Project 1

**Pseudocode:**

***Open File and Read Course/Vector Data***

class Course {

int courseID

string courseName

vector <string> PreReqs

}

1. Read Data from File {

Open File

If unable to open, print error message

If line has less than two values in it

Print error

Else read parameters

If third parameter exists

Loop until no longer remain

Else

Return error

Close File

}

1. ***Create Objects and Store Them in Data Structure***

vector <Course\*> CreateSchedule (vector<string> contents) {

Initialize course vector

Initialize string token to store data

Loop through file

While not at the end of the file

For each line in file

For the first and second value

Use pushback to add value to vector

If third parameter exists

Loop pushbacks until a new line is created

Return courses

}

1. ***Searching the Data Structure***

Ask for user input

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

Loop through vector

If input is the same as CourseID

Print information

For each PreReqs of course

Print information

Else output “Course not in schedule.”

}

***2.***

***a. Open File***

Open file “CourseInfo”

If file cannot be opened:

Print “Error”

Return

***b. Reading Data from Line***

CourseTree = new Tree <Course>()

For each line that is in the file

Split the line into parts using commas

Make token number = amount of parts

1. ***Validate number of parts***

If the length of the parts after this are less than 2:

Print “ Error: At least two parameters needed”

Return

1. ***Validate that the course exists***

If the length of the parts is greater than 2 but data could not be read:

Print “Error: Course Information could not be found or file format error”

Return

1. ***Create Course Object***

Course = new Course(tokens[0], tokens[1], tokens[2])

Add course to the CourseTree

Print “File Validated”

Return

1. ***Print Course Information***

Void printCourseInfo(Tree<Course> courses, String courseNumber) {

Print by alphabetical order

Print (“CourseNumber:” + course.courseNumber)

Print("Course Title: " + course.courseTitle)

Print("Prerequisites: " + course.prerequisites)

Else: “No Prerequisites Required”

}

***3.***

1. ***Pseudocode for opening file and parsing/reading the data***

Use file stream to open file

If value is -1, then file is not found

Else the file is found

While not at end of file:

Read every line

If value is less than 2

Return error

If value is 3 or more, but in the first parameter somewhere

Return error

Else

Continue

1. ***Pseudocode for creating course object structure***

Initialize Course structure

Loop through file while not at the end of it

For every line:

For 1st and 2nd value:

Add course ID and Name

Any value greater than 2:

Add prerequisites until line ends

End

Define Binary Tree Class

Create Root pointing to null

Insert

If root is null

This becomes the current course

If less than root add left node

If node is null

Add course number

Else

If number is less than leaf add a left node

Else

If number is greater than leaf add right node

If number is greater than root

Add right node

If node is null

Add course number

Else

If number less than lead

Add left node

If greater

Add right node

1. ***Search and print tree***

Ask for input

Search for user input

If course/s found

Print information about course and prerequisites in alphabetical order

Sort from lowest to highest

If course not found

Return error

If root is not null

Move left

Output if found

Move right

Output if found

Else

Return error

***Run Time Analysis***

***Vector:***

|  |  |  |  |
| --- | --- | --- | --- |
|  | Cost Per Line | # of Times Executed | Big O Value |
| Running Time | O(1) | O(n) | O(n) |
| Reading the File | O(1) | O(n) | O(n) |
| Creating Course Objects | O(1) | O(n) | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Cost Per Line | # of Times Executed | Big O Value |
| Running Time | O(1) | O(1) | O(1) |
| Reading the File | O(1) | O(1) | O(1) |
| Creating Course Objects | O(1) | O(n) | O(n) |

***Hash Table:***

|  |  |  |  |
| --- | --- | --- | --- |
|  | Cost Per Line | # of Times Executed | Big O Value |
| Running Time | O(log n) | O(n) | O(log n) |
| Reading the File | O(1) | O(n) | O(1) |
| Creating Course Objects | O(log n) | O(n) | (O n log n) |

***Binary Search Tree:***

***Advantages and Disadvantages:***

All these data structures have their advantages and disadvantages regarding use in code. Vectors are very fast for loading data but slow in the process of sorting. Hash tables can be large enough to prevent collisions, leading to most of their values being O(1). This means that hash lookup is significantly faster at loading sorted data than the other structures. This is due to its use of keys implemented with an array. The binary search tree tends to be near O(log n), which means that it slows down heavily when loading data but is very consistent for orders and queries. In terms of which data structure would be the best to use for the sorted data, hash tables would be the best. They offer quick and frequent lookup capabilities, as well as the ability to avoid collisions. It is quicker and more balanced than both the search tree and vector sorting.