## Menu:

Function MainMenu():

While true:

Print menu options

Get user choice

If choice is "Load Data Structure":

Call LoadDataStructure("CourseInfo.txt")

Else if choice is "Print Course List":

Call PrintCourseList()

Else if choice is "Print Course":

Get course number from user

Call PrintCourse(courseNumber)

Else if choice is "Exit":

Exit program

Function PrintCourseList():

Sort courses in data structure alphanumerically

For each course in sorted list:

Print course number and title

Function PrintCourse(courseNumber):

If course exists in data structure:

Print course details

Else:

Print "Course not found”

## Hashtable Pseudocode:

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

course = courses.find(courseNumber)

If course is not found:

Return 0

totalPrerequisites = new Set()

For each prerequisite p in course.prereqs:

totalPrerequisites.add(p)

recursivelyAddPrerequisites(p, totalPrerequisites, courses)

Print totalPrerequisites.size()

Function recursivelyAddPrerequisites(String courseNumber, Set totalPrerequisites, Hashtable<Course> courses):

course = courses.find(courseNumber)

If course is not found:

Return

For each prerequisite p in course.prereqs:

If p is not in totalPrerequisites:

totalPrerequisites.add(p)

recursivelyAddPrerequisites(p, totalPrerequisites, courses)

void printSampleSchedule(Hashtable<Course> courses):

// Implementation would depend on additional details about scheduling logic

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

course = courses.find(courseNumber)

If course is found:

Print course information and prerequisites

Else:

Print "Course not found"

## Tree Pseudocode:

Function numPrerequisiteCourses(Tree<Course> courses, String courseNumber):

course = courses.find(courseNumber)

If course is not found:

Return 0

totalPrerequisites = new Set()

For each prerequisite p in course.prereqs:

totalPrerequisites.add(p)

recursivelyAddPrerequisites(p, totalPrerequisites, courses)

Print totalPrerequisites.size()

void printSampleSchedule(Tree<Course> courses):

// Implementation would depend on additional details about scheduling logic

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

course = courses.find(courseNumber)

If course is found:

Print course information and prerequisites

Else:

Print "Course not found"

## Big O(n) Analysis:

**Reading File and Creating Course Objects:**

Opening the file: O(1)

Reading each line: O(n) where n is the number of lines in the file

The worst run case for the file is O(n) where there are a lot of empty lines within the data. The best run case being O(m) where m = the lines of important data. If we wanted to shorten runtime, all we would need to do is validate our data to remove bloat.

### Data Structures Analysis:

**1. Vector:**

Advantages: Simple to implement, direct access to elements.

Disadvantages: Sorting the vector (for printing in alphanumeric order) is O(n log n), and finding a specific course is O(n) in the worst case.

**2. Hash Table:**

Advantages: Very quick insertion and lookups (O(1))

Disadvantages: No inherent order, so printing all courses alphanumerically requires collecting all keys, sorting them O(k log k) where k is the number of keys, and then accessing each course, which makes it inefficient for ordered operations.

**3. Binary Search Tree:**

Advantages: Good alphanumeric sorting and balance between insertion and sorting speeds

Disadvantages: Can become unbalanced depending on the order of insertions, potentially degrading to O(n) for lookups and insertions.

**Recommendation**

It is important we look up searches alphanumerically and since insertion and deletion aren’t as important in this project, a Tree data structure would be optimal. Trees offer balanced speeds and are consistent so it’s to find problems in other parts of the program.