fit@hcmus

DATA STRUCTURES & ALGORITHMS

Lecture 6: TREES – Part 2

Balanced Binary Search Tree

Lecturer: Dr. Nguyen Hai Minh

CONTENT

- Introduction
- AVL Trees
- Red-Black Trees

Introduction

- ***Balanced search tree***: A search-tree data structure for which a height of $O(\log_2 n)$ is guaranteed when implementing a dynamic set of n items.
- Examples:
  - AVL trees
  - 2-3 trees
  - 2-3-4 trees
  - B-trees
  - Red-black trees

AVL TREES

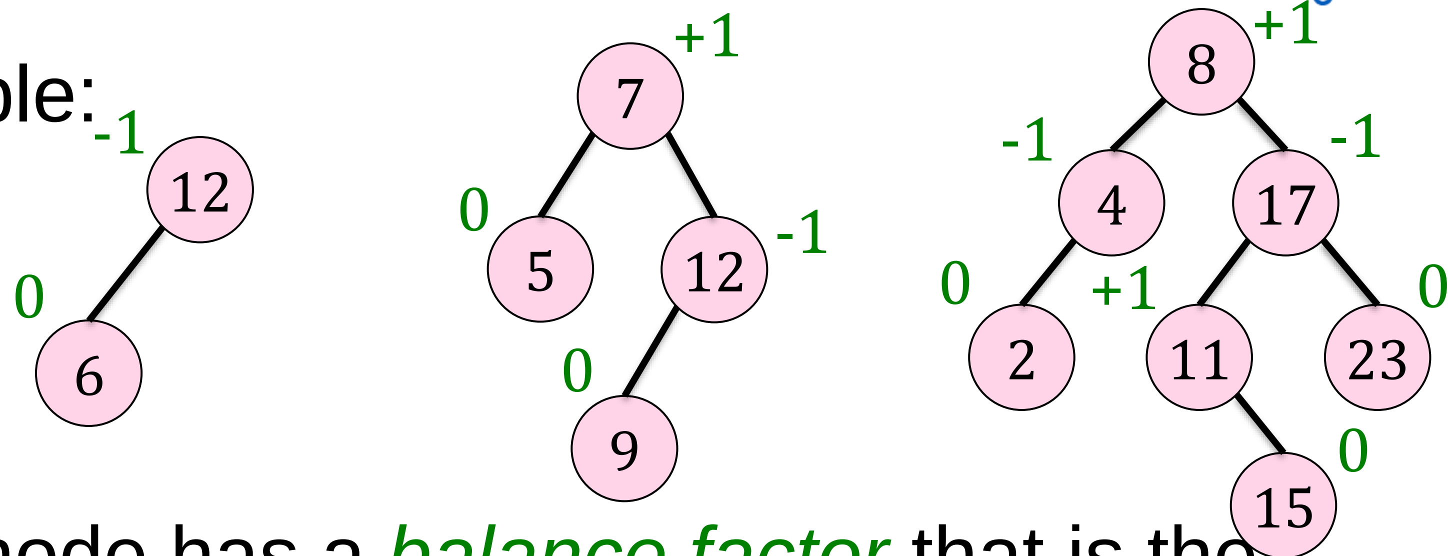
- The Tree ADT
- Tree Traversal

AVL trees

- Proposed by **A**del'son-**V**el'skii and **L**andis in 1962.
- An AVL tree (originally called an admissible tree) is one in which the height of the left and right subtrees of every node differ by at most one.

AVL trees

- Example:



- Each node has a *balance factor* that is the difference between the heights of the left and right subtrees.
  - A balance factor is the height of the right subtree minus the height of the left subtree.
  - Values: 0, -1, +1

AVL trees

- Minimum number of nodes in an AVL tree:

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

  - $AVL_0 = 1$
  - $AVL_1 = 2$
- Height of an AVL tree:

$\boxed{O(\log_2 n)}$

$$\log_2(n + 1) \le h < 1.44\log_2(n + 2) - 0.328$$

- **Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\log_2 n)$ time on an AVL tree with n nodes.

Balancing an AVL tree

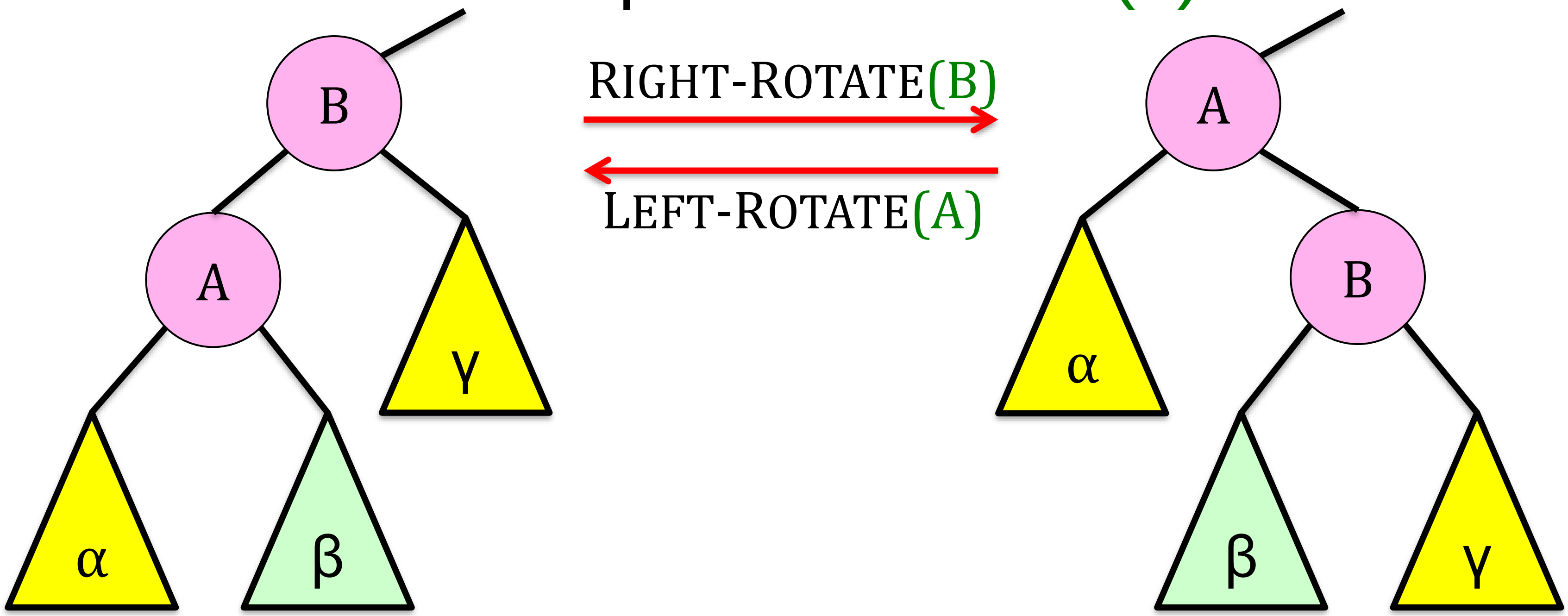
- If balance factor of a node x <-1 or >+1 (when INSERT or DETELE a node from an AVL tree): the tree needs to be balanced.
  - By ***"rotation"*** *(re-balance the subtree rooted with x)*

Balancing an AVL tree

- Let *x* be the unbalanced node
- 4 cases:
  1. Left-left unbalanced (LL): RIGHT-ROTATE(*x*)
  2. Left-right unbalanced (LR): LEFT-ROTATE(*x.left*), then RIGHT-ROTATE(*x*)
  3. Right-right unbalanced (RR): LEFT-ROTATE(*x*)
  4. Right-left unbalanced (RL): RIGHT-ROTATE(*x.right*), then LEFT-ROTATE(*x*)

Rotations

- Rotations maintain the inorder ordering of keys:
  - $a \in \alpha$, $b \in \beta$, $c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$.
- A rotation can be performed in $O(1)$ time.

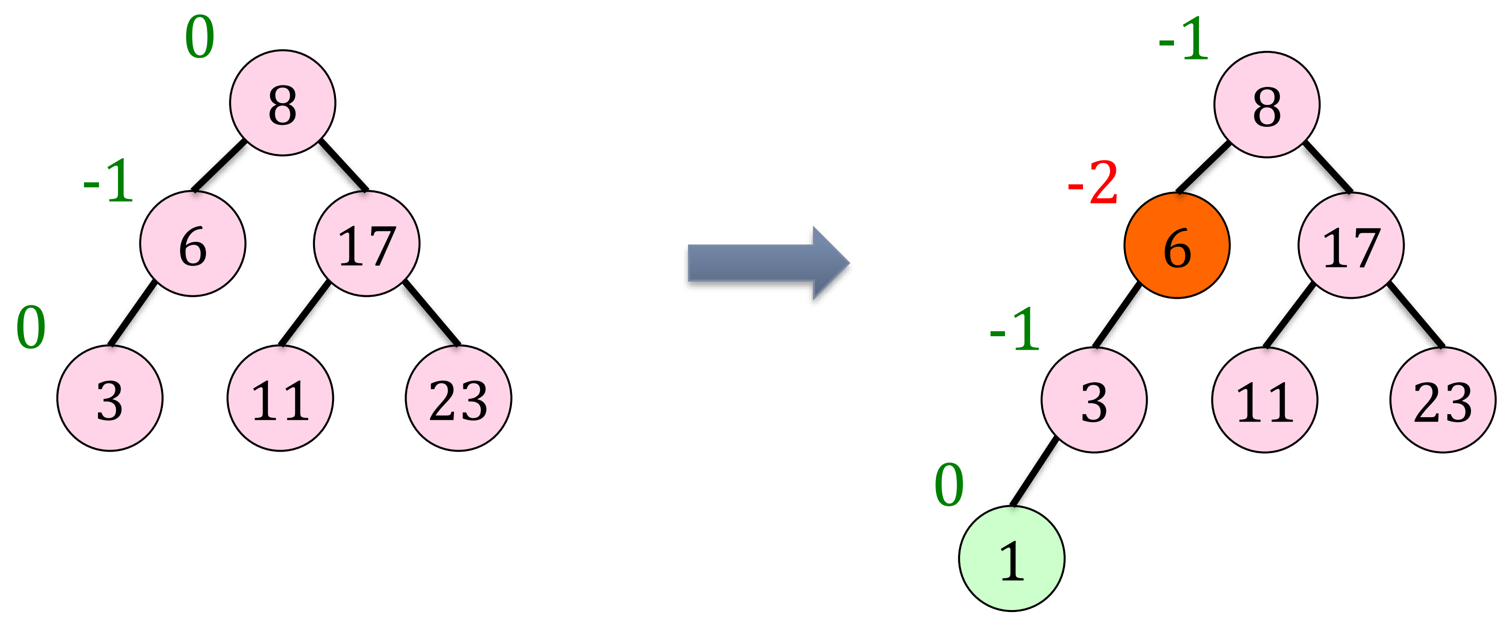


Insertion into an AVL tree

- The INSERT operation is performed in the same with insertion in a binary search tree.
- If INSERT results in an unbalanced tree, perform the appropriate rotation(s) to restore its balance.

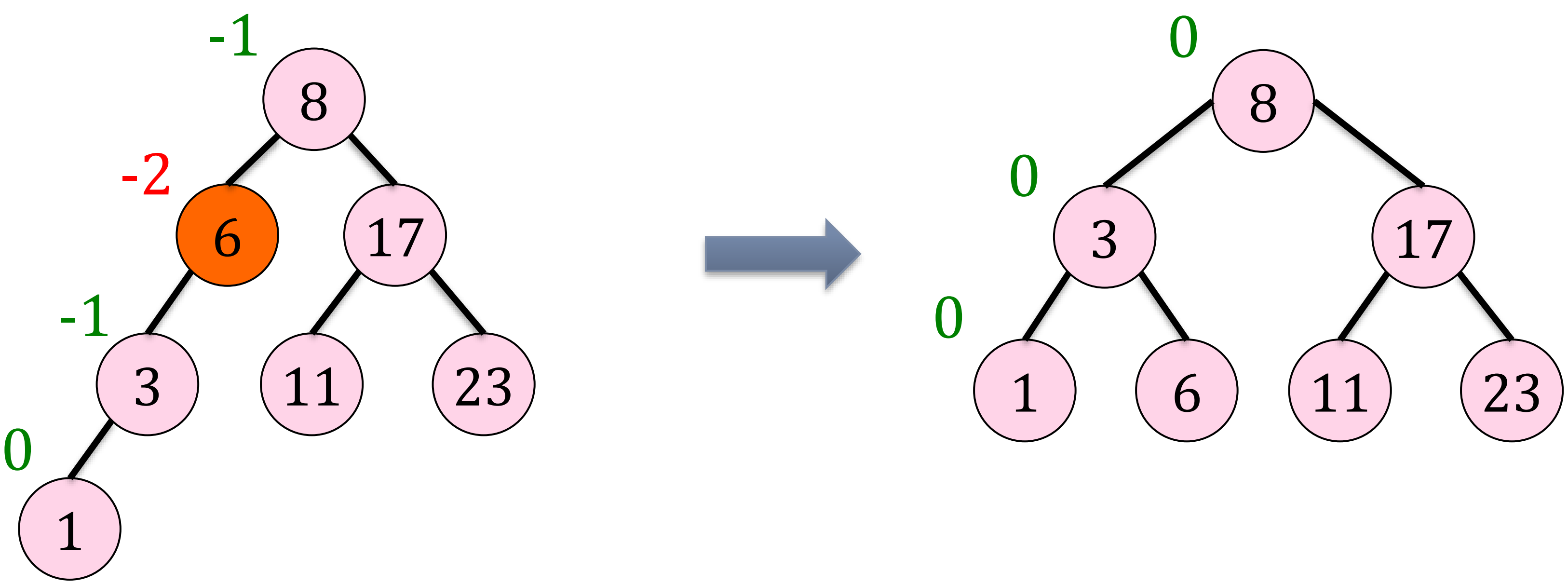
Case 1: Left-left Unbalanced

- Result of insert 1 into the tree



Case 1: Left-left Unbalanced

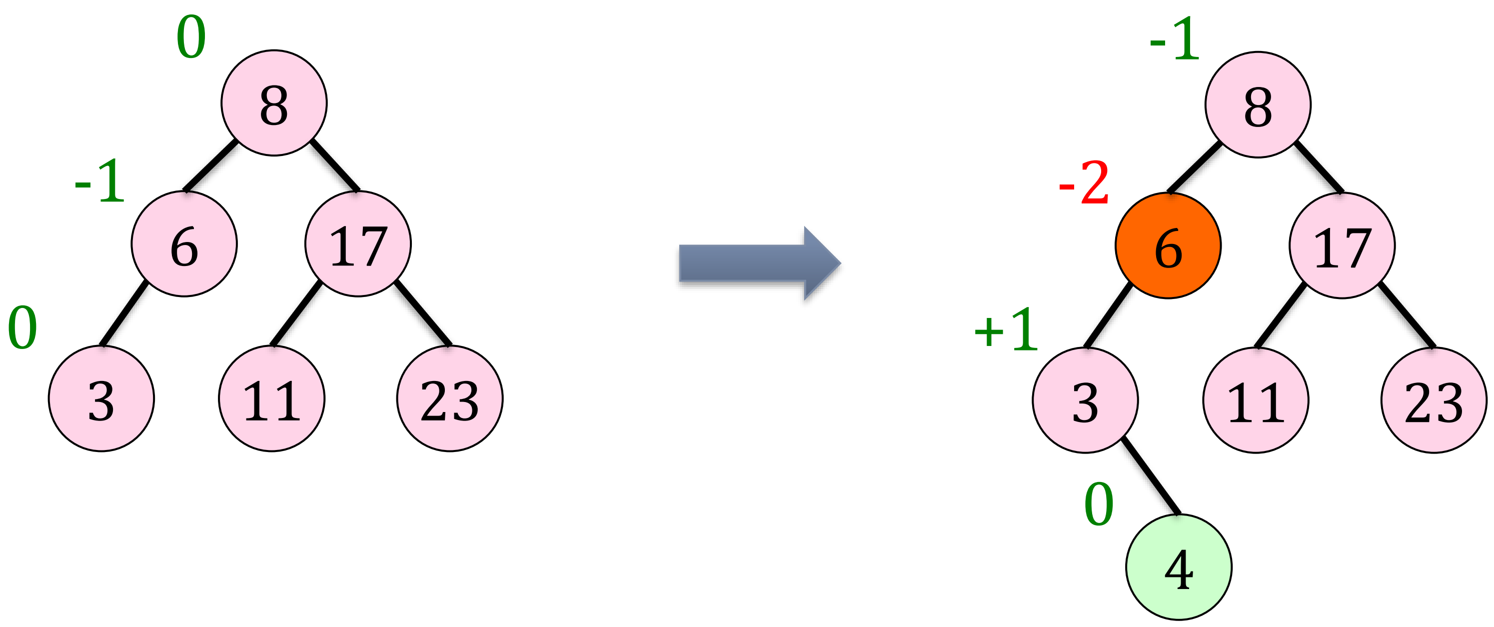
- RIGHT-ROTATE(6)



- Even better than the original tree!

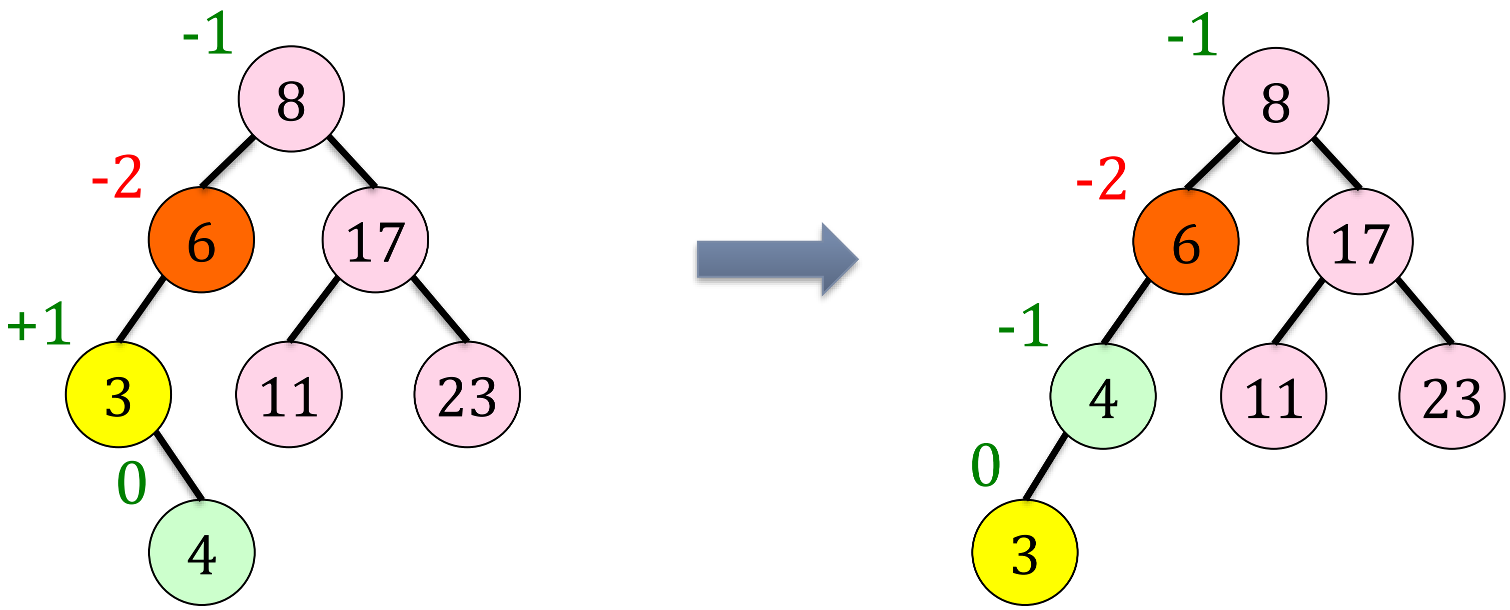
Case 2: Left-right Unbalanced

- Result of insert 4 into the tree



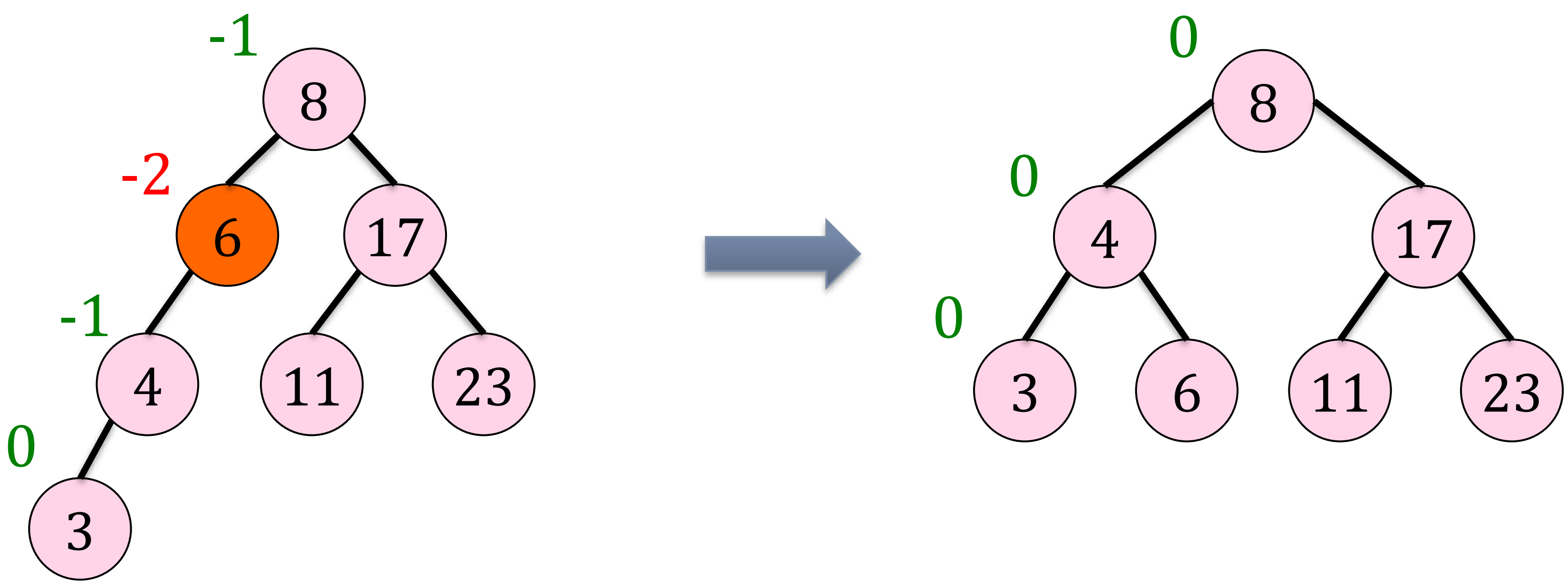
Case 2: Left-right Unbalanced

- LEFT-ROTATE(3) → Transformed to case 1 (LL)



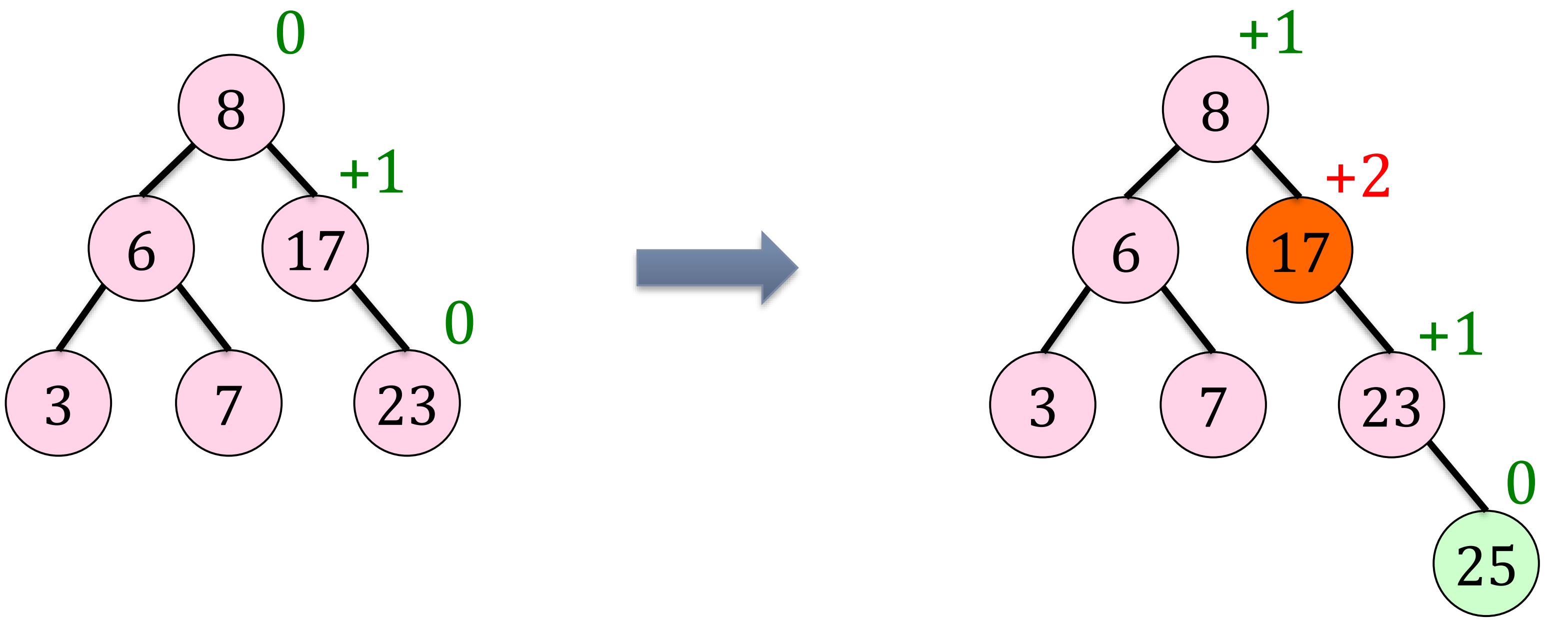
Case 2: Left-right Unbalanced

- RIGHT-ROTATE(6)



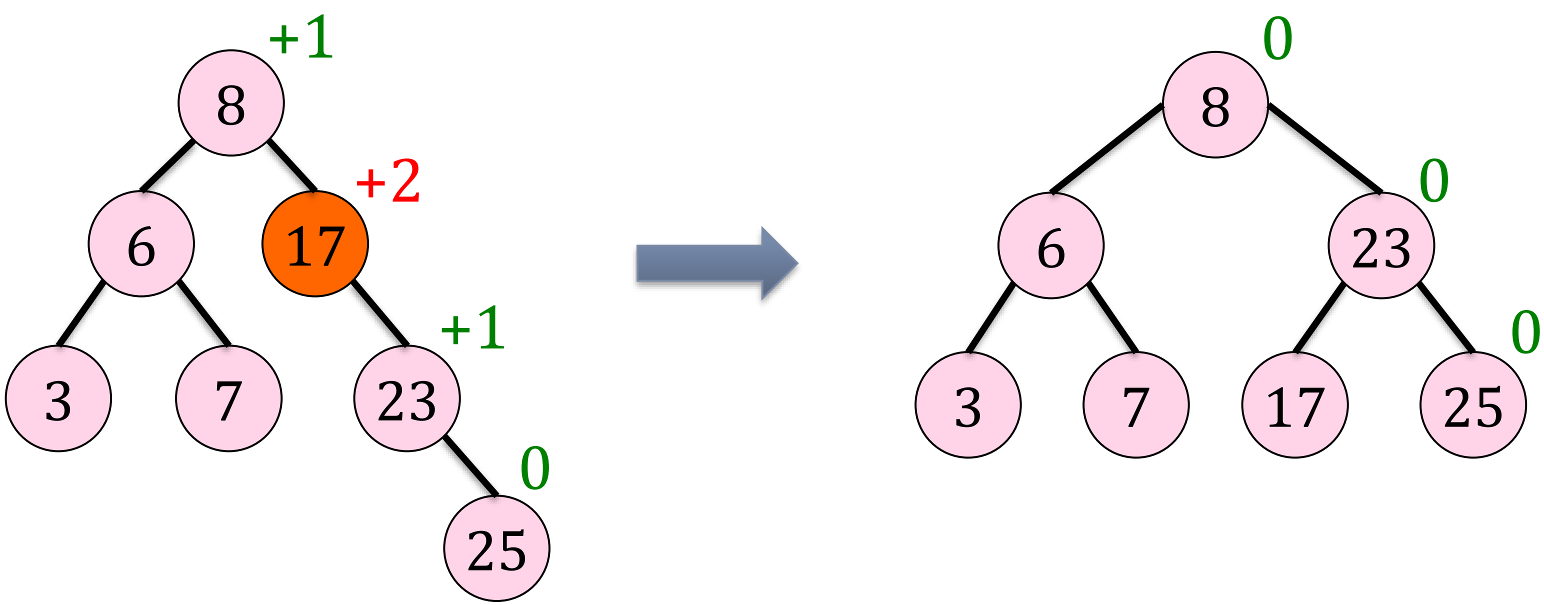
Case 3: Right-right Unbalanced

- Result of insert 25 into the tree



Case 3: Right-right Unbalanced

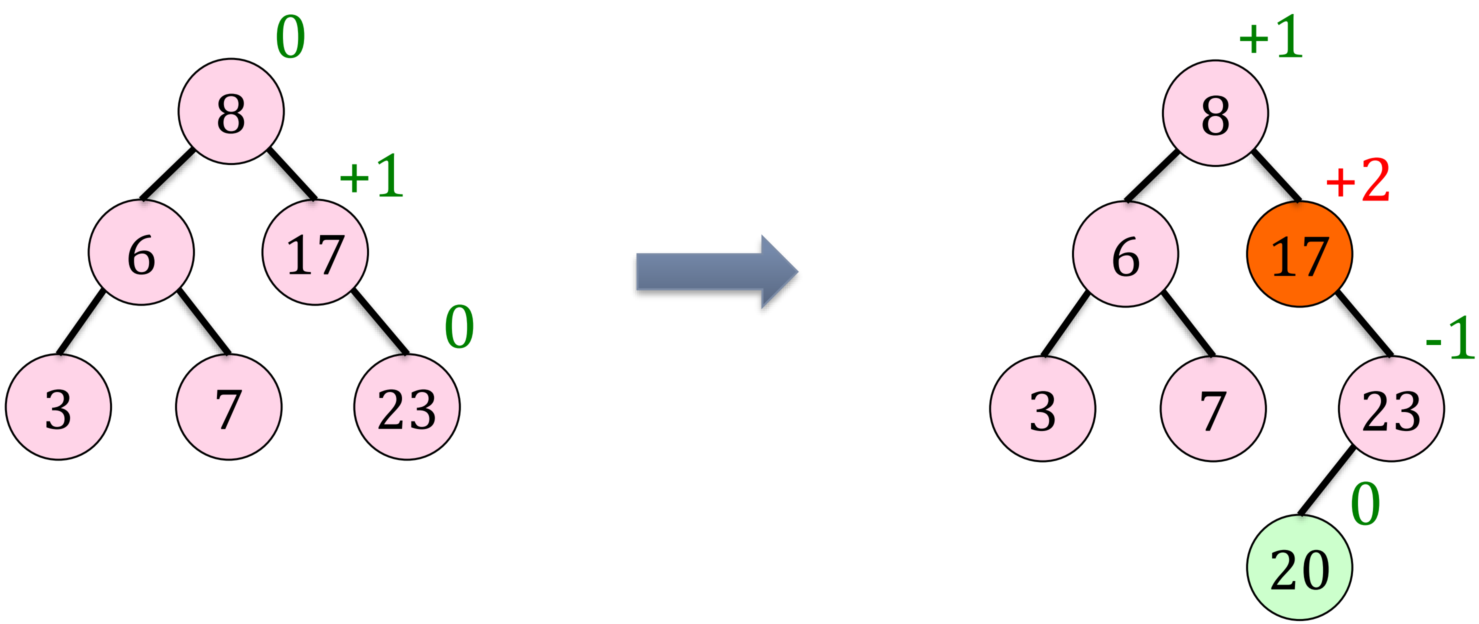
- LEFT-ROTATE(17)



- Even better than the original tree!

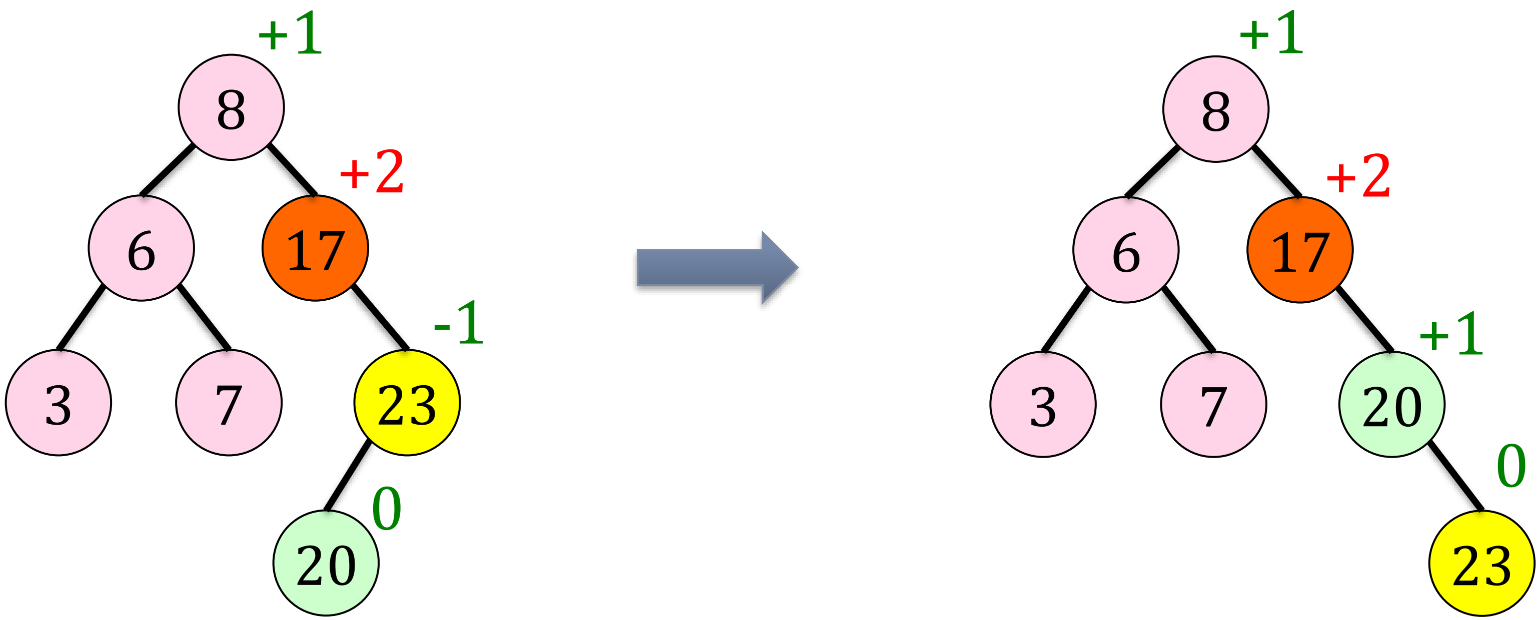
Case 4: Right-left Unbalanced

- Result of insert 20 into the tree



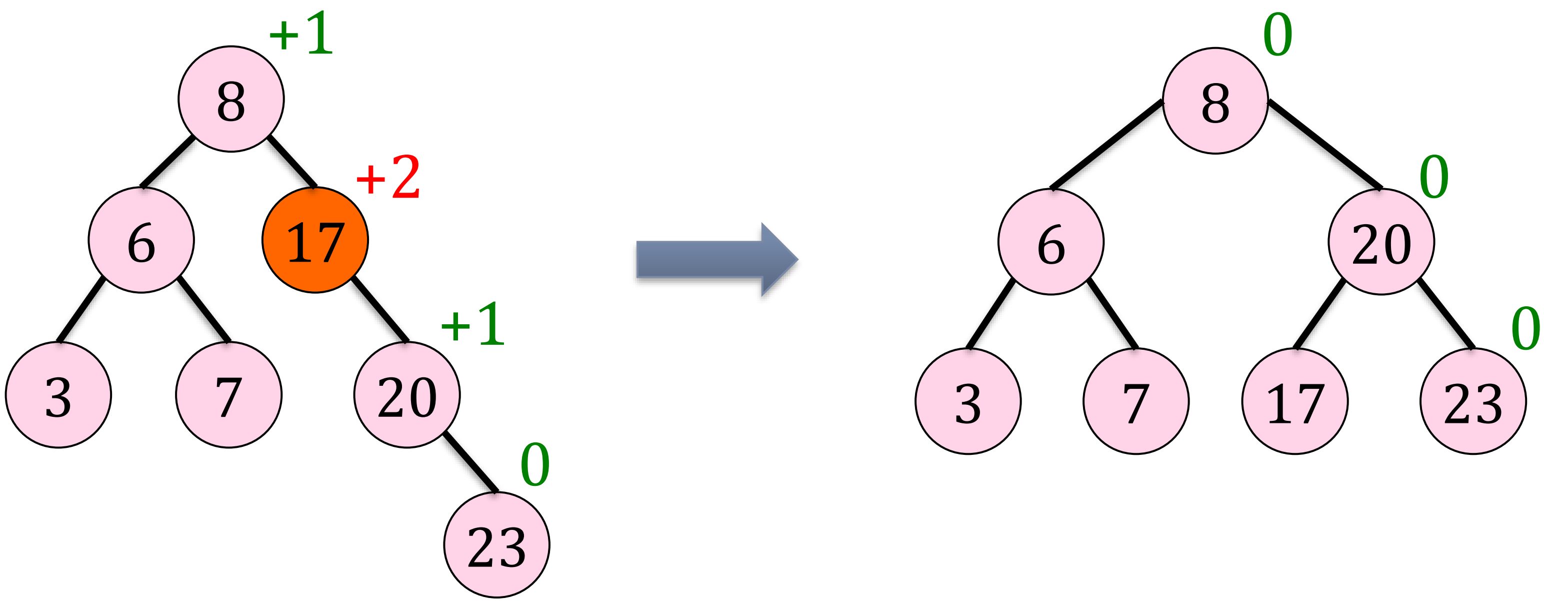
Case 4: Right-left Unbalanced

- RIGHT-ROTATE(23) → Transformed to case 3 (RR)



Case 4: Right-left Unbalanced

- LEFT-ROTATE(17)

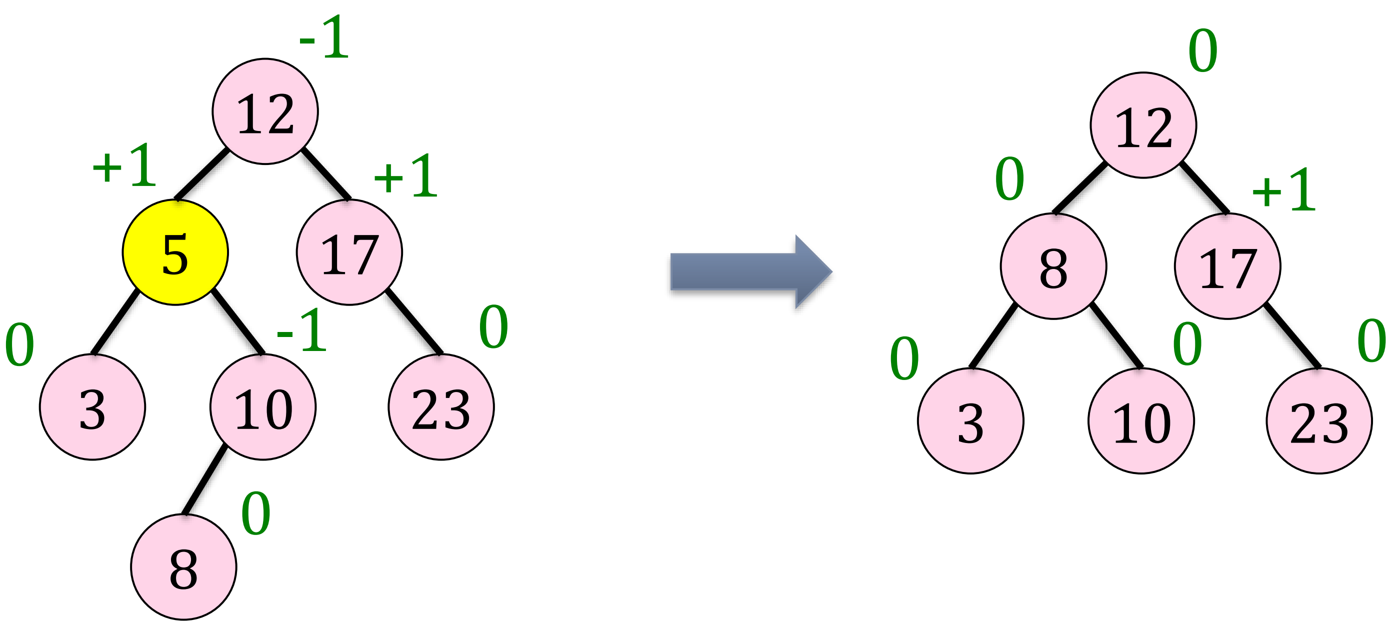


Deletion from an AVL tree

- The DELETE operation is performed in the same with deletion in a binary search tree.
- After a node *x* is deleted:
  - Balance factors are updated from the parent of *x* up to the root.
  - For **each node in this path** whose balance factor becomes +2/-2, perform the appropriate rotation(s) to restore the balance of the tree.
  - Notice that the rebalancing does not stop after the first unbalanced node is rotated.

Deletion without balancing

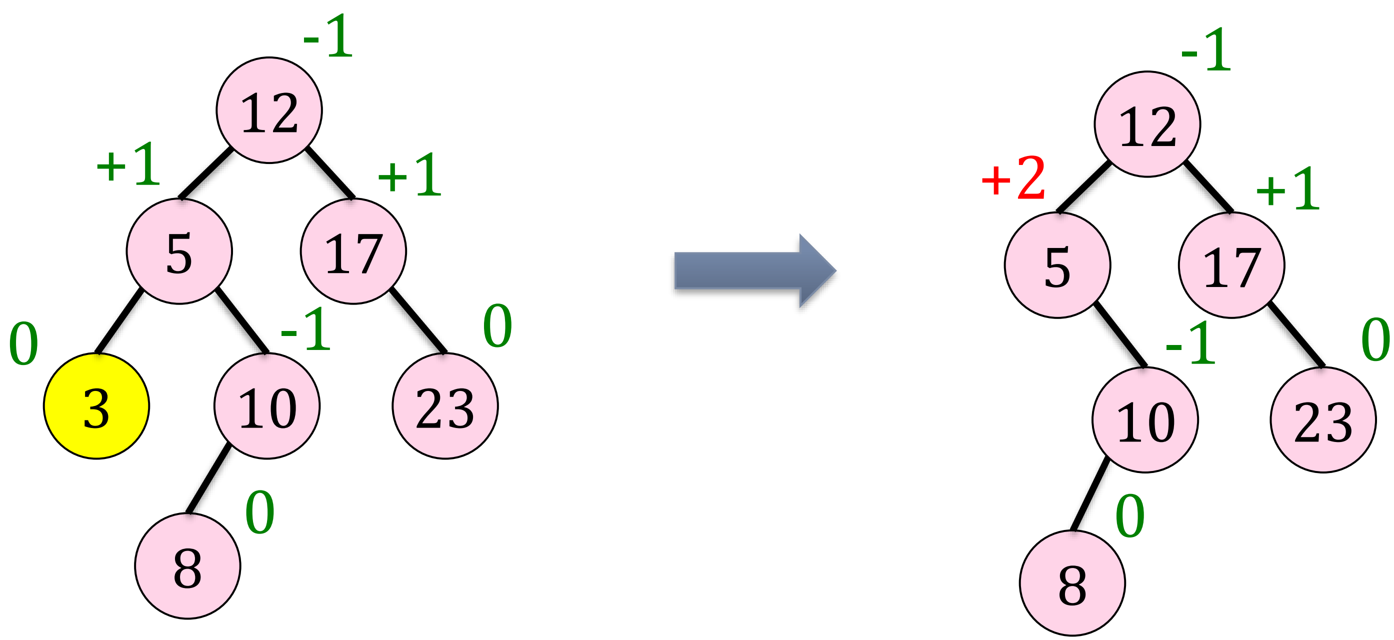
- Delete 5 → Replace it by 8



- The tree is still in balance → no rotation

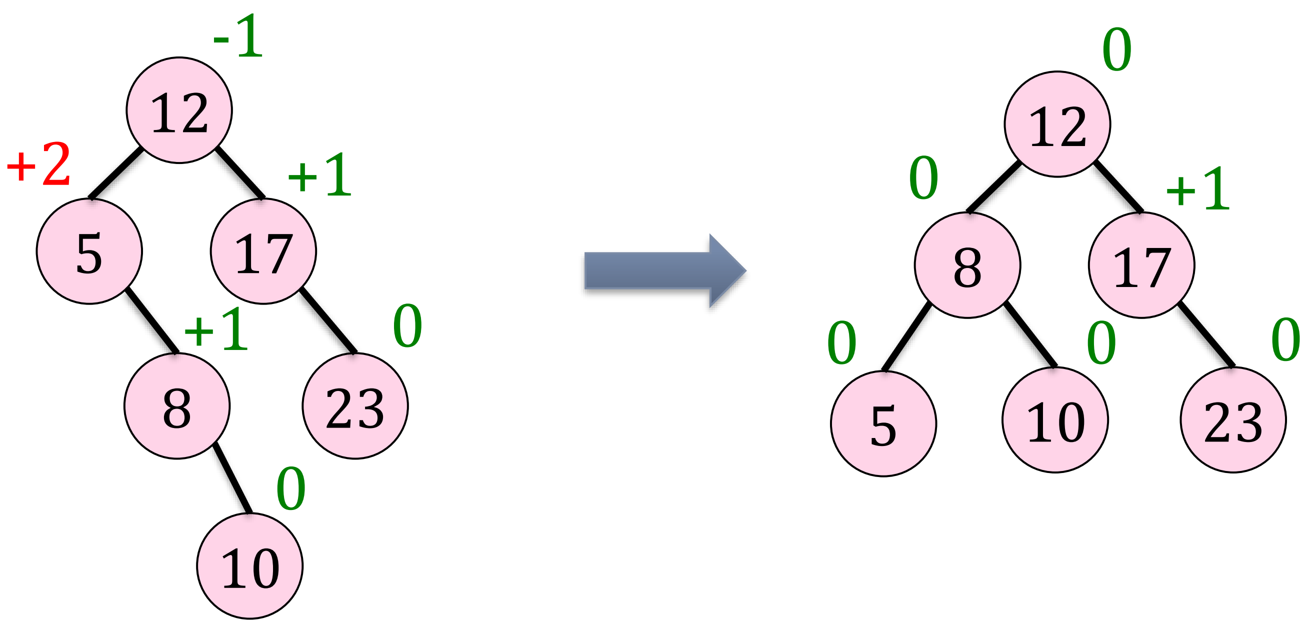
Deletion that needs balancing

- Delete 3 → RL unbalanced → case 4 (RL)



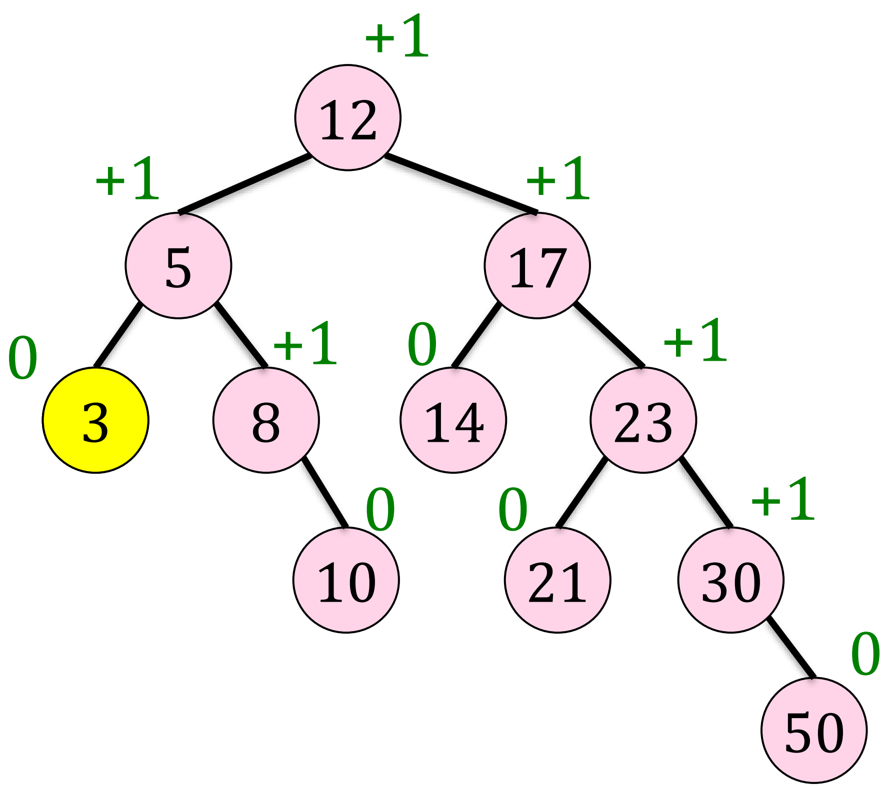
Deletion that needs balancing

- RIGHT-ROTATE(10), then LEFT-ROTATE(5)



Worst case of deletion

- What happens when we delete 3?



Analysis

- INSERT:
  - Find a place to insert: O(*h*).
  - Perform 1 or 2 rotations.

$O(\log_2 n)$

- DELETE:
  - Find a replaced node: O(*h*).
  - Perform rotations on the path from the deleted node up to the root: O(*h*).

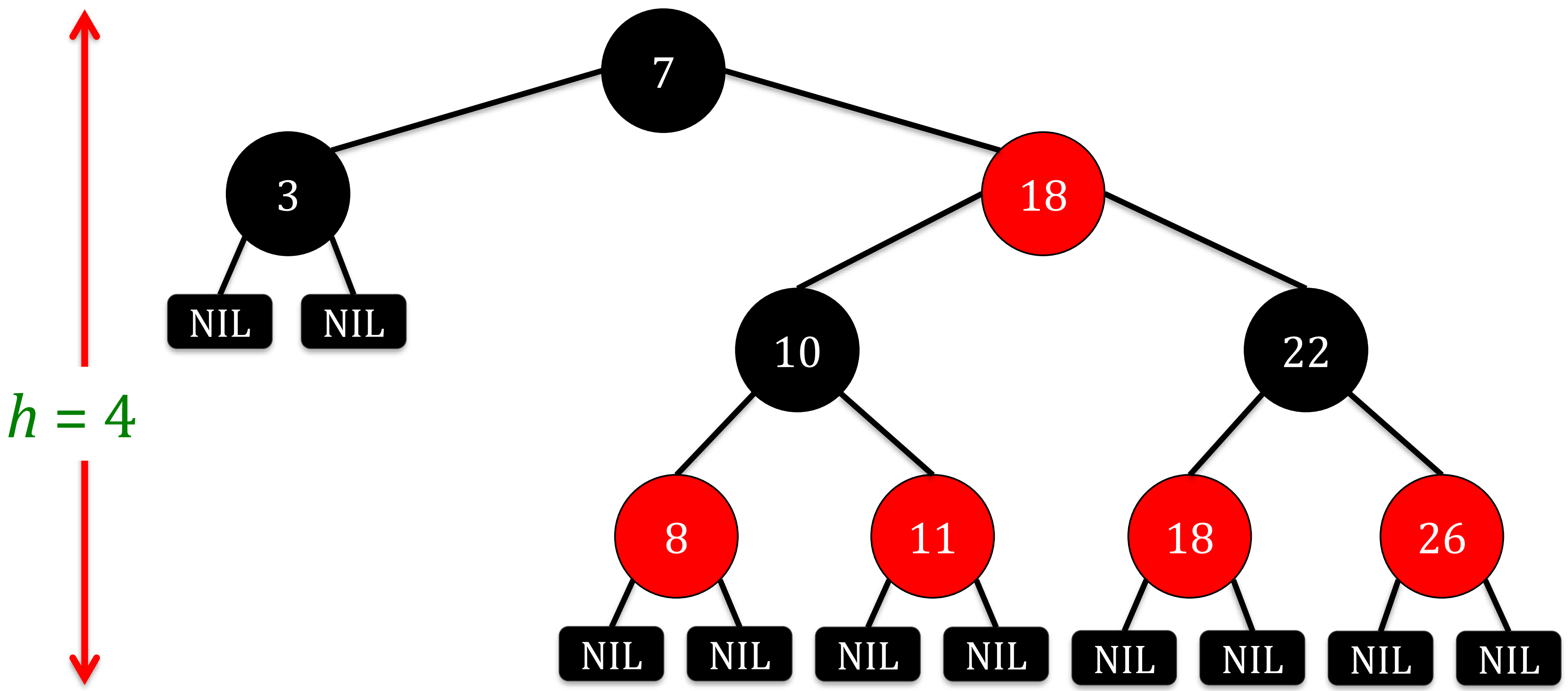
$O(\log_2 n)$

RED-BLACK TREES

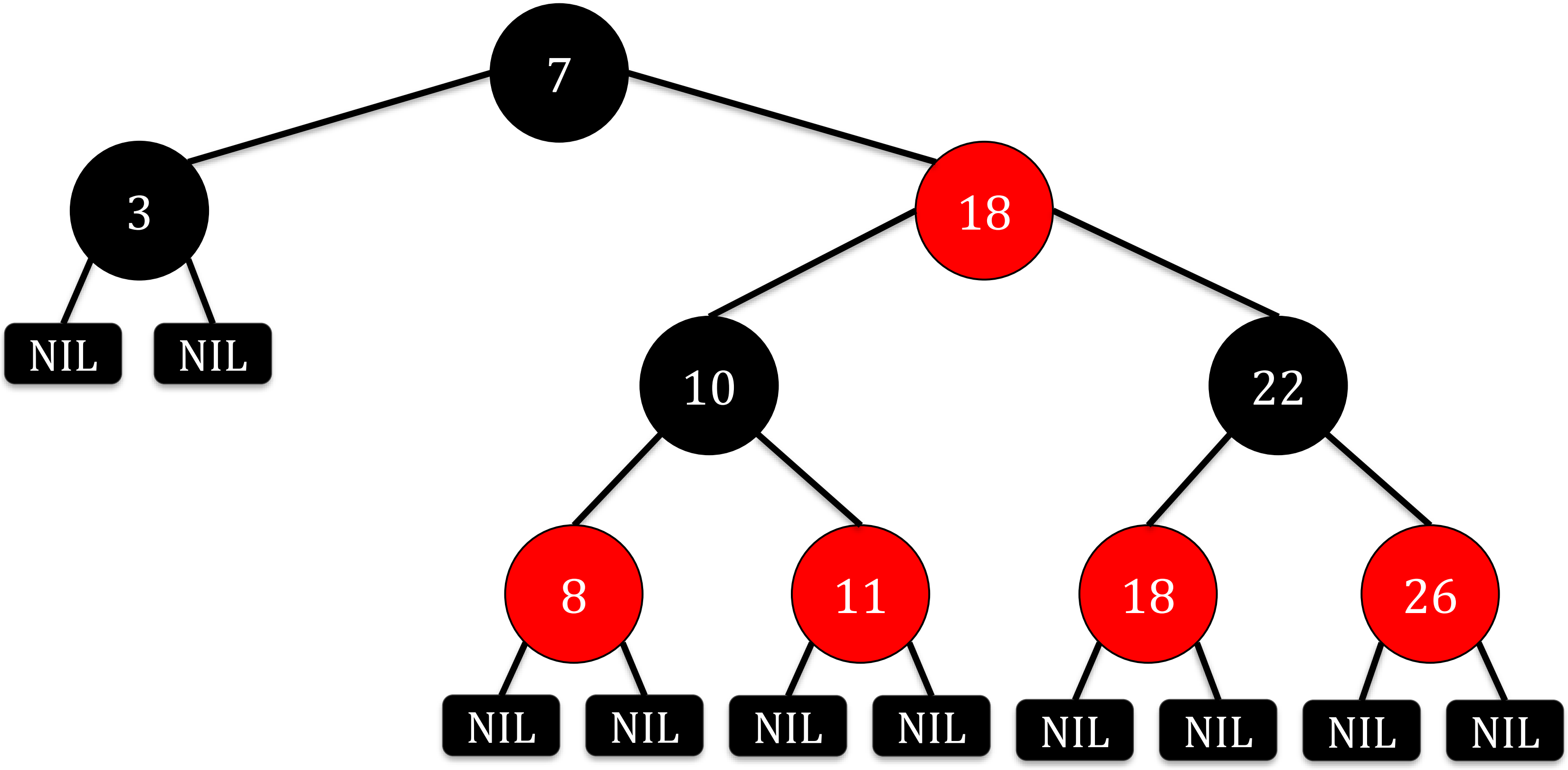
Red-black trees

- Red-black tree (proposed by Rudolf Bayer in 1972) is a kind of self-balancing binary search tree. Each node of the tree has an extra bit as color field.
- Red-black properties:
  1. Every node is either red or black.
  2. The root is black.
  3. The leaves (NIL's) are black.
  4. If a node is red, then both of its children are black.
  5. All simple paths from any node *x* to a descendent leaf have the same number of black nodes = *black-height(x)*

Example of a red-black tree

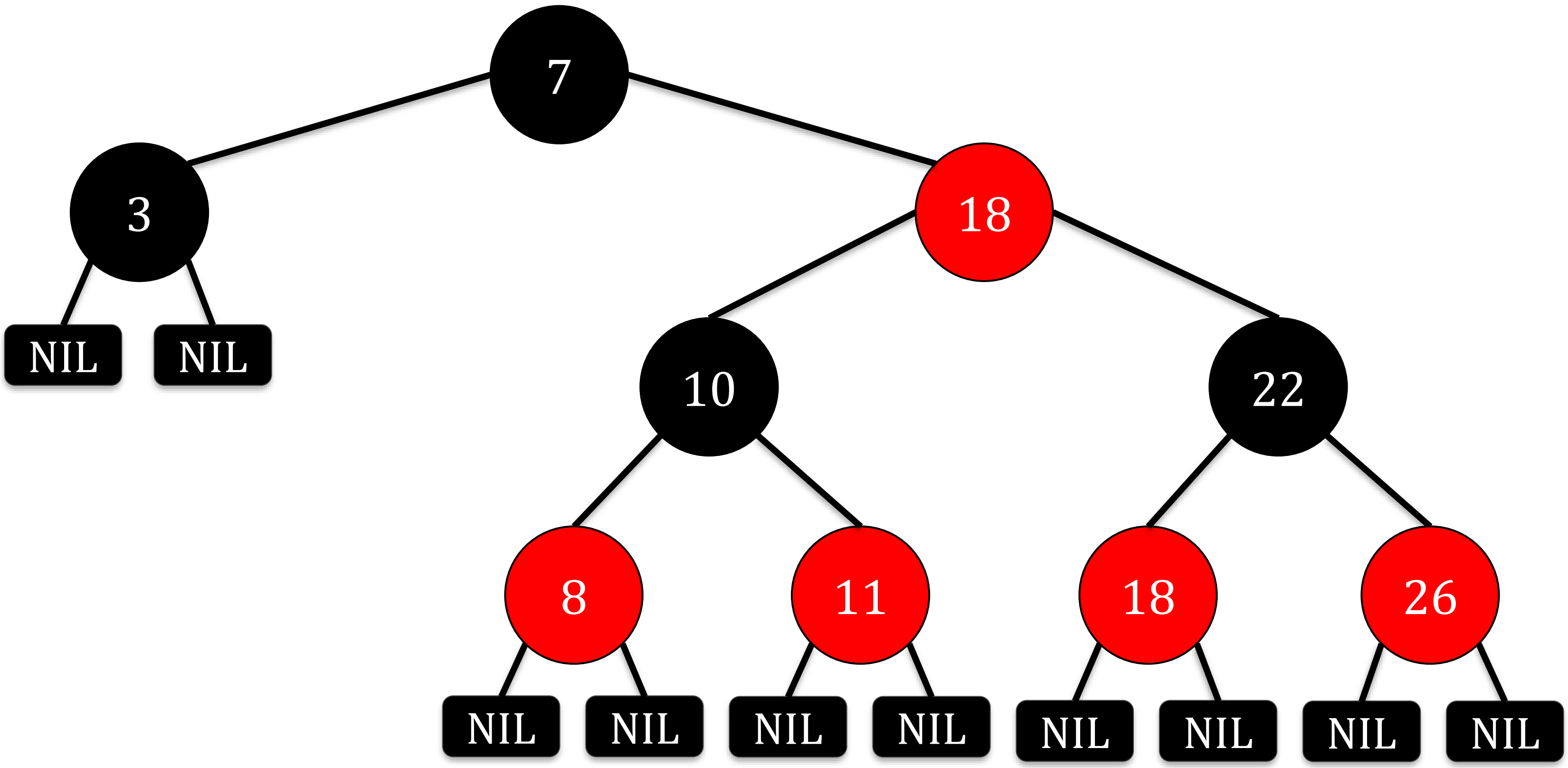


Example of a red-black tree



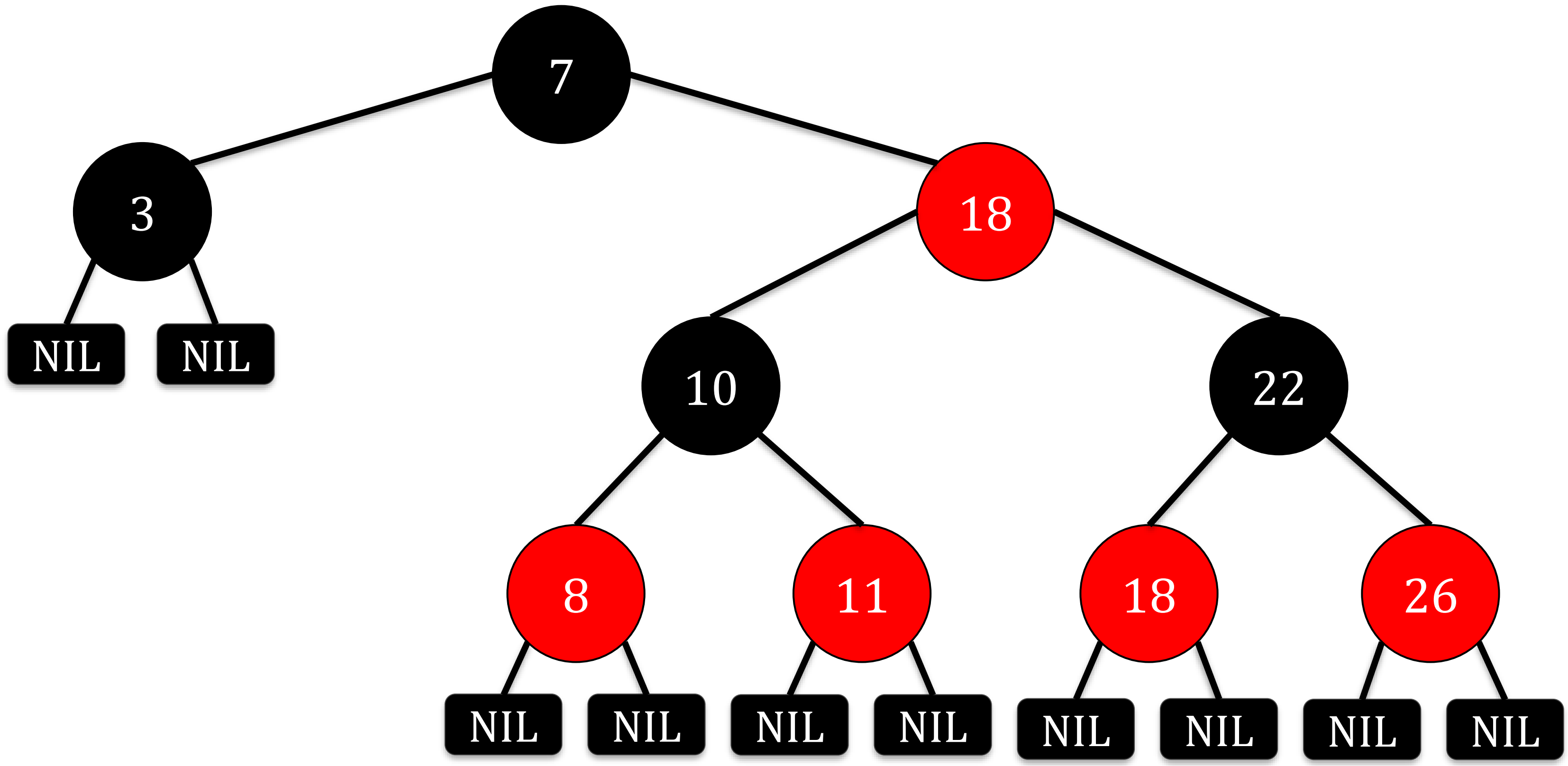
1. Every node is either red or black.

Example of a red-black tree



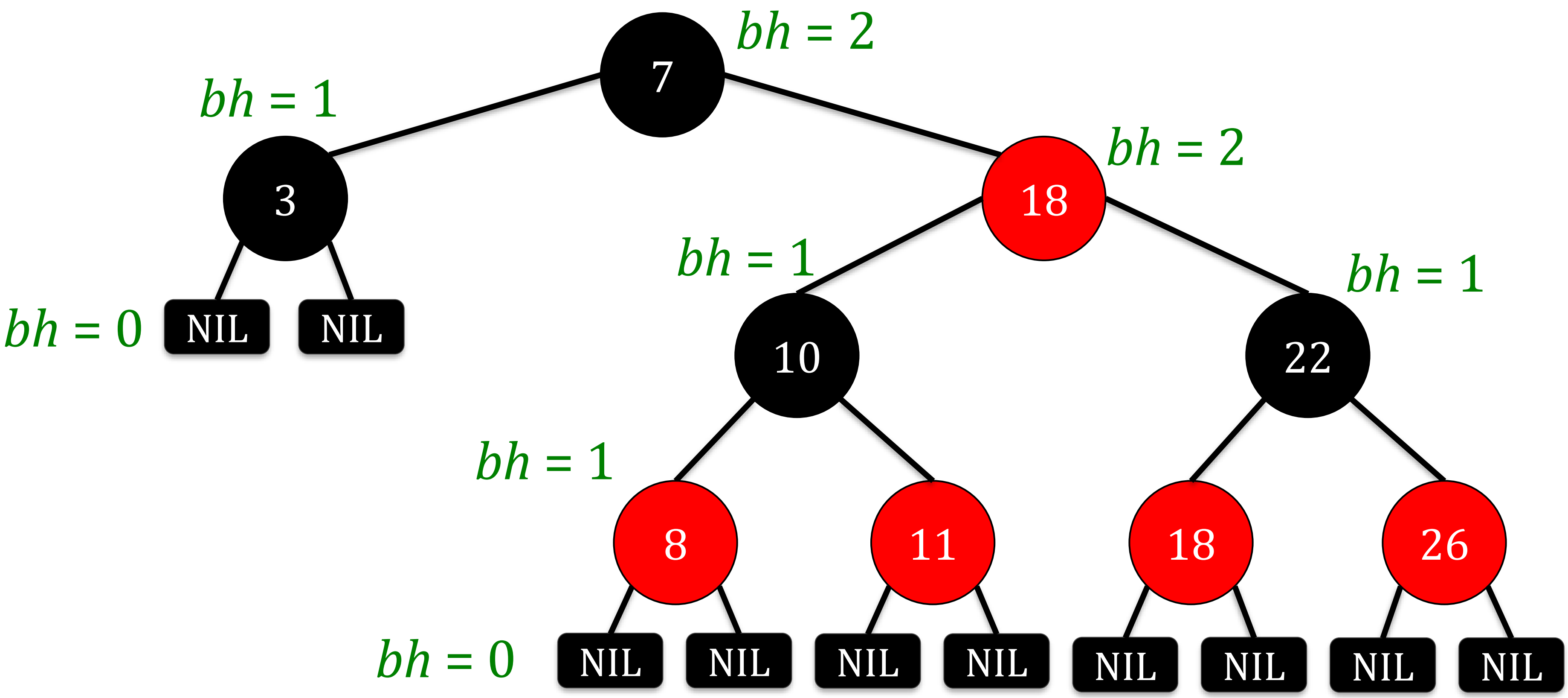
2.+3. The root and leaves (NIL's) are black.

Example of a red-black tree



4. If a node is red, then both its children are black.

Example of a red-black tree



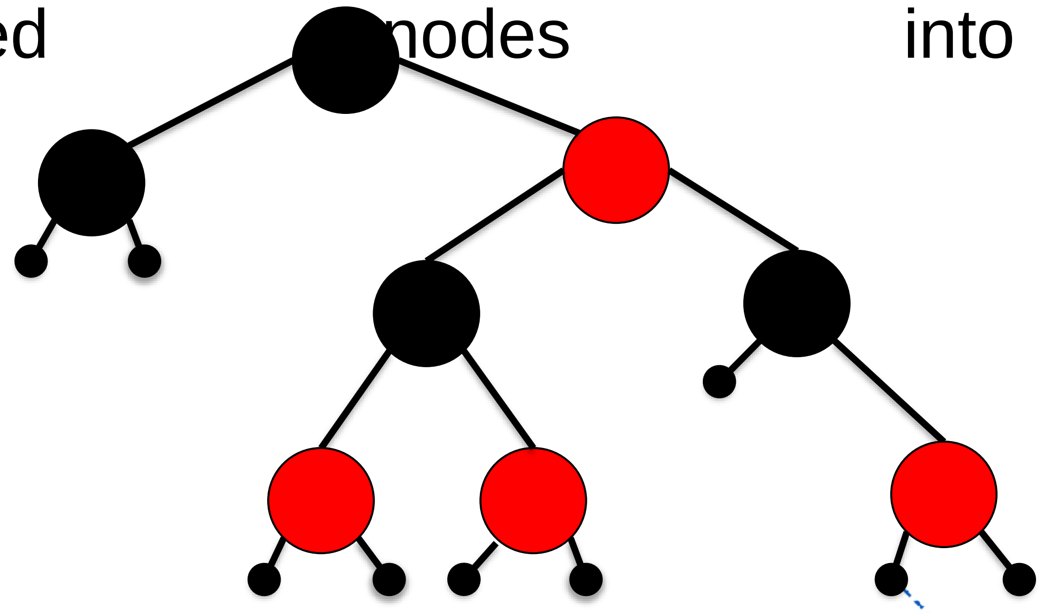
5. All simple paths from any node *x* to a descendent leaf have the same number of black nodes = *black-height(x)*

Height of a red-black tree

- **Theorem**. A red-black tree with n keys has height

$$h \le 2\log_2(n+1)$$

- *Proof.* INTUITION:
  - Merge red nodes into their black parents

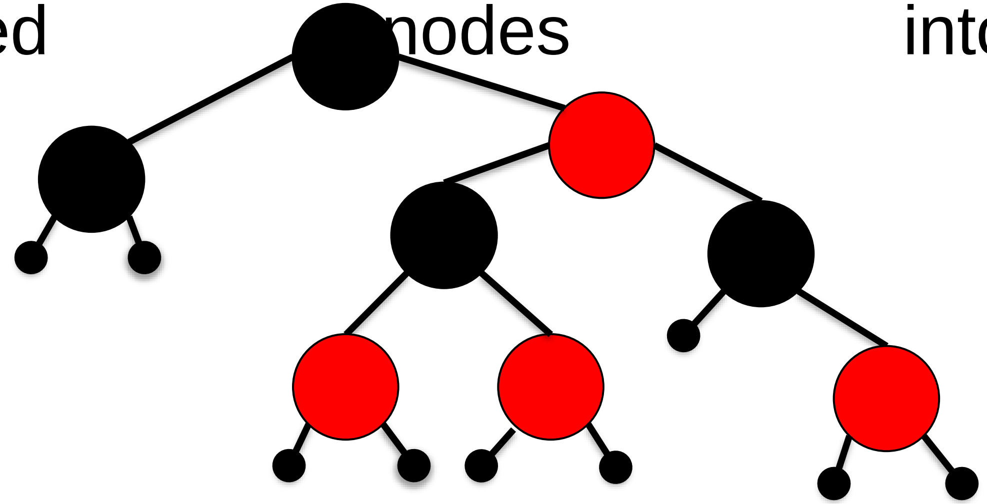


Height of a red-black tree

- **Theorem**. A red-black tree with n keys has height

$$h \le 2 \log_2(n+1)$$

- *Proof.* INTUITION:
  - Merge red nodes into their black parents

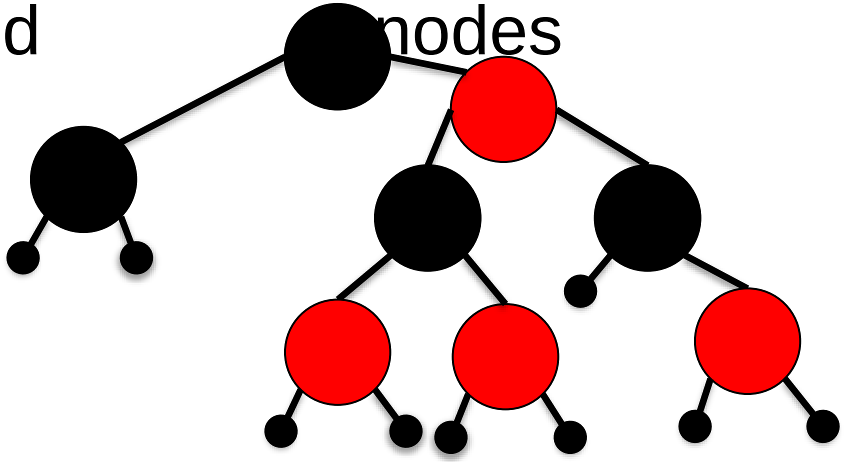


Height of a red-black tree

- **Theorem**. A red-black tree with n keys has height

$$h \leq 2\log_2(n+1)$$

- *Proof.* INTUITION:
  - Merge red nodes into their black parents

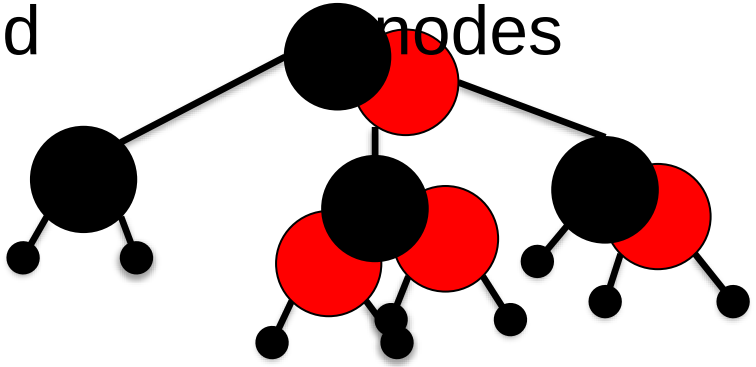


Height of a red-black tree

- **Theorem**. A red-black tree with n keys has height

$$h \le 2 \log_2(n+1)$$

- *Proof.* INTUITION:
  - Merge red nodes into their black parents

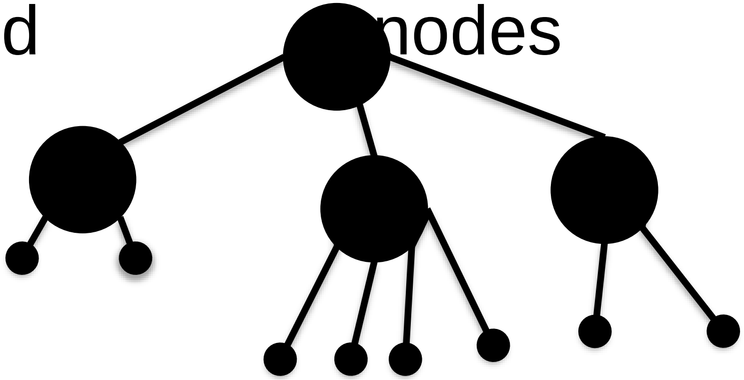


Height of a red-black tree

- **Theorem**. A red-black tree with n keys has height

$$h \le 2 \log_2(n+1)$$

- *Proof.* INTUITION:
  - Merge red nodes into their black parents

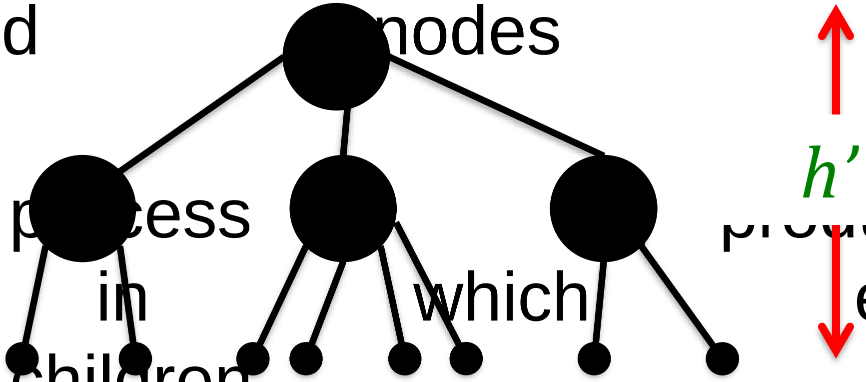


Height of a red-black tree

- **Theorem**. A red-black tree with n keys has height

$$h \leq 2 \log_2(n+1)$$

- *Proof.* INTUITION:
  - Merge red nodes into their black parents
  - This process produces a tree in which each node has 2, 3, or 4 children.
  - The 2-3-4 tree has uniform depth *h'* of leaves.



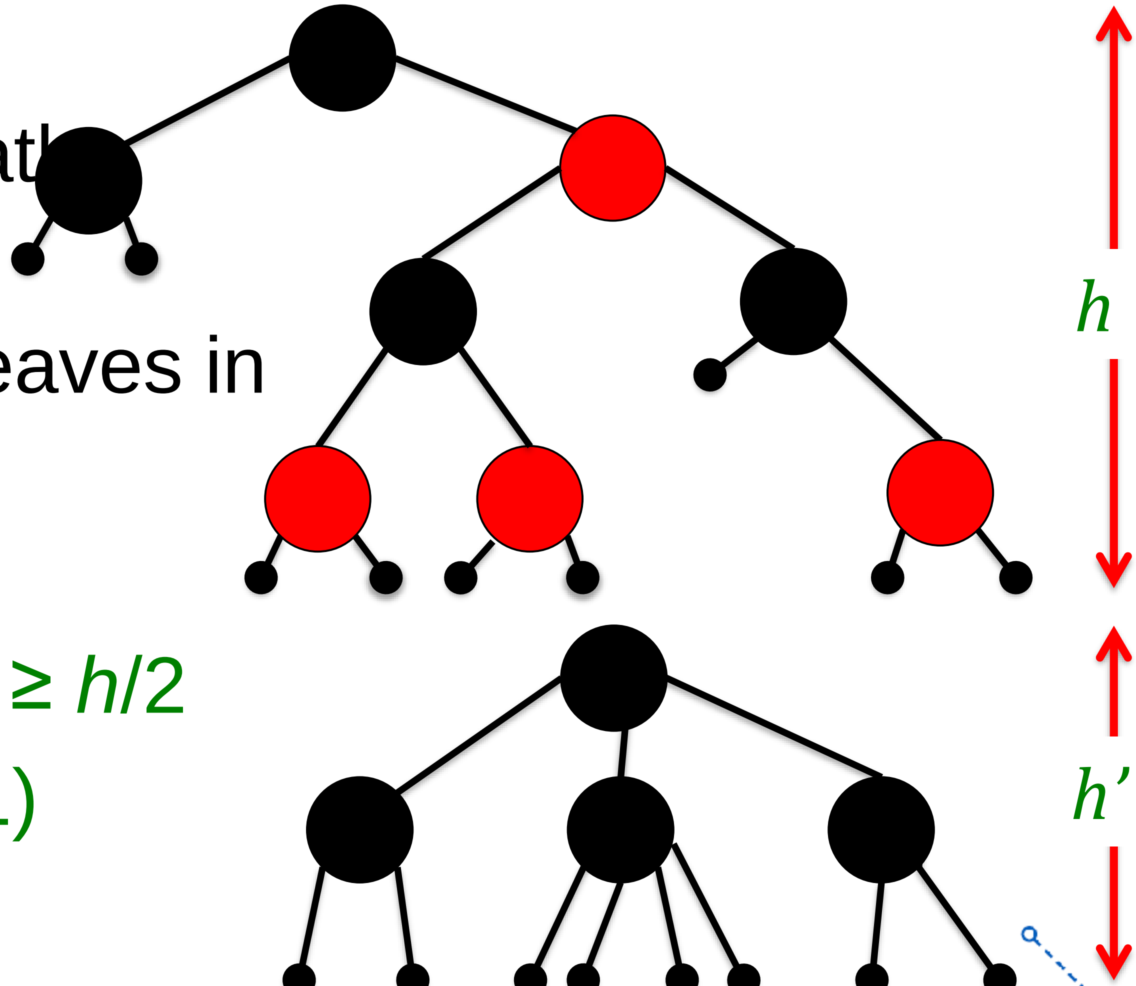
Proof (continued)

- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in each tree is $n+1$

→ $n + 1 \geq 2^{h'}$

→ $\log_2(n + 1) \geq h' \geq h/2$

→ $h \leq 2 \log_2(n + 1)$



Query operations

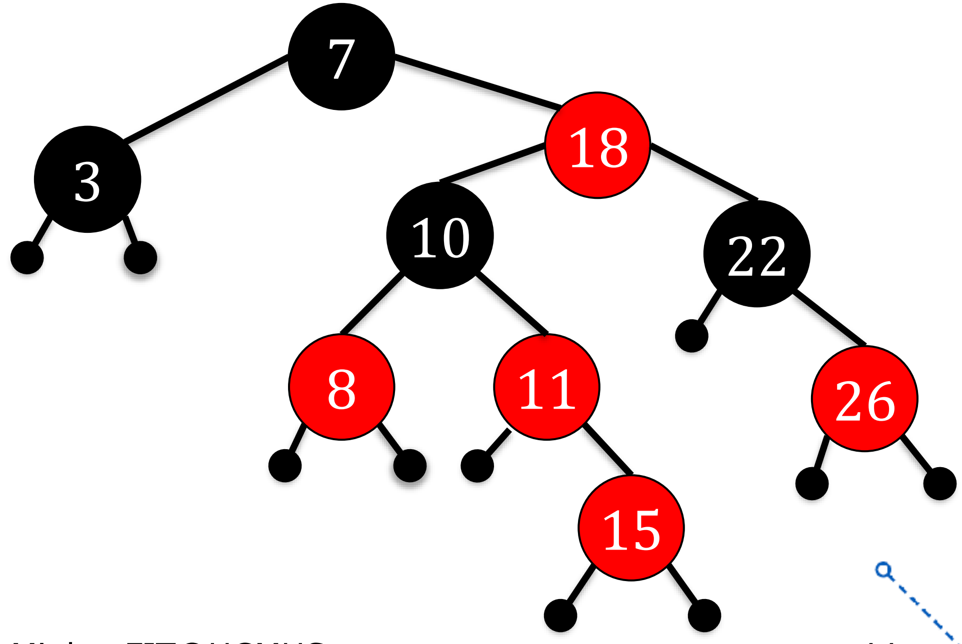
- **Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\log_2 n)$ time on a red-black tree with n nodes.

Modifying operations

- The operations INSERT and DELETE cause modifications to the red-black tree:
  - the operation itself,
  - color changes,
  - restructuring the links of the tree via ***"rotations"***.

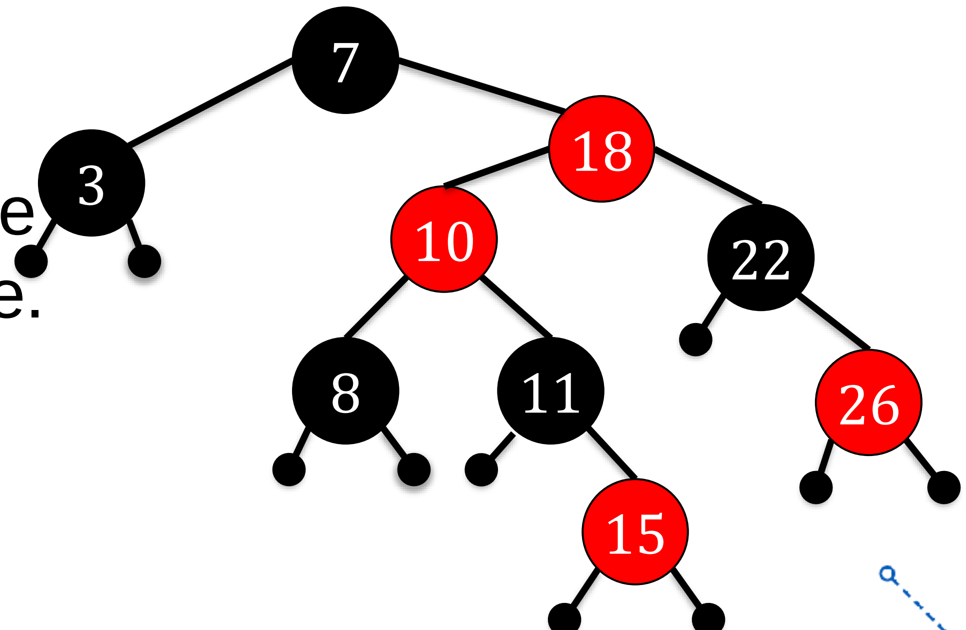
Insertion into a red-black tree

- IDEA: Insert *x* in tree. Color *x* red. Red-black property 2&4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.
- Example:
  - Insert *x* = 15.



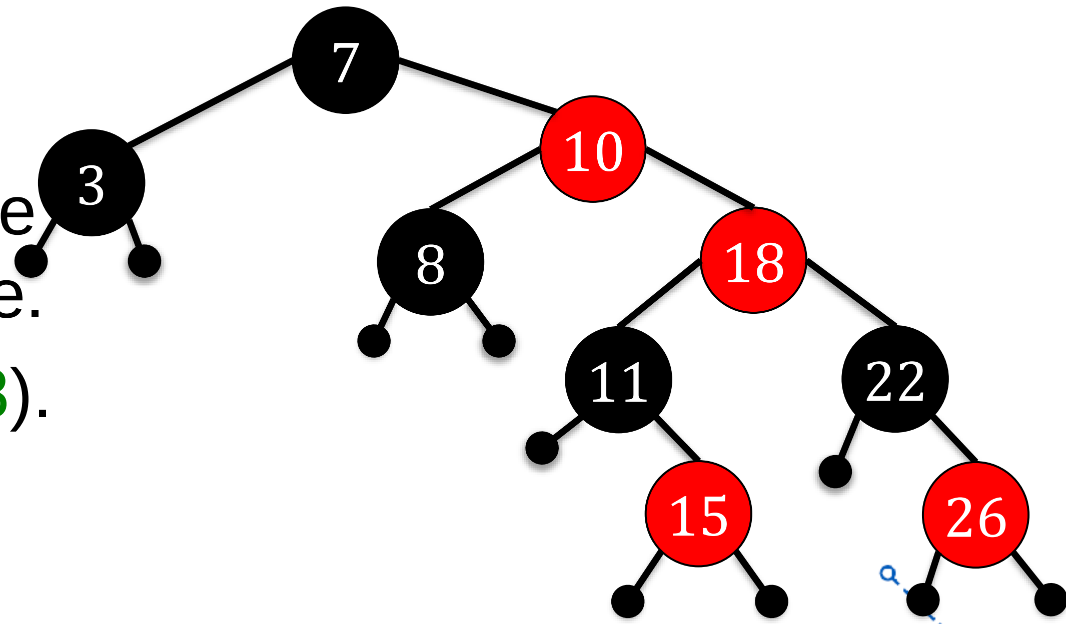
Insertion into a red-black tree

- IDEA: Insert *x* in tree. Color *x* red. Red-black property 2&4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.
- Example:
  - Insert *x* = 15.
  - Recolor, moving the violation up the tree.



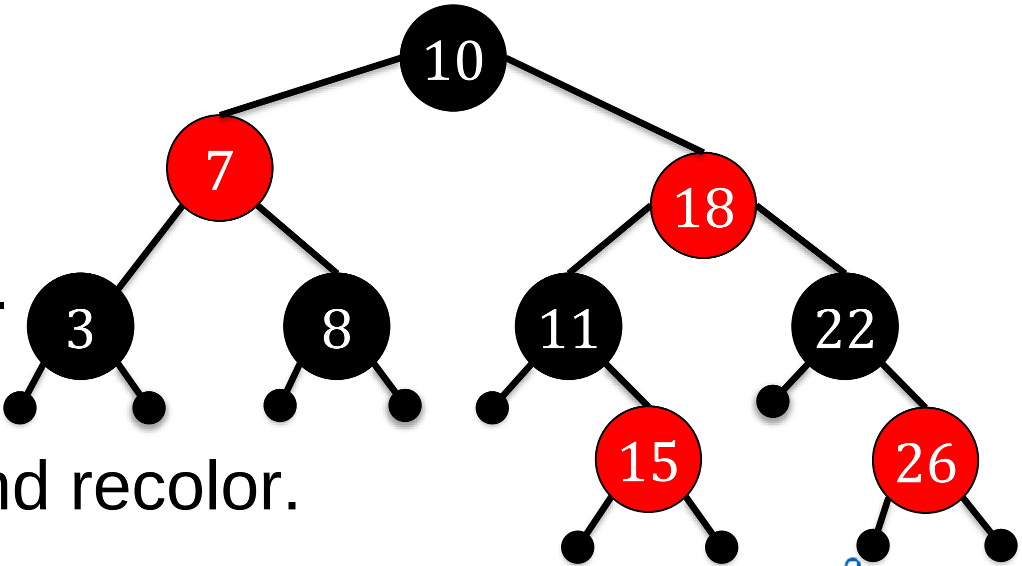
Insertion into a red-black tree

- IDEA: Insert *x* in tree. Color *x* red. Red-black property 2&4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.
- Example:
  - Insert *x* = 15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).

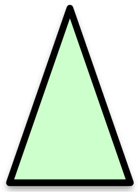
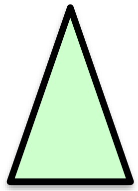


Insertion into a red-black tree

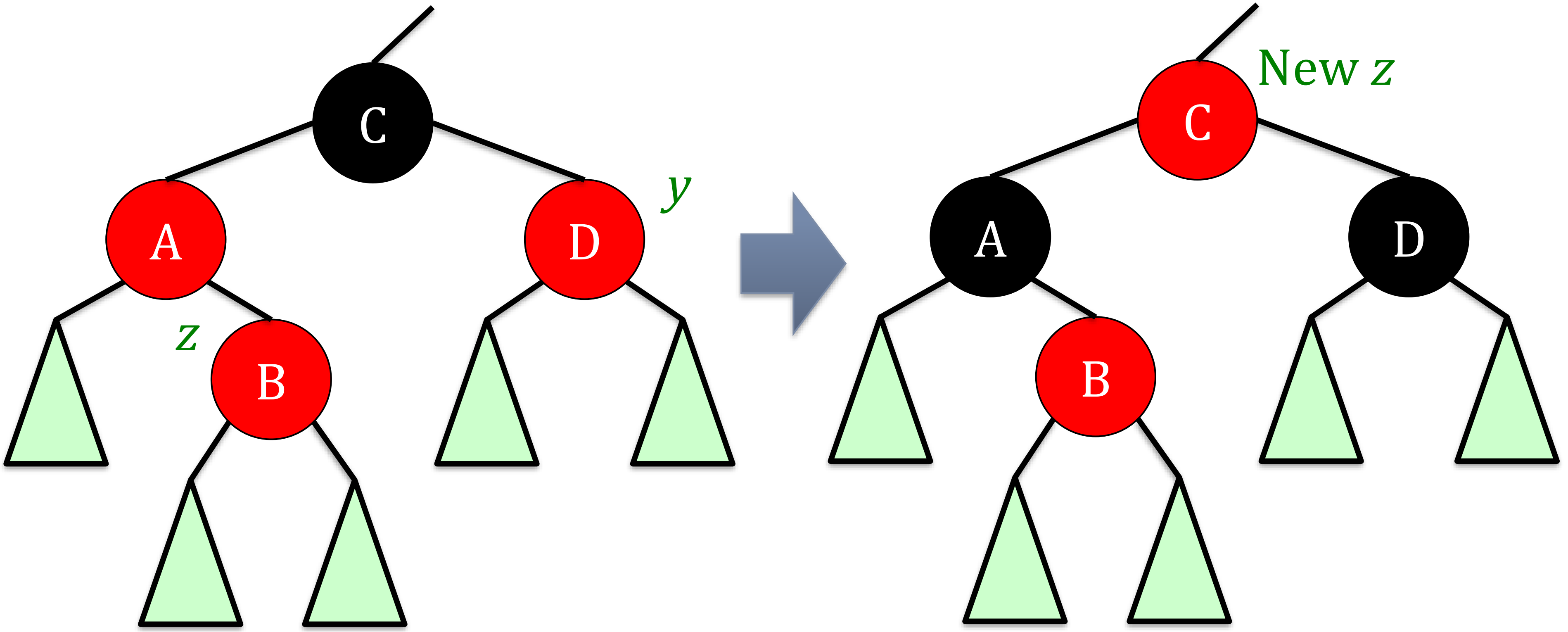
- IDEA: Insert *x* in tree. Color *x* red. Red-black property 2&4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.
- Example:
  - Insert *x* = 15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).
  - LEFT-ROTATE(7) and recolor.



Graphical notation

- Let  denote a subtree with a black root.
- All  ’s have the same black-height.

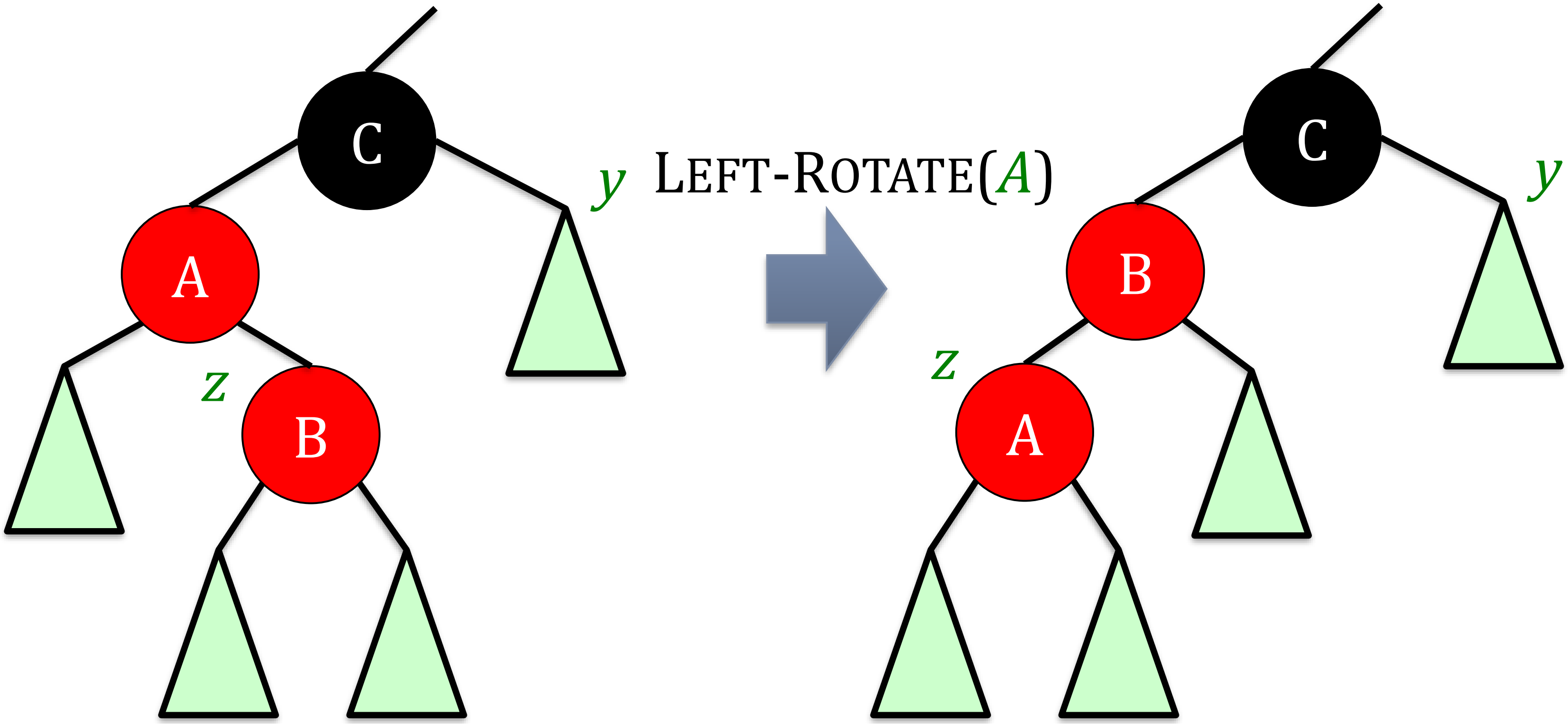
Case 1



(Or, children of *A* are swapped.)

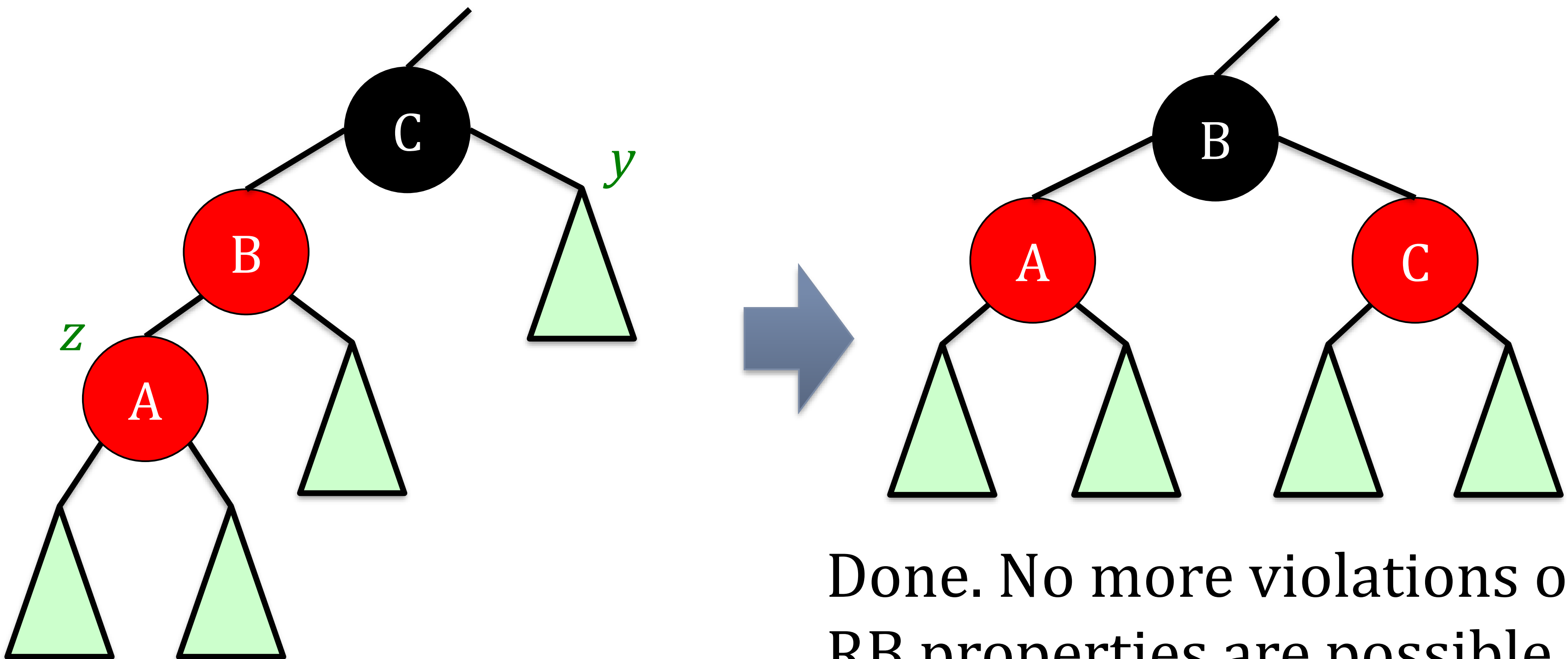
Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.

Case 2



Transform to Case 3.

Case 3



Done. No more violations of RB properties are possible.

Analysis

- RB-INSERT: $\boxed{O(\log_2 n)}$
  - Go up the tree performing Case 1, which only recolors nodes.
  - If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.
- RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook)

$\boxed{O(\log_2 n)}$

AVL trees vs Red-black trees

	AVL trees	Red-black trees
Balance	More strict	Less strict
Max height	$1.44 \log_2(n+2) - 0.328$	$2 \log_2(n+1)$
INSERT	Slower	Faster
DELETE	Slower	Faster
SEARCH	Faster	Slower
TASKS	Look-up intensive tasks	Insert/delete intensive tasks

Some applications

- AVL trees:
  - Not much real-life applications.
  - Case-study: Documents indexing
- Red-black trees:
  - Java: java.util.TreeMap , java.util.TreeSet .
  - C++ STL: map, multimap, multiset.
  - Linux kernel: completely fair scheduler

Q&A