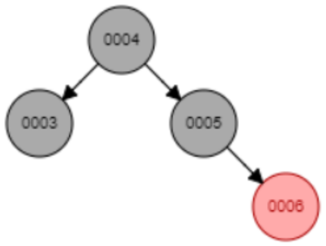
Jacob Christiansen
CSCI 2270 - Data structures and algorithms
Instructor: Hoenigman
Assignment 7

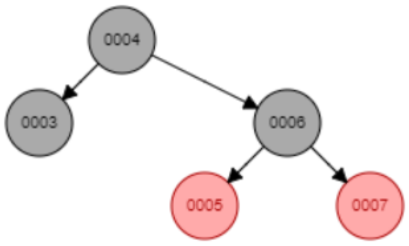
Question 1: *Does inserting a node into a red-black tree, re-balancing, and then deleting it result in the original tree?*

No, you will not get the same tree as before the insertion. The reason for this is because of the re-balancing of the tree, as well as the re-coloring. Here is some images of an example tree.

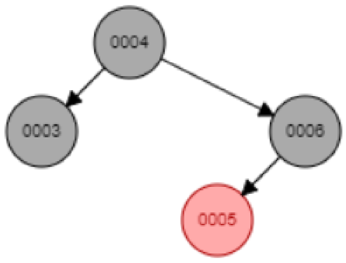
Before inserting **7**:



After inserting **7**:



After deleting **7**:



Here is a step-by-step process of how the algorithm handles both the insertion and deletion.
INSERTING ***7****:*

a. Search the tree and find where **7** should be placed. In this tree it follows the path **4**->**5**->**6**->[Insert **7** Here]
b. Insert **7** after **6**. **6**'s rightChild is now **7**, **7**'s parent is **6**, and **7**'s children are **NULL**.
c. Check colors to make sure that there are no conflicts. In this case there is! **6** is red, and **7** is also red. Red nodes cannot have red children.
d. To handle this, it can't just recolor **6** to black, because then the tree would be unbalanced. So instead, a left-rotation must be performed, with **6** as the argument. **6** becomes **4**'s rightChild, **5** becomes **6**'s leftChild, and **7** remains **6**'s rightChild. All of the parent/children pointers are set between each. Finally, since there is still a color conflict, **6** is changed to black, and **5** to red.

DELETING ***7****:*

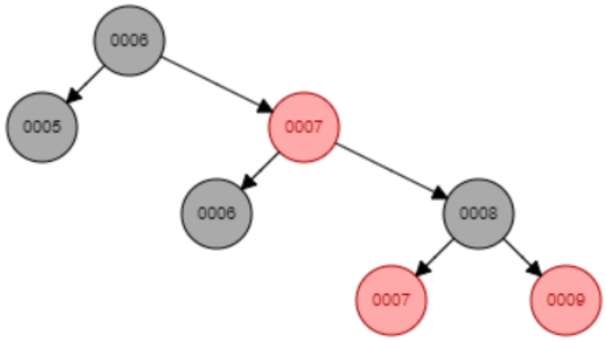
a. Search and find the node with the given value **7**. In this tree it follows the path **4**->**5**->**6**->**7** [found]
b. Check if the node is the root of the tree (it's not)
c. Check if the node has two children, one child, or zero children
d. Check if the node is a left or right child
e. The **7** node has no children, and is a right child
f. Set **7**'s parent equal to **NULL**
g. Set **6**'s rightChild equal to **NULL**
h. Delete **7**
i. Check colors to make sure that there are no conflicts (there aren't)

Now we have complete the insertion and deletion. Our current tree is now different because of our rebalancing, so instead of **6** being a rightChild to **5**, now **5** is a leftChild to **6**.

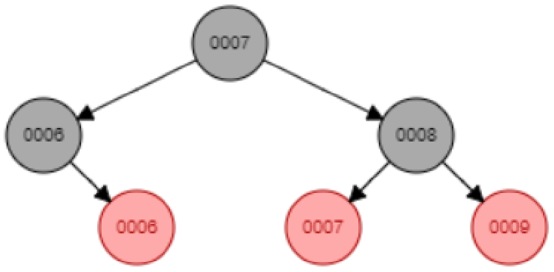
Question 2: *: Does deleting a node with no children from a red-black tree, re-balancing, and then reinserting it with the same key always result in the original tree?*

No, for mostly the same reason as before, a re-balance in a tree will switch where the same node will be placed, before and after. Here is an example:

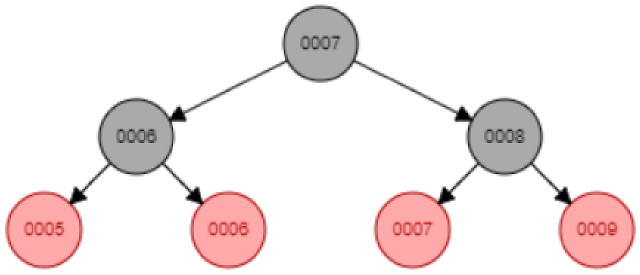
Before deleting **5**:



After deleting **5**:



After reinserting **5**:



<u>Here is a step-by-step process of how the algorithm handles both the deletion and insertion.</u>
*DELETING **7**:*

a. Search and find the node with the given value **5**. In this tree it follows the path **6**->**5** [found]
b. Check if the node is the root of the tree (it's not)
c. Check if the node has two children, one child, or zero children
d. Check if the node is a left or right child
e. The **5** node has no children, and is a left child
f. Set **5**'s parent equal to **NULL**
g. Set **6**'s leftChild equal to **NULL**
h. Delete **5**

i. Check colors to make sure that there are no conflicts. In this case there is! **6** is black, and **5** is also black. Because **5** is gone, the tree is unbalanced.
j. To handle this, a left-rotation must be performed, in this case with **7**, the next red node, as the argument. **6**(org. root) becomes **7**'s leftChild, **6**(child of **7**) becomes **6**(org. root)'s rightChild, and **7**'s rightChild branch remains the same. All of the parent/children pointers are set between each. Finally, there is still a color conflict, **7** is changed to black (as it's the root), and **6**(child of the other **6**) to red.

INSERTING ***7****:*

a. Search the tree and find where **5** should be placed. In this tree it follows the path **7**->**6**(black)->[Insert **5** Here]
b. Insert **5** after **6**(black). **6**(black)'s leftChild is now **5**, **5**'s parent is **6**(black), and **5**'s children are **NULL**.
c. Check colors to make sure that there are no conflicts. In this case there aren't.

Now we have complete the deletion and insertion. Our current tree is now different because of our rebalancing, so instead of **5** being a leftChild to **6**(org. root), now **5** is a leftChild to **6**(org. root) which is a leftChild to **7**(new root). Also, **5** is now a red node and not a black node.