Matthew Oliphant

CS-300: DSA – Analysis and Design

Southern New Hampshire University

# August 7, 2021

# CS 300 Pseudocode Document

// Create course object

Struct Course

courseId holds each courses Id

courseName holds each courses name

preReq1 holds the current courses first pre-req.

preReq2 holds the current courses second pre-req.

Constructor for a course object

// Define how program opens file, parses & reads each line & validates

Menu 1 loads courses, sending function a csvPath and data structure

Create Parser object and give it the csvPath to open the file

Validate the data with Try:

For the number of lines in the CSV courses file, parse each line

Define a course and then read data from each line

Set course Id, Name, PreReq1, & PreReq2 (ex. File[i][0])

Send the course to be added/ inserted to data structure

Validate the data with Catch:

Display error

// Print out the course info and pre-reqs.

Get number of pre-reqs. - Int numPrerequisiteCourses

Add each pre-req. to data structure

printCourseInformation

For (each pre-req in the data structure)

Check that pre-req matches at least one course number

Print pre-req. information

// Create a menu

Menu option #1:

Begin timing the runtime

Load data into data structure (defined above)

Calculate & display the runtime

Menu option #2:

Begin timing the runtime

Sort the course info by alphanumeric course num (low to high)

Print sample schedule

Calculate & display the runtime

Menu option #3:

Begin timing the runtime

Search for course

Print course data if found or message if not

Calculate & display the runtime

Menu option #4:

Exit the while loop that is used to display the menu

Print final message

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

// Vector pseudocode

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

**totalPrerequisites = prerequisites of course c**

**for each prerequisite p in totalPrerequisites**

**add prerequisites of p to totalPrerequisites**

**return totalPrerequisites**

}

void printSampleSchedule(Vector<Course> courses) {

**for each course c in courses**

**print course information**

**// Need to know format for displaying info**

}

void printCourseInformation(Vector<Course> courses, String courseNumber) {

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

**totalPrerequisites = 0**

**for all courses**

**if the vector index is not empty**

**check if there is a next node**

**for each prerequisite of the course (next nodes)**

**add the prerequisite to totalPrerequsites**

**return totalPrerequisites**

}

void printSampleSchedule(Hashtable<Course> courses) {

**for each course c in courses**

**if node exists**

**print course information**

**check for next node**

**// Need to know format for displaying info**

**// Should this schedule show an example path to graduation?**

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

**for all courses**

**locate the vector index using courseNumber**

**if a course in this bucket is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

**totalPrerequisites = child nodes of course c**

**for each prerequisite p in totalPrerequisites**

**add childs of p to totalPrerequisites**

**return number of totalPrerequisites**

}

void printSampleSchedule(Tree<Course> courses) {

**for each course c in courses (parent node)**

**print course information (child node)**

**// Need to know how method should display info**

}

void printCourseInformation(Tree<Course> courses, String courseNumber) {

**for all parent nodes**

**if the course id is the same as courseNumber**

**print out the course information**

**for each prerequisite (child) of the course (parent)**

**print the prerequisite course information**

}

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

| **Code: Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define Vector** | 1 | 1 | 1 |
| **For rows in csv file** | 1 | n | n |
| **Define course** | 1 | 1 | 1 |
| **Read course info** | 4 | n | 4n |
| **Push back course** | 1 | n | n |
| **Total Cost** | | | 6n + 2 |
| **Runtime** | | | O(n) |

| **Code: Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define HashTable** | 1 | 1 | 1 |
| **For rows in csv file** | 1 | n | n |
| **Define course** | 1 | 1 | 1 |
| **Read course info** | 4 | n | 4n |
| **Create hash key** | 1 | n | n |
| **Node to the key index** | 1 | n | n |
| **If root, insert node** | 3 | 1 | 3 |
| **Else add root if empty** | 5 | n | 5n |
| **Else locate & insert** | 4 | n | 4n |
| **Total Cost** | | | 16n + 5 |
| **Runtime** | | | O(n) |

| **Code: Binary Search Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define BST** | 1 | 1 | 1 |
| **For rows in csv file** | 1 | n | n |
| **Define course** | 1 | 1 | 1 |
| **Read course info** | 4 | n | 4n |
| **If course is root, insert** | 1 | 1 | 1 |
| **If courseId is > curr node** | 1 | 1 | 1 |
| **Add node to correct left** | 3 | n | 3n |
| **Else add to correct right** | 4 | n | 4n |
| **Total Cost** | | | 12n + 4 |
| **Runtime** | | | O(n) |

**Pros and Cons**

*Vector*:

Pros – Fast to push back into a vector O(1); Consumes O(n) worst case space complexity; Most other actions are O(n) time complexities for the worst case

Cons – Have to move each value when inserting to middle of list O(n); resizing a vector usually means their will be empty/ wasted indexes

*Hash Table*:

Pros – Do not necessarily have to search the entire vector to find a number O(n); Can be implemented in multiple ways (Chaining, Linear Probing, Quadratic Probing, Direct Hashing)

Cons – Have to deal with collisions; Possible to have entire list in one vector index or a lot of values in a few indexes; May be expensive to store on heap; Resizing is same as vector

*Binary (Search) Tree*:

Pros – Using a BST vs a regular tree will sort each course in alphanumeric order O(n) and can be printed in-order; BST resizes each time a new node is added, but resizes as needed

Cons – It’s possible for BST to be unbalanced if it’s not randomly sorted (mostly sorted data is bad for BSTs); Plain trees may be hard to print in sorted order

Each of these data structures would provide a pretty similar time and space complexity O(n). For this project, I would recommend using whichever data structure the developer is most comfortable with, but I will more than likely use a tree. I think it’s very useful that the data can be sorted when inserted if a BST is used and from there the print algorithms are fairly straight forward (in-order traversal).