Jon Marvin

8/5/2024

CS 300 – Project 1

SNHU

**Pseudocode**

Main function() //Menu

READ command arguments

STORE arguments as CSV

IF no command arguments

LOAD default CSV

LOOP while menu option does not equal ‘9’

OUTPUT menu options

GET input from user and store in menuChoice

GET input from user and store in dataChoice

VALIDATE user input

IF user input does not equal 1, 2, 3, 4, or 9

ERROR invalid selection

IF input equals 1

IF Binary Search Tree

CALL loadBids and store CSV data in Binary Search Tree bst

IF vector

CALL loadBids and store CSV data in vector courseList

IF Hash Table

CALL loadBids and store CSV data in Hash Table courseTable

OUTPUT CSV record count

IF input equals 2

IF Binary Search Tree

CALL validateTree() passing bst

IF vector

CALL validateList() and pass courseList

IF Hash Table

CALL validateTable and pass courseTable

IF input equals 3

GET user value

STORE user value in userSearch

IF Binary Search Tree

CALL printCourseTree() and pass userSearch

IF vector

CALL printCourseList() and pass userSearch

IF Hash Table

CALL printCourseTable() and pass userSearch

IF input equals 4

IF Binary Search Tree

CALL printTree()

IF vector

CALL sortList()

CALL printList()

IF Hash Table

CALL sortTable()

CALL printTable()

IF input equals 9

EXIT application

END

Structure couse {}

courseID

courseName

preCount

preList

CLASS BinaryTree{}

-structure node

Course

Right Pointer

Left Pointer

-root

+printTree()

+BinaryTree()

CLASS Hash Table

-structure bucket

Course

Key

Next Pointer

+hash()

+printTable()

+list<>hashTable

sortList()

GET vector to sort

IF lowest index greater than or equal to highest index

RETURN nothing

CALL partition()

SET lowEndIndex equal to partition value

RECURSIVE CALL quicksort and pass vector, lowest index, lowEndIndex

RECURSIVE CALL quicksort and pass vector, lowEndIndex plus one, highest index

END

Partition()

GET vector to partition – lowest and highest indexes

DETERMINE median index

SET pivot equal to median index

LOOP until lowest index is greater than or equal to highest index

LOOP through vector to find element larger than pivot

REPLACE lowest index with new element

LOOP through vector to find element smaller than pivot

REPLACE highest index with new element

SWITCH vector elements at new highest and lowest indexes

REPLACE lowest index by incrementing by 1

REPLACE highest index by decrementing by 1

RETURN highest index

END

printList()

LOOP through courseList

OUTPUT courseID and courseName

LOOP 0 to preCount

FOR each course in courseList

OUTPUT courseID

END

printTree()

CREATE node pointer called root

SET root equal to NULL

IF node is null

RETURN node

RECURSIVELY CALL node left pointer to find leftmost node

OUTPUT courseID and courseName

LOOP 0 to preCount

FOR each course in courseList

OUTPUT courseID

RECURSIVELY CALL node right pointer to find rightmost node

END

printTable()

CREATE node pointer

SET pointer to beginning of nodes

LOOP through list from beginning

OUTPUT courseID from tempCourse

OUTPUT courseName from tempCourse

LOOP 0 to preCount

FOR each course in preList

CALL printCourse() and pass preList

END

**Run Time Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N)  (depending on collisions) | O(log N) |
| **Search** | O(N) | O(1) – O(N)  (depending on collisions) | O(log N) – O(N)  (depending on tree balance) |
| **Sort/Print** | O(N log N) (quick sort) | O(N)  (assumes table is ordered) | O(N)  (in order) |

**Advantage Evaluation**

Choosing a data structure depends mostly on how you are accessing the data and the frequency at which the data will be used. If data is rarely needed, it is not frequently used, therefore the only advantage is on the initial load. The initial load into an unsorted vector can be incredibly fast, but sorting afterward is slower than the other two. A hash table could be better than a binary search tree for data that needs to be accessed more frequently, but this depends on the efficiency of the hash function and the balance of the tree. A binary tree does not require sorting like a hash function, leading to less demand on memory if lists do not need to be stored. Regardless, both the hash table and binary tree would be better options for sorting than the vector.

**Recommendation**

Assuming the data is read and printed infrequently but searched regularly, the Hash Table would be the best choice. This is based on the assumption that the hash function and table size are created to limit collisions to see performance closer to O(1) than O(N).