Datastructuren

Data Structures

Hendrik Jan Hoogeboom
Mark van den Bergh

Informatica – LIACS
Universiteit Leiden

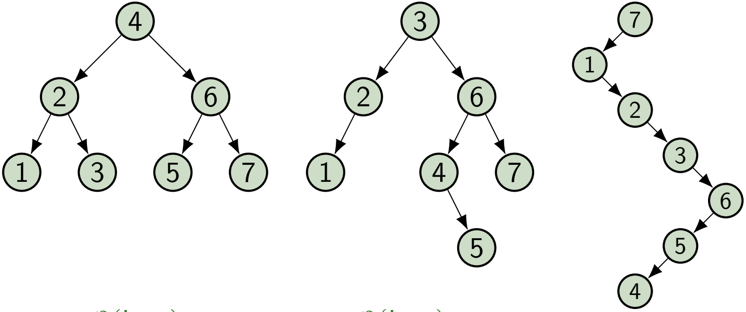
najaar 2024

Table of Contents I

Contents

binary trees

accessing average node



$\mathcal{O}(\lg n)$

$\mathcal{O}(\lg n)$
average tree

$\mathcal{O}(n)$

STL container classes

helper: `pair`

sequences: *contiguous*: `array` (fixed length),
`vector` (flexible length),
`deque` (double ended),
linked: `forward_list` (single), `list` (double)

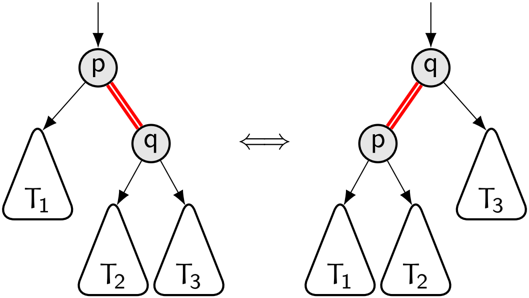
adaptors: *based on one of the sequences*:
`stack` (LIFO), `queue` (FIFO),
based on binary heap: `priority_queue`

associative: *based on balanced trees*:
`set`, `map`, `multiset`, `multimap`

unordered: *based on hash table*:
`unordered_set`, `unordered_map`,
`unordered_multiset`,
`unordered_multimap`

Contents

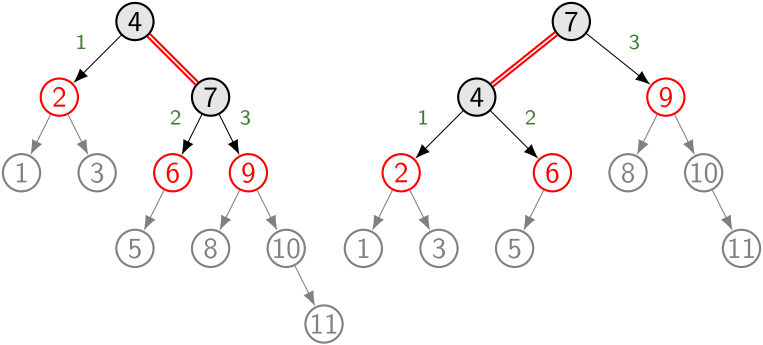
single rotation



$$T_1 \; \textcircled{p} \; (\; T_2 \; \textcircled{q} \; T_3 \;) = (\; T_1 \; \textcircled{p} \; T_2 \;) \; \textcircled{q} \; T_3$$

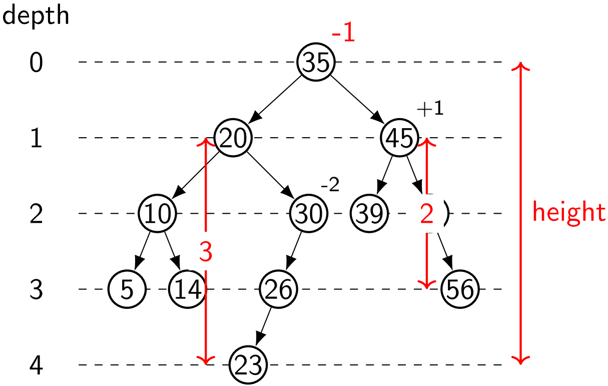
note: implementation needs parent (for pointer to root p vs q)

example



Contents

balance factor



AVL trees

Features:

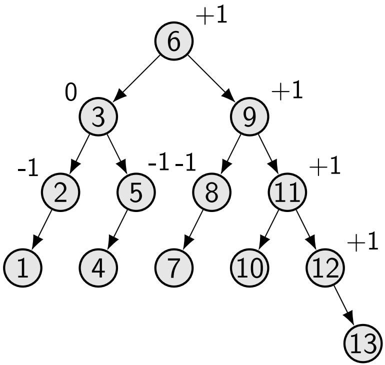
- height balanced binary search tree, *logarithmic height* and logarithmic search time.
- rebalancing after inserting a key using (at most) *one single/double rotation* at the *lowest unbalanced node* on the search path to the new key.
- rebalancing after deletion might need a rotation at every level of the search path (bottom-up).

Definition

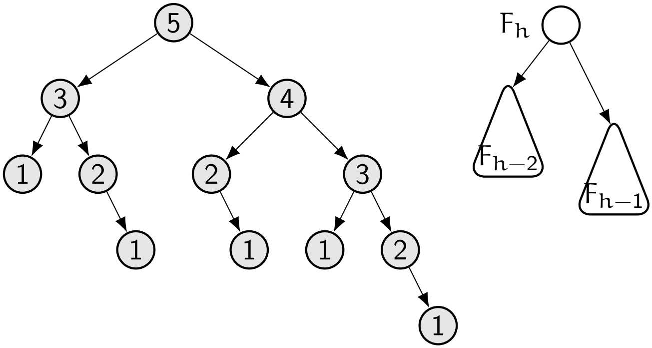
An *AVL-tree* is a *binary search tree* in which for each node the *heights of both its subtrees differ by at most one*.
The difference in height at a node in a binary tree (right minus left) is called the *balance factor* of that node.

- BST
- balance {-1,0,+1}   each node

example

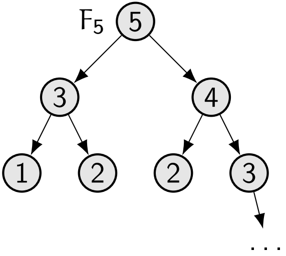


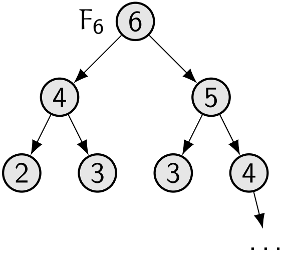
Fibonacci tree 'worst' AVL tree



aantal knopen: $f_h = f_{h-2} + f_{h-1} + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^h$

Fibonacci tree easy complexity



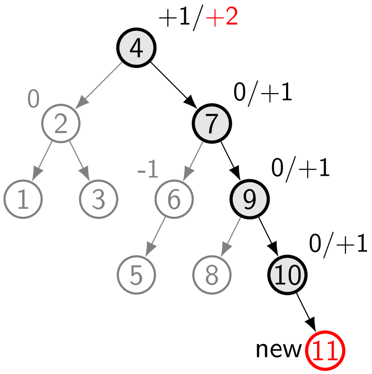


aantal knopen $F_{2k} : n \geqslant 2^{k/2}$

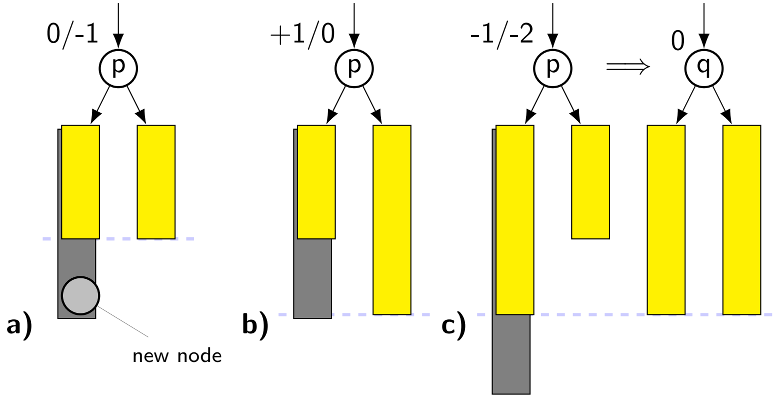
complexiteit : $k \leqslant 2 \lg n$

Contents

adding a key



adding in left subtree, bottom-up view

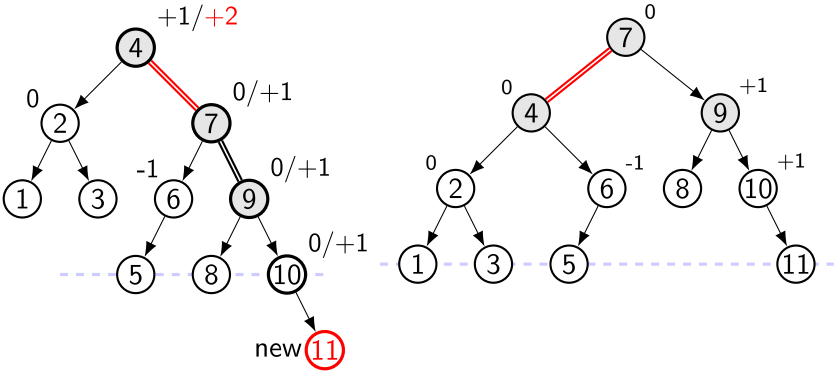


ok, go up

ok, STOP

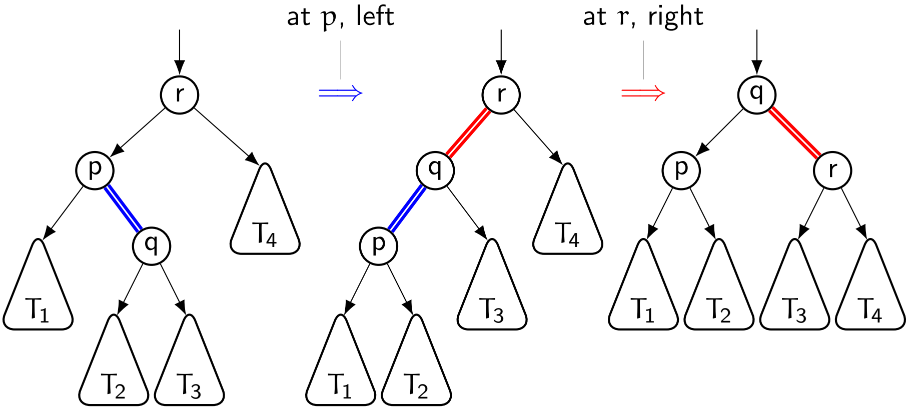
rebalance, STOP

example: adding 11

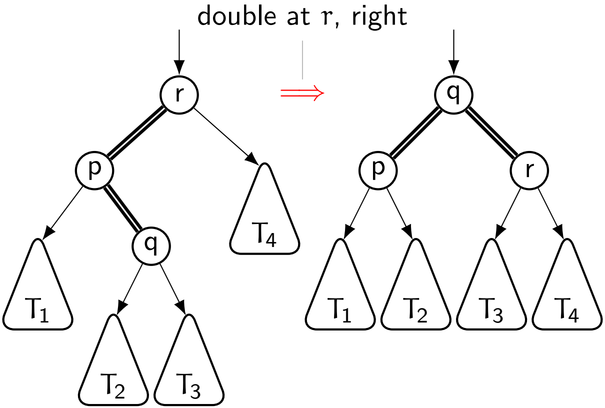


inbalance at 4, RR-case, single rotation at 4, left

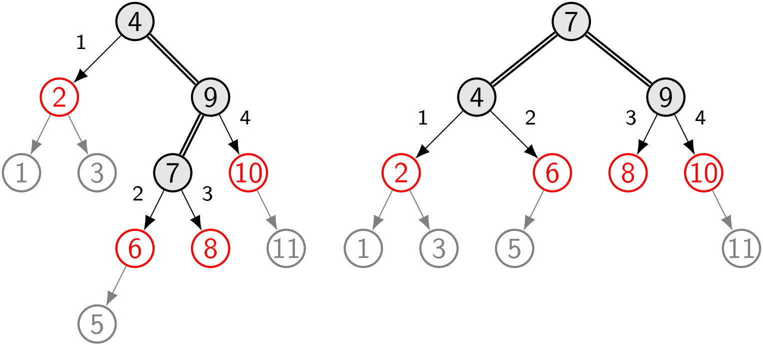
double rotation



double rotation (in one step)



example



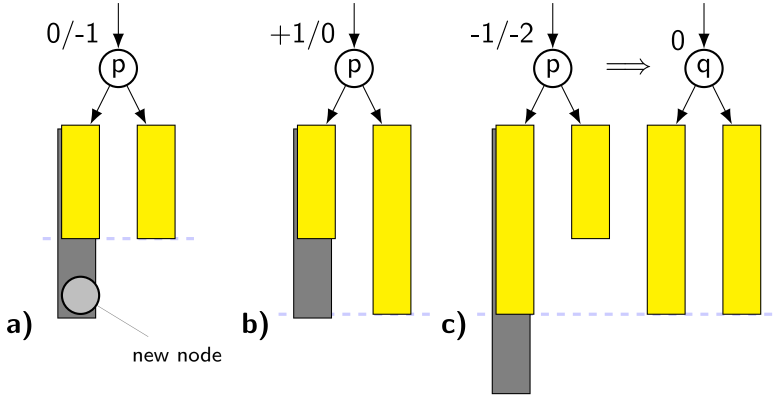
rebalance

bottom up

1. $0 \mapsto \pm 1$ (go up)
2. $\pm 1 \mapsto 0$ (done)
3. $\pm 1 \mapsto \pm 2$ (lowest position of unbalance)
   - LL RR single rotation
   - LR RL double rotation

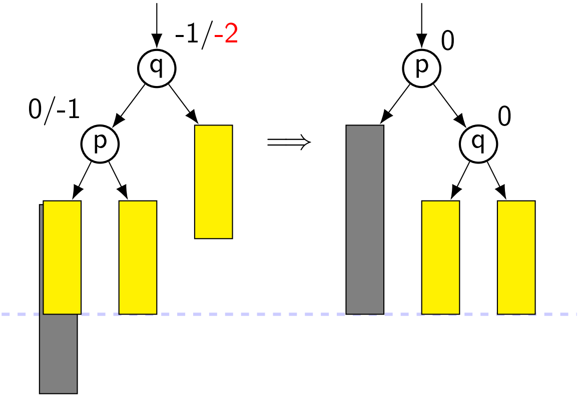
   (then done)

adding in left subtree, bottom-up view

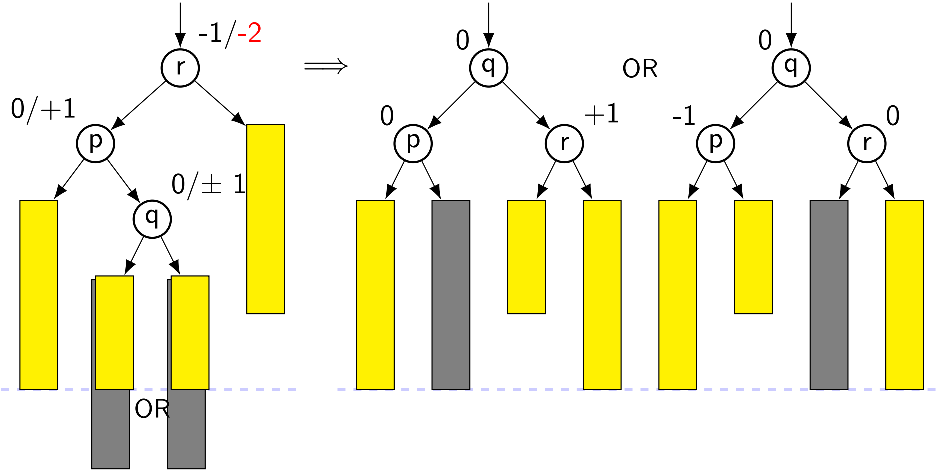


ok, go up　　ok, STOP　　rebalance, STOP

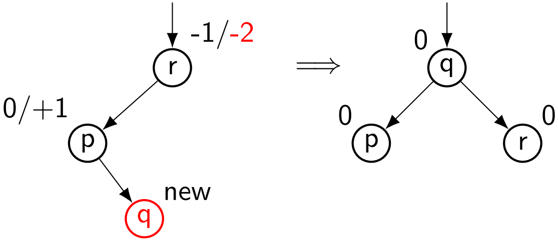
rebalance LL-case



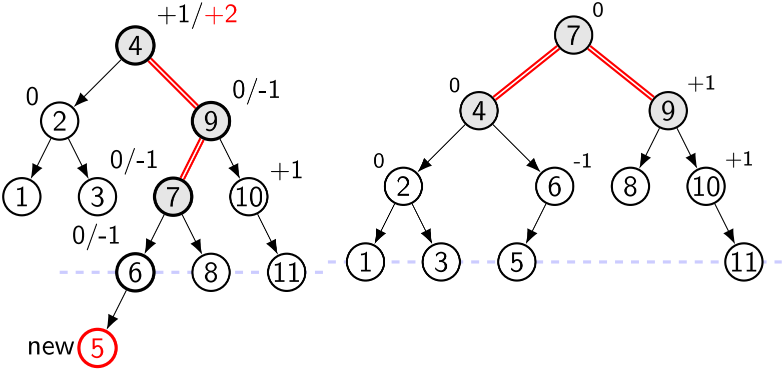
rebalance LR-cases



special LR-case

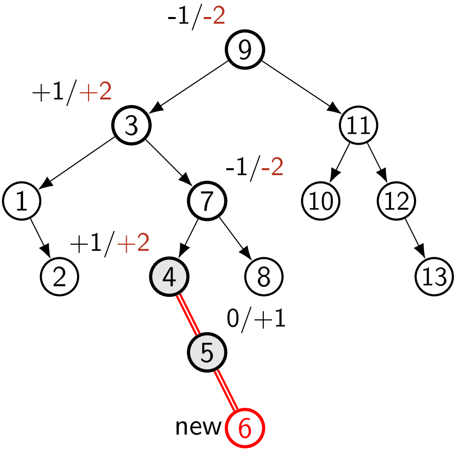


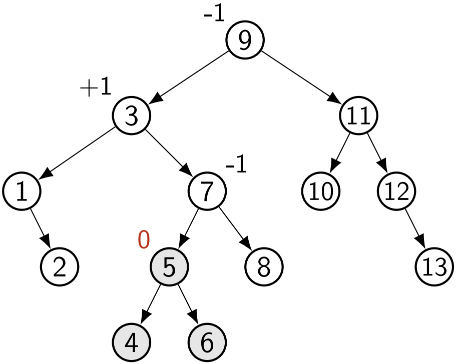
example: adding 5



inbalance at 4, RL-case, double rotation at 4, left

example: adding 6

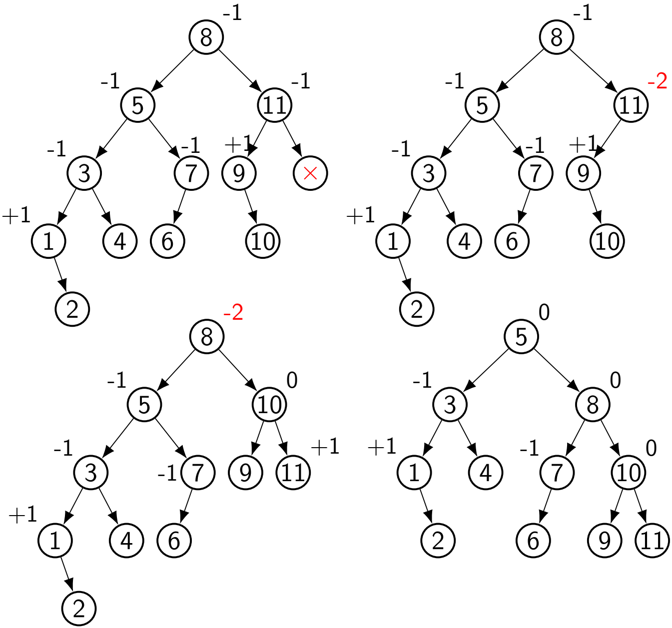




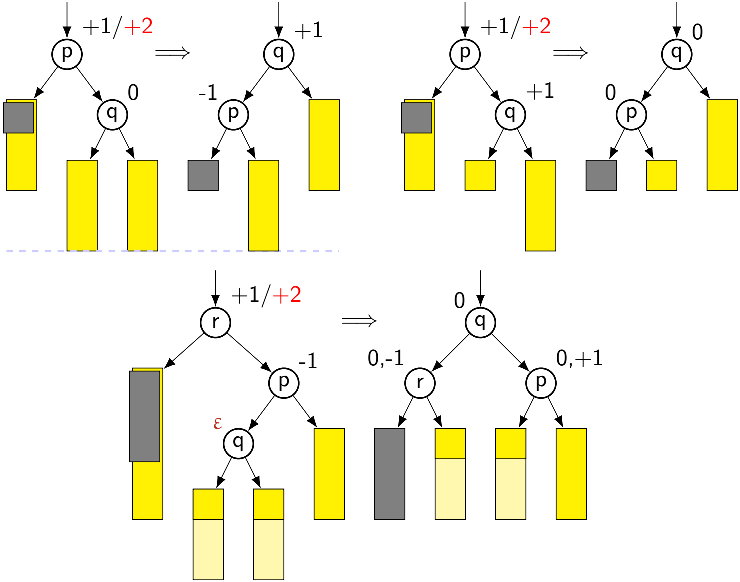
lowest inbalance at 4, RR-case, single rotation at 4, left

Contents

deletion (cascade)



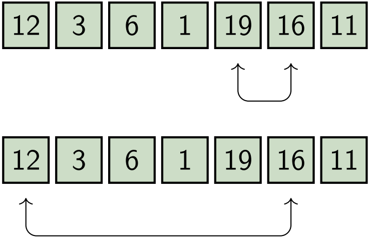
deletion (cases)

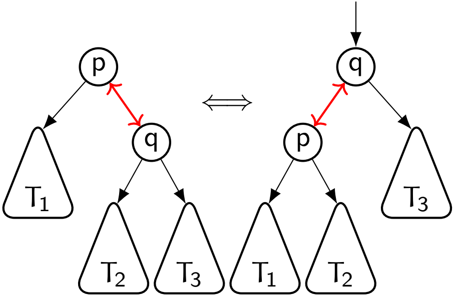


Contents

move to front heuristics

unordered list: often-searched items move to front for faster access

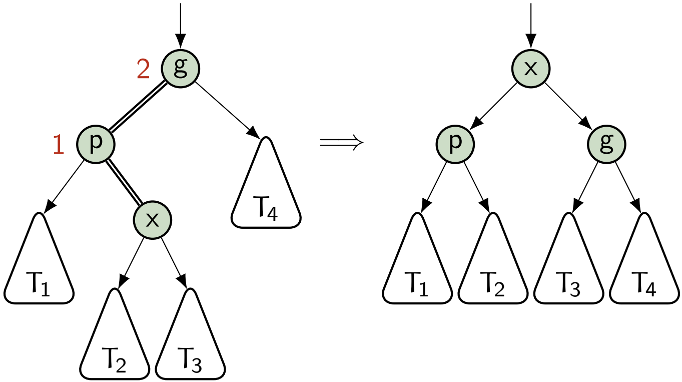




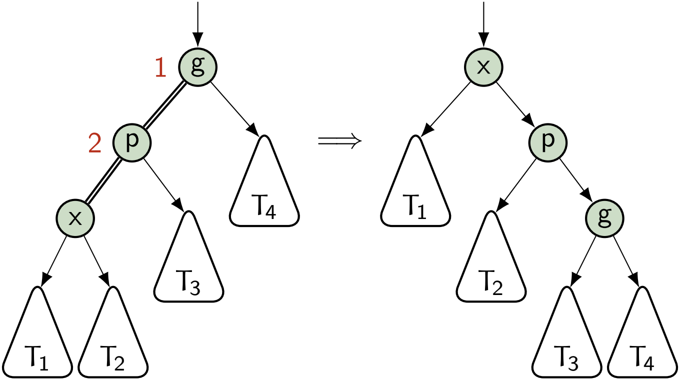
splay trees

- Simple implementation, no bookkeeping.
  self organizing
- Any sequence of K operations (insert, find) has an *amortized* complexity of $\mathcal{O}(K \log n)$
- move item to root *two levels* at a time
- zig-zig step differs from bottom-up rotation

splay zig-zag

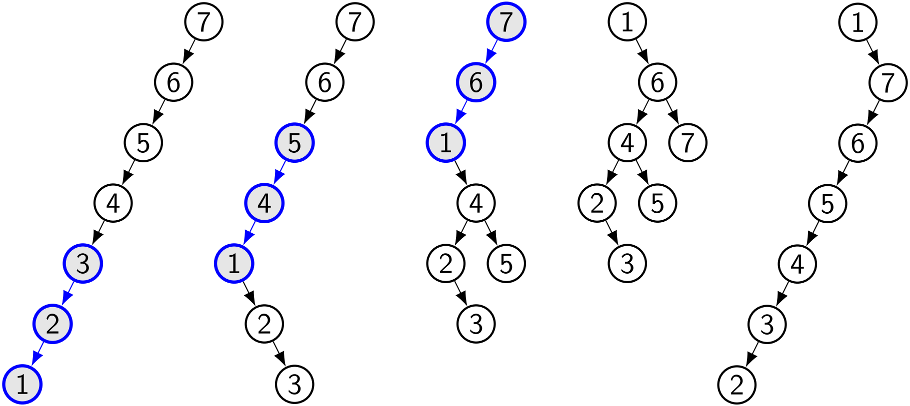


splay zig-zig



different order than bottom-up rotations

example splay linear tree



end.