Dubon, Katerine

CS 300

October 12, 2024

Professor Rissover

**Pseudocode**

Function main () { //Menu Loop   
 Access and note command line arguments   
 Save the parameter as the location of a CSV file.   
 If there are no command arguments, the default CSV file path will be loaded.   
 Keep looping until the choice is not equal to '9'.   
 Menu block for display   
 Obtain input from the user and save it in the variable menuChoice to determine the program's task.   
 Receive input from the user and save it in the dataChoice variable //determine the appropriate data structure to utilize   
 Confirm the user's input.   
 Throw an error if the choice is not a number between 1 and 4, or 9.   
 If selection is equal to '1'   
 Invoke the file parser and populate each data structure with the loaded data.   
 If it is a BinarySearchTree   
 Invoke loadBids function to save CSV information in BinarySearchTree bst.   
 If the vector is not true   
 Invoke the loadBids function and save the CSV information in a vector named courseList.   
 If the condition is a HashTable   
 //Implement loadBids with a hash function that arranges the map in increasing order   
 Utilize loadBids to retrieve CSV information and save it in HashTable courseTable.   
 Determine the total count of entries in the CSV document.   
 If option is equal to '2'   
 Verify the List   
 If it is a BinarySearchTree   
 Invoke the validateTree() function with the argument bst.   
 If the vector is in another case.   
 Invoke validateList() function with courseList as the parameter.   
 If the condition is HashTable.   
 Invoke validateTable() with courseTable as parameter.   
 If option is equal to '3'   
 Retrieve course information and display it.   
 Receive input value from user for searching and Save in userSearch variable.   
 If a BinarySearchTree is present   
 Invoke printCourseTree() with userSearch as the argument.   
 If the vector is different

Invoke printCourseList() with userSearch as the argument.   
 If HashTable is not true.   
Invoke printCourseTable() with userSearch as an argument.   
 If the value of choice is ‘4’   
//Display every course in alphabetical sequence   
 In the event that BinarySearchTree is present.   
Evoke the printTree() function.   
 If a vector is not true.   
Invoke the sortList() function.   
Invoke printList()   
 If the condition is a HashTable.   
Invoke the sortTable() function.   
Invoke printTable()   
 If the Choice is '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()**

Obtain the vector to be sorted along with its minimum and maximum indices.   
 If the lowest index is equal to or exceeds the highest index, then do not return anything.   
 Invoke the partition() function.   
 Assign lowEndIndex the value returned by the partition function.

Call quicksort repeatedly by passing the vector, smallest index, and lowEndIndex (mentioned earlier).   
 Call quicksort again recursively, passing the vector, the lowEndIndex increased by one, and the highest index.

**End**

**partition**()

Obtain the vector that needs to be divided, along with the starting index and ending index.   
 Find the vector element located at the middle index between the lowest and highest indices.   
 Assign this vector element to be the pivot.   
 Keep iterating until the minimum index is equal to or higher than the maximum index.   
 Iterate through the vector starting from the smallest index and continue until reaching a vector element that is greater than the pivot.   
 Replace the smallest index with the position of this element.   
 Iterate through the vector starting from the smallest index until a vector element less than the pivot is encountered.   
 Replace the highest index with the position of this element.   
 Exchange the elements of the vector located at the newest highest and lowest

position.   
 Increment the lowest index by one through overwriting it.   
 Decrease the top index to replace it with a lower value.   
 Give back the maximum index.

**End**

**printTree**()

Instantiate a new pointer called root of type Node.   
 Assign NULL to root   
 Verify if Node is empty and if it is, then exit.   
 Invoke the left pointer of Node recursively to locate the Node farthest to the left.   
 Display courseID and courseName on the console.   
 Repeat from 0 to the value of preCount.   
 For every course on the preList.   
 Display courseID on the console.   
 Invoke the right pointer of Node recursively to locate the furthest right Node.

**End**

**printTable()**

Instantiate a new pointer to a Node object and assign it the memory address of the start of the nodes.   
 Iterate through the list, beginning from the start.   
 Display the courseID inside the Course struct located in tempCourse on the console.   
 Display the courseName in Course struct located in tempCourse on the console.   
 Iterate from 0 to preCount.   
 For every course in preList   
 Invoke the printCourse() function with prelist as the argument.

**End**

**Run time Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary tree |
| Loading Data | O (1) | O (1) – O (N)  *\*Relies on the presence of collisions* | O (log N) |
| Search | O (n) | O (1) – O (N)  \* *Relies on the presence of collisions* | O (log N) – O (N)  \**hinges on the equilibrium of the tree* |
| Sort/Print | O (N log N)  \**utilizing the quick sort algorithm* | O(N)  *\*Takes for granted that the table is constructed sequentially* | O (N)  \**traversal in order* |

# **Advantage Analysis**

Each of the three data structures has its own pros and cons. Adding data to an unsorted vector quickly with an append method is very efficient, but the slowest part comes when sorting it afterwards. If a hash table is sufficiently big to eliminate collisions, it could consistently function at its average Θ(1). Nonetheless, as time and memory are both limited, the table must be capable of managing collisions, resulting in the hash table falling within the range of O(1) to O(N). The binary tree's performance tends to be most stable around O(log N) based on how the data is accessed. If the tree becomes highly unbalanced, such as when sorted data is inserted, the Binary Tree's performance decreases to O(N). The choice of data structure depends on the method and frequency of data access. For instance, if the data is only required to be loaded occasionally, there are no benefits beyond the first load. If the data requires frequent searches, the hash table might be more effective than the binary tree, given a well-designed hash function or an extremely imbalanced tree. Finally, the binary tree can be traversed in order without needing to be sorted, potentially saving memory if both sorted and unsorted "lists" are unnecessary to store. In addition, the binary tree and hash table will outperform and be favored over sorting the vector.

# Recommendation

The belief is that the data will be read into memory only occasionally, printed rarely, but searched regularly, making the Hash Table the preferred choice. Nevertheless, it is essential to optimize the hash function and table size in order to reduce collisions and ensure that the code runs more efficiently towards O(1) rather than O(N).