Programming Assignment #4: Reflections and Kindred Spirits

COP 3502, Spring 2020

Due: Sunday, March 29, *before* 11:59 PM

Abstract

This program is designed to get you thinking recursively with binary trees. Rather than solving one big problem, you will code up solutions to three smaller problems, each of which will rely on recursion to some degree.

This time around, I'm not giving you a slew of function definitions that essentially pre-define the structure of your program. Accordingly, this assignment will challenge you to think creatively, both in terms of how to solve the problems, as well as how to make appropriate use of helper functions as you plan out how to structure your code – especially with the *kindredSpirits()* function. This will be fun.

Deliverables

KindredSpirits.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Overview, Part 1 of 2: Reflections

Given two binary trees, we say that one is a reflection of the other if they are symmetric in terms of both their structure and their node values. For example, the following trees are reflections of one another:

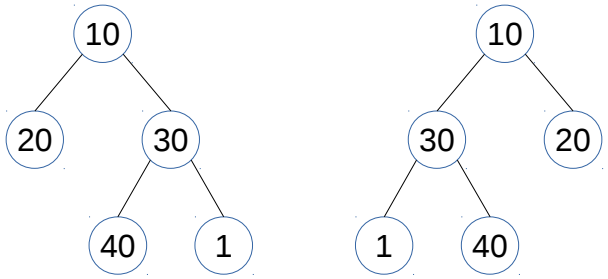

Figure 1: *Two trees that are reflections of one another.*

The following trees are *not* reflections of one another, because although they are symmetric images of one another *structurally*, they are not symmetric in terms of their *values*:

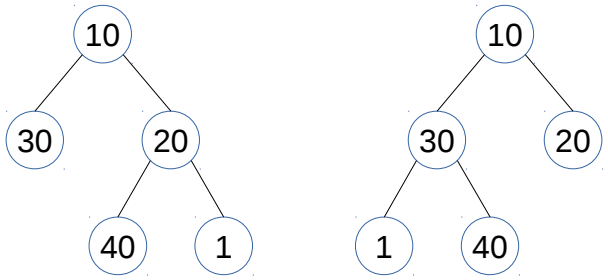

Figure 2: *Two structurally symmetric trees that are not reflections of one another because they are lack symmetry with respect to their node values.*

Because the following binary trees are not structurally symmetric (and therefore they also cannot be symmetric in terms of the values contained in each node), they are not reflections of one another:

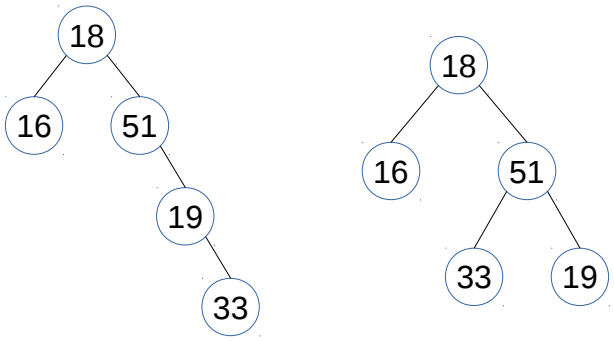

Figure 3: *Structurally asymmetric trees cannot be reflections of one another.*

The following binary trees are reflections of one another:

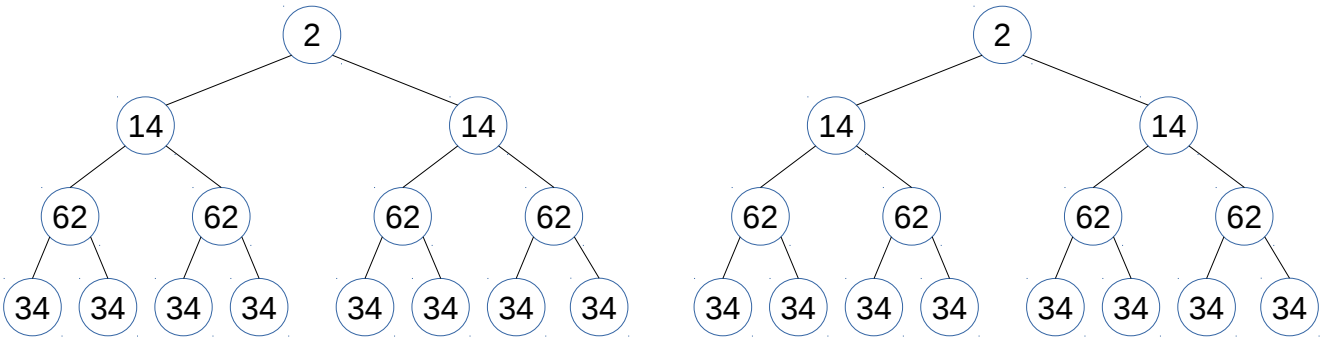

Figure. 4: *Two trees that are reflections of one another. They are perfect binary trees. They are also large and in charge.*

Recall that the tree above is a perfect binary tree. You might ask yourself, “Is it always the case that a perfect binary tree is a reflection of itself?” The answer is, “No!” Here’s a counterexample that shows that a perfect binary tree is not always a reflection of itself:

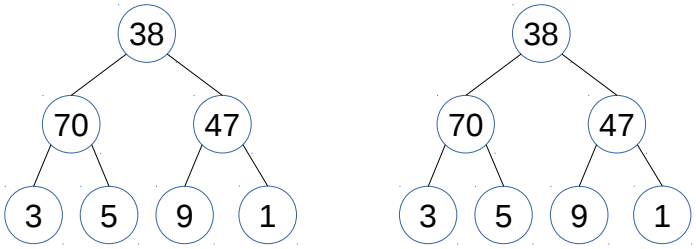

Figure 5: *This perfect binary tree is not a reflection of itself. Despite its structural symmetry, it is not symmetric to itself with respect to its node values.*

Any binary tree with a single node is a reflection of itself. For example:

Figure 6: *Two trees that are reflections of one another.*

However, it is *not* the case that all binary trees with a single node are reflections of one another. For example:

Figure 7: *Two trees that are not reflections of one another, because they are not symmetric with respect to their values.*

Finally, note that the empty tree is a reflection of itself:

Figure 8: *Two trees (both empty) that are reflections of one another. This example is serene and beautiful, and brings with it an overwhelming sense that everything is going to be okay.*

2. Overview, Part 2 of 2: Kindred Spirits

We say that two binary trees are *kindred spirits* if the preorder traversal of one of the trees corresponds to the postorder traversal of the other tree. For example, the following trees are kindred spirits:

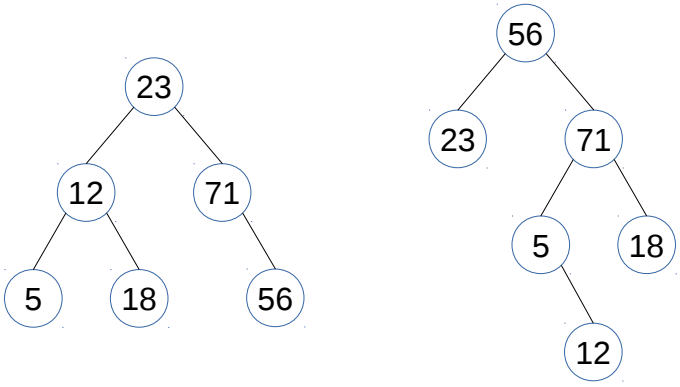

Figure 9: *These trees are kindred spirits. The preorder traversal of the tree on the left is 23, 12, 5, 18, 71, 56, which corresponds to the postorder traversal of the tree on the right.*

Note that the trees above are still kindred spirits even if we swap their order:

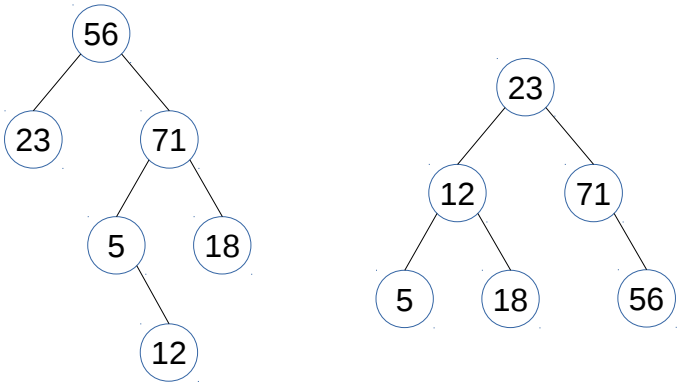

Figure 10: *These trees from Figure 9 are still kindred spirits, despite the fact that the preorder traversal of the tree on the right now matches the postorder traversal of the tree on the left, instead of the other way around.*

3. Binary Tree Node Struct (KindredSpirits.h)

You must use the node struct we have specified in *KindredSpirits.h* without any modifications. You **must** *#include* the header file in your *KindredSpirits.c* source file like so:

```
#include "KindredSpirits.h"
```

Note that the capitalization of *KindredSpirits.c* matters! Filenames are case sensitive in Linux, and that is of course the operating system we'll be using to test your code.

The node struct is defined in *KindredSpirits.h* as follows:

```
typedef struct node
{
   int data;                    // each node holds a single integer
   struct node *left, *right;  // pointers to node's left and right children
} node;
```

4. Output

The functions you write for this assignment should not produce any output. If your functions cause anything to print to the screen, it might interfere with our test case evaluation. Be sure to disable or remove any *printf()* statements you have in your code before submitting this assignment.

5. Function Requirements

You have a lot of leeway with how to approach this assignment. There are only five required functions, and you may write helper functions as you see fit. For the *kindredSpirits()* function, you will likely need quite a few helper functions, and a good measure of creative thinking and/or cleverness.

Function descriptions for this assignment are included on the following page. Please do not include a *main()* function in your submission.

```
int isReflection(node *a, node *b);
```

Description: A function to determine whether the trees rooted at *a* and *b* are reflections of one another, according to the definition of "reflection" given above. This must be implemented recursively.

Returns: 1 if the trees are reflections of one another, 0 otherwise.

```
node *makeReflection(node *root);
```

Description: A function that creates a new tree, which is a reflection of the tree rooted at *root*. This function must create an entirely new tree in memory. As your function creates a new tree, it must *not* destroy or alter the structure or values in the tree that was passed to it as a parameter. Tampering with the tree rooted at *root* will cause test case failure.

Returns: A pointer to the root of the new tree. (This implies, of course, that all the nodes in the new tree must be dynamically allocated.)

```
int kindredSpirits(node *a, node *b);
```

Description: A function that determines whether the trees rooted at *a* and *b* are kindred spirits. (See definition of "kindred spirits" above.) The function must not destroy or alter the trees in any way. Tampering with these trees will cause test case failure.

Special Restrictions: To be eligible for credit, the worst-case runtime of this function cannot exceed O(*n*), where *n* is the number of nodes in the larger of the two trees being compared. This function must also be able to handle arbitrarily large trees. (So, do not write a function that has a limit as to how many nodes it can handle.) You may write helper functions as needed.

Returns: 1 if the trees are kindred spirits, 0 otherwise.

```
double difficultyRating(void);
```

Returns: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Returns: A reasonable and realistic estimate (greater than zero) of the number of hours you spent on this assignment.

6. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *KindredSpirits.c*, *KindredSpirits.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm is fairly straightforward, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*KindredSpirits.c*, *KindredSpirits.h*, *test-all.sh*, the test case files, and the *sample_output* folder).
2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

```
scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
```

> **Warning:** Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

```
bash test-all.sh
```

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd KindredSpiritsProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type (`cd project\ 4`), or the entire name needs to be wrapped in double quotes.

7. Running the Provided Test Cases Individually

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. Here's how to do that:

1. Place all the test case files released with this assignment in one folder, along with your *KindredSpirits.c* file.
2. At the command line, *cd* to the directory with all your files for this assignment, and compile your source file with one of our test cases (such as *testcase01.c*) like so:

   ```
   gcc KindredSpirits.c testcase01.c
   ```
3. To run your program and redirect the output to *output.txt*, execute the following command:

   ```
   ./a.out > output.txt
   ```
4. Use *diff* to compare your output to the expected (correct) output for the program:

   ```
   diff output.txt sample_output/testcase01-output.txt
   ```

   If the contents of *output.txt* and *testcase01-output.txt* are exactly the same, *diff* won't have any output:

   ```
   seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
   seansz@eustis:~$ _
   ```

   If the files differ, *diff* will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

Super Important: *Remember, using the test-all.sh script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.*

8. Style Restrictions (*Super Important!*)

These are the same as in the previous assignment. Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your *#include* statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

- ★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(void)* instead of *int main (void)*. Similarly, use *printf("...")* instead of *printf ("...")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

9. Special Restrictions (*Super Important!*)

1. As always, you must avoid the use of global variables, mid-function variable declarations, and system calls (such as `system("pause")`).
2. Do not read from or write to any files. File I/O is forbidden in this assignment.
3. Be sure you don't write anything in *KindredSpirits.c* that conflicts with what's given in *KindredSpirits.h*. Namely, do not try to define a node struct in *KindredSpirits.c*, since your source file will already be importing the definition of a node struct from *KindredSpirits.h*.
4. No shenanigans. For example, if you write a *kindredSpirits()* function that always returns 1, you might not receive any credit for the test cases that it happens to pass.
5. Your *KindredSpirits.c* file **must not** include a *main()* function. If it does, your code will fail to compile during testing, and you will not receive credit for this assignment.
6. Be sure to include your name and NID as a comment at the top of your source file.

10. Deliverable (Submitted via Webcourses, not Eustis)

Submit a single source file, named *KindredSpirits.c*, via Webcourses. The source file must contain definitions for all the required functions listed above. Be sure to include your name and NID as a comment at the top of your source file. Don't forget `#include "KindredSpirits.h"` in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also compile on Eustis with both of the following:

```
gcc -c KindredSpirits.c
gcc KindredSpirits.c testcase01.c
```

Continued on the following page...

11. Grading

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

80%	Correct output for test cases used in grading (and possibly memory leak checks)
10%	Appropriate use of functional decomposition (see below)
10%	Follows all style and special restrictions and includes adequate comments and whitespace

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* results expected. Even minor deviations will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Note also that your functions should not print anything to the screen. If they do, it will interfere with the output we generate while testing, resulting in incorrect test case results and an unfortunate loss of points.

Additional points will be awarded for style (commenting and whitespace practices) and functional decomposition (i.e., don't write a 300-line *kindredSpirits()* function; break it up into meaningful functions!). The graders will inspect your *isReflection()* and *kindredSpirits()* functions to ensure they aren't just bogus functions that always returns 1. Shenanigans like that might result in severe point deductions. Also, don't forget that the runtime of *kindredSpirits()* should not exceed O(*n*), where *n* is the number of nodes in the larger of the two trees being compared.