Menu Pseudocode:

BEGIN

// Define the data structure to store course information

DATA\_STRUCTURE = {} // Dictionary to store course information, where key is course code and value is (title, prerequisites)

FUNCTION LOAD\_DATA\_FROM\_FILE(file\_path):

OPEN file\_path FOR READING AS file

WHILE NOT end of file:

READ line FROM file

PARSE line INTO course\_code, course\_title, prerequisites

ADD (course\_title, prerequisites) TO DATA\_STRUCTURE WITH KEY course\_code

CLOSE file

FUNCTION PRINT\_COURSES\_ALPHANUMERICALLY():

SORT DATA\_STRUCTURE BY course\_title

FOR EACH course IN DATA\_STRUCTURE:

PRINT course\_code + ": " + course\_title

FUNCTION PRINT\_COURSE\_DETAILS(course\_code):

IF course\_code EXISTS IN DATA\_STRUCTURE:

course\_title, prerequisites = DATA\_STRUCTURE[course\_code]

PRINT "Course Title: " + course\_title

PRINT "Prerequisites: " + prerequisites

ELSE:

PRINT "Course not found"

// Main program loop

WHILE TRUE:

PRINT "Menu:"

PRINT "1. Load file data"

PRINT "2. Print alphanumerically ordered list of courses"

PRINT "3. Print course title and prerequisites"

PRINT "9. Exit"

CHOICE = GET\_USER\_INPUT()

IF CHOICE = 1:

FILE\_PATH = GET\_USER\_INPUT("Enter file path: ")

CALL LOAD\_DATA\_FROM\_FILE(FILE\_PATH)

PRINT "Data loaded successfully"

ELSE IF CHOICE = 2:

CALL PRINT\_COURSES\_ALPHANUMERICALLY()

ELSE IF CHOICE = 3:

COURSE\_CODE = GET\_USER\_INPUT("Enter course code: ")

CALL PRINT\_COURSE\_DETAILS(COURSE\_CODE)

ELSE IF CHOICE = 9:

PRINT "Exiting program"

BREAK

ELSE:

PRINT "Invalid choice. Please enter a valid option."

END

Pseudocode That will print out a list of courses:

BEGIN

// Define the data structures

VECTOR courses // Vector to store course entries

HASH\_TABLE course\_table // Hash table to store course entries with course\_code as the key

TREE course\_tree // Binary search tree to store course entries with course\_code as the key

// Define functions for Vector

FUNCTION LOAD\_DATA\_TO\_VECTOR(file\_path):

OPEN file\_path FOR READING AS file

WHILE NOT end of file:

READ line FROM file

PARSE line INTO course\_code, course\_title, prerequisites

ADD (course\_code, course\_title, prerequisites) TO VECTOR courses

CLOSE file

FUNCTION SORT\_VECTOR\_BY\_COURSE\_CODE():

SORT VECTOR courses BY course\_code ALPHANUMERICALLY

FUNCTION PRINT\_SORTED\_COURSES\_VECTOR():

CALL SORT\_VECTOR\_BY\_COURSE\_CODE()

FOR EACH entry IN VECTOR courses:

PRINT entry.course\_code + ": " + entry.course\_title

// Define functions for Hash Table

FUNCTION LOAD\_DATA\_TO\_HASH\_TABLE(file\_path):

OPEN file\_path FOR READING AS file

WHILE NOT end of file:

READ line FROM file

PARSE line INTO course\_code, course\_title, prerequisites

ADD (course\_code, (course\_title, prerequisites)) TO HASH\_TABLE course\_table

CLOSE file

FUNCTION GET\_SORTED\_COURSE\_CODES():

COURSE\_CODES = EMPTY LIST

FOR EACH key IN HASH\_TABLE course\_table:

ADD key TO COURSE\_CODES

SORT COURSE\_CODES ALPHANUMERICALLY

RETURN COURSE\_CODES

FUNCTION PRINT\_SORTED\_COURSES\_HASH\_TABLE():

COURSE\_CODES = CALL GET\_SORTED\_COURSE\_CODES()

FOR EACH course\_code IN COURSE\_CODES:

course\_title, prerequisites = HASH\_TABLE course\_table[course\_code]

PRINT course\_code + ": " + course\_title

// Define functions for Binary Search Tree

FUNCTION INSERT\_INTO\_TREE(tree\_node, course\_code, course\_title, prerequisites):

IF tree\_node IS NULL:

RETURN NEW NODE WITH course\_code, course\_title, prerequisites

IF course\_code < tree\_node.course\_code:

tree\_node.left = CALL INSERT\_INTO\_TREE(tree\_node.left, course\_code, course\_title, prerequisites)

ELSE:

tree\_node.right = CALL INSERT\_INTO\_TREE(tree\_node.right, course\_code, course\_title, prerequisites)

RETURN tree\_node

FUNCTION LOAD\_DATA\_TO\_TREE(file\_path):

OPEN file\_path FOR READING AS file

WHILE NOT end of file:

READ line FROM file

PARSE line INTO course\_code, course\_title, prerequisites

ROOT OF TREE course\_tree = CALL INSERT\_INTO\_TREE(ROOT OF TREE course\_tree, course\_code, course\_title, prerequisites)

CLOSE file

FUNCTION PRINT\_INORDER\_TREE(node):

IF node IS NOT NULL:

CALL PRINT\_INORDER\_TREE(node.left)

PRINT node.course\_code + ": " + node.course\_title

CALL PRINT\_INORDER\_TREE(node.right)

FUNCTION PRINT\_SORTED\_COURSES\_TREE():

CALL PRINT\_INORDER\_TREE(ROOT OF TREE course\_tree)

// Main program logic

FILE\_PATH = GET\_USER\_INPUT("Enter the file path: ")

PRINT "Loading data into vector..."

CALL LOAD\_DATA\_TO\_VECTOR(FILE\_PATH)

PRINT "Printing sorted courses (Vector)..."

CALL PRINT\_SORTED\_COURSES\_VECTOR()

PRINT "Loading data into hash table..."

CALL LOAD\_DATA\_TO\_HASH\_TABLE(FILE\_PATH)

PRINT "Printing sorted courses (Hash Table)..."

CALL PRINT\_SORTED\_COURSES\_HASH\_TABLE()

PRINT "Loading data into binary search tree..."

CALL LOAD\_DATA\_TO\_TREE(FILE\_PATH)

PRINT "Printing sorted courses (Binary Search Tree)..."

CALL PRINT\_SORTED\_COURSES\_TREE()

END

Evaluation:

In evaluating the runtime and memory requirements for handling course data, the hash table emerges as the most suitable data structure due to its efficiency. Opening the file and reading each line involves a constant time complexity of O(1) and a linear time complexity of O(n) respectively, where n represents the number of courses. Parsing each line, validating prerequisites, and creating course objects each have complexities of O(n⋅m), O(n⋅p), and O(n) respectively, with mmm being the maximum number of components per line and p being the maximum number of prerequisites. Among the data structures considered, vectors offer efficient appending but require linear time for searching, making them less suitable for large datasets where fast lookup is essential. Binary search trees (BSTs) provide efficient average-case performance but can degrade to linear time in unbalanced scenarios, resulting in a worst-case complexity of O(n2). In contrast, hash tables offer constant-time average-case performance for both insertions and lookups, leading to an overall efficient complexity of O(n). Thus, the hash table's superior performance in managing frequent lookups and validating prerequisites makes it the optimal choice for efficiently handling and processing course data.

Considering the advisor’s requirements for efficiently managing and validating course data, the hash table stands out as the most suitable data structure. Vectors, while simple and efficient for indexed access, suffer from linear search times and potential memory issues when handling large datasets. Binary Search Trees (BSTs) offer ordered data and efficient average-case performance when balanced, but their worst-case time complexity can degrade to linear if the tree becomes unbalanced, and maintaining balance adds implementation complexity. In contrast, hash tables provide constant-time average-case complexity for both insertions and lookups, making them highly effective for frequent prerequisite validation and course management. Despite their higher memory overhead and potential worst-case scenarios, hash tables generally offer superior performance and scalability for the given requirements, making them the optimal choice for efficiently handling course data.Top of Form

Based on the analysis of vector, hash table, and binary search tree (BST) data structures, I recommend using a hash table for managing and validating course data. The hash table's primary advantage lies in its average-case constant-time complexity O(1) for both insertions and lookups, which is crucial for efficiently handling frequent prerequisite checks and course data validation. This efficiency is particularly valuable given the need to quickly determine if prerequisites exist for each course, especially as the number of courses grows.

While vectors provide simplicity and fast indexed access, they are inefficient for search operations, requiring linear time O(n), which can become problematic with larger datasets. Binary search trees, though capable of maintaining sorted data with efficient O(logn) time complexity when balanced, suffer from potential performance degradation to O(n) if unbalanced, and their balancing complexities add to the implementation effort.

The hash table, despite its higher memory overhead and worst-case performance of O(n) due to potential collisions, generally offers superior performance in practical scenarios where good hashing strategies are employed. This makes it the most suitable choice for the task, balancing performance and scalability effectively. Thus, the hash table's efficient average-case operations align well with the requirements for managing and validating a potentially large set of course data, making it the recommended data structure for this application.

Bottom of Form