# 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

void openFile(){

get return value for open file

if value is less than 0

Output “File not found”

Else

Open file

}

string loadData(Vector<Course> courses){

Create List courses

Open file using parser libraries

Loop until eof

If first and second string are true

Add the first String to courseID

Add the second String to courseName

Loop until file handler is empty

++I

Concatenate String named preReq for each prerequisite

Add preCount

Add preNames

Return courses

}

searchCourses(Vector<Course> courses, Course c){

for all courses

if courses equals c

return courses

}

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(i=0; I < courses.length, ++i)

output “Course Number: “

output courses[i].courseID

output courses[i].courseName

output “preReqs Number: ”

output courses[i].preCount

output courses[i].preNames

}

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

void openFile(){

get return value for open file

if value is less than 0

Output “File not found”

Else

Open file

}

int loadData(Hashtable<Course> courses){

Create List courses

courses = new hash table with file length

Open file using parser libraries

Loop until eof

If first and second string are true

Add the first String to courses.number

Add the second String to course.name

Loop until file handler is empty

while next line not empty

append courses.preCount

append courses.preName

hashInsert(courses)

Return courses

}

int searchCourses(Hashtable<Course> courses){

key = Hash(courses.number)

node = courses(key)

while node not empty

if node data number equal course number

return node

else

get next node

}

void hashInsert(Hashtable<Course> courses){

Insert a course

Create the key for a given course

Retrieve node using key

If no entry found for the key

Assign this node to the key position

Else if node is not used

Assign old node key unit to UINT\_MAX

Set to key

Set old node to course and old node next to nullptr

Else find the next open node

Add new node to the end

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

for(i=0; I < courses.length, ++i)

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for(i=0; I < courses.length, ++i)

output “Course Number: “

output courses[i].courseID

output courses[i].courseName

output “preReqs Number: ”

output courses[i].preCount

output courses[i].preNames

}

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 course**

**print the prerequisite course information**

}

// Tree pseudocode

void openFile(){

get return value for open file

if value is less than 0

Output “File not found”

Else

Open file

}

courseInsert(tree, node) {

if (tree⇢root is null)

tree⇢root = node

node⇢left = null

node⇢right = null

else

cur = tree⇢root

while (cur is not null)

if (node⇢key < cur⇢key)

if (cur⇢left is null)

cur⇢left = node

cur = null

else

cur = cur⇢left

else

if (cur⇢right is null)

cur⇢right = node

cur = null

else

cur = cur⇢right

node⇢left = null

node⇢right = null

}

int searchCourses(Tree<Course> tree){

cur = tree⇢root

while (cur is not null)

if (key == cur⇢key)

return cur

else if (key < cur⇢key)

cur = cur⇢left

else

cur = cur⇢right

return null

}

}

int numPrerequisiteCourses(Tree<Course> tree) {

for(i=0; I < courses.length, ++i)

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> tree) {

for(i=0; I < courses.length, ++i)

output “Course Number: “

output courses[i].courseID

output courses[i].courseName

output “preReqs Number: ”

output courses[i].preCount

output courses[i].preNames

}

void printCourseInformation(Tree<Course> tree, 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**

}

### Menu:

Main(){

switch

case 1: load data structures

loadData()

case 2: print course list

printCourseInformation()

case 3: Print course

output searchCourses()

case 4: Exit

Break

## 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) |

### Vector:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create List courses | 1 | 1 | 1 |
| Open file using parser libraries | 1 | 1 | 1 |
| Loop until eof | 1 | n | n |
| If first and second string are true | 1 | n | n |
| Add the first String to courseID | 1 | n | n |
| Add the second String to courseName | 1 | n | n |
| Loop until file handler is empty | 1 | n | n |
| ++i | 1 | n | n |
| Concatenate String named preReq for each prerequisite | 1 | n | n |
| Add preCount | 1 | n | n |
| Add preNames | 1 | n | n |
| Return courses | 1 | 1 | 1 |
| **Total Cost** | | | 9n+3 |
| **Runtime** | | | O(n) |

### Hash Table:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create List courses | 1 | 1 | 1 |
| courses = new hash table with file length | 1 | 1 | 1 |
| Open file using parser libraries | 1 | 1 | 1 |
| Loop until eof | 1 | n | n |
| If first and second string are true | 1 | n | n |
| Add the first String to courses.number | 1 | n | n |
| Add the second String to course.name | 1 | n | n |
| while next line not empty | 1 | n | n |
| append courses.preCount | n | n | n |
| append courses.preName | n | n | n |
| hashInsert(courses) | 1 | 1 | 1 |
| Return courses | 1 | 1 | 1 |
| **Total Cost** | | | 7n+n5 |
| **Runtime** | | | O(n) |

### Binary Tree:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| if (tree⇢root is null) | 1 | 1 | 1 |
| tree⇢root = node | 1 | 1 | 1 |
| node⇢left = null | 1 | 1 | 1 |
| node⇢right = null | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| cur = tree⇢root | 1 | 1 | 1 |
| while (cur is not null) | 1 | n | n |
| if (node⇢key < cur⇢key) | 1 | n | n |
| if (cur⇢left is null) | 1 | n | n |
| cur⇢left = node | 1 | 1 | 1 |
| cur = null | 1 | 1 | 1 |
| else | 1 | n | n |
| cur = cur⇢left | 1 | 1 | 1 |
| else | 1 | n | n |
| if (cur⇢right is null) | 1 | 1 | 1 |
| cur⇢right = node | 1 | 1 | 1 |
| cur = null | 1 | 1 | 1 |
| else | 1 | n | n |
| cur = cur⇢right | 1 | 1 | 1 |
| node⇢left = null | 1 | 1 | 1 |
| node⇢right = null | 1 | 1 | 1 |
| **Total Cost** | | | 6n+15 |
| **Runtime** | | | O(log N) |

Analysis

After analyzing the loading of information into each vector, hash table, and binary search tree. The binary search tree remains my favorite method. While the tree will be called more times to load information, it does so more efficiently. During the big O analysis, it’s seen that if you take a value into the tree, it will only be used on one of the 3 possibilities before it is sorted into the tree. This is more efficient and more organized than either the vector or hash table.

With the hash table, the algorithm just keeps loading new information in front of the original information. This is taxing on memory and would be difficult to handle larger databases of information. The hash table is useful with its keys though. Instead of the algorithm searching for entire words, the hash table assigns each data entry a key for it to be found later. This saves time on run time and can be easier on the memory.

Lastly, the vector is hugely inefficient. Its load algorithm is very basic and just loads everything one by one. When it comes time to search inside, there will be many values that return the same information unless you search through the remainder of the entries attached to one data piece. In the application we made, we need to find courses that have prerequisites attached to them. The vector would need to make entries for every course and would have all the prerequisites attached to the back side.

Thus, if I were to recommend an algorithm for the use of the program we worked on, I would go with the binary search tree every time.