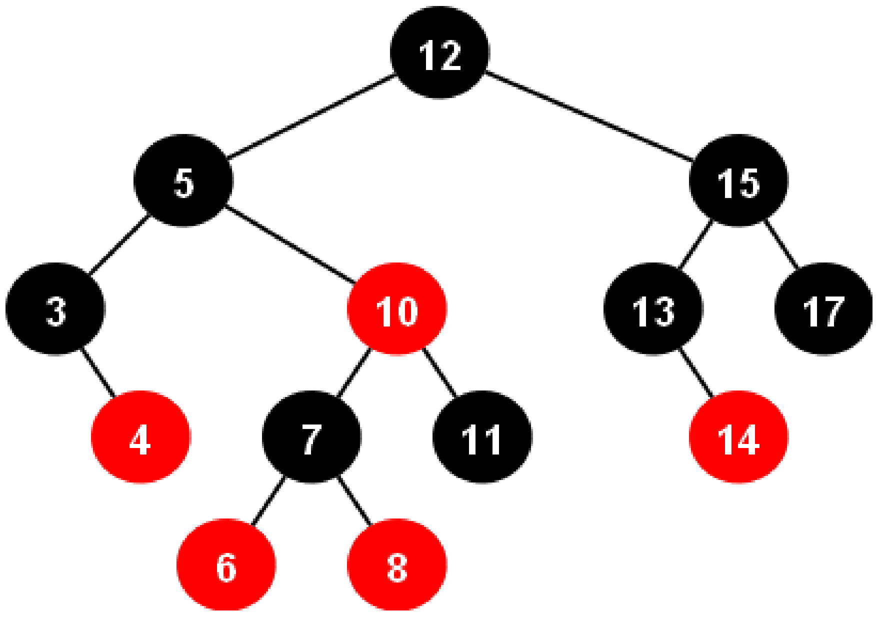
CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

Homework 7

- Due: Wednesday, November 18th, 11:59pm (PST)
- Directory name for this homework (case sensitive): `hw7`
  - This directory should be in your `hw_username` repository
  - This directory needs its own `README.md` file
  - You should provide a `Makefile` to compile your code for each problem.

Problem 1 (Red-Black Trees, 10%)

Consider the following initial configuration of a Red-Black Tree:



Draw the tree representation of the Red-Black tree after each of the following operations. Your operations are done in **sequence**, so your tree should have 17 values in it when you're done. Make sure to clearly indicate each of your final answers.

- Insert 1
- Insert 13.5
- Insert 14.5
- Insert 9

We highly recommend you try to solve these by hand before using any tools to verify your answers.

Problem 2 (Binary Search Tree Iterators, 20%)

We are providing for you a file `bst.h` (in the homework-resources repository) which implements a simple binary search tree. You will need to implement the iterator, so that it traverses the tree using an in-order traversal.

You may add any helper functions you like. A successor helper function (which returns the next largest value after the current node) may be particularly useful.

Problem 3 (Red-Black Tree Insertion, 30%)

We are providing you a half-finished file `rbbst.h` (in the homework-resources repository) for implementing a Red-Black Tree. It builds on the file you completed for the previous question.

Complete this file by implementing the `add()` function for Red-Black Trees. You are strongly encouraged to use private/protected helper functions.

Note (that may not make sense until you have started coding): Your RBTree will inherit from your BST. This means the root member of BST is a Node type pointer that points to a RBNode (since you will need to create RBNodes with `color` values for your RBBST), which is fine because a base Node pointer can point at derived RBNodes. If you need to call RBNode-specific functions (i.e. members that are part of RBNode but not Node) then one simple way is to downcast your Node pointer to an RBNode pointer, as in `static_cast<RedBlackNode<K,V>*>(root)` to temporarily access the RBNode-specific behavior/data.

Problem 4 (Recursion and Backtracking - Graph Coloring, 40%)

Write a program to 4-color a map using recursion and backtracking. The input file (whose name will be passed at the command line) will have three parameters on the first line: how many "countries" there are in the map, the number of rows in the map, and the number of columns in the map. The map will never be bigger than 80 by 80 characters, and will contain at most 15 countries. The countries are denoted by characters 'A', 'B', ... up to the letter for the highest country number. Here is an example of what the input might look like:

```
5 6 13
AAAAAACCCCCCC
AAAAAACCCCCCD
BBBAAACCCCCDD
BBAAAACCEEDDD
BBBBBACDEEEDD
BBBBBBDDDDDDD
```

Each country will always be contiguous; in other words, you can always walk from any point in a country to any other just going horizontally or vertically. We will say that two countries share a border if at least one square of one country is horizontally or vertically adjacent to at least one square of the other. In the example above, A shares a border with B and C, and D shares a border with B and C and E, and C and E also share a border. A and D do not share a border, and neither do B and C, because they only touch diagonally.

A map is properly colored if each country is assigned a color, and no two countries which share a border have the same color. (Otherwise, you couldn't tell apart where one starts and the other ends.) Of course, one can always color a map by giving each country its own color, but we want to reuse colors as much as possible, in the sense that the total number of distinct colors is minimized. For instance, in the example above, we could make A and D red, B and C blue, and E green, minimizing the total number of colors to 3.

The famous Four-Color Theorem guarantees that there is always a way to color any map with just 4 colors. You are to write a program that actually **finds** such a coloring. Unless you were going to read thousands of pages of math proofs to understand how to do this differently, we strongly recommend using recursion and backtracking to solve this problem. While backtracking is slow, our guarantee that you have at most 15 countries (and only 4 colors) means that you only need to check at most about a billion cases, and backtracking will truncate this significantly. So it should run fast enough if you are careful.

The output should be a valid coloring that outputs the color assigned to each country, one per line. For the example coloring we gave for the input above, you would output (numbering the colors 1, 2, 3, 4):

```
A 1
B 2
C 2
D 1
E 3
```

Of course, there are many other colorings. You are welcome to output any one you like, or multiple solutions if you prefer. The following is not necessary (you get full credit without), but may amuse you: try to use as few colors as possible. That is, use only 3 colors if the map can be colored with just 3, and in the rare cases where 2 colors are enough, use just two.

To exercise your RB-Tree implementation above, you **MUST** maintain a map (not an array or list) of each country to its current color value, and that map **must be your RB-Tree implementation**. Failure to use your RB-Tree implementation for this purpose will result in a deduction of 10% of the possible points for this problem.

As always, no memory leaks should be present in your program.

In case you want more non-trivial test cases, here are a few. (If your solution does not use Backtracking, there is a decent chance it will fail on one or both of these inputs.)

Example 1

```
6 4 8
FFFFFFFF
FEEEDDDF
FAABBCCF
FFFFFFFF
```

Here is a valid output for this input:

```
A 1
B 2
C 3
D 1
E 3
F 4
```

Example 2 (Larger)

```
6 10 15
FFFFFFFFFFFFFFF
FFFFFFFFFFFFFFF
FFEEEDDDDDDDDFF
FFEEEDDDDDDDDFF
FFEEEDDDDDDDDFF
FFAABBBBBBCCCFF
FFAABBBBBBCCCFF
FFAABBBBBBCCCFF
FFFFFFFFFFFFFFF
FFFFFFFFFFFFFFF
```

Valid output:

```
A 1
B 2
C 3
D 1
E 3
F 4
```

Feel free to generate your own map files and post them on Piazza along with the possible valid color assignments.

Chocolate Problem (Fast Map Coloring)

Value: 2 Chocolate bars. (As with all chocolate problems, this problem is entirely optional. Solving will does not give you extra points for the class, just actual chocolate. If you solve it, please e-mail the solution directly to your instructor, and don't submit it via github. If you work as a group, you share the chocolate you earn.)

Write a **fast** map coloring algorithm that never uses more than 6 colors. Your algorithm should be able to handle thousands of countries in a large map. Test it on such inputs. In addition to code, please submit a mathematical proof that your algorithm never uses more than 6 colors.

Hints: 1. The adjacency relationship between countries gives you a planar graph. (Look up the definition.) What can you prove about the smallest degree of a node in a planar graph? 2. What would be a good time to color a node with small degree, and why?