Dan Stull

CS300

06/06/2022

Project One Pseudocode

- - - - - - - - - - - - - - - - - - - - - - - -

**Binary search tree**

- - - - - - - - - - - - - - - - - - - - - - - -

**struct Course {}**

*courseID*

*courseName*

*prereqCount*

*prereqlist*

Course() (constructor) {courseID = courseName = ””; prereqCount = 0; prereqList = “”}

**Class BinaryTree{}**

-struct *Node*

*Course*

*right* pointer

*left* pointer

-*root*

*+printTree()*

+*BinaryTree()*

**printTree**()

**Start**

**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**

**printCourseTree()**

**Start**

**While** the current is not null

Loop through BinaryTree

**If** userSearch equals course in BinaryTree()

**Output** course

**End**

- - - - - - - - - - - - - - - - - - - - - - - -

**Hashtable**

- - - - - - - - - - - - - - - - - - - - - - - -

**struct Course {}**

*courseID*

*courseName*

*prereqCount*

*prereqlist*

Course() (constructor) {courseID = courseName = ””; prereqCount = 0; prereqList = “”}

**Class HashTable{}**

-struct *bucket*

*Course*

Key

Next pointer

+*hash()*

*+printTable()*

+List<> *hashTable*

**printCourseTable()**

**Start**

**Loop** through HashTable()

**If** userSearch equals course in HashTable()

**Output** course

**End**

**printTable()**

**Start**

**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** p**rintCourse**() passing *prelist*

**End**

- - - - - - - - - - - - - - - - - - - - - - - -

**Vector**

- - - - - - - - - - - - - - - - - - - - - - - -

**struct Course {}**

*courseID*

*courseName*

*prereqCount*

*prereqlist*

Course() (constructor) {courseID = courseName = ””; prereqCount = 0; prereqList = “”}

**printCourseList()**

**Start**

**Loop** through Course

**If** userSearch equals course in Course

**Output** course

**End**

**sortList()**

**Start**

**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**()

**Start**

**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()**

**Start**

**Loop** through *courseList*

**Output** to console: *courseID, courseName,*

**Loop** 0 to *prereeqCount*

**For each** *Course* in *prereqList*

**Output** to console: *courseID*

**End**

- - - - - - - - - - - - - - - - - - - - - - - -

**Main**

- - - - - - - - - - - - - - - - - - - - - - - -

***//All three (vector, hash and BST) are represented in the following code, to save space***

**Main** Function()

**Read** command arguments

**Store** argument as CSV file path

**If** no file is found, resort to default storage location

**while** the choice is not equal to the number 9

**Output** menu block

**Get** user input; **Store** in *input* //what data structure to use

**If** choice equals ‘1’

**If** BinarySearchTree

**Call** loadBids and store CSV data in BinarySearchTree *bst*

**Else** vector

**Call** loadBids and store CSV data in vector *courseList*

**Else** **If** HashTable

**Call** loadBids and store CSV data in HashTable *courseTable*

**Output** number of records in the CSV file

**If** choice equals ‘2’

//Print each course in alphabetic order

**If** BinarySearchTree

**Call printTree()**

**Else** vector

**Call** **sortList()**

**Call printList()**

**Else** **If** HashTable

**Call** **sortTable()**

**Call printTable()**

**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** vector

**Call** **printCourseList()** passing *userSearch*

**Else** **If** HashTable

**Call** **printCourseTable()** passing *userSearch*

**If** Choice equals ‘9’

**Exit** the application

**Output** ‘Good bye’

**End**

- - - - - - - - - - - - - - - - - - - - - - - -

**RUN TIME ANALYSIS**

For my runtime evaluation, I decided to focus on the loading data portion of the program.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **BST** | **Hash Table** |
| **Loading the data** | 0(1) | 0(log N) | 0(1) to 0(N) |

Loading data to a vector that is unsorted in very easy and fast. The issues it runs into come later, when sorting is called. The Hash Table is very fast, but if it is not large enough, it can get bogged down by collisions (more than one piece of data occupying the same space), hence the Big O of between 0(1) and 0(N). The BST has a more consistent performance, coming in around 0(log N). But if the tree gets to far out of balance, one side larger than the other, and then more data is loaded, the BST can slow down to somewhere around 0(N). But a great thing about the BST is that it does need sorting, since it is initially stored with order in mind.

**RECOMMENDATION**

This recommendation is hard. With BST not needing to be sorted, saving on memory, and its more consistent performance, I tend to lean in that direction. But if the Hash Table is formatted correctly, meaning that it is large enough, limiting collisions, it is not a bad choice either. But for this project I will lean toward BST.