Technische Universität Braunschweig

Algorithmen und Datenstrukturen – Übung #4

AVL-Bäume, Heaps

Arne Schmidt

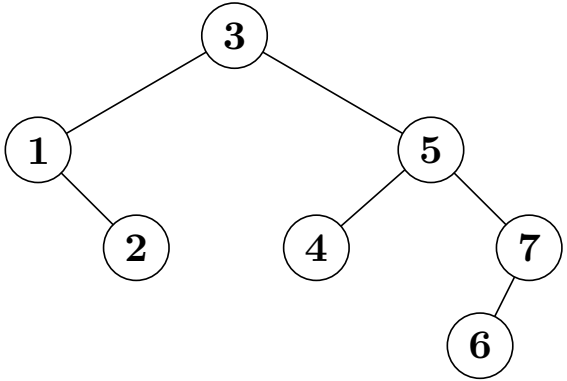
09.01.2020

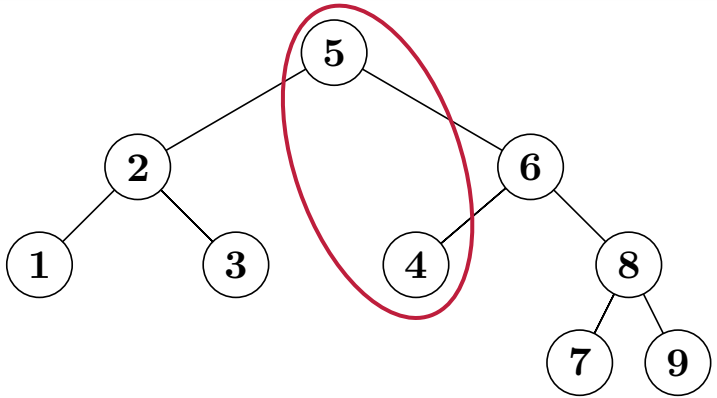
AVL-Bäume

AVL-Bäume – Definition

Ein AVL-Baum besitzt folgenden Eigenschaften:

- Er ist ein binärer Suchbaum.
- Höhe des linken und rechten Teilbaums eines Knotens unterscheidet sich um maximal 1.





AVL-Bäume – Operationen

Operationen für binäre Suchbäume funktionieren auch für AVL-Bäume, d.h. wir können:

- Insert,
- Delete,
- Minimum/Maximum,
- Predecessor/Successor,
- …

ausführen.

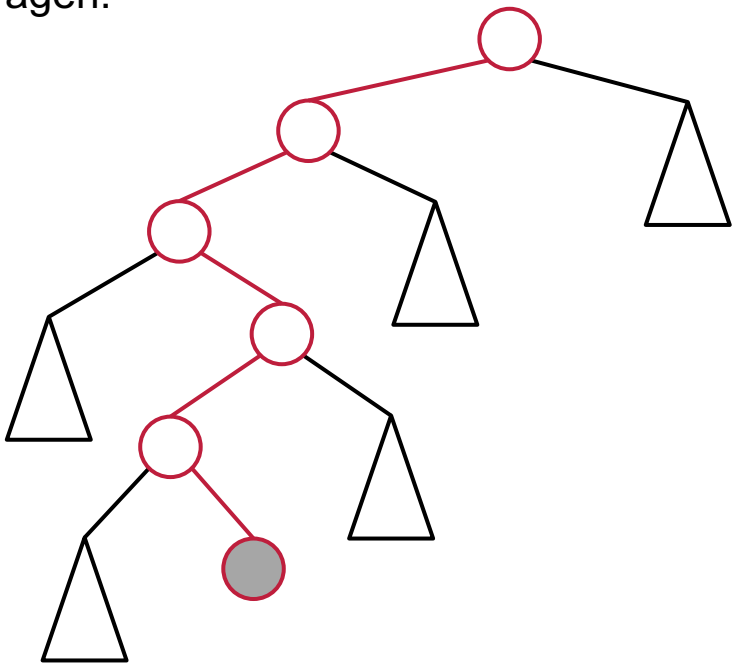
Wir müssen nur auf die Balancierung nach diesen Operationen achten.
Kritisch sind nur *Insert* und *Delete.*

AVL-Bäume – Restructure

Bei Insert und Delete stellen sich nun folgende Fragen:

1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?

Zu 1.: Nur Knoten, die auf dem Pfad von der Wurzel zum eingefügten/gelöschten Knoten liegen



AVL-Bäume – Restructure

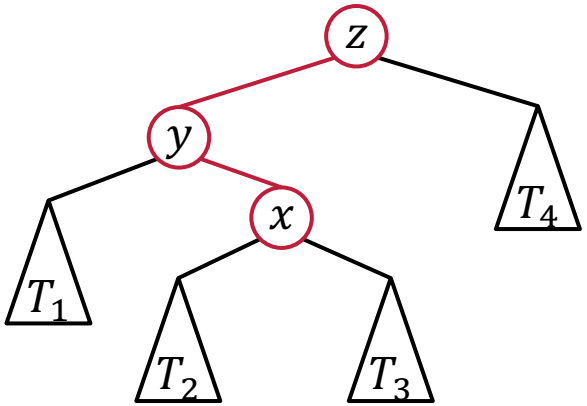
Bei Insert und Delete stellen sich nun folgende Fragen:

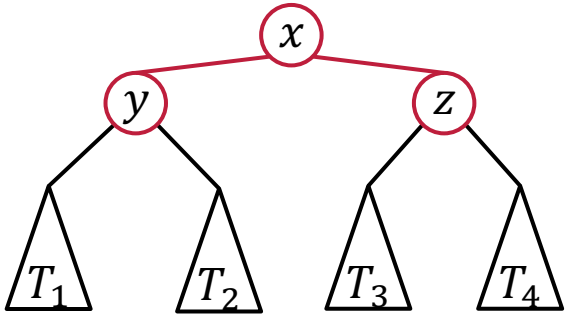
1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?

Zu 2.: Betrachte unbalancierten Knoten z, sein Kind y und dessen Kind x.

Sortiere Elemente aufsteigend und rotiere entsprechend:

1. $x \leq y \leq z$
2. $y \leq x \leq z$
3. $z \leq x \leq y$
4. $z \leq y \leq x$

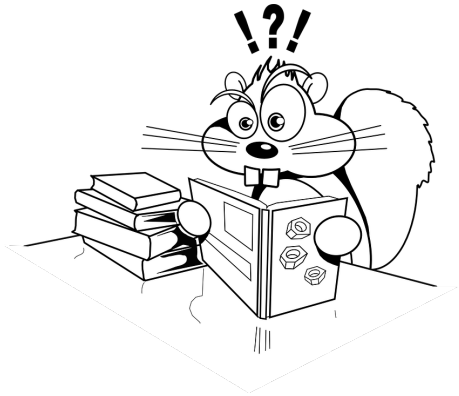




AVL-Bäume – Restructure

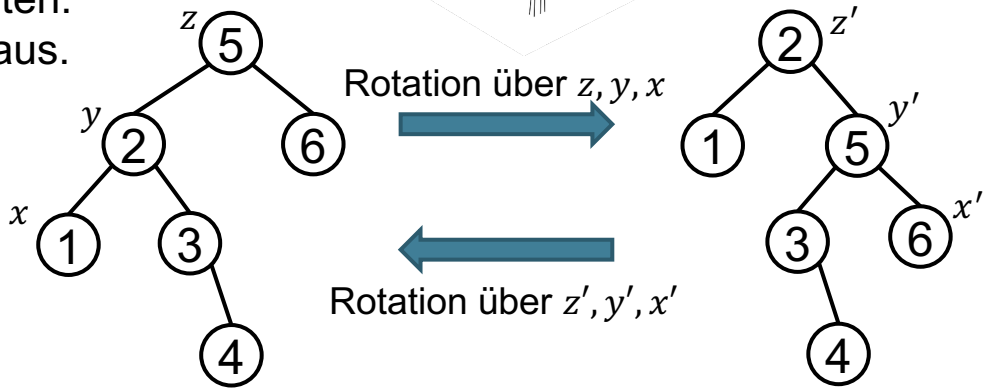
Bei Insert und Delete stellen sich nun folgende Fragen:

1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?



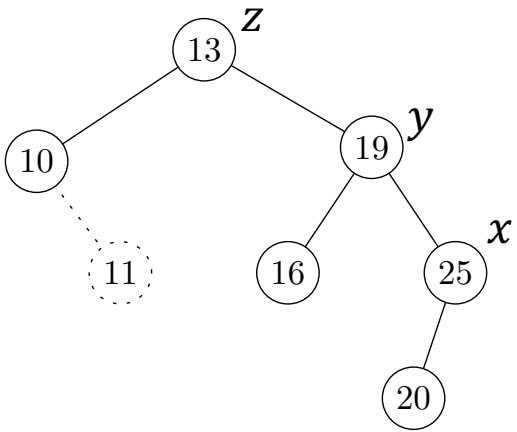
Zu 3.:

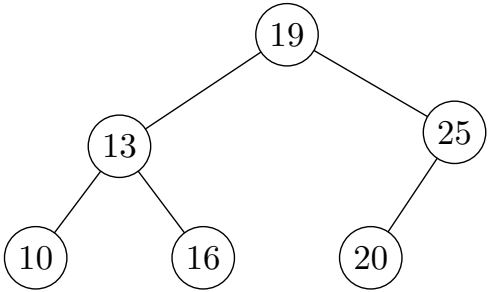
- Starte bei tiefstem unbalancierten Knoten.
- Wähle Kinder (x, y) nach deren Höhe aus.



AVL-Bäume – Beispiele

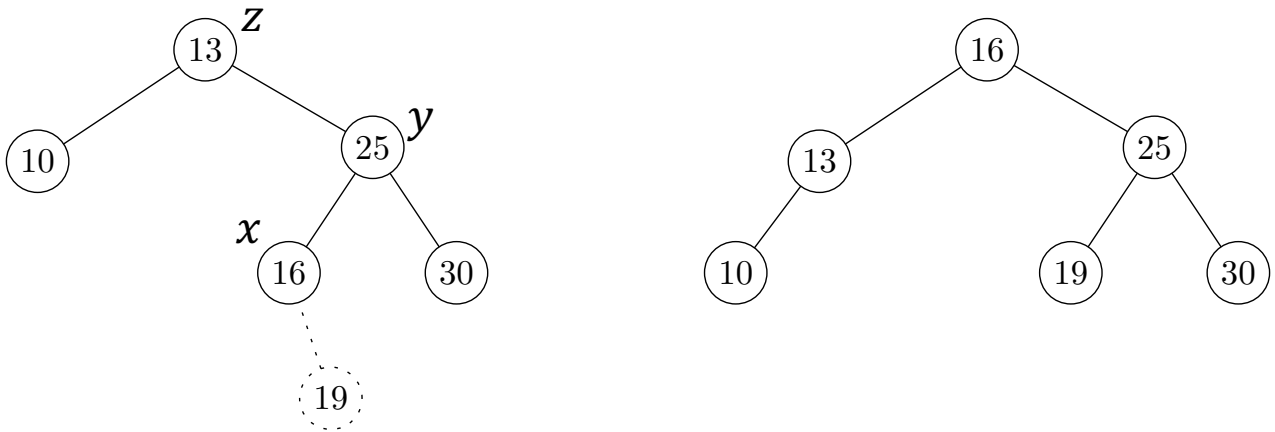
Delete(T, 11)





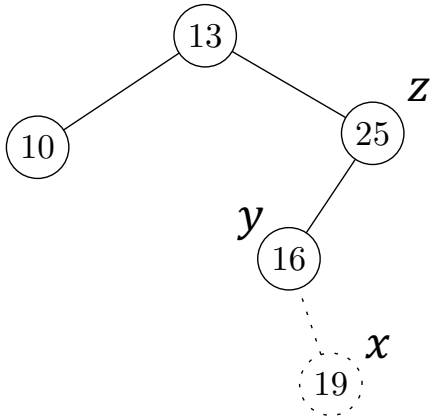
AVL-Bäume – Beispiele

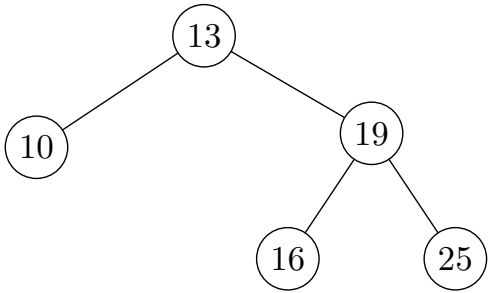
Insert(T, 19)



AVL-Bäume – Beispiele

Insert(T, 19)





AVL-Bäume – Knoten-Balancierung

Wie gut sind AVL-Bäume bzgl. der Knoten balanciert?

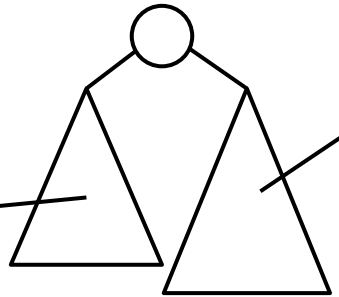
Sei n_L die Anzahl der Knoten im linken Teilbaum und n_R die Anzahl der Knoten im rechten Teilbaum der Wurzel. Dann gilt für einen AVL-Baum der Höhe h:

$$\max \frac{n_R}{n_L} \in \Theta\left(\left(\frac{2}{\varphi}\right)^h\right) \approx \Theta\left(1{,}24^h\right)$$

φ bezeichnet den goldenen Schnitt mit $\varphi := \frac{1+\sqrt{5}}{2}$

Ein AVL-Baum der Höhe h besitzt maximal $2^h - 1$ Knoten.

Ein AVL-Baum der Höhe h besitzt mindestens $F_{h+2} - 1$ Knoten.



Zunächst: Fibonacci-Zahlen

Die n-te Fibonacci Zahl F_n ist die Summe der beiden vorhergehenen Fibonacci Zahlen, d.h. $F_n := F_{n-1} + F_{n-2}$

Es gilt

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Es gilt $F_n \in \Theta(\varphi^n)$

Beweis:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \geq \frac{1}{2\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = \frac{1}{2\sqrt{5}}\varphi^n \text{ und}$$

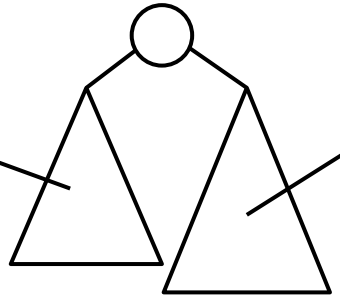
$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \leq \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = \frac{1}{\sqrt{5}}\varphi^n \qquad \blacksquare$$

AVL-Bäume – Knoten-Balancierung

Ein AVL-Baum der Höhe h besitzt mindestens $F_{h+2} - 1$ Knoten.

Ein AVL-Baum der Höhe h besitzt maximal $2^h - 1$ Knoten.

Minimaler Suchbaum der Höhe $h - 2$



Vollständiger Suchbaum der Höhe $h - 1$

$$\frac{n_R}{n_L} \leq \frac{2^{h-1}-1}{\frac{1}{2\sqrt{5}}\varphi^h - 1} \leq \frac{2^{h-1}-1}{\frac{1}{4\sqrt{5}}\varphi^h} \text{ für } h \geq 7 \leq 4\sqrt{5} \cdot \frac{2^{h-1}}{\varphi^h} = 2\sqrt{5}\left(\frac{2}{\varphi}\right)^h$$

Also:

$$\max \frac{n_R}{n_L} \in \Theta\left(\left(\frac{2}{\varphi}\right)^h\right)$$

Mit

$$c_1 = 2\sqrt{5}$$

$$c_2 = \frac{\sqrt{5}}{4}$$

und

$$n_0 = 7$$

$$\frac{n_R}{n_L} \geq \frac{2^{h-1}-1}{\frac{1}{\sqrt{5}}\varphi^h - 1} \geq \frac{\frac{1}{2}2^{h-1}}{\frac{1}{\sqrt{5}}\varphi^h} \text{ für } h \geq 2 \geq \frac{\sqrt{5}}{4} \cdot \frac{2^h}{\varphi^h} = \frac{\sqrt{5}}{4}\left(\frac{2}{\varphi}\right)^h$$

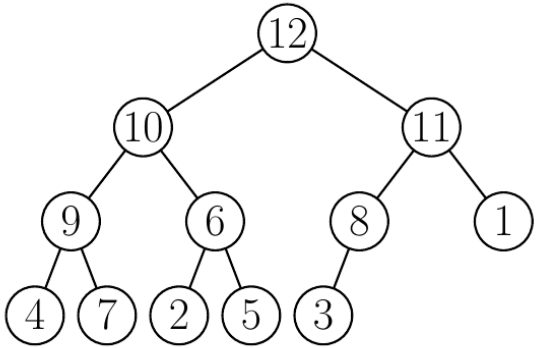
Max-Heaps

Max-Heaps – Definition

Ein Max-Heap ist ein binärer Baum mit folgenden Eigenschaften:

- Jeder Knoten besitzt einen Schlüssel
- Ebene $i < h$ besitzt 2^{i-1} Knoten, wobei h die Höhe des Baumes ist.
- Auf Ebene h sind die linken $n - 2^{h-1} + 1$ Positionen besetzt.
- Der Schlüssel jedes Knotens ist mindestens so groß wie die seiner beiden Kinder.

$$A = [12, 10, 11, 9, 6, 8, 1, 4, 7, 2, 5, 3]$$

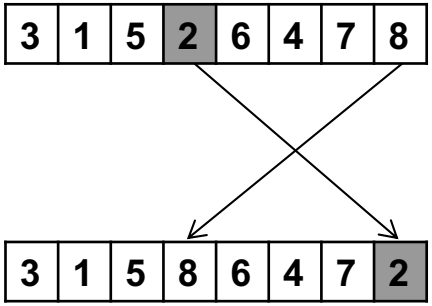


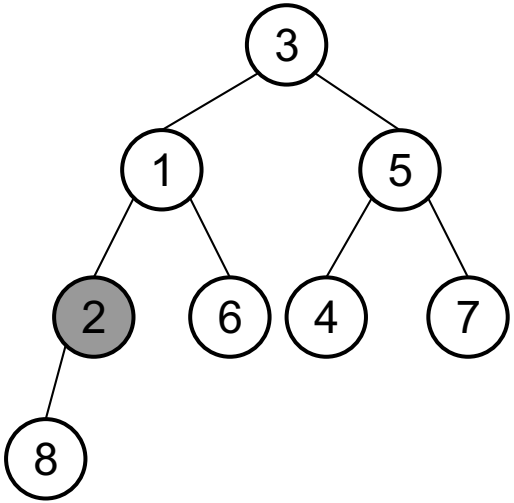
Build-Max-Heap

```
function BUILD-MAX-HEAP(A)
    heap-größe[A] := länge[A]
    for i = ⌊länge[A] / 2⌋ down to 1 do
        MAX-HEAPIFY(A, i)
```

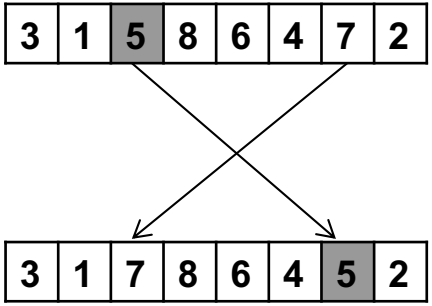
```
function MAX-HEAPIFY(A, i)
    ℓ := links(i), r := rechts(i)
    if ℓ ≤ heap-größe[A] und A[ℓ] > A[i] then
        max := ℓ
    else
        max := i
    if r ≤ heap-größe[A] und A[r] > A[max] then
        max := r
    if max ≠ i then
        Vertausche A[max]und A[i]
        MAX-HEAPIFY(A, max)
```

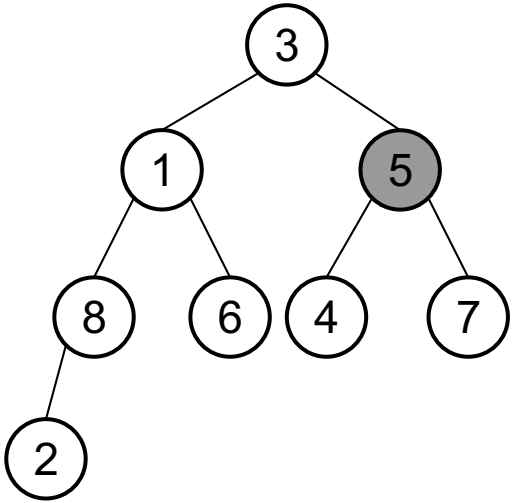
Build-Max-Heap



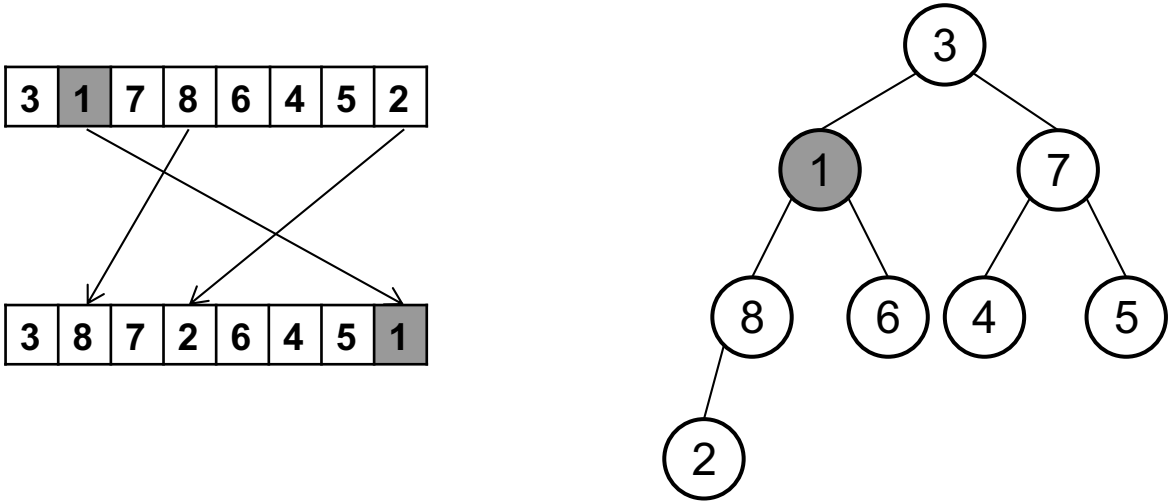


Build-Max-Heap

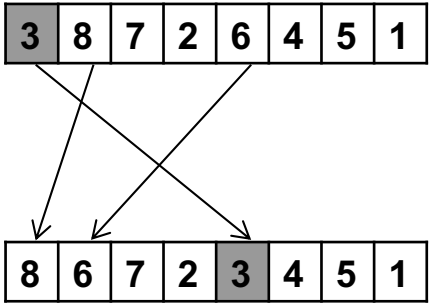


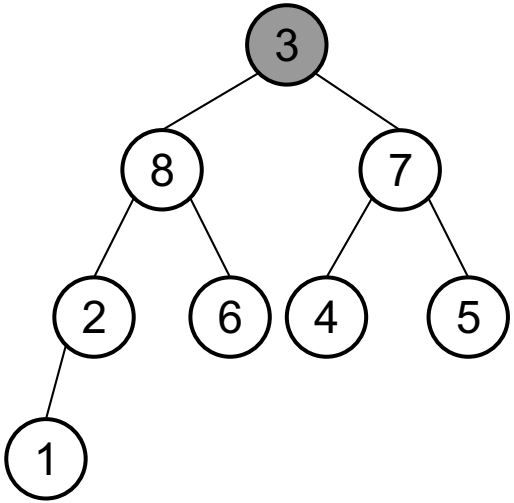


Build-Max-Heap



Build-Max-Heap

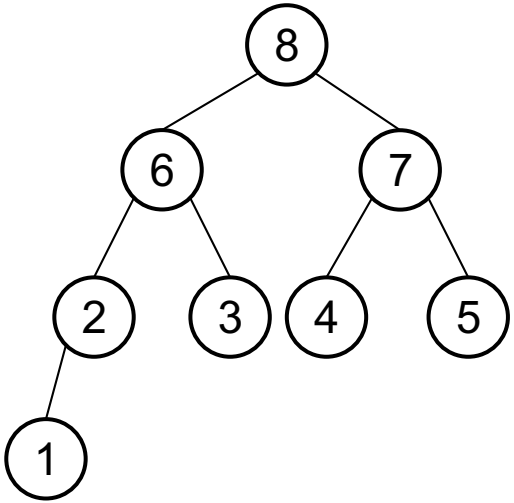




Build-Max-Heap

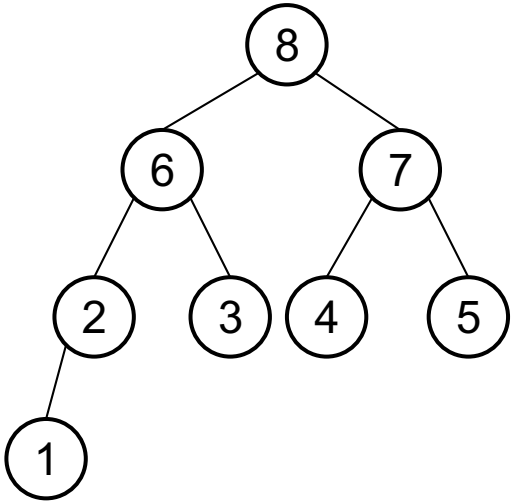
Fertig!

8	6	7	2	3	4	5	1



Heapsort

```
function SORT(A)
    BUILD-MAX-HEAP(A)
    for i ←length(A) downto 2 do
        vertausche A[1] und A[i]
        heap-größe[A] ← heap-größe[A] − 1
        MAX-HEAPIFY(A, 1)
    end for
end function
```



```
function SORT(A)
    BUILD-MAX-HEAP(A)
    for i ←length(A) downto 2 do
        vertausche A[1] und A[i]
        heap-größe[A] ← heap-größe[A] − 1
        MAX-HEAPIFY(A, 1)
    end for
end function
```

8	6	7	2	3	4	5	1
7	6	5	2	3	4	1	8
6	3	5	2	1	4	7	8
5	3	4	2	1	6	7	8
4	3	1	2	5	6	7	8
3	2	1	4	5	6	7	8
2	1	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Heapsort – Laufzeit

```
function SORT(A)
    BUILD-MAX-HEAP(A)
    for i ←length(A) downto 2 do
        vertausche A[1] und A[i]
        heap-größe[A] ← heap-größe[A] − 1
        MAX-HEAPIFY(A, 1)
    end for
end function
```

Build-Max-Heap: $O(n)$

In der For-Schleife:

1. Vertauschen: $O(1)$
2. Vertauschen: $O(1)$
3. Max-Heapify: $O(\log n)$

Für n Iterationen ist das $O(n \log n)$

Heapsort – Korrektheit

Heapsort ist ein Sortierverfahren.

Frage: Welche Beweistechnik ist das?

Beweis:

- Zunächst benutzen wir einen Max-heap.
  Das gibt uns das größte Element an der ersten Stelle.
- In Iteration i sind noch $n - i + 1$ Felder des Arrays aktiv.
- Angenommen, wir haben zu Beginn der Iteration i einen Max-Heap auf den aktiven Feldern.
- Nach dem Tauschen des i größten Elements nach hinten und nach Verkleinern des Arrays:
  $\Rightarrow$ U.U. kein Max-Heap vorhanden.
- Aber: Jeder Knoten außer Wurzel erfüllt Max-Heap Eigenschaft.
- Also: Max-Heapify auf Wurzelknoten reicht aus, um einen Max-Heap wiederherzustellen!
- Wir tauschen also in jeder Iteration das richtige Element nach hinten!