Tammy Wilson

CS 300 Analysis and Design

Module Six

Project One

Professor. Cory Thoma

April 10,2025

**Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure**

Each step in your pseudocode has a certain cost in terms of time and memory. Opening the file once, so it takes a small amount of time (O(1)). Reading each line runs n times because there are n courses in the file (O(n)). Splitting each line into various parts (course number, name, prerequisites) happens for each line, and the number of parts can vary (O(m)). Checking if the format is correct is quick (O(1)) but checking prerequisites in a set can take a little longer possibly up to O(n) in the worst case. Creating and storing a course in a vector is fast (O(1)), though if the vector needs more space, it could temporarily slow down (O(n)). Overall, reading and processing the file takes O(n \* m) time because it loops through every course and its prerequisites. Memory usage also depends on the number of courses and how much data each course holds, making it O(n \* m).

**Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.**

Each line in the pseudocode has a cost based on how many times it runs. Opening the file happens once, so its cost is 1. Reading each line of the file runs n times, since there are n courses, making its cost O(n). Splitting each line into various parts (course number, name, and prerequisites) happens for each course and can have m parts, leading to a cost of O(m) per course. Checking if the format is correct happens once per line and has a cost of O(1). Checking prerequisites may take O(1) if it is quick, but in the worst case, it could check all n courses and take O(n) time. Creating and storing a course object is fast, with a cost of 1, but adding it to a vector can take O(1) unless the vector needs to grow, which could make it O(n) for that moment. Keeping track of course numbers in a set is usually O(1) per course. Finally, closing the file happens once and has a cost of 1. Overall, the worst-case time complexity of reading and storing courses is O(n \* m) + O(n²) for prerequisite validation, and the memory usage grows with O(n \* m) since each course holds multiple details.

**Based on the advisor’s requirements, analyze each of the vector, hash table, and tree data structures. Explain the advantages and disadvantages of each structure in your evaluation.**

Each data structure—vector, hash table, and binary search tree—has benefits and drawbacks when used in the advising program. Vectors are simple and store courses in order, making them great for printing a sorted list. However, searching for a course in a vector can be slow because it requires checking each course one by one. Hash tables allow fast lookups, meaning you can find a course quickly. They are great for searching but do not keep courses in a sorted order, which means extra work is needed if a sorted list is required. Binary search trees (BSTs) automatically keep courses sorted, making them an excellent choice for displaying courses in order. They also allow faster searching compared to vectors, but they can become unbalanced, making some searches slower. Choosing the best structure depends on the program’s needs—vectors work well for printing sorted lists, hash tables are best for quick lookups, and BSTs are a good middle ground if sorting and searching are both important.

**Provide justification for your recommendation based on the Big O analysis results and your analysis of the three data structures.**

Based on the analysis of different data structures, I recommend using a hash table for the advising program. A hash table is the best choice because it allows quick searches, meaning you can find a course instantly instead of scanning through a long list. Unlike a vector, which takes O(n) time to search, a hash table finds a course in O(1) time in most cases. While a binary search tree (BST) also helps with searching, it takes O(log n) time, which is slower than a hash table.

The downside of a hash table is that it does not keep courses in order, so printing a sorted list would require extra work. However, since searching for courses happens more often than printing, the speed advantage makes hash tables the better choice. Insertion and deletion are also fast (O(1)), compared to O(n) in a vector and O(log n) in a BST. Although hash tables use slightly more memory because of their internal structure, this tradeoff is acceptable for improving speed.

Overall, since the advising system needs fast course lookups, a hash table is the most efficient option. Sorting can be done separately when needed, making this the best balance between speed and functionality.

**Pseudocode for Prerequisites**

// Define a structure for a Course

struct Course {

string courseNumber; // Stores the course number

string courseName; // Stores the course name

vector<string> prerequisites; // Stores prerequisites

};

// Function to load and validate course data from a file

vector<Course> loadCourseData(string fileName) {

// Open the file

ifstream file(fileName);

if (!file.is\_open()) {

cout << "Error: Unable to open file" << endl;

return {}; // Return an empty vector

}

// Initialize vector to store courses

vector<Course> courses;

set<string> courseNumbers; // Track valid course numbers

// Read each line from the file

string line;

while (getline(file, line)) {

stringstream ss(line);

string token;

vector<string> tokens;

// Parse the line into tokens

while (getline(ss, token, ',')) {

tokens.push\_back(token);

}

// Validate the line format

if (tokens.size() < 2) {

cout << "Error: Invalid line format (less than 2 parameters)" << endl;

continue;

}

// Extract course detail

string courseNumber = tokens[0];

string courseName = tokens[1];

vector<string> prerequisites;

// Validate prerequisites

for (size\_t i = 2; i < tokens.size(); ++i) {

if (courseNumbers.find(tokens[i]) == courseNumbers.end()) {

cout << "Error: Prerequisite " << tokens[i] << " does not exist in the file" << endl;

continue;

}

prerequisites.push\_back(tokens[i]);

}

// Create a Course object and store it

Course course = {courseNumber, courseName, prerequisites};

courses.push\_back(course);

courseNumbers.insert(courseNumber);

}

// Close the file

file.close();

// Return the loaded course data

return courses;

}

// Function to print course information

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

for (const Course& course : courses) {

if (course.courseNumber == courseNumber) {

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

cout << "Course Name: " << course.courseName << endl;

if (course.prerequisites.empty()) {

cout << "No prerequisites" << endl;

} else {

cout << "Prerequisites:" << endl;

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

cout << " - " << prerequisite << endl;

}

}

return;

}

}

cout << "Error: Course " << courseNumber << " not found" << endl;

}

// Function to print all courses sorted alphanumerically

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

if (courses.empty()) {

cout << "No courses available. Load data first." << endl;

return;

}

// Sort courses by course number

vector<Course> sortedCourses = courses;

sort(sortedCourses.begin(), sortedCourses.end(), [](Course a, Course b) {

return a.courseNumber < b.courseNumber;

});

cout << "Courses (sorted):" << endl;

for (const Course& course : sortedCourses) {

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

}

}

**Pseudocode for Menu**

// Define the Course structure

struct Course {

string courseNumber;

string courseName;

vector<string> prerequisites;

};

// Function to load course data from a file

vector<Course> loadCourseData(string fileName) {

ifstream file(fileName);

if (!file.is\_open()) {

cout << "Error: Unable to open file" << endl;

return {};

}

vector<Course> courses;

set<string> courseNumbers;

string line;

while (getline(file, line)) {

stringstream ss(line);

string token;

vector<string> tokens;

while (getline(ss, token, ',')) {

tokens.push\_back(token);

}

if (tokens.size() < 2) {

cout << "Error: Invalid line format (less than 2 parameters)" << endl;

continue;

}

string courseNumber = tokens[0];

string courseName = tokens[1];

vector<string> prerequisites;

for (size\_t i = 2; i < tokens.size(); ++i) {

prerequisites.push\_back(tokens[i]);

}

Course course = {courseNumber, courseName, prerequisites};

courses.push\_back(course);

courseNumbers.insert(courseNumber);

}

file.close();

return courses;

}

// Function to print an alphanumerically sorted list of courses

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

if (courses.empty()) {

cout << "No courses available. Load data first." << endl;

return;

}

vector<Course> sortedCourses = courses;

sort(sortedCourses.begin(), sortedCourses.end(), [](Course a, Course b) {

return a.courseNumber < b.courseNumber;

});

cout << "Courses (sorted):" << endl;

for (const Course& course : sortedCourses) {

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

}

}

// Function to print details of an individual course

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

for (const Course& course : courses) {

if (course.courseNumber == courseNumber) {

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

cout << "Course Name: " << course.courseName << endl;

if (course.prerequisites.empty()) {

cout << "No prerequisites" << endl;

} else {

cout << "Prerequisites:" << endl;

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

cout << " - " << prerequisite << endl;

}

}

return;

}

}

cout << "Error: Course " << courseNumber << " not found" << endl;

}

// Menu function

void displayMenu() {

vector<Course> courses;

int choice;

do {

cout << "\nMenu:" << endl;

cout << "1. Load course data" << endl;

cout << "2. Print sorted list of courses" << endl;

cout << "3. Print course details" << endl;

cout << "9. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

string fileName;

cout << "Enter the file name: ";

cin >> fileName;

courses = loadCourseData(fileName);

break;

}

case 2:

printSortedCourses(courses);

break;

case 3: {

string courseNumber;

cout << "Enter the course number: ";

cin >> courseNumber;

printCourseDetails(courses, courseNumber);

break;

}

case 9:

cout << "Exiting the program. Goodbye!" << endl;

break;

default:

cout << "Invalid option. Please try again." << endl;

}

} while (choice != 9);

}

// Main function

int main() {

displayMenu();

return 0;

}

**Pseudocode for Sorting and Printing Courses**

**Using Vector**

// Define a Course structure

struct Course {

string courseNumber;

string courseName;

vector<string> prerequisites;

};

// Function to sort and print courses using a vector

void printSortedCoursesVector(vector<Course> courses) {

if (courses.empty()) {

print "No courses available. Load data first."

return

}

// Sort courses alphanumerically

sort(courses.begin(), courses.end(), (Course a, Course b) -> return a.courseNumber < b.courseNumber)

// Print sorted courses

print "Courses in sorted order:"

for each course in courses:

print course.courseNumber + " - " + course.courseName

}

**Using Hash Table**

// Define a Course structure

struct Course {

string courseNumber;

string courseName;

vector<string> prerequisites;

};

// Function to sort and print courses stored in a hash table

void printSortedCoursesHashTable(unordered\_map<string, Course> courseMap) {

if (courseMap.empty()) {

print "No courses available. Load data first."

return

}

// Extract course numbers and sort them

vector<string> sortedKeys

for each key in courseMap:

add key to sortedKeys

sort(sortedKeys.begin(), sortedKeys.end())

// Print sorted courses

print "Courses in sorted order:"

for each key in sortedKeys:

print key + " - " + courseMap[key].courseName

}

**Using Search Binary Tree**

// Define a TreeNode structure for the BST

struct TreeNode {

Course course

TreeNode \*left

TreeNode \*right

};

// Function to insert a course into the BST

TreeNode\* insertBST(TreeNode\* root, Course course) {

if (root == null) {

create new TreeNode with course

return new node

}

if (course.courseNumber < root.course.courseNumber) {

root.left = insertBST(root.left, course)

} else {

root.right = insertBST(root.right, course)

}

return root

}

// Function to traverse and print courses in sorted order

void inOrderTraversal(TreeNode\* root) {

if (root == null) {

return

}

inOrderTraversal(root.left)

print root.course.courseNumber + " - " + root.course.courseName

inOrderTraversal(root.right)

}