Environment

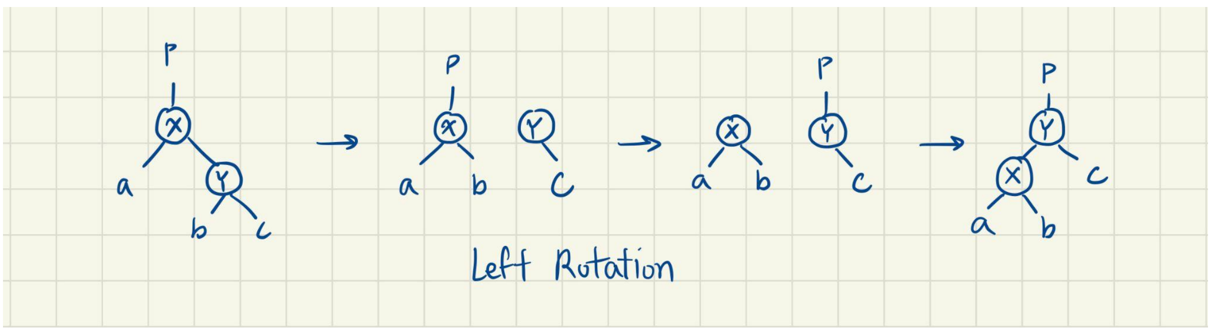
I used C++ version 13.0.0 to perform this program.
The below commands in a terminal can successfully run my program.

```
~ % g++ rbtree.cpp -o name
~ % ./name
```

Report

This homework was a lot harder than the previous homework in my opinion. Red Black Tree has a lot of rules to maintain and is not easy to comprehend. The hardest part is to maintain its color, especially in the deletion function.

The rotation function is almost identical to the AVL Tree's rotation function except some details such as AVL Tree's "NULL" will be set to "NIL" if they are the leaf nodes. I have learned the AVL Tree in data structure from last semester so it was not hard to understand. Take leftRotate for example: leftRotate happens when the right branch of the tree is too long and causing the tree to be unbalanced. To fix this problem, let *X* be the top of the rotation and *Y* be *X*'s right child. Save *X*'s new right child to be *Y*'s left child. If *Y*'s left child is not NIL, then save its parent to be *X*. Switch *Y*'s new parent to be that of *X*'s. If *X*'s parent is NULL, then it's the root node, so set *Y* to be the new root. If it's not NULL and *X* was the left/right child, make *X*'s parent to have *Y* in the left/right. Finally set *Y*'s left children to be *X* and *X*'s parent to be *Y*. This concludes the leftRotate function. As for rightRotate function, it would be the exact same concept but with the opposite left/right direction.



The insertion function contains of two parts: insert the new node into the tree and fix the color. To insert the new node, first create a new node using the newNode function. Then use a while loop to find the location to insert the node correctly. If the node is the root or if it lays on the second layer, no colors have to be fixed, otherwise, fix the color. To fix the color, we have to know the new node's uncle. Let node's parent be *P*, uncle be *U*, and grandparent be *G*. All the cases below will terminate a big while loop if the node becomes a root or *P* is black.

Case 1: Both *P* and *U* are red.
  → Change *p* and *U* to black and *G* to red.
  → Change node to be *G*.
Case 2: *P* is a right child, *P* is red and *U* is black
  → Perform a right rotation on *P* only if the node is a left child and change node to be *P*.
  → Change *P* to black and *G* to red.
  → Perform a left rotation on *G*.
Case 3: *P* is a left child, *P* is red and *U* is black
  → Perform a left rotation on *P* only if the node is a right child and change node to be *P*.
  → Change *P* to black and *G* to red.
  → Perform a right rotation on *G*.

The deletion function consists of three parts: find the desired node, delete the node, and fix the color. To find the desired node, use a while loop to search through the tree like a normal binary search tree. To delete the node, let the node be Z and remember the original color of the node to be *color* and:

Case 1: *Z* has no left child.
  → Remember the *Z*'s right child as *X*.
  → Swap *Z* and its right child.
Case 2: *Z* has no right child.
  → Remember the *Z*'s left child as *X*.
  → Swap *Z* and its left child.
Case 3: *Z* has both children.
  → Set *Y* to be the smallest node on the right branch.
  → Update *color* to be *Y*'s color.
  → Let *X* be *Y*'s right child.
  → If *Y* is just below *Z*, let *X*'s parent be *Y*.
  → Otherwise, swap *Y* and its right child, then move *Z*'s right child, to be *Y*'s right child.
  → Swap *Z* and *Y*.
  → Delete *Z*.

For the color fix in deletion, if *color* is red, then we're done. However, if *color* is black, we have to fix it. Create a node *s* with *s* being X's sibling and let *p* be X's parent. The following cases will terminate if X becomes the root or if X's color is red.

Case 1: If *X* is a left child and *X* and *s* are both red.

→ Change *s*'s color to black and *p*'s color to red.
→ Perform a left rotation on *p* if *X* is a left child.
→ Change *s*'s color to red and *X* be p if both *s*'s children are black.
→ Change *s*'s left child to black, s to red, and perform a right rotation on *s* if *s*'s right child is black and left child is red.
→ Update *s*'s color to be *p*'s color, *p*'s color to black, and perform a left rotation on *p*.

Case 2: If *X* is a right child and *X* and *s* are both red.

→ Change *s*'s color to black and *p*'s color to red.
→ Perform a right rotation on *p* if *X* is a right child.
→ Change *s*'s color to red and *X* be p if both *s*'s children are black.
→ Change *s*'s right child to black, s to red, and perform a left rotation on *s* if *s*'s left child is black and right child is red.
→ Update *s*'s color to be *p*'s color, *p*'s color to black, and perform a right rotation on *p*.