# 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) {

prompt user to enter desired course as a string

iterate through vector to find matching courseID

if found, then cout courseID, courseName

cout the PR courses and ask if student has already completed them

if yes then cout the course and cout other courses that have either NO PR or the already completed PRs alphabetized by lowest value first alphabetically then highest value last.

if not found then tell user that it does not exist and to check their spelling

}

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) {

Each course has a key and that key corresponds to a specific bucket

Prompt user to enter the desired courseID

Take that key and cout the objects that correspond to that course’s bucket of PRs

Any sorting method will work here because we are only dealing with at most 2 PRs

Increment counter every time we find a PR

Cout that counter

}

void printSampleSchedule(Hashtable<Course> courses) {

prompt user for courseID

Find the corresponding PRs from that key

Ask user if the student has already completed the PRs

If yes then alphabetize based off of lowest value of courseID OR courseName first alphabetically and highest value last.

display courses that have NO PRs or that have only the PRs that were indicated as completed

If no then indicate to user that PRs must be completed.

}

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

prompt user to enter courseID

use that key to access the object in whatever bucket it may be in

cout that course object’s id, name, PR1, and PR2

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

prompt user for courseID

iterate through tree until node found

given course will have at most 2 PRs so a left and right node

if left node (PR1)

increment counter by 1

if right node

increment counter by 1

cout the counter

}

void printSampleSchedule(Tree<Course> courses) {

prompt user for courseID

find node within tree

Place all nodes into a vector and organize alphabetically based off either courseID OR courseName.

Complete this by comparing values for each string

Lowest value is first alphabetically.

cout all nodes prior to that given node

cout each courseID, and course name

}

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

prompt user for courseID

iterate through tree until node found

cout nodes courseID, courseName, PR1, and PR2 if applicable

}

## 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** | **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) |

| **Hashtable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **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** | | | 3n + 2 |
| **Runtime** | | | O(n) |

| **Tree** | **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) |

Evaluation:

There are many factors to consider when determining what type of structure we will choose for this problem.

A benefit of choosing a vector is that we have all of our objects in one each to iterate structure. There is no real nested loops we will need to iterate through the courseVector. This will make for quick iteration. However, we will need to iterate through it once to find the course and possibly a second or third time to find the PR objects.

A benefit of using a hashtable is that we simply need to search for a key rather than a courseID. Also, the way that PR’s will be grouped prior to or subsequent to the course will make for fewer iterations when it comes to printing the objects themselves. However, we will likely run into multiple objects in the same bucket and therefore this solid structure will likely become more congested. We would also have to have courses in multiple times for a more easy searching process rather than trying to have arbitrary pointers for each course pointing to the same PR. This could also cause issues when it comes to a course having no PR OR a course being a PR to multiple courses.

Lastly, the tree might be the best approach in terms of speed of execution. However, when it comes to the structuring of the tree, we will likely run into duplicated courses as PR’s. We will also likely run into a problem where a course doesn’t have a PR but ones that we expected to place after it do. We cannot have any gaps in our tree. We also will have to implement code to find the midpoint or best point to start the tree as well as ensure that the tree is in proper order. All of these structuring steps will most certainly add up to create more code and more time than is necessary.

For these reasons I suggest using the vector approach for this problem. The ONLY issue that we will run into when it comes to the vector approach is having to iterate through it at most 3 times. This is not the most efficient method, however, given the possible issues that we could run into with the tree or hash table, this method will give us the most predictability and reliability on both the backend and the front end. The vector is so flexible because we could even store the individual parts of the courses as different indices in the vector if we wanted to. This would make the iteration less efficient, however. Storing the course objects in the Vector is the best approach to this situation.

1. Create course struct
   1. courseObjID
   2. courseObjName
   3. courseObjPR1
   4. courseObjPR2
   5. void PrintAllCourses() {}
   6. void PrintCourse() {}
2. void Menu() {
   1. cout “ABCU Course Planner”
   2. cout “1. Load Data Structure”
   3. cout “2. Print Course List”
   4. cout “3. Print course”
   5. cout “9. Exit”
   6. cout “Please make a selection from the list above”
   7. get user input into menuInput variable }
3. void ClearScreen() {
   1. cout \n multiple times }
4. void LoadDataStructure() {
   1. Prompt user for fileName
   2. Create file object….ifstream file
   3. Open file……file.open(filename)
   4. File open verification
      1. If (!file.is\_open()) {
         1. Cout file failed to open. }
   5. While loop (getline(file, line)) {
      1. Stringstream ss(line)
      2. Getline(ss, courseID, ‘,’)
      3. Getline(ss, courseName, ‘,’)
      4. Getline(ss, PR1, ‘,’)
      5. Getline(ss, PR2, ‘,’)
      6. Create course object from Course struct….Course course
      7. Save values to Course atts
         1. course.courseObjID = courseID
         2. course.courseObjName = courseName
         3. course.courseObjPR1 = PR1
         4. course.courseObjPR2 = PR2
      8. courseVector.push\_back(course)
      9. fileLoaded bool set to true
      10. Close file
      11. PostLoadMenu() }
5. PrintCourseList()
   1. ClearScreen()
   2. Verify file is loaded
      1. No
         1. Tell user to load data structure
         2. Call Menu()
      2. Yes
         1. For loop
            1. Print courseId at i
            2. Print courseName at i
            3. Print coursePR1 at i
            4. Print coursePR2 at i
6. PrintCourse()
   1. ClearScreen()
   2. Verify file is loaded
      1. NO
         1. Tell user to load data structure
      2. Yes
         1. Ask user what courseID they’d like
         2. For (int I = 0; I <= courseVector.size(); i++)
            1. If courseSelection == courseVector.at(i).courseObjID)

Cout << courseVector.at(i).courseObjID

Repeat for Name

Repeat for PR1

Repeat for PR2

* + - * 1. Else if (I == courseVector.size())

Cout << course does not exist

PostLoadMenu()

1. Exit()
   1. Cout Exiting Program