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;
}
```


1

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

2

CS11 HW10: Rack-O

In this assignment, you will use the new OOP concepts and C++ syntax you've learned about to finish a partial implementation of a board game! Specifically, you will be fleshing out and testing some classes that together will make a 2-player game of Rack-O.

This assignment has a LOT of parts: you need to understand the rules of the game, take stock of all the files given to you, and work through the 3 major parts of this assignment, testing as you go. This spec has been designed to walk you through these parts in a step-by-step manner, but it will only serve you if you start the assignment early.

After reading our learning objectives for this assignment and the rules for the game, we suggest playing the game a few times with our demo program to make sure you understand the rules (the commands to get the code and run the demo are on the next page).

(If the official game rules and this assignment description ever disagree, follow the assignment description.)

Learning Objectives

The purpose of this assignment is to:

- Practice working with classes, methods, and member variables
- Practice implementing class methods
- Learn how to test an individual class in isolation from the complete program
- Complete a large-scale project using all the knowledge you've gained from CS11 • Implement a real-world card game with a C++ program

Game Rules

Setup:

Rack-O uses a deck of 60 cards, each with a number 1-60 (so there are no duplicate numbers). After shuffling, deal out a "rack" of 5 cards from the deck to each player. A "rack" is basically a hand of cards (and will be referred to as a Hand in the code), but in which the order of the cards matters, as you'll see shortly. The last step in the game setup is to discard the top card from the deck to form the "discard pile". The top card of the discard pile is "face up" (i.e., the players should be able to see what that card is on their turn - unlike the top card of the draw deck, which is "face down" and hidden.)

Turns:

Once each player has their rack, they will take turns performing one action on each turn until someone wins the game. A player wins by having a rack that is in ascending order from left-to-right, i.e., each card in the rack is greater than any card to the left of it - this is why the order of the cards matters. If a player wins at the end of their turn, the game ends immediately (the other player does not get a turn).

On a player's turn, they may choose to either take the top card of the discard pile (which is "face up", so 1

they know what number it is) or the top card of the draw deck (which is “face down”, so they do not know what number it is). They may then choose one card in their rack to replace with the card they just drew. (If they drew a card from the draw deck, they may also choose to discard the card they just drew after seeing its number, instead of replacing a card in their rack.)

The discarded card is then placed in the discard pile and becomes the new top (“face up” card) of that

pile. If the draw deck runs out before anyone wins, the game ends in a draw.

Let’s gooo!!

We’ve created some object-oriented starter code for you and some test files. You will not need to create any classes from scratch in this assignment, but you will implement some methods, and eventually, the core game rules. To fetch the starter code, run the command:

```
pull-code11 hw10
```

BEFORE MOVING ON, make sure you understand how the game works by playing the demo program. Feel free to play with another person and see if you can win:

```
rack-o demo deck0.txt
```

Do not move onto the next section until you feel like you could explain how the game works, based on how the demo operates. You are welcome to discuss how the game works with other students!

Part 1 (of 3) - a Deck of Cards

Understanding the Code

Let’s look at the starter code that you retrieved... There’s a lot of files. Don’t panic! :) We’ll work through it one piece at a time. For this section, we’re going to focus on the Card class, the Deck class, and the test deck program that tests the Deck class.

The Card class

Open the card.h file. This “header file” contains the definition for the Card class, which represents a card in the game. Here is a copy of the class definition, with a few parts removed (so we can focus on the important parts):

```
class Card {
  public:
    Card(int cardNumber);

    int getNum();

  private:
    int cardNum;
};
```

Starting from the bottom up, the Card class has a data member, cardNum, that represents the number value of the card (i.e., 1 - 60). It's "private" so that users of this class cannot change the card's number after the card is created.

The getNum() method is a getter method that returns the cardNum value. It's public, so users of the Card class can read the number on the card, but not modify it.

Finally, the Card(int cardNumber) constructor is called when a new instance of the class is created. The cardNumber argument is the number that the card will represent. It will be assigned to the private cardNum member variable when the card is created, but the number cannot be modified after that.

Now open the card.cpp "source" file and update the header comment at the top of the file with your name and the date. This file contains the definitions (implementations) for the Card class' methods. The approach of putting the class declaration in a header (.h) file and the method implementations in a source (.cpp) file is textbook OOP design; note that the .cpp file has to #include the .h file near the top so it can refer to the Card definition without generating compiler errors. Look at the implementations of the Card constructor and the getNum() methods and make sure you understand what they're doing; feel free to talk to other students about these if you're confused! Don't worry about the rest of the methods in the file — these are used to display the card visually on the screen and you don't need to understand what any of those methods are doing unless you feel up for a challenge.

Now open deck.h. First note that the file has the line #include "card.h" at the top; this is necessary because the Deck class will use Card objects! Now, look at the Deck class itself:

```
class Deck {
   public:
      const static int MAX_DECK_SIZE = 60;

      Deck(string deckFileName);
      ~Deck();

      bool isEmpty();
      Card *draw();

   private:
      int topCard;
      Card *deck[MAX_DECK_SIZE];
};
```

MAX DECK SIZE is a constant (captured with const and int). You can completely ignore the static part — the important thing is that you can use MAX DECK SIZE like a variable, except that you cannot change it's value; it will always be 60. It represents the most cards that can be in the deck: when you shuffle all the cards, but before you deal any out, there will be this many cards in the deck.

Now skip down to the private member variables. There's a deck array that represents the cards in the deck. Each array element is a pointer to a Card. We're using pointers so that, as we pass cards from deck to racks (hands) to discard pile, we keep only one copy of a Card object in memory for each of the numbers 1 - 60, and then we just pass around the pointer to that Card.

But an array has a fixed size, and the deck will get smaller as we deal cards off of it. That's where the topCard member variable comes in! topCard is the array index of the card on the top of the deck (and the

last array element is the card on the bottom). The topCard variable will start at 0, and then it will

be updated to the index of the new top of the deck each time a card is dealt. Thus, changing the topCard member is how we represent cards being drawn (and “removed”) from the deck; once the deck array is populated, its contents will never actually be changed during the game.

Now, let’s go over the methods in the class (feel free to open up deck.cpp so you can compare the description of these methods with the actual code implementing them). The Deck(string deckFileName) constructor takes a filename argument when a Deck object is created. This constructor will read the cards from the file, in order, create a Card object for each one, and store pointers to those objects in the deck array. It will also set topCard to 0. The deck files provided contain the numbers 1 - 60, with a different number on each line. This lets us store the deck order in a file so we can use it over and over again for testing purposes, instead of creating a random shuffle which would make it harder to create repeatable tests.

This class also has a destructor, ~Deck(), that will be called when a Deck object is destroyed (either by calling delete for a Deck object on the heap, or when a Deck object on the stack goes out of scope). The implementation of this destructor deletes any heap-allocated Card objects that are sill in the deck (but not any that were previously dealt from the deck).

Next, we have the isEmpty() method that returns true if the deck is empty (has no cards left), or false if the deck still has any cards in it.

Finally, draw() draws one card off the top of the deck, and returns a pointer to that card. The card that is returned should be effectively “removed” from the deck so that it can never be returned by any subsequent calls to draw() (how are we capturing “removing” cards from the deck?).

Now (if you haven’t already done so) open the deck.cpp file. Head down to the implementations for isEmpty() and draw(). OOPS! Looks like the professor forgot to finish implementing this class! You’re going to have to complete the implementation of these two functions yourself...but before writing any code, let’s think about what you need to do...

Written

Before doing any coding, answer the following questions in the document that you’ll eventually save as written.pdf and submit to Gradescope (at the end of this assignment). Your answers will help you figure out what to do in the programming section:

1. How can you tell whether there are any cards left in the deck or not?
2. What needs to happen to topCard each time a card is drawn from the deck?

Answers:

1. If topCard = MAX_DECK_SIZE, then you know all the cards from the deck have been drawn.
2. Each time a card is drawn from the deck, topCard increments by 1.

Programming

Don’t start implementing anything yet. While you work on the Deck class, you may want to test your Deck implementation, but the rest of the Rack-O program isn’t complete yet! Fortunately, we’ve provided a test program in test deck.cpp that will produce a program that runs a series of tests on the Deck class. Open

the file and take a look at what it is doing. This is not the program we are trying to create (i.e., it's not Rack-O), but it is a program that tests a single class of the program in isolation. This is a very useful thing to do for larger programs! We are providing a completed test program for the Deck, but you will be responsible for creating a test program yourself in the next part, so it's worth looking at what test deck.cpp is doing. You are free to ask other students about the provided code in test deck.cpp (or ask questions in OH or on piazza), but you should not be discussing with other students how you will code up the incomplete functions in deck.cpp.

4

To compile the test program, run this command:

```
g++ -g -o test_deck -Wall -Wextra test_deck.cpp deck.cpp card.cpp
```

To run it, provide the filename for a deck file on the command line, such as:

```
./test_deck deck0.txt
./test_deck deck1.txt
...etc.
```

Try running it on the starter code, before you implement anything in deck.cpp. You'll see a lot of output lines that begin with "FAILED" — these are tests of the Deck class that have failed! If you correctly implement the isEmpty() and draw() methods, the test program should no longer output those lines! If it is successful, the only output should be:

```
Failed tests are listed above. If there is no output, then it all succeeded!
```

Now complete the implementation of the isEmpty() and draw() methods in deck.cpp — you will turn in deck.cpp but should *not* modify deck.h, card.h, or card.cpp. You may (and probably need to) change the return statements that are currently there.

Do not move onto the next part of this assignment until you have passed the tests in test deck.cpp.

Part 2 (of 3) - a Hand (rack) of Cards

Understanding the Code

Now we'll take a look at the Hand class, which represents a player's rack. Open hand.h and take a look at the class. It has an array of Card pointers, representing the cards in the rack, in order. There's a constructor and a destructor; the constructor takes 5 Card pointers for the 5 cards to initially put in the rack. (This helps ensure there are always exactly 5 cards in a Hand.)

There is again a print() method for which you do not need to worry about the implementation. This is another one that's just for displaying the cards on the screen, and we've provided a complete and correct implementation for you.

isWinningHand() returns true if the hand meets the condition for winning, and false if it does not.

replaceCard(Card *newCard) replaces a card in the Hand with a new card. A pointer to the new card is given as a method argument, and it returns a pointer to the card that was replaced and removed from the hand. This method should be interactive — that is, after calling the print() function (to print out the hand) and printing a newline, it should prompt the user to choose which card to replace by entering that card's index number. (The prompt should be: ‘‘Enter the index of the card you'd like to discard: ’’; no other print

statements should be added to this method.)

Take a look at the hand.cpp file. Looks like the professor forgot to implement a couple of the methods again!

Written

For this part of the written portion, answer the following questions:

5

3. What condition do the game rules say makes a rack a “winning hand”?

The cards at hand go in ascending order from left to right.

4. What is at least one test you could run on a Hand class to verify that isWinningHand() works properly?
Do a print line statement if it works and you can change the cards on a different file that go in ascending order.

5. What is at least one test you could run on a Hand class to verify that replaceCard() works properly?

Have the program print out the card that you have replaced by storing it in another variable, and then print out the number of the index card you want in your hand.

Programming

Now complete the implementation of the isWinningHand() and replaceCard() methods in hand.cpp — you will turn in hand.cpp but should *not* modify hand.h. You may (and probably need to) change the return statements that are currently there.

Like before, there is a test program to run tests on the Hand class and make sure it works. Open the test hand.cpp file. It has been prepared with some code that reads cards from a deck file (it doesn’t create a Deck object — we want to isolate components as much as possible to test them, so we’ll only use the Hand and Card classes). Then it creates 5 Cards and makes a Hand with them. However, there aren’t actually any tests implemented! It’s up to you to implement the tests that you thought of in the written questions.

Note that the provided code in test hand.cpp is just there to demonstrate how to read cards from a file and create a Hand object. Feel free to use the provided code as-is, change it however you wish, or even scratch it and write your own thing — you have full agency to do whatever you want to in test hand.cpp to implement your tests correctly! (Also note that we demonstrated creating Cards from a file, but you may also find it useful to create Card instances with hard-coded literal numbers. Either way is fine. It’s your choice to do whatever makes your tests work best!)

To compile the test program, run this command:

```
g++ -g -o test_hand -Wall -Wextra test_hand.cpp hand.cpp card.cpp
```

To run it, provide the filename for a deck file on the command line, such as:

```
./test_hand deck0.txt
```

./test_hand deck1.txt
...etc.

Do not move onto the next part of this assignment until you have passed your tests in test card.cpp.

Part 3 (of 3) - finishing the Game

Understanding the Code

For this part, you may want to look briefly at the discard pile.h and discard pile.cpp files to get a rough understanding of the DiscardPile class. The class is fully implemented, so you don't need to do anything here, but the code you're about to write will need to use the discard(Card *discard) method, which puts the argument card on top of the discard pile, and the takeDiscard() method, which returns the card that is on top of the discard pile.

Now let's take a look at the game.h file. The Game class represents the game itself. Therefore, it has all the pieces that the game needs: a deck, a discard pile, and a rack (hand) for each player. The public play() method plays the game! It will run the entire game from setup until someone wins (or the deck runs out and a tie is declared).

The private methods are all helper methods that are called (directly or indirectly) when the play() method runs. takeTurn(int playerNum) runs a single game turn for the player whose number is given in the argument. (Player numbers start at 0, not 1; we're CS folks here...)

takeDiscard(int playerNum) runs when a player chooses to take the top discard card on their turn. drawFromDeck(int playerNum) runs when a player choses to draw a card from the deck on their turn. Both are called from the play() method.

But that lazy professor forgot to finish the game! Open game.cpp and check out the code that's there. Looks like you'll need to implement takeTurn(), takeDiscard(), and drawFromDeck() yourself. Note that you are required to call both takeDiscard() and drawFromDeck() from your takeTurn() function — you are *not* allowed to skip implementing them by putting all the code in takeTurn() directly. Also, you need to follow the style guide limitations on the length of a function.

You can also look at the game's main() in rack-o.cpp, if you want. It's pretty straightforward, since it makes the Game class do most of the work.

Written

For this part of the written portion, answer the following question:

6. What are the steps that happen on a player's turn, according to the game rules? (Note that some of them are optional or conditional on the player choices, so be sure to be clear about what happens in what conditions.) This will form your implementation plan for the last programming part of the assignment.

Answer: The hand of a player is printed and then the discard pile is printed. The player can either choose from the discard pile, which the card will be face up, or draw from the deck. If the player chooses from the discard pile, they are then asked which index and that picked up card will go into the indexed position of the user and the old card will go into the discard pile. If the user chooses to draw from the deck, they are then asked whether to keep the card or not, and if they do they will continue to be asked which index to put on the picked up card. They will then pick and that card will go into the indexed position and the old card will go into the discard pile. By the end of each turn, check if a player has won.

Programming

Point breakdown: 10 for compiling, 5 for passing valgrind, 35 for passing the provided tests, and 30 for passing our additional tests.

Finish the game by implementing the Game methods in game.cpp! Be sure to test the game thoroughly. Run the demo program and diff the output so it matches exactly. (Pay attention to where blank lines are and are not inserted into the output; those could be easy to miss.)

We have provided some test input files, but they will only play a complete game with the deck file that they were designed to work with. The prefix of the test input file will name the deck file it will work with (e.g., deck1 discard test.in only works with the deck1.txt deck file).

To compile your Rack-O program, run this huge command:

```
g++ -g -o rack-o -Wall -Wextra card.cpp hand.cpp discard_pile.cpp deck.cpp game.cpp rack-o.cpp
```

You can avoid having to type out all the .cpp files by making a sub-directory in hw10, moving any .cpp files not listed above into that sub-directory, and then compiling all the remaining .cpp files together with:

```
g++ -g -o rack-o -Wall -Wextra *.cpp
```

Once you have your executable program, you can run it by providing the filename for a deck file on the command line, such as:


```
./rack-o deck0.txt < rack-o_random_test.in
./rack-o deck1.txt < rack-o_draw_test.in
...etc.
```

Make sure you also test with valgrind to make sure you have 0 leaks and 0 errors!

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!

- No modifications can be made to *any* of the .h files. Due to this requirement, you cannot change the names, inputs, or return types of any of the methods provided to you by these class definitions.
- 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++ -Wall -Wextra card.cpp hand.cpp discard pile.cpp deck.cpp game.cpp rack-o.cpp • Your solution may not include any external packages other than those provided in the starter files.

- 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.
- Make sure you can run and diff your program using the exact commands shown above.

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.

- All methods you edit must be preceded by a function contract.
- Every method you edit should stay within the 30 line limit for functions. (Don't worry about shortening a method if you don't edit it.)
- You should use diff to test your programs, and only feel confident about passing those tests if diff does not produce any output whatsoever.
- Update the block comment of each program you edit with your name and the date you started working on it.
- For any methods you edit, follow the typical line length, indentation, naming, and commenting con ventions 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.
- Files deck.cpp, hand.cpp, and game.cpp containing your updates to fully implement the game. • 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 Grade scope. Second, submit the deck.cpp, hand.cpp, game.cpp and README files (the command will only submit those files) by running this command on the Halligan servers in your hw10 directory:

```
submit11-hw10
```

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

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.