Project 1

Pseudocode:

Struct Course contains

courseID

courseName

preCount

prelist

Course()

Constructor

courseID is equal to 0

courseName is equal to blank string

preCount set to 0

prelist set to blank string

Class BinaryTree{}

-struct Node

Course

Right pointer

Left pointer

-root

+printTree()

+BinaryTree()

Class HashTable{}

-struct bucket

Course

Key

Nxt Pointer

+hash()

+printTable()

+List<>hashTable

Main()

Read cmd args

Store args as csv path

If no cmd args load default csv path

Loop while choice is not 9

Output menu

Get user input

Store input in userChoice

Get user input

Store input in userDataChoice

Validate user input

If choice is not 1-4 or 9 throw error

If choice is 1

If BinarySearchTree

ZCall loadBids and store csv data in binarySearchTree

Else if vector

Call loadBids and store CSV data in vector courseList

Else if HashTable

Cxall loadBids and store csv data in HashTable courseTable

Output # of records in csv file

If choice is 2

If binarySearchTree

Call ValidateTree()

Else if vector

Call validate List()

Else if Hashtable

Call validate table

If choice is 3

Get user value to search for and Store in userSearch

If BinarySearch tree

Call printCourseTree()

Else if vector

Call printCourseList()

Else if hashTable

Call printCourseTable

If choice is 4

If binarySearchTree

Call printTree()

Else if vector

Call sortList()

Call printList()

Else if HashTable

Call sortable()

Call printTable()

If choice is 9

Exit

end

PseudoCode for sort list

Get vector to sort, low and high indexes

If low index

If greater than or equal to highest index

Return nothing

Call partition()

lowEndIndex equal to value returned

recursively call quicksort passing vector, low index, and lowEnd index

recursively call quicksort passing vector, lowEndIndex+1, and high index

PseudoCode for partition()

Get vector, low index and high index

Determine vector element at midpoint between high and low indexes

Set pivot equal to element at that location

Loop until low index is greater than nor equal to high index

Loop vector from low index until vector element larger than pivot is found

Write over low index with that elements position

Loopp through vector from low index until vector element smaller than pivot found

Write over high index with that elements position

Swap vector elements at the new high and low indexes

Overwrite low index via incrementing it one

Do the same to highest but decrement it by one

Return highest index

Pseudocode for printList()

Loop through courseList

Output courseID, courseName,

Loop 0 to preCount

For each Course in prelist

Output to console: courseId

Pseudocode for printTree()

Declare New node pointer root

Root equal to NULL

If node is null

Return

Recursively call nodes left pointer to find left most node

Output courseId, CourseNamed

Loop 0 to preCount

For each course in prelist

Output courseId

Call recursively Node’s right pointer to find right most node

Pseudocode for printTable()

New node pointer set equal to address of nodes beginning

Loop through list

Output courseID

Output courseName

Loop 0 to preCount

For each course in prelist

Call printCourse()

Run Time Analysis

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Binary Tree | Hash Table |
| Load Data | O(1) | O(log N) | O(1) – O(N)  Depending on collisions |
| Search | O(n) | O(logN) – O(N) | O(1) – O(N)  Depending on collisions |
| Sort and Print | O(N log N) via quicksort | O(N) | O(N) |

Advantages and Disadvantages

Each data structure will have advantages and disadvantages. For example, loading data into an unsorted vector via an append method will run very quickly, however, sorting it later will slow the performance down. Hash tables can in theory always operate at O(1) if it were large enough to avoid all collisions. This isn’t realistic however, and as such the table will need to handle some collisions which will put its speed to around O(1) – O(N) for loading and searching data. Binary trees operate fairly consistently around O(logN) depending on how data is read in, however if the tree becomes unbalanced the Binary tree will slow down to O(N). Depending on the uses of the program will decide what is the preferrable approach to storing and reading data.

Recommendation

Based on the premise of the project, it appears that data will need to be read into memory infrequently, completely printed rarely, but will need to be searched very often. As this is the case, a Hash Table would be my suggestion. This does mean the hash function and table size need to be optimized to ensure collisions occur as infrequently as possible and that the code approaches an O(1) run-time vs. O(N).