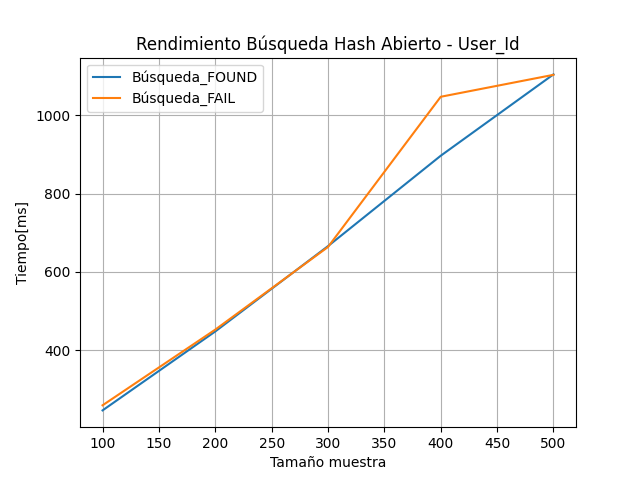
**6.2 Tabla Hash con hashing abierto**

**6.2.1 Uso de clave User\_Id vs User\_Name**

****

* No se observa mayor diferencia en el uso de diferentes claves. Ambas claves son almacenadas en la lista sin mayor costo adicional.
* Se observa una complejidad de tiempo cercana a O(n Log n)

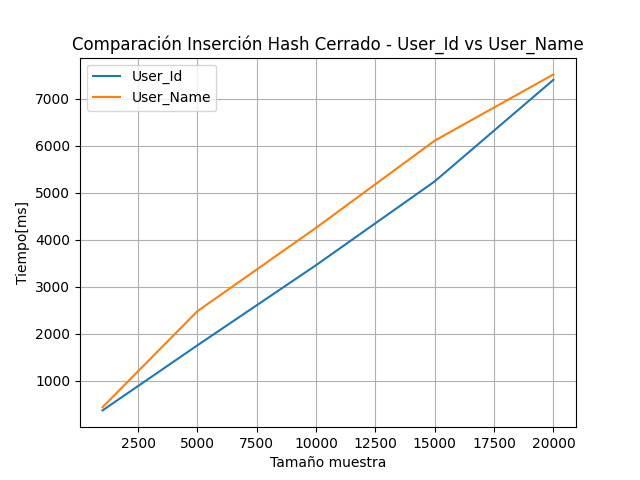
**6.1.3 Complejidad de tiempo en búsquedas, usando User\_Id**



* Salvo la excepción de la muestra 400, no se observa mayor diferencia de rendimiento
* Se observa una complejidad de tiempo cercana a O(n Log n)

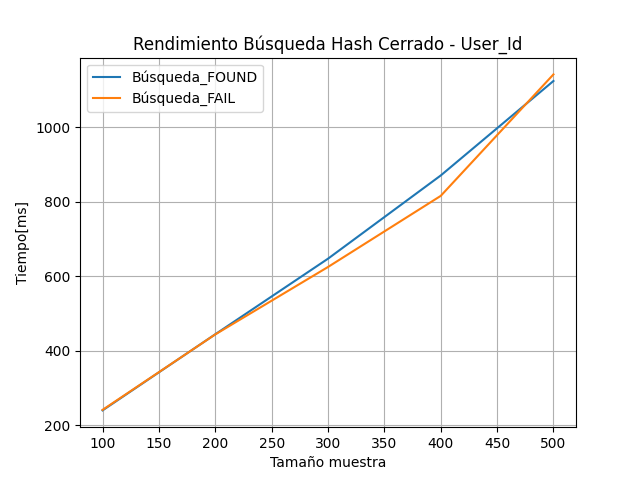
## 6.3 Tabla Hash con hashing cerrado

**6.2.1 Uso de clave User\_Id vs User\_Name**

****

* En este caso sí se observa un impacto en la conversión de String a Int para el almacenamiento de la llave, siendo ganador en gran parte de la muestra el uso de User\_Id. Sin embargo, en muestras de tamaño pequeños o muy grandes, la diferencia casi no existe.
* Se observa una complejidad de tiempo, promediando ambas curvas, cercana a O(n Log n)

**6.1.3 Complejidad de tiempo en búsquedas, usando User\_Id**

****

* Sin mayores diferencia de rendimiento, se observa una complejidad de tiempo cercana a O(n Log n)

## 6.4 Comparaciones entre estructuras

## 6.4.1 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)

## 6.4.2 Comparación de búsquedas

|  |  |
| --- | --- |
|  |  |

* Se observa que el rendimiento para la búsqueda de datos insertados es muy similar, parecido a O(n Log n), pero se debe tener en cuenta que, en el paso anterior de la inserción, usar Tablas Hash tiene un rendimiento muy superior. Debe ser considerado el proceso completo.

# 7. Referencias

[1] Wikipedia, Selection sort. [En línea]. Disponible: https://en.wikipedia.org/wiki/Selection\_sort

[2] Wikipedia, Merge sort. [En línea]. Disponible: https://en.wikipedia.org/wiki/Merge\_sort

[3] Wikipedia, Quick sort. [En línea]. Disponible: https://es.wikipedia.org/wiki/Quicksort

[4] Wikipedia, Sort STL. [En línea]. Disponible: https://en.wikipedia.org/wiki/Sort\_(C%2B%2B)

[5] Medium, Algorithm Analysis & Time Complexity Simplified. [En línea]. Disponible: https://randerson112358.medium.com/algorithm-analysis-time-complexity-simplified-cd39a81fec71

[6] Wikipedia, Multiplicación de matrices. [En línea]. Disponible: https://es.wikipedia.org/wiki/Multiplicación\_de\_matrices

[7] De User:Bilou - Trabajo propio, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1538693

[8] Github, Explicación del algoritmo de Strassen. [En línea]. Disponible: https://alu0100881677.github.io/DAA\_L2\_1\_Strassen/Strassen.html

[9] Github, Herramienta para la generación de datasets para algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/datasets

[10] Github, Outputs de ejecución de algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/output

[11] Github, Archivos para graficar. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/csv

[12] Github, Herramienta para la generación de datasets para multiplicación de matrices. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/datasets

[13] Github, Outputs de ejecución de algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/output

[14] Github, Archivos para graficar. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/csv

[15] Github, Gráficos para algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/gr%C3%A1ficos

[16] Github, Gráficos para multiplicación de matrices. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/gr%C3%A1ficos

[17] Github, Código fuente herramienta análisis algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code

[18] Github, Código fuente herramienta análisis algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code