# 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

void searchCourse(Vector<Course> courses, String courseNumber) {

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

//Hash Table - Milestone 2

// Function to search and print course information using a hash table

void searchCourse(HashTable<Course> courses, String courseNumber) {

// Check if the courseNumber exists in the hash table

if courses.containsKey(courseNumber):

course = courses.get(courseNumber)

// Print the course information

print "Course Number:", course.courseNumber

print "Course Name:", course.name

if course.prerequisites is not empty:

print "Prerequisites:"

// Print all prerequisites

for each prerequisite in course.prerequisites:

print "- ", prerequisite

else:

print "Error: Course not found"

}

//Binary Search Tree – Milestone 3

// Function to search and print course information using a binary search tree

void searchCourse(Tree<Course> courses, String courseNumber) {

// Search for the course in the binary search tree

Course foundCourse = courses.search(courseNumber)

// If the course is found, print its information

if foundCourse is not null:

print "Course Number:", foundCourse.courseNumber

print "Course Name:", foundCourse.name

// Print prerequisites, if any

if foundCourse.prerequisites is not empty:

print "Prerequisites:"

for each prerequisite in foundCourse.prerequisites:

print "- ", prerequisite

else:

// Print an error message if the course is not found

print "Error: Course not found"

}

## 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.

| **c** | **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) |

| **c** | **Line Cost** | **# Times Executes** | | **Total Cost** | | |
| --- | --- | --- | --- | --- | --- | --- |
| Course foundCourse = courses.search(courseNumber) | log n | 1 | | log n | | |
| if foundCourse is not null: | 1 | 1 | | 1 | | |
| print "Course Number:", foundCourse.courseNumber | 1 | 1 | | 1 | | |
| print "Course Name:", foundCourse.name | 1 | 1 | | 1 | | |
| if foundCourse.prerequisites is not empty: | 1 | 1 | | 1 | | |
| print "Prerequisites:" | 1 | 1 (if prerequisites not empty) | | 1 | | |
| for each prerequisite in foundCourse.prerequisites: | 1 | m (m = number of prerequisites) | | m | | |
| print "- ", prerequisite | 1 | m | | m | | |
| print "Error: Course not found" | 1 | 1 (if course is not found) | | 1 | | |
| Course foundCourse = courses.search(courseNumber) | log n | 1 | | log n | | |
| Total Cost (if course is found and has prerequisites) | | | log n + 5 + 2m | |
| Total Cost (if course is not found) | | | log n + 1 | |
| Runtime (if course is found and has prerequisites) | | | O(log n + m) | |
| Runtime (if course is not found) | | | O(log n) | |
| **c** | **Line Cost** | **# Times Executes** | | **Total Cost** | | |
| if courses.containsKey(courseNumber): | 1 | 1 | | 1 | | |
| course = courses.get(courseNumber) | 1 | 1 (if condition is true) | | 1 | | |
| print "Course Number:", course.courseNumber | 1 | 1 (if condition is true) | | 1 | | |
| print "Course Name:", course.name | 1 | 1 (if condition is true) | | 1 | | |
| if course.prerequisites is not empty: | 1 | 1 (if condition is true) | | 1 | | |
| print "Prerequisites:" | 1 | 1 (if condition is true and prerequisites not empty) | | 1 | | |
| for each prerequisite in course.prerequisites: | 1 | m (m = number of prerequisites) | | m | | |
| print "- ", prerequisite | 1 | m | | m | | |
| print "Error: Course not found" | 1 | 1 (if condition is false) | | 1 | | |
| if courses.containsKey(courseNumber): | 1 | 1 | | 1 | | |
| Total Cost (if course is found and has prerequisites) | | | | 5+2m | |
| |  | | --- | | **Total Cost (if course is not found)** |  |  | | --- | |  | | | | | 2 | |
| Runtime (if course is found and has prerequisites) | | | | O(m) | |
| |  | | --- | | **Runtime (if course is not found)** |  |  | | --- | |  | | | | | O(1) | |
| **File Input** |  |  | |  | | |
| **c** | **Line Cost** | **# Times Executes** | | **Total Cost** | | |
| file = open(filename, "r") | 1 | 1 | | 1 | | |
| validCourseNumbers = Set() | 1 | 1 | | 1 | | |
| courses = Vector() | 1 | 1 | | 1 | | |
| for each line in file: | 1 | n | | n | | |
| splitLine = line.split(",") | 1 | n | | n | | |
| if length of splitLine < 2: | 1 | n | | n | | |
| print "Error: Insufficient data on line", line | 1 | n (worst-case) | | n | | |
| courseNumber = splitLine[0].trim() | 1 | n | | n | | |
| name = splitLine[1].trim() | 1 | n | | n | | |
| if length of splitLine > 2: | 1 | n | | n | | |
| prerequisites = splitLine[2].split(";") | 1 | n | | n | | |
| prerequisites = [] | 1 | n | | n | | |
| isValid = True | 1 | n | | n | | |
| for each prerequisite in prerequisites: | 1 | m\*n (m=avg. # of prerequisites) | | m\*n | | |
| prerequisite = prerequisite.trim() | 1 | m\*n | | m\*n | | |
| if prerequisite is not in validCourseNumbers: | 1 | m\*n | | m\*n | | |
| print "Error: Prerequisite", prerequisite, "not found | 1 | m\*n (worst-case) | | m\*n | | |
| isValid = False | 1 | m\*n | | m\*n | | |
| if not isValid: | 1 | n | | n | | |
| course = createCourseObject(courseNumber, name, prerequisites) | 1 | n | | n | | |
| courses.add(course) | 1 | n | | n | | |
| validCourseNumbers.add(courseNumber) | 1 | n | | n | | |
| file.close() | 1 | 1 | | 1 | | |
| print "Error: Unable to open or read file" | 1 | 1 (worst-case) | | 1 | | |
| file = open(filename, "r") | 1 | 1 | | 1 | | |
| validCourseNumbers = Set() | 1 | 1 | | 1 | | |
| courses = Vector() | 1 | 1 | | 1 | | |
| for each line in file: | 1 | n | | n | | |
| splitLine = line.split(",") | 1 | n | | n | | |
| if length of splitLine < 2: | 1 | n | | n | | |
| **Total Cost** | | | | 7n + 5m\*n + 3 | | |
| **Runtime** | | | | O(n+m\*n) | | |
| **Course Object** |  |  | |  | | |
| **c** | **Line Cost** | **# Times Executes** | | **Total Cost** | | |
| course = Course() | 1 | 1 | | 1 | | |
| course.courseNumber = courseNumber | 1 | 1 | | 1 | | |
| course.name = name | 1 | 1 | | 1 | | |
| course.prerequisites = prerequisites | 1 | 1 | | 1 | | |
| return course | 1 | 1 | | 1 | | |
| **Total Cost** | | | | 5 | | |
| **Runtime** | | | | O(1) | | |

| **Loading Data into a Hash Table** |  |  |  |
| --- | --- | --- | --- |
| **c** | **Line Cost** | **# Times Executes** | **Total Cost** |
| file = open(filename, "r") | 1 | 1 | 1 |
| coursesHashTable = HashTable() | 1 | 1 | 1 |
| validCourseNumbers = Set() | 1 | 1 | 1 |
| for each line in the file: | 1 | n | n |
| splitLine = line.split(",") | 1 | n | n |
| if the length of splitLine < 2: | 1 | n | n |
| print "Error: Insufficient data on line" | 1 | n (worst-case) | n |
| courseNumber = splitLine[0].trim() | 1 | n | n |
| validCourseNumbers.add(courseNumber) | 1 | n | n |
| file.seek(0) | 1 | 1 | 1 |
| for each line in the file: | 1 | n | n |
| splitLine = line.split(",") | 1 | n | n |
| if the length of splitLine < 2: | 1 | n | n |
| print "Error: Insufficient data on line" | 1 | n (worst-case) | n |
| courseNumber = splitLine[0].trim() | 1 | n | n |
| name = splitLine[1].trim() | 1 | n | n |
| if the length of splitLine > 2: | 1 | n | n |
| prerequisites = splitLine[2].split(";") | 1 | n | n |
| prerequisites = [] | 1 | n | n |
| for each prerequisite in prerequisites: | 1 | m\*n (m = avg. # of prerequisites) | m\*n |
| prerequisite = prerequisite.trim() | 1 | m\*n | m\*n |
| if prerequisite is not in validCourseNumbers: | 1 | m\*n | m\*n |
| print "Error: Prerequisite", prerequisite, "not found | 1 | m\*n (worst-case) | m\*n |
| break | 1 | m\*n | m\*n |
| course = createCourseObject(courseNumber, name, prerequisites) | 1 | n | n |
| coursesHashTable.put(courseNumber, course) | 1 | n | n |
| file.close() | 1 | 1 | 1 |
| print "Error: Unable to open or read file" | 1 | 1 (worst-case) | 1 |
| **Total Cost** | | | 12n + 5m\*n + 5 |
| **Runtime** | | | O(n + m\*n) |
| **Load File into Binary Search Tree** |  |  |  |
| **c** | **Line Cost** | **# Times Executes** | **Total Cost** |
| file = open(filename, "r") | 1 | 1 | 1 |
| validCourseNumbers = Set() | 1 | 1 | 1 |
| for each line in file: | 1 | n | n |
| splitLine = line.split(",") | 1 | n | n |
| if length of splitLine < 2: | 1 | n | n |
| print "Error: Insufficient data on line" | 1 | n (worst-case) | n |
| courseNumber = splitLine[0].trim() | 1 | n | n |
| validCourseNumbers.add(courseNumber) | 1 | n | n |
| file.seek(0) | 1 | 1 | 1 |
| for each line in file: | 1 | n | n |
| splitLine = line.split(",") | 1 | n | n |
| if length of splitLine < 2: | 1 | n | n |
| print "Error: Insufficient data on line" | 1 | n (worst-case) | n |
| courseNumber = splitLine[0].trim() | 1 | n | n |
| name = splitLine[1].trim() | 1 | n | n |
| if length of splitLine > 2: | 1 | n | n |
| prerequisites = splitLine[2].split(";") | 1 | n | n |
| prerequisites = [] | 1 | n | n |
| isValid = True | 1 | n | n |
| for each prerequisite in prerequisites: | 1 | m\*n (m = avg. # of prerequisites) | m\*n |
| prerequisite = prerequisite.trim() | 1 | m\*n | m\*n |
| if prerequisite is not in validCourseNumbers: | 1 | m\*n | m\*n |
| print "Error: Prerequisite", prerequisite, "not found | 1 | m\*n (worst-case) | m\*n |
| isValid = False | 1 | m\*n | m\*n |
| if not isValid: | 1 | n | n |
| continue | 1 | n | n |
| course = createCourseObject(courseNumber, name, prerequisites) | 1 | n | n |
| bst.insert(course) | log n | n | n\*log n |
| file.close() | 1 | 1 | 1 |
| print "Error: Unable to open or read file" | 1 | 1 (worst-case) | 1 |
| **Total Cost** | | | 13n + 5mn + nlog n + 4 |
| **Runtime** | | | O(n log n + m\*n) |

**Evaluation**

The vector data structure offers many advantages in the simplicity of its use and how intuitive it is to use. It provides direct access through the use of index making it simple to iterate over elements. This offers efficiency when operations require accessing elements in sequence. On the other hand, inherently a search for an element will have a time complexity of O(n), which could be inefficient for large datasets. Adding elements into the middle or removing elements requires shifting of multiple elements. This leads to inefficiency for dynamic operations that facilitate a high frequency of insertions and deletions. From the Big O analysis for loading course data, we can see this has a total cost of 11n+3m+2 with a runtime of O(n+m).  
  
The hash table provides many advantages due to its dynamic updatability. It has an average-case time complexity of O(1) for insertions, deletions, and searches. This makes it efficient for searching elements according to their key. In addition, it provides dynamic resizing to automatically handle resizing of the table. In the worst-case time complexity can result in O(n) if there are a high frequency of collisions. Proper use of a quality hash function is required in order to minimize collisions, and also holds elements in no defined order. The total cost for this function was 12n+6m+2 with a runtime of O(n+m).

The binary search tree has several advantages as well due to the nodes it employs. This helps to provide an average-case O(log n) time complexity for updates and lookups of elements. These elements are stored within a sorted order providing useful and efficient range queries to be performed. This helps to afford its ability to handle dynamic operations in an efficient manner. Unfortunately, this results in a performance decrease to O(n) if an unbalance occurs in the tree. This will facilitate more complex implementation compared to vectors and hash tables to confer these benefits. This practice of balancing the tree will introduce additional overhead in performance and time investment to correctly and efficiently implement.   
  
When considering the advantages and disadvantages of each data structure a recommendation can be made for which structure best suits the purposes of this project. With the Big O analysis in mind the Hash Table would be a great recommendation for the challenges presented by this project. This is due to the ability to quickly and efficiently search, wherein the base average-case of time complexity is O(1). This is crucial in providing quick data retrieval of the course information. Due to the nature of course schedules changing semester to semester and as students drop or add courses early in the semester the importance of efficiently managing insertions and deletions must be stressed. Herein, the hash table again provides O(1) average-case time complexity for insertions and deletions. This makes it very suitable for dynamic data wherein removals, updates, and insertions are common. Hash table implementation is practically straightforward compared to self-balancing binary search trees. This can facilitate more efficient coding as well as simplifying maintenance. The consistency in performance of the hash table is one of its benefits for such a large data set of such a dynamic nature. This is kept in mind despite the worst-case performance possibly degrading to O(n) due to collisions. This is minimized through implementation of a sound hash function and providing proper sizing resulting in practice routine efficient performance of the hash table. The Big O analysis effectively demonstrated how the hash table had the lowest complexity for search operations. While a tree would offer sorted order and balanced search times, the added complexity and possibilities for decreased performance in unbalanced scenarios make it less suitable for the data featured in this project. Vectors despite being simple are simply not efficiently enough for the necessary search operations due to the sequential nature of its search operations. With efficiency and simplicity in mind this makes the hash table the most balanced approach for a data structure selection.

**File Input Pseudocode:**

//Function to load course data from a file

function loadCourseDataFromFile(filename):

try:

// Open the file in read mode

file = open(filename, "r")

// Initialize a set to keep track of valid course numbers to ensure prerequisite validation

validCourseNumbers = Set()

// Initialize a vector to store course objects

courses = Vector()

// Iterate through each line in the file

for each line in file:

// Split the line by commas to separate course number, name, and prerequisites

splitLine = line.split(",")

// Check if the line has at least course number and name

if length of splitLine < 2:

// Print error message for insufficient data and skip to the next line

print "Error: Insufficient data on line", line

continue

// Extract the course number and trim any surrounding whitespace

courseNumber = splitLine[0].trim()

// Extract the course name and trim any surrounding whitespace

name = splitLine[1].trim()

// Check if there are prerequisites listed and extract them

if length of splitLine > 2:

// Split prerequisites by semicolon and trim whitespace

prerequisites = splitLine[2].split(";")

else:

// If no prerequisites, initialize an empty list

prerequisites = []

// Flag to check if all prerequisites are valid

isValid = True

// Validate each prerequisite

for each prerequisite in prerequisites:

// Trim any surrounding whitespace from the prerequisite

prerequisite = prerequisite.trim()

// Check if the prerequisite is in the set of valid course numbers

if prerequisite is not in validCourseNumbers:

// Print error message for invalid prerequisite and set isValid flag to False

print "Error: Prerequisite", prerequisite, "not found for course", courseNumber

isValid = False

break

// If any prerequisite is invalid, skip adding this course

if not isValid:

continue

// Create a course object using the extracted data

course = createCourseObject(courseNumber, name, prerequisites)

// Add the created course object to the courses vector

courses.add(course)

// Add the course number to the set of valid course numbers for future validation

validCourseNumbers.add(courseNumber)

// Close the file after processing all lines

file.close()

except IOError:

// Print error message if the file cannot be opened or read

print "Error: Unable to open or read file"

// Return the vector containing all valid course objects

return courses

**Course Object Pseudocode:**

// Course structure definition

struct Course:

attributes:

courseNumber // Number/code of the course, e.g., CS101

name // Name of the course, e.g., Introduction to Computer Science

prerequisites (list) // List of prerequisite courses required for this course

// Function to create a Course object

function createCourseObject(courseNumber, name, prerequisites):

course = Course() // Create a new instance of the Course struct/object

course.courseNumber = courseNumber // Assign the course number to the object

course.name = name // Assign the course name to the object

course.prerequisites = prerequisites // Assign the list of prerequisites to the object

return course // Return the created Course object

**Print Course Information Pseudocode:**

// Function to print course information

function printCourseInformation(courses, courseNumber):

found = false // Flag to track if the course with given courseNumber is found

for each course in courses: // Iterate through each course in the list of courses

if course.courseNumber equals courseNumber: // Check if the current course's number matches the requested courseNumber

print "Course Number:", course.courseNumber // Print the course number

print "Course Name:", course.name // Print the course name

if course.prerequisites is not empty: // Check if there are any prerequisites for this course

print "Prerequisites:" // Print a header for prerequisites

for each prerequisite in course.prerequisites: // Iterate through each prerequisite

print "- ", prerequisite // Print each prerequisite

found = true // Set found flag to true since the course was found

break // Exit the loop since we found the course we were looking for

if not found: // If the course was not found (found flag remains false)

print "Error: Course not found" // Print an error message indicating the course was not found

**Loading Data into a Hash Table**

// Function to load course data from a file into a hash table

function loadCourseDataFromFileToHashTable(filename):

try:

// Open the file

file = open(filename, "r")

// Initialize a hash table to store courses

coursesHashTable = HashTable()

// Initialize a set to keep track of valid course numbers

validCourseNumbers = Set()

// Read all lines first to capture all course numbers

for each line in the file:

// Split the line by comma

splitLine = line.split(",")

// Validate data length (at least courseNumber and name)

if the length of splitLine < 2:

print "Error: Insufficient data on line"

continue

courseNumber = splitLine[0].trim()

validCourseNumbers.add(courseNumber)

// Reset file pointer to beginning

file.seek(0)

// Process each line to create course objects and store in hash table

for each line in the file:

// Split the line by comma

splitLine = line.split(",")

// Validate data length (at least courseNumber and name)

if the length of splitLine < 2:

print "Error: Insufficient data on line"

continue

courseNumber = splitLine[0].trim()

name = splitLine[1].trim()

// Split prerequisites by semicolon

if the length of splitLine > 2:

prerequisites = splitLine[2].split(";")

else:

prerequisites = []

// Validate each prerequisite

for each prerequisite in prerequisites:

prerequisite = prerequisite.trim()

if prerequisite is not in validCourseNumbers:

print "Error: Prerequisite", prerequisite, "not found for course", courseNumber

break

// Create and add the course object to hash table

course = createCourseObject(courseNumber, name, prerequisites)

coursesHashTable.put(courseNumber, course)

// Close the file

file.close()

except IOError:

print "Error: Unable to open or read file"

return coursesHashTable

**Printing Course Information from Hash Table**

// Function to search and print course information using a hash table

function searchCourse(coursesHashTable, courseNumber):

// Check if the courseNumber exists in the hash table

if coursesHashTable.containsKey(courseNumber):

course = coursesHashTable.get(courseNumber)

// Print the course information

print "Course Number:", course.courseNumber

print "Course Name:", course.name

if course.prerequisites is not empty:

print "Prerequisites:"

// Print all prerequisites

for each prerequisite in course.prerequisites:

print "- ", prerequisite

else:

print "Error: Course not found"

**Example of main function()**

function main():

// Load course data from file to hash table

coursesHashTable = loadCourseDataFromFileToHashTable("courses.txt")

// Example course number to search

courseNumber = "CS101"

// Print course information for the given course number

searchCourse(coursesHashTable, courseNumber)

main()

**Load File into Binary Search Tree**

// Function to load course data from a file into a binary search tree

function loadCourseDataFromFileToBST(filename, bst):

try:

// Open the file

file = open(filename, "r")

// Initialize a set to keep track of valid course numbers

validCourseNumbers = Set()

// Read all lines first to capture all course numbers

for each line in file:

// Split the line by comma

splitLine = line.split(",")

// Validate data length (at least courseNumber and name)

if length of splitLine < 2:

print "Error: Insufficient data on line"

continue

courseNumber = splitLine[0].trim()

validCourseNumbers.add(courseNumber)

// Reset file pointer to beginning

file.seek(0)

// Process each line to create course objects and store in BST

for each line in file:

// Split the line by comma

splitLine = line.split(",")

// Validate data length (at least courseNumber and name)

if length of splitLine < 2:

print "Error: Insufficient data on line"

continue

courseNumber = splitLine[0].trim()

name = splitLine[1].trim()

if length of splitLine > 2:

prerequisites = splitLine[2].split(";")

else:

prerequisites = []

// Validate each prerequisite

isValid = True

for each prerequisite in prerequisites:

prerequisite = prerequisite.trim()

if prerequisite is not in validCourseNumbers:

print "Error: Prerequisite", prerequisite, "not found for course", courseNumber

isValid = False

break

if not isValid:

continue

// Create and add the course object to BST

course = createCourseObject(courseNumber, name, prerequisites)

bst.insert(course)

// Close the file

file.close()

except IOError:

print "Error: Unable to open or read file"

**Print Course Information from Binary Search Tree**

// Function to print course information from a binary search tree

function printCourseInformationFromBST(bst, courseNumber):

// Search for the course in the BST

course = bst.search(courseNumber)

if course is not null:

// Print the course information

print "Course Number:", course.courseNumber

print "Course Name:", course.name

if course.prerequisites is not empty:

print "Prerequisites:"

// Print all prerequisites

for each prerequisite in course.prerequisites:

print "- ", prerequisite

else:

print "Error: Course not found"

**Main Function Example**

// Example of main function

function main():

// Initialize a binary search tree

bst = BinarySearchTree()

// Load course data from file into the binary search tree

loadCourseDataFromFileToBST("courses.txt", bst)

// Example course number to search

courseNumber = "CS101"

// Print course information for the given course number

printCourseInformationFromBST(bst, courseNumber)

main()  
  
  
**Module Six**  
  
// Function to display the menu options and handle user input

function displayMenu():

print "Welcome to the Course Management System!"

loop:

print "\nMenu Options:"

print "1. Load course data from file"

print "2. Print alphanumerically ordered list of Computer Science courses"

print "3. Print course information and prerequisites"

print "9. Exit program"

input = get user input ("Enter your choice: ")

if input == "1":

filename = get user input ("Enter the filename to load data from: ")

// Load course data into all data structures (vector, hash table, BST)

vectorCourses = loadCourseDataFromFileToVector(filename)

hashTableCourses = loadCourseDataFromFileToHashTable(filename)

binarySearchTreeCourses = loadCourseDataFromFileToBST(filename)

print "Course data loaded successfully!"

else if input == "2":

// Print alphanumerically ordered list of Computer Science courses

print "Alphanumerically ordered list of Computer Science courses:"

printAlphanumericallyOrderedCourses(vectorCourses)

else if input == "3":

courseNumber = get user input ("Enter the course number to print information: ")

// Print course information from vector data structure

printCourseInformationFromVector(vectorCourses, courseNumber)

// Print course information from hash table data structure

printCourseInformationFromHashTable(hashTableCourses, courseNumber)

// Print course information from binary search tree data structure

printCourseInformationFromBST(binarySearchTreeCourses, courseNumber)

else if input == "9":

print "Exiting program..."

break

else:

print "Invalid choice. Please enter a valid option."

// Function to print alphanumerically ordered list of Computer Science courses from vector data structure

function printAlphanumericallyOrderedCourses(vectorCourses):

// Sort courses by courseNumber (assuming courseNumber is a string)

sortedCourses = sort(vectorCourses) // Sorting logic depends on the programming language used

// Print sorted courses

for each course in sortedCourses:

if course.department == "Computer Science": // Assuming there's a department attribute

print course.courseNumber, "-", course.name

// Function to print course information from vector data structure

function printCourseInformationFromVector(vectorCourses, courseNumber):

found = false

for each course in vectorCourses:

if course.courseNumber == courseNumber:

print "Course Number:", course.courseNumber

print "Course Name:", course.name

if course.prerequisites is not empty:

print "Prerequisites:"

for each prerequisite in course.prerequisites:

print "- ", prerequisite

found = true

break

if not found:

print "Error: Course not found"

// Function to print course information from hash table data structure

function printCourseInformationFromHashTable(hashTableCourses, courseNumber):

if hashTableCourses.containsKey(courseNumber):

course = hashTableCourses.get(courseNumber)

print "Course Number:", course.courseNumber

print "Course Name:", course.name

if course.prerequisites is not empty:

print "Prerequisites:"

for each prerequisite in course.prerequisites:

print "- ", prerequisite

else:

print "Error: Course not found"

// Function to print course information from binary search tree data structure

function printCourseInformationFromBST(binarySearchTreeCourses, courseNumber):

course = binarySearchTreeCourses.search(courseNumber)

if course is not null:

print "Course Number:", course.courseNumber

print "Course Name:", course.name

if course.prerequisites is not empty:

print "Prerequisites:"

for each prerequisite in course.prerequisites:

print "- ", prerequisite

else:

print "Error: Course not found"

// Example of main function to start the program

function main():

// Initialize variables to hold data structures

vectorCourses = Vector<Course>()

hashTableCourses = HashTable<Course>()

binarySearchTreeCourses = BinarySearchTree<Course>()

// Call displayMenu to start the menu system

displayMenu()

// Start the main function

main()