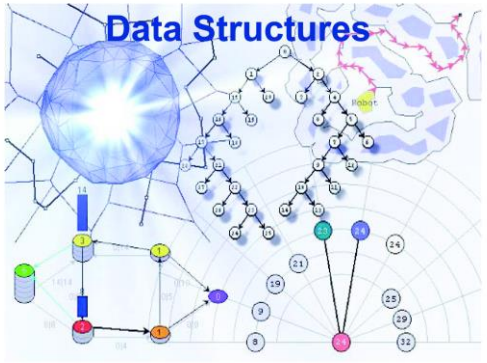
BBM 201
DATA STRUCTURES

Lecture 8:
Balanced Trees



Balanced Search Trees

- The height of a binary search tree is sensitive to the order of insertions and deletions.
  - The height of a binary search tree is between $\lceil \log_2(N+1) \rceil$ and N.
  - So, the worst case behavior of some BST operations are O(N).
- There are various search trees that can retain their balance despite insertions and deletions.
  - AVL Trees
  - 2-3 Trees
  - 2-3-4 Trees
  - Red-Black Trees
- In these height balance search trees, the run time complexity of insertion,deletion and retrieval operations will be $O(\log_2 N)$ at worst case.

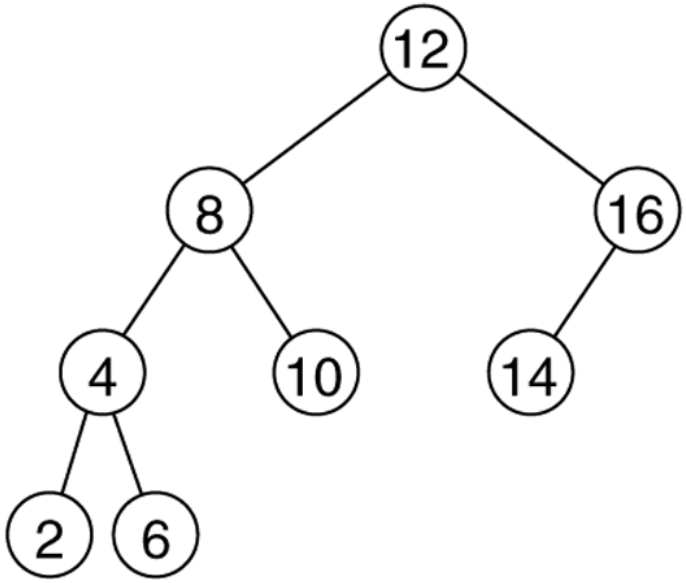
AVL Trees

- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

Definition:

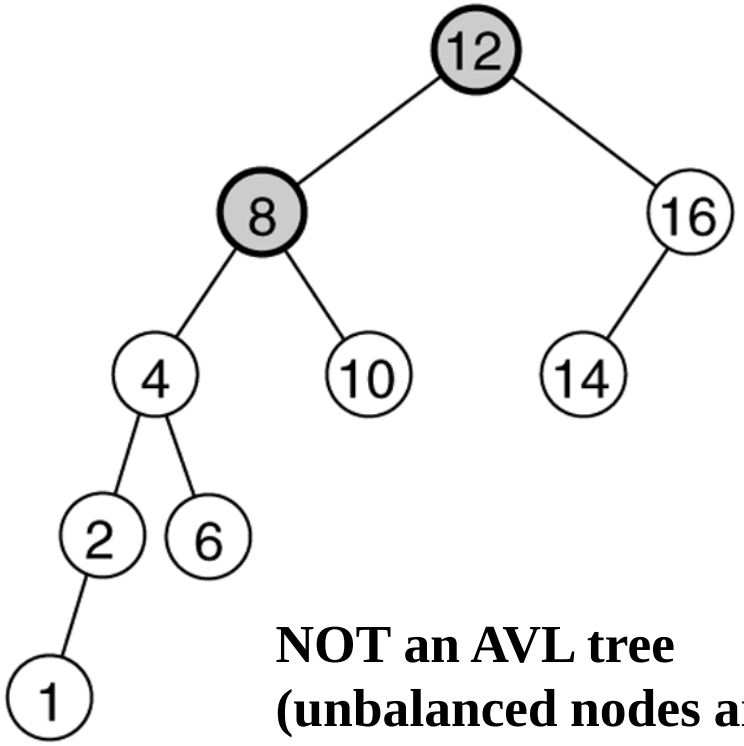
An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.

AVL Tree Examples



an AVL tree

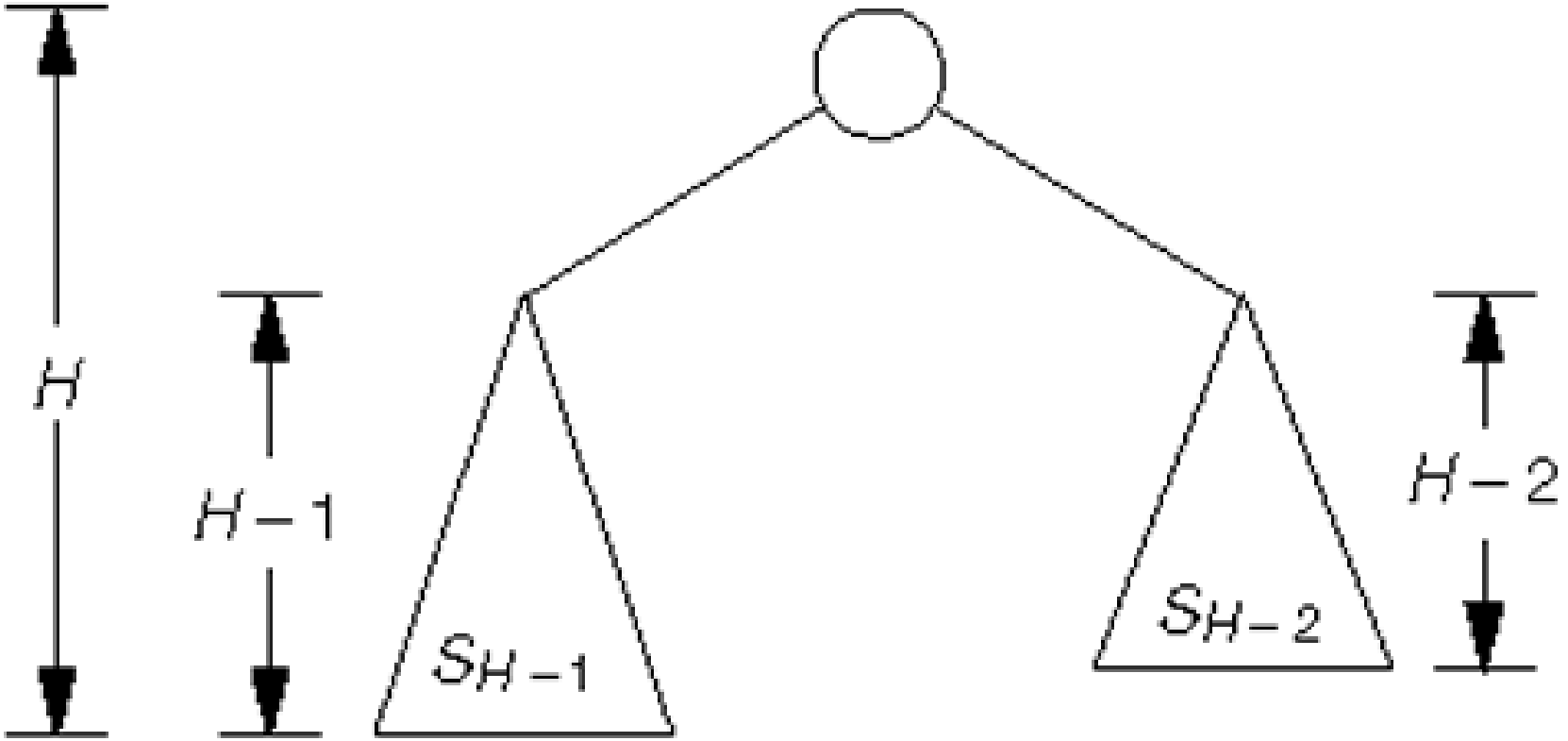
(a)



NOT an AVL tree (unbalanced nodes are darkened)

(b)

A Minimum Tree of height *H*



AVL Tree Properties

- The depth of a typical node in an AVL tree is very close to the optimal $log_2 N$.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or delete) in an AVL tree could destroy the balance.
  - ➔ It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

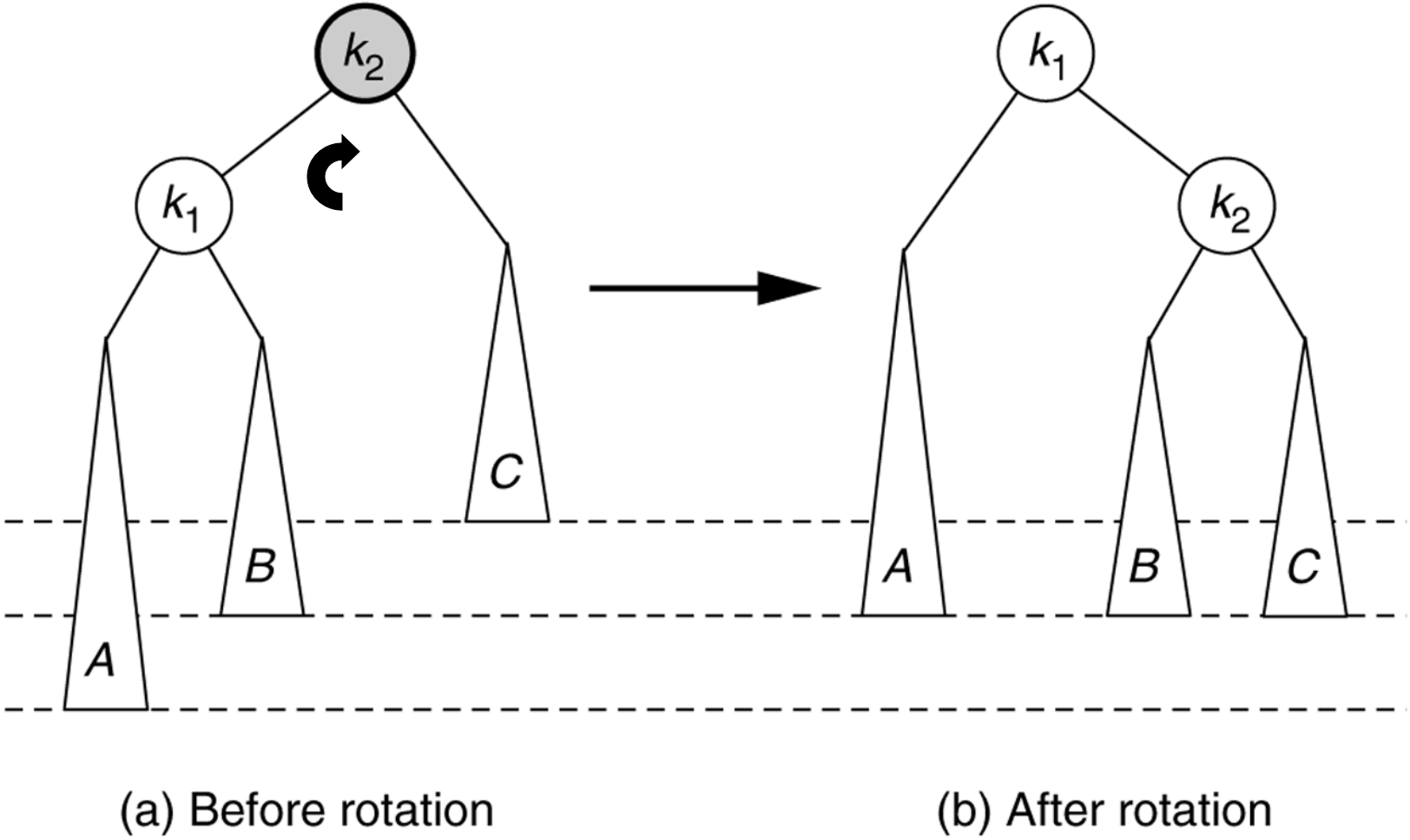
Balance Operations

- Balance is restored by tree ***rotations***.
- There are 4 cases that we might have to fix.
  1. Single Right Rotation
  2. Single Left Rotation
  3. Double Right-Left Rotation
  4. Double Left-Right Rotation

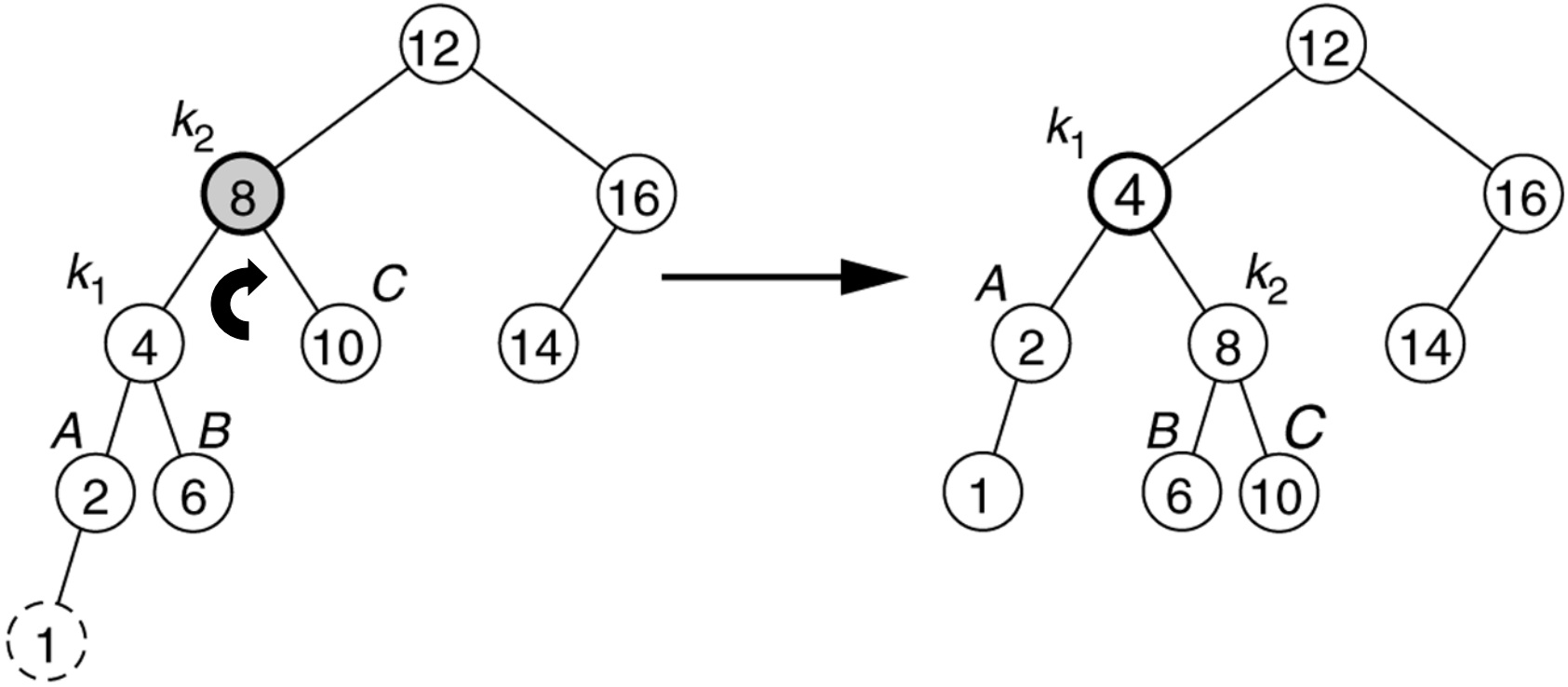
Single Rotation

- A single rotation switches the roles of the parent and child while maintaining the search order.
- We rotate between a node and its child (left or right).
  - Child becomes parent.
  - Parent becomes right child in case 1, left child in case 2.
- The result is a binary search tree that satisfies the AVL property.

Case 1 – Single Right Rotation



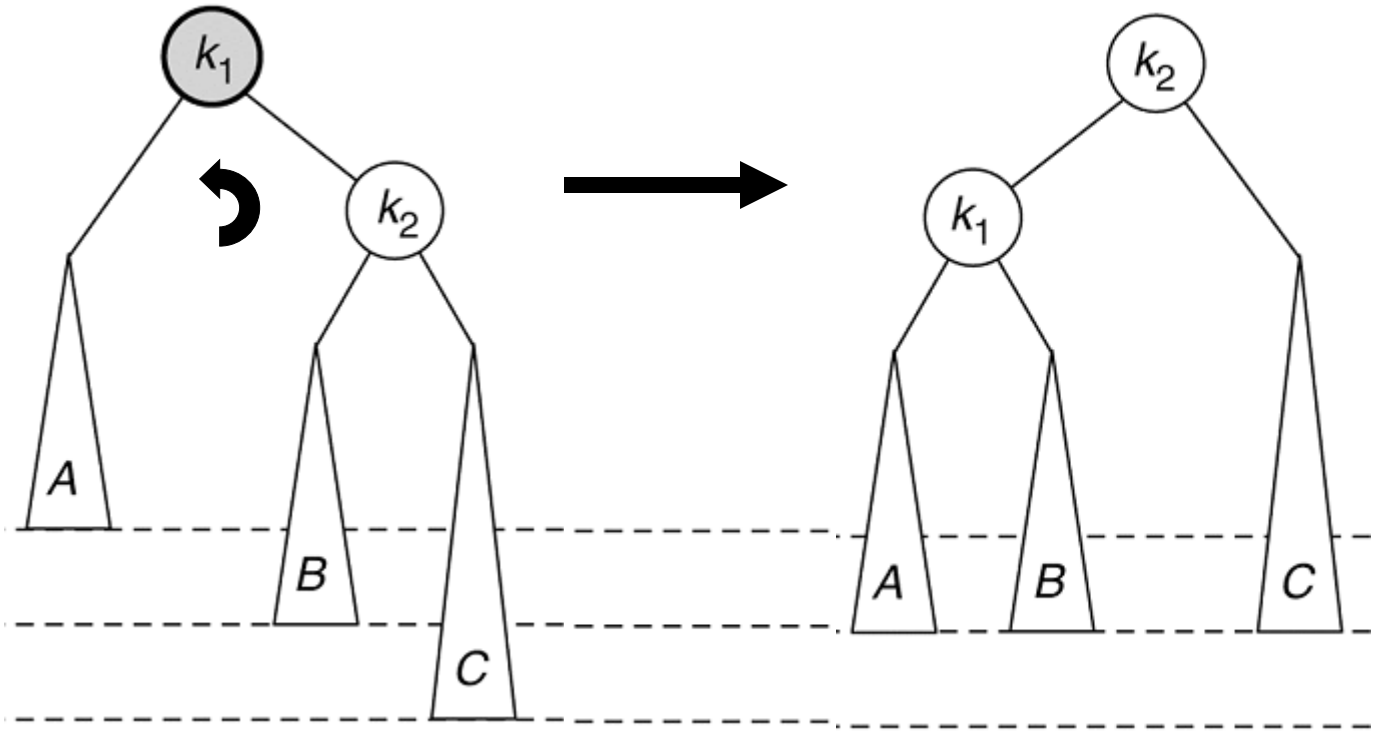
Case 1 – Single Right Rotation – Example



(a) Before rotation

(b) After rotation

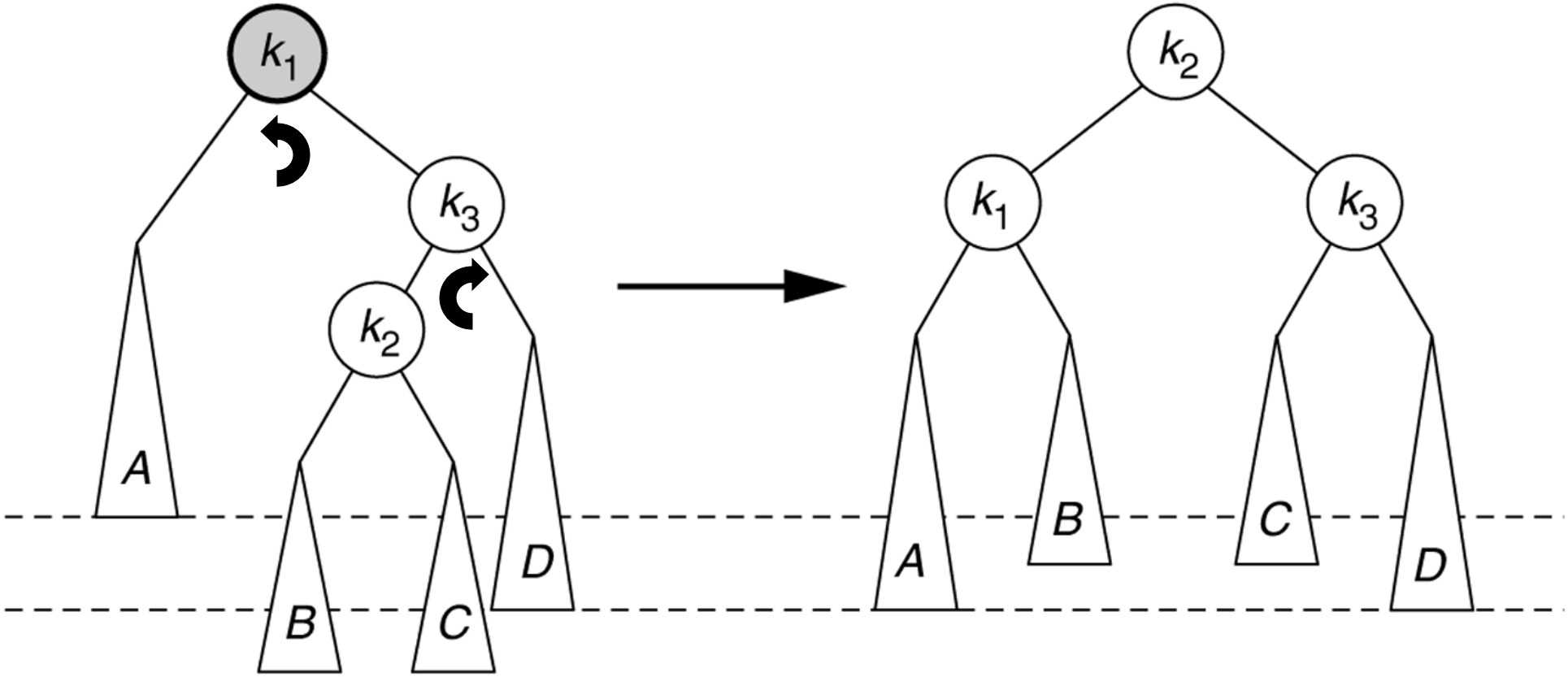
Case 2 – Single Left Rotation



Before Rotation

After Rotation

Case 3 –Double Right-Left Rotation

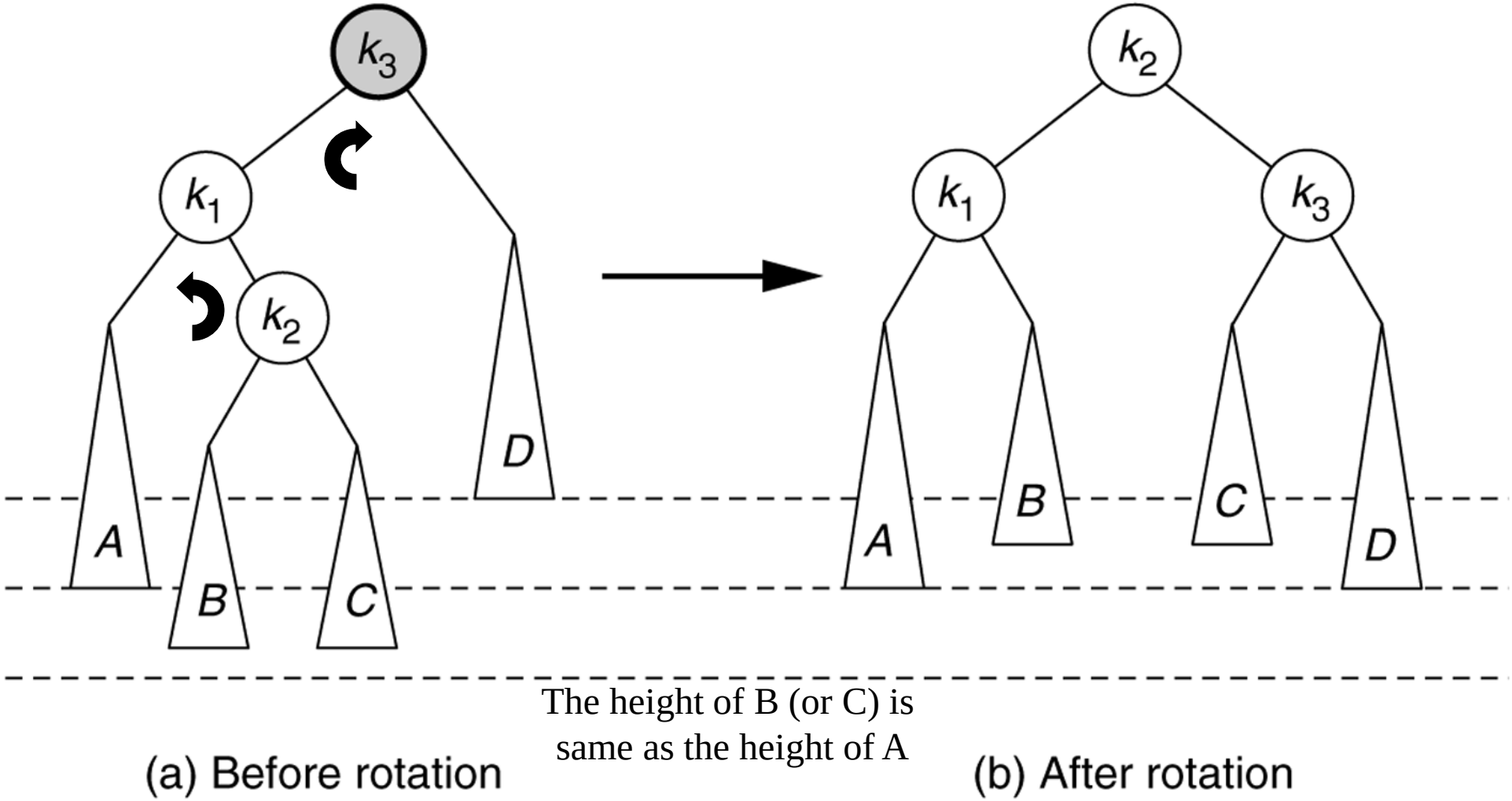


The height of B (or C) is same as the height of D

(a) Before rotation

(b) After rotation

Case 4 –Double Left-Right Rotation

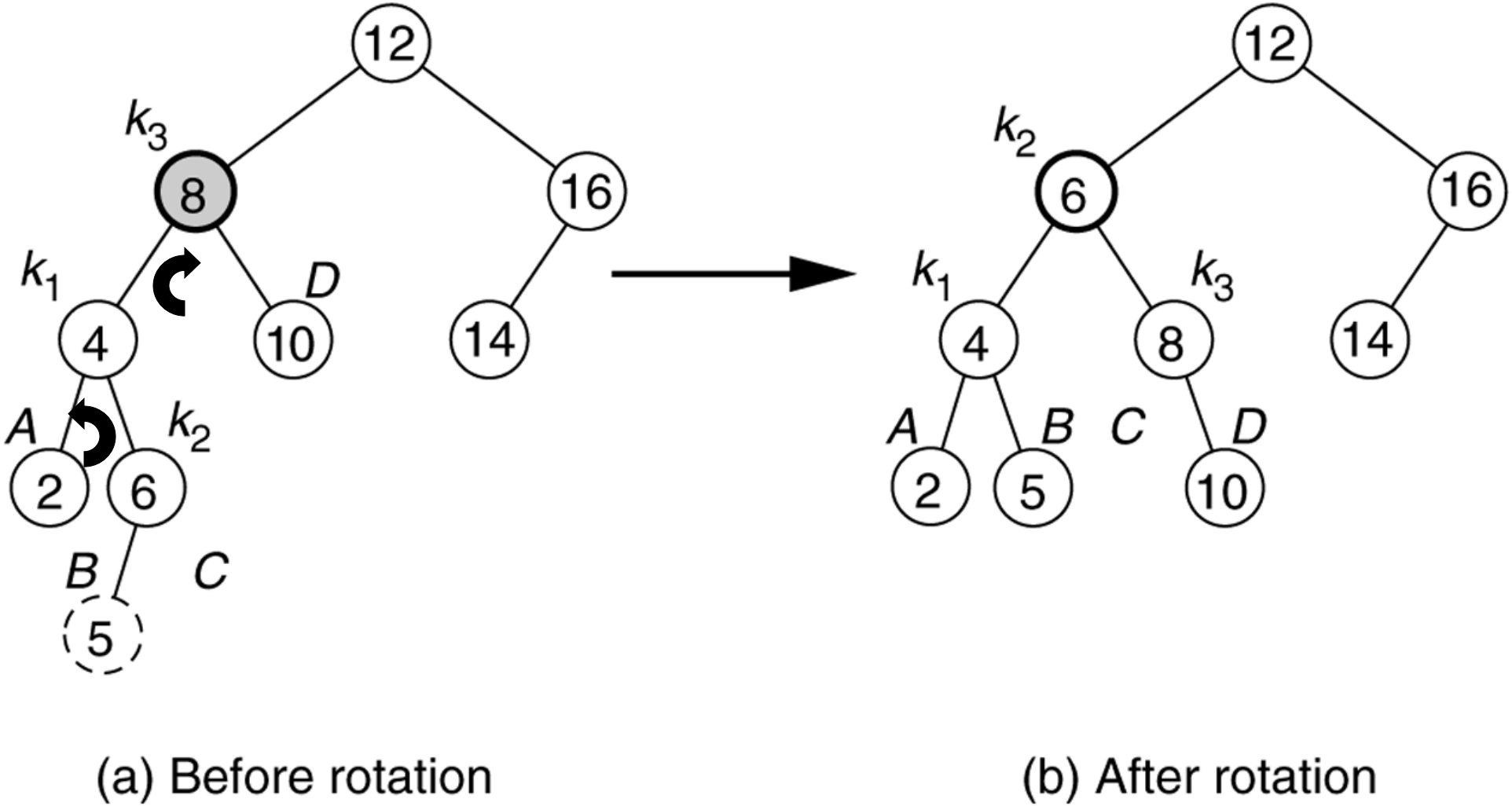


The height of B (or C) is same as the height of A

(a) Before rotation

(b) After rotation

Case 4 –Double Left-Right Rotation – Example



(a) Before rotation

(b) After rotation

Insertion – Analysis

- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus the rotation takes O(1) time.
- Hence insertion is O(logN)

AVL Tree -- Deletion

- Deletion is more complicated.
- Deletion of a node x from an AVL tree requires the same basic ideas, including single and double rotations, that are used for insertion.
- We may need more than one rebalance on the path from deleted node to root.
- Deletion is O(logN)

AVL Tree – Deletion Method

- First delete the node (to be deleted) same as the deletion in a binary search tree.
  - ***Leaf Node :*** Delete the node
  - ***Node with Single Child:*** Delete the node, the parent of the deleted node points the single child of the deleted node after deletion.
  - ***Node with Two Children:*** Find the inorder successor of the node (to be deleted) and swap them. Then delete the inorder successor.

- Walk through from the deleted node back to the root, and rebalance the nodes on the path if it is required.
- We'll trace the effects of this deletion on height through all the nodes on the path from the deleted node *x* back to the root.

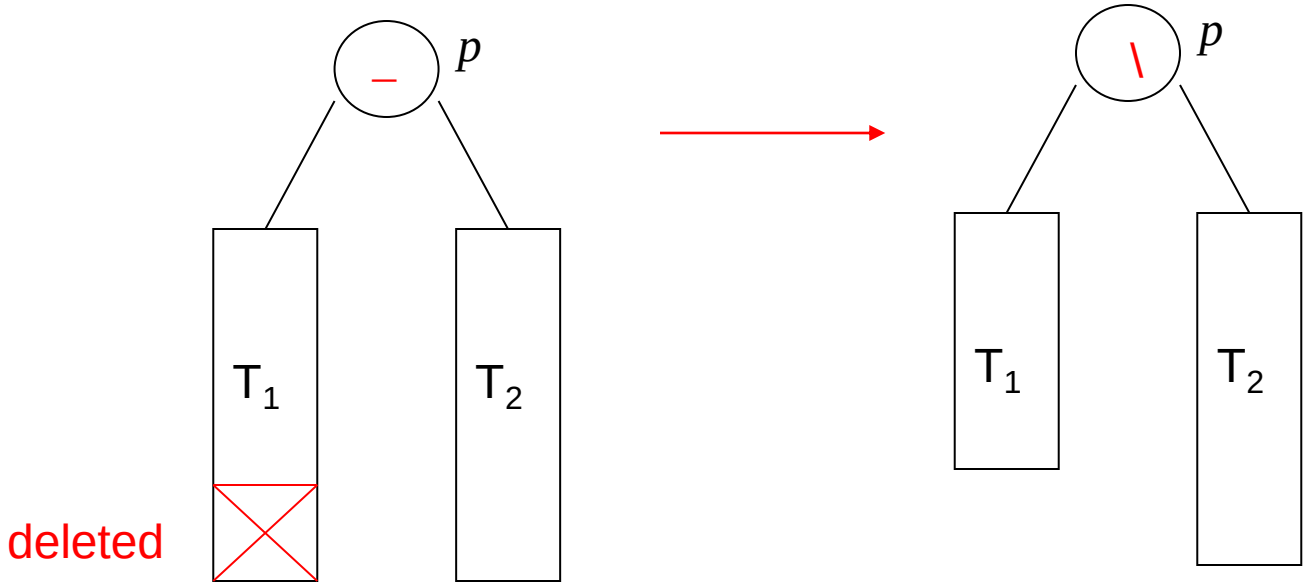
AVL Tree – Deletion Method (cont.)

- A boolean variable shorter shows if the height of a subtree has been shortened.
- Each node of the AVL tree is associated a ***balance factor.***
  - ***left high*** -- left subtree has height greater than right subtree
  - ***right high*** -- right subtree has height greater than left subtree
  - ***equal*** -- left and right subtrees have same height
- The action to be taken at each node depends on
  - the value of shorter
  - balance factor of the node
  - sometimes the balance factor of a child of the node.
- shorter is initially true.
- The following steps are to be done for each node *p* on the path from the parent of x to the root, provided shorter remains true.
- When shorter becomes false, the algorithm terminates.

Case 1

Case 1: ***The current node p has balance factor equal.***

- Change the balance factor of *p* to *right-high*.
- shorter becomes false

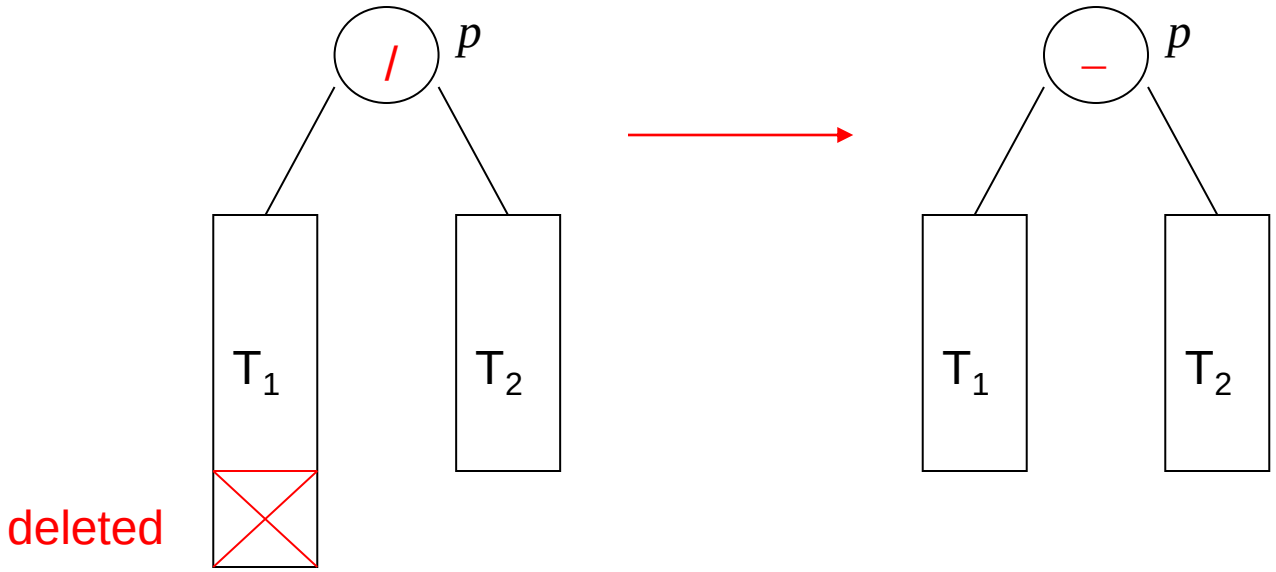


- No rotations
- Height unchanged

Case 2

Case* 2: *The balance factor of p is not equal, and the taller subtree was shortened.

- Change the balance factor of *p* to equal
- Leave shorter true.



- No rotations
- Height reduced

Case 3

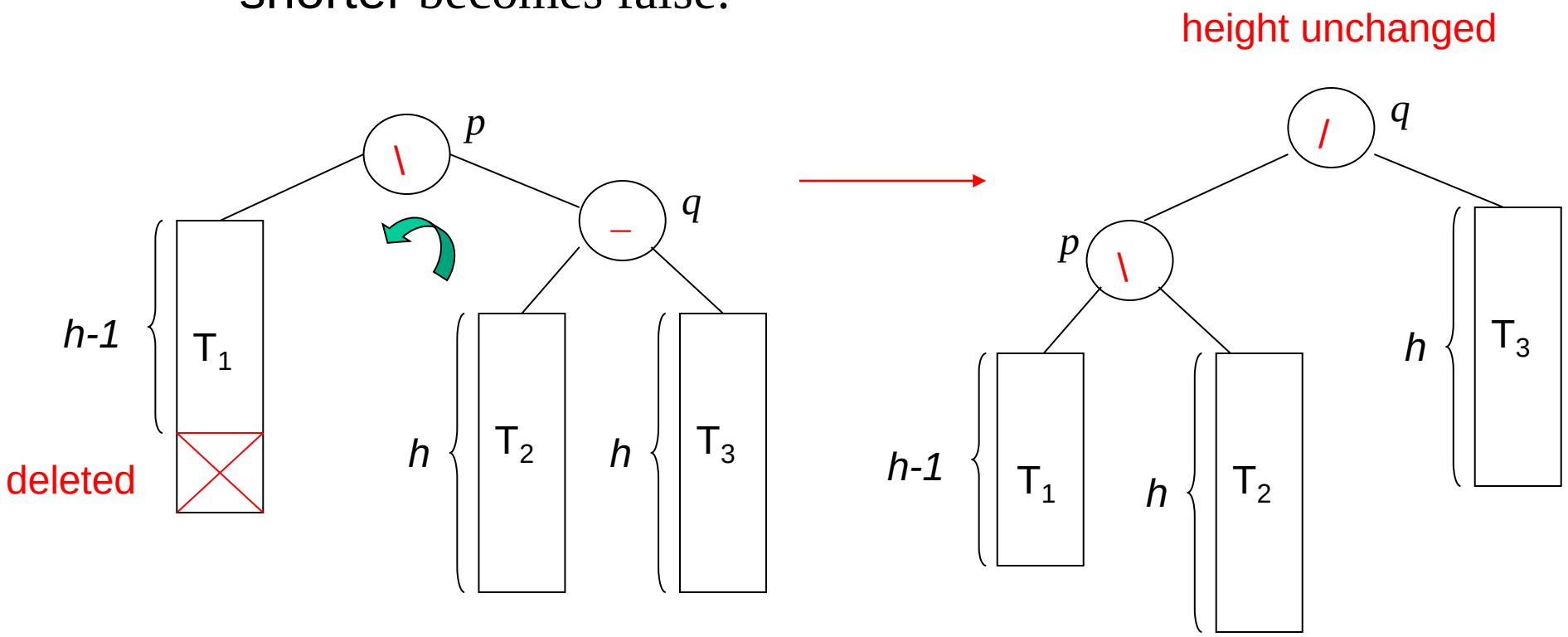
Case* 3: *The balance factor of p is not equal, and the shorter subtree was shortened.

- Rotation is needed.
- Let *q* be the root of the taller subtree of *p*.
- We have three sub-cases according to the balance factor of *q*.

Case 3a

Case* 3*a*: *The balance factor of q is equal.

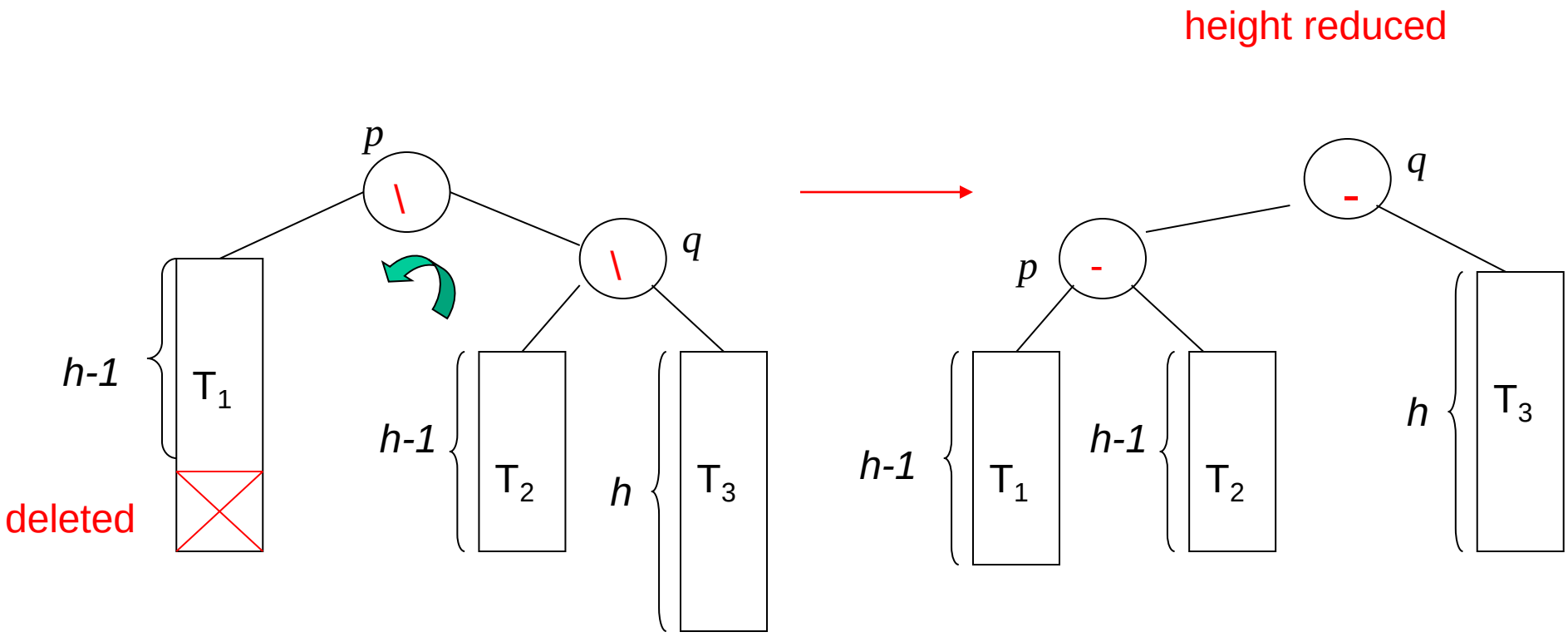
- Apply a single rotation
- shorter becomes false.



Case 3b

Case* 3*b*: *The balance factor of q is the same as that of p.

- Apply a single rotation
- Set the balance factors of *p* and *q* to equal
- leave shorter as true.

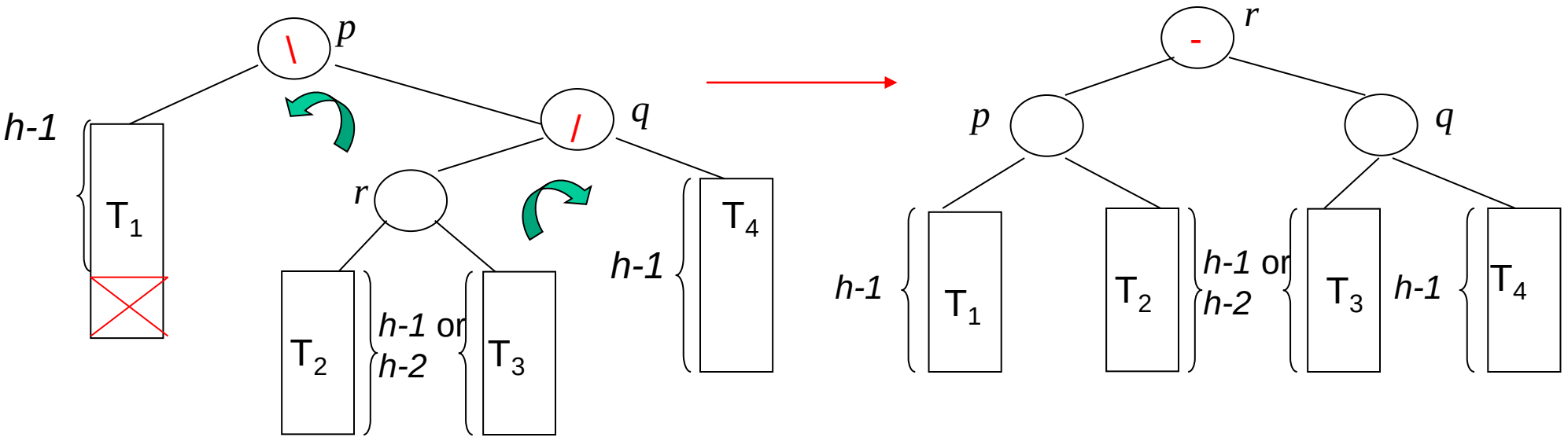


Case 3c

Case 3c: The balance factors of p and q are opposite.

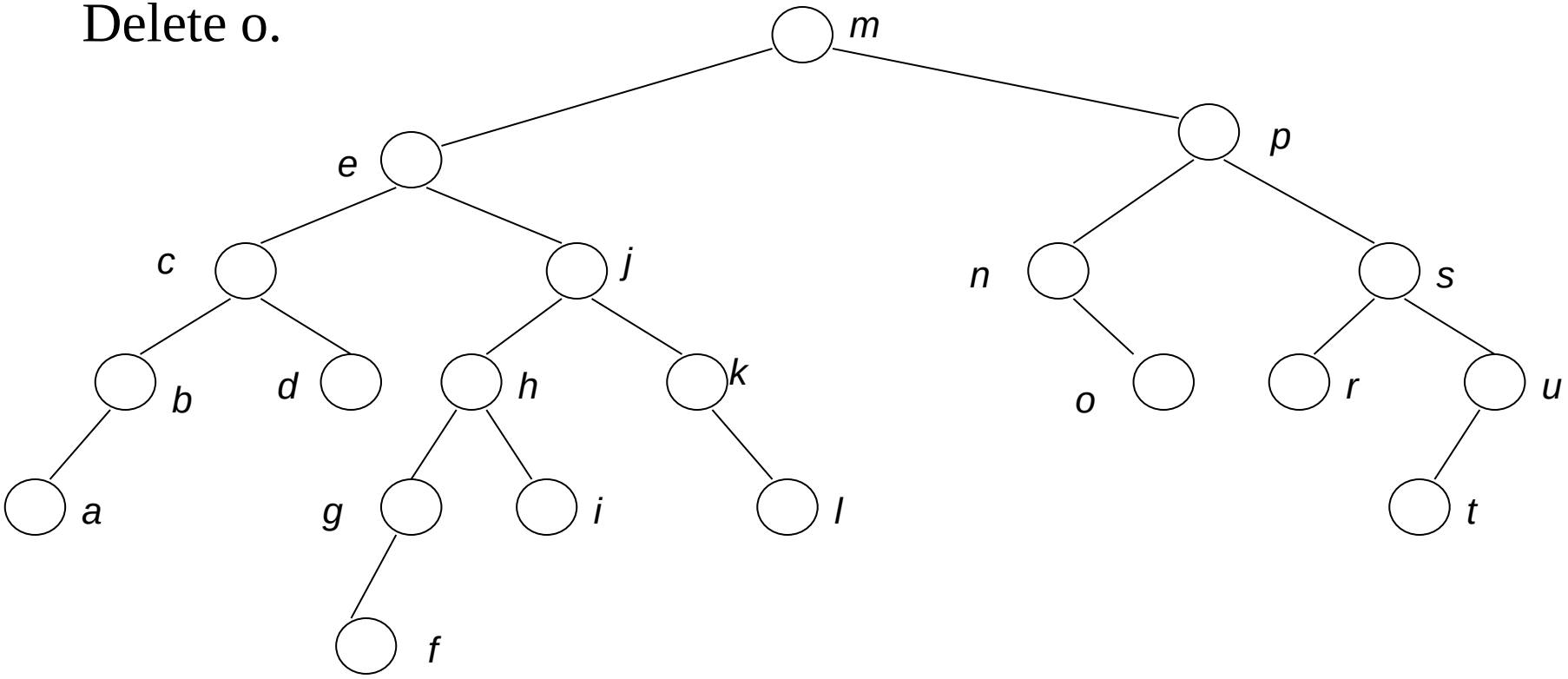
- Apply a double rotation
- set the balance factors of the new root to equal
- leave shorter as true.

height reduced



Example

Delete o.



Worst Cases of AVL Trees

H	minN	logN	H / logN
4	7	2,81	1,42
5	12	3,58	1,39
6	20	4,32	1,39
7	33	5,04	1,39
8	54	5,75	1,39
9	88	6,46	1,39
10	143	7,16	1,40
11	232	7,86	1,40
12	376	8,55	1,40
13	609	9,25	1,41
14	986	9,95	1,41
15	1.596	10,64	1,41
16	2.583	11,33	1,41
17	4.180	12,03	1,41
18	6.764	12,72	1,41
19	10.945	13,42	1,42
20	17.710	14,11	1,42
30	**2.178.308**	**21,05**	**1,42**
40	**267.914.295**	**28,00**	**1,43**
50	**32.951.280.098**	**34,94**	**1,43**

What is the minumum number of nodes in a height N AVL tree?

minN(0) = 0
minN(1) = 1
minN(2) = 2
minN(3) = 4
.

minN(h) = minN(h-1) + minN(h-2) + 1

Maximum height of a N-node AVL tree is less than 1.44 logN