CS 362 - Midterm

Question I

1. **Summary**
   My overall testing strategy has several components. First, I would do some manual testing to try to cover a particular equivalence partition of possible inputs. The goal in this case is to make sure that the program gets the right output for a select few important cases. This will involve positive, negative, and boundary cases enumerated below. Second, because I am concerned about how the code will handle unusual characters, I would design a random tester. This tester will be guaranteed to produce positive test cases, and so we can test whether the program is correctly handling all possible characters that might be stored in the files. Third, I would use coverage software to improve the coverage of these tests.
2. **Manual Testing**
   The positive cases for my manual testing will be those for which either something or nothing is outputted to the screen. I will be treating the "negative" cases as those for which the program should produce some kind of error. So, for example, if a string is not found in a file, I will be treating that as a positive case where nothing is outputted. But if the program is given an invalid number of arguments, that will be considered a negative case where an error should be outputted. Since the assignment description was not specific about this, I will assume the program will output to stderr some specific error.

   The positive cases include cases where the string in in.txt is only partially found in out.txt, the case where there are multiple instances of the in.txt string found in out.txt, and the case where the match is case-insensitive but not case sensitive.

   The negative cases include those where the number of arguments are incorrect, when the arguments do not specify files that exist, or when the arguments specify files that do exist but the permissions are not available for reading.

   The border cases include the case where the string in in.txt matches a string in out.txt except that in.txt has an extra newline character or carriage return character and the cases where the input or output files are empty.

   - **Positive Cases**
     - Does the program correctly output nothing when the string in in.txt matches a string in out.txt in a case insensitive way but not in a case sensitive way?
       As an example, I would check that nothing is outputted to stdout if in.txt held cs362 and out.txt held "Welcome to CS362 Software Engineering II."
     - Does the program correctly output multiple strings to stdout when the same string from in.txt is found on multiple lines in out.txt?
       I would write a program that writes a string to in.txt and then embeds that string as a substring of a longer string. I would do this multiple times and then assert that the output redirected from stdout matches the multiple strings I have stored in the testing program.
     - Does the program correctly output a line only once even if it has duplicates of the same string found in in.txt? For example, suppose that in.txt has the contents "Hello" and out.txt has only the contents "Hello Hello
       n" on one line. In this case, the line should be outputted once, not twice.

To test this, I would run the program with the above example contents and check that only one line was outputted to stdout.

- **Negative Cases**
  - Does the program output an error to stderr if the program is called with a number of arguments other than two?
    I would write a script that runs ./paraphrase with zero arguments, one argument, three arguments, and 1000 arguments, and then checks that in each case, a message was written to stderr and nothing to stdout.
  - Does the program output an error to stderr if the program is called where the either the first argument or second argument does not exist?
    I would create a script that runs ./paraphrase with an argument naming a file that does not exist, then checks that a message was written to stderr and nothing to stdout. I would do this for both argument 1 and 2.
  - Does the program output an error to stderr if the program is called where the second argument filename that exists but without the permissions for reading?
    I would write a script that creates both in.txt and out.txt but without proper read permissions, and then run the script to see if it outputs an error to stderr. Since the default behavior of such scripts is likely to send an error to stderr if a file attempts to be opened without proper permissions, whether the code correctly catches this bug depends on the details of how the script should be designed. For example, it might not want to output a specific error and just output whatever error would normally be thrown by opening a file without proper permissions.
- **Boundary Cases**
- Does the program correctly output nothing to stdout when the only difference between the contents of in.txt contain a newline character or carriage return which does not appear in out.txt?
  To test this, I would create two near-identical strings, one to be written to in.txt and one to out.txt. The one to be written to in.txt would be one that is the same as the one to be written to out.txt except that the in.txt string will be concatenated with one newline character. I would repeat this process for the carriage return. This case is a kind of boundary case to ensure that it treats newline characters as themselves characters that must exist in out.txt for the line to be outputted to stdout.
- Does the program correctly output 100,000 line to stdout when the string in in.txt is found on those same 100000 lines in out.txt? (or some other arbitrarily large value)
  The purpose of this boundary case is just to see how it handles cases where the number of lines is very large.
- Does the program correctly output 1 line to stdout when the string is found on the last line of the code?
  To test this, I would write a program that fills some arbitrary number of lines with garbage, and then fills the last line with a string that contains the substring written to in.txt.
- Does the program correctly output 1 line to stdout when the string is found on the first line of the code?
  To test this, I would write a program that fills some arbitrary number of lines with garbage, and then fills the first line with a string that contains the substring written to in.txt.

3. **Random Testing**

   There are many interesting cases that might be missed in manual testing. Two obvious ones to test would be (1) whether the program can handle cases where the character data is unusual and (2) whether it consistently gets the right number of lines outputted to stdout.

For the first case, I would design my random tester to produce a random string to be written to in.txt. I would then write a function called embedString that would embed that string as a substring of different randomly generated string and append that new string to out.txt along with a newline character. I would store this randomly generated string and check that the redirected output from stdout matches character for character.

For the second case, I would modify the above approach slightly. Because we are interested in counting the number of lines that get outputted, I would create a second string to be appended to out.txt. I would check that this second string does not contain the substring written to in.txt. Therefore, there would be two possible string that could be appended to out.txt. I would randomly generate an integer representing the number of outputs we would expect, and I would write that many lines of the randomly generated string to out.txt. I would then generate another random integer and write the string that doesn't contain the in.txt string. Since I would know how many lines contain the substring, I would check that it outputs the same number of lines to stdout. I would iterate this millions of times to check that it is working for arbitrary sizes of lines.

4. **Coverage**

   Assuming this is white-box testing, I would use some kind of coverage software in both manual and random testing to ensure close to 100% branch and statement coverage. I would then use the results from this percent coverage to modify my tests so that (1) in the manual cases, I have 100% coverage and (2) in the random case, I increase the percentage of those branches that are rarely taken. Based on my interpretation of how the source code would be written, I would guess that there aren't any issues involving path coverage.

Question II I chose three test cases that cover multiple cases. Case 1 is the case where n does not exist in container C. This case is really testing that the function does not delete anything in the container after calling removeAll with a value that doesn't exist in C. The only way to test this with just the provided signatures is to assert that size hasn't changed. Case 2 is that n exists some known k number of times in C. I will run this case with one negative value, one positive value, and zero to ensure the same behavior for negatives, positives, and zero. For all three values, I use both size and get to test correctness. The final case is when C is completely empty. This boundary case is testing whether the function does nothing and throws no error.

(a) Test Case 1: n does not exist in C

The below code will create a container that does not contain c. It will then copy that container, call removeAll, and then check that the size has not changed.

```
struct container C, testC;
&C = newContainer();
&testC = newContainer();

// populate C with values less than 100
for (int i = 0; i < 100; i++)
{
  add(i, &C);
}
```

```
// copy memory to testC
memcpy(&C, &testC, sizeof(struct container C));

// run function on testC
removeAll(101, &testC);

// check that size hasn't changed, i.e. nothing removed
assert(size(&C) == size(&testC));
```

(b) Test Case 2: n exists in C k times

The below code will create a container that contains n. It will then copy that container to another, call removeAll, and then check that the size has decreased by the number of times, k, that n was added to c. I will also called get(&C, n) and assert that it returns false. I will do this three times, one for positive k, one for negative k, and one for 0.

```
struct container C, blankC;
&C = newContainer();
&blankC = newContainer();

// populate C with values less than |100|
for (int i = 0; i < 100; i++)
{
  add(i, &C);
  add(-i, &C);
}

memcpy(&blankC, &C, sizeof(struct container));

for (int i = 0; i < 100; i++)
{
  add(300, &C);        // will check that number of instances
      of 300 == 100
}

removeAll(300, &C);
assert(size(&C) == 100);
assert(get(&C, 300) == 0);

memcpy(&C, &blankC, sizeof(struct container));

for (int i = 0; i < 100; i++)
{
  add(-300, &C);        // will check that number of instances
      of -300 == 100
}

removeAll(-300, &C);
```

```
assert(size(&C) == 100);
assert(get(&C, 300) == 0);

memcpy(&C, &blankC, sizeof(struct container));

for (int i = 0; i < 100; i++)
{
  add(0, &C);        // will check that number of instances of
     -300 == 100
}

removeAll(0, &C);
assert(size(&C) == 100);
assert(get(&C, 0) == 0);
```

(c) Test Case 3: C is completely empty

The below code will create a container that has no values. Then it will call removeAll and check that size is 0 afterwards.

```
struct container C, testC;
&C = newContainer();

// run function on testC
removeAll(1, &C);

// check that size hasn't changed, i.e. nothing removed
assert(size(&testC) == 0);
```