CS-102 Design Principles: Project 7

Due date: Monday, April 10 at 11:00pm

In this project, you will implement two different versions of `remove` for binary search trees, and write a `height` method that you can use to compare the efficiency of the two versions. There will not be an autograder for this project, because your final project is open-ended (and thus will not have one), and because it's likely that none of your future courses in computer science will use autograders. You need to practice writing your own tests.

You may use any code that we have written together in class (without citation), but not any code taken from anywhere else.

The test files for this project will be of this form:

```
400 300 600 500 550 350 450 625 650 325
500
```

All entries in the first row will be unique positive integers; there will be at least one integer in the first row. You can assume that the number in the second row also appears somewhere in the first row.

Your program should build a binary search tree from the keys that appear in the first row, inserting them from left to right. Then your program should remove the node that has the key that appears in the second row, and it should remove that node in two ways: *normal remove* (as described and practiced during day 27) and *JR-remove* (as discussed during day 26 and then written in pseudocode in codelet 27). Your program should output each binary tree, after each remove, using a reasonable method of your choice that would allow us to reconstruct your tree ourselves. Lastly, your program should output the height of each binary tree that results from each of the two kinds of remove method.

My guess is that JR-remove will result (sometimes? always?) in trees of greater height than before the remove, whereas normal remove will result (sometimes? always?) in trees of lesser height. You are not required to reflect on these matters, but I thought I would explain my motivation in assigning this project. We are testing a hypothesis! Of course, we could also "test" the hypothesis on paper, using logic. You will likely do some of that sort of testing in CS-230.

Getting started

There is no Github assignment to accept, nor is there any starter code; however, you are welcome to start with any code that we have written in class or in lab. When you are ready, click on the assignment on Gradescope for this project, and submit your files.

Requirements

You should separate your files as you learned in lab:

- ☐ Each class has its own `.h` and `.cpp` files; your class declaration should be in the `.h` file, and your method definition(s) should be in the `.cpp` file.
- ☐ Put header guards in the `.h` file, using the convention `FILENAME_H`.
- ☐ Compile all of the `.cpp` files to `.o` object files for each class.
- ☐ Link in the object files when compiling your code.

You should also take care to write code that has a clean `valgrind` report.

Grading

Your grade will come from:

75%	correctness + passing our tests
25%	style

Projects passing zero tests will receive at most 70% of the project grade (assuming perfect style). Passing tests increases the grade proportionally up from 70%.

The elements of style that we will assess on this project are:

- ☐ You used `.h` files correctly.
- ☐ Your helper functions should be no more than about 20 lines long (not including spacing for curly braces, and not counting comments); each function should have one purpose, and be named according to that purpose.
- ☐ Use appropriate programming constructs; e.g., don't use a loop when an if statement is sufficient, a while loop when a for loop is better, and so on.
- ☐ Prune your dead code: remove commented out code, unused variables, other code not contributing to the overall project.

  We will give you one aggregate grade for the rest of these style elements:

- ☐ Include all file and function header comments, using the required format.
- ☐ Declare variables near their initial use.
- ☐ Indent properly and use correct placement of curly braces and parentheses.
- ☐ Use correct spacing around operators.
- ☐ Use reasonable variable names in proper naming style: using `lower_case` naming in local scope, and `UPPER_CASE` naming for global constants.