Pseudocode

**Vector**:

FUNCTION loadCourses(fileName)

OPEN fileName FOR reading

IF file cannot be opened

PRINT "Error: Cannot open file"

RETURN

INITIALIZE vector courses

WHILE NOT end of file

READ line

PARSE line INTO courseNumber, courseTitle, prerequisites

IF courseNumber IS NULL OR courseTitle IS NULL

PRINT "Error: Invalid course format"

CONTINUE

FOR EACH prerequisite IN prerequisites

IF prerequisite NOT IN course numbers

PRINT "Error: Prerequisite course not found - " + prerequisite

CONTINUE

CREATE course OBJECT

SET course.number = courseNumber

SET course.title = courseTitle

SET course.prerequisites = prerequisites

ADD course TO courses

CLOSE file

RETURN courses

Now that we have the code up and running and it can read the file we will move on to creating and storing the course.

STRUCT Course

STRING number

STRING title

VECTOR prerequisites

FUNCTION createCourse(courseNumber, courseTitle, prerequisites)

INITIALIZE Course course

SET course.number = courseNumber

SET course.title = courseTitle

SET course.prerequisites = prerequisites

RETURN course

Then we would create a way to search the courses. Since we are in the mindset this isn’t the only class in our ABC University we need to make it easier to find the class and give it a course number and add a null if there isn’t any. We will also let it print the course details in it.

FUNCTION searchCourse(courses, courseNumber)

FOR EACH course IN courses

IF course.number EQUALS courseNumber

RETURN course

RETURN NULL

FUNCTION printCourseInfo(course)

IF course IS NULL

PRINT "Course not found"

RETURN

PRINT "Course Number: " + course.number

PRINT "Course Title: " + course.title

PRINT "Prerequisites: "

IF course.prerequisites IS EMPTY

PRINT "None"

ELSE

FOR EACH prerequisite IN course.prerequisites

PRINT prerequisite

Lastly we are using one of the classes that was in the ABC University Pdf as the example.

courses = loadCourses("courses.txt")

courseNumberToSearch = "CSCI300"

course = searchCourse(courses, courseNumberToSearch)

printCourseInfo(course)

**Hash Table:**

Define struct Course

courseNumber: string

title: string

prerequisites: list of strings

End struct

Next im going to Open and Read file

Function loadCoursesFromFile(filePath: string)

Declare file as File

Open file at filePath

If file cannot be opened

Print "Error: File cannot be opened"

Return

End If

Next im going to Check for format errors and check for Parse for Each line

Declare hashTable as HashTable

Declare allCourses as Dictionary

While not end of file

Read line from file

Split line by commas into tokens

If number of tokens < 2

Print "Error: Incorrect format in line: " + line

Continue

End If

Declare course as Course

course.courseNumber = tokens[0]

course.title = tokens[1]

For i = 2 to length of tokens - 1

If tokens[i] is not in allCourses

Print "Error: Prerequisite " + tokens[i] + " does not exist for course " + course.courseNumber

Else

Add tokens[i] to course.prerequisites

End If

End For

Add course to allCourses with key course.courseNumber

Call hashTable.insert(course)

End While

Close file

End Function

Next im going to create my course object and store in the Hash Table

Function insert(hashTable: HashTable, course: Course)

key = hashFunction(course.courseNumber)

If hashTable[key] is empty

hashTable[key] = new list

End If

Add course to hashTable[key]

End Function

Lastly im going to print the course information and prerequisites

Function printAllCourses(hashTable: HashTable)

For each key in hashTable

For each course in hashTable[key]

Print "Course Number: " + course.courseNumber

Print "Title: " + course.title

If length of course.prerequisites > 0

Print "Prerequisites: " + join(course.prerequisites, ", ")

Else

Print "Prerequisites: None"

End If

End For

End For

End Function

**Binary Search Tree:**

function loadCourseData(filePath):

# Open the file

file = open(filePath, "r")

# Initialize a list to store all courses

allCourses = []

# Read each line in the file

for line in file:

# Split the line by commas

courseData = line.strip().split(',')

# Check if the line has at least 2 elements (courseNumber and name)

if length(courseData) < 2:

print("Error: Line does not contain at least a course number and name.")

continue

# Extract courseNumber and name

courseNumber = courseData[0]

courseName = courseData[1]

# Extract prerequisites if any

prerequisites = []

if length(courseData) > 2:

prerequisites = courseData[2:]

# Create a course object

course = Course(courseNumber, courseName, prerequisites)

# Add the course to the list

allCourses.append(course)

# Close the file

file.close()

# Validate the courses and prerequisites

if not validateCourses(allCourses):

print("Error: Validation failed.")

return None

# Insert courses into the binary search tree

bst = BinarySearchTree()

for course in allCourses:

bst.insert(course)

return bst

Once we got that done we need to validate the courses as well as the prerequisites.

function validateCourses(courses):

# Create a set of courseNumbers

courseNumbers = set()

for course in courses:

courseNumbers.add(course.courseNumber)

# Check if each prerequisite exists as a course

for course in courses:

for prerequisite in course.prerequisites:

if prerequisite not in courseNumbers:

print("Error: Prerequisite", prerequisite, "for course", course.courseNumber, "does not exist.")

return False

return True

Next I would actually create the course object for the binary tree

class Course:

def \_\_init\_\_(self, courseNumber, name, prerequisites):

self.courseNumber = courseNumber

self.name = name

self.prerequisites = prerequisites

class TreeNode:

def \_\_init\_\_(self, course):

self.course = course

self.left = None

self.right = None

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, course):

if self.root is None:

self.root = TreeNode(course)

else:

self.\_insert(self.root, course)

def \_insert(self, node, course):

if course.courseNumber < node.course.courseNumber:

if node.left is None:

node.left = TreeNode(course)

else:

self.\_insert(node.left, course)

else:

if node.right is None:

node.right = TreeNode(course)

else:

self.\_insert(node.right, course)

def printTree(self):

self.\_printTree(self.root)

def \_printTree(self, node):

if node is not None:

self.\_printTree(node.left)

print("Course Number:", node.course.courseNumber, "Course Name:", node.course.name, "Prerequisites:", node.course.prerequisites)

self.\_printTree(node.right)

Then I would print the course information as well as the prerequisites.

function printCourseInformation(bst):

bst.printTree()

Lastly I would show you what example I used

filePath = "path/to/course\_data.txt"

bst = loadCourseData(filePath)

if bst is not None:

printCourseInformation(bst)

**Menu:**

function displayMenu():

print("Menu:")

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

print("2. Print all courses (alphanumerically ordered)")

print("3. Print course information")

print("9. Exit")

function main():

bst = None

while True:

displayMenu()

choice = input("Enter your choice: ")

if choice == "1":

filePath = input("Enter the path to the course data file: ")

bst = loadCourseData(filePath)

if bst is not None:

print("Course data loaded successfully.")

else:

print("Failed to load course data.")

elif choice == "2":

if bst is None:

print("Error: Course data not loaded. Please load the data first.")

else:

printCourseInformation(bst)

elif choice == "3":

if bst is None:

print("Error: Course data not loaded. Please load the data first.")

else:

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

printIndividualCourse(bst, courseNumber)

elif choice == "9":

print("Exiting the program.")

break

else:

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

**Print Sorted Information:**

function printCoursesVector(courseVector):

# Sort the vector by course number alphanumerically

courseVector.sort(key=lambda course: course.courseNumber)

# Print the sorted courses

for course in courseVector:

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

function printCoursesHashTable(courseHashTable):

# Extract course numbers and sort them

sortedCourseNumbers = sort(courseHashTable.keys())

# Print the courses in sorted order

for courseNumber in sortedCourseNumbers:

course = courseHashTable[courseNumber]

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

function printCoursesBST(bst):

bst.printTreeAlphanumeric()

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, course):

if self.root is None:

self.root = TreeNode(course)

else:

self.\_insert(self.root, course)

def \_insert(self, node, course):

if course.courseNumber < node.course.courseNumber:

if node.left is None:

node.left = TreeNode(course)

else:

self.\_insert(node.left, course)

else:

if node.right is None:

node.right = TreeNode(course)

else:

self.\_insert(node.right, course)

def printTreeAlphanumeric(self):

self.\_printTreeAlphanumeric(self.root)

def \_printTreeAlphanumeric(self, node):

if node is not None:

self.\_printTreeAlphanumeric(node.left)

print("Course Number:", node.course.courseNumber, "Course Name:", node.course.name)

self.\_printTreeAlphanumeric(node.right)

Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| file = open(filePath, "r") | 1 | 1 | 1 |
| courseVector = [] | 1 | 1 | 1 |
| for line in file | 1 | N | N |
| courseData = line.strip().split(',') | 1 | N | N |
| if length(courseData) < 2 | 1 | N | N |
| print("Error: Line does not contain at least a course number and name.") | 1 | Rare | Ignored |
| prerequisites = [] | 1 | N | n |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hashtable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| file = open(filePath, "r") | 1 | 1 | 1 |
| courseHashTable = {} | 1 | 1 | 1 |
| for line in file | 1 | N | N |
| courseData = line.strip().split(',') | 1 | N | N |
| if length(courseData) < 2 | 1 | N | N |
| print("Error: Line does not contain at least a course number and name.") | 1 | Rare | ignored |
| prerequisites = [] | 1 | N | n |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Search Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| file = open(filePath, "r") | 1 | 1 | 1 |
| allCourses = [] | 1 | 1 | 1 |
| for line in file | 1 | N | N |
| courseData = line.strip().split(',') | 1 | N | N |
| if length(courseData) < 2 | 1 | N | N |
| print("Error: Line does not contain at least a course number and name.") | 1 | Rare | Ignored |
| prerequisites = [] | 1 | N | n |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Cost** | **Runtime** | **Memory** |
| **Vector** | 5n+5 | 0(n) | 0(n) |
| **Hash Table** | 6n + 5 | 0(n) | 0(n) |
| **Binary Search Tree** | 8n + 7 (avg), n^2 + n + 7 (worst) | 0(n) (avg) 0(n^2) (worst) | 0(n) |

Each data structure has there advantages and disadvantages. For Vector there advantages its pretty easy to sort and have fast iteration. As for the disadvantages is has a fixed size or has slow deletions. For Hash table there advantages is they have efficient use of memory and for there disadvantages they have unordered data. Lastly for Binary Search Tree there advantages are ordered data and for there disadvantages is complex implementation. For me I would recommend using Binary Search Tree for the Big O. It does really well on maintain order, efficiency and printing.