# 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**

}

void openFile(Vector<Course> course){

file = open("course\_information.txt")

}

void readFile(Vector<Course> course){

openFile(Vector<Course> course)

while file.next is not null

line = read next line

}

void parseFile(Vector<Course> course){

readFile(Vector<Course> course)

if line is not null

courseNumber, courseTitle, prereqs = parse

}

void checkFormat(Vector<Course> course){

parseFile(Vector<Course> course)

if line is not null and contains >= 2 params

courseNumber, courseTitle, prereqs = parse line

else print error message

}

void createCourseObject(courseNumber, courseTitle, prereqs){

String courseNumber = this.courseNumber;

String courseTitle = this.courseTitle;

String prereqs = this.prereqs;

}

Course course = new Course(courseNumber, courseTitle, prereqs)

courses = empty vector

courses.append(course)

bool search(String courseNumber){

for each course in courses

if course.courseNumber == courseNumber

return true

print course.courseNumber

print course.courseTitle

print course.prereqs

break

return false

}

// Hashtable pseudocode

void loadFile(file){

open file

for line in file{

delimiter = space/new line

section = split document by delimited

courseNumber = section1

courseTitle = section2

prereqs = section3

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

for prereq in courses[i].prereqs{

print prereq

}

}

void printSampleSchedule(Hashtable<Course> courses) {

course = courses[i]

set course.number = courseNumber

set course.title = courseTitle

set course.prereqs = prereqs

add course to courses, key = courseNumber

}

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

for course in courses{

print courseNumber

print courseTitle

print prereqs

}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

open file

for line in file

courseInfo = line

prereqs = getPrerequisites(courseInfo)

if prereqs = 0, then return “no prereqs”

else return “prereqs: “ + prereqs

}

void printSampleSchedule(Tree<Course> courses) {

schedule = []

root = courses.prereqs(0)

for course in schedule

generateSchedule(left, schedule)

schedule.add(course)

generateSchedule(right, schedule)

print schedule

}

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

course = courses.binarysearch(courseNumber)

print course.courseNumber

print course.title

print course.prereqs

}

Void menu(){

While true:

print "Menu:"

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

userChoice = getInput()

case userChoice == 1:

open data\_file

for line in data\_file:

course = createCourse(courseNumber, courseTitle, prereqs)

dataStructure.add(course)

close data\_file

break

case userChoice == 2:

courseList = dataStructure.sort();

for course in courseList

print course

break

case userChoice == 3:

course = dataStructure.find(courseNumber)

print course.courseTitle

print course.prereqs

break

case userChoice == 4:

exit

}

Void printCourseList(){

dataStructure.sortIgnoreCase()

for course in dataStructure:

if courseTitle.include(“Computer Science”):

print course

}

## 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) |

Vector Runtime Analysis:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **numPrerequisiteCourses** | 3 | N | N |
| **printSampleSchedule** | 1 | 1 | 1 |
| **printCourseInformation** | 4 | N | N |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

Advantage:

* fewer code in each block
* total cost depends on number of times executed
* simple implementation

Disadvantage:

* fixed and predictable performance: no better, no worse
* stringent data structure: modification requires more cost

Hashtable Runtime Analysis:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **loadFile** | 6 | N | N |
| **numPrerequisiteCourses** | 1 | N | N |
| **printSampleSchedule** | 5 | 1 | 1 |
| **printCourseInformation** | 3 | N | N |
| **Total Cost** | | | 3n + 1 |
| **Runtime** | | | O(n) |

Advantage:

* fast data retrieval
* total cost predictable and follows a pattern
* comprehensive coding

Disadvantage:

* bulky coding: needs to be inclusive and explicit
* hash collision: might return exceptions that increases cost if two keys target the same value

Tree Runtime Analysis:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **numPrerequisiteCourses** | 5 | N | N |
| **printSampleSchedule** | 5 | 1 | 1 |
| **printCourseInformation** | 4 | N | Log N |
| **Total Cost** | | | N + log N + 1 |
| **Runtime** | | | O(log n) |

Advantage:

* best worst-case runtime compared to the others
* reduces the overall CPU and memory usage
* structure follows binary pattern that reduces overall complexity

Disadvantage:

* possibility of pointing nullptr if modification is not correctly performed
* might result in data loss if deletion method is not performed correctly
* although overall complexity is O(log n), complexity for each code block might vary, resulting in faster process for certain functions while slower for others

As a conclusion, my preferred data structure is binary tree because it yields a better overall worst-case complexity of O(log n). One thing I also like about binary search trees is that it can be deemed as a functional combination of vectors and hashtables but with higher efficiency. BST has a very ordered structure, just like vectors, to allow direct visualization and easy understanding of the backend logic. When searching for elements, BST will keep performing binary searches instead of going straight from left to right. When making modifications to the tree structure, any child nodes and branches can be dynamically shifted to the appropriate position without worrying about the stability of the structure.