Project One

**Data Structure Psuedocode**

**Vector Data Structure**:

**File Reading and Data Loading**:

Function LoadCourseData(filename)

Open file with name filename

If file is successfully opened then

Vector<course> courses

While not end of file

Read a line from the file

Course course = ParseLine(line)

If course is not null Then

Add course to courses

End If

End While

Else

Print “Error opening file”

End If

End Function

**Line Parsing and Validation**:

Function ParseLine(line) Returns Course

Split line into tokens separated by commas

If the number of tokens is less than 2 Then

Print “Invalid input”

Return null

End If

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2 onwards]

Return new Course(courseNumber, courseTitle, prerequisites)

End Function

**Course Object Creation**:

Class Course

Properties: courseNumber, courseTitle, prerequisites

Constructor(courseNumber, courseTitle, prerequisites)

Initialize properties with provided values

End Constructor

End Class

**Printing Course Information and Prerequisites**:

Function printCourseInformation(courses, courseNumber)

For all courses

If the course.courseNumber is the same as courseNumber

Print out the course information

For each prerequisite of the course

Print the prerequisite information

End For

End If

End For

End Function

**Hash Table Data Structure**:

File Reading and Data Loading:

Function LoadCourseData(filename)

Try

Open file filename for reading

hashtable = CreateHashTable()

While not end of file

line = ReadLine(file)

Course course = ParseLine(line)

If course is not null Then

hashtable.Insert(course.courseNumber, course)

End If

End While

Catch FileError

Print "Error opening file"

Finally

Close file

End Try

End Function

**Line Parsing and Validation**:

Function ParseLine(line) Returns Course

Split line into tokens separated by commas

If the number of tokens is less than 2 Then

Print “Invalid input”

Return null

End If

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2 onwards]

Return new Course(courseNumber, courseTitle, prerequisites)

End Function

**Course Object Creation**:

Class Course

Properties: courseNumber, courseTitle, prerequisites

Constructor(courseNumber, courseTitle, prerequisites)

Initialize properties with provided values

End Constructor

End Class

**Printing Course Information and Prerequisites**:

Function PrintCourseInformation(hashtable)

For each courseNumber in hashtable.GetKeys()

Course = hashtable.Get(courseNumber)

Print "Course Number:", Course.courseNumber

Print "Course Title:", Course.courseTitle

If Course.prerequisites is not empty

Print "Prerequisites:", JoinStrings(Course.prerequisites, ", ")

End If

End For

End Function

**Tree Data Structure**:

File Reading and Data Loading:

Function LoadDataAndValidate(fileName)

tree = InitializeTree()

Open file fileName

If file is successfully opened then

While not end of file

line = ReadLine(file)

Course course = ParseLine(line)

If course is not null Then

InsertIntoTree(tree, course)

End If

End While

Else

Print "Error opening file"

End If

Close file

End Function

**Line Parsing and Validation**:

Function ParseLine(line) Returns Course

Split line into tokens separated by commas

If the number of tokens is less than 2 Then

Print “Invalid input”

Return null

End If

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2 onwards]

Return new Course(courseNumber, courseTitle, prerequisites)

End Function

**Course Object Creation**:

Class Course

Properties: courseNumber, courseTitle, prerequisites

Constructor(courseNumber, courseTitle, prerequisites)

Initialize properties with provided values

End Constructor

End

**Printing Course Information and Prerequisites**:

Function PrintCourseInformation(tree, courseNumber)

Node = FindNode(tree.root, courseNumber)

If Node is null Then

Print "Course not found"

Return

End If

Print "Course Number:", Node.course.courseNumber

Print "Name:", Node.course.name

Print "Prerequisites:"

For each prerequisite in Node.course.prerequisites

PrintPrerequisiteInfo(tree.root, prerequisite)

End For

End Function

**Menu Psuedocode**

Class Program

DataStructure courses

Function Main()

Initialize courses based on chosen data structure

Loop

DisplayMenu()

choice = GetUserChoice()

Select Case choice

Case "1"

LoadDataStructure()

Case "2"

PrintCourseList()

Case "3"

PrintCourse()

Case "4"

ExitProgram()

Case Else

Print "Invalid choice. Please try again."

End Select

End Loop

End Function

Function DisplayMenu()

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

End Function

Function GetUserChoice() Returns String

Print "Enter your choice: "

Return ReadLine()

End Function

Function LoadDataStructure()

filename = "CourseInformation.txt"

LoadCourseData(filename, courses) // Assuming LoadCourseData is defined elsewhere

Print "Data loaded successfully."

End Function

Function PrintCourseList()

If courses.IsEmpty() Then

Print "Please load data structure first."

Else

sortedCourses = courses.SortAlphanumerically()

For each course in sortedCourses

Print course.courseNumber + " - " + course.courseTitle

End For

End If

End Function

Function PrintCourse()

If courses.IsEmpty() Then

Print "Please load data structure first."

Else

Print "Enter course number: "

courseNumber = ReadLine()

course = courses.Find(courseNumber)

If course Is Null Then

Print "Course not found."

Else

Print "Course Title: " + course.courseTitle

Print "Prerequisites: " + String.Join(", ", course.prerequisites)

End If

End If

End Function

Function ExitProgram()

Print "Exiting program. Goodbye!"

Exit // Exits the loop and ends the program

End Function

End Class

Sort and Print Course List

**Vector**:

Function PrintCourseListVector(courses)

Sort courses by courseNumber using an alphanumeric comparison

For each course in courses

Print course.courseNumber + ", " + course.courseTitle

End For

End Function

**Hash Table:**

Function PrintCourseListHashTable(coursesHashTable)

courseNumbers = GetAllKeys(coursesHashTable)

Sort courseNumbers alphanumerically

For each courseNumber in courseNumbers

course = coursesHashTable.Get(courseNumber)

Print course.courseNumber + ", " + course.courseTitle

End For

End Function

**Binary Tree**:

Function PrintCourseListTree(root)

If root is not null Then

PrintCourseListTree(root.left)

Print root.course.courseNumber + ", " + root.course.courseTitle

PrintCourseListTree(root.right)

End If

End Function

**Worst Case Run-Time Analysis and Memory Usage**

**Vector**: For a vector, reading each line will cost O(1) per run. Inserting will be O(1). After loading, sorting the courses alphanumerically will be O(n log n), which is the most significant factor for a vector. For memory, each course is stored once, and the vector will grow. The memory cost is from storing data.

**Hash Table**: Inserting a course should be O(1). Sorting will be O(n log n), which is more expensive. Memory usage is higher due to the storage of keys, values, and the more complex data structure.

**Binary Search Tree**: Loading data for a tree is O(log n) for an insertion. If it is multiple, it will cost O(n log n). In-Order traversal prints all courses in sorted order and costs O(n) because each node will be visited once. Memory usage is O(n) for storing nodes, which gets more expensive than vectors because of the need to traverse so many nodes.

**Evaluation**: Vectors are simple and have efficient memory usage but have to sort when they are loaded which adds to setup. Hash tables provide fast lookup and insertion but does not maintain a order. This makes it require an extra step when sorted data is displayed. Collisions take more memory. A binary search tree has efficient lookup and insertion. It can have longer insertion times and high memory overhead if it is imbalanced heavily.

**Recommendation**: For the display course program, which will not store a lot when the program is originally created(Only 8), a tree data structure should be chosen. Due to the even amount of classes and the small amount of nodes it will be traversed quickly. Insertion times will also be fast. A vector is more simple, so if for some reason creation efficiency was a problem that could be done, but that is not the case here. A tree will work as long as it is balanced, so if many courses are added a vector could also be used. For this project, the final recommendation is a **search tree**.

The chart below shows complexity and usage when there are no exceptions such as having an unbalanced binary tree. Also assumes each line costs 1 unless a function is called.