# 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 Structure Creation

Constructor Structure/Data for courseInfo (courseId, courseTitle, coursePrereq1, coursePrereq2)

String courseId

String courseTitle

String coursePrereq1

String coursePrereq2

Vector <Course> CourseList Construct - Empty

ReadFileAndAddtoVector():

String courseId  
 String courseTitle

String coursePrereq1  
 String coursePrereq2

String temp

Open File

While(readChar is not NULL)

While (readChar not /n)

While (readChar not “,”)

If courseTitle != empty && courseID != empty

Append readChar to temp;

If (temp has “ “)

courseTitle = temp

else

courseId = temp

temp = “”

else

While (readChar not “,”)

Append readChar to temp;

coursePrereq1 = temp

temp = “”

if (readChar is “/n”)

coursePrerqe2 = “ “  
New CourseList(courseId, courseTitle, coursePrereq1, coursePrereq2)

courseID = “”  
 courseTitle = “”

coursePrereq1 = “”

coursePrereq2 = “”

Close File

// 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 Loop – (I = 0, I < courseList.size(), ++i

Print courseId | courseTitle | coursePrereq1 | coursePrereq2

}

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 Structure Creation

Hashtable Class

Create Structure for Nodes to hold courseInfo Objects

Multiple Constructors allowed for creating Nodes

Create Vector to track node structures

Hashtable Creation

Create empty nodes into vector using size of current CourseList

Hashtable Logic

Return key % size of the vector

Hashtable – Place Into Table

Iterate through Vector of Course List

Create key based on logic and using the courseId as the input

Using result of logic, place node object with current course information into appropriate space on the table matching the key.

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

//Similar to Vector Search of All Courses that exists.

//All items need to be iterated through in order to count all prereqs.

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) {

Iterate through vector of Nodes from front to end

while node place is less than the size of the vector

if node is not null

print node details

if node is empty

skip

}

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

Take courseNumber and put through Hashtable Logic

Return Key

Search key in Hashtable List of Courses

If arrayspace is empty

Return “Not Found”

Else

If arrayspace == courseNumber

Return Course ID, Course Title, Course Prereqs

Else arrayspace does not match course Number

Move to next array space

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

initialize totalPrerequisites

if node is empty

quit looking at current node

Call this method again and start at the left node of the current one.

Check the numbers of prereqs in the current node and add it to the totalprereqs

Call this method again on the node to the right of the current one.

print totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

if node is not null

return

call this method and look at the course in the left node.

print the current course information such as the Course ID, the course title, and the course prerequisites.

Call this method and look at the course on the right of the current one.

}

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

While the coursnumber is not null

compare the course number to the current course

if search course is smaller than current course

move to the left node

if search course is larger than the current course

move to the right node

if search course is a match to the current course

return Course ID, Course Title, Prerequisite #1 and

prerequisite #2.

If course number does come up null

Print a message saying the Course ID entered was not

found.

}

//Menu for ABCU Advising Program

void mainMenu {

create a variable user input

While loop as long as userInput is not 4. {

Print the menu as follow:

1. Load Data Structures
2. Print Ordered Course List by Course ID
3. Print Course Pre-Requisites for Specific Course Title
4. Exit

receive input and assign to userInput

Using Switch Method here to return proper information.

Case 1:

Call ReadFileAndAddtoVector() // This will change based on the data structure best suited for this information.

Break;

Case 2:

Call sortMethod() // Examples of Sorting Methods will be below in the next part of the project.

Break;

Case 3:

Use the printCourseInformation for whichever data structure is chosen as the best and most efficient one for this project.

Break;

Case 4:

Break; // loop should end here.

}

}

//Sorting the Different Data Structures

//Vector Structures

In module two, used the quicksort method.

void quickSort(vector<Bid>& bids, int begin, int end) {

set mid equal to 0

Base case: If there are 1 or zero bids to sort,

partition is already sorted otherwise if begin is greater

than or equal to end then return

Partition bids into low and high such that

midpoint is location of last element in low

recursively sort low partition (begin to mid)

recursively sort high partition (mid+1 to end)

}

//Hashtable Structure Sort

Sorting method has to be taken into account during the insert portion. For ordered lists here, the best approach may be chaining to by specific course numbers such as 100s, 200s, 300s and so on.

//Binary Structure Sort

This item is ordered by CourseID when inserted into list.

EVALUATION

## Vector Analysis

| **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) |

## Hashtable Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n^2 | n^2 |
| **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** | | | n^2+3n+1 |
| **Runtime** | | | O(n^2) |

## Tree Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 2 | Log(n) | Log(n) + 2 |
| **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** | | | Log(n)\*n + 3 |
| **Runtime** | | | O(n) |

Analysis

There are some distinct advantages and disadvantages between the different data types that have been referenced above. We will go through each of the different data structures to talk about each one.

Starting with the Vector Data Structure. The vector data structure can easily be manipulated by adding, removing, or sorting. Each of these functions does have to be done separately though and each time a new item is added, a sort will have to be completed again to make sure they are in the correct order. The vector structure itself is not self-maintained and will require that the programmer is always allowing functions to be available for proper maintenance by the administration or whoever will be accessing and appending the data. When printing, the data, it is straight forward as all the data is sorted ahead of being printed. Searching can also be quite a bit long since if the search is for the last item, it would require going through each of the items all the way through.

The hash table structure has some very good advantages and can technically be one of the fastest in terms of searching for specific course information. When searching for a specific course, it can be easily pointed to the nearest location of that course ID and be found very quickly. The downside, this method of data structure must be very detailed in the logic being used to point these items to their assigned area in the array. For this reason of relying on the key and logic formula, storing this data in order can be very difficult and make it harder to sort depending on the need for being in alphanumerical order. With what was previously discussed earlier, although the hash tables can be the fasted for search for specific data, they can be a much longer time when having to go through each item to work with each object.

The hanging tree or binary tree structure can be very useful for this type of data. In this project, the tree can be self-maintained as items are sorted when they are entered into the data structure as they are sorted by how the programmer identifies it is going to be easiest, in this case the course ID. Searching the trees for a specific course will be faster than the vector since the trees will cut the search time in half by only searching from the middle and out but is still slower than the hash tables. Being able to print items in order is an easy task to do than it would be in the hash tables. Another advantage of hanging trees is that they do not take more space than is necessary since they only need space for items that are currently in the tree. Another disadvantage of hash tables which may allocate space that may not be used depending on the logic of the keys.

Taking all of these into considerations, I have made the following recommendation based on our needs for this project:

For this project, I’m recommending using the Binary Tree Data Structures that will allow for some self-maintenance of the structure and keep as close to in order as possible from insertion of data. We will be sacrificing some search speeds that would be stronger in the hash tables but in the overall view of the project, the Binary Tree System will be faster than the vector data structures and take out the complexity of the hash table data structure.