**CS 300 MODULE 6 PROJECT 1**

**Vector**

//Open file, read data, parse, error check

Open the file using a file stream

If the file opened successfully:

Initialize an empty vector to store course objects

For each line in the file:

Read the line from the file

Parse the line to extract course information and prerequisites

Validate the format of the line:

If the line does not have at least two parameters:

Display an error message indicating the format is incorrect

Continue to the next line

If there are prerequisites listed:

Check if each prerequisite exists as a course in the file:

If a prerequisite does not exist:

Display an error message indicating the missing prerequisite

Continue to the next line

Create a course object using the extracted information

Add the course object to the vector

Close the file

Else:

Display an error message indicating the file could not be opened

//Create course objects, store data structure

Initialize an empty vector to store course objects

Open the file using a file stream

If the file opened successfully:

For each line in the file:

Read the line from the file

Parse the line to extract course information and prerequisites

Create a course object using the extracted information

Add the course object to the vector

Close the file

Else:

Display an error message indicating the file could not be opened

//Search for and print course info and prerequisites

Prompt the user to enter a course number to search for

Search for the course number in the vector data structure

If the course is found:

Print the course information (course number, title)

If the course has prerequisites:

Print the list of prerequisites

Else:

Display a message indicating the course was not found

**Hash Table**

// Hashtable pseudocode

// Function to count the total number of prerequisite courses for a given course

int numPrerequisiteCourses(Hashtable<Course> courses, String courseNumber):

totalPrerequisites = 0

course = courses.get(courseNumber)

if course is null:

print "Course not found."

return 0

stack = create new Stack

stack.push(course)

while stack is not empty:

currentCourse = stack.pop()

prerequisites = currentCourse.getPrerequisites()

if prerequisites is not empty:

totalPrerequisites = totalPrerequisites + prerequisites.length

for each prerequisite in prerequisites:

prerequisiteCourse = courses.get(prerequisite)

if prerequisiteCourse is not null:

stack.push(prerequisiteCourse)

print "Total number of prerequisite courses for", courseNumber, ":", totalPrerequisites

return totalPrerequisites

// Function to print out a sample schedule of courses

void printSampleSchedule(Hashtable<Course> courses):

for each course in courses:

print course.courseNumber, ":", course.title

// Function to print course information and prerequisites

void printCourseInformation(Hashtable<Course> courses, String courseNumber):

course = courses.get(courseNumber)

if course is null:

print "Course not found."

return

print "Course Number:", course.courseNumber

print "Title:", course.title

prerequisites = course.getPrerequisites()

if prerequisites is empty:

print "Prerequisites: None"

else:

print "Prerequisites:"

for each prerequisite in prerequisites:

print " ", prerequisite

**Binary Search Tree**

Function LoadDataFromFile(filename):

Open the file with the given filename

If the file is successfully opened:

Create an empty Binary Search Tree (BST) to store course information

For each line in the file:

Read the line and split it into courseNumber, courseTitle, and prerequisites

Create a new Course object with courseNumber and courseTitle

If prerequisites is not empty:

Split prerequisites by ',' to get individual prerequisite course numbers

For each prerequisiteNumber in prerequisites:

Create a new Prerequisite object with prerequisiteNumber

Add the Prerequisite object to the Course's prerequisites list

Insert the Course object into the Binary Search Tree using the courseNumber as the key

Close the file

Return the Binary Search Tree with course information

Else:

Print an error message "Failed to open the file"

Return null

**Create course objects and store in BST**

Class Course:

Data members:

string courseNumber

string courseTitle

List<Course> prerequisites

Function CreateCourseObject(courseNumber, courseTitle, prerequisites):

Create a new Course object with courseNumber and courseTitle

If prerequisites is not empty:

Split prerequisites by ',' to get individual prerequisite course numbers

For each prerequisiteNumber in prerequisites:

Create a new Prerequisite object with prerequisiteNumber

Add the Prerequisite object to the Course's prerequisites list

Return the Course object

**Print course info**

Function Print CourseInformationAndPrerequisites(bst, courseNumber):

Set courseFound = false

For each course in bst using in-order traversal:

If the course.courseNumber is equal to courseNumber:

Print the course information (course.courseNumber and course.courseTitle)

If course.prerequisites is not empty:

Call a recursive function PrintPrerequisites(course.prerequisites)

Set courseFound = true

Break

If courseFound is false:

Print "Course not found in the list"

Function PrintPrerequisites(prerequisites):

For each prerequisite in prerequisites:

Print the prerequisite.courseNumber and prerequisite.courseTitle

If prerequisite.prerequisites is not empty:

Call PrintPrerequisites(prerequisite.prerequisites)

function load\_data\_structure(file\_name, data\_structure):

open file\_name

for each line in file:

parse line to extract course information

insert course information into data\_structure

close file

function print\_course\_list(data\_structure):

sorted\_courses = sort\_courses\_alphanumerically(data\_structure)

for each course in sorted\_courses:

print course information

function print\_course(course\_code, data\_structure):

course = find\_course(course\_code, data\_structure)

if course is not null:

print course title

print prerequisites

function main():

data\_structure = create\_data\_structure() // Choose appropriate data structure (vector, hash table, or tree)

while true:

print "Menu:"

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

choice = read\_user\_input()

choice == 1:

file\_name = read\_file\_name\_from\_user()

load\_data\_structure(file\_name, data\_structure)

else if choice == 2:

print\_course\_list(data\_structure)

else if choice == 3:

course\_code = read\_course\_code\_from\_user()

print\_course(course\_code, data\_structure)

else if choice == 4:

exit\_program()

else:

print "Invalid choice. Please choose a valid option."

// Call the main function to start the program

main()

**Evaluate the run-time and memory of data structures that could be used to address the requirements**:

For a vector, we have the task of opening and reading the file, which is (n number of courses). Per line, we would have O(1) with n number of lines. The worst-case run time would be O(n).

With creating a course object, we would have n courses, with O(1) per line and n number of lines. The worst-case runtime would be O(n).

For a hash table, opening and reading the file – n number of courses. With a constant O(1) per n lines. The worst-case runtime would still be O(n).

Creating a course object and then inserting it into the hash table. As always, n courses, with O(1) per n lines if hashing and insertion are constant, with the worst-case runtime as O(n).

For a BST, opening and reading the file with n courses. Per line would be O(1), with n lines, and a worst-case runtime of O(n).

Creating a course object and inserting it into a BST. Per line runtime – O(log n) if the BST is balanced, with n number of lines, and a worst-case runtime of O(n log n).

**Advantages and disadvantages of each structure**

A vector is simple to implement and easy to access. It is, however, a little harder to insert and delete items due to memory reallocation during insertions and deletions.

A hash table, depending on the type of search method used, is fast when inserting, deleting, or retrieving an element. However, due to how they manage collisions, can be inconvenient and might take on a lot of space.

A binary search tree has great sorting, and due to this a fast search function.

Depending on how data is read, printed, and, or searched the hash table looks like a good choice. As stated before, the speed while completing operations like inserting, retrieving, and deleting data on a hash table is quite fast. Also, prerequisites can be managed while using the hash table’s key-value structure.

Sorry the document is so long. The previous pseudocode alone took up 8+ pages. I apologize for this.