# 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

Class Course {

String courseNumber

String courseTitle

Vector <string> prerequisits

}

// load data from file and put into vector

// create empty vector

Vector <course> loadCoursesFromFile (string fileName) {

Vector <course> courses

}

Try {

For (each line in file) {

Create course object

Set created object attributes with values from file line

Add object to course vector

}

Close file

// Error possibles

}

Catch (error) {

Print error message unexpected error

}

// file format error

Catch (fileError) {

Print error message file format error

}

// couse does not exist error

Catch (courseDNEerror) {

Print error message course does not exist

}

Return courses

// checks if there are more than two parameters in a course

Bool courseValid (course) {

return

}

// validate loaded course

Bool validateCourses(Vector<course> courses) {

For each course in courses {

If (not courseValid) {

Throw fileError

}

For (each prerequisite in course) {

If (not courseExists) {

Throw courseDNEerror

}

}

}

Return true

}

// check if course is in vector

Bool courseExist (Vector<course> Courses) {

For (each course in courses) {

If (courseNum = coursenumber) {

Return courseExists true

}

}

Return courseExists false

}

// print course information

VOID printCourseInfo(Vector<course>courses) {

For (each course in courses) {

If (course = courseNum) {

Print (courseNum)

Print(courseTitle)

Print(coursePrerequisits)

}

}

}

// Hastable code

Vector<course> loadCoursesFromFile(string, fiePath)

Courses = empty Vector<Course>

Try

File = open(file)

For (each lineInFile) {

CouseData = paramaters

If (courseData < 2) {

Print(error)

}

}

Return empty Vector<course>

couseNum = courseData[0]

courseTitle = courseData[1]

prerequisits = new empty list

if (length courseData > 2) {

for (i; i<length.courseData; i++) {

prerequisite = findCourseNum(courses, courseData[i])

}

}

If (perrequisit <= 0) {

Print(error)

}

Return empty Vector<course>

Prerequisits.add(prerequisit)

Course = createCourse(courseNum, courseTitle, prerequisits)

Courses.add(course)

Close file

Return courses

Course createCourse(String courseNum, courseTitle, List<course>prerequisits)

Course = new course

Course.courseNum = courseNum

Course.courseTitle = courseTitle

Course.prerequisits = prerequisits

Return course

VOID printSample(Vecor<course> courses) {

For (course in courses) {

Print(CourseInfo(courses, course.courseNum))

}

}

VOID printCourseInfo(Vecor<course> courses) {

For (course in courses) {

If (course.courseNum == courseNum) {

Print(course.course.Num)

Print(course.courseTitle)

Print(prerequisits)

}

}

}

// Tree code

Class treeNode

Course data

List<treeNode> children

Tree<course> loadFromFile

VOID instertCourse

Int numPrerequisitCourses(Tree<course> courses, String courseNum)

Node = findCourseNode(courses, courseNum)

If (node not found) {

Print(courseNotFoundError)

}

Return countPrerequisits

VOID printSample(Tree<course> courses)

VOID printCourseInfo

Node = findCourseNode(courses, courseNum)

If (node not found) {

Print(courseNotFoundError)

}

Return

Print CourseInfoPrerequists

// Hashtable pseudocode

FUNC IsValidLine()

Tokens = Split(line, “,”)

IF (length(tokens) > 2)

True

Else

False

FUNC LoadData(file, hashtable)

Try

Open file

While (not at the end of file)

Line = ReadNextLine()

IF (isValidLine)

Course = Parse Data

Add course to HashTable

ELSE

Print “Invalid line”

Close file

Catch

Print “File Error”

FUNC ParseData()

Tokens = Split(line “,”)

courseNum = tokens[0]

courseTitle = tokens[1]

Prereqs = empty list

IF (length.tokens > 2)

Prereqs = Split(tokens[2] “,”)

courseObj = CreateCourseObj(courseNum, CourseTitle, prereqs)

RETURN courseObj

FUNC CreateCourseObj(courseNum, courseTitle, Prereqs)

Course = New CourseObj

Course.CourseNum = courseNum

Course.CourseTitle = courseTitle

Course.CoursePrereqs = prereqs

FUNC PrintCourseInfo(hashtable, courseNum)

Hashkey = GenHashKey(courseNum)

If (hashtabel contains hashtable[hashKey])

Course = hashtable[hashKey]

Print ”Course Number +(course.courseNum)”

Print “Course Title +(course.courseTitle)”

IF (course.Prereqs.Length > 0)

Print “Prerequisits +(course.Prereqs)”

Else

Print “Error”

Else

Print “Course not found”

// Tree pseudocode

Struct Course

String courseTitle

String courseNum

Vector<string> prereqs

Struct Node

Course course

FOR (course)

Create key

IF(key found)

RETURN node->course

IF(key not found)

RETURN course

Class Tree

Private:

Node\* root

VOID addNode(Node\* node, Course course)

Public:

BinarySeachTree()

VOID InOrder()

VOID Inster(Course course)

CourseSearch(string courseNum)

BinarySearchTree::BinarySeachTree

Root = NULLPTR

VOID BinarySearchTree::Inset(Course course)

IF(root == NULLPTR)

Root = new Node(course)

ELSE

This->addNode(root, course)

Course BinarySearchTree::Search(string courseNum)

Node\* current = root

WHILE (current != NULLPTR)

IF (courseNum)

Return current->course

IF (current node.COMPARE courseNum > 0)

Current = current->left

ELSE

Current = current->right

Course course

RETURN course

Int totalPrereqs = 0

Int numPrereqsCourses(Tree<Courses>, string courseNum)

Course course = course.search(courseNum)

WHILE (course->Prereqs != 0)

FOR(prereqs in course->prereqs)

Courses.search(course->prereqs->courseNum)

+= totalPrereqs

VOID printCourseInfo(Tre<course> courses, String courseNum)

Course course.search(courseNum)

COUT courseInfo

WHILE(course->prereqs != 0)

FOR(prereqs in course->prereqs)

Courses.search(course->prereqs->courseNum)

COUT prereqsCourseInfo

Course parseLine(vector<string>line)

IF(line.size == 2)

Course newCourse

Course.courseNum = line[0]

Course.courseNum = line[1]

SET course prereqs to empty vector

RETURN newCourse

ELSE

Vector<string>tempPrereqs

FOR(int I; i<line.size;i++)

TempPrereqs.pushback(line[i])

Course newCourse

Course.courseTitle = line[0]

Course.courseNum = line[1]

SET course prereqs to tempPrereqs

RETURN newCourse

Int main()

Tree\* tree = new Tree()

Vector<string>temp

String line

Ifstream infile(“file”)

WHILE(getline(infile,line))

Stringstream ss(line)

WHILE(s.good())

String substr

Getline(ss, substr, “,”)

Temp.pushback(substr)

Tree.insert(parseLine(temp))

Temp.clear

// Menu

String courseKey;

Course course;

Int choice1 = 0;

Int choice2 = 0;

While (choice != 4) {

Print(“MENU:”);

Print(1. Load Schedule);

Print(2. Display);

Print(3. Remove Course);

Print(4. Exit);

Print(Select Option);

CIN choice1;

}

Case 1:

LoadCourses()

Break

Case 2:

While (choice2 == 0) {

COUT (1: Schedule)

COUT (2: Course)

COUT (Select Option)

SWITCH(choice2) {

Case 1:

COUT (schedule);

BREAK;

Case 2:

COUT (Enter course number);

CIN courseNum;

Course = schedule.Search(courseNum);

If(course == NULL){

COUT (Coures not found);

}

ELSE {

COUT (course);

}

BREAK

}

}

Choice2 == 0;

BREAK;

Case 3:

COUT (Enter course Number);

CIN courseNum;

IF (courseNum = NULL) {

COUT (course does not exist);

BREAK

}

ELSE {

REMOVE courseNum from schedule<>;

BREAK;

}

}

## 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.

| **Reading file & creating courses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Intialize fstream & get contents of file** | 1 | 1 | 1 |
| **Initial string line to line of file** | 1 | 1 | 1 |
| **Initial lineStream to get content of lines in file** | 1 | 1 | 1 |
| **Initalize token to hold a single word of a line** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
|  | 1 | 1 | 1 |
| **Get line** | 1 | N | N |
| **lineStream get current line** | 1 | N | N |
| **Count = 1** | 1 | N | N |
| **Course course for line in file** | 1 | N | N |
| **Token from lineStream to ‘,’** | 1 | 2n | 2n |
| **If count == 1** | 1 | N | N |
| **course courseName = token** | 1 | N | N |
| **I++** | 1 | N | N |
| **Else if count == 2** | 1 | N | N |
| **Course courseName = token** | 1 | N | N |
| **I++** | 1 | N | N |
| **else** |  |  |  |
| **If token != NULLPTR** | 1 | N | N |
| **Add token to course Prereqs** | 1 | N | N |
| **Else print error** | 1 | 1 | 1 |
| **I++** | 1 | N | N |
| **If count < 2** | 1 | 1 | 1 |
| **COUT “Error in file format”** | 1 | 1 | 1 |
| **PUSH course** | 1 | N | N |
| **Clear lineStream** | 1 | n | N |
| **Total Cost** | | | 17n + 6 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Cost | Times Executed | Total Cost |
| Initalize fstream to get file contents | 1 | 1 | 1 |
| Initialize string to hold file line | 1 | 1 | 1 |
| Initialize stringstream to get line contents | 1 | 1 | 1 |
| Intitalize token to hold a single word in line | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| Intialize count to hold token for each line in file | 1 | 1 | 1 |
| Get line from file | 1 | N | N |
| Fill lineStream with curr line | 1 | N | N |
| Count = 1 | 1 | N | N |
| Course course for line in file | 1 | N | N |
| Get token from lineStream set ‘,’ | 1 | 2n | 2n |
| If count == 1 | 1 | N | N |
| Course courseNum = token | 1 | N | N |
| I++ | 1 | N | N |
| Else If count ==2 | 1 | N | N |
| Course courseName = token | 1 | N | N |
| i++ | 1 | n | n |
| Else |  |  |  |
| if token != nullptr | 1 | n-1 | N |
| Add token to Prereqs | 1 | n-1 | N |
| Else COUT file error | 1 | 1 | 1 |
| I++ | 1 | n-1 | N |
| If count < 2 | 1 | 1 | 1 |
| COUT file format error | 1 | 1 | 1 |
| Add course to hashtable | N | N | N^2 |
| Clear line | 1 | n | n |
| Total Cost | | | N^2+16n+6 |
| Runtime | | | O(n^2) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line cost | Times exeuted | Total Cost |
| Create key for course | 1 | 1 | 1 |
| Create node\* node | 1 | 1 | 1 |
| If (node == nullptr) | 1 | 1 | 1 |
| New node newCourse (course, key) | 1 | 1 | 1 |
| Insert newCourse at position[key] | 1 | 1 | 1 |
| Else if (node->key == UINT\_MAX) | 1 | 1 | 1 |
| Node->key = key | 1 | 1 | 1 |
| Node->course = course | 1 | 1 | 1 |
| Node->next = course | 1 | 1 | 1 |
| Else |  |  |  |
| While (node != nullptr) | 1 | N | n |
| Node = nodes->next | 1 | 1 | 1 |
| New node newCourse (course, key) | 1 | 1 | 1 |
| Nodes new = newCourse | 1 | 1 | 1 |
| Total cost | | | N+3 |
| Runtime | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line cost | Times executed | Total cost |
| Initialize fstream fileStream to get contents of file | 1 | 1 | 1 |
| Initialize string line to hold a single line in file | 1 | 1 | 1 |
| Initialize stringstream lineStream to get contents of each line | 1 | 1 | 1 |
| Initialize string token to hold a single word in line | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| Intialize count to hold token for each line in file | 1 | 1 | 1 |
| Get line from file | 1 | N | N |
| lineStream = curr line | 1 | N | N |
| Count = 1 | 1 | N | N |
| Course course for each line in file | 1 | N | N |
| Get token from lineStream set ‘,’ | 1 | 2n | 2n |
| If count == 1 | 1 | N | N |
| Course courseNum = token | 1 | N | N |
| I++ | 1 | N | N |
| Else if count == 2 | 1 | N | N |
| Course courseName = token | 1 | N | N |
| I++ | 1 | N | N |
| Else |  |  |  |
| If token in tree | 1 | N | N |
| Add token to prereqs | 1 | N | N |
| Else output error | 1 | 1 | 1 |
| I++ | 1 | N | N |
| If count < 2 | 1 | 1 | 1 |
| Cout “File format error” | 1 | 1 | 1 |
| Course into tree | N | N | N |
| Clear line | 1 | n | N |
| Total cost | | | N^2+16n+6 |
| Runtime | | | O(n^2) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line cost | Times executed | Total Cost |
| If (course courseNum < curr node courseNum) | 1 | 1 | 1 |
| If (node->leftchild empty) | 1 | 1 | 1 |
| New Node(course) -> left | 1 | 1 | 1 |
| Else traverse nodes left sub tree | 1 | N | N |
| Else |  |  |  |
| If (node rightchild empty) | 1 | 1 | 1 |
| New Node(course) right sub tree | 1 | 1 | 1 |
| Else traverse nodes tight sub tree | 1 | N | N |
| If (root empty) root = new Node(course) | 1 | 1 | 1 |
| Else addNode(root, course) | N+3 | 1 | N+3 |
| Total cost | | | N+3 |
| Runtime | | |  |

Pros

Vector

1. Easy and fast implementation
2. Binary search is very quick
3. Insertion can always happen

HashTable

1. Immediate access to all items in table
2. Adding and delete keeps consistent time
3. Have the potential to be the fastest data structure if implemented very intentionally and well planned and structured

Tree

1. Produces items always in order
2. Fast speed for sorting or accessing

Cons

Vector

1. Removing items is a slow process as it has to shift all other items
2. Takes up larger space depending on other constraints

HashTable

1. Takes up the most space
2. Can occasionally cause long delays due to elements being randomly stored

Tree

1. For good tree sorting performance it must be started correctly
2. Entire balance of the tree for sorting determined by first value