CSI 403
ALGORITHMS AND DATA STRUCTURES

Lecture 14 – More Height-Balanced Trees:
Red-Black Trees (Deleting)

J Marques de Carvalho

Review

- In BSTs (Binary Search Trees), the dynamic set operators (INSERT, DELETE, SEARCH, PREDECESSOR, SUCCESSOR, MAX, and MIN) all (can) take $O(h)$ time, where h is the height of the tree.
- In a height balanced tree, $h \approx lg\ N$
- In a degenerate tree, $h = N$
- We need to have a way to keep the tree balanced as we perform repeated insertions and deletions.

Review, cont’d

- RBTs (Red-Black Trees):
  - A different way of keeping a tree height-balanced
  - All leaf nodes AND the root's parent are $T.nil$
  - Five rules:

    (1) All nodes are colored either “red” or “black”

    (2) The root is black

    (3) All leaves are black

    (4) If a node is red, then both of its children are black

    (5) For any node, the number of black nodes along any path to the leaves is the same as for any *other* such path to the leaves

Review, cont'd

- RBTs (Red-Black Trees):
  - BST property is always preserved
  - BST most of the operators work:
    - SEARCH, SUCCESSOR, PREDECESSOR, MIN, MAX
  - INSERT and DELETE are different than in "plain" BST
    - Fixing the tree requires *rotations*
    - RBT's have three different cases (kinds of problems) we may have to fix as the result of an insertion

Review, cont’d

- Rotations
  - Change the tree structure, but preserve the BST property
  - Works by changing a limited (constant) number of pointers
    - “Local” adjustment
    - The amount of work done in a rotation is bound by some constant; *not* the height of the tree nor the number of nodes in it.

Review, cont'd

- RBT Insertion
  - Do a "plain" search and insert the new node as red
  - May violate one or more rules
  - Do a fixup
    - May be in one of three cases
    - Fixing one case may solve the problem; may move us to another case (1 → 2 → 3).
    - Once we've fixed the problem(s) at any one level, move up two more levels and try again until we hit a level with no problems to fix. From there up to the root, the tree is valid.

Review, cont'd

- RBT Insertion – The Three cases:
  - 1: z is red, $z.p$ is red, & z's uncle y is red
    - Re-color z, z's parent, and z's uncle (no rotation)
  - 2: z's uncle y is black, z is a right child
    - Do a left rotation around z. Takes us immediately to
  - 3: z's uncle y is black, z is a left child
    - Recolor z's parent and grandparent
    - Do a right rotation around z's grandparent
  - Move z up two levels ($z \leftarrow z$'s parent's parent), and try again as long as z's parent is red

Deleting a Node from an RBT

- RBT Deletion:
  - Only slightly more complicated than insertion
  - Similar to insert (at least from a high level)
    - Find node to delete
    - Delete / splice out (just like regular BST)
    - Clean up whatever we might have messed up
    - Cleanup runs in $O(\lg n)$ time
    - Leaves the tree balanced (h stays $O(\lg n)$)

RB-DELETE: the Pseudocode

```
RB-DELETE (T, z)                z points at the node to delete; it's not a key.
   if  z.left == T.nil or z.right == T.nil
      y = z
   else y = TREE-PREDECESSOR(z)
   if y.left != T.nil then x = y.left else x = y.right
   x.p = y.p
   if y.p == T.nil
      T.root = x
   else if y == y.p.left
             y.p.left = x
        else y.p.right = x
   if y != z
      z.key = y.key  //also copy y's satellite data (if any) into z
   if y.color == BLACK then RB-DELETE-FIXUP(T, x)
Return(y)
```

RB-DELETE-FIXUP: the Pseudocode

```
RB-DELETE-FIXUP (T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color = RED
            w.color = BLACK                                       CASE 1
            x.p.color = RED
            LEFT-ROTATE(T, x.p)
            w = x.p.right
        if w.left.color == black and w.right.color == BLACK
            w.color = RED                                         CASE 2
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK                              CASE 3
                w.color = RED
                RIGHT-ROTATE(T, w)
                w = x.p.right
             w.color = x.p.color                                  CASE 4
             x.p.color = BLACK
             w.right.color = BLACK
             LEFT-ROTATE(T, x.p)
             x = T.root
    else (same as then clause with "right" and "left" exchanged)
    x.color = BLACK
```

RB-DELETE-FIXUP: the Pseudocode

```
RB-DELETE-FIXUP(T, x)
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left              // is x a left child?
3          w = x.p.right             // w is x's sibling
4          if w.color == RED
5              w.color = BLACK
6              x.p.color = RED                     case 1
7              LEFT-ROTATE(T, x.p)
8              w = x.p.right
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED                       case 2
11             x = x.p
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK
15                 w.color = RED                   case 3
16                 RIGHT-ROTATE(T, w)
17                 w = x.p.right
18             w.color = x.p.color
19             x.p.color = BLACK
20             w.right.color = BLACK               case 4
21             LEFT-ROTATE(T, x.p)
22             x = T.root
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

Deleting a Node from an RBT

- In RB-DELETE-FIXUP(T, x), what is x?
  - "The node passed to RB-DELETE-FIXUP is one of two nodes: either the node that was y's sole child before y [from RB-DELETE] was spliced out if y had a child that was not the sentinel $T.nil$, or, if y had no children, x is the sentinel $T.nil$. In the latter case, the unconditional assignment in line 7 ($x.p = y.p$) guarantees that x's parent is now the node that was previously y's parent, whether x is a key-bearing internal node, or the sentinel, $T.nil$."
  - In other words,
    - y is the node that was spliced out
    - x is either:
      - y's sole non-sentinel child before y was spliced out, or
      - The sentinel if y had no children.

Deleting a Node from an RBT

- If y is black, what could we have violated?
  1) All nodes are colored either "red" or "black"
  2) The root is black

     If y is the root and x is red, then the root has become red.
  3) All leaves are black
  4) If a node is red, then its children are both black

     if x and $y.p$ are both red, then #4 is violated.

Deleting a Node from an RBT

- If y is black, what could we have violated?
  5) For any node, the black height is the same for all paths to the leaves

     Any path containing y now has 1 fewer black nodes.

     Fix by giving x an "extra black"

     Add 1 to the black height on paths containing x

     Now #5 is OK, but #1 is not.

     x is now either *doubly black* (if x.color == BLACK) or *red and black* (if x.color == RED)

RB-DELETE-FIXUP

- The idea behind the fixup routine:
- Move the extra black up the tree until

  x points to a red & black node → turn it into a black node,

  x points to the root → just remove the extra black, or

  we can do certain rotations and re-colorings and finish.

Within the `while` loop:

x always points to a non-root doubly-black node.

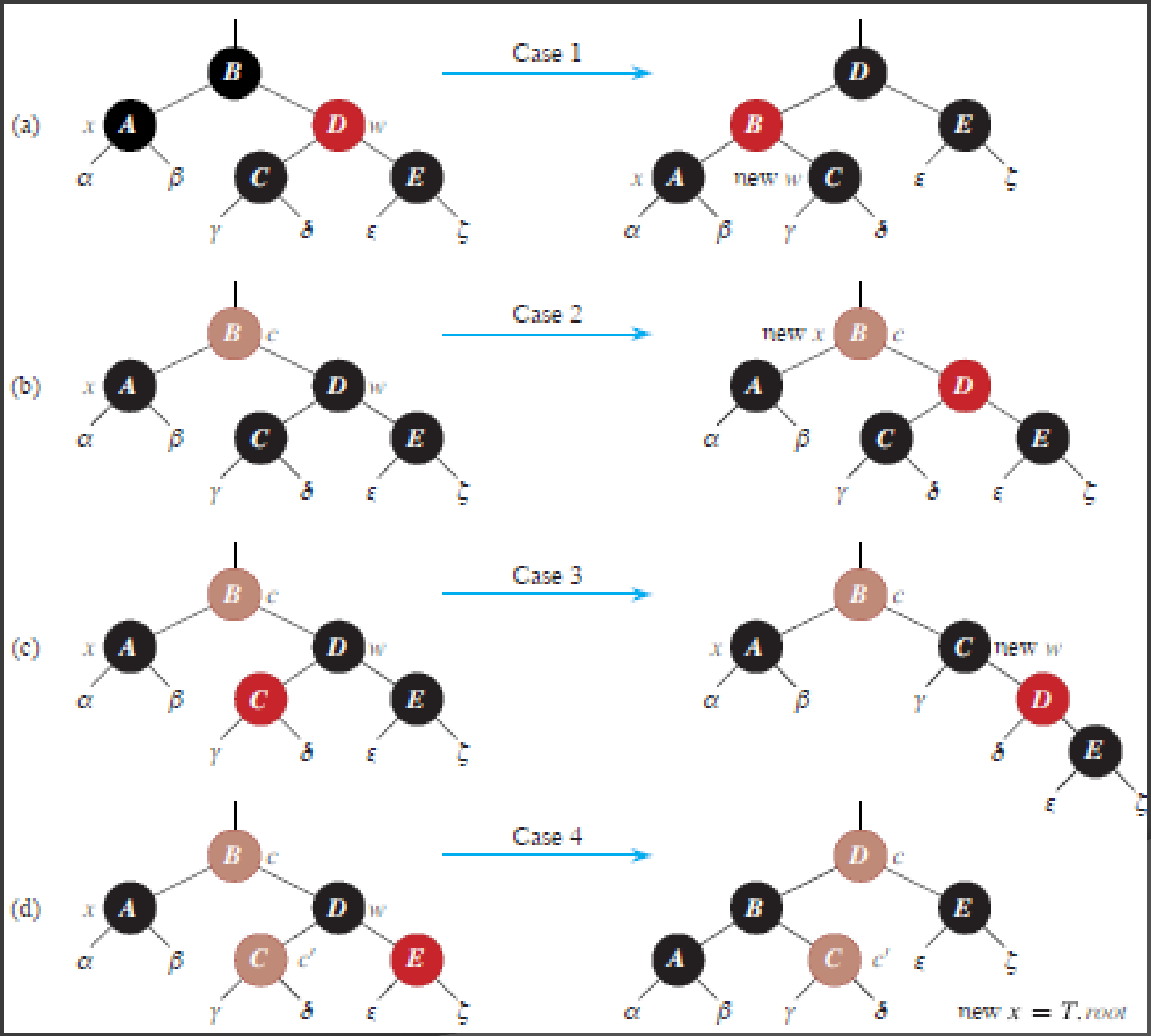
w is x's sibling.

w cannot be $T.nil$, since that would violate property 5 at $x.p$.

RB-DELETE-FIXUP (2)

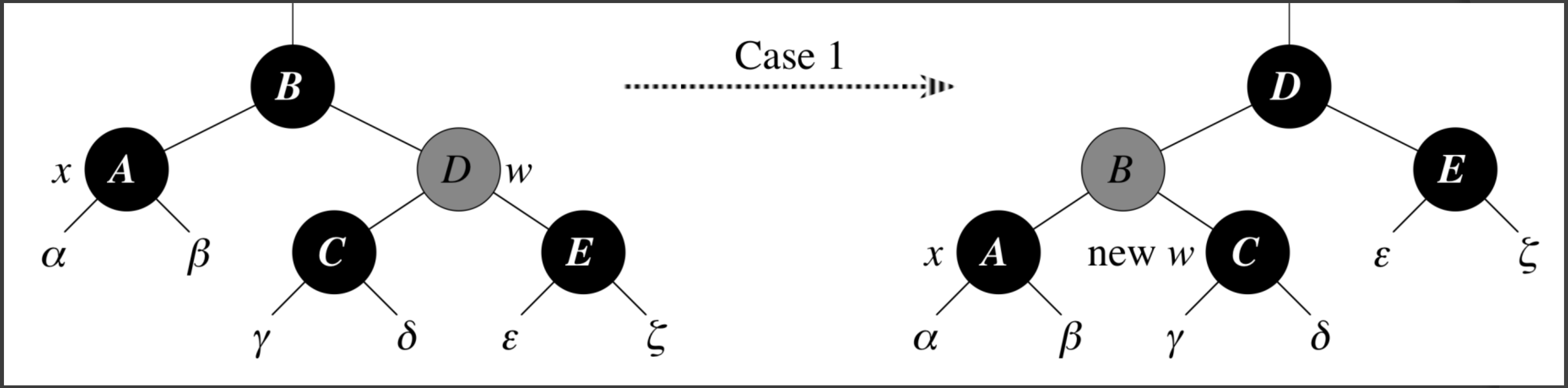
- There are 8 cases, 4 of which are symmetric to the other 4.
- As with insertion, the cases are not mutually exclusive (fixing one case may create a problem that falls into another case).
- We'll look at cases in which x is a left child.

RB-DELETE-FIXUP (3)



Case 1

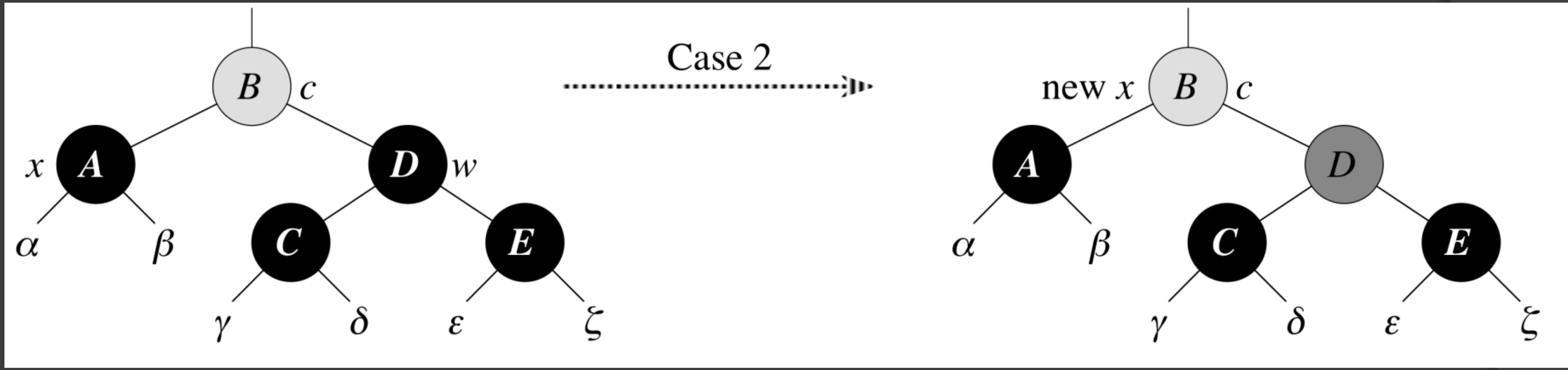
- w is red and must have black children (otherwise property 4 violated)



- w must have black children
- Make w black and $x.p$ red
- Then left rotate on $x.p$
- New sibling of x was a child of w before rotation, so it must be black
- Go immediately to case 2, 3, or 4

Case 2

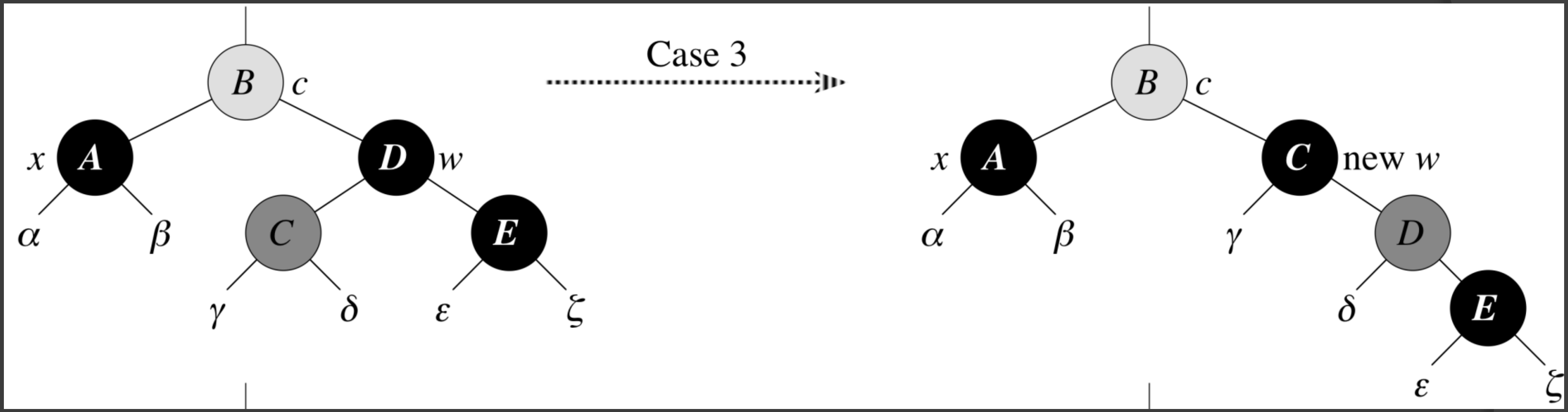
- w is black and both of w's children are black



- Take 1 black off x (singly black) and off w (red)
- Move that black to $x.p$
- Do the next iteration with $x.p$ as the new x
- If we entered this case from case 1, then $x.p$ was red
- New x is red & black → color of new x is RED → loop terminates. The new x is made black in the last line

Case 3

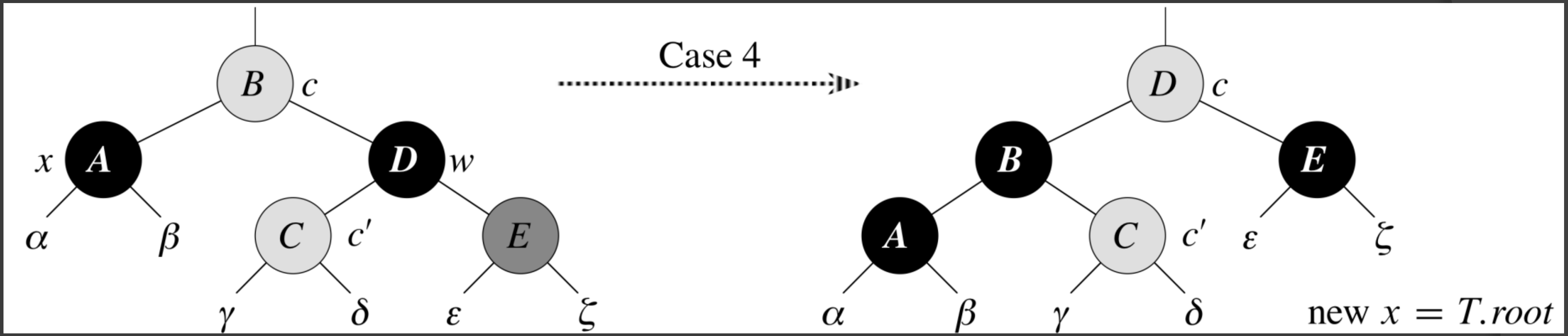
- w is black, w's left child is red, and w's right child is black



- Make w red and w's left child black
- Then right rotate on w
- New sibling w of x is black with a red right child. Go on immediately to Case 4

Case 4

- w is black, w's left child is black, and w's right child is red



- Make w be $x.p$'s color (c)
- Make $x.p$ black and w's right child black
- Then left rotate on $x.p$
- Remove the extra black on x (x is now singly black) without violating any red-black properties
- All done. Setting x to root causes the loop to terminate

Analysis of Deletion

- $O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP
- Within RB-DELETE-FIXUP:
  - Case 2 is the only case where more iterations occur
  - x moves up 1 level
  - Hence, no more than $O(\lg n)$ iterations
  - Cases 1, 3, & 4 have 1 rotation each, so there can't me any more than 3 rotations in all
- Hence, $O(\lg n)$ time (total)