(NOTE: I genuinely have no idea what happened to the formatting for most pages past #10. I tried to fix it over the course of an hour and got nowhere.)

VECTOR:

Structure (course)

STRUCT Course:

string courseID

string courseName

list<string> prereqName

int prereqCount

END STRUCT

MAIN function:

Declare fileName to be a string

Print out a message to have the user enter the file name

Read that file name

IF the filename input is EMPTY

Set the default fileName variable to be ‘courses.csv’

Declare a boolean variable called dataLoaded and set it to FALSE

WHILE TRUE

Print out the following menu options:

1. Load Data

2. Print All Courses

3. Print Course Info

0. Exit

Prompt the user to enter a menu choice

Read the user's menu choice

SWITCH on the menu choice:

CASE 1:

IF the fileParser function is unable to load the file

Print an error stating that the file is unable to be found

ELSE IF the validCourses function call returns FALSE

Print an error stating that the course data is invalid

ELSE

Print a message saying the course data has been successfully loaded

Set dataLoaded to TRUE

CASE 2:

IF dataLoaded is FALSE

Print a message stating that course data must be loaded before printing

ELSE

Call the function to print the course list (sorted if needed)

CASE 3:

IF dataLoaded is FALSE

Print a message stating that course data must be loaded before searching

ELSE

Declare the userSearch variable to be a string

Print out the userSearch input statement to prompt for a course ID

Read that userSearch input value

IF the course ID from userSearch is not found in the data structure

Print out an error stating an invalid ID

ELSE

Call the function to print the information for the course

CASE 0:

Print a message saying "Exiting program"

Exit the loop or program

DEFAULT:

Print "Invalid menu option. Valid options are 1,2,3, or 0"

File Parsing:   
   
Open file ‘courses.csv’ as a file

IF the file does not exist:

Print out an error stating that the file does not exist and then kill the program

WHILE there ARE ines to be read within the CSV file:

Declare a variable newCourse as Course

IF the FIRST AND SECOND strings are there

SET the FIRST string to be newCourse.courseID

SET the second string to be newCourse.courseName

IF another item IS found:  
 Add that string to be newCourse.prereqNames

APPEND newCourse to courseList vector

Course Validation:

Create a temporaryCourse variable with the type being course

Create a valid variable with a BOOLEAN value of TRUE to start

Set up a for loop to validate each course

IF valid is FALSE

break the loop

Set up a loop to go from 0 to the prereqCount value

SET temporaryCourse to be equal to searchList (this is going to be the prereqName token)

IF temporaryCourse.courseID is empty

SET valid to be FALSE

Return valid

List Searching:

Create a temporaryCourse variable with the type being course

Set up a loop to go through the list of each course name

IF the string is the same as the courseID

SET the temporaryCourse value to course

Return temporaryCourse

Printing Course Info:

printInfo(string)

Create a temporaryCourse variable with the type being course

Set that temporaryCourse variable to be equal to searchList(string)

Print out the courseID

Print out the courseName

Set up a loop going from zero to prereqCount

For every course that is in prereqName

Call the printInfo function while passing in the prereqName list to print out all the prerequisites for a course

HASH TABLE:

Structure Definition:

STRUCT Course:

CourseID as a string

CourseName as a string

PrereqNames as a list of strings

PrereqCount as an integer

Default values for the bucket:

CourseID = ‘ ‘

CourseName = ‘ ‘

PrereqNames = list string

PrereqCount = 0

Hash Table Class:

STRUCT Bucket:

Course

Key

Next Pointer

Hash function()

+printAll()

+list <hashTable>

MAIN function:

Declare fileName to be a string

Print out a message to have the user enter the file name

Read that file name

IF the filename input is EMPTY

Set the default fileName variable to be ‘courses.csv’

Declare a boolean variable called dataLoaded and set it to FALSE

WHILE TRUE

Print out the following menu options:

1. Load Data

2. Print All Courses

3. Print Course Info

0. Exit

Prompt the user to enter a menu choice

Read the user's menu choice

SWITCH on the menu choice:

CASE 1:

IF the fileParser function is unable to load the file

Print an error stating that the file is unable to be found

ELSE IF the validCourses function call returns FALSE

Print an error stating that the course data is invalid

ELSE

Print a message saying the course data has been successfully loaded

Set dataLoaded to TRUE

CASE 2:

IF dataLoaded is FALSE

Print a message stating that course data must be loaded before printing

ELSE

Call the function to print the course list (sorted if needed)

CASE 3:

IF dataLoaded is FALSE

Print a message stating that course data must be loaded before searching

ELSE

Declare the userSearch variable to be a string

Print out the userSearch input statement to prompt for a course ID

Read that userSearch input value

IF the course ID from userSearch is not found in the data structure

Print out an error stating an invalid ID

ELSE

Call the function to print the information for the course

CASE 0:

Print a message saying "Exiting program"

Exit the loop or program

DEFAULT:

Print "Invalid menu option. Valid options are 1,2,3, or 0"

File Parsing:

Open the file ‘courses.csv’ as a file

IF the file does NOT exist:

Print an error stating the file does not exist and then kill the program

WHILE there ARE lines to be read in the file

Declare a variable newCourse as Course

IF the FIRST AND SECOND strings are there:

SET the FIRST string to be newCourse.courseID

SET the SECOND string to be newCourse.courseName

IF another item IS found:

Add that string to newCourse.prereqNames

INSERT newCourse into the courseTable and use the newCourse.courseID value as the key

List Searching:

Create tempCourse of type BUCKET

SET tempCourse to the bucket at the hash location of type STRING

Loop through the list for each course

IF the string is the same as the courseID

SET tempCourse to be Course

Return tempCourse

Course Validation:

Create a temporaryCourse variable with the type being Course

Create a valid variable with a BOOLEAN value of TRUE to start

Set up a FOR loop to validate each course

IF valid IS FALSE:

Break the loop

WHILE tempCourse -> next IS NOT null:

Set up a loop to go from 0 to the value of “prereqCount”

SET temporaryCourse to be equal to “searchList” (this will be the prereqName value token)

IF tempCourse courseID is empty

SET valid to be false

Return the valid variable

Print Course Info:

Create tempCourse of type BUCKET

SET tempCourse to be equal to hash(string)

Loop through all the buckets at tempCourse

Print out the CourseID

Print out the CourseName

Set up a loop going from 0 to prereqCount

For every course that is in prereqName

Call the printInfo function passing in the prereqName list to print out all prerequisites for a course

BINARY SEARCH TREE:

Structure Definition:

STRUCT Course:

CourseID as a string

CourseName as a string

PrereqNames as a list of strings

PrereqCount as an integer

Default values for the course

CourseID = ‘ ‘

CourseName = ‘ ‘

PrereqNames = list string

PrereqCount = 0

Binary Search Tree Class:

InsertNode(Course newCourse

SearchList(String courseID)

PrintCourses(String courseID)

ValidCourses()

MAIN function:

Declare fileName to be a string

Print out a message to have the user enter the file name

Read that file name

IF the filename input is EMPTY

Set the default fileName variable to be ‘courses.csv’

Declare a boolean variable called dataLoaded and set it to FALSE

WHILE TRUE

Print out the following menu options:

1. Load Data

2. Print All Courses

3. Print Course Info

0. Exit

Prompt the user to enter a menu choice

Read the user's menu choice

SWITCH on the menu choice:

CASE 1:

IF the fileParser function is unable to load the file

Print an error stating that the file is unable to be found

ELSE IF the validCourses function call returns FALSE

Print an error stating that the course data is invalid

ELSE

Print a message saying the course data has been successfully loaded

Set dataLoaded to TRUE

CASE 2:

IF dataLoaded is FALSE

Print a message stating that course data must be loaded before printing

ELSE

Call the function to print the course list (sorted if needed)

CASE 3:

IF dataLoaded is FALSE

Print a message stating that course data must be loaded before searching

ELSE

Declare the userSearch variable to be a string

Print out the userSearch input statement to prompt for a course ID

Read that userSearch input value

IF the course ID from userSearch is not found in the data structure

Print out an error stating an invalid ID

ELSE

Call the function to print the information for the course

CASE 0:

Print a message saying "Exiting program"

Exit the loop or program

DEFAULT:

Print "Invalid menu option. Valid options are 1,2,3, or 0"

File Parser Function:

Open the file ‘courses.csv’ as a file

IF the file does NOT exist:

Print an error stating the file does not exist and then kill the program

WHILE there ARE lines to be read in the file

Declare a variable newCourse as Course

IF the FIRST AND SECOND strings are there:

SET the FIRST string to be newCourse.courseID

SET the SECOND string to be newCourse.courseName

IF another item IS found:

Add that string to newCourse.prereqNames

INSERT newCourse into the courseTree and use the newCourse.courseID value as the key

Searching The List:

SearchList(Node\* tempCourse, string courseID):

IF tempCourse IS NULL:  
 return NULL

ELSE IF tempCourse.courseData.CourseID == courseID:

return the tempCourse variable

ELSE IF the courseID is LESS THAN tempCourse.courseData.CourseID:

Return the node that is to the left of the one that is being searched for

Course Validation:

ValidCourses():

Set valid to be a BOOLEAN value as TRUE to start

For EACH course within the tree:

For EACH prerequisite course within course.PrereqNames:

Set the tempCourse node to be equal to the searchList function and pass in (root, prereqList)

IF tempCourse IS NULL:

return FALSE

Return the valid variable

Printing Courses:  
 printInfo(string):

Create a tempCourse variable of type NODE

SET that tempCourse variable to be equal to searchList

IF tempCourse IS NOT NULL:

Print out the CourseID

Print out the CourseName

Set up a loop to go from 0 to the value of prereqCount

For EVERY course that is in prereqNames:

Call the printInfo function passing in the prereqNames list

RUN-TIME TABLES:  
VECTOR:

|  |  |  |  |
| --- | --- | --- | --- |
| LINE | COST | EXECUTE COUNT | TOTAL COST |
| Create local list ‘temporaryList’ | O(1) | 1 | O(1) |
| Open file as string | O(1) | 1 | O(1) |
| Loop to pull strings until EoF | O(1) | N | O(N) |
| Assign newCourse to be type Course | O(1) | N | O(N) |
| If FIRST AND SECOND strings are there | O(1) | N | O(N) |
| Set FIRST to be courseID in struct | O(1) | N | O(N) |
| Set SECOND to be courseName in struct | O(1) | N | O(N) |
| Loop to go until no more values in row | O(N) | N | O(N^2) |
| IF another IS found | O(1) | N | O(N) |
| Increment prereqCount | O(1) | N | O(N) |
| Combine string ‘prereqName” for each item | O(1) | N | O(N) |
| Add prereqCount to be the same in struct | O(1) | N | O(N) |
| Add prereqName to be the same in struct | O(1) | N | O(N) |
| Return temporaryList | O(1) | 1 | O(1) |

WORST CASE OF O(N^2)

HASH TABLE:

|  |  |  |  |
| --- | --- | --- | --- |
| LINE | COST | EXECUTE COUNT | TOTAL COST |
| Create hash table named ‘courseTable’ | O(1) | 1 | O(1) |
| Open file as string | O(1) | 1 | O(1) |
| Loop to pull strings until EoF | O(1) | N | O(N) |
| If FIRST AND SECOND strings are there | O(1) | N | O(N) |
| Set FIRST to be newCourse.courseID | O(1) | N | O(N) |
| Set SECOND to be newCourse.courseName | O(1) | N | O(N) |
| Loop to go until no more values in row | O(N) | N | O(N^2) |
| IF another IS found | O(1) | N | O(N) |
| Add new item to newCourse.prereqNames | O(1) | N | O(N) |
| Insert newCourse into the courseTable | O(N) (collisions) | N | O(N^2) (collisions) |

WORST CASE OF O(N^2)

BINARY SEARCH TREE:

|  |  |  |  |
| --- | --- | --- | --- |
| LINE | COST | EXECUTE COUNT | TOTAL COST |
| Create hash table named ‘courseTree’ | O(1) | 1 | O(1) |
| Open file as string | O(1) | 1 | O(1) |
| Assign newCourse to be type Course | O(1) | N | O(N) |
| Loop to pull strings until EoF | O(1) | N | O(N) |
| If FIRST AND SECOND strings are there | O(1) | N | O(N) |
| Set FIRST to be newCourse.courseID | O(1) | N | O(N) |
| Set SECOND to be newCourse.courseName | O(1) | N | O(N) |
| Loop to go until no more values in row | O(N) | N | O(N^2) |
| IF another IS found | O(1) | N | O(N) |
| Add new item to newCourse.prereqNames | O(1) | N | O(N) |
| Insert newCourse into the courseTree | O(log N) | N | O(N log N) |

WORST CASE OF O(N^2)

ANALYSIS:

VECTOR:

Advantages:

* Simple to implement
* Dynamic resizability
* Efficiency when displaying a full list of

Disadvantages:

* Slower linear searching for a specific element
* Data must be manually sorted in order
* Can be very memory heavy to run due to certain resizing
* Limied flexibility when it comes to insertions and deletion of data

HASH TABLES:

Advantages:

* Very fast data access
* Flexible sizing
* Efficient memory usage
* Efficient data lookup

Disadvantages:

* No proper ordering without manually setting it up
* Collisions can be extremely hard to handle (degrades performance heavily)
* Memory usage is higher than most other data structures
* Requires efficient hashing function to work well

BINARY SEARCH TREE:

Advantages:

* Decently efficient with searching, insertion, and deletion of data
* Very flexible
* Supports quick traversal due to BDS or DFS searching

Disadvantages:

* Not ideal for a fixed data set like the course list
* Can be much harder to maintain
* Insertions can cost a lot of time and memory
* Space complexity is higher due to pointers

RECOMMENDATION:

Overall, I would recommend the usage of a Vector solely because of how simple the data being worked with is. There are a few reasons why it works best with this simple data set. First off, a vector is extremely easy to implement as there is no need to search by a specific key value like you would in a Hash Table. Another good reason is that they allow for highly efficient iterations of data which helps response time and usability on lower end machines. Knowing that this is a smaller data set the main disadvantage can be easily mitigated. In this case the linear searching for elements won’t be an issue because of how small it is. If there were a few dozen times more courses to sort through, then something such as a BST would be a better idea to use being able to split a tree down the middle and go from there.