# 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

Void loadfile(string fileName) {

OPEN file

IF file is opened correctly

​ SET contents equal to the open file

​ IF contents is null

​​ PRINT error message

​​ RETURN

​ SET a new vector called cInfo to hold course info

​ WHILE not at the end of the file

​​READ the data by parsing each line and separate variables by commas

​​ IF cNumber AND cTitle equal null

​​​PRINT an error message “Some information is missing”

​​​ RETURN

​​ ELSE

CREATE NEW course object temp with cNumber, cTitle, preq1, and preq2

​​​ ADD temp to courseList​

​​ ADD cNumber to cInfo

​ SET current node to head

​ SET a boolean isHere to false

​FOR LOOP where I is less than the length of courseList; i++

​​IF  SEARCH (courseList for cInfo[i]) does not equal null

​​​ SET isHere to true

​ IF isHere is false

​​PRINT an error message stating that some prerequisites may not be included​​

}

Class Course {

Int courseNumber

String Title

Int prerequisite1

Int prerequisite2

}

Bool openFile(sting fileName) {

Bool isFine equals true

​SET a new vector called cInfo to hold course info

​ WHILE not at the end of the file

​​ READ the data by parsing each line

​​ SET a STRING cNumber to hold course number parsed

​​ SET a STRING cTitle to hold course title parsed

​​ SET a STRING preq1 to hold course prerequisites parsed

​​ SET a STRING preq2 to hold course prerequisites parsed

​​CREATE NEW course object temp with cNumber, cTitle, preq1, and preq2

​​ ADD cNumber to cInfo

IF cInfo doesn’t equal null

ADD cInfo to courses linked list

ELSE

isFine equals false

RETURN isFine

}

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 addCourse(Course course, Vector<courses> courses) {

Course.push\_back(course)

}

void printSampleSchedule(Vector<Course> courses) {

sort(courses.begin(), courses.end())

**for each course in courses**

**print out 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**

}

// Hashtable pseudocode

class Hashtable {

struct Node {

Course course

Node\* next

Unsigned int key

}

Unsigned int Default size

Vector<Node> table

}

Int Hashtable<Course> Hash(string courseNumber){

RETURN atoi(courseNumber.c\_str()) % size

}

bool loadFile(string filename) {

bool isFine = true

OPEN file

IF file is opened correctly

​ SET contents equal to the open file

​ IF contents is null

​​ PRINT error message

​​ RETURN

​ SET a new vector called cInfo to hold course info

​ WHILE not at the end of the file

​​READ the data by parsing each line and separate variables by commas

​​ IF cNumber AND cTitle equal null

​​​PRINT an error message “Some information is missing”

​​​ RETURN

​​ ELSE

CREATE NEW course object temp with cNumber, cTitle, preq1, and preq2

​​​ ADD temp to courseList​

​​ ADD cNumber to cInfo

​ SET current node to head

​ SET a boolean isHere to false

​FOR LOOP where I is less than the length of courseList; i++

​​IF  SEARCH (courseList for cInfo[i]) does not equal null

​​​ SET isHere to true

​ IF isHere is false

​​PRINT an error message stating that some prerequisites may not be included​​

Return isHere

}

Void addCourse(Course course) {

FOR each item in parsed vector, I++

INITIALIZE a pointer node with the course info as parameters

IF  courseId is not already in hash vector

​​INSERT pointer node into hash table with Insert() with the courseId and node as Parameters

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

int total = 0

for each course in courses

if course.prerequisite does not equal null

for each prerequisite of the course

total++

**print total**

return total

}

void printSampleSchedule(Hashtable<Course> courses) {

Hashtable<Course> tempTable = courses

Course temp = courses.begin()

FOR each course in courses

FOR each course in tempTable

IF course.id less than temp.id

Temp equals course

**PRINT temp**

DELETE temp from tempTable

}

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 course**

**print the prerequisite course information**

}

// Tree pseudocode

Class Tree {

Course course

Node\* left

Node\* right

Node(){}

Node(Course courseName) {}

Node\* root

}

Void inOrder(Node\* node) {

IF node is not null

inOrder(node.left)

**PRINT course info**

IF node.prerequisite is not null

FOR each prerequisite in node

**Print prerequisite**

inOrder(node.right)

}

Void ordered() {

inOrder(root)

}

Bool openFile(string filename) {

OPEN file

Bool isFine equals true

SET a node pointer called temp

IF file is opened correctly

​​ SET node contents equal to the open file

​​ IF contents is null

​​​​ PRINT error message

​​​​ RETURN

​​SET a new vector called cInfo to hold course info and to be root

​ Temp equals root

​​ WHILE not at the end of the file

​​​​READ the data by parsing each line and separate variables by commas

​​​​ IF cNumber AND cTitle equal null

​​​​​​PRINT an error message “Some information is missing”

isFine equals false

​​​​​​ RETURN

​​​​ ELSE IF cNumber is less than temps cNumber

Temps left equals course with cNumber, cTitle, preq1, and preq2

​

​​ ELSE

​​​​​Temps right equals course with cNumber, cTitle, preq1, and preq2

​​​

​​​ SET current node to temp

RETURN isFine

}

Void addCourse(Course course) {

Node\* current = root

IF root is empty

Root equals course

ELSE

IF course.id is less than current

IF current.left is null

ADD course to current.left

ELSE

Recursively traverse left nodes in tree

ELSE

IF current.right is null

ADD course to current.right

ELSE

Recursively traverse right nodes in tree

}

Node\* recursiveSearch(Node\* node, string courseNum) {

IF node is not null

IF key equals node.key

RETURN node

ELSE IF key is less than node.key

RETURN recursiveSearch(node.left, key)

ELSE

RETURN recursiveSearch(node.right, key)

}

RETURN null

}

Int countPreCourses(\*Node node) {

IF node is null

RETURN 0

ELSE

Int count is 0

Count += countPreCourses(node.left)

Count += countPreCourses(node.right)

RETURN count

}

int numPrerequisiteCourses(Tree<Course> courses) {

return countPreCourses(courses.root)

}

void printSampleSchedule(Tree<Course> courses) {

ordered()

}

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

return recursiveSearch(courses.root, courseNumber)

}

// Menu

Create object to hold courses and data called courses

String courseId

Course course

Int choice1

Int choice2

WHILE choice does not equal 9

PRINT “Menu:”

PRINT “1. Load course schedule”

PRINT “2. Print course list”

PRINT “3. Print course specifics”

PRINT “9. Exit”

PRINT “Enter choice: “

SET choice1 to input received

SWITCH(choice1)

Case 1:

loadFile(filename, courses)

break

case 2:

printSampleSchedule(courses)

break

case 3:

WHILE choice2 is 0

PRINT “Enter course id: “

SET courseId to input

printCourseInformation(courses, courseId)

Choice2 equals 1

Case 4:

PRINT “Goodbye”

EXIT program

## 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

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create vector** | 1 | 1 | 1 |
| **For each line in file** | 1 | n | n |
| **create new node** | 1 | n | n |
| **If error** | 1 | 1 | 1 |
| **For each prerequisite** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **Push\_back** | 1 | n | n |
| **Push\_back prerequisite course information** | 1 | n | n |
| **Print prerequisites** | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

// Hash Table

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Hashtable** | 1 | 1 | 1 |
| **Loadfile()** | 1 | n | n |
| **Create new node for each line** | 1 | n | n |
| **If prerequisites** | 1 | n | n |
| **If error** | 1 | n | n |
| **Add node to list** | 1 | n | n |
| **Addcourse()** | 1 | n | n |
| **If null** | 1 | n | n |
| **Assign course info to node** | 4 | n | 4n |
| **Add node to list** | 1 | n | n |
| **Set next to null or UNIT\_MAX** | 1 | n | n |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 14n + 1 |
| **Runtime** | | | O(n) |

// Tree

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create tree** | 1 | 1 | 1 |
| **loadFile()** | 1 | n | n |
| **For each line in file** | 1 | n | n |
| **Add course** | 1 | n | n |
| **Create vector** | 1 | n | n |
| **If prerequisites** | 1 | n | n |
| **If null make root** | 1 | 1 | 1 |
| **If error** | 1 | 1 | 1 |
| **If node is less than root** | 1 | n | n |
| **Set node to left** | 1 | n | n |
| **If node is greater than root** | 1 | n | n |
| **Set node to right** | 1 | n | n |
| **Print node** | 1 | 1 | 1 |
| **Print prerequisites** | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 10n + 1 |
| **Runtime** | | | O(n) |

**Explanation of data structures:**

Vectors:

The vector structure is the fastest and simplest function for reading and adding courses to the list. It is not like the other methods, when parsed from a file it is simply added to the end. This makes it much easier to put in order because the vector data structure is a one-way list, so the ordering algorithms are simple. The disadvantage of using the vector is when using the search function. Since it is one way, if the item you’re looking for is at the end, you must iterate through every item in the list.

Hashtables:

The hash table is great for search functions. Since each node has a key and tells the algorithm exactly where the item is. However, this key-value pair means that each node must have a key and search for a spot to put the data. Another downside is that hash table cannot be sorted, that is the reason for having keys. To print, we would have to iterate over every item, sort it individually, and then print. This would not be ideal for this program.

Binary Trees:

Binary trees are like vectors, except that they travel in 2 directions, left and right. This makes the search functions much faster and more efficient. Binary trees are great because the data is sorted as its added. In the worst case, it would only search half of the list because it compares values, and the nodes tell the algorithm which direction to go. Trees can also be more cumbersome and difficult to implement.

**Recommendation:**

For this project, we need to parse a file and add it to a list. We also need to be able to add an object later without breaking the program. Lastly, we need the capability of printing in alphanumeric order. For this program, I would recommend the Vector list data structure. Not only is it the fastest, but simplest to implement and add new courses. It would take the longest to search the algorithm, but once sorted, would be simple to iterate over and print each course and its prerequisites.