Gavin Bliss

CS 300

6-2 Project One

04/03/2024

**Vector Data Structure Pseudocode:**

// Define a course class that stores course data

class Course

properties:

courseNumber: String

title: String

prereq: Vector<String>

constructor(courseNumber, title, prereq)

this.courseNumber = courseNumber

this.title = title

this.prereq = prereq

// Function for loading courses from file

Function loadCourseFromFile(filePath) returns Vector<Course>

Initialize courseVector as Vector<Course>

Initialize fileStream to open filePath

If fileStream is invalid

Print “Unable to open file.”

Return courseVector

While not the end of fileStream

Read line from fileStream

Parse line to courseNumber, title, and prerequisites

If courseNumber or title is missing

Print “Missing course number/title”

Proceed to next iteration

Initialize prereqVector as Vector<String>

If there are prerequisites

For each prerequisite in prereq

If prereq exists in courseVector

Add prereq to prereqvector

Else

Print “Prereq course not found.”

Create a new course object with courseNumber, title, and prereqVector

Add the course object to courseVector

Close the fileStream

Return courseVector

// Function to print course info

Function printCourseInfo(courseVector, courseNumber

For each course in courseVector

If course.courseNumber is equal to courseNumber

Print “Course Number: “ + course.courseNumber

Print “Title: “ + course.title

If course.prereq is not empty

Print “Prerequisites:”

For each prerequisite in course.prereq

Print prereq

Else

Print “There are no prerequisites for this course.”

return

Print “Course not found.”

// Main program

filePath = “path/course/courseInfo.txt”

coursevector = loadCoursesFromFile(filePath)

// Example

courseNumber = “CS300”

printCourseInfo(courseVector, courseNumber)

**Hash Table Data Structure Pseudocode:**

// open & read file

Procedure openAndReadFile(filename)

Try to open filename

If file not found

Print “Error: File not found.”

End program

endIf

While not end of file

Read a line

If line validated

course = parseLine(line)

addCourseToHashTable(course)

endIf

endWhile

endProcedure

//validate line format

Function validateLine(line)

Split line with commas into elements

If number of elements is < 2  
 return false

endIf

For each element in prerequisites of line

If element is not in courseNumbers

return false

endIf

Next

return true

endFunction

// parse line and create course object

Function parseLine(line)

Split line by comma into courseNumber, courseTitle, prerequisite

Create new course object with courseNumber, courseTitle, prerequisite

return course object

endFunction

// add course to hash table

Procedure addCourseToHash(course)

If hashTable does not contain course.courseNumber

add course.courseNumber as key and course as value to hashTable

endIf

endProcedure

//printing course info and prerequisites

Procedure printCourseInfo(hashTable)

For each courseNumber in hashTable keys

Course = hashTable[courseNumber]

Print course.courseNumber and course.courseTitle

If course.prerequisites is not empty

Print “Course prerequisites are: “

For each prerequisite in course.prerequisites

Print prerequisite

Next

Else

Print “Course has no prerequisites.”

endIf

Next

endProcedure

**Binary Tree Data Structure Pseudocode:**  
// Initialize a tree data structure for storing courses

Tree courseTree

// Initialize a dictionary for quick access to course objects by courseNumber

Dictionary courseDictionary

// Open & Read File

Procedure OpenAndReadFile(fileName)

Try

Open fileName for reading

If file is successfully opened

Read file line by line

For each line, call ProcessLine(line)

End If

Else

Print "File cannot be accessed."

Catch any exceptions

Print "Error occurred reading the file."

End Try

ValidatePrerequisites() // Ensure all prerequisites are valid now that courses are loaded

End Procedure

// Parse & Validate File Data

Procedure ProcessLine(line)

Split line into tokens separated by commas

If number of tokens < 2

Print "Error: each line must contain at least a course number and course title."

Return

End If

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens from index 2 to end if any

// Temporarily store course data

TemporarilyStoreCourse(courseNumber, courseTitle, prerequisites)

End Procedure

Procedure TemporarilyStoreCourse(courseNumber, courseTitle, prerequisites)

course = new Course(courseNumber, courseTitle)

// Initially, just store the prerequisites list as it is; validation will occur later

course.prerequisites = prerequisites

courseDictionary.add(courseNumber, course)

End Procedure

// Validate prerequisites after courses have been loaded

Procedure ValidatePrerequisites()

For each course in courseDictionary

validatedPrerequisites = []

For each prerequisite in course.prerequisites

If prerequisite is not in courseDictionary

Print "Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist."

Else

validatedPrerequisites.add(prerequisite)

End If

Next

course.prerequisites = validatedPrerequisites // Update with validated prerequisites

InsertIntoTree(course) // Insert the fully validated course into tree

Next

End Procedure

// Method to insert objects into a tree data structure is available

Procedure InsertIntoTree(course)

courseTree.insert(course.courseNumber, course)

End Procedure

// Create Course Objects & Store Them in Tree Data Structure

Class Course

Properties: courseNumber, courseTitle, prerequisites (initially empty list)

Constructor(courseNumber, courseTitle)

this.courseNumber = courseNumber

this.courseTitle = courseTitle

End Constructor

End Class

// Print Course Info from Tree

Procedure PrintCourseInfo(node)

// This recursive function performs an in-order traversal of the binary search tree

If node is not null

PrintCourseInfo(node.left)

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

Print "Title: " + node.course.courseTitle

If node.course.prerequisites is not empty

Print "Prerequisites: " + Join node.course.prerequisites with ", "

Else

Print "Prerequisites: None"

End If

PrintCourseInfo(node.right)

End If

End Procedure

**Menu Pseudocode:**

Procedure Main()

dataLoaded = false

dataStructure = InitializeDataStructure() // Placeholder for the specific data structure initialization

While True

Print "=== Main Menu ==="

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

choice = GetUserInput("Choose an option: ")

If choice == "1" Then

filePath = GetUserInput("Enter the file path: ")

success = LoadDataStructure(filePath, dataStructure)

dataLoaded = success

If success Then

Print "Data loaded successfully."

Else

Print "Failed to load data."

End If

ElseIf choice == "2" Then

If Not dataLoaded Then

Print "Please load the data structure first."

Else

PrintCourseList(dataStructure)

End If

ElseIf choice == "3" Then

If Not dataLoaded Then

Print "Please load the data structure first."

Else

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

PrintCourse(dataStructure, courseNumber)

End If

ElseIf choice == "4" Then

Print "Exiting program..."

ExitProcedure() // Clean up and exit

Else

Print "Invalid option, please try again."

End If

End While

End Procedure

// Abstracted procedures for specific actions

Procedure LoadDataStructure(filePath, dataStructure) Returns Boolean

// Abstracted logic to load the data structure, may vary based on the type (vector, hash, tree)

// Returns true if successful, false otherwise

End Procedure

Procedure PrintCourseList(dataStructure)

// Abstracted logic to print the course list

// Assumes dataStructure has a method to print or return courses in sorted order

End Procedure

Procedure PrintCourse(dataStructure, courseNumber)

// Abstracted logic to find and print a specific course's details

End Procedure

Procedure ExitProcedure()

// Any cleanup before exiting the program

// Exit the program

End Procedure

**Print Out List Pseudocode:**  
  
**Vector:**  
Procedure PrintCoursesFromVector(vector)

Sort vector by courseNumber using a sorting algorithm (e.g., mergesort or quicksort)

For each course in vector

Print course.courseNumber + " - " + course.title

Next

End Procedure

**Hash Table:**  
Procedure PrintCoursesFromHashTable(hashTable)

Initialize courseNumbers as an empty list

For each key in hashTable keys

Add key to courseNumbers

Next

Sort courseNumbers alphanumerically

For each courseNumber in courseNumbers

course = hashTable[courseNumber]

Print course.courseNumber + " - " + course.title

Next

End Procedure

**Binary Tree:**

Procedure PrintCoursesFromTree(node) // node is the root of the BST

If node is not null

PrintCoursesFromTree(node.left)

Print node.course.courseNumber + " - " + node.course.title

PrintCoursesFromTree(node.right)

End If

End Procedure

**Step-by-Step Analysis**:  
  
Open file for reading: cost = *O*(1), operation occurs once

Reading data from file: cost = O(n), each of the n lines is read just once.

Parsing each line, course number, title, and prerequisites if they exist: cost per line = O(1), cost total = O(n). Parsing happens n times, once per line.

Checking for formatting errors: cost per line = O(1), cost total = O(n). Conducted once for each of the n lines.

**Creating Course Objects:**

Procedure CreateCourseObjectsFromFile(filePath)

Initialize courseList as an empty list

Open file at filePath for reading

While not end of file

Read line from file

Parse line to extract courseNumber, title, and prerequisites

Create a new Course object with extracted data

Add the new Course object to courseList

End While

Close the file

Return courseList

End Procedure  
  
We can sum up the total cost to get the total worst case running time for reading the file and creating course objects we see O(1) + O(n) + O(n) + O(n) = O(n) The linear complexity of O(n) comes from the operation that scale with the number of lines in the file, which are reading the lines, parsing, and then creating course objects.

**Cost Analysis:**

1. Procedure CreateCourseObjectsFromFile(filePath)

2. Initialize courseList as an empty list

3. Open file at filePath for reading

4. While not end of file

5. Read line from file

6. Parse line to extract courseNumber, title, and prerequisites

7. Create a new Course object with extracted data

8. Add the new Course object to courseList

9. End While

10. Close the file

11. Return courseList

12. End Procedure

Line 2 initializes a list, cost = O(1), executed once.  
Line 3 opens a file, costs O(1), executed once.

Lines 4-9 present a while loop which iterates once for each course, meaning it executes n times.

Line 5 reads a line from the file, costs O(1) per iteration, n times in total.

Line 6 parses the line, cost O(1) per iteration, n times total.

Line 7 creates a new course object, cost = O(1) per iteration, n times total.

Line 8 adds the course object to the course list, cost = O(1) per iteration, n times in total.

Line 10 closes the file, cost = O(1), executed once.

Line 11 returns the list, cost = O(1), executed once.

**Advantages and Disadvantages:**

Vectors prove to be on the simpler side of the available data structures which makes them excellent for storing lists of elements. Because they have a relatively straightforward implementation, it allows them to be easier to manipulate, and basic operations like traversal, access, and modifying are more approachable. The elements within a vector are kept in contiguous memory locations, which can lead to better performance.

On the other hand, due to vectors using dynamic resizing, they can suffer from performance issues, especially if they are large. Along with this, they are not exceptionally flexible when it comes to inserting and deleting elements. They can remove or add towards their end, but in the middle, they require the shifting of elements to maintain order.

Hash tables offer fast access to elements via average case constant time (*O*(1)) complexity for inserting, deleting, and lookups. To pair with this, elements within a hash table are accessed through unique keys which helps simplify data retrieval and manipulation as it provides direct access to any element without the need to traverse the data structure.

Unfortunately, hash tables may end up consuming more memory than needed as they are not the strongest for space efficiency. It is also worth noting that if a collision occurs in a situation where the keys hash to the same index, the performance of a hash table can degrade to linear time (*O*(*n*)).

A binary tree offers a well-ordered structure through maintaining elements in a sorted order inherently. This is a major benefit for applications that require ordered data as they enable efficient in-order traversal to retrieve elements sequentially. They offer a hierarchal structure which is excellent for displaying relationships, such as file systems or organizational structures. Binary trees can also dynamically grow or shrink as elements are added and removed.

Binary trees can however present significant complexity, especially when compared to data structures such as vectors. Carrying out operations like inserting, deleting, and lookup require recursive algorithms that can appear less intuitive than linear data structures.

**Recommendation: Binary Search Tree**

Given the needs for operations like listing courses in alphanumeric order, searching for specific courses, displaying their information, and updating course prerequisites, a binary search tree is the most fitting data structure.

A binary search tree will offer the ability to effectively insert and delete, while also maintaining an ordered sequence of courses. A vector may prove simpler and more efficient in some cases, their linear search time and cost of keeping sorted will lead towards why it should be avoided in the instance. Hash tables will not offer efficient support for ordered operations.

The binary tree will ensure necessary operations like inserting, deleting, and searching can be managed effectively and efficiently, while also supporting data manipulation and retrieval in an ordered fashion. This aligns well with the needs of the program, especially when considering the need to display prerequisites in sorted order or finding courses quickly.