Assignment 5

Oceane Andreis
CSCI 2270 Section 310

PS: This assignment 5 is organized with the question first, then the answer, then an example, and then the explanation. My explanation includes explaining the whole insert/delete function given as well as which part of the insert/delete function applies to the question/answer. I tried to be as detailed as possible because I wanted to learn and understand as much as possible. (Thus my explanation is very long)

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

To answer this question we first need to establish the properties of a red black tree. (These will also be used throughout my examples):

Property 1: A node is either red or black. Property 2: The root node is black.
Property 3: Every leaf(NULL) node is black. Property 4: If a node is red, then both of its children must be black. Property 5: For each node in the tree, all paths from that node to the leaf nodes contain the same number of black nodes.(Taken from Visualizing Data Structures Book)

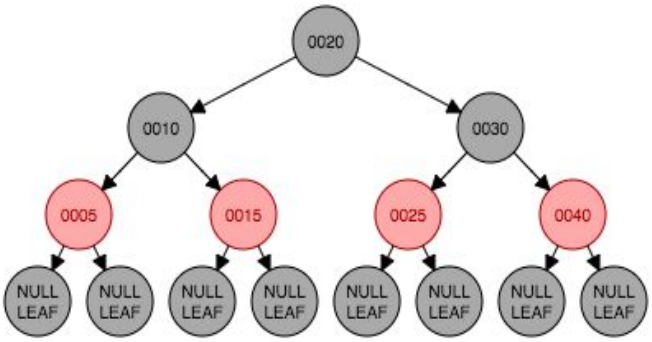
When inserting a node into a red-black tree there are a few cases that affect those properties thus when we have a violation we need to rectify it and rebalance the tree. We can do that with left and right rotations and recoloration.
Left and right rotations change the structure of the tree but since they are inverse of each other if you rotate in the other direction it will return to its original state.

In the case, where we are inserting a node, rebalancing it, and then deleting that node it will result in the original tree in some cases. I will be showing two cases where it does not go back to its original and one case where it does go back to the original.

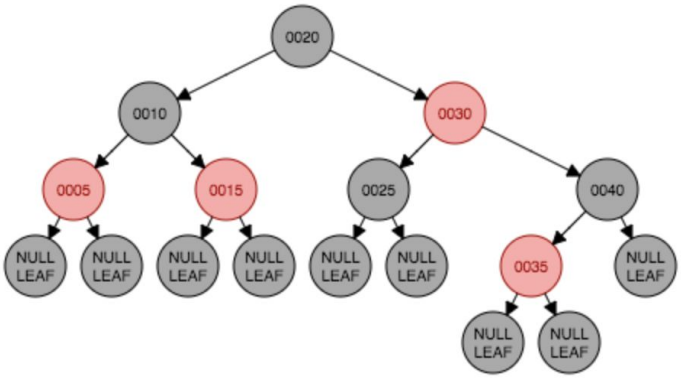
Below, in example 1, that is the case where when inserting a node into a red-black tree, re-balancing, and then deleting it result in a tree different than the original tree.
In example 2, that is the case where when inserting a node into a red-black tree, re-balancing, and then deleting it result in a tree different than the original tree.
Both cases show two different trees in the end.
In example 3, that is the case where when inserting a node into a red-black tree, re-balancing, and then deleting it result in the original tree. Adding a red node to a red node = not the same tree. Adding to a black node = the same tree (Example below).

Example 1.1: Insert 20,10,30,5,25,15,40,35(Delete 35)

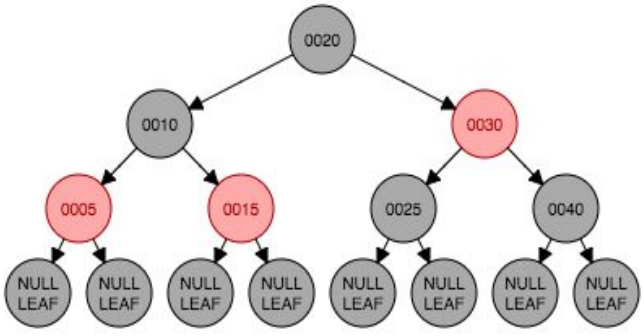
1. This is the original Red Black tree. Property 1 and 2 are displayed here



2. Here, I added 35. (When I add a node it's always red and we needed to recolor)

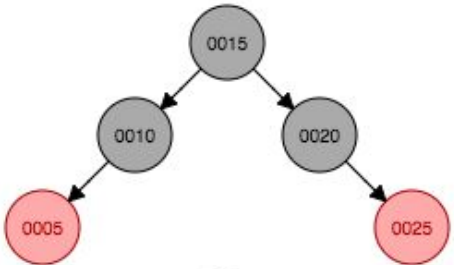


3. Final tree after deleting: I deleted 35 and the tree might symmetrically look the same but now 30 is a red node and 25 and 40 are black nodes. Hence, not the same as the original tree.

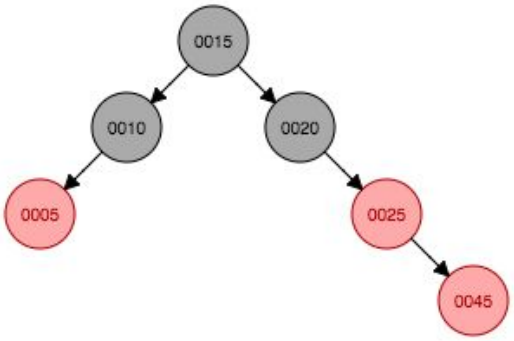


Example 1.2: Insert 10,15,20,25,5,45 (delete 45)

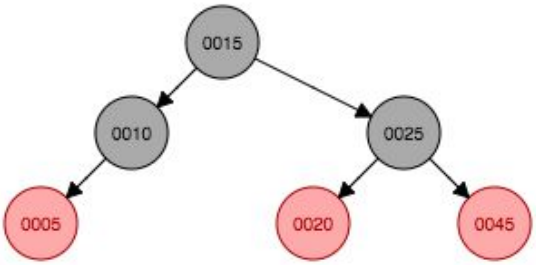
1. 1.This is the original RBT



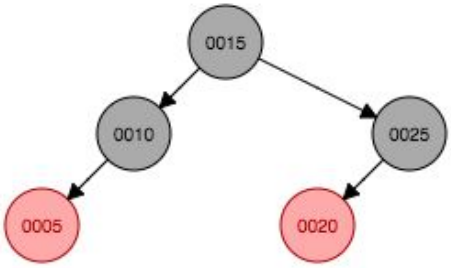
2. I added 45 and as we can see there are two red nodes in a row. Thus, this is a violation. So we need to rebalance it.



3. The results of the re-balancing: We rebalanced the tree since the node and parent are both red. We did a single rotate left to fix the violation.



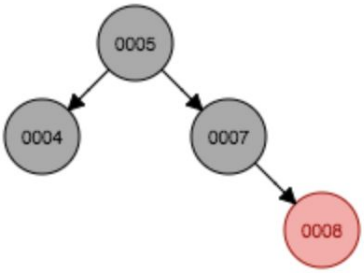
4. Final tree after deleting: I deleted 45 and it does not look like the original tree. (The tree doesn't need to rebalance after the deletion)



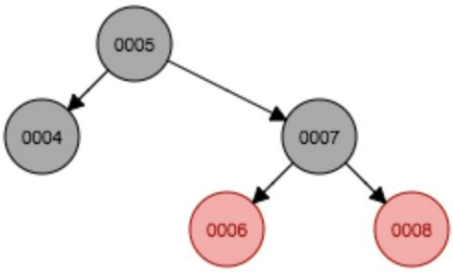
Example 1.3:

Insert 5,4,7,8,6 (delete 6)

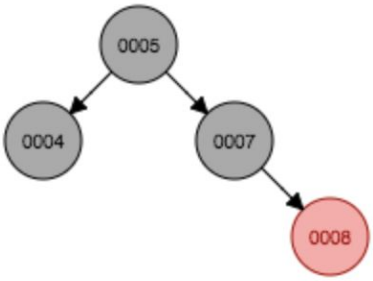
1. This is the original RBT



2. We insert 6, we don't need to rebalance the tree because it's already balanced.



3. Final tree after deleting 6
   (Same as original)



Explanation:

X is a pointer that's color red which we want to insert.While what we are inserting is not the root and that its parents is red; there are a few cases that follows to re balance the tree.

if(uncle.color == red):(Case 1)
That's the case when the new node inserted is red and that its uncle is red. You have to recolor the parent and the uncle to be black and then the parent of those to be red (affects grandparents)(red, black, red, black kind of organization).
Hence, in that if statement the recoloring is applied.

The first else statement:(Case 2)
Another case is when the new node's uncle is black and that the new node is a right child then you want to do a left rotate.

Then another scenario(Case 3) which can follow that previous scenario is when the node is a left child and that its uncle is black. Then you need to recolor the parent of that new node and its grandparent. Then you want to do a right rotation on the grandparents.

Insert algorithm

```
redBlackInsert(value)
      { x = insert(value) //add a node to the tree as a red node
      while(x != root and x.parent.color == red){
            If(parent == x.parent.parent.left){
                  uncle = x.parent.parent.right
                  if(uncle.color == red){
                        x.parent.color = black
                        uncle.color = black
                        x.parent.parent.color = red
                        x = x.parent.parent
                  }else{
                        if(x == x.parent.right){
                              x = x.parent
                              leftRotate(x)
                        }
                        x.parent.color = black
                        x.parent.parent.color = red
                        rightRotate(x.parent.parent)
                  }
            }else{
                  //x.parent is a right child. Swap left and right for algorithm }
      }
      root.color = black
}
```

The last else is if we are adding the new node to the right side of the tree(Thus, we would have to copy all the previous conditions but change the algorithm so it applies to the right instead of the left)
In my first example, when I added the new node, the uncle is red thus it is like uncle.color == red(case 1, highlighted). Thus we have to recolor.
In my second example, the violation was the new node was red and its parent was red. Thus we just needed to do a left rotation on the node with a key of 20. In my third example, we didn't need to rebalance when we inserted or deleted the node.

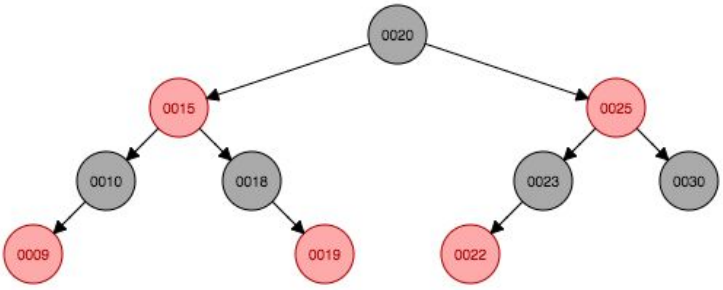
Question 2: Does deleting a node with no children from a red-black tree, rebalancing, and then reinserting it with the same key always result in the original tree?

Using back the properties from question 1, deleting a node with no children from a red-black tree, re-balancing, and then reinserting it with the same key does not always result in the original tree.
When deleting a node there are 3 cases to consider. When the node has no children, or one child, or two children.
For this question, we are approaching it from the no children case. When deleting a node with no children we can have the violation in my example below (which is the violation that violates Property 5: For each node in the tree, all paths from that node to the leaf nodes contain the same number of black nodes).

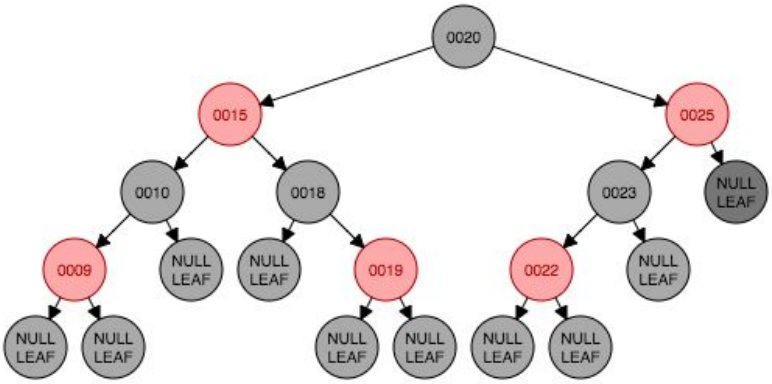
In the example 1 below, I deleted a node with no children and a violation happens thus in that specific case I need to perform a rotation. When I go and reinsert the node I deleted, the tree doesn't look the same (because of the rotation) and so then the node (30) is added to the right and its grandparent is no longer the same as the original tree.

Example 2.1: Insert 20,25,15,10,23,18,30,9,22,19

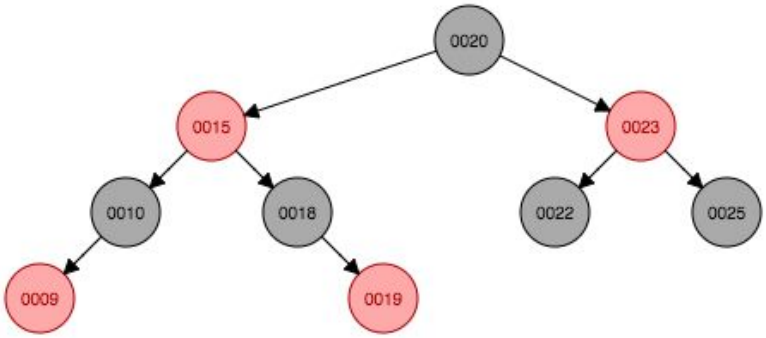
1. This is the original RBT.
   We want to remove 30 and it doesn't have any children.



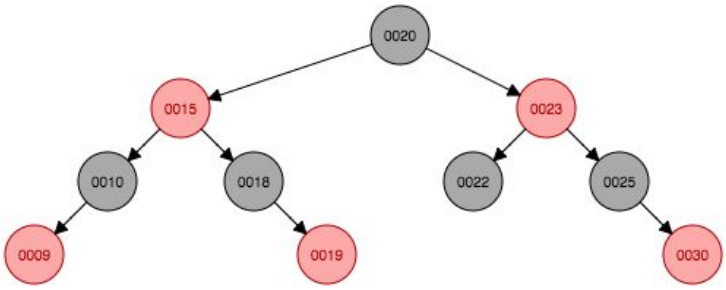
2. We removed 30, now we are in violation of Property 5. We need one rotation(right rotation) to fix the double-blackness.



3. The results of the rebalancing:
   After a right rotate on 25 this is what the tree looks like.



4. This is what the tree looks like after we re insert 30. It does not look like the original tree.(We don't need to re balance)



Explanation:

Before deleting a node you need to find the node in the tree.

If the node we want to delete is not the root then we have many scenarios where we can delete a node.

If the node we want to delete doesn't have children then we want to set the node we want to delete to nullNode and then we set that to be x.(We also have to re balance it, which will be explained below)

If the node we want to delete has two children we want to find the minimum in the right side of that subtree(The subtree that the node that wants to be deleted belongs to).

If the node we want to delete has one child then that child will be the replacement node. It will replace the node that was deleted (its spot).

In that last else, since this code was only for the case where the node to delete is the left child of its parent we would have to repeat the cases for 0,1,2 children again and change it so it works for the right child.

If the node that replaced the deleted node is black then we have to rebalance the tree.

In my example 2.1, since we only needed to delete a leaf only the first part of this delete function applies(Highlighted part).

Delete algorithm

```
redBlackDelete(value){
      node = search(value)
      nodeColor = node.color
      if(node != root){
            if (node.leftChild == nullNode and node.rightChild == nullNode){
            //no children
                  node.parent.leftChild = nullNode x = node.leftChild
            } else if (node.leftChild != nullNode and node.rightChild !=
      nullNode){ //two children
                  min = treeMinimum(node.rightChild)
                  nodeColor = min.color //color of replacement
                  x = min.rightChild
                  if (min == node.rightChild){
                        node.parent.leftChild = min
                        min.parent = node.parent
                        min.leftChild = node.leftChild
                        min.leftChild.parent = min
                  } else {
                        min.parent.leftChild = min.rightChild
                        min.rightChild.parent = min.parent
                        min.parent = node.parent
                        node.parent.leftChild = min
                        min.leftChild = node.leftChild
                        min.rightChild = node.rightChild
                        node.rightChild.parent = min
                        node.leftChild.parent = min
                  }
                  min.color = node.color //replacement gets nodes color
            }else{ //one child
                              x = node.leftChild
                              node.parent.leftChild = x
                              x.parent = node.parent
                  }else{
                        //repeat cases of 0, 1, or 2 children
                        //replacement node is the new root
                        //parent of replacement is nullNode
                  }
                  if (nodeColor == BLACK){
                        RBBalance(x)
                  }
                  delete node
      }
```

Explanation:

We need to rebalance the tree when x is not the root and when x is black.

If x is the left child of a black node(its parent is black)(Case 1). S is the sibling of x and if the sibling of x is red then we need to recolor S to be black and the parent of S and x to be red. Then we have to do a left rotation on the parent of x.

In the if statement where the children of S are both black (Case 2) then we need to recolor S to be red and have x point to its parent.

If the left child of S is red and if the right child of S is black (Case 3) then we have to recolor the left child of S to be black and we need to recolor S to be red. Then we need to perform a right rotation on S.

Then if the left child of S is black and if the right child of S is red (Case 4). We have to recolor s to be the same color as x's parent(which is also s's parent). Then we set x's parent to be black and the right child of s to be black. Then we perform a left rotate on x's parent and set x to the root and that makes it exit the while loop. Outside of the while loop we want to make sure to recolor the root to be black.

Red-black rebalancing after delete

```
RBBalance(x){
      while (x != root and x.color == BLACK){
            if (x == x.parent.leftChild){
                  s = x.parent.rightChild
                  if (s.color == RED){ //Case 1
                        s.color = BLACK
                        x.parent.color = RED
                        leftRotate(x.parent)
                        s = x.parent.rightChild
                  }
                  if (s.leftChild.color == BLACK and s.rightChild.color ==
                  BLACK){ //Case 2
                        s.color = RED
                        x = x.parent

                  } else if (s.leftChild.color == RED and s.rightChild.color
                  == BLACK){ //Case 3
                        s.leftChild.color = BLACK
                        s.color = RED
                        rightRotate(s)
                        s = x.parent.rightChild
                  }else{
                        s.color = x.parent.color //Case 4
                        x.parent.color = BLACK
                        s.rightChild.color = BLACK
                        leftRotate(x.parent)
                        x = root
                  }
            }else{
                  //x is a right child //exchange left and right
            }
      }
      x.color = BLACK
}
```

In my example 2.1, I needed to rebalance the tree because it violated property 5. My example follow case 3 but on the right side. Since my

node was black and didn't have any children but its sibling had a left child that's red and a right child that's black(null) and then I just needed to do one right rotation on node 25.