**opens the file, reads the data from the file, parses each line, and checks for file format errors**

coursePharser(string fileName) {

file equals open fileName

rawLine equals get file.readLine

For rawLine does not equal endOfFile {

courses[].number equals upto first comma

courses[].title equals upto first comma or end of rawLine

If rawline does not equal endOfFile

courses[].prequisite[0] equals rawline up to next comma or end of line

If rawline does not equal endOfFile

courses[].prequisite[1] equals rawline up to next comma or end of line

}

HashTable equals size of courses

for (all course in couses) {

HashTableInsert(course)

Insert(course)

}

}

**course objects and store them in the appropriate data structure**

course {

number

name

perquisite[]

}

BinarySearchTree::BinarySearchTree() {

root equals NULL;

}

Insert(Course course) {

if(root not equals Null) {

root equals course

}

else {

addNode(root, course)

}

}

addNode(Node\* node, Course course) {

if(course.number is less then node->number) {

if(node->left is Null){

node->left equals course

}

else {

addNode(node->left, course)

}

}

else {

if(node->right is Null){

node->right equals course

}

else {

addNode(node->right, course)

}

}

}

HashTable:Insert(Bid bid) {

key = hash(bidId)

If(node at key location equals NULL) {

Add bid and key to node at key location

}

ElseIf(node at key location not equals NULL) {

If(node->next is not empty) {

move to next location on chain

}

add bid and key to end of chain

}

}

**print out course information and prerequisites**

printCourses() {

PrintList()

PrintAll()

InOrder();

}

PrintList() {

current = head

While(current != nullptr) {

print course information

current = current->next

}

}

PrintAll() {

current equals top of nodes

for( all node in nodes) {

IF(current->key not equals NULL) {

print course information

while(node has more chained nodes) {

print course information

move to next node on chain

}

}

set current to next node in table

}

}

InOrder() {

inOrder(root)

}

inOrder(Node\* node) {

if (node not equals Null) {

inOrder(node->left)

display course info

inOrder(node->right)

}

}

MENU

menu(string filename) {

While(input is not '9') {

display menu choices

get input

if(input is '1') {

coursepharser(filename)

Sort(courses)

}

ElseIf(input is '2') {

printCourses()

}

ElseIf(input is '3') {

printOneCourse(course)

}

}

}

SORT

Sort(){

LinkedSort()

HashSort()

}

LinkedSort(linkedList) {

beforeCurrent equals linkedList->head

current = linkedList->head->Next

while(current is not Null) {

nextNode equals current->next

position = insertPosition(linkedList, current->number)

if(position equals beforeCurrent){

beforeCurrent equals current

}

else{

ListRemove(linkedList, beforeCurrent)

if(position equals Null) {

prepend(linkedList, current)

}

else{

listInsert(position, current)

}

current equals nextNode

}

}

}

insertPosition(list, number) {

current equals Null

nextNode equals list->head

While(nextNode is not Null and number > nextNode->number) {

current equals nextNode

nextNode equals nextNode->next

}

Return current

}

## Runtime Analysis

For charts see excel

Looking at the three different data structures and the two that stand out are the tree and vector.

Both have the runtime of O(n) for input and data structuring, but I think that this is not the whole story so let’s look at all three.

The hash table which is showing O(n^2) takes a long time to set up the table with all the courses but once the table is filled the hash will have the courses in alphabetical order which will save on runtime when it is time to print out the list.

The Vector as the best time now and uses the fewest lines to do so but still must be sorted for output. While that may be good if the program will output in different orders that is not the case for this program.

The Tree has a similar runtime as the vector while using a few more lines to do so. The advantage for this data structure is that the tree sorts as the nodes are added so like the hash table it is ready to print alphabetically.

Looking at the three’s runtime and taking into consideration the need for the program to output alphabetically my choice for the best structure is the Tree.