Emily Bolyard

10/09/2024

CS 300 Project One

**Reading Files**

Use fstream to open the file

Create a method loadCourses containing string csvPath and dataStruct

Open the file

IF value is less than one

RETURN file not found

ELSE

File is found

WHILE not the end of file

Read through each line

IF less than two values on a line

RETURN error

ELSE read the parameters

IF third or more of the parameters in first parameter continue

ELSE

RETURN error

Close the file

**Vector Pseudocode**

CREATE class Course with vector courseInfo

create identifiers for information in course:

courseNumber

courseName

vector string prerequisites

For elements of row after two

Add the value into vector prerequisites

search through course data

Loop through the vector

IF input is equal to courseNumber

PRINT course information for each prerequisite in the course

PRINT out Course Number Course Name Pre-requisites

PRINT out courseNumber courseName and prerequisites

Go through each element in prerequisites

PRINT out each element

**Hash Table Pseudocode**

create a hashTable class

initialize course Node nodes

create the variables

courseId, courseTitle, prerequisite1, prerequisite2, courseNumber

create a temp item to hold the values

create current to hold the values that points to next

WHILE not the end of the file

Loop through file

Create unsigned int key

IF node at key isn’t found

Insert node at hash key %

ELSE

IF node points to key is equal to UINT\_MAX

Node points to key equals key

Node points to next equals null

Node points to course = course

ELSE

WHILE node points to next not equal to null

Node equals node points to next

New node course and key

Create hashTable printAll

FOR unsigned int I = 0; I is less than tableSize; increment by 1

IF key is not equal to UINT\_MAX

PRINT courseId, prerequisite1, prerequisite2

WHILE next is not equal to null

Node equals node points to next

PRINT courseId, prerequisite1, prerequisite2

RETURN

**Binary Search Tree Pseudocode**

Create Course

loop through file

WHILE not the end of file

FOR first and second values

Add the courseId and courseName

IF there is a third value

Add the prerequisite

Create Binary Tree Class

Create root that points to null

Create insert function

IF root is equal to null then the current Course is equal to root

ELSE

IF course number is less than root add left

Left is equal to null add courseNumber

ELSE

IF courseNumber is less than node add left

courseNumber is greater than node add right

ELSE

IF courseNumber is greater than root add right

Right is equal to null add courseNumber

ELSE

IF courseNumber is less than node add left

courseNumber is greater than node add right

Create Search function

Get input

Create print function

IF root is equal to null

Look left

PRINT output

Look right

PRINT output

**Menu Pseudocode**

Create a loop for the menu

WHILE choice not equal to 4

Print the menu choices 1: load courses, 2:print course list, 3: print course 4: exit program

Create a switch case for choices

Case 1 loadCourses

Case 2 printSorted

Case 3 printCourseInformation

Case 4 end the program

**Print Sorted List**

Create a sorted print method printSorted

Create a partition method partition with vector course and courses int begin and int end

Set low to first value

Set high to last value

Set mid to low plus (high minus low) divided by two

Set the pivot to mid

WHILE pivot is less than high

DECREMENT high

Swap the low values with the high values

Set temp equal to low

Set low equal to high

Set high equal to temp

Create a method for quick sort quickSort with vector course and courses in begin int end

Set low equal to begin

Set high equal to end

IF begin is greater than or equal to end

RETURN

Set low to partition with course, low, high

Call quickSort

Create a print method for course

Print courseId, courseName, coursePrerequisite

Loop through the vector

FOR i is equal to zero and I is less than course.size() increment i

Print courses

Create method inOrder BinarySearchTree Node nodes

If node is not equal to null

Look at left side

inOrder node points to left

print courseId, courseName, course.prerequisites

check next right

inOrder node points right

print courseId, courseName, course.prerequisites

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **Print the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

**Vector Runtime Analysis**

**Vector Evaluation**

The vector data structure has a runtime that coincides with the numbers of elements within the set, this would put it on an O(n) runtime. Except when inserting a course into a vector then each line has to be parsed. This would put vector parsing on an O(n^2) runtime. When printing it would also require an O(n^2) runtime since each course can have a different number of prerequisites. Overall the vector data structure has an O(n) runtime as it is completely dependent on the input size.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **If the course not equal to UINT\_MAX** | 1 | n | n |
| **Print course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **Print prerequisite information** | 1 | n | n |
| **Create node points to next** | 1 | n | n |
| **If node not equal to null** | 1 | n | n |
| **print the course information** | 1 | n | n |
| **Total Cost** | | | 8n+1 |
| **Runtime** | | | O(1) |

**Hash Table Runtime Analysis**

**Hash Table Evaluation**

The hash table data structure is very efficient when it comes to insertions, searches, printing, and removals. The hash table runs at the equivalent of an O(1) runtime, and an O(n) runtime as the worst-case scenario. The worst-case scenario only occurs if there is a collision within the data structure.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Node not equal to null** | 1 | n | n |
| **Call with left pointer** | 1 | n | n |
| **Print the course information** | 1 | n | n |
| **for each prerequisite of the course** | 1 | n\*n | n\*n |
| **print the prerequisite course information** | 1 | n | n |
| **Call with right pointer** |  |  |  |
| **Total Cost** | | | 6n \* n |
| **Runtime** | | | O(n) |

**Binary Search Tree Runtime Analysis**

**Binary Search Tree Evaluation**

The binary search tree data structure runs fairly consistently at an O(log n) runtime for insertions, deletions, searches, and printing. If the binary tree becomes unbalanced, then it could have a runtime of O(n).

**Advantages and Disadvantages**

All three data structures each have their own advantages and disadvantages. Starting with the vector data structure, while the unsorted vector can be one of the fastest options, when the vector is sorted then it slows down significantly to make it one of the slowest running data structures. The hash table operates the fastest of the three data structures giving it a big advantage, however collisions must be avoided, or it is slowed down. The advantage of a binary search tree is that it operates at the most consistent speed of the data structures, the disadvantage being that if the tree becomes unbalanced then it is significantly slowed down. Picking a data structure also heavily depends on how you are accessing the data, and how often you will be searching through the data. For frequent searching the has table may be a better option than the binary tree. The vector data structure would be the last choice as it tends to be the most inefficient of the three.

**Recommendation**

My recommendation would be to use the Hash Table as the information will need to be searched fairly frequently, and the Hash Table has the fastest runtime for constant searching compared to the other two data structures as long as the code is programmed to avoid collisions. This will work the best for the program accommodating for the possibility of the course list growing in the future. Looking through each structure reading and parsing the file, along with printing the data come out to the same measurements, so we really need to look at the insertion and deletion times. When looking at the data structures we can also see that the space required for the structures is also the same. Basing my answer off of the time it takes each data structure to go through the list, I would pick the hash table for this program.

**Resources**

Algorithm Analysis: Big O Notation Explained, Pixel Profits (2024) https://medium.com/@pixelprofits/algorithm-analysis-big-o-notation-explained-69809cc9b4e4

Introduction to Running Time Analysis https://www.teach.cs.toronto.edu/~csc110y/fall/notes/09-runtime/01-introduction.html