Brandon Barrett

CS 300 Analysis and Design

Professor Michael Susalla

4/13/2024

Module 6 Project 1

Main Function() //Menu Loop

Read command line arguments

Store argument as CSVFilePath

If no command line arguments, load default CSV file path

Loop while menuChoice is not equal to ‘9’

Output menu options

Get user input; Store in menuChoice //specifies the program's action

Get user input; Store in dataChoice //specifies the data structure to use

Validate user input

If menuChoice is not in range 1-4 or 9, throw an error

If menuChoice equals ‘1’

//Call file parser and load data into each data structure

If dataChoice is BinarySearchTree

Call loadBids and store CSV data in BinarySearchTree bst

Else If dataChoice is vector

Call loadBids and store CSV data in vector courseList

Else If dataChoice is HashTable

//loadBids to ensure hash function orders the map in ascending order

Call loadBids and store CSV data in HashTable courseTable

Output the number of records in the CSV file

If menuChoice equals ‘2’

//Validate the List

If dataChoice is BinarySearchTree

Call validateTree() passing bst

Else If dataChoice is vector

Call validateList() passing courseList

Else If dataChoice is HashTable

Call validateTable() passing courseTable

If menuChoice equals ‘3’

//Search and print course

Get user value to search for and Store in userSearch

If dataChoice is BinarySearchTree

Call printCourseTree() passing userSearch

Else If dataChoice is vector

Call printCourseList() passing userSearch

Else If dataChoice is HashTable

Call printCourseTable() passing userSearch

If menuChoice equals ‘4’

//Print each course in alphabetic order

If dataChoice is BinarySearchTree

Call printTree()

Else If dataChoice is vector

Call sortList()

Call printList()

Else If dataChoice is HashTable

Call sortTable()

Call printTable()

If menuChoice equals ‘9’

Exit the application

Output ‘Goodbye’

End

End

Struct Course {}

courseID

courseName

preCount

prelist

Course() (constructor) {

courseID = courseName = “”;

preCount = 0;

preList = “”;

}

End

Class BinaryTree{}

Struct Node

Course

right pointer

left pointer

End

root

printTree()

BinaryTree()

End

Class HashTable{}

Struct bucket

Course

Key

Next pointer

End

hash()

printTable()

List<> hashTable

sortList()

partition()

printList()

End

Run Time Analysis:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary Tree |
| Loading Data | O(n) | O(n) | O(n log n) |
| Search | O(1) | O(1) | O(log n) |
| Sort/Print | O(n log n) | O(n log n) | O(n) |

Advantages and Disadvantages:

Each data structure has its advantages and disadvantages. For a vector, it has a simple implementation and its sequential access is fast, however, its dynamic resizing can lead to occasional performance hits. Insertions and deletions at arbitrary positions can be slow due to shifting elements.

Hash tables have a constant-time average case for insertion, deletion, and search operations which is efficient for large datasets with uniform distribution. However, its worst-case time complexity for some operations can degrade to O(n). it also requires a good hash function for optimal performance.

A binary search tree maintains sorted order, making sorting and printing in alphabetical order straightforward (O(n)). It also can ensure good performance in worst-case scenarios. However, it can become unbalanced and has a more complex implementation than vectors and hash tables.

Recommendation:

Based on the analysis and the requirements provided by the advisor, the Binary Search Tree seems to be the most suitable data structure for this scenario. Searching for a course by ID can be done in O(log n) time, which is better than both vectors and hash tables. Binary search trees also make sorting and printing more straightforward with a time complexity of O(n). Compared to hash tables, binary search trees generally require less memory overhead for storing pointers.