1. Pseudocode from Previous Pseudocode Assignments and Update as Necessary

For each data structure—vector, hash table, and binary search tree (BST)—we need to resubmit and update the pseudocode for:

- Opening the file, reading, parsing, and checking for errors
- Creating course objects
- Printing course information

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:

Vector:

```
void loadFile(String fileName, Vector<Course> courseList) {
    Open file "course_data.txt"
    While not end of file:
        String line = readLine from file
        String[] data = split(line, ",")

        if data.length < 2:
            print "Error: Line must contain at least a course number and title"
            Continue to the next line

        String courseNumber = data[0]
        String courseTitle = data[1]
        List<String> prerequisites = []

        for (int i = 2; i < data.length; i++) {
            prerequisites.add(data[i])
        }

        Course course = new Course(courseNumber, courseTitle, prerequisites)
        courseList.add(course)
    }
    Close file
}
```

Hash Table:

```
void loadFile(String fileName, HashTable<String, Course> courseTable) {
    Open file "course_data.csv"

    While not end of file:
        String line = readLine from file
        String[] data = split(line, ",")

        if data.length < 2:
            print "Error: Missing course information"
            Continue to the next line

        String courseNumber = data[0]
        String courseTitle = data[1]
        List<String> prerequisites = []

        for (int i = 2; i < data.length; i++) {
            prerequisites.add(data[i])
        }

        Course course = new Course(courseNumber, courseTitle, prerequisites)
        courseTable.insert(courseNumber, course)
    }
    Close file
}
```

Binary Search Tree:

```
void loadFile(String fileName, Tree<Course> courseTree) {
    Open file "course_data.txt"

    While not end of file:
        String line = readLine from file
        String[] data = split(line, ",")

        if data.length < 2:
            print "Error: Invalid course data format"
            Continue to the next line

        String courseNumber = data[0]
        String courseTitle = data[1]
        List<String> prerequisites = []

        for (int i = 2; i < data.length; i++) {
            prerequisites.add(data[i])
        }

        Course course = new Course(courseNumber, courseTitle, prerequisites)
        courseTree.insert(course)
    }
    Close file
}
```

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

Vector:

```
void createCourse(Vector<Course> courseList, String[] data) {
    String courseNumber = data[0]
    String courseTitle = data[1]
    List<String> prerequisites = []

    for (int i = 2; i < data.length; i++) {
        prerequisites.add(data[i])
    }

    Course course = new Course(courseNumber, courseTitle, prerequisites)
    courseList.add(course)
}
```

Hash Table:

```
void createCourse(HashTable<String, Course> courseTable, String[] data) {
    String courseNumber = data[0]
    String courseTitle = data[1]
    List<String> prerequisites = []

    for (int i = 2; i < data.length; i++) {
        prerequisites.add(data[i])
    }

    Course course = new Course(courseNumber, courseTitle, prerequisites)
    courseTable.insert(courseNumber, course)
}
```

Binary Search Tree:

```
void createCourse(Tree<Course> courseTree, String[] data) {
    String courseNumber = data[0]
    String courseTitle = data[1]
    List<String> prerequisites = []

    for (int i = 2; i < data.length; i++) {
        prerequisites.add(data[i])
    }

    Course course = new Course(courseNumber, courseTitle, prerequisites)
    courseTree.insert(course)
}
```

c. Design Pseudocode That Will Print Out Course Information and Prerequisites

Vector:

```
void printCourseInfo(Vector<Course> courseList, String courseNumber) {
    for (Course course : courseList) {
        if course.courseNumber equals courseNumber:
            print "Course Number: " + course.courseNumber
            print "Course Title: " + course.courseTitle
            if course.prerequisites is not empty:
                print "Prerequisites: " + course.prerequisites
            else:
                print "No prerequisites"
            break
    }
}
```

Hash Table :

```
void printCourseInfo(HashTable<String, Course> courseTable, String courseNumber) {
    Course course = courseTable.get(courseNumber)
    if course is null:
        print "Course not found"
    else:
        print "Course Number: " + course.courseNumber
        print "Course Title: " + course.courseTitle
        if course.prerequisites is not empty:
            print "Prerequisites: " + course.prerequisites
        else:
            print "No prerequisites"
    }
}
```

Binary Search Tree:

```
void printCourseInfo(Tree<Course> courseTree, String courseNumber) {
    Course course = courseTree.search(courseNumber)
    if course is null:
        print "Course not found"
    else:
        print "Course Number: " + course.courseNumber
        print "Course Title: " + course.courseTitle
        if course.prerequisites is not empty:
            print "Prerequisites: " + course.prerequisites
        else:
            print "No prerequisites"
    }
}
```

2. Create Pseudocode for a Menu

Menu Pseudocode:

```
void displayMenu() {
    while true:
        print "1. Load data from file"
        print "2. Print sorted list of courses"
        print "3. Print specific course information"
        print "9. Exit"

        String choice = get user input
        switch (choice):
            case "1":
                loadFile("course_data.txt", dataStructure)
                break
            case "2":
                printSortedCourses(dataStructure)
                break
            case "3":
                String courseNumber = get user input for course number
                printCourseInfo(dataStructure, courseNumber)
                break
            case "9":
                exit program
            default:
                print "Invalid choice"
}
```

3. Design Pseudocode That Will Print Out the List of Courses in Alphanumeric Order

a. Sort the Course Information by Alphanumeric Course Number from Lowest to Highest

Vector:

```
void sortCourses(Vector<Course> courseList) {
    sort(courseList by course.courseNumber)
}
```

Hash Table:

```
void sortCourses(HashTable<String, Course> courseTable) {
    List<Course> courseList = courseTable.values()
    sort(courseList by course.courseNumber)
}
```

Binary Search Tree:

```
void sortCourses(Tree<Course> courseTree) {
    courseTree.inOrderTraversal()
}
```

b. Print the Sorted List to a Display

Vector:

```
void printSortedCourses(Vector<Course> courseList) {
    for (Course course : courseList):
        print course.courseNumber + ": " + course.courseTitle
}
```

Hash Table:

```
void printSortedCourses(HashTable<String, Course> courseTable) {
    List<Course> courseList = courseTable.values()
    sort(courseList by course.courseNumber)
    for (Course course : courseList):
        print course.courseNumber + ": " + course.courseTitle
}
```

Binary Search Tree (BST):

```
void printSortedCourses(Tree<Course> courseTree) {
    courseTree.inOrderTraversal()
    for (Course course : courseTree):
        print course.courseNumber + ": " + course.courseTitle
}
```

Big O Analysis

a. Analysis of Worst-Case Running Time and Memory Usage

1. **Vector:**
   - **File reading:**
     - Each line of the file is processed once to extract course data. This is an O(n) operation where n is the number of courses.
   - **Creating course objects:**
     - For each course, inserting it into the vector is O(1), so the overall complexity for n courses is O(n).
   - **Search/Print Function:**
     - Searching through the vector requires linear search (O(n)).

- **Memory Usage:**
  - Vectors require O(n) space to store all course objects.

2. **Hash Table:**
   - **File reading:**
     - Same as vector, reading the file and parsing the courses is O(n).
   - **Creating course objects:**
     - Inserting into a hash table is O(1) in the average case, making this operation O(n) overall for n courses.
   - **Search/Print Function:**
     - Searching for a specific course in a hash table is O(1) in the average case, making it efficient for lookups.
   - **Memory Usage:**
     - Hash tables require O(n) space for storing course objects but may require extra space for managing hash collisions.
3. **Binary Search Tree:**
   - **File reading:**
     - Like the vector and hash table, reading the file is O(n).
   - **Creating course objects:**
     - Inserting each course into the BST is O(log n) if the tree remains balanced, making this O(n log n) in total.
   - **Search/Print Function:**
     - Searching for a specific course is O(log n) for a balanced tree.
   - **Memory Usage:**
     - BSTs require O(n) space for storing course objects.

i. Cost per Line of Code

For simplicity, the cost per line is 1 for non-function calls. When a line calls a function (example: insert() in vector, hash table, or tree), the cost will correspond to the running time of that function:

- **Vector:**
  - Insert: O(1)
  - Search: O(n)

- **Hash Table:**
  - Insert: O(1) (average case)
  - Search: O(1) (average case)
- **BST:**
  - Insert: O(log n)
  - Search: O(log n)

ii. Assumptions

- Assume the cost for a line to execute is 1, unless it calls a function (as explained above).
- There are n courses stored in the data structure.

Data Structure	Advantages	Disadvantages
Vector	- Simple to implement and understand. - Efficient O(1) insertion at the end. - Best for storing sequential data where frequent lookups are not required.	- Inefficient O(n) search time for large datasets. - Sorting takes O(n log n), adding computational overhead. - Lack of quick lookup capabilities due to no inherent indexing structure beyond linear search.
Hash Table	- Fast average O(1) search and insert times. - Ideal for large datasets where frequent search and insertion is needed.	- Can have collisions that degrade performance to O(n) in the worst case. - Requires more memory to manage hash functions and keys. - Does not inherently store data in sorted order, so sorting requires additional
Binary Search Tree (BST)	- Data is inherently kept in sorted order, no need for additional sorting. - Efficient O(log n) search for balanced trees.	- Insertion and search efficiency can degrade to O(n) if the tree is unbalanced. - More complex to implement. - Tree rebalancing may be required to maintain O(log n) efficiency.

6. Recommendation Based on Big O Analysis

Based on the Big O analysis and the advantages/disadvantages:

- **Recommendation**: I would recommend using the Hash Table for this project:
  - **Fast Insertions and Lookups**: With average-case O(1) for both insertions and lookups, hash tables provide the fastest access time among the three data structures.
  - **Memory Usage**: Hash tables use O(n) memory, the same as vectors and trees, so there is no additional memory overhead compared to other structures.
  - **Efficient Searching**: Since you need to search for courses based on their course numbers, hash tables will be the most efficient in this case, with O(1) average-case search time.
  - **Sorted Output**: Although hash tables do not inherently maintain order, you can convert the table to a list and sort it when needed for displaying course lists. Sorting adds O(n log n), but the overall performance is still better compared to vectors for frequent lookups.

Justification: The **Hash Table** offers a good balance between speed and space efficiency for insertion and lookup operations, making it the best fit for this application where quick course lookups are required.

References:

Brass, P. (2011). *Advanced data structures*. Cambridge University Press.

Vijayalakshmi Pai, G. A. (2024). *A textbook of data structures and algorithms, Volume 3*. O'Reilly Media.