INSTITUT FÜR
INFORMATIK
FREIBURG

Theory I
Algorithm Design and Analysis

(3 - Balanced trees, AVL trees)

Prof. Th. Ottmann

Balanced Trees

A class of binary search trees is balanced, if each of the three dictionary operations

find
insert
delete

of keys for a tree with *n* keys can always (in the worst case) be carried out in *O*(log *n*) steps.

Possible balancing conditions:

height condition →AVL-Bäume
weight condition → BB[α]-Bäume
structural conditions → Bruder-, 2-3-, a-b-, B-Bäume

Goal: Height of a tree with *n* keys is always in *O*(log *n*).

AVL trees

Developed by Adelson-Velskii and Landis (1962)

- Search, insertion and deletion of a key in a randomly created standard search tree with n keys can be done, on average, in $O(\log_2 n)$ steps.
- However, the worst case complexity is $\Omega(n)$.
- Idea of AVL trees: modified procedures for insertion and deletion, which prevents the tree from degenerating.
- Goal of AVL trees: height is in $O(\log_2 n)$ and search, insertion and deletion can be carried out in logarithmic time.

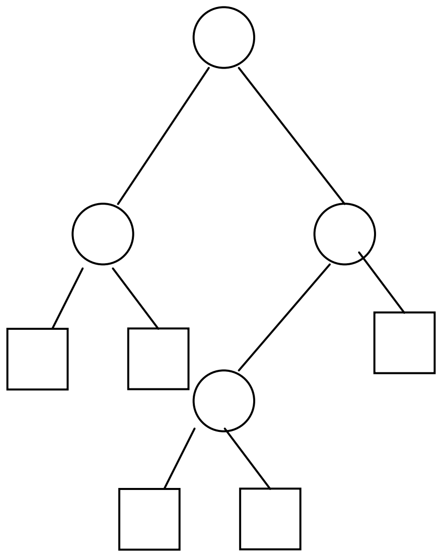
Definition of AVL trees

Definition: A binary search tree is called AVL tree or height-balanced tree, if for each node v the height of the right subtree $h(T_r)$ of v and the height of the left subtree $h(T_l)$ of v differ by at most 1.

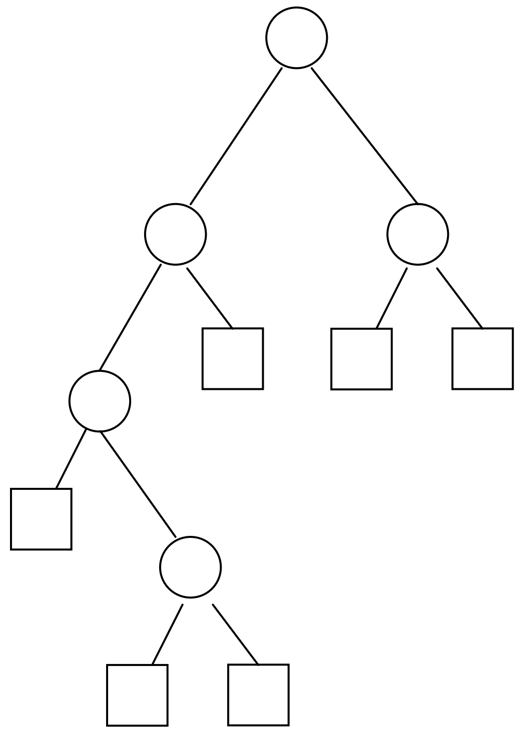
Balance factor:

$$bal(v) = h(T_r) - h(T_l) \in \{-1, 0, +1\}$$

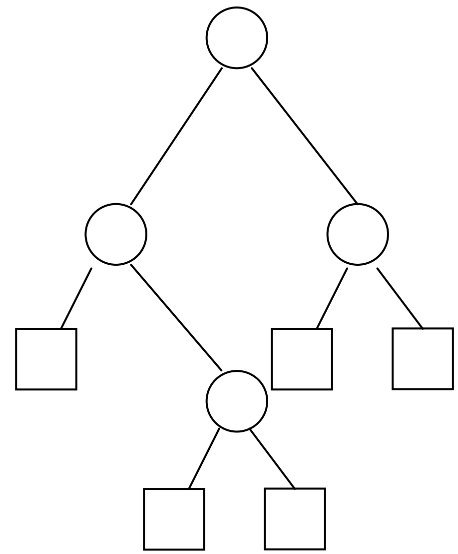
Examples



AVL tree



not an AVL tree



AVL tree

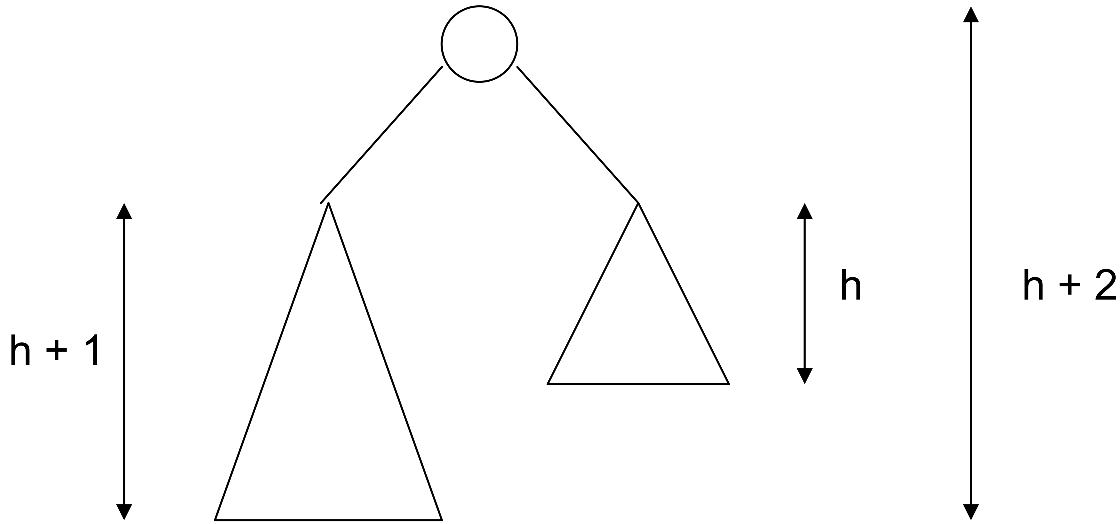
Properties of AVL trees

- AVL trees cannot degenerate into linear lists.
- AVL trees with n nodes have a height in $O(\log n)$.

Apparently:

- An AVL tree of height 0 has 1 leaf
- An AVL tree of height 1 has 2 leaves
- An AVL tree of height 2 with a minimal number of leaves has 3 leaves
- ...
- How many leaves does an AVL tree of height h with minimal number of leaves have?

Minimal number of leaves of AVL trees of height *h*



Hence: An AVL tree of height *h* has at least F_{h+2} leaves, where

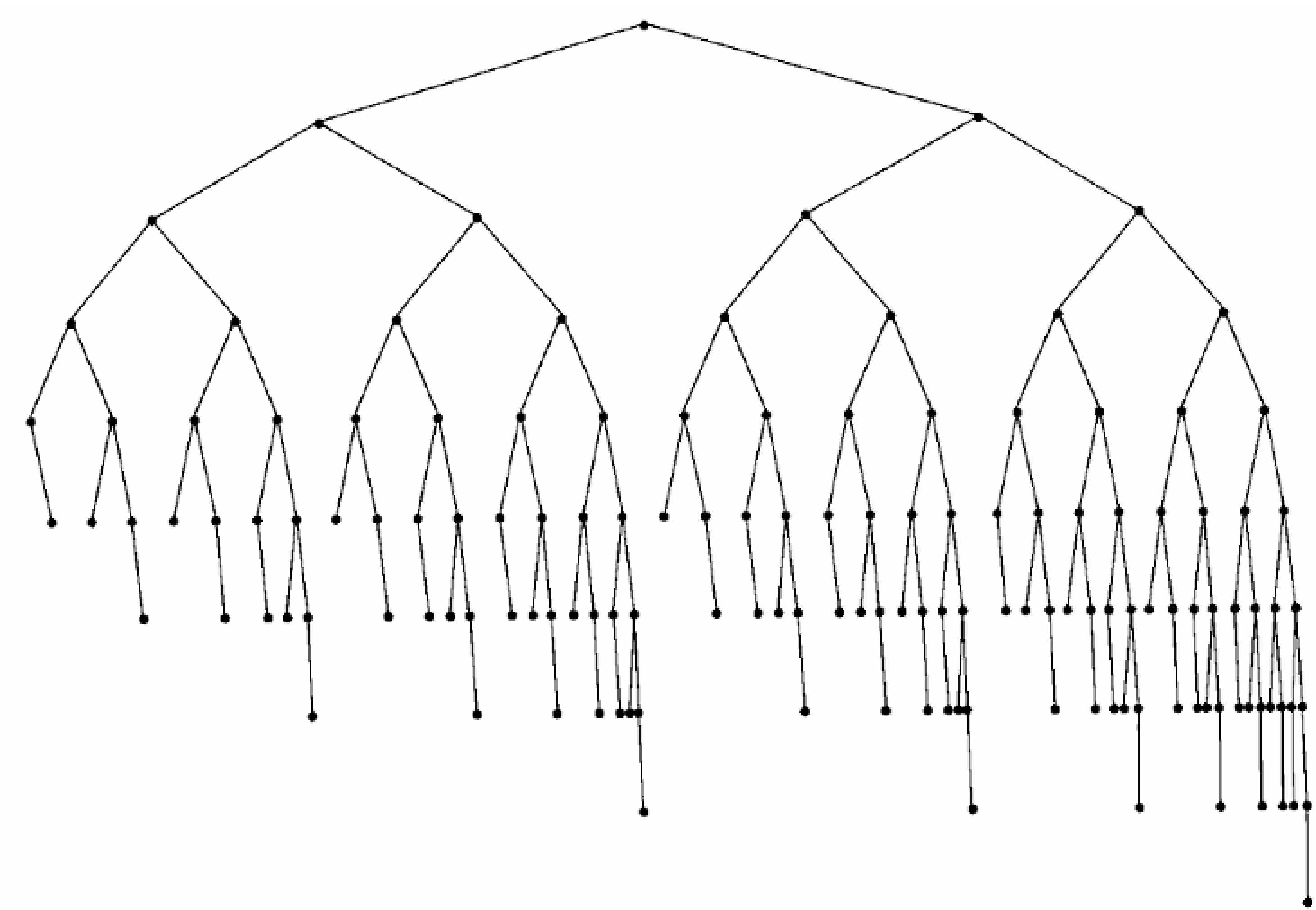
$$F_0 = 0$$

$$F_1 = 1$$

$$F_{i+2} = F_{i+1} + F_i$$

$\leadsto$ F_i is the *i*-th Fibonacci number.

Minimal AVL tree of height 9



Height of an AVL tree

Theorem: The height h of an AVL tree with n leaves (and n-1 internal nodes is at most $c \cdot \log_2 n + 1$, i.e.

$$h \le c \cdot \log_2 n + 1, \text{ with a constant } c.$$

Proof: For the Fibonacci numbers we know

$$F_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+1}\right) \approx 0.7236... * 1.618...^h$$

Since

$$n \ge F_{h+2} \approx 1.894... * 1.618 ...^h$$

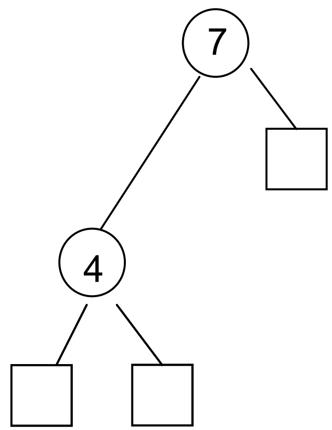
we get

$$h \le \frac{1}{\log_2 1.618...} * \log_2 n - \frac{\log_2 0.7236...}{\log_2 1.618...} \le 1.44.. \log_2 n + 1.$$

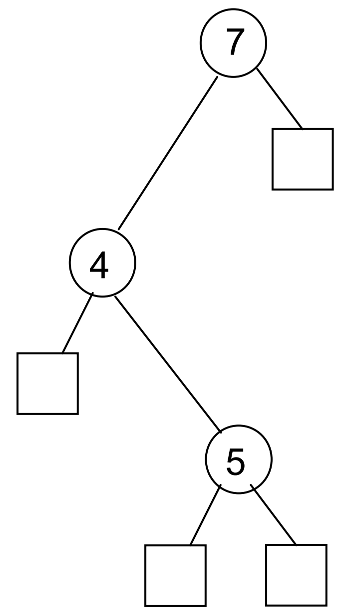
Insertion in an AVL tree

- For each modification of the tree we have to guarantee that the AVL property is maintained.

Original situation:



After inserting key 5:



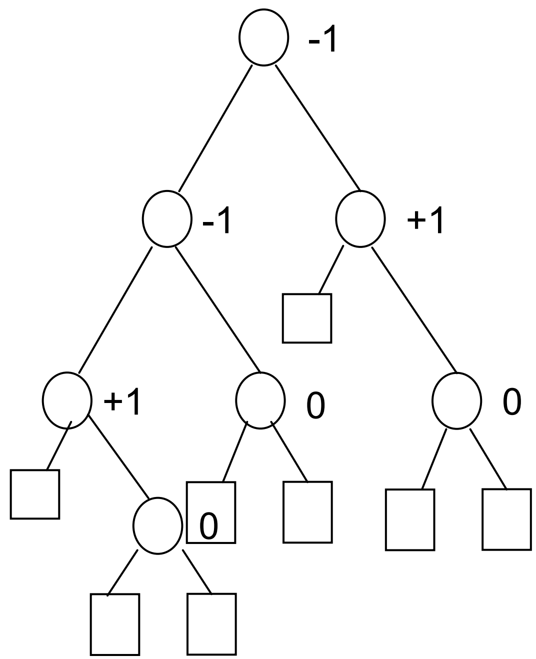
Problem: How can we modify the new tree such that it will be an AVL tree?

Storing the balance factors in the nodes

- In order to restore the AVL property it is sufficient to store, in each node, the balance factor.
- According to the definition

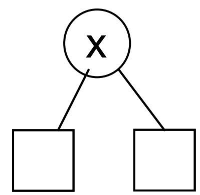
$$bal(p) = h(p.right) - h(p.left) \in \{-1, 0, +1\}$$

Example:



Different situations for insertions in an AVL tree

1. The tree is empty: create a single node with two leaves, store *x* in it. Done!

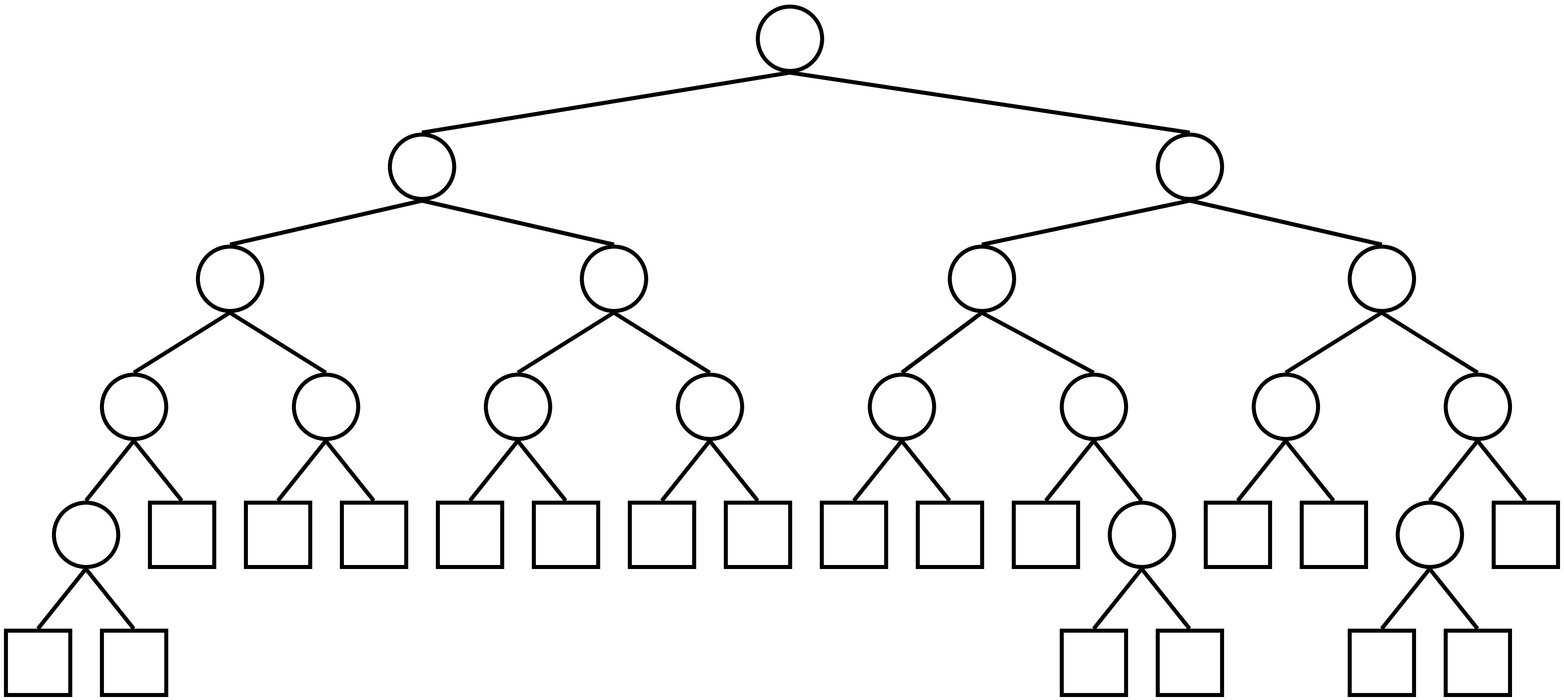


2. The tree is not empty and the search ends in a leaf.

   Let node *p* be the parent of the leaf where the search ended.
   Since bal(p) $\in$ {-1,0,1}, we know that either

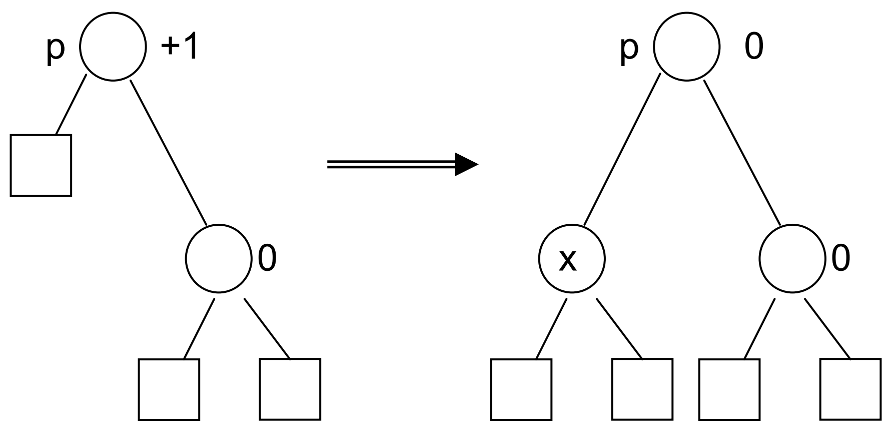
- the left child of *p* is a leaf, but not the right one (case 1) or
- the right child of *p* is a leaf, but not the left one (case 2) or
- both children of *p* are leaves (case 3).

Example of an AVL tree



Overall height unchanged (1)

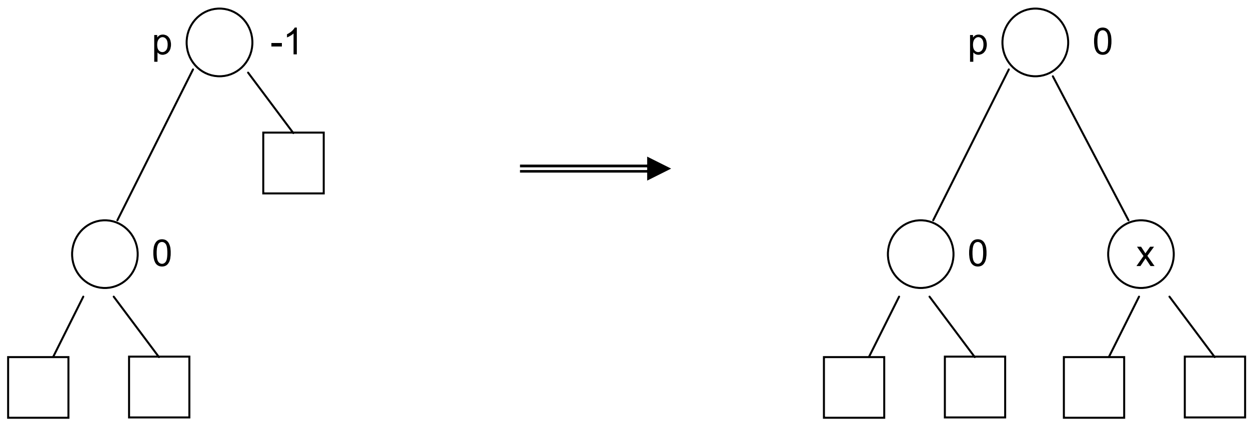
Case 1: [*bal*(*p*) = +1] and *x* < *p.key*, since the search ends at a leaf with parent *p*.



done!

Overall height unchanged (2)

Case 2: [*bal*(*p*) = -1] and *x* > *p.key*, since the search ends at a leaf with parent *p*.



done!

Both cases are uncritical:

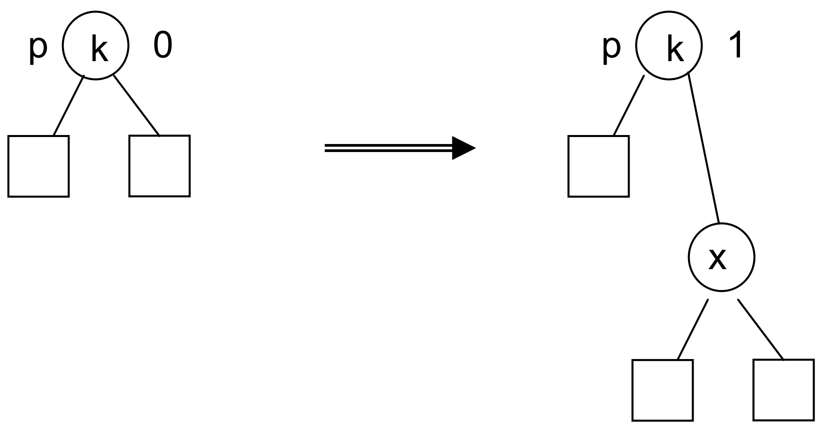
The height of the subtree containing *p* does not change.

The critical case

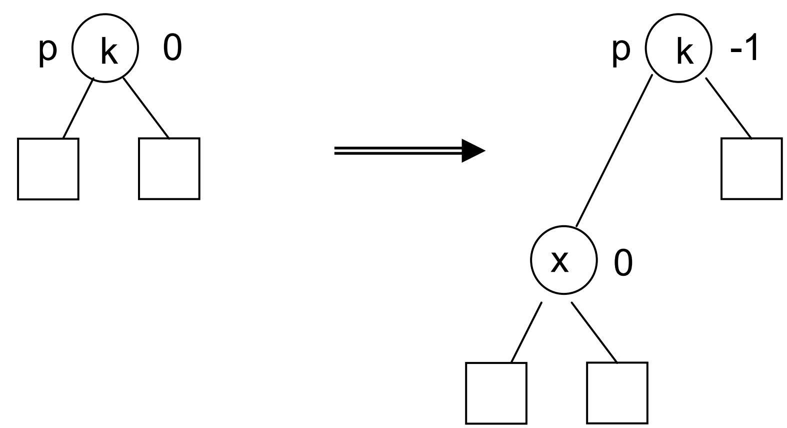
Case 3: [*bal*(*p*) = 0] Then both children of *p* are leaves. The height increases!

We distinguish the cases whether the new key x must be inserted as the right or left child of p:

[*bal(p) =0 and x > p.key*]



[*bal(p) =0 and x < p.key*]



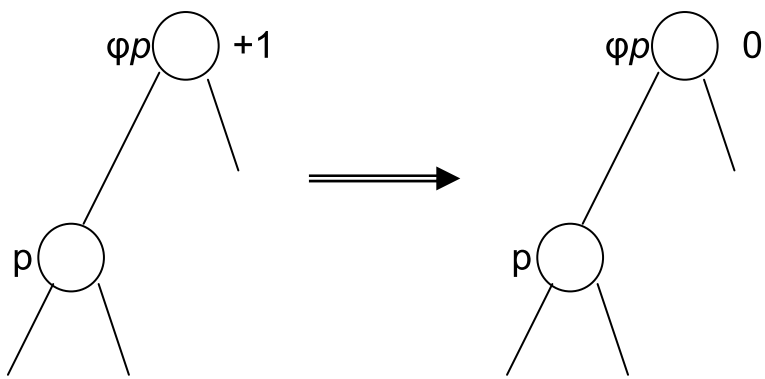
- In both cases we need a procedure *upin*(*p*) which traces back the search path, checks the balance factors and carries out restructuring operations (so-called rotations or double rotations).

The procedure *upin*(p)

- When *upin*(p) is called, we always have *bal*(p) $\in$ {-1, +1} and the height of the subtree rooted in p has increased by 1.
- *upin*(p) starts at p and goes upwards stepwise (until the root if necessary).
- In each step it tries to restore the AVL property.
- In the following we concentrate on the situation where p is the left child of its parent φp.
- The situation where p is the right child of its parent φp is handled similarly.

Case 1: $bal(\varphi p) = 1$

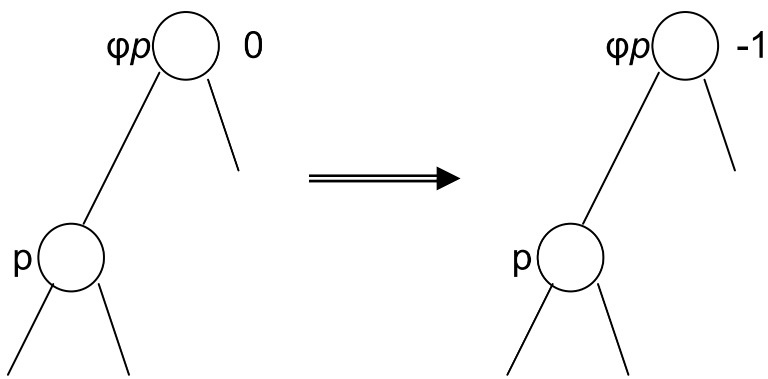
1. The parent φp has balance factor +1. Since the height of the subtree rooted in p (the left child of φp) has increased by 1, it is sufficient to set the balance factor of φp to 0:



done!

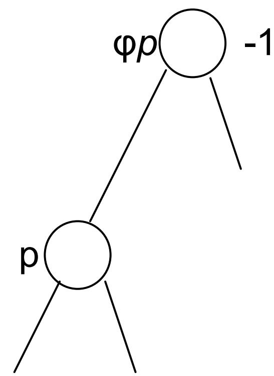
Case 2: *bal*(φ*p*) = 0

2. The parent φ*p* has balance factor 0. Since the height of the subtree rooted in *p* (the left child of φ*p*) has increased by 1, the balance factor of φ*p* changes to -1. Since the height of the subtree rooted in φ*p* has also changed, we must call *upin* recursively with φ*p* as the argument.



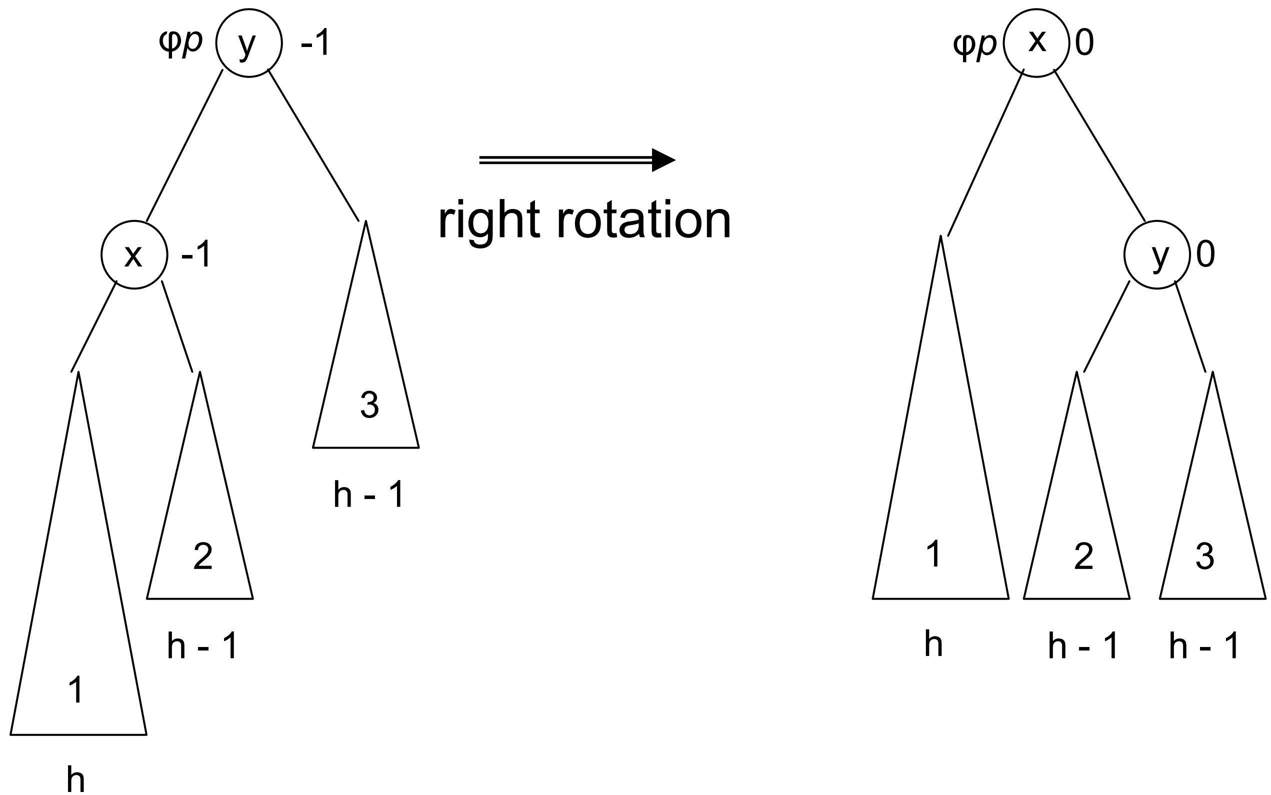
upin(φ*p*)

The critical case 3: *bal*(φ*p*) = -1



- If *bal*(φ*p*) = -1 and the height of the left subtree of φ*p* (rooted in *p*) has increased by 1, the AVL property is now violated in φ*p*.
- In this case we have to restructure the tree.
- Again we distinguish two cases: *bal*(*p*) = -1 (case 3.1) and *bal*(*p*) = +1 (case 3.2).
- The invariant for the call of *upin(p)* is *bal*(*p*) ≠ 0. The case *bal*(*p*) = 0 can therefore not occur!

Case 3.1: *bal*(φ*p*) = -1 and *bal*(*p*) = -1



done!

Is the resulting tree still a search tree?

We must guarantee that the resulting tree fulfils the

1. search tree condition and the
2. AVL property.

Search tree condition: Since the original tree was a search tree, we know that

all keys in tree 1 are smaller than x.

all keys in tree 2 are greater than x and smaller then y.

all keys in tree 3 are greater than y (and x).

Hence, the resulting tree also fulfils the search tree condition.

Is the resulting tree balanced?

AVL property: Since the original tree was an AVL tree, we know:

- since *bal*(φ*p*) = -1, tree 2 and tree 3 have the same height *h*-1.
- since *bal*(*p*) = -1 after the insertion, tree 1 has height *h*, while tree 2 has height *h*-1.

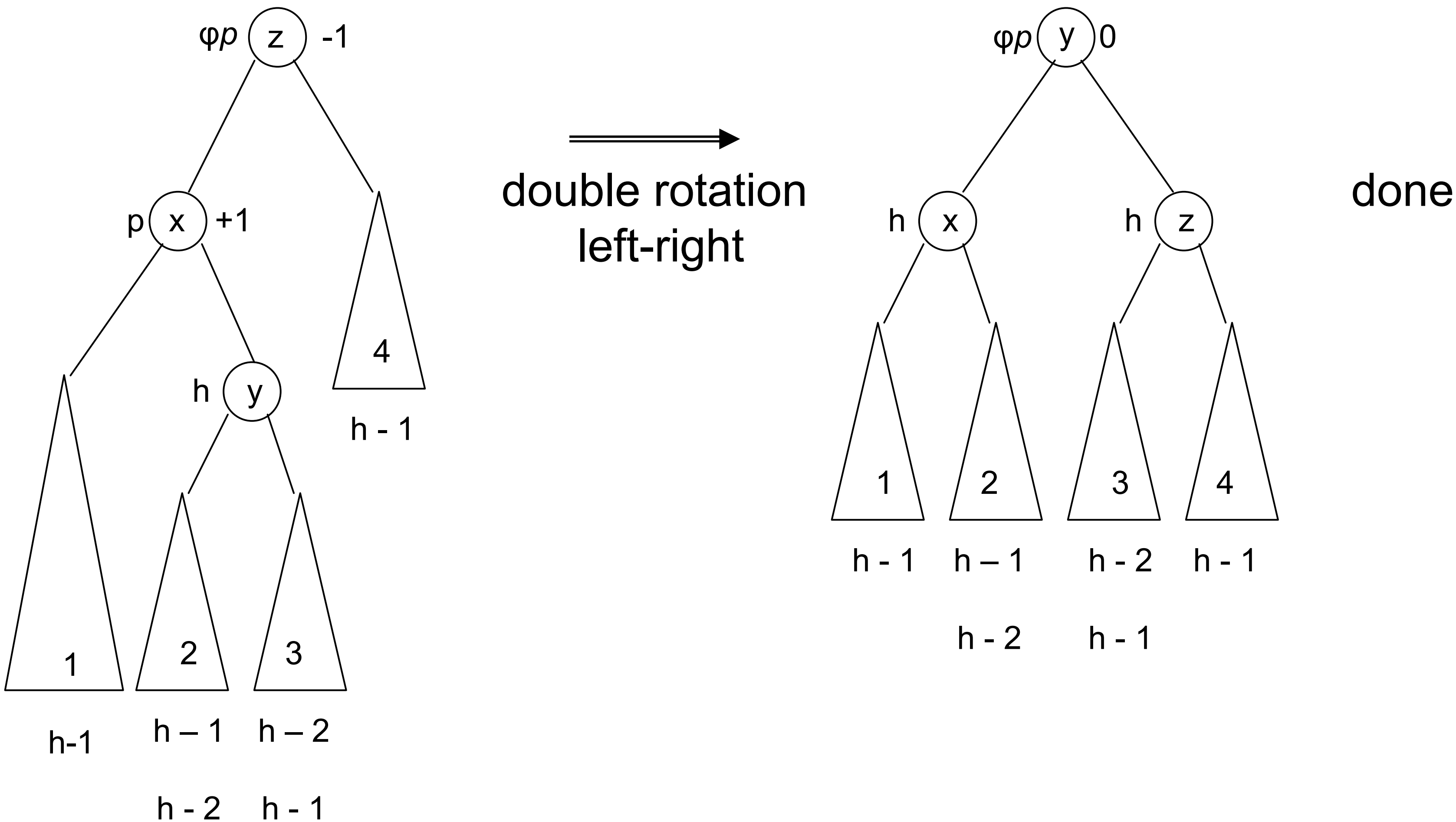
  Hence, after the rotation:

- The node containing *y* has balance factor 0.
- Node φ*p* has balance factor 0.

Thus, the AVL property has been restored.

Case 3.2: *bal*(φ*p*) = -1 and *bal*(*p*) = +1

Case 3.2: *bal*(φ*p*) = -1 and *bal*(*p*) = +1



Properties of the subtrees

1. The new key must have been inserted into the right subtree of *p*.
2. Trees 2 and 3 must have different height, since otherwise the method *upin* would not have been called.
3. The only possible combination of heights in trees 2 and 3 is therefore (h-1, h-2) and (h-2, h-1), unless they are empty.
4. Since *bal*(*p*) = 1, tree 1 must have height h-1
5. Finally, tree 4 also must have height h-1 (because *bal*(φ*p*) = -1).

Hence, the resulting tree also fulfils the AVL property.

Search tree condition

We have:

1. All keys in tree 1 are smaller than x.
2. All keys in tree 2 are smaller than y but greater than x.
3. All keys in tree 3 are greater than y and x but smaller than z.
4. All keys in tree 4 are greater than x, y and z.

Hence, the tree resulting from the double rotation is also a search tree.

Remarks

- We have only considered the case where *p* is the left child of its parent φ*p*.
- The case where *p* is the right child of its parent φ*p* is handled similarly.
- For an efficient implementation of the method *upin*(*p*), we have to create a list of all visited nodes during the search for the insert position.
- Then we can use this list during the recursive calls to proceed to the parent and carry out the necessary rotations or double rotations.

Insertion in a non-empty AVL tree

Search for x ends in a leaf with parent p

1. Right child of p not a leaf, $x < p.key$ → Append as left child of p, done.
2. Left child of p not a leaf, $x > p.key$ → append as right child of p, done.
3. Both children of p are leaves: append x as child of p and call *upin*(p).

The method *upin*(p):

1. p is left child of φp

   (a) $bal(\varphi p) = +1$ → $bal(\varphi p) = 0$, done.

   (b) $bal(\varphi p) = 0$ → $bal(\varphi p) = -1$, *upin*(φp)

   (c) i. $bal(\varphi p) = -1$ und $bal(p) = -1$ right rotation, done.

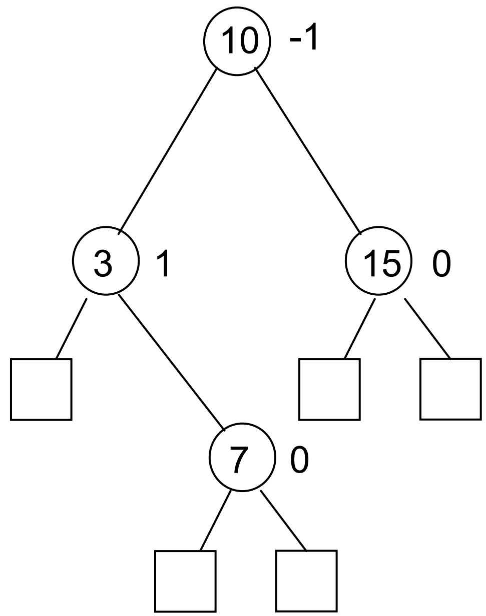
   ii. $bal(\varphi p) = -1$ und $bal(p) = +1$ double rotation left-right, done.

2. p is righter child of φp.

   ...

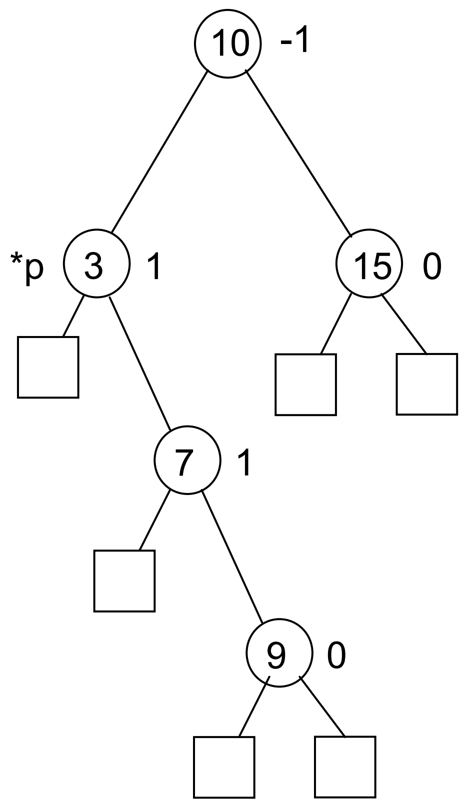
An example (1)

Original situation:



An example (2)

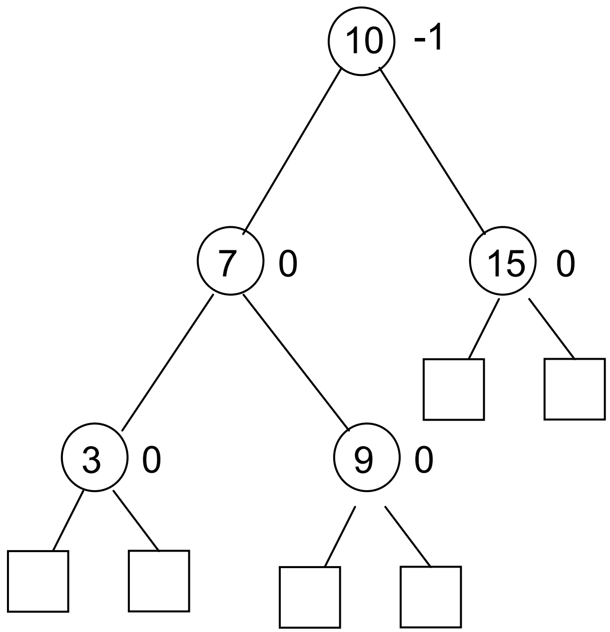
Insert key 9:



AVL property is violated!

An example (3)

Left rotation at *p yields:



An example (4)

Insertion of 8 followed by double rotation (left-right):