Programming Assignment #3: Reflections and Kindred Spirits

Fall 2017

Due: Sunday, December 10, *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.

Attachments

KindredSpirits.h, testcase{01-16}.c, output{01-16}.txt

Deliverables

KindredSpirits.c

(Note: Capitalization of your filename matters!)
(This project has been adapted from Sean Szumlanski's projects)

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:

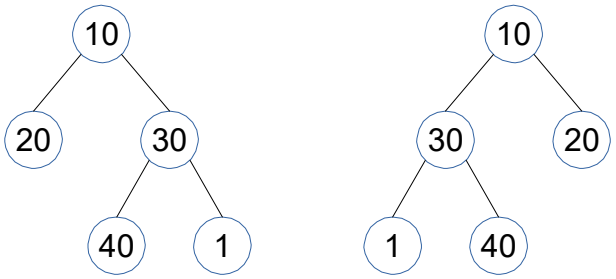

Figure 1: *Two trees that are reflections of one another.*

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*:

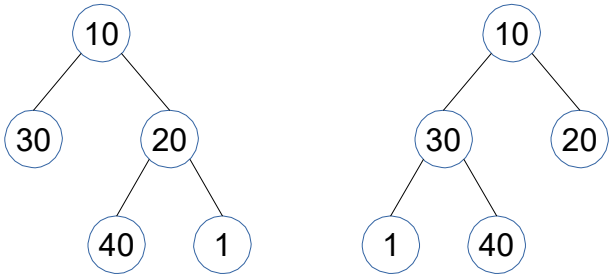

Figure 2: *Two structurally symmetric trees that are not reflections of one another because they are lack symmetry with respect to their node values.*

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:

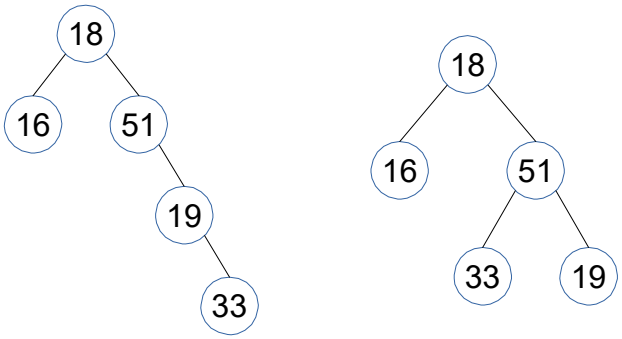

Figure 3: *Structurally asymmetric trees cannot be reflections of one another.*

The following binary trees are reflections of one another:

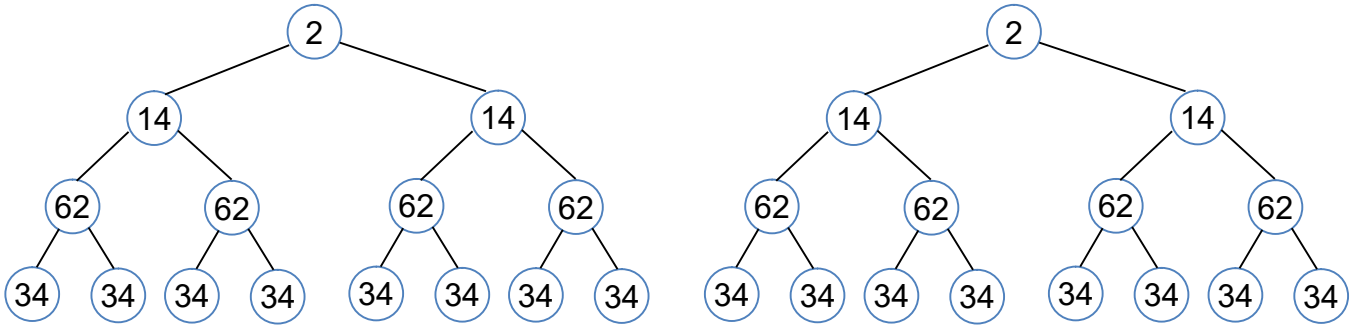

Figure. 4: *Two trees that are reflections of one another. They are perfect binary trees. They are also large and in charge.*

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:

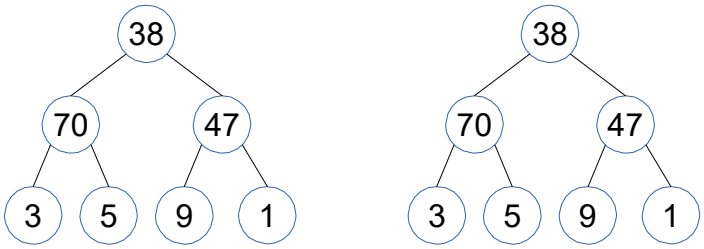

Figure 5: *This perfect binary tree is not a reflection of itself. Despite its structural symmetry, it is not symmetric to itself with respect to its node values.*

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 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:

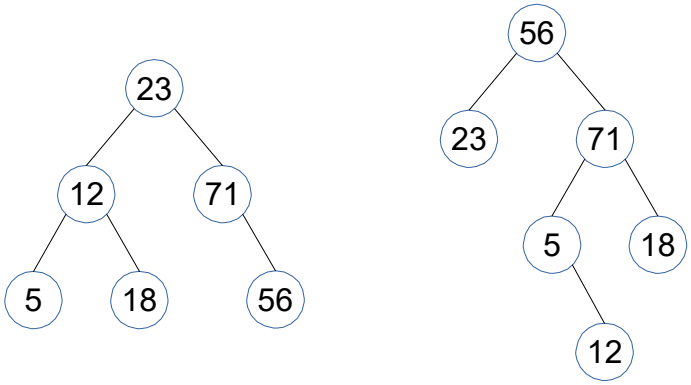

Figure 9: *These trees are kindred spirits. The preorder traversal of the tree on the left is 23, 12, 5, 18, 71, 56, which corresponds to the postorder traversal of the tree on the right.*

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

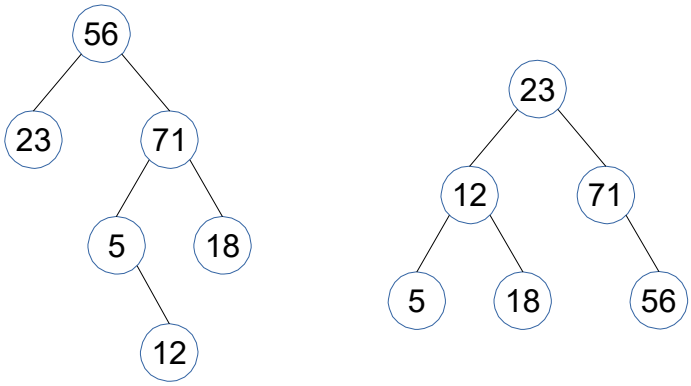

Figure 10: *These trees from Figure 9 are still kindred spirits, despite the fact that the preorder traversal of the tree on the right now matches the postorder traversal of the tree on the left, instead of the other way around.*

3. Binary Tree Node Struct (KindredSpirits.h)

You must use the node struct we have specified in `KindredSpirits.h` without any modifications. You will have to `#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;
   struct node *left, *right;
} node;
```

If you're using Code::Blocks, you will need to add the `KindredSpirits.h` header file to your project. See the instructions in Section 8, "Compilation and Testing (Code::Blocks)."

4. Test Cases

As always, I've included several test cases, which show some ways in which I might test your code. These test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively.

5. 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!!!*

6. Function Requirements

You have a lot of freedom 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: An estimate (greater than zero) of the number of hours you spent on this assignment.

The remaining pages of this document contain compilation instructions for Linux/Mac (pg. 7) and Windows (pg. 8), further restrictions and guidelines for your program (see pgs. 8 and 9), and grading criteria (pg. 9).

7. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

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

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc KindredSpirits.c testcase01.c -o KindredSpirits.exe
```

...and then run the program using:

```
./KindredSpirits.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./KindredSpirits.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
navid@ubuntu:~$ diff whatever.txt output01.txt
navid@ubuntu:~$ _
```

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

```
navid@ubuntu:~$ diff whatever.txt output01.txt
1c1
< fail whale :(
---
> Success!
navid@ubuntu:~$ _
```

8. Compilation and Testing (Code::Blocks)

The key to getting your program to include a custom header file in Code::Blocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in Code::Blocks, importing KindredSpirits.h and the KindredSpirits.c file you've created (even if it's just an empty file so far).

1. Start Code::Blocks.
2. Create a New Project (*File → New → Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "KindredSpirits").
6. Pause to reflect on life a bit. Isn't this amazing?
7. Choose a folder (e.g., Desktop) where Code::Blocks can create a subdirectory for the project.
8. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into Code::Blocks. Then right click the tab for **each** file and choose "Add file to active project."

– or –

2. Go to *Project → Add Files*…. Then browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

9. Deliverables

Submit a single source file, named `KindredSpirits.c`, via Canvas. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "KindredSpirits.h"` in your source code. Your program must compile without any special flags, as in :

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

10. Special Restrictions

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 submit the `KindredSpirits.h` header file with your code. You should only submit `KindredSpirits.c`. We will use our own version of the header file when testing your program.

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. Be sure to include your name and NID as a comment at the top of your source file.

5. 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.

6. 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.

11. Grading

The *expected* scoring breakdown for this programming assignment is:

80%	Correct output for test cases used in grading
10%	Appropriate use of functional decomposition
10%	Comments and whitespace

Note! Your program must be submitted via Canvas, and it must compile 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.

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.