Implementing a Personal Playlist Manager in C

Data Structures. Pepe Gallardo, 2025.

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

Introduction

Today, we're diving into the practical application of doubly linked lists by building a personal playlist manager in C. This project will give you hands-on experience with dynamic data structures, reinforce your understanding of memory management, and show you how fundamental computer science concepts power the apps you use every day.

Motivation: Behind the Music

In the era of digital music, playlist management is a core feature of every streaming service. By building your own playlist manager, you will not only practice essential programming skills but also create a tool that mirrors real-world software. This exercise will demystify how these services handle complex operations like adding, removing, and reordering songs behind the scenes.

The Playlist Challenge

Imagine you're a software developer at a new music streaming startup. Your first task is to build the core playlist system. This system must allow users to create playlists, add and remove songs in various ways, sort their tracks, and navigate through the playlist for playback. Your implementation will be the backbone of the entire app!

The Data Structure

We will use a **doubly linked list** to represent our playlist. This structure is perfect for the job because it allows us to move both forwards and backwards through the list efficiently, which is essential for a music player.

Here are the C `struct`s you will be working with:

```c
#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's break this down:

1. **`struct Node`**: Represents a single song in the playlist.
   - `songName`: A character array to store the song's name (up to 30 characters plus a `'\0'` terminator).
   - `p_next`: A pointer to the next `Node` in the list.

    - `p_previous`: A pointer to the previous `Node` in the list.

2. **`struct PlayList`**: Represents the entire playlist.
    - `p_first`: A pointer to the very first `Node` in the list.
    - `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 marks the beginning and end of the playlist, respectively.

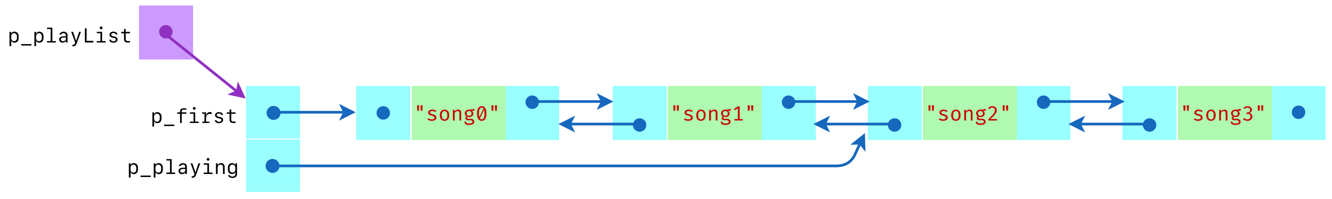

Figure 1: Doubly Linked List representing a PlayList

Key Behavior: Managing the `p_playing` Pointer

The `p_playing` pointer is crucial for user experience, and its behavior is strictly defined. Your implementation must adhere to the following rules, which are verified by the test suite:

- **Initialization**: A new, empty playlist must have its `p_playing` pointer set to `NULL`.
- **First Song Added**: When the very first song is added to an empty list (via any insertion function), `p_playing` must point to this new song.
- **Insertion into a Non-Empty List**: If a new song is added to a playlist that already contains songs, **`p_playing` must not move**. It should remain on the same song that was playing before the insertion.
- **Deletion of the Playing Song**: If the currently playing song is deleted:
    - The `p_playing` pointer must be **reset to point to the new first song** (`p_first`) of the updated list.
    - If this deletion makes the list empty, `p_playing` correctly becomes `NULL` (as `p_first` will be `NULL`).
- **Deletion of a Non-Playing Song**: If a song *other than* the currently playing one is deleted, `p_playing` remains unchanged.
- **Sorting**: After sorting the playlist with `PlayList_sort`, `p_playing` must be **reset to point to the new first song** of the now-sorted list.

Your Mission: Function Implementation

Your task is to implement the following functions in the file `PlayList.c`. Adhere strictly to the function prototypes and the behavior described.

Setup and Teardown

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

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

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

- **Purpose**: Frees all memory associated with the playlist (all nodes and the playlist structure itself). It must also set the user's playlist pointer to `NULL`.
- **Argument**: `p_p_playList` - A pointer to the pointer to the `PlayList` structure.

Insertion Operations

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

- **Purpose**: Creates a new node and inserts it at the beginning of the playlist.

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

- **Purpose**: Creates a new node and appends it to the end of the playlist.

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

- **Purpose**: Inserts a new song into a playlist that is assumed to be already sorted, maintaining alphabetical order.

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

- **Purpose**: Creates a new node and inserts it immediately after a specified `targetSong`.
- **Returns**: `true` if successful, `false` if `targetSong` is not found.

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

- **Purpose**: Creates a new node and inserts it immediately before a specified `targetSong`.
- **Returns**: `true` if successful, `false` if `targetSong` is not found.

Deletion Operations

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

- **Purpose**: Removes the first song from the playlist.

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

- **Purpose**: Removes a specific song from the playlist.
- **Returns**: `true` if the song was found and deleted, `false` otherwise.

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

- **Purpose**: Removes all nodes from the playlist, making it empty.

Playback and Navigation

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

- **Purpose**: Returns the name of the song currently indicated by `p_playing`.
- **Returns**: A pointer to a **newly heap-allocated string** containing the song name. The caller is responsible for `free`ing this memory. Returns `NULL` if no song is playing.

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

- **Purpose**: Moves the `p_playing` pointer to the next song. Does nothing if at the end of the list.

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

- **Purpose**: Moves the `p_playing` pointer to the previous song. Does nothing if at the beginning.

Utility Functions

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

- **Purpose**: Displays the contents of the playlist by printing each song name to the console, one per line.

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

- **Purpose**: Sorts the playlist in alphabetical order using the **bubble sort** algorithm.
- **Challenge**: What is the time complexity of this sorting algorithm?

Project Structure & Execution

Your project is organized into several files, but you only need to edit `PlayList.c`.

- `PlayList.h`: The header file containing all struct definitions and function prototypes. Do not modify.
- `PlayList.c`: **This is where you will write your implementation.**
- `Main.c`, `Demo.c`, `Suite.c`: Files that set up the execution environment and contain the test suite. Do not modify.

How to Run Your Code: Demo vs. Test Suite

The project has two execution modes, controlled by macros in `main.c`:

1. **Test Suite Mode (`RUN_TESTS`)**: This is the **default mode** and your primary goal. It runs a comprehensive suite of unit tests that will verify every aspect of your implementation, from logic to memory management.
2. **Demonstration Mode (`RUN_DEMO`)**: This mode runs a simple demonstration program. It is useful for your own manual debugging but does not guarantee your code is correct.

To switch between modes, open `Main.c` and change the `#define` directives at the top before compiling and running your program:

To run the demonstration:

```
#undef RUN_TESTS
#define RUN_DEMO
```

To run the unit tests (this should be your default):

```
#define RUN_TESTS
#undef RUN_DEMO
```

Testing: Your Path to Victory

Your success in this lab is measured by one metric: **passing the entire unit test suite.**

The provided test suite is your best friend. It is designed to catch every common mistake and will be the ultimate judge of your implementation. It automatically checks for:

- **Functional Correctness**: Does your code work as expected in normal situations?
- **Edge Cases**: Does your code handle empty lists, single-song lists, and operations at the list's boundaries?
- **Memory Management**: Does your code `malloc` and `free` memory correctly, with **no memory leaks**?
- **Assertion Contracts**: Does your code fail gracefully with an `assert` when given invalid input (like a `NULL` playlist)?

Run the tests early and often. Read the test names in `Suite.c` to understand exactly what is being tested. A passing test suite is a completed assignment!

Conclusion

By completing this lab, you will gain deep, practical experience with one of the most fundamental data structures in computer science. Remember, the key to success is to break the problem down. Implement and test one function at a time. Start with the basics (`PlayList_new`, `insertAtFront`) and build up to the more complex operations. Don't hesitate to ask for help if you get stuck. Good luck and may your playlists always be in tune!