CS-300 Project One

Menu Pseudocode

BEGIN

DECLARE dataStructure

DECLARE userChoice AS INTEGER = 0

WHILE userChoice != 9 DO

DISPLAY "Welcome to ABCU Course Planner"

DISPLAY "--------------------------------"

DISPLAY "1. Load data from file"

DISPLAY "2. Print course list (A-Z by course number)"

DISPLAY "3. Print one course (details and prerequisites)"

DISPLAY "9. Exit program"

DISPLAY "--------------------------------"

DISPLAY "Enter your choice: "

READ userChoice

IF userChoice == 1 THEN

DISPLAY "Enter file name (e.g., courses.txt): "

READ filename

CALL LoadDataFromFile(filename) → dataStructure

DISPLAY "File loaded successfully."

ELSE IF userChoice == 2 THEN

IF dataStructure is empty THEN

DISPLAY "No data loaded. Please load the file first."

ELSE

DISPLAY "Course List (A–Z):"

CALL PrintAllCourses(dataStructure)

ENDIF

ELSE IF userChoice == 3 THEN

IF dataStructure is empty THEN

DISPLAY "No data loaded. Please load the file first."

ELSE

DISPLAY "Enter course number (e.g., CS200): "

READ courseNum

CALL PrintCourseDetails(dataStructure, courseNum)

ENDIF

ELSE IF userChoice == 9 THEN

DISPLAY "Goodbye!"

ELSE

DISPLAY "Invalid selection. Please choose 1, 2, 3, or 9."

ENDIF

DISPLAY newline

ENDWHILE

END

Print Course List Pseudocode

PROCEDURE PrintAllCourses(dataStructure)

IF dataStructure is empty THEN

DISPLAY "No courses available."

RETURN

ENDIF

// For Vector

IF using Vector THEN

SORT vector by course number (A–Z)

FOR each course IN vector DO

DISPLAY course.number + ", " + course.title

IF course.prereqs is empty THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + join(course.prereqs, ", ")

ENDIF

ENDFOR

ENDIF

// For Hash Table

IF using Hash Table THEN

GET all keys from hash table

SORT keys alphanumerically

FOR each key IN sorted keys DO

RETRIEVE course = hashTable.Get(key)

DISPLAY course.number + ", " + course.title

IF course.prereqs is empty THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + join(course.prereqs, ", ")

ENDIF

ENDFOR

ENDIF

// For Binary Search Tree

IF using Binary Search Tree THEN

PERFORM in-order traversal of BST

FOR each node visited DO

DISPLAY node.course.number + ", " + node.course.title

IF node.course.prereqs is empty THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + join(node.course.prereqs, ", ")

ENDIF

ENDFOR

ENDIF

END PROCEDURE

Vector Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Read each line of file | 1 | n | n |
| Split into tokens | 1 | n | n |
| Create course object | 1 | n | n |
| Append to vector | 1 | n | n |
| Search through vector | 1 | n | n |
| Total Cost |  |  | 5n |
| Runtime |  |  | 0(n) |

**Advantages:**

* Easy to implement
* Preserves insertion order
* Great for sequential reads

**Disadvantages:**

* Searching and insertion in sorted order are O(n)
* Duplicates require manual checking

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| For each non-blank line in file | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Validate minimum tokens | 1 | n | n |
| Create course object | 1 | n | n |
| Compute hash & insert | 1 | n | n |
| Total cost |  |  | 5n |
| Runtime |  |  | 0(n) |

**Advantages:**

* Fast average lookups (O(1))
* Efficient for large data sets

**Disadvantages:**

* Not stored in order
* Collisions may slow performance
* Requires more memory than a vector

**Binary Search Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| For each non-blank line in file | 1 | n | n |
| Split line into tokens | 1 | n | n |
| Validate minimum tokens | 1 | n | n |
| Create course object | 1 | n | n |
| BST insert | 1 | n\*log n | n\*log n |
| Total cost |  |  | n\*log n + 3 |
| Runtime |  |  | 0(n log n) |

**Advantages:**

* Maintains sorted order
* Efficient searching and inserting (O(log n) average)

**Disadvantages:**

* Degrades to O(n) if unbalanced
* More complex to implement

After comparing the vector, hash table, and binary search tree implementations, the hash table is the most efficient data structure for this course planner program. It provides constant-time average lookups and insertions, which makes retrieving course information by course number extremely fast. While the vector is simple to implement, it becomes inefficient for searching large datasets because it requires linear time. The binary search tree improves search efficiency and naturally stores data in sorted order, but it can degrade to linear performance if the tree becomes unbalanced. The hash table offers the best balance of speed and practicality for this application, with the option to sort the keys only when displaying the full course list.