Project One

Mohamed Aziz Zaghdoudi

# Vector

Struct Course

String courseNumber

String name

List<String> prerequisites

End Struct

Open and Read File

Function ReadFile(String filename)

List<String> lines = []

File file = Open(filename)

While (file has more lines)

String line = file.ReadLine()

lines.Append(line)

End While

file.Close()

Return lines

End Function

Parse and Validate Data

Function ParseAndValidateData(List<String> lines)

Dictionary<String, Course> courses = {}

Set<String> courseNumbers = Set()

List<String> errors = []

// First Pass: Parse lines and create course objects

For Each line in lines

List<String> tokens = line.Split(',')

// Check for at least two parameters (courseNumber and name)

If tokens.Length < 2

errors.Append("Error: Line does not have at least two parameters: " + line)

Continue

String courseNumber = tokens[0].Trim()

String name = tokens[1].Trim()

List<String> prerequisites = []

For i = 2 to tokens.Length - 1

String prerequisite = tokens[i].Trim()

prerequisites.Append(prerequisite)

// Create course object

Course course = new Course

course.courseNumber = courseNumber

course.name = name

course.prerequisites = prerequisites

// Add to dictionary and set

courses[courseNumber] = course

courseNumbers.Add(courseNumber)

End For

// Second Pass: Validate prerequisites

For Each course in courses.Values

For Each prerequisite in course.prerequisites

If Not courseNumbers.Contains(prerequisite)

errors.Append("Error: Prerequisite does not exist: " + prerequisite + " for course " + course.courseNumber)

End If

End For

End For

If errors.Length > 0

For Each error in errors

Print(error)

End For

Return null

Else

Return courses

End If

End Function

Store Course Objects in Vector

Function StoreCourses(Dictionary<String, Course> courses)

List<Course> courseList = []

For Each course in courses.Values

courseList.Append(course)

End For

Return courseList

End Function

Search and Print Course Information

Function SearchAndPrintCourse(List<Course> courseList, String courseNumber)

For Each course in courseList

If course.courseNumber == courseNumber

Print("Course Number: " + course.courseNumber)

Print("Name: " + course.name)

If course.prerequisites.Length > 0

Print("Prerequisites: " + String.Join(", ", course.prerequisites))

Else

Print("Prerequisites: None")

End If

Return

End If

End For

Print("Course not found: " + courseNumber)

End Function

# Hash Table

1. Open the File, Read Data, Parse Lines, and Check for Format Errors

Procedure LoadDataIntoHashTable(file\_path):

Open the file

file = OpenFile(file\_path, "r")

Initialize a hash table

hash\_table = CreateHashTable()

Initialize an empty list to track course numbers for validation

course\_numbers = []

Read the file line by line

while line = ReadLine(file):

Split the line into parts based on commas

parts = Split(line, ",")

Check if there are at least two parts (course number and title)

if Length(parts) < 2:

Print("Error: Line does not have at least two parameters.")

Continue

Extract the course number, title, and prerequisites

course\_number = Trim(parts[0])

course\_title = Trim(parts[1])

prerequisites = []

if Length(parts) > 2:

prerequisites = parts[2:Length(parts)]

prerequisites = Map(Trim, prerequisites)

Add the course number to the list for validation

Append(course\_numbers, course\_number)

Create a course object and store in hash table

course = CreateCourse(course\_number, course\_title, prerequisites)

InsertIntoHashTable(hash\_table, course\_number, course)

Validation of prerequisites

for course in Values(hash\_table):

for prereq in course.prerequisites:

if prereq not in course\_numbers:

Print("Error: Prerequisite", prereq, "for course", course.course\_number, "does not exist.")

Close the file

CloseFile(file)

Return hash\_table

2. Create Course Objects and Store in the Hash Table

Procedure CreateCourse(course\_number, course\_title, prerequisites):

course = NewCourse()

course.number = course\_number

course.title = course\_title

course.prerequisites = prerequisites

Return course

3. Print Course Information and Prerequisites from the Hash Table

Procedure PrintCourseInformation(hash\_table):

Iterate over each course in the hash table

for course\_number, course in Items(hash\_table):

Print("Course Number:", course.number)

Print("Course Title:", course.title)

if Length(course.prerequisites) == 0:

Print("Prerequisites: None")

else:

Print("Prerequisites:", Join(course.prerequisites, ", "))

Print("") Blank line for separation

# Tree

Pseudocode for Opening the File and Reading Data

Function LoadCoursesFromFile(filePath)

// Open the file for reading

Open file at filePath

// Initialize an empty dictionary to store course information

courses = new Dictionary()

// Initialize a list to keep track of prerequisites for validation

prereqCheckList = new List()

// Loop through each line in the file

While not End of file

// Read a line from the file

line = Read line from file

// Split the line into parts by comma

parts = Split line by ","

// Check if the line has at least two parts (course number and title)

If Length of parts < 2

Print "Error: Line does not have enough parameters."

Continue to next iteration

// Extract course number, title, and prerequisites

courseNumber = parts[0]

courseTitle = parts[1]

prerequisites = parts[2:] // All parts after the second one are prerequisites

// Create a course object

course = new Course(courseNumber, courseTitle, prerequisites)

// Add course to the dictionary

courses[courseNumber] = course

// Add prerequisites to the check list

For each prereq in prerequisites

prereqCheckList.Add(prereq)

End While

// Validate prerequisites

For each prereq in prereqCheckList

If prereq not in courses

Print "Error: Prerequisite " + prereq + " does not exist as a course."

Return courses

End Function

Pseudocode for Course Object and Data Structure

Class Course

Attributes:

courseNumber

courseTitle

prerequisites

Function Course(courseNumber, courseTitle, prerequisites)

this.courseNumber = courseNumber

this.courseTitle = courseTitle

this.prerequisites = prerequisites

End Class

// Define Binary Search Tree Node structure

Class Node

Attributes:

course

left

right

Function Node(course)

this.course = course

this.left = null

this.right = null

End Class

// Define Binary Search Tree structure

Class BinarySearchTree

Attributes:

root

Function BinarySearchTree()

this.root = null

Function Insert(course)

If root is null

root = new Node(course)

Else

addNode(root, course)

Function addNode(node, course)

If course.courseNumber < node.course.courseNumber

If node.left is null

node.left = new Node(course)

Else

addNode(node.left, course)

Else

If node.right is null

node.right = new Node(course)

Else

addNode(node.right, course)

Function InOrder()

inOrderTraversal(root)

Function inOrderTraversal(node)

If node is not null

inOrderTraversal(node.left)

Print "Course: " + node.course.courseNumber + ", Title: " + node.course.courseTitle

Print "Prerequisites: " + Join(node.course.prerequisites, ", ")

inOrderTraversal(node.right)

End Class

Pseudocode for Processing the File and Printing Course Information

Function Main()

// Define the file path

filePath = "courses.txt"

// Load courses from the file

courses = LoadCoursesFromFile(filePath)

// Create a Binary Search Tree

bst = new BinarySearchTree()

// Insert each course into the BST

For each course in courses

bst.Insert(courses[course])

// Print course information in order

bst.InOrder()

End Function

# Printing out the list of courses

// Function to load data into a vector

function loadFileDataToVector(fileName)

vector = new Vector()

open file with fileName

for each line in file

parse line

if line has formatting errors

print error message

else

course = createCourse(line)

vector.add(course)

close file

return vector

// Function to load data into a hash table

function loadFileDataToHashTable(fileName)

hashTable = new HashTable()

open file with fileName

for each line in file

parse line

if line has formatting errors

print error message

else

course = createCourse(line)

hashTable.put(course.courseNumber, course)

close file

return hashTable

// Function to load data into a tree

function loadFileDataToTree(fileName)

tree = new Tree()

open file with fileName

for each line in file

parse line

if line has formatting errors

print error message

else

course = createCourse(line)

tree.insert(course)

close file

return tree

// Function to create a course object from a parsed line

function createCourse(line)

split line by delimiter

courseNumber = splitLine[0]

courseTitle = splitLine[1]

prerequisites = splitLine[2 to end]

return new Course(courseNumber, courseTitle, prerequisites)

// Function to print the sorted course list for vector

function printSortedCourseListFromVector(vector)

sort vector by courseNumber

for each course in vector

print "Course Number: " + course.courseNumber + ", Course Title: " + course.courseTitle

// Function to print the sorted course list for hash table

function printSortedCourseListFromHashTable(hashTable)

sortedKeys = hashTable.keys().sort()

for each courseNumber in sortedKeys

course = hashTable.get(courseNumber)

print "Course Number: " + course.courseNumber + ", Course Title: " + course.courseTitle

// Function to print the sorted course list for tree

function printSortedCourseListFromTree(tree)

sortedCourses = tree.inOrderTraversal()

for each course in sortedCourses

print "Course Number: " + course.courseNumber + ", Course Title: " + course.courseTitle

# Menu Options

function showMenu()  
 print "1. Load file data"  
 print "2. Print course list"  
 print "3. Print course information"  
 print "9. Exit"  
   
function handleMenuSelection(option)  
 switch option  
 case 1:  
 loadFileData()  
 case 2:  
 printCourseList()  
 case 3:  
 printCourseInformation()  
 case 9:  
 exit program

# Evaluation of Data Structures

Vector  
  
 Analysis  
- Time Complexity:  
 - Reading and parsing file: O(n)  
 - Creating course objects: O(1) per course  
 - Sorting: O(n log n)  
 - Searching: O(n)  
- Space Complexity: O(n) for storing courses  
  
 Advantages  
- Simple implementation  
- Good for small to medium-sized datasets  
  
 Disadvantages  
- Inefficient search and insert operations for large datasets  
  
Hash Table  
  
Analysis  
-Time Complexity:  
 - Reading and parsing file: O(n)  
 - Creating course objects: O(1) per course  
 - Inserting into hash table: O(1) average case  
 - Searching: O(1) average case  
- Space Complexity: O(n) for storing courses  
  
Advantages  
- Fast insertion and search operations  
- Efficient for large datasets  
  
 Disadvantages  
- Potential for collisions  
- Requires more memory due to hashing overhead  
  
 Tree (Binary Search Tree)  
  
 Analysis  
- Time Complexity:  
 - Reading and parsing file: O(n)  
 - Creating course objects: O(1) per course  
 - Inserting into tree: O(log n) average case  
 - Searching: O(log n) average case  
 - Sorting: O(n) for in-order traversal  
- Space Complexity: O(n) for storing courses  
  
Advantages  
- Sorted order is maintained  
- Efficient for search, insert, and delete operations  
  
Disadvantages  
- More complex implementation  
- Performance degrades to O(n) in the worst case for unbalanced trees  
  
Recommendation  
  
Based on the analysis, the Hash Table is recommended for this program due to its average-case constant time complexity for insertions and searches, making it highly efficient for large datasets. Although it has potential collision issues, these can be mitigated with a good hash function. The memory overhead is justified by the performance gains in terms of speed.  
  
The Vector is suitable for smaller datasets and simpler implementations but falls short in terms of search efficiency. The Tree offers a good balance of sorted order and efficiency but requires careful implementation to avoid performance degradation.