**CS-300**

**Project One**

**Jorge Cisneros Rodriguez**

**SNHU**

**Pseudocode and Runtime Analysis**

**Opening and Reading the File**

**Vector**

// Define a function to load course data from a file into a vector

Function loadCoursesIntoVector(filePath: String) -> Vector<Course>:

courses = Vector<Course>()

file = open(filePath, "r")

If file is null:

Print "Error: File cannot be opened"

Return courses

While not end of file:

line = readLine(file)

parts = split(line, ",")

If length(parts) < 2:

Print "Error: Invalid file format"

Continue

courseNumber = parts[0]

courseName = parts[1]

prerequisites = Vector<String>()

For i from 2 to length(parts) - 1:

prerequisites.append(parts[i])

course = Course(courseNumber, courseName, prerequisites)

courses.append(course)

close(file)

For each course in courses:

For each prerequisite in course.prerequisites:

If not courseExists(prerequisite, courses):

Print "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist."

Return courses

Function courseExists(courseNumber: String, courses: Vector<Course>) -> Boolean:

For each course in courses:

If course.courseNumber == courseNumber:

Return True

Return False

**Hash Table**

// Define a function to load course data from a file into a hash table

Function loadCoursesIntoHashTable(filePath: String) -> HashTable<Course>:

courses = HashTable<Course>()

file = open(filePath, "r")

If file is null:

Print "Error: File cannot be opened"

Return courses

While not end of file:

line = readLine(file)

parts = split(line, ",")

If length(parts) < 2:

Print "Error: Invalid file format"

Continue

courseNumber = parts[0]

courseName = parts[1]

prerequisites = Vector<String>()

For i from 2 to length(parts) - 1:

prerequisites.append(parts[i])

course = Course(courseNumber, courseName, prerequisites)

courses.put(courseNumber, course)

close(file)

For each course in courses.values():

For each prerequisite in course.prerequisites:

If not courseExists(prerequisite, courses):

Print "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist."

Return courses

Function courseExists(courseNumber: String, courses: HashTable<Course>) -> Boolean:

Return courses.containsKey(courseNumber)

**Binary Search Tree (BST)**

// Define a function to load course data from a file into a binary search tree

Function loadCoursesIntoTree(filePath: String) -> BinarySearchTree<Course>:

coursesTree = BinarySearchTree<Course>()

file = open(filePath, "r")

If file is null:

Print "Error: File cannot be opened"

Return coursesTree

While not end of file:

line = readLine(file)

parts = split(line, ",")

If length(parts) < 2:

Print "Error: Invalid file format"

Continue

courseNumber = parts[0]

courseName = parts[1]

prerequisites = Vector<String>()

For i from 2 to length(parts) - 1:

prerequisites.append(parts[i])

course = Course(courseNumber, courseName, prerequisites)

coursesTree.insert(course)

close(file)

For each course in coursesTree:

For each prerequisite in course.prerequisites:

If not courseExists(prerequisite, coursesTree):

Print "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist."

Return coursesTree

Function courseExists(courseNumber: String, coursesTree: BinarySearchTree<Course>) -> Boolean:

Return coursesTree.contains(courseNumber)

**Struct Definition for Course**

// Define a struct to hold course information

Struct Course:

courseNumber: String

courseName: String

prerequisites: Vector<String>

// Constructor

Function Course(courseNumber: String, courseName: String, prerequisites: Vector<String>):

this.courseNumber = courseNumber

this.courseName = courseName

this.prerequisites = prerequisites

**Searching and Printing Course Information**

// Define a function to search and print course information

Function searchCourse(courses: Vector<Course>, courseNumber: String):

For each course in courses:

If course.courseNumber == courseNumber:

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.courseName

If length(course.prerequisites) > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " " + prerequisite

Else:

Print "No prerequisites"

Return

Print "Course not found"

Function printAllCourses(courses: Vector<Course>):

sortedCourses = sort(courses by courseNumber)

For each course in sortedCourses:

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.courseName

If length(course.prerequisites) > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " " + prerequisite

Else:

Print "No prerequisites"

#### Hash Table

// Define a function to search and print course information

Function searchCourse(courses: HashTable<Course>, courseNumber: String):

If courses.containsKey(courseNumber):

course = courses.get(courseNumber)

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.courseName

If length(course.prerequisites) > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " " + prerequisite

Else:

Print "No prerequisites"

Else:

Print "Course not found"

Function printAllCourses(courses: HashTable<Course>):

sortedCourses = sort(courses.values() by courseNumber)

For each course in sortedCourses:

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.courseName

If length(course.prerequisites) > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " " + prerequisite

Else:

Print "No prerequisites"

#### Binary Search Tree (BST)

// Define a binary search tree class to store courses

Class BinarySearchTree<T>:

root: Node<T>

Class Node<T>:

data: T

left: Node<T>

right: Node<T>

Function Node(data: T):

this.data = data

this.left = null

this.right = null

Function BinarySearchTree():

this.root = null

Function insert(data: T):

If this.root == null:

this.root = Node(data)

Else:

insertNode(this.root, data)

Function insertNode(node: Node<T>, data: T):

If data < node.data:

If node.left == null:

node.left = Node(data)

Else:

insertNode(node.left, data)

Else:

If node.right == null:

node.right = Node(data)

Else:

insertNode(node.right, data)

Function contains(data: T) -> Boolean:

Return containsNode(this.root, data)

Function containsNode(node: Node<T>, data: T) -> Boolean:

If node == null:

Return False

If data == node.data:

Return True

ElseIf data < node.data:

Return containsNode(node.left, data)

Else:

Return containsNode(node.right, data)

Function inOrderTraversal(node: Node<T>, visit: Function(T)):

If node != null:

inOrderTraversal(node.left, visit)

visit(node.data)

inOrderTraversal(node.right, visit)

Function forEach(visit: Function(T)):

inOrderTraversal(this.root, visit)

Function searchCourse(coursesTree: BinarySearchTree<Course>, courseNumber: String):

course = coursesTree.find(courseNumber)

If course is not null:

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.courseName

If length(course.prerequisites) > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " " + prerequisite

Else:

Print "No prerequisites"

Else:

Print "Course not found"

Function printAllCourses(coursesTree: BinarySearchTree<Course>):

visit = Function(course: Course):

Print "Course Number: " + course.courseNumber

Print "Course Name: " + course.courseName

If length(course.prerequisites) > 0:

Print "Prerequisites:"

For each prerequisite in course.prerequisites:

Print " " + prerequisite

Else:

Print "No prerequisites"

coursesTree.forEach(visit)

### Pseudocode for a Menu

// Define the menu function

Function menu():

coursesVector = Vector<Course>()

coursesHashTable = HashTable<Course>()

coursesTree = BinarySearchTree<Course>()

Repeat:

Print "Menu Options:"

Print "1. Load file data into the data structure"

Print "2. Print an alphanumerically ordered list of all the courses"

Print "3. Print the course title and the prerequisites for any individual course"

Print "9. Exit"

choice = readInt()

If choice == 1:

Print "Enter file path:"

filePath = readString()

coursesVector = loadCoursesIntoVector(filePath)

coursesHashTable = loadCoursesIntoHashTable(filePath)

coursesTree = loadCoursesIntoTree(filePath)

ElseIf choice == 2:

Print "Choose data structure: 1. Vector, 2. Hash Table, 3. Tree"

structureChoice = readInt()

If structureChoice == 1:

printAllCourses(coursesVector)

ElseIf structureChoice == 2:

printAllCourses(coursesHashTable)

ElseIf structureChoice == 3:

printAllCourses(coursesTree)

ElseIf choice == 3:

Print "Enter course number:"

courseNumber = readString()

Print "Choose data structure: 1. Vector, 2. Hash Table, 3. Tree"

structureChoice = readInt()

If structureChoice == 1:

searchCourse(coursesVector, courseNumber)

ElseIf structureChoice == 2:

searchCourse(coursesHashTable, courseNumber)

ElseIf structureChoice == 3:

searchCourse(coursesTree, courseNumber)

ElseIf choice == 9:

Print "Exiting program."

Break

Else:

Print "Invalid choice. Please try again."

**Runtime Analysis**

| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree** |
| --- | --- | --- | --- |
| Load Data | O(n) | O(n) | O(n log n) |
| Search for a Course | O(n) | O(1) | O(log n) |
| Print All Courses | O(n log n) (for sorting) | O(n log n) (for sorting) | O(n) |
| Memory Usage | O(n) | O(n) | O(n) |
| Insert Course | O(1) | O(1) | O(log n) |
| Validate Prerequisites | O(n^2) | O(n) | O(n log n) |

### Advantages and Disadvantages

#### Vector

* **Advantages**:
  + Simple to implement and use.
  + Provides dynamic resizing.
* **Disadvantages**:
  + Linear time complexity for search operations.
  + Inefficient for large datasets due to O(n) search time.

#### Hash Table

* **Advantages**:
  + Constant time complexity for search operations.
  + Efficient memory usage.
* **Disadvantages**:
  + Collisions can affect performance.
  + Does not maintain order of elements.

#### Binary Search Tree

* **Advantages**:
  + Maintains order of elements.
  + Logarithmic time complexity for search and insert operations.
* **Disadvantages**:
  + Can degrade to linear time complexity if not balanced.
  + More complex to implement than a vector or hash table.

### Recommendation

Based on the Big O analysis and the requirements of the program, I recommend using a **Hash Table** for storing course information. The constant time complexity for search operations and efficient memory usage make it the best choice for quickly retrieving course data and validating prerequisites.