MENU

Main Function()

Create a variable choice of type Integer

Create a data structure named courseData (Vector, Hash Table, or Tree based on implementation)

Set courseData to an empty data structure

While True:

Print "Menu Options:"

Print "1: Load file data into the data structure"

Print "2: Print an alphanumerically ordered list of all Computer Science courses"

Print "3: Print course title and prerequisites for a course"

Print "9: Exit the program"

Get user input for choice

If choice == 1:

Call loadFileData() with courseData

Else If choice == 2:

Call printSortedCourses() with courseData

Else If choice == 3:

Print "Enter course ID to search:"

Get user input for courseID

Call printCourseDetails() with courseData and courseID

Else If choice == 9:

Print "Exiting program."

Break

Else:

Print "Invalid choice. Please try again."

End While

End

loadFileData Function

Function loadFileData(DataStructure courseData)

Print "Enter CSV file path:"

Get user input for filePath

If filePath is empty:

Set filePath to default path

Call txtParser(filePath) and store the result in courseData

Print "Data loaded successfully."

End

printSortedCourses Function

Function printSortedCourses(DataStructure courseData)

Create a variable sortedCourses of type List<Course>

If courseData is empty:

Print "Data not loaded. Please load data first."

Return

If courseData is a Vector:

Sort courseData alphanumerically by courseID and store in sortedCourses

Else If courseData is a Hash Table:

Convert courseData to a list, sort alphanumerically by courseID, and store in sortedCourses

Else If courseData is a Tree:

Perform an in-order traversal and store the result in sortedCourses

Print "Computer Science Courses in Alphanumeric Order:"

For each course in sortedCourses:

Print course.courseID, course.courseName

End

printCourseDetails Function

Function printCourseDetails(DataStructure courseData, String courseID)

Create a variable course of type Course

If courseData is empty:

Print "Data not loaded. Please load data first."

Return

If courseData is a Vector:

Set course to the result of searchList(courseID)

Else If courseData is a Hash Table:

Set course to the result of searchTable(courseID)

Else If courseData is a Tree:

Set course to the result of searchCourse(courseData, courseID)

If course is null:

Print "Course not found."

Else:

Print "Course ID: " + course.courseID

Print "Course Name: " + course.courseName

If course.preCount > 0:

Print "Prerequisites:"

For each prerequisite in course.preList:

Print prerequisite

End

## Vector Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 2 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| print the prerequisite course information | 2 | n | n |
| \*\*Total Cost\*\* |  |  | \*\*6n + 1\*\* |
| \*\*Runtime\*\* |  |  | \*\*O(n)\*\* |

## Hash Table Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 2 | n | 2n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 1 | 1 | 1 |
| for each prerequisite of the course | 2 | n | 2n |
| print the prerequisite course information | 4 | n | 4n |
| \*\*Total Cost\*\* |  |  | \*\*9n + 1\*\* |
| \*\*Runtime\*\* |  |  | \*\*O(n)\*\* |

## Tree Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for all courses | 1 | n | n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 2 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| print the prerequisite course information | 4 | n | 4n |
| \*\*Total Cost\*\* |  |  | \*\*8n + 1\*\* |
| \*\*Runtime\*\* |  |  | \*\*O(n)\*\* |

EVALUATION

The evaluation of the three data structures—vector, hash table, and tree—highlights their respective strengths and weaknesses for the requirements of the advising program. When analyzing the runtime, reading the file and creating course objects is O(n) for all structures, as each line must be processed. Loading data into a vector is efficient with O(1) append operations, resulting in a total runtime of O(n). However, searching and sorting within a vector require O(n log n) or O(n²) time, making it less optimal for frequent lookups or sorting. Hash tables, on the other hand, provide O(1) average time complexity for insertions and lookups, offering a significant advantage for direct access to specific courses. Sorting, while not inherently supported, can be achieved by converting the hash table to a list, requiring O(n log n). Trees, particularly binary search trees, maintain a naturally sorted order with O(log n) insertions and lookups and O(n) for in-order traversal, which is ideal for printing sorted courses. However, trees introduce additional complexity with balancing requirements and slower lookups compared to hash tables.

In terms of memory usage, all three data structures require O(n) space, with vectors being the simplest and hash tables and trees incurring some overhead for hash functions or tree pointers, respectively. While vectors are easy to implement and effective for sequential operations, they lack efficiency for dynamic searches and sorting. Trees are advantageous for maintaining order but are slower than hash tables for frequent operations. Therefore, the hash table is the most suitable data structure for this program. Its O(1) average runtime for insertions and lookups aligns well with the program’s need for efficient searches, and occasional sorting can be handled effectively with conversion to a list. Hash tables balance performance and simplicity, making them the optimal choice for the advising program.