Jordan Poston

CS-300-10540

8/07/2025

Project One

Open and Read File:

Open file using fstream

IF return value == -1, produce Error message (File not found)

ELSE open file and read

WHILE the file has not reached the end, parse each line

IF a line has fewer than 2 values, produce Error message

ELSE read parameters

IF there are more than 2 non-redundant parameters

Continue reading

ELSE

Return Error message

Close

Course Struct:

Create struct (Course) using the following: Course ID, Course Name, Prereqs

Define vector <Course>

Loop all the way through file

Create a Course Object

WHILE not at the end of the file

DO this for each line in file

For 1st and 2nd values

Utilize push-back to add to Course Objects

IF there is a 3rd value

Utilize push-back to add value until next line

Hash Table:

Create Class HashTable to hold fields for <Course>

Declare variable CourseID, Course Name, Prereq1, Prereq2, and PrereqCount

Set PrereqCount = 0

Set a maximum value

Generate a temporary structure to hang onto these values

Generate a current element for values with current pointer pointing to next value

WHILE not at the end of the file

Loop for each line

Use temporary structure to hold 1st and 2nd values

IF there is a 3rd value

Add to course

Insert values to Course

Binary Tree:

Create class binaryTree

Set root pointing to null

Define a method to insert classes to tree

IF root = null

Course = root

ELIF course # < root

Add to left

IF left = null

Add course # in spot

ELSE

IF course # < leaf

Insert as left leaf

IF course # > leaf

Insert as right leaf

ELIF course # > root

Add to right

IF right = null

Add course # in spot

ELSE

IF course # < leaf

Insert as left leaf

IF course # > leaf

Insert as right leaf

Vector Print Method:

Prompt user for input

Loop through list for courses

IF user input == CourseID

Cout course info

For prereqs

Cout prereq course info

Hash Table Print Method:

Prompt user for input

Set input = key

IF key is located in <Course>

Cout course info

For Course prereqs

Cout prereq course info

Binary Tree Print Method:

Prompt user for input

Generate method for printing

IF root ≠ null

Traverse tree (left)

Cout course info, if found

Traverse tree (right)

Cout course info, if found

User Menu:

Set Input = 0;

WHILE input ≠ 9

Cout menu options and prompt user for input

Option 1: Load file into chosen data structure

Option 2: Organize courses into ordered list and cout list

Option 3: Cout course title and prereqs for specified course

ELSE (Option 9)

Exit Program

Output Ordered List:

Implement a partition for vector <Course>

Set lowest index to first element, set highest index to last element

Determine midPoint for index [lowPoint + ((highPoint – lowPoint) / 2)

Set pivot = midpoint

WHILE pivot < highPoint

Loop through vector by decrementing highPoint

Sort lowerValues left of pivot and higherValues to right of pivot

Set tempPoint to lowPoint

Set lowPoint to highpoint

Set highPoint to tempPoint

Sorting:

Implement quickSort function

Set lowPoint to first element, highPoint to last element, and midPoint to 0

IF first element ≥ last element

Return value

Call lowerValues to partition

Place recursive calls for quickSort

quickSort (vector, lowPoint, lowerValues)

quickSort (vector, lowerValues + 1, highPoint)

Cout method to output course listings and details (ID, name, prereqs)

End

Runtime Analysis:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary Tree |
| File Read | O(1) | O(1) – O(n) | O(log n) |
| Inserting Objects | O(n) | O(1) – O(n) | O(log n) – O(n) |
| Sorting | O(n log n) | O(n) | O(n) |

Each of the data structures has strengths and weaknesses. Vectors are fairly simple structures, which makes them slower for searching functions and they do require a sorting method. The search functions are slower because a vector must loop through every item in a list until the correct item is found. On the plus side, they boast the quickest times for adding objects and reading files.

Conversely, hash tables allow for quick queries when searching and have direct access via key, but they do require extra steps to implement a specific order for the list, and their performance can lag due to collisions. This would require some extra work implementing a list and could cause performance issues down the line.

Binary trees are unique in that they require slightly more time to insert new items and retrieve results from a search (O (log n)) than hash tables, but they naturally preserve an alphanumeric order and have effective search capabilities.

Recommendation:

I recommend the binary search tree. Looking at the runtime analysis above, binary trees do not have the quickest or most efficient-seeming runtimes, but they have some other positives that outweigh these short-comings. The ordered traversal and output of a binary tree is tailor-made for applications like this. A binary tree would be the most efficient structure for searching our list of prerequisites. Binary trees are also optimal for larger datasets, so we do not need to worry about performance lag as the set becomes larger. Vectors are not optimal for larger datasets, and hash tables require extra sets for maintaining ordered lists. This makes the binary search tree the best overall option.