# 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

bool openAndCheckFile(filename){

//read each line

While(get){

Getline with comma as splitter

If line does not have 2 or more parameters print error and return false

If there is a prerequisite compare to other courses in vector

If prereq does not exist print error and return false

}

Print file is okay

return true

}

}

//assuming course data object already exists

Void storedataobject(filename){

//make sure file exists and is correctly formatted

If openandCheckFile(filename) = !true {

return

}

//go line by line

While(getline(file, line){

//store the line AND split each line by comma

While (getline(courseLine, courseElement, ‘,’){

Create New course object

Assign first element to new course object course number

Assign second element to course Name

Add preceding elements to course prerequisites

Courses.push\_back(courseObject)

}

}

}

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) {

while iterator < vector.size

print course

if course has prerequisites

print course prerequisites

}

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

Void storedataobject(filename){

//create vector

New hashVector<Course> courses

//make sure file exists

If openandCheckFile(filename) = !true {

return

}

//go line by line

While(getline(file, line){

//store the line AND split each line by comma

While (getline(courseLine, courseElement, ‘,’){

Create New course node

Assign first element to new course node course number

Use course number to create a key

Assign second element to course Name

Add preceding elements to course prerequisites

If key is bigger than vector size make vector larger

Store course node at vector key number

If course node at key is already taken chain or place in new bucket/node

}

}

}

int numPrerequisiteCourses(Hashtable<Course> courses, String CourseNumber) {

int numofprereq = 0

turn courseNumber into key

if hashtable(key) exists

loop through prereqs

if prereq exists numofprereq++

return numofprereq

}

void printSampleSchedule(Hashtable<Course> courses) {

while iterator < vector.size

print course

if course has prerequisites

loop prerequisites

print prerequisites

}

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

turn course number into int then into key

use key to search hashtable

if node exists

print course information

else

print course doesn’t exist

}

// Tree pseudocode

Void storedataobject(filename){

//make sure file exists and is correctly formatted

If openandCheckFile(filename) = !true {

return

}

//go line by line

While(getline(file, line){

//store the line

//create a node to store course id and prerequisite ids

//split line by comma to access individual attributes

While (getline(courseLine, courseElement, ‘,’){

//set the name id and prerequisite to proper variables

Create New course object

Move through tree looking for empty variables

if (current is larger than root)

if (left is empty) {

currentleft = object;

// else recurse down the left node

else

current = current left

else

// if no right node

// this node becomes right

if (currentright is empty) {

currentright = object;

else

// recurse down the right node

Current = current.right

}

int numPrerequisiteCourses(Tree<Course> courses) {

current = root

counter = 0

while (current isn’t null)

visit node

if prerequite present

counter + 1

if current.left isn’t null

current = current.left

else

current = current.right

return counter

}

void printSampleSchedule(Tree<Course> courses) {

//start at root

Current = root

//print lefttree

Printsampleschedule(current.left)

//print root

Cout << courseinfo

//print rightside

Printsampleschedule(current.right)

}

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

//set current into node

Current = node

While(current is not null)

If(match is found)

Cout<<courseinfo

If(current is larger)

Current = current.left

Else

Current = current.right

//if match isn’t found return null

Cout << match not found

}

//more than likely main

Void displayMenu(){

Userchoice = 0

While(userchoice is not -1)

Userchoice = getinput

Switch (userchoice)

Case 1

Loaddatastructure()

Case 2

Printcourselist()

Case 3

Printcourse(courseId)

Case 4

Userchoice = -1

Print(thank you for using the course system. Goodbye)

Return 0

}

## 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) |

| **Code for using file and creating courses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open the file** | 1 | 1 | 1 |
| **Check if file is open** | 1 | 1 | 1 |
| **Read the data from the file** | 1 | 1 | 1 |
| **For each line of file** | 1 | n | n |
| **For each word in line** | 1 | n | n |
| **Add course info into new course** | 1 | n | n |
| **Add new course into data structure** | 1 | n | n |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

Final evaluation and Decision

The benefits of using a vector data structure are its dynamic nature. The vector can shrink or grow to match the needs of the application. The disadvantage is that it has to reassign every element after the deletion/ insertion of an element. The other disadvantage is that the vector has to move through all elements during a search making it unideal for large batches of data.

The advantages of a hash table are its ease of implementation and its average retrieval, and insertion speed of constant time. The disadvantages however are that the hash table cannot be iterated over and are prone to a thing called collisions. Which is when the key made for the provided data and used to access the location of said data is the exact same as another key. There are many ways to handle this but it makes the speed drop. Collisions are almost unavoidable in large data sets which is a negative as well.

The advantages of the binary search tree are that it comes already sorted and is fast as far as searching, insertion, and deletion due to being able to traverse the tree based on certain parameters. The disadvantage of a binary search tree is that it is somewhat slow compared to some other data structures especially in regards to retrieval of a random value.

After due consideration I recommend a binary search tree as the data structure for the course catalogue. The deciding factor in this decision is the necessity of a sorted list. The binary search tree while not the fastest is automatically sorted and all that needs to be done to print it in order as was requested is to change 2 or 3 lines of code. For the hash table this is a very large negative. Hash tables by design are not sorted and even making a linked hash table would not allevitate the speed advantage the bst has by already being sorted. The only other option is to make an array that sorts as things are put in as well as the hash table which completely defeats the point. While the vector has a sort function the sort must be done as an additional step in during the creation of the vector or as a step after which gives the binary search tree an advantage over it as well.