Johnathan Dumke

CS 300

06-9-2025

Project One

Pseudocode:

Main Function() // Menu Loop

Read command-line arguments

Store argument as CSV file path

If no command-line arguments, set filePath = "courses.csv"

Loop while menuChoice ≠ 9

Display menu options

Prompt user for menuChoice // What action to perform

Prompt user for dataChoice // Select data structure

Validate menuChoice

If not in [1–4, 9], display error and continue

If menuChoice == 1

// Load CSV data into selected data structure

If BinarySearchTree

Call loadCoursesFromFile(filePath) → store in bst

Else if vector

Call loadCoursesFromFile(filePath) → store in courseList

Else if HashTable

Call loadCoursesFromFile(filePath) → store in courseTable

Display number of courses loaded

If menuChoice == 2

// Validate data structure

If BinarySearchTree

Call validateTree(bst)

Else if vector

Call validateList(courseList)

Else if HashTable

Call validateTable(courseTable)

If menuChoice == 3

// Search and print course

Prompt user to enter courseID → userSearch

If BinarySearchTree

Call printCourseTree(userSearch)

Else if vector

Call printCourseList(userSearch)

Else if HashTable

Call printCourseTable(userSearch)

If menuChoice == 4

// Print all courses in alphanumeric order

If BinarySearchTree

Call printTree()

Else if vector

Call sortList(courseList, 0, courseList.size - 1)

Call printList(courseList)

Else if HashTable

Call sortTable(courseTable)

Call printTable(courseTable)

If menuChoice == 9

Display "Goodbye"

Exit application

Course Structure

STRUCT: Course

courseID

courseName

preCount

preList (list of strings)

CONSTRUCTOR Course():

courseID = ""

courseName = ""

preCount = 0

**Binary Search Tree**

plaintext

CopyEdit

CLASS: BinaryTree

STRUCT: Node

Course course

Node left

Node right

BinaryTree:

root

+insert()

+search()

+inOrderTraversal()

+printTree()

**Hash Table**

plaintext

CopyEdit

CLASS: HashTable

STRUCT: Bucket

Course course

String key

Bucket next

HashTable:

+hash()

+insert()

+printTable()

List of Buckets: hashTable

FUNCTION: loadCoursesFromFile(filePath)

OPEN file at filePath FOR reading

FOR EACH line IN file:

SPLIT line BY commas INTO tokens

IF number of tokens < 2:

DISPLAY "Formatting error: invalid line"

CONTINUE

SET courseID = tokens[0]

SET courseName = tokens[1]

SET preList = tokens[2 to end]

CREATE new Course with courseID, courseName, preList

INSERT Course into selected data structure

CLOSE file

**Sorting and Printing Functions**

plaintext

CopyEdit

FUNCTION: sortList(courseList, low, high)

IF low ≥ high, RETURN

SET partitionIndex = partition(courseList, low, high)

CALL sortList(courseList, low, partitionIndex)

CALL sortList(courseList, partitionIndex + 1, high)

FUNCTION: partition(courseList, low, high)

SET pivot = middle element

WHILE low ≤ high:

WHILE courseList[low] < pivot, INCREMENT low

WHILE courseList[high] > pivot, DECREMENT high

IF low ≤ high:

SWAP courseList[low], courseList[high]

INCREMENT low, DECREMENT high

RETURN high

FUNCTION: printList(courseList)

FOR EACH course IN courseList:

DISPLAY courseID, courseName

IF course.preCount > 0:

DISPLAY prerequisites from preList

FUNCTION: printTree(node)

IF node is null, RETURN

CALL printTree(node.left)

DISPLAY courseID, courseName

IF course.preList is not empty:

DISPLAY prerequisites

CALL printTree(node.right)

FUNCTION: printTable(courseTable)

FOR EACH bucket IN courseTable:

WHILE bucket is not null:

DISPLAY courseID, courseName

IF preList is not empty:

DISPLAY prerequisites

bucket = bucket.next

Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Task | Vector | Hash Table | Binary Tree |
| Loading Data | O(1) | O(1) – O(n) | O(log n) – O(n) |
| Searching | O(n) | O(1) – O(n) | O(log n) – O(n) |
| |  | | --- | | Sorting/Printing |  |  | | --- | |  | | O(n log n) | O(n) | O(n) |

**Analysis**

All three have their advantages and disadvantages. The vector can be the easiest when it comes to appending data. But search is linear (O(n)) and sorting is the slowest at O(n log n). The hash table has the fastest lookup time on average (O(1)), ideal for frequent course searches. The downside is that its performance depends on a good hash function and may degrade to O(n) in worst-case scenarios due to collisions. The last one is binary tree and gives predictable log-time performance and keeps data naturally sorted through in-order traversal.If it becomes unbalanced the performance drops to O(n).

**Recommendation**

Based on the use case and the three options I believe the hash table it the best choice. It has the fastest average search time. And is ideal for students or advisors enquiring course requirements often. If a hash function is well designed it will help avoid performance degradation, keeping operations near O(1). The Binary search tree is a strong alternative when sorted order is a priority, but it carries more complexity and performance risk if unbalanced. And for vector it is best for simplicity but not for performance.