**Vector**

BEGIN

FUNCTION ReadFile(fileName)

OPEN file fileName

IF file cannot be opened THEN

PRINT "Error: Cannot open file"

RETURN

ENDIF

INITIALIZE courseList AS VECTOR

WHILE NOT END OF FILE

READ line

SPLIT line by comma INTO courseID, courseName, prerequisites

IF courseID or courseName is empty THEN

PRINT "Formatting error in line:", line

CONTINUE

ENDIF

CREATE newCourse AS COURSE(courseID, courseName, prerequisites)

APPEND newCourse TO courseList

ENDWHILE

CLOSE file

RETURN courseList

END FUNCTION

FUNCTION PrintCourses(courseList)

FOR EACH course IN courseList

PRINT course.courseID, course.courseName, course.prerequisites

ENDFOR

END FUNCTION

END

**Hash Table**

BEGIN

FUNCTION ReadFile(fileName)

OPEN file fileName

IF file cannot be opened THEN

PRINT "Error: Cannot open file"

RETURN

ENDIF

INITIALIZE courseTable AS HASH TABLE

WHILE NOT END OF FILE

READ line

SPLIT line by comma INTO courseID, courseName, prerequisites

IF courseID or courseName is empty THEN

PRINT "Formatting error in line:", line

CONTINUE

ENDIF

CREATE newCourse AS COURSE(courseID, courseName, prerequisites)

INSERT newCourse INTO courseTable WITH KEY courseID

ENDWHILE

CLOSE file

RETURN courseTable

END FUNCTION

FUNCTION PrintCourses(courseTable)

FOR EACH courseID, course IN courseTable

PRINT course.courseID, course.courseName, course.prerequisites

ENDFOR

END FUNCTION

END

**Binary Search Tree**

BEGIN

CLASS TreeNode

PROPERTY course

PROPERTY left

PROPERTY right

END CLASS

FUNCTION InsertCourse(root, newCourse)

IF root IS NULL THEN

RETURN NEW TreeNode(newCourse)

ENDIF

IF newCourse.courseID < root.course.courseID THEN

root.left = InsertCourse(root.left, newCourse)

ELSE

root.right = InsertCourse(root.right, newCourse)

ENDIF

RETURN root

END FUNCTION

FUNCTION ReadFile(fileName)

OPEN file fileName

IF file cannot be opened THEN

PRINT "Error: Cannot open file"

RETURN

ENDIF

INITIALIZE root AS NULL

WHILE NOT END OF FILE

READ line

SPLIT line by comma INTO courseID, courseName, prerequisites

IF courseID or courseName is empty THEN

PRINT "Formatting error in line:", line

CONTINUE

ENDIF

CREATE newCourse AS COURSE(courseID, courseName, prerequisites)

root = InsertCourse(root, newCourse)

ENDWHILE

CLOSE file

RETURN root

END FUNCTION

FUNCTION PrintCoursesInOrder(root)

IF root IS NOT NULL THEN

PrintCoursesInOrder(root.left)

PRINT root.course.courseID, root.course.courseName, root.course.prerequisites

PrintCoursesInOrder(root.right)

ENDIF

END FUNCTION

END

**Menu**

BEGIN

FUNCTION DisplayMenu()

PRINT "1. Load Course Data"

PRINT "2. Print Sorted Course List"

PRINT "3. Print Course Details"

PRINT "9. Exit"

PRINT "Enter choice: "

END FUNCTION

FUNCTION Main()

INITIALIZE data AS NULL, isLoaded AS FALSE

DO

CALL DisplayMenu()

READ choice

SWITCH(choice)

CASE 1:

PRINT "Enter file name: "

READ fileName

data = ReadFile(fileName)

isLoaded = (data IS NOT NULL)

PRINT (isLoaded ? "Data loaded." : "Load failed.")

BREAK

CASE 2:

IF NOT isLoaded THEN PRINT "Load data first." ELSE PrintCourses(data)

BREAK

CASE 3:

IF NOT isLoaded THEN PRINT "Load data first."

ELSE

PRINT "Enter course ID: "

READ courseID

course = SearchCourse(data, courseID)

PRINT (course ? course.courseID, course.courseName, course.prerequisites : "Not found.")

ENDIF

BREAK

CASE 4:

PRINT "Exiting."

EXIT

DEFAULT:

PRINT "Invalid choice."

END SWITCH

WHILE choice != 9

END FUNCTION

END

**Vector Implementation (Sorting & Printing)**

BEGIN

FUNCTION SortAndPrintCourses(courseList)

SORT courseList BY courseID (ascending)

PRINT "Computer Science Courses:"

FOR EACH course IN courseList

PRINT course.courseID, course.courseName

ENDFOR

END FUNCTION

END

**Hash Table Implementation (Sorting & Printing)**

BEGIN

FUNCTION SortAndPrintCourses(courseTable)

INITIALIZE courseList AS EMPTY VECTOR

FOR EACH courseID, course IN courseTable

APPEND course TO courseList

ENDFOR

SORT courseList BY courseID (ascending)

PRINT "Computer Science Courses:"

FOR EACH course IN courseList

PRINT course.courseID, course.courseName

ENDFOR

END FUNCTION

END

**Binary Search Tree (BST) Implementation (Sorting & Printing)**

BEGIN

FUNCTION PrintCoursesInOrder(root)

IF root IS NOT NULL THEN

PrintCoursesInOrder(root.left)

PRINT root.course.courseID, root.course.courseName

PrintCoursesInOrder(root.right)

ENDIF

END FUNCTION

END

**Big O Analysis of Reading the File & Creating Course Objects**

Vector Implementation

* Insertion: *O*(1) (Appending to the end)
* Total Complexity:
  + Reading + Parsing: *O(n)*
  + Creating Objects: *O(n)*
  + Appending to Vector: *O(n)*
  + Final Complexity: *O(n)*

Hash Table Implementation

* Insertion into Hash Table: *O(1)* per element → *O(n)* total
* Total Complexity:
  + Reading + Parsing: *O(n)*
  + Creating Objects: *O(n)*
  + Hash Table Insertion: *O(n)*
  + Final Complexity: *O(n)*

Tree Implementation (Binary Search Tree - BST)

* Insertion into BST: *O(log n)* per element → *O(n log n)* total
* Total Complexity:
  + Reading + Parsing: *O(n)*
  + Creating Objects: *O(n)*
  + BST Insertion: *O(n log n)*
  + Final Complexity: *O(n log n)*

**Evaluation**

Advantages & Disadvantages of Each Structure

**Vector**

Advantages

* Fast insertion O(1)O(1)O(1) when appending
* Simple implementation
* Efficient memory usage (low overhead)

Disadvantages

* Slow searching O(n)O(n)O(n) (must scan entire vector)
* Sorting required O(nlog⁡n)O(n \log n)O(nlogn) before printing

**Hash Table**

Advantages

* Fast lookups O(1)O(1)O(1) (direct access using key)
* Efficient insertion O(1)O(1)O(1)

Disadvantages

* High memory overhead (storing keys, values, handling collisions)
* No inherent ordering (must extract & sort for alphabetical order)

**Binary Search Tree (BST)**

Advantages

* Automatically sorted order (in-order traversal)
* Faster than vector for searching O(log⁡n)O(\log n)O(logn)

Disadvantages

* Slower insertion O(log⁡n)O(\log n)O(logn) compared to vector/hash table
* More complex implementation
* Unbalanced BST can degrade to O(n)O(n)O(n) in worst case (e.g., if data is already sorted)

**Recommendation**

Best Choice: Hash Table

* Fastest lookup speed, making it ideal for searching specific course details.
* Efficient insertion, making it fast to load courses from the file.
* Although it lacks ordering, we can extract data into a vector and sort it only when needed, which is more efficient than keeping the entire structure sorted at all times.

Alternative Choice: Vector (if memory is a concern)

* If memory efficiency is a priority, a vector may be better because it has less overhead compared to a hash table. However, searching would be much slower, making it less efficient for real-time lookups.