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

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](#_Toc133954985)

[2. Descripción de los algoritmos a ser comparados 5](#_Toc133954986)

[2.1 Ordenamiento 5](#_Toc133954987)

[2.1.1 Selection Sort 5](#_Toc133954988)

[2.1.2 Merge Sort 6](#_Toc133954989)

[2.1.3 Quick Sort 7](#_Toc133954990)

[2.1.4 Sort Interno C++ 8](#_Toc133954991)

[2.1.5 Tabla resumen algoritmos de ordenamiento a analizar en este informe 9](#_Toc133954992)

[2.2 Multiplicación de matrices 10](#_Toc133954993)

[2.2.1 Algoritmo iterativo cúbico tradicional 11](#_Toc133954994)

[2.2.2 Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos 12](#_Toc133954995)

[2.2.3 Algoritmo Strassen 13](#_Toc133954996)

[3. Descripción de los datasets 14](#_Toc133954997)

[3.1 Datasets para algoritmos de ordenamiento 14](#_Toc133954998)

[3.1.1 Inputs 14](#_Toc133954999)

[3.1.2 Outputs 16](#_Toc133955000)

[3.1.3 Archivos para graficar 16](#_Toc133955001)

[3.2 Datasets para multiplicación de matrices 17](#_Toc133955002)

[3.1.1 Inputs 17](#_Toc133955003)

[3.1.2 Outputs 19](#_Toc133955004)

[3.1.3 Archivos para graficar 19](#_Toc133955005)

[4. Resultados experimentales 20](#_Toc133955006)

[4.1 Medición rendimiento algoritmos de ordenamiento 21](#_Toc133955007)

[4.1.1 Caso 1: Datos desordenados repetidos 21](#_Toc133955008)

[4.1.2 Caso 2: Datos desordenados con valores únicos 22](#_Toc133955009)

[4.1.3 Caso 3: Datos ordenados repetidos 23](#_Toc133955010)

[4.1.4 Caso 4: Datos ordenados reversa (descendente) 24](#_Toc133955011)

[4.1.5 Caso 5: Datos parcialmente ordenados 25](#_Toc133955012)

[4.2 Medición rendimiento para multiplicación de matrices 26](#_Toc133955013)

[4.2.1 Caso 1: Matrices cuadradas 26](#_Toc133955014)

[4.2.2 Caso 2: Matrices cuadradas con tamaño potencia de 2 27](#_Toc133955015)

[4.2.3 Caso 3: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas variables, columnas fijas) 28](#_Toc133955016)

[4.2.4 Caso 4: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas y columnas variables) 29](#_Toc133955017)

[4.2.5 Caso 5: Matrices rectangulares con matriz A (filas y columnas variables) y matriz B (filas variables y columnas fijas) 30](#_Toc133955018)

[5. Conclusiones 31](#_Toc133955019)

[5.1 Algoritmos de ordenamiento 31](#_Toc133955020)

[5.2 Multiplicación de matrices 32](#_Toc133955021)

[6. Referencias 33](#_Toc133955022)

# 1. Introducción

**Se introduce de manera correctitud y coherente el tema a tratar. Se menciona el objetivo general de la tarea y una breve descripción de las conclusiones obtenidas**

**La temática y las conclusiones se introducen de manera adecuada**

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

A continuación será descrito el marco experimental, tanto los detalles de construcción como los diferentes datasets utilizados.

Además del análisis global entre estructuras, serán utilizadas diferentes claves para ver el comportamiento

Serán presentados gráficos y tablas de tiempo y ejecución

Se mostrará que el árbol tiene O(Log n)

Se mostrará que hashign cerrado tiene

Se mostrará que hashing abierto tiene

Finalmente, en el apartado de conclusiones se revisarán aquiellos puntos notables del análisis con indicaciones

**\*Sobre el dataset que es fijo y que es bueno para el árbol**

**\*Sobre hashing cuál es finalmente la diferencia entre abierto y cerrado, sus ventajas y desventajas**

# 2. Descripción de las estructuras de datos

**Se describen, como mínimo, las estructuras de datos comparadas: Árbol binario, tabla hash con hashing abierto, tabla hash con hashing cerrado. Cada estructura debe ser descrita en un párrafo, incluyendo referencias bibliográficas de ser necesario.**

**Se presenta de manera correcta la descripción de las 3 estructuras de datos solicitadas**

## 2.1 Árbol binario de búsqueda

BST, o Binary Search Tree en inglés, se traduce como árbol de búsqueda binario en español. 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 Binary Search Tree, cada nodo contiene un valor único y se divide en un subárbol izquierdo y uno derecho. La clave característica de un BST 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 BST se basa en el principio de "divide and conquer". Al realizar una búsqueda en un BST, 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.

En resumen, un Binary Search Tree (árbol de búsqueda binario) es una estructura de datos ordenada que permite una búsqueda eficiente en tiempo logarítmico. Es útil cuando se necesita realizar búsquedas frecuentes en una colección de elementos y se mantiene el equilibrio del árbol para asegurar un rendimiento óptimo.

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

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

## 2.2 Tabla Hash con hashing abierto

Una tabla hash con hashing abierto es una estructura de datos que utiliza una función hash para almacenar y recuperar elementos de manera eficiente. En lugar de resolver las colisiones mediante el uso de listas enlazadas (hashing cerrado), en el hashing abierto, se emplea una estrategia que involucra explorar otras ubicaciones en la tabla hash cuando se produce una colisión.

La tabla hash en el hashing abierto consta de una serie de "ranuras" o "buckets", cada una de las cuales puede contener un elemento o estar vacía. Cada elemento se inserta en una ranura de acuerdo con el valor que produce la función hash aplicada a su clave. Si una ranura está ocupada, se utiliza una estrategia de exploración para buscar la siguiente ranura disponible.

Existen diferentes técnicas de exploración utilizadas en el hashing abierto, algunas de las cuales son:

1. Exploración lineal: Si se produce una colisión en una ranura, se examinan las ranuras consecutivas hasta encontrar una ranura vacía. La exploración se realiza de manera lineal, moviéndose secuencialmente a través de las ranuras.

2. Exploración cuadrática: En lugar de moverse linealmente, se utiliza una secuencia cuadrática para determinar las ranuras a explorar después de una colisión. Esto ayuda a evitar agrupamientos y distribuye los elementos colisionantes de manera más uniforme.

3. Exploración por doble hashing: En esta técnica, se utiliza una segunda función hash para determinar los desplazamientos entre las ranuras a explorar. La segunda función hash se aplica cuando ocurre una colisión y se calcula un nuevo valor de salto para buscar una nueva ranura.

Cada técnica de exploración tiene sus ventajas y desventajas, y su elección depende del contexto y los requisitos específicos del problema.

Es importante tener en cuenta que en el hashing abierto, la tabla hash debe tener suficiente capacidad para evitar un alto grado de colisiones, lo que podría afectar negativamente el rendimiento. Además, si la tabla se llena por completo, puede ser necesario redimensionarla para evitar la degradación del rendimiento.

En resumen, una tabla hash con hashing abierto es una estructura de datos que utiliza una función hash y estrategias de exploración para manejar colisiones. Permite almacenar y recuperar elementos de manera eficiente, explorando ranuras alternativas cuando se produce una colisión. La elección de la estrategia de exploración y el manejo adecuado de la capacidad de la tabla son aspectos importantes para garantizar un rendimiento óptimo.

Diagrama hashing abierto

## 2.3 Tabla Hash con hashing cerrado

Por supuesto, aquí tienes una explicación más teórica de una tabla hash con hashing cerrado y uso de listas enlazadas.

Una tabla hash con hashing cerrado y uso de listas enlazadas combina dos técnicas para manejar colisiones en una tabla hash. En lugar de simplemente reemplazar el valor existente cuando se produce una colisión, se utiliza una lista enlazada para almacenar varios elementos con la misma posición hash.

El proceso de inserción en una tabla hash con hashing cerrado y listas enlazadas es el siguiente:

1. Se calcula el valor hash de la clave utilizando una función de hash. Esta función mapea la clave a un número entero, que se utilizará como índice en la tabla hash.

2. Se verifica si la posición calculada está ocupada. Si está vacía, se inserta el elemento en esa posición.

3. Si la posición está ocupada, se agrega el nuevo elemento al final de la lista enlazada asociada a esa posición.

En el caso de búsqueda en una tabla hash con hashing cerrado y listas enlazadas:

1. Se calcula el valor hash de la clave utilizando la misma función de hash.

2. Se accede a la posición correspondiente en la tabla hash.

3. Se recorre la lista enlazada asociada a esa posición y se busca la clave deseada. Si se encuentra, se devuelve el valor correspondiente. Si no se encuentra, se considera que la clave no está presente en la tabla.

En cuanto a la eliminación de elementos:

1. Se calcula el valor hash de la clave.

2. Se accede a la posición correspondiente en la tabla hash.

3. Se recorre la lista enlazada asociada a esa posición y se busca la clave a eliminar. Si se encuentra, se elimina el elemento de la lista. Si no se encuentra, no se realiza ninguna acción.

La ventaja de utilizar listas enlazadas en caso de colisión es que se puede almacenar un número ilimitado de elementos con la misma posición hash. Sin embargo, el acceso a los elementos puede ser un poco más lento debido a la necesidad de recorrer la lista enlazada para buscar un elemento en particular.

En resumen, una tabla hash con hashing cerrado y uso de listas enlazadas es una estructura de datos que combina el hashing cerrado para asignar claves a posiciones en una tabla y el uso de listas enlazadas para almacenar múltiples elementos con la misma posición hash. Esto permite manejar colisiones y almacenar varios elementos en una misma posición de la tabla.

\*Diagrama hashing Cerrado

# 3. Código y documentación

El código completo se encuentra en el repositorio XXX

3.1 Implementación de árbol binario

Para la implementación de árbol se utilizó la librería

https://github.com/KadirEmreOto/AVL-Tree

AVL-Tree

AVL Tree is a **self-balancing** binary search tree which guarantees O(logN) time complexity for insertion, deletion, and look-up operations.

This version is implemented in C++ language, and allows multiple elements with same values. It means the tree can be used as **multiset**.

#### How to use it:

The tree can store any type of data which has **greater than** (>) operator. Build-in types such as int, long long, and std::string are well-tested, however user-defined structures, even has > operator, may cause some exceptions.

Algunos métodos y su complejidad de tiempo

Inserción, O(Log n)

Búsqueda de elemento en la posición “índice” del árbol, O(Log n)

Entregar el tamaño del árbol, valor constante

Limpiar árbol, O(n)

Mostrar árbol, O(n)

Se basa en la librería  Red Black Trees (**RB Trees**) que viene incorporada en STL de C++

Hecha por Kadir Emre Oto en 2018

Esta implementación permite instanciar, indistintamente, un árbol de string o de long

The tree can store any type of data which has **greater than** (>) operator. Build-in types such as int, long long, and std::string are well-tested, however user-defined structures, even has > operator, may cause some exceptions.

3.2 Implementación de Hashing Abierto

Para su construcción se utilizó una idea base provista por ChatGPT, la cual se modificó y adaptó a la necesidad del proyecto.

Se crearon clases que soporten User\_Id y User\_Name como claves separadas, con sus características propias (Long y String)

Para el caso de String se incorporó una función diferente para el hashing, asociado a la naturaleza propia de los caracteres

3.3 Implementación de Hashing Cerrado

Para su construcción se utilizó una idea base provista por ChatGPT, la cual se modificó y adaptó a la necesidad del proyecto.

Se crearon clases que soporten User\_Id y User\_Name como claves separadas, con sus características propias (Long y String)

Describir la función utilizada

**Hashing Cerrado user\_id**

# Método de la división

# k: clave a la cual aplicaremos la función hash

# n: tamaño de la tabla hash

def h1(k, n):

return k % n

**3.4 Medición de tiempos de ejecución**

**Se utilizó la librería “Chrono”, que permite medira**

**Cada tarea ejecutada fue medida en base a un promedio de ejecución de 10 intentos.**

**Cada ejecución registra 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 código**

**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 liubrería MatPlotLib y que permite generar gráficos customizados de acuerdo a las pruebas requeridas por el proyecto**

* **Mencionar la cantidad de buckets**
* **Mencionar OUTPUT si fuese necesario**

# 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:**

**Input 1.csv -> 1000 registros**

**Input 2.csv -> 5000 registros**

**Input 3.csv -> 10.000 registros**

**Input 4.csv -> 15.000 registros**

**Input 5.csv -> 20.000 registros**

input 1, 85KB

input 2, 418KB

input 3, 816KB

input 4, 1,2MB

input 5, 1,7MB

|  |
| --- |
| inputxx.txt |
| **university;user\_id;user\_name;number\_tweets;friends\_count;followers\_count;created\_at** |
| **Dato 1** |
| **…** |
| **Dato n** |

Cada archivo contiene una cabecera que fue ignorada en cada lectura del mismo.

**university;user\_id;user\_name;number\_tweets;friends\_count;followers\_count;created\_at**

**4.3 Archivos de búsqueda (datos “encontrables” y no “encontrables”)**

**Se crearon 5 datasets con datos extraídos del archivo base, con distinto tamaño y peso para realizar búsquedas**

**Input 1.csv -> 100 registros, 8KB**

**Input 2.csv -> 200 registros, 17KB**

**Input 3.csv -> 300 registros, 25KB**

**Input 4.csv -> 400 registros, 34KB**

**Input 5.csv -> 500 registros, 42KB**

|  |
| --- |
| inputxx.txt |
| **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 concantena una letra “X” para que no aparezca en el resultado**

### 4.1.3 Archivos para graficar

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

|  |
| --- |
| nombre\_algoritmo\_ordenamiento\_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)]

4.4 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.

El código fuente para algoritmos de ordenamiento está disponible en [[17](#diecisiete)] y para multiplicación de matrices en [[18](#dieciocho)].

# 5. Resultados experimentales

**Incluye una sección describiendo los experimentos realizados, resumiendo de manera adecuada los resultados obtenidos por medio de tablas y figuras.**

**Los resultados entregados están bien explicados y resumidos. Se entregan resultados de tiempo de ejecución y espacio de las estructuras. Además, se utilizan los dos tipos de claves en la comparación: User ID y User Name.**

Forma de realizar las mediciones (ordenamiento y multiplicación de matrices)

* Generación de datasets
* Utilización de script bash para la llamada de cada dataset

Ejemplo, para la ejecución de script de multiplicación de matrices:

|  |
| --- |
| num\_datasets=5  for (( i=1; i <= $num\_datasets; ++i ))  do  make num\_dataset=$i  python3 plot.py csv/csv$i/standard\_results.csv  python3 plot.py csv/csv$i/transpose\_results.csv  python3 plot.py csv/csv$i/strassen\_results.csv  python3 plot\_dos.py csv/csv$i/standard\_results.csv csv/csv$i/transpose\_results.csv  python3 plot\_todos.py csv/csv$i/standard\_results.csv csv/csv$i/transpose\_results.csv csv/csv$i/strassen\_results.csv  done |

Este script ejecuta el llamado a make y luego muestra en pantalla los gráficos generados a partir de los archivos CSV producidos.

Para calcular el tiempo de ejecución, se utilizó la librería Chrono de C++:

|  |
| --- |
| long long execution\_time\_ms(Func function, const vector<int> &A, string alg) {  auto start\_time = std::chrono::high\_resolution\_clock::now();  function(A, alg);  auto end\_time = std::chrono::high\_resolution\_clock::now();  return std::chrono::duration\_cast<std::chrono::microseconds>(end\_time - start\_time).count();  } |

Donde “alg” corresponde al algoritmo a medir (ej: Quick sort)

Para graficar se usó la librería matplotlib de Python. Los gráficos para algoritmos de ordenamiento se encuentran en [[15](#quince)] y los gráficos asociados a la multiplicación de matrices se encuentran en [[16](#dieciseis)].

## 4.1 Medición rendimiento algoritmos de ordenamiento

### 4.1.1 Caso 1: Datos desordenados repetidos

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

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. De ello se puede inferir que el peor fue “Mergesort”, y que “Quicksort” y “SortInterno” tuvieron un desempeño muy similar.

### 4.1.2 Caso 2: Datos desordenados con valores únicos

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

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. De ello se puede inferir que el peor fue “Mergesort”, y que el “SortInterno” fue ligeramente superior al “Quicksort”.

### 4.1.3 Caso 3: Datos ordenados repetidos

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

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. En este caso “Quicksort” tuvo el peor rendimiento, siendo el mejor “SortInterno” seguido por “Mergesort”.

### 4.1.4 Caso 4: Datos ordenados reversa (descendente)

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

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. Pese a la cercanía del desempeño en el gráfico, “SortInterno” sigue siendo el que tiene mejor desempeño por una amplia diferencia.

### 4.1.5 Caso 5: Datos parcialmente ordenados

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

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. Para tamaños menores (20.000) no es tan notoria como al llegar a tamaños grandes (100.000), donde nuevamente “Sort Interno” tiene mejor desempeño.

## 4.2 Medición rendimiento para multiplicación de matrices

### 4.2.1 Caso 1: Matrices cuadradas

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

Observaciones:

* El algoritmo “Strassen” tiene un muy bajo desempeño. De acuerdo al gráfico, al pasar del tamaño 600 la diferencia con los otros algoritmos se hace demasiado evidente.
* Se separó el gráfico de “Strassen” para comparar el desempeño de los otros dos. Se infiere que el algoritmo “Standard” se acerca a O(n2).

### 4.2.2 Caso 2: Matrices cuadradas con tamaño potencia de 2

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

Observaciones:

* El algoritmo “Strassen”, si bien tiene bajo desempeño, en esta ocasión no se aleja tanto del rendimiento de sus otros competidores.
* Se separó el gráfico de “Strassen” para comparar el desempeño de los otros dos. A tamaños pequeños no se ve gran diferencia, pero al llegar a los 1000 la diferencia es notable.

### 4.2.3 Caso 3: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas variables, columnas fijas)

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

Observaciones:

* No se dan las condiciones para realizar la ejecución del algoritmo “Strassen”, por lo que fue eliminado del análisis
* Se infiere que el algoritmo “Standard” se acerca a O(n2). En este caso, desde el primer momento “Transpose” es el claro ganador.

### 4.2.4 Caso 4: Matrices rectangulares con matriz A (filas fijas, columnas variables) y matriz B (filas y columnas variables)

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

Observaciones:

* No se dan las condiciones para realizar la ejecución del algoritmo “Strassen”, por lo que fue eliminado del análisis
* Se infiere que el algoritmo “Standard” se acerca a O(n2). A tamaños pequeños de matrices la diferencia en rendimiento no es tan grande.

### 4.2.5 Caso 5: Matrices rectangulares con matriz A (filas y columnas variables) y matriz B (filas variables y columnas fijas)

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

Observaciones:

* No se dan las condiciones para realizar la ejecución del algoritmo “Strassen”, por lo que fue eliminado del análisis
* Se infiere que el algoritmo “Standard” se acerca a O(n2). Existe una mínima diferencia al comenzar, que luego se hace evidente con la victoria de “Transpose” por mejor desempeño.

# 6. Conclusiones

Incluye una secciones con conclusiones y reflexiones generadas a partir de desarrollo del trabajo.

Las conclusiones son correctas y están respaldadas por los resultados mostrados en la sección de experimentos.

## 6.1 Árbol de búsqueda – AVL Tree­

El uso de un árbol balanceado y de la librería AVLTree, que garantizaba tiempo de complejidad O(Log n), hacía esperar de dsfdsfsdfsdf

Si bien esta expectativa se cumple a nivel individual, al compararse con el uso de tabla Hash (con hashing abierto y cerrado), pareciera dar la impresión de que solamente eso se cumple a niveles pequeños de datos

Gráfico

Pero si se hace un acercamiento al gráfico al comienzo de su ejecución, con menor cantidad de gráficos se puede observar la curva carcaterítiscasd

Gráfico con Zoom

Sobre User\_Id y User\_Name

Por la construcción de la librería y el rendimiento observado en los gráficos, se ve que el algoritmos funciona mejor cuando se trata de User\_Name, dato de tipo String

Este comportamiento se puede atribuir a las siguientes causas:

* Los largos de datos del dataset de User\_Name son menores que los largos de datos de User\_Id
* dsfdsfdsf

Comparación entre inserción y búsqueda – General

La inserción tiene una complejididad de tiempo de…

Y respecto a la búsqueda, se puede observar que tiene una complejidad de efefef

## 6.2 Tabla Hash con hashing cerrado

**Se espera una**

## 6.3 Tabla Hash con hashing abierto

## 6.4 Uso de claves User\_Id vs User\_Name en Árbol

## 6.5 Uso de claves User\_Id vs User\_Name en Tabla Hash con hashing abierto

## 6.6 Uso de claves User\_Id vs User\_Name en Tabla Hash con hashing cerrado

## 6.7 Comparación de inserción de datos usando clave User\_Id en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado

## 6.8 Comparación de inserción de datos usando clave User\_Name en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado

## 6.9 Comparación de búsqueda de datos usando clave User\_Id en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado

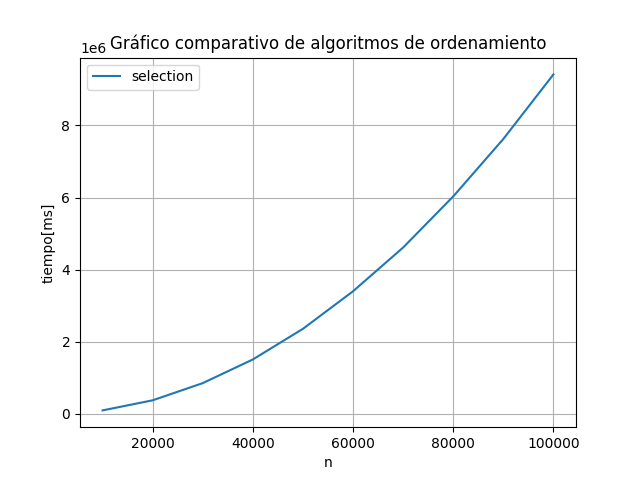
## 6.10 Comparación de búsqueda de datos usando clave User\_Name en Árbol, Tabla Hash con hashing abierto y Tabla Hash con hashing cerrado

Extra

Comparación más o menos buckets

Pruebas con otras funciones de Hashing

* El algoritmo “Selection Sort” es el de peor desempeño luego de todas las pruebas. En los gráficos se ve claramente su orden cuadrático O(n2):



* El algoritmo “Sort Interno” fue el de mejor desempeño luego de todas las pruebas
* El algoritmo “Quicksort” tuvo un desempeño similar al “Sort Interno” cuando los datos venían **desordenados con datos únicos o repetidos**. Su peor desempeño lo tuvo con datos **ordenados.**
* El algoritmo “Mergesort” ocupó generalmente el segundo lugar en cuando a tiempo de procesamiento.
* El algoritmo “Sort Interno” es el de más fácil implementación en C++, ya que no requiere la programación del algoritmo y solamente precisa del comando sort.
* La diferencia de rendimiento entre cada algoritmo se hace más evidente al aumentar el tamaño de la muestra.
* Se esperaba, de acuerdo a su definición, que los algoritmos “Quicksort”, “Merge sort”y “Sort Interno” tuviesen rendimientos similares, sin embargo, eso no ocurrió.

Por lo tanto, se puede concluir que, si bien existen algoritmos con mejor o peor desempeño, debe ser analizado en primera instancia el tipo de input a procesar, su tamaño y su grado de desorden.

## 5.2 Multiplicación de matrices

* El algoritmo “Strassen” para matrices cuadradas no tuvo un buen desempeño comparado con el algoritmo “Standard” y “Transpuesta”, sin embargo para matrices cuadradas de tamaño potencia de dos, y considerando matrices de pequeño tamaño, sí podría considerase competitivo respecto al tiempo.
* Respecto a los otros dos algoritmos, el de mejor desempeño siempre fue el de “Transpuesta”, destacándose aún más con matrices de gran tamaño.

# 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