# 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

}

void printSampleSchedule(Vector<Course> courses) {

for all courses

for all data in course information

print the course information

}

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**

}

int partition(Vector<Course> courses, int begin, int end) {

low = begin

high = end

get mid

for all courses

while low is less than mid

++low

while mid is less than high

--high

if low >= high

return

else

swap low with high

++low

--high

}

void quickSort(Vector<Course> courses, int begin, int end) {

unsigned int mid = 0

if begin is greater than or equal to end

return

mid = partition courses from beginning to end

quickSort(courses, begin, mid)

quickSort(courses, mid+1, end)

}

void printAll(Vector<Course> courses) {

for all courses

print the course information

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

size = size of courses

totalPrerequisites = 0

key = courseNum

bucket = hash(course.key)

for all buckets in HashTable

if courses[bucket] is not empty and courses[bucket]->key==key

add prerequisites of p to totalPrerequisites

bucket = (bucket + 1) % size

return totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

key = courseNum

bucket = hash(course.key)

for all courses

if courses[bucket]->data is empty

bucketList = HashTable(hash item->key)

node = the new node

node->next = empty

node->data = courseNum

Append(bucketList, node) to list

else

return data at node

get user input course

for all courses

if the course at node->data has no collisions

print the courses[bucket]->data

else if the course at node->data has collisions

while (node->data is not null)

print node->data

node->next

else

go to next bucket

}

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

hashVal = courseNumber % key

for all courses

if bucket[courses] is the same as courseNumber

create new pointer node at head

print node->data

for each prerequisite of the course

print the prerequisite course information

}

int partition(Hashtable<Course> courses, int begin, int end) {

low = begin

high = end

get mid

node = new node at key

while node is not null

for all courses

while low is less than mid

++low

while mid is less than high

--high

if low >= high

return

else

swap low with high

++low

--high

}

void quickSort(Hashtable<Course> courses, int begin, int end) {

unsigned int mid = 0

if begin is greater than or equal to end

return

mid = partition courses from beginning to end

quickSort(courses, begin, mid)

quickSort(courses, mid+1, end)

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses)

node = root of tree

totalPrerequisites = 0

while node is not null

for all prerequisites p in totalPrerequisites

if prerequisites in node->data == 0

return

else if prerequisites in node->data > 0

add p to totalPrerequisites

if node->right is not null

add p to totalPrerequisites

traverse right

else if node->left is not null

add p to totalPrerequisites

traverse left

return totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

node = root of tree

while node is not null

if key = node->key

return data at node + new line

else if key < node->key

traverse left

else

traverse right

}

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

curr = root of Tree

key = courseNumber

while curr is not null

if key = curr->key

if totalPrerequisites > 0

print course information for curr

print prerequisites

else

print course information for curr

else if key < curr->key

traverse left

else

traverse right

return

}

void printInOrder(Tree<Course> courses) {

new node at root

if node is null

return

printInOrder(node->left)

print node

printInOrder(node->right)

}

// Load courses and display menu pseudocode

void loadCourses(csvPath) {

initialize parser file

vector header = get header

for every line in file:

Course course

course id = course[i][0]

course title = course[i][1]

course preRequisites = course[i][2]

insert course into data structure

}

int main(int argc, char\* argv[]) {

string csvPath, courseKey

switch(argc):

case 2:

csvpath = argv[1]

courseKey = key

break

case 3:

csvPath = argv[1]

bidKey = argv[2]

break  
 default:

csvPath = path of courses file

courseKey = key

while choice != 9:

cout options to user

case 1:

initialize data structure (courses)

loadBids(csvPath, courses)

case 2:

printSampleSchedule(courses)

case 3:

printCourseInformation(courses)

}

## 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.

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | 4 | 4 |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | 3 | 3 |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

| **HashTable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | 4 | 4 |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | 3 | 3 |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | 5 | 5 |
| **if the course is the same as courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | 2 | 2 |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

After analyzing the efficiency of the three data structures, they share the same common characteristics but trees and hash tables are more efficient overall. They can easily handle large amounts of data which includes organizing and searching for data within the tree or hash table at efficient speeds. The vector data structure is not as fast or efficient as the tree and hash table data structures when it comes to a large number of elements but in this case, there are only eight entries to be analyzed so a vector would be a good choice in this situation. Overall, they are all strong data structures to use in this situation but the hash table is a desirable method to sort data for this program due to a small number of elements and integer key values that can be quickly sorted and printed due to hash tables efficient speeds. If there were more elements to place within the data structure, a tree would be more efficient but in this case, a hash table would perform better than the tree and vector.