# 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) {

**Sort courses by courseNumber**

**For each course in courses**

**Print "Course Number: " + course.courseNumber + ", Title: " + course.title**

}

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

**For each course in courses**

**If course.courseNumber is equal to courseNumber**

**Print "Course Number: " + course.courseNumber**

**Print "Course Title: " + course.title**

**If course.prerequisites.size() is 0**

**Print "No prerequisites."**

**Else**

**Print "Prerequisites: " + Join(course.prerequisites, ", ")**

**Return**

**Print "Course not found."**

**End Function**

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

**If courses does not contain courseNumber**

**Return 0**

**Initialize an empty Hashtable for visited courses to handle cyclic dependencies**

**Initialize a queue or stack for DFS or BFS**

**Push courseNumber onto queue/stack**

**While queue/stack is not empty**

**String currentNumber = pop queue/stack**

**Course current = courses.get(currentNumber)**

**For each prerequisite p of current**

**If p is not in visited**

**Add p to visited**

**Push p onto queue/stack**

**Return size of visited**

}

void printSampleSchedule(Hashtable<Course> courses) {

**Convert courses keys to a list and sort it alphanumerically**

**For each key in sorted keys**

**Course course = courses.get(key)**

**Print "Course Number: " + course.courseNumber + ", Title: " + course.title**

}

}

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

**Calculate hash key for courseNumber**

**Retrieve Course object 'course' from 'courseTable' using hash key**

**If course is not found:**

**Print "Course not found."**

**Return**

**Print "Course Number: " + course.number**

**Print "Course Name: " + course.name**

**Print "Prerequisites:"**

**If course has prerequisites:**

**For each prerequisite in course.prerequisites:**

**Print prerequisite**

**Else:**

**Print "None"**

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

**Course c = find course in tree courses with courseNumber**

**If c is not found**

**Return 0**

**Initialize an empty Set for visited courses to handle cyclic dependencies**

**Initialize a queue or stack for DFS or BFS**

**Push c onto queue/stack**

**While queue/stack is not empty**

**Course current = pop queue/stack**

**For each prerequisite p of current**

**If p is not in visited**

**Add p to visited**

**Course prerequisiteCourse = find course in tree courses with p**

**Push prerequisiteCourse onto queue/stack**

**Return size of visited**

}

void printSampleSchedule(Tree<Course> courses) {

**Perform in-order traversal on courses tree**

**For each visited course**

**Print "Course Number: " + course.courseNumber + ", Title: " + course.title**

}

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

**Find the Course object in the tree using courseNumber**

**If Course object is found:**

**Print course information for the found Course object**

**For each prerequisite in the Course object's prerequisites:**

**Recursively call printCourseInformation with tree and prerequisite**

**Else:**

**Print "Course not found"**

}

## 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.

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** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | O(n) |

Hash

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 2 | 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** | 2 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

Tree

| **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** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 8n + 1 |
| **Runtime** | | | O(n) |

**Vector**

**Advantages:**

* Simple and straightforward to implement.
* Direct access to elements by index.
* Maintains order of insertion, which simplifies tasks such as printing courses in the order they were added.

**Disadvantages:**

* Lookup, insertion, and deletion operations (except at the end) are O(n), which can be inefficient for large datasets.
* Requires manual sorting to print courses in alphanumeric order, which is an additional O(n log n) operation.

**Hash Table**

**Advantages:**

* Efficient for scenarios where quick access to elements based on keys is required.

**Disadvantages:**

* Does not maintain the order of elements, requiring additional steps to sort keys for printing in alphanumeric order.
* Hash collisions can degrade performance to O(n) in the worst case, although this is rare with a good hash function and sizable hash table.

**Tree**

**Advantages:**

* Maintains elements in a sorted order, facilitating efficient in-order traversal for printing (O(n) for traversing all elements).
* Offers O(log n) lookup, insertion, and deletion operations in the average case for balanced trees, which is more efficient than vectors for large datasets.
* Efficiently handles operations that require elements to be ordered.

**Disadvantages:**

* Can degrade to O(n) operations in the worst case when the tree becomes unbalanced (e.g., inserting elements in sorted order without rebalancing).

I would recommend using a vector for the course management system. The choice is because of vectors' simplicity and their ability to maintain the order of courses as they are added, which better the advisor's requirements for listing and accessing course information efficiently. While vectors may have linear search times (O(n)), this trade-off is good given the ease of implementation and the likely manageable size of the course dataset, where the impact on performance would be minimal.