# 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).

Int main() {

Filepath = path to csv

Datastructure = pick a datastructre

Choice = 0

While choice is not 9

Display menu

Get input from user and save to choice

If choice is 1

OpenFileAndReadData(filepath)

If choice is 2

PrintSampleSchedule(allCourses)

If choice is 3

PrintCourseInformation(allcourses, courseNumber)

If choice Is 9

Exit the program

}

Class Course

Private

String courseNumber

String courseName

Vector<string> prereqs

Public

Course\* left

Course\* right

Constructor(string courseNumber, string courseName, vector<string> prereqs) {

This objects courseNumber = courseNumber

This objects courseName = courseName

This objects prereqs = prereqs

Constructor() {

This objects courseNumber = “empty”

This objects courseName = “empty”

Declare this objects prereqs vector

}

}

Class BSTree

Public:

Course\* root

Void OpenAndReadFileData(Course root, string filepath) {

Open file

For each line in the file

If line[0] == null or line[1] == null

Throw exception and exit

courseNumber = Line[0]

courseName = Line[1]

vector<string> prereqs

while there is another field on the line

prereqs.append(field)

for each prereq in prereqs

flag = false

for each line in the file

if line[0] == prereq.courseNumber

flag = true

break

if flag == true

continue

else

throw error

AddCourseToDataStructure(BSTree.root, courseNumber, courseName, prereqs)

}

Void AddCourseToDataStructure(Course root, string courseNumber, string courseName, vector<string> prereqs) {

course = Course(courseNumber, courseName, prereqs)

if root is equal to null

root = course

course->left = null

course->right = null

else

cur = root of tree

while cur is not null

if node courseNum less than cur courseNum

if cur->left is null

cur->left = course

cur = null

else

cur = cur->left

else

if cur->right = null

cur->right = course

cur = null

else

cur = cur->right

node->left = null

node->right = null

}

Course findCourse(Course root, string courseNumber) {

cur = root

While cur isn’t null

If cur.courseNumber is equal to courseNumber

Return cur

If cur.courseNumber is greater than courseNumber

Cur = cur.left

Else

Cur = cur.right

}

// Tree pseudocode

int numPrerequisiteCourses(Course root, Course course\_a) {

numPrereqs = 0

cur = root of tree

while true

if cur = course\_a

return numPrereqs

else if course\_a > cur

if cur->right == null

return numPrereqs

else

cur = cur->right

numPrereqs += 1

else if course\_a < cur

if cur->left == null

numPreqres += 1

return numPrereqs

else

cur =cur->left

numPreqreqs += 1

}

void printSampleSchedule(Course node) {

if node is null

return

print courseNumber, title, and prereq names prereq

printSampleSchedule(node->left)

print(SampleSchedule(node->right)

}

void printCourseInformation(Course root, String courseNumber) {

Course mycourse = findCourse(root, courseNumber)

Print each field of mycourse

}

Void sortandPrintComputerScienceCourses(Course root) {

# A BST is already sorted

inOrder(root)

}

void inOrder(Node\* node) {

if (course == nullptr) {

return;

}

inOrder(course->left);

if (course.courseNumber[0-2] == “CSC”) {

cout << node->course.courseNumber << ": " << node->course.courseName << "

for each prereq in prereqs

cout << course.prereq

}

inOrder(course->right);

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses, Course course\_a) {

total is equal to the courses at hash course\_a

for each prerequisite of the prerequisites

total += courses at hash course\_a

print total

}

void printSampleSchedule(Hashtable<Course> courses) {

for key, value in courses

print name

if course has prerequisites

for each prerequisite

print prerequisite

}

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

for each course in courses

if the course is equal to courseNumber

print course info

for each prerequisite in the course

print prerequisite course info

}

Void sortandPrintComputerScienceCourses(Vector<Course> allcourses) {

Declare sortedVector

For each course in allCourses

If course.CourseNumber at index’s 0-2 == “CSC”

sortedVector.append(course)

sort(sortedVector)

for each course in sortedVector

printCourseInformation(course.courseNumber)

}

Void sort(HashTable<Course> allCourses) {

Int I, j, min\_index

For (I = 0; I < n-1; i++) {

Min\_index = i

For (j = i+1; j< n; increment j) {  
 if (allCourses.at(j) < allCourses.at(min\_index)) {

Min\_index = j;

}

If (min\_index not equal to i) {

Int temp = allCourses.at(min\_index)

allCourses.at(min\_index)= allCourses.at(i)

allcourses.at(i) = temp

}

}

}

}

// 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 prerequisite

for each prerequisite

print prerequisite

}

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 sortandPrintComputerScienceCourses(Vector<Course> allcourses) {

Declare sortedVector

For each course in allCourses

If course.CourseNumber at index’s 0-2 == “CSC”

sortedVector.append(course)

sort(sortedVector)

for each course in sortedVector

printCourseInformation(course.courseNumber)

}

Void sort(Vector<Course> allCourses, int n) {

Int I, j, min\_index

For (I = 0; I < n-1; i++) {

Min\_index = i

For (j = i+1; j< n; increment j) {  
 if (allCourses.at(j) < allCourses.at(min\_index)) {

Min\_index = j;

}

If (min\_index not equal to i) {

Int temp = allCourses.at(min\_index)

allCourses.at(min\_index)= allCourses.at(i)

allcourses.at(i) = temp

}

}

}

}

## 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 (readingTheFile))

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| For each line in the file | 1 | n | n |
| If line[0] == null or line[1] == null | 1 | n | n |
| Throw exception and exit | 1 | 1 | 1 |
| courseNumber = Line[0] | 1 | n | n |
| courseName = Line[1] | 1 | n | n |
| vector<string> prereqs | 1 | n | n |
| while there is another field on the line | 1 | n | n |
| prereqs.append(field) | 1 | n | n |
| for each prereq in prereqs | 1 | n | n |
| flag = false | 1 | n | n |
| for each line in the file | 1 | n | n |
| if line[0] == prereq.courseNumber | 1 | n | n |
| flag = true | 1 | 1 | 1 |
| break | 1 | 1 | 1 |
| if flag == true | 1 | n | n |
| continue | 1 | 1 | 1 |
| else | 1 | n | n |
| throw error | 1 | 1 | 1 |
| Constructor Course() | 2n + 1 | n | 3n + 1 |
| **Total Cost** | | | 16n + 7 |
| **Runtime** | | | O(n) |

Vector (creatingCourseObjects)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| This objects courseNumber = courseNumber | 1 | n | n |
| This objects courseName = courseName | 1 | n | n |
| This objects prereqs = prereqs | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

Runtime Analysis

The advantage of the vector data structure mostly comes from the simplicity of its implementation. However, for this application I do not think it is ideal. The vector data structure requires the user to iterate over each course when searching for a given course. For inserts, the entire vector contents after the insert must be lifted and moved one space to the right. Also, items are merely appended to the end of the array when inserted. As such, when we want to print out a sorted list of computer science courses we must first sort the list and then print it. When sorting the list, algorithms are available which sort the list with high and low runtime and memory complexity. For example, there are slow algorithms such as selection sort and bubble sort which could be used to sort the vector or lightning quick ones like the quicksort or the radix sort. Likewise, some algorithms sort by creating a new sorted list alongside the old one whereas others do all of the sorting in place. The algorithm I used here was the selection sort which is simple to implement, yet slow with a potential complexity of O(n^2). However, all sorting is done in place and as such the space complexity is O(1).

The Hash table data structure has many of the advantages of the vector data structure because it uses a vector under the covers. However, unlike the vector, it’s inserts and lookups are lightning fast with a runtime complexity of O(1) because, due to the hash function, the program knows exactly where to look to find a given item in a list. No iterating is required except for that which is done when using chaining if there is a hash collision. However, in order to print out all the computer science courses in order, since under the covers the hash table is using a vector, the runtime and memory complexities are the same. The underlying vector must first be sorted and then iterated through while checking to see if each course is a computer science course and printing it out in accordance.

The binary search tree data structure has a major advantage over the previous two data structure for the purposes of this program. When inserting data into a binary search tree it is sorted by default (and by definition). As such, when the time comes to print a binary search tree all that needs to be done is to iterate over the tree using an inOrder traversal and printing out each computer science course. No sorting is needed because the tree is sorted on insert. As such, printing out a list of computer science courses will have a runtime complexity of O(n) and space complexity of O(1). Furthermore, lookups in a binary search tree are lightning fast with a runtime complexity of O(logn). The same advantages gained from fast lookup speed give the BST an equivalently quick insert time.

In conclusion, the Binary Search Tree is the data structure I recommend for this program, due to it being ideal for printing a sorted list of classes. This is due to it being sorted by definition. While lookups in a binary search tree aren’t quite as fast as a hash table, this application does not need to be optimized for that. And while a vector has fast inserts to the end of the list, if you need to order a vector, then it takes O(n) comparisons to get each item in the right place. As such, the binary search tree is the better fit, allowing for quick, ordered insertions and ordered lookups.