Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Estructura de datos y análisis de algoritmos

Laboratorio 2 - SpotMusic

Christofer Rodríguez

Profesor: Mario Inostroza

Ayudante: Estaban Silva

Santiago – Chile
2-2020

Tabla de contenidos

Índice de figuras

Índice de tablas

CAPÍTULO 1. INTRODUCCIÓN

La música de fondo es un elemento fundamental dentro del mundo de las transmisiones en vivo, ya que esta influye directamente en la cantidad de personas que ven la transmisión. Por esto se debe construir un programa en el lenguaje de programación C que, a partir de archivos de entrada que contienen información sobre una biblioteca musical, genere un archivo de salida con las canciones ordenadas por los puntos de popularidad de la canción, el orden alfabético de sus artistas y por la duración de la canción. Además, se debe generar una playlist de canciones que cumplan con las preferencias de genero y duración de la streamer, estas preferencias son entregadas mediante otro archivo de entrada.

Los objetivos de este informe son explicar la implementación de dicho programa, analizar la complejidad tanto del programa en general como de las funciones que lo componen, para finalmente se analizar el alcance y limitaciones de la solución empleada.

La manera de llevar a cabo estos objetivos será, primeramente, explicar los conceptos fundamentales para comprender este documento en el marco teórico, luego se explicará y analizará paso a paso tanto las estructuras, como los algoritmos utilizados para la implementación del programa en la descripción de la solución, además se estudiará su complejidad algorítmica, lo que nos permitirá llegar a expresiones matemáticas que ayudarán a entender su comportamiento. Para finalizar, en el análisis de resultados se examinarán las falencias, limitaciones, ventajas, la eficiencia algorítmica y se comparara el tiempo teórico con el tiempo real de ejecución del programa creado.

CAPITULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

2.1.1 Marco teórico

- **Algoritmo:** Procedimiento que mediante una serie de pasos transforma una entrada o un conjunto de estas en una salida.

- **Eficiencia algorítmica:** Análisis de la cantidad de recursos utilizados por un determinado algoritmo, estos recursos pueden ser, por ejemplo: tiempo o memoria.

- **Struct:** Declaración de estructura en la cual se especifica una secuencia de variables que pueden ser de distinto tipo, posteriormente sirve como plantilla para la creación de esta estructura.

- **Puntero:** Tipo de dato cuyo valor es una dirección de memoria.

- **Lista simplemente enlazada:** Secuencia de nodos conectados de manera lineal con elementos homogéneos (mismo tipo de dato) de largo indeterminada. Recibe el nombre de simplemente enlazada, ya que cada nodo solamente conoce al elemento que le sigue, no a su elemento anterior.

- **Nodo:** Estructura que contiene una parte de datos, estos pueden ser más de uno y de diferentes tipos, mientras que la otra parte es un puntero que, en este caso, solamente apunta al siguiente elemento de la lista.
- **Arreglo:** Variables estructuradas, donde cada elemento se almacena de forma consecutiva dentro de la memoria, estos pueden ser de diferentes tipos de datos.

- **String:** Arreglo que contiene un carácter en cada una de sus posiciones, corresponde a la representación de una cadena de texto con largo definido.

2.1.2 Algoritmos y estructura de datos

Para la solución de este problema se decidió utilizar una lista simplemente enlazada para almacenar toda la información correspondiente a cada canción de la biblioteca y una lista simplemente enlazada para guardas los identificadores numéricos de los géneros preferidos, para luego con estos identificadores seleccionar las canciones que serán parte de la playlist.

A continuación, se enumerará de manera breve las etapas que sigue el programa para encontrar la solución del problema, cada una de estas serán explicada de manera más detallada a lo largo de esta sección:

1. Leer la información contenida en un archivo de entrada con la información inicial de las canciones.
2. Extraer la información y almacenarla en su respectiva estructura representativa.
3. Generar una lista simplemente enlazada con las canciones.
4. Leer la información contenida en el archivo de entrada con la información de los artistas.
5. Se actualiza la información de las canciones con sus respectivos artistas.
6. Leer la información del archivo de entrada que tiene la información de los géneros musicales.
7. Actualizar la información de las canciones con sus respectivos géneros musicales.
8. Generar una nueva lista con las canciones ordenadas según su puntaje de popularidad.
9. Crear un archivo de salida con las canciones ordenadas.
10. Leer el archivo de entrada que contiene las preferencias de la playlist.
11. Crear una lista simplemente enlazada con los identificadores de los géneros preferidos.
12. Generar un archivo de salida con las canciones que compondrán las playlist.

Cabe señalar que los puntos 1, 2 y 3, se realizan a la vez, mientras se lee la información, se generan las estructuras de datos; así mismo los puntos 4 y 5, 6 y 7, 10 y 11 se realizan a la vez con su respectivo par, mientras se lee la información del archivo, se actualiza la información de cada canción.

Tanto para la representación de los postulantes como las soluciones, se utilizó el tipo de dato **struct**, esta representación fue definida por el nombre **Postulante** y está compuesta por tres elementos:

Para la solución de este problema, se utilizaron dos estructuras de datos del tipo **struct**, una de ellas fue definida como **nodo** y la otra como **nodoDato**.

La primera de estas es la estructura **nodo**, esta estructura tiene como función representar una canción perteneciente a la biblioteca musical. Esta contiene toda la información relacionada a una canción, siendo estas: un **int** con sus puntos de popularidad, **string** para su nombre, un **int** con su duración en segundos, el identificador numérico de su artista, un **string** para el nombre de su artista, el identificador numérico de su género, **string** para el género y un puntero que apunta al siguiente elemento de la lista simplemente enlazada respectivamente.

```
typedef struct nodo{
    int puntaje;
    char nombre[50];
    int duracion;
    int idArtista;
    char artista[25];
    int idGenero;
    char genero[25];
    struct nodo *sig;
} nodo;
```

Ilustración 1: Estructura del tipo de dato nodo

La segunda estructura es el **nodoDato**, esta estructura será utilizada para almacenar el identificador numérico de los géneros musicales preferidos para generar la playlist. Esta esta compuesta solamente de dos elementos, el primero de ellos en una variable del tipo **int** para guardar dicho identificador y un puntero que apunta al siguiente elemento de la lista simplemente enlazada.

```
typedef struct nodoDato{
    int dato;
    struct nodoDato *sig;
} nodoDato;
```

Ilustración 2: Estructura del tipo nodoDato

Antes de comenzar a explicar el funcionamiento del programa, es necesario explicar las funciones básicas que operan sobre las estructuras de dato, de esta manera la explicación del programa será más fluida.

Crear una estructura de dato nodo: Mediante la función **crearNodo**, se crea una representación de una canción, esta función recibe como argumentos el puntaje de la canción, su nombre, su duración, el identificador de su artista, el nombre de su artista, el identificador de su genero y el nombre del genero musical. Primero se crea un **nodo** y se asigna memoria para dicho nodo, de esta manera esta estructura seguirá existiendo fuera de la función, luego

se asigna cada valor en sus respectivas variables y este nodo apuntará a **NULL**, ya que será el último elemento de la lista, finalmente se retornará el **nodo** creado.

```
nodo *crearNodo(int puntaje, char *nombre, int duracion, int idArtista, char *artista, int idGenero, char *genero){
    nodo *aux;
    aux = (nodo*)malloc(sizeof(nodo));
    aux->puntaje = puntaje;
    strcpy(aux->nombre, nombre);
    aux->duracion = duracion;
    aux->idArtista = idArtista;
    strcpy(aux->artista, artista);
    aux->idGenero = idGenero;
    strcpy(aux->genero, genero);
    aux->sig = NULL;

    return aux;
}
```

Ilustración 3: Función crearNodo

Crear una estructura de tipo **nodo** tiene un $T(n) = 11C$, con un $O(C)$.

Crear una estructura del tipo nodoDato: La función **crearNodoDato** se encarga de crear una estructura del tipo **nodoDato**. Esta recibe como argumento solamente un **int** correspondiente al valor que se desea almacenar en nodo, primero se crea una variable del tipo **nodoDato** y se asigna memoria para este, se almacena el valor entregado, se apunta a **NULL** como siguiente elemento y se retorna el nodo creado.

```
nodoDato *crearNodoDato(int dato){
    nodoDato *aux;
    aux = (nodoDato*)malloc(sizeof(nodoDato));
    aux->dato = dato;
    aux->sig = NULL;

    return aux;
}
```

Ilustración 4: Función creaNodoDato

Crear una estructura del tipo **nodoDato** posee un $T(n) = 5C$ y un orden $O(C)$.

Agregar un nodo al final de una lista simplemente enlazada de nodos: La función **agregarFinalNodo** es la encargada de agregar un **nodo** al final de una lista simplemente enlazada de nodos. Se comienza comprobando si la lista se encuentra vacía, si este es el caso, la variable **L** apuntará al nodo agregado como su primer elemento y será retornado. Si la lista no se encuentra vacía, se crea una variable auxiliar del tipo **nodo** para recorrer la lista hasta llegar al final, una vez se llega al final, se asigna la dirección de memoria del nodo a agregar como el siguiente elemento del nodo final. Una vez hecho esto, la lista es retornada.

```
nodo *agregarFinalNodo(nodo *L, nodo *nodoAgregar){

    //En caso de que la lista este vacia
    if(L == NULL){
        L = nodoAgregar;
        return L;
    }

    nodo *aux = L;
    //Caso general
    while(aux->sig != NULL){
        aux = aux->sig;
    }

    aux->sig = nodoAgregar;

    return L;
}
```

Ilustración 5: Función agregarFinalNodo

La función **agregarFinalNodo** posee un $T(n) = 4C + nC$ y un $O(n)$.

Agregar un nodoDato al final de una lista simplemente enlazada: Al igual que la función **agregarFinalNodo** la función **agregarFinalDato** se encargar de agregar un nodo al final de una lista simplemente enlazada. De la misma manera que la función anterior, en caso de que la lista se encuentre vacía, el nodo será agregado como el primer elemento de esta lista, en cambio si la lista no se encuentra vacía, esta será recorrida con ayuda de una variable auxiliar hasta llegar al final y se agregará el nodo al final de esta.

```
nodoDato *agregarFinalDato(nodoDato *L, nodoDato *nodoAgregar){
    nodoDato *aux = L;

    if(aux == NULL){
        L = nodoAgregar;
        return L;
    }

    while(aux->sig != NULL){
        aux = aux->sig;
    }

    aux->sig = nodoAgregar;

    return L;
}
```

Ilustración 6: Función agregarFinalDato

Agregar un **nodoDato** al final de una lista posee un $T(n) = 3C + nC$ y un $O(n)$.

Debido a que la duración de cada canción y la duración máxima de la playlist es entregada en forma de **string** con el formato de "minutos: segundos", se crearon dos algoritmos para transformar dicho **string** a un entero que contiene la duración en segundos y otra que convierte el entero a su representación en **string**, esto con el objetivo de trabajar de manera más fácil con el tiempo.

El algoritmo **tiempoSegundos** realiza la conversión de **string** a **int**, recibe como argumento el **string** con la duración de la canción, luego calcula el largo del **string**, de esta manera se sabe hasta que índice del **string** se debe acceder y se inicializan variables de tipo entero para almacenar la información leída. Se recorre el **string** con un ciclo **while** mientras no llegue al carácter que separa los minutos de los segundos, en cada ciclo transforma uno de los caracteres a entro y lo suma con la cantidad anterior de minutos. Lugo se aumenta en uno el índice del **string** para no leer el carácter separador y mediante otro ciclo **while** se recorre el resto del **string** transformando un carácter en entero y luego guardándolo en la variable de segundos. Finalmente se transforman los minutos a segundo y se retorna la suma total de los minutos más segundos.

```
int tiempoSegundos(char* tiempo){
    int largo = strlen(tiempo);
    int minutos = 0;
    int segundos = 0;
    int i = 0;

    //Se recorre la parte del string que contiene los minutos
    while (tiempo[i] != ':')
    {
        //Se transforma el caracter a un numero entero
        int aux = tiempo[i] - 48;
        minutos = (minutos*10) + aux;
        i += 1;
    }
    //Se suma uno para no leer el caracter ':'
    i += 1;

    //Se lee el resto del string que corresponde a los segundos
    while (i < largo)
    {
        //Se transforma el caracter en un entero
        int aux = tiempo[i] - 48;
        segundos = (segundos*10) + aux;
        i += 1;
    }
    //Se transforman los minutos en segundos y se suma con el resto de los segundos
    segundos = (minutos*60) + segundos;

    return segundos;
}
```

Ilustración 7: Función tiempoSegundos

Si bien el anterior algoritmo tiene un $O(n)$, dentro del contexto del problema, donde solamente se acepta como tiempo máximo 59:59 minutos, el algoritmo tiene un $T(n) = 20C$ y un $O(C)$.

El algoritmo que realiza la acción contraria es la función **segundosTiempo**, la cual, a partir de un **int** con la duración en segundos, genera un **string** con la estructura definida para la duración. Para comenzar se asigna memoria para el arreglo de caracteres y un **string** auxiliar para almacenar los segundos, luego a partir de la duración en segundos se calculan los minutos y los segundos de duración; una vez calculados son almacenados en **strings** diferentes, para finalmente retornar un **string** con la unión de ambos.

```
char* segundosTiempo(int tiempo){
    char * duracionString = malloc(8);
    char segundosChar[2];

    //Se calculan la cantidad de minutos y segundos
    int minutos = tiempo/60;
    int segundos = tiempo%60;

    //Se almacenan los minutos dentro del string de duracion
    sprintf(duracionString, "%d", minutos);
    //Se agrega el separador de minutos-segundos
    strcat(duracionString, ":");

    //En caso de que los segundos sean menor a 10, se agrega un cero delante de ellos
    if(segundos < 10){
        strcat(duracionString, "0");
    }

    //Se convierten los segundos a string y se agregan al string de la duracion
    sprintf(segundosChar, "%d", segundos);
    strcat(duracionString, segundosChar);

    //Se retorna la duracion como string
    return duracionString;
}
```

Ilustración 8: Función segundosTiempo

Realizar esta transformación tiene un $T(n) = 13C$ y un orden $O(C)$.

Para comenzar la ejecución del programa, lo primero que se hace es leer el archivo de entrada que contiene la información inicial de las canciones de la biblioteca, el nombre de este archivo es entregado como argumento a la función **leerCanciones**, que retornará una lista simplemente enlazada de **nodos** con la información de las canciones en cada uno de sus elementos.

EL algoritmo comenzará creando una variable del tipo **FILE** para el archivo de entrada que será abierto en modo lectura, en caso de que el archivo de entrada no exista, se retornará una lista vacía, de esta manera fuera del algoritmo podremos verificar si existió algún error durante la lectura del archivo de entrada.

```
nodo *leerCanciones(char const* archivoCanciones){
    FILE *fp;
    fp = fopen(archivoCanciones, "r");

    nodo *listaCanciones = NULL;

    if(fp == NULL){
        return listaCanciones;
    }
```

Ilustración 9: Lectura del archivo canciones

Una vez comprobado que el archivo haya sido abierto correctamente, se crean variables auxiliares del tipo **string** e **int** para almacenar la información de cada canción presente en el archivo. Con la ayuda de un ciclo **while** se recorre el archivo de texto mientras no se llegue al final de este, en cada ciclo la información leída es almacenada en su respectiva variable auxiliar, luego se transforma la duración de la canción a segundos con el algoritmo **tiempoSegundos** y se crea un **nodo** con la información contenida en las variables auxiliares, dejando temporalmente vacía las componentes correspondientes al artista y al género, una vez creado el nodo, es añadido al final de una lista simplemente enlazada de **nodos** que contiene las canciones. Antes de finalizar, se cierra el archivo de texto y se retorna la lista de canciones.

```
char nombre[25], duracionString[8];
int duracion;
int idArtista, idGenero, puntaje;

while(feof(fp) == 0){
    fscanf(fp, "%s", nombre);
    fscanf(fp, "%s", duracionString);
    fscanf(fp, "%d", &idArtista);
    fscanf(fp, "%d", &idGenero);
    fscanf(fp, "%d", &puntaje);

    duracion = tiempoSegundos(duracionString);
    nodo *aux = crearNodo(puntaje, nombre, duracion, idArtista, "", idGenero, "");
    listaCanciones = agregarFinalNodo(listaCanciones, aux);
}
```

Ilustración 10: Lectura y creación de las canciones

La función **leerCanciones** cuenta con un $T(n) = 14C + 40nC + 3n^2C$ y con un $O(n^2)$.

Ahora que contamos con una lista que contiene las canciones de la biblioteca, procedemos a leer el archivo de entrada que contiene la información de los artistas, esto lo hacemos entregando como argumento el nombre del archivo de texto y la lista de canciones a la función **leerArtistas**.

Este algoritmo, al igual que con el primer archivo de entrada, abre dicho archivo en modo lectura y comprueba que se haya abierto correctamente, una vez comprobado que no exista un error, se crea una variable **string** auxiliar para guardar el nombre del artista y una de tipo **int** para almacenar el identificador numérico de dicho artista. Mediante un ciclo **while** se recorre el archivo de texto mientras no se llegue al final, en cada ciclo se extrae la información del artista y esta información, junto con la lista de canciones es entregada a la función **agregarArtistas**.

```
char artista[25];
int idArtista;

while(feof(fp) == 0){
    fscanf(fp, "%d", &idArtista);
    fscanf(fp, "%s", artista);
    listaCanciones = agregarArtistas(listaCanciones, idArtista, artista);
}
```

Ilustración 11: Lectura del archivo artistas

Dicho algoritmo se encarga de recorrer la lista de canciones con ayuda de un **nodo** auxiliar mientras no se llegue al final de esta. En cada ciclo se comprueba si la canción actual pertenece al artista entregado, si le pertenece, su nombre es agregado en su correspondiente variable dentro de la estructura, una vez recorrida la lista, esta es retornada.

```
nodo *agregarArtistas(nodo *L, int id, char *artista){
    nodo *aux = L;

    while(aux != NULL){
        if(aux->idArtista == id){
            strcpy(aux->artista, artista);
        }
        aux = aux->sig;
    }
    return L;
}
```

Ilustración 12: Función agregarArtistas

El algoritmo **agregarArtistas** posee un $T(n) = 2C + 3nC$, con un orden de $O(n)$.

Este algoritmo es llamado dentro de la función **leerArtistas** hasta que se hayan agregado todos los artistas a sus respectivas canciones; una vez hecho esto, se cierra el archivo de texto y se retorna la lista de canciones actualizada.

Esta función tiene un $T(n) = 9C + 4nC + 3n^2C$, con un $O(n^2)$.

Para que la información de las canciones este completa, solo resta leer el archivo de entrada con la información de los géneros musicales dichas canciones, esto se logra con la función **leerGeneros,** la cual recibe como argumentos el nombre del archivo de texto y la lista con las canciones.

Una vez dentro, se abre el archivo de texto en modo lectura y se comprueba que no existan errores al momento de abrir este archivo, si no existen errores al igual que en la función **leerArtistas** se crean variables auxiliares para guardar el identificador numérico del género y su nombre, luego se recorre el archivo con un ciclo **while** mientras no se llegue al final. En cada uno de estos ciclos, se extrae la información del archivo y es entregada junto con la lista de canciones a la función **agregarGeneros**.

```
char genero[25];
int idGenero;

while(feof(fp) == 0){
    fscanf(fp, "%d", &idGenero);
    fscanf(fp, "%s", genero);
    listaCanciones = agregarGeneros(listaCanciones, idGenero, genero);
}
```

Ilustración 13: Lectura del archivo de géneros

Esta función se encarga de recorrer la lista de canciones hasta llegar a su final mediante un ciclo **while**, en cada ciclo se comprueba si la canción actual pertenece al género entregado, si es así el nombre del género es añadido en su respectiva variable dentro de la estructura; una vez agregado el género a sus respectivas canciones, se retorna la lista actualizada.

```
nodo *agregarGeneros(nodo *L, int id, char *genero){
    nodo *aux = L;

    while(aux != NULL){
        if(aux->idGenero == id){
            strcpy(aux->genero, genero);
        }
        aux = aux->sig;
    }
    return L;
}
```

Ilustración 14: Función agregarGeneros

La función **agregarGeneros** tiene un $T(n) = 2C + 3nC$, junto con un $O(n)$.

Esta función anterior es llamada dentro de **leerGeneros** hasta que se haya agregado cada género a su respectiva canción; una vez realizado, se cierra el archivo y se retorna la lista actualizada.

El algoritmo **leerGeneros** posee un $T(n) = 9C + 4nC + 3n^2C$, por lo que tiene un $O(n^2)$.

Como ya contamos con toda la información de las canciones de la biblioteca, podemos ordenar la lista respecto a su puntaje de popularidad, para esto entregamos la lista de canciones al algoritmo **ordenarCanciones**. Esta función está encargada de generar una nueva lista simplemente enlazada de **nodos** compuesta por los mismos elementos de la lista original, pero en su correspondiente orden.

Para lograr esto, con ayuda de un ciclo **while** mientras existan elementos en la lista original se procederá a obtener la posición del elemento con mayor "jerarquía" dentro de la lista de canciones, este índice será obtenido con el algoritmo **obtenerIndiceMayor** que recibe como argumento solamente la lista de canciones.

Primero se comprobará si la lista entregada tiene solo un elemento, en caso de que sea así, la función retornará 1. En el caso general crearemos variables auxiliares para guardar la información del elemento mayor, estas variables serán inicializadas con los datos del primer elemento de la lista, de esta manera comenzaremos a recorrer la lista comprobando si es que el nuevo elemento revisado tiene mayor jerarquía que el elemento considerado como mayor. Dentro de este proceso existen 3 posibles casos, el primero de estos es que el elemento actual tenga mayor puntaje que el elemento considerado como mayor, si esto ocurre, se actualizarán los datos del elemento mayor con los del elemento actual y se seguirá recorriendo la lista.

```
if(aux->puntaje > puntajeMayor){
    indiceMayor = i;
    strcpy(artistaMayor, aux->artista);
    duracionMayor = aux->duracion;
    puntajeMayor = aux->puntaje;
}
```

Ilustración 15: Elemento con mayor puntaje

Los otros 2 casos ocurren en caso de que ambos elementos tengan igual puntaje, si esto ocurre, primero se comprueba si el elemento actual tiene precedencia alfabética con respecto a su artista sobre el actual elemento mayor, si este es el caso, la información del elemento mayor será reemplazada con la información del elemento actual. Si ambos elementos tienen igual cantidad de puntaje y además pertenecen al mismo artista, se comprueba si la canción actual tiene menor duración que el elemento mayor actual, si esto ocurre se reemplazará la información del elemento mayor con los valores del elemento actual. Una vez revisada toda la lista, se retorna el índice del elemento mayor.

```
else if(aux->puntaje == puntajeMayor){
    //Si el elemento actual va antes alfabeticamente, este se registra como el nuevo elemento mayor
    if(strcmp(aux->artista, artistaMayor) == -1){
        indiceMayor = i;
        duracionMayor = aux->duracion;
        strcpy(artistaMayor, aux->artista);
        puntajeMayor = aux->puntaje;
    }
    //Si ambas canciones tienen el mismo artista, se comprueba su duracion
    else if(strcmp(aux->artista, artistaMayor) == 0){
        if(aux->duracion < duracionMayor){
            indiceMayor = i;
            duracionMayor = aux->duracion;
            strcpy(artistaMayor, aux->artista);
            puntajeMayor = aux->puntaje;
        }
    }
}
```

Ilustración 16: Elementos con igual puntaje

El cálculo de este índice tiene un $T(n) = 7C + 9nC$ y un orden igual a $O(n)$.

Una vez obtenido el índice del mayor elemento de la lista, existen posibles casos, el primero de ellos es que el elemento con mayor jerarquía es aquel que se encuentra en la primera posición, si este es el caso, con ayuda de un **nodo** auxiliar se hace que la lista de canciones original apunte al segundo elemento como si fuera el primero y el antiguo primer elemento es agregado al final de la nueva lista de canciones ordenadas.

```
//En caso de que el mayor elemento sea el primero
if(indiceMayor == 1){
    otroAux = L;
    L = otroAux->sig;
    otroAux->sig = NULL;
    //Se agrega el elmento a la nueva lista
    agregarFinalNodo(listaOrdenada, otroAux);
}
```

Ilustración 17: Elemento mayor en primera posición

El segundo caso ocurre cuando el índice del mayor elemento es mayor a 1, en este caso median un ciclo **while** y un **int** auxiliar que lleve la cuenta de las posiciones, se recorre la lista hasta llegar a la posición anterior a la mayor, una vez llegado, al igual que el otro caso se actualiza la cabeza de la lista de canciones original y el elemento mayor es agregado al final de la lista de canciones ordenadas. Una vez extraídas todas las canciones de la lista original, se retorna la nueva lista con las canciones ordenadas

```
else{
    //Se busca la posicion anterior al elemento mayor de la li
    while(j < indiceMayor-1){
        aux = aux->sig;
        j += 1;
    }

    //Caso general donde el elemento mayor se encuentra en una

    otroAux = aux->sig;
    aux->sig = otroAux->sig;
    otroAux->sig = NULL;
    //Se agrega el elmento a la nueva lista
    listaOrdenada = agregarFinalNodo(listaOrdenada, otroAux);
}
```

Ilustración 18: Índice mayor a 1

Este algoritmo de ordenamiento tiene un $T(n) = 2C + 18nC + 10n^2C$ y un $O(n^2)$.

Con la lista de canciones ya ordenada, procedemos a generar el primero de los archivos de salida, este corresponde al archivo con las canciones ordenadas de manera decreciente con respecto a su puntaje (descendiente alfabéticamente y con respecto a su duración en casos necesarios), para esto utilizaremos la función **generarSalidaCanciones**, que recibe como argumento solamente la lista de canciones ordenada.

Este algoritmo se encargar de crear un archivo de texto con las canciones de la biblioteca ordenada. Primero crea un archivo de tipo **FILE** y lo abre en modo escritura con el nombre "salida.out", si no existe un archivo con este nombre dentro del directorio, se creará este archivo y si ya existe, este será sobrescrito. Luego mediante un ciclo **while** se recorre la lista de canciones ordenadas hasta llegar al final de esta, en cada uno de los ciclos se escribe la información de la canción actual en una línea dentro del archivo de salida. Una vez se escribe la información de todas las canciones de la lista, se cierra el archivo de texto y no se retorna nada.

```
void generarSalidaCanciones(nodo *L){
    FILE* salida;
    //Se abre el archivo "salida.out" en modo escritura (lo creara si no existe o lo remplazara si existe)
    salida = fopen ("salida.out", "w");

    nodo *aux = L;

    while(aux != NULL){
        fprintf(salida, "%d;%s;%s;%s;%s\n", aux->puntaje, aux->nombre, segundosTiempo(aux->duracion), aux->artista, aux->genero);
        aux = aux->sig;
    }
    fclose(salida);
    return;
}
```

Ilustración 19: Generación del archivo de salida.out

El generar este archivo de salida tiene un $T(n) = 5C + 2nC$ y un $O(n)$.

Para poder generar la playlist de canciones, es necesario leer el ultimo archivo de entrada el cual contiene la información correspondiente a las preferencias para la generación de la playlist, antes de leer el archivo se crea un variable **int** que servirá para guardar la duración máxima de la playlist a generar, la dirección de memoria de esta variable es entregada, junto con el nombre del ultimo archivo de entrada a la función **leerPreferencias**.

Una vez dentro de la función, al igual que las anteriores lecturas de archivos, se abre el archivo de texto en modo lectura y se comprueba que no existan errores al momento de abrirlo; una vez comprobado que no existieran errores se crea un **string** auxiliar para almacenar la duración máxima de la playlist, luego se modifica por referencia el valor de la duración de la playlist luego de transformarlo a **int**. Después de leer la duración máxima, se procede a leer las preferencias de género por medio de un ciclo **while**, en el que en cada uno de sus ciclos se extrae el identificador numérico del genero y se crea una estructura del tipo **nodoDato** con él, para luego ser agregado al final de una lista simplemente enlazada de **nodoDatos** con los géneros preferidos. Este ciclo se repite hasta llegar al final del archivo, cuando esto ocurre, se cierra este archivo y se retorna la lista que contiene las preferencias de género.

```
char duracion[8];
fscanf(fp, "%s", duracion);
*duracionPl  = tiempoSegundos(duracion);


int aux = 0;

while(feof(fp) == 0){
    fscanf(fp, "%d", &aux);
    nodoDato *otroAux = crearNodoDato(aux);
    listaPreferencias = agregarFinalDato(listaPreferencias, otroAux);
}
```

Ilustración 20: Lectura de las preferencias

La función **leerPreferencias** posee un $T(n) = 10C + 6nC + n^2C$, con un $O(n^2)$.

En este momento ocurre la parte más importante del programa, la generación del archivo de salida con la playlist con la función **crearPlaylist**, que recibe como argumentos, la lista de canciones ordenada, la lista con las preferencias de género y la duración máxima de la playlist a generar.

Antes de comenzar a seleccionar las canciones, se crea una lista simplemente enlazada de **nodos** la cual servirá para almacenar las canciones ya agregadas a la playlist, una variable **int** para llevar la cuenta de la duración actual de la playlist y 2 variables de tipo **nodo** para ayudar a recorrer la lista. Luego crea una variable del tipo **FILE** para el archivo de salida y se abre en modo escritura.

Mediante un ciclo **while**, se agregarán canciones hasta que se llegue o se superé el limite máximo de duración para la playlist y en ente proceso, existen 4 posibles casos.

3 de estos casos provienen del caso general, este ocurre cuando aun quedan canciones que cumplen con el género preferido dentro de la lista de canciones ordenadas. Se comienza extrayendo el primer elemento de la lista de preferencias que corresponde al identificador del género preferido, luego se comprueba el primer caso posible, este es el caso donde el primer elemento de la lista de canciones cumple con el género musical buscado, si es así, se actualiza la cabeza de la lista de canciones para que apunte al segundo elemento y la canción que cumple con el género buscado, es escrita dentro del archivo de salida, luego se vuelve a agregar el identificador del genero actual al final de la lista de preferencias, de esta manera se podrá lograr añadir las canciones de manera alternada. Una vez añadido el identificador a la lista, se actualiza la duración actual de la playlist, y se agrega la canción dentro de la lista de canciones agregadas cambiando su puntero siguiente por **NULL**.

```
//Caso en el que el primer elemento de la lista de canciones cumple con el genero buscado
if(auxCancion->idGenero == auxPref->dato){
    otroAux = auxCancion->sig;
    auxCancion->sig = otroAux->sig;
    //Se escribe la informacion de la cancion en el archivo de salida
    fprintf(salida, "%d;%s;%s;%s;%s\n", otroAux->puntaje, otroAux->nombre, segundosTiempo(otroAux->duracion), otroAux->artista, otroAux->genero);
    //Se actualiza el nuevo primer elemento de la lista de ids de generos preferidos
    listaPreferencias = listaPreferencias->sig;
    auxPref->sig = NULL;
    //Se agrega el id del genero acual al final de la lista de preferencias
    listaPreferencias = agregarFinalDato(listaPreferencias, auxPref);
    //Se actualiza la duracion actual de la playlist
    duracionActual += otroAux->duracion;
    otroAux->sig = NULL;
    //Se agrega la cancion agregada al final de la lista de canciones ya agregadas
    cancionesAgregadas = agregarFinalNodo(cancionesAgregadas, otroAux);
}
```

Ilustración 21: Agregar elemento en la primera posición

El segundo caso ocurre cuando el primer elemento no cumple con el genero buscado. Mediante un ciclo **while** se recorre la lista de canciones buscando la posición anterior a una canción que cumpla con el género buscado, si se encuentra una canción con el genero buscado, se hace que el elemento la posición anterior a él apunte al elemento que lo sigue y después, al igual que el anterior caso, se procede a escribir su información dentro del archivo de salida, se actualiza la duración actual de la playlist, se agrega el identificador del género al final de la lista de preferencias y se agrega la canción escrita a la lista de canciones ya agregadas.

```
if(auxCancion->sig != NULL){
    otroAux = auxCancion->sig;
    auxCancion->sig = otroAux->sig;
    //Se escribe la informacion de la cancion en el archivo de salida
    fprintf(salida, "%d;%s;%s;%s;%s\n", otroAux->puntaje, otroAux->nombre, segundosTiempo(otroAux->duracion), otroAux->artista, otroAux->genero);
    //Se actualiza el nuevo primer elemento de la lista de ids de generos preferidos
    listaPreferencias = listaPreferencias->sig;
    auxPref->sig = NULL;
    //Se agrega el id del genero acual al final de la lista de preferencias
    listaPreferencias = agregarFinalDato(listaPreferencias, auxPref);
    //Se actualiza la duracion actual de la playlist
    duracionActual += otroAux->duracion;
    otroAux->sig = NULL;
    //Se agrega la cancion agregada al final de la lista de canciones ya agregadas
    cancionesAgregadas = agregarFinalNodo(cancionesAgregadas, otroAux);
}
```

Ilustración 22: Agregar elemento en posición distinta a 1

El tercero de los casos es cuando se recorre la lista buscando una canción que pertenezca al género buscado y se llega al final sin encontrar una canción perteneciente a este. Cuando esto

sucede, se elimina el identificador de género, ya que no quedan canciones nuevas pertenecientes a ese género dentro de la lista de canciones.

```
//Caso en donde no se encontro una cancion que cumpla con el genero buscado
else if (auxCancion->sig == NULL){
    //Se actualiza al nuevo primer elemento de la lista de preferencias
    listaPreferencias = listaPreferencias->sig;
    free(auxPref);
}
```

Ilustración 23: Caso donde no se encuentra una canción del género buscado

El ultimo de los casos sucede cuando la lista de preferencias está vacía, lo que significa que no queda ninguna canción que perteneciente a los géneros de preferencia que no haya sido agregada, en este caso se procede mediante otro ciclo **while** agregar canciones de la lista de canciones agregadas hasta cumplir el tiempo máximo de la playlist. Para esto se extrae el primer elemento de la lista, se escribe dentro del archivo de salida, se actualiza la duración actual de la playlist y se vuelve a agregar la canción al final de la lista de canciones agregadas, de esta manera si se vuelven a agregar todas las canciones ya agregadas, se repetirá el ciclo.

```
if (listaPreferencias == NULL){
    nodo *cancionRepetida;
    //Se agregan canciones repetidas mientras no se haya alcanzado el limite de tiempo
    while(duracionActual < duracionPlaylist){
        cancionRepetida = cancionesAgregadas;
        //Se escribe la informacion de la cancion en el archivo de salida
        fprintf(salida, "%d;%s;%s;%s;%s\n", cancionRepetida->puntaje, cancionRepetida->nombre, segundosTiempo(cancionRepetida->duracion),
        //Se actualiza la duracion actual de la playlist
        duracionActual += cancionRepetida->duracion;
        //Se actualiza el primer elemento de la lista de canciones repetidas
        cancionesAgregadas = cancionesAgregadas->sig;
        cancionRepetida->sig = NULL;
        //Se agrega la cancion agregada al final de la lista de canciones ya agregadas
        cancionesAgregadas = agregarFinalNodo(cancionesAgregadas, cancionRepetida);
    }
```

Ilustración 24: Caso donde se agregan canciones repetidas

Para finalizar, ya sea dentro del ciclo de agregar canciones repetidas o mientras se siguen agregando canciones nuevas, si se llega al máximo con una duración igual, se cierra el archivo y termina la ejecución del programa. En cambio, si se supera el límite máximo de duración se escribe al final del archivo la cantidad de tiempo que se superó el límite máximo de duración, se cierra el archivo y termina la ejecución del programa.

La función que genera el archivo de salida con la playlist posee un $T(n) = 22C + 17nC + 2n^2C)$, con un $O(n^2)$.

2.2 Análisis de resultados

El programa creado resuelve el problema de manera completa, este es capaz de leer múltiples archivos de entrada, generar la representación para cada canción de la biblioteca, ordenar las canciones de manera descendiente con respecto a sus puntajes de popularidad, generar un archivo de salida con las canciones ordenadas y generar un archivo de salida con una playlist que cumpla con las preferencias entregadas.

A pesar todo lo anteriormente nombrado, debido a la manera de abordar el problema, el programa también tiene sus falencias.

2.2.1 Falencias

A pesar de que la solución creada para este problema logra solucionar el problema de manera completa, tiene una falencia con respecto la función **crearPlaylist**. La falencia presente en esta función, es que no realiza exactamente un tarea específica, esto debido a que en un inicio se tenia planificado implementar una estructura de datos de tipo cola para almacenar las canciones ya agregadas y los identificadores de géneros preferidos, pero ya que no se puedo lograr implementar de manera satisfactoria la estructura de colas, dentro función **crearPlaylist** se intenta emular el comportamiento de una cola al volver a agregar el elemento al final de la misma lista. Por lo anteriormente descrito, el código que compone esta función no es modular y es en cierto grado desordenado.

2.2.2 Complejidad algorítmica

Calcular el tiempo de ejecución teórico del programa es complejo, debido a que la cantidad de instrucciones no depende solamente de un factor, depende tanto de la cantidad de canciones ingresadas, la duración máxima de la playlist y la duración de las canciones que componen la playlist. Por lo anterior se utilizo el caso de prueba con una playlist de 45:00 minutos, 10 canciones con una duración promedio de aproximadamente 3:30 minutos y donde se debieron agregar canciones repetidas a la playlist.

Luego de calcular la complejidad algorítmica da cada algoritmo utilizado en la ejecución del programa y bajo las condiciones descritas, se llegó a la siguiente expresión matemática:

$$T(n) = 79C + 91nC + 20n^2C$$

Esta expresión representa la cantidad de instrucciones que tendrá que ejecutar el programa para poder solucionar el problema en el peor de los casos.

De lo anterior también podemos obtener que el orden del programa es $O(n^2)$, a pesar de que se dejaron de lado expresiones, este es el orden que caracteriza el algoritmo usado.

Como podemos ver en las siguientes ilustraciones, un algoritmo de orden n^2, incluso son entradas grandes de datos, la cantidad de instrucciones no crece tan drásticamente.

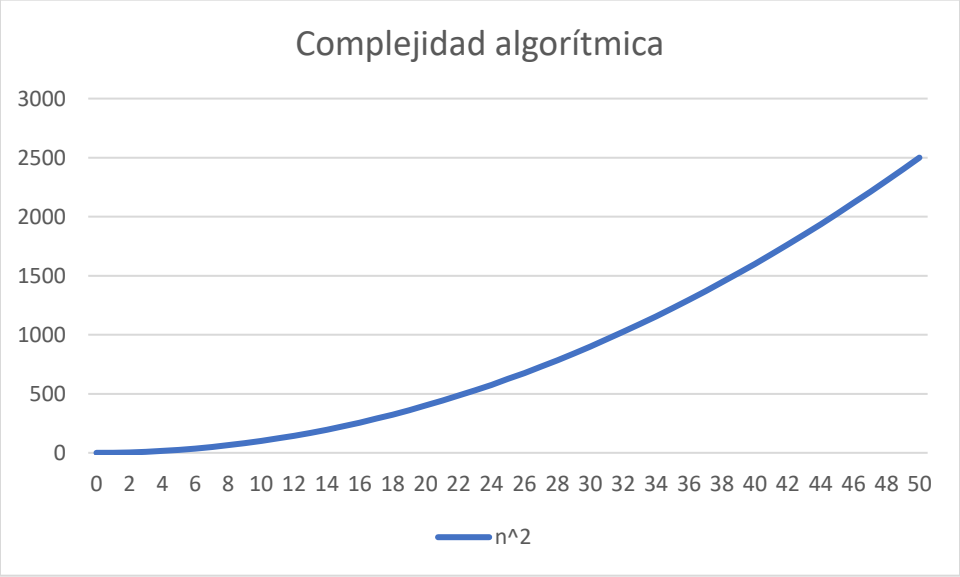

Ilustración 25: Grafico de complejidad algorítmica

Para una mayor comprensión del tiempo de ejecución del programa, se registro el periodo de ejecución con distintos tamaños de entrada. Esto se hizo agregando los siguientes elementos dentro del código:

Primero se importó la librería **time.h**, se crearon las variables para almacenar la hora al momento del inicio y al momento del término. Esta primera variable es inicializada con la hora del equipo antes de comenzar a leer el archivo de entrada.

```
clock_t tiempo_inicio, tiempo_final;
double segundos;
tiempo_inicio = clock();
```

Ilustración 26: Variables para tomar el tiempo

Cuando termina la generarse el archivo de salida se almacena la hora del equipo en la variable **tiempo_final** y se muestra por pantalla la diferencia entre las horas.

```
tiempo_final = clock();
segundos = (double)(tiempo_final - tiempo_inicio)/CLOCKS_PER_SEC;
printf("%f", segundos);
```

Ilustración 27: Obtención del tiempo de ejecución

El tiempo obtenido lo podemos ver en la siguiente tabla:

Tabla 1: Tamaño de entrada vs tiempo de ejecución

Tamaño de entrada	Tiempo real (segundos)
5	0.034
10	0.122
15	0.238
20	0.359
25	0.506
30	0.707

Al igual que en el gráfico, podemos ver claramente que debido al orden de complejidad que posee el programa, incluso con un tamaño de entrada igual a 30, la ejecución del programa sigue demorándose menos de 1 segundo en encontrar la solución, por lo que podemos decir que se llegó a una solución eficiente.

2.3 Conclusión

Se logró resolver el problema planteado de manera completa y eficiente, mediante el análisis de complejidad algorítmica del programa, se puede notar que al trabajar con un tipo de estructura como lo son las listas simplemente enlazadas, mejora enormemente la eficiencia del programa cuando trabajamos con elementos dinámicos, además al analizar cada parte del programa se puede observar las ventajas y desventajas de cada algoritmo utilizado, lo cual será algo a considerar en futuros laboratorios o instancias en las que se tenga que programar. Como principal resultado se puede decir que es de vital importancia considerar las estructuras de datos que mejor se adecuen al contexto en que nos encontramos, además teniendo en cuenta la utilización de memoria y el tiempo de ejecución de nuestro programa.

Referencias

Microsoft (2016). Declaraciones de estructura. (Recuperado 11/12/2020) https://docs.microsoft.com/es-es/cpp/c-language/structure-declarations?view=msvc-160

REAL ACADEMIA ESPAÑOLA: *Diccionario de la lengua española*, 23.ª ed., [versión 23.3 en línea]. (Recuperado 11/12/2020). https://dle.rae.es

Tolentino, N. (2014). Listas enlazadas. (Recuperado 11/12/2020) https://www.monografias.com/trabajos101/las-istas-enlazadas/las-istas-enlazadas.shtml

Universidad Federico Santa María (2020). Punteros – Programación. (Recuperado 11/12/2020) http://progra.usm.cl/apunte/c/punteros.html