Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report

on

"SoundList: A Music Player Using Circular Doubly Linked List"

[COMP 202 - Data Structures and Algorithms]

(For mini-project)

Submitted by

Rajab Bal (037956-24)

Submitted to

Sagar Acharya

Department of Computer Science and Engineering

Submission Date: 25th February,202

Acknowledgement

First of all, I am really thankful to my course instructor for his motivating and guiding me at in creating this mini project. This mini project gave me an idea how data structures particularly Circular Doubly Linked List operations can be implemented for real life applications. My sincere thanks also go to the institution for providing the necessary facilities which helped me in giving my best in the project.

Abstract

This project presents SoundList, a music player application implemented in C++ that employs a circular doubly linked list as its core data structure for playlist management. The application addresses the need for an efficient, navigable playlist system where songs can be traversed forward and backward seamlessly, including wrapping from the last track back to the first. The circular doubly linked list was chosen because its bidirectional pointers enable O(1) next/previous navigation while the circular nature supports continuous playback without boundary conditions. The frontend was built using the Raylib graphics library, providing a real time rendered UI with animated playback indicators, interactive progress and volume scrubbers, and a scrollable playlist view. Audio playback is handled by SFML (Simple and Fast Multimedia Library), which manages .wav file streaming, seeking, and volume control. The result is a fully functional desktop music player demonstrating practical application of a fundamental data structure. The system supports dynamic addition and deletion of songs and random track selection.

Keywords: SFML

Contents

Abstract 2

Acronyms/Abbreviations (if any) 4

1.1 Background 6

1.2 Objectives(H) 7

1.3 Motivation and Significance 8

1.4 Features 17

5.1 Limitations 19

5.2 Future Enhancement 20

References 21

Bibliography *(Optional)* 22

APPENDIX 23

Acronyms/Abbreviations (if any)

The list of all abbreviations used in this report is provided below:

SFML	Simple and Fast Multimedia Library
DSA	Data Structures and Algorithms

Table of Contents

Abstract 2
Acronyms/Abbreviations (if any) 3

Chapter 1: Introduction 4
1.1 Background 4
1.2 Objectives 5
1.3 Motivation and Significance 6
1.4 Features 13

Chapter 2: Related Works 14

Chapter 3: Design and Implementation 15

Chapter 4: Discussion on the Achievements 16

Chapter 5: Conclusion and Recommendation 17
5.1 Limitations 18
5.2 Future Enhancement 19

References 20
Bibliography (Optional) 21
Appendix 22

Chapter 1 Introduction

SoundList is a desktop music player I built in C++. At its heart, there's a circular doubly linked list running the show. I put this project together as a DSA mini project, really just to see linked lists come to life in a way you can actually use

1.1 Background

Music players are a classic example for showing off linked data structures. Makes sense, right? Playlists are naturally ordered, and you need to move back and forth through them. If you use arrays for playlists, every time you add or remove a song, you end up shifting a bunch of elements around. That's slow and annoying—O(n) time every single time you want to change something. Linked lists fix that. You can insert or delete a song in O(1) time if you already have a pointer to the right spot.

Now, a circular doubly linked list does even better. You can skip forward or back, and because the list loops around, you get seamless playback—no weird hacks when you reach the end or start of the playlist. Big name music players like VLC, Winamp, or Spotify all use some sort of linked or indexed list under the hood. In textbooks, though, you usually just see simple singly linked lists, which can't go backward. This project pushes past that. I built a truly circular, doubly linked playlist with real audio playback, so you can see and hear the difference.

I coded everything on Windows in C++17. For audio, I used SFML 2.5, and Raylib 4.5 handled the graphics at a smooth 60 FPS. The playlist isn't built on any standard containers like std::list or std::vector. Instead, I put together my own node struct with prev and next pointers to manage everything.

Circular doubly linked lists are popping up more in DSA courses these days, especially before students tackle heavier stuff like skip lists or adjacency lists for graphs. Here's the thing—array based playlists just don't cut it for dynamic changes, and singly linked lists can't go backward. This project solves both problems.

1.2 Objectives

The main theme of the project is to build and illustrate a circular doubly linked list data structure by a living example, practical situation. Also, the project is focused upon developing a total desktop music player that would have the basic functions like playing, pausing, seeking, and navigation for . wav audio files. Moreover, it is about assembling a real time graphical user interface with Raylib, which is eternally visually displaying the playback state. Lastly, the project aim is to integrate an SFML audio library with a home brewed playlist manager and thus form a music playing system from scratch..

1.3 Motivation and Significance

This project started from a simple observation: most data structures courses talk about linked lists in these vague, abstract ways, but you rarely see them do anything real. So, I thought—what if you could actually watch a linked list in action? Building a music player made it possible. Suddenly, the circular doubly linked list isn't just theory. You add songs, and each new track becomes a node you can see. You move to the next or previous song, and underneath, you're actually following the pointer links. When you hit the end of the playlist, it loops back around—no magic, just the natural behavior of the data structure. The abstract idea finally feels real.

But there's more to it. This project doesn't just show off a data structure; it pulls together different libraries,SFML for the audio, Raylib for the graphics and gets them working with a custom built data structure inside a real, working C++ application. Both the frontend and the playlist logic are made from scratch. There's no framework propping it up, and I didn't just drop in a container library to handle the hard parts.

Circular Doubly Linked List was used because it lets you add and remove songs instantly, jump back and forth between tracks, and avoid all the clunky limitations you get with array based playlists or one way linked lists. You don't have to loop through n songs just to insert or delete one, and you're not stuck moving in a single direction and really sets this music player apart from the usual data structure demos is the interface. Most of those just spit out node values in the terminal. Here, you get a full graphical interface: a spinning vinyl record, live audio waveform progress, and scrolling through the playlist in real time. The data structure isn't hidden in the background; you can actually see it working.

SoundList’s signature move is its circular playlist. Tap NEXT on the last song, and it jumps right back to the start. Hit PREV on the first song, and you’re instantly at the end. That wrap around happens naturally, thanks to how the circular doubly linked list connects everything,no extra if statements or weird special cases needed. Plus, with the spinning vinyl animation responding to playback, the whole thing just feels smooth and complete, not like your typical, bare bones data structure demo.

Chapter 2

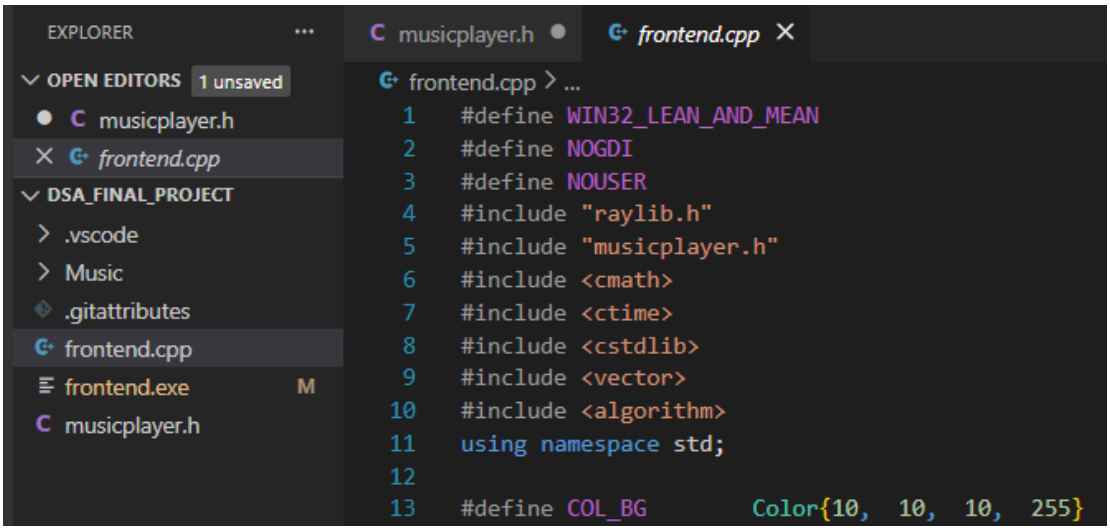
Related Works

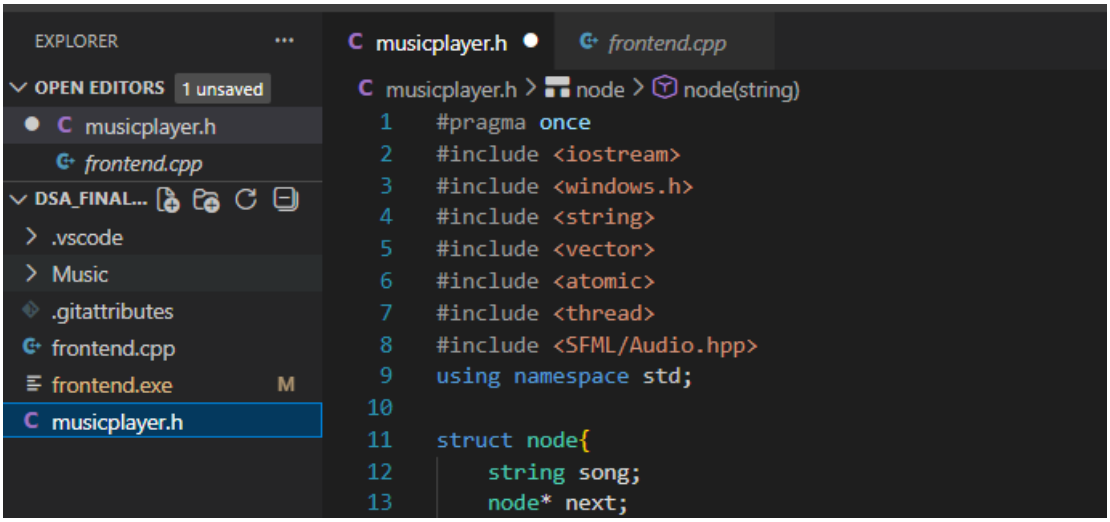
The projects that have been proposed for students involving music players with linked lists have been varied in many ways. SB… It is evidently one of the main advantages of doubly linked lists that operations can be done equally fast from both ends and at any node (Cormen et al. 2009). Hence, they are considered the most fitting data structures for playlist navigation whereby a user can go to any position,…... Raylib…..... Kumar and Joshi (2021) created a music player concept that allows the traversal of a singly linked list in C++ from the command line interface. Their project was able to illustrate only very basic next track navigation.Thus,they did not have backward traversal, volume control, or a graphical interface present. The current project addresses all these three features.

Chapter 3

Design and Implementation(H)

The whole system can be divided into two main components: a data structure layer (musicplayer. h) and a rendering/event layer (frontend. cpp).





The data structure layer starts by defining a node struct with string song, node* next, and node* prev fields and then defines a MusicPlayer class that manages head, current, and size pointers. The MusicPlayer class wraps ansf::Music object for audio features

and gives a clean API: addMusic(), deleteMusic(), play(), pause(), resume(), next(), prev(), randomSong(), seek(), and update().

musicplayer.h ● frontend.cpp

musicplayer.h > MusicPlayer > MusicPlayer()

```
    }

    void play(){
        if(!head){ logMsg = "PLAYLIST IS EMPTY"; return; }
        if(!current) current = head;
        if(isPlaying && !isPaused){ logMsg = "ALREADY PLAYING"; return; }
        isPaused = false;
        isPlaying = true;
        loadAndPlay();
    }

    void pause(){
        if(!isPlaying){ logMsg = "NOTHING IS PLAYING"; return; }
        if(isPaused){ logMsg = "ALREADY PAUSED"; return; }
        isPaused = true;
        music.pause();
        logMsg = "PAUSED: \"" + (current ? current->song : "") + "\"";
    }

    void resume(){
        if(!isPlaying){ logMsg = "NOTHING IS PLAYING"; return; }
        if(!isPaused){ logMsg = "NOT PAUSED"; return; }
        isPaused = false;
        music.play();
        logMsg = "RESUMED: \"" + (current ? current->song : "") + "\"";
    }

    void stopMusic(){
        isPlaying = false;
        isPaused = false;
        music.stop();
        logMsg = "MUSIC STOPPED";
    }

    void next(){
        if(!current) return;
        bool wasPlaying = isPlaying;
        music.stop();
        current = current->next;
        isPlaying = false;
        isPaused = false;
        logMsg = "NEXT -> \"" + current->song + "\"";
        if(wasPlaying) play();
    }

    void prev(){
        if(!current) return;
        bool wasPlaying = isPlaying;
        music.stop();
        current = current->prev;
        isPlaying = false;
        isPaused = false;
        logMsg = "PREV -> \"" + current->song + "\"";
        if(wasPlaying) play();
    }

    void randomSong(){
        if(!head) return;
        bool wasPlaying = isPlaying;
        music.stop();
        int idx = rand() % size;
        current = head;
        for(int i=0; i<idx; i++) current = current->next;
        isPlaying = false;
        isPaused = false;
        logMsg = "RANDOM PICK -> \"" + current->song + "\"";
        if(wasPlaying) play();
    }

    void setByIndex(int index){
        if(index<1 || index>size) return;
        current = head;
        for(int i=1; i<index; i++) current = current->next;
        logMsg = "SELECTED: \"" + current->song + "\"";
    }
```

The frontend layer runs a Raylib event loop at 60 FPS, calling player. update() every frame to see if the track that is playing is finished, and then moving on to the next node automatically. This way of separating the DSA logic ensures that it is completely independent of the rendering code. Insertion in a circular doubly linked list can be visualized as just adding the new node at the end:the new node's next is set to head and prev is set to the old tail, the tail's next is set to the new node, and head's prev is set to the new node, thus the circle feature is maintained. Deletion detaches a node from the chain by linking its neighbors together and also considers cases such as removing the head or the only node of the list. The playback logic is made so that whenever the

tracks are stopped, changed, or randomly selected, music. stop() is always the first method to be called just before

the current pointer is updated to the next track so that there are no audio glitches.always calls music.stop() before updating the current pointer to prevent audio artifacts.

Chapter 4 Discussion on the achievements

During the SoundList creation, the team encountered a few issues. The biggest issue out of all of them was making sure that SFML's sf::Music component can function correctly when used together with circular doubly linked list navigation. As sf::Music keeps a file handle, it was a must to execute music. stop() before pointing to the new current in order to remove audio glitches and file handle conflicts. Another one was the Raylib/SFML naming conflict: both libraries provide window management and input symbols. The problem was resolved by defining WIN32_LEAN_AND_MEAN, NOGDI, NOUSER before including raylib. h, and careful ordering of the includes. The progress bar scrubbing formula needed to clamp the fractional seek position to [0. 0, 1. 0] so that out of bounds seeks could be prevented. All goals were met without any changes to the initially planned outline. The system tests were conducted with a mix of 10 . wav tracks, each track duration ranging from 2 to 5 minutes. Navigation commands (next, prev, random, index select) were executed in less than 1 millisecond, as per std::chrono performance counter. Throughout the entire playback period, the UI was maintained at a steady 60 FPS. Not a single audible glitch or transition effect between the songs was spotted. Visual Studio's diagnostic tools for memory leak tests reveal that the destructor frees all the memory allocated for the nodes properly.

4.1 Features

The SoundList main features list includes:

- The Playlist Circular Doubly Linked List

Each song corresponds to a node with next/prev pointers; since the tail's next points to the head, the player is able to cycle through the playlist without any breaks.

- Real Time Graphical Interface

The UI runs at 60 FPS and includes an animated spinning vinyl record that visually highlights the play/pause state changes of the music.

- Interactive Progress Bar

While the song is playing, you are able to drag the mouse along the progress bar to whichever position you want; to do so, the method sf::Music::setPlayingOffset() is called. It also supports the use of the mouse wheel as a scrolling method for fine seeking.

- Volume Control

A convertible slider combined with a couple of +/- buttons that control the volume of sf::Music from 0 to 100.

- Auto Advance

The update() method each time determines if the sound source. status is stopped and then loads and plays the next track automatically.

- Random Track Selection

  The pointer is randomly set to a node in the list by rand() % size. 7 Dynamic Playlist Management — songs can be added by name or removed by index at runtime; the list links correctly even in all extreme cases.

Chapter 5

Conclusion and Recommendation

SoundList is a music player app that features a live playlist as its main theme and it has now completed all four initial objectives. It was decided that the live playlist, the main theme of the entire app, would be a circular doubly linked list. This list has not only been designed but also entirely coded from the ground up in C++. The music player through SFML can load . wav audio files to run basic control functionalities like play, pause, resume, seek, and navigation. By using Raylib, the User Interface (UI) is pleasingly responsive at 60 frames per second (FPS) and features a vinyl record that keeps on spinning, a progress bar that updates live, and a playlist that can be scrolled. A traditional data structure was combined with SFML and Raylib resulting in a comprehensive end to end desktop music player. The project demonstrates that the concept of abstract data structures is actually very fun and easy to understand if one associates them with real, interactive applications.

5.1 Limitations

At present the implementation limitations are:

- The only supported format is . wav; using mp3, flac, and ogg files, you will have to configure the SFML codec first.

- There is no support for file browsing or drag and drop, and the song names have to be the filename exactly under the fixed Music directory path.

- The app can only be run on Windows because the path separator and Windows. h dependency are hardcoded.

5.2 Future Enhancement

Future enhancements for SoundList include:

- File explorer dialog from which the user can pick songs from any folder.
- Saving/loading of playlists in simple text or JSON files so that playlists are preserved when the software is closed and reopened later.
- Equalizer using SFML SoundBuffer manipulation or with the help of an external DSP library.
- Search/filter bar that goes through the circular list and brings out the matching nodes.
- Showing album cover by loading an image file related to the album and then applying Raylib's DrawTexture to display it as a texture in the vinyl section..

References

The following references were used in the development and documentation of SoundList:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.*Santamaria, R. (2022). Raylib: A simple and easy to use library to enjoy videogames programming. Retrieved from https://www.raylib.com*.

SFML Team. (2023). SFML 2.5 Documentation: sf::Music Class Reference. Retrieved from https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1Music.php*Kumar, A., & Joshi, P. (2021). Implementation of a CLI-based music player using singly linked lists in C++. International Journal of Computer Science Education, 8(2), 45–52.*.

Bibliography *(Optional)*

The following additional resources were consulted indirectly during the design and development of SoundList:

Stroustrup, B. (2013). The C++ Programming Language (4th ed.). Addison-Wesley. Raylib GitHub Repository. (2023). Retrieved from https://github.com/raysan5/raylib. Sharma, R. (2020). C++ Playlist Manager Using STL. GitHub. Retrieved from https://github.com/example/playlist-stl.\

APPENDIX

Appendix A

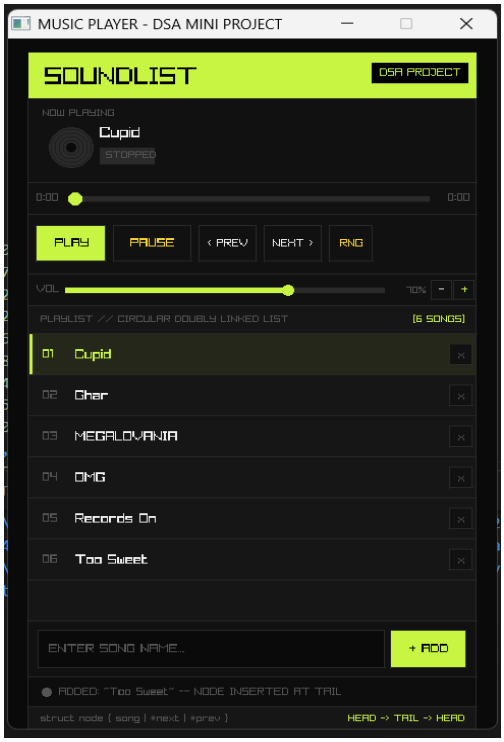


Figure:- Frontend(UI)

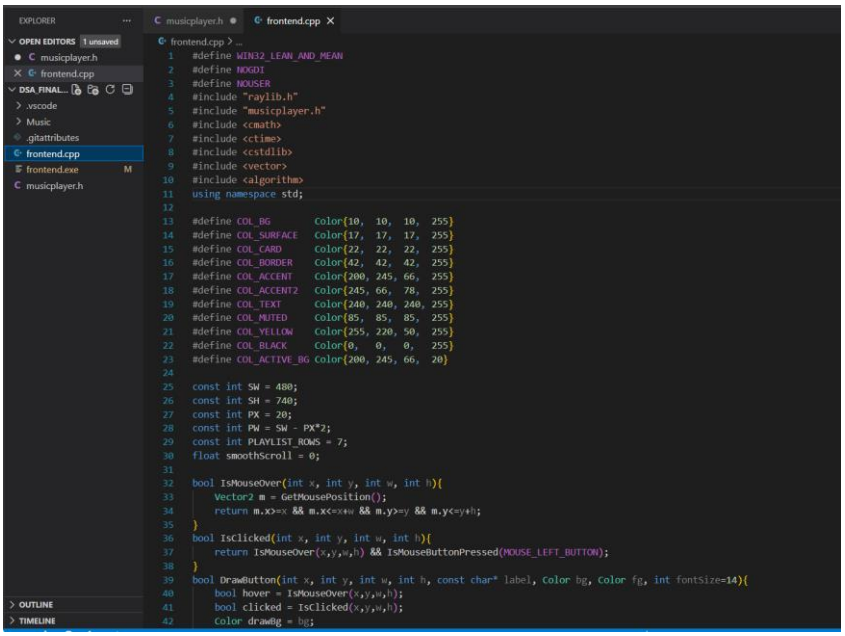


Figure- FrontEnd(Code)

Appendix B

musicplayer.h ● frontend.cpp

musicplayer.h > MusicPlayer > MusicPlayer()

```
169     }
170
171
172     void seek(float delta){
173         if(!isPlaying || music.getDuration().asSeconds() == 0) return;
174         float total = music.getDuration().asSeconds();
175         float cur = music.getPlayingOffset().asSeconds();
176         float newPos = cur + delta * total * 0.1f;
177         music.setPlayingOffset(sf::seconds(newPos));
178         logMsg = "SEEKED TO " + to_string((int)newPos) + "s";
179     }
180
181     // Returns 0.0 to 1.0 real progress
182     float getProgress(){
183         if(!isPlaying || music.getDuration().asSeconds() == 0) return 0.0f;
184         return music.getPlayingOffset().asSeconds() / music.getDuration().asSeconds();
185     }
186
187     string getTimeElapsed(){
188         int s = (int)music.getPlayingOffset().asSeconds();
189         return to_string(s/60) + ":" + (s%60<10?"0":"") + to_string(s%60);
190     }
191
192     string getTimeDuration(){
193         int s = (int)music.getDuration().asSeconds();
194         if(s == 0) return "0:00";
195         return to_string(s/60) + ":" + (s%60<10?"0":"") + to_string(s%60);
196     }
197
198     // Volume 0-100
199     void setVolume(float v){
200         v = max(0.0f, min(100.0f, v));
201         music.setVolume(v);
202     }
203
204     float getVolume(){
205         return music.getVolume();
206     }
207
208     // Call every frame — auto advance when song ends
209     void update(){
210         if(isPlaying && !isPaused){
211             if(music.getStatus() == sf::SoundSource::Status::Stopped){
212                 current = current->next;
213                 loadAndPlay();
214             }
215         }
216     }
217
218     vector<pair<string,bool>> getSongs(){
219         vector<pair<string,bool>> songs;
220         if(!head) return songs;
221         node* temp = head;
222         do{
223             songs.push_back({temp->song, temp==current});
224             temp = temp->next;
225         } while(temp != head);
226         return songs;
227     }
228
229     ~MusicPlayer(){
230         music.stop();
231         if(!head) return;
232         node* temp = head;
233         do{
234             node* next = temp->next;
235             delete temp;
236             temp = next;
237         } while(temp != head);
238     }
239 };
```

Fig;- Time and Volume(Backend Code)