Dawneen Graham

CS300

Mod 6:2

Runtime Evaluation

So, for this one, I thought it would be neat if the user could choose between a vector, hash table, or a tree. I didn’t choose to do this for my final project, I went with a simple vector because it was an easier choice with the small number of courses. If one was wanting data presented in alphanumerically then I believe a binary tree would be most beneficial. It takes O(log(n)) time to find an item in a binary tree, after printing, it would be O(log(n) + m) (explained more in depth below.) If someone was wanting to have overall greatest time efficiency. However, if you wanted to have the data in a specific order (alphabetically by courseNumber or courseName) then a tree would be more efficient since it sorts items during insertion while you would have to sort the entire hash table separately.

bool isVector = 0;

bool isTable = 0;

bool isTree = 0;

int main(){

//We need some kind of menu

//Ask the user for a filename

//Print a menu asking which data structure the user would like to use (vector, hashtable, tree)

int choice = 0;

cout << "Choose an option for which data structure to use \n1. Vector \n2. Hashtable \n3. Binary Tree"

cin >> choice;

switch(choice){

case 1:

isVector = 1;

break;

case 2:

isTable = 1;

break;

case 3:

isTree = 1;

break;

default:

cout << "Not a valid input" << endl;

}

//Based on that input, within the parser, store the data into the corresponding data structure using the file the user inputted

//PS: Would probably use Typenames or an Abstract Base Class (ABC) or something to get around having to rename the functions differently for each structure

//Print a sample schedule (printSampleSchedule) or if the user wants to print information for a single course (printCourseInformation) or exit the program

//If printSampleSchedule, just print the schedule

//If printCourseInformation, ask for the courseNumber

//If the user enters "exit", we could also just have them press 0 or something, then the program should just continue to return 0 and finish up

return 0;

}

//Library for making a file into a stream so it can be parsed using getline, etc.

#include <fstream>

//Library for doing the same as above but with a string instead of using a file

#include <sstream>

using namespace std;

//Assume some Data Structure named "data" already exists as part of the same class

//Assume there is a class for courses, called "Course" that has a string courseNumber,

//a string name, and a vector of strings of prerequisites

//Need filename because we need to know which file we're using

void csvParser(string fileName){

//Make a stream of the current file so that it can be parsed

fstream file(fileName); //+1

//

if(file.is\_open()){

//Making a vector so I can check the prerequisites for each course and make sure they exist

vector<Course> checkCourses; //+1

//The current line I'm on of the file

string line; //+1

//Gets rid of the header before we start parsing

getline(file, line); //+k, where k is the length of the string

//A while loop that goes line-by-line through the file

while(getline(file, line)){

string courseNumber; //+1

string name; //+1

vector<string> prerequisites; //+1

//Make a stream out of "line", the current line, so that you can call getline on it and split it into parts easily

stringstream currLine(line); //+k, where k is the length of the string

//get all the line before the first comma

getline(currLine, courseNumber, ','); //+k, where k is the length of the string

//make sure there was actually a value, and throw an error if there wasn't

if(courseNumber.empty()){

throw new runtime\_error("There's not at least 2 parameters"); //+1

}

//The first time this runs, I should be getting the name of the course, otherwise, I want to add a prerequisite

bool firstLoop = true; //+1

//The string I'll use to store each part between commas, one part at a time

string breakup; //+1

//sub while loop that runs through the current line, comma-by-comma

while(getline(currLine, breakup, ',')){ //+k, where k is the length of the string

//Again, getting the name the first time

if(firstLoop){

name = breakup; //+1

}

//If not the first loop, push back a new prerequisite

else{

prerequisites.push\_back(breakup); //+1

}

//Make sure firstloop only runs one time. So set it to false to stop it after the first loop. Self-explanatory

firstLoop = false; //+1

} //Runs n times, for each course, since each course is one line, in total this while loop costs O((1+1+1+k) \* n) = O(3k\*n) = O(kn)

//Again, just making sure that name actually has a value, like how I did with the coursenumber

//If not, there wasn't enough parameters and use the same error as above

if(name.empty()){

throw new runtime\_error("There's not at least 2 parameters"); //+1

}

//Create a new course using the info from the line

Course course(courseNumber, name, prerequisites); //+1

//insert this new course into the data structure

data.insert(course); //Depends on the data structure, assuming a tree it costs O(log(n)) to insert into a tree

//also insert this new course into the checker for later

checkCourses.push\_back(course); //+1

} //In total, this costs O(kn + log(n) + 9 + 2k) = O(kn + log(n))

//Goes through every course

for(auto course : checkCourses){

//Looks through the prerequisites in each course //+1 n times

for(auto prerequisite : course.prerequisites){

//For each prerequisite, make sure that prerequisite can be found in the data structure //O(log(n)) to find in a tree, happens m times for m prereq's

//If not, it should throw an error since it's invalid //+1

//This is done to verify every prerequisite is a valid entry, you will NOT need to use this vector ever again (unless you want to)

if(!data.find(prerequisite)){

throw new runtime\_error(prerequisite << " is not a course"); //+1

}

} // total = O((log(n) + 1) \* m) where m is the number of prerequisites in a course = O(mlog(n) + m) = O(mlog(n))

} total = O(mlog(n) \* n)

} //Total = O(n\*mlog(n) + kn + log(n)) = O(n\*mlog(n) + kn) where k is the string length of each line, and m is the amount of prerequisites for each of n courses

}

void printSampleSchedule(Vector<Course> courses){

//Run a sorting algorithm to order the vectors alphanumerically += log(n)

//(The best sorting algorithms take O(nlog(n)) time worst case where n is the amount of items)

//for all n courses in my courses vector

//Print the course info += 1 cost, happens n times in the for loop. So in total + n

//Final runtime = O(nlog(n) + n\*1) = O(nlog(n))

}

void printSampleSchedule(Tree<Course> courses){

//A tree is already stored in order alphanumerically if you used the courseNumber as the value to compare while inserting

//Check if the root is null and if it is just return 0, costs + 1

//If the left node is not empty + 1

//numPrerequisiteCourse(The\_node\_to\_the\_left) + overall complexity

//Print current course's name, putting it after visiting left and before visiting the right prints the tree in order = +1 cost

//If the right node is not empty

//numPrerequisiteCourse(The\_node\_to\_the\_right) + overall complexity

//Recursively gets each node and its children in order climbing down on the left then back up to the root then to the right side of the tree

//This only happens one node at a time for each node, the recursive call makes sure each node is reached exactly once

//Basically, a function with a cost of 1, is being called for n nodes

//In total = O(n\*1) = O(n)

}

void printSampleSchedule(HashTable<Course> courses){

//A hashtable is not sorted automatically, this hashtable has lists as each spot in the table

//So get all n items from EVERY list in the hashtable and put them in a vector = n time because you get each item once += n

//Then call a sorting method on this vector, this will costs O(nlog(n)) time for the best algorithm worst case += nlog(n)

//Then, for each item in sorted vector

//Print item += 1 cost, happens n times

// += n

//In total O(n + nlog(n) + n) = O(nlogn + 2n) = O(nlogn)

}

Because we have to print our classes in alphanumeric order, and trees are already sorted, I would recommend using a tree for this program if it had more courses to choose from.

The overall average of the worst cases is better for the tree program than the hash table or vector programs.