Lecture 21: Red-Black Trees III

2023/10/11

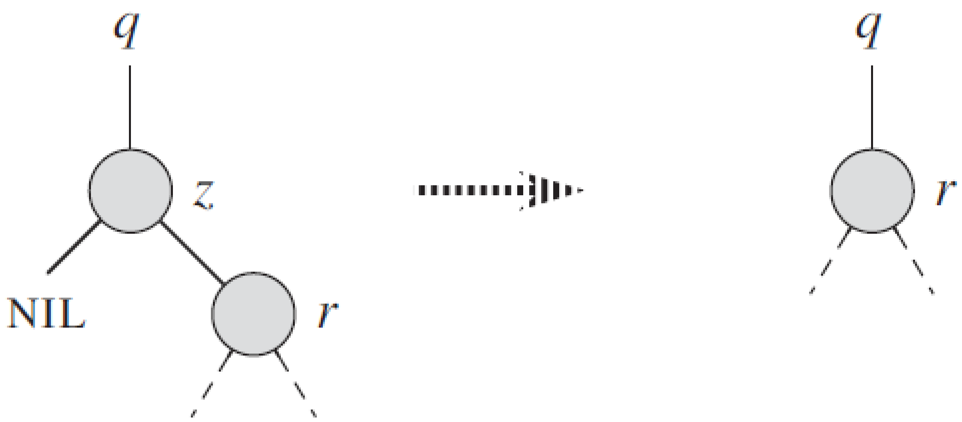
詹博华（中国科学院软件研究所）

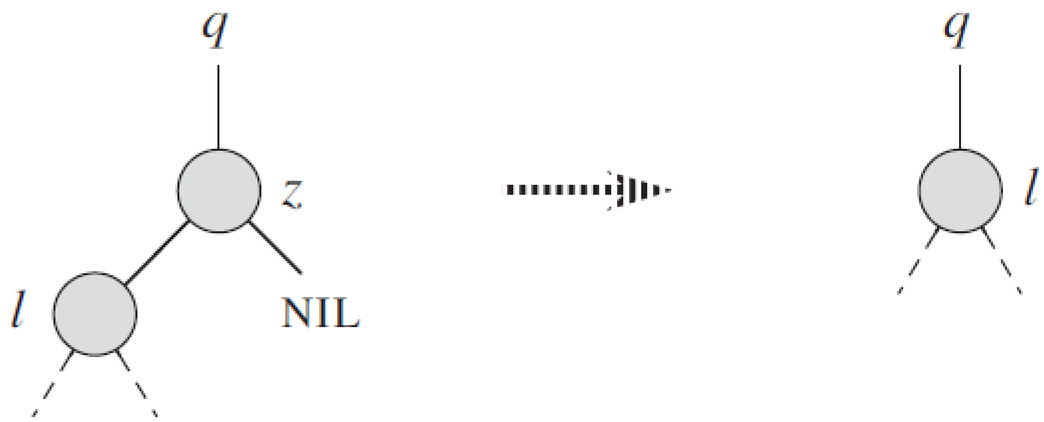
Deletion on Binary Search Trees

- Deletion is more complex than insertion, consisting of several cases.
- Suppose we wish to delete node z.
  - Case 1: z has no left child – replace z by its right child.
  - Case 2: z has no right child – replace z by its left child.
  - Case 3: z has both left and right child
    - Let y be z's successor (smallest element larger than z, or the minimum element in the right subtree of z).
    - Remove y from the tree (replace by its right child), then replace z by y.

Deletion: first two cases

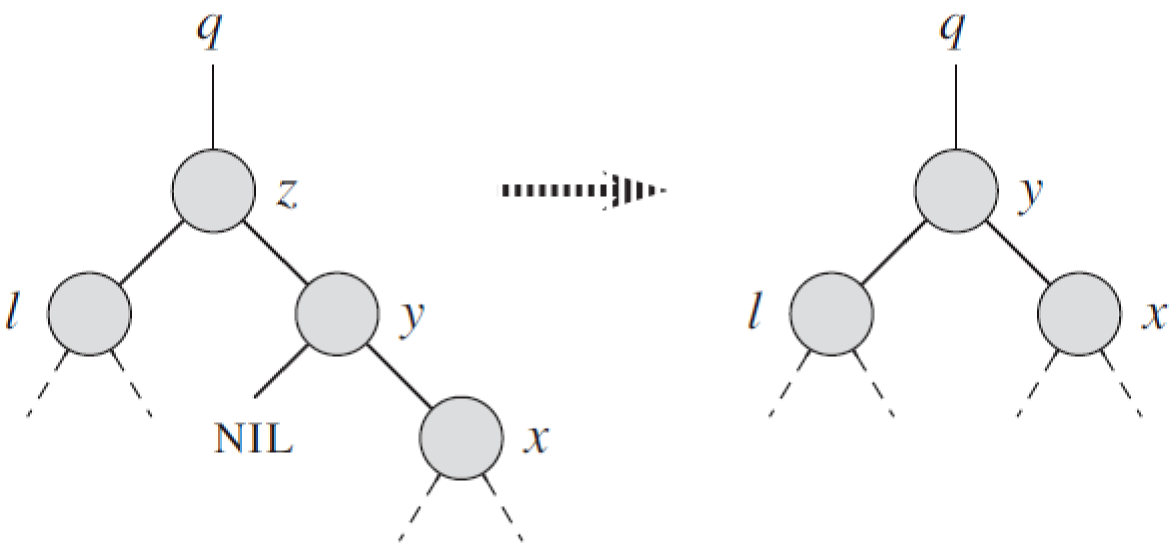
- When z has only one child node, replace z by its child.





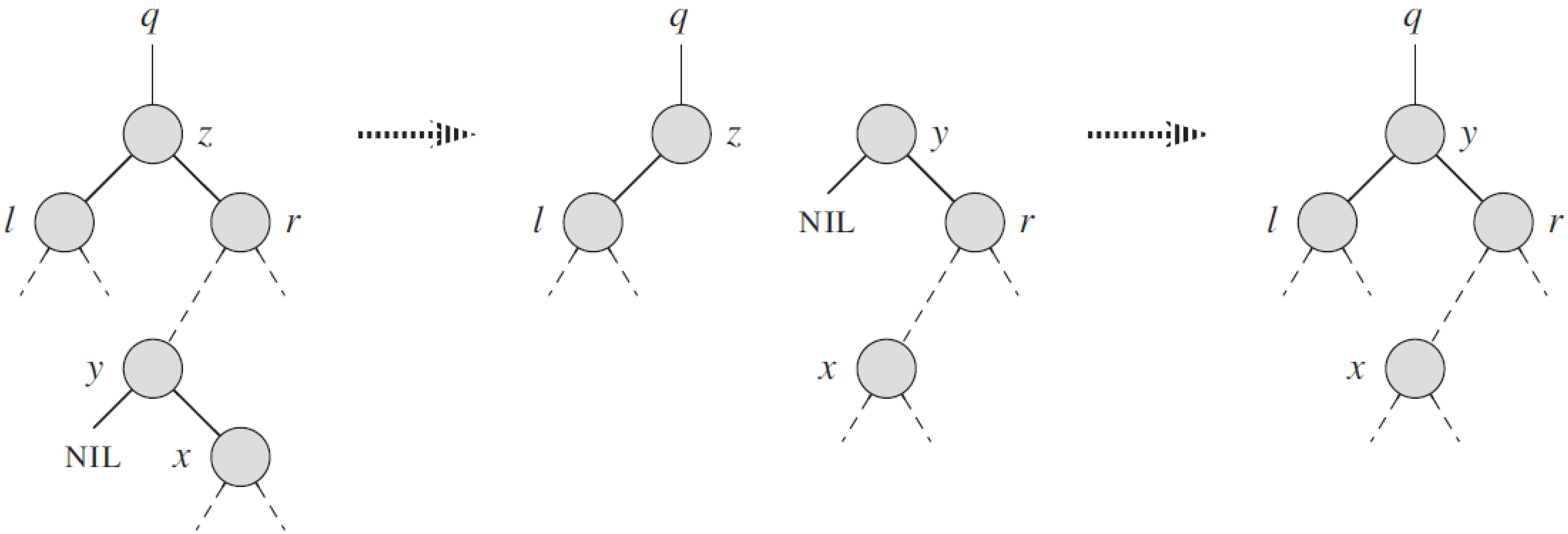
Deletion: case 3(a)

- When z's successor y is an immediate child of z:



Deletion: case 3(b)

- When z's successor y is not an immediate child of z.



Deletion: implementation

- Transplant: replace node u by v.
- Delete: remove node z.

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Exercise

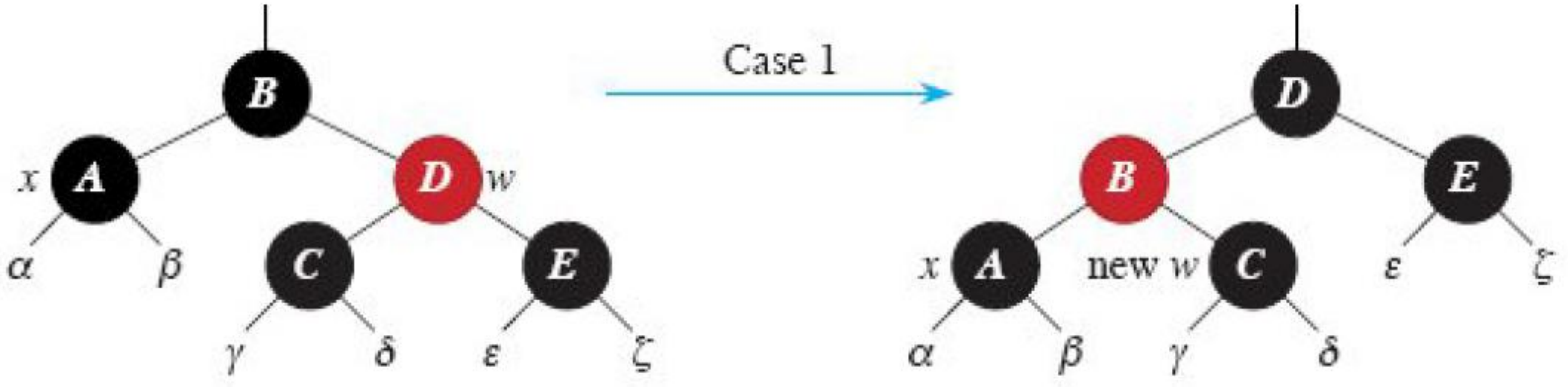
- Give concrete examples for each of the above cases.

Deletion on Red-Black Trees

- Main issue: the node removed may be black, in which case the invariant may be violated after removal.
- We imagine putting an **extra black** on the position where a node is removed (call it x), then attempt to remove this extra black.
- Divide into four cases (see the following).

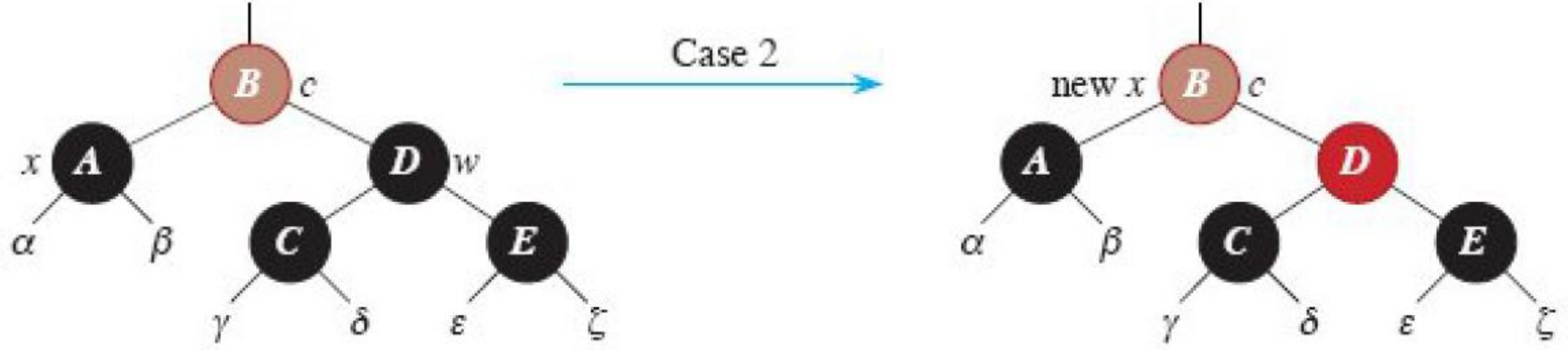
Fixup for deletion: case 1

- The sibling of x is red:
  - Recolor B and D, then rotate, so the sibling of x becomes black.



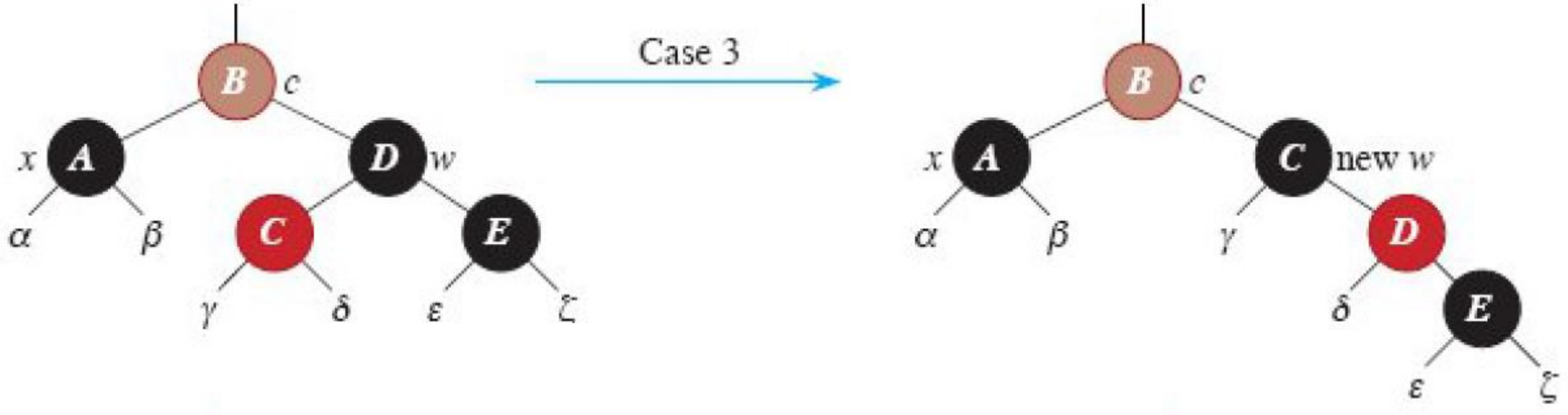
Fixup for deletion: case 2

- Sibling of x is black, with two black child nodes.
  - Recolor D, moving x one step higher.



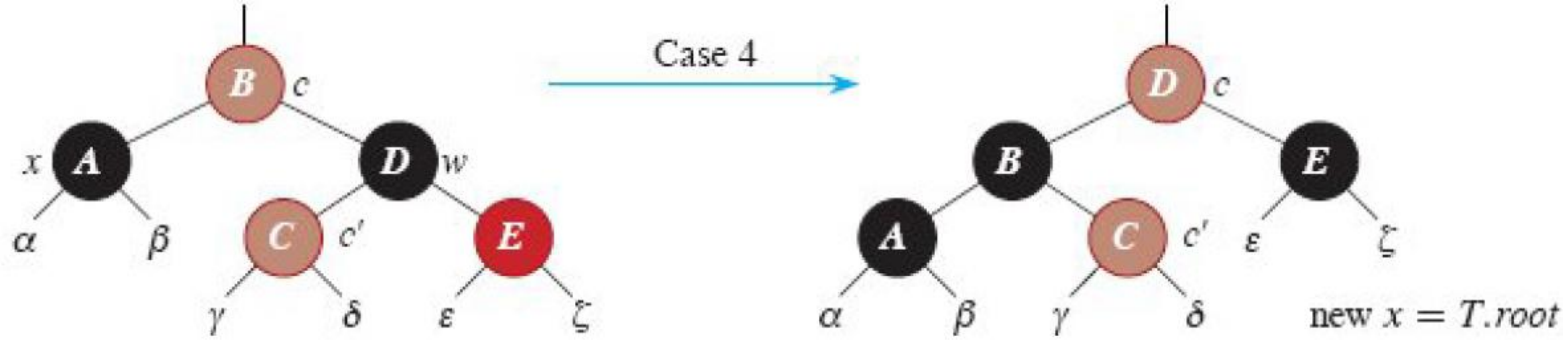
Fixup for deletion: case 3

- Sibling of x is black, and whose left child node is red and right child node is black.
  - Recolor C and D, then perform a right rotation.



Fixup for deletion: case 4

- Sibling of x is black, and whose right child node is red.
  - Recolor B, D and E, and perform a left rotation.



Exercise

- Perform some deletion starting from the tree built in the previous lecture, starting from:

  1, 2, 3, 4, 5, 6, 7, 8

Other Self-Balancing Trees

- **AVL Trees (Problem 13-3)**
  - Height balanced: for any node x, the height of left and right subtrees of x differ by at most 1. The height is maintained at each node (as $x.h$).
  - If an insertion makes height of left and right subtree differ by 2 for some node x, perform rotations to restore the invariant.
- **Treaps (Problem 13-4)**
  - Make use of randomization. Inspired by the following observation: if we have all items at the beginning and insert them in the random order, then the resulting tree will be most likely balanced.
  - Assign a *priority* to each node. The tree follows heap order according to priority.

Other Self-Balancing Trees

- **Splay Trees**
  - A very interesting example of self-balancing tree introduced by Sleator and Tarjan in 1985.
  - Insert, delete and search have $O(\log n)$ amortized complexity.
  - As each leaf node is accessed, it is rotated to the root following a particular procedure, making the tree balanced in the process.
- **B-Trees**
  - Next lecture!