AVL Trees

Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

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 at the end of each insertion and deletion.
  - AVL Trees
  - 2-3 Trees
  - 2-3-4 Trees
  - Red-Black Trees

Binary search tree

- In these height balanced search trees, the run time complexity of insertion, deletion, and retrieval operations is $O(\log_2 N)$ at the 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.

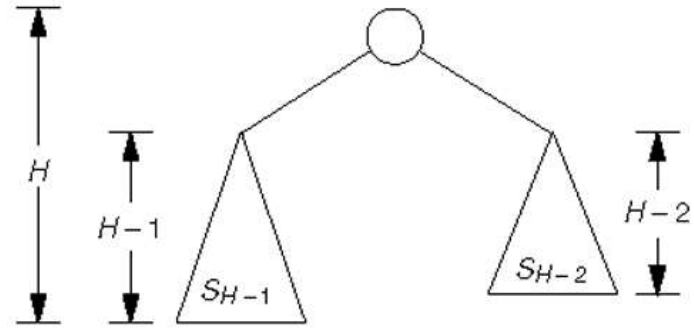
not complete

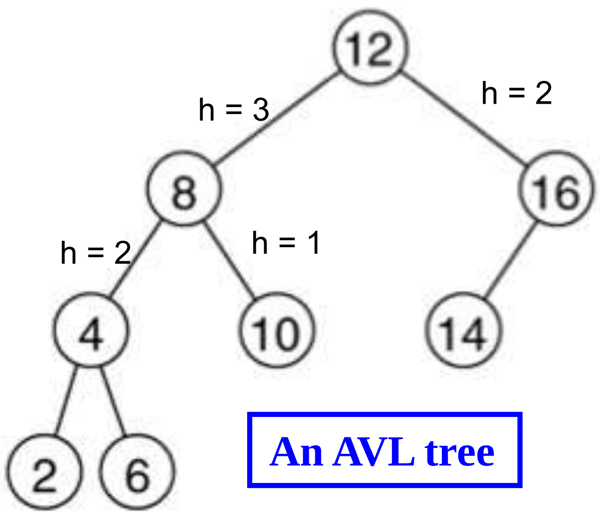
perfect balance -> every path has same height

AVL Trees

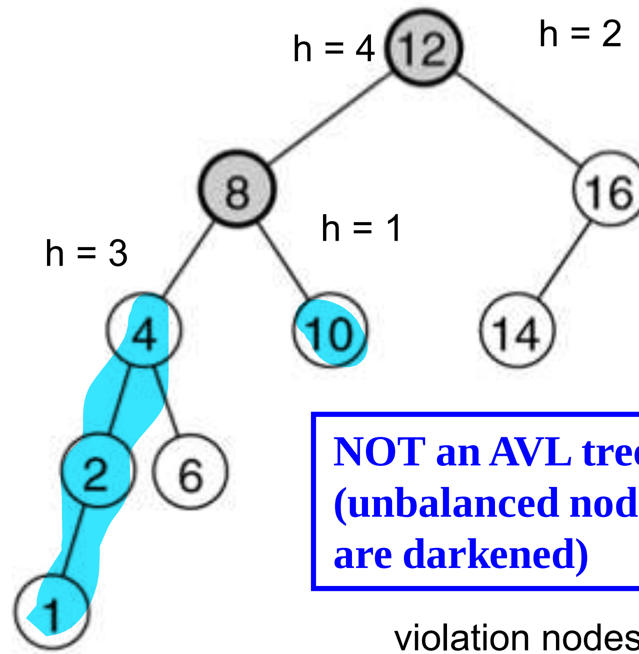
Definition:

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.





An AVL tree



NOT an AVL tree (unbalanced nodes are darkened)

violation nodes

AVL Trees -- Properties

- The depth of a typical node in an AVL tree is very close to the optimal log_2N.

- 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.**

AVL Tree -- Insertions

- Insert is the same as **Binary Search Tree** insertion
- Then, starting from the insertion point, check for balance at each node
- It is enough to perform correction "rotation" only **at the first node** where imbalance occurs
  - On the path from the inserted node to the root.

AVL -- Insertions

- An AVL violation might occur in four possible cases:

1) Insertion into left subtree of left child of node *n*
2) Insertion into right subtree of left child of node *n*
3) Insertion into left subtree of right child of node *n*
4) Insertion into right subtree of right child of node *n*

node n = violation node

- (1) and (4) are mirror cases
- (2) and (3) are mirror cases

- If insertion occurs "**outside**" (1 & 4), then perform **single rotation**.
- If insertion occurs "**inside**" (2 & 3), then perform **double rotation.**

AVL Trees -- Balance Operations

- Balance is restored by tree ***rotations***.

- There are four different cases for rotations:
  1. Single Right Rotation
  2. Single Left Rotation
  3. Double Right-Left Rotation
  4. Double Left-Right Rotation

AVL Trees -- Single Rotation

- A single rotation switches the roles of the parent and the child while maintaining the search order.

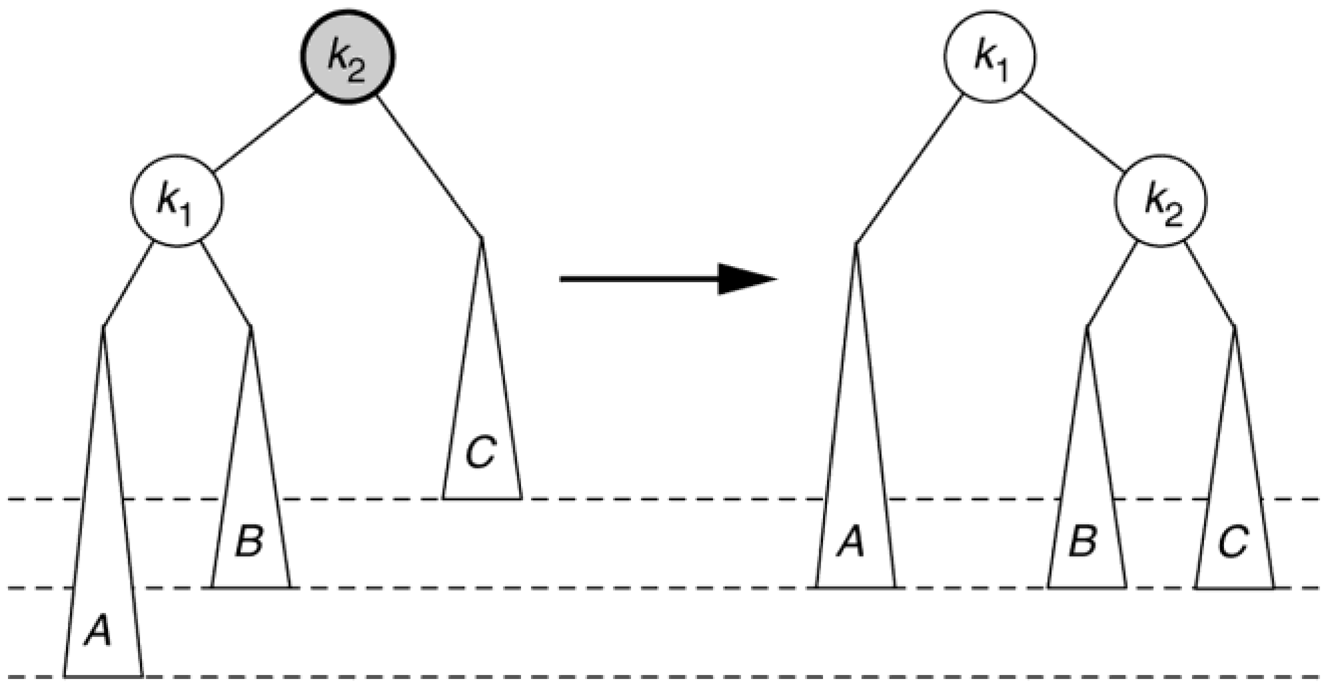
- We rotate between a node and its child (left or right).
  - Child becomes parent
  - Parent becomes right child in Case 1 (single right rotation)

    Parent becomes left child in Case 2 (single left rotation)

- The result is a binary search tree that satisfies the AVL property.

Case 1 -- Single Right Rotation

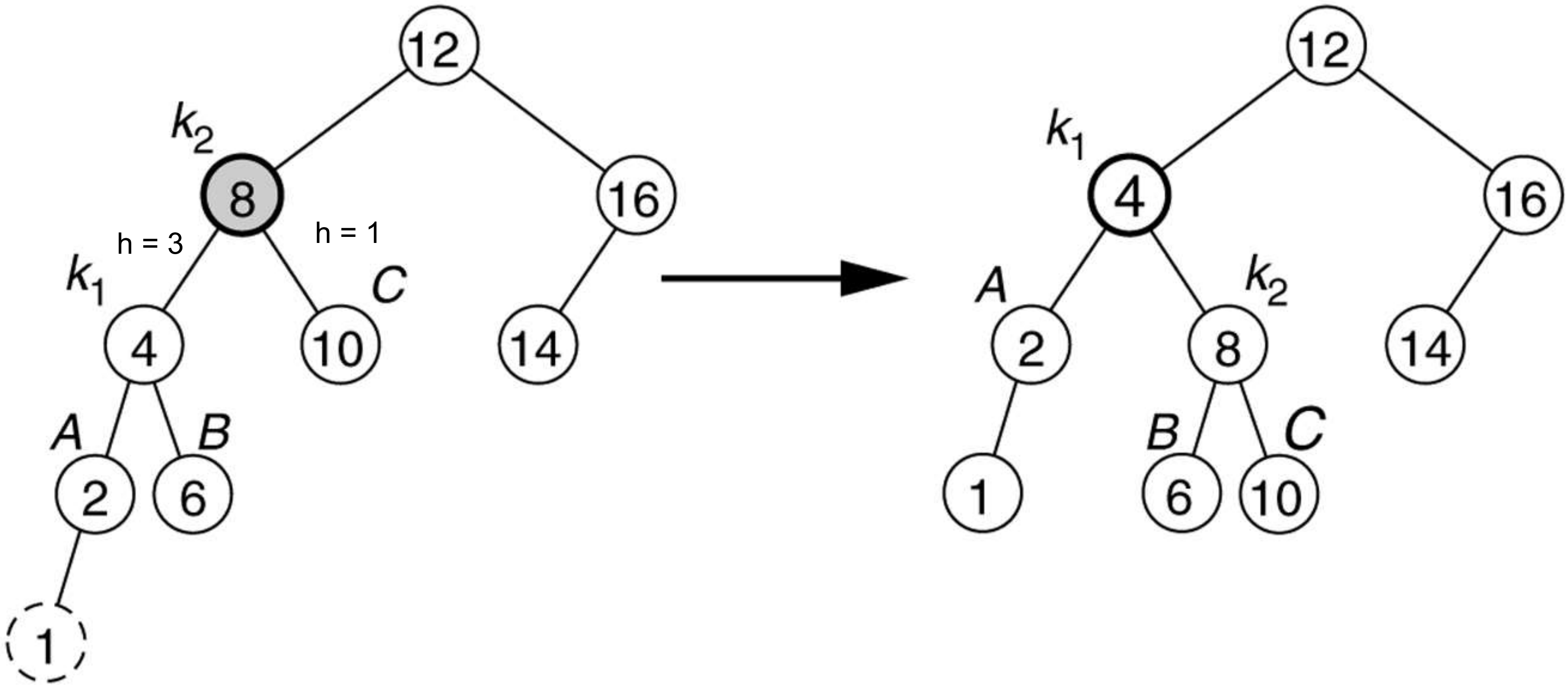
inorder traversal should be same after all rotations (binary search tree)



(a) Before rotation (b) After rotation

Child becomes parent
Parent becomes right child

Case 1 -- Single Right Rotation

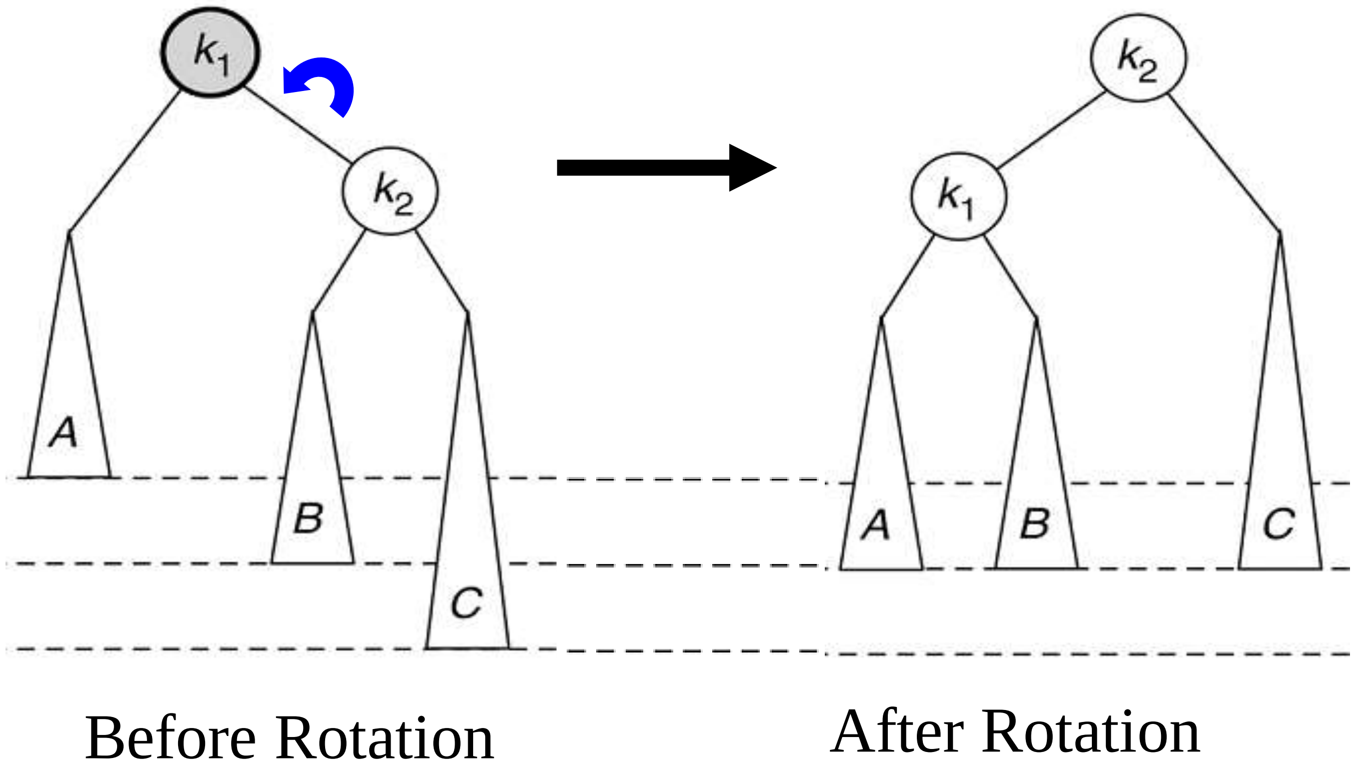


(a) Before rotation

(b) After rotation

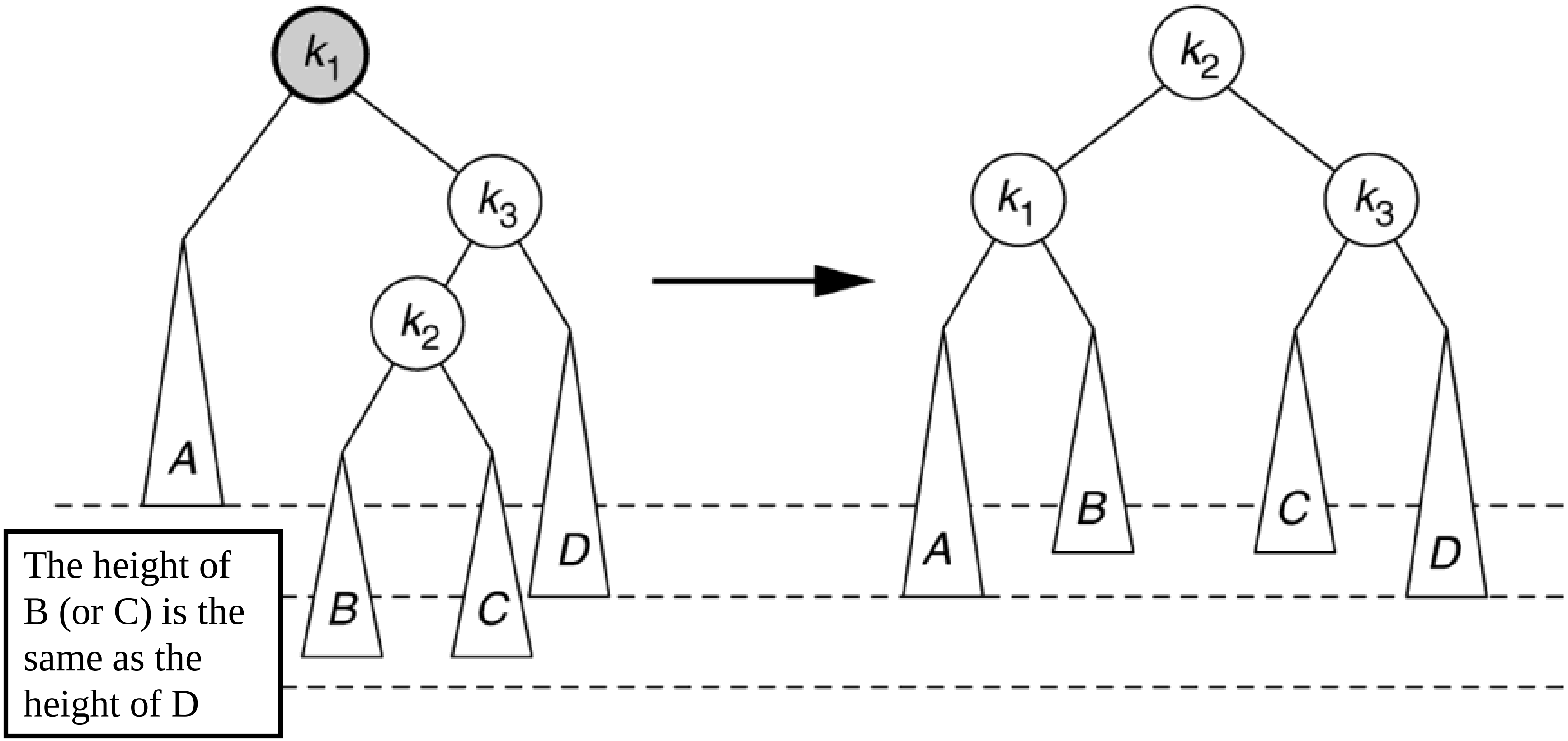
Child becomes parent
Parent becomes right child

Case 2 – Single Left Rotation



Child becomes parent
Parent becomes left child

Case 3 -- Double Right-Left Rotation



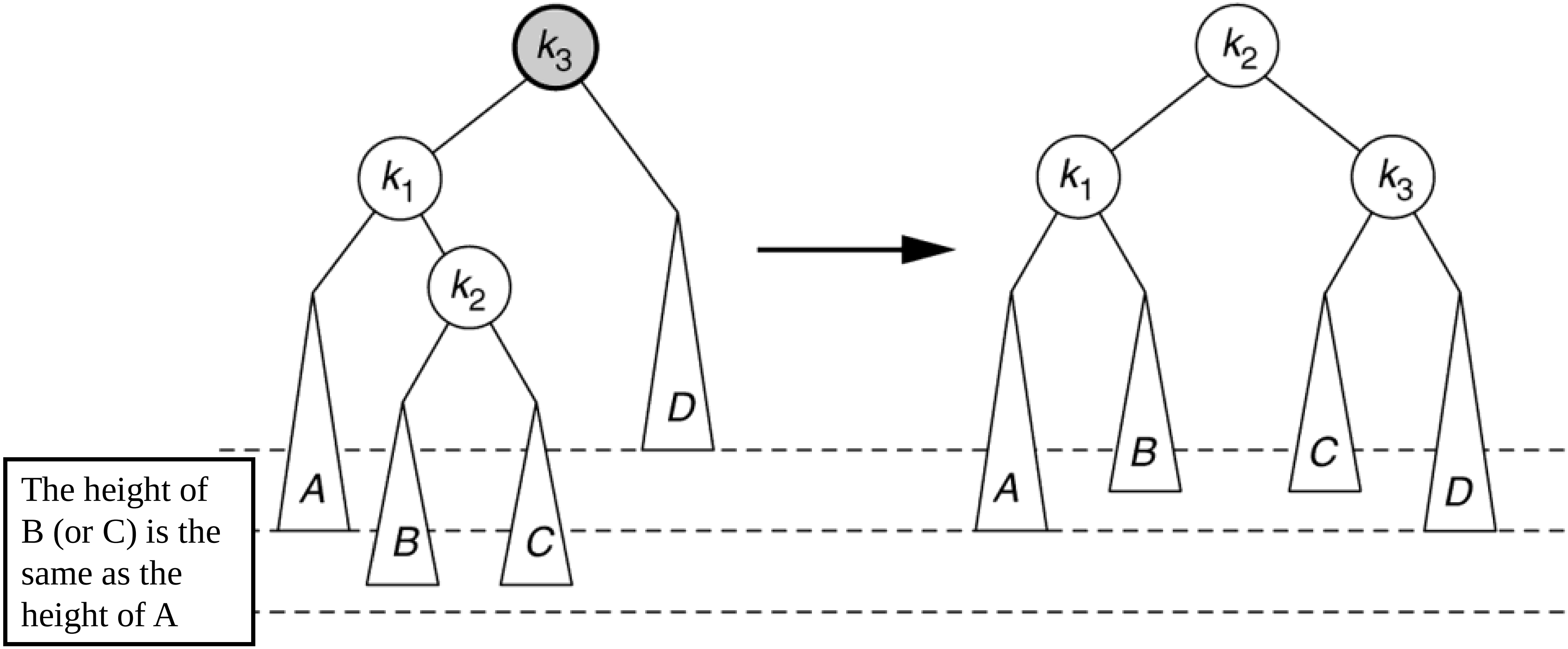
(a) Before rotation

(b) After rotation

First perform single right rotation on k2 and k3
Then perform single left rotation on k2 and k1

(after right rotation
k2 would be parent of k3)

Case 4 -- Double Left-Right Rotation



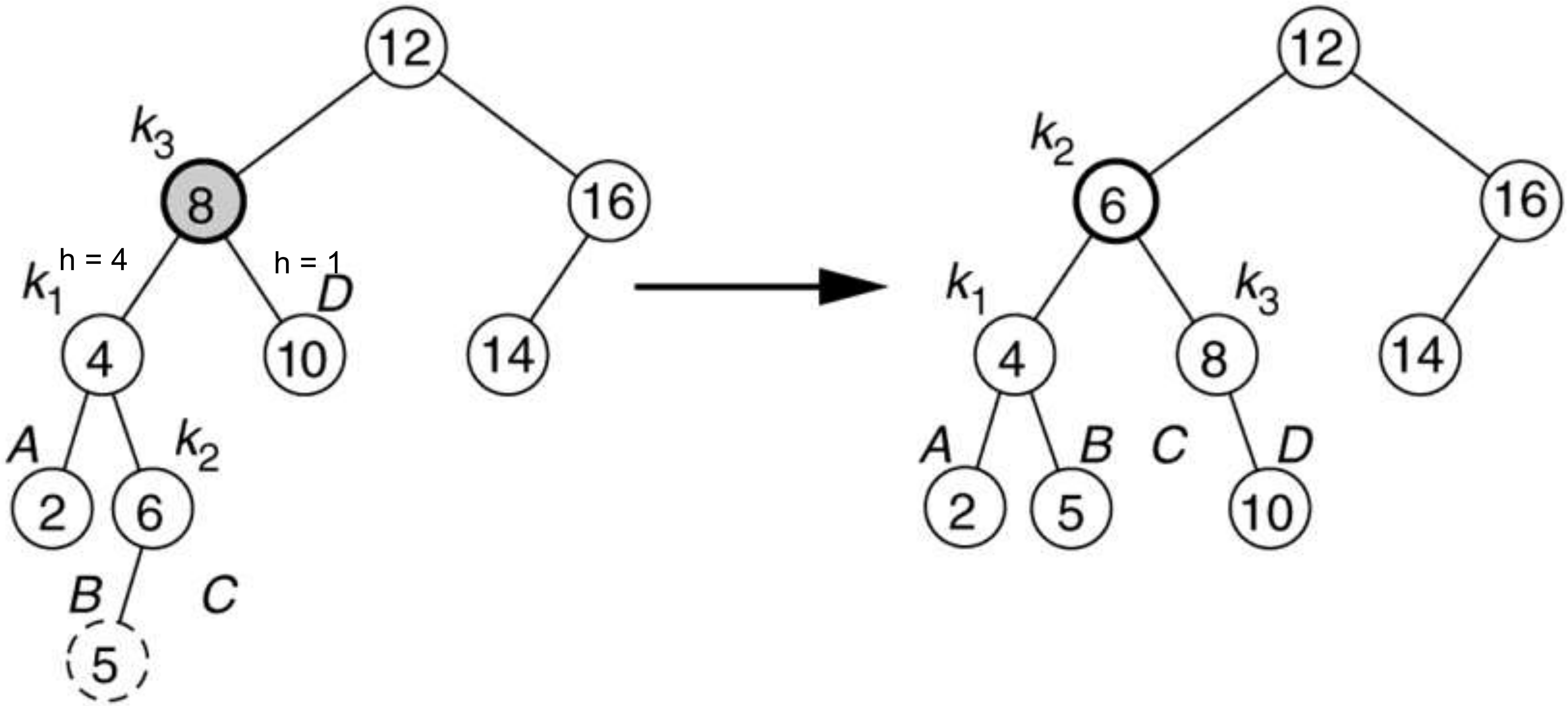
(a) Before rotation

(b) After rotation

First perform single left rotation on k2 and k1
Then perform single right rotation on k2 and k3

after left rotation
k2 would be parent of k1

Case 4 -- Double Left-Right Rotation



(a) Before rotation

(b) After rotation

no violation occurs at k1 and k2

First perform single left rotation on k2 and k1
Then perform single right rotation on k2 and k3

AVL Trees -- Insertion

- It is enough to perform rotation only at the first node
  - Where imbalance occurs
  - On the path from the inserted node to the root.
- The rotation takes O(1) time.
- After insertion, only nodes that are on the path from the insertion point to the root can have their balances changed.
- Hence insertion is O(logN)

AVL Trees -- Insertion

***Exercise*:** Starting with an empty AVL tree, insert the following items

7 6 5 4 3 2 1 8 9 10 11 12

Check the following applet for more exercises.

http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

AVL Trees -- Deletion

- Deletion is more complicated.
  - It requires both single and double rotations
  - We may need more than one rebalance operation (rotation) on the path from the deleted node back to the root.
- Steps:
  - First delete the node the same as deleting it from a binary search tree
    - Remember that a node can be either ***a leaf node*** or ***a node with a single child*** or ***a node with two children***
  - Walk through from the deleted node back to the root and rebalance the nodes on the path if required
    - Since a rotation can change the height of the original tree
- Deletion is O(logN)
  - Each rotation takes O(1) time
  - We may have at most h (height) rotations, where h = O(logN)

AVL Trees -- Deletion

- For the implementation
  - We have a `shorter` flag that shows if a subtree has been shortened
  - Each node is associated with a `balance factor`
    - ***left-high*** the height of the left subtree is higher than that of the right subtree
    - ***right-high*** the height of the right subtree is higher than that of the left subtree
    - ***equal*** the height of the left and right subtrees is equal

- In the deletion algorithm
  - Shorter is initialized as true
  - Starting from the deleted node back to the root, take an action depending on
    - The value of shorter
    - The balance factor of the current node
    - Sometimes the balance factor of a child of the current node
  - Until shorter becomes false

AVL Trees -- Deletion

Three cases according to the balance factor of the current node

1. The balance factor is equal

   ➔ ***no rotation***

2. The balance factor is not equal and the taller subtree was shortened

   ➔ ***no rotation***

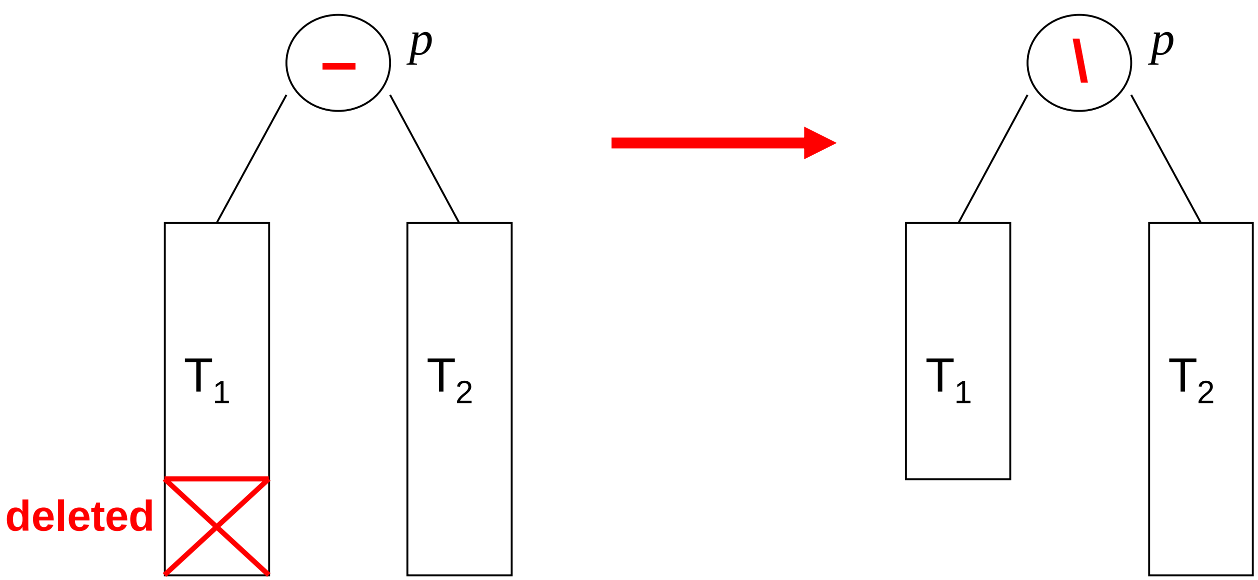
3. The balance factor is not equal and the shorter subtree was shortened

   ➔ ***rotation is necessary***

AVL Trees -- Deletion

<u>Case 1</u>: *The balance factor of p is equal.*

- Change the balance factor of *p* to *right-high* (or *left-high*)
- Shorter becomes false



No rotations

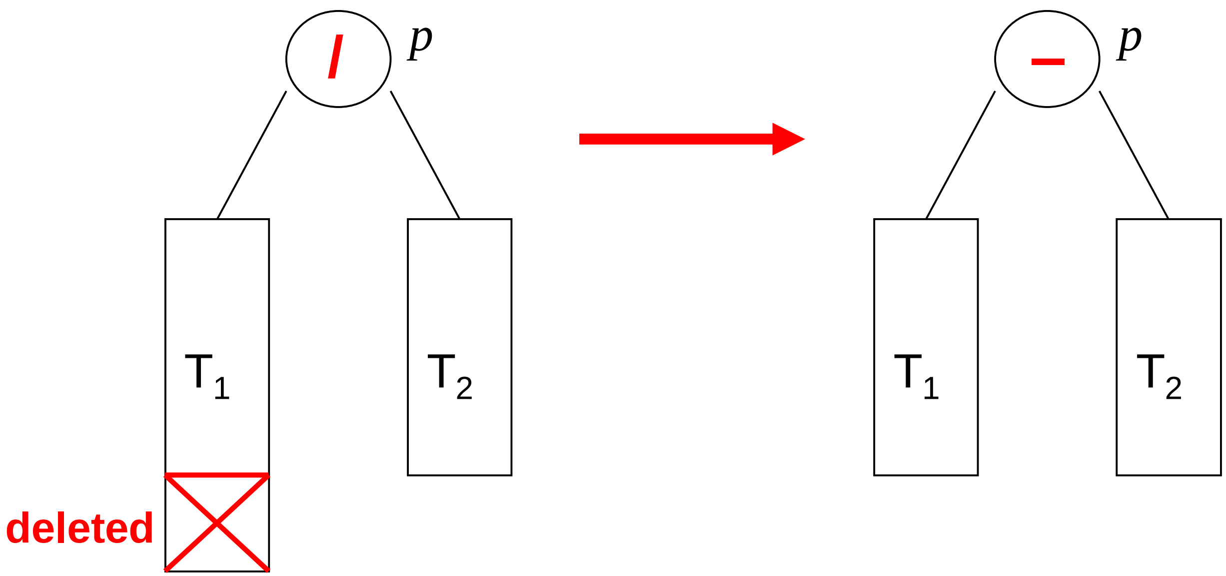
Height unchanged

height = 1 + T2 still

AVL Trees -- Deletion

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

- Change the balance factor of *p* to *equal*
- Shorter remains true   // check for upper nodes since height is changed



No rotations
Height reduced

height 1 reduced

AVL Trees -- Deletion

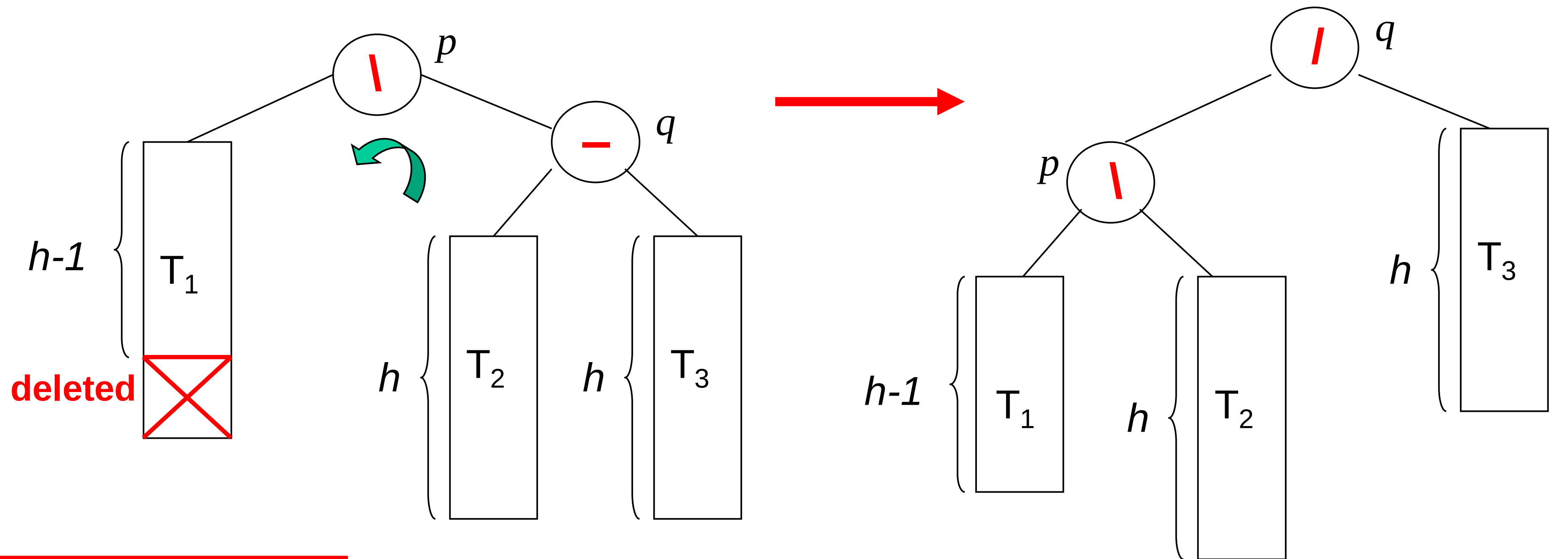
Case 3: ***The balance factor of p is not equal and the shorter subtree is shortened.***

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

AVL Trees -- Deletion

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

- Apply a single rotation
- Change the balance factor of *q* to *left-high* (or *right-high*)
- Shorter becomes false



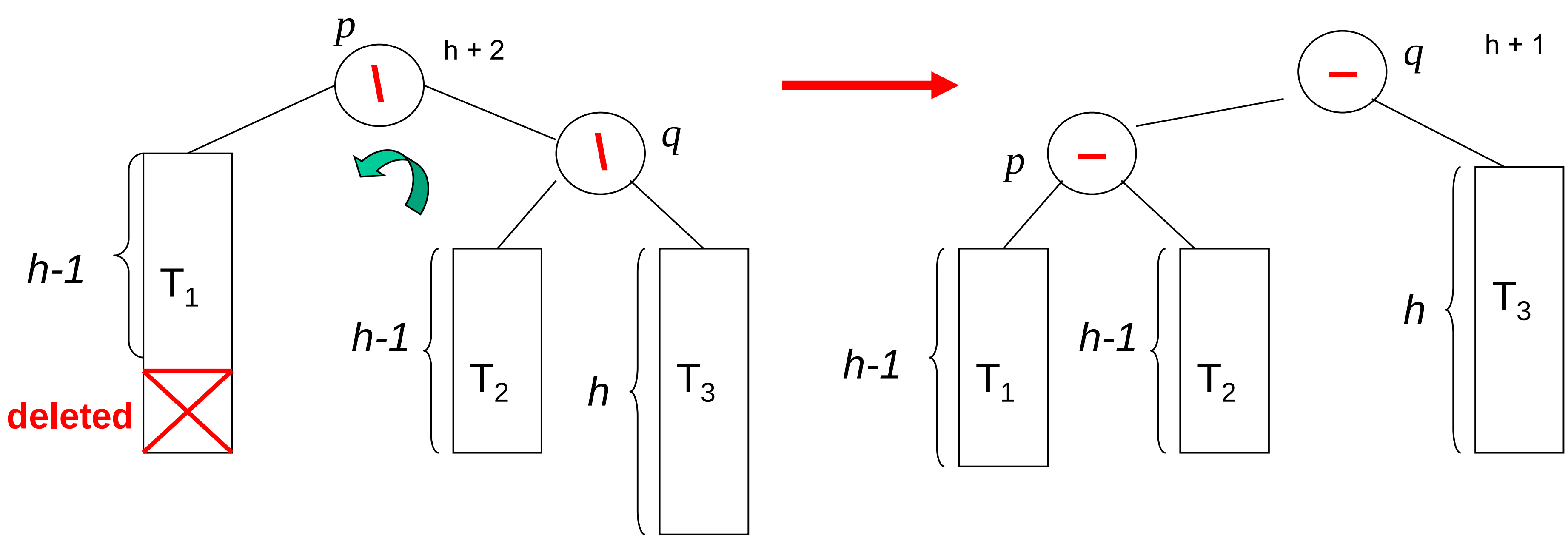
Single rotation
Height unchanged

AVL Trees -- Deletion

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

- Apply a single rotation
- Change the balance factors of *p* and *q* to *equal*
- Shorter remains true

since height is reduced

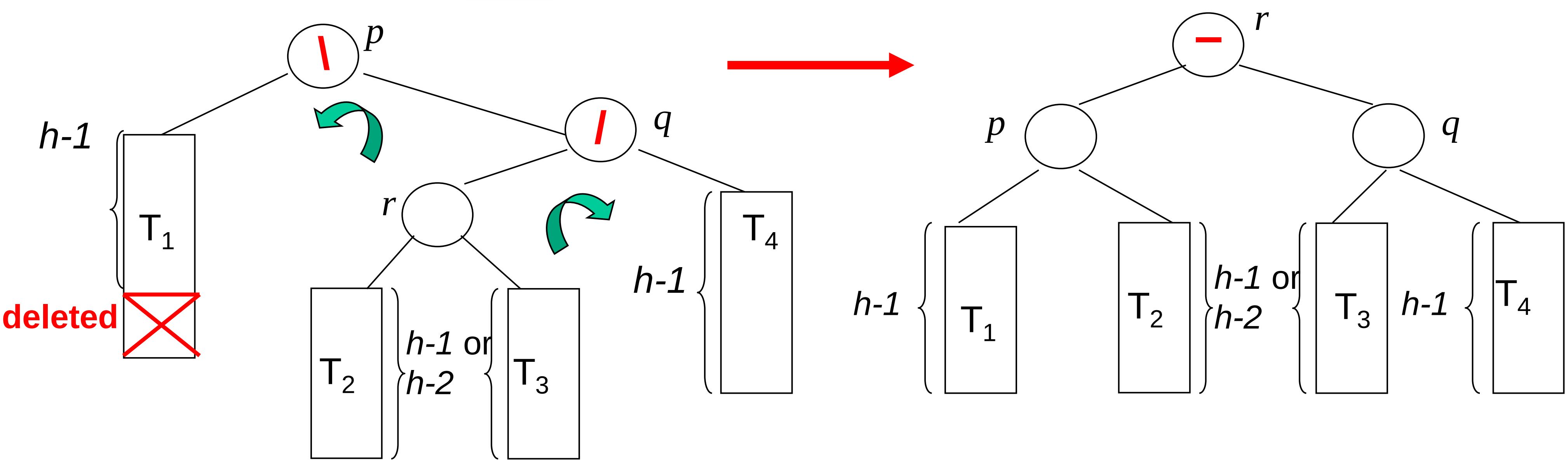


Single rotation
Height reduced

AVL Trees -- Deletion

Case 3c: ***The balance factor of q is the opposite of that of p.***

- Apply a double rotation
- Change the balance factor of the new root to *equal*
- Also change the balance factors of *p* and *q*
- Shorter remains true

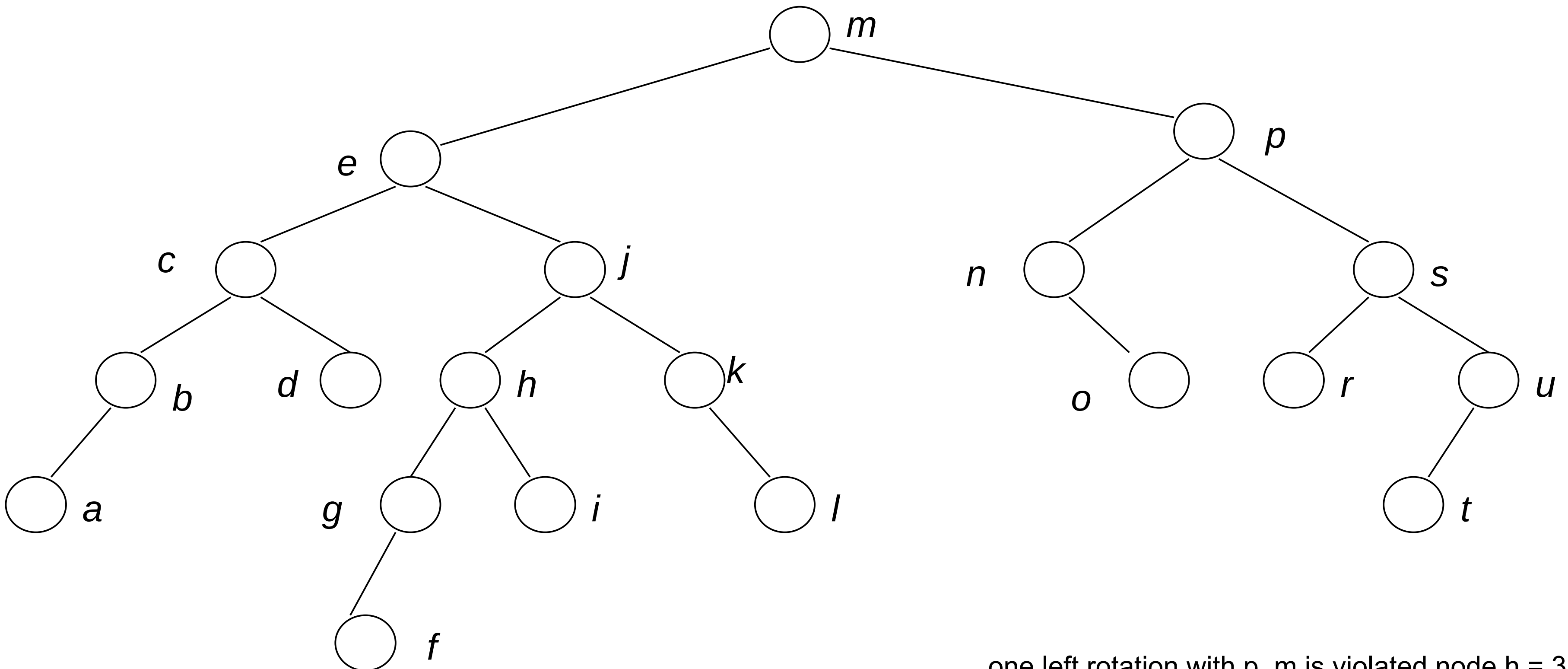


Double rotation

Height reduced

AVL Trees -- Deletion

Exercise**:** Delete *o* from the following AVL tree



one left rotation with p, m is violated node h = 3, h = 5
right rotate

Check the following applet for more exercises.

http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

AVL Trees -- Analysis

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 minimum number of nodes in an AVL tree?

minN(0) = 0

minN(1) = 1

minN(2) = 2

minN(3) = 4

...

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

Max height of an N-node AVL tree is less than 1.44 log N