Project 1

Rachel Iris Aldava

CS-300

Prof. David Ostrowski

Oct 8, 2023

NOTE: Due to formatting challenges within Microsoft word, the pseudocode has been submitted in another file named pseudocode.txt

Assumptions:

OUTPUT will display information directly to end user (c++: std::cout, Java: System.out.println, Python: print)

OPEN FILE will facilitate the opening of a .csv file

GET NEXT LINE will facilitate a line-by-line reading of the aforementioned .csv file

QUICKSORT() will employ the quicksort algorithm, which all students in CS-300 ought to have already implemented in previous courses no less than twice.

**BEGIN PSEUDOCODE**

cost|# executes |total cost

0| 0| 0| // Defines Course objects with some specialized get methods

0| 0| 0| DEFINE class Course:

1| 1| 1| DEFINE constructor Course(id, name, prerequisiteStrings):

1| 1| 1| SET object.id AS id

1| 1| 1| SET object.name AS name

1| 1| 1| SET object.prerequisiteStrings AS prerequisiteStrings

1| 1| 1| DEFINE object.prerequisites AS Vector<Course>

1| 1| 1| DEFINE object.fullPrerequisites AS Vector<Course>

Total cost: 6 (to instantiate a single course object with given information)

Runtime: O(1)

cost|# executes |total cost

0| 0| 0| // Calls (GET NEXT LINE), and returns a Course object

0| 0| 0| DEFINE STATIC METHOD

1| 1| 1| Course nextCourse(dataStream dataStream) {

1| 1| 1| SET string line AS dataStream(GET NEXT LINE)

1| 1| 1|

1| 1| 1| SET info AS empty string vector

1| 1| 1| SET cell AS empty string

1| k| k| FOR each character in line

1| k| k| IF character IS NOT ","

1|k-j|k-j| ADD character to cell

1|j-1|j-1| ELSE

1|j-1|j-1| APPEND info WITH cell

1|j-1|j-1| SET cell AS empty string

1| 1| 1|

1| 1| 1| IF cell IS NOT an empty string

1| 1| 1| APPEND info WITH cell

0| 0| 0|

1| 1| 1| IF SIZE OF info is LESS THAN OR EQUAL TO 1

1| 1| 1| RETURN NULL (or THROW exception)

1| 1| 1| ELSE IF SIZE OF info EQUALS 2

0| 0| 0| // Return a course with no prerequisites

1| 1| 1| RETURN new Course(info[0], info[1], [])

1| 1| 1| ELSE:

0| 0| 0| // Return a course with prerequisites

1| 1| 1| RETURN new Course(info[0], info[1], info[FROM 2 TO end])

1| 1| 1| }

Total cost (worst case): 10 + 3k + 2j (where k is the characters in a given line and j is the number of commas in the line)

Total cost (worst case estimate): 2400 (A doctoral capstone with 80 course prerequisites and a lengthy course title)

Runtime: O(1)

cost|# executes |total cost

0| 0| 0| // An instance method which prints information for one course

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void printCourseInformation(){

1| 1| 1| OUTPUT: "ID: {object.id}"

1| 1| 1| OUTPUT: " Name: {object.name}"

0| 0| 0|

1| 1| 1| SET hasPrerequisites AS Boolean

1| 1| 1| IF length of object.prerequisites vector EQUALS 0

1| 1| 1| SET hasPrerequisites as FALSE

1| 1| 1| ELSE

1| 1| 1| SET hasPrerequisites as TRUE

0| 0| 0|

1| 1| 1| IF hasPrerequisites

1| 1| 1| SET outputString AS empty string

0| 0| 0|

1| n| n| FOR each course in object.prerequisites

1| n| n| ADD course.id TO outputString

0| 0| 0|

1| n| n| IF this is not the final iteration, ADD ", " to

outputString

0| 0| 0|

1| 1| 1| OUTPUT: " Prerequisites: {outputString}"

0| 0| 0|

1| 1| 1| else:

1| 1| 1| OUTPUT: " Prerequisites: None"

0| 0| 0|

1| 1| 1| IF hasPrerequisites AND there are more

object.fullPrerequisites than object.prerequisites

1| 1| 1| SET outputString AS empty string

0| 0| 0|

1| n| n| FOR each course in object.fullPrerequisites

1| n| n| ADD course.id TO outputString

0| 0| 0|

1| n| n| IF this is not the final iteration, ADD

", " to outputString

0| 0| 0|

1| 1| 1| OUTPUT: " Full Prerequisites:

{outputString}"

1| 1| 1| }17 6

Total cost (worst case): 6n + 16 (A course with n-1 prerequisites and n full prerequisites)

Runtime: O(n)

cost|# executes |total cost

0| 0| 0| // Used to calculate a full list of Course dependencies

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| Vector<Course> getFullPrerequisites(){

0| 0| 0|

1| 1| 1| SET recursivePrerequisites AS COPY OF

object.prerequisites

0| 0| 0|

1| 1| 1| IF LENGTH OF recursivePrerequisites EQUALS 0

1| 1| 1| return empty vector

0| 0| 0|

1| n| n| FOR prerequisite IN recursivePrerequisites

n!| n| n!| SET localPrereq AS

prerequisite.getFullPrerequisites()

0| 0| 0|

1| n| n| FOR entry IN localPrereq

1| n| n| IF entry NOT IN recursivePrerequisites

1| n| n| APPEND recursivePrerequisites WITH entry

0| 0| 0|

1| 1| 1| RETURN recursivePrerequisites

1| 1| 1| }

NOTE: There is no real-world course which has every course as a prerequisite, nor will every course have every course as a prerequisite

Thus, we should anchor our worst-case scenario with reality rather than conclude a n! runtime.

If a doctoral capstone ultimately requires 80 courses, and if each course is listed as a prerequisite on 5 other courses

then the worst case total cost would be (80\*5) \* 11 = 4400

Given the walled nature of prerequisites, I believe a runtime of: O(n) is not unreasonable.

cost|# executes |total cost

0| 0| 0| // Defines Vector objects

1| 1| 1| DEFINE class Vector:

1| 1| 1| DEFINE constructor Vector(Vector courseList):

1| 1| 1| SET object.courseList AS courseList

Total cost: 3 (to instantiate a single vector object with given information)

Runtime: O(1)

cost|# executes |total cost

0| 0| 0| DEFINE STATIC METHOD

1| 1| 1| Vector load(dataStream dataStream){

0| 0| 0|

0| 0| 0| // initialize the vector

1| 1| 1| DEFINE vector as Vector(Vector <Course>)

1| 1| 1| DEFINE Course course

0| 0| 0|

0| 0| 0| // Load courses

1| n| n| WHILE TRUE:

1| n| n| SET course AS Course.nextCourse(dataStream)

1| n| n| IF course EXISTS

1| n| n| vector.courseList.append(course)

1| n| n| ELSE

1| 1| 1| break

0| 0| 0|

0| 0| 0| // Calculate first-level prerequisites

n^2| 1|n^2| CALL vector.refreshPrerequisites()

0| 0| 0|

0| 0| 0| // Calculate recursive prerequisites

n^2| 1|n^2| CALL vector.refreshFullPrerequisitesTree()

0| 0| 0|

1| 1| 1| RETURN vector

1| 1| 1| }

Total cost: 2n^2 + 5n + 6

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void refreshPrerequisites(){

1| 1| 1| DEFINE Course prerequisite

1| n| n| FOR course IN object.CourseList

1| n| n| FOR string IN course.prerequisiteStrings

n| n|n^2| SET prerequisite AS self.getCourseByID(string)

1| n| n| IF prerequisite:

1| n| n| APPEND object.courseList WITH prerequisite

1| 1| 1| }

Total cost: n^2 + 4n + 3

Runtime: O(n^2)

0| 0| 0| // Used to calculate a full list of Course dependencies

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| getFullPrerequisites(){

1| n| n| FOR course IN object.CourseList

n| n|n^2| SET course.fullPrerequisites AS course.getFullPrerequisites()

Total cost: n^2 + n + 1

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This looks up a Course using a string, or failing that,

returns NULL

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| course getCourseByID(string string){

1| n| n| FOR ALL course in object.courselist

1| n| n| IF course.id EQUALS string

1| 1| 1| RETURN course

1| 1| 1| RETURN NULL

1| 1| 1| }

Total cost (worst case): 2n + 2

Runtime: O(n)

cost|# executes |total cost

0| 0| 0| // This calls printInfo() for all courses in the vector

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void printCourseInformation(){

0| 0| 0| // We will assume given knowledge of quicksort and its implementation. Apparently Python and Java use Timsort O(n log n) as the default sorting algorithm, but I haven't implemented it before, so it would feel like cheating to use it

0| 0| 0| // SOURCE: https://en.wikipedia.org/wiki/Timsort

n^2| 1| 1| QUICKSORT() object.courseList ON KEY element.id

// SOURCE: https://en.wikipedia.org/wiki/Quicksort

0| 0| 0|

1| n| n| FOR course in flattenedList:

n| n|n^2| course.printInfo()

1| 1| 1| }

Total cost: 2n^2 + n + 3

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // Defines Hashtable objects

0| 0| 0| DEFINE class Hashtable:

1| 1| 1| DEFINE constructor Hashtable(tableSize):

1| 1| 1| SET object.tableSize AS tableSize

1| 1| 1| SET object.courseList AS empty vector <vector <Course>>

1| 1| 1|

1| 1| 1| // If the chosen language does not allow for populating directly

1| 4| 4| FOR number in tableSize

1| 4| 4| APPEND object.courseList WITH NULL

Total cost: 13 (For a hash table with size 4, which is the value I chose for the small testing size)

Runtime: O(1)

cost|# executes |total cost

0| 0| 0| // Drives high-level logic for creating a hash table of course objects

0| 0| 0| DEFINE STATIC METHOD

1| 1| 1| Hashtable load (dataStream dataStream) {

0| 0| 0|

0| 0| 0| // init hash table

1| 1| 1| hashtable = Hashtable(4) // FIXME: increase number

prior to release

0| 0| 0|

0| 0| 0| // Load course

1| 1| 1| SET Course course AS Course.nextCourse(dataStream)

0| 0| 0|

1| n| n| WHILE course EXISTS

1| n| n| SET hash AS hashtable.getHash(course.id)

0| 0| 0|

1| n| n| IF hashtable.courseList[hash] DOES NOT EXIST

1| n| n| SET hashtable.courseList[hash] AS

EMPTY vector <Course>

0| 0| 0|

1| n| n| APPEND hashtable.courseList[hash] WITH course

0| 0| 0|

1| n| n| SET course AS Course.nextCourse(dataStream)

0| 0| 0|

0| 0| 0| // Generate prerequisites

n^2| 1|n^2| CALL hashtable.refreshPrerequisites()

0| 0| 0|

0| 0| 0| // Generate full prerequisites

n^2| 1|n^2| CALL hashtable.refreshFullPrerequisites()

0| 0| 0|

1| 1| 1| return hashtable

1| 1| 1| }

Total cost: 2n^2 + 6n + 5

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This changes the prerequisites for each course to the initial state, then prunes orphan courses.

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void refreshPrerequisites(){

1| 4| 4| FOR ALL hash in object.courselist

1| n| n| FOR ALL course in hash

1| n| n| FOR ALL string in course.prerequisiteStrings

n| n|n^2| SET prerequisite AS self.getCourseByID(string)

1| n| n| IF prerequisite EXISTS

1| n| n| APPEND course.prerequisites WITH prerequisite

0| 0| 0|

1| n| n| ELSE

1| n| n| OUTPUT "ERROR! Attempted to import {course.id}

prerequisite {string} but failed to locate any

course with that ID"

1| 1| 1| }

Total cost (worst case): n^2 + 6n + 6

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This calls getFullPrerequisites() for all courses in the hashlist

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void refreshFullPrerequisites(){

1| 4| 4| FOR hash in object.courseList

1| n| n| FOR course in hash

n| n|n^2| SET course.fullPrerequisites AS

course.getFullPrerequisites()

1| 1| 1| }

Total cost: n^2 + n + 5

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This looks up a Course using a string, or failing that, returns NULL

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| course getCourseByID(string searchedId){

1| 1| 1| SET index AS object.getHash(searchedId)

0| 0| 0|

1| 1| 1| IF self.courseList[index] EXISTS

1| n| n| FOR ALL course in self.courseList[index]

1| n| n| IF course.id EQUALS string

1| 1| 1| RETURN course

1| 1| 1| RETURN NULL

1| 1| 1| }

Total cost (worst case): 2n + 5 (If all courses share the same hash. Noteably, a sufficiently large and balanced table would make this approach O(1)

Runtime: O(n)

cost|# executes |total cost

0| 0| 0| // This calls printInfo() for all courses in the hashlist

0| 0| 0| // I cannot seem to think of any way special way to

alphabetize a hashtable other than flattening it and

treating it as a vector

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void printCourseInformation(){

1| 1| 1| DEFINE flattenedList

1| 4| 4| FOR hash in object.courseList

1| n| n| FOR course in hash

1| n| n| APPEND flattenedList WITH course

0| 0| 0|

1| n| n| FOR course in flattenedList:

n| n|n^2| course.printInfo()

1| 1| 1| }

Total cost: 2n^2 + 3n + 3

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This generates a hash based off a given string

0| 0| 0| DEFINE PRIVATE INSTANCE METHOD

1| 1| 1| unsigned int getHash(String string){

1| 1| 1| SET int hash as 5381

1| 8| 8| FOR each char in string

1| 8| 8| MULTIPLY hash WITH 33

1| 8| 8| COMPUTE an int value from the char, then ADD to hash

1| 1| 1| SET hash AS hash mod object.tableLength

0| 0| 0|

1| 1| 1| RETURN absolute value of hash

1| 1| 1| }

Total cost: 29

Runtime: O(1)

cost|# executes |total cost

0| 0| 0| // Defines Tree objects

1| 1| 1| DEFINE class Tree:

1| 1| 1| DEFINE constructor Tree(node):

1| 1| 1| SET object.root AS node

Total cost: 3 (given data)

Runtime: O(1) (given data)

cost|# executes |total cost

1| 1| 1| Tree load (dataStream dataStream) {

0| 0| 0|

0| 0| 0| // initialize the tree

1| 1| 1| tree = Tree(Node())

0| 0| 0|

0| 0| 0| // Load course

n| 1| n| SET Course course AS Course.nextCourse(dataStream)

0| 0| 0|

1| n| n| WHILE course EXISTS

n| n|n^2| tree.root.insert(course)

n| n|n^2| SET Course course AS

Course.nextCourse(dataStream)

0| 0| 0|

0| 0| 0| // Generate prerequisites

n^2| 1|n^2| CALL tree.refreshPrerequisites()

0| 0| 0|

0| 0| 0| // Generate full prerequisites

n^2| 1|n^2| CALL tree.refreshFullPrerequisites()

0| 0| 0|

1| 1| 1| return tree

1| 1| 1| }

Total cost: 4n^2 + n + 4

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This method verifies that course prerequisites actually exist

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void refreshPrerequisites(node DEFAULTS TO NULL){

1| 1| 1| IF node does NOT EXIST: SET node AS object.root

0| 0| 0|

1| 1| 1| IF node.left EXISTS AND node.left.course EXISTS

n| n|n^2| object.refreshFullPrerequisites(node.left, root)

0| 0| 0|

1| 1| 1| IF node.right EXISTS AND node.left.course EXISTS

n| n|n^2| object.refreshFullPrerequisites(node.right, root)

0| 0| 0|

1| n| n| FOR each element in node.course.prerequisiteStrings

n| n|n^2| CALL Tree.getCourseByID(element) and SAVE OUTPUT AS match

1| n| n| IF match is found

1| n| n| APPEND node.course.prerequisites WITH match

1| n| n| ELSE

1| n| n| OUTPUT "ERROR! Attempted to import {node.course.id}

prerequisite {element}, but failed to locate any

course with that ID"

1| 1| 1| }

NOTE: The above line-by-line summaries don't fully capture the behavior of this method; the recursion obfuscates things

// In reality, the performance is similar to the other two data structures with essentially identical worst-case values

Total cost: n^2 6n + 5

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // This calls getFullPrerequisites() for all courses in the tree

1| 1| 1| DEFINE INSTANCE METHOD

1| 1| 1| void refreshFullPrerequisites(node DEFAULTS TO NULL){

0| 0| 0|

1| 1| 1| IF node does NOT EXIST: SET node AS object.root

0| 0| 0|

1| 1| 1| IF node.left EXISTS AND node.left.course EXISTS

1| n| n| object.refreshFullPrerequisitesTree(node.left)

0| 0| 0|

1| 1| 1| IF node.right EXISTS AND node.left.course EXISTS

1| n| n| object.refreshFullPrerequisitesTree(node.right)

0| 0| 0|

n| n|n^2| node.course.fullPrerequisites AS

node.course.getFullPrerequisites()

1| 1| 1| }

Total cost: n^2 + n + 6

Runtime: O(n^2)

0| 0| 0| // Used to calculate a full list of Course dependencies

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| getFullPrerequisites(){

1| n| n| FOR course IN object.CourseList

1| n| n| FOR prerequisite IN course

n| n|n^2| SET fullPrereqs AS prerequisite.getFullPrerequisites()

1| n| n| FOR prereq IN fullPrereqs

1| n| n| IF prereq NOT IN course.fullPrerequisites

1| n| n| APPEND course.fullPrerequisites WITH prereq

Total cost: n^2 + 5n + 1

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| // Matches a course object with a given string

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| Course getCourseByID(searchedID){

1| 1| 1| IF there is no course stored in the object.root node:

1| 1| 1| RETURN NULL

1| 1| 1| SET currNode AS object.root

0| 0| 0|

1| n| n| WHILE TRUE

1| n| n| IF currNode OR currNode.course do NOT EXIST

1| 1| 1| RETURN VOID

1| n| n| IF currNode.course.id EQUALS searchedId

1| 1| 1| RETURN currNode.course

1| n| n| ELSE IF currNode.course.id is LESS THAN

searchedId

1| n| n| IF there is a left node

1| n| n| SET currNode AS currNode.left

1| n| n| ELSE

1| 1| 1| RETURN NULL

1| n| n| ELSE IF currNode.course.id is GREATER THAN

searchedId

1| n| n| IF there is a right node

1| n| n| SET currNode AS currNode.right

1| n| n| ELSE

1| 1| 1| RETURN NULL

1| 1| 1| }

Total cost (case: element is at the bottom of a tree with no left nodes): 6n + 4

Runtime: O(n)

cost|# executes |total cost

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void print(){

1| 1| 1| IF node does NOT EXIST: SET node AS object.root

0| 0| 0|

1| 1| 1| IF this.left EXISTS

1| n| n| self.left.print()

n| n| n| self.printCourseInformation

1| 1| 1| IF this.right EXISTS

1| n| n| self.right.print()

1| 1| 1|

1| 1| 1| }

Total cost: n^2 + n + 6

Runtime: O(n^2)

cost|# executes |total cost

0| 0| 0| DEFINE class Node:

1| 1| 1| DEFINE constructor Course(course DEFAULTS TO NULL):

1| 1| 1| SET object.left AS NULL

1| 1| 1| SET object.right AS NULL

1| 1| 1| SET object.course AS course

Total cost: 4

Runtime: O(1)

cost|# executes |total cost

0| 0| 0| DEFINE INSTANCE METHOD

1| 1| 1| void insertNode(Course course)

1| 1| 1| IF this node's course exists

1| 1| 1| IF course.id is LESS THAN this node's course id

1| 1| 1| IF there is an empty node on the left

1| 1| 1| SET this node's left side AS new

node(course)

1| 1| 1| ELSE

1| n| n| call this.left.insertNode(course)

1| 1| 1| ELSE IF course.id is GREATER THAN this node's

course id

1| 1| 1| IF there is an empty node on the right

1| 1| 1| SET this node's right side AS new

node(course)

1| 1| 1| ELSE

1| n| n| call this.right.insertNode(course)

1| 1| 1| ELSE

1| 1| 1| SET this node's course AS course

1| 1| 1| }

Total cost (worst case)n + 6:

Runtime: O(n)

DEFINE METHOD

void Menu(){

DEFINE courses AS NULL

SET menu AS vector <string> {

"Menu:",

" 1. Load Courses",

" 2. Print Course List",

" 3. Print Course",

" 4. Exit",

"Enter choice:"

}

WHILE TRUE

FOR line in menu

OUTPUT line

GET user input AS input

IF input EQUALS "1"

SET inputStream AS language.open(courses.csv)

// Choose one:

//SET courses AS Vector.load(inputStream)

//SET courses AS Hashtable.load(inputStream)

//SET courses AS Tree.load(inputStream)

IF input EQUALS "2"

IF courses EQUALS NULL

OUTPUT "Please load courses before attempting to print them"

Menu()

RETURN

courses.printCourseInformation()

IF input EQUALS "3"

IF courses EQUALS NULL

OUTPUT "Please load courses before attempting to print them"

Menu()

RETURN

OUTPUT "Please enter the course ID (Example: MATH101):"

GET user input AS course

SET course AS courses.getCourseByID(course)

IF course EXISTS:

course.printInfo()

ELSE

OUTPUT "Course not found"

IF input EQUALS "4"

RETURN

}

DEFINE METHOD

void main(){

Menu()

}

**END PSEUDOCODE**

Again, Microsoft Word is unpredictable across machines and the above text may contain formatting errors. Please accept pseudocode.txt as the definitive submission of the pseudocode.

**Analysis:**

From the above pseudocode, along with line-by-line summaries of the runtime of various actions, much of complexity seems to come from the Course class methods which all three data structures share. Because each data structure will call these methods an equal number of times, we can cancel out their complexity when considering the performance of the three data structures. In my project, all three classes share these methods: getCourseByID(), refreshPrerequisites(), refreshFullPrerequisites(), print(), and load().

Let us start by examining the differences in the getCourseByID() methods:

**Searching: *getCourseByID()***

Vector:

cost|# executes |total cost

1| 1| 1| course getCourseByID(string string){

1| n| n| FOR ALL course in object.courselist

1| n| n| IF course.id EQUALS string

1| 1| 1| RETURN course

1| 1| 1| RETURN NULL

For the vector, the search is very simple: It will search through the list one-by-one in order before it stops to find one. While the worst case would be at the end of the list, the average would be near the center. Thus, it would have around a n/2 average time.

Hashtable:

cost|# executes |total cost

1| 1| 1| course getCourseByID(string searchedId){

1| 1| 1| SET index AS object.getHash(searchedId)

1| 1| 1| IF self.courseList[index] EXISTS

1| n| n| FOR ALL course in self.courseList[index]

1| n| n| IF course.id EQUALS string

1| 1| 1| RETURN course

1| 1| 1| RETURN NULL

The hash table adds a step to the vector by retrieving a hash value, but then looks up a sub-vector using the hash as an index, thus potentially skipping a large number of elements. The worst case would be that all courses are located in the same hash, so it would be equal performance to a vector, but in a suitably large and balanced table the runtime would approach O(1), which is a vast improvement.

Tree:

cost|# executes |total cost

1| 1| 1| Course getCourseByID(searchedID){

1| 1| 1| IF there is no course stored in the object.root node:

1| 1| 1| RETURN NULL

1| 1| 1| SET currNode AS object.root

1| n| n| WHILE TRUE

1| n| n| IF currNode OR currNode.course do NOT EXIST

1| 1| 1| RETURN VOID

1| n| n| IF currNode.course.id EQUALS searchedId

1| 1| 1| RETURN currNode.course

1| n| n| ELSE IF currNode.course.id is LESS THAN searchedId

1| n| n| IF there is a left node

1| n| n| SET currNode AS currNode.left

1| n| n| ELSE

1| 1| 1| RETURN NULL

1| n| n| ELSE IF currNode.course.id > searchedId

1| n| n| IF there is a right node

1| n| n| SET currNode AS currNode.right

1| n| n| ELSE

1| 1| 1| RETURN NULL

For a binary tree, it nearly behaves like a linked list in the worst-case scenario, which would be moving through each element one-by-one in an all-left or all-right sided tree. In this way, the worse case is identical to the vector. However, in the average case, in a balanced binary search tree, each step in the process eliminates half of the search candidates. Because of this behavior, it would have an average efficiency of O(log(n))

**Traversal and general operations**: *refreshPrerequisites(), refreshFullPrerequisites(), print()*

The refreshPrerequisites(), refreshFullPrerequisites() and print() methods all involve fully traversing through the data structure and calling a method on each course within the data structure. Because of the similarities between these three methods, we only need to examine one of them. As it happens, refreshFullPrerequisites() is usually the shortest, so we will examine that in detail:

Vector:

cost|# executes |total cost

1| 1| 1| getFullPrerequisites(){

1| n| n| FOR course IN object.CourseList

canceled SET course.fullPrerequisites AS course.getFullPrerequisites()

Total cost: n + 1

Runtime: “O(n)”

Hashtable:

cost|# executes |total cost

1| 1| 1| void refreshFullPrerequisites(){

1| h| h| FOR hash in object.courseList

1| n| n| FOR course in hash

canceled SET course.fullPrerequisites AS course.getFullPrerequisites()

Total cost: n + 1 + h

Runtime: “O(n)”

Tree:

cost|# executes |total cost

1| 1| 1| void refreshFullPrerequisites(node DEFAULTS TO NULL){

1| 1| 1| IF node does NOT EXIST: SET node AS object.root

1| 1| 1| IF node.left EXISTS AND node.left.course EXISTS

1| n| n| object.refreshFullPrerequisitesTree(node.left)

1| 1| 1| IF node.right EXISTS AND node.left.course EXISTS

1| n| n| object.refreshFullPrerequisitesTree(node.right)

canceled SET node.course.fullPrerequisites AS

node.course.getFullPrerequisites()

Total cost: n + (n-1)

Runtime: “O(n)”

After simplifying these methods, it seems like their basic performance is essentially identical, with the vector slightly outperforming the rest. Hashtable has extra steps directly proportional to the size of the table. If it’s a wide table that means fewer iterations through each element in the hash; contrariwise if the table is deep then there will be fewer hashes to check through. The tree has some added steps of logically checking if the next nodes exist, which could end up scaling to n-1 in the worst case.

**Alphabetization***:* ***print()***

In storing the course data, my code makes no attempt to keep them in an alphabetized state. Because the Tree datastructure uses the plaintext course number as a key, the tree datastructure happens to be capable of outputting an alphabetized list of courses with nearly the same runtime as it takes to traverse the tree. This decision will lead to the tree structure to be sub-optimally balanced, which affects traversal time. For the vector and hashtable structures, my code requires the use of a sorting algorithm (I used quicksort, but there are better options which are more complicated). The addition of this sorting algorithm, coupled with the fact that my code does not retain alphabetization, causes the worst case runtime of the print() command for the vector and hashtable to increase to O(n^2). The only reason the runtime of tree.print() is o(n) is due to a design decision which enables an alphabetical traversal.

Both vector.print() and hashtable.print() runtimes could be improved to O(n) if the data are stored in an alphabetical manner. This could be achieved for vectors by calling the sorting algorithm each time a new batch of Courses is imported. Because universities infrequently add courses to the roster, perhaps at most three times per year, such an increase in runtime of importing courses would be acceptable. For a hashtable, there are some expanded options. Some implementations of this type of structure enable strings to be used as keys and enable these keys to be naturally stored and retrieved in an alphanumeric order. If such a structure is an option, this would enable a reduced course import runtime and enable an O(n) runtime for hashtable.print(). Failing this, hash keys could be stored in a vector, which could in turn be alphabetized in the manner described above.

**Instantiation: *load()***

Finally, let us examine the process to load the courses into memory (again, canceling out common method calls and referencing previous simplifications):

Vector:

cost|# executes |total cost

1| 1| 1| Vector load(dataStream dataStream){

1| 1| 1| DEFINE vector as Vector(Vector <Course>)

1| 1| 1| DEFINE Course course

1| n| n| WHILE TRUE:

|Canceled| SET course AS Course.nextCourse(dataStream)

1| n| n| IF course EXISTS

1| n| n| APPEND vector.courseList.append WITH course

1| n| n| ELSE

1| 1| 1| break

| “O(n)” | CALL vector.refreshPrerequisites()

| “O(n)” | CALL vector.refreshFullPrerequisitesTree()

1| 1| 1| RETURN vector

Total cost: 6n + 5

Runtime: “O(n)”

Hashtable:

cost|# executes |total cost

1| 1| 1| Hashtable load (dataStream dataStream) {

1| 1| 1| hashtable = Hashtable(4)

|Canceled| SET Course course AS Course.nextCourse(dataStream)

1| n| n| WHILE course EXISTS

1| n| n| SET hash AS hashtable.getHash(course.id)

1| n| n| IF hashtable.courseList[hash] DOES NOT EXIST

1| n| n| DEFINE hashtable.courseList[hash]

1| n| n| APPEND hashtable.courseList[hash] WITH course

|Canceled| SET course AS Course.nextCourse(dataStream)

| “O(n)” | CALL hashtable.refreshPrerequisites()

| “O(n)” | CALL hashtable.refreshFullPrerequisites()

1| 1| 1| return hashtable

Total cost: 7n + 3

Runtime: “O(n)”

Tree:

cost|# executes |total cost

1| 1| 1| Tree load (dataStream dataStream) {

1| 1| 1| tree = Tree(Node())

|Canceled| SET Course course AS Course.nextCourse(dataStream)

1| n| n| WHILE course EXISTS

n| n|n^2| tree.root.insert(course)

|Canceled| SET Course course AS Course.nextCourse(dataStream)

| “O(n)” | CALL tree.refreshPrerequisites()

| “O(n)” | CALL tree.refreshFullPrerequisites()

1| 1| 1| return tree

Total cost: n^2 + n + 3

Runtime: O(n^2)

From these simplified costs, we can see that vector is the least expensive to import new courses, but is followed closely by hashtable. The reason hashtable.load() has a slightly higher cost is because the hashtable.gethash() (O(1)) increases the runtime by n because it is called n times. The worst performance comes from the tree structure because each time a new course is imported there must be a partial traversal of the tree until an open spot is found.

As stated in the alphabetization section, there is an option to alphabetize the vector and hashtable structures whenever new courses are loaded. Such an increase using a O(n^2) algorithm would increase both of these runtimes to similar levels of the tree structure, unless there exists a better alternative for storing the hashtable (as previously noted).

**Conclusions:**

Due to the infrequency of adding new courses, I do not believe the runtime of initializing the data structure should play a major decision. I also do not believe that storage or memory space is any great concern because the number of courses will not be difficult to hold in memory; I believe we can safely assume that *n* will be less than 10,000 for any existing university. Also, because of the limited number of courses, I do not believe that the decision between either of these three is very consequential; I believe all three will have sufficient performance for the given task. However, a decision must ultimately be made; I believe that a hashtable will offer the best performance for course searching without any notable sacrifice to traversal time. I believe that alphabetization can be achieved at time of course import, thus allowing for high performance output of an alphabetical list.