Pseudocode for Loading Course Information:

1. Initialize an empty vector data structure to store course objects.

2. Open the course information file for reading.

- If the file does not exist or cannot be opened, display an error message and exit.

3. For each line in the file:

4. Read a line from the file.

5. Parse the line to extract course data (courseNumber, courseTitle, prerequisites).

6. Check for data format errors:

- If the line does not contain at least two parameters, display an error message and skip

this line.

- For each prerequisite mentioned at the end of the line:

- Ensure that the prerequisite exists as a course in the file (search the vector data

structure).

- If not found, display an error message and skip this line.

7. Create a new course object with the extracted course data.

8. Store the course object in the vector data structure.

9. Close the course information file.

Pseudocode for Printing Course Information and Prerequisites:

10. Prompt the user for the course number to search for.

11. Search the vector data structure for the course with the specified course number.

- If found, retrieve the course object.

- If not found, display an error message and exit.

12. Print course information and prerequisites:

- Print the course number, course title, and prerequisites for the selected course.

# Vector pseudocode

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

for each course c in courses

printCourseInformation(courses, c.courseNumber)

}

void printCourseInformation(Vector<Course> courses, String courseNumber) {

for each course c in courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

# Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for each bucket in courses

for each course c in bucket

printCourseInformation(courses, c.courseNumber)

}

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

hashcode = hash(courseNumber)

bucket = courses.buckets[hashcode]

for each course c in bucket

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

# Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

inOrderTraversal(courses.root)

}

void inOrderTraversal(Node<Course> node) {

if node is not null

inOrderTraversal(node.left)

printCourseInformation(node.course)

inOrderTraversal(node.right)

}

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

searchResult = searchTree(courses.root, courseNumber)

if searchResult is not null

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

Node<Course> searchTree(Node<Course> node, String courseNumber) {

if node is null or node.course.courseNumber is equal to courseNumber

return node

if courseNumber is less than node.course.courseNumber

return searchTree(node.left, courseNumber)

else

return searchTree(node.right, courseNumber)

}

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 8 | n | O(n\*P) |
| **if the course is the same as courseNumber** | 4 | n | O(n\*P) |
| **print out the course information** | 6 | n | O(n\*P) |
| **for each prerequisite of the course** | 6 | n | n |
| **print the prerequisite course information** | 6 | n | O(n\*P) |
| **Total Cost** | | | n \* P |
| **Runtime** | | | O(n \*p) |

1. Vector (Dynamic Array)

Advantages:

Fast Random Access: Elements can be accessed in constant time using an index.

Contiguous Memory: Memory is allocated in a contiguous block, enhancing cache locality.

Dynamic Resizing: Can dynamically resize to accommodate a variable number of elements.

Disadvantages:

Insertion/Deletion Overhead: Inserting or deleting elements in the middle of the vector can be inefficient.

Dynamic Resizing Overhead: Resizing the vector may require copying elements to a new location.

Fixed Size: Initial size needs to be defined, and resizing can be resource-intensive.

2. Hash Table

Advantages:

Fast Search/Insert/Delete: Operations are generally O(1) on average if the hash function is well-designed.

Dynamic Sizing: Can dynamically resize, adapting to the number of elements.

Disadvantages:

Hash Collisions: If poorly managed, collisions can degrade performance.

Deterministic Space: May consume more memory than needed due to hash collisions.

Hash Function Complexity: Designing an effective hash function can be challenging.

3. Tree (Binary Search Tree)

Advantages:

Ordered Structure: Allows for efficient searching and range queries.

Dynamic Structure: Can be easily adapted to insertions and deletions without dynamic resizing.

Disadvantages:

Not Balanced: Without balancing, the tree can degrade into a linked list, affecting performance.

Complexity: Implementing balancing algorithms adds complexity.

Search Time Variation: In the worst case, searching can be O(n) if the tree is unbalanced.

The choice between a hash table and a binary search tree depends on factors such as dataset size, memory constraints, and the importance of maintaining order in the data structure. If the dataset is expected to be large and search efficiency is critical, a hash table is recommended. If maintaining order is a priority and the dataset is manageable, a self-balancing binary search tree could be a good alternative.

**Open file, Read Data, Parse Each line, and Check for format errors:**

try:

open file with course information

for each line in file:

split line into tokens

if length of tokens < 2:

print "Error: Insufficient parameters on line"

else:

courseNumber = tokens[0]

title = tokens[1]

prerequisites = tokens[2:]

if not existsCourse(courseNumber):

print "Error: Course does not exist for prerequisites on line"

else:

createCourseObject(courseNumber, title, prerequisites)

except FileNotFound:

print "Error: File not found"

except Exception as e:

print "Error:", e

**Create Course Objects and store in vector data structure:**

courseVector = empty vector

function createCourseObject(courseNumber, title, prerequisites):

course = new Course(courseNumber, title, prerequisites)

add course to courseVector

function existsCourse(courseNumber):

for each course in courseVector:

if course.courseNumber == courseNumber:

return true

return false

**Print Course Information and Prerequisites for Tree Data Structures:**

function printCourseInformation(courseTree):

inOrderTraversal(courseTree.root)

function inOrderTraversal(node):

if node is not null:

print node.course information

for each prerequisite in node.course.prerequisites:

printPrerequisiteInformation(prerequisite)

inOrderTraversal(node.left)

inOrderTraversal(node.right)

function printPrerequisiteInformation(course):

print course information

for each prerequisite in course.prerequisites:

printPrerequisiteInformation(prerequisite)

Menu Pseudocode:

--------------------

Function load\_data\_structure(data\_structure, file\_path):

// Load data from the file into the specified data structure

// Implementation depends on the type of data structure

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

// Print an alphanumerically ordered list of all courses

// Implementation depends on the type of data structure

// Sort the courses before printing

Function print\_course(data\_structure, course\_number):

// Print the course title and prerequisites for the specified course

// Implementation depends on the type of data structure

Function exit\_program():

// Exit the program

Function sort\_courses(data\_structure):

// Sort the courses in the data structure in alphanumeric order

// Implementation depends on the type of data structure

Function print\_sorted\_list(data\_structure):

// Print the sorted list of courses to the display

// Implementation depends on the type of data structure

Main Program:

--------------------

Initialize data\_structure

Loop until user chooses to exit:

Display Menu

Get user choice

Switch user choice:

Case 1:

// Load Data Structure

file\_path = Prompt user for file path

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

Case 2:

// Print Course List

print\_course\_list(data\_structure)

Case 3:

// Print Course

course\_number = Prompt user for course number

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

Case 4:

// Exit

exit\_program()

Default:

Display "Invalid choice. Please choose again."

End Loop