2/15/2024

**Project One**

// Code for the menu

**Main()**

Read command arguments

Store as CSV path

IF no argument

Load default path

// while loop

While user input != 9

Display menu options

// Instructions for the program

Get input from user to store in menuInput

// Selects proper data structure

Get user input to store under dataInput

Validate input from user

IF user input != 1-4 || 9  
 ERROR

IF user input == 1  
 // Call parser to load information into data structures

IF BST

Call loadBids then store data from CSV in BST

ELIF hash table

//load bids for hash func to organize by ascending

Call loadBids then store data from CSV in courseTable

ELIF Vector

Call loadBids then store data from CSV in courseList

IF user input == 2

// list validation

IF BST

Call treeVal() pass BST

IF HashTable

Call tableVal() pass courseTable

IF Vector

Call listVal() pass courseList

IF user input == 3

// search and display course

Obtain user input to search, then store in userSearch

IF BST

printCourseTree() pass userSearch

ELIF Vector

printCourseList() pass userSearch

ELIF HashTable

printCourseTable() pass userSearch

IF user input == 4

// print courses sorted alphabetically

IF BST

printTree()

ELIF Hash

sortTable()

printTable()

ELIF Vector

sortList()

printList()

IF user input == 9

EXIT

**END**

// Define structures for course data

**Struct Course {**

courseTitle

courseNum

prerequisites

// Constructor

Course() {courseNum = courseTitle = ””; preReqCount = 0; preList = “”  
 **}**

**Class BinaryTree()**

- node struct

Left ptr

Right ptr

Course

- root

+printTree()

+BinaryTree()

**Class HashTable[**

-bucket struct

Course

Key

next ptr

+hash()

+printall()

+List<>HashTable

}

**SortList()**

Obtain vector, get lowest and highest indexes

IF lowest >= highest

Return nothing

Partition() func

LowIndex = partition value

Quicksort pass vector, lowest index from the top

Quicksort pass vector, lowest index+1, highest index

**END**

**Partition()**

Obtain vector, get lowest and highest indexes

Determine value present at midpoint between indexes

Pivot = midpoint element

LOOP until lowest >= highest

LOOP starting at lowest until element greater than pivot is found

Rewrite lowest with this element

LOOP starting at lowest until element smaller than pivot is found

rewrite highest with this element

Switch places of highest and lowest elements

rewrite lowest increase size by 1

rewrite highest decrease size by 1

Return highest

**END**

**PrintTable()**

New nodeptr set to node beginning

LOOP through list from beginning

display courseNum from course struct

display courseTitle from course struct

LOOP 0 to preReqCount

for each course found in preList

printCourse() pass preList

**END**

**PrintTree()**

New nodeptr = root

Root = NULL

IF node = null

return

Call via node left ptr

Display courseNum and courseTitle

LOOP 0 to preReqCount

for each course found in preList

display courseNum

Call via node right ptr

**END**

**PrintList()**

LOOP courseList

Display courseNum and courseTitle

LOOP 0 to preReqCount

for each course found in preList

display courseNum

**END**

**Run Time Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vectors | HashTable | Binary Search Tree |
| Load Data | O(1) | O(1) -> O(N)  *(Dependent upon collisions being present)* | O(log N) |
| Searching | O(n) | O(1) - O(N)  *(Dependent upon collisions being present)* | O(log N) -> O(N)  *(dependent upon the tree’s balance)* |
| Print and Sort | O(N log N)  *(via quicksort algorithm)* | O(N log N) | O(N)  (*traversal order)* |

Among the three options, there are both advantages and disadvantages.

Using a vector would be one-dimensional, meaning the handling of data in this case would be advantageous. Uploading the data via append will operate quite fast. Less memory is utilized; and elements can be removed using operations like pop\_back() or erase(). Vectors in C++ can store multiple data types by using templates. As vectors typically utilize contiguous memory allocation, they offer better cache performance compared to the other data structures. Sorting can become slow if the vector contains a large number of elements especially when utilizing the quicksort algorithm. Sorting can be slower in comparison to balanced binary search trees, but it is still operable.

Utilizing a hash table enables organization and storage of information. The table has the ability to be called throughout via a key. Some major advantages of a hash table include the ability to add, remove, and call throughout the project. They also have the capability to create unique elements and can be synchronized. Hash tables should optimally be able to perform at a level of O(1) if they were large enough to prevent collisions. However, as both memory and time cannot be infinite, the table should be able to account for some collisions, making the table operate on a level between O(1) and O(N). Collision resolution strategies like probing and chaining are used to keep the time complexity at a level closer to O(1), but can rise to O(N) in the worst-case scenarios depending on the load factor and quality of the hash. If you must sort the elements in a hash table, you must first extract them into another data structure such as a vector before applying the sorting algorithm. This can reduce the performance advantage found with using hash tables. However, hash tables are not inherently designed for sorting, with their primary functionality being efficient key-based access.

Utilizing a binary tree allows for good organization, with the ability to store data in either the right or left, both of which can be expanded as needed. With a binary tree you are also able to search at any point similarly to the hash table, but it does take longer for insertion and deletion of data. The tree typically maintains a sorted order by nature (left < parent < right) allowing the BST to be accessed in order by default. This makes memory usage more efficient if both the unsorted and sorted lists don’t have to be stored. When it comes to sorting, vectors are superior as they are ordered containers and are better suited to that operation. However, frequent modifications can affect the performance of sorting algorithms on vectors, but your choice of sorting algorithm, the vector’s size, and pattern of both insertion and deletion are contributing factors.

If the data is only loaded every now and then, there is no real advantage following the initial load. On the other hand, if the data is searched regularly, I believe the hash table can perform better than the tree under the assumption that the hash is well-made and efficient, or the tree is not balanced properly. The efficiency of a hash table depends on various factors, like the quality of the hash function, the distribution of keys, and the collision handling. Unbalanced trees, like skewed binary search trees, can downgrade performance in terms of search and insertion operations. The frequency of data access and also other factors like memory usage, data distribution, expected operations (for example search, insert, delete), and the importance of maintaining sorted order should be considered when choosing an appropriate data structure.

Of each data structure, I would personally recommend the use of hash tables. Although there is the disadvantage of the table running at a slower speed, the benefits far outweigh this drawback. Hash tables may also suffer from collisions, which can negatively affect performance. Optimizing the table size and hash function can help mitigate this issue. The hash table allows for greater organization. It has already been established that we are looking to sort courses, etc., but if you were to add further functionality or items, there would not be issues doing so. To maximize the hash table, the table size and hash function should be optimized to mitigate collision. This can be done through separate chaining (using linked lists or other structures to store any collided keys) or open addressing (rehashing or probing) to handle collisions. The best case scenario would be optimizing the code to operate at a level close to O(1) rather than O(N).