Gentian Hoxha

[Gentian.hoxha@snhu.edu](mailto:Gentian.hoxha@snhu.edu)

Mod 6

04/21/2024

**Main Function()**

**// Initializes the main menu loop  
Read command line arguments:**

* **Store the argument as the path for the CSV file**
* **If no arguments are provided, use the default CSV file path**

**Main Menu Loop  
Repeat until the choice is ‘9’:**

* **Display the menu**
* **Prompt for user input to select a program action (stored in menuChoice)**
* **Prompt for user input to select the data structure to use (stored in dataChoice)**
* **Validate the user input:**
  + **If the choice is not within '1-4' or '9', display an error message**

**Operations Based on User Choice:**

* **If choice is ‘1’:**
  + **Depending on the selected data structure:**
    - **BinarySearchTree: Load CSV data into a BinarySearchTree object**
    - **Vector: Load CSV data into a vector courseList**
    - **HashTable: Load CSV data into a HashTable object, ordering the entries in ascending order**
  + **Display the number of records loaded**
* **If choice is ‘2’:**
  + **Validate the data structure:**
    - **BinarySearchTree: Validate the tree structure**
    - **Vector: Validate the list**
    - **HashTable: Validate the table**
* **If choice is ‘3’:**
  + **Search for a specific course based on user input and display the results:**
    - **BinarySearchTree: Search and print from the tree**
    - **Vector: Search and print from the list**
    - **HashTable: Search and print from the table**
* **If choice is ‘4’:**
  + **Print all courses in alphabetical order:**
    - **BinarySearchTree: Print directly from the tree**
    - **Vector: Sort the list, then print**
    - **HashTable: Sort the table, then print**
* **If choice is ‘9’:**
  + **Exit the application**
* **Display "Good bye" upon exiting**

**End of Main Function()**

**Data Structures and Associated Methods:  
Struct Course:**

* **Attributes: courseID, courseName, preCount, preList**
* **Constructor initializes attributes to default values**

**Class BinaryTree:**

* **Node structure includes: Course, right pointer, left pointer**
* **Root pointer**
* **Methods for tree operations, including printing**

**Class HashTable:**

* **Bucket structure includes: Course, Key, Next pointer**
* **List to maintain buckets**
* **Methods for hash operations, including printing**

**Sorting and Printing Functions:  
sortList():**

* **Perform quicksort on the vector**
* **Recursively partition and sort sub-vectors**

**partition():**

* **Establish a pivot and reorganize elements based on their comparison to the pivot**
* **Return the partition index**

**printList():**

* **Iterate through courseList**
* **Display each course's ID and name, and iterate through its prerequisites**

**printTree():**

* **Recursive function to traverse the tree in-order and print each node**

**printTable():**

* **Iterate through the hash table**
* **Print course details and prerequisites for each entry**

Analysis of Data Structure Advantages and Recommendations

Each of the three data structures presents unique strengths and weaknesses. Loading data into an unsorted vector via an append operation is remarkably efficient, though sorting the vector afterward is notably slow. Hash tables can theoretically operate at an average time complexity of Θ(1) if they are sufficiently large to avoid any collisions. However, practical constraints on time and memory mean that collisions must be accommodated, potentially varying the performance from O(1) to O(N).Binary trees typically operate around O(log N), although this can worsen to O(N) if the data input causes the tree to become unbalanced, such as when sorted data is inputted. The choice of data structure should be guided by the specific needs of data access and frequency. For instance, if data is only loaded occasionally, the initial loading speed may not offer ongoing benefits. However, if frequent searches are required, a hash table might outperform a binary tree, especially with a well-crafted hash function and in cases where the tree structure is unbalanced.

Binary trees, which do not require sorting to be traversed in order, can also be more memory-efficient, avoiding the need to maintain both sorted and unsorted versions of the data. Generally, both binary trees and hash tables may offer superior performance compared to sorting vectors. Given that data will be loaded into memory infrequently, printed rarely, but searched frequently, a hash table is recommended. This choice hinges on the need for an optimized hash function and table size to minimize collisions and maintain performance closer.

**Citations**

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/4/section/1

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/4/section/2

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/1

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/1/section/10

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/1/section/9

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/1

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/2

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/8