# Basic Structs within Hashtable

Course struct defines structure containing course information

**struct Course** {

string courseID - unique course identifier(example: CSCI101)

string courseTitle - title of course

vector<String> preRequisites - list of courseIDs that are prerequisites

unsigned int key - hash key

**Course(string courseID, string courseTitle)** default constructor

courseID = courseID;

courseTitle = courseTitle

key = UINT\_MAX

**Course(string courseID, string courseTitle, vector<string> preRequisites)** overload

courseID = courseID;

courseTitle = courseTitle

preRequisites = preRequisites

}

# HashTable Boilerplate

vector containing courses used in hashtable

**vector<Course> courses;**

Table size value used in hashtable for hashing each element

**unsigned int tableSize**

default constructor

**HashTable Constructor()**

courses.resize(tablesize)

Parameterized constructor to set specific tablesize

**HashTable Constructor (int size**) -- overload

tableSize = size;

courses.resize(tableSize);

delete courses from the beginning

**Hashtable destructor**

courses.erase(courses.begin)

hashing method to return unique key for each hash item

**int hash (int key)**

return key % tableSize

# Hashtable Methods

insert new course into hash table

**Insert(Course course)**

create a key by hashing courseID

check if node already exists using key

if key is found

print error message and return

if key is null

assign this node to the key position

else if node is not used

assing old node key to UNIT\_MAX,

set to key,

set old node to bid

set old node next to null pointer

else find the next open node and add new newNode to end

while next node is not null

set prev node to next

Search courses by id in hash table

**Course Search(string courseID)**

Course course

create the key for the given course

if entry found for the key

return node course

if no entry found for the key (node is null)

return course

while node not equal to nullptr

if the current node matches

return current node

node is equal to next node

if all else fails return course

# Course Validation Logic

Checks basic validation to ensure the line is at least 2 arguments and throw if not

**validateLine(line)**

split the string by ","

check at least 2 items are in the array

if < 2

throw error

output message showing line number, content, and "not enough arguments"

Validate vector of preRequisites to ensure ids exist in course list

**validatePrerequisiteCourses(vector preRequisiteCourses)**

loop through preRequisite courses vector

hash courseID for prereq course

search hashtable for key containing this hash

if found

check next preRequisite

else

throw error showing course and remove from vector

# Quick Sort Logic

partitioning and sorting logic used in print methods

**int partition(vector<Course>& courses, int begin, int end)**

set low and high equal to begin and end

pick the middle element as pivot point

while not done

(We are comparing course.title)

keep incrementing low index while courses[low] < courses[pivot]

keep decrementing high index while courses[pivot] < courses[high]

If there are zero or one elements remaining,

all courses are partitioned. Return high

else swap the low and high courses

move low and high closer ++low, --high

return high;

**void quickSort(vector<Course>& courses, int begin, int end**)

int mid = 0

if (begin >= end) return

mid = partition(courses, begin, end)

quickSort(courses, begin, mid)

quickSort(courses, mid + 1, end)

# Print Logic

Iterate through hashtable, insert into temporary vector to sort, and print sorted data

**printCourseList(Hashtable courses)**

if courses is null

output “No Courses”

return

create new vector sortedList

loop through courses(from node begin to nodes.size)

if key doesn't equal default key value (UINT\_MAX)

add to sortedList vector

quickSort(courses, 0, courses.size() - 1);

loop through courses(from node begin to nodes.size)

if key doesn't equal default key value (UINT\_MAX)

printCourse(courseID)

Print course info

**printCourse(string courseID)**

search(courseID)  
 cout << courseID << “ “ << courseTitle << “ “ << “Prerequisites: “

iterate through prerequisites vector and print each entry  
 if vector is empty, print “N/A”

Print Menu Text

**printMenu()**

output the following lines:

“Menu:”  
“1: Load Courses”  
“2: Display all courses”  
“3: Find Course”  
“9: Exit”  
“Enter option”

# Menu Logic

load courses from file

**Hashtable<Course> loadCourses(string filePath, Hashtable courses)**

output "loading file" to console

initialize hashTable courses

load file

create hashTable(rowCount)

for(i = 0; i < rowCount; i++)

validateLine() - Check if line is valid before writing to hashtable

If basic validation passes, write to hashtable

if valid

Course course

course.courseID = file[i][0] - id is the first argument in line

add courseID to courses hashtable

course.courseTitle = file[i][1] - title is second

add remaining arguments to course.preRequisites vector (we will

validate these once we have all courses)

courses.Insert(course)

otherwise print error

Call menu functions, get user choice and dispatch actions

**loadMenu()**

int choice = 0

Hashtable courseTable = nullptr

string csvPath = (some hardcoded path for file)

printMenu()

cin >> choice

switch

case 1:

caseTable = new courseTable()

loadCourses(csvPath, courseTable)

case 2:

printCourseList(courseTable)

case 3:

cout << “Enter ID for course: ”

string courseID

cin >> courseID

printCourse(courseID)

case 9:

exit

default:

output “Please enter a valid choice”

recursively call printMenu()