**Name: damian le**

**Date: 08/10/2025**

**Project 1:**

### Main Menu

This pseudocode manages the user interface and directs the program flow.

// Global data structures (choose one for implementation)

DECLARE coursesVector AS Vector of CourseObjects

DECLARE coursesHashTable AS HashTable

DECLARE coursesTree AS BinarySearchTree

DECLARE dataLoaded AS Boolean = FALSE

PROCEDURE MainMenu()

DECLARE choice AS Integer

WHILE choice IS NOT 9

DISPLAY "ABCU Course Advising Program"

DISPLAY "1. Load Course Data"

DISPLAY "2. Print Course List"

DISPLAY "3. Search for a Course"

DISPLAY "9. Exit"

GET choice

SWITCH choice

CASE 1:

// Assume LoadCourses() reads from a file and populates the chosen data structure

CALL LoadCourses()

dataLoaded = TRUE

DISPLAY "Data loaded successfully."

CASE 2:

IF dataLoaded IS TRUE THEN

// This call will be implemented differently based on the data structure

CALL PrintSortedCourseList()

ELSE

DISPLAY "Please load data first (Option 1)."

END IF

CASE 3:

IF dataLoaded IS TRUE THEN

DECLARE courseIdToFind AS String

DISPLAY "Enter course number (e.g., CSCI101):"

GET courseIdToFind

// SearchCourse() will find and print one course's details

CALL SearchCourse(courseIdToFind)

ELSE

DISPLAY "Please load data first (Option 1)."

END IF

CASE 9:

DISPLAY "Exiting program. Goodbye!"

DEFAULT:

DISPLAY "Invalid option. Please try again."

END SWITCH

END WHILE

END PROCEDURE

### Print Sorted Course List

#### Vector Implementation

PROCEDURE PrintSortedCourseList\_Vector(coursesVector)

// STEP 1: Sort the vector alphanumerically by course ID.

// The cost of a good sorting algorithm is O(N log N).

CALL Sort(coursesVector) // Assume a built-in or implemented sort function

// STEP 2: Iterate through the now-sorted vector and print.

// This loop runs N times.

FOR each course IN coursesVector

DISPLAY course.id, course.title

END FOR

END PROCEDURE

#### Hash Table Implementation

PROCEDURE PrintSortedCourseList\_HashTable(coursesHashTable)

// STEP 1: Create a temporary list to hold course data for sorting.

DECLARE sortableList AS List

// STEP 2: Iterate through the hash table and add each course to the list.

// This process visits N items.

FOR each course IN coursesHashTable

ADD course TO sortableList

END FOR

// STEP 3: Sort the temporary list.

// The cost of sorting is O(N log N).

CALL Sort(sortableList)

// STEP 4: Iterate through the sorted list and print.

// This loop runs N times.

FOR each course IN sortableList

DISPLAY course.id, course.title

END FOR

END PROCEDURE

#### Binary Search Tree (BST) Implementation

PROCEDURE PrintSortedCourseList\_Tree(coursesTree)

// Call the in-order traversal function starting at the root of the tree.

// The cost of visiting every node is O(N).

CALL InOrderTraversal(coursesTree.root)

END PROCEDURE

// Recursive helper function for traversal

PROCEDURE InOrderTraversal(node)

// If the node is null, we've reached the end of a branch.

IF node IS NOT NULL THEN

// 1. Traverse the left subtree

CALL InOrderTraversal(node.left)

// 2. Visit (print) the current node

DISPLAY node.course.id, node.course.title

// 3. Traverse the right subtree

CALL InOrderTraversal(node.right)

END IF

END PROCEDURE

# Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N) | O(log N) |
| **Search** | O(n) | O(1) – O(N) | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N) | O(N) |

# Advantage Analysis

When you're picking how to store your data, you have to think about what you'll be doing with it most of the time. For this program, we need to load courses, look up a single course, and print the whole list in order.

A simple list, or **vector**, is really fast for that first step of loading the data. But after that, it's not very good. If you want to find one course, you have to look through every single item until you find it. And to print a sorted list, you have to do a whole new sort every single time, which is very slow. It’s just not a good fit for the important jobs.

So, it really comes down to two better options: the **hash table** and the **binary search tree**.

The cool thing about a **binary search tree** is that it keeps the courses in order all the time. This makes printing the whole list really fast. Finding one course is also pretty quick. The main problem is that it's a bit slower to load everything at the start, and if the data comes in already sorted, the tree can get lopsided and slow way down.

The **hash table**, on the other hand, has one huge thing going for it: it's incredibly fast for searching. When an advisor needs to look up one course, a hash table can find it almost instantly. The downside is that the data is not kept in any order. So, if you want to print the sorted list, you have to pull everything out and sort it from scratch, which takes time.

In the end, you have to choose what's best for the person using the program. An advisor is going to spend all day looking up one course at a time, and they need that to be fast. For that reason, the **hash table is the best choice**. You give up some speed on printing the whole list, but that's a task they won't do very often. It's worth it to make the most common part of the job as fast as possible.

# Recommendation

the **hash table is** the best choice for this advising program.

justification: The most important job for an advisor using this software is looking up a single course to check its prerequisites or title, and they'll be doing this over and over all day. The hash table does this one job faster than any other option, providing a nearly instant response (O(1)). This makes the program feel responsive and efficient for its main purpose.