**MENU PSEUDOCODE**

BEGIN PROGRAM

DECLARE courseList AS empty list

DECLARE dataLoaded AS FALSE

WHILE TRUE

DISPLAY "Menu:"

DISPLAY "1. Load course data"

DISPLAY "2. Print all Computer Science courses (sorted)"

DISPLAY "3. Print course details and prerequisites"

DISPLAY "9. Exit"

PROMPT "Enter your choice: "

READ userChoice

IF userChoice == 1 THEN

CALL LoadCourseData(courseList)

SET dataLoaded = TRUE

DISPLAY "Course data loaded successfully."

ELSE IF userChoice == 2 THEN

IF dataLoaded == FALSE THEN

DISPLAY "Please load the course data first (Option 1)."

ELSE

CALL PrintSortedCourses(courseList)

END IF

ELSE IF userChoice == 3 THEN

IF dataLoaded == FALSE THEN

DISPLAY "Please load the course data first (Option 1)."

ELSE

PROMPT "Enter course number (e.g., CSCI200): "

READ courseNumber

CALL PrintCourseDetails(courseList, courseNumber)

END IF

ELSE IF userChoice == 9 THEN

DISPLAY "Exiting program. Goodbye!"

BREAK

ELSE

DISPLAY "Invalid option. Please try again."

END IF

END WHILE

END PROGRAM

FUNCTION LoadCourseData(courseList)

OPEN file "courses.txt"

FOR EACH line IN file

PARSE line INTO courseNumber, courseName, prerequisites

CREATE courseObject WITH parsed data

ADD courseObject TO courseList

END FOR

CLOSE file

END FUNCTION

FUNCTION PrintSortedCourses(courseList)

SORT courseList BY courseNumber

FOR EACH course IN courseList

IF courseNumber STARTS WITH "CSCI"

DISPLAY courseNumber + ": " + courseName

END IF

END FOR

END FUNCTION

FUNCTION PrintCourseDetails(courseList, courseNumber)

FOR EACH course IN courseList

IF course.courseNumber == courseNumber THEN

DISPLAY "Course: " + course.courseName

IF course.prerequisites IS EMPTY THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + course.prerequisites

END IF

RETURN

END IF

END FOR

DISPLAY "Course not found."

END FUNCTION

**VECTOR PSEUDOCODE**

FUNCTION PrintSortedCourses\_Vector(courseVector)

DECLARE sortedVector AS copy of courseVector

SORT sortedVector BY courseNumber (alphanumeric ascending)

FOR EACH course IN sortedVector

IF course.courseNumber STARTS WITH "CSCI"

DISPLAY course.courseNumber + ": " + course.courseName

END IF

END FOR

END FUNCTION

**HASH TABLE PSEUDOCODE**

FUNCTION PrintSortedCourses\_HashTable(courseHashTable)

DECLARE tempList AS empty list

FOR EACH key IN courseHashTable

IF key STARTS WITH "CSCI"

ADD courseHashTable[key] TO tempList

END IF

END FOR

SORT tempList BY courseNumber (alphanumeric ascending)

FOR EACH course IN tempList

DISPLAY course.courseNumber + ": " + course.courseName

END FOR

END FUNCTION

**BINARY SEARCH TREE PSEUDOCODE**

FUNCTION PrintSortedCourses\_Tree(courseTree)

CALL InOrderTraversal(courseTree.root)

END FUNCTION

FUNCTION InOrderTraversal(node)

IF node IS NULL

RETURN

END IF

CALL InOrderTraversal(node.left)

IF node.course.courseNumber STARTS WITH "CSCI"

DISPLAY node.course.courseNumber + ": " + node.course.courseName

END IF

CALL InOrderTraversal(node.right)

END FUNCTION

**Evaluation**

Vector

Vectors are simple and efficient for storing sequential data. Inserting each course is a constant-time operation, and the overall runtime for reading and storing data is linear, O(n). Memory usage is compact and contiguous, which benefits iteration and cache performance. However, vectors do not maintain any inherent order, so sorting is required for ordered output.

Hash Table

Hash tables offer fast average-case insertion and lookup, making them excellent for direct access to individual courses by course number. The runtime for insertion is typically O(1) per course, resulting in O(n) overall. However, in the worst case—due to hash collisions or poor distribution—the runtime can degrade to O(n²). Memory usage is higher due to bucket overhead, and since hash tables do not maintain order, additional sorting is needed for ordered output.

Binary Search Tree (BST)

BSTs, especially when balanced, provide efficient insertion and naturally maintain sorted order. Each insertion takes O(log n) time, resulting in O(n log n) overall. This structure is particularly useful for applications that require ordered traversal, such as printing courses alphanumerically. However, if the tree becomes unbalanced (e.g., inserting sorted data into an unbalanced tree), performance can degrade to O(n²). Memory usage is moderate, with each node storing pointers to children.

Recommended Structure: Binary Search Tree (BST)

Justification:

* Sorted Output: BST inherently supports in-order traversal for alphanumeric sorting without extra sorting steps.
* Scalability: With balanced insertion, runtime remains efficient at O(n log n).
* Conceptual Fit: The prerequisite structure in your flowchart aligns well with tree-based logic, making it easier to extend functionality later (e.g., dependency resolution).

If simplicity and direct access are more important than sorted output, a hash table could be a secondary choice. But for advising systems that prioritize ordered display and structured relationships, BST is the most robust and scalable option.

Let me know if you'd like to visualize this with a Gantt chart or integrate it into your final documentation.