AADS Lab week 4

At this stage you should be able to handle a violation case where there is a **red uncle**. This involves some recolouring and recursion. You should be able to verify that the correct operations have taken place using a pre-order traversal, assuming that your toString() now prints the colour of the node. As I said last week, you need to provide a valid test case.

Handling a black-coloured uncle node

As you should be aware, there are four sub-cases here (two are mirror images, so effectively there are two).

I would suggest attempting to handle one of the easier sub-cases (either Left-Left or Right-Right) initially. Then create a tree that will invoke that code and, from there, you can verify its operation using a pre-order traversal.

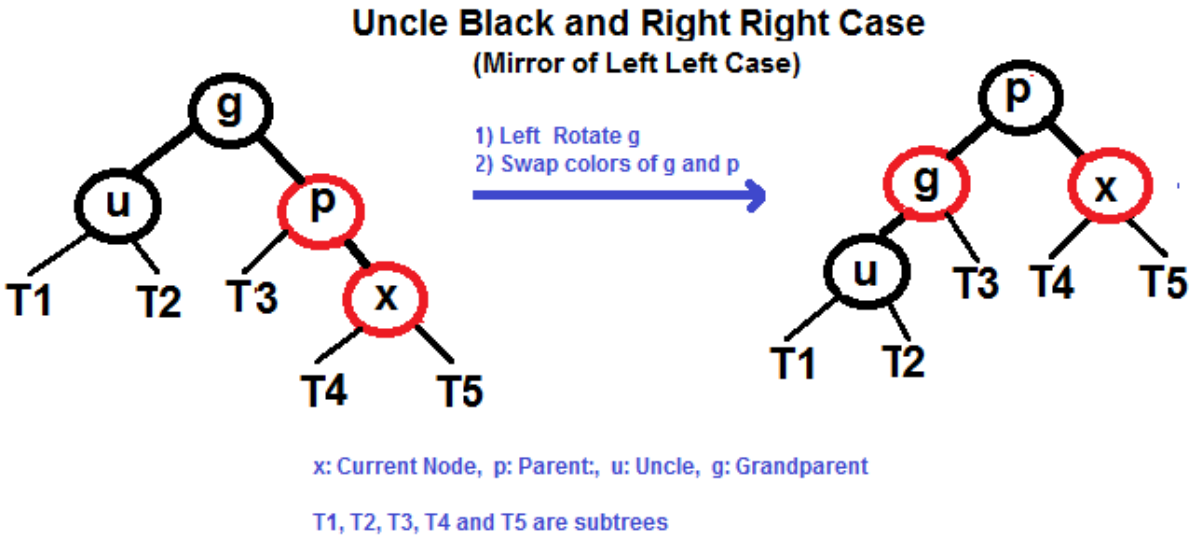
What should your test-case be?

Since the assumption is that null (or empty) leaf nodes are coloured black, the following insertions into a tree would invoke the Right-Right case: 1,2,3.

Pen and paper exercise: Step through each insertion and assess what the operations should be at each stage.

Implementation

Assuming that we're trying to implement and validate the Right-Right case, shown below:



We already have a method called **void** handleRedBlack(Node<T> newNode)

At the moment (assuming you've completed the previous practical) it checks if there is a violation, i.e. the parent of the new Red node is itself Red.

If that is the scenario we check for the specific case. Already we've handled a Red uncle.

If this is not the case, I'd like to be absolutely sure that we're dealing with a Black uncle before going on to figure out which subcase we're faced with. Something like this would suffice (you could perhaps raise an exception or output some debug info if this test does not pass):

```
//Check if uncle is Black (four subcases to handle)
//http://www.geeksforgeeks.org/red-black-tree-set-2-insert/
else if((uncle == null) || !uncle.nodeColourRed)
```

Figuring out which of the four cases it actually is

Before you concentrate specifically on the Right-Right case, see if you can build up the conditional code which figures out which of the cases you're faced with.

```
if (current situation is Left-Left)
{
   System.out.println("Left-Left case detected");
}
else if (current situation is Left-Right)
{
   System.out.println("Left-Right case detected");
}
else if....
```

And what conditions should the if else-if statements test for?

Answer: it's simply a matter of mapping the diagrams in Appendix A into conditions. Each of the sub-cases differ in the **parent's location relative to the grandparent** and the new node's **location relative to the parent**.

Exercise: when you've completed this, create trees to test each of the four scenarios.

Handle Right-Right

Assuming we established that it is a Right-Right case. You can then call a method to perform the required operations. Something like:

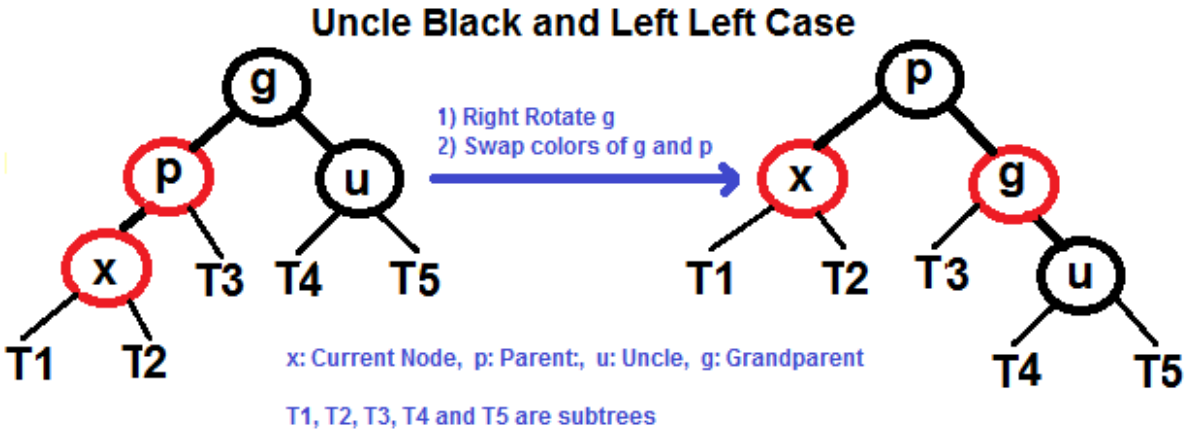
```
applyLeftLeftCase(???)
```

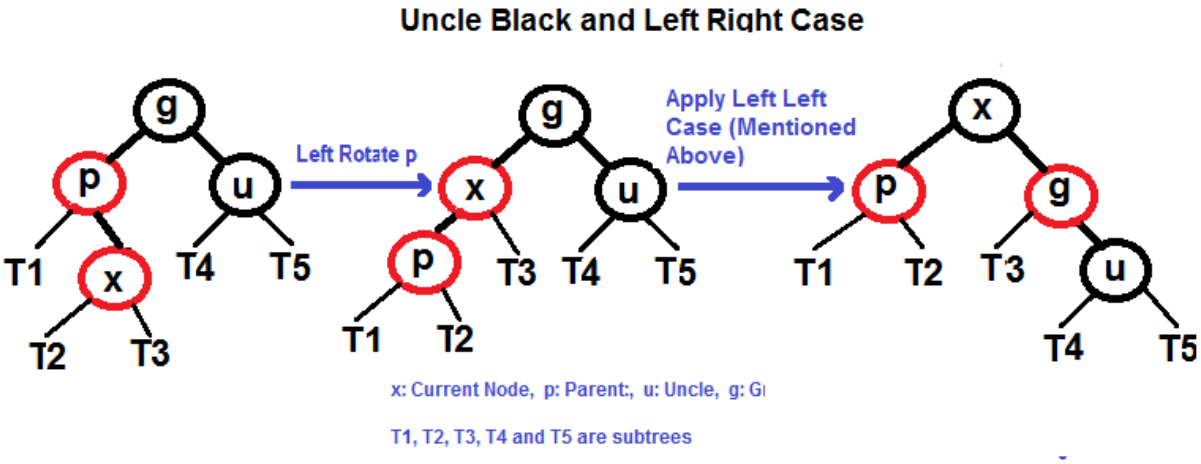
I've left the parameter list (and the return type) for you to figure out.

Note: Although your test case (insert 1,2,3) presents a scenario where the grandparent is the root node, this might not always be the case (our balancing could easily be in a subtree many levels deep in the tree structure).
If you do get the code to work today using this assumption, then you'll need to eventually change it for these other scenarios.

Appendix A: Subcases for Black Uncle Node

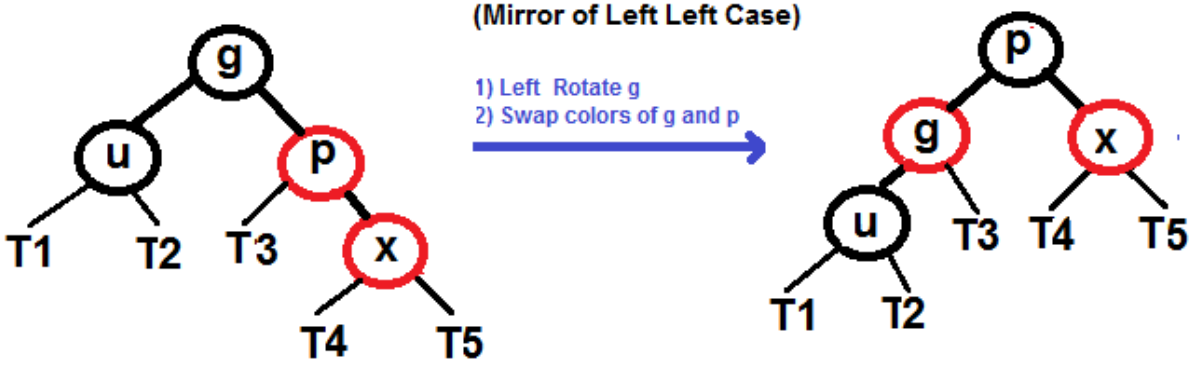
Remember, if you're building trees to test for each of these sub-cases, it is valid for the uncle node to be null.





Uncle Black and Right Right Case

(Mirror of Left Left Case)

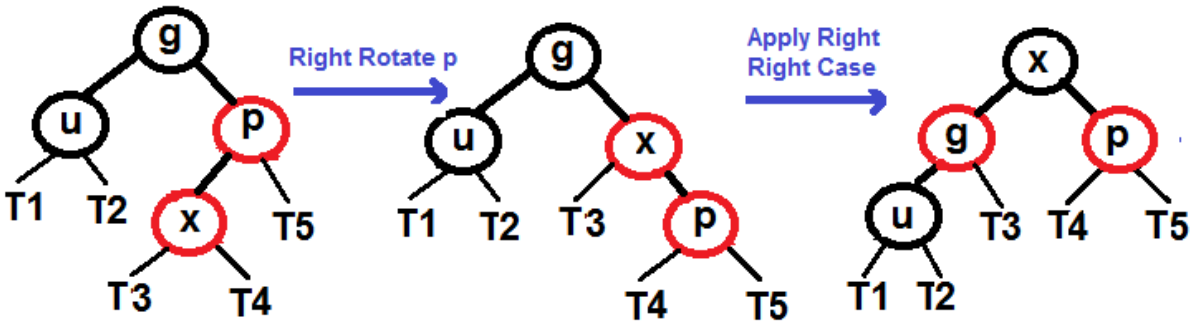


x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

Uncle Black and Right Left Case

(Mirror of Left Right Case)



x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees