Amanda Stone

CS300: Project One

### Updated Pseudocode from Milestone 1, 2, and 3

#### **Step 1: Opening and Reading the File**

FUNCTION LoadCoursesFromFile(fileName):

TRY:

OPEN file with name fileName in read mode

CATCH exception:

PRINT "Error: Unable to open file."

RETURN []

INITIALIZE an empty list called courseLines

FOR each line IN file:

TRIM whitespace from line

IF line IS NOT empty:

APPEND line TO courseLines

CLOSE file

RETURN courseLines

FUNCTION ValidateAndParse(courseLines):

INITIALIZE an empty dictionary called courseDict

FOR each line IN courseLines:

SPLIT line BY commas INTO tokens

IF LENGTH(tokens) < 2:

PRINT "Error: Invalid line format. Each line must have at least a course number and title."

RETURN {}

ASSIGN tokens[0] TO courseNumber

ASSIGN tokens[1] TO courseTitle

ASSIGN tokens[2:] TO prerequisites (if any)

IF courseNumber EXISTS IN courseDict:

PRINT "Error: Duplicate course number found: " + courseNumber

RETURN {}

CREATE a new Course object (courseNumber, courseTitle, prerequisites)

ADD course TO courseDict WITH courseNumber AS the key

FOR each Course object IN courseDict:

FOR each prerequisite IN Course.prerequisites:

IF prerequisite IS NOT a key IN courseDict:

PRINT "Error: Prerequisite " + prerequisite + " does not exist in the course list."

RETURN {}

RETURN courseDict

#### **Step 2: Parsing and Validating Data**

FUNCTION ValidateAndParse(courseLines):

INITIALIZE an empty dictionary called courseDict

FOR each line IN courseLines:

SPLIT line BY commas INTO tokens

IF LENGTH(tokens) < 2:

PRINT "Error: Invalid line format. Each line must have at least a course number and title."

RETURN {}

ASSIGN tokens[0] TO courseNumber

ASSIGN tokens[1] TO courseTitle

ASSIGN tokens[2:] TO prerequisites (if any)

IF courseNumber EXISTS IN courseDict:

PRINT "Error: Duplicate course number found: " + courseNumber

RETURN {}

CREATE a new Course object (courseNumber, courseTitle, prerequisites)

ADD course TO courseDict WITH courseNumber AS the key

FOR each Course object IN courseDict:

FOR each prerequisite IN Course.prerequisites:

IF prerequisite IS NOT a key IN courseDict:

PRINT "Error: Prerequisite " + prerequisite + " does not exist in the course list."

RETURN {}

RETURN courseDict

#### **Step 3: Storing Courses in a Binary Search Tree**

FUNCTION StoreCoursesInTree(courseDict, tree):

FOR each courseNumber IN courseDict:

RETRIEVE the Course object FROM courseDict USING courseNumber

INSERT the Course object INTO the binary search tree (tree)

#### **Step 4: Validating Prerequisites for User Feedback**

FUNCTION ValidatePrerequisites(courses):

FOR EACH course IN courses:

FOR EACH prerequisite IN course.prerequisites:

IF NOT CourseExists(prerequisite, courses):

PRINT "Warning: Prerequisite " + prerequisite + " does not exist for course " + course.courseNumber

FUNCTION CourseExists(courseNumber, courses):

FOR EACH course IN courses:

IF course.courseNumber == courseNumber:

RETURN TRUE

RETURN FALSE

#### **Step 5: Printing Course Information**

FUNCTION PrintCourseInformation(courseDict):

FOR each courseNumber IN SORTED keys OF courseDict:

RETRIEVE Course object FROM courseDict USING courseNumber

PRINT "Course Number: " + Course.courseNumber

PRINT "Title: " + Course.title

IF Course.prerequisites IS NOT empty:

PRINT "Prerequisites: " + JOIN(Course.prerequisites, ", ")

ELSE:

PRINT "Prerequisites: None"

#### **Step 6: Main Program**

START:

PROMPT user FOR file name

CALL LoadCoursesFromFile(fileName) AND STORE the result IN courseLines

IF courseLines IS EMPTY:

PRINT "File validation failed or file could not be loaded."

EXIT program

CALL ValidateAndParse(courseLines) AND STORE the result IN courseDict

IF courseDict IS EMPTY:

PRINT "Data parsing failed due to validation errors."

EXIT program

PRINT "Courses successfully loaded."

PROMPT user FOR course number to view

CALL PrintCourseInformation(courseDict) FOR the specified course

// OPTIONAL: Store courses in a binary search tree for efficient traversal

INITIALIZE an empty binary search tree (tree)

CALL StoreCoursesInTree(courseDict, tree)

// OPTIONAL: Perform additional validations

CALL ValidatePrerequisites(courseDict)

END

### Evaluation of Run Time and Memory

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree** |
| **Loading Data** | *O(n)* | *O(n)* | *O(n)O(n)*O(n) |
| **Parsing and Validation** | O(n2⋅p) | *O(n⋅p)* | *O(n⋅p⋅log ⁡n)* |
| **Insertion** | *O(n)* | *O(n)* | *O(n⋅log⁡ n)* |
| **Total Loading Time** | *O(n2⋅p)* | *O(n⋅p)* | *O(n⋅p⋅log⁡n)* |
| **Searching for a Course** | *O(n)* | *O(1)* | *O(log⁡n)* |
| **Memory Usage** | Low | Moderate to High | Moderate |

### **Key:**

* ***n*:** Number of courses.
* ***p*:** Average number of prerequisites per course.
* **Vector Pros/Cons:** Simple and memory-efficient but suffers from poor runtime for validation and searching.
* **Hash Table Pros/Cons:** Excellent lookup and validation efficiency, but higher memory overhead and no order preservation.
* **Binary Search Tree Pros/Cons:** Balanced runtime performance and maintains order but requires self-balancing for optimal efficiency.

### Recommendation

**Final Recommendation**: **Hash Table!**

Hash tables are the optimal data structure for this project because they provide constant time operations for lookups, insertions, and validations. While memory overhead is slightly higher and order is not maintained, these drawbacks are outweighed by their scalability and efficiency in handling large datasets. For this project's focus on validation and prerequisite checking, the hash table is the clear winner!