**Project 1**

**Vector Pseudocode**

File Reader Pseudocode

Declare vector<Course> courses

Open file

If file is not open

Print error “File did not open”

Else

string line to move through file

while line is not at the end of file

line = get line

if line has less than 2 parameters

print error “file is not formatted correctly.”

return

Else

String check

String prereq

Prereq = 3 parameter in line

While check is not at end of file

courseCheck = get line

courseCheck = the 1 parameter of the line

If check does not equal prereq

Print error “A prereq does not exist”

return

Add line to courses respectively (course number, name. prereq1, prereq2, …, prereqN)

Get next line

Close file

Search and Print Pseudocode

printCourseInfo(vector<Course> courses, string courseID)

size = size of courses

for I = 0; I < size; ++i

if courses(i).coursenumber is equal to courseID

print course information as stored in vector

if course has prereq(s)

print information for the prereqs

**Hash Table Pseudocode**

File Reader Pseudocode

Declare HashTable<Course> courses

Open file

If file is not open

Print error “File did not open”

Else

string line to move through file

while line is not at the end of file

line = get line

if line has less than 2 parameters

print error “file is not formatted correctly.”

return

Else

String check

String prereq

Prereq = 3 parameter in line

While check is not at end of file

courseCheck = get line

courseCheck = the 1 parameter of the line

If check does not equal prereq

Print error “A prereq does not exist”

return

Add line to courses respectively (course number, name. prereq1, prereq2, …, prereqN)

Get next line

Close file

Search and Print Pseudocode

printCourseInfo(HashTable<Course> courses, string courseID)

find key = courseID % size of hashtable

create a node that starts at key

for the size of nodes

if nodes(i).key is not equal to max

node = node at key

while node is not null

print course information as stored in hash table at key

if course has prereq(s)

print information for the prereqs

node is equal to next node

**Tree Pseudocode**

File Reader Pseudocode

Declare Binary Search Tree courses

Open file

If file is not open

Print error “File did not open”

Else

string line to move through file

while line is not at the end of file

line = get line

if line has less than 2 parameters

print error “file is not formatted correctly.”

return

Else

String check

String prereq

Prereq = 3 parameter in line

While check is not at end of file

courseCheck = get line

courseCheck = the 1 parameter of the line

If check does not equal prereq

Print error “A prereq does not exist”

return

Add line to courses respectively (course number, name. prereq1, prereq2, …, prereqN)

Get next line

Close file

Search and Print Pseudocode

printCourseInfo(BinarySearchTree courses, string courseID)

key = courseID

create a node that starts at root

while node is not null

if node key is equal to courseId

print course information as stored in hash table at key

if course has prereq(s)

print information for the prereqs

else if node key is > courseId

node is equal to node->left

else

node is equal to node->right

**Menu Pseudocode**

Print menu

Ask for user input

Int userInput

Case 1: load data

Run read csv file

Load into data structure

Case 2: print courses

Run print all function

Case 3: Print course

Search data structure

Print specific course data

Case 4: exit

Exit program

**Print Pseudocode**

Course current = first item in structure

Course tmp

While current is not null

If current > next

Tmp = current

Current = next

Next = tmp

Else

Current is equal to next item

**Runtime Analysis**

When reading a file, opening and closing the file would both cost 1 as per line of code for each of them. Along this thinking, each line of code we run through would be a cost of 1. This would mean for each line in the file we read, that will cost 1. Therefore, the cost to read the file would be 2 plus however many lines are in the file. Same thing methodology can be applied to creating course objects. Parsing a line would be a cost of 1 and however many times it executes, creating the course object would be the same. It changes when you add it to the data structure because at that point it depends on the structure used. A vector has a big O of O(n). The hashtable would have O(1) as its big O, where the tree would have O(log(n)).

**Data Structure Analysis**

We have 3 data structures to look at. All of them have their pros and cons. Vectors would be obvious because they are easy to use and manipulate. They are easy to add onto and the way they are stored in memory makes them easy to access, but when it comes to inserting or deleting anything in the middle, it is costly. When we look at hash tables, they are a little more complicated to work with because you must make sure collisions are dealt with. However, inserting and search for items is fast. This allows for quicker runtimes, but depends because items are not stored in order with many changes that happen. Finally, the tree is the easiest to manipulate as far as inserting, searching, and deleting. This makes the runtime for these functions fast. A tree requires more memory to store everything, but all of the elements stay where they are put unless changed by somebody.

**Recommendation**

I plan of using the tree to do my code. Since we are sorting these courses by course ID treating and sorting them by greater than less than will be perfect. Also, it may not have the fastest runtime in all areas, I believe it provides the best options to search and insert as needed. For me, the tree makes more sense with the items we are dealing with and what we need to do with them. Plus in the assignments we have completed, the tree was more complicated for me to understand but, once I got it up and running it made the most sense.