National University
Of Computer and Emerging Sciences

PROJECT REPORT

TOPIC: Music Hub: Recommending Songs based on User Behavior

GROUP MEMBERS :

Dania Zehra	22K-4152	[BSCS-3A]
Vaniya Rehan	22K-4501	[BSCS-3A]
Arisha Rehan Chotani	22K-4569	[BSCS-3A]

INSTRUCTOR : Sir Jawwad Shamsi

Project Description

The project, titled "MusicHub," is a console-based music application that allows users to manage playlists, explore song recommendations, and interact with their profiles as well. The application incorporates features such as playlist management, user authentication, and song recommendation based on user behavior.

Methodology (including diagram)

Object-Oriented Approach

The project employs classes for songs, playlists, and users, facilitating a structured and scalable design. Each class encapsulates specific functionalities, contributing to a well-organized codebase.

Playlist Management

Linked lists are utilized for efficient playlist management, allowing users to seamlessly add, remove, shuffle, and sort songs within playlists. The modular design ensures flexibility and ease of maintenance.

User Authentication

To secure user information, a hash table is implemented for user authentication. This ensures the safe storage and retrieval of user credentials. Boost C++ libraries for serialization enable persistent storage and loading of user data.

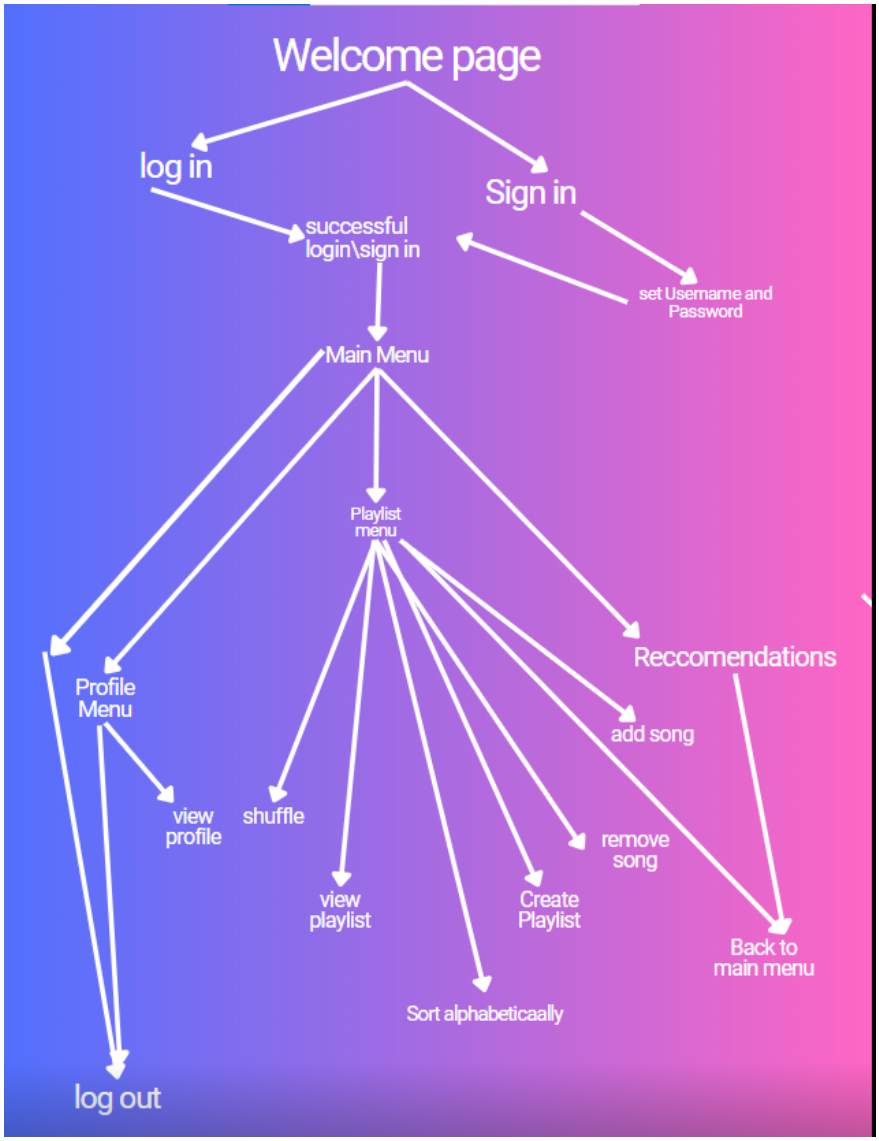
Boost C++ Libraries

Boost libraries play a crucial role in the project, particularly in the serialization of user data. This feature enhances data persistence, allowing users to seamlessly access their playlists and preferences across sessions.

Crypto C++ Libraries (Hashing)

Crypto libraries are employed for generating unique keys, enhancing password security through hashing. This ensures that user passwords are stored securely, adding an extra layer of protection.

Figure 1: Project Diagram



Justification of Data Structures used

Linked List (Playlist)

Linked lists are chosen for playlist management due to their dynamic nature, facilitating efficient insertion, removal, and traversal of songs within playlists.

Stack (Listening History)

A stack is implemented to manage the listening history, adhering to the Last-In-First-Out (LIFO) principle.

Hash Table (User Authentication)

The use of a hash table ensures the quick and secure retrieval of user credentials, contributing to robust user authentication.

Unordered Maps

Unordered maps are employed to enhance certain functionalities, providing a dynamic and efficient data structure.

AVL Trees (Search Songs and Recommendation Scores)

AVL trees are utilized for searching songs, contributing to reduced search times and optimized recommendation scores.

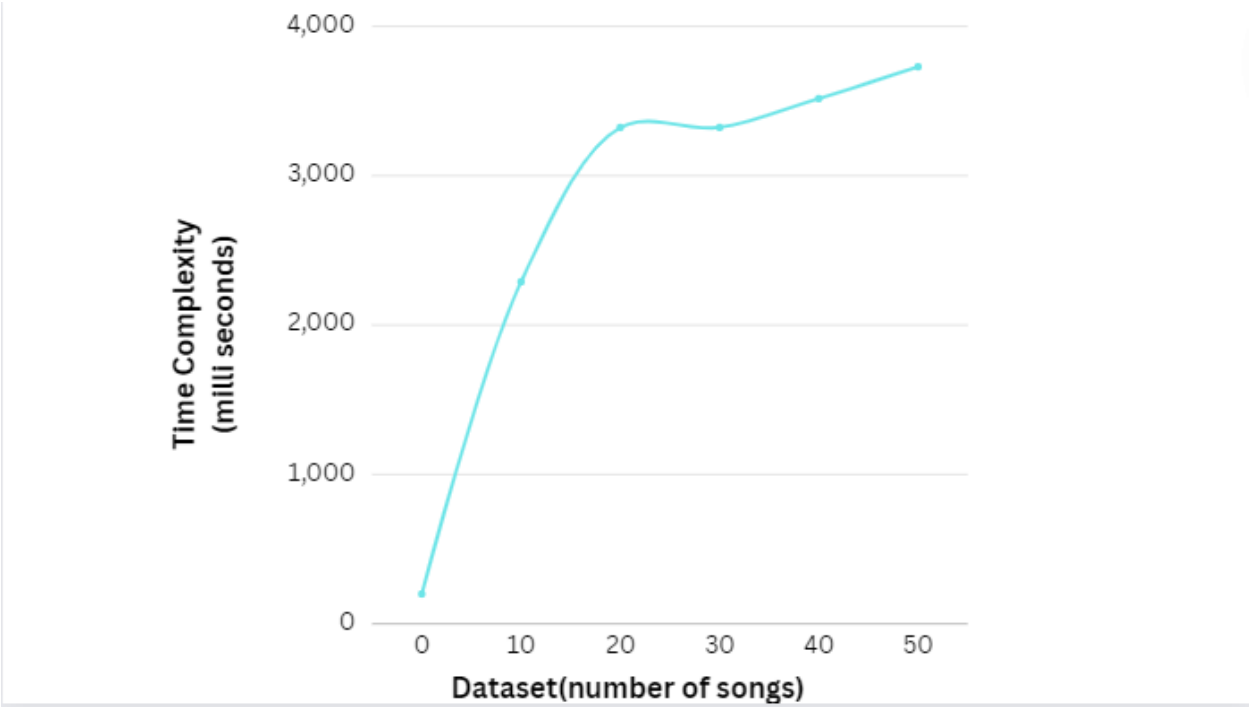
Results

The project includes functionalities for displaying playlists, adding and removing songs, shuffling playlists, and sorting them alphabetically. User authentication ensures secure login, and the application provides a basic

structure for exploring recommendations.

Chart:

Figure 2: representing a time complexity graph



Conclusion

The MuSicHub project successfully implements core features of a music application, allowing users to manage playlists and access song recommendations. The chosen data structures provide efficiency in playlist management and secure user authentication. The use of Boost C++ libraries and Crypto enhances data persistence. Future

improvements could include more advanced recommendation algorithms, a graphical user interface, and additional features for an enhanced user experience.

Appendix : Code

Menu Driven :

Header File:

```
#pragma once
#include <iostream>
#include <windows.h>
#include <unordered_map>
#include <string>
#include <stack>
#include <queue>
#include <fstream>
#include <list>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/access.hpp>
#include "boost/serialization/deque.hpp"
#include "boost/serialization/stack.hpp"
#include <boost/serialization/queue.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/optional.hpp>
#include <unordered_map>
#include<boost/serialization/unordered_map.hpp>
#include <boost/serialization/serialization.hpp>
#include <vector>

using namespace std;


class Song {
private:
```

```
    string id;
    string name;
    float duration;
    float energy;
    float key;
    float loudness;
    float speechiness;
    float acousticness;
    float instrumentalness;
    float liveness;
    float tempo;
    int streams;
    float score;

public:
    Song();
    Song(string id, const string& n, float dur, float en, float k, float
loud, float spch, float acous, float instr, float liv, float temp) :
id(id), name(n), duration(dur), energy(en), key(k), loudness(loud),
speechiness(spch), acousticness(acous), instrumentalness(instr),
liveness(liv), tempo(temp),score(CalcScore()){}
    void setID(string ID);
    void setScore(float score);
    void setStreams(int stream);
    void setName(string n);;
    void setDuration(float dur);
    void setEnergy(float en);
    void setKey(float k);
    void setLoudness(float l);
    void setSpeechiness(float sp);
    void setAcousticness(float ac);
    void setInstrumentalness(float inst);
    void setLiveness(float liv);
    void setTempo(float temp);

    string getName();
    float getDuration();
    float getEnergy();
    float getKey();
    float getLoudness();
    float getSpeechiness();
    float getAcousticness();
    float getInstrumentalness();
    float getLiveness();
    float getTempo();
    float getScore();
    struct HashFunction {
```

```
        size_t operator()(const Song& song) const {
            return hash<string>()(song.name);
        }
    };
    struct EqualityComparison {
        bool operator()(const Song& lhs, const Song& rhs) const {
            if (lhs.name == rhs.name) {
                return true;
           }
            return false;
        }
    };
    float CalcScore()const;
    void displayInfo();

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& name;
        ar& duration;
        ar& energy;
        ar& key;
        ar& loudness;
        ar& speechiness;
        ar& acousticness;
        ar& instrumentalness;
        ar& liveness;
        ar& tempo;
    }
};

class SongNode {
public:
    Song* song;
    SongNode* next;

    SongNode() : song(nullptr), next(nullptr) {}
    SongNode(Song* songPtr) : song(songPtr), next(nullptr) {}

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& song;
        ar& next;
    }
};
```

```
class Playlist
{
private:
    SongNode* head;
    SongNode* getNodeAtPosition(int pos);
    int getCount();
public:
    string name;
    Playlist();
    Playlist(const string& playlistName) : name(playlistName),
head(nullptr) {}
    void addSong(Song* song);
    void removeSong(const string& songName);
    void displayPlaylist();
    void shufflePlaylist();
    Playlist& operator=(const Playlist& other);
    float calculatePlaylist();
    ~Playlist();
    SongNode* merge(SongNode* left, SongNode* right);
    SongNode* mergeSort(SongNode* head);
    void sortPlaylist();

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& name;
        ar& head;
    }
};

void PrintStack(stack<Playlist*> p1) {
    cout << "Playlist Names in the Stack:" << endl;
    while (!p1.empty()) {
        cout << p1.top()->name << endl;
        p1.pop();
    }
}

class User {
public:
    string username;
    string password;
    stack<Playlist*> SavedPlaylists;
    stack<Song*> ListeningHistory;
    stack<Song*> LikedSongs;
    unordered_map<Song, int,Song::HashFunction,Song::EqualityComparison>
MostListenedToSongs;
```

```
    User(string usern, string passw);
    User();
    void displayProfile();
    float CalculateRecScore();
    boost::optional<Playlist*> SearchPlaylist(const std::string& n);

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& username;
        ar& password;
        ar& SavedPlaylists;
        ar& ListeningHistory;
        ar& LikedSongs;
        ar& MostListenedToSongs;
    }
};

class UserHashTable {
    vector<list<User>> HashTable;
    static const size_t TABLE_SIZE = 100;
    size_t HashFunction(string key);

public:
    void LoadFromFile();
    UserHashTable() : HashTable(TABLE_SIZE) {};
    void SaveToFile();
    void InsertUser(User& user);
    boost::optional<User> ValidateLogin(const string& username, const
string& password);

private:
    string hashpassword(const string& password);
};
class UserMenu {
public:
    void SignUpPrompt();
    void LoginPrompt();
    void Welcome();
    void MainMenu(User&u1,int value=0);
    void ProfileMenu(User&u1);
    void PlaylistMenu(User& u1);
    void PlaylistChoice(int choice,User&u1);
    void Logo();
};
```

```
#include "Header.h"
#include "emoji.h"
#include <windows.h>
#include<cwchar>
#include <iostream>
#include <conio.h>
#include <fstream>
#include <stack>
#include "cryptlib.h"
#include "sha.h"
#include "hex.h"
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/access.hpp>
#include "boost/serialization/deque.hpp"
#include "boost/serialization/stack.hpp"
#include <boost/serialization/queue.hpp>
#include <boost/optional.hpp>
#include<boost/unordered_map.hpp>
#include <unordered_map>
#include <algorithm>
#include<thread>
#include<chrono>
#include<cfloat>
using namespace std;

UserMenu m1;

float Song::getScore() {
    return score;
}
string Song::getName()
{
    return name;
}
float Song::getDuration()
{
    return duration;
}
float Song::getEnergy()
{
    return energy;
}
float Song::getKey()
{
    return key;
```

```
}
float Song::getLoudness()
{
    return loudness;
}
float Song::getSpeechiness()
{
    return speechiness;
}
float Song::getAcousticness()
{
    return acousticness;
}
float Song::getInstrumentalness()
{
    return instrumentalness;
}
float Song::getLiveness()
{
    return liveness;
}
float Song::getTempo()
{
    return tempo;
}
void Song::setName(string n)
{
    name = n;
}
void Song::setDuration(float dur)
{
    duration = dur;
}
void Song::setEnergy(float en)
{
    energy = en;
}
void Song::setKey(float k)
{
    key = k;
}
void Song::setLoudness(float loud)
{
    loudness = loud;
}
void Song::setSpeechiness(float spch)
{
```

```
    speechiness = spch;
}
void Song::setAcousticness(float ac)
{
    acousticness = ac;
}
void Song::setInstrumentalness(float instr)
{
    instrumentalness = instr;
}
void Song::setLiveness(float live)
{
    liveness = live;
}
void Song::setTempo(float t) {
    tempo = t;
}
void Song::setID(string ID) {
    id = ID;
}

void Song::displayInfo()
{
    cout << "Name: " << name << endl;
    cout << "Duration: " << duration << endl;
    cout << "Energy: " << energy << endl;
    cout << "Key: " << key << endl;
    cout << "Loudness: " << loudness << endl;
    cout << "Speechiness: " << speechiness << endl;
    cout << "Acousticness: " << acousticness << endl;
    cout << "Instrumentalness: " << instrumentalness << endl;
    cout << "Liveness: " << liveness << endl;
    cout << "Tempo: " << tempo << endl;
}

float Song::CalcScore() const {
    float score = (energy + key + loudness + speechiness + acousticness +
instrumentalness + liveness + tempo) / 8;
    return score;
}
Song::Song() {

}

SongNode* Playlist::getNodeAtPosition(int pos)
{
    SongNode* current = head;
```

```
    for (int i = 0; i < pos && current != NULL; ++i) {
        current = current->next;
    }
    return current;
}

int Playlist::getCount() {
    int count = 0;
    SongNode* current = head;
    while (current != NULL) {
        ++count;
        current = current->next;
    }
    return count;
}
Playlist::Playlist() {
    head = NULL;
    name = " ";
}
void Playlist::addSong(Song* song)
{
    SongNode* newSong = new SongNode(song);
    if (head == NULL)
    {
        head = newSong;
    }
    else
    {
        SongNode* temp = head;
        while (temp->next != NULL)
        {
            temp = temp->next;
        }
        temp->next = newSong;
    }
    cout << song->getName() << " has been added to " << name << "." <<
endl;
}

void Playlist::removeSong(const string& songName) {
    SongNode* current = head;
    SongNode* prev = NULL;

    while (current != NULL && current->song->getName() != songName) {
        prev = current;
        current = current->next;
    }
```

```
    if (current != NULL) {
        if (prev != NULL) {
            prev->next = current->next;
        }
        else {
            head = current->next;
        }
        delete current;
        cout << songName << " has been removed from " << name << "." <<
endl;
    }
    else {
        cout << songName << " is not in " << name << "." << endl;
    }
}
void Playlist::displayPlaylist()
{
    cout << "----------------------------------------------------" << endl;
    cout << "Playlist: " << name << endl;
    cout << "----------------------------------------------------" << endl;
    SongNode* current = head;
    string temp;
    while (current != NULL)
    {
        temp = current->song->getName();
        cout << temp << endl;
        current = current->next;
    }
    cout << endl;
}
void Playlist::shufflePlaylist() {
    srand(time(0));
    int count = getCount();
    for (int i = count - 1; i > 0; --i) {
        int j = rand() % (i + 1);
        SongNode* songI = getNodeAtPosition(i);
        SongNode* songJ = getNodeAtPosition(j);
        swap(songI->song, songJ->song);
    }
    cout << name << " has been shuffled." << endl;
}
Playlist& Playlist::operator=(const Playlist& other) {
    if (this != &other) {
        while (head != NULL) {
            SongNode* temp = head;
            head = head->next;
```

```
            delete temp;
        }
        this->name = other.name;
        SongNode* otherCurrent = other.head;
        SongNode* current = NULL;
        while (otherCurrent != NULL) {
            Song* newSong = new Song(*otherCurrent->song);
            SongNode* newSongNode = new SongNode(newSong);

            if (current == NULL) {
                head = newSongNode;
            }
            else {
                current->next = newSongNode;
            }

            current = newSongNode;
            otherCurrent = otherCurrent->next;
        }
    }
    return *this;
}
Playlist::~Playlist() {
    while (head != NULL) {
        SongNode* temp = head;
        head = head->next;
        delete temp->song;
        delete temp;
    }
}
SongNode* Playlist::merge(SongNode* left, SongNode* right) {
    if (left == NULL) {
        return right;
    }
    if (right == NULL) {
        return left;
    }

    if (left->song->getName() < right->song->getName()) {
        left->next = merge(left->next, right);
        return left;
    }
    else {
        right->next = merge(left, right->next);
        return right;
    }
}
```

```
float Playlist::calculatePlaylist() {
    float playlistScore = 0.0;
    SongNode* current = head;
    int i = 1;
    while (current != nullptr) {
        playlistScore += current->song->CalcScore();
        current = current->next;
        i++;
    }
    return playlistScore / i;
}

SongNode* Playlist::mergeSort(SongNode* head) {
    if (head == NULL || head->next == NULL)
    {
        return head;
    }
    SongNode* slow = head;
    SongNode* fast = head->next;
    while (fast != NULL && fast->next != NULL)
    {
        slow = slow->next;
        fast = fast->next->next;
    }
    SongNode* left = head;
    SongNode* right = slow->next;
    slow->next = NULL;
    left = mergeSort(left);
    right = mergeSort(right);
    return merge(left, right);
}
void Playlist::sortPlaylist()
{
    head = mergeSort(head);
    cout << name << " has been sorted alphabetically." << endl;
}
class AVLNode
{
public:
    Song song;
    AVLNode* left;
    AVLNode* right;
    int height;

    AVLNode(Song s) : song(s), left(NULL), right(NULL), height(1) {}
};
```

```
class AVLTree
{
private:
    AVLNode* root;

    int height(AVLNode* node)
    {
        return (node == NULL) ? 0 : node->height;
    }

    int balanceFactor(AVLNode* node)
    {
        return (node == NULL) ? 0 : height(node->left) -
height(node->right);
    }

    void updateHeight(AVLNode* node)
    {
        node->height = 1 + max(height(node->left), height(node->right));
    }

    AVLNode* rotateRight(AVLNode* y)
    {
        AVLNode* x = y->left;
        AVLNode* T2 = x->right;

        x->right = y;
        y->left = T2;

        updateHeight(y);
        updateHeight(x);

        return x;
    }

    AVLNode* rotateLeft(AVLNode* x)
    {
        AVLNode* y = x->right;
        AVLNode* T2 = y->left;

        y->left = x;
        x->right = T2;

        updateHeight(x);
        updateHeight(y);

        return y;
```

```
}
AVLNode* insert(AVLNode* node, Song song)
{
    if (node == NULL)
    {
        return new AVLNode(song);
    }

    if (song.getName() < node->song.getName())
    {
        node->left = insert(node->left, song);
    }
    else if (song.getName() > node->song.getName())
    {
        node->right = insert(node->right, song);
    }
    else
    {
        return node;
    }

    updateHeight(node);

    int balance = balanceFactor(node);
    if (balance > 1)
    {
        if (song.getName() < node->left->song.getName())
        {
            return rotateRight(node);
        }
        else
        {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
    }
    if (balance < -1) {
        if (song.getName() > node->right->song.getName())
        {
            return rotateLeft(node);
        }
        else
        {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }
    }
```

```
        return node;
    }
    Song* search(AVLNode* node, const string& lowerUserInput) {
        if (node == NULL) {
            return NULL;
        }
        string lowerSongName = toLower(node->song.getName());
        if (lowerSongName.find(lowerUserInput) != string::npos) {
            return &(node->song);
        }

        if (lowerUserInput < lowerSongName) {
            return search(node->left, lowerUserInput);
        }
        else {
            return search(node->right, lowerUserInput);
        }
    }
    string toLower(const string& str) {
        string result;
        for (size_t i = 0; i < str.length(); ++i) {
            result += tolower(str[i]);
        }
        return result;
    }


    void inOrderTraversal(AVLNode* node)
    {
        if (node == NULL)
        {
            return;
        }
        string n;
        inOrderTraversal(node->left);
        n = node->song.getName();
        cout << n << endl;
        inOrderTraversal(node->right);
    }

public:
    AVLTree() : root(NULL) {}

    void insert(Song song) {
        root = insert(root, song);
    }
```

```
    void displayInOrder()
    {
        cout << "In-Order Traversal:" << endl;
        inOrderTraversal(root);
    }
    Song* search(const string& songName) {
        string lowerUserInput = toLower(songName);
        return search(root, lowerUserInput);
    }
};

void readfileAndInsert(AVLTree& avlTree)
{
    string filename = "FinalDataSet.txt";
    ifstream file(filename.c_str());

    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    string line;
    while (getline(file, line))
    {
        stringstream ss(line);
        string token;

        Song song;
        getline(ss, token, ',');
        song.setID(token);
        getline(ss, token, ',');
        song.setName(token);
        getline(ss, token, ',');
        song.setDuration(atof(token.c_str()));
        getline(ss, token, ',');
        song.setEnergy(atof(token.c_str()));
        getline(ss, token, ',');
        song.setKey(atof(token.c_str()));
        getline(ss, token, ',');
        song.setLoudness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setSpeechiness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setAcousticness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setInstrumentalness(atof(token.c_str()));
        getline(ss, token, ',');
```

```
        song.setLiveness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setTempo(atof(token.c_str()));
        getline(ss, token, ',');
        song.setScore(atof(token.c_str()));
        avlTree.insert(song);
    }

    file.close();
}

class AVLNode2 {
public:
    Song song;
    AVLNode2* left;
    AVLNode2* right;
    int height;

    AVLNode2(Song s) : song(s), left(NULL), right(NULL), height(1) {}
};

// AVLTree2 class
class AVLTree2 {
private:
    AVLNode2* root;

    int height(AVLNode2* node) {
        return (node == NULL) ? 0 : node->height;
    }

    int balanceFactor(AVLNode2* node) {
        return (node == NULL) ? 0 : height(node->left) -
height(node->right);
    }

    void updateHeight(AVLNode2* node) {
        node->height = 1 + max(height(node->left), height(node->right));
    }

    AVLNode2* rotateRight(AVLNode2* y) {
        AVLNode2* x = y->left;
        AVLNode2* T2 = x->right;

        x->right = y;
        y->left = T2;

        updateHeight(y);
```

```
    updateHeight(x);

    return x;
}

AVLNode2* rotateLeft(AVLNode2* x) {
    AVLNode2* y = x->right;
    AVLNode2* T2 = y->left;

    y->left = x;
    x->right = T2;

    updateHeight(x);
    updateHeight(y);

    return y;
}

AVLNode2* insertScore(AVLNode2* node, Song song) {
    if (node == NULL) {
        return new AVLNode2(song);
    }

    if (song.getScore() < node->song.getScore()) {
        node->left = insertScore(node->left, song);
    }
    else if (song.getScore() > node->song.getScore()) {
        node->right = insertScore(node->right, song);
    }
    else {
        return node; // Ignoring duplicates
    }

    // Update height
    updateHeight(node);

    // Rebalance the tree
    int balance = balanceFactor(node);
    if (balance > 1) {
        if (song.getScore() < node->left->song.getScore()) {
            return rotateRight(node);
        }
        else {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
    }
```

```
        if (balance < -1) {
            if (song.getScore() > node->right->song.getScore()) {
                return rotateLeft(node);
            }
            else {
                node->right = rotateRight(node->right);
                return rotateLeft(node);
            }
        }

        return node;
    }

    Song* search(AVLNode2* node, const string& lowerUserInput) {
        if (node == NULL) {
            return NULL;
        }

        string lowerScore = toLower(to_string(node->song.getScore()));
        if (lowerScore.find(lowerUserInput) != string::npos) {
            return &(node->song);
        }

        if (lowerUserInput < lowerScore) {
            return search(node->left, lowerUserInput);
        }
        else {
            return search(node->right, lowerUserInput);
        }
    }

    string toLower(const string& str) {
        string result;
        for (size_t i = 0; i < str.length(); ++i) {
            result += tolower(str[i]);
        }
        return result;
    }

    void inOrderTraversal(AVLNode2* node) {
        if (node == NULL) {
            return;
        }
        inOrderTraversal(node->left);
        cout << "Score: " << node->song.getScore() << " - Name: " <<
node->song.getName() << endl;
        inOrderTraversal(node->right);
```

```
    }
    Song* closestSong;
    float targetScore;
    float minDifference;

    // Helper function for finding the closest score
    void findClosestScoreHelper(AVLNode2* node) {
        if (node == NULL) {
            return;
        }
        float currentDifference = abs(targetScore -
node->song.getScore());

        if (currentDifference < minDifference) {
            minDifference = currentDifference;
            closestSong = &(node->song);
        }
        if (targetScore < node->song.getScore()) {
            findClosestScoreHelper(node->left);
        }
        else {
            findClosestScoreHelper(node->right);
        }
    }
    Song* findClosestScore(AVLNode2* node, float targetScore, float&
minDifference, Song* closestSong) {
        if (node == NULL) {
            return closestSong;
        }

        float currentDifference = abs(node->song.getScore() -
targetScore);
        if (currentDifference < minDifference) {
            minDifference = currentDifference;
            closestSong = &(node->song);
        }

        if (targetScore < node->song.getScore()) {
            return findClosestScore(node->left, targetScore,
minDifference, closestSong);
        }
        else {
            return findClosestScore(node->right, targetScore,
minDifference, closestSong);
        }
    }
```

```
public:
    AVLTree2() : root(NULL) {}

    void insertScore(Song song) {
        root = insertScore(root, song);
    }

    Song* search(AVLNode2* node, float targetScore) {
        if (node == NULL) {
            return NULL;
        }

        float nodeScore = node->song.getScore();
        if (targetScore == nodeScore) {
            return &(node->song);
        }

        if (targetScore < nodeScore) {
            return search(node->left, targetScore);
        }
        else {
            return search(node->right, targetScore);
        }
    }

    void displayInOrder() {
        cout << "In-Order Traversal:" << endl;
        inOrderTraversal(root);
    }
    Song* findClosestScore(float target) {
        closestSong = NULL;
        targetScore = target;
        minDifference = FLT_MAX;

        findClosestScoreHelper(root);

        return closestSong;
    }
    Song* searchClosestScore(float targetScore) {
        float minDifference = FLT_MAX;
        Song* closestSong = NULL;
        return findClosestScore(root, targetScore, minDifference,
closestSong);
    }
};
void readfileAndInsert2(AVLTree2& AVLTree2) {
    string filename = "FinalDataSet.txt";
```

```
    ifstream file(filename.c_str());

    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << endl;
        return;
    }

    string line;
    while (getline(file, line)) {
        stringstream ss(line);
        string token;

        Song song;
        getline(ss, token, ',');
        song.setID(token);
        getline(ss, token, ',');
        song.setName(token);
        getline(ss, token, ',');
        song.setDuration(atof(token.c_str()));
        getline(ss, token, ',');
        song.setEnergy(atof(token.c_str()));
        getline(ss, token, ',');
        song.setKey(atof(token.c_str()));
        getline(ss, token, ',');
        song.setLoudness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setSpeechiness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setAcousticness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setInstrumentalness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setLiveness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setTempo(atof(token.c_str()));
        getline(ss, token, ',');
        song.setStreams(atof(token.c_str()));
        song.setScore(song.CalcScore());
        AVLTree2.insertScore(song);
    }

    file.close();
}
void UserHashTable::SaveToFile() {
    ofstream ofs("user_data.txt");
    boost::archive::text_oarchive oarchive(ofs);
    for (const auto& userVec : HashTable) {
```

```
        for (const auto& user : userVec) {
            oarchive << user;
        }
    }
    cout << "User data saved successfully." << endl;
}
size_t UserHashTable::HashFunction(string key) {
    return hash < string>{}(key) % TABLE_SIZE;
}
string UserHashTable::hashpassword(const string& password) {
    using namespace CryptoPP;
    SHA1 hash;
    byte digest[SHA1::DIGESTSIZE];
    hash.CalculateDigest(digest, (const byte*)password.c_str(),
password.length());
    HexEncoder encoder;
    std::string hashedPassword;
    encoder.Attach(new StringSink(hashedPassword));
    encoder.Put(digest, sizeof(digest));
    encoder.MessageEnd();
    return hashedPassword;
}

void UserHashTable::InsertUser(User& user) {
    size_t index = HashFunction(user.username);
    user.password = hashpassword(user.password);
    HashTable[index].emplace_back(user);
}

void UserHashTable::LoadFromFile() {
    {
        ifstream ifs("user_data.txt");
        boost::archive::text_iarchive iarchive(ifs);

        try {
            while (true) {
                User loadedUser;
                iarchive >> loadedUser;
                InsertUser(loadedUser);
            }
        }
        catch (const boost::archive::archive_exception& e) {
            if (e.code ==
boost::archive::archive_exception::input_stream_error) {
                cout << "End of file reached." << endl;
            }
            else {
```

```
                cerr << "Error during deserialization: " << e.what() <<
endl;
            }
        }
    }

}
boost::optional<User> UserHashTable::ValidateLogin(const string& username,
const string& password) {
    size_t index = HashFunction(username);
    for (const auto& user : HashTable[index]) {
        if (user.username == username) {
            string hashEnteredPassword = hashpassword(password);
            cout << "Stored Hash: " << user.password << endl;
            cout << "Entered Hash: " << hashEnteredPassword << endl;
            if (user.password == hashEnteredPassword) {
                return boost::make_optional(user);
            }
        }
    }
    return boost::none;
}
void UserMenu::PlaylistMenu(User& u1) {
    Logo();
    int choice;
    cout << "1. Display Playlist" << endl
        << "2. Add Song to Playlist" << endl
        << "3. Remove Song from Playlist" << endl
        << "4. Shuffle Playlist" << endl;
    cin >> choice;
    PlaylistChoice(choice, u1);
}
void Continue(User&u1) {
    int choice;
    cout << "Enter 1 to Return to Main Menu" << endl;
    cin >> choice;
    switch (choice) {
    case 1:
    {
        m1.MainMenu(u1);
        break;
    }
    default: {
        Continue(u1);
    }
    }
}
```

```
void UserMenu::PlaylistChoice(int choice, User& u1) {
    string playlistname;
    cout << "Enter playlist to perform the action on: ";
    cin >> playlistname;

    boost::optional<Playlist*> playlistResult =
u1.SearchPlaylist(playlistname);

    if (playlistResult) {
        Playlist* playlist = *playlistResult;

        switch (choice) {
        case 1:
            playlist->displayPlaylist();
            Continue(u1);
            break;

        case 2: {
            AVLTree avlTree;

            string filename = "FinalDataSet.txt";

            ifstream file(filename.c_str());

            if (!file.is_open()) {
                cerr << "Error opening file: " << filename << endl;
                break;
            }

            readfileAndInsert(avlTree);

            string find;
            cout << "Enter the song you wish to search:" << endl;
            cin >> find;
            Song* foundSong = avlTree.search(find);
            string str1;
            float dur;
            if (foundSong != NULL)
            {
                cout << "Song found:" << endl;
                str1 = foundSong->getName();
                dur = foundSong->getDuration();
                cout << str1 << endl;
                cout << "Duration:" << dur << endl;
                playlist->addSong(foundSong);
            }
            else
```

```
            {
                cout << "Song not found." << endl;
            }
            Continue(u1);
            break;
        }

        case 3: {
            string songName;
            cout << "Enter the name of the song to remove: ";
            cin >> songName;
            playlist->removeSong(songName);
            Continue(u1);
            break;
        }

        case 4:
            playlist->shufflePlaylist();
            Continue(u1);
            break;

        default:
            cout << "Invalid choice. Please enter a valid option." <<
endl;
        }
    }
    else {
        cout << "Playlist not found." << endl;
    }
}

void UserMenu::ProfileMenu(User& u1) {
    Logo();
    cout << "1. View Profile" << endl
        << "2. Logout" << endl;

    int choice;
    cout << "Enter your choice: ";
    cin >> choice;

    switch (choice) {
    case 1:
        u1.displayProfile();
        Continue(u1);
        break;

    case 2:
```

```
        cout << "Logging out..." << endl;
        break;

    default:
        cout << "Invalid choice. Please enter a valid option." << endl;
        ProfileMenu(u1);
    }
}

void UserMenu::MainMenu(User&u1,int value) {
    system("cls");
    int choice;
    cout << "Main Menu:" << endl;
    cout << "1. Profile Menu" << endl << "2.Playlist Menu" << endl <<
"3.Recommendations" << endl;
    cin >> choice;
    switch (choice) {
    case 1: {
        ProfileMenu(u1);
        break;
    }
    case 2:
    {
        PlaylistMenu(u1);
        break;
    }
    case 3:
    {
        AVLTree2 avlTree2;

        string filename2 = "FinalDataSet.txt";

        ifstream file2(filename2.c_str());

        if (!file2.is_open()) {
            cerr << "Error opening file: " << filename2 << endl;
            break;
        }

        readfileAndInsert2(avlTree2);
        float userScore = 11.1;
        Song* closestSong = avlTree2.searchClosestScore(userScore);

        if (closestSong != NULL) {
            cout << "Closest matching song found:" << endl;
            cout << "Score: " << closestSong->getScore() << " - Name: " <<
closestSong->getName() << endl;
```

```
        }
        else {
            cout << "No matching song found." << endl;
        }
        Continue(u1);
        break;
    }
    }
}
void UserMenu::Welcome() {

    Logo();
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD cursorpos;
    cursorpos.X = 45;
    cursorpos.Y = 6;
    SetConsoleCursorPosition(consoleHandle, cursorpos);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN |
FOREGROUND_RED | FOREGROUND_BLUE);
    cout << "Press ";
    SetConsoleTextAttribute(consoleHandle,FOREGROUND_BLUE);
    cout << "TAB ";
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN |
FOREGROUND_RED | FOREGROUND_BLUE);
    cout<<"To SignUp!";
    cursorpos.X = 45;
    cursorpos.Y = 7;

    SetConsoleCursorPosition(consoleHandle, cursorpos);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN |
FOREGROUND_RED | FOREGROUND_BLUE);
    cout << "Press ";
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE);
    cout << "Enter";
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN |
FOREGROUND_RED | FOREGROUND_BLUE);
    cout << " To Login!";

    while (1) {
        if (GetAsyncKeyState(VK_RETURN)) {
            LoginPrompt();
            break;
        }
        if (GetAsyncKeyState(VK_TAB)) {
            SignUpPrompt();
            break;
        }
```

```
    }
}

void UserMenu::Logo() {
    SetConsoleOutputCP(CP_UTF8);
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD consoleMode;
    GetConsoleMode(consoleHandle, &consoleMode);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE |
FOREGROUND_INTENSITY);
    cout << "\t\t\t\t\t\tMuSicHub" << endl;
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_INTENSITY);
    cout << "\t\t\t\tRecommending Songs Through User Behaviour" << endl;
    COORD cursorpos;
    cursorpos.X = 85;
    cursorpos.Y = 2;
    SetConsoleCursorPosition(consoleHandle, cursorpos);

    float dist;
    for (int i = 0; i <= 2 * 3; i++) {
        for (int j = 0; j <= 2 * 3; j++) {
            dist = sqrt((i - 3) * (i - 3) +
                (j - 3) * (j - 3));
            if (dist > 3 - 0.5 && dist < 3 + 0.5)
                cout << emojicpp::emojize(":musical_note:");
            else
                cout << " ";
        }

        cursorpos.Y++;
        SetConsoleCursorPosition(consoleHandle, cursorpos);
    }
}

void UserMenu::LoginPrompt() {
    system("cls");
    Logo();
    string username; char password[100];
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD consoleMode;
    COORD coord;
    coord.X = 50;
    coord.Y = 2;
    SetConsoleCursorPosition(consoleHandle, coord);
    GetConsoleMode(consoleHandle, &consoleMode);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_INTENSITY);
    cout << "Enter username : " << endl;
```

```
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE |
FOREGROUND_GREEN | FOREGROUND_RED);
    SetConsoleCursorPosition(consoleHandle, coord);
    GetConsoleMode(consoleHandle, &consoleMode);
    coord.X = 68;
    coord.Y = 2;
    SetConsoleCursorPosition(consoleHandle, coord);
    cin >> username;
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_INTENSITY);
    coord.X = 50;
    coord.Y = 3;
    SetConsoleCursorPosition(consoleHandle, coord);
    cout << "Enter password : " << endl;
    coord.X = 68;
    coord.Y = 3;
    SetConsoleCursorPosition(consoleHandle, coord);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE |
FOREGROUND_GREEN | FOREGROUND_RED);
    int k=1;
    while (password[k - 1] != '\r')
    {
        password[k] = _getch();
        if (password[k - 1] != '\r')
        {
            cout << "*";
        }
        k++;
    }
    UserHashTable uhs;
    uhs.LoadFromFile();
    string pass_a = password;
    boost::optional<User> validatedUser = uhs.ValidateLogin(username,
password);
    if (uhs.ValidateLogin(username, pass_a)) {
        cout << "Login Successful!" << endl;
        User u1 = *validatedUser;
        MainMenu(u1);
    }
    else {
        cout << "Login Unsuccessful!" << endl;
        Welcome();
    }
}

float User::CalculateRecScore() {
   float totalScore = 0.0;
```

```
    stack<Playlist*> tempStack = SavedPlaylists;
    int i = 1;
    while (!tempStack.empty()) {
        Playlist* currentPlaylist = tempStack.top();
        tempStack.pop();
        i++;
        totalScore += currentPlaylist->calculatePlaylist();
    }
    return totalScore / i;
}
boost::optional<Playlist*> User::SearchPlaylist(const std::string& n) {
    Playlist* temp;
    std::stack<Playlist*> tempStack = SavedPlaylists;
    while (!tempStack.empty()) {
        temp = tempStack.top();
        if (temp->name == n) {
            return boost::optional<Playlist*>(temp);
        }
        tempStack.pop();
    }
    return boost::none;
}
User::User() {

}
void User::displayProfile() {
    HANDLE conhandle = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD cord;
    cord.X = 2;
    cord.Y = 3;
    SetConsoleCursorPosition(conhandle, cord);
    cout << "Your Playlists : : ";
    PrintStack(SavedPlaylists);
}
User::User(string username, string password) {
    this->username = username;
    this->password = password;
}
void UserMenu::SignUpPrompt() {
    bool signUpSuccess = false;

        system("cls");
        Logo();
        string username;
        char password[100];
        HANDLE conhandle = GetStdHandle(STD_OUTPUT_HANDLE);
        COORD cord;
```

```
        cord.X = 40;
        cord.Y = 3;
        SetConsoleCursorPosition(conhandle, cord);
        cout << "Enter Username : ";
        cord.Y++;
        SetConsoleCursorPosition(conhandle, cord);
        cin >> username;
        cord.Y++;
        SetConsoleCursorPosition(conhandle, cord);
        cout << "Enter Password : ";
        cord.Y++;
        SetConsoleCursorPosition(conhandle, cord);
        int k = 1;
        while (password[k - 1] != '\r') {
            password[k] = _getch();
            if (password[k - 1] != '\r') {
                cout << "*";
            }
            k++;
        }

        {

            std::ofstream ofs("user_data.txt");
            boost::archive::text_oarchive oarchive(ofs);

            User user1(username, password);

            oarchive << user1;
            cout << "Signed Up Successfully" << endl;
            system("pause");
            MainMenu(user1);

        }
}

void Song::setStreams(int str) {
    streams = str;
}
void Song::setScore(float scor) {
    score = scor;
}

Song* search(AVLNode2* node, float targetScore) {
    if (node == NULL) {
        return NULL;
    }
```

```
    float nodeScore = node->song.getScore();
    if (targetScore == nodeScore) {
        return &(node->song);
    }

    if (targetScore < nodeScore) {
        return search(node->left, targetScore);
    }
    else {
        return search(node->right, targetScore);
    }
}

int main() {


   m1.Welcome();
   return 0;
}
```

Automated :

Header Files :

```
#pragma once
#include <iostream>
#include <windows.h>
#include <unordered_map>
#include <string>
#include <stack>
#include <queue>
#include <fstream>
#include <list>
#include <optional>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/access.hpp>
#include "boost/serialization/deque.hpp"
```

```
#include "boost/serialization/stack.hpp"
#include <boost/serialization/queue.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/optional.hpp>
#include <unordered_map>
#include<boost/serialization/unordered_map.hpp>
#include <boost/serialization/serialization.hpp>
#include <vector>

using namespace std;


class Song {
private:
    string id;
    string name;
    float duration;
    float energy;
    float key;
    float loudness;
    float speechiness;
    float acousticness;
    float instrumentalness;
    float liveness;
    float tempo;
    int streams;
    float score;

public:
    Song();
    Song(string id, const string& n, float dur, float en, float k, float loud,
float spch, float acous, float instr, float liv, float temp) : id(id),
name(n), duration(dur), energy(en), key(k), loudness(loud), speechiness(spch),
acousticness(acous), instrumentalness(instr), liveness(liv),
tempo(temp),score(CalcScore()){}
    void setID(string ID);
    void setScore(float score);
    void setStreams(int stream);
    void setName(string n);;
    void setDuration(float dur);
    void setEnergy(float en);
    void setKey(float k);
    void setLoudness(float l);
    void setSpeechiness(float sp);
    void setAcousticness(float ac);
    void setInstrumentalness(float inst);
    void setLiveness(float liv);
```

```
    void setTempo(float temp);

    string getName();
    float getDuration();
    float getEnergy();
    float getKey();
    float getLoudness();
    float getSpeechiness();
    float getAcousticness();
    float getInstrumentalness();
    float getLiveness();
    float getTempo();
    float getScore();
    struct HashFunction {
        size_t operator()(const Song& song) const {
            return hash<string>()(song.name);
        }
    };
    struct EqualityComparison {
        bool operator()(const Song& lhs, const Song& rhs) const {
            if (lhs.name == rhs.name) {
                return true;
           }
            return false;
        }
    };
    float CalcScore()const;
    void displayInfo();

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& name;
        ar& duration;
        ar& energy;
        ar& key;
        ar& loudness;
        ar& speechiness;
        ar& acousticness;
        ar& instrumentalness;
        ar& liveness;
        ar& tempo;
    }
};

class SongNode {
public:
```

```
    Song* song;
    SongNode* next;

    SongNode() : song(nullptr), next(nullptr) {}
    SongNode(Song* songPtr) : song(songPtr), next(nullptr) {}

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& song;
        ar& next;
    }
};

class Playlist
{
private:
    SongNode* head;
    SongNode* getNodeAtPosition(int pos);
    int getCount();
public:
    string name;
    Playlist();
    SongNode* getHead();
    Playlist(const string& playlistName) : name(playlistName), head(nullptr)
{}
    void addSong(Song* song);
    void removeSong(const string& songName);
    void displayPlaylist();
    void shufflePlaylist();
    Playlist& operator=(const Playlist& other);
    float calculatePlaylist();
    ~Playlist();
    SongNode* merge(SongNode* left, SongNode* right);
    SongNode* mergeSort(SongNode* head);
    void sortPlaylist();
    Song* getRandomSong();

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar& name;
        ar& head;
    }
};

void PrintStack(stack<Playlist*> p1) {
```

```
    cout << "Playlist Names in the Stack:" << endl;
    while (!p1.empty()) {
        cout << p1.top()->name << endl;
        p1.pop();
    }
}

class User {
public:
    string username;
    string password;
    stack<Playlist*> SavedPlaylists;
    stack<Song*> ListeningHistory;
    stack<Song*> LikedSongs;
    unordered_map<Song, int,Song::HashFunction,Song::EqualityComparison>
MostListenedToSongs;

    User(string usern, string passw);
    User();
    void PlayPlaylist(Playlist *p1);
    void displayProfile();
    float CalculateRecScore();
    Playlist* getRandomPlaylist()const;
    boost::optional<Playlist*> SearchPlaylist(const std::string& n);

    friend class boost::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
       ar& username;
        ar& password;
        ar& SavedPlaylists;
        ar& ListeningHistory;
       ar& LikedSongs;
        ar& MostListenedToSongs;
    }
};

class UserHashTable {
    vector<list<User>> HashTable;
    static const size_t TABLE_SIZE = 100;
    size_t HashFunction(string key);

public:
    void LoadFromFile();
    UserHashTable() : HashTable(TABLE_SIZE) {};
    void SaveToFile();
    void InsertUser(User& user);
```

```
    boost::optional<User> ValidateLogin(const string& username, const string&
password);

private:
    string hashpassword(const string& password);
};
class UserMenu {
public:
    void SignUpPrompt();
    void LoginPrompt();
    void Welcome();
    void MainMenu(User&u1,UserHashTable& uhs);
    void ProfileMenu(User&u1,UserHashTable&uhs);
    void PlaylistMenu(User& u1,UserHashTable &uhs);
    void PlaylistChoice(int choice,User&u1,UserHashTable &uhs);
    void Logo();
};
```

Source Code:

```
#include "Header.h"
#include "emoji.h"
#include <windows.h>
#include<cwchar>
#include <iostream>
#include <conio.h>
#include <fstream>
#include <stack>
#include "cryptlib.h"
#include "sha.h"
#include "hex.h"
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/access.hpp>
#include "boost/serialization/deque.hpp"
#include "boost/serialization/stack.hpp"
#include <boost/serialization/queue.hpp>
#include <boost/optional.hpp>
#include<boost/unordered_map.hpp>
#include <unordered_map>
#include <algorithm>
#include<cfloat>
#include <chrono>
#include <thread>
#include <cstdlib>
```

```
#include <ctime>
using namespace std;

UserMenu m1;

template <typename Func>
auto measureExecutionTime(Func func) {
    auto start = std::chrono::high_resolution_clock::now();
    func();
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end
- start);
    return duration.count();
}
void delay(int milliseconds) {
    std::this_thread::sleep_for(std::chrono::milliseconds(milliseconds));
}
bool randomFunc() {
    return(rand() % 2 == 1);
}
int randPlaylistOption()
{
    return rand() % 6 + 1;
}
int randProfileOption() //for profile and welcome
{
    return rand() % 2 + 1;
}
int randMainOption()
{
    return rand() % 3 + 1;
}

float Song::getScore() {
    return score;
}
string Song::getName()
{
    return name;
}
float Song::getDuration()
{
    return duration;
}
float Song::getEnergy()
{
    return energy;
```

```
}
float Song::getKey()
{
    return key;
}
float Song::getLoudness()
{
    return loudness;
}
float Song::getSpeechiness()
{
    return speechiness;
}
float Song::getAcousticness()
{
    return acousticness;
}
float Song::getInstrumentalness()
{
    return instrumentalness;
}
float Song::getLiveness()
{
    return liveness;
}
float Song::getTempo()
{
    return tempo;
}
void Song::setName(string n)
{
    name = n;
}
void Song::setDuration(float dur)
{
    duration = dur;
}
void Song::setEnergy(float en)
{
    energy = en;
}
void Song::setKey(float k)
{
    key = k;
}
void Song::setLoudness(float loud)
{
```

```
    loudness = loud;
}
void Song::setSpeechiness(float spch)
{
    speechiness = spch;
}
void Song::setAcousticness(float ac)
{
    acousticness = ac;
}
void Song::setInstrumentalness(float instr)
{
    instrumentalness = instr;
}
void Song::setLiveness(float live)
{
    liveness = live;
}
void Song::setTempo(float t) {
    tempo = t;
}
void Song::setID(string ID) {
    id = ID;
}

void Song::displayInfo()
{
    cout << "Name: " << name << endl;
    cout << "Duration: " << duration << endl;
    cout << "Energy: " << energy << endl;
    cout << "Key: " << key << endl;
    cout << "Loudness: " << loudness << endl;
    cout << "Speechiness: " << speechiness << endl;
    cout << "Acousticness: " << acousticness << endl;
    cout << "Instrumentalness: " << instrumentalness << endl;
    cout << "Liveness: " << liveness << endl;
    cout << "Tempo: " << tempo << endl;
}

float Song::CalcScore() const {
    float score = (energy + key + loudness + speechiness + acousticness +
instrumentalness + liveness + tempo) / 8;
    return score;
}
Song::Song() {

}
```

```
SongNode* Playlist::getHead() {
    return head;
}
SongNode* Playlist::getNodeAtPosition(int pos)
{
    SongNode* current = head;
    for (int i = 0; i < pos && current != NULL; ++i) {
        current = current->next;
    }
    return current;
}
float Playlist::calculatePlaylist() {
    float playlistScore = 0.0;
    SongNode* current = head;
    int i = 1;
    while (current != nullptr) {
        playlistScore += current->song->CalcScore();
        current = current->next;
        i++;
    }
    return playlistScore / i;
}
int Playlist::getCount() {
    int count = 0;
    SongNode* current = head;
    while (current != NULL) {
        ++count;
        current = current->next;
    }
    return count;
}
Playlist::Playlist() {
    head = NULL;
    name = " ";
}
void Playlist::addSong(Song* song)
{
    SongNode* newSong = new SongNode(song);
    if (head == NULL)
    {
        head = newSong;
    }
    else
    {
        SongNode* temp = head;
        while (temp->next != NULL)
        {
```

```
            temp = temp->next;
        }
        temp->next = newSong;
    }
    cout << song->getName() << " has been added to " << name << "." << endl;
}

void Playlist::removeSong(const string& songName) {
    SongNode* current = head;
    SongNode* prev = NULL;

    while (current != NULL && current->song->getName() != songName) {
        prev = current;
        current = current->next;
    }

    if (current != NULL) {
        if (prev != NULL) {
            prev->next = current->next;
        }
        else {
            head = current->next;
        }
        delete current;
        cout << songName << " has been removed from " << name << "." << endl;
    }
    else {
        cout << songName << " is not in " << name << "." << endl;
    }
}
void Playlist::displayPlaylist()
{
    cout << "-------------------------------------------------" << endl;
    cout << "Playlist: " << name << endl;
    cout << "-------------------------------------------------" << endl;
    SongNode* current = head;
    string temp;
    while (current != NULL)
    {
        temp = current->song->getName();
        cout << temp << endl;
        current = current->next;
    }
    cout << endl;
}
void Playlist::shufflePlaylist() {
    srand(time(0));
```

```
    int count = getCount();
    for (int i = count - 1; i > 0; --i) {
        int j = rand() % (i + 1);
        SongNode* songI = getNodeAtPosition(i);
        SongNode* songJ = getNodeAtPosition(j);
        swap(songI->song, songJ->song);
    }
    cout << name << " has been shuffled." << endl;
}
Song* Playlist::getRandomSong(){
    if (!head) {
        return nullptr; // Return nullptr if the playlist is empty
    }

    int totalSongs = getCount();

    // Generate a random index
    int randomIndex = rand() % totalSongs;

    // Traverse the playlist to the randomly selected position
    SongNode* randomNode = getNodeAtPosition(randomIndex);

    // Return the randomly selected song
    return (randomNode) ? randomNode->song : nullptr;
}
Playlist& Playlist::operator=(const Playlist& other) {
    if (this != &other) {
        while (head != NULL) {
            SongNode* temp = head;
            head = head->next;
            delete temp;
        }
        this->name = other.name;
        SongNode* otherCurrent = other.head;
        SongNode* current = NULL;
        while (otherCurrent != NULL) {
            Song* newSong = new Song(*otherCurrent->song);
            SongNode* newSongNode = new SongNode(newSong);

            if (current == NULL) {
                head = newSongNode;
            }
            else {
                current->next = newSongNode;
            }

            current = newSongNode;
```

```
            otherCurrent = otherCurrent->next;
        }
    }
    return *this;
}
Playlist::~Playlist() {
    while (head != NULL) {
        SongNode* temp = head;
        head = head->next;
        delete temp->song;
        delete temp;
    }
}
SongNode* Playlist::merge(SongNode* left, SongNode* right) {
    if (left == NULL) {
        return right;
    }
    if (right == NULL) {
        return left;
    }

    if (left->song->getName() < right->song->getName()) {
        left->next = merge(left->next, right);
        return left;
    }
    else {
        right->next = merge(left, right->next);
        return right;
    }
}
SongNode* Playlist::mergeSort(SongNode* head) {
    if (head == NULL || head->next == NULL)
    {
        return head;
    }
    SongNode* slow = head;
    SongNode* fast = head->next;
    while (fast != NULL && fast->next != NULL)
    {
        slow = slow->next;
        fast = fast->next->next;
    }
    SongNode* left = head;
    SongNode* right = slow->next;
    slow->next = NULL;
    left = mergeSort(left);
    right = mergeSort(right);
```

```
    return merge(left, right);
}
void Playlist::sortPlaylist()
{
    head = mergeSort(head);
    cout << name << " has been sorted alphabetically." << endl;
}
class AVLNode
{
public:
    Song song;
    AVLNode* left;
    AVLNode* right;
    int height;

    AVLNode(Song s) : song(s), left(NULL), right(NULL), height(1) {}
};

class AVLTree
{
private:
    AVLNode* root;

    int height(AVLNode* node)
    {
        return (node == NULL) ? 0 : node->height;
    }

    int balanceFactor(AVLNode* node)
    {
        return (node == NULL) ? 0 : height(node->left) - height(node->right);
    }

    void updateHeight(AVLNode* node)
    {
        node->height = 1 + max(height(node->left), height(node->right));
    }

    AVLNode* rotateRight(AVLNode* y)
    {
        AVLNode* x = y->left;
        AVLNode* T2 = x->right;

        x->right = y;
        y->left = T2;

        updateHeight(y);
```

```
    updateHeight(x);

    return x;
}

AVLNode* rotateLeft(AVLNode* x)
{
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

    y->left = x;
    x->right = T2;

    updateHeight(x);
    updateHeight(y);

    return y;
}
AVLNode* insert(AVLNode* node, Song song)
{
    if (node == NULL)
    {
        return new AVLNode(song);
    }

    if (song.getName() < node->song.getName())
    {
        node->left = insert(node->left, song);
    }
    else if (song.getName() > node->song.getName())
    {
        node->right = insert(node->right, song);
    }
    else
    {
        return node;
    }

    updateHeight(node);

    int balance = balanceFactor(node);
    if (balance > 1)
    {
        if (song.getName() < node->left->song.getName())
        {
            return rotateRight(node);
        }
```

```
        else
        {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
    }
    if (balance < -1) {
        if (song.getName() > node->right->song.getName())
        {
            return rotateLeft(node);
        }
        else
        {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }
    }

    return node;
}
Song* search(AVLNode* node, const string& lowerUserInput) {
    if (node == NULL) {
        return NULL;
    }
    string lowerSongName = toLower(node->song.getName());
    if (lowerSongName.find(lowerUserInput) != string::npos) {
        return &(node->song);
    }

    if (lowerUserInput < lowerSongName) {
        return search(node->left, lowerUserInput);
    }
    else {
        return search(node->right, lowerUserInput);
    }
}
string toLower(const string& str) {
    string result;
    for (size_t i = 0; i < str.length(); ++i) {
        result += tolower(str[i]);
    }
    return result;
}


void inOrderTraversal(AVLNode* node)
{
```

```
        if (node == NULL)
        {
            return;
        }
        string n;
        inOrderTraversal(node->left);
        n = node->song.getName();
        cout << n << endl;
        inOrderTraversal(node->right);
    }
    AVLNode* getIthNode(AVLNode* node, int i) const {
        if (!node) {
            return nullptr;
        }

        int leftSize = node->left ? node->left->height : 0;

        if (i == leftSize + 1) {
            return node;
        }
        else if (i < leftSize + 1) {
            return getIthNode(node->left, i);
        }
        else {
            return getIthNode(node->right, i - leftSize - 1);
        }
    }


public:
    AVLNode* getRandomNode() const {
        int totalNodes = root ? root->height : 0;
        int randomIndex = 1 + rand() % totalNodes;
        return getIthNode(root, randomIndex);
    }
    AVLTree() : root(NULL) {}

    void insert(Song song) {
        root = insert(root, song);
    }
    void displayInOrder()
    {
        cout << "In-Order Traversal:" << endl;
        inOrderTraversal(root);
    }
    Song* search(const string& songName) {
        string lowerUserInput = toLower(songName);
```

```
        return search(root, lowerUserInput);
    }
};

void readfileAndInsert(AVLTree& avlTree)
{
    string filename = "FinalDataSet.txt";
    ifstream file(filename.c_str());

    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    string line;
    while (getline(file, line))
    {
        stringstream ss(line);
        string token;

        Song song;
        getline(ss, token, ',');
        song.setID(token);
        getline(ss, token, ',');
        song.setName(token);
        getline(ss, token, ',');
        song.setDuration(atof(token.c_str()));
        getline(ss, token, ',');
        song.setEnergy(atof(token.c_str()));
        getline(ss, token, ',');
        song.setKey(atof(token.c_str()));
        getline(ss, token, ',');
        song.setLoudness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setSpeechiness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setAcousticness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setInstrumentalness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setLiveness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setTempo(atof(token.c_str()));
        getline(ss, token, ',');
        song.setScore(atof(token.c_str()));
        avlTree.insert(song);
    }
```

```
    file.close();
}

class AVLNode2 {
public:
    Song song;
    AVLNode2* left;
    AVLNode2* right;
    int height;

    AVLNode2(Song s) : song(s), left(NULL), right(NULL), height(1) {}
};

// AVLTree2 class
class AVLTree2 {
private:
    AVLNode2* root;

    int height(AVLNode2* node) {
        return (node == NULL) ? 0 : node->height;
    }

    int balanceFactor(AVLNode2* node) {
        return (node == NULL) ? 0 : height(node->left) - height(node->right);
    }

    void updateHeight(AVLNode2* node) {
        node->height = 1 + max(height(node->left), height(node->right));
    }

    AVLNode2* rotateRight(AVLNode2* y) {
        AVLNode2* x = y->left;
        AVLNode2* T2 = x->right;

        x->right = y;
        y->left = T2;

        updateHeight(y);
        updateHeight(x);

        return x;
    }

    AVLNode2* rotateLeft(AVLNode2* x) {
        AVLNode2* y = x->right;
        AVLNode2* T2 = y->left;
```

```
    y->left = x;
    x->right = T2;

    updateHeight(x);
    updateHeight(y);

    return y;
}

AVLNode2* insertScore(AVLNode2* node, Song song) {
    if (node == NULL) {
        return new AVLNode2(song);
    }

    if (song.getScore() < node->song.getScore()) {
        node->left = insertScore(node->left, song);
    }
    else if (song.getScore() > node->song.getScore()) {
        node->right = insertScore(node->right, song);
    }
    else {
        return node; // Ignoring duplicates
    }

    // Update height
    updateHeight(node);

    // Rebalance the tree
    int balance = balanceFactor(node);
    if (balance > 1) {
        if (song.getScore() < node->left->song.getScore()) {
            return rotateRight(node);
        }
        else {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
    }
    if (balance < -1) {
        if (song.getScore() > node->right->song.getScore()) {
            return rotateLeft(node);
        }
        else {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }
```

```
        }

        return node;
    }

    Song* search(AVLNode2* node, const string& lowerUserInput) {
        if (node == NULL) {
            return NULL;
        }

        string lowerScore = toLower(to_string(node->song.getScore()));
        if (lowerScore.find(lowerUserInput) != string::npos) {
            return &(node->song);
        }

        if (lowerUserInput < lowerScore) {
            return search(node->left, lowerUserInput);
        }
        else {
            return search(node->right, lowerUserInput);
        }
    }

    string toLower(const string& str) {
        string result;
        for (size_t i = 0; i < str.length(); ++i) {
            result += tolower(str[i]);
        }
        return result;
    }

    void inOrderTraversal(AVLNode2* node) {
        if (node == NULL) {
            return;
        }
        inOrderTraversal(node->left);
        cout << "Score: " << node->song.getScore() << " - Name: " <<
node->song.getName() << endl;
        inOrderTraversal(node->right);
    }
    Song* closestSong;
    float targetScore;
    float minDifference;

    void findClosestScoreHelper(AVLNode2* node) {
        if (node == NULL) {
            return;
```

```
        }
        float currentDifference = abs(targetScore - node->song.getScore());

        if (currentDifference < minDifference) {
            minDifference = currentDifference;
            closestSong = &(node->song);
        }
        if (targetScore < node->song.getScore()) {
            findClosestScoreHelper(node->left);
        }
        else {
            findClosestScoreHelper(node->right);
        }
    }
    Song* findClosestScore(AVLNode2* node, float targetScore, float&
minDifference, Song* closestSong) {
        if (node == NULL) {
            return closestSong;
        }

        float currentDifference = abs(node->song.getScore() - targetScore);
        if (currentDifference < minDifference) {
            minDifference = currentDifference;
            closestSong = &(node->song);
        }

        if (targetScore < node->song.getScore()) {
            return findClosestScore(node->left, targetScore, minDifference,
closestSong);
        }
        else {
            return findClosestScore(node->right, targetScore, minDifference,
closestSong);
        }
    }

public:
    AVLTree2() : root(NULL) {}

    void insertScore(Song song) {
        root = insertScore(root, song);
    }

    Song* search(AVLNode2* node, float targetScore) {
        if (node == NULL) {
            return NULL;
        }
```

```
        float nodeScore = node->song.getScore();
        if (targetScore == nodeScore) {
            return &(node->song);
        }

        if (targetScore < nodeScore) {
            return search(node->left, targetScore);
        }
        else {
            return search(node->right, targetScore);
        }
    }

    void displayInOrder() {
        cout << "In-Order Traversal:" << endl;
        inOrderTraversal(root);
    }
    Song* findClosestScore(float target) {
        closestSong = NULL;
        targetScore = target;
        minDifference = FLT_MAX;

        findClosestScoreHelper(root);

        return closestSong;
    }
    Song* searchClosestScore(float targetScore) {
        float minDifference = FLT_MAX;
        Song* closestSong = NULL;
        return findClosestScore(root, targetScore, minDifference,
closestSong);
    }
};
void readfileAndInsert2(AVLTree2& AVLTree2) {
    string filename = "FinalDataSet.txt";
    ifstream file(filename.c_str());

    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << endl;
        return;
    }

    string line;
    while (getline(file, line)) {
        stringstream ss(line);
        string token;
```

```
        Song song;
        getline(ss, token, ',');
        song.setID(token);
        getline(ss, token, ',');
        song.setName(token);
        getline(ss, token, ',');
        song.setDuration(atof(token.c_str()));
        getline(ss, token, ',');
        song.setEnergy(atof(token.c_str()));
        getline(ss, token, ',');
        song.setKey(atof(token.c_str());
        getline(ss, token, ',');
        song.setLoudness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setSpeechiness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setAcousticness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setInstrumentalness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setLiveness(atof(token.c_str()));
        getline(ss, token, ',');
        song.setTempo(atof(token.c_str()));
        getline(ss, token, ',');
        song.setStreams(atof(token.c_str()));
        song.setScore(song.CalcScore());
        AVLTree2.insertScore(song);
    }

    file.close();
}
void UserHashTable::SaveToFile() {
    ofstream ofs("user_data.txt");
    boost::archive::text_oarchive oarchive(ofs);
    for (const auto& userVec : HashTable) {
        for (const auto& user : userVec) {
            oarchive << user;
        }
    }
    cout << "User data saved successfully." << endl;
}
size_t UserHashTable::HashFunction(string key) {
    return hash < string>{}(key) % TABLE_SIZE;
}
string UserHashTable::hashpassword(const string& password) {
    using namespace CryptoPP;
```

```
    SHA1 hash;
    byte digest[SHA1::DIGESTSIZE];
    hash.CalculateDigest(digest, (const byte*)password.c_str(),
password.length());
    HexEncoder encoder;
    std::string hashedPassword;
    encoder.Attach(new StringSink(hashedPassword));
    encoder.Put(digest, sizeof(digest));
    encoder.MessageEnd();
    return hashedPassword;
}

void UserHashTable::InsertUser(User& user) {
    size_t index = HashFunction(user.username);
    user.password = hashpassword(user.password);
    HashTable[index].emplace_back(user);
}

void UserHashTable::LoadFromFile() {
    {
        ifstream ifs("user_data.txt");
        boost::archive::text_iarchive iarchive(ifs);

        try {
            while (true) {
                User loadedUser;
                iarchive >> loadedUser;
                InsertUser(loadedUser);
            }
        }
        catch (const boost::archive::archive_exception& e) {
            if (e.code ==
boost::archive::archive_exception::input_stream_error) {
                cout << "End of file reached." << endl;
            }
            else {
                cerr << "Error during deserialization: " << e.what() << endl;
            }
        }
    }

}
Playlist* User::getRandomPlaylist() const {
    if (SavedPlaylists.empty()) {
        return nullptr;  // Return nullptr if the stack is empty
    }
```

```
    // Copy the stack to a vector for random access
    std::vector<Playlist*> playlists;

    // Populate the vector using a while loop
    auto tempStack = SavedPlaylists; // Make a copy of the original stack
    while (!tempStack.empty()) {
        playlists.push_back(tempStack.top());
        tempStack.pop();
    }

    // Seed the random number generator with the current time
    std::srand(static_cast<unsigned>(std::time(nullptr)));

    // Generate a random index
    int randomIndex = std::rand() % playlists.size();

    // Return the randomly selected playlist
    return playlists[randomIndex];
}

boost::optional<User> UserHashTable::ValidateLogin(const string& username,
const string& password) {
    size_t index = HashFunction(username);
    for (const auto& user : HashTable[index]) {
        if (user.username == username) {
            string hashEnteredPassword = hashpassword(password);
            cout << "Stored Hash: " << user.password << endl;
            cout << "Entered Hash: " << hashEnteredPassword << endl;
            if (user.password == hashEnteredPassword) {
                return boost::make_optional(user);
            }
        }
    }
    return boost::none;
}
std::string generateRandomString(int length) {
    // Define the set of characters from which the random string will be
composed
    const std::string characters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

    // Seed the random number generator with the current time
    std::srand(static_cast<unsigned>(std::time(nullptr)));

    std::string randomString;
    for (int i = 0; i < length; ++i) {
        // Generate a random index to pick a character from the set
```

```
        int randomIndex = std::rand() % characters.size();
        // Append the randomly selected character to the string
        randomString += characters[randomIndex];
    }

    return randomString;
}

void UserMenu::PlaylistMenu(User& u1,UserHashTable &uhs) {
    Logo();
    system("cls");
    cout << "                                                                    1.
Display Playlist" << endl
        << "                                                                   2. Add
Song to Playlist" << endl
        << "                                                                   3.
Remove Song from Playlist" << endl
        << "                                                                   4.
Shuffle Playlist" << endl
        << "                                                                   5.
Create Playlist" << endl
        << "                                                                   6.
Sort Playlist in Alphabatical Order" << endl;
    int choice = randPlaylistOption();
    switch (choice) {

    case 5:
    {
        string name;
        int choice;
        cout << "Enter a name for playlist" << endl;
        name = generateRandomString(10);
        Playlist playlist(name);
        u1.SavedPlaylists.push(&playlist);
        cout << "Playlist Created.\n \t \t \t \t \tEnter 1 to return to
PlaylistMenu\n \t \t \t \t \tEnter 2 to return to MainMenu" << endl;
        choice = randProfileOption();
        switch (choice) {
        case 1: {
            m1.PlaylistMenu(u1,uhs);
        }
        case 2: {
            m1.MainMenu(u1,uhs);
        }
        }

    }
```

```
    case 1:case 2:case 3: case 4: case 6: {
        PlaylistChoice(choice,u1,uhs);
        break;
    }
   }
}
void Continue(User&u1,UserHashTable&uhs) {
    cout << "Enter 1 to Return to Main Menu" << endl;
    delay(1000);
    int choice = 1;
    switch (choice) {
    case 1:
    {
        m1.MainMenu(u1,uhs);
        break;
    }
    default: {
        Continue(u1,uhs);
    }
    }
}
void UserMenu::PlaylistChoice(int choice, User& u1, UserHashTable&uhs) {
    string playlistname;
    cout << "Enter playlist to perform the action on: ";
    playlistname = randPlaylistOption();

    boost::optional<Playlist*> playlistResult =
u1.SearchPlaylist(playlistname);

    if (playlistResult) {
        Playlist* playlist = *playlistResult;

        switch (choice) {
        case 1:
            playlist->displayPlaylist();
            Continue(u1,uhs);
            break;

        case 2: {
            AVLTree avlTree;

            string filename = "FinalDataSet.txt";

            ifstream file(filename.c_str());

            if (!file.is_open()) {
                cerr << "Error opening file: " << filename << endl;
```

```
        break;
    }

    readfileAndInsert(avlTree);

    string find;
    cout << "Enter the song you wish to search:" << endl;
    find = avlTree.getRandomNode()->song.getName();
    Song* foundSong = avlTree.search(find);
    string str1;
    float dur;
    if (foundSong != NULL)
    {
        cout << "Song found:" << endl;
        str1 = foundSong->getName();
        dur = foundSong->getDuration();
        cout << str1 << endl;
        cout << "Duration:" << dur << endl;
        playlist->addSong(foundSong);
    }
    else
    {
        cout << "Song not found." << endl;
    }
    Continue(u1,uhs);
    break;
}

case 3: {
    string songName;
    cout << "Enter the name of the song to remove: ";
    songName = playlist->getRandomSong()->getName();
    playlist->removeSong(songName);
    Continue(u1,uhs);
    break;
}

case 4:{
    playlist->shufflePlaylist();
    Continue(u1,uhs);
    break;
}
case 6: {
    playlist->mergeSort(playlist->getHead());
    playlist->displayPlaylist();
    break;
```

```
        }
        default:
            cout << "Invalid choice. Please enter a valid option." << endl;
        }
    }
    else {
        cout << "Playlist not found." << endl;
    }
}

void UserMenu::ProfileMenu(User& u1,UserHashTable &uhs) {
    system("cls");
    Logo();
    cout << "
1. View Profile" << endl
        << "                                                    2. Logout" <<
endl;

    int choice = randProfileOption();

    switch (choice) {
    case 1:
    {    u1.displayProfile();
    Continue(u1, uhs);
    break;
    }
    case 2:
    {   cout << "Logging out..." << endl;
    uhs.SaveToFile();
    break;
    }
    default:
        cout << "Invalid choice. Please enter a valid option." << endl;
        ProfileMenu(u1,uhs);
    }
}

void UserMenu::MainMenu(User&u1,UserHashTable &uhs) {
    system("cls");
    int choice;
    cout << "                                                Main Menu:" <<
endl;
    cout << "                                                1. Profile Menu" <<
endl <<
            "                                                2.Playlist Menu" <<
endl <<
```

```
            "                                                    3.Recommendations"
<< endl;
    choice = randMainOption();
    switch (choice) {
    case 1: {
        ProfileMenu(u1,uhs);
        break;
    }
    case 2:
    {
        PlaylistMenu(u1,uhs);
        break;
    }
    case 3:
    {
        AVLTree2 avlTree2;

        string filename2 = "FinalDataSet.txt";

        ifstream file2(filename2.c_str());

        if (!file2.is_open()) {
            cerr << "Error opening file: " << filename2 << endl;
            break;
        }

        readfileAndInsert2(avlTree2);
        float userScore = u1.CalculateRecScore();
        cout << u1.CalculateRecScore();
        Song* closestSong = avlTree2.searchClosestScore(userScore);

        if (closestSong != NULL) {
            cout << "Closest matching song found:" << endl;
            cout << "Score: " << closestSong->getScore() << " - Name: " <<
closestSong->getName() << endl;
        }
        else {
            cout << "No matching song found." << endl;
        }
        Continue(u1,uhs);
        break;
    }
    }
}
```

```
void UserMenu::Welcome() {

    Logo();
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD cursorpos;
    cursorpos.X = 45;
    cursorpos.Y = 6;
    SetConsoleCursorPosition(consoleHandle, cursorpos);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN | FOREGROUND_RED |
FOREGROUND_BLUE);
    cout << "Press ";
    SetConsoleTextAttribute(consoleHandle,FOREGROUND_BLUE);
    cout << "TAB ";
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN | FOREGROUND_RED |
FOREGROUND_BLUE);
    cout<<"To SignUp!";
    cursorpos.X = 45;
    cursorpos.Y = 7;

    SetConsoleCursorPosition(consoleHandle, cursorpos);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN | FOREGROUND_RED |
FOREGROUND_BLUE);
    cout << "Press ";
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE);
    cout << "Enter";
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_GREEN | FOREGROUND_RED |
FOREGROUND_BLUE);
    cout << " To Login!";

    delay(10000);
    while (1) {
        if (randomFunc()) {
            LoginPrompt();
            break;
        }
        if (!randomFunc()) {
            SignUpPrompt();
            break;
        }
     /*  if (GetAsyncKeyState(VK_RETURN)) {
            LoginPrompt();
            break;
        }
        if (GetAsyncKeyState(VK_TAB)) {
            SignUpPrompt();
            break;
        }*/
```

```
    }
}

void UserMenu::Logo() {
    SetConsoleOutputCP(CP_UTF8);
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD consoleMode;
    GetConsoleMode(consoleHandle, &consoleMode);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE |
FOREGROUND_INTENSITY);
    cout << "\t\t\t\t\t\tMuSicHub" << endl;
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_INTENSITY);
    cout << "\t\t\t\tRecommending Songs Through User Behaviour" << endl;
    COORD cursorpos;
    cursorpos.X = 85;
    cursorpos.Y = 2;
    SetConsoleCursorPosition(consoleHandle, cursorpos);

    float dist;
    for (int i = 0; i <= 2 * 3; i++) {
        for (int j = 0; j <= 2 * 3; j++) {
            dist = sqrt((i - 3) * (i - 3) +
                (j - 3) * (j - 3));
            if (dist > 3 - 0.5 && dist < 3 + 0.5)
                cout << emojicpp::emojize(":musical_note:");
            else
                cout << " ";
        }

        cursorpos.Y++;
        SetConsoleCursorPosition(consoleHandle, cursorpos);
    }
}

void UserMenu::LoginPrompt() {
    system("cls");
    Logo();
    string username; char password[100];
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD consoleMode;
    COORD coord;
    coord.X = 50;
    coord.Y = 2;
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_INTENSITY);
    SetConsoleCursorPosition(consoleHandle, coord);
    GetConsoleMode(consoleHandle, &consoleMode);
    cout << "Enter username : " << endl;
```

```
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE | FOREGROUND_GREEN
| FOREGROUND_RED);
    SetConsoleCursorPosition(consoleHandle, coord);
    GetConsoleMode(consoleHandle, &consoleMode);
    coord.X = 68;
    coord.Y = 2;
    SetConsoleCursorPosition(consoleHandle, coord);
    cin >> username;
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_INTENSITY);
    coord.X = 50;
    coord.Y = 3;
    SetConsoleCursorPosition(consoleHandle, coord);
    cout << "Enter password : " << endl;
    coord.X = 68;
    coord.Y = 3;
    SetConsoleCursorPosition(consoleHandle, coord);
    SetConsoleTextAttribute(consoleHandle, FOREGROUND_BLUE | FOREGROUND_GREEN
| FOREGROUND_RED);
    int k=1;
    while (password[k - 1] != '\r')
    {
        password[k] = _getch();
        if (password[k - 1] != '\r')
        {
            cout << "*";
        }
        k++;
    }
    UserHashTable uhs;
    uhs.LoadFromFile();
    string pass_a = password;
    boost::optional<User> validatedUser = uhs.ValidateLogin(username,
password);
    if (uhs.ValidateLogin(username, pass_a)) {
        cout << "\nLogin Successful!" << endl;
        User u1 = *validatedUser;
        MainMenu(u1,uhs);
    }
    else
    {
        cout << "Login Unsuccessful!" << endl;
        Welcome();
    }
}
float User::CalculateRecScore() {
    float totalScore = 0.0;
```

```
    stack<Playlist*> tempStack = SavedPlaylists;
    int i = 1;
    while (!tempStack.empty()) {
        Playlist* currentPlaylist = tempStack.top();
        tempStack.pop();
        i++;
        totalScore += currentPlaylist->calculatePlaylist();
    }
    return totalScore / i;
}
boost::optional<Playlist*> User::SearchPlaylist(const std::string& n) {
    Playlist* temp;
    std::stack<Playlist*> tempStack = SavedPlaylists;
    while (!tempStack.empty()) {
        temp = tempStack.top();
        if (temp->name == n) {
            return boost::optional<Playlist*>(temp);
        }
        tempStack.pop();
    }
    return boost::none;
}
User::User() {

}
void User::displayProfile()
{
    system("cls");
    HANDLE conhandle = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD cord;
    cord.X = 50;
    cord.Y = 3;
    SetConsoleCursorPosition(conhandle, cord);
    cout << "Your Playlists : : ";
    PrintStack(SavedPlaylists);
}
User::User(string username, string password) {
    this->username = username;
    this->password = password;
}
void UserMenu::SignUpPrompt() {
    bool signUpSuccess = false;

        system("cls");
        Logo();
        string username;
        char password[100];
```

```
        HANDLE conhandle = GetStdHandle(STD_OUTPUT_HANDLE);
        COORD cord;
        cord.X = 40;
        cord.Y = 3;
        SetConsoleCursorPosition(conhandle, cord);
        cout << "Enter Username : ";
        cord.Y++;
        SetConsoleCursorPosition(conhandle, cord);
        cin >> username;
        cord.Y++;
        SetConsoleCursorPosition(conhandle, cord);
        cout << "Enter Password : ";
        cord.Y++;
        SetConsoleCursorPosition(conhandle, cord);
        int k = 1;
        while (password[k - 1] != '\r') {
            password[k] = _getch();
            if (password[k - 1] != '\r') {
                cout << "*";
            }
            k++;
        }

        {

            std::ofstream ofs("user_data.txt",ios::app);
            boost::archive::text_oarchive oarchive(ofs);

            User user1(username, password);

            oarchive << user1;
            cout << "\n                                Signed Up Successfully" <<
endl;
            system("pause");
         /*   MainMenu(user1);*/

        }
}

void Song::setStreams(int str) {
    streams = str;
}
void Song::setScore(float scor) {
    score = scor;
}

Song* search(AVLNode2* node, float targetScore) {
```

```
    if (node == NULL) {
        return NULL;
    }

    float nodeScore = node->song.getScore();
    if (targetScore == nodeScore) {
        return &(node->song);
    }

    if (targetScore < nodeScore) {
        return search(node->left, targetScore);
    }
    else {
        return search(node->right, targetScore);
    }
}

int main() {
    auto startTime = std::chrono::high_resolution_clock::now();
    User u1;
    UserMenu u;
    UserHashTable uhs;
    uhs.LoadFromFile();
    u.MainMenu(u1,uhs);
    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(endTime - startTime);
    std::cout << "Execution Time: " << duration.count() << " microseconds" <<
std::endl;

}
```