# 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).

**// File Input Pseudocode**

// Opens text file and appends each line to an entry in the

// courseInfo vector

Vector<String> Reader(String fileName) {

Vector<String> courseInfo

open file filename

if opening file failed

throw Error opening file failed

for each line in file

Append line to courseInfo

Close file filename

Parser(courseInfo)

return courseInfo

}

// Function to test the validity of a read file vector

// Will throw errors if formatting is incorrect or inconsistent

Vector<String> Parser(Vector<String> courseInfo) {

Vector<String> courseNames

Vector<String> prerequisites

for course in courseInfo

// If two parameters exist at least one “,” should appear

if “,” does not exist in course

throw Error file format error

else if course has more than one “,”

for every “,” remaining + 1

add substring from previous “,” to next “,” to prerequisites

add substring from start of course to first “,” to courseNames

for every prereq in prerequisites

if prereq is not in courseNames

throw Error course missing

return courseInfo

}

**// Course Object Pseudocode**

// Course structure object for storing data

Struct Course {

String courseNumber

String courseName

Vector<String> prerequisites

}

// Vector implementation for loading courses

void loadCourses(String fileName, Vector<Course>\* courses) {

courseInfo = Reader(fileName)

for line in courseInfo

Course course = new Course

course.courseNumber = courseInfo line substring from start

to first “,”

if courseInfo line has exactly one “,”

course.courseName = courseInfo line remaining substring

else

course.courseName = courseInfo line substring from

first “,” to second “,”

for each remaining substring separated by “,” + 1

append substring to course.prerequisites

append course to courses

}

// Hashtable implementation for loading courses

void loadCourses(String fileName, Hashtable<Course>\* courses) {  
 courseInfo = Reader(fileName)

for line in courseInfo

Course course = new Course

course.courseNumber = courseInfo line substring from start to first “,”

if courseInfo line has exactly one “,”

course.courseName = courseInfo line remaining substring

else

course.courseName = courseInfo line substring from first “,” to second “,”

for each remaining substring separated by “,” + 1

append substring to course.prerequisites

insert course into courses using course.courseNumber as the key

}

// Tree implementation for loading courses

void loadCourses(String fileName, Tree<Course>\* courses) {

courseInfo = Reader(fileName)

for line in courseInfo

Course course = new Course

course.courseNumber = courseInfo line substring from start to first “,”

if courseInfo line has exactly one “,”

course.courseName = courseInfo line remaining substring

else

course.courseName = courseInfo line substring from first “,” to second “,”

for each remaining substring separated by “,” + 1

append substring to course.prerequisites

insert course into courses using course.courseNumber

}

**// Print Course Information**

// Vector pseudocode

// Function to output properly formatted course information

// on a specific course given

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

for every course in courses

if the course.courseNumber is equal to courseNumber

output course.courseNumber

output “ : “

output course.courseName

if the length of course.prerequisites is greater than

0

Output “ | “

for every prerequisite in course.prerequisites

output prerequisite

output “ “

}

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) {

}

// Hashtable pseudocode

// Function to output properly formatted course information

// on a specific course given

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

unsigned integer key = hashed integer conversion of courseNumber

Course course = &(courses at key)

if course is not empty and courseNumber equals course.courseNumber

output course.courseNumber

output “ : “

output course.courseName

if the length of course.prerequisites is greater than 0

Output “ | “

for every prerequisite in course.prerequisites

output prerequisite

output “ “

while course is not empty

if course.courseNumber equals courseNumber

output course.courseNumber

output “ : “

output course.courseName

if the length of course.prerequisites is greater than 0

Output “ | “

for every prerequisite in course.prerequisites

output prerequisite

output “ “

course = next course

}

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

unsigned integer key = hashed integer conversion c.courseNumber

Course course = &(courses at key)

integer totalPrerequisites = 0

if course is not empty and course.courseNumber == c.courseNumber

for each prerequisite in course.prerequisites

totalPrerequisites += 1

return totalPrerequisites

while course is not empty

if course.courseNumber == c.courseNumber

for each prerequisite in course.prerequisites

totalPrerequisites += 1

return totalPrerequisites

course = next course

return totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

}

// Tree pseudocode

// Function to output properly formatted course information

// on a specific course given

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

Node node = courses root node

courseFinder(node, courseNumber)

course = node.course

if course is not equal to null pointer

output course.courseNumber

output “ : “

output course.courseName

if the length of course.prerequisites is greater than

0

Output “ | “

for every prerequisite in course.prerequisites

output prerequisite

output “ “

}

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

courseNumber = c.courseNumber

Node node = courses root node

integer totalPrerequisites = 0

courseFinder(node, courseNumber)

course = node.course

if course is not equal to null pointer

for each prerequisite in course.prerequisites

totalPrerequisites += 1

return totalPrerequisites

}

// Helper function used to search through courses

void courseFinder(Node\* node, String courseNumber) {

if node equals null pointer or node.course.courseNumber equals courseNumber

return

else if node.course.courseNumber is greater than courseNumber

courseFinder(node->left, courseNumber)

else

courseFinder(node->right, courseNumber)

}

void printSampleSchedule(Tree<Course> courses) {

}

**// Print Schedule Pseudocode**

// Vector implementation of a sorted print

void PrintSorted(Vector<Course> courses) {

Sort(courses)

for every course in courses

PrintFormatter(course)

}

// Hashtable implementation of a sorted print

void PrintSorted(Hashtable<Course> courses) {

Vector<Course> courseList = new Vector

for every node in courses

if node is not empty

append course node to courseList

Sort(courseList)

For every course in courseList

PrintFormatter(course)

}

// Tree implementation of a sorted print

Void PrintSorted(Tree<Course> courses) {

root = root node of courses

inOrder(root)

}

// Helper function to handle sorting the vector and hashtable

void Sort(Vector<Course>\* courses) {

Sort courses with the selection sort algorithm using course.courseNumber from lowest to highest

return courses

}

// Helper function to format and a given course

void PrintFormatter(Course course) {

output course.courseNumber

output “ : “

output course.courseName

if the length of course.prerequisites is greater than

0

Output “ | “

for every prerequisite in course.prerequisites

output prerequisite

output “ “

}

// Helper function to sort and print a tree

void inOrder(Node\* node) {

if node does not equal null pointer

inOrder(node->left)

PrintFormatter(Course at node)

inOrder(node->right)

}

**// Menu Pseudocode**

// Main menu

int main() {

get fileName as String

get data structure for courses

choice = “”

while choice is not “d”

output “a. Load Data Structure”

output “b. Print Course List”

output “c. Print Course”

output “d. Exit”

choice = get user input

if choice == “a”

loadCourses(fileName, courses)

else if choice == “b”

PrintSorted(courses)

else if choice == “c”

output “enter course number”

courseNumber = get user input

printCourseInformation(courses, courseNumber)

}

## 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.

**Evalutation**

| **Vector 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** | 5 | 1 | 5 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | n | 2n |
| **Total Cost** | | | 5n+5 |
| **Runtime** | | | O(n) |

| **Hashtable Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Getting starting course | 2 | 1 | 2 |
| If course is not empty and courseNumber equals course.courseNumber | 1 | 1 | 1 |
| **While course is not empty** | 1 | n | n |
| **If course.courseNumber equals courseNumber** | 1 | n | n |
| **Print course info** | 5 | 1 | 5 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | n | 2n |
| **Course = next course** | 1 | n | n |
| **Total Cost** | | | 6n+8 |
| **Runtime** | | | O(n) |

| **Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Node node = courses root node** | 1 | 1 | 1 |
| **courseFinder(node, courseNumber)** | 1 | Log(N) | Log(N) |
| **Course = node.course** | 1 | 1 | 1 |
| **If course is not equal to null pointer** | 1 | 1 | 1 |
| **Print course info** | 5 | 1 | 5 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | 1 | 2 |
| **Total Cost** | | | Nlog(n)+5 |
| **Runtime** | | | O(nlog(n)) |

**Advantages and Disadvantages**

According to the evaluation for the given function the Big O runtime of both Hashtables and Vectors is equivalent but a Tree data structure would take longer in a worst case scenario. Best case scenarios were not analyzed. Vectors have the benefit of seemingly being the fastest but offering less security than a hash table. Tree offer seemingly no benefit over hashtables or vectors.

**Recommendation**

I believe for the particular use case given, a vector data structure is the best choice for speed and efficiency. Security is not important in this case as the data being stored is not sensitive in any way which makes the slight performance hit with hashtables worthless. Trees also seemingly offer no benefit over vectors or hashtables.