UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Estructura de Datos y Análisis de Algoritmos

Laboratorio N°2

Alumno: Israel Arias Panez.

Profesor: Mario Inostroza.
Ayudante: Esteban Silva.

Santiago - Chile
2-2020

TABLA DE CONTENIDOS

Índice de Figuras

CAPÍTULO 1. INTRODUCCIÓN

En este segundo laboratorio de la asignatura Análisis de algoritmos y estructura de datos se presenta el enunciado “SpotMusic”, en el cual se presenta la situación en la cual por una serie de sucesos como una posible guerra mundial y la expansión del virus que nos aqueja en la actualidad, la gran parte de la población a nivel mundial se ha debido mantener dentro de sus casas, lo que ha producido un aumento de la creación de contenido en línea en plataformas como: Youtube, Twitch o Tiktok. Todo el contenido creado en dichas plataformas tiene un factor fundamental: la música de fondo utilizada, debido a que afecta directamente en el numero de viewers o consumidores que tiene la transmisión.

Bajo ese contexto la streamer Wendy Geek nos solicita organizar su música con el fin de poder mantener un buen nivel de audiencia en sus trasmisiones, para ello en primer lugar será necesario el organizar su biblioteca de canciones tomando como criterio principal los puntos de popularidad de las canciones, luego por orden alfabético del autor en los casos de las canciones que tengan los mismos puntos de popularidad y finalmente por la duración de las canciones, en caso de las canciones con los mismos puntos de popularidad y mismo autor. Una vez organizada la biblioteca de la streamer se solicita la creación de una playlist, la cual debe incorporar las canciones más populares de acuerdo con los géneros de preferencia de ella y que también se encuentren dentro del tiempo estimado de duración de la transmisión, los datos serán entregados por la streamer en archivos de textos.

Para lograr generar la solución pedida por la streamer, se aplicarán técnicas, diseños y modelos con el conocimiento proveniente de la ciencia de la computación y de los algoritmos y estructura de datos, los cuales son conocimientos adquiridos en las clases de la asignatura, en la solución se ocuparán algoritmos de ordenamiento, estructuras de datos, uso de punteros, direcciones de memoria, alocación de memoria, etc. En especial para esta experiencia de laboratorio se ha solicitado el uso de estructura de datos basadas en listas para la creación de la solución. También se hará el análisis del algoritmo de la solución implementada, en base al tiempo y su complejidad.

En esta introducción del informe se ha contextualizado el problema a resolver en este laboratorio, en la próxima sección se hará el análisis del problema, haciendo énfasis en la metodología que se siguió para resolverlo, estructuras de datos usadas, entre otros. Luego en la sección de Algoritmos y estructura de datos se presentarán los algoritmos usados para el desarrollo de la solución, detallando su funcionamiento y tiempo T(n) y O( ). A continuación, en la sección de Análisis de resultados, se describirá los resultados obtenidos de la solución implementada, se analizarán también las falencias detectadas y propuestas de cómo mejorar estas mismas y otros ámbitos de la solución. Finalmente, en la sección de Conclusión se indicará el grado de logro de los objetivos iniciales y una conclusión respecto al trabajo en este laboratorio.

CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

2.1.1 Marco teórico

Para el mejor entendimiento de la solución que será planteada y del informe en general, conviene definir algunos conceptos que serán nombrados.

- Algoritmo: Conjunto de instrucciones o reglas definidas no ambiguas que permite solucionar un problema u otras tareas.
- Metodología: Conjunto de procedimientos racionales para alcanzar un objetivo.
- Archivo: Conjunto de bytes que son almacenados en un dispositivo, por ejemplo, un archivo de texto plano.
- Función: Pequeña parte de un programa que realiza una tarea en particular, recibiendo una entrada, procesándola y devolviendo una salida.
- Lista enlazada: Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior. Es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos.
- Cola: Una cola o Queue es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pull por el otro. También se le llama estructura FIFO (First In First Out), debido a que el primer elemento en entrar será también el primero en salir.
- Struct: Declaración de un tipo de dato compuesto que permite almacenar conjuntos de datos en un bloque de memoria simulando una “estructura”.
- Tiempo de ejecución: Es el intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo.
- Complejidad: Es una descripción de la cantidad de tiempo que lleva ejecutar un algoritmo.
- Estructura de datos: Es una forma particular de organizar datos en una computadora para que puedan ser utilizados de manera eficiente.
- CPU: Es el hardware dentro de un ordenador u otros dispositivos programables, que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.

2.1.2 Metodología de solución y herramientas utilizadas.

La metodología por seguir para la resolución de este laboratorio es el uso de una programación modular en el lenguaje de programación C o también llamada como “divide y vencerás”, la cual consiste en dividir el problema en subproblemas a fin de ir solucionándolos poco a poco.
Como datos de entrada se nos han entregado cuatro archivos:

- “artistas.in”, el cual contiene el nombre de los artistas con su id identificador.
- “generos.in”, el cual contiene el nombre de los géneros musicales con su id identificador.
- “canciones.in”, el cual contiene el nombre, duración, id del artista, id del genero al que pertenece y puntos de popularidad de la canción.
- “preferencias.in” el cual contiene la duración aproximada de la transmisión, junto al/los id de géneros solicitados para la creación de la playlist.

El programa seguirá la siguiente metodología de solución:
1) Leer el archivo de entrada “artistas.in”.
2) Recolectar la información del archivo (artistas e id) de cada artista en una lista enlazada.
3) Leer el archivo de entrada “generos.in”.
4) Recolectar la información del archivo (nombre del género e id) de cada género en una lista enlazada.
5) Leer el archivo de entrada “canciones.in”
6) Recolectar la información del archivo (nombre, duración, id del artista, id del género y puntos de popularidad) de cada canción en una lista enlazada.
7) Leer el archivo de entrada “preferencias.in”.
8) Recolectar el tiempo total de la playlist en dos variables, luego recolectar el id de los géneros en una cola (Queue).
9) Ordenar la lista de canciones de mayor a menor respecto a la popularidad de las canciones.
10) Volver a ordenar la lista de canciones ya ordenada, ahora respecto a orden alfabético del autor en casos donde dos o más canciones tengan la misma popularidad y también orden por duración de canción, en casos donde coincida la popularidad y autor de distintas canciones, quedando la canción con menos duración antes que la canción con más duración.
11) Creación de archivo de salida “salida.out” la cual es la biblioteca de canciones ordenada.
12) Creación de playlist en una lista enlazada, según requisitos pedidos en el archivo “preferencias.in”, se hace en base a la lista de canciones ordenada considerando los requisitos pedidos.
13) Creación de archivo de salida “playlist.out” la cual contiene la playlist creada según los requisitos pedidos.

14) Liberación de memoria de las listas enlazadas y cola ocupadas.

Para la lectura de archivos se usara la función fopen() con el modo de lectura.
Para trabajar de una manera más simple se crearon tres structs:

- Struct artista: la cual contiene dos parámetros, nombre e id del artista.
- Struct género: la cual contiene dos parámetros, nombre e id del género.
- Struct canción: la cual contiene seis parámetros: nombre de la canción, duración en minutos, minutos de duración, segundos de duración, id del artista, id del genero y puntos de popularidad de la canción.

Para la recolección de datos para los puntos 2), 4) y 6) se hará uso de la estructura de datos "lista enlazada simple", se creara una lista enlazada y luego usando un algoritmo iterativo se recorrerá el archivo leído de principio a fin, almacenando todos los datos dentro de una struct, la cual será distinta, dependiendo si almacenamos artistas, géneros o canciones, luego la instancia de esa struct se almacenara dentro de la lista enlazada, así hasta llegar al fin del documento, los valores se conseguirán con el uso de la función fscanf().

Para la recolección de datos del punto 8) se hizo uso de la estructura de datos "cola", la cual fue implementada a través de una lista enlazada simple, luego el procedimiento de recolección fue similar para los otros casos, el uso de un algoritmo iterativo para recorrer el documento y guardar todas las preferencias, para este caso no se utilizo ninguna struct, simplemente se encolaron todas las preferencias que se fueron encontrando.

Para el punto 9) se hará uso del algoritmo iterativo visto en catedra "Bubble Sort", sin embargo, fue modificado para que funcionara en listas enlazadas y ordenara de mayor a menor, en base a la popularidad de las canciones dentro de la lista enlazada.

Para los puntos 11) y 13 hará uso de algoritmos iterativos para la escritura del archivo, además se debe velar respetar las estructuras del formato de escritura solicitados, los cuales son visibles en la figura 1 y 2.

```
10;dakiti;3:25;bad_bunny;trap
10;cut_my_lip;4:42;twenty_one_pilots;rap
9;safaera;4:55;bad_bunny;trap
7;dynamite;3:19;bts;k-pop
6;ride;3:34;twenty_one_piots;rap
5;lane_boy;4:13;twenty_one_pilots;rap
```

Figura 1: Ejemplo formato de archivo de salida "salida.out".

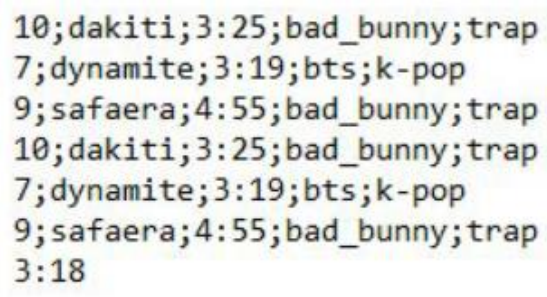


Figura 2: Ejemplo formato de archivo de salida “playlist.out”.

2.1.3 Algoritmos y estructuras de datos

Antes de describir los algoritmos usados para la metodología presentada en el punto 2.1.2, se presentarán las estructuras de datos que se utilizaran en la creación de esta solución, sin embargo, cabe preguntarse en un inicio, ¿por qué las estructuras de datos ayudan a facilitar la implementación de soluciones?

“Una “estructura de datos” es una **colección de valores**, la relación que existe entre estos valores y las operaciones que podemos hacer sobre ellos; en pocas palabras se refiere a cómo los datos están organizados y cómo se pueden administrar. Una estructura de datos describe el **formato** en que los valores van a ser almacenados, cómo van a ser accedidos y modificados, pudiendo así existir una gran cantidad de estructuras de datos. Las estructuras de datos son útiles porque siempre manipularemos datos, y si los datos están organizados, esta tarea será mucho más fácil.” [1]

Entonces al tener los datos organizados en estructura de datos, la implementación de la solución se facilitará, entonces a continuación se presentarán las estructuras de datos utilizadas a fin de facilitar la creación de la solución:

La implementación de la enlazada, junto a sus modificaciones para adecuarse a lo solicitado:

```
//TDA Lista Enlazada
typedef struct nodo nodo;

struct nodo{
    cancion cancion;
    artista artista;
    genero genero;
    int preferencia;
    struct nodo *sig; //puntero al siguiente nodo en la lista
};
```

Figura 3: Implementación lista enlazada

Como es posible observar en la figura 3, la lista enlazada fue modificada a lo que se conoce habitualmente, en vez de poseer un puntero al siguiente nodo y un dato, esta posee el puntero al siguiente nodo y espacio para almacenar cuatro datos distintos: canción, artista, genero y preferencia. Siendo canción, artista y genero structs cuya representación será presentada a continuación.

La struct cancion es la siguiente:

```
//Cada cancion tendra las siguientes caracteristicas:
struct cancion{
    char nombreCancion[60];
    int duracionMinutos;
    int duracionSegundos;
    int idArtista;
    int idGenero;
    int popularidad;
};
```

Figura 4: Struct cancion.

Como es posible apreciar en la figura 4 y como se menciono en el punto 2.1.2 del presente informe, la struct cancion almacenara el nombre, duración, id de artista, id de genero y puntaje de popularidad de una canción.

La struct artista es la siguiente:

```
//Cada artista tiene las siguientes caracteristicas:
struct artista{
    int idArtista;
    char nombreArtista[60];
};
```

Figura 5: Struct artista

Como también fue mencionado en el punto 2.1.2 del presente informe, la struct artista almacena el id y el nombre de un artista.

La struct género es la siguiente:

```
//Cada genero tiene las siguientes caracteristicas:
struct genero{
    int idGenero;
    char nombreGenero[60];
};
```

Figura 6: Struct genero.

La struct genero por su parte almacena el id y el nombre para un género musical.

Entonces tomando en consideración las figuras 3,4,5 y 6. La lista enlazada fue modificada de tal manera, para que permita almacenar distintas, canciones, artistas, géneros y preferencias, pudiendo así almacenar al mismo tiempo todos los tipos de datos nombrados o solamente uno de ellos, por ejemplo solo almacenar artistas o solo almacenar canciones, sin necesidad de estar forzados a que siempre un nodo de la lista enlazada deba siempre tener un struct canción, artista, género y una preferencia al mismo tiempo.

Por otro lado, la cola fue implementada tomando de base una lista enlazada:

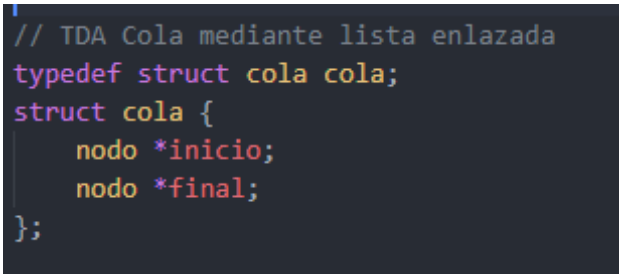


Figura 7: Cola.

Como es posible observar en la figura 7, la Cola es una lista enlazada, la cual tiene punteros apuntando al inicio y al final de la lista enlazada. También tiene sus operaciones típicas del TDA, como encolar y desencolar.

Una vez presentadas las estructuras de datos que se usaron para la creación de la solución, se proseguirá a explicar los algoritmos de cada punto de la metodología de solución presentada en el punto 2.1.2 del informe:

Empezando por el punto uno de la metodología de solución presentada en el punto 2.1.2 Leer el archivo de entrada “artistas.in”. Para eso se ha implementado la siguiente función:

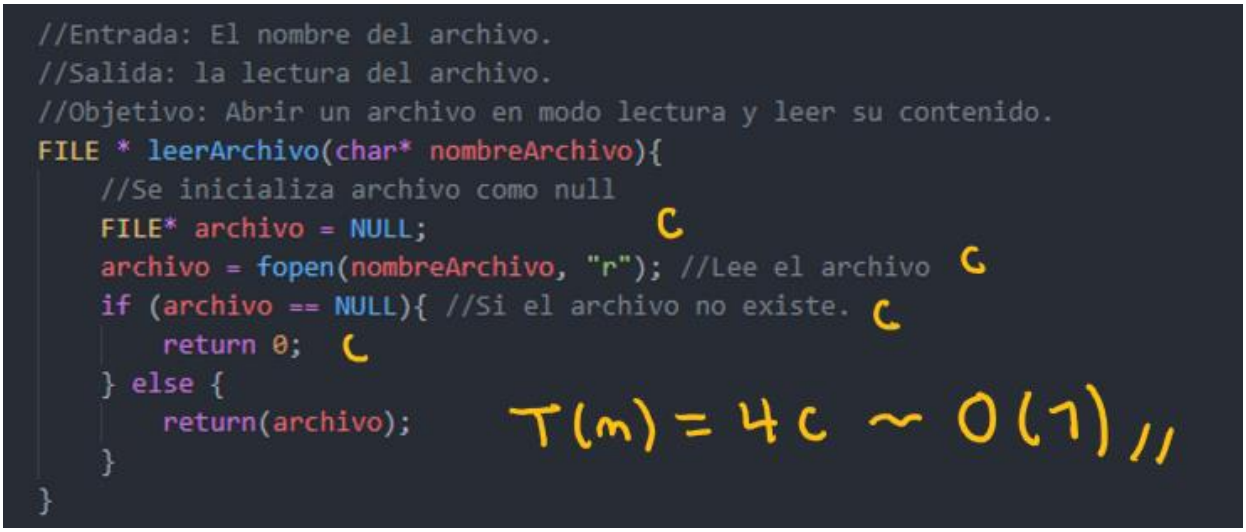


Figura 8: Función leerArchivo.

La función leerArchivo tiene como misión el abrir un archivo en modo lectura para poder leer su contenido, recibe como entrada el nombre del archivo y su salida en caso de que se haya encontrado el archivo por su nombre es el resultado de la función fopen(), en caso contrario retorna un cero.
La función tiene una complejidad O(1).
Para este punto 1) planteado en la metodología, la función será llamada con el nombre de archivo “artistas.in”.

Para el punto número dos de la metodología “Recolectar la información del archivo (artistas e id) de cada artista en una lista enlazada”, se hizo uso de la siguiente función:

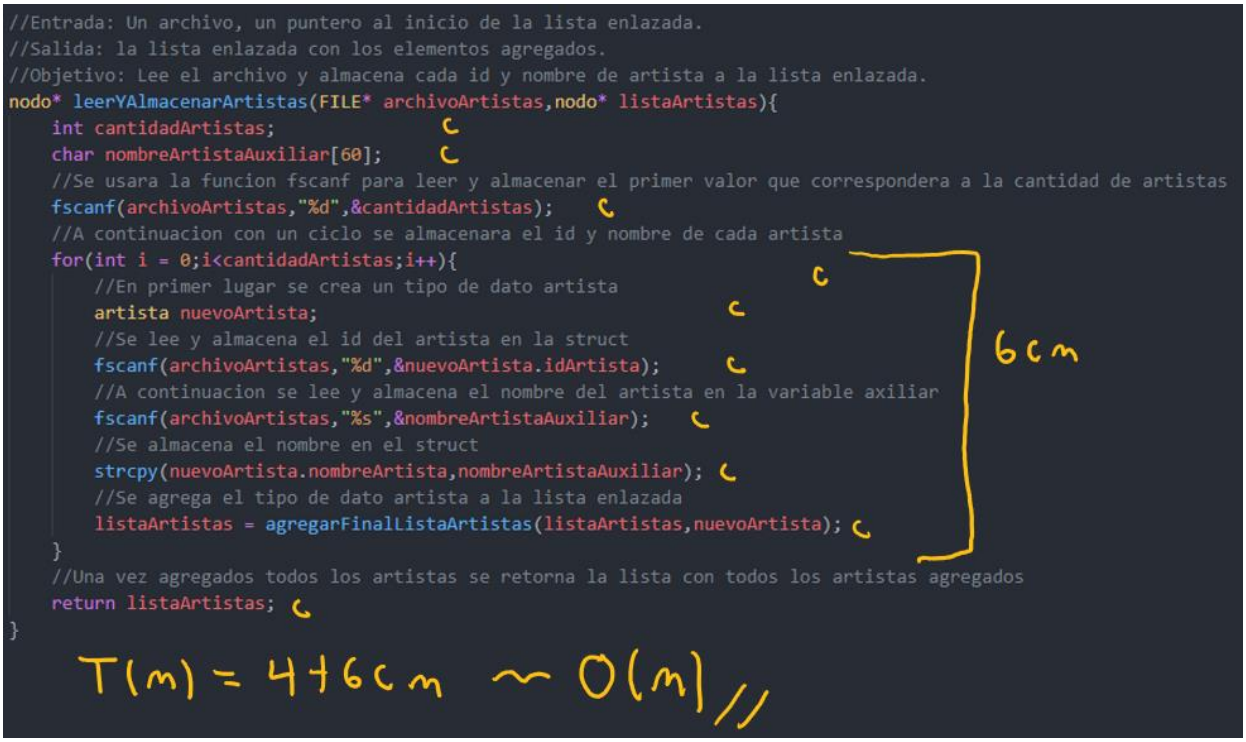


Figura 9: Función leerYAlmacenarArtistas.

La función recibe como entrada la lectura del documento de artistas de la función de la figura 8 y una lista enlazada vacía. El algoritmo recorre todo el documento haciendo uso de fscanf para leer la información dentro del archivo, en primer lugar lee la cantidad de artistas, leyendo el primer valor del documento, además se hace uso de strcpy para almacenar los nombres, cabe destacar que lo que se almacena en la lista enlazada es una instancia de la struct artista, se crea una variable llamada nuevoArtista, en la cual se guarda el id y el nombre del artista, luego esa variable se almacena al final de la lista enlazada, haciendo uso de la función agregarFinalListaArtistas, luego se pasa a leer el próximo artista del documento, así sucesivamente hasta que la variable i del ciclo for llegue a la cantidad de artistas leída inicialmente, finalmente se retorna la lista enlazada con todos los artistas encontrados, el algoritmo tiene una complejidad O(n). Dentro del main se cierra el documento “artistas.in” una vez terminada la ejecución de esta función.

Para el tercer punto de la metodología: “Leer el archivo de entrada “generos.in” ” se utilizara la misma función presentada en la figura 8 para la lectura del archivo, con entrada el archivo “generos.in”.

Para el cuarto punto: “Recolectar la información del archivo (nombre del género e id) de cada género en una lista enlazada.” Se implemento la siguiente función:

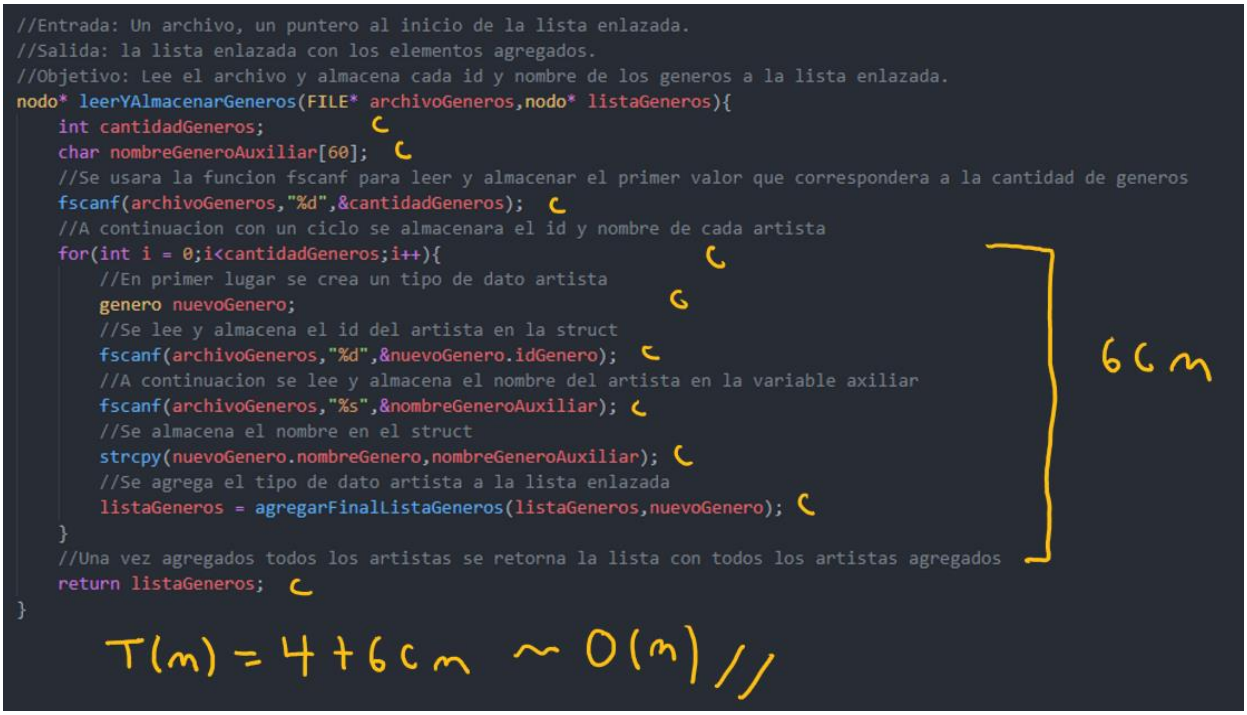


Figura 10: Función leerYAlmacenarGeneros.

La función leerYAlmacenarGeneros funciona de una manera similar a la función leerYAlmacenarArtistas presentada en la figura 9, la única diferencia que posee es que la variable que se genera pertenece a la struct genero y se invoca a la función agregarFinalListaGeneros, la cual agrega al final de la lista enlazada el nuevo genero encontrado, como entrada recibe el archivo de géneros leído anteriormente y la lista de géneros vacía, el algoritmo tiene una complejidad O(n). El archivo “generos.in” se cierra una vez en el main una vez termina la ejecución de esta función.

Para el quinto punto de la metodología: “Leer el archivo de entrada “canciones.in” ” se utilizó la misma función presentada en la figura 8 para la lectura del archivo, con entrada el archivo “canciones.in”.

Para el sexto punto de la metodología: “Recolectar la información del archivo (nombre, duración, id del artista, id del género y puntos de popularidad) de cada canción en una lista enlazada.” Se implemento la siguiente función:

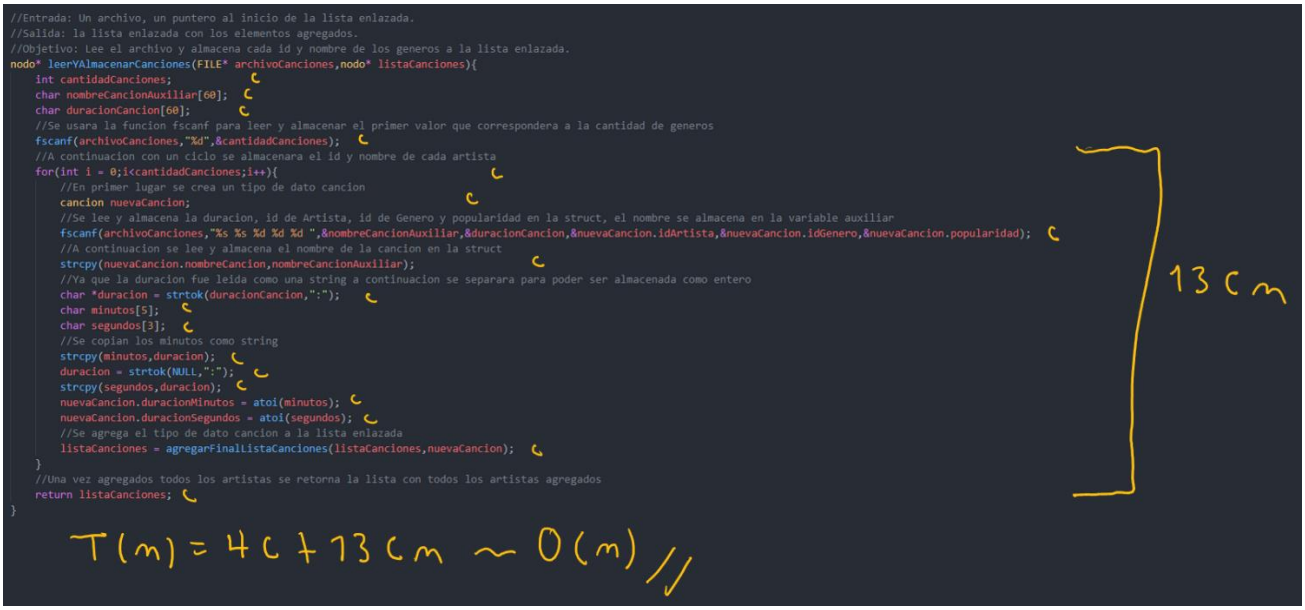
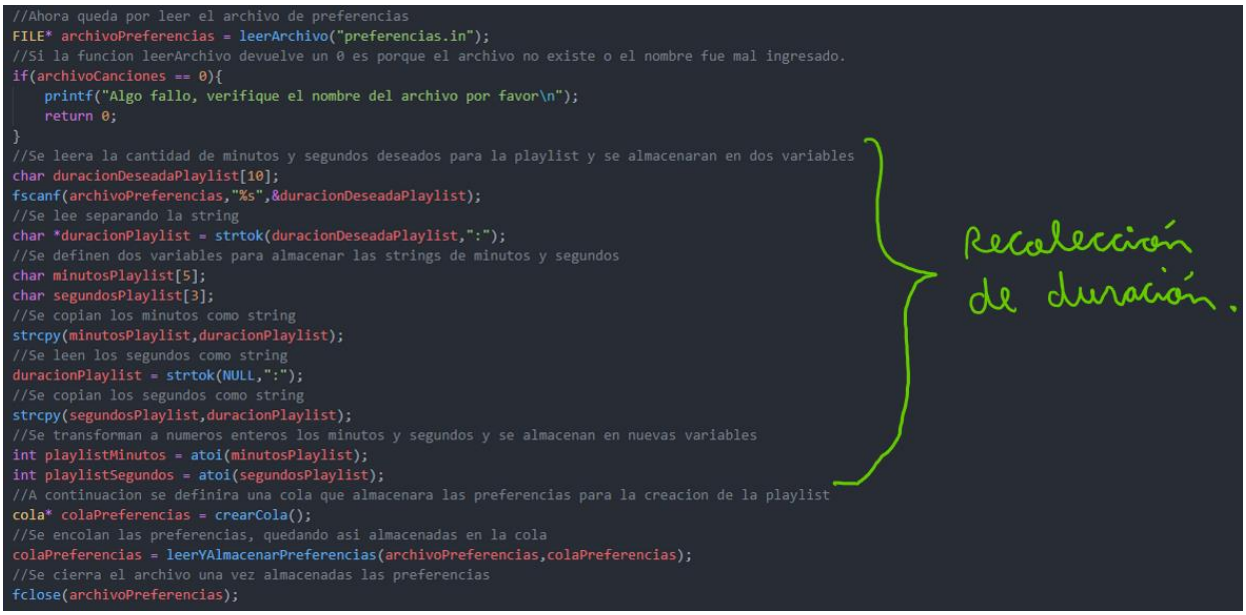


Figura 11: Función leerYAlmacenarCanciones.

La función nuevamente funciona de una manera similar a las funciones presentadas en las figuras 9 y 10: recorriendo el documento y leyendo con fscanf, sin embargo para poder leer la canción, aparte de crear la variable auxiliar del struct canción, también se hizo uso de la función strtok, para poder separar el tiempo de la canción en dos strings, una como minutos y otra como segundos, luego esas strings fueron transformadas a entero usando atoi, luego esos datos se agregan a la variable nuevaCancion y por último se agrega la canción a la lista enlazada usando la función agregarFinalListaCanciones.

Para el séptimo punto de la metodología: “Leer el archivo de entrada “preferencias.in”.” se utilizó la misma función de lectura presentada en la figura 8, con entrada el archivo “preferencias.in”.

Para el octavo punto de la metodología: “Recolectar el tiempo total de la playlist en dos variables, luego recolectar el id de los géneros en una cola (Queue).” En primer lugar, se debe recolectar el tiempo pedido de duración de la playlist, para ello se sigue el siguiente algoritmo:




```
//Ahora queda por leer el archivo de preferencias
FILE* archivoPreferencias = leerArchivo("preferencias.in");
//Si la funcion leerArchivo devuelve un 0 es porque el archivo no existe o el nombre fue mal ingresado.
if(archivoCanciones == 0){
    printf("Algo fallo, verifique el nombre del archivo por favor\n");
    return 0;
}
//Se leera la cantidad de minutos y segundos deseados para la playlist y se almacenaran en dos variables
char duracionDeseadaPlaylist[10];
fscanf(archivoPreferencias,"%s",&duracionDeseadaPlaylist);
//Se lee separando la string
char *duracionPlaylist = strtok(duracionDeseadaPlaylist,":");
//Se definen dos variables para almacenar las strings de minutos y segundos
char minutosPlaylist[5];
char segundosPlaylist[3];
//Se copian los minutos como string
strcpy(minutosPlaylist,duracionPlaylist);
//Se leen los segundos como string
duracionPlaylist = strtok(NULL,":");
//Se copian los segundos como string
strcpy(segundosPlaylist,duracionPlaylist);
//Se transforman a numeros enteros los minutos y segundos y se almacenan en nuevas variables
int playlistMinutos = atoi(minutosPlaylist);
int playlistSegundos = atoi(segundosPlaylist);
//A continuacion se definira una cola que almacenara las preferencias para la creacion de la playlist
cola* colaPreferencias = crearCola();
//Se encolan las preferencias, quedando asi almacenadas en la cola
colaPreferencias = leerYAlmacenarPreferencias(archivoPreferencias,colaPreferencias);
//Se cierra el archivo una vez almacenadas las preferencias
fclose(archivoPreferencias);
```

Figura 12: Recolección de duración de la playlist.

El algoritmo de recolección de duración es igual al implementado dentro de la función leerYAlmacenarCanciones presentado en la figura 11, sin embargo, esta fue usada dentro del main, ya que la duración de la playlist es necesaria para luego ser usada como entrada en próximas funciones. Al igual que en la función leerYAlmacenarCanciones se usa fscanf para leer la string de duración de la playlist, la cual luego se separa en minutos y segundos con strtok para finalmente transformarla y almacenarla en dos variables como enteros usando atoi.

Luego para la lectura de las preferencias, en este caso se usará una cola para almacenar los datos, esto debido a que la cola luego facilitará la creación de la combinación de la playlist. La función implementada para recolectar las preferencias es la siguiente:

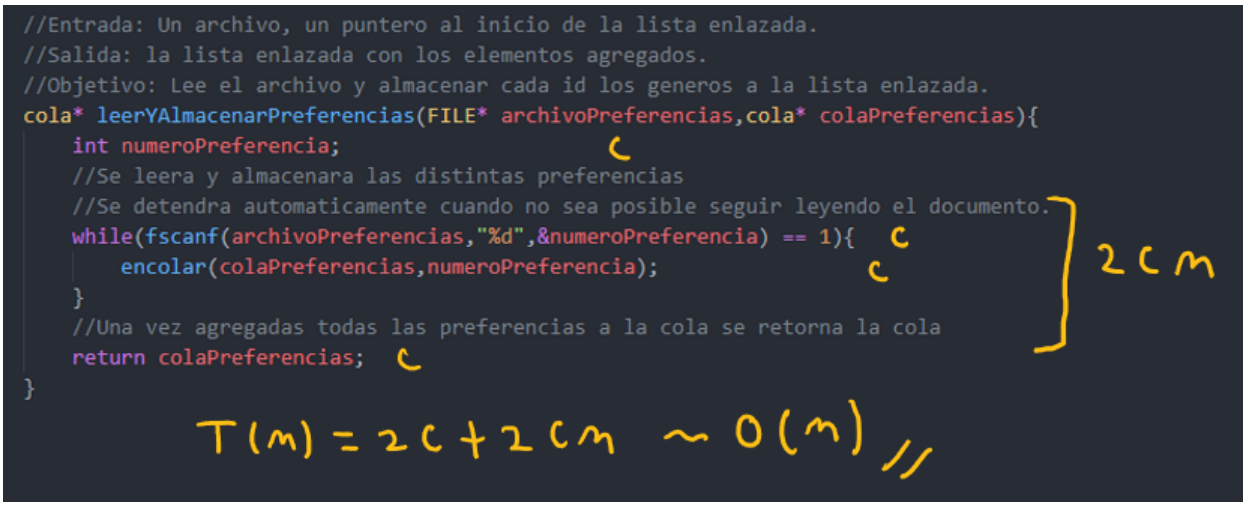


Figura 13: Función leerYAlmacenarPreferencias.

Como se puede observar en la figura 13, en este caso se recorre el documento hasta que ya no es capaz de leer datos (fscanf toma el valor de 1 cuando ya no puede leer datos), en cada ciclo encola el numero de la preferencia leída con fscanf, una vez recolectadas todas las preferencias retorna la cola de preferencias, la función tiene una complejidad de O(n). El archivo abierto se cierra una vez terminada la ejecución de esta función.

Terminado el octavo punto de la metodología ya se encuentran leídos y almacenados todos los datos de entrada, entonces ahora empieza el proceso de ordenar la biblioteca de canciones, entonces para el punto 9) “Ordenar la lista de canciones de mayor a menor respecto a la popularidad de las canciones.” Se implementó la siguiente función:

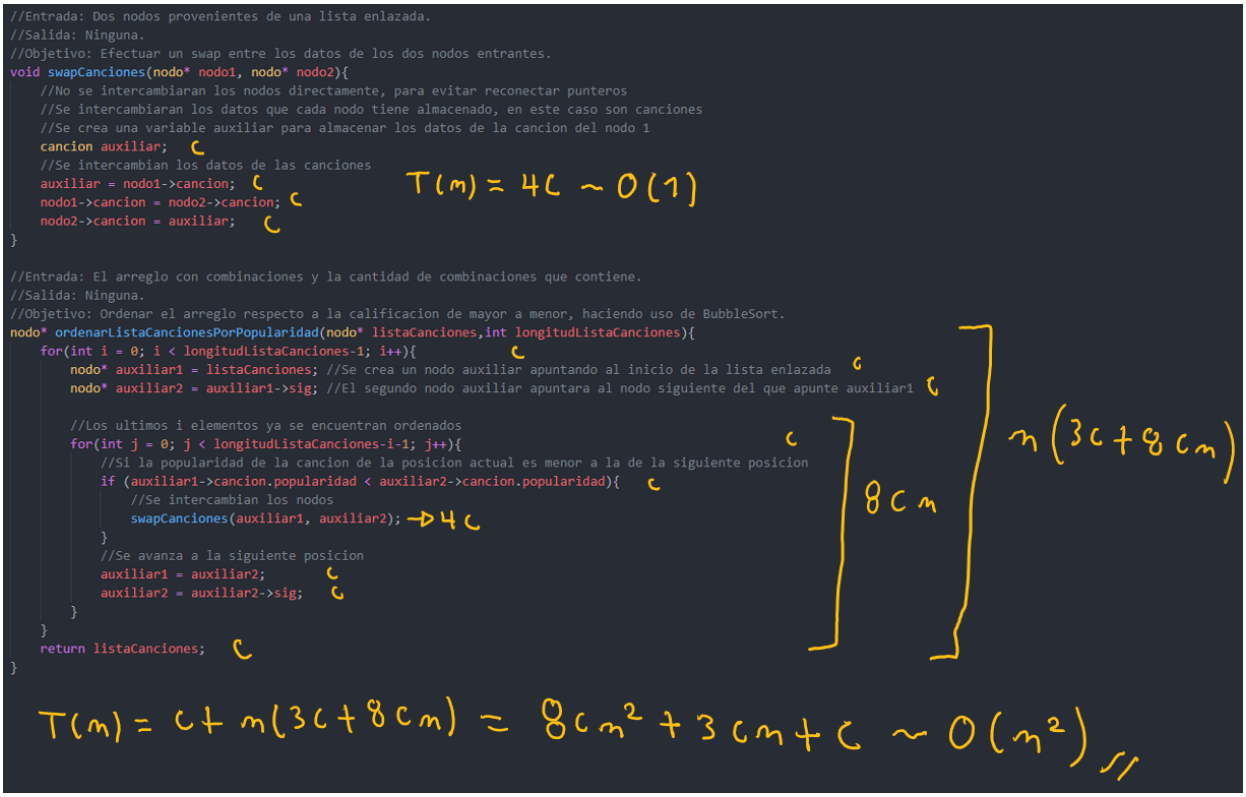


Figura 14: Función ordenarListaCancionesPorPopularidad.

La función implementada es el algoritmo de ordenamiento “Bubble sort”, sin embargo, fue modificado para que fuera capaz de funcionar con listas enlazadas y aparte ordene de mayor a menor fijándose en la popularidad de las canciones, en los casos que es necesario hacer swap, el intercambio no se hace cambiando los nodos de posición, si no que se intercambian los datos que contienen los nodos, en este caso se intercambian las canciones dentro de los nodos. La entrada es la lista de canciones junto a la longitud de esta misma, el retorno es la lista de canciones ordenada de mayor a menor según la popularidad de las canciones.

Para el décimo punto de la metodología: “10) Volver a ordenar la lista de canciones ya ordenada, ahora respecto a orden alfabético del autor en casos donde dos o más canciones tengan la misma popularidad y también orden por duración de canción, en casos donde coincida la popularidad y autor de distintas canciones, quedando la canción con menos duración antes que la canción con más duración.” Se implementó la siguiente función:

Figura 15: Función ordenarListaCancionesMismaPopularidad.

Esta función también es un Bubble sort modificado, recibe como entrada la lista de canciones ya ordenada por popularidad, la longitud de la lista de canciones y la lista enlazada que contiene a los artistas, lo primero que hace la función es verificar si la canción de la posición actual tiene la misma popularidad que la canción de la posición siguiente de la lista, en caso de no tener la misma popularidad no hace nada y avanza a la siguiente posición. En caso de tener la misma popularidad se verifica si tienen el mismo artista, en caso de tenerlo se verifica si es necesario aplicar el swap de canciones de acuerdo al tiempo si no están en orden, en caso de no tener al mismo artista de autor se consigue el nombre de los artistas por su id, usando la función encontrarArtistaPorId, luego se verifica si se encuentran ordenados alfabéticamente usando strcmp, si no están ordenados alfabéticamente se ejecuta el swap, si no se deja tal cual como esta. Este ciclo se ejecuta hasta que se recorra toda la lista de canciones y tiene una complejidad O(n^3) en su peor caso.

Para el onceavo punto de la metodología: “Creación de archivo de salida “salida.out” la cual es la biblioteca de canciones ordenada.” Se implementó la siguiente función:

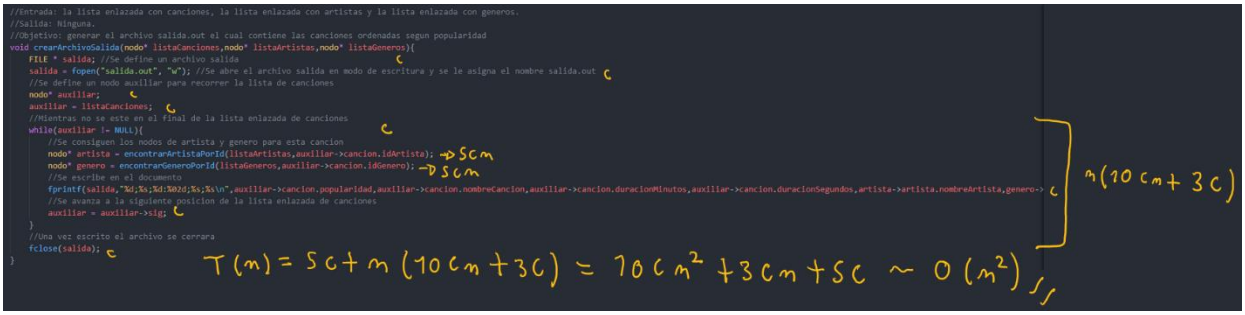


Figura 16: Función crearArchivoSalida.

La función recibe como entradas la lista de canciones, la lista de artistas y la lista de géneros, en primer lugar abre un nuevo archivo en modo de escritura con el nombre “salida.out”, lo cual crea el archivo de texto, luego mediante un ciclo va leyendo posición por posición la lista de canciones que ya se encuentra ordenada, busca el artista y el género de la canción actual por sus id y luego escribe en el archivo usando fprintf toda la canción con el formato requerido, el cual se puede ver en la figura 1, para escribir los segundos se usa %02d, ya que esto le agrega un 0 al segundo en caso de que los segundos de duración de la canción sean menores a 10, o sea sean de un digito, una vez recorrida toda la lista de canciones y escritas todas las canciones al documento, se cierra el documento creado. La función tiene un O(n^2).

Para el doceavo punto de la metodología: “Creación de playlist en una lista enlazada, según requisitos pedidos en el archivo “preferencias.in”, se hace en base a la lista de canciones ordenada considerando los requisitos pedidos.” Se implementó la siguiente función:

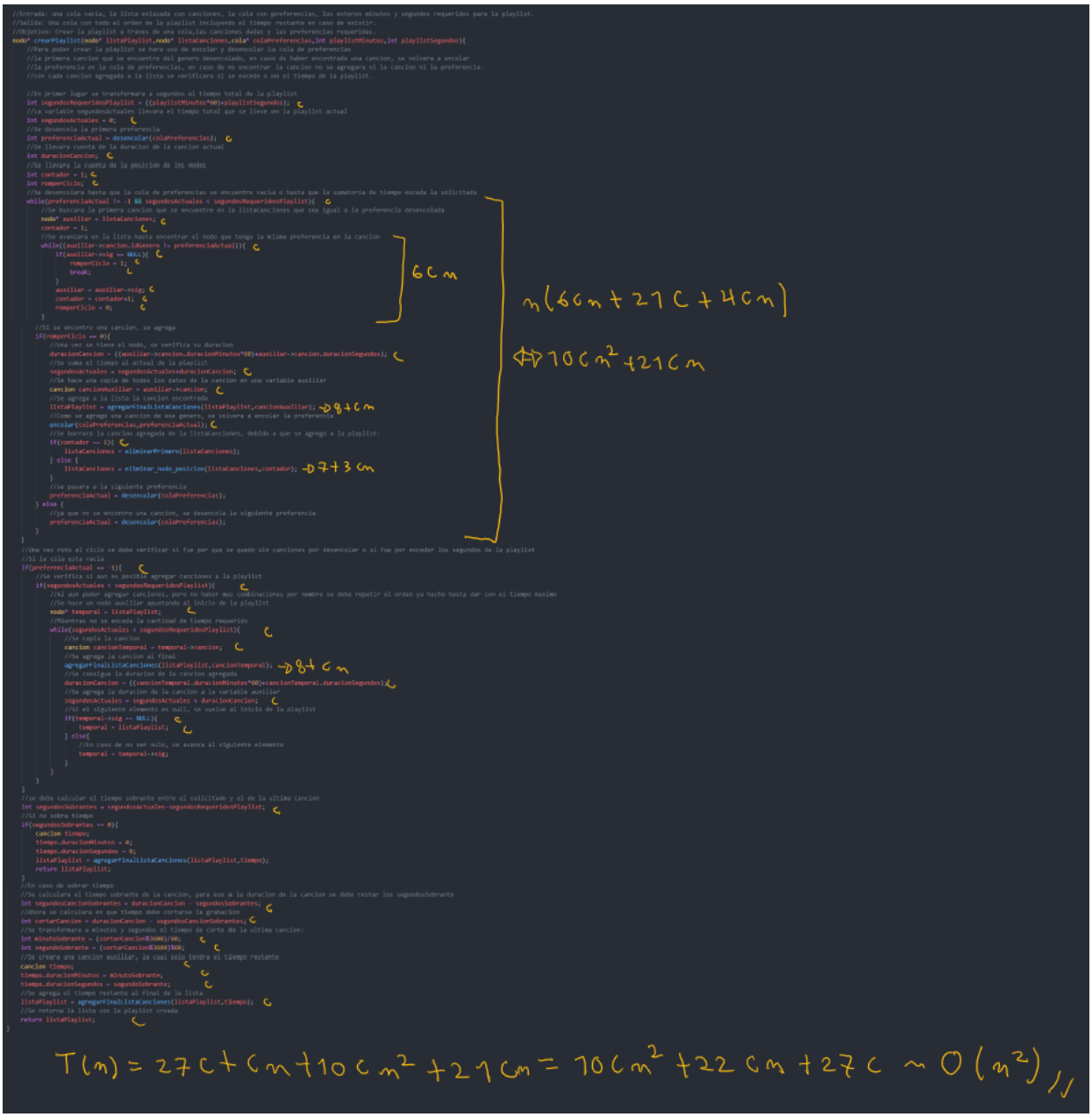


Figura 17: Función crearPlaylist.

La función crearPlaylist funciona con la siguiente lógica para generar la playlist:

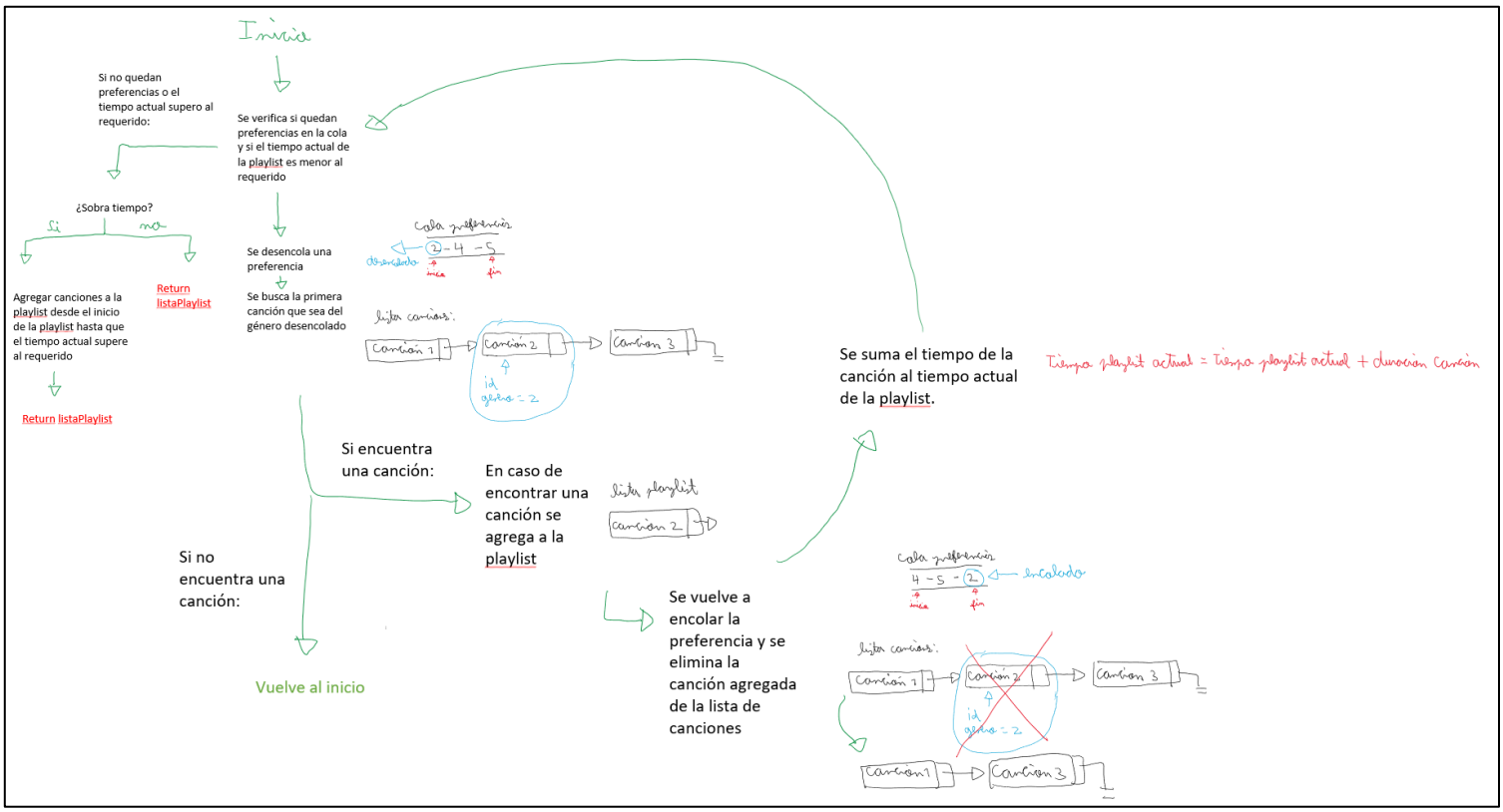


Figura 18: Lógica función crearPlaylist

Como se puede ver en el diagrama de la figura 18, la función va desencolando preferencias, desencola una preferencia, busca la primera canción con el mismo id de género que la preferencia en la lista de canciones que se encuentra ordenada anteriormente, si encuentra una la agrega a la lista de la playlist y la borra de la lista de canciones, suma el tiempo de la canción agregada al tiempo actual de la playlist, vuelve a encolar la preferencia y repite el ciclo, el cual se rompe cuando el tiempo de la playlist supera o iguala al tiempo requerido o ya no quedan preferencias para desencolar en la cola de preferencias, lo cual puede ocurrir debido a que si no se encuentran canciones para el id de preferencia desencolado, no se vuelve a encolar ese id de preferencia. Una vez roto el ciclo, en caso de que aun el tiempo actual de la playlist no sea igual o mayor al requerido inicialmente, se empezaran a agregar canciones de inicio a fin de la lista de la playlist en la misma lista de la playlist, hasta que se cumpla esa condición, en caso de que el tiempo actual sea mayor o igual, se retorna la lista enlazada con la playlist generada. La función recibe como entradas una listaPlaylist vacía, la lista ordenada de canciones, la cola de preferencias, los minutos y segundos requeridos para la playlist, retorna la lista playlist con toda la playlist generada, teniendo en su última posición el tiempo en el cual se debe cortar la ultima canción en caso de que sea necesario. El algoritmo tiene una complejidad de O(n^2).

Para el treceavo punto de la metodología: “Creación de archivo de salida “playlist.out” la cual contiene la playlist creada según los requisitos pedidos.” Se implementó la siguiente función:

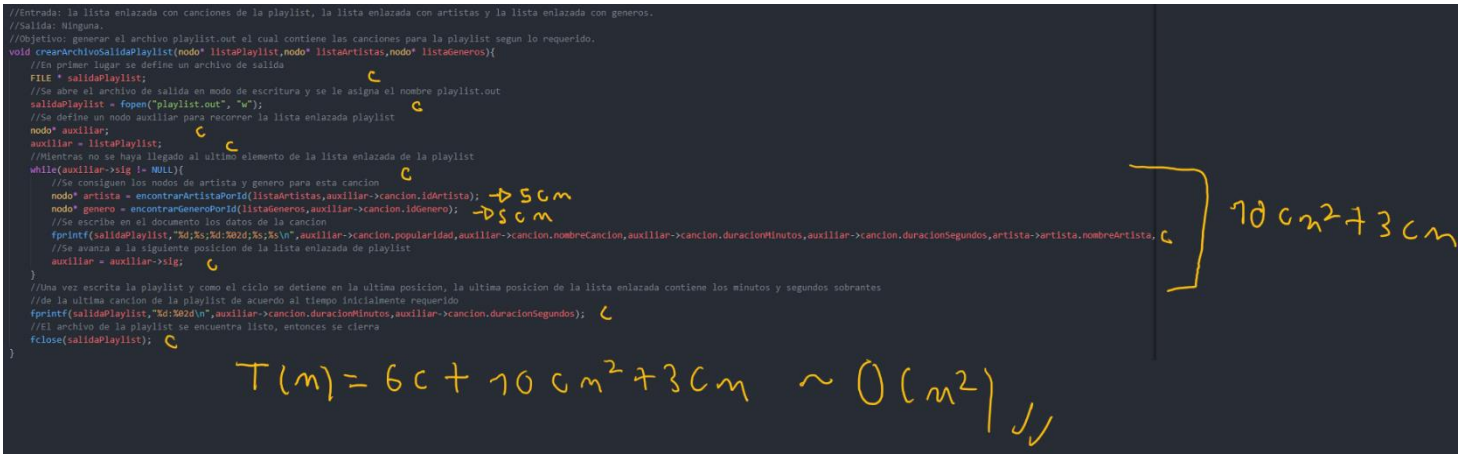


Figura 19: Función crearArchivoSalidaPlaylist.

El algoritmo recibe como entrada la listaPlaylist, la lista de artistas y la lista de géneros. En un inicio abre un archivo en el formato “write” con el nombre “playlist.out”, luego recorre la listaPlaylist de principio hasta el último elemento, escribiendo cada canción en el archivo creado, siguiendo el formato pedido el cual se puede ver en la figura 2, una vez se termina el ciclo el auxiliar queda apuntando al ultimo elemento de la lista playlist, el cual contiene el tiempo en el cual se debe “cortar” la ultima canción ingresada, se escribe el tiempo en el archivo y se cierra el mismo, terminando así la función, la función tiene una complejidad O(n^2) y funciona de manera muy similar a la función crearArchivoSalida de la figura 16, con la diferencia que aquí también se almacena el tiempo de corte final.

Para el punto catorceavo y final de la metodología propuesta inicialmente: “Liberación de memoria de las listas enlazadas y cola ocupadas.” Todo lo requerido ya esta cumplido hasta este punto, sin embargo, falta liberar la memoria ocupada por las listas enlazadas y la cola usadas en la creación de la solución, para ello se implementaron dos funciones:

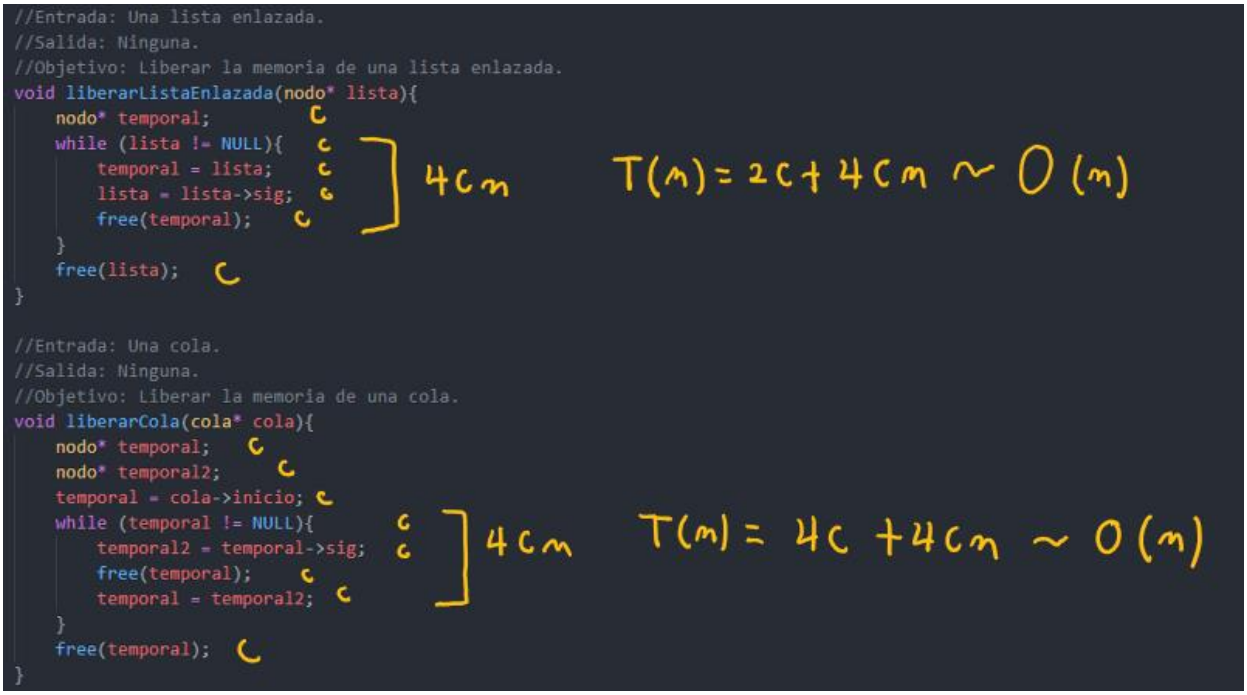


Figura 20: Funciones liberarListaEnlazada y liberarCola.

En ambas funciones se hace el siguiente procedimiento: Se recorre de principio a fin la lista/cola liberando cada elemento mientras se recorre.

Con esto se terminaría toda la subdivisión de problemas propuesta en el punto 2.2.1 con la cual se resolvió el problema propuesto en este laboratorio.

2.2 Análisis de los resultados

En distintos casos probados el programa funciona de buena manera y cumple con generar ambos archivos de salida especificados, a continuación, un ejemplo de las salidas generadas con los archivos de entrada entregados:

```
18;another_one_bites_the_dust;3:34;queen;rock
18;radio_ga_ga;5:48;queen;rock
17;blinding_lights;3:20;the_weeknd;r&b
15;in_your_eyes;3:57;the_weeknd;r&b
14;lovesick_girls;3:12;blackpink;k-pop
13;i_feel_it_coming;4:29;the_weeknd;r&b
12;playing_with_fire;3:17;blackpink;k-pop
10;sit_next_to_me;4:03;foster_the_people;alternativa
9;stay;3:50;blackpink;k-pop
7;houdini;3:22;foster_the_people;alternativa
```

Figura 21: Archivo de salida generado con el programa “salida.out”.

```
17;blinding_lights;3:20;the_weeknd;r&b
14;lovesick_girls;3:12;blackpink;k-pop
10;sit_next_to_me;4:03;foster_the_people;alternativa
15;in_your_eyes;3:57;the_weeknd;r&b
12;playing_with_fire;3:17;blackpink;k-pop
7;houdini;3:22;foster_the_people;alternativa
13;i_feel_it_coming;4:29;the_weeknd;r&b
9;stay;3:50;blackpink;k-pop
17;blinding_lights;3:20;the_weeknd;r&b
14;lovesick_girls;3:12;blackpink;k-pop
10;sit_next_to_me;4:03;foster_the_people;alternativa
15;in_your_eyes;3:57;the_weeknd;r&b
12;playing_with_fire;3:17;blackpink;k-pop
2:19
```

Figura 22: Archivo de salida generado con el programa “playlist.out”.

Respecto a los tiempos de ejecución, tomando en cuenta todos los algoritmos y funciones presentadas en la sección 2.1.3 el programa en su completitud tendría una complejidad $O(n^3)$, que es la complejidad más alta de todas las funciones implementadas, la cual definitivamente podría ser más eficiente y mejorarse, ya sea con el refinamiento de los mismos algoritmos implementados en los cuales se alcanzó esa complejidad, o el uso de distintos algoritmos como un quickSort en vez de un bubbleSort o el uso de otras estructuras de datos. Considerando la complejidad del programa, considero esa la principal falencia de la solución implementada, debido a los altos tiempos de ejecución que podrían presentarse con grandes cantidades de canciones de entrada.

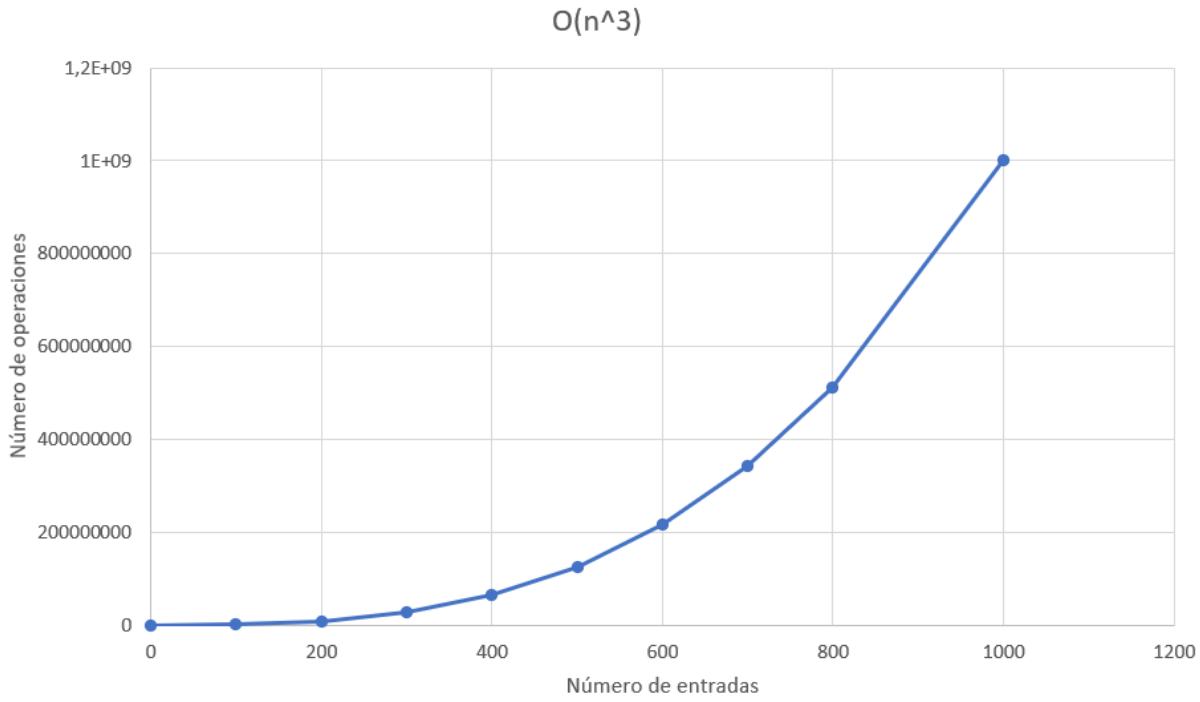


Figura 23: Gráfico de medición de tiempo por complejidad.

La figura 23 muestra el gráfico para un algoritmo de complejidad $O(n^3)$ según sus números de entrada, a continuación se mostrará el gráfico de medición de tiempo real de programa, la cual fue medida usando la librería time.h y las funciones: start = clock(); y end = clock(); al programa se le irán insertando distintas cantidades de canciones y se graficara su tiempo real de ejecución:

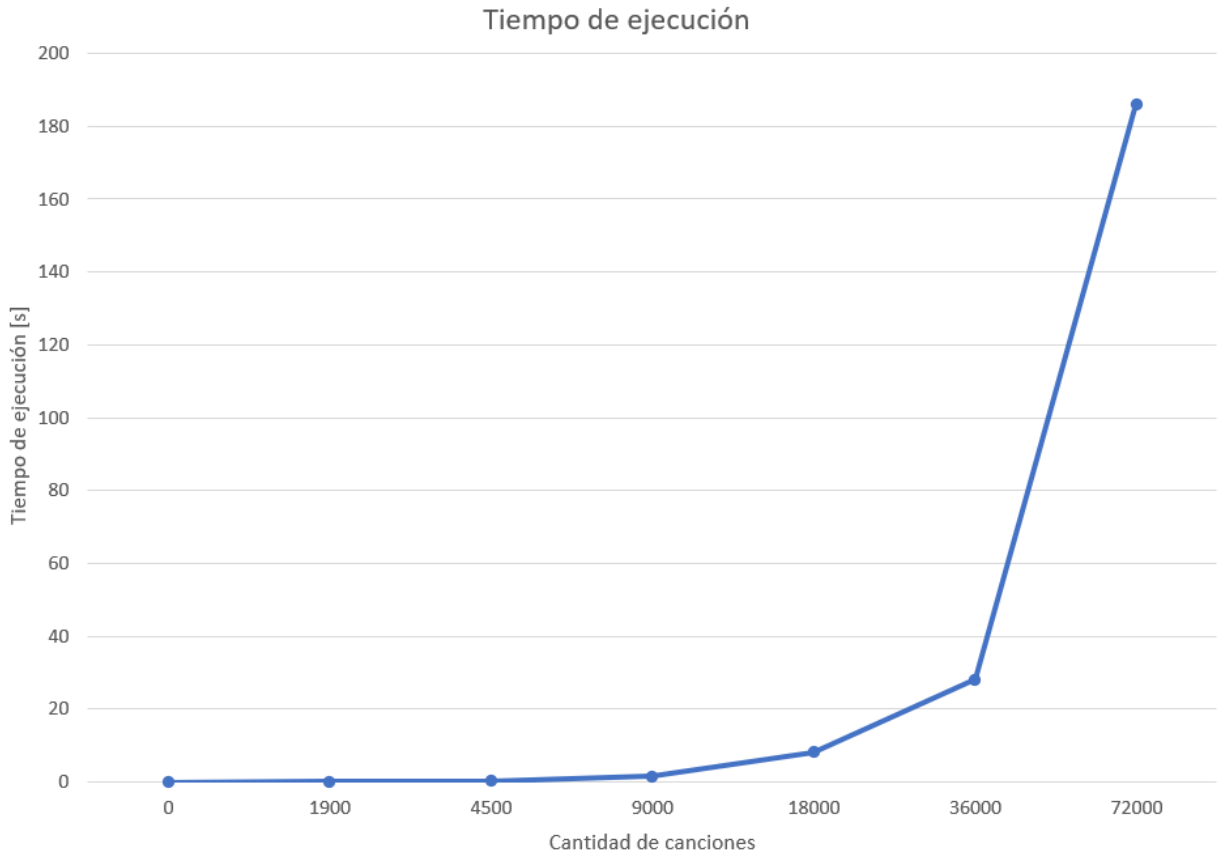


Figura 24: Gráfico de mediciones de tiempo.

El tiempo es respecto al tiempo en el cual la CPU del equipo estuvo ocupado ejecutando el programa, por ende, este tiempo puede ser mayor o menor dependiendo de las características de cada equipo.

Haciendo un análisis del gráfico de la figura 24, el cual presenta medición de tiempo real para distintas entradas, desde 0 hasta 72.000 canciones insertadas, es posible determinar que efectivamente existe una fuerte correlación del tiempo de ejecución real con el tiempo de ejecución teórico calculado y como se había inferido anteriormente, el algoritmo es poco efectivo mientras más cantidad de datos deba analizar.

2.3 Conclusión

Para concluir, se logró a cabalidad el objetivo principal de este laboratorio, que era el crear una solución que en primer lugar organizará la biblioteca de canciones de acuerdo con su puntaje de popularidad y luego generará una playlist de esas canciones de acuerdo con las preferencias ingresadas y el tiempo que se deseaba durará dicha playlist. Sin embargo, la solución creada presenta el problema que se aprecia en los gráficos de la figura 23 y 24 del punto 2.2 del informe: es ineficiente para cantidades de datos grandes, por ende, si se intentara generar una playlist de por ejemplo 72.000 personas, que es la última medición realizada en el grafico de la figura 24, el algoritmo tomaría una enorme cantidad de tiempo en lograr ordenar la biblioteca y generar la playlist. La principal mejora que se le puede hacer al programa creado es el de optimizar mejor los algoritmos usados para poder trabajar con grandes cantidades de datos sin que el orden del algoritmo sea de $O(n^3)$ y por tanto tenga altos tiempos de ejecución.

Para finalizar, esta experiencia de trabajo de laboratorio sirvió para darse cuenta de la importancia de los distintos algoritmos y estructura de datos, debido a que al implementar la solución usando listas enlazadas simples y colas en combinación de structs para los distintos tipos de datos, fue posible darse cuenta el como facilitan mucho el trabajo para la generación de distintos algoritmos, que serían mucho más difíciles implementarlos sin el uso de las estructuras de datos usadas para el desarrollo de esta solución.

2.4 Referencias

[1] Sena M. (2019). *Estructuras de datos.* (Recuperado 10/11/2020). https://medium.com/techwomenc/estructuras-de-datos-a29062de5483