Introduction to Algorithms

Lecture 7

Recap

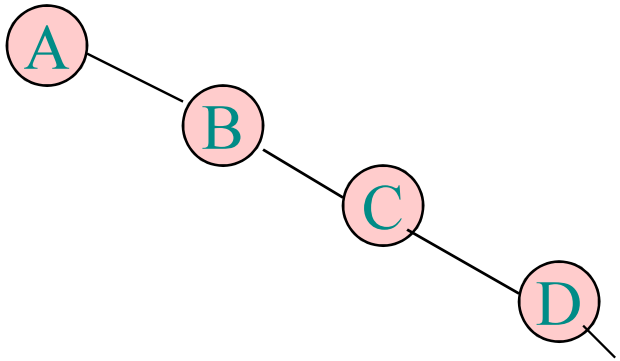
- Binary-Search Trees
- Operations
  - Walk
  - Search, Minimum, Maximum
  - Insert, delete

Today

- Red-black trees
- Rotations
- Insertion
- Deletion
- AVL-Trees

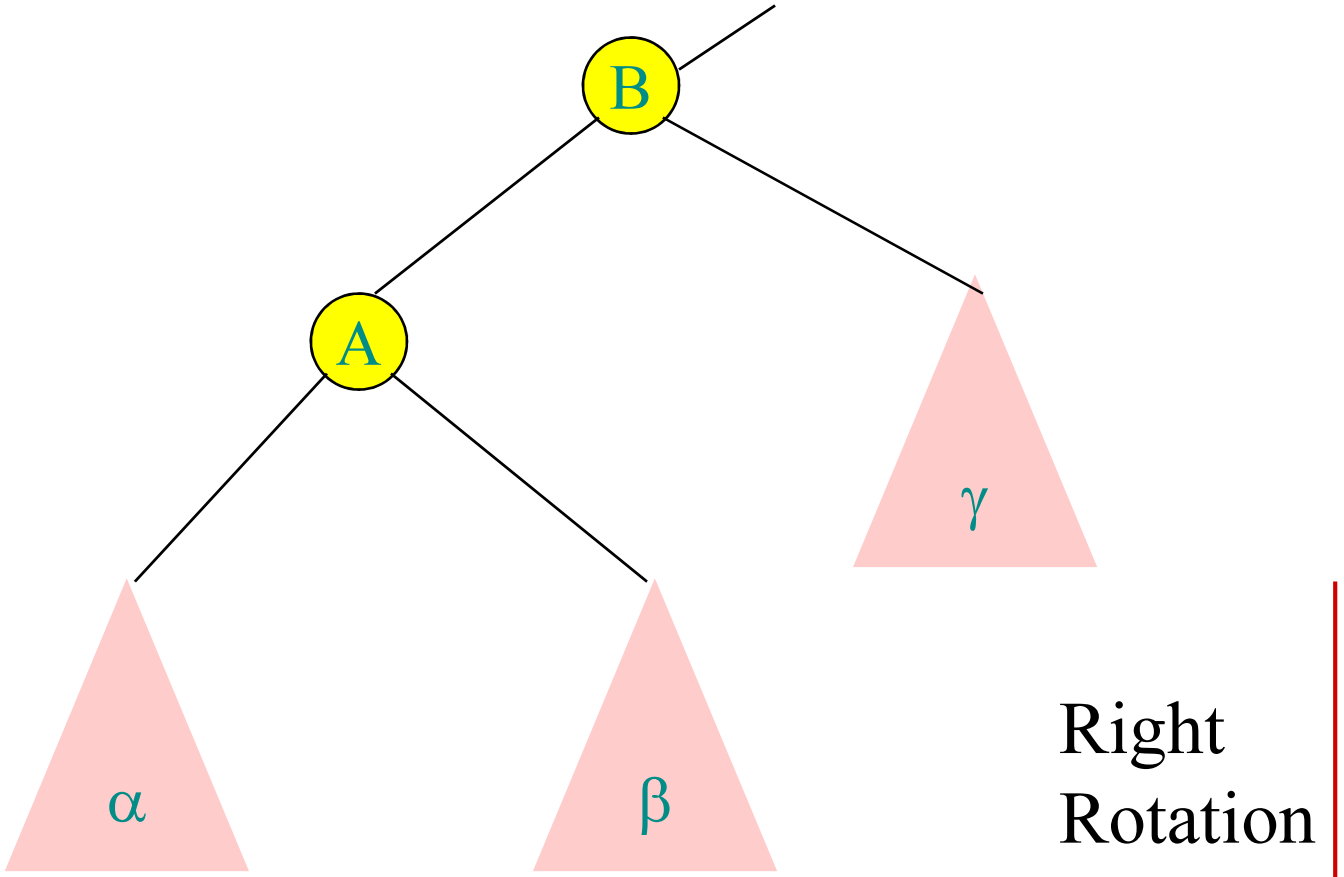
Balanced Search Trees

- Worst case height in unbalanced search trees is $O(n)$.

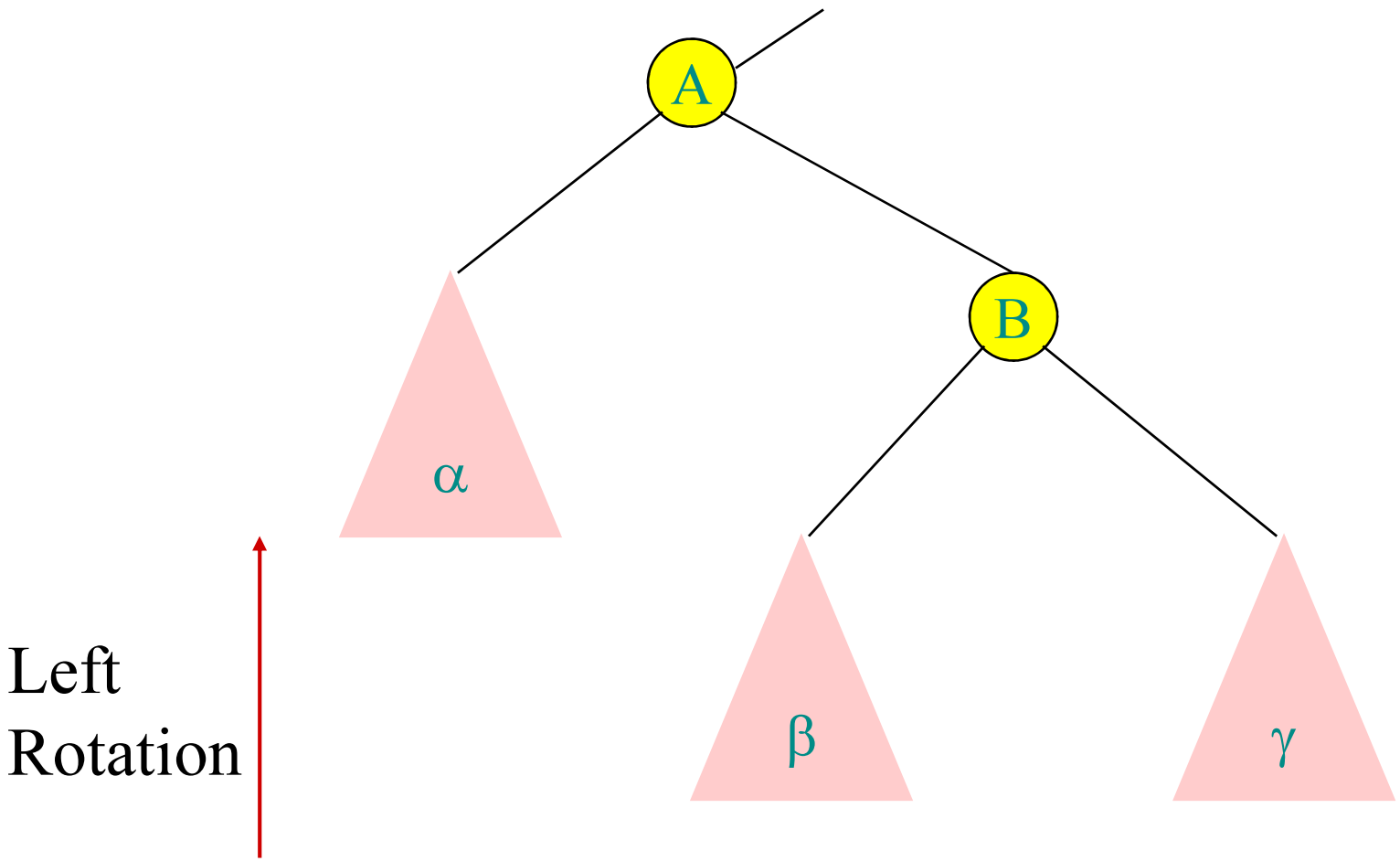


- Balanced search trees guarantee height $O(\lg n)$.
- Many kinds of balanced trees.

Rotation



Rotation



Rotation

- Basic operation for maintaining balanced trees.
- Maintains "inorder" key ordering:

$$\forall a \in \alpha, b \in \beta, c \in \gamma$$

we have

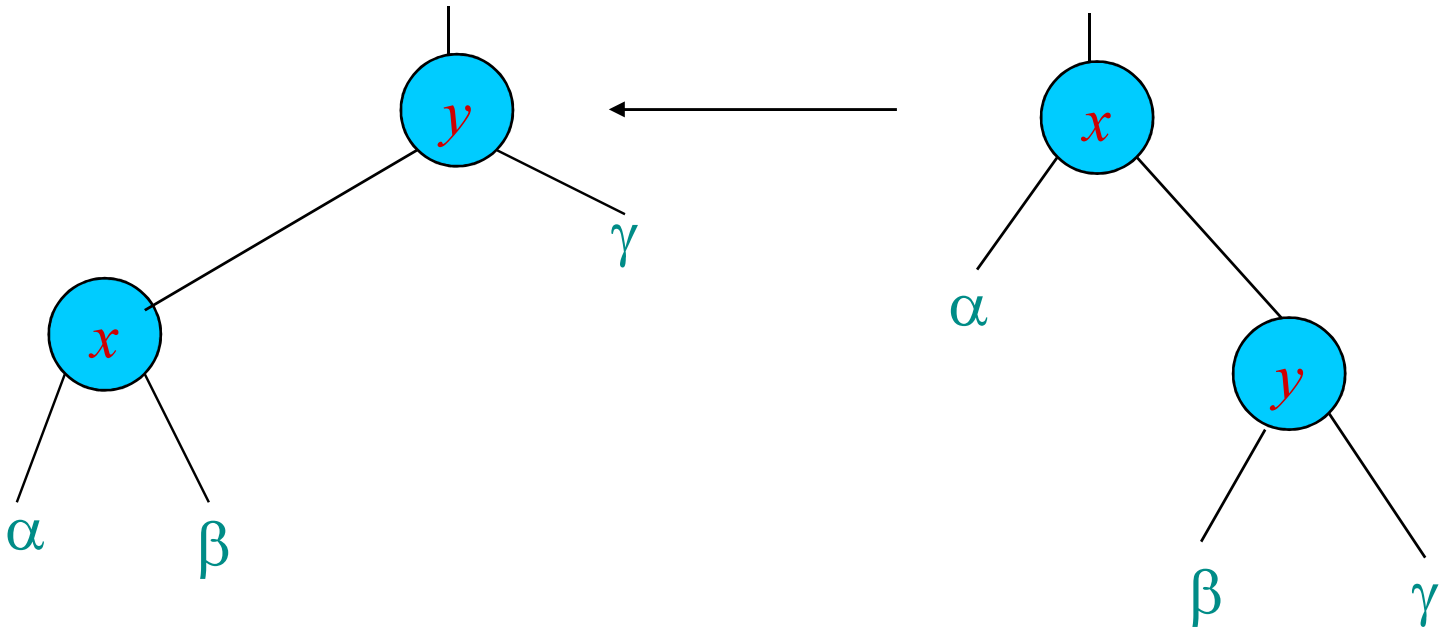
$$a \le A \le b \le B \le c$$

- Depth(α) decrease by 1.
- Depth(β) stays same.
- Depth(γ) increases by 1.
- Rotation takes $O(1)$ time.

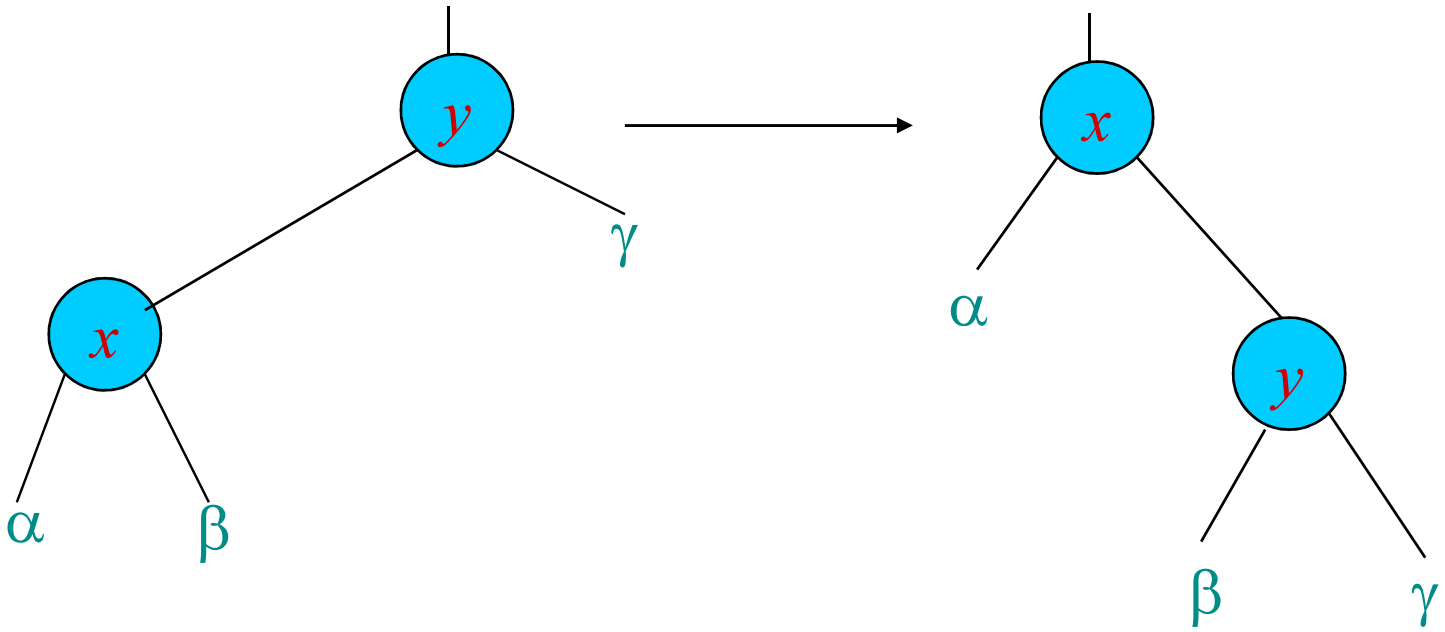
Left-Rotation

```
Left-Rotate(T, x)
1.    y ← right[x]
2.    right[x] ← left[y]
3.    if left[y] ≠ NIL
4.       then p[left[y]] ← x
5.    p[y] ← p[x]
6.    if p[x] = NIL
7.       then root[T] ← y
8.       else if x = left[p[x]]
9.                then left[p[x]] ← y
10.               else right[p[x]] ← y
11.   left[y] ← x
12.   p[x] ← y
```

Left-Rotation



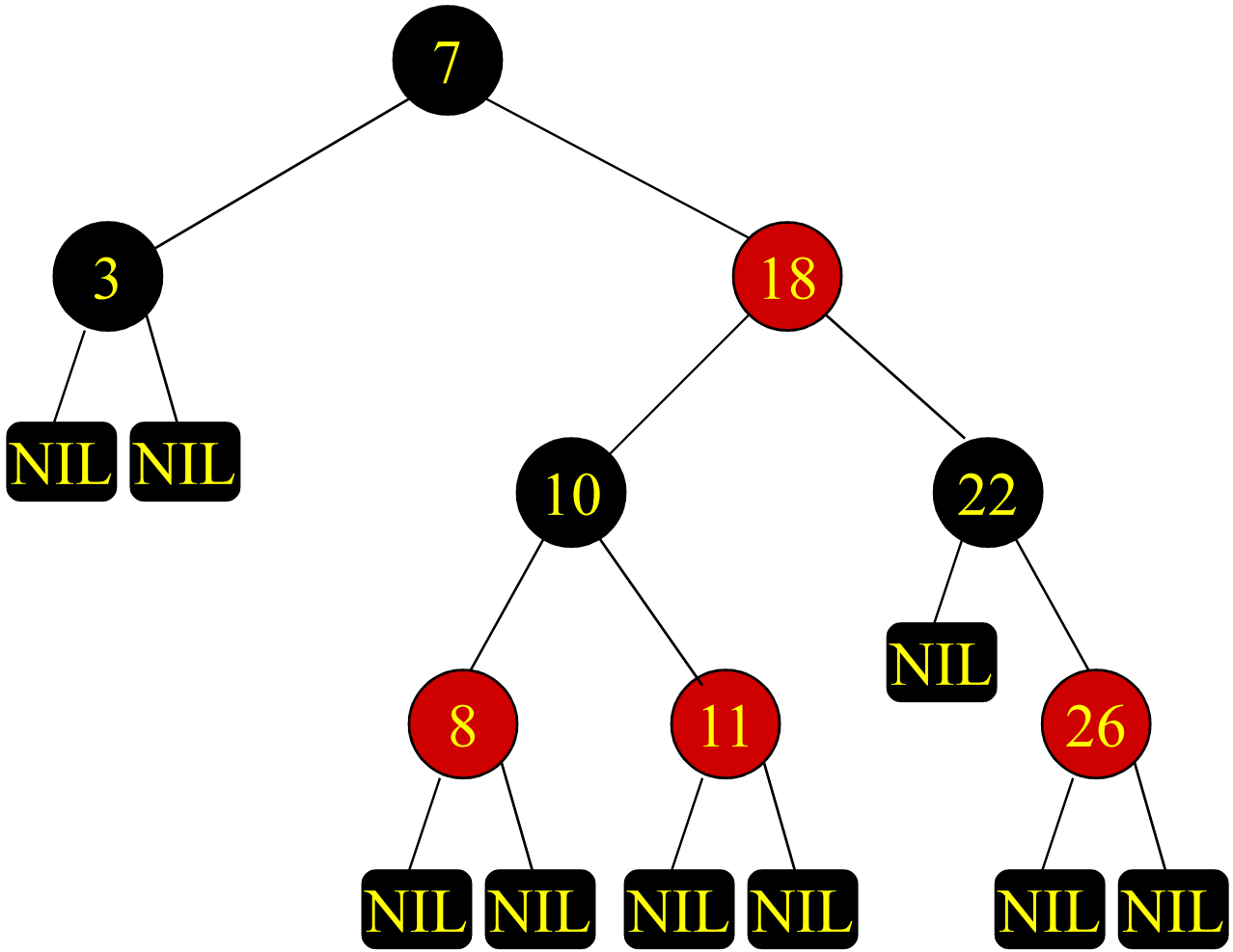
Right-Rotation



Red-Black Trees

- Every node is either red or black.
- Root and leaves (NIL) are black.
- If a node is red, then both children are black.
- All paths from any node x to a descendant leaf have same number of black nodes (Black-Height(x)).

Red-Black Tree Example



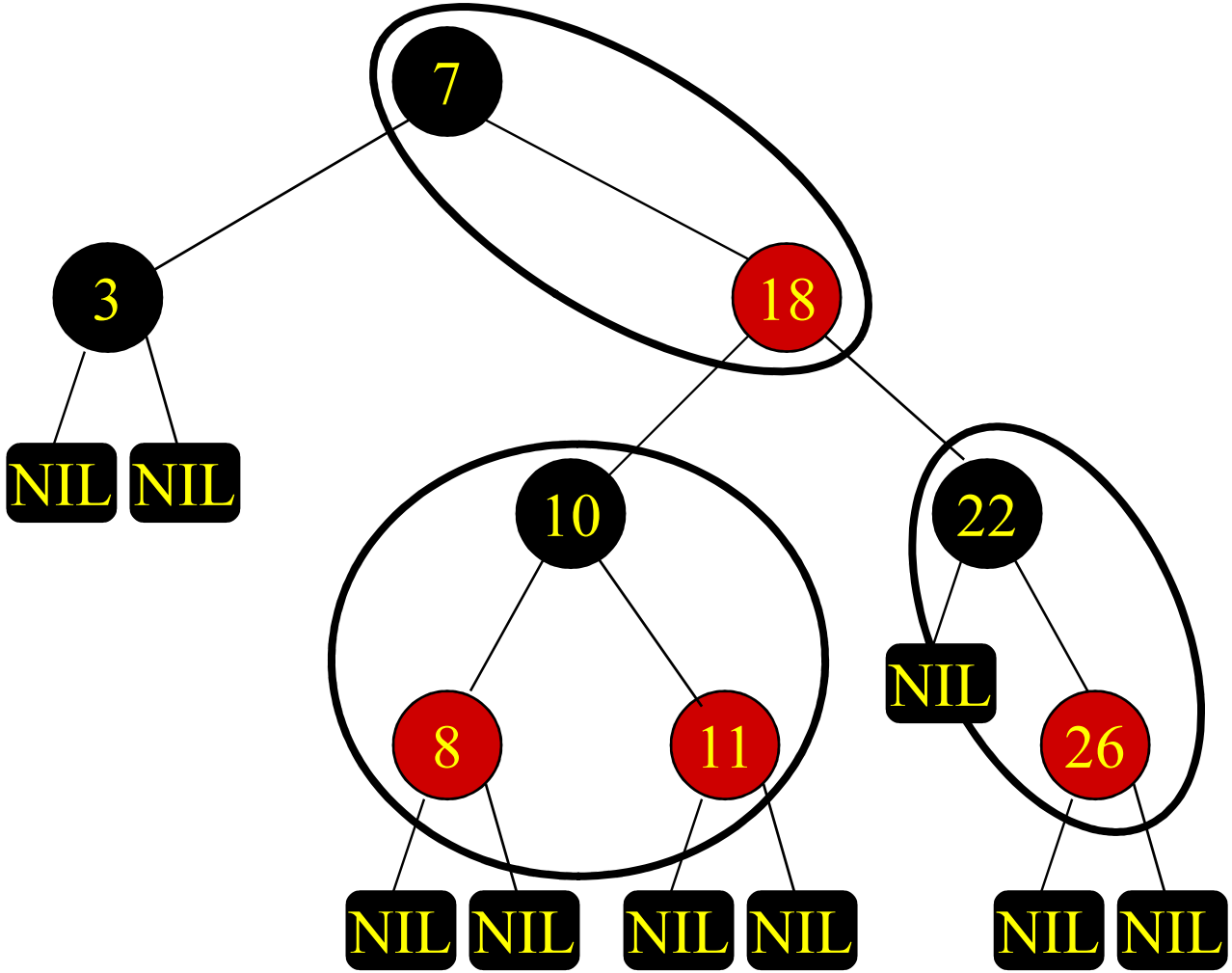
Red-Black Trees

Theorem. A red-black tree with n keys has height $h \leq 2\lg(n+1)$.

Proof (Intuition):

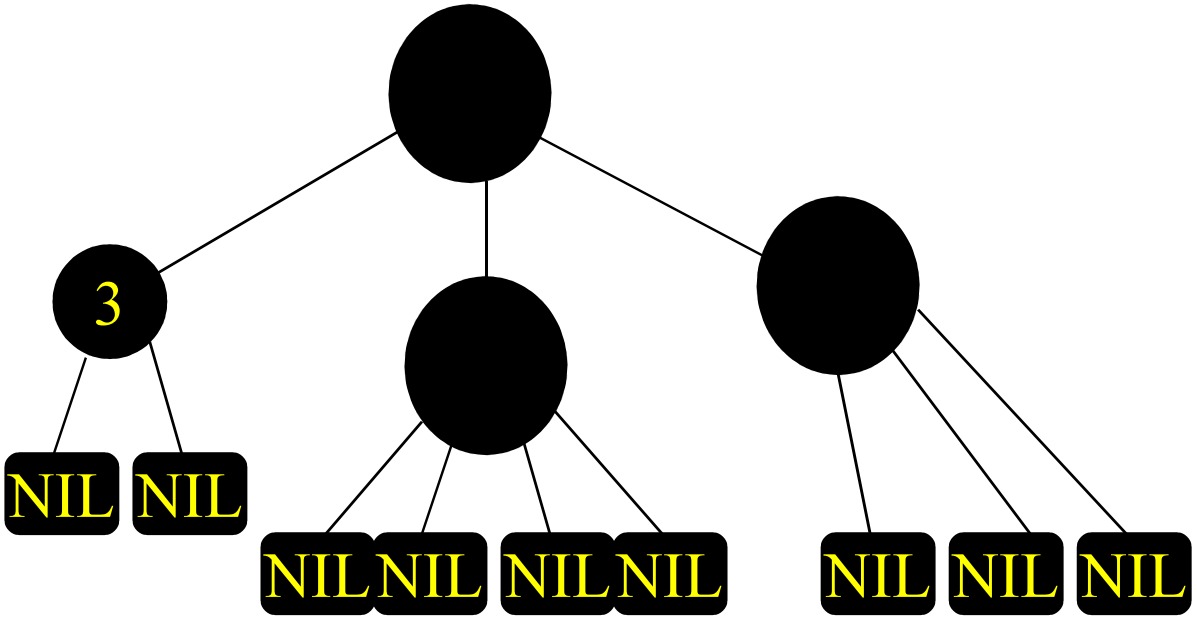
"Merge" the red nodes into their parents.

Red-Black Trees



Red-Black Trees

- Produces a tree with nodes having 2, 3 or 4 children.



Red-Black Trees

- Height h' of new tree is black height of original tree.
- $h' \geq h/2$
- $n + 1$ leaves $\rightarrow$ $n + 1 \geq 2^{h'}$
- $\lg(n + 1) \geq h' \geq h/2$.

Corollary:

Search, Min, Successor, etc. take $O(h) = O(\lg n)$ time on a red-black tree!

Red-Black Insertion

- Insert x into tree.
- Color x red.
- Red-black property 1 still holds.
- Red-black property 2 still holds (inserted node has NIL's for children).
- Red-black property 4 still holds (x replaces a black NIL and x has NIL children).

Red-Black Insertion

- If $p[x]$ is red, then property 3 is violated.
- To correct, we move violation up tree until we find a place at which it can be fixed!
- No new violations are introduced.
- For each iteration, there are six possible cases.

Loop Invariant for RB-Trees

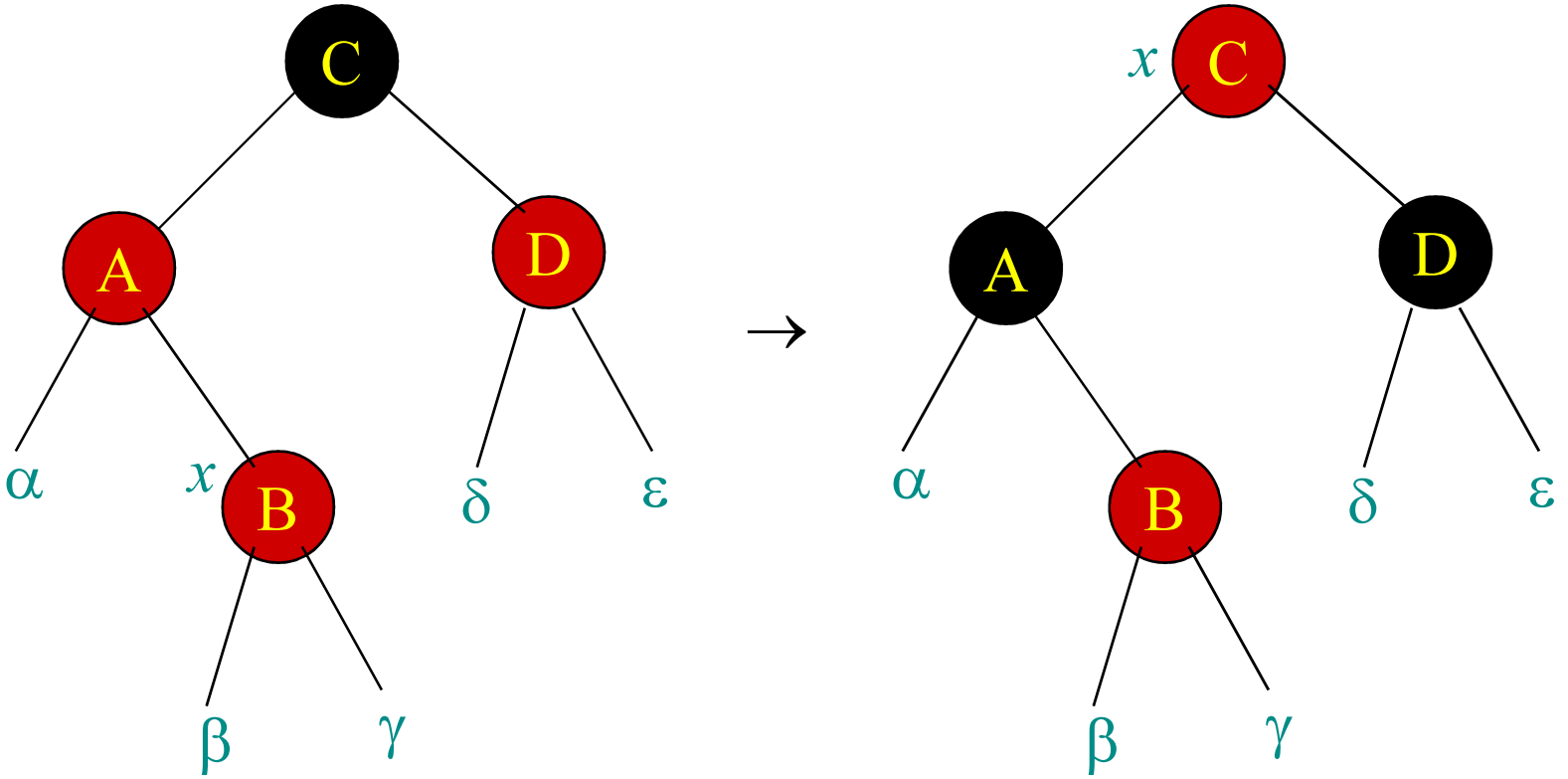
At the start iteration of the loop:

a. Node x is red.
b. If $p[x]$ is the root, then $p[x]$ is black.
c. If there is a violation of the red-black properties, there is at most one violation, and it is of either property 2 or property 3.

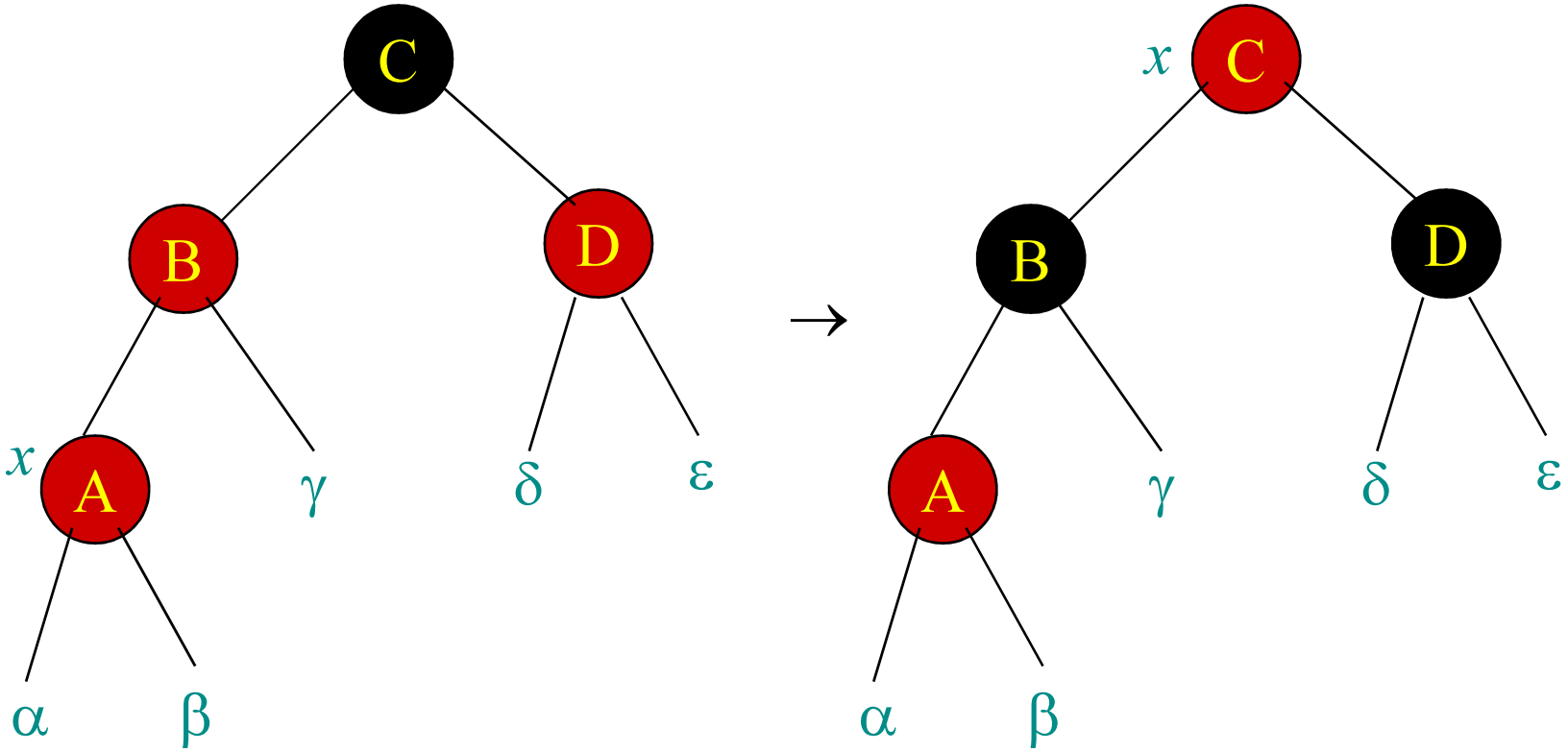
Case 1

- x's parent is the left child of x's grandparent.
- x's parent's sibling (x's uncle) is red.
- Then,
  - $color[p[x]] \leftarrow$ BLACK
  - $color[right[p[p[x]]]] \leftarrow$ BLACK
  - $color[p[p[x]]] \leftarrow$ RED
  - $x \leftarrow p[p[x]]$.

Case 1 Example 1



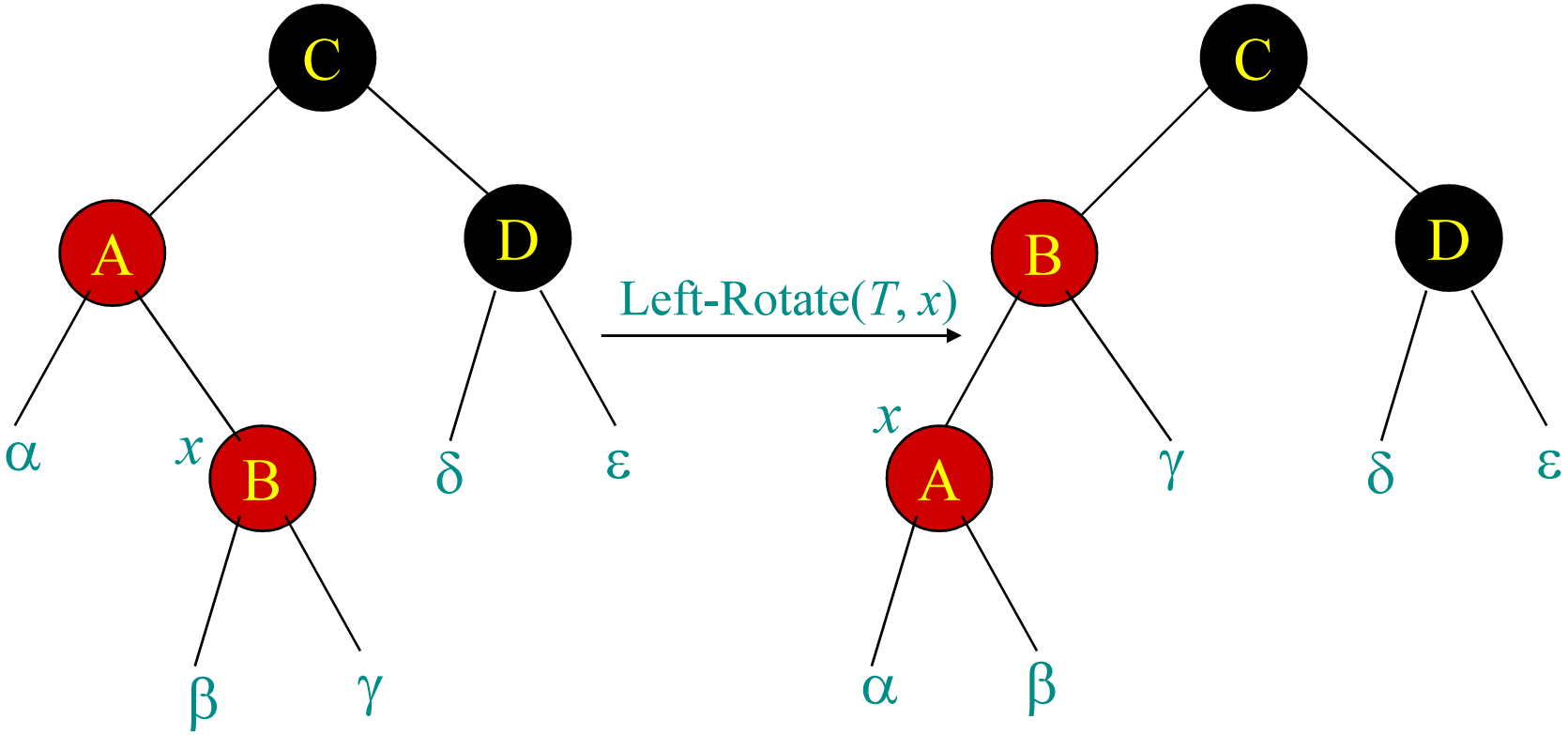
Case 1 Example 2



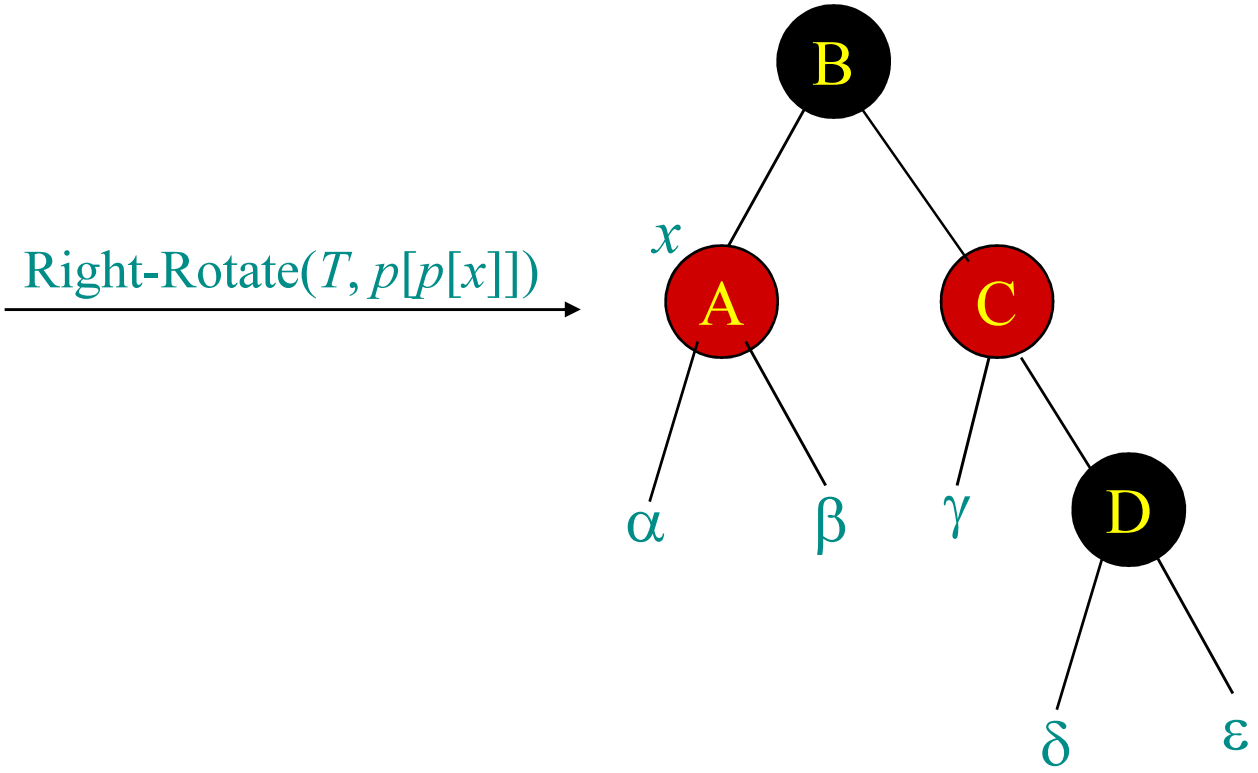
Case 2

- x's parent is the left child of x's grandparent.
- x's uncle is black.
- x is the right child of $p[x]$.
- Then,
  - $x \leftarrow p[x]$
  - Left-Rotate(T, x)
  - $color[p[x]] \leftarrow$ BLACK
  - $color[p[p[x]]] \leftarrow$ RED
  - Right-Rotate(T, $p[p[x]]$)

Case 2 Example



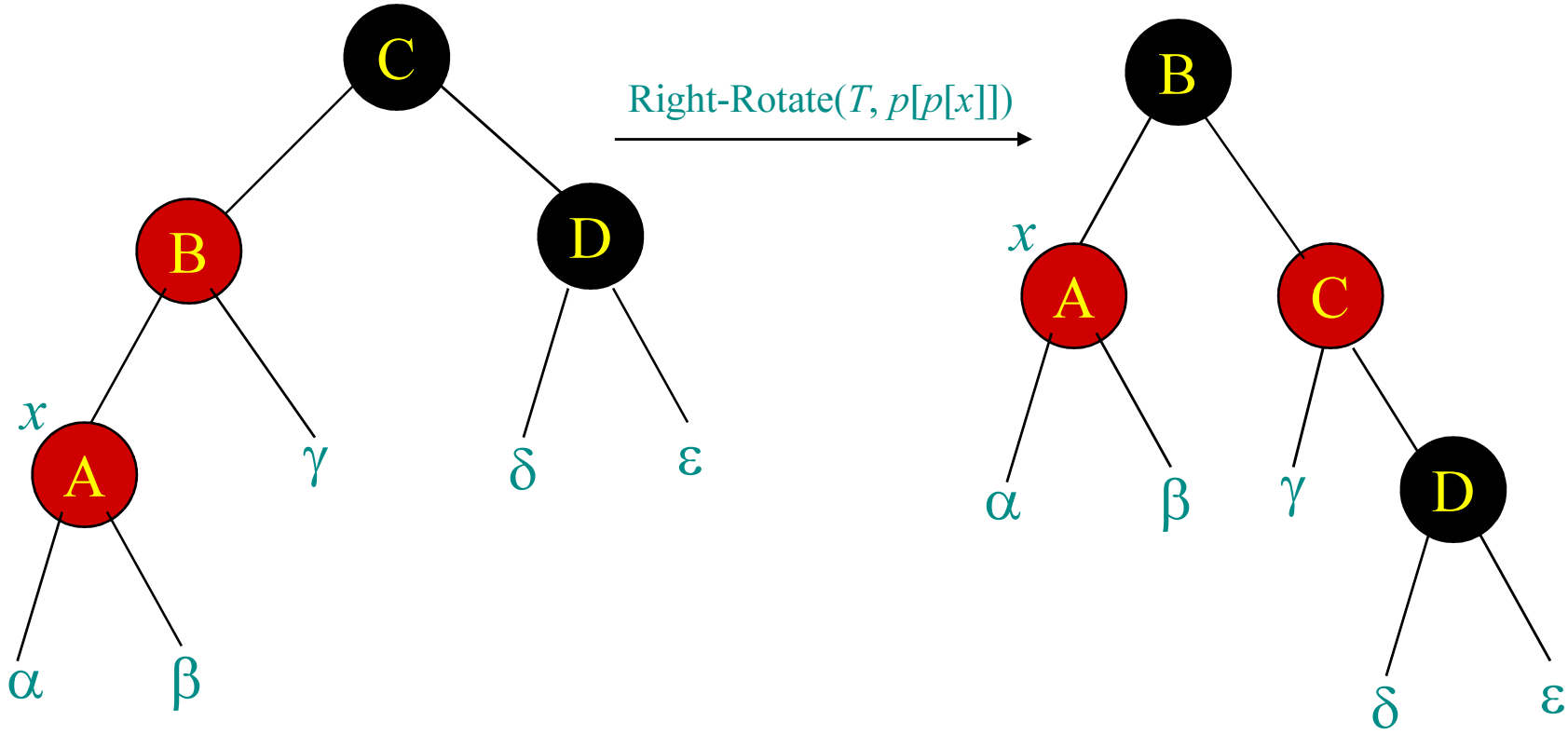
Case 2 Example (cont.)



Case 3

- x's parent is the left child of x's grandparent.
- x's uncle is black.
- x is the left child of $p[x]$.
- Then,
  - $color[p[x]] \leftarrow$ BLACK
  - $color[p[p[x]]] \leftarrow$ RED
  - Right-Rotate(T, $p[p[x]]$)

Case 3 Example



Red-Black Insertion (cont.)

Additional Notes

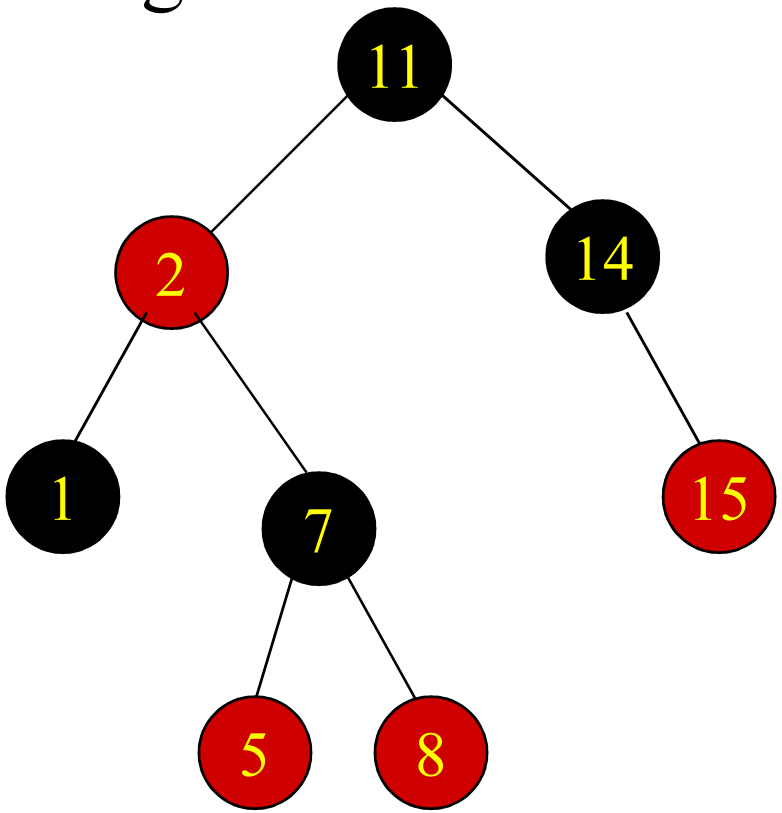
- Cases 4,5,6 symmetrical to 1,2,3 (x's parent is the right child of x's grandparent).
- After case 2 or 3, no further correction is necessary.

Insertion Pseudocode

```
RB-Insert(T, x)
1.      Tree-Insert(T, x)
2.      color[x] ← RED
3.      while x ≠ root[T] and color[p[x]] = RED
4.          do if p[x] = left[p[p[x]]]
5.                then y ← right[p[p[x]]]
6.                     if color[y] = RED
7.                       then color[p[x]] ← BLACK
8.                            color[y] ← BLACK
9.                            color[p[p[x]]] ← RED
10.                           x ← p[p[x]]
11.                    else if x = right[p[x]]
12.                           then x ← p[x]
13.                                Left-Rotate(T, x)
14.                          color[p[x]] ← BLACK
15.                          color[p[p[x]]] ← RED
16.                          Right-Rotate(T, p[p[x]])
17.               else (same as then clause [line 5] with "right" and "left" exchanged)
18.     color[root[T]] ← BLACK
```

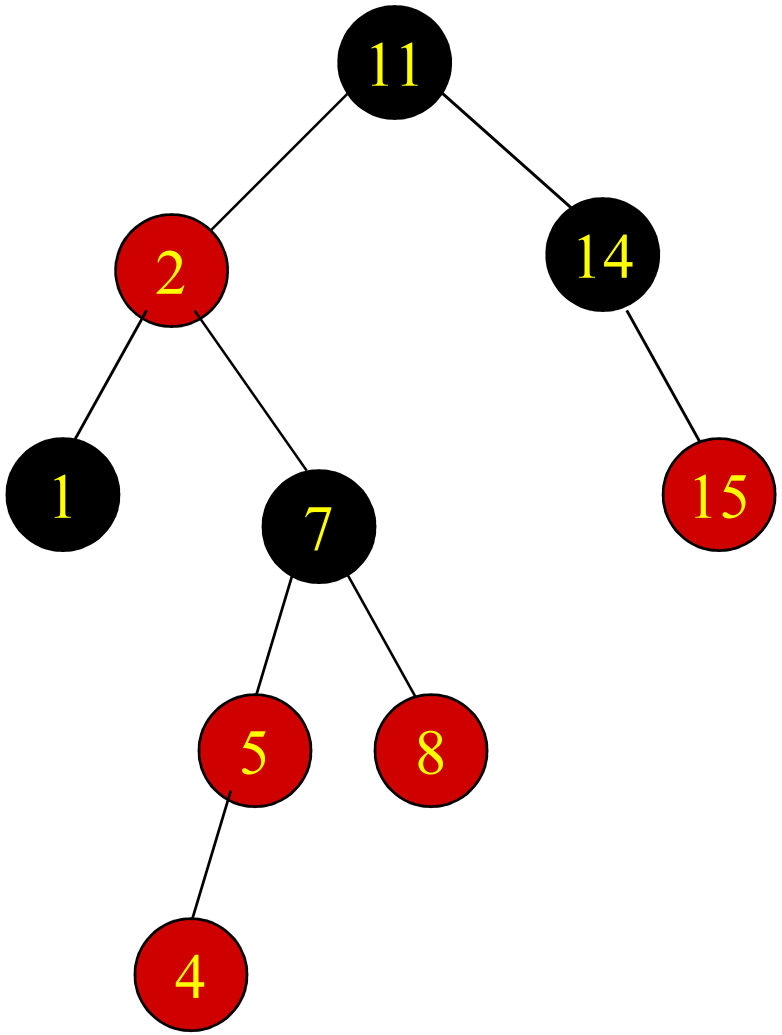
Insertion Example

- Use RB-Insert to insert element with key 4 into following tree.



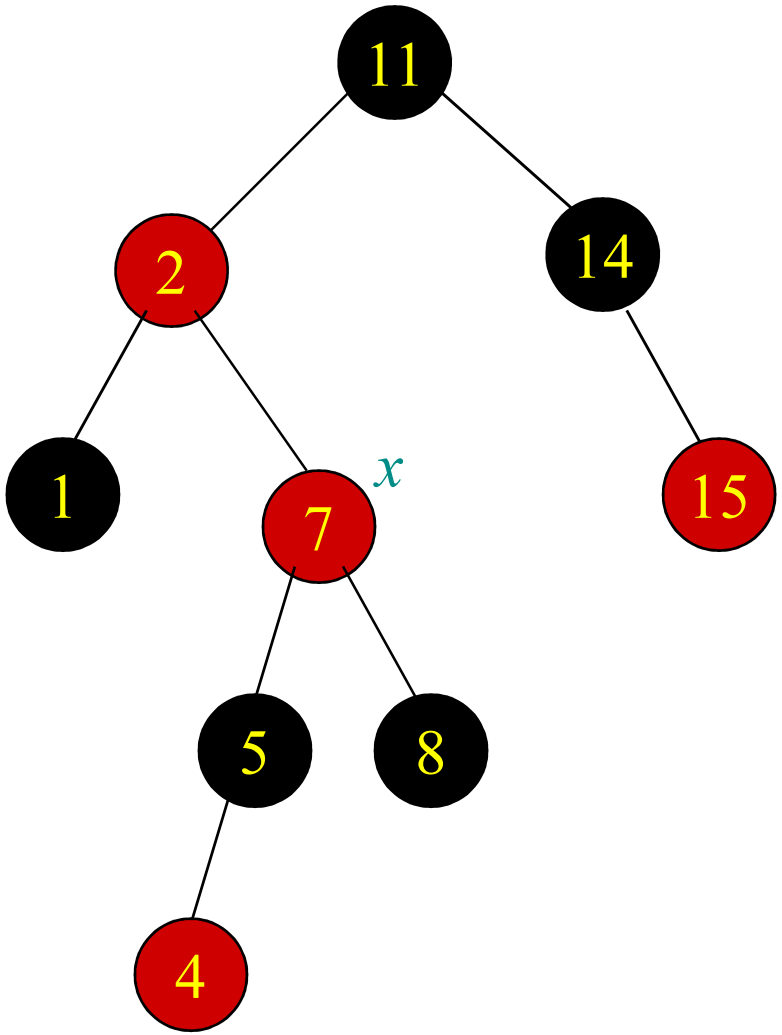
Insertion Example (cont.)

- Insert x.



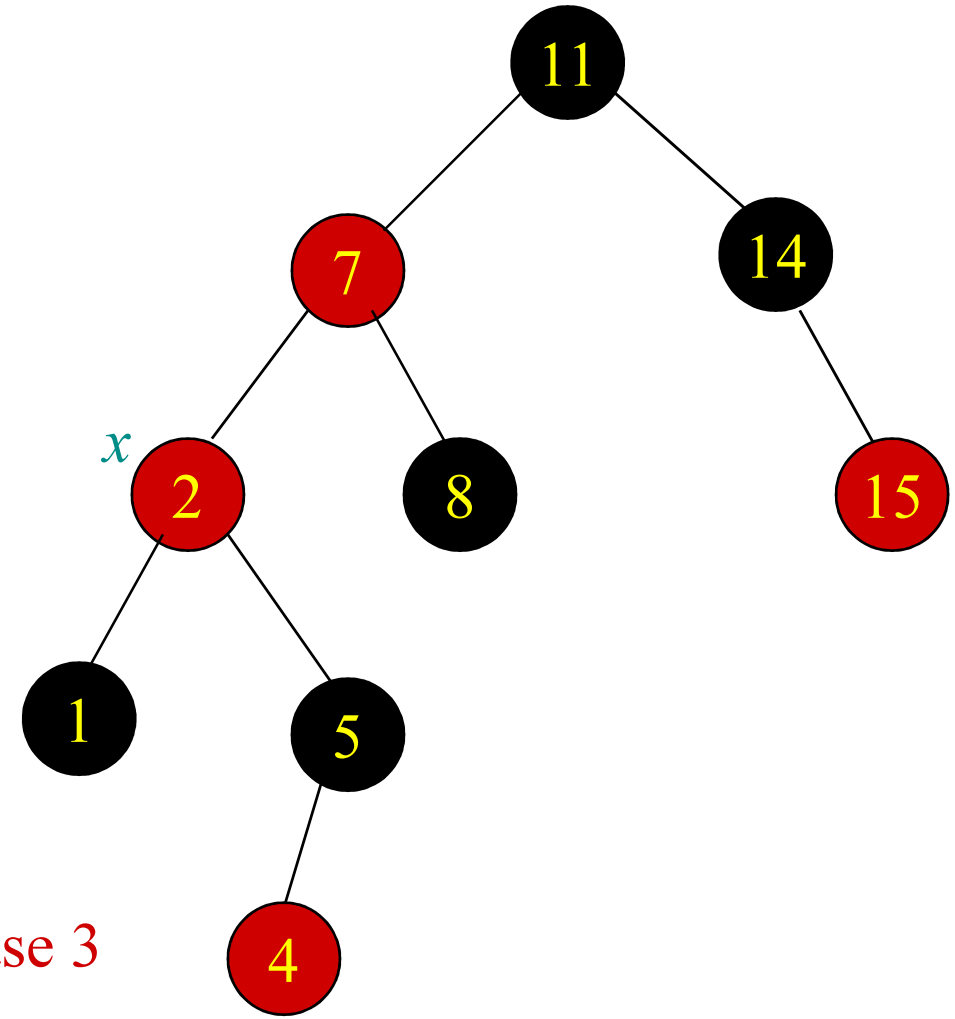
•Case 1 ($p[x] = left[p[p[x]]]$; $color(uncle(x)) = \text{red}$)

Insertion Example (cont.)



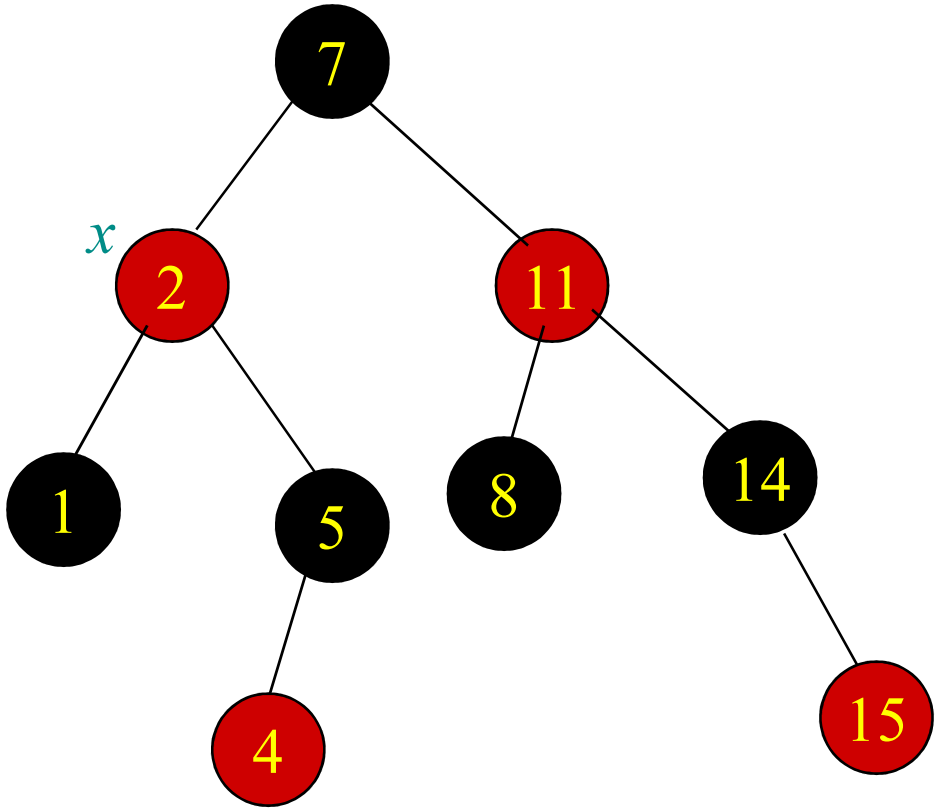
- Case 2

Insertion Example (cont.)



- Case 3

Insertion Example (cont.)



- Done!

Analysis of Insertion

- Each case in loop takes $O(1)$ time.
- Loop moves up tree performing case 1.
- Loop terminates after case 2 or 3.
- Total time required is $O(h) = O(\lg n)$.

Red-Black Delete

- Same running time as Insert
- Pseudocode in CLRS.

Recall the rules for BST deletion

- If vertex to be deleted is a leaf, just delete it.
- If vertex to be deleted has just one child, replace it with that child.
- If vertex to be deleted has two children, replace the value of by it's in-order predecessor's value then delete the in-order predecessor (a recursive step).

What can go wrong?

- If the delete node is red?

  Not a problem – no RB properties violated

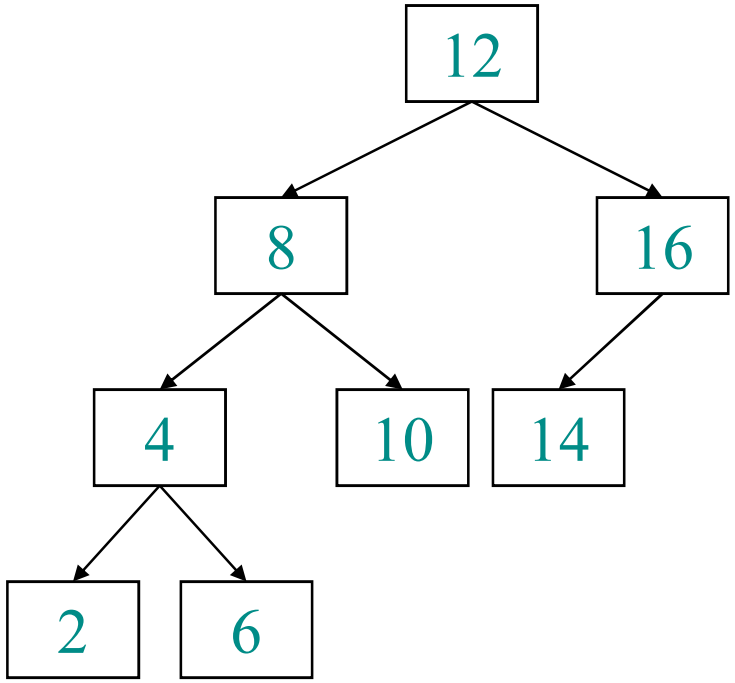
- If the deleted node is black?

  If the node is not the root, deleting it will change the black-height along some path

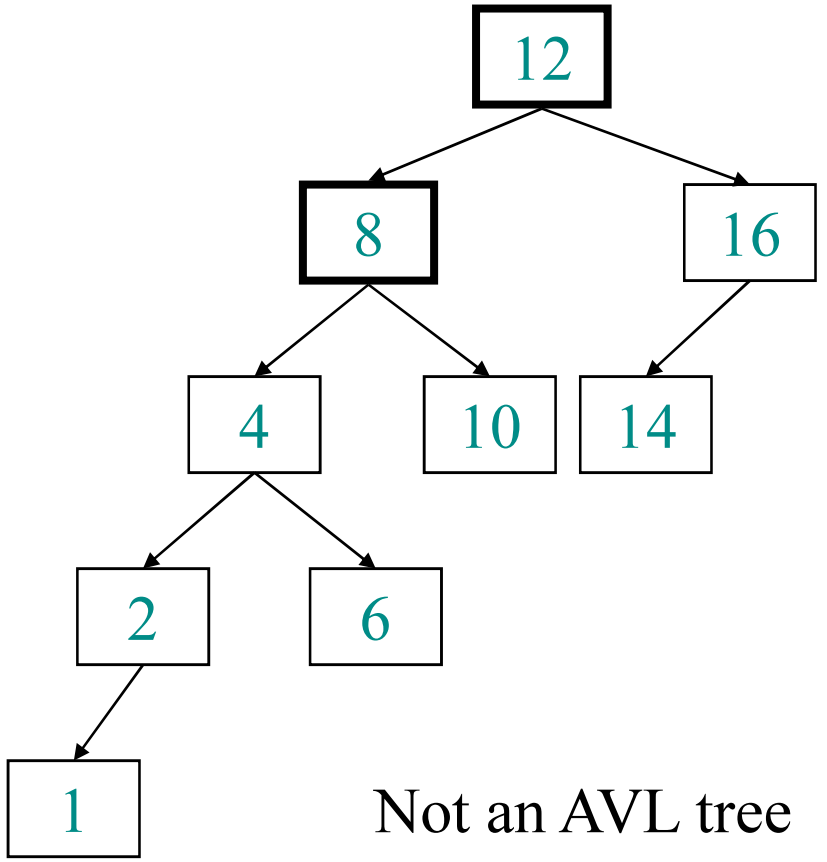
AVL Trees

- Balanced Trees: After insert and delete operations we "fix" the tree to keep it (almost) balanced.
- **AVL Tree:** A binary search tree with the following additional balance property: For any node in the tree, the height of the left and right subtrees can differ by 1 at most.
- Note that we require this balance property for every node, not just the root.

Example



An AVL tree



Not an AVL tree

The Maximal Height of an AVL Tree

Definition S_h: the size of the smallest AVL tree with height h.

Claim: $S_h = S_{h-1}+S_{h-2}+1$ ($S_0 = 1$; $S_1 = 2$)

sketch of proof: The smallest AVL tree of height h is composed of a root, one subtree which is the smallest AVL tree of height h-1, and another subtree which is the smallest AVL tree of height h-2 (see next slide for illustration).

The Maximal Height of an AVL Tree

Reminder: **Fibonacci numbers** are defined recursively by:

$F_0 = 0$; $F_1 = 1$; $F_i = F_{i-1} + F_{i-2}$ [Fact: $F_i > (1.3)^i$ for $i > 3$]

Claim: $S_h = F_{h+3} - 1$

The proof is by a simple induction, using the fact that $S_h = S_{h-1} + S_{h-2} + 1$

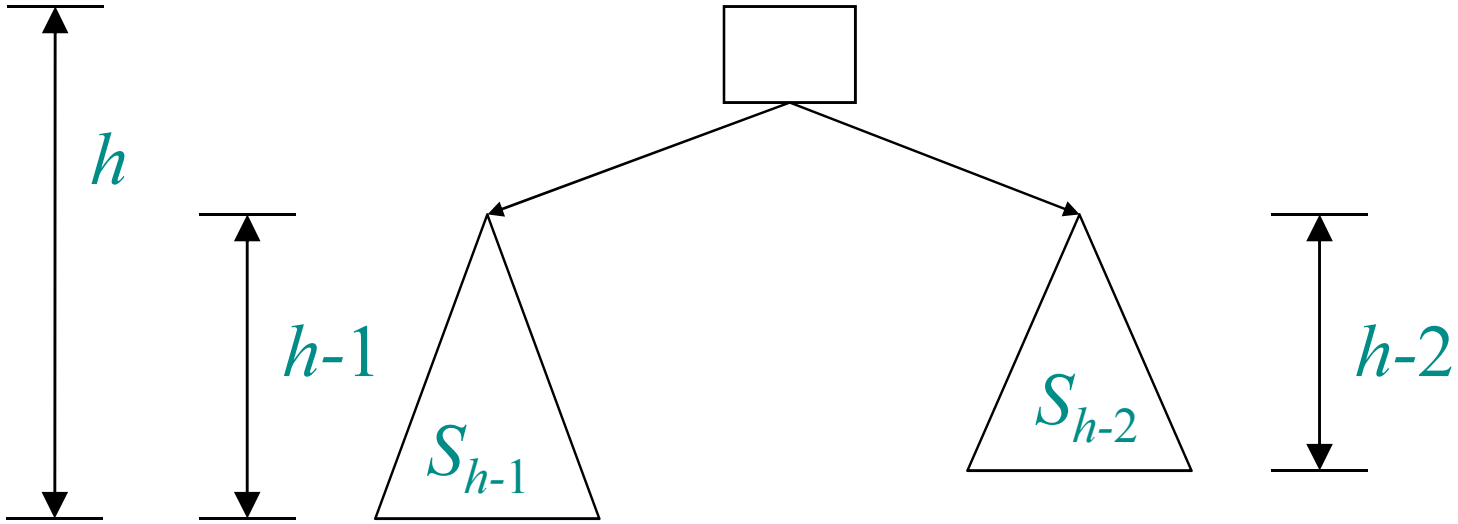
Theorem: For any AVL tree with n nodes and height h:

$h = O(\log n)$.

Proof:

$$n \geq S_h = F_{h+3} - 1 \geq (1.3)^{h+3} - 1 \Rightarrow h \leq \log_{1.3}(n+1) \Rightarrow h = O(\log n)$$

Minimal AVL tree of height h



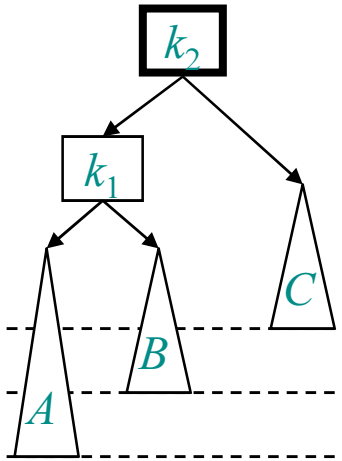
- Since the root's height is h, one of its sons' height must be h-1. From the balance condition, the other son has height either h-1 or h-2. From minimality, we get that the sons has S_{h-1} and S_{h-2} nodes, respectively.

How to maintain balance

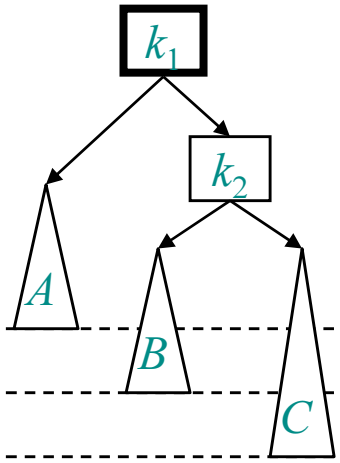
- General rule: after an insert or delete operation, we fix all nodes that got unbalanced.
- Since the height of any subtree has changed by at most 1, if a node is not balanced this means that one son has a height larger by exactly two than the other son.
- Next we show the four possible cases that cause a height difference of 2. In all figures, marked nodes are unbalanced nodes.

Four imbalance cases

Case 1: The left subtree is higher than the right subtree, and this is caused by the left subtree of the left child.

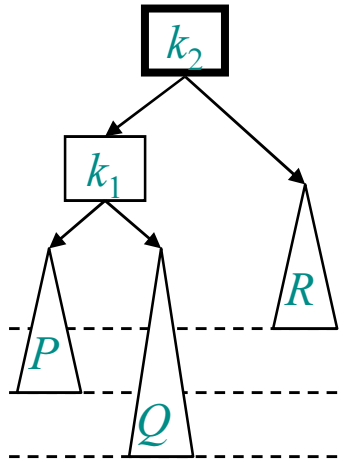


Case 4:
The symmetric case to case 1

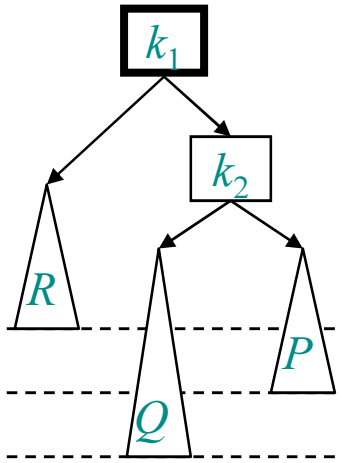


Four imbalance cases

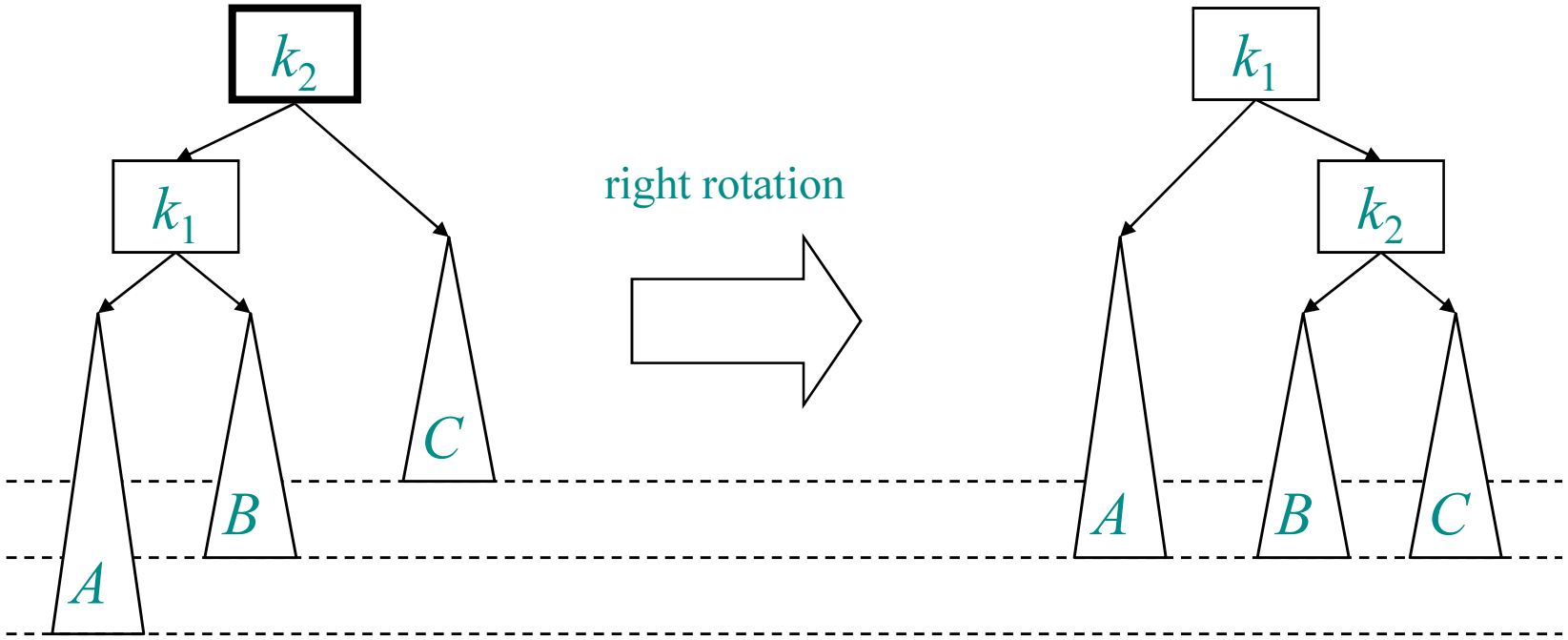
Case 2: The left subtree is higher than the right subtree, and this is caused by the right subtree of the left child.



Case 3:
The symmetric case to case 2

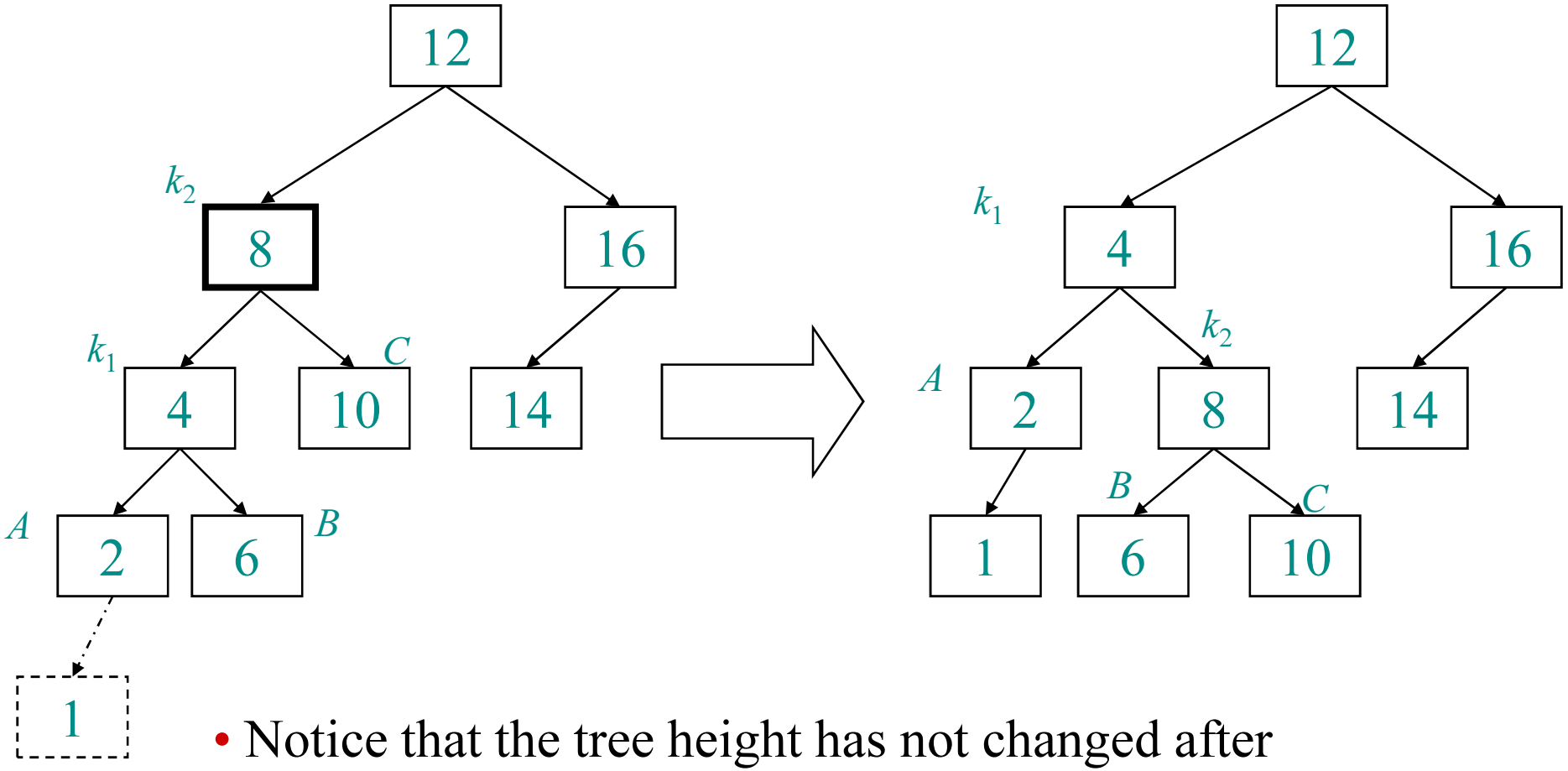


Single Rotation - Fixing case 1



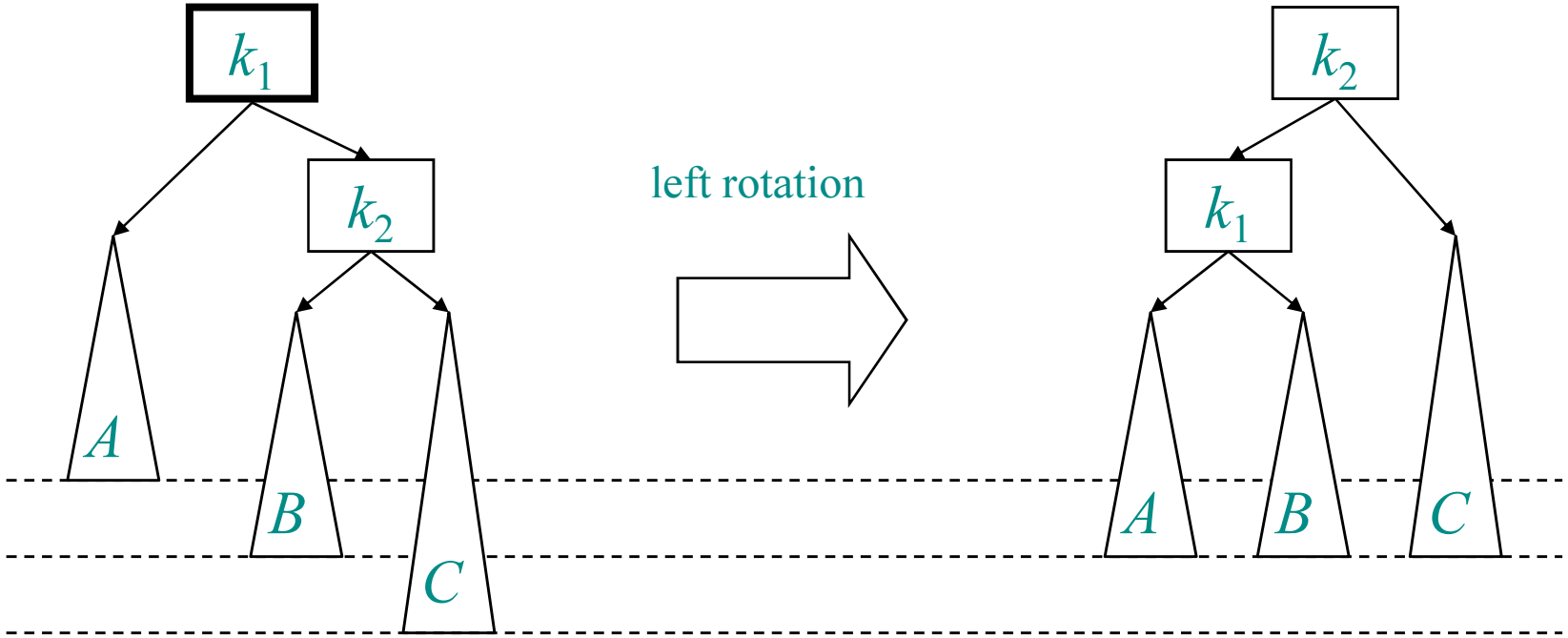
- The rotation takes $O(1)$ time. Notice that the new tree is a legal search tree.
- For insert - it must be the case that subtree A had been increased, so after the rotation, k_1 has height as before the insert.
- For delete, it must be the case that C had been decreased, so after the rotation, k_1 has height shorter by 1.

Example (caused by insertion)



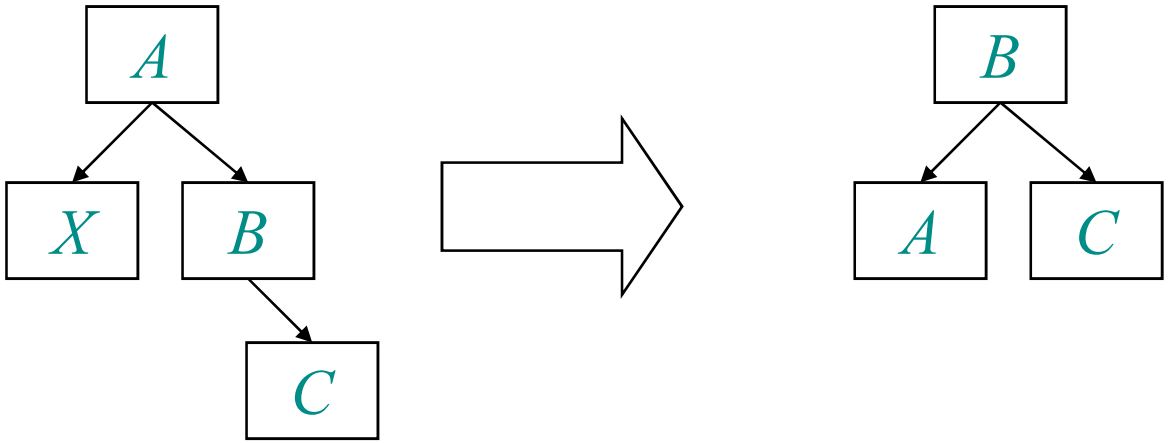
- Notice that the tree height has not changed after the rotation (since it was an insert operation).

Single Rotation for Case 4



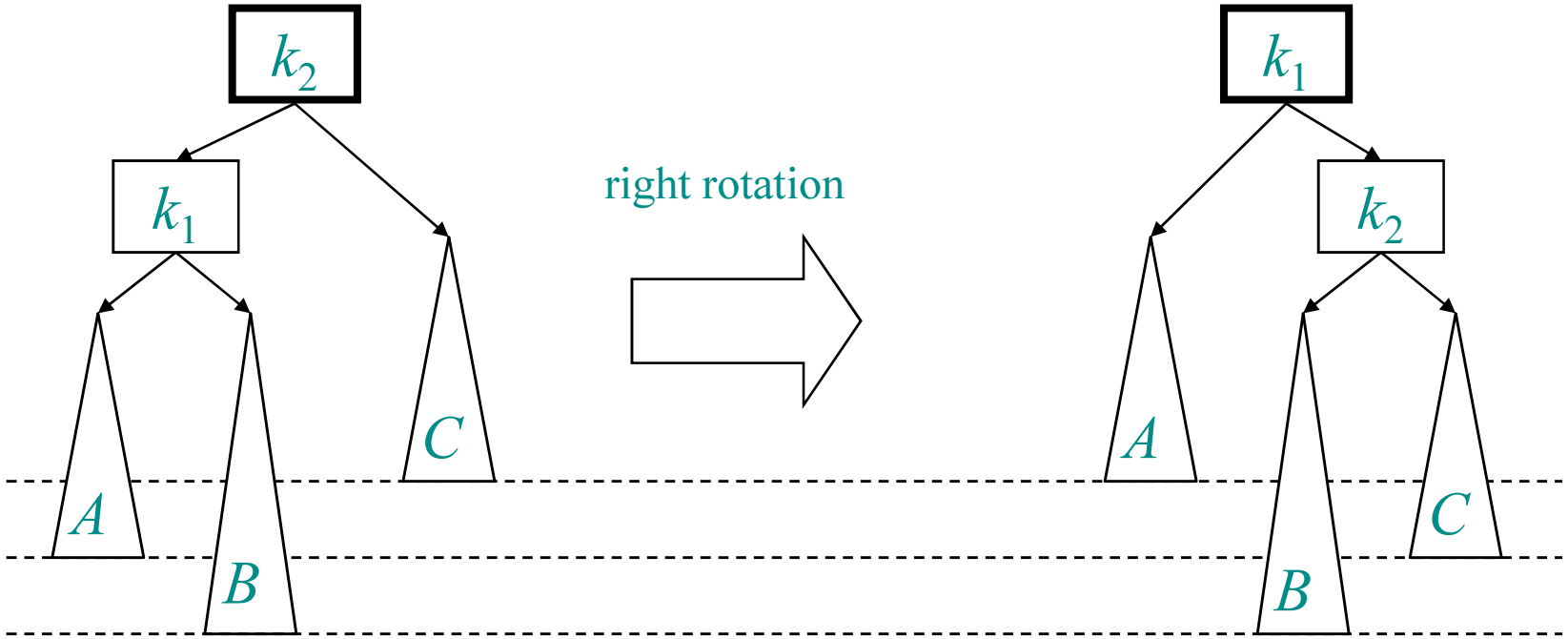
Example (caused by deletion)

- Deleting X and performing a single rotation:



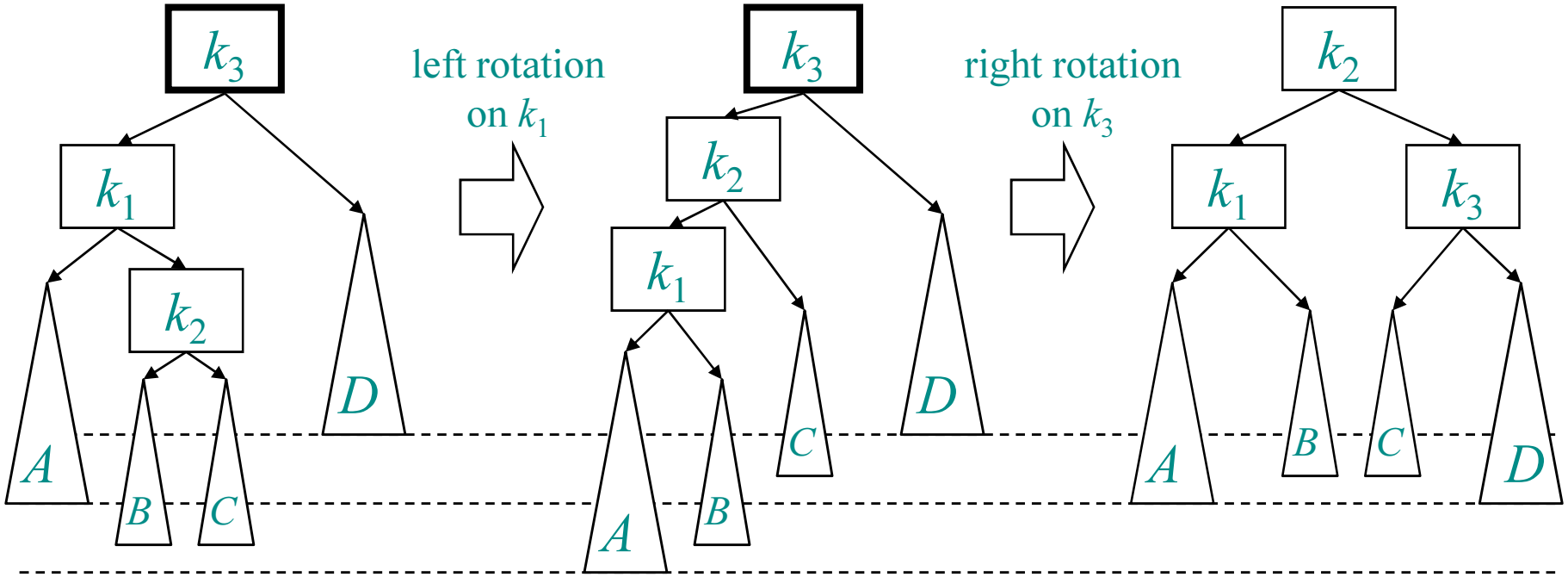
- For the rotation, k_1 is node A, and k_2 is node B. We make k_2 the root, and k_1 its left son.
- Notice that the tree height has changed after the rotation (since it was a delete operation).

Fixing case 2 - first try...



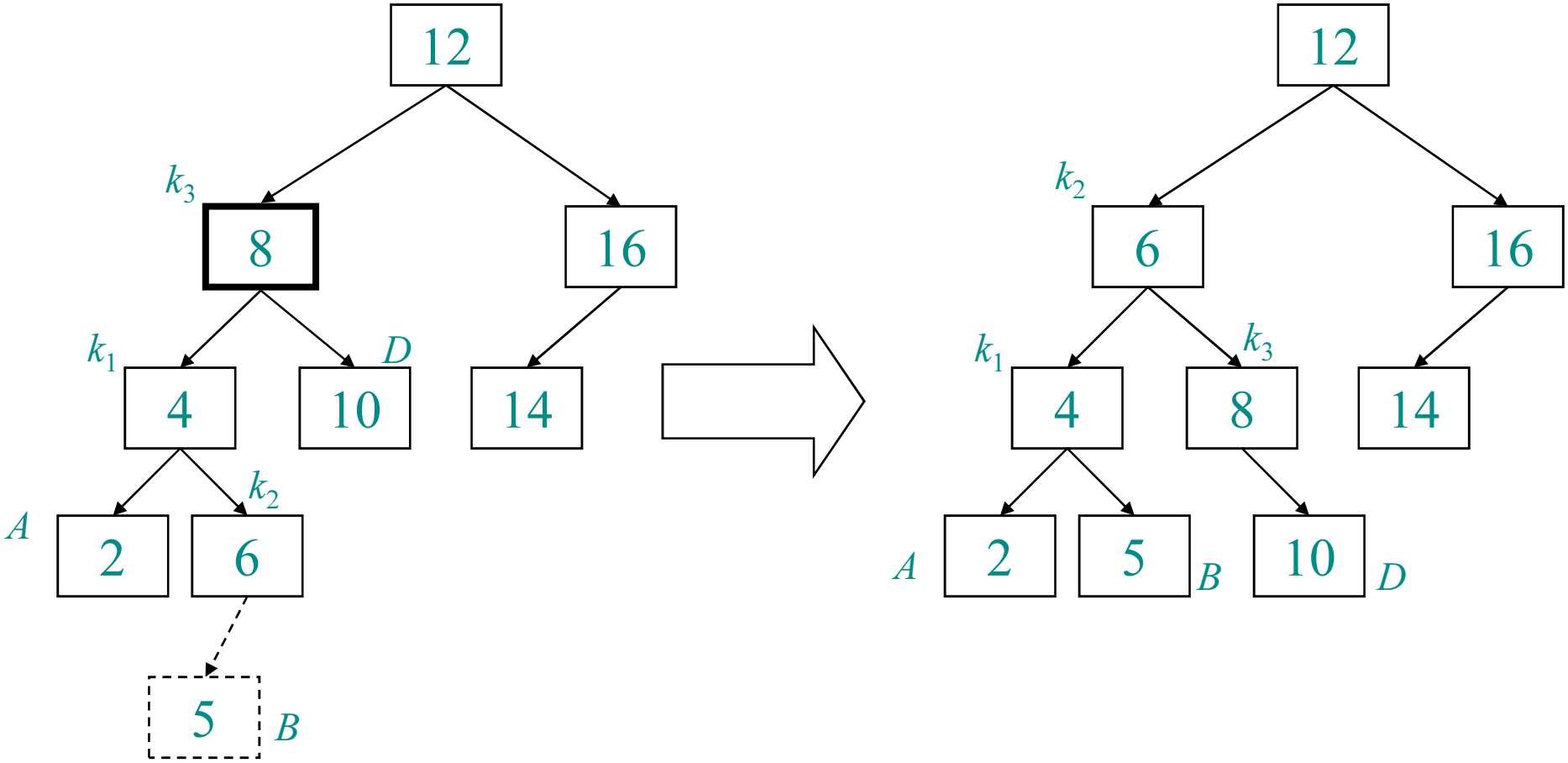
Single rotation doesn't help - the tree is still not balanced!

Double Rotation to fix case 2

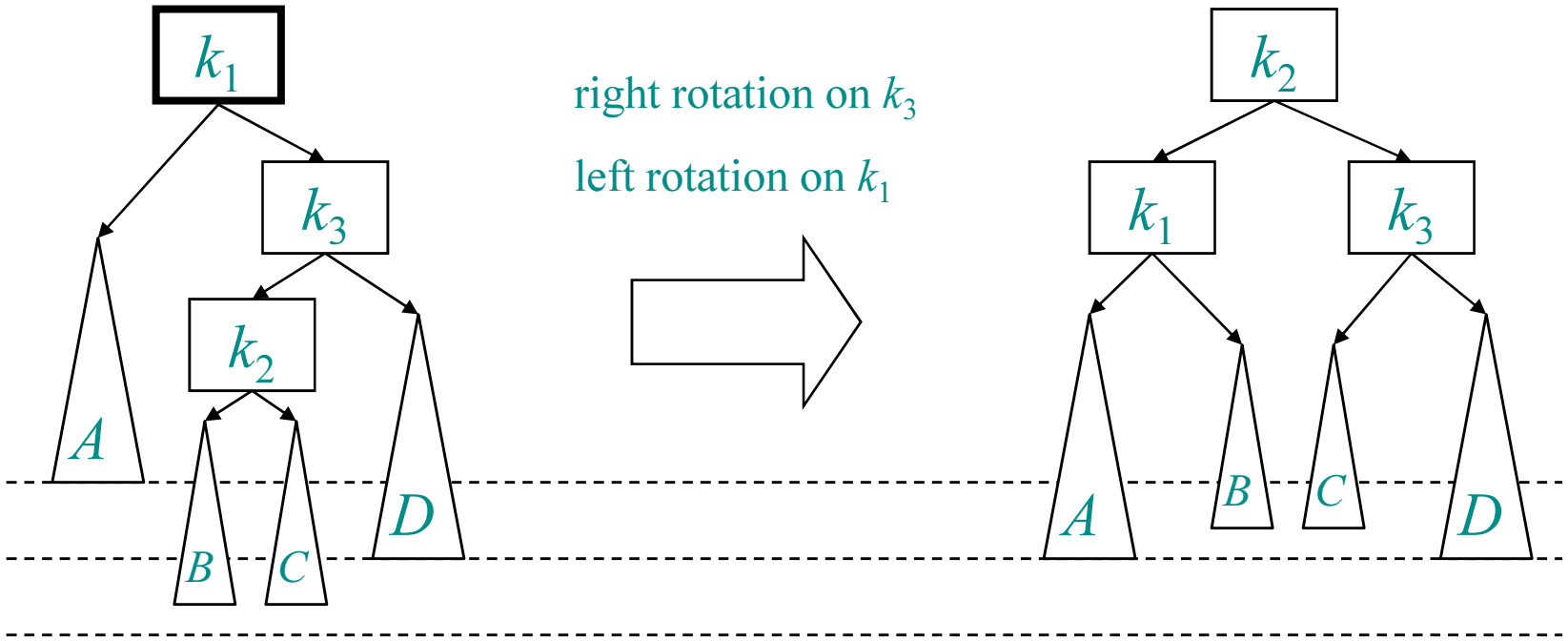


- After insertion - original height (of the root) stays the same.

Example (caused by insertion)

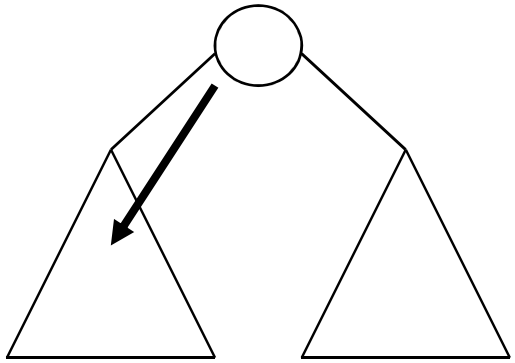


Double Rotation to fix case 3



Insert and delete operations

- First, we insert/delete the element, as in regular binary search tree, and then we re-balance.
- Observation: only nodes on the path from the root to the leaf we changed may become unbalanced.



If we went left from the root, then the right subtree was not changed, thus it remains balanced.
This continues when we go down the tree.

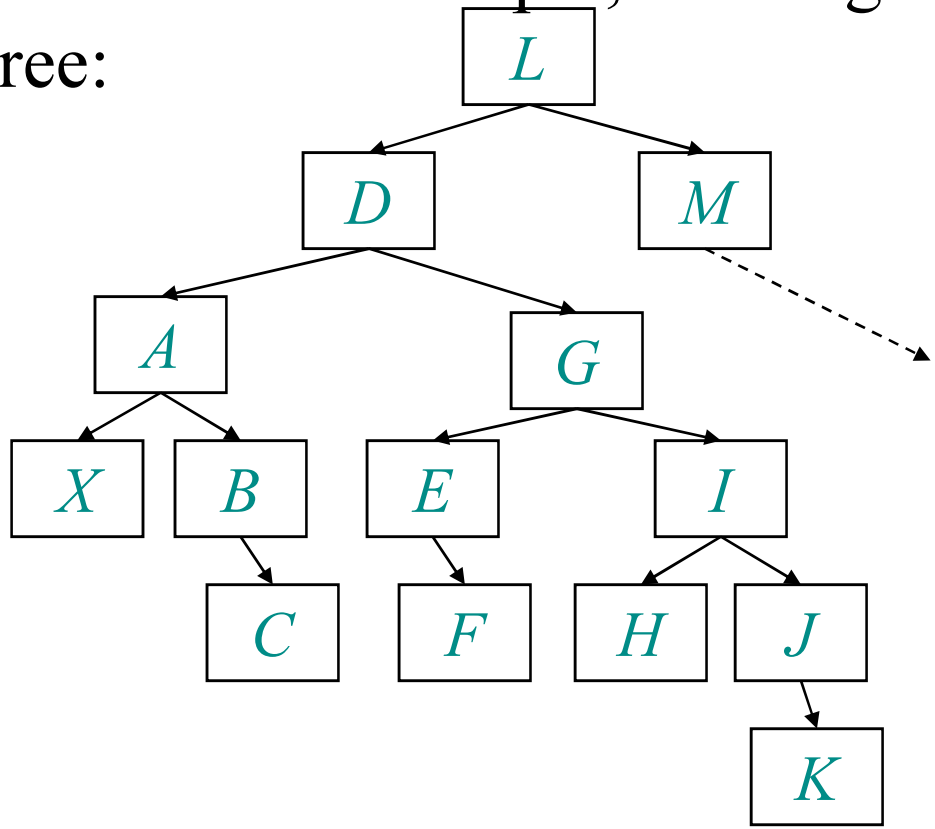
Insert and delete operations (continued)

- After adding/deleting a leaf, start to go up back to the root, and while going up, re-balance every node on the way (if needed). The path is $O(\log n)$ long, and each node balance takes $O(1)$, thus the total time for every operation is $O(\log n)$.

- In fact, in the insertion we can do better - after the first balance (when going up), the subtree that was balanced has height as before, so all higher nodes are now balanced again. We can find this node in the pass down to the leaf, so one pass is enough.

Delete requires two passes

- In more sophisticated balanced trees (like red-black and B-trees), delete also requires one pass. Here this is not the case. For example, deleting *X* in the following tree:



A note about implementation

- Converting an algorithm to a real program requires much thought. When done correctly, many bugs are avoided.
- Important principles:
  - Do it as general as possible, without cut & paste. Although more complicated to design, it will reduce your total work time.
  - Methods should be short and simple. If some method becomes too complicated, you missed something, and this is a sure bug!

Excercise

- 13.1-5, 13.2-3, 13.2-4, 13.3-3, 13.4-6