# CS 300 Pseudocode Document

## //Vector

STRUCT Course:

courseID (string)

courseName (string)

preList (List of strings)

FUNCTION txtParser(filePath):

DECLARE tempList as List of Course

OPEN file at filePath

FOR each line in file:

SPLIT line by commas into tokens

IF tokens.length >= 2:

CREATE newCourse of type Course

newCourse.courseID = tokens[0]

newCourse.courseName = tokens[1]

FOR i from 2 to tokens.length - 1:

ADD tokens[i] to newCourse.preList

ADD newCourse to tempList

RETURN tempList

FUNCTION validateList(courseList):

FOR each course in courseList:

FOR each prereq in course.preList:

IF NOT existsCourse(courseList, prereq):

RETURN false

RETURN true

FUNCTION existsCourse(courseList, courseID):

FOR each course in courseList:

IF course.courseID == courseID:

RETURN true

RETURN false

FUNCTION searchList(courseList, searchID):

FOR each course in courseList:

IF course.courseID == searchID:

RETURN course

RETURN new empty Course

FUNCTION printCourse(courseList, courseID):

SET course = searchList(courseList, courseID)

IF course.courseID is "":

DISPLAY "Course not found."

RETURN

DISPLAY "Course ID: " + course.courseID

DISPLAY "Course Name: " + course.courseName

IF course.preList is empty:

DISPLAY "No prerequisites."

ELSE:

DISPLAY "Prerequisites:"

FOR each prereq in course.preList:

DISPLAY prereq

FUNCTION printCourseList(courseList):

SORT courseList by courseID

FOR each course in courseList:

DISPLAY course.courseID + ": " + course.courseName

## // Hash Table

STRUCT Course:

courseID (string)

courseName (string)

preList (List of strings)

FUNCTION txtParser(filePath):

DECLARE tempTable as empty HashTable of Course

OPEN file at filePath

FOR each line in file:

SPLIT line by commas into tokens

IF tokens.length >= 2:

CREATE newCourse of type Course

newCourse.courseID = tokens[0]

newCourse.courseName = tokens[1]

FOR i from 2 to tokens.length - 1:

ADD tokens[i] to newCourse.preList

INSERT newCourse into tempTable using newCourse.courseID as key

RETURN tempTable

FUNCTION validateTable(courses):

FOR each course in courses:

FOR each prereq in course.preList:

IF NOT courses.containsKey(prereq):

RETURN false

RETURN true

FUNCTION searchCourse(courses, courseID):

IF courses.containsKey(courseID):

RETURN courses.get(courseID)

ELSE:

RETURN new empty Course

FUNCTION printCourse(courses, courseID):

SET course = searchCourse(courses, courseID)

IF course.courseID == "":

DISPLAY "Course not found."

RETURN

DISPLAY "Course ID: " + course.courseID

DISPLAY "Course Name: " + course.courseName

IF course.preList is empty:

DISPLAY "No prerequisites."

ELSE:

DISPLAY "Prerequisites:"

FOR each prereq in course.preList:

DISPLAY prereq

FUNCTION printCourseList(courses):

SET courseIDs = keys of courses

SORT courseIDs

FOR each id in courseIDs:

DISPLAY id + ": " + courses[id].courseName

**// Binary Search Tree Pseudocode**

## FUNCTION searchCourse(courseTree, courseNumber):

## FOR each course in courseTree DO

## IF course.courseNumber == courseNumber THEN

## PRINT course.courseNumber, course.title, course.description

## FOR each prereqNumber in course.prerequisites DO

## FOR each c in courseTree DO

## IF c.courseNumber == prereqNumber THEN

## PRINT c.courseNumber, c.title, c.description

## END IF

## END FOR

## END FOR

## END IF

## END FOR

## FUNCTION printInOrder(node):

## IF node is not null:

## CALL printInOrder(node.left)

## DISPLAY node.course.courseID + ": " + node.course.courseName

## CALL printInOrder(node.right)

**// Menu Pseudocode (Applies to All Structures)**

FUNCTION displayMenu():   
 While True:   
 DISPLAY "1. Load Data"

DISPLAY "2. Print Course List"

DISPLAY "3. Print Course Info"

DISPLAY "9. Exit"

PROMPT user for choice

SWITCH choice:

CASE 1:

CALL txtParser(filePath)

CASE 2:

CALL printCourseList(dataStructure)

CASE 3:

PROMPT user for course ID

CALL printCourse(dataStructure, courseID)

CASE 9:

EXIT program

DEFAULT:

DISPLAY "Invalid option."

END FUNCTION

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector | Hash | Binary |
| Load file and create objects | O(n) | O(n) | O(n log n) |
| Search for a course | O(n) | O(1) | O(log n) |
| Print all courses (sorted) | O(n log n) | O(n log n) | O(n) (in-order) |

N = number of courses in the file  
  
**Advantages and Disadvantages Analysis of Data Structures**

When determining the most appropriate data structure for implementing the academic advising program at ABCU, it is important to evaluate the runtime efficiency, ease of implementation, and suitability of each structure to the specific needs of the application. This program must load a list of courses, search for individual courses by ID, verify prerequisite validity, and print all courses in alphanumeric order. The three structures evaluated are a vector (list), a hash table, and a binary search tree (BST).

**Vector**

Vectors, or dynamic arrays, are one of the simplest data structures to implement. They allow for straightforward storage and retrieval of data in sequential order and are particularly easy to manage for small datasets. The main advantage of using a vector is its simplicity. Inserting elements is efficient (O(1) on average), and data can be iterated through easily when validating prerequisites or printing course information.

However, a vector does not provide efficient search capabilities. Searching for a specific course or checking if a prerequisite exists requires a linear search, resulting in a time complexity of O(n). Additionally, printing the course list in alphanumeric order requires sorting, which adds O(n log n) overhead. For large datasets or applications requiring frequent lookups, this can become a performance bottleneck.

**Hash Table**

Hash tables offer the fastest performance for direct access tasks such as course lookup and prerequisite validation. With an average-case search and insertion time complexity of O(1), hash tables are ideal for applications that require frequent key-based access. This structure allows advisors to search for a course by its ID or validate that a prerequisite exists almost instantaneously.

The primary limitation of hash tables is their lack of inherent order. Since course IDs are stored in hashed positions, additional steps are needed to extract and sort keys when displaying a list of courses. Despite this, the efficiency gained in search and validation operations makes the hash table the most effective choice for a system focused on fast user queries and data consistency.

**Binary Search Tree (BST)**

Binary search trees provide a balanced solution between ordered data storage and efficient access. A properly balanced BST maintains an average-case time complexity of O(log n) for insertion, search, and deletion. One of its biggest advantages in this context is the ability to retrieve data in sorted order using an in-order traversal without the need for additional sorting. This makes it well-suited for generating alphanumerically ordered course lists.

However, BSTs are more complex to implement and maintain. In particular, unbalanced trees can degrade performance to O(n), making them less reliable in scenarios with unpredictable or skewed input data. They also lack the near-instantaneous access time of hash tables for single-element lookups.

**Final Recommendation**

I recommend using the Hash Table as the primary data structure for this program. It offers the fastest course lookup and prerequisite validation times, both of which are key requirements for this advising application. While it doesn’t maintain order, the list of course keys can be sorted before displaying, which balances performance and ease of use. Its efficiency and constant-time access make it ideal for working with large sets of courses.