Project One

CS300: DSA Analysis and Design

Menu:

FUNCTION MainMenu()  
 DECLARE loaded <- false  
 LOOP  
 PRINT "1. Load Data"  
 PRINT "2. Print Course List (alphanumeric)"  
 PRINT "3. Print Course Info"  
 PRINT "9. Exit"  
 READ choice  
 IF choice = 1 THEN  
 loaded <- LoadDataForSelectedStructure() // vector, hash, or tree  
 ELSE IF choice = 2 THEN  
 IF NOT loaded THEN PRINT "Load data first." ELSE PrintCourseList()  
 ELSE IF choice = 3 THEN  
 IF NOT loaded THEN PRINT "Load data first."  
 ELSE READ target; PrintCourseInfo(target)  
 ELSE IF choice = 9 THEN  
 BREAK  
 ELSE  
 PRINT "Invalid selection."  
 END IF  
 END LOOP  
END FUNCTION

Helpers:

STRUCT Course {

courseNumber: String,

title: String,

prerequisites: List<String>

}  
  
// CSV helpers  
FUNCTION splitCSV(line): List<String>  
FUNCTION trimAll(tokens): List<String>  
FUNCTION filterPrequisites(Tokens): List<String>

File I/O, and validation: // CSV Parse

FUNCTION LoadCoursesFromFile(path) // RETURNS Vector<Course>  
 file <- openFile(path)  
 IF file = null THEN PRINT "File error"; RETURN empty Vector<Course> END IF  
 lineNo <- 0  
 courses <- empty Vector<Course>  
 WHILE NOT EOF(file) DO  
 lineNo <- lineNo + 1  
 line <- readline(file)  
 parts <- trimAll(splitCSV(line))  
 IF len(parts) < 2 THEN PRINT "Line " + lineNo + " missing id or title"; CONTINUE END IF  
 c <- Course()  
 c.courseNumber <- parts[0]  
 c.title <- parts[1]  
 c.prerequisites <- []  
 IF len(parts) > 2 THEN  
 FOR i FROM 2 TO len(parts) - 1 DO  
 p <- parts[i]  
 IF p != "" THEN APPEND p TO c.prerequisites END IF  
 END FOR  
 END IF  
 APPEND c TO courses  
 END WHILE  
 IF NOT validatePrerequisites(courses) THEN RETURN empty Vector<Course> END IF  
 RETURN courses  
END FUNCTION  
  
FUNCTION validatePrerequisites(courses: Vector<Course>)

// RETURNS Boolean  
 ids <- Set<String>()  
 FOR each c IN courses DO INSERT c.courseNumber INTO ids END FOR  
 FOR each c IN courses DO

IF c.courseNumber IN c.prerequisites THEN PRINT "Self prerequisite: " + c.courseNumber; RETURN false END IF  
 seen <- Set<String>()  
 FOR each p IN c.prerequisites DO

IF p IN seen THEN PRINT "Duplicate prerequisite " + p + " for " + c.courseNumber;

RETURN false

END IF  
 INSERT p INTO seen  
 IF p NOT IN ids THEN PRINT "Missing course for prerequisite " + p + " (in " + c.courseNumber + “)";

RETURN false

END IF  
 END FOR  
 END FOR  
 RETURN true  
END FUNCTION

Vector Implementation:

GLOBAL vecCourses <- empty Vector<Course>  
  
FUNCTION LoadData\_Vector(path) // RETURNS Boolean  
 vecCourses <- LoadCoursesFromFile(path)  
 RETURN len(vecCourses) > 0  
END FUNCTION  
  
FUNCTION PrintCourseInfo\_Vector(target: String)  
 FOR each c IN vecCourses DO  
 IF toUpper(c.courseNumber) = toUpper(target) THEN  
 PRINT c.courseNumber + ": " + c.title  
 IF len(c.prerequisites) = 0 THEN PRINT "Prerequisites: none"  
 ELSE PRINT "Prerequisites: " + join(c.prerequisites, ", ")  
 END IF  
 RETURN  
 END IF  
 END FOR  
 PRINT "Course not found."  
END FUNCTION  
  
FUNCTION PrintCourseList\_Vector()  
 SORT vecCourses BY c.courseNumber ASC  
 FOR each c IN vecCourses DO PRINT c.courseNumber + ": " + c.title END FOR  
END FUNCTION

Hash Table implementation:

GLOBAL tableSize <- 181

GLOBAL courseTable <- Array<List<Course>>(tableSize, each empty)

// Basic hash function using simple character sum (can be replaced with something stronger)

FUNCTION Hash(key: String) // RETURNS Integer

hashValue <- 0

FOR each ch IN key DO

hashValue <- hashValue + ASCII(ch)

END FOR

RETURN hashValue MOD tableSize

END FUNCTION

FUNCTION LoadData\_Hash(path: String) RETURNS Boolean

courses <- LoadCoursesFromFile(path)

IF len(courses) = 0 THEN RETURN false END IF

// Clear table before insert

FOR i FROM 0 TO tableSize - 1 DO

courseTable[i] <- empty List<Course>

END FOR

FOR each c IN courses DO

idx <- Hash(c.courseNumber)

APPEND c TO courseTable[idx]

END FOR

RETURN true

END FUNCTION

FUNCTION PrintCourseInfo\_Hash(target: String)

idx <- Hash(target)

bucket <- courseTable[idx]

FOR each c IN bucket DO

IF toUpper(c.courseNumber) = toUpper(target) THEN

PRINT c.courseNumber + ": " + c.title

IF len(c.prerequisites) = 0 THEN

PRINT "Prerequisites: none"

ELSE

PRINT "Prerequisites: " + join(c.prerequisites, ", ")

END IF

RETURN

END IF

END FOR

PRINT "Course not found."

END FUNCTION

FUNCTION PrintCourseList\_Hash()

ids <- empty List<String>

FOR i FROM 0 TO tableSize - 1 DO

FOR each c IN courseTable[i] DO

APPEND c.courseNumber TO ids

END FOR

END FOR

SORT ids ASC

FOR each id IN ids DO

idx <- Hash(id)

bucket <- courseTable[idx]

FOR each c IN bucket DO

IF c.courseNumber = id THEN

PRINT c.courseNumber + ": " + c.title

BREAK

END IF

END FOR

END FOR

END FUNCTION

Binary Search Tree implementation:

STRUCT Node {

course: Course,

left: Node,

right: Node

}

GLOBAL root <- null

FUNCTION InsertBST(node: Node, c: Course) // RETURNS Node

IF node = null THEN

newNode <- Node()

newNode.course <- c

newNode.left <- null

newNode.right <- null

RETURN newNode

END IF

IF toUpper(c.courseNumber) < toUpper(node.course.courseNumber) THEN

node.left <- InsertBST(node.left, c)

ELSE

node.right <- InsertBST(node.right, c)

END IF

RETURN node

END FUNCTION

FUNCTION LoadData\_BST(path: String) RETURNS Boolean

courses <- LoadCoursesFromFile(path)

IF len(courses) = 0 THEN RETURN false END IF

root <- null

FOR each c IN courses DO

root <- InsertBST(root, c)

END FOR

RETURN true

END FUNCTION

FUNCTION SearchBST(node: Node, key: String)

// RETURNS Course or null

IF node = null

RETURN null

END IF

current <- toUpper(node.course.courseNumber)

target <- toUpper(key)

IF current = target THEN

RETURN node.course

ELSE IF target < current THEN

RETURN SearchBST(node.left, key)

ELSE

RETURN SearchBST(node.right, key)

END IF

END FUNCTION

FUNCTION PrintCourseInfo\_BST(target: String)

result <- SearchBST(root, target)

IF result = null THEN

PRINT "Course not found."

ELSE

PRINT result.courseNumber + ": " + result.title

IF len(result.prerequisites) = 0 THEN

PRINT "Prerequisites: none"

ELSE

PRINT "Prerequisites: " + join(result.prerequisites, ", ")

END IF

END IF

END FUNCTION

FUNCTION InOrderTraversal(node: Node)

IF node = null THEN RETURN

InOrderTraversal(node.left)

PRINT node.course.courseNumber + ": " + node.course.title

InOrderTraversal(node.right)

END FUNCTION

FUNCTION PrintCourseList\_BST()

InOrderTraversal(root)

END FUNCTION

Runtime Analysis / Recommendation:

The hash table is best for fast searches, the BST is best for ordered output, and the vector is easiest to implement but scales poorly with large datasets.

| Operation | Vector | Hash Table | Binary Search Tree |
| --- | --- | --- | --- |
| Insert all courses | O(N) push\_back | O(N) avg | O(N log N) avg |
| Lookup course by ID | O(N) | O(1) avg | O(log N) avg |
| List Courses (Alphanumeric) | O(N log N) | O(N log N) | O(N) |
| Print all courses | O(N) | O(N) | O(N) |
|  |  |  |  |
| Memory Usage | O(N) | O(N+B) | O(N) |

*Assuming: B = number of hash buckets; Total hash memory is for courses + empty buckets*

For this use case I recommend a hash table. Courses are identified by a unique course number, making them great for key based access. A hash table provides constant lookup times on average. This is good for retrieving a specific course and its prerequisites efficiently (required in Option 3). Because prerequisites reference other course IDs, using a hash table ensures that these dependencies can be resolved quickly without scanning the entire dataset. This structure supports clear mapping between course IDs and their associated data, to include prerequisite validation and display.