Red-Black Trees

Alexandre David
B2-206

Why?

- Operations on binary search tree in *O(height)* but
  - this is bad if the height is large.
  - Unbalanced trees give large heights.
  - $\Rightarrow$ Keep trees balanced.
- Red-back trees = binary search trees with a color per node (red/black) that is approximately balanced.
- ? What is a balanced tree?

Balanced Search Trees

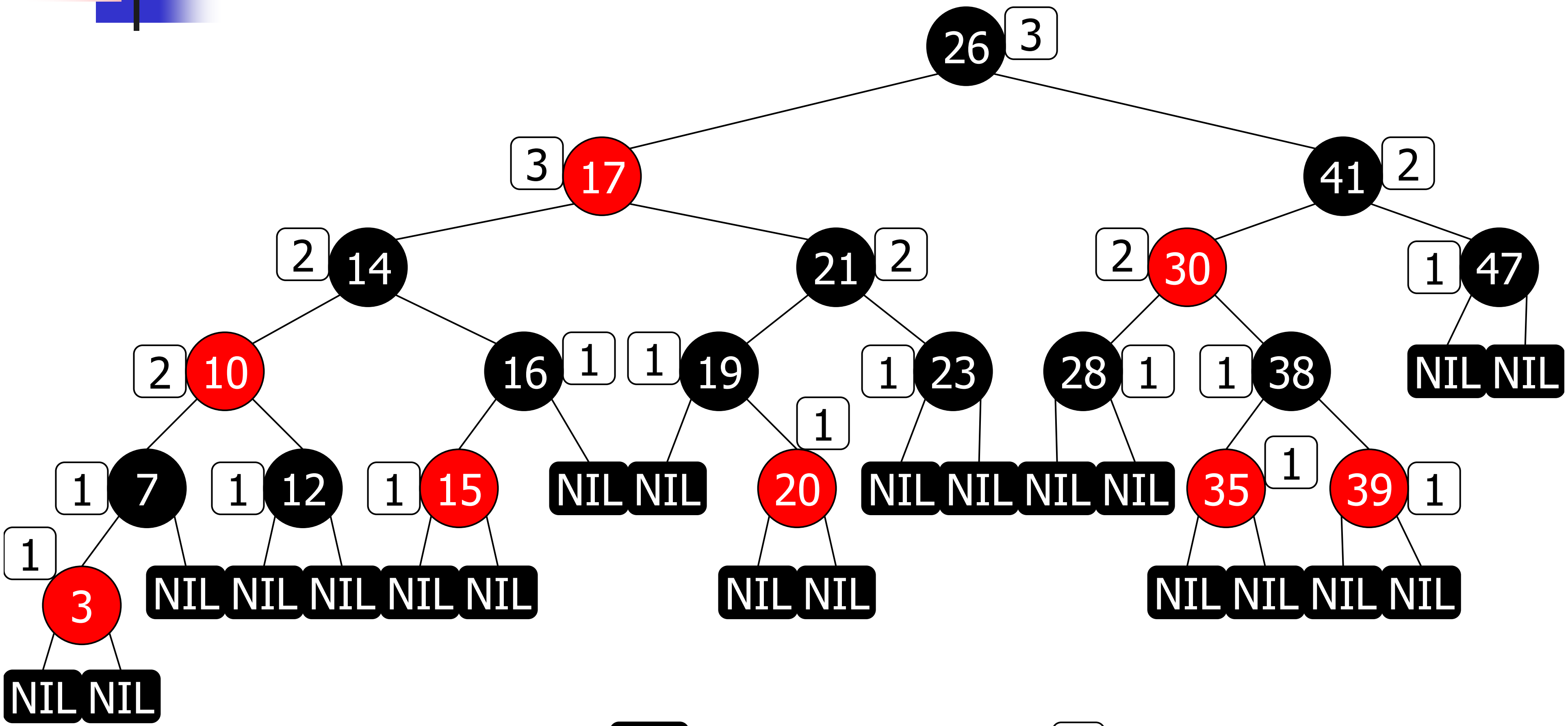
- Balanced search trees: Search-tree data structure for which a height in $O(\lg n)$ is *guaranteed* when implementing a dynamic set with n item.
- Examples:
  - AVL trees
  - B-trees
  - Red-black trees
  - ...

(chapter 13)

Red-Black Trees

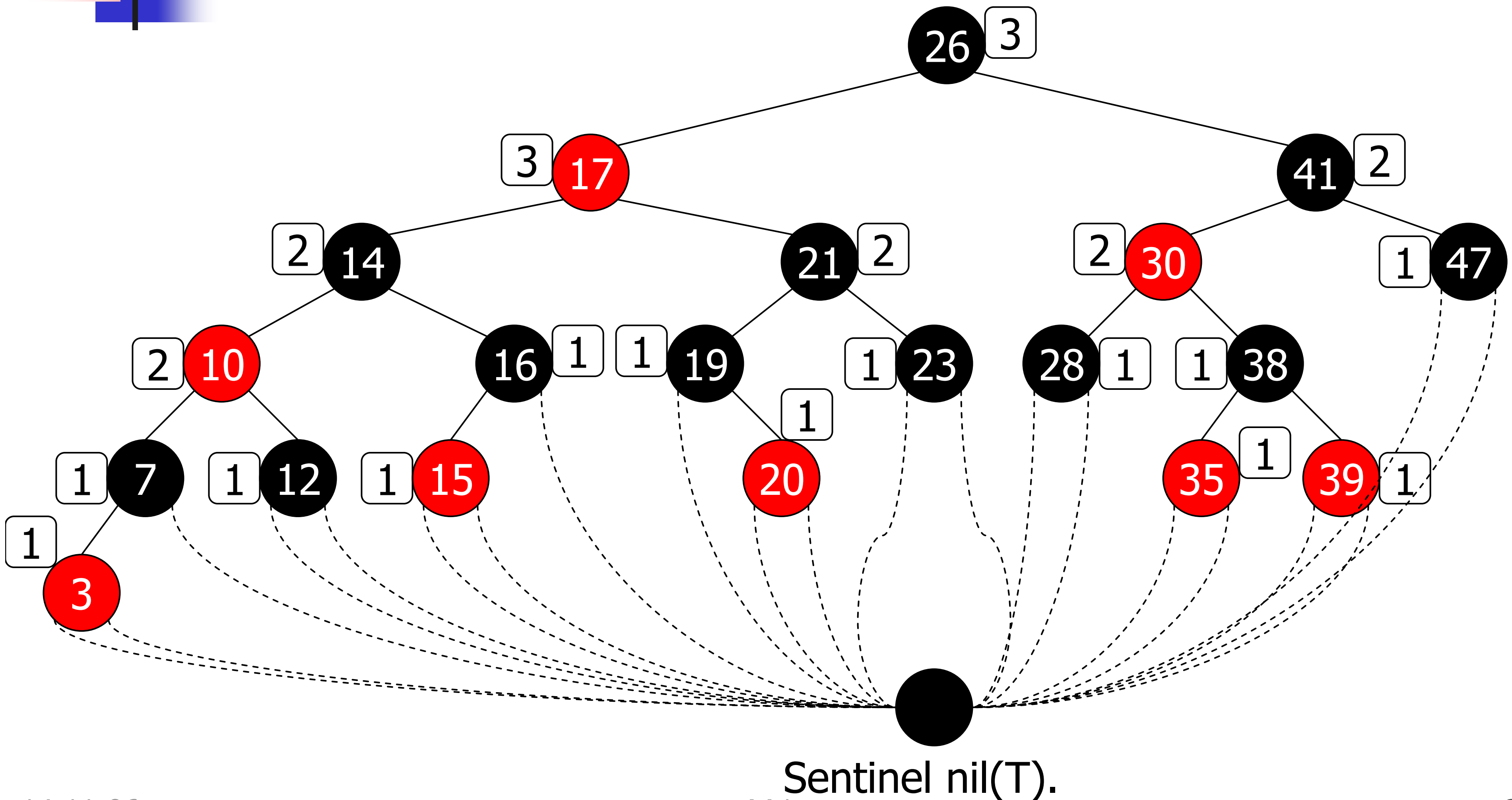
- Binary search trees satisfying red-black properties:
  1. Every node is either red or black.
  2. The root and leaves (NIL) are black.
  3. If a node is red, then its parents are black.
     - Never two reds in a row.
  4. All simple paths from any node *x* to a descendant leaf have the same number of black nodes = *black-height(x)*.

Example



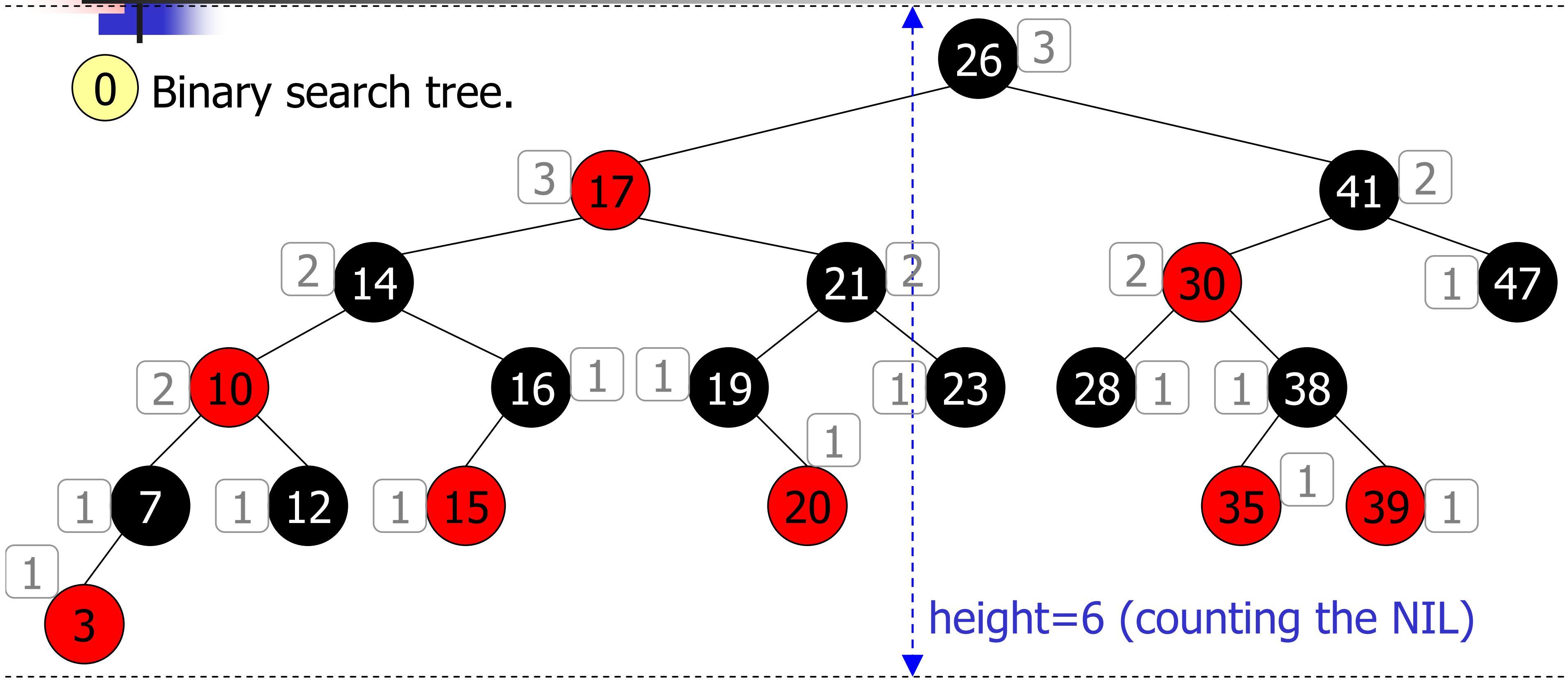
All NIL have black height 0.

Example – Simplified



Example – In Practice

0 Binary search tree.



1 Every node is either read or black.

2 The root and leaves (hidden) are black.

3 If a node is red then its parents are black.

4 Black-height property.

Height

- Bound on the height in function of the number of nodes:
  - height ≤ $2\lg(n+1)$.
  - Because red-black trees are almost balanced.
- Proof:
  - Sub-trees of x contain at least $2^{bh(x)}-1$ nodes (# of nodes in sub-binary tree, by induction on the height of x).
  - $bh(root) \geq h/2$ so $n \geq 2^{h/2}-1 \Rightarrow h \leq 2\lg(n+1)$.

The Point

- Most operations are linear in function of the height.
- The height is bounded in $O(\lg n)$.
- Most operations are bounded in $O(\lg n)$!

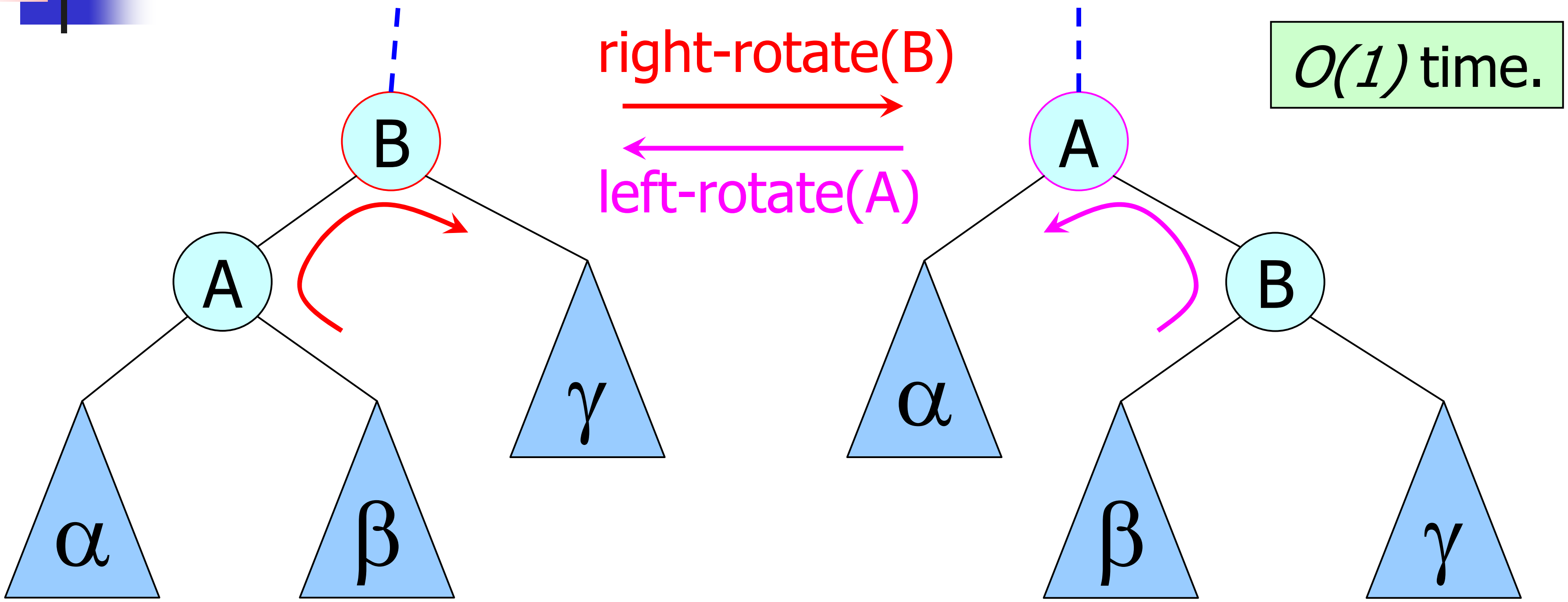
- Corollary: The operations search, min, max, successor, and predecessor run in $O(\lg n)$ time on a red-black tree with n nodes.

Modifying Operations

- The operations insert and delete modify the red-black tree:
  - insert/delete a node,
  - color changes,
  - + restructure the links of the tree via rotations.

Keep the red-black tree properties!

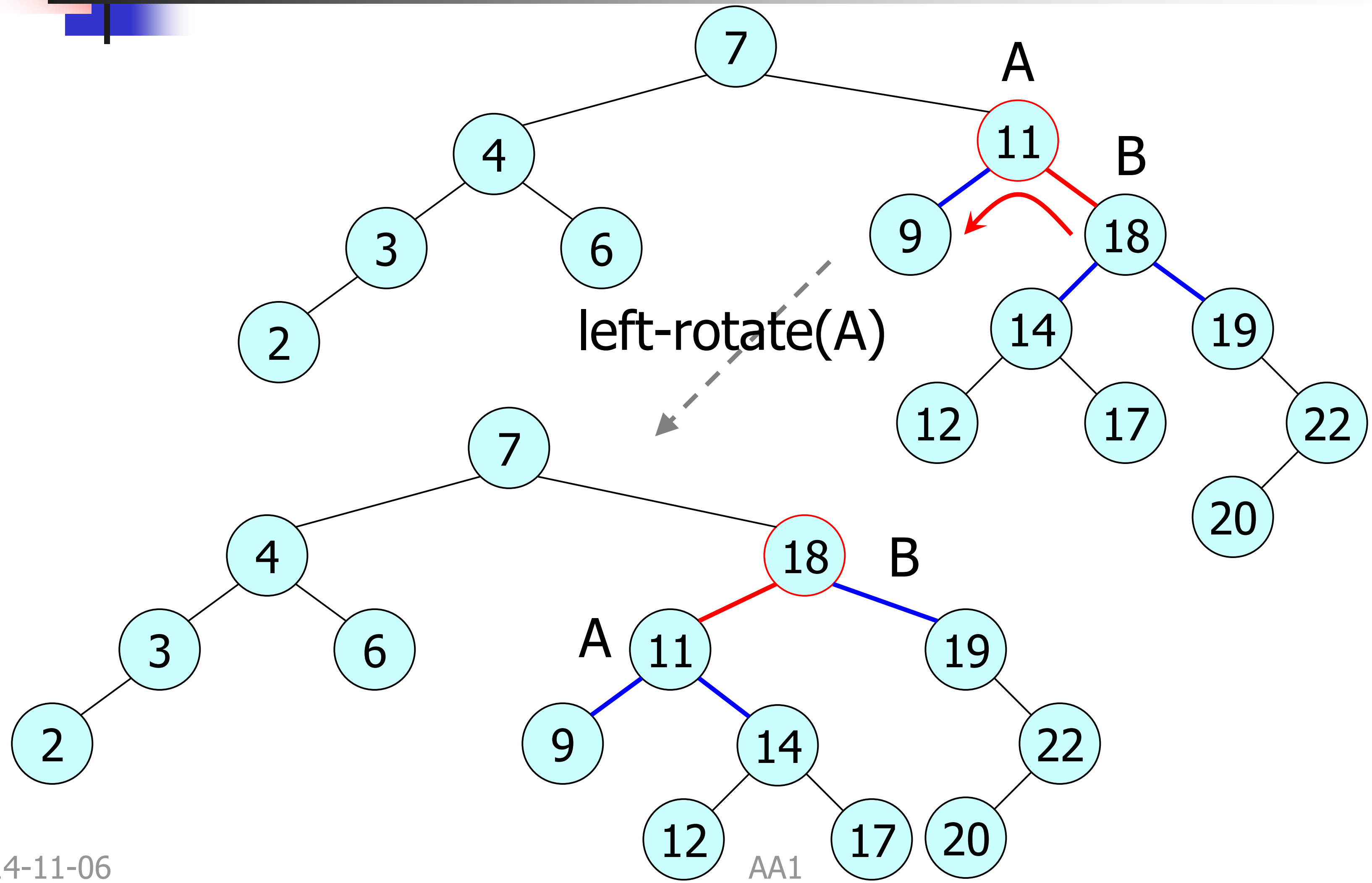
Rotations



Important property: rotations maintain the in-order ordering of keys ⇒ binary search tree property maintained.

$$\forall a \in \alpha, \forall b \in \beta, \forall c \in \gamma : a \le A \le b \le B \le c$$

Example – Left-rotate

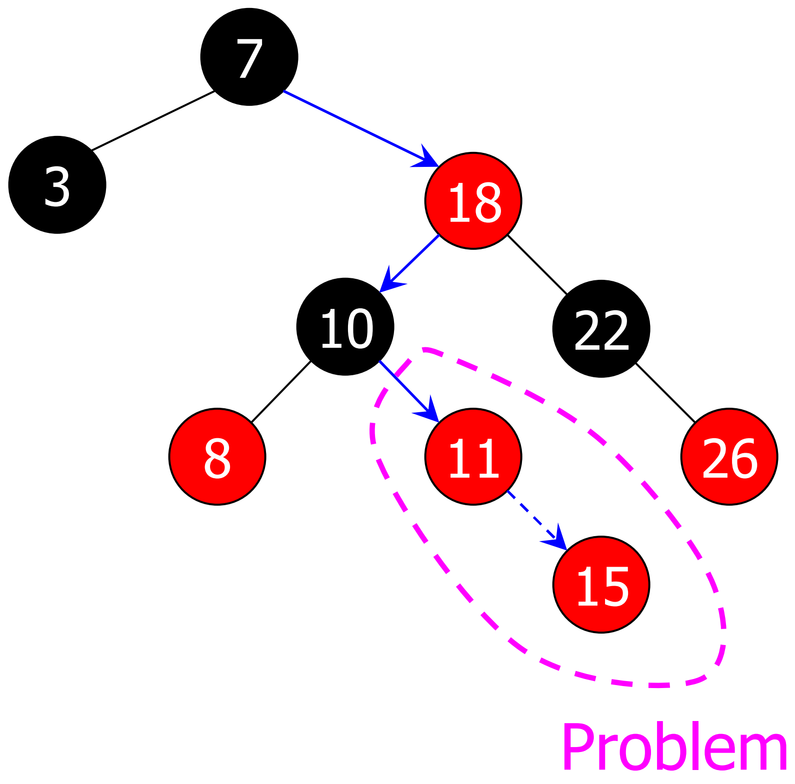


Insertion

- Idea:
  - Insert x in the binary search tree.
  - Color x red.
  - Only red-black property 3 may be violated.
  - Move the violation up the tree by re-coloring until it can be fixed by rotations and re-coloring.

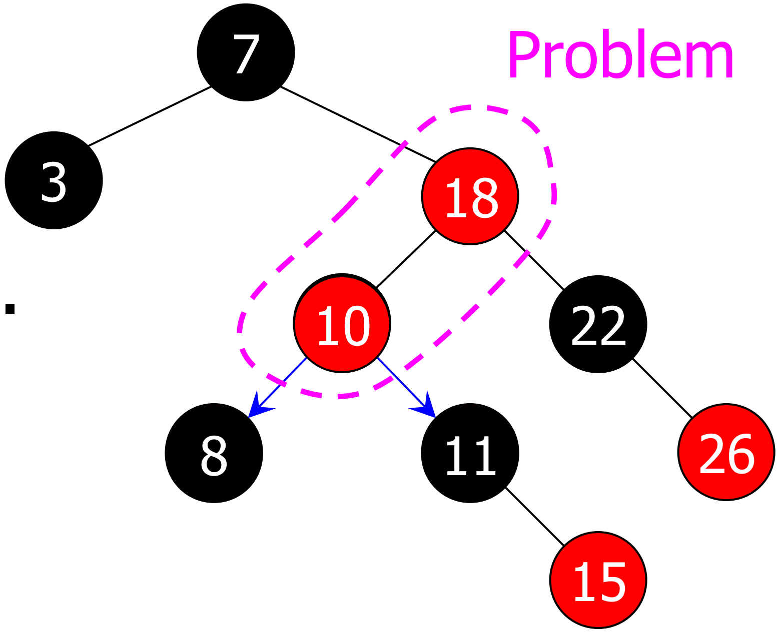
Insertion - Example

- Insert x = 15.



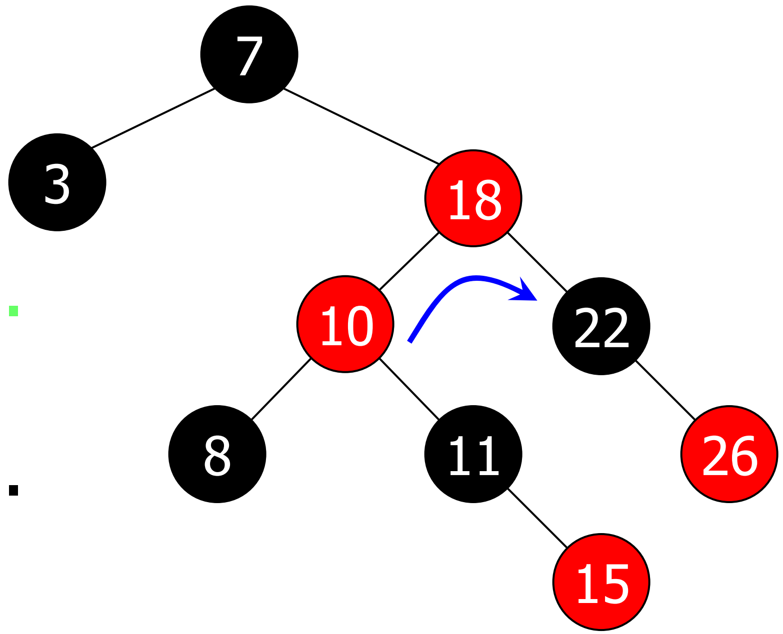
Insertion - Example

- Insert x = 15.
- Recolor, moving the violation up the tree. Black-height unchanged.



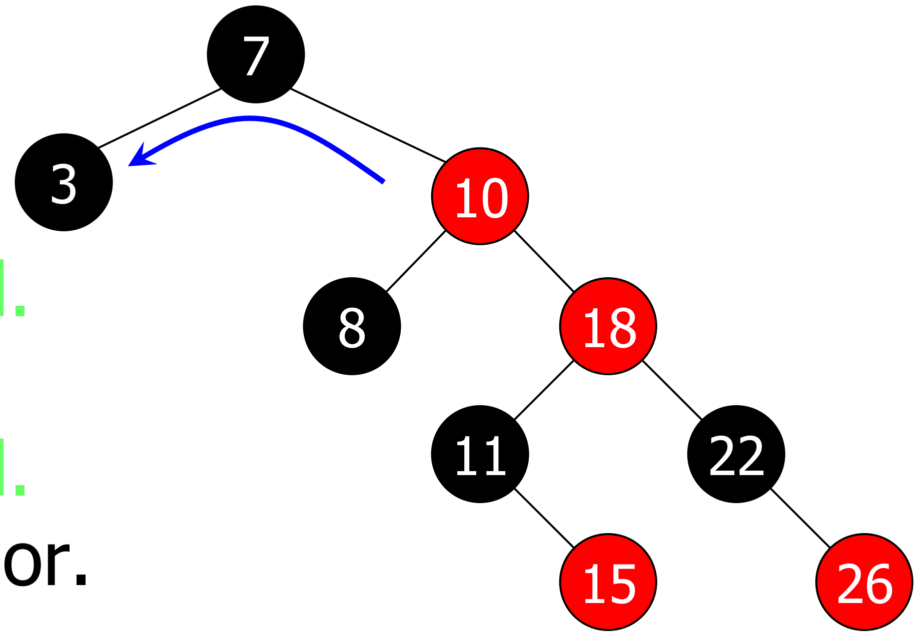
Insertion - Example

- Insert x = 15.
- Recolor, moving the violation up the tree. Black-height unchanged.
- Right-rotate(18). Black-height unchanged.



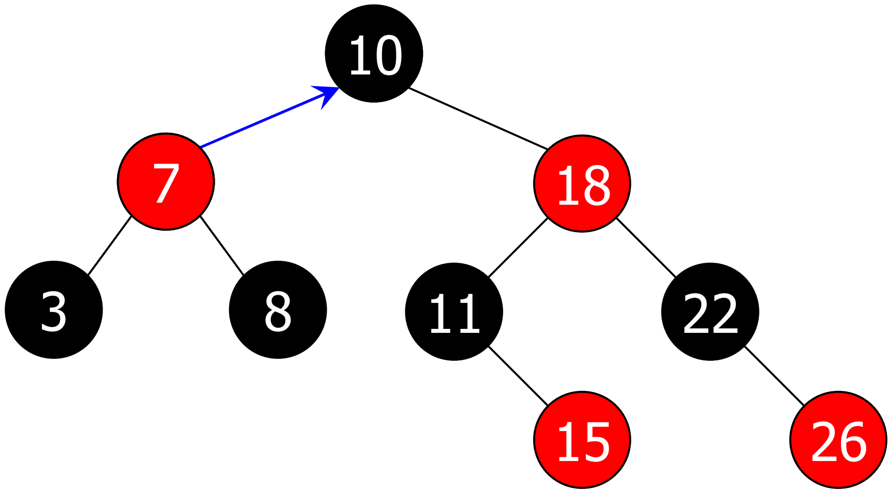
Insertion - Example

- Insert x = 15.
- Recolor, moving the violation up the tree. Black-height unchanged.
- Right-rotate(18). Black-height unchanged.
- Left-rotate(7) and recolor.

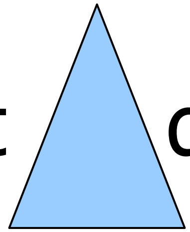
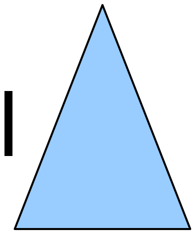


Insertion - Example

- Insert x = 15.
- Recolor, moving the violation up the tree. Black-height unchanged.
- Right-rotate(18). Black-height unchanged.
- Left-rotate(7) and recolor.



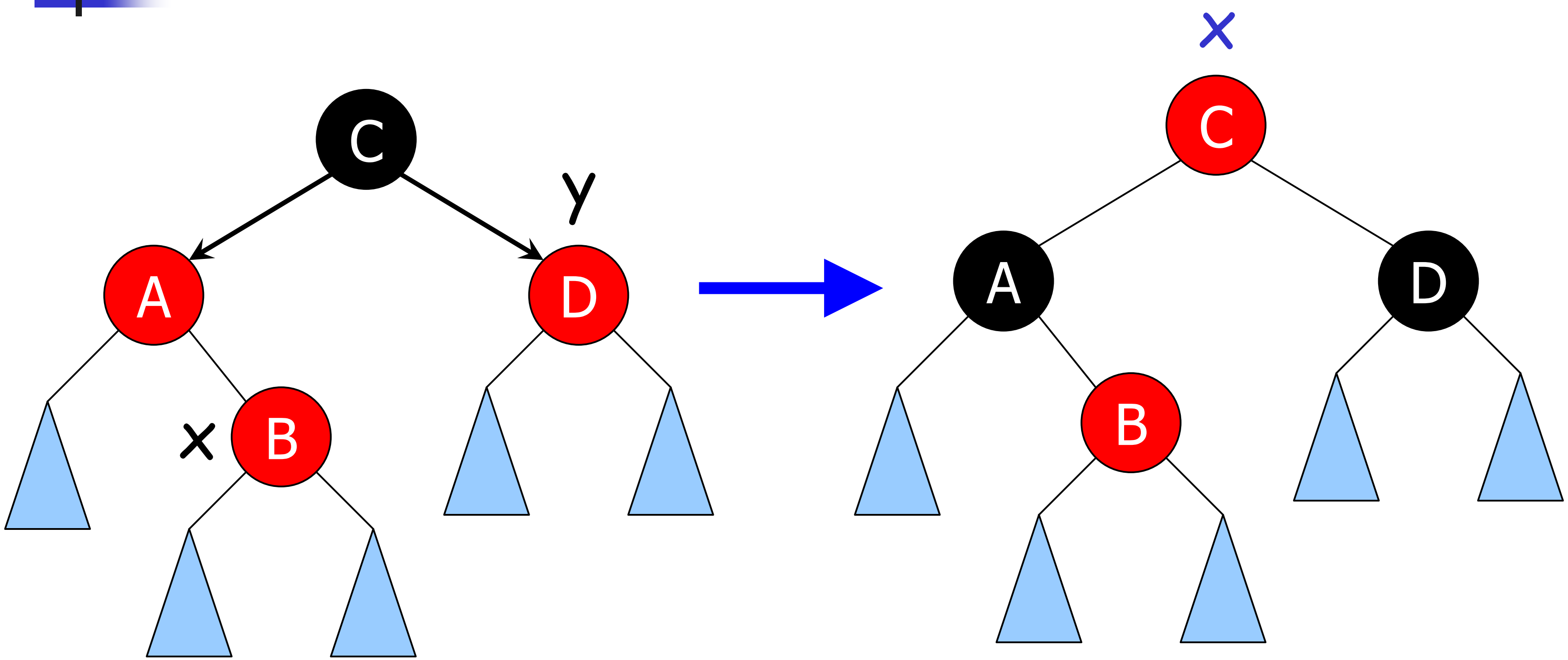
Algorithm for Insertion

- Graphical notations:
- Let  denote a sub-tree with a black root.
- All  have the same black-height (from the root).

Algorithm for Insertion

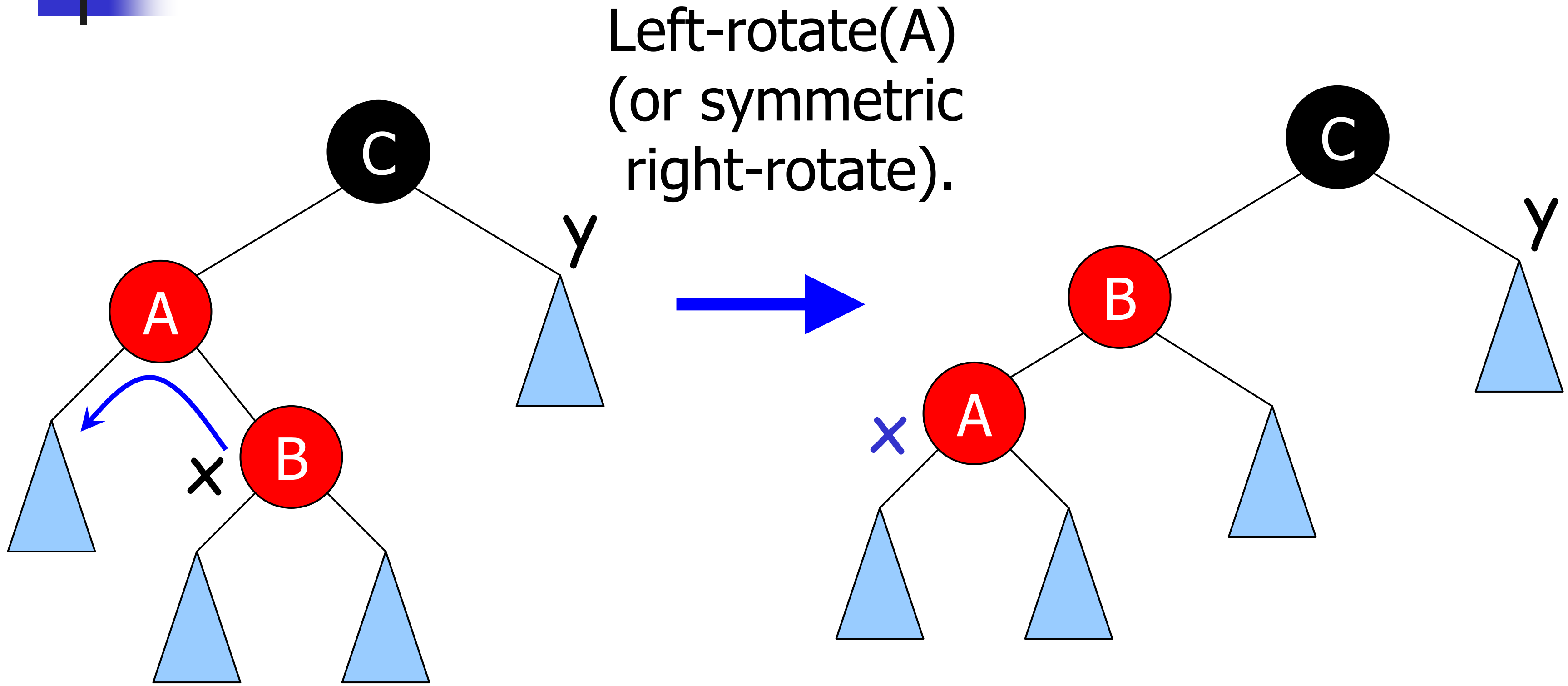
- Identify one of 3 possible cases, each describing a pattern for re-coloring or rotation (left or right):
  - Case 1: Recolor and recurse.
  - Case 2: Rotate & transform to case 3.
  - Case 3: Rotate.

Case 1: Recolor



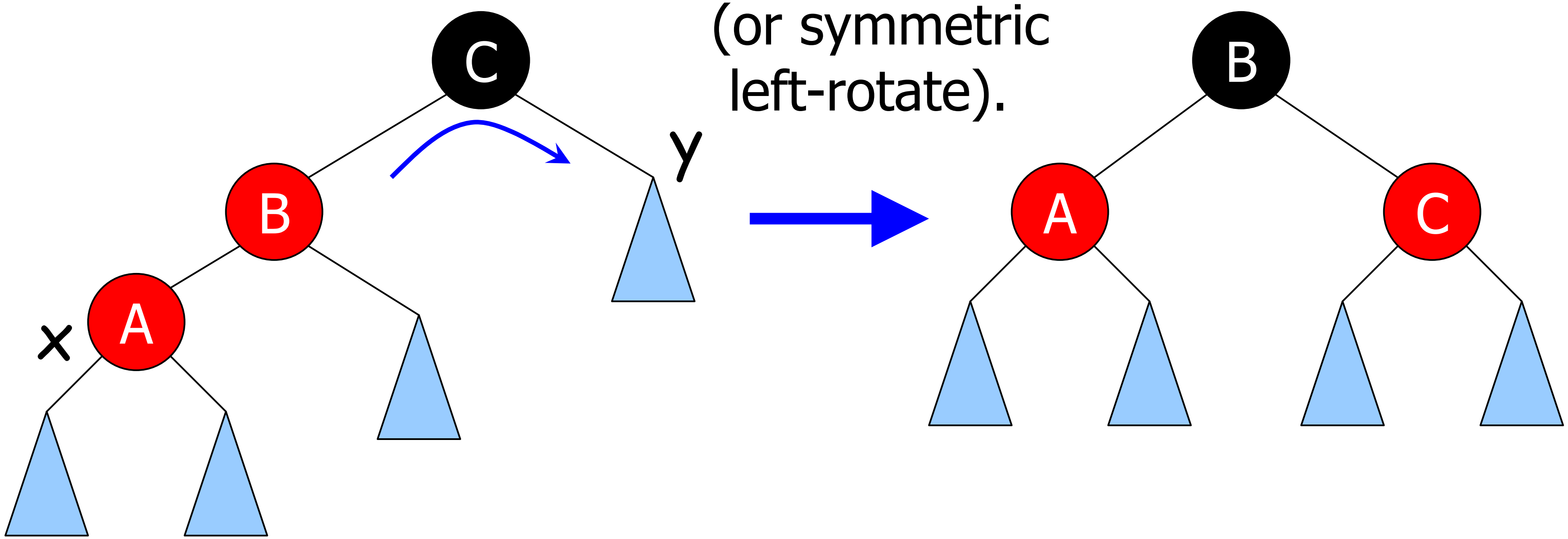
... and recurse.

Case 2: Rotate & case 3.



… and case 3.

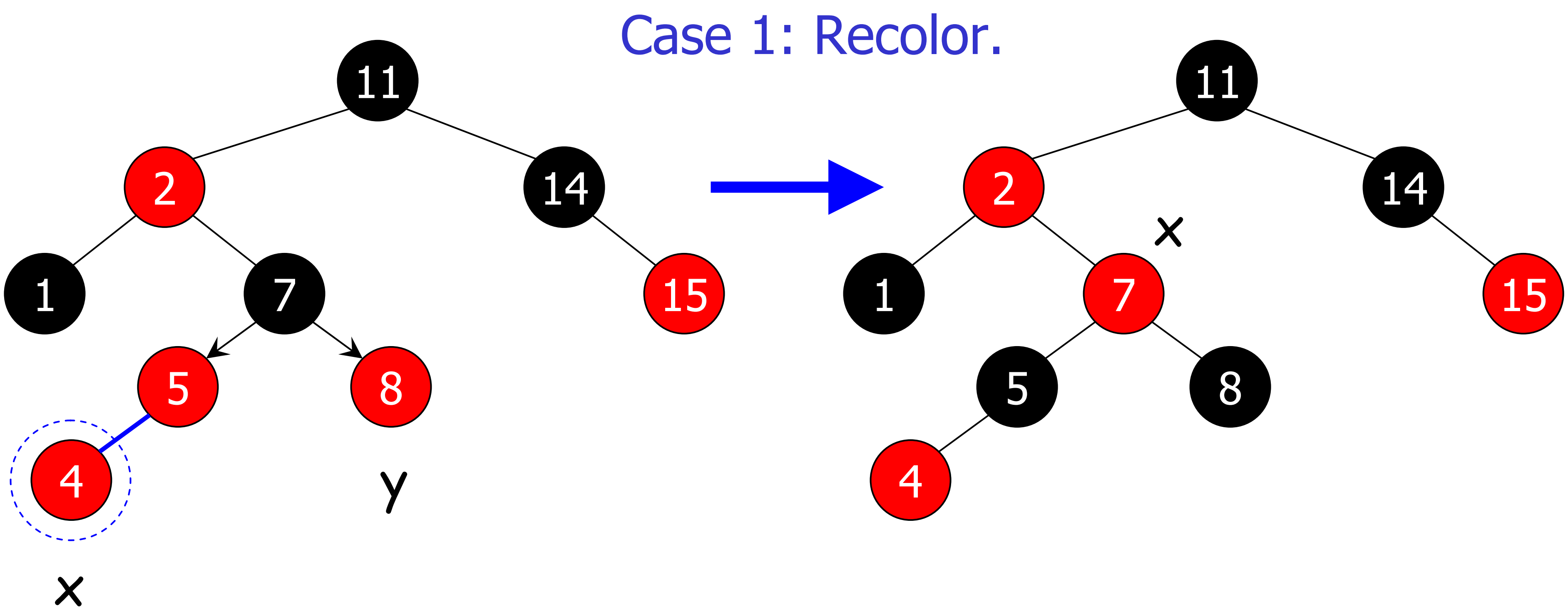
Case 3: Rotate



Right-rotate(C)
(or symmetric
left-rotate).

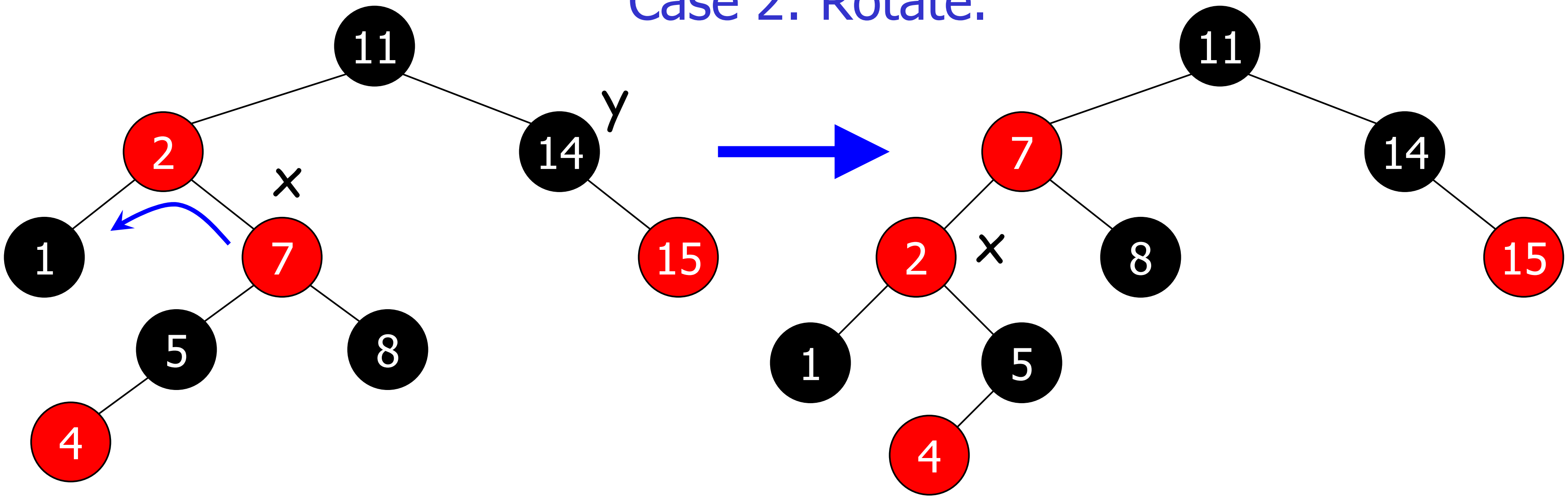
Done!

Example

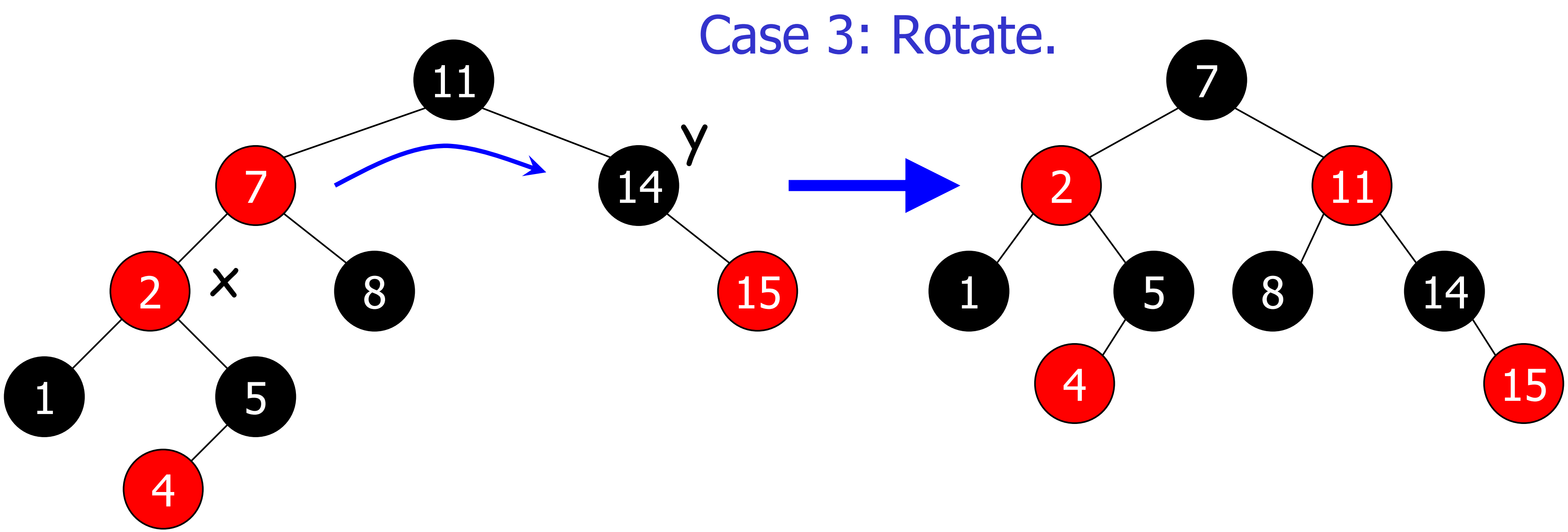


Example

Case 2: Rotate.



Example



Analysis

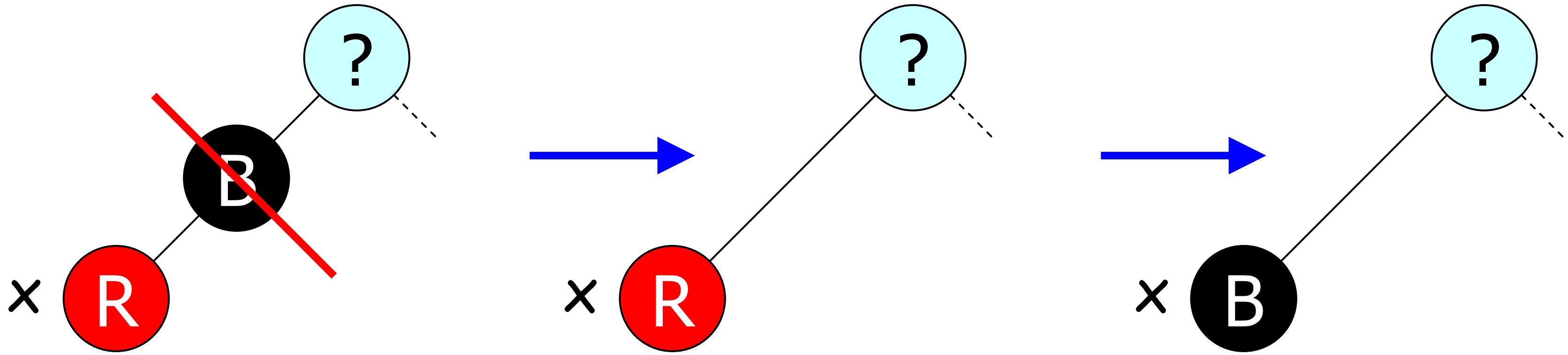
- Case 1 can go up the tree.
- Case 2 performs 2 rotations (incl. case 3).
- Case 3 performs 1 rotation.
- Running time: $O(\lg n)$ with at most 2 rotations.

Deletion

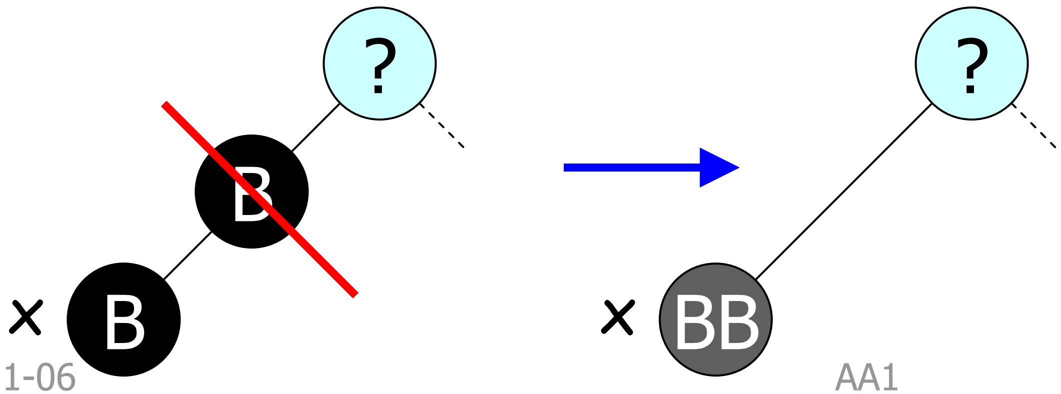
- Binary search tree deletion of a node + fix the red-black tree.
  - Deletion of a red node is easy – nothing more to do.
  - Deletion of a black node: 4 cases.
    The node to be deleted has at most one child.
    Let's call it x.
  - Note: Deleted node here refers to the node removed from the tree – may be different from the original node we wanted to delete, see binary search tree deletion.

Deletion - Start

Trivial: If x is red, color is black and stop.

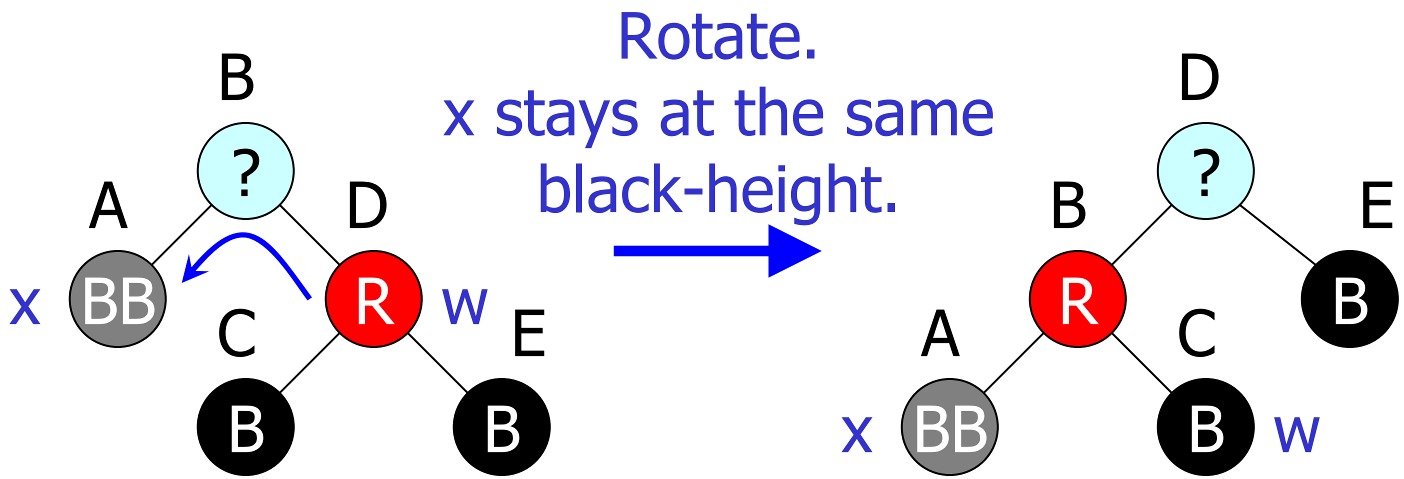


Otherwise x is black, mark it double black.



Deletion – Case 1

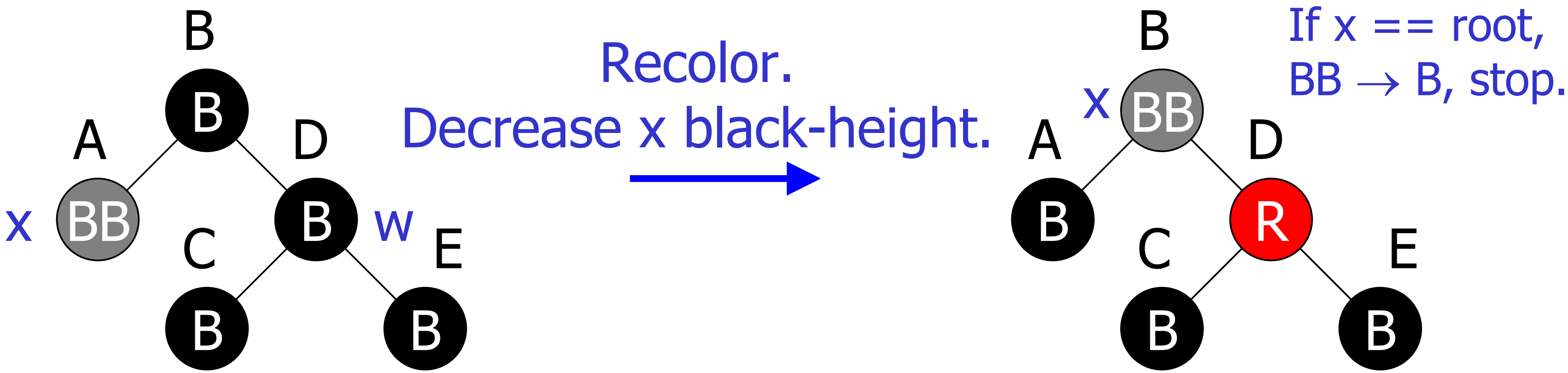
If x's sibling is red.



Case 2b, B will be colored black.

Deletion – Case 2

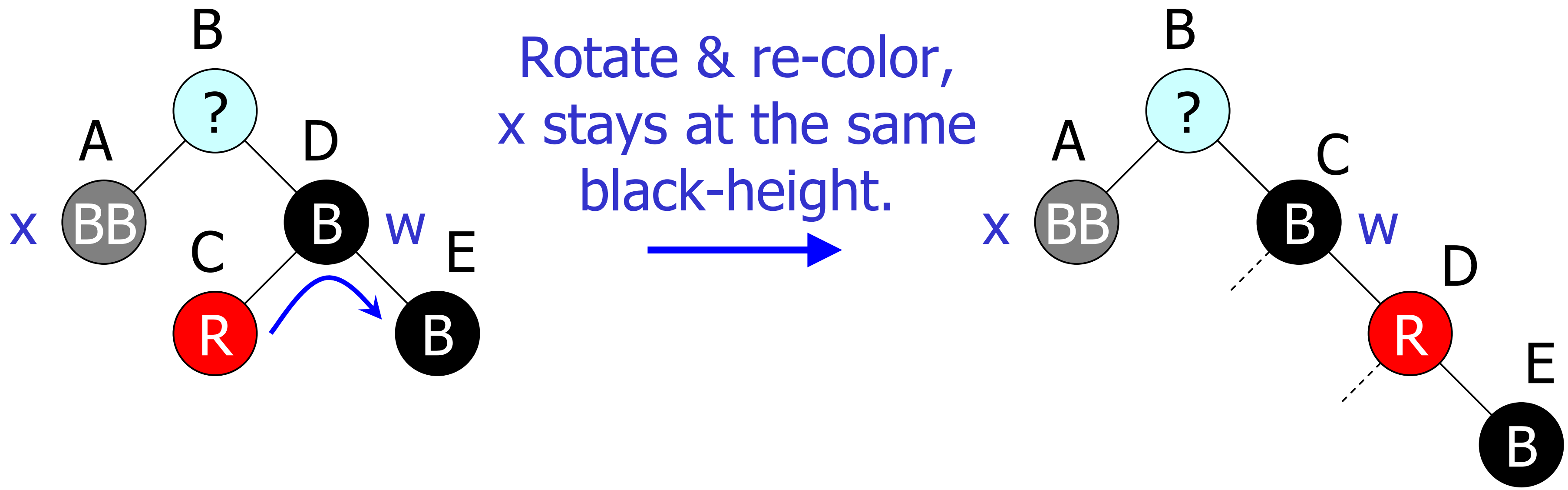
(a) If x's sibling is black and x's parent is black and…



(b) If x's sibling is black and x's parent is red and…

Deletion – Case 3

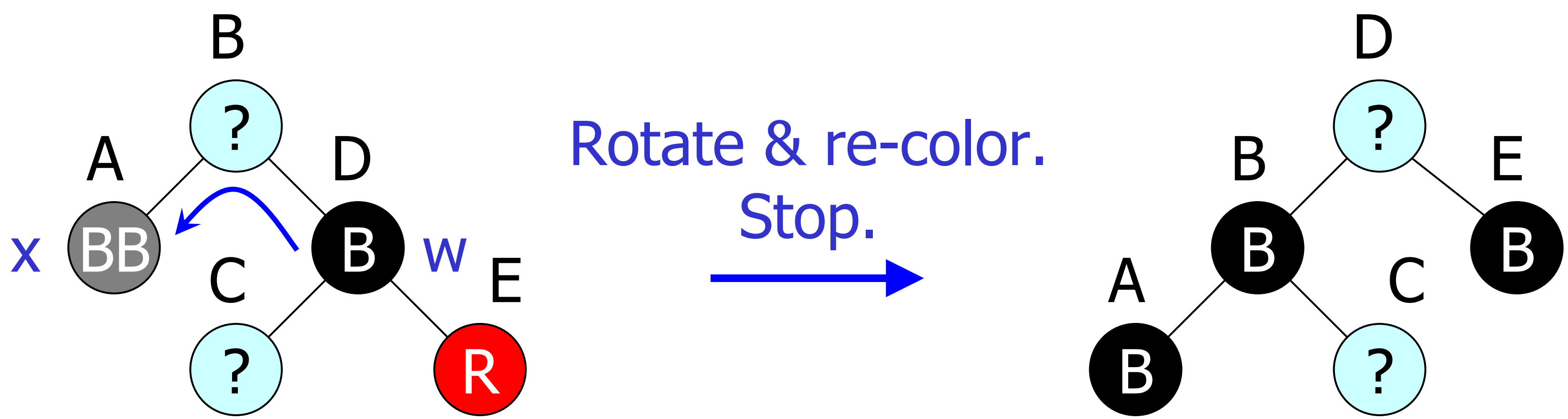
If x's sibling is black and sibling's children are red + black.



Case 4.

Deletion – Case 4

If x's sibling is black and sibling's children are ? + red.



Deletion - Correctness

- We keep the invariant that the tree respects the red-black properties, with special treatment of the black-height (BB counts for 2 B).
- At every step we make progress:
  - Case 1 $\rightarrow$ Case 2b.
  - Case 2a $\rightarrow$ x goes up, recurse $\rightarrow$ will terminate.
  - Case 2b $\rightarrow$ Stop.
  - Case 3 $\rightarrow$ Case 4.
  - Case 4 $\rightarrow$ Stop.
- All configurations are treated.