# CS 300 Pseudocode Document

//**Menu**

MENU:

1. Load course data

2. Print alphanumerically ordered list of all courses

3. Print course title and prerequisites for a given course

9. Exit

WHILE true:

PRINT "Select an option:"

PRINT "1: Load course data"

PRINT "2: Print alphanumerically ordered list of all courses"

PRINT "3: Print course title and prerequisites for a given course"

PRINT "9: Exit"

choice = GET user input

IF choice == 1:

CALL load\_course\_data()

ELSE IF choice == 2:

CALL print\_sorted\_courses()

ELSE IF choice == 3:

CALL print\_course\_info()

ELSE IF choice == 9:

EXIT program

ELSE:

PRINT "Invalid option. Please try again."

**//Vector - Milestone 1**

FUNCTION readCourseDataFromFile(filename):

try:

file = open(filename, "r") // Open file in read mode

for each line in file:

courseData = split(line, ",")

if length(courseData) < 2:

print "Error: Insufficient parameters in line:"

continue // Skip to the next line

courseNumber = courseData[0] //assigning the data

courseTitle = courseData[1] //assigning the data

prerequisites = []

for i from 2 to length(courseData) - 1:

prerequisites.append(courseData[i]) //add the prerequisites to the list if any exist

if not validatePrerequisites(prerequisites):

print "Error: Prerequisites not found for course:", courseNumber

continue // Skip to the next line

course = createCourse(courseNumber, courseTitle, prerequisites)

addCourseToVector(course)

file.close()

except FileNotFound:

print "Error: File not found."

except IOError:

print "Error: Unable to read file."

FUNCTION validatePrerequisites(prerequisites):

for each prerequisite in prerequisites:

if not courseExists(prerequisite):

return false

return true

FUNCTION createCourse(courseNumber, courseTitle, prerequisites):

course = new Course(courseNumber, courseTitle, prerequisites)

return course

FUNCTION addCourseToVector(course):

coursesVector.append(course)

FUNCTION searchCourse(courseNumber):

for each course in coursesVector:

if course.courseNumber == courseNumber:

printCourseInfo(course)

return

print "Course not found."

FUNCTION printCourseInfo(course):

print "Course Number:", course.courseNumber

print "Course Title:", course.courseTitle

print "Prerequisites:", course.prerequisites

FUNCTION print\_sorted\_courses():

SORT courses BY course\_id

FOR each course IN courses:

PRINT course.course\_id, course.course\_title

**//Hash Table - Milestone 2**

struct Course {

String courseNumber

String name

List<String> prerequisites

}

FUNCTION loadCourseData(String filename) {

// Open the file

file = open(filename)

// Initialize an empty list to store the lines of the file

List<String> lines = empty list

// Read each line from the file

while file has next line {

line = read line from file

// Add the line to the list

lines.append(line)

}

// Close the file

close(file)

// Return the list of lines

return lines

}

// Function to validate the course data

FUNCTION validateCourseData(List<String> lines) {

// Initialize a set to store all course numbers

Set<String> courseNumbers = empty set

// First pass: Collect all course numbers

for each line in lines {

// Split the line by commas

tokens = split(line, ',')

// There should be at least two tokens (courseNumber and name)

if length(tokens) < 2 {

print "Error: Each line must contain at least a course number and a name"

return false

}

// Add the course number to the set

courseNumber = tokens[0]

courseNumbers.add(courseNumber)

}

// Second pass: Validate prerequisites

for each line in lines {

// Split the line by commas

tokens = split(line, ',')

// Extract prerequisites (if any)

for i from 2 to length(tokens) - 1 {

prerequisite = tokens[i]

// Check if the prerequisite exists in the set of course numbers

if not courseNumbers.contains(prerequisite) {

print "Error: Prerequisite " + prerequisite + " does not exist"

return false

}

}

}

// If all validations pass

return true

}

// Function to create a hash table and store courses

FUNCTION storeCoursesInHashTable(List<String> lines) {

// Initialize an empty hash table

HashTable<String, Course> courseTable = new HashTable()

// For each line in the list of lines

for each line in lines {

// Split the line by commas

tokens = split(line, ',')

// Extract course number and name

courseNumber = tokens[0]

name = tokens[1]

// Extract prerequisites

List<String> prerequisites = empty list

for i from 2 to length(tokens) - 1 {

prerequisites.append(tokens[i])

}

// Create a new Course object

Course course = new Course()

course.courseNumber = courseNumber

course.name = name

course.prerequisites = prerequisites

// Insert the course into the hash table

courseTable.insert(courseNumber, course)

}

return courseTable

}

// Function to print course information and prerequisites

FUNCTION printCourseInformation(HashTable<String, Course> courseTable) {

// For each key in the hash table

for each courseNumber in courseTable.keys() {

// Retrieve the course object

Course course = courseTable.get(courseNumber)

// Print course information

print "Course Number: " + course.courseNumber

print "Course Name: " + course.name

print "Prerequisites: "

// Print prerequisites

if length(course.prerequisites) == 0 {

print "None"

} else {

for each prerequisite in course.prerequisites {

print prerequisite

}

}

print "" // Print a blank line for spacing

}

}

// Main program function

FUNCTION main() {

// Load the course data from the file

List<String> lines = loadCourseData("courses.txt")

// Validate the course data

if validateCourseData(lines) {

// Store courses in the hash table

HashTable<String, Course> courseTable = storeCoursesInHashTable(lines)

// Print course information and prerequisites

printCourseInformation(courseTable)

} else {

print "Invalid course data format."

}

}

FUNCTION print\_sorted\_courses():

SORT course\_ids IN courses BY course\_id

FOR each course\_id IN sorted course\_ids:

course = courses[course\_id]

PRINT course.course\_id, course.course\_title

**//Binary Search Tree – Milestone 3**

FUNCTION loadCourseData(fileName):

Open file with fileName in read mode

If file cannot be opened:

Print "Error: Cannot open file"

Return

Initialize an empty list 'lines'

While not end of file:

Read a line from the file

Append the line to 'lines'

Close the file

Return 'lines'

FUNCTION parseAndValidateData(lines):

Initialize an empty dictionary 'courseDict'

For each line in 'lines':

Split line by commas into a list 'courseData'

If length of 'courseData' is less than 2:

Print "Error: Invalid course data format"

Return None

courseNumber = courseData[0]

courseName = courseData[1]

prerequisites = courseData[2:] (if any)

For each prerequisite in 'prerequisites':

If prerequisite not in 'courseDict':

Print "Error: Prerequisite", prerequisite, "for course", courseNumber, "not found"

Return None

Add courseNumber, courseName, and prerequisites to 'courseDict'

Return 'courseDict'

Class Course:

FUNCTION \_\_init\_\_(self, courseNumber, courseName, prerequisites):

self.courseNumber = courseNumber

self.courseName = courseName

self.prerequisites = prerequisites

Class TreeNode:

Function \_\_init\_\_(self, course):

self.course = course

self.left = None

self.right = None

Class BinarySearchTree:

FUNCTION \_\_init\_\_(self):

self.root = None

FUNCTION insert(self, course):

If self.root is None:

self.root = TreeNode(course)

Else:

self.\_insert(self.root, course)

FUNCTION \_insert(self, node, course):

If course.courseNumber < node.course.courseNumber:

If node.left is None:

node.left = TreeNode(course)

Else:

self.\_insert(node.left, course)

Else:

If node.right is None:

node.right = TreeNode(course)

Else:

self.\_insert(node.right, course)

FUNCTION buildCourseTree(courseDict):

bst = BinarySearchTree()

For each courseNumber in courseDict:

courseName = courseDict[courseNumber]['courseName']

prerequisites = courseDict[courseNumber]['prerequisites']

course = Course(courseNumber, courseName, prerequisites)

bst.insert(course)

Return bst

FUNCTION printCourseTree(node):

If node is not None:

printCourseTree(node.left)

Print "Course Number:", node.course.courseNumber

Print "Course Name:", node.course.courseName

If node.course.prerequisites is not empty:

Print "Prerequisites:", ', '.join(node.course.prerequisites)

Else:

Print "Prerequisites: None"

printCourseTree(node.right)

FUNCTION main(fileName):

lines = loadCourseData(fileName)

If lines is None:

Return

courseDict = parseAndValidateData(lines)

If courseDict is None:

Return

bst = buildCourseTree(courseDict)

printCourseTree(bst.root)

FUNCTION in\_order\_traversal(node):

IF node IS NOT NULL:

in\_order\_traversal(node.LEFT)

PRINT node.course.course\_id, node.course.course\_title

in\_order\_traversal(node.RIGHT)

FUNCTION print\_sorted\_courses():

in\_order\_traversal(courses.ROOT)

## Evaluation of Runtime and Memory

### Data Structure Analysis

#### Vector

**Load Course Data**: O(n)

* **Adding course to vector**: O(1) (due to dynamic array resizing)
* **Memory Usage**: O(n) for storing courses
* **Advantages**: Simple to implement, efficient for sequential access.
* **Disadvantages**: Not efficient for lookups or deletions (O(n) time).

**Cost**:

* Opening the file: O(1)
* Loop: O(n)
  + Parsing: O(1)
  + Creating object: O(1)
  + Adding to vector: O(1) amortized
* Closing the file: O(1)

**Total Runtime**: O(n)

#### Hash Table

**Load Course Data**: O(n)

* **Adding course to hash table**: O(1) average case
* **Memory Usage**: O(n) for storing courses, plus extra for hash table structure
* **Advantages**: Efficient lookups, insertions, and deletions (O(1) average case).
* **Disadvantages**: Potential for hash collisions, more complex implementation, uses more memory for maintaining hash table.

**Cost**:

* Opening the file: O(1)
* Loop: O(n)
  + Parsing: O(1)
  + Creating object: O(1)
  + Adding to hash table: O(1) average
* Closing the file: O(1)

**Total Runtime**: O(n)

#### Binary Search Tree (BST)

**Load Course Data**: O(n log n) average case

* **Adding course to BST**: O(log n) average case
* **Memory Usage**: O(n) for storing courses, plus additional for tree pointers
* **Advantages**: Maintains sorted order, efficient lookups and deletions (O(log n) average case).
* **Disadvantages**: Worst-case O(n) time for unbalanced tree, more complex implementation.

**Cost**:

* Opening the file: O(1)
* Loop: O(n)
  + Parsing: O(1)
  + Creating object: O(1)
  + Adding to BST: O(log n) average
* Closing the file: O(1)

**Total Runtime**: O(n log n) average

### Recommendation

**Hash Table** is the recommended data structure based on the analysis. Here are the reasons:

* **Efficiency**: Hash tables provide average-case O(1) time complexity for insertions, lookups, and deletions, which is optimal for the requirements of the advising program.
* **Handling Large Data**: For a large number of courses, hash tables handle the operations efficiently without the risk of becoming unbalanced as with a BST.

***Operating Platform***

**Recommendation:** Linux Server Platform

**Reason:** Linux is a robust, scalable, and cost-effective server platform. It offers excellent performance, security, and support for various programming languages and frameworks. Additionally, Linux servers are highly customizable and suitable for distributed environments, which is crucial for expanding the game across different operating systems.

***Operating System Architectures***

Linux OS Architecture:

Kernel: The core component of Linux, managing system resources, and hardware communication.

System Libraries: Provide a standard interface for software applications to interact with the kernel.

System Utilities: Include essential tools for system maintenance and administration.

User Space: The environment where user applications and services run, isolated from direct hardware access, enhancing security and stability.

#### Storage Management

**Recommendation**: Use of a Distributed File System (DFS)

* **Reason**: A DFS like Hadoop Distributed File System (HDFS) or GlusterFS provides redundancy, fault tolerance, and scalability, essential for handling the game's data across multiple servers and locations.

#### Memory Management

***Linux Memory Management****:*

* **Virtual Memory**: Uses a combination of RAM and disk space to provide a larger address space than physically available memory.
* **Paging**: Divides memory into pages, allowing for efficient memory allocation and deallocation.

#### Distributed Systems and Networks

**Recommendation**: Use of Microservices Architecture with RESTful APIs

* **Reason**: Microservices architecture divides the application into smaller, independent services that communicate over a network using standard protocols such as HTTP/HTTPS. This approach ensures flexibility, scalability, and ease of maintenance.

#### Security

**Security Measures**:

* **Data Encryption**: Use of SSL/TLS for encrypting data in transit between clients and servers.
* **Authentication and Authorization**: Implementation of OAuth2.0 or JWT for secure access control.