PROBLEM DEFINITION

The project aims to create a music playlist manager that leverages a doubly linked list data structure to help users organize and manage their music collections. The primary goal is to develop a user-friendly application with core functionalities for managing music playlists, including adding, removing, and navigating through music tracks.

The problem definition encompasses the following key aspects:
1.Playlist Creation and Management: Users should be able to create multiple playlists, each identified by a unique name, and manage them effectively.

2.Music Track Information: The application should store essential information for each music track, such as song title. To extend its functionality, additional metadata like artist, album, duration, and genre can be considered.

3.Adding and Removing Music Tracks: Users should be able to add and remove music tracks from their library, and these tracks should be organized within playlists.

4.Playback Control: Basic playback control features, including playing the current song, skipping to the next track, and rewinding to the previous track, should be implemented.

5.Error Handling: The application must handle common scenarios such as adding duplicate tracks, attempting to play songs in empty playlists, or encountering errors when manipulating the playlist.

6.User Interface: While the code is currently console-based, further development can include the creation of a graphical user interface (GUI) for a more user-friendly interaction.

7.File I/O: Users should be able to save and load playlists to and from files, ensuring their music libraries persist between sessions.

8.Efficiency: Optimizing the application's performance by considering more efficient data structures and algorithms can enhance the user experience, especially with large music libraries.

The project, as defined, will serve as an introduction to linked lists and provide a practical application for data structure and algorithm concepts in the context of a digital music library. Successful completion of the project will result in a user-friendly music playlist manager, allowing users to organize, manage, and enjoy their music collections efficiently.

PROBLEM EXPLANATION

The project entails the development of a Music Playlist Manager using a doubly linked list data structure, a versatile and efficient way to manage and organize music collections. This digital music library manager aims to provide users with a user-friendly platform for managing their music playlists and tracks.

The project's core functionalities and features are outlined as follows:

1.Playlist Creation and Management: Users will have the ability to create multiple playlists, each identified by a unique name. These playlists will serve as containers for organizing music tracks.

2.Music Track Information: The application will store crucial information for each music track, including its title. To enhance the user experience, additional metadata like artist, album, duration, and genre can be considered for inclusion.

3.Adding and Removing Music Tracks: Users will be able to add new music tracks to their library and remove existing ones as needed. These tracks will be organized within playlists, allowing for a structured music collection.

4.Playback Control: Basic playback control features will be implemented, enabling users to play the current song, skip to the next track, and rewind to the previous one.

5.Error Handling: The application will incorporate error handling mechanisms to manage common scenarios, such as preventing the addition of duplicate tracks, handling attempts to play songs in empty playlists, and addressing errors during playlist manipulation.

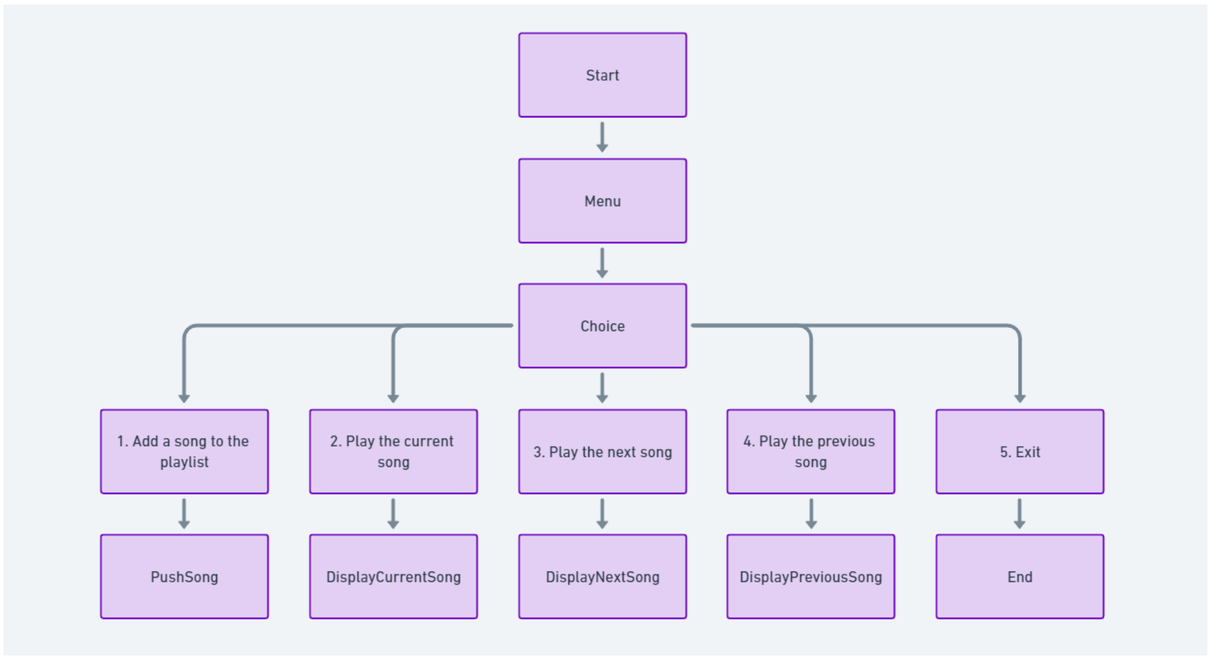
6.User Interface: While the code provided is console-based, the project can be extended to include the development of a graphical user interface (GUI) to make the application more user-friendly and visually appealing.

7.File I/O: Users will have the option to save their playlists and music libraries to files and load them from files. This ensures that their music collections persist between sessions.

8.Efficiency: To enhance the user experience, optimizing the application's performance can be explored by considering more efficient data structures and algorithms. This is especially important when dealing with larger music libraries.

The project's significance lies in its practical application of data structure and algorithm concepts within the context of a digital music library. It introduces users to the use of linked lists as a fundamental data structure for managing dynamic collections of data efficiently. Upon successful completion, users will have access to a fully functional music playlist manager that empowers them to organize, manage, and enjoy their music collections seamlessly.

DIAGRAM:


Start
Menu
Choice
1. Add a song to the playlist
2. Play the current song
3. Play the next song
4. Play the previous song
5. Exit
PushSong
DisplayCurrentSong
DisplayNextSong
DisplayPreviousSong
End

DATA STRUCTURE:

A linked list is a fundamental data structure used in computer science and programming to organize and store a collection of data elements. Unlike arrays, which store elements in contiguous memory locations, linked lists consist of nodes, each containing both data and a reference (or link) to the next node in the sequence. This structure allows for dynamic memory allocation and efficient insertion and deletion operations.

Key operations associated with a stack linked list structure are:

1. **Insertion:** This operation involves adding a new element (node) to the linked list. Insertions can occur at the beginning, end, or any position within the list.
2. **Deletion:** Deletion is the process of removing a node from the linked list. You can delete nodes from the beginning, end, or any specific position in the list.
3. **Traversal:** Traversal is the act of visiting and inspecting each node in the linked list sequentially. It's essential for accessing, modifying, or displaying the elements.
4. **Search:** Searching involves finding a specific element or node within the linked list based on a particular value or criteria.
5. **Access (or Get):** This operation allows you to retrieve the value of a node at a specific position in the linked list, given an index or other criteria.
6. **Update (or Modify):** Updating or modifying a node entails changing the value or content of a specific node in the linked list.
7. **Size (or Length):** The size operation returns the number of nodes in the linked list, indicating its length or size.
8. **Concatenation (or Merge):** This operation combines two linked lists into a single linked list, typically by connecting the last node of the first list to the first node of the second list.
9. **Reversal:** Reversing a linked list involves changing the order of nodes, flipping the direction of the pointers, and making the last node the new head of the list.

Applications of Linked list Explained

Linked lists find applications in various domains and scenarios:

1.Dynamic Memory Allocation: Linked lists enable efficient allocation and deallocation of memory, making them valuable in managing resources in dynamic environments.

2.Data Structure Building Blocks: They serve as the basis for building other data structures like stacks, queues, and hash tables, contributing to efficient data management.

3.Text Editors and Word Processors: Linked lists are used to represent text content, allowing for efficient insertion and deletion of characters or words.

4.Music and Playlist Management: Linked lists facilitate the creation and management of music playlists, supporting operations like adding, removing, and reordering songs.

5.Browser History: Browsers use linked lists to manage the history of visited web pages, allowing easy navigation backward and forward.

6.Task Management: To-do lists and task management applications employ linked lists for adding, removing, and organizing tasks.

7.Compiler Symbol Tables: In compilers, linked lists are utilized to manage symbol tables, making symbol lookups and updates efficient.

8.File Systems: Linked lists help represent the hierarchical structure of directories and files in file systems.

9.Image Processing: Pixels in images can be organized using linked lists for image manipulation and processing.

10.Undo/Redo Functionality: Software applications use linked lists to implement undo and redo features for tracking changes and actions.

IMPLEMENTATION OF CODE:

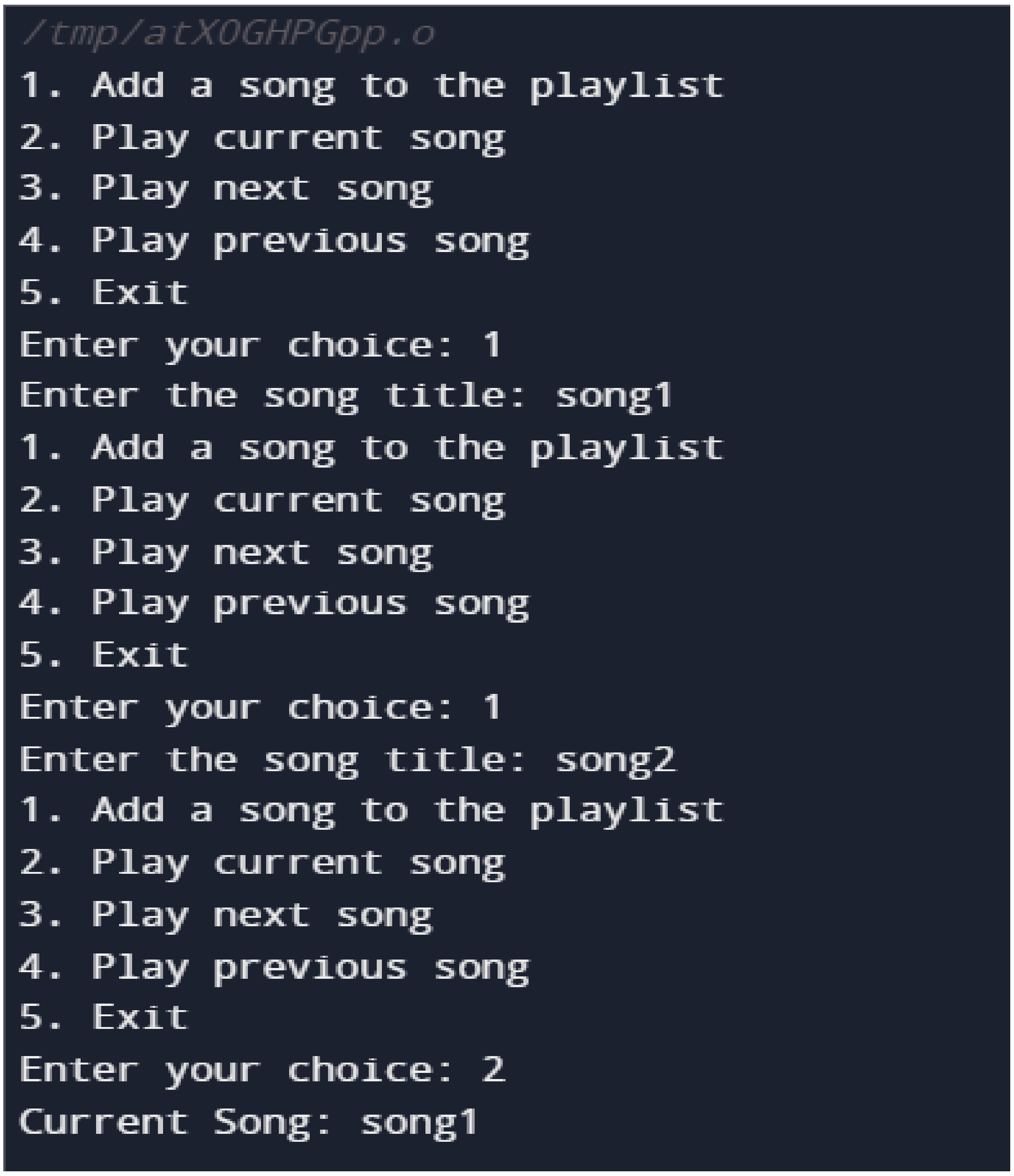
```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Define a structure for a music node

typedef struct MusicNode {

   char title[100];

   struct MusicNode* next;

   struct MusicNode* prev;

} MusicNode;


// Create a global pointer for the current song in the playlist

MusicNode* currentSong = NULL;


// Function to add a new song to the playlist

void addSong(MusicNode** playlist, char title[]) {

   MusicNode* newSong = (MusicNode*)malloc(sizeof(MusicNode));

   strncpy(newSong->title, title, 100);
```

```
    newSong->next = NULL;

    newSong->prev = NULL;


    if (*playlist == NULL) {

        *playlist = newSong;

        currentSong = *playlist;

    } else {

        MusicNode* current = *playlist;

        while (current->next != NULL) {

            current = current->next;

        }

        current->next = newSong;

        newSong->prev = current;

    }

}


// Function to display the current song

void displayCurrentSong() {

    if (currentSong != NULL) {
        printf("Current Song: %s\n", currentSong->title);
    } else {
        printf("Playlist is empty.\n");
    }
}

// Function to play the next song
void playNextSong() {
    if (currentSong != NULL && currentSong->next != NULL) {
        currentSong = currentSong->next;
        printf("Playing Next Song: %s\n", currentSong->title);
    } else {
        printf("No next song in the playlist.\n");
    }
}

// Function to play the previous song
void playPreviousSong() {
    if (currentSong != NULL && currentSong->prev != NULL) {
        currentSong = currentSong->prev;

        printf("Playing Previous Song: %s\n", currentSong->title);

    } else {

        printf("No previous song in the playlist.\n");

    }

}


// Function to free the memory of the playlist

void freePlaylist(MusicNode** playlist) {

    MusicNode* current = *playlist;

    while (current != NULL) {

        MusicNode* temp = current;

        current = current->next;

        free(temp);

    }

    *playlist = NULL;

}


int main() {
```

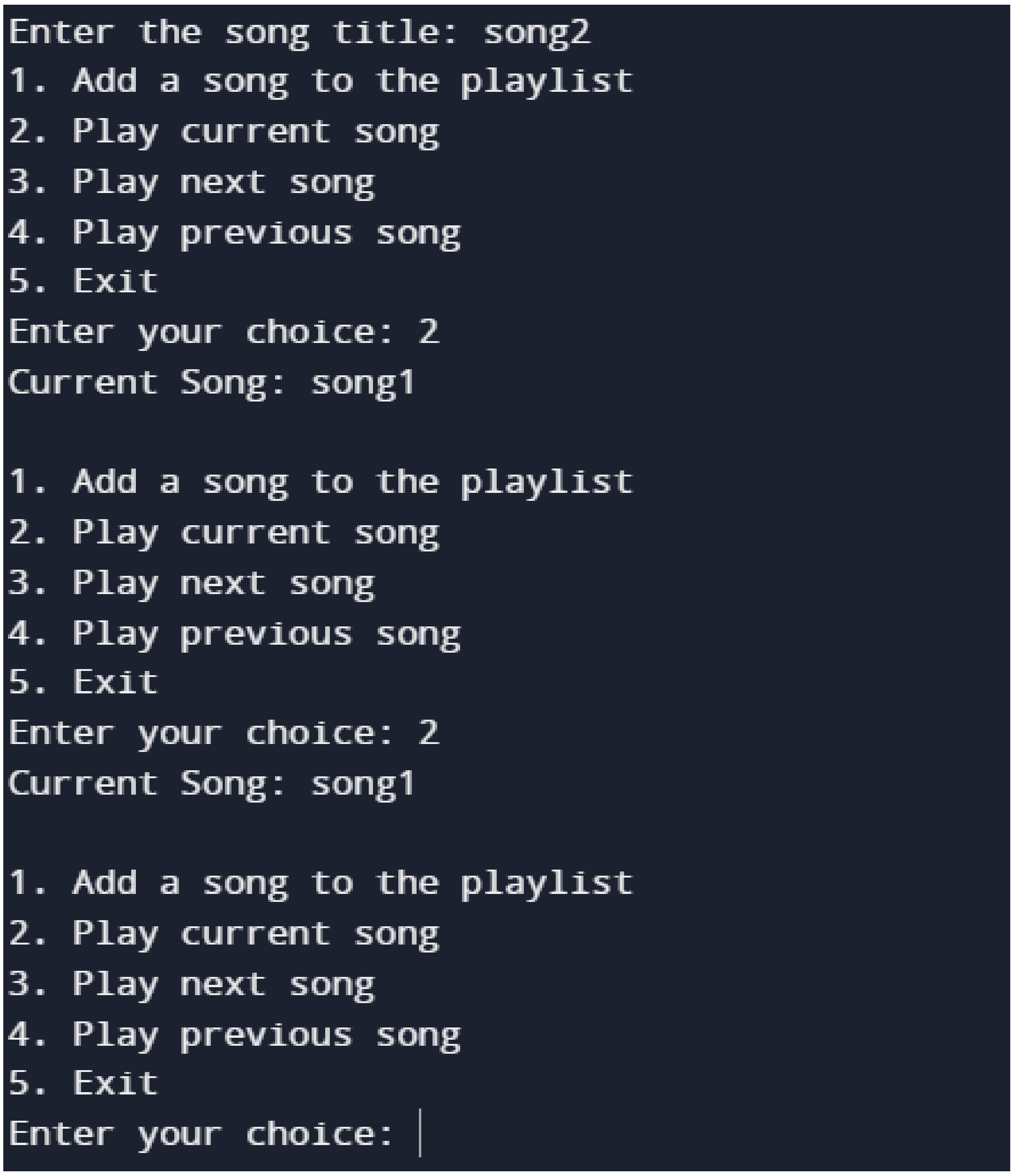
```
MusicNode* playlist = NULL;


int choice;

char title[100];


while (1) {

    printf("\n1. Add a song to the playlist\n");

    printf("2. Play current song\n");

    printf("3. Play next song\n");

    printf("4. Play previous song\n");

    printf("5. Exit\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);


    switch (choice) {

        case 1:

            printf("Enter the song title: ");

            scanf("%s", title);

            addSong(&playlist, title);
```

```
                break;
            case 2:
                displayCurrentSong();
                break;
            case 3:
                playNextSong();
                break;
            case 4:
                playPreviousSong();
                break;
            case 5:
                freePlaylist(&playlist);
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    return 0;
}
```

OUTPUT:



```
/tmp/atXOGHPGpp.o
1. Add a song to the playlist
2. Play current song
3. Play next song
4. Play previous song
5. Exit
Enter your choice: 1
Enter the song title: song1
1. Add a song to the playlist
2. Play current song
3. Play next song
4. Play previous song
5. Exit
Enter your choice: 1
Enter the song title: song2
1. Add a song to the playlist
2. Play current song
3. Play next song
4. Play previous song
5. Exit
Enter your choice: 2
Current Song: song1
```



```
Enter the song title: song2
1. Add a song to the playlist
2. Play current song
3. Play next song
4. Play previous song
5. Exit
Enter your choice: 2
Current Song: song1

1. Add a song to the playlist
2. Play current song
3. Play next song
4. Play previous song
5. Exit
Enter your choice: 2
Current Song: song1

1. Add a song to the playlist
2. Play current song
3. Play next song
4. Play previous song
5. Exit
Enter your choice:
```

CONCLUSION:

In summary, the code serves as a basic prototype for a music playlist manager using a linked list. It demonstrates fundamental functionalities but requires substantial refinement and expansion. Enhancements needed include the addition of a user-friendly graphical interface, robust error handling, support for multiple playlists, file I/O for data persistence, music metadata, and efficiency optimizations. User feedback, cross-platform compatibility, and security measures are essential considerations. Performance optimization and features like playlist sharing and social integration can elevate the user experience. Clear documentation is vital for users and developers. In essence, with these enhancements, the code can evolve into a more comprehensive and user-centric music management application.