# 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).

// Menu pseudocode

//Converting the initial char to an int and making the user choice a string prevents erroneous data

string choice = "0";

int userChoice = choice [0] - '0';

**while (userChoice != 9) {**

**cout " 1. Load Data Structure" endl;**

**cout " 2. Print Course List" endl;**

**cout " 3. Print Course" endl;**

**cout " 9. Exit" endl;**

// loop will break once user enters 9

// 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  
 print course name  
 if course has prerequisits  
 for each prereuisit  
 print prerequisit**

}

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) {

**totalPrerequisites = Hashtable[c]  
for each prerequisite p in totalPrerequisites  
 add prerequisites in Hashtable[p] to totalPrerequisites  
 print number of totalPrerequisites**

}

void printSampleSchedule(Hashtable<Course> courses) {

**for all key, value pair in courses  
 print key course name  
 if value has prerequisits  
 for each prerequisits  
 print prerequisits**

}

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 Hashtable[course]**

**print the prerequisite course information**

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

**totalPrerequisites = left and right child of Node c  
 for each prerequisite p in totalPrerequisites  
 add left and right Nodes of node p to totalPrerequisites  
print number of totalPrerequisites**

}

void printSampleSchedule(Tree<Course> courses) {

**for all Nodes as courses  
 print course name  
 if course has left node  
 print left node as prerequisit  
 if course has right node  
 print right node as prerequisit**

}

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

**for all Nodes  
 if the course is the same as courseNumber  
 print out the node's information  
 if course has left node  
 print left node as prerequisite course information  
 if course has right node  
 print right node as prerequisite course information**

**end Function**

**else**

**if course has left node  
 goto left node  
 if course has right node  
 goto right node**

}

/\*\*  
\* Utilizing the public Insert method  
\*  
\* Determining & setting correct alphanumeric location & if the current child is a nullptr,  
\* adds the node to the BST, else keeps traversing BST until a nullptr is found  
\*/ **void CourseBST::addNode(Node\* node, Course course) {**

// Current courseNum is less than the current node's courseNum

**if (node->course.courseNum.compare(course.courseNum) > 0) {**

**if (node->left == nullptr)**

**node->left = new Node(course);**

**else**

**this->addNode(node->left, course);**

**}**

// Current courseNum is equal or greater than the courseNum of current node  
**else {**

**if (node->right == nullptr)**

**node->right = new Node(course);**

**else**

**this->addNode(node->right, course);**

**}**

**}**

/\*\*  
\* Recursively prints the loaded courses in-order  
\*/

**void CourseBST::printSampleSchedule(Node\* node) {**

**if (node != nullptr){**

**printSampleSchedule(node->left);**

**cout << node->course.courseNum << ", " <<node**

**>course.courseName << endl;**

**printSampleSchedule(node->right);**

**}**

**return;  
}**

## 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) |

**Evaluation**

*Vectors:*

Would generally not be the ideal solution because though it has the ability to store lists of items (ordered collection of elements) such that you do not need to search through the list for a specific item and it takes less space, you will however need to search the entire vector for any key value. Sorted vector as would be the case in our course list would not be best when it comes to insertions/deletions but per the advisor’s requirements this could have an edge over the other two data structures.

*Hash table:*

In my opinion, this would also not be the Ideal solution per the instructor’s requirements because it is perfect for retrieval and storing key-value pairs and access to any key-value and it is relatively faster when searching by key, which unlike vectors will require an entire search of the vector. Hash maps are simply buckets (array) with methods to deal with collisions and have data size limitations and leads on performance O(log n) but all of these will have to rely on small records and database libraries.

*Tree (BST):*

This would be the best solution because keys can be sorted by just doing Inorder Traversal of trees that Hash Tables would naturally not be able to do. In addition, trees (BSTs) seem much easier to implement compared to hashing. Implementing customized BST is easy whereas with hashing you will have to rely on libraries provided by programming languages. Sorted vector would be same as BST in terms of performance.

*Choice:*

Though not the most ideal solution, I think I will plan to use Vector in my code just for the sake of efficiency, orderliness (ordered list) and the fact that indices are known which will help the retrieval quicker.