# CS 300 Pseudocode Document

## Joshua Battaglia

// File Reading pseudocode:

open file

if file not found

print “file not found”

else

while not end of file

if greater than one value in line

if less than three values in line

read parameters

else

if third value is found in first value of other line

continue

else

print error

else

print error

close file

// 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<Course> courses) {

course = element of courses

for each course in courses

if course has prerequisite AND prerequisite in courses

courses[course – 1] = prerequisite

print ordered 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:

int numPrerequisiteCourses(Hashtable<Course> courses) {

count = 0

for all courses

if the course has prerequisite

increment count

print count

}

void printSampleSchedule(Hashtable<Course> courses) {

for node begin to end iterate

if key not equal to Max

print courseNumber, name, and prerequisite

node = next node

while node not equal to nullptr

print courseNumber, name, and prerequisite

node = next node

}

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

key = courseNumber

for course in courses

if entry found for the key

return course of courseNumber

if no entry is found for the key

return course

while node is not null

if key not found

return course

if the current matches

return course

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

if node is not equal to null

recursively traverse left of current node

recursively traverse right of current node

display current node course information

}

void printSampleSchedule(Tree<Course> courses) {

if node is not equal to null

traverse left course nodes, recursively

display course node starting from left

traverse right course nodes, and display recursively

}

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

set current node to equal root

while current course node not equal null

if current courseNumber node equals courseNumber

return current course

else if current courseNumber node is less than courseNumber

current node equals left of current node

else

current node equals right of current node

return course

}

## Analysis:

Data structures all have their strengths and weaknesses when being used in a program and/or application. The first example is the vector data structure which is similar to a list. The advantage of using a vector is the total runtime when searching a vector for a particular value of data. In our example the vector has the shortest runtime at 4n+2. The disadvantage of using the vector is the worst case will always include searching every element in the vector until the value is found.

For our next data structure, hashtables, we must note that hashtables themselves cannot be sorted. Instead a key is created for a specified location and easily searched within the table. Although this seem easier than the process used for a vector, the implementation of the hashtable is slow since elements must be extracted, then sorted. This data structure would not be ideal for a course sorting application, but rather something along the lines of secure password encryption and storage. Total runtime for the hashtable is 10n+1.

Lastly we have binary trees which can be quickly sorted but is not as easily implemented as the previous structures. When looking at how the courses are visually represented, it shares similarities with a tree diagram. It allows to easily find prerequisite courses and although slightly slower than a vector, be able to print all classes. I think the binary tree is the best choice for the course application since the ability to sort and find prerequisite courses for a particular class is the main purpose for this application.

**Runtime Analysis Charts:**

| **Code Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create vector | 1 | 1 | 1 |
| **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 + 2 |
| **Runtime** | | | O(n) |

| **Code Hashtable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create hashtable | 1 | 1 | 1 |
| **for course in courses** | 1 | n | n |
| **if entry found for the key** | 1 | n | n |
| **return course of courseNumber** | 1 | n | n |
| **if no entry is found for the key** | 1 | n | n |
| **return course** | 1 | n | n |
| **while node is not null** | 1 | n | n |
| **if key not found** | 1 | n | n |
| **return course** | 1 | n | n |
| **if the current matches** | 1 | n | n |
| **return course** | 1 | n | n |
| **Total Cost** | | | 10n+1 |
| **Runtime** | | | O(n) |

| **Code Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create tree | 1 | 1 | 1 |
| **while current course node not equal null** | 1 | n | n |
| **if current courseNumber node equals courseNumber** | 1 | n | n |
| **return current course** | 1 | n | n |
| **else if current courseNumber node is less than courseNumber** | 1 | n | n |
| **current node equals left of current node** | 1 | n | n |
| **else**  **current node equals right of current node** | 2 | n | 2n |
| **Total Cost** | | | 7n+1 |
| **Runtime** | | | O(n) |