CH08-320201

Algorithms and Data Structures

ADS

Lecture 15

Dr. Kinga Lipskoch

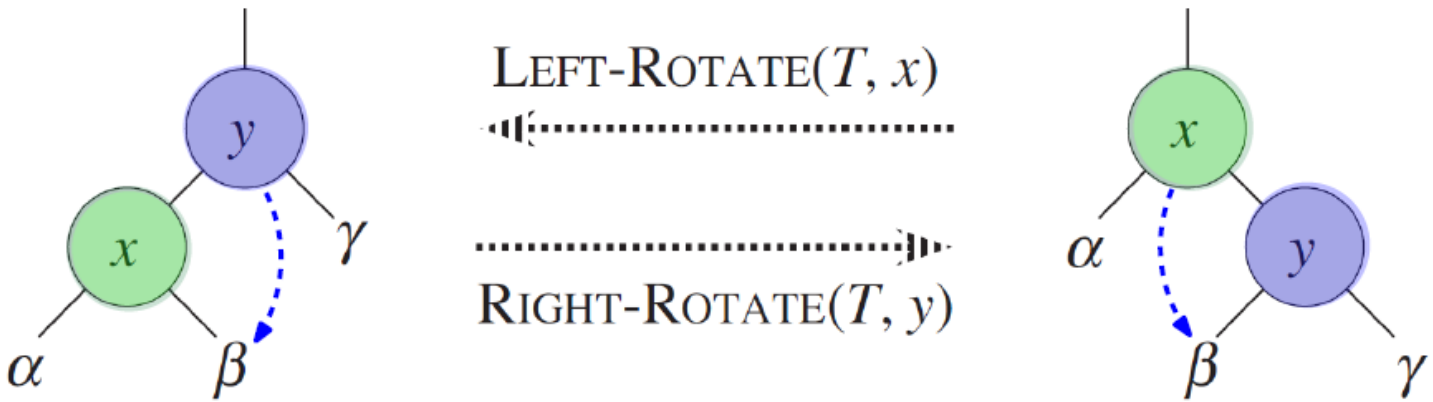
Spring 2019

Operations

- Querying
  - Search/Minimum & Maximum/Successor & Predecessor
  - Just as in normal BST
  - $O(\lg n)$
- Modifying
  - Tree-Insert/Tree-Delete $\rightarrow O(\lg n)$
  - But, need to guarantee red-black tree properties:
    - must change color of some nodes
    - change pointer structure through rotation

Rotations (1)

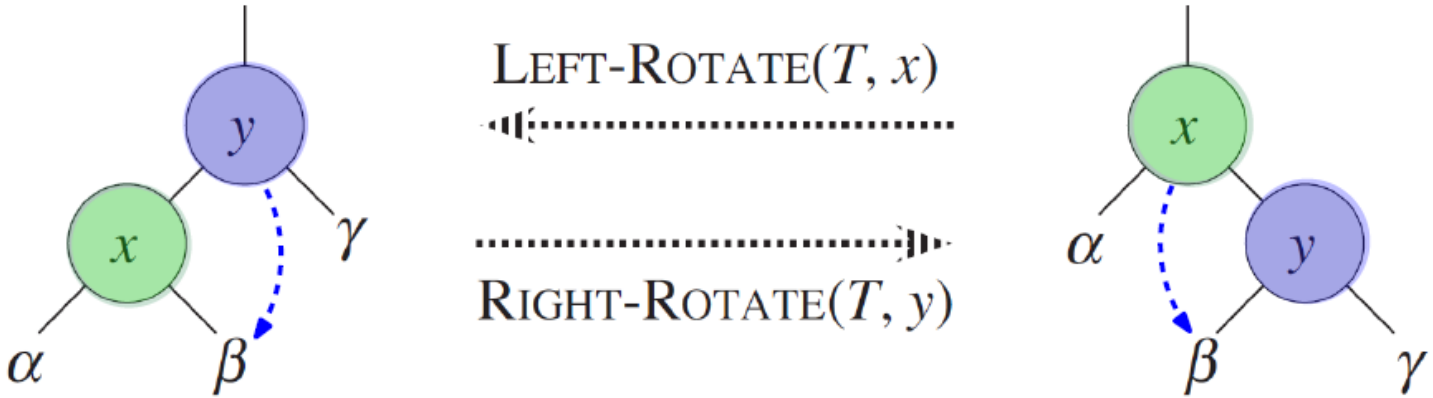
- *Right-Rotate*(T, y):
  - node y becomes right child of its left child x.
  - new left child of y is former right child of x.
- *Left-Rotate*(T, x):
  - node x becomes left child of its right child y.
  - new right child of x is former left child of y.



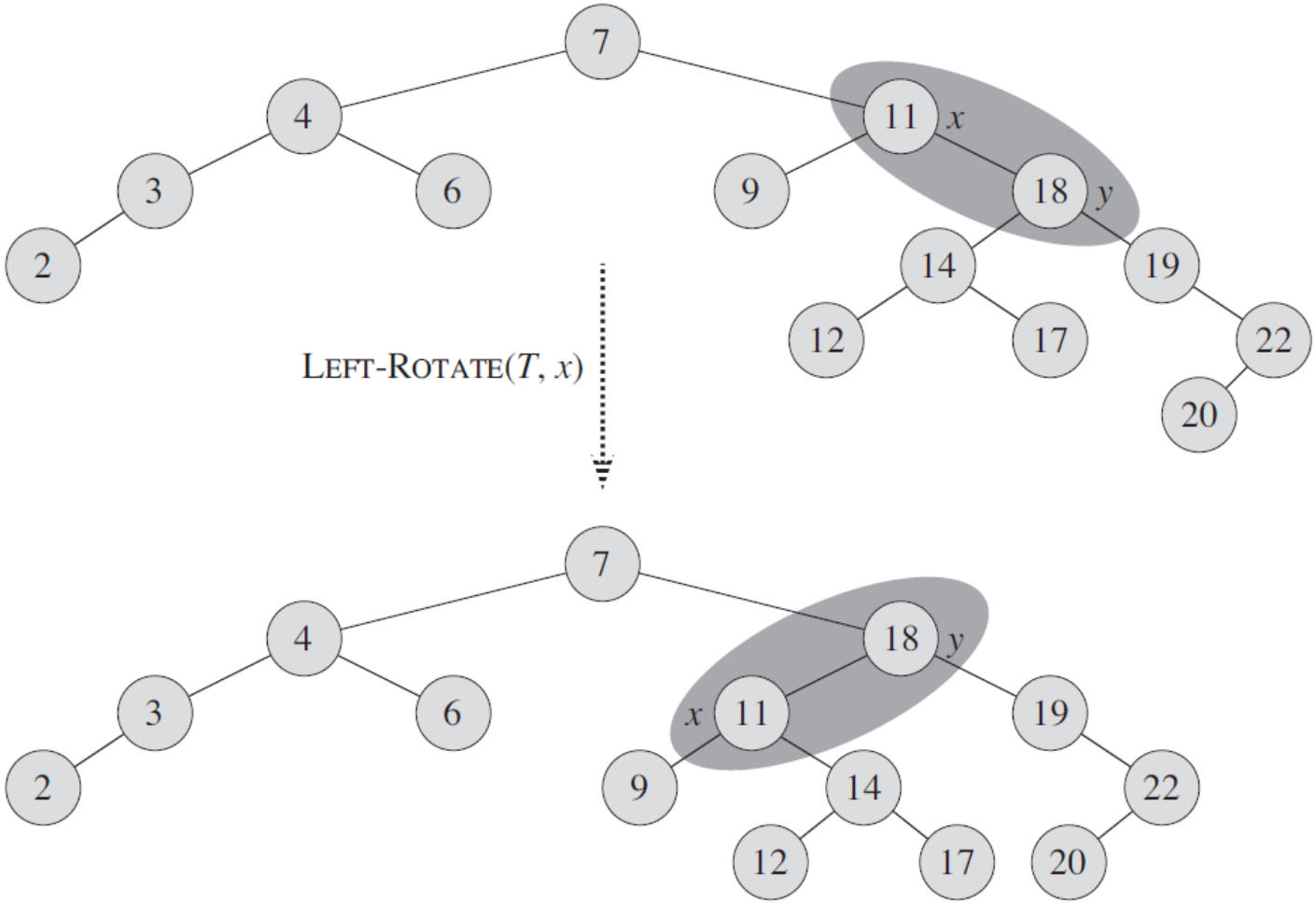
Rotations (2)

BST property is preserved:

- (left): $key(\alpha) \leq x.key \leq key(\beta) \leq y.key \leq key(\gamma)$
- (right): $key(\alpha) \leq x.key \leq key(\beta) \leq y.key \leq key(\gamma)$



Rotation: Example



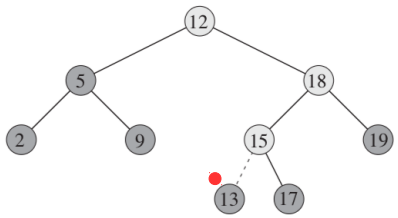
Rotation Pseudocode

```
LEFT-ROTATE(T, x)
 1   y = x.right                  // set y
 2   x.right = y.left             // turn y's left subtree into x's right subtree
 3   if y.left ≠ T.nil
 4       y.left.p = x
 5   y.p = x.p                    // link x's parent to y
 6   if x.p == T.nil
 7       T.root = y
 8   elseif x == x.p.left
 9       x.p.left = y
10   else x.p.right = y
11   y.left = x                   // put x on y's left
12   x.p = y
```

Time complexity: $O(1)$

Insertion

```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```




```
RB-INSERT(T, z)
 1   y = T.nil
 2   x = T.root
 3   while x ≠ T.nil
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == T.nil
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
14   z.left = T.nil
15   z.right = T.nil
16   z.color = RED
17   RB-INSERT-FIXUP(T, z)
```

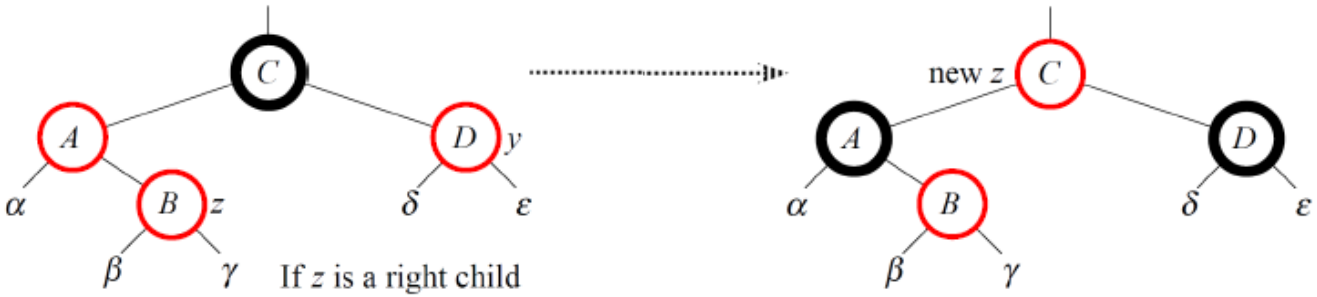
Fixing Red-Black Tree Properties

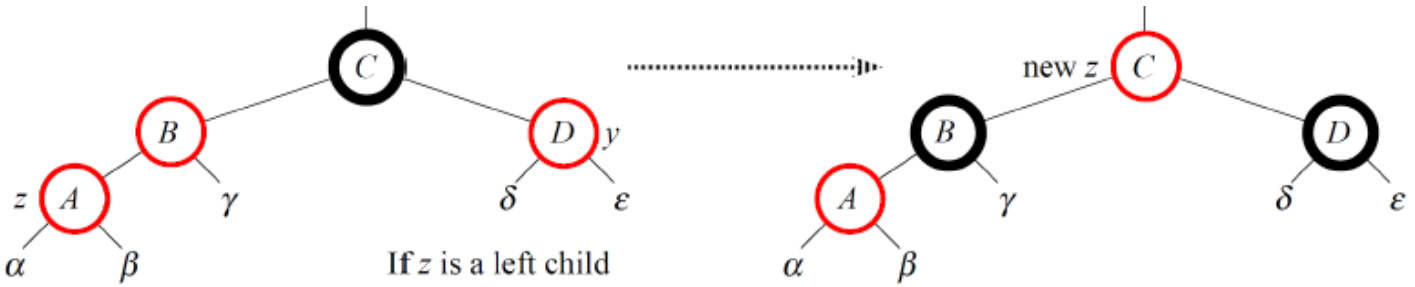
- We are inserting a **red** node to a valid red-black tree.
- Which properties may be violated?
  1. Duh: Cannot be violated. ✓
  2. RooB: Violated if inserted node is root. ✗
  3. LeaB: Inserted node is not a leaf, i.e., no violation. ✓
  4. BredB: Violated if parent of inserted node is red. ✗
  5. BH: Not affected by red nodes, i.e., no violation. ✓

Fixing BredB

- BredB for node z is violated, if $z.p$ is red.
- Then, $z.p.p$ is black. (BredB property)
- We need to consider different cases depending on the uncle y of z, i.e., the child of $z.p.p$ that is not $z.p$.
- There are 6 cases:
  - $z.p$ is left child of $z.p.p$
    - y is red (Case 1)
    - y is black
      - z is right child of $z.p$ (Case 2)
      - z is left child of $z.p$ (Case 3)
  - $z.p$ is right child of $z.p.p$
    - y is red (symmetric to Case 1)
    - y is black
      - z is right child of $z.p$ (symmetric to Case 3)
      - z is left child of $z.p$ (symmetric to Case 2)

Case 1 (Red Uncle)



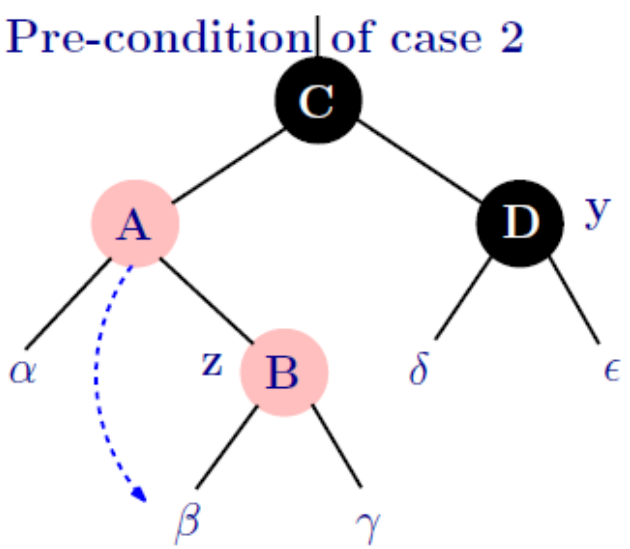


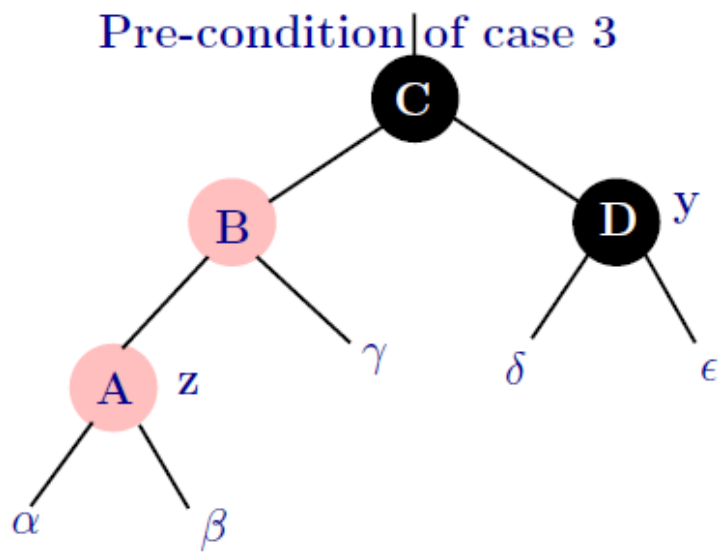

```
2     if z.p == z.p.p.left
3         y = z.p.p.right
4         if y.color == RED
5             z.p.color = BLACK
6             y.color = BLACK
7             z.p.p.color = RED
8             z = z.p.p
```

Case 1

There is a new problem, if *z.p.p.p* is red.
Algorithm needs to continue with *z.p.p*.

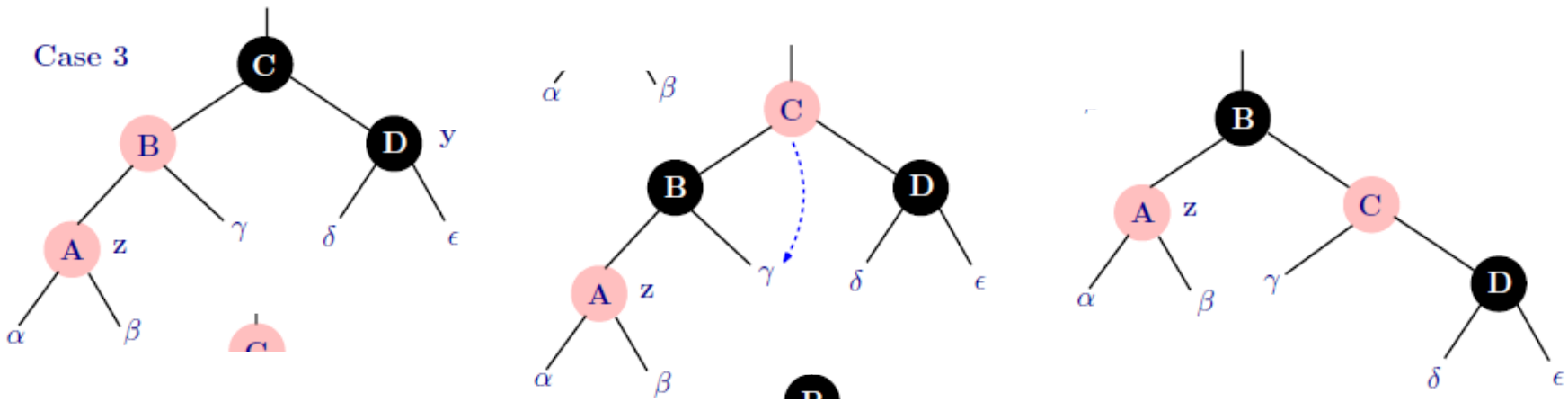
Case 2 (Black Uncle, z Right Child)






```
9     else if z == z.p.right
10        z = z.p
11        LEFT-ROTATE(T, z)        Case 2
```

Case 3 (Black Uncle, z Left Child)




```
12
13
14
```

$z.p.color = \text{BLACK}$
$z.p.p.color = \text{RED}$
RIGHT-ROTATE$(T, z.p.p)$

Case 3

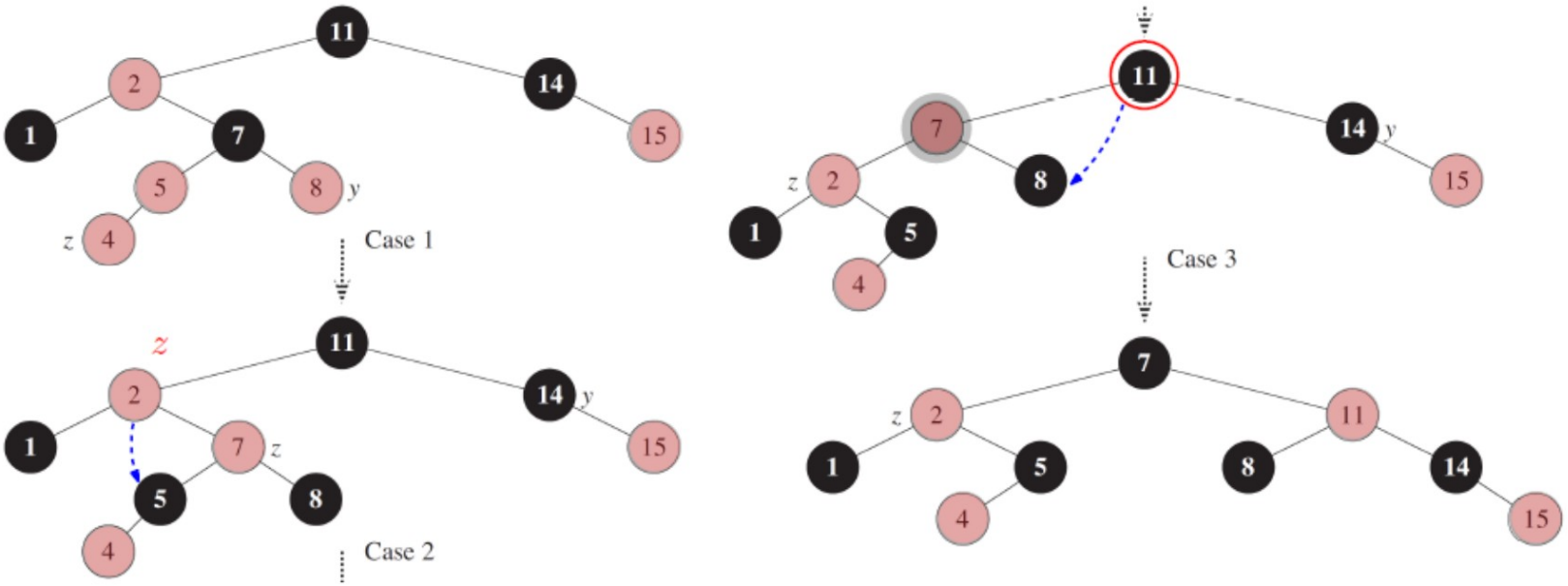
Putting It All Together

- We need to put the 3 cases (and the 3 symmetric cases) together.
- Moreover, we need to propagate the considerations upwards (see Case 1).
- Finally, we have to fix RooB.

RB-INSERT-FIXUP(T, z)

```
1   while z.p.color == RED
2       if z.p == z.p.p.left
3           y = z.p.p.right
4           if y.color == RED
5                                  z.p.color = BLACK
6                                  y.color = BLACK
7                                  z.p.p.color = RED                 Case 1
8                                  z = z.p.p
9           else if z == z.p.right
10                                   z = z.p
11                                   LEFT-ROTATE(T, z)               Case 2
12                                 z.p.color = BLACK
13                                 z.p.p.color = RED                 Case 3
14                                 RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

Insert Example



Time Complexity

- In worst case, we have to go all the way from the leaf to the root along the longest path within the tree.
- Hence, running time is $O(h) = O(\lg n)$ for the fixing of the red-black tree properties.
- Overall, running time for insertion is $O(h) = O(\lg n)$.
- Example for building up a red-black tree by iterated node insertion:
  `http://www.youtube.com/watch?v=vDHFF4wjWYU`