**Colin Aheron Project One**

STRUCT Course

courseNumber: string

name: string

prerequisites: vector of string

END STRUCT

STRUCT TreeNode

course: Course

left: TreeNode

right: TreeNode

END STRUCT

// Global variables

dataStructureType: integer // 1 for vector, 2 for hash table, 3 for BST

vectorCourses: vector of Course

hashTableCourses: hash table of Course

bstRoot: TreeNode

// Utility Functions

FUNCTION splitLine(line: string, delimiter: char) RETURNS vector of string

tokens: vector of string

currentToken: string = ""

FOR EACH character IN line

IF character EQUALS delimiter THEN

IF currentToken IS NOT empty THEN

ADD currentToken TO tokens

currentToken = ""

END IF

ELSE

APPEND character TO currentToken

END IF

END FOR

IF currentToken IS NOT empty THEN

ADD currentToken TO tokens

END IF

RETURN tokens

END FUNCTION

// Menu Functions

FUNCTION displayMenu()

PRINT "1. Load data from file"

PRINT "2. Print all courses"

PRINT "3. Print course information"

PRINT "9. Exit"

PRINT "Enter your choice: "

END FUNCTION

FUNCTION mainMenu()

WHILE true

displayMenu()

READ choice

SWITCH choice

CASE 1:

PRINT "Enter filename: "

READ filename

PRINT "Choose data structure (1: Vector, 2: Hash Table, 3: BST): "

READ dataStructureType

loadCoursesFromFile(filename)

CASE 2:

printSortedCourseList()

CASE 3:

PRINT "Enter course number: "

READ courseNumber

printCourseInfo(courseNumber)

CASE 9:

PRINT "Exiting program"

RETURN

DEFAULT:

PRINT "Invalid option. Please try again."

END SWITCH

END WHILE

END FUNCTION

// File Reading and Data Loading

FUNCTION loadCoursesFromFile(filename: string)

OPEN file with filename

IF file fails to open THEN

PRINT "Error: Unable to open file"

RETURN

END IF

// Clear existing data

CLEAR vectorCourses

CLEAR hashTableCourses

bstRoot = NULL

FOR EACH line IN file

tokens = splitLine(line, ',')

IF tokens.size < 2 THEN

PRINT "Error: Invalid line format for course data"

CONTINUE to next line

END IF

newCourse = CREATE new Course

newCourse.courseNumber = tokens[0]

newCourse.name = tokens[1]

FOR i = 2 TO tokens.size - 1

ADD tokens[i] TO newCourse.prerequisites

END FOR

// Add to appropriate data structure

IF dataStructureType EQUALS 1 THEN

ADD newCourse TO vectorCourses

ELSE IF dataStructureType EQUALS 2 THEN

INSERT newCourse INTO hashTableCourses with key newCourse.courseNumber

ELSE IF dataStructureType EQUALS 3 THEN

bstRoot = insertCourseBST(bstRoot, newCourse)

END IF

END FOR

CLOSE file

PRINT "Data loaded successfully"

END FUNCTION

// Vector Implementation

FUNCTION sortCoursesVector(courses: vector of Course)

FOR i = 0 TO courses.size - 1

FOR j = 0 TO courses.size - i - 2

IF courses[j].courseNumber > courses[j+1].courseNumber THEN

SWAP courses[j] AND courses[j+1]

END IF

END FOR

END FOR

END FUNCTION

FUNCTION printSortedCourseList\_Vector()

sortCoursesVector(vectorCourses)

FOR EACH course IN vectorCourses

PRINT course.courseNumber + ": " + course.name

END FOR

END FUNCTION

FUNCTION printCourseInfo\_Vector(courseNumber: string)

FOR EACH course IN vectorCourses

IF course.courseNumber EQUALS courseNumber THEN

PRINT "Course Number: " + course.courseNumber

PRINT "Name: " + course.name

PRINT "Prerequisites:"

IF course.prerequisites IS empty THEN

PRINT " None"

ELSE

FOR EACH prereq IN course.prerequisites

PRINT " " + prereq

END FOR

END IF

RETURN

END IF

END FOR

PRINT "Course not found."

END FUNCTION

// Hash Table Implementation

FUNCTION printSortedCourseList\_HashTable()

courseNumbers: vector of string

FOR EACH entry IN hashTableCourses

ADD entry.key TO courseNumbers

END FOR

// Sort course numbers

FOR i = 0 TO courseNumbers.size - 1

FOR j = 0 TO courseNumbers.size - i - 2

IF courseNumbers[j] > courseNumbers[j+1] THEN

SWAP courseNumbers[j] AND courseNumbers[j+1]

END IF

END FOR

END FOR

FOR EACH number IN courseNumbers

course = hashTableCourses.get(number)

PRINT course.courseNumber + ": " + course.name

END FOR

END FUNCTION

FUNCTION printCourseInfo\_HashTable(courseNumber: string)

IF hashTableCourses.contains(courseNumber) THEN

course = hashTableCourses.get(courseNumber)

PRINT "Course Number: " + course.courseNumber

PRINT "Name: " + course.name

PRINT "Prerequisites:"

IF course.prerequisites IS empty THEN

PRINT " None"

ELSE

FOR EACH prereq IN course.prerequisites

PRINT " " + prereq

END FOR

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

// Binary Search Tree Implementation

FUNCTION insertCourseBST(root: TreeNode, course: Course) RETURNS TreeNode

IF root IS NULL THEN

RETURN new TreeNode(course)

END IF

IF course.courseNumber < root.course.courseNumber THEN

root.left = insertCourseBST(root.left, course)

ELSE IF course.courseNumber > root.course.courseNumber THEN

root.right = insertCourseBST(root.right, course)

END IF

RETURN root

END FUNCTION

FUNCTION printSortedCourseList\_BST(node: TreeNode)

IF node IS NOT NULL THEN

printSortedCourseList\_BST(node.left)

PRINT node.course.courseNumber + ": " + node.course.name

printSortedCourseList\_BST(node.right)

END IF

END FUNCTION

FUNCTION printCourseInfo\_BST(node: TreeNode, courseNumber: string)

IF node IS NULL THEN

PRINT "Course not found."

RETURN

END IF

IF courseNumber EQUALS node.course.courseNumber THEN

PRINT "Course Number: " + node.course.courseNumber

PRINT "Name: " + node.course.name

PRINT "Prerequisites:"

IF node.course.prerequisites IS empty THEN

PRINT " None"

ELSE

FOR EACH prereq IN node.course.prerequisites

PRINT " " + prereq

END FOR

END IF

ELSE IF courseNumber < node.course.courseNumber THEN

printCourseInfo\_BST(node.left, courseNumber)

ELSE

printCourseInfo\_BST(node.right, courseNumber)

END IF

END FUNCTION

// Wrapper functions to call appropriate implementation based on dataStructureType

FUNCTION printSortedCourseList()

IF dataStructureType EQUALS 1 THEN

printSortedCourseList\_Vector()

ELSE IF dataStructureType EQUALS 2 THEN

printSortedCourseList\_HashTable()

ELSE IF dataStructureType EQUALS 3 THEN

printSortedCourseList\_BST(bstRoot)

END IF

END FUNCTION

FUNCTION printCourseInfo(courseNumber: string)

IF dataStructureType EQUALS 1 THEN

printCourseInfo\_Vector(courseNumber)

ELSE IF dataStructureType EQUALS 2 THEN

printCourseInfo\_HashTable(courseNumber)

ELSE IF dataStructureType EQUALS 3 THEN

printCourseInfo\_BST(bstRoot, courseNumber)

END IF

END FUNCTION

**Evaluation**

**for vector:**

FUNCTION loadCoursesFromFile(filename: string)

OPEN file with filename // Cost: 1

IF file fails to open THEN // Cost: 1

PRINT "Error: Unable to open file" // Cost: 1 (if file fails to open)

RETURN // Cost: 1 (if file fails to open)

END IF

CLEAR vectorCourses // Cost: 1

FOR EACH line IN file // Executes n times

tokens = splitLine(line, ',') // Cost: O(k)

IF tokens.size < 2 THEN // Cost: 1

PRINT "Error: Invalid line format for course data" // Cost: 1 (if invalid line)

CONTINUE to next line // Cost: 1 (if invalid line)

END IF

newCourse = CREATE new Course // Cost: 1

newCourse.courseNumber = tokens[0] // Cost: 1

newCourse.name = tokens[1] // Cost: 1

FOR i = 2 TO tokens.size - 1 // Executes m times

ADD tokens[i] TO newCourse.prerequisites // Cost: 1

END FOR

ADD newCourse TO vectorCourses // Cost: 1 (amortized)

END FOR

CLOSE file // Cost: 1

PRINT "Data loaded successfully" // Cost: 1

END FUNCTION

Worst-case time complexity for Vector: O(n \* (k + m))

**Hash Table:**

The analysis is the same as the vector implementation, with one difference:

INSERT newCourse INTO hashTableCourses with key newCourse.courseNumber // Cost: O(1) average case, O(n) worst case

Worst-case time complexity for Hash Table: O(n^2 + n \* (k + m))

**Binary Search Tree:**

The analysis is the same as the vector implementation, with one difference:

bstRoot = insertCourseBST(bstRoot, newCourse) // Cost: O(log n) average case, O(n) worst case

Worst-case time complexity for BST: O(n^2 + n \* (k + m))

**Analysis**

|  |  |
| --- | --- |
| **Data Structure** | **Worst-Case Time Complexity** |
| Vector | O(n \* (k + m)) |
| Hash Table | O(n^2 + n \* (k + m)) |
| Binary Search Tree | O(n^2 + n \* (k + m)) |

n = number of courses

k = maximum length of any line in the file

m = maximum number of prerequisites for any course

For all three data structures - vector, hash table, and binary search tree - the process of reading the file and parsing each line is essentially the same. The program reads each of the n lines, splits it into tokens (taking time proportional to the line length k), and then processes the course information, including its prerequisites (taking time proportional to m). This common process contributes O(n \* (k + m)) to the runtime for all three structures.

The key difference comes in how each structure stores the newly created course object. For the vector, adding a new element is typically a constant-time operation, not significantly affecting the overall runtime. However, for the hash table and binary search tree, the worst-case scenario for inserting a new element can take time proportional to the number of courses already stored (n).

**Recommendation:**

The hash table and BST have the same worst-case time complexity due to potential collisions in the hash table and an unbalanced tree in the BST. However, their average-case performance would be better, with the hash table typically being O(n \* (k + m)) and a balanced BST being O(n \* (k + m + log n)).

The vector implementation has the best worst-case time complexity for this specific operation of reading the file and creating course objects.