**2&3 Menu + Print pseudocode**

Void menu(){

Choice = null;

While(choice is not 5){

Cout << 1. loaddata << nextline << 2. Print courses << next line << 3.Print course << next line << quit

}

Cin >> choice;

Switch(choice){

Case 1:

int dataStructure = null;

switch(dataStructure)

case 1:

//Loads all items in file into the Vector

Vector.addItems(fileName)

case 2:

//load all items into hashmap

HashMap.addItems(fileName)

case 3:

//load alll items into BinaryTree

BinaryTree.addItems(fileName)

break

)

Case 2:

fileReader;

fileReader opens(fileToRead)

Vector <String> listToPrint

Vector <String> courseName

//Store lines ones by one to be evaluated for formatting

While(its not the end of file){

String line

//add each line to vector

getnextline(inthisfile, store to LINE)

ListToPrint.pushback(line)

}

//loop through the vector and add all the courses names to a vector

For(length of the vector listToPrint){

String line = line at index

String name = line.substr(starting at 0, “,”)

courseName.add(name)

Sort(courseName.begin(), to the end);

//looping the sorted name list first we will check every class

For(int I = 0; while I < courseName length; ++i){

For(int j = 0; j < listToPrintSize(); j++){

Line = listToPrint.at(j)

//once the class is found print all the information together

If(line.find(courseName.at(i)) ){

Cout << listToPrint.at(I)

}

}

}

break

Case 3:

userChoice = null

cin >> userChoice

fileReader;

fileReader opens(fileToRead)

//Store lines ones by one to be evaluated for formatting

While(its not the end of file){

String line

//add each line to vector

getnextline(inthisfile, store to LINE)

Size\_t pos = line.find(“,”)

if (pos != string::npos)

String substr = line.subStr(0,pos);

if(subStr == the userChoice){

cout << the lines coursename and prereqs

}

break

Case 4:

exit loop

}

}

**Evaluation.**

**4.**

1. For my program the file will be opened using a file reader that will read the file until there is nothing left to be read. While reading the file the reader will store each line to a String variable named line. Once stored each line will be added to a vector that is holding each line of the string. After storing every line, we will now evaluate that vector with a for loop. We will loop through saving all the classes names to a new vector for checking of prerequisites. Next, we loop through the original vector holding all the class information then we will use a string stream to split this information. The information will be turned into substrings every time a “,” occurs. Once everything is broken into parts we will have our course code, course names, and prerequisite information. Once by one we will check all the requirements.
2. HashMap - For my hash map implementation of the function we can see the run time would usually be O(1) reason being is because unless we have a collision it will be on average O(1). But since we know the average is not the worst case we evaluate that and see that the worst case can be O(n^2). This would occur when n is the number of lines in the input file that is being parsed, and we also have multiple collisions in our HashMap.

BinaryTree - For my Binary Tree implementation we have a worst case scenario of O(n\*log(n)). This is because O(n) would be the number of lines that are in our input file. Then we look further down for our binary tree function which has an insertion time O(log(n)) this together will give us O(n\*log(n)). The worst-case scenario would mostly play out in situations where we have a tree being inserted in ascending or descending our this would cause us to have a chain of nodes that only have a right or a left child.

Vector – For my vector implementation the worst case scenario would be O(n^2). Where n is the number of lines that are within my input file because when you input into a vector it is O(n), but without having an exact vector size we will run into situations where it has to be resized into a new memory location with more capacity. This will result in us have O(n^2) for the insertions of n.

**5.** Based on what I have learned so far, each of these 3 data structures really depend on what your program requires. This is because they all have advantages and disadvantages. Vectors are great if you need constant access time, and your data is not required to be ordered. HashMap are great if you want the fast access time of O(1), but that can be a problem If lots of collisions are happening. HashMap are also not ordered in a particular way which would not help if you need that order. Last we have the Binary Tree, which is slower than the HashMap, but can be ordered and is still very efficient with the insertion time of O(log N)). The only problem with that is a binary tree can be unbalanced if not properly down and could hinder the performance.

**6.** For this particular situation I would lean towards using a HashMap. This is because it is by far the best runtime with O(1) on average. I do not imagine a world where there are millions of classes being inputted and causing lots of collisions. This would mean the HashMap would run efficiently and be a great choice. I also think due to the program not really needing to be in order that this would make the HashMap a great choice. So, due to my analysis I would choose the HashMap for the data structure for this particular program.