Introduction to Red-Black Trees

Red-Black trees are a very common implementation of a **self-balancing binary search tree**. Essentially, insert or delete operations may cause the tree to become unbalanced (we'll focus on insertion, mainly). The tree must realise this when it happens and then take corrective action before the entire insert/delete operation finishes/returns.

Extra data structure requirements:

A *red-black tree* is a binary search tree with one extra attribute for each node: the *colour*, which is either red or black. We also need to keep track of the parent of each node.

Don't worry about the *exact* reasons why we need those extra elements ***yet***. Suffice to say for now: the colour attribute helps us to maintain the balance of the tree. The *parent* reference allows us to rotate the tree when we need to re-balance (during an insert/delete) via some form of tree rotation.

Properties of a Red-Black Tree

These properties make very little sense as standalone properties. However, when we consider adherence to these properties we should being to see how they help to self-regulate the tree.

Also, just note that we consider leaf nodes (null nodes) as actual nodes for the purposes of the algorithms and data structure.

1. Red-Black Tree must be a Binary Search Tree.
2. The Root node must be Black
3. The children of a Red coloured Node must both be Black (a Red node cannot have a Red child node).
4. In **all** paths from a root to a leaf there must be the same number of Black nodes.

Two ancillary point to note are: every time we do an insert, the node we insert is *initially* coloured Red; Every leaf (Null node) must be considered Black.

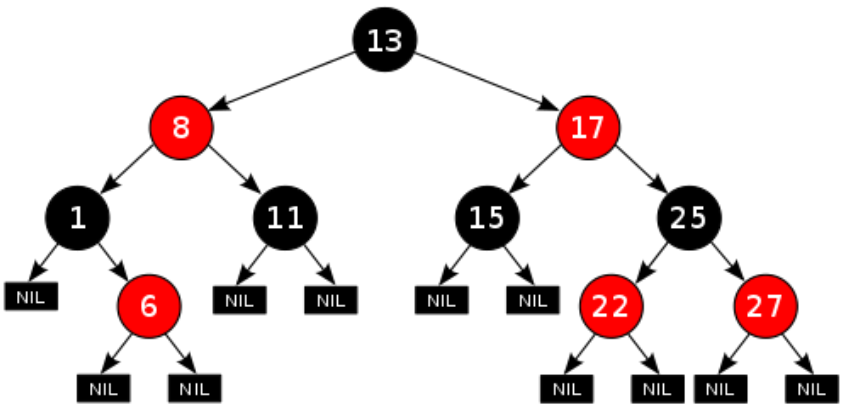

Figure 1: A Red-Black tree (it satisfies all the properties defined.)

Insertion into a Red-Black Tree

We perform a normal insert operation (colouring the new node Red.) This can result is two primary possibilities.

1. The insertion has not violated any of the properties, i.e. the tree is still a Red-Black Tree.
2. We violated one of the properties (at a cursory examination, I think we can only violate one of the properties via our insertion.)
   We must take corrective action to return our tree to a valid Red-Black Tree; this is called *balancing* the tree.

We use two tools to do balancing:

1. Recolouring.
2. Rotation.

Before we look at the actual balancing requirements of a Red-Black tree let's take a slight detour into rotations.

Rotating a Tree/Sub-Tree

Hopefully from the diagrams you can begin to appreciate the role of rotations in balancing. Some texts refer to **"A" as the root** (pre-rotation, of course) and **"B" as the pivot**. Since a subtree is itself a tree, this could refer to any sub-portion of an already existing tree (assuming that there are enough nodes to allow the operation to complete.)

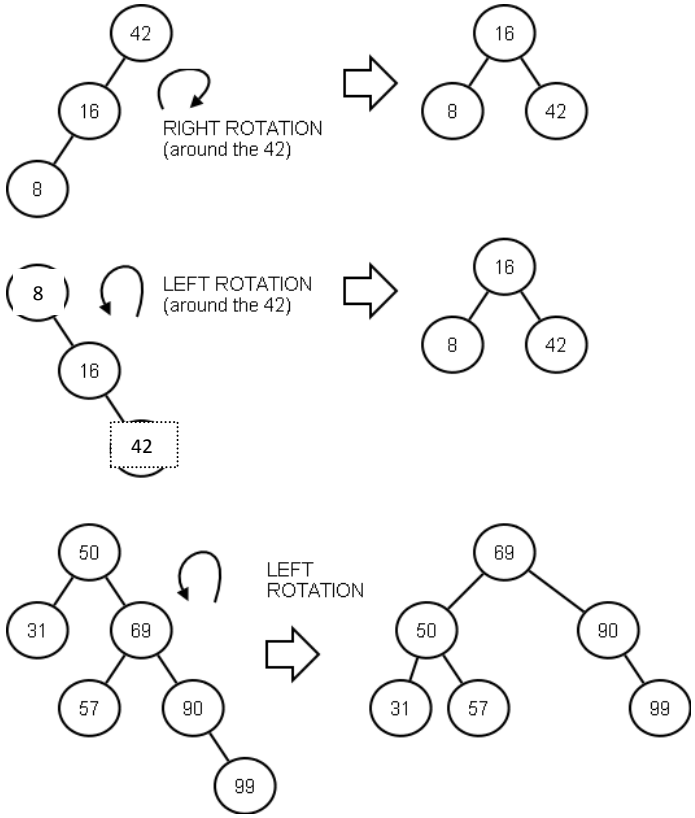

Figure 2: Some illustrations of right and left rotations

In general:

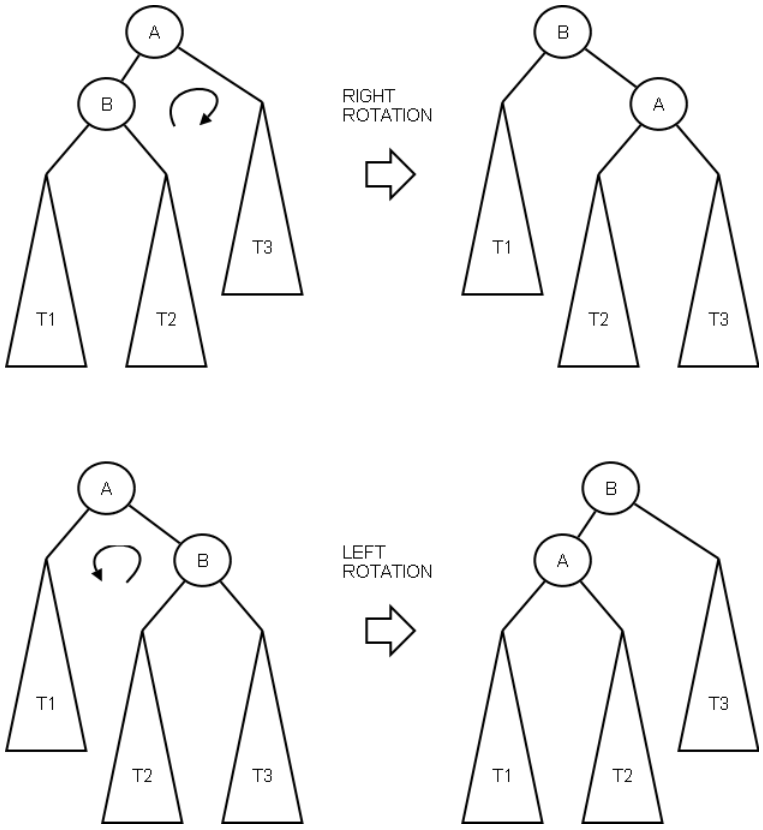

Figure 3: Generalised tree rotations.

As an in-class exercise, we will attempt to establish what operations are required for a left rotation. Probably the simplest way is to use the tree in the diagram with nodes 31, 50, 57, 69, 90, 99 for analysis.

After that you should be able to turn it into code:

1. Write a method called `public void rotateLeft()` for our BinarySearchTree.
   Note: this is just a test method that will rotate the whole tree. Later we can modify it to rotate a given sub-tree.

2. The code is only a few lines long. Hint: you may need to use temporary reference variables (in case, as you perform operations, you overwrite some references that you will need access to later.)

3. How could you then verify - from your test program - that the tree had indeed rotated? Do you think, for example, that printing out an in-order traversal will achieve this?

4. After you have successfully implemented and verified the method, provide a `public void rotateRight()` method.

Potentially Useful WebLinks for Tree Rotation

https://en.wikipedia.org/wiki/Tree_rotation

http://www.csanimated.com/animation.php?t=Tree_rotation

http://tommikaikkonen.github.io/rbtree/