# Objektorientierte Programmierung

## Sortieren (Teil 2)

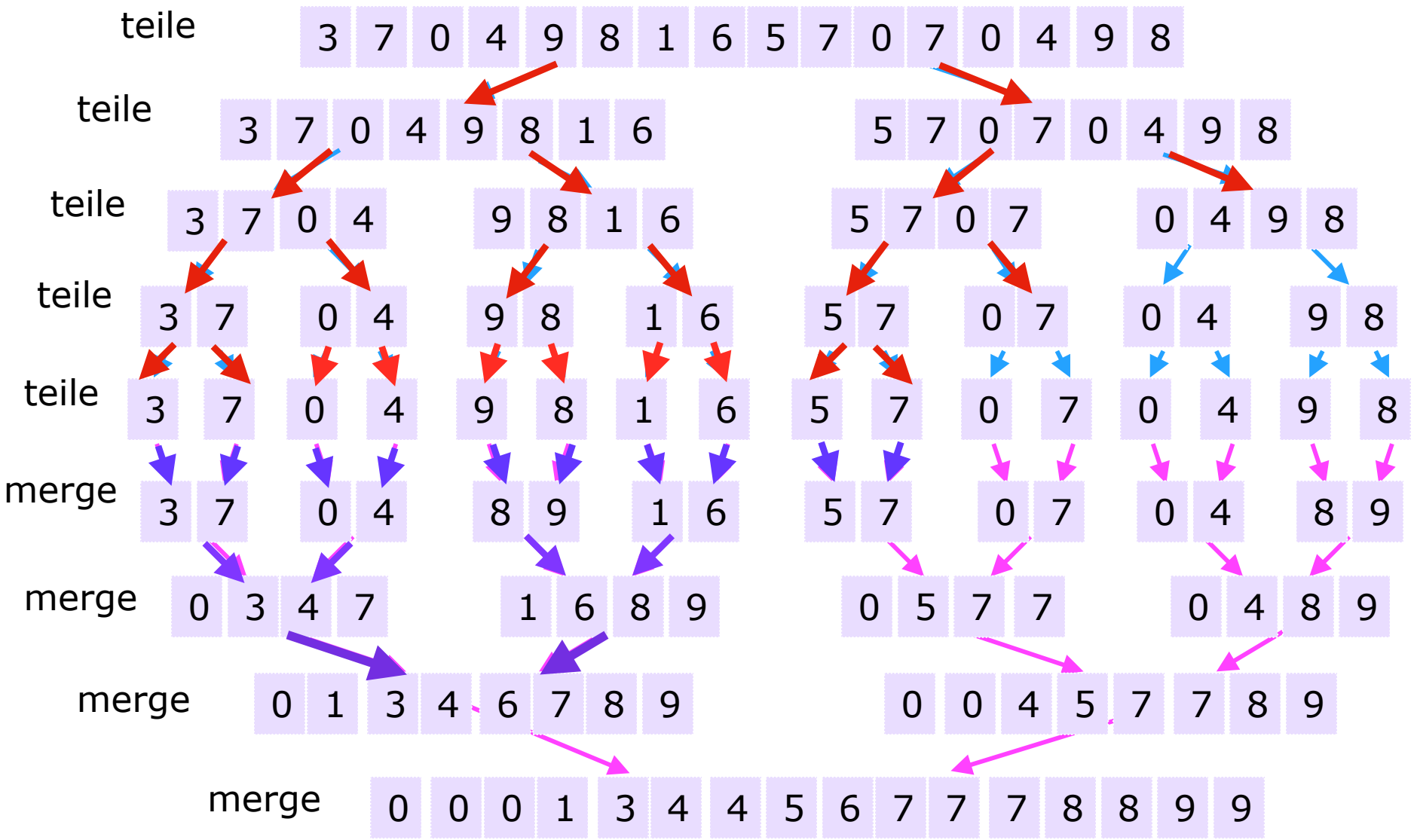
Imperativ!

SoSe 2020

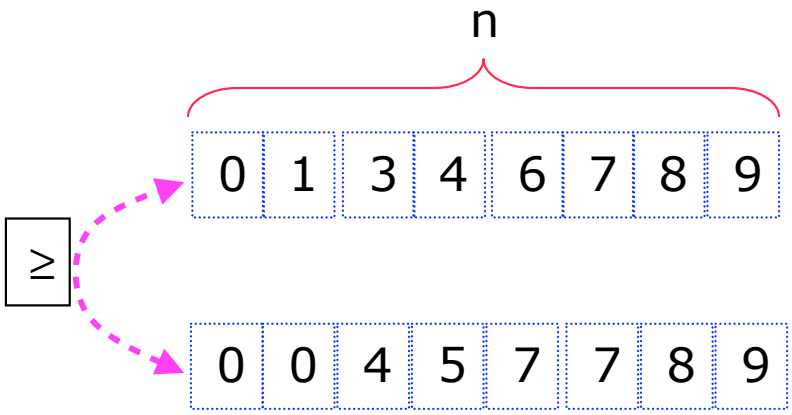
Prof. Dr. Margarita Esponda

# Merge-Sort-Algorithmus





# Merge-Algorithmus

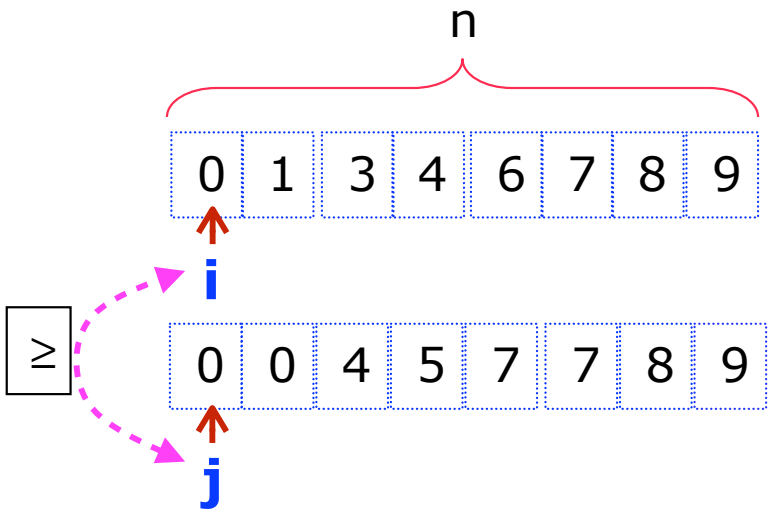


# Merge-Algorithmus

Hilfsarray

**res**

[]

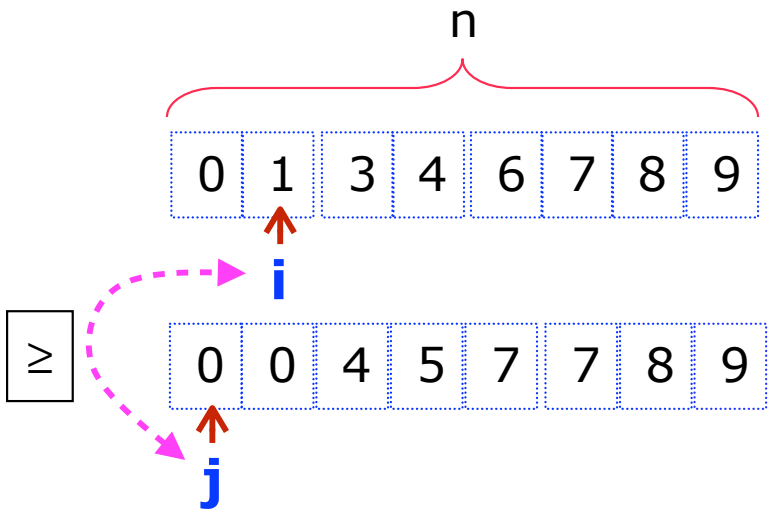


Freie Universität Berlin

# Merge-Algorithmus

Hilfsarray

**res**

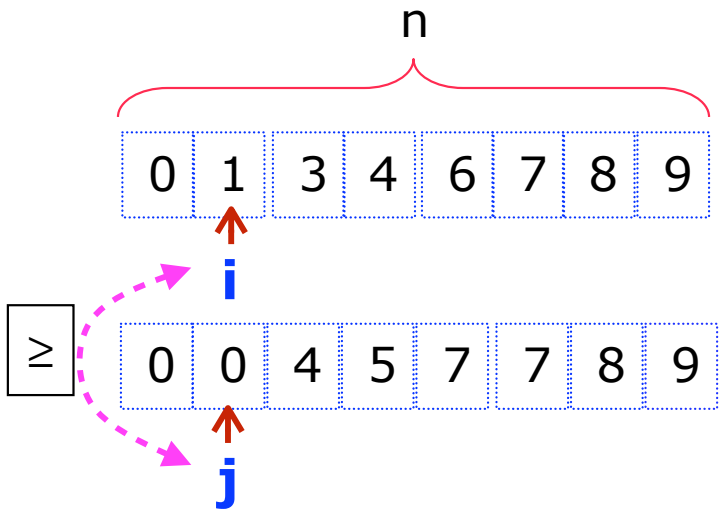
Freie Universität Berlin

# Merge-Algorithmus

Hilfsarray

**res**

| 0 | 0 |
|---|---|

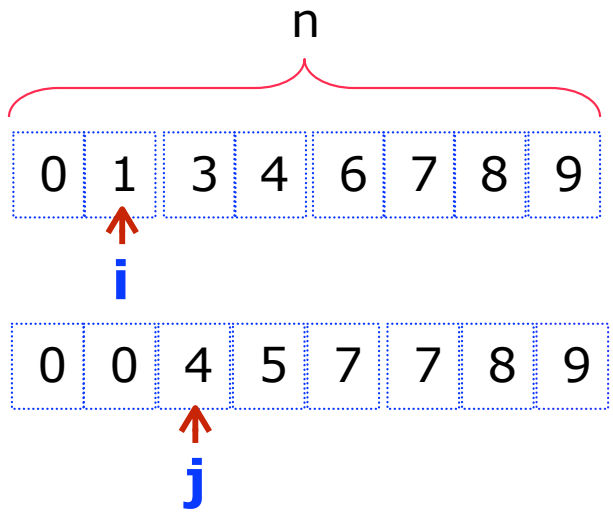


# Merge-Algorithmus

Hilfsarray

**res**

| 0 | 0 | 0 |
|---|---|---|

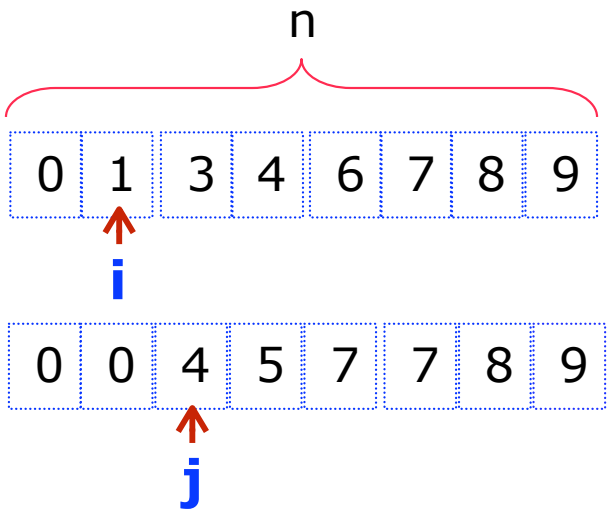


# Merge-Algorithmus

Hilfsarray

**res**

| 0 | 0 | 0 | 1 |
|---|---|---|---|

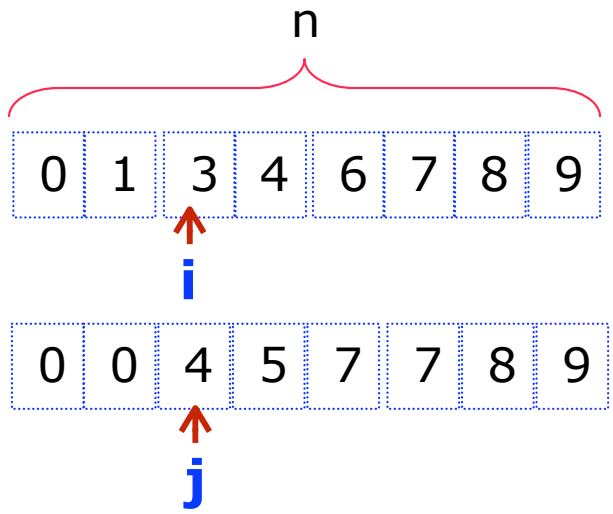


# Merge-Algorithmus

Hilfsarray

**res**

| 0 | 0 | 0 | 1 | 3 |
|---|---|---|---|---|

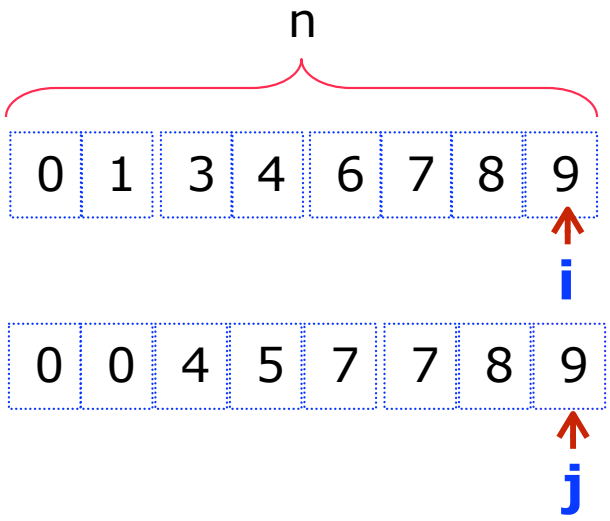


# Merge-Algorithmus

Hilfsarray

**res**

| 0 | 0 | 0 | 1 | 3 | 4 | 4 | 5 | 6 | 7 | 7 | 7 | 8 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

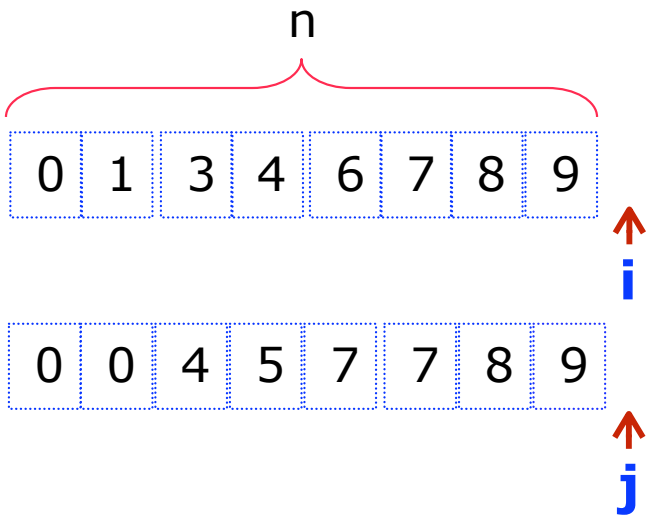


# Merge-Algorithmus

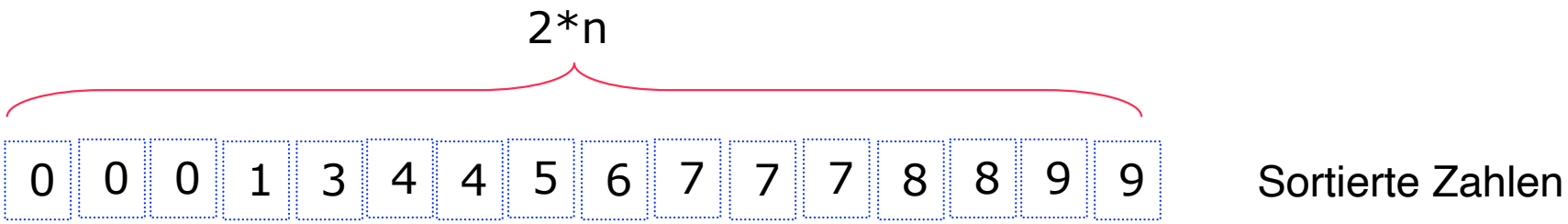
Hilfsarray

**res**

| 0 | 0 | 0 | 1 | 3 | 4 | 4 | 5 | 6 | 7 | 7 | 7 | 8 | 8 | 9 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|



# Merge-Algorithmus




Wir hatten ursprünglich zwei sortierte Mengen mit Länge **n.**

Nach jedem Vergleich wird eine Zahl sortiert,

d.h. im schlimmsten Fall haben wir **2*n - 1** Vergleiche.

$$T(n) = 2n - 1 = O(n)$$

# Merge-Sort-Algorithmus

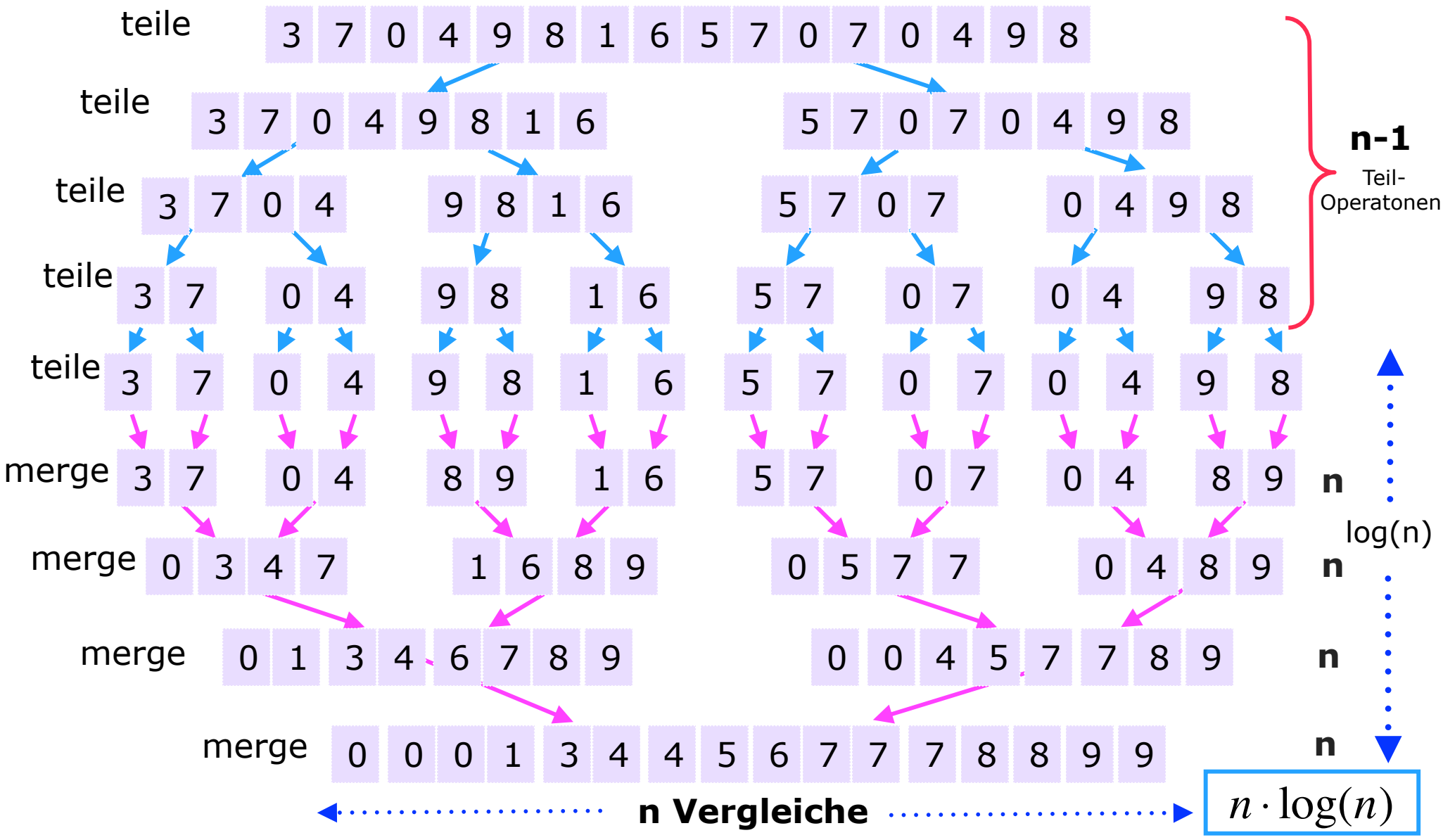
```
def merge(low, high):
        res = []
        i, j = 0, 0
        while i<len(low) and j<len(high):
                if low[i] <= high[j]:
                        res.append(low[i])
                        i = i+1
                else:
                        res.append(high[j])
                        j = j+1
        res = res + low[i:]
        res = res + high[j:]
        return res
```

Speicherverbrauch?

Rekursiv

```
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A) // 2
        return merge( mergesort(A[:m]), mergesort(A[m:]) )
```

# Merge-Sort-Algorithmus



# Merge-Sort-Algorithmus

Eine Teilung kostet $c_1$

Drei Vergleiche kosten $c_2$

$$T(n) = \underbrace{c_1(n-1)}_{\text{Teiloperationen}} + \underbrace{c_2 \cdot n \cdot \log(n)}_{\text{Vergleiche}}$$

$$T(n) = O(n \cdot \log(n))$$

# Merge-Sort-Algorithmus

- **1945** von **John von Neumann** entwickelt
- Teile und Herrsche Algorithmus
- **stabil**
- **nicht in-place**
- **O(n) zusätzlichen Speicherbedarf!**
- Komplexität
  - **O(n·log(n))** im schlimmsten Fall
  - **O(n·log(n))** im besten Fall

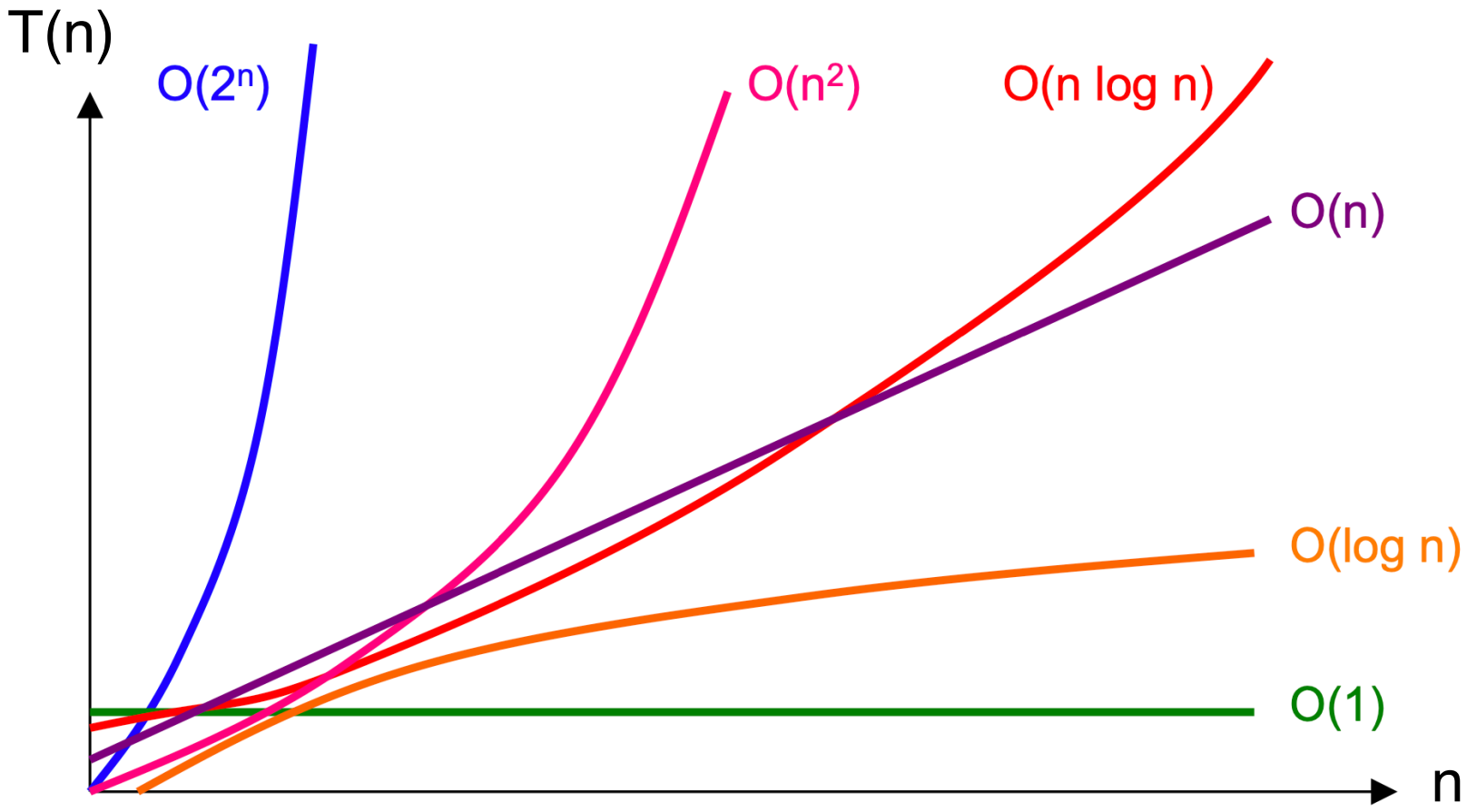
# Insert-Sort vs. Merge-Sort

Vergleiche

| n | Insert-Sort | Merge-Sort |
|---|---|---|
| **8** | 36 | 24 |
| **16** | 136 | 64 |
| **32** | 528 | 160 |
| **$2^{10}$** | 524 800 | 10 240 |
| **$2^{20}$** | 549 756 338 176 | 20 971 520 |

Freie Universität Berlin

# Wachstumsfunktionen



# Shellsort

Shellsort ist eines der am längsten (1959) bekannten Sortierverfahren.

Der Urheber ist Donald .L. Shell.

Die Idee des Verfahrens ist es, die Daten als zweidimensionales Feld zu arrangieren und spaltenweise zu sortieren.

Nach dieser Grobsortierung werden die Daten als schmaleres zweidimensionales Feld wieder angeordnet und wiederum spaltenweise sortiert.

Das Ganze wiederholt sich, bis zum Schluss das Feld nur noch aus einer Spalte besteht.

Die Spalten werden alle parallel mit Hilfe des Insertsort-Algorithmus sortiert.

# Shellsort

Sei

9 0 2 2 6 3 7 | 1 9 0 2 6 3 7 | 4 8 5 6 3 7

die zu sortierende Datenfolge

# Shellsort

9 0 2 2 6 3 7

1 9 0 2 6 3 7

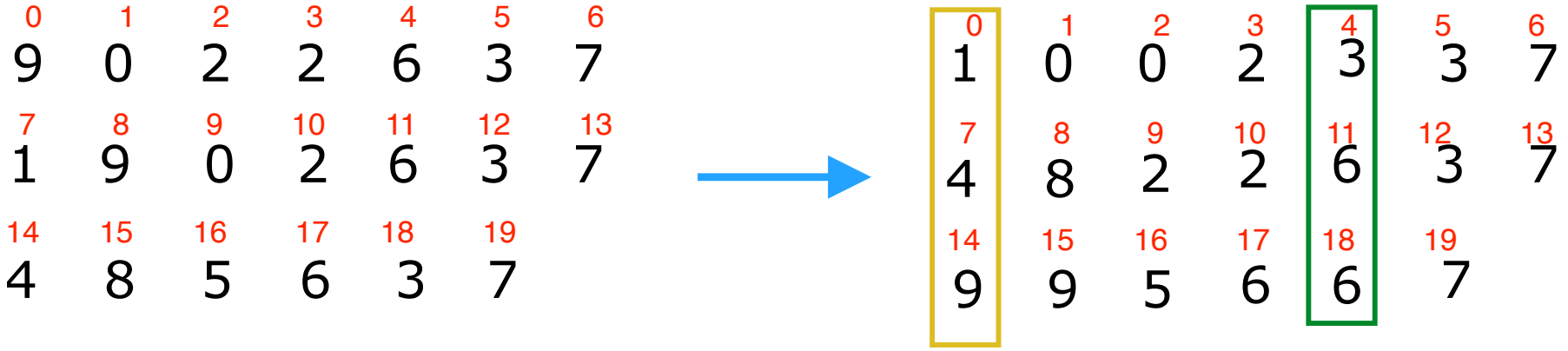
4 8 5 6 3 7

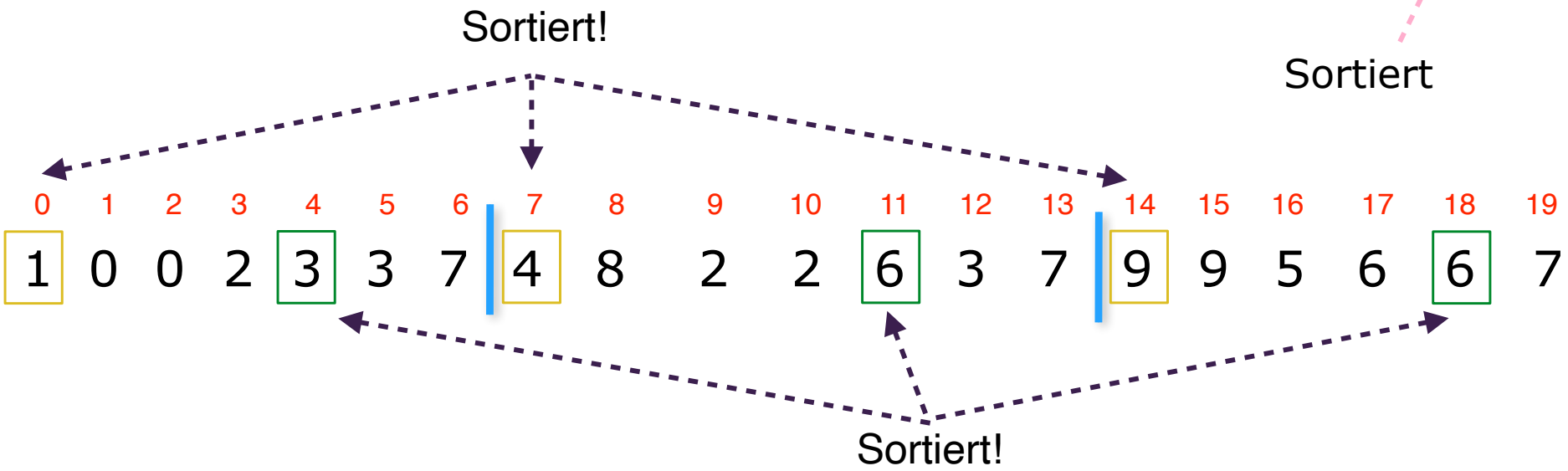
# Shellsort

| | | | | | | |
|---|---|---|---|---|---|---|
| 9 | 0 | 2 | 2 | 6 | 3 | 7 |
| 1 | 9 | 0 | 2 | 6 | 3 | 7 |
| 4 | 8 | 5 | 6 | 3 | 7 | |

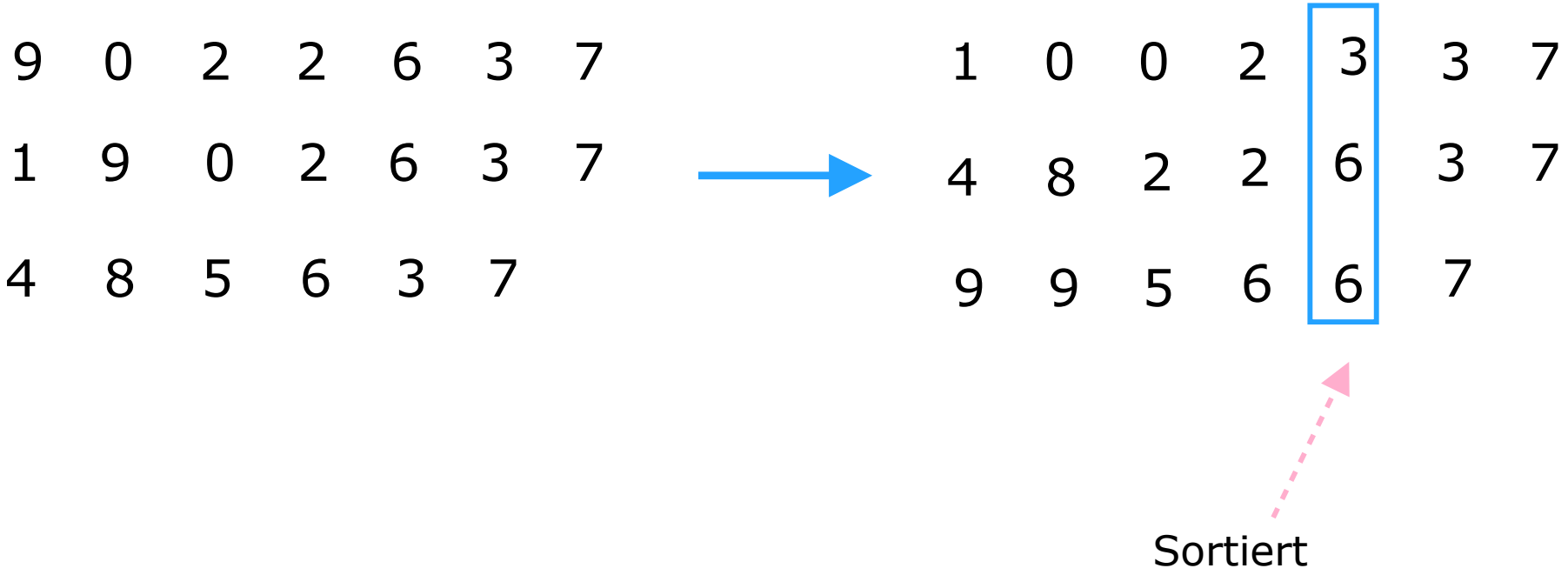
Die Spalten werden sortiert

# Shellsort


0 1 2 3 4 5 6
9 0 2 2 6 3 7
7 8 9 10 11 12 13
1 9 0 2 6 3 7
14 15 16 17 18 19
4 8 5 6 3 7
0 1 2 3 4 5 6
1 0 0 2 3 3 7
7 8 9 10 11 12 13
4 8 2 2 6 3 7
14 15 16 17 18 19
9 9 5 6 6 7


Sortiert!
Sortiert
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1 0 0 2 3 3 7 4 8 2 2 6 3 7 9 9 5 6 6 7
Sortiert!

# Shellsort


9 0 2 2 6 3 7
1 9 0 2 6 3 7
4 8 5 6 3 7
1 0 0 2 3 3 7
4 8 2 2 6 3 7
9 9 5 6 6 7
Sortiert


1 0 0 2 3 3 7 4 8 2 2 6 3 7 9 9 5 6 6 7

# Shellsort

1 0 0

2 3 3

7 4 8

2 2 6

3 7 9

9 5 6

6 7

# Shellsort

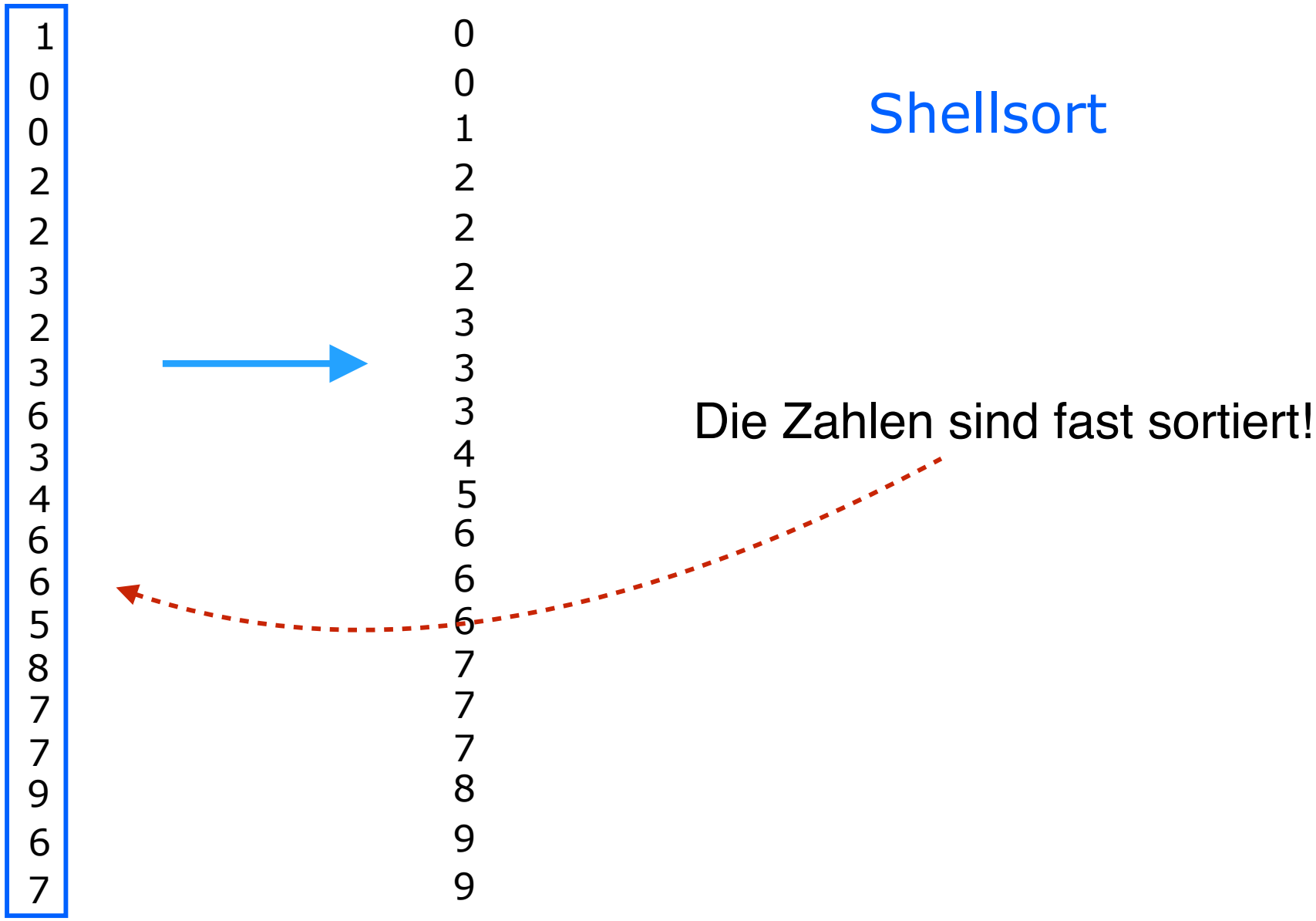
| | | |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 3 | 3 |
| 7 | 4 | 8 |
| 2 | 2 | 6 |
| 3 | 7 | 9 |
| 9 | 5 | 6 |
| 6 | 7 | |

Die Spalten werden sortiert

Freie Universität Berlin

# Shellsort

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | | 1 | 0 | 0 |
| 2 | 3 | 3 | | 2 | 2 | 3 |
| 7 | 4 | 8 | | 2 | 3 | 6 |
| 2 | 2 | 6 | → | 3 | 4 | 6 |
| 3 | 7 | 9 | | 6 | 5 | 8 |
| 9 | 5 | 6 | | 7 | 7 | 9 |
| 6 | 7 | | | 9 | 7 | |


1 0 0 2 2 3 2 3 6 3 4 6 6 5 8 7 7 9 6 7
0 0 1 2 2 2 3 3 3 4 5 6 6 6 7 7 7 8 9 9
Shellsort
Die Zahlen sind fast sortiert!

# Shellsort

```
def shellSort(A):
        seg_size = len(A)//2
        while seg_size > 0:
            for start_i in range(seg_size):
                jump_InsertSort(A, start_i, seg_size)
            seg_size = seg_size // 2
```

```
def jump_InsertSort(A, start, step):
        for i in range(start+step, len(A), step):
            value = A[i]
            j = i
            while  j >= step  and  A[j-step] > value:
                A[j] = A[j-step]
                j = j - step
            A[j] = value
```

# Shellsort

Wenn die Feldbreiten geschickt gewählt werden, reichen jedes mal wenige Sortierschritte aus, um die Daten spaltenweise zu sortieren.

Es gibt noch kein mathematisches Modell, um für beliebige Datenmengen zu entscheiden, welche die optimale Segmentierungssequenz ist.

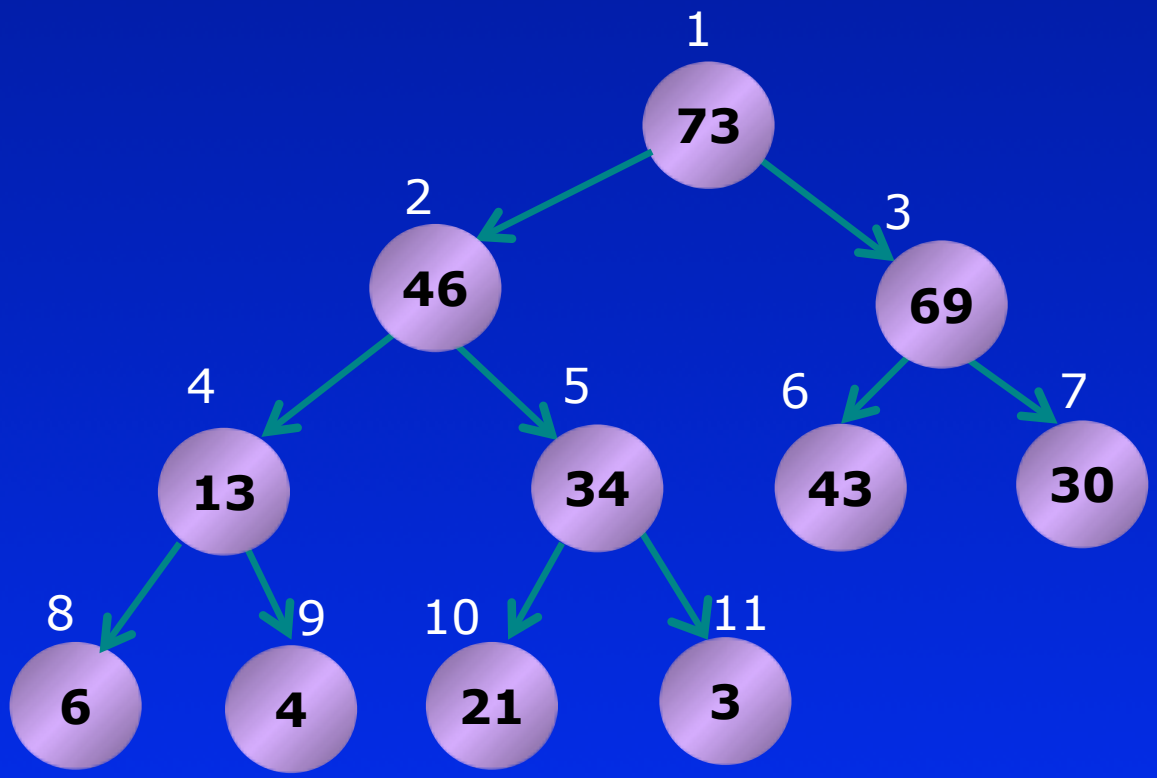
Eigenschaften:

- **nicht stabil**
- **die Komplexität hängt von der Segmentierung ab**

| | | |
|---|---|---|
| Mersenne-Zahlen | $1, 3, 15, \ldots, 2^k-1$ | $O(n^{1,5})$ |

**magic = [1391376, 463792, 198768, 86961, 33936, 13776, 4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1]**

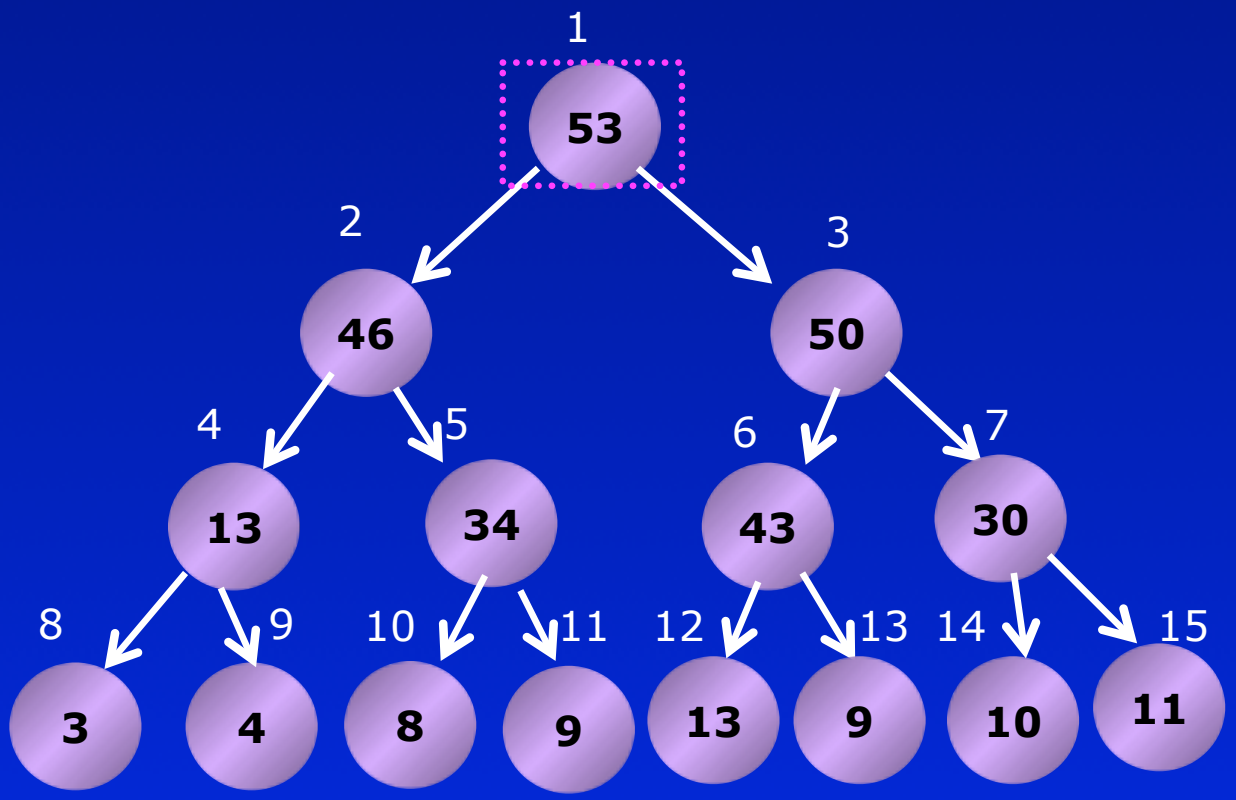
# Heapsort

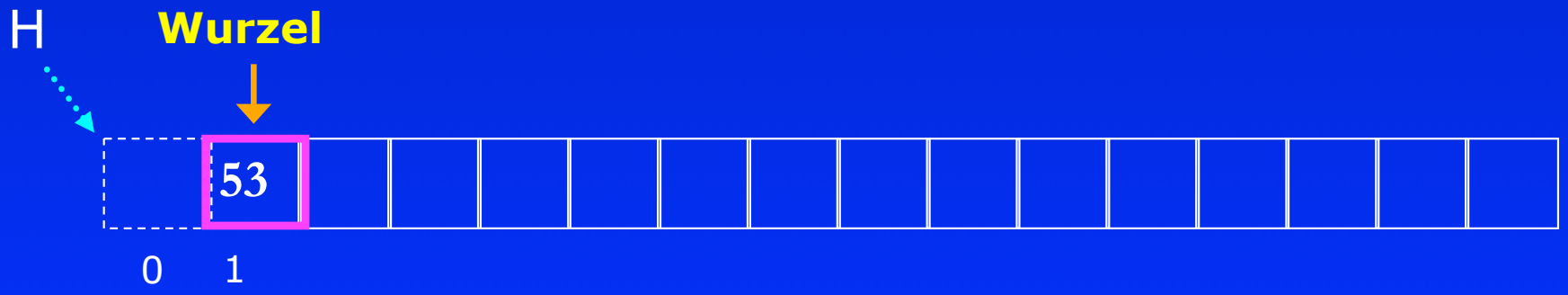


# Heapsort

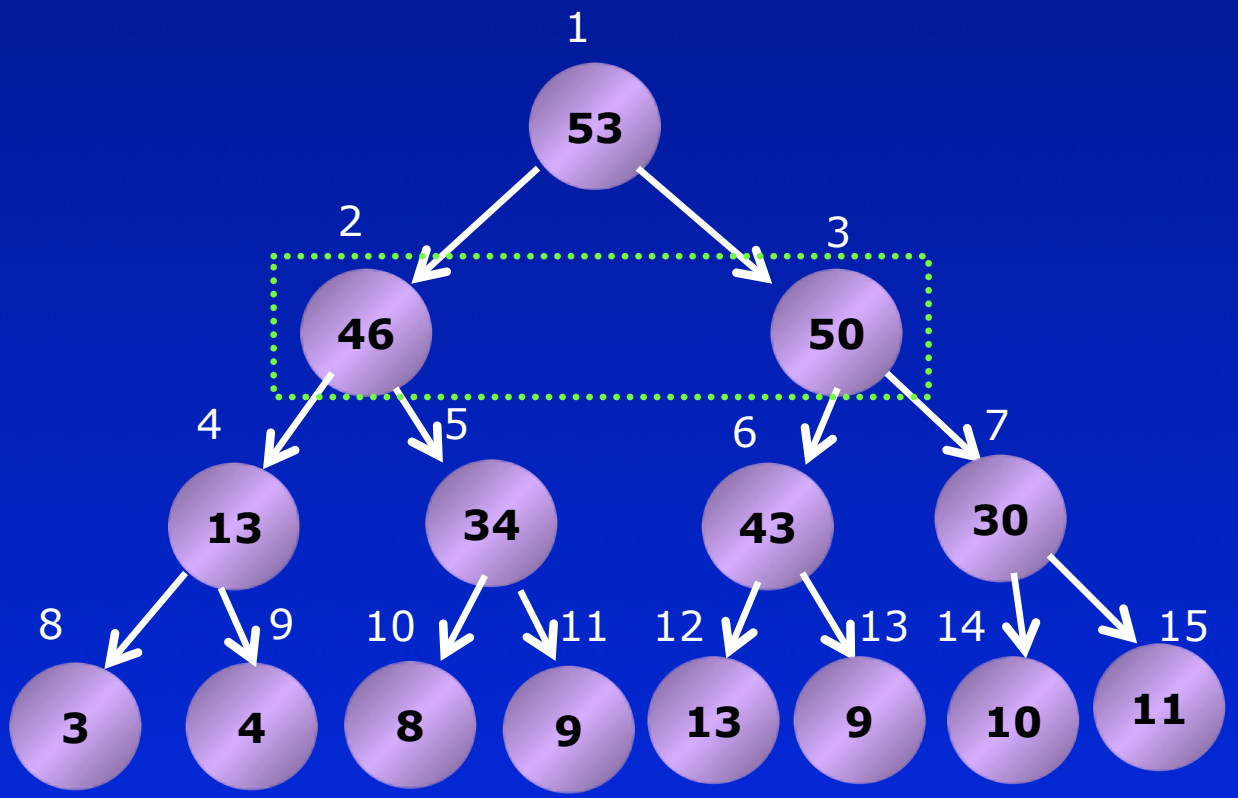
- **1964** von Robert W. Floyd entwickelt
- hat die gleiche *worst-case*-Komplexität wie **Mergesort** (**n×log(n)**)
- aber den Vorteil, dass die Sortierung am Ort geschieht.
- es wird eine zusätzliche **virtuelle** Datenstruktur (der **Heap**) verwendet, um die zu sortierenden Daten intern zu verwalten.

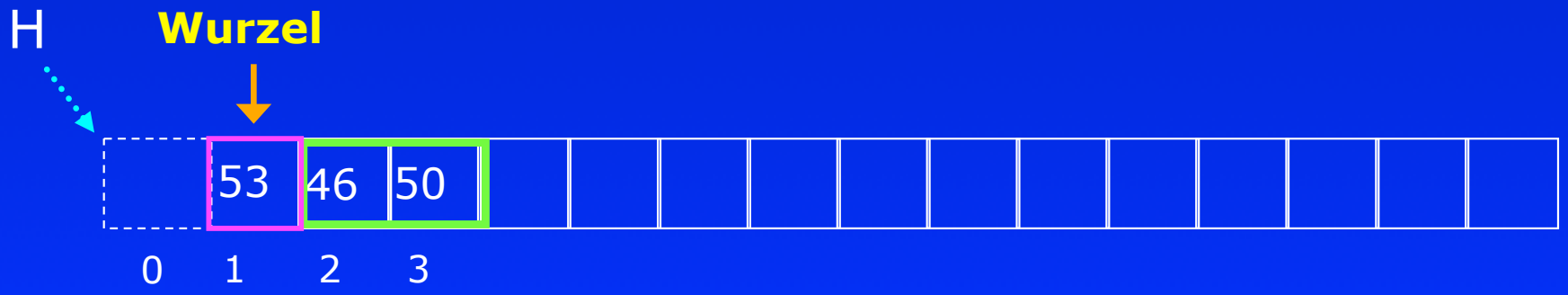
# Heap


1
53
2
46
3
50
4
13
5
34
6
43
7
30
8
3
9
4
10
8
11
9
12
13
13
9
14
10
15
11

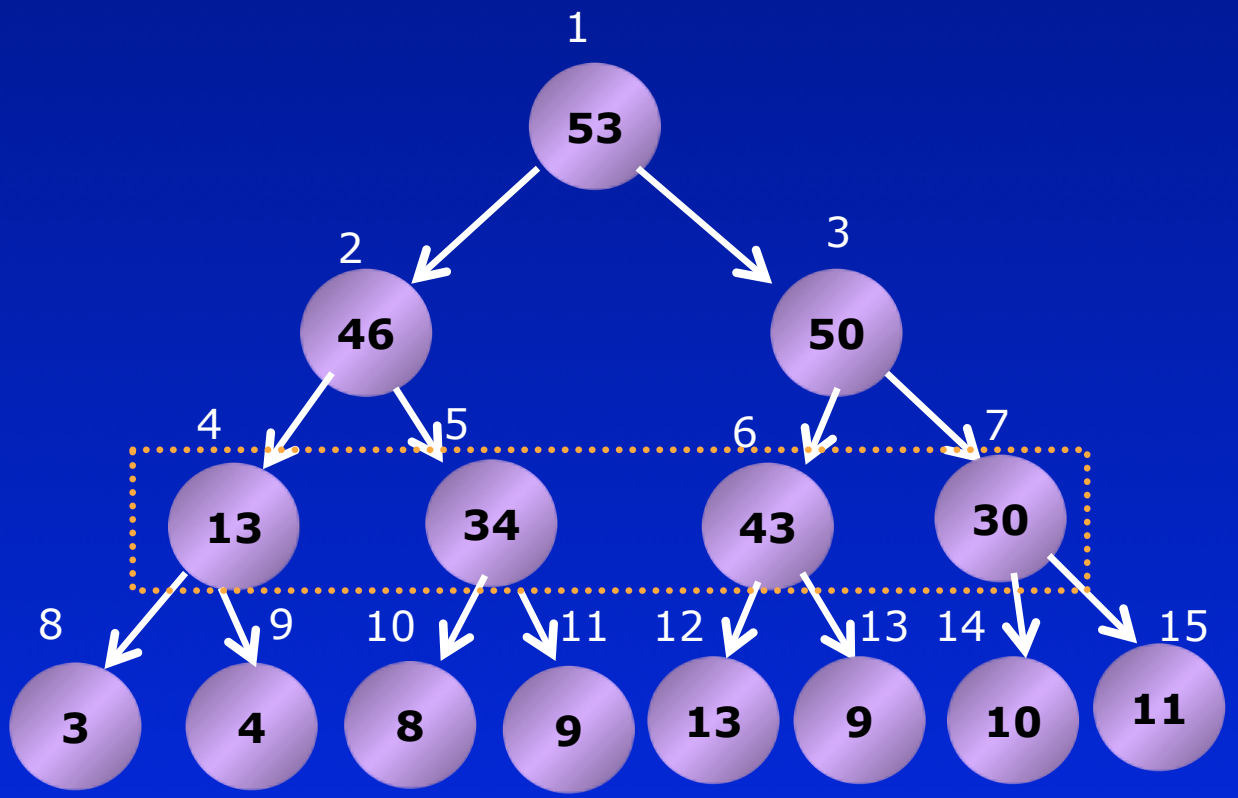

H
Wurzel
53
0
1

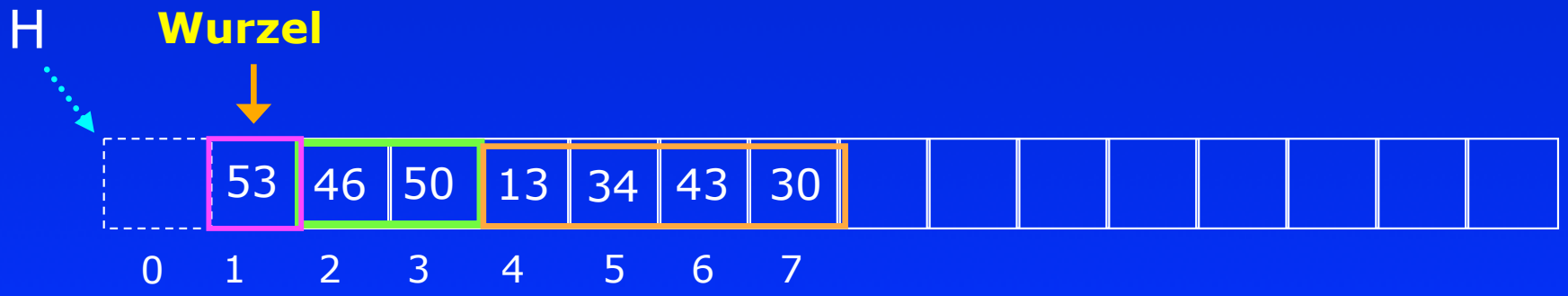
# Heap


1
53
2
46
3
50
4
13
5
34
6
43
7
30
8
3
9
4
10
8
11
9
12
13
13
9
14
10
15
11

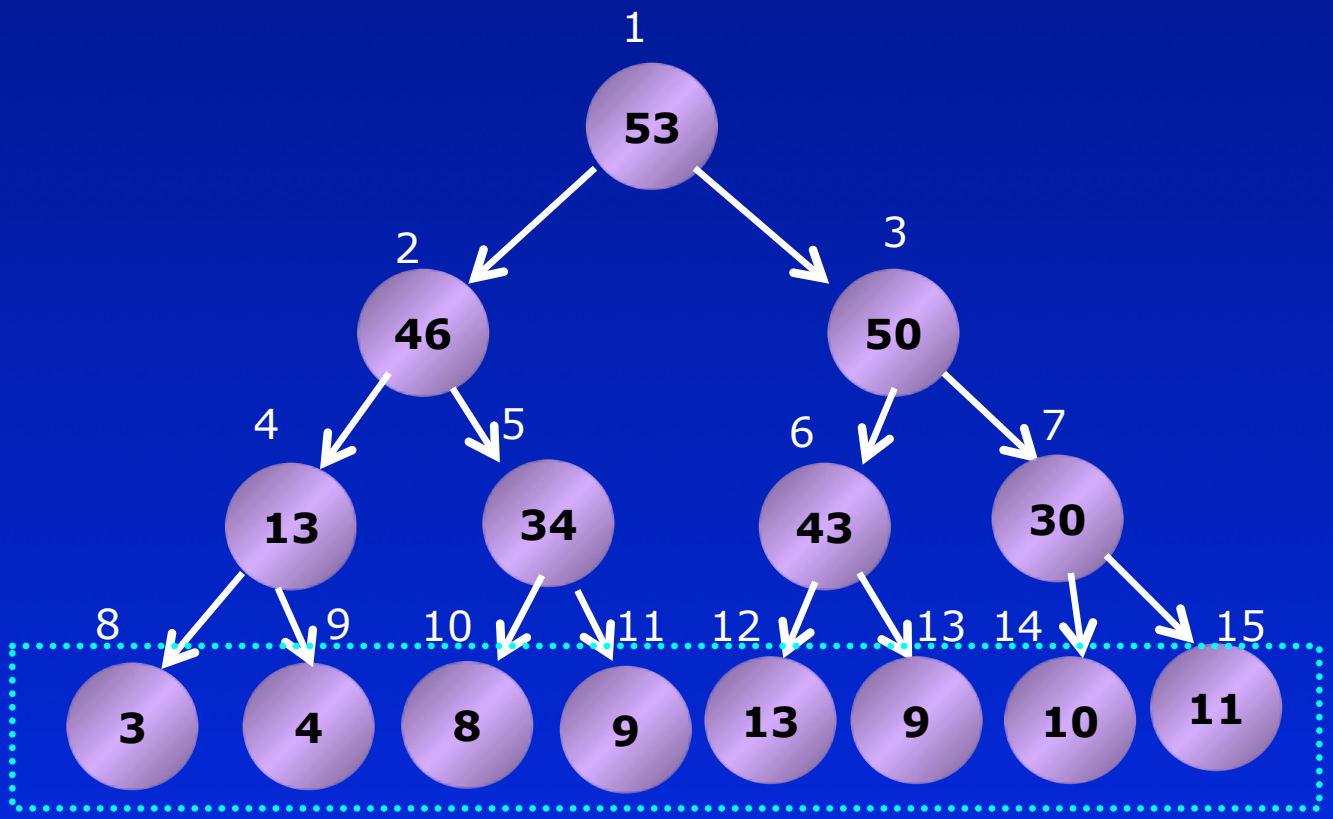

H
Wurzel
53
46
50
0
1
2
3

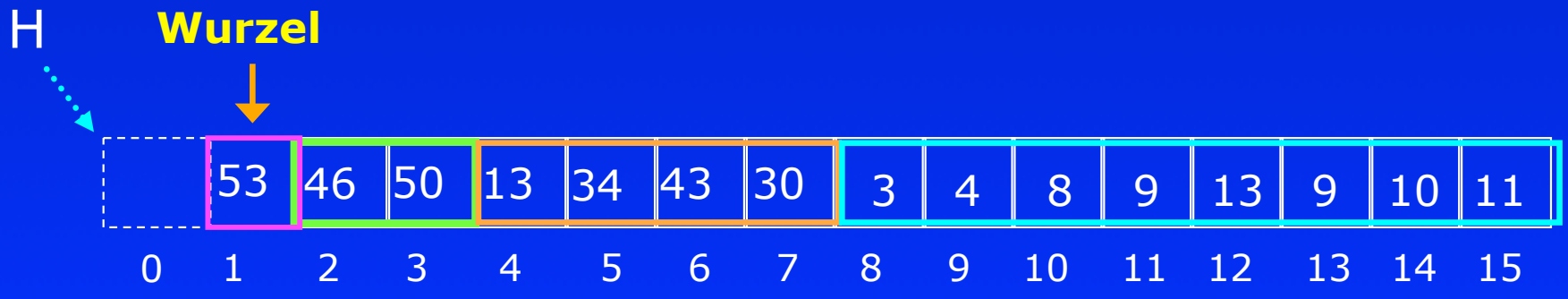
# Heap


1
53
2
46
3
50
4
13
5
34
6
43
7
30
8
3
9
4
10
8
11
9
12
13
13
9
14
10
15
11


H
Wurzel
53
46
50
13
34
43
30
0
1
2
3
4
5
6
7

# Heap

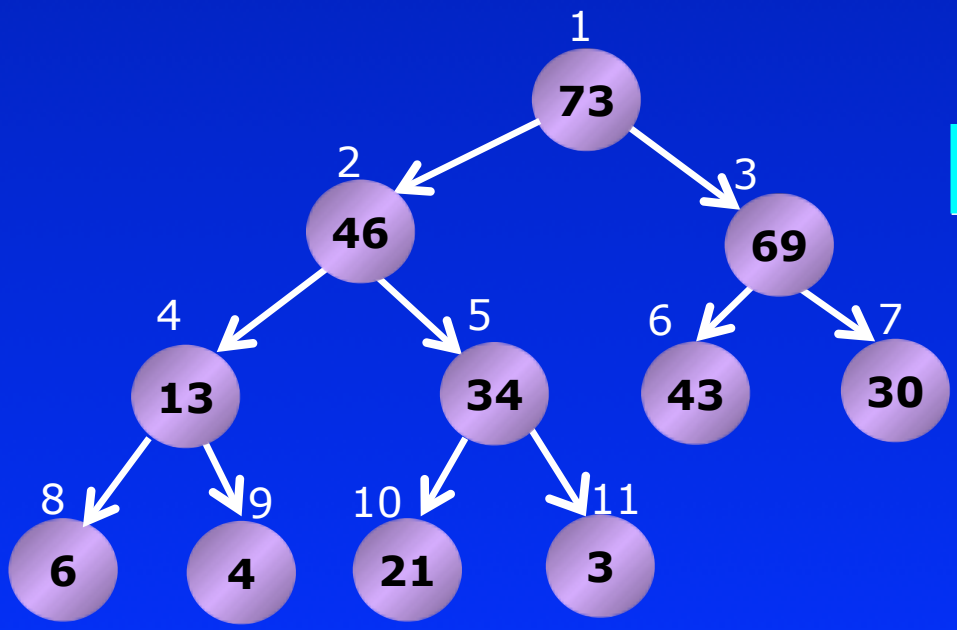

1
53
2
46
3
50
4
13
5
34
6
43
7
30
8
3
9
4
10
8
11
9
12
13
13
9
14
10
15
11

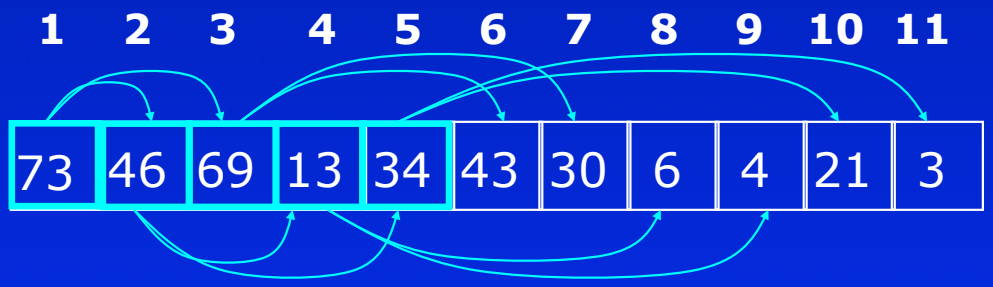

H
Wurzel
53
46
50
13
34
43
30
3
4
8
9
13
9
10
11
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

# Heap

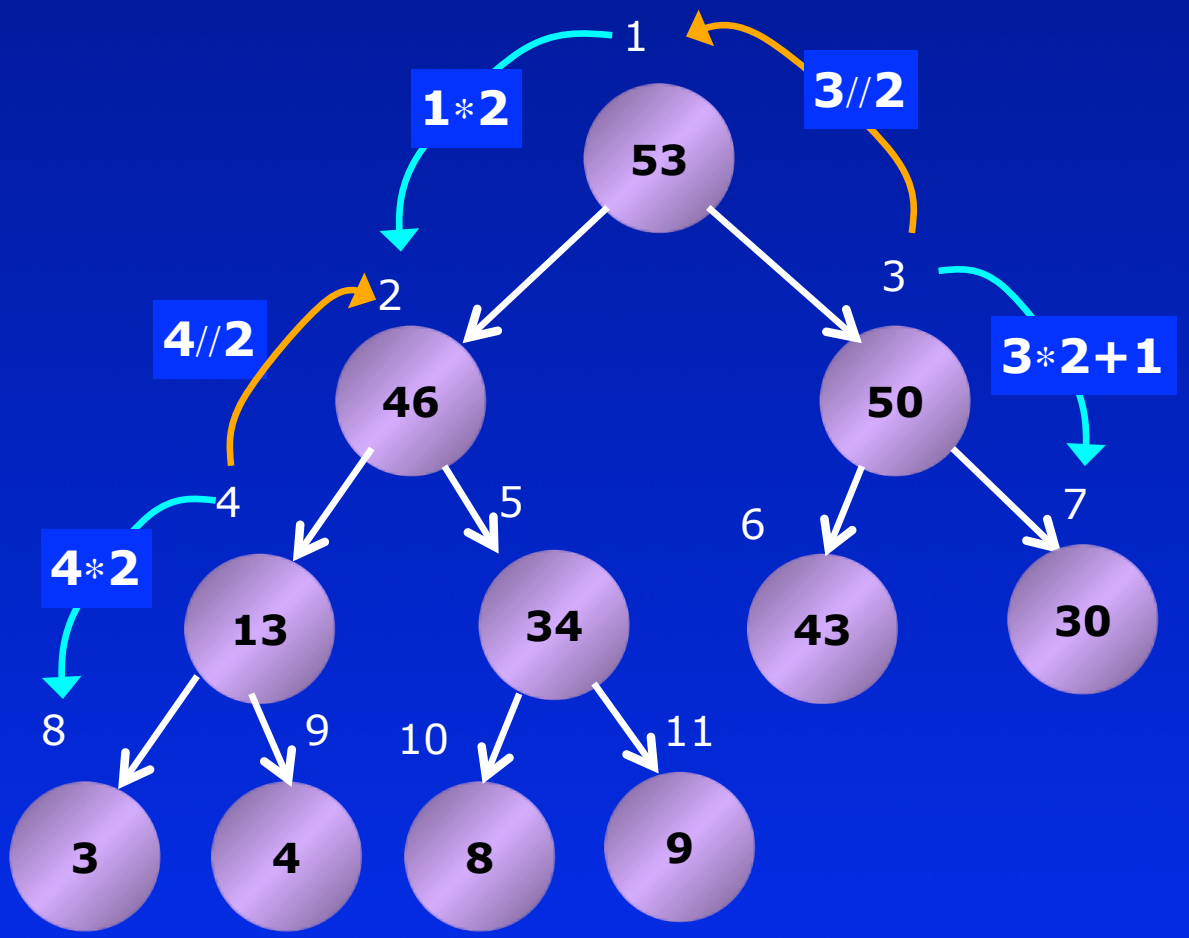
**Heap**
oder
**Max-Heap**

ist ein **Binärbaum**, der **in einem Feld gespeichert** wird und die Eigenschaft hat, dass das Element, das in jedem Knoten des Baums gespeichert ist, größer oder gleich ist als alle Elemente seines linken und rechten Unterbaums.



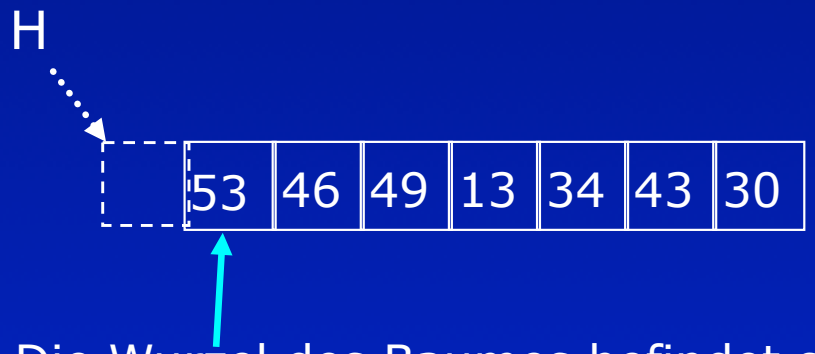



# Navigieren im Heap

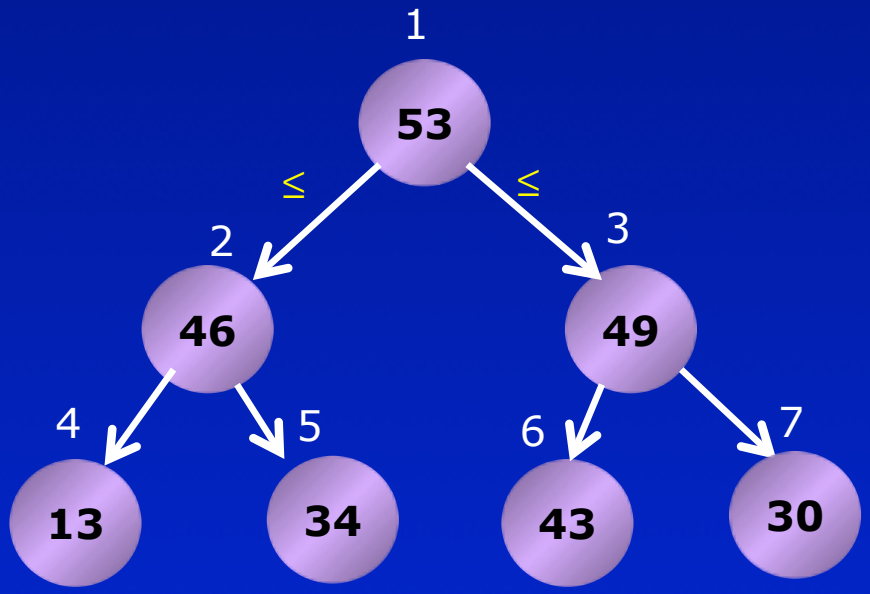



Das linke Kind einer beliebigen Position **i** des Heaps ist gleich **i*2**
Das rechte Kind einer beliebigen Position **i** ist gleich **i*2+1**
Das Elternteil eines beliebigen Kindes **i** ist gleich **i/2** (ganzzahlige Division)

# Heap-Hilfsfunktionen




Die Wurzel des Baumes befindet sich immer in der Position **1** des Feldes (**H[1]**).




Wir können für eine beliebige Position **i** in unserem Array folgende Funktionen definieren:

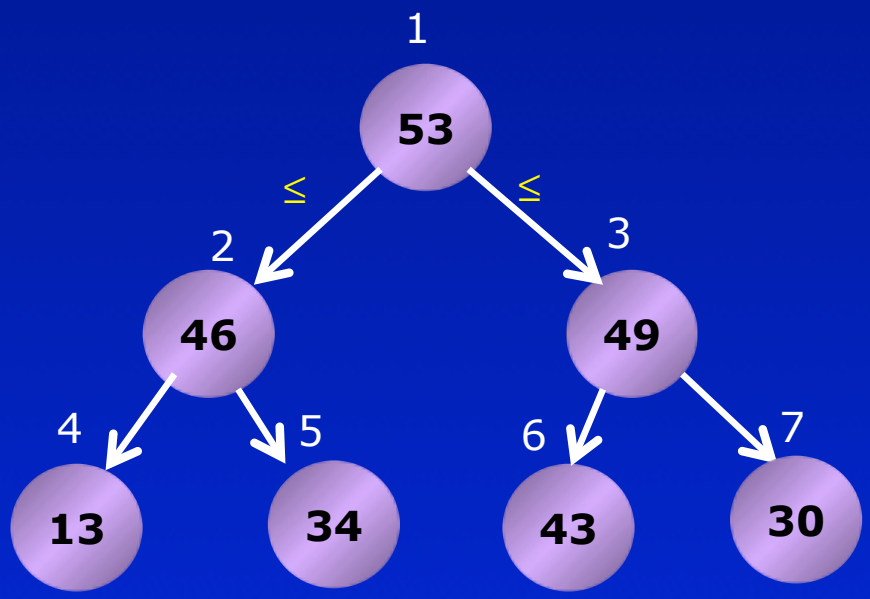
```
def parent(i):
    return i//2
```

```
def left(i):
    return i*2
```

```
def right(i):
    return i*2+1
```

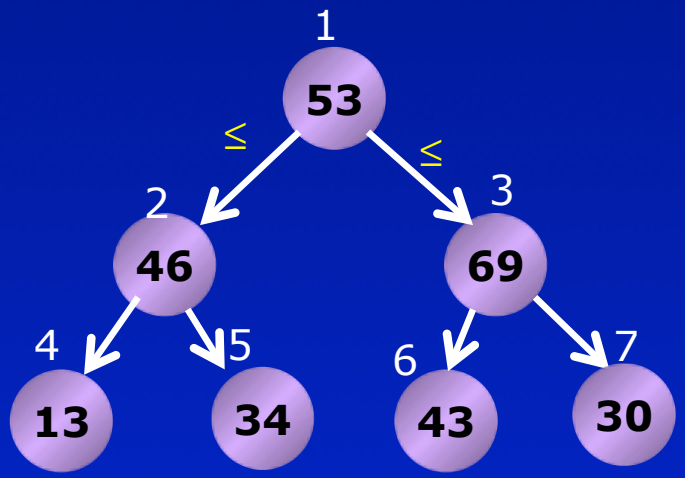
# Heap-Hilfsfunktionen

Für die Berechnung der Funktionen **parent**, **left** und **right** ist es günstiger, den **Heap** ab der Position **1** des Feldes zu speichern.




Die Größe des **Heaps** wird an der Position **0** des Feldes gespeichert (H[0]).

# Heap-Hilfsfunktionen


Folgende zwei Funktionen definieren wir, um unseren **Heapsort**-Algorithmus übersichtlicher zu machen.

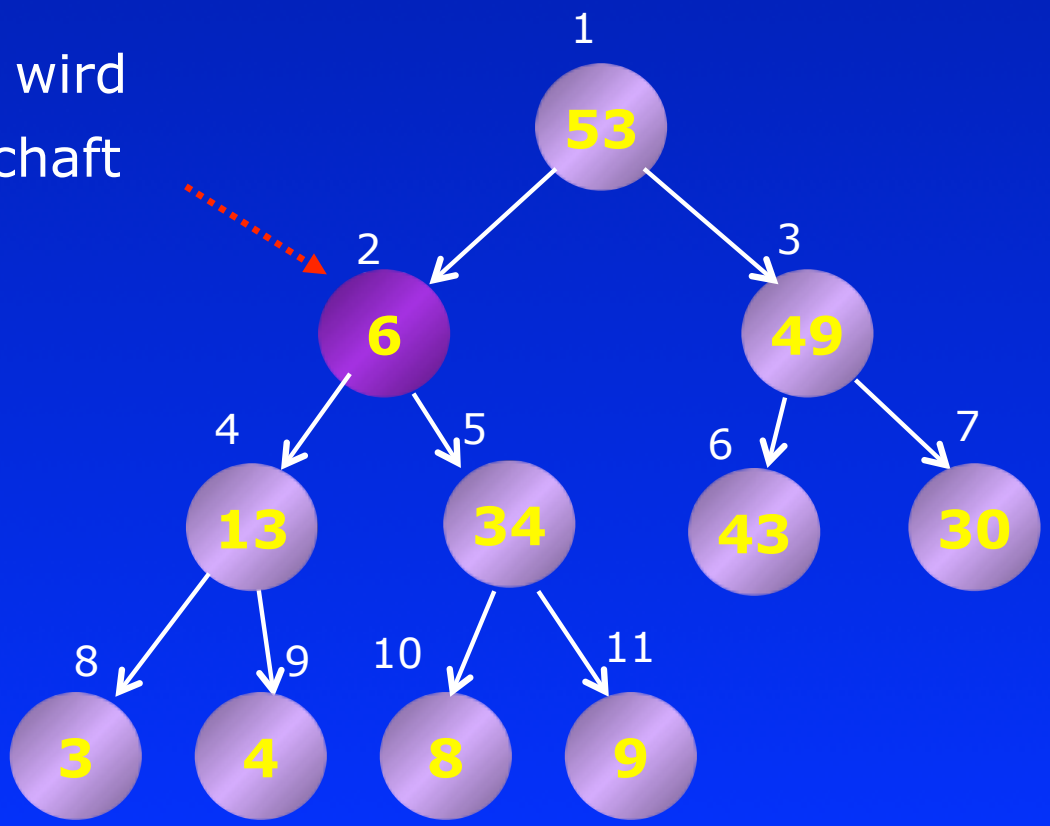
```
def heap_size(H):
    return H[0]
```

```
def dec_heap_size(H):
    H[0] = H[0]-1
```

# max_heapify-Funktion

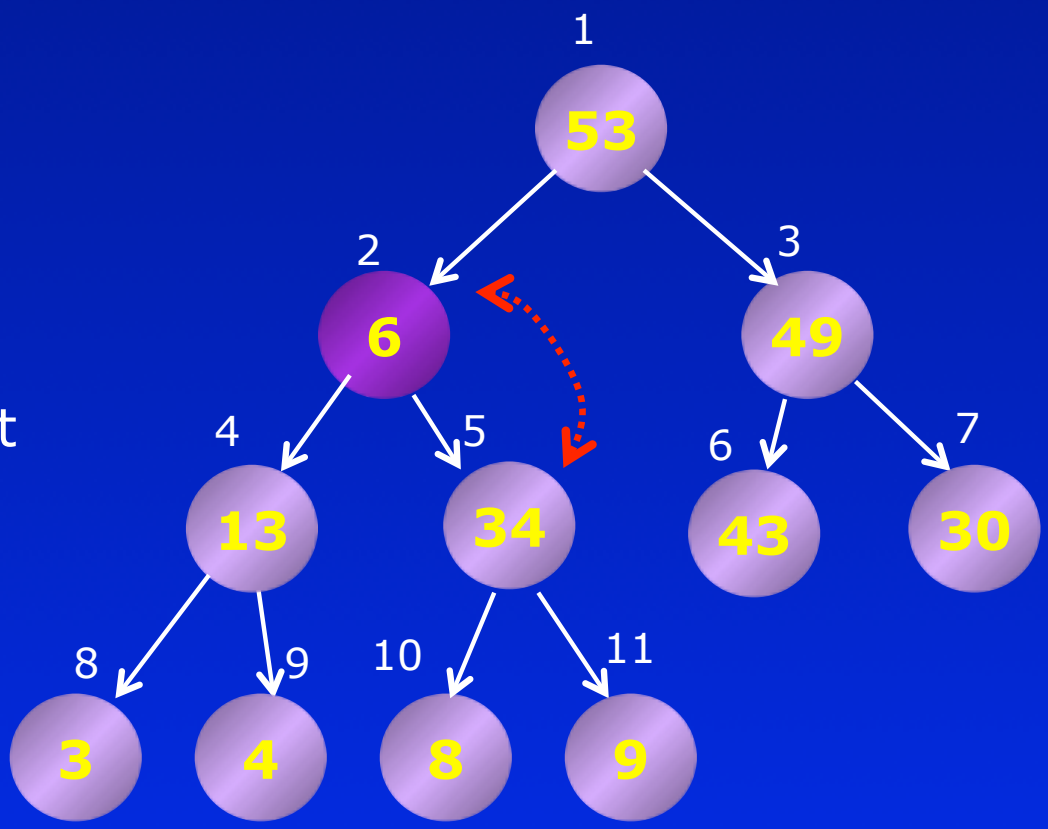
Die **max_heapify-Funktion** soll einen beliebigen Knoten des Baumes bekommen und testen, ob die **heap**-Eigenschaft erfüllt wird. Wenn das nicht der Fall ist, wird der Fehler korrigiert, indem das größte von beiden Kindern gegen den jeweiligen Knoten vertauscht wird.

In der Position 2 wird die Heap-Eigenschaft verletzt.



# max_heapify-Funktion

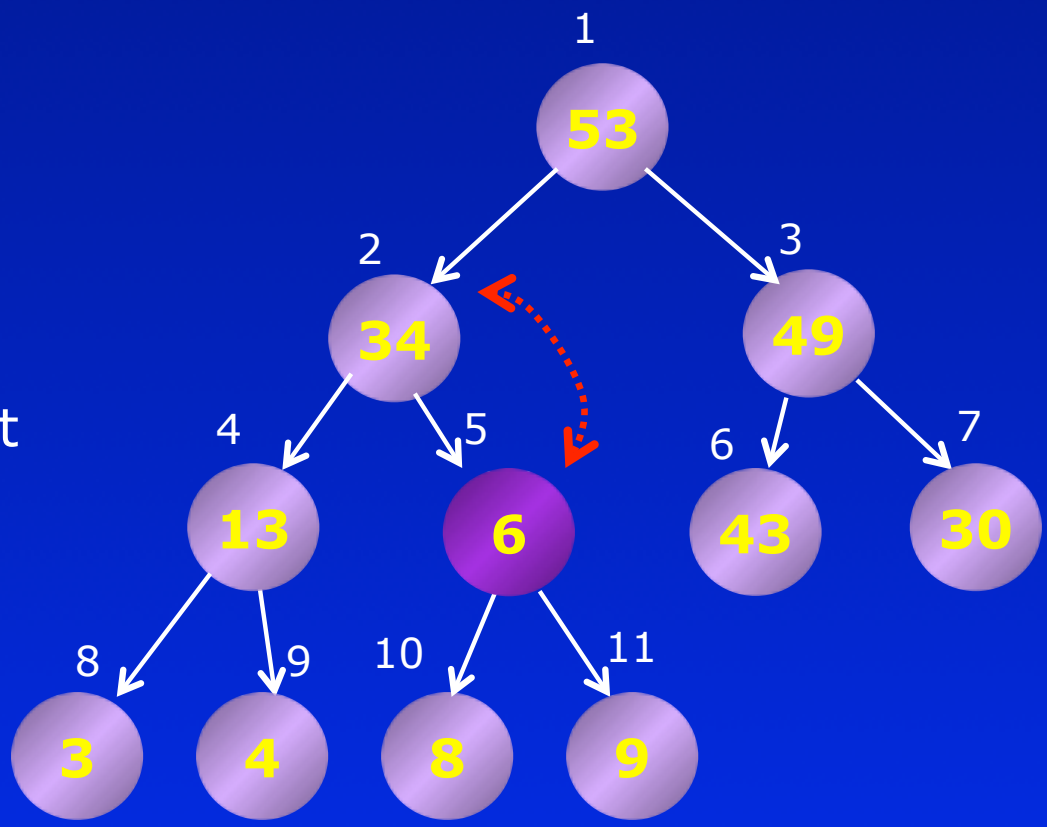
Das größere von beiden Kindern wird mit dem Element in Position **2** vertauscht, wenn es gleichzeitig größer als das Element in Position **2** ist.




Nach dem Vertauschen ist die heap-Eigenschaft an der Position 5 verloren gegangen; deswegen muss an dieser Stelle die max_heapify-Funktion **rekursiv** aufgerufen werden.

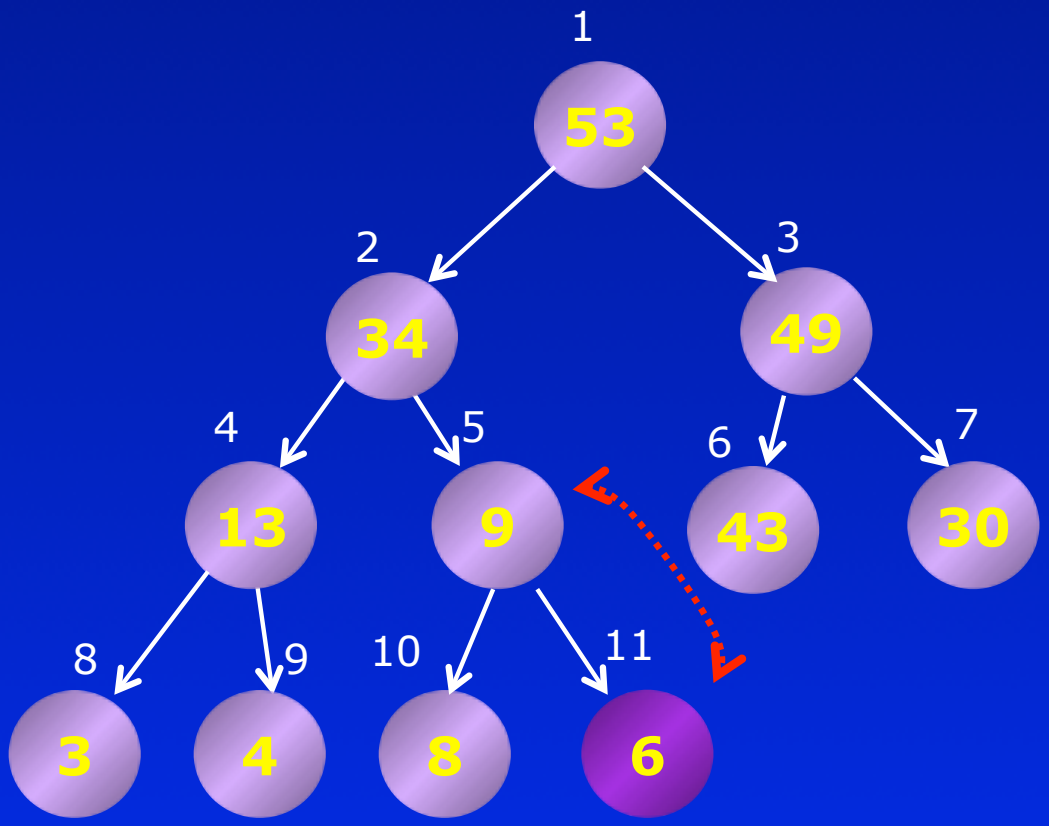
# max_heapify-Funktion

Das größere von beiden Kindern wird mit dem Element in Position **2** vertauscht, wenn es gleichzeitig größer als das Element in Position **2** ist.




Nach dem Vertauschen ist die heap-Eigenschaft an der Position 5 verloren gegangen; deswegen muss an dieser Stelle die max_heapify-Funktion **rekursiv** aufgerufen werden.

# max_heapify-Funktion




Die Position **11** des Baumes hat keine Kinder mehr und erfordert deswegen keine weiteren Korrekturen.

# max_heapify-Funktion

Die **max_heapify**-Funktion bekommt ein Feld **H** und eine Position des H-Feldes als Parameter.

```
def max_heapify ( H, pos ):
```

Die Funktion geht davon aus, dass das linke und rechte Kind der angegebenen Position die **max_heap**-Eigenschaft besitzen und überprüft zuerst nur, ob die heap-Eigenschaft an der angegebenen Position erfüllt wird. Wenn das der Fall ist, wird nichts getan und die Ausführung der Funktion wird beendet.

Wenn die heap-Eigenschaft nicht erfüllt wird, wird das Problem korrigiert, indem die angegebene Position mit dem größeren von beiden Kindern vertauscht wird. Dann wird die Funktion rekursiv mit dem veränderten Kind aufgerufen, weil die heap-Eigenschaft des veränderten Kinderknotens eventuell nicht mehr erfüllt wird.

# max_heapify-Funktion

```
def max_heapify (H, pos):
        left_t = left (pos)
        right_t = right(pos)
        if left_t<=heap_size(H)  and  H[left_t]>H[pos]:
                biggest = left_t
        else:
                biggest = pos
        if right_t<=heap_size(H)  and  H[right_t]>H[biggest]:
                biggest = right_t
        if biggest != pos:
                H[pos], H[biggest] = H[biggest], H[pos]
                max_heapify( H, biggest )
```

# build_heapify-Funktion

Wie können wir aus einer beliebigen Zahlenreihe einen Heap konstruieren?

Zuerst müssen die Zahlen ab der Position **1** gespeichert werden.

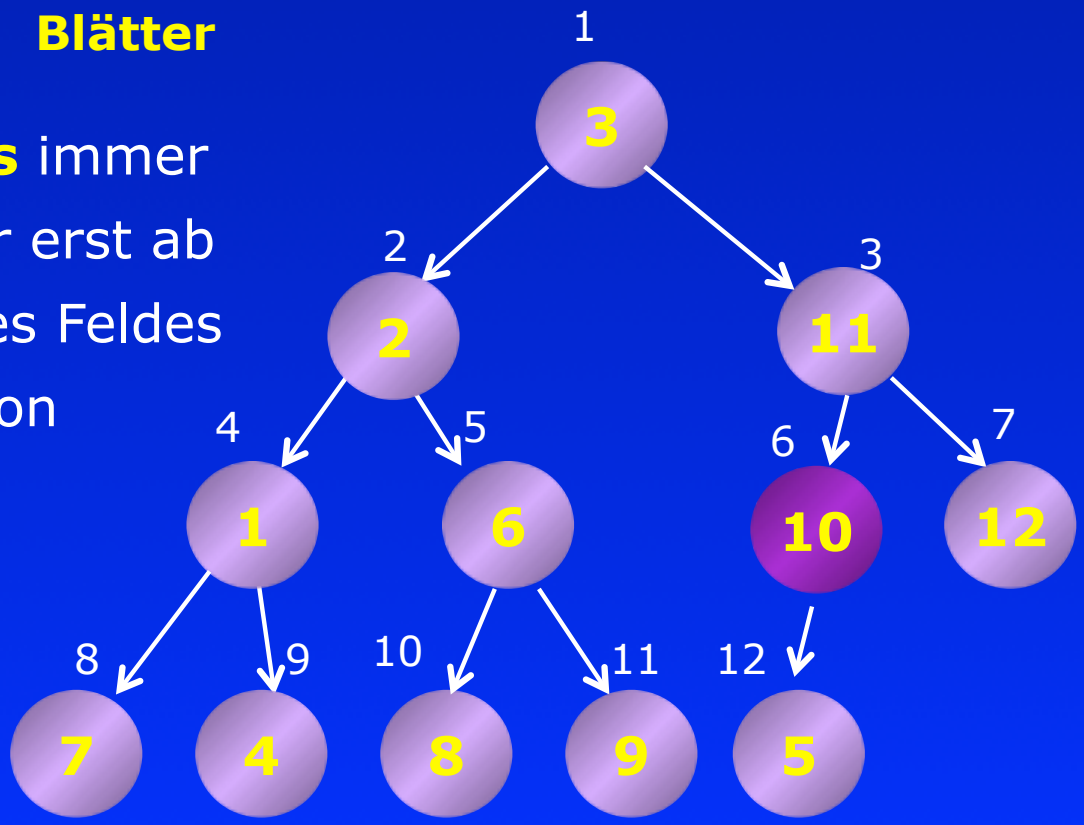
H

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **12** | 3 | 2 | 11 | 1 | 6 | 10 | 12 | 7 | 4 | 8 | 9 | 5 |

**Blätter**

Weil die Hälfte des **Heaps** immer Blätter sind, brauchen wir erst ab Ende des ersten Hälfte des Feldes die **max_heapify**-Funktion aufzurufen.

In der Position **0** des **H**-Feldes speichern wir die Größe unseres **Heaps**.

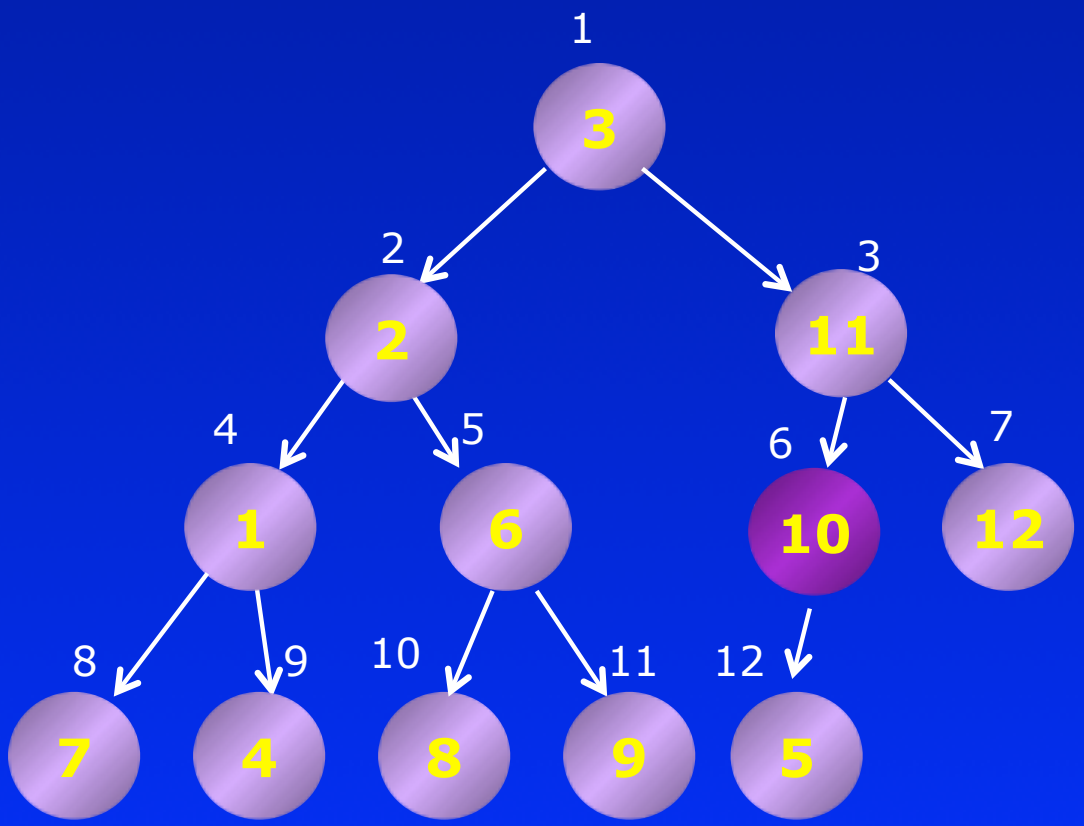


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 1 | 6 | 10 | 12 | 7 | 4 | 8 | 9 | 5 |

i

**max_heapify ( H, 6 )**

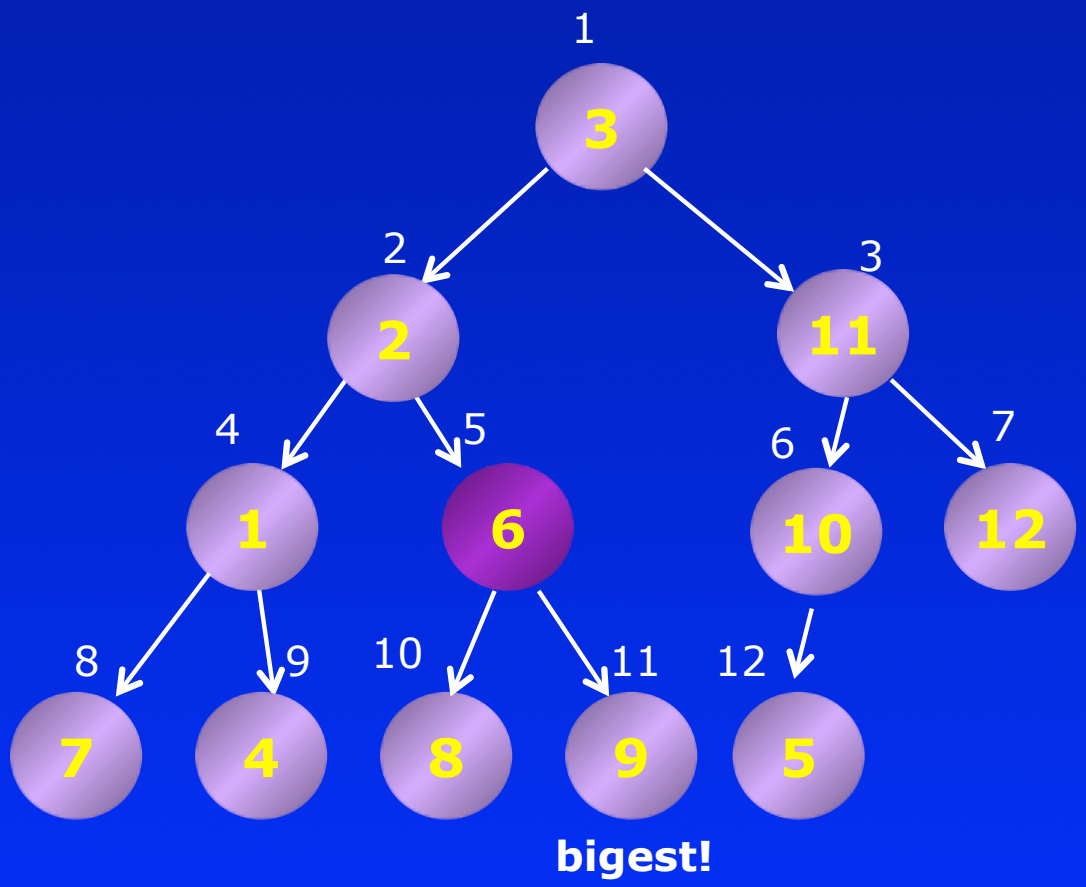


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 1 | 6 | 10 | 12 | 7 | 4 | 8 | 9 | 5 |

i

**max_heapify ( H, 5 )**

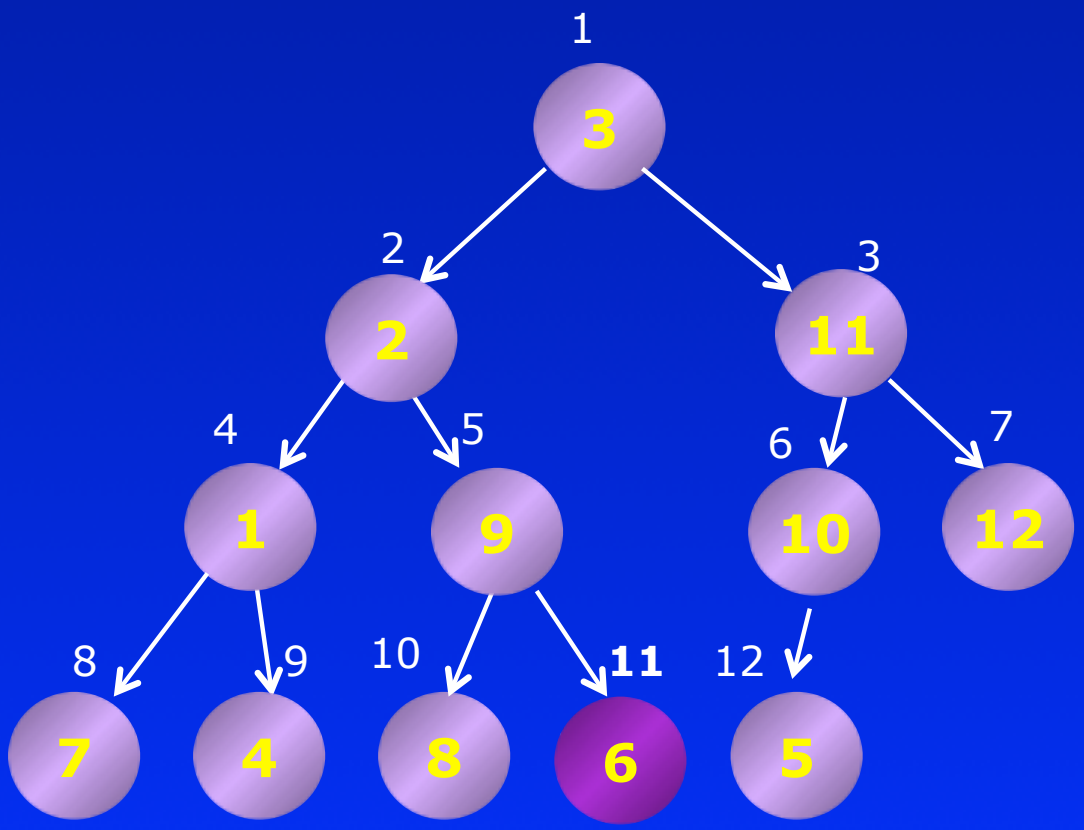


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 1 | 9 | 10 | 12 | 7 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 5 )**



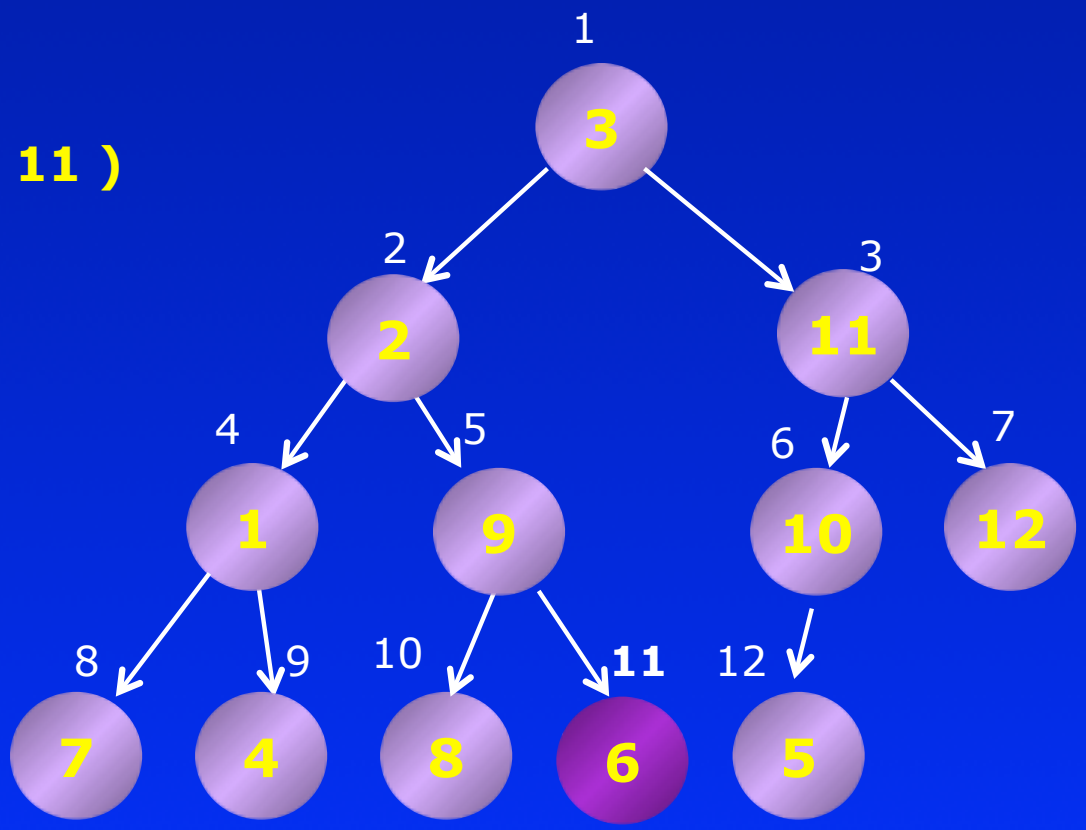
# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 1 | 9 | 10 | 12 | 7 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 5 )**

**max_heapify ( H, 11 )**

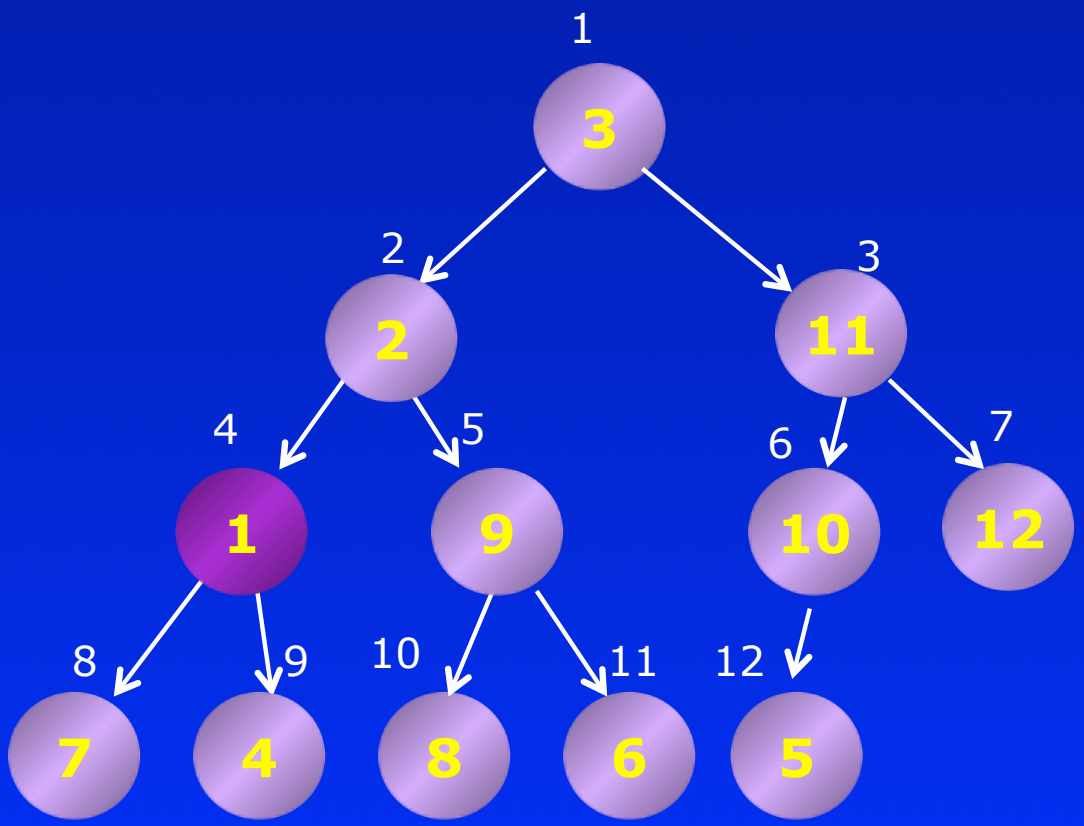


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 1 | 9 | 10 | 12 | 7 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 4 )**



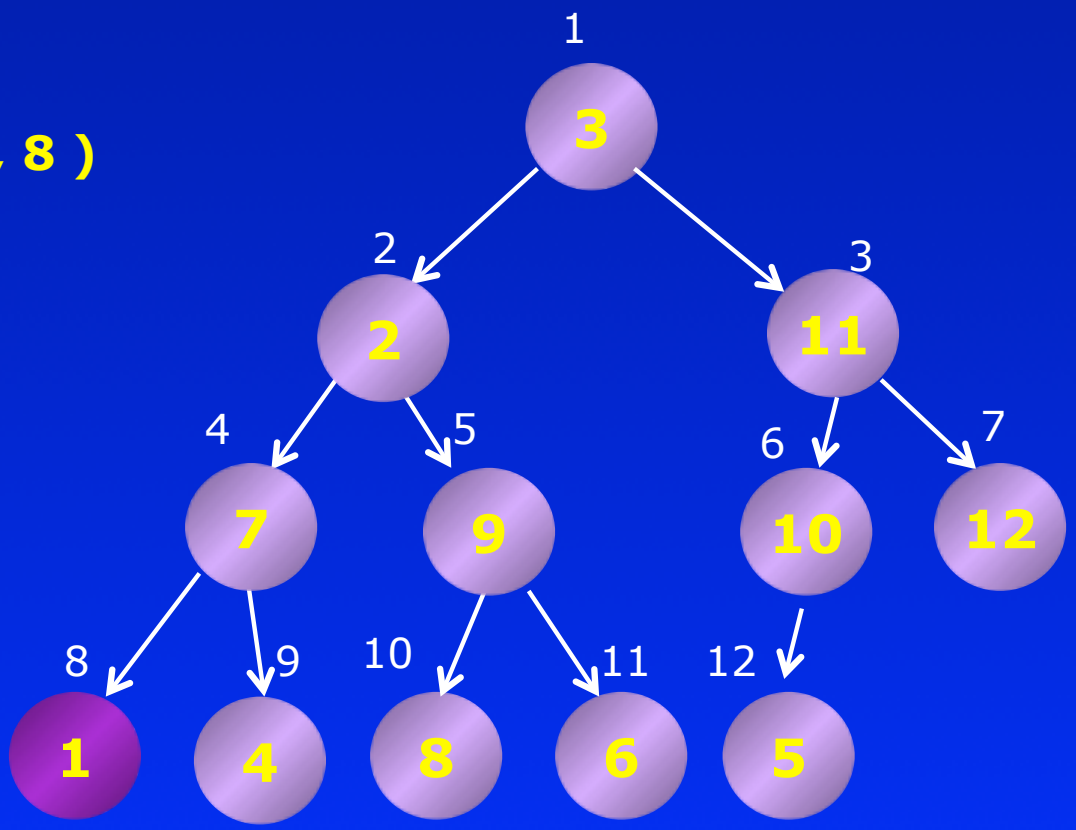
# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 7 | 9 | 10 | 12 | 1 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 4 )**

**max_heapify ( H, 8 )**

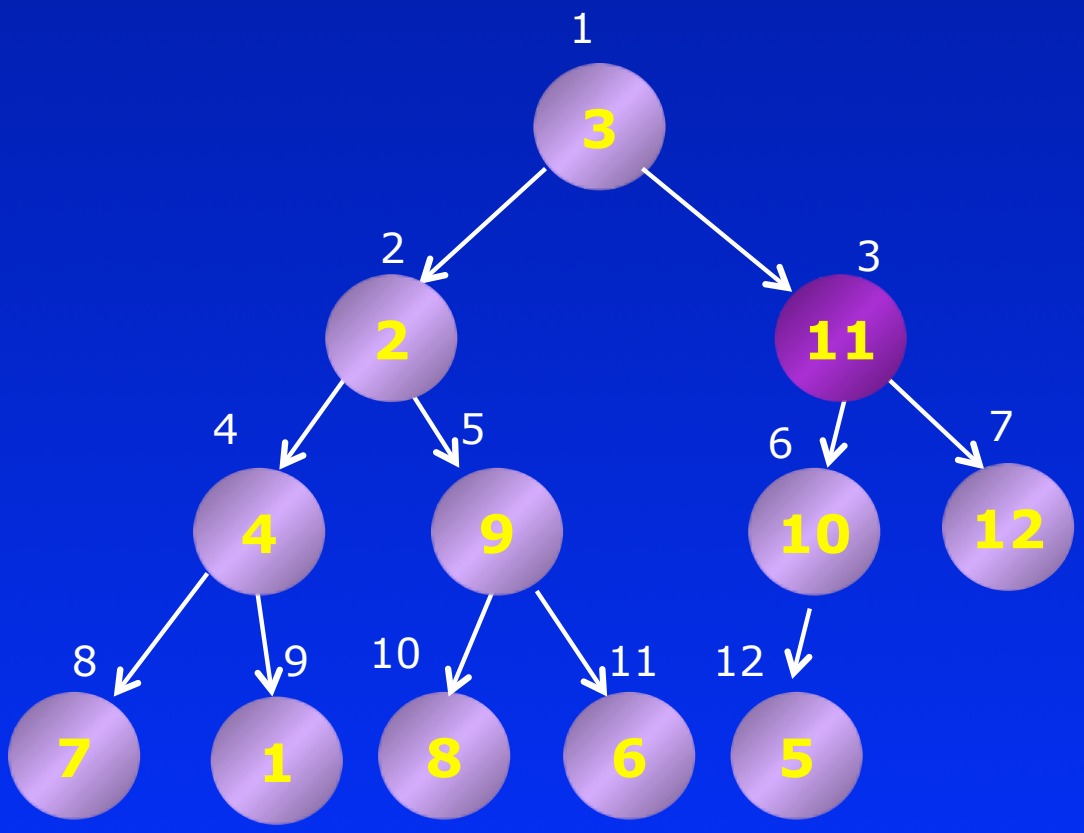


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 11 | 7 | 9 | 10 | 12 | 1 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 3 )**



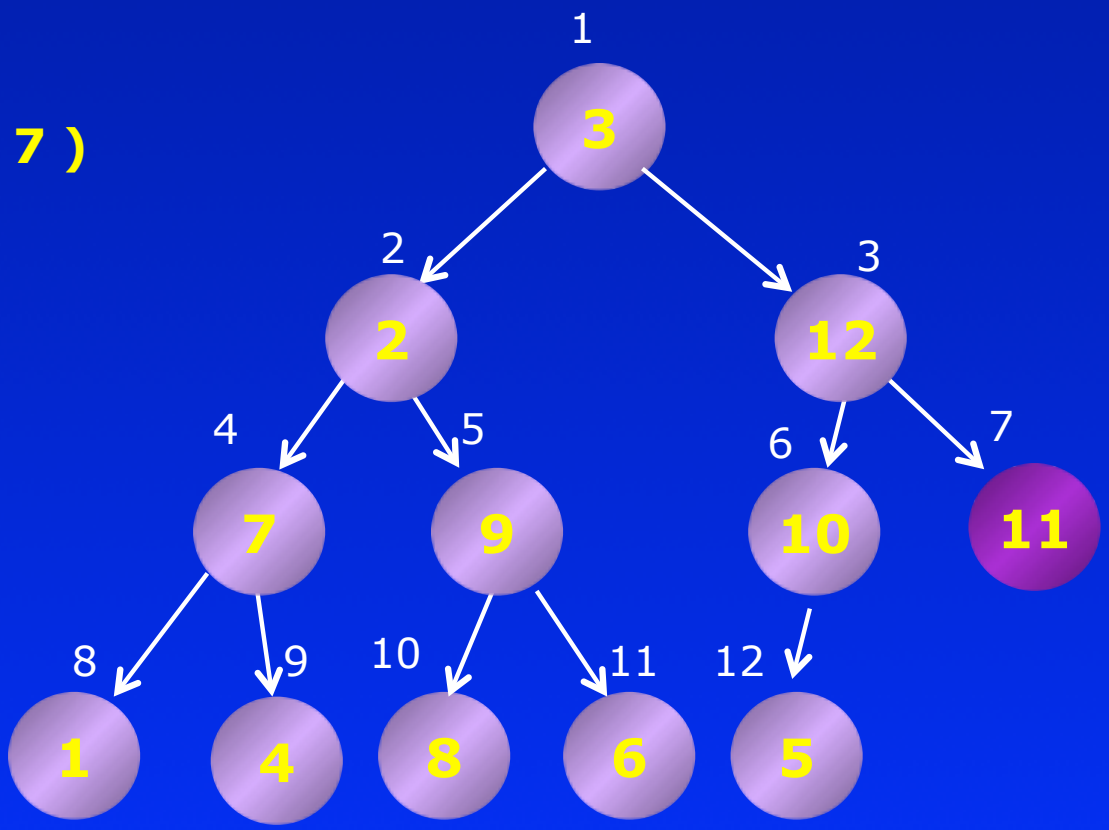
# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 12 | 7 | 9 | 10 | 11 | 1 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 3 )**

**max_heapify ( H, 7 )**

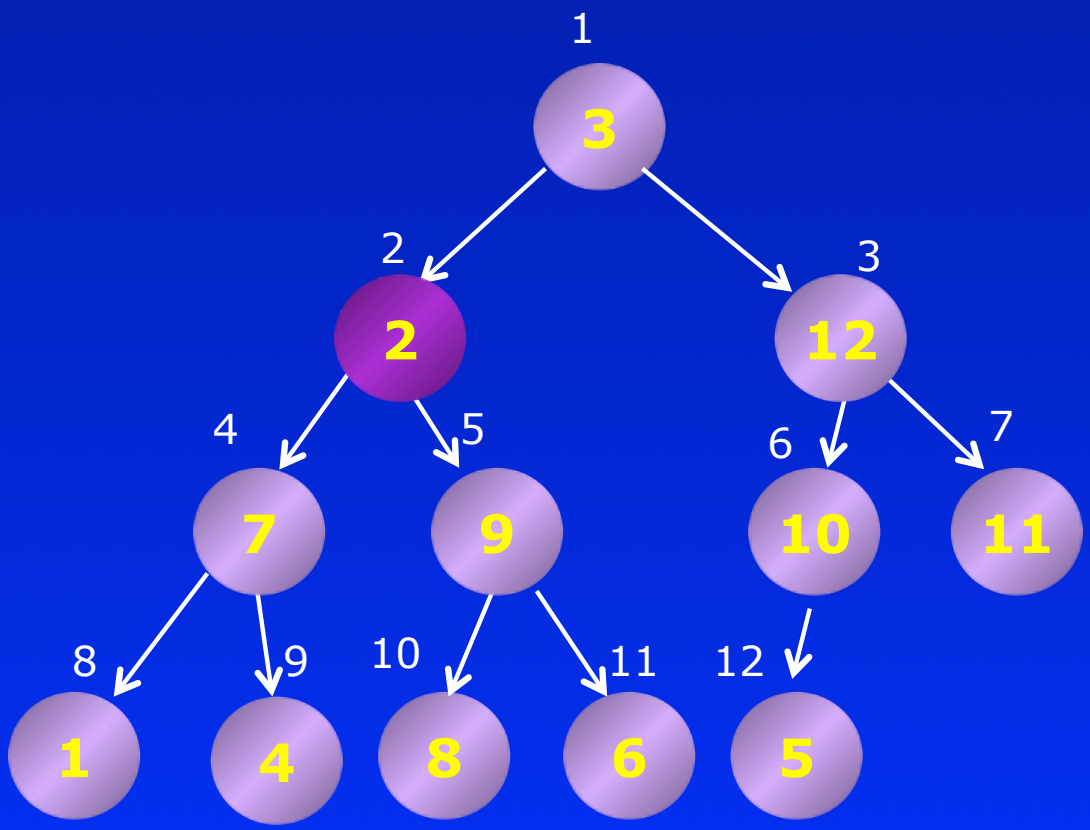


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 12 | 7 | 9 | 10 | 11 | 1 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 2 )**



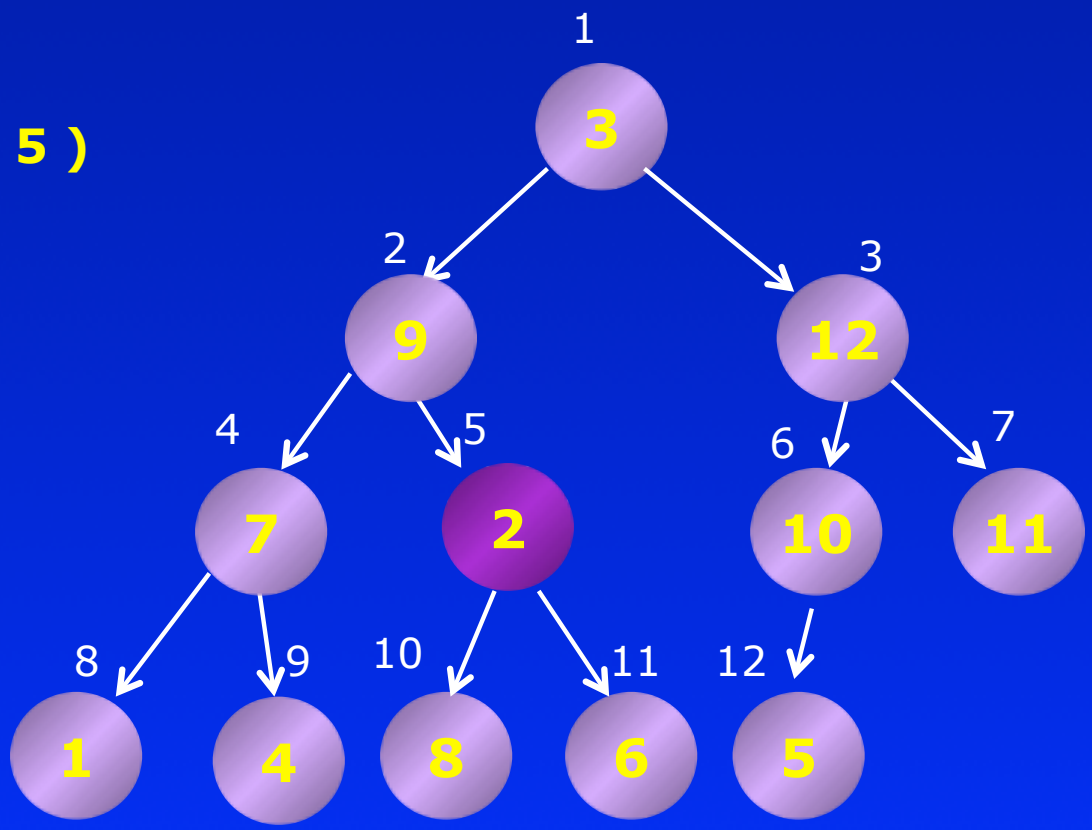
# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 12 | 7 | 2 | 10 | 11 | 1 | 4 | 8 | 6 | 5 |

i

**max_heapify ( H, 2 )**

**max_heapify ( H, 5 )**



# build_heapify-Funktion

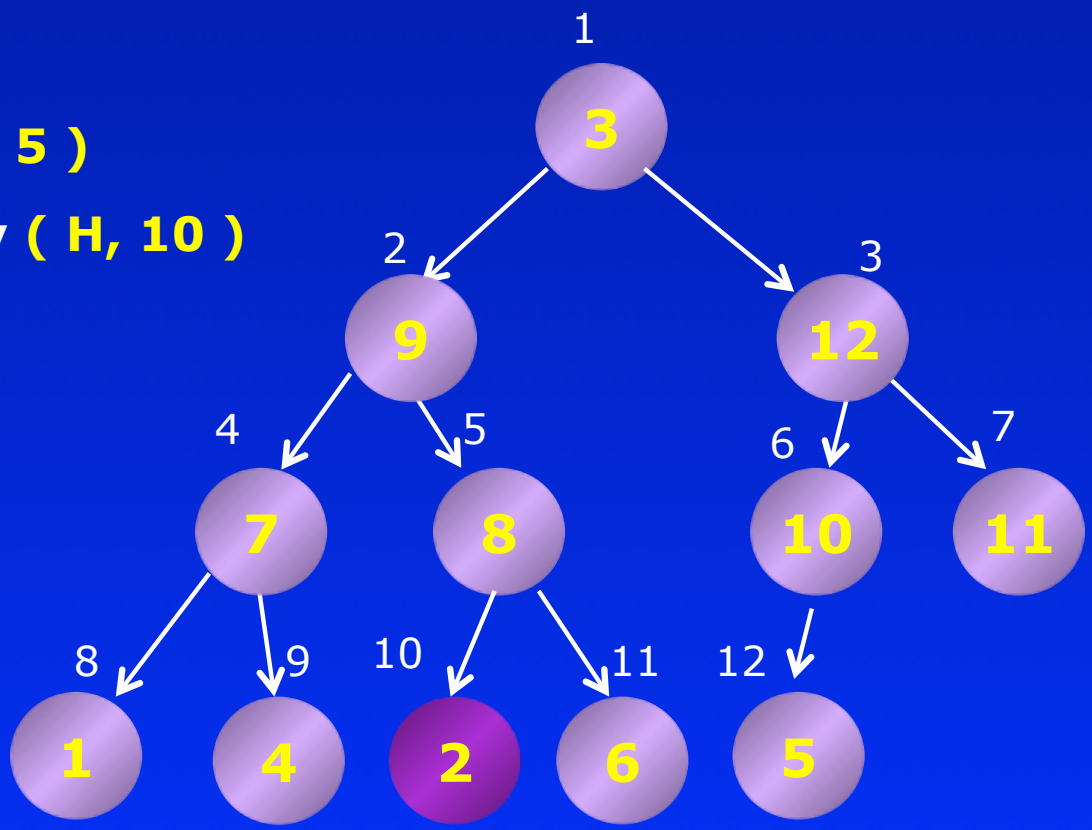
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 12 | 7 | 8 | 10 | 11 | 1 | 4 | 2 | 6 | 5 |

i

**max_heapify ( H, 2 )**

**max_heapify ( H, 5 )**

**max_heapify ( H, 10 )**

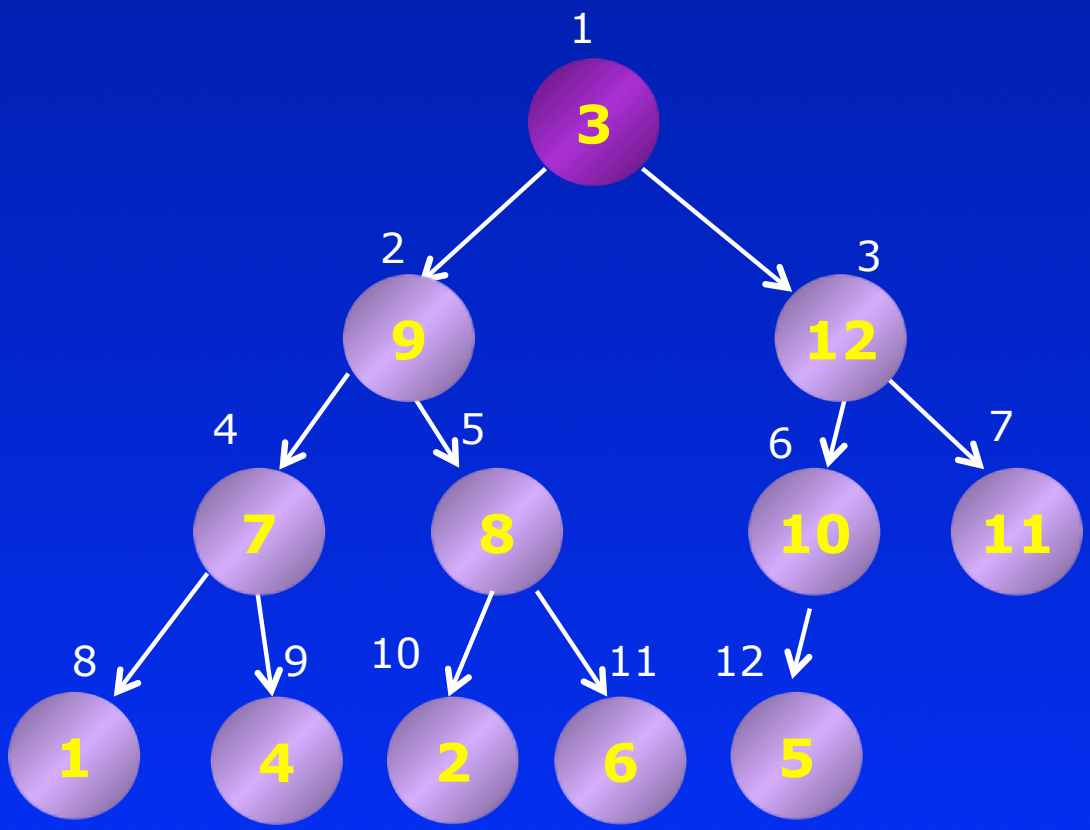


# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 12 | 7 | 8 | 10 | 11 | 1 | 4 | 2 | 6 | 5 |

i

**max_heapify ( H, 1 )**



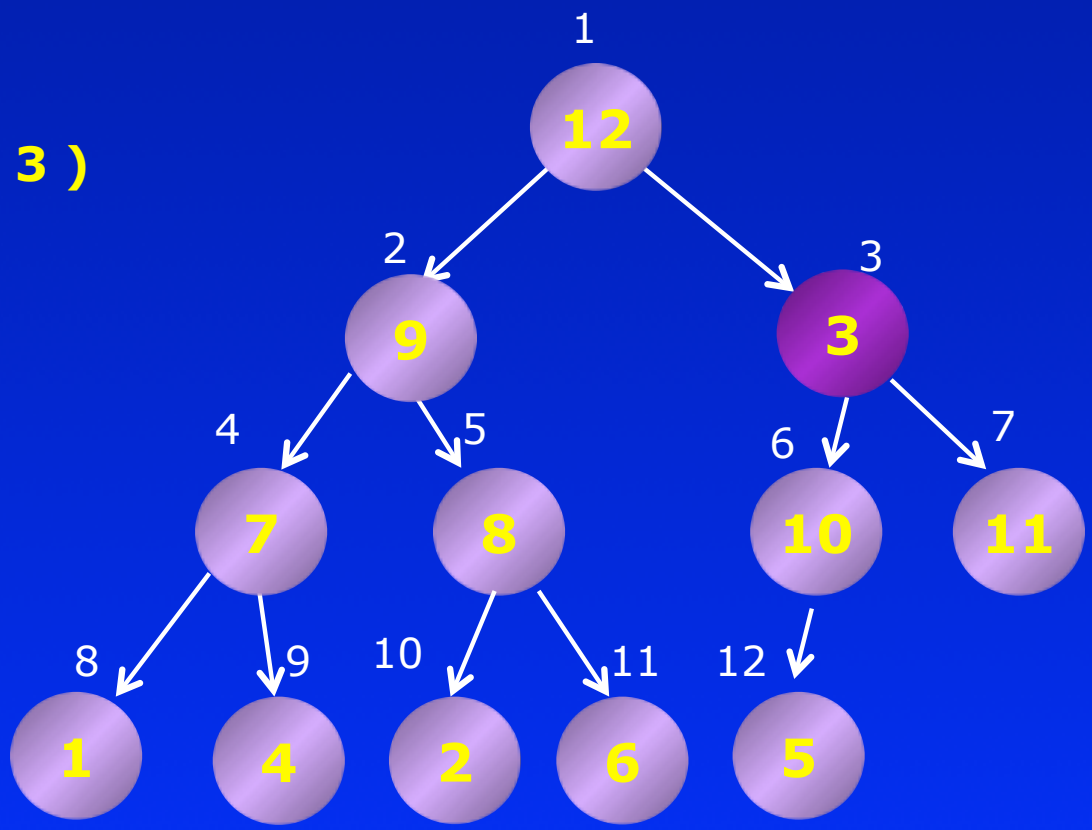
# build_heapify-Funktion

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 3 | 7 | 8 | 10 | 11 | 1 | 4 | 2 | 6 | 5 |

i

**max_heapify ( H, 1 )**

**max_heapify ( H, 3 )**



# build_heapify-Funktion

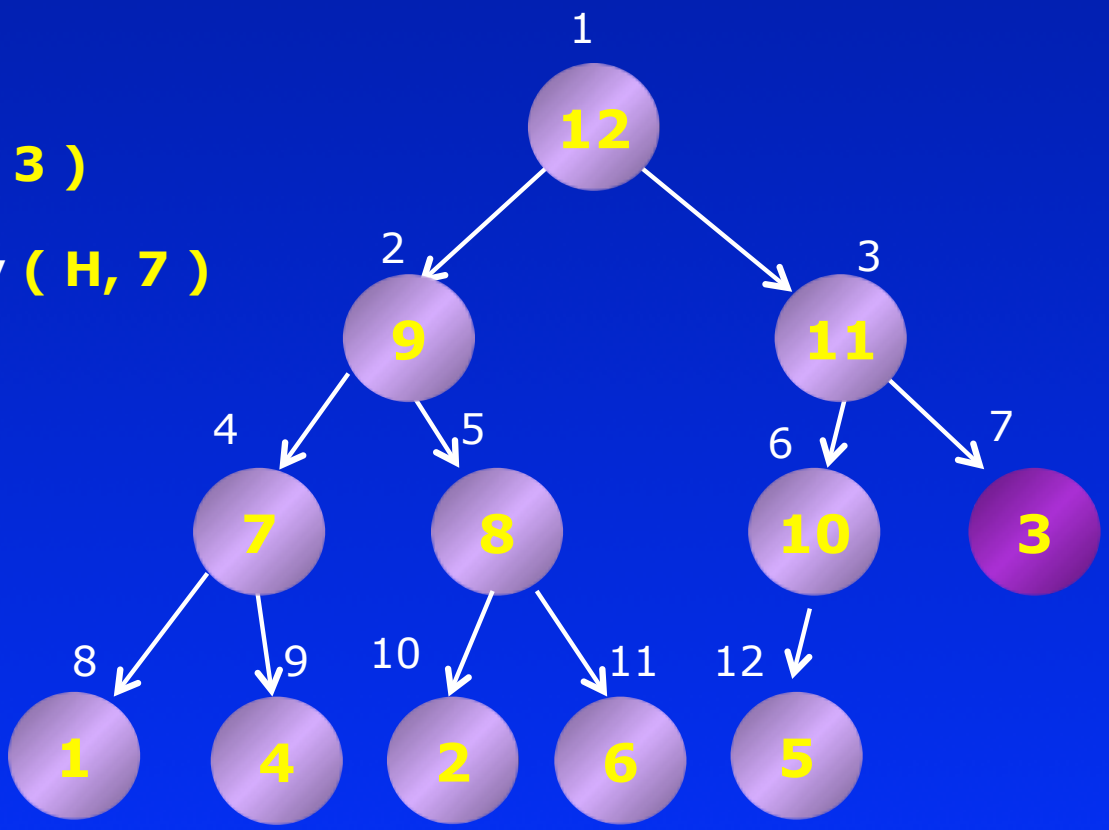
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 11 | 7 | 8 | 10 | 3 | 1 | 4 | 2 | 6 | 5 |

i

**max_heapify ( H, 1 )**

**max_heapify ( H, 3 )**

**max_heapify ( H, 7 )**



# build_heapify-Funktion

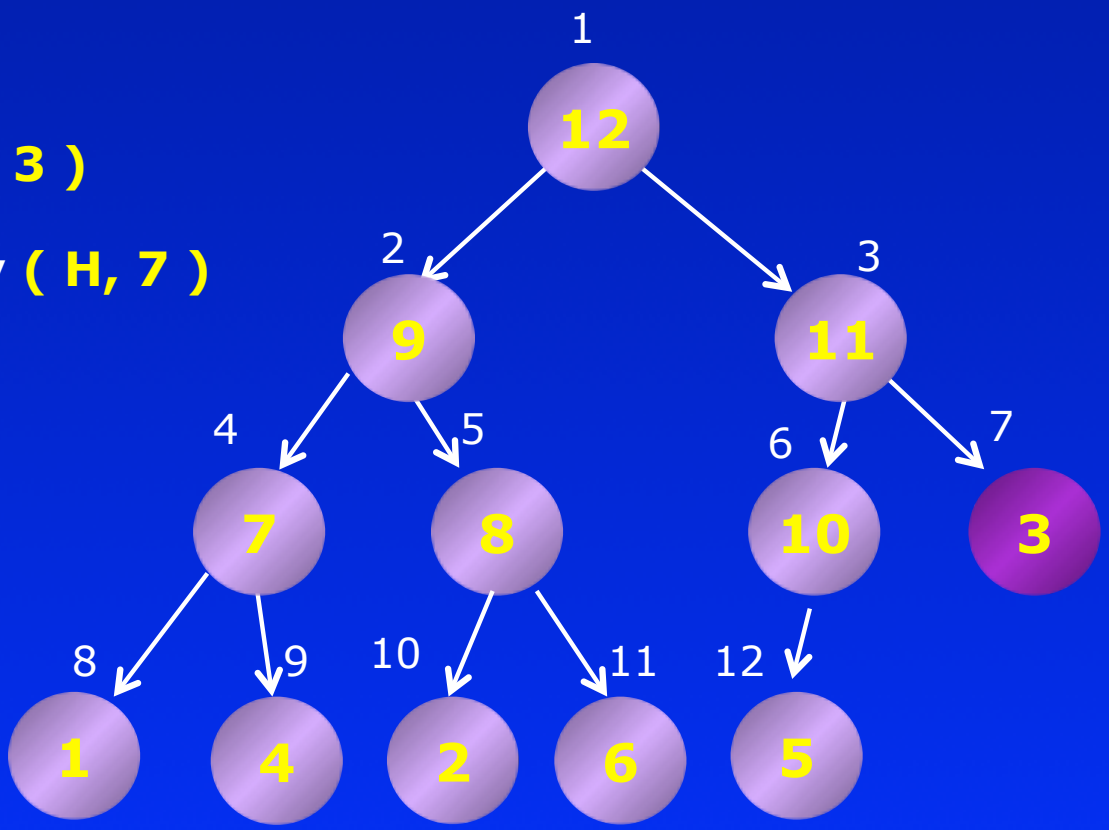
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 3 | 7 | 8 | 10 | 11 | 1 | 4 | 2 | 6 | 5 |

i

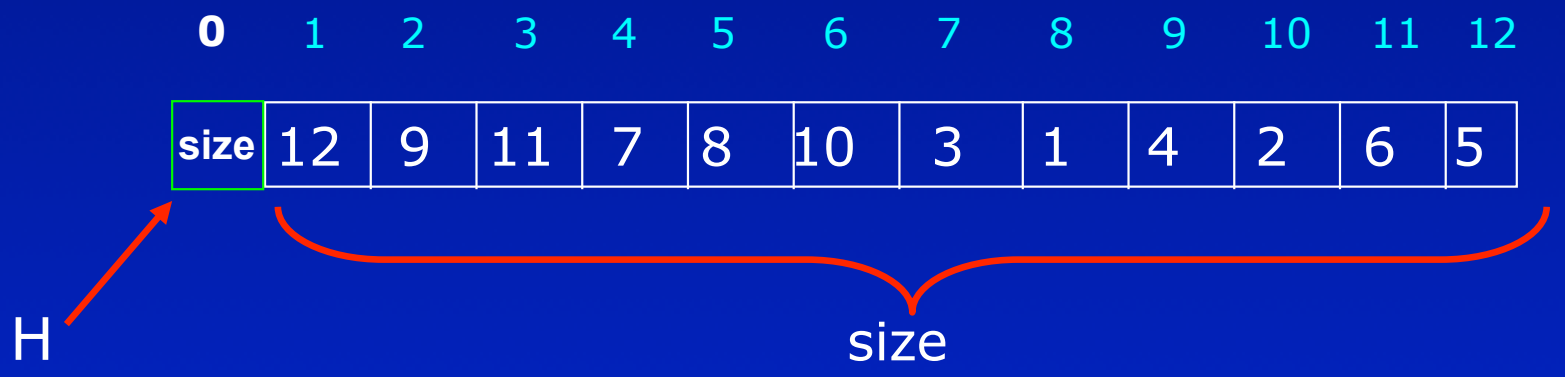
**max_heapify ( H, 1 )**

**max_heapify ( H, 3 )**

**max_heapify ( H, 7 )**



# build_heapify-Funktion




```
def  build_max_heap(H):
        H[0] = len(H)-1
        for i in range(heap_size(H)//2, 0, -1):
                max_heapify( H, i )
```

# Heapsort-Funktion

**1** Das Array mit den zu sortierenden Zahlen wird zuerst in ein Heap verwandelt.

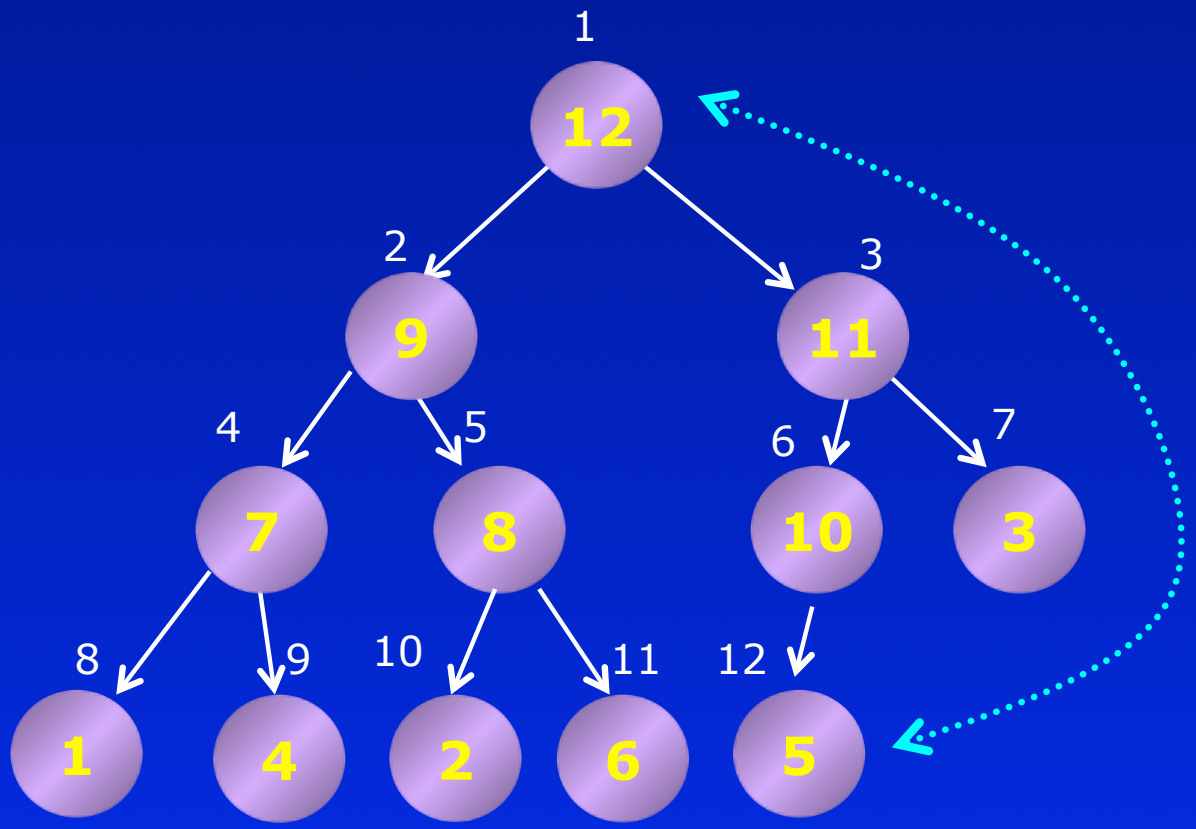
**2** Das Element an der Wurzel des Heaps wird gegen das letzte Element des Heaps vertauscht.

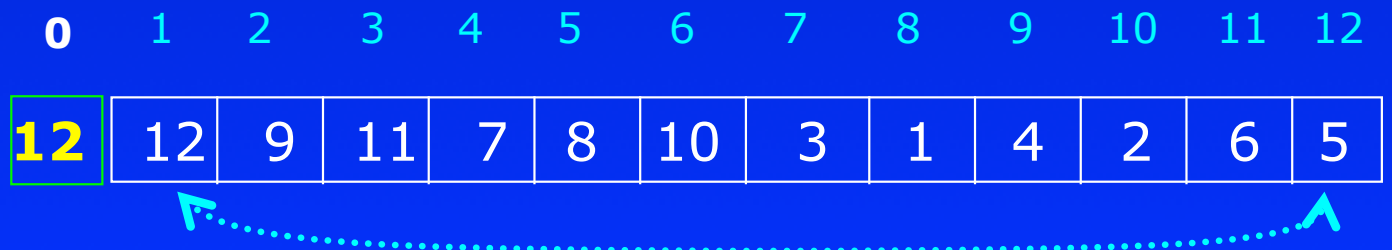
**3** Die Heap-Größe wird um eins dekrementiert.

**4** Die Funktion **max_heapify** wird mit der Position 1 (Wurzel) des Heaps aufgerufen.

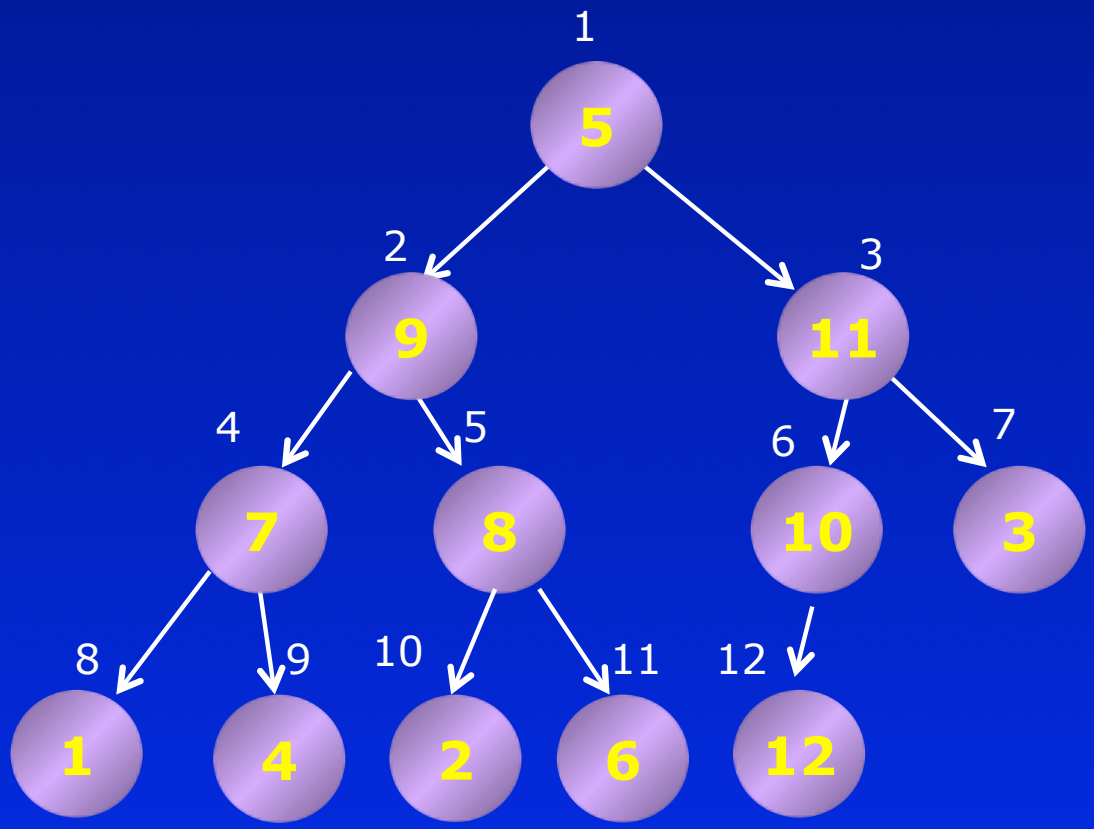
**5** Die Schritte **2** bis **4** werden wiederholt, solange der Heap größer oder gleich zwei ist.

# Heapsort-Funktion



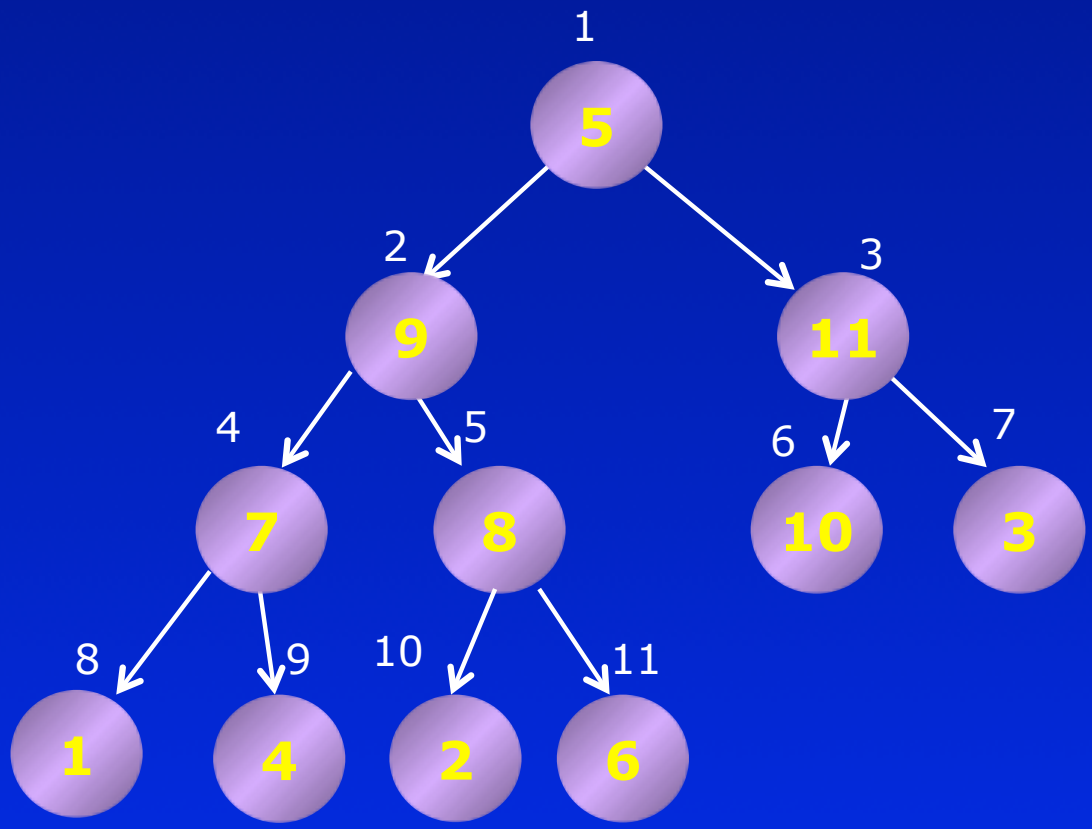
# Heapsort-Funktion




Erstes sortiertes Element

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **12** | 5 | 9 | 11 | 7 | 8 | 10 | 3 | 1 | 4 | 2 | 6 | 12 |

# Heapsort-Funktion



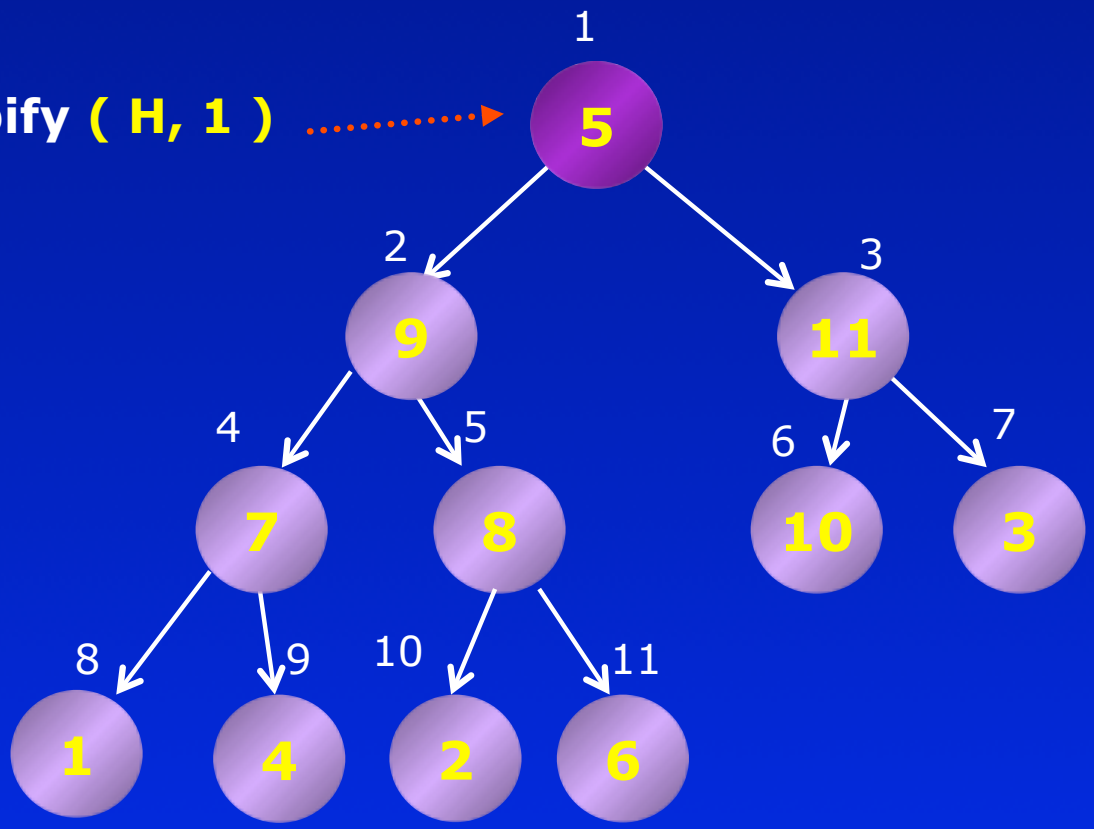

nicht mehr in dem Heap!

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 5 | 9 | 11 | 7 | 8 | 10 | 3 | 1 | 4 | 2 | 6 | 12 |

Die Heap-Größe wird um eins verkleinert

# Heapsort-Funktion

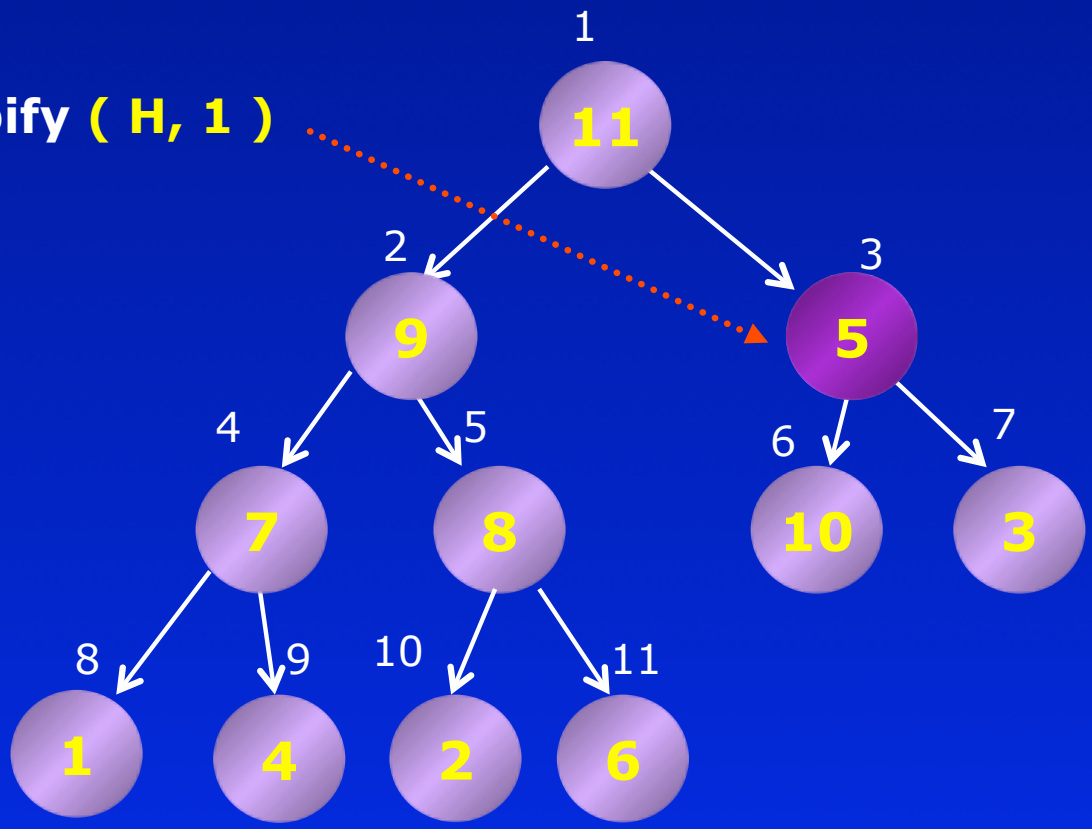
**max_heapify ( H, 1 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 5 | 9 | 11 | 7 | 8 | 10 | 3 | 1 | 4 | 2 | 6 | 12 |

# Heapsort-Funktion

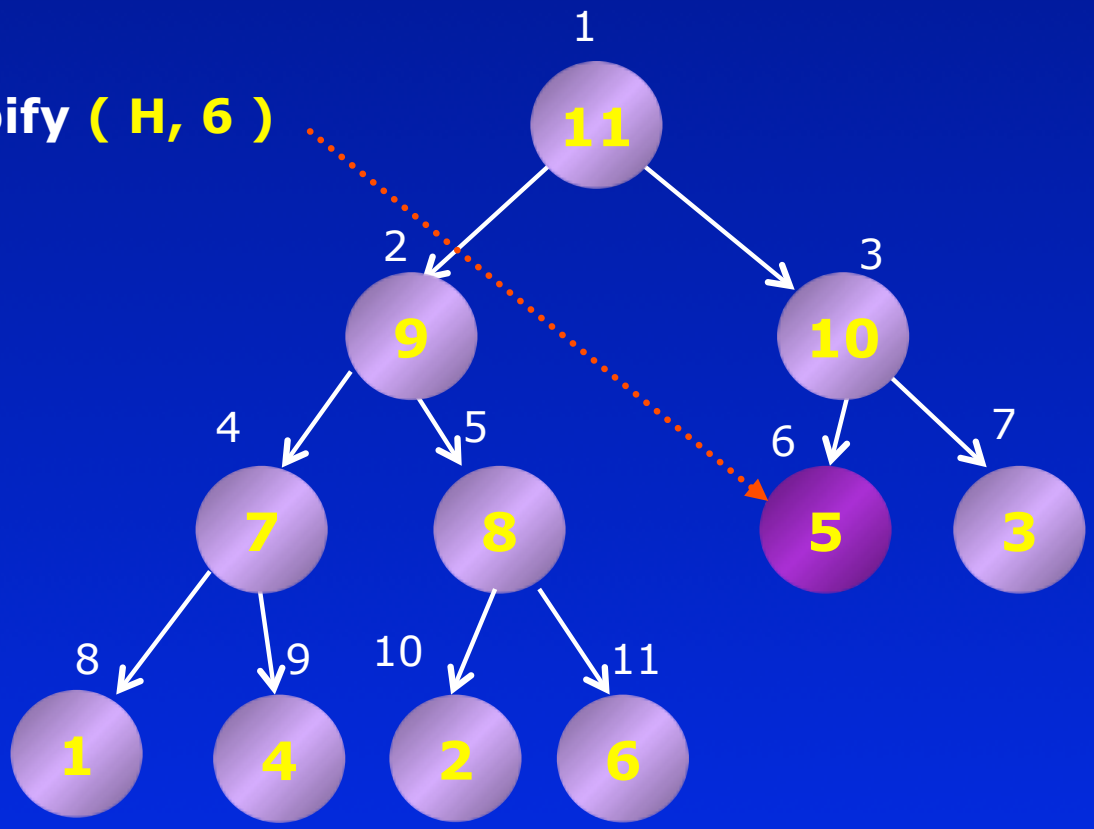
**max_heapify ( H, 1 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 11 | 9 | 5 | 7 | 8 | 10 | 3 | 1 | 4 | 2 | 6 | 12 |

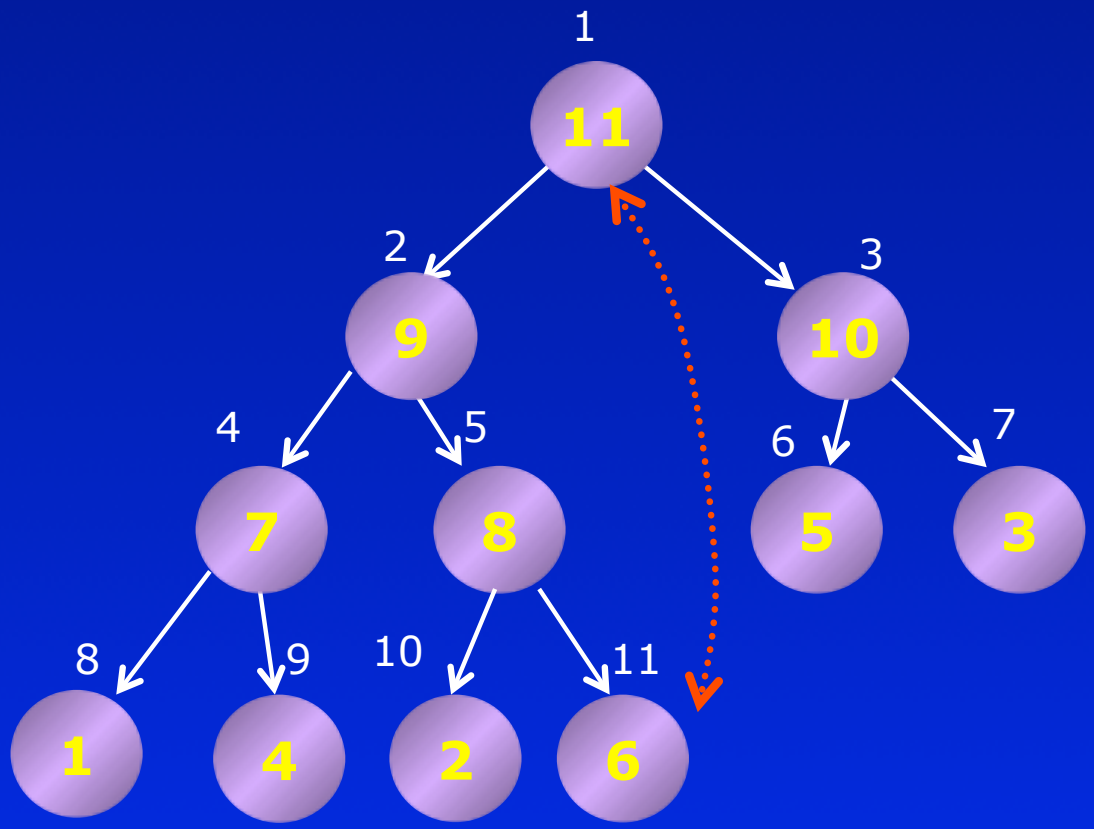
# Heapsort-Funktion

**max_heapify ( H, 6 )**



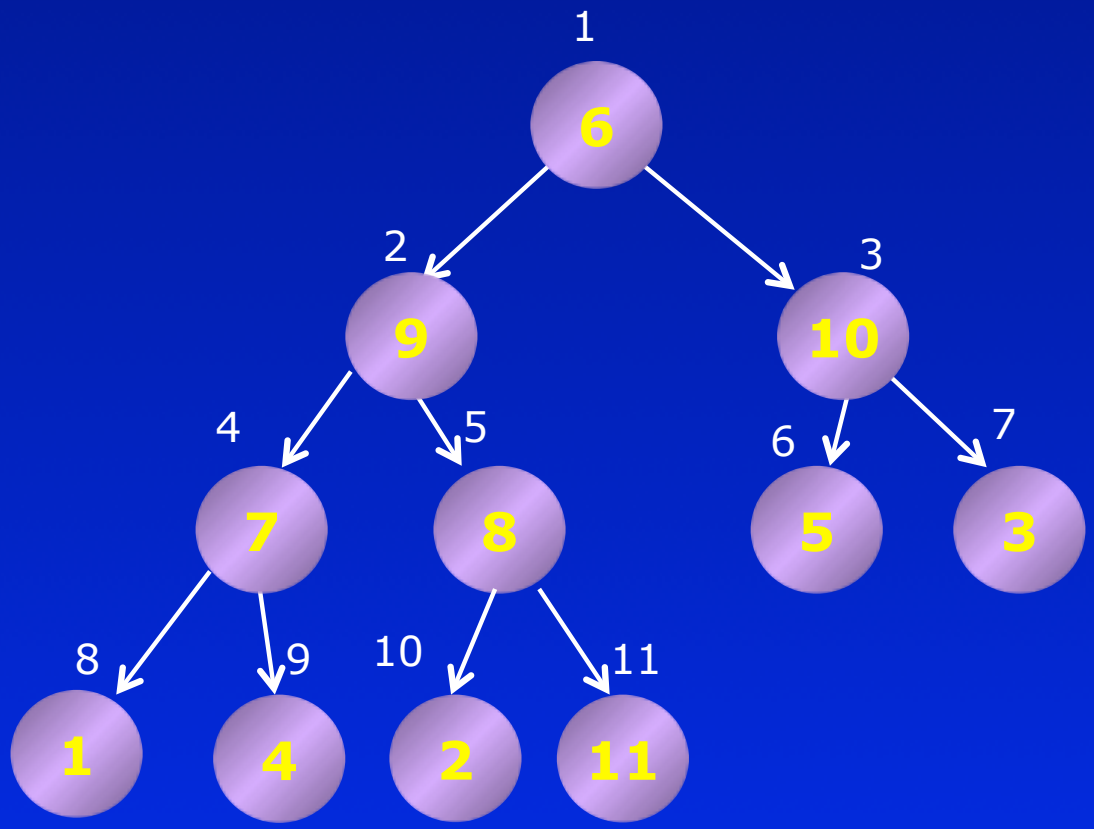

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 11 | 9 | 10 | 7 | 8 | 5 | 3 | 1 | 4 | 2 | 6 | 12 |

# Heapsort-Funktion



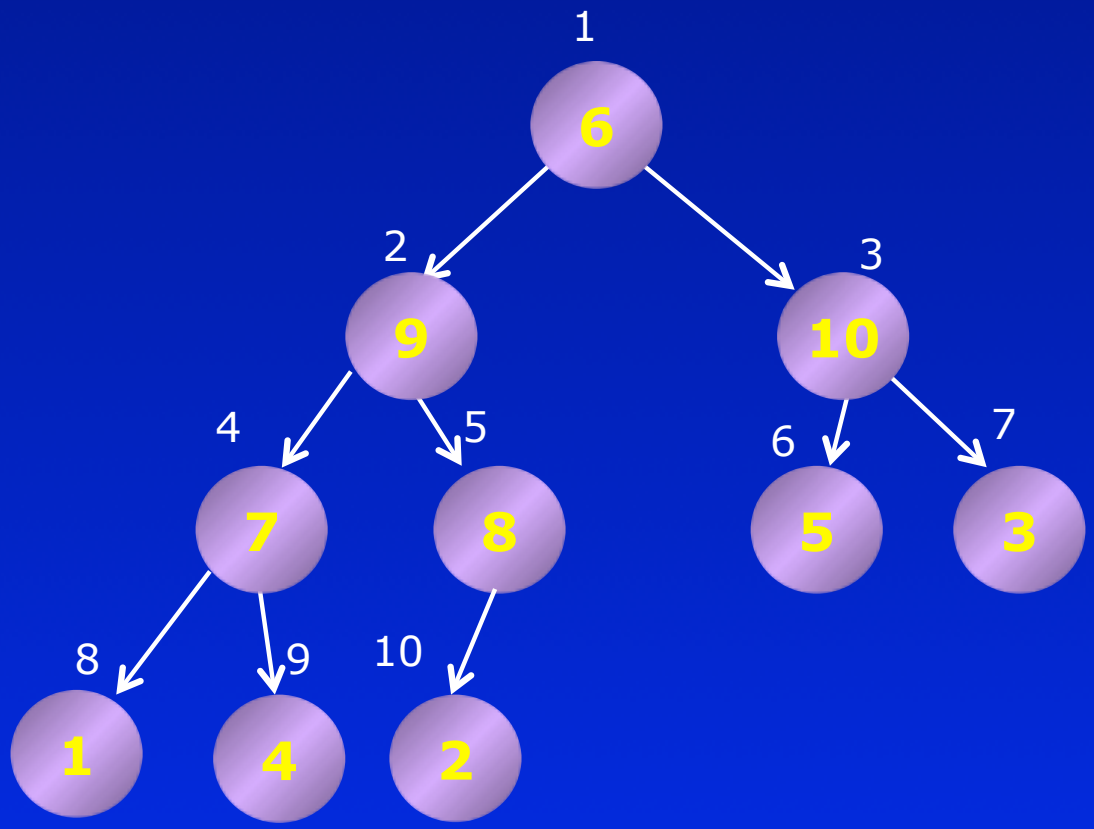

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 6 | 9 | 10 | 7 | 8 | 5 | 3 | 1 | 4 | 2 | 11 | 12 |

# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 6 | 9 | 10 | 7 | 8 | 5 | 3 | 1 | 4 | 2 | 11 | 12 |

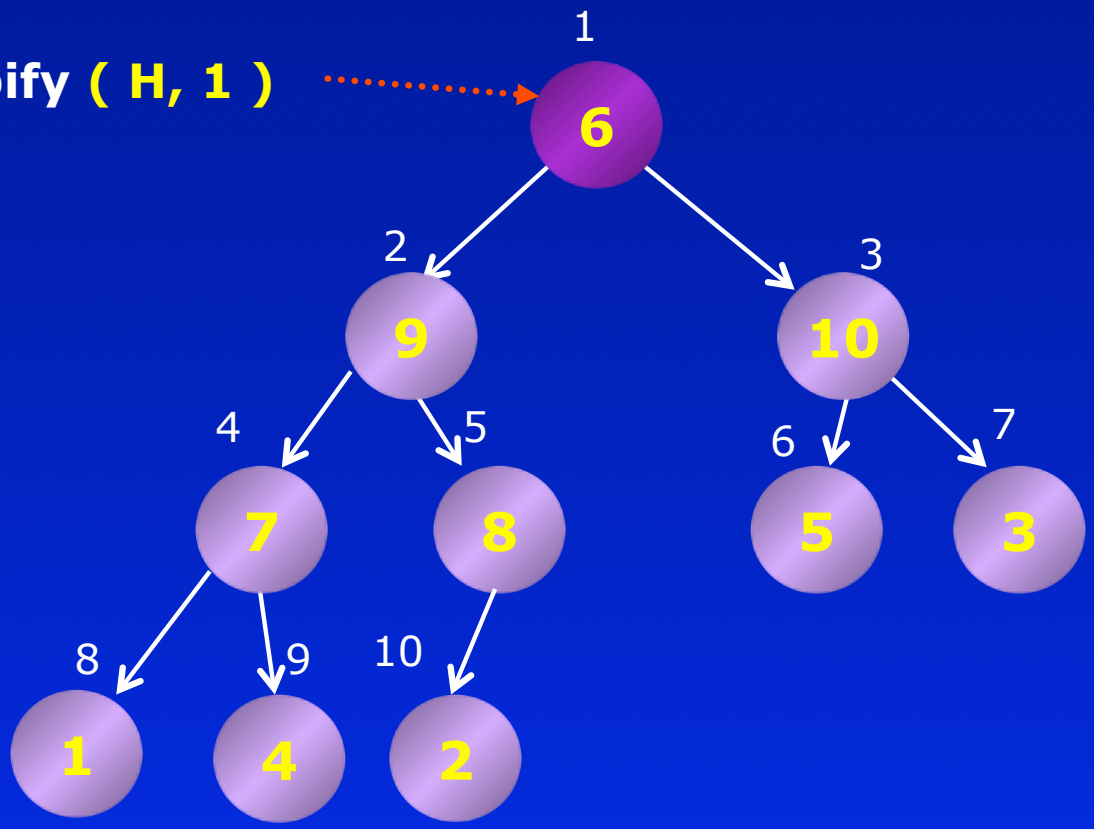
# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 6 | 9 | 10 | 7 | 8 | 5 | 3 | 1 | 4 | 2 | 11 | 12 |

# Heapsort-Funktion

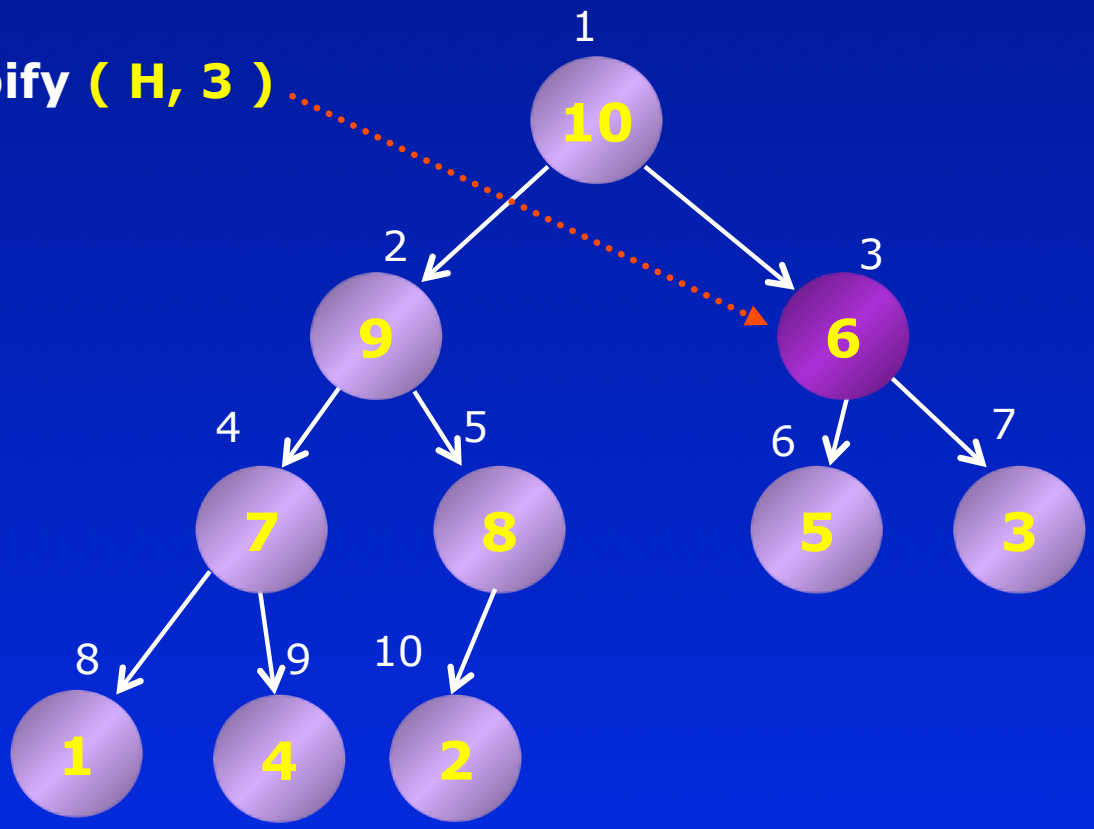
**max_heapify ( H, 1 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **10** | 6 | 9 | 10 | 7 | 8 | 5 | 3 | 1 | 4 | 2 | 11 | 12 |

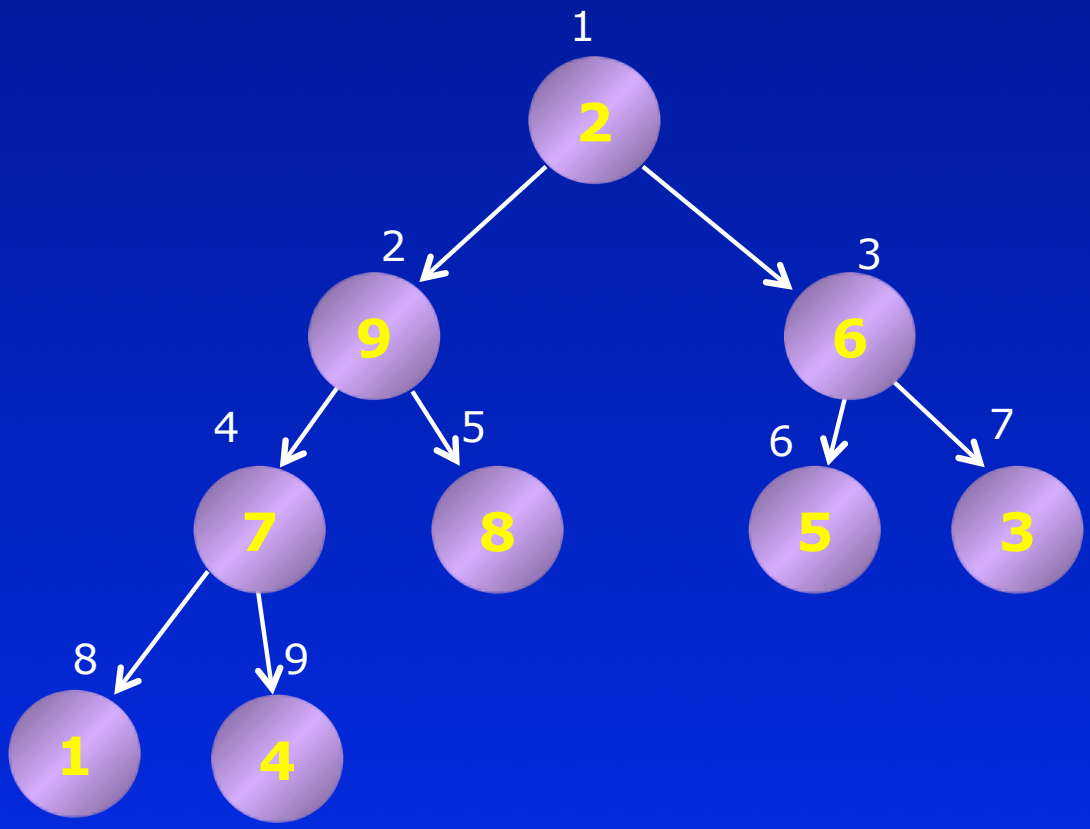
# Heapsort-Funktion

**max_heapify ( H, 3 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **10** | 10 | 9 | 6 | 7 | 8 | 5 | 3 | 1 | 4 | 2 | 11 | 12 |

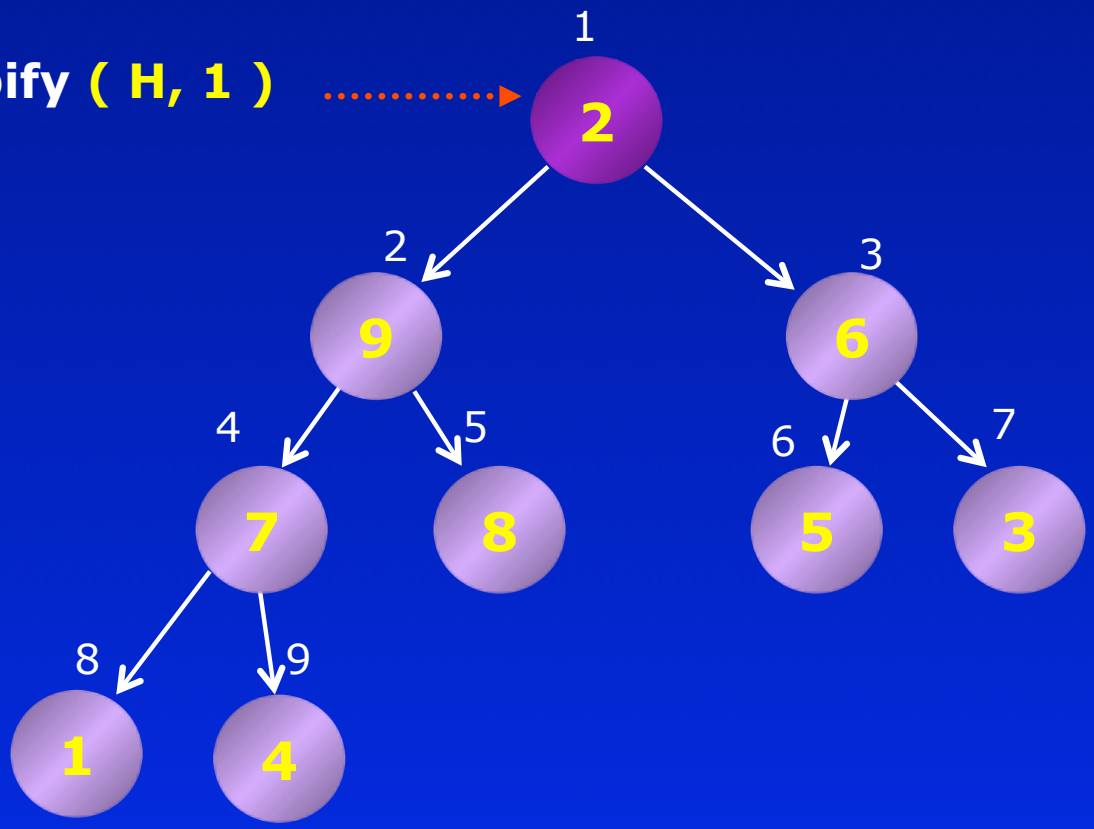
# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 2 | 9 | 6 | 7 | 8 | 5 | 3 | 1 | 4 | 10 | 11 | 12 |

# Heapsort-Funktion

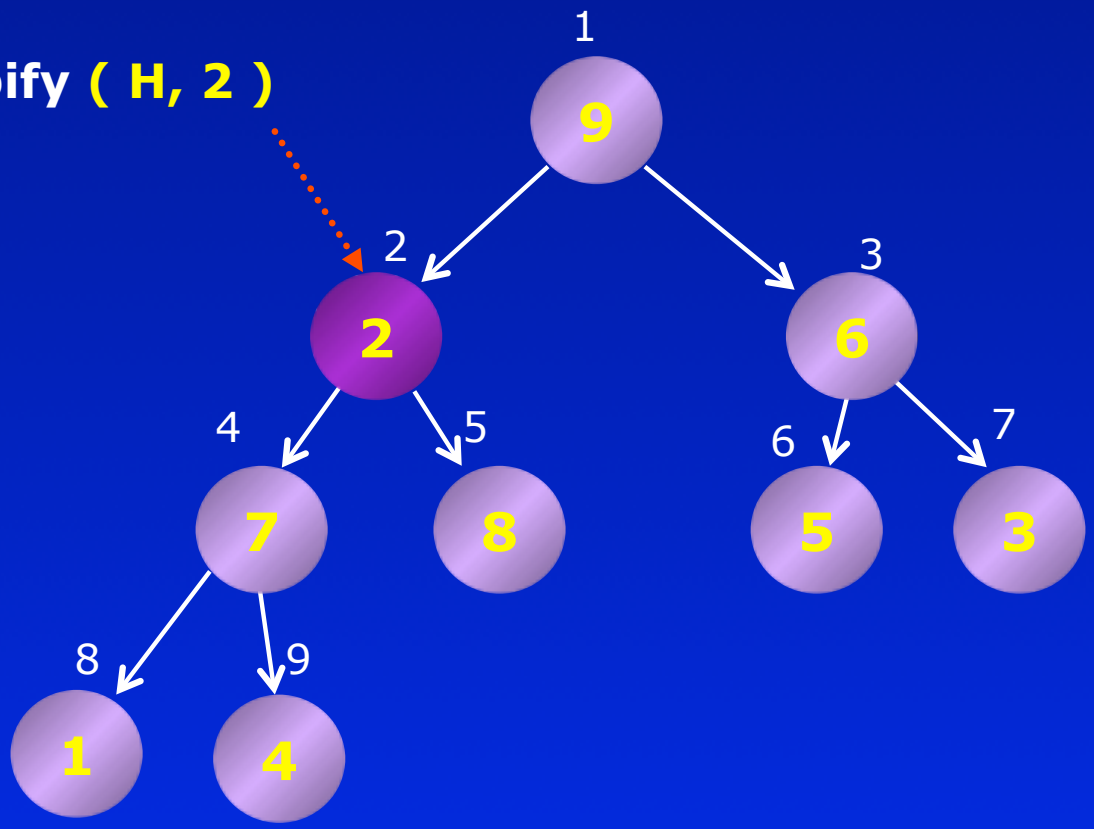
**max_heapify ( H, 1 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 2 | 9 | 6 | 7 | 8 | 5 | 3 | 1 | 4 | 10 | 11 | 12 |

# Heapsort-Funktion

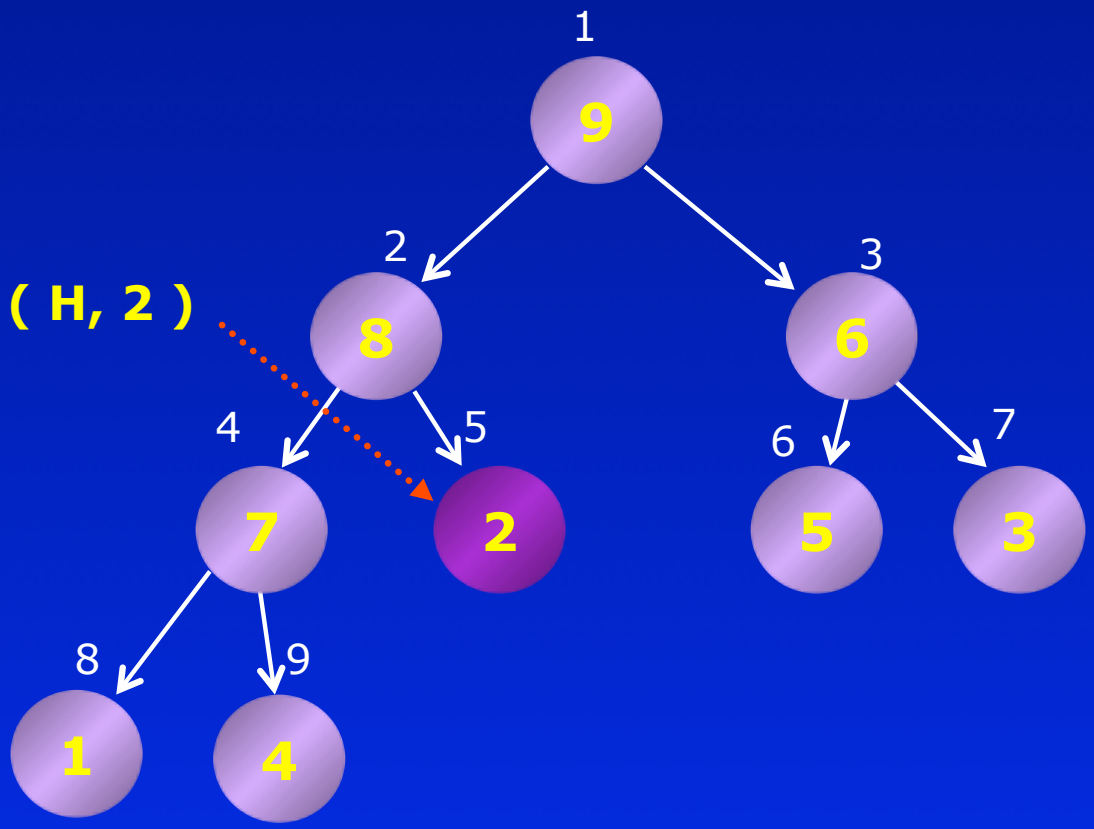
**max_heapify ( H, 2 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 2 | 6 | 7 | 8 | 5 | 3 | 1 | 4 | 10 | 11 | 12 |

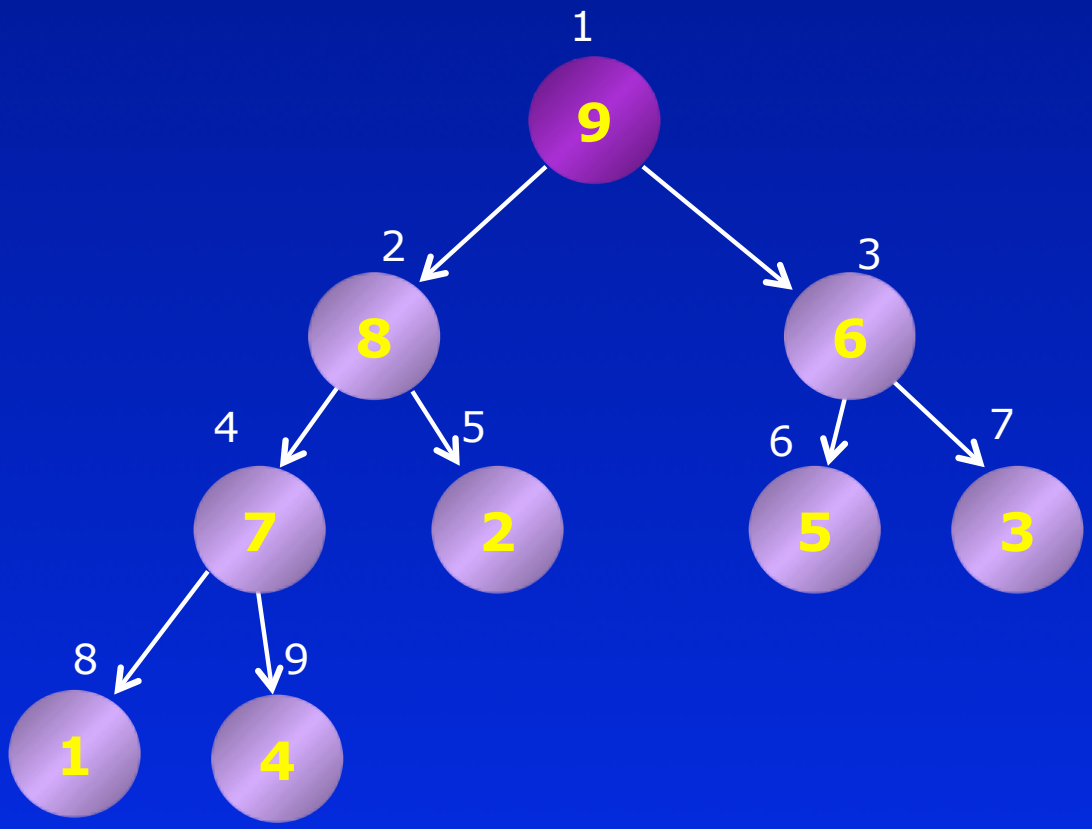
# Heapsort-Funktion

**max_heapify ( H, 2 )**



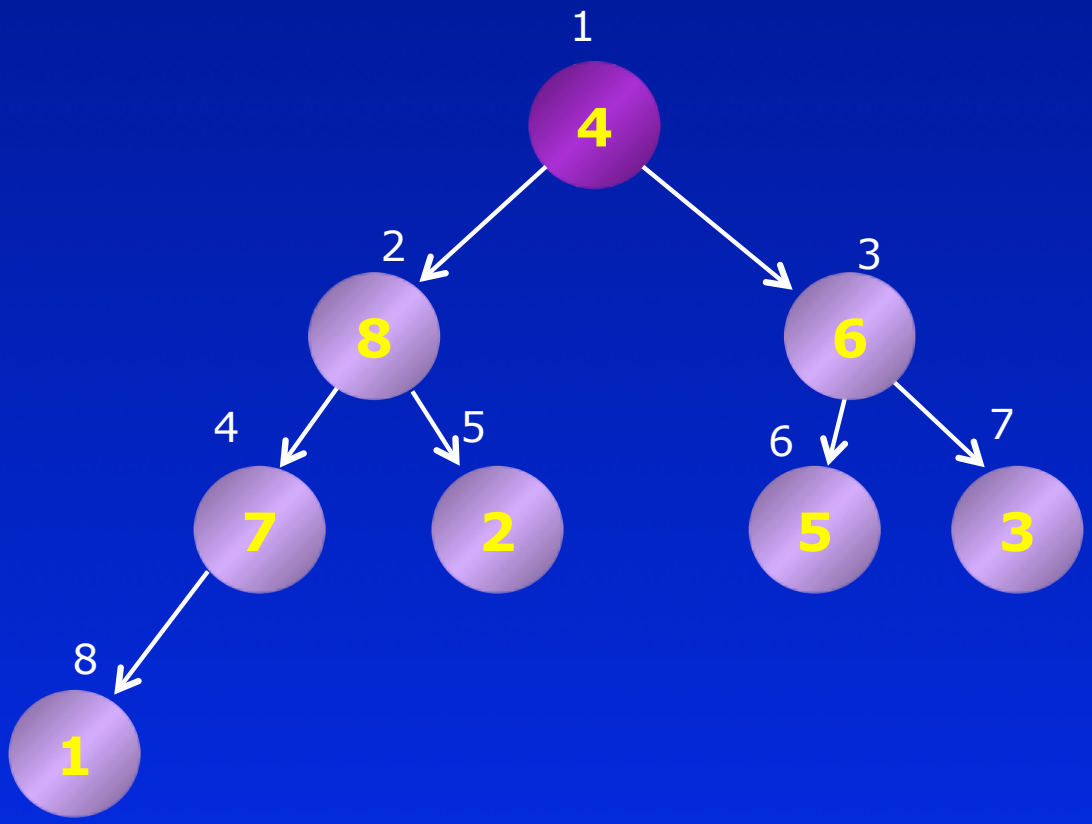

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 8 | 6 | 7 | 2 | 5 | 3 | 1 | 4 | 10 | 11 | 12 |

# Heapsort-Funktion



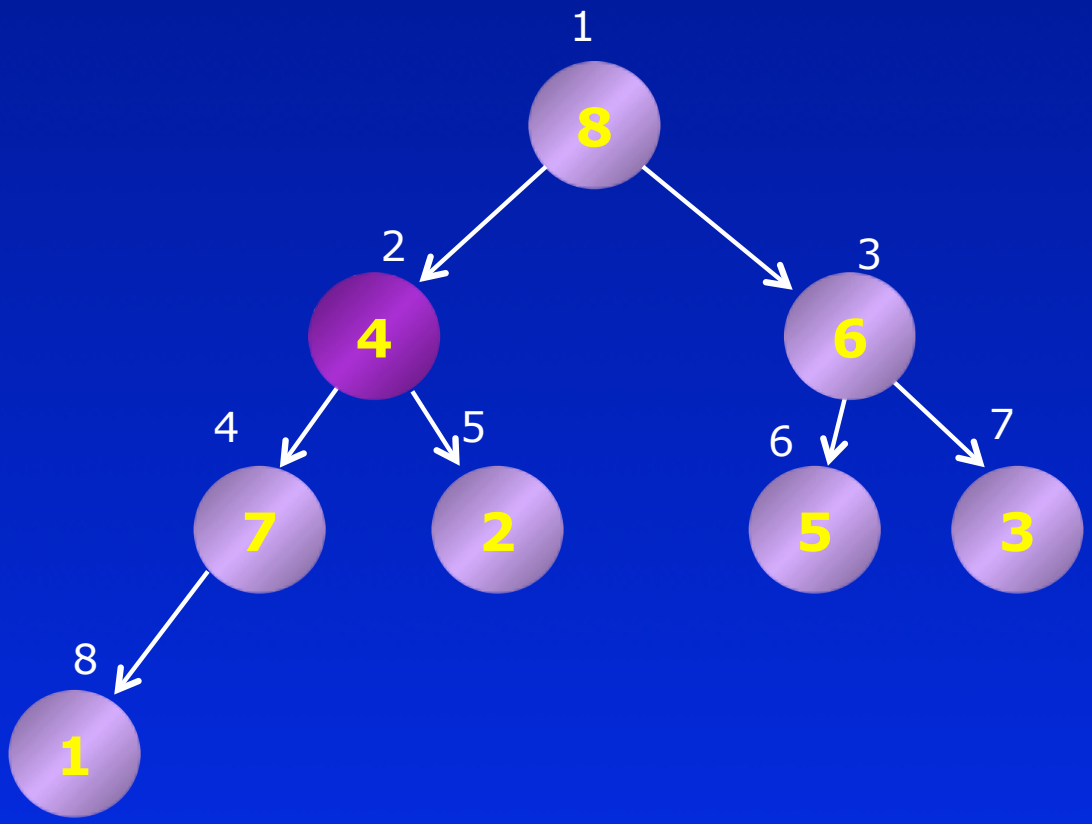

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 8 | 6 | 7 | 2 | 5 | 3 | 1 | 4 | 10 | 11 | 12 |

# Heapsort-Funktion



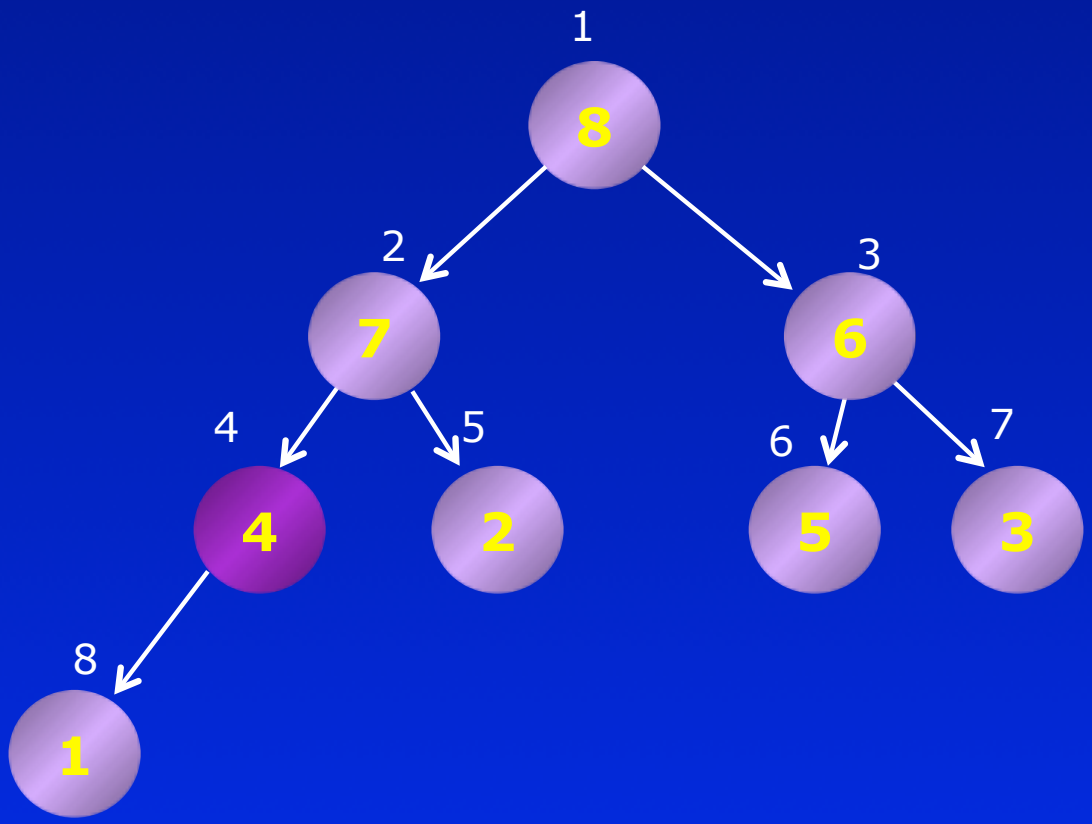

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 4 | 8 | 6 | 7 | 2 | 5 | 3 | 1 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



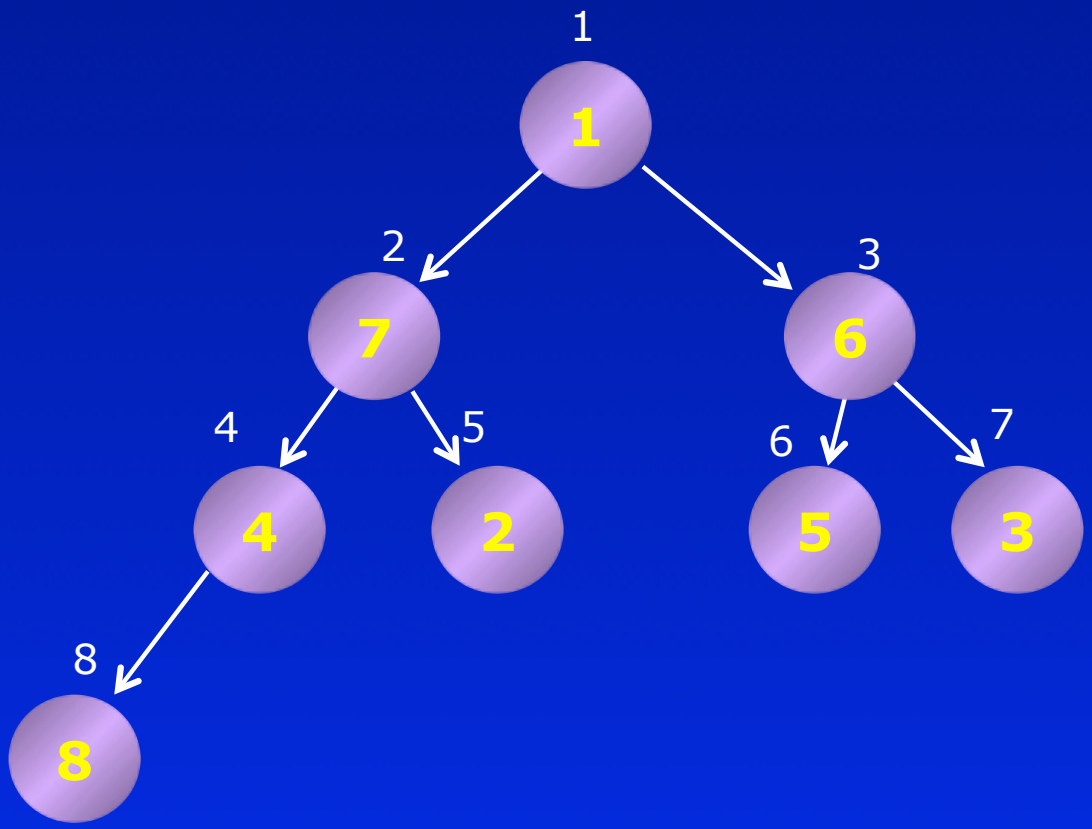

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 8 | 4 | 6 | 7 | 2 | 5 | 3 | 1 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 8 | 7 | 6 | 4 | 2 | 5 | 3 | 1 | 9 | 10 | 11 | 12 |

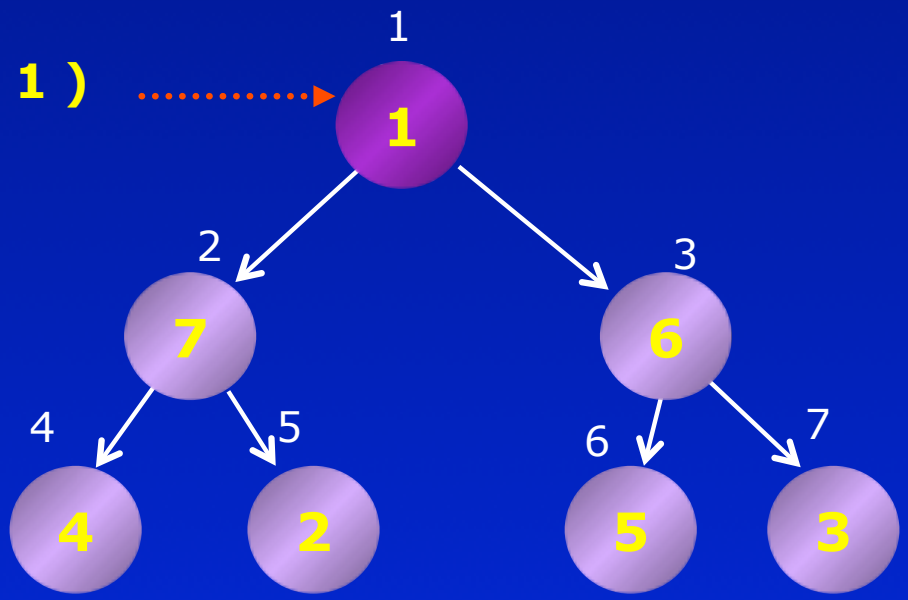
# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 1 | 7 | 6 | 4 | 2 | 5 | 3 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion

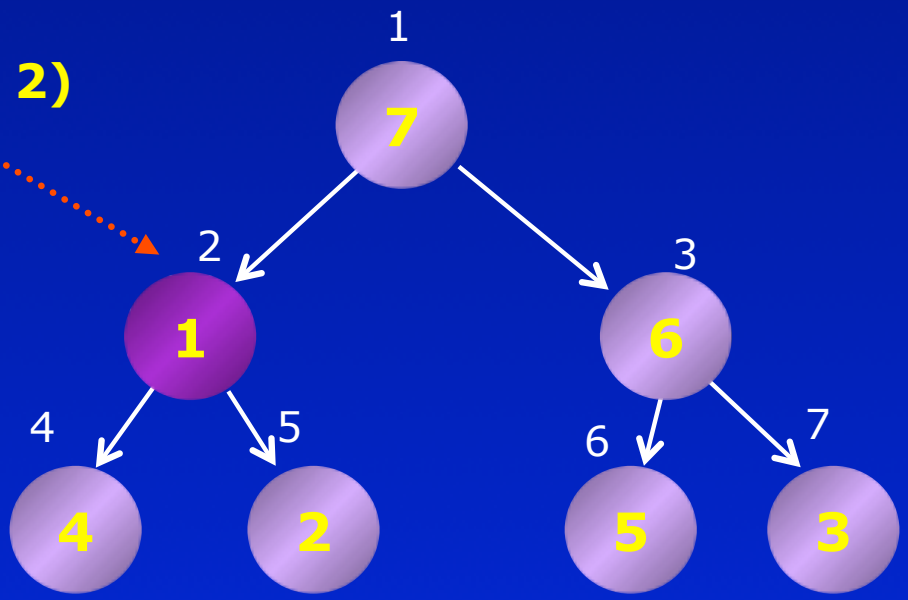
**max_heapify ( H, 1 )**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | 7 | 6 | 4 | 2 | 5 | 3 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion

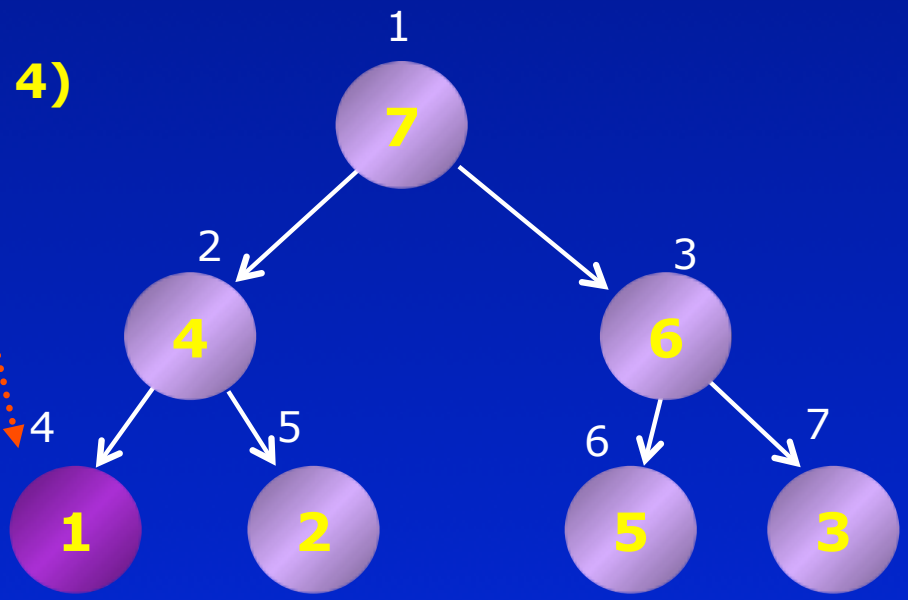
**max_heapify ( H, 2)**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 1 | 6 | 4 | 2 | 5 | 3 | 8 | 9 | 10 | 11 | 12 |

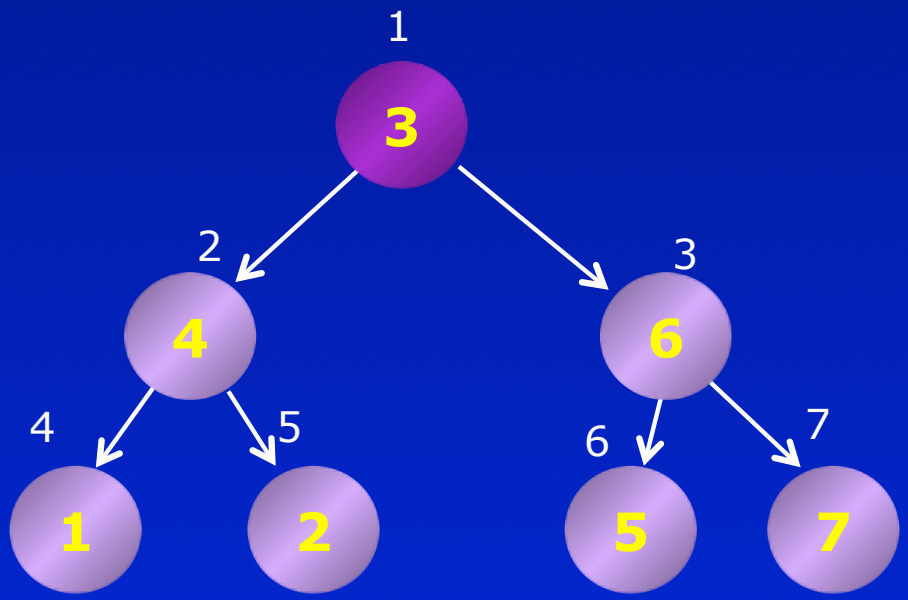
# Heapsort-Funktion

**max_heapify ( H, 4)**




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 4 | 6 | 1 | 2 | 5 | 3 | 8 | 9 | 10 | 11 | 12 |

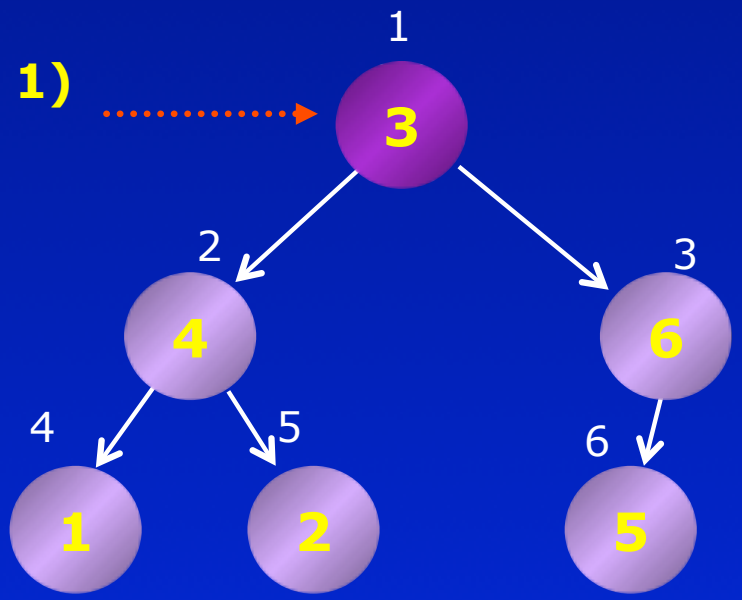
# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 3 | 4 | 6 | 1 | 2 | 5 | 7 | 8 | 9 | 10 | 11 | 12 |

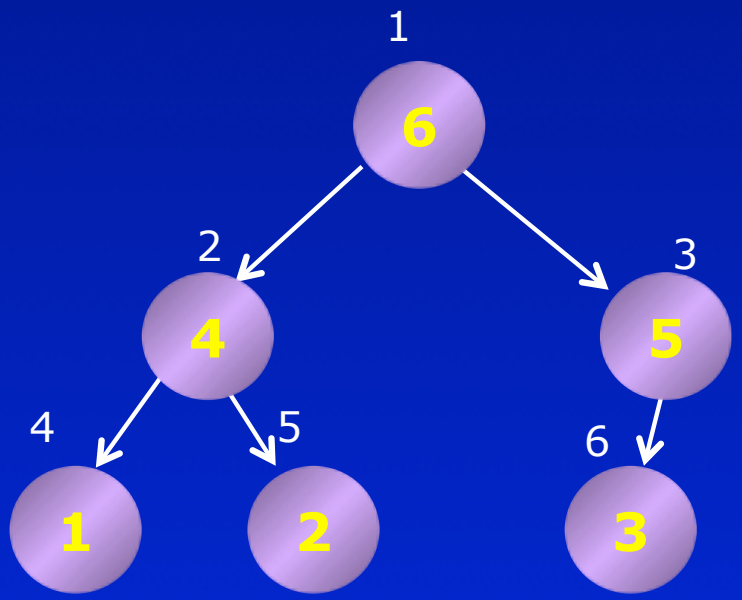
# Heapsort-Funktion

**max_heapify ( H, 1)**



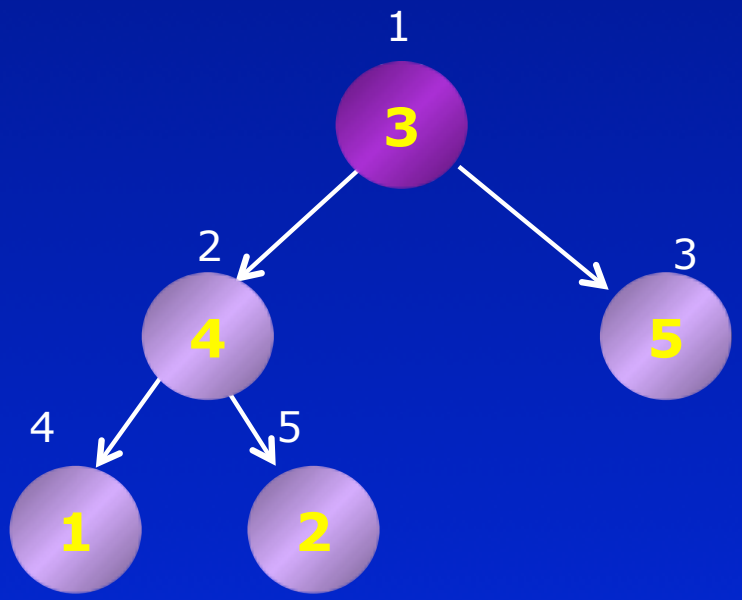

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 3 | 4 | 6 | 1 | 2 | 5 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



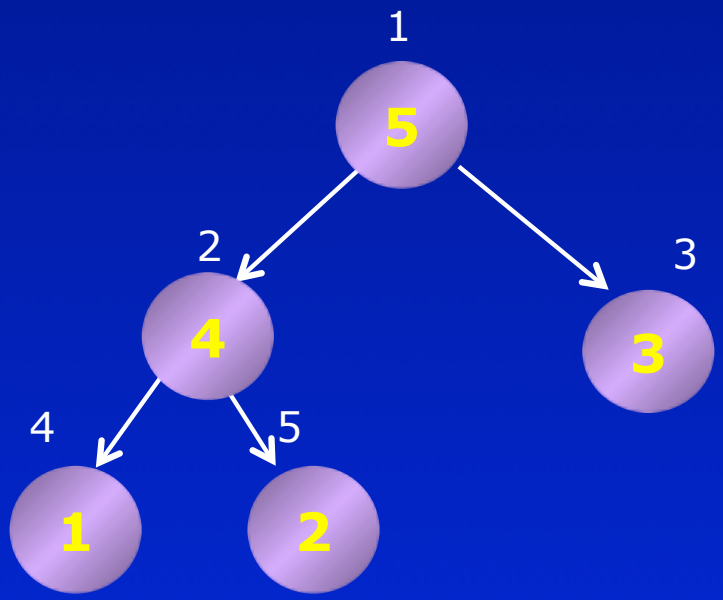

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 6 | 4 | 5 | 1 | 2 | 3 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



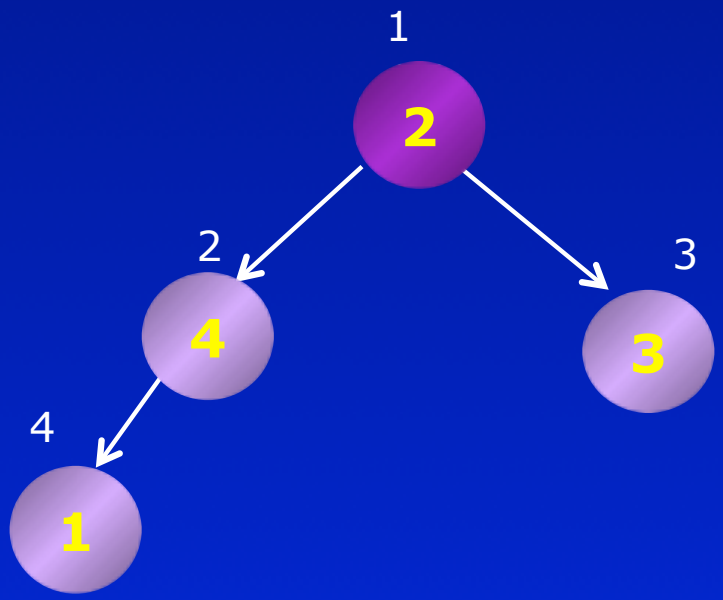

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 5 | 1 | 2 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



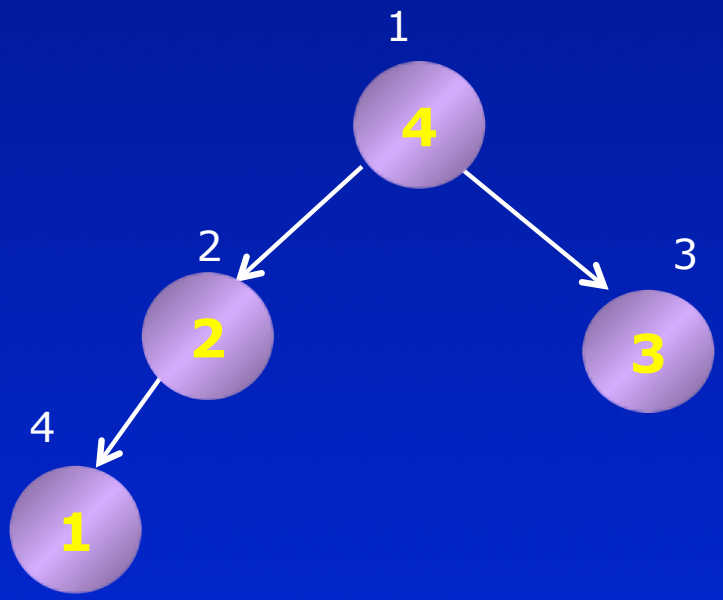

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 5 | 4 | 3 | 1 | 2 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



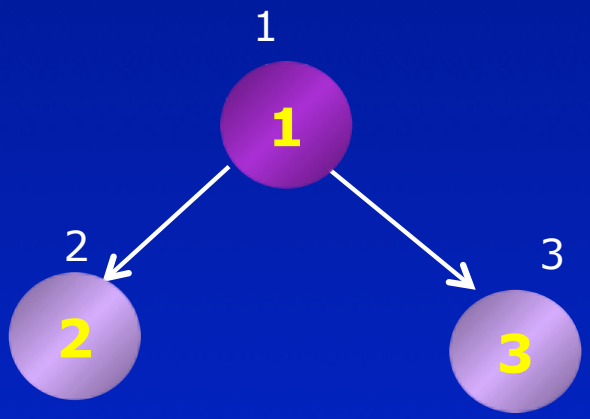

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 4 | 3 | 1 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



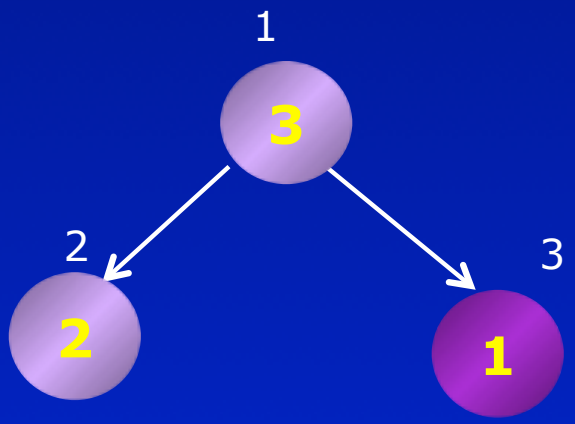

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 4 | 3 | 1 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



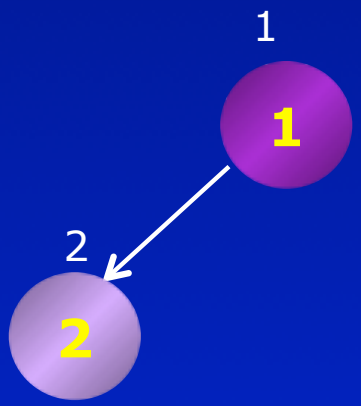

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **3** | 2 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



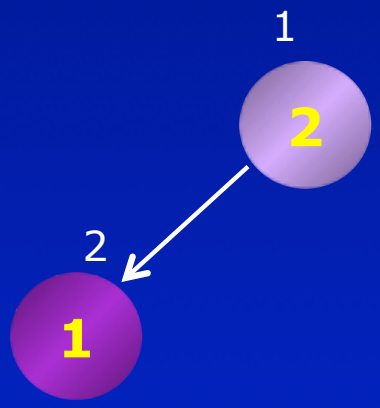

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion



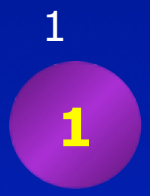

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 2 | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Heapsort-Funktion




# Heapsort-Funktion

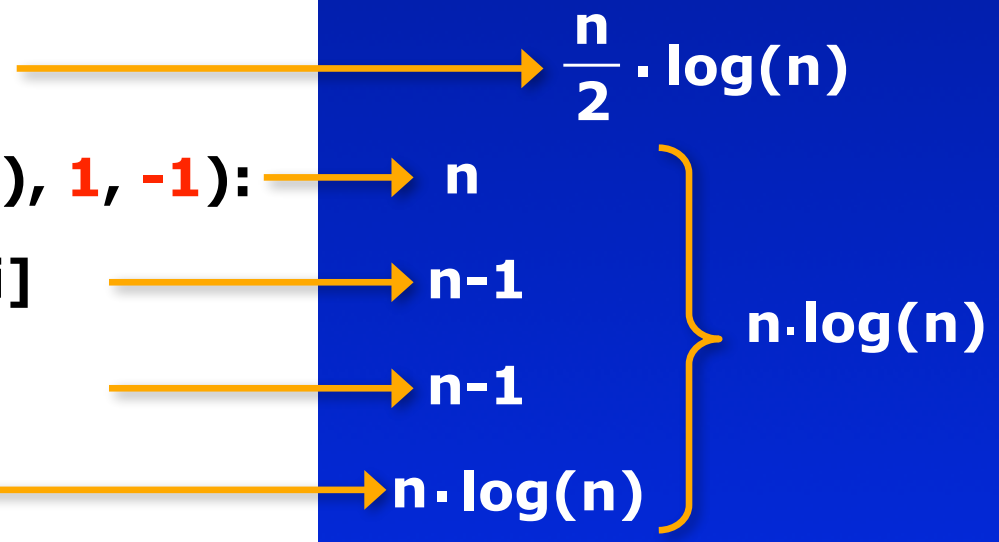
# Heapsort-Funktion

```
def heapsort(H):
    build_max_heap(H)
    for i in range( heap_size(H), 1, -1):
        H[i], H[1] = H[1], H[i]
        dec_heap_size(H)
        max_heapify( H, 1 )
    dec_heap_size(H)
```

# Heapsort-Funktion

Komplexität

```
def  heapsort(H):
        build_max_heap(H)
        for i in range( heap_size(H), 1, -1):
                H[i], H[1] = H[1], H[i]
                dec_heap_size(H)
                max_heapify( H, 1 )
        dec_heap_size(H)
```




Zeitkomplexität = $O(n \cdot \log(n))$