Informatik II

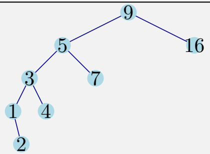
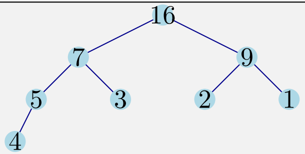
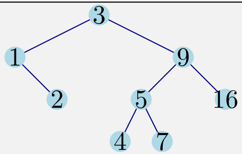
Übung 7

FS 2019

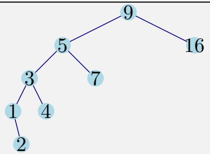
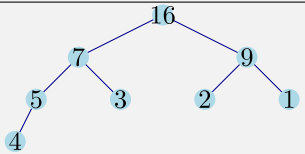
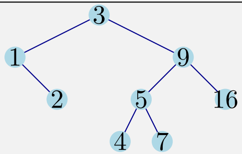
Program Today

1. Recap Binary Trees
2. Repetition Lectures
   - AVL Condition
   - AVL Insert
3. In-Class-Exercises

Comparison of binary Trees

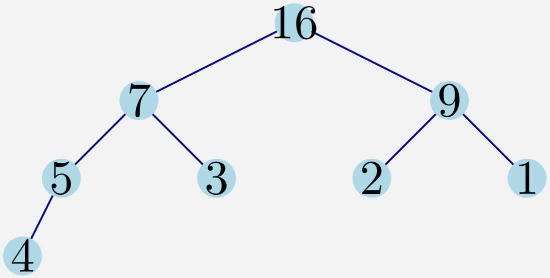
	Search trees	**Heaps** Min- / Max- Heap	**Balanced trees** AVL, red-black tree
in Java:		PriorityQueue	TreeSet
			
Insertion	$\mathcal{O}(h(T))$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Search	$\mathcal{O}(h(T))$	$\mathcal{O}(n)$ (!!)	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(h(T))$	Search + $\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Comparison of binary Trees

	Search trees	Heaps Min- / Max- Heap	Balanced trees AVL, red-black tree
in Java:		PriorityQueue	TreeSet
			
Insertion	$\mathcal{O}(h(T))$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Search	$\mathcal{O}(h(T))$	$\mathcal{O}(n)$ (!!)	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(h(T))$	Search + $\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Recall: $\mathcal{O}(\log n) \leq \mathcal{O}(h(T)) \leq \mathcal{O}(n)$

Recall: Pre- / In- / Post- Order

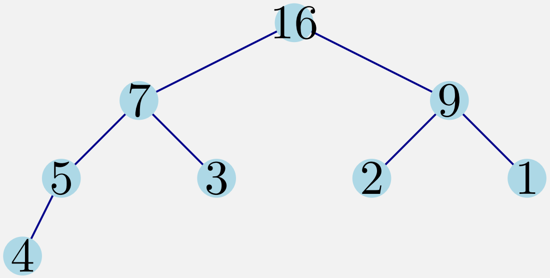


Pre-order:

In-order:

Post-order:

Recall: Pre- / In- / Post- Order

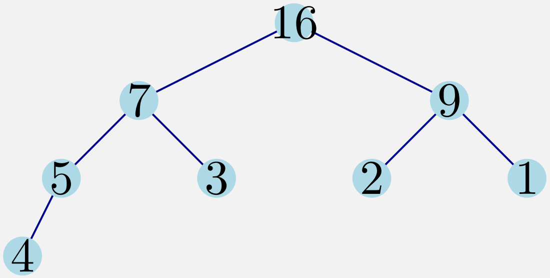


Pre-order: 16 7 5 4 3 9 2 1

In-order:

Post-order:

Recall: Pre- / In- / Post- Order

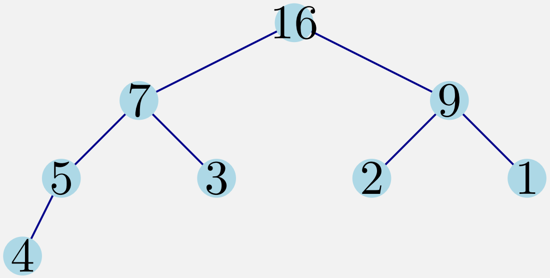


Pre-order: 16 7 5 4 3 9 2 1

In-order: 4 5 7 3 16 2 9 1

Post-order:

Recall: Pre- / In- / Post- Order



Pre-order: 16 7 5 4 3 9 2 1

In-order: 4 5 7 3 16 2 9 1

Post-order: 4 5 3 7 2 1 9 16

Repetition: Binary Trees, Inserting a Key

Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).

MinHeap

- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).

Repetition: Binary Trees, Inserting a Key

Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).

MinHeap

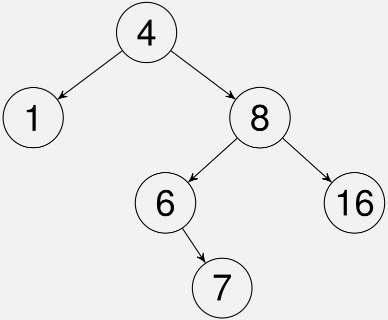
- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).

Exercise: Insert $4, 8, 16, 1, 6, 7$ into empty Tree/Heap.

Repetition: Binary Trees, Inserting a Key

Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).



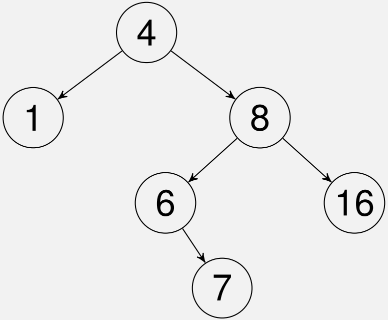
MinHeap

- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).

Repetition: Binary Trees, Inserting a Key

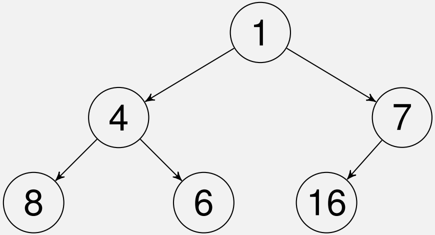
Binary Search Trees

- Search for Key.
- Insert at the reached empty leaf (`null`).



MinHeap

- Insert at the very back of the Array.
- Restore Heap-Condition: `siftUp` (climb successively).



Repetition: Binary Trees, Deleting a Key

Binary Search Trees

- Replace key k by symmetric successor n.
- Careful: What about right child of n?

MinHeap

- Replace key by last element of the array.
- Restore Heap-Condition: `siftDown` *or* `siftUp`.

Repetition: Binary Trees, Deleting a Key

Binary Search Trees

- Replace key k by symmetric successor n.
- Careful: What about right child of n?

MinHeap

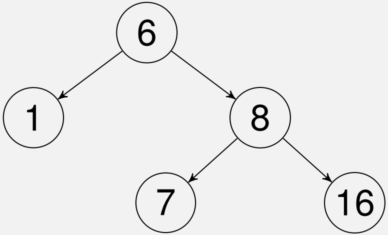
- Replace key by last element of the array.
- Restore Heap-Condition: `siftDown` *or* `siftUp`.

Exercise: Delete 4 from Example Tree/Heap.

Repetition: Binary Trees, Deleting a Key

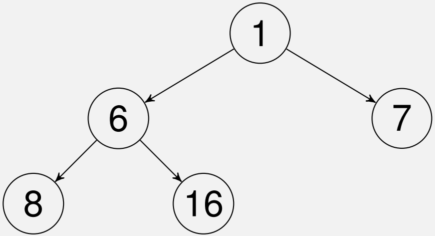
Binary Search Trees

- Replace key k by symmetric successor n.
- Careful: What about right child of n?



MinHeap

- Replace key by last element of the array.
- Restore Heap-Condition: `siftDown` *or* `siftUp`.



Java: Delete from MinHeap

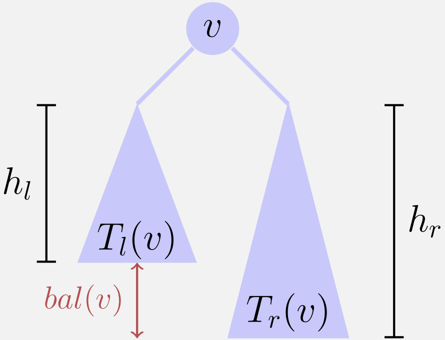
Problem! How to find a key in a MinHeap?

$\Rightarrow$ We usually only take care of root deletions (Extract-Min).

Balance of a node

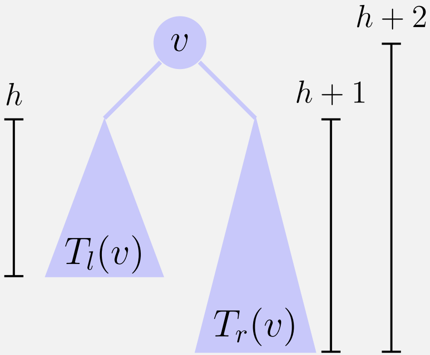
The height *balance* of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

$$\mathrm{bal}(v) := h(T_r(v)) - h(T_l(v))$$

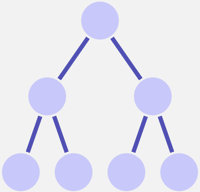


AVL Condition

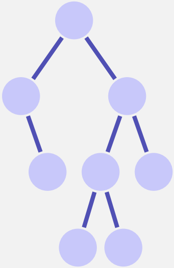
AVL Condition: for eacn node v *of a tree* $\mathrm{bal}(v) \in \{-1, 0, 1\}$



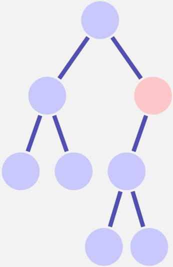
(Counter-)Examples



AVL tree with height 2



AVL tree with height 3



No AVL tree

Insertion

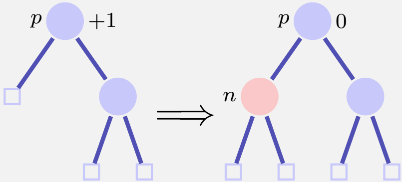
Balance

- Keep the balance stored in each node
- Re-balance the tree in each update-operation

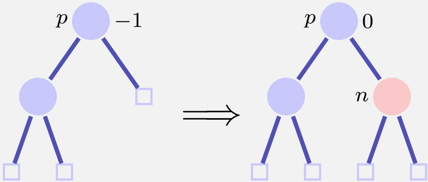
New node n is inserted:

- Insert the node as for a search tree.
- Check the balance condition increasing from n to the root.

Balance at Insertion Point



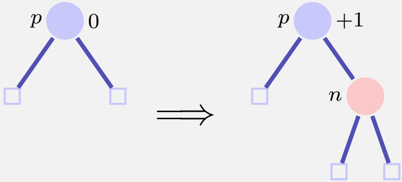
case 1: $\mathrm{bal}(p) = +1$



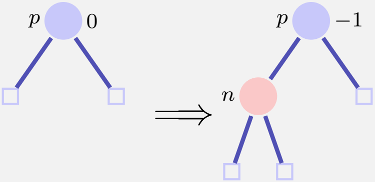
case 2: $\mathrm{bal}(p) = -1$

Finished in both cases because the subtree height did not change

Balance at Insertion Point



case 3.1: $\mathrm{bal}(p) = 0$ right



case 3.2: $\mathrm{bal}(p) = 0$, left

Not finished in both case. Call of `upin(p)`

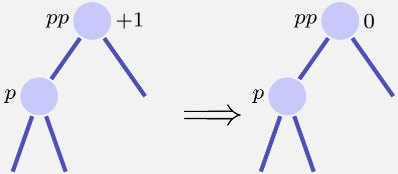
upin(p) - invariant

When `upin(p)` is called it holds that

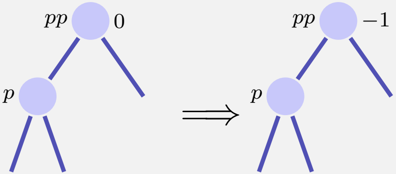
- the subtree from p is grown and
- $\mathrm{bal}(p) \in \{-1, +1\}$

upin(p)

Assumption: p is left son of pp[1]



case 1: $\mathrm{bal}(pp) = +1$, done.



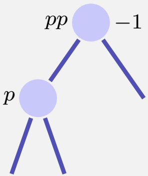
case 2: $\mathrm{bal}(pp) = 0$, `upin(pp)`

In both cases the AVL-Condition holds for the subtree from pp

[1] If p is a right son: symmetric cases with exchange of $+1$ and -1

upin(p)

Assumption: p is left son of pp



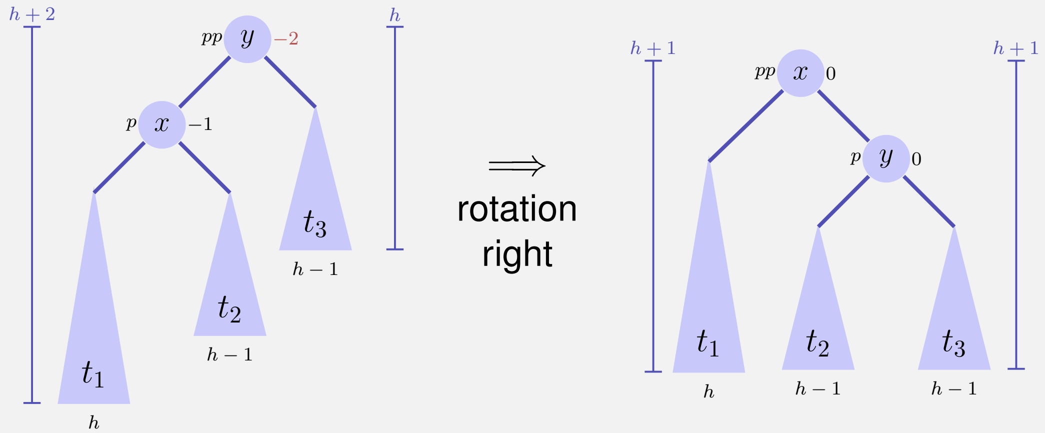
case 3: $\mathrm{bal}(pp) = -1$,

This case is problematic: adding n to the subtree from pp has violated the AVL-condition. Re-balance!

Two cases $\mathrm{bal}(p) = -1$, $\mathrm{bal}(p) = +1$

Rotations

case 1.1 $\mathrm{bal}(p) = -1.$ [2]



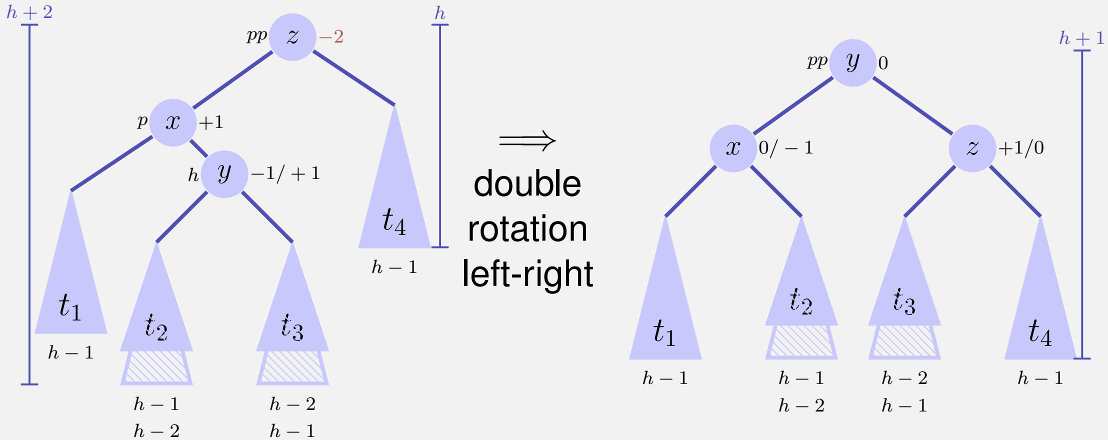
$\Longrightarrow$

rotation right

[2] p right son: $\Rightarrow \mathrm{bal}(pp) = \mathrm{bal}(p) = +1$, left rotation

Rotations

case 1.1 $\mathrm{bal}(p) = -1.$ [3]



[3] p right son $\Rightarrow \mathrm{bal}(pp) = +1$, $\mathrm{bal}(p) = -1$, double rotation right left

Augment a Tree

Exercise:
Augment the nodes `n` of a binary search tree with their heights `n.height`. Make sure the height stays consistent when nodes are inserted.

[Start here:
`https://expert.ethz.ch/print/ifbaug2/SS19/e07_examples`]

Questions / Suggestions?