# CS 300 Pseudocode Document

## Example Function Signatures

Below is an example of a function signature that you can use as a guide to help address the program requirements using each data structure for the milestones. The pseudocode for finding and printing course information is also given below and depicted in bold to help you get started. The provided pseudocode is for a vector data structure, so you may use this pseudocode in your first milestone as is. The hash table and tree structures are also shown below. But these structures are left for you to do in future milestones.

//Vector - Milestone 1

**File Input**  
1. Define a function to load courses from a file:  
 - Function name: loadCourses  
 - Parameters: filename (string), courses (vector of Course objects)  
2. Open the file with the given filename.  
3. If the file cannot be opened, print an error message and exit the function.  
4. For each line in the file:  
 - Split the line into tokens using commas as separators.  
 - If there are less than two tokens, print an error message and skip to the next line.  
 - Create a new Course object.  
 - Set the courseNumber and name of the Course object using the first two tokens.  
 - For any remaining tokens, add them to the prerequisites of the Course object.  
 - Add the Course object to the courses vector.  
5. Close the file.  
6. For each Course object in the courses vector:  
 - For each prerequisite in the Course object:  
 - Check if the prerequisite exists as a course in the courses vector.  
 - If the prerequisite does not exist, print an error message.

**Course Object Pseudocode**  
1. Define a Course structure with the following fields:  
 - courseNumber (string)  
 - name (string)  
 - prerequisites (vector of strings)

**Print Course Information Pseudocode**  
1. Define a function to search for a course and print its information:  
 - Function name: searchCourse  
 - Parameters: courses (vector of Course objects), courseNumber (string)  
2. For each Course object in the courses vector:  
 - If the courseNumber of the Course object matches the given courseNumber:  
 - Print the course number and name.  
 - If the course has prerequisites:  
 - Print "Prerequisites:" and list each prerequisite.  
 - If the course has no prerequisites, print "No prerequisites."  
 - Exit the function.  
3. If no course with the given courseNumber is found, print "Course not found."

//Hash Table - Milestone 2

**File Input**

// Function to load data from a file into a hash table  
function loadCourses(fileName):  
 // Open the file  
 file = open(fileName)  
   
 // Initialize an empty hash table  
 courseTable = HashTable()  
   
 // Read each line from the file  
 for line in file:  
 // Split the line by commas to separate course data  
 courseData = split(line, ',')  
   
 // Check if the line has at least two parameters  
 if length(courseData) < 2:  
 print("Error: Each line must have at least a course number and title.")  
 continue  
   
 // Extract course number and title  
 courseNumber = courseData[0]  
 courseTitle = courseData[1]  
   
 // Extract prerequisites  
 prerequisites = []  
 for i in range(2, length(courseData)):  
 prerequisites.append(courseData[i])  
   
 // Create a course object  
 course = Course(courseNumber, courseTitle, prerequisites)  
   
 // Add the course to the hash table  
 courseTable.insert(courseNumber, course)  
   
 // Validate prerequisites  
 for course in courseTable:  
 for prereq in course.prerequisites:  
 if not courseTable.contains(prereq):  
 print("Error: Prerequisite " + prereq + " for course " + course.courseNumber + " does not exist.")  
   
 // Close the file  
 file.close()  
   
 return courseTable

**Course Object**  
// Course class to hold course data  
class Course:  
 function \_\_init\_\_(courseNumber, courseTitle, prerequisites):  
 this.courseNumber = courseNumber  
 this.courseTitle = courseTitle  
 this.prerequisites = prerequisites

**Print Course Information**  
// Function to print course information  
function printCourseInformation(courseTable, courseNumber):  
 // Search for the course in the hash table  
 course = courseTable.get(courseNumber)  
   
 if course == null:  
 print("Course not found.")  
 return  
   
 // Print course information  
 print("Course Number: " + course.courseNumber)  
 print("Course Title: " + course.courseTitle)  
   
 // Print prerequisites  
 if length(course.prerequisites) == 0:  
 print("Prerequisites: None")  
 else:  
 print("Prerequisites: ")  
 for prereq in course.prerequisites:  
 prereqCourse = courseTable.get(prereq)  
 if prereqCourse != null:  
 print(prereqCourse.courseNumber + ": " + prereqCourse.courseTitle)  
 else:  
 print(prereq + ": Course information not available.")

//Tree Data Structure - Milestone 3

**Open and Read File**

Function OpenAndReadFile(filePath):

Open file at filePath

If file is not open:

Print "Error: Unable to open file"

Return

For each line in file:

Split line by commas into courseData

Call ValidateCourseData(courseData)

If courseData is valid:

Call CreateAndStoreCourse(courseData)

Else:

Print "Error: Invalid course data format"

Close file

**Validate Course Data**

Function ValidateCourseData(courseData):

If length of courseData < 2:

Return False

For each prerequisite in courseData from index 2 to end:

If prerequisite is not in courseNumbers:

Return False

Return True

**Create and Store Course**

Function CreateAndStoreCourse(courseData):

Create Course object with courseData[0] as courseNumber, courseData[1] as name, and remaining as prerequisites

Call InsertCourseInTree(Course)

Add courseNumber to courseNumbers

**Insert Course in Tree**

Function InsertCourseInTree(course):

If treeRoot is null:

Set treeRoot to new TreeNode(course)

Else:

Call InsertNode(treeRoot, course)

**Insert Node**

Function InsertNode(node, course):

If course.courseNumber < node.course.courseNumber:

If node.left is null:

Set node.left to new TreeNode(course)

Else:

Call InsertNode(node.left, course)

Else:

If node.right is null:

Set node.right to new TreeNode(course)

Else:

Call InsertNode(node.right, course)

**Print Course Information**

Function PrintCourseInfo(courseNumber):

Call SearchCourse(treeRoot, courseNumber)

Function SearchCourse(node, courseNumber):

If node is null:

Print "Course not found"

Return

If courseNumber == node.course.courseNumber:

Print "Course Number: " + node.course.courseNumber

Print "Course Name: " + node.course.name

Print "Prerequisites: " + node.course.prerequisites

Else If courseNumber < node.course.courseNumber:

Call SearchCourse(node.left, courseNumber)

Else:

Call SearchCourse(node.right, courseNumber)

**Main Function**

Function Main():

Set courseNumbers to empty list

Set treeRoot to null

Call OpenAndReadFile("course\_data.txt")

Call PrintCourseInfo("CSCI200")

//Completed Pseudocode – Final Project

**Sorting and Printing Courses in Alphanumeric Order**

Function sortAndPrintCoursesVector(courses):

Sort courses by course number

For each Course object in courses:

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.name

**Menu Implementation**

// Main function to display the menu and handle user input

Function Main():

coursesVector = []

courseTable = HashTable()

treeRoot = null

While True:

Print "Menu:"

Print "1. Load Courses"

Print "2. Print All Courses (Alphanumeric Order)"

Print "3. Print Course Information"

Print "9. Exit"

choice = Get user input

If choice == 1:

// Load courses based on user preference for data structure

Print "Choose data structure: 1. Vector 2. Hash Table 3. Tree"

dataChoice = Get user input

If dataChoice == 1:

Call loadCoursesVector("filename.txt", coursesVector)

Else If dataChoice == 2:

Call loadCoursesHashTable("filename.txt", courseTable)

Else If dataChoice == 3:

Call loadCoursesTree("filename.txt", treeRoot)

Else If choice == 2:

// Print all courses

Print "Choose data structure: 1. Vector 2. Hash Table 3. Tree"

dataChoice = Get user input

If dataChoice == 1:

Call sortAndPrintCoursesVector(coursesVector)

Else If dataChoice == 2:

Call sortAndPrintCoursesHashTable(courseTable)

Else If dataChoice == 3:

Call sortAndPrintCoursesTree(treeRoot)

Else If choice == 3:

// Print course information

courseNumber = Get user input ("Enter course number:")

Print "Choose data structure: 1. Vector 2. Hash Table 3. Tree"

dataChoice = Get user input

If dataChoice == 1:

Call searchCourseVector(coursesVector, courseNumber)

Else If dataChoice == 2:

Call searchCourseHashTable(courseTable, courseNumber)

Else If dataChoice == 3:

Call searchCourseTree(treeRoot, courseNumber)

Else If choice == 9:

Print "Exiting program."

Break

## Example Runtime Analysis

When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This particular example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Don’t forget to include your charts in Project One. You will submit Project One in Module Six.

**Vector Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

**Tree Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **traverse the tree to find courseNumber** | 1 | log n (avg), n (worst case) | log n (avg), n (worst case) |
| **if course found, for each prerequisite** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | **2n + log n** (worst case), **log n** (average case) |
| **Runtime** | | | **O(n)** (worst case), **O(log n)** (average) |

**Hash Table Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **compute the hash for courseNumber** | 1 | 1 | 1 |
| **access the course in the hash table** | 1 | 1 | 1 |
| **if course exists, for each prerequisite** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | **O(n)** (worst case), **O(1)** (average) |

**Justification**

Vector:

Advantages: Simple to implement and understand. Suitable for small datasets where linear search is acceptable.

Disadvantages: Searching is slow (O(n)), and sorting is required for each print operation.

Hash Table:

Advantages: Fast search and insertion on average (O(1)). Suitable for large datasets.

Disadvantages: Sorting is still O(n log n). Worst-case search time can degrade to O(n) if there are many collisions.

Binary Search Tree:

Advantages: Efficient for searching (O(log n) on average) and printing courses in sorted order (O(n)).

Disadvantages: Insertion and search times can degrade to O(n) in the worst case if the tree becomes unbalanced.

**Recommendation**

Based on the runtime analysis:

Hash Table is recommended if fast search times are crucial and the dataset is large, despite the potential for collisions.

Binary Search Tree is recommended if the data needs to be frequently printed in sorted order and balancing the tree can be ensured.