# 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<Course> courses) {

for each course in courses

print course.getName()

if course has prerequisites

for each prerequisite in course.getPrerequisites()

print "- " + prerequisite.getName()

}

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) {

totalPrerequisites = courses.get(c);

for each prerequisite p in totalPrerequisites

add prerequisites in courses.get(p) to totalPrerequisites

return the number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for each course in courses

print course.getName()

if course has prerequisites

for each prerequisite in course.getPrerequisites()

print "- " + prerequisite.getName()

}

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

for each course in courses

if course.getKey() is equal to courseNumber

print out the course information

for each prerequisite in course.getValue().getPrerequisites()

print the prerequisite course information

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

for each prerequisite p in totalPrerequisites

addPrerequisitesRecursively(courses, p)

return totalPrerequisites.size()

}

void printSampleSchedule(Tree<Course> courses) {

printCourseAndPrerequisites(courses.getRoot(), 0)

}

void printCourseAndPrerequisites(Node<Course> node, int level) {

for all Nodes as courses

print course name

if course has left node

print left node as prerequisit

if course has right node

print right node as prerequisit}

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

for all Nodes

if the course is the same as courseNumber

print out the node's information

if course has left node

print left node as prerequisite couse information

if course has right node

print right node as prerequisite couse information

end Function

else

if course has left node

goto left node

if course has right node

goto 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) |

Vector Data Structure:

File Reading: The operation involves reading each line of the file, which takes O(n) time per line, where n is the number of lines. For a file with m lines, the total time complexity is O(n \* m).

Course Object Creation: Parsing each line and creating course objects involves extracting attributes, each taking O(1) time. With k attributes per course, creating one course object takes O(k) time. Thus, creating all course objects takes O(n \* k) time.

Hashtable Data Structure:

File Reading: Similar to the Vector data structure, file reading takes O(n \* m) time.

Course Object Creation: The time complexity for creating all course objects remains O(n \* k), as in the Vector data structure.

Tree Data Structure:

File Reading: Like the Vector and Hashtable data structures, file reading takes O(n \* m) time.

Course Object Creation: The time complexity for creating all course objects remains O(n \* k), as in the Vector and Hashtable data structures.

Advantages and Disadvantages:

Vector:

Advantages: Simple implementation, efficient for sequential access, contiguous memory allocation, suitable for small to medium-sized datasets.

Disadvantages: Slow insertion and deletion operations for large datasets, potential performance issues with resizing.

Hashtable:

Advantages: Fast lookup and insertion time on average, suitable for large datasets, supports key-value pairs, efficient random access.

Disadvantages: Memory overhead due to hashing, potential collisions, lack of order, potential rehashing overhead.

Tree:

Advantages: Efficient insertion, deletion, and lookup operations, maintains order, suitable for sorted data, supports traversal operations.

Disadvantages: Memory overhead for pointers, potential imbalance leading to worst-case performance, may require balancing for optimal performance.

Recommendation:

Considering the runtime complexities and operation characteristics, employing a Hashtable data structure emerges as the most appropriate choice. Hashtables provide swift lookup and insertion times, which are advantageous for handling extensive datasets efficiently. Moreover, they strike a balance between memory utilization and performance, rendering them well-suited for this task.