STUDENT NAME

CS-300: DSA: Analysis and Design

28 November 2023

**Project One**

**PART ONE: OPENING & READING THE FILE**

Open file with fstream

**IF** file does not open

**OUTPUT** failure message

**RETURN** 1

**ELSE** file is opened

**WHILE** not end of file

**IF** there’s less than two values in line

**OUTPUT** error message

**RETURN**

**ELSE** there’s at least two values in line

**IF** there is more than two values in line

**IF** third or more parameter is valid elsewhere

**RETURN**

**ELSE** third or more parameter is not valid elsewhere

**OUTPUT** error message

**RETURN**

Close file

**PART TWO: CREATING COURSE OBJECTS**

**DECLARE** class for course

**INITIALIZE** course

**INITIALIZE** course name

**INITIALIZE** prerequisites

**DECLARE VECTOR**

**LOOP** through file

**INITIALIZE** class

**ASSIGN** first line as course ID

**ASSIGN** second line as course name

**WHILE** not end of line

**ASSIGN** prerequisites

Add to vector

**PART THREE: CREATE TREE**

**DECLARE** tree

Create object of course class

**FUNCTION** to add node to tree

**IF** root is empty

Create root

**IF** node is less than root

**IF** left node doesn’t exist

**ASSIGN** node to left

**ELSE**

Add node to left

**IF** node is greater than root

**IF** right node doesn’t exist

**ASSIGN** node to right

**ELSE**

Add node to right

**FUNCTION** to assign courses

LOOP through file

Create object of course class

ASSIGN first value to course ID

ASSIGN second value to course name

WHILE not the end of line

ASSIGN prerequisites

Insert to tree

**PART FOUR: PRINT COURSE INFORMATION VECTOR**

**GET** input for course ID

**WHILE** course vector is not empty

**IF** input equals course

**OUTPUT** course ID

**OUTPUT** course name

**WHILE** course has prerequisites

**OUTPUT** prerequisites

**PART FIVE: PRINT COURSE INFORMATION HASH TABLE**

**GET** input for course ID

**ASSIGN** course ID as key

**IF** current node equals key

**OUTPUT** course ID

**OUTPUT** course name

**WHILE** course has prerequisites

**OUTPUT** prerequisites

**IF** node points to null

**RETURN** null

**ELSE** node points to not null

**IF** key equals course ID

**OUTPUT** course ID

**OUTPUT** course name

**WHILE** course has prerequisites

**OUTPUT** prerequisites

Point to next node

**PART SIX: PRINT COURSE INFORMATION TREE**

**GET** input for course ID

**ASSIGN** current node to root

**WHILE** current node is not null

**IF** current node equals input

**OUTPUT** course ID

**OUTPUT** course name

**WHILE** course has prerequisites

**OUTPUT** prerequisites

**IF** course ID is less than root

**ASSIGN** current node to left

**ELSE** course ID is greater than root

**ASSIGN** current node to right

**PART SEVEN: MENU**

**DECLARE** variable for input

**WHILE** program is running (true)

**OUTPUT** menu option one – load data structure

**OUTPUT** menu option two – print course list

**OUTPUT** menu option three – print course

**OUTPUT** menu option four – quit

**SWITCH** for option menu

Case 1: load data structure

Case 2: print entire course list

Case 3: print specific course

Case 4: **ASSIGN** running program as false

**PART EIGHT: PRINT SORTED LIST**

**PARTITION FOR SORT**

**ASSIGN** lowest to first element in vector

**ASSIGN** highest to last element in vector

**ASSIGN** midpoint to middle of vector

**ASSIGN** pivot as midpoint

**WHILE** lowest element is less than midpoint

**INCREASE** lowest element

**WHILE** midpoint is less than highest

**DECREASE** highest

**IF** lowest is greater than highest

**RETURN**

**ELSE** lowest is not greater than highest

Swap lowest for the highest

**INCREASE** lowest

**DECREASE** highest

**RETURN** highest element value

**QUICKSORT():**

**ASSIGN** midpoint

**IF** first element is greater than last element

**RETURN**

**ASSIGN** midpoint with **PARTITION**

**CALL QUICKSORT()**

**CALL QUICKSORT()** to sort midpoint

Runtime Analysis

**VECTOR**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **# of lines** | **# of time it executes** | **Total** |
| Create vector | 1 | 1 | 1 |
| Loop for each line | 1 | N | N |
| Create instance of course class | 1 | N | N |
| Assign course ID  Assign course name | 2 | N | 2n |
| While there is prerequisite | 1 | N | N |
| Assign prerequisite | 1 | N | N |
| Add another item to vector | 1 | N | N |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(N) |

**HASH TABLE**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **# of lines** | **# of time it executes** | **Total** |
| Create table | 1 | 1 | 1 |
| Assign key | 1 | N | N |
| If no node matches key | 1 | N | N |
| Assign node to key | 1 | N | N |
| Else node matches key | 1 | N | N |
| Assign node key to max  Assign key  Assign course  Assign pointer | 4 | N | 4n |
| Else | 1 | N | N |
| Find next node that’s not null | 1 | N | N |
| Add node | 1 | N | N |
| Loop for each line | 1 | N | N |
| Create instance of course | 1 | N | N |
| Assign course ID  Assign course name | 2 | N | 2n |
| While there is prerequisite | 1 | N | N |
| Assign prerequisite | 1 | N | N |
| Add course | 1 | N | N |
| **Total Cost** | | | 18n + 1 |
| **Runtime** | | | O(N) |

**TREE**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **# of lines** | **# of time it executes** | **Total** |
| If root is null | 1 | 1 | 1 |
| Create root | 1 | N | N |
| If node is less than root | 1 | N | N |
| If no node to left | 1 | N | N |
| Assign node to left | 1 | N | N |
| Else | 1 | N | N |
| Add node to left | 1 | N | N |
| If node is greater than root | 1 | N | N |
| If no node to right | 1 | N | N |
| Assign node to right | 1 | N | N |
| Else | 1 | N | N |
| Add node to right | 1 | N | N |
| Loop for each line | 1 | N | N |
| Create instance of course | 1 | N | N |
| Assign course ID  Assign course name | 2 | N | 2n |
| While there is prerequisite | 1 | N | N |
| Assign prerequisite | 1 | N | N |
| Add course | 1 | N | N |
| **Total Cost** | | | 18n + 1 |
| **Runtime** | | | O(N) |

Evaluation

All three methods have a positive and negative outcome. I am surprised by the total cost, as I thought hash table would be the fastest due to being able to search according to a key. Between the three though, a vector was the fastest with a total cost of 7n + 1 while hash table and tree tied with 18n + 1. Although the hash table has the ability to search quite quickly due to the use of a key, in the worst case it has to loop through the entire table before a key is found while the best case is that the key is found in the initial search. This is the same as a vector but, a vector has less that needs to be assigned in regards to a node. This is true as well for the tree. The biggest downside for the tree is that the height can create a factor of causing a slower runtime. Similar to the hash table, it has a good ability of searching as long as the height does not get too big. It is harder to implement than a hash table but is faster than a vector in the sense of finding a node, in the best case.

I think for my project, I will be using a vector. Although this may hinder some runtime in sorting, if implemented well, it can be made more useful than the length of implementing a binary tree which would be my second option due to its ability to search.