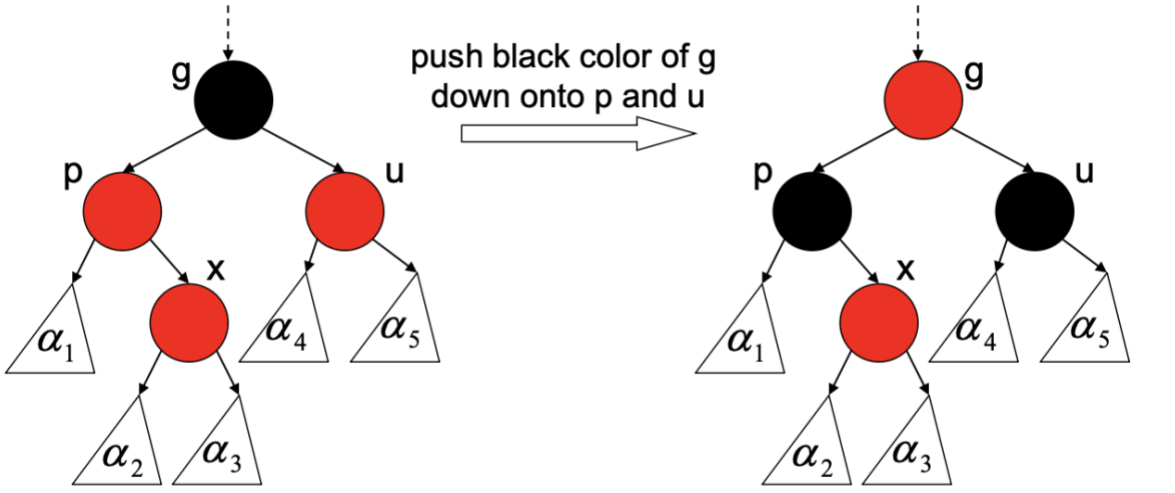
Keep black height

Insert

```
RBT_Insert (T, x) { // T: the tree, x: a new node to be inserted
  color[ x ] = RED;   // the new node will be red
  BST_Insert(T, x);  // insert as if we were using an ordinary BST
  FixColoring(T, x); // fix the "red-red" coloring problem if exists
}
```

There are three cases by which the insertion might violate the properties of an RBT.

- Case 1: The uncle (*these nodes are male*) of the child in the red-red pair is also red, and x is the right child of p.




x : the child in the red-red pair
p : the parent in the red-red pair
u : the uncle of x

- Note that, *bh*(g) increases by 1.
- *bh* of all the other nodes stay the same.
- Hence, RBT property 5 is preserved.

root of $\alpha_i$ must be black (there is only one red-red pair problem in the tree)

79

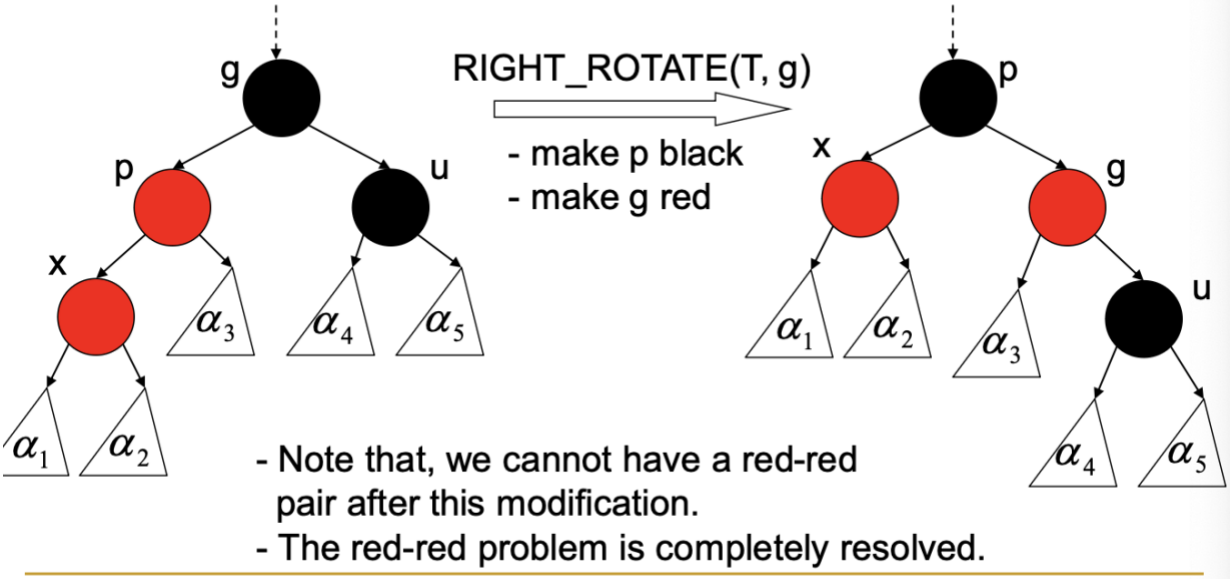
As explained in the slide, black height, that is the number of black nodes following the path from a node to a leaf, stays the

same since pushing the colors down does not affect bh's, expect of g's, which is incremented by 1 since its children became black.

After recoloring, however, the problem (red-red nodes) might continue to exist as g is now red, and it might have a red parent and uncle. In that case, we repeat the same steps until reaching the root. If the root becomes red because of recoloring, then it is colored to black at the end.

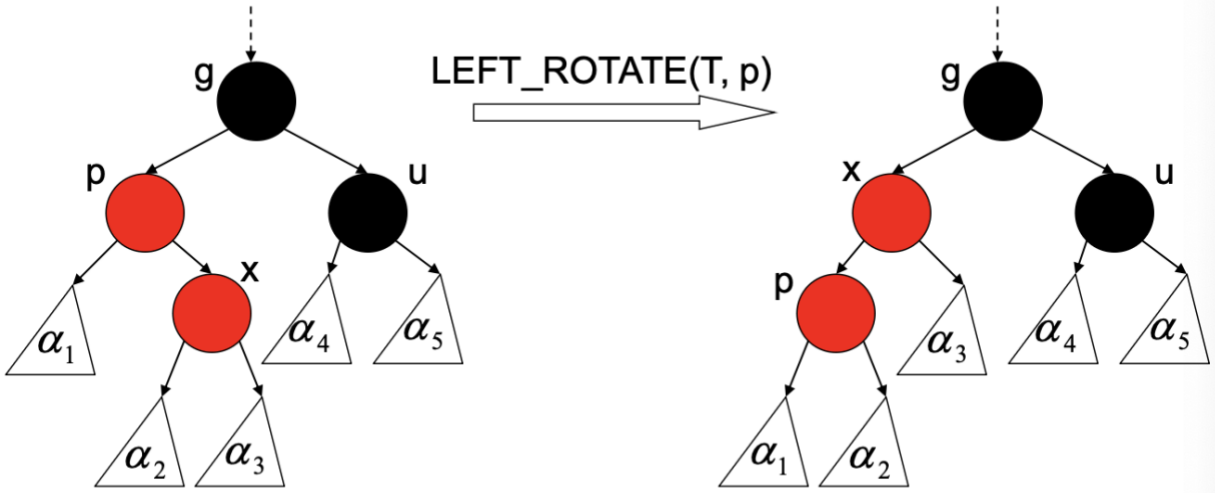
As explained above, we can observe that in the worst-case every node's black height above (and including) g will be incremented by 1. Fortunately, we can achieve this without increased time complexity by simply incrementing the grandparent's black height by one each time recoloring is applied. Thus, both grandparent's children and g itself preserve bh property.

- Case 3: x is left child of p, p is left child of g, u is black.




If case 3 exists after insertion, one can observe that black height of any node above g (before rotation) remains the unchanged. Examining the nodes that have moved, let's first check bh's of them. Assume that bh(x) = n; then bh(p) = n, bh(g) = n, bh(u) = n-1, where n is an arbitrary number. Considering these values, after the rotation we can see that bh(x) is the same since it has the same children. Also, bh(g) is the same since it has u and a_3 subtree as children, and u has the same children before the rotation as well, thus unchanged. Therefore, we can say there is no need to traverse the tree, or even change any values of bh.

- Case 2: x is right child of p, p is left child of g, uncle is black.




- It seems that we did not solve anything, we still have the red-red pair.
- Actually, we have transformed it into case 3, hence it will be immediately solved.

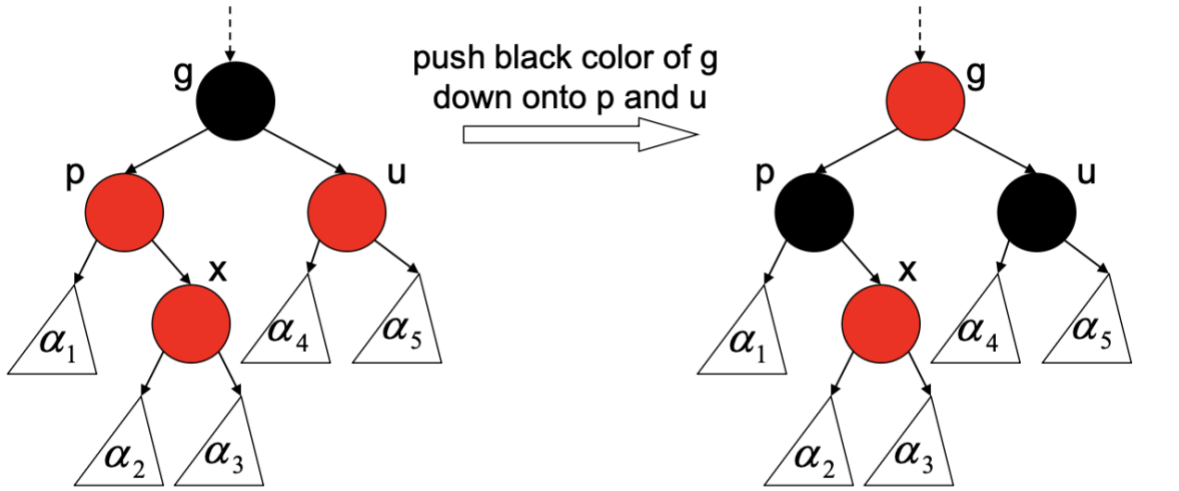
root of $\alpha_1$, $\alpha_2$, and $\alpha_3$ must be black (there is only one red-red pair problem in the tree)

In this case, same observations as above can be done. Assume that bh(x) = n; then bh(p) = n, bh(g) = n, bh(u) = n-1. After the rotation: bh(x) = n since p's new child a_2 was of x's and x had bh(x) = n, meaning that bh(p) is unchanged. Also, bh(g) and bh(u) remains the same since bh's of nodes underneath them is the same as explained. Thus, we don't need to change the bh value of any node in this rotation. After that the problem is turned into case 3 (previous case), in which there is no need to change bh values as explained previously.

In other cases, in which a red node is inserted as a child of a black node, there is no violation of RBT properties that can occur and trivially black height of any of nodes in the tree is not effected.

In all these cases there is no need to traverse the tree or a subtree to fix bh values. Only change is that when case 1 applies, there a constant time operation [bh(g) += 1], and this is repeated for new grandparent of g if case 1 shows up because of recoloring. Yet since grandparent is recolored anyway, so this operation does not add to the worst-case time complexity of FixColoring, and it is still O(h).

- Case 1: The uncle (*these nodes are male*) of the child in the red-red pair is also red, and x is the right child of p.




x : the child in the red-red pair
p : the parent in the red-red pair
u : the uncle of x

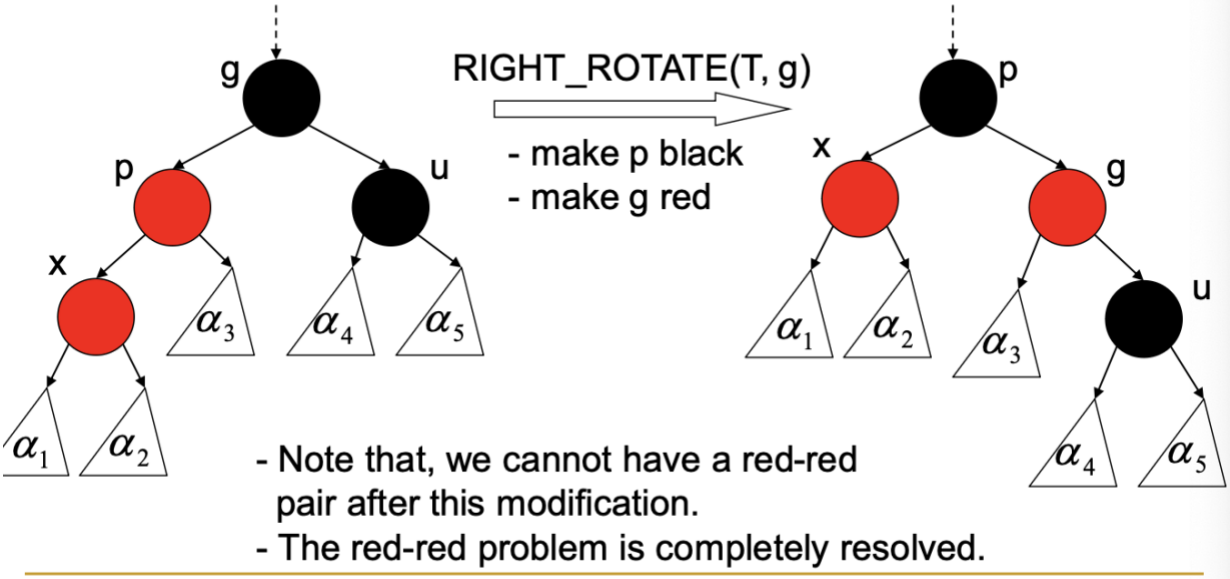
- Note that, *bh*(g) increases by 1.
- *bh* of all the other nodes stay the same.
- Hence, RBT property 5 is preserved.

root of $\alpha_i$ must be black (there is only one red-red pair problem in the tree)



Considering case 1, since there is no rotation is done, nodes' depth is not changed, thus there is no need to traverse the tree to fix the depth values.

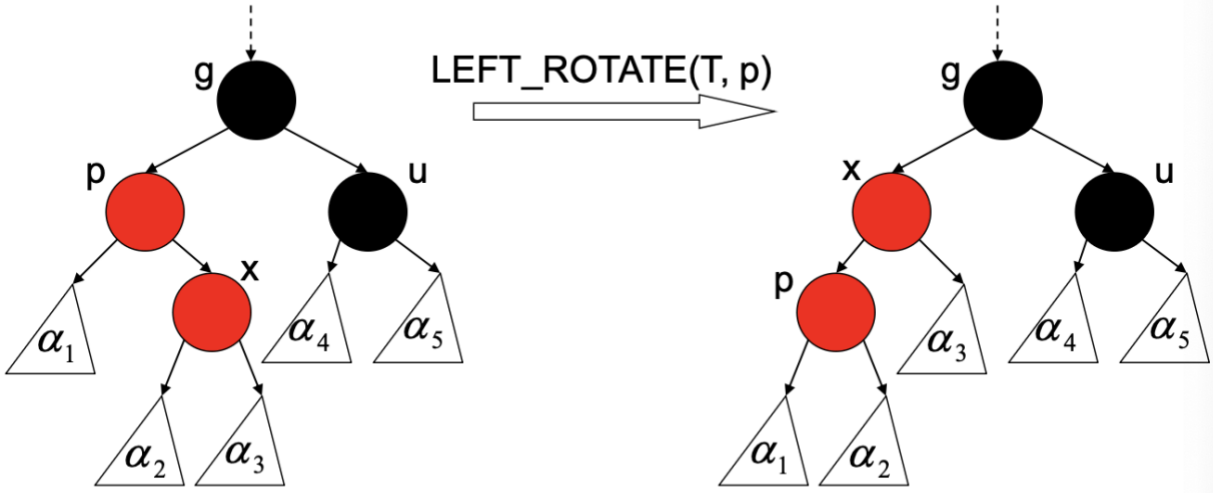
- Case 3: x is left child of p, p is left child of g, u is black.




In case 3, because of rotation some nodes' depth value is changed. Assume that depth(x) = d; then depth(p) = d – 1, depth(g) = d – 2, depth(u) = d – 1. After rotation, depth(x) = d – 1, depth(p) = d – 2, depth(g) = d – 1, depth(u) = d. Moreover, depth of nodes in a_1 and a_2 is decreased by 1, nodes in a_4 and a_5 is increased by 1. Only nodes in a_3 subtree keep their depth values. Because we need to update the values of each node within subtrees (except a_3), and also we know that the tree is balanced, meaning that there cannot be considerable more amount of nodes in a_3 than of other subtrees, we can say that there needs to be O(k) amount of computation, that is we need to traverse and fix depth values of each node within these subtrees (and x, p, g, u) where k is the amount of nodes below g (in the original state). Therefore, at worst-case this situation would occur when g is the root of the tree, and the number of nodes

that is needed to be traversed and fixed would-be O(n) where n is the number of nodes in the tree.

- Case 2: x is right child of p, p is left child of g, uncle is black.




- It seems that we did not solve anything, we still have the red-red pair.
- Actually, we have transformed it into case 3, hence it will be immediately solved.

root of $\alpha_1$, $\alpha_2$, and $\alpha_3$ must be black (there is only one red-red pair problem in the tree)

In case 2, after the rotation depth values changed are of node p, x and subtrees a_1, a_3. Furthermore, in this state problem is turned into case 3, in which O(n) computation for traversing and fixing nodes' depth values is required. Also, since a_3 is also changed in this case (in only case 3 it was unchanged), simply all the nodes below and including g is required to be fixed. Therefore, at worst-case scenario g would be the root meaning that all the nodes in the tree is required to be fixed. Thus, at worst-case fixing adds O(n) time complexity to insertion.