```
MAIN
  DISPLAY "1. Load course data"
  DISPLAY "2. Print all courses in alphanumeric order"
  DISPLAY "3. Print course title and prerequisites"
  DISPLAY "9. Exit"

  DECLARE choice AS INTEGER

  WHILE choice != 9
    INPUT choice

    IF choice == 1 THEN
      CALL LoadCourseData()
    ELSE IF choice == 2 THEN
      CALL PrintCoursesInOrder()
    ELSE IF choice == 3 THEN
      CALL PrintCourseDetails()
    ELSE IF choice == 9 THEN
      DISPLAY "Exiting program..."
    ELSE
      DISPLAY "Invalid choice. Please select a valid option."

END MAIN

LoadCourseData()
  DISPLAY "Loading course data..."
```

```
    // Pseudocode for loading data into vector
    CALL LoadDataIntoVector()
    // Pseudocode for loading data into hash table
    CALL LoadDataIntoHashTable()
    // Pseudocode for loading data into tree
    CALL LoadDataIntoTree()
    DISPLAY "Course data loaded successfully."

END LoadCourseData

PrintCoursesInOrder()
    DISPLAY "Printing courses in alphanumeric order..."
    // Pseudocode for printing courses from vector
    CALL PrintCoursesFromVector()
    // Pseudocode for printing courses from hash table
    CALL PrintCoursesFromHashTable()
    // Pseudocode for printing courses from tree
    CALL PrintCoursesFromTree()
    DISPLAY "Courses printed successfully."

END PrintCoursesInOrder

PrintCourseDetails()
    DECLARE courseNumber AS STRING
    DISPLAY "Enter the course number: "
    INPUT courseNumber
```

```
  // Pseudocode for printing course details from vector
  CALL PrintCourseDetailsFromVector(courseNumber)
  // Pseudocode for printing course details from hash table
  CALL PrintCourseDetailsFromHashTable(courseNumber)
  // Pseudocode for printing course details from tree
  CALL PrintCourseDetailsFromTree(courseNumber)

END PrintCourseDetails

LoadDataIntoVector()
  // Implement loading data into a vector
  DECLARE vector AS LIST
  OPEN "course_data.txt" FOR READING
  WHILE NOT END OF FILE
    READ line
    PARSE line INTO course
    ADD course TO vector
  CLOSE FILE

END LoadDataIntoVector

LoadDataIntoHashTable()
  // Implement loading data into a hash table
  DECLARE hashTable AS DICTIONARY
  OPEN "course_data.txt" FOR READING
  WHILE NOT END OF FILE
```

```
    READ line
    PARSE line INTO course
    ADD course TO hashTable WITH KEY course.number
  CLOSE FILE

END LoadDataIntoHashTable

LoadDataIntoTree()
  // Implement loading data into a tree
  DECLARE tree AS BINARY_SEARCH_TREE
  OPEN "course_data.txt" FOR READING
  WHILE NOT END OF FILE
    READ line
    PARSE line INTO course
    INSERT course INTO tree
  CLOSE FILE

END LoadDataIntoTree

PrintCoursesFromVector()
  SORT vector BY course.number
  FOR EACH course IN vector
    DISPLAY course.number, course.title

END PrintCoursesFromVector
```

```
PrintCoursesFromHashTable()
  DECLARE courseNumbers AS LIST OF KEYS IN hashTable
  SORT courseNumbers
  FOR EACH number IN courseNumbers
    DISPLAY number, hashTable[number].title

END PrintCoursesFromHashTable


PrintCoursesFromTree()
  // In-order traversal of the binary search tree to print courses
  CALL InOrderTraversal(tree.root)

END PrintCoursesFromTree


InOrderTraversal(node)
  IF node IS NOT NULL
    CALL InOrderTraversal(node.left)
    DISPLAY node.course.number, node.course.title
    CALL InOrderTraversal(node.right)

END InOrderTraversal


PrintCourseDetailsFromVector(courseNumber)
  FOR EACH course IN vector
    IF course.number == courseNumber
      DISPLAY course.title
```

```
            DISPLAY "Prerequisites: ", course.prerequisites
            RETURN

    DISPLAY "Course not found."

END PrintCourseDetailsFromVector

PrintCourseDetailsFromHashTable(courseNumber)
    IF courseNumber EXISTS IN hashTable
        DISPLAY hashTable[courseNumber].title
        DISPLAY "Prerequisites: ", hashTable[courseNumber].prerequisites
    ELSE
        DISPLAY "Course not found."

END PrintCourseDetailsFromHashTable

PrintCourseDetailsFromTree(courseNumber)
    DECLARE node AS tree.root
    WHILE node IS NOT NULL
        IF courseNumber == node.course.number
            DISPLAY node.course.title
            DISPLAY "Prerequisites: ", node.course.prerequisites
            RETURN
        ELSE IF courseNumber < node.course.number
            node = node.left
        ELSE
```

```
      node = node.right

  DISPLAY "Course not found."

END PrintCourseDetailsFromTree
```

Runtime Analysis

Scenarios	**Vector**	**Hash Table**	**Binary Search Tree**
Loading Data	• Reading the file and parsing each line: O(n) • Inserting into vector: O(1) per insertion, O(n) total.	• Reading the file and parsing each line: O(n) • Inserting into Has Table O(1) per insertion, O(n) total.	• Reading the file and parsing each line: O(n) • Inserting into BST: O(n log n) per insertion, O(n log n) total
Printing Courses in Alphanumeric Order	• Sorting: O(n log n) • Printing: O(n)	• Collecting keys and sorting: O(n log n) • Printing: O(n)	• In-order traversal O(n)
Memory Usage	• Memory for storing n courses: O(n)	• Memory for storing n courses with additional overhead for hash table structure: O(n)	• Memory for storing n courses: O(n)
Advantages	• Simple to implement.	• Efficient for lookups, insertions, and	• Efficient for sorted order retrieval.

	• Efficient for iterating over all courses.	deletions: O(1) average case • Good for quickly accessing individual courses.	• Balanced BSTs offer efficient insertions, deletions and lookups.
Disadvantages	• Inefficient for frequent insertions and deletions. • Sorting is required to print courses in order.	• Hash collisions can degrade performance. • Requires additional memory for hash table overhead.	• Performance can degrade to O(n) if the tree becomes unbalanced. • More complex to implement and maintain compared to vectors and hash tables.

Recommendation:

Based on the Big O analysis and the advantages/disadvantages of each data structure, I recommend using a hash table for this application.

Efficiency for Insertions and Lookups allows the hash table to provide O(1) average case time complexity for insertions and lookups, which is efficient for loading and accessing course data.

Printing Sorted Courses although requiring sorting to print courses in order, this provides an O(n log n) time complexity, comparable to the vector.

Memory Overhead is very acceptable given the significant performance benefits in lookup and insertion operations.

Practicality is very strong with hash tables, being straightforward to implement and handle large datasets efficiently, making them suitable for this application.