# CS 300 Pseudocode Document Nick Glidden

## 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) {

set totalPrerequisites equal to zero

set c equal to the head of courses

set flag equal to true

while (flag)

if currentNode has prerequisites

for all prerequisites listed

increment totalPrerequisites by one

if c.next() is not null (there’s more vectors)

c = c.next()

else (there’s no more vectors)

set flag equal to false

}

void printSampleSchedule(Vector<Course> courses) {

if courses length is equal to one

print course number

print course information

if course has prerequisites

for all prerequisites listed

print prerequisites

else

set start as head of courses

break list into two parts

sort both parts

use merge sort to bring the two back together

set start as head of newly created, sorted courses

while (true)

print course number

print course information

if currentNode has prerequisites

for all prerequisites listed

print prerequisites

else

if c.next() is not null (there’s more vectors)

c = c.next()

else (there’s no more vectors)

break

}

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

set start as head of courses

bool printed = false

while(true)

if start.courseNumber is equal to courseNumber

print course number

print course information

if currentNode has prerequisites

for all prerequisites listed

print prerequisites

else

if c.next() is not null (there’s more vectors)

c = next()

else (there’s no more vectors)

break

if printed is false

print course number does not exist message

}

void loadCourses(string filePath, Vector<Course> courses) {

use the fstream library to read the csv file, passing in the filePath and set it equal to the file variable.

if the file is open

use a while statement, with the conditional being getline() in file

turn line into a stringstream named stringline

define an array of type string, named courseData

while stringline is good (built in method for stringstreams)

parse stringline by comma by using getLine on the stringline, saving each item as a new variable called data, with a ‘,’ as the separator

append data to courseData

use courseData to create a Course object named c, using the Course struct

push c to Vector<Course> courses by calling addNode, with para,maters as c and Vector<Course> courses.

else

print “error opening the input file” message

return Vector<Course> courses

void addNode(Courses new) {

if head of Vector<Courses> courses is empty

set head equal to new

else

set tail equal to new

}

int main() {

define a string variable named inputFile with the url of input data

use a while statement to keep the programming running unless the exit command is pressed

print the following instructions:

(1) load data structure

(2) print course list

(3) print course

(4) exit

enter command:

use a switch statement to accept user input, and route it to the proper logic.

case 1 (load data structure)

call loadCourses, passing in inputFile

break

case 2 (print course list)

call printSampleSchedule, passing in inputFile

break

case 3 (print course)

ask the user for input in the form of a course number

printCourseInformation, passing in the users input

break

default (if an invalid option was inputted)

print a “uh oh. Please try again: message.

break

end of while statement

return 0

}

**Hashtable Pseudocode**

int numPrerequisiteCourses(Hashtable<Course> courses) {

set totalNumber equal to zero

for all keys in courses

if courses[key] contains prerequisites

for all prerequisites returned from courses[key] increment totalNumber

print totalNumber

}

void printSampleSchedule(Hashtable<Course> courses) {

create an empty array called unsortedOutput

for all keys in courses

create an empty string called toPrint

concatenate value.courseNumber to toPrint

concatenate value.courseInformation to toPrint

if value.prerequisites is not equal to null

for all prerequisites in value.prerequisites

concatenate prerequisite to toPrint

append toPrint to unsortedOutput

set sortedOutput equal to the sorted (ascending) unsortedOutput

for each item in unsortedOutput

print item

}

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

set bool printed equal to false

for all keys in courses

if value.courseNumber is equal to courseNumber

print value.courseName

print value.courseNumber

if value.prerequisites is not equal to null

for all prerequisites in value.prerequisites

print prerequisite

set printed to true

break

if printed is equal to false

print course number does not exist message

}

void loadCourses(string filePath, Hashtable <Course> courses) {

use the fstream library to read the csv file, passing in the filePath and set it equal to the file variable.

if the file is open

use a while statement, with the conditional being getline() in file

turn line into a stringstream named stringline

define an array of type string, named courseData

while stringline is good (built in method for string streams)

parse stringline by comma by using getLine on the stringline, saving each item as a new variable called data, with a ‘,’ as the separator

append data to courseData

use courseData to create a new object named new, using the Course struct

push new to Hashtable <Course> courses by calling the addNode function, passing in new as a parameter.

else

print “error opening the input file” message

return Hashtable <Course> courses

void addNode(Course current, Course new) {

generate a new key via hash

set a variable named exisitingItem equal to the value found within the Hashtable <Course> courses at that specified key

if exisitingItem is empty

create a new entry with the key, and the information from new

else

call addNode again, with the same parameters (trying a new key)

}

int main() {

define a string variable named inputFile with the input data

use a while statement to keep the programming running unless the exit command is pressed

print the following instructions:

(1) load data structure

(2) print course list

(3) print course

(4) exit

enter command:

use a switch statement to accept user input, and route it to the proper logic.

case 1 (load data structure)

call loadCourses, passing in inputFile

break

case 2 (print course list)

call printSampleSchedule, passing in inputFile

break

case 3 (print course)

ask the user for input in the form of a course number

printCourseInformation, passing in the users input

break

default (if an invalid option was inputted)

print a “uh oh. Please try again: message.

break

end of while statement

return 0

}

**Tree Pseudocode**

int numPrerequisiteCourses(Tree<Course> courses) {

set totalNumber equal to zero

if root node or root node prerequisites are equal to null

print no prerequisites message

else

set curr as root node

for all nodes in preorder(curr.left)

if node has children

increment totalNumber by number of children

for all nodes in preorder(curr.right)

if node has children

increment totalNumber by number of children

print totalNumber

}

void printSampleSchedule(Tree<Course> courses) {

create associative array called unsortedNodes

if root node is equal to null

print no courses available message

set curr as root node

append unsortedNodes (curr.courseNumber, pointer to curr)

for all nodes in preorder(curr.left)

append unsortedNodes (nodes.courseNumber, pointer to node)

for all nodes in preorder(curr.right)

append unsortedNodes (nodes.courseNumber, pointer to node)

set sortedNodes as a sorted by key copy of unsortedNodes

for each key in sortedNodes

print value.courseNumber

print value.courseInformation

if value.prerequisites is not equal to null

for all prerequisites in value.prerequisites

print prerequisites

}

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

function check(node) {

if node.courseNumber is equal to courseNumber

print node.coursenumber

print node.courseinformation

if node.prerequisites is not equal to null

for all prerequisites in value.prerequisites

print prerequisites

}

if rootnode is equal to null

print no courses to search message

set curr as root node

set printed as false

if check(curr)

for all nodes in preorder(curr.left)

check(curr)

for all nodes in preorder(curr.right)

check(curr)

if printed is equal to false

print course number does not exist message

}

void loadCourses(string filePath, Tree<Courses> courses) {

use the fstream library to read the csv file, passing in the filePath and setting it equal to the file variable.

if the file is open

use a while statement, with the conditional being getline() in file

turn line into a stringstream named stringline

define an array of type string, named courseData

while stringline is good (built in method for string streams)

parse stringline by comma by using getLine on the stringline, saving each item as a new variable called data, with a ‘,’ as the separator

append data to courseData

use courseData to create an node, using the Course struct

push the node to Tree<Courses> courses by calling the addNode function, passing in the node

else

print “error opening the input file” message

return Tree<Courses> courses

void addNode(Course currentNode, Course newNode) {

if currentNode is empty (the start of the tree is empty)

set currentNode equal to newNode

else

if currentNodes courseNumber is larger than newNodes courseNumber

if there is no left node

set the left node of currentNode equal to newNode

else (there is a left node)

call addNode, passing in the left node and newNode

else currentNodes courseNumber is smaller than newNodes courseNumber

if there is no right node

set the right node of currentNode equal to newNode

else (there is a right node)

call addNode, passing in the right node and newNode

}

int main() {

define a string variable named inputFile with the input data

use a while statement to keep the programming running unless the exit command is pressed

print the following instructions:

(1) load data structure

(2) print course list

(3) print course

(4) exit

enter command:

use a switch statement to accept user input, and route it to the proper logic.

case 1 (load data structure)

call loadCourses, passing in inputFile

break

case 2 (print course list)

call printSampleSchedule, passing in inputFile

break

case 3 (print course)

ask the user for input in the form of a course number

printCourseInformation, passing in Tree<Courses> courses and the users input

break

default (if an invalid option was inputted)

print a “uh oh. Please try again: message.

break

end of while statement

return 0

}

## 

## 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.

*Linked List Data Structure*

| **Code** | **Line Cost** | **# of Times** | **Total**  **Cost** |
| --- | --- | --- | --- |
| **Print number of prerequistes** *(numPrerequisiteCourses)* | 5 | n | n |
| **Print all courses and info** *(printSampleSchedule())* | 7 | n | n |
| **Print course information** *(printCourseInformation())* | 8 | n | n |
| **Loading the input** *(loadCourses())* | 4 | n | n |
| **Adding a course** *(addNode())* | 2 | 1 | 2 |
| **Total Cost** | | | 4n + 2 |
| **Runtime** | | | O(n) |

*Hash Map Data Structure*

| **Code** | **Line Cost** | **# of Times** | **Total**  **Cost** |
| --- | --- | --- | --- |
| **Print number of prerequistes** *(numPrerequisiteCourses)* | 5 | n | n |
| **Print all courses and info** *(printSampleSchedule())* | 8 | n | n |
| **Print course information** *(printCourseInformation())* | 4 | 1 | 4 |
| **Loading the input** *(loadCourses())* | 6 | n | n |
| **Adding a course** *(addNode())* | 4 | 1 | 4 |
| **Total Cost** | | | 3n+ 8 |
| **Runtime** | | | O(n) |

*Tree Data Structure*

| **Code** | **Line Cost** | **# of Times** | **Total**  **Cost** |
| --- | --- | --- | --- |
| **Print number of prerequistes** *(numPrerequisiteCourses)* | 2 | n | n |
| **Print all courses and info** *(printSampleSchedule())* | 12 | 1 | 12 |
| **Print course information** *(printCourseInformation())* | 4 | n | n |
| **Loading the input** *(loadCourses())* | 4 | n | n |
| **Adding a course** *(addNode())* | 11 | 1 | 11 |
| **Total Cost** | | | 3n + 23 |
| **Runtime** | | | O(n) |

**Advantages, Disadvantages, and Recommendation**

*Linked List*

I believe one of the major advantages of a linked lists is its ability to be easily added too. When you have data that can be unordered, its really an incredible structure to use. As you can see in my analysis, adding a course rquires the least amount of memory and time, by quite a bit. What starts to become a downfall of linked lists however, is searching and sorting. In both of these categories, needing to transverse a list from head to tail starts to be a downfall.

*Hash Table*

A hash table, in my opinion, is one of the best ways to store this type of information, especially when needing to perform these type of actions to it. A great advantage of a hash table is that its really easy to search, add to, and delete items. All of these things can be done often in very fast speeds, because a hash table doesn’t require O(n) time to do those things if implemetend correctly. A disadvantage of a hash table, is the collisions. When you have more and more collisions in a table, accessing a key’s values becomes more of O(n) time complexity, rather than O(1).

*Tree*

A tree data structure really shines when it is balanced. Thankfuly, we implemented a balanced tree, and we were able to benefit from a lot of its advtanges. These advantages include being able to sort through data quickly, since the tree is already balanced and in a sorted order. Hwoeever, because of the way a tree strucutes itself, searching, deleting, and adding nodes almost always requires O(n) time complexity. This is much slower than both the linked list and a hash table, and is a reason for concern when talking about data that will go through lots of changes.

*My Data Structure Choice Mving Forward*

I will be using the hash table moving forward, because of its incredible accessibility. Generating hashes, and having quick and easy ways to add and search the data is a really large advantage in this use case. I will work to try and minimize collisions, and optimize further if possible.