



Technische
Universität
Braunschweig



Algorithmen und Datenstrukturen – Übung #5

Mergesort, Master-Theorem, Heapsort

Ramin & Chek-Manh

09.01.2025

Mergesort

Mergesort

Algorithmus (grob):

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

```
function MERGESORT( $A, p, r$ )  
  if  $p < r$  then  
     $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$   
    MERGESORT( $A, p, q$ )  
    MERGESORT( $A, q + 1, r$ )  
    MERGE( $A, p, q, r$ )
```

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$							
2. $A =$							
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$							
2. $A =$							
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$							
2. $A =$							
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$							
2. $A =$							
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$							
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$							
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$							
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$							
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$							
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$							

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1						

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1	2					

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1	2	3				

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1	2	3	4			

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1	2	3	4	5		

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1	2	3	4	5	6	

Mergesort

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

$A =$	4	2	6	3	1	5	7
1. $A =$	2	4					
2. $A =$			3	6			
3. $A =$	2	3	4	6			
4. $A =$					1	5	
5. $A =$					1	5	7
6. $A =$	1	2	3	4	5	6	7

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Sei $T(n)$ die Laufzeit von Mergesort für n Elemente

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Sei $T(n)$ die Laufzeit von Mergesort für n Elemente

Schritt 1: $T\left(\frac{n}{2}\right)$

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Sei $T(n)$ die Laufzeit von Mergesort für n Elemente

Schritt 1: $T\left(\frac{n}{2}\right)$

Schritt 2: $T\left(\frac{n}{2}\right)$

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Sei $T(n)$ die Laufzeit von Mergesort für n Elemente

Schritt 1: $T\left(\frac{n}{2}\right)$

Schritt 2: $T\left(\frac{n}{2}\right)$

Schritt 3: $O(n)$

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Sei $T(n)$ die Laufzeit von Mergesort für n Elemente

Schritt 1: $T\left(\frac{n}{2}\right)$

Schritt 2: $T\left(\frac{n}{2}\right)$

Schritt 3: $O(n)$

Zusammen ergibt das:

Mergesort - Laufzeit

Algorithmus (grob):

1. Sortiere linke Hälfte
2. Sortiere rechte Hälfte
3. Füge beide Hälften zusammen (merge)

Sei $T(n)$ die Laufzeit von Mergesort für n Elemente

Schritt 1: $T\left(\frac{n}{2}\right)$

Schritt 2: $T\left(\frac{n}{2}\right)$

Schritt 3: $O(n)$

Zusammen ergibt das:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

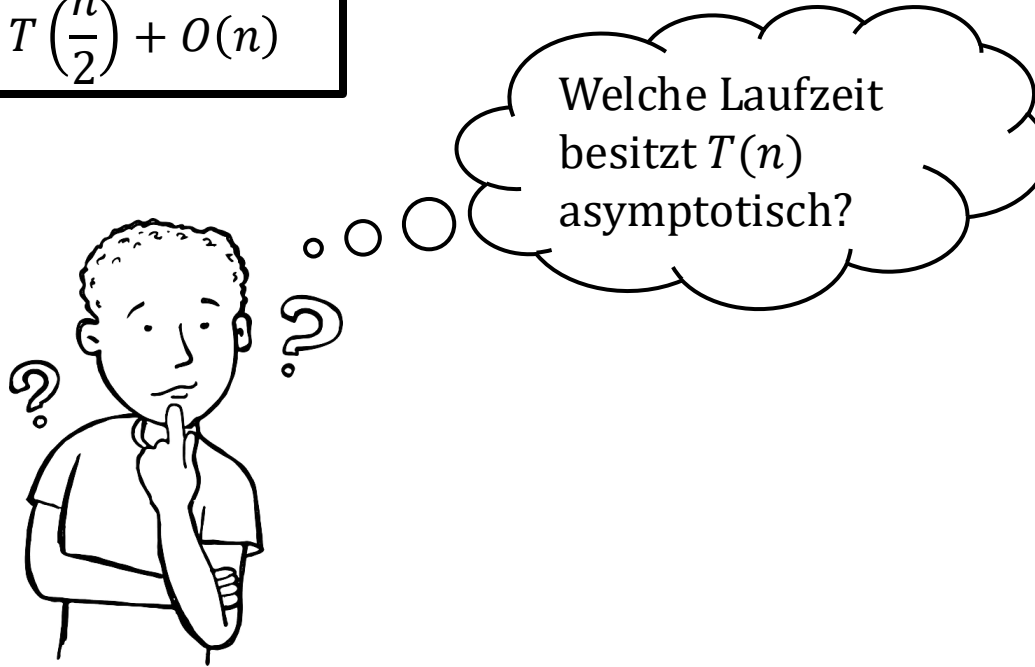
Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$



Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$



Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

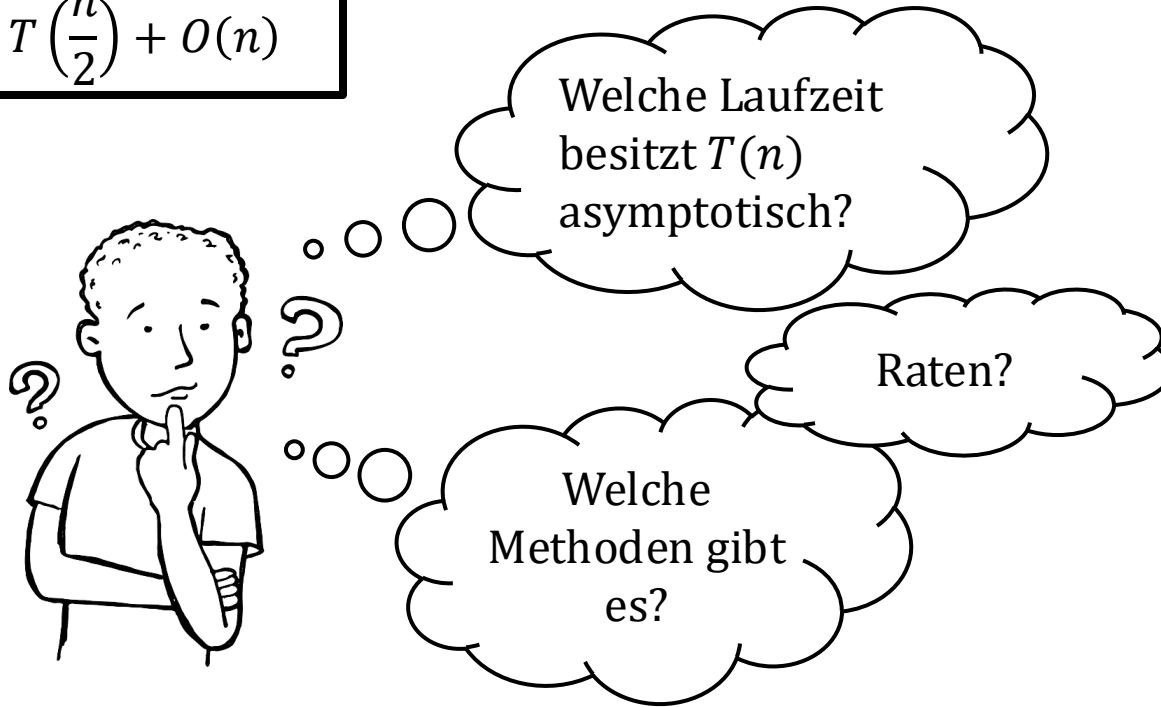


Welche Laufzeit
besitzt $T(n)$
asymptotisch?

Welche
Methoden gibt
es?

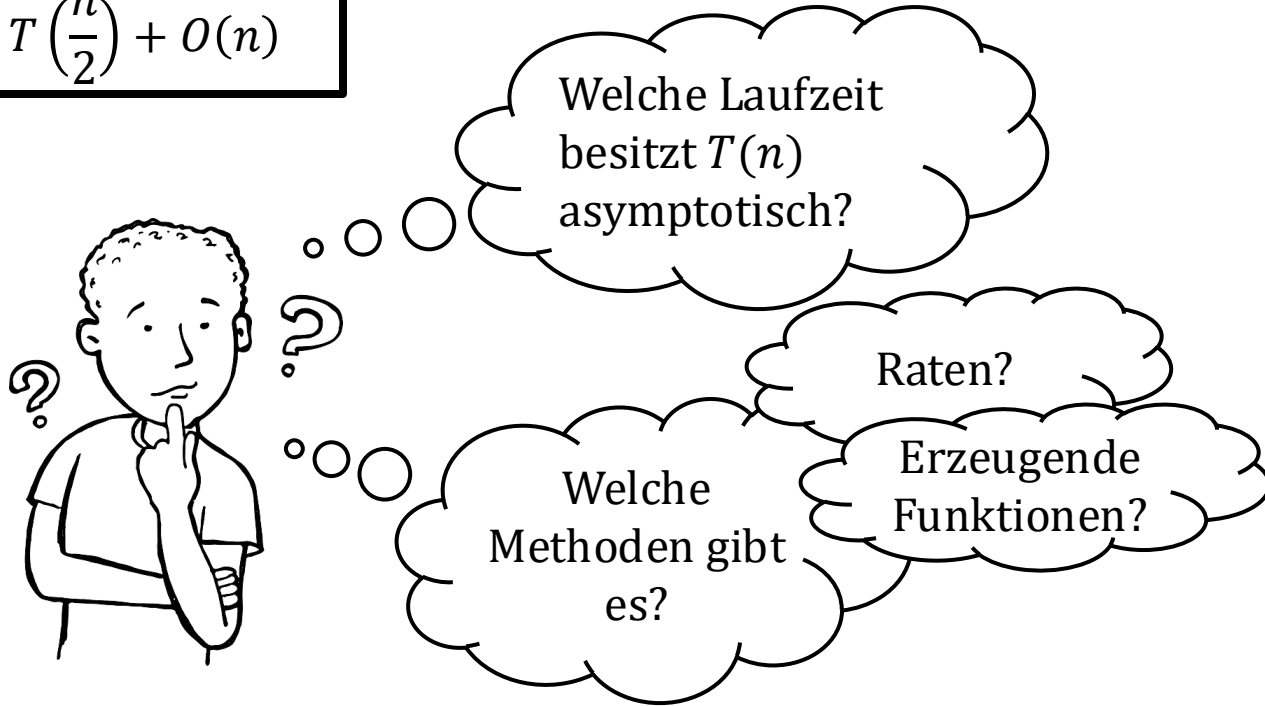
Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$



Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$



Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$



Welche Laufzeit
besitzt $T(n)$
asymptotisch?

Raten?

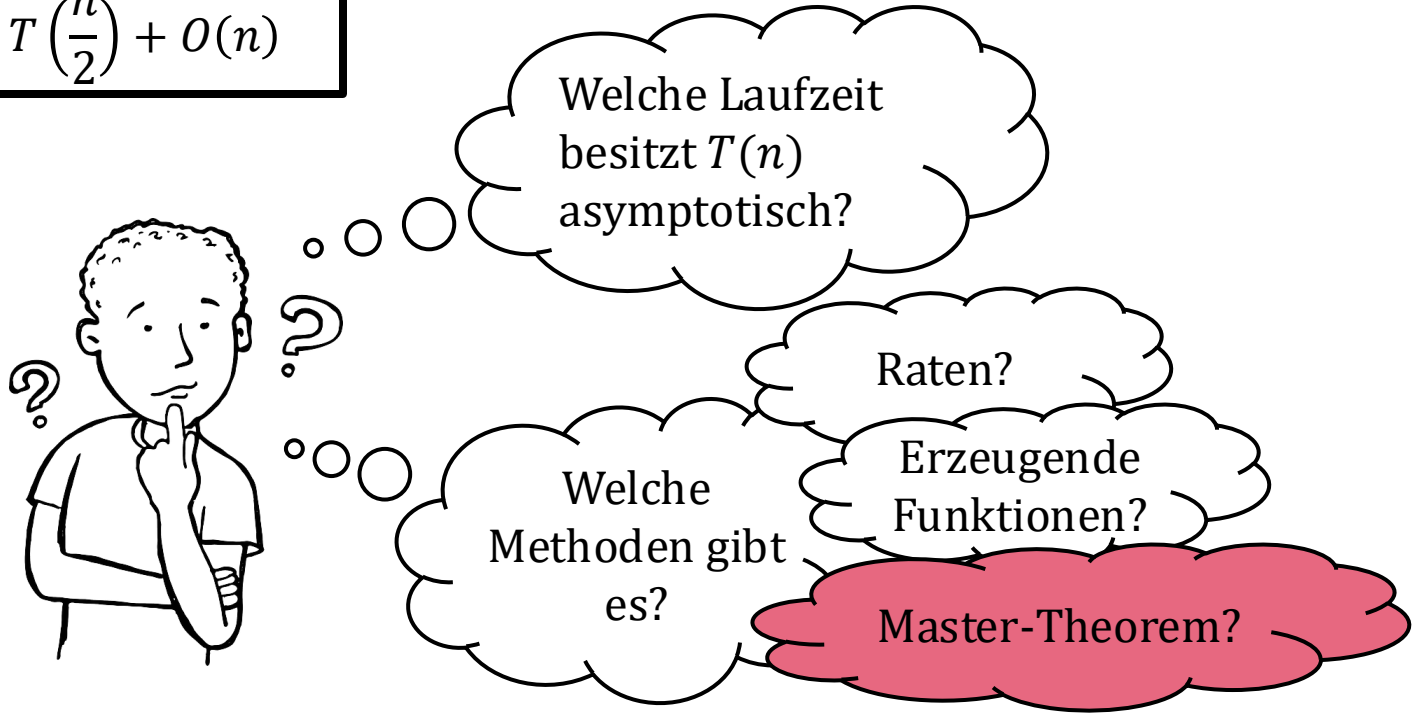
Erzeugende
Funktionen?

Master-Theorem?

Welche
Methoden gibt
es?

Lösen von Rekursionsgleichungen

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$



Master-Theorem

Das Master-Theorem



Das Master-Theorem

Satz: Sei $T: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion der Form

$$T(n) := \sum_{i=1}^m T(\alpha_i \cdot n) + \Theta(n^k),$$

Das Master-Theorem

Satz: Sei $T: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion der Form

$$T(n) := \sum_{i=1}^m T(\alpha_i \cdot n) + \Theta(n^k),$$

wobei $m \in \mathbb{N}$, $k \in \mathbb{R}$ und $0 < \alpha_i < 1$ für alle $i \in \{1, \dots, m\}$.

Das Master-Theorem

Satz: Sei $T: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion der Form

$$T(n) := \sum_{i=1}^m T(\alpha_i \cdot n) + \Theta(n^k),$$

wobei $m \in \mathbb{N}$, $k \in \mathbb{R}$ und $0 < \alpha_i < 1$ für alle $i \in \{1, \dots, m\}$.

Dann gilt

Das Master-Theorem

Satz: Sei $T: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion der Form

$$T(n) := \sum_{i=1}^m T(\alpha_i \cdot n) + \Theta(n^k),$$

wobei $m \in \mathbb{N}$, $k \in \mathbb{R}$ und $0 < \alpha_i < 1$ für alle $i \in \{1, \dots, m\}$.

Dann gilt

$$T(n) \in \begin{cases} \Theta(n^k), & \text{falls } \sum_{i=1}^m \alpha_i^k < 1 \\ \Theta(n^k \log n), & \text{falls } \sum_{i=1}^m \alpha_i^k = 1 \\ \Theta(n^c) \text{ mit } \sum_{i=1}^m \alpha_i^c = 1, & \text{falls } \sum_{i=1}^m \alpha_i^k > 1 \end{cases}$$

Master-Theorem (Beispiele)

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$

Also:

$$m = 9, k = 2 \text{ und } \alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$

Also:

$$m = 9, k = 2 \text{ und } \alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + \underbrace{7n \log n - 8n + 15n^2}_{\Theta(n^2)}$$

Also:

$$m = 9, k = 2 \text{ und } \alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + \underbrace{7n \log n - 8n + 15n^2}_{\Theta(n^2)}$$

Also:

$$m = 9, k = 2 \text{ und } \alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + \underbrace{7n \log n - 8n + 15n^2}_{\Theta(n^2)}$$

Also:

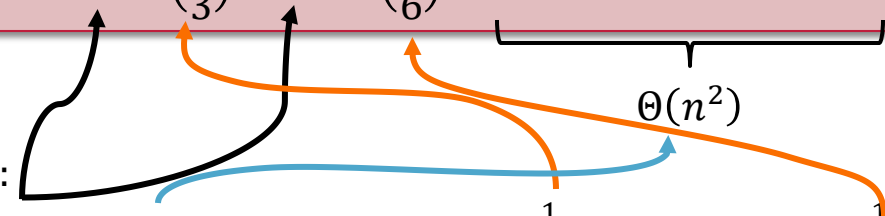
$m = 9, k = 2$ und $\alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + \underbrace{7n \log n - 8n + 15n^2}_{\Theta(n^2)}$$

Also:

$m = 9, k = 2$ und $\alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$



Mit

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + \underbrace{7n \log n - 8n + 15n^2}_{\Theta(n^2)}$$

Also:

$m = 9, k = 2$ und $\alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$

Mit

$$\sum_{i=1}^9 \alpha_i^2 = 3 \cdot \left(\frac{1}{3}\right)^2 + 6 \cdot \left(\frac{1}{6}\right)^2 = \frac{1}{3} + \frac{1}{6} < \frac{1}{2} + \frac{1}{2} = 1$$

Beispiel 1

$$U(n) = 3 \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + \underbrace{7n \log n - 8n + 15n^2}_{\Theta(n^2)}$$

Also:

$m = 9, k = 2$ und $\alpha_1 = \dots = \alpha_3 = \frac{1}{3}, \alpha_4 = \dots = \alpha_9 = \frac{1}{6}$

Mit

$$\sum_{i=1}^9 \alpha_i^2 = 3 \cdot \left(\frac{1}{3}\right)^2 + 6 \cdot \left(\frac{1}{6}\right)^2 = \frac{1}{3} + \frac{1}{6} < \frac{1}{2} + \frac{1}{2} = 1$$

folgt $U(n) \in \Theta(n^2)$

Beispiel 2

$$V(n) = 7n^2 + 5n^3 + 27 \cdot V\left(\frac{n}{3}\right) - 7n$$

Beispiel 2

$$V(n) = 7n^2 + 5n^3 + 27 \cdot V\left(\frac{n}{3}\right) - 7n$$

Also:

$$m = 27, k = 3 \text{ und } \alpha_1 = \dots = \alpha_{27} = \frac{1}{3}$$

Beispiel 2

$$V(n) = 7n^2 + 5n^3 + 27 \cdot V\left(\frac{n}{3}\right) - 7n$$

Also:

$$m = 27, k = 3 \text{ und } \alpha_1 = \dots = \alpha_{27} = \frac{1}{3}$$

Mit

Beispiel 2

$$V(n) = 7n^2 + 5n^3 + 27 \cdot V\left(\frac{n}{3}\right) - 7n$$

Also:

$$m = 27, k = 3 \text{ und } \alpha_1 = \dots = \alpha_{27} = \frac{1}{3}$$

Mit

$$\sum_{i=1}^{27} \alpha_i^3 = 27 \cdot \left(\frac{1}{3}\right)^3 = \frac{27}{27} = 1$$

Beispiel 2

$$V(n) = 7n^2 + 5n^3 + 27 \cdot V\left(\frac{n}{3}\right) - 7n$$

Also:

$$m = 27, k = 3 \text{ und } \alpha_1 = \dots = \alpha_{27} = \frac{1}{3}$$

Mit

$$\sum_{i=1}^{27} \alpha_i^3 = 27 \cdot \left(\frac{1}{3}\right)^3 = \frac{27}{27} = 1$$

folgt $V(n) \in \Theta(n^3 \log n)$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Also suche c , sodass gilt

$$\sum_{i=1}^9 \alpha_i^c = 1$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Also suche c , sodass gilt

$$\sum_{i=1}^9 \alpha_i^c = 1$$

$$\Leftrightarrow 9 \cdot \left(\frac{1}{3}\right)^c = 1$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Also suche c , sodass gilt

$$\sum_{i=1}^9 \alpha_i^c = 1$$

$$\Leftrightarrow 9 \cdot \left(\frac{1}{3}\right)^c = 1$$

$$\Leftrightarrow 9 = 3^c$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Also suche c , sodass gilt

$$\sum_{i=1}^9 \alpha_i^c = 1$$

$$\Leftrightarrow 9 \cdot \left(\frac{1}{3}\right)^c = 1$$

$$\Leftrightarrow 9 = 3^c$$

$$\Leftrightarrow \log_3 9 = c$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Also suche c , sodass gilt

$$\sum_{i=1}^9 \alpha_i^c = 1$$

$$\Leftrightarrow 9 \cdot \left(\frac{1}{3}\right)^c = 1$$

$$\Leftrightarrow 9 = 3^c$$

$$\Leftrightarrow \log_3 9 = c$$

$$\Leftrightarrow 2 = c$$

Beispiel 3

$$W(n) = 9 \cdot W\left(\frac{n}{3}\right) + 7n$$

Also:

$$m = 9, k = 1 \text{ und } \alpha_1 = \dots = \alpha_9 = \frac{1}{3}$$

Da

$$\sum_{i=1}^9 \alpha_i^1 = 9 \cdot \left(\frac{1}{3}\right)^1 = 3 > 1$$

müssen wir das c suchen!

Also suche c , sodass gilt

$$\sum_{i=1}^9 \alpha_i^c = 1$$

$$\Leftrightarrow 9 \cdot \left(\frac{1}{3}\right)^c = 1$$

$$\Leftrightarrow 9 = 3^c$$

$$\Leftrightarrow \log_3 9 = c$$

$$\Leftrightarrow 2 = c$$

Also:

$$W(n) \in \Theta(n^2)$$

Beispiel 4

$$U(n) = \frac{1}{2} \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$

Beispiel 4

$$U(n) = \frac{1}{2} \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$



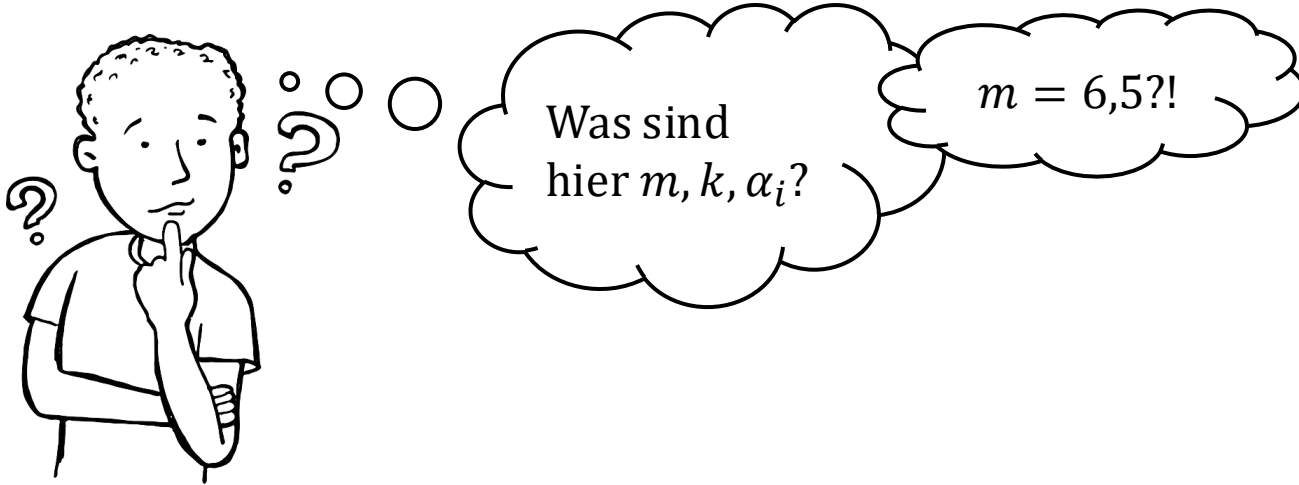
Beispiel 4

$$U(n) = \frac{1}{2} \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$



Beispiel 4

$$U(n) = \frac{1}{2} \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$



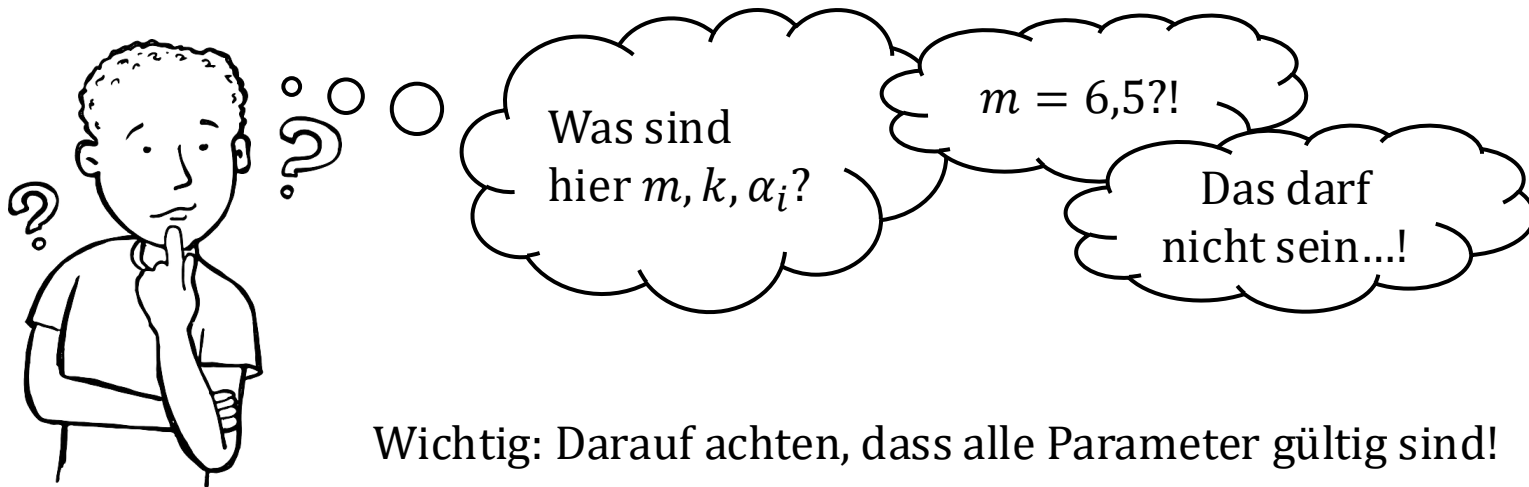
Beispiel 4

$$U(n) = \frac{1}{2} \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$



Beispiel 4

$$U(n) = \frac{1}{2} \cdot U\left(\frac{n}{3}\right) + 6 \cdot U\left(\frac{n}{6}\right) + 7n \log n - 8n + 15n^2$$



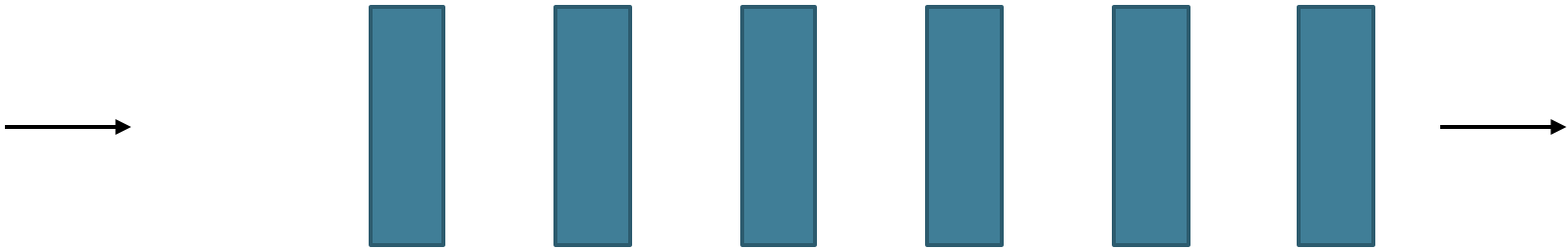
Wichtig: Darauf achten, dass alle Parameter gültig sind!

(cooles Zeug mit)

Max-Heaps

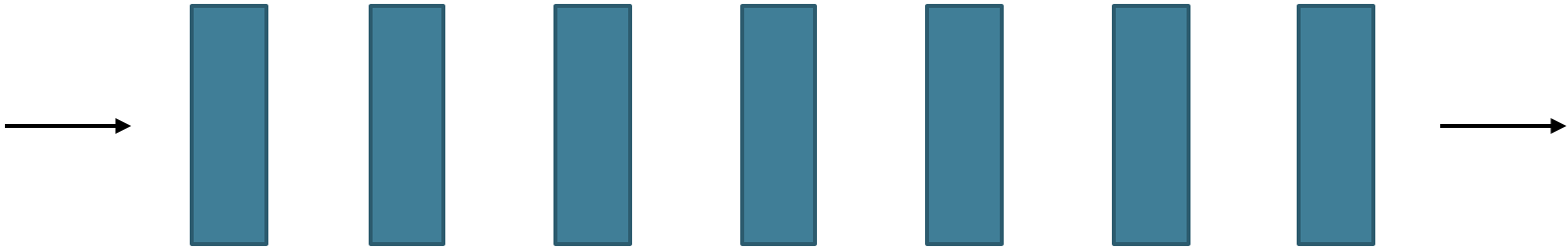
Priority Queue

Queue



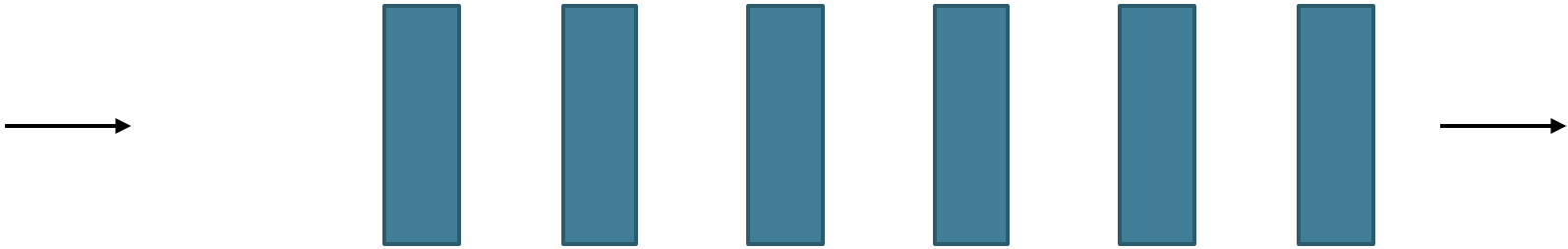
Priority Queue

Queue



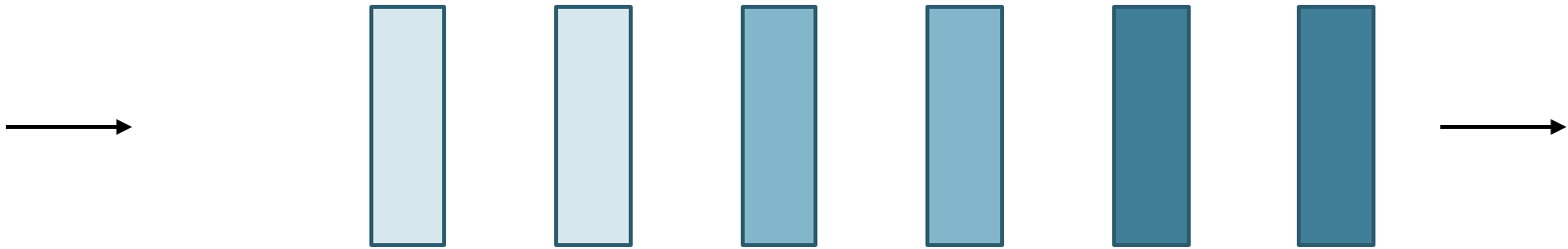
Priority Queue

Queue



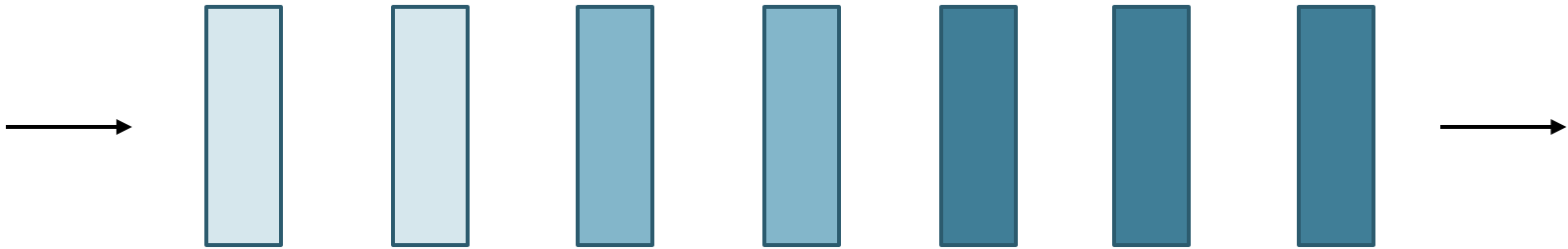
Priority Queue

Priority Queue



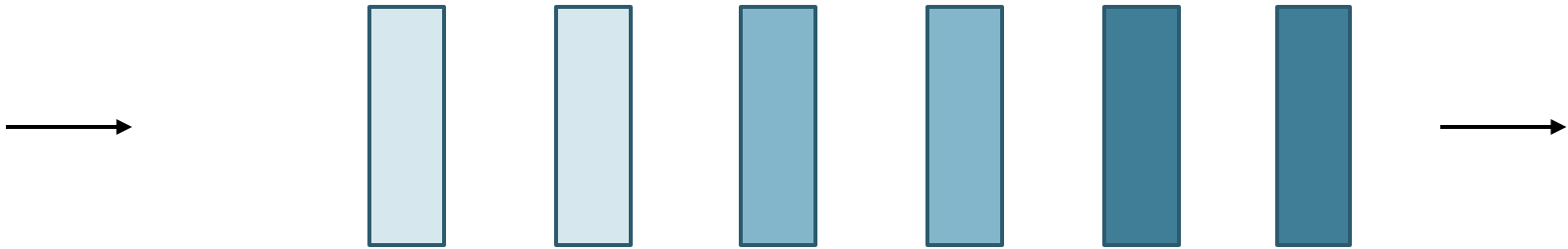
Priority Queue

Priority Queue



Priority Queue

Priority Queue



Operationen

Operationen

Insert(A, x): Füge x zu A hinzu.

Operationen

Insert(A, x): Füge x zu A hinzu.

max(A): Gebe das größte Element in x aus.

Operationen

Insert(A, x): Füge x zu A hinzu.

max(A): Gebe das größte Element in x aus.

extract_max(A): ... und lösche es aus A

Heaps

Heaps

Heaps werden in der Regel als Arrays gespeichert:

Heaps

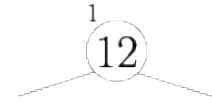
Heaps werden in der Regel als Arrays gespeichert:

$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$

Heaps

Heaps werden in der Regel als Arrays gespeichert:

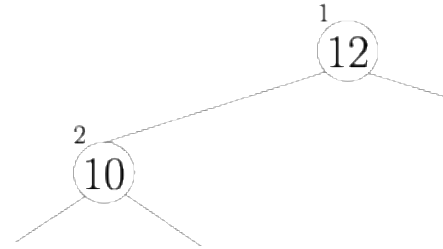
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

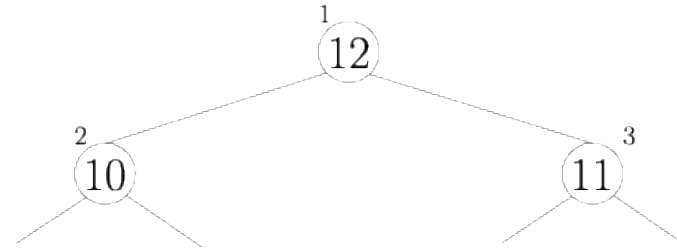
$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

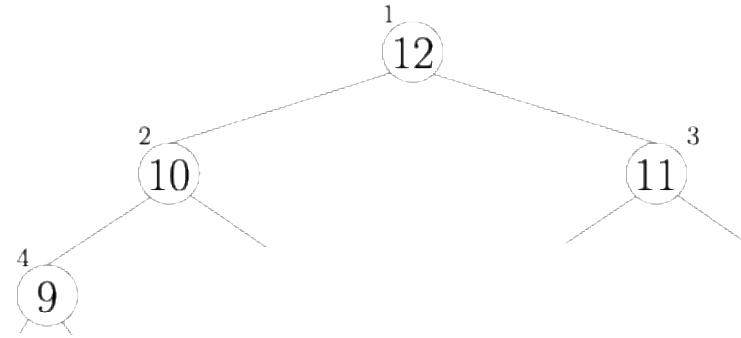
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

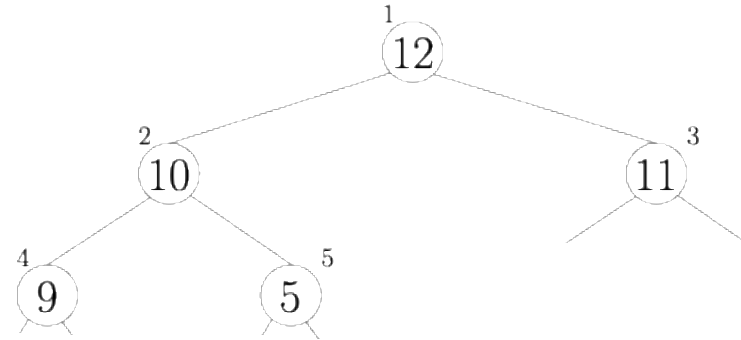
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

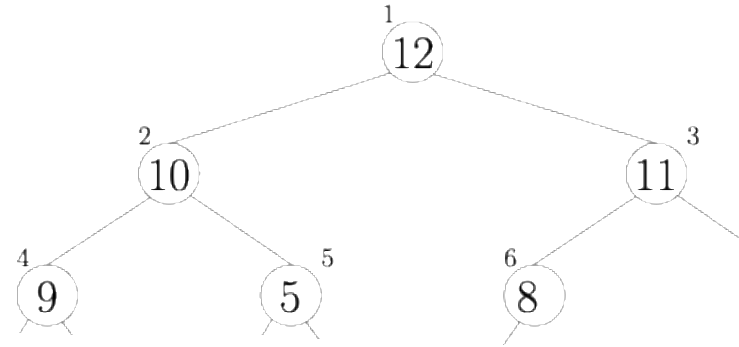
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

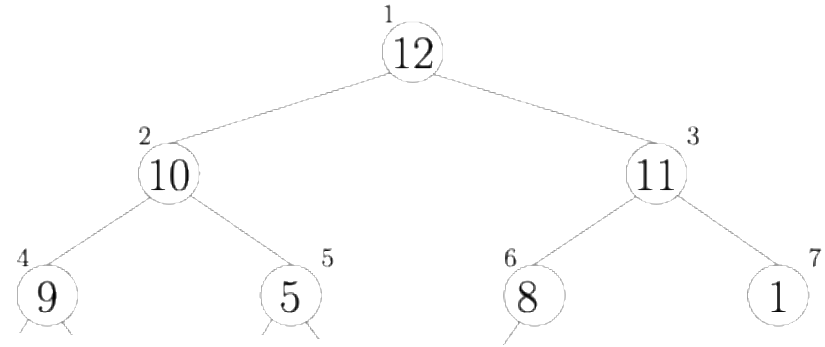
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

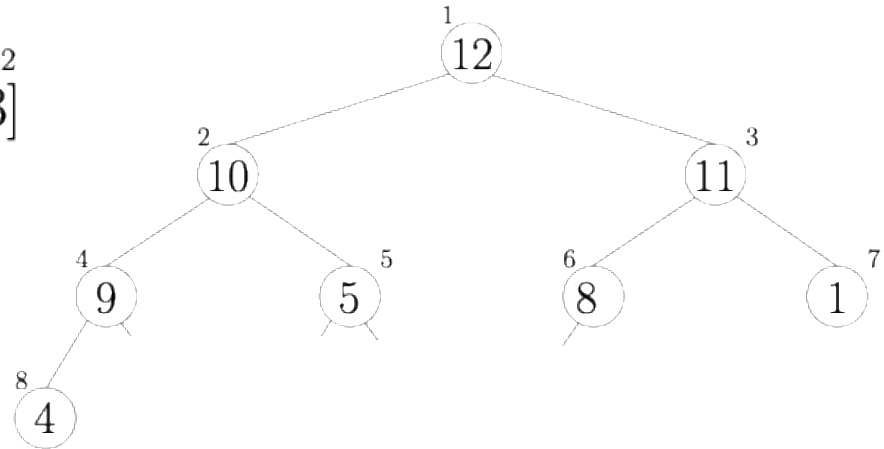
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

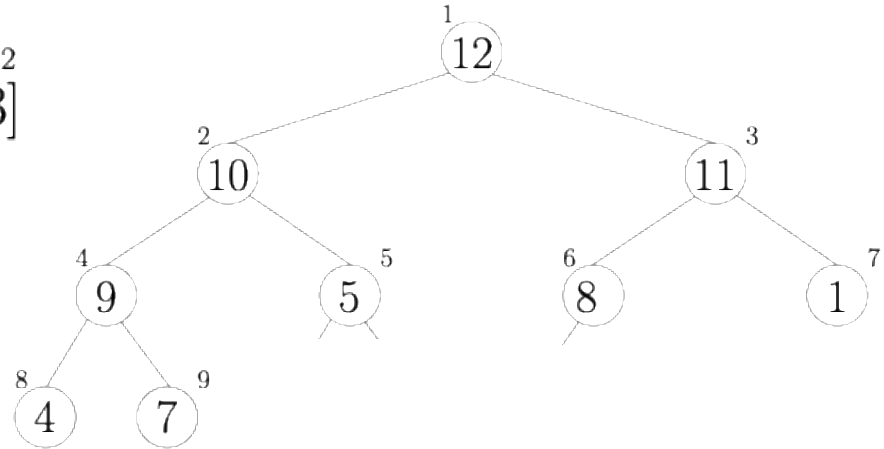
$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

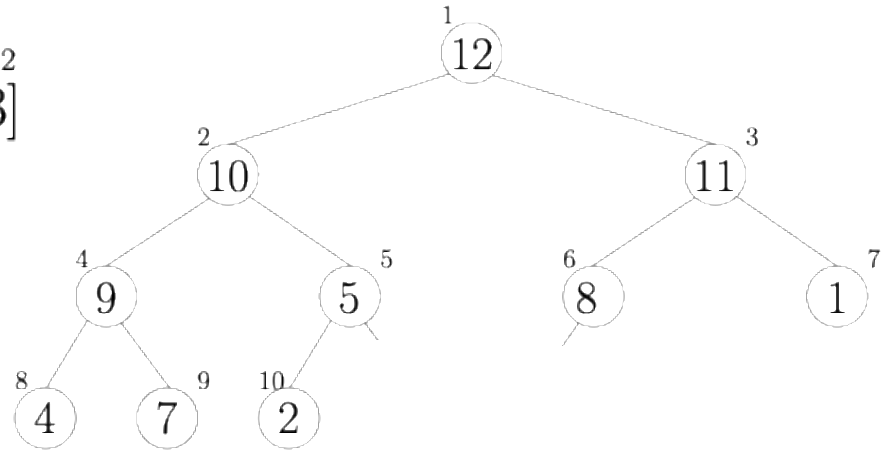
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

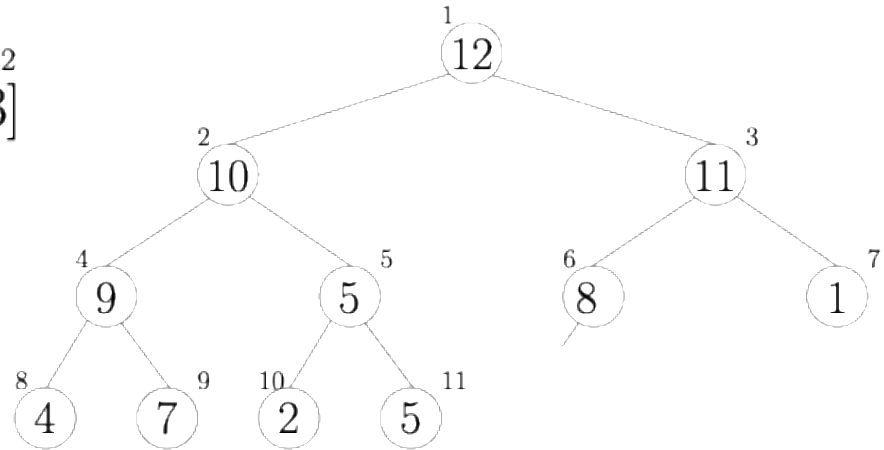
$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$



Heaps

Heaps werden in der Regel als Arrays gespeichert:

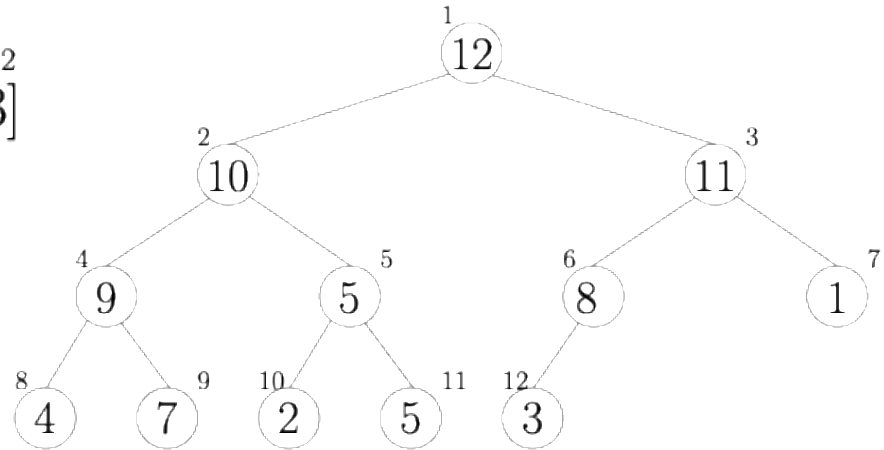
$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$



Heaps

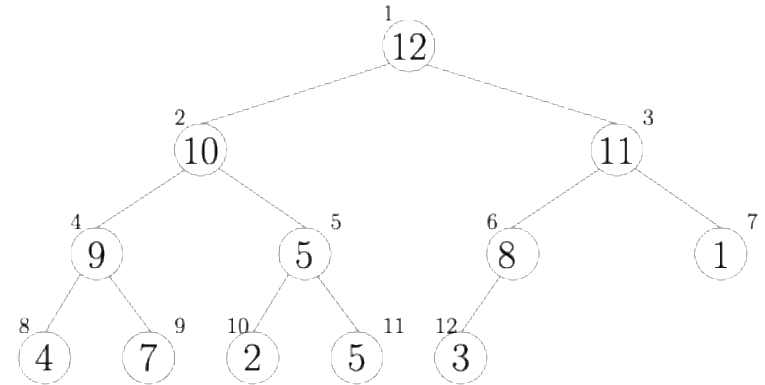
Heaps werden in der Regel als Arrays gespeichert:

$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

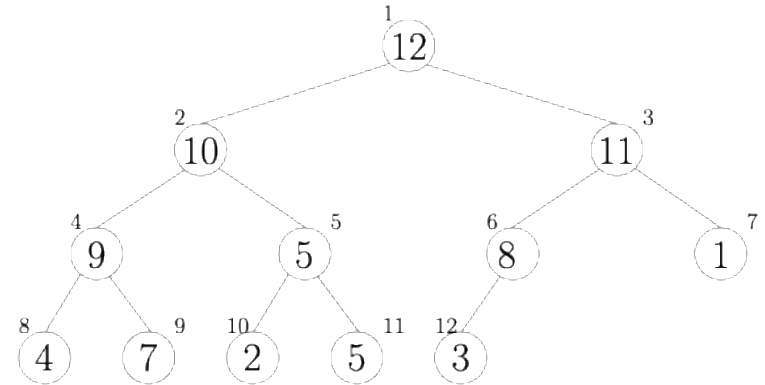
$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$



Heaps

$$A = \overset{1}{12}, \overset{2}{10}, \overset{3}{11}, \overset{4}{9}, \overset{5}{5}, \overset{6}{8}, \overset{7}{1}, \overset{8}{4}, \overset{9}{7}, \overset{10}{2}, \overset{11}{5}, \overset{12}{3}$$

Wurzel: Erstes Element ($i = 1$).

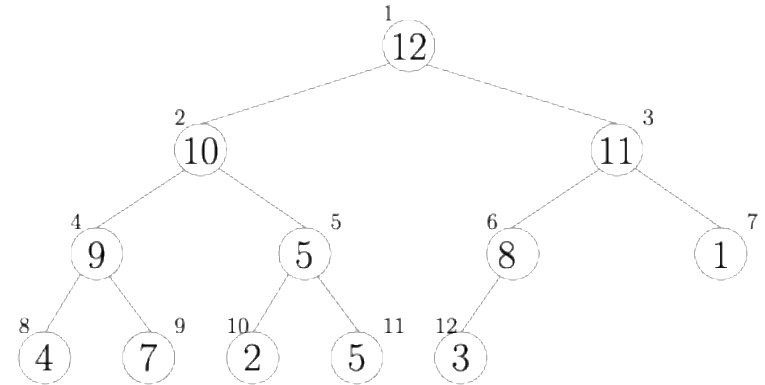


Heaps

$$A = \overset{1}{12}, \overset{2}{10}, \overset{3}{11}, \overset{4}{9}, \overset{5}{5}, \overset{6}{8}, \overset{7}{1}, \overset{8}{4}, \overset{9}{7}, \overset{10}{2}, \overset{11}{5}, \overset{12}{3}$$

Wurzel: Erstes Element ($i = 1$).

$parent(i)$: $i/2$.



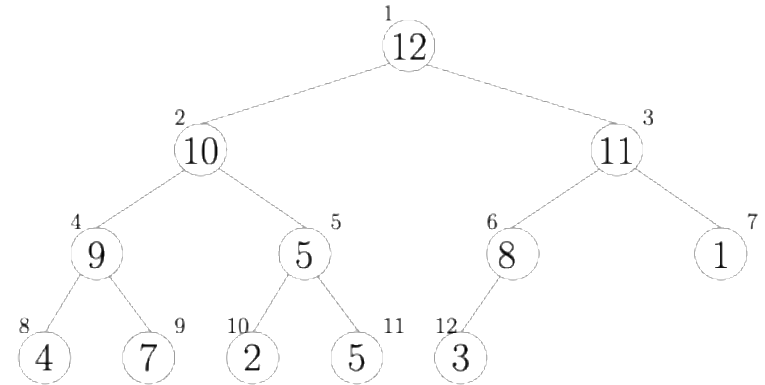
Heaps

$$A = \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ [12, & 10, & 11, & 9, & 5, & 8, & 1, & 4, & 7, & 2, & 5, & 3] \end{matrix}$$

Wurzel: Erstes Element ($i = 1$).

$parent(i): i/2$.

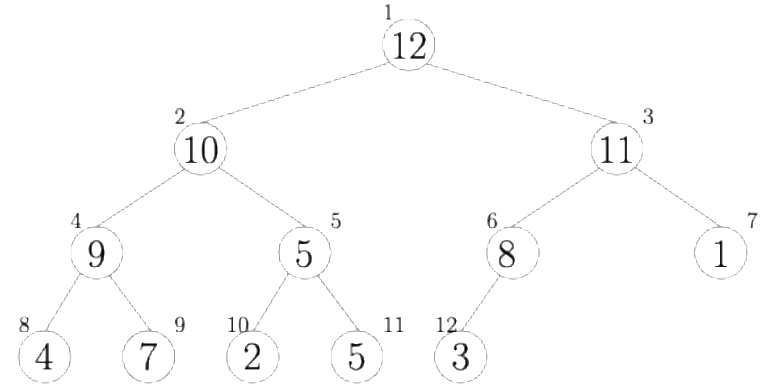
$left(i): 2i;$ $right(i): 2i + 1$.



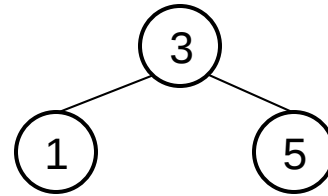
Max-Heaps

$$A = \overset{1}{12}, \overset{2}{10}, \overset{3}{11}, \overset{4}{9}, \overset{5}{5}, \overset{6}{8}, \overset{7}{1}, \overset{8}{4}, \overset{9}{7}, \overset{10}{2}, \overset{11}{5}, \overset{12}{3}$$

Der Schlüssel jedes Knotens ist mindestens so groß wie die seiner beiden Kinder.



Max-Heaps



Heap-Operations

Heap-Operations

build_max_heap(A): konstruiert einen Max Heap aus

Heap-Operations

build_max_heap(A): konstruiert einen Max Heap aus
einem unsortierten Array

Heap-Operations

build_max_heap(A): konstruiert einen Max Heap aus
einem unsortierten Array

max_heapify: Repariere die Heap Eigenschaft in einem Teilbaum.

Heap-Operations

build_max_heap(A): konstruiert einen Max Heap aus
einem unsortierten Array

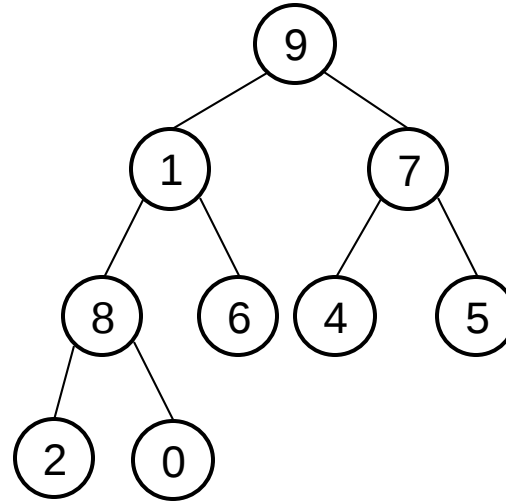
max_heapify: Repariere die Heap Eigenschaft in einem Teilbaum.

Max Heapify

Als Array

9	1	7	8	6	4	5	2	0
---	---	---	---	---	---	---	---	---

Als Baum

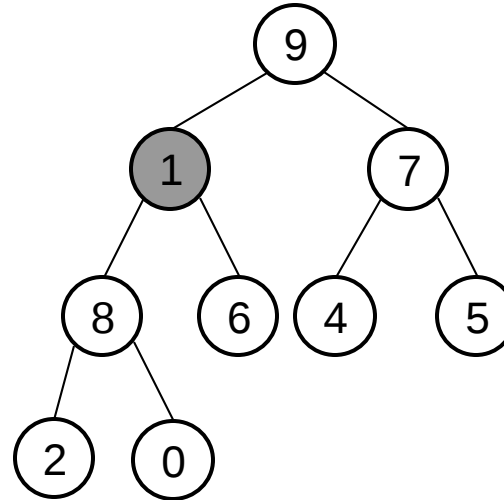


Max Heapify

Als Array

9	1	7	8	6	4	5	2	0
---	---	---	---	---	---	---	---	---

Als Baum

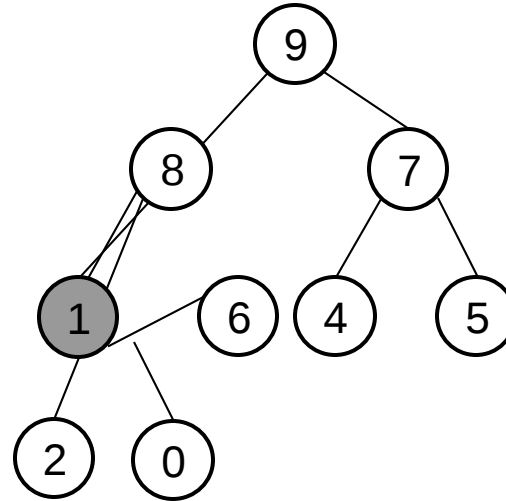


Max Heapify

Als Array

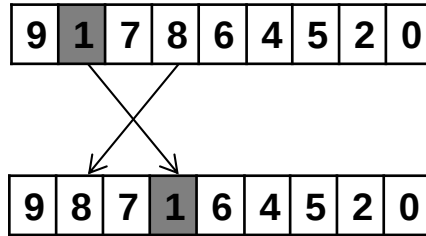
9	1	7	8	6	4	5	2	0
---	---	---	---	---	---	---	---	---

Als Baum

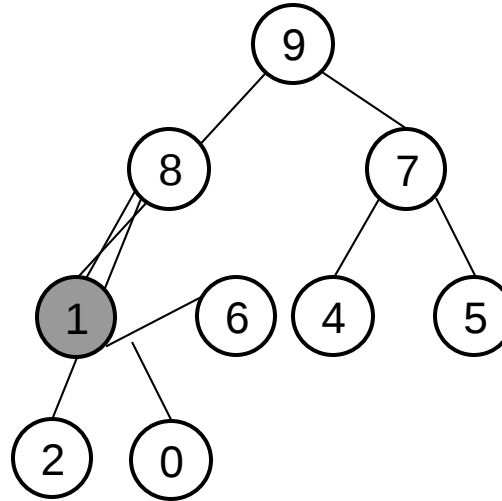


Max Heapify

Als Array

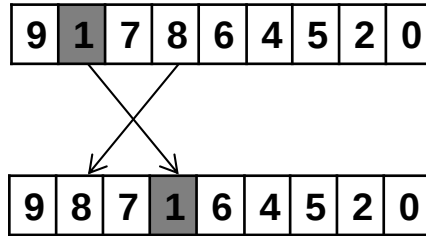


Als Baum

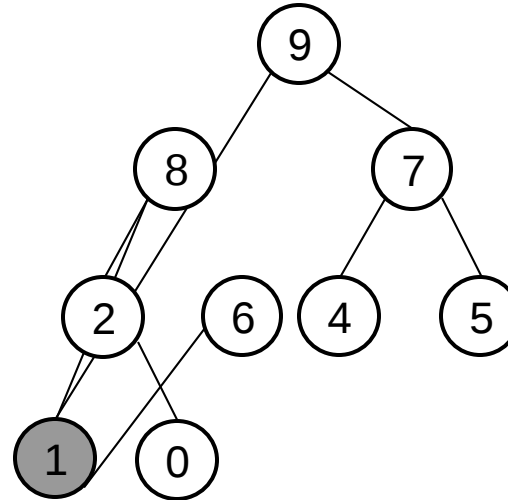


Max Heapify

Als Array

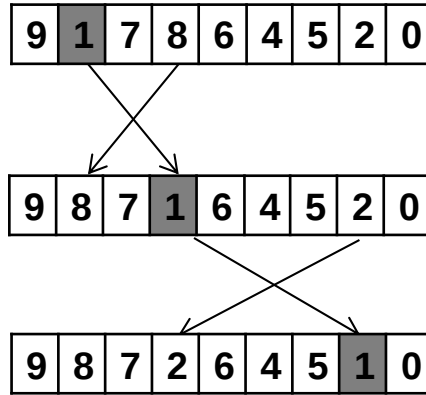


Als Baum

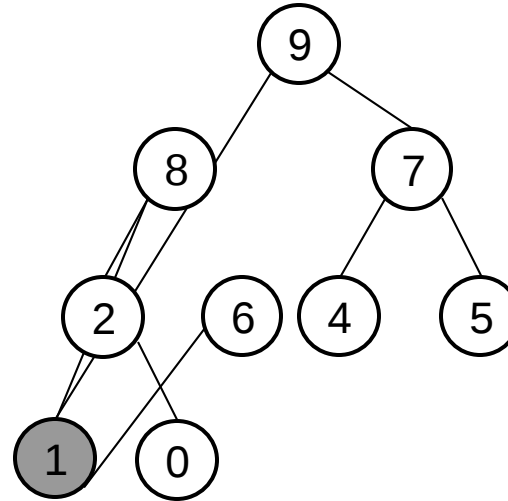


Max Heapify

Als Array

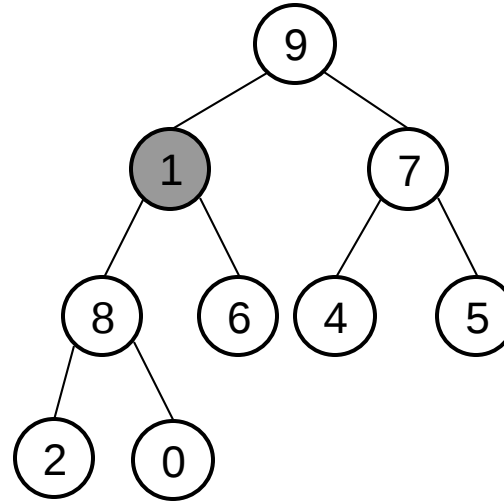


Als Baum



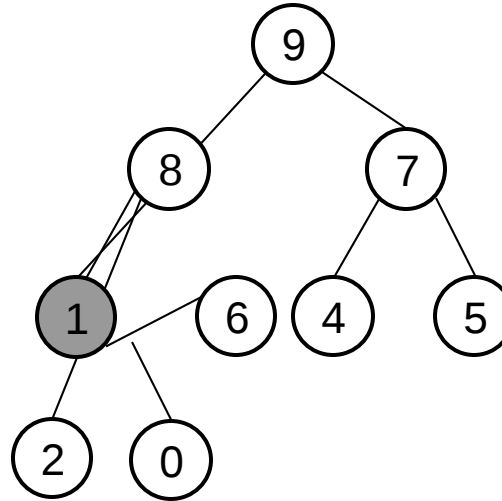
Max Heapify

Wir nehmen an, dass die Teilbäume $left(i)$ und $right(i)$ bereits Max Heaps sind.



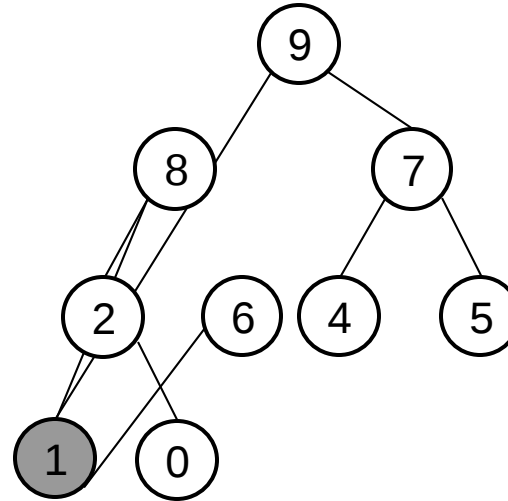
Max Heapify

Wir nehmen an, dass die Teilbäume $left(i)$ und $right(i)$ bereits Max Heaps sind.

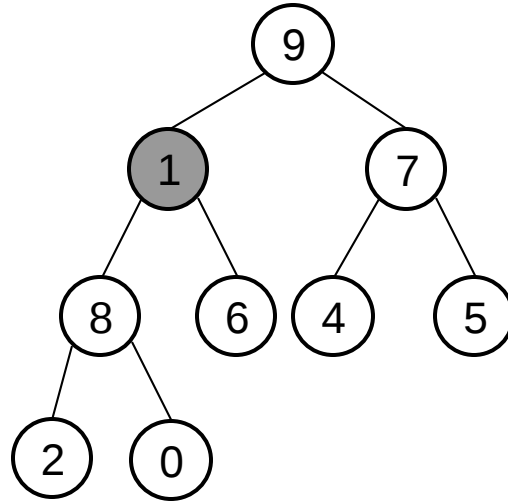


Max Heapify

Wir nehmen an, dass die Teilbäume $left(i)$ und $right(i)$ bereits Max Heaps sind.

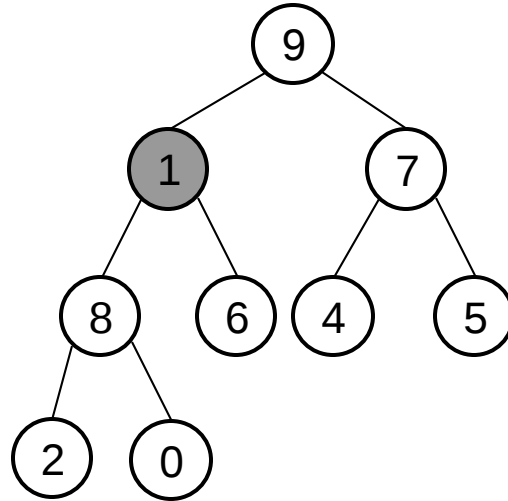


Max Heapify(A,i)



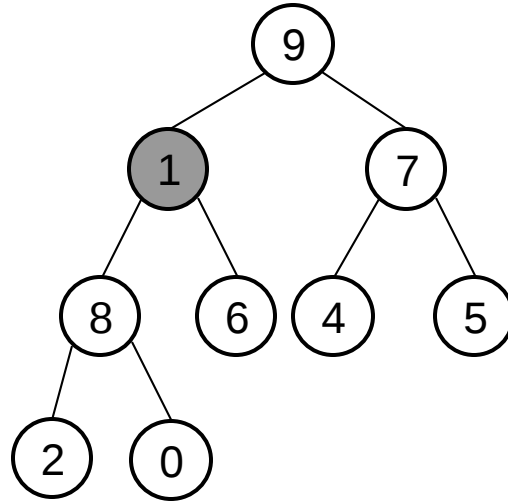
Max Heapify(A,i)

Height $h \in O(\log n)$



Max Heapify(A,i)

Height $h \in O(\log n)$
 $\rightarrow \text{max_heapify}(A, i) \in O(\log n)$



Max Heapify(A,i)

Height $h \in O(\log n)$
 $\rightarrow \text{max_heapify}(A, i) \in O(\log n)$

```
1: function MAX-HEAPIFY( $A, i$ )
2:    $\ell := \text{links}(i), r := \text{rechts}(i)$ 
3:   if  $\ell \leq \text{heap-größe}[A]$  und  $A[\ell] > A[i]$  then
4:      $\text{max} := \ell$ 
5:   else
6:      $\text{max} := i$ 
7:   if  $r \leq \text{heap-größe}[A]$  und  $A[r] > A[\text{max}]$  then
8:      $\text{max} := r$ 
9:   if  $\text{max} \neq i$  then
10:    Vertausche  $A[\text{max}]$  und  $A[i]$ 
11:    MAX-HEAPIFY( $A, \text{max}$ )
```

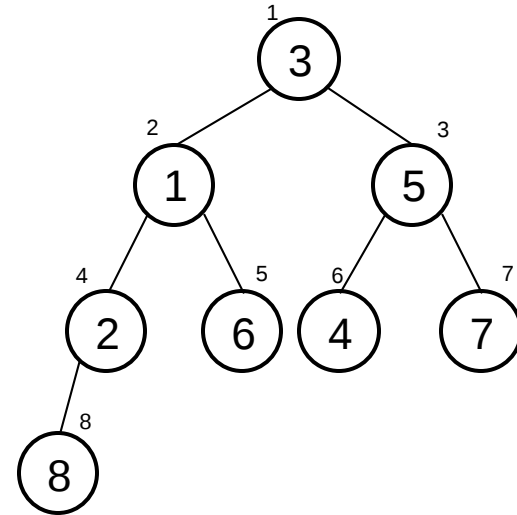
Build-Max-Heap

Build-Max-Heap

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

Build-Max-Heap

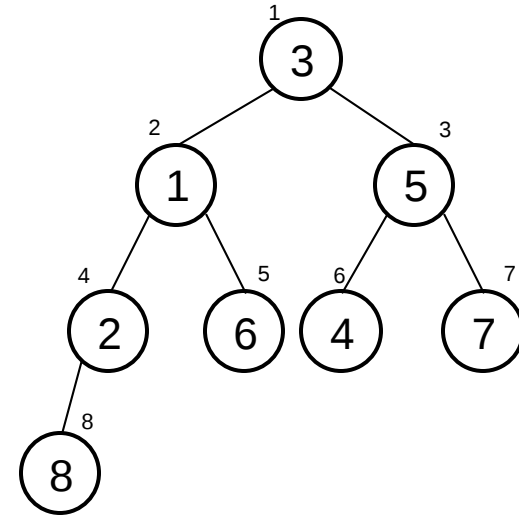
1	2	3	4	5	6	7	8
3	1	5	2	6	4	7	8



Build-Max-Heap

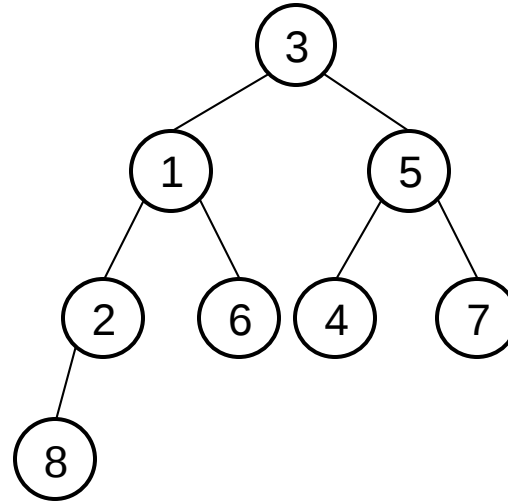
```
function BUILD-MAX-HEAP( $A$ )  
  heap-größe[ $A$ ] := länge[ $A$ ]  
  for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
    MAX-HEAPIFY( $A, i$ )
```

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



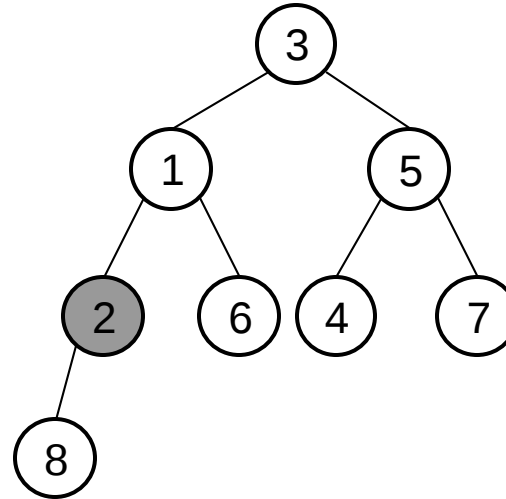
Build-Max-Heap

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



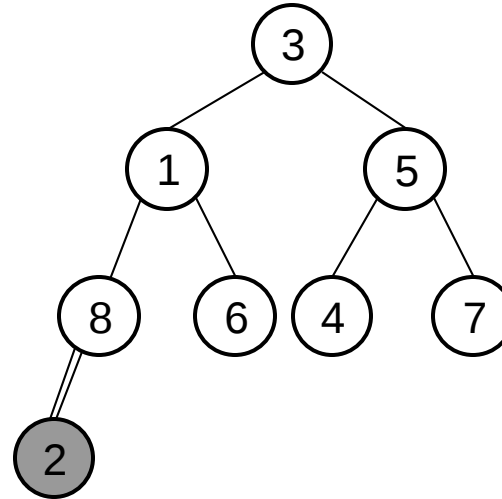
Build-Max-Heap

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

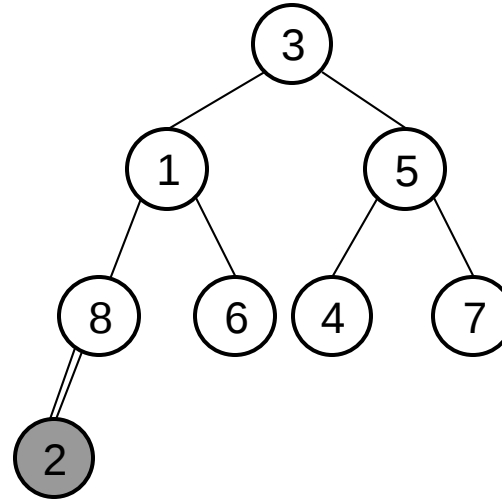
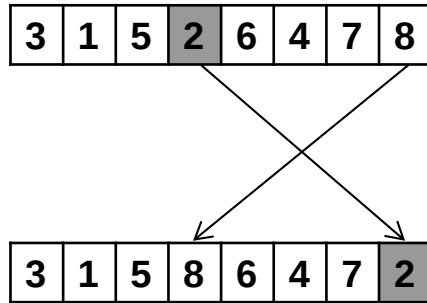


Build-Max-Heap

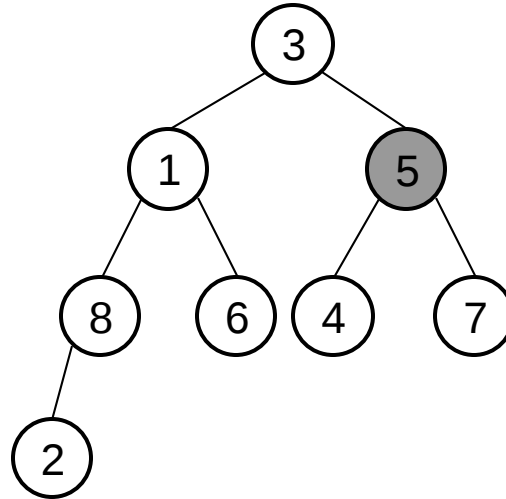
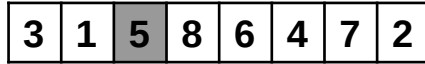
3	1	5	2	6	4	7	8
---	---	---	---	---	---	---	---



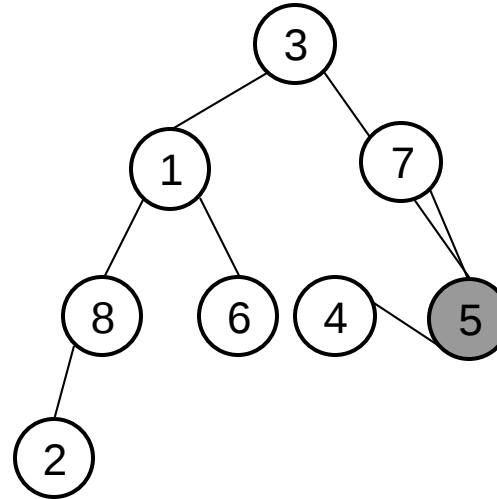
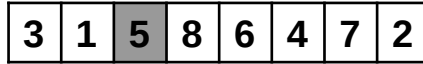
Build-Max-Heap



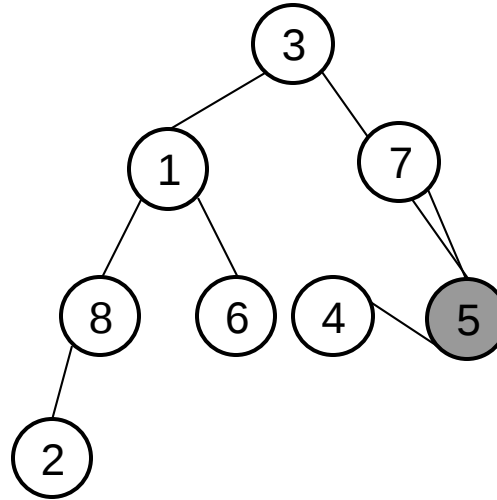
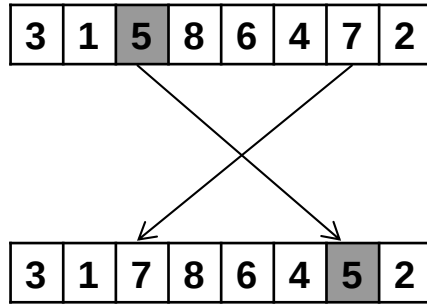
Build-Max-Heap



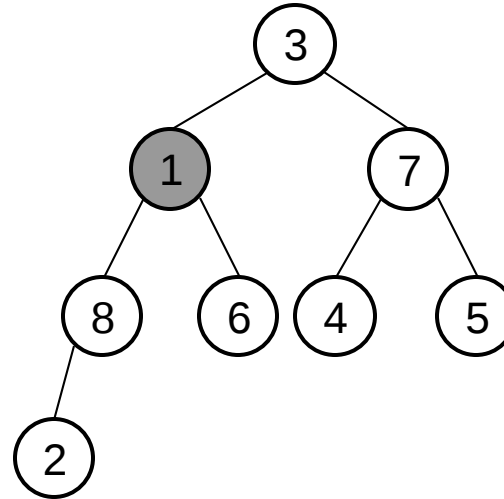
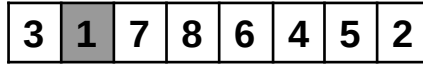
Build-Max-Heap



Build-Max-Heap

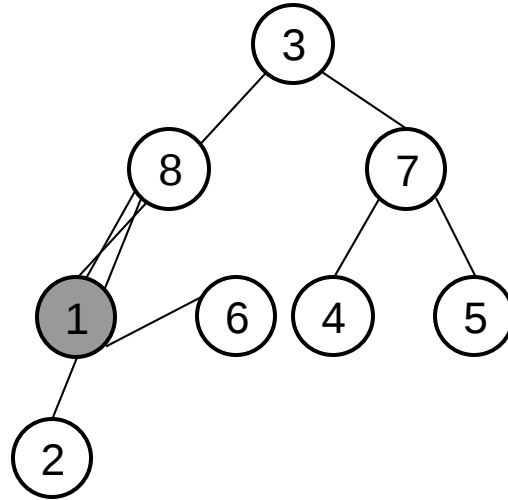


Build-Max-Heap



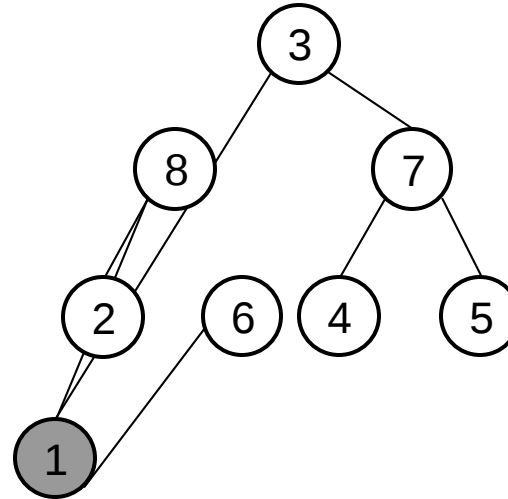
Build-Max-Heap

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

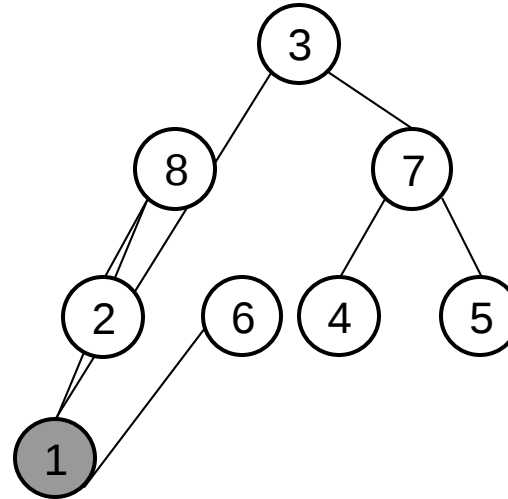
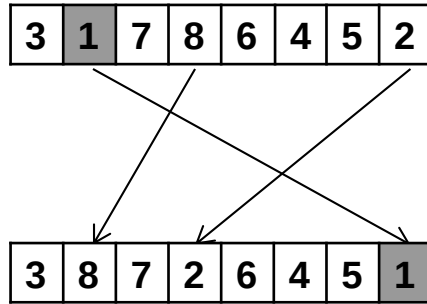


Build-Max-Heap

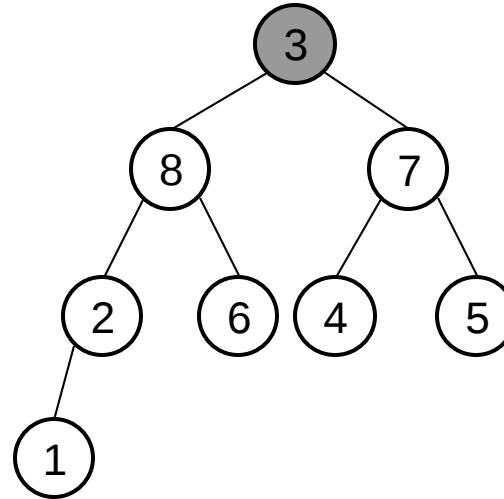
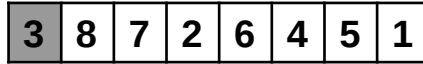
3	1	7	8	6	4	5	2
---	---	---	---	---	---	---	---



Build-Max-Heap

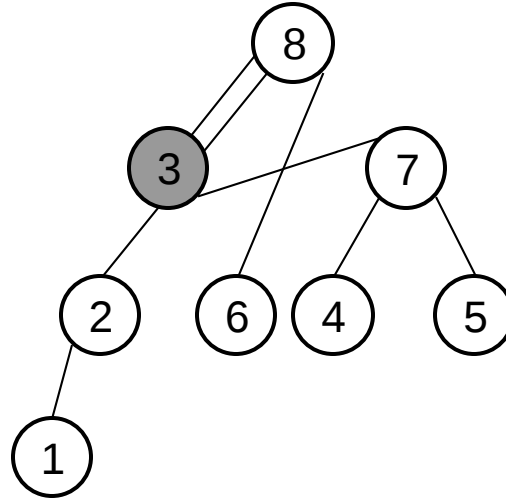


Build-Max-Heap



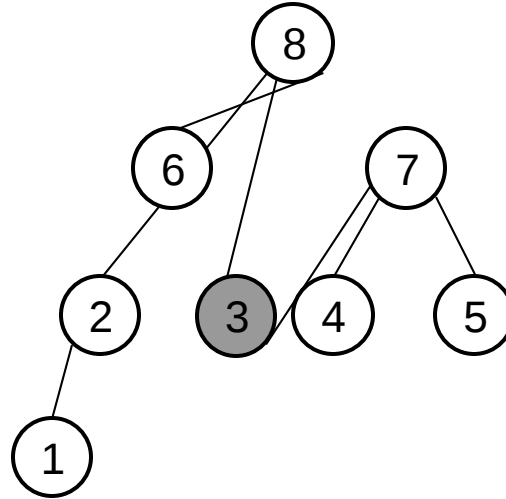
Build-Max-Heap

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

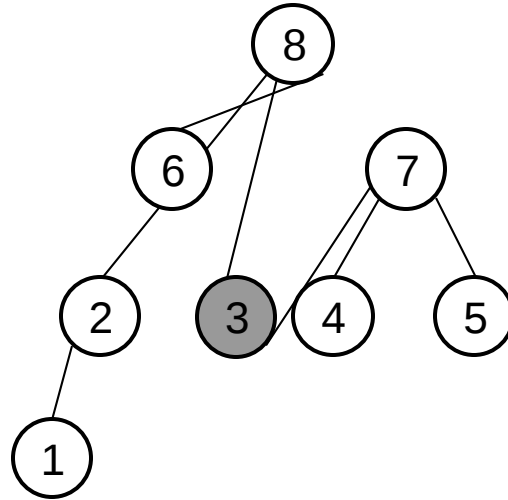
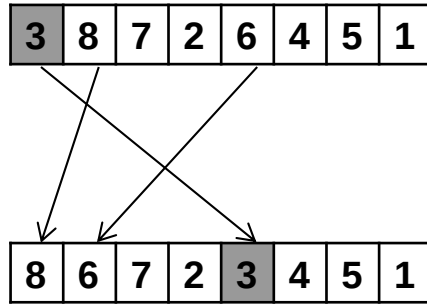


Build-Max-Heap

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

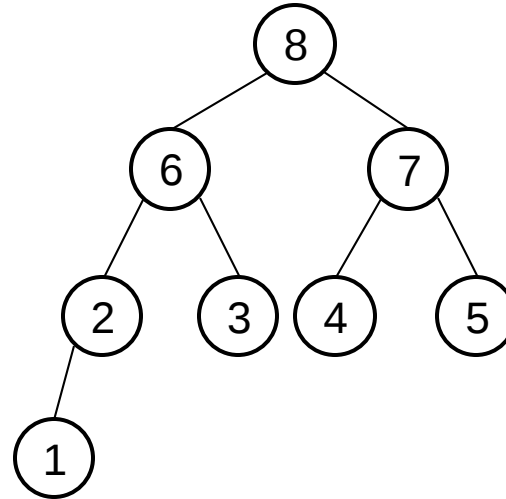


Build-Max-Heap



Build-Max-Heap

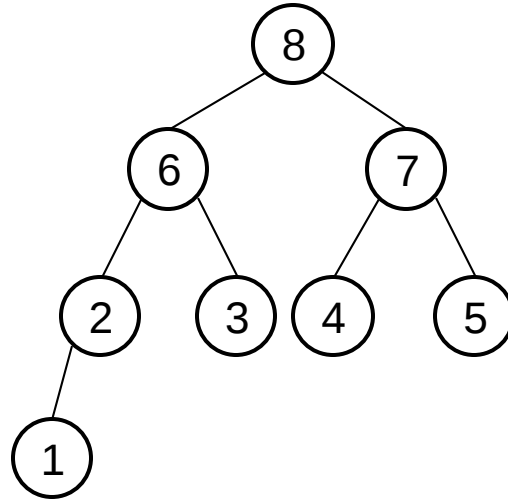
8	6	7	2	3	4	5	1
---	---	---	---	---	---	---	---



Build-Max-Heap

Fertig!

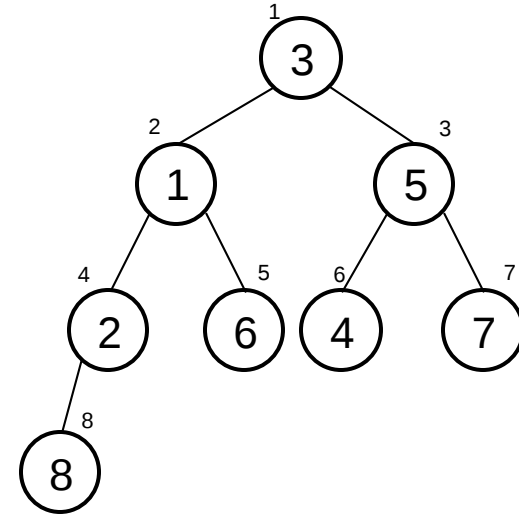
8	6	7	2	3	4	5	1
---	---	---	---	---	---	---	---



Build-Max-Heap (Laufzeit)

```
function BUILD-MAX-HEAP( $A$ )  
  heap-größe[ $A$ ] := länge[ $A$ ]  
  for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
    MAX-HEAPIFY( $A, i$ )
```

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

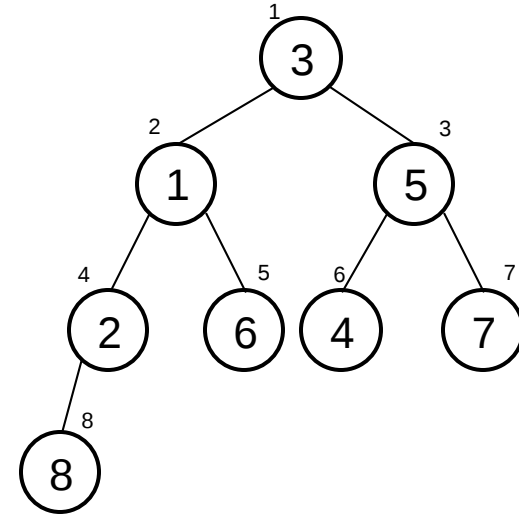


Build-Max-Heap (Laufzeit)

```
function BUILD-MAX-HEAP( $A$ )  
  heap-größe[ $A$ ] := länge[ $A$ ]  
  for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
    MAX-HEAPIFY( $A, i$ )
```

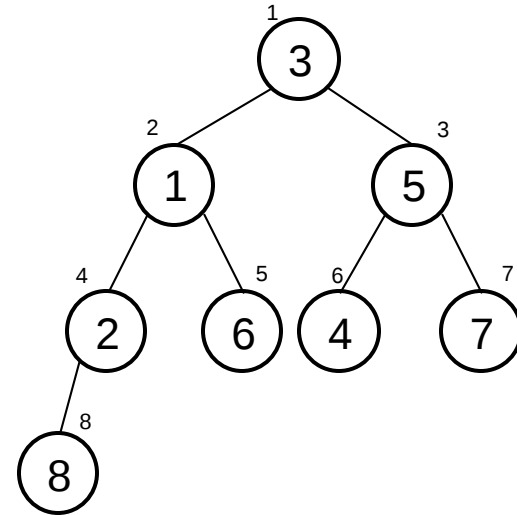
$n \cdot \text{max_heapify} \rightarrow O(n \log n)$

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



Build-Max-Heap (Laufzeit) Besser!

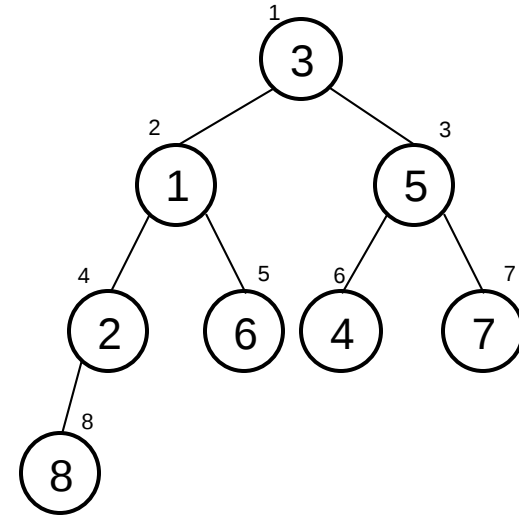
1	2	3	4	5	6	7	8
3	1	5	2	6	4	7	8



Build-Max-Heap (Laufzeit) Besser!

Für Blätter benötigt *max_heapify* $O(1)$ Operationen.

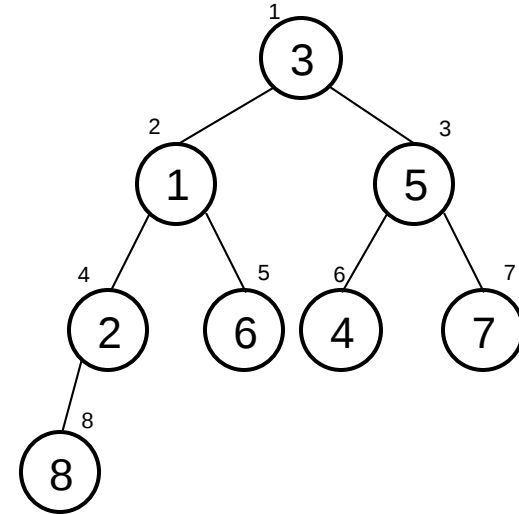
1	2	3	4	5	6	7	8
3	1	5	2	6	4	7	8



Build-Max-Heap (Laufzeit) Besser!

Für Blätter benötigt *max_heapify* $O(1)$
Operationen.
Generell: Knoten der Höhe l benötigen $O(l)$
Operationen.

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



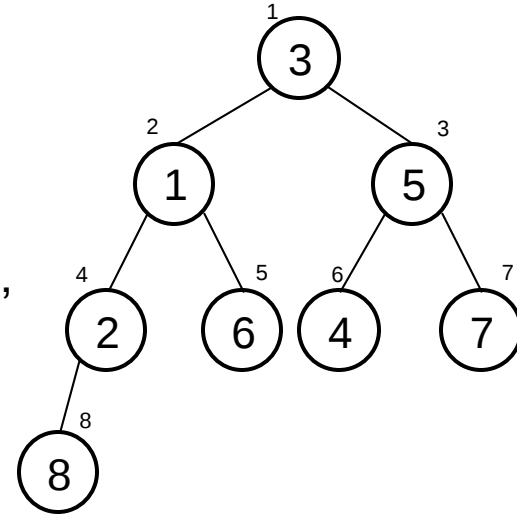
Build-Max-Heap (Laufzeit) Besser!

Für Blätter benötigt max_heapify $O(1)$ Operationen.

Generell: Knoten der Höhe l benötigen $O(l)$ Operationen.

$n/4$ Knoten der Höhe 1, $n/8$ Knoten der Höhe 2,..., und 1 Knoten der Höhe $\log n$.

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

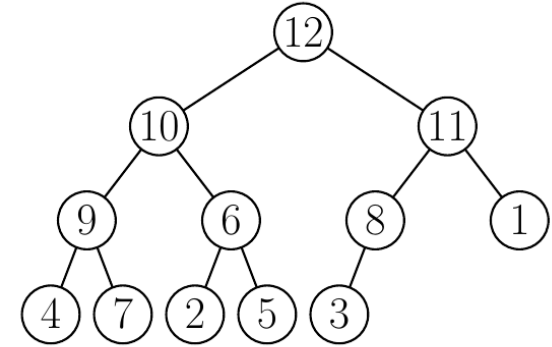


Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```

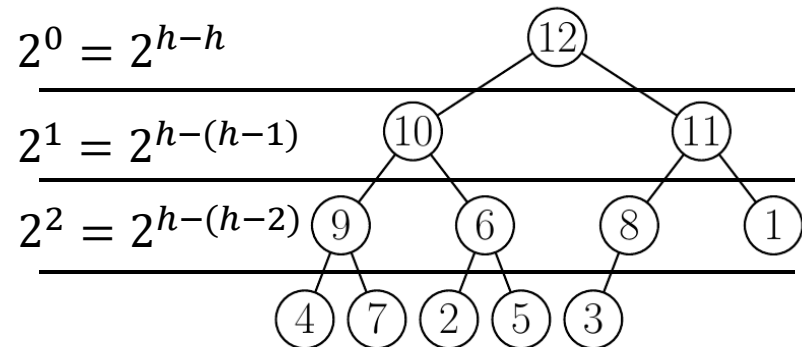
Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```



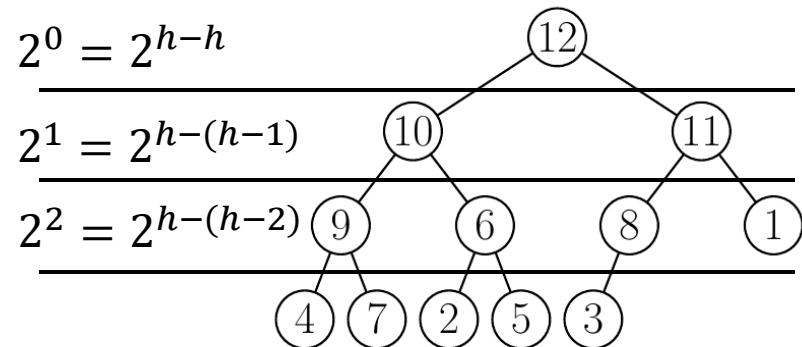
Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```



Build-Max-Heap: Laufzeit

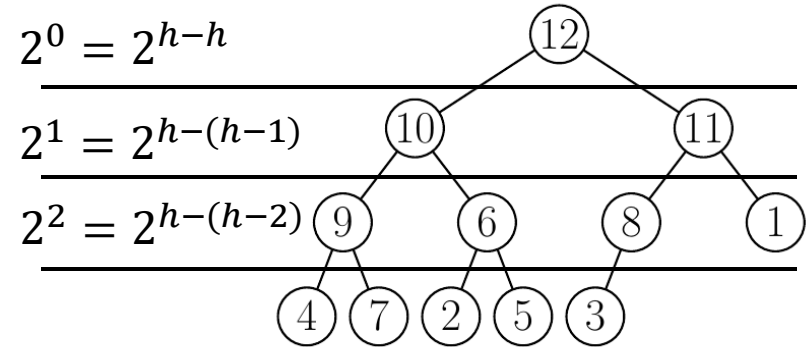
```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```



Laufzeit:

Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```

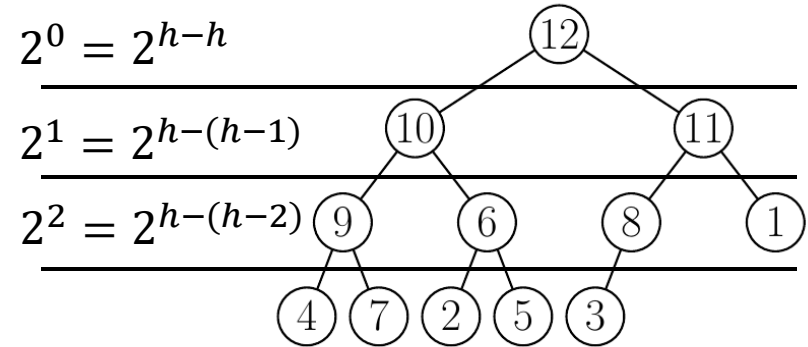


Laufzeit:

$$O\left(\sum_{j=1}^h 2^{h-j} \cdot j\right)$$

Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```

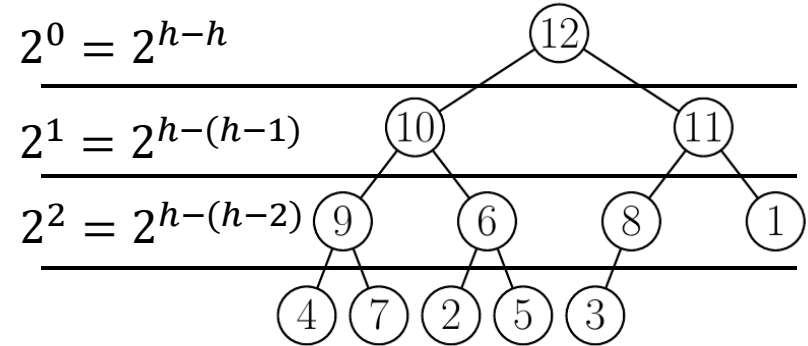


Laufzeit:

$$O\left(\sum_{j=1}^h 2^{h-j} \cdot j\right) = O\left(2^h \cdot \sum_{j=1}^h \frac{j}{2^j}\right)$$

Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```

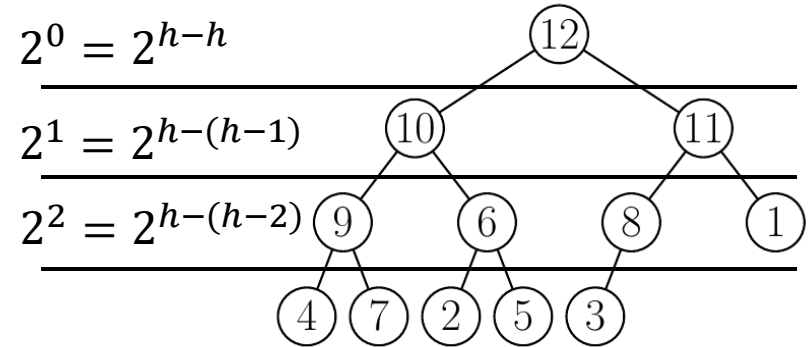


Laufzeit:

$$O\left(\sum_{j=1}^h 2^{h-j} \cdot j\right) = O\left(2^h \cdot \sum_{j=1}^h \frac{j}{2^j}\right)$$

Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```



Laufzeit:

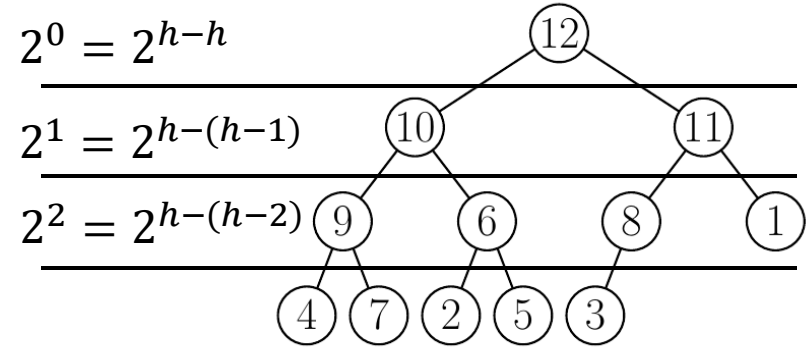
$$O\left(\sum_{j=1}^h 2^{h-j} \cdot j\right) = O\left(2^h \cdot \sum_{j=1}^h \frac{j}{2^j}\right)$$

Per Induktion:

$$\sum_{j=1}^h \frac{j}{2^j} = 2 - \frac{h+2}{2^h}$$

Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP(A)  
2:   heap-größe[A] := länge[A]  
3:   for i =  $\lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY(A, i)
```



Laufzeit:

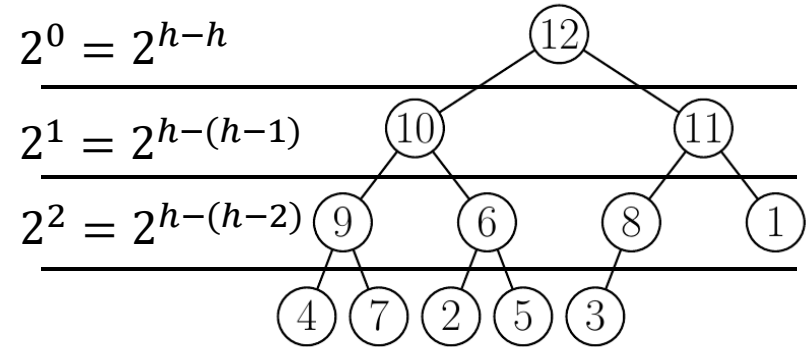
$$O\left(1 + \sum_{j=1}^h 2^{h-j} \cdot j\right) = O\left(2^h \cdot \sum_{j=1}^h \frac{j}{2^j}\right) = O(2^h)$$

Per Induktion:

$$\sum_{j=1}^h \frac{j}{2^j} = 2 - \frac{h+2}{2^h}$$

Build-Max-Heap: Laufzeit

```
1: function BUILD-MAX-HEAP( $A$ )  
2:   heap-größe[ $A$ ] := länge[ $A$ ]  
3:   for  $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$  down to 1 do  
4:     MAX-HEAPIFY( $A, i$ )
```



Laufzeit:

$$O\left(1 + \sum_{j=1}^h 2^{h-j} \cdot j\right) = O\left(2^h \cdot \sum_{j=1}^h \frac{j}{2^j}\right) = O(2^h) = O(n)$$

Per Induktion:

$$\sum_{j=1}^h \frac{j}{2^j} = 2 - \frac{h+2}{2^h}$$

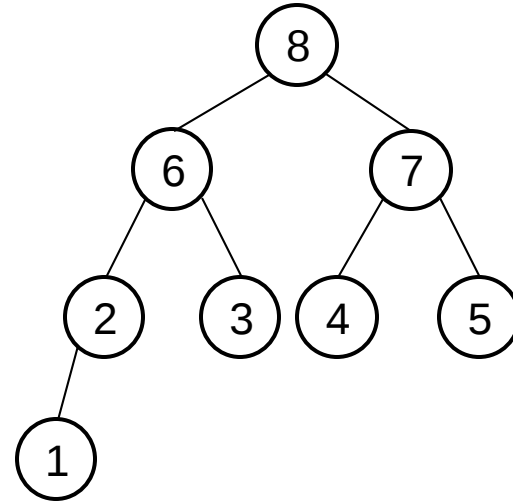
Heapsort

Heapsort

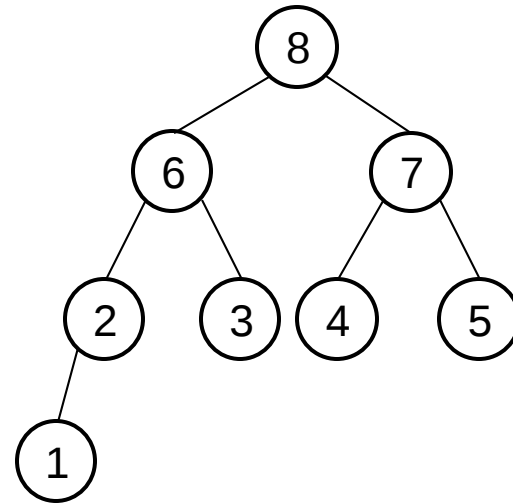
8	6	7	2	3	4	5	1
---	---	---	---	---	---	---	---

Heapsort

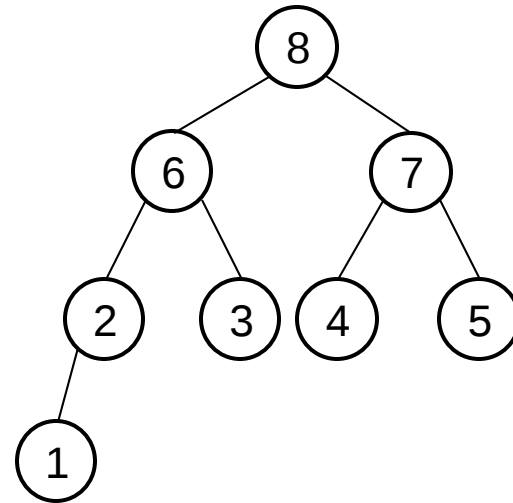
8	6	7	2	3	4	5	1
---	---	---	---	---	---	---	---



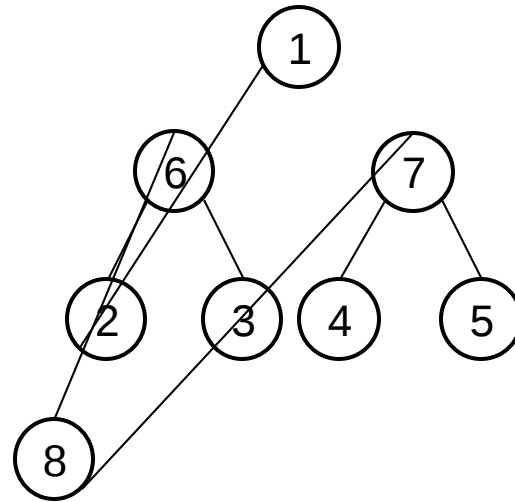
Heapsort



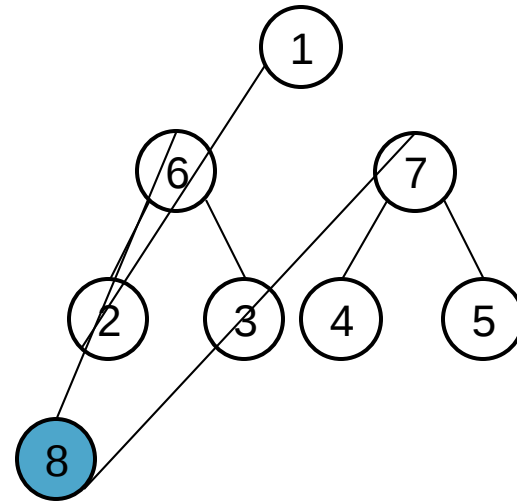
Heapsort



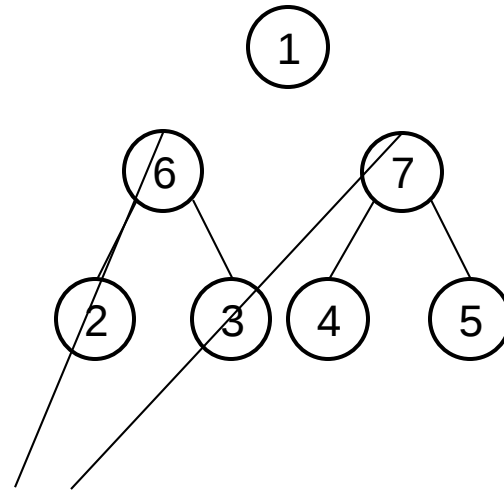
Heapsort



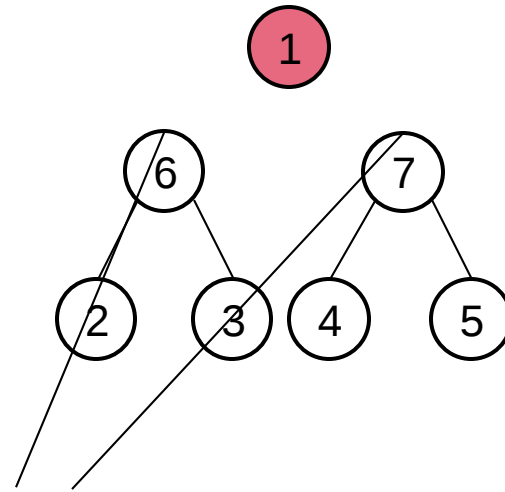
Heapsort



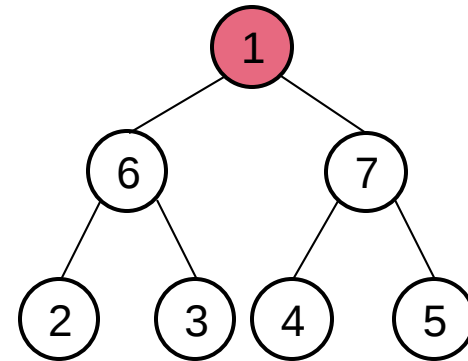
Heapsort



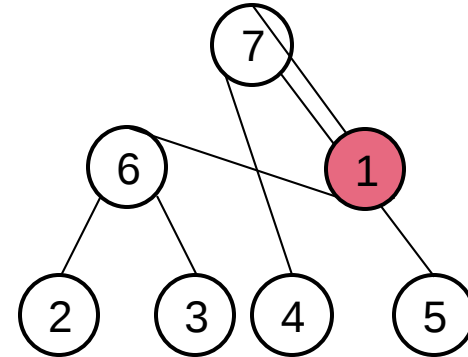
Heapsort



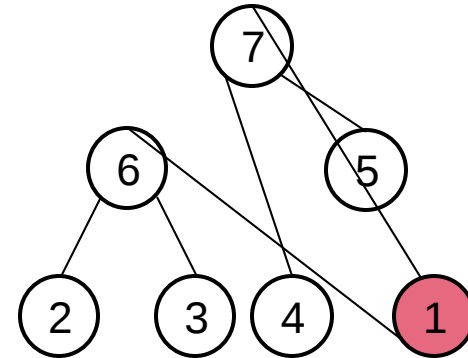
Heapsort



Heapsort

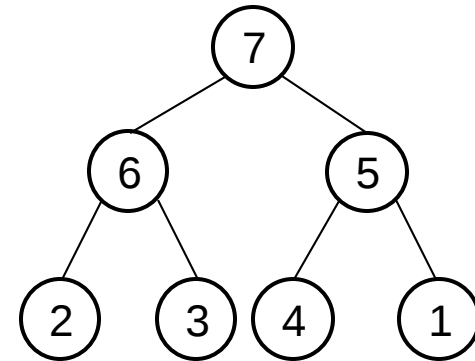


Heapsort



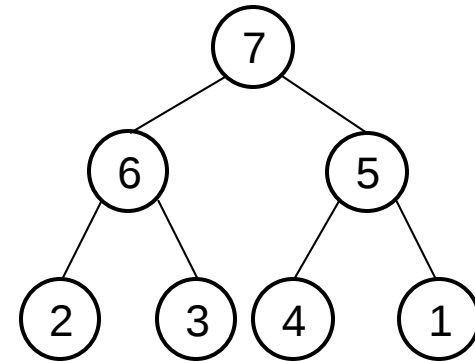
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



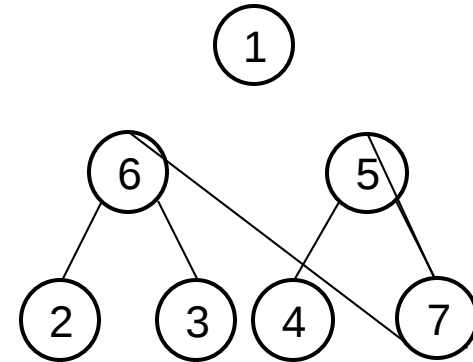
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



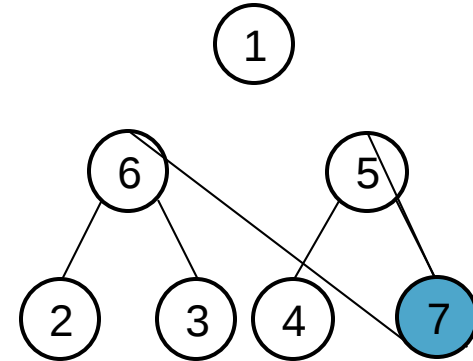
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



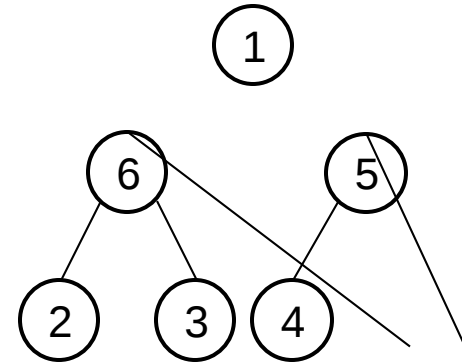
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



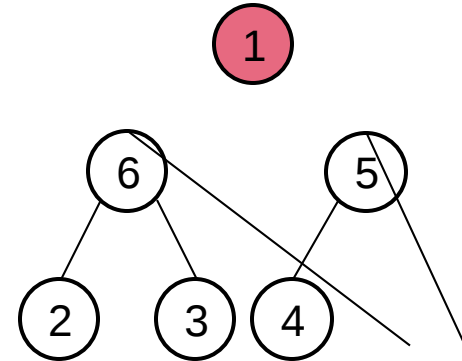
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



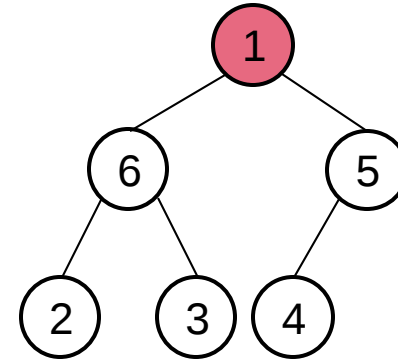
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



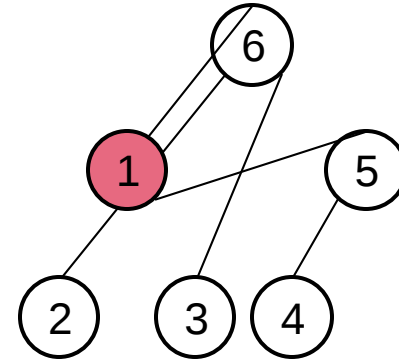
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



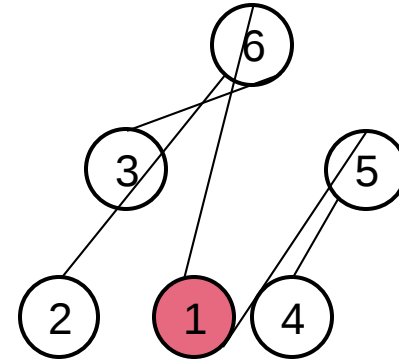
Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```



Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

Heapsort

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

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

Heapsort

```
function SORT(A)  
  BUILD-MAX-HEAP(A)  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY(A, 1)  
  end for  
end function
```

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

Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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

Heapsort

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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

Heapsort

```
function SORT(A)  
  BUILD-MAX-HEAP(A)  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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

Heapsort

```
function SORT(A)  
  BUILD-MAX-HEAP(A)  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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

Heapsort

```
function SORT(A)  
  BUILD-MAX-HEAP(A)  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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

Heapsort

```
function SORT(A)  
  BUILD-MAX-HEAP(A)  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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

Heapsort

```
function SORT(A)  
  BUILD-MAX-HEAP(A)  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    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  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

Heapsort – Laufzeit

Build-Max-Heap:

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

Heapsort – Laufzeit

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

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

1. Vertauschen:

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

1. Vertauschen: $O(1)$

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

1. Vertauschen: $O(1)$
2. Verkleinern:

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

1. Vertauschen: $O(1)$
2. Verkleinern: $O(1)$

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

1. Vertauschen: $O(1)$
2. Verkleinern: $O(1)$
3. Max-Heapify:

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

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

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

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

Für n Iterationen ist das

Heapsort – Laufzeit

```
function SORT( $A$ )  
  BUILD-MAX-HEAP( $A$ )  
  for  $i \leftarrow \text{length}(A)$  downto 2 do  
    vertausche  $A[1]$  und  $A[i]$   
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$   
    MAX-HEAPIFY( $A, 1$ )  
  end for  
end function
```

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

In der For-Schleife

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

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

Heapsort – Korrektheit

Heapsort ist ein Sortierverfahren.

Beweis:

Wir zeigen: Nach der i -ten Iteration besitzen wir wieder einen Max-Heap auf den restlichen Feldern

→ Wir können das nächstgrößere Element an die richtige Stelle sortieren.

- Zu Beginn besitzen wir einen Max-Heap.
 - das größte Element ist an der ersten Stelle.
- Angenommen, wir haben zu Beginn der Iteration i einen Max-Heap auf den aktiven $n - i + 1$ Feldern.
- Nach dem Tauschen des i -größten Elements nach hinten und nach Verkleinern des Arrays:
 - U.U. kein Max-Heap vorhanden. (vor Max-Heapify-Aufruf)
- Aber: Jeder Knoten außer Wurzel erfüllt Max-Heap Eigenschaft.
- Also: Max-Heapify auf Wurzelknoten reicht aus, um einen Max-Heap wiederherzustellen!
- → Damit haben wir wieder einen Max-Heap auf den restlichen Feldern.
- *Wir tauschen also in jeder Iteration das richtige Element nach hinten!*

```
function SORT(A)
  BUILD-MAX-HEAP(A)
  for  $i \leftarrow \text{length}(A)$  downto 2 do
    vertausche  $A[1]$  und  $A[i]$ 
     $\text{heap-größe}[A] \leftarrow \text{heap-größe}[A] - 1$ 
    MAX-HEAPIFY( $A, 1$ )
  end for
end function
```

Fragen?

