Project One: Pseudocode

Parker Burkard

10/10/2025

**Structuring the Courses:**

Struct Course

String courseName

String courseId //because courses feature both integers and string

Vector <String> Prerequisites (courseName, courseId)

**Reading the data file:**

Open fstream

If file doesn’t open, print an error message

Else if found, then open the file

When reading the file

While not at the end of the file, read each line from the file

Split each line into parameters

Make certain that each line contains at least two parameters

If not, print an error message

Create a course object and set the CourseId and title from the parameters

If there are more than two parameters

For each remaining parameter, if a parameter is not a course on file, then print an error message

Else, add parameter to the Prerequisites

Close the file

**Create the Course objects:**

First Create a struct for Course that includes string Course Name and string Course ID

Initialize Course Vector through CourseInfo

Loop this through the file

WHILE the user isn’t at the end of the file

For each line in the file, for the first and second values

Use the pushback method to add a value to the Vector

IF a third value also exists

Use the pushback method to add a value into a new line

**Searching the Data Structure:**

Prompt the user for an input

Loop through the Vector

IF the input is the same as the Course ID

Print out the course information

For each prerequisite course needed

Print out the prerequisite information (courseName, courseId)

**Binary Search Tree:**

Struct Node

Course courseData

Create a key for each course

If a key is found

Return the node with the course

Else //no key entry found

Return the course

For inserting a new course

If the root is null

Create a new node with the course data and return the new node

If the new courseId is less than the courseId of the root

Insert the new courseId as the left child to the root

Else If the new courseId is greater than the courseId of the root

Insert the new courseId as the right child to the root

Return the root

**Search for the Course:**

Search courseId

If the root is null, then return null and end loop

If the courseId is the root, then return the root

Else If the courseId is smaller than the root

Search down the path of the left child and return the courseId when found

If courseId still isn’t found then return null

Else If the courseId is larger than the root

Search down the path of the right child and return the courseId when found

If courseId still isn’t found then return null

**Hash Table:**

Struct Bucket

Course courseData

Create a key for each course

Hash Insert courseData

Print the Hash table

**Main:**

String the csvPath, bidKey

Output menu and prompt user to enter a number (1, 2, 3, or 9)

IF user does not enter a correct number

Print an error message telling user that the number they picked isn’t correct

IF user does not enter a number

Print an error message telling user to enter a correct number

While user does not choose 9

IF user enters 1 //load data into file structure

Load file data

IF Binary Tree selected

Call courseData and store CSV data into Binary Search Tree bst

IF Vector

Call courseData and store CSV data into Vector courseInfo

IF Hash Table

Call courseData and store CSV data into Hash Table courseTable

IF user enters 2 //Print the list in order

If Binary Tree selected

printTree(courseId)

IF Vector selected

sortList(courseId)

printList(courseId)

IF Hash Table selected

sortTable(courseId)

printTable(courseId)

IF user enters 3 //Print the course title and prerequisites

Prompt user to enter courseName and store as courseSearch

IF Binary Tree is selected

Print courseTree(courseId) passing courseSearch

IF Vector selected

Print courseList(courseId) passing courseSearch

IF Hash Table selected

Print courseTable(courseId) passing courseSearch

IF user enters 9 //Exits program

Exit the program

**Sorting the List:**

Have the vector sort from lowest index to the highest index

IF the lowest index is greater than or equal to the highest index

Return null

Use the partition function

Set the low-end index equal to the return value called by the partition function

Quicksort passing the vector, lowest index, and low-end index

Quicksort passing the vector, low-end index +1, and highest index

**Partition Function:**

Get the lowest index and the highest index

Find the midpoint between the highest and lowest

Set the pivot point to the midpoint

Loop this until the lowest index is greater than or equal to the highest index

Go through the vector from the lowest index until a vector larger than the pivot is found

Make this the new lowest index

Go through the vector from the lowest index until a vector smaller than the pivot is found

Make this the new highest index

Swap the vectors at the new highest and lowest indexes

Increment the lowest index by one

Decrement the highest index by one

Return the highest index

**Printing:**

For Binary Tree:

Print the courseNumber and the courseName

If there are any Prerequisites for the courses

Print out each the Prerequisite(s) for each course

For Vector:

IF the input is the same as the Course ID

Print out the course information

For each prerequisite course needed

Print out the prerequisite information

For Hash Table:

Print the CourseNumber, Title, Prerequisites

**Run Time Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N) | O(log N) |
| **Search** | O(n) | O(1) – O(N) | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N) | O(N) |

**Advantages/Disadvantages:**

All three data structures have their advantages and their disadvantages. Loading data into an unsorted vector using an append method is incredibly fast but sorting it later has the slowest performance, while sorting may be fastest using a binary tree.

A hash table in theory could always operate at its average O(1) if the hash table were large enough to prevent all collisions, but since neither time or memory are infinite the table needs to be able to handle some collisions which would push the hash table somewhere between O(1) – O(N).

The binary tree will tend to operate most consistently at or near O(log N) depending on how the data is read in. If the tree becomes heavily unbalanced then the Binary Tree slows down to O(N).

Which data structure to choose depends on how much data will be accessed and how frequently. For example, if the data only needs to be loaded occasionally, there are no advantages after the initial load. If the data needs to be searched often and no updates are being made, the hash table could be better than the binary tree assuming an efficient and well-designed hash function or a very unbalanced tree.

The binary tree doesn’t need to be sorted and can be traversed in order which could save some memory if both the sorted and unsorted lists do not need to be stored. Moreover, the binary tree and the hash table will perform better and be preferable than sorting with the vector method.

**Recommendation**:

For this data structure, the best method would most likely be hash table. This would work best because the data is being searched frequently, but not being sorted/printed often. The data isn’t being updated often and mostly just being read so having a table to structure the data would work best. The next best option is the binary search tree as that is the best in sorting data (in my opinion).