# Levi Winters

# Professor Ling

# CS 300

# 11/30/2023

# **Project One**

// Vector pseudocode

function numPrerequisiteCourses(courses as Vector, c as Course)

totalPrerequisites ← getPrerequisites(c)

for each p in totalPrerequisites

totalPrerequisites ← totalPrerequisites + getPrerequisites(p)

print length(totalPrerequisites)

end function

function printSampleSchedule(courses as Vector)

for each course in courses

print getName(course)

for each pre in getPrerequisites(course)

print getName(pre)

end function

function printCourseInformation(courses as Vector, courseNumber)

for each course in courses

if getNumber(course) = courseNumber

print getInfo(course)

for each pre in getPrerequisites(course)

print getInfo(pre)

end function

function openFiles(fileName, courses as Vector)

try

openFile fileName

while not endOfFile(fileName)

line ← readLine(fileName)

tokens ← split(line)

if invalid(tokens)

print "Invalid format"

continue

course ← Course()

setNumber(course, tokens[0])

setName(course, tokens[1])

setPrerequisites(course, tokens[2])

add(courses, course)

closeFile(fileName)

catch FileNotFoundException

print "File not found"

catch IOException

print "IO Exception occurred"

end function

// Hashtable pseudocode

function numPrerequisiteCourses(courses as Hashtable)

totalPrerequisites ← get(courses, c)

count ← 0

for each p in totalPrerequisites

totalPrerequisites ← totalPrerequisites + get(courses, p)

count ← count + 1

print count

end function

function printSampleSchedule(courses as Hashtable)

for each course in keys(courses)

print getName(course)

if hasPrerequisites(courses, course)

for each pre in get(courses, course)

print getName(pre)

end function

function printCourseInformation(courses as Hashtable, courseNumber)

for each course in keys(courses)

if getNumber(course) = courseNumber

print getInfo(course)

for each pre in get(courses, course)

print getInfo(pre)

end function

function openFiles(fileName, courses as Hashtable)

try

openFile fileName

while not endOfFile(fileName)

line ← readLine(fileName)

tokens ← split(line)

if invalid(tokens)

print "Invalid format"

continue

course ← Course()

setNumber(course, tokens[0])

setName(course, tokens[1])

setPrerequisites(course, tokens[2])

put(courses, tokens[0], course)

closeFile(fileName)

catch FileNotFoundException

print "File not found"

catch IOException

print "IO Exception occurred"

end function

// Tree pseudocode

function numPrerequisiteCourses(courses as Tree, c as Node)

totalPrerequisites ← empty set

add(totalPrerequisites, getLeft(c))

add(totalPrerequisites, getRight(c))

count ← 0

for each p in totalPrerequisites

add(totalPrerequisites, getLeft(p))

add(totalPrerequisites, getRight(p))

count ← count + 1

print count

end function

function printSampleSchedule(courses as Tree)

for each node in courses

print getName(node)

if hasLeft(node)

print getName(getLeft(node))

if hasRight(node)

print getName(getRight(node))

end function

function printCourseInformation(courses as Tree, courseNumber, c as Node)

for each node in courses

if getNumber(node) = courseNumber

print getInfo(node)

if hasLeft(c)

print getInfo(getLeft(c))

if hasRight(c)

print getInfo(getRight(c))

end function

function openFiles(fileName, courses as Tree)

try

openFile fileName

while not endOfFile(fileName)

line ← readLine(fileName)

tokens ← split(line)

if invalidTokens(tokens)

print "Invalid format"

continue

course ← Course()

setNumber(course, tokens[0])

setName(course, tokens[1])

setPrerequisites(course, tokens[2])

add(courses, course)

closeFile(fileName)

catch FileNotFoundException

print "File not found"

catch IOException

print "IO Exception occurred"

end function

// Global courses data structure

courses = initialize empty

while true

// Display menu for user to select option

printMenu()

option = getUserOption()

if option == 1:

// Load data

fileName = getFilenameFromUser()

courses = loadCourses(fileName)

else if option == 2:

// Print courses

printCourses(courses)

else if option == 3:

// Print course details

courseNumber = getAndValidateCourseNumber()

printCourse(courses, courseNumber)

else if option == 4:

// Exit program

break

// Load course data from file

function loadCourses(filename)

courses = create empty data structure

try

open file filename

while not end of file

line = read next line

course = parseAndCreateCourse(line)

add course to courses

close file

catch file errors

print error message

return courses

// Parse line from file into Course object

function parseAndCreateCourse(line)

tokens = splitLine(line)

if invalidTokens(tokens)

return null

course = createCourse()

populateCourse(course, tokens)

return course

// Print courses in alphabetical order

function printCourses(courses)

sortedCourses = sortCoursesAlphabetically(courses)

for each course in sortedCourses

print course.name

## **Runtime Analysis**

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | *O*(1) |
| **Read line** | 1 | n | *O*(*n*) |
| **Split** | 1 | n | *O*(*n*) |
| **Add** | 1 | n | *O*(*n*) |
| **Close file** | 1 | 1 | *O*(1) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Hashtable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Open file** | 1 | 1 | *O*(1) |
| **Read line** | 1 | n | *O*(*n*) |
| **Split** | 1 | n | *O*(*n*) |
| **Put** | 1 | n | *O*(*n*) |
| **Get** | 1 | n | *O*(*n*) |
| **Close file** | 1 | 1 | *O*(1) |
| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Open file** | 1 | 1 | *O*(1) |
| **Read line** | 1 | n | *O*(*n*) |
| **Split** | 1 | n | *O*(*n*) |
| **Add** | log(*n*) | n | *O*(nlog(*n*)) |
| **Get Left/Right** | 1 | n | *O*(*n*) |
| **Close file** | 1 | 1 | *O*(1) |

The vector operations have a total runtime of O(n) for reading in the file and creating the course objects. This is because the primary costs are from the line-by-line read and per-line operations like split and add, each with a cost that scales linearly with the number of courses n. The open and close file costs are constant. So vector storage allows for simple linear-time processing.

Similarly to the vector, the total hashtable runtime is O(n) for file reading and object creation. The read, split and close file costs are the same. The put and get costs are higher than vector as they are hash computations, but their total cost over n courses remains linear so the overall runtime is still O(n). The hashtable provides fast key-based lookup in linear time.

The tree has a higher overall cost of O(nlog(n)) due to the insert operation being O(log(n)) per node inserted in a balanced tree. All other operations like read, split, get left/right scale linearly with the number of courses n. But the logarithmic insertion leads to a higher total cost. The benefit of the tree is efficient searching and ordering of the hierarchical data.

Now that all three data structures have been analyzed, I recommend using a hashtable as the preferred data structure for my code. Based on the Big O analysis results, the decision is challenging, as both the vector and hash table exhibit a runtime complexity of O(n).