In order to fulfill the requirements set forth by the academic advisor for a Computer Science course program (ABC), the program should be able to print a sorted list of all Computer Science courses, as well as the title and prerequisites for any given course. To accomplish this, the program must use one of three data structures: a vector, hash table, or tree.

Here's the pseudocode for each data structure:

**1. Vector:**

vector<Course> courseList

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

courseList.push\_back(course)

function printCourseList():

sort(courseList)

for course in courseList:

print(course.number)

function printCourse(courseNum):

for course in courseList:

if course.number == courseNum:

print(course.title)

print(course.prereqs)

Explanation:

* The vector<Course> courseList: This line declares a vector, which is an array-like data structure, containing objects of type Course. The vector will store all of the courses that the program manages.
* The function readCourseData(file): This function accepts a filename as input and reads the data from the file. For each line in the file, the line is separated into a course number and its corresponding prerequisites (which are separated by commas), and a new Course object is created with this information. The new Course object is then added to the courseList vector.
* The function printCourseList(): This function sorts the courseList vector by course number and then prints the course numbers of each course in the list.
* The function printCourse(courseNum): This function accepts a course number as input and searches the courseList vector for a Course object with a matching course number. If a match is found, the function prints out the title of the course and its prerequisites.

Vector:

* Opening the file: 1 line of code, executes once.
* Reading the data from the file: n lines of code, executes once per line.
* Parsing each line: 4 lines of code, executes once per line.
* Creating a course object: 1 line of code, executes once per line.
* Storing course objects in the vector: 1 line of code, executes once per line.

Total cost = 1 + n + 4n + n + n = 6n + 1

Therefore, the worst-case running time of reading the file and creating course objects using a vector data structure is O(n).

Advantages of using a vector data structure:

* Easy to use and implement.
* Provides direct access to elements using an index.
* Efficient memory management.

Disadvantages of using a vector data structure:

* Insertion and deletion operations are slow for large vectors.
* Requires contiguous memory, which limits the size of the vector.

**2. Hash Table:**

hashTable<Course> courseTable

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

courseTable.insert(course.number, course)

function printCourseList():

courseList = courseTable.getKeys()

sort(courseList)

for courseNum in courseList:

print(courseNum)

function printCourse(courseNum):

course = courseTable.get(courseNum)

if course != null:

print(course.title)

print(course.prereqs)

Explanation:

* The hash table courseTable is defined, with the number attribute of each course being the key.
* The readCourseData() function reads course data from a file, creates a new Course object for each line, and then inserts the course into the courseTable using its number as the key.
* The printCourseList() function retrieves all the keys (i.e., course numbers) from the courseTable, sorts them in ascending order, and prints each course number to the console.
* The printCourse() function accepts a courseNum parameter, retrieves the course object associated with that number from the courseTable, and if the course exists, it prints the course's title and prereqs attributes to the console.
* Overall, this pseudocode uses a hash table to efficiently store and retrieve courses by their course number. It also provides two functions, one to print a list of all course numbers and another to print the details of a specific course given its number.

Hash Table:

* Opening the file: 1 line of code, executes once.
* Reading the data from the file: n lines of code, executes once per line.
* Parsing each line: 4 lines of code, executes once per line.
* Creating a course object: 1 line of code, executes once per line.
* Storing course objects in the hash table: O(1) amortized cost, executes once per line.

Total cost = 1 + n + 4n + n + n = 6n + 1

Therefore, the worst-case running time of reading the file and creating course objects using a hash table data structure is also O(n).

Advantages of using a hash table data structure:

* Provides constant-time average case performance for insertion, deletion, and lookup operations.
* Efficient memory management.
* Suitable for large datasets.

Disadvantages of using a hash table data structure:

* Can be slower than a vector for small datasets.
* Hash collisions can occur, leading to a degradation in performance.

**3. Tree:**

class TreeNode:

Course course

TreeNode leftChild

TreeNode rightChild

class CourseTree:

TreeNode root

function insertCourse(node, course):

if node is null:

node = new TreeNode(course)

else if course.number < node.course.number:

node.leftChild = insertCourse(node.leftChild, course)

else:

node.rightChild = insertCourse(node.rightChild, course)

return node

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

root = insertCourse(root, course)

function printCourseList():

inOrderTraversal(root)

function inOrderTraversal(node):

if node is not null:

inOrderTraversal(node.leftChild)

print(node.course.number)

inOrderTraversal(node.rightChild)

function printCourse(courseNum):

node = findNode(root, courseNum)

if node is not null:

print(node.course.title)

print(node.course.prereqs)

function findNode(node, courseNum):

if node is null or node.course.number == courseNum:

return node

else if courseNum < node.course.number:

return findNode(node.leftChild, courseNum)

else:

return findNode(node.rightChild, courseNum)

Explanation:

* The code utilizes a binary search tree to store Course objects. The insertCourse() function is used to add new courses to the tree.
* The readCourseData() function reads in course data from a file and uses insertCourse() to add them to the tree.
* The printCourseList() function performs an in-order traversal of the tree to display the course numbers in ascending order.
* In-order traversal is achieved with the use of the inOrderTraversal() function. The printCourse() function prints the title and prerequisites of a specific course number.
* This is achieved by using the findNode() function to locate the TreeNode containing the desired course.
* The findNode() function follows the binary search tree property to search the left or right subtree, depending on the value of the course number, until it finds the desired course or reaches a null node.
* The implementation provides a highly efficient way of searching and retrieving course data from a large collection, with most operations having a time complexity of O(log n).

Tree:

* Opening the file: 1 line of code, executes once.
* Reading the data from the file: n lines of code, executes once per line.
* Parsing each line: 4 lines of code, executes once per line.
* Creating a course object: 1 line of code, executes once per line.

Storing course objects in the tree: O(log n) cost per insertion, executes once per line.

Total cost = 1 + n + 4n + n(log n) + n(log n) = 5n(log n) + 2n + 1

Therefore, the worst-case running time of reading the file and creating course objects using a tree data structure is O(n(log n)).

Advantages of using a tree data structure:

* Provides efficient search, insertion, and deletion operations.
* Suitable for large datasets.
* Keeps the data in sorted order.

Disadvantages of using a tree data structure:

* Requires more memory than a hash table or vector.
* Slower insertion and deletion operations compared to a hash table.

The pseudocode for opening the file, reading the data, and creating course objects would be the same for each data structure, so the Big O analysis would be the same for each. The pseudocode would look something like this:

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

// add course to data structure

* The readCourseData(file) function is a general-purpose pseudocode function that reads course data from a file and inserts each course into a data structure.

**Pseudocode for Menu:**

while true:

display\_menu()

choice = get\_user\_choice()

if choice == "Load Data Structure":

data = load\_data()

print("Data loaded successfully.")

elif choice == "Print Course List":

if not data:

print("Error: no data loaded.")

else:

course\_list = get\_course\_list(data)

print\_course\_list(course\_list)

elif choice == "Print Course":

if not data:

print("Error: no data loaded.")

else:

course\_id = get\_course\_id()

course\_data = get\_course\_data(course\_id, data)

print\_course(course\_data)

elif choice == "Exit":

print("Goodbye!")

break

else:

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

Explanation:

The pseudocode for menu is laying out the structure of a program that allows the user to interact with a set of options presented as a menu. The while loop at the beginning of the pseudocode is an infinite loop that will keep the program running until the user chooses to exit.

The first step is to display the menu and prompt the user to make a choice. The code then checks which choice the user made and takes the appropriate action. In this case, the code allows the user to load data into a data structure, print a list of all the courses in the Computer Science department, or print information about an individual course. If the user chooses to exit the program, the code breaks out of the while loop and terminates.

The pseudocode assumes that there are specific functions that have been defined elsewhere in the code to perform each of these tasks. These functions would contain the specific instructions and commands to implement each option presented in the menu.

Based on the Big O analysis, we can see that the vector and hash table data structures have the same worst-case running time of O(n), while the tree data structure has a worst-case running time of O(n(log n)). Since the input size is not very large, we can choose either vector or hash table for our implementation. However, the hash table provides constant-time average case performance for insertion, deletion, and lookup operations, which is more efficient than the vector. I recommend using the hash table data structure to implement the course schedule. In addition to the constant-time average case performance for key operations, hash tables provide a simple and efficient way to store and access data, especially when the keys are unique and there is no need for maintaining order. On the other hand, vectors are useful when we need to maintain a sequence of items and access them by index, but they require more memory and potentially slower insertions and deletions when the size of the vector changes. While trees can offer advantages such as ordered traversal and efficient search for a range of values, they may be more complex to implement and maintain, and their performance depends heavily on the balance and structure of the tree. Therefore, considering the advisor's requirements and the Big O analysis, the hash table data structure seems to be the most suitable choice for our course scheduling program.