[1]

Gestor de Lista de Reproducción de Canciones En C++

Esther Chunga Pacheco, Piero Fabricio Poblette Andía y Allison Mayra Usedo Quispe
Emails:
echungap@ulasalle.edu.pe
ppoblettea@ulasalle.edu.pe
ausedoq@ulasalle.edu.pe

Resumen - El proyecto desarrolla un gestor de listas de reproducción en C + + que optimiza la gestión de grandes volúmenes de datos musicales mediante estructuras avanzadas como Trie y vectores dinámicos. Se incluye búsqueda eficiente, ordenamientos personalizados y manipulación dinámica de listas. Las pruebas se realizaron con más de un millón de canciones, demostrando eficiencia en memoria y rendimiento.

Índice de Términos -C + +, estructuras de datos, gestión de canciones, listas de reproducción, Trie.

I. Introducción

Este proyecto presenta un gestor de lista de reproducción diseñado en C + + que combina el uso de estructuras de datos avanzadas y un manejo eficiente de memoria.

El programa desarrollado implementa una lista de reproducción dinámica que permite realizar operaciones como agregar, eliminar, reordenar y buscar canciones. Las pruebas se realizaron con un conjunto de datos que incluye atributos como popularidad, año, género y características acústicas de más de un millón de canciones, cubriendo un período desde el año 2000 hasta 2023.

El objetivo de este proyecto es aplicar conocimientos sobre estructuras de datos, como Trie y vectores dinámicos, en un problema práctico, demostrando su utilidad en la gestión de datos masivos.

II. Objetivos

Los objetivos principales de este proyecto

Diseñar un programa eficiente para gestionar listas de reproducción usando estructuras avanzadas.

1. Implementar funcionalidades clave:
   - Agregar y eliminar canciones.
   - Reorganizar canciones según criterios como popularidad, duración o año.
   - Búsqueda rápida por prefijo en nombres de canciones o artistas.
2. Optimizar el manejo dinámico de memoria.
3. Proveer vistas dinámicas de la lista con opciones de filtrado y ordenamiento.

III. Metodología

A. *Estructuras de Datos Implementadas*

1. Clase Cancion
   Representa una canción mediante atributos como artist_name, track_name, track_id, popularity, y características acústicas como danceability y energy.
2. Trie
   Esta estructura permite realizar búsquedas rápidas por prefijo en los nombres de canciones y artistas, optimizando los tiempos de búsqueda en listas grandes.
3. Vectores Dinámicos
   Se utilizan para almacenar la lista de canciones de forma dinámica, permitiendo la adición y eliminación eficiente de elementos.

B. *Funcionalidades Principales*

1. Agregar Canciones
   Permite añadir canciones con todos sus atributos relevantes a la lista de reproducción.
2. Eliminar Canciones
   Las canciones pueden eliminarse mediante su ID o nombre.
3. Reorganizar Canciones
   Los usuarios pueden modificar el orden de las canciones en la lista de reproducción.
4. Búsqueda por Prefijo
   Utilizando Trie, se realizan búsquedas rápidas para encontrar canciones o artistas con nombres similares.
5. Vistas Dinámicas
   Los usuarios pueden filtrar y ordenar canciones según popularidad, duración u otros atributos, en orden ascendente o descendente.

[1]

IV. Código Principal

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <string>
#include <vector>
#include <windows.h>
#include <algorithm>
#include <locale>
#include <list>
using namespace std;

class Cancion {
    // Definición de atributos y métodos
para manejar cada canción.
};

void Cancion::imprimir() const {
    // Método que imprime los detalles de
cada canción.
}

// Funciones para manejar la lista de
canciones.
const Cancion*
MostrarSeleccion(vector<const Cancion*>
result);
// Muestra una lista de canciones y
permite hacer una selección.


// Funciones para insertar y eliminar
canciones en un Trie (para búsqueda
eficiente).
void insertKey(TrieNode* root, const
Cancion* cancion, const string& key);
// Inserta una canción en el Trie
utilizando una clave.

bool deleteFromTrie(TrieNode* root, const
string& key);
// Elimina una canción del Trie utilizando
la clave.

void collectWords(TrieNode* curr, string
currentPrefix, vector<const Cancion*>&
result);
// Recolecta las canciones que coinciden
con un prefijo dado.

const Cancion*
getWordsWithPrefix(TrieNode* root, const
string& prefix);
// Obtiene las canciones que coinciden con
un prefijo específico.
```

```
// Funciones relacionadas con la tabla
hash para géneros de canciones.
int hashFunction(const string& key);
// Función hash para calcular el índice
basado en una clave (género de la
canción).

void insertarPorGenero(const Cancion*
cancion, const string& genre);
// Inserta una canción en la tabla hash
según su género.

const Cancion* mostrarPorGenero(const
string& genre);
// Muestra las canciones asociadas a un
género específico utilizando la tabla
hash.

const Cancion* imprimirTablaGeneros();
// Imprime todas las canciones agrupadas
por género.


// Funciones para manejar el índice de
canciones por año.
void insertarYear(int indiceCategoria,
const Cancion* cancion);
// Inserta una canción en el índice según
su año de publicación.

const Cancion* buscarYear(int
indiceCategoria);
// Busca canciones en un índice de año
específico.

void eliminar(int indiceCategoria, const
Cancion* cancion);
// Elimina una canción en un índice de año
específico.

const Cancion* mostrarYear();
// Muestra las canciones agrupadas por
año.


// Funciones para manipular la lista de
reproducción de canciones.
void insertar(const Cancion* c);
// Inserta una canción en la lista de
reproducción.

void agregar_cancion(const Cancion& c);
// Agrega una canción a la lista de
reproducción.

void eliminar_cancion(const string&
nombreCancion);
// Elimina una canción de la lista de
reproducción usando su nombre.
```

```
void cambiar_orden(int posicion_actual,
int nueva_posicion);
// Cambia el orden de una canción en la
lista de reproducción según las posiciones
indicadas.

void imprimirCanciones() const;
// Imprime todas las canciones de la lista
de reproducción.

NodoLista* obtenerCabeza();
// Obtiene el primer nodo de la lista de
canciones.

void llenarTrie(TrieNode* root);
// Llena el Trie con las canciones de la
lista de reproducción.

void llenarIndiceYear(IndiceCancionesYear*
year);
// Llena el índice de canciones por año.

void llenarTablaHashPorGenero(TablaHash&
tabla);
// Llena la tabla hash de géneros con las
canciones de la lista.

~ListaReproduccion();
// Destructor que limpia los nodos de la
lista de reproducción.

void freeTrie(TrieNode* node);
// Libera la memoria de los nodos del
Trie.

float validarFloat(const string& str);
// Valida y convierte un string a float.

string limpiarCadena(const string& input);
// Limpia una cadena de texto de
caracteres no deseados.

void leerArchivoCSV(const string&
nombreArchivo, TablaHash& tablaHashGenre,
ListaReproduccion& lista, TrieNode* root,
IndiceCancionesYear* year);
// Lee un archivo CSV, procesa las
canciones y las inserta en la lista, Trie
y tabla hash.

void menu(ListaReproduccion& lista);
// Muestra el menú principal y maneja las
acciones del usuario.

int main() {
    // Código principal que inicializa
estructuras de datos y llama a las
funciones correspondientes.
    SetConsoleOutputCP(CP_UTF8);
    ListaReproduccion lista;

    lista.root = new TrieNode();
    lista.tablaHashGenre = new
TablaHash(82);
    lista.year = new
IndiceCancionesYear();

    leerArchivoCSV("spotify_data.csv",
*(lista.tablaHashGenre), lista,
lista.root, lista.year);
    lista.llenarTrie(lista.root);
    lista.llenarIndiceYear(lista.year);

lista.llenarTablaHashPorGenero(*(lista.tab
laHashGenre));

    menu(lista);

    freeTrie(lista.root);

    return 0;
}
```

V. IMPLEMENTACIÓN

A. Clase Cancion

La clase Cancion almacena los atributos de cada canción y ofrece métodos para acceder a los datos de la canción.

```
class Cancion {
private:
    string artist_name, track_name,
track_id, genre;
    int popularity, year;
    float danceability, energy, loudness,
speechiness, acousticness;
    float instrumentalness, liveness,
valence, tempo;
    bool mode;
    int duration_ms, time_signature;
public:
    // Constructor
    Cancion(string artist_name, string
track_name, string track_id, int
popularity, int year, string genre,
            float danceability, float
energy, int key, float loudness, bool
mode, float speechiness,
            float acousticness, float
instrumentalness, float liveness, float
valence, float tempo,
            int duration_ms, int
time_signature)
```

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

    // Método para imprimir los detalles
de la canción
    void imprimir() const {
        cout << "Artista: " <<
artist_name << ", Canción: " <<
track_name << ", Género: " << genre << ",
Año: " << year << endl;
    }
    // Métodos de acceso para obtener los
atributos de la canción
    string getArtistName() const { return
artist_name; }
    string getTrackName() const { return
track_name; }
    string getTrackId() const { return
track_id; }
    string getGenre() const { return
genre; }
    int getYear() const { return year; }
};
```

B. Implementación Estructura TrieNode y Funciones del Trie

El Trie se utiliza para realizar búsquedas por prefijo de forma eficiente. Cada nodo del Trie contiene los caracteres de las canciones, lo que permite buscar canciones que comienzan con un prefijo dado.

Estructura TrieNode

```
struct TrieNode {
    TrieNode* child[256];  //
Arreglo de punteros a hijos.
    vector<const Cancion*>
canciones;  // Lista de canciones
asociadas.
    bool wordEnd;  // Marca si es el
final de una palabra.

    TrieNode() {
        wordEnd = false;
        fill(begin(child),
end(child), nullptr);  // Inicializa
los punteros a nullptr.
    }

    int getIndex(char ch) {
        // Retorna el índice
correspondiente al carácter.
        if (ch >= 'a' && ch <= 'z')
return ch - 'a';
        if (ch >= 'A' && ch <= 'Z')
return ch - 'A' + 26;
        if (ch == ' ') return 52;
        if (ch == '\'') return 53;
        if (ch == '-') return 54;
        if (ch == '.') return 55;
```

Fig. 1.Estructura TrieNode

Método insertKey

```
void insertKey(TrieNode* root, const
Cancion* cancion, const string& key) {
    TrieNode* curr = root;
    for (char c : key) {
        int index = curr->getIndex(c);
        if (index == -1) continue;  //
Salta caracteres no válidos.

        if (!curr->child[index]) {
            curr->child[index] = new
TrieNode();  // Crea un nuevo nodo si no
existe.
        }
        curr = curr->child[index];
    }
    curr->wordEnd = true;  // Marca el
final de la palabra.
    curr->canciones.push_back(cancion);  //
Almacena la canción en el nodo.
}
```

Fig. 2 Método insertKey

Método deleteFromTrie

```
bool deleteFromTrie(TrieNode* root, const string&
key, int depth = 0) {
    if (!root) return false;

    if (depth == key.size()) {
        if (root->wordEnd) {
            root->wordEnd = false;  // Elimina la
palabra.
            root->canciones.clear();  // Elimina
las canciones asociadas.
        }
        // Si el nodo no tiene hijos, lo elimina.
        for (int i = 0; i < 57; ++i) {
            if (root->child[i]) return false;
        }
        delete root;
        return true;
    }

    int index = root->getIndex(key[depth]);
    if (index == -1 || !root->child[index])
return false;

    bool shouldDelete =
deleteFromTrie(root->child[index], key, depth +
1);

    // Elimina el nodo si no tiene palabras
asociadas ni hijos.



if (shouldDelete) {
        root->child[index] = nullptr;
        if (!root->wordEnd &&
root->canciones.empty()) {
            for (int i = 0; i < 57; ++i)
{
                if (root->child[i])
return false;
            }
            delete root;
            return true;
        }
    }

    return false;
}
```

Fig. 3 Método deleteFromTrie

Método collectWords

```
void collectWords(TrieNode* curr, string
currentPrefix, vector<const Cancion*>& result) {
    if (!curr) return;

    if (curr->wordEnd) {
        result.insert(result.end(),
curr->canciones.begin(), curr->canciones.end());  //
Añade canciones al resultado.
    }

    for (int i = 0; i < 76; ++i) {
        if (curr->child[i]) {
            char nextChar = (i < 26) ? 'a' + i : (i <
52) ? 'A' + (i - 26) : i == 52 ? ' ' : i == 53 ? '\''
: '-';  // Traduce el índice a un carácter.
            collectWords(curr->child[i], currentPrefix
+ nextChar, result);
        }
    }
}
```

Fig. 4 Método collectWords

Método getWordsWithPrefix

```
const Cancion*
getWordsWithPrefix(TrieNode* root,
const string& prefix) {
    TrieNode* curr = root;
    vector<const Cancion*> result;

    for (char c : prefix) {
        int index = curr->getIndex(c);
        if (index == -1 ||
!curr->child[index]) {
            cout << "No encontrado." <<
endl;
            return nullptr;
        }
        curr = curr->child[index];
    }

    collectWords(curr, prefix, result);
// Recoge todas las canciones con el
prefijo.
    const Cancion* seleccionada =
MostrarSeleccion(result);  // Muestra
las canciones encontradas.
    return seleccionada;
}
```

Fig. 5 Método getWordsWithPrefix

Explicación:

- TrieNode: Cada nodo representa una posible parte de una palabra (en este caso, el nombre del artista o canción).
- insertKey: Inserta una canción asociada a un prefijo o clave dentro del Trie.
- deleteFromTrie: Elimina una clave (y sus canciones asociadas) del Trie, liberando la memoria.
- collectWords: Recolecta todas las canciones que corresponden a un prefijo, navegando por los nodos hijos.
- getWordsWithPrefix: Busca canciones que coincidan con un prefijo, mostrando las coincidencias.

C. Tabla Hash

Esta estructura nos ayudará a proporcionar una manera eficiente de buscar y clasificar canciones en función de su género, optimizando el tiempo de respuesta en la inserción y búsqueda de canciones.

Declaración de la clase TablaHash

```
class TablaHash {
private:
    vector<vector<const Cancion*>>
tablaGenero;  // Contenedor de canciones,
indexado por hash de género
    int capacidad;  // Tamaño de la tabla
hash
```

Fig. 6. Declaración de la clase TablaHash

Función Hash (hashFunction)

```
    int hashFunction(const string& key)
{
        int hash = 0;
        for (char c : key) {
            hash = (hash + int(c)) %
capacidad;  // Calcula el valor hash a
partir del género
        }
        return hash;  // Devuelve el
índice calculado
    }
```

Fig. 7. Función Hash (hashFunction)

Constructor (TablaHash(int cap))

```
public:
    TablaHash(int cap) : capacidad(cap)
{
        tablaGenero.resize(capacidad);
// Ajusta el tamaño de la tabla
    }
```

Fig. 8. Constructor (TablaHash(int cap))

Método insertarPorGenero

```
    void insertarPorGenero(const Cancion*
cancion) {
        int index =
hashFunction(cancion->getGenre());  //
Obtiene el índice del género

tablaGenero[index].push_back(cancion);
// Inserta la canción en el índice
correspondiente
    }
```

Fig. 9. Método insertarPorGenero

Método mostrarPorGenero

```
    const Cancion* mostrarPorGenero(const
string& genre) {
        int index = hashFunction(genre);
// Obtiene el índice del género
        vector<const Cancion*> canciones;
// Lista para almacenar canciones
encontradas
        // Recorre las canciones en la
lista correspondiente al índice
        for (const auto& cancion :
tablaGenero[index]) {
            if (cancion->getGenre() ==
genre) {  // Verifica si el género
coincide

canciones.push_back(cancion);  // Si
coincide, agrega la canción a la lista
            }
        }
        // Si se encuentran canciones,
devuelve la primera
        if (!canciones.empty()) {
            return canciones[0];  //
Devuelve la primera canción que coincida
        }
        return nullptr;  // Si no se
encuentra ninguna, retorna nullptr
    }
```

Fig. 10. Método mostrarPorGenero

Método imprimirTablaGeneros

```
    void imprimirTablaGeneros() {
        for (int i = 0; i < capacidad; ++i)
{  // Recorre cada índice de la tabla
            if (!tablaGenero[i].empty()) {
// Si el índice contiene canciones
                cout << "Índice " << i << ":
";
                for (const auto& cancion :
tablaGenero[i]) {
                    cout << cancion->nombre
<< " (" << cancion->getGenre() << ") ";
                }
                cout << endl;
            }
        }
    }
```

Fig. 11. Método imprimirTablaGeneros

Explicación

- hashFunction: Esta función genera un valor numérico (hash) a partir de una cadena de texto (el género de la canción). El valor hash se utiliza para distribuir las canciones y permiten realizar búsquedas rápidas.
- insertarPorGenero: Inserta una canción en la tabla hash, asociándose con un índice que corresponde a su género. La canción se almacena en el vector adecuado dentro de la tabla hash, basado en el valor calculado por la hashFunction.
- mostrarPorGenero: Busca y retorna canciones que coincidan con un género específico. Calcula el índice en la tabla usando la hashFunction y recorre las canciones buscando las que coinciden con el género deseado.
- imprimirTablaGeneros: Recorre toda la tabla hash e imprime las canciones almacenadas en cada índice. Si existen canciones en un índice específico, las selecciona y las muestra utilizando la función MostrarSeleccion.

D. Clase IndiceCancionesYear

Para organizar y gestionar canciones según el año de su lanzamiento. Utiliza un índice basado en el rango de años, almacenando las canciones en vectores dentro de un arreglo. Esta estructura nos permite un acceso rápido a las canciones por su año, facilitando operaciones como inserción, búsqueda, eliminación y visualización de canciones.

```
Constructor

IndiceCancionesYear() {
    indice.resize(24);
}
```

Fig. 12. Constructor

```
insertarYear

void insertarYear(int
indiceCategoria, const Cancion*
cancion) {
    if (indiceCategoria >= 2000 &&
indiceCategoria <= 2023) {

indice[indiceCategoria-2000].push_bac
k(cancion);
    }
}
```

Fig. 13. InsertarYear

buscarYear

```
const Cancion* buscarYear(int
indiceCategoria) {
    if (indiceCategoria >= 2000 &&
indiceCategoria <= 2023) {
        const Cancion* seleccionada =
MostrarSeleccion(indice[indiceCategoria-20
00]);
        return seleccionada;
    } else {
        cout << "Índice fuera de rango!"
<< std::endl;
        return {};
    }
}
```

Fig. 14. BuscarYear

eliminar

```
void eliminar(int indiceCategoria, const
Cancion* cancion) {
    if (indiceCategoria >= 0 &&
indiceCategoria < indice.size()) {
        auto& canciones =
indice[indiceCategoria];
        for (auto it = canciones.begin();
it != canciones.end(); ++it) {
            if (*it == cancion) {
                canciones.erase(it);
                delete cancion;
                return;
            }
        }
        std::cout << "Canción no
encontrada!" << std::endl;
    } else {
        std::cout << "Índice fuera de
rango!" << std::endl;
    }
}
```

Fig. 15. Eliminar

mostrarYear

```
const Cancion* mostrarYear() {
    for(int i=0; i<=23; i++){
        const Cancion* seleccionada =
MostrarSeleccion(indice[i]);
        return seleccionada;
    }
}
```

Fig. 15. MostrarYear

Explicación

- IndiceCancionesYear: Esta clase organiza las canciones por el año de su lanzamiento, utilizando un índice basado en los años. Cada canción se almacena en el vector correspondiente al año de su lanzamiento.
- insertarYear:Permite insertar una canción en el índice según el año proporcionado. Si el año está dentro del rango, la canción se agrega al vector correspondiente al año.
- buscarYear: Busca una canción dentro del índice según el año proporcionado. Si el año es válido, se selecciona una canción del índice correspondiente; si no, se muestra un mensaje de error.
- eliminar: Permite eliminar una canción de un año específico dentro del índice. Si la canción se encuentra en el año indicado, se elimina y se libera la memoria asociada a la canción. Si no se encuentra, muestra un mensaje de error.
- mostrarYear: Muestra una canción de cualquier año dentro del rango, que se selecciona una canción al azar del índice de cada año.

E. Clase NodoLista

Es un nodo utilizado en una estructura de lista enlazada. Cada nodo contiene una canción (Cancion) y un puntero al siguiente nodo en la lista. Es una estructura básica utilizada para almacenar y enlazar canciones en un formato secuencial.

```
class NodoLista {
public:
    Cancion cancion;
    NodoLista* siguiente;

    NodoLista(const Cancion& c) :
cancion(c), siguiente(nullptr) {}
};
```

Fig. 16. Clase NodoLista

Explicación

- NodoLista: Cada nodo contiene una canción (cancion) y un puntero siguiente que apunta al siguiente nodo de la lista.
- Cancion cancion: Es un objeto de la clase Cancion que se almacena en el nodo. Almacena los datos de una canción (como el nombre, el artista, etc.).
- NodoLista(const Cancion& c): Es el constructor. Recibe una canción como parámetro y la almacena en el nodo. Inicializa el puntero siguiente como nullptr, lo que significa que, al principio, el nodo no tiene un siguiente nodo.

F. Clase ListaReproduccion

Gestiona una lista de canciones organizadas en una estructura de lista enlazada, pero también integra funcionalidades adicionales como la manipulación de datos a través de un Trie (para búsquedas de artistas), una Tabla Hash (para clasificar por género) y un índice por año. Estas estructuras permiten realizar operaciones de inserción, eliminación y búsqueda de canciones de manera eficiente.

Clase ListaReproduccion

```
class ListaReproduccion {
private:
    NodoLista* cabeza;
    NodoLista* ultimo;
    int tamano;
public:
    ListaReproduccion() : cabeza(nullptr),
ultimo(nullptr), tamano(0) {
        root = new TrieNode();
        tablaHashGenre = new TablaHash(82);
        year = new IndiceCancionesYear();
    }
    TrieNode* root;
    TablaHash* tablaHashGenre;
    IndiceCancionesYear* year;
```

Fig. 17. Clase ListaReproduccion

Métodos

insertar(const Cancion* c)

```
void insertar(const Cancion* c) {
    NodoLista* nuevo = new NodoLista(*c);
    if (!cabeza) {
        cabeza = ultimo = nuevo;
    } else {
        ultimo->siguiente = nuevo;
        ultimo = nuevo;
    }
    tamano++;

    if (root) {
        insertKey(root, c,
c->getArtistName());
    }

    if (tablaHashGenre) {
        tablaHashGenre->insertarPorGenero(c,
c->getGenre());
    }

    if (year) {
        year->insertarYear(c->getYear(), c);
    }
}
```

Fig. 18. insertar(const Cancion* c)

agregar_cancion(const Cancion& c)

```
void agregar_cancion(const Cancion& c) {
    NodoLista* nuevo = new NodoLista(c);
    if (!cabeza) {
        cabeza = ultimo = nuevo;
    } else {
        ultimo->siguiente = nuevo;
        ultimo = nuevo;
    }
    tamano++;
}
```

Fig. 18. agregar_cancion(const Cancion& c)

eliminar_cancion(const string& nombreCancion)

```
void eliminar_cancion(const string& nombreCancion)
{
    NodoLista* temp = cabeza;
    NodoLista* anterior = nullptr;

    while (temp && temp->cancion.getTrackName() !=
nombreCancion) {
        anterior = temp;
        temp = temp->siguiente;
    }

    if (temp) {
        if (anterior) {
            anterior->siguiente = temp->siguiente;
        } else {
            cabeza = temp->siguiente;
        }
        delete temp;
        tamano--;
        cout << "Canción eliminada: " <<
nombreCancion << endl;
    } else {
        cout << "Canción no encontrada: " <<
nombreCancion << endl;
    }
}
```

Fig. 19. eliminar_cancion(const string& nombreCancion)

cambiar_orden(int posicion_actual, int nueva_posicion)

```
void cambiar_orden(int posicion_actual, int
nueva_posicion) {
    if (posicion_actual < 0 ||
posicion_actual >= tamano || nueva_posicion
< 0 || nueva_posicion >= tamano) {
        cout << "Posición no válida.\n";
        return;
    }
    if (posicion_actual == nueva_posicion)
return;

    NodoLista* temp = cabeza;
    NodoLista* anterior_actual = nullptr;

    for (int i = 0; i < posicion_actual;
++i) {
        anterior_actual = temp;
        temp = temp->siguiente;
    }
    NodoLista* cancion_a_mover = temp;
```

```
 if (anterior_actual) {
        anterior_actual->siguiente =
cancion_a_mover->siguiente;
    } else {
        cabeza =
cancion_a_mover->siguiente;
    }

    temp = cabeza;
    NodoLista* anterior_nueva = nullptr;
    for (int i = 0; i < nueva_posicion;
++i) {
        anterior_nueva = temp;
        temp = temp->siguiente;
    }

    if (anterior_nueva) {
        cancion_a_mover->siguiente =
anterior_nueva->siguiente;
        anterior_nueva->siguiente =
cancion_a_mover;
```

Fig. 20. cambiar_orden(int posicion_actual, int nueva_posicion)

imprimirCanciones()

```
void imprimirCanciones() const {
    NodoLista* temp = cabeza;
    while (temp) {
        cout << "Artista: " <<
temp->cancion.getArtistName()
             << ", Título: " <<
temp->cancion.getTrackName() <<
endl;
        temp = temp->siguiente;
    }
}
```

Fig. 21. imprimirCanciones()

llenarTrie(TrieNode* root)

```
void llenarTrie(TrieNode* root) {
    NodoLista* temp = cabeza;
    while (temp) {
        insertKey(root,
&temp->cancion,
temp->cancion.getArtistName());
        temp = temp->siguiente;
    }
}
```

Fig. 22. llenarTrie(TrieNode* root)

**llenarIndiceYear(IndiceCancionesYea
r* year)**

```
void
llenarIndiceYear(IndiceCancionesYear
* year) {
    NodoLista* temp = cabeza;
    while (temp) {

year->insertarYear(temp->cancion.get
Year(), &temp->cancion);
        temp = temp->siguiente;
    }
}
```

Fig. 23. llenarIndiceYear(IndiceCancionesYear* year)

llenarTablaHashPorGenero(TablaHash& tabla)

```
void llenarTablaHashPorGenero(TablaHash&
tabla) {
    NodoLista* temp = cabeza;
    while (temp) {

tabla.insertarPorGenero(&temp->cancion,
temp->cancion.getGenre());
        temp = temp->siguiente;
    }
}
```

Fig. 24. llenarTablaHashPorGenero(TablaHash& tabla)

Explicación

1. ListaReproduccion (Clase Principal):
   - cabeza: Puntero al primer nodo de la lista.
   - ultimo: Puntero al último nodo.
   - tamano: Cuenta el número de canciones en la lista.
   - root: Puntero al nodo raíz de un Trie, usado para búsquedas por artista.
   - tablaHashGenre: Tabla hash que organiza canciones por género.
   - year: Índice de canciones organizado por año de publicación.
2. NodoLista (Nodo de la Lista):
   - cancion: Objeto de la clase Cancion que almacena los datos de la canción.
   - siguiente: Puntero al siguiente nodo de la lista.
3. NodoLista(const Cancion& c):
   Constructor del nodo que recibe una canción, la almacena y establece el puntero siguiente como nullptr.
4. Métodos de ListaReproduccion:
   - insertar(): Añade una canción al final de la lista y actualiza las estructuras de datos .
   - eliminar_cancion(): Elimina una canción buscando por su nombre.
   - cambiar_orden(): Permite mover una canción de una posición.
   - imprimirCanciones(): Muestra todas las canciones de la lista.

G. *freeTrie(TrieNode* node)*

Es responsable de liberar la memoria utilizada por el Trie. El Trie es una estructura de datos que se utiliza para realizar búsquedas rápidas, por lo que es crucial liberar la memoria cuando ya no sea necesario.

```
void freeTrie(TrieNode* node) {
    if (!node) return; // Base case,
si el nodo es nulo, no hacer nada

    // Liberar recursivamente los
hijos de este nodo
    for (int i = 0; i < 76; ++i) {  //
76 es el número máximo de hijos
posibles (en base a caracteres)
        if (node->child[i]) {
            freeTrie(node->child[i]);
        }
    }
    delete node;  // Liberar la
memoria del nodo actual
}
```

Fig. 25. freeTrie(TrieNode* node)

H. *validarFloat(const string& str)*

Valida si una cadena de texto se puede convertir a un número de tipo float. Si la conversión falla, devuelve 0.0f.

```
float validarFloat(const string&
str) {
    try {
        return stof(str);  //
Intentar convertir la cadena a float
    } catch (...) {
        return 0.0f;  // Si ocurre
un error, devolver 0.0f
    }
}
```

Fig. 26. validarFloat(const string& str)

I. *limpiarCadena(const string& input)*

Esta función limpia una cadena de texto, eliminando caracteres especiales y convirtiendo caracteres con acentos a sus versiones sin acento. Esto es útil para normalizar entradas de texto y permitir búsquedas más consistentes.

```
string limpiarCadena(const string& input) {
    string resultado;
    for (char c : input) {
        switch (c) {
case 'á': case 'Á': resultado += 'a'; break;
case 'é': case 'É': resultado += 'e'; break;
case 'í': case 'Í': resultado += 'i'; break;
case 'ó': case 'Ó': resultado += 'o'; break;
case 'ú': case 'Ú': resultado += 'u'; break;
case 'ñ': case 'Ñ': resultado += 'n'; break;
case 'ç': case 'Ç': resultado += 'c'; break;
            default:
                if (isalnum(c) || isspace(c)
|| c == '-' || c == '\'' || c == '!' || c ==
'(' || c == ')') {
                    resultado += c;
                }
                break;
        }
    }
    return resultado;
}
```

Fig. 27. limpiarCadena(const string& input)

J. *leerArchivoCSV(const string& nombreArchivo, TablaHash& tablaHashGenre, ListaReproduccion& lista, TrieNode* root, IndiceCancionesYear* year)*

Lee un archivo CSV y carga las canciones en una lista de reproducción, mientras actualiza varias estructuras de datos como un Trie, tabla hash y un índice por año.

```
void leerArchivoCSV(const string&
nombreArchivo, TablaHash& tablaHashGenre,
ListaReproduccion& lista, TrieNode* root,
IndiceCancionesYear* year ) {
    ifstream archivo(nombreArchivo);
    if (!archivo.is_open()) {
        cout << "No se pudo abrir el
archivo.\n";
        return;
    }

    string linea;
    if (getline(archivo, linea)) {}  //
Ignorar encabezados del archivo CSV.

    while (getline(archivo, linea)) {
        stringstream ss(linea);
        string dummy, artist_name,
track_name, track_id, genre;
        int popularity, year, key,
duration_ms, time_signature;
        float danceability, energy,
loudness, speechiness, acousticness;
        float instrumentalness, liveness,
valence, tempo;
        bool mode;
```

Fig.28.leerArchivoCSV(const string& nombreArchivo, TablaHash& tablaHashGenre, ListaReproduccion& lista, TrieNode* root, IndiceCancionesYear* year)

K. menu(ListaReproduccion& lista)

Se implementa un menú interactivo para que el usuario pueda interactuar con las listas de reproducción, agregar canciones, eliminar canciones y más.

```
void menu(ListaReproduccion& lista) {
    map<string, ListaReproduccion>
listasReproduccion;
    int opcion;

    do {
        cout << "\n=== Menú ===" << endl;
        cout << "1. Crear una nueva lista de
reproducción" << endl;
        cout << "2. Seleccionar una lista
para trabajar" << endl;
        cout << "3. Salir" << endl;
        cout << "Ingrese su opción: ";
        cin >> opcion;
        cin.ignore(); // Limpiar el buffer
de entrada.

        switch (opcion) {
            case 1: {
                string nombreLista;
                cout << "Ingrese el nombre
de la nueva lista de reproducción: ";
                getline(cin, nombreLista);

                if
(listasReproduccion.find(nombreLista) !=
listasReproduccion.end()) {



 cout << "Ya existe una lista con ese
nombre. Intente con otro nombre." << endl;
                } else {
                    ListaReproduccion
nuevaLista;
                    nuevaLista.root = new
TrieNode();

nuevaLista.tablaHashGenre = new
TablaHash(82);
                    nuevaLista.year = new
IndiceCancionesYear();

                    // Agregar la nueva
lista al mapa

listasReproduccion[nombreLista] =
nuevaLista;

                    cout << "Lista de
reproducción '" << nombreLista << "' creada
con éxito." << endl;
                }
                break;
            }
            // Resto del menú...
        }
    } while (opcion != 3);
}
```

Fig. 29. menu(ListaReproduccion& lista)

L. int main

Formato de codificación de salida de la consola a para asegurar que se manejen correctamente los caracteres especiales y acentuados (como á, é, í, etc.).

```
Inicialización de la lista de
reproducción

ListaReproduccion lista;

lista.root = new TrieNode();
lista.tablaHashGenre = new
TablaHash(82);
lista.year = new IndiceCancionesYear();
```

Fig. 30. Inicialización de la lista de reproducción

```
Lectura del archivo CSV

leerArchivoCSV("spotify_data.csv",
*(lista.tablaHashGenre), lista,
lista.root, lista.year);
```

Fig. 31. Lectura del archivo CSV

```
Llenado de las estructuras de datos

lista.llenarTrie(lista.root);
lista.llenarIndiceYear(lista.year);
lista.llenarTablaHashPorGenero(*(lista.t
ablaHashGenre));
```

Fig. 32. Llenado de las estructuras de datos

```
Llamada al menú de interacción

menu(lista);
```

Fig. 33. Llamada al menú de interacción

```
Liberación de memoria

freeTrie(lista.root);
```

Fig. 34. Liberación de memoria

Explicación

1. Inicialización de las estructuras: (Trie, TablaHash y IndiceCancionesYear).
2. Lectura de los datos: Se leen los datos del archivo CSV y se agregan a las estructuras.
3. Población de estructuras: Se llenan las estructuras con los datos de las canciones para optimizar la búsqueda.
4. Interacción con el usuario: El menú permite que el usuario gestione las listas de reproducción y las canciones.
5. Liberación de memoria: Finalmente, se libera la memoria utilizada por el Trie.

VI. TABLAS

A. Estructura del Conjunto de Datos

	Descripción
track_id	Identificador único de la canción.
track_name	Nombre de la canción.
artist_name	Nombre del artista o banda.
genre	Género musical.
year	Año de lanzamiento.
popularity	Popularidad (escala de 0 a 100).
danceability	Nivel de bailabilidad (0 a 1).
energy	Energía de la canción (0 a 1).

B. Operaciones Soportadas por el Gestor

Operación	Descripción	Estructura de Datos Usada
Agregar canción	Inserta una nueva canción en la lista de reproducción.	Lista de reproducción
Eliminar canción	Elimina una canción específica por su nombre.	Lista de reproducción
Búsqueda por prefijo	Encuentra canciones que comiencen con un prefijo dado.	Trie
Reorganizar canciones	Cambia el orden según criterios como popularidad o año.	Lista dinámica
Filtrar por género	Obtiene canciones de un género específico.	Tabla hash
Filtrar por año	Encuentra canciones lanzadas en un año determinado.	Índice de años

C. Comparación de Estructuras de Datos

Estructura de Datos	Ventajas
Trie	Búsqueda rápida por prefijo.
Tabla Hash	Acceso rápido para búsquedas exactas.
Lista Dinámica	Manejo flexible y fácil inserción/eliminación.

VII. PRUEBAS Y RESULTADOS

D. Pruebas

Se utilizaron datos de prueba obtenidos de un subconjunto del archivo proporcionado (archive.zip). Este subconjunto contenía una selección representativa de canciones, abarcando múltiples géneros y rangos de popularidad, para garantizar una evaluación adecuada de todas las funcionalidades implementadas.

E. Resultados

1. Operaciones
   - Agregar canciones: Se realizaron pruebas para verificar la correcta inserción de canciones, logrando una operación en tiempo O(1) promedio gracias al uso de vectores dinámicos.
   - Eliminar canciones:Se eliminan eficientemente utilizando su track_id o track_name, con un mínimo en el uso de memoria.
2. Búsquedas
   Utilizando la estructura Trie, las búsquedas por prefijo para nombres de canciones o artistas mostraron tiempos lineales respecto al tamaño del prefijo O(k), asegurando un rendimiento óptimo en conjuntos de datos grandes.
3. Ordenamiento y Reorganización
   El ordenamiento por atributos como popularidad y duración se logró utilizando algoritmos de clasificación eficientes como std::sort, manteniendo un desempeño cercano a O(nlogn).
4. Reproducción Aleatoria
   La funcionalidad de reproducción aleatoria generó secuencias sin repeticiones utilizando generadores de números aleatorios estándar en C++.

VIII. CONCLUSIÓN

La implementación de estructuras de datos como vectores dinámicos y Trie permitió realizar operaciones como inserción, eliminación, búsqueda y cambio de orden con alta eficiencia.El programa manejó correctamente las pruebas de gran volumen de datos, manteniendo un uso eficiente de memoria y tiempos de respuesta aceptables.

IX. Trabajo Futuro

1. Interfaz Gráfica
   La inclusión de una interfaz gráfica puede mejorar significativamente la experiencia del usuario, permitiendo un acceso más intuitivo a las funcionalidades.
2. Optimización para Listas Extremadamente Grandes
   Se puede explorar el uso de estructuras de datos avanzadas o paralelización para manejar conjuntos de datos que superen el millón de registros, maximizando la escalabilidad del sistema.

Apéndice

A. Código Principal

Incluye el código completo o los fragmentos más relevantes del programa. Asegúrate de formatearlo correctamente con fuente monoespaciada y agrega comentarios en las partes clave.

B. Diseño de Estructuras de Datos

Describe en detalle las estructuras de datos utilizadas (por ejemplo, Trie, vectores, listas) con diagramas y explicaciones de su funcionamiento interno.

Reconocimiento

El desarrollo de este proyecto no habría sido posible sin el apoyo de diversas personas y recursos. Agradezco especialmente a nuestro docente, quien proporcionó los conocimientos y guías necesarias para comprender y aplicar estructuras de datos en problemas prácticos. Asimismo, valoro enormemente el acceso al conjunto de datos detallado, que permitió probar y validar las funcionalidades del gestor de listas de reproducción.

Este proyecto es un reflejo del esfuerzo conjunto y destaca el impacto positivo del trabajo colaborativo.

Biografía Autor(es)

Esther Chunga Pacheco, Piero Fabricio Poblette Andía y Allison Mayra Usedo Quispe son estudiantes de Ingeniería de Software en la Universidad La Salle. Juntos, comparten una pasión por el desarrollo de soluciones tecnológicas eficientes y optimizadas para el manejo de datos masivos.

El equipo ha trabajado de manera colaborativa en este proyecto, combinando sus habilidades individuales para desarrollar un sistema robusto y funcional que aborde las necesidades de la gestión de datos y optimización.