AVL Trees

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

Balanced trees

- We saw that most of the algorithms in BSTs are $O(h)$
  - But $h = O(n)$ in the worst-case
- So it makes sense to keep trees **"balanced"**
  - Many different ways to define what "balanced" means
  - In all of them: $h = O(\log n)$
- Eg. **complete** are one type of balanced tree (see Heaps)
  - But it's hard to maintain both BST and complete properties together
- **AVL**: a different type of balanced trees

AVL Trees

- An AVL tree is a BST with an extra property:

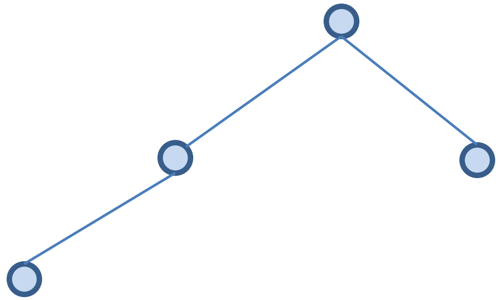
  For **all nodes**: $|\text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})| \leq 1$

- In other words, no subtree can be much shorter/taller than the other

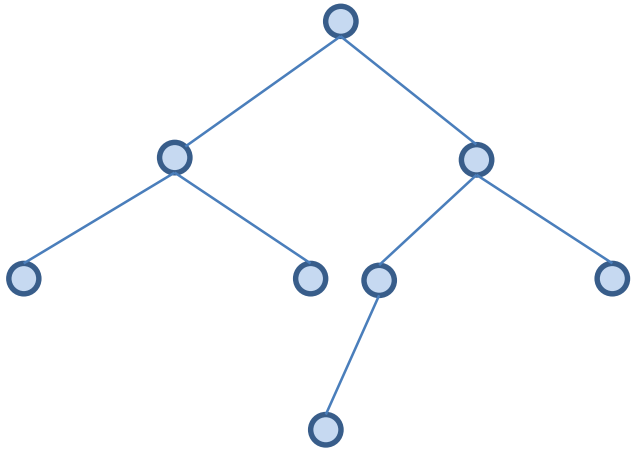
- Recall: **height** is the longest path from the root to some leaf
  - tree with only a root: height 0
  - empty tree: height -1

- Named after Russian mathematicians Adelson-Velskii and Landis

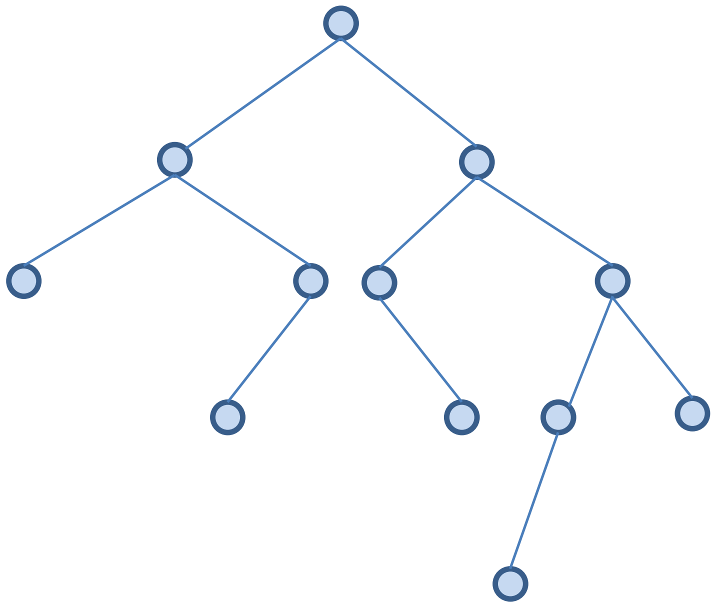
Example – AVL tree



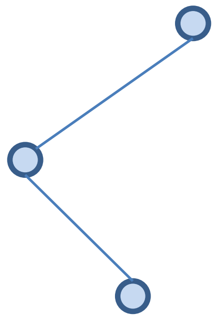
Example – AVL tree



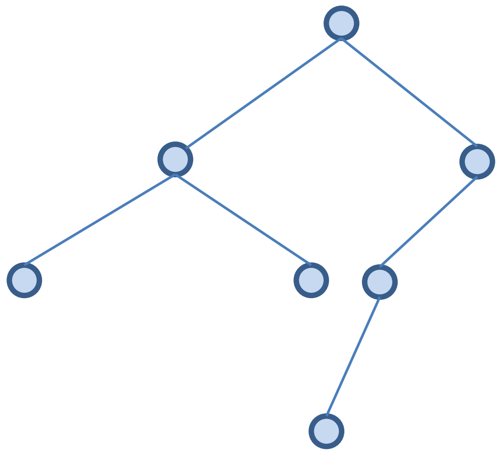
Example – AVL tree



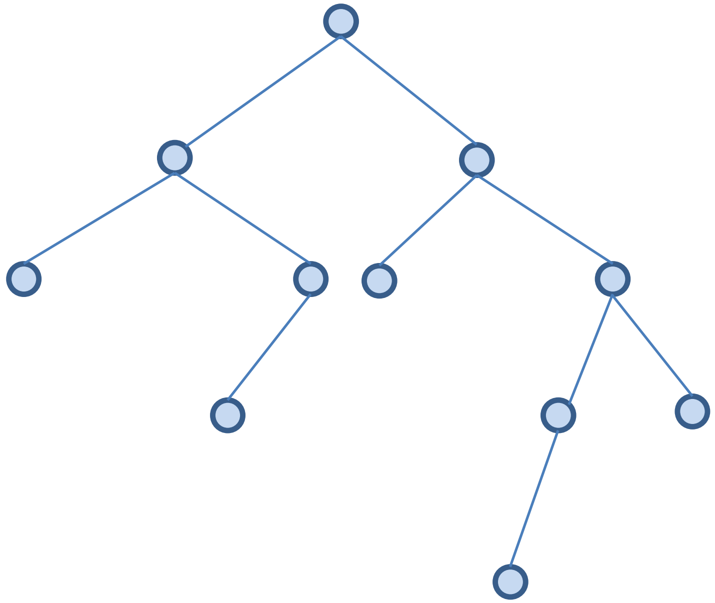
Example – Non-AVL tree



Example – Non-AVL tree



Example – Non AVL tree



The desired property

- In an AVL tree: $h = O(\log n)$
  - Proving this is not hard
- $n(h)$: **minimum number of nodes** of an AVL tree with height h
- We show that $h \leq 2 \log n(h)$
  - by **induction on** h
  - induction works very well on recursive structures!
- The base cases hold trivially (why?)
  - $n(0) = 1$
  - $n(1) = 2$

The desired property

- Inductive step
  - Assume $\frac{h}{2} \leq \log n(h)$ for all $h < k$
  - Show that it holds for an AVL tree of height $h = k$
- **Both subtrees** of the root have height at least $h - 2$
  - because of the AVL property!
  - So $n(k) \geq 2n(k-2) \qquad\qquad (1)$
- Induction hypothesis for $h = k - 2$
  - $\frac{k-2}{2} \leq \log n(k-2)$
- From (1) we take $\log$ on both sides and apply the ind. hypothesis
  - $\log n(k) \geq 1 + \log n(k-2) \geq 1 + \frac{k-2}{2} = \frac{k}{2}$

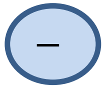
Balance factor

A node can have one of the following "balance factors"

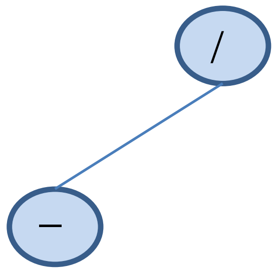
Balance factor	Meaning
`-`	Sub-trees have equal heights
`/`	Left sub-tree is 1 higher
`//`	Left sub-tree is > 1 higher
`\`	Right sub-tree is 1 higher
`\\`	Right sub-tree is > 1 higher

Nodes `-`, `/`, `\` are AVL.
Nodes `//`, `\\` are not AVL.

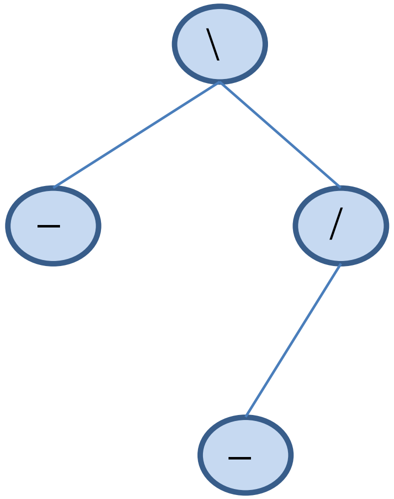
Example AVL Tree



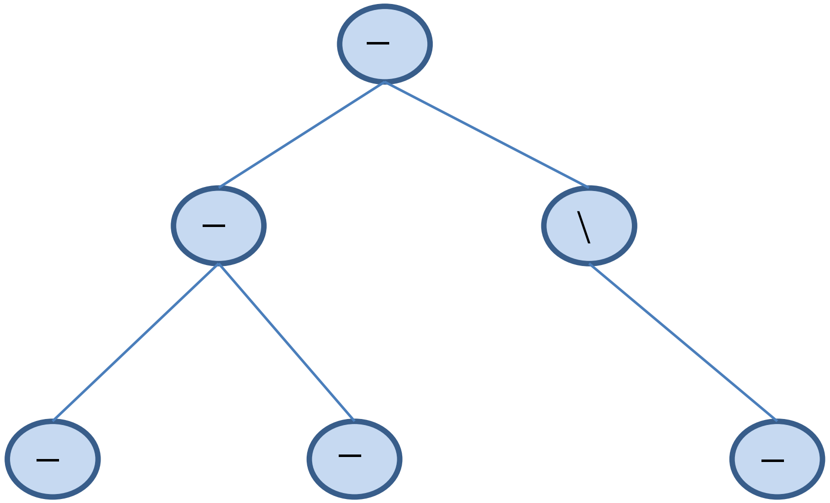
Example AVL Tree



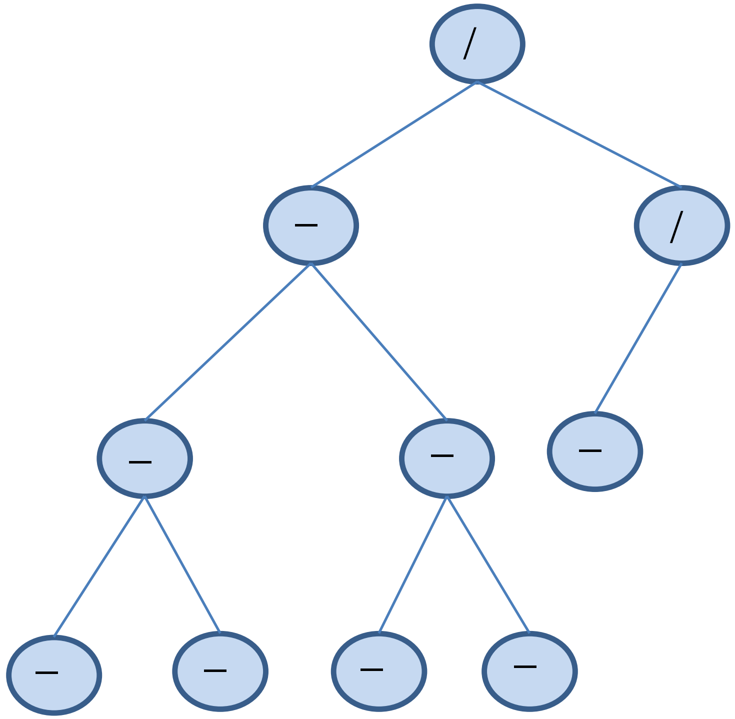
Example AVL Tree



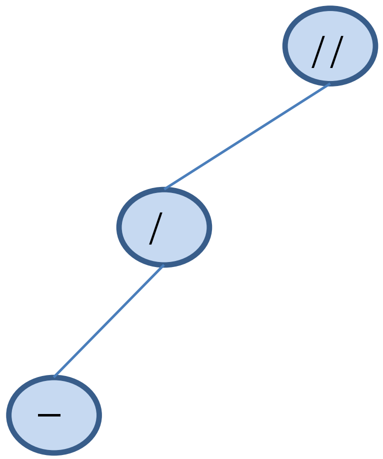
Example AVL Tree



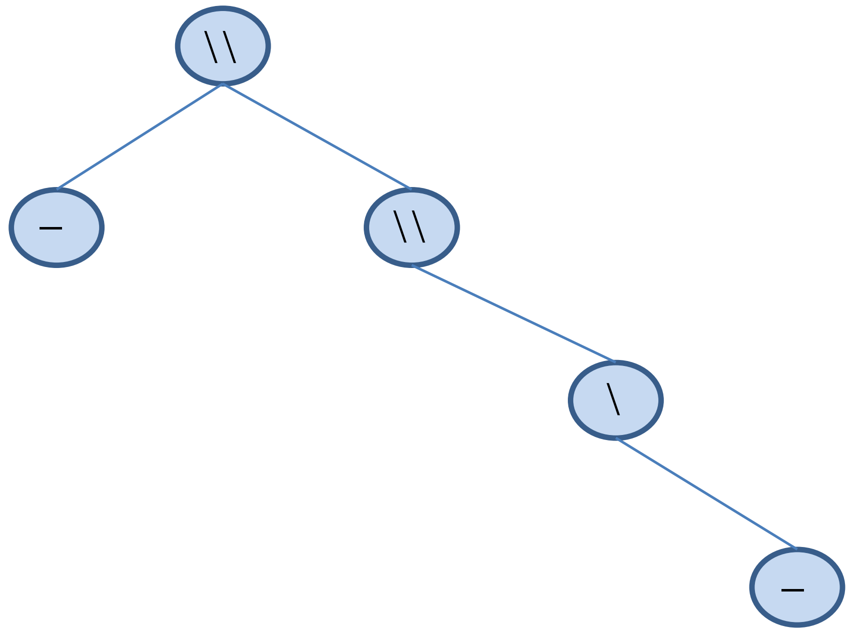
Example AVL Tree



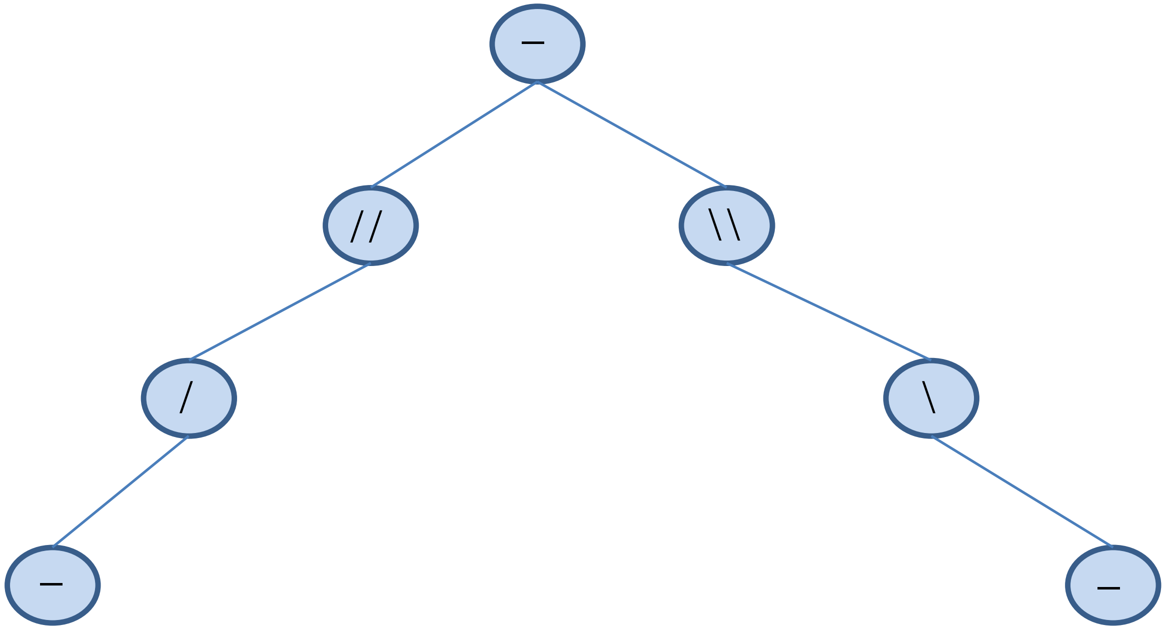
Example non-AVL Tree



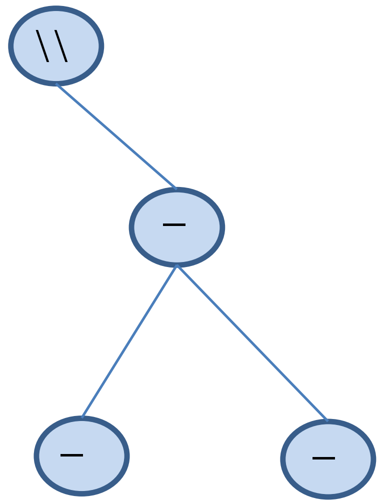
Example non-AVL Tree



Example non-AVL Tree



Example non-AVL Tree



Operations in an AVL Tree

- Same as those of a BST
- Except that we need to **restore** the AVL property
  - after **inserting** a node
  - or **deleting** a node
- We do this using **rotations**

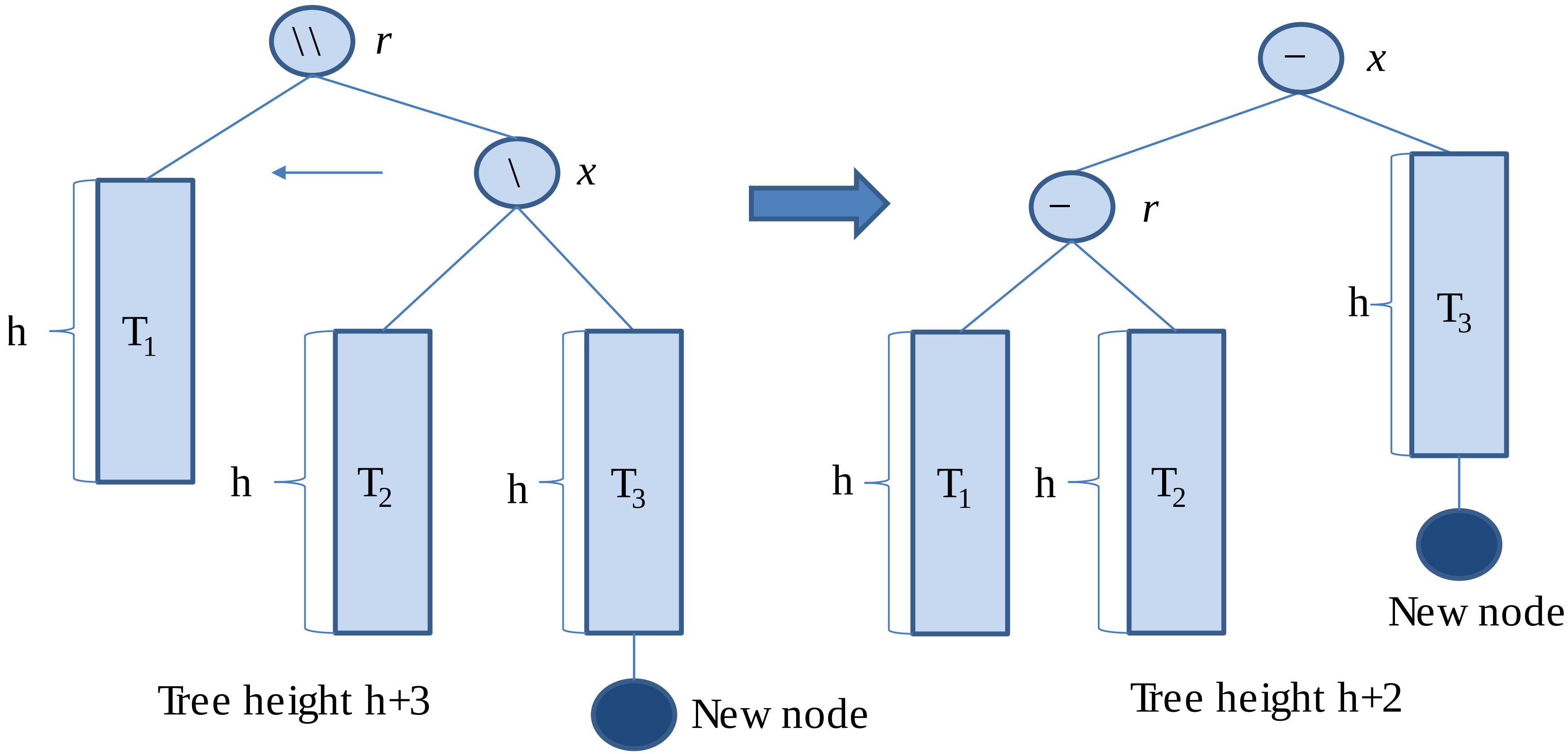
Recursive AVL restore

- Restoring the AVL property is a **recursive** operation
- It happens during an insert or delete
  - Which are both recursive
  - When their recursive calls are **unwinding** towards the root
- So when we restore a node r, its **children** are already restored **AVL trees**

AVL restore after insert

- Assume r became `\\` after an insert (the case `//` is symmetric)
- Let x be the **root** of the **right subtree**
  - The new value was inserted under x (since r is `\\`)
- What can be the **balance factor** of x?
  - `\\` and `//` are not possible since the child x is **already restored**
- Case 1: x is `\`
  - A **left-rotation** on r restores the property!
  - Both r and x become `-` (easily seen in a drawing)

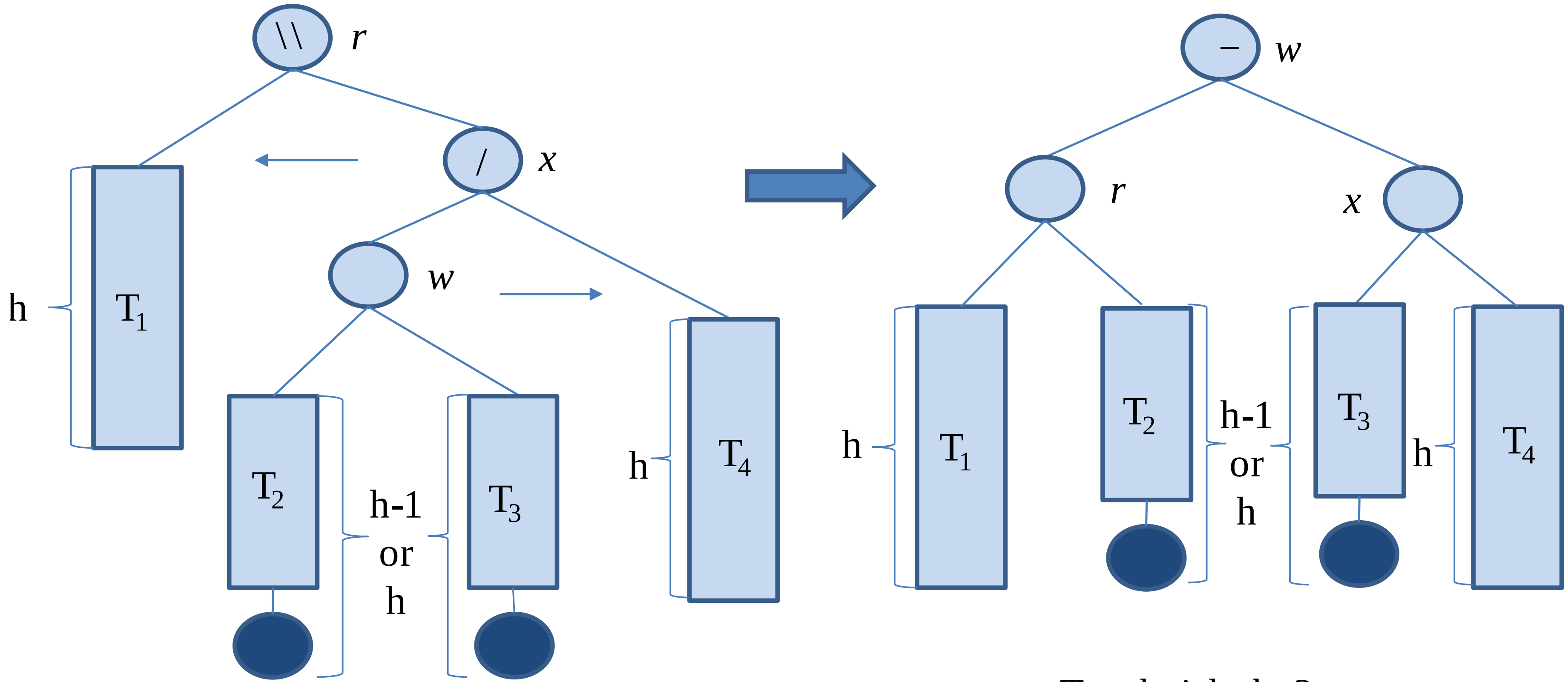
Insert: single left rotation at r



AVL restore after insert

- Case 2: x is `/`
  - This is more tricky
  - A left-rotation on r (as before) might cause x to become `//`
- We need to do a **double** right-left rotation
  - First **right-rotation** on x
  - Then **left-rotation** on r
- The left-child w of x becomes the new root
  - w becomes `-`
  - r becomes `-` or `/`
  - x becomes `-` or `\`

Insert: double right-left rotation at x and r



One of T_2 or T_3 has the new node and height h
Tree height h+3

Tree height h+2

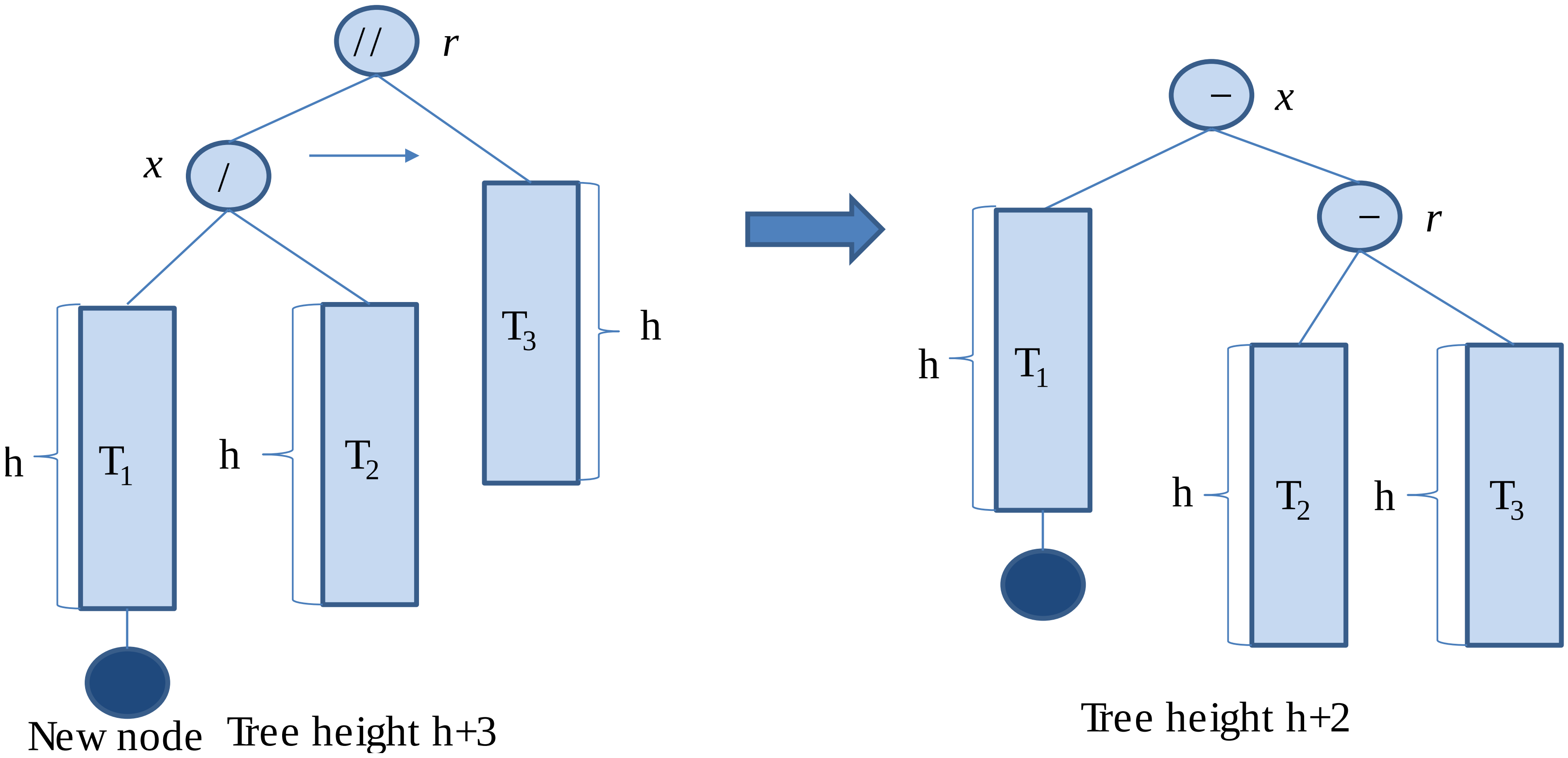
AVL restore after insert

- Case 3: x is `-`
- This in fact **cannot happen**!
    - Assume both subtrees of x have height h
    - Then the left subtree of r also must have height (h)
    - Otherwise AVL would be violated **before** the insert (see the drawings)

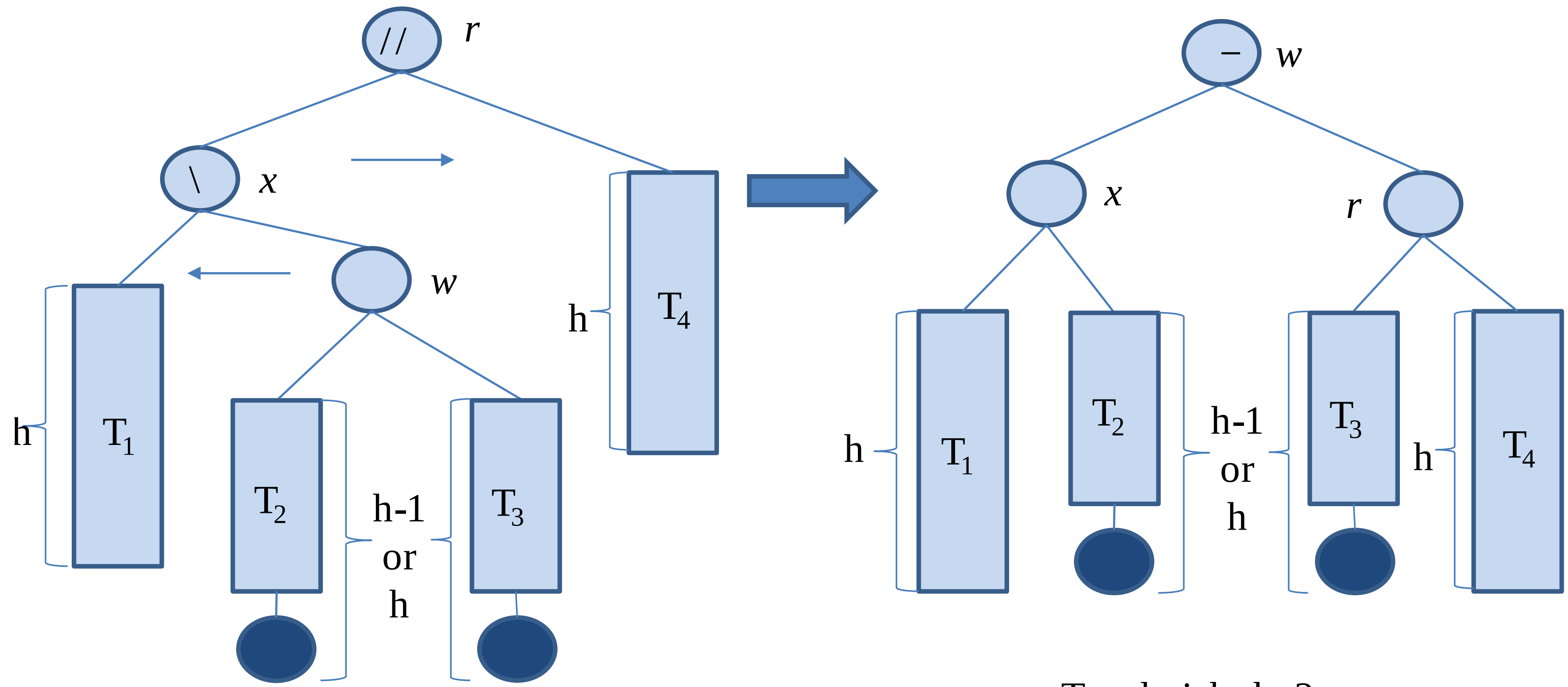
Symmetric case

- The case when x becomes `//` is **symmetric**
- We need to consider the BF of its **left-child** x
  - x is `/` : we do a **single right** rotation at r
  - x is `\` : we do a **double left-right** rotation at x and r
  - x is `-` : **impossible**

Insert: single right rotation at r



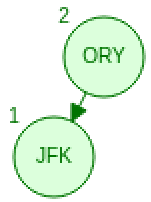
Insert: double left-right rotation at x and r



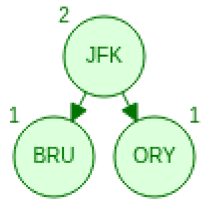
One of T_2 or T_3 has the new node and height h
Tree height h+3

Tree height h+2

Insert example

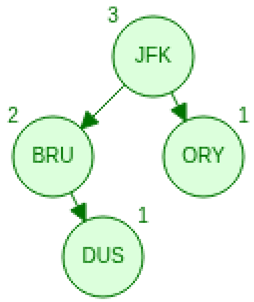


Insert example



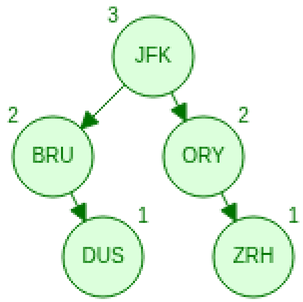
Inserting BRU, causes single right-rotate at ORY

Insert example



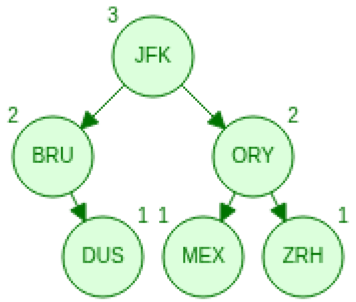
Inserting DUS

Insert example



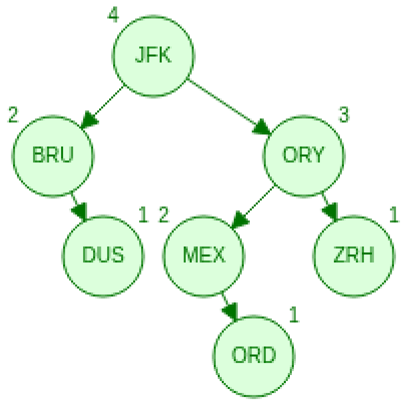
Inserting ZRH

Insert example



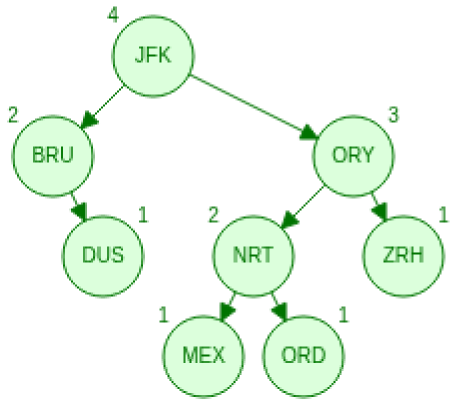
Inserting MEX

Insert example



Inserting ORD

Insert example

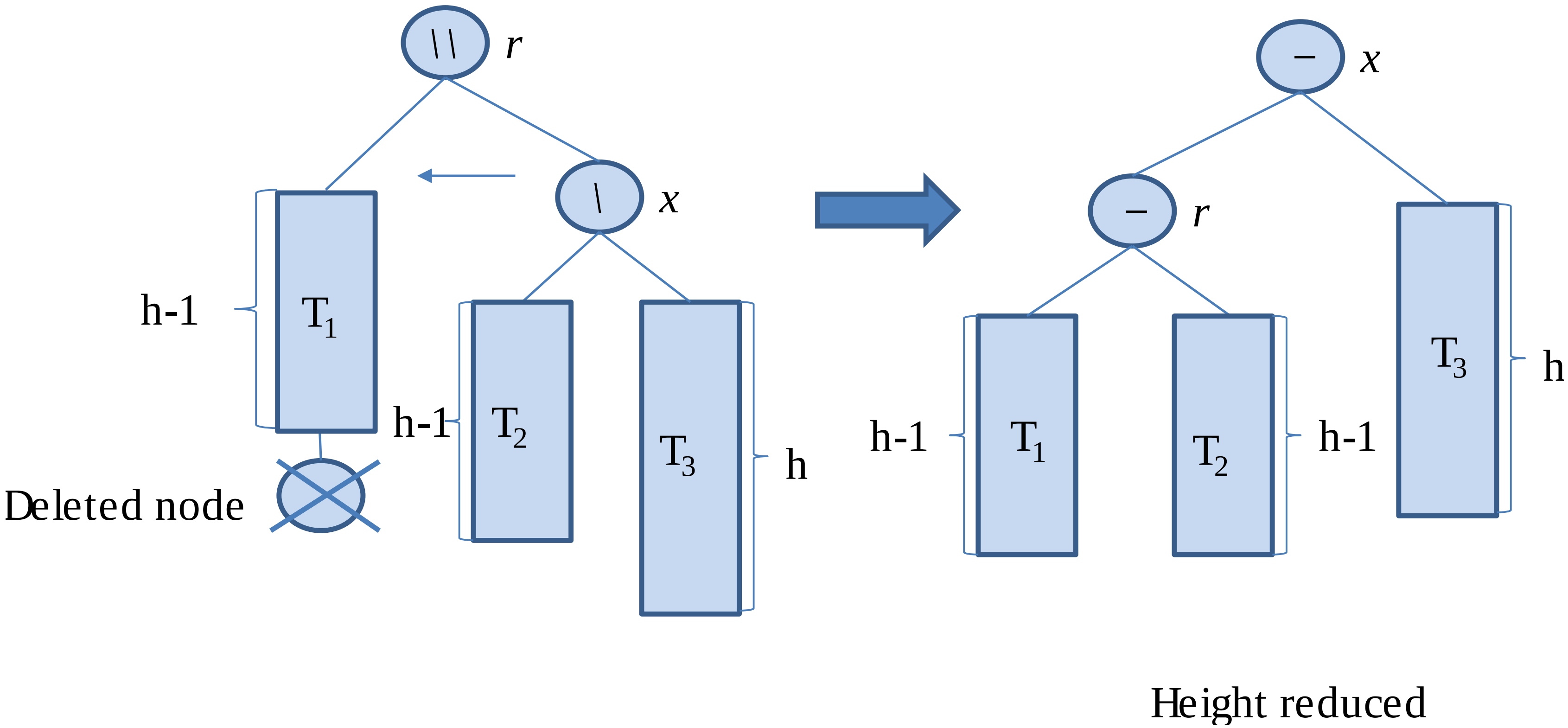


Inserting NRT, causes double right-left rotation at ORD and MEX

AVL restore after delete

- Assume r became `\\` after delete (the case `//` is symmetric)
- Let x be the **root** of the **right-subtree**
  - The value was deleted from the left sub-tree (since r is `\\`)
- What can be the **balance factor** of x?
  - `\\` and `//` are not possible since the child x is **already restored**
- Case 1: x is `\`
  - A **left-rotation** on r restores the property!
  - Both r and x become `-` (easily seen in a drawing)

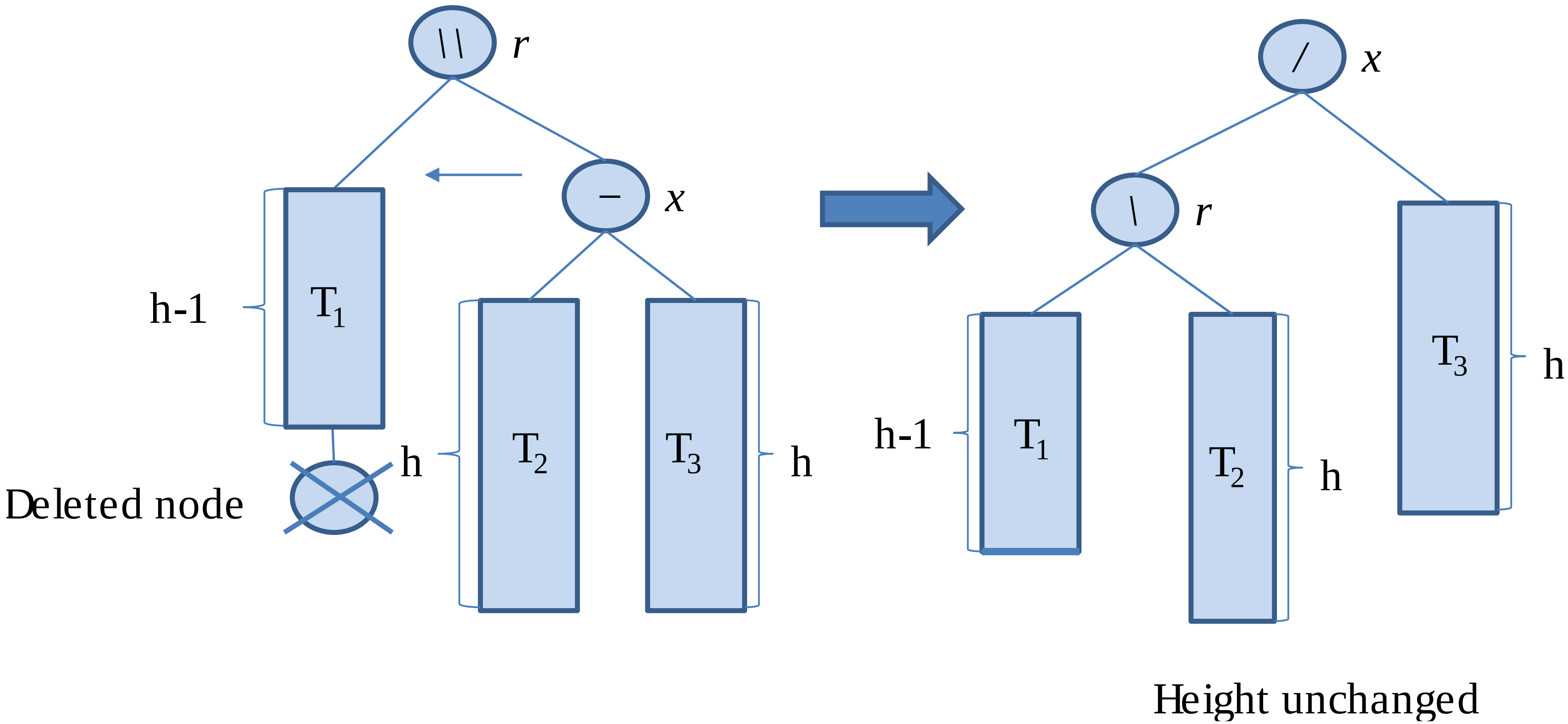
Delete: single left-rotation at r



AVL restore after delete

- Case 2: x is `-`
  - After a **delete** this is possible!
  - A **left-rotation** on r again restores the property
  - r becomes `\`, x becomes `/`

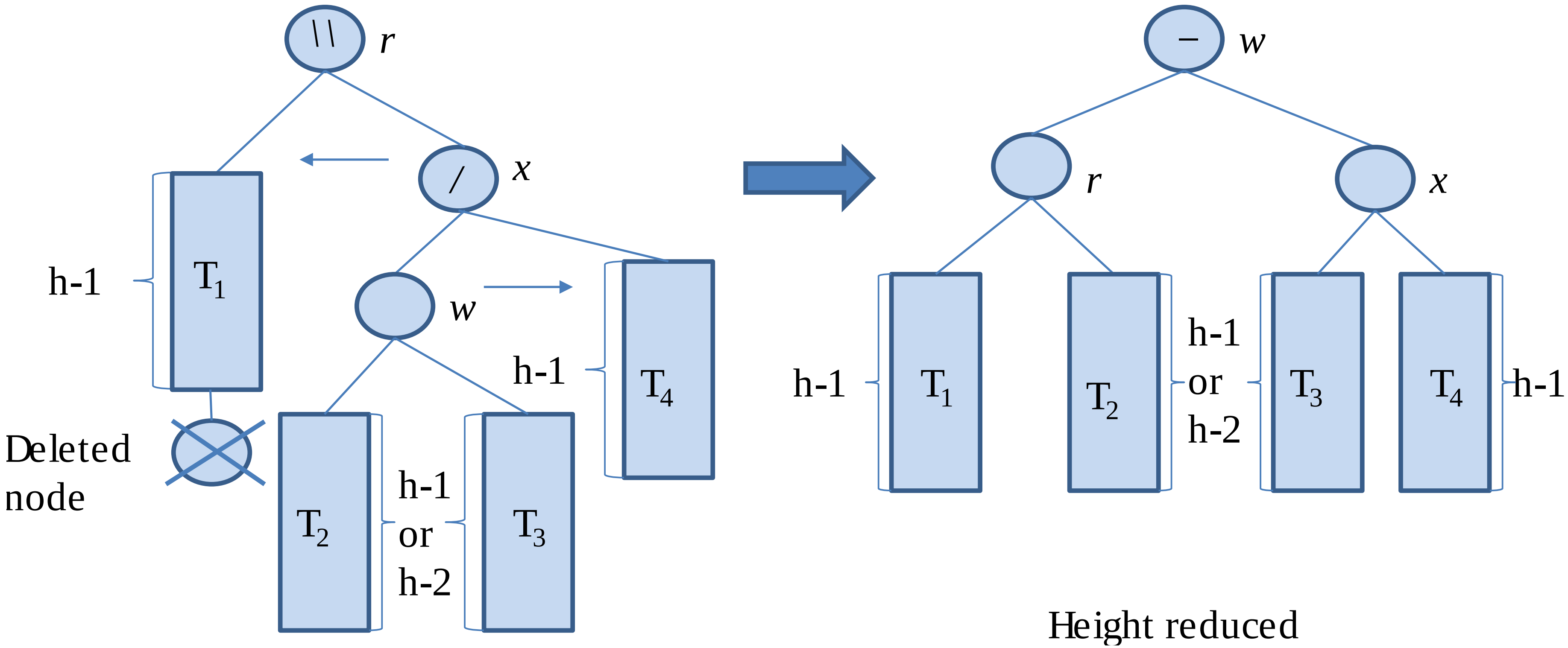
Delete: single left-rotation at r



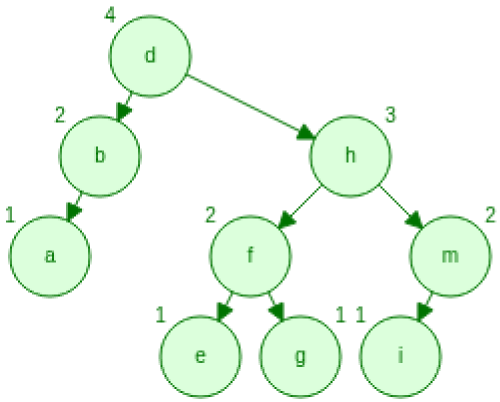
AVL restore after delete

- Case 3: x is `/`
  - This is more tricky
  - A left-rotation on r (as before) might cause x to become `//`
- We need to do a **double** right-left rotation
  - First **right-rotation** on x
  - Then **left-rotation** on r
- The left-child w of x becomes the new root
  - w becomes `-`
  - r becomes `-` or `/`
  - x becomes `-` or `\`

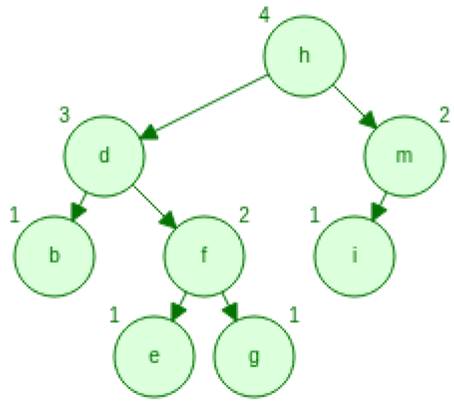
Delete: double right-left rotation at r



Delete example

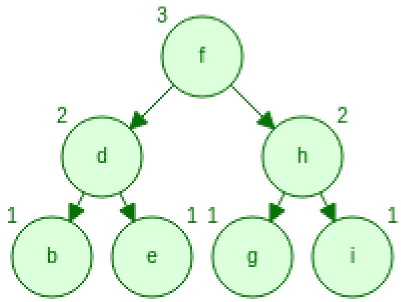


Delete example



Deleting a, causes single left-rotate at d

Delete example



Deleting m, causes double left-right rotation at d and h

Complexity of operations on AVL trees

- Search on BST is $O(h)$
  - So $O(\log n)$ for AVL, since $h \leq 2\log n$
- Insert/delete on BST is $O(h)$
  - We add at most on rotation at each step, each rotation is $O(1)$
  - So also $O(\log n)$
- Interesting fact
  - During insert **at most one rotation** will be performed!
  - Because both rotations we saw **decrease** the height of the sub-tree

Implementation details

- We need to keep the **height** of each subtree
  - to compute the balance factors
  - If we need to save memory we can store **only** the balance factors
- Restoring after both insert and delete are similar
  - We can treat them together

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.* Chapter 9. Section 9.8.
- R. Kruse, C.L. Tondo and B.Leung. *Data Structures and Program Design in C.* Chapter 9. Section 9.4.
- M.T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition. Section 10.2