# CS 300 Pseudocode Document

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

// ABC University Course Management System Pseudocode

// Structure to represent a course

struct Course {

string code;

string title;

vector<string> prerequisites;

};

// Function to load course data from a file into a hashtable

void loadCourseData(unordered\_map<string, Course>& courseMap, string filename) {

OPEN ifstream file(filename);

IF (file.is\_open()) {

DEFINEstring line;

WHILE (getline(file, line)) {

INITIALIZE string code, title, prerequisitesString;

// Parse the line to extract course code, title, and prerequisites

// ...

IF (/\* line is correctly formatted \*/) {

// Check if all prerequisites already exist in the courseMap

DEFINE bool prerequisitesExist = true;

FOR (const auto& prerequisite : prerequisites) {

IF (courseMap.find(prerequisite) == courseMap.end()) {

SET prerequisitesExist = false;

break;

}

}

IF (prerequisitesExist) {

// Create a new course object

INITIALIZE Course newCourse;

// Set the course code and title

// ...

// Parse the prerequisites string and add them to the course's prerequisites vector

// ...

// Add the course object to the courseMap with the course code as key

SET courseMap[code] = newCourse;

} ELSE {

// Display an error message indicating missing prerequisites

// ...

}

} ELSE {

// Display an error message indicating file format error

// ...

}

}

CALL file.close();

} ELSE {

// Display an error message indicating file open error

// ...

}

}

// Function to search for a course and print its information and prerequisites

void searchCourse(const unordered\_map<string, Course>& courseMap, string courseCode) {

if courseMap contains the 'courseCode':

course = courseMap[courseCode]

print the course code, title, and prerequisites

else:

print an error message indicating course not found

}

// Function to print a sample course listing

void printSampleListing(const unordered\_map<string, Course>& courseMap) {

for each course in courseMap:

print the course code and title

if the course has prerequisites:

for each prerequisite in the course's prerequisites:

print the prerequisite course code

}

// Function to count the number of prerequisites for a course

int numPrerequisiteCourses(const unordered\_map<string, Course>& courseMap, const Course& course) {

int totalPrerequisites = course.prerequisites.size()

for each prerequisite in course.prerequisites:

if courseMap contains the prerequisite:

prerequisiteCourse = courseMap[prerequisite]

totalPrerequisites += numPrerequisiteCourses(courseMap, prerequisiteCourse)

return totalPrerequisites

}

/ Function to print the course information and prerequisites

void printCourseInformation(const unordered\_map<string, Course>& courseMap, const string& courseNumber) {

if courseMap contains the 'courseNumber':

course = courseMap[courseNumber]

print the course code, title, and prerequisites

for each prerequisite in the course's prerequisites:

if courseMap contains the prerequisite:

prerequisiteCourse = courseMap[prerequisite]

print the prerequisite course information

else:

print an error message indicating course not found

}

// Function to print the course list in alphanumeric order for the Computer Science department

void printCourseList(const unordered\_map<string, Course>& courseMap) {

// Create a vector to store the course codes of Computer Science department

DEFINE vector<string> csCourses;

// Iterate over the courseMap to find the courses in the Computer Science department

FOR (const auto& pair : courseMap) {

DEFINE const Course& course = pair.second;

// Check if the course code starts with "CS" to indicate it belongs to the Computer Science department

IF (course.code.substr(0, 2) == "CS") {

CALL csCourses.push\_back(course.code);

}

}

// Sort the course codes in alphanumeric order

CALL sort(csCourses.begin(), csCourses.end());

// Print the course codes and titles in alphanumeric order

FOR (const string& courseCode : csCourses) {

DEFINE const Course& course = courseMap.at(courseCode);

OUTPUT << "Course Code: " << course.code << endl;

OUTPUT << "Course Title: " << course.title << endl;

OUTPUT << endl;

}

}

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

void printCourse(const unordered\_map<string, Course>& courseMap, const string& courseCode) {

IF (courseMap.count(courseCode) > 0) {

DEFINE const Course& course = courseMap.at(courseCode);

OUTPUT << "Course Code: " << course.code << endl;

OUTPUT << "Course Title: " << course.title << endl;

OUTPUT << "Prerequisites: ";

IF (course.prerequisites.empty()) {

OUTPUT << "None";

} ELSE {

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

OUTPUT << prerequisite << " ";

}

}

OUTPUT << endl;

} ELSE {

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

}

}

// Main function

int main() {

declare an unordered\_map of Course objects: courseMap

declare a string variable: filename

declare a string variable: courseCode

CALL the filename of the course data file

read the filename

PROMPT user to INPUT

1: LOAD FILE

2: PRINT SORTED COURSE LIST

3: PRINT COURSE

9: EXIT PROGRAM

SWITCH (USERINPUT):

CASE 1:

CALL loadCourseData(courseMap, filename)

CASE 2:

prompt the user to enter a course code to search

read the course code from the user

searchCourse(courseMap, courseCode)

OUTPUT “Printing the course list for the Computer Science department:"

printCourseListAlphaNumeric(courseMap);

CASE 3:

CALL printSampleListing(courses)

DEFINE string courseCodeToPrint;

OUTPUT << "Enter a course code to print its information: ";

INPUT >> courseCodeToPrint;

CALL printCourse(courseMap, courseCodeToPrint);

CASE 9:

break;

return 0;

}

## Example Runtime Analysis

1. Vector:
   * Runtime:
     + Accessing elements by index: O(1)
     + Searching for a course by code: O(n)
     + Inserting a new course: O(1) amortized, but O(n) if the vector needs to be resized
     + Memory:
       1. Memory usage for storing the courses: O(n)
2. Hashtable (unordered\_map):
   * Runtime:
     + Accessing elements by course code: O(1) on average
     + Searching for a course by code: O(1) on average
     + Inserting a new course: O(1) on average
     + Memory:
       1. Memory usage for storing the courses: O(n)
3. Binary Search Tree (BST):
   * Runtime:
     + Accessing elements by course code: O(log n) on average, but O(n) in the worst case if the tree is highly unbalanced
     + Searching for a course by code: O(log n) on average, but O(n) in the worst case
     + Inserting a new course: O(log n) on average, but O(n) in the worst case if the tree is highly unbalanced
     + Memory:
       1. Memory usage for storing the courses: O(n)

In terms of runtime, the hashtable (unordered\_map) offers constant-time (O(1)) access and insertion, making it the most efficient choice for searching and inserting courses by their code. The vector has a linear search time, making it less efficient for these operations. The BST provides logarithmic-time (O(log n)) search and insertion on average, but it can become inefficient (O(n)) if the tree becomes unbalanced.

In terms of memory usage, both the vector and hashtable have similar characteristics, requiring O(n) memory to store the courses. The BST may require slightly more memory due to the additional pointers needed for tree structure. This is not necessarily a case against the Binary Search Tree overall, as it seems to be the most well-rounded of the three data structure options under consideration.

Based on the requirements of the C++ file, the hashtable (unordered\_map) data structure seems to be the most suitable choice, as it provides efficient lookup and insertion times, and its memory usage is comparable to the vector. Vector does match hashtable in certain operations, but overall, it was not desirable when considering scalability and other factors. While the Binary Search Tree does have the best balance of the project’s overall requirements, this balance is negated by the speed with which the hashtable data structure performs critical, otherwise resource-heavy operations for the project.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | m | m |
| **for each prerequisite of the course** | 1 | m | m |
| **print the prerequisite course information** | 1 | m | O(m) |
| **Total Cost** | | | 3n+3m+2 |
| **Runtime** | | | O(n+m) |