Brandon Randall

CS-300

8/6/23

Menu Pseudocode

This menu will work for all 3 data structures so I am going to use Data Structure courses as a cover all.

int menu(String csvPath) {  
 create a new data structure, courses

create int choice = 0

WHILE(choice does not equal 4) {

print "Menu:" then new line

print " 1. Load courses" then new line

print " 2. Print Course List" then new line

print " 3. Print Course" then new line

print " 4. Exit" then new line

print "Enter choice: "

assign user input to choice

SWITCH(using choice as a parameter) {

case 1:

call loadCourses method using csvPath as a parameter to load courses from CSV into courses data structure  
 break

case 2:

Call printSampleSchedule using courses to print all values

break

case 3:

print “enter course number” then new line

create int courseID = to user input

call printCourseInformation using courses and courseID

break

case 4:

set choice = to 4

}

}

print "Good bye." then new line

return 0

}

**Runtimes**

**(Assume anything that does not have a O notation next to it is constant time.)**

Like my pseudocode for loading courses, they are all similar.

**Vector:**

set numPrerequisites = to 0

Initialize Vector<Course> courses

Create a Parser object, file, to parse the provided file using the csvPath string

Use a Try/Catch statement to ensure the file can be read properly   
 TRY {

FOR( int I equals 0, I is less than the number of rows in the file, increment I) { (**O(n))**

IF ( file [I] [0] is empty OR file [I] [1] is empty){

print out that the file is improperly formatted on row I.

CONTINUE to the next row

}

initialize Course course.

assign course ID variable with file [I] [0]

assign course Name variable with file [I] [1]

IF ( file [I] [2] is not empty) {

initilize int k equals 2

initialize vector<string> prerequisites\_t

WHILE( file [I] [k] is not empty) { (**O(n))**

FOR( int J equals 0, J is less than the number of rows in the file, increment J) { (**O(n))**

IF( file [j] [0] is the same as file [I] [k] ) {

pushback file [I] [k] to prerequisites\_t

break the loop

}

}

increment k

set course prerequisites equal to prerequisites\_t

add prerequisites size to numPrerequisites

}

}

add course to courses using insert method

}

CATCH{

IF the file does not open correctly, catch an error

}

IF (courses is empty ) {

print out that the file is improperly formatted or empty.

}

sort courses (**O(n^2))**

return courses

}

Within this there’s 1 overarching FOR loop, then a nested WHILE loop within an IF statement, and finally a nested FOR loop within that loop. Assuming worst case we are at O(n\*n\*n) or O(n^3). Most other statements within the algorithm are on constant time at O(1), with the exclusion of the insert statement, meaning without the insert statement we have O(1) \* O(n^3) = O(n^3). This is the same for all three data structures because the same algorithm was used to add courses to the data structure. Where things may change is in the insertion methods.

For a vector I would add the objects then run a quicksort over them. adding the courses with Push\_Back would be O(1) within the O(n^3) loops, so nothing changes there. A quicksort would be O(log base 2 \* n). This sort is faster because it utilizes recursion to cut n in half after every iteration. Typically, a quicksort would have a worst case of O(n^2). So overall, the runtime would be O(n^3) + O(n^2) leading to a worst case of O(n^3).

Quicksort algorithm psuedocode:

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

initialize the midpoint equal to 0.

IF begin is greater than or equal to end {

This would mean either the vector has 1 or no elements or the low value is greater than the high value, which would affect the outcome. End the method

here.

}

set midpoint equal to the middle of the partitioned vector courses. (Partition(courses, begin, end)) **(O(n^2))**

call quickSort to sort the low partitioned valued from beginning to midpoint

(quickSort(courses, begin, midpoint))

call quickSort to sort the high partitioned valued from midpoint + 1 to end

(quickSort(courses, midpoint + 1, end))

**(Both of these calls are O(Log base 2 n) because they both divide n by 2.)**

}

**(This would be O(n^2) + O(log base 2 n) = O(n^2))**

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

initialize low equal to begin.

initialize high equal to end.

initialize pivot as the midpoint between low and high. ( low + (high - low)/2)

initialize done as false.

WHILE done is false { (**O(n))**

WHILE the title of the course at index low is less than the title of the course at index pivot { **(O(n))**

incriment low.

}

WHILE the title of the course at index pivot is less than the title of the course at index high { **(O(n))**

decrement high.

}

IF low is greater than or equal to high {

set done to true.

}

ELSE {

swap the course at index low with the course at index high.

increment low

decrement high.

}

}

return the value of high.

}

**(Overall worst-case runtime of O(n) \* O(n+n) = O(n) \* 2O(n) = O(n) \* O(n) = O(n^2))**

**Hash Table:**

HashTable<course> loadCourses (string csvPath) {

set numPrerequisites = to 0

HashTable\* hashTable = new Hashtable\*;

Create a Parser object, file, to parse the provided file using the csvPath string

Use a Try/Catch statement to ensure the file can be read properly  
 TRY {

FOR( int I equals 0, I is less than the number of rows in the file, increment I) { (**O(n))**

IF ( file [I] [0] is empty OR file [I] [1] is empty) {

print out that the file is improperly formatted on row I.

CONTINUE to the next row

}

initialize Course course.

assign course ID variable with file [I] [0]

assign course Name variable with file [I] [1]

IF ( file [I] [2] is not empty) {

initilize int k equals 2

initialize vector<string> prerequisites\_t

WHILE( file [I] [k] is not empty) {(**O(n))**

FOR( int J equals 0, J is less than the number of rows in the file, increment J) {(**O(n))**

IF( file [j] [0] is the same as file [I] [k] ) {

pushback file [I] [k] to prerequisites\_t

break the loop

}

}

increment k

}

set course prerequisites equal to prerequisites\_t

add prerequisites size to numPrerequisites

add course to hashTable using the insert method.

}

CATCH{

IF the file does not open correctly, catch an error

}

sort HashTable

return HashTable

}

Just like with the Vector, the main FOR loop, including the insert method, is O(n^3). The insert method for a HashTable does not have any loops. The insertion method also works to sort the Table by utilizing the hash value of the node key. This way all added course nodes will be added to the internal vector in order based on their hash value.

void HashTable::Insert(Course course) {

// create the key for the given course

int key = hash(atoi(course.courseId.c\_str()));

// retrieve node using key

Node\* currNode = &(nodes.at(key));

// if no entry found for the key

if (currNode == nullptr){

// assign this node to the key position

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

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

}

else {

// else if node is not used

if (currNode->key == UINT\_MAX) {

//set currNode key and course to provided values and the next value to null

currNode->key = key;

currNode->course = course;

currNode->next = nullptr;

}

}

}

**Overall O(1). There are no loops or outside functions to call, everything is in constant time.**

Again, just like with a vector, the whole functions runtime is O(n^3).

**Binary Tree:**

BinarySearchTree\* loadCourses (string csvPath) {

set numPrerequisites = to 0

BinarySearchTree\* BST = new BinarySearchTree();

Create a Parser object, file, to parse the provided file using the csvPath string

Use a Try/Catch statement to ensure the file can be read properly  
 TRY {

FOR( int I equals 0, I is less than the number of rows in the file, increment I) {(**O(n))**

IF ( file [I] [0] is empty OR file [I] [1] is empty) {

print out that the file is improperly formatted on row I.

CONTINUE to the next row

}

initialize Course course.

assign course ID variable with file [I] [0]

assign course Name variable with file [I] [1]

IF ( file [I] [2] is not empty) {

initilize int k equals 2

initialize vector<string> prerequisites\_t

WHILE( file [I] [k] is not empty) { (**O(n))**

FOR( int J equals 0, J is less than the number of rows in the file, increment J) { (**O(n))**

IF( file [j] [0] is the same as file [I] [k] ) {

pushback file [I] [k] to prerequisites\_t

break the loop

}

increment k

}

set course prerequisites equal to prerequisites\_t

add prerequisites size to numPrerequisites

add course to BST using the insert method.

}

CATCH{

IF the file does not open correctly, catch an error

}

return BST

}

A Binary Tree is similar to a HashTable in that it uses nodes and it’s insert method sorts it.

void BinarySearchTree::Insert(Course course) {

// if root equal to null ptr

if (root == nullptr) {

// root is equal to new node course

root = new Node(course);

}

else {

// add Node root and course

addNode(root, course);

}

}

void BinarySearchTree::addNode(Node\* node, Course course) {

// if node is larger then add to left

if (node->course.courseId.compare(course.courseId) > 0) {

// if no left node

if (node->left == nullptr) {

node->left = new Node(course);

}

else {

//recurse down the left node

addNode(node->left, course);

}

}

else {

// if no right node

if (node->right == nullptr) {

// this node becomes right

node->right = new Node(course);

}

else {

// recurse down the right node

addNode(node->right, course);

}

}

}

**Just like with a HashTable, this is also O(1) for insertion and sort. The overall runtime will again, be O(n^3)**

So, for the insertion and sorting of classes, we have an equal runtime notation of O(n^3), even though the Hashtable and Binary Tree insertion methods are faster than the vectors.

**Next up is counting prerequisites.**   
I made algorithms for these functions initially, but all three data structures could use the same O(1) notation algorithm. A class variable int numPrerequisites is need to be created and then when prerequisites are assigned during the load function, we add the size of the prerequisite vector to this variable. Then we have the numPrerequisites function return this variable:

numPrerequisites(){

return numPrerequisites

}

**This would all be O(1) notation and would not affect the load functions runtime as well.**

**Next we look at the print Course function.**

**Vector:**

void printCourse(vector<course> courses, string courseID) {

FOR( course in courses ){ **(O(n))**

IF (courses ID is the same as courseID) {

print course id

print course name

FOR( int I = 0, if I is less than course prerequisites length, increment I) { **(O(n))**

print prerequisite.

}

}

}

}

**There is an overall FOR loop and then a nested FOR loop as well. Assuming worst case, the big O notation for this is O(n \* n) O(n^2).**

**HashTable:**

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

create int Key and assign it the hash value of the number value of courseNumber

create a new node, currNode, and assign it to the node of courses(Key)

print information of currNode’s course

}

**Overall, this is in constant time, O(1. There are no loops or functions that add time.**

**Binary Tree:**

void printCourseInformation(Tree<Course> courses, String courseNumber) {

create a new node, currNode, and assign it the value of courses root

While(currNode is not null){ **(O(n))**

IF(currNode’s course’s ID is larger than courseNumber){

currNode = left node from currNode

}

IF(currNode’s course’s ID is smaller than courseNumber){

currNode = right node from currNode

}

}

print currNode course information

}

**Overall, this is O(n) due to the while loop.**

For this function, HashTables are the best.

**Lastly we have the Print Schedule function.**

**Vector:**

void printCourses(vector<course> courses, string courseID) {

FOR( course in courses ){ **(O(n))**

print course id

print course name

FOR( int I = 0, if I is less than course prerequisites length, increment I) { **(O(n))**

print prerequisite.

}

}

}

**This is essentially the same as the print course function without the if statement. The notaton is the same O(n).**

**HashTable:**

void printSampleSchedule(Hashtable<Course> courses) {

create a new node, currNode, and assign it the value of the first node in courses

WHILE ( currNode is not null) { **(O(n))**

print course information within currNode

set currNode equal to next node after currNode

}

}

**The while loop makes this O(n).**

**Binary Tree:**

void printSampleSchedule(Tree<Course> courses) {

utilize an inOrder method to print course information from courses **(O(n))**

}

**The inOrder method would be a recursive method that does not change the value of n. It would essentially be a for each loop to iterate through the tree from first value to last. This makes it O(n).**

void inOrder(Node\* root) {

if (node != nullptr) {

inOrder(node->left) **(O(n))**

output course information

inOrder(node->right) **(O(n))**

}

}

**O(n) + O(n) = O(n)**

**Evaluation**

**Vector: O(n^3) / O(1) / O(n) / O(n)**

**HashTable: O(n^3) / O(1) / O(1) / O(n)**

**Binary Tree: O(n^3) / O(1) / O(n) / O(n)**

Based on the needs and the Big O notation evidence of the algorithms, I would recommend using a HashTable for this project. While all three data structures have the same times for Loading, counting prerequisites, and schedule printing, the hash table is faster for finding single courses. While this does not make it superior to the others in every way, it does give it an advantage. Outside of runtime, hash tables are simple to sort, search, and print information from. Vectors are simple, but even If I were to reevaluate the loading algorithm it would still have a worst case of O(n^2) to add and sort the courses, even if a different sort technique was used. Vectors also require more runtime to add and delete values that are not in the back of the vector. Anytime a value is added or deleted outside of the back of the vector, every other value behind that value is also shifted, thus adding and deleting in a vector always has a worst case of O(n). hash tables do utilize a vector to store values, but these values are not stored directly behind eachother, with indexes of 0,1,2,3,..., they are stored based on the hash value of the key, with indexes such as 1, 24, 63, 45, ... Since these values are not added to the vector, but rather assigned to the vector at a specific index, it does not shift the vector the same way as a normal vector insert would. Binary trees work similarly, by creating nodes and changing what they point to, rather than storing them in another structure. Where binary tree’s lack over hash tables is in their access speed. While a hash table can be accessed using a key, leading to a constant time access speed, a binary tree is like a vector in that it needs to be “iterated” over to find a value.