**Vector Pseudocode**

struct Course {

String courseNumber

String courseTitle

List<String> prerequisites

}

function loadDataFromFile(fileName):

open file fileName for reading

initialize an empty vector called courses

for each line in the file:

split the line by commas into tokens

if the length of tokens is less than 2:

print "Error: Line does not contain enough data"

continue

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = an empty list

if the length of tokens is greater than 2:

for each token in tokens[2:]:

prerequisites.append(token)

// Create Course object

course = new Course(courseNumber, courseTitle, prerequisites)

// Validate prerequisites

for each prerequisite in prerequisites:

if the prerequisite does not exist as a courseNumber in courses:

print "Error: Prerequisite " + prerequisite + " not found for course " + courseNumber

continue

// Add course object to courses vector

courses.append(course)

function searchCourse(Vector<Course> courses, String courseNumber):

for each course in courses:

if course.courseNumber == courseNumber:

// Print course information

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

// Print prerequisites if any

if prerequisites is not empty:

print "Prerequisites:"

for each prereq in course.prerequisites:

// Search for the prerequisite course and print information

print " - " + prereq + ": " + getCourseTitle(prereq, courses)

return

print "Error: Course not found"

function getCourseTitle(String courseNumber, Vector<Course> courses):

for each course in courses:

if course.courseNumber == courseNumber:

return course.courseTitle

return "Course Not Found"

function printCourseList(Vector<Course> courses):

// Sort courses alphanumerically by courseNumber

courses.sort\_by(courseNumber) // Assuming sorting function is available

for each course in courses:

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

function menu():

courses = empty vector

while True:

print "Menu:"

print "1. Load file data into the data structure"

print "2. Print alphanumerically ordered list of all courses"

print "3. Print course title and prerequisites for a specific course"

print "9. Exit"

option = input("Please choose an option: ")

if option == 1:

fileName = input("Enter the file name: ")

loadDataFromFile(fileName)

else if option == 2:

printCourseList(courses)

else if option == 3:

courseNumber = input("Enter the course number: ")

searchCourse(courses, courseNumber)

else if option == 9:

print "Exiting the program."

break

else:

print "Invalid option. Please try again."

**Hash Table Pseudocode**

class HashTable:

field size

field table

constructor(size):

this.size = size

this.table = new array of size

function \_hash(key):

return hash(key) % this.size

function put(key, value):

index = \_hash(key)

if table[index] is empty:

table[index] = new list

append (key, value) to table[index]

function get(key):

index = \_hash(key)

if table[index] is not empty:

for each (storedKey, storedValue) in table[index]:

if storedKey == key:

return storedValue

return null

function contains(key):

return get(key) is not null

class Course:

field courseNumber

field title

field prerequisites

constructor(courseNumber, title):

this.courseNumber = courseNumber

this.title = title

this.prerequisites = empty list

function parseFile(filename):

open file "filename" for reading

courses = new HashTable(100) // Create a new hash table for courses

for each line in file:

if line is empty:

continue

tokens = split line by commas

// Validate if at least course number and title are present

if size of tokens < 2:

print "Error: Missing course number or title"

continue

courseNumber = tokens[0]

title = tokens[1]

// Create a new Course object

course = new Course(courseNumber, title)

// If there are prerequisites, add them

for i from 2 to size of tokens - 1:

course.prerequisites.append(tokens[i])

// Store the course in the hash table

courses.put(courseNumber, course)

close file

return courses

function validatePrerequisites(courses):

for each course in courses:

courseObj = courses.get(course)

// Check if all prerequisites are valid

for each prerequisite in courseObj.prerequisites:

if not courses.contains(prerequisite):

print "Error: Prerequisite " + prerequisite + " for course " + courseObj.courseNumber + " does not exist"

function printCourseInformation(courses):

for each course in courses:

courseObj = courses.get(course)

print "Course: " + courseObj.courseNumber + " - " + courseObj.title

if courseObj.prerequisites is empty:

print " No prerequisites"

else:

print " Prerequisites:"

for each prerequisite in courseObj.prerequisites:

prerequisiteCourse = courses.get(prerequisite)

print " " + prerequisiteCourse.courseNumber + " - " + prerequisiteCourse.title

function searchCourse(courses, courseNumber):

if courses.contains(courseNumber):

courseObj = courses.get(courseNumber)

print "Course: " + courseObj.courseNumber + " - " + courseObj.title

if courseObj.prerequisites is empty:

print " No prerequisites"

else:

print " Prerequisites:"

for each prerequisite in courseObj.prerequisites:

prerequisiteCourse = courses.get(prerequisite)

print " " + prerequisiteCourse.courseNumber + " - " + prerequisiteCourse.title

else:

print "Error: Course " + courseNumber + " not found"

function sortCourses(courses):

sortedCourses = sort courses by courseNumber alphanumerically

return sortedCourses

function displayMenu():

print "1. Load data from file"

print "2. Print sorted list of courses"

print "3. Search for a specific course"

print "9. Exit"

function main():

filename = "courses.txt" // example filename

courses = empty HashTable // Initialize the courses hash table

while true:

displayMenu()

choice = input("Enter your choice: ")

if choice == 1:

courses = parseFile(filename) // Load course data from file

validatePrerequisites(courses) // Validate prerequisites for loaded courses

else if choice == 2:

sortedCourses = sortCourses(courses) // Sort courses

for each course in sortedCourses:

courseObj = courses.get(course)

print "Course: " + courseObj.courseNumber + " - " + courseObj.title

else if choice == 3:

courseNumber = input("Enter course number to search: ")

searchCourse(courses, courseNumber) // Search and print course details

else if choice == 9:

print "Exiting program."

Break

**Binary Search Tree Pseudocode**

// Course class definition

class Course:

String courseNumber

String courseTitle

List<String> prerequisites

function constructor(courseNumber, courseTitle, prerequisites):

this.courseNumber = courseNumber

this.courseTitle = courseTitle

this.prerequisites = prerequisites

// Binary Search Tree (BST) Class Definition

class BinarySearchTree:

Node root

// Function to insert a course into the BST

function insert(course):

if (root is null):

root = new Node(course) // If the root is null, create the first node

else:

insertAt(root, course) // Otherwise, insert the course at the correct position

// Function to insert a course starting at a specific node

function insertAt(node, course):

if (course.courseNumber < node.course.courseNumber):

if (node.left is null):

node.left = new Node(course) // If left child is null, insert here

else:

insertAt(node.left, course) // Otherwise, recursively insert to the left

else:

if (node.right is null):

node.right = new Node(course) // If right child is null, insert here

else:

insertAt(node.right, course) // Otherwise, recursively insert to the right

// Node class for BST

class Node:

Course course

Node left

Node right

function constructor(course):

this.course = course

this.left = null // Left child is initially null

this.right = null // Right child is initially null

// Function to print all course information in sorted order (in-order traversal)

function printCourses(courses):

if (courses is not null):

inOrderTraversal(courses.root) // Perform an in-order traversal starting from the root

// In-order traversal function to visit all nodes (courses) in sorted order

function inOrderTraversal(node):

if (node is not null):

inOrderTraversal(node.left) // Visit the left subtree

printCourse(node.course) // Print the current node's course

inOrderTraversal(node.right) // Visit the right subtree

// Function to print the details of a course

function printCourse(course):

print("Course Number:", course.courseNumber)

print("Course Title:", course.courseTitle)

if (course.prerequisites is not empty):

print("Prerequisites:", join(course.prerequisites, ", ")) // Print the prerequisites as a comma-separated list

else:

print("No prerequisites.") // Print if there are no prerequisites

// Function to search for a specific course by its course number

function searchCourse(courses, courseNumber):

course = searchInBST(courses, courseNumber)

if (course != null):

printCourse(course) // If course found, print its details

else:

print("Error: Course not found.") // If course not found, display error message

// Function to search for a course in the BST starting from the root node

function searchInBST(courses, courseNumber):

return searchNode(courses.root, courseNumber) // Perform the search in the BST

// Recursive function to search for a course starting from a specific node

function searchNode(node, courseNumber):

if (node is null):

return null // If node is null, return null indicating course not found

if (courseNumber == node.course.courseNumber):

return node.course // If the course number matches, return the course

if (courseNumber < node.course.courseNumber):

return searchNode(node.left, courseNumber) // If course number is smaller, search left subtree

else:

return searchNode(node.right, courseNumber) // If course number is larger, search right subtree

// Function to load course data from a file

function loadCourseData(fileName):

// Open the file in read mode

file = open(fileName, "r")

// Initialize an empty binary search tree to store courses

courses = new BinarySearchTree<Course>()

// Initialize an empty hash table to keep track of course numbers for validation

courseNumbers = new HashTable<String, Boolean>()

// Read each line from the file

while (line = readLine(file)) is not null:

// Parse the line into tokens based on a delimiter (e.g., commas)

tokens = split(line, ",")

// Validate if the line has at least two parameters (course number and title)

if (length(tokens) < 2):

print("Error: Course information is incomplete on line:", line)

continue

// Extract course number, title, and prerequisites

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = extractPrerequisites(tokens[2:]) // Handle empty prerequisites as well

// Validate if prerequisites are valid courses

validPrerequisites = true

for prerequisite in prerequisites:

if (prerequisite not in courseNumbers):

print("Error: Invalid prerequisite course number:", prerequisite)

validPrerequisites = false

break

// If prerequisites are valid, create course object and add to tree

if (validPrerequisites):

course = new Course(courseNumber, courseTitle, prerequisites)

courses.insert(course)

courseNumbers[courseNumber] = true

// Close the file after reading

close(file)

return courses

// Function to extract prerequisites from the tokens (handles empty prerequisites)

function extractPrerequisites(tokens):

prerequisites = []

for token in tokens:

if (token is not empty):

prerequisites.append(token)

return prerequisites

// Menu for interacting with the program

function displayMenu():

print("Menu:")

print("1. Load course data from file")

print("2. Print all courses in alphanumeric order")

print("3. Search for a course by course number")

print("9. Exit")

choice = getUserChoice()

if choice == 1:

fileName = getFileNameFromUser()

courses = loadCourseData(fileName)

print("Courses loaded successfully.")

else if choice == 2:

if (courses != null):

printCourses(courses)

else:

print("Error: Courses not loaded.")

else if choice == 3:

courseNumber = getCourseNumberFromUser()

searchCourse(courses, courseNumber)

else if choice == 9:

print("Exiting program.")

exit()

else:

print("Invalid choice. Please try again.")

// Recurse to show menu again

displayMenu()

// Function to get user choice for menu

function getUserChoice():

choice = getUserInput() // Get input from user

return choice

// Function to get file name input from user

function getFileNameFromUser():

fileName = getUserInput() // Get file name from user

return fileName

// Function to get course number input from user

function getCourseNumberFromUser():

courseNumber = getUserInput() // Get course number from user

return courseNumber

// Main program

function main():

// Start the menu system

displayMenu()

**Runtime Analysis**

Vector

|  |  |  |  |
| --- | --- | --- | --- |
| Section of Code | Line Cost | # Times Executed | Total Cost |
| Open file for reading and initialize empty vector | O(1) | 1 | O(1) |
| Iterate over each line, spilt tokens, and check validity | O(1) | N | O(n) |
| Extract course info and initialize prerequisite list | O(1) | N | O(n) |
| Handle prerequisite tokens and create course object | O(1) | N | O(n) |
| Validate prerequisites by searching for them in courses vector | O(n) | N | O(n^2) |
|  |  | Total Cost | O(n^2) |

Hash Table

|  |  |  |  |
| --- | --- | --- | --- |
| Section of Code | Line Cost | # Times Executed | Total Cost |
| Parse file and load courses | O(1) | n | O(n) |
| For each course, check if prerequisites are valid | O(1) | N | O(n) |
| Check if prerequisites exists are valid. | O(1) | N | O(n) |
| Create course objects | O(1) | N | O(n) |
|  |  | Total Cost | O(n) |

Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| Section of code | Line Cost | # Times Executed | Total Cost |
| Open the file | O(1) | 1 | O(1) |
| Initialize the binary search tree | O(1) | 1 | O(1) |
| Initialize the hash table | O(1) | n | O(n) |
| Read each line from the file | O(1) | n | O(n) |
| Parse the line | O(1) | n | O(n) |
| Validate prerequisites | O(n) | n | O(n^2) |
| Insert course into data structure | O(log n) | n | O(n log n) |
|  |  | Total Cost | O(n^2 + n log n) |

**Evaluation of Data Structures**

The vector-based approach is straightforward to implement, making it an attractive option for smaller datasets. This method involves reading a file, iterating through each line, and splitting tokens into their respective components. However, a significant drawback of this approach is how it handles prerequisites. For each course, the program must search through the entire list of courses to confirm whether its prerequisites exist. This results in significant inefficiencies as the number of courses increases, especially since the search is repeated for every course. Consequently, this method becomes slower and more cumbersome as the dataset grows.

In contrast, the hash table-based approach provides substantial improvements in efficiency. Hash tables store course information in a manner that allows for quick lookups, enabling the program to check if a prerequisite exists swiftly. This leads to faster processing since prerequisite validation does not require a complete search of the dataset each time. The overall process remains efficient, even with an increasing number of courses, making the hash table a highly scalable solution. This is particularly beneficial when frequent lookups are necessary, such as when validating prerequisites.

The Binary Search Tree (BST) approach has its advantages, particularly in maintaining a sorted structure of data. While inserting courses into the BST is relatively fast, validating prerequisites can still be slow. Each lookup necessitates traversing the tree to find the corresponding course, and this search becomes time-consuming as the number of courses increases, rendering it less efficient than the hash table. Although the BST offers benefits like sorting, its inefficiency in prerequisite validation makes it less suitable for this scenario, where quick validation is critical.

**Recommendation**

Based on the analysis of these three data structures, I recommend using the hash table to manage and validate courses and their prerequisites. The primary advantage of the hash table is its ability to deliver quick lookups, which makes prerequisite validation significantly faster compared to the vector approach, where the program must search through all courses for each prerequisite. While the BST has sorting advantages, its performance in prerequisite validation does not match the speed of a hash table, particularly as the number of courses increases.

The hash table strikes the best balance between simplicity and efficiency. Its capability to quickly check the existence of a prerequisite in the course list ensures that the process remains fast and scalable, even with a large number of courses. This makes the hash table the most practical choice for handling prerequisite validation, as it minimizes the need for slow, repetitive searches and ensures smoother overall performance.

In conclusion, while both the vector and BST approaches have their own merits, the hash table excels in performance for this specific task. Its rapid lookup capabilities make it the optimal data structure for efficiently managing and validating course prerequisites, especially when dealing with large datasets where efficiency is a top priority.