# 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

return totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

for all courses:

if course is on schedule:

print course name and time

}

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**

}

void printByAlphaSort(Vector<Course> courses)

alphaSortedCourses = clone(courses)

quickSort(alphaSortedCourses, 0, alphaSortedCourses.length() -1)

for course in alphaSortedCourses:

print course information

}

Vector<Course> quickSort(Vector<Course>, start, end){

int mid = 0;

if (begin >= end) {

return;

}

int lowIndex = partition(bids, begin, end);

quickSort(bids, begin, lowIndex);

quickSort(bids, lowIndex + 1, end);

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

return size of courses

}

void printSampleSchedule(Hashtable<Course> courses) {

for courseBucket in courses:  
 for courses in bucket:

print course name and time

}

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

courseInformation = courses[courseNumber]

print the courseInformation

for each prerequisite of courseInformation:

print prerequisite course information

}

printByAlphaSort(Hashtable<Course> courses) {

Vector coursesVector;

for courseBucket in courses:  
 for courses in bucket:

courseVector.add(course)

quickStort(courseVector, 0, courseVector.lenght())

for course in courseVector:

print course information

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

for child in courses root node:  
 count ++

return count

}

void printSampleSchedule(Tree<Course> courses) {

if node is on schedule:

print root name and time

for children of node:

printSamepleSechedule(child)

}

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

Tree<Course> findCourse(Tree<Course> root, courseNumber) {

if root is null:

return null course  
 if root’s course number is courseNumber:

return course node

else:

for children of root:

course = printCourseInformation(child)

if course is not null:

return child

return null course

}

root = findCourse(courseNumber)

print root node course information

for child in root:  
 print child course information

}

void printAlphaSort(Tree<Course> courses) {

Tree alphaCourses = clone(courses)

alphaSort(alphaCourses)

}

void alphaSort(courses, alphaCourses) {  
 Vector coursesVector;

createVector(coursesVector, courses.root)

for course in coursesVector:

Node node = new Node()

node→course = course

placeAlphaSortNode(alphaCorses, node)

}

void createVector(Vector<course> courses, Node node) {  
 if Node is null:  
 return

else:  
 courses.add(node.course)

createVector(courses, node.left)

createVector(courses, node.right)

}

void placeAlphaSortNode(Tree<course> AlphaCourses, Node node) {

node curr, prev

while curr is not null:

if current.title < node.title:

prev = current

current = current.left

else:

prev = current

current = current.left

if prev.left.title < node.title:

prev.left = node

else:

prev.right = node

}

**basic functions:**

struct course {  
 int ID;

vector prerequisiteIds;

string title;

string description;

}

void menu() {

int selection;

while (selection != 4) {

print( “ Menu:

1. Load Courses

2. Display All Course Information

3. Display a Single Course’s Information

4. Exit

)

executeMenuSelection(selection);

}

}

void executeMenuSelection(int selection) {

case 1:

readFile()

case2:

printCourseInformation()

case3:

string inputId = print(“input course ID:”)

try { cast to int }

try { printcourseInformation(courseID)}

except { print( invalid input) or print( courseID not found) }  
 Case4:

return

}

void readFile() {

for line in file:

Course course = new Course()

try { put each value in course }

except {print (invalid input)}

hashtable.insert(course)

}

## 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) |

**Vector printCourseInformation**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **totalPrerequisites = prerequisites of course c** | 1 | 1 | 1 |
| **for each prerequisite in totalPrerequisites** | 1 | n | n |
| **add prerequisites of p to totalPrerequisites** | 1 | n | n |
| **Return totalPrerequisites** | 1 | 1 | 1 |
|  |  |  |  |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

**Vector printSampleSchedule**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For all courses** | 1 | n | n |
| **If course is on schedule** | 1 | n\*m | n\*m |
| **Print course name and time** | 1 | m | m |
| **Total Cost** | | | 2n \* 2m |
| **Runtime** | | | O(nm) |

**Hashtable numPrerequisiteCourses**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **return size of courses** | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

**Hashtable printSampleSchedule**

**(n = number of rows in table, m = number of courses)**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for courseBucket in courses** | 1 | n | n |
| **for courses in bucket:** | 1 | n \* m | n \* m |
| **print course name and time** | 1 | n \* m | n \* m |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

**Hashtable printCourseInformation**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **courseInformation = courses[courseNumber]** | 1 | 1 | 1 |
| **print the courseInformation** | 1 | 1 | 1 |
| **for each prerequisite of courseInformation:** | 1 | n | n |
| **print prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 2 + 2n |
| **Runtime** | | | O(n) |

**Tree numPrerequisiteCourses**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for child in courses root node:** | 1 | n | n |
| **count ++** | 1 | n | n |
| **return count** | 1 | 1 | 1 |
| **Total Cost** | | | 1 + 2n |
| **Runtime** | | | O(n) |

**Tree printSampleSchedule**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **print root name and time** | 1 | 1 | 1 |
| **for child of root:** | 1 | n | n |
| **printSamepleSechedule(child)** | 1 | n | n |
| **Total Cost** | | | 1 + 2n |
| **Runtime** | | | O(n) |

**Tree printCourseInformation**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **if root is null:** | 1 | n | n |
| **return null course** | 1 | n | n |
| **if root’s course number is courseNumber:** | 1 | n | n |
| **return course node** | 1 | n | n |
| **else** |  |  |  |
| **for children of root:** | 1 | n | n |
| **course = findCourse(child)** | 1 | n | n |
| **if course is not null:** | 1 | n | n |
| **return child** | 1 | n | n |
| **return null course** | 1 | n | n |
| **...** |  |  |  |
| **print root node course information** | 1 | 1 | 1 |
| **for child in root** | 1 | n | n |
| **print child course information** | 1 | n | n |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

Data structure Analysis

**Vector:**

Vectors can be very easy and intuitive to use, but the downside is often very slow computations. In this particular use case, honestly, the data structure does not matter too much due to low amounts of data, and using a vector would likely make no difference. However, if there were to be much larger amount of data, the inefficiency would likely be noticeable, especially if this were to be utilized in an environment were speed was required.

**Hash Table:**

Hash Tables are very efficient and effective at what they do. In this case, they can retrieve information at near constant time with little additional space. I think a hash table could be highly effective, as students will likely only need to retrieve the information in the hash table. Additionally, the relationships between the courses can be saved in the table itself. However, their weakness comes in when it comes to representing the relationship that object in a hash table have to one another. A poorly created or maintained hash table may have a high number of collisions. Updating courses would likely become complicated as well, since you would need to check every single course to ensure the prerequisites were still correct. This would only happen occasionally. When It comes to sorting, their really is no realistic way to use a hash table for this. That is why if I were to implemented this solution I would create a separate sorted, pre-computed, vector of course pointers that updates with new inserts and removals.

**Binary Tree:**

Binary trees can be highly effective with the right kind of data. They have consistently short searches and inserts. It would be much more common in this application that the information would need to be retrieved. While binary trees are fast about this, they are not near constant, like a hash table is. Still, it would be fairly effective, but perhaps overly complicated for such a simple project.

**The Best Option:**

In my opinion the best option for this project would be the hash map. While inserts may be slower than on a binary tree, It would be much faster to doing searches. This would be the bottleneck of the program, since much of the core operations of this application would be retrieving the course information. Compared to using and array or vector, the hash table is a clear winner. In an array or vector, searching is O(n). This would slow down the application significantly.