CS 300

Jasmine Procaccini

Project One

4/11/25

**Pseudocode**

**1.**

**//Data Verification**

Function LoadData()

open file courseInformation

For each line in courseInformation{

//validate line format

Split into tokens separated by “,”

If < 2 tokens present:{

Print “Error: invalid input. Line skipped”

Continue to next line}

new Course object (courseNumber = token 1, courseTitle = token 2)

If > 2 tokens present:{

(Prerequisites = Tokens 3 - end)}

//validate prerequisites

If Prerequisites exist: {

For each Prerequisite in Prerequisites

If Prerequisite does not equal any courseNumber

Print “Error: invalid prerequisite. Line skipped”

Continue to next line}

//store course data (next part of pseudocode)

Add Course to Courses

}

Close file courseInformation

**2.**

**//Create Course Object**

//Create Courses vector and Course object to store data

vector <Course> Courses

Struct Course {

String courseNumber

String courseTitle

Vector<String> Prerequisites (list their courseNumbers)

}

open file courseInformation

//store course data

For each line in courseInformation{

Split into tokens separated by “,”

new Course object (courseNumber = token 1, courseTitle = token 2)

If > 2 tokens present:{

(Prerequisites = Tokens 3 - end)}

Add Course to Courses

}

Close file courseInformation

**3.**

**//Print Data Vector**

Function PrintCourseInfo()

void searchCourse(Vector<Course> courses, String courseNumber) {

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

//create hash table

Initialize an empty hash table, courseHashTable

For each course in Courses{

Add the course to courseHashTable with the courseNumber as the key and the course as the value}

Initialize an empty list, sortedKeys

For each key in courseHashTable{

Add the key to sortedKeys}

Sort sortedKeys in alphanumeric order

For each key in sortedKeys{

Print courseHashTable[key]}

//create binary tree

Define binary tree class

Create root points to null

Create Insert method

If root is null:

Current course is root

Else if courseNumber is less than root:

Add to left branch

If left is null:

Add courseNumber

Else:

If courseNumber is less than leaf, add left

If courseNumber is greater than leaf, add right

Else if courseNumber is greater than root:

Add to right branch.

//search + print for binary tree

Get input

Create PrintTreeData method

If root is not null:

Go left, output if found

Go right, output if found

Else:

Output error

**//Menu**

Int UserChoice = 0

While UserChoice is not 9:

Print “Menu:”

Print “1. Load Data”

Print “2. Print List of Courses”

Print “3. Print Course Information”

Print “9. Exit”

UserChoice = get input

IF UserChoice is 1:

Call LoadData()

ELSE IF UserChoice is 2:

If Courses is not empty:

Call PrintCourses()

Else:

Print “Please load the data into the data structure first”

ELSE IF UserChoice is 3:

If Courses is not empty:

Get user input for CourseNumber

PrintCourseInfo(CourseNumber)

Else:

Print “Please load the data into the data structure first”

ELSE IF UserChoice is 9:

Exit program

ELSE:

Print “Sorry, that choice was invalid. Please select an option from the menu.”

**//Print Course List**

//vector

Function PrintCourses()

Sort Courses in alphanumeric order

For each Course in Courses:

Print CourseTitle

//hash table

Function PrintCourses()

For each key in Courses:

Add key to SortedKeys

Sort sortedKeys in alphanumeric order

For each key in sortedKeys{

Print courseHashTable[key]}

//binary tree

Function PrintCourses()

For each course in an inorder traversal of Courses:

Print course

**Evaluation**

**Vector:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | 1(n) |

**Hash Table:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 2 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 2 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

**Binary Search Tree:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 8n + 1 |
| **Runtime** | | | O(n) |

I would recommend using the vector structure for this project. The overall cost is significantly better than the other two options. Therefore, the runtime is more efficient since there’s less to run through to perform each individual function. One disadvantage of using a vector is that you’ll have to search the list to find a specific course, iterating through each item until the match is found. This can definitely slow things down. However, the vector offers the fastest and most efficient method for reading files and adding course objects, and it’s the most straightforward. One benefit of hash tables is the ability to search lists quickly, but I think that for this project specifically the fact that hash tables don’t let the table itself be printed is a big enough negative to outweigh the positive. Binary trees allow for quicker sorting than vectors do, but are a more complicated option.