**CS 300**

**Project One**

**Jean Karst**

**Function Signatures**

**Milestone one- Vector Data Structure**

Start Program

// Step 1: Define a Course object

Define a Course with:

- Course Number

- Course Title

- List of Prerequisites

Create an empty list called CourseList // This will store all Course objects

// Step 2: Open the file and read course data

Ask the user to enter the file name

Try to open the file

If the file cannot be opened:

Print "Error: Could not open file"

End Program

// Step 3: Read each line from the file

For each line in the file:

Split the line by commas into parts

If the number of parts is less than 2:

Print "Error: Each line must include at least a course number and title"

Skip to the next line

Create a new Course object

Set Course Number to the first part

Set Course Title to the second part

If there are more parts:

For each part after the second:

Add it to the Course's list of prerequisites

Add the Course object to CourseList

Close the file

// Step 4: Validate prerequisites

For each Course in CourseList:

For each prerequisite in the Course:

Check if there is a Course in CourseList with that Course Number

If not:

Print "Error: Prerequisite course [prerequisite] not found in the file"

// Step 5: Print Course Information

Ask the user to enter a Course Number to look up

Set found to false

For each Course in CourseList:

If the Course Number matches the user's input:

Print "Course Number: [Course Number]"

Print "Course Title: [Course Title]"

If the Course has prerequisites:

Print "Prerequisites:"

For each prerequisite:

Print " - [Prerequisite]"

Else:

Print "No prerequisites"

Set found to true

If found is false:

Print "Course not found in the list."

End Program

**Milestone 2- Hash Table Data Structure**

Function loadCourses(filePath):  
  
 Open the file  
 If file can't open, show error and stop  
  
 Create empty hash table  
  
 For each line in the file:  
 Split the line by commas into parts  
 If parts has less than 2 items:  
 Show error and skip to next line  
  
 Set courseNumber = first part  
 Set courseName = second part  
 Set prerequisites = rest of the parts  
  
 Make a Course object with courseNumber, courseName, prerequisites  
 Add Course to hash table with courseNumber as key  
  
 For each Course in hash table:  
 For each prerequisite in Course:  
 If prerequisite not in hash table:  
 Show error for missing prerequisite  
  
 Return the hash table

**Design pseudocode to show how to create course objects and store them in the appropriate data structure**.

Class Course:  
 courseNumber: String  
 courseName: String  
 prerequisites: List of Strings  
  
 Constructor(courseNumber, courseName, prerequisites):  
 Set this.courseNumber = courseNumber  
 Set this.courseName = courseName  
 Set this.prerequisites = prerequisites

**Design pseudocode that will print out course information and prerequisites**.

Function printCourse(hashTable, courseNumber):  
  
 If courseNumber not in hashTable:  
 Show "Course not found" and stop  
  
 Get the Course from hashTable  
 Print courseNumber and courseName  
  
 If prerequisites list is empty:  
 Print "No prerequisites"  
 Else:  
 Print each prerequisite's courseNumber and name (if found)

**Milestone #3- Tree Data Structure**

Start

Define Structure Course:

courseNumber: String

courseTitle: String

prerequisites: List of Strings

Define Structure TreeNode:

course: Course

left: TreeNode

right: TreeNode

Initialize BSTreeRoot as NULL

Open file "courses.txt" for reading

If file cannot be opened:

Print "Error opening file"

Exit program

For each line in file:

Split line by commas into tokens

If number of tokens < 2:

Print "Error: Each line must have at least a course number and title"

Continue to next line

Set courseNumber = tokens[0]

Set courseTitle = tokens[1]

Set prerequisites = all remaining tokens (if any)

For each prereq in prerequisites:

If prereq does not exist in file data (optional second pass or stored list):

Print "Warning: Prerequisite not found for " + courseNumber

Create Course object with courseNumber, courseTitle, and prerequisites

Call insertCourse(BSTreeRoot, Course)

Function insertCourse(node: TreeNode, newCourse: Course) returns TreeNode:

If node is NULL:

Create new TreeNode

Set node.course = newCourse

Set node.left = NULL

Set node.right = NULL

Return node

If newCourse.courseNumber < node.course.courseNumber:

node.left = insertCourse(node.left, newCourse)

Else:

node.right = insertCourse(node.right, newCourse)

Return node

Function printInOrder(node: TreeNode):

If node is not NULL:

printInOrder(node.left)

Print node.course.courseNumber + ": " + node.course.courseTitle

If node.course.prerequisites is not empty:

Print "Prerequisites: " + join prerequisites with ", "

printInOrder(node.right)

Call printInOrder(BSTreeRoot)

End

**Runtime Analysis**

# Vector Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Open file | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Parse line | 1 | n | n |
| Create course object | 1 | n | n |
| Append to vector | 1 | n | n |

# Hash Table Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Open file | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Parse line | 1 | n | n |
| Create course object | 1 | n | n |
| Insert into hash table | 1 | n | n |

# Binary Search Tree Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Open file | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Parse line | 1 | n | n |
| Create course object | 1 | n | n |
| Insert into BST | log n | n | n log n |

Each data structure offers unique strengths and trade-offs. Vectors are simple to implement and efficient for appending elements, with low memory overhead; however, they perform poorly for search operations as they require O(n) time to find elements. Hash tables provide constant-time (O(1)) average performance for insertion and lookup, making them ideal for quickly accessing data by unique keys such as course numbers. Their main drawbacks include potential hash collisions and moderate memory usage due to internal buckets. Binary Search Trees (BSTs) maintain elements in sorted order, making them suitable for in-order traversals and range queries. While balanced BSTs offer O(log n) operations for insert and search, they can degrade to O(n) in the worst case if not self-balancing. Additionally, BSTs typically consume more memory than vectors due to pointer storage in each node.

Based on the runtime analysis and the evaluation of each data structure, the hash table is the most suitable choice for storing and accessing course information. It provides the best average-case performance with constant-time (O(1)) complexity for insertions and lookups, which aligns well with the program’s need to efficiently load and retrieve courses by their unique course number. While vectors are simple and memory-efficient, their linear search time makes them less ideal for rapid access. Binary Search Trees offer sorted data and efficient traversal capabilities, but their performance can degrade significantly if the tree becomes unbalanced. Considering the advisor’s requirements for quick and reliable course retrieval, the hash table provides the optimal balance of speed, scalability, and usability.