# Data Structures

## Lecture 4
## AVL and WAVL Trees

Haim Kaplan and Uri Zwick

November 2014

Last updated: April 16, 2018

# Balanced search trees

O(log $n$) worst-case time for all operations

AVL trees (1962)

Red-Black trees (1972)

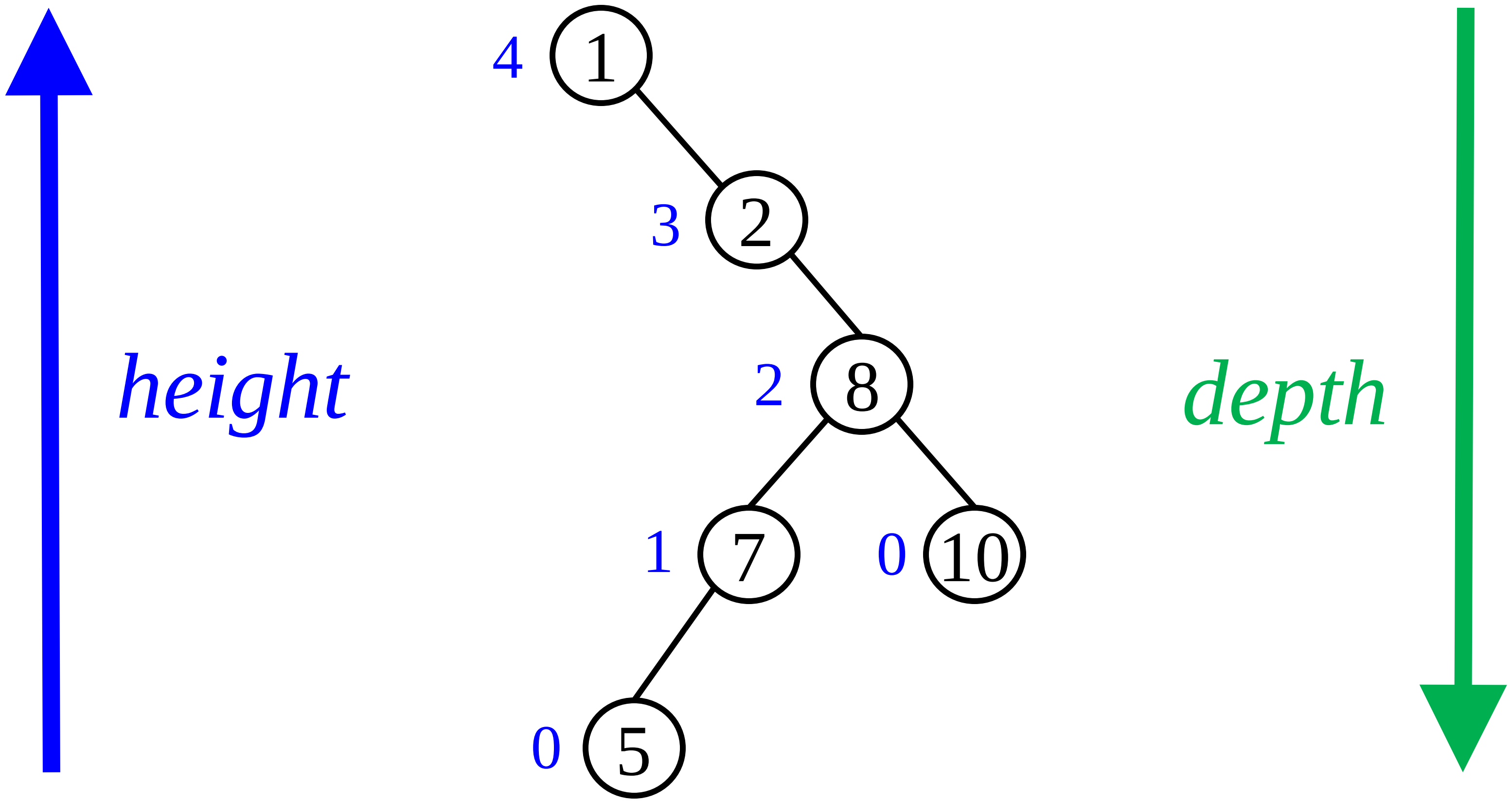
⋮

WAVL trees (2009)

Splay trees (amortized bounds)

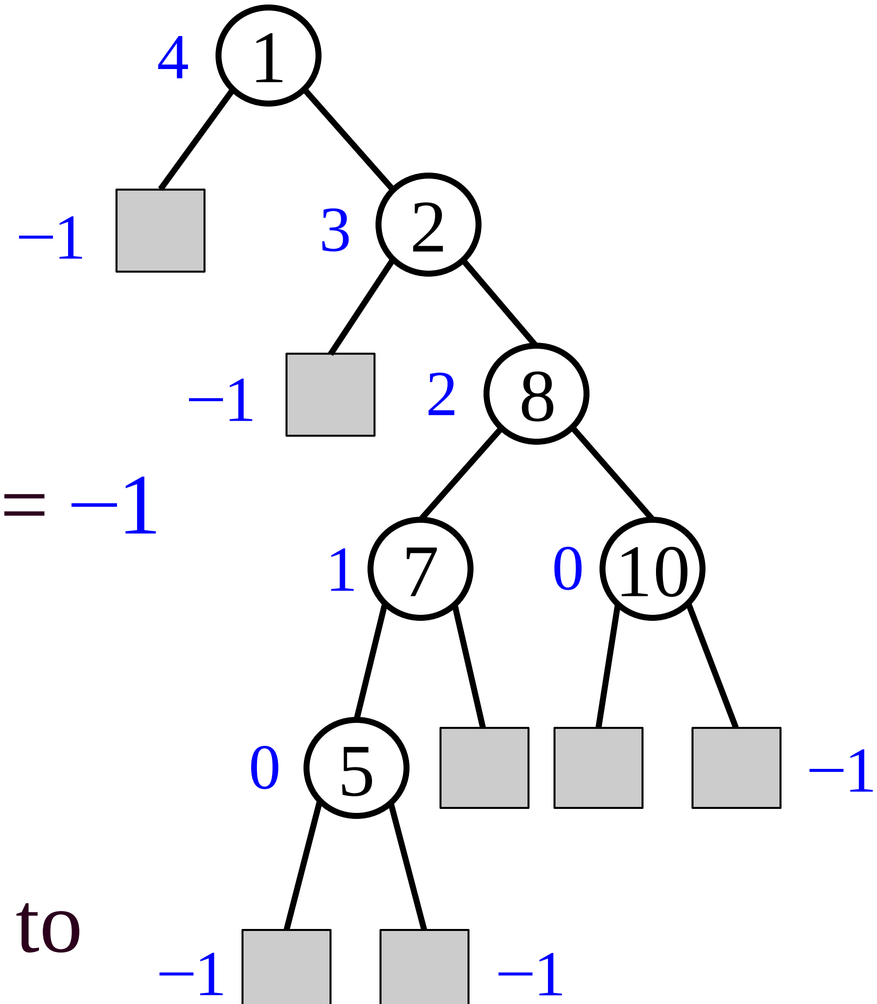
B-trees (non-binary)

# *Height* – Length of longest path to leaf




$$height(x) = 1 + \max\{height(x.left), height(x.right)\}$$

# External leaves




Height of a leaf = 0

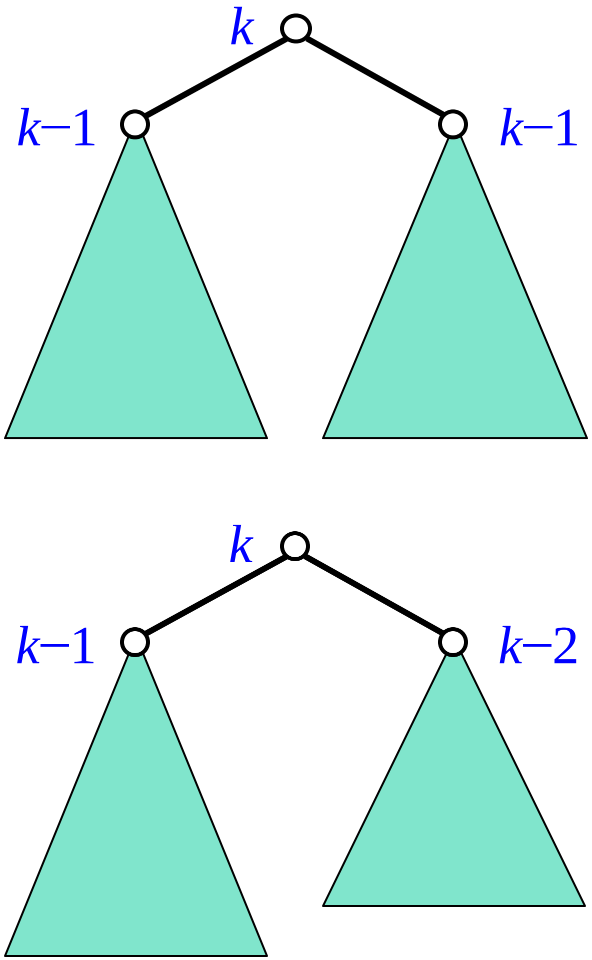
Height of a external leaf = −1

Unifies the treatment of base cases

A single object EXT used to represent all external leaves

# AVL trees

## [Adel'son-Vel'skii, Landis (1962)]

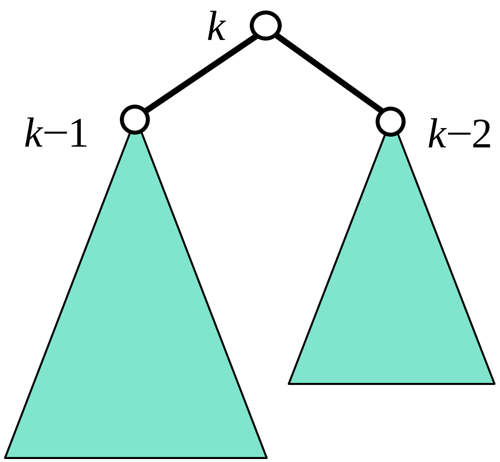



AVL trees are search trees

The *height* of two siblings differs by at most 1

Need only one extra bit per node

# AVL trees
## [Adel'son-Vel'skii, Landis (1962)]




$S_k$ – minimal number of nodes in an AVL tree of height $k$

$$S_k = S_{k-1} + S_{k-2} + 1$$

$$S_{-1} = 0\ ,\ S_0 = 1$$

By induction: $S_k = F_{k+3} - 1 \geq F_{k+2} \geq \phi^k$

$$height \leq \log_\phi n \leq 1.4404 \log_2 n$$

# Fibonacci numbers

$$F_k = F_{k-1} + F_{k-2}\ ,\ \ k \geq 2$$

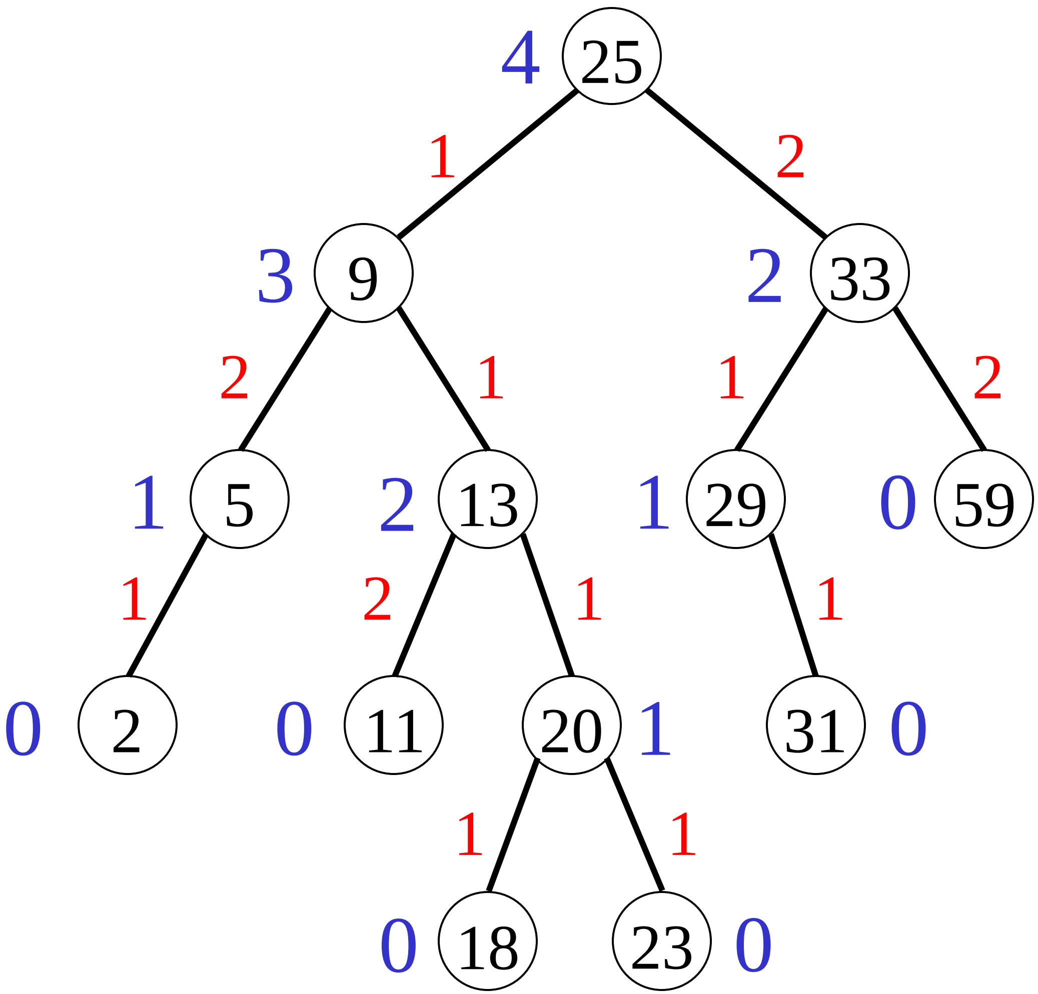
$$F_0 = 0\ ,\ \ F_1 = 1$$

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $F_n$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |

$$F_k \ \geq \ \phi^{k-2}$$

$$\phi = \tfrac{1+\sqrt{5}}{2} \simeq 1.618$$

# An AVL tree



# The challenge

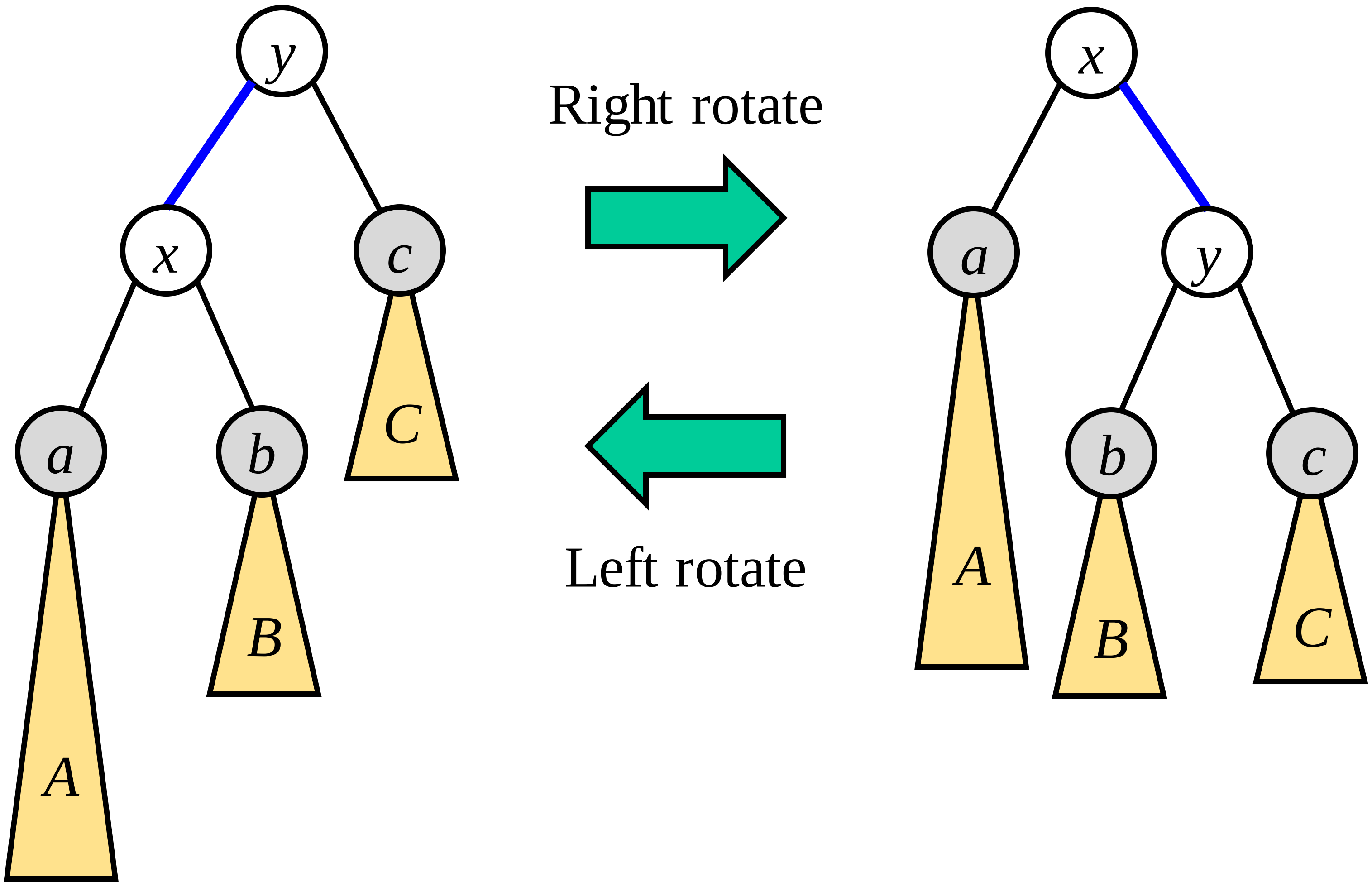
If we insert or delete a node from an AVL tree, the resulting tree is not necessarily an AVL tree

After insertions and deletions we may need to restructure the tree

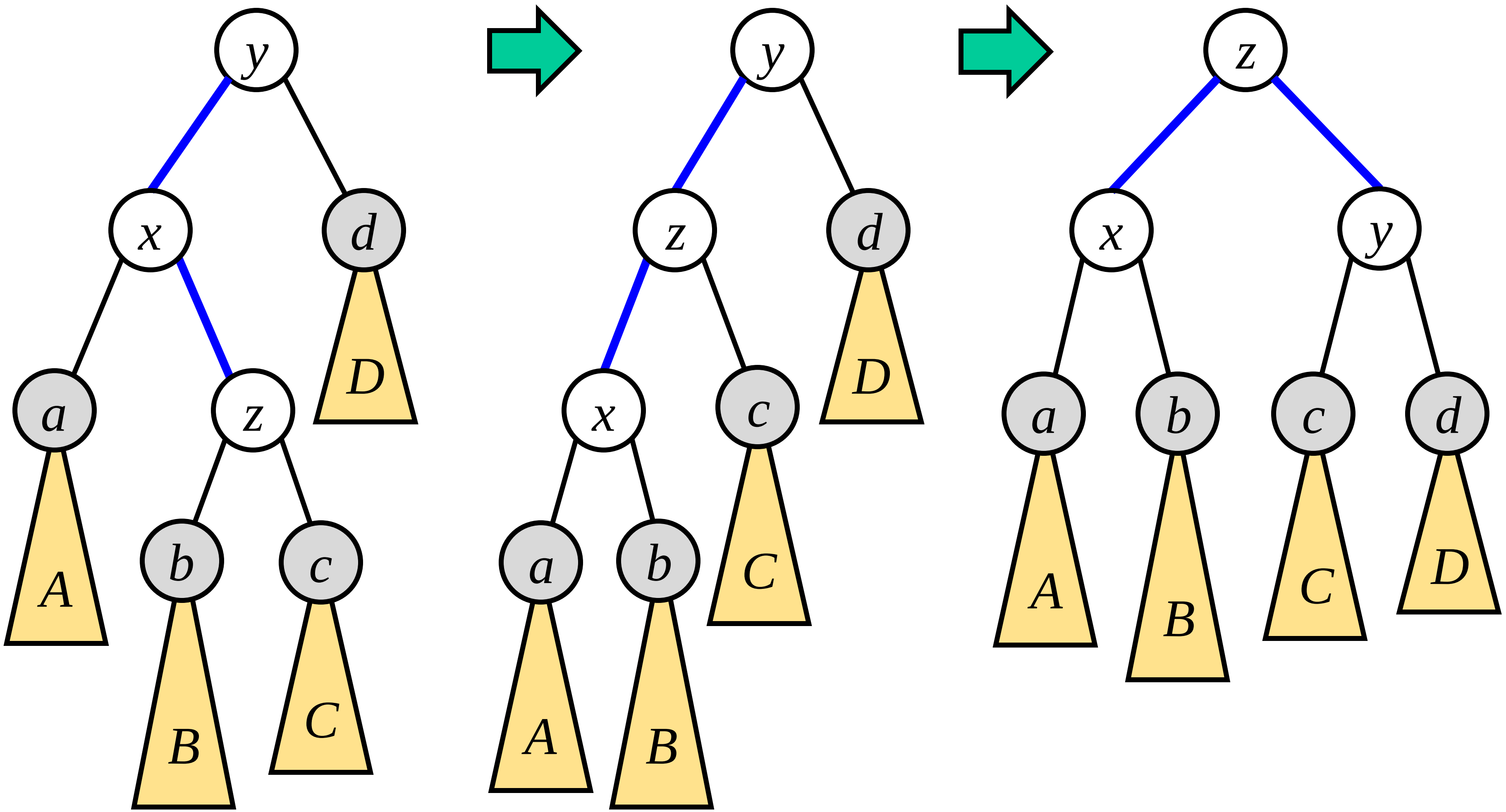
To restructure the tree we use rotations

We want to update the tree in O(log $n$), or less

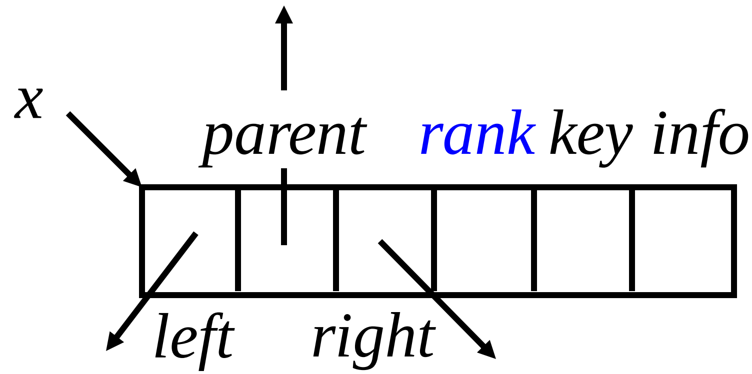
# Rotations



# Double Rotation



# Nodes in AVL trees



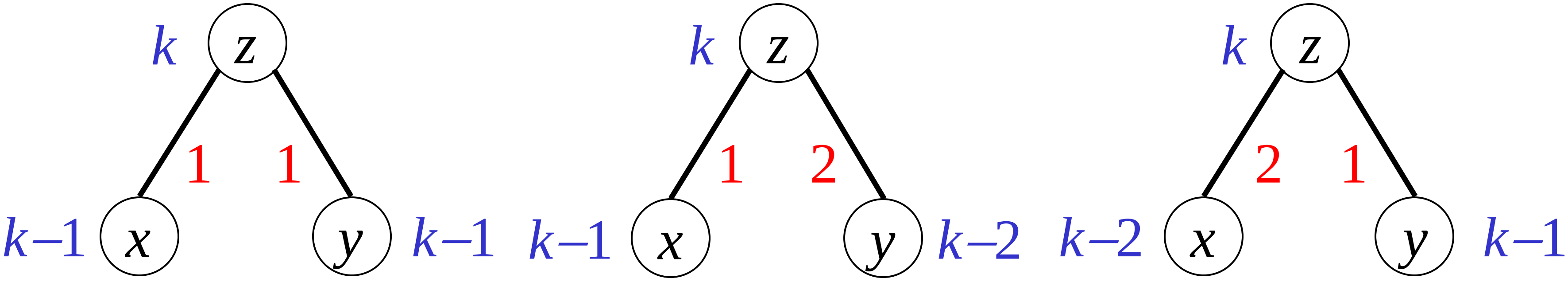

Each node has a *rank*

Each edge has a *rank difference:*
*x.parent.rank − x.rank*

Before/after each insert/delete *rank = height*

(Enough to keep *rank parity*)

# Nodes in AVL trees




Each node has a *rank*

Each edge has a *rank difference*

Before/after each insert/delete *rank = height*

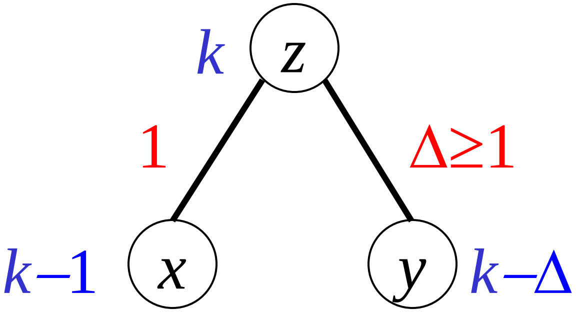
Every node is a 1,1-, 1,2- or 2,1-node

# *rank* = *height*

**Lemma:** If all leaves have *rank* 0, all *rank differences* are positive, and each parent has a child or *rank difference* 1, then the *rank* of each node is equal to its *height*

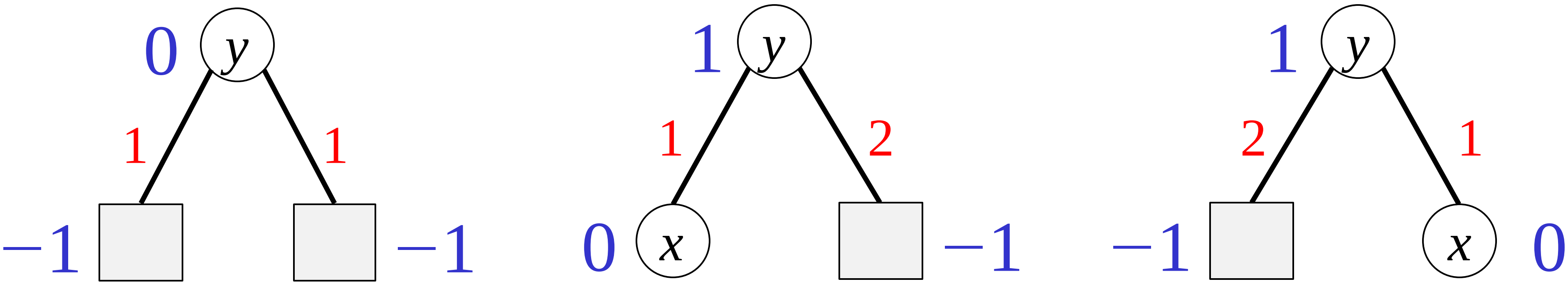
Easy proof by induction:

$k = 1 + \max\{\, k{-}1 \,,\, k{-}\Delta \,\}$



# Nodes in AVL trees

## Internal and external leaves




*rank* of a leaf $= 0$

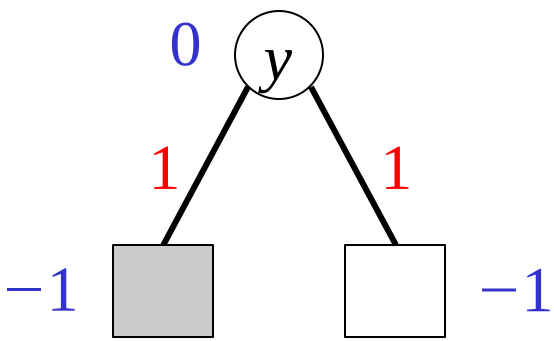
*rank* of a external leaf $= -1$

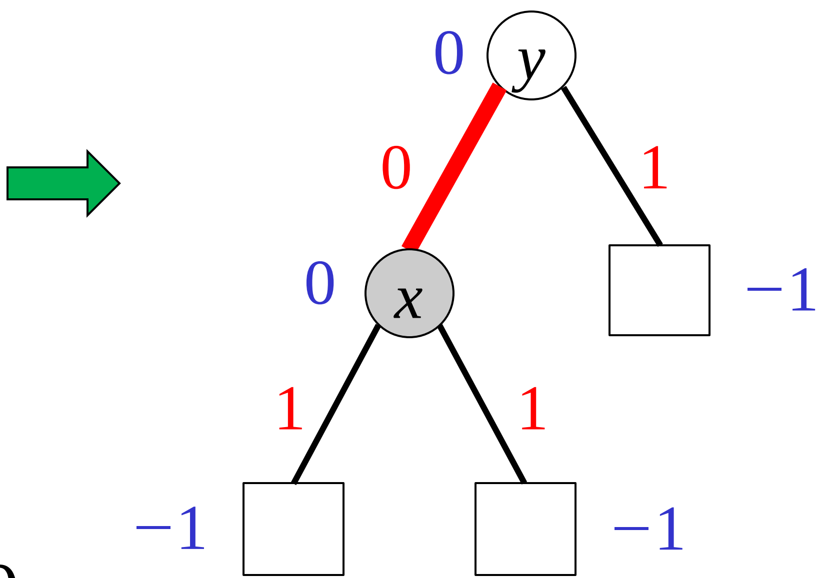
# AVL insertion

# AVL Insertion

Replace an external leaf by a new node $x$

**Case A:** The parent $y$ is a leaf



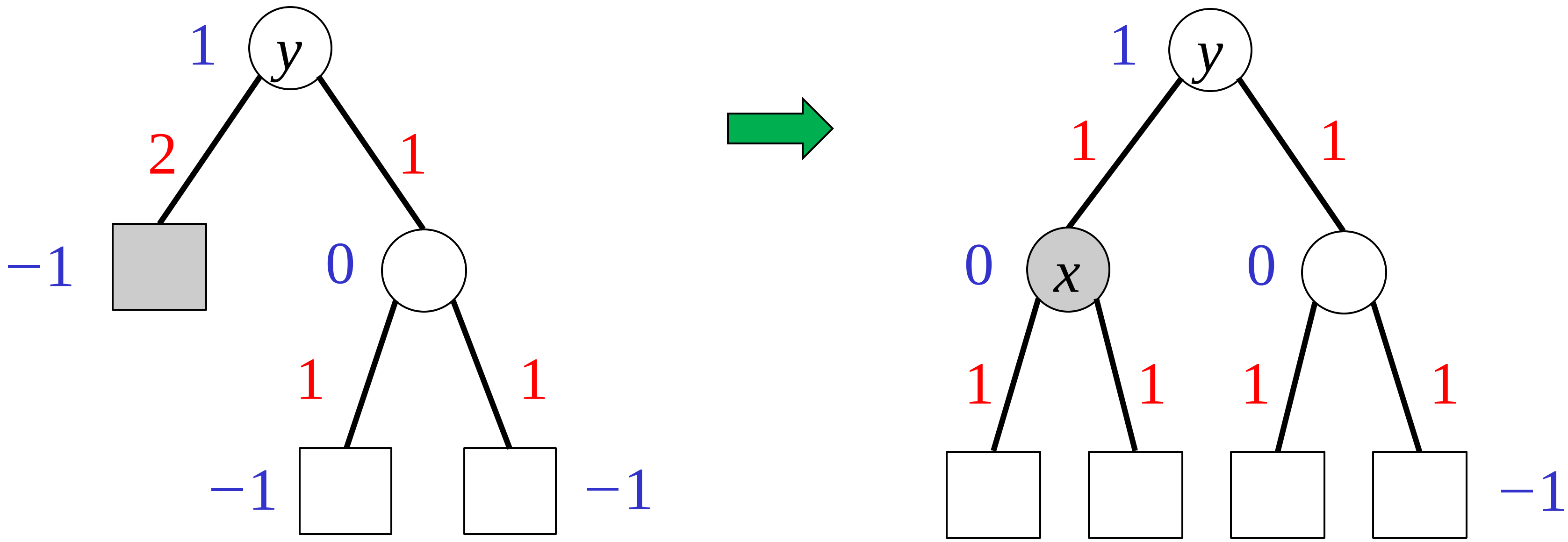

Give $x$ a *rank* of 0

Problem: *rank difference* 0

# AVL Insertion

Replace an external leaf by a new node *x*

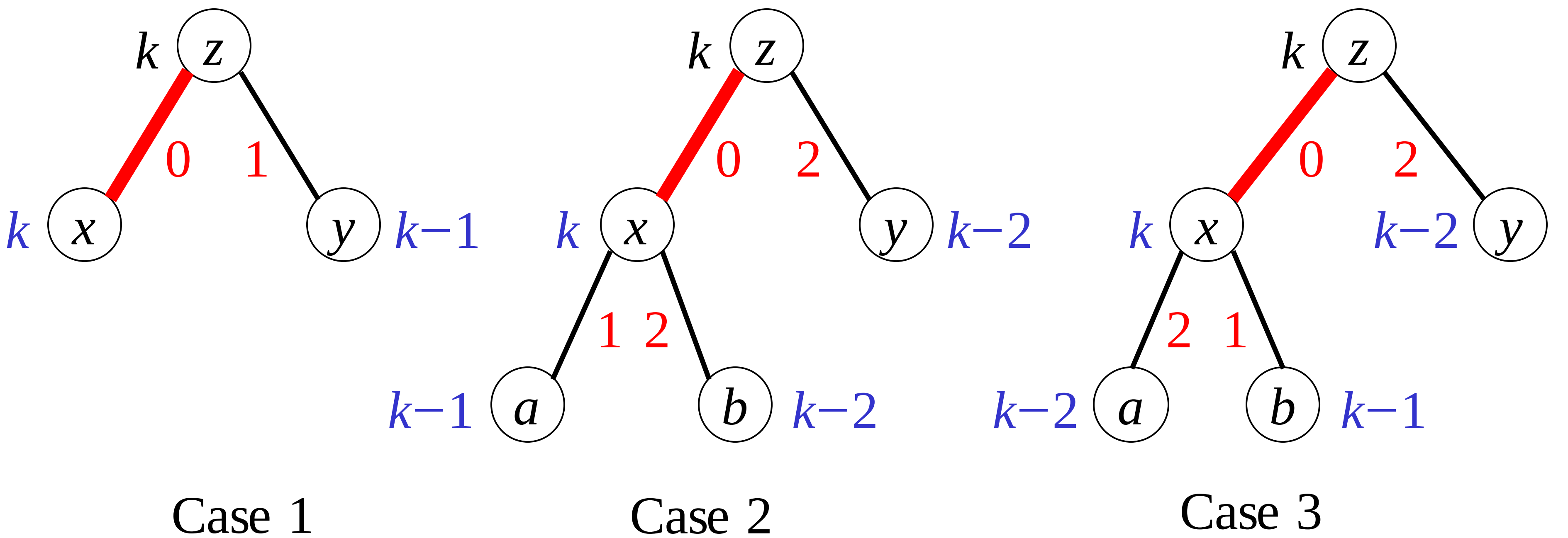
**Case B:** The parent *y* is not a leaf




Resulting tree is a valid AVL tree

# AVL: Insertion rebalancing

## 3 cases (up to symmetry)

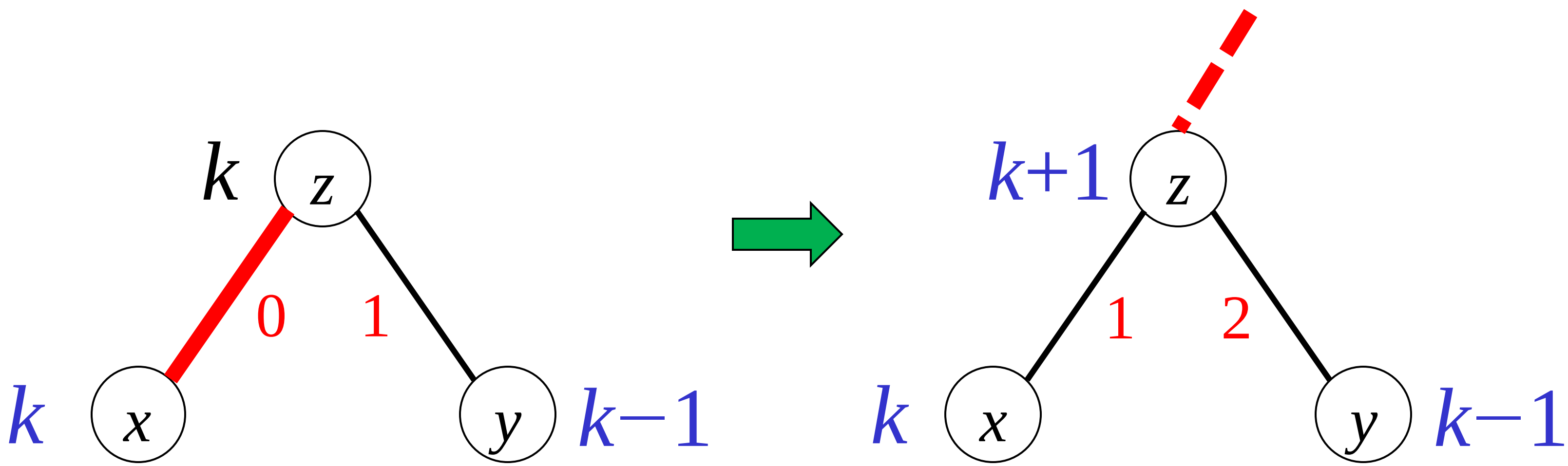



*x* is the *only* node with *rank difference* 0

In cases 2 and 3, *x* is a {1,2}-node

# Rebalancing after insertion

## **Case 1:** Promote

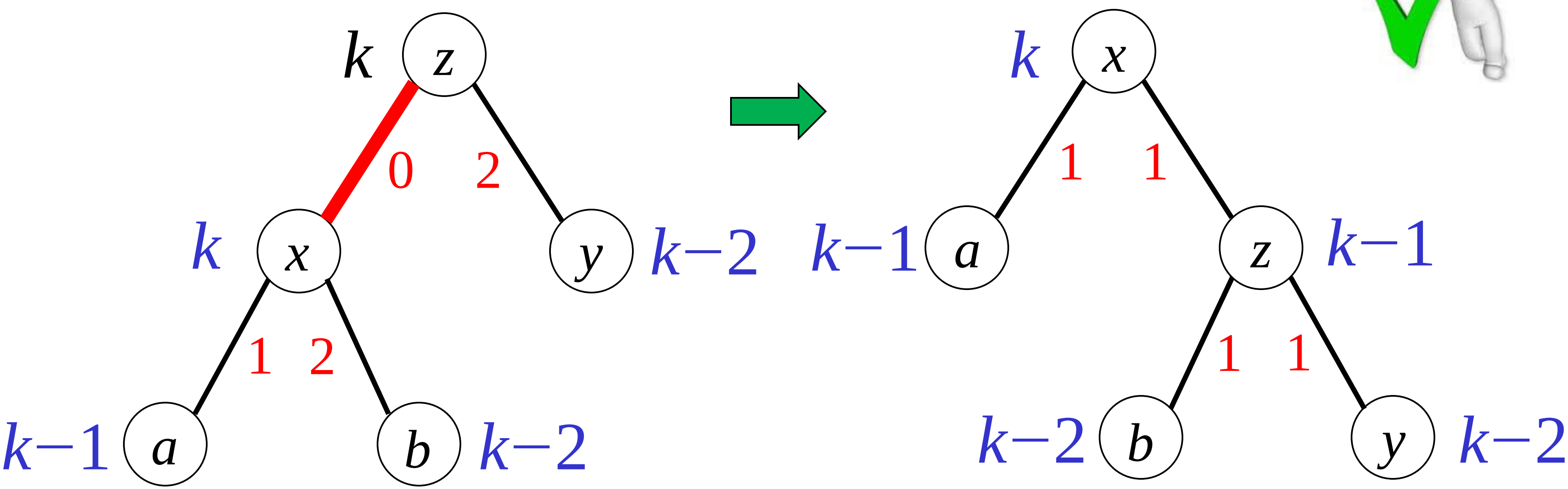



Promote $z$, i.e., increase its *rank* by 1

Problem is either fixed or moved up

# Rebalancing after insertion

## **Case 2:** Single rotation

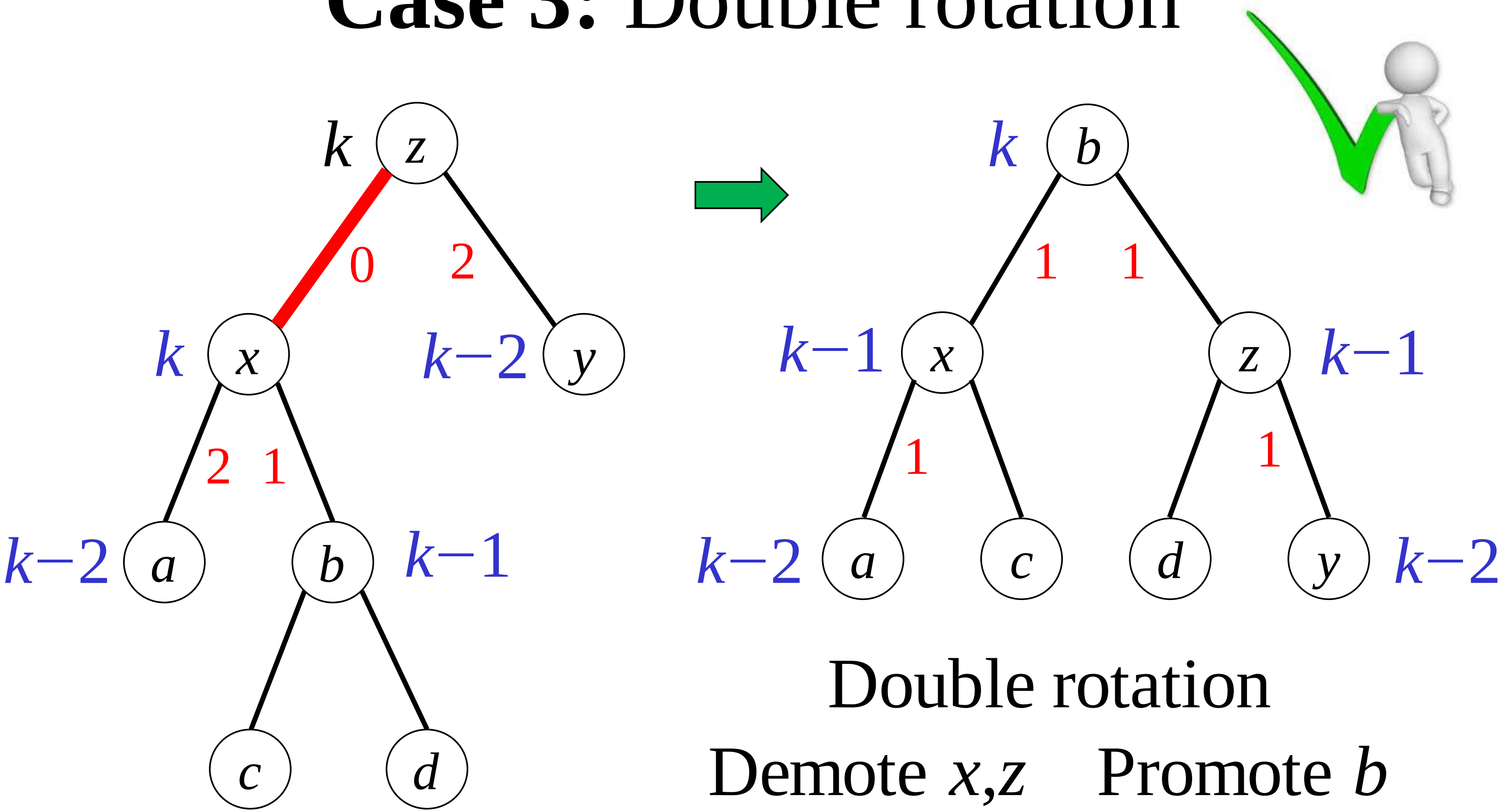



Rotate right Demote *z*

Rebalancing complete!

# Rebalancing after insertion

## **Case 3:** Double rotation




Double rotation

Demote *x*,*z*    Promote *b*

Rebalancing complete!

# AVL Insertion - Summary

*Find* insertion point

*Insert* new node

*Rebalance*

Number of *promotions* $\leq$ *height* $=$ $O(\log n)$

Number of *rotations* $\leq$ 2

Worst-case time $=$ $O(\textit{height})$ $=$ $O(\log n)$

What is the *amortized* number of *rebalancing steps*?

# AVL Insertion

## *Amortized* number of *rebalancing steps*

*Potential* = $\Phi$ =
number of 0,1- 1,1-nodes

The insertion itself, and each rebalancing step change the potential by at most a constant

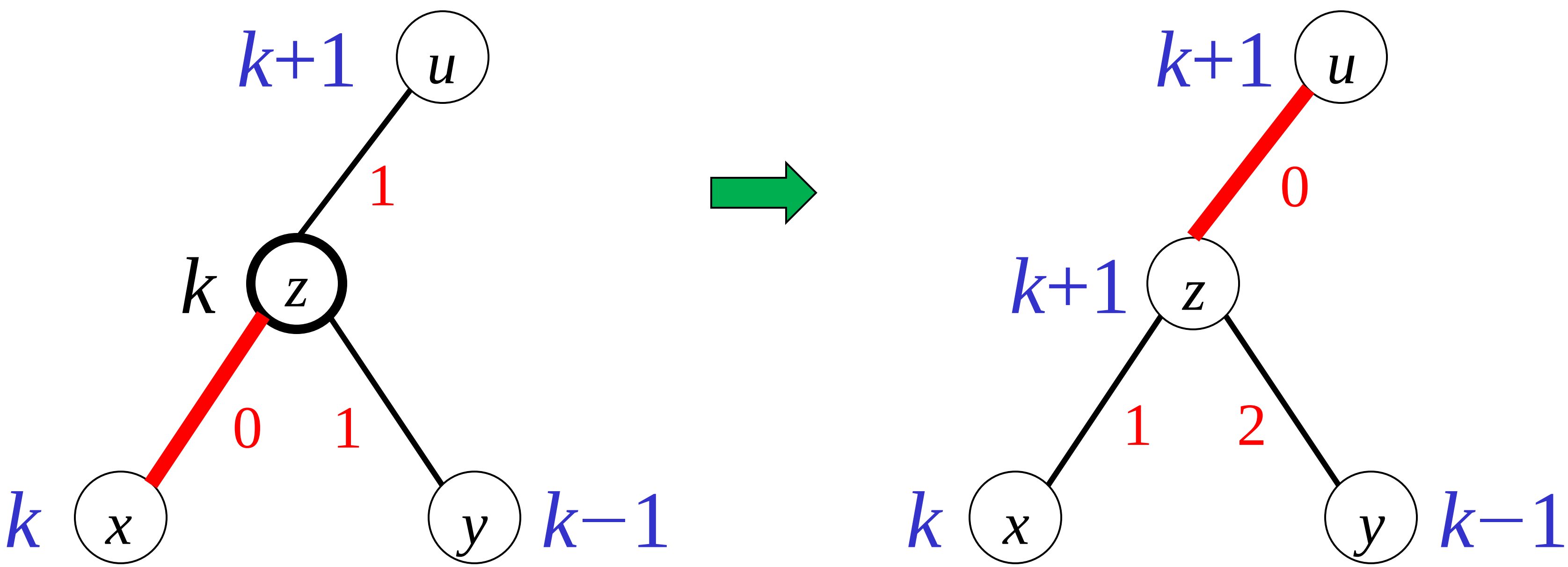
Promotions are the only *non-terminal* cases

*Non-terminal* promotions ***decrease*** the potential

*amort*( #*steps* ) = O(1)

# Rebalancing after insertion

## *Non-terminal* promote




Potential of $u$ (and of $x$,$y$) does not change

$z$ looses its potential: $\Delta\Phi = -1$

*Decrease* in potential pays for this step!

# AVL
# deletion

# AVL Deletion

Replace item to be deleted with its successor or predecessor, if needed

Delete the appropriate node

Perform a sequence of *demotions*, *rotations* and *double rotations* to restore balance

Somewhat more complicated then insertion

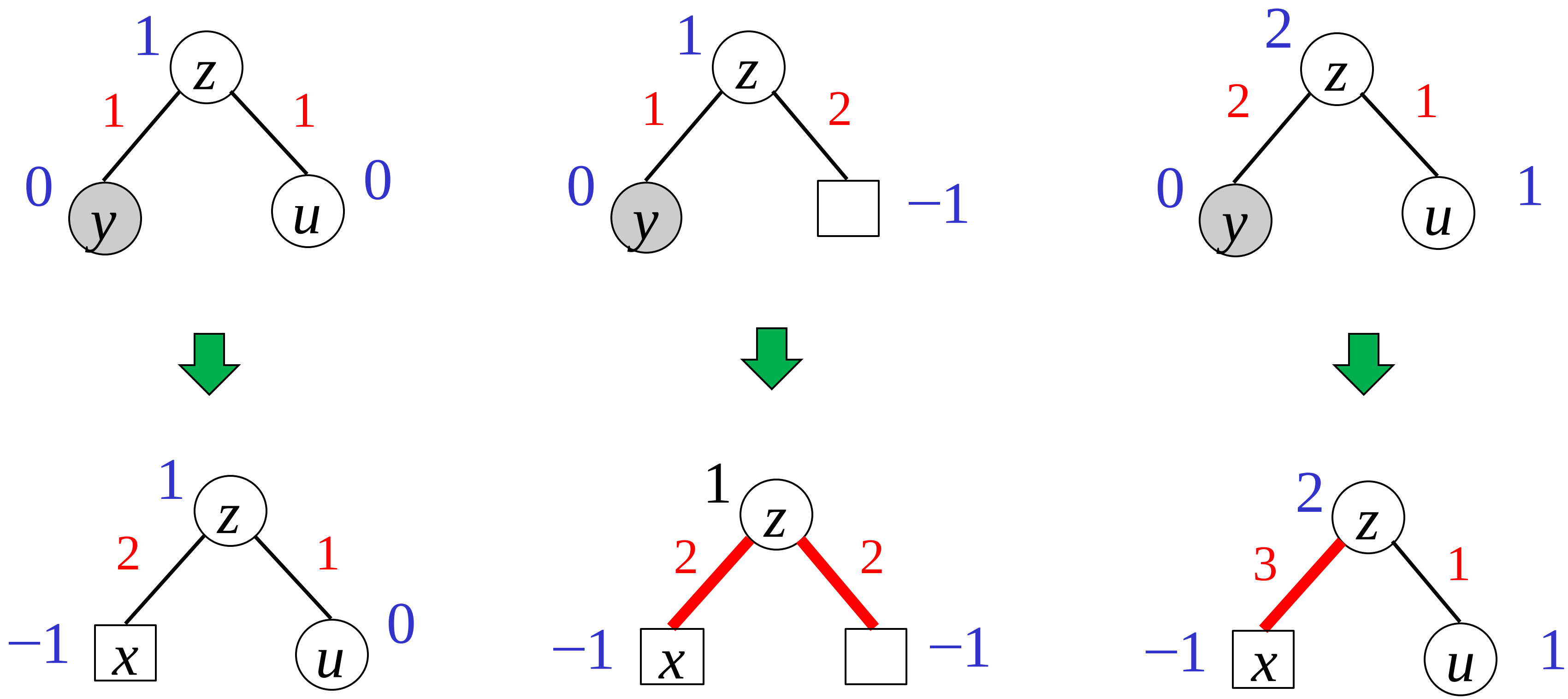
*Rotations* are not terminal cases

Total deletion time = $O(height) = O(\log n)$

What is the *amortized* cost of rebalancing?

# AVL Deletion

## Deleting a leaf *y*

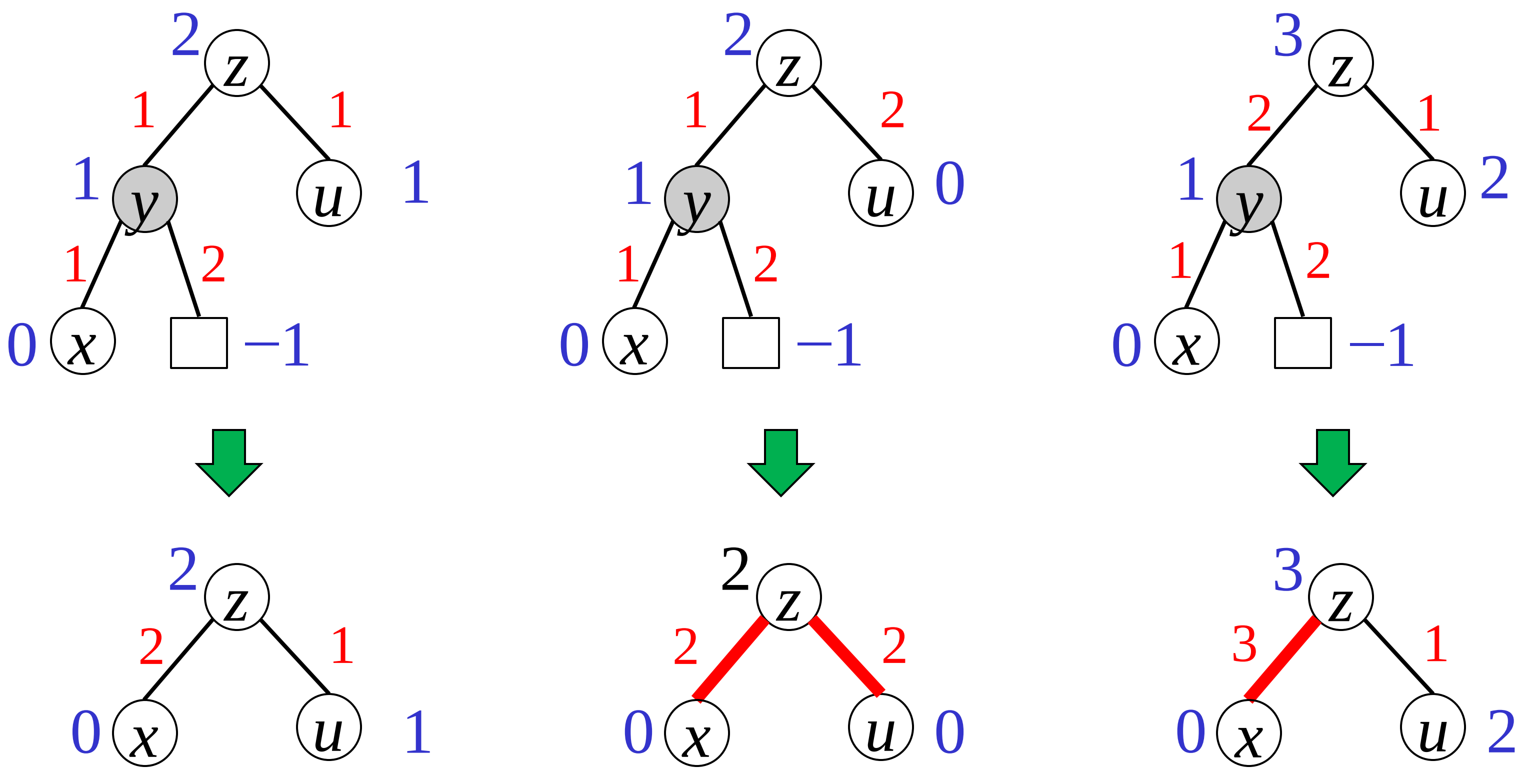



Problem
(Demote *z*)

Problem

# AVL Deletion

## Deleting a unary node $y$

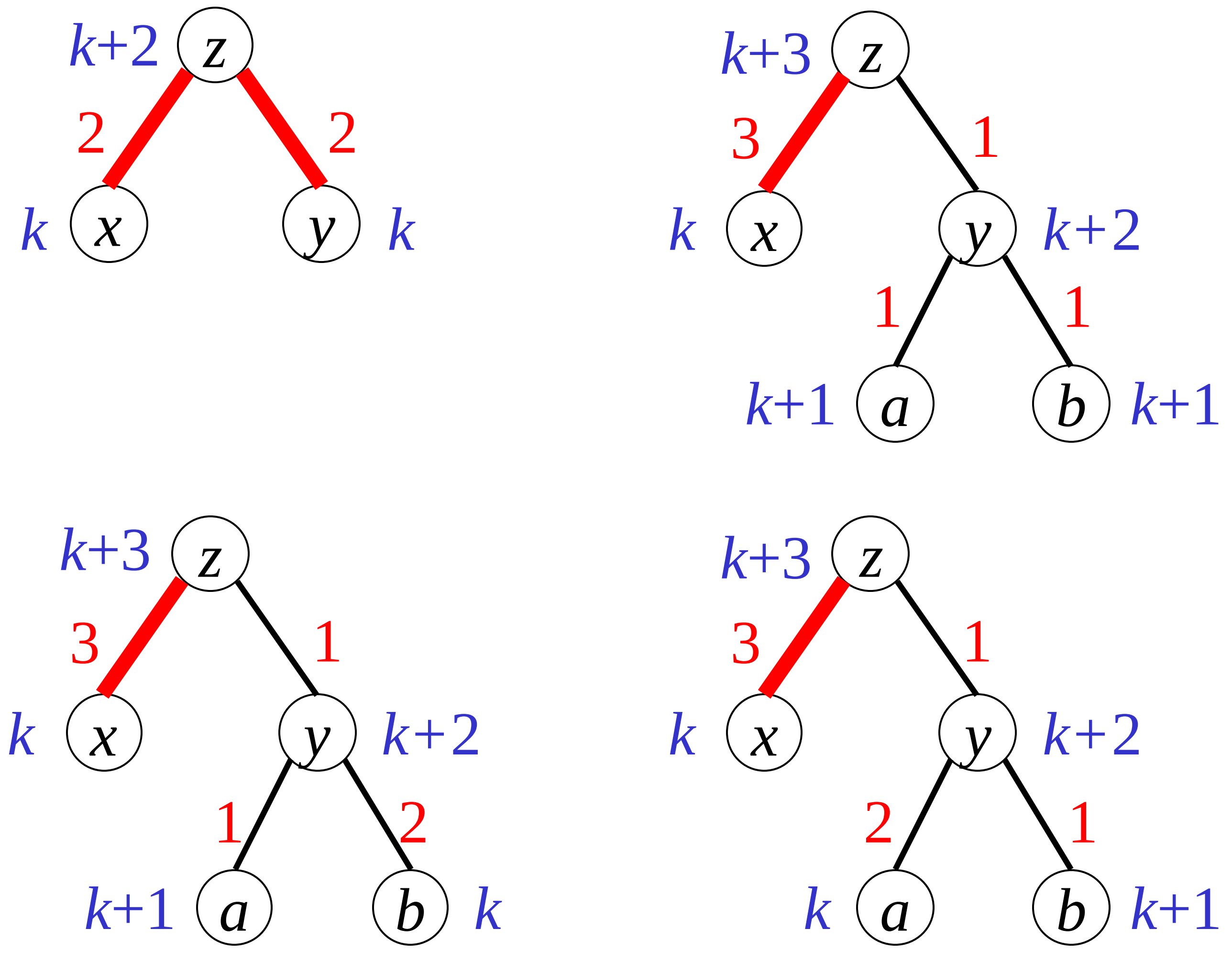



Problem
(Demote $z$)

Problem

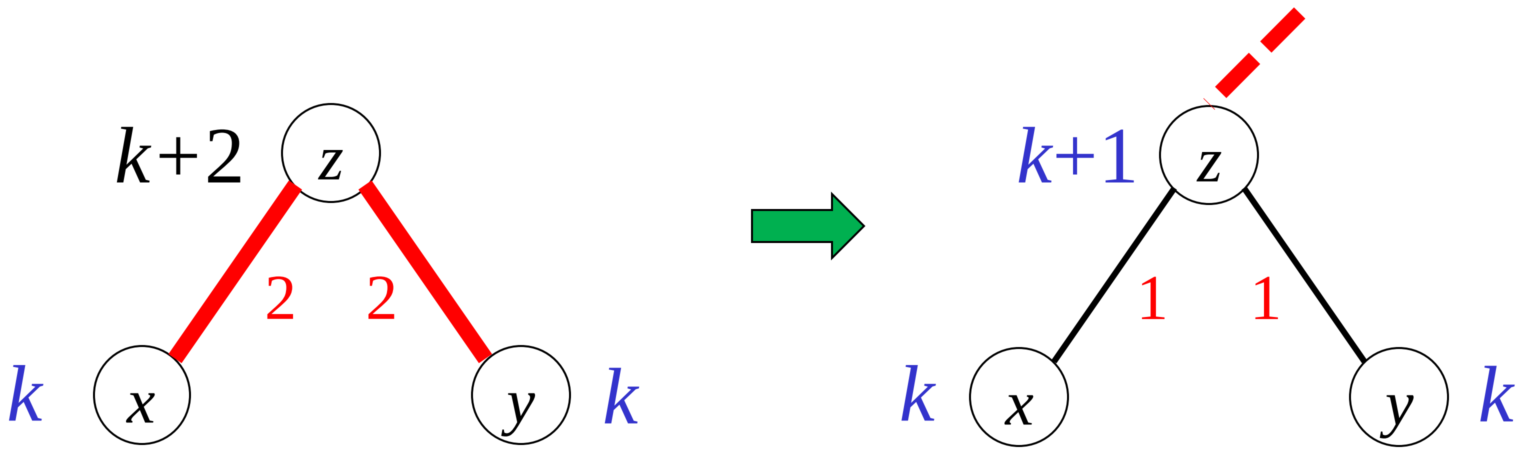
# AVL: Deletion rebalancing

## 4 cases (up to symmetry)



# AVL: Rebalancing after deletion

## **Case 1:** Demote

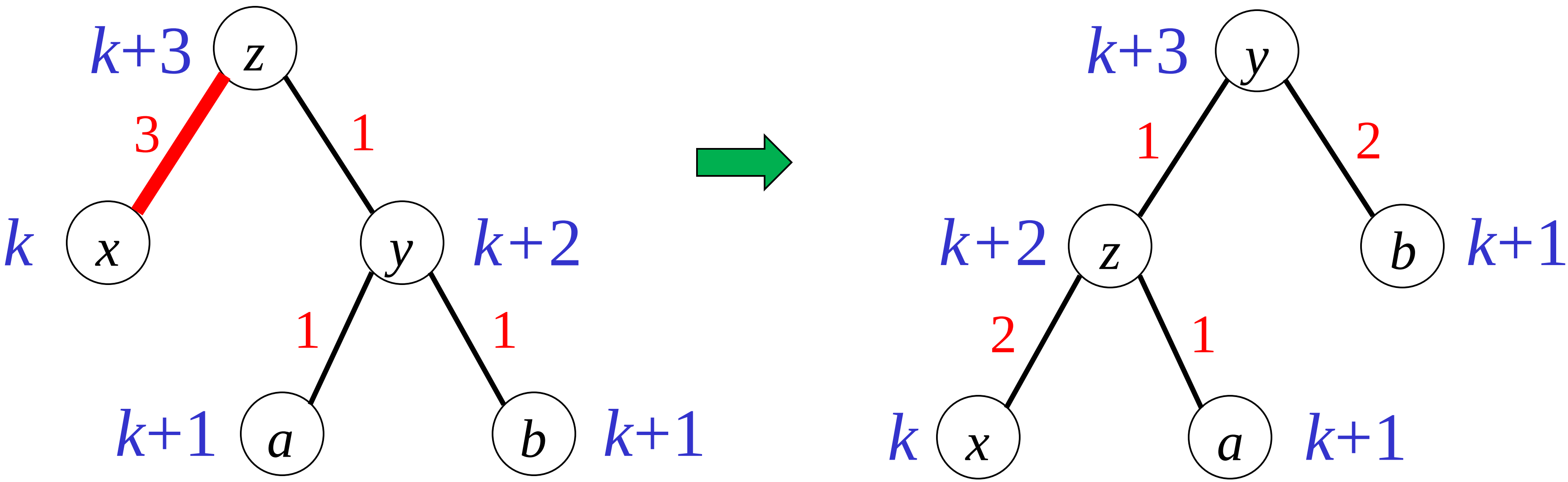



Demote $z$, i.e., decrease its *rank* by 1

Problem is either fixed or moved up

# AVL: Rebalancing after deletion

## **Case 2:** Single Rotation (a)

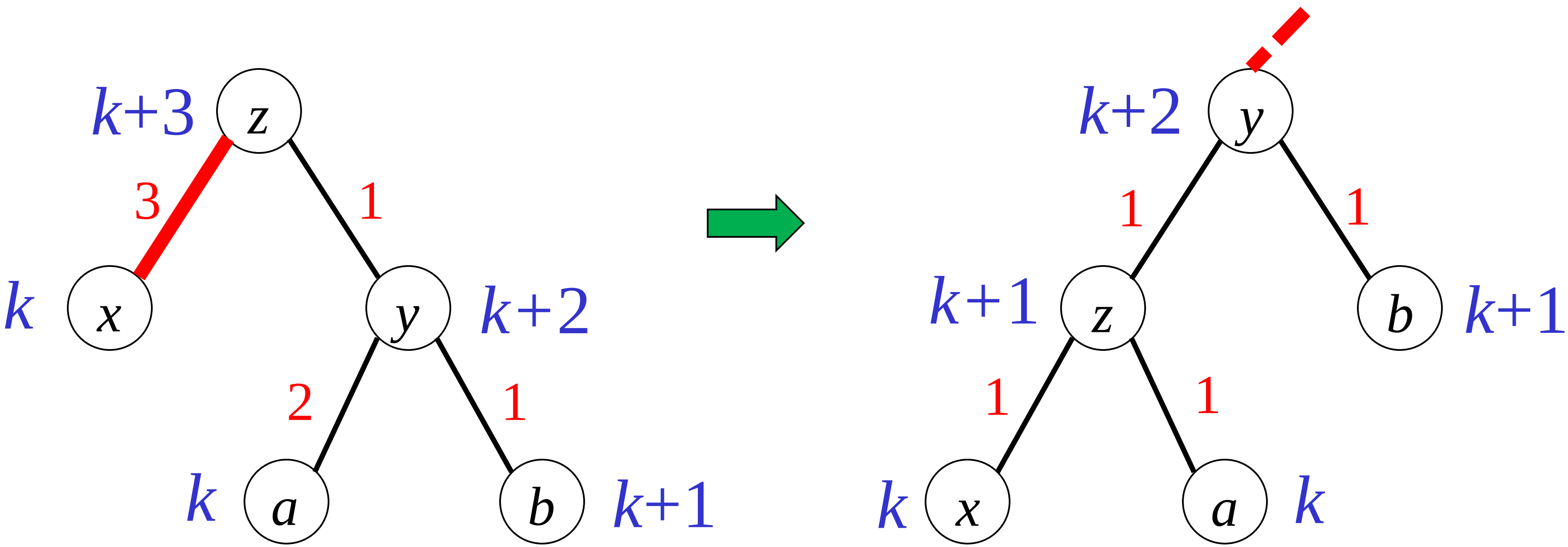



Rotate left

Demote *z* Promote *y*

# AVL: Rebalancing after deletion

## **Case 3:** Single Rotation (b)

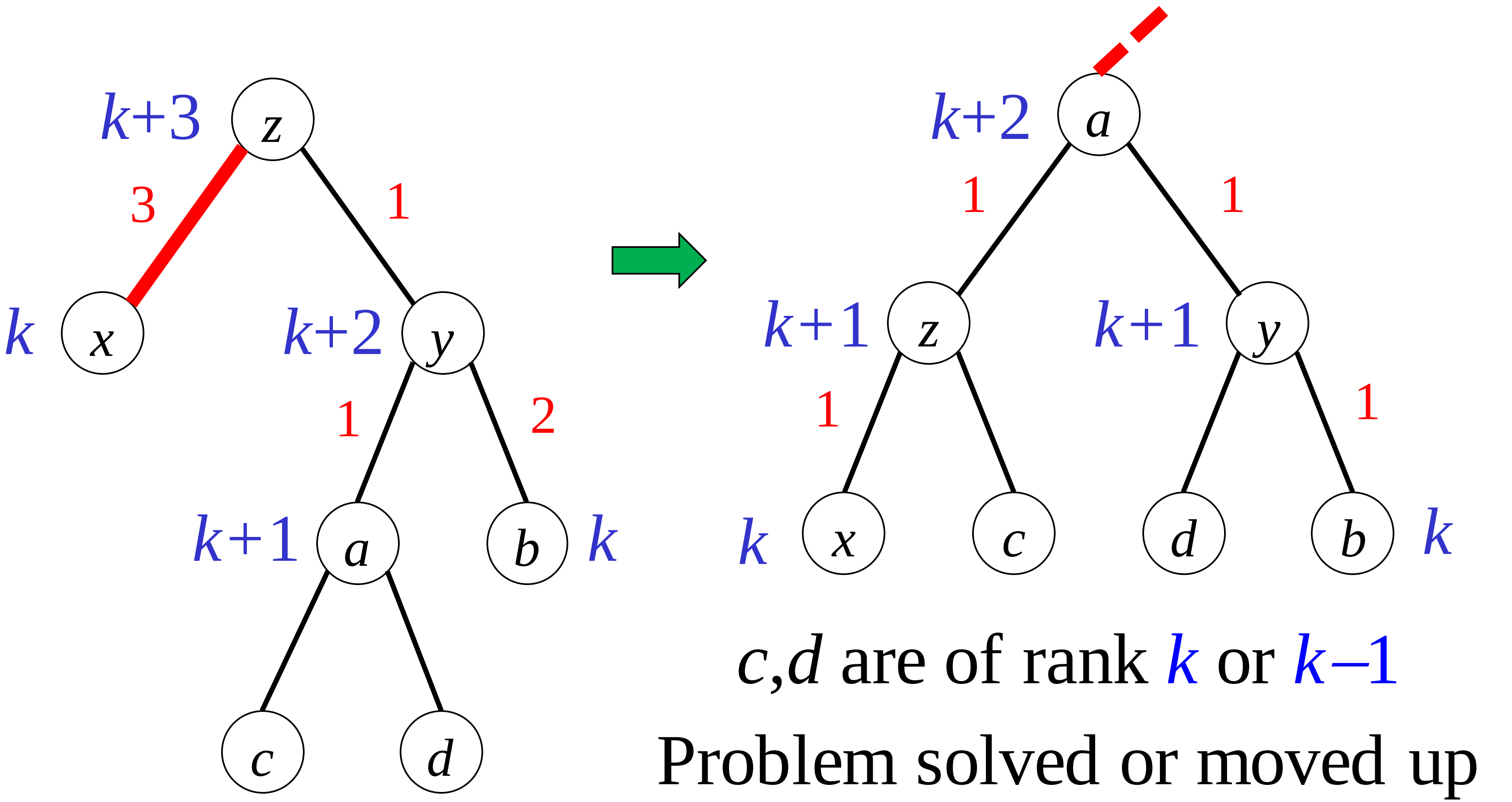



Rotate left
Demote *z* twice

Problem solved or moved up

# AVL: Rebalancing after deletion

## **Case 4:** Double Rotation




*c*,*d* are of rank $k$ or $k$–1

Problem solved or moved up

# AVL Deletion - Summary

Replace item to be deleted with
its successor or predecessor, if needed

Delete the appropriate node

Perform a sequence of *demotions*,
*rotations* and *double rotations*

Somewhat more complicated then insertion

*Rotations* are not terminal cases

Total deletion time = O(*height*) = O(log *n*)

What is the *amortized* cost of *rebalancing*?

# AVL Trees – Cost of *rebalancing*

*Worst-case* cost of *rebalancing*, after both **insertions** and **deletions**, is $O(\log n)$

If there are only **insertions**, the *amortized* cost of *rebalancing*, as we saw, is $O(1)$

If there are only **insertions**, and then only **deletions**, the *amortized* cost of *rebalancing* is again $O(1)$

But, if **insertions** and **deletions** are intermixed, the *amortized* cost of *rebalancing* may be $\Omega(\log n)$

Can the *amortized* cost of *rebalancing*, in the general case, be brought down to $O(1)$?

# WAVL trees

## [Haeupler-Sen-Tarjan (2009)]

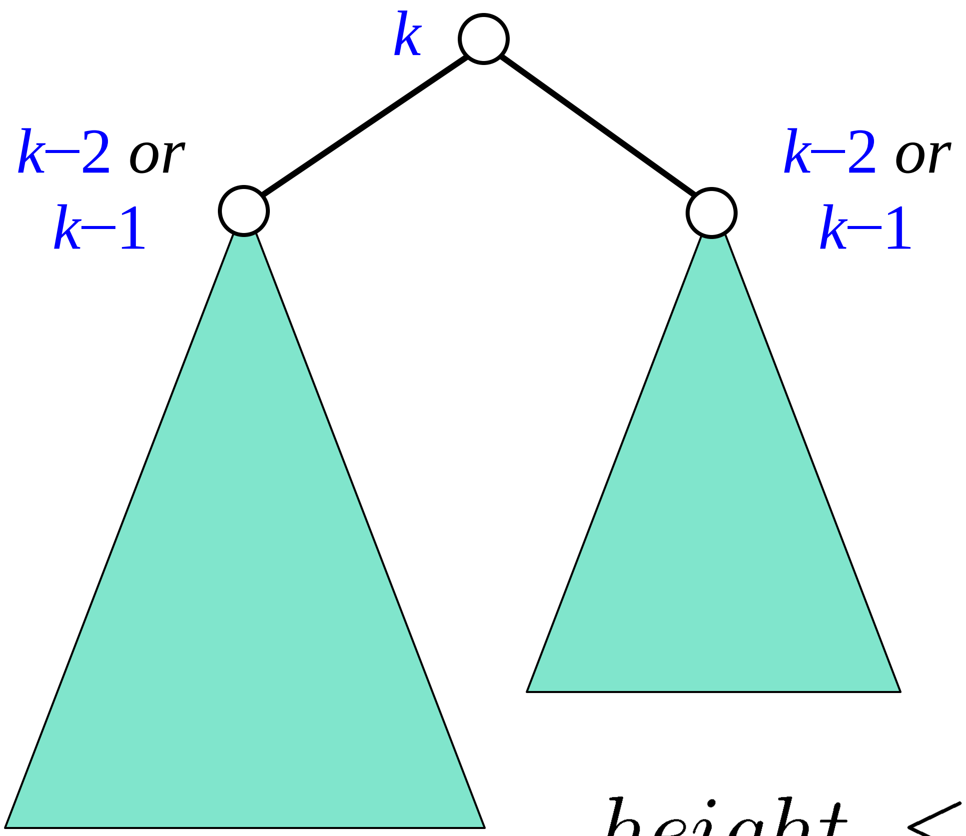
At most two *rotations*
both in insertions and deletions

*Amortized* cost of *rebalancing* is O(1)

Allow 2,2-nodes
Every (internal) leaf is still a 1,1-node

$rank \neq height$

# WAVL trees
## [Haeupler-Sen-Tarjan (2009)]




WAVL trees are search trees

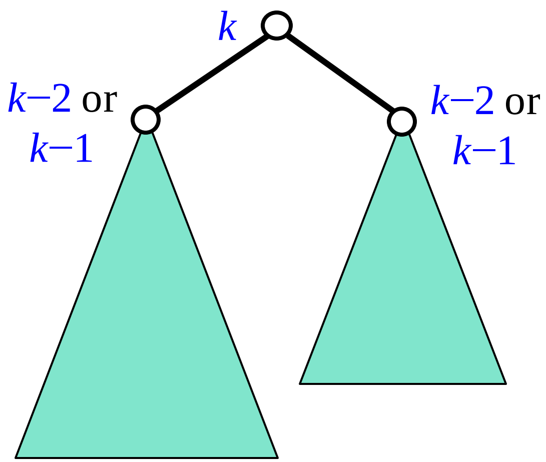
Each *rank-difference* is 1 or 2

*rank* of each (internal) leaf is 0

$$height \leq rank \leq 2\,height$$

# WAVL trees

## [Haeupler-Sen-Tarjan (2009)]




$S_k$ – minimal number of nodes in an WAVL tree of *rank* $k$

$$S_k = 2S_{k-2} + 1$$

$$S_{-1} = 0 \quad , \quad S_0 = 1$$

By induction: $S_k \geq 2^{\lceil k/2 \rceil}$

$$height \leq rank \leq 2\log_2 n$$

# WAVL Insertion

Exactly like AVL insertion

No 2,2-nodes created

2,2-nodes may be destroyed

(Check this!)

If there are only insertions,
WAVL trees are just AVL trees

# WAVL deletion

# WAVL Deletion

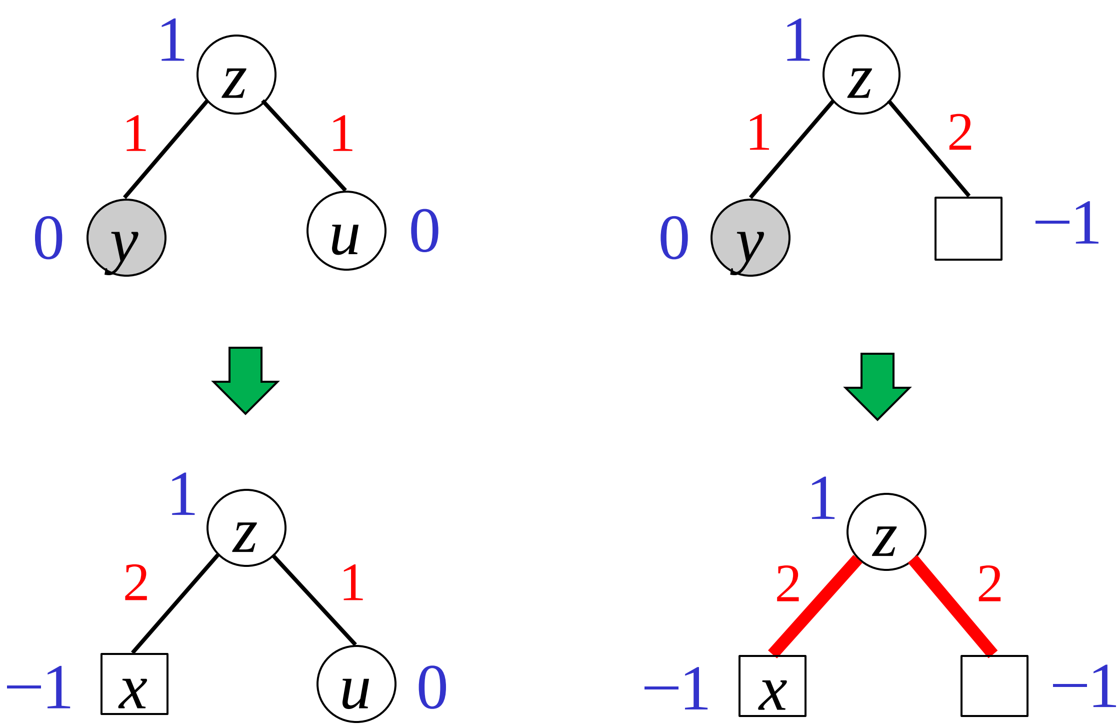
New cases to consider

Deletion becomes somewhat simpler

*Rotations* are now terminal cases

# WAVL Deletion

## Deleting a leaf *y*

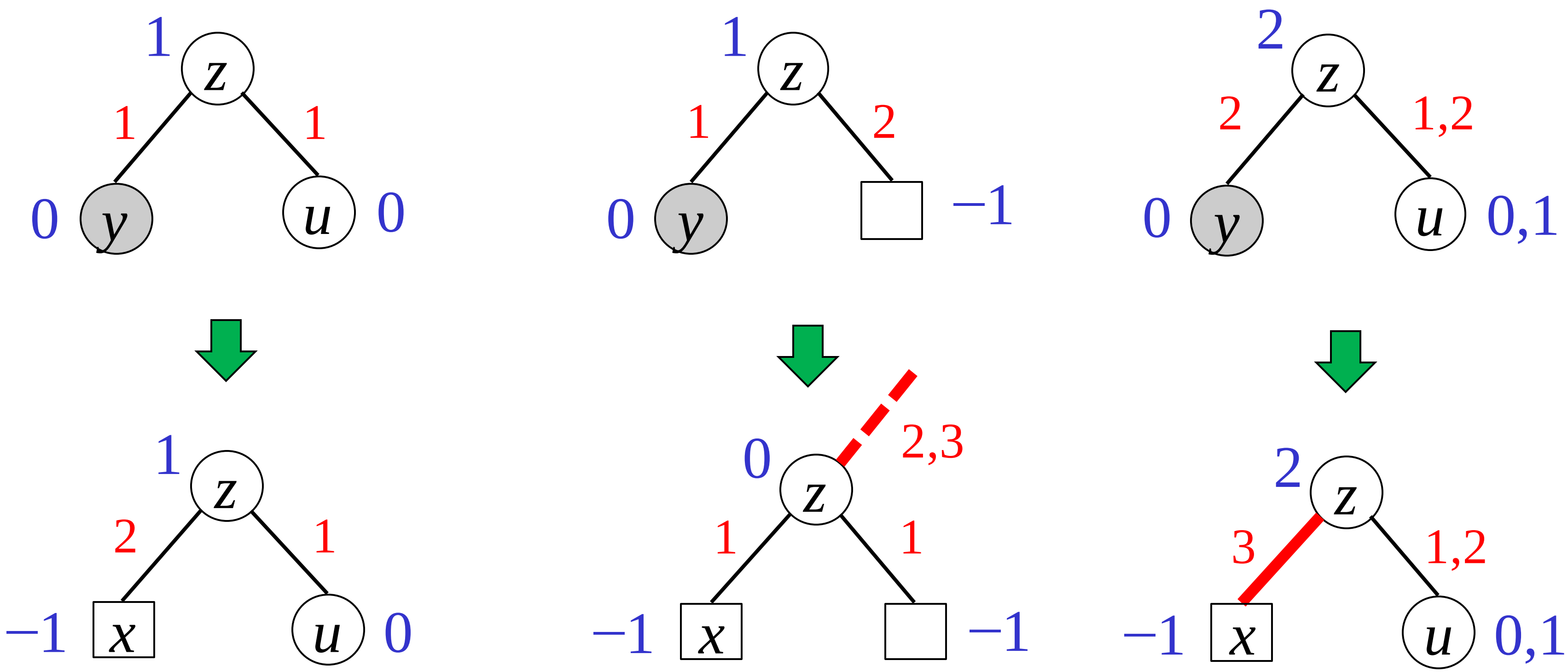



Problem! (Why?)

(Demote *z*)

# WAVL Deletion

## Deleting a leaf *y*

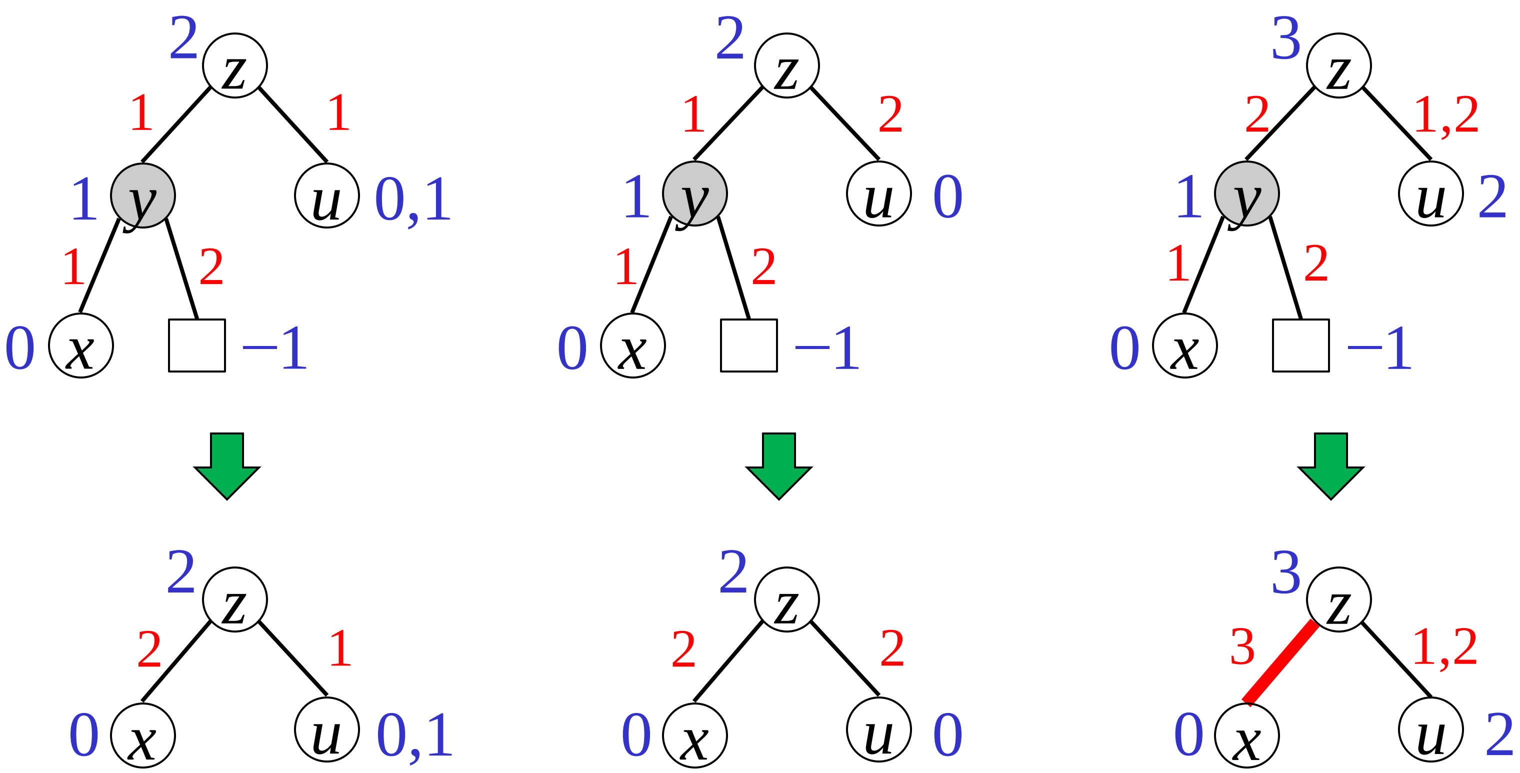



Demote *z*
(May cause a problem)

Problem

# WAVL Deletion

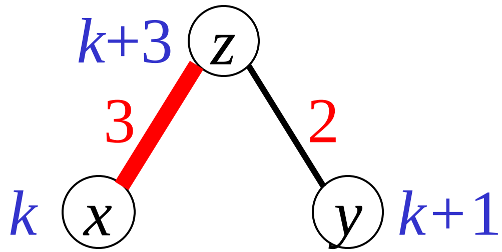
## Deleting a unary node *y*



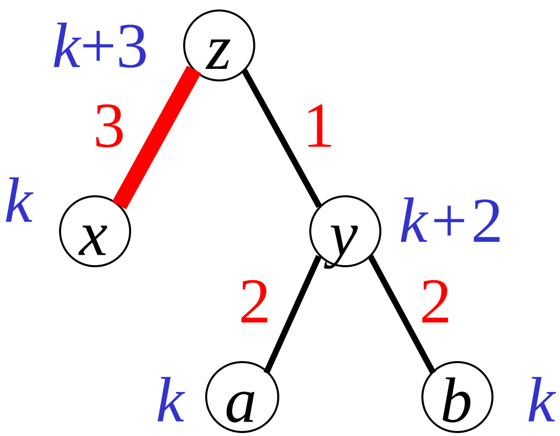

Problem

# WAVL: Deletion rebalancing cases

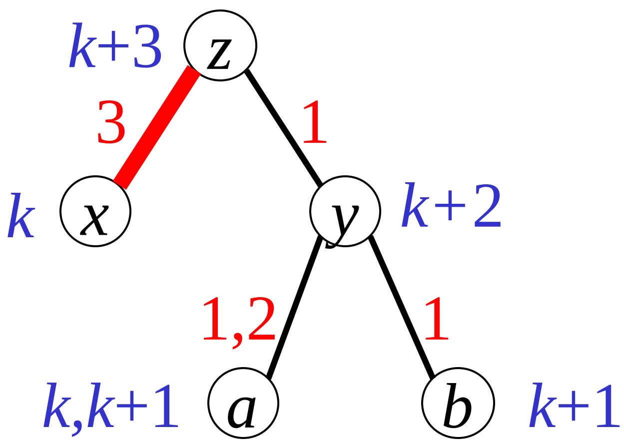
## 4 cases (up to symmetry)



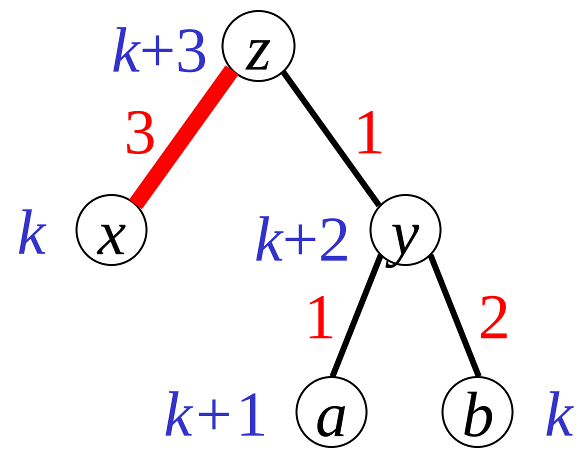

Case 1: Demote




Case 2: Double Demote



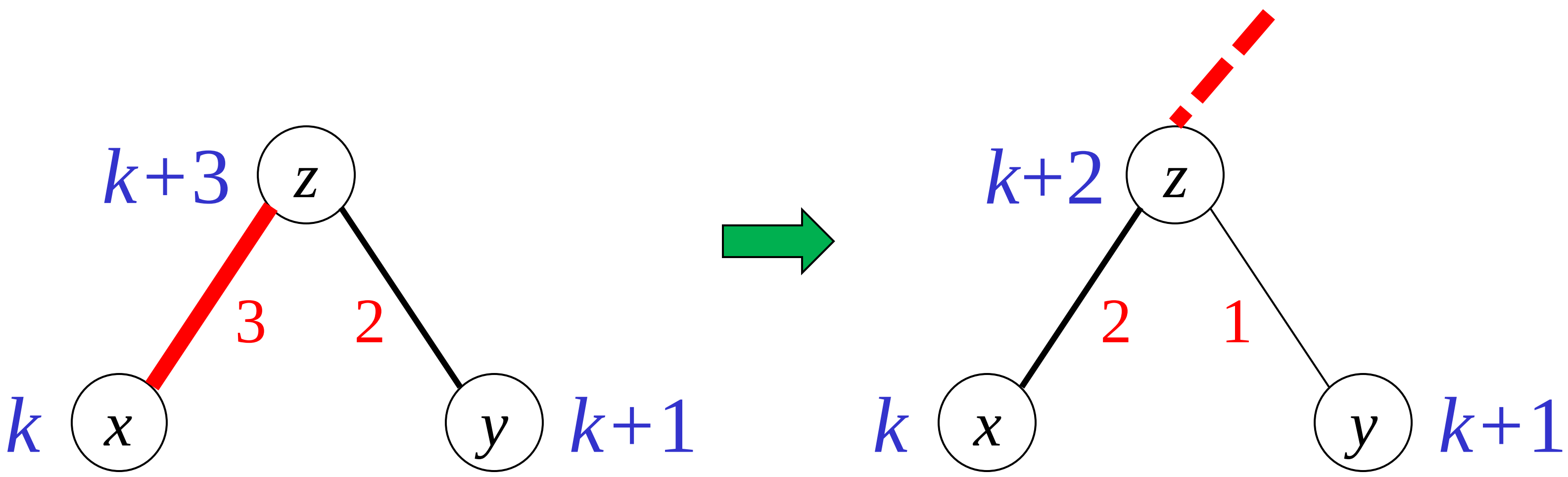

Case 3: Rotate




Case 4: Double Rotate

# WAVL: Rebalancing after deletion

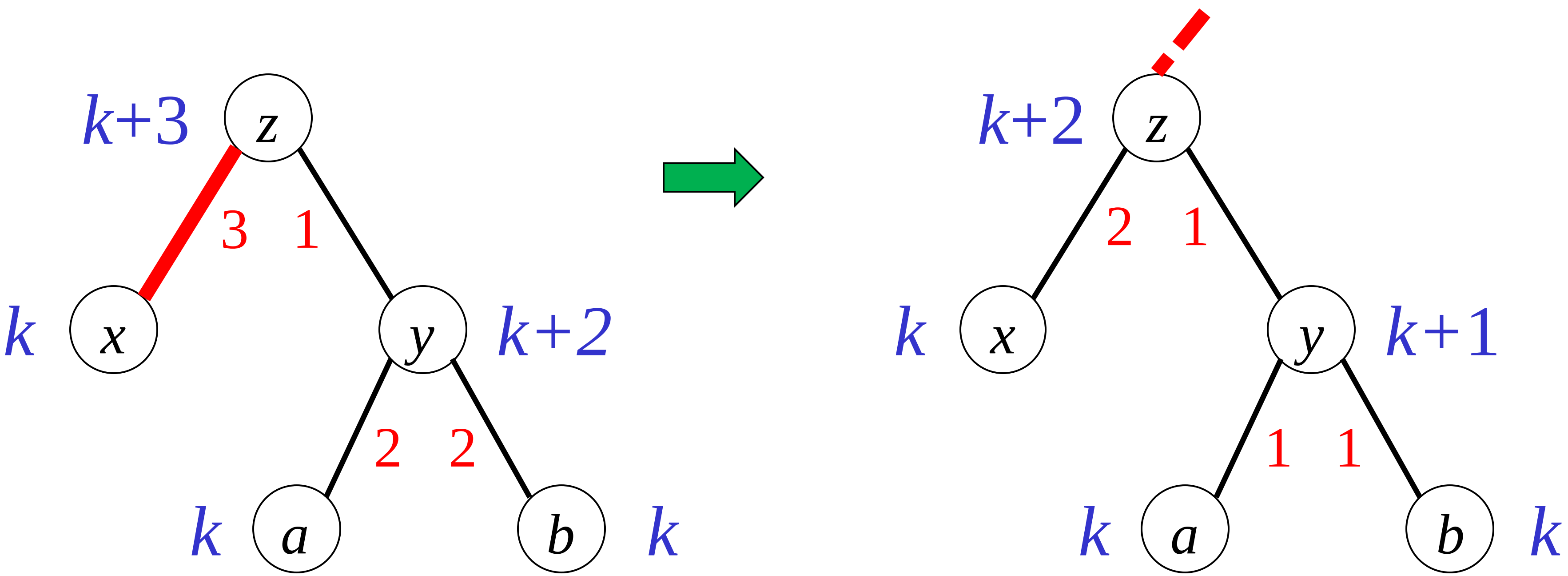
## **Case 1:** Demote




Problem is either fixed or moved up

# WAVL: Rebalancing after deletion

## **Case 2:** Double demote

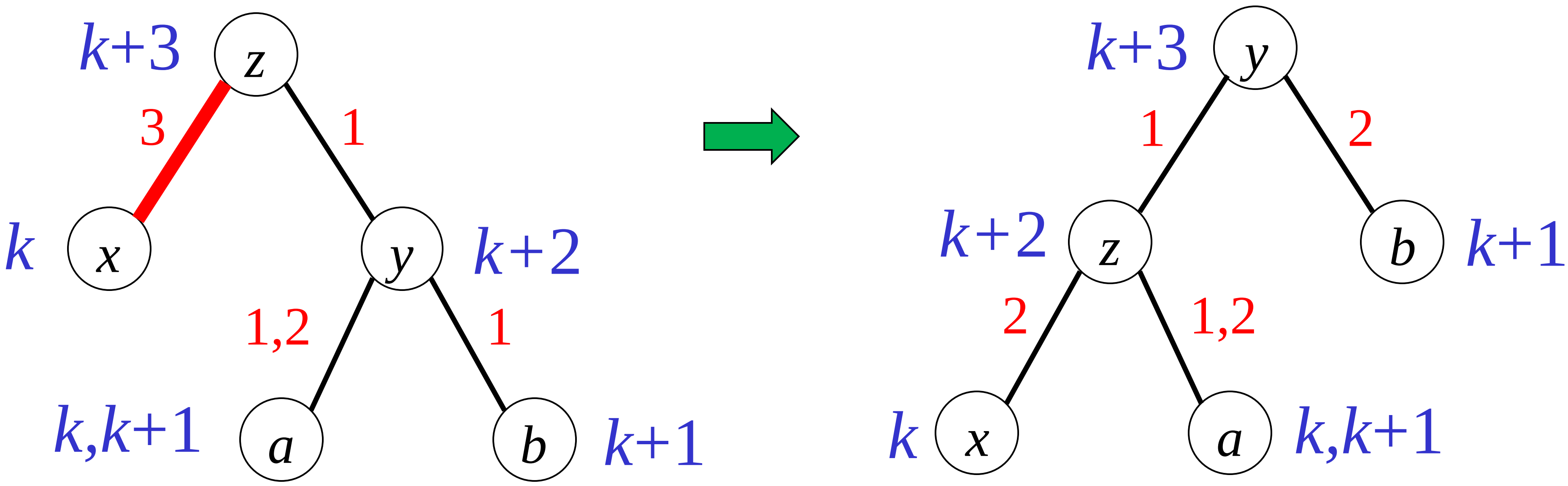



Demote *z* and *y*

Problem either solved or moved up

# WAVL: Rebalancing after deletion

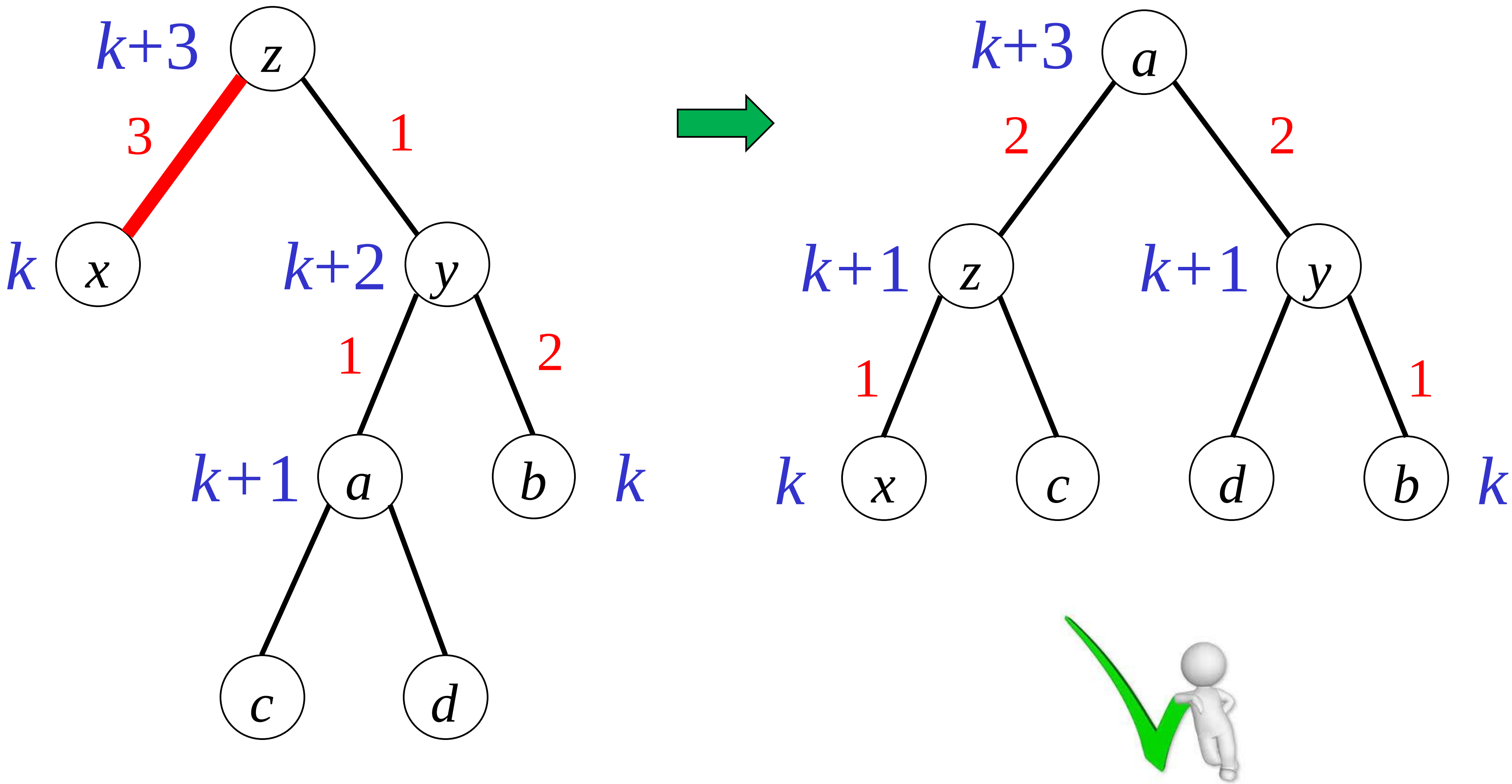
## **Case 3:** Rotate




If $z$ is a 2,2-leaf, demote it

# WAVL: Rebalancing after deletion

## **Case 4:** Double Rotate



# WAVL Insertion/Deletion - Summary

\# *promotions/demotions* $\leq$ *height* $= O(\log n)$

Number of *rotations* $\leq 2$

Worst-case time $= O($*height*$) = O(\log n)$

What is the *amortized* number of *rebalancing steps*?

# WAVL Insertion/Deletion

## *Amortized* number of *balancing steps*

$$Potential = \Phi =$$
$$\text{(number of 0,1- and 1,1-nodes)}$$
$$+\ 2 \times \text{(number of 3,2- 2,2-nodes)}$$

Insertions/Deletions themselves, and each rebalancing step, change the potential by at most a constant

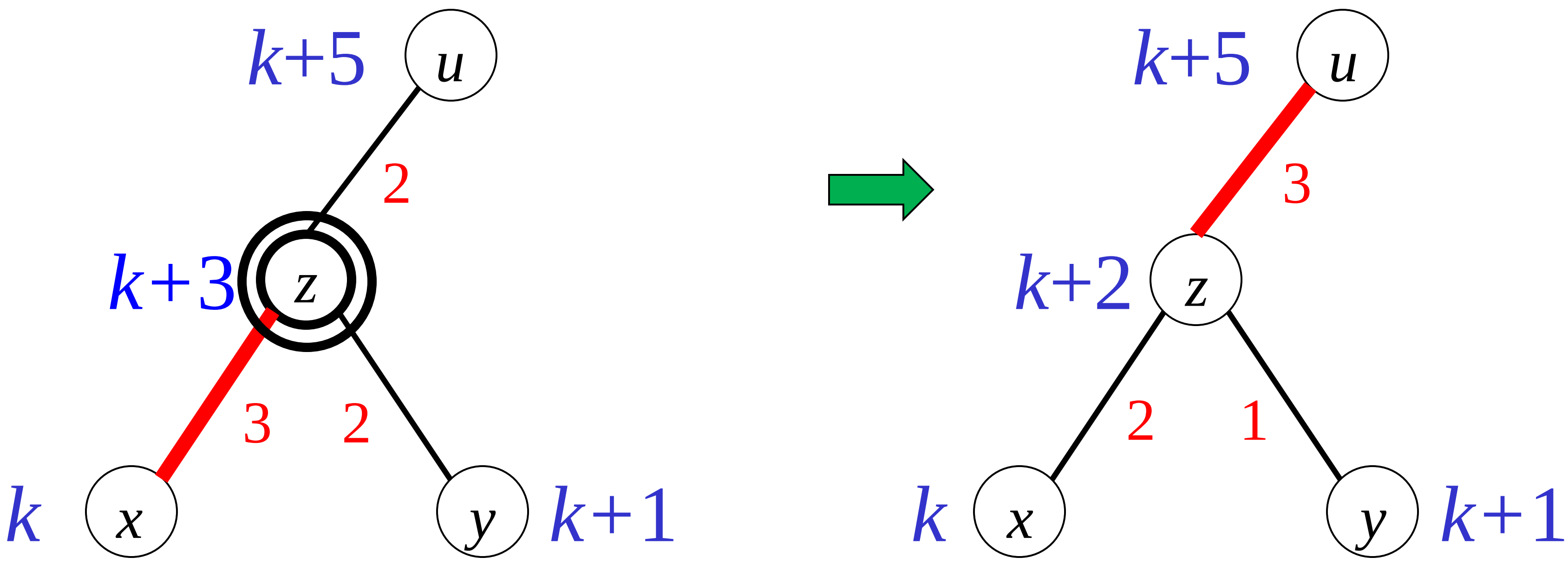
Promotions/Demotions are the only *non-terminal* steps

*Non-terminal* steps ***decrease*** the potential

$$amort(\ \#steps\ ) = O(1)$$

# WAVL: Rebalancing after deletion

## *Non-terminal* Demote

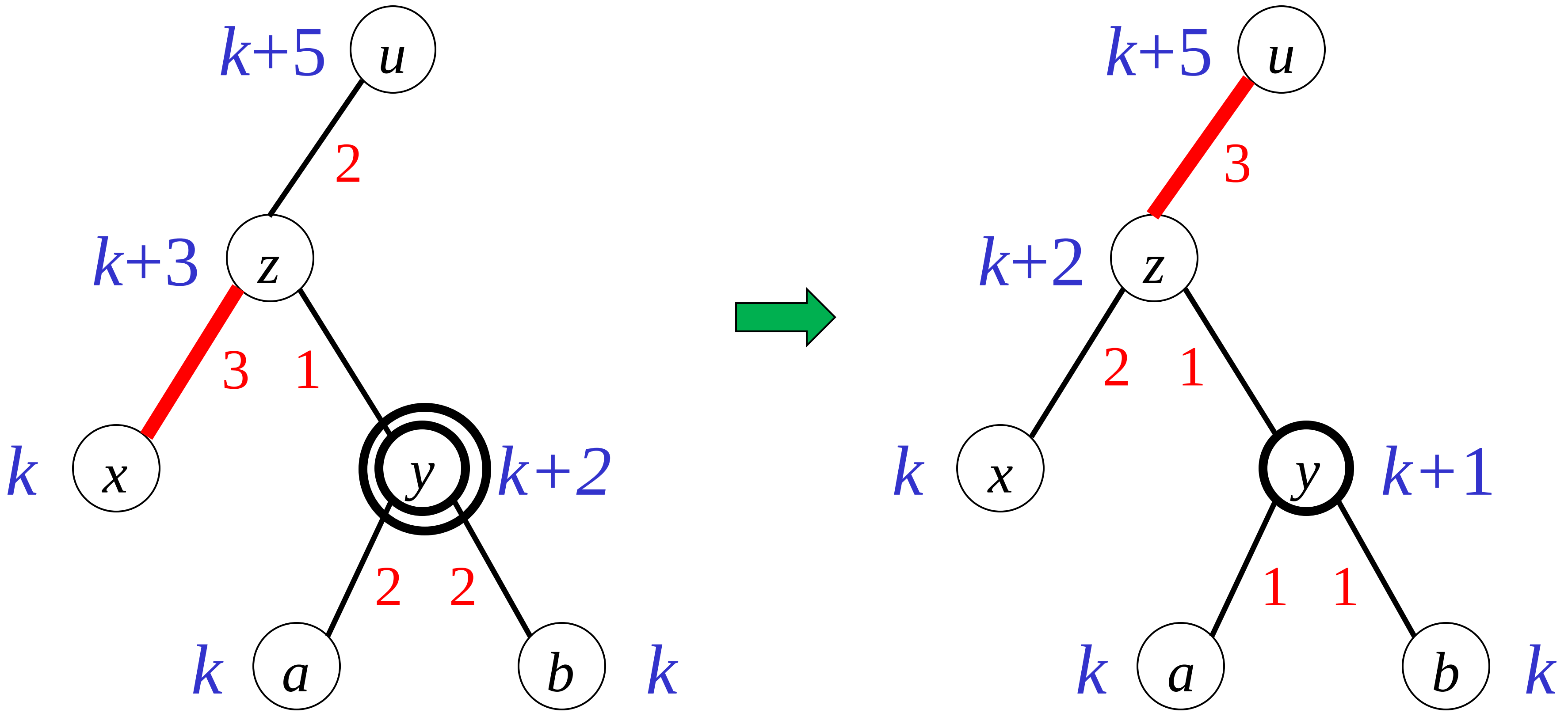



Potential of $u$ (and of $x$,$y$) does not change

Potential of $z$ drops from 2 to 0: $\Delta\Phi = -2$

# WAVL: Rebalancing after deletion

## *Non-terminal* Double Demote




Potential of $y$ drops from 2 to 1: $\Delta\Phi = -1$

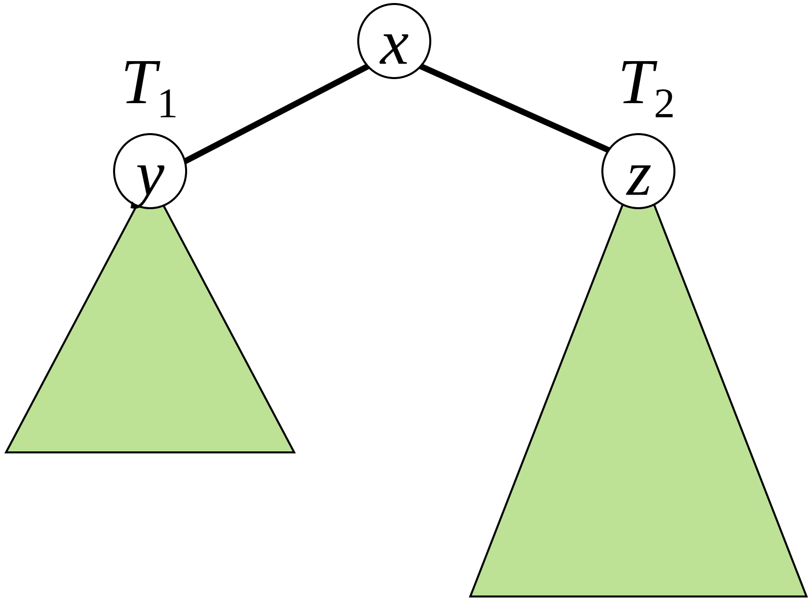
# AVL vs. WAVL

| | Depth | Number of Rotations per update | Amortized cost of rebalancing |
|---|---|---|---|
| AVL | $1.45 \log_2 n$ | $O(\log n)$ | $O(\log n)$ |
| WAVL | $2 \log_2 n$ | 2 | $O(1)$ |

**Theorem:** The depth of a WAVL tree generated by a sequence of $m$ insertions, and an arbitrary number of deletions, is at most $\log_\phi m \le 1.45 \log_2 m$

# Joining and Splitting binary search trees

# **Joining** two binary search trees




Join($T_1$,$x$,$T_2$)

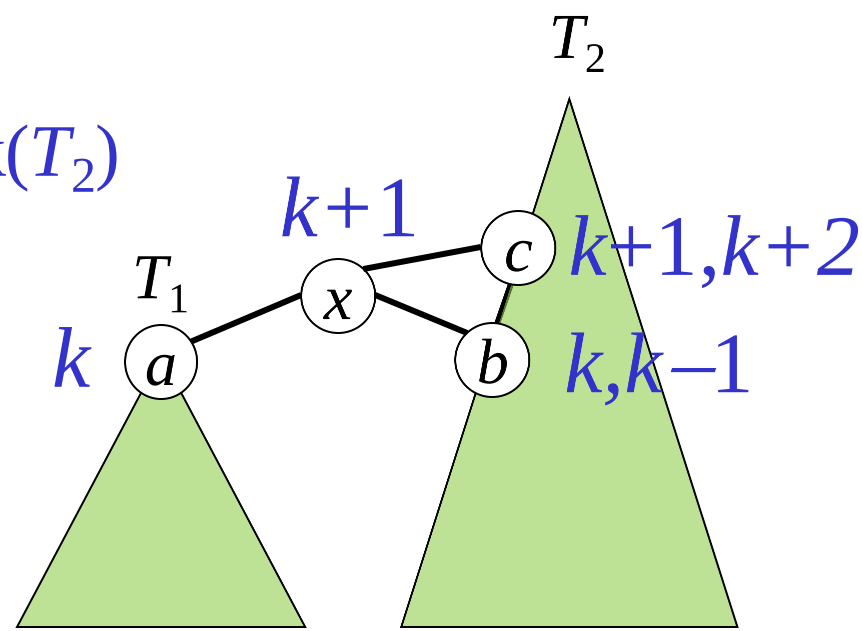
Suppose that all keys in $T_1$ are less than $x.key$ and that all keys in $T_2$ are greater than $x.key$

The tree formed is a valid search tree, but may be very unbalanced, even if $T_1$ and $T_2$ are balanced

# **Joining** two (W)AVL trees efficiently

Assume

$rank(T_1) \leq \mathrm{rank}(T_2)$




$\mathrm{Join}(T_1, x, T_2)$

$b$ – first vertex on the left spine of $T_2$ with $rank \leq k$

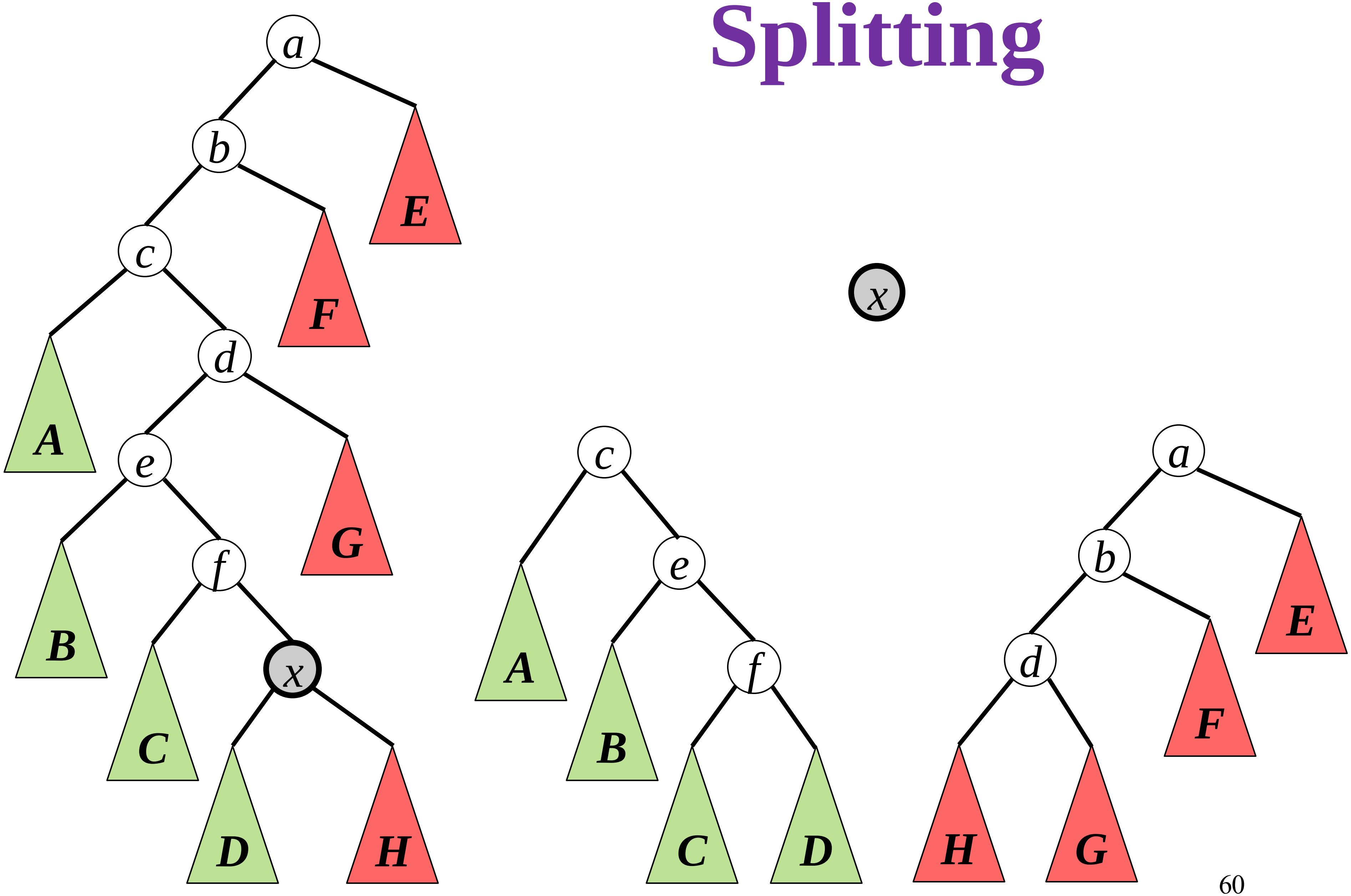
Do rebalancing from $x$, if needed

$O(\log n)$ time

$O(\ rank(T_2) - rank(T_1) + 1\ )$ time

(if *ranks* maintained explicitly)

# Splitting



# Splitting with efficient joins

Suppose we need to join $T_1, T_2, \ldots, T_k$
where $rank(T_1) \le rank(T_2) \le \ldots \le rank(T_k)$

Suppose $rank(\text{Join}(T_1,\ldots,T_i)) \le rank(T_i) + c$

$$O\left(\sum_{i=2}^{k} \left| rank(T_i) - rank(Join(T_1, \ldots, T_{i-1})) \right| + 1\right)$$
$$= O\left(\sum_{i=2}^{k} rank(T_i) - rank(T_{i-1}) + 1\right)$$
$$= O\left(rank(T_k) - rank(T_1) + k\right) = O(\log n)$$

# Rank and Select

# Additional dictionary operations

Select($D$,$i$) – Return the $i$-th largest item in $D$ (indices start from 0)

Rank($D$,$x$) – Return the ***RANK*** of $x$ in $D$, (i.e., the number of items $x$ is larger than)

Can we still use **(W)AVL** trees?

Keep sub-tree sizes!

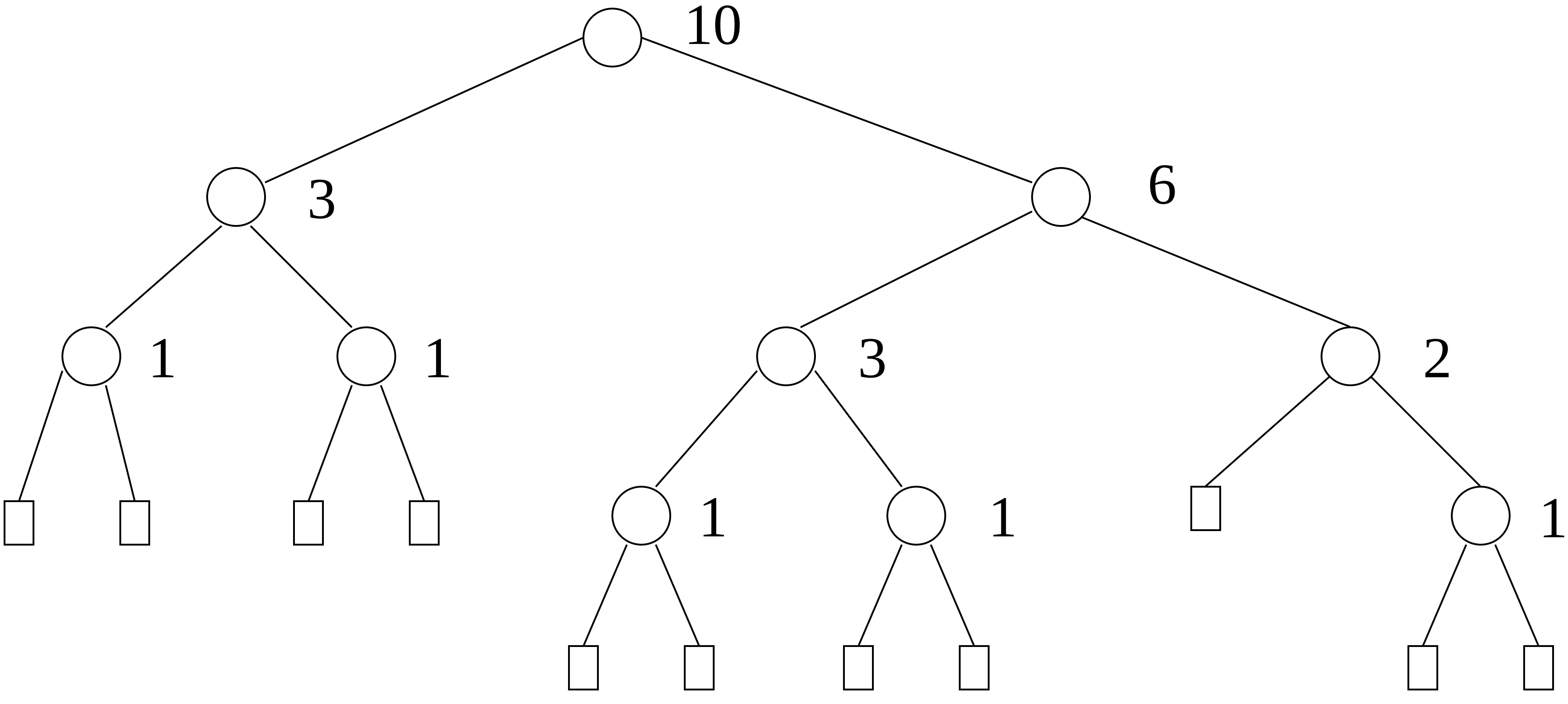
# Caution!

Rank now has two meanings…

Which is unfortunate…

This is the established terminology…

# Sub-tree sizes




$$x.size = x.left.size + x.right.size + 1$$

$$EXT.size = 0$$

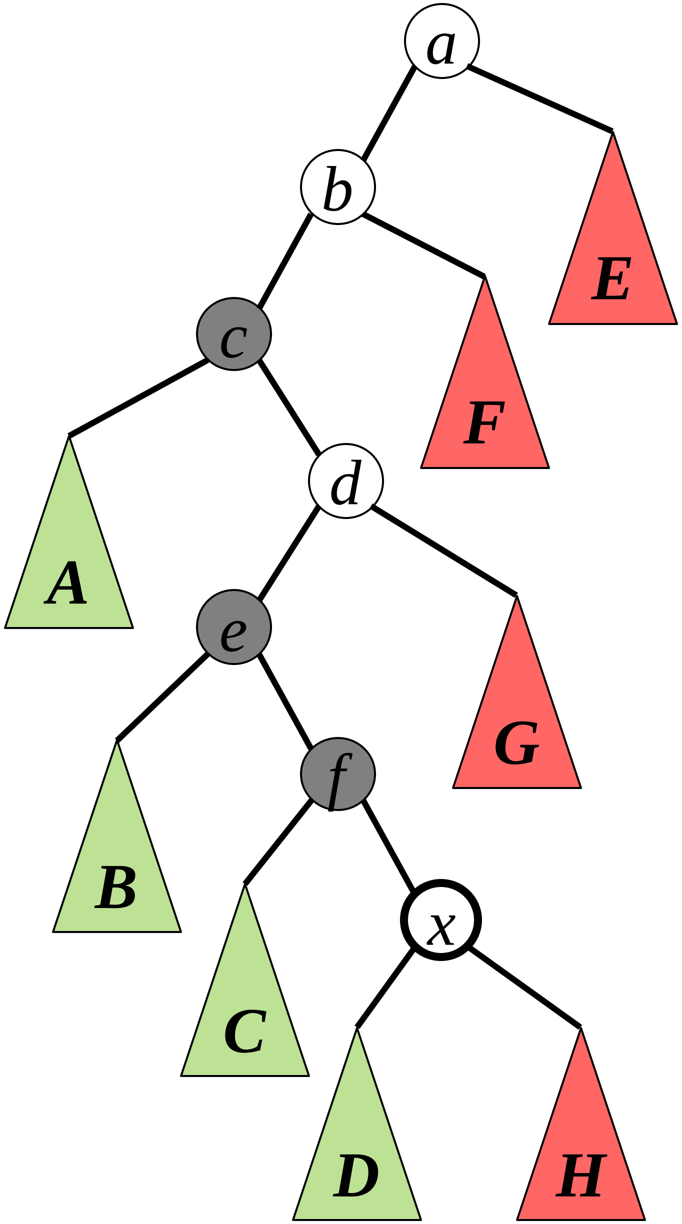
# Selection

**Function** `Select`$(x, i)$

$r \leftarrow x.left.size$

**if** $i = r$ **then**
  **return** $x$
**else if** $i < r$ **then**
  **return** `Select`$(x.left, i)$
**else**
  **return**
  `Select`$(x.right, i - r - 1)$

Note: $0 \leq i < n$




# RANK

$$RANK(x) = (size(A)+1) + (size(B)+1) + (size(C)+1) + size(D)$$

# RANK

**Function** $\texttt{Rank}(T, x)$

$r \leftarrow x.left.size$

$y \leftarrow x$

**while** $y \neq T.root$ **do**

  **if** $y = y.parent.right$ **then**

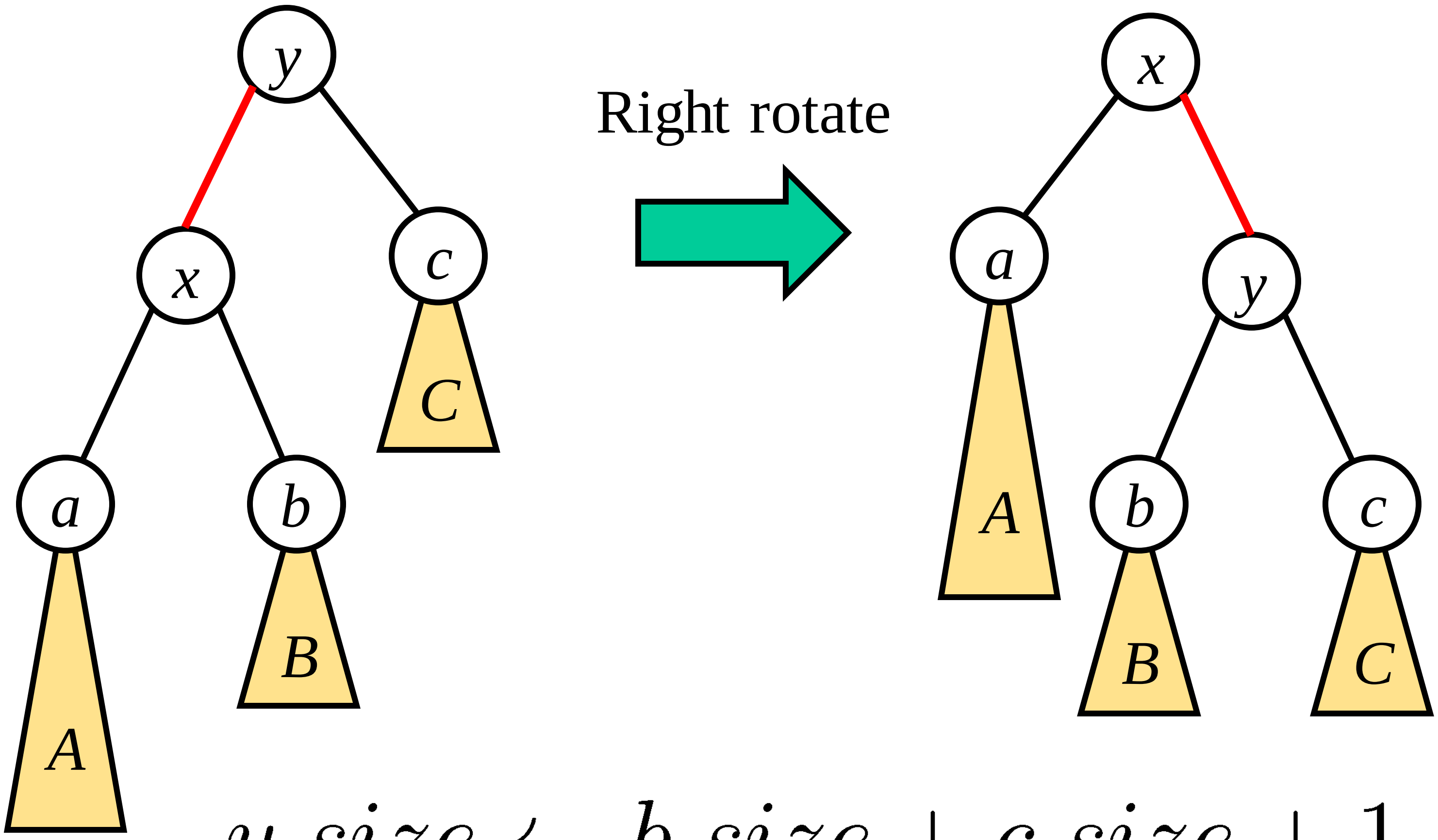
    $r \leftarrow r + y.parent.left.size + 1$

  $y \leftarrow y.parent$

**return** $r$

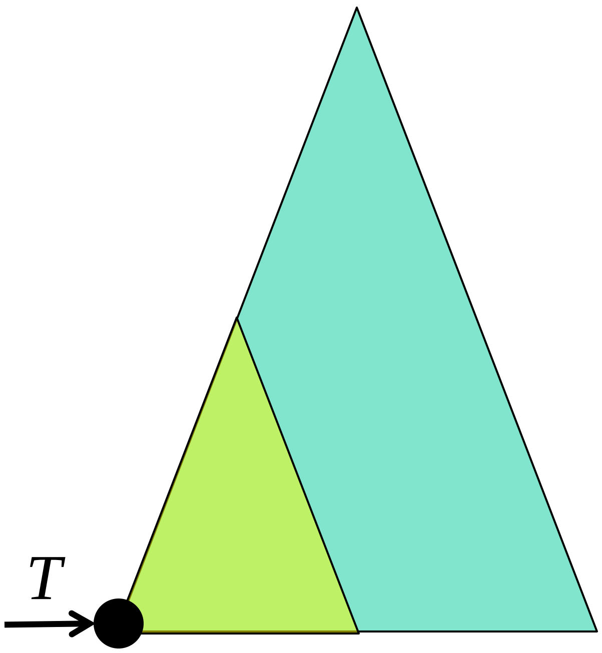
Recall that *EXT.size*=0

# Easy to maintain sizes




$$y.size \leftarrow b.size + c.size + 1$$
$$x.size \leftarrow a.size + y.size + 1$$

# Finger Search trees

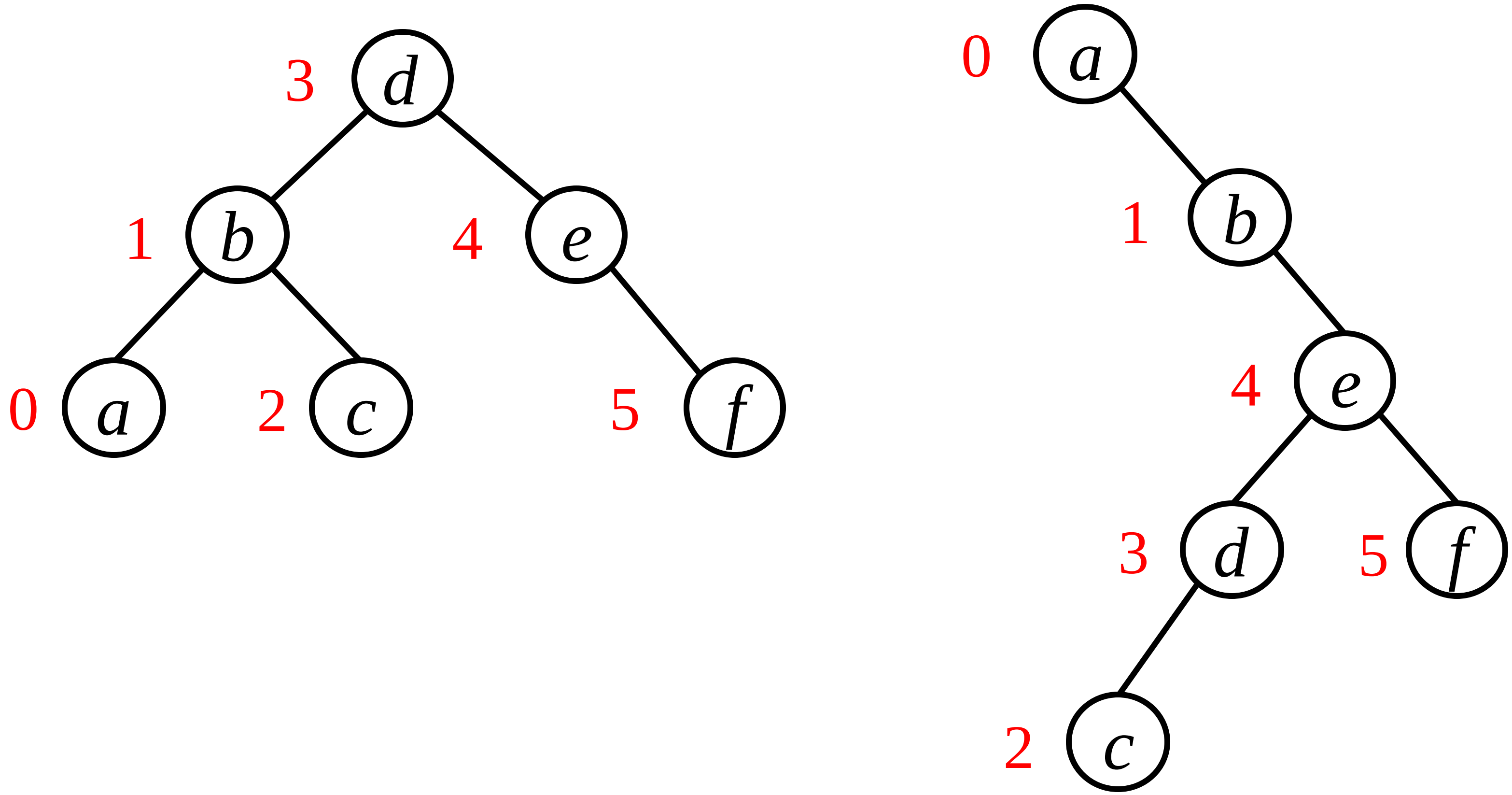



Maintain a pointer to the **minimum** element

*Select*($T$,$k$) in O(log $k$) time

# Lists as Trees

$[\ a\ b\ c\ d\ e\ f\ ]$



# Lists as Trees

Maintain the items in a tree:
*i*-th item in node of rank *i*

List-Node → Tree-Node

Tree-Nodes have no explicit keys
(Implicitly maintained ranks play the role of keys)

Retrieve(*i*) → Select(*i*)

Select, Insert-Rebalance and Delete-Rebalance
do not use keys

# Lists as Trees: Insert

To insert a node $z$ in the $i$-th position, where $0 \leq i < n$:

Find the current node of rank $i$.

If it has no left child, make $z$ its left child.

Otherwise, find its predecessor and make $z$ its right child.

To insert a node $z$ in the last position ($i = n$):

Find the last node and make $z$ its right child

Fix the tree

# Lists as Trees: Delete

Delete a node *z* in the same way
a node is removed from a search tree

# Implementation of lists

| | Circular arrays | Doubly Linked lists | Balanced Trees |
|---|---|---|---|
| Insert/Delete-First/Last | O(1) | O(1) | O(log $n$) |
| Insert/Delete($i$) | O($i$+1) | O($i$+1) | O(log $n$) |
| Retrieve($i$) | O(1) | O($i$+1) | O(log $n$) |
| Concat | O($n$+1) | O(1) | O(log $n$) |
| Split($i$) | O($i$+1) | O($i$+1) | O(log $n$) |

O($i$+1) can be replaced by O(min{$i$+1,$n$−$i$})

# Splay Trees (Self-adjusting trees)

[Sleator-Tarjan (1983)]

Do not maintain any balance!

When a node is accessed, **splay** it to the root

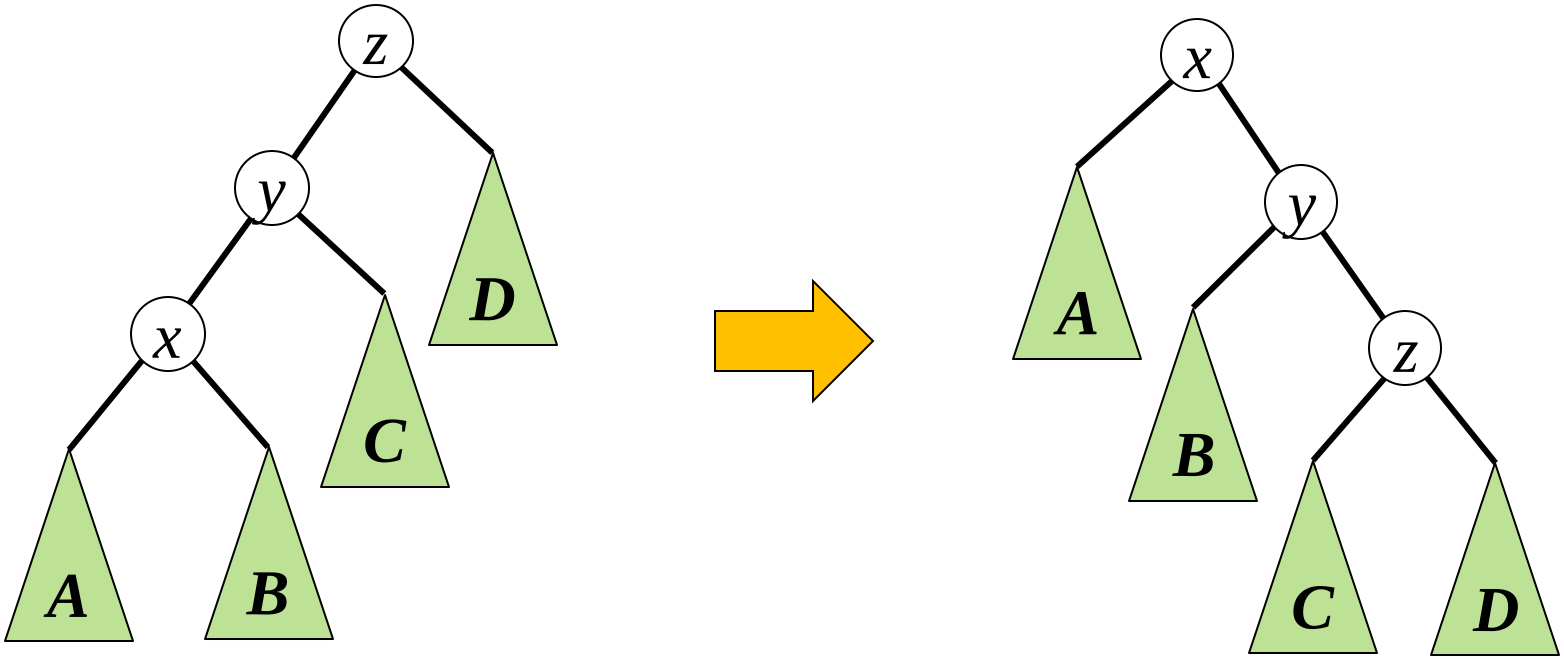
A node is splayed using a sequence of **zig-zig** and **zig-zag** steps

*Amortized* cost of each operation is $O(\log n)$

Total cost of $n$ operation is $O(n \log n)$

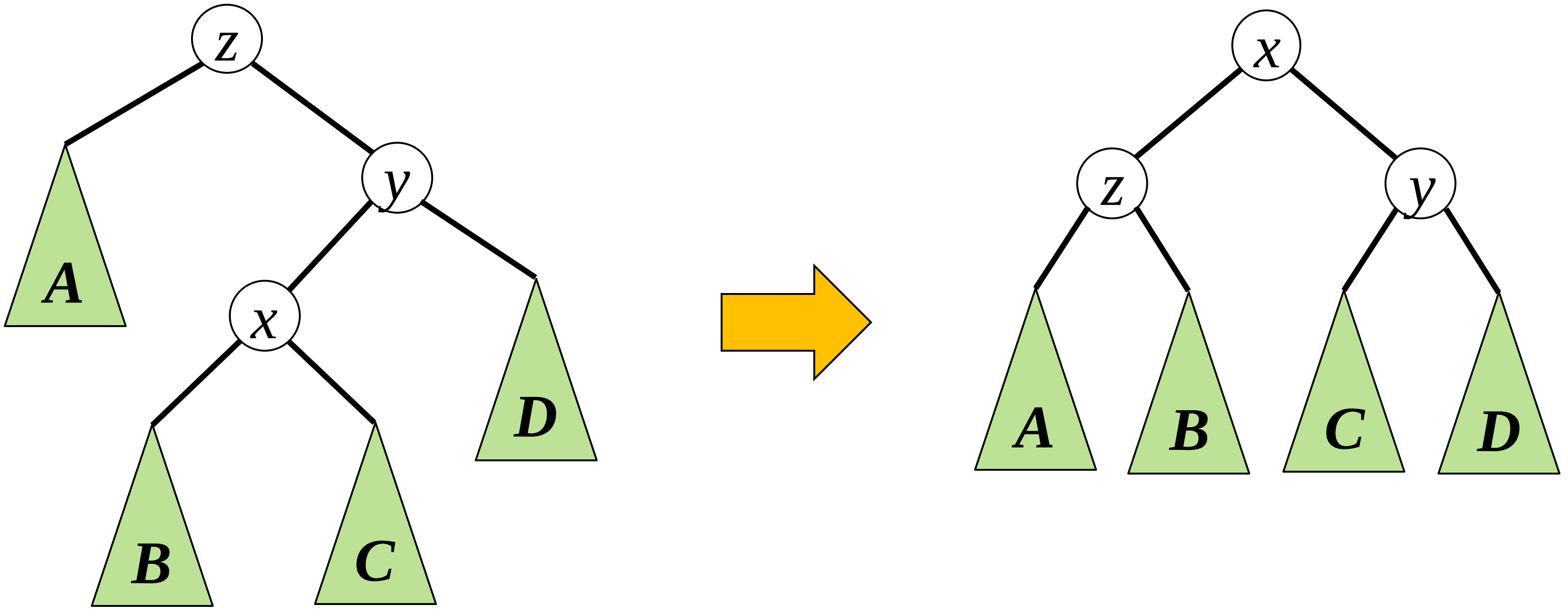
Many other amazing properties

# Zig-Zig



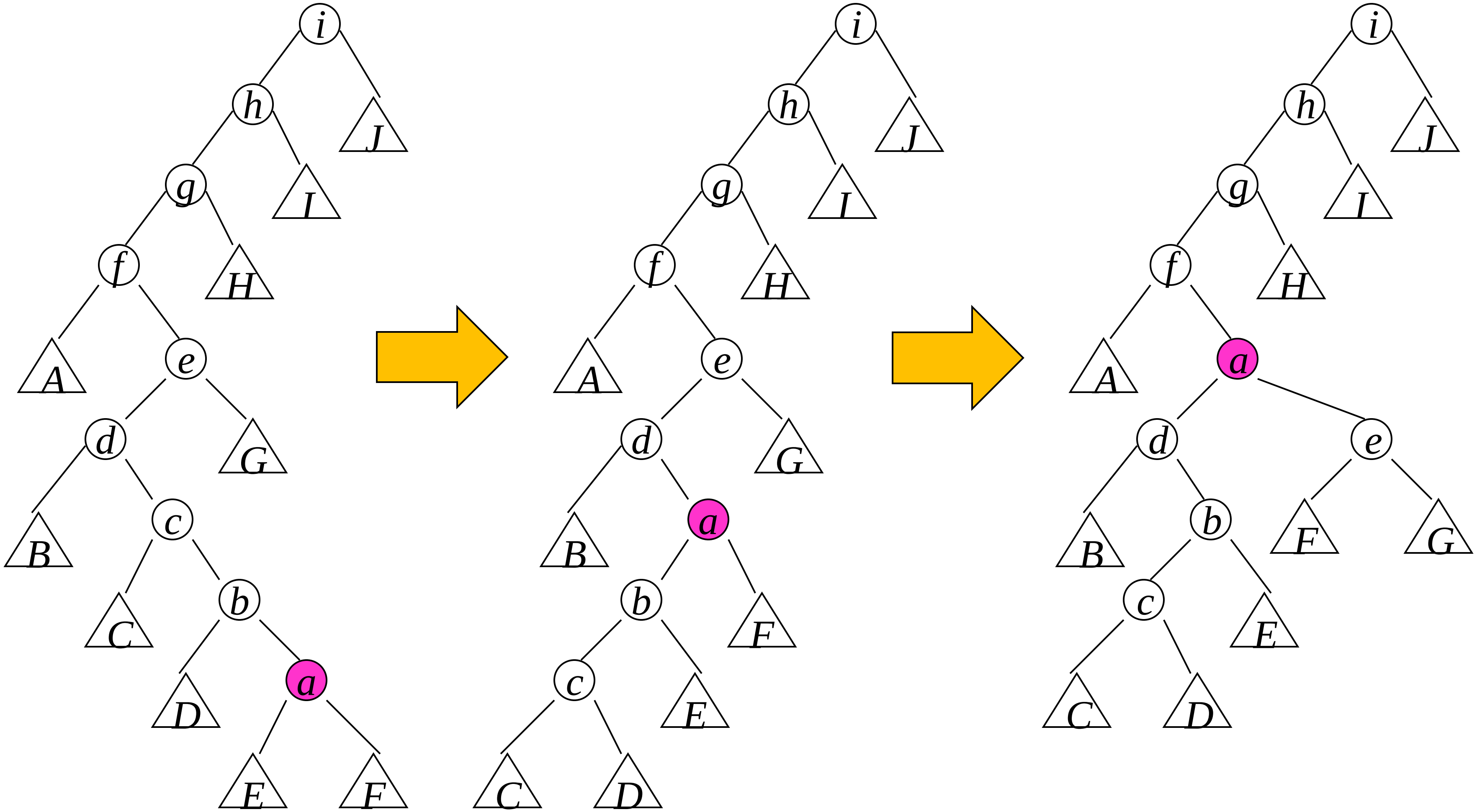

Rotate *y*-*z* left, then rotate *x*-*y* left

# Zig-Zag

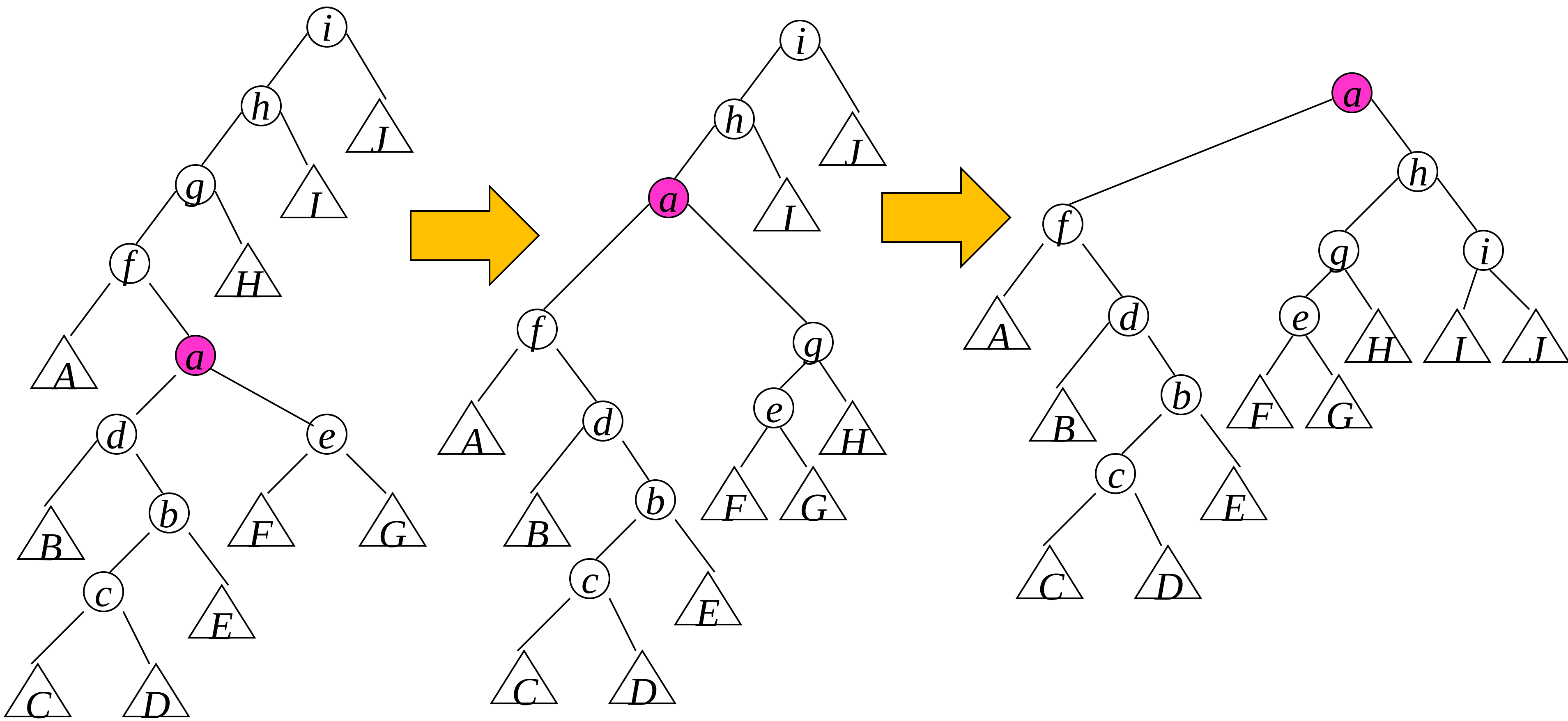



Rotate *x*-*y* right, then rotate *x*-*z* left

# Splaying (example)



# Splaying (example cont)



# Splay Trees (Self-adjusting trees)

[Sleator-Tarjan (1983)]

Amortized cost of each operation is $O(\log n)$

Total cost of $n$ operation is $O(n \log n)$

Many other amazing properties

Some intriguing open problems

Play with them yourself:
http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html