# 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) {

DECLARE local vector of Courses => schedule

DECLARE placeholder index of 0 => i=0

FOR schedule length is less then 4

FOR starting at index loop through vector

IF courses at index does not contain prereqs

ADD course to schedule

INCREMENT i

PRINT schedule

}

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

**for all course**

**s**

**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) {

DECLARE vector of Courses => currCourses

FOREACH Course in courses

IF course doesn’t contains preReqs

CONTINUE

FOREACH Course in currCourses.prereqs

If currCourses contains course prereq

continue

Else

ADD current prereq to currCourses

Return currCourses amount => currCourses.Size()

}

void printSampleSchedule(Hashtable<Course> courses) {

DECLARE local Hashtable of Courses => schedule

DECLARE placeholder index of 0 => i=0

WHILE schedule size is less then 4

FOR starting at index loop through courses

IF courses at index does not contain prereqs

ADD course to schedule

INCREMENT i

PRINT schedule

}

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

DECLARE key value based on search string

IF courses at key is not nullPtr and equals string input

PRINT all data in course number

ELSE IF courses at key is not nullPtr

WHILE course isn’t nullptr

IF course key isnt UINT\_MAX and matches search string

PRINT all data in course number

ELSE

SET course equal to next course => course = course->next

ELSE

PRINT class could not be found in dataset

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

DECLARE vector currCourses to hold return values

FOR EACH course in courses

FOREACH coursePrereq in course

IF currCourses contains coursePrereq

add coursePreReq to currCourses

RETURN numCourses size

}

void printSampleSchedule(Tree<Course> courses) {

DECLARE local vector of Courses => schedule

DECLARE placeholder index of 0 => i=0

WHILE schedule length is less then 4

FOR starting at index loop through courses

IF courses at index does not contain prereqs

ADD course to schedule

INCREMENT i

PRINT schedule

}

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

DECLARE Key key for courseNumber

DECLARE pointer node at node.at(key)

IF node equals nullptr or uINT\_MAX

PRINT ERROR message

Return

WHILE Node doesn’t equal nullptr

IF doesn’t equal max and node at courseNumeber equals courseNumber

PRINT node.courseNumber data

Return

SET node equal to nextNode in dataset

}

Menu and alphanumeric order pseudocode:

Void displayMainMenu(){

DECLARE int for user input and initialize with 0

DECLARE bool done and set to false

WHILE done == false

PRINT menuoptions for userInput

GET userInput

IF userInput is not numeric || userInput is less than 0

PRINT ERROR message

CONTINUE

CASE userInput == 1

CALL load dataStructure

CALL sort and pass in data read

CASE userInput == 2

PRINT CourseList

CASE userInput == 3

GET searchInput

searchCourses(searchInput)

CASE userInput == 9

SET done = True

DEFAULT

PRINT ERROR message

CONTINUE

PRINT goodbye message

EXIT Program

}

/\*

I would write the program to sort the courses when they are read into the program for more efficient searches and printing. Assuming there is no reason to leave them unsorted when reading them into the program, This would be a more efficient way to access the data.

\*/

Node\* insert(Course course){

IF root == nullptr

RETURN new Node(course)

IF course number(key) is less then root->courseNumber

SET Root->left = insert(root->left course)

ELSE

SET root->right = insert(root->right)

Return root

}

SortData(Vector(Courses) data)){

FOREACH item in data

SET root = insert(course at data index )

}

printData(Node\* node){

IF node == nullptr

RETURN

printData(node->left)

PRINT cout course data

printData(node->right)

}

## Example Runtime Analysis and Recommendation

The advisors at ABCU have been working with us to develop program that will read their data into a program and structure the data into a useable form. We have been investigating the advantages and disadvantages with each datatype in relation to their particular problem and usage. Starting with Vectors, we know that the data would be held in contiguous memory. This allows us to utilize and cache efficiently. Additionally, we can access elements by their index which is more efficient than traversing nodes to look for data. One drawback with Vectors in their inefficiency with insertion and deletion. Vectors are also not very efficient for memory when working with large datasets. With ABCU I think we could potentially have issues at scale with Vectors.

Similar to Vectors, Hashtables have fast access to data within the table and are dynamically sized. Hashtables also based on key value pairs which can add easier to understand structure to the dataset. Hashtables can have issues when it comes to collisions in their keys. This can slow down operations and require additional resources to hold data within the table. Another issue with hashtables is that they are not ordered as efficiently as other data types and cause searching for values to be less efficient than other data types. Depending on the operation within our program hashtables could be a good choice. However, since one of the primary tasks of our program will be searching for and displaying data, I believe Trees will be a better option.

Trees have the fastest searching methods when compared to other data types. Trees are also the most memory efficient since they are not stored in contiguous memory, they do not have empty buckets similar to hashtables. Additionally, Trees are ordered which will allow us to display course with better readability to our users. One issue that Trees can have occurs when inserting and deleting elements. There are additionally computations that are required in a tree that are not present in a hashtable. With our program this is a minor issue since currently we will not need to alter the data after it is read into the program. With the current program that has been defined I believe that the tree will be the best option for ABCU’s data to be worked with inside the program.

Vectors:

printCourseInformation(Vector<Course> courses, String courseNumber)

| **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) |

numPrerequisiteCourses (Vector<Course> courses, Course c)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Declare placeholder vector to return and set equal to prereqs of Course c** | 1 | 1 | 1 |
| **Foreach prereq p in total preqs** | 1 | n | N |
| **Add prereq of p to totalPrereqs** | 1 | n | n |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

PrintSampleSchedule (Vector<Course> courses)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Declare vector of Courses** | 1 | 1 | 1 |
| **Declare index int of 0** | 1 | 1 | 1 |
| **FOR schedule is less then 4** | 1 | 4 | 4 |
| **for each prerequisite of the course** | 1 | n | n |
| **IF the course at the index does not contain prereqs** | 1 | n | n |
| **Add course to schedule** | 1 | 4 | 4 |
| **Increment i** | 1 | n | n |
| **PRINT schedule of local Courses that were added.** | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 11 |
| **Runtime** | | | O(n) |

HashTables:

numPrerequisiteCourses(Hashtable<Course> courses)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **DECLARE vector of Courses** | 1 | 1 | 1 |
| **FOREACH Course in courses** | 1 | n | n |
| IF course doesn’t contain prereqs | 1 | 1 | 1 |
| **Continue** | 1 | 1 | 1 |
| **FOREACH Course in currCourses.prereqs** | 1 | n | n |
| **If currCourses contains course prereq** | 1 | 1 | 1 |
| **Continue** | 1 | 1 | 1 |
| **Else** | 1 | -1 | -1 |
| **ADD current prereq to currCourses** | 1 | 1 | 1 |
| **Return currCourses amount => currCourses.Size()** | 1 | 1 | 1 |
| **Total Cost** | | | N^2 + 6 |
| **Runtime** | | | O(n^2) |

printSampleSchedule(Hashtable<Course> courses)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **DECLARE local hashtable** | 1 | 1 | 1 |
| DECLARE index of i | 1 | 1 | 1 |
| WHILE schedule size is less then 4 | 1 | 4 | 4 |
| FOR courses starting at index | 1 | N | N |
| IF courses at index does not contain prereqs | 1 | n | N |
| Add course to schedule | 1 | 4 | 4 |
| INCREMENT i | 1 | n | N |
| PRINT schedule | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 11 |
| **Runtime** | | | O(n) |

printCourseInformation (Hashtable<Course> courses, String courseNumber)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **DECLARE key of search string** | 1 | 1 | 1 |
| **IF courses at key is not nullPtr and equals string input** | 1 | 1 | 1 |
| **PRINT all data in course number** | 1 | 1 | 1 |
| **ELSE IF courses at key is not nullPtr** | 1 | -1 | -1 |
| **WHILE course isn’t nullptr** | 1 | n | N |
| **IF course key isnt UINT\_MAX and matches search string** | 1 | 1 | 1 |
| **PRINT all data in course number** | 1 | 1 | 1 |
| **Else** | 1 | -1 | -1 |
| **SET course equal to next course => course = course->next** | 1 | n | N |
| **ELSE** | 1 | -1 | -1 |
| **PRINT class could not be found in dataset** | 1 | 1 | 1 |
| **Total Cost** | | | 2n+2 |
| **Runtime** | | | O(n) |

Trees:

numPrerequisiteCourses(Tree<Course> courses)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE vector currCourses to hold return values | 1 | 1 | 1 |
| FOR EACH course in courses | 1 | N | N |
| FOREACH coursePrereq in course | 1 | N | n |
| IF currCourses contains coursePrereq | 1 | 1 | 1 |
| add coursePreReq to currCourses | 1 | 1 | 1 |
| RETURN numCourses size | 1 | 1 | 1 |
| **Total Cost** | | | N^2 + 4 |
| **Runtime** | | | O(n^2) |

printSampleSchedule(Tree<Course> courses)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE local vector of Courses => schedule | 1 | 1 | 1 |
| DECLARE placeholder index of 0 => i=0 | 1 | 1 | 1 |
| WHILE schedule length is less then 4 | 1 | 4 | 4 |
| FOR starting at index loop through courses | 1 | N | N |
| IF courses at index does not contain prereqs | 1 | n | N |
| ADD course to schedule | 1 | 4 | 4 |
| INCREMENT i | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 11 |
| **Runtime** | | | O(n) |

printCourseInformation(Tree<Course> courses, String courseNumber)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE Key key for courseNumber | 1 | 1 | 1 |
| DECLARE pointer node at node.at(key) | 1 | 1 | 1 |
| IF node equals nullptr or uINT\_MAX | 1 | 1 | 1 |
| PRINT ERROR message | 1 | 1 | 1 |
| Return | 1 | 1 | 1 |
| WHILE Node doesn’t equal nullptr | 1 | n | N |
| IF doesn’t equal max and node at courseNumeber equals courseNumber | 1 | N | N |
| PRINT node.courseNumber data | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| SET node equal to nextNode in dataset | 1 | n | N |
|  |  |  |  |
| **Total Cost** | | | 3n + 7 |
| **Runtime** | | | O(n) |