Pseudocode and Runtime Analysis

Pseudocode for Vector

```
Step 1: File Input and Validation
function LoadCourses(filePath):
    open file at filePath
    if file cannot be opened:
        print "Error: Cannot open file"
        return empty vector

    create empty vector called courses

    for each line in the file:
        split line by comma into:
            - courseNumber
            - courseName
            - prerequisites (if any)

        if less than two tokens are found:
            print "Error: Invalid format on line"
            continue to next line

        create Course object using the courseNumber, courseName, prerequisites
        add Course to the courses vector

    close file
    return courses

Step 2: Create Course Object Structure
struct Course:
    courseNumber: string
    courseName: string
    prerequisites: list of strings
```

```
function CreateCourse(courseNumber, courseName, prerequisites):
    initialize new Course object
    set courseNumber
    set courseName
    set prerequisites
    return Course object

Step 3: Print Course Information and Prerequisites
function PrintCourseInformation(courses, courseNumber):
    courseFound = false
    for each course in courses:
        if course.courseNumber == courseNumber:
            print "Course Number: " + course.courseNumber
            print "Course Name: " + course.courseName
            if prerequisites exist:
                print "Prerequisites: " + join(prerequisites, ", ")
            else:
                print "No prerequisites"
            courseFound = true
            break
    if not courseFound:
        print "Course not found."

Step 4: Menu Functionality
function MainMenu():
    filePath = "courses.csv"
    courses = LoadCourses(filePath)

    while true:
        print "1. Load Courses"
        print "2. Print All Courses"
        print "3. Find Course"
        print "9. Exit"
        userChoice = get user input

        if userChoice == 1:
```

```
        courses = LoadCourses(filePath)
    elif userChoice == 2:
        PrintAllCourses(courses)
    elif userChoice == 3:
        courseNumber = get user input
        PrintCourseInformation(courses, courseNumber)
    elif userChoice == 9:
        break
    else:
        print "Invalid choice"

Step 5: Print All Courses in Alphanumeric Order
function PrintAllCourses(courses):
    sort courses by courseNumber in ascending order
    for each course in courses:
        PrintCourseInformation(courses, course.courseNumber)
```

Pseudocode for Hash Table

```
Step 1: File Input and Validation
function LoadCourses(filePath):
    open file at filePath
    if file cannot be opened:
        print "Error: Cannot open file"
        return empty hash table

    create an empty hash table called courses

    for each line in the file:
        split line by comma into:
            - courseNumber
            - courseName
```

```
        - prerequisites (if any)

      if less than two tokens are found:
         print "Error: Invalid format on line"
         continue to next line

      create Course object using the courseNumber, courseName, prerequisites
      insert Course into the hash table

   close file
   return courses

Step 2: Create Course Object Structure
struct Course:
   courseNumber: string
   courseName: string
   prerequisites: list of strings

function CreateCourse(courseNumber, courseName, prerequisites):
   initialize new Course object
   set courseNumber
   set courseName
   set prerequisites
   return Course object

Step 3: Print Course Information and Prerequisites
function PrintCourseInformation(courses, courseNumber):
   course = SearchCourse(courses, courseNumber)
   if course is not null:
      print "Course Number: " + course.courseNumber
      print "Course Name: " + course.courseName
      if prerequisites exist:
         print "Prerequisites: " + join(prerequisites, ", ")
      else:
         print "No prerequisites"
   else:
```

```
    print "Course not found."

Step 4: Menu Functionality
function MainMenu():
  filePath = "courses.csv"
  courses = LoadCourses(filePath)

  while true:
    print "1. Load Courses"
    print "2. Print All Courses"
    print "3. Find Course"
    print "9. Exit"
    userChoice = get user input

    if userChoice == 1:
      courses = LoadCourses(filePath)
    elif userChoice == 2:
      PrintAllCourses(courses)
    elif userChoice == 3:
      courseNumber = get user input
      PrintCourseInformation(courses, courseNumber)
    elif userChoice == 9:
      break
    else:
      print "Invalid choice"

Step 5: Print All Courses in Alphanumeric Order
function PrintAllCourses(courses):
  sort courses by courseNumber in ascending order
  for each course in courses:
    PrintCourseInformation(courses, course.courseNumber)
```

Pseudocode for Binary Search Tree

```
Step 1: File Input and Validation
function LoadCourses(filePath):
   open file at filePath
   if file cannot be opened:
      print "Error: Cannot open file"
      return empty binary search tree

   create an empty binary search tree called courses

   for each line in the file:
      split line by comma into:
         - courseNumber
         - courseName
         - prerequisites (if any)

      if less than two tokens are found:
         print "Error: Invalid format on line"
         continue to next line

      create Course object using the courseNumber, courseName, prerequisites
      insert Course into the binary search tree

   close file
   return courses

Step 2: Create Course Object Structure
struct Course:
   courseNumber: string
   courseName: string
   prerequisites: list of strings

function CreateCourse(courseNumber, courseName, prerequisites):
   initialize new Course object
   set courseNumber
   set courseName
```

```
    set prerequisites
    return Course object

Step 3: Print Course Information and Prerequisites
function PrintCourseInformation(tree, courseNumber):
    course = SearchCourse(tree, courseNumber)
    if course is not null:
        print "Course Number: " + course.courseNumber
        print "Course Name: " + course.courseName
        if prerequisites exist:
            print "Prerequisites: " + join(prerequisites, ", ")
        else:
            print "No prerequisites"
    else:
        print "Course not found."

Step 4: Menu Functionality
function MainMenu():
    filePath = "courses.csv"
    courses = LoadCourses(filePath)

    while true:
        print "1. Load Courses"
        print "2. Print All Courses"
        print "3. Find Course"
        print "9. Exit"
        userChoice = get user input

        if userChoice == 1:
            courses = LoadCourses(filePath)
        elif userChoice == 2:
            PrintAllCourses(courses)
        elif userChoice == 3:
            courseNumber = get user input
            PrintCourseInformation(courses, courseNumber)
        elif userChoice == 9:
```

```
        break
    else:
        print "Invalid choice"

Step 5: Print All Courses in Alphanumeric Order
function PrintAllCourses(tree):
  if tree is not empty:
    PrintAllCourses(tree.left)
    PrintCourseInformation(tree, tree.root.courseNumber)
    PrintAllCourses(tree.right)
```

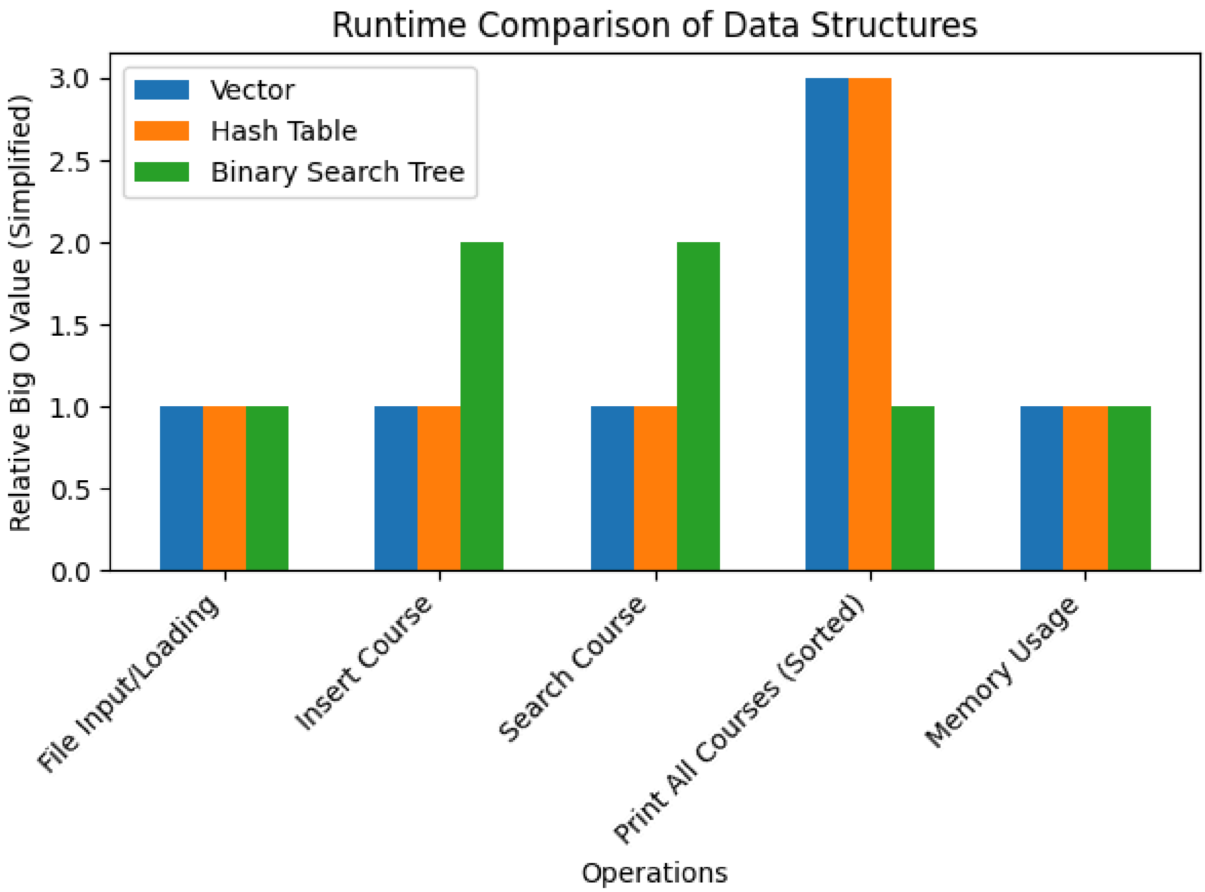
Runtime Analysis

A more detailed breakdown of the runtime for each key function is required. Here's the **Big O analysis** for the pseudocode:

1. Run Time and Memory Evaluation of Data Structures

For each data structure (Vector, Hash Table, Binary Search Tree), the pseudocode written already accounts for **opening the file**, **reading data**, **parsing lines**, and **creating Course objects**. We need to evaluate the **worst-case running time (Big O)** for each operation, focusing on loading data, inserting into the structure, and searching for courses.

Big O Time Complexity and Memory Usage Evaluation



S/N	Operation	Vector	Hash Table	Binary Search Tree
1.	File Input/Loading	O(n)	O(n)	O(n)
2.	Insert Course	O(1)	O(1)	O(log n) on average
3.	Search Course	O(n)	O(1)	O(log n) (O(n) worst-case if unbalanced)
4.	Print All Courses (Sorted)	O(n log n) (sorting)	O(n log n) (sorting)	O(n) (since it's already sorted)
5.	Memory Usage	O(n)	O(n)	O(n)

2. Cost Per Line of Code

The analysis below assumes that the **cost per line of code** is **1**, unless it involves calling a function, in which case the price will reflect the **function's running time**:

Vector:

- **Loading data**: For each course, the program splits the line, validates, and creates the object. This happens in **O(n)** because each line (n courses) is processed independently.
- **Insert into vector**: Since the vector allows constant-time insertions at the end, this operation runs in **O(1)**.
- **Search Course**: Searching a vector requires iterating through all courses for a worst-case time of **O(n)**.
- **Sorting** (when printing): To print in alphanumeric order, a **sort** operation is required: **O (n log n)**.

Hash Table:

- **Loading data**: Similar to vector, this is **O(n)** since each line is processed and validated.
- **Insert into hash table**: Hash table insertions are generally **O(1)** unless a collision occurs (which can be avoided with a good hash function).
- **Search Course**: Searching is **O(1),** as hash tables are designed for fast lookups.
- **Sorting**: Hash tables do not maintain order, so printing in order requires **O(n log n)** sorting.

Binary Search Tree:

- **Loading data**: Given balanced insertions, processing each line and inserting it into a binary search tree takes **O(n log n)** time. Worst-case insertions are **O(n)** for an unbalanced tree.
- **Search Course**: Due to the nature of the binary search tree, search time is O(log n) on average, though it could degrade to **O(n)** if unbalanced.
- **Printing (in-order traversal)**: Binary search trees are naturally sorted, so printing all courses in order takes **O(n)** (no extra sorting needed).

3. Advantages and Disadvantages of Each Data Structure

Each structure has strengths and weaknesses based on how you need to use it. Here's a breakdown:

S/N		Advantages	Disadvantages
1.	Vector	Easy to implement and manage. Best for small datasets or where simplicity is important	Searching is slow with **O(n)** in the worst case. Printing requires sorting, which is **O(n log n)**.

2.	Hash Table	Very fast lookups and insertions with average **O(1)** time complexity. Efficient for large datasets where fast searching is key.	Does not maintain sorted order. Printing in alphanumeric order requires an additional **O(n log n)** sorting step.
3.	Binary Search Tree	Naturally maintains data in sorted order, making **in-order traversals** efficient for printing. Efficient searching with **O(log n)** in the average case.	Insertions and searches degrade to **O(n)** if the tree becomes unbalanced. Slightly more complex to implement than a vector or hash table.

4. Recommendation Based on Big O Analysis

Given the requirements of the academic advisors at ABCU—particularly needing to print the courses in **alphanumeric order** and search by course number—the **Binary Search Tree (BST)** offers the most balanced performance:

- **Searches** are **O(log n)** (on average).
- **In-order traversal** allows printing the courses in **O(n)** time without extra sorting.
- While unbalanced trees could degrade performance, using a self-balancing tree (such as AVL or Red-Black) can prevent that.

Thus, I recommend the **Binary Search Tree** for the program. The sorted order requirement plays a crucial role here, and BSTs naturally handle this efficiently. The other structures (Vector and Hash Table) require additional sorting steps, which adds complexity and inefficiency to the program.