1. Resubmit pseudocode from previous pseudocode assignments and update as necessary:

- Vector:

- a. Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors:

```
function loadCoursesFromFile(filename):
    // Open the file for reading
    file = open(filename, "r")

    // Initialize an empty vector for courses
    courses = Vector<Course>()

    // Read each line in the file
    for each line in file:
        // Split the line by commas
        data = line.split(",")

        // Check if the line has at least two parameters (course number and title)
        if length(data) < 2:
            print "Error: Line does not have enough parameters"
            continue

        // Extract course number, title, and prerequisites
        courseNumber = data[0]
        courseTitle = data[1]
        prerequisites = data[2:]

        // Create a new course object
        course = Course(courseNumber, courseTitle, prerequisites)

        // Store the course object in the vector
        courses.append(course)

    // Close the file
    file.close()

    return courses
```

- b. Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file:

```
class Course:
    function __init__(self, number, title, prerequisites):
```

```
        self.number = number
        self.title = title
        self.prerequisites = prerequisites
```

- c. Design pseudocode that will print out course information and prerequisites:

```
function printCourseInfo(courses, courseNumber):
    // Iterate through the vector to find the course
    for each course in courses:
        if course.number == courseNumber:
            // Print the course information
            print "Course Number:", course.number
            print "Course Title:", course.title
            print "Prerequisites:"
            for each prereq in course.prerequisites:
                print " -", prereq
            return

    // If course is not found
    print "Course", courseNumber, "not found"
```

- Hash Table:

- a. Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors:

```
function loadCoursesFromFile(filename):
    // Open the file for reading
    file = open(filename, "r")

    // Initialize an empty hash table for courses
    courses = HashTable<Course>()

    // Read each line in the file
    for each line in file:
        // Split the line by commas
        data = line.split(",")

        // Check if the line has at least two parameters (course number and title)
        if length(data) < 2:
            print "Error: Line does not have enough parameters"
            continue

        // Extract course number, title, and prerequisites
        courseNumber = data[0]
        courseTitle = data[1]
```

```
        prerequisites = data[2:]

        // Validate prerequisites
        for each prereq in prerequisites:
            if not courses.contains(prereq):
                print "Error: Prerequisite course", prereq, "does not exist"
                continue

        // Create a new course object
        course = Course(courseNumber, courseTitle, prerequisites)

        // Store the course object in the hash table
        courses.put(courseNumber, course)

    // Close the file
    file.close()

    return courses
```

- b. Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file:

```
class Course:
    function __init__(self, number, title, prerequisites):
        self.number = number
        self.title = title
        self.prerequisites = prerequisites
```

- c. Design pseudocode that will print out course information and prerequisites:

```
function printCourseInfo(courses, courseNumber):
    // Check if the course exists in the hash table
    if not courses.contains(courseNumber):
        print "Course", courseNumber, "not found"
        return

    // Get the course object from the hash table
    course = courses.get(courseNumber)

    // Print the course information
    print "Course Number:", course.number
    print "Course Title:", course.title
    print "Prerequisites:"
    for each prereq in course.prerequisites:
        print " -", prereq
```

- Binary Search Tree:

- a. Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors:

```
function loadCoursesFromFile(filename):
    // Open the file for reading
    file = open(filename, "r")

    // Initialize an empty binary search tree for courses
    courses = BinarySearchTree<Course>()

    // Read each line in the file
    for each line in file:
        // Split the line by commas
        data = line.split(",")

        // Check if the line has at least two parameters (course number and title)
        if length(data) < 2:
            print "Error: Line does not have enough parameters"
            continue

        // Extract course number, title, and prerequisites
        courseNumber = data[0]
        courseTitle = data[1]
        prerequisites = data[2:]

        // Create a new course object
        course = Course(courseNumber, courseTitle, prerequisites)

        // Insert the course object into the binary search tree
        courses.insert(course)

    // Close the file
    file.close()

    return courses
```

- b. Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file:

```
class Course:
    function __init__(self, number, title, prerequisites):
        self.number = number
        self.title = title
        self.prerequisites = prerequisites
```

- c. Design pseudocode that will print out course information and prerequisites:

```
function printCourseInfo(courses, courseNumber):
    // Search for the course in the binary search tree
    course = courses.search(courseNumber)

    // If course is found, print the information
    if course is not None:
        print "Course Number:", course.number
        print "Course Title:", course.title
        print "Prerequisites:"
        for each prereq in course.prerequisites:
            print " -", prereq
    else:
        print "Course", courseNumber, "not found"
```

2. Create pseudocode for a menu (universal):

- a. Option 1: Load the file data into the data structure.
  - Display message: "Enter the filename: "
  - Load the courses using `loadCoursesFromFile(filename)`
  - Store the courses in the data structure

- b. Option 2: Print an alphanumerically ordered list of all the courses.
  - Sort the courses by course number
  - Print the sorted list of courses

- c. Option 3: Print the course title and prerequisites for any individual course.
  - Display message: "Enter course number: "
  - Search for the course in the data structure
  - If course is found, print course title and prerequisites
  - If course is not found, display "Course not found" message

- d. Option 9: Exit the program.
  - Display message: "Exiting program."
  - Break the loop to exit

3. Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order:

- a. Sort the course information by alphanumeric course number from lowest to highest.
  - If using a vector or a list:
    - Use a sorting algorithm to sort the list of courses based on the `courseNumber`.
    - Example: `sortedCourses = sort(courses, by="courseNumber")`

- If using a hash table:
  - Convert the hash table to a list or array.
  - Sort the resulting list based on the `courseNumber`.
  - Example: `sortedCourses = sort(hashTable.toList(), by="courseNumber")`
- If using a binary search tree:
  - Perform an in-order traversal of the tree to naturally get the courses in sorted order.
  - Example: `inOrderTraversal(root)`

- b. Print the sorted list to a display.
  - Iterate through the sorted list (or the result of the in-order traversal if using a binary search tree).
  - For each course in the sorted list:
    - Print "Course Number: [course number]"
    - Print "Course Title: [course title]"

4. Evaluate the run time and memory of data structures that could be used to address the requirements:

For a Vector, the worst-case time complexity for reading the file, parsing each line, and creating course objects is O(n). Here, each line of the file is processed once, and inserting an element into the vector is O(1) per operation, leading to a total time complexity of O(n). The memory usage for a vector is also O(n), as it requires storage for each course object.

In the case of a Hash Table, the time complexity for inserting elements is O(1) on average due to direct indexing, but in the worst case—such as when there are many hash collisions—the time complexity can increase to O(n) for a single operation, making the total time complexity O(n^2) in the worst case. Memory usage remains O(n), though it may be higher depending on how collisions are resolved, such as with chaining or open addressing.

The Binary Search Tree (BST) has a time complexity of O(log n) per insertion if the tree remains balanced, leading to an overall time complexity of O(n log n) for processing all n lines. However, in the worst case, if the tree becomes unbalanced (essentially becoming a linked list), the insertion time complexity could degrade to O(n), making the total time complexity O(n^2).

The memory usage for a BST is O(n) as it needs to store each node, which corresponds to each course.

5. Analyze each of the vector, hash table, and tree data structures

Vector:

- Advantages: Vectors are simple to implement and offer constant time complexity O(1) for accessing elements by index. They are also efficient in terms of memory usage, with a time complexity of O(n) for operations like reading data, parsing, and inserting elements sequentially.
- Disadvantages: The main drawback of using vectors is their inefficiency in insertion and deletion operations when the order of elements matters, as these operations can degrade to O(n) in the worst case. Additionally, if the vector needs to grow in size, this could involve costly resizing operations.

Hash Table:

- Advantages: Hash tables offer average-case constant time complexity O(1) for insertion, deletion, and search operations, making them very efficient for large datasets where quick access is necessary. They are well-suited for scenarios where the data does not need to be ordered.
- Disadvantages: The primary disadvantage of hash tables is the potential for hash collisions, which can lead to degraded performance, especially in the worst case, where the time complexity can increase to O(n). Managing collisions requires additional memory and can complicate the implementation.

Binary Search Tree (BST):

- Advantages: Binary Search Trees maintain elements in a sorted order, allowing for efficient search, insertion, and deletion operations, particularly in balanced trees where

these operations have a time complexity of O(log n). This is useful when the data needs to be ordered and quickly accessed.

- Disadvantages: The main disadvantage of BSTs arises when the tree becomes unbalanced, which can lead to a worst-case time complexity of O(n) for operations. This unbalanced state can degrade performance significantly, turning the BST into a linear structure similar to a linked list.

Operation	Vector	Hash Table	Binary Search Tree (BST)
Reading and Parsing File	O(n)	O(n)	O(n)
Inserting Data	O(1)	O(1) avg, O(n) worst	O(log n) balanced, O(n) worst
Searching for a Course	O(n)	O(1) avg, O(n) worst	O(log n) balanced, O(n) worst
Sorting Data	O(n log n)	O(n log n)	O(n) (in-order traversal)
Memory Usage	O(n)	O(n)	O(n)

6. Recommendation:

After analyzing the three data structures—Vector, Hash Table, and Binary Search Tree (BST)—I recommend using a Hash Table for this program. The Hash Table's ability to provide constant time complexity O(1) for most operations makes it the most efficient option, especially when the primary tasks involve frequent insertions, deletions, and lookups. Although it has the potential for hash collisions, the average case performance remains superior compared to Vectors and Binary Search Trees, particularly for large datasets.

The BST, while efficient for ordered data, poses a risk of becoming unbalanced, which could severely impact performance. The Vector, although simple, does not provide the same level of efficiency for dynamic operations as the Hash Table. Therefore, based on the Big O analysis and the program's requirements, the Hash Table is the optimal choice for balancing speed and memory usage in this scenario.