Martin Vaca

CS300

08/11/24

Project One

2. Menu

Main Function():

Read command line arguments

Store the argument as CSV file path

If no command line arguments are provided, load the default CSV file path

Loop while the choice is not '9':

Display the menu options

Get user input and store in menuChoice (user's action choice)

Get user input and store in dataChoice (selected data structure)

Validate user input:

If menuChoice is not 1-4 or 9, display an error message

If menuChoice equals '1':

// Load data from CSV into the selected data structure

If dataChoice is BinarySearchTree:

Call loadCourses and store CSV data in BinarySearchTree (bst)

Else If dataChoice is Vector:

Call loadCourses and store CSV data in Vector (courseList)

Else If dataChoice is HashTable:

Call loadCourses and store CSV data in HashTable (courseTable)

Display the number of records loaded from the CSV file

If menuChoice equals '2':

// Validate and print the course list

If dataChoice is BinarySearchTree:

Call validateTree(bst)

Else If dataChoice is Vector:

Call validateList(courseList)

Else If dataChoice is HashTable:

Call validateTable(courseTable)

If menuChoice equals '3':

// Search for and print a specific course's information

Get user input for the course number to search

If dataChoice is BinarySearchTree:

Call printCourseTree(bst, userSearch)

Else If dataChoice is Vector:

Call printCourseList(courseList, userSearch)

Else If dataChoice is HashTable:

Call printCourseTable(courseTable, userSearch)

If menuChoice equals '4':

// Print all courses in alphanumeric order

If dataChoice is BinarySearchTree:

Call printTree(bst)

Else If dataChoice is Vector:

Call sortList(courseList)

Call printList(courseList)

Else If dataChoice is HashTable:

Call sortTable(courseTable)

Call printTable(courseTable)

If menuChoice equals '9':

Exit the application

Print "Goodbye"

3.

Vector:

function sortList(Vector<Course> courseList):

// Quick sort implementation to sort courses by courseID

function partition(Vector<Course> courseList, int low, int high):

// Partition function for quick sort

function printList(Vector<Course> courseList):

for each Course in courseList:

print courseID, courseName

for each prerequisite in preList:

print prerequisite

Hash:

Class HashTable:

struct Bucket:

Course courseData

unsigned int key

Bucket\* nextPointer

List<Bucket> hashTable

HashTable():

// Initialize the hash table

unsigned int hash(String key):

// Hash function to calculate the hash value

void printTable():

// Implement method to print the entire hash table

BST:

Class BinaryTree:

struct Node:

Course courseData

Node\* rightPointer

Node\* leftPointer

Node\* root

BinaryTree():

root = null

void printTree():

// Implement in-order traversal to print the tree

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector | Hash Table | BST |
| Loading Data | O(1) per insertion, O(n) total | O(1) per insertion, O(1) to O(n) depending on collisions | O(log n) per insertion, O(n log n) total |
| Searching | O(n) | O(1) (O(n) in worst-case with many collisions) | O(log n) (O(n) if unbalanced) |
| **Sort/Print** | O(n log n) for sorting, O(n) for printing | O(n log n) for sorting keys, O(n) for printing | O(n) for in-order traversal |

**Advantages and disadvantages**

Each data structure—vector, hash table, and binary search tree (BST)—offers unique advantages depending on the application's needs. Vectors are simple and efficient for small datasets, with quick insertion at the end, but they struggle with searching and sorting, as these operations require linear time (O(n)) and O(n log n) time. This makes vectors less ideal when frequent searching or maintaining order is important.

Hash tables excel in fast data retrieval with an average-case O(1) time for insertion and search, making them ideal for quick lookups. However, they can suffer from performance issues due to collisions, which can degrade efficiency to O(n) in the worst case. Hash tables do not maintain order among elements. Binary search trees provide a good balance with O(log n) time for insertion and search and they naturally maintain sorted order, making them suitable for applications requiring ordered data. However, BSTs can degrade to O(n) if they become unbalanced, which requires additional effort to manage.

**Recommendation**

I recommend using the hash table for the implementation due to its superior performance in search and retrieval operations, with an average-case time complexity of O(1) for both insertion and search. This efficiency is crucial for frequent lookups, which aligns with the advisor's requirements. While hash tables may experience collisions that could degrade performance to O(n), the overall benefits do outweigh the risks, especially compared to vectors, which are less efficient in searching and sorting, and binary search trees, which can become inefficient if unbalanced. The hash table offers the best balance of speed and functionality for the application.