Implementing a Personal Playlist Manager in C

Data Structures. Pepe Gallardo, 2024.

Dpto. Lenguajes y Ciencias de la Computación. University of Málaga

---

Introduction

In this lab session we will be diving into the practical application of doubly linked lists by implementing a personal playlist manager. This project will help you gain hands-on experience with dynamic data structures and reinforce your understanding of memory management in C.

Motivation

In the era of digital music, playlist management is a core feature of many music applications. By building a playlist manager, you will not only practice important programming concepts but also create a practical tool that mirrors real-world software. This exercise will help you understand how music streaming services might handle playlist operations behind the scenes.

The Playlist Challenge

Imagine you are a software developer tasked with creating a playlist management system for a new music streaming app. Your system needs to allow users to create playlists, add and remove songs, navigate through the playlist, and perform various other operations.

Data Structure

We will use a doubly linked list to represent our playlist. Here are the structures we will be working with:

```
#define MAX_NAME_LENGTH 30

struct Node {
    char songName[MAX_NAME_LENGTH + 1];
    struct Node* p_next;
    struct Node* p_previous;
};

struct PlayList {
    struct Node* p_first;
    struct Node* p_playing;
};
```

Let us break down these structures:

1. `struct Node`:
   - `songName`: An array of characters to store the song name. It can hold up to 30 characters plus a '\0' terminator.
   - `p_previous`: A pointer to the previous node in the doubly linked list.
   - `p_next`: A pointer to the next node in the doubly linked list.
2. `struct PlayList`:
   - `p_first`: A pointer to the first node in the playlist.
   - `p_playing`: A pointer to the node of the song that is currently playing.

In this exercise we will assume that all song names are unique (i.e., no duplicate songs in the playlist).

The picture in Figure 1 shows a visual representation of a playlist with four songs. The `p_first` pointer points to the first node in the playlist, and the `p_playing` pointer points to the node of the song that is currently playing (named "song2" in this example). Each node contains the song name (a real implementation would also include other song information such as artist, album, etc.) and pointers to the next and previous nodes. Notice that the `p_previous`

pointer of the first node and the `p_next` pointer of the last node are NULL which indicates the beginning and end of the playlist, respectively.

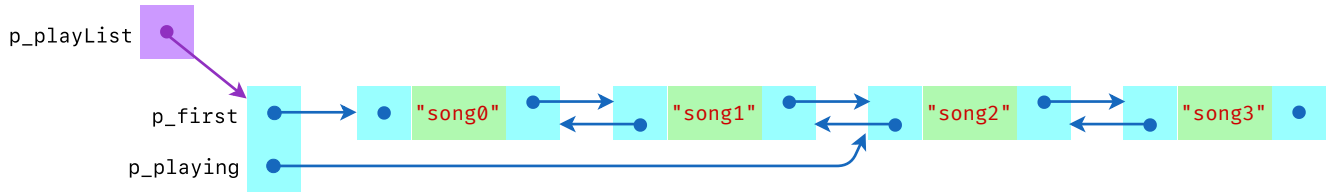

Figure 1: Doubly Linked List representing a PlayList

Function Descriptions

Here are the function prototypes you will need to implement, along with explanations of their arguments and purpose:

```
struct PlayList* PlayList_new();
```

- Purpose: Allocates memory for a new `PlayList` and initializes it as empty. En empty playlist has both `p_first` and `p_playing` pointers set to NULL.
- Returns: A pointer to the newly created playlist.

```
void PlayList_insertAtFront(struct PlayList* p_playList, const char* songName);
```

- Purpose: Creates a new node with the given song name and inserts it at the beginning of the playlist.
- Arguments:
  - `p_playList` - A pointer to the `PlayList` structure.
  - `songName` - The string (song name) to be inserted.

```
void PlayList_insertInOrder(struct PlayList* p_playList, const char* songName);
```

- Purpose: For this function, we assume that the input playlist keeps its song in alphabetical order. This function should insert a new song into the playlist maintaining alphabetical order.
- Arguments:
  - `p_playList` - A pointer to the sorted `PlayList` structure.
  - `songName` - The string (song name) to be inserted.

```
void PlayList_insertAtEnd(struct PlayList* p_playList, const char* songName);
```

- Purpose: Creates a new node with the given song name and appends it to the end of the playlist.
- Arguments:
  - `p_playList` - A pointer to the `PlayList` structure.
  - `songName` - The string (song name) to be inserted.

```
bool PlayList_insertAfter(struct PlayList* p_playList, const char *targetSong, const char* newSong);
```

- Purpose: Creates a new node with the given song name and inserts it after a specified song in the playlist.
- Arguments:
  - `p_playList` - A pointer to the `PlayList` structure.
  - `targetSong` - The song after which to insert the new song.
  - `newSong` - The string (song name) to be inserted.
- Returns: `true` if successful, `false` if failed (e.g., if target song not found).

```
bool PlayList_insertBefore(struct PlayList* p_playList, const char *targetSong, const char* newSong);
```

- Purpose: Creates a new node with the given song name and inserts it before a specified song in the playlist.
- Arguments:
  - `p_playList` - A pointer to the `PlayList` structure.
  - `targetSong` - The song before which to insert the new song.
  - `newSong` - The string (song name) to be inserted.
- Returns: `true` if successful, `false` if failed (e.g., if target song not found).

```
void PlayList_deleteFromFront(struct PlayList* p_playList);
```

- Purpose: Removes the node corresponding to the first song from the playlist.
- Argument: `p_playList` - A pointer to the `PlayList` structure.

```
bool PlayList_deleteSong(struct PlayList* p_playList, const char *songName);
```

- Purpose: Removes the node corresponding to a specific song from the playlist.
- Arguments:
  - `p_playList` - A pointer to the `PlayList` structure.
  - `songName` - The string (song name) to be deleted.
- Returns: `true` if the song was found and deleted, `false` otherwise.

```
void PlayList_print(const struct PlayList* p_playList);
```

- Purpose: Displays the contents of the playlist, printing each song name in the playlist.
- Argument: `p_playList` - A pointer to the `PlayList` structure to be printed.

```
void PlayList_sort(struct PlayList* p_playList);
```

- Purpose: Sorts the playlist in alphabetical order using bubble sort.
- Argument: `p_playList` - A pointer to the `PlayList` structure.
- Challenge: what is the time complexity of this sorting algorithm? What is the most efficient way to implement the sorting function?

```
void PlayList_deleteAll(struct PlayList* p_playList);
```

- Purpose: Removes all nodes from the playlist making it empty.
- Argument: `p_playList` - A pointer to the `PlayList` structure.

```
char* PlayList_playingSong(const struct PlayList* p_playList);
```

- Purpose: Returns the name of the song that is currently playing.
- Argument: `p_playList` - A pointer to the `PlayList` structure.
- Returns: A pointer to a newly heap-allocated string containing the playing song name. The caller is responsible for freeing the memory of the returned string when it is no longer needed.

```
void PlayList_playNext(struct PlayList* p_playList);
```

- Purpose: Moves the playing song pointer to the next song. No action if at the end.
- Argument: `p_playList` - A pointer to the `PlayList` structure.

```
void PlayList_playPrevious(struct PlayList* p_playList);
```

- Purpose: Moves the playing song pointer to the previous song. No action if at the beginning.
- Argument: `p_playList` - A pointer to the `PlayList` structure.

```
void PlayList_free(struct PlayList** p_p_playList);
```

- Purpose: Frees all memory associated with the playlist (both the nodes and the playlist structure). It also sets the pointer to the playlist to NULL.
- Argument: `p_p_playList` - A pointer to the pointer to the `PlayList` structure.

Implementation Task

Your task is to implement these functions in the file named `PlayList.c`. Ensure that your implementation adheres to the function prototypes provided and correctly manages the doubly linked list structure.

Implementation Guidelines

1. Handle edge cases, such as inserting into an empty list or deleting the last remaining song.
2. Maintain the integrity of the doubly linked list by correctly updating the `p_next` and `p_previous` pointers of affected nodes during all operations and the `p_first` and `p_playing` pointers of the playlist.
3. Consider how to maintain the `p_playing` pointer when performing operations that might affect its position.
4. For `insertBefore` and `insertAfter` functions, handle cases where the target song is not found or the playlist is empty.
5. When implementing the sorting function, consider the efficiency of your algorithm.

Testing Your Implementation

To test your implementation, complete the `main` function in file `main.c` to demonstrate the use of each of your playlist operations. Create scenarios that test edge cases and verify that your playlist maintains its integrity after multiple operations.

Conclusion

By completing this lab, you will gain practical experience with doubly linked lists, deepen your understanding of data structures, and create a useful playlist management system. This exercise will help you appreciate how fundamental data structures are used in real-world applications.

Remember, the key to success in this lab is to break the problem down into smaller, manageable pieces. Start with the basic operations and build up to the more complex ones. Do not hesitate to debug your code frequently and ask for help if you get stuck.