Vector  
  
Function loadDataFromFile(fileName: String): Vector<Course>

Open fileName for reading

courses = Vector<Course>()

While not end of file:

line = read next line from file

items = splitLine(line)

If items.size < 2:

Print "Error: Lines must have at least a course number and name."

Continue

course = new Course()

course.courseNumber = items[0]

course.name = items[1]

For i from 2 to items.size - 1:

course.prerequisites.add(items[i])

courses.add(course)

Close file

Return courses

Function printCourseInformation(courses: Vector<Course>, courseNumber: String)

For all courses:

If the course's number == courseNumber:

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

Print "Prerequisites:"

For each prerequisite of the course:

prerequisiteCourse = findCourseByNumber(courses, prerequisite)

If prerequisiteCourse is not null:

Print prerequisiteCourse.name + " " + prerequisiteCourse.courseNumber

Else:

Print "Prerequisite courses are missing."  
  
Hash

Function loadCoursesFromFile(fileName: String) returns Hashtable

Open fileName for reading

courses = new Hashtable

While not end of file

line = read line from file

If line has less than 2 parameters

print "Error: Invalid line format" and continue to next line

courseData = split line by comma

courseNumber = courseData[0]

courseTitle = courseData[1]

prerequisites = if courseData.length > 2 then courseData[2...] else []

course = new Course(courseNumber, courseTitle, prerequisites)

courses.add(courseNumber, course)

return courses

Function printCourseInformation(courses: Hashtable, courseNumber: String)

course = courses.get(courseNumber)

If course is null

print "Error: Course not found"

Else

print "Course Number: " + course.number

print "Course Title: " + course.title

If not course.prerequisites.isEmpty()

print "Prerequisites:"

For each prerequisite in course.prerequisites

print " - " + prerequisite

Else

print "No prerequisites"

Binary Tree  
Function LoadCoursesFromFile(filePath)

Declare Tree coursesTree

Open file at filePath

While not end of file

Read line from file

Parse line into courseData (split by commas)

If LENGTH of courseData < 2

Print "Formatting error in line: " + line

Continue

Declare Course newCourse

newCourse.courseNumber = courseData[0]

newCourse.name = courseData[1]

For i FROM 2 TO LENGTH of courseData

Add courseData[i] to newCourse.prerequisites

Insert newCourse into coursesTree

Close file

Function PrintCourseInformation(coursesTree, courseNumber)

Declare Course course = FindCourse(coursesTree, courseNumber)

If course IS NOT NULL

Print "Course Number: " + course.courseNumber

Print "Name: " + course.name

Print "Prerequisites: "

For each prerequisite IN course.prerequisites

Print prerequisite

Else

Print "Course not found"

Menu  
Function mainMenu(dataStructure)

While True

Print "Select an option:"

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course Information"

Print "4. Exit"

option = getUserInput()

If option == "1"

fileName = "course\_data.txt" // Assume this is the name of the file containing course data.

loadDataStructure(dataStructure, fileName)

Else If option == "2"

printCourseList(dataStructure)

Else If option == "3"

courseNumber = getUserInput("Enter the course number: ")

printCourse(dataStructure, courseNumber)

Else If option == "4"

Print "Exiting program."

break

Else

Print "Invalid option. Please try again."

Function loadDataStructure(dataStructure, fileName)

If dataStructure is Vector

data = loadDataFromFile(fileName) // Refer to specific loading function

Else If dataStructure is Hashtable

data = loadCoursesFromFile(fileName)

Else If dataStructure is Tree

data = LoadCoursesFromFile(fileName)

Print "Data loaded successfully."

Function printCourseList(dataStructure)

If dataStructure is Vector

Sort dataStructure by courseNumber // Assume sorting is done here

For each course in dataStructure

Print course.courseNumber + ": " + course.name

Else If dataStructure is Hashtable

Print courses in alphanumeric order from the hashtable

Else If dataStructure is Tree

Perform in-order traversal and print each course

Function printCourse(dataStructure, courseNumber)

If dataStructure is Vector

course = findCourseByNumber(dataStructure, courseNumber)

Else If dataStructure is Hashtable

course = dataStructure.get(courseNumber)

Else If dataStructure is Tree

course = FindCourse(dataStructure, courseNumber)

If course is not null

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.name

Print "Prerequisites: "

For each prerequisite in course.prerequisites

Print prerequisite

Else

Print "Course not found"

Function getUserInput(prompt = "")

Print prompt

Return input()  
  
Function printSortedCourseList\_Vector(courses: Vector<Course>)

Sort courses by courseNumber // Assuming a built-in sort function that sorts based on courseNumber

Print "List of Courses:"

For each course in courses

Print course.courseNumber + ": " + course.name  
  
Function printSortedCourseList\_HashTable(courses: Hashtable)

courseNumbers = courses.keys()

Sort courseNumbers

Print "List of Courses:"

For each number in courseNumbers

course = courses.get(number)

Print course.courseNumber + ": " + course.name  
Function printSortedCourseList\_Tree(root: Node)

If root is not null

printSortedCourseList\_Tree(root.left) // Visit left child

Print root.course.courseNumber + ": " + root.course.name // Visit node

printSortedCourseList\_Tree(root.right) // Visit right child  
Function mainMenu(dataStructure)

While True

Print "Select an option:"

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course Information"

Print "4. Exit"

option = getUserInput()

If option == "1"

fileName = "course\_data.txt" // This should be adapted based on actual file handling logic

loadDataStructure(dataStructure, fileName)

Else If option == "2"

If dataStructure is Vector

printSortedCourseList\_Vector(dataStructure)

Else If dataStructure is Hashtable

printSortedCourseList\_HashTable(dataStructure)

Else If dataStructure is Tree

printSortedCourseList\_Tree(dataStructure.root) // Assuming the tree structure has a root attribute

ElseIf option == "3"

courseNumber = getUserInput("Enter the course number: ")

printCourse(dataStructure, courseNumber)

ElseIf option == "4"

Print "Exiting program."

break

Else

Print "Invalid option. Please try again."

**Run-Time Evaluation**

Vector Data Structure

Operation: Sequential insertion into a vector.

Pseudocode Analysis:

Open file: O(1)

Read each line: O(1) per line

Parse and create a course object: O(1) per course

Append to vector: O(1) amortized, potentially O(n) when resizing is needed

Total Cost:

Reading and parsing: O(n)

Total insertion: O(n) on average; O(n^2) in the worst case due to possible frequent resizing

Advantages:

Simple implementation and direct indexing.

Efficient for small datasets or when infrequent modifications are made.

Disadvantages:

Resizing can significantly impact performance for large datasets.

Requires additional sorting for ordered operations, which is O(n log n).

Hash Table Data Structure

Operation: Hash-based insertion.

Pseudocode Analysis:

Open file: O(1)

Read each line: O(1) per line

Parse and create a course object: O(1) per course

Insert into hash table: O(1) average case, O(n) worst case due to collisions

Total Cost:

Reading and parsing: O(n)

Total insertion: O(n) on average; O(n^2) in the worst case due to collisions

Advantages:

Fast access for lookups, insertions, and deletions on average.

Direct access to courses by their course number.

Disadvantages:

Performance heavily depends on the quality of the hash function and collision resolution strategy.

Memory overhead might be significant due to storage of hash buckets and handling collisions.

Tree Data Structure (Balanced like AVL or Red-Black)

Operation: Ordered insertion into a balanced binary search tree.

Pseudocode Analysis:

Open file: O(1)

Read each line: O(1) per line

Parse and create a course object: O(1) per course

Insert into tree: O(log n)

Total Cost:

Reading and parsing: O(n)

Total insertion: O(n log n)

Advantages:

Maintains sorted order naturally, facilitating ordered operations like in-order traversal.

Provides balanced query performance.

Disadvantages:

More complex to implement and manage, especially balancing operations.

Slower insertion compared to unbalanced binary search trees due to rebalancing.

**Recommendation for Data Structure Usage**

Based on the analysis, the tree data structure, specifically a balanced binary search tree is recommended due to its ability to efficiently manage ordered data with log scale insertion and retrieval times, making it good for operations requiring both data retrieval and maintenance of order. While more complex to implement the benefits in terms of scalability and performance outweigh the initial complexity particularly for applications like a course advising system where quick access to sorted data is frequently required.