**Vector:**

function loadCourseData(fileName)

open file with fileName // opens and checks if file is opened successfully.

If file cannot be opened

Print error message “file not found”

Return

// creates empty vector to store the data

courseData <- empty vector

// reads each line from given file

line <- read line from file

while line is not empty

// split lines by comma to get individual tokens

tokens <- split line by comma

if length of tokens < 2 // check if line has 2 parameters

print error message “error: needs at least 2 parameters.”

Continue to next line // skip to next line

courseNumber <- tokens[0]

courseTitle <- tokens[1]

prerequisites <- empty vector // creates empty vector to store prerequisites

for I from 2 length of tokens // Processing prerequisites

prerequisite <- tokens[i]

if prerequisite not gound in coursedData // check if the prerequisite exists

print error message “Prerequisite coursed not found: ” + prerequisite

else

add prerequisite to prerequisites vector

// creating new course object with extracted data

Create new Course object with courseNumber, courseTitle, and prrequisites

Add new course object to courseData vector // adds new course object to courseData

Close file

Return courseData

vector<Course> courseInfo // initialize vector to store course objects

for reach line in file

while not EOF // loop through file

for first and second value in line // extract course number and title

use push\_back to add value to courseInfo vector

if a third value exists // check for prerequisites

use push\_back to add value to prerequisites vector until new line

Print “Enter course number: ”

Input <- read user input

For each course in courseInfo vector // loop through courseInfo vector

If input is the same as course.courseNumber // check if number matches

Print “Course number: ” + course.courseNumber

Print “Course title: ” + course.courseTitle

Print “Prerequisites: ”

If prerequisites is empty

Print “none”

Else

For each prerequisite in course.prerequisites // print each prerequisite

Print prerequisite

**Hash:**

Function LoadCourseDataFromFile(filename):

// Open the file for reading

file = OpenFile(filename)

// Check if the file was opened

If file does not exist:

Return ERROR: "File not found"

// Initialize hash table to store course data

courseHashTable = InitializeHashTable()

// Read each line from the file

While not EndOfFile(file):

line = ReadLine(file)

// Split the line into course parameters

courseParameters = ParseLine(li ne)

// Check for file format errors

If Length(courseParameters) < 2:

Return ERROR: "Less than two parameters on a line"

// Extract course details from parameters

courseNumber = courseParameters[0]

courseTitle = courseParameters[1]

prerequisites = courseParameters[2:]

// Validate prerequisites

For each prerequisite in prerequisites:

If not CourseExistsInHashTable(prerequisite, courseHashTable):

Return ERROR: "Prerequisite not found"

// Create course object

newCourse = CreateCourseObject(courseNumber, courseTitle, prerequisites)

// Insert course object into hash table

InsertCourseIntoHashTable(newCourse, courseHashTable)

// Close the file

CloseFile(file)

Return courseHashTable

Procedure PrintAllCourses(courseHashTable):

// Iterate through each bucket in the hash table

For each bucket in courseHashTable:

// Check if the bucket is not empty

If not IsBucketEmpty(bucket):

// Iterate through each course in the bucket

For each course in bucket:

// Print course information

PrintCourseInformation(course)

// Print prerequisites

For each prerequisite in course.prerequisites:

PrintPrerequisiteInformation(prerequisite)

**Tree:**

Call file to open

DECLARE fileHandle: File

Try

Open fileHandle for reading at filePath

Return fileHandle

Catch any errors

Print "Error opening file"

Exit program

Procedure parseLine(line: string) returns courseInfo: Course

DECLARE courseInfo: Course

DECLARE tokens: Array of strings

tokens = split line by delimiter ','

IF length of tokens < 2 Then

Print "Error: Insufficient parameters in line"

Return null

END IF

courseInfo.courseNumber = tokens[0]

courseInfo.title = tokens[1]

IF length of tokens > 2 Then

FOR i from 2 to length of tokens - 1

prerequisite = tokens[i]

IF not exists in courseMap[prerequisite] Then

Print "Error: Prerequisite not found for course " + courseInfo.courseNumber

Return null

END IF

courseInfo.prerequisites.add(prerequisite)

End For

END IF

Return courseInfo

Procedure loadDataFromFile(filePath: string) returns root: Node

DECLARE root: Node

DECLARE fileHandle: File

DECLARE line: string

DECLARE courseInfo: Course

root = null

fileHandle = openFile(filePath)

While not endOfFile(fileHandle) Do

Read line from fileHandle

courseInfo = parseLine(line)

IF courseInfo is not null Then

root = insert(root, courseInfo) // Insert course into the tree

Else

Print "Error parsing line: " + line

END IF

END While

Close fileHandle

Return root

Procedure insert(root: Node, courseInfo: Course) returns Node

IF root is null Then

CREATE a new node with courseInfo and return

END IF

IF courseInfo.courseNumber < root.course.courseNumber Then

Set root.left = insert(root.left, courseInfo)

Else

Set root.right = insert(root.right, courseInfo)

END IF

RETURN root

Procedure printCourseInfo(root: Node)

IF root is not null Then

printCourseInfo(root.left)

Print "Course Number: " + root.course.courseNumber

Print "Title: " + root.course.title

Print "Prerequisites: " + root.course.prerequisites.toString()

printCourseInfo(root.right)

END IF

**Menu:**

Function main():

Data structure = empty\_data\_structure

WHILE true:

Display\_menu\_options()

Choice = get\_user\_choice()

If choice == 1:

Load\_data\_structure(data\_structure)

Else if choice == 2:

Print\_course\_list(data\_structure)

Else if choice == 3:

Print\_course(data\_structure)

Else if choice == 4:

Exit\_program()

Else:

Display\_invalid\_choice\_message()

Function display\_menu\_options():

Print “Menu: ”

Print “1. Load data structure”

Print “2. Print Course List”

Print “3. Print Course”

Print “4. Exit”

function get\_user\_choice():

choice = get\_user\_input("Enter your choice: ")

return choice

function load\_data\_structure(data\_structure):

file\_path = get\_user\_input("Enter file path to load data: ")

// code to read file and populate data\_structure

Print "Data loaded successfully."

function print\_course\_list(data\_structure):

// code to retrieve course list from data\_structure and sort alphabetically

sorted\_course\_list = sort\_course\_list(data\_structure.course\_list)

Print “Course List: "

for course in sorted\_course\_list:

print(course)

function print\_course(data\_structure):

course\_code = get\_user\_input("Enter course code: ")

// code to retrieve course information and prerequisites from data\_structure

course\_info = data\_structure.get\_course\_info(course\_code)

if course\_info is not None:

Print “Course Title:", course\_info.title

Print “Prerequisites:", course\_info.prerequisites

else:

Print "Course not found."

function exit\_program():

Print “Exiting program."

exit()

function display\_invalid\_choice\_message():

Print "Invalid choice. Please try again."

**Alphanumeric:**

Function PrintComputerScienceCourses(courseInfo: Vector<Course>)

// Create a sorting function to sort the courses by alphanumeric course number

Function AlphaSort(courseNumber: String)

// Create a character array from the course number string

courseNumberChars = courseNumber.toCharArray()

// Create two integers to track alphabet and number characters

alphabetIndex = 0

numberIndex = 0

// Sort the character array

For i = 0 to courseNumberChars.length - 1

If courseNumberChars[i] < '0' or courseNumberChars[i] > '9'

// Alphabetic character

alphabetIndex += 1

Else

// Numeric character

numberIndex += 1

End For

// Return the sorted string

Return new String(courseNumberChars)

End Function

// Sort the courses by alphanumeric course number

Sort courseInfo by AlphaSort(course.courseNumber)

// Print the sorted list of courses

For each course in courseInfo

Print "Course Number: " + course.courseNumber Print "Course Title: " + course.courseTitle

Print "Prerequisites: "

If course.prerequisites.size() = 0

Print "None"

Else

For each prerequisite in course.prerequisites

Print prerequisite

Print ""

End For

End Function

**Vector:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create vector** | 1 | 1 | 1 |
| **For each line in file** | 1 | n | n |
| **Create vector course item** | 1 | n | n |
| **While not EOF** | 1 | n | n |
| **For each course** | 1 | n | n |
| **Append prerequisite** | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

**Hash:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize hash table** | 1 | 1 | 1 |
| **Read each line from file** | 1 | n | n |
| **Parse each line** | 1 | n | n |
| **Check file for format errors** | 1 | n | n |
| **Extract course details** | 1 | n | n |
| **Validate prerequisites** | 1 | 1 | 1 |
| **Create course objects** | 1 | n | n |
| **Insert course object into hash table** | 1 | n | n |
| **Print all courses** | 1 | n | n |
| **For each bucket in hash table** | 1 | n | n |
| **If not empty, print prerequisite** | 1 | n | n |
| **Total Cost** | | | 10n + 1 |
| **Runtime** | | | O(n) |

**Tree:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Create token** | 1 | n | n |
| **Parse each line** | 1 | n | n |
| **Assign tokens to course info** | 1 | n | n |
| **Create new node** | 1 | n | n |
| **Load data from file** | 1 | 1 | n |
| **Insert method** | 0 | 0 | 0 |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

Advantages and disadvantages of each structure:

Vector:

Advantages – requires less memory overhead compared to the other data structures. Simple and straightforward implementation. Efficient for retrieval operations.

Disadvantages – Insertion and deletion is slower compared to the others, due to the need for shifting elements. Doesn’t maintain sorted order, additional effort is needed to achieve sorting for printing.

Hash:

Advantages – great for insertion, deletion and retrieval operations for larger datasets. Requires less memory compared to BST. Good performance for storing and retrieving when keys are distributed evenly.

Disadvantages – Not great at handling collisions, which can lead to performance degradation. Doesn’t inherently maintain sorted order, additional effort is needed to achieve sorting for printing.

Tree:

Advantages – Efficient at insertion, searching, and deletion especially for large datasets. Maintains data in sorted order. Allows easy traversal of data in sorted order, such as in-order traversal for printing.

Disadvantages – Requires more memory compared to the other data structures, May involve more complex implementation.

**My recommendation:**

My recommendation would be binary search tree, especially a balanced one offers the best efficiency for the tasks at hand. Inserting, deletion, and search operations in my analysis all have average time complexity of O(log n). BST inherently maintains sorted orders based on keys so it is perfect for printing alphanumeric order. Additionally, It provides a way to traverse the tree in-order which would result in printing the courses in sorted order without needing additional sorting operations. BST’s offer logarithmic time complexity (O(log n)) for searching for a specific course, ‘n’ being the number of nodes in the tree. Compared to hash tables, BSTs have a more predictable memory usage and do not suffer collision resolution or load factors. Vectors on the other hand do not offer efficient searching or sorting operations compared to BSTs or hash tables.