CS 300 Project One – Subset – Data Structures and Runtime Analysis

Chris Marrs

10/21/2023

**Project Goals:**

* Designs will support three data structures, vector, hash table, and tree.
* Evaluate the run-time and memory of data structures that could be used to address the requirements.

Class Course

courseNumber: int

courseTitle: string

prerequisites: vector<string>

Course (courseNumber, courseTitle, prerequisites) // constructor()

this.courseNumber = courseNumber

this.courseTitle = courseTitle

this.prerequisites = prerequisites

//Course

//Class

validateCourseFile(fileName: string):

open fileName for reading as file // cost = 1

if file does not open: // cost = 1

display "Error opening the file"

exit

courseList = empty list // to store all courses // cost = 1

prereqList = empty list // to store all prerequisites // cost = 1

for each line in file: // cost = n

split the line by comma into parts // cost = n

if the length of parts is less than 2: // cost = n

display "Error: Insufficient data on line: ", line

exit

courseNumber = parts[0].trim() // cost = n

courseTitle = parts[1].trim() // cost = n

add courseNumber to courseList // cost = n (add is linear)

prerequisites = empty vector<string> // cost = n

if length of parts > 2: // cost = n

prerequisites = parts[2..end].trim() // cost = n

for each prerequisite in prerequisites: // cost = n

add prerequisite to prereqList // cost = 2n (worst case)

//for // cost = n

//for // cost = n

close file // cost = 1

// check if every prerequisite exists in courseList

for each prerequisite in prereqList: // cost = n

if prerequisite not in courseList: // cost = n

display "Error: Prerequisite", prerequisite, "does not exist as a course."

exit

//for // cost = n

display "File validation successful!" // cost = 1

//validateCourseFile () // total cost = 17n + 6 = O(n)

// Vector based methods

processCourseFileVector(fileName: string):

open fileName for reading as file // cost = 1

if file does not open

error(“error opening the file",filename)

exit

coursesVector = new empty vector // cost = 1

for each line in file // cost = n

split the line by comma into fields // cost = n

courseNumber = fields[0].trim() // cost = n

courseTitle = fields[1].trim() // cost = n

prerequisites = empty vector<string> // cost = n

if length of fields > 2 // cost = n

prerequisites = fields[2..end].trim() // cost = n

else

prerequisites = []

courseObj = new Course object(courseNumber, courseTitle, prerequisites[0..2])

// cost = n

add courseObj to coursesVector // cost = n (add is linear)

//for // cost = n

close file // cost = 1

return coursesVector // cost = 1

// processCourseFile() // total cost = 10n + 4 = O(n)

// Hash Table based methods

processCourseFileHashTable (fileName: string):

open fileName for reading as file // cost = 1

if file does not open // cost = 1

error(“error opening the file",filename)

exit

coursesHashTable = empty dictionary // cost = 1

prerequisites \_info = empty list // cost = 1

for each line in file // cost = n

split the line by comma into fields // cost = n

courseNumber = fields[0].trim() // cost = n

courseTitle = fields[1].trim() // cost = n

if length of fields > 2 // cost = n

prerequisites = fields[2..end].trim() // cost = n

else

prerequisites = []

courseObj = new Course object(courseNumber, courseTitle, prerequisites)

// cost = n\*O(log(n)) = n log(n)

// (assuming worst case)

add courseNumber, courseObj to coursesHashtable // cost = n

//for // cost = n

close file // cost = 1

return coursesHashTable // cost = 1

// processCourseFileHashtable()

// total cost= n log(n) + 9n + 6 = O(n log (n))

// Binary Search Tree based methods

Class TreeNode:

course: Course

children: list of TreeNode

Constructor(course: Course):

Set this.course = course

Set this.children = []

End Class

addNode(parentNode: TreeNode, newNode: TreeNode):

add newNode to parentNode.children // cost = O(n)

//addNode()

processCourseFileTree(fileName: string):

open fileName for reading as file // cost = 1

if file does not open: // cost = 1

display "Error opening the file"

exit

rootNode = new TreeNode(new Course(0, "Root", [])) // cost = 1

for each line in file: // cost = n

split the line by comma into fields // cost = n

courseNumber = fields[0].trim() // cost = n

courseTitle = fields[1].trim() // cost = n

If length of parts > 2: // cost = n

prerequisites = fields[2..end].trim() // cost = n

else

prerequisites = []

courseObj = new Course(courseNumber, courseTitle, prerequisites)

// cost = n

courseNode = new TreeNode(courseObj) // cost = n

addNode(rootNode, courseNode) // cost = n\*O(n) = n2

// (assuming worst case)

//for // cost = n

close file // cost = 1

return rootNode // cost = 1

//processCourseFileTree() // total cost = n2+9n+5 = O(n2)

**Data Structure Evaluation**

The goal of this evaluation is to compare and contrast the advantages and disadvantages of the three data structures that were part of the evaluation, vector, hash table, and binary search tree.

The functional requirements are the basis of the evaluation:

* Print a list of all the Computer Science courses in alphanumeric order.
* For a given course, print out its title and prerequisites.

The criterion for the evaluation is ease of development, maintainability, and performance complexity.

*Ease of development*

Both the vector data structure and the hash table data structure are straightforward and relatively easy to implement for adding data to the data structure as well as accessing the data afterwards. The BST data structure requires more code for node creation initially and addition to the structure.

*Maintainability*

The vector and hash table data structures have a simpler code implementation and would be easier to maintain and enhance. The BST data structure is more complicated and would be more difficult to maintain and enhance.

*Performance Complexity*

The vector data structure has an O(N) value for the verification and loading of the data. With the addition of the sorted list requirement, there is O(N2) value.

Implementing the hash table via a dictionary has an O(N log(N)) value and provides inherent ascending data sorting. The requirement for an ascending sorted list would have no additional impact based on n-items in the file.

The binary search tree data structure has an O(N2) value for the requirements as listed.

**Recommendation**

The key-value aspect of the data lends itself to a hash table data structure from the start. Adding in the complexity analysis and the overall comparison of all three data structures, I would recommend the hash table data structure for use in this project. The O(N log(N)) value for the hash table data structure is consistently better than the O(N2) value of the other two. The code development is straight forward as is the maintenance of the code. The key-value aspect is easy to implement and use.