Shanna Boyte

CS 300

Project One

**Pseudocode:**

**Vector:**

Create Course

courseNumber: String

courseTitle: String

coursePrereq: String

Create a Vector to hold Course object

Use fstream to open file

If file cannot be found,

Return -1 and state “Error Opening File”

Else if file can be read,

While not end of file,

Read through each line,

If there are less than two parameters in each line

Return “Error” message and continue to next line

Else read parameters,

If there is a third or more parameter

If third or more parameters are in the first parameter,

Parse line into courseNumber, courseTitle, prerequisites

Create Course object

Add Course to vector

continue

Else,

Return Error

Close file

Create new Course object

Set Course.courseNumber = courseNumber

Set Course.courseTitle = courseTitle

Set Course. coursePrereq = prerequisites

Return Course

Function loadCourses(csvPath)

Create empty vector<Course> courses

Open the CSV file at csvPath

If file not opened successfully

Display error and return empty courses

For each row in the file

Create new Course object named course

courseNumber = row[0]

courseTitle = row[1]

For each column starting at row[2] to row[end]

If column is not empty

Add row[i] to coursePrereq list

Add course to courses vector

Close the file

Return courses

End Function

Request Input for courseNumber

Loop through Vector

If courseNumber does not match any stored in Vector,

Print “Error, course number does not match”

Else,

Print out data for the course stored

courseNumber, courseTitle, coursePrereq

Return

**Hash Table**:

Structure Course

String courseNumber

String courseTitle

List<String> prerequisites

Open file at filePath

If file cannot be opened

Print "Error: Could not open file."

Return

Create empty HashTable called Courses

While not end of file

Read line from file into variable line

Create empty set called courseNumbers

For each line in Courses

Split line by comma into tokens

If number of tokens < 2

Print "Error: Invalid format. Line must contain at least 2 parameters.”

Return

courseNumber = tokens[0]

courseName = tokens[1]

Add courseNumber to Courses

For each line in courses

Split line by comma into tokens

For i from 2 to length of tokens

prerequisite = tokens[i]

If prerequisite NOT in Courses

Print "Error: Prerequisite " + prerequisite + " does not exist in course list."

Return

Return Courses

EndFunction

Create empty hash table called courseTable

For each line in Courses

Split line by comma into tokens

Create new Course object called course

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = Empty list

For i from 2 to length of tokens

Add tokens[i] to course.prerequisites

Insert course into courseTable with key = course.courseNumber

Return courseTable

Function PrintCourse (courseTable)

Create list called sortedCourseNumbers = keys of courseTable sorted alphabetically

For each courseNumber in sortedCourseNumbers

course = courseTable[courseNumber]

Print "Course Number: " + course.courseNumber

Print "Title: " + course.courseTitle

If course.prerequisites is empty

Print "Prerequisites: None"

Else

Print "Prerequisites: " + join course.prerequisites with ", "

Print newline

Return

**Binary Tree:**

Create Structure Course

String courseNumber

String courseTitle

List<String> prerequisites

End Structure

Use fstream to open file

If file cannot be opened

Print "Error: Could not open file."

Return -1

Create empty list called Courses

While not End of File

Read each line in Courses

Split line by comma into tokens

If number of tokens is less than 2

Print "Error: Invalid format. Each line must have at least a course number and title."

Else read parameters

If there are greater than 2 tokens, continue

Return Null

Define Binary Tree Class

Course data

TreeNode left

TreeNode right

End

Create Insert Method

If root is Null

Create new TreeNode

Set root.data = newCourse

Set root.left = Null

Set root.right = Null

Return root

If root is Null, current course is root

Else courseNumber is less than root, add left

If left is null, add course number

Else courseNumber is greater than root, add right

If right is null, add newCourse number

Return root

Create Print Method

Ask for Input

If root is not null

Traverse left, output if found

Traverse right, output if found

**Menu:**

Function DisplayMenu()

Repeat

Display "1. Load Data"

Display "2. Print All Courses"

Display "3. Print Specific Course"

Display "9. Exit"

Get user input

If input == 1

Call loadData

Else If input == 2

Call sortCourses()

Call printAllCourses()

Else If input == 3

Ask user for course number

Call printCourse(courseNumber)

Else If input == 9

Exit loop

Else

Display "Error"

Until input == 9

End Function

**Print list in alphanumeric order:**

**Vector:**

Create new vector<Course>

For each course in the CS program

Add the course to vector

End

Sort vector in alphanumeric order

For each Course in vector

Print Course

End

**Hash Table**:

Create empty Hash Table<Course>

For each Course in the CS program

Add the course to hashTable with courseNumber and Course

End

Initialize empty list, sortedCourses

Add courseNumber and Course to sortedCourses

End

Sort sortedCourses in alphanumeric order

For each courseNumber in sortedCourses

Print courseHashTable[courseNumber

End

**Binary Tree:**

Initialize empty Binary Tree, courseTree

For each course in the CS program

Insert Course into courseTree with courseNumber and Course

End

For each course in an inorder traversal of courseTree

Print course

End

**Evaluation:**

**Vector:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line** | **#Times Executed** | **Total Cost** |
| **Opening the file** | 1 | 1 | 1 |
| **Parsing each line** | 1 | n | n |
| **Creating Couse Objects** | 1 | n | n |
| **Inserting into Vector** | 1 | n | n |

Runtime: O(n)

**Hash Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line** | **# Times Executed** | **Total Cost** |
| **Opening the file** | **1** | **n** | **n** |
| **Parsing each line** | **1** | **n** | **n** |
| **Creating Course Objects** | **1** | **n** | **n** |
| **Inserting into Hash Table** | **1** | **n** | **n** |

**Runtime: O(n)**

**Binary Tree:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line** | **# Times Executed** | **Total Cost** |
| **Opening the file** | **1** | **1** | **1** |
| **Parsing each line** | **1** | **n** | **n** |
| **Creating Course Objects** | **1** | **n** | **n** |
| **Inserting into Binary Tree** | **Log n** | **n** | **N log n** |

**Runtime: O(n log n)**

**Analysis:**

**Vector:**

A vector is an adaptive array which allows effective component search and insert. Because it doesn't require complicated data structures or balancing methods, it is very easy to use and implement. A vector's O(1) random access time, which makes it very easy to access any element by list, is one of its main benefits. Also, the average insertion cost stays constant because resizing the vector occurs in an amortized fashion and putting elements at the end of a vector is typically O(1). Vectors do have limitations, though. For instance, because elements must be moved, insertions or deletions at locations other than the vector's end take O(n) time. Another con is that when the vector reaches its limit in size, it must be adjusted.

**Hash Table:**

When quick lookups, insertions, and deletions are needed, a hash table is a great option. These operations are very efficient since the average time complexity is O(1) when the key (in this case, a course ID) is hashed. Data is stored in hash tables using a list, and each element's value is hashed to an index in the list for easy access. A hash table's O(1) average delay for lookups and insertions is one of its greatest benefits. Hash tables do have some disadvantages, though. An extra sorting step would be necessary if the program needed to release courses in a sorted order because they don't keep any sequence of the elements. This would add O(n log n) time cost. Even with these limitations, hash tables generally very fast and would be perfect if sorting is not a major issue and the advising program needs fast access to course information based on each student's unique course ID.

**Binary Tree:**

As an ordered data structure, a Binary Tree contains a maximum of two descendants for each node, with the right being more than the parent node and the left being less. This feature automatically keeps the tree sorted, which makes it the perfect option for situations where keeping the data's order is important. For larger datasets, the time spent inserting and searching for elements in a balanced Binary Tree is O(log n), making it efficient. However, the tree could change into a linked list and the time complexity for insertion and search operations may decrease to O(n) if it becomes unbalanced (for example, if members are placed in sorted order). f keeping a sorted list of courses is necessary for the advising program, a balanced binary tree would be a great option. It guarantees effective insertion, deletion, and searching with the further advantage of automatically preserving order. Depending on the needs of the project, this might be taken into consideration because a balanced tree is more difficult to create and maintain than a vector or hash table.

**Conclusion**:

The Hash Table is the best option for the advising program based on the Big O analysis and the advisor's needs. A hash table provides O(1) average time complexity for the quick lookups and retrievals of course information that the application needs based on each student's unique course ID. In addition, the hash table is perfect for managing big datasets due to its ease of use and speed. Here, the main benefits are the speed at which courses may be retrieved using their unique IDs and the effectiveness of insertions and lookups.