# CS 300 Pseudocode Document

David Robey

1/18/22

## 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).

// Course Class object

Private String CourseNum

Private Vector<string> prereqs

Private String courseTitle

//constructor empty default

Class(){

Course num

Course title

New vector

}

//setters

Public setCourseTitle(string title){

Set this object’s courseTitle to title  
}

Public setCourseNum(string courseNum){

Sets this object’s courseNum to CourseNum

}

//adds prereqs to vector prereqs

void addPreReqs(String prereqs) {

append prereq string to end of vector

}

//getters

Public String getCourseTitle(){

Returns course title

}

Public String courseNm(){

Returns course num

}

Void printPrereques(){

For each index in prereq

Print each prereq

}

//Parse file into vector

void parseFile(Vector<Course> courses, string Line)

Create new null course object

String Line = Line

Course object.addTitle this line to first “, “

While(getNext does not equal “”)

course.addPrereq of string to “, “

Add course object into vector

//loadFile

void loadFile(Vector<Course> courses, file fileName){

Try:

Open file with inputFileStream(fileName)

Open fileName

If file is open

While(is good)

Get line and store into string Line

Parse(courses, Line)

Else

Close fileName

Throw exception “can’t read file”

}

// 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**

}

Void partition(vector<courses>, lowIndex, highIndex){

Pick middle element as pivot

while (false)

Increment lowIndex while course[lowIndex] < pivot

Decrement highIndex while pivot < course[highIndex]

If zero or one elements remain, then all numbers are

partitioned. Return highIndex.

else

Swap numbers[lowIndex] and numbers[highIndex]

Update lowIndex and highIndex

return highIndex

}

Void quicksort(vector<course> courses, lowIndex, highIndex){

if lowIndex is greater than or equal to highIndex

return

lowEndIndex becomes Partition(courses, lowIndex, highIndex)

Quicksort(numbers, lowIndex, lowEndIndex)

Quicksort(numbers, lowEndIndex + 1, highIndex)

}

Void printAlphaNumeric(vector<course> courses){

sort using partition and quicksort methods

for each element in courses print course information

}

// Hashtable pseudocode

//each node has a list of pre-reqs

int numPrerequisiteCourses(Hashtable<Course> courses) {

int count = 0

search for class

count = node’s prereqlist.size()

return count

}

void printSampleSchedule(Hashtable<Course> courses) {

search for course

if prereqs is null

print course information

else

printSampleSchedule(last prereq)

}

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

search for course using hash function

compare courseNumber to what was searched

print course information

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

count = 0

if course is not nullptr

count = count + numPrerequisiteCourses of left node

count = count + numPrerequisiteCourses of right node

return count

}

void printSampleSchedule(Tree<Course> courses) {

if course not nullptr

printSampleSchedule of left node

cout course

printSampleSchedule of right node

}

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

while current is not null

if current is same as course number

print course information

return

compare if not less than

traverse left

else

traverse right

return

}

//menu

void main(){

create instance of <course>

while true

print menu options

try 1

create new vector/hash/tree

load file using loadfile()

try 2

print alphanumeric course (binary tree would make this the easiest)

try 3

ask what course number

search for course

if found

print course information of searched course

else print course was not found.

else exit

}

## 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** |
| search for course using hash key | 1 | 1 | 1 |
| compare courseNumber to what was searched | 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** | | | 2n + 3 |
| **Runtime** | | | O(n) |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| while current is not null | 1 | n | n |
| if current is same as course number  print course information  return | 1 | 1 | 2 |
| traverse left | 1 | n | n |
| else | 1 | 1 | 1 |
| Traverse right | 1 | n | n |
| **Total Cost** | | | 2 log n + 2 |
| **Runtime** | | | O(log n) |

The advantages of a vector are the quick appending and popping of nodes within the vector class, it also allows for simple printing of the contents in a O(n). The big disadvantage of this is the sorting. Depending on how the nodes are loaded into vector, it can take the longest out of the three. This is unfortunate because it uses the least amount of memory to store.

The advantages of the binary search tree is the quick sorting of the tree. The tree itself doesn’t need to be sorted to iterate through in low to high or high to low. The disadvantages of this binary search tree is that it takes longer than the other container classes to insert a new item, but it handles the sorting there on the spot which could be considered an advantage. The other disadvantage, though not specified for this assignment is the longer time it takes to remove an item. Especially when its in the middle and not the end of a tree. Again, it also does fix the sorting of the tree as it is done though.

The advantages of the hash table makes it great for searching do it using a key that points to the value. This key is often times something like the ASCII value divided by a number that is relevant which will put that into a hopefully unique location. If its full, you put it in another location. These locations are called buckets. The problem with this method is it can or can not be hard to sort the data. Just going by an ASCII value and thinking it’s sorted can lead to having two items that should be next to eachother on opposite ends of the list. The other disadvantage of this is the amount of memory it uses. It is storing the key, creating a new hash to resize and then rehashing each value into a new key. There are also other disadvantages like the amount of time it takes to implement.

I personally advise the use of binary search tree for this application. The reason I recommend this is because of the design of the binary search tree that will keep the courses in alphabetical order. This means less memory time will be used in swapping and sorting. It may have a little longer search for individual searches of will be O(log n) where the has can have O(n). The memory that the Hash tree uses is a lot more. The vector has the slowest run times when inserting a new node at a O(1), the binary tree uses a O(log n) in regards to insertion. And the hash has a best case O(1) and worst case O(n) depending on the and key generated. Due to the possibility of O(n) Due to the overall stability of the binary search tree where it isn’t the best at any particular category for this instance, but certainly isn’t the worst at any particular instance. I feel this would be the best option to choose.