# 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

}

ABCU Course readFile(File ABCU, lines[])

courseNumber[], courseTitle[], prereq[],line

bool error= true

While (not EOF ABCU)

courseInfo[]=Split (readLine(ABCU, line), delimeter=,)

append line to lines

if(length of courseInfo<2)

error=false

break

courseNumber[i]=courseInfo[0]

courseTitle[i]=courseInfo[0]

|  |
| --- |
| ++ |

if(length of courseInfo>2)

for c=2 to length of courseInfo

prereq[j]=courseInfo[c]

++

If error==true

For each PR in prereq

If PR not in courseNumber

Error=false

ABCU Course

Constructor course(line)

Number=split(line, delimeter=,)>2

Prereq=split(line){2 to length of split)line, delimter=,)]

createObject(Courses<Course>, File ABCU)

Lines[]=””

If readFile(ABCU, Lines)==true

For each line in lines

Add new course(line) to courses

Main()

createObject(Courses, ABCU)

courseNumber=Input()

if courses is empty

cout “No object was found in file”

else

printCourseInfo (courses, courseNumber)

void printSampleSchedule(Vector<Course> courses) {

 for all key, value pair in courses  
       print key course name  
           if value has prereq  
               for each prereq  
                   print prereq

}

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 printalphanumeric(Vecotr<course> courses, string course)

Std:: sort(myvector.begin(), myvector.end(), comparFunction);

Print “Sorted vector”

For(std::vector<std::string>::iterator it=myvector.begin();it!=myvector.end(); ++i);

print new line

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrereq=Hashtable[c]

for each prereq p in totalPrereq

add prereq in Hashtable[p] to totalPrereq

print number of totalPrereq

}

void printSampleSchedule(Hashtable<Course> courses) {

for all key, value pair in courses

print key course name

if value has prereq’s

for each prereq

print prereq

}

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

for all courses

if the course is same as courseNum

print out courseInfo

for each prereq of the Hashtable[course]

print prereq course info

}

Void printAlphanumeric(Hashtable<Course> courses, String course)

Arrays.sort(table.keySet().toArray())

Print hashtable

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrereq=left and right child of Node C

for each prereq p in totalPrereq

add left and right nodes of node p to totalPrereq

print number of totalPrereq

}

void printSampleSchedule(Tree<Course> courses) {

for all nodes as courses

print course name

if course has left node

print left node as preReq

if course has right node

print right node as preReq

}

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

for all nodes

if the course is the same as courseNum

print out all the node’s info

if course has left node

print left node as preReq course info

if course has right node

print right node as preReq course info

break

else

if course has left node

current node is left node

if course has right node

current node is right node

}

void

//Menu pseudocode

While choice is not 0

Print “Menu:”

Print”1. Load course

Print”2. Print course list”

Print”3. Print course”

Print”0. Exit”

Print” enter choice: “

Cin choice

Switch (choice)

Case 1:

dS= new data structure()

loadsCourse(csvPath, dS)

print courses

break

Case 2:

dS-InOrder()

break

Case 3:

for all nodes

if the course is the same as courseNum

print out all the node’s info

if course has left node

print left node as preReq course info

if course has right node

print right node as preReq course info

break

else

if course has left node

current node is left node

if course has right node

current node is right node

Case 4: if choice ==0

Print “Good Bye”

break

int main

std:: vector<std::string> files;

if file is not EOF

files.push\_back(course)

else

break

std::sort(files.begin(), files.end().compareNat)

for (int i=o, i<(int)files.size(); i++)std::cout<<files[i]+”\n”

return

std::hashtable

## 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 | 0(log N) |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Evaluation:

Each of the three data sort types, were used to sort the course info based on their number of prerequisites and course number. In the run time analysis, we were able to see the difference in the various data sorts. When searching for a class numerically it made sense to use a BST because it required the least amount of searching. A BST however is time consuming to add or delete nodes from, due to the reorganization of the entire tree. A hash table is good to sort an unorganized list but harder to search for classes when needed. Hash table collisions and constant table resizing in unavoidable and time consuming. A vector sort works well to organize smaller lists and search them. It’s an easy way to implement a sort algorithm, although the draw backs come from the runtime when adding and searching the vector.

Recommendation:

I would recommend using a Binary search tree algorithm. This would be my recommendation because it allows for the quickest sort of the three. While its downfall come sfrom adding or deleting nodes from the tree; if there is very little modification to the tree after it is created it will allows for the quickest sort and search. The best case runtime for a BST is O(log N) and the worst is O(N) which allows for the quickest search based on the length of the tree and reducing the need to explore every component of the tree itself.