**Ron Milano**

**CS-300**

**Project 1**

**October 13, 2024**

# ***Pseudocode***

Main Function() //Menu Loop

Read cmd arguments

Store argument as CSV file path

If no cmd arguments load default CSV file path

Loop while choice is not equal to ‘9’

Output menu block

Get user input; Store in *menuChoice* //what the program is to do

Get user input; Store in *dataChoice* //what data structure to use

Validate user input

If choice is not 1-4 or 9 then throw an error

If choice equals ‘1’

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

If BinarySearchTree

Call loadBids and store CSV data in BinarySearchTree *bst*

Else If vector

Call loadBids and store CSV data in vector *courseList*

Else If HashTable

//loadBids to have a hash function that orders the map in ascending order

Call loadBids and store CSV data in HashTable *courseTable*

Output number of records in the CSV file

If choice equals ‘2’

//Validate the List

If BinarySearchTree

Call validateTree() passing *bst*

Else If vector

Call validateList() passing *courseList*

Else If HashTable

Call validateTable() passing *courseTable*

If choice equals ‘3’

//Search and print course

Get user value to search for and Store in *userSearch*

If BinarySearchTree

Call printCourseTree() passing *userSearch*

Else If vector

Call printCourseList() passing *userSearch*

Else If HashTable

Call printCourseTable() passing *userSearch*

If choice equals ‘4’

//Print each course in alphabetic order

If BinarySearchTree

Call printTree()

Else If vector

Call sortList()

Call printList()

Else If HashTable

Call sortTable()

Call printTable()

If Choice equals ‘9’

Exit the application

Output ‘Good bye’

End

struct Course {}

*courseID*

*courseName*

*preCount*

*prelist*

Course() (constructor) {courseID = courseName = ””; preCount = 0; preList = “”}

Class BinaryTree{}

-struct *Node*

*Course*

*right* pointer

*left* pointer

-*root*

*+printTree()*

+*BinaryTree()*

Class HashTable{}

-struct *bucket*

*Course*

Key

Next pointer

+*hash()*

*+printTable()*

+List<> *hashTable*

sortList()

Get vector to sort, lowest vector and highest vector

If lowest index if greater than/equal to highest index return nothing

Call partition() function

Set *lowEndIndex* equal to the value returned by the partition

Recursively call quicksort passing the vector, lowest index, and *lowEndIndex*

Recursively call quicksort passing the vector, *lowEndIndex* plus one, and highest index

End

partition()

Get the vector to partition, the lowest index and highest index

Determine the vector element at the midpoint between the lowest and highest index

Set pivot equal to this element

Loop until the lowest index is greater than/equal to the highest index

Loop through the vector from lowest index until a vector element larger than the pivot is found

Overwrite lowest index with this position

Loop through the vector from lowest index until a vector smaller than the pivot is found

Overwrite highest index with this element’s position

Swap the vector elements at the new highest and lowest index

Overwrite the lowest index by incrementing it one

Overwrite the highest index by decrementing it one

Return the highest index

End

printList()

Loop through *courseList*

Output to console: *courseID, courseName,*

Loop 0 to *preCount*

For each *Course* in *preList*

Output to console: *courseID*

End

printTree()

Create new Node pointer named root

Set root to NULL

Check if Node is null and if so return

Call via recursion Node’s left pointer which will find the left most Node

Output to console: *courseID, courseName,*

Loop 0 to *preCount*

For each *Course* in *preList*

Output to console: *courseID*

Call via recursion Node’s right pointer which will find the right most Node

End

printTable()

Create a new Node pointer and Set to the address of the nodes beginning

Loop through the list; starting at the beginning

Output *courseID* in Course struct found within *tempCourse* to console

Output *courseName* in Course struct found within *tempCourse* to console

Loop 0 to *preCount*

For each *Course* in *preList*

Call printCourse() passing *prelist*

End

# ***Run Time Analysis***

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

# ***Advantage Analysis***

All three data structures have advantages and disadvantages. Utilizing an append method is very quick to upload data, however, sorting it is a much slower process.

A hash table can operate at its average Θ(1) if the hash table were large enough. However, this hash table needs to be able to handle some collisions since time and memory are both not infinite.

The binary tree will mostly operate near O(log N) depending on the data. If the tree were to become unbalanced, then the Binary Tree slows down to O(N).

When choosing between data structures, it depends on how the data will be accessed. It also depends on how often the data will be accessed. For example, if the data only needs to be loaded from time to time, there is no advantage after the initial load phase. If the data needs to be searched for often, then the table needs to operate as efficiently as possible.

The binary tree is able to save memory since it does not need to be sorted. For this reason, both the binary tree and the hash table will perform better than vector sorting.

# ***Recommendation***

As long as the data will only be accessed from time to time, not often printed, but searched often, the has table would be preferred for this application.