Jacob Rethmeier

Professor Lusby

Cs-300

2/16/2025

CS 300 Project One

**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 throw an error

If choice equals ‘1’

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

If BinarySearchTree

Call loadCourses and store CSV data in BinarySearchTree bst

Else If vector

Call loadCourses and store CSV data in vector courseList

Else If HashTable

Call loadCourses 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 ‘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 vector to sort, lowest index of vector and highest index of vector

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

Call partition() function

Set lowEndIndex equal to the value returned by the partition function

Recursively call quicksort passing the vector, lowest index, and lowEndIndex (from above)

Recursively call quicksort passing the vector, lowEndIndex (from above) plus one, and highest index

End

partition()

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

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

Set pivot equal to this vector element

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 lowest index with this element’s position

Loop through the vector from lowest index until a vector element 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 |
| Loading Data | O(1) | O(1) – O(N)  \*depends on if there is collisions | O(log N) |
| Search | O(n) | O(1) – O(N)  \*depends on if there are collisions | O(log N) – O(N)  \*depends on balance of the tree |
| Sorting/Printing | O(N log N)  \*Using quick sort | O(N)  \*assumes the table is created in order | O(N)  \*in order traversal |

**Advantage Analysis**

Vectors are a very simple and efficient way to not only store but also manipulate data. When searching through a vector it requires a linear scan which would make it be a subpar performer when doing constant lookups. The sorting operations can also be significantly slower since a vector must be rearranged to maintain order.

The benefits for hash tables are that they offer rapid search times compared to the other options. This makes them a most likely the best choice for applications that require constant lookups. The downside is that if there are collisions in the table it can worsen performance, making it essential to implement a good hashing strategy. On top of that, hash tables also require a larger sized memory.

Binary search trees provide us with an organized way to store data as well as allowing efficient searching, provided the tree remains balanced. The potential downside for the binary search tree is if it becomes skewed the search times will be far worse. Binary trees are useful for maintaining a sorted order without the need for any additional sorting functions.

**Recommendation**

Based on the overall needs for this program, I would personally recommend the hash table because it is the best option due to its fast search capabilities. Maintaining O(1) efficiency in lookups is very critical for the function and while a vector might be easier to implement and a binary search tree allows for natural ordering, the hash table is the most practical choice for frequent and quick data retrieval which is the most essential function of this program. With this recommendation it would be vital that an effective hashing technique is used to reduce any potential causes of collisions.