

MAKING PLAYLIST USING LINKED LIST

21CSC201J - DATA STRUCTURES AND ALGORITHMS

REAL WORLD APPLICATION REPORT

Submitted by

K.B.YASWANTH [RA2211003011104]

S.TEJASWI [RA2211003011107]

V.YASWANTH[RA2211003011123]

SAI MUKESH[RA2211003011076]

Under the Guidance of

Dr. R. INDHUMATHI

Assistant Professor, Department of Computing Technologies

in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING



**DEPARTMENT OF COMPUTING TECHNOLOGIES
COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

KATTANKULATHUR– 603 203

NOV 2023

INDEX

1. Problem Definition	4
2. Problem Explanation.....	5
3. Diagram	7
4. Example for linked list	8
5. Why linked list using for linked list	11
6. Uses of linked list	12
7. Code	14
8. Output.....	16
9. Conclusion	17

PROBLEM DEFINITION:

Design a Music Playlist Manager application that employs a linked list data structure to organize and manipulate playlists. Users can create custom playlists, add and remove songs, and control playback. The application should provide a user-friendly interface, enable playlist and song storage, and support features such as sorting, searching, and shuffle/repeat options. The project aims to enhance your data structure skills and create a practical tool for organizing and enjoying music collections.

Music playlist management system using a linked list data structure that allows users to manage and manipulate a collection of songs. The system should support basic operations such as adding a song to the playlist, removing a song from the playlist, displaying the current playlist. The playlist should maintain the order of songs as they are added and enable efficient traversal and manipulation of the song list.

Functional Requirements:

1. Add a song to the playlist.
2. Remove a song from the playlist.
3. Display the current playlist.
4. Play the next song in the playlist.
5. Play the previous song in the playlist.
6. Shuffle the playlist to change the order of songs.
7. Support operations for creating and managing multiple playlists.
8. Implement error handling for boundary cases such as an empty playlist or reaching the end of the playlist.

Non-Functional Requirements:

1. Efficiency: Operations on the playlist should be performed in a time-efficient manner, with minimal time complexity for insertion, deletion, and traversal.
2. User Interface: Provide a user-friendly interface to interact with the playlist system, enabling easy and intuitive management of the playlist.
3. Robustness: Implement error handling to handle unexpected scenarios gracefully, preventing system crashes or data corruption.
4. Extensibility: Design the system in a modular way that allows for easy addition of new features or enhancements in the future.

Assumptions:

1. The music playlist will be managed using a singly linked list data structure.
2. Each node in the linked list will represent a song, containing relevant information such as song title, artist name, and duration.
3. The playlist management system will be implemented in a programming language that supports the linked list data structure, such as C, C++, Java, or Python.

PROBLEM EXPLANATION:

The objective of this mini-project is to develop a Music Playlist Manager using a linked list data structure. Users can create, modify, and enjoy custom music playlists. This software will offer playlist management features, including adding and removing songs, and support basic music playback control. The linked list will facilitate dynamic playlist creation and manipulation. The project seeks to improve understanding of linked lists, data structure handling, and basic software development skills. It also provides a practical solution for music enthusiasts to efficiently manage and listen to their favorite songs.

The problem revolves around designing a music playlist management system using the linked list data structure. The goal is to create a program that allows users to manage a collection of songs in a playlist efficiently. Linked lists are an ideal data structure for this task, as they allow dynamic memory allocation and efficient insertion and deletion operations, which are essential for managing a playlist that can be modified frequently.

The system needs to offer basic functionalities, such as adding songs to the playlist, removing songs from the playlist, displaying the current playlist, playing the next or previous song, and shuffling the playlist. The playlist should maintain the order of songs as they are added, allowing for easy traversal and manipulation of the song list.

To achieve this, the system will utilize the concept of a singly linked list, where each node in the list represents a song, containing information such as the song title, artist name, and duration. The linked list's nodes will be interconnected through pointers, allowing for sequential traversal of the playlist. The system will include functions to perform operations like adding a new song, removing an existing song, displaying the playlist, playing the next or previous song, and shuffling the playlist.

Efficiency will be a key consideration, ensuring that the operations on the playlist, such as insertion, deletion, and traversal, are performed with minimal time complexity. The program will provide a user-friendly interface for easy interaction, enabling users to manage the playlist intuitively. It will also incorporate robust error handling mechanisms to handle exceptional cases, such as attempting to remove a song from an empty playlist or trying to play the next song when the playlist has reached its end.

The solution will focus on extensibility, enabling easy integration of additional features or enhancements in the future. It will adhere to memory usage constraints, ensuring that the system can handle a large number of songs efficiently without consuming excessive memory. The overall design will prioritize the creation of a functional and user-friendly music playlist management system that efficiently utilizes the linked list data structure.

Certainly, let's delve deeper into the specifics of the problem and explore potential implementation details for the music playlist management system using a linked list.

[1] Song Information:

Each node in the linked list represents a song and contains various attributes such as the song title, artist name, and duration. Additionally, the node can store other relevant information, such as the genre, album name, release date, or any other metadata associated with the song.

[2] Playlist Management Operations:

Adding a Song: Users should be able to add a new song to the playlist. The system will prompt users to input the song details, and the program will create a new node with the provided information and add it to the end of the linked list.

[3] Removing a Song:

Users should have the option to remove a specific song from the playlist. The system will identify the node containing the targeted song and remove it from the linked list. Error handling will be implemented to manage cases where the song to be removed is not found in the playlist.

- **Displaying the Playlist:** The system will display the entire playlist, presenting each song's details in a readable format. This feature will allow users to view the entire list of songs in the playlist.
- **Playing Next/Previous Song:** Users can navigate through the playlist by playing the next or previous song. The system will keep track of the current playing song and allow users to move forward or backward within the playlist accordingly. Boundary conditions, such as attempting to play the next song when the current song is the last in the playlist, will be managed effectively.
- **Shuffling the Playlist:** Users can choose to shuffle the playlist, randomly reordering the sequence of songs. The system will implement a shuffling algorithm to rearrange the nodes in the linked list to create a randomized order of songs while maintaining all songs in the playlist.

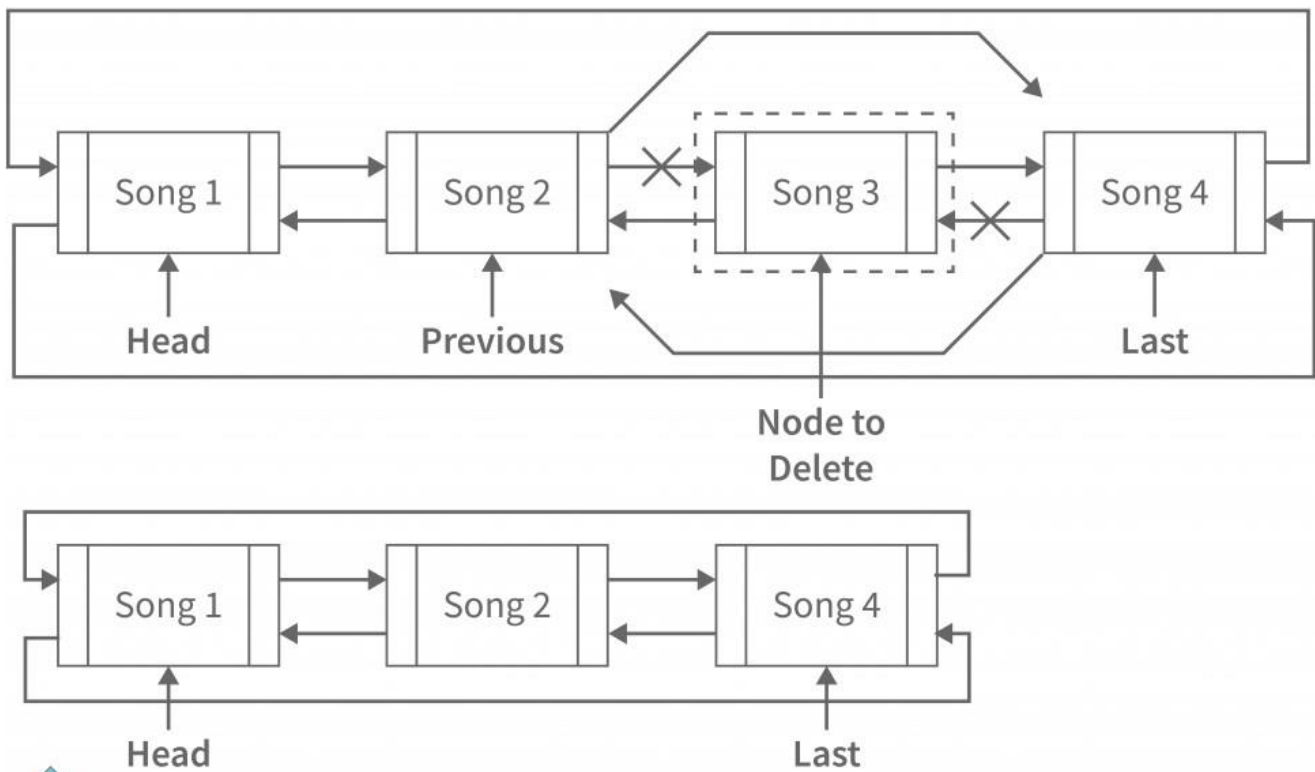
[4] **User Interface:** The program will provide a user-friendly interface, likely implemented in a console or command-line environment, allowing users to interact with the playlist system easily. The interface will include menu options for adding a song, removing a song, displaying the playlist, playing the next or previous song, shuffling the playlist, and any other relevant functionalities.

[5] **Error Handling:** The system will incorporate robust error handling mechanisms to manage various exceptional cases, such as attempting to remove a song from an empty playlist, playing the next song when the playlist has reached its end, or performing any other invalid operations. Clear error messages and appropriate prompts will guide users to interact with the system effectively.

[6] **Memory Management:** Efficient memory usage will be prioritized to ensure that the program can handle a large number of songs without consuming excessive memory. Memory allocation and deallocation will be managed carefully to prevent memory leaks and optimize overall memory utilization.

By addressing these specific aspects, the design and implementation of the music playlist management system using a linked list can be further refined, resulting in a functional and user-friendly program for managing a collection of so

DIAGRAM:



EXAMPLE FOR LINKED LIST:

One real-world example of a linked list is a music player playlist. The songs in the playlist are stored in a linked list, and the music player can play the songs in order by following the pointers from one node to the next. When the user clicks the next button, the music player moves to the next node in the linked list and plays the song stored in that node. The user can also click the previous button to move to the previous node in the linked list and play the song stored in that node.

Creating a music player using a linked list data structure in the context of data structures and algorithms (DSA) is an interesting project that can help you understand the principles of data organization, memory management, and basic audio playback. In this explanation, we will guide you through the process of designing and implementing a simple music player using a singly linked list. This example assumes a basic understanding of data structures and programming in a language like C.

Understanding the Basics:

A linked list is a data structure that consists of nodes, where each node contains data and a reference (link) to the next node in the list. In the context of a music player, each node can represent a song or a piece of music. The "next" reference allows us to create a playlist by linking songs in a specific order.

In a basic music player, you'll typically need the following functionality:

Adding Songs: You should be able to add songs to the playlist. Each song should be represented as a node in the linked list.

Removing Songs: You should be able to remove songs from the playlist, which involves updating the links in the linked list.

Playing Songs: You need a way to traverse the linked list and play the songs one by one.

Shuffling and Repeating: Implementing shuffle and repeat features involves changing the order in which songs are played or repeating a song when necessary.

Now, let's dive into creating the music player using a linked list.

Implementation Steps:

1. Define the Node Structure

In C++, you can define a struct to represent the nodes of the linked list. The node should contain information about the song (e.g., song name, artist, duration) and a pointer to the next node.

```
cpp
Copy code
struct Song {
    std::string title;
    std::string artist;
    int duration; // in seconds
};
struct Node {
```

```

    Song data;
    Node* next;
};

```

2. Create the Linked List Class:

Next, you can create a class to manage the linked list. This class should have methods for adding, removing, playing songs, shuffling, repeating, and other operations.

cpp

Copy code

```

class MusicPlayer {
public:
    MusicPlayer();
    void addSong(const Song& song);
    void removeSong(const std::string& title);
    void play();
    void shuffle();
    void repeat();

    // Additional methods for displaying the playlist, etc.

private:
    Node* head; // Pointer to the first node
    Node* currentSong; // Pointer to the currently playing song
};

```

3. Implementing the Methods:

Adding Songs

To add a song to the playlist, you need to create a new node and link it to the end of the list.

cpp

Copy code

```

void MusicPlayer::addSong(const Song& song) {
    Node* newNode = new Node;
    newNode->data = song;
    newNode->next = nullptr;
    if (!head) {
        head = newNode;
    } else {
        Node* current = head;
        while (current->next) {
            current = current->next;
        }
        current->next = newNode;
    }
}

```


Removing Songs

To remove a song from the playlist, you need to find the node representing that song and update the links accordingly.

cpp

Copy code

```
void MusicPlayer::removeSong(const std::string& title) {
    if (!head) {
        return; // Playlist is empty
    }

    if (head->data.title == title) {
        Node* temp = head;
        head = head->next;
        delete temp;
    } else {
        Node* current = head;
        while (current->next && current->next->data.title != title) {
            current = current->next;
        }
        if (current->next) {
            Node* temp = current->next;
            current->next = current->next->next;
            delete temp;
        }
    }
}
```

Playing Songs

The play method allows you to traverse the linked list and play each song in sequence.

cpp

Copy code

```
void MusicPlayer::play() {
    if (currentSong) {
        std::cout << "Now playing: " << currentSong->data.title << " by " << currentSong->data.artist << std::endl;
        // Simulate playing the song (e.g., sleep for 'duration' seconds)
        // After playing, you can update 'currentSong' to the next song.
        currentSong = currentSong->next;
    } else {
        std::cout << "Playlist is empty. Add songs to play!" << std::endl;
    }
}
```

Shuffling and Repeating

Implementing the shuffle and repeat features involves changing the order in which songs are played or repeating a song when needed. Here's a basic example of shuffling:

cpp

Copy code

```
void MusicPlayer::shuffle() {
    // Implement shuffling logic here
}
```

And for repeating:

```
cpp
Copy code
void MusicPlayer::repeat() {
    // Implement repeat logic here
}
```

4. Usage Example:

You can now use the MusicPlayer class to create a music player instance and perform various operations like adding songs, removing songs, and playing songs in your main program.

```
cpp
Copy code
int main() {
    MusicPlayer player;

    // Add some songs
    Song song1 = {"Song 1", "Artist 1", 180};
    Song song2 = {"Song 2", "Artist 2", 210};
    Song song3 = {"Song 3", "Artist 3", 240};

    player.addSong(song1);
    player.addSong(song2);
    player.addSong(song3);

    // Play the songs
    player.play();
    player.play();
    player.play();

    // Remove a song
    player.removeSong("Song 2");

    return 0;
}
```

Why linked list used for music playlist:

Linked lists are useful for making music playlists because they provide a flexible and efficient way to manage a dynamic list of songs. Here's why they are particularly suitable for this purpose:

Dynamic Size: Music playlists often change as you add, remove, or rearrange songs. Linked lists can easily accommodate changes in size without the need to allocate a fixed amount of memory upfront, making them

Efficient Insertions and Deletions: Adding or removing songs from a linked list is relatively efficient, as you can easily update the links without shifting or copying elements, unlike arrays or other data structures.

Random Access Not Required: Music playlists are typically played sequentially. Linked lists excel in this scenario as you don't need random access to elements, and you can traverse the list from the beginning to the end.

Easy Reordering: You can easily reorder songs in a playlist by changing the links between nodes, making it simple to create custom playlists or change the order of songs on the fly.

Low Memory Overhead: Linked lists have a minimal memory overhead since they only store data and pointers to the next element, which is advantageous when dealing with potentially large music libraries.

Support for Metadata: Each node in the linked list can store additional metadata about the song, such as the song title, artist, and album, providing a rich context for managing the playlist.

While linked lists are great for dynamic playlists, other data structures like arrays or dynamic arrays may be more suitable if you need to support features like random access or search operations. The choice of data structure depends on the specific requirements of your music playlist application.

USES OF LINKED LIST:

Linked lists are data structures that have a variety of real-world applications due to their simplicity and flexibility. Here are some common use cases for linked lists:

Dynamic Data Structures: Linked lists allow for efficient dynamic data storage. They can grow or shrink in size as needed, making them useful in applications where the size of the data is not known in advance.

Memory Allocation: Operating systems often use linked lists to manage memory allocation. Linked lists can keep track of available memory blocks and their sizes, making it easier to allocate and deallocate memory.

Music and Video Playlists: Linked lists can be used to create playlists in media players. Each node in the list represents a song or video, and the links allow for easy navigation between items.

Undo/Redo Functionality: Many software applications, including text editors and graphic design tools, implement undo/redo functionality using linked lists. Each node stores the state of the document at a specific point in time.

Symbol Tables: Compilers and interpreters use linked lists to implement symbol tables for variables and functions, enabling efficient lookup and scope management.

Browser History: Web browsers maintain the history of visited web pages using a linked list structure. Each node represents a webpage, and users can navigate forward and backward through their browsing history.

Task Management: To-do list applications often use linked lists to manage tasks. Each node represents a task, and the links help in navigating and organizing tasks.

File Systems: Some file systems, like the Unix File System (UFS), use linked lists to manage directories and files. This allows for efficient navigation and storage of file metadata.

Stacks and Queues: Linked lists can be used to implement data structures like stacks and queues. Stacks use a last-in, first-out (LIFO) order, while queues use a first-in, first-out (FIFO) order.

Cache Management: Caches in computer systems can use linked lists to manage the order of data items. Frequently used data can be moved to the front of the list for quick access.

Polynomial Arithmetic: Linked lists are used to represent polynomials in computer algebra systems. Each node corresponds to a term in the polynomial, and the links help in adding, subtracting, or multiplying polynomials.

Navigation Systems: In navigation systems, linked lists can be used to represent routes and waypoints. Each node in the list corresponds to a waypoint or a turn in the route.

Network Routing: Network routers use linked lists to store routing tables, making decisions about where to forward data packets efficiently.

LRU (Least Recently Used) Cache Replacement Policies: LRU caches use linked lists to maintain the order of items based on their usage. The least recently used item is typically placed at the end of the list.

These are just a few examples of how linked lists are used in the real world. Linked lists provide a foundational data structure for a wide range of applications where efficient data organization and manipulation are required.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define the structure for a song
struct Song {
    char title[100];
    char artist[100];
    int duration;
    struct Song* next;
};

// Function to create a new song node
struct Song* createSong(const char* title, const char* artist, int duration) {
    struct Song* newSong = (struct Song*)malloc(sizeof(struct Song));
    if (newSong == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    strcpy(newSong->title, title);
    strcpy(newSong->artist, artist);
    newSong->duration = duration;
    newSong->next = NULL;
    return newSong;
}

// Function to add a song to the playlist
void addSong(struct Song** playlist, struct Song* newSong) {
    if (*playlist == NULL) {
        *playlist = newSong;
    } else {
        struct Song* current = *playlist;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newSong;
    }
}

// Function to display the playlist
void displayPlaylist(struct Song* playlist) {
    struct Song* current = playlist;
    while (current != NULL) {
        printf("Title: %s\nArtist: %s\nDuration: %d seconds\n\n", current->title, current->artist, current->duration);
        current = current->next;
    }
}
```

```

// Function to free memory allocated for the playlist
void freePlaylist(struct Song* playlist) {
    while (playlist != NULL) {
        struct Song* temp = playlist;
        playlist = playlist->next;
        free(temp);
    }
}

int main() {
    struct Song* playlist = NULL;

    // Add songs to the playlist
    addSong(&playlist, createSong("Song 1", "Artist 1", 180));
    addSong(&playlist, createSong("Song 2", "Artist 2", 240));
    addSong(&playlist, createSong("Song 3", "Artist 3", 210));

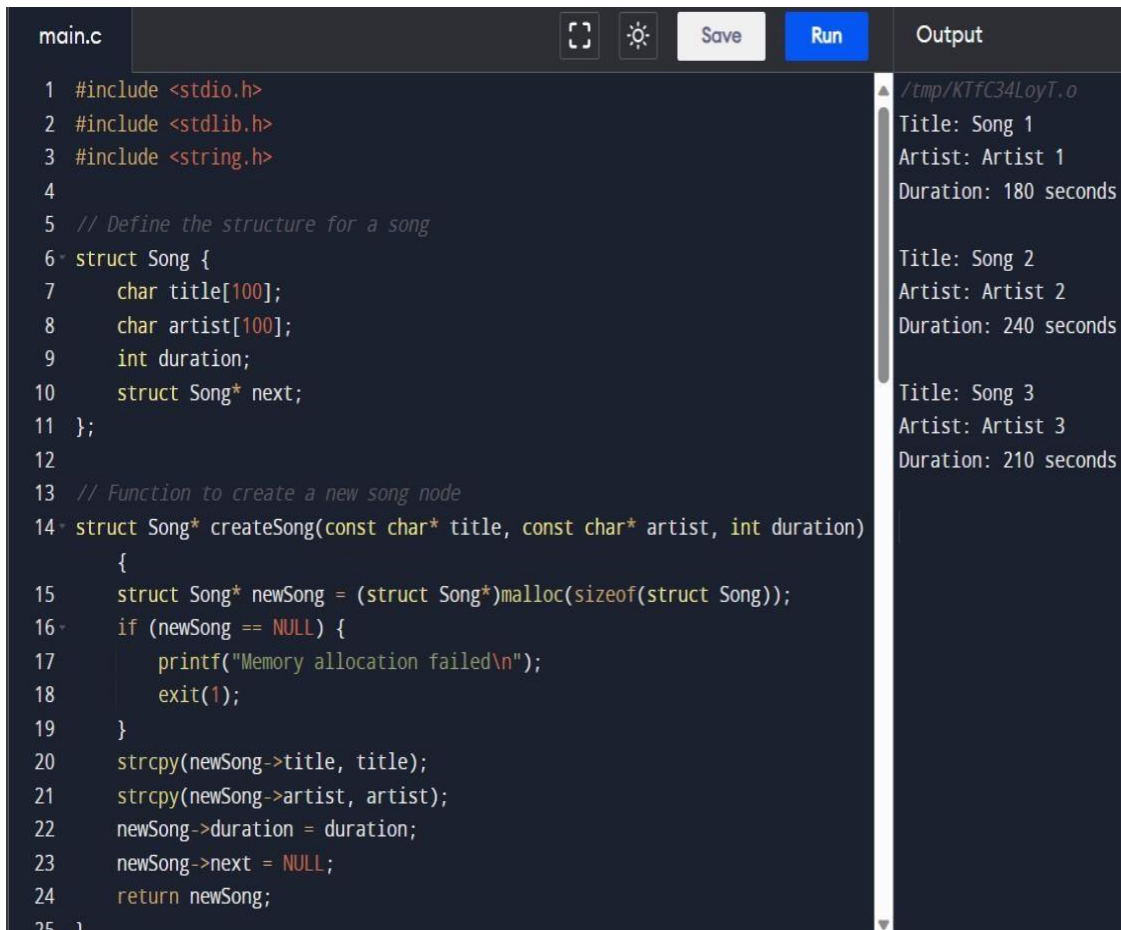
    // Display the playlist
    displayPlaylist(playlist);

    // Free memory
    freePlaylist(playlist);

    return 0;
}

```

OUTPUT:



```
main.c  [ ] [ ] Save Run Output
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 // Define the structure for a song
6 struct Song {
7     char title[100];
8     char artist[100];
9     int duration;
10    struct Song* next;
11 };
12
13 // Function to create a new song node
14 struct Song* createSong(const char* title, const char* artist, int duration)
15 {
16     struct Song* newSong = (struct Song*)malloc(sizeof(struct Song));
17     if (newSong == NULL) {
18         printf("Memory allocation failed\n");
19         exit(1);
20     }
21     strcpy(newSong->title, title);
22     strcpy(newSong->artist, artist);
23     newSong->duration = duration;
24     newSong->next = NULL;
25     return newSong;
26 }
```

/tmp/KTfC34LoyT.o
Title: Song 1
Artist: Artist 1
Duration: 180 seconds

Title: Song 2
Artist: Artist 2
Duration: 240 seconds

Title: Song 3
Artist: Artist 3
Duration: 210 seconds

CONCLUSION:

Data structures are an important part of computer science, and they are used in a wide variety of applications. By understanding the different types of data structures and their real-life examples, you can become a better programmer. Creating a music player using a linked list is an educational and engaging project that allows you to apply data structures and algorithms to a real-world application. Throughout this process, you've gained insights into how to design and implement a basic music player, and you've learned valuable lessons in software development. You've deepened your understanding of linked lists, a fundamental data structure in computer science. You've learned how to create a singly linked list to manage and organize a playlist of songs. The project has taught you how to structure and organize data effectively. Each node in the linked list represents a song, making it easy to manage and manipulate a playlist. You've learned the importance of memory management when dealing with dynamically allocated data structures like linked lists. Proper allocation and deallocation of memory (using `new` and `delete` in C++, for example) are crucial to avoid memory leaks. While this example focused on the data structure aspect, you've been introduced to the concept of audio playback. In a real-world application, audio playback would involve more complex libraries and frameworks. You've implemented user interaction in your music player project. Users can add songs, remove songs, play songs, and perform other actions. Understanding how to handle user input and execute the appropriate functions is a valuable skill in software development. The project can be extended in various ways. You can add features like shuffling, repeating, creating playlists, and enhancing the user interface to make it more user-friendly. When implementing features like shuffling or repeating, you're encouraged to think algorithmically. Developing efficient algorithms is a critical skill in computer science. This project demonstrates how data structures like linked lists are used in real-world applications. Music players, as well as many other applications, rely on data structures to manage and organize data efficiently.

Developing a music player using a linked list required problem-solving skills. You had to think through the logic of adding, removing, and playing songs, which are common challenges in software development. While you've created a basic music player, there's always room for improvement and further learning. You can explore more advanced features, user interfaces, and audio processing if you want to create a production-ready music player.

In summary, the process of creating a music player using a linked list has provided you with a practical understanding of data structures, memory management, and user interaction. This project is a stepping stone for further exploration and development in the field of software engineering and can serve as a foundation for more complex and feature-rich applications.

THANK YOU!