# CS 300 Project One

Vector:

STRUCT Course

courseID: STRING

courseName: STRING

preReqs: LIST OF STRING

END STRUCT

MAIN

DECLARE courses AS VECTOR OF Course

REPEAT

DISPLAY menu:

1 - Load data

2 - Print course list

3 - Search course

9 - Exit

READ choice

CASE choice OF

1:

PROMPT for filePath

CALL loadCourses(filePath, courses)

CALL validateCourses(courses)

2:

CALL sortCourses(courses)

CALL printCourses(courses)

3:

PROMPT for courseID

CALL searchCourse(courses, courseID)

9:

DISPLAY "Exiting."

END CASE

UNTIL choice == 9

END MAIN

FUNCTION loadCourses(filePath, courses)

OPEN file

IF file fails THEN DISPLAY error; RETURN

FOR each line in file:

SPLIT by commas INTO parts

IF parts.length < 2 THEN CONTINUE

CREATE Course c with ID = parts[0], Name = parts[1]

FOR i = 2 TO parts.length-1

APPEND parts[i] to c.preReqs

ENDFOR

APPEND c to courses

CLOSE file

END FUNCTION

FUNCTION sortCourses(courses)

SORT courses BY courseID ascending

END FUNCTION

FUNCTION printCourses(courses)

FOR EACH course IN courses

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

END FUNCTION

FUNCTION searchCourse(courses, id)

FOR EACH course IN courses

IF course.courseID == id THEN

DISPLAY course info and prerequisites

RETURN

DISPLAY "Course not found"

END FUNCTION

FUNCTION validateCourses(courses)

FOR EACH course IN courses

FOR EACH prereq IN course.preReqs

IF prereq NOT FOUND in courses

DISPLAY warning

END FUNCTION

Hash:

FUNCTION processCourses(filename, searchID)

DEFINE HashTable courseTable

OPEN file

IF file fails THEN DISPLAY error; RETURN

FOR each line:

IF line empty THEN CONTINUE

SPLIT into parts

IF parts.length < 2 THEN CONTINUE

CREATE Course with ID, Name, prereqs

INSERT into courseTable with key = ID

FOR each course IN courseTable:

FOR prereq IN course.prereqs:

IF prereq NOT IN courseTable THEN DISPLAY warning

IF searchID NOT IN courseTable THEN DISPLAY not found

ELSE:

DISPLAY course info and valid prereqs

END FUNCTION

Tree:

FUNCTION loadCourses(filePath)

DECLARE courseTree AS BinarySearchTree

OPEN file

IF file fails THEN DISPLAY error; RETURN

FOR each line:

SPLIT into parts

IF parts.length < 2 THEN CONTINUE

CREATE Course with ID, Name, prereqs

INSERT into courseTree

FOR EACH course IN courseTree

FOR EACH prereq IN course.prereqs

IF NOT FOUND IN tree THEN DISPLAY warning

END FUNCTION

FUNCTION searchAndPrint(courseTree, courseID)

course = courseTree.search(courseID)

IF NULL THEN DISPLAY not found; RETURN

DISPLAY course info and prereqs (searched via tree)

END FUNCTION

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector | Hash Table | Binary Search Tree |
| Load File (n lines) | O(n) | O(n) | O(n log n) |
| Insert Course | O(1) append | O(1) average | O(log n) average |
| Validate Prereqs | O(n²) | O(n·1) = O(n) | O(n log n) |
| Search Course by ID | O(n) | O(1) avg | O(log n) avg |
| Print Sorted List | O(n log n) | O(n log n) | O(n) (in-order) |

Advantages

Each of the three data structures, vector, hash table, and binary search tree, offers distinct advantages and disadvantages depending on data access methods. Vectors facilitate fast data loading, adding new course entries in constant time, resulting in an overall load time of O(n). However, they do not retain order, requiring O(n log n) for sorting, with searches taking O(n) per operation, which can lead to diminished performance with larger datasets. Hash tables shine with average-case performance for insertions and lookups at Θ(1), making them suitable for frequent operations. Nevertheless, their worst-case performance can reach O(n) due to collisions, and generating a sorted course list necessitates O(n log n) for key extraction and sorting. Meanwhile, binary search trees, particularly self-balancing types like AVL or Red-Black Trees, preserve sorted order and provide O(log n) for insertions and searches, while offering O(n) for ordered listings through in-order traversal. They strike a balance between speed and structure, eliminating the need for separate storage for sorted and unsorted data. Ultimately, the selection hinges on balancing data loading, search frequency, and the necessity for sorted output. Hash tables excel for frequent searches with a single data load, whereas binary search trees are preferable for maintaining ordered listings.

Recommendation

Considering the advising system's context and anticipated usage at ABCU, it seems practical to conclude that course data will only be loaded into memory occasionally. While printing the entire list of courses may be infrequent, the system frequently needs to support queries for individual courses and their prerequisites. Given this usage pattern, a hash table is the most suitable data structure.

The hash table offers average-case constant time performance for both insertions and lookups, making it ideal for quick and repeated searches by course ID—a fundamental requirement for the system. Although additional functionality is needed to sort keys for display purposes, this step is only necessary in specific situations and does not hinder the performance benefits of frequent lookups. To ensure optimal operation, it is crucial to develop a high-quality hash function and carefully select the table size to minimize collisions.

In summary, the hash table strikes the best balance between performance and implementation complexity for this application, effectively meeting core functional requirements while maintaining high responsiveness as the course catalog expands.