Dylan Lovin

CS-300

2/16/25

**Project 1:**

**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 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 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

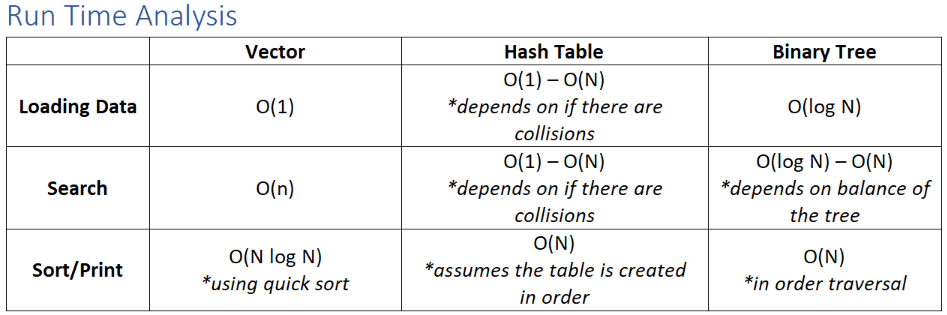
Output courseName in Course struct found within tempCourse to console

Loop 0 to preCount

For each Course in preList

Call printCourse() passing prelist

End



**Advantage Analysis**

Each data structure comes with its own strengths and weaknesses. Using an unsorted vector with an append method ensures rapid data insertion, but sorting it later results in the slowest performance.

A hash table theoretically maintains an average time complexity of Θ(1) if it is large enough to avoid collisions entirely. However, practical constraints on memory and time mean that some collisions are inevitable, potentially increasing the complexity to a range between O(1) and O(N).

A binary tree generally operates at O(log N) but can slow down to O(N) if it becomes unbalanced—such as when inserting already sorted data. The overall efficiency of the tree is influenced by how the data is inserted and managed.

The choice of data structure largely depends on the frequency and type of data access. If data is only loaded occasionally, performance differences beyond the initial insertion may be negligible. However, if frequent searches are required, a hash table may outperform a binary tree, provided the hash function is efficient and the tree remains balanced.

One key advantage of a binary tree is that it inherently maintains order, allowing in-order traversal without requiring additional sorting. This can be beneficial when both sorted and unsorted versions of the data do not need to be stored separately. Additionally, in many cases, both the binary tree and the hash table offer better performance than sorting an unsorted vector.

**Recommendation**

If the data is expected to be loaded into memory infrequently, printed in its entirety only on rare occasions, but searched frequently, then a hash table is likely the best choice. However, to ensure optimal performance, the hash function must be well-designed, and the table size should be carefully chosen to minimize collisions. This helps maintain efficiency closer to O(1) rather than degrading toward O(N).