Ryan Hatch  
SNHU  
April 13, 2024

**Pseudocode for Courses**  
  
  
  
  
**//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Class Definition  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\**

Class Course:

Properties:

- courseNumber: String

- name: String

- prerequisites: Vector<String>

Constructor(courseNumber, name, prerequisites):

this.courseNumber = courseNumber

this.name = name

this.prerequisites = prerequisites

**//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Shared Functions  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\**

Function validateLineFormat(line):

tokens = Split line by commas

Return len(tokens) >= 2 **// Ensure there are at least a course number and name**

Function parseLine(line):

tokens = Split line by commas

courseNumber = tokens[0]

name = tokens[1]

prerequisites = tokens[2..] **// Might be empty**

Return (courseNumber, name, prerequisites)

Function courseExists(courses, courseNumber):

For each course in courses:

If course.courseNumber == courseNumber:

Return True

Return False

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\  
// Vector Pseudocode  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\**

Function loadCoursesFromFile(filePath):

Initialize courses as Vector<Course>

Open file at filePath for reading

If file is not open:

Print "Error opening file"

Return courses

While not end of file:

line = Read next line from file

If validateLineFormat(line) is True:

courseData = parseLine(line)

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

courses.push\_back(course)

Else:

Print "Error in file format for line: " + line

Close file

Return courses

Function validateLineFormat(line):

tokens = Split line by commas

Return len(tokens) >= 2 **// Ensure there are at least a course number and name**

Function parseLine(line):

tokens = Split line by commas

courseNumber = tokens[0]

name = tokens[1]

prerequisites = tokens[2..] **// Might be empty**

Return (courseNumber, name, prerequisites)

Function courseExists(courses, courseNumber):

For each course in courses:

If course.courseNumber == courseNumber:

Return True

Return False

Function printCourseInformation(courses: Vector<Course>, courseNumber: String):

For each course in courses:

If course.courseNumber == courseNumber:

Print "Course: " + course.name + " (" + course.courseNumber + ")"

If course.prerequisites.size() > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

printCourseInformation(courses, prerequisite) **// Recursive call to print prerequisites details**

Return

Print "Course not found"

Function sortAndPrintCourses(courses: Vector<Course>):

Sort courses by courseNumber using a suitable sort algorithm (like quicksort)

For each course in courses:

Print course.courseNumber + ": " + course.name

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\  
// **Hashtable Pseudocode**  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\

Function loadCoursesIntoHashtable(filePath):

Initialize courses as Hashtable<String, Course>

Open file at filePath for reading

If file is not open:

Print "Error opening file"

Return courses

While not end of file:

line = Read next line from file

If validateLineFormat(line) is True:

courseData = parseLine(line)

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

courses.put(course.courseNumber, course) **// Using 'put' for hashtable**

Else:

Print "Error in file format for line: " + line

Close file

Return courses

Function printSortedCoursesHashtable(courses: Hashtable<String, Course>):

Initialize courseNumbers as Vector<String>

For each courseNumber in courses.keys():

courseNumbers.push\_back(courseNumber)

Sort courseNumbers

For each courseNumber in courseNumbers:

course = courses.get(courseNumber)

Print course.courseNumber + ": " + course.name

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\  
// Class Definition for Tree Nodes and Tree Structure  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\\**

Class TreeNode:

Properties:

- course: Course

- left: TreeNode

- right: TreeNode

Constructor(course):

this.course = course

this.left = null

this.right = null

Class CourseTree:

Properties:

- root: TreeNode

Constructor():

this.root = null

**// Function to load courses into a binary search tree from a file**

Function loadCoursesIntoTree(filePath):

Initialize tree as CourseTree

Open file at filePath for reading

If file is not open:

Print "Error opening file"

Return tree

While not end of file:

line = Read next line from file

If validateLineFormat(line) is True:

courseData = parseLine(line)

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

tree.root = insertCourseIntoTree(tree.root, course) **// Recursive insertion**

Else:

Print "Error in file format for line: " + line

Close file

Return tree

**// Recursive function to insert a new course into the tree based on course number**

Function insertCourseIntoTree(node, course):

If node is null:

Return new TreeNode(course)

If course.courseNumber < node.course.courseNumber:

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

Else if course.courseNumber > node.course.courseNumber:

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

Return node

**// Function to perform in-order traversal and print courses in alphanumeric order**

Function inOrderTraversal(node):

If node is null:

Return

inOrderTraversal(node.left)

Print node.course.courseNumber + ": " + node.course.name

inOrderTraversal(node.right)

**// Function to initiate in-order traversal from the root of the tree**

Function printSortedCoursesTree(root):

inOrderTraversal(root)

## Vector Runtime Analysis:

| **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 | P | P |
| **print the prerequisite course information** | 1 | p | p |
| **Total Cost** | | | 4n + 4p + 1 |
| **Runtime (assuming p < n)** | | | O(n +p) |

## HashTable Runtime Analysis:

| **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 | P | P |
| **print the prerequisite course information** | 1 | p | p |
| **Total Cost** | | | n + 2 + 2p |
| **Runtime assuming good hashing without collisions:** | | | O(n +p) |
| **Runtime assuming worst case with collisions:** | | | O(n^2) |

## Tree Runtime Analysis:

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

**P is the average number of prerequisites per course. In the worst case, if a BST becomes unbalanced, the operation could simply degrade to O(n).**

## Evaluation Summary:

**Vector:**

* **Advantages**: Simple to implement, retains the order of the elements as they are inserted, and also supports random access.
* **Disadvantages**: Inefficient searches (linear time), and end insertions can be slow if resizing is frequently needed.

**Hash Table**:

* **Advantages**: Typically offers fast performance for insertions and lookups because of the constant time complexity.
* **Disadvantages**: Does not inherently maintain an order, can suffer from performance drops during collisions, along with the added complexity of rehashing and managing hash functions.

**Binary Search Tree:**

* **Advantages**: Keeps data sorted, which is optimal for in-order traversals, and offers logarithmic time complexity for most operations if the tree is balanced.
* **Disadvantages**: More complex to implement, especially when ensuring the tree stays balanced.

## Recommendations and Justifications:

Considering the structure of the course data, which includes prerequisites that also need to be orderly displayed, a binary search tree would be the most suitable choice for several reasons:

* Efficiently retrieves and displays the courses and their prerequisites sorted in order through in-order traversal.
* The complexity of implementing a Binary Search Tree is manageable, especially with available modern libraries that support self-balancing trees.
* Binary Search Trees naturally represent hierarchical structures, which aligns well with the prerequisite relationships among courses.

Even through vectors can be straightforward and preserve insertion order, they do not suit the needs of frequently accessing courses by course number, which would require inefficient linear searches. Hash tables, though quick for lookups, fail to maintain course order, which make it necessary to repeat the sorting of keys for display. With this, a self-balancing tree ensures that there is efficient lookup, insertion, and a natural in-order traversal of courses.

**Notes:**

* The choice of which data structure to use might vary based on the dataset size and how often it changes. For smaller datasets with less updates, more simple structures like vectors or hash tables might be adequate.
* For larger or frequently updated course data, a self-balancing BST would manage changes more effectively and efficiently.