Morgan Reed

08/10/2024  
CS 300

**Project One**

# **Pseudocode**

Main Function()

Read the cmd argument, then:

Store the argument as a CSV file path.

If there are no cmd arguments to load, then load the default CSV file path.

Loop while choice is not equal to ‘9,

Then output the menu block

Get the users input; then store in the menuChoice //*which guides the program on what it needs to do*

Get the users input; then store in the dataChoice //*what data structure the program is going to use*

Validate the user input,

If the choice is not 1-4 or 9, produce an error to the user.

If the user’s choice is ‘1’:

If BinarySearchTree:

Call the loadBids and store the CSV data in BinarySearchTree bst

Else if vector:

Callthe loadBids and store the CSV data in vector courseList

Else if HashTable:

Call the loadBids and store the CSV data in HashTable courseTable.

Output the number of records stored in the CSV file.

If the user’s choice is ‘2’:

If BinarySearchTree:

Call validateTree() passing bst

Else if vector:

Call validateList() passing courseList

Else if HashTable:

Call validateTable() passing courseTable

If the user’s choice is ‘3’,

Get the user value to search for and store in the userSearch

If BinarySearchTree:

Call printCourseTree() passing userSearch

Else if vector:

Call printCourseList() passing userSearch

Else if HashTable:

Call printCourseTable() passing userSearch

If the user’s choice is ‘4’, // this is going to sort them in alphabetical order.

If BinarySearchTree:

Call printTree()

Else If vector:

Call sortList()

Call printList()

Else If HashTable:

Call sortTable()

Call printTable()

If the user choice equals ‘9’:

Exit the application

Output ‘Goodbye!’

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 the vector to sort, starting with the lowest index of vector and ending in the highest index of vector.

If the lowest index if greater than or equal to the highest index, then nothing gets returned.

Call on the partition() function

Set the lowEndIndex to equal the value being returned by the partition function

Then, call on the quicksort passing the vector, starting at the lowest index, and then the lowEndIndex

Then, call on quicksort passing the vector, the lowEndIndex plus one, and then highest index

End

partition()

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

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

Set the pivot to equal the vector element above

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

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

Overwrite the lowest index with this element’s position, then

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

Overwrite the highest index with this element’s position

Swap the vector elements at the new highest and lowest index

Overwrite the lowest index by increasing it by one

Overwrite the highest index by lowering it by one.

Return the highest index

End

printList()

Loop through the courseList

Output to the console: “courseID, courseName,”

Loop 0 to the preCount

For each course in prelist:

Output to console: courseID

End

printTree()

Create a new Node pointer named “root”,

Set the root to null.

Check if the node is null, and if it is, return.

Call the recursion node’s left pointer, which will find the furthest node to the left

Output to console: “courseID, courseName,”

Loop 0 to preCount

For each course in the prelist:

Output to console: “courseID”

Call the recursion node’s right pointer which will find the node furthest to the right.

End

printTable()

Create a new node pointer, and set it to address the beginning of the node.

Loop through the list; starting at the beginning.

Output the “courseID” in course found within tempCourse to the user.

Output “courseName” in course found within tempCourse to the user.

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 of the data structures have their advantages and disadvantages. Loading the data into a unsorted vector using the “append method” is really fast, but it is super slow when it is sorted at a later time.

The hash table *could* continuously operate at an average speed if the hash table was large enough to prevent all its collisions, but there is not enough time or memory to make it run smoothly.

The binary tree is the most consistent, depending on how the data is read, as long as it remains balanced. If something happened to make it lose its balance, then it causes its speed to slow down noticeably.

A programmer must take in consideration how they want to use the data to have the most efficiency. For example, if the data needs to be searched often, then the hash table could be better than the binary tree.

# **Recommendation**

If the data used will only need to be printed once in a while, but gets searched more often, then the Hash Table would be the best choice. But, this also means that the hash function and table size need to be optimized to limit any collisions.