**Project One: Data Structure Analysis and Pseudocode Design**

Introduction

In this project, I demonstrate my understanding of non-coding development methodologies and the evaluation of complex data structures through advanced algorithmic designs. The goal is to develop a program for the Computer Science department at ABCU that prints a list of all Computer Science courses in alphanumeric order and, for a given course, prints out its title and prerequisites.

## Pseudocode Design

### Vector Data Structure

Class Course

String courseNumber

String name

Vector<String> prerequisites

Function Course(courseNumber, name, prerequisites)

this.courseNumber = courseNumber

this.name = name

this.prerequisites = prerequisites

#### Load Courses from File

Function LoadCoursesFromFile(fileName)

Vector<Course> courses = new Vector<Course>()

File file = openFile(fileName)

while not endOfFile(file)

String line = readLine(file)

Vector<String> tokens = split(line, ',')

if tokens.size() < 2

print("Error: Insufficient data in line: " + line)

continue

String courseNumber = tokens[0]

String name = tokens[1]

Vector<String> prerequisites = new Vector<String>()

for i from 2 to tokens.size() - 1

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, name, prerequisites)

courses.add(course)

closeFile(file)

sortCourses(courses) // Sort courses in alphanumeric order

return courses

Function sortCourses(courses)

// Sort courses based on courseNumber

sort courses by courseNumber

#### Print Course Information

Function printCourseInformation(courses, courseNumber)

for course in courses

if course.courseNumber == courseNumber

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

print("Name: " + course.name)

print("Prerequisites: ")

for prerequisite in course.prerequisites

print(prerequisite)

return

print("Course not found")

### Hash Table Data Structure

#### Load Courses from File

Function loadCoursesFromFile(fileName)

HashTable<String, Course> courses = new HashTable<String, Course>()

File file = openFile(fileName)

while not endOfFile(file)

String line = readLine(file)

String[] tokens = split(line, ',')

if tokens.length < 2

print("Error: Insufficient data in line: " + line)

continue

String courseNumber = tokens[0]

String name = tokens[1]

List<String> prerequisites = new List<String>()

for i from 2 to tokens.length - 1

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, name, prerequisites)

courses.put(courseNumber, course)

closeFile(file)

return courses

#### Print Course Information

Function printCourseInformation(courses, courseNumber)

if courses.containsKey(courseNumber)

Course course = courses.get(courseNumber)

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

print("Name: " + course.name)

print("Prerequisites: ")

for prerequisite in course.prerequisites

print(prerequisite)

else

print("Course not found")

### Tree Data Structure

#### Load Courses into a Tree

Function LoadCoursesFromFileToTree(fileName)

Tree<Course> coursesTree = new Tree<Course>()

File file = openFile(fileName)

while not endOfFile(file)

String line = readLine(file)

List<String> tokens = split(line, ',')

if tokens.size() < 2

print("Error: Insufficient data in line: " + line)

continue

String courseNumber = tokens[0]

String name = tokens[1]

List<String> prerequisites = new List<String>()

for i from 2 to tokens.size() - 1

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, name, prerequisites)

coursesTree.insert(course)

closeFile(file)

return coursesTree

#### Print Course Information

Function printCourseInformation(coursesTree, courseNumber)

Course course = coursesTree.find(courseNumber)

if course is not null

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

print("Name: " + course.name)

print("Prerequisites: ")

for prerequisite in course.prerequisites

print(prerequisite)

else

print("Course not found")

### Menu Design Pseudocode

Function mainMenu()

while true

print("Welcome to ABCU Computer Science Course Management System")

print("1. Load Course Data")

print("2. Print Course List")

print("3. Print Course Details")

print("4. Search for a Course")

print("5. Exit")

choice = getUserInput("Enter your choice: ")

if choice == "1"

fileName = getUserInput("Enter the filename to load: ")

loadCoursesFromFile(fileName)

print("Course data loaded successfully.")

elif choice == "2"

printCourseList()

elif choice == "3"

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

printCourseInformation(courseNumber)

elif choice == "4"

searchQuery = getUserInput("Enter course name or number to search: ")

searchAndPrintCourse(searchQuery)

elif choice == "5"

print("Exiting the program. Thank you!")

break

else

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

## Runtime Analysis

**LoadCoursesFromFileToTree Function**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Read line from file** | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Check if tokens size < 2 | 1 | n | n |
| Assign course number and name | 1 | n | n |
| Loop through prerequisites | 1 | n \* avg(p) | n \* avg(p) |
| Check if prerequisite exists | 1 | n \* avg(p) | n \* avg(p) |
| Add prerequisites to vector | 1 | n \* avg(p) | n \* avg(p) |
| Create and add Course object | 1 | n | n |
| **Total Cost** | | | **5n + 3n \* avg(p)** |
| **Runtime** | | | O(n \* avg(p)) |

### Notes

* **n**: Total number of courses in the file.
* **avg(p):** Average number of prerequisites per course.
* **Line Cost:** The cost of executing a single operation or line of pseudocode.
* **# Times Executes**: The number of times each operation is executed.
* **Total Cost**: The total cost of each operation, calculated as Line Cost multiplied by # Times Executes.
* **Runtime Complexity**: The overall time complexity of each operation.
* **Advantages**:
  + Simple and easy to implement.
  + Provides efficient access and storage for a small number of courses.
  + Good for scenarios where the course list doesn't change often.
* **Disadvantages**:
  + Inefficient for large datasets due to O(n) access time.
  + Adding or removing courses can be costly, especially if resizing is needed.
  + Linear search for courses and prerequisites can be slow.

## Hash Table Data Structure

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Check if courseNumber in hashTable | 1 | 1 | 1 |
| Retrieve course from hashTable | 1 | 1 | 1 |
| Print course details | 1 | 1 | 1 |
| Check if prerequisite in hashTable | 1 | n | n |
| Retrieve prerequisite course | 1 | n | n |
| Print prerequisite details | 1 | n | n |
| **Total Cost** | | | 3n + 3 |
| **Runtime** | | | O(n) |

### Notes

* **n**: Number of prerequisites for a course. This affects how many times certain operations are done.
* **Line Cost**: How much time one line of code takes to run. Usually, this is a very quick, constant time.
* **# Times Executes**: How many times we do each operation. For some steps, this depends on the number of prerequisites (n).
* **Total Cost**: The overall time taken for each operation, found by multiplying how often it happens (# Times Executes) by how long it takes (Line Cost).
* **Runtime Complexity**: The total time it takes to run all operations. Here, it mostly depends on the number of prerequisites (n), making it O(n).
* **Advantages**:
  + Fast access to individual courses, typically O(1) on average.
  + Efficient for scenarios where frequent lookups of course information are required.
  + Good for handling a large number of courses.
* **Disadvantages**:
  + Hash collisions can degrade performance to O(n) in the worst case.
  + Maintaining an alphanumeric order of courses is not easy.
  + More complex creating compared to a vector.

**LoadCoursesFromFileToTree Function**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Read line from file** | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Add course number to set | 1 | n | n |
| **Check and add prerequisites** | 1 | n\*m | n\*m |
| **Insert course into tree** | Log(n) | n | n\*log(n) |
| **Total Cost** | | | 3n + n \* m + n \* log(n) |
| **Runtime** | | | O(nm + n log(n)) |

### **Notes**

* **n**: Number of lines in the file, usually equal to the number of courses.
* **m**: Average number of prerequisites per course.
* **Line Cost**: Time taken for one line of code to run. Generally, this is a quick, constant time.
* **# Times Executes:** How often each operation is performed. This varies depending on the number of courses (n) and their prerequisites (m).
* **Total Cost**: The overall time for each operation, calculated by multiplying the frequency of the operation (# Times Executes) by the time it takes (Line Cost).
* **Runtime Complexity:** The total time to complete all operations. In this case, it's influenced by both the number of courses (n) and the average number of prerequisites (m), leading to a complexity of O(nm + n log(n)).
* **Advantages**:
  + Maintains courses in an ordered structure.
  + Efficient access time of O(log n) for balanced trees.
  + Better performance than vectors for large data.
* **Disadvantages**:
  + More complex to implement and maintain than vectors or hash tables.
  + Slower access time compared to hash tables (O(log n) vs. O(1)).
  + Balancing the tree can add overhead, especially when frequently adding or removing courses.

## Conclusion and Recommendation

Based on the runtime analysis, the hash table data structure is the most efficient for accessing individual course information due to its average O(1) access time. However, if the requirement for maintaining an alphanumeric order of courses is a priority, a balanced tree structure like AVL or Red-Black Tree would be better besides its O(log n) access time, as it maintains order. The vector structure, while simple, is less efficient for large datasets due to its O(n) access time for course information.

My recommendation for the ABCU Computer Science department's program would be to use a **balanced tree structure** if alphanumeric ordering is needed, or a **hash table** if quick access to individual courses is more important. This decision should be based on the specific requirements and priorities of the department.