CS11 HW9: Advising Period

In this assignment, you will continue developing your recursive problem-solving and programming skills by focusing on a truly recursive data structure: binary search trees! You'll also get some practice reasoning about complexity (feel free to review the complexity slides if you need a refresher).

The starter code, data files, and provided input testing files for this assignment can be retrieved by entering the following command:

```
▶ pull-code11 hw09
```

Learning Objectives

The purpose of this assignment is to

- Practice analyzing and thinking about complexity
- Live more of that sweet binary search tree life
- Practice designing and implementing recursive solutions
- Get a (somewhat fabricated) glimpse into the professional lives of your CS11 professors

Written

For the written portion of this assignment, you will answer some questions concerning complexity. Your responses to the following questions must be typed, saved as `written.pdf`, and uploaded to Gradescope.

1. List the following Big-O time complexities from the fastest to the slowest:
   O(`n` log `n`), O(n^2), O(1), O(2^n), O(log `n`), O(`n`)
2. Does Big-O complexity describe an upper bound, an average bound, or a lower bound?

For each of the following equations, please write the Big-O complexity that they would equate to:

3. $n^3 + 16n^2 + 2n + 5$
4. $4\log n + 2n^2 + n\log n$

For the following code snippets, please compute the Big-O time complexity with respect to the value of `n`:

5.
```
string mystr = "bigsecret";
int n = mystr.length();

for(int i = 0; i < n; i++){
    mystr[i] = mystr[i] + 3;
}
```

6.
```
string mystr = "mysteriesarefun!";
int n = mystr.length();

for(int i = 1; i < n + 1; i *= 2){
    cout << mystr[i - 1] << endl;
}
```

7.
```
string mystr = "dupeletters";
int n = mystr.length();

for(int i = 0; i < n; i++){
    for(int j = i+1; j < n; j++){
        if(mystr[i] == mystr[j]){
            cout << "Duplicate!!" << endl;
        }
    }
}
```

Programming

Point breakdown: 5 for compiling, 20 for passing the provided tests, 45 for passing our additional tests (some of these additional tests just test a single query; the points to be gained from those are specified next to each query title below), 10 for passing valgrind on every test with no errors and no "definitely lost" memory

After a couple hectic weeks of midterms, nothing calms the nerves like.....*advising period*. The blissful serenity of cramming 50 life-coaching sessions into a two-week period is exactly the sort of restful salve that we all needed. This year, the faculty responsibilities have been divided into two distinct roles: advising and supervising. Fun!

Here's the plan: some faculty members will directly advise students, and some faculty members will supervise up to two other faculty members. These roles (advising and supervising) will never be mixed. That is to say, if a faculty member is assigned advisees, they will not supervise any faculty members. If a faculty member is supervising other faculty members, they will not be assigned advisees.

One last thing: The person charged with assigning these roles LOVES alphabetizing things. Meticulous care has been taken to ensure that, for every faculty member in a supervisory role, the name of their first supervisee (and all subsequent supervisees of that supervisee) will alphabetically *precede* their own. Similarly, the name of their second supervisee (and all subsequent supervisees of that supervisee) will alphabetically *follow* their own. Thus, if Karen is a supervisor, James could potentially be her first supervisee, but not her second. Bureaucracy, amirite?!

What we've provided

The provided starter code in `advising.cpp` does the heavy lifting of reading in a data file provided as a command-line argument, and building a binary tree data structure to hold that data. Much like your `mutations` and `lineage` programs, `advising.cpp` allows the user to perform a series of queries. We have provided two data files, `advising1.data` and `advising2.data`, for you to work with. We have also provided a working "print" query, executed by typing '`p`' at the query prompt, which prints the supervision hierarchy for every group of advisees (so if a faculty member has no advisees, directly or indirectly, they will not be

printed out). Give it a whirl!

A discerning eye will realize that the starter code is providing a blueprint for how to use this assignment's data structure. **Make sure you read and understand the starter code before attempting to implement new queries.** For testing, we have provided a demo program which can be run as follows:

- `advising_demo advising1.data`

The demo program can be used to create ground truth files to `diff` against. Finally, we have provided test input files `advising_test1.in` (which goes with data file `advising1.data`) and `advising_test2.in` (which goes with `advising2.data`), which use many of the queries described below. It is in your best interest to write other, simpler test input files so you can test a single query at a time!

Your program

Your program must support three new queries. You will implement these queries by creating functions and filling in the "`YOUR CODE HERE`" sections of the query prompt loop in `main`. You are **required** to implement the first query, Total Advisees, recursively. However, you will likely find that *all* of these queries are best implemented with a recursive solution:

Total Advisees (20 points): This query is executed by typing '`t`' at the query prompt, followed by the name of the faculty member whose advisees should be counted. Your program should then print the total number of student advisees under that faculty member. This includes not only direct advisees, but also advisees of other faculty members that the specified faculty member is supervising (either directly or indirectly). **You are required to implement this query recursively.**

Slackers (20 points): This query, executed by typing '`s`' at the query prompt, prints the name of any faculty member who both has no direct advisees *and* is not supervising any other faculty members (literally what are they doing?!). These names should be printed in alphabetical order.

Quit: This query, executed by typing '`q`' at the query prompt, quits the program. BUT WAIT! The *idiot* who wrote your starter code forgot to free any of the memory that they used. You must finish the job so that the resulting program can pass `valgrind`. That is, `valgrind` should report no errors and no "definitely lost" memory when you run the following command (with either data file provided as a command-line argument):

- `valgrind ./advising advising1.data`

Optional starter queries

If you would like to get practice working with this data structure before you tackle the recursive queries, we recommend implementing the following two (optional) starter queries. Note that you don't have to implement these at all (that's why they're optional), and if you do implement them, they aren't worth any extra credit. They are simply here for you to get more practice working with binary trees before tackling the required queries above:

Add Advisee: This query is executed by typing '`a`' at the query prompt, followed by the name of the faculty member who you would like to assign an advisee to. So long as this person is not already supervising other faculty members, your program should increment their advisee count and print a success message. If the faculty member is already in a supervisory role, your program should print a failure message and return to prompting for queries.

Add Supervisee: This query is executed by typing '`f`' at the query prompt, followed by two arguments: the name of the faculty member to whom you'd like to assign a supervisee, and the name of the supervisee that you are assigning them. This query is very similar to the previous one, but has many more edge cases to think through. Consult the demo program for guidance.

Requirements and expectations for your solutions

Here are the requirements that your code must follow. *Violating any of these requirements will result in a score of 0 points for a functional correctness portion of this assignment.* Make sure that you check your work against these requirements well before the submission deadline!

- Your program must be valid C++ code and compile successfully. Specifically, running the following command on the Halligan servers must produce no error messages:
  ```
  ▶ g++ -o advising -Wall -Wextra advising.cpp
  ```
- Your solution may not include any external packages other than `<iostream>`, `<fstream>`, `<string>`, and `<stdlib.h>`.
- You should only use exit() to quit the program if a file name wasn't provided on the command line, or if something goes wrong when you try to open the file named. **You may not use exit() in any other case, like for the quit query.**
- Make sure you can run and diff your program using the exact commands shown above. Too often students assume that their program should take in input in a way that differs from these commands, leading to many tests failing; it is your responsibility to make sure you're following our directions!

Here are our expectations for your code. *Ignoring any of these expectations may result in an unsatisfactory grade for either a functional correctness portion or a style portion of the assignment.* Feel free to ask a member of the course staff to look at your work if you're not sure whether it meets an expectation here.

- Your `advising.cpp` program must implement the Total Advisees query with a recursive function. No loops may appear in that functions or any functions called by that recursive function.
- All functions you write (except `main()`) must be preceded by a function contract.
- No function (except the functions provided in the starter code) should be larger than 30 lines (a few lines longer isn't a huge deal).
- You should use `diff` to test your programs, and only feel confident about passing those tests if `diff` does not produce any output whatsoever.
- Your programs should begin with a header comment with your name, the name of the program file, the date you started working on it, and a succinct description of the program.
- Follow the typical line length, indentation, naming, and commenting conventions covered in the course style guide.

How to organize and submit your homework

Your submission for this assignment will comprise these files:

- A PDF file `written.pdf` containing your responses to the written questions.
- A file `advising.cpp` containing your solution to the programming problem.

- A file `README` identifying yourself, anyone with whom you have collaborated or discussed the assignment, and any other information you wish to pass on to us.

When you are ready to submit, you will do so in two parts. First, submit your `written.pdf` file to Gradescope. Second, submit the `advising.cpp` and `README` files by running this command on the Halligan servers in your `hw09` directory:

```
▶ submit11-hw09
```

Do submit early, then keep submitting until your work is complete; we always grade the last submission received before the late token deadline.