# CS 300 Pseudocode Document

## 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

Method for loading file and distributing to vector

File variable (in stream file name)

File lines vector

Prerequisite items vector

Course numbers vector

Course name vector

Lines container for split items

Open file

While not at end of file

If file open

get line, assigns data to file lines vector, 8 items

Close file

For loop to go through file lines vector

Split string at commas assign items to lines container

if split length less than 2

output error message for bad line

else

continue past line

first split item is course number

second split item is course name

n split items are course prerequisites

/\*

I think that is vaguely correct, might have to read the individual items until a comma, and then assign those to the course vectors until the line is done instead. What I have right now would make it hard to assign the prereq’s. But I think my plan would be to reference the locations of the items in the vector, because the first course number would have the same position as the first course name. I'll have to rethink how this will work for the prerequisite strings.

\*/

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

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

}

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

Class Hashtable

Privates

Structure of course

course variables

Course constructor

Course initialized constructor

Vector course courses to hold file information

Public methods

Print schedule

Print course info

Number of prereqs

Method to generate hashkey using course number

Unsigned key is set to course number modulo table size

Method for loading courses to hash table

variables

Course number

Course name

Prerequisites

Table size or Course int max (8 classes, double that value for open spots to limit collisions)

Open file

If file open

read file, get row total

For loop start at 0 go to row total

-populate hash table, get lines separating lines at commas

-get course information at whatever the positions are in the gotten lines and the integer of the row in the for loop -Insert course into hash table (use course id to hash keys)

//I will need to implement an if-else block that checks for the proper amount of items in each line from the file, there is no check in place to make sure information is correct

int numPrerequisiteCourses(Hashtable<Course> courses) {

Variable number of prerequisites is 0

Generate hashkey

Access node at hashkey in courses

return prerequisites for courses to integer

}

void printSampleSchedule(Hashtable<Course> courses) {

For loop going through hash table beginning to end

if node not empty

output courses course number, name, prereqs

}

//fix me, might need a sort function so that entries from file are in order

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

Get hashkey for course number

Access node at hashkey in courses

If course number is found

output course name and prerequisites

}

======================================================================

// Tree pseudocode

Throw catch error

Using error catch throw to validate file and make sure all lines have correct qualities

Load courses(course list file, and bst)

Open file

If file is open

read file

While file isn’t at the end

Try, For loop to go through lines of the file

insert get lines into the tree

Catch validating the files contents

Add/create course using variable node

If root node is null

root is new node

Else if root node is greater than node

left is new node

Else

right is new node

//fixme make a search method to go through tree

int numPrerequisiteCourses(Tree<Course> courses) {

Prerequisites counter

while prerequisite isn’t null

call a search method using prerequisite to find courses

If course prerequisite is equal to courses

Call numprerequisitecourses using the current node

Increment prerequisites counter

Return

Print prerequisites counter

}

void printSampleSchedule(Tree<Course> courses) {

while prerequisite isn’t null

call a search method using prerequisite to find courses

If course prerequisite is equal to courses

Call numprerequisitecourses using the current node

Print course name, number, and info

}

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

Print course number associated node’s course name

}

======================================================================

//Menu Pseudocode

While user input isn’t ‘4’

Output Main Menu

Output 1 Load Data Structure

Output 2 Print Course List

Output 3 Print Course

Output 4 Exit

Output Enter User Input

If user input is 1

output enter user input (tree, hash, vector)

if user input is tree

initialize tree structure above

else if user input is hash

initialize hash table structure above

else if user input is vector

initialize vector structure above

else

output unrecognized entry! Enter tree/hash/vector

Else if user input is 2

if structure is initialized

run print course list function

Else

Output choose data structure first!!

Else if user input is 3

If structure is initialized

Output enter user input

Run print function using the user input

Else

Output choose data structure first!!

Else if user input is 4

break

Else

output enter menu option 1-4!!

=====================================================================================

//Print in alphanumeric order

-----------------------------------------------------------------------

VECTOR

Compare strings function

If string size 1 is equal to other string size 2

return string1 < string2

Else

return string size 1 < string size 2

Method sortcourses(vector<course> courses)

sort method the vector, lowest to highest via compare strings function

------------------------------------------------------------------------------------------------------------

HASH

Order node to store the next node in the order

Current node to store the node we are at

Swap node to store the node we are swapping to

For all keys

Order node is the node key is pointing to

Order node’s next node is the current node

While order node isnt pointing to nullptr

If Order nodes course number string is greater than the current nodes course number string

Swap them using the swap node variable as a temp value

For all keys

Output nodes from head to tail

-----------------------------------------------------------------------------------------------------------------------------------------

TREE

Output left branch

Output right branch

//everything should already be sorted along the branches, because that’s how the branches are designed

=====================================================================================

## Runtime Analysis

| **Method for loading and reading the files**  **vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create file** | 1 | 1 | 1 |
| **Create lines vector** | 1 | 1 | n |
| **Vectors for course splits** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
| **While not at end of file** | 1 | n | n |
| **If file open** |  |  |  |
| **Get line** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **For loop to go through the lines vector** | 1 | n | n |
| **Split the strings at the commas and assign items to split vector** | 1 | n | n |
| **If split length less than 2** | 1 | 1 | 1 |
| **Output error message** | 1 | 1 | 1 |
| **else** | 1 | 1 | 1 |
| **continue** | 1 | 1 | 1 |
| **For loop going through split containers (course info) by amount of splits** | 1 | n | n |
| **Course number is the first split and goes in course number vector** | 1 | n | n |
| **Course description is the second split and goes in course description vector** | 1 | n | n |
| **Course pre reqs is last splits and goes in prereqs vector, loading blank if no prerequisites** | 1 | n | n |
| **Total Cost** | | | 8n + 13 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Method for loading and reading the files**  **hash** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Course number** | 1 | 1 | 1 |
| **prerequisites** | 1 | 1 | 1 |
| **Course name** | 1 | 1 | 1 |
| **Table size or course max**  **( \* 2 to limit collisions)** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
| **If file open** | 1 | 1 | 1 |
| **Read file, get row total** | 1 | 1 | 1 |
| **For loop from 0 to row total** | 1 | n | n |
| **Get lines, separating lines at commas** | 1 | n | n |
| **Get course information from position of gotten line and row number** | 1 | n | n |
| **Get hash keys using course number** | 1 | n | n |
| **Insert courses into hash table** | 1 | n | n |
| **Total Cost** | | | 5n + 7 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Method for loading and reading the files**  **tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Open file** | 1 | 1 | 1 |
| **If file is open** | 1 | 1 | 1 |
| **Read file** | 1 | 1 | 1 |
| **While file isn’t at the end** | 1 | n | n |
| **Try for loop to go through lines of the file** | 1 | n | n |
| **Getline is all the course information in a line** | 1 | n | n |
| **Add course using node** | 1 | n | n |
| **If root node is null** | 1 | n | n |
| **Root is new node** | 1 | n | n |
| **Else if root node is greater than node** | 1 | n | n |
| **Left node is new node** | 1 | n | n |
| **Else** | 1 | n | n |
| **Right node is new node** | 1 | n | n |
| **Catch, validating getline length** | 1 | n | n |
| **Total Cost** | | | 11n + 3 |
| **Runtime** | | | O(n) |

=====================================================================================

Comparison

Vectors as a data structure are very straightforward and simple, the memory it uses only accounts for the elements stored and doesn’t worry about the pointers like a linked list would. Inserting and deleting items from the end of the list is very quick but doing either anywhere else in the vector can be time consuming as all the other elements will need to be rearranged. The rearranging trouble stems from the vector’s linear nature, all the elements will be kept in sequence, which can come in handy when randomly accessing the middle via the index of the elements.

Hash tables are uniformly fast at searching, deleting, and adding elements. However, that can be a drawback if the element pool is low, because you would be wasting all that efficiency. Then, simply adding more values to the hash table can prove detrimental to the performance due to collisions at the keys that are generated for the added values, to get the desired effect the table would need to be at least double the size of the element pool and the pool would need to be large to take advantage of the tables efficiency.

Binary search trees can be very fast, when they are sorted. Sorting the elements within bst makes it easier to insert and delete items, it has a logarithmic speed in comparison to the other options. When unsorted the performance is lacking, and there is no action with a constant speed in a bst.

=====================================================================================

Recommendation

I think I would choose and recommend the tree structure, after rereading the speeds I think that it would be the most efficient option especially given the relatively small pool of courses. The hash table would not run as fast comparatively and the vector would not sort or insert as fast as the bst. The added bonus of the bst being sorted at the end of its setup, usually, clinches the decision for me.

I realize now that I have done the big o analysis wrong for the bst so I would have to go back over it some other time, but I will say here at the end that it should have a logarithmic-n runtime instead.