# UNIVERSIDAD DIEGO PORTALES

**FACULTAD DE INGENIERÍA Y CIENCIAS**

**ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES**


---

## Estructura de Datos y Algoritmos

## Informe de Laboratorio N°3

---

**Profesor:**

**Alan Toro**

**Estudiante:**

**Gabriel González**

Santiago, Chile

Junio de 2024

# Índice

# 1. Resumen

En este laboratorio, se desarrolló la segunda parte del programa `Stopify`, añadiendo nuevas características centradas en la búsqueda y ordenamiento de datos. Se incorporaron nuevas estructuras de datos, como los árboles binarios, que ayudan a tener una búsqueda más rápida y a tener una organización más eficiente de los elementos. Además, se implementaron y compararon cuatro algoritmos de ordenamiento, en base al tiempo que tardan en ordenar por completo el grupo de canciones entregadas.

Entre los algoritmos de ordenamiento utilizados se encuentran el *Bubble Sort*, *Insertion Sort*, *Merge Sort* y *Quick Sort*, tras su aplicación y comparación, se determino que **en orden de mejor a peor** los algoritmos se clasifican de la siguiente forma *Merge Sort*, *Quick Sort*, *Insertion Sort* y por último, *Bubble Sort*.

Por ultimo, se realizó un pequeño análisis y comparación de los Arboles Binarios de Búsqueda **(BST)** balanceados y no balanceados, donde se determino que los arboles balanceados son más eficientes que los no balanceados.

# 2. Proceso de planificación

Para desarrollar el Laboratorio, primero se desarrolló una base sobre la cual implementar las estructuras y algoritmos solicitados, la cual se basaba en gran medida en la implementación pasada de `Stopify`, se creo la clase Song, la cual se puede implementar, al igual que en la primera implementación del programa, como *record class*, pero se optó por una implementación convencional de clase.

Tras esto, se implemento la base de la estructura de Arbol y se enlistaron los algoritmos en otro archivo auxiliar, dando por terminada la fase de planificación.

# 3. Implementación

La implementación de la clase *Song*, en este caso, *Cancion* se implemento de la siguiente forma

## 3.1. Cancion

```
1  package Implementacion;
2
3  public class Cancion {
4      private String nombre;
5      private int duracion;
6      private int id;
7
8      public Cancion(String nombre, int duracion) {
9          this.nombre = nombre;
10         this.duracion = duracion;
11     }
12
13     public String getNombre() { return nombre; }
14
15     public int getDuracion() { return duracion; }
16
17     public int getId() { return id; }
18
19     public void setId(int id) { this.id = id; }
20 }
```

## 3.2. Algoritmos de Ordenamiento

Si bien en la guía práctica no se solicita crear una implementación propia de los algoritmos de ordenamiento, de todas formas se implementaron los algoritmos a utilizar en un archivo propio, para poder entender mejor el funcionamiento de estos y facilitar el proceso de encontrar errores, este no fue un proceso sencillo ya que la implementación tuvo muchos errores, pero finalmente se implementaron todos los métodos correctamente, la implementación completa se encuentra en el archivo ***Sorts.java*** incluido en la carpeta donde ***Algoritmos***, a continuacion se enumeran y describen los algoritmos implementados

1. Merge Sort: Algoritmo de ordenamiento basado en el 'divide y vencerás'. Separa el arreglo principal en subarreglos unitarios y luego los reúne, ordenándolos en el proceso. Este algoritmo tiene una complejidad $\mathcal{O}(n \log n)$.
2. Quick Sort: Algoritmo de ordenamiento que también se basa en el 'divide y vencerás'. Se selecciona un pivote y y se divide el arreglo en dos subarreglos, uno con elementos menores que el pivote y otro con elementos mayores. Luego, ordena los subarreglos recursivamente. Su complejidad promedio es $\mathcal{O}(n \log n)$, aunque en el peor caso puede ser $\mathcal{O}(n^2)$.
3. Insertion Sort: Algoritmo de ordenamiento que construye la lista ordenada de a un elemento, tiene una complejidad de $\mathcal{O}(n^2)$ en el peor caso, pero $\mathcal{O}(n)$ en el mejor caso cuando los datos ya están casi ordenados.
4. Bubble Sort: Algoritmo de ordenamiento sencillo, se basa en comparar y intercambiar elementos adyacentes repetidamente hasta que la lista esté ordenada, de forma que 'burbujeen' hacía su posición. Su complejidad es $\mathcal{O}(n^2)$.

## 3.3. Arboles

A continuacion se muestra la implementación base de los arboles binarios, empleados en el código final

```
1  package Implementacion;
2
3  public class Tree {
4      public static class Node {
5          private Cancion data;
6          private Node left, right;
7
8          public Node(Cancion data){
9              this.data = data;
10             left = right = null;
11         }
12
13         public Cancion getData(){ return data; }
14
15         public Node getLeft(){ return left; }
16
17         public Node getRight(){ return right; }
18
19         public void setLeft(Node left){ this.left = left; }
20
21         public void setRight(Node right){ this.right = right; }
22     }
23
24     private Node root;
25
26     public Tree(){
27         root = null;
28     }
29
30     public Node getRoot(){ return root; }
```

### 3.3.1. Métodos de arboles binarios

```
    public Node insertar(Node root, Cancion data) {
        if (root == null) {
            root = new Node(data);
            return root;
        }

        Node current = root;
        Node parent = null;
        while (current != null) {
            parent = current;
            if (data.getNombre().compareTo(current.getData().getNombre()) < 0) {
                current = current.getLeft();
            } else if (data.getNombre().compareTo(current.getData().getNombre()) > 0) {
                current = current.getRight();
            } else {
                return null;
            }
        }

        if (data.getNombre().compareTo(parent.getData().getNombre()) < 0) {
            parent.setLeft(new Node(data));
        } else {
            parent.setRight(new Node(data));
        }

        return root;
    }

    public Node buscarPorNombre(Node root, String nombre){
        if (root == null || root.getData().getNombre().equals(nombre)) {
            return root;
        }

        if (nombre.compareTo(root.getData().getNombre()) < 0) {
            return buscarPorNombre(root.getLeft(), nombre);
        }

        return buscarPorNombre(root.getRight(), nombre);
    }

    public Node buscarPorDuracion(Node root, int duracion){
        if (root == null || root.getData().getDuracion() == duracion) {
            return root;
        }

        if (duracion < root.getData().getDuracion()) {
            return buscarPorDuracion(root.getLeft(), duracion);
        }

        return buscarPorDuracion(root.getRight(), duracion);
    }
}
```

Este código incluye los métodos para insertar y buscar por nombre o duración. La complejidad en el peor caso para las operaciones de inserción y búsqueda es $\mathcal{O}(n)$, donde n es el número de nodos en el árbol.

# 4. Implementación Stopify

La implementación de `Stopify` es muy larga para introducirla en el informe, por lo que solo se enlistarán los métodos junto con una breve descripción de cada uno, complementando a los comentarios que hay en el código.

- `insertar`: Añade una canción a la lista ($\mathcal{O}(n)$)
- `llamarGenerador`: Con la finalidad de hacer más simple y automatizada la prueba final de las comparaciones, se implemento este método que le envía el argumento con el total de canciones a generar a `TestGenerator.java`, el cual se ejecuta y **vuelve a generar** los tres archivos .csv con nuevos datos y un recuento actualizado, es una implementación simple pero que requirio investigación previa, ya que desconocía las funciones que hacen posible esta implementación.
- `cargarCanciones(String nombreArchivo)`: Lee el archivo .csv, crea las canciones (objetos) con los respectivos datos y los añade a la lista ($\mathcal{O}(n)$)
- `cargarDuraciones(String nombreArchivo)`: Lo mismo que el método anterior, pero modificado levemente para extraer correctamente las duraciones del archivo ($\mathcal{O}(n)$)
- `vaciarCanciones`: Vacía la lista de canciones $\mathcal{O}(n)$

# 5. Ejecución del programa

La ejecución del programa realiza una serie de pruebas en orden (Orden del arreglo - Creación de arboles - Busqueda por nombre - Busqueda por duración) y registra los resultados después de cada iteración, a continuación se muestra el output del programa tras una ejecución con un máximo de 100.000 canciones (se intento probar con 1.000.000 pero Bubble Sort tardaba demasiado en terminar el ordenamiento, por lo que se redujo el limite de las pruebas):

```
N° de canciones: 10

Bubble Sort: 983999 nanosegundos (9.83999E-4 segundos.)
Insertion Sort: 34800 nanosegundos (3.48E-5 segundos.)
Quick Sort: 23400 nanosegundos (2.34E-5 segundos.)
Merge Sort: 27800 nanosegundos (2.78E-5 segundos.)

N° de canciones: 100

Bubble Sort: 1230801 nanosegundos (0.001230801 segundos.)
Insertion Sort: 230901 nanosegundos (2.30901E-4 segundos.)
Quick Sort: 63300 nanosegundos (6.33E-5 segundos.)
Merge Sort: 79500 nanosegundos (7.95E-5 segundos.)

N° de canciones: 1.000

Bubble Sort: 10593600 nanosegundos (0.0105936 segundos.)
Insertion Sort: 5721300 nanosegundos (0.0057213 segundos.)
Quick Sort: 641200 nanosegundos (6.412E-4 segundos.)
Merge Sort: 786300 nanosegundos (7.863E-4 segundos.)

N° de canciones: 10.000

Bubble Sort: 514689100 nanosegundos (0.5146891 segundos.)
Insertion Sort: 117036400 nanosegundos (0.1170364 segundos.)
Quick Sort: 3083100 nanosegundos (0.0030831 segundos.)
Merge Sort: 3627900 nanosegundos (0.0036279 segundos.)

N° de canciones: 100.000

Bubble Sort: 63915612500 nanosegundos (63.9156125 segundos.)
Insertion Sort: 23471582400 nanosegundos (23.4715824 segundos.)
Quick Sort: 21550699 nanosegundos (0.021550699 segundos.)
Merge Sort: 29780899 nanosegundos (0.029780899 segundos.)

--------------------------------------------------------------------------------

Tiempo para construir el Árbol Desbalanceado: 3100401 ns (0.003100401 segundos.)
Tiempo para construir el Árbol Balanceado: 987000 ns (9.87E-4 segundos.)

N° de Queries (nombre): 10000

Tiempo para buscar canciones por nombre en el Árbol Desbalanceado: 1022400 ns (0.0010224 segundos.)
Tiempo para buscar canciones por nombre en el Árbol Balanceado: 550701 ns (5.50701E-4 segundos.)

N° de Queries (duración): 10000

Tiempo para buscar canciones por duración en el Árbol Desbalanceado: 754400 ns (7.544E-4 segundos.)
Tiempo para buscar canciones por duración en el Árbol Balanceado: 595601 ns (5.95601E-4 segundos.)
```

# 6. Análisis de los resultados

A continuación, se traspasan los datos obtenidos a una tabla, **la prueba se realizo un total de 8 veces**, se escogió este output por ser el más homogéneo de todas las pruebas, ya que al generarse las listas de forma aleatoria, el comportamiento puede variar, pero en este se muestra una curva de crecimiento como la esperada en un caso estándar, se ve que Bubble Sort es el peor, seguido por Insertion Sort, y en competencia está Merge y Quick Sort, se nota algo curioso, y es que Merge Sort es ligeramente peor que Quick Sort en todos los casos, se comentará esto más adelante:

| N° de canciones | Bubble Sort | Insertion Sort | Quick Sort | Merge Sort |
|---|---|---|---|---|
| 10 | 983999 ns | 34800 ns | 23400 ns | 27800 ns |
| 100 | 1230801 ns | 230901 ns | 63300 ns | 79500 ns |
| 1,000 | 10593600 ns | 5721300 ns | 641200 ns | 786300 ns |
| 10,000 | 514689100 ns | 117036400 ns | 3083100 ns | 3627900 ns |
| 100,000 | 63915612500 ns | 23471582400 ns | 21550699 ns | 29780899 ns |

Cuadro 1: Comparación de métodos de ordenamiento

Aquí se detallan los tiempos de ejecución para construir árboles desbalanceados y balanceados. Estas mediciones representan el rendimiento observado durante la construcción de los árboles utilizando 100.000 canciones, como es de esperarse, por la definición misma de los arboles, el árbol balanceado es más rápido y eficiente.

| Tipo de Árbol | Tiempo (ns) |
|---|---|
| Desbalanceado | 3100401 ns (0.003100401 segundos) |
| Balanceado | 987000 ns (9.87E-4 segundos) |

Cuadro 2: Comparación de tiempos de construcción de árboles

Por ultimo, se enlistan los tiempos para realizar las búsquedas en base a los queries generados, de nuevo, el arbol balanceado es superior:

| Tipo de Búsqueda | Tiempo (ns) |
|---|---|
| Por Nombre (Desbalanceado) | 1022400 ns (0.0010224 segundos) |
| Por Nombre (Balanceado) | 550701 ns (5.50701E-4 segundos) |
| Por Duración (Desbalanceado) | 754400 ns (7.544E-4 segundos) |
| Por Duración (Balanceado) | 595601 ns (5.95601E-4 segundos) |

Cuadro 3: Comparación de tiempos de búsqueda en árboles

### 6.1. Caso especial: Merge Sort

En el caso de la comparacion Merge - Quick Sort, se ve que Quick Sort es mas eficiente en todos los casos, esto sucede por la cantidad de datos analizados, ya que en promedio, Quick Sort tiene complejidad $(n \log n)$, al igual que el **peor** caso de Merge Sort, entonces puede pasar que en ciertos arreglos, Quick sea mejor que Merge, esta probabilidad disminuye al aumentar el tamaño de los datos, para confirmar esta teoría se ejecutó Merge Sort y Insertion Sort con 500.000 - 1.000.000 - 1.500.000 canciones, obteniendo los siguientes resultados:

| N° de canciones | Quick Sort | Merge Sort |
|---|---|---|
| 500,000 | 415199800 ns (0.4151998 segundos) | 349227200 ns (0.3492272 segundos) |
| 1,000,000 | 1064163400 ns (1.0641634 segundos) | 828521101 ns (0.828521101 segundos) |
| 1,500,000 | 1318022100 ns (1.3180221 segundos) | 1280740500 ns (1.2807405 segundos) |

Cuadro 4: Caso Merge - Quick

Aquí se observa claramente que a partir de mas o menos 500.000 datos, Merge Sort toma ventaja sobre Quick Sort, por lo que merge sort es superior en grandes cantidadesp pero menos consistente

## 7. Conclusión

Tras analizar todos los datos obtenidos, se puede concluir lo siguiente:

- Merge Sort es el algoritmo superior sobre el medio millon de datos, al menos ordenando por nombre y de tipo canción, esto puede variar si hablamos de otros objetos o tipos de datos
- Los arboles balanceados se construyen mas rapido y buscan datos más rápidos

En caso personal, considero que Merge Sort es mejor, dado a que ambos no son in-place, Merge Sort tiene mejor o similar rendimiento al de Quick Sort y Merge es un algoritmo estable, por lo que lo preferiría para una implementación similar, me hubiera gustado compararlo con Heap Sort, dado a que estamos trabajando con arboles binarios.