Freie Universität Berlin

Objektorientierte Programmierung

Sortieren (Teil 2)

SoSe 2018

Oliver Wiese

Sortieralgorithmen

Vergleichs-
Algorithmen

quadratisch

Bubble-Sort $\mathbf{O}(n^2)$

Insert-Sort $\mathbf{O}(n^2)$

Selection-Sort $\mathbf{O}(n^2)$

Quicksort

Mergesort

Heap-Sort

Counting-Sort

Radix-Sort

Bucket-Sort

Teile und Herrsche

"Divide und Conquer"

Viele Probleme lassen sich nicht mit trivialen Schleifen lösen und haben gleichzeitig den Vorteil, dass eine rekursive Lösung keine überflüssigen Berechnungen verursacht.

Solche Probleme lassen sich in Teilprobleme zerlegen, deren Lösung keine überlappenden Berechnungen beinhalten.

Lösungsschema:

Divide:

Teile ein Problem in zwei oder mehrere kleinere ähnliche Teilprobleme, die (rekursiv) isoliert behandelt werden können.

Conquer:

Löse die Teilprobleme auf dieselbe Art (rekursiv).

Merge:

Füge die Teillösung zur Gesamtlösung zusammen.

Quick-Sort-Algorithmus

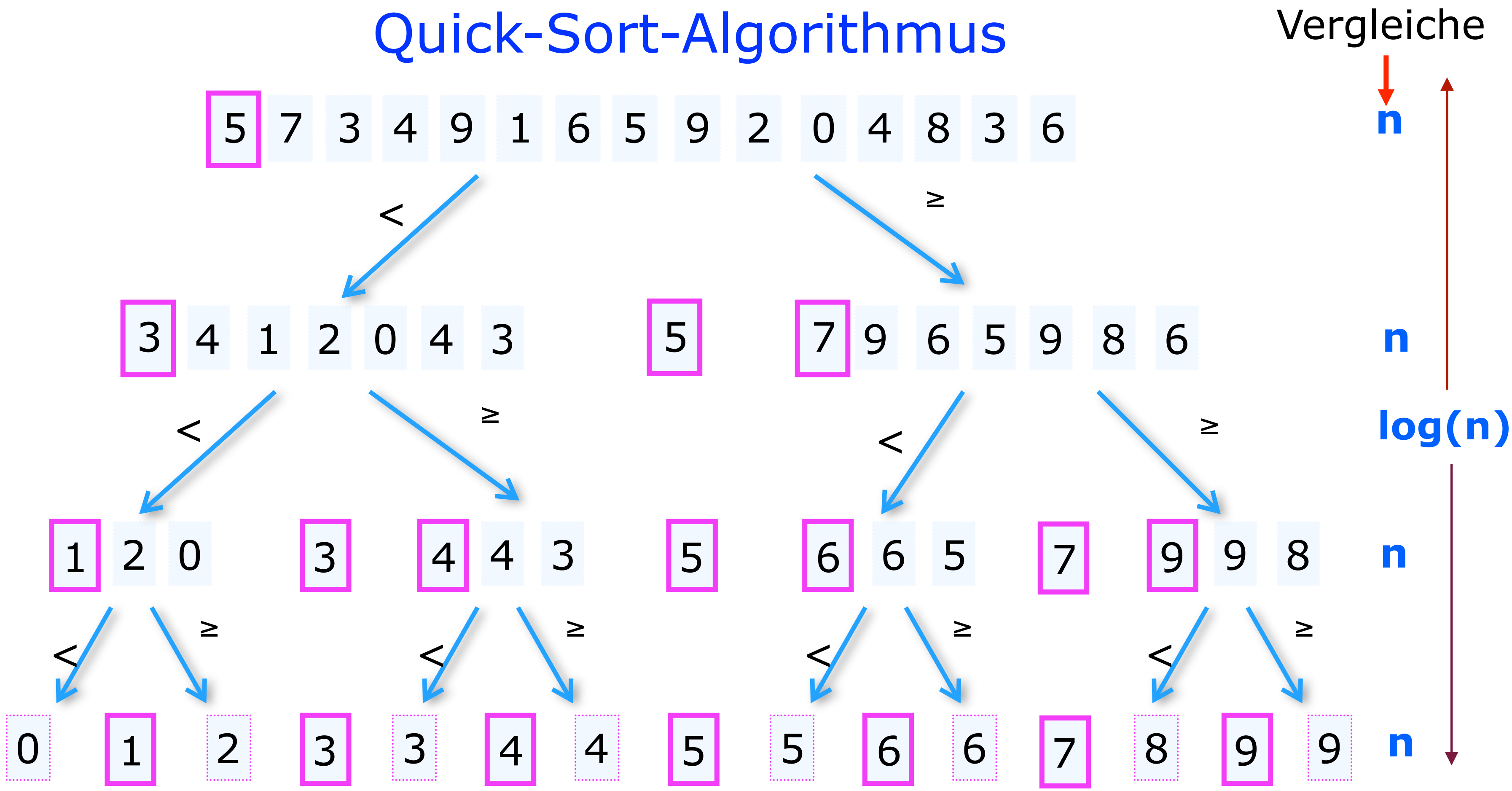
Grundidee:

1) Ein Element (**Pivot**) aus dem Array wird gewählt

2) Alle Zahlen des Arrays werden mit dem Pivot-Element verglichen und während des Vergleichsdurchlaufs in zwei Bereiche umorganisiert (**Partitionierung**). Der erste Bereich beinhaltet die Zahlen, die kleiner als das Pivot-Element sind und der zweite alle, die größer oder gleich sind. Am Ende des Durchlaufs wird das Pivot-Element zwischen beider Bereichen positioniert.

3) Nach jeder Partitionierung wird der Quicksort-Algorithmus rekursiv mit beiden Teilbereichen ausgeführt (solange die Teilbereiche mehr als ein Element beinhalten).

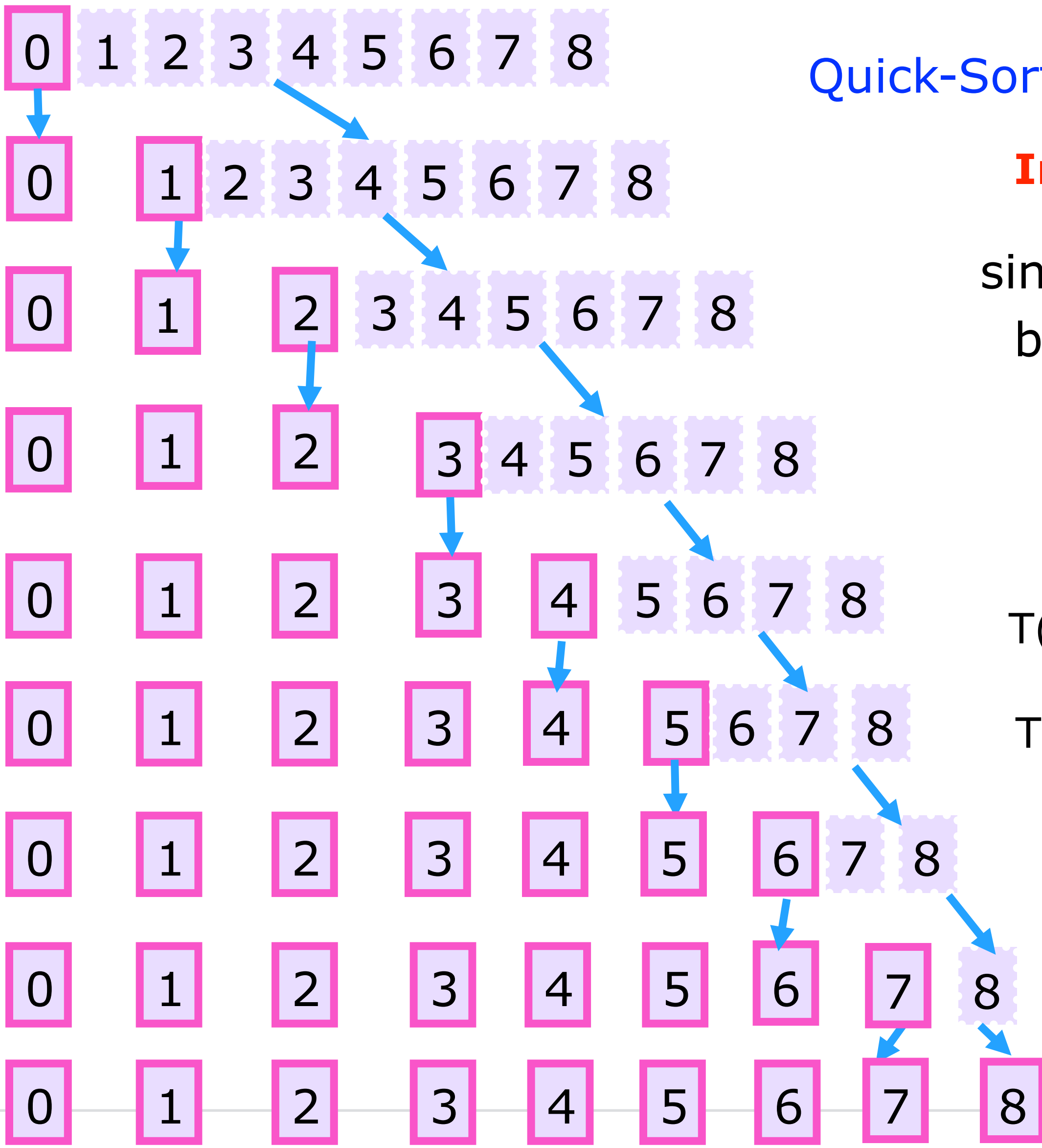
Quick-Sort-Algorithmus



Der Quicksort-Algorithmus funktioniert am besten, wenn die Teillisten fast gleich gross sind.

im besten Fall!

T(n) = n (log (n))



Quick-Sort-Algorithmus

Im schlimmsten Fall!

sind alle Elemente
bereits sortiert.

Anzahl der Vergleiche

$$T(n) = n + (n-1) + \ldots 2 + 1$$

$$T(n) = c_1 n^2 + c_2 n$$

$$T(n) = \mathbf{O}(\ n^2\ )$$

Quick-Sort-Algorithmus naiv

```
def quick_sort(seq):
    if len(seq)>1:
        q1 = [s for s in seq[1:] if s<seq[0]]
        q2 = [s for s in seq[1:] if s>=seq[0]]
        return quick_sort(q1) + [seq[0]] + quick_sort(q2)
    else:
        return seq
```

Speicherverbrauch?   Komplexität der Verkettungsfunktion (+)?

Wenn die Zahlen sortiert sind, stürzt das Programm bereits bei kleinen Arrays ab.

Freie Universität Berlin

Quicksort-Algorithmus

imperativ!

Rekursive Implementierung

```
def quicksort (A, low, high ):
    if  low<high:
        m = partition(A, low, high )
        quicksort ( A,  low,  m-1 )
        quicksort ( A,  m+1,  high )
```

	low														high
A	5	7	3	4	9	1	6	5	9	2	0	4	8	3	6

Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

low high

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

pivot

Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort-Algorithmus

Freie Universität Berlin

i

low high

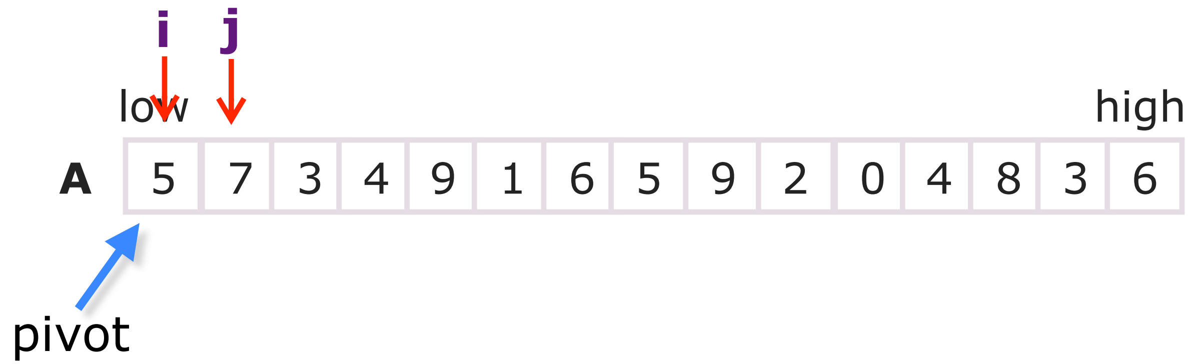
A	5	7	3	4	9	1	6	5	9	2	0	4	8	3	6

pivot

Sortieren am Ort

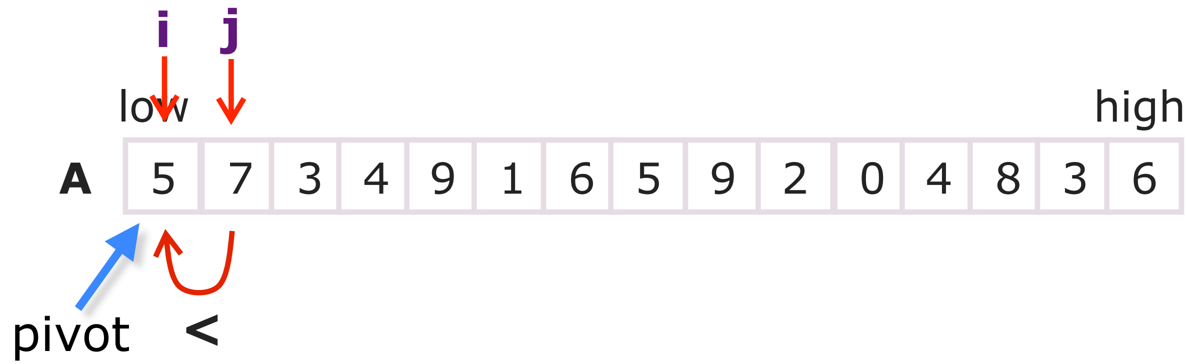
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort-Algorithmus



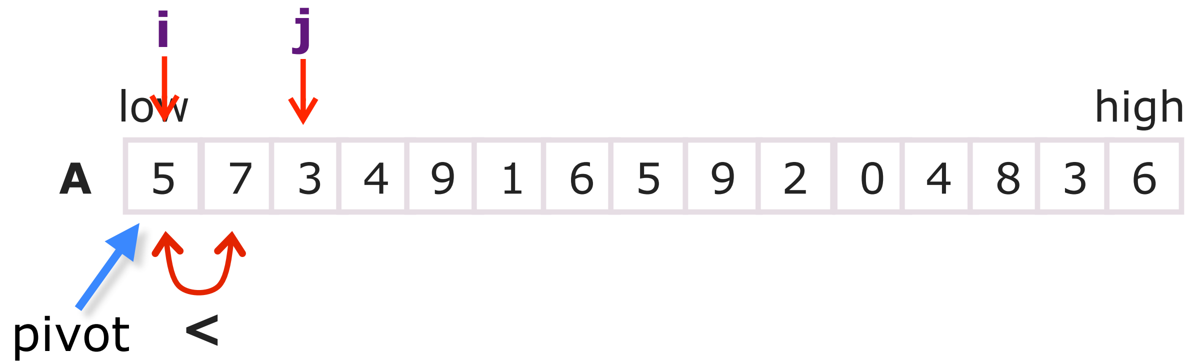
Sortieren am Ort

```python
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



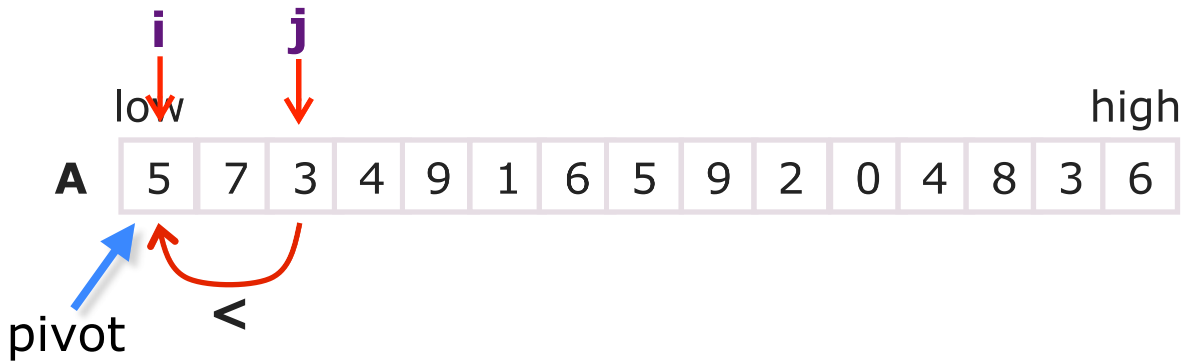
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



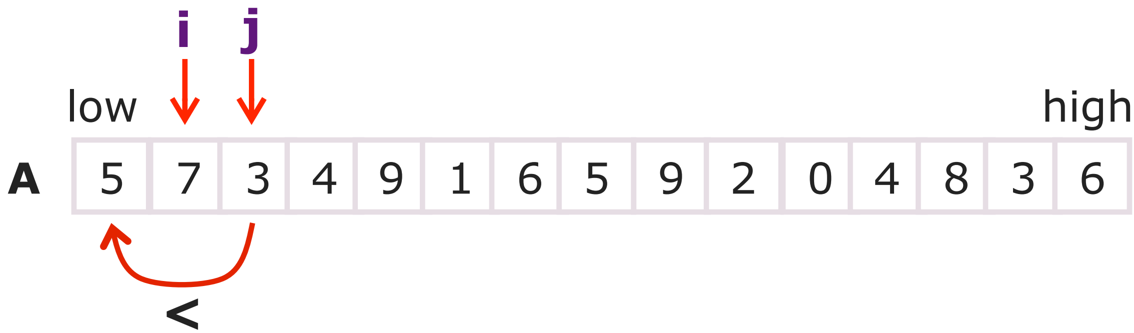
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



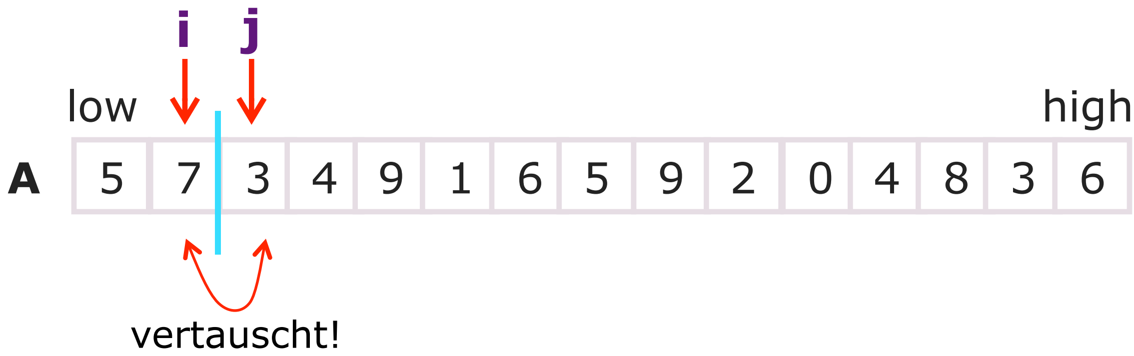
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



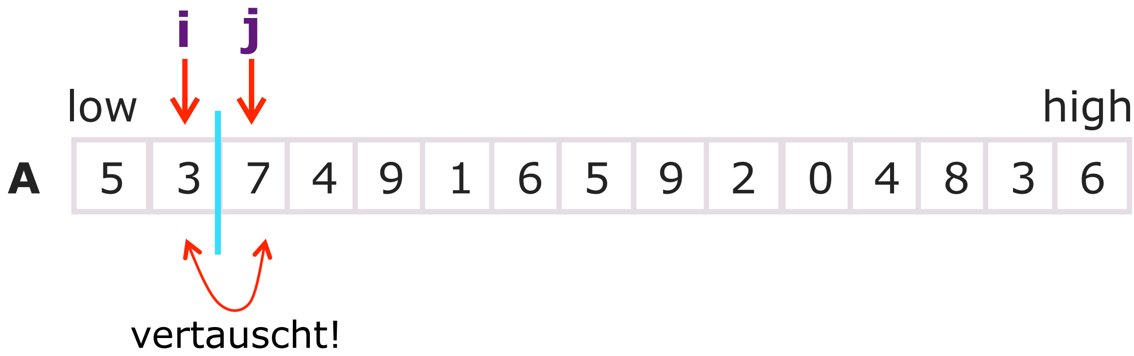
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



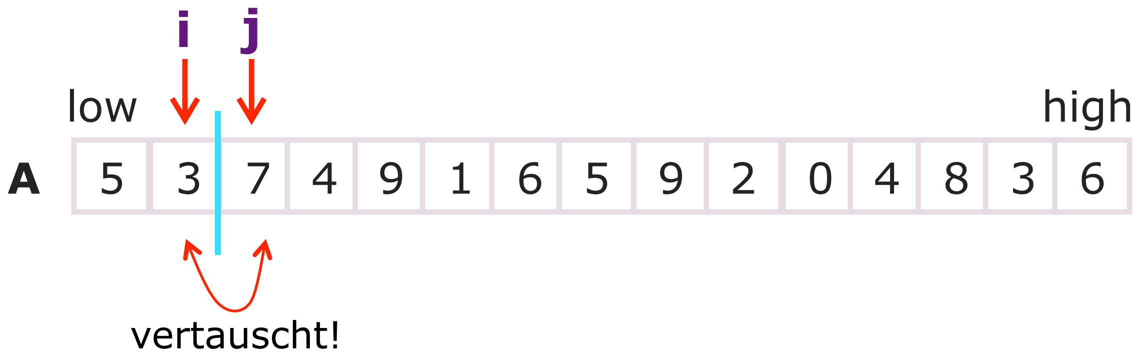
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



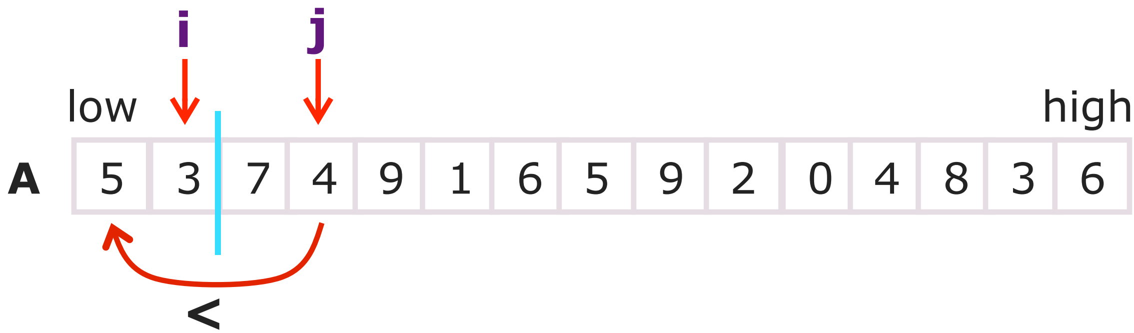
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

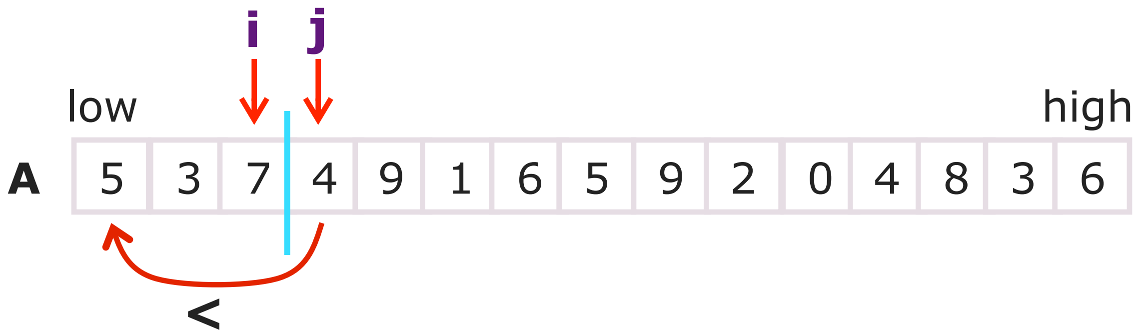
		i		j											
	low														high
A	5	3	7	4	9	1	6	5	9	2	0	4	8	3	6

Sortieren am Ort

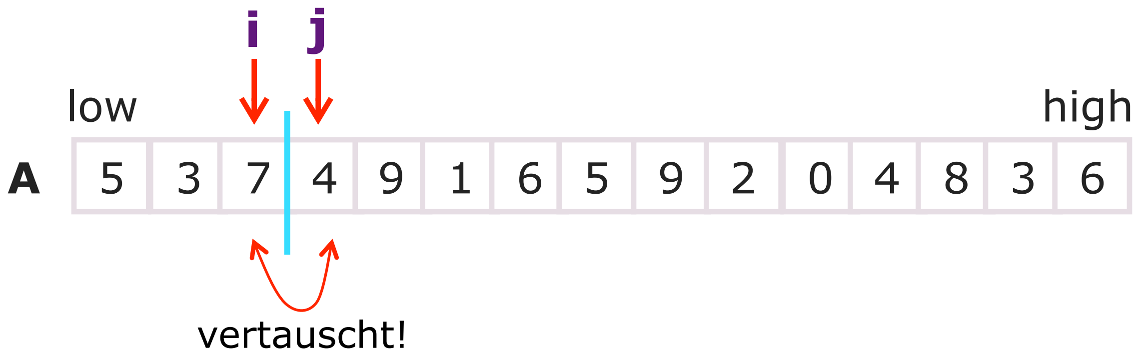
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



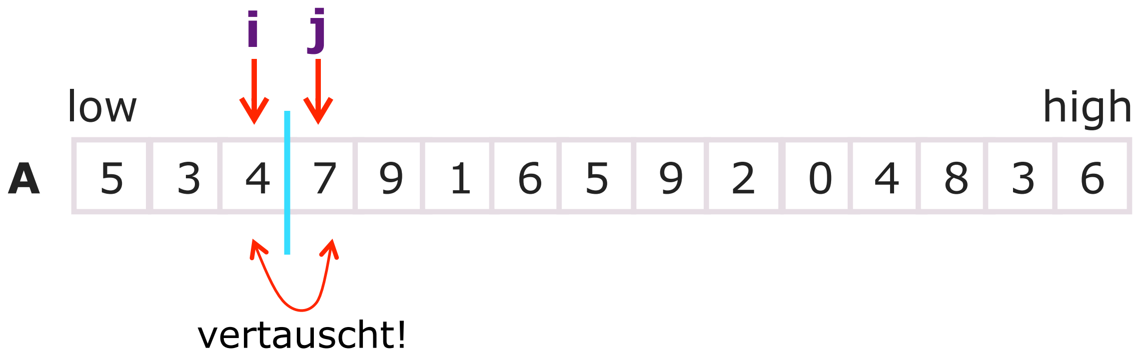

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



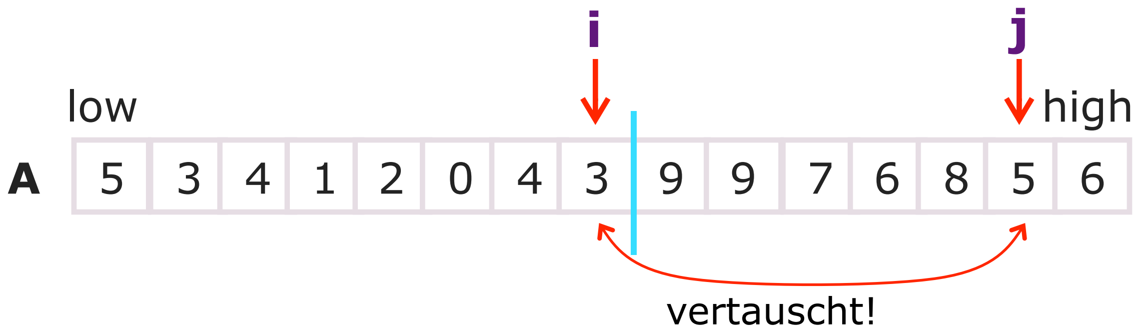

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



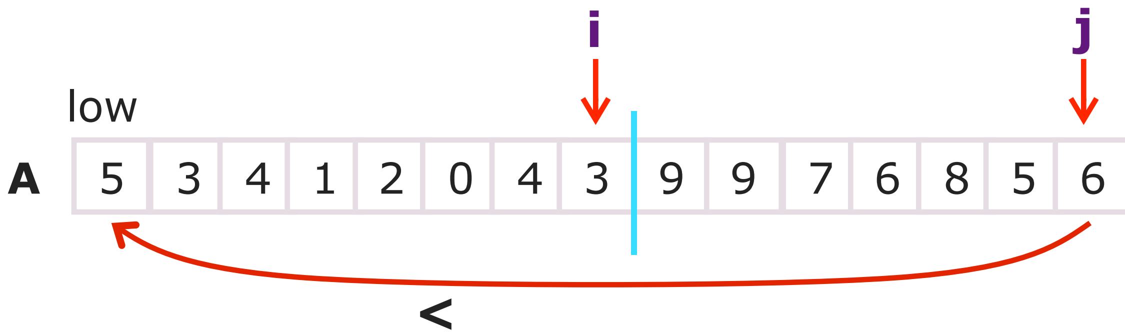

```python
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



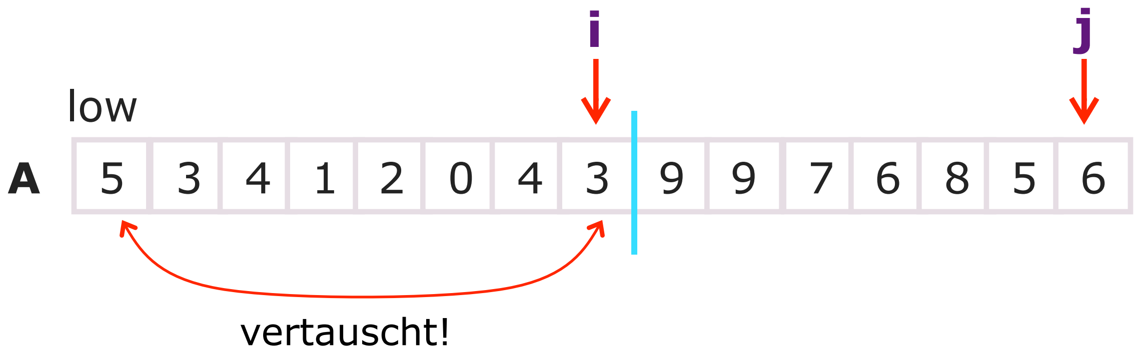

```python
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```




```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

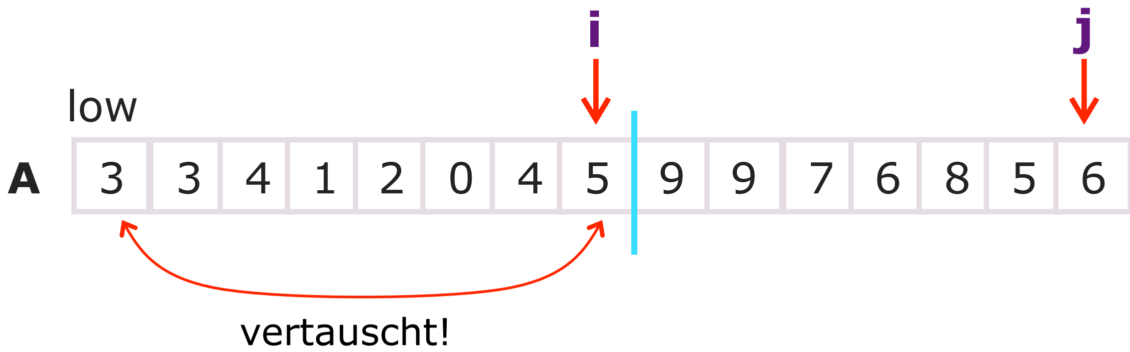



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



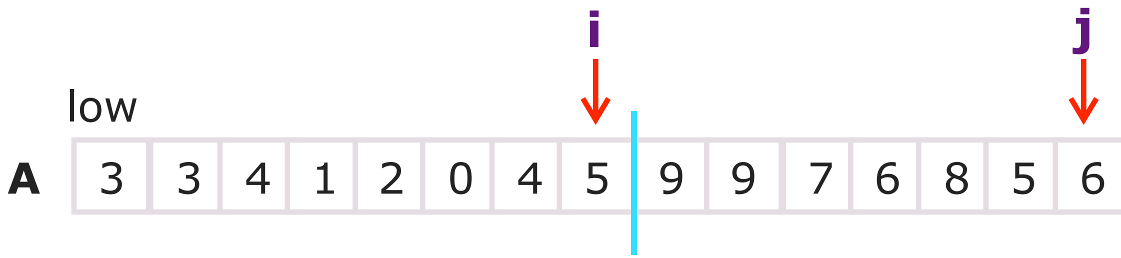

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort-Algorithmus

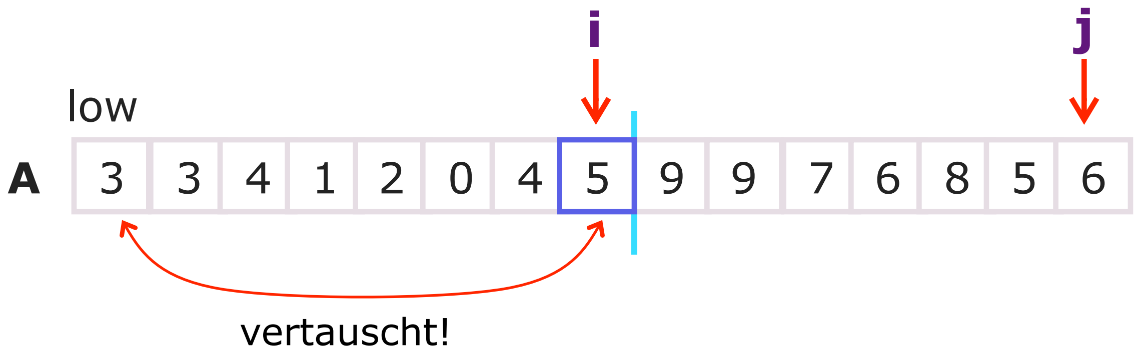



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort-Algorithmus

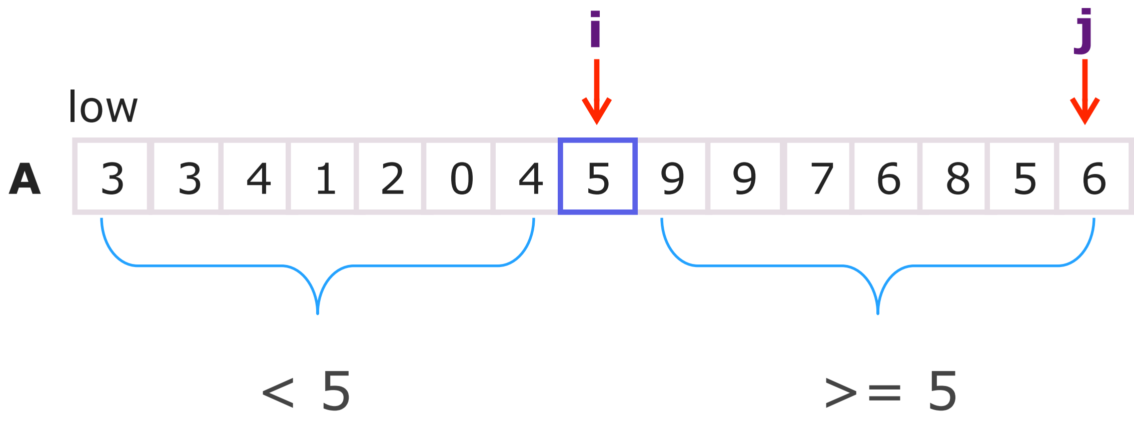



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```



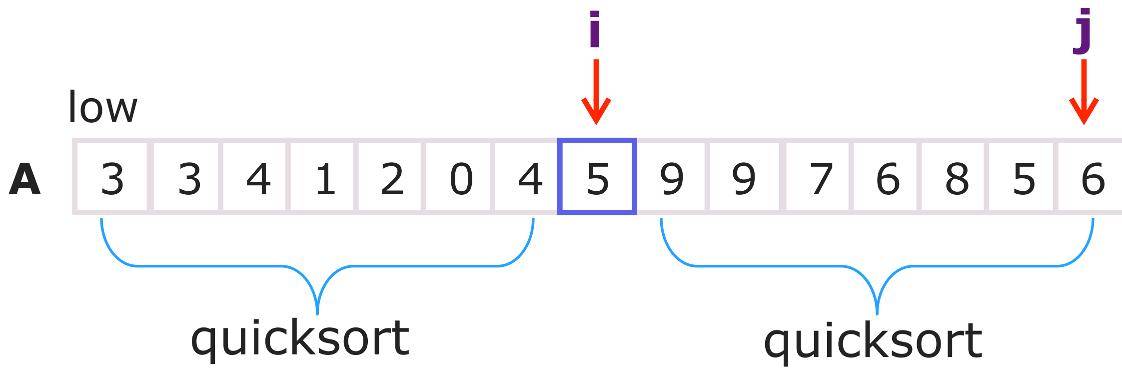

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



Freie Universität Berlin

Quicksort -Algorithmus




```
def quicksort (A, low, high ):
        if  low<high:
                m = partition(A, low, high )
                quicksort ( A,  low,  m-1 )
                quicksort ( A,  m+1,  high )
```

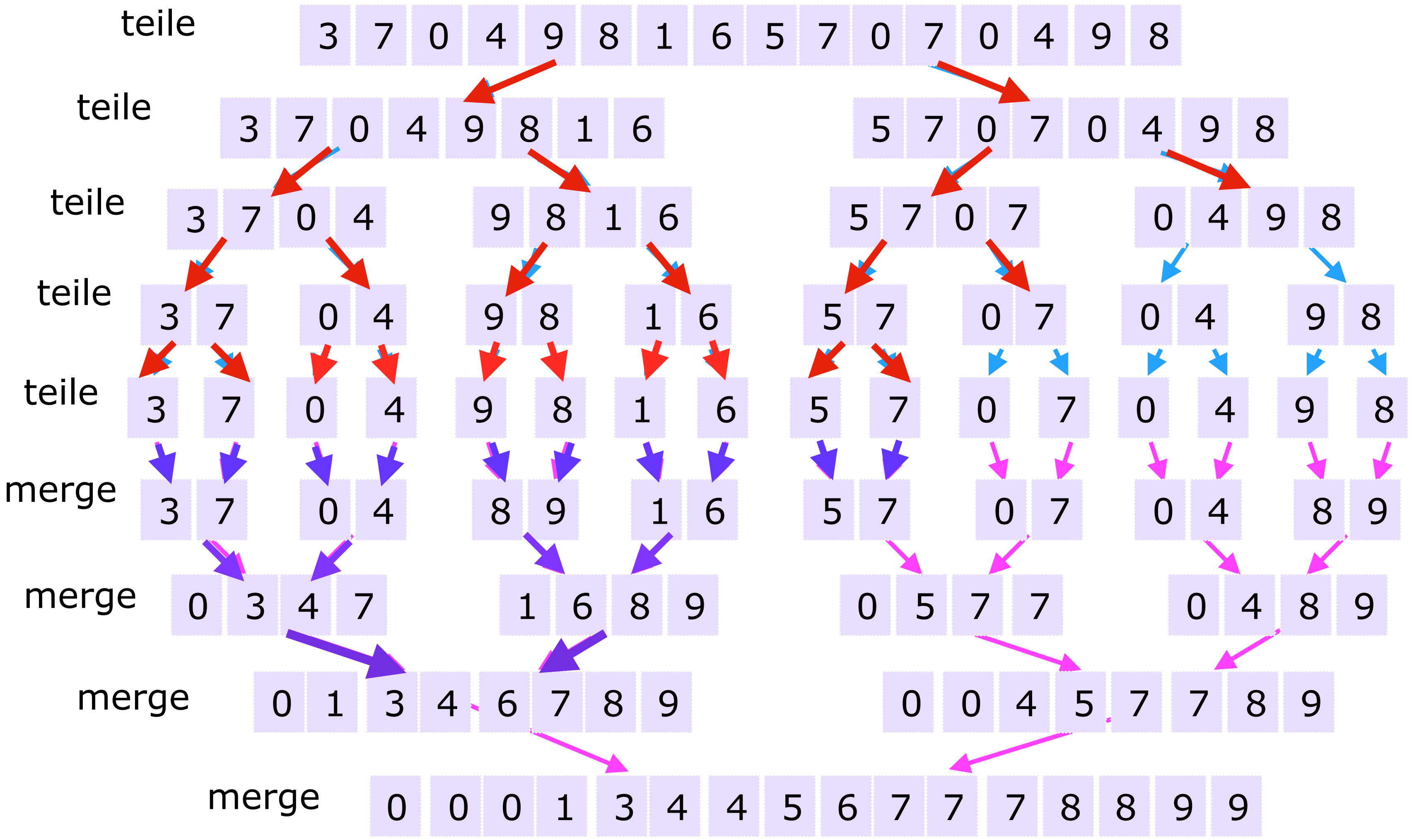
Quicksort-Algorithmus

- **1962** von **Hoare** entwickelt
- ist einer der beliebtesten Sortieralgorithmen
- **einfache** Implementierung
- **in-place**
- Komplexität
  - **O(n·log(n))** im Durchschnitt
  - **O(n^2)** im schlimmsten Fall
- **nicht stabil!**

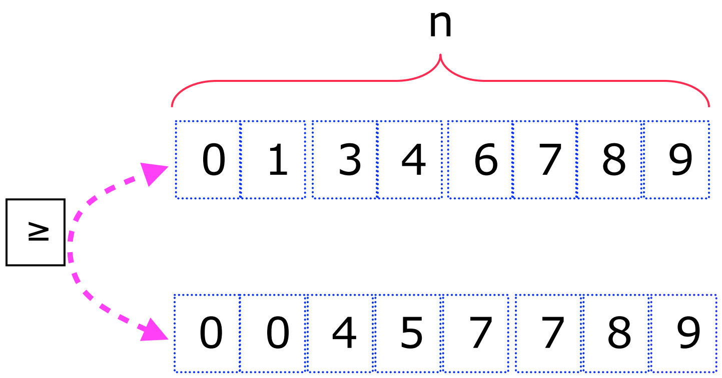
https://www.youtube.com/watch?v=ywWBy6J5gz8

Merge-Sort-Algorithmus

Freie Universität Berlin



Merge-Algorithmus



Merge-Algorithmus

$$\overbrace{\boxed{0}\ \boxed{0}\ \boxed{0}\ \boxed{1}\ \boxed{3}\ \boxed{4}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{7}\ \boxed{7}\ \boxed{8}\ \boxed{8}\ \boxed{9}\ \boxed{9}}^{2*n}$$

Sortierte Zahlen

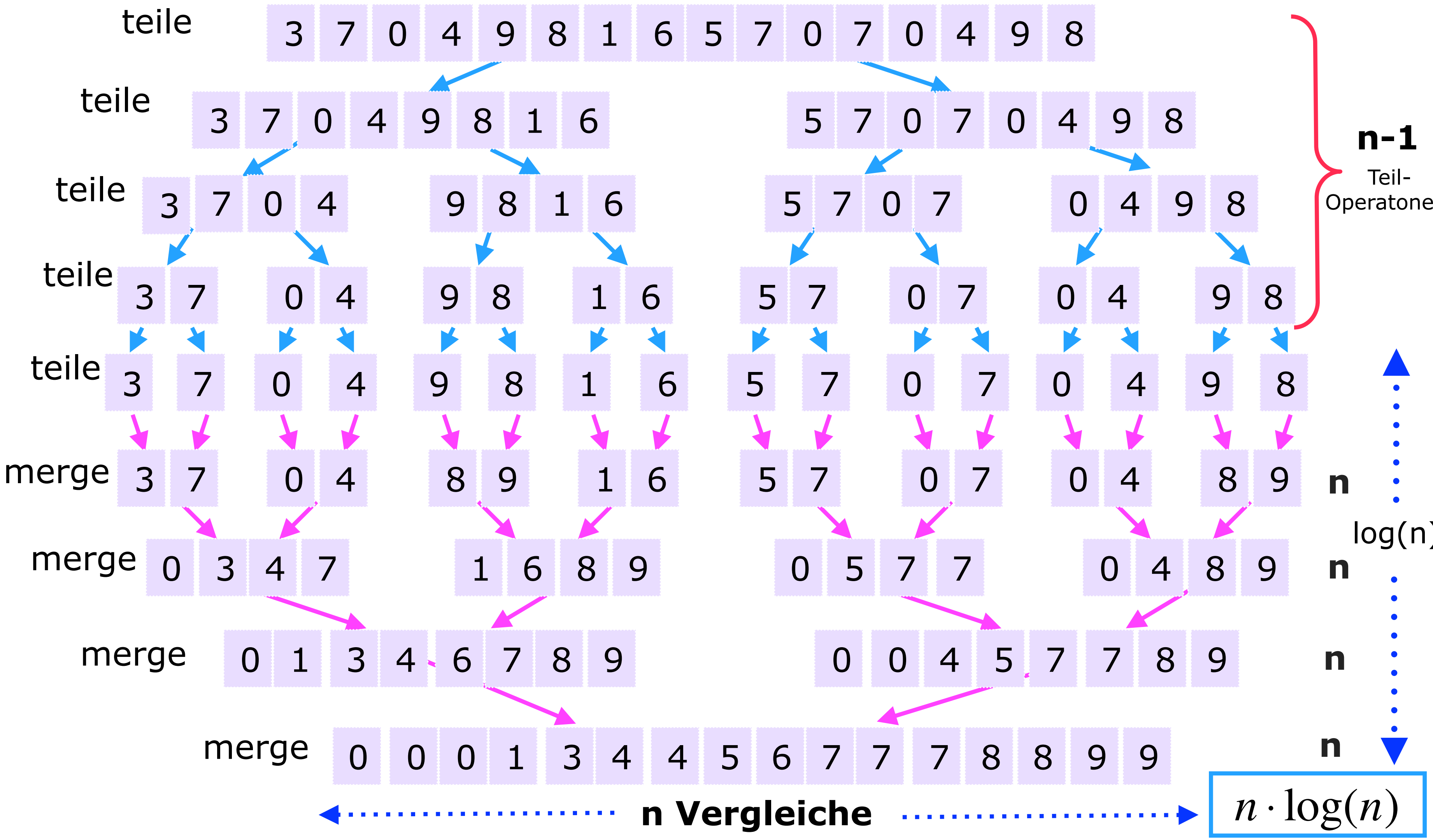
Wir hatten ursprünglich zwei sortierte Mengen mit Länge **n.**

Nach jedem Vergleich wird eine Zahl sortiert,

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

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

Merge-Sort-Algorithmus



Merge-Sort-Algorithmus

Eine Teilung kostet c_1

Ein Vergleich kostet c_2

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

$$\boxed{T(n) = O(n \cdot \log(n))}$$

Merge-Sort-Algorithmus

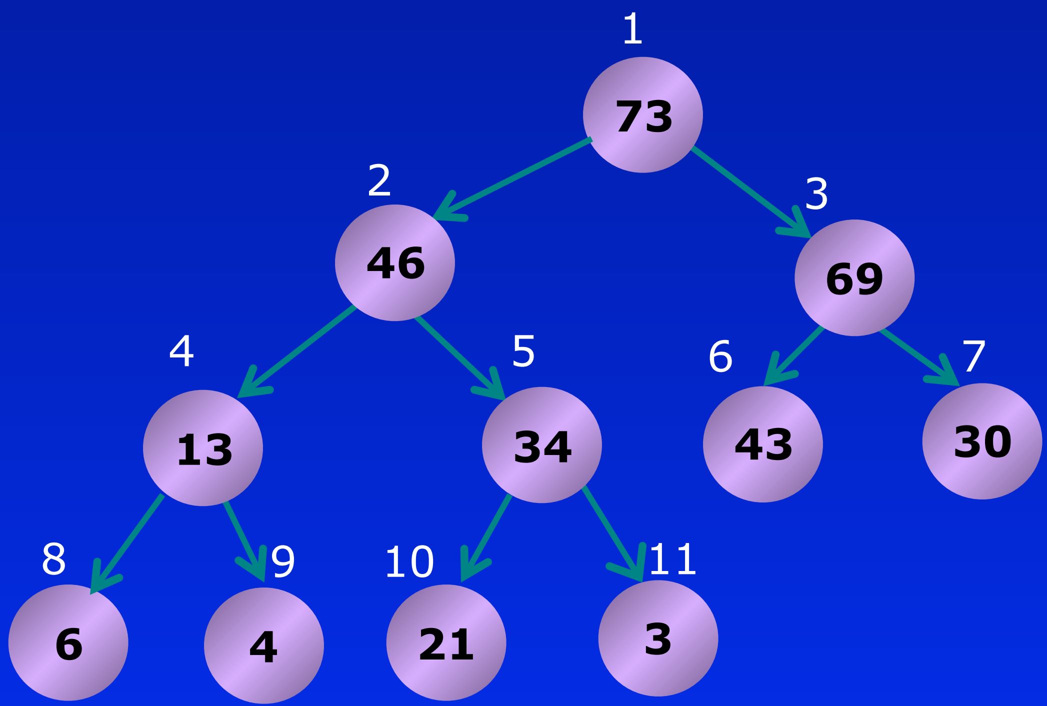
```
def merge(low, high):
        res = []
        i, j = 0, 0
        while i<len(low) and j<len(high):
                if low[i] <= high[j]:
                        res.append(low[i])
                        i = i+1
                else:
                        res.append(high[j])
                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

Heapsort

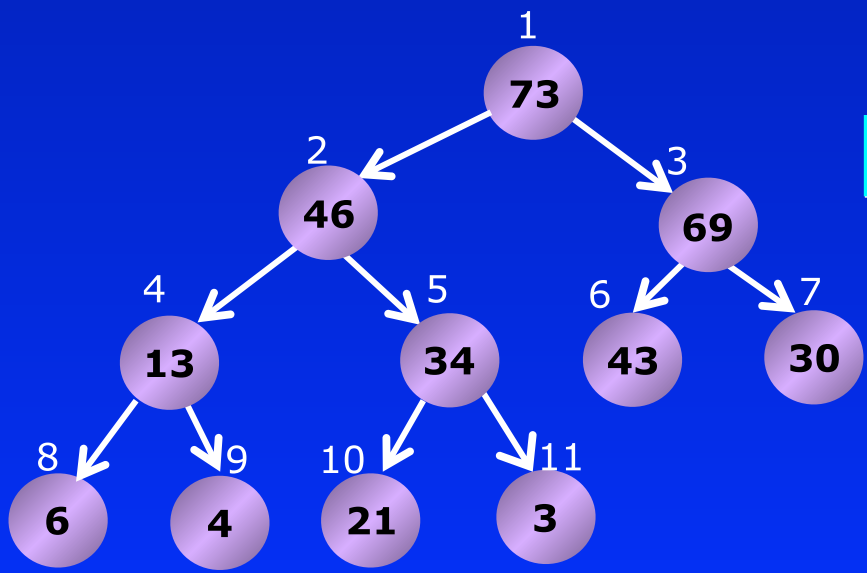


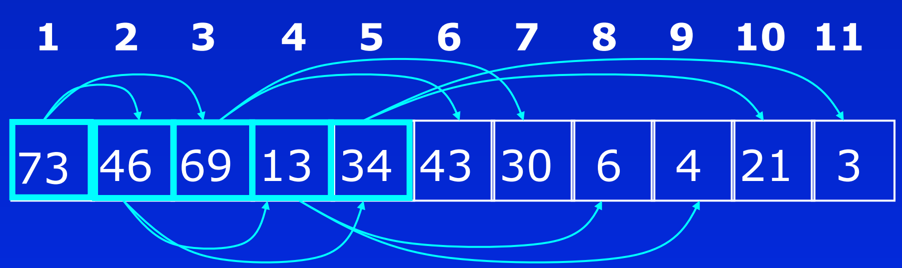
Heapsort

- **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.

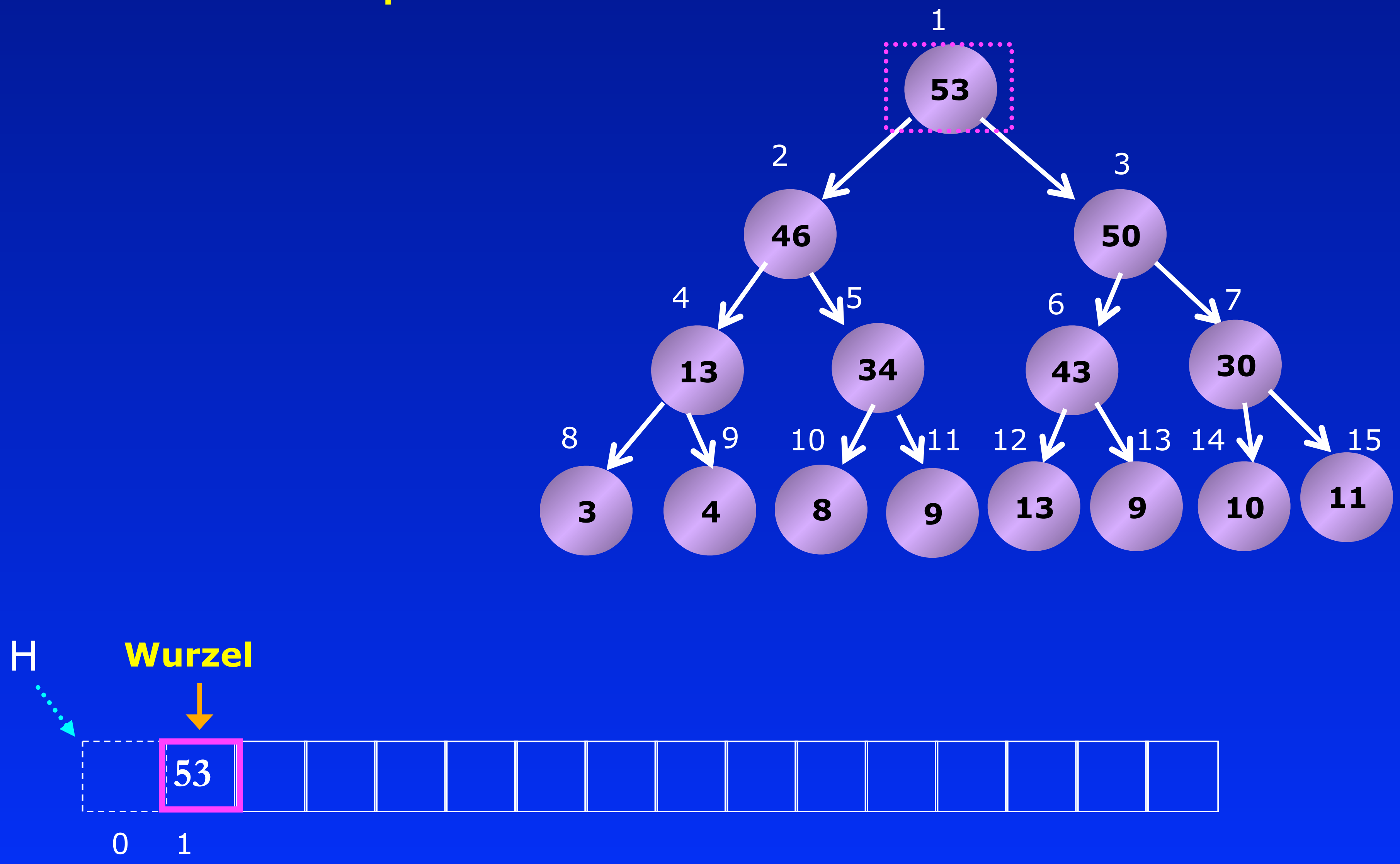
Heap

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.

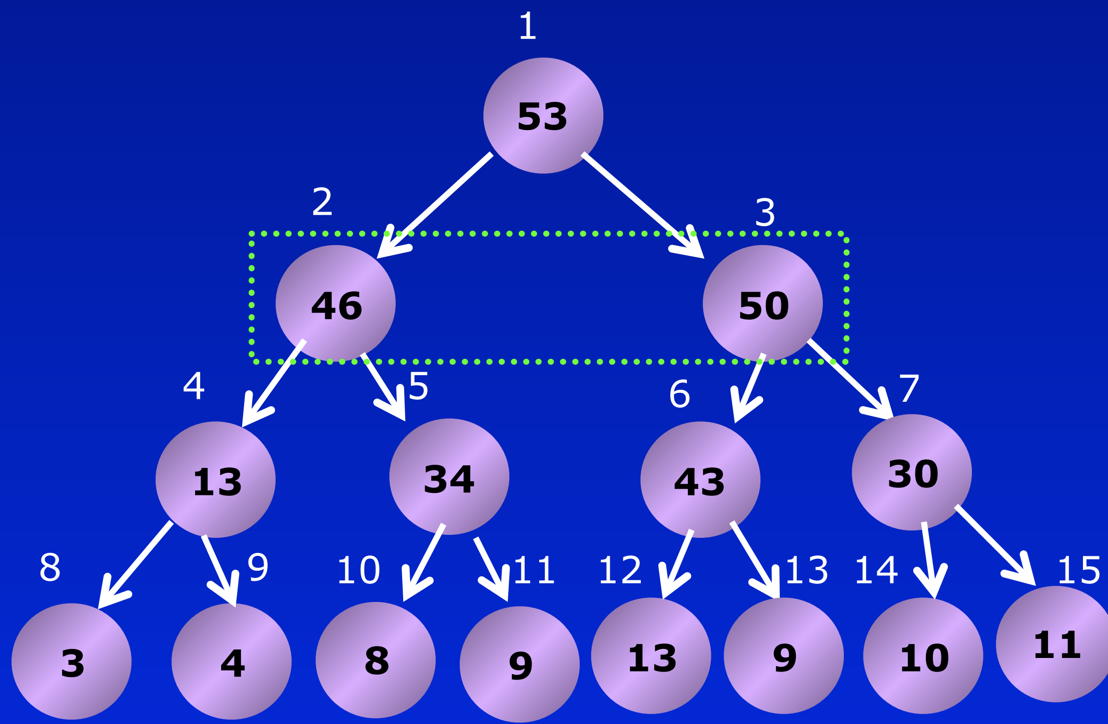




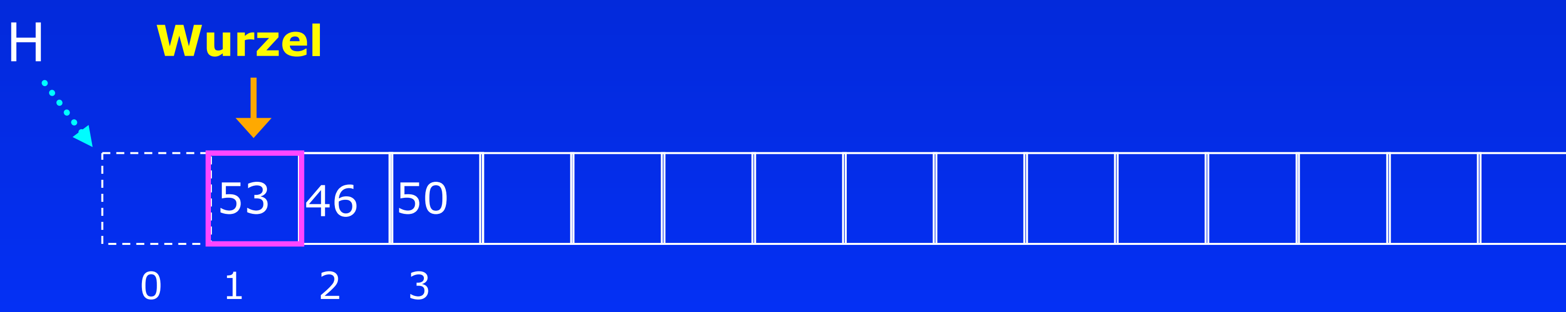
Heap


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

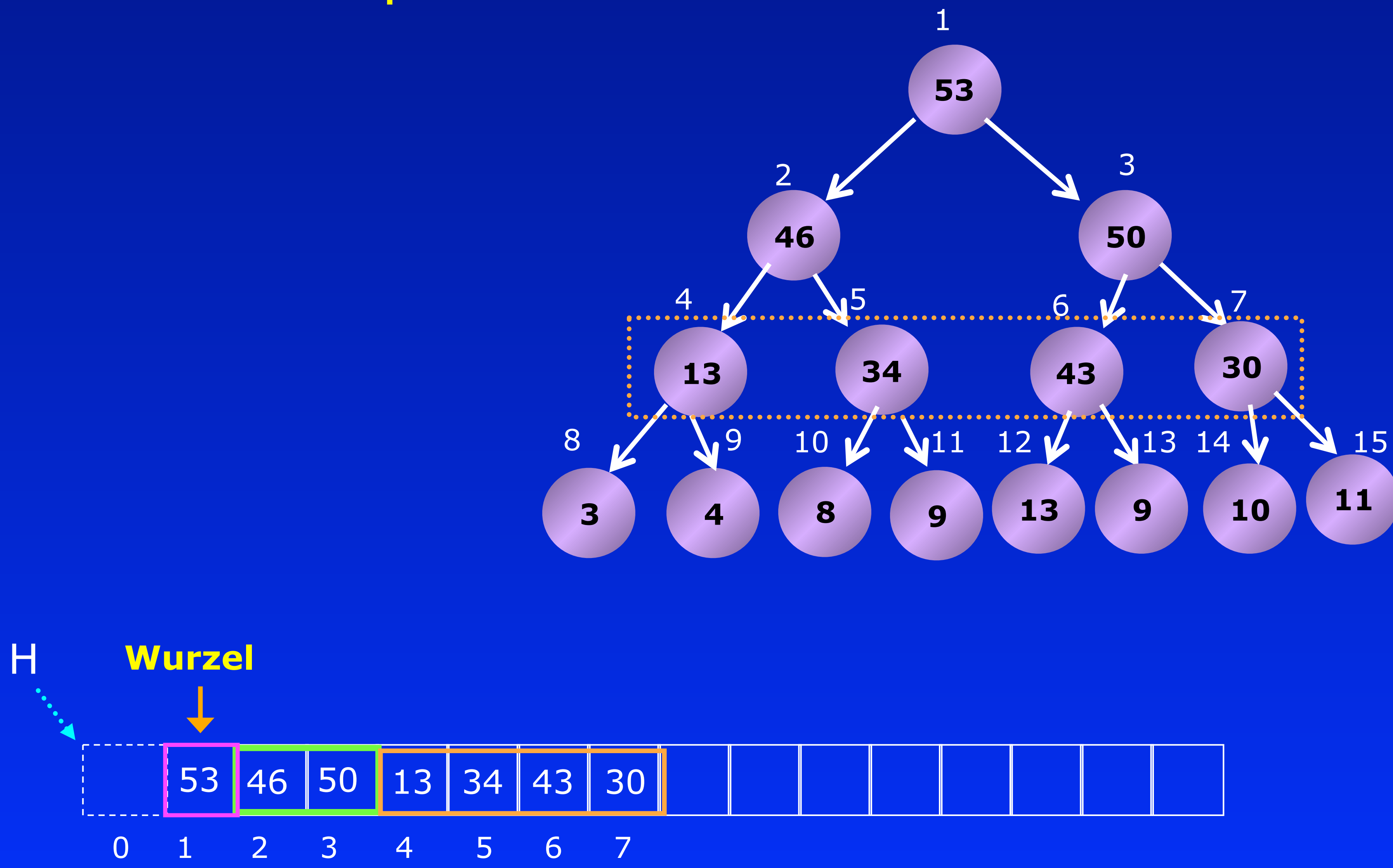


1
53
2
3
46
50
4
5
6
7
13
34
43
30
8
9
10
11
12
13
14
15
3
4
8
9
13
9
10
11

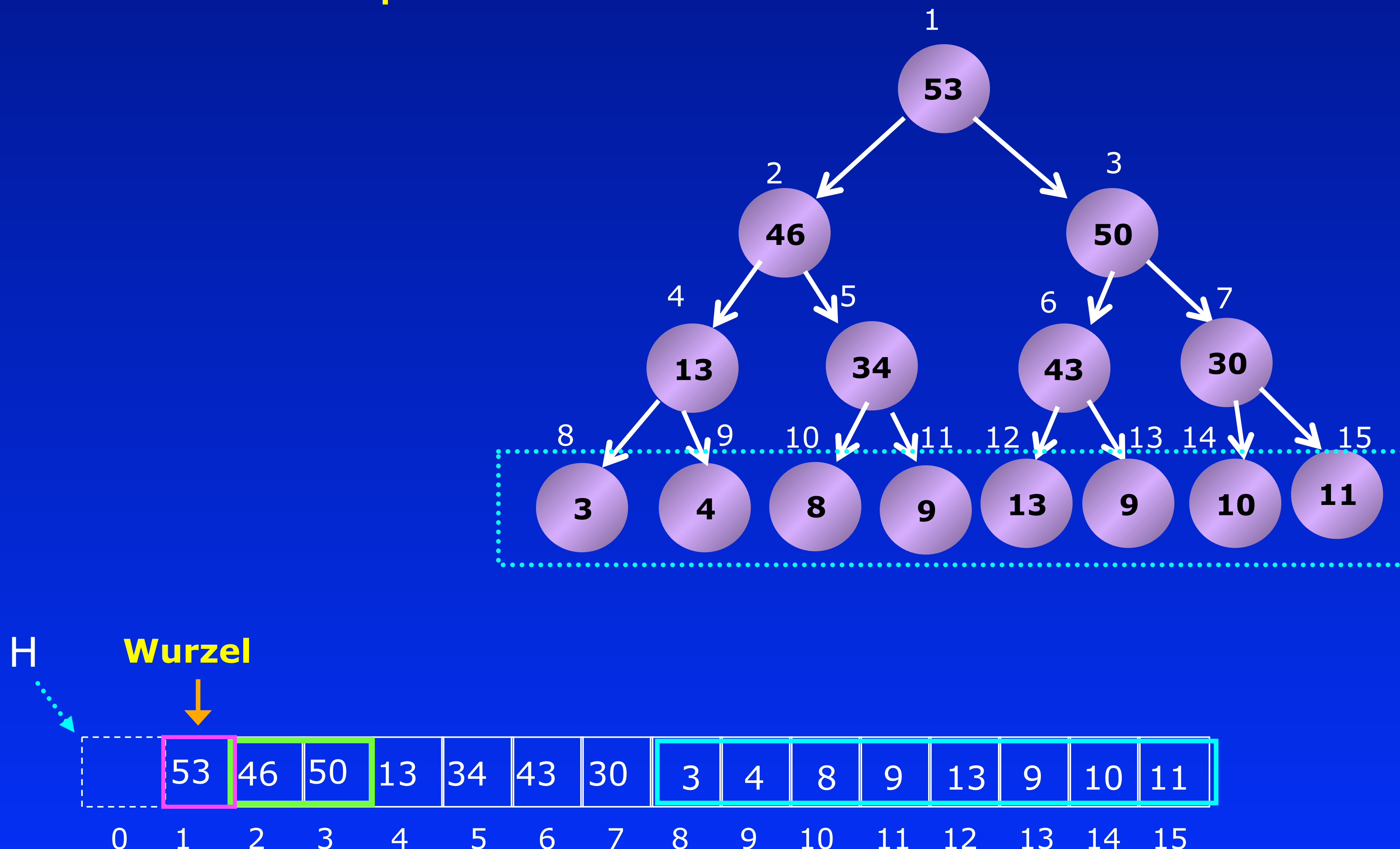


H
Wurzel
53
46
50
0
1
2
3

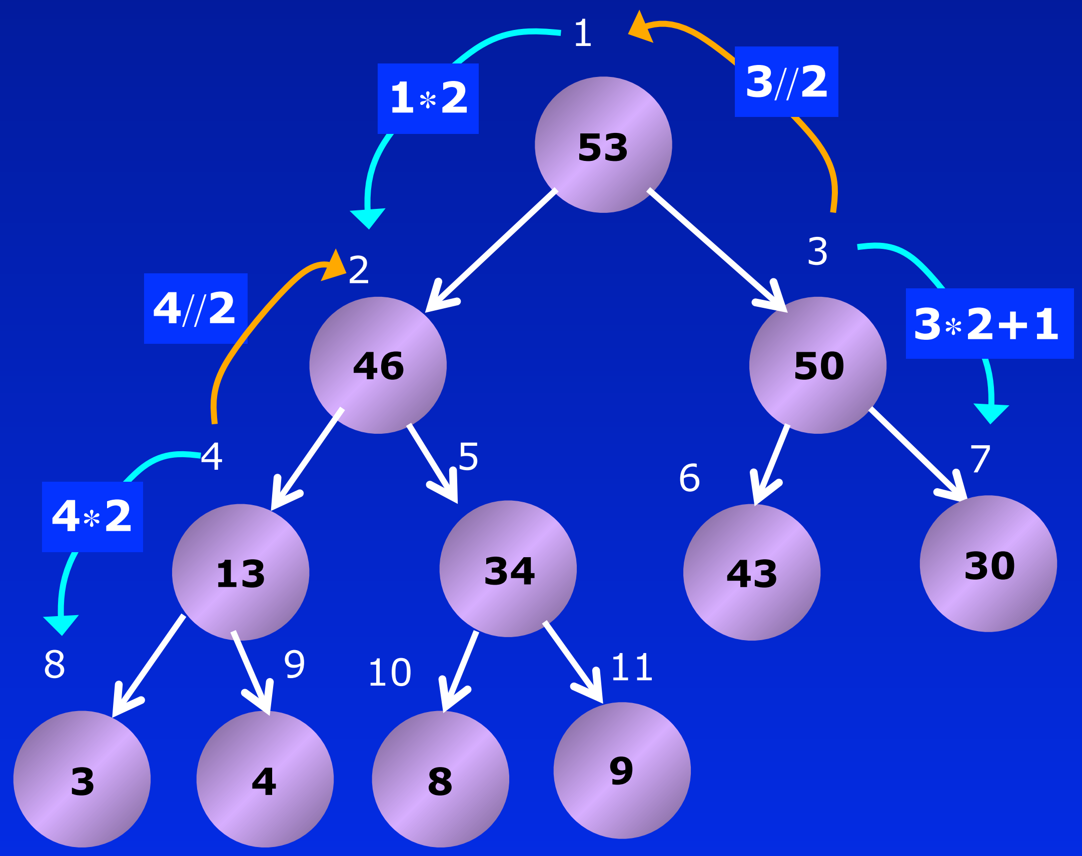
Heap


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


53
46
50
13
34
43
30
3
4
8
9
13
9
10
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

Navigieren im Heap

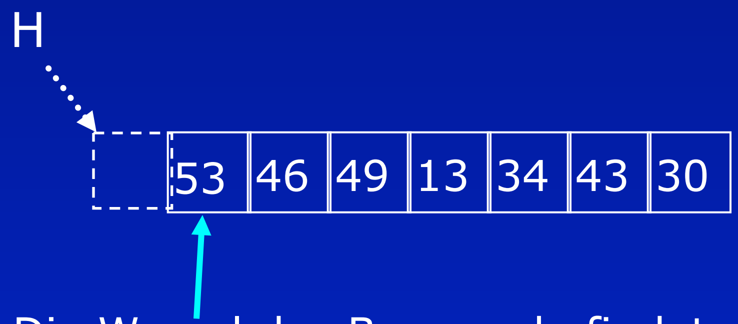


Das linke Kind einer beliebigen Position **i** des Heaps ist gleich **i*2**

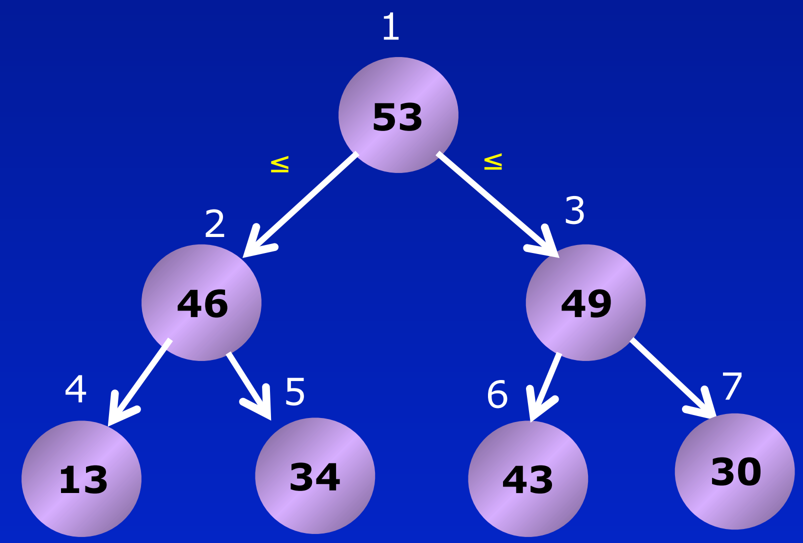
Das rechte Kind einer beliebigen Position **i** ist gleich **i*2+1**

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:

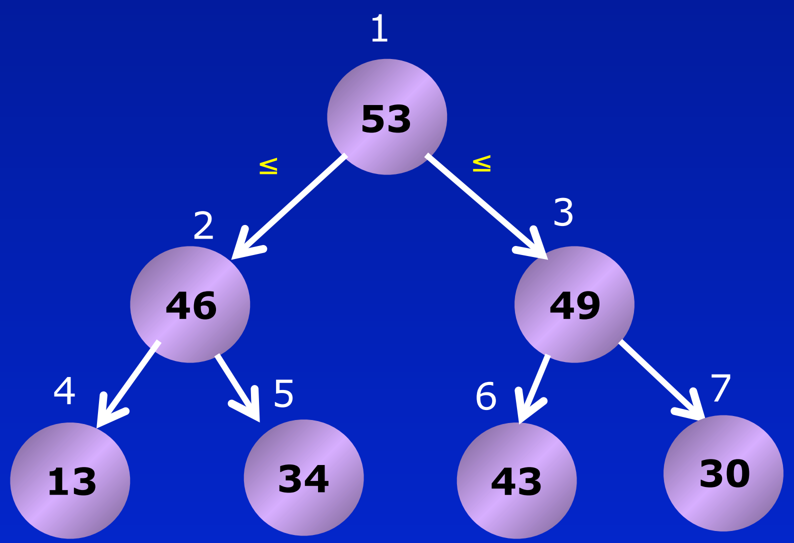
```
def parent(i):
        return i//2
```

```
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.

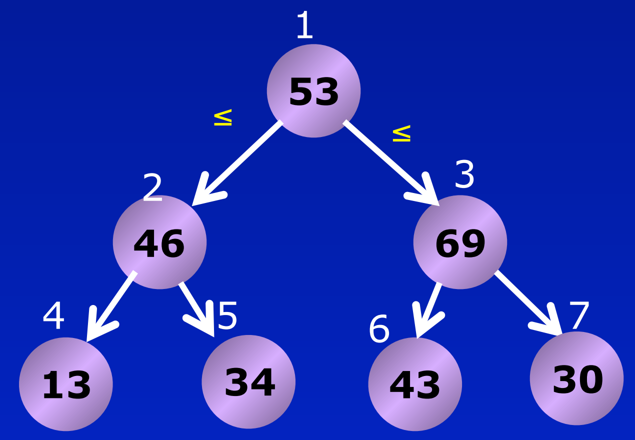


Die Größe des **Heaps** wird an der Position **0** des Feldes gespeichert (H[0]).

Heap-Hilfsfunktionen

H

7	53	46	69	13	34	43	30
0	1	2	3	4	5	6	7



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

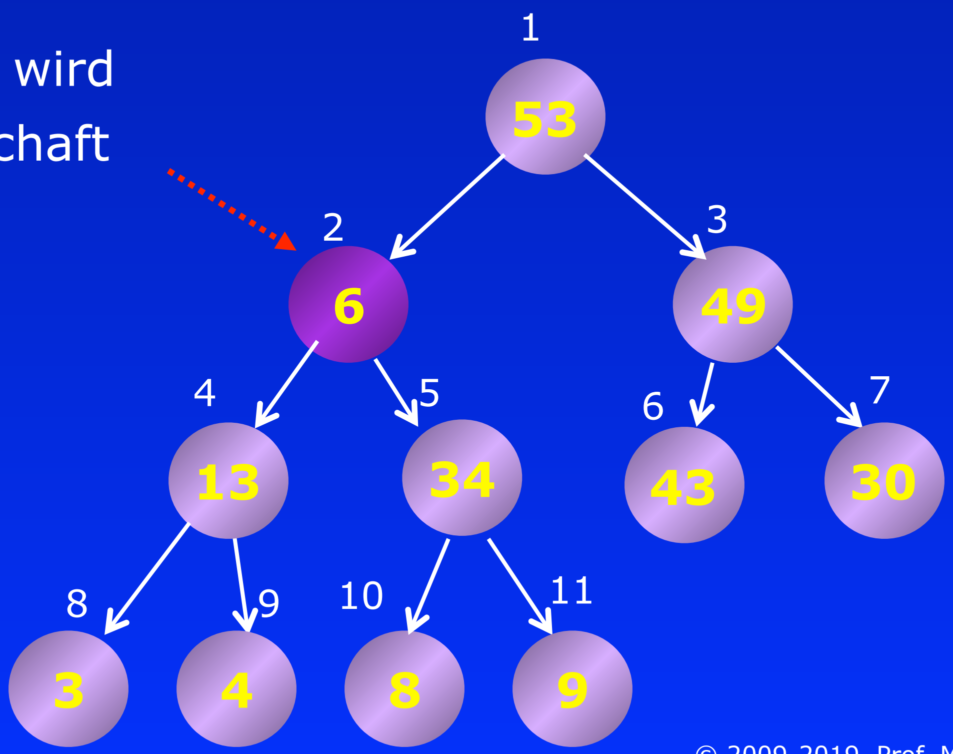
```
def  heap_size(H):
        return H[0]
```

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

max_heapify-Funktion

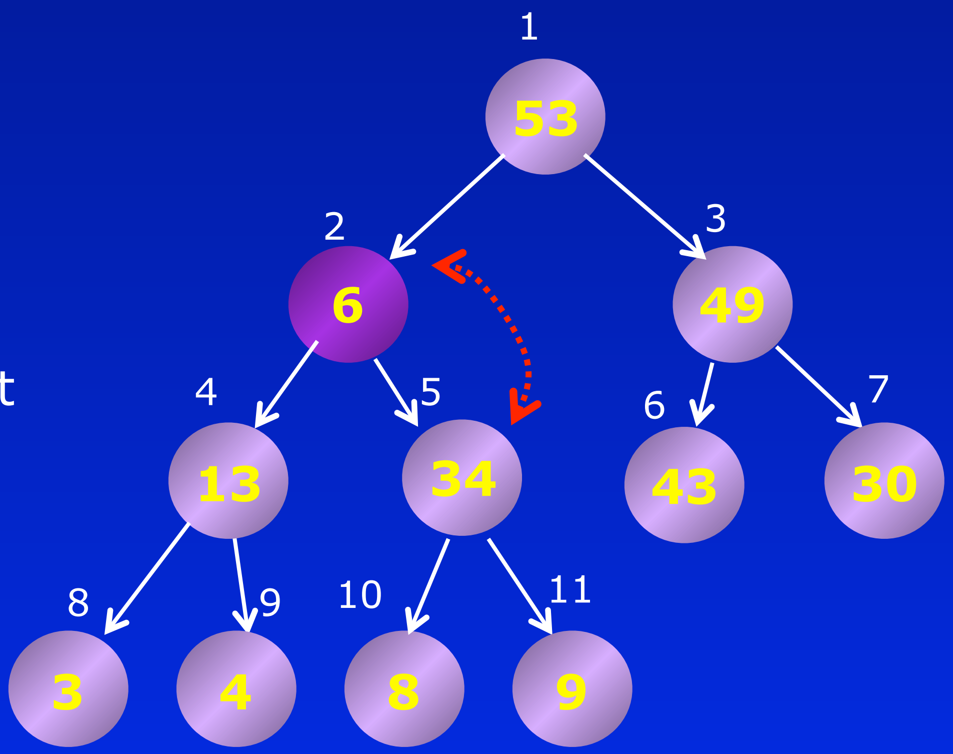
Die **max_heapify-Funktion** soll einen beliebigen Knoten des Baumes bekommen und testen, ob die **heap**-Eigenschaft erfüllt wird. Wenn das nicht der Fall ist, wird der Fehler korrigiert, indem das größte von beiden Kindern gegen den jeweiligen Knoten vertauscht wird.

In der Position 2 wird die Heap-Eigenschaft verletzt.



max_heapify-Funktion

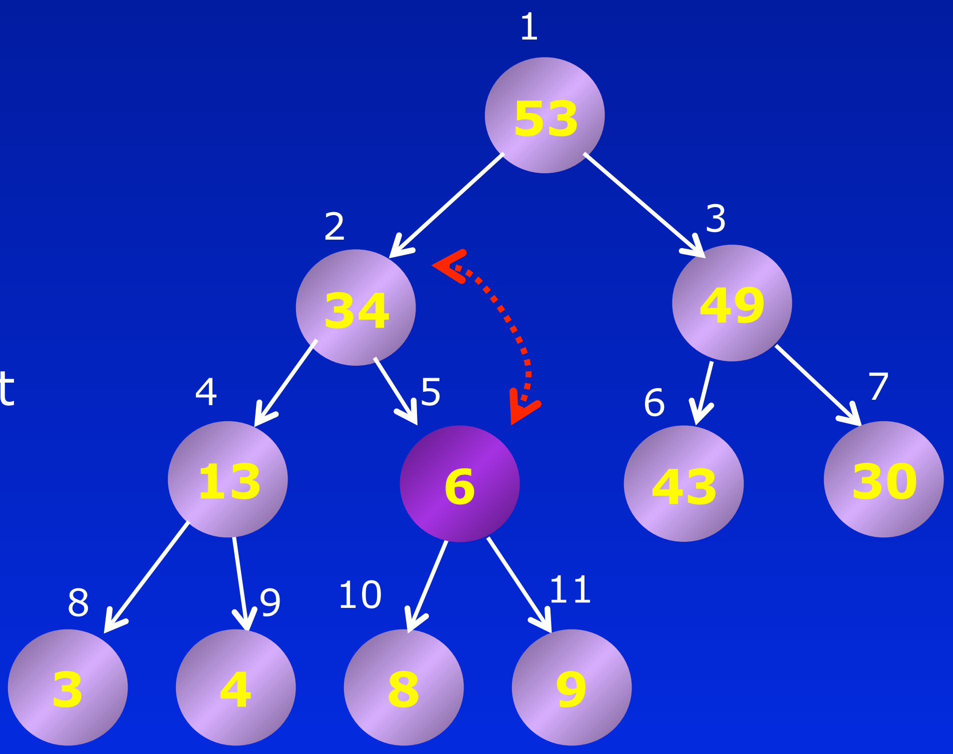
Das größere von beiden Kindern wird mit dem Element in Position **2** vertauscht, wenn es gleichzeitig größer als das Element in Position **2** ist.



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.

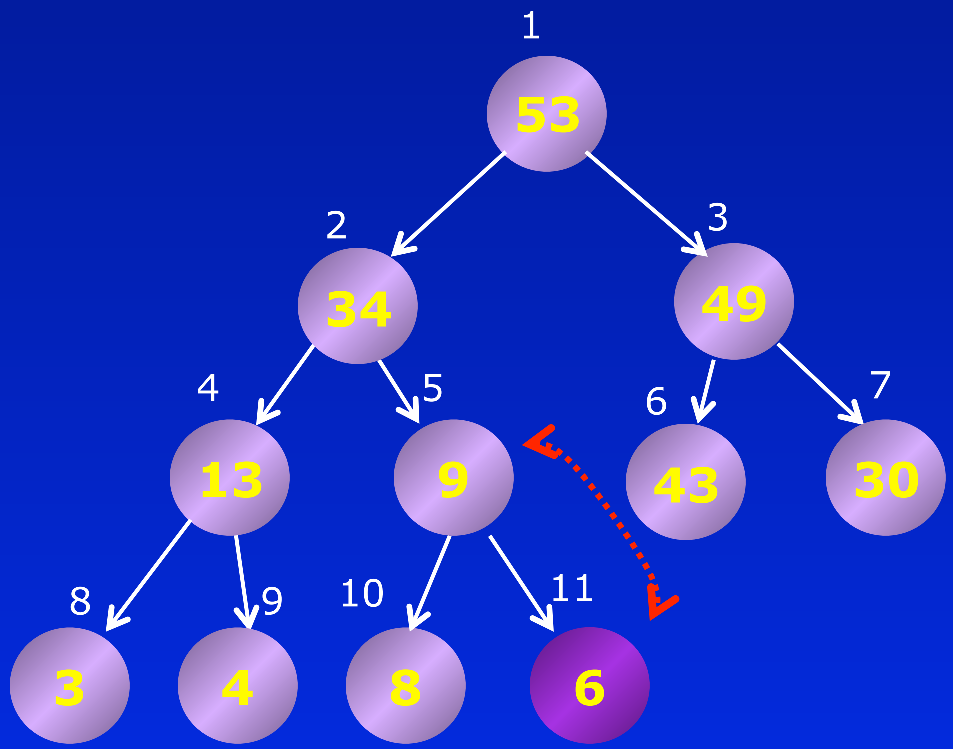
max_heapify-Funktion

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.

```python
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.

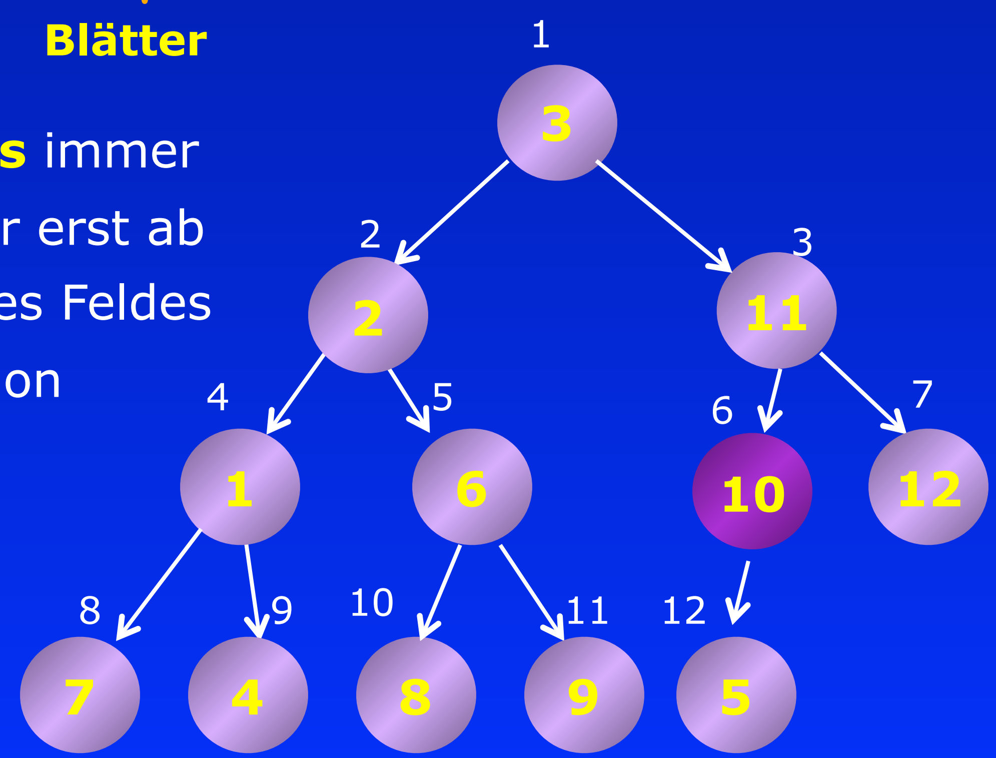
H

	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

$\underbrace{\text{12 7 4 8 9 5}}_{\textbf{Blätter}}$

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**.

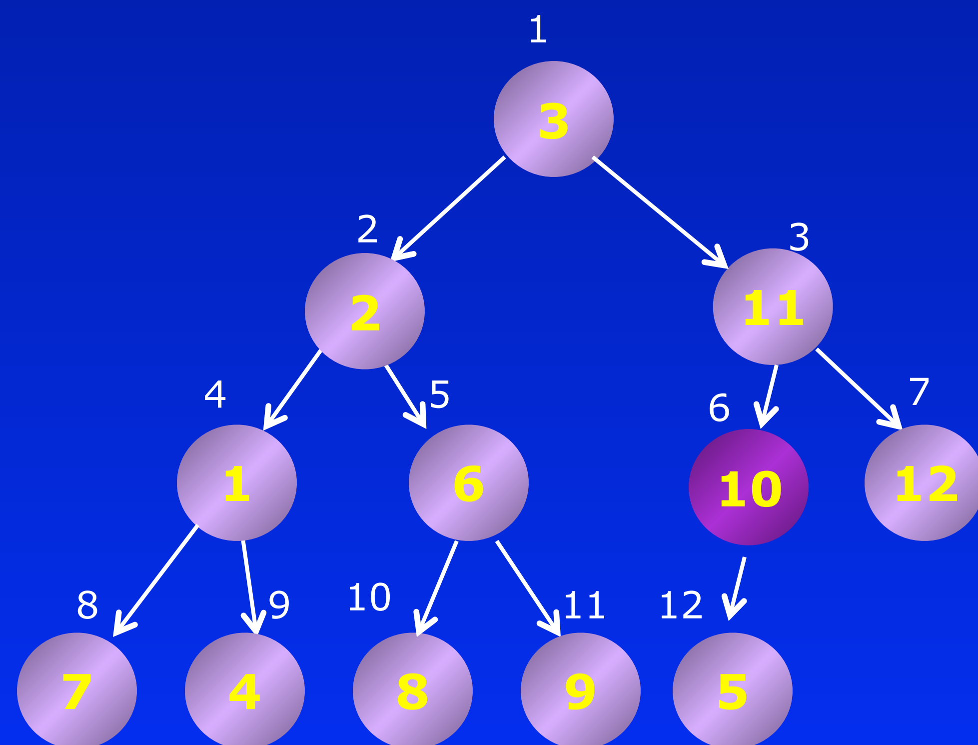


build_heapify-Funktion

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)

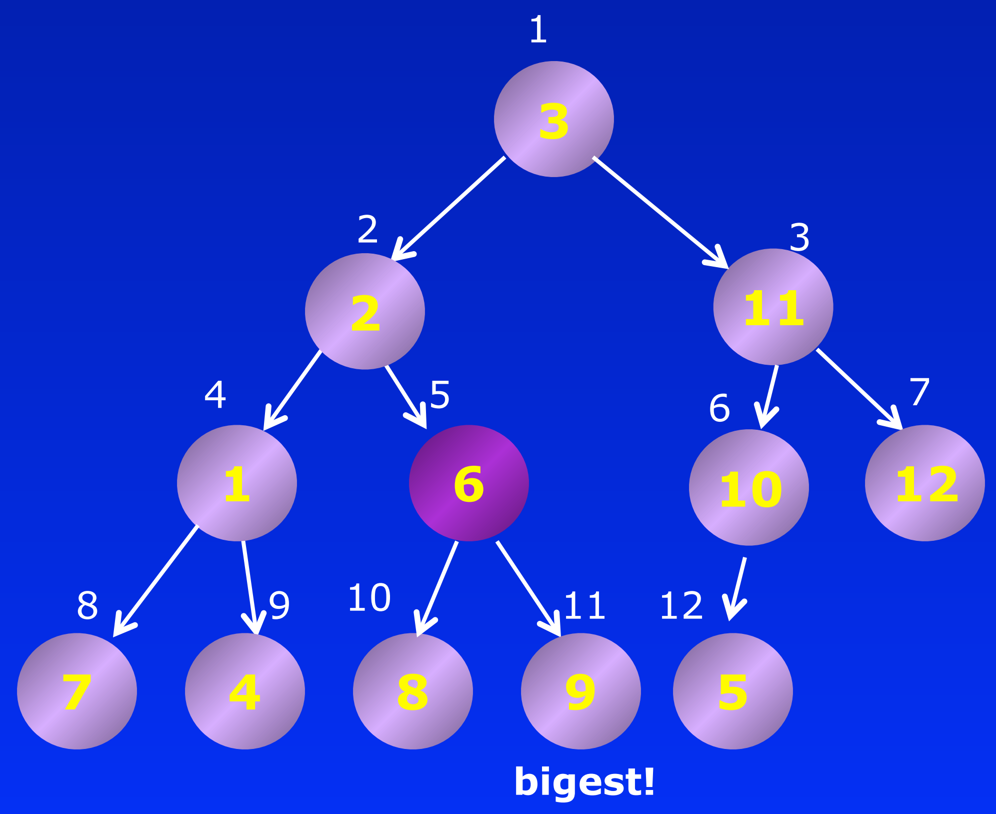


build_heapify-Funktion

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)

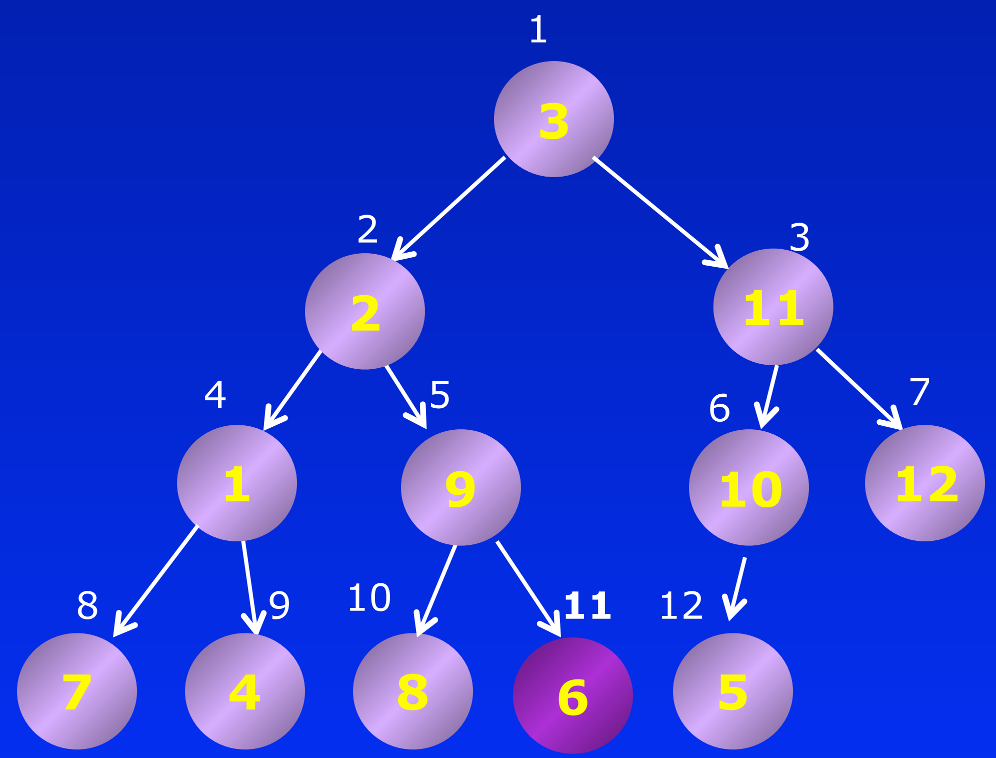


build_heapify-Funktion

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)



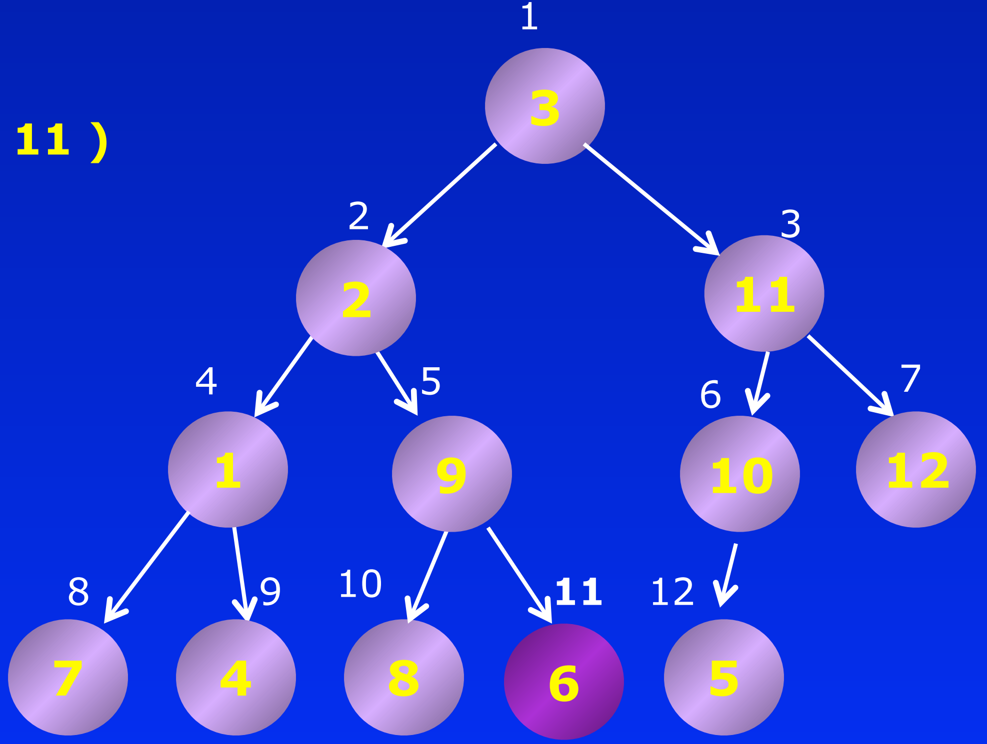
build_heapify-Funktion

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)

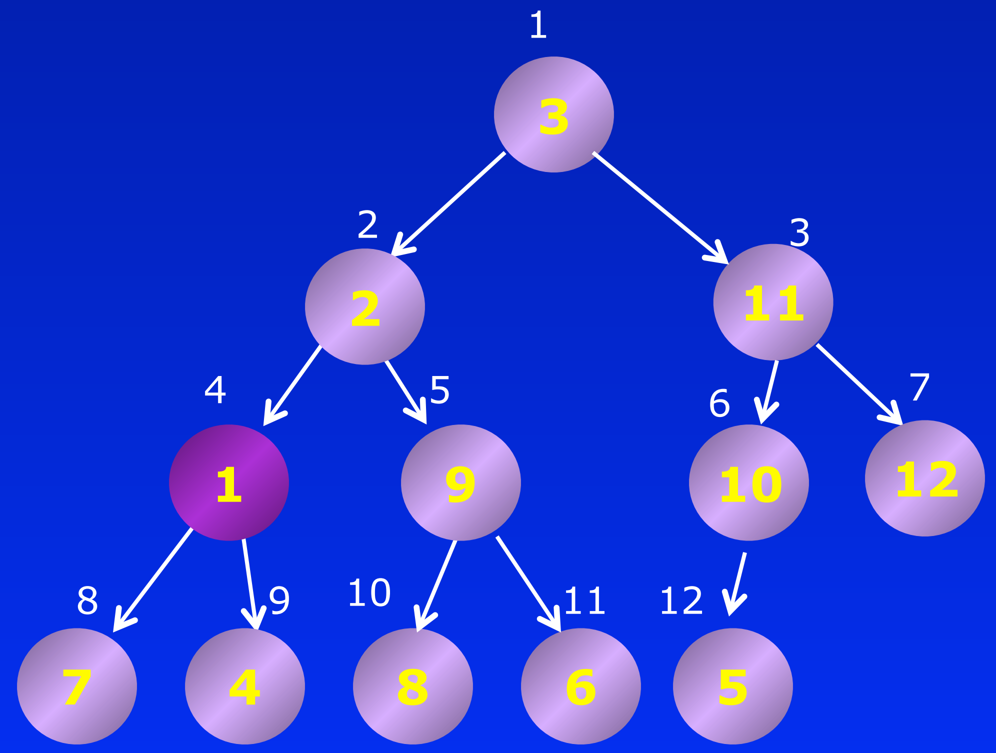


build_heapify-Funktion

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)



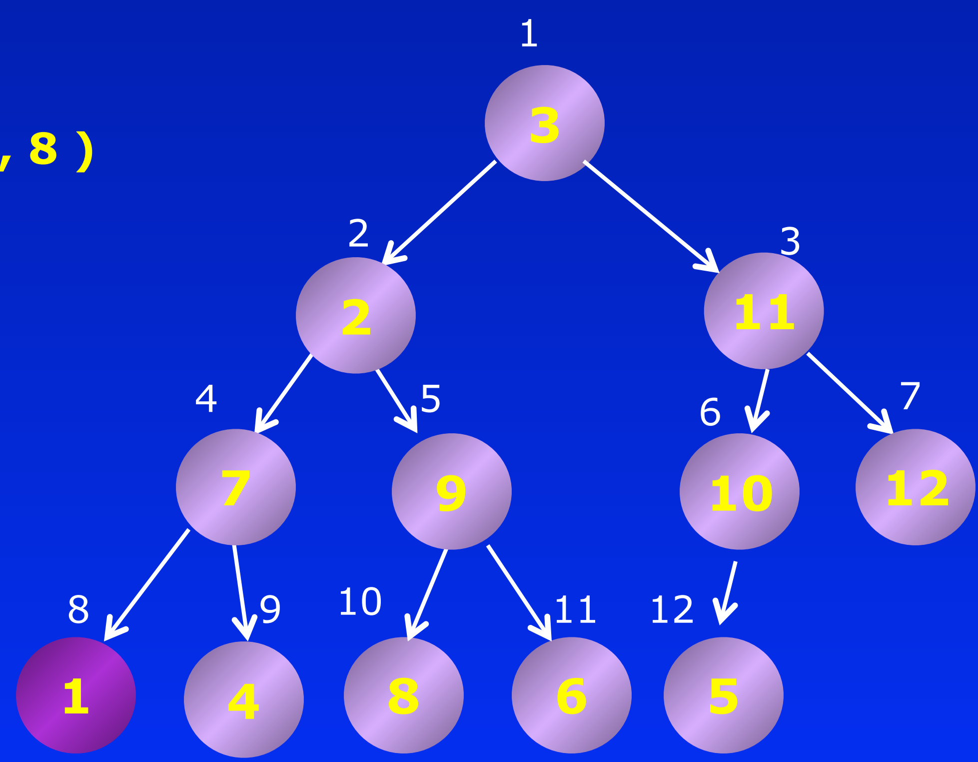
build_heapify-Funktion

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)

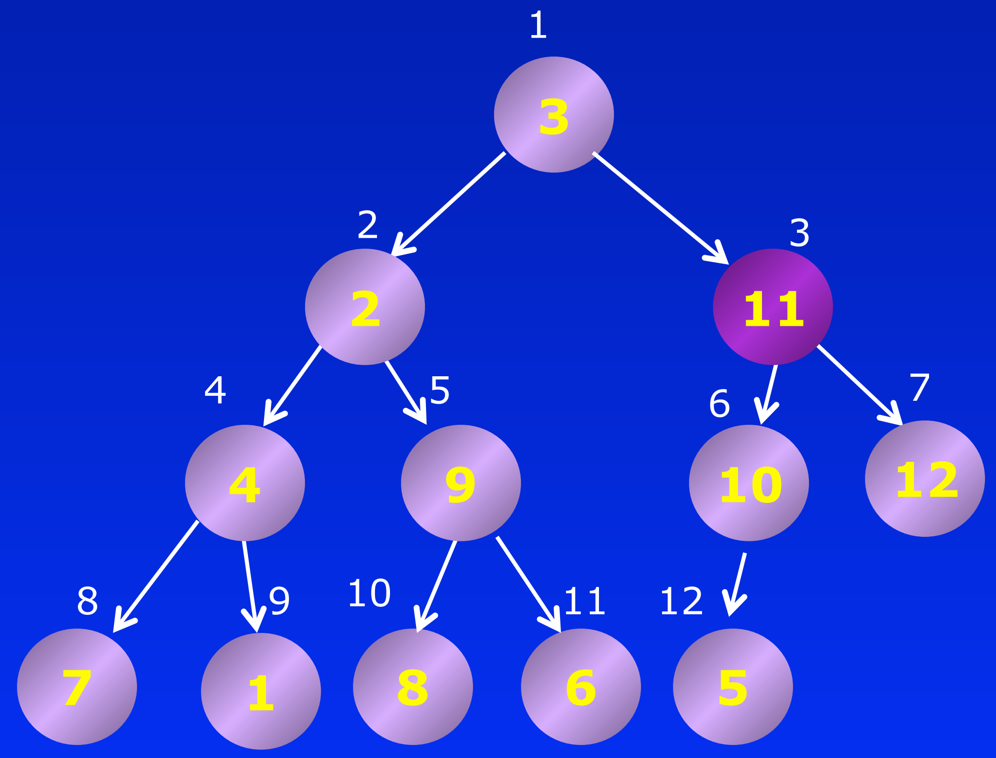


build_heapify-Funktion

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)



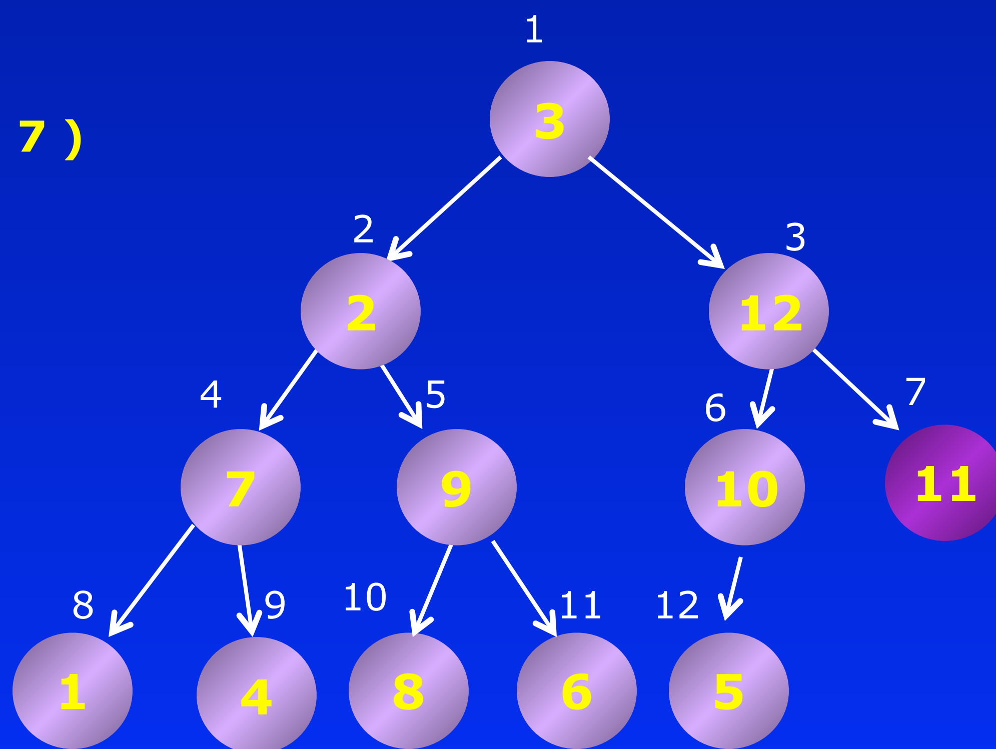
build_heapify-Funktion

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)

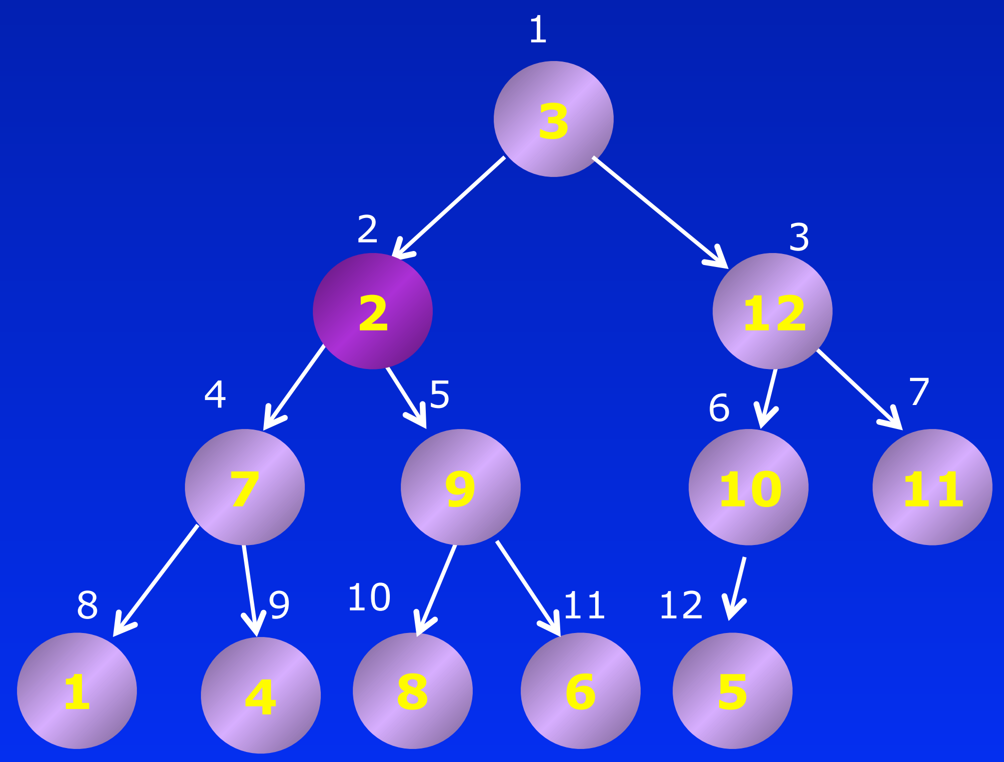


build_heapify-Funktion

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)



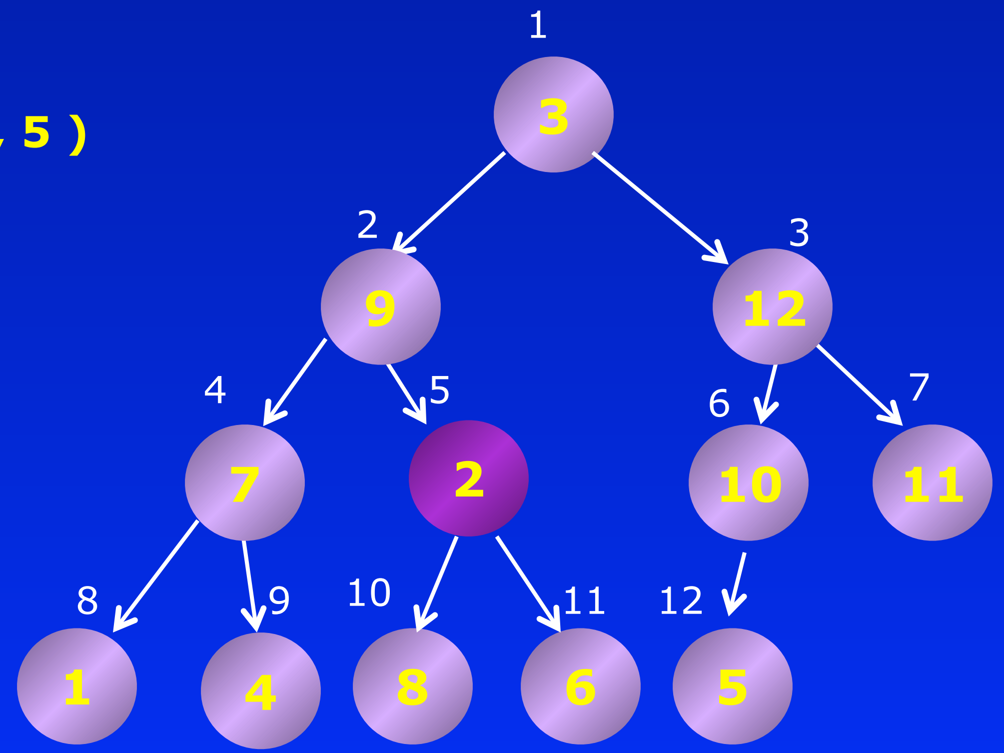
build_heapify-Funktion

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

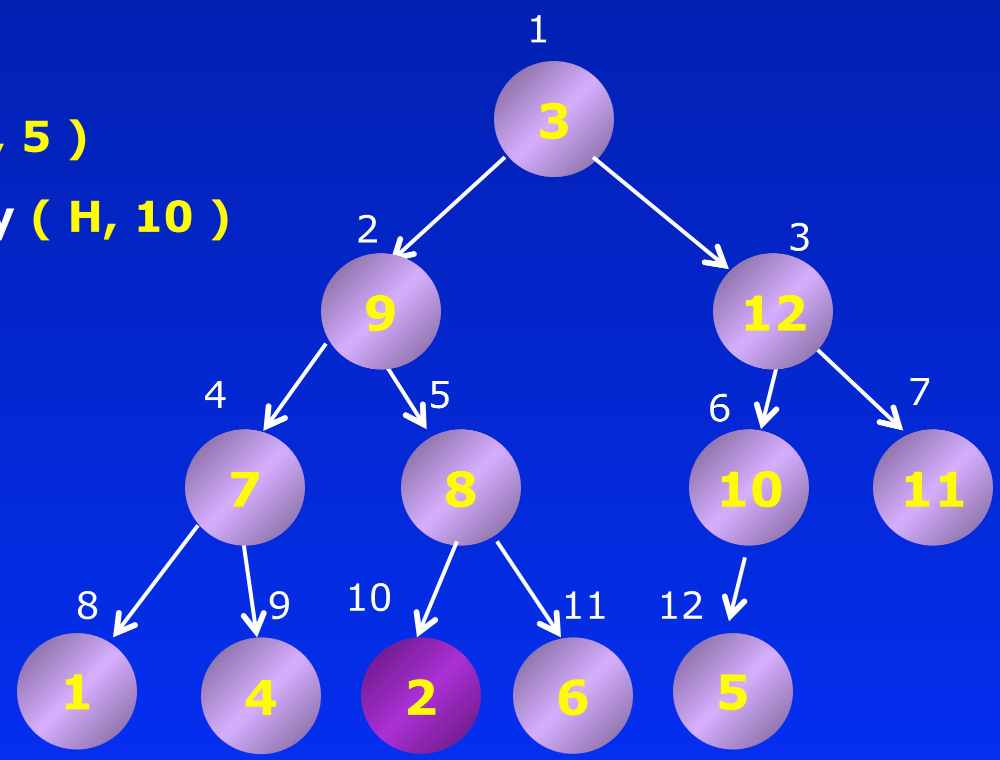
1	2	3	4	5	6	7	8	9	10	11	12
3	9	12	7	8	10	11	1	4	2	6	5

i

max_heapify (H, 2)

max_heapify (H, 5)

max_heapify (H, 10)

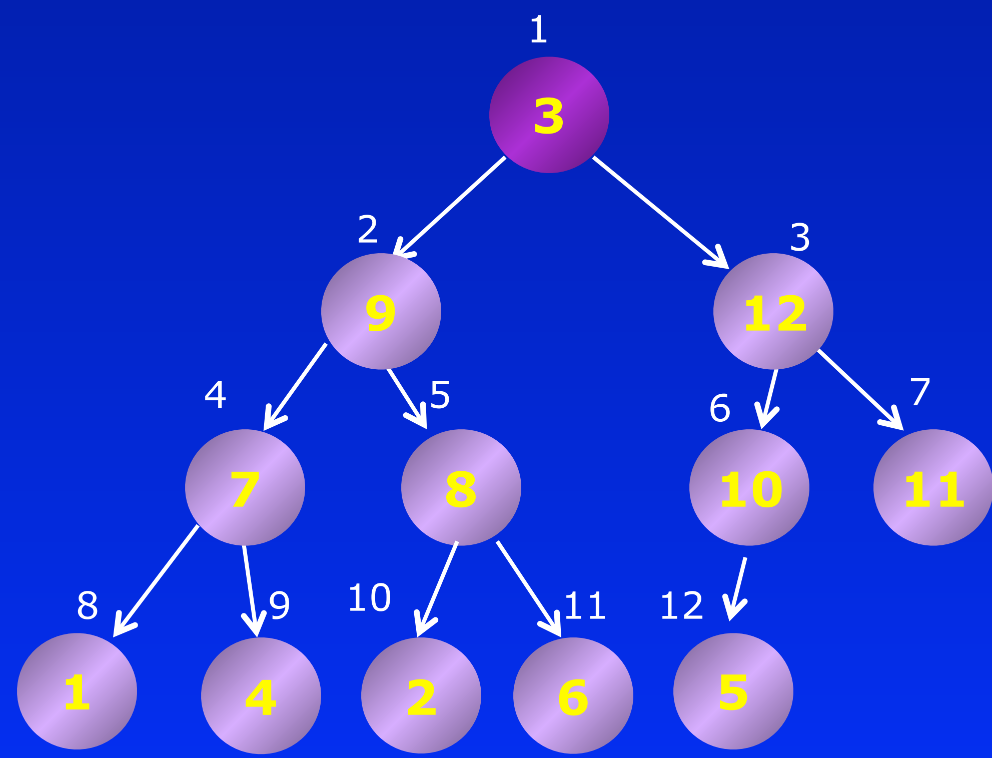


build_heapify-Funktion

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)



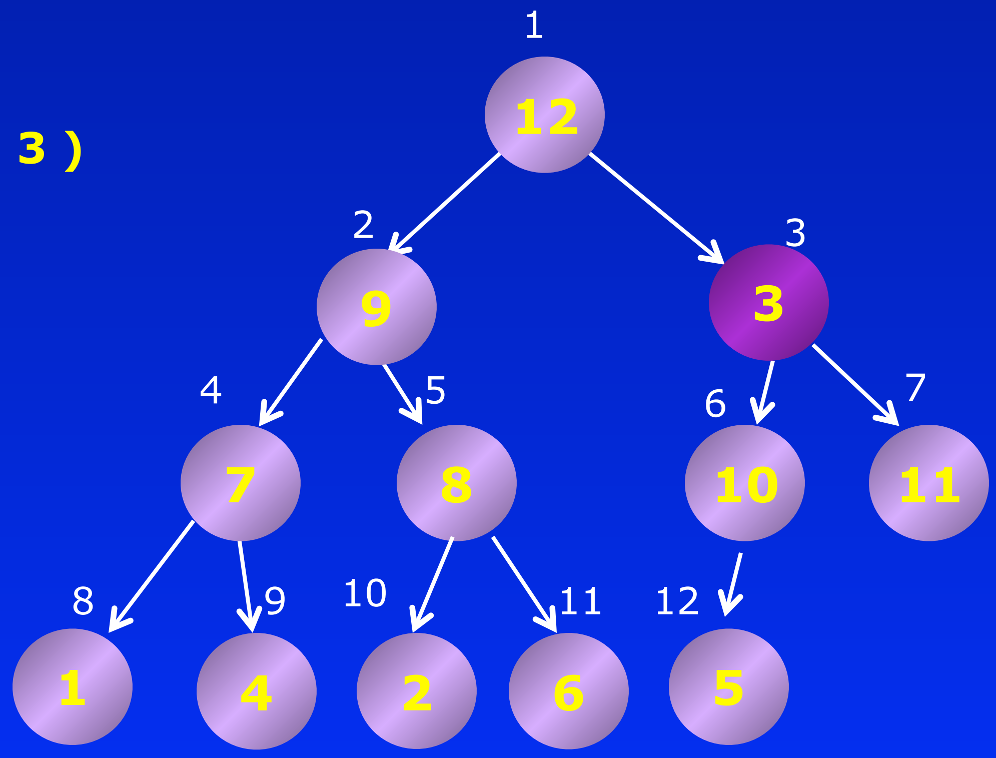
build_heapify-Funktion

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

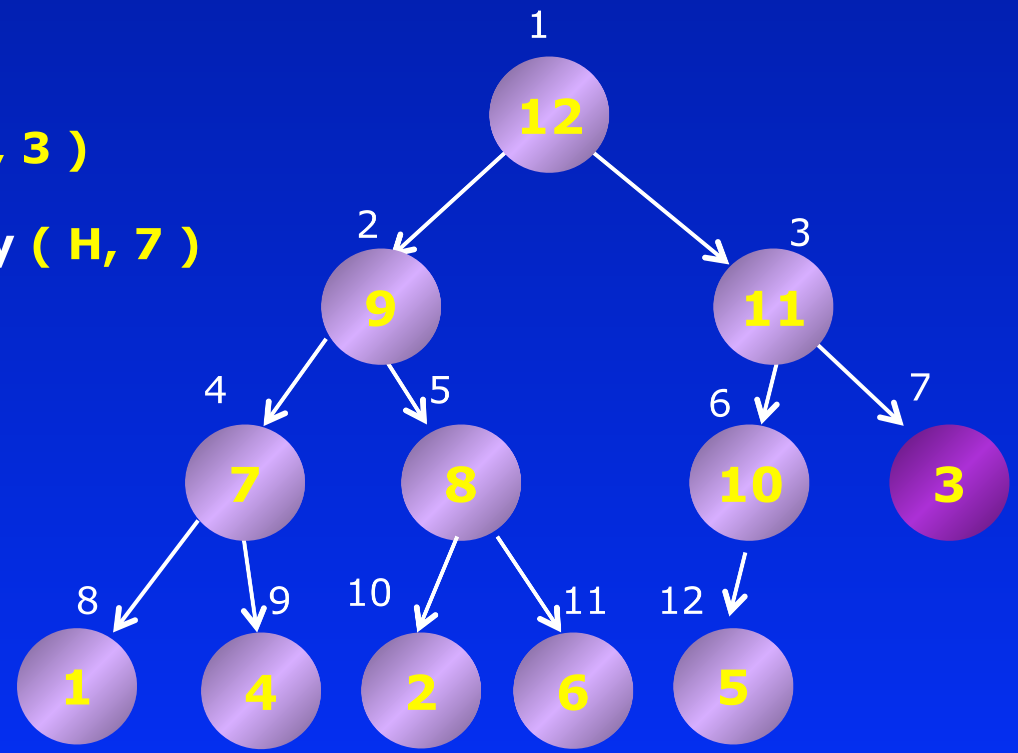
1	2	3	4	5	6	7	8	9	10	11	12
12	9	11	7	8	10	3	1	4	2	6	5

i

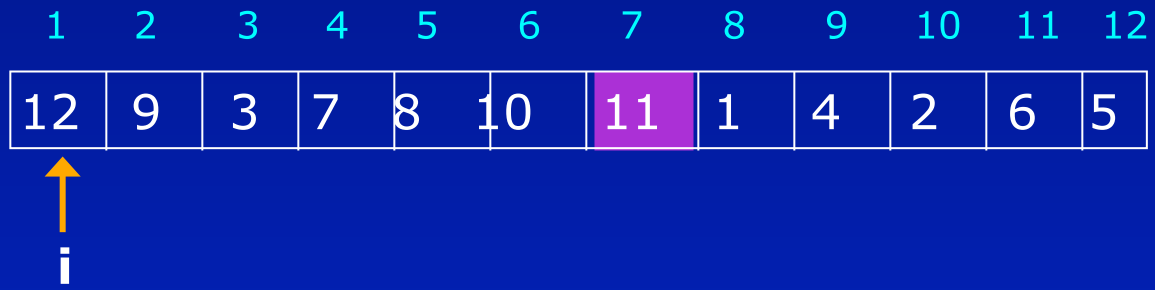
max_heapify (H, 1)

max_heapify (H, 3)

max_heapify (H, 7)



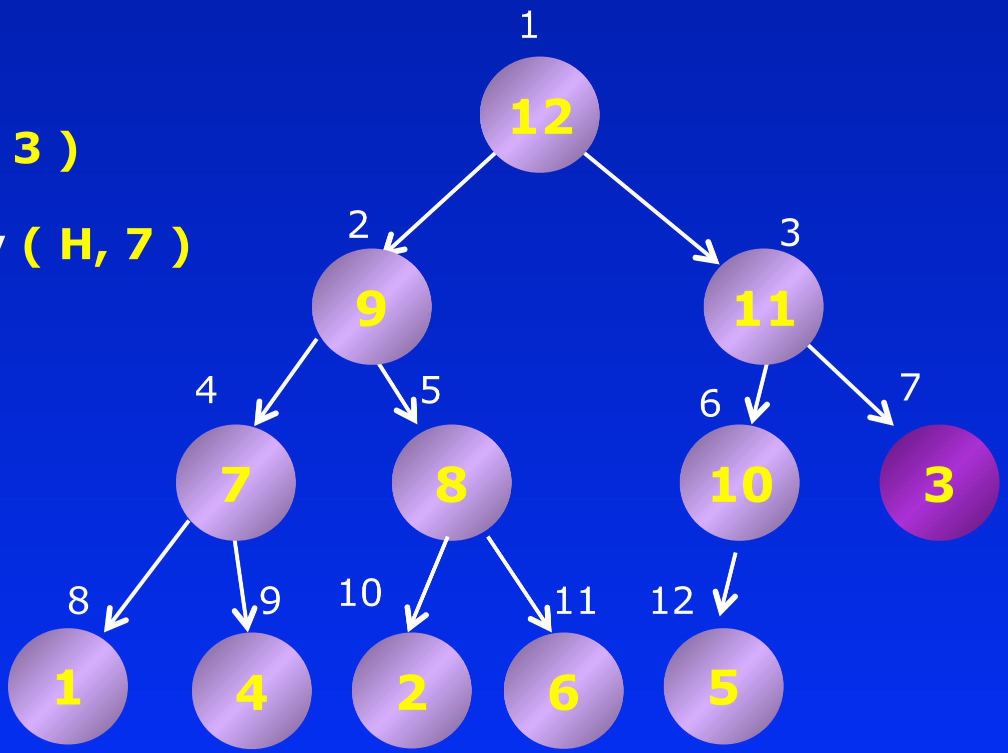
build_heapify-Funktion



max_heapify (H, 1)

max_heapify (H, 3)

max_heapify (H, 7)



build_heapify-Funktion

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

H

size

```
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.

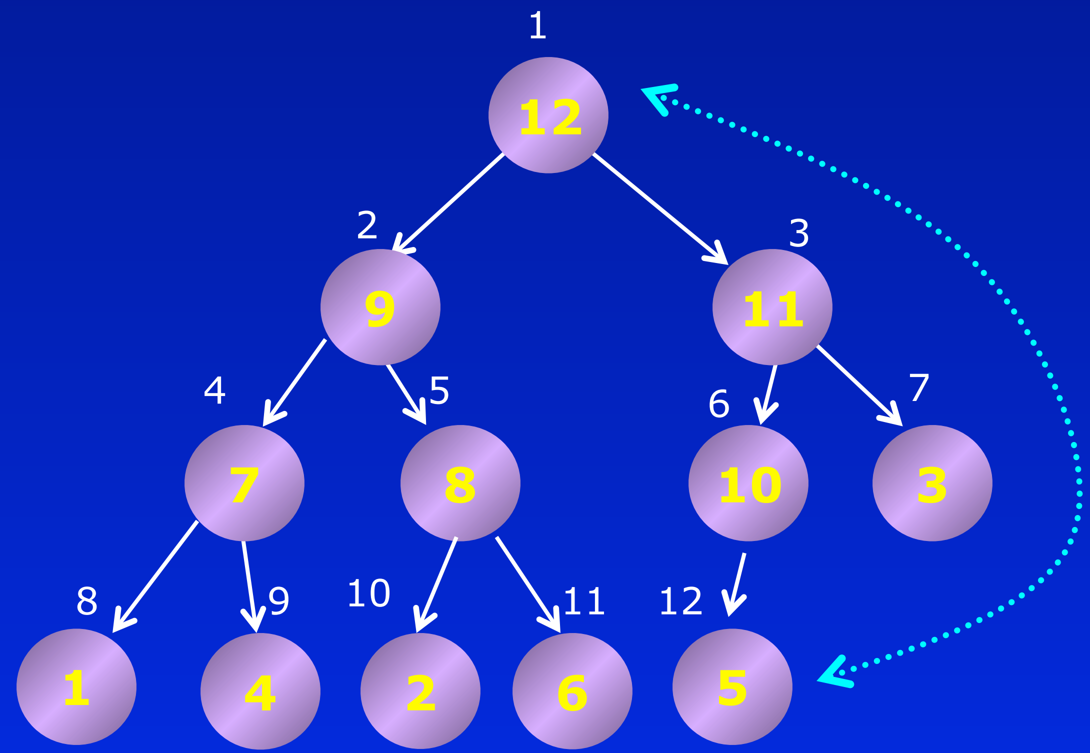
2 Das Element an der Wurzel des Heaps wird gegen das letzte Element des Heaps vertauscht.

3 Die Heap-Größe wird um eins dekrementiert.

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

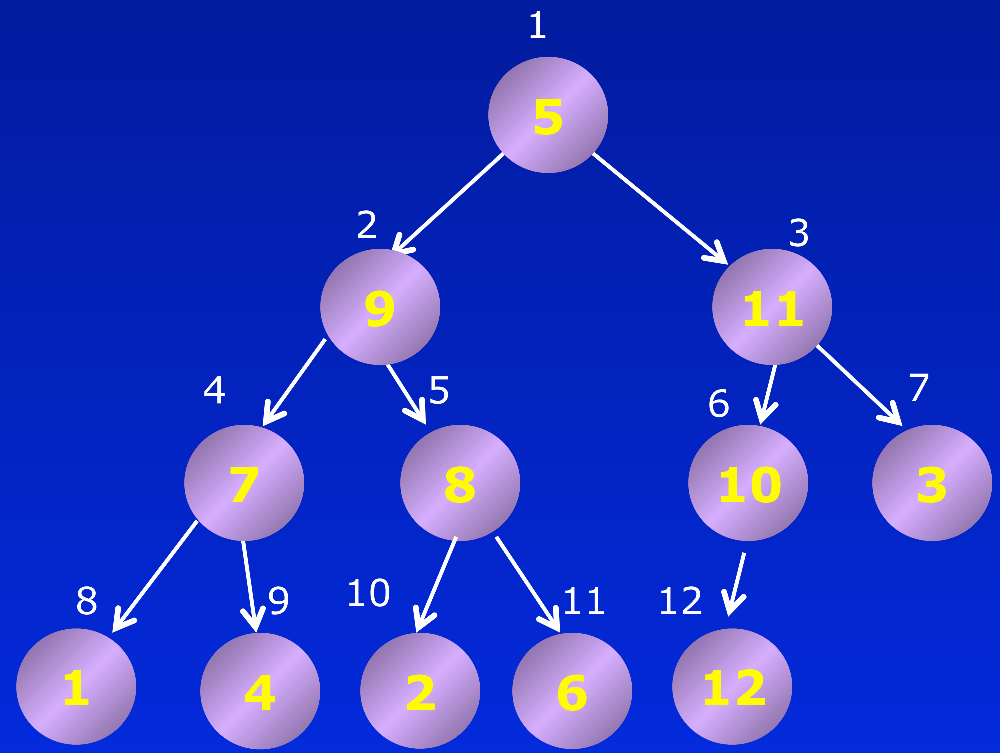
5 Die Schritte **2** bis **4** werden wiederholt, solange der Heap größer oder gleich zwei ist.

Heapsort-Funktion



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

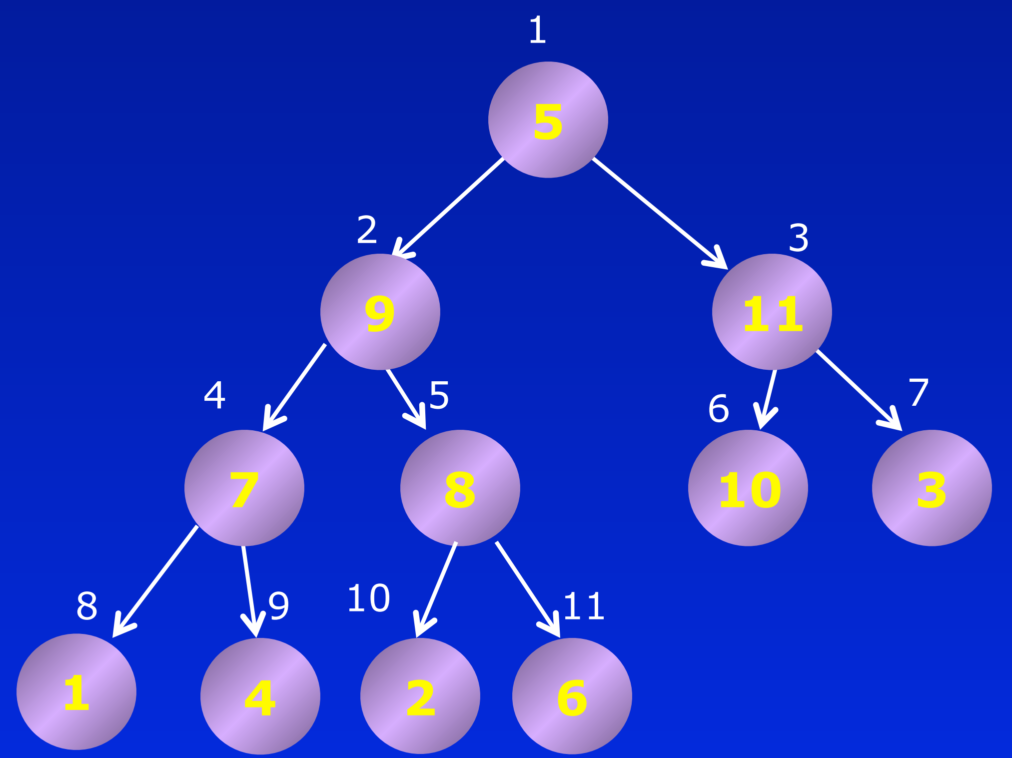
Heapsort-Funktion



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



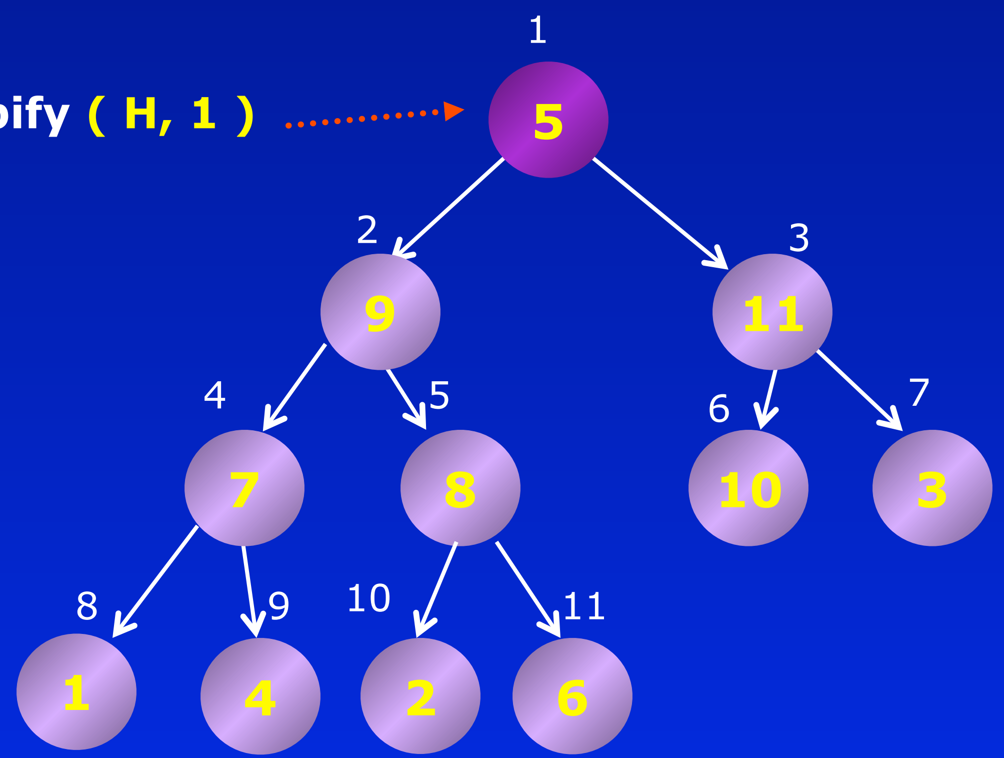
nicht mehr in
dem Heap!

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

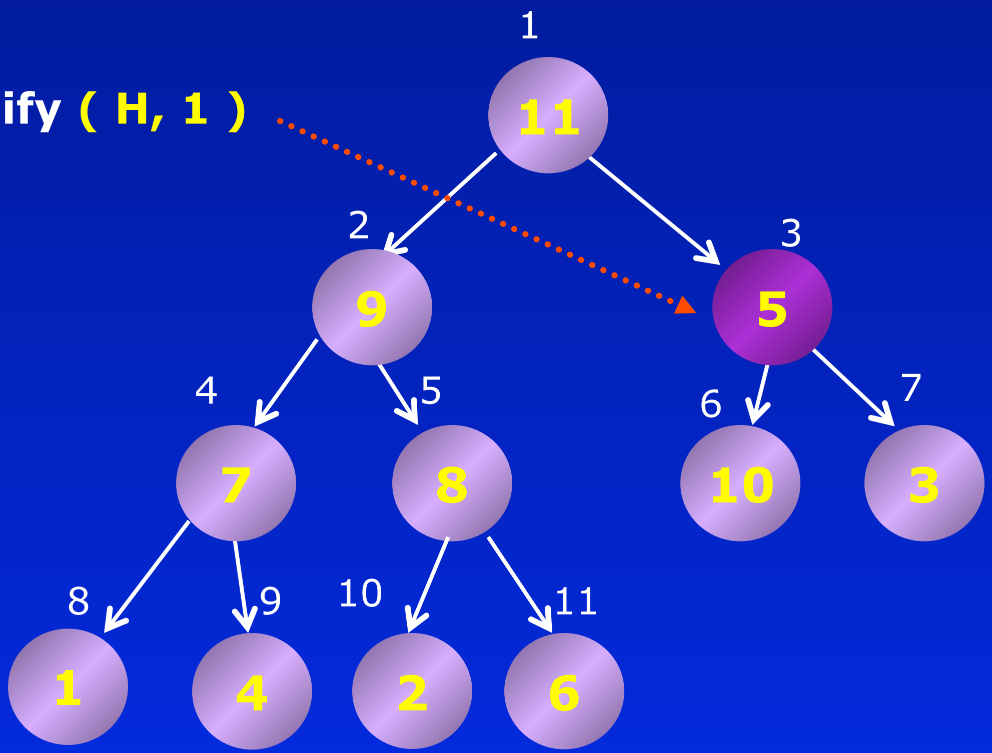
max_heapify (H, 1)



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

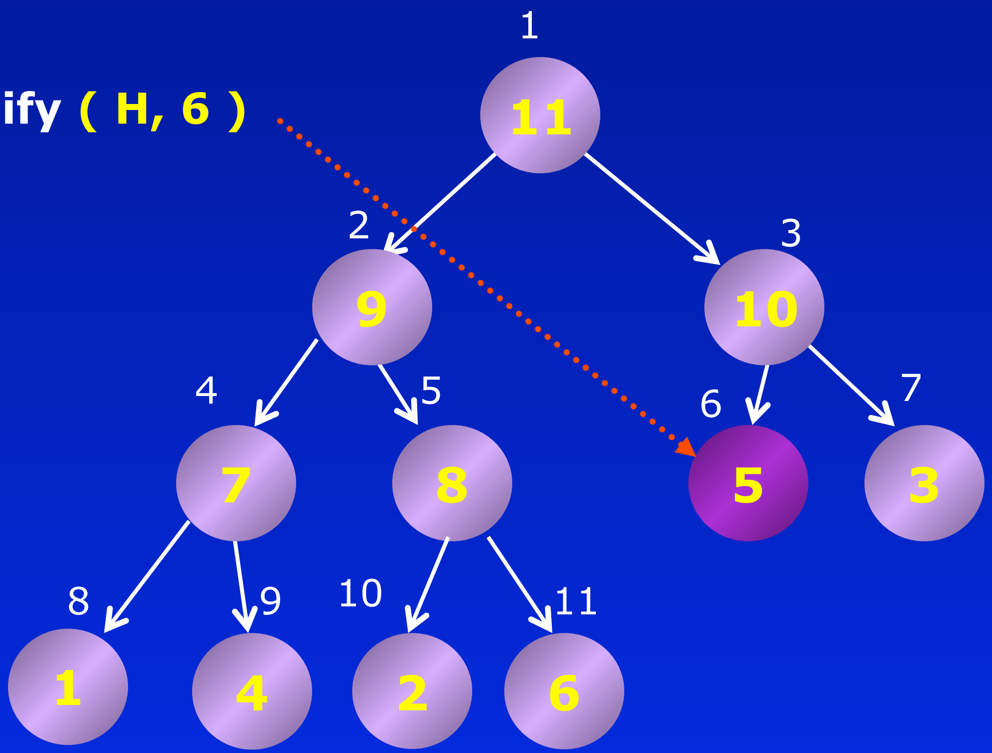
max_heapify (H, 1)



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

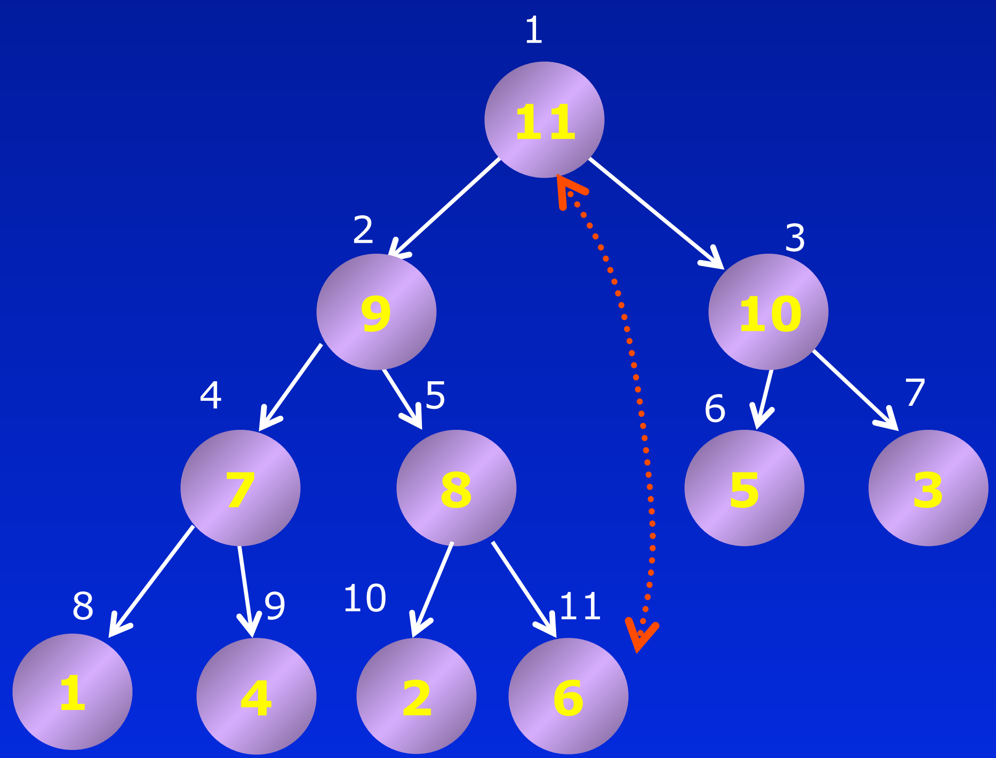
Heapsort-Funktion

max_heapify (H, 6)



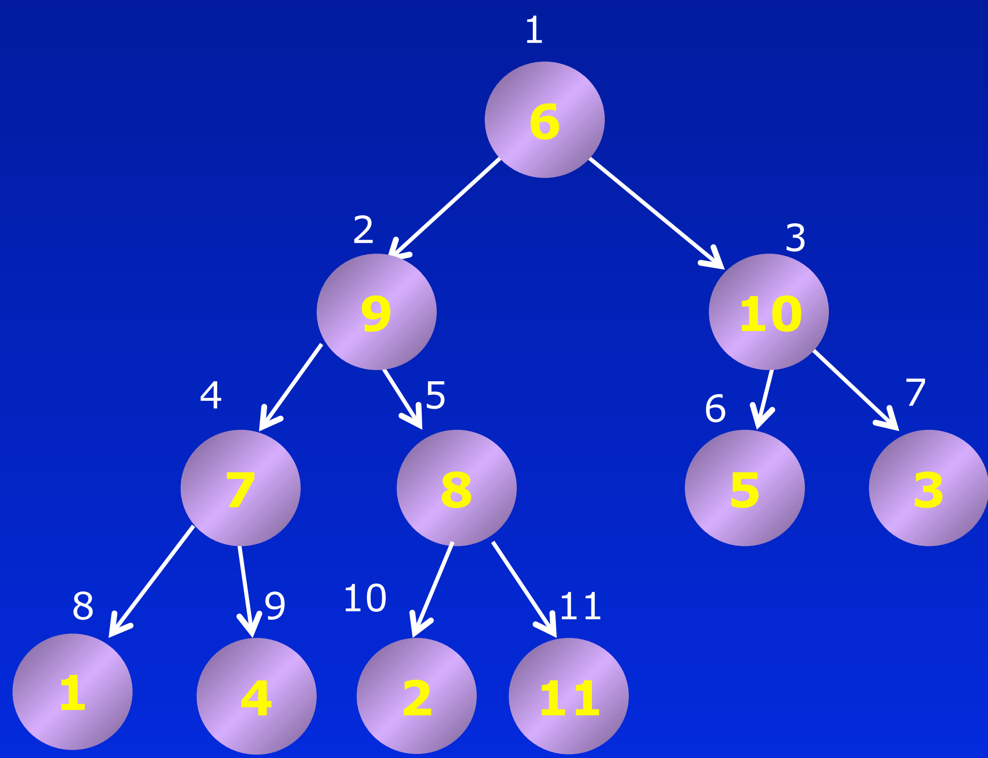
0	1	2	3	4	5	6	7	8	9	10	11	12
11	11	9	10	7	8	5	3	1	4	2	6	12

Heapsort-Funktion



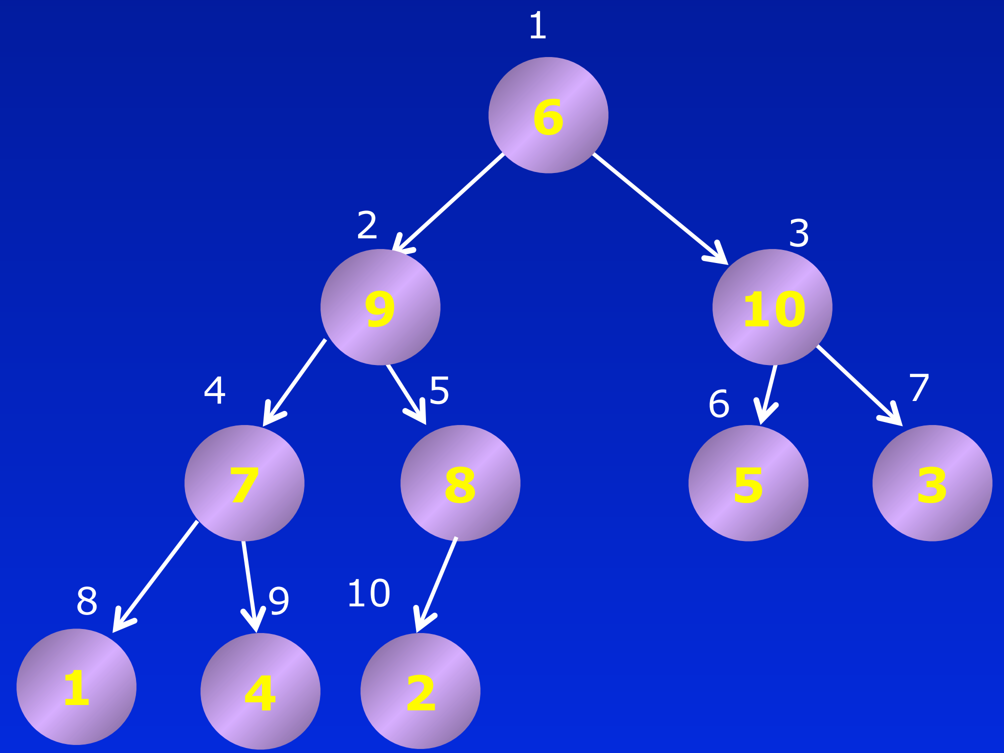
0	1	2	3	4	5	6	7	8	9	10	11	12
11	6	9	10	7	8	5	3	1	4	2	11	12

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

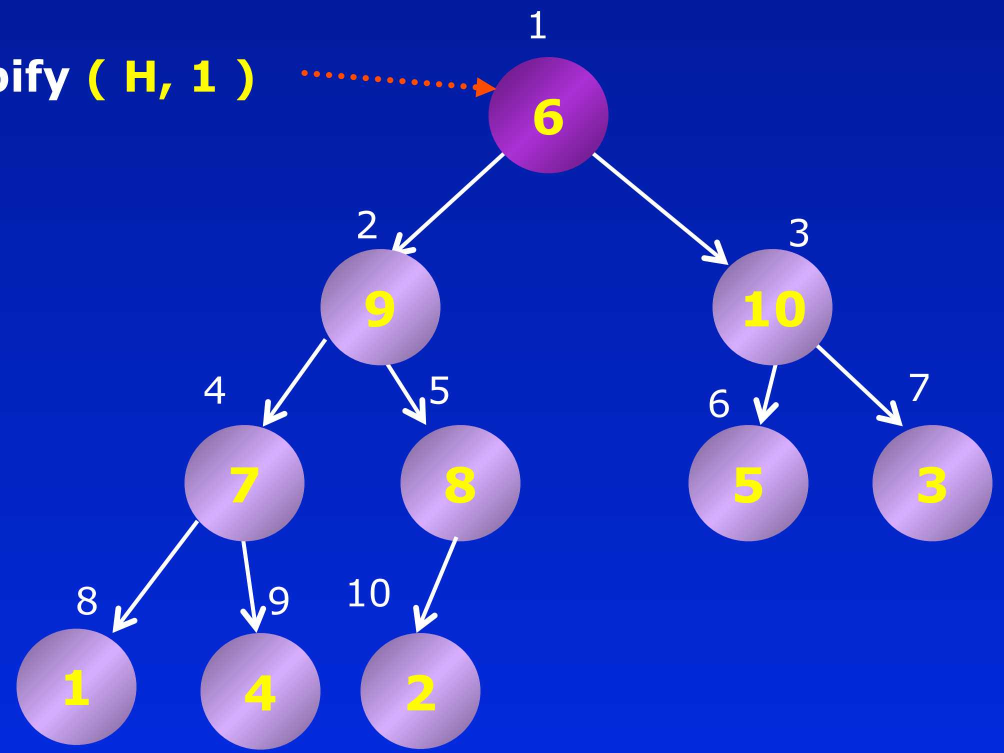
Heapsort-Funktion



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

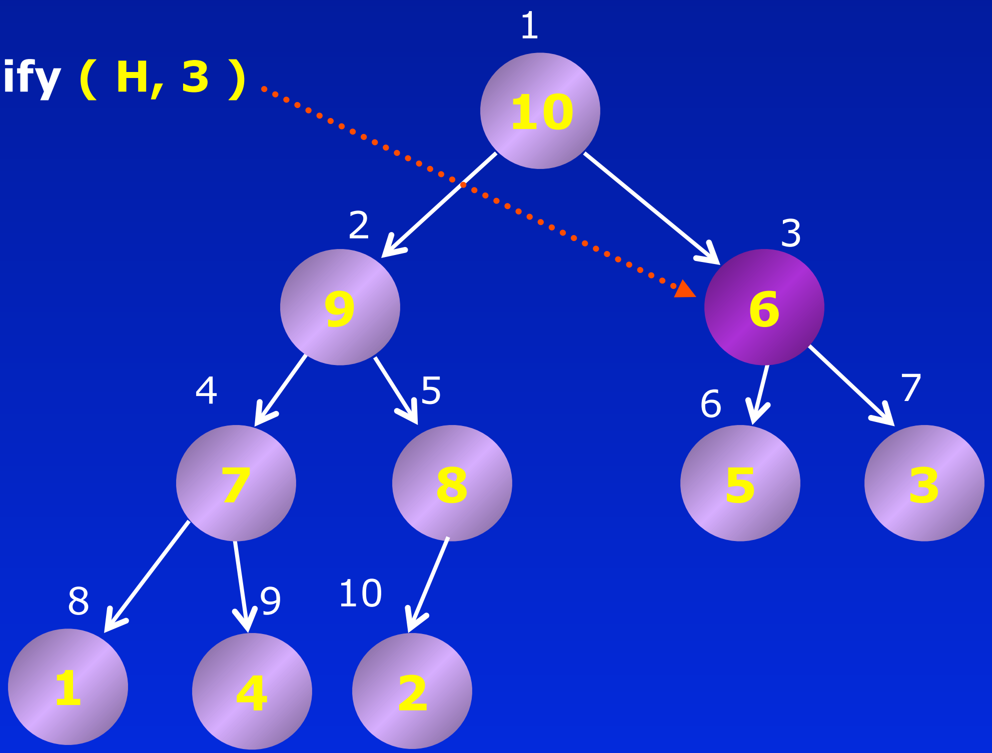
max_heapify (H, 1)



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

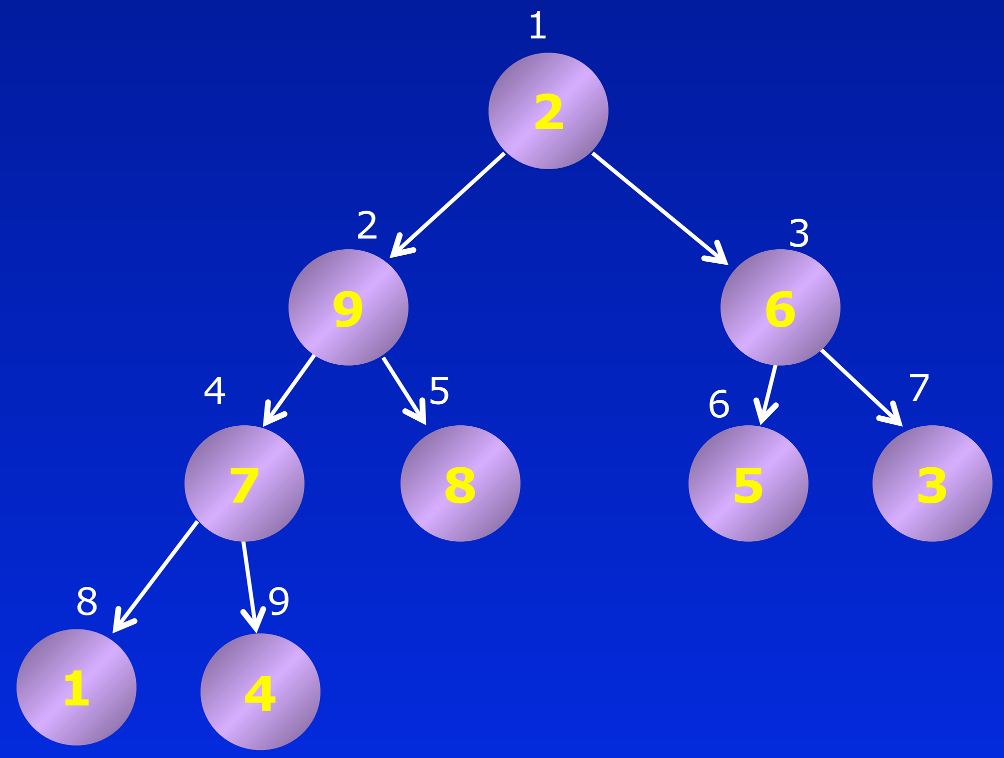
Heapsort-Funktion

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

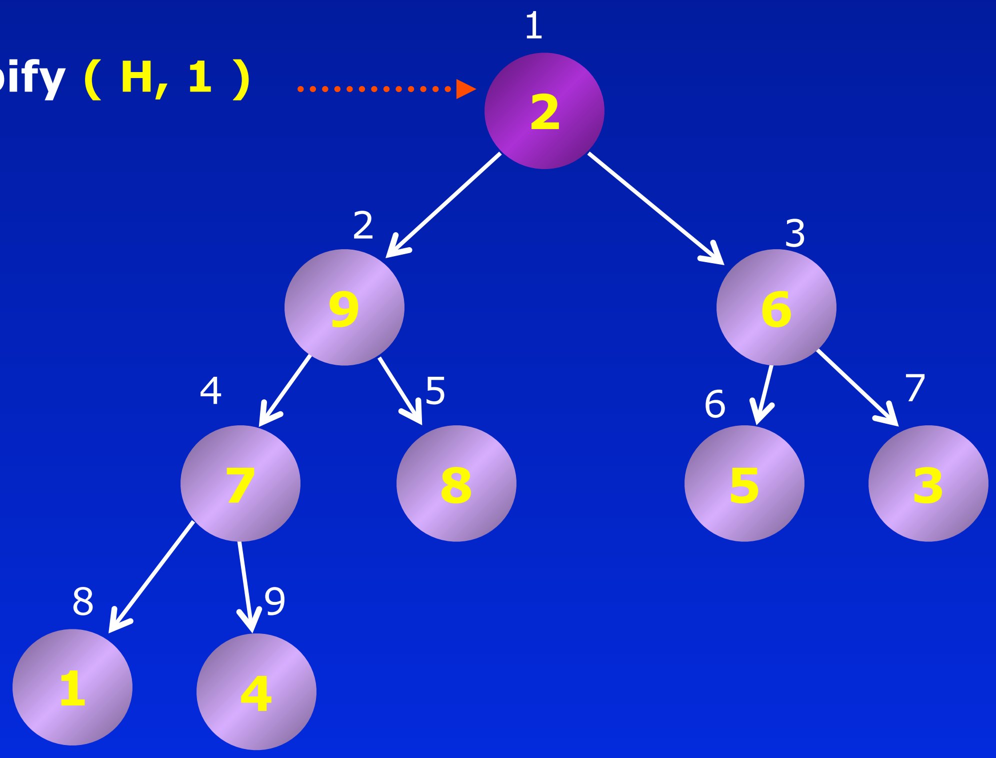
Heapsort-Funktion



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

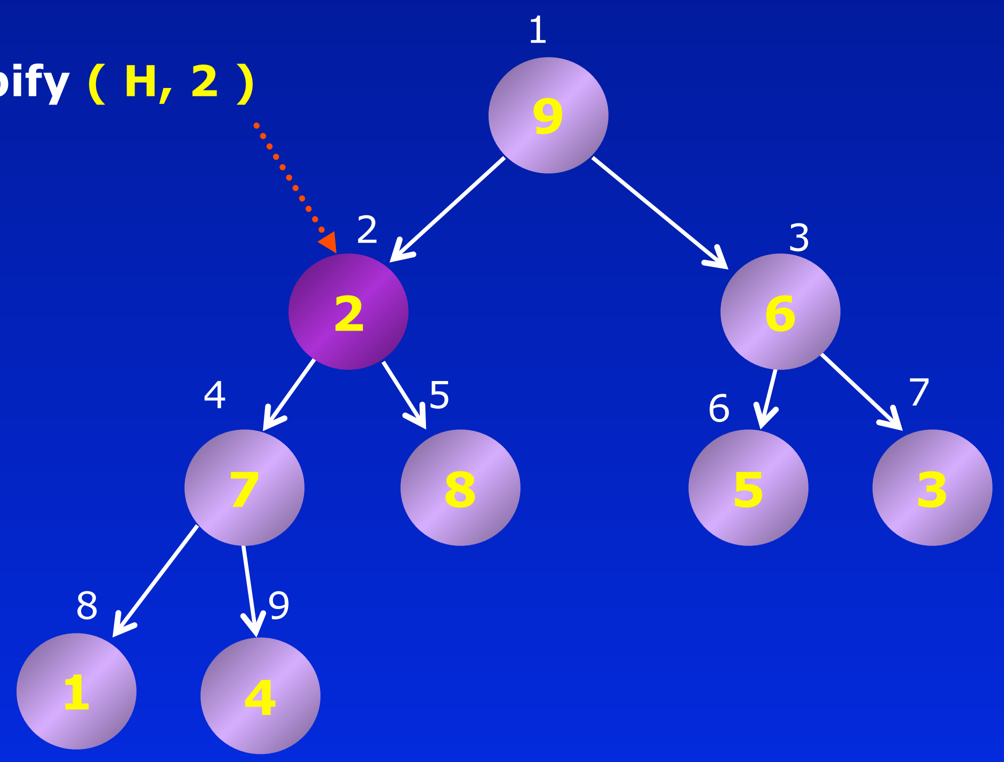
max_heapify (H, 1)



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

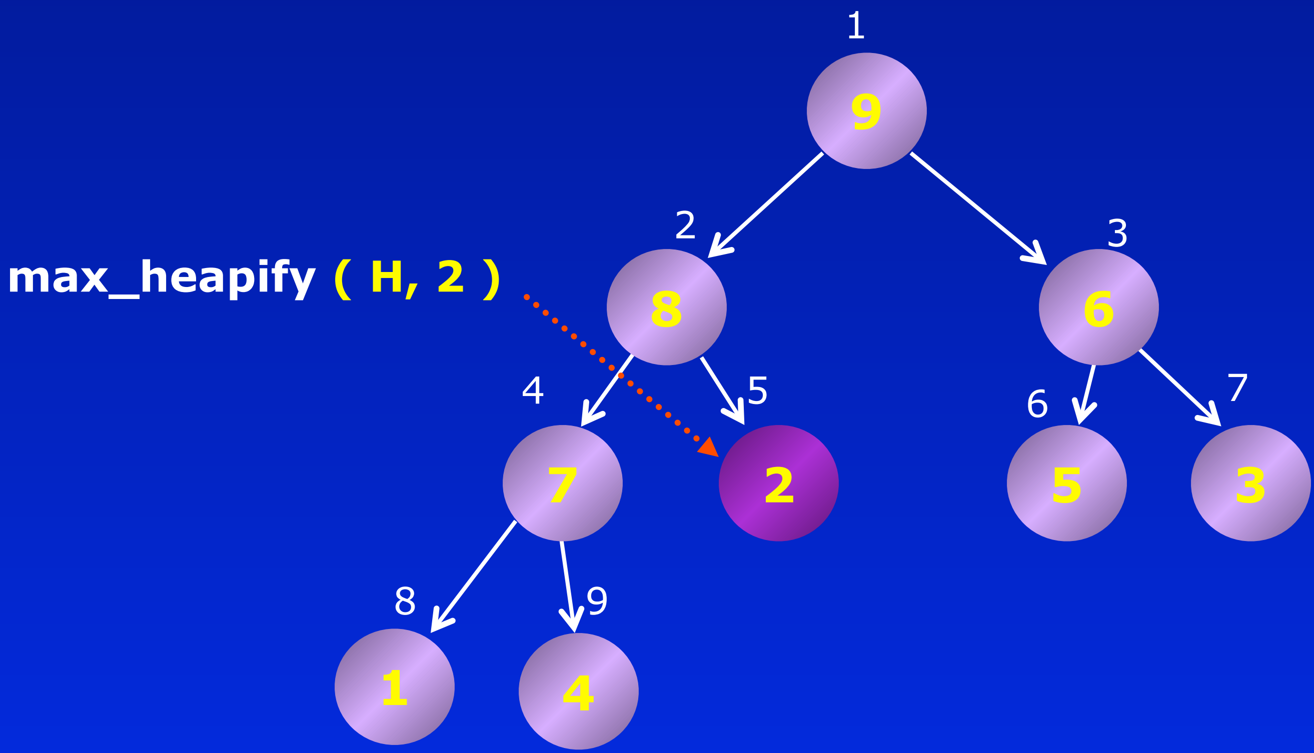
Heapsort-Funktion

max_heapify (H, 2)



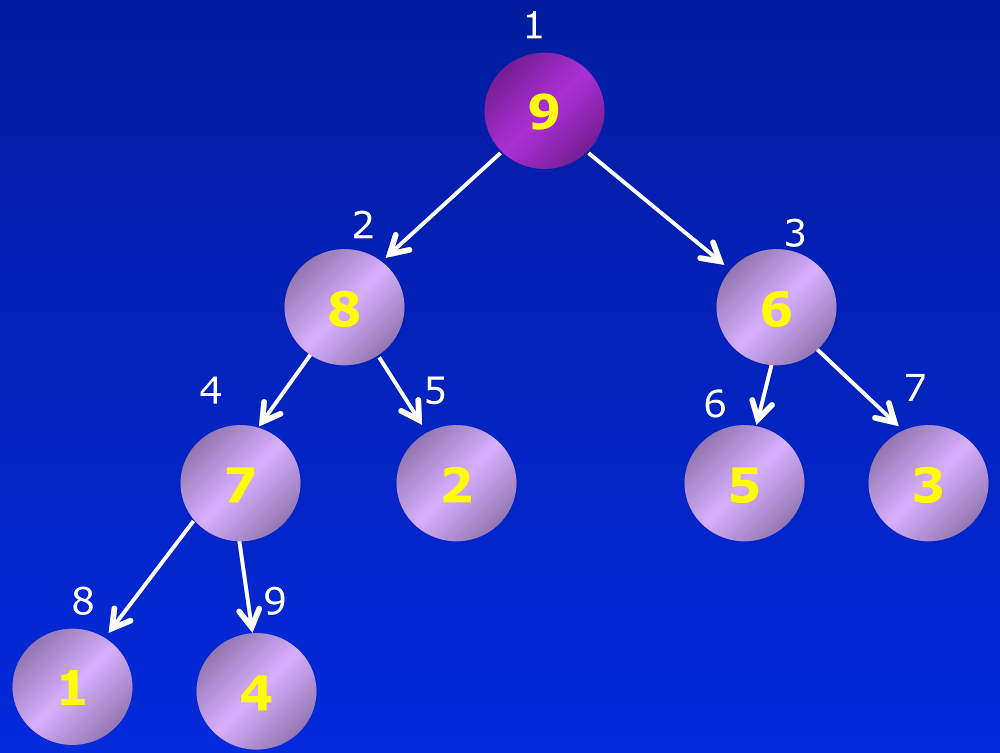
0	1	2	3	4	5	6	7	8	9	10	11	12
9	9	2	6	7	8	5	3	1	4	10	11	12

Heapsort-Funktion



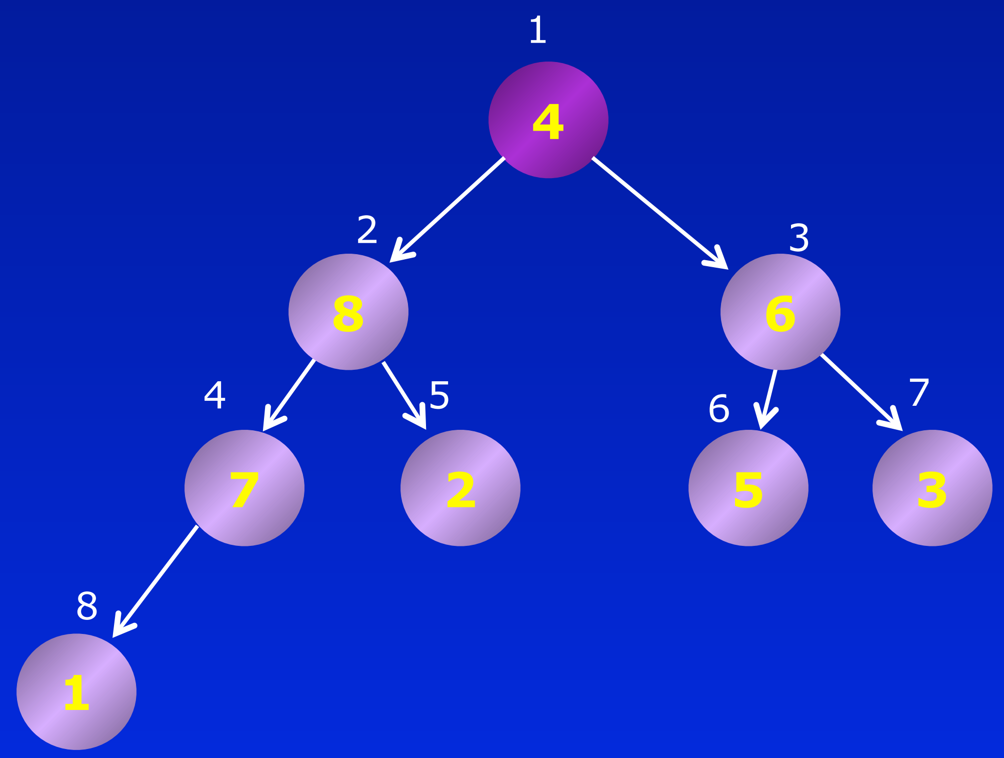
0	1	2	3	4	5	6	7	8	9	10	11	12
9	9	8	6	7	2	5	3	1	4	10	11	12

Heapsort-Funktion



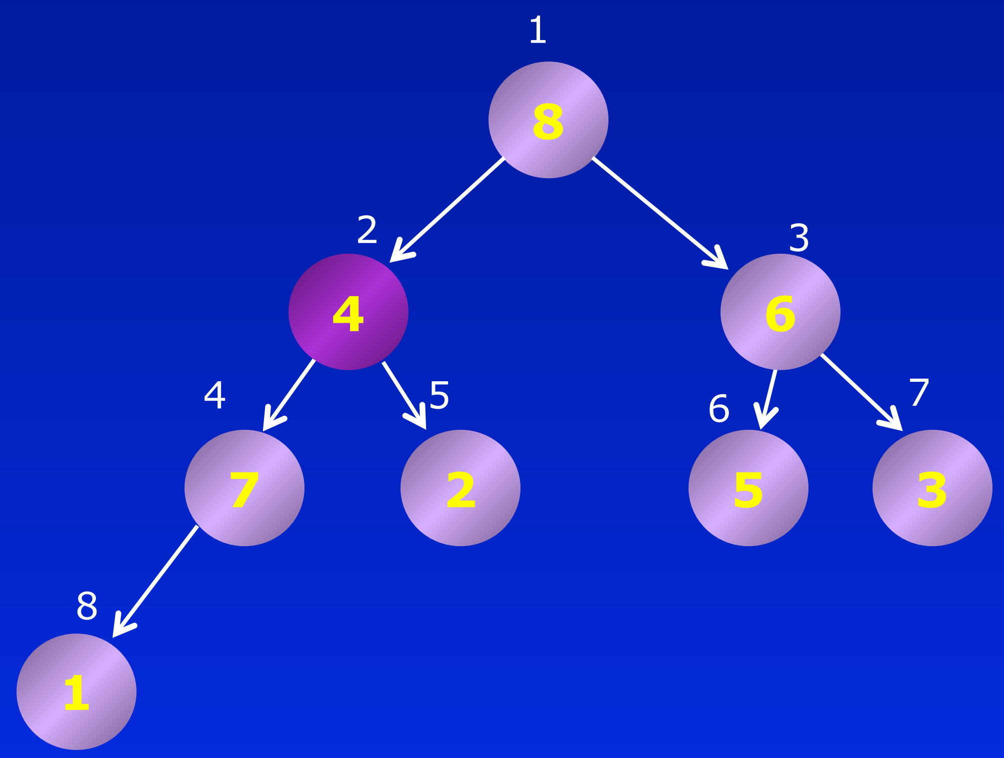
0	1	2	3	4	5	6	7	8	9	10	11	12
9	9	8	6	7	2	5	3	1	4	10	11	12

Heapsort-Funktion



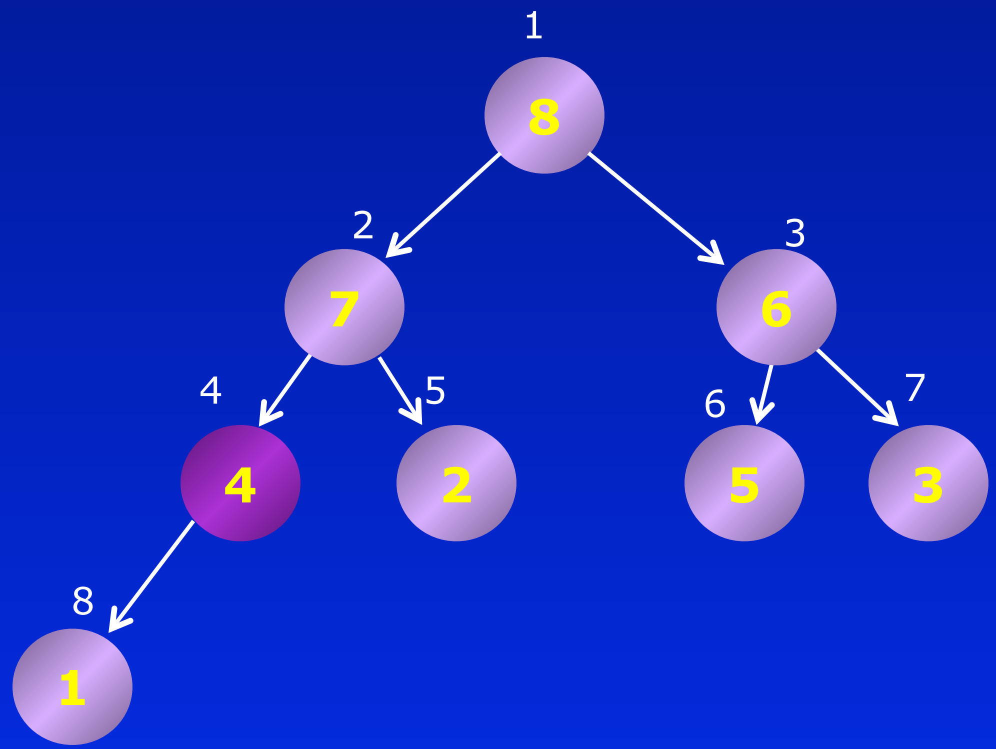
0	1	2	3	4	5	6	7	8	9	10	11	12
8	4	8	6	7	2	5	3	1	9	10	11	12

Heapsort-Funktion



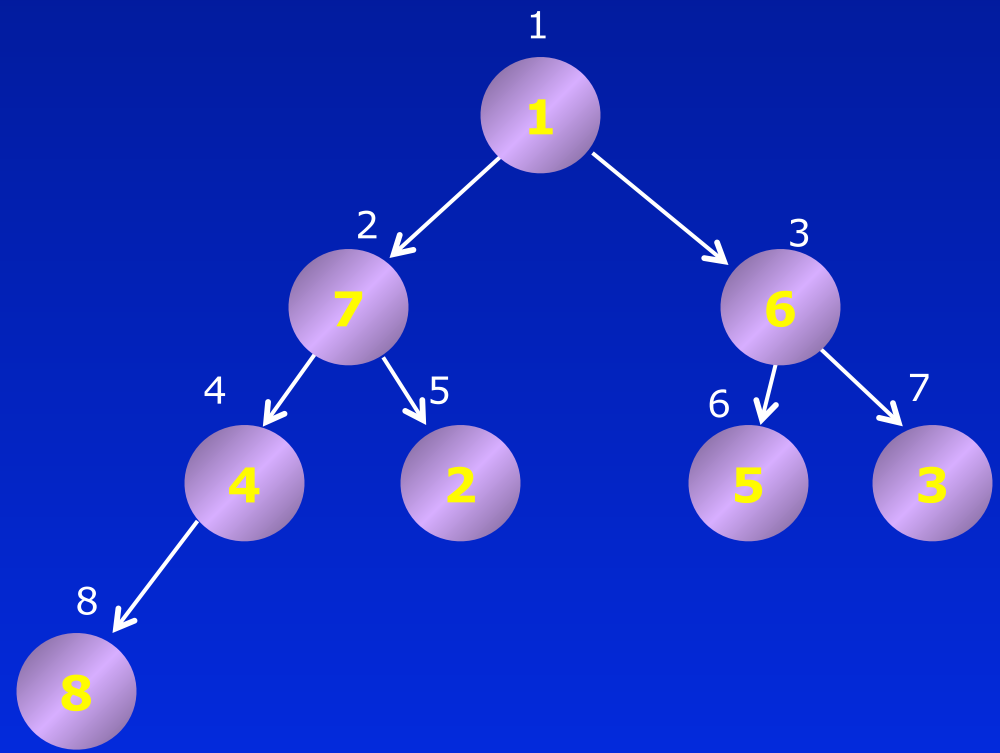
0	1	2	3	4	5	6	7	8	9	10	11	12
8	8	4	6	7	2	5	3	1	9	10	11	12

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

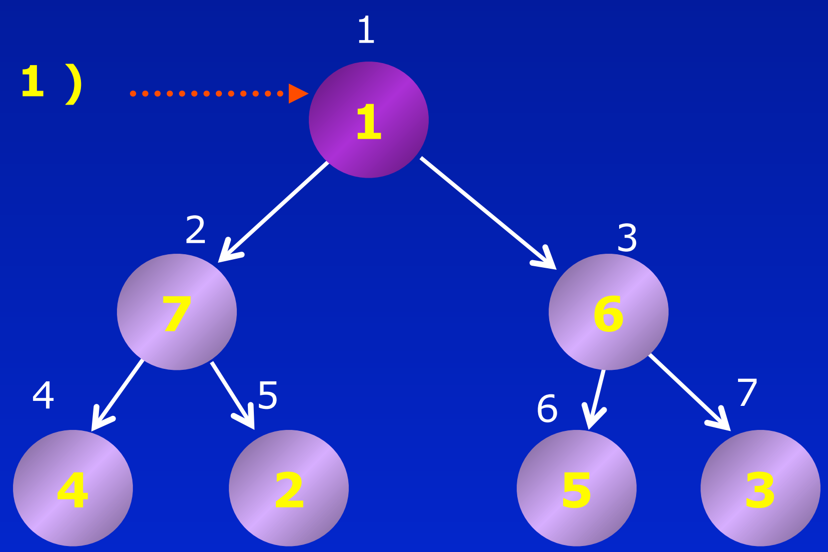
Heapsort-Funktion



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

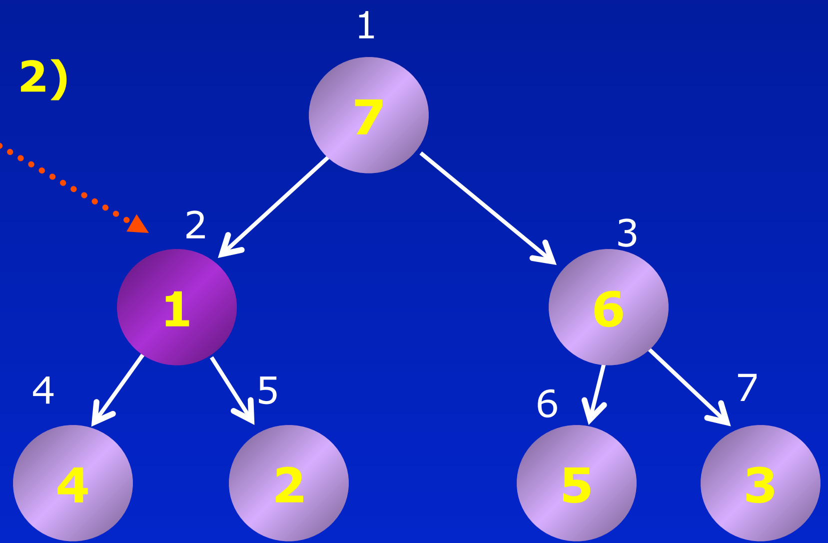
max_heapify (H, 1)



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

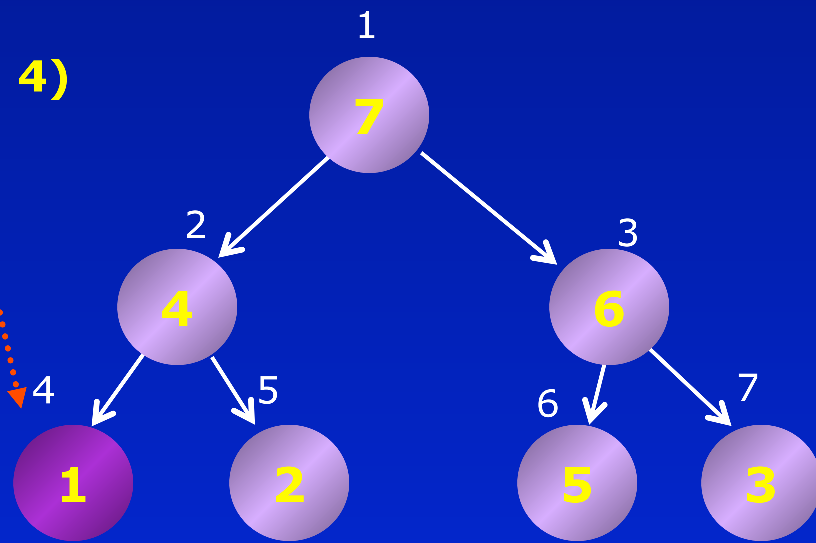
max_heapify (H, 2)



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

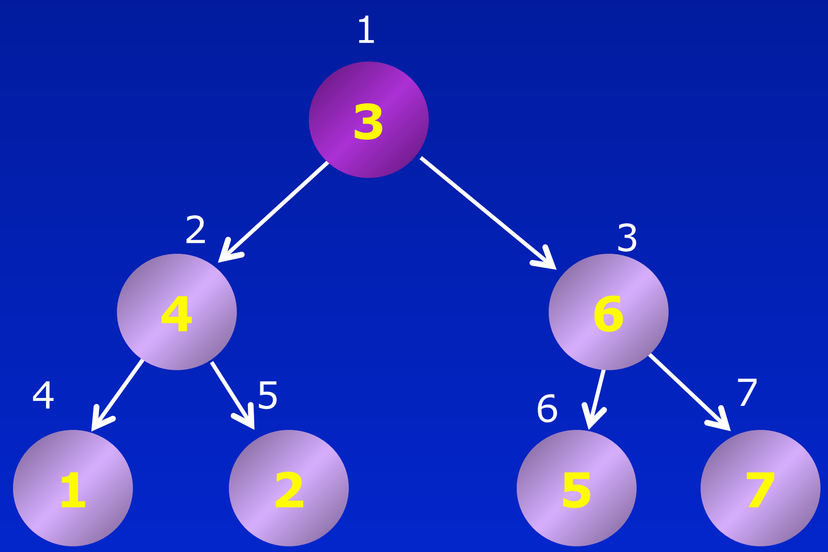
Heapsort-Funktion

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

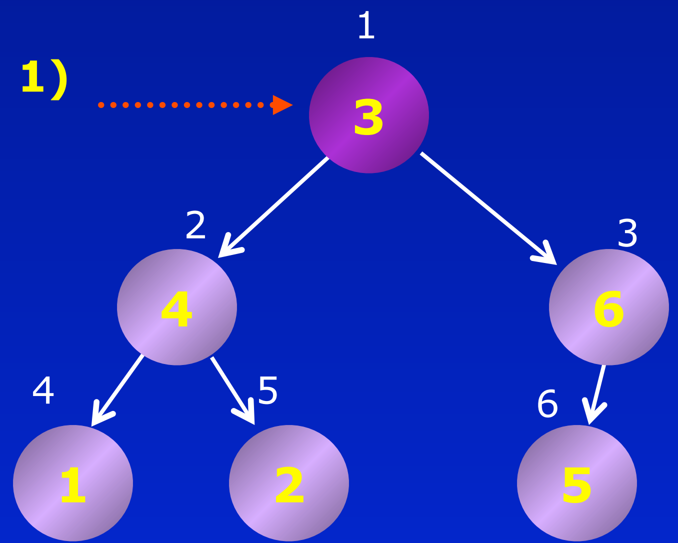
Heapsort-Funktion



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

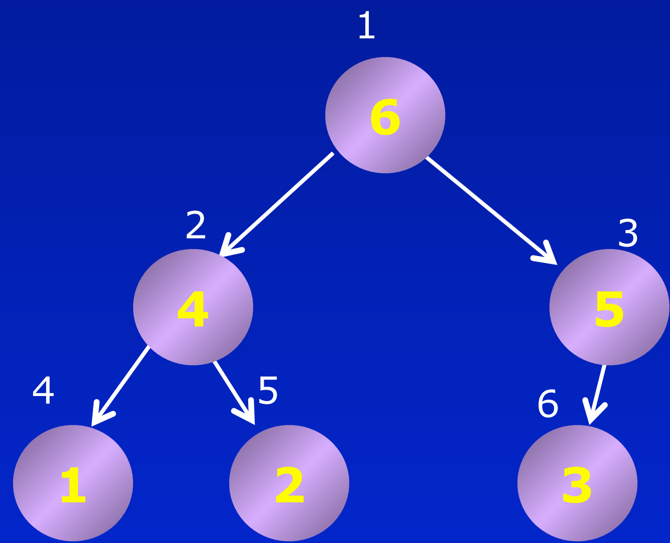
Heapsort-Funktion

max_heapify (H, 1)



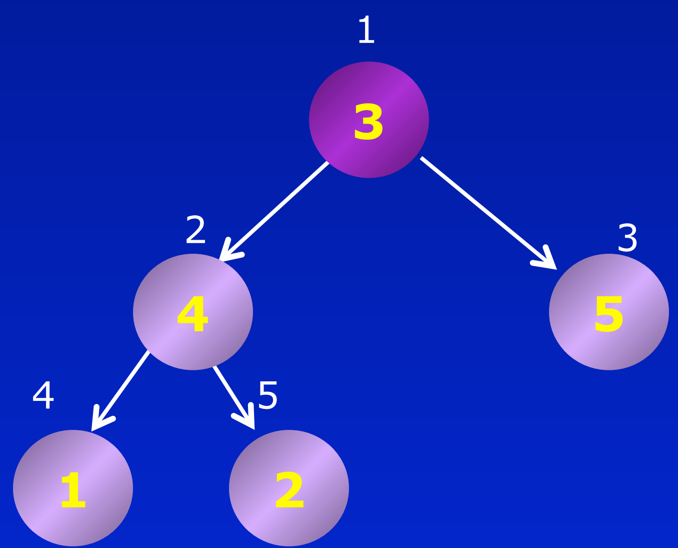
0	1	2	3	4	5	6	7	8	9	10	11	12
6	3	4	6	1	2	5	7	8	9	10	11	12

Heapsort-Funktion



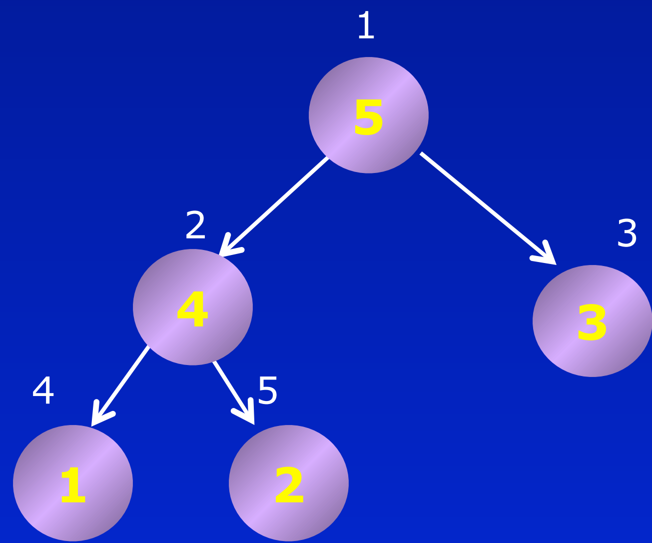
0	1	2	3	4	5	6	7	8	9	10	11	12
6	6	4	5	1	2	3	7	8	9	10	11	12

Heapsort-Funktion



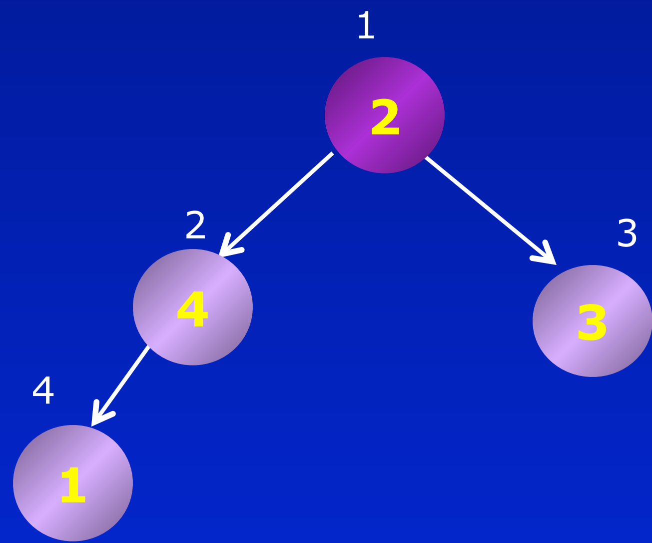
0	1	2	3	4	5	6	7	8	9	10	11	12
5	3	4	5	1	2	6	7	8	9	10	11	12

Heapsort-Funktion



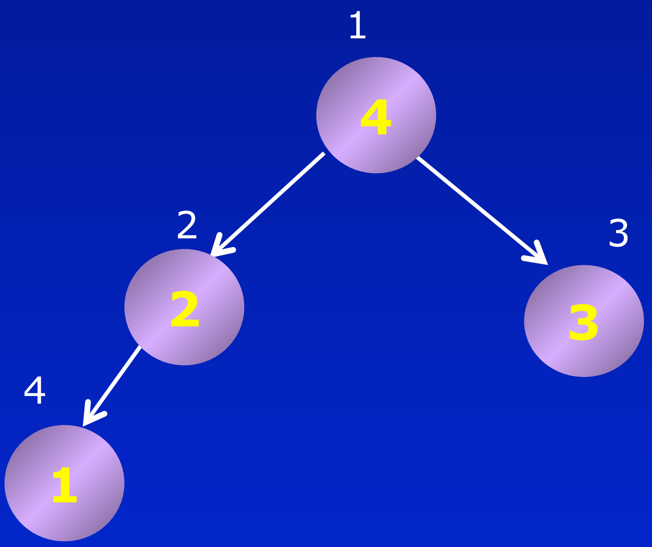
0	1	2	3	4	5	6	7	8	9	10	11	12
5	5	4	3	1	2	6	7	8	9	10	11	12

Heapsort-Funktion



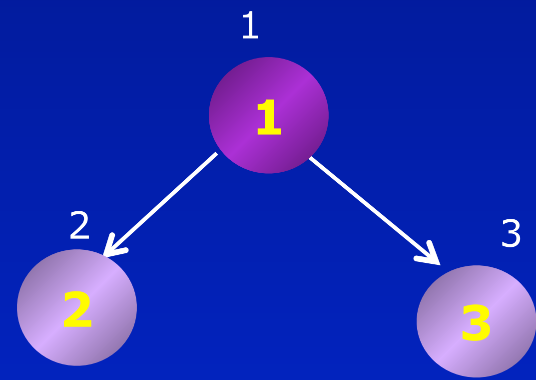
0	1	2	3	4	5	6	7	8	9	10	11	12
4	2	4	3	1	5	6	7	8	9	10	11	12

Heapsort-Funktion



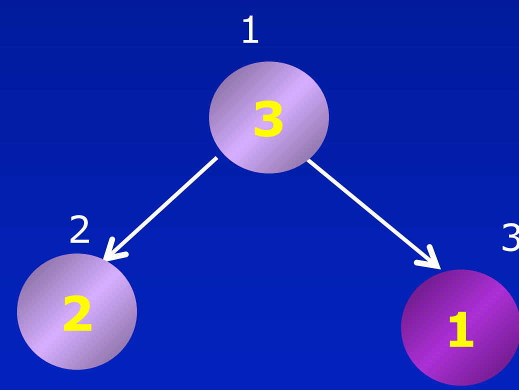
0	1	2	3	4	5	6	7	8	9	10	11	12
4	2	4	3	1	5	6	7	8	9	10	11	12

Heapsort-Funktion



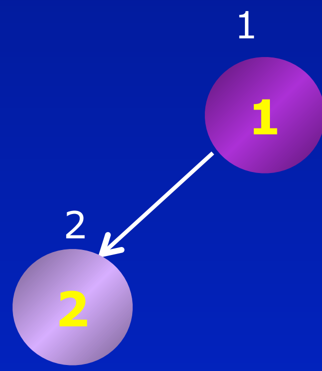
0	1	2	3	4	5	6	7	8	9	10	11	12
3	2	4	3	4	5	6	7	8	9	10	11	12

Heapsort-Funktion



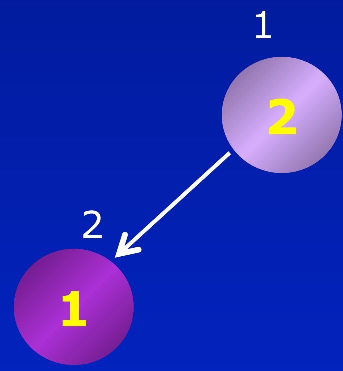
0	1	2	3	4	5	6	7	8	9	10	11	12
3	3	2	1	4	5	6	7	8	9	10	11	12

Heapsort-Funktion



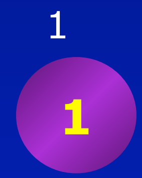
0	1	2	3	4	5	6	7	8	9	10	11	12
2	1	2	3	4	5	6	7	8	9	10	11	12

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



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

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

Zeitaufwand?

```
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)
```

build_max_heap(H) → $n \cdot \log(n)/2$

for i in range(heap_size(H), 1, -1): → $n-1$

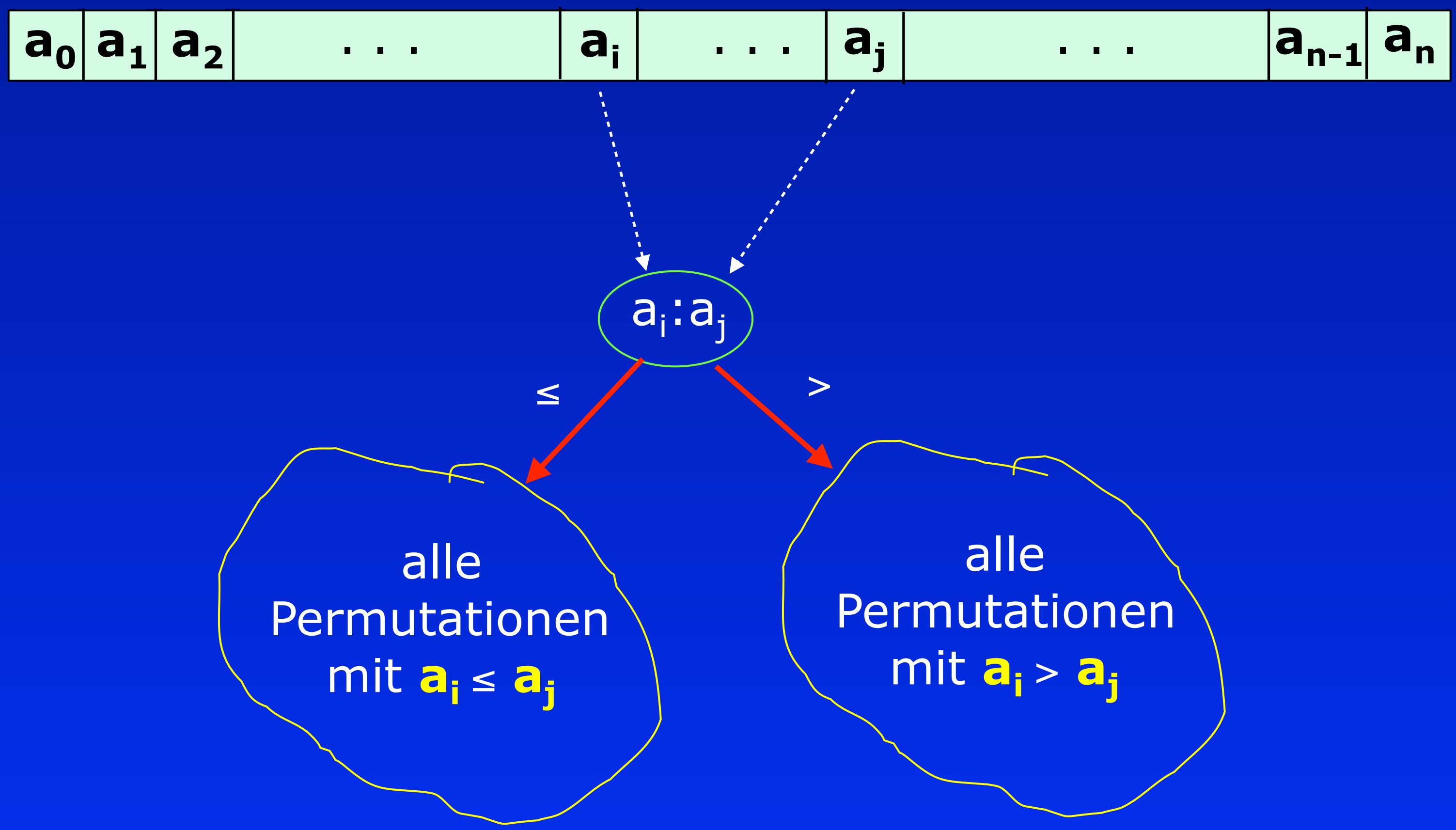
max_heapify(H, 1) → $\log(n)$

$\left.\begin{matrix} n-1 \\ \log(n) \end{matrix}\right\} n \cdot \log(n)$

Zeitkomplexität = $\boxed{O(n \cdot \log(n))}$

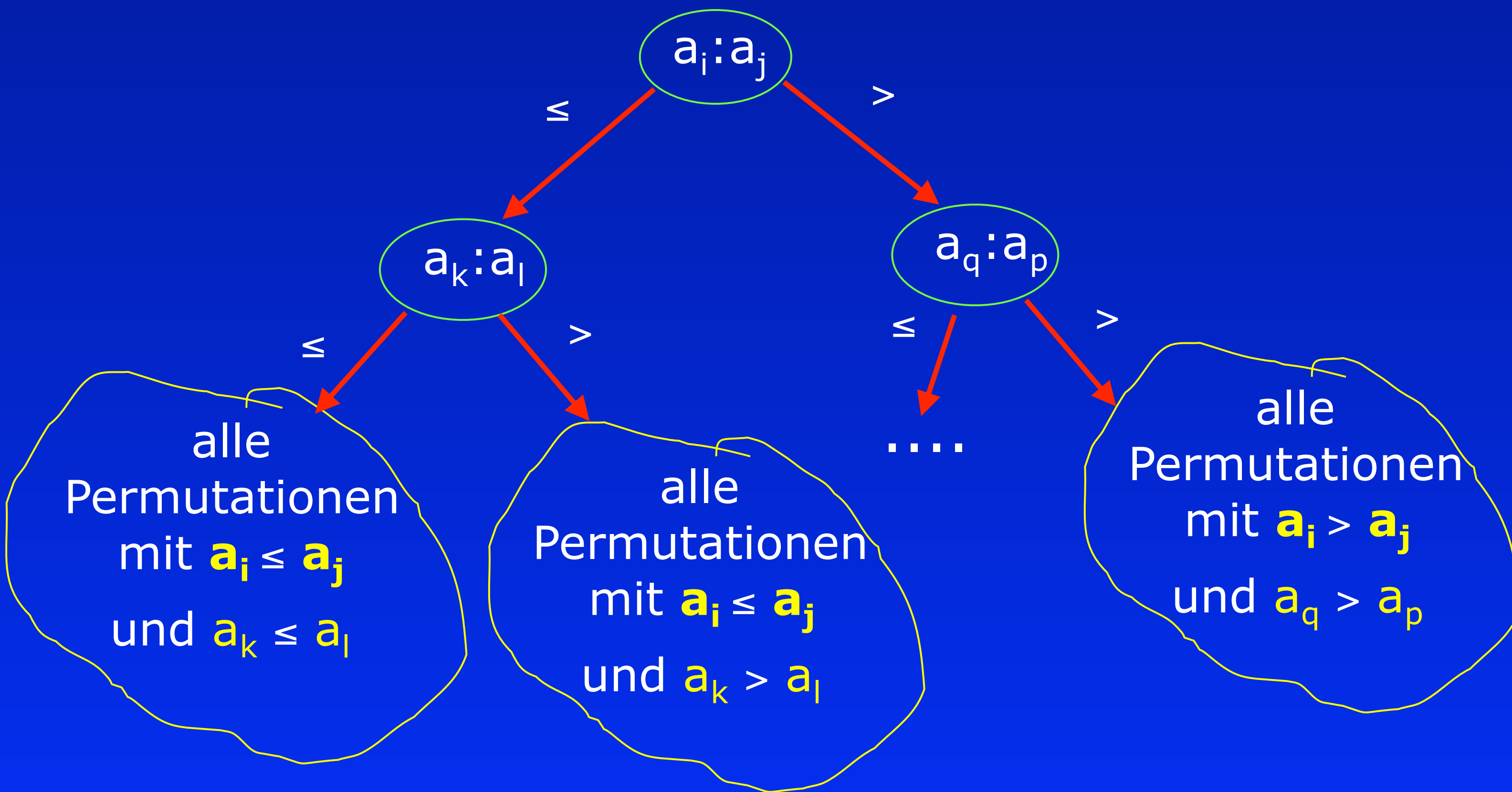
Ist es möglich besser zu sortieren?

Entscheidungsbaum

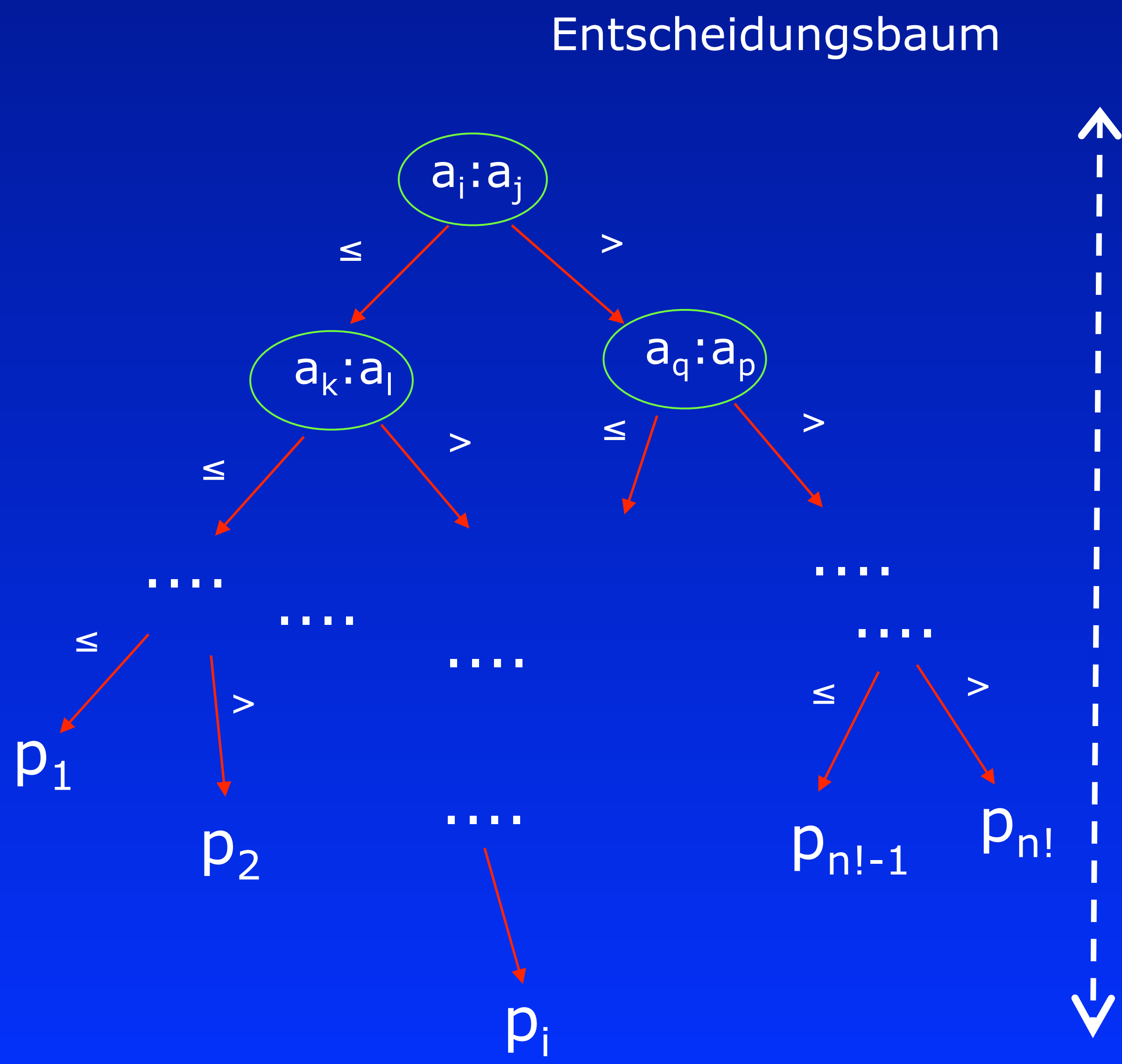


Ist es möglich besser zu sortieren?

Entscheidungsbaum

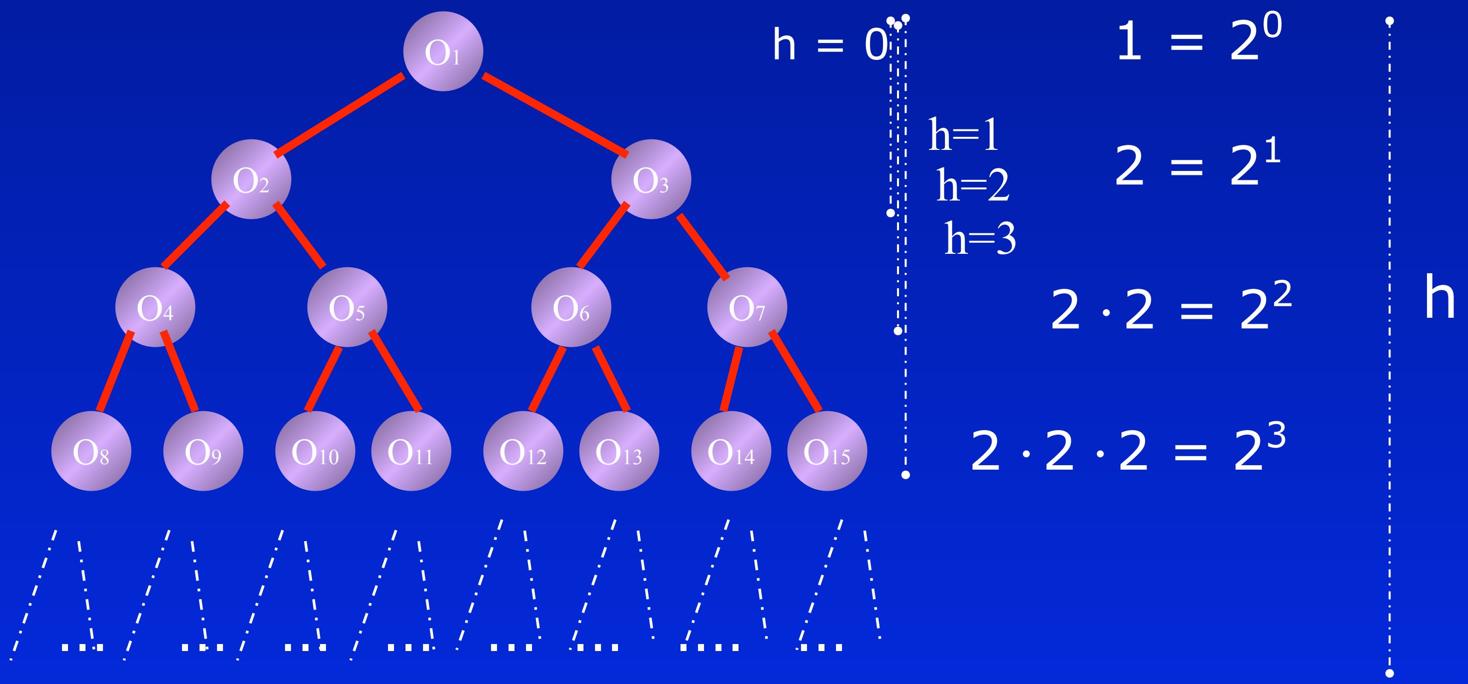


Ist es möglich besser zu sortieren?



Der längste Weg im Entscheidungsbaum zwischen der Wurzel und einem Blatt stellt die größte Anzahl der Vergleiche dar, die der Algorithmus ausführt.

Ist es möglich besser zu sortieren?



Ein Binärbaum mit hohem **h** hat maximal $\mathbf{2^h}$ Blätter

$$\mathbf{h \approx \log_2(n)}$$

Ist es möglich schneller zu sortieren? Nein!

Behauptung: Jeder Vergleichsalgorithmus braucht im schlimmsten Fall mindestens $\Omega(\mathbf{n \cdot \log(n)})$ Vergleiche.

Wir brauchen nur die Höhe des Entscheidungsbaums zu bestimmen.

Nehmen wir an, wir haben einen Baum mit Höhe **h** und **b** Blättern

$\mathbf{n!} \leq \mathbf{b}$ denn jede Permutation ist im Baum

Weil ein Binärbaum mit Höhe **h** maximal $\mathbf{2^h}$ Blätter hat, gilt

$$\mathbf{n!} \leq \mathbf{b} \leq \mathbf{2^h}$$

Wenn wir den Logarithmus berechnen, gilt

$\mathbf{\log(n!)} \leq \log(\mathbf{2^h})$ weil der Logarithmus eine monoton wachsende Funktion ist.

$$\mathbf{\log(n!)} \leq \mathbf{h}$$

$\mathbf{\log(n!)} = \Omega\,(\mathbf{n \cdot \log(n)})$ nach Stirling

$$\boxed{\mathbf{h} = \Omega\,(\mathbf{n \cdot \log(n)})}$$

Counting sort

Wenn die Daten, die sortiert werden sollen, ganzzahlige Werte mit einem kleinen Wertebereich zwischen **0** und **k** sind, ist es möglich, Zahlen zu sortieren ohne diese direkt zu vergleichen.

Die Zeitkomplexität, die man dabei hat, ist linear.

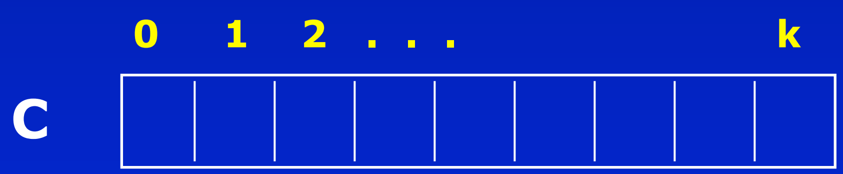
$$T(n) = O(n)$$

Anzahl der Daten, die sortiert werden sollen

Counting sort

Der "Counting sort"-Algorithmus benutzt zwei Hilfsfelder.

1. ein **C**-Feld, das so groß ist wie der Wertebereich (**k=8**) der Zahlen



2. ein **B**-Feld, in dem die sortierten Zahlen am Ende gespeichert werden.

Counting sort

Nehmen wir an, wir wollen die Zahlen des **A**-Feldes sortieren.

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

Das C-Feld wird mit Nullen initialisiert.

0 1 2 . . . 8

C	0	0	0	0	0	0	0	0	0

Wertebereich = (0-8)

Counting sort

	0	1	2	3	.	.	.									.	.	.	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7	

	0	1	2	.	.	.			8
C	0	0	0	0	0	0	0	0	0

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[0]

	0	1	2	. . .					8
C	0	0	0	0	0	0	0	0	0

Counting sort

A

0	1	2	3	. . .												. . .	n
4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[0] → C[A[0]]

C

0	1	2	. . .					8
0	0	0	0	0	0	0	0	0

Counting sort

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[0] C[A[0]]

0 1 2 . . . 8

C	0	0	0	0	0	0	0	0	0

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[0] → C[A[0]] → C[A[0]]=C[A[0]]+1

	0	1	2	. . .					8
C	0	0	0	0	0	0	0	0	0

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[0]

C[A[0]]=C[A[0]]+1

	0	1	2	. . .					8
C	0	0	0	0	1	0	0	0	0

Counting sort

	0	1	2	3	.	.	.									.	.	.	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7	

A[1]

C[A[1]]=C[A[1]]+1

	0	1	2	.	.	.			8
C	0	0	0	0	1	0	0	0	0

Counting sort

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[1]

0 1 2 . . . 8

C	0	0	0	0	1	0	0	0	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[2]

	0	1	2	. . .					8
C	0	0	0	0	1	0	0	0	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[2]

C[A[2]]=C[A[2]]+1

	0	1	2	. . .					8
C	0	0	0	0	1	0	0	0	1

Counting sort

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[2]

C[A[2]]=C[A[2]]+1

0 1 2 . . . 8

C	1	0	0	0	1	0	0	0	1

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[3]

C[A[3]]=C[A[3]]+1

	0	1	2	. . .					8
C	1	1	0	0	1	0	0	0	1

Counting sort

0 1 2 3 . . . . . . n

A																		
	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[4]

C[A[4]]=C[A[4]]+1

0 1 2 . . . 8

C									
	2	1	0	0	1	0	0	0	1

Counting sort

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[5]

C[A[5]]=C[A[5]]+1

0 1 2 . . . 8

C	2	1	0	0	1	0	1	0	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[6]

C[A[6]]=C[A[6]]+1

	0	1	2	. . .					8
C	2	1	0	0	1	0	1	1	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[7]

C[A[7]]=C[A[7]]+1

	0	1	2	. . .					8
C	2	2	0	0	1	0	1	1	1

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[8]

C[A[8]]=C[A[8]]+1

	0	1	2	. . .					8
C	2	2	0	0	2	0	1	1	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[9]

C[A[9]]=C[A[9]]+1

	0	1	2	. . .					8
C	2	3	0	0	2	0	1	1	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[10]

C[A[10]]=C[A[10]]+1

	0	1	2	. . .					8
C	2	3	0	0	2	1	1	1	1

Counting sort

	0	1	2	3	. . .												. . .	n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[11]

C[A[11]]=C[A[11]]+1

	0	1	2	. . .					8
C	2	3	0	0	2	1	1	2	1

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[12]

C[A[12]]=C[A[12]]+1

	0	1	2	. . .					k
C	2	3	0	0	2	1	1	2	2

Counting sort

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[14]

C[A[14]]=C[A[14]]+1

	0	1	2	. . .					k
C	2	4	0	0	2	1	2	2	2

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[15]

C[A[15]]=C[A[15]]+1

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	2	2

Counting sort

0 1 2 3 . . . . . . n

A																		
	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[16]

C[A[16]]=C[A[16]]+1

0 1 2 . . . k

C									
	2	4	1	0	2	1	2	2	3

Counting sort

	0	1	2	3	. . .											. . .		n
A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[17]

C[A[17]]=C[A[17]]+1

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

= 18

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

+2

	0	1	2	3	. . .											. . .		n
B																		

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	. . .					k
C	2	6	1	0	2	1	2	3	3

	0	1	2	3	. . .											. . .		n
B																		

Counting sort

	0	1	2	.	.	.			k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	.	.	.			k
C	2	6	7	0	2	1	2	3	3

	0	1	2	3	.	.	.									.	.	. n
B																		

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

$$C[i] = C[i] + C[i-1]$$

	0	1	2	. . .					k
C	2	6	7	7	2	1	2	3	3

	0	1	2	3	. . .											. . .		n
B																		

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	. . .					k
C	2	6	7	7	9	1	2	3	3

	0	1	2	3	. . .												. . .	n
B																		

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	. . .					k
C	2	6	7	7	9	10	2	3	3

	0	1	2	3	. . .												. . .	n
B																		

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	. . .					k
C	2	6	7	7	9	10	12	3	3

	0	1	2	3	. . .												. . .	n
B																		

Counting sort

	0	1	2	. . .					k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

	0	1	2	. . .					k
C	2	6	7	7	9	10	12	15	3

	0	1	2	3	. . .											. . .		n
B																		

Counting sort

	0	1	2 . . .						k
C	2	4	1	0	2	1	2	3	3

C[i] = C[i] + C[i-1]

Jetzt beinhalte C[i] alle Elemente, die kleiner oder gleich **i** sind.

	0	1	2 . . .						8
C	2	6	7	7	9	10	12	15	18

9 Zahlen sind kleiner oder gleich 4

	0	1	2	3	. . .												. . .	n
B																		

Counting sort

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[17]

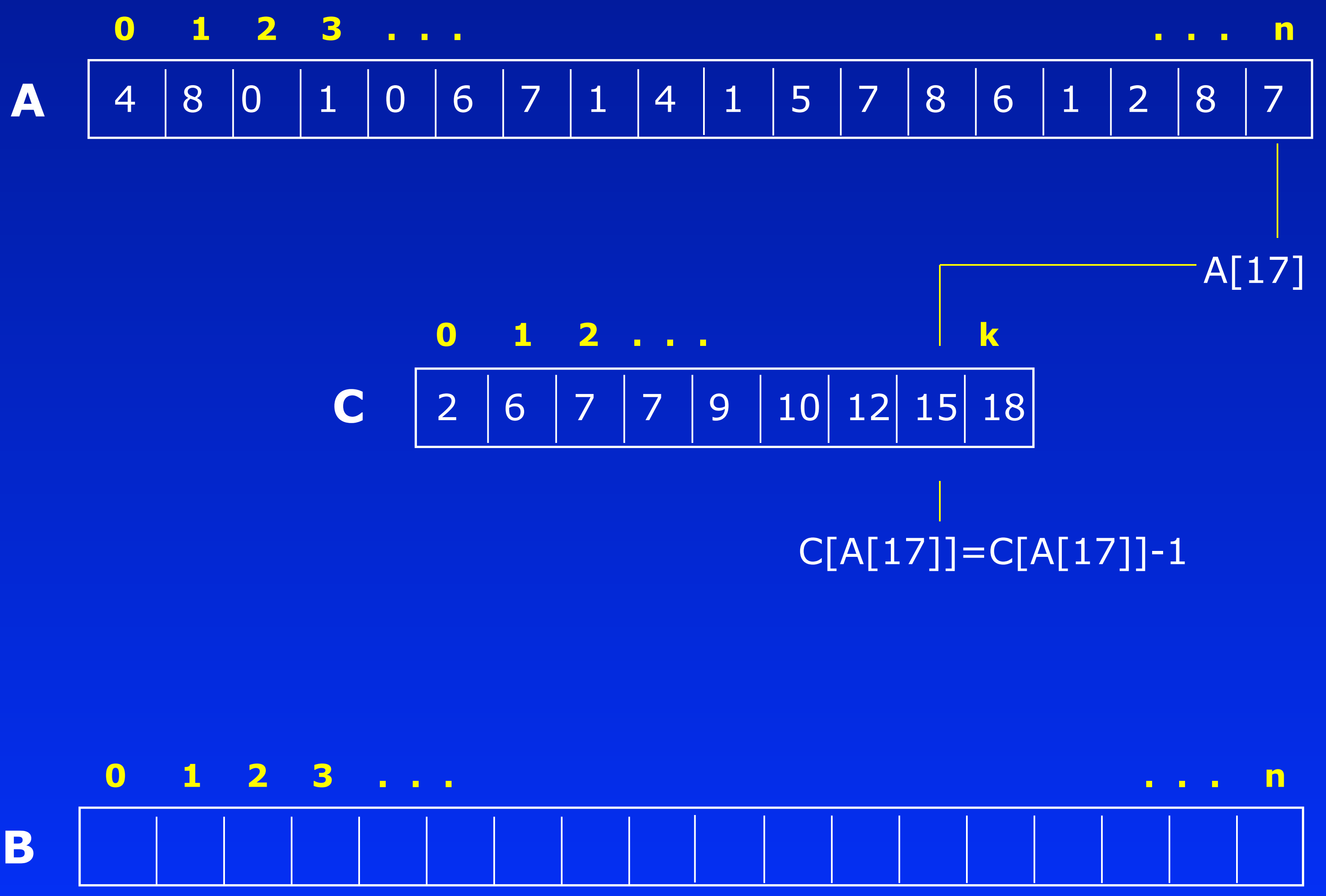
0 1 2 . . . k

C	2	6	7	7	9	10	12	15	18

0 1 2 3 . . . . . . n

B																		

Counting sort



Counting sort

0 1 2 3 . . . . . . n

A

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

A[17]

0 1 2 . . . k

C

2	6	7	7	9	10	12	**14**	18

C[A[17]]=C[A[17]]-1

0 1 2 3 . . . . . . n

B

Counting sort

0 1 2 3 . . . . . . n

A	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[17]

0 1 2 . . . k

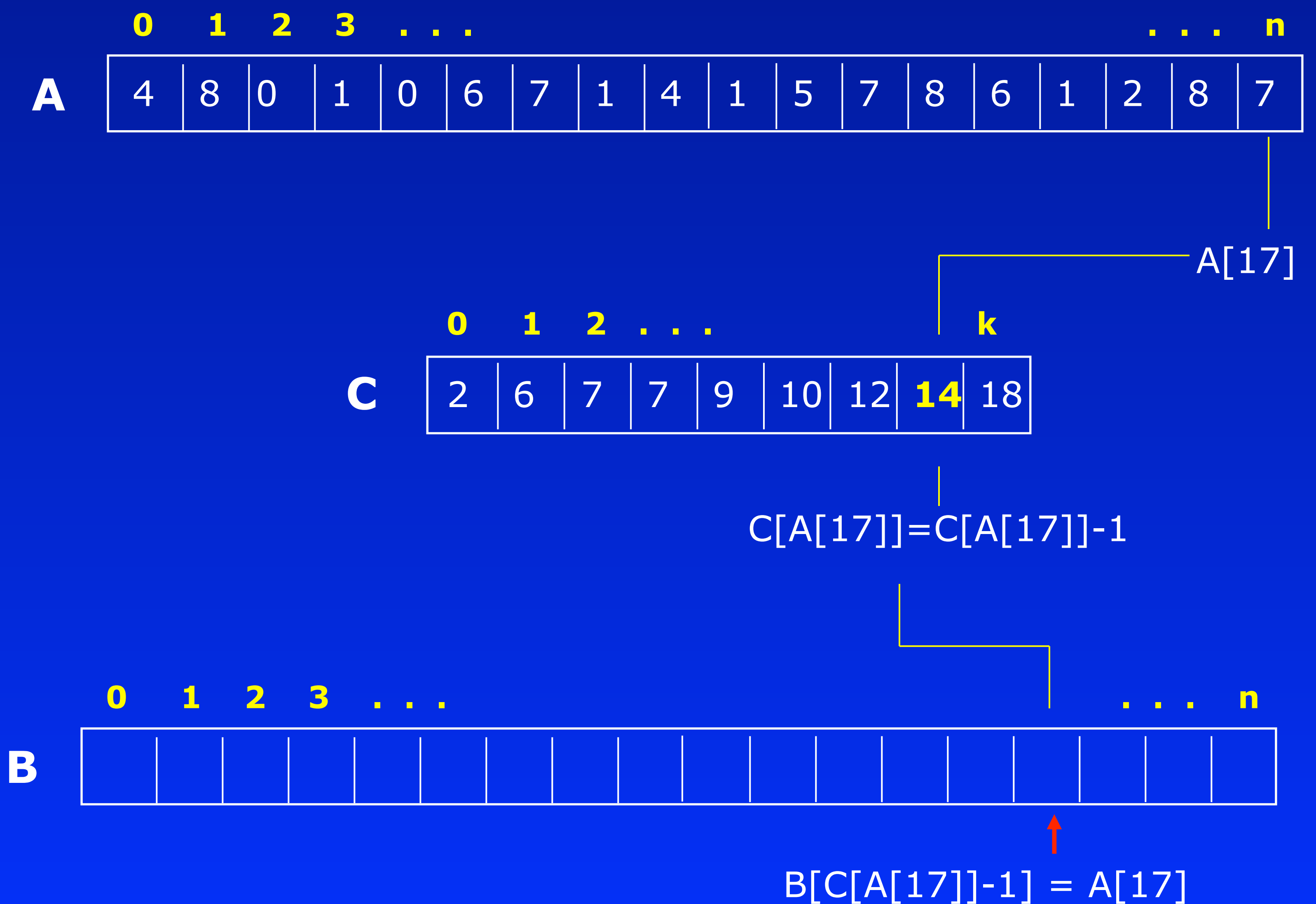
C	2	6	7	7	9	10	12	**14**	18

C[A[17]]=C[A[17]]-1

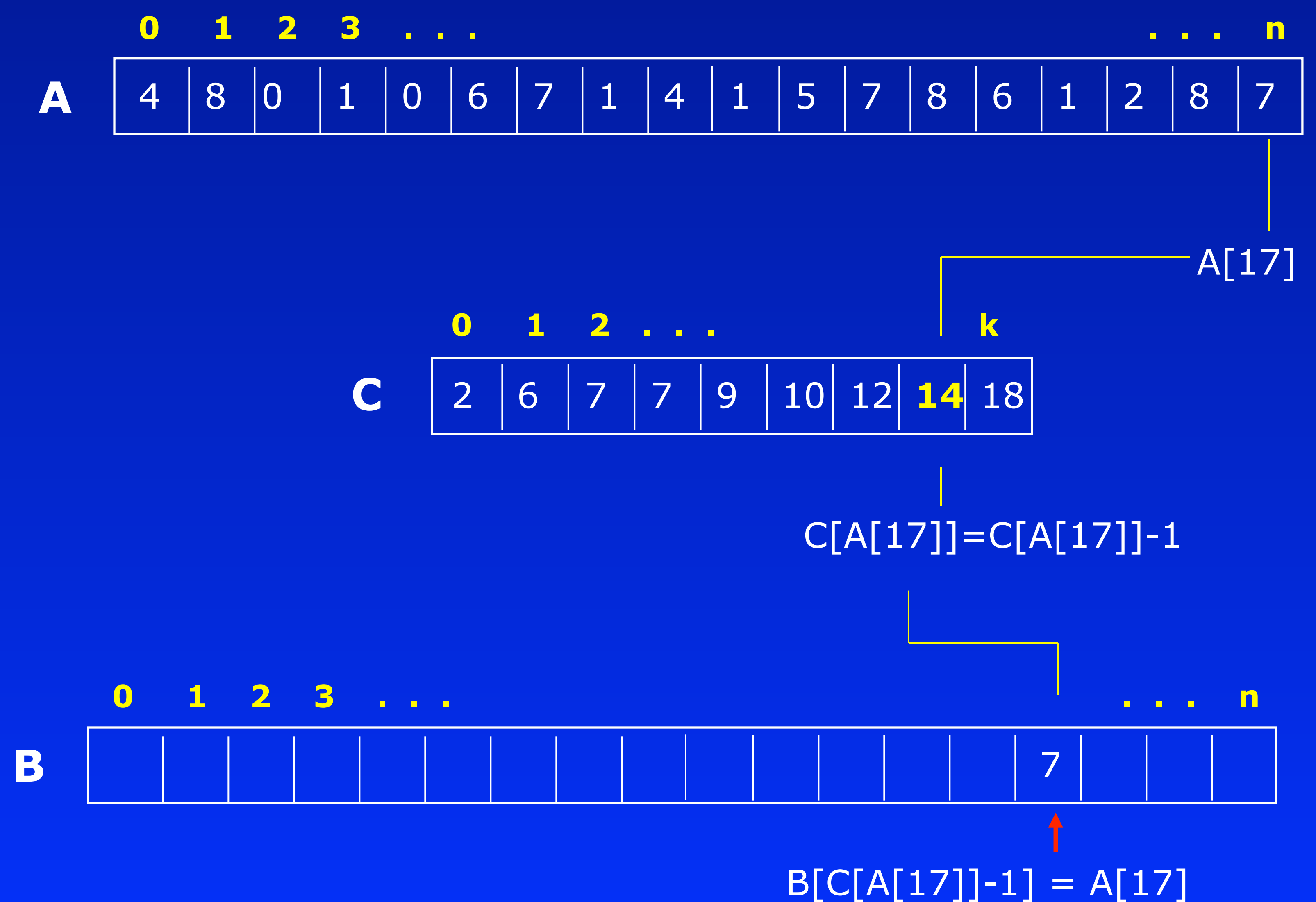
0 1 2 3 . . . . . . n

B																		

Counting sort



Counting sort



Counting sort

0 1 2 3 . . . . . . n

A

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

A[16]

0 1 2 . . . k

C

2	6	7	7	9	10	12	**14**	18

0 1 2 3 . . . . . . n

B

														7			

Counting sort

0 1 2 3 . . . . . . n

A																		
	4	8	0	1	0	6	7	1	4	1	5	7	8	6	1	2	8	7

A[16]

0 1 2 . . . k

C									
	2	6	7	7	9	10	12	**14**	17

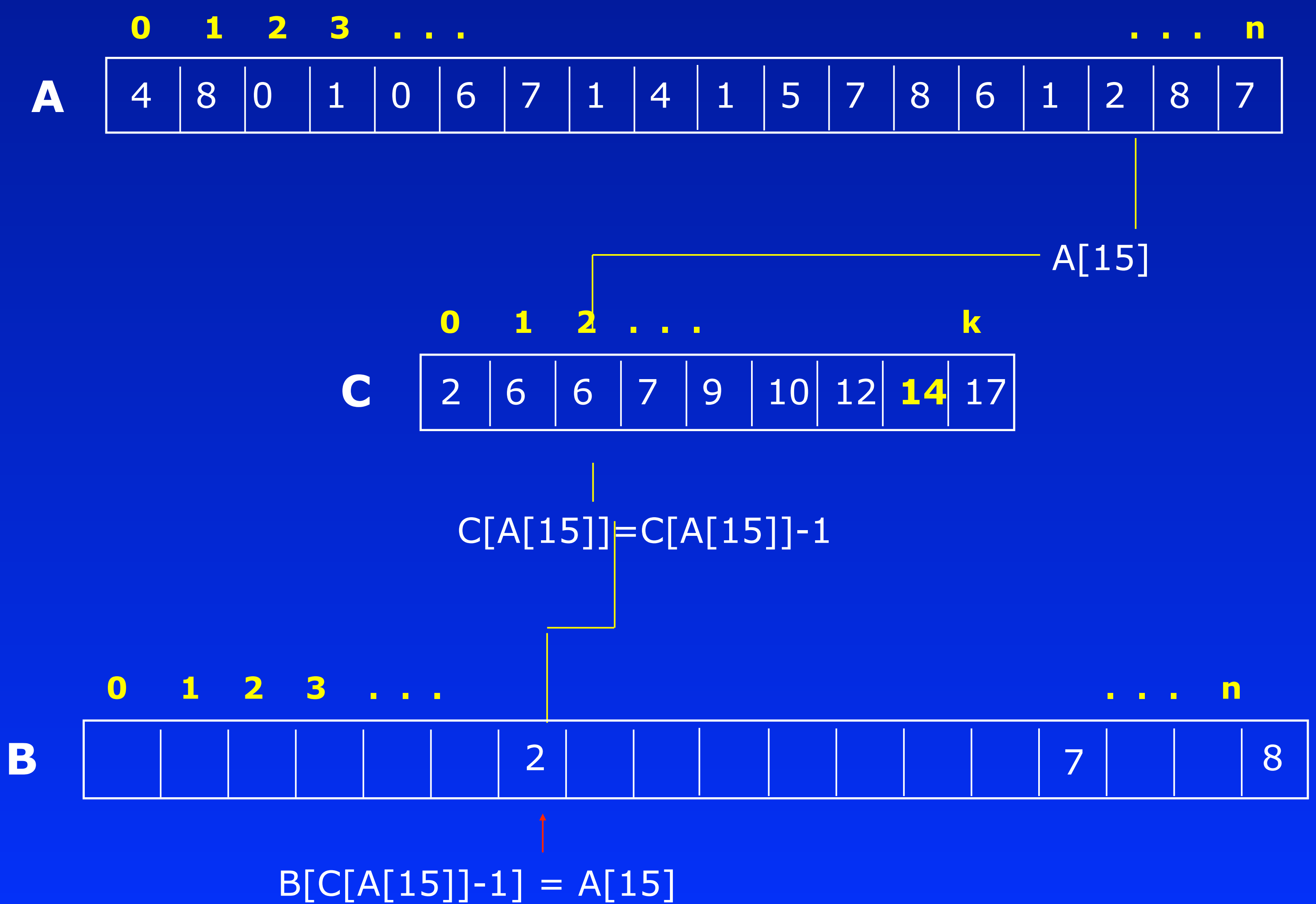
C[A[16]]=C[A[16]]-1

0 1 2 3 . . . . . . n

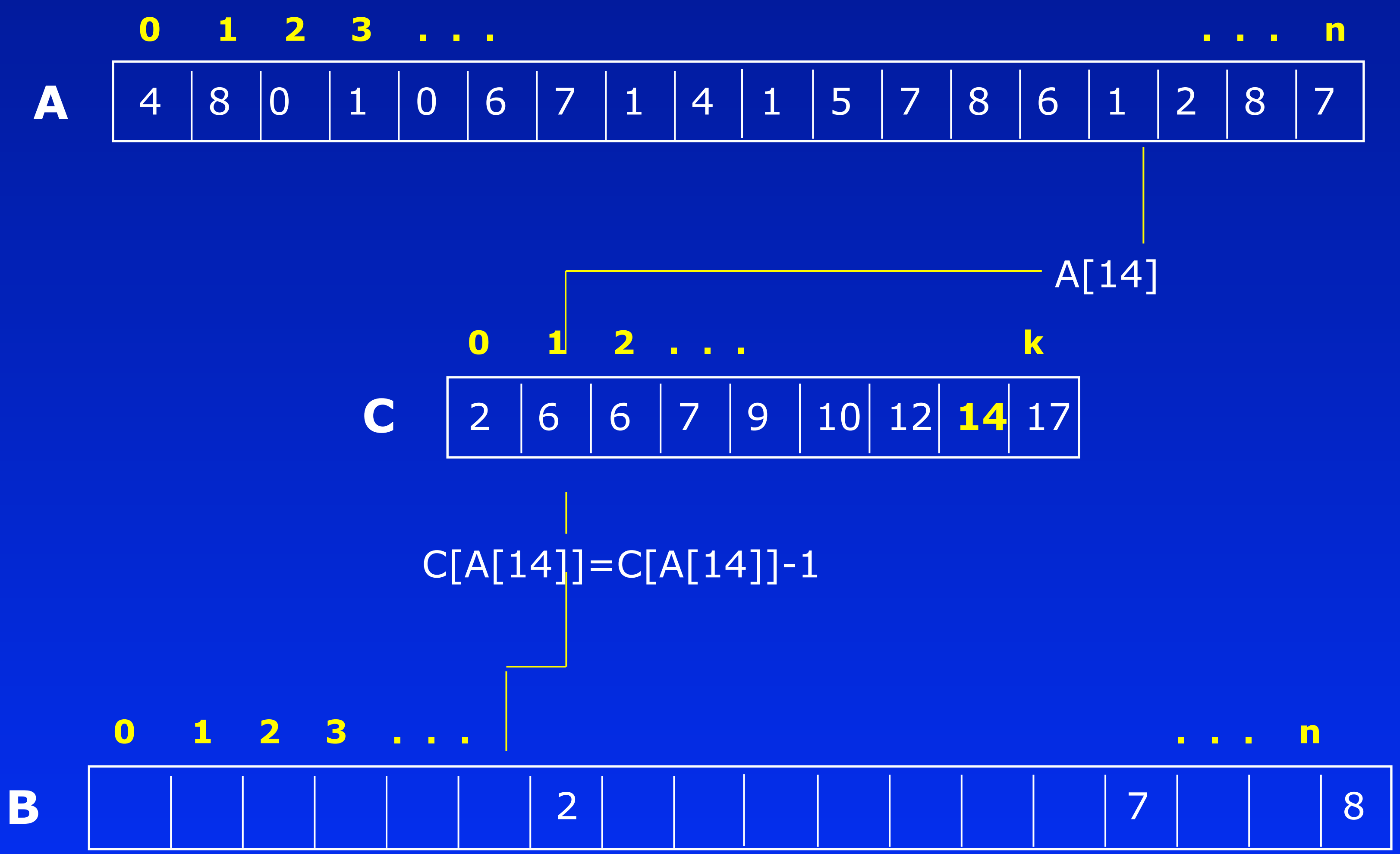
B																		
														7				8

B[C[A[16]]-1] = A[16]

Counting sort

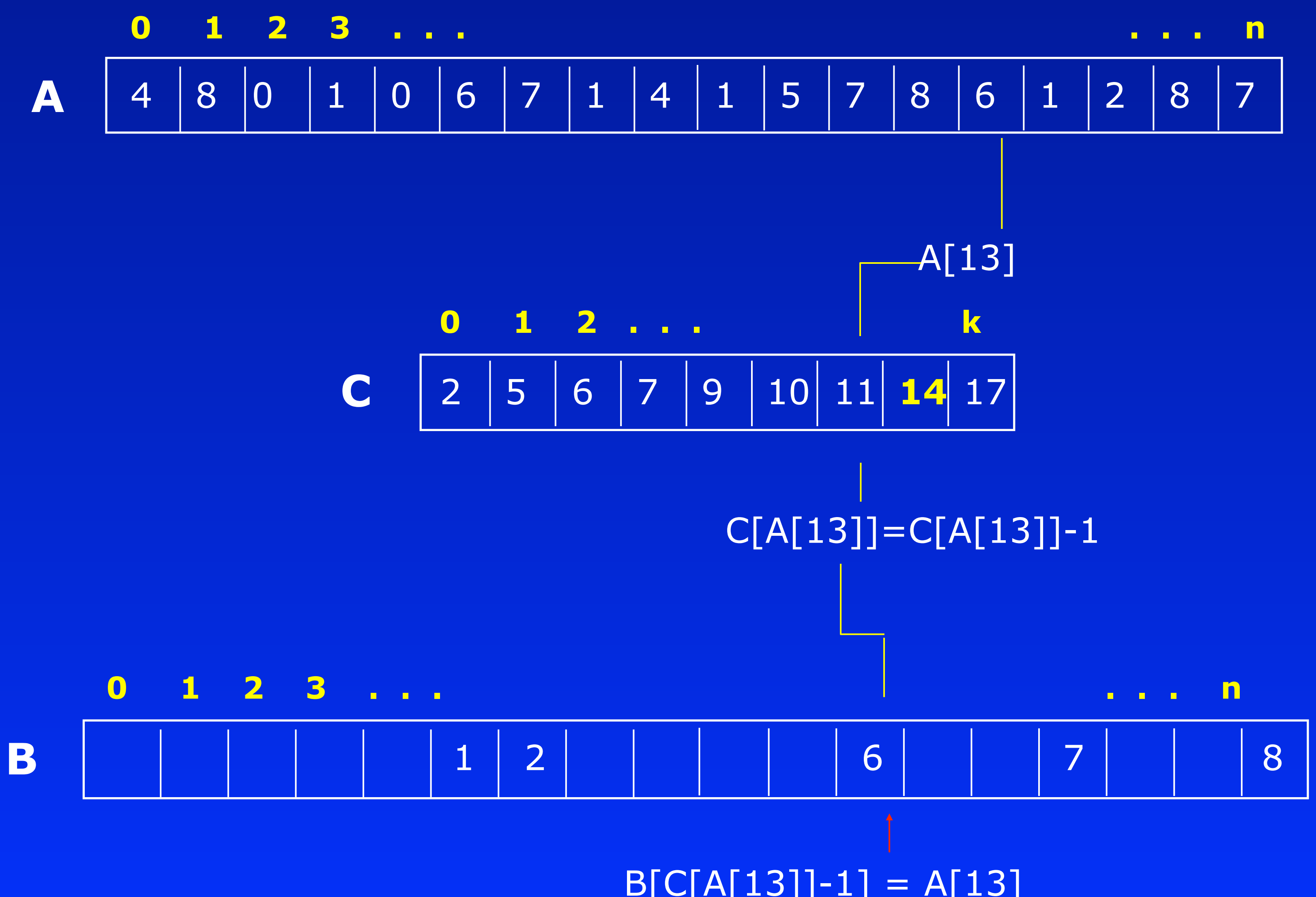


Counting sort



Counting sort

Counting sort



Counting sort

Counting sort

```
def counting_sort(A, k):
        size = len(A)
        B = [0 for i in range(0, size)]
        C = [0 for i in range(0, k+1)]

        for j in range(0, size):
                C[A[j]] += 1
        for i in range(1, k+1):
                C[i] += C[i-1]
        for j in range(size-1, -1, -1):
                C[A[j]] -= 1
                B[C[A[j]]] = A[j]
        return B
```

Eine wichtige Eigenschaft des **Counting-Sort**-Algorithmus ist, dass er stabil ist.

Counting sort

Im Array sind nur die Adressen (Referenzen)

1
2
3
4
5
6
7
8
9
10
11
12
13

Datenbank

Adresse	Nachname	Vorname	Semester
1	Hagenauer	Daniel Bemd	2
2	Herr	Dennis	3
3	Jäger	Oliver	1
4	Knoll	Eugen	3
5	Le	Huy Cuong	5
6	Lenz	Markus	7
7	Meister	Mark	9
8	Niesner	Frank	1
9	Rudewig	Günter	2
10	Savas	Garnze	1
11	Singe	Alexander	1
12	Wäldchen	Rene Sebastian	5
13	Wörner	Marcel	3

Counting sort

Im Array sind
nur die
Adressen

3
8
10
11
1
9
2
4
13
5
12
6
7

Datenbank

Adresse	Nachname	Vorname	Semester
1	Hagenauer	Daniel Bemd	2
2	Herr	Dennis	3
3	Jäger	Oliver	1
4	Knoll	Eugen	3
5	Le	Huy Cuong	5
6	Lenz	Markus	7
7	Meister	Mark	9
8	Niesner	Frank	1
9	Rudewig	Günter	2
10	Savas	Garnze	1
11	Singe	Alexander	1
12	Wäldchen	Rene Sebastian	5
13	Wörner	Marcel	3

Sortiert nach Semester

Counting sort

Im Array sind nur die Adressen

3
8
10
11
1
9
2
4
13
5
12
6
7

Sortiert nach Semester

Datenbank

Adresse	Nachname	Vorname	Semester
1	Hagenauer	Daniel Bemd	2
2	Herr	Dennis	3
3	Jäger	Oliver	1
4	Knoll	Eugen	3
5	Le	Huy Cuong	5
6	Lenz	Markus	7
7	Meister	Mark	9
8	Niesner	Frank	1
9	Rudewig	Günter	2
10	Savas	Garnze	1
11	Singe	Alexander	1
12	Wäldchen	Rene Sebastian	5
13	Wörner	Marcel	3

Counting sort

Im Array sind
nur die
Adressen

3
8
10
11
1
9
2
4
13
5
12
6
7

Datenbank

Adresse	Nachname	Vorname	Semester
1	Hagenauer	Daniel Bemd	2
2	Herr	Dennis	3
3	Jäger	Oliver	1
4	Knoll	Eugen	3
5	Le	Huy Cuong	5
6	Lenz	Markus	7
7	Meister	Mark	9
8	Niesner	Frank	1
9	Rudewig	Günter	2
10	Savas	Garnze	1
11	Singe	Alexander	1
12	Wäldchen	Rene Sebastian	5
13	Wörner	Marcel	3

Sortiert nach Semester

Counting sort

Im Array sind
nur die
Adressen

3
8
10
11
1
9
2
4
13
5
12
6
7

Sortiert nach Semester

Datenbank

Adresse	Nachname	Vorname	Semester
1	Hagenauer	Daniel Bemd	2
2	Herr	Dennis	3
3	Jäger	Oliver	1
4	Knoll	Eugen	3
5	Le	Huy Cuong	5
6	Lenz	Markus	7
7	Meister	Mark	9
8	Niesner	Frank	1
9	Rudewig	Günter	2
10	Savas	Garnze	1
11	Singe	Alexander	1
12	Wäldchen	Rene Sebastian	5
13	Wörner	Marcel	3

Radix sort

1887 Entwickelt für das Sortieren von Lochkarten

```
radixsort ( A, d) {
    for  i in range(d):
        stablesort(A, i)
```

Countingsort

$T(n) = O(n)$

- Zahlen werden ziffernweise sortiert
- die niedrigsten Stellenwerte zuerst
- alles nur mit einem stabilen Algorithmus (essentiell)
- auch gut für das Sortieren von zusammengesetzte Datenstrukturen

Beispiel: Sortieren nach Datum

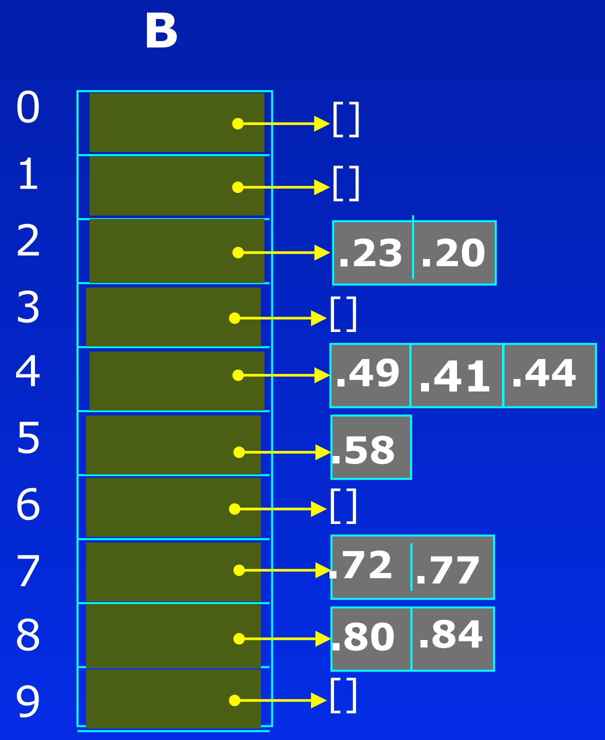
Bucket-Sort

- Die zu sortierenden Daten müssen gleich verteilt über den Wertebereich [0,1) sein.
- **Not-In-Place**

  zusätzlicher Speicherplatz (**O(n)**) wird benötigt
- linearer Aufwand **O(n)**
- Grundidee ist den Wertebereich [0,1) in **m** kleinere Wertebereiche zu teilen und *Buckets* dafür zu definieren.
- Die Zahlen werden in den dazugehörigen *Buckets* verteilt und innerhalb diesen sortiert.
- Zum Schluss werden die Zahlen der Reihe nach aus den *Buckets* ausgegeben.
- Eine Hilfsarray von verketteten Listen wird für die *Buckets* verwendet.

Bucket-Sort

A
.58
.23
.49
.72
.77
.41
.20
.44
.80
.84



Bucket-Sort

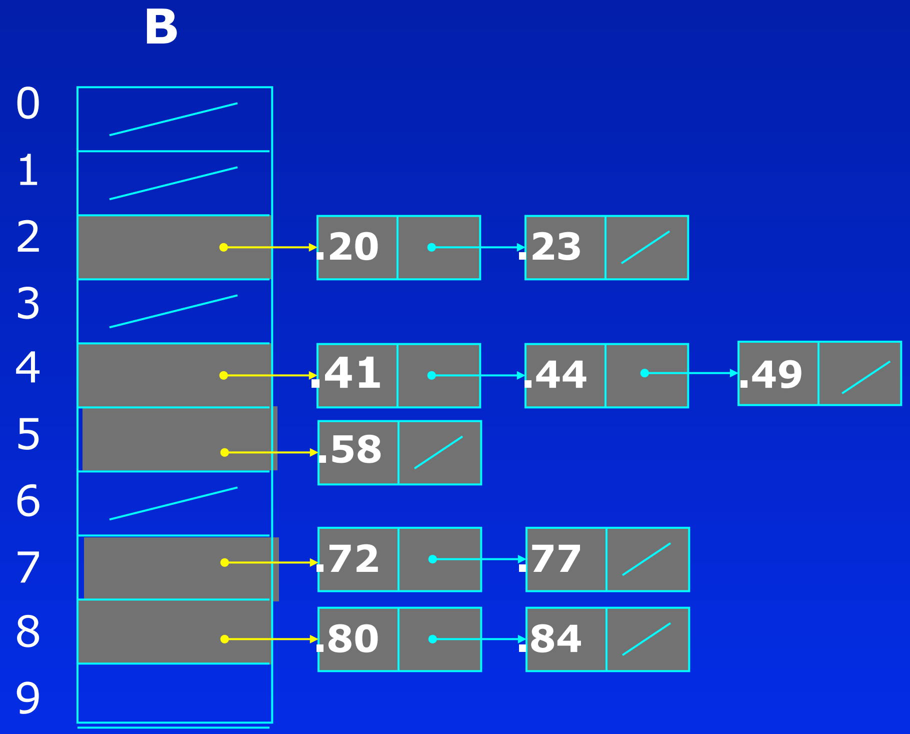
```
def bucketsort ( A ):
    n = len(A)
    B = [ []  for  i  in  range(n) ]
    for i in range(n):
            B[math.floor((A[i])*10)].append(A[i])
    for i in range(0,n):
            insertsort(B[i])
    R = []
    for i in range(n):
            R = R+B[i]
    return R
```

Obama in Google

Bucket-Sort

A

.58
.23
.49
.72
.77
.41
.20
.44
.80
.84



Sortieralgorithmen

- Vergleichs-Algorithmen
  - quadratisch
    - Bubble-Sort **O**(n^2)
    - Insert-Sort **O**(n^2)
    - Selection-Sort **O**(n^2)
  - n (log (n))
    - Quicksort **O**$(n \cdot \log(n))$ Durchschnittsfall
    - Mergesort **O**$(n \cdot \log(n))$
    - Heap-Sort **O**$(n \cdot \log(n))$
- Ohne Vergleiche
  - linear
    - Counting-Sort **O**(n)
    - Radix-Sort **O**(n)
    - Bucket-Sort **O**(n)