# CS 300 Pseudocode Document

Brian Bentley

10/5/2022

## 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

int numPrerequisiteCourses(Hashtable<Course> courses) {

**totalPrerequisites = Hashtable[c]  
 for each prerequisite p in totalPrerequisites  
 add prerequisites in Hashtable[p] to totalPrerequisites  
 print number of totalPrerequisites**

}

void printSampleSchedule(Hashtable<Course> courses) {

**for all key, value pair in courses  
 print key course name**

**if value has prerequisits  
 for each prerequisits  
 print prerequisits**

**}**

void printCourseInformation(Hashtable<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 Hashtable[course]  
 print the prerequisite course information**

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

**totalPrerequisites = left and right child of Node c  
for each prerequisite p in totalPrerequisites  
 add left and right Nodes of node p to totalPrerequisites  
 print number of totalPrerequisites**

}

void printSampleSchedule(Tree<Course> courses) {

**for all Nodes as courses  
 print course name  
 if course has left node  
 print left node as prerequisite  
 if course has right node  
 print right node as prerequisite**

}

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 course information  
if course has right node  
 print right node as prerequisite course information**

**end Function**

**else**

**if course has left node  
 goto left node  
if course has right node  
 goto right node**

}

Void openFiles() {

**opens files**

**create function with two parameters**

**open file**

**read data**

**parse each line**

**check for course title**

**check for course number**

**if the file is free of errors,**

**check to see if prerequsite exists in course file**

**createCourseObj (create parameters)**

{

**intialize variables for courses, and read file;**

**open the file to read again**

**while file is open**

**store the course obj in a vector data structure**

}  
}

// Menu pseudocode

void menu((sortingMethod)<Course> courses) {

**while user\_input != -1 {**

**switch(user\_input):**

**print “Press 1 to load data structure”**

**print “Press 2 to print course list”**

**print “Press 3 to print course”**

**print “Press 4 to exit”**

**}**

**case:1 user\_input == 1 {**

**load\_data\_structure()**

**}**

**Case:2 user\_input == 2 {**

**printCourseList()**

**}**

**case:3 user\_input == 3 {**

**printCourseInformation()**

**case:4 {**

**user\_input == -1**

**print “Good luck!”**

**}**

// Print course list pseudocode

Void printCourseList(sortedList)

**for all nodes**

**if coursenumber greater than current coursenumber**

**sortedList move course to the right**

**else**

**sortedList move course to the left**

**print sortedList**

## 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) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **open file** | 1 | n | n |
| **read data** | 1 | n | n |
| **parse each line** | 1 | n | n |
| **check for course name** | 1 | n | n |
| **check for course number** | 1 | n | n |
| **create input line** | 1 | n | n |
| **create course object** | 1 | 1 | 1 |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | O(n) |

# Evaluation

Here I will consider the advantages of our data structures. The first one would be vectors. Vectors are one-dimensional and using them for our assignment would work here. We are only opening a file and reading the courses within that file. This type of data structure also uses less memory than the other algorithms. The biggest problem with using this method would be that the elements in our list could not be deleted.

The hash table stores and organizes data in a “dictionary” style. It can be used throughout the project by calling the key. We can call information using those keys, including moving and deleting them. The disadvantage of using the has table for this project is that synchronization increases speed and runtime.

Data trees are the best algorithm for organization and data can be moved into right or left branches. These branches can also be expanded extensively. A program can run searches throughout the entire program. This type of sorting algorithm also increases program runtime and takes longer to modify the tree structures.

# Recommendation

My recommendation for this project would be to use hash tables. I am more comfortable using hash tables personally since I come from a python heavy background, and we use dictionaries all the time. For this program however, a hash table makes sense since you will still need to modify the lists since classes are added and removed all the time. Schools frequently re-number classes as well, and you want the ability to modify your list in this situation. Although the sort time and modification time might increase a little in this case we aren’t dealing with massive databases with millions of nodes where that might begin to truly matter.