Felix Carela
12/03/2023
CS-300 DSA: Analysis and Design
Dr. Webb

6-2 Submit Project One

File Reading and Parsing:

Function readFile(fileName):

    Open fileName for reading

    Create an empty list courses

    While not end of file:

        Read a line from the file

        If the line is not empty:

            Parse the line into courseNumber, name, and prerequisites

            Create a Course object with parsed data

            Add the Course object to courses list

    Close the file

    Return courses

Course Object:

```
Class Course:

    Constructor(courseNumber, name, prerequisites):

        this.courseNumber = courseNumber

        this.name = name

        this.prerequisites = prerequisites (list)
```

Data Structure-Specific Pseudocode:

Vector:

```
Class CourseVector:

    Constructor():

        this.courses = empty vector



    Function addCourse(course):

        Add course to this.courses



    Function printCourses():

        Sort this.courses based on courseNumber

        For each course in this.courses:
```

Print course details

Function printCourseDetails(courseNumber):

For each course in this.courses:

If course.courseNumber equals courseNumber:

Print course details and prerequisites

Hash Table:

Class CourseHashTable:

Constructor():

this.courses = empty hash table

Function addCourse(course):

Add course to this.courses with courseNumber as key

Function printCourses():

Create a sorted list of courseNumbers from this.courses keys

For each courseNumber in sorted list:

Print details of this.courses[courseNumber]

```
    Function printCourseDetails(courseNumber):

        If courseNumber in this.courses:

            Print details of this.courses[courseNumber]
```

Tree:

```
Class CourseTree:

    Constructor():

        this.root = null



    Function addCourse(course):

        Insert course into the tree based on courseNumber



    Function inOrderTraversal(node):

        If node is not null:

            inOrderTraversal(node.left)

            Print node's course details

            inOrderTraversal(node.right)
```

```
  Function printCourses():

    Call inOrderTraversal(this.root)



  Function printCourseDetails(courseNumber):

    Search for the node with courseNumber

    If found, print course details
```

Menu Driven User Interface:

```
Function mainMenu(dataStructure):

  Loop indefinitely:

    Display options: Load Data, Print Course List, Print Course, Exit

    Get user choice

    If choice is Load Data:

      Read file and load data into dataStructure

    Else if choice is Print Course List:

      Call dataStructure.printCourses()

    Else if choice is Print Course:

      Get courseNumber from user

      Call dataStructure.printCourseDetails(courseNumber)
```

Else if choice is Exit:

Break from the loop

Evaluation:

Analysis of Pseudocode Steps

1. **Reading the file and creating course objects**:
    - **Line Cost**: Assume the cost for each line of code is 1.
    - **Number of Executions**: If there are n courses in the file, the loop to read and parse each course will execute n times.
    - **Total Cost**: For each course, you're reading a line, parsing it, and creating a course object. Assuming these operations are constant time (O(1)), the total cost for this operation is O(n).

Data Structure Analysis

1. **Vector**:
    - **Advantages**:

      Simplicity in implementation.

      Efficient for indexed access and iteration.

      Maintains insertion order, which is useful for printing courses in the order they were added.

    - **Disadvantages**:

      Insertion can be costly if the vector needs to resize (worst-case O(n)).

      Searching for a specific course or prerequisite is O(n) as it requires linear traversal.

2. **Hash Table**:
    - **Advantages**:

      Fast lookup for courses (average case O(1)).

      Efficient for scenarios where the course number is known.

    - **Disadvantages**:

Does not maintain insertion order.

Handling collisions can be complex.

Worst-case lookup time can degrade to O(n) (although rare with a good hash function).

3. **Tree (e.g., Binary Search Tree)**:
    - **Advantages**:

        Maintains a sorted order, which is beneficial for ordered printing.

        Lookup, insertion, and deletion operations can be efficient (average case O(log n)).

    - **Disadvantages**:

        Can become unbalanced, degrading performance to O(n) in the worst case.

        More complex implementation than a vector or hash table.

Recommendation

Given three data structures:

For a simple implementation with ordered data: A **vector** is suitable if the data set is not too large and efficiency in insertion or search is not a primary concern.

For fast lookup and search operations: A **hash table** is ideal, especially when you need to access courses directly by their number.

For balanced performance and ordered data: A **tree** structure (like a balanced BST) is recommended. It offers a good trade-off between insertion, search, and maintaining order.

Based on the Big O analysis and considering the functionalities like printing courses in order and searching for specific courses and their prerequisites, a **tree** might be the most balanced choice. It efficiently supports the required operations while maintaining the data in a sorted order, which is beneficial for some of the program's key functionalities.