# Struct Prerecs

//this will give us a structure to load prerequisites into

String preCourse

# Struct Course

//outlines the content of each course that will be stored in the vector of courses

String courseNumber

String courseTitle

Vector <Prerecs> coursePrerecs //this will set up the vector that the prerecs will be stored in

# Class HashTable

// this gives the hash table something to live in: it constructs nodes to populate, including a key, pointers to the next node, a vector of courses, and a table size

Private

Struct Node // gives us some type of structure to put courses into

Course course

Unsigned int key

Node\* next

Node() // node default constructor

Key = UINT\_MAX

Next = nullptr

Node(Course aCourse) : Node() // initializer with just aCourse

Course = aCourses

Node(Course aCourse, unsigned int aKey) : Node(aCourse) // initializer with aCourse and key

Key = aKey

Vector <Node> nodes

Unsigned int tableSize = 0 // this will need to be defined in global definitions

Unsigned int hash(int key)

Public:

HashTable( unsigned int size)

Virtual ~HashTable()

Void Insert( Course course)

Void PrintAll()

Void hashToVector()

# HashTable(unsigned int size)

// This assigns the hash table’s size using the load courses function, specifically the iterator that gets the total number of courses, well lines really, in the course.txt file.

This->tableSize = size

Nodes.resize(size)

# ~HashTable // Destructor

// this erases the hash table once the class is destroyed

Nodes.erase(nodes.begin())

# Hash( int key)

// this determines the key using a modulus of the table size

Return key % tableSize // this will return an integer that is some modulus of the tableSize

# Insert (Course course)

// This will allow the creation of a node that represents a course and its contents

Unsigned key = hash(atoi(course. courseNumber.c\_str()))

Node\* oldNode = &(nodes.at(key)

If (oldNode == nullptr)

Node\* newNode = new Node(course, key)

nodes.insert(nodes.begin() + key, (\*newNode))

else

if ( oldNode -> key == UINT\_MAX)

oldNode->key = key

oldNode->course = course

oldNode->next = nullptr

else

while (oldNode-> next != nullptr) // this skips nodes that already have stuff in them

oldNode = oldNode->next // iterate

oldNode -> next = new Node(course, key) // unless oldNode -> next == nullptr then insert the course information and its key

# displayCourse(Course course)

/\* this displays the contents of each individual node in this format:

MAT123: Made up math class that’s been debunked |

No Prerequisites

OR

MAT122, MAT121, MAT(Insert number here)

\*/

cout << course.courseNumber << “: “ << course.courseTitle << “ | “ << endl

if (coursePrerecs.size() == 0)

cout << “No Prerequisites.” <<endl;

else

for(int I = 0; I < coursePrerecs.size(); ++i)

if(I < coursePrerecs.size()-1)

cout << coursePrerecs.at(i) << “, “

else

cout << coursePrerecs.at(i) << endl;

return

# hashToVector()

/\*this will take our hashTable and populate a temporary course list which will then be sent to a quick sort function. The idea fo this, specifically the iterator function came from the PrintAll function out of the original hash table assignment. Its just hacked and modified.\*/

Vector <Course> courseList // create a vector of courses to transcribe the hashtable into

//hash table iterator, taken directly from the PrintAll function in the original assignment.

For (auto it = nodes.begin(); it != nodes.end(); ++it)//iterator 1

Course course // create a new course structure

If(it->key != UINT\_MAX) // if the key is unassigned

//transcribe variables from the hash tableto the course structure

course.courseNumber = It-> course.courseNumber

course.courseTitle = It->course.coursetitle

// if there are course prerecs within the hashtable’s course transcribe them…

if (it->course.coursePrerecs.size() > 0)

int I

//use a for loop to extract information from the hash tables course prerecs

for (I = 0, i<it->course.coursePrerecs.size, i++)//iterator 2

//capture and populate the vector of prerecs contained in the vector of courses from the hash tables courses

course.coursePrerecs.push\_back(it->course.coursePrerecs.at(i))

// end Iterator 2 :: meaning iterate to the next prerequisite or get out of the loop

courseList.push\_back(course)// push the course structure with the populated fields into the vector

// if there are no course prerequisites add the course to the list

Node\* node= it->next

else

courseList.push\_back(course)

//end iterator 1 :: meaning iterate to the next node to capture its course information or get out of this loop

Node\* node= it->next

// now that weve populated our courseList vector time to send it to the quick sort factory using determined variables.

Int begin = 0

Int end = couresList.size()-1

courseQuickSort( courseList, begin, end) // function to sort the courseList

PrintAll(courseList) // function to print out the alphabetized contents of courseList

# courseQuickSort(Course courseList, int begin, int end)

// this is set up nearly identical to the quick sorting coding assignment that we did in module 2 using recursive calls to sort things out.

int p = 0

// if there isn’t anything in the courseList

if (begin >= end)

return

p = partitionCourses(courseList, begin, end)

quicksort(courseList, begin, p)

quicksort(courseList, p+1, end)

# partitionCourses(Course courseList, int begin, int end )

/\* this is set up nearly identical to the quick sorting coding assignment that we did in module 2 using recursive calls to sort things out\*/

int low = begin

int hi = end

//mid point string that is the courseNumber based on a computation using courseList size

string pivot = courseList[low+(hi-low)/2].courseNumber

while(low < hi)

//compare the alphanumeric values at the courseList at low’s Course number to that of Pivot. If its greater than pivot iterate to the next courseNumber.

while (courseList[low].courseNumber.compare(pivot) >0)

++low

// compare the alphanumeric values at the courseList at hi’s courseNumber to that of Pivot. If its less than pivot, iterate to the next courseNumber

while (courseList[hi].courseNumber.compare(pivot) < 0)

--hi

//Swap low and hi courseList

If ( low < hi )

Swap(courseList.at(low), courseList.at(hi))

++low

--hi

Return hi

# PrintAll(Course courseList)

// this is simply an iterator that rolls through the sorted list and prints it.

Int I

For(i=0, I < courseList, i++)// iterator 1

Cout << courseList.at(i).courseNumber << “ :: “ <<courseList.at(i).courseTitle<< endl

# loadCourses(string somePath, HashTable\* hashTable)

// this loads parses the *text file*. I find this interesting because we haven’t actually, or I haven’t actually parsed a text file to extract its data. Going to give it my best shot.

cout << “Loading TXT file “ << somePath

ifstream inputFile(“somePath\some\_file.txt”)

if(!inputfile.is\_open())

cerr << “Error opening file.” << endl

// The following will determine tableSize

Int lineCount = 0 \\ iterator used to adjust the size of the table, therefore hash

String countLine

// while there actually is a line to count in some\_file iterate the line count

While( getline( somePath\some\_file.txt, countLine))

lineCount++

//resize the hash table using the final lineCount Variable

HashTable(lineCount)

try

string line // this is to capture each line

// while loop to capture each line in the text file

while(getline(some\_file.txt, line)// iterator 1

Course course // create a course structure

stringstream ss(line) // get stringstream going to read the line

string info // this is to capture each course

int iterator = 0 // used to keep track of where we are in the line

//while loop to capture each bit within the text file using a comma as a delimiter.

while( getLine(ss, info, ‘,’) // iterator 2

if(iterator == 0) // this is the course number

course.courseNumber = info

if(iterator == 1) // this is the full title

course.CourseTitle = info

if(iterator >=2) // this is where it starts to add prerecs to the vector of prerecs

course.course.Prerecs.push\_back(info)

Iterator++ // iterate the iterator so that the next delimiter can be input

//iterator 2 end:: meaning there is no other ‘,’ delimiters in the line

hashTable->Insert(course) // insert the course into the hash table

//iterator 1 end:: meaning its either time to look a the next line, or if there are no other lines get out of the while loop and then close the file

file.close() // close the text file once we’re done adding courses

# QualityControl()

//Iterator to make it through the hash tables courses

For (auto i = nodes.begin(); i != nodes.end(); ++i)//iterator 1

// if the course contained in I has some course prerecs…

If(i -> key != UINT\_MAX && i->course. coursePrerecs.size()>0)

Int j

// second iterator to roll through the course prerecs

For ( j = 0; j < i->course.coursePrerecs.size(); ++j)//iterator 2

// third iterator to compare each of the prerecs to each of the courses within the hash table

For( auto k = nodes.begin(); k != nodes.end(); ++k)//iterator 3

If(k->key !=UINT\_MAX &&)

//if j’s course number matches a course in I,

If(i->course.coursePrerecs.at(j).compare(k->course.courseNumber) == 0)

// iterate iterator 1

Node\* node = i->next

// if prerecs at j don’t match k’s course number

If(i->course.coursePrerecs.at(j).compare(k->course.courseNumber) != 0)

//iterate iterator

Node\* node = k->next

Else

Cout << “ Course Prerequisite doesn’t match any of the courses! :: Quality Control error! “ << endl;

Return 0

int i // iterator to make it through the course prerequisites

//iterator to iterate through courses

for (i = 0, i < node->course.prerecs.size(), i++) // iterator 1

If( node != nullptr &&

node->key != UINT\_MAX &&

node-> course.Prerecs.size() > 0)

int j // iterator to make it through the prerecs

while ( j < course.Prerecs.size )

If(node == nullptr || node-> == UINT\_MAX)

Return course

While (node != nullptr)

If(node->key != UINT\_MAX && node->course.courseNumber.compare(courseNumber) == 0)

Return node-> course

Node = node->next

Return course

# Search(string courseNumber)

Unsigned int key = hash(atoi(courseNumber.c\_str()))

Node \*node = &(nodes.at(key))

If( node != nullptr && node->key != UINT\_MAX && node-> course.courseNumber.compare(courseNumber) == 0)

Return node-> course

If(node == nullptr || node-> == UINT\_MAX)

Return course

While (node != nullptr)

If(node->key != UINT\_MAX && node->course.courseNumber.compare(courseNumber) == 0)

Return node-> course

Node = node->next

Return course

# main( )

String filePath, bidKey

Int choice = 0

HashTable\* courseTable

Course course

courseTable = new HashTable()

clock\_t = ticks

clock\_t = tTicks

### \\ menu

while ( choice != 9 )

cout << "Menu:" << endl

cout << " 1. Load Courses" << endl

cout << " 2. Get Course Information" << endl

cout << " 9. Exit" << endl

cout << "Enter choice: "

cin >> choice;

### case 1 Load courses

cout << “What file are you looking for?”<< endl

cin << somePath

ticks = clock()

tTicks = clock()

courses = loadCourses(somePath, couresTable)

ticks = clock() – ticks

cout << “Courses loaded”<< endl

cout << “time to load courses:” << ticks << “Clock ticks” << endl

cout << “time it took to load courses in seconds: << ticks\*1.0 / CLOCKS\_PER\_SEC << endl

ticks = clock()

QualityControl(courseTable)

ticks = clock() – ticks

cout << “time to run QC:” << ticks << “Clock ticks” << endl

cout << “time it took to run QC in seconds: << ticks\*1.0 / CLOCKS\_PER\_SEC << endl

tTicks = clock() – tTicks

cout << “time to Load courses and QC them:” << tTicks << “Clock ticks” << endl

cout << “time it took in seconds: << tTicks\*1.0 / CLOCKS\_PER\_SEC << endl

break

### case 2 Get Course Information

cout << “Do you want to see all courses(a) or do you want to see just one course(b)?” << endl

getLine(cin, case)

#### if (case == “a”) // display all courses

// this sends the course table off to hashToVector()

courseTable-> hashToVector()

return

#### if ( case == “b”) // search for a single course

cout << “Which course are you looking for? << endl;

getLine(cin, choice))

cout << “searching for “ << choice << endl;

courseTable->displayCourse()

return

cout << “Good bye!”<< endl