# 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

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 courses) {

for all courses

print course name

if course has prerequisites

for each prerequisite

print prerequisite

}

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 courses, Course c) {

totalPrerequisites = Hashtable[c]

for each prerequisite p in totalPrerequisites

add prerequisites in Hashtable[p] to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable courses) {

for all key, value pair in courses

print key course name

if value has prerequisites

for each prerequisites

print prerequisites

}

void printCourseInformation(Hashtable courses, String courseNumber) {

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the Hashtable[course]

print the prerequisite course information

}

// Tree pseudocode

int numPrerequisiteCourses(Tree courses, Node c) {

totalPrerequisites = left and right child of Node c

for each prerequisite p in totalPrerequisites

add left and right Nodes of node p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree courses) {

for all Nodes as courses

print course name

if course has left node

print left node as prerequisite

if course has right node

print right node as prerequisite

}

void printCourseInformation(Tree courses, String courseNumber) {

for all Nodes

if the course is the same as courseNumber

print out the node's information

else if course has left node

print left node as prerequisite course information

else if course has right node

print right node as prerequisite course information

end

else

if course has left node

go to left node

if course has right node

go to right node

}

## 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) |

Menu Pseudocode

While option does not equal 9

Print “1. Load Data”

Print “2. Print Course List”

Print “3. Print Course”

Print “9. Exit”

Prompt user for input

Switch User menu input

Case 1:

Program loads course data

break;

Prompt user for input

Case 2:

Print course number and name

break;

Prompt user for input

Case 3:

Prompt user to enter course number to search

User input Course number

Print Course number, course name, and prerequisites course numbers

break;

Prompt user for input

Case 9:

Print “Adios”

Print Alphanumeric Order

SET mid to low + (hi - low) / 2

SET pivot to courseName(mid)

WHILE courseName(low) is less than pivot

SET low to low + 1

ENDWHILE

WHILE pivot is less than courseName(hi)

SET hi = hi - 1

ENDWHILEIF low is greater than or equal to hi

RETURN

ELSE

CREATE temp to courseName(low)

SET courseName(low) to courseName(hi)

SET courseName(hi) to temp

SET low to low + 1

SET hi to hi - 1

ENDIF

RETURN hi

FUNCTION main

CALL quicksort(courseName, 0, SIZE - 1)

DISPLAY "Sorted in alphabetical order: "

FOR each course

DISPLAY course info

ENDFOR

ENDFUNCTION

Evaluation

Vector

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| totalPrequisites calc | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Initalize Course Vector | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| Print | 1 | n | n |
|  |  |  |  |
|  |  |  |  |
| Total Cost | | | 3n + 2 |
| Runtime | | | O(n) |

Hash Table

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| totalPrerequisites calc | 1 | 1 | 1 |
| for the course is the same as courseNumber | 1 | n | n |
| Create Hash Table | 1 | n | n |
| Add prerequisites to hash table | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | N |
| print the prerequisite course information | 1 | n | N |
|  |  |  |  |
|  |  |  |  |
| Total Cost | | | 4n + 2 |
| Runtime | | | O(n) |

Binary Search Tree

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| totalPrerequisites calc | 1 | 1 | 1 |
| if the course is the same as courseNumber | 1 | n | n |
| Create Binary Tree | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| print the prerequisite course information | 1 | n | n |
| Else if | 1 | n | n |
| Left node | 1 | n | n |
| Right node | 1 | n | n |
| Total Cost | | | 6n + 2 |
| Runtime | | | O(n) |

Vectors will be fast at reading and adding objects but will be slower when searching since it goes one by one. The hash table is fast at searching since it uses keys to map items but is slow at sorting alphanumeric. The binary search tree is fast at sorting objects, but it takes more time to make modifications.

Based on the information above, I recommend vectors due to the minimal runtime. Since all three have their advantages and disadvantages, vectors can complete the task the fastest overall. Vector’s speed when reading and adding objects outweighs its lackluster search and print times.