Practice 8

Mayra Cristina Berrones Reyes

March 18, 2020

1 Introduction

The subject of this practice is Red-Black trees. But before we enter in detail on this type of trees, let us see some basic concepts first.

1.1 Binary trees

A binary tree is a data structure whose elements have at most two children, and they are typically named right and left child. The difference between this data structure and arrays, lists, queues or stacks, is that trees are hierarchical structures. They can be used if you want to store informations that is normally in a hierarchy form, such as a file system for a computer. The top node is called root of the tree. The nodes directly under it are called its children. When this nodes have more nodes below they are called parent nodes.

The main applications for binary trees can be [1]:

- Manipulate data that is ordered in a hierarchy form.
- The way is built, makes it easier to search for an element.
- It gives you a better manipulation of sorted lists of data.
- It can be used in router algorithms.

1.2 Balanced binary tree

A binary tree is balanced if the height of the tree is $\mathcal{O}(\log n)$ where n is the number of nodes. They are different forms in which a binary tree can obtain this balance. For Example, AVL tree maintains $\mathcal{O}(\log n)$ height by making sure that the difference between heights of left and right subtrees is no more than 1. Red-Black trees maintain the same property of height by making sure that the number of black nodes on every root to leaf paths are same and there are no adjacent red nodes.

2 Red-Black trees

A Red-Black tree is a self balancing binary search, in which the node contains extra information about the color of the node, that is either red or black. A Red-Black tree must satisfy the following properties [3]:

1. **Red-Black property:** Every node is colored either red or black.
2. **Root property:** The root is black.
3. **Leaf property:** Every leaf (NIL) is black.
4. **Red property:** If a red node has children then, the children are always black.
5. **Depth property:** For each node, any simple path from this node to any of its descendant leafs has the same black depth (the number of black nodes).

For the first and second properties, they are really self explicatory. The nodes can only be colored red or black, and as a rule, the root node must be black. In the introduction we mentioned which one is the root node.

Now we move to the third property, which says that every leaf is black. By leaf we mean a node that has no children. In this type of tree, when a node has no children, we put two nodes with null leaves, and this two nodes must be colored black.

The fourth property states that if a red node has children, then the children must always be black. With this property we make sure that red nodes are not adjacent to one another, which is important in order to comply with the fifth property that says for each node, any simple path from this node to any of its descendant leafs has the same black depth, which means it has the same number of black nodes on its way up. It is important to remember that this is a self balancing tree, so this limitation on property 5 helps to ensure that any path from root to leaf is balanced from all the other leaves. If this properties are not meet, then we need to apply certain steps to ensure that they do.

This steps or operations can be divided as follows [2]:

1. Rotating the sub-tree in a Red-Black tree
2. Left-right and right-left rotate.
3. Inserting an element into a Red-Black tree.

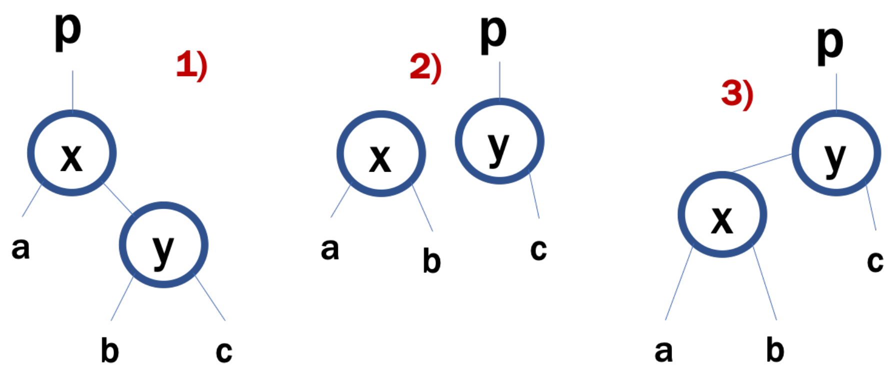


Figure 1: Example of rotating left sub-tree in a Red-Black tree

The first operation is used to maintain the properties of the tree when we use the algorithms of insertion or deletion and the properties shown above are not meet.

In Figure 1 we se an example of the algorithm used to rotate to left. To do this we have the original tree in example 1). Then we see that y has a left node. We give this node as a right child to x and move y upwards to be connected with the root node. Finally in example 3) we connect the node x to be the left child of node y. In case of right rotation is the same, but from example 3) to 1), and instead of conecting the left child of the moving node, is the right one, as we can see in Figure 2

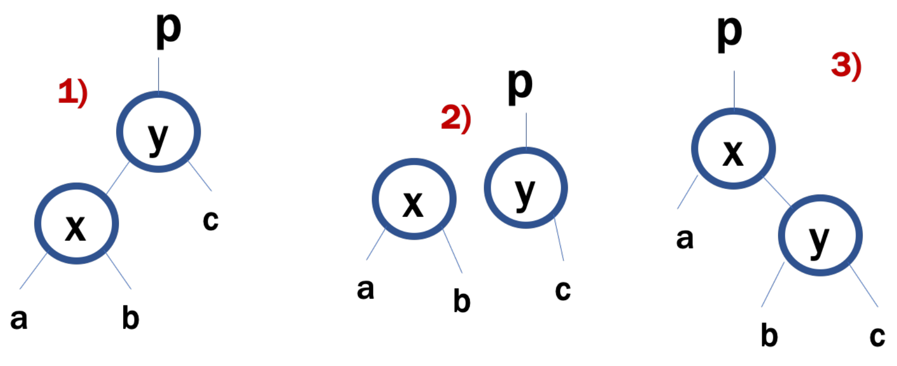


Figure 2: Example of rotating right sub-tree in a Red-Black tree

The left-right and right-left rotate are more or less the same as the first one, but in this case the arrangement of nodes are moved first to the left and then to the right, and vice versa as we can see in Figure 3. The example of right-left is similar to Figure 3, but we begin with right rotation.

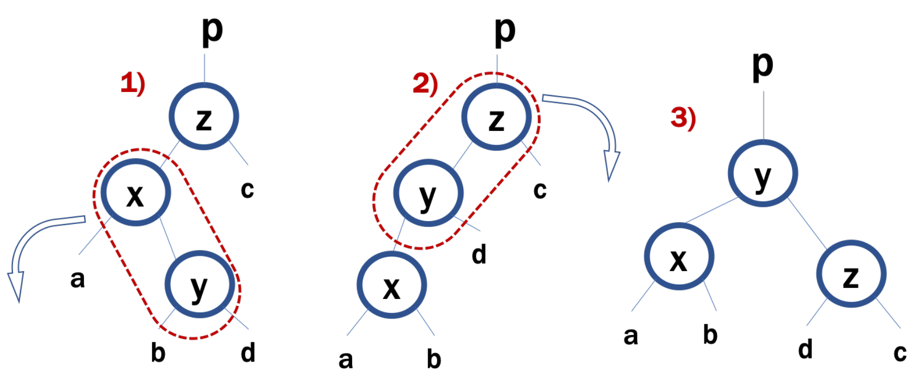


Figure 3: Example of rotate left-right

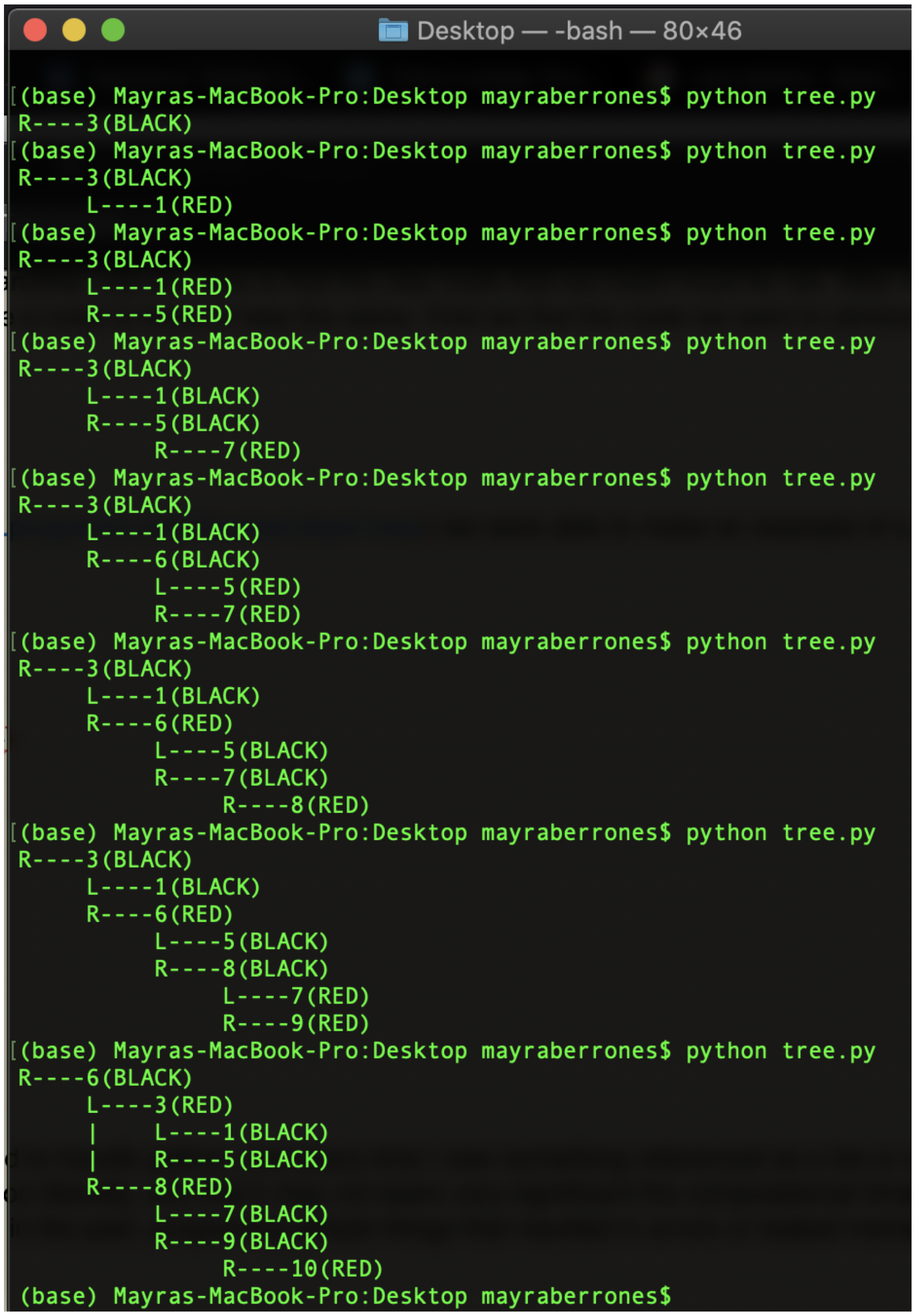
Then we have the third operation, which is inserting a new node. In this case another rule we have is that the new node that we insert must be red. After we insert this node, if the tree does not match the properties, we can either recolor nodes, or make rotations. To delete a node is more or less the same. First we find the node we want to eliminate, we take it out, and then we make operations to balance the tree.

3 Example

With the help of the code of the tutorial in Programiz [1] we were able to make an example of a Red-Black tree. We go inserting one node at the time in the order of *3, 1, 5, 7, 6, 8, 9, 10*. In Figure 4 we see the results on the console. When there is an indentation in the lines, it means the following nodes are the children of the node above.

To be able to understand it better and the steps it took to arrive to the final tree, we used a tool to build Red-Black trees [2] that we can find online. In each caption we explained the steps and operations made to ensure that the properties of the Red-Black tree were fulfilled.

[1] `https://www.programiz.com/dsa/red-black-tree`
[2] `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`



```
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
     L----1(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
     L----1(RED)
     R----5(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
     L----1(BLACK)
     R----5(BLACK)
          R----7(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
     L----1(BLACK)
     R----6(BLACK)
          L----5(RED)
          R----7(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
     L----1(BLACK)
     R----6(RED)
          L----5(BLACK)
          R----7(BLACK)
               R----8(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----3(BLACK)
     L----1(BLACK)
     R----6(RED)
          L----5(BLACK)
          R----8(BLACK)
               L----7(RED)
               R----9(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$ python tree.py
R----6(BLACK)
     L----3(RED)
     |    L----1(BLACK)
     |    R----5(BLACK)
     R----8(RED)
          L----7(BLACK)
          R----9(BLACK)
               R----10(RED)
(base) Mayras-MacBook-Pro:Desktop mayraberrones$
```

Figure 4: Output of the code to make Red-Black trees

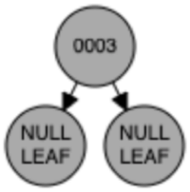


Figure 5: Here we see the first node as the parent node, and its two null nodes. This tree satisfies all the properties.

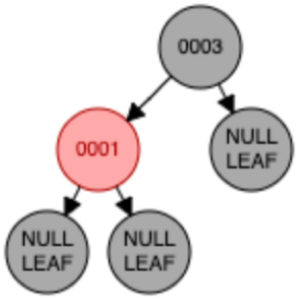


Figure 6: Now we add a red node with value 1. As is smaller than 3, it goes to the left. This does not need further manipulation.

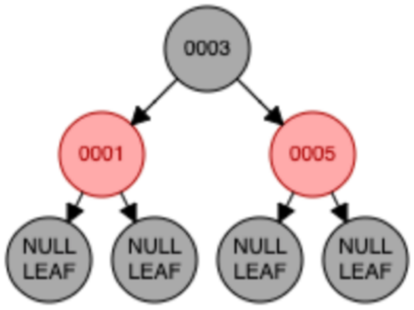


Figure 7: Now we add a red node with value 5. As is bigger than 3, it goes to the right. This does not need further manipulation.

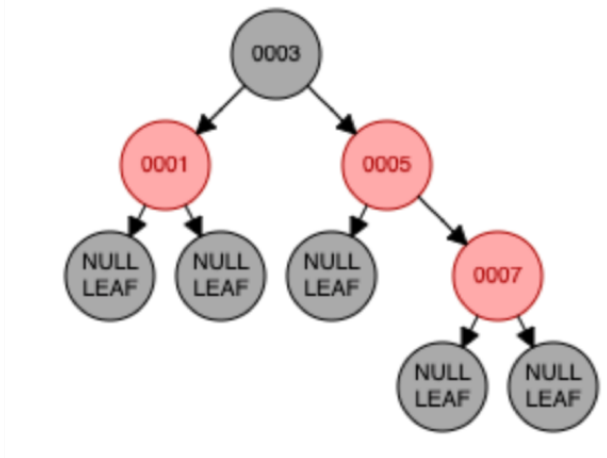


Figure 8: We add a red node with the value of 7. As is bigger than 3, it goes to the right, and it is also bigger than 5 so it goes to the right. Here we have a violation of the properties, since two red nodes can not be adjacent to one another.

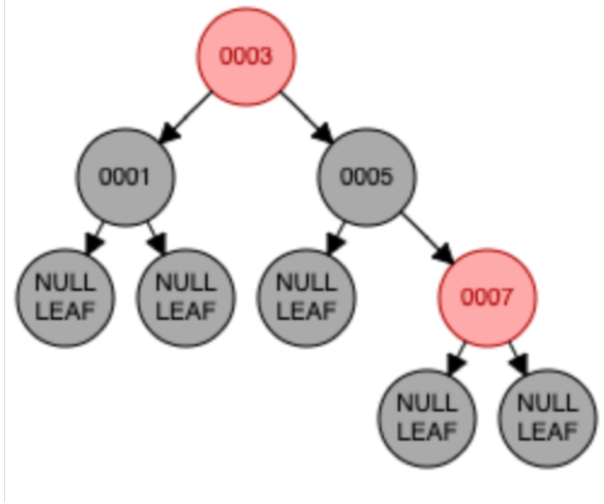


Figure 9: To fix the problem in tree 8, we change the color of the parent node 5, and the uncle node 1. With this result we violate another property, in which the root node has to be black.

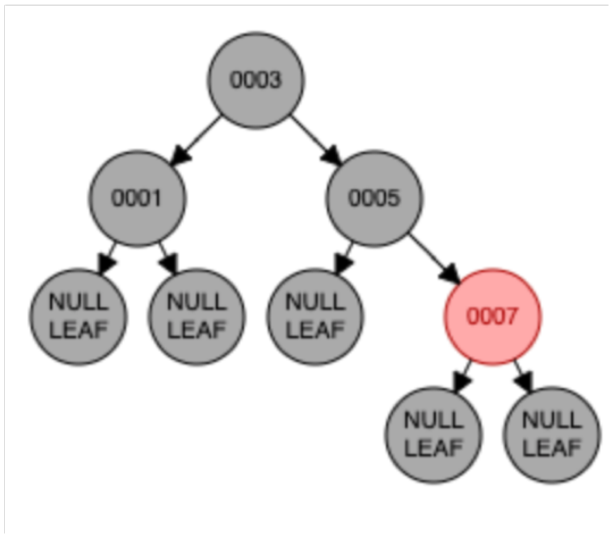


Figure 10: We change the color of the root node, and now we have a tree that meets all the properties.

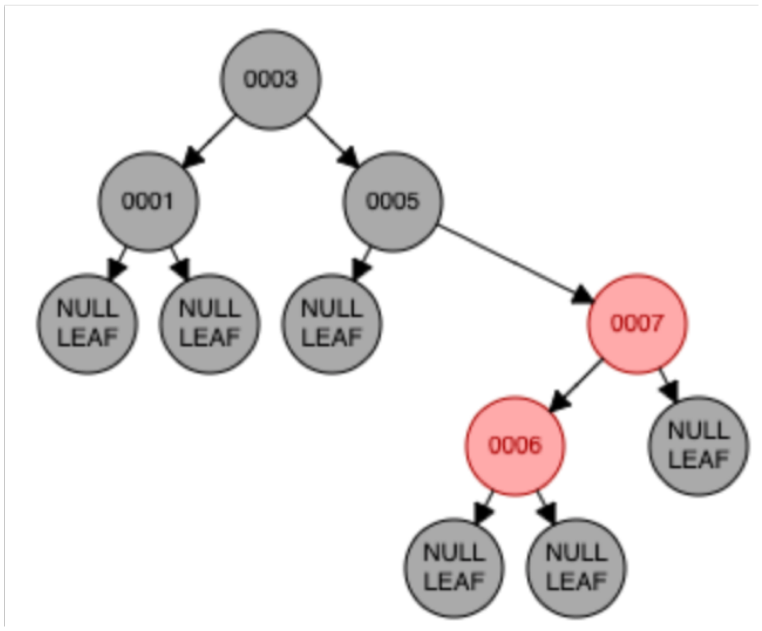


Figure 11: Now we add another red node with the value of 6. It is bigger than 3 and 5, so it moves to the right, and in node 7, it moves to the left. Here we violate the property that two red nodes can not be adjacent to one another. We can not simply change the color of the node, because we would be infringing on the depth property. So we need to make a right-left operation.

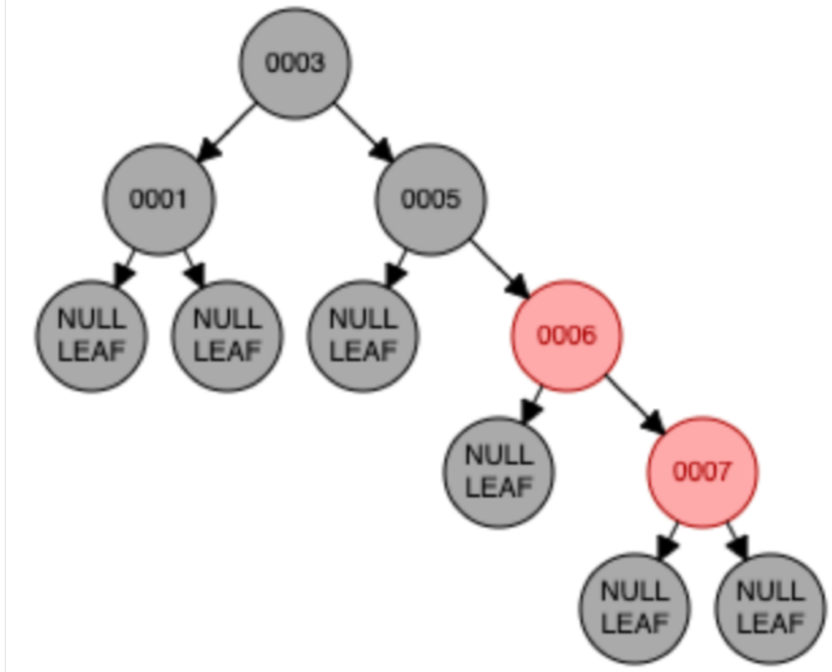


Figure 12: Now that we made the right rotation with node 7, we need to do the left rotation with the node 5

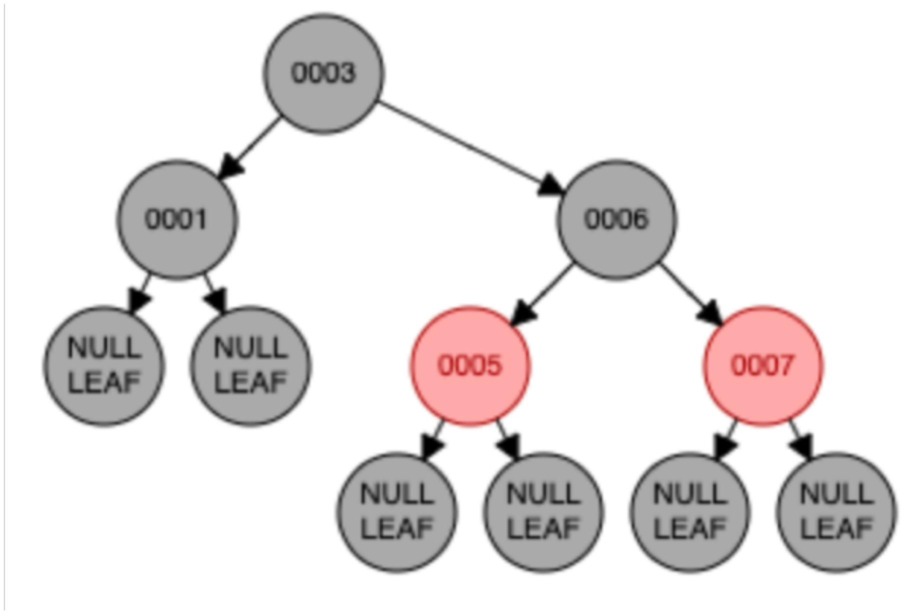


Figure 13: To finish this operation, we need to change the color of the node 6 to black and the node 5 to red. Now we have a tree that meets all the properties.

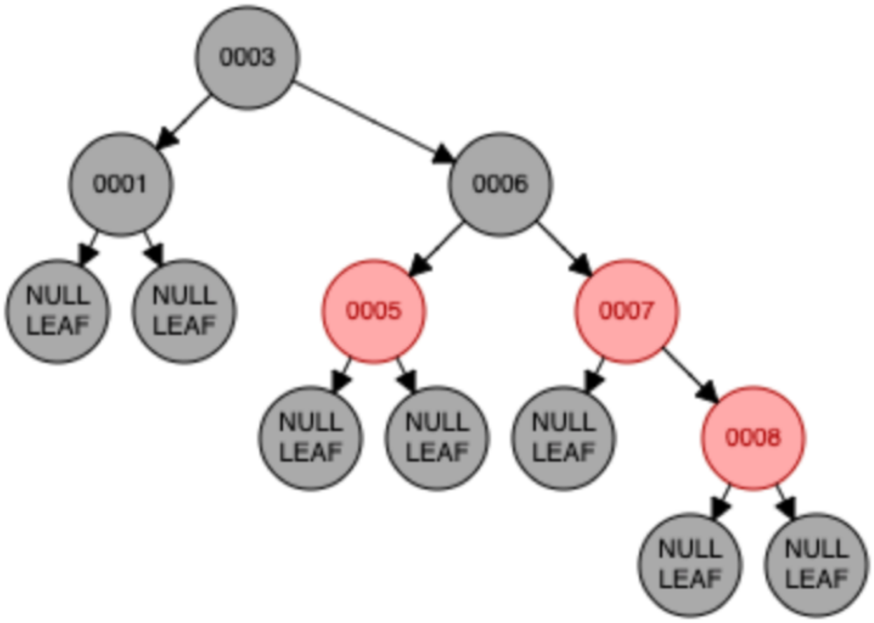


Figure 14: We insert a red node with the value 8. Since it is bigger than 3, 6 and 7, it moves to the right. This tree violates the property of two red nodes, so we need to change the color of the parent and the uncle of node 8.

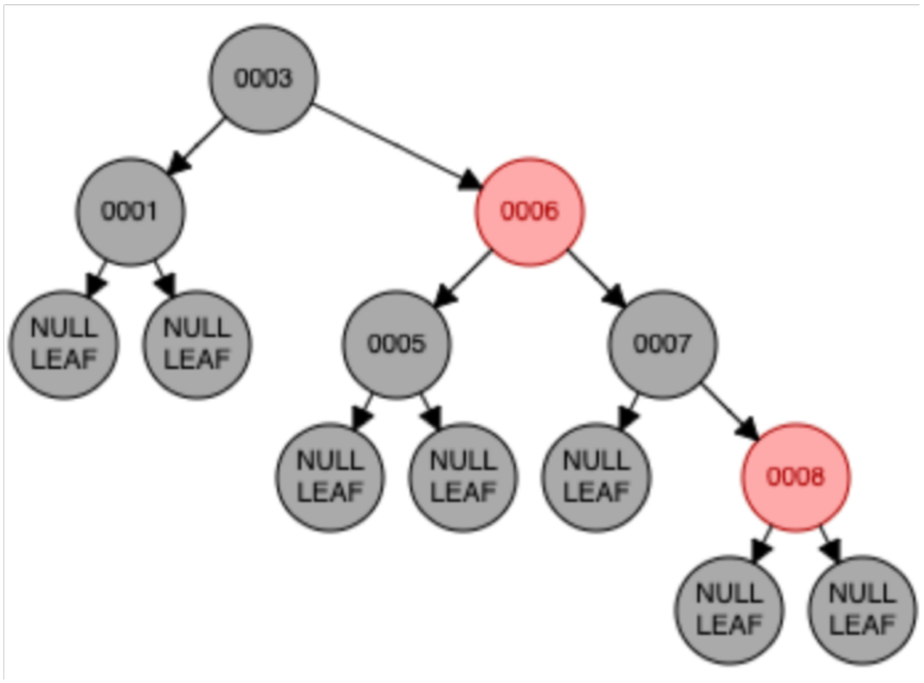


Figure 15: Now that we fixed the colors, all properties are meet.

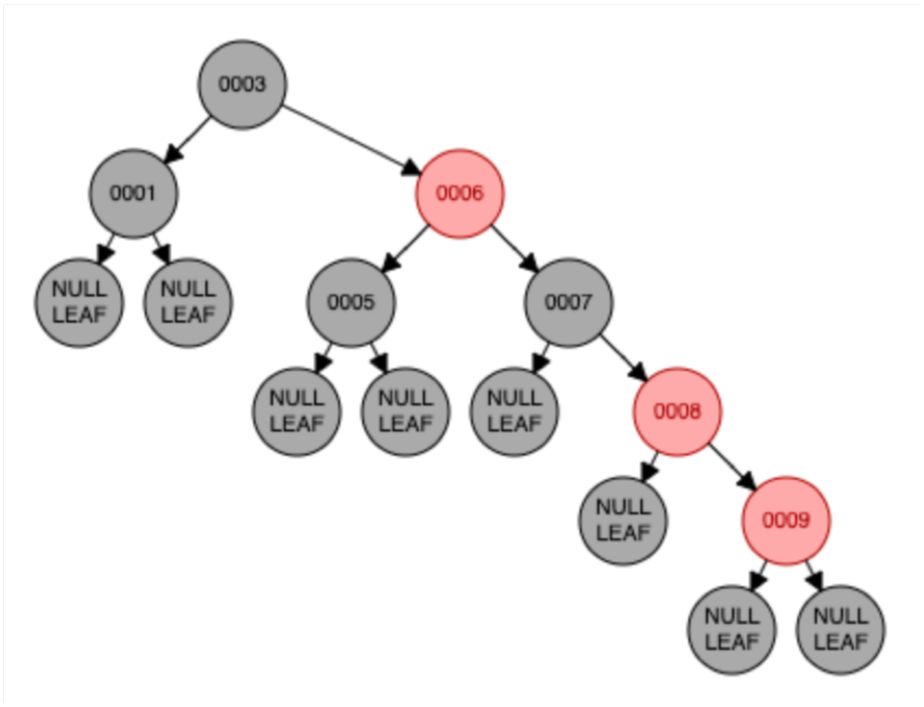


Figure 16: We add a red node with the value 9. It is bigger than 3, 6, 7, and 8, so it moves to the right. This tree violates the rule of two red nodes. Changing colors will still be infringing properties, so first, we make a left rotation.

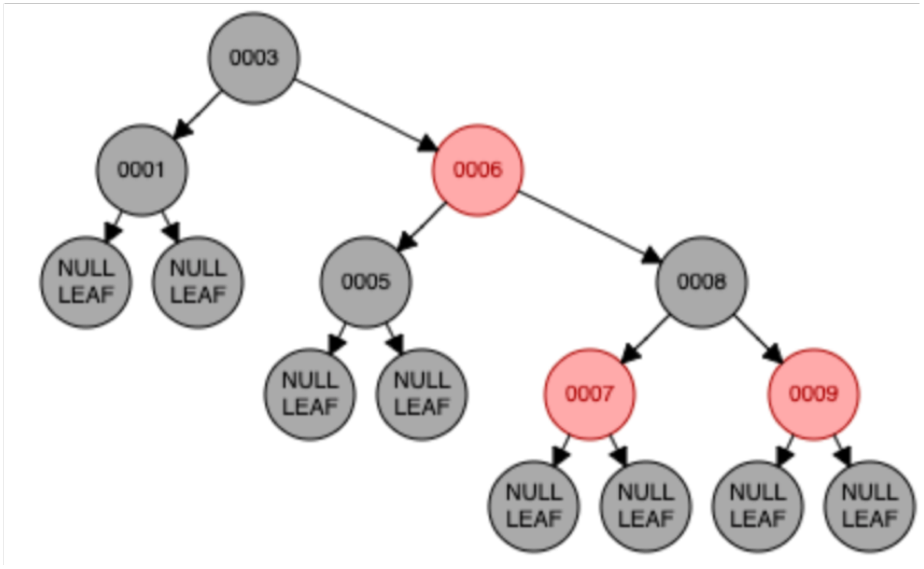


Figure 17: After the rotation, we can now change the color of node 8 to black and node 7 to red.

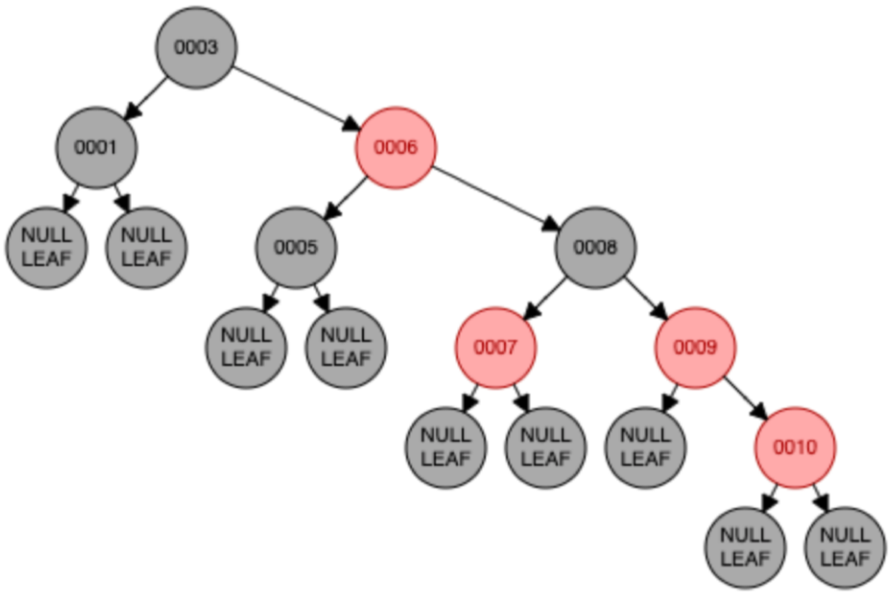


Figure 18: We now add a red node with the value 10. This tree violates the rule of two red nodes, so we change the color of the parent node 9 and the uncle node 7.

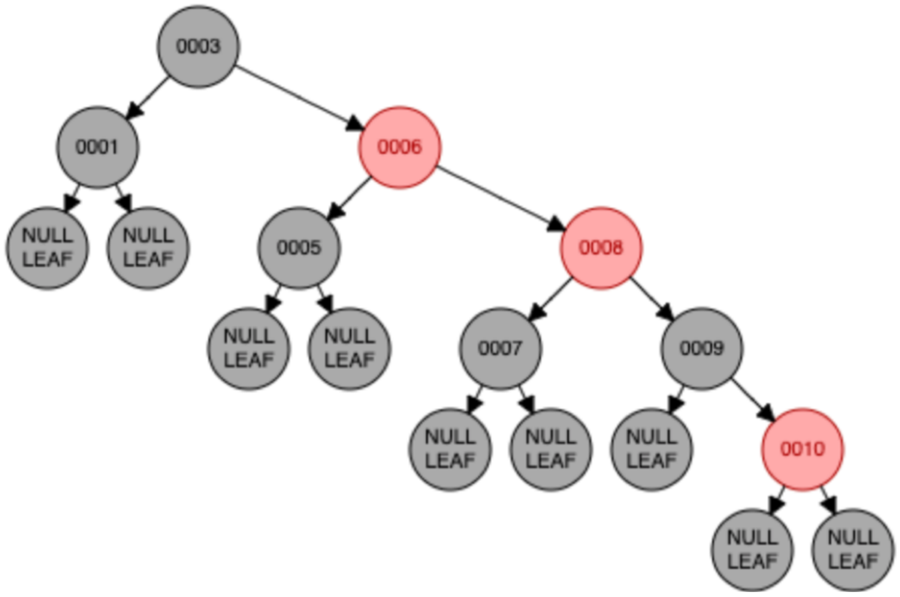


Figure 19: Now that we switched the colors, we are still violating the rule of two red nodes, and changing colors again will infringe in other properties, so we make a left rotation of node 6.

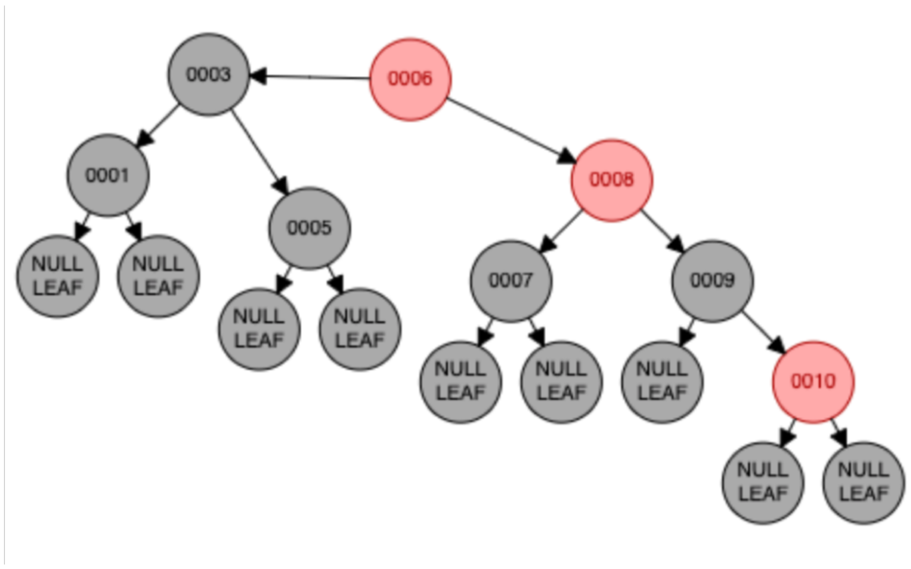


Figure 20: After the rotation, the node 6 is now violating the rule of the root node needing to be black.

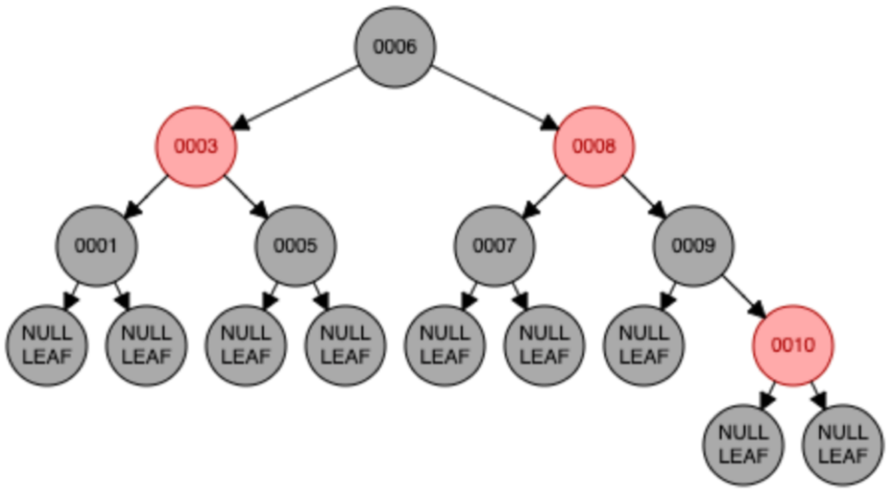


Figure 21: We change the color of the root node 6 to black, and the node 3 to red and we now have a tree that meets all the properties. Now, if we compare this last tree with the last result of the console in Figure 4 we see that they match, concluding this experiment.

4 Conclusions

I had a bit of trouble understanding the rules of this tree, mainly because I did not understand at first the use of the red and black nodes, but after learning the properties and the different operations to balance the tree, I understood the function of the red and black, and it was actually fun to play with the different options to move the nodes. It helped a lot to be able to visualize it on the web page mentioned in the example.

This is my first time working with trees, so it was really interesting to find out how they worked, and they seemed a better choice to structure your data than a queue or a stack, if the goal is to find an element more efficiently.

5 Reference

References

[1] Binary tree. `https://www.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/`

[2] Data structures and algorithms. `https://www.cs.auckland.ac.nz/software/AlgAnim/red_black_op.html`

[3] Red-Black tree. `https://www.programiz.com/dsa/red-black-tree`