# 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 all courses

print the name of the course

if (course has prerequisites){

print each prerequisite

}

}

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

input courseNumber

//while vector isn’t empty

while(vector > 0){

if(input == courseNumber){

output courseNumber<< output courses

while(Prerequisite == true)[

print prerequisite

}

}

}

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

total prerequites equal the hashtable[c]

for (each prerequisite p in totalPrerequites){

add prerequisite in the hashtable[p]

print the totalPrerequisite

}

}

void printSampleSchedule(Hashtable<Course> courses) {

for (all key, value pair in the courses){

print key course name

if (the value has prerequisites){

for (each prerequisite){

print prerequisite

}

}

}

}

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

get courseNumber

key = courseNumber

assign this\* node to match node.at(key)

if(node == node.at(key){

return course

}

if(node == null){

return null

}  
 else{

If(key== courseNumber){

Node = next node

}

}

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = left and right child of Node

for (each prerequisite in totalPrerequisite){

add left and right Nodes of node p to totalPrerequisite}

// this adds prerequisite courses if any are found

}

void printSampleSchedule(Tree<Course> courses) {

for (all Nodes as courses){

print course name

if (course has left Node){

print the left Node prerequisite}

if (course has right Node){

print the right Node prerequisite}

// this searches all the courses and prints both right and left

}

}

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

get courseNumber;

assign current node to the root

while(current node != null){

if (current course is the same as courseNumber){

return current, output courseNumber << course

while(numPrerequisiteCourse > 0){

print totalPreprequite

}

}

if(courseNumber < root){

current node = left node

}

else{

Current node = right node

}

}

//Menu

//Set choice to 0 for menu case options

Int choice = 0

While(choice != 4){

Out menu choices(load course files, print course list, print a course, exit)

Input users choice

Case 1: load course files

Case 2: print the course list

Case 3: print a select course

Case 4: exit

}

**PRINT SORTED LIST**

printSorted(courses)

create partition(vector<course>&courses, int begin, int end){

sort(courses.begin(),courses.end())

}

displayCourse(courses){

cout<<course.courseId<<”: “ <<course.title<< “ | “ << course.prerequisite<< endl;

loop through display courses

for(int I =0;i<courses.size();i++){

displayCourse

}

}

Create order for the tree

If(node!= null)

Check left side

Node = left node

cout<<course.courseId<<”: “ <<course.title<< “ | “ << course.prerequisite<< endl;

Check right side

Node = right node

cout<<course.courseId<<”: “ <<course.title<< “ | “ << course.prerequisite<< endl;

## 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** |
| --- | --- | --- | --- |
| **Create vector** | 1 | 1 | 1 |
| **if the course is the same as courseNumber** | 1 | n | n |
| **Create the course information** | 1 | n | n |
| **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 |
| Create hash table | 1 | 1 | 1 |
| Create key | 1 | n | n |
| if the entry is found for key | 1 | n | n |
| Assign node to key | 1 | n | n |
| else | 1 | n | n |
| Assign old key the next node pointer | 2 | n | 2n |
| Else | 1 | n | n |
| Find next node | 1 | n | n |
| Add new Node | 1 | n | n |
| For each new line | 1 | n | n |
| Create the course item | 1 | n | n |
| For prerequisite | 1 | n | n |
| For inserting course items | 1 | n | n |
| **Total cost** |  | | 14n + 1 |
| **Runtime** |  | | O(n) | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tree** | Line cost | #Time executes | Total Cost |
| For add nodes | 1 | 1 | 1 |
| For root if it is not null | 1 | n | n |
| If node is less than root, add left | 1 | n | n |
| Else current becomes left | 1 | n | n |
| If node is greater than right | 1 | n | n |
| Else current becomes right | 1 | n | n |
| For lines in file | 1 | n | n |
| Create course item | 1 | n | n |
| For prerequisite | 1 | n | n |
| For inserting course items | 1 | n | N |
| **Total Cost** |  |  | 9n +1 |
| **Runtime** |  |  | 0(n) |

Each structure is going to have advantages and disadvantages for their programs. Vector is the quickest of the methods. However, it may be quick at reading the files and adding, it is slow searching. It must read every item throughout the list of the vector until it finds its selected course. This is not convenient for the items at the end of the list.

HashTables are good for fast data accessing. It can provide great searching, inserting, and deleting. However, the inserting and deleting are slower. Also, hashing cannot be sorted. This will cause trouble if you try to print the courses and need them sorted.

Trees have quick sorting. They can search the best due to their sorting within the leaves of the tree.

This project I would either recommend a tree or vector. This would depend on its size. If we are only coding a program for ABCU’s Computer Science department, I would pick a vector. The list of courses would not be too long so there would be no need for a more complex system. However, if this project had the potential to be used in the other departments as well then, the tree would be needed. There would be too many courses for all departments to search if the program were a vector.