# Minunni Project One

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

print course name and number at course name in i

}

}

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) {

int totalPrereq = 0

while next course hash is not null {

find number of prerequisites located in key of hash

add number to totalPrereq }

return totalPrereq

}

void printSampleSchedule(Hashtable<Course> courses) {

while next course hash is not null {

print key of each hash }

}

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

while next course hash is not null {

if courseNum is equal to key value of courses hash{

print course data at that hash } }

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

if(root is not null){

Node\* current = courses->root

numPrerequisiteCourses (current->left)

print out class name and number of prerequisite courses

numPrerequisiteCourses (courses->right)}

}

void printSampleSchedule(Tree<Course> courses) {

if (node != nullptr) {

cout << print out course number, name, and prerequisites

postOrder(node->left);

postOrder(node->right);

}

}

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

Node\* current = root;

while (current is not null) {

if (courseNumber == current course number) {

print current course;

}

else if (courseNumber is larger than current course num) {

current becomes left leaf

}

else {

current becomes right leaf

}

}

}

**Menu Pseudocode**

String choice is input from user

Switch (user) {

Case 1:

Clock start

Load data set

Clock end

Print to user time it took to load and size of data set

Case 2:

Call printSampleSchedule(data set)

Case 3:

Call printCourseInformation(data set)

Case 4:

Print “Good bye.”

Return 0

**Alphanumeric Order Pseudocde**

void sortCourses(Vector<Course> courses){

int min;

int result = courses size

for (int pos = 0 less than result – 1 {

min = pos;

for (int i = pos + 1; i < result; i++) {

if courses at i < courses at min {

min = i } }

if min and pos are no longer equal {

swap values

for length of courses

print courses

}

void sortCourses(Hashtable<Course> courses){

initial is the first hash in the hash table

while next value in courses is not null {

compare values and sort based on results }

}

void sortCourses(Trees<Course> courses, Node node){

if node’s course name is larger than courses’ {

if there is no left subtree {

insert node to left side

else {

recurse node to compare to left subtree } }

else {

if there is no right subtree {

insert node to right side

else {

recurse node to compare to right subtree } }

call inOrder function to print classes in order

}

## Runtime Analysis

| **Code (Vector)** | **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 (Hash map)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **While next course is not null** | 1 | n | n |
| **If courseNum is equal to key value of hash** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

| **Code (Tree)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **While current is not null** | 1 | Log(n) | Log(n) |
| **If courseNum is equal to current’s courseNum** | 1 | Log(n) | Log(n) |
| **print out the course information** | 1 | 1 | 1 |
| **Else if current courseNum is smaller than courseNum** | 1 | Log(n) | Log(n) |
| **Current becomes left node** | 1 | Log(n) | Log(n) |
| **Total Cost** | | | 4(log(n)) + 1 |
| **Runtime** | | | O(log(n)) |

**Advantages and Disadvantages**

The advantage of using a vector as the chosen data structure is that it is the simplest structure to work with. The major disadvantages are that it is much less efficient than the other two structures and is not optimal to use for more complex data or larger data sets. An advantage of using a hash map is that it is more efficient due to its use of keys to represent values. It is more complicated to code compared to vectors, however, and while it can process faster, a hash map at its worst case can have the same O notation as a vector. Trees have the advantage of being logarithmic O time and can be more efficient than the other two structures. A disadvantage of a tree is that if data values are already in order before being put into a tree, it will become a larger tree and take more resources to use the tree because of that.

**Recommendation**

After looking at all three data structures, I think working with trees would be the best option. Trees are one of the quicker data structures to work with and are easier to implement than a hash map while also taking advantage of a linked list system.