**Module Six Project One**

William Neal

SNHU

CS-300, DSA Analysis & Design

Professor Susalla

8/1/23

Scenario

The academic advisors in the Computer Science department at ABCU are very happy with the pseudocode you completed. You are now prepared to move forward with expanding the pseudocode to directly respond to the two items advising hopes to accomplish with this program. Remember, your program will need to do the following:

Print a list of all the Computer Science courses in alphanumeric order.

For a given course, print out its title and prerequisites.

You will be writing pseudocode to address each of advising’s requirements, and you will do so for each of the data structures you already started to explore in the previous assignments (vector, hash table, and tree). Then you will perform a runtime analysis to determine which data structure will be the best to use when you begin coding in the next project.

Directions

In previous milestones, you wrote a description for the Course object that will be stored in different data structures. To be able to determine the running time of each of those data structures in this application, you will need to finish writing all the pseudocode for the rest of the code and perform a Big O analysis.

Pseudocode

Resubmit pseudocode from previous pseudocode assignments and update as necessary. In the previous assignments, you created pseudocode for each of the three data structures (vector, hash table, and tree). Be sure to resubmit the following pseudocode for each data structure.

Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.

Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.

Design pseudocode that will print out course information and prerequisites.

Create pseudocode for a menu. The menu will need to perform the following:

Load Data Structure: Load the file data into the data structure. Note that before you can print the course information or the sorted list of courses, you must load the data into the data structure.

Print Course List: This will print an alphanumerically ordered list of all the courses in the Computer Science department.

Print Course: This will print the course title and the prerequisites for any individual course.

Exit: This will exit you out of the program.

Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order. Continue working with the Pseudocode Document linked in the Supporting Materials section. Note that you will be designing for the same three data structures that you have been using in your previous pseudocode milestones (vector, hash table, and tree). This time you will create the final pieces of pseudocode that you will need for ABCU’s advising program. To complete this part of the process, do the following:

Sort the course information by alphanumeric course number from lowest to highest.

Print the sorted list to a display.

Evaluation

Evaluate the run-time and memory of data structures that could be used to address the requirements. In a previous assignment, you created pseudocode to do the following:

Define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.

Show how to create course objects, so that one course object holds data from a single line from the input file.

Using this pseudocode written for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This should not include the pseudocode written for the menu or the sample schedule above. To do this, do the following:

Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure.

Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.

Based on the advisor’s requirements, analyze each data structure (vector, hash table, and tree). Explain the advantages and disadvantages of each structure in your evaluation.

Now that you have analyzed all three data structures, make a recommendation for which data structure you will plan to use in your code. Provide justification for your recommendation, based on the Big O analysis results and your analysis of the three data structures.

So this is the pseudocode I wrote for vectos:

I want to define a class "Course"

I should have "course#", "courseTitle", and "preReqs"

Then I’m going to define a method in class "Course" for printing that information above

I am going to call it “printCourseInfo”

Print "courseNumber", "courseTitle"

Throw an if statement in there for "preReqs" is not empty/null then print it!

Initialize an empty list and call it "courses" (so original I know)

Next I want to define function "parseDaLines" which will need to take in strings

I want to split on the "," and throw that into a different list which I am going to call "paramS"

But if the length of "paramS" is < 2 I want to throw an error "Bro get it together you know that aren’t right, That boy aint right" Then I need an else statement.

I need a new object "course" of the class "Course"

I need to set "course.course#" to the 1st item in "paramS"

I need to set "course.courseTitle" to the 2nd item in "paramS"

Next I need an if statement which should be like: the length of "paramS" is >2

Then set "course.preReqs" to the rest of the items in "paramS"

I will need an Else for setting "course.preReqs" to an empty list

I want to return that object "course" I need that for later

Now I’m going to make a new function called "validatePreReqs" that takes in my list "courses"

Like you said I am going to throw in a for each loop and loop through the "course" in "courses"

Then another for each loop for "preReqs" in "course.preReqs"

Now I am going to say if “preReqs" not in the course # of "courses"

Error Will Robinson! "That isn’t a preReq. preReqs are courses my friend"

Now we gotta open the file with the course data

Now more fun, another for each loop which will loop through each line in the file

Parse the line using " parseDaLines" + the resulting course data to "courses"

Validate my spaghetti code with "validatePreReqs"

Close the file, hopefully to never be opened again

Well now I need something to search with so make a new function call it "searchCourse" but remember it’s gotta take a course #

FOR EACH !!! "course" in "courses" Lets say if "course.course#" is == to the inputed course #

Then its time to print out that "course.printCourseInfo"

Return, return!

Otherwise print "You got a typo, or it aint a real class, better luck next time mate"

THIS IS THE VECTOR ALPHANMUMRIC:

Again IF courses is empty or == null throw and error and print “load your data” return

ELSE sort “courses” in ascending order by course# then print “your sorting to perfection:” The actual sorting function should be FOR EACH! “course” in “course” print “course numba: “ + course.course# and “course title” + course.courseTitle.

Now this is the pseudocode I wrote for hash table:

Alright let’s start off by defining and initializing a hash table our key can be the course # and value is Course object which will hold title and preReqs

Next lets open the file, our hash table has been initialized so our data can be stored

Lets read the file, trim space, PARSE! Should be parasing on comma, and of course if-else statement for any nonsense like less than 2 params or grab course # and title and if for preReqs

Next I am I am going to for/for each loop through the courses. I am going to check if that course # == something in the hash table. If statement, if it does throw an error. Else if add a new entry to the hash table with course # (again as a key, remember our first step), and add the new data to Course object

Okay now we have to loop through the hash table, we need to look at the Course values/data. Find the preReqs and if statement Prereqs course # doesn’t exist in the hash table ERROR ERROR!

If statement, if no errors found then time to Print like fedex kinoks, we need to print all data, so all the values in Course. Course #, title, and preReqs.

This feels like a punishment. Theres no order in a hash table, so I have to extract everything sort it and display it. Let’s try:

IF course hashTable is empty || == null, then print “load the hash table man”. Return. I am going to make a list and call it theOrderOfTheHashTableListOfCoursesInAlphanumericOrder. Let’s FOR EACH! Okay so for each course# in our course hashTable add course# to our list theOrderOfTheHashTableListOfCoursesInAlphanumericOrder. Now sorting theOrderOfTheHashTableListOfCoursesInAlphanumericOrder in alphanumeric order, FOR EACH! Course# in theOrderOfTheHashTableListOfCoursesInAlphanumericOrder list grab the course from our ol fellow the hashTable using course# as the key, then print “ In the Order of Courses in Alphanumeric Order of the Higher-Order:” + course# + courseTitle.

And finally, I have the revised pseudocode for trees, I had used the format from before when I saw in the guidelines “You should store each course object into the vector data structure. Once the entire file has been processed, the vector data structure will have multiple course objects, one per line in the file.” Which was under section #2, I should have clarified with you. I saw I lost points for this, so here is the revision.

Let’s start off by creating a class nodesOnNodes with course#, courseTitle, preReqs, leftChild and rightChild.

Then let’s create a class for defining the following: insert (implying a node), inOrderTraverse (also implying a node) and findaNode (implying a course#).

Next let’s initialize our Binary Tree and then open our file

Let’s throw in an IF statement, saying that if the file is not open throw an error.

Now let’s have a while loop say while not the end of the file, read a line, and parse on our comma and into our course#, courseTitle, and preReqs. As well as create a new “node” object with our parsed data and we want to call insert.node

Coolieo, now let’s create a function for insertion. Okay say IF the root is null set it to node. And IF our nodes course# is less than our roots course# lets insert. ELSE IF call our insert on roots rightChild.

Alright time to traverse. Now say IF node is == null, call our inOrderTraverse on leftChild. Then we need to print the nodes course#, courseTitle, and preReqs. Now call inOrderTraverse on the rightChild.

Lastly, we need to create something to find everything. So, lets say IF root == null || (or) course# == course# then return our root. Now IF course# < roots course# then call findaNode on our roots leftChild. And very lastly ELSE call findaNode on our roots rightChild.

Alright onto a menu:

Now go with binary tree because its fresh in my mind. Let’s make a binary tree object call it “trees”.

Next make a function call it menu, inside WHILE true we are going to print:

“menu”

“1. Load the Data”

“2. Print the Course List”

“3. Print Course”

“4. Exit”

Now we have to take in input from the user, call it thePriceIsRight

IF thePriceIsRight == 1 then loadData

ELSE IF thePriceIsRight == 2 then printCourseList

ELSE IF thePriceIsRight == 3 then printCourse

ELSE IF thePriceIsRight == 4 break loop

ELSE print “that’s invalid, there are four choices, one, two, three, four. This isn’t complex. Sesame street has an excellent character to help you with this Count von Count, look him up he should be able to help you with this.”

Alright now we need to loadData

So open the file and if it doesn’t open print an error message

FOR EACH line in the file, parse line into our course#, courseTitle, and preReqs

Create a new node object again call it node, and call trees.insert

Close our file, print “the file loaded, make another selection”

Alright next we need to printCourseList

IF the tree is null/empty the print “load the data first, its choice 1/one ah ah ah”

Then return

Lastly, we need printCourse, which is a little more tricky. But it starts out same as before, IF tree is null/empty then print “load the data”. And return. IF the tree is viable then input course#. We should call findaNode(course#) on our tree. IF its found then print that data ELSE print “it wasn’t found try again.. or don’t.”

Alright part 3, alphanumeric order

I am going to do the binary tree here and then mark in red for the vector and hash table under each of their respective pseudocodes.

So the binary tree I really want to traverse the tree in order so it should look like this

IF tree is empty, throw error and print “load the data so we can sort it to your specific liking”

ELSE print “course in alphanumeric broski:” then call function alphaNumericTime.

alphaNumericTime takes a param of node. Lets say IF node is != null then call our inOrderTraverse(findaNode.left) and print findaNode.course# and findaNode.CourseTitle. lastly we want to call inOrderTranverse(findaNode.right).

Take a look at Evaluation now.

Okay so the question is about big O notation and run times. I am going to start with the following:

Opening the file should be O(1)

FOR EACH! Line in the file O(n)

Parsing the line O(1)

Total would be O(n)?

Okay now let’s look at my favourite which is vectors! I think at the end of the day this should be the best choice, but I am going to work through everything and see if my assumption is correct.

So storing each course would be O(1)

Retrieving any course would be O(n)

And sorting which is a pain would be O(n log n)?

I think my disadvantages would be O(n) time if search index isn’t known meaning you’d have to traverse the whole vector.

But my advantages here would be constant O(1) if the index is known. Also course lists change, so resizing would be great on a vector.

Alright onto hash tables,

Storing each course into the hash table would also be O(1) I think unless something is wrong for a bigger file it could be O(n).

Retrieving a course would be O(1).

Yuck, not happy with this but it looks like hash tables are better at retrieving courses.

Disadvantages would be massive hash collisions, and of course our sorting problem. BUT you trade that for O(1) insert, search and delete. I guess we could use a hash table and then for sorting functionality put our course objects into a different reference object.

Finally, our binary tree,

So starting out inserting and storage would be O(log n)

Retrieving would be also O(log n)

And memory usage would also be O(n)

Huge memory usage is the big disadvantage here. Also, if anything in here degenerates you’ve got a big, big mess on your hands. However you can get insert, delete, and search on O(log n) because everything is maintained sorted it gets some efficiency points.

The last question is making a recommendation. This is hard, if we are saying all other things being equal, meaning the course list will not change. No additions, deletions, or updates then clearly our best choice is a hash table because of its speed. However, it’s sorting is terrible, and I know that’s functionality we want and have used. So, I think binary tree is my final answer given the information I have. I can insert, delete which will satisfy my neurotic anxiety filled mind of what-if’s while still having great search functionality. I really wish the best answer was a vector, they are just so easy to implement but alas the tech world is a “ruthless and unforgiving taskmaster”. May prod be functional and my stories on the board be easy. Should someone call me out in retrospective may the scrum master have mercy on their soul.