# DESIGN AND ANALYSIS OF ALGORITHMS – CSCI 612

**NAME:** AADITHYA GOPALAKRISHNA BHARADWAJ

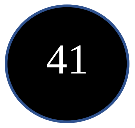
**ZID:** Z1862641

**ASSIGNEMENT NUMBER, SEMESTER:** Assignment 8, SPRING 2019

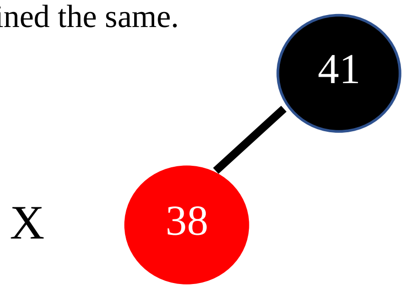
======================================================================

**1. Show the red-black trees that result after successively inserting each of the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree. For each key, show each tree after inserting each key.**

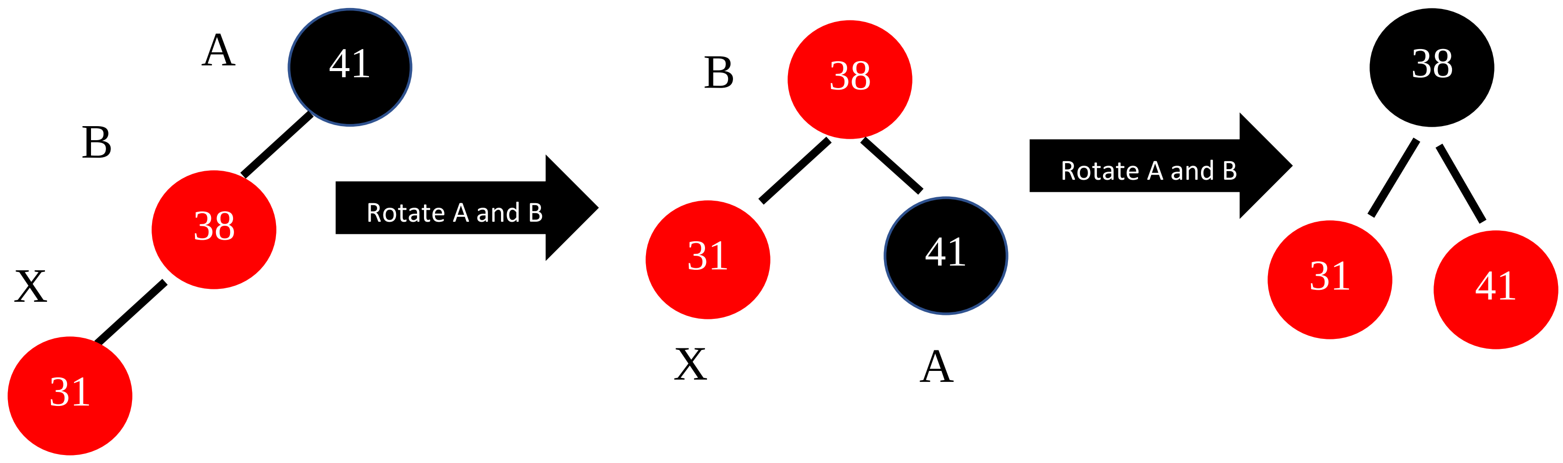
**Answer.)** **Step 1:** Insert 41



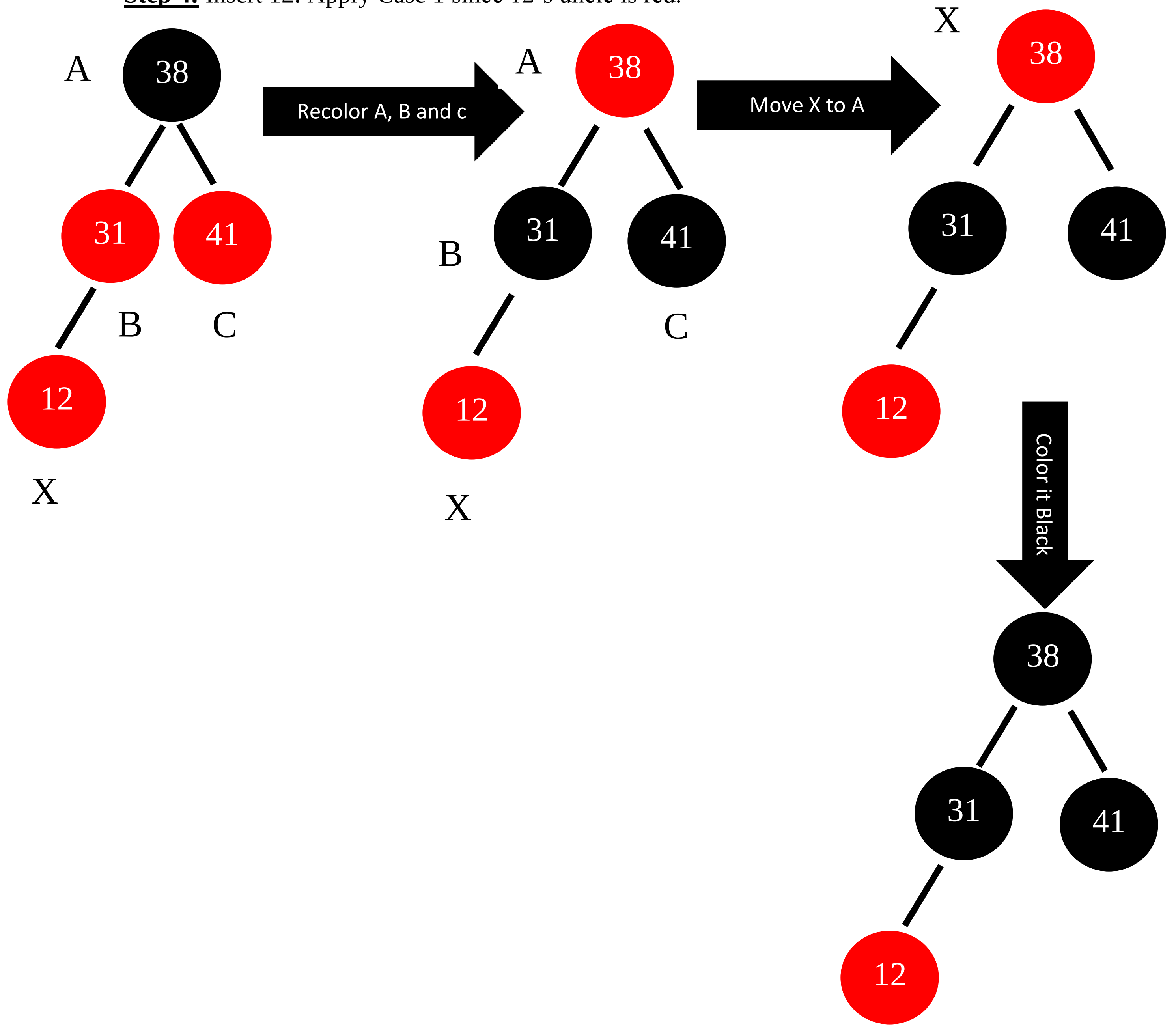
**Step 2:** Insert 38 - Insert a red node. No cases have to be applied because the number of black nodes from the root to all nulls has remained the same.



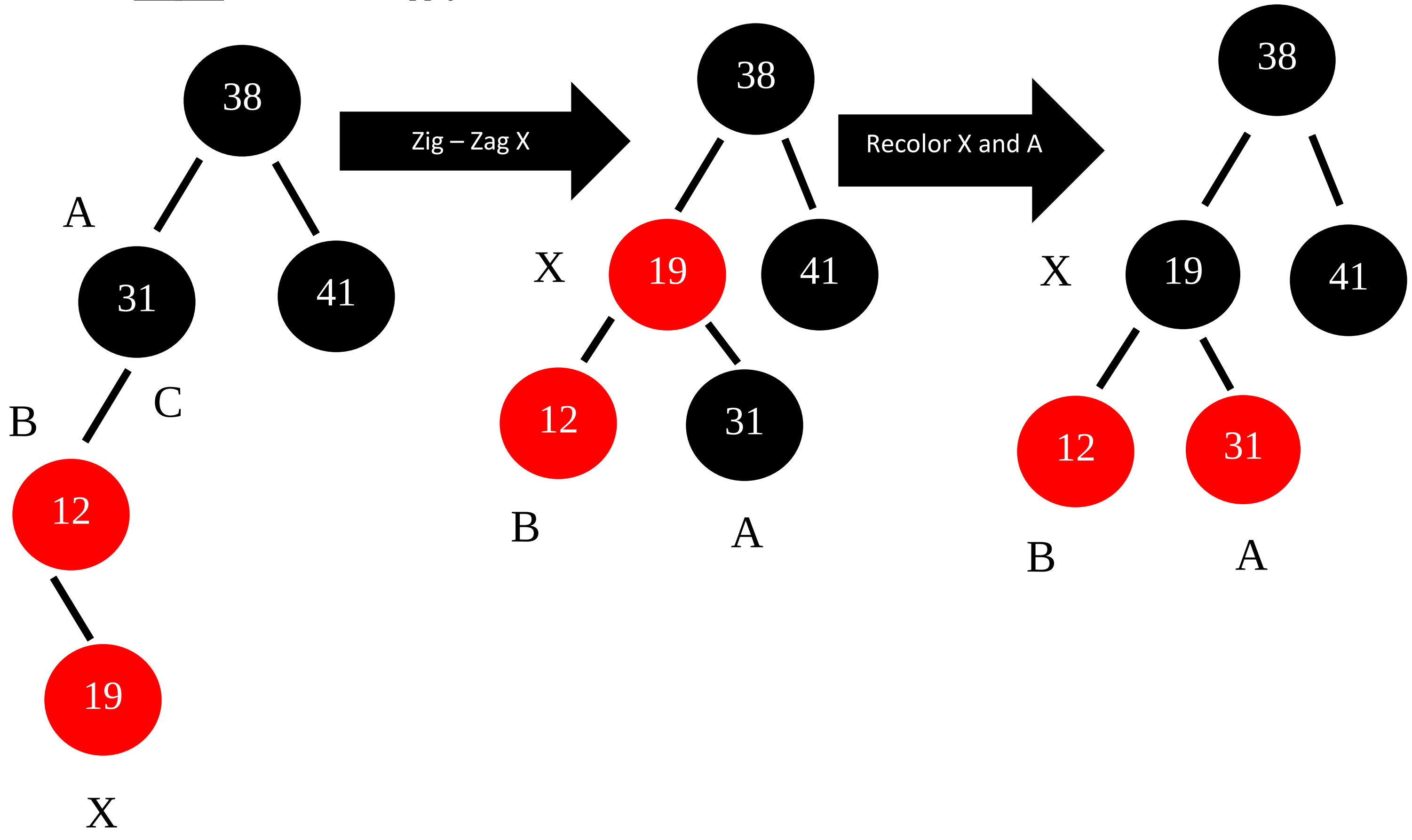
**Step 3:** Insert 31 and apply Case 3 since 31's uncle is Black.



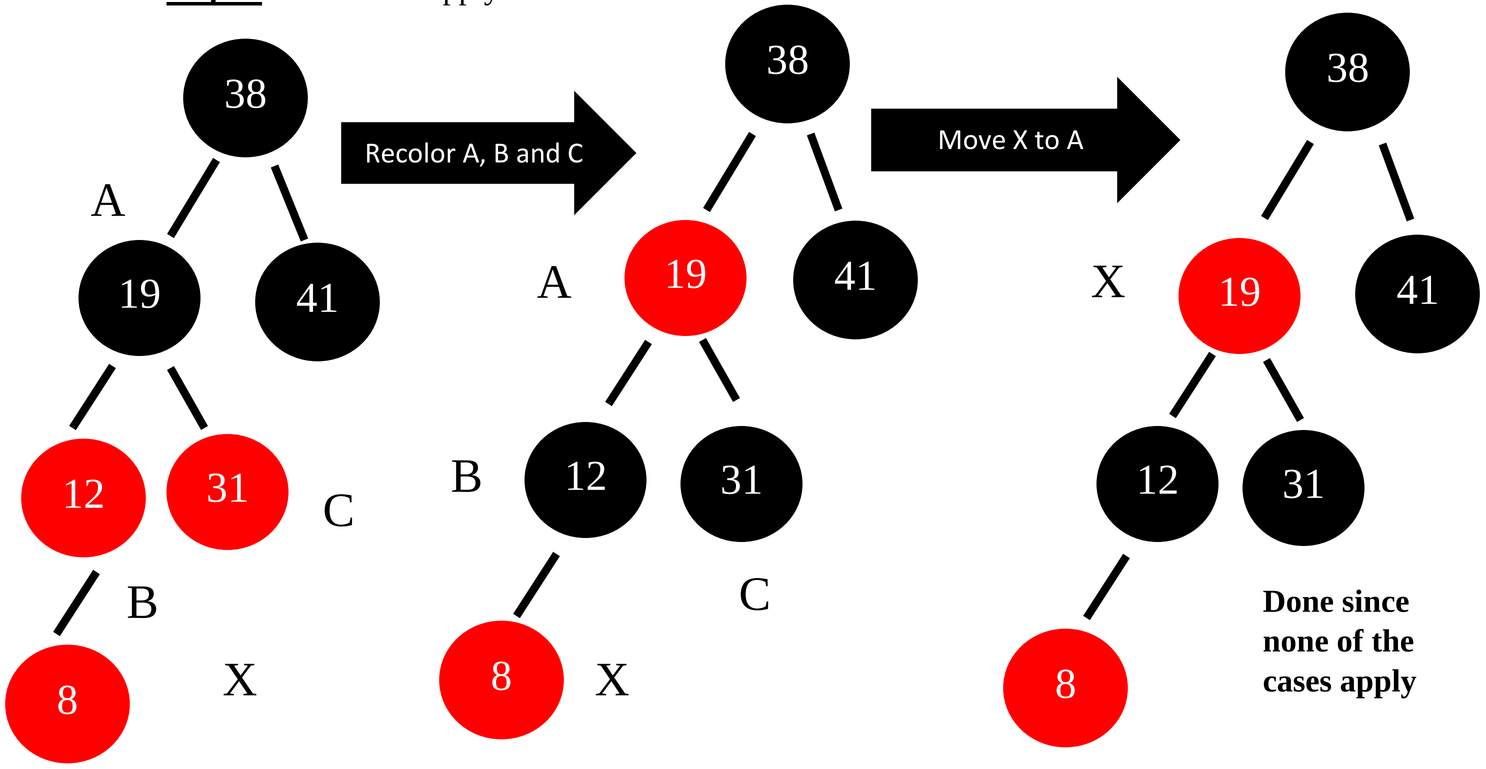
**Step 4:** Insert 12: Apply Case 1 since 12's uncle is red.



**Step 5:** Insert 19 and apply case 2



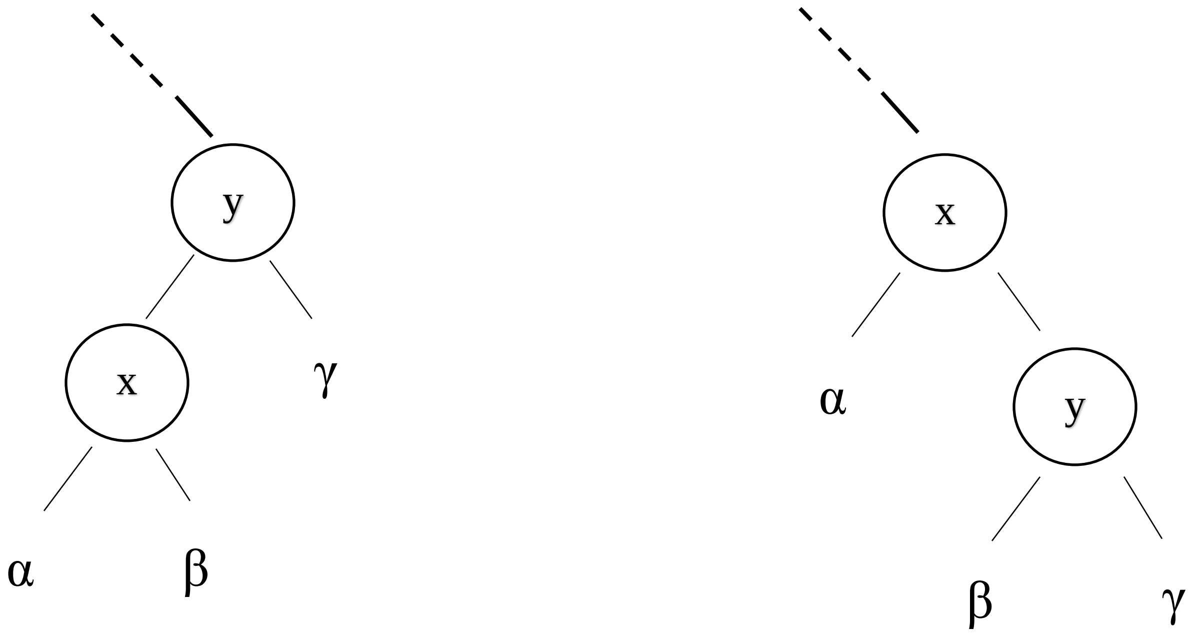
**Step 6:** Insert 8 and apply Case 1



**Done since none of the cases apply**

**2.) Argue that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using O(n) rotations. Hint: First show that at most n-1 right rotations suffice to transform the tree into a right-going chain.)**

**Answer.)** Let us assume that leaves are full-fledged nodes, and we ignore the sentinels. we start by showing that with at most ***n −1* right rotations**, we can convert any binary search tree into one that is just a right-going chain. The idea is simple. Let us define the ***right spine*** as the root and all descendants of the root that are reachable by following only right pointers from the root. A binary search tree that is just a right-going chain has all n nodes in the right spine. As long as the tree is not just a right spine, repeatedly find some node ***y*** on the right spine that has a non-leaf left child ***x*** and then perform a right rotation on ***y***:



(In the above figure, note that any of α, β, and γ can be an empty subtree.) Observe that this right rotation adds xx to the right spine, and no other nodes leave the right spine. Thus, this right rotation increases the number of nodes in the right spine by 11. Any binary search tree starts out with at least one node the root in the right spine. Moreover, if there are any nodes not on the right spine, then at least one such node has a parent on the right spine. Thus, at most *n−1* right rotations are needed to put all nodes in the right spine, so that the tree consists of a single right-going chain.

If we knew the sequence of right rotations that transforms an arbitrary binary search tree *T* to a single right-going chain $T^1$, then we could perform this sequence in reverse turning each right rotation into its inverse left rotation to transform $T^1$ back into *T*.

Therefore, here is how we can transform any binary search tree $T_1$ into any other binary search tree $T_2$. Let $T^1$ be the unique right-going chain consisting of the nodes of $T_1$ (which is the same as the nodes of $T_2$). Let r = < $r_1, r_2, \ldots, r_k$ > be a sequence of right rotations that transforms $T_1$ to $T^1$ and let $r^1$ = < $r_1', r_2', \ldots, r_k'$⟩ be a sequence of right rotations that transforms $T_2$ to $T^1$. We know that there exist sequences $r$ and $r'$ with $k' \leq n - 1$. For each right rotation $ri'$, let $li'$ be the corresponding inverse left rotation. Then the sequence $< r_1, r_2, \ldots, r_k, l_k', l'_{k'-1}, \ldots, l_2', l_1' >$ transforms $T_1$ to $T_2$ in at most $2n - 2$ rotations.

==============================================================

**3.) Can the black-heights of nodes in a red-black tree be maintained as fields in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not.**

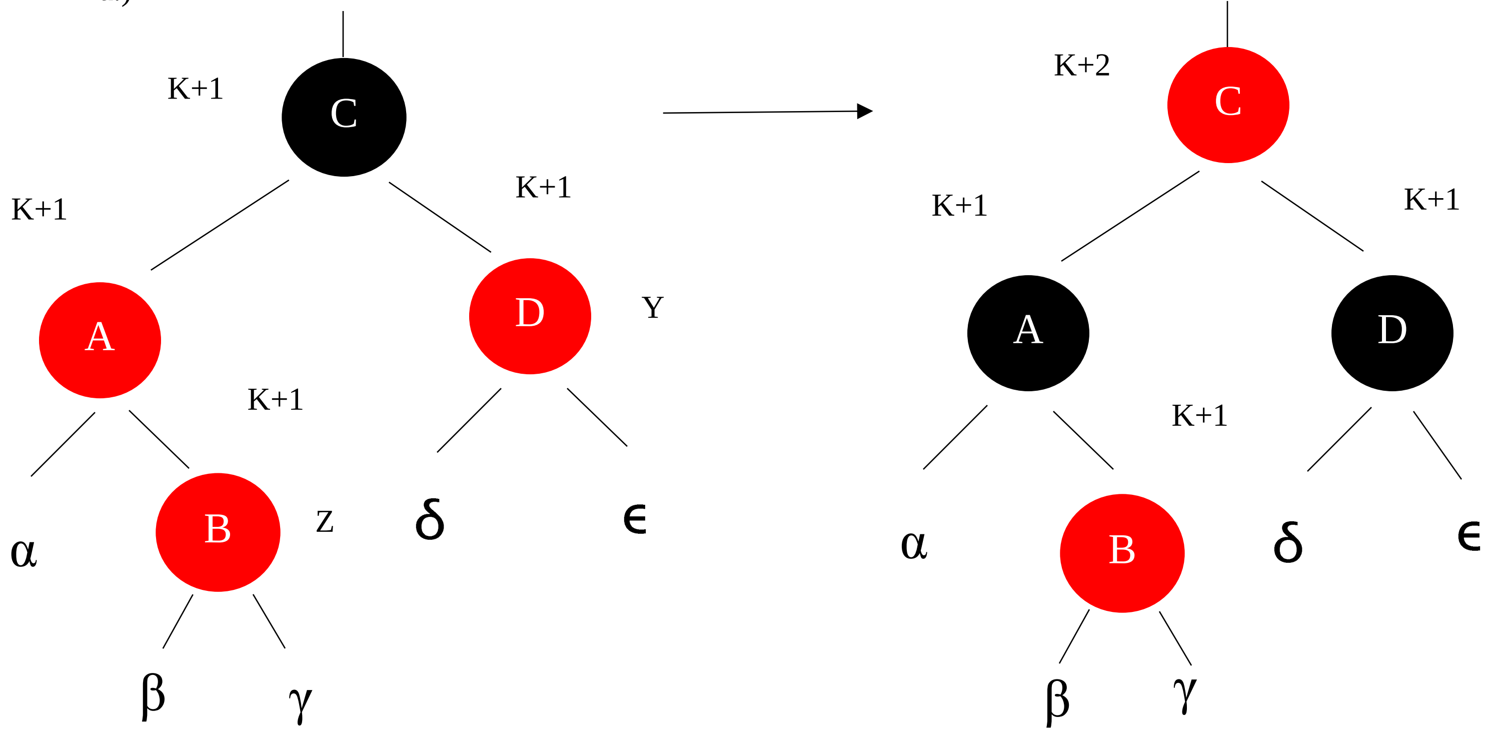
**Answer.)** Yes, we can maintain black-heights as attributes in the nodes of a red-black tree without affecting the asymptotic performance of the red-black tree operations. Because the black-height of a node can be computed from the information at the node and its two children. Actually, the black-height can be computed from just one child's information: the black-height of a node is the black-height of a red child, or the black height of a black child plus one. The second child does not need to be checked because of property 5 of red-black trees.

Within the RB-INSERT-FIXUP and RB-DELETE-FIXUP procedures are color changes, each of which potentially cause $O(\lg n)$ black-height changes. Let us show that the color changes of the fixup procedures cause only local black-height changes and thus are constant-time operations. Assume that the black-height of each node **<u>x</u>** is kept in the attribute ***<u>x.bh</u>***.

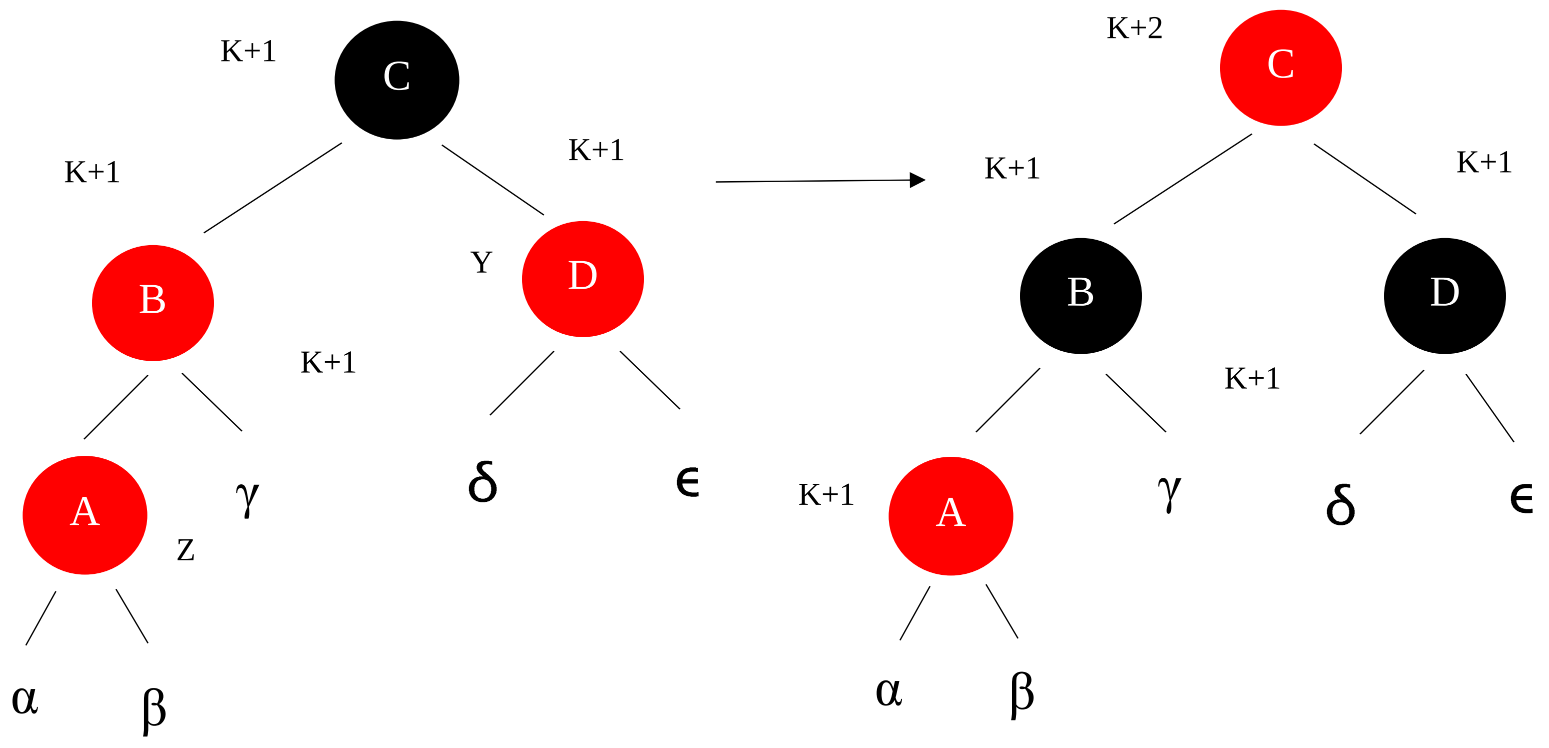
For RB-INSERT-FIXUP, there are 3 cases to examine.

**<u>Case 1:</u>** z’s uncle is red

a.)



b.)



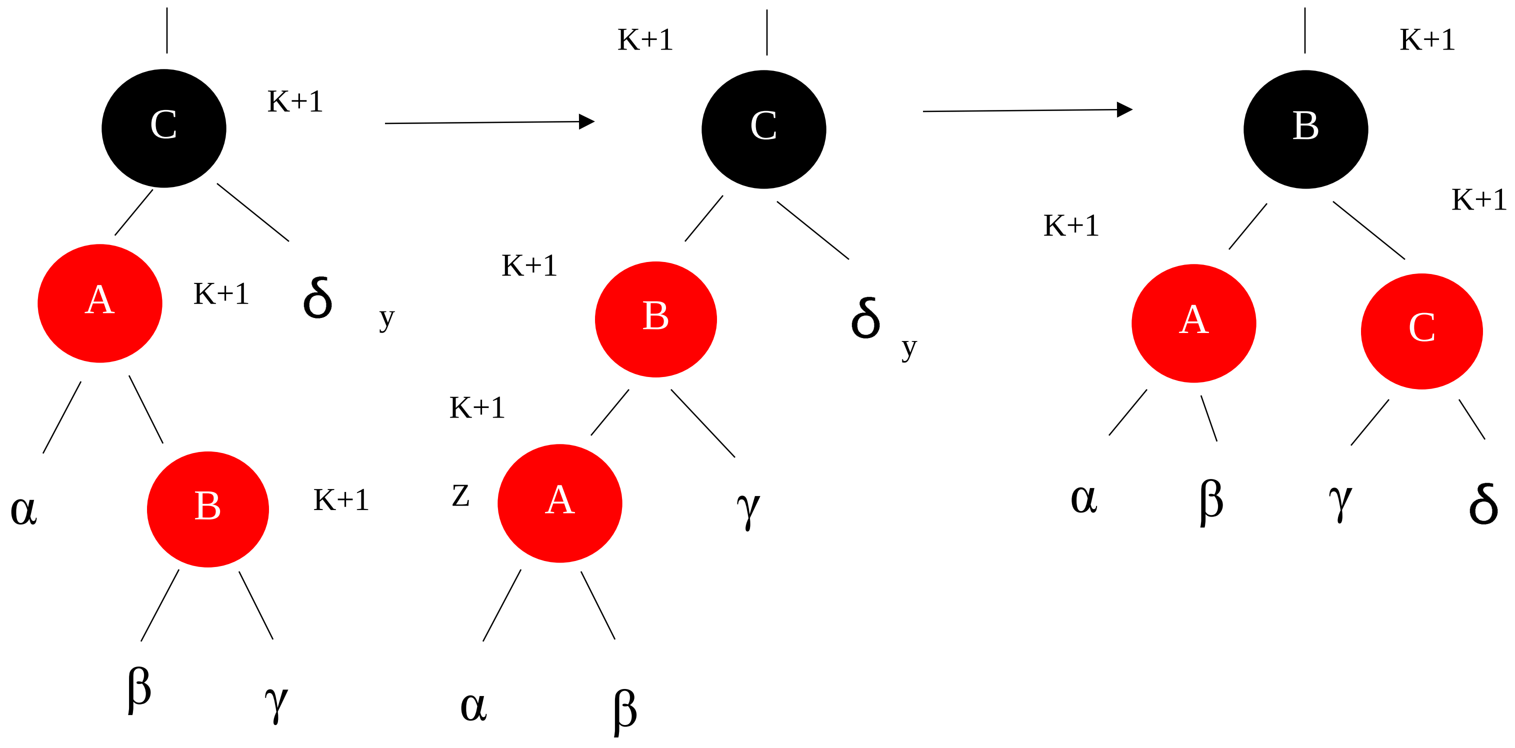
- Before color changes, suppose that all subtrees $\alpha$, $\beta$, $\gamma$, $\delta$, and $\epsilon$ have the same black-height $k$ with a black root, so that nodes $A$, $B$, $C$, and $D$ have blackheights of $k+1$.

- After color changes, the only node whose black-height changed is node $C$. To fix that, add $z.p.p.bh = z.p.p.bh+1$ after line 7 in RB-INSERT-FIXUP.

- Since the number of black nodes between $z.p.p$ and $z$ remains the same, nodes above $z.p.p$ are not affected by the color change.

**Case 2:** z's uncle y is black, and z is a right child.

**Case 3:** z's uncle y is black, and z is a left child.

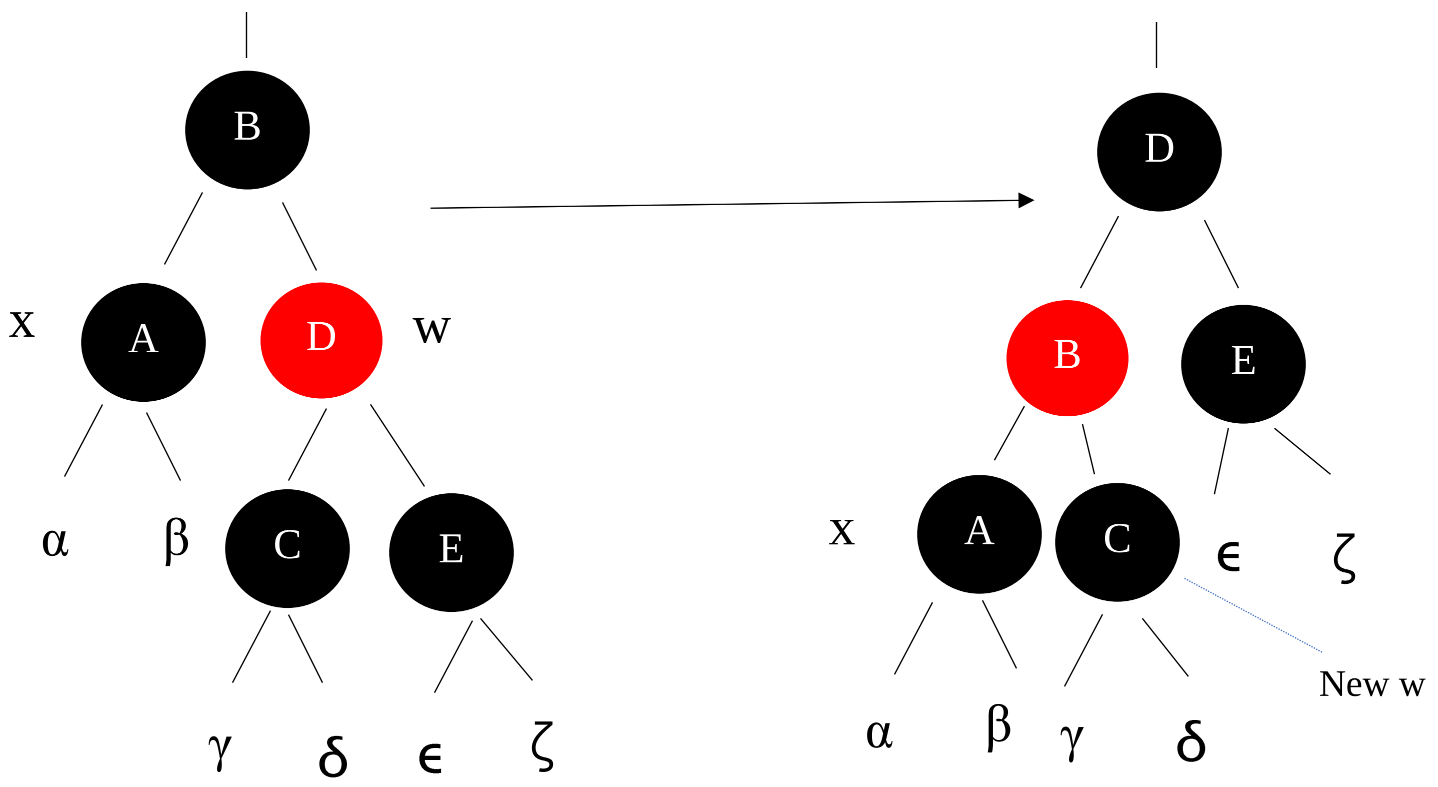


- With subtrees *α*, *β*, *γ*, *δ*, *ϵ* of black-height *k*, we see that even with color changes and rotations, the black-heights of nodes *A*, *B*, and *C* remain the same (*k*+1).

Thus, RB-INSERT-FIXUP maintains its original *O(lg n)* time.

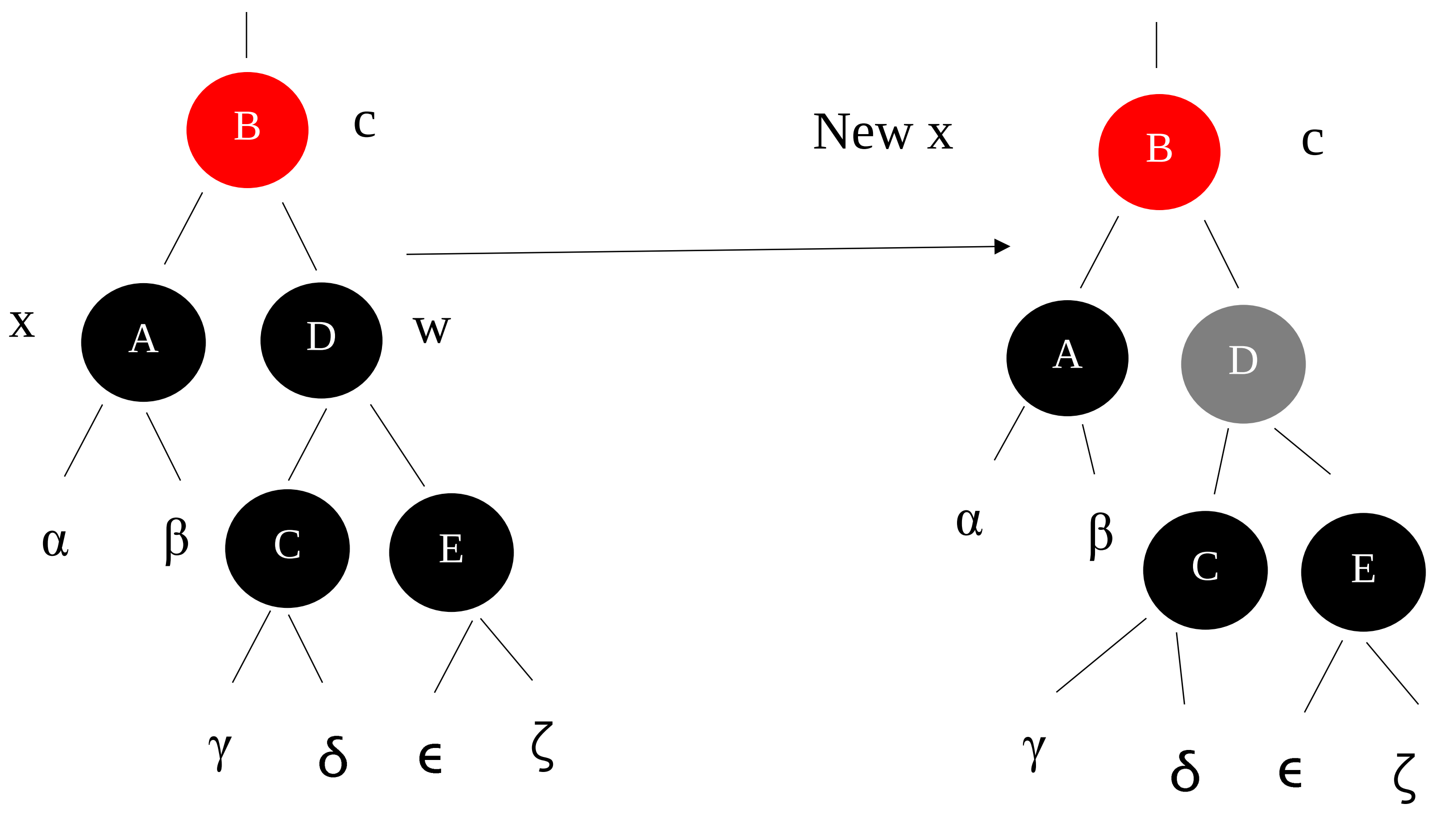
For RB-DELETE-FIXUP, there are 4 cases to examine.

**Case 1:** x's sibling w is red



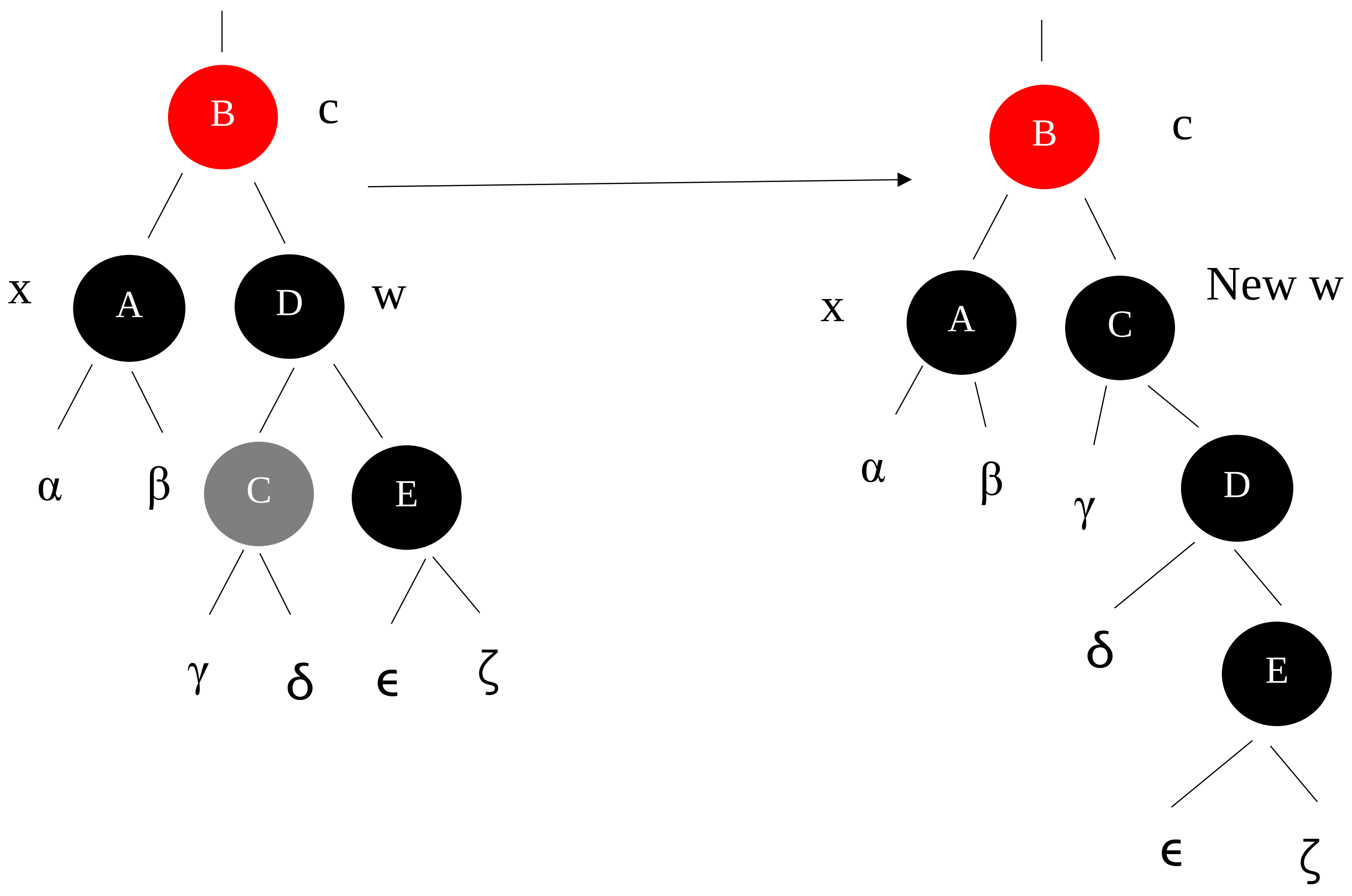
- Even though case 1 changes colors of nodes and does a rotation, blackheights are not changed.
- Case 1 changes the structure of the tree, but waits for cases 2, 3, and 4 to deal with the "extra black" on *x*.

**Case 2:** x’s sibling w is black, and both of w’s children are black.



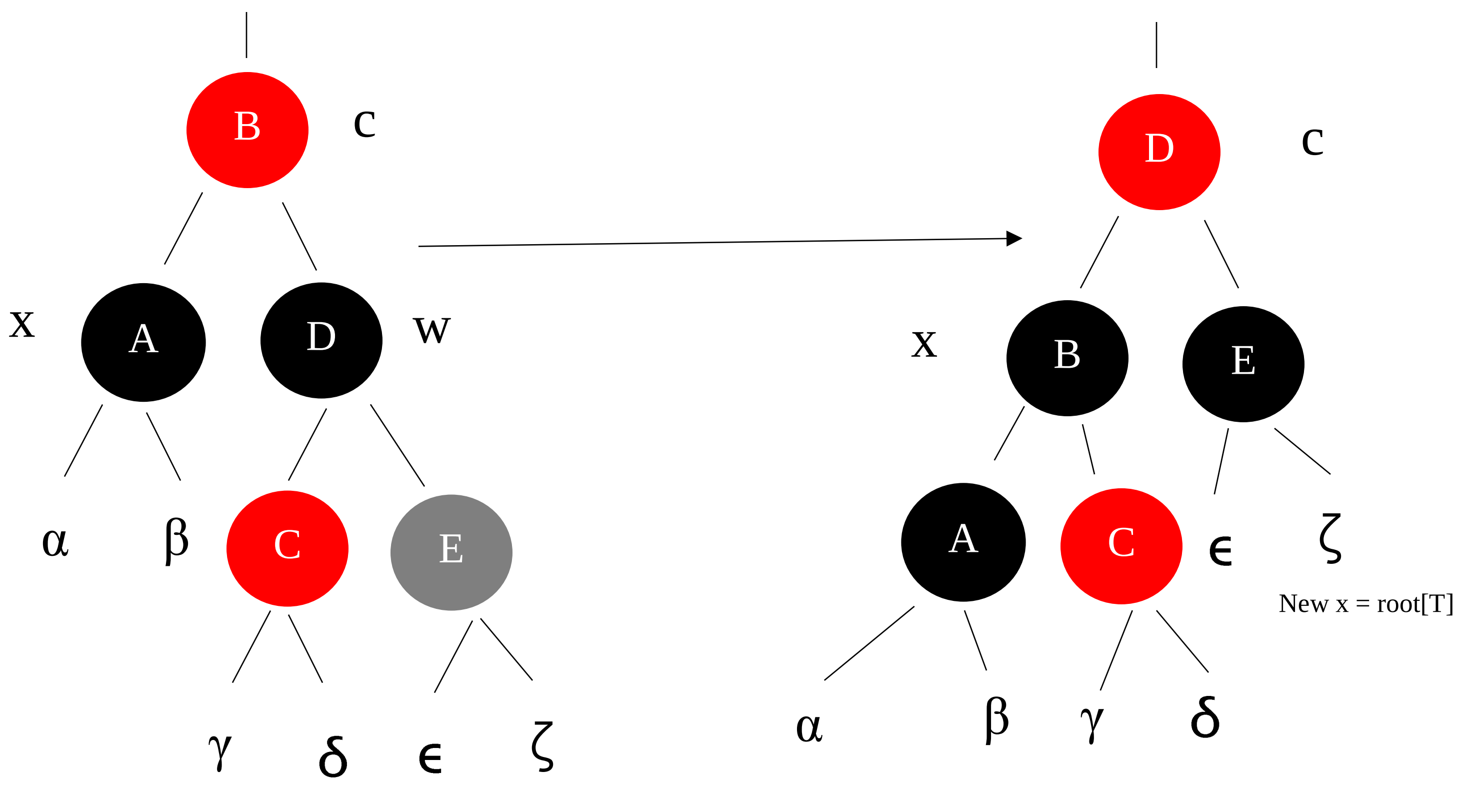
- *w* is colored red, and *x*'s "extra" black is moved up to *x.p*.
- Now we can add *x.p.bh* = *x.bh* after line 10 in RB-DELETE-FIXUP.
- This is a constant-time update. Then, keep looping to deal with the extra black on *x.p*.

**Case 3:** *x*'s sibling w is black, *w*'s left child is red, and *w*'s right child is black.



- Regardless of the color changes and rotation of this case, the black-heights don't change.
- Case 3 just sets up the structure of the tree, so it can fall correctly into case 4.

**Case 4:** *x*'s sibling *w* is black, and *w*'s right child is red.



- Nodes *A*, *C*, and *E* keep the same subtrees, so their black-heights don't change.
- Add these two constant-time assignments in RB-DELETE-FIXUP after line 20:

$$x.p.bh = x.bh + 1$$
$$x.p.p.bh = x.bh + 1$$

- The extra black is taken care of. Loop terminates.

Thus, RB-DELETE-FIXUP maintains its original $O(\lg n)$ time.

Therefore, we conclude that black-heights of nodes can be maintained as attributes in red-black trees without affecting the asymptotic performance of red-black tree operations.

For the second part of the question, no, we cannot maintain node depths without affecting the asymptotic performance of red-black tree operations. The depth of a node depends on the depth of its parent. When the depth of a node changes, the depths of all nodes below it in the tree must be updated. Updating the root node causes $n-1$ other nodes to be updated, which would mean that operations on the tree that change node depths might not run in $O(n\lg n)$ time.