# 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) {

}

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) {

}

void printSampleSchedule(Hashtable<Course> courses) {

}

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

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

}

void printSampleSchedule(Tree<Course> courses) {

}

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

}

## 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) |

**Binary Search Tree**

Course Binary search tree{

Private:  
 Node\* root;  
 void addNode(Node\* node, Course course);

Void printSampleSchedule(Node\* node);  
 void printCourseInformation(Node\* node, string courseNum);

public:  
 CourseBinarySearch();  
 virtual ~ CourseBinarySearch();

void insert(Course course);

int numPrerequisiteCourses(Course course);

void printSampleSchedule();

void printCourseInformation(string courseNum);

}

Void CourseBinarySearch:: printSampleSchedule(Node\* node){

If(node != nullptr){

printSampleSchedule(node->left);

cout << node->course.courseNum << “ “ << node->course.courseName << endl;

printSampleSchedule(node->right);

}

Return;

}

Void CourseBinarySearch:: printCourseInformation(Node\* curr, string courseNum){

While (curr != nullptr){

If(curr->course.courseNum.compare(courseNum) == 0){

Cout<< curr->course.courseNum <<” “ <<curr->course.courseName << endl;  
  
unsigned int size = numPrerequisiteCourses(curr->course);

Cout << “Prerequisite: “;

Unsigned int I = 0;

For(I = 0; I < size; i++){

Cout << curr->course. Prerequisites.at(i);

If(i==0){

Cout << ”No prerequisites required.” << endl;

Return;

}

Else if (course.Num.compare(curr->course.courseNum) <0){

Curr = curr->left;

}

Else{

Curr = curr->right;

}

}

}

}

**Hash Table**

Class HashTable {

Private:  
struct Node

Course\* course  
unsigned int Key  
Node\* next  
  
Node()  
key = UINT\_MAX

Next = Null pointer

Node(Course aCourse)  
Course = aCourse

Vector<Node> = nodes

Unsigned int table size = DEFAULT\_SIZE

Unsigned int hash(int key)

Public:  
HashTable()

HashTable(unsigned int size)

Virtual ~HashTable()

int numPrerequisiteCourses(Hashtable<Course> courses)

Void PrintSampleSchedule(HashTable<Course> courses)

Void printCourseInformation(Hashtable<Course> courses, String courseNumber)  
  
**Adding courses to the Hash Table**

unsigned key = hash(atoi(course.Num.c\_str()));Create a key for the course   
Node\* oldNode = &(nodes.at(key));create a node to retrieve the node using a key  
  
if (oldNode == nullptr)if the node is equal to null pointer  
Node\* newNode = new Node(course, key); create a new node new course with a course and a key  
nodes.insert(nodes.begin() + key, (\*newNode)); insert the new course contents into the hash table at the position of the key  
  
else if (oldNode->key == UINT\_MAX) else if nodes key is equal to UINT\_MAX  
oldNode->key = key;  
oldNode->course = course;  
oldNode->next = nullptr;

Else   
while (oldNode->next != nullptr) {while oldNode->next is not equal to null pointer  
oldNode = oldNode->next; create a new node new Course with acourse and key  
}

oldNode->next = new Node(course, key); oldNode->next to newCourse

**Printing the course info**for (auto i = nodes.begin(); i != nodes.end(); i++) for every node in the table  
if (i->key != UINT\_MAX) if the node’s key is not equal to UINT\_MAX  
cout <<"key "<< i->key<<": " << i->course.Name << " | " << i->course.num << endl; output the node’s key  
print the node’s course  
Node\* node = i->next; make the node equal to the next iter  
while (node != nullptr) while node does not equal null pointer  
cout << "key " << node-> key << ": " << node->course.Name << i->course.Num << endl; output the node’s key

Print the listNode’s course  
node = node->next; set the node to equal node->next

**Removing a Course**unsigned key = hash(atoi(CourseNum.c\_str()));// set key equal to hash atoi CourseNum cstring

nodes.erase(nodes.begin() + key);// erase node begin and key

**Searching the hashtable for a course and if it is found return the course if not return nothing**

unsigned key = hash(atoi(course.courseNum.c\_str()));// create the key for the given course Name

Node\* node = &(nodes.at(key));

if (node != nullptr && node->key == UINT\_MAX) {// if entry found for the key If the node is not equal to null pointer and node->key equal to UINT\_MAX  
return node->course //return node course  
  
if (node == nullptr || node->key == UINT\_MAX) if the node is equal to null pointer or node->key equal to UINT\_MAX  
return course; return course  
  
while (node != nullptr) while the node does not equal null pointer  
if (node->key != UINT\_MAX && node->course.courseNum.compare(courseNum) == 0)if the current node matches return node->course; return it

node = node->next; point the list node to the next list node

return course; Return an empty object

**Vector   
Defining a Course**  
struct course  
string courseNum, courseName;

**Displaying Courses**

Void displayCourse(Course course)

cout << course. courseName << ": " << course.courseNum << endl;

Return

**Getting Courses**Course getCourse()

Cout << “Enter course Name: “

Cin.ignore()

Getline(cin, course. courseName)

cout << "Enter course Number: ";

getline(cin, course.courseNum);

return course;

**Loading Courses**

Vector<Course> loadCourse(string csvPath){

Vector<Course> course; Defining the vector

Csv::Parser file = csv::Parser(csvPath) Start the csv parser with the given file path

Make a try catch loop that will continue to read the file until there are no rows left  
try{

For(int I = 0;l I <file.rowCount(); i++){

Create the data structure for the course

Course course;

course.courseNum = file[i][1];

course. courseName = file[i][0];

push the course back to the end  
courses.push\_back(course);

}

}  
catch (csv::Error& e) {

Std::cerr << e.what() << sted::endl;

}

Return courses;

}

**Removing specific courses**

removeCourse();{

cout << “Enter the Course Number to be removed: ”

cin >> courseNum;

vector.erase(courseNum);

}

**Defining the int main and related functions**

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

**Choosing the file**

Switch (argc){

Case 2:

csvPath = argv[1];

break;

default:

csvPath = “class input file";

}

**Defining the vector that holds all of the course**

Vector<Course> courses;

**Defining the timer variable**

Clock\_t ticks;

**Making a working Menu**

Int choice = 0;

While (choice != 9){

cout << "Menu:" << endl;

cout << " 1. Load Courses" << endl;

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

cout << " 3. Course Information" << endl;

cout << " 9. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

}

switch(choice){

case 1:

ticks = clock(); make the timer variable before loading the courses

courses = loadCourses(csvPath); call the load course function

cout << courses.size() << “ courses read” << endl;

calculating the elapsed time

ticks = clock() – ticks;

cout << “time: “ << ticks << “ clock ticks” << endl;

cout << “time: “ << ticks \* 1.0 / CLOCKS\_PER\_SEC << “ seconds” << endl;

break;

case 2:

loop to display all of the courses that were read  
 for(int I = 0; I < courses.size(); i++){

displayCourse(courses [i]);

}

Cout << endl;

Break;

Case 3:

cout << "What course do you want to know about? " << endl;

cin >> courseNum;  
std::transform(courseNum.begin(), courseNum.end(), courseNum.begin(), ::toupper ); converts all lowercase letters to upper case

course->Search(courseNum);

break;

}

}  
cout << “Good bye.” << endl;

Return 0;

}