# UNIVERSIDAD DIEGO PORTALES

**FACULTAD DE INGENIERÍA Y CIENCIAS**

**ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES**


---

## Estructura de datos y algoritmos

## Informe de Laboratorio 2

---

**Profesor:**

**Rodrigo Álvarez**

**Estudiante:**

**Gabriel González**

Santiago, Chile

Mayo del 2024

# Índice

# 1. Resumen

El laboratorio consiste en implementar usando dos tipos de estructuras de datos un 'reproductor de música' bajo el nombre de **Stopify**, este reproductor está compuesto por tres clases principales.

- **Canción**
- **Playlist**
- **Reproductor**

Se hicieron dos implementaciones diferentes, una utilizando **Arrays** como la estructura principal para almacenar los datos, la segunda utilizando **Listas Enlazadas y Queue** (las cuales fueron definidas e implementadas en su totalidad en otros archivos, **no** se usaron las clases predefinidas de Java)

# 2. Implementación de Estructuras de Datos

El codigo de estas implementaciones es bastante extenso para explicarlo método a método, pero sigue un enfoque similar al de la implementación por defecto de Java, en este documento solo se enlistarán los métodos y se definirán las complejidades de cada uno.

## 2.1. Lista Enlazada

```
package Implementaciones;

public class listaEnlazada<E> {

    private Node<E> head;
    private int tamano;

    public static class Node<E> {
        private E data;
        private Node<E> next;

        public Node(E data) {
            this.data = data;
        }
    }

    public listaEnlazada() {
        head = null;
        tamano = 0;
    }

    public boolean isEmpty() { return head == null;} // O(1)

    public int size() { return tamano; } // O(1)

    public void add(E Data) { ... } // O(N)

    public void addAt(int index, E data) { // O(N)

    public E remove() { ... } // O(N)

    public boolean remove(E data) { ... } // O(N)

    public E get(int index) { ... } // O(N)

    public void set(int index, E data) { ... } // O(N)

```

### 2.2. Queue

```
package Implementaciones;

public class cola<E> extends listaEnlazada<E>{

    public cola() {
    }

    public void encolar(E data) {
        add(data);
    }

    public E desencolar() {
        return remove();
    }

    public E frente() {
        return get(0);
    }
}
```

## 3. Implementación de Clases

### 3.1. Canción

Esta clase es la encargada de almacenar los datos de una canción (objeto), dicho objeto está compuesto de 4 atributos:

- ***String*** Nombre
- ***String*** Artista
- ***String*** Álbum
- **Int** Duración

La implementación de esta clase se puede realizar de dos maneras, una es la implementación convencional de la clase, estableciendo un constructor, getters, setters y sus métodos, de la misma forma, se puede implementar como 'record class', este es un tipo de clase de Java que es muy útil cuando solo se usará la clase para almacenar datos, el constructor, los getters, los setters y el método **.equals()** se implementan automáticamente, la implementación es de solo una linea, aunque se pueden añadir métodos extra que utilicen las variables, pero no las modifiquen, dado a que se declaran como **final**.

#### 3.1.1. Implementación como Record Class

A continuación se muestra el código de la implementación de **Canción** como record class:

```
public record Cancion(String Nombre, String Artista, String Album, int Duracion) {

    public void imprimirInformacion() {
        System.out.println("Nombre: " + Nombre);
        System.out.println("Artista: " + Artista);
        System.out.println("Album: " + Album);
        System.out.println("Duración: " + Duracion + " segundos");
    }
}
```

Preferentemente, yo usaría está implementación, pero solo se encuentra disponible desde Java 14 en adelante, así que la implementación a usar en **Stopify** será la implementación convencional de la clase, para hacer el código más universal, la cual se mostrará a continuación.

### 3.1.2. Implementación convencional

```
public class Cancion {

    private final String nombre;
    private final String artista;
    private final String album;
    private final int duracion;

    public Cancion(String nombre, String artista, String album, int duracion) {
        this.nombre = nombre;
        this.artista = artista;
        this.album = album;
        this.duracion = duracion;
    }

    public String getNombre() {
        return nombre;
    }

    public String getArtista() {
        return artista;
    }

    public String getAlbum() {
        return album;
    }

    public int getDuracion() {
        return duracion;
    }

    public void getInformacion() {
        System.out.println("Nombre: " + getNombre());
        System.out.println("Artista: " + getArtista());
        System.out.println("Album: " + getAlbum());
        System.out.println("Duracion: " + getDuracion() / 60 + ":" + getDuracion() % 60);
    }

    public boolean equals(Cancion cancion) {
        if (cancion != null) {
            if (cancion.getNombre() != getNombre()) {
                return false;
            } else if (cancion.getArtista() != getArtista()) {
                return false;
            } else if (cancion.getAlbum() != getAlbum()) {
                return false;
            } else if (cancion.getDuracion() != getDuracion()){
                return false;
            } else {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

## 3.2. Playlist

Esta clase es la encargada de almacenar un grupo de objetos de tipo **Cancion**, la clase está compuesta de 2 atributos y 3 métodos.

Atributos:

- ***String*** Nombre
- Canciones

Métodos:

- Añadir a Playlist (anadirCancion)
- Eliminar de Playlist (eliminarCancion)
- Intercambiar Elementos (editarPlaylist)

### 3.2.1. Implementación con Array

Implementar está clase tuvo un poco de dificultad en la planificación, dado que los arreglos son una estructura estática y una implementación de este tipo requiere un enfoque dinámico, para poder lograr un comportamiento adecuado del programa con un uso responsable de memoria, se implementó la clase de la siguiente forma.

1. Se implemento la clase con un arreglo de tamaño 10 para almacenar las canciones
2. Se implementó una variable de control en el arreglo (**indice**, el cual es un entero que almacena la posición del primer espacio vacío en el arreglo, donde se puede agregar un objeto, esta variable también se usa para saber si el arreglo está lleno o aún tiene espacio disponible)
3. Se implementó el método **anadirCancion** de forma que tuviera control sobre el tamaño del arreglo y tuviera la capacidad de redimensionarlo de ser necesario.
4. También se usó un enfoque nuevo para la clase **eliminarCancion**, la cual tiene ahora la responsabilidad de de no dejar espacios vacíos en el arreglo tras eliminar una canción, para no tener un arreglo fragmentado y optimizar al máximo el uso de memoria

A continuación, se muestra la implementación final, explicada por partes

**Inicialización de la clase**

```
1  public class Playlist {
2      private String Nombre;
3      private Cancion[] Canciones;
4      private int indice;
5
6      public Playlist(String Nombre) {
7          this.Nombre = Nombre;
8          Canciones = new Cancion[10];
9          indice = 0;
10     }
11
12     // Getters y Setters
13
14     public String getNombre() { return Nombre; }
15     public void setNombre(String Nombre) { this.Nombre = Nombre; }
16     public Cancion[] getCanciones() { return Canciones; }
17     public Cancion getCancion(int index){ return Canciones[index]; }
18     public int size() { return indice - 1; }
19
20     // Métodos principales...
21
```

### Añadir canción

Este método recibe una canción, verifica si el índice donde va a insertar está dentro de los límites del arreglo, si no lo está, significa que el arreglo está lleno, si el arreglo aún tiene espacio, se añade la canción, si no hay espacio, se crea un nuevo arreglo con 1.5 veces el tamaño del arreglo original, se copian todas las canciones al nuevo arreglo y se cambia la referencia del arreglo original, al terminar, inserta la canción y actualiza el índice.

```
22      public void anadirCancion(Cancion c) {
23          if (indice >= Canciones.length) {
24              Cancion[] arregloAuxiliar = new Cancion[(int) (Canciones.length * 1.5)];
25              System.arraycopy(Canciones, 0, arregloAuxiliar, 0, Canciones.length);
26              Canciones = arregloAuxiliar;
27          }
28          Canciones[indice] = c;
29          indice++;
30      }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por *System.arraycopy(...)***

### Eliminar canción

Este método recibe una canción, luego, comienza a iterar por el arreglo desde el primer elemento en adelante en busca de la canción utilizando el método **.equals()**, al encontrar una coincidencia, establece el elemento en el índice como null, tras esto, se llama a otro bucle for (si es que la canción no estaba en la ultíma posición), iniciando desde el indice $i$ hasta la penúltima posición del arreglo, para mover las canciones una posición a la izquierda, compactando el arreglo, luego, se actualizan las variables y se asegura que no se duplique la última canción.

```
31      public boolean eliminarCancion(Cancion c) {
32          for (int i = 0; i < Canciones.length; i++) {
33              if (Canciones[i].equals(c)) {
34                  Canciones[i] = null;
35
36                  if (i < Canciones.length - 1) {
37                      for(int j = i; j < Canciones.length - 1; j++) {
38                          Canciones[j] = Canciones[j + 1];
39                      }
40                      Canciones[Canciones.length - 1] = null;
41                  }
42
43                  indice--;
44
45                  return true;
46              }
47          }
48          return false;
49      }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por el *bucle 'compuesto'*** [1]

[1] Si bien al haber un bucle anidado podría parecer que el método tiene complejidad $\mathcal{O}(N^2)$, en realidad el arreglo solo se recorre una vez, si no coincide la canción en ningún momento, el arreglo sale luego de iterar una vez, si encuentra la canción, entra al segundo bucle for, que comienza a iterar desde la posición en la que se está y termina de iterar en la penúltima posición, luego de esto se **retorna true**, lo que actúa como un **break** ya que sale por completo del método

**Editar playlist**

Este método recibe dos enteros que actuarán como índices de las canciones a intercambiar, primero, se verifica que los índices a intercambiar estén dentro del rango donde pueden haber canciones (entre 0 y el índice actual), si están, se almacena la canción en el índice 2 en una canción temporal, se sobrescribe esta canción con la canción del índice 1 y luego se sobrescribe la canción en el índice 1 con la canción temporal, si los índices no son los correctos, simplemente sale del método, en esta parte también se podría lanzar una excepción para dejar más claro el error, pero preferí dejarlo así:

```
50      public void editarPlaylist(int i, int j) {
51          if(i < indice && j < indice && i >= 0 && j >= 0) {
52              Cancion temp = Canciones[i];
53
54              Canciones[i] = Canciones[j];
55              Canciones[j] = temp;
56          } else {
57              return;
58          }
59      }
```

**Complejidad del método:** $\mathcal{O}(1)$

### 3.2.2. Implementación con Lista Enlazada

En esta implementación se usaron las implementaciones propias de Linked List **(listaEnlazada** y Queue **(cola)** definidas al principio, esta implementación es mucho más corta, eficiente y comprensible que la primera.

#### Inicialización de la clase

```
import Implementaciones.listaEnlazada;

public class Playlist {
    private String Nombre;
    private listaEnlazada<Cancion> canciones;

    Playlist(String Nombre) {
        this.Nombre = Nombre;
        canciones = new listaEnlazada<>();
    }

    public String getNombre() { return Nombre; }

    public void setNombre(String Nombre) { this.Nombre = Nombre; }

    public listaEnlazada<Cancion> getCanciones() { return canciones; }

    public Cancion getCancion(int index) { return canciones.get(index); }

```

#### Añadir canción

El método recibe una canción y añade el nuevo nodo a la lista usando **.add(*E Dato*)**

```
    public void agregarCancion(Cancion cancion) {
        canciones.add(cancion);
    }
```

**Complejidad del método:** $\mathcal{O}(N)$ **definida por *.add(E Dato)***

#### Eliminar canción

El método recibe una canción y la elimina usando el método **.remove(*E Dato*)**

```
    public void eliminarCancion(Cancion cancion) {
        canciones.remove(cancion);
    }
```

**Complejidad del método:** $\mathcal{O}(N)$ **definida por *.remove(E Dato)***

#### Editar playlist

Este método recibe dos índices, almacena ambas canciones temporalmente y las intercambia utilizando ***.set(int Indice, E Dato)***

```
    public void editarPlaylist(int i, int j) {
        Cancion cancion1 = canciones.get(i);
        Cancion cancion2 = canciones.get(j);

        canciones.set(i, cancion2);
        canciones.set(j, cancion1);
    }
```

**Complejidad del método:** $\mathcal{O}(N)$ **definida por *.set(int Indice, E Dato)*, con N = Índice**

## 3.3. Reproductor

Esta clase es la encargada de gestionar la cola de reproducción y de reproducir las mismas, está compuesta de dos atributos, un grupo de objetos de tipo Playlist y un grupo de objetos de tipo Canción (Cola de reproducción)

Atributos:

- Playlists
- Canciones (Cola de reproducción)

Métodos:

- Añadir canción a cola.
- Eliminar canción de la cola.
- Reproducir una cola de canciones en orden.
- Reproducir una playlist en orden.
- Reproducir una playlist de forma aleatoria.
- Reproducir dos playlists aleatoriamente de forma intercalada.

### 3.3.1. Implementación con Array

**Inicialización de la clase**

```
import Implementaciones.listaEnlazada;
import java.util.Random;

public class Reproductor {

    private Playlist[] playlists = new Playlist[10];
    private Cancion[] colaReproduccion = new Cancion[10];
    private int indice;

    Reproductor() {
        indice = 0;
    }

    public Playlist[] getPlaylists() { return playlists; }

    public Playlist getPlaylist(int index) { return playlists[index]; }

    public int size() { return (indice - 1); }

    public void anadirPlaylist(Playlist p) {...}

```

### Añadir canción a la cola

Este método recibe una instancia de la clase **Cancion** y la agrega a una cola de reproducción (**colaReproduccion**), este método actúa de la misma forma que **anadirCanción** en la clase Playlist.

```
22      public void agregarCancionCola(Cancion c) {
23          if (indice >= colaReproduccion.length) {
24              Cancion[] arregloAuxiliar = new Cancion[(int) (colaReproduccion.length * 1.5)];
25              System.arraycopy(colaReproduccion, 0, arregloAuxiliar, 0, colaReproduccion.length);
26              colaReproduccion = arregloAuxiliar;
27          }
28          colaReproduccion[indice] = c;
29          indice++;
30      }
```

**Complejidad del método:** $\mathcal{O}(N)$ **definida por** ***System.arraycopy(...)***

### Eliminar canción de la cola

Este método recibe una instancia de la clase **Cancion** y la elimina de la cola de reproducción (**colaReproduccion**), este método actúa de la misma forma que **eliminarCancion** en la clase Playlist.

```
31      public void eliminarCancionCola(Cancion c) {
32          for (int i = 0; i < colaReproduccion.length; i++) {
33              if (colaReproduccion[i].equals(c)) {
34                  colaReproduccion[i] = null;
35
36                  for (int j = i; j < colaReproduccion.length - 1; j++) {
37                      colaReproduccion[j] = colaReproduccion[j + 1];
38                  }
39
40                  colaReproduccion[colaReproduccion.length - 1] = null;
41                  indice--;
42              }
43          }
44      }
```

**Complejidad del método:** $\mathcal{O}(N)$ **definida por el** ***bucle 'compuesto'***

### Reproducir una cola de canciones en orden.

Este método recibe una lista de canciones (listaEnlazada¡Cancion¿lista) y la reproduce, usando **.remove()** para recuperar y eliminar el primer elemento de la lista, luego, llama a **.mostrarInformación()** para imprimir los detalles de la canción y así 'reproducirla'.

```
45      public void reproducirColaEnOrden(listaEnlazada<Cancion> lista) {
46          while (!lista.isEmpty()) {
47              lista.remove().imprimirInformacion();
48          }
49      }
```

**Complejidad del método:** $\mathcal{O}(N)$ **definida por el** ***bucle while***

### Reproducir una playlist en orden

Este método recibe un objeto de tipo **Playlist**, del cual reproduce todos sus elementos usando un bucle for.

```
50  public void reproducirPlaylistEnOrden(Playlist p) {
51      for (int i = 0; i < p.getCanciones().length; i++) {
52          p.getCancion(i).imprimirInformacion();
53      }
54  }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por el *bucle for*, con $N$ = Número de canciones en playlist**

### Reproducir una playlist en orden aleatorio

Este método recibe un objeto de tipo **Playlist** y reproduce sus canciones en un orden aleatorio. Si la playlist está vacía, imprime un mensaje indicando que está vacía. En caso contrario, crea una nueva lista de canciones llamada **colaAleatoria**, copia las canciones de la playlist original en esta lista utilizando el método **System.arraycopy()**, y luego utiliza un bucle for para intercambiar aleatoriamente las canciones en la lista **colaAleatoria** usando un generador de números aleatorios (la implementación del método está al final).

```
55  public void reproducirPlaylistAleatorio(Playlist p) {
56      if (p.getCanciones().length == 0) {
57          System.out.println("La playlist está vacía");
58      } else {
59          Cancion[] colaAleatoria = desordenar(p.getCanciones());
60
61          for (Cancion cancion : colaAleatoria) {
62              cancion.imprimirInformacion();
63          }
64      }
65  }
```

**Complejidad del método: $\mathcal{O}(N)$, donde $N$ es el número de canciones en la playlist, esta complejidad se define por el método .desordenar(Cancion[])**

**Reproducir intercalando aleatoriamente entre dos playlists**

Este método recibe dos objetos de tipo **Playlist**, `p1` y `p2`, y reproduce sus canciones intercaladamente en orden aleatorio. Primero, crea una nueva lista de canciones llamada `colaIntercalada`, luego, crea dos colas con las canciones de ambas playlist, las desordena y las encola de forma intercalada. tras esto, las reproduce.

```
66      public void reproducirIntercalandoAleatorio(Playlist p1, Playlist p2) {
67          Cancion[] colaIntercalada = new Cancion[p1.getCanciones().length + p2.getCanciones().length];
68
69          Cancion[] colaAleatoriaP1 = desordenar(p1.getCanciones());
70          Cancion[] colaAleatoriaP2 = desordenar(p2.getCanciones());
71
72          int menor = Math.min(colaAleatoriaP1.length, colaAleatoriaP2.length);
73
74          for(int i = 0; i < menor; i++) {
75              int j = 0;
76
77              colaIntercalada[j] = colaAleatoriaP1[i];
78              colaIntercalada[j + 1] = colaAleatoriaP2[i];
79
80              j+= 2;
81          }
82
83          for (Cancion cancion : colaIntercalada) {
84              cancion.imprimirInformacion();
85          }
86      }
```

**Complejidad del método:** $\mathcal{O}(N)$, con n el numero de canciones de la playlist más grande.

### 3.3.2. Implementación con Lista Enlazada

**Inicialización de la clase**

```
import Implementaciones.listaEnlazada;
import Implementaciones.cola;

import java.util.Random;

public class Reproductor {

    private listaEnlazada<Cancion> listaPlaylists = new listaEnlazada<>();
    private cola<Cancion> colaReproduccion = new cola<>();

    Reproductor() {
    }

    // Getters

    public listaEnlazada<Cancion> getListaPlaylists() {
        return listaPlaylists;
    }

    public cola<Cancion> getColaReproduccion() {
        return colaReproduccion;
    }
    ...
```

**Agregar una canción a la cola de reproducción**

Este método recibe una instancia de la clase **Cancion** y la agrega a la cola de reproducción (`colaReproduccion`). Utiliza el método `encolar()` para agregar la canción a la cola.

```
    public void agregarCancionCola(Cancion c) {
        colaReproduccion.encolar(c);
    }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por *.encolar(c)***

**Eliminar una canción de la cola de reproducción**

Este método recibe una instancia de la clase **Cancion** y elimina la canción correspondiente de la cola de reproducción (`colaReproduccion`). Utiliza el método `desencolar()` para eliminar la canción de la cola.

```
    public void eliminarCancionCola(Cancion c) {
        colaReproduccion.desencolar();
    }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por *.desencolar(c)***

**Reproducir la cola de reproducción en orden**

Este método reproduce todas las canciones en la cola de reproducción (`colaReproduccion`) en orden. Utiliza un bucle while que continúa hasta que la cola esté vacía. En cada iteración, se elimina la canción al principio de la cola utilizando el método `desencolar()` y se imprime su información utilizando el método `imprimirInformacion()`.

```
    public void reproducirColaEnOrden() {
        while (!colaReproduccion.isEmpty()) {
            colaReproduccion.desencolar().imprimirInformacion();
        }
    }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por el bucle*while***
**Reproducir una playlist en orden**

Este método recibe un objeto de tipo **Playlist** y reproduce todas sus canciones en orden. Utiliza un bucle for para iterar sobre todas las canciones en la playlist. En cada iteración, obtiene la canción en la posición actual utilizando el método `getCancion(i)` y luego imprime su información utilizando el método `imprimirInformacion()`.

```
35      public void reproducirPlaylistEnOrden(Playlist p) {
36          for (int i = 0; i < p.getCanciones().size(); i++) {
37              p.getCancion(i).imprimirInformacion();
38          }
39      }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por el bucle*for***
**Reproducir una playlist en orden aleatorio**

Este método recibe un objeto de tipo **Playlist** y reproduce todas sus canciones en un orden aleatorio. Primero, verifica si la playlist está vacía. Si lo está, imprime un mensaje indicando que está vacía y sale del método. En caso contrario, utiliza un generador de números aleatorios para reorganizar aleatoriamente las canciones en la playlist. Luego, utiliza un bucle for para iterar sobre todas las canciones en el nuevo orden y las imprime utilizando el método `imprimirInformacion()`.

```
40      public void reproducirPlaylistAleatorio(Playlist p) {
41          if (p.getCanciones().isEmpty()) {
42              System.out.println("La playlist está vacía");
43              return;
44          } else {
45              Random random = new Random();
46              int n = p.getCanciones().size();
47              for (int i = n - 1; i > 0; i--) {
48                  int j = random.nextInt(i + 1);
49                  Cancion temp = p.getCanciones().get(i);
50                  p.getCanciones().set(i, p.getCanciones().get(j));
51                  p.getCanciones().set(j, temp);
52              }
53
54              for (int i = 0; i < n; i++) {
55                  p.getCancion(i).imprimirInformacion();
56              }
57          }
58      }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por el bucle*for***

**Reproducir intercalando aleatoriamente entre dos playlists**

Este método recibe dos objetos de tipo **Playlist**, `p1` y `p2`, y reproduce sus canciones intercaladamente en un orden aleatorio. Primero, verifica si alguna de las playlists está vacía. Si alguna está vacía, imprime un mensaje indicando esto y termina la ejecución del método. En caso contrario, crea una cola de tipo `cola<Cancion>` llamada `colaIntercalada` para almacenar las canciones intercaladas. Luego, calcula la longitud de ambas playlists y determina la longitud mayor. Después, utiliza un bucle for para intercalar las canciones de ambas playlists en la cola `colaIntercalada`. Finalmente, utiliza el método auxiliar `desordenarCola` para desordenar la cola intercalada y luego reproducir las canciones en el nuevo orden.

```
59      public void reproducirIntercalandoAleatorio(Playlist p1, Playlist p2) {
60          if (p1.getCanciones().isEmpty() || p2.getCanciones().isEmpty()) {
61              System.out.println("Alguna de las playlists está vacía");
62          } else {
63              cola<Cancion> colaIntercalada = new cola<>();
64
65              cola<Cancion> cola1 = new cola<>();
66              cola<Cancion> cola2 = new cola<>();
67
68              for (int i = 0; i < p1.getCanciones().size(); i++) {
69                  cola1.encolar(p1.getCancion(i));
70              }
71
72              for (int i = 0; i < p2.getCanciones().size(); i++) {
73                  cola2.encolar(p2.getCancion(i));
74              }
75
76              cola1 = desordenarCola(cola1);
77              cola2 = desordenarCola(cola2);
78
79              while (!cola1.isEmpty() && !cola2.isEmpty()) {
80                  colaIntercalada.encolar(cola1.desencolar());
81                  colaIntercalada.encolar(cola2.desencolar());
82              }
83
84              while (!colaIntercalada.isEmpty()) {
85                  colaIntercalada.desencolar().imprimirInformacion();
86              }
87          }
88      }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por los bucles*for***
**Método auxiliar: Desordenar una cola**

Este método toma una cola de canciones como entrada y la desordena aleatoriamente. Crea una nueva cola llamada `listaDesordenada` y un generador de números aleatorios. Luego, mientras la cola original no esté vacía, elige aleatoriamente un índice en el rango del tamaño de la cola y desencola la canción en ese índice, agregándola a la nueva cola. Finalmente, devuelve la cola desordenada.

```
89      public cola<Cancion> desordenarCola(cola<Cancion> Cola) {
90          cola<Cancion> listaDesordenada = new cola<>();
91          Random random = new Random();
92
93          while (!Cola.isEmpty()) {
94              int n = Cola.size();
95              int i = random.nextInt(n);
96              Cancion cancion = Cola.desencolar();
```

```
97              listaDesordenada.addAt(i, cancion);
98          }
99          return listaDesordenada;
100         }
101     }
```

**Complejidad del método: $\mathcal{O}(N)$ definida por el bucle*while***

# 4. Conclusión

Tras terminar el laboratorio, me doy cuenta de las grandes diferencias entre una estructura de datos dinámica y una estática, las dificultades que entrega manejar distintas cantidades de datos con un enfoque estático en vez de uno dinámico.