Bryce Cooperrider

CS-300

12/3/23

Project 1

Pseudocode:

Main function()

Read command arguments

Store argument as csv file path

If no arguments given, load default csv file path

Loop while choice is not equal to “9”

Output menu text

//input for which menu choice to use

Get user input, store in menuChoice

//input for which data structure to use

Get user input, store in dataStructChoice

Validate menuChoice

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

If menuChoice equals “1”

If dataStructChoice equals linkedList

Call loadBids and store csv data in vector courseList

Else If dataStructChoice equals hashTable

Call loadBids and store csv data in hashTable courseTable

Else If dataStructChoice equals binarySearchTree

Call loadBids and store csv data in binarySearchTree bst

Output the number of records from the csv file

If menuChoice equals “2”

If dataStructChoice equals linkedList

Call validateList() passing courseList

If dataStructChoice equals hashTable

Call validateTable() passing courseTable

If dataStructChoice equals binarySearchTree

Call validateTree() passing bst

If menuChoice equals “3”

Get user value to search and store in userSearch

If dataStructChoice equals linkedlist

Call printCourseList() passing userSearch

Else If dataStructChoice equals hashTable

Call printCourseTable() passing userSearch

Else If dataStructChoice equals binarySearchTree

Call printCourseTree() passing userSearch

If menuChoice equals “4”

If dataStructChoice equals linkedList

Call sortList()

Call printList()

Else if dataStructChoice equals hashTable

Call sortTable()

Call printable()

Else if dataStructChoice equals binarySearchTree

Call printTree()

If menuChoice equals “9”

Exit the program

Output “Good bye”

End

Struct course {}

courseId

courseName

preReqCount

preReqList

course()

Class hashTable{}

Struct bucket

Course

Key

Next pointer

Hash()

printTable()

class BinaryTree{}

struct node

course

left pointer

right pointer

root

printTree()

binaryTree()

sortList()

get vector to sort, with the lowest and highest index of the vector

if the lowest index is greater than or equal to highest index, return nothing

call partition()

set lowEndIndex equal to the value returned from partition function

call quicksort passing the vector, lowest index, and lowEndIndex

call quicksort passing the vector, lowEndIndex plus 1, and highest index

partition()

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

determine the element at the midpoint of the vector

set the pivot equal to the midpoint

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

loop through the vector from the lowest index until a larger vector element is found

overwrite the lowest index with this element’s position

loop through the vector from the lowest index until a smaller vector element is found

overright the highest index with this element’s position

return the highest index

end

printList()

loop through courseList

output to the console, courseId, and courseName

loop 0 to preReqCount

for each course in preReqList

output to console courseId

end

printTable()

create a new node pointer and set it to the address of the beginning node

loop through the list starting at the beginning

output to console courseId

output to console courseName

loop 0 to preReqCount

for each course in preReqList

output to console courseId

end

printTree()

create new node pointer called root

set root equal to NULL

check if node is null, and return if so

call with recursion the node’s left pointer, which will find the left most node

output to console courseId, courseName

loop 0 to preReqCount

for each course in preReqList

output to console courseId

end

Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | LinkedList | Hash Table | Binary Tree |
| Loading Data | O(1) | O(1) – O(n)  Depends on if there are collisions | O(log n) – O(n)  Depends if the tree is balanced |
| Search | O(n) | O(1) – O(n)  Depends on if there are collisions | O(log n) – O(n)  Depends if the tree is balanced |
| Sort and Print | O(n log n) | O(log n) – O(n log n)  Depends if the table is sorted or not | O(n) – O(n^2)  Depends if the tree is balanced |

Advantages:

The three data structures each have their own advantages and disadvantages. For example, loading data into an unsorted vector is very fast but if you have to sort it later, its runtime slows down drastically.

Hash tables theoretically can get down to O(1) runtime for many functions assuming no collisions happen. That means we would need a table large enough to prevent all collisions, which isn’t realistic due to memory constraints within a system. Because we know collisions will happen, the runtime averages between O(1) – O(n).

Binary trees will consistently operate around O(log n). Depending on what we need to store and why, this could be a good method. Binary trees can slowdown in runtime though to O(n) depending if the tree is balanced or not.

Recommendation:

Due to the assignment we are doing, data will not need to be read into memory frequently, and will likely not be completely printed frequently either. We will be using search quite heavily though, in order to find out what classes are and their required prerequisites. Because of these factors, the hash table would be best for this assignment. It should be noted that we need to optimize the hash table to get it closer to O(1) than O(n).