# 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).

// Vector pseudocode

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

allCourses = empty Vector<Course>

// Opens/Closes the file and parses the Information

try:

file = open("CourseInformation.txt", "r")

for each line in file:

components = splitLine(line)

if length(components) >= 3:

courseNumber = components[0]

courseName = components[1]

prerequisitesData = components[2]

prerequisites = parsePrerequisites(prerequisitesData)

course = new Course(courseNumber, courseName, prerequisites)

allCourses.add(course)

else:

except FileError as e:

print("Error:", e)

file.close()

// Validate the input course 'c'

if not any(course.courseNumber equals c.courseNumber for each course in allCourses):

print("Error: Course", c.courseNumber, "does not exist.")

return 0

// Get the prerequisites of the input course 'c'

totalPrerequisites = c.prerequisites

// Iterate through each prerequisite and add its prerequisites to the list

for each prerequisite in c.prerequisites:

// Validate the existence of the prerequisite in the file

if not any(course.courseNumber equals prerequisite for each course in allCourses):

print("Error: Prerequisite", prerequisite, "for course", c.courseNumber, "does not exist.")

// Get the Course object for the prerequisite and add its prerequisites

totalPrerequisites.addAll(course.prerequisites for each course in allCourses if course.courseNumber equals prerequisite)

}

void printSampleSchedule(Vector<Course> courses) {

// Sort the course information by alphanumeric course number from lowest to highest

sortedCourses = sort(courses, by: courseNumber)

// Print the sorted list to a display

for each course in sortedCourses:

print("Course Number:", course.courseNumber)

print("Course Name:", course.name)

print("Prerequisites:")

for each prerequisite in course.prerequisites:

print("- ", prerequisite)

}

void printCourseInformation(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**

}

void menu(Vector<Course> courses) {

while true:

print("Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

choice = readInput()

if choice == 1:

numPrerequisiteCourses(courses)

print("Data loaded.")

else if choice == 2:

if courses is null:

print("Error: Data structure not loaded. Please load data first.")

else:

printSampleSchedule(courses)

else if choice == 3:

if courses is null:

print("Error: Data structure not loaded. Please load data first.")

else:

print("Enter course number:")

courseNumber = readInput()

printCourseInformation(courses, courseNumber)

else if choice == 4:

break

else:

print("Invalid choice. Please try again.")

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses, Course c) {

allCourses = empty Hashtable<String, Course>

// Opens and Closes the file and parses the Information

try:

file = open("CourseInformation.txt")

for each line in file:

components = splitLine(line)

if length(components) >= 2:

courseNumber = components[0]

courseName = components[1]

prerequisites = components[2:]

course = new Course(courseNumber, courseName, prerequisites)

allCourses.put(courseNumber, course)

file.close()

except FileError as e:

print("Error:", e)

// Validate the input course 'c'

if not courses.contains(c.courseNumber):

print("Error: Course", c.courseNumber, "does not exist.")

return 0

// Get the prerequisites of the input course 'c'

totalPrerequisites = c.prerequisites

// Iterate through each prerequisite and add its prerequisites to the list

for each prerequisite in c.prerequisites:

// Validate the existence of the prerequisite in the hashtable

if not allCourses.contains(prerequisite):

print("Error: Prerequisite", prerequisite, "for course", c.courseNumber, "does not exist.")

return 0

// Get the Course object for the prerequisite and add its prerequisites

prerequisiteCourse = allCourses.get(prerequisite)

totalPrerequisites.addAll(prerequisiteCourse.prerequisites)

// Return the total number of prerequisites

return length(totalPrerequisites)

}

void printSampleSchedule(Hashtable<Course> courses) {

// Get all course numbers from the hashtable

courseNumbers = courses.keys()

// Sort the course numbers in alphanumeric order

sortedCourseNumbers = sort(courseNumbers)

// Print the courses in alphanumeric order

for each courseNumber in sortedCourseNumbers:

course = courses.get(courseNumber)

print(course.courseNumber + " - " + course.title)

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

if courses.contains(courseNumber):

course = courses.get(courseNumber)

print("Course Number: " + course.courseNumber)

print("Title: " + course.title)

if length(course.prerequisites) > 0

print("Prerequisites:")

for prereq in course.prerequisites

print(prereq)

else:

print("Course not found.")

}

Void menu() {

courses = null

while true:

print("Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

choice = readInput()

if choice == 1:

numPrerequisiteCourses(courses)

print("Data loaded.")

else if choice == 2:

if courses is null:

print("Error: Data structure not loaded. Please load data first.")

else:

printSampleSchedule(courses)

else if choice == 3:

if courses is null:

print("Error: Data structure not loaded. Please load data first.")

else:

print("Enter course number:")

courseNumber = readInput()

printCourseInformation(courses, courseNumber)

else if choice == 4:

break

else:

print("Invalid choice. Please try again.")

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

Openfile("Course Information")

If the file cannot be opened:

Print(error “File could not be opened”)

Initialize an empty tree to store course objects

Allcources = Tree<Course.

For (each line in the file:)

Read the line.

Split the line into tokens;

//Make sures there is at least 2 parameters.

If (tokens < 2)

Print (error “line does not have enough parameters”).

Continue to the next line++

Extract the course number, title, and prerequisites from the tokens

if (prerequisites > tokens minus 2)

Print(error “missing prerequisites”).

Continue to the next line++

if (any prerequisite does not exist as a course in the file)

print(error “indicating the missing prerequisite”)

Continue to the next line.

Create a new Course object with the extracted course number, title, and prerequisites.

Add the Course object to the tree based on its course number.

File.close();

For each course in tree allcourses:

For each prerequisite of the course:

Check if the prerequisite exists as a course in the tree:

Print(error “the prerequisite does not exist”).

}

void printSampleSchedule(Tree<Course> courses) {

// Loop through each course in the tree and print its information

for each Course course in courses:

if node is not null:

// Visit the left subtree

inOrderTraversal(node.left)

// Print the current node

print(node.courseNumber + " - " + node.title)

// Visit the right subtree

inOrderTraversal(node.right)

}

void printCourseInformation(Tree<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**

}

Void menu() {

courses = null

while true:

print("Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course")

print("4. Exit")

choice = readInput()

if choice == 1:

numPrerequisiteCourses(courses)

print("Data loaded.")

else if choice == 2:

if courses is null:

print("Error: Data structure not loaded. Please load data first.")

else:

printSampleSchedule(courses)

else if choice == 3:

if courses is null:

print("Error: Data structure not loaded. Please load data first.")

else:

print("Enter course number:")

courseNumber = readInput()

printCourseInformation(courses, courseNumber)

else if choice == 4:

break

else:

print("Invalid choice. Please try again.")

}

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This 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.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 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) |

Vector

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Split line into components | 1 | n | n |
| Create Course object | 1 | n | n |
| Add Course object to allCourses vector | 1 | n | n |
| Validate input course 'c' | 1 | 1 | 1 |
| Get prerequisites of input course 'c' | 1 | 1 | 1 |
| Iterate through prerequisites | 1 | n | n |
| Validate existence of prerequisite | 1 | n | n |
| Get Course object for prerequisite | 1 | n | n |
| Add prerequisites to totalPrerequisites | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost**  **Runtime** |  |  | **8n + 4**  O(n) |

Hashtable

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Split line into components | 1 | n | n |
| Create Course object and add to hashtable | 1 | n | n |
| Validate input course 'c' | 1 | 1 | 1 |
| Get prerequisites of input course 'c' | 1 | 1 | 1 |
| Iterate through prerequisites | 1 | n | n |
| Validate existence of prerequisite | 1 | n | n |
| Get Course object for prerequisite | 1 | n | n |
| Add prerequisites to totalPrerequisites | 1 | n | n |
| Return total number of prerequisites | 1 | 1 | 1 |
| Close file | 1 | 1 | 1 |
| **Total Cost**  Runtime |  |  | 7n + 5  O(n) |

Tree

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Extract course info | 1 | n | n |
| Create Course object and add to tree | 1 | n | n |
| Check prerequisites | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost**  **Runtime** |  |  | **5n + 2**  O(n) |

**Explain the advantages and disadvantages of each structure in your evaluation.**

Vector:

Advantages:

Provides fast access to elements by index (O(1) time complexity).

Maintains the order of elements, which may be important for representing a sequence of courses.

Simple to use and understand, with straightforward operations for adding, accessing, and removing elements.

Disadvantages:

Inserting or removing elements from the middle of the vector can be slow (O(n) time complexity) as it requires shifting elements.

Resizing the vector can be costly if done frequently, as it may involve allocating new memory and copying elements.

Hash Table:

Advantages:

Provides fast insertion, deletion, and retrieval of elements on average (O(1) time complexity for each operation in the average case).

Ideal for quickly checking the existence of elements, such as checking if a course exists in the data structure.

Disadvantages:

May have higher memory overhead compared to other data structures.

Worst-case time complexity for operations can be O(n) if there are many collisions (i.e., many elements hash to the same value), although this is rare with a well-designed hash function.

Tree:

Advantages:

Provides efficient insertion, deletion, and retrieval of elements in a sorted order (O(log n) time complexity for each operation in a balanced tree).

Ensures that elements are stored in a specific order, which can be useful for maintaining prerequisite relationships between courses.

Disadvantages:

May require additional logic to balance the tree to maintain efficient performance, especially if the tree becomes unbalanced due to insertions or deletions.

Requires more memory compared to hash tables or vectors, especially for large datasets.

**make a recommendation for which data structure you will plan to use in your code.**

Based on the analysis of the three data structures (vector, hash table, and tree) for the advising program, the recommended choice is a hash table. Hash tables provide efficient insertion, deletion, and retrieval of elements on average, with O(1) time complexity for each operation in the average case. This efficiency is crucial for quickly accessing course information and checking prerequisite relationships. Additionally, hash tables are ideal for rapid existence checks, such as verifying if a course exists in the data structure or if a prerequisite course is valid. While hash tables may have higher memory overhead compared to vectors, they are generally more memory-efficient than trees, particularly for larger datasets. Moreover, hash tables are relatively simple to use and implement, with straightforward operations for adding, accessing, and removing elements. Overall, a hash table offers a good balance of efficiency and simplicity, making it a suitable choice for the advising program's requirements.