Eric Schooler-Davison

Aline Yurik

CS-300

24 November 2024

Project One Milestones

**File Interactions**

USING fstream to add the ability to interact with the files

OPEN file

CREATE function to load course.csv file (file path, data type)

IF file is found

WHILE the end of file has not been reached

READ each line of the file

IF the file is shorter than two values

RETURN error then EXIT

ELSE IF file at the end of the line has another line in the file with the same course number

CONTINUE

CLOSE file

**Storing objects and data structure**

INITALIZE courseId, courseName, prereq

INITALIZE vector named Course with value file path

FOR each iteration of I, while I is less than file row count, iterate

DECLARE data structure Course course

INITALIZE courseId for each ID in the file

INITALIZE courseName for each name in file

WHILE the file line has not ended

INITALIZE prereq for the file

**Search for Data**

INPUT courseId

FOR each iteration of i, while i is less than vector size, iterate i

IF courseId is found in vector

PRINT courseId, courseName, prereq

ELSE

RETURN error code

**Hash Table Pseudocode**

CONSTRUCTOR Course

CONSTRUCT variables courseID, courseName, prereq, nullptr

CLASS HashTable

CONSTRUCT buckets and variable currNode, next, key

CONSTRUCT public hash, print, and hashTable

CONSTRUCT size

CLASS hash with integer key

RETURN key % size or key divided by size remainder

CLASS insert information to the hash table

INITALIZE integer key for the given course

RETRIEVE the current node set as currNode using key

IF the current node is not nullptr

SET nextNode to the key position

ELSE IF the node is not currently in use and empty

SET the current node to equal the current input

ELSE

WHILE the node is not empty

SET current node to the next node

CLASS print all information from the hash table

FOR every node in the table

IF the key is not the max size

OUTPUT courseID, courseName, prereq

WHILE the next node exists

OUTPUT courseID, courseName, prereq

SET the node to the next node

CLASS MAIN

OPEN the csv file with course data

INITALIZE data from the csv file

TRY

FOR each row in the file

INITALIZE a structure for each item

Course course

course.courseId

course.courseName

course.prereq

CALL insert function with course

CATCH if the csv returns an error

**Tree Pseudocode**

CLASS binary tree

CREATE a root that points to nullptr

CREATE current to keep track

IF root is null, current course is root

ELSE IF is course number is less than root, add left

IF left is null, add courseID

ELSE

IF course number is less than leaf, add left

IF course number is more than leaf, add right

ELSE if course number is more than root, add to right

IF right is null, add courseID

ELSE

IF course number is less than leaf, add left IF course number is more than leaf, add right

DEFINE search and print functions

IF root is not null

IF node is smaller than the current node

OUTPUT node courseID, courseName, prereq

MOVE left

IF node is greater than the current node

OUTPUT node courseID, courseName, prereq

MOVE right

**Menu pseudocode**

FUNCTION display menu

PRINT “1: Load File

PRINT “2: Display All Files

PRINT “3: Display Selected File

PRINT “9: Exit

DECLARE int menu variable for input

SWITCH (menu)

CASE 1:

LOAD the data file into the data structure using INSERT function

CASE 2:

PRINT alphanumerically ordered list using PRINT function

CASE 3:

PRINT the course title and prereq for the selected course

CASE 9:

PRINT “Goodbye”

BREAK

DEFAULT

PRINT “Invalid Input”

RETURN error invalid input

RETURN 0

**Print Sorted List by Alpha**

DECLARE sorted print method sortedPrint(courses)

DECLARE variables sort, using char to define strings in an array

SORT array using alpha built in functions

DECLARE int alpha and int number

WHILE alpha is less than 97

Alpha++

FOR each in alpha

IF iteration is less than 97 then set the number into to ++

ELSE iterate the alpha int

RETURN courses

|  |  |  |  |
| --- | --- | --- | --- |
| **VECTOR** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total**  **Cost** |
| Declare vector | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | n | n |
| Declare vector course items (courseId, name, prereq) | 3 | n | n |
| Print the prerequisite course information | 1 | n | n |
| **Total Cost** | | | 5n+1 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **HASH TABLE** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total**  **Cost** |
| DECLARE hash table | 1 | 1 | 1 |
| DECLARE key | 1 | 1 | 1 |
| FOR each line of the file | 1 | n | n |
| RETRIEVE the current node set as currNode using key | 1 | n | n |
| IF the current node is not nullptr | 1 | n | n |
| SET nextNode to the key position | n | n | n |
| ELSE IF the node is not currently in use and empty | 1 | n | n |
| SET the current node to equal the current input | n | n | n |
| ELSE | 1 | n | n |
| WHILE the node is not empty | n | n | n |
| SET current node to the next node | n | n | n |
| **Total Cost** | | | 7n+2 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **TREE** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total**  **Cost** |
| DECLARE tree | 1 | 1 | 1 |
| DECLARE current | 1 | 1 | 1 |
| FOR each line of the file | 1 | n | n |
| IF root is null, current course is root | 1 | n | n |
| ELSE IF is course number is less than root, add left | 1 | n | n |
| IF left is null, add courseID | 1 | n | n |
| ELSE | 1 | n | n |
| IF course number is less than leaf, add left | 1 | n | n |
| IF course number is more than leaf, add right | 1 | n | n |
| ELSE if course number is more than root, add to right | 1 | n | n |
| IF right is null, add courseID | 1 | n | n |
| ELSE | 1 | n | n |
| IF course number is less than leaf, add left | 1 | n | n |
| IF course number is more than leaf, add right | 1 | n | n |
| **Total Cost** | | | 14n+2 |
| **Runtime** | | | O(n) |

**Vectors**

Vectors are a clean and simple solution for data structure sorting and handle small amounts of data incredibly well. The main drawback is that searches become exponential the more data you add to that structure. If I were to compare vectors to other data types, vectors would win only if the data structure was small enough to consider declaration and runtime of the structure itself an issue. In this case it would not be the most effective as we are sorting large data sets where return time is important for each use case.

**Hash Tables**

Hash tables require more overhead to set up but sort data structures more efficiently as multiple linked lists can be compiled into as large a structure is required for its general purpose. It may take more resources to declare and set up/run a hash table, but the speed of return would make it worth it. Unlike vectors the hash table can be expanded as you need to suit faster run times. However, the need for declared sets would require the program to iterate through each linked list header until it found the data set which can make the process take more time if the data set grows too large.

**Trees**

Trees are the most efficient way to sort extremely large data sets in my opinion. When you have a large data set, unlike a hash table where the compiler would need to parse through each line until it found the right header, the tree structures itself in such a way that it only needs to interact with a small fraction of the data. Trees do, however, have the most overhead and require more resources to set up and run, but save themselves with the compiling time needed to interact or edit the data structure. Considering the magnitude of the data used for course files, and the ability to easily adapt to larger and larger structures, trees make a strong contender.

Each data structure has its advantages and disadvantages, to me the main difference is what size and type of data you are storing. For instance, I recently was playing a video game that has an outdated hash table structure that it was using for retrieving .dat files and textures. The game did not preprocess so most files remained in .zip format until they were needed, this created unnecessary slow down for newer systems that run at much higher frame rates than what the developers intended. While a hash table was a perfect structure at 60 or 30 FPS, at 177 FPS the structure was outdated and incapable of keeping up. A mod was created to attempt to fix this by separating the data structures into a more organized format that used linked lists organized in greater numbers that preprocessed out of .zip files. My suggestion would be to use a tree as the structure itself would be fastest at retrieving data as opposed to uploading data. The main purpose of this function would be to retrieve information more than upload it, so the need for fast upload would be unnecessary. This is why in computer programming there is no such thing as a right answer, only a best answer with the information you have. At the time the game developers I mentioned chose the right structure, but as technology changed it became incapable of adapting to new systems. With the information I have I chose trees but that could change depending on the data I receive.