“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

CARRERA: INGENIERÍA DE SOFTWARE

REPRODUCTOR DE MÚSICA PROGRAMADO EN LENGUAJE C Y C++

ESTRUCTURA DE DATOS

ESTUDIANTES:

Ortiz Castañeda Jorge Luis
Huamani Huamani Jhordan Steven Octavio (No hace nada)
Flores Leon Miguel Angel

DOCENTE:
Luque Mamani Edson Fracisco

Arequipa, Perú
30 de noviembre de 2024

Índice

I. Introducción

En este informe se presenta el desarrollo de una aplicación para un reproductor de música, implementada utilizando los lenguajes de programación C y C++, con la interfaz gráfica desarrollada mediante el framework Qt. El objetivo principal de la aplicación es proporcionar una experiencia de usuario intuitiva y fluida, facilitando la reproducción, búsqueda y gestión de canciones en un entorno de fácil acceso. La aplicación incorpora las funciones fundamentales basadas en la metodología CRUD (Crear, Leer, Actualizar y Eliminar), que es esencial para la manipulación y gestión de los datos de las canciones. Mediante este enfoque, el sistema permite al usuario agregar nuevas canciones a la biblioteca, leer la información almacenada, actualizar los detalles de las canciones existentes y eliminar aquellas que ya no sean necesarias.

Además de estas funcionalidades básicas, se ha integrado un sistema de búsqueda avanzada, que permite a los usuarios encontrar canciones de manera eficiente utilizando diversos criterios, tales como año de lanzamiento, género musical, autor, o incluso palabras clave dentro de los metadatos de las canciones. Esta funcionalidad proporciona una experiencia más personalizada y rápida al gestionar grandes colecciones de música.

Para la organización interna de los datos, hemos optado por utilizar listas enlazadas como estructura de datos principal. Esta elección se debe a su flexibilidad, eficiencia y facilidad de implementación en C y C++. Las listas enlazadas permiten un manejo eficiente de la memoria y proporcionan un rendimiento óptimo al manipular datos de tamaño variable, lo que resulta especialmente útil cuando se gestionan grandes cantidades de canciones.

La interfaz gráfica de usuario (GUI) ha sido desarrollada utilizando Qt, un framework de desarrollo de aplicaciones multiplataforma, lo que permite que la aplicación sea compatible con diferentes sistemas operativos sin necesidad de modificaciones significativas. Qt proporciona herramientas poderosas para crear interfaces intuitivas, y su integración con C++ facilita la creación de una aplicación robusta y eficiente.

II. ¿Por qué utilizar Linked List?

Aunque la estructura actual del proyecto se basa en árboles B, en las primeras etapas consideramos utilizar listas enlazadas debido a su facilidad de implementación y a la mayor legibilidad que ofrecen durante el proceso de revisión y difusión del código.

En este sentido, el uso de listas enlazadas nos permitió implementar una lógica básica que facilitó la comprensión de la complejidad involucrada en el manejo de miles o millones de datos provenientes de un archivo CSV (separado por comas).

III. Hablemos de los BTree

Los B-trees son una estructura de datos auto-equilibrada que garantiza búsquedas, inserciones y eliminaciones eficientes. Están diseñados específicamente para manejar grandes volúmenes de datos y optimizar el acceso en memoria secundaria como discos. Cada nodo en un B-tree puede contener múltiples claves, lo que reduce la profundidad del árbol y minimiza los accesos necesarios para realizar operaciones. Una ventaja clave sobre las listas enlazadas es su tiempo de búsqueda de $O(\log n)$, frente al $O(n)$ de las listas, ya que estas últimas requieren un recorrido secuencial para localizar un elemento.

IV. Utilizando Tries para la Búsquedas

Un trie es una estructura de datos en forma de árbol que se utiliza para almacenar palabras de manera que los prefijos compartidos entre ellas comparten un espacio común en el árbol. Esto hace que sea una opción ideal para realizar búsquedas eficientes basadas en prefijos.

Para buscar palabras por su prefijo, primero se construye el trie utilizando todas las palabras del conjunto. A partir de la raíz del trie, se sigue el camino que corresponde a los caracteres del prefijo. Si el prefijo existe en el trie, se identifica el nodo final del prefijo. Desde este nodo, se realiza un recorrido para recopilar todas las palabras completas que tienen dicho prefijo.

El trie permite realizar búsquedas rápidas y es especialmente útil cuando se trabaja con un gran número de palabras y se necesitan respuestas en tiempo eficiente. Además, aprovecha la estructura jerárquica para minimizar redundancias, ya que los prefijos comunes se almacenan una sola vez.

V. Utilizando Vectores para accesos inmediatos

VI. Uso del Vector para Buscar una Canción Aleatoria

Un **vector** es una estructura de datos secuencial que permite acceder a sus elementos en

tiempo constante ($O(1)$) mediante índices. Para buscar una canción aleatoria por su nombre:

- Almacenar las canciones en el vector, donde cada entrada contiene los atributos de la canción, incluyendo el nombre.
- Para seleccionar una canción aleatoria, generar un índice aleatorio entre 0 y el tamaño del vector menos uno ($n-1$), y acceder al elemento en esa posición.
- Esto es eficiente debido a que tanto el acceso por índice como la generación del índice aleatorio son operaciones rápidas.

VI-A. Aplicación de Paralelismo en el Ordenamiento por Atributo

El ordenamiento de canciones por atributo, como el nombre, el artista o la popularidad, puede optimizarse aplicando técnicas de paralelismo. Este enfoque divide el vector en múltiples subgrupos que se ordenan de manera simultánea en diferentes hilos o procesos. El procedimiento sería:

1. **División del vector:** El vector se divide en k subgrupos, donde k es el número de hilos disponibles. Cada subgrupo contiene una parte igual (o casi igual) de los elementos.
2. **Ordenamiento local:** Cada subgrupo es ordenado independientemente utilizando un algoritmo eficiente como Quicksort o Mergesort.
3. **Fusión paralela:** Los subgrupos ordenados se combinan en un único vector ordenado mediante un proceso de fusión paralelo, donde los hilos comparan y seleccionan elementos en cada paso.

VI-B. Ventajas del Paralelismo en Ordenamiento

El uso de paralelismo reduce significativamente el tiempo total de ordenamiento, especialmente en conjuntos de datos grandes. La complejidad se aproxima a $\frac{O(n \log n)}{k}$, donde n es el número de canciones y k el número de hilos. Además, este enfoque escala bien en arquitecturas con múltiples núcleos, aprovechando al máximo los recursos del hardware disponible.

En resumen, el vector es útil para acceder rápidamente a una canción aleatoria, y el paralelismo permite optimizar procesos intensivos como el ordenamiento por atributo de manera eficiente y escalable.

VII. Explicación del Código para el Backend

VII-A. btree.h

```cpp
#ifndef BTREE_H
#define BTREE_H

#include "cancion.h"
#include <vector>
#include <string>

class BTreeNode {
public:
    std::vector<Cancion> keys;
    std::vector<BTreeNode*> children;
    bool isLeaf;
    int t;

    BTreeNode(int t, bool isLeaf);

    void insertNonFull(Cancion k);
    void splitChild(int i, BTreeNode* y);
    void traverse();
    BTreeNode* search(const std::string&
        ↪ key, bool searchByArtist);
    void searchAll(const std::string& key,
        ↪ bool searchByArtist,
        ↪ std::vector<Cancion>& result);

};

class BTree {
public:
    BTreeNode* root;
    int t;

    BTree(int t);

    void traverse();
    BTreeNode* search(const std::string&
        ↪ key, bool searchByArtist);
    void insert(Cancion k);
    void searchAll(const std::string& key,
        ↪ bool searchByArtist,
        ↪ std::vector<Cancion>& result);
};

#endif // BTREE_H
```

Listing 1. Cabecera de un Árbol B

Este código define las clases necesarias para implementar un **B-Tree**, una estructura de datos balanceada útil para almacenar grandes conjuntos de datos de manera eficiente. Está dividido en dos partes principales: la clase **BTreeNode** para los nodos individuales del árbol y la clase **BTree** para representar el árbol completo.

VII-A1. Directivas de Preprocesador: El archivo comienza con las directivas de preprocesador `#ifndef`, `#define` y `#endif` para evitar que el archivo de cabecera sea incluido múltiples veces en el programa. Luego, incluye los archivos necesarios: `cancion.h`, que probablemente define la estructura o clase **Cancion**, y las bibliotecas estándar **vector** y **string**.

VII-A2. Clase BTreeNode: La clase **BTreeNode** representa un nodo del B-Tree. Contiene:

- Un vector de objetos **Cancion** llamado **keys** que almacena las claves en el nodo.

- Un vector de punteros a nodos **children** que almacena los hijos del nodo.
- Un booleano **isLeaf** que indica si el nodo es una hoja.
- Un entero **t** que representa el grado mínimo del B-Tree.

Además, la clase **BTreeNode** incluye varios métodos importantes:

- **Constructor**: Inicializa un nodo con el grado mínimo **t** y un indicador de si es **hoja**.
- **insertNonFull**: Inserta una **clave** en un nodo que no está **lleno**.
- **splitChild**: Divide un **hijo lleno** en dos nodos más **pequeños**.
- **traverse**: Recorre y muestra las **claves** en el nodo y sus **hijos**.
- **search**: Busca una **clave** en el nodo y sus **descendientes**, con la opción de buscar por **nombre del artista** (**searchByArtist**).
- **searchAll**: Busca todas las **canciones** que coincidan con un **criterio** y las almacena en un **vector de resultados**.

VII-A3. Clase BTree: La clase **BTree** representa el **árbol** en su totalidad. Contiene:

- Un **puntero root** que apunta al **nodo raíz** del **árbol**.
- Un **entero t** que especifica el **grado mínimo** del **árbol**.

Sus métodos incluyen:

- **Constructor**: Inicializa un **árbol vacío** con el **grado mínimo t**.
- **traverse**: Recorre y muestra todas las **claves** del **árbol**.
- **search**: Busca una **clave** en todo el **árbol** utilizando la función correspondiente en los **nodos**.
- **insert**: Inserta una **clave** en el **árbol**, manejando casos especiales como cuando la **raíz** está **llena**.
- **searchAll**: Realiza una **búsqueda exhaustiva** en todo el **árbol** y devuelve los **resultados coincidentes**.

VII-A4. Características del B-Tree: Este código utiliza vectores y punteros para manejar la estructura jerárquica del B-Tree, lo que permite una gestión dinámica de memoria y escalabilidad para almacenar claves y nodos. La estructura es adecuada para aplicaciones como bases de datos y sistemas de archivos donde la eficiencia en la búsqueda y actualización de datos es crucial.

VII-B. btree.cpp

```cpp
#include "btree.h"
#include <iostream>

BTree::BTree(int t) : t(t), root(nullptr) {}

BTreeNode::BTreeNode(int t, bool isLeaf) : t(t), isLeaf(isLeaf) {
    keys.reserve(2 * t - 1);
    children.reserve(2 * t);
}

void BTreeNode::traverse() {
    int i;
    for (i = 0; i < keys.size(); i++) {
        if (!isLeaf) {
            children[i]->traverse();
        }
        keys[i].imprimirDatos();
    }
    if (!isLeaf) {
        children[i]->traverse();
    }
}

BTreeNode* BTreeNode::search(const std::string& key, bool searchByArtist) {
    int i = 0;
    while (i < keys.size() && (searchByArtist ? keys[i].artist_name : keys[i].track_name) < key) {
        i++;
    }

    if (i < keys.size() && (searchByArtist ? keys[i].artist_name : keys[i].track_name) == key) {
        return this;
    }

    if (isLeaf) {
        return nullptr;
    }

    return children[i]->search(key, searchByArtist);
}

void BTreeNode::searchAll(const std::string& key, bool searchByArtist, std::vector<Cancion>& result) {
    int i = 0;
    while (i < keys.size() && (searchByArtist ? keys[i].artist_name : keys[i].track_name) < key) {
        i++;
    }

    if (i < keys.size() && (searchByArtist ? keys[i].artist_name : keys[i].track_name) == key) {
        result.push_back(keys[i]);
    }

    if (isLeaf) {
        return;
    }

    for (int j = 0; j <= keys.size(); j++) {
```

```
        children[j]->searchAll(key, searchByArtist, result);
    }
}

void BTree::traverse() {
    if (root != nullptr) {
        root->traverse();
    }
}

BTreeNode* BTree::search(const std::string& key, bool searchByArtist) {
    return (root == nullptr) ? nullptr : root->search(key, searchByArtist);
}

void BTree::searchAll(const std::string& key, bool searchByArtist, std::vector<Cancion>& result) {
    if (root != nullptr) {
        root->searchAll(key, searchByArtist, result);
    }
}

void BTree::insert(Cancion k) {
    if (root == nullptr) {
        root = new BTreeNode(t, true);
        root->keys.push_back(k);
    } else {
        if (root->keys.size() == 2 * t - 1) {
            BTreeNode* s = new BTreeNode(t, false);
            s->children.push_back(root);
            s->splitChild(0, root);

            int i = 0;
            if ((s->keys[0].artist_name < k.artist_name) || (s->keys[0].track_name < k.track_name)) {
                i++;
            }
            s->children[i]->insertNonFull(k);

            root = s;
        } else {
            root->insertNonFull(k);
        }
    }
}

void BTreeNode::insertNonFull(Cancion k) {
    int i = keys.size() - 1;

    if (isLeaf) {
        keys.push_back(k);
        while (i >= 0 && (keys[i].artist_name > k.artist_name || keys[i].track_name > k.track_name)) {
            keys[i + 1] = keys[i];
            i--;
        }
        keys[i + 1] = k;
    } else {
        while (i >= 0 && (keys[i].artist_name > k.artist_name || keys[i].track_name > k.track_name)) {
            i--;
        }
        if (children[i + 1]->keys.size() == 2 * t - 1) {
            splitChild(i + 1, children[i + 1]);
            if ((keys[i + 1].artist_name < k.artist_name) || (keys[i + 1].track_name < k.track_name)) {
                i++;
            }
        }
        children[i + 1]->insertNonFull(k);
    }
}

void BTreeNode::splitChild(int i, BTreeNode* y) {
    BTreeNode* z = new BTreeNode(y->t, y->isLeaf);
    z->keys.assign(y->keys.begin() + t, y->keys.end());
    y->keys.resize(t - 1);

    if (!y->isLeaf) {
        z->children.assign(y->children.begin() + t, y->children.end());
        y->children.resize(t);
    }

    children.insert(children.begin() + i + 1, z);
    keys.insert(keys.begin() + i, y->keys[t - 1]);
}
```

Listing 2. Código de un Árbol B

Este código implementa un *Árbol B* (B-tree), una estructura de datos autoequilibrada que permite realizar operaciones de búsqueda, inserción y eliminación en tiempo logarítmico. Se utiliza comúnmente en bases de datos y sistemas de archivos debido a su eficiencia en la gestión de grandes volúmenes de datos. El código está compuesto por dos clases principales: **BTree** y **BTreeNode**. La clase **BTree** es la estructura de alto nivel que maneja el árbol en su conjunto, mientras que **BTreeNode** representa un nodo individual del árbol.

VII-B1. Desarrollando la Clase BTree : La clase **BTree** tiene como miembro principal la raíz del árbol (**root**) y un parámetro **t** que especifica el orden del árbol. El constructor de **BTree** toma un valor **t** como argumento y establece la raíz en **nullptr**. La clase **BTree** implementa varias funciones importantes:

- **traverse**: Recorre el árbol imprimiendo los datos de cada nodo, comenzando desde la raíz. Si el nodo actual no es una hoja, la función recurre a sus hijos antes de imprimir sus claves.
- **search**: Busca una clave en el árbol. Si la raíz es **nullptr**, lo que indica que el árbol está vacío, la función retorna **nullptr**. Si la raíz existe, delega la

búsqueda a la función **search** de los nodos.

- **searchAll**: Realiza una búsqueda en todo el árbol y recopila todas las instancias de una clave en un vector de resultados, dependiendo del criterio de búsqueda (ya sea por nombre del artista o por nombre de la pista).
- **insert**: Inserta una nueva canción en el árbol. Si la raíz está vacía, crea un nuevo nodo raíz con la canción, pero si la raíz ya está llena, se divide en dos nodos, y luego se inserta la canción en el nodo adecuado.

VII-B2. Desarrollando la Clase BTreeNode: Recordemos que la clase **BTreeNode** representa un nodo del árbol y contiene varias claves **keys**, hijos **children**, y un indicador **isLeaf** que señala si el nodo es una hoja. Esta clase también tiene funciones importantes para manejar el nodo:

- **Constructor**: Inicializa el valor de **t**, establece si el nodo es una hoja y reserva espacio para las claves y los hijos.
- **traverse**: Recorre el nodo y sus hijos, imprimiendo las claves almacenadas. Si el nodo no es una hoja, también recurre a sus hijos.
- **search**: Busca una clave dentro del nodo actual, comparando las claves de los nodos hijos. Si la clave se encuentra en el nodo, la función retorna el nodo; de lo contrario, la búsqueda continúa en el hijo correspondiente.
- **searchAll**: Realiza una búsqueda recursiva para encontrar todas las instancias de una clave en el árbol. Compara la clave con las claves de los nodos hijos y agrega todas las canciones que coinciden en el vector de resultados.
- **insertNonFull**: Maneja la inserción de una clave en un nodo que no está lleno, asegurándose de que las claves se mantengan ordenadas. Si el nodo no es una hoja, primero recurre al hijo adecuado antes de insertar la clave. Si el nodo es una hoja, la clave se inserta directamente en la posición correcta.

VII-B3. Función splitChild: La función **splitChild** maneja la división de un nodo lleno. Cuando un nodo tiene más claves de las que puede almacenar (es decir, cuando el número de claves alcanza **2 * t - 1**), se divide en dos nodos. La clave del medio del nodo original se mueve hacia el nodo padre, y el nodo original se divide en dos partes, creando un nuevo nodo. Si el nodo no es una hoja, los hijos también se dividen.

VII-C. cancion.h

```
#ifndef CANCION_H
#define CANCION_H

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Cancion {
public:
    int id;
    string artist_name;
    string track_name;
    string track_id;
    int popularity;
    int year;
    string genre;
    double danceability;
    double energy;
    int key;
    double loudness;
    int mode;
    double speechiness;
    double acousticness;
    double instrumentalness;
    double liveness;
    double valence;
    double tempo;
    int duration_ms;
    int time_signature;

    Cancion() : id(0), popularity(0),
        ↪ year(0), danceability(0.0),
        ↪ energy(0.0), key(0),
        ↪ loudness(0.0), mode(0),
        ↪ speechiness(0.0),
        ↪ acousticness(0.0),
        ↪ instrumentalness(0.0),
        ↪ liveness(0.0), valence(0.0),
        ↪ tempo(0.0), duration_ms(0),
        ↪ time_signature(0) {}

    Cancion(int id, string artist_name,
        ↪ string track_name, string
        ↪ track_id, int popularity, int
        ↪ year,
                string genre, double
                    ↪ danceability,
                    ↪ double energy, int
                    ↪ key, double
                    ↪ loudness, int mode,
                double speechiness,
                    ↪ double
                    ↪ acousticness,
                    ↪ double
                    ↪ instrumentalness,
                    ↪ double liveness,
                double valence, double
                    ↪ tempo, int
                    ↪ duration_ms, int
                    ↪ time_signature);

    void imprimirDatos();
    void reproducirCancion();
};

#endif // CANCION_H
```

Listing 3. Cabecera de la Clase Canción

VII-C1. Introducción: Este código define la cabecera de una clase **Cancion**, que se utiliza para representar una canción con varias propiedades

musicales y de análisis de datos. La clase contiene atributos como el nombre del artista, el nombre de la pista, la popularidad, el año de lanzamiento, y diversos parámetros relacionados con las características de la canción, como la danza, energía, tono, entre otros. La cabecera también declara dos funciones principales: **imprimirDatos** y **reproducirCancion**.

VII-C2. Miembros de la clase: La clase **Cancion** tiene varios atributos públicos que representan distintas características de una canción, incluyendo:

- **id**: Identificador único de la canción.
- **artist_name**: Nombre del artista o banda.
- **track_name**: Nombre de la pista.
- **track_id**: Identificador único de la pista.
- **popularity**: Popularidad de la canción en una escala.
- **year**: Año de lanzamiento de la canción.
- **genre**: Género musical de la canción.
- **danceability**: Índice de cuán fácil es bailar al ritmo de la canción.
- **energy**: Nivel de energía de la canción.
- **key**: Tono musical de la canción.
- **loudness**: Volumen de la canción en decibelios.
- **mode**: Modalidad de la canción (mayor o menor).
- **speechiness**: Proporción de habla en la canción.
- **acousticness**: Nivel de acusticidad de la canción.
- **instrumentalness**: Porcentaje de la canción que es instrumental.
- **liveness**: Proporción de audibilidad en un contexto en vivo.
- **valence**: Indicador del estado de ánimo de la canción.
- **tempo**: Tempo de la canción en beats por minuto.
- **duration_ms**: Duración de la canción en milisegundos.
- **time_signature**: Firma temporal de la canción (por ejemplo, 4/4).

VII-C3. Constructores: La clase **Cancion** tiene dos constructores:

- **Cancion()**: Constructor por defecto que inicializa todos los atributos con valores predeterminados.
- **Cancion(int id, string artist_name, string track_name, string track_id, int popularity, int year, string genre, double danceability, double energy, int key, double loudness, int mode, double speechiness, double acousticness, double instrumentalness, double liveness, double valence, double tempo, int duration_ms, int time_signature)**: Constructor que recibe parámetros específicos para inicializar los atributos de la canción con valores definidos al crear una instancia de la clase.

VII-C4. Funciones: La clase **Cancion** declara las siguientes funciones:

- **imprimirDatos()**: Función miembro que probablemente se encargará de imprimir la información de la canción en un formato legible.
- **reproducirCancion()**: Función miembro que probablemente se utilizará para reproducir la canción, aunque la implementación no está incluida en esta cabecera.

VII-C5. Conclusión: La clase **Cancion** está diseñada para encapsular una serie de propiedades que describen una canción y sus características relacionadas. Esta estructura es útil para representar canciones en una aplicación o base de datos que maneje información musical, permitiendo fácilmente la manipulación y presentación de datos musicales.

VII-D. cancion.cpp

```
#include "cancion.h"

Cancion::Cancion(int id, string artist_name, string track_name, string track_id, int popularity, int year,
                 string genre, double danceability, double energy, int key, double loudness, int mode,
                 double speechiness, double acousticness, double instrumentalness, double liveness,
                 double valence, double tempo, int duration_ms, int time_signature)
    : id(id), artist_name(artist_name), track_name(track_name), track_id(track_id), popularity(popularity),
      year(year), genre(genre), danceability(danceability), energy(energy), key(key), loudness(loudness),
      mode(mode), speechiness(speechiness), acousticness(acousticness), instrumentalness(instrumentalness),
      liveness(liveness), valence(valence), tempo(tempo), duration_ms(duration_ms), time_signature(time_signature)
      {}
```

```
11
12 void Cancion::imprimirDatos() {
13     cout << "|" << setw(5) << this->id
14          << "|" << setw(30) <<
                ↪ this->artist_name
15          << "|" << setw(30) <<
                ↪ this->track_name
16          << "|" << setw(30) <<
                ↪ this->track_id
17          << "|" << setw(5) <<
                ↪ this->popularity
18          << "|" << setw(5) << this->year
19          << "|" << setw(10) << this->genre
20          << "|" << setw(5) <<
                ↪ this->danceability
21          << "|" << setw(5) << this->energy
22          << "|" << setw(5) << this->key
23          << "|" << setw(5) <<
                ↪ this->loudness
24          << "|" << setw(5) << this->mode
25          << "|" << setw(5) <<
                ↪ this->speechiness
26          << "|" << setw(5) <<
                ↪ this->acousticness
27          << "|" << setw(5) <<
                ↪ this->instrumentalness
28          << "|" << setw(5) <<
                ↪ this->liveness
29          << "|" << setw(5) << this->valence
30          << "|" << setw(5) << this->tempo
31          << "|" << setw(5) <<
                ↪ this->duration_ms
32          << "|" << setw(5) <<
                ↪ this->time_signature
33          << "|" << endl;
34 }
35
36 void Cancion::reproducirCancion() {
37     cout << "Reproduciendo: " <<
           ↪ this->track_name << " - " <<
           ↪ this->artist_name << endl;
38 }
```

Listing 4. Código de un Clase Canción

VII-D1. Introducción: Este código define la cabecera de una clase **Cancion**, que se utiliza para representar una canción con varias propiedades musicales y de análisis de datos. La clase contiene atributos como el nombre del artista, el nombre de la pista, la popularidad, el año de lanzamiento, y diversos parámetros relacionados con las características de la canción, como la danza, energía, tono, entre otros. La cabecera también declara dos funciones principales: **imprimirDatos** y **reproducirCancion**.

VII-D2. Miembros de la clase: La clase **Cancion** tiene varios atributos públicos que representan distintas características de una canción, incluyendo:

- **id**: Identificador único de la canción.
- **artist_name**: Nombre del artista o banda.
- **track_name**: Nombre de la pista.
- **track_id**: Identificador único de la pista.
- **popularity**: Popularidad de la canción en una escala.
- **year**: Año de lanzamiento de la canción.
- **genre**: Género musical de la canción.
- **danceability**: Índice de cuán fácil es bailar al ritmo de la canción.
- **energy**: Nivel de energía de la canción.
- **key**: Tono musical de la canción.
- **loudness**: Volumen de la canción en decibelios.
- **mode**: Modalidad de la canción (mayor o menor).
- **speechiness**: Proporción de habla en la canción.
- **acousticness**: Nivel de acusticidad de la canción.
- **instrumentalness**: Porcentaje de la canción que es instrumental.
- **liveness**: Proporción de audibilidad en un contexto en vivo.
- **valence**: Indicador del estado de ánimo de la canción.
- **tempo**: Tempo de la canción en beats por minuto.
- **duration_ms**: Duración de la canción en milisegundos.
- **time_signature**: Firma temporal de la canción (por ejemplo, 4/4).

VII-D3. Constructores: La clase **Cancion** tiene dos constructores:

- **Cancion()**: Constructor por defecto que inicializa todos los atributos con valores predeterminados.
- **Cancion(int id, string artist_name, string track_name, string track_id, int popularity, int year, string genre, double danceability, double energy, int key, double loudness, int mode, double speechiness, double acousticness, double instrumentalness, double liveness, double valence, double tempo, int duration_ms, int time_signature)**: Constructor que recibe parámetros específicos para inicializar los atributos de la canción con valores definidos al crear una instancia de la clase.

VII-D4. Funciones: La clase **Cancion** declara las siguientes funciones:

- **imprimirDatos()**: Función miembro que probablemente se encargará de imprimir la información de la canción en un formato legible.
- **reproducirCancion()**: Función miembro que probablemente se utilizará para reproducir la canción, aunque la implementación no está incluida en esta cabecera.

VII-E. playlist.h

```cpp
#ifndef PLAYLIST_H
#define PLAYLIST_H

#include "cancion.h"
#include "btree.h"
#include <vector>
#include <fstream>
#include <sstream>
#include <random>

class PlayList {
public:
    BTree* btree;
    std::vector<Cancion> todasLasCanciones; // Vector para almacenar todas las canciones

    PlayList(int t);
    ~PlayList();

    void agregarCancion(Cancion& cancion);
    vector<Cancion> buscarPorNombre(const std::string& nombre, bool searchByArtist);
    void cargarCSV(const std::string& nombre_archivo);
    void imprimirCanciones();
    void ordenarPorAtributo(const std::string& atributo);
    Cancion reproduccionAleatoria();
    bool actualizarCancion(int id, const Cancion& nuevaCancion);
};

#endif // PLAYLIST_H
```

Listing 5. Cabecera de la Clase Playlist

VII-E1. Introducción: Este código define la cabecera de la clase **PlayList**, que gestiona una lista de canciones, almacenando las canciones en un *Árbol B* y un vector. La clase ofrece varias funciones para agregar canciones, buscar canciones por nombre, cargar canciones desde un archivo CSV, imprimir las canciones almacenadas, ordenarlas según un atributo específico, y realizar la reproducción aleatoria de canciones. También incluye una función para actualizar la información de una canción existente.

VII-E2. Atributos: La clase **PlayList** tiene los siguientes atributos principales:

- **btree**: Un puntero a un objeto de la clase **BTree** que organiza las canciones en un árbol para facilitar las búsquedas y otras operaciones.
- **todasLasCanciones**: Un vector de objetos **Cancion** que almacena todas las canciones de la lista de reproducción.

VII-E3. Constructor y Destructor: La clase **PlayList** tiene un constructor y un destructor:

- **PlayList(int t)**: El constructor toma un parámetro **t**, que representa el grado mínimo del *Árbol B* (**BTree**) utilizado para almacenar las canciones. Inicializa el puntero **btree** y crea el vector **todasLasCanciones**.
- **PlayList()**: El destructor se encarga de liberar cualquier recurso utilizado por la clase, como el puntero **btree**.

VII-E4. Funciones: La clase **PlayList** proporciona las siguientes funciones miembros:

- **agregarCancion(Cancion& cancion)**: Esta función agrega una canción al vector **todasLasCanciones** y a la estructura del *Árbol B* (**btree**).
- **buscarPorNombre(const std::string& nombre, bool searchByArtist)**: Permite buscar canciones en la lista de reproducción por nombre, ya sea por el nombre del artista o por el nombre de la pista.
- **cargarCSV(const std::string& nombre_archivo)**: Carga las canciones desde un archivo CSV, parseando el contenido del archivo y agregando las canciones a la lista y al *Árbol B*.
- **imprimirCanciones()**: Imprime la lista de todas las canciones almacenadas en el vector **todasLasCanciones**.
- **ordenarPorAtributo(const std::string& atributo)**: Ordena las canciones de la lista de reproducción por un atributo específico, como el nombre, la popularidad o el año.
- **reproduccionAleatoria()**: Devuelve una canción aleatoria de la lista de reproducción.
- **actualizarCancion(int id, const Cancion& nuevaCancion)**: Actualiza los detalles de una canción existente, identificada por su **id**, con la nueva información proporcionada en **nuevaCancion**.

VII-F. playlist.cpp

```cpp
// playlist.cpp
#include "playlist.h"
#include <iostream>
#include <algorithm>
#include <execution>

PlayList::PlayList(int t) {
    btree = new BTree(t);
}

PlayList::~PlayList() {
    delete btree;
}

void PlayList::agregarCancion(Cancion& cancion) {
    btree->insert(cancion);
    todasLasCanciones.push_back(cancion); // Añadir la canción al vector
}

vector<Cancion> PlayList::buscarPorNombre(const std::string& nombre, bool searchByArtist) {
```

```
    vector<Cancion> resultados;
    btree->searchAll(nombre, searchByArtist, resultados);
    return resultados;
}

void PlayList::cargarCSV(const std::string& nombre_archivo) {
    ifstream archivo(nombre_archivo);
    if (!archivo.is_open()) {
        cerr << "No se pudo abrir el archivo " << nombre_archivo << endl;
        return;
    }

    string linea;
    getline(archivo, linea);  // Leer la cabecera del CSV
    while (getline(archivo, linea)) {
        stringstream ss(linea);
        string token;

        auto leerCampo = [&ss]() {
            string campo;
            char ch;
            bool dentroComillas = false;

            while (ss.get(ch)) {
                if (ch == '"' && !dentroComillas) {
                    dentroComillas = true;
                } else if (ch == '"' && dentroComillas) {
                    if (ss.peek() == ',') {
                        ss.get();
                        break;
                    }
                    dentroComillas = false;
                } else if (ch == ',' && !dentroComillas) {
                    break;
                } else {
                    campo += ch;
                }
            }
            return campo;
        };

        int id = stoi(leerCampo());
        string artist_name = leerCampo();
        string track_name = leerCampo();
        string track_id = leerCampo();
        int popularity = stoi(leerCampo());
        int year = stoi(leerCampo());
        string genre = leerCampo();
        double danceability = stod(leerCampo());
        double energy = stod(leerCampo());
        int key = stoi(leerCampo());
        double loudness = stod(leerCampo());
        int mode = stoi(leerCampo());
        double speechiness = stod(leerCampo());
        double acousticness = stod(leerCampo());
        double instrumentalness = stod(leerCampo());
        double liveness = stod(leerCampo());
        double valence = stod(leerCampo());
        double tempo = stod(leerCampo());
        int duration_ms = stoi(leerCampo());
        int time_signature = stoi(leerCampo());

        Cancion cancion(id, artist_name, track_name, track_id, popularity, year, genre, danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo, duration_ms, time_signature);
        agregarCancion(cancion);
    }

    archivo.close();
}

void PlayList::imprimirCanciones() {
    btree->traverse();
}

void PlayList::ordenarPorAtributo(const std::string& atributo) {
    auto comparar = [&atributo](const Cancion& a, const Cancion& b) {
        if (atributo == "popularidad") {
            return a.popularity < b.popularity;
        } else if (atributo == "anio") {
            return a.year < b.year;
        } else if (atributo == "artista") {
            return a.artist_name < b.artist_name;
        } else if (atributo == "cancion") {
            return a.track_name < b.track_name;
        } else if (atributo == "genero") {
            return a.genre < b.genre;
        } else if (atributo == "duracion") {
            return a.duration_ms < b.duration_ms;
        } else if (atributo == "tempo") {
            return a.tempo < b.tempo;
        }
        return false;
    };

    sort(std::execution::par,todasLasCanciones.begin(), todasLasCanciones.end(), comparar);

    cout << "Canciones después de ordenar por " << atributo << ":" << endl;
    for (const auto& cancion : todasLasCanciones) {
        if (atributo == "popularidad") {
            cout << cancion.popularity << " - " << cancion.track_name << endl;
        } else if (atributo == "anio") {
            cout << cancion.year << " - " << cancion.track_name << endl;
        } else if (atributo == "artista") {
            cout << cancion.artist_name << " - " << cancion.track_name << endl;
        } else if (atributo == "cancion") {
            cout << cancion.track_name << endl;
        } else if (atributo == "genero") {
            cout << cancion.genre << " - " << cancion.track_name
```

```
                << endl;
        } else if (atributo == "duracion")
            {
            cout << cancion.duration_ms <<
                " ms - " <<
                cancion.track_name <<
                endl;
        } else if (atributo == "tempo") {
            cout << cancion.tempo << " - "
                << cancion.track_name
                << endl;
        }
    }
}

Cancion PlayList::reproduccionAleatoria() {
    if (todasLasCanciones.empty()) {
        cout << "No hay canciones en la
            lista de reproducción." <<
            endl;
        return Cancion(); // Devolver una
            canción por defecto
    }

    static bool seeded = false;
    if (!seeded) {
        srand(time(0));
        seeded = true;
    }

    int indiceAleatorio = rand() %
        todasLasCanciones.size();
    todasLasCanciones[indiceAleatorio].reproducirCancion();
    return
        todasLasCanciones[indiceAleatorio];
}

bool PlayList::actualizarCancion(int id,
    const Cancion& nuevaCancion) {
    for (auto it =
        todasLasCanciones.begin(); it
        != todasLasCanciones.end();
        ++it) {
        if (it->id == id) {
            *it = nuevaCancion; //
                Actualizar la canción
                en el vector
            btree->insert(nuevaCancion);
                // Insertar la nueva
                versión de la canción
                en el B-Tree
            return true;
        }
    }
    return false;
}
```

Listing 6. Código de la Clase Playlist

VII-F1. Introducción: Este archivo implementa las funciones definidas en la cabecera **playlist.h**. Estas funciones gestionan una lista de reproducción de canciones, integrando las funcionalidades de un *Árbol B* (**BTree**) y un vector para optimizar el almacenamiento y las operaciones de búsqueda, ordenación, carga desde un archivo CSV, y reproducción aleatoria.

VII-F2. Funciones Implementadas: Se explican las funciones de la clase **PlayList** en detalle:

- **PlayList(int t)**: Constructor que inicializa el puntero **btree** como un nuevo *Árbol B* de grado **t**.
- **~PlayList()**: Destructor que libera la memoria asignada para el *Árbol B* (**btree**).
- **agregarCancion(Cancion& cancion)**: Agrega una canción al vector **todasLasCanciones** y la inserta en el *Árbol B*.
- **buscarPorNombre(const std::string& nombre, bool searchByArtist)**: Realiza una búsqueda de canciones en el *Árbol B*, filtrando por nombre de la pista o del artista según el valor de **searchByArtist**, y devuelve un vector con los resultados.
- **cargarCSV(const std::string& nombre_archivo)**: Carga canciones desde un archivo CSV. Lee cada línea, extrae los campos utilizando un método que maneja comillas y separadores, y crea objetos **Cancion** que se añaden a la lista de reproducción.
- **imprimirCanciones()**: Recorre el *Árbol B* y muestra las canciones almacenadas, aprovechando su capacidad de ordenamiento.
- **ordenarPorAtributo(const std::string& atributo)**: Ordena las canciones del vector **todasLasCanciones** según un atributo especificado. La comparación se realiza mediante un **lambda**, y el ordenamiento se optimiza usando **std::execution::par** para paralelismo.
- **reproduccionAleatoria()**: Selecciona una canción aleatoria del vector **todasLasCanciones**, asegurando un comportamiento aleatorio con una semilla única. Si la lista está vacía, retorna un objeto **Cancion** por defecto.
- **actualizarCancion(int id, const Cancion& nuevaCancion)**: Busca una canción en el vector por su **id**. Si la encuentra, la actualiza y la reinserta en el *Árbol B* con los nuevos datos.

VII-F3. Aspectos Clave:

- **Uso del *Árbol B***: Optimiza las búsquedas al almacenar las canciones de forma ordenada, permitiendo búsquedas eficientes.
- **Carga desde CSV**: Maneja correctamente campos con comillas o separadores dentro de los datos, asegurando la integridad de la información cargada.
- **Ordenamiento Paralelo**: La función de ordenación aprovecha la biblioteca **std::execution** para realizar operaciones más rápidas en procesadores multinúcleo.

- **Reproducción Aleatoria**: Utiliza una semilla única para garantizar un comportamiento aleatorio consistente en diferentes ejecuciones.
- **Actualización de Canciones**: Garantiza que las modificaciones a una canción se reflejen tanto en el vector como en el *Árbol B*, preservando la sincronización de los datos.

VII-G. trie.h

```
#ifndef TRIE_H
#define TRIE_H

#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>

using namespace std;

struct TrieNode {
    unordered_map<char, TrieNode*>
        ↪ children;
    bool isleaf;

    TrieNode() : isleaf(false) {}
};

class Trie {
public:
    Trie();
    ~Trie();
    bool startsWith(const string& prefix);
    void insert(const string& word);
    void deleteTrie(TrieNode* node);
    vector<string>
        ↪ findWordsWithPrefix(const
        ↪ string& prefix);

private:
    TrieNode* root;
    void findAllWords(TrieNode* node,
        ↪ string currentPrefix,
        ↪ vector<string>& words);
};

#endif
```

Listing 7. Cabecera de la Clase Trie

VII-G1. Librerías incluidas::

- **<iostream>:** Se utiliza para entrada/salida estándar.
- **<unordered_map>:** Se utiliza para implementar un mapa hash eficiente.
- **<string>:** Proporciona soporte para manejar cadenas de texto.
- **<vector>:** Permite almacenar listas dinámicas de palabras.

VII-G2. Estructura TrieNode:

- **children:** Es un mapa que asocia caracteres (**char**) con punteros a nodos hijos.
- **isleaf:** Es un booleano que indica si el nodo representa el final de una palabra.
- **TrieNode():** Constructor que inicializa **isleaf** como **false**.

VII-G3. Clase Trie:

- **Trie():** Constructor que inicializa la raíz del trie.
- **~Trie():** Destructor para liberar memoria asociada con los nodos del trie.
- **bool startsWith(const string& prefix):** Método que verifica si existe alguna palabra en el trie que comience con un prefijo dado.
- **void insert(const string& word):** Inserta una nueva palabra en el trie.
- **void deleteTrie(TrieNode* node):** Libera de forma recursiva la memoria de un nodo y todos sus hijos.
- **vector<string>findWordsWithPrefix(const string& prefix):** Encuentra todas las palabras que comienzan con un prefijo dado.

VII-G4. Miembros privados de la clase Trie::

- **TrieNode* root:** Representa el nodo raíz del trie.
- **void findAllWords(TrieNode* node, string currentPrefix, vector<string>& words):** Función auxiliar que encuentra todas las palabras en el subárbol de un nodo dado, añadiéndolas a un vector.

#endif: Marca el final de la directiva de inclusión condicional, completando la protección contra múltiples inclusiones.

VII-H. trie.cpp

```
#include "Trie.h"

Trie::Trie() {
    root = new TrieNode();  // Crear la
        ↪ raíz del Trie
}

Trie::~Trie() {
    deleteTrie(root);  // Liberar memoria
        ↪ de los nodos del Trie
}

void Trie::deleteTrie(TrieNode* node) {
    for (auto& pair : node->children) {
        deleteTrie(pair.second);
    }
    delete node;
}

void Trie::insert(const string& word) {
    TrieNode* node = root;
    for (char c : word) {
        if (node->children.find(c) ==
            ↪ node->children.end()) {
            node->children[c] = new
                ↪ TrieNode();
        }
        node = node->children[c];
    }
    node->isleaf = true;
}

vector<string>
    ↪ Trie::findWordsWithPrefix(const
    ↪ string& prefix) {
    TrieNode* node = root;
```

```
    vector<string> words;

    for (char c : prefix) {
        if (node->children.find(c) ==
            ↪ node->children.end()) {
            return words;  // Prefijo no
                ↪ encontrado
        }
        node = node->children[c];
    }

    findAllWords(node, prefix, words);
    return words;
}

void Trie::findAllWords(TrieNode* node,
    ↪ string currentPrefix,
    ↪ vector<string>& words) {
    if (node->isleaf) {
        words.push_back(currentPrefix);
    }

    for (auto& pair : node->children) {
        findAllWords(pair.second,
            ↪ currentPrefix + pair.first,
            ↪ words);
    }
}
```

Listing 8. Código de la Clase trie

VII-H1. Constructor y Destructor:

- **Trie::Trie():**
  - Inicializa un objeto de la clase **Trie**, creando la raíz del Trie mediante la instrucción **root = new TrieNode();**.
- **Trie:: Trie():**
  - Libera toda la memoria utilizada por los nodos del Trie mediante la llamada a la función auxiliar **deleteTrie(root);**.

VII-H2. Función auxiliar deleteTrie(TrieNode node):*

- Recorre recursivamente los hijos de un nodo utilizando un bucle **for (auto& pair : node-¿children)**.
- Para cada hijo, se llama a **deleteTrie(pair.second)**.
- Finalmente, libera la memoria del nodo actual con **delete node;**.

VII-H3. Método insert(const string& word):

- Inserta una palabra en el Trie carácter por carácter.
- Utiliza un puntero **node** que inicialmente apunta a la raíz.
- Para cada carácter **c** en la palabra:
  - Si **node-¿children.find(c)** no encuentra el carácter, se crea un nuevo nodo para **c**.
  - El puntero **node** avanza al nodo correspondiente a **c**.
- Al final de la palabra, se marca el nodo actual como hoja mediante **node-¿isleaf = true**.

VII-H4. Método findWordsWithPrefix(const string& prefix):

- Encuentra todas las palabras que comienzan con un prefijo dado.
- Inicializa un puntero **node** apuntando a la raíz y un vector **words** vacío.
- Recorre los caracteres del prefijo:
  - Si algún carácter no existe en **node-¿children**, retorna el vector vacío.
  - Avanza el puntero **node** al hijo correspondiente al carácter actual.
- Llama a **findAllWords(node, prefix, words)** para encontrar todas las palabras en el subárbol.
- Retorna el vector **words**.

VII-H5. Función auxiliar findAllWords(TrieNode node, string currentPrefix, vector¡string¿ & words):*

- Realiza una búsqueda en profundidad para recopilar todas las palabras desde un nodo dado.
- Si el nodo es una hoja (**node-¿isleaf**), agrega el prefijo actual al vector **words**.
- Recorre recursivamente todos los hijos del nodo actual:
  - Llama a **findAllWords(pair.second, currentPrefix + pair.first, words)** para cada hijo, concatenando el carácter actual al prefijo.

VII-I. menu.h

```
#include "playlist.h"
#include <iostream>
#include <chrono>

using namespace std;

class Menu {
public:
    PlayList playlist;
    Menu() : playlist(5) {
        cout << "Inicializando menú y
            ↪ cargando lista de
            ↪ reproducción..." << endl;
    }

    // Destructor
    ~Menu() {
        cout << "Liberando recursos y
            ↪ cerrando el menú..." <<
            ↪ endl;
    }
    void lectura_csv();
    void interfaz_menu();
    void menu_busqueda(int numero_opcion);
    void menu_ordenamiento(int
        ↪ numero_opcion);
    void menu_reproduccion_aleatoria(int
        ↪ numero_opcion);
    void menu_impresion(int numero_opcion);
    void menu_actualizar_cancion(int
        ↪ numero_opcion);
};
```

Listing 9. Cabecera de la Clase Menú

VII-I1. Clase Menu:

- **Propósito:**
  - Gestionar la interacción del usuario con una lista de reproducción a través de diversas opciones de menú.

VII-I2. Atributos de la clase:

- **PlayList playlist:**
  - Es una instancia de la clase **PlayList**.
  - Se inicializa con un tamaño de **5**, tal como se especifica en el constructor de **Menu**.

VII-I3. Constructor Menu():

- Inicializa el objeto **playlist** con un tamaño de **5**.
- Muestra un mensaje en la consola indicando que el menú se está inicializando y que la lista de reproducción está siendo cargada:

```
Inicializando mer
```

VII-I4. Destructor Menu():

- Se ejecuta automáticamente cuando el objeto **Menu** es destruido.
- Libera los recursos utilizados por el menú y muestra el mensaje:

```
Liberando recurso
```

VII-I5. Métodos públicos:

- **void lectura_csv():**
  - Permite cargar datos desde un archivo CSV para poblar la lista de reproducción.
- **void interfaz_menu():**
  - Gestiona la interacción principal con el usuario, mostrando opciones de menú.
- **void menu_busqueda(int numero_opcion):**
  - Permite realizar búsquedas específicas en la lista de reproducción, según la opción seleccionada por el usuario.
- **void menu_ordenamiento(int numero_opcion):**
  - Ofrece opciones para ordenar la lista de reproducción utilizando diferentes criterios.
- **void menu_reproduccion_aleatoria(int numero_opcion):**
  - Permite reproducir canciones de forma aleatoria, basándose en la opción seleccionada.
- **void menu_impresion(int numero_opcion):**
  - Imprime información de la lista de reproducción según los criterios establecidos.
- **void menu_actualizar_cancion(int numero_opcion):**
  - Ofrece opciones para actualizar los datos de una canción específica en la lista de reproducción.

VII-J. menu.cpp

```cpp
#include "menu.h"
#include <iostream>

//menú principal

void Menu::lectura_csv(){
    // ================================
        ↪ LECTURA DEL CSV
        ↪ ================================
    cout << "Leyendo datos desde archivo
        ↪ CSV..." << endl;
    auto inicioLectura =
        ↪ chrono::high_resolution_clock::now();

    playlist.cargarCSV("Pruebas.csv");

    auto finLectura =
        ↪ chrono::high_resolution_clock::now();
    auto duracionLectura =
        ↪ chrono::duration_cast<chrono::seconds>(finLectu
        ↪ - inicioLectura).count();
    cout << "Archivo cargado en " <<          ..
        ↪ duracionLectura << " segundos."
        ↪ << endl;
}

void Menu::interfaz_menu() {
    int numero_opcion = 0;
    Menu::lectura_csv();
    // Bucle infinito para mantener el
        ↪ menú en ejecución hasta que el
        ↪ usuario elija salir
    while (numero_opcion != 5) {
        cout << "\nSeleccione una opción:
            ↪ " << endl;
        cout << "[1] Búsqueda" << endl;
        cout << "[2] Ordenamiento" << endl;
        cout << "[3] Reproducción
            ↪ Aleatoria" << endl;
        cout << "[4] Impresión" << endl;
        cout << "[5] Salir \n>> ";  //
            ↪ Opción para salir
        cin >> numero_opcion;
        cout << "\n";
        // Ejecuta la acción según la
            ↪ opción seleccionada
        if (numero_opcion == 1) {
            menu_busqueda(numero_opcion);
        }
        else if (numero_opcion == 2) {
            menu_ordenamiento(numero_opcion);
        }
        else if (numero_opcion == 3) {
            menu_reproduccion_aleatoria(numero_opcion);
        }
        else if (numero_opcion == 4) {
            playlist.imprimirCanciones();
        }
        else if (numero_opcion == 5) {
            cout << "Saliendo del menú..."
                ↪ << endl;
            break;  // Sale del bucle y
                ↪ termina el programa
        }
        else {
            cout << "Opción no válida." <<
                ↪ endl;
        }
    }
}

void Menu::menu_busqueda(int
    ↪ numero_opcion) {
    cout << "Seleccione un tipo de
        ↪ Búsqueda: " << endl;
```

```
    cout << "[1] Por Nombre de Canción" << endl;
    cout << "[2] Por Nombre de Artista" << endl;
    cout << "[3] Salir" << endl;
    cin >> numero_opcion;

    string nombreBusqueda;
    vector<Cancion> resultados;

    switch (numero_opcion) {
        case 1:
            cout << "Ingrese el prefijo del nombre de la canción: ";
            cin.ignore();
            getline(cin, nombreBusqueda);
            {
                vector<string> canciones = playlist.cancionesTrie.fi
                if (!canciones.empty()) {
                    cout << "Canciones encontradas con el prefijo '" << nombreBusqueda << "':\n";
                    for (size_t i = 0; i < canciones.size(); ++i) {
                        cout << "[" << i + 1 << "] " << canciones[i] << endl;
                    }
                    cout << "Seleccione el número de la canción que desea buscar: ";
                    int seleccion;
                    cin >> seleccion;
                    if (seleccion > 0 && seleccion <= canciones.size()) {
                        nombreBusqueda = canciones[selecci - 1];
                        resultados = playlist.buscarPo false);
                        if (!resultados.empt {
                            for (auto& cancion : resultados) {
                                cancion.imprimir
                            }
                        } else {
                            cout << "No se encontró ninguna canción con el nombre " << nombreBusqued << ".\n";
                        }
                    } else {
                        cout << "Selección no válida.\n";
                    }
                } else {
                    cout << "No se encontró ninguna canción con el prefijo " << nombreBusqueda << ".\n";
                }
            }
            break;
        case 2:
            cout << "Ingrese el prefijo del nombre del artista: ";
            cin.ignore();
            getline(cin, nombreBusqueda);
            {
                vector<string> artistas = playlist.artistasTrie.findWordsWithl
                if (!artistas.empty()) {
                    cout << "Artistas encontrados con el prefijo '" << nombreBusqueda << "':\n";
                    for (size_t i = 0; i < artistas.size(); ++i) {
                        cout << "[" << i + 1 << "] " << artistas[i] << endl;
                    }
                    cout << "Seleccione el número del artista que desea buscar: ";
                    int seleccion;
                    cin >> seleccion;
                    if (seleccion > 0 && seleccion <= artistas.size()) {
                        nombreBusqueda = artistas[seleccion - 1];
                        resultados = playlist.buscarPorNombre(no true);
                        if (!resultados.empty()) {
                            for (auto& cancion : resultados) {
                                cancion.imprimirDatos();
                            }
                        } else {
                            cout << "No se encontró ningún artista con el nombre " << nombreBusqueda << ".\n";
                        }
                    } else {
```

```
                            cout << "Selección no válida.\n";
                        }
                    } else {
                        cout << "No se encontró ningún artista con el prefijo " << nombreBusqueda << ".\n";
                    }
                }
                break;
            case 3:
                break;
            default:
                cout << "Opción no válida." << endl;
                break;
        }
}

void Menu::menu_ordenamiento(int numero_opcion) {
    cout << "Seleccione un tipo de Ordenamiento: " << endl;
    cout << "[1] Por Popularidad" << endl;
    cout << "[2] Por Año" << endl;
    cout << "[3] Por Nombre del Artista" << endl;
    cout << "[4] Por Nombre de la Canción" << endl;
    cout << "[5] Por Género" << endl;
    cout << "[6] Por Duración" << endl;
    cout << "[7] Por Tempo" << endl;
    cout << "[8] Salir" << endl;
    cin >> numero_opcion;

    switch (numero_opcion) {
        case 1:
            playlist.ordenarPorAtributo("popular
            break;
        case 2:
            playlist.ordenarPorAtributo("anio");
            break;
        case 3:
            playlist.ordenarPorAtributo("artista
            break;
        case 4:
            playlist.ordenarPorAtributo("cancion
            break;
        case 5:
            playlist.ordenarPorAtributo("genero"
            break;
        case 6:
            playlist.ordenarPorAtributo("duracio
            break;
        case 7:
            playlist.ordenarPorAtributo("tempo");
            break;
        case 8:
            break;
        default:
            cout << "Opción no válida." << endl;
            break;
    }
}

void Menu::menu_reproduccion_aleatoria(int numero_opcion){
    // ================================ REPRODUCCIÓN ALEATORIA ================================
    cout << "\nReproduciendo canción aleatoria..." << endl;
    playlist.reproduccionAleatoria();
}

void Menu::menu_actualizar_cancion(int numero_opcion) {
    int idActualizacion;
    cout << "Ingrese el ID de la canción a actualizar: ";
    cin >> idActualizacion;

    BTreeNode* nodo = playlist.btree->search(to_string(idActualizacion), false);
    Cancion* cancion = nullptr;

    if (nodo) {
        for (auto& c : nodo->keys) {
            if (c.id == idActualizacion) {
                cancion = &c;
                break;
            }
        }
    }

    if (!cancion) {
        cout << "No se encontró una canción con el ID " << idActualizacion << " para actualizar.\n";
        return;
    }

    int opcion;
    do {
        cout << "\nSeleccione el atributo a modificar: " << endl;
        cout << "[1] Nombre del Artista" << endl;
        cout << "[2] Nombre de la Canción" << endl;
        cout << "[3] ID del Track" << endl;
        cout << "[4] Popularidad" << endl;
        cout << "[5] Año" << endl;
        cout << "[6] Género" << endl;
        cout << "[7] Danceability" << endl;
        cout << "[8] Energy" << endl;
        cout << "[9] Key" << endl;
        cout << "[10] Loudness" << endl;
        cout << "[11] Mode" << endl;
        cout << "[12] Speechiness" << endl;
        cout << "[13] Acousticness" << endl;
        cout << "[14] Instrumentalness" << endl;
        cout << "[15] Liveness" << endl;
        cout << "[16] Valence" << endl;
        cout << "[17] Tempo" << endl;
        cout << "[18] Duración en ms" << endl;
        cout << "[19] Time Signature" << endl;
        cout << "[20] Salir" << endl;
        cout << ">> ";
        cin >> opcion;

        switch (opcion) {
            case 1:
                cout << "Ingrese el nuevo nombre del artista: ";
                cin.ignore();
                getline(cin, cancion->artist_name);
                break;
            case 2:
                cout << "Ingrese el nuevo nombre de la
```

```
                    ↪ canción: ";
241                 cin.ignore();
242                 getline(cin,
                        ↪ cancion->track_name);
243                 break;
244             case 3:
245                 cout << "Ingrese el nuevo
                        ↪ ID del track: ";
246                 cin.ignore();
247                 getline(cin,
                        ↪ cancion->track_id);
248                 break;
249             case 4:
250                 cout << "Ingrese la nueva
                        ↪ popularidad: ";
251                 cin >> cancion->popularity;
252                 break;
253             case 5:
254                 cout << "Ingrese el nuevo
                        ↪ año: ";
255                 cin >> cancion->year;
256                 break;
257             case 6:
258                 cout << "Ingrese el nuevo
                        ↪ género: ";
259                 cin.ignore();
260                 getline(cin,
                        ↪ cancion->genre);
261                 break;
262             case 7:
263                 cout << "Ingrese la nueva
                        ↪ danceability: ";
264                 cin >>
                        ↪ cancion->danceability;
265                 break;
266             case 8:
267                 cout << "Ingrese la nueva
                        ↪ energy: ";
268                 cin >> cancion->energy;
269                 break;
270             case 9:
271                 cout << "Ingrese el nuevo
                        ↪ key: ";
272                 cin >> cancion->key;
273                 break;
274             case 10:
275                 cout << "Ingrese el nuevo
                        ↪ loudness: ";
276                 cin >> cancion->loudness;
277                 break;
278             case 11:
279                 cout << "Ingrese el nuevo
                        ↪ mode: ";
280                 cin >> cancion->mode;
281                 break;
282             case 12:
283                 cout << "Ingrese la nueva
                        ↪ speechiness: ";
284                 cin >>
                        ↪ cancion->speechiness;
285                 break;
286             case 13:
287                 cout << "Ingrese la nueva
                        ↪ acousticness: ";
288                 cin >>
                        ↪ cancion->acousticness;
289                 break;
290             case 14:
291                 cout << "Ingrese la nueva
                        ↪ instrumentalness: ";
292                 cin >>
                        ↪ cancion->instrumentalness;
293                 break;
294             case 15:
295                 cout << "Ingrese la nueva
                        ↪ liveness: ";
296                 cin >> cancion->liveness;
297                 break;
298             case 16:
299                 cout << "Ingrese la nueva
                        ↪ valence: ";
300                 cin >> cancion->valence;
301                 break;
302             case 17:
303                 cout << "Ingrese el nuevo
                        ↪ tempo: ";
304                 cin >> cancion->tempo;
305                 break;
306             case 18:
307                 cout << "Ingrese la nueva
                        ↪ duración en ms: ";
308                 cin >>
                        ↪ cancion->duration_ms;
309                 break;
310             case 19:
311                 cout << "Ingrese el nuevo
                        ↪ time signature: ";
312                 cin >>
                        ↪ cancion->time_signature;
313                 break;
314             case 20:
315                 cout << "Saliendo de la
                        ↪ actualización de
                        ↪ canción..." << endl;
316                 break;
317             default:
318                 cout << "Opción no
                        ↪ válida." << endl;
319                 break;
320         }
321     } while (opcion != 20);
322
323     cout << "Canción actualizada:\n";
324     cancion->imprimirDatos();
325 }
```

Listing 10. Cabecera de la Clase Menú

VII-J1. Clase Menu:

- **Propósito:**
  - Gestionar las interacciones del usuario mediante un menú principal, ofreciendo funcionalidades relacionadas con una lista de reproducción musical.

VII-J2. Métodos de la clase:

- **void lectura_csv()**
  - **Propósito:** Cargar datos desde un archivo `CSV` en la lista de reproducción.
  - **Descripción:**
    - Muestra un mensaje indicando que la lectura del archivo está en proceso.
    - Registra el tiempo de inicio y finalización de la operación.
    - Utiliza el método `cargarCSV` del objeto `playlist` para procesar el archivo `"Pruebas.csv"`.
    - Calcula y muestra la duración total de la carga en segundos.
- **void interfaz_menu()**
  - **Propósito:** Proveer una interfaz interactiva para el usuario.
  - **Descripción:**
    - Llama al método `lectura_csv()` para cargar datos al inicio.

- Ofrece un menú con las siguientes opciones:
  1. Búsqueda.
  2. Ordenamiento.
  3. Reproducción aleatoria.
  4. Impresión de la lista de reproducción.
  5. Salir.
- Ejecuta el método correspondiente según la opción seleccionada.
- Valida entradas no válidas y muestra un mensaje de error en caso necesario.

- **void menu_busqueda(int numero_opcion)**
  - **Propósito:** Buscar canciones o artistas en la lista de reproducción.
  - **Descripción:**
    - Permite elegir entre búsqueda por:
      - Nombre de la canción.
      - Nombre del artista.
    - Solicita un prefijo para realizar la búsqueda.
    - Muestra los resultados encontrados usando los `Trie` de canciones o artistas.
    - Permite seleccionar un resultado específico para obtener más detalles.
- **void menu_ordenamiento(int numero_opcion)**
  - **Propósito:** Ordenar la lista de reproducción por diferentes atributos.
  - **Descripción:**
    - Ofrece una lista de atributos por los cuales se puede ordenar, como: `popularidad`, `año`, `género`, `duración`, entre otros.
    - Llama al método `ordenarPorAtributo` del objeto `playlist` con el atributo seleccionado.
- **void menu_reproduccion_aleatoria(int numero_opcion)**
  - **Propósito:** Reproducir canciones de manera aleatoria.
  - **Descripción:**
    - Llama al método `reproduccionAleatoria` del objeto `playlist`.
    - Muestra un mensaje indicando que una canción está siendo reproducida aleatoriamente.
- **void menu_actualizar_cancion(int numero_opcion)**
  - **Propósito:** Permitir al usuario actualizar los datos de una canción específica.
  - **Descripción:**
    - Solicita al usuario el `ID` de la canción a actualizar.
    - Utiliza el `BTree` de `playlist` para buscar la canción correspondiente.
    - Ofrece un submenú con una lista de atributos editables:
      - Ejemplo de atributos: `nombre del artista`, `popularidad`, `género`, `duración en ms`, entre otros.
    - Actualiza el atributo seleccionado con el nuevo valor proporcionado por el usuario.
    - Imprime los datos de la canción actualizada al finalizar.

VII-K. main.cpp

```cpp
#include "menu.h"
#include <iostream>
using namespace std;

int main() {
    Menu menu;
    menu.interfaz_menu();
    return 0;
}
```

Listing 11. Código del main

Este archivo contiene la función **main**, el punto de entrada de la aplicación. Esencialmente, su propósito es inicializar el menú principal e invocar la interfaz interactiva para que el usuario pueda gestionar la lista de reproducción de canciones.

VII-K1. Descripción de la Función `main`:

- **Inicialización del Objeto Menu**:
  - Se crea un objeto de la clase **Menu**, definido en el archivo **menu.h**.
  - Este objeto representa la lógica de interacción con el usuario y contiene las funcionalidades necesarias para navegar por el programa.
- **Llamada a `interfaz_menu()`**:
  - La función `interfaz_menu()` es invocada sobre el objeto **menu**.
  - Su objetivo es presentar las opciones del programa al usuario y gestionar las entradas para realizar acciones como agregar canciones, buscar, ordenar, o reproducir aleatoriamente.
- **Terminación del Programa**:
  - La función retorna 0 al sistema operativo, indicando que la ejecución del programa se realizó exitosamente.

VII-K2. Aspectos Clave:

- **Modularidad**: La clase **Menu** encapsula la lógica de interacción con el usuario, separando las responsabilidades de la función **main**, lo que mejora la claridad y mantenibilidad del código.
- **Punto de Entrada Único**: El uso de una función **main** simple permite que el programa sea fácil de comprender y extender en el futuro.

VIII. Qt Creator: Entorno ideal para nuestro Frontend

Qt Creator es una herramienta destacada para el desarrollo de interfaces de usuario debido a su combinación de características, facilidad de uso y flexibilidad. Proporciona un entorno de desarrollo integrado (IDE) diseñado específicamente para trabajar con Qt, un marco de trabajo ampliamente utilizado para crear aplicaciones gráficas multiplataforma. Uno de los aspectos más valiosos de Qt Creator es su diseñador visual, que permite a los desarrolladores crear interfaces gráficas de manera intuitiva mediante un sistema de arrastrar y soltar, lo que acelera significativamente el proceso de diseño. Además, las interfaces creadas se integran directamente con el código subyacente, lo que simplifica la conexión entre la lógica de la aplicación y su presentación gráfica.

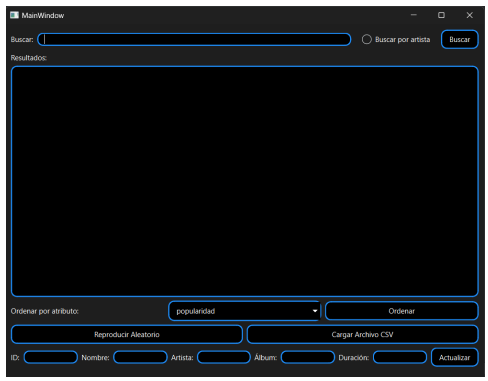
Otra ventaja importante es que Qt Creator soporta múltiples lenguajes de programación, siendo C++ el principal, pero también permite el uso de QML, un lenguaje declarativo diseñado específicamente para interfaces de usuario modernas y fluidas. Esto da a los desarrolladores la capacidad de trabajar con herramientas avanzadas para crear aplicaciones visualmente atractivas y altamente responsivas.

La herramienta también se destaca por su capacidad multiplataforma, lo que permite diseñar una interfaz en un sistema operativo y ejecutarla sin cambios en otros como Windows, macOS, Linux, e incluso en dispositivos móviles con Android o iOS. Esto hace que Qt Creator sea ideal para proyectos que necesitan ser distribuidos en múltiples entornos.

Otra razón por la que es considerada una buena herramienta es su documentación extensa y su comunidad activa. Esto facilita encontrar soluciones a problemas comunes y aprender a utilizar sus funcionalidades de manera eficiente. Además, Qt Creator incluye características como autocompletado de código, integración con control de versiones, depuración avanzada y herramientas de análisis de rendimiento, que contribuyen a un flujo de trabajo más rápido y eficiente.

Por último, su capacidad para manejar proyectos complejos y su compatibilidad con estándares modernos lo convierten en una elección sólida tanto para desarrolladores individuales como para equipos de desarrollo que buscan crear aplicaciones profesionales con interfaces de usuario de alta calidad.

VIII-A. Home de la Interfas de Usuario



La interfaz corresponde a una aplicación con un diseño oscuro y bordes resaltados en azul. En la parte superior, se encuentra un cuadro de texto acompañado por un botón de búsqueda, etiquetado como ”Buscar”. Adicionalmente, hay una opción seleccionable identificada como ”Buscar por artista”.

En el centro de la interfaz, se presenta un área rectangular grande denominada Resultados”, destinada a mostrar información o listas de elementos. Este espacio está delimitado por un contorno azul.

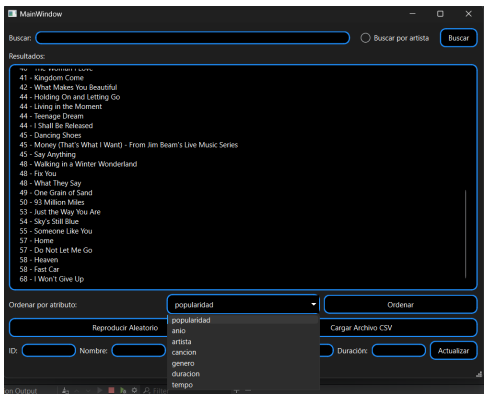
Debajo de esta sección principal, hay un menú desplegable titulado .Ordenar por atributo”, que parece estar configurado por defecto en ”popularidad”. A la derecha de este, se encuentra un botón para ejecutar la acción de ordenar.

Más abajo, hay varios botones y campos de texto. A la izquierda se encuentra el botón Reproducir Aleatorio”, seguido de Çargar Archivo CSV”. Junto a estos, se localizan etiquetas y campos vacíos identificados como ÏD”, ”Nombre”, .Artista”, ”Álbumz ”Duración”. A la derecha de estos campos hay un botón llamado .Actualizar”.

El diseño es minimalista y parece orientado a una funcionalidad de gestión de datos o reproducción de música, basado en las etiquetas visibles.

En esta nueva versión de la interfaz, se observa que en la sección Resultados.ªhora se muestran datos concretos. Estos datos consisten en una lista numerada de canciones con títulos y, en algunos casos, información adicional. La lista parece estar desplegada completamente, y su contenido sugiere un sistema de gestión o reproducción de música.

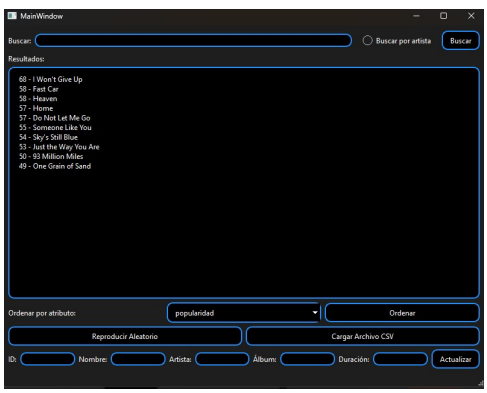
VIII-B. Ordenar de la Interfas de Usuario



En el menú desplegable .ºrdenar por atributo", se nota una ampliación de las opciones disponibles. Ahora, además de "popularidad", se incluyen los siguientes atributos: .ªño", .ªrtista", çanción", "género", "duraciónz "tempo". Esto indica que la funcionalidad de ordenamiento es más versátil y permite organizar la lista de resultados en función de varios criterios.

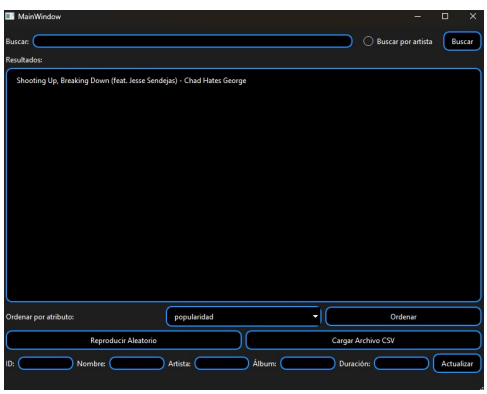
La interacción con los resultados y el menú desplegable sugiere una experiencia más dinámica y adaptable a las necesidades del usuario.

VIII-C. Limitación a diez superiores



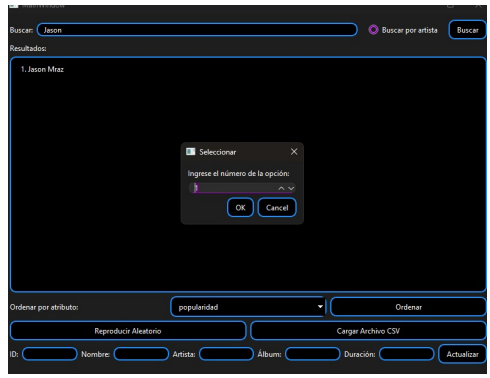
Aquí solo se muestran los diez primeros con más coincidencia.

VIII-D. Reproducción Aleatoria



Aquí tenemos la reproducción aleatoria.

VIII-E. Búsqueda Óptima



Aquí tenemos la búsqueda exitosa por completación de palabras.