# 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) {

}

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

1. int numPrerequisiteCourses(Hashtable<Course> courses) { Open the Course Information file
2. Initialize an empty hash table called 'courseHashTable'
3. Read the data from the file line by line

a. For each line:

i. Split the line into a list of tokens using the delimiter (e.g., comma)

ii. Validate the list of tokens: - If the list has less than two parameters (courseNumber and title), report a format error and exit

iii. Extract courseNumber, title, and prerequisites (if any) from the list of tokens

iv. Validate the prerequisites: - For each prerequisite, check if it exists as a courseNumber in the file (e.g., by searching the previously processed lines or using a separate hash set) - If a prerequisite does not exist, report a format error and exit

v. Create a new course object with the extracted courseNumber, title, and prerequisites

vi. Store the course object in the courseHashTable using courseNumber as the key

1. Close the Course Information file

}

void printSampleSchedule(Hashtable<Course> courses) {

}

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

1. Define a function called 'printCourseInformation' that takes the courseHashTable as input
2. For each key-value pair (courseNumber and course object) in the courseHashTable:
   1. Print the courseNumber and the course title b. If the course object has prerequisites:
      1. Print "Prerequisites:"
      2. For each prerequisite in the course object's prerequisites:
         1. Retrieve the prerequisite course object from the courseHashTable using the prerequisite course number
         2. Print the prerequisite course number and title

Call the 'printCourseInformation' function with the courseHashTable as input

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) { Declare a variable "fileName" and assign the path of the file "CourseInformation.txt"

Open the file using the variable "fileName"

Declare a variable "lineNumber" and initialize it to 1

For each line in the file:

Increment "lineNumber" by 1

Split the line by comma into an array of strings called "tokens"

If the length of "tokens" is less than 2:

Print "Error: not enough parameters on line" and the value of "lineNumber"

Continue to the next line

If the length of "tokens" is greater than 3:

Print "Error: too many parameters on line" and the value of "lineNumber"

Continue to the next line

Assign the first token to a variable "courseNumber"

Assign the second token to a variable "courseTitle"

If the length of "tokens" is 3:

Assign the third token to a variable "prerequisite"

Check if there is a course in the file with the courseNumber "prerequisite"

If not, print "Error: prerequisite course not found on line" and the value of "lineNumber"

}

void printSampleSchedule(Tree<Course> courses) {

Create an empty vector called "courses"

For each line in the file:

Split the line by comma into an array of strings called "tokens"

Create a new course object called "course"

Assign the first token to the "courseNumber" property of "course"

Assign the second token to the "courseTitle" property of "course"

If the length of "tokens" is 3:

Assign the third token to the "prerequisite" property of "course"

Push "course" to the "courses" vector

Close the file

}

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

Create a new empty tree called "courseTree"

For each course in the "courses" vector:

Add the course to the "courseTree" with the courseNumber as the key

For each key in the "courseTree"

Print the key

Print the courseTitle of the course

If the course has a prerequisite:

Print "Prerequisite: " and the prerequisite courseNumber

Print a new line

}

## 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) |

Data Structures Analysis

a. Vector:

* Advantages: Simple implementation, quick access to elements by index, dynamic resizing, good for iterating through elements.
* Disadvantages: Insertion and deletion can be slow (O(n) worst-case) since elements may need to be shifted, not efficient for searching elements unless sorted.

b. Hash Table:

* Advantages: Fast average-case access, insertion, and deletion time (O(1)), good for searching elements with unique keys.
* Disadvantages: Space complexity can be high, hash function and resizing might be complex, poor worst-case performance.

c. Tree (Binary Search Tree):

* Advantages: Efficient searching, insertion, and deletion (O(log n) average-case), sorted data structure, and balanced memory usage.
* Disadvantages: Complex implementation, might become unbalanced (rebalancing needed), not efficient for random access by index.

1. Recommendation

Based on the analysis, we recommend using a hash table for this problem. The justification is as follows:

* Since the Big O value is O(n \* (T(parseLine) + T(createCourse))), choosing a data structure that provides fast average-case access, insertion, and deletion time is crucial.
* In the case of course objects, the courses can have unique identifiers (e.g., course codes) which can act as keys for the hash table.
* The hash table provides average-case O(1) complexity for access, insertion, and deletion, which is better than a vector (O(n) for insertion and deletion) and a binary search tree (O(log n) for insertion, deletion, and search).

Although a hash table has some disadvantages like space complexity and potential worst-case performance issues, the average-case performance benefits outweigh these drawbacks for the given problem.