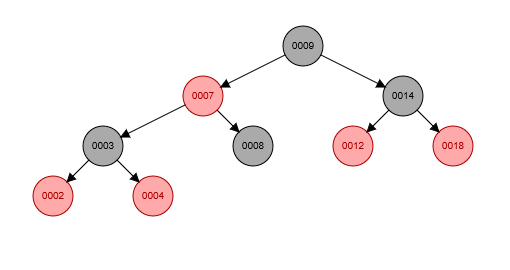
Assignment Seven

Muntaha Pasha

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

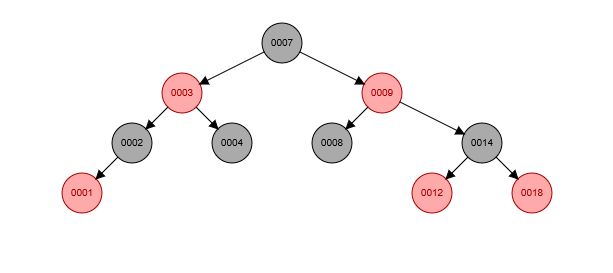
* *My answer is that this will* ***NOT*** *result in the original tree, because when the node to be added is red and it is added TO a Red node, the tree will have to rotate, and it will no longer be the same when you delete the same node from the tree. For example, let’s take this tree here.*



* *Let’s say I add a 1 to this tree. The entire tree will have to rotate and balance itself out in order to make sure there are equal black nodes from the root to the base for all of the branches in the tree.*

***Insert One***

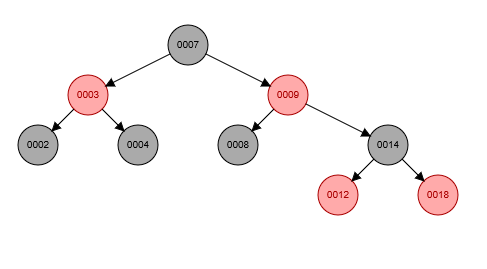
*The Algorithm for insert will be called, x will be a red node, while x isn’t the root, and the parent color is red, it will enter the loop. So it’ll go down 9’s LEFT branch, starting at 7, and making it’s way down to 3, to 2, because it’s checking the value of each number with “x” (x=1 in our case) and seeing if it’s greater or less than the value we are currently on. When it gets to 2, it will be set as 2’s LEFT child, following the properties of an RB Tree, its parent will be 2, its grandparent will be 3, and the uncle will be 4. It will not go down to 4, because the “if(uncle.color==red)” condition will fail here given that 4 is a BLACK node. So because the uncle node WAS red, and x is it’s parents LEFT child which is true, so it will perform a LEFT rotate, and x will have 2 NULL children.*



* *Following the algorithm for insert, the 1 inserted is a RED node, and the tree calls RB Balance so that there is equivalent black nodes from the root to base for all branches. So the 2, and the 14 fulfill that requirement. Now, when I delete 1, the tree remains the same, and the root is now 7 as opposed to being 9. 1’s uncle is 4, grandparent is 3, parent is 2.*

***Delete/Rebalance One***

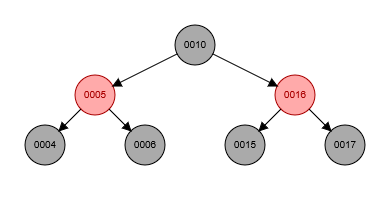
*The Algorithm for deleting one is that it would search for the value of “one” to delete, it would find it, and see that it has 2 NULL children, so it makes the delete process much easier, because it’s a leaf node it doesn’t have to be rebalanced. It would simply delete one, its parents would not need any recolorization, rebalance would not be called. According to the algorithm, because one does NOT have any children, the following code would run: node.parent.leftChild = nullNode and then x = node.leftChild. This would set its parents values to NULL nodes and then the value, in our case the number one, would be deleted.*



* *As you can see, the Red Black tree still fulfills the requirements assigned to it. There is an equal number of black nodes from the root to base for each branch, but now the original tree properties have not remained the same. 9 was the root, and was initially the black node, but is now red, and has 2 black children instead of 1 black and 1 red child. By doing these the original tree properties have not remained the same, so I have to* ***disagree*** *with the statement and say that no, inserting a node into a red-black tree, re-balancing, and then deleting it* ***does not result in the original tree****. Adding 1, which was a red node, to 2, which was also another red node caused the tree to rebalance itself making 7 the root as opposed to 9, and it had to recolor the original nodes to fit the rebalancing.*

***Question Two:*** *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?*

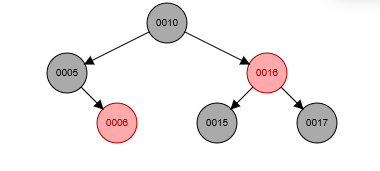
* *My answer is YES, it will always result in the original tree when the node to be deleted is a left node. It will always be the same if it is a left node with no children and is black.*

**

* *This is my tree here. Say I want to delete 4. 4’s parent is 5, uncle is 16, and grandparent is 10.*

***Delete/Rebalance Four:***

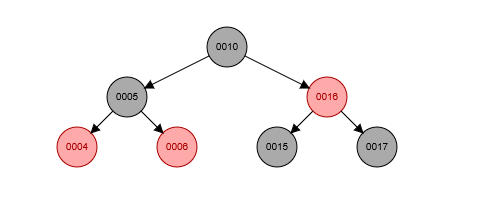
* *The algorithm would start, it would search for the value provided which is 4, it’ll see that 4 is not the root, then go down to the next condition, and see that it is a black leaf node with NO children, so it would then call Rebalance on 4, while it’s not the root and it’s black, which are true statements, it will enter the while loop, say that if 4 is it’s parent left child, which is it, then s (for sibling) is 4’s parent’s right child. In this case, 6. It’ll see that both nodes of the s.leftChild and s.rightChild are black, and therefore will recolor 6 to be red, and recolor 5 to be black. After rebalance is called, 4 will be deleted. The tree still holds all of it’s properties, so we are good.*

**

* *After deleting, we need to see how the tree behaves once we re-insert 4 into the tree.*

***Insert Four***

* *Insert algorithm will be called on 4, and it will use the exact same format as all other insertions of nodes. The algorithm will first add the node as a red node to the tree, off the left branch of the tree. Is it less than 9? Yes. Less than 6? Yes, and add it to the left of 6 which was initially a NULL child. So now it’s the left child of 5, 5 is 4’s parent, 4’s grandparent is 10, and 4’s uncle is 16. 4 will be a RED node because it is a newly added node, and both of its children will be NULL because it’s a leaf node. So, after inserting, we have the exact same tree, but 5’s children are now red instead of black.*

**

* *So, as shown step by step, the tree does NOT change in its structural integrity. 4 started as the left child of 5, it ended as the left child of 5, its uncle was always 16, and its grandparent was always 10 no matter the delete or the insert. The parent color changed from red to black, because when 4 was inserted it was no longer a black child, and a red parent MUST have two black nodes as children. The sibling color changed from black to red, and 4 also changed from black to red using the rebalance, but besides that, 4 has the same family structure, and the tree itself has the same root, and not much changed besides the colors.*