Programming Assignment #4: Reflections

COP 3502, Fall 2015

Due: Wednesday, October 28, *before* 11:59 PM

Abstract

This is a very short program designed to get you thinking recursively with binary trees. You will code up two small recursive functions. This assignment is designed to help prepare you for Exam #2, as well as to give you a slightly less complex and lengthy assignment than the ones we've seen so far this semester, to help buffer your assignment average a bit. Also, it will be fun.

Attachments

Reflections.h, sample-main.c

Deliverables

Reflections.c

(Note: Capitalization of your filename matters!)

1. Overview

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:

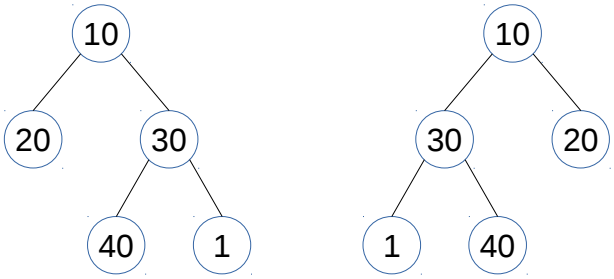

Figure 1: *Two trees that are reflections of one another.*

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*:

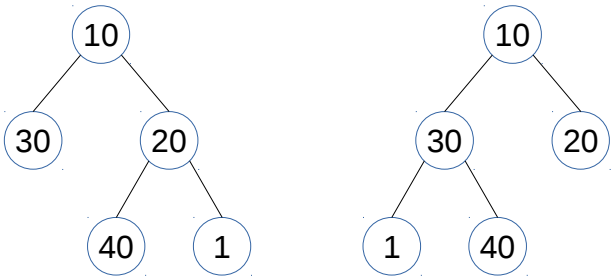

Figure 2: *Two structurally symmetric trees that are not reflections of one another because they are lack symmetry with respect to their node values.*

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:

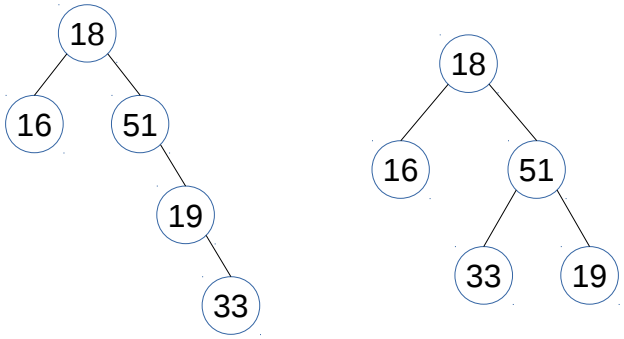

Figure 3: *Structurally asymmetric trees cannot be reflections of one another.*

The following binary trees are reflections of one another:

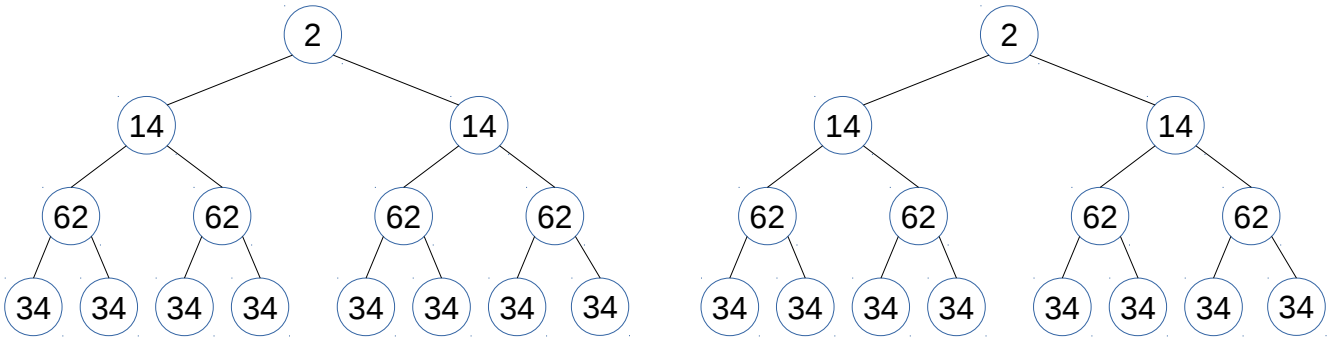

Figure. 4: *Two trees that are reflections of one another. They are perfect binary trees. They are also large and in charge.*

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:

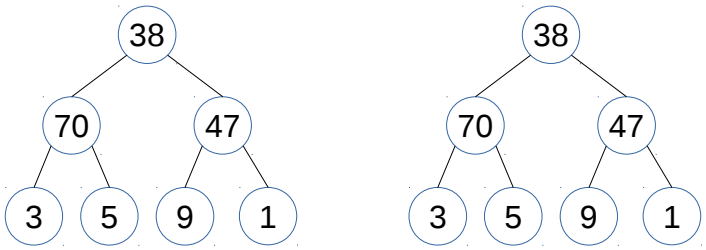

Figure 5: *This perfect binary tree is not a reflection of itself. Despite its structural symmetry, it is not symmetric to itself with respect to its node values.*

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. Binary Tree Node Struct (Reflections.h)

You must use the node struct we have specified in `Reflections.h` without any modifications. You should `#include` the header file from `Reflections.c` like so:

```
#include "Reflections.h"
```

Note that the capitalization of `Reflections.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 `Reflections.h` as follows:

```
typedef struct node
{
   int data;
   struct node *left, *right;
} node;
```

If you're using Code::Blocks, you will need to add Reflections.h header file to your project. See the instructions in Section 7, "Compilation and Testing (Code::Blocks)."

3. Sample main() Files

You do not need to include a `main()` function in your submission. (There's no penalty if you include a `main()` function, though. I'll just ignore it during testing.)

I've included a sample `main()` function in `sample-main.c`, which shows some ways in which I might unit test your code. You can plunk it into your `Reflections.c` file and compile and run to test your code, but you should also create your own test cases if you want to test your code comprehensively.

4. Output

Your program should not produce any output. If your functions cause anything to print with 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 four required functions, and you may write helper functions as you see fit.

You don't have to include a `main()` function in your submission. It's fine if you do, but I'll be replacing it with my own `main()` function when I test your code.

```
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.)

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

> **Returns:** Return 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:** Return 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. 6) and Windows (pg. 7), further restrictions and guidelines for your program (see pgs. 7 And 8), and grading criteria (pg. 8).

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

To compile at the command line, ensure that `Reflections.h` is in the local directory, then type:

```
gcc -c Reflections.c
```

If you have a `main()` function in your source file, you can compile and run the program without the -c flag:

```
gcc Reflections.c
```

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

```
./a.out
```

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

```
gcc Reflections.c -o Reflections.exe
```

...and then run the program using:

```
./Reflections.exe
```

Your `isReflection()` and `makeReflection()` functions should not produce any output, but if your `main()` function is set up to produce output for testing purposes, running the program could potentially dump a lot of information 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:

```
./Reflections.txt > whatever.txt
```

This will create a file called `whatever.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files. You can see whether the contents of two files match exactly by typing, e.g.:

```
diff whatever1.txt whatever2.txt
```

If the contents of `whatever1.txt` and `whatever2.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever1.txt whatever2.txt
seansz@eustis:~$ _
```

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

```
seansz@eustis:~$ diff whatever1.txt whatever2.txt
3c3
< Success!
---
> Fail whale!
seansz@eustis:~$ _
```

7. 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 Reflections.h and the Reflections.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., "Reflections").
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."

You should now be good to go. Try to build and run the project (F9). As a friendly reminder: Even if you develop your code with Code::Blocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 6, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

8. Deliverables

Submit a single source file, named `Reflections.c`, via Webcourses. 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 "Reflections.h"` in your source code. Your program should compile on Eustis using the following (since you won't necessarily have defined a `main()` function):

```
gcc -c Reflections.c
```

9. Special Restrictions

1. Do not use global variables (although `#define` directives are totally fine).

2. Be sure to declare all your variables at the tops of your functions. No mid-function variable declarations.

3. Do not submit the `Reflections.h` header file with your code. Just submit `Reflections.c`. We will use our own version of the header file when testing your program.

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 an `isReflection()` function that always returns 1, you might not receive any credit for the test cases that it happens to pass.

10. Grading

The expected scoring breakdown for this programming assignment is:

90%	Unit Testing (see details below)
10%	Comments and Whitespace

Your program 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.

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

For this program, we will be unit testing your code. That means we will devise tests that call your functions directly. We will leave any helper functions that you've written intact, so if your functions rely on helper functions, they will still work. However, we will not call your `main()` function (if you choose to include one).

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `isReflection()` function to ensure it isn't just a bogus function that always returns 1. Shenanigans like that might result in severe point deductions.