Pravishna Nand

CS-300

12/3/23

6-2 Submit Project One

**# Pseudocode for loading data into the vector data structure**

# Define a data structure to store course information

data\_structure = Vector()

# Define a function to open the file, read, parse, and validate data

function load\_course\_data(file\_path):

try:

# Open the file for reading

file = open(file\_path, 'r')

# Initialize variables for error checking

line\_number = 0

prerequisites\_exist = Set() # A set to track existing prerequisites

# Read each line in the file

for line in file:

line\_number += 1 # Increment line number

# Split the line into tokens using a delimiter (e.g., comma)

tokens = split\_line(line)

# Check if the line has at least two parameters

if length(tokens) >= 2:

course\_number = tokens[0]

course\_title = tokens[1]

prerequisites = tokens[2:] if length(tokens) > 2 else [] # Extract prerequisites

# Validate prerequisites

for prerequisite in prerequisites:

if prerequisite not in prerequisites\_exist:

# Handle error: Prerequisite not found

print(f"Error: Prerequisite {prerequisite} on line {line\_number} not found.")

close\_file(file)

return

# Create a course object and add it to the data structure

course = create\_course(course\_number, course\_title, prerequisites)

append(data\_structure, course)

# Update prerequisites\_exist set

for prerequisite in prerequisites:

prerequisites\_exist.add(prerequisite)

else:

# Handle error: Invalid line format

print(f"Error: Invalid line format on line {line\_number}.")

close\_file(file)

return

# Close the file

close\_file(file)

except FileNotFoundError:

# Handle error: File not found

print("Error: File not found.")

except IOError:

# Handle error: Input/output error

print("Error: Input/output error.")

# Define a function to split a line into tokens

function split\_line(line):

# Split the line into tokens using a delimiter (e.g., comma)

tokens = line.split(',')

return tokens

# Define a function to create a course object

function create\_course(course\_number, course\_title, prerequisites):

course = {}

course['number'] = course\_number

course['title'] = course\_title

course['prerequisites'] = prerequisites # Store prerequisites as a list

return course

# Define a function to search and print course information

function print\_course\_info(course\_number\_to\_search):

found = False

# Iterate through the courses in the data structure

for course in data\_structure:

if course['number'] == course\_number\_to\_search:

found = True

print("Course Number:", course['number'])

print("Course Title:", course['title'])

if length(course['prerequisites']) > 0:

print("Prerequisites:")

for prerequisite in course['prerequisites']:

print("- ", prerequisite)

else:

print("No prerequisites for this course.")

if not found:

# Handle error: Course not found

print("Error: Course not found.")

# Main program

function main():

file\_path = "course\_data.txt" # Replace with the actual file path

load\_course\_data(file\_path)

# Test the print\_course\_info function

course\_number\_to\_search = "CS101" # Replace with the desired course number

print\_course\_info(course\_number\_to\_search)

# Call the main function to start the program

main()

**# Pseudocode for a Hash Table for Storing Course Information**

# Define the structure for a course

Structure Course:

courseNumber: String

courseTitle: String

prerequisites: List of Strings

# Define the structure for a hash table node

Structure HashNode:

key: String (courseNumber)

value: Course

next: HashNode

# Define the HashTable class

Class HashTable:

Initialize a hash table data structure:

Create an array of nodes (buckets) to represent the hash table.

Initialize the size of the hash table (tableSize).

Function to calculate the hash value for a key (courseNumber):

Input: key (String)

Output: hashValue (Integer)

Implement a hash function to convert the key into a unique hash value.

Ensure the hash value falls within the range [0, tableSize - 1].

Function to insert a course into the hash table:

Input: course (Course)

Calculate the hash value for the courseNumber using the hash function.

Create a new HashNode with the key (courseNumber) and value (course).

If the bucket at the calculated hash value is empty:

Set the new node as the head of the linked list in the bucket.

Else:

Traverse the linked list to check for an existing node with the same key (courseNumber).

If found, update the value (course) of the existing node.

If not found, append the new node to the end of the linked list.

Function to search for a course by courseNumber:

Input: courseNumber (String)

Output: course (Course)

Calculate the hash value for the given courseNumber.

Search for the course within the linked list at the calculated hash value.

If found, return the course.

If not found, return None.

Function to remove a course by courseNumber:

Input: courseNumber (String)

Calculate the hash value for the given courseNumber.

Search for the course within the linked list at the calculated hash value.

If found, remove the course (node) from the linked list.

Ensure proper memory management by deallocating the removed node.

Function to print all courses in the hash table:

Loop through each index (bucket) in the hash table:

Traverse the linked list in each bucket and print course details.

# Main program execution:

Initialize a hash table using the HashTable class.

Display a menu with options to:

Load course data from a CSV file into the hash table.

Display all courses in the hash table.

Find a course by courseNumber and display its details.

Remove a course by courseNumber.

Exit the program.

Handle user input and execute the corresponding functionality.

Ensure proper error handling for various scenarios.

**# Pseudocode for a Tree**

Define a structure for a course with fields: courseNumber, name, and prerequisites (e.g., Course).

Define a binary tree data structure to store course objects (e.g., BinaryTree).

Define a function to validate course data:

Function ValidateCourseData(courseData):

Split courseData by ',' into tokens

If the number of tokens < 2:

Display "Error: Insufficient data for a course."

Return false

End If

courseNumber <- tokens[0]

name <- tokens[1]

If the number of tokens > 2:

prerequisites <- tokens[2 to end]

For each prerequisite in prerequisites:

If no course exists with prerequisite as courseNumber:

Display "Error: Prerequisite not found for", courseNumber

Return false

End If

End For

End If

Return true

Function CreateCourse(courseData):

Split courseData by ',' into tokens

courseNumber <- tokens[0]

name <- tokens[1]

If the number of tokens > 2:

prerequisites <- tokens[2 to end]

Else:

prerequisites <- empty list

End If

course <- Course(courseNumber, name, prerequisites)

Return course

Define a function to load course data from a file:

Function LoadCourseDataFromFile(filename, tree):

Try to open the file at filename

If file cannot be opened:

Display "Error: File not found."

Return

End Try

For each line in the file:

If line is empty:

Continue

End If

If not ValidateCourseData(line):

Display "Skipping invalid course data:", line

Continue

End If

course <- CreateCourse(line)

tree.Insert(course)

End For

Close the file

Function PrintCourseAndPrerequisites(course):

Display "Course Number:", course.courseNumber

Display "Name:", course.name

If course.prerequisites is not empty:

Display "Prerequisites:"

For each prerequisite in course.prerequisites:

Display prerequisite.courseNumber, prerequisite.name

Call PrintCourseAndPrerequisites(prerequisite)

End For

End If

Define a function to print course data:

Function PrintCourseData(tree):

For each course in tree (e.g., using in-order traversal):

Call PrintCourseAndPrerequisites(course)

Initialize an empty binary tree (e.g., binarySearchTree) to store course objects.

Call LoadCourseDataFromFile("CourseInformation.csv", binarySearchTree) to load data from the file "CourseInformation.csv" into the binarySearchTree.

Call PrintCourseData(binarySearchTree) to print course information and prerequisites.

**# Pseudocode for a Menu**

# Define a function to display the menu options and handle user input

Function DisplayMenu():

Repeat until user selects 'Exit':

Display Menu Options:

1. Load Data Structure

2. Print Course List

3. Print Course

4. Exit

Prompt user for choice (1, 2, 3, or 4)

# Use a switch or if-else statements to perform actions based on user choice

If user choice is 1:

Call LoadDataStructure()

ElseIf user choice is 2:

Call PrintCourseList()

ElseIf user choice is 3:

Prompt user for the course number to print

Call PrintCourse(courseNumber)

ElseIf user choice is 4:

Display "Exiting the program."

Else:

Display "Invalid choice. Please select a valid option."

# Define a function to load data into the data structure

Function LoadDataStructure():

# Implement logic to load data from a file into the data structure

# Ensure that data is properly validated and loaded

# Define a function to print an alphanumerically ordered list of all courses in the Computer Science department

Function PrintCourseList():

# Implement logic to retrieve and print the course list in sorted order

# Define a function to print the course title and prerequisites for an individual course

Function PrintCourse(courseNumber):

# Implement logic to search for the course by course number

# Print the course title and prerequisites if found

# Handle errors if the course is not found

# Main program execution

Display "Welcome to the Course Information Program!"

DisplayMenu() # Call the menu function to start the program

# Ensure proper error handling and validation in each function.

**# Pseudocode to Print Courses in Alphanumeric Order**

# Define a function to print the list of Computer Science courses in alphanumeric order using a vector

Function PrintCoursesInVector(vector\_data\_structure):

# Sort the vector by courseNumber in alphanumeric order

Sort vector\_data\_structure by courseNumber

# Print the sorted list of courses

For each course in vector\_data\_structure:

Print "Course Number:", course.courseNumber

Print "Course Title:", course.courseTitle

Print "Prerequisites:", course.prerequisites

End For

# Define a function to print the list of Computer Science courses in alphanumeric order using a hash table

Function PrintCoursesInHashTable(hash\_table\_data\_structure):

# Create an empty list to store courses

course\_list = []

# Iterate through the hash table

For each bucket in hash\_table\_data\_structure:

For each node in bucket:

Append node.value (course) to course\_list

End For

End For

# Sort the course\_list by courseNumber in alphanumeric order

Sort course\_list by courseNumber

# Print the sorted list of courses

For each course in course\_list:

Print "Course Number:", course.courseNumber

Print "Course Title:", course.courseTitle

Print "Prerequisites:", course.prerequisites

End For

# Define a function to print the list of Computer Science courses in alphanumeric order using a binary tree

Function PrintCoursesInTree(binary\_tree\_data\_structure):

# Create an empty list to store courses

course\_list = []

# Traverse the binary tree in-order to populate course\_list

InOrderTraversal(binary\_tree\_data\_structure.root, course\_list)

# Sort the course\_list by courseNumber in alphanumeric order

Sort course\_list by courseNumber

# Print the sorted list of courses

For each course in course\_list:

Print "Course Number:", course.courseNumber

Print "Course Title:", course.courseTitle

Print "Prerequisites:", course.prerequisites

End For

# Main program execution

# You would call the respective function based on the data structure you are using:

# For vector: PrintCoursesInVector(vector\_data\_structure)

# For hash table: PrintCoursesInHashTable(hash\_table\_data\_structure)

# For binary tree: PrintCoursesInTree(binary\_tree\_data\_structure)

For Vector:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | m | n\*m |
| **Total Cost** | | | n+1+n\*m |
| **Runtime** | | | O(n\*m) |

For Hash:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | m | m |
| **print the prerequisite course information** | 1 | m | M^2 |
| **Total Cost** | | | N+m^2+m+3 |
| **Runtime** | | | O(n+m^2) |

For Tree:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | log(n) | log(n) |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | m | m |
| **print the prerequisite course information** | 1 | m | M^2 |
| **Total Cost** | | | n+m^2+m+log(n)+3 |
| **Runtime** | | | 0(n+m^2+log(n)) |

In analyzing the suitability of data structures for the advisor's requirements, each structure presents distinct advantages and disadvantages. The Vector data structure offers simplicity in terms of data storage and retrieval, making it easy to load and print courses. However, its linear search for courses and prerequisites results in a runtime complexity of O(n \* m), where n is the number of courses and m is the average number of prerequisites. The Hash Table, on the other hand, excels in search operations, achieving O(n + m^2) complexity, thanks to its efficient key-based retrieval. However, it requires more memory due to its hash buckets. Lastly, the Tree data structure's binary search tree structure provides efficient search operations with a runtime complexity of O(n + m^2 + log(n)). Yet, its memory requirements and complex traversal make it slightly less straightforward to implement. Thus, the choice of data structure should consider the trade-offs between memory usage and runtime efficiency based on the specific needs of the advising program.

Based on the Big O analysis and a thorough evaluation of the three data structures—Vector, Hash Table, and Tree—I would recommend using the Hash Table as the most suitable data structure for the advising program. The Hash Table offers a favorable balance between runtime efficiency and memory usage, with a runtime complexity of O(n + m^2), where n is the number of courses and m is the average number of prerequisites. This complexity ensures that search operations are efficient and scale well with increasing data size. Additionally, the Hash Table provides the advantage of key-based retrieval, reducing the need for linear searches and improving overall program responsiveness. While the Vector and Tree structures have their merits, the Hash Table's performance characteristics align more closely with the program's requirements, making it the recommended choice for storing and retrieving course information efficiently.