# 6-2 Submit Project One- CS 300 Pseudocode Document

Pseudocode for Loading Data into Vector Data Structure:

1. Open the file for reading.

2. Initialize an empty vector to store course objects.

Loop through each line in the file:

3. Read the line from the file.

4. Split the line into tokens using comma as a delimiter.

5. Check if the number of tokens is at least 2.

a. If not, display an error message and skip to the next line.

6. Extract courseNumber and name from the tokens.

7. Create a new course object with courseNumber and name.

If there are prerequisites (more than 2 tokens):

8. Loop through the prerequisite tokens.

a. Check if the prerequisite courseNumber exists in the vector.

i. If not, display an error message and skip to the next line.

b. Add the prerequisite course to the current course's prerequisites list.

9. Add the course object to the vector.

10. Close the file.

Pseudocode for Searching and Printing Course Information:

1. Prompt the user for the courseNumber they want information about.

2. Search the vector for a course with the specified courseNumber.

3. If found, display course information:

a. CourseNumber

b. CourseName

c. Prerequisites (if any)

4. If not found, display an error message indicating that the course was not found.

Pseudocode for Printing Course Information using a Vector Data Structure (provided as an example in the Pseudocode Document):

1. Loop through each course in the vector:

a. Display CourseNumber and CourseName.

b. If the course has prerequisites, display them.

c. Move to the next line.

—---------------------------------------------------------------------

Pseudocode for Loading Data into Hash Table: (Assuming a ‘Course’ and ‘HashTable’ class)

1. Open and Read File Pseudocode:

Algorithm OpenAndReadFile(filePath, hashTable)

Try to open file at filePath

If successful

While not end of file

Read a line from the file

ParseLineAndValidate(line) // Check for correct format and prerequisites

If valid, proceed to CreateAndStoreCourseObject

Else, display error message and skip to the next line

End While

Else

Display error message - unable to open file

End If

Close the file

End Algorithm

2. Parse Line and Validate Pseudocode:

Algorithm ParseLineAndValidate(line)

Split line into tokens using comma as delimiter

If tokens count < 2

Display error message - insufficient parameters on the line

Return invalid

End If

courseNumber = tokens[0]

courseName = tokens[1]

For i = 2 to tokens count - 1

prerequisite = tokens[i]

If not CourseExists(prerequisite)

Display error message - prerequisite course not found

Return invalid

End If

End For

Return valid

End Algorithm

3. Create and Store Course Object Pseudocode:

Algorithm CreateAndStoreCourseObject(tokens, hashTable)

Create a new Course object with courseNumber and courseName from tokens

For i = 2 to tokens count - 1

Add prerequisites to the Course object

End For

hashTable.Insert(courseNumber, CourseObject)

End Algorithm

4. Print Course Information Pseudocode:

Algorithm PrintCourseInformation(hashTable)

For each entry in hashTable

Display courseNumber and courseName

If course has prerequisites

Display prerequisites

End For

End Algorithm

5. CourseExists Function Pseudocode:

plaintext

Copy code

Function CourseExists(courseNumber)

If hashTable contains courseNumber

Return true

Else

Return false

End If

End Function.

----------------------------------------------------------------------

# Improved Pseudocode for Tree Data Structure

# Function to Open and Read the File

function openAndReadFile(filename, tree):

try:

file = open(filename, 'r') # Open the file in read mode

processFile(file, tree) # Call the function to process the file

file.close() # Close the file

except FileError as e:

print("Error opening/reading the file:", e)

# Function to Process the File

function processFile(file, tree):

for line in file.readlines():

validateLine(line) # Call the function to validate the line

course = createCourseObject(line) # Call the function to create a course object

insertCourseIntoTree(tree, course) # Call the function to insert the course into the tree

# Function to Validate Line Format

function validateLine(line):

tokens = line.split(',') # Split the line into tokens

if len(tokens) < 2:

raise FormatError("Invalid line format. Each line should have at least two parameters.")

for i in range(2, len(tokens)):

if not courseExists(tokens[i]):

raise PrerequisiteError("Prerequisite not found:", tokens[i])

# Function to Check if Course Exists

function courseExists(courseNumber):

# Check if the courseNumber exists in the tree

return searchTree(tree, courseNumber) is not None

# Function to Create Course Object

function createCourseObject(line):

tokens = line.split(',') # Split the line into tokens

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:] if len(tokens) > 2 else [] # Get prerequisites if available

return Course(courseNumber, courseTitle, prerequisites)

# Function to Insert Course into Tree

function insertCourseIntoTree(tree, course):

# Insert the course into the tree data structure

insertIntoTree(tree, course)

# Function to Print Course Information

function printCourseInformation(tree, courseNumber):

# Search for the course in the tree

node = searchTree(tree, courseNumber)

if node is not None:

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

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

print("Prerequisites:", node.course.prerequisites)

else:

print("Course not found.")

# Function to Create Empty Tree

function createEmptyTree():

# Create and return an empty tree data structure

return BinaryTree()

# Function to Search Tree for Course

function searchTree(tree, courseNumber):

# Search the tree for a course with the given courseNumber

return searchNode(tree.root, courseNumber)

# Function to Search Node in Tree

function searchNode(node, courseNumber):

# Recursive function to search for a course in the tree

if node is None or node.course.courseNumber == courseNumber:

return node

leftResult = searchNode(node.left, courseNumber)

rightResult = searchNode(node.right, courseNumber)

return leftResult if leftResult is not None else rightResult

# Function to Insert into Tree

function insertIntoTree(tree, course):

# Insert the course into the tree using appropriate logic

tree.root = insertNode(tree.root, course)

# Function to Insert Node into Tree

function insertNode(node, course):

# Recursive function to insert a course node into the tree

if node is None:

return Node(course)

if course.courseNumber < node.course.courseNumber:

node.left = insertNode(node.left, course)

elif course.courseNumber > node.course.courseNumber:

node.right = insertNode(node.right, course)

return node

# Pseudocode for Menu

# Function to Display Menu Options

function displayMenu():

print("Menu:")

print("1. Load Data Structure")

print("2. Print Course List")

print("3. Print Course Information")

print("4. Exit")

# Function to Load Data Structure

function loadDataStructure():

# Implement the logic to load data into the chosen data structure

# This may involve calling functions from previous pseudocode

# Function to Print Alphanumerically Ordered Course List

function printCourseList():

# Implement the logic to print an alphanumerically ordered list of courses

# This may involve calling functions from previous pseudocode

# Function to Print Course Information

function printCourseInformation():

courseNumber = input("Enter the course number to print information: ")

# Implement the logic to print the course title and prerequisites for the specified course

# This may involve calling functions from previous pseudocode

# Main Program

dataStructureLoaded = False # Flag to track whether data structure is loaded

while True:

displayMenu() # Display menu options

choice = input("Enter your choice (1-4): ")

if choice == "1":

loadDataStructure() # Load data into the data structure

dataStructureLoaded = True

elif choice == "2":

if dataStructureLoaded:

printCourseList() # Print alphanumerically ordered course list

else:

print("Error: Data structure not loaded. Please load data first.")

elif choice == "3":

if dataStructureLoaded:

printCourseInformation() # Print course information for a specific course

else:

print("Error: Data structure not loaded. Please load data first.")

elif choice == "4":

print("Exiting the program. Goodbye!")

break # Exit the loop and end the program

else:

print("Invalid choice. Please enter a number between 1 and 4.")

—---------------------------------------------------------------------

## 

Example Runtime Analysis for Printing Course Information using Tree Data Structure

| **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 | n | n |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | k | k |
| **Total Cost** | | | 4n + k |
| **Runtime** | | | O(n+k) |

Advantages and Disadvantages of the Tree Data Structure:

Advantages:

-Efficient searching, insertion, and deletion with a balanced tree.

-Maintains a sorted order, facilitating alphabetical or numerical traversals.

-Allows for easy retrieval of a specific course based on its course number.

Disadvantages:

-Requires additional logic for balancing (e.g., AVL or Red-Black trees), adding some complexity.

-Slightly more memory overhead compared to a vector due to pointers and structure.

-May not be as straightforward to implement as a vector or hash table.

Advantages and Disadvantages of the Vector Data Structure:

Advantages:

-Simple and easy to implement.

-Efficient for sequential access to course information.

Disadvantages:

-Searching for a specific course may require iterating through the entire vector.

-Adding or removing courses may involve shifting elements, leading to performance overhead.

Advantages and Disadvantages of the Hash Table Data Structure:

Advantages:

-Provides fast average-case time complexity for searching, insertion, and deletion.

-Efficient for direct access to course information using course numbers.

Disadvantages:

-Hash collisions may occur, impacting performance.

-May not maintain the order of insertion if that is a requirement.

Based on the runtime analysis and the specific requirements of the advising program at ABCU, the recommended data structure is the Tree Data Structure. The balanced tree offers efficient searching, insertion, and deletion operations with a runtime complexity of O(log n). It also maintains a sorted order, aligning well with the need to print an alphanumerically ordered list of Computer Science courses. While slightly more complex than a vector, the benefits of efficient searching and order maintenance make the tree an appropriate choice, striking a balanced trade-off between simplicity and functionality for the advising program.