Práctica 1.

Manejo de estructuras de datos en C

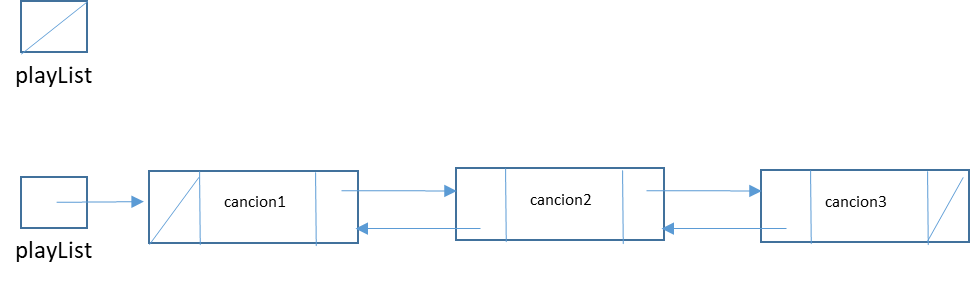
Realizar un programa que permita gestionar una playlist de las canciones preferidas del usuario. Las canciones se implementarán como cadenas de caracteres ya que solo se almacenará su nombre.

Asumiremos que no puede haber dos nodos con el mismo nombre de canción.

Entre las operaciones que podrá realizar podemos destacar:

* Crear una playList vacía
* Añadir una canción al principio de la playList
* Añadir una canción al final de la playList
* Añadir una canción en una PlayList ordenada de canciones
* Mostrar la playList
* Eliminar Todas las canciones de la playList
* Eliminar una canción de la playList
* Eliminar la primera canción de la playList
* Saber cual es la canción actual
* Pasar a la siguiente canción
* Pasar a la canción anterior
* Ordenar alfabéticamente la PlayList

Existen multitud de posibles estructuras de datos que pueden implementarse y utilizarse para manejar esta playList pero siempre debemos buscar la alternativa que sea un compromiso entre eficiencia y simplicidad. En este caso, la recomendación es utilizar una lista doblemente enlazada.



Observa que en cada nodo hay un puntero al siguiente y un puntero al anterior. En ese caso, la estructura de datos sería:

struct Node {

const char\* data;

struct Node\* prev;

struct Node\* next;

};

typedef struct Node \* DList;

Recuerda que si vamos a implementar esta estructura de datos para que pueda ser utilizada en la gestión de mi playList podemos definir un interfaz donde se exponga de forma pública las operaciones que podemos hacer con ella. **Este interfaz se define en un fichero con extensión .h**

¿Qué operaciones se implementarán?

Al menos las que se describen a continuación, pero se pueden definir aquellas que se necesiten y que puedan ser de utilidad para manejar la estructura.

/\*\*

 \* @brief Función para crear un nuevo nodo. Devuelve

 \* un puntero al nodo creado.

 \*

 \* @param value

 \* @return DList

 \*/

DList createNode(const char\* value);

//

/\*\*

 \* @brief Inserción al frente (al principio)

 \*

 \* @param head

 \* @param value

 \*/

void insertAtFront(DList\* head, const char\* value);

/\*\*

 \* @brief  Inserción al final.

 \* Lo inserta como último elemento

 \*

 \* @param head

 \* @param value

 \*/

void insertAtEnd(DList\* head, const char\* value);

/\*\*

 \* @brief Inserción después de una canción. Busca una

 \* canción y si existe añade un nuevo nodo con la nueva canción

 \* después de la canción existente.

 \*

 \* @param head

 \* @param cancion: Canción existente en la lista

 \* @param value: Nueva canción a añadir

 \*/

void insertAfter(DList \*head, const char \*cancion, const char\* value);

/\*\*

\* @brief Inserción antes de una canción. Busca una

 \* canción y si existe añade un nuevo nodo con la nueva canción

 \* antes de la canción existente.

 \*

 \*

 \* @param head

 \* @param cancion: Canción existente en la lista

 \* @param value: Nueva canción a añadir

 \*/

void insertBefore(DList\* head, const char \*cancion, const char\* value);

/\*\*

 \* @brief Eliminación desde el frente. Se eliminan

 \* todos los nodos por el primero de la lista.

 \*

 \* @param head

 \*/

void deleteFromFront(DList\* head);

/\*\*

 \* @brief Mostrar toda lista

 \*

 \* @param head

 \*/

void printList (DList head);

/\*\*

 \* @brief Borrar un elemento de la lista si existe.

 \* Devuelve 1 si lo borra, 0 si no existe la canción

 \*

 \* @param list

 \* @param value

 \* @return int

 \*/

int deleteNode(DList \*list, const char \*value);

/\*\*

 \* @brief Procedimiento para insertar un nodo de manera ordenada

 \*

 \* @param lista

 \* @param valor

 \*/

void insertarOrdenado(DList\* lista, const char\* valor);

/\*\*

 \* @brief Procedimiento para ordenar la lista utilizando el algoritmo de burbuja

 \*

 \* @param lista

 \*/

void ordenarLista(DList\* lista);

/\*\*

 \* @brief Borro todos los elementos de la lista

 \*

 \* @param head

 \*/

void DeleteAll (DList \*head);