“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

1 Introducción **4**

2 ¿Por qué utilizar Linked List? **5**

3 Hablemos de los BTree **5**

4 Explicación del Código para el Backend **6**

4.1 btree.h 6

4.1.1 Directivas de Preprocesador 7

4.1.2 Clase BTreeNode 7

4.1.3 Clase BTree 8

4.1.4 Características del B-Tree 8

4.2 btree.cpp 8

4.2.1 Desarrollando la Clase BTree 12

4.2.2 Desarrollando la Clase BTreeNode 13

4.2.3 Función splitChild 14

4.3 cancion.h 14

4.3.1 Introducción 15

4.3.2 Miembros de la clase 15

4.3.3 Constructores 16

4.3.4 Funciones 17

4.3.5 Conclusión 17

4.4 cancion.cpp 17

4.4.1 Introducción 19

4.4.2 Miembros de la clase 19

4.4.3 Constructores 20

4.4.4 Funciones 20

4.5 playlist.h 20

4.5.1 Introducción 21

4.5.2 Atributos 22

4.5.3 Constructor y Destructor 22

4.5.4 Funciones 22

4.5.5 Conclusión 23
4.6 playlist.cpp 23
4.7 menu.h 28
4.8 menu.cpp 28
4.9 main.cpp 36

5 Qt Creator: Entorno ideal para nuestro Frontend **36**

1. 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.

2. ¿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).

3. 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.

Esto hace que los B-trees sean ideales para bases de datos y sistemas de archivos, donde se busca optimizar el tiempo de acceso. En contraste con las listas enlazadas, que son lineales, los B-trees son jerárquicos y mantienen las claves ordenadas dentro de los nodos, facilitando tanto la búsqueda binaria como las operaciones de división o combinación necesarias para mantener el balance. Además, los B-trees son especialmente útiles para datos en disco, ya que cada nodo está diseñado para ajustarse al tamaño de un bloque de memoria o sector de disco, reduciendo las operaciones de entrada y salida. La capacidad de contener múltiples claves en un nodo también mejora la eficiencia del almacenamiento en comparación con las listas enlazadas, que requieren espacio adicional para punteros por cada elemento. Aunque las listas enlazadas son simples y rápidas para insertar y eliminar elementos en posiciones arbitrarias, su rendimiento general se degrada cuando los datos crecen significativamente o cuando las búsquedas son frecuentes. Por estas razones, los B-trees son una elección superior para sistemas donde la eficiencia en el manejo de grandes conjuntos de datos es crucial.

4. Explicación del Código para el Backend

4.1. btree.h

```
#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);
```

```
36 };
37
38 #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.

4.1.1. 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**.

4.1.2. 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**.

4.1.3. 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**.

4.1.4. 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.

4.2. btree.cpp

```
1 #include "btree.h"
2 #include <iostream>
3
4 BTree::BTree(int t) : t(t), root(nullptr) {}
```

```
5
6  BTreeNode::BTreeNode(int t, bool isLeaf) : t(t), isLeaf(isLeaf) {
7      keys.reserve(2 * t - 1);
8      children.reserve(2 * t);
9  }
10
11 void BTreeNode::traverse() {
12     int i;
13     for (i = 0; i < keys.size(); i++) {
14         if (!isLeaf) {
15             children[i]->traverse();
16         }
17         keys[i].imprimirDatos();
18     }
19     if (!isLeaf) {
20         children[i]->traverse();
21     }
22 }
23
24 BTreeNode* BTreeNode::search(const std::string& key, bool
       searchByArtist) {
25     int i = 0;
26     while (i < keys.size() && (searchByArtist ? keys[i].artist_name :
           keys[i].track_name) < key) {
27         i++;
28     }
29
30     if (i < keys.size() && (searchByArtist ? keys[i].artist_name : keys
           [i].track_name) == key) {
31         return this;
32     }
33
34     if (isLeaf) {
35         return nullptr;
36     }
37
38     return children[i]->search(key, searchByArtist);
39 }
40
41 void BTreeNode::searchAll(const std::string& key, bool searchByArtist,
       std::vector<Cancion>& result) {
```

```
42      int i = 0;
43      while (i < keys.size() && (searchByArtist ? keys[i].artist_name :
            keys[i].track_name) < key) {
44          i++;
45      }
46
47      if (i < keys.size() && (searchByArtist ? keys[i].artist_name : keys
            [i].track_name) == key) {
48          result.push_back(keys[i]);
49      }
50
51      if (isLeaf) {
52          return;
53      }
54
55      for (int j = 0; j <= keys.size(); j++) {
56          children[j]->searchAll(key, searchByArtist, result);
57      }
58  }
59
60  void BTree::traverse() {
61      if (root != nullptr) {
62          root->traverse();
63      }
64  }
65
66  BTreeNode* BTree::search(const std::string& key, bool searchByArtist) {
67      return (root == nullptr) ? nullptr : root->search(key,
            searchByArtist);
68  }
69
70  void BTree::searchAll(const std::string& key, bool searchByArtist, std
        ::vector<Cancion>& result) {
71      if (root != nullptr) {
72          root->searchAll(key, searchByArtist, result);
73      }
74  }
75
76  void BTree::insert(Cancion k) {
77      if (root == nullptr) {
78          root = new BTreeNode(t, true);
```

```
79          root->keys.push_back(k);
80      } else {
81          if (root->keys.size() == 2 * t - 1) {
82              BTreeNode* s = new BTreeNode(t, false);
83              s->children.push_back(root);
84              s->splitChild(0, root);
85
86              int i = 0;
87              if ((s->keys[0].artist_name < k.artist_name) || (s->keys
                    [0].track_name < k.track_name)) {
88                  i++;
89              }
90              s->children[i]->insertNonFull(k);
91
92              root = s;
93          } else {
94              root->insertNonFull(k);
95          }
96      }
97  }
98
99  void BTreeNode::insertNonFull(Cancion k) {
100     int i = keys.size() - 1;
101
102     if (isLeaf) {
103         keys.push_back(k);
104         while (i >= 0 && (keys[i].artist_name > k.artist_name || keys[i
                ].track_name > k.track_name)) {
105             keys[i + 1] = keys[i];
106             i--;
107         }
108         keys[i + 1] = k;
109     } else {
110         while (i >= 0 && (keys[i].artist_name > k.artist_name || keys[i
                ].track_name > k.track_name)) {
111             i--;
112         }
113         if (children[i + 1]->keys.size() == 2 * t - 1) {
114             splitChild(i + 1, children[i + 1]);
115             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.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

4.3. 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;
```

```
32
33      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) {}
34
35      Cancion(int id, string artist_name, string track_name, string
            track_id, int popularity, int year,
36                      string genre, double danceability, double energy, int
                            key, double loudness, int mode,
37                      double speechiness, double acousticness, double
                            instrumentalness, double liveness,
38                      double valence, double tempo, int duration_ms, int
                            time_signature);
39
40      void imprimirDatos();
41      void reproducirCancion();
42  };
43
44  #endif // CANCION_H
```

Listing 3: Cabecera de la Clase Canción

4.3.1. 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**.

4.3.2. 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).

4.3.3. 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.

4.3.4. 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.

4.3.5. 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.

4.4. cancion.cpp

```
1 #include "cancion.h"
2
3 Cancion::Cancion(int id, string artist_name, string track_name, string
     track_id, int popularity, int year,
4                 string genre, double danceability, double energy, int
                      key, double loudness, int mode,
5                 double speechiness, double acousticness, double
                      instrumentalness, double liveness,
6                 double valence, double tempo, int duration_ms, int
                      time_signature)
```

```
7       : id(id), artist_name(artist_name), track_name(track_name),
            track_id(track_id), popularity(popularity),
8         year(year), genre(genre), danceability(danceability), energy(
              energy), key(key), loudness(loudness),
9         mode(mode), speechiness(speechiness), acousticness(acousticness),
              instrumentalness(instrumentalness),
10        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

4.4.1. 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**.

4.4.2. 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).

4.4.3. 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.

4.4.4. 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.

4.5. 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

4.5.1. 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.

4.5.2. 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.

4.5.3. 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**.

4.5.4. 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**.

4.5.5. Conclusión

La clase **PlayList** proporciona una implementación eficiente para manejar listas de canciones, integrando la funcionalidad de búsqueda en un *Árbol B* con la posibilidad de gestionar canciones almacenadas en un vector. Además, facilita la carga de datos desde archivos CSV y la manipulación de las canciones mediante funciones de ordenación, actualización y reproducción aleatoria.

4.6. playlist.cpp

```
1  // playlist.cpp
2  #include "playlist.h"
3  #include <iostream>
4  #include <algorithm>
5  #include <execution>
6
7  PlayList::PlayList(int t) {
8      btree = new BTree(t);
9  }
10
11 PlayList::~PlayList() {
12     delete btree;
13 }
14
15 void PlayList::agregarCancion(Cancion& cancion) {
16     btree->insert(cancion);
17     todasLasCanciones.push_back(cancion); // Añadir la canción al
           vector
18 }
```

```
19
20 vector<Cancion> PlayList::buscarPorNombre(const std::string& nombre,
       bool searchByArtist) {
21     vector<Cancion> resultados;
22     btree->searchAll(nombre, searchByArtist, resultados);
23     return resultados;
24 }
25
26 void PlayList::cargarCSV(const std::string& nombre_archivo) {
27     ifstream archivo(nombre_archivo);
28     if (!archivo.is_open()) {
29         cerr << "No se pudo abrir el archivo " << nombre_archivo <<
               endl;
30         return;
31     }
32
33     string linea;
34     getline(archivo, linea);  // Leer la cabecera del CSV
35     while (getline(archivo, linea)) {
36         stringstream ss(linea);
37         string token;
38
39         auto leerCampo = [&ss]() {
40             string campo;
41             char ch;
42             bool dentroComillas = false;
43
44             while (ss.get(ch)) {
45                 if (ch == '"' && !dentroComillas) {
46                     dentroComillas = true;
47                 } else if (ch == '"' && dentroComillas) {
48                     if (ss.peek() == ',') {
49                         ss.get();
50                         break;
51                     }
52                     dentroComillas = false;
53                 } else if (ch == ',' && !dentroComillas) {
54                     break;
55                 } else {
56                     campo += ch;
57                 }
```

```
            }
            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) {
```

```
96          if (atributo == "popularidad") {
97              return a.popularity < b.popularity;
98          } else if (atributo == "anio") {
99              return a.year < b.year;
100         } else if (atributo == "artista") {
101             return a.artist_name < b.artist_name;
102         } else if (atributo == "cancion") {
103             return a.track_name < b.track_name;
104         } else if (atributo == "genero") {
105             return a.genre < b.genre;
106         } else if (atributo == "duracion") {
107             return a.duration_ms < b.duration_ms;
108         } else if (atributo == "tempo") {
109             return a.tempo < b.tempo;
110         }
111         return false;
112     };
113
114     sort(std::execution::par,todasLasCanciones.begin(),
        todasLasCanciones.end(), comparar);
115
116     cout << "Canciones después de ordenar por " << atributo << ":" <<
        endl;
117     for (const auto& cancion : todasLasCanciones) {
118         if (atributo == "popularidad") {
119             cout << cancion.popularity << " - " << cancion.track_name
                << endl;
120         } else if (atributo == "anio") {
121             cout << cancion.year << " - " << cancion.track_name << endl
                ;
122         } else if (atributo == "artista") {
123             cout << cancion.artist_name << " - " << cancion.track_name
                << endl;
124         } else if (atributo == "cancion") {
125             cout << cancion.track_name << endl;
126         } else if (atributo == "genero") {
127             cout << cancion.genre << " - " << cancion.track_name <<
                endl;
128         } else if (atributo == "duracion") {
129             cout << cancion.duration_ms << " ms - " << cancion.
                track_name << endl;
```

```
130         } else if (atributo == "tempo") {
131             cout << cancion.tempo << " - " << cancion.track_name <<
                endl;
132         }
133     }
134 }
135
136 Cancion PlayList::reproduccionAleatoria() {
137     if (todasLasCanciones.empty()) {
138         cout << "No hay canciones en la lista de reproducción." << endl
            ;
139         return Cancion(); // Devolver una canción por defecto
140     }
141
142     static bool seeded = false;
143     if (!seeded) {
144         srand(time(0));
145         seeded = true;
146     }
147
148     int indiceAleatorio = rand() % todasLasCanciones.size();
149     todasLasCanciones[indiceAleatorio].reproducirCancion();
150     return todasLasCanciones[indiceAleatorio];
151 }
152
153 bool PlayList::actualizarCancion(int id, const Cancion& nuevaCancion) {
154     for (auto it = todasLasCanciones.begin(); it != todasLasCanciones.
        end(); ++it) {
155         if (it->id == id) {
156             *it = nuevaCancion; // Actualizar la canción en el vector
157             btree->insert(nuevaCancion); // Insertar la nueva versión
                de la canción en el B-Tree
158             return true;
159         }
160     }
161     return false;
162 }
```

Listing 6: Código de la Clase Playlist

4.7. 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 7: Cabecera de la Clae Menú

4.8. menu.cpp

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

//menú principal

void Menu::lectura_csv(){
    // ================================ LECTURA DEL CSV
       ================================
```

```
8       cout << "Leyendo datos desde archivo CSV..." << endl;
9       auto inicioLectura = chrono::high_resolution_clock::now();
10
11      playlist.cargarCSV("spotify_data.csv");
12
13      auto finLectura = chrono::high_resolution_clock::now();
14      auto duracionLectura = chrono::duration_cast<chrono::seconds>(
            finLectura - inicioLectura).count();
15      cout << "Archivo cargado en " << duracionLectura << " segundos." <<
            endl;
16  }
17
18  void Menu::interfaz_menu() {
19      int numero_opcion = 0;
20      Menu::lectura_csv();
21      // Bucle infinito para mantener el menú en ejecución hasta que el
            usuario elija salir
22      while (numero_opcion != 5) {
23          cout << "\nSeleccione una opción: " << endl;
24          cout << "[1] Búsqueda" << endl;
25          cout << "[2] Ordenamiento" << endl;
26          cout << "[3] Reproducción Aleatoria" << endl;
27          cout << "[4] Impresión" << endl;
28          cout << "[5] Salir \n>> ";  // Opción para salir
29          cin >> numero_opcion;
30          cout << "\n";
31          // Ejecuta la acción según la opción seleccionada
32          if (numero_opcion == 1) {
33              menu_busqueda(numero_opcion);
34          }
35          else if (numero_opcion == 2) {
36              menu_ordenamiento(numero_opcion);
37          }
38          else if (numero_opcion == 3) {
39              menu_reproduccion_aleatoria(numero_opcion);
40          }
41          else if (numero_opcion == 4) {
42              playlist.imprimirCanciones();
43          }
44          else if (numero_opcion == 5) {
45              cout << "Saliendo del menú..." << endl;
```

```
46              break;  // Sale del bucle y termina el programa
47          }
48          else {
49              cout << "Opción no válida." << endl;
50          }
51      }
52  }
53
54  void Menu::menu_busqueda(int numero_opcion) {
55      cout << "Seleccione un tipo de Búsqueda: " << endl;
56      cout << "[1] Por Nombre de Canción" << endl;
57      cout << "[2] Por Nombre de Artista" << endl;
58      cout << "[3] Salir" << endl;
59      cin >> numero_opcion;
60
61      string nombreBusqueda;
62      vector<Cancion> resultados;
63
64      switch (numero_opcion) {
65          case 1:
66              cout << "Ingrese el nombre de la canción: ";
67              cin.ignore();
68              getline(cin, nombreBusqueda);
69              resultados = playlist.buscarPorNombre(nombreBusqueda, false
                    );
70              if (!resultados.empty()) {
71                  for (auto& cancion : resultados) {
72                      cancion.imprimirDatos();
73                  }
74              } else {
75                  cout << "No se encontró ninguna canción con el nombre "
                        << nombreBusqueda << ".\n";
76              }
77              break;
78          case 2:
79              cout << "Ingrese el nombre del artista: ";
80              cin.ignore();
81              getline(cin, nombreBusqueda);
82              resultados = playlist.buscarPorNombre(nombreBusqueda, true)
                    ;
83              if (!resultados.empty()) {
```

```
84                  for (auto& cancion : resultados) {
85                      cancion.imprimirDatos();
86                  }
87              } else {
88                  cout << "No se encontró ningún artista con el nombre "
                        << nombreBusqueda << ".\n";
89              }
90              break;
91          case 3:
92              break;
93          default:
94              cout << "Opción no válida." << endl;
95              break;
96      }
97  }
98
99  void Menu::menu_ordenamiento(int numero_opcion) {
100     cout << "Seleccione un tipo de Ordenamiento: " << endl;
101     cout << "[1] Por Popularidad" << endl;
102     cout << "[2] Por Año" << endl;
103     cout << "[3] Por Nombre del Artista" << endl;
104     cout << "[4] Por Nombre de la Canción" << endl;
105     cout << "[5] Por Género" << endl;
106     cout << "[6] Por Duración" << endl;
107     cout << "[7] Por Tempo" << endl;
108     cout << "[8] Salir" << endl;
109     cin >> numero_opcion;
110
111     switch (numero_opcion) {
112         case 1:
113             playlist.ordenarPorAtributo("popularidad");
114             break;
115         case 2:
116             playlist.ordenarPorAtributo("anio");
117             break;
118         case 3:
119             playlist.ordenarPorAtributo("artista");
120             break;
121         case 4:
122             playlist.ordenarPorAtributo("cancion");
123             break;
```

```
124             case 5:
125                 playlist.ordenarPorAtributo("genero");
126                 break;
127             case 6:
128                 playlist.ordenarPorAtributo("duracion");
129                 break;
130             case 7:
131                 playlist.ordenarPorAtributo("tempo");
132                 break;
133             case 8:
134                 break;
135             default:
136                 cout << "Opción no válida." << endl;
137                 break;
138         }
139 }
140 
141 void Menu::menu_reproduccion_aleatoria(int numero_opcion){
142     // ============================== REPRODUCCIÓN ALEATORIA
        ==============================
143     cout << "\nReproduciendo canción aleatoria..." << endl;
144     playlist.reproduccionAleatoria();
145 }
146 
147 void Menu::menu_actualizar_cancion(int numero_opcion) {
148     int idActualizacion;
149     cout << "Ingrese el ID de la canción a actualizar: ";
150     cin >> idActualizacion;
151 
152     BTreeNode* nodo = playlist.btree->search(to_string(idActualizacion)
        , false);
153     Cancion* cancion = nullptr;
154 
155     if (nodo) {
156         for (auto& c : nodo->keys) {
157             if (c.id == idActualizacion) {
158                 cancion = &c;
159                 break;
160             }
161         }
162     }
```

```
163
164     if (!cancion) {
165         cout << "No se encontró una canción con el ID " <<
                idActualizacion << " para actualizar.\n";
166         return;
167     }
168
169     int opcion;
170     do {
171         cout << "\nSeleccione el atributo a modificar: " << endl;
172         cout << "[1] Nombre del Artista" << endl;
173         cout << "[2] Nombre de la Canción" << endl;
174         cout << "[3] ID del Track" << endl;
175         cout << "[4] Popularidad" << endl;
176         cout << "[5] Año" << endl;
177         cout << "[6] Género" << endl;
178         cout << "[7] Danceability" << endl;
179         cout << "[8] Energy" << endl;
180         cout << "[9] Key" << endl;
181         cout << "[10] Loudness" << endl;
182         cout << "[11] Mode" << endl;
183         cout << "[12] Speechiness" << endl;
184         cout << "[13] Acousticness" << endl;
185         cout << "[14] Instrumentalness" << endl;
186         cout << "[15] Liveness" << endl;
187         cout << "[16] Valence" << endl;
188         cout << "[17] Tempo" << endl;
189         cout << "[18] Duración en ms" << endl;
190         cout << "[19] Time Signature" << endl;
191         cout << "[20] Salir" << endl;
192         cout << ">> ";
193         cin >> opcion;
194
195         switch (opcion) {
196             case 1:
197                 cout << "Ingrese el nuevo nombre del artista: ";
198                 cin.ignore();
199                 getline(cin, cancion->artist_name);
200                 break;
201             case 2:
202                 cout << "Ingrese el nuevo nombre de la canción: ";
```

```
203                 cin.ignore();
204                 getline(cin, cancion->track_name);
205                 break;
206             case 3:
207                 cout << "Ingrese el nuevo ID del track: ";
208                 cin.ignore();
209                 getline(cin, cancion->track_id);
210                 break;
211             case 4:
212                 cout << "Ingrese la nueva popularidad: ";
213                 cin >> cancion->popularity;
214                 break;
215             case 5:
216                 cout << "Ingrese el nuevo año: ";
217                 cin >> cancion->year;
218                 break;
219             case 6:
220                 cout << "Ingrese el nuevo género: ";
221                 cin.ignore();
222                 getline(cin, cancion->genre);
223                 break;
224             case 7:
225                 cout << "Ingrese la nueva danceability: ";
226                 cin >> cancion->danceability;
227                 break;
228             case 8:
229                 cout << "Ingrese la nueva energy: ";
230                 cin >> cancion->energy;
231                 break;
232             case 9:
233                 cout << "Ingrese el nuevo key: ";
234                 cin >> cancion->key;
235                 break;
236             case 10:
237                 cout << "Ingrese el nuevo loudness: ";
238                 cin >> cancion->loudness;
239                 break;
240             case 11:
241                 cout << "Ingrese el nuevo mode: ";
242                 cin >> cancion->mode;
243                 break;
```

```
244             case 12:
245                 cout << "Ingrese la nueva speechiness: ";
246                 cin >> cancion->speechiness;
247                 break;
248             case 13:
249                 cout << "Ingrese la nueva acousticness: ";
250                 cin >> cancion->acousticness;
251                 break;
252             case 14:
253                 cout << "Ingrese la nueva instrumentalness: ";
254                 cin >> cancion->instrumentalness;
255                 break;
256             case 15:
257                 cout << "Ingrese la nueva liveness: ";
258                 cin >> cancion->liveness;
259                 break;
260             case 16:
261                 cout << "Ingrese la nueva valence: ";
262                 cin >> cancion->valence;
263                 break;
264             case 17:
265                 cout << "Ingrese el nuevo tempo: ";
266                 cin >> cancion->tempo;
267                 break;
268             case 18:
269                 cout << "Ingrese la nueva duración en ms: ";
270                 cin >> cancion->duration_ms;
271                 break;
272             case 19:
273                 cout << "Ingrese el nuevo time signature: ";
274                 cin >> cancion->time_signature;
275                 break;
276             case 20:
277                 cout << "Saliendo de la actualización de canción..." <<
                        endl;
278                 break;
279             default:
280                 cout << "Opción no válida." << endl;
281                 break;
282         }
283     } while (opcion != 20);
```

```
284
285     cout << "Canción actualizada:\n";
286     cancion->imprimirDatos();
287 }
```

Listing 8: Cabecera de la Clase Menú

4.9. main.cpp

```
1 #include "menu.h"
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     Menu menu;
7     menu.interfaz_menu();
8     return 0;
9 }
```

Listing 9: Código del main

5. 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.