Ethan Fancher

Project 1

**Menu code:**

// Course class definition

class Course {

public:

string courseNumber;

string title;

vector<string> prerequisites;

};

// Function to load data into the data structure

vector<Course> loadDataStructure(string filename) {

vector<Course> courses;

ifstream file(filename);

string line;

if (file.is\_open()) {

while (getline(file, line)) {

if (line.empty()) {

// Skip empty lines

continue;

}

Course course;

// Parse and populate course object from the line

// ...

courses.push\_back(course);

}

file.close();

} else {

cout << "Error opening file: " << filename << endl;

// Handle the error as desired

}

return courses;

}

// Function to print the alphanumerically ordered list of courses

void printCourseList(const vector<Course>& courses) {

// Sort the courses by course number

vector<Course> sortedCourses = courses;

sort(sortedCourses.begin(), sortedCourses.end(),

[](const Course& c1, const Course& c2) {

return c1.courseNumber < c2.courseNumber;

});

// Print the sorted list of courses

for (const auto& course : sortedCourses) {

cout << course.courseNumber << " - " << course.title << endl;

}

}

// Function to print the course title and prerequisites for an individual course

void printCourse(const vector<Course>& courses, const string& courseNumber) {

// Find the course in the data structure

auto it = find\_if(courses.begin(), courses.end(),

[&courseNumber](const Course& course) {

return course.courseNumber == courseNumber;

});

if (it != courses.end()) {

// Print course information

cout << "Course Number: " << it->courseNumber << endl;

cout << "Title: " << it->title << endl;

// Print prerequisites

cout << "Prerequisites: ";

if (it->prerequisites.empty()) {

cout << "None";

} else {

for (const auto& prerequisite : it->prerequisites) {

cout << prerequisite << " ";

}

}

cout << endl;

} else {

cout << "Course not found." << endl;

// Handle the error as desired

}

}

int main() {

string filename = "CourseData.txt"; // Replace with your file name

vector<Course> courses;

int choice;

bool exitMenu = false;

while (!exitMenu) {

// Print the menu options

cout << "Menu:" << endl;

cout << "1. Load Data Structure" << endl;

cout << "2. Print Course List" << endl;

cout << "3. Print Course" << endl;

cout << "4. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

// Load Data Structure

courses = loadDataStructure(filename);

cout << "Data structure loaded successfully." << endl;

break;

case 2:

// Print Course List

if (!courses.empty()) {

printCourseList(courses);

} else {

cout << "Data structure is empty. Load the data first." << endl;

}

break;

case 3:

// Print Course

if (!courses.empty()) {

string courseNumber;

Previous code:

using namespace std;

// Structure to hold bid information

struct Bid {

string bidId;

string title;

string fund;

double amount;

Bid() {

amount = 0.0;

}

};

// Node structure for chaining in the hash table

struct Node {

Bid bid;

unsigned key;

Node\* nextNodePtr;

Node() {

key = UINT\_MAX;

nextNodePtr = nullptr;

}

Node(Bid myBid) : Node() {

bid = myBid;

}

Node(Bid myBid, unsigned newKey) : Node(myBid) {

key = newKey;

}

};

class HashTable {

private:

vector<Node> myNodes;

unsigned setSize;

unsigned int hash(int key);

public:

HashTable();

HashTable(unsigned size);

~HashTable();

void Insert(Bid bid);

void PrintAll();

void Remove(string bidId);

Bid Search(string bidId);

};

unsigned int HashTable::hash(int key) {

return key % setSize;

}

HashTable::HashTable() {

setSize = DEFAULT\_SIZE;

myNodes.resize(setSize);

}

HashTable::HashTable(unsigned size) {

setSize = size;

myNodes.resize(setSize);

}

HashTable::~HashTable() {

myNodes.clear();

}

void HashTable::Insert(Bid bid) {

unsigned key = hash(atoi(bid.bidId.c\_str()));

Node\* prevNode = &(myNodes.at(key));

if (prevNode == nullptr) {

Node\* nextNode = new Node(bid, key);

myNodes.insert(myNodes.begin() + key, (\*nextNode));

}

else {

if (prevNode->key == UINT\_MAX) {

prevNode->key = key;

prevNode->bid = bid;

prevNode->nextNodePtr = nullptr;

}

else {

while (prevNode->nextNodePtr != nullptr) {

prevNode = prevNode->nextNodePtr;

}

}

}

}

void HashTable::PrintAll() {

for (unsigned int i = 0; i < myNodes.size(); ++i) {

cout << myNodes[i].bid.bidId << ": " << myNodes[i].bid.title << " | "

<< myNodes[i].bid.amount << " | " << myNodes[i].bid.fund << endl;

}

}

void HashTable::Remove(string bidId) {

unsigned key = hash(atoi(bidId.c\_str()));

myNodes.erase(myNodes.begin() + key);

}

Bid HashTable::Search(string bidId) {

Bid bid;

unsigned key = hash(atoi(bidId.c\_str()));

Node\* node = &(myNodes.at(key));

if (node != nullptr && node->key != UINT\_MAX && node->bid.bidId.compare(bidId) == 0) {

return node->bid;

}

if (node == nullptr || node->key == UINT\_MAX) {

return bid;

}

while (node != nullptr) {

if (node->key != UINT\_MAX && node->bid.bidId.compare(bidId) == 0) {

return node->bid;

}

node = node->nextNodePtr;

}

return bid;

}

void loadBids(string csvPath, HashTable\* hashTable) {

cout << "Loading CSV file " << csvPath << endl;

csv::Parser file = csv::Parser(csvPath);

for (unsigned int i = 0; i < file.rowCount(); ++i) {

Bid bid;

bid.bidId = file[i][0];

bid.title = file[i][1];

bid.fund = file[i][2];

bid.amount = stod(file[i][3]);

hashTable->Insert(bid);

}

}

// Vector

void openFile(string filename) {

try {

file = open(filename, "r");

return file;

} catch (IOError) {

print "Error opening file.";

return NULL;

}

}

void readDataFromFile(string filename, Vector<Course> courseVector) {

file = openFile(filename);

if (file == NULL) {

return;

}

for (line in file) {

if (line.isEmpty()) {

continue;

}

courseData = line.split(","); // Assuming comma-separated values

if (courseData.length >= 2) {

courseNumber = courseData[0];

courseTitle = courseData[1];

prerequisites = [];

if (courseData.length > 2) {

for (i = 2; i < courseData.length; i++) {

prerequisite = courseData[i];

prerequisites.append(prerequisite);

}

}

course = createCourseObject(courseNumber, courseTitle, prerequisites);

courseVector.append(course);

} else {

print "Invalid line format: " + line;

}

}

file.close();

}

void printCourseInfo(Vector<Course> courseVector, string courseNumber) {

foundCourse = NULL;

for (course in courseVector) {

if (course.courseNumber == courseNumber) {

foundCourse = course;

break;

}

}

if (foundCourse == NULL) {

print "Course not found.";

return;

}

print "Course Number: " + foundCourse.courseNumber;

print "Course Title: " + foundCourse.courseTitle;

if (foundCourse.prerequisites.length > 0) {

print "Prerequisites: ";

for (prerequisite in foundCourse.prerequisites) {

print prerequisite;

}

}

}

// Tree

// Course class definition

class Course {

public:

string courseNumber;

string title;

vector<string> prerequisites;

};

// Function to parse a line and create a Course object

Course parseCourseLine(string line) {

Course course;

string delimiter = ",";

// Extract course number

size\_t pos = line.find(delimiter);

course.courseNumber = line.substr(0, pos);

line.erase(0, pos + delimiter.length());

// Extract course title

pos = line.find(delimiter);

course.title = line.substr(0, pos);

line.erase(0, pos + delimiter.length());

// Extract prerequisites (if any)

while ((pos = line.find(delimiter)) != string::npos) {

string prerequisite = line.substr(0, pos);

course.prerequisites.push\_back(prerequisite);

line.erase(0, pos + delimiter.length());

}

// Add the last prerequisite (if any)

if (!line.empty()) {

course.prerequisites.push\_back(line);

}

return course;

}

// Function to validate file format and create course objects

vector<Course> processCourseFile(string filename) {

vector<Course> courses;

unordered\_map<string, bool> prerequisitesMap;

ifstream file(filename);

string line;

if (file.is\_open()) {

while (getline(file, line)) {

if (line.empty()) {

// Skip empty lines

continue;

}

Course course = parseCourseLine(line);

courses.push\_back(course);

// Store prerequisites in a map for validation

for (const auto& prerequisite : course.prerequisites) {

prerequisitesMap[prerequisite] = true;

}

}

file.close();

// Check for file format errors

for (const auto& course : courses) {

for (const auto& prerequisite : course.prerequisites) {

if (!prerequisitesMap.count(prerequisite)) {

cout << "File format error: Prerequisite '" << prerequisite << "' not found." << endl;

// Handle the error as desired

}

}

}

} else {

cout << "Error opening file: " << filename << endl;

// Handle the error as desired

}

return courses;

}

// Function to print course information and prerequisites

void printCourseInformation(const vector<Course>& courses) {

for (const auto& course : courses) {

cout << "Course: " << course.courseNumber << endl;

cout << "Title: " << course.title << endl;

cout << "Prerequisites: ";

if (course.prerequisites.empty()) {

cout << "None";

} else {

for (const auto& prerequisite : course.prerequisites) {

cout << prerequisite << " ";

}

}

cout << endl << endl;

}

}

int main() {

string filename = "CourseInformation.txt"; // Replace with your file name

vector<Course> courses = processCourseFile(filename);

// Print course information

printCourseInformation(courses);

return 0;

}

**Evaluation:**

1. Vector:

- openFile(filename): Cost = 1, Executes once.

- readDataFromFile(filename, courseVector):

- Loop over each line in the file: Cost = n, Executes n times.

- Split line into courseData: Cost = 1, Executes n times.

- Loop over each prerequisite in courseData: Cost = m (average number of prerequisites per course), Executes n \* m times.

- Append course to courseVector: Cost = 1, Executes n times.

- printCourseInfo(courseVector, courseNumber):

- Loop over each course in courseVector: Cost = n, Executes once.

- Check course.courseNumber == courseNumber: Cost = 1, Executes n times.

- Print course information: Cost = 1, Executes once.

Total running time for vector implementation: O(n \* m)

Advantages: Simple to implement, provides fast random access to elements by index.

Disadvantages: Insertion and deletion of elements can be slow, especially in the middle of the vector.

2. Hash Table:

- processCourseFile(filename):

- Loop over each line in the file: Cost = n, Executes n times.

- Split line into courseData: Cost = 1, Executes n times.

- Loop over each prerequisite in courseData: Cost = m, Executes n \* m times.

- Create and store course objects in hash table: Cost = 1 (expected constant time for insertion), Executes n times.

- Store prerequisites in a map for validation: Cost = 1 (expected constant time for insertion), Executes n \* m times.

- Check for file format errors: Cost = m (average number of prerequisites per course), Executes n \* m times.

- printCourseInformation(courses):

- Loop over each course in courses: Cost = n, Executes once.

- Print course information: Cost = 1, Executes once.

Total running time for hash table implementation: O(n \* m)

Advantages: Provides fast access to course objects by course number, efficient insertion and deletion of elements.

Disadvantages: Slightly more complex to implement than a vector, requires a hashing function.

3. Tree:

- parseCourseLine(line):

- Extract course number: Cost = 1, Executes once.

- Extract course title: Cost = 1, Executes once.

- Loop over each prerequisite in line: Cost = m, Executes m times.

- Create course object: Cost = 1, Executes once.

- processCourseFile(filename):

- Loop over each line in the file: Cost = n, Executes n times.

- Parse each line and create course objects: Cost = m (average number of prerequisites per course), Executes n \* m times.

- printCourseInformation(courses):

- Loop over each course in courses: Cost = n, Executes once.

- Print course information: Cost = 1, Executes once.

Total running time for tree implementation: O(n \* m)

Advantages: Provides hierarchical organization of courses, efficient searching and traversal operations.

Disadvantages: Insertion and deletion operations can be slower compared to a hash table or vector.

Based on the Big O analysis results and the evaluation of the three data structures, the hash table implementation seems to be the most suitable choice. It offers fast access to course objects by course number, efficient insertion and deletion of elements, and the ability to handle potential file format