Project One Documentation

Darren Sheftic

Southern New Hampshire University

CS300: DSA – Analysis and Design

**Vector Pseudocode**

Vector<Course> courses = []

loadDataIntoVector(String filePath)

file = OPEN("CourseInformation.txt")

IF file is NOT null

FOR each line in file

items = SPLIT(line, ",")

IF length(items) >= 2

course = Course()

course.courseNumber = items[0]

course.title = items[1]

IF length(items) > 2

FOR i = 2 to length(items)

prerequisite = items[i]

course.prerequisites.add(prerequisite)

courses.add(course)

ELSE

PRINT("File format error: Not enough parameters on line ", currentLine)

END FOR

CLOSE(file)

ELSE

PRINT("Error: Unable to open the file")

END IF

printCourseInformation(Vector<Course> courses, String courseNumber)

found IS FALSE

FOR each course in courses

IF course.courseNumber IS courseNumber

PRINT("Course Number: ", course.courseNumber)

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

IF NOT course.prerequisites.isEmpty()

PRINT("Prerequisites:")

FOR each prerequisite in course.prerequisites

PRINT(" - ", prerequisite)

END FOR

ELSE

PRINT("No prerequisites")

END IF

found IS TRUE

RETURN

END IF

END FOR

PRINT("Course not found: ", courseNumber)

numPrerequisiteCourses(Vector<Course> courses, Course c)

totalPrerequisites = 0

FOR each course in courses

IF course.courseNumber IS c.courseNumber

FOR each prerequisite in course.prerequisites

totalPrerequisites = totalPrerequisites + 1

END FOR

BREAK

END IF

END FOR

PRINT("Number of prerequisites for ", c.courseNumber, ": ", totalPrerequisites)

**Hash Table Pseudocode**

Hashtable<String, Course> coursesHashTable = new Hashtable<String, Course>()

loadDataIntoHashTable(String filePath)

file = OPEN(“CourseInformation.txt”)

IF file IS NOT null

FOR each line in file

items = SPLIT(line, ",")

IF LENGTH(items) >= 2

course = Course()

course.courseNumber = items[0]

course.title = items[1]

course.prerequisites = [] // Initialize prerequisites list

IF LENGTH(items) > 2

FOR i = 2 to LENGTH(items)

prerequisite = items[i]

course.prerequisites.add(prerequisite)

coursesHashTable.insert(course.courseNumber, course)

ELSE

PRINT("File format error: Not enough parameters on line ", currentLine)

END FOR

CLOSE(file)

ELSE

PRINT("Error: Unable to open the file")

END IF

numPrerequisiteCourses(Hashtable<Course> courses, String courseNumber)

course = courses.get(courseNumber)

IF course IS null

PRINT("Course not found: ", courseNumber)

RETURN

ELSE

RETURN LENGTH(course.prerequisites)

printCourseInformation(Hashtable<Course> courses, String courseNumber)

course = courses.get(courseNumber)

IF course IS null

PRINT("Course not found: ", courseNumber)

ELSE

// Print course information

PRINT("Course Number: ", course.courseNumber)

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

PRINT("Prerequisites: ", course.prerequisites.join(", "))

**Tree Pseudocode**

Tree<String, Course> coursesTree = new Tree<String, Course>()

loadDataIntoTree(String filePath)

file = OPEN(“CourseInformation.txt”)

IF file IS NOT null

FOR each line in file

items = SPLIT(line, ",")

IF LENGTH(items) >= 2

course = Course()

course.courseNumber = items[0]

course.title = items[1]

course.prerequisites = [] // Initialize prerequisites list

IF LENGTH(items) > 2

FOR i = 2 to LENGTH(items)

prerequisite = items[i]

course.prerequisites.add(prerequisite)

coursesTree.insert(course.courseNumber, course)

ELSE

PRINT("File format error: Not enough parameters on line ", currentLine)

END FOR

CLOSE(file)

ELSE

PRINT("Error: Unable to open the file")

END IF

numPrerequisiteCourses(Tree<Course> courses, String courseNumber)

course = courses.get(courseNumber)

IF course IS null

PRINT("Course not found: ", courseNumber)

RETURN

ELSE

RETURN LENGTH(course.prerequisites)

printCourseInformation(Tree<Course> courses, String courseNumber)

course = courses.get(courseNumber)

IF course IS null

PRINT("Course not found: ", courseNumber)

ELSE

PRINT("Course Number: ", course.courseNumber)

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

PRINT("Prerequisites: ", course.prerequisites.join(", "))

**Main Pseudocode**

printSorted(Vector<Course> courses)

n = LENGTH(courses)

quickSort(courses, 0, n-1)

// Print the sorted list of courses

FOR each course in courses

PRINT("Course Number: ", course.courseNumber, ", Course Title: ", course.title)

END FOR

quickSort(courses, low, high)

IF low < high

// Partition the array and get the pivot index

pivotIndex = partition(courses, low, high)

// Recursively sort elements before and after the pivot

quickSort(courses, low, pivotIndex - 1)

quickSort(courses, pivotIndex + 1, high)

partition(courses, low, high)

pivot = courses[high]

i = low - 1

FOR j = low to high - 1

IF compareCourses(courses[j], pivot) < 0

i = i + 1

SWAP(courses[i], courses[j])

END IF

END FOR

SWAP(courses[i + 1], courses[high])

RETURN i + 1

compareCourses(course1, course2)

// Compare course numbers in alphanumeric order

IF course1.courseNumber < course2.courseNumber

RETURN -1

ELSE IF course1.courseNumber > course2.courseNumber

RETURN 1

ELSE

RETURN 0

END IF

SWAP(course1, course2)

// Swap two courses

temp = course1

course1 = course2

course2 = temp

MAIN function

PRINT menu()

WHILE choice() IS NOT 4

PRINT("Menu:")

PRINT("1. Load Data Structure")

PRINT("2. Print Course List")

PRINT("3. Print Course Information")

PRINT("4. Exit")

choice = INPUT

SWITCH choice

CASE 1:

loadDataIntoVector()

loadDataIntoHashTable()

loadDataIntoTree()

BREAK

CASE 2

printSorted(Courses)

CASE 3

printCourseInformation(courseNumber)

CASE 4

RETURN

END SWITCH

END WHILE

END MAIN

TERMINATE

**Evaluation**

* Vector:
  + Advantages: Simple, easy to implement, and offers O(n) runtime for most operations.
  + Disadvantages: Slower for searching and may not be efficient for large datasets or frequent search operations.
* Hash Table:
  + Advantages: Provides O(1) (constant time) access for most operations, efficient for searching.
  + Disadvantages: More complex to implement than a vector, potential collisions affecting performance.
* Tree:
  + Advantages: Provides efficient searching (O(log n)), maintains data in a structured way.
  + Disadvantages: Slightly more complex to implement than a vector, potentially slower insertions and deletions than a hash table.

Based on the analysis of runtime complexity (Big O) and considering the requirements of efficient searching and reasonable memory usage, the recommended data structure is the **Hash Table**. To see the analysis look to the next page.

**Analysis**

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each line in file** | 1 | n | n |
| **Splitting items** | 1 | n | n |
| **Creating a Course object** | 1 | n | n |
| **Adding Course to vector** | 1 | n | n |
| **Print course information** | 1 | n | n |
| **Total Cost** | | | 5n |
| **Runtime** | | | O(n) |

**Hash Table**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For each line in file** | 1 | n | n |
| **Splitting items** | 1 | n | n |
| **Creating a Course object** | 1 | n | n |
| **Adding Course to Hash Table** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n+5 |
| **Runtime** | | | O(1) |

**Binary Search Tree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | log n |
| **Splitting items** | 1 | n | 1 |
| **Creating Course object** | 1 | n | 1 |
| **Inserting Course to Tree** | Log n | n | 1 |
| **print the prerequisite course information** | 1 | n | 1 |
| **Total Cost** | | | log n + 3 |
| **Runtime** | | | O(log n) |