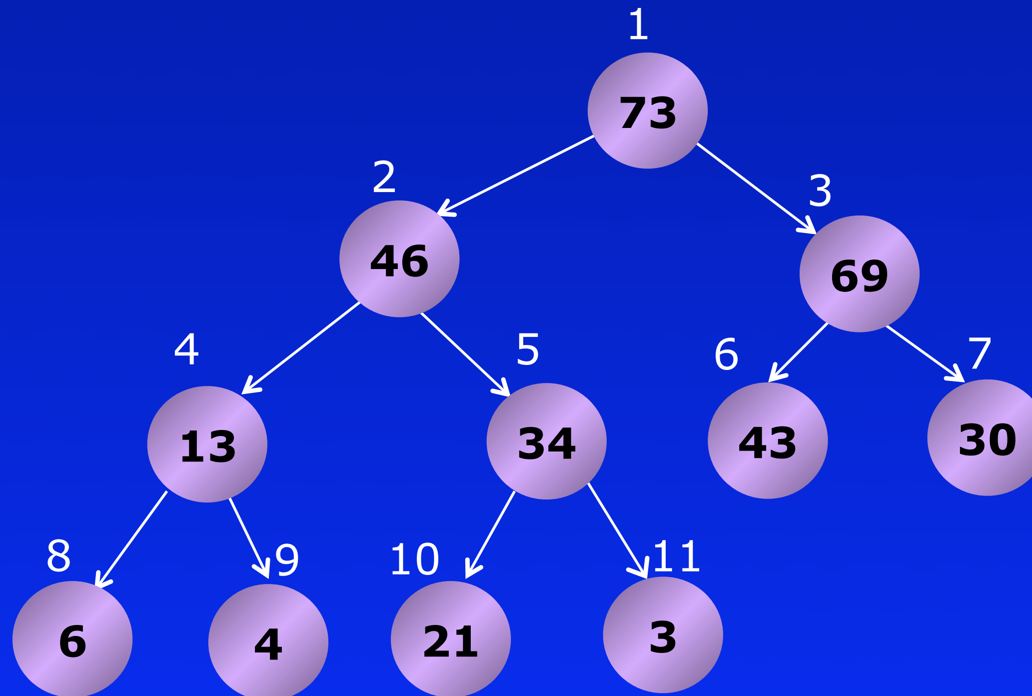
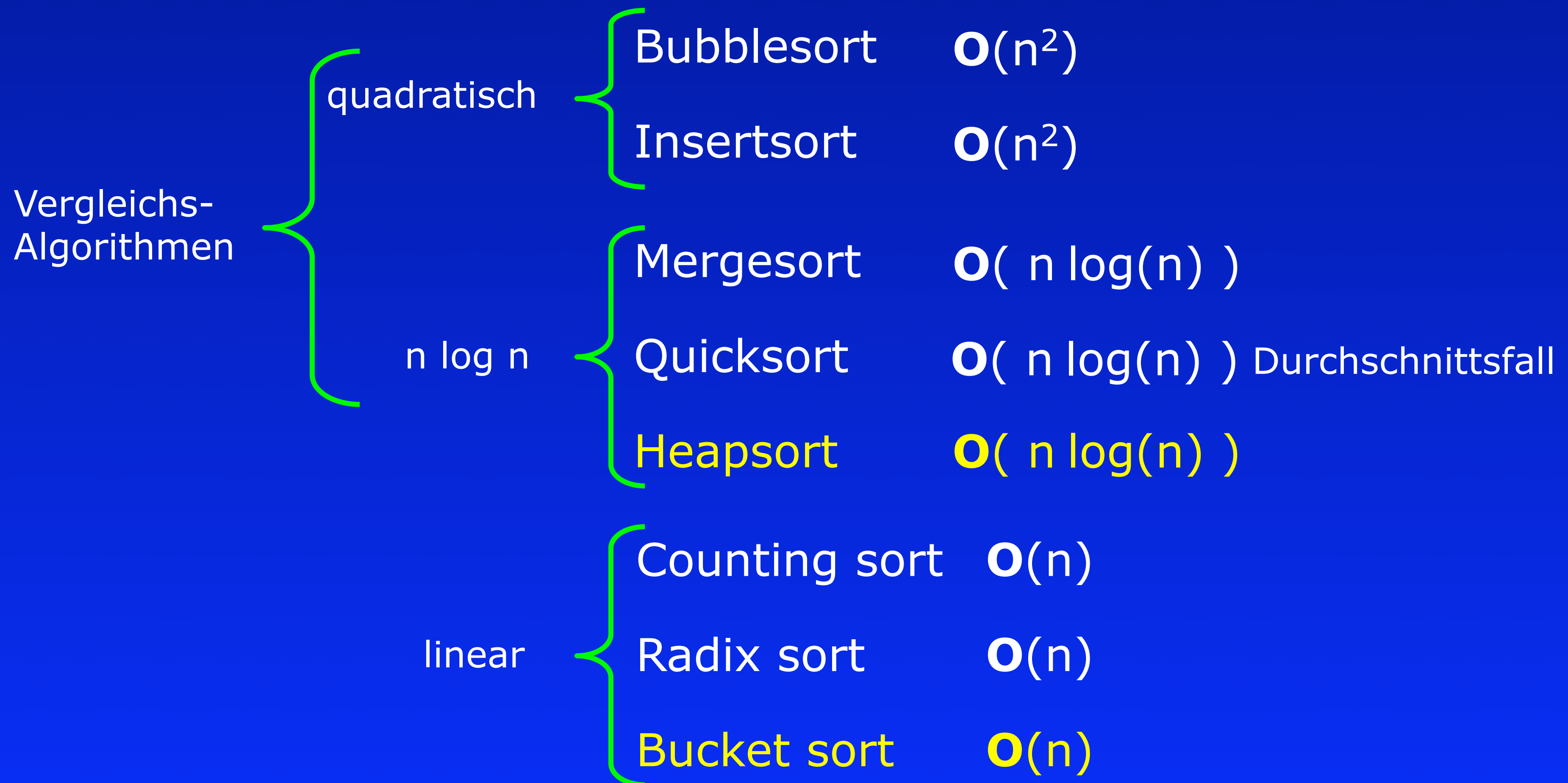
Algorithmen und Programmieren II

Sortieralgorithmen (Teil III)

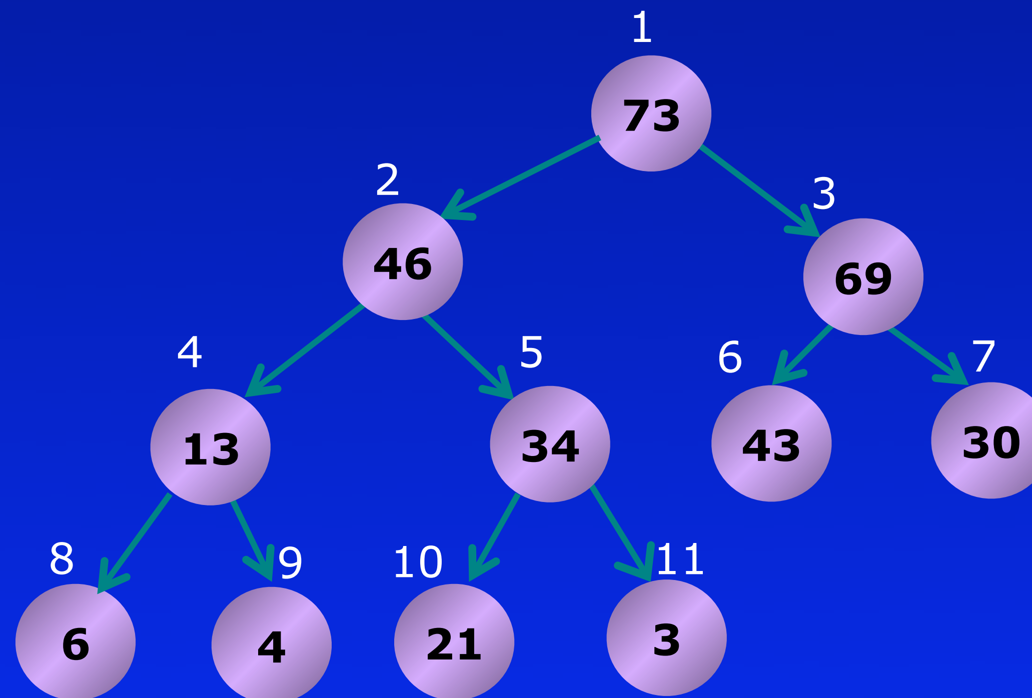


Prof. Dr. Margarita Esponda

Sortieralgorithmen



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.

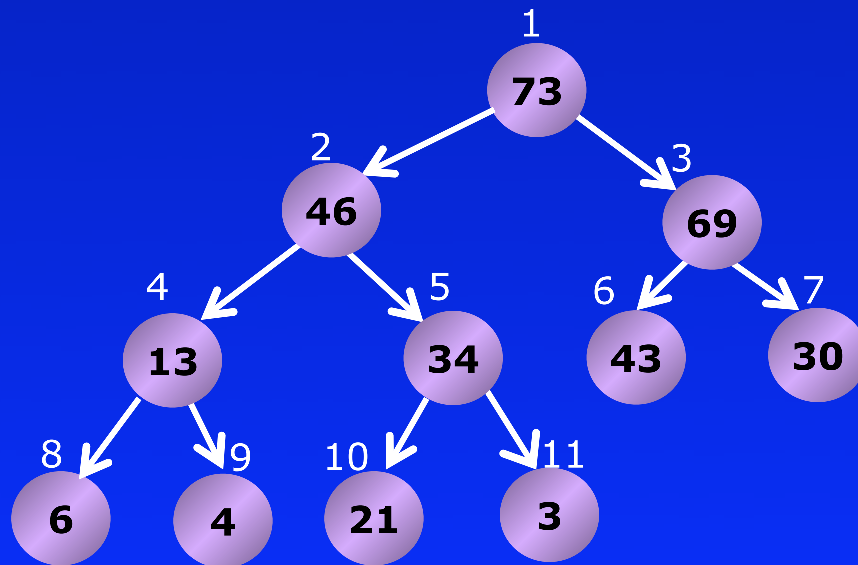
Heap

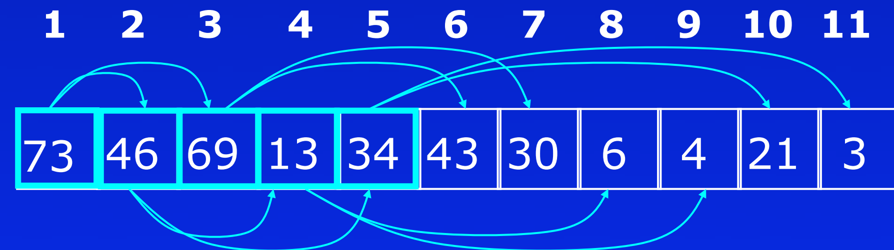
Heap

oder

Max-Heap

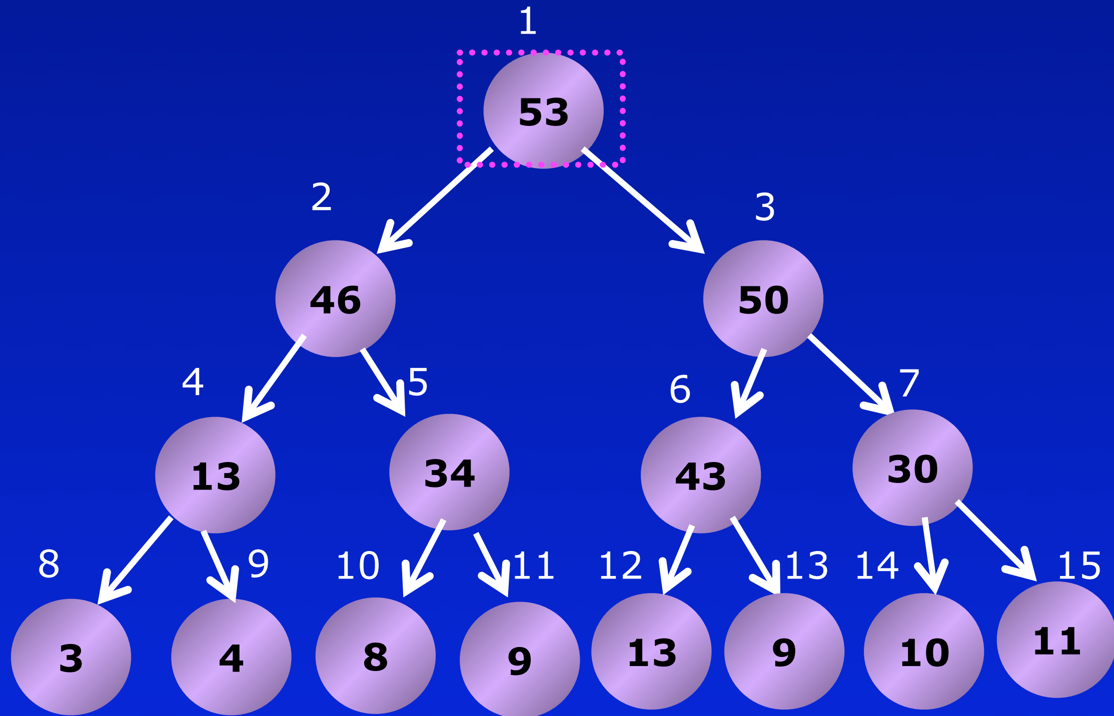
ist ein **Binärbaum**, der **in einem Feld gespeichert** wird und die Eigenschaft hat, dass das Element, das in jedem Knoten des Baums gespeichert ist, größer oder gleich ist als alle Elemente seines linken und rechten Unterbaums.

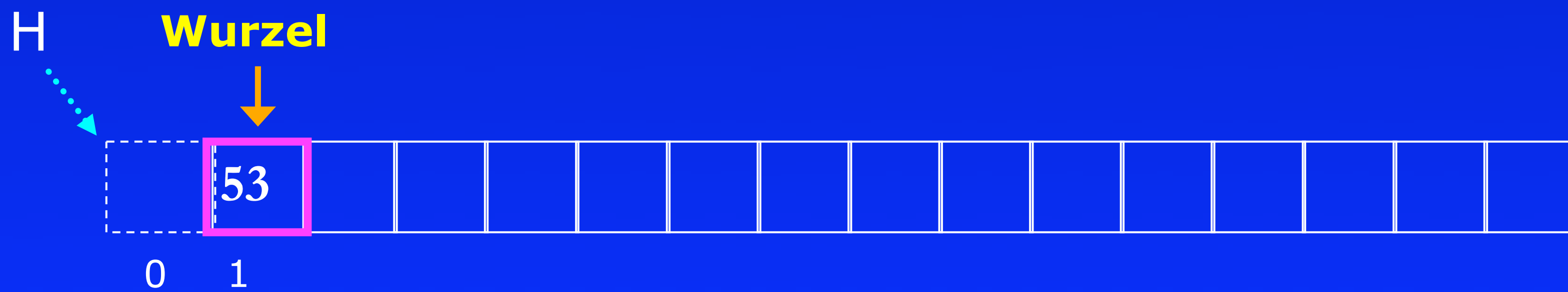




M. Esponda

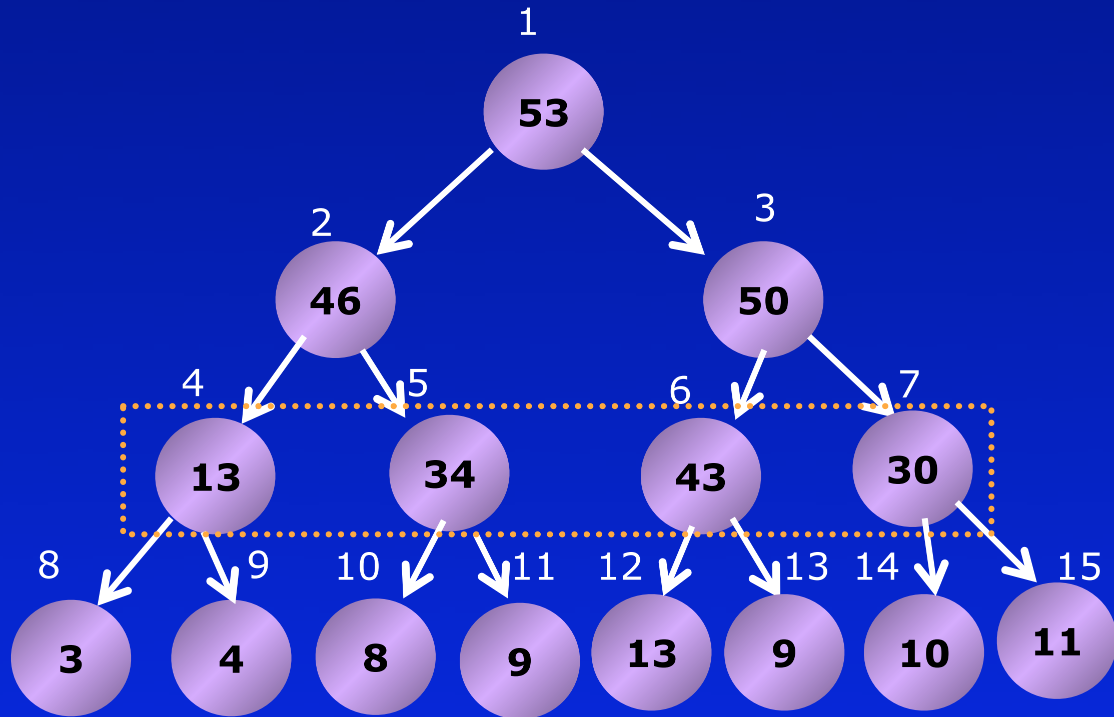
Heap

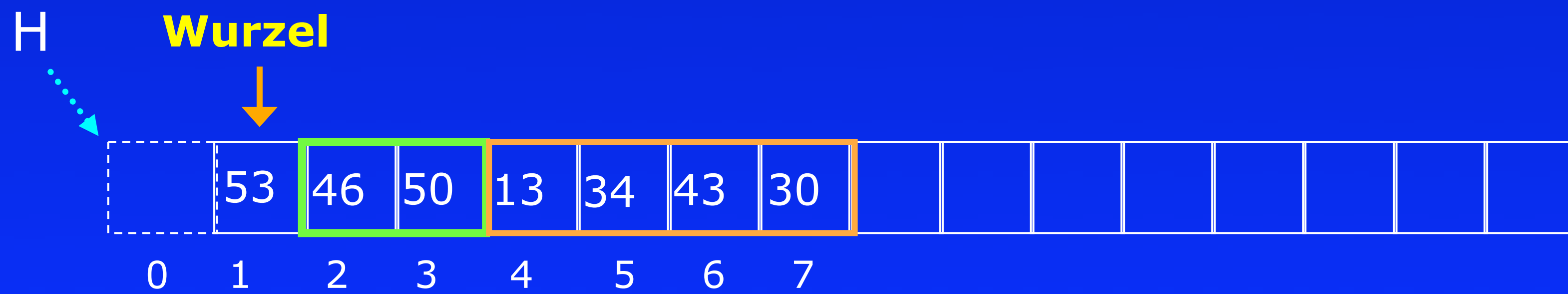




M. Esponda

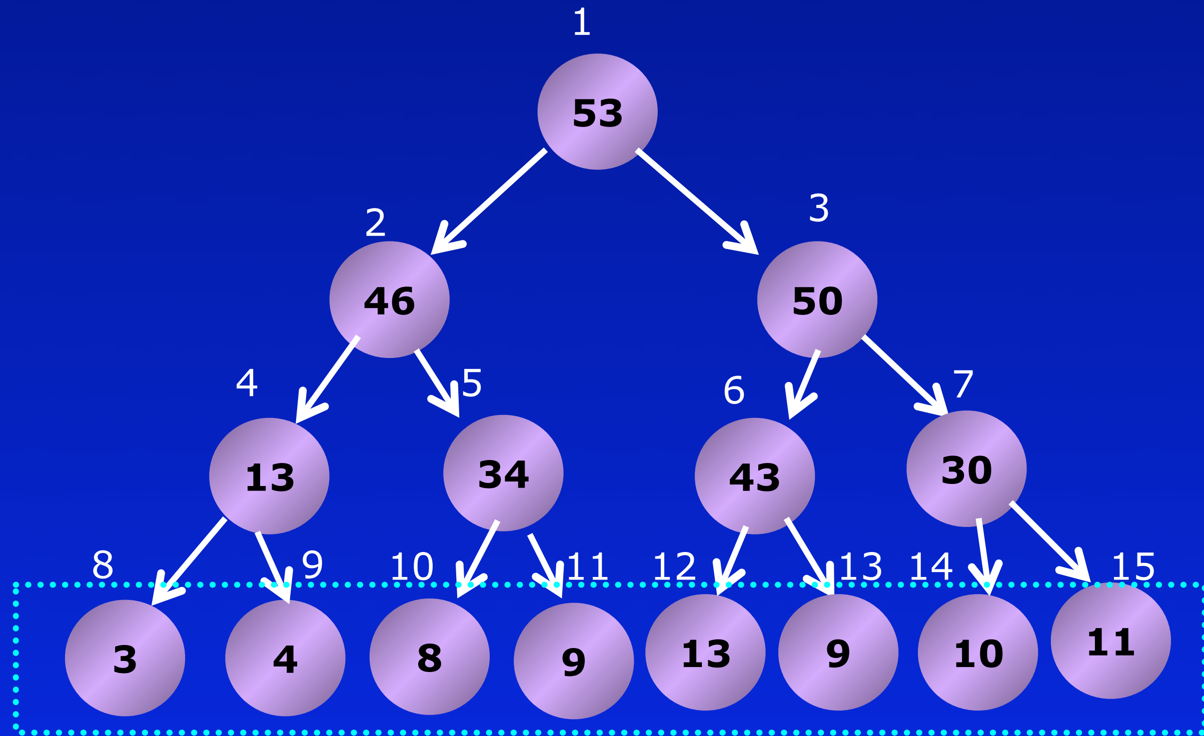
Heap

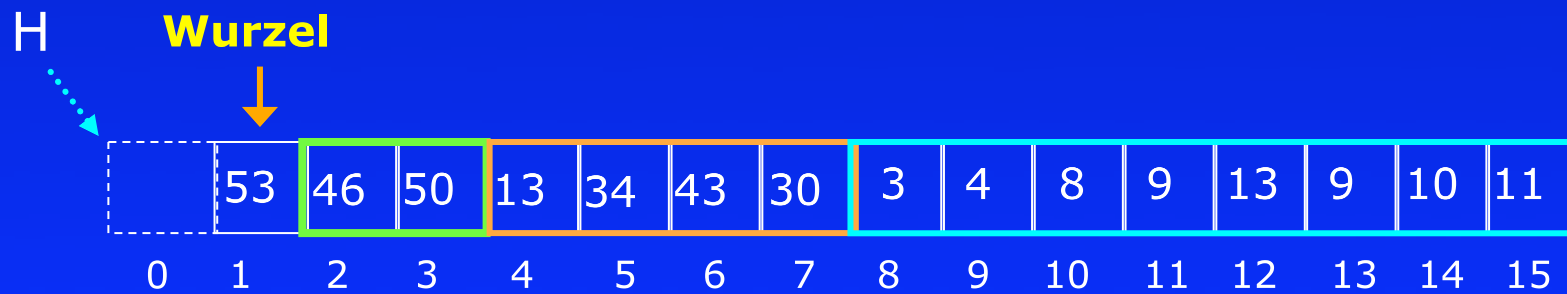




M. Esponda

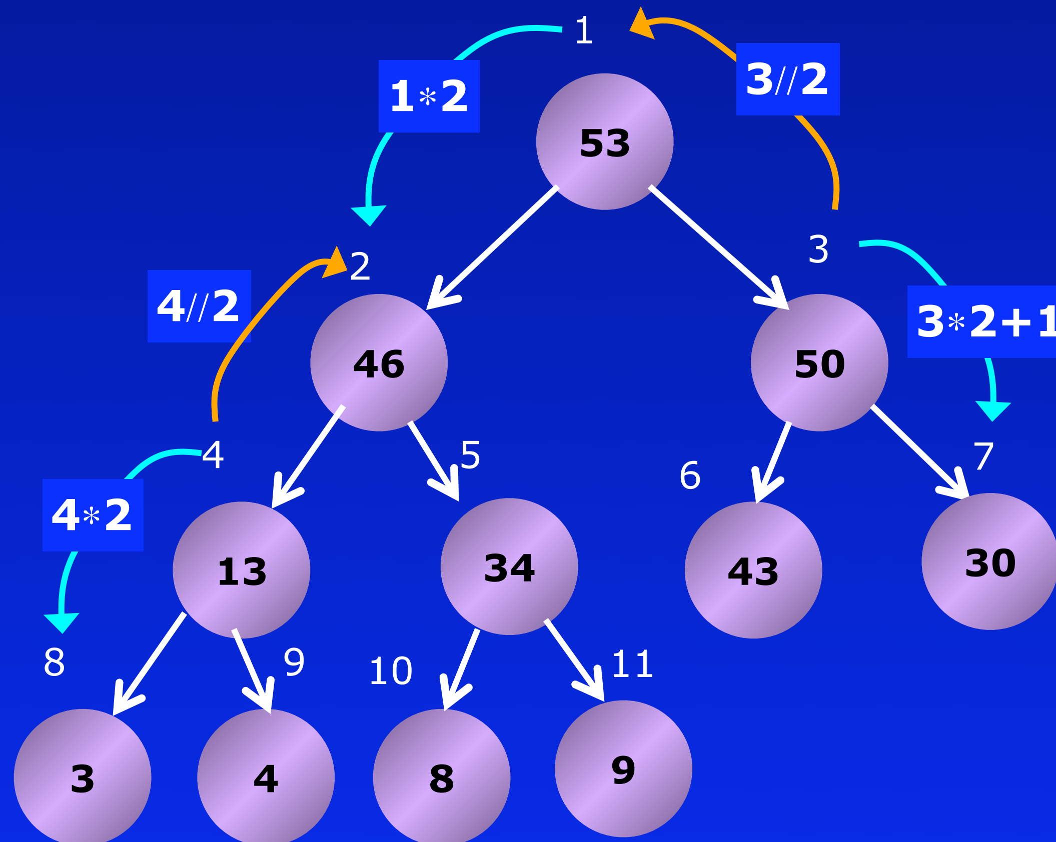
Heap





M. Esponda

Navigieren im Heap

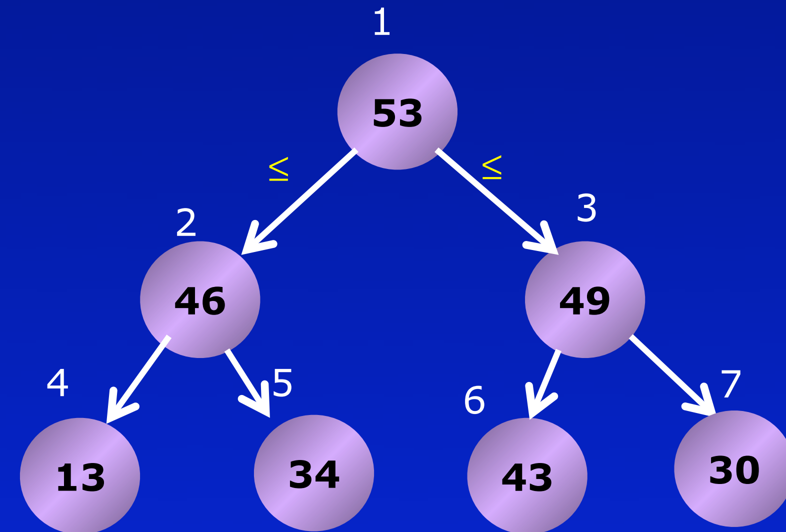


Das linke Kind einer beliebigen Position **i** des Heaps ist gleich **i*2**
Das rechte Kind einer beliebigen Position **i** ist gleich **i*2+1**
Das Elternteil eines beliebigen Kindes **i** ist gleich **i/2** (ganzzahlige Division)

M. Esponda

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:

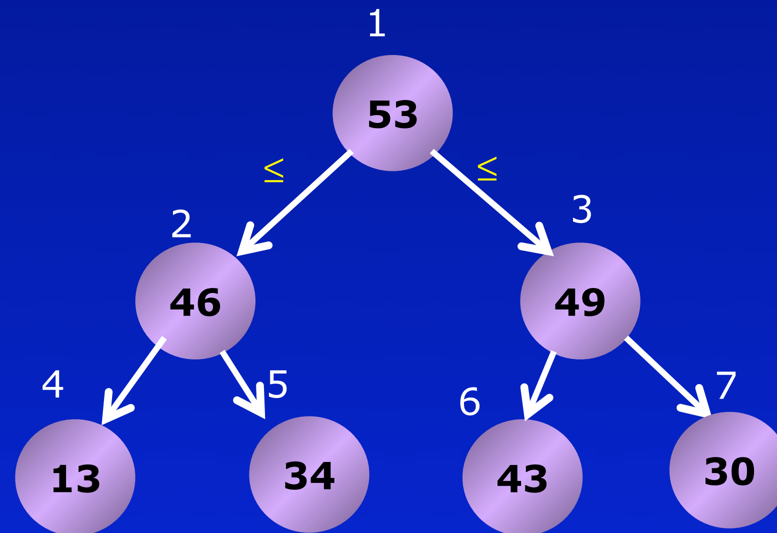
```
def parent(i):
        return i//2
```

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

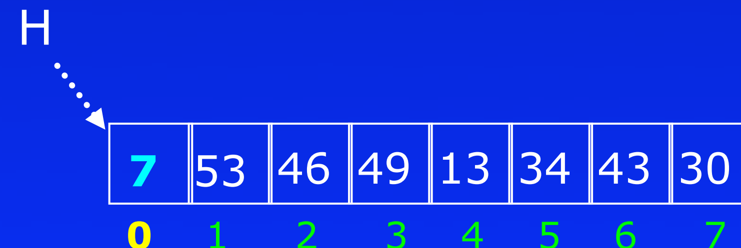
```
def right(i):
        return i*2+1
```

Heap-Hilfsfunktionen

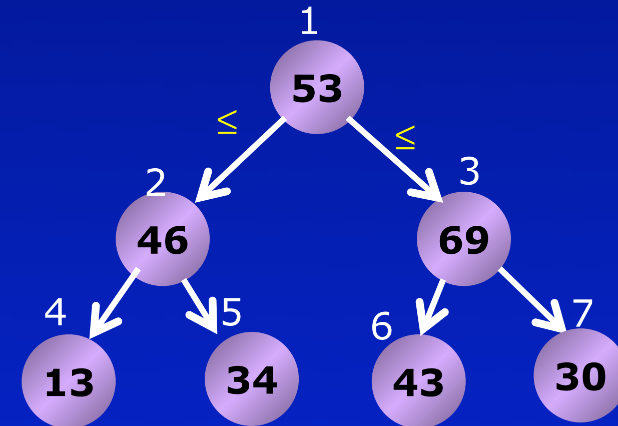
Für die Berechnung der Funktionen **parent**, **left** und **right** ist es günstiger, den **Heap** ab der Position **1** des Feldes zu speichern.



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.

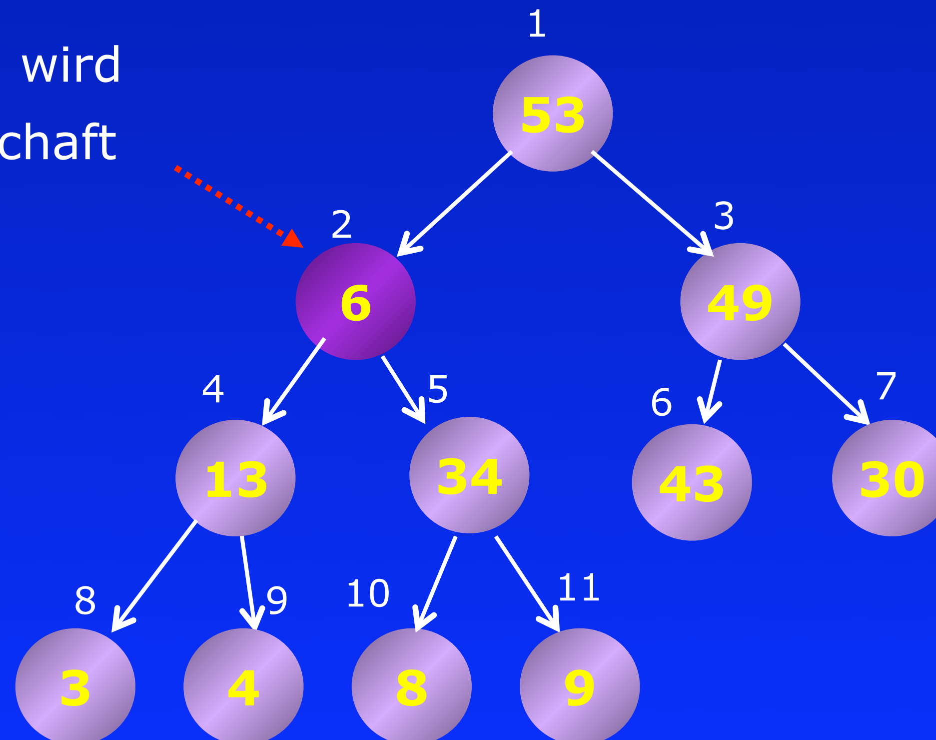
```
def  heap_size(H):
         return H[0]
```

```
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.

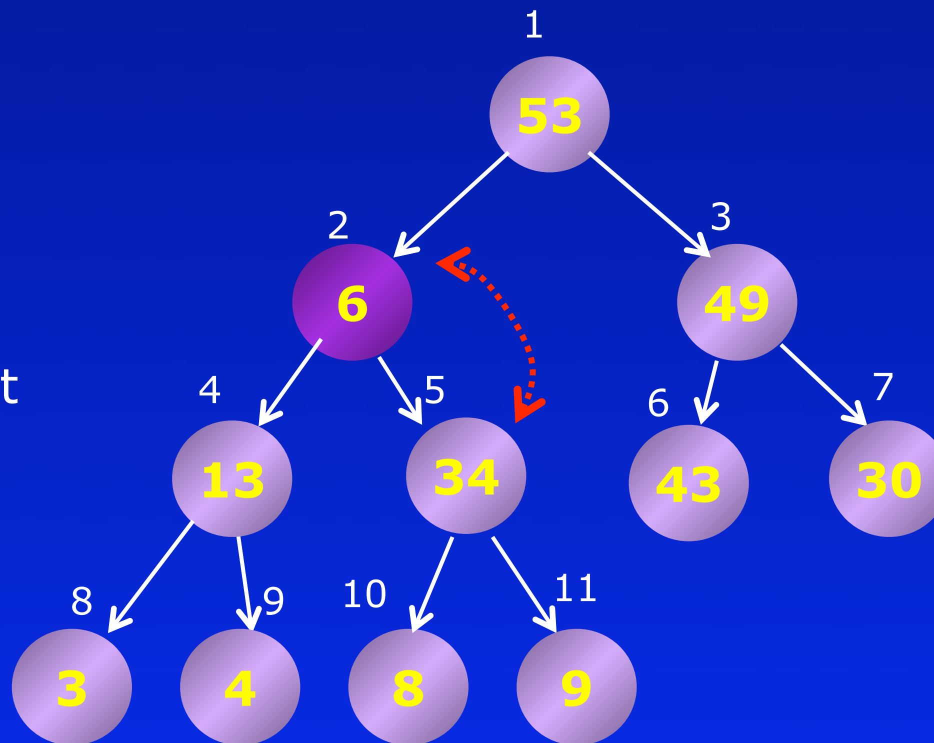
In der Position 2 wird die Heap-Eigenschaft verletzt.



M. Esponda

max_heapify-Funktion

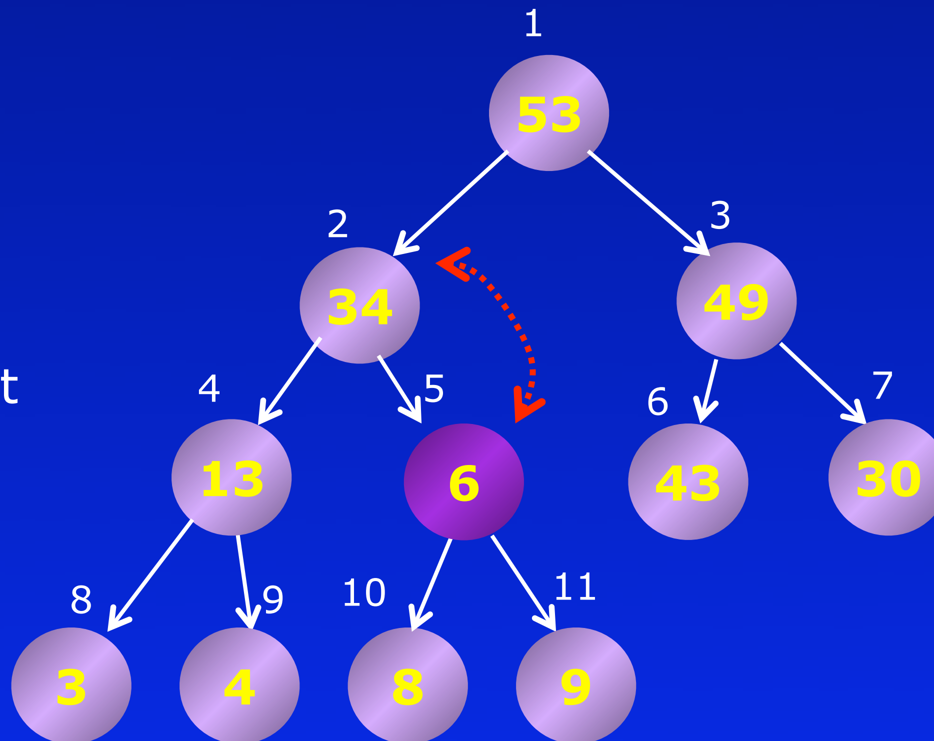
Das größere von beiden Kindern wird mit dem Element in Position **2** vertauscht, wenn es gleichzeitig größer als das Element in Position **2** ist.



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

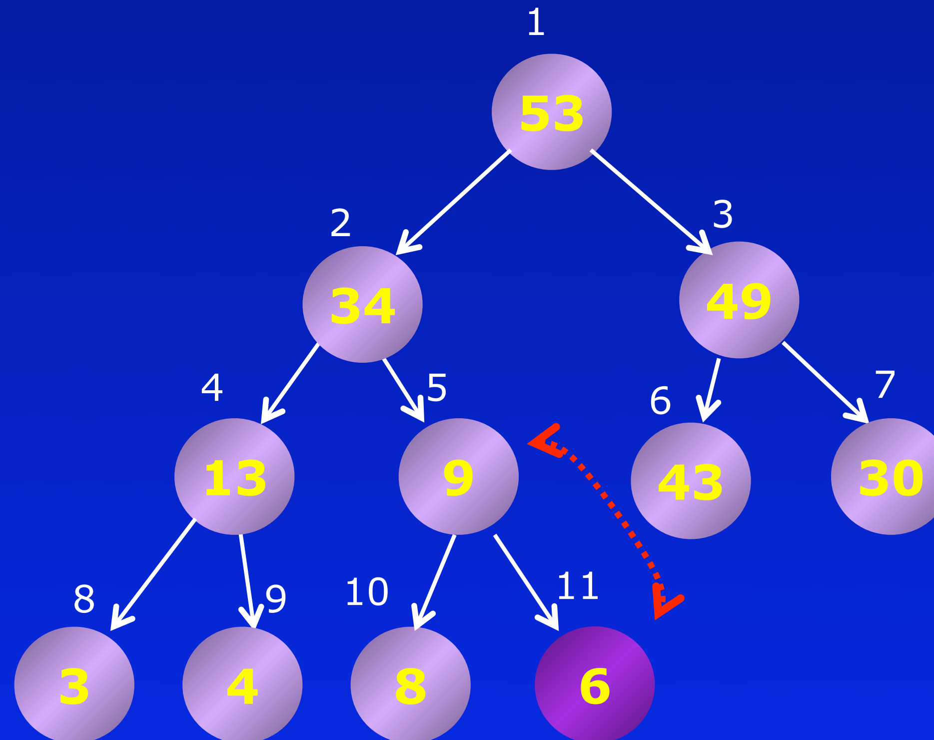
Das größere von beiden Kindern wird mit dem Element in Position **2** vertauscht, wenn es gleichzeitig größer als das Element in Position **2** ist.



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.

M. Esponda

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.

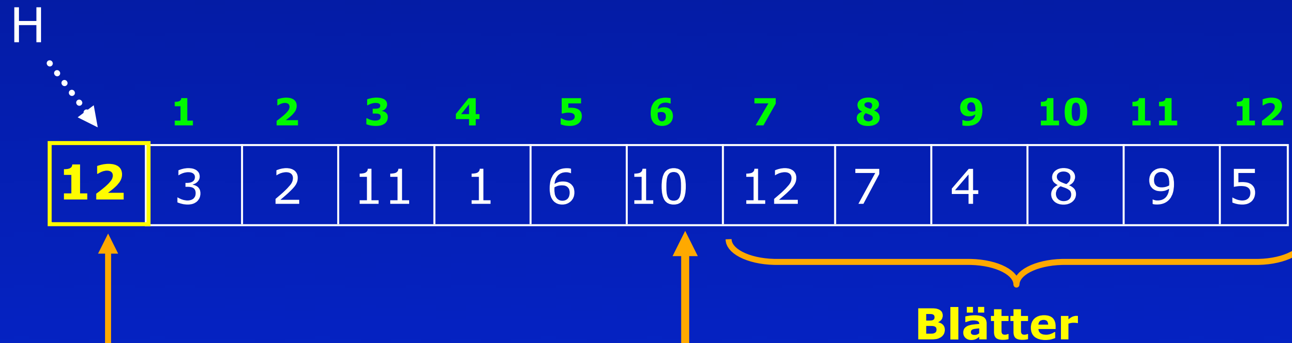
max_heapify-Funktion

```
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]:
                bigest = left_t
        else:
                bigest = pos

        if right_t<=heap_size(H)  and  H[right_t]>H[bigest]:
                bigest = right_t

        if bigest != pos:
                H[pos], H[bigest] = H[bigest], H[pos]
                max_heapify( H, bigest )
```

build_heapify-Funktion

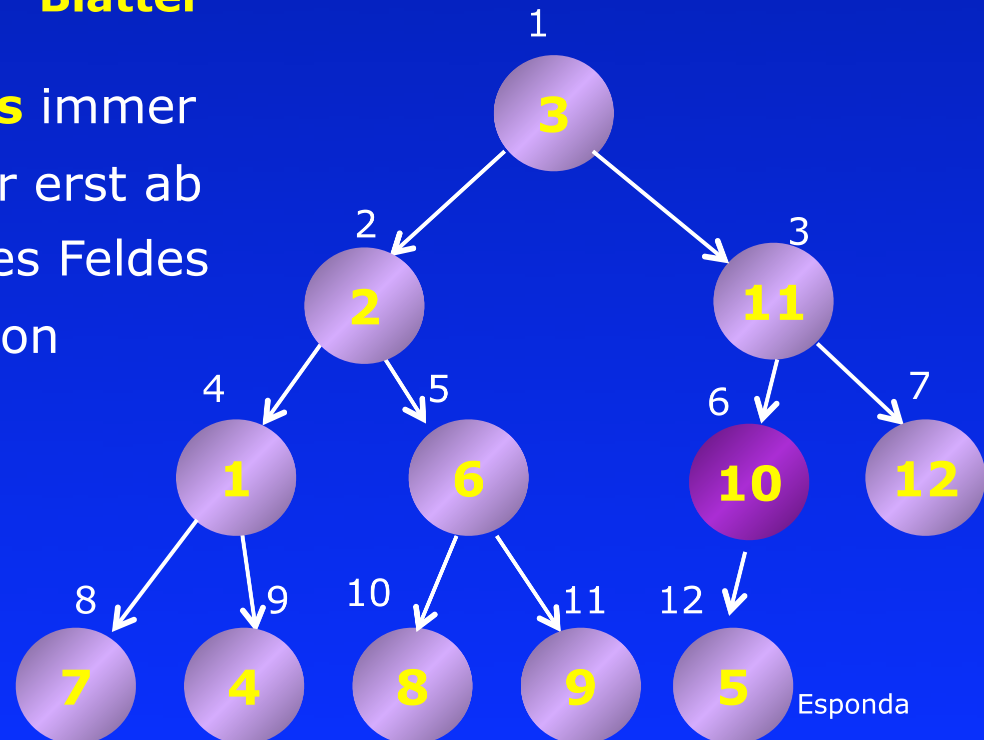
Wie können wir aus einer beliebigen Zahlenreihe einen Heap konstruieren?

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

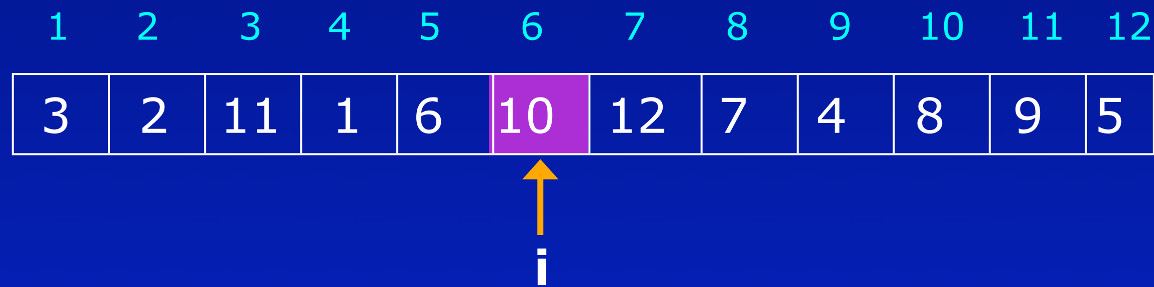


Weil die Hälfte des **Heaps** immer Blätter sind, brauchen wir erst ab Ende des ersten Hälfte des Feldes die **max_heapify**-Funktion aufzurufen.

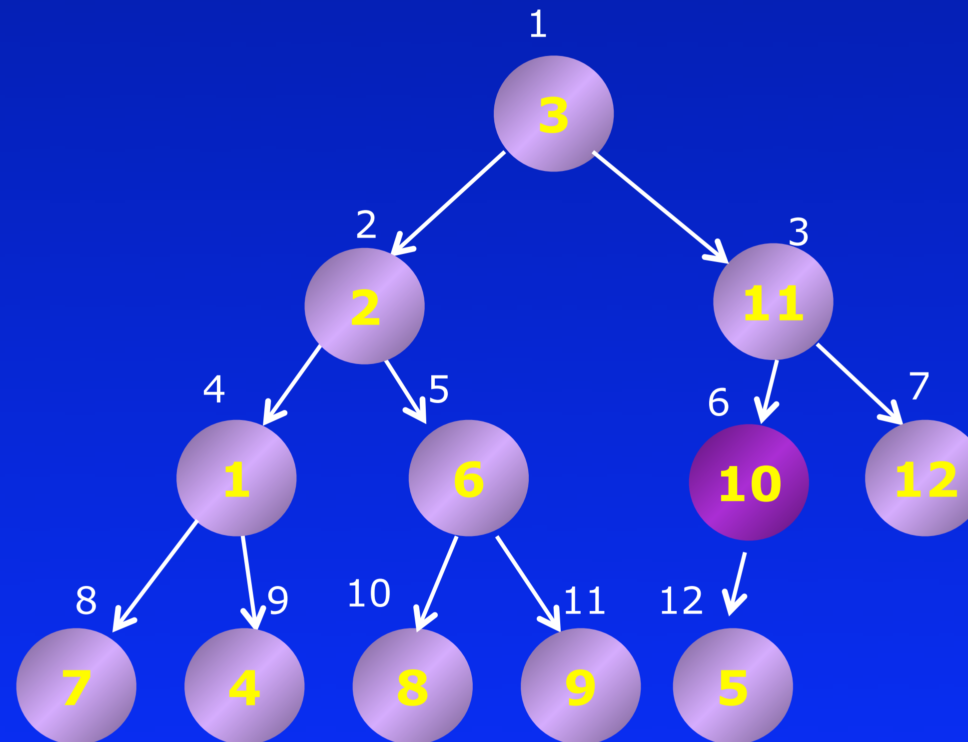
In der Position **0** des **H**-Feldes speichern wir die Größe unseres **Heaps**.



build_heapify-Funktion



max_heapify (H, 6)



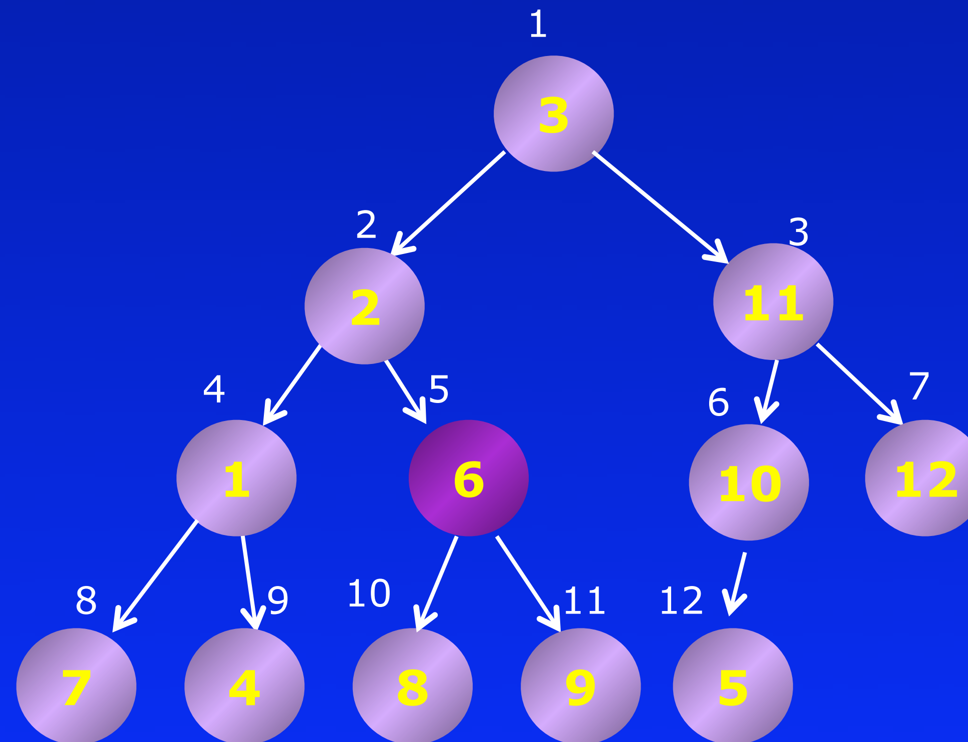
M. Esponda

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)

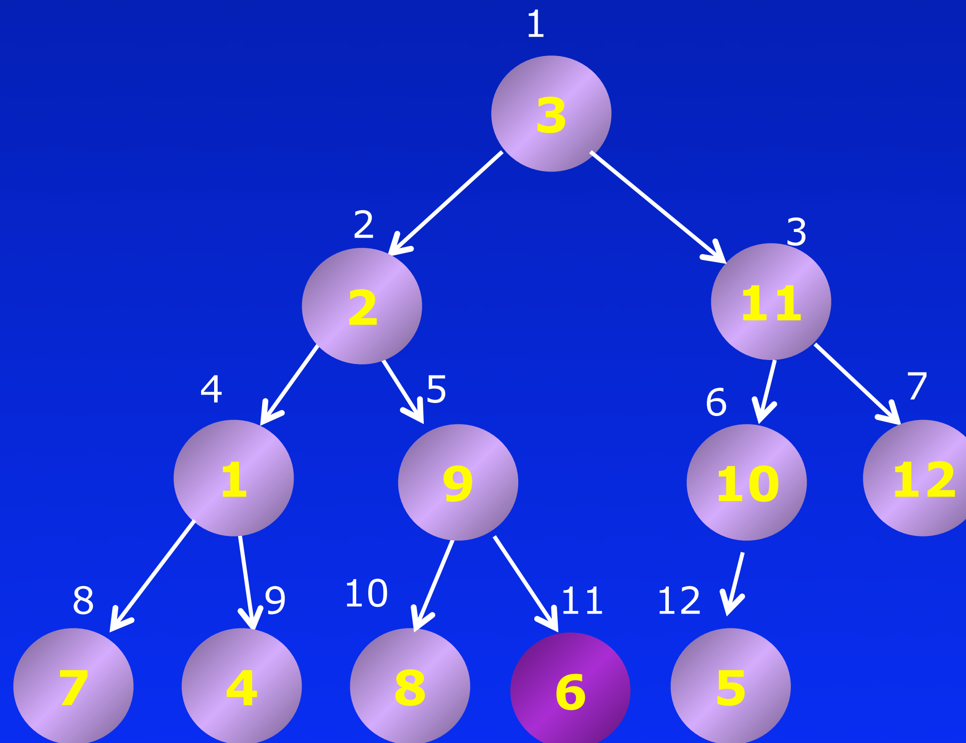


build_heapify-Funktion

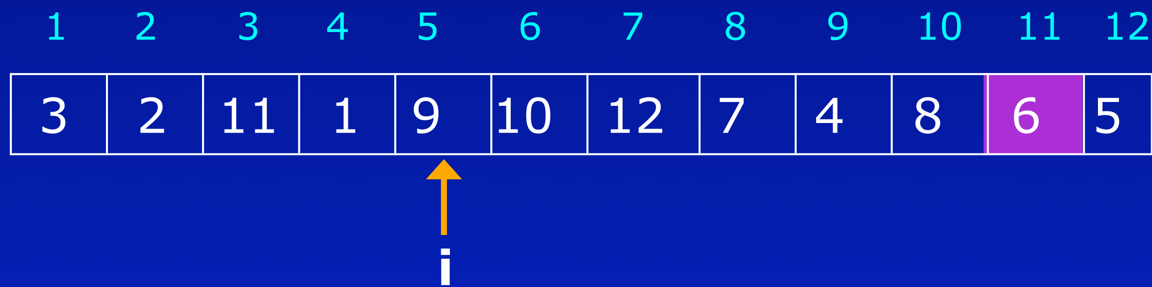
1	2	3	4	5	6	7	8	9	10	11	12
3	2	11	1	9	10	12	7	4	8	6	5

i

max_heapify (H, 5)

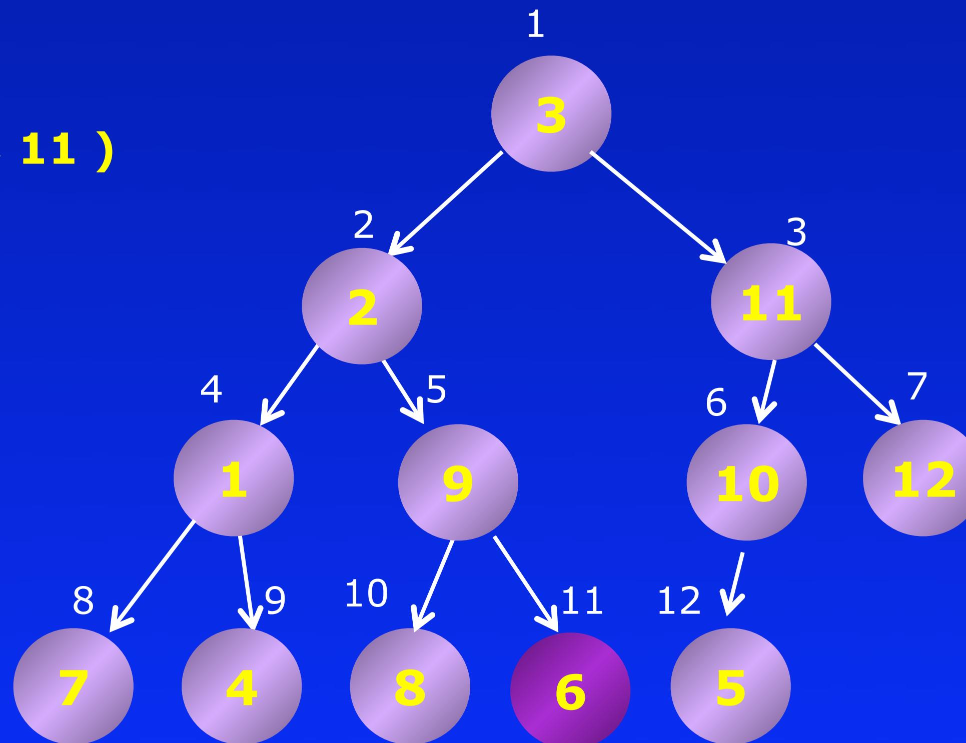


build_heapify-Funktion



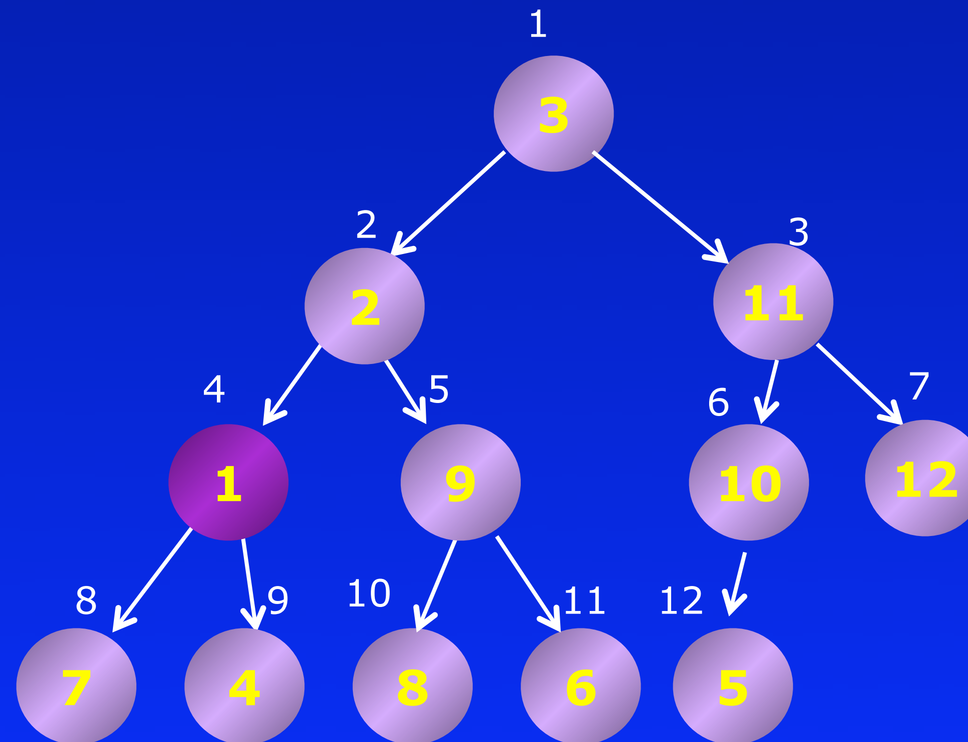
max_heapify (H, 5)

max_heapify (H, 11)



build_heapify-Funktion

max_heapify (H, 4)

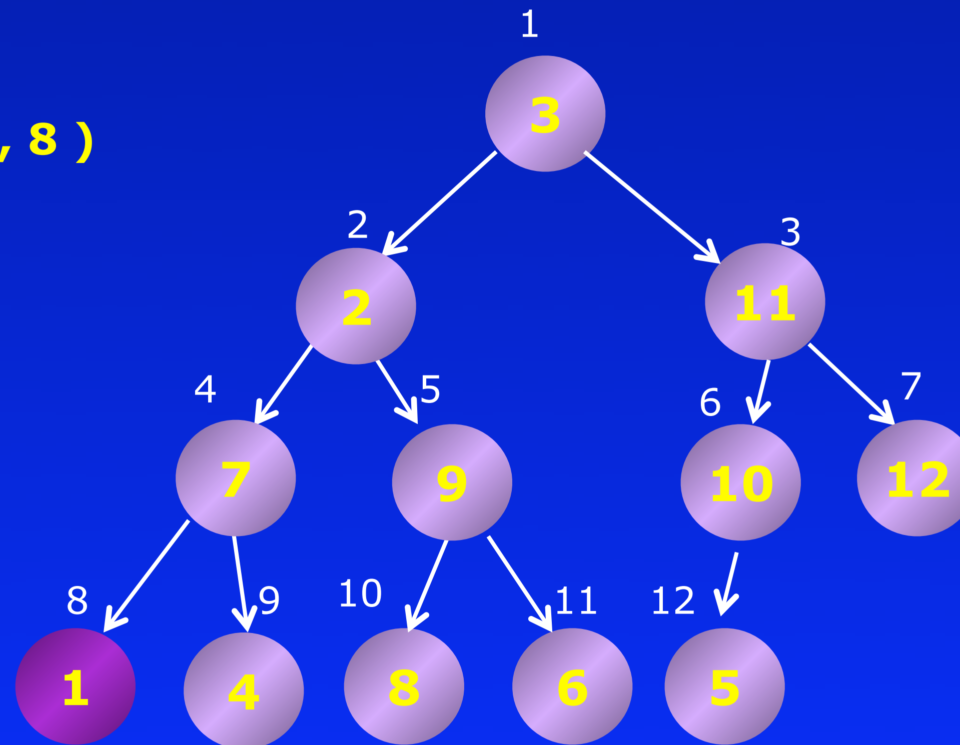


M. Esponda

build_heapify-Funktion

max_heapify (H, 4)

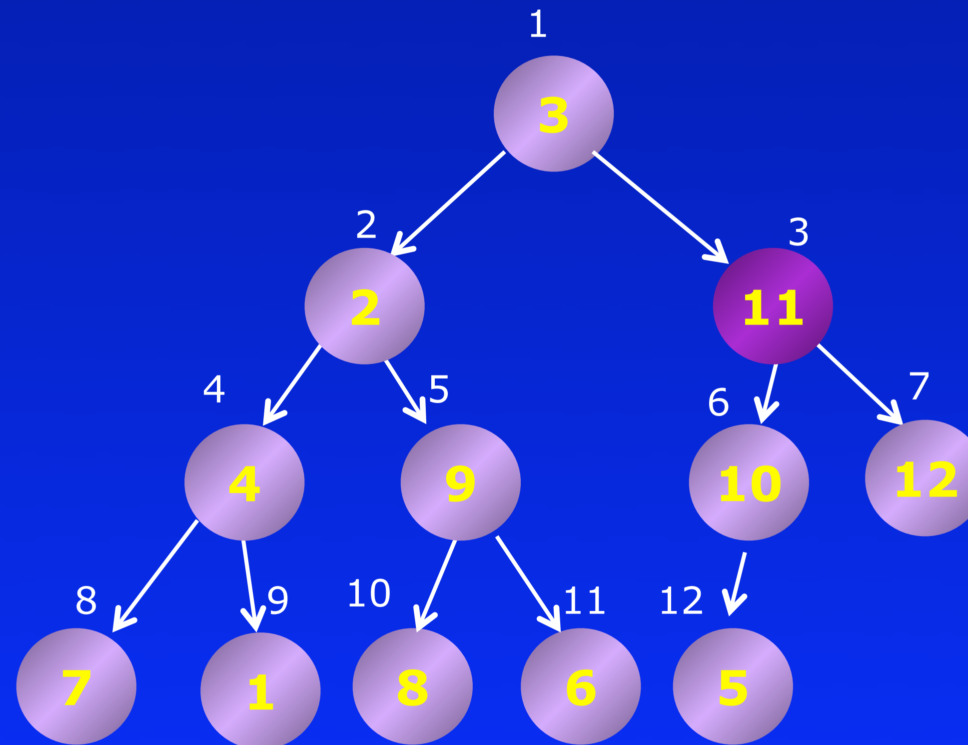
max_heapify (H, 8)



M. Esponda

build_heapify-Funktion

max_heapify (H, 3)

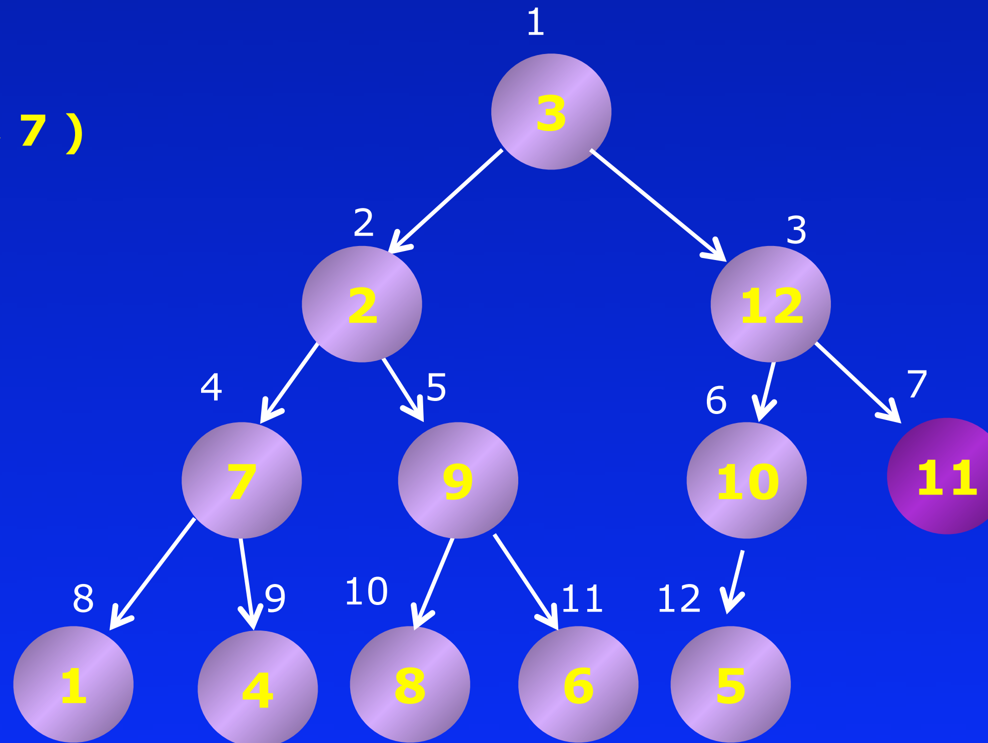


M. Esponda

build_heapify-Funktion

max_heapify (H, 3)

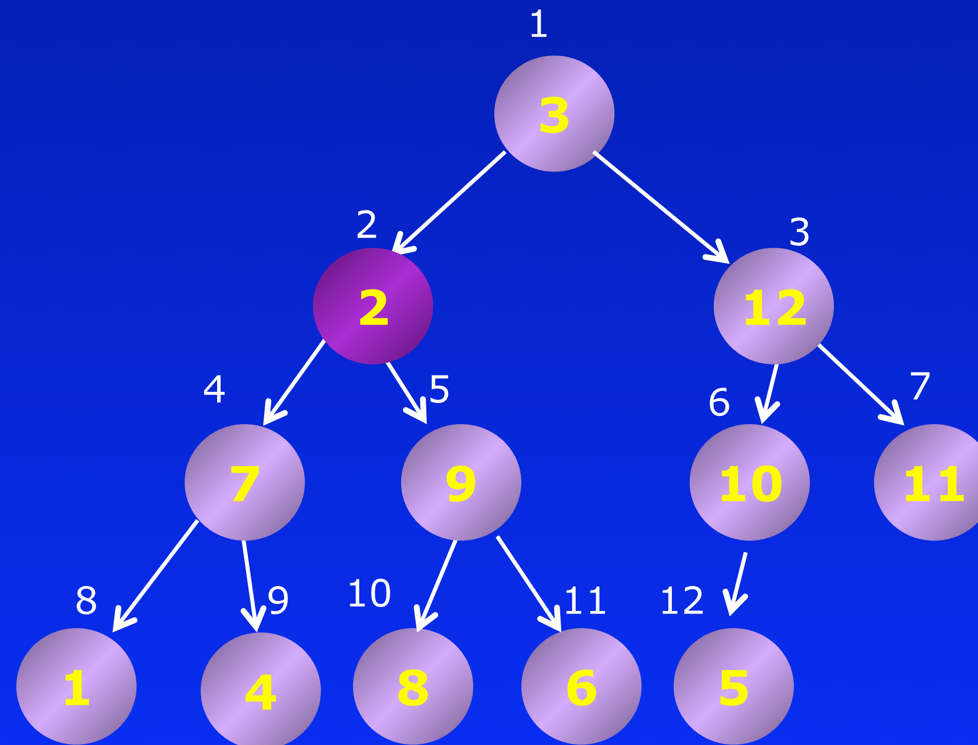
max_heapify (H, 7)



M. Esponda

build_heapify-Funktion

max_heapify (H, 2)

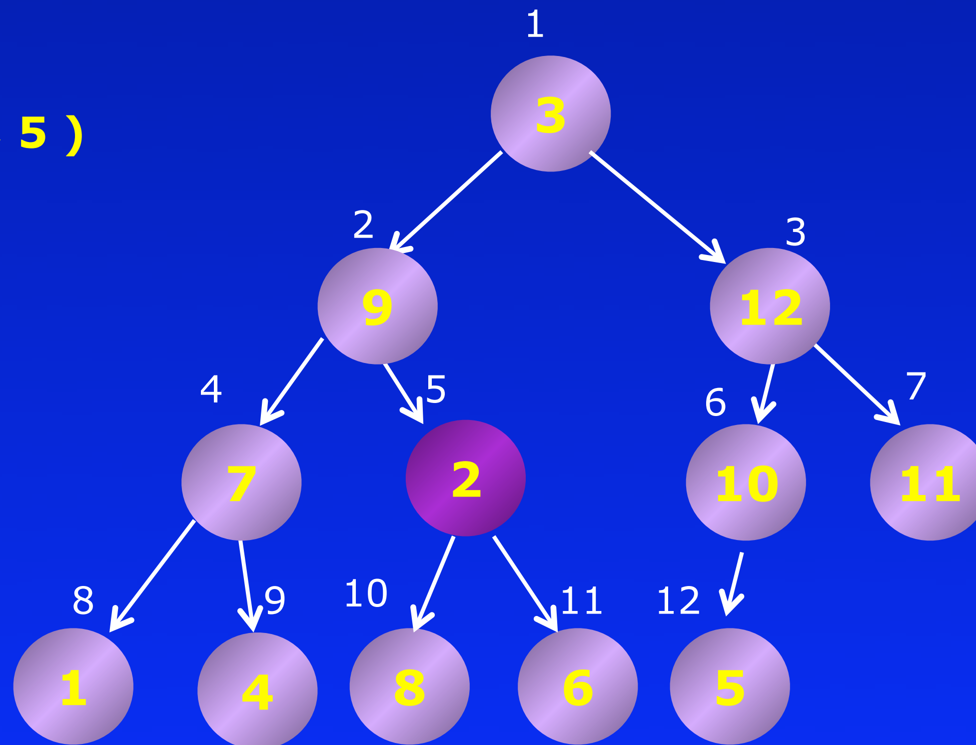


M. Esponda

build_heapify-Funktion

max_heapify (H, 2)

max_heapify (H, 5)



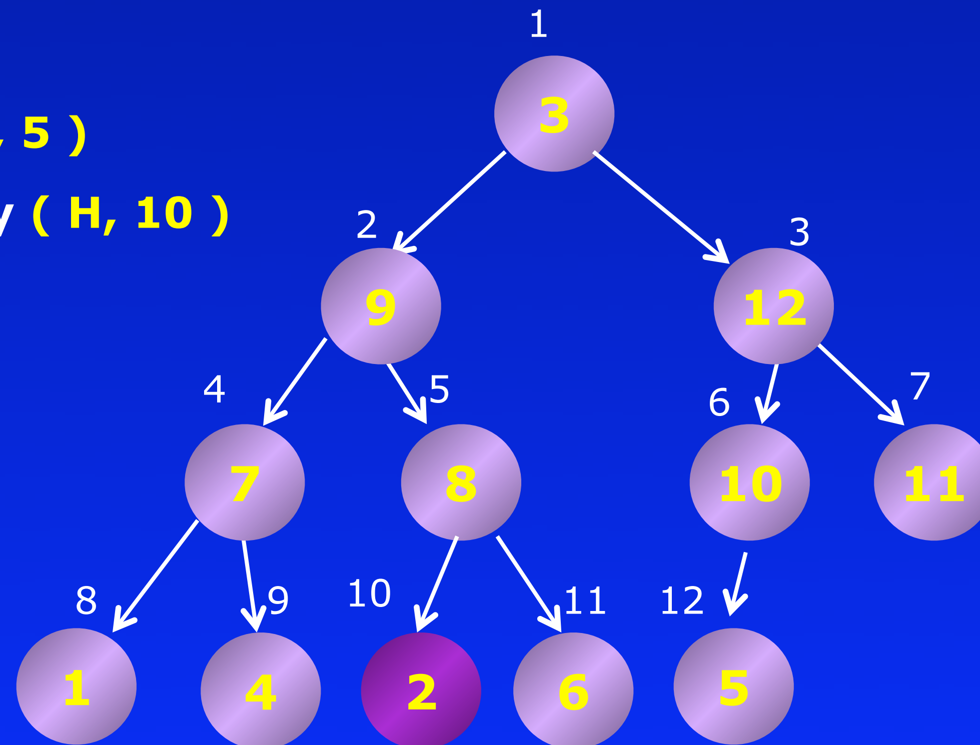
M. Esponda

build_heapify-Funktion

max_heapify (H, 2)

max_heapify (H, 5)

max_heapify (H, 10)



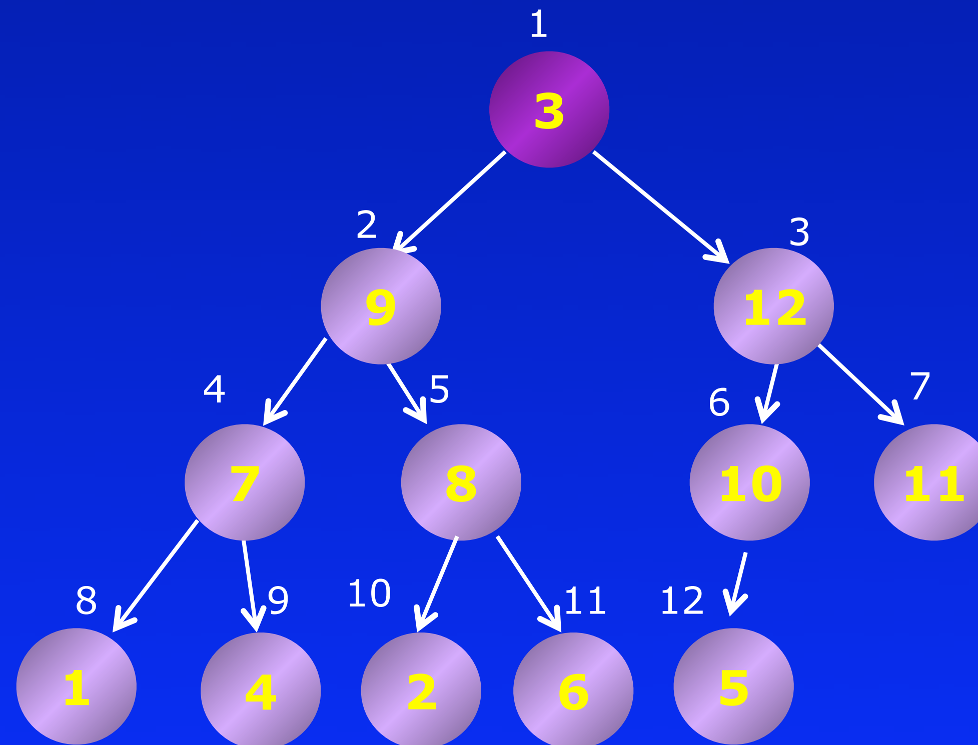
M. Esponda

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)



M. Esponda

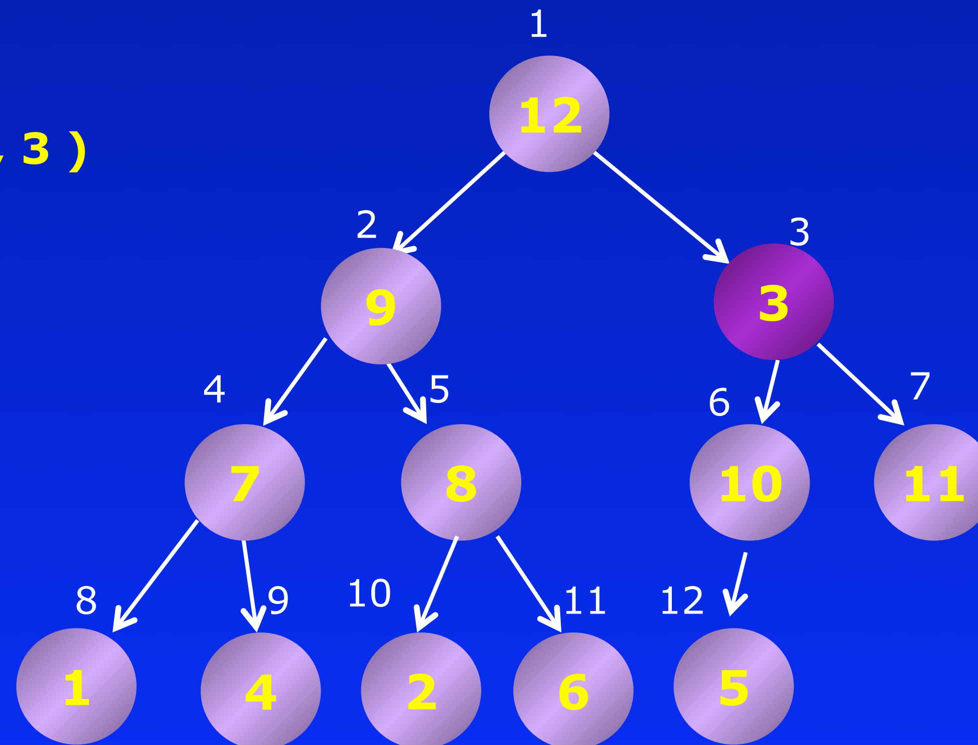
build_heapify-Funktion

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

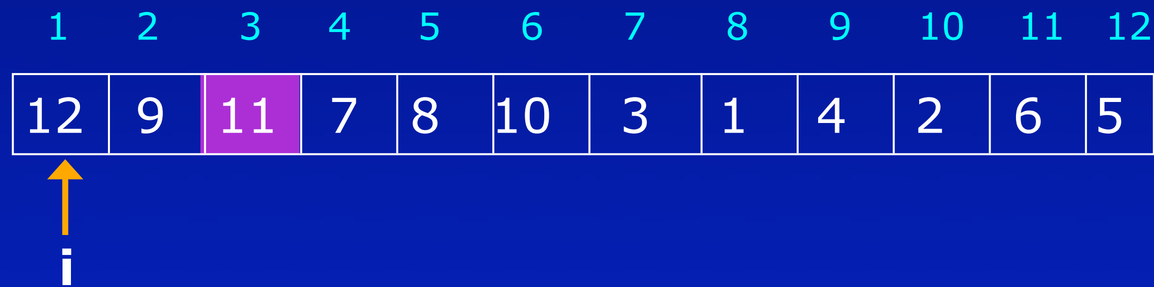
max_heapify (H, 1)

max_heapify (H, 3)



M. Esponda

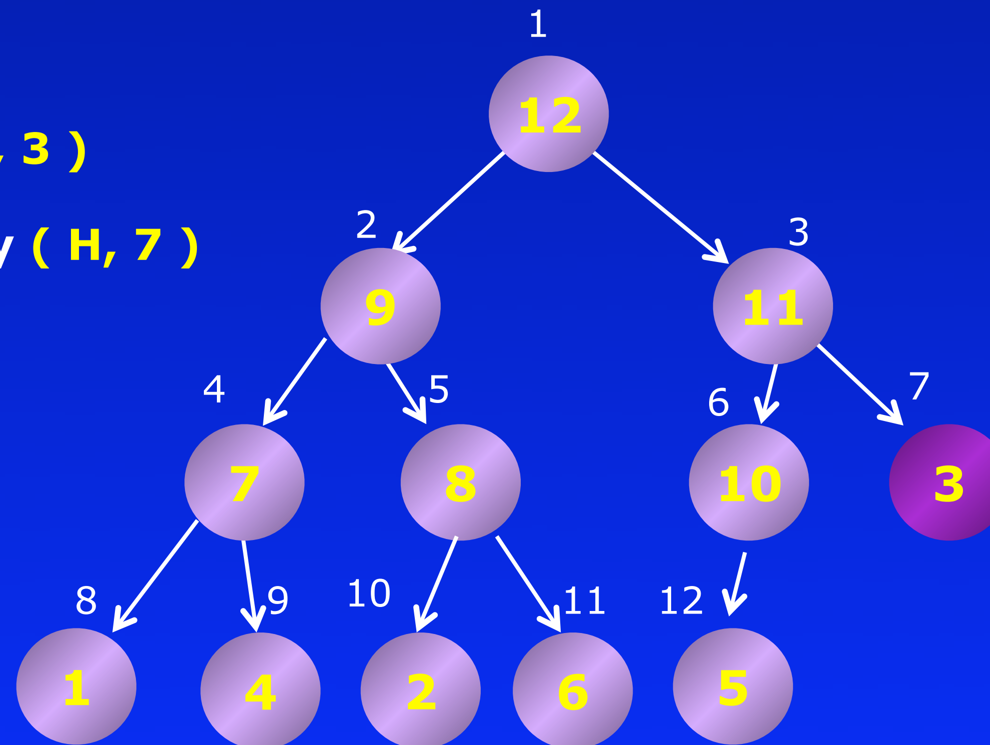
build_heapify-Funktion



max_heapify (H, 1)

max_heapify (H, 3)

max_heapify (H, 7)



M. Esponda

build_heapify-Funktion

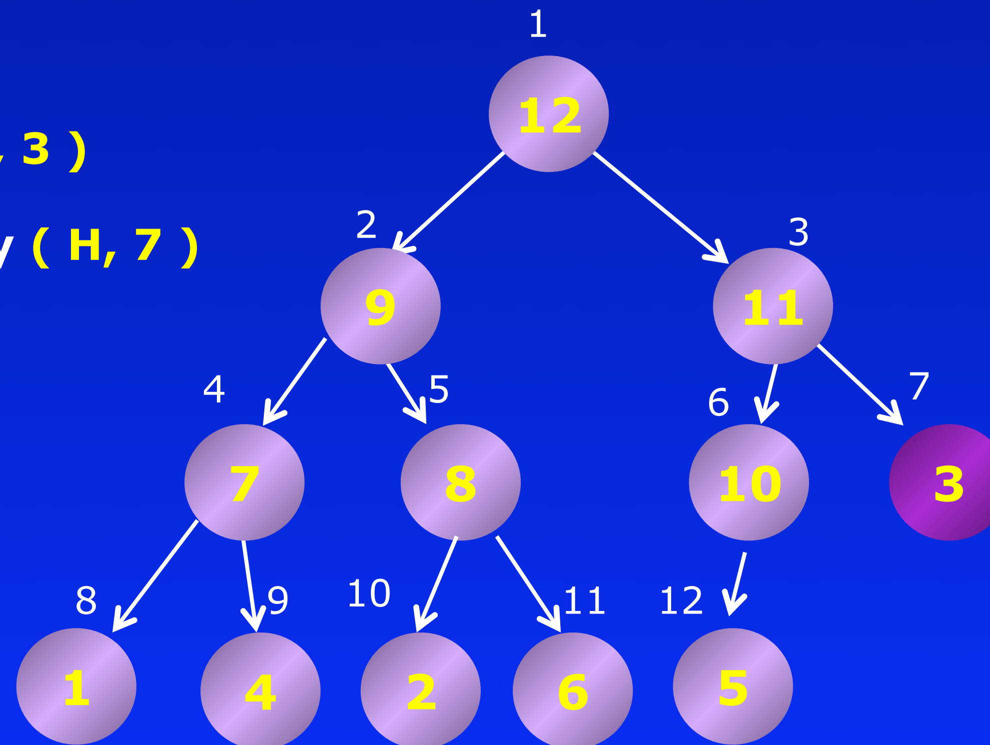
1	2	3	4	5	6	7	8	9	10	11	12
12	9	3	7	8	10	11	1	4	2	6	5

i

max_heapify (H, 1)

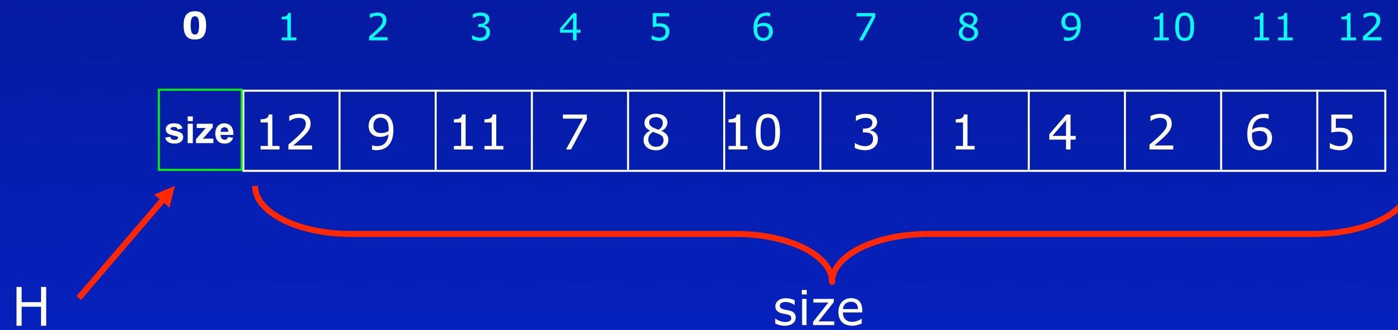
max_heapify (H, 3)

max_heapify (H, 7)



M. Esponda

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.

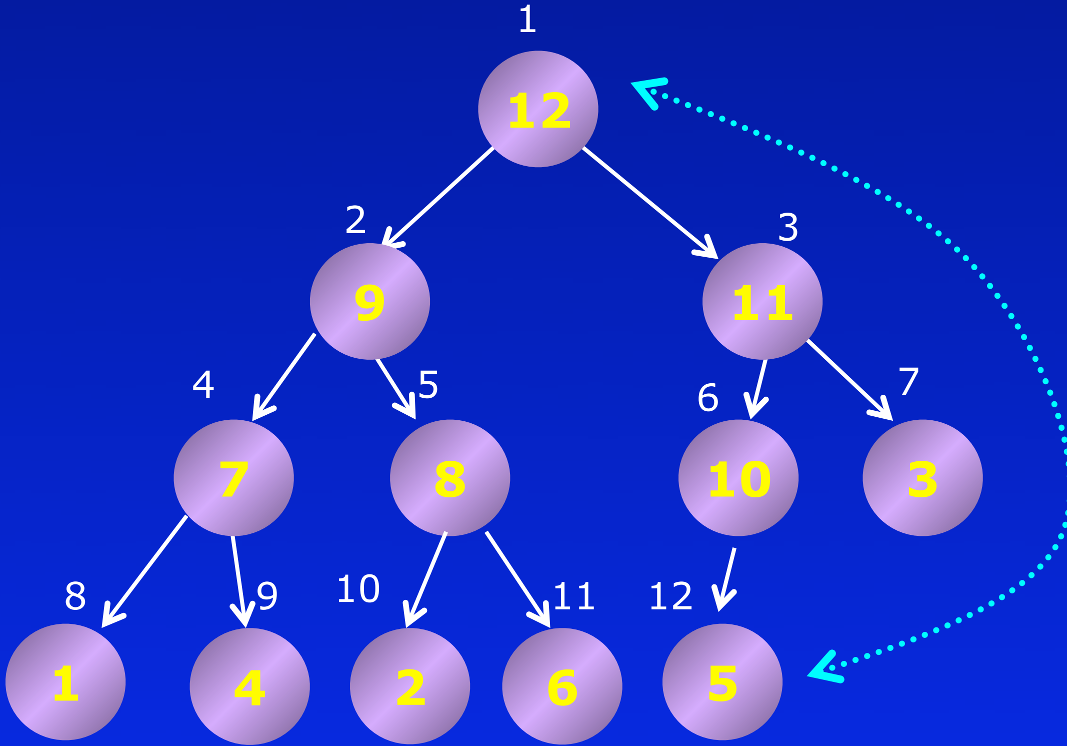
2 Das Element an der Wurzel des Heaps wird gegen das letzte Element des Heaps vertauscht.

3 Die Heap-Größe wird um eins dekrementiert.

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

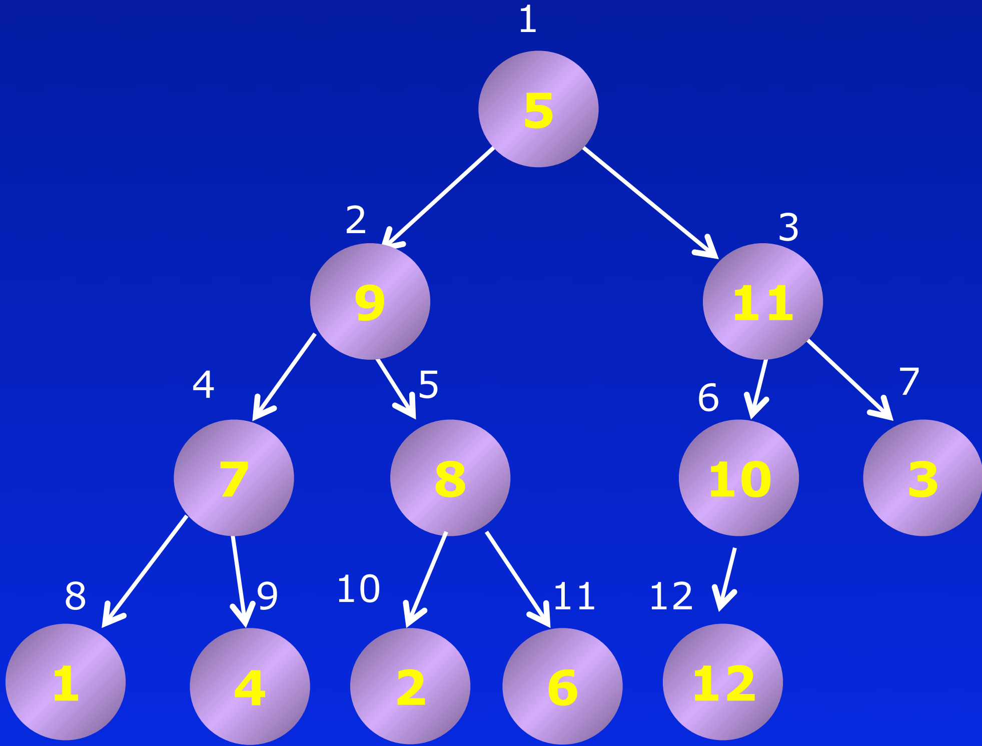
5 Die Schritte **2** bis **4** werden wiederholt, solange der Heap größer oder gleich zwei ist.

Heapsort-Funktion



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

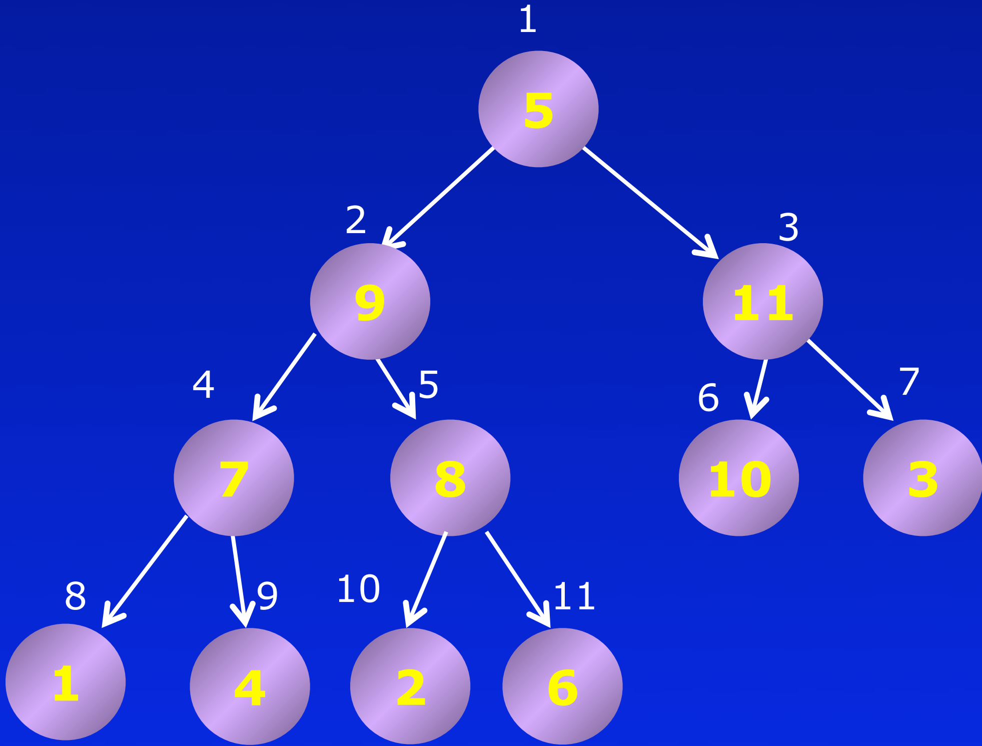
Heapsort-Funktion



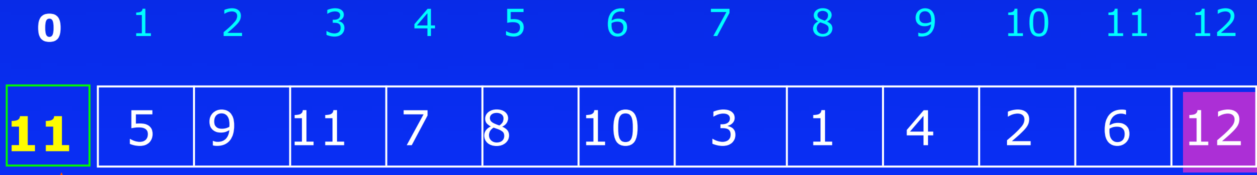
Erstes sortiertes Element

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



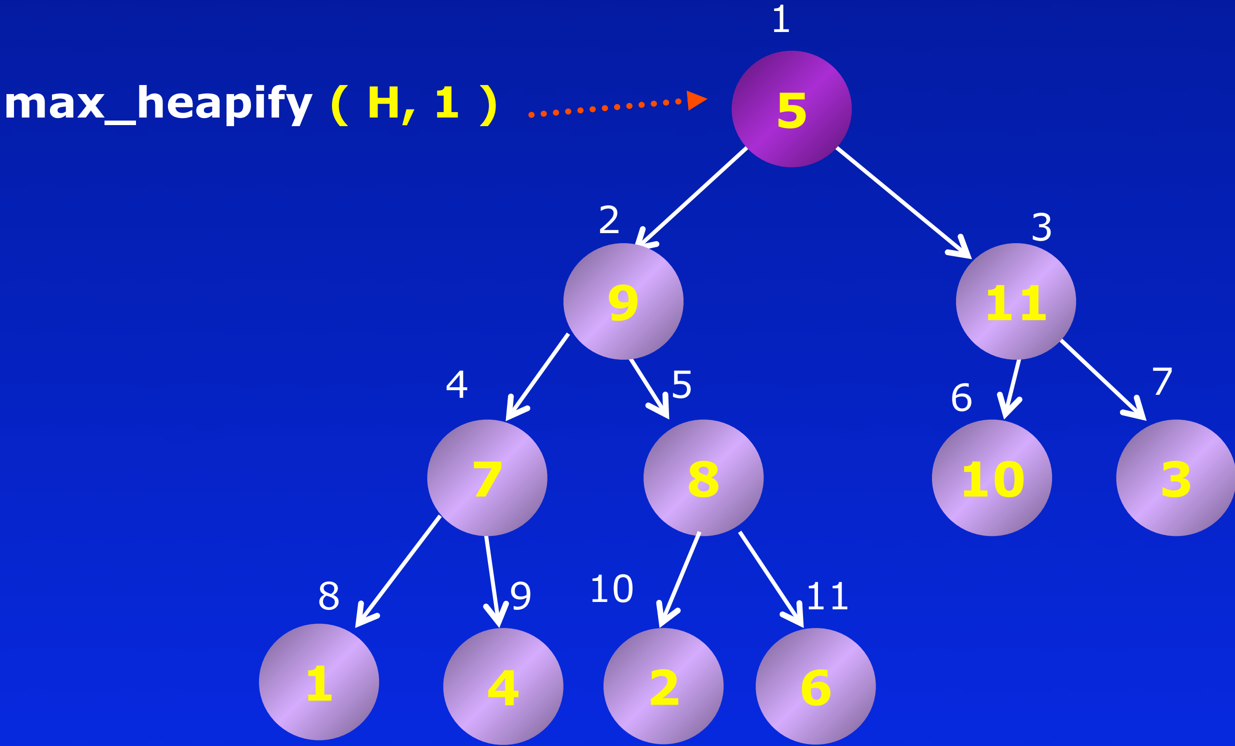
nicht mehr in
dem Heap!



Die Heap-Größe wird um eins verkleinert

M. Esponda

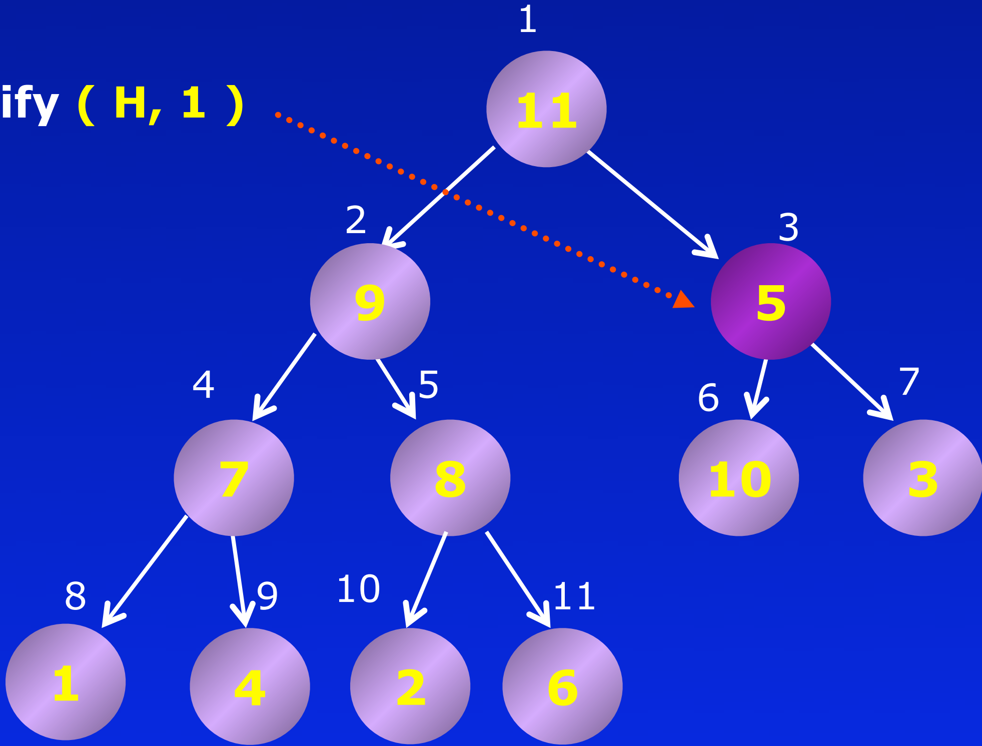
Heapsort-Funktion



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

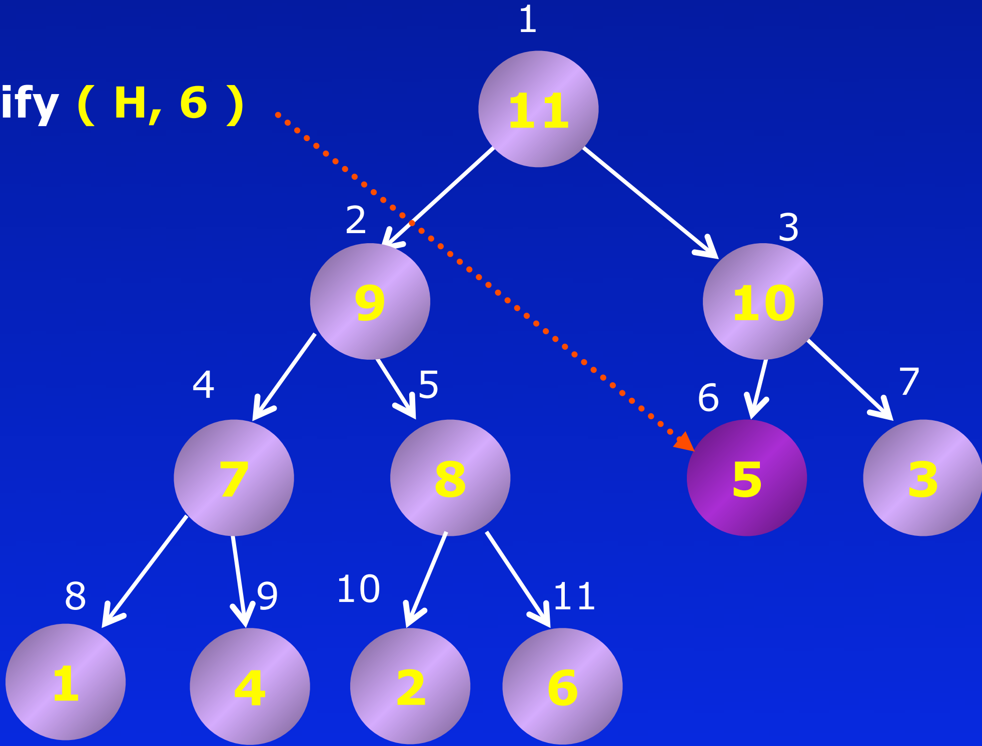
max_heapify (H, 1)



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

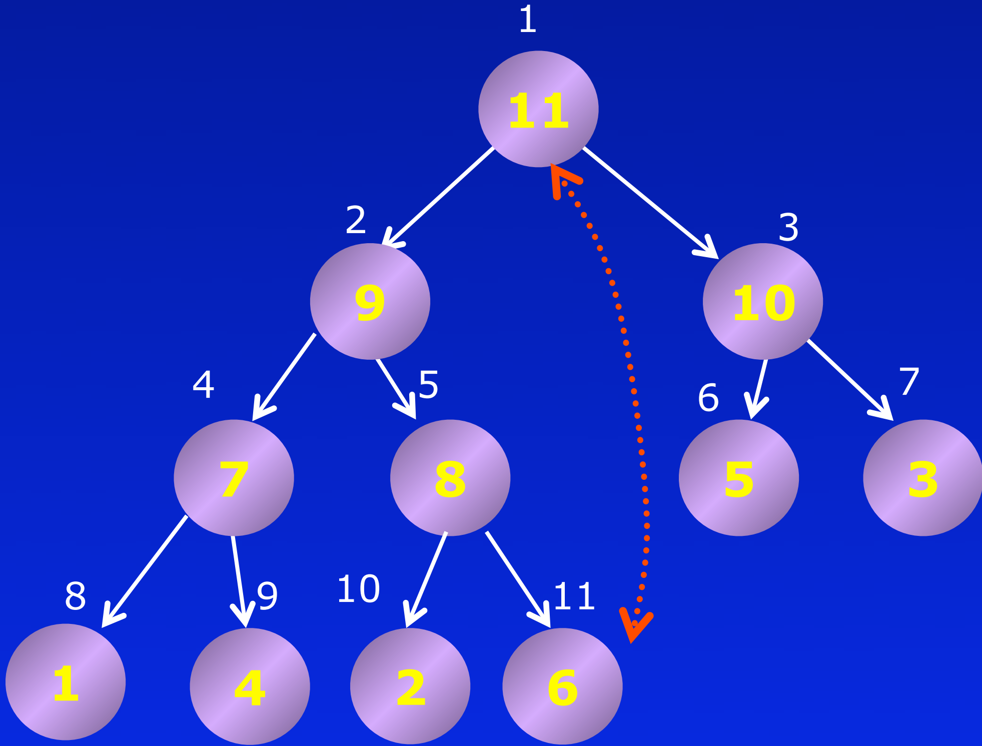
max_heapify (H, 6)



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

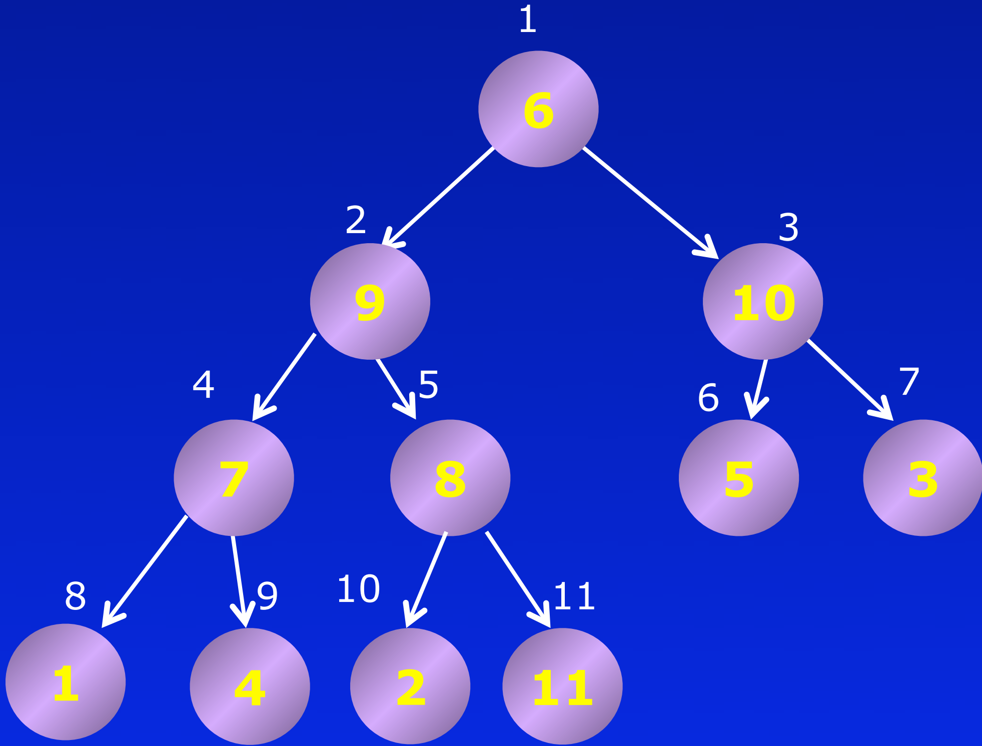
M. Esponda

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

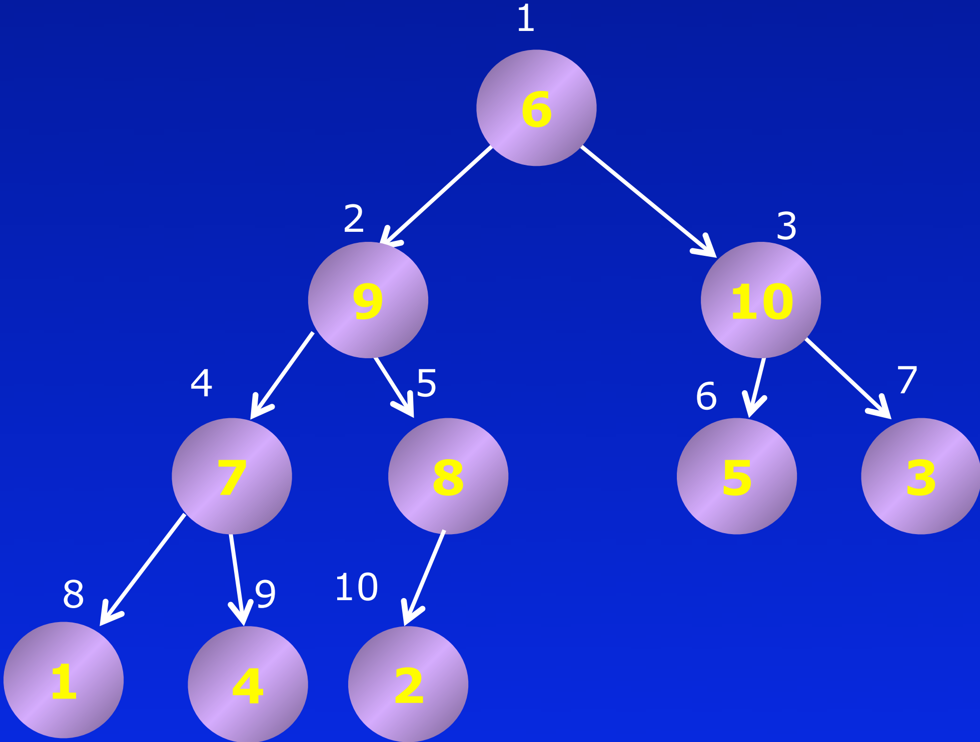
Heapsort-Funktion



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

M. Esponda

Heapsort-Funktion

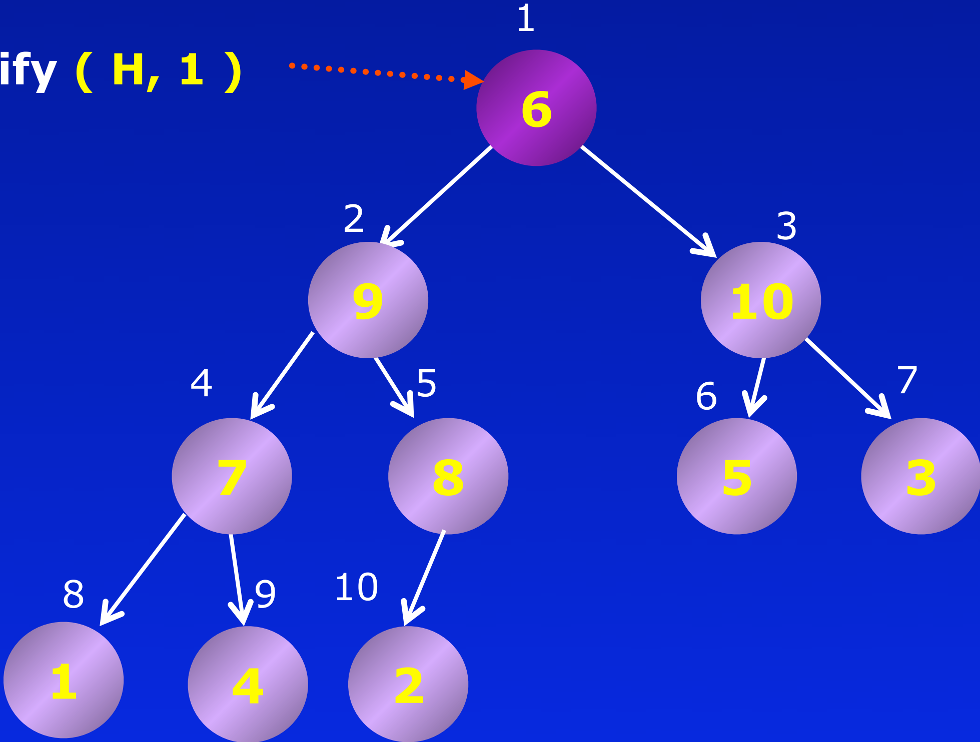


0	1	2	3	4	5	6	7	8	9	10	11	12
10	6	9	10	7	8	5	3	1	4	2	11	12

M. Esponda

Heapsort-Funktion

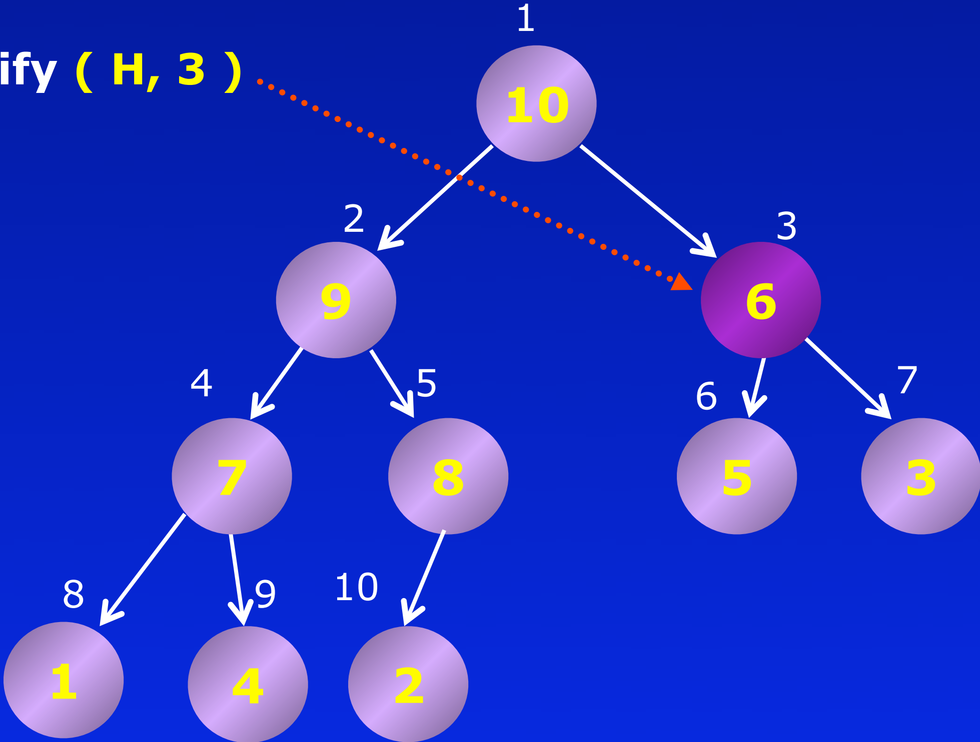
max_heapify (H, 1)



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

M. Esponda

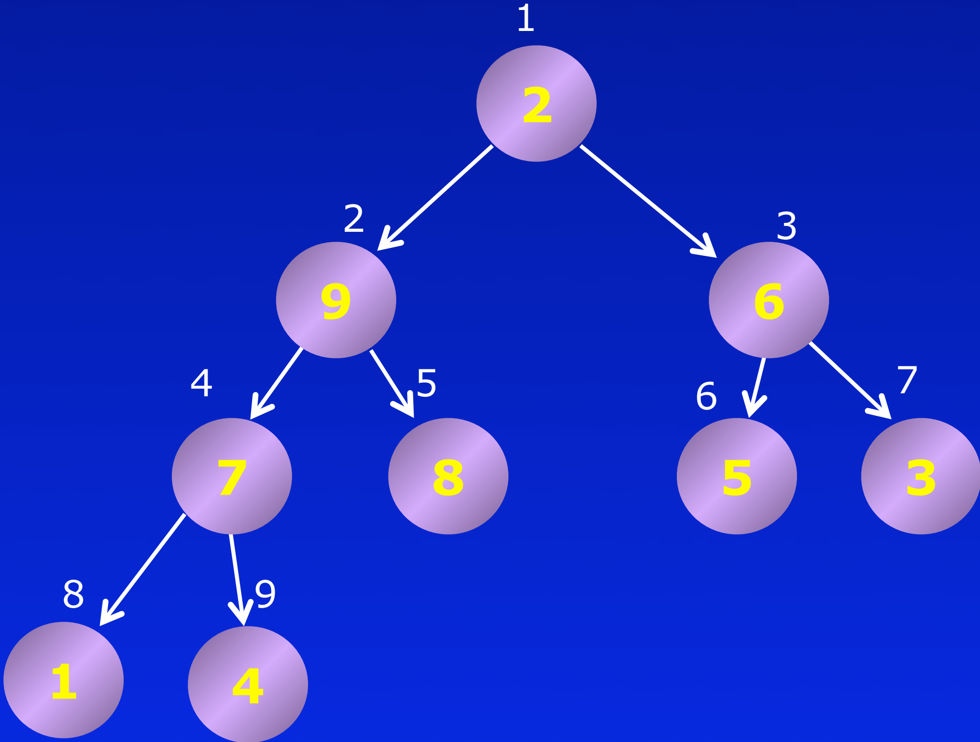
Heapsort-Funktion



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

M. Esponda

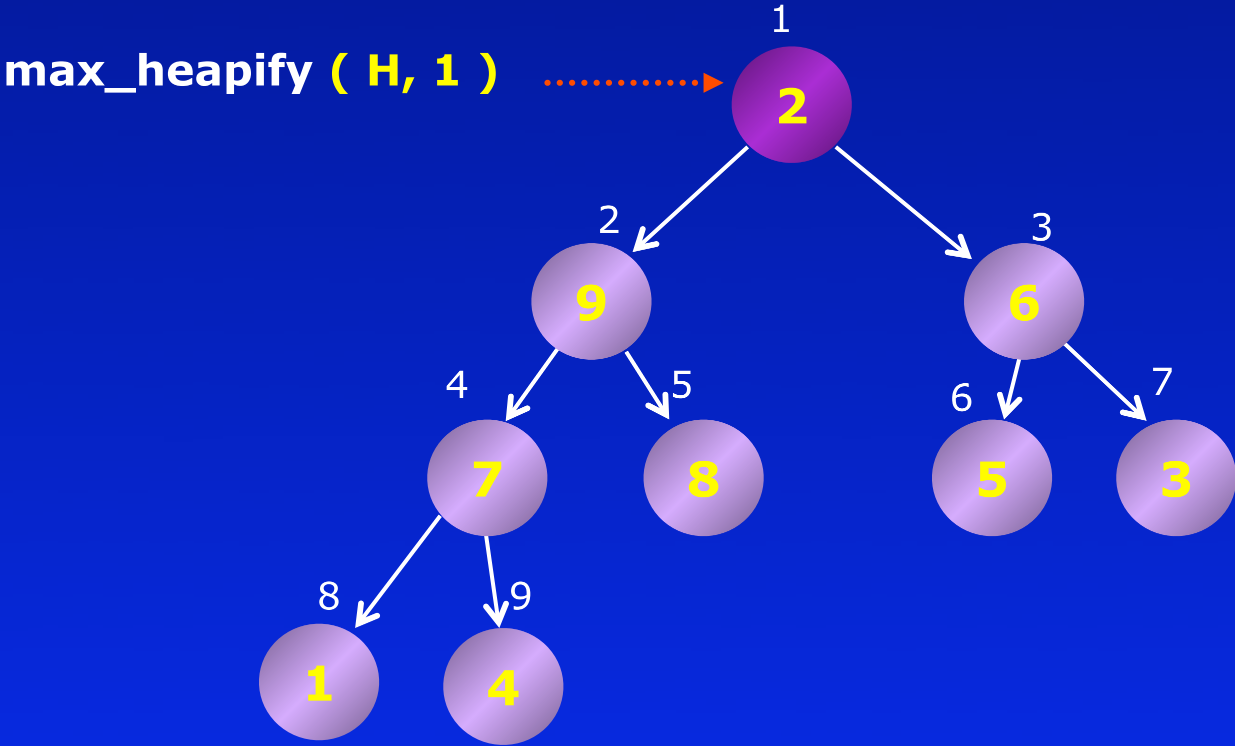
Heapsort-Funktion



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

M. Esponda

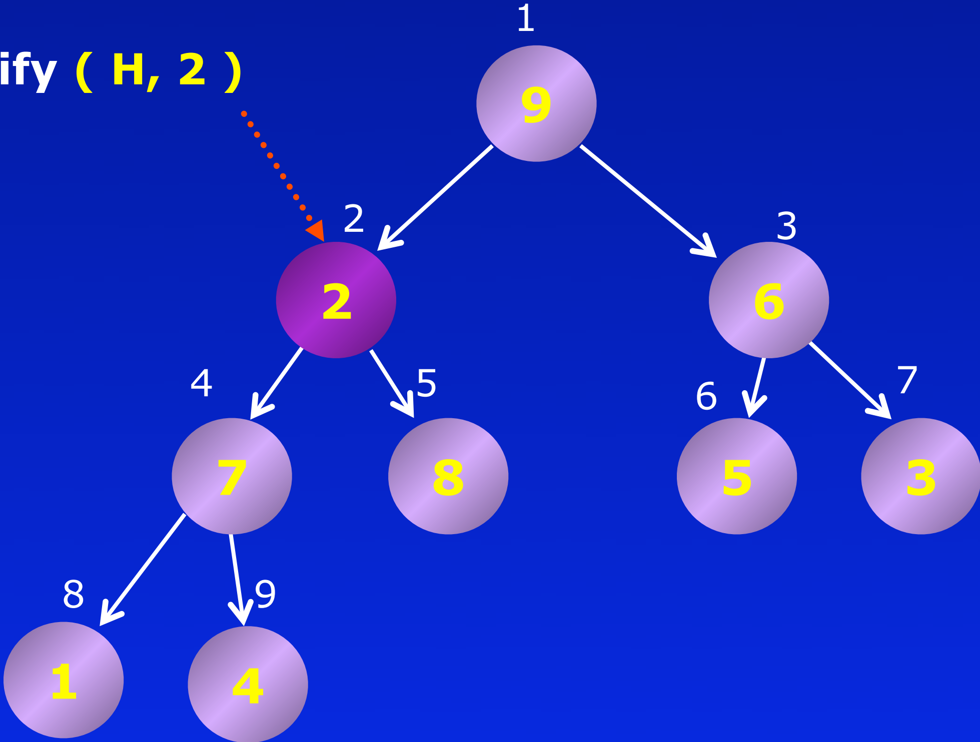
Heapsort-Funktion



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

M. Esponda

Heapsort-Funktion



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

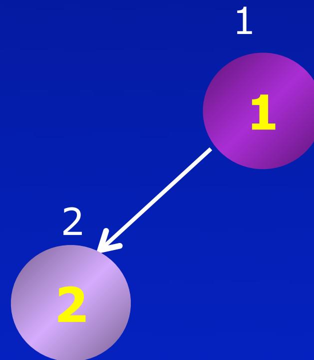
M. Esponda

Heapsort-Funktion

max_heapify (H, 2)

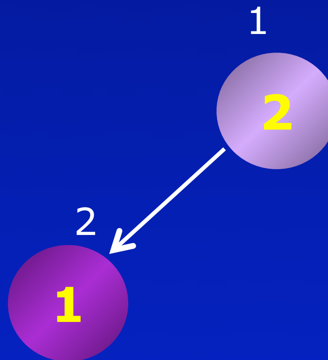
0	1	2	3	4	5	6	7	8	9	10	11	12
9	9	8	6	7	2	5	3	1	4	10	11	12

Heapsort-Funktion



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

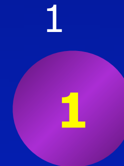
Heapsort-Funktion



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

1



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) → $n \times \log(n)$

for-Schleife → $n-1$

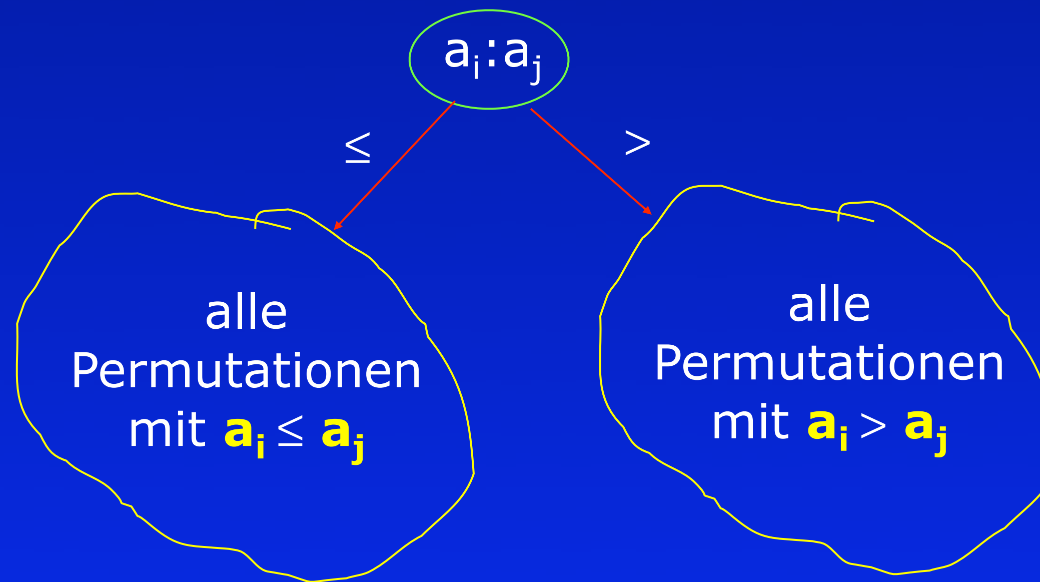
max_heapify → $\log(n)$

Schleife insgesamt: $n \times \log(n)$

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

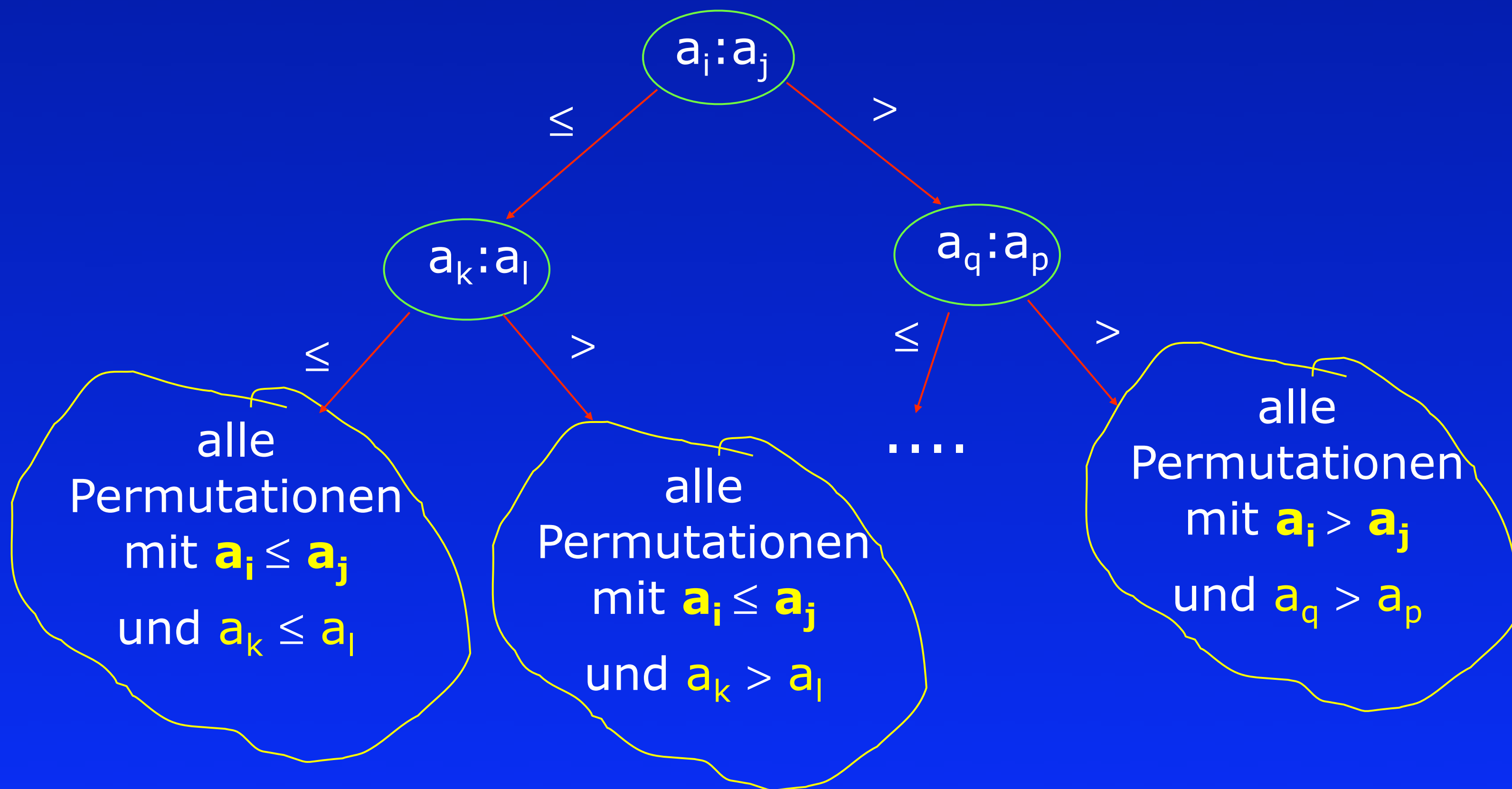
Ist es möglich besser zu sortieren?

Entscheidungsbaum



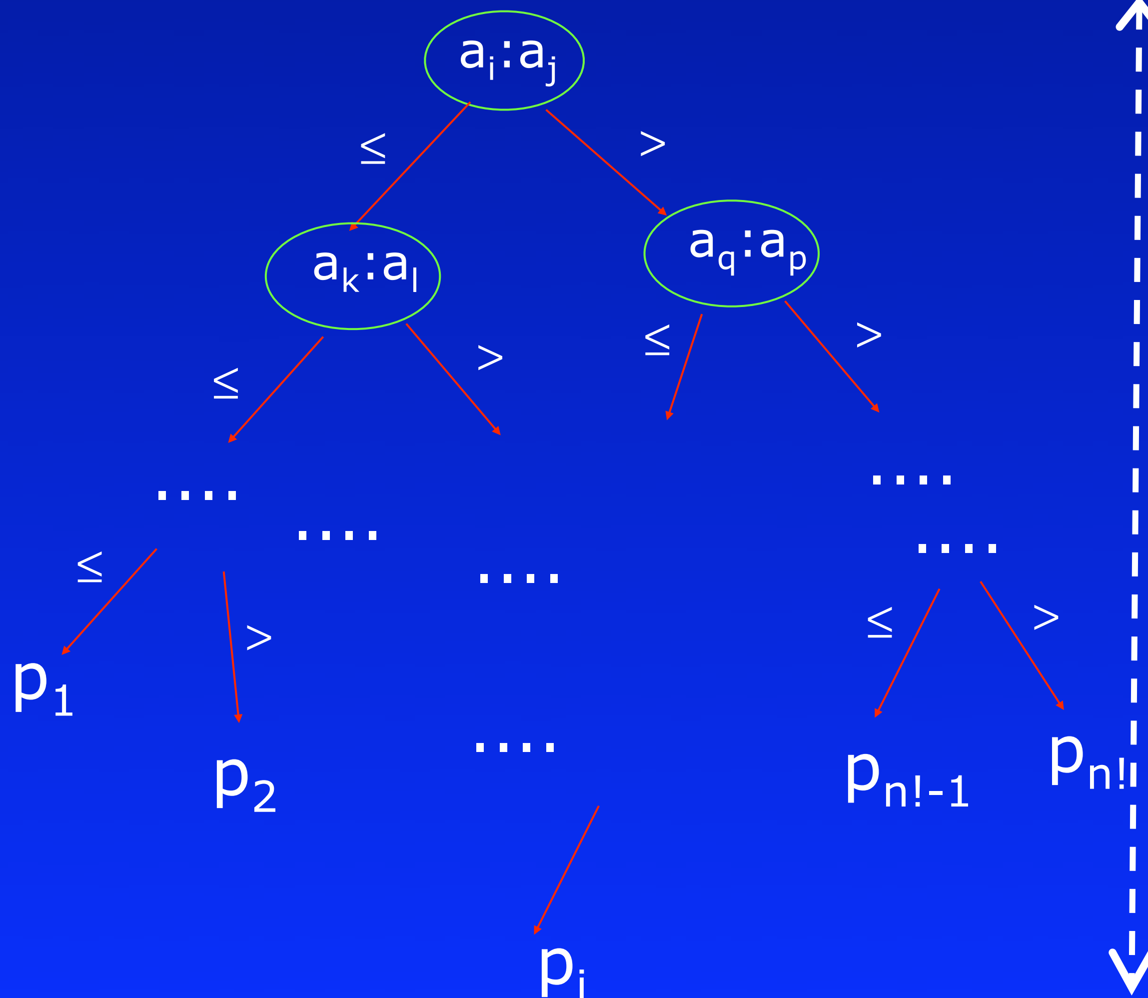
Ist es möglich besser zu sortieren?

Entscheidungsbaum



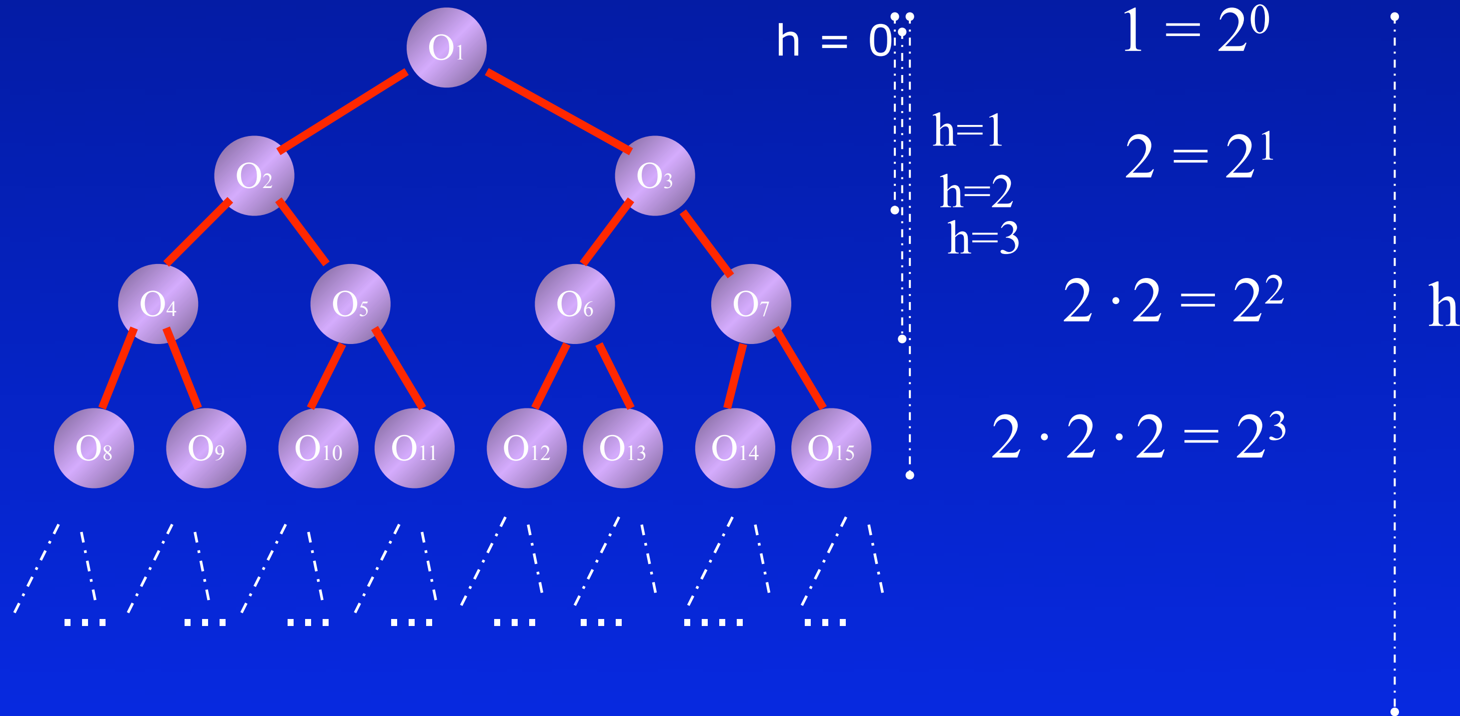
M. Esponda

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

$$h \approx \log_2(n)$$

M. Esponda

Ist es möglich schneller zu sortieren? Nein!

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

Wir brauchen nur die Höhe des Entscheidungsbaums zu bestimmen.

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

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

Weil ein Binärbaum mit Höhe h maximal 2^h Blätter hat, gilt

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

Wenn wir den Logarithmus berechnen, gilt

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

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

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

M. Esponda

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.

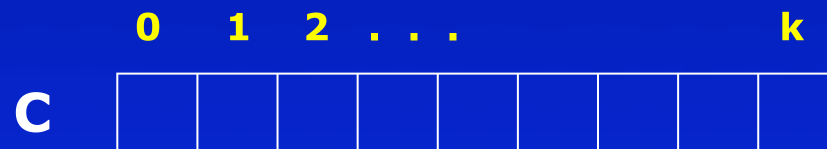
$$T(n) = O(n)$$

Anzahl der Daten, die sortiert werden sollen

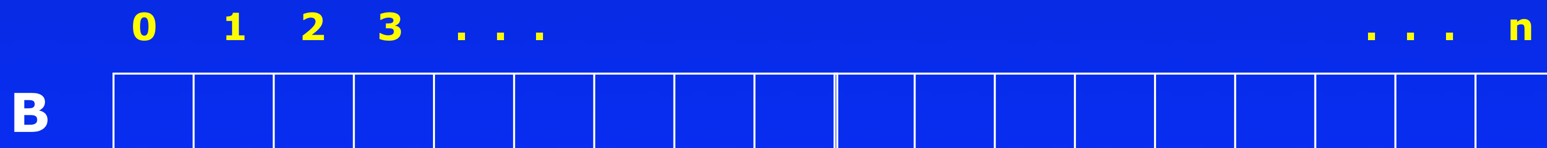
Counting sort

Der "Counting sort"-Algorithmus benutzt zwei Hilfsfelder.

1. ein C-Feld, das so groß ist wie der Wertebereich 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.

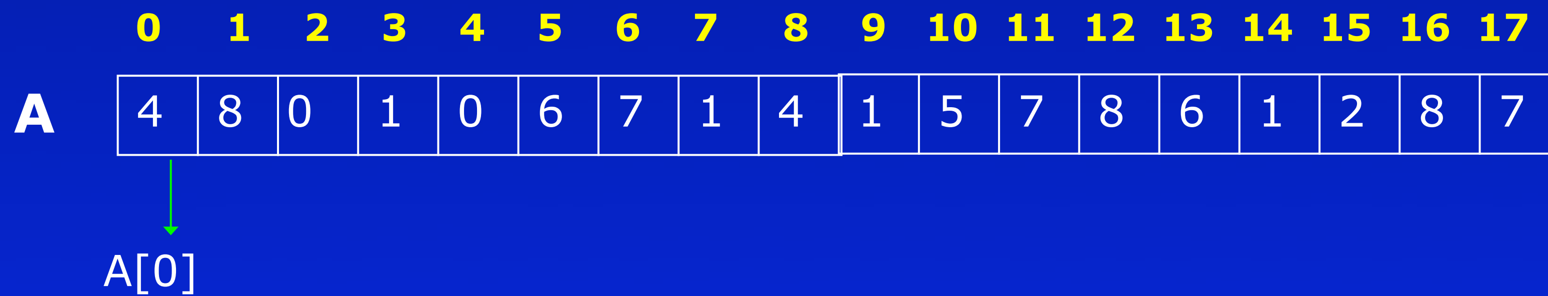
	0	1	2	. . .					k
C	0	0	0	0	0	0	0	0	0

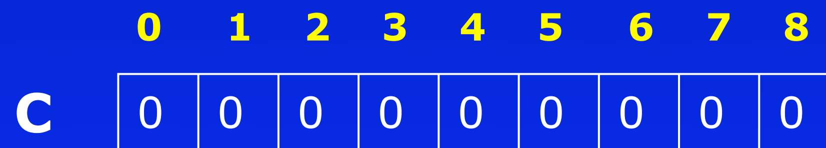
Counting sort

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

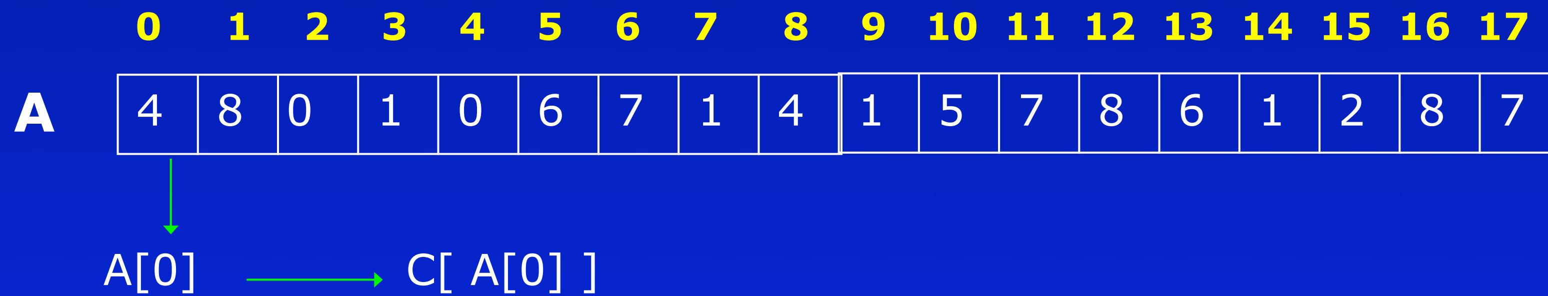
	0	1	2	3	4	5	6	7	8
C	0	0	0	0	0	0	0	0	0

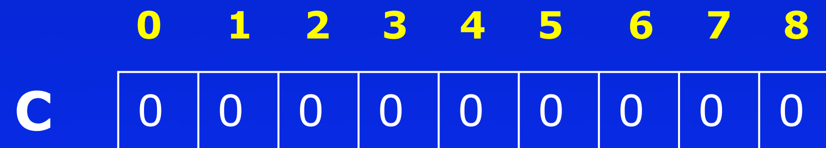
Counting sort



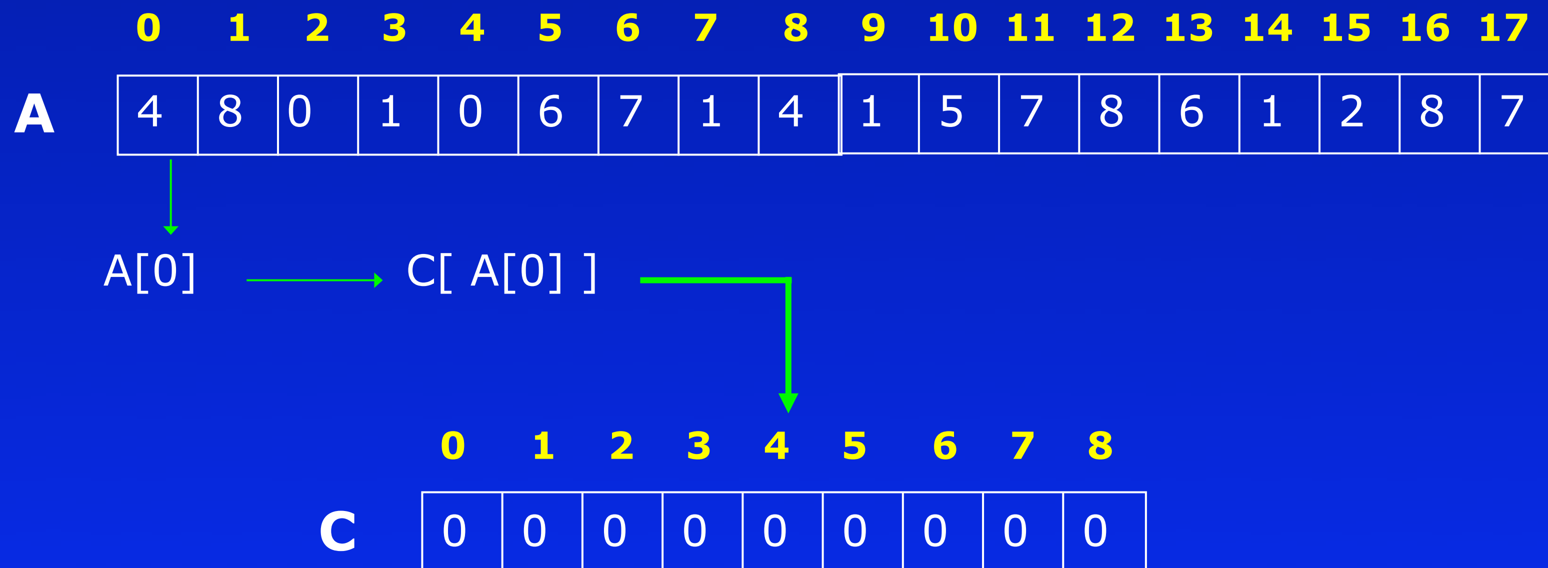


Counting sort

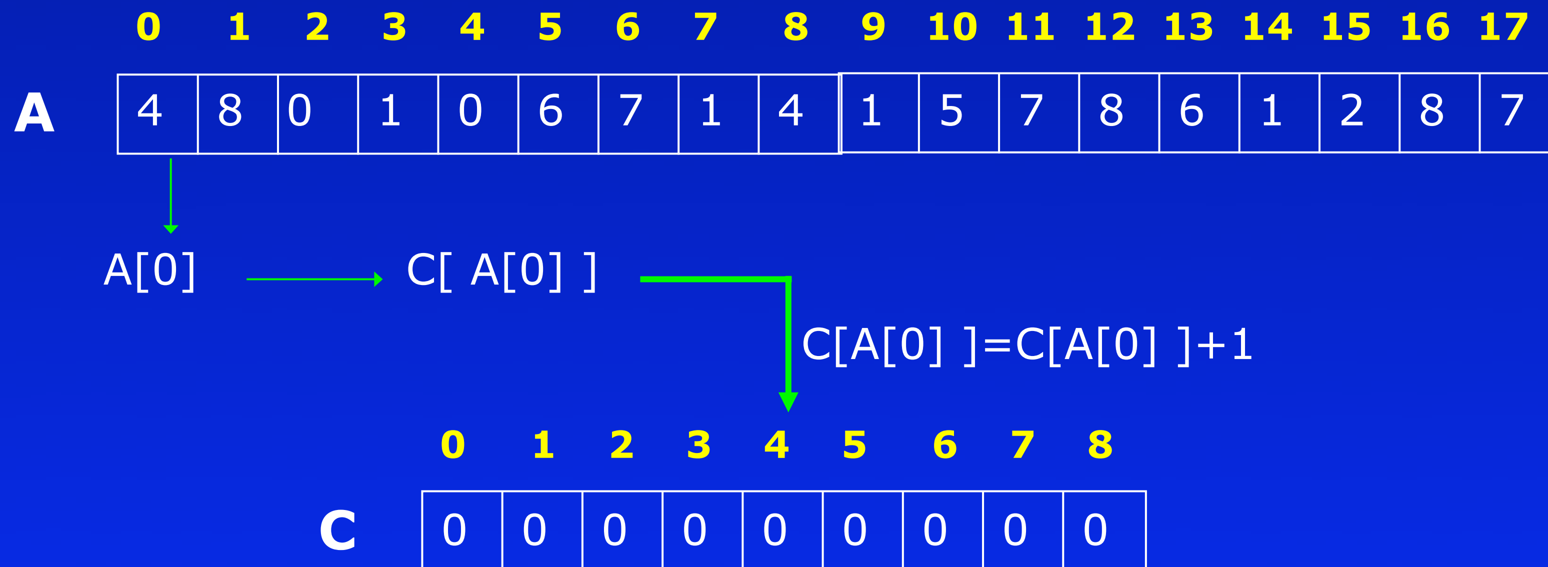




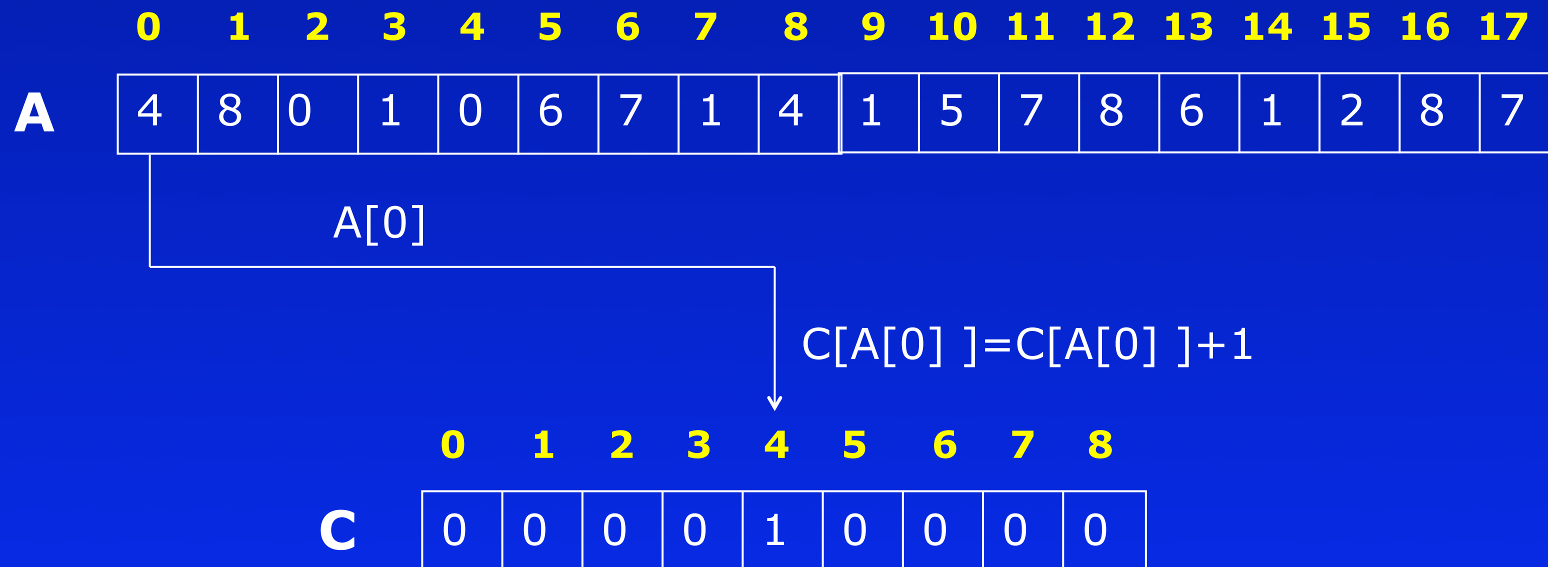
Counting sort



Counting sort



Counting sort

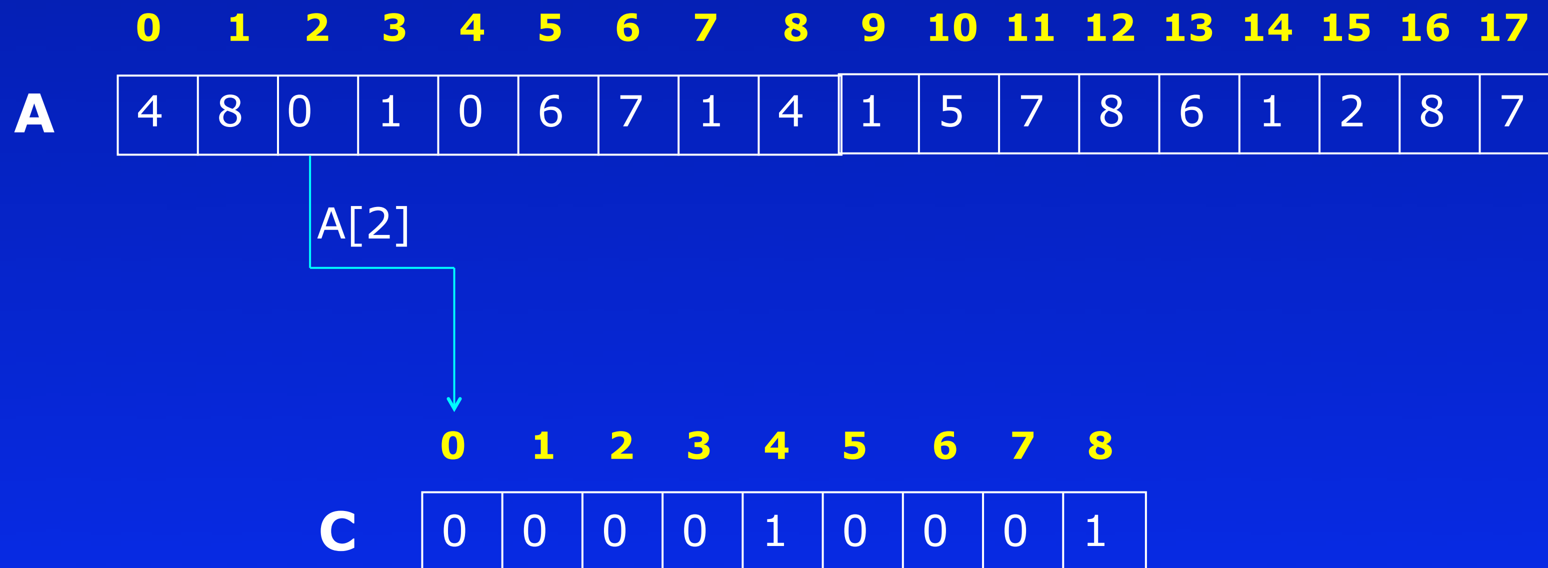


Counting sort

M. Esponda

Counting sort

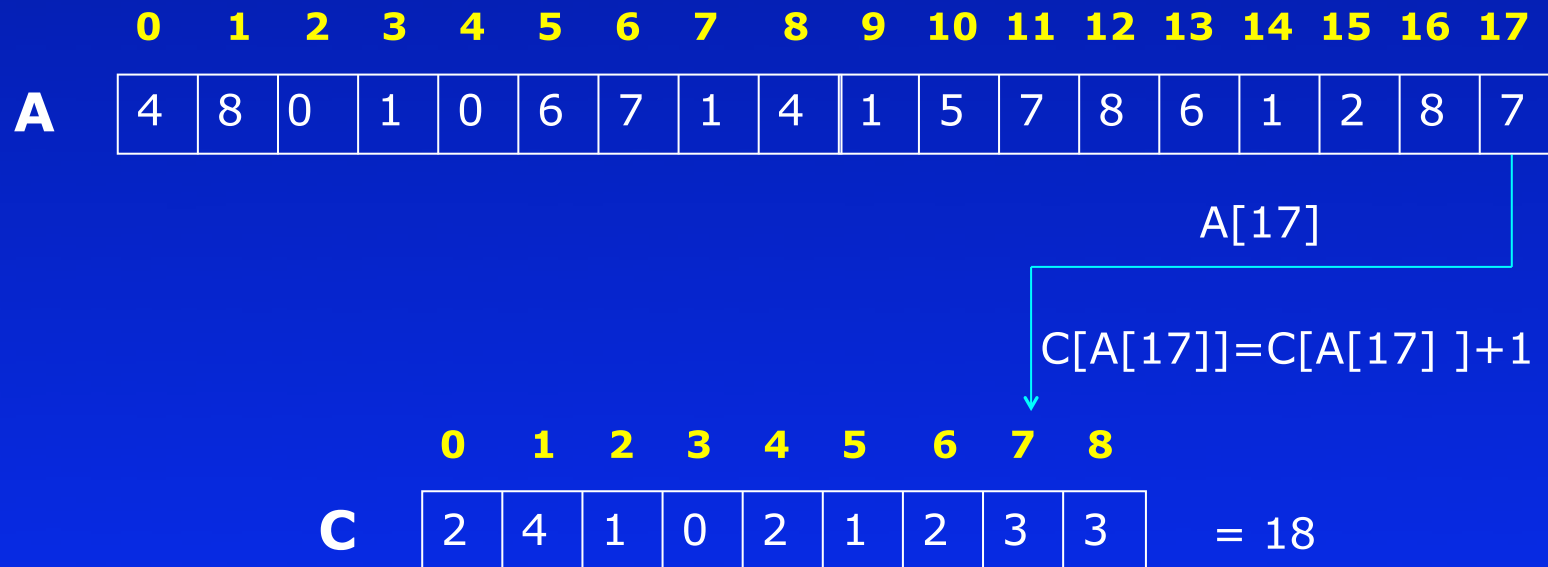
Counting sort



Counting sort

M. Esponda

Counting sort



Counting sort

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

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

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

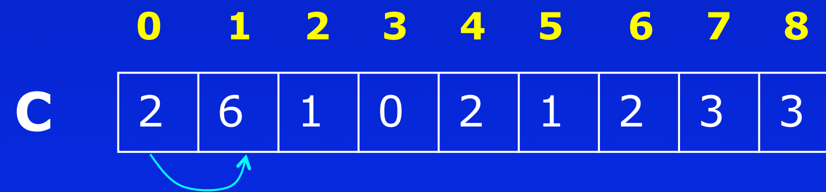
	0	1	2	3	. . .												. . .	n
B																		

M. Esponda

Counting sort

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

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



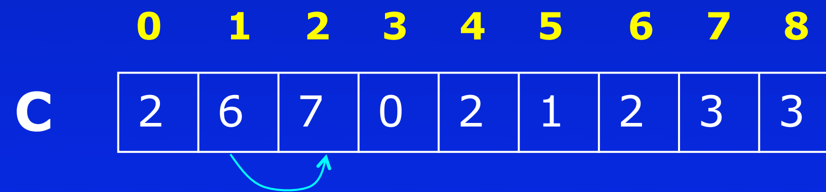
	0	1	2	3	. . .										. . .		n
B																	

M. Esponda

Counting sort

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

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



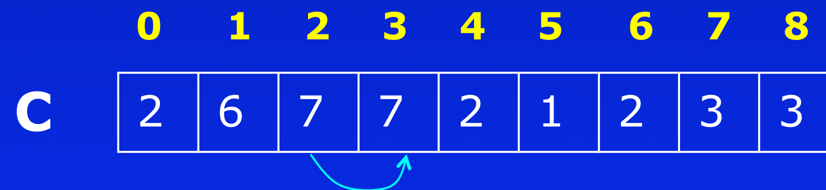
	0	1	2	3	. . .											. . .	n
B																	

M. Esponda

Counting sort

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

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



0 1 2 3 . . . . . . n

B

M. Esponda

Counting sort

	0	1	2	3	4	5	6	7	8
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	3	4	5	6	7	8
C	2	6	7	7	9	10	12	15	18

9 Zahlen sind kleiner oder gleich 4

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

M. Esponda

Counting sort

A

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

A[17]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	15	18

B

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

Counting sort

A

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

A[17]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	15	18

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

B

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

Counting sort

A

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

A[17]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	14	18

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

B

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

Counting sort

A

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

A[17]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	14	18

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

B

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

Counting sort

A

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

A[17]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	14	18

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

B

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

M. Esponda

Counting sort

A

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

A[17]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	14	18

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

B

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

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

M. Esponda

Counting sort

A

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

A[16]

C

0	1	2	3	4	5	6	7	8
2	6	7	7	9	10	12	14	18

B

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

M. Esponda

Counting sort

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

A[16]

	0	1	2	3	4	5	6	7	8
C	2	6	7	7	9	10	12	14	17

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

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

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

Counting sort

A

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

A[15]

C

0	1	2	3	4	5	6	7	8
2	6	6	7	9	10	12	14	17

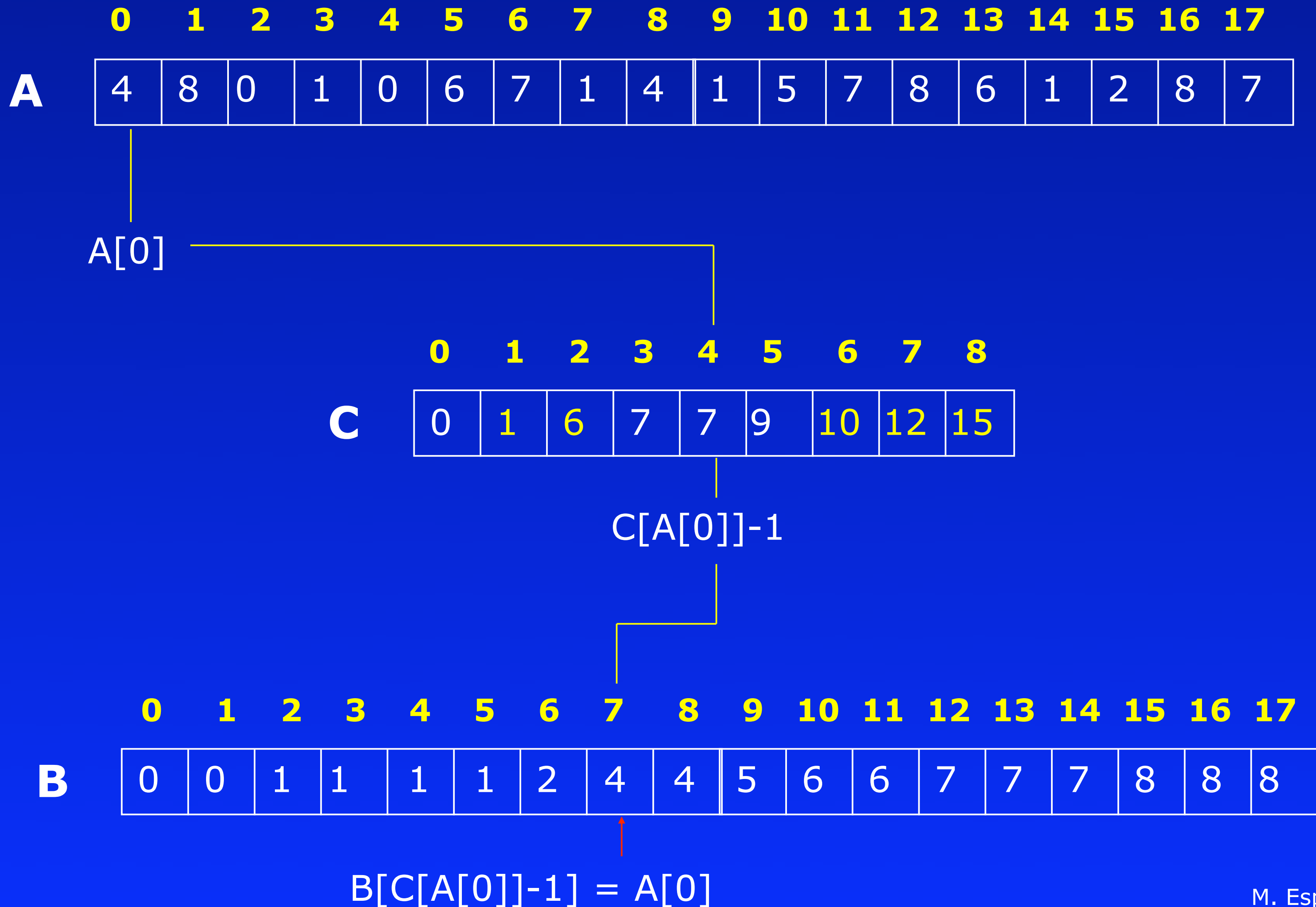
C[A[15]]=C[A[15]]-1

B

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

B[C[A[15]]-1] = A[15]

Counting sort



M. Esponda

Counting sort

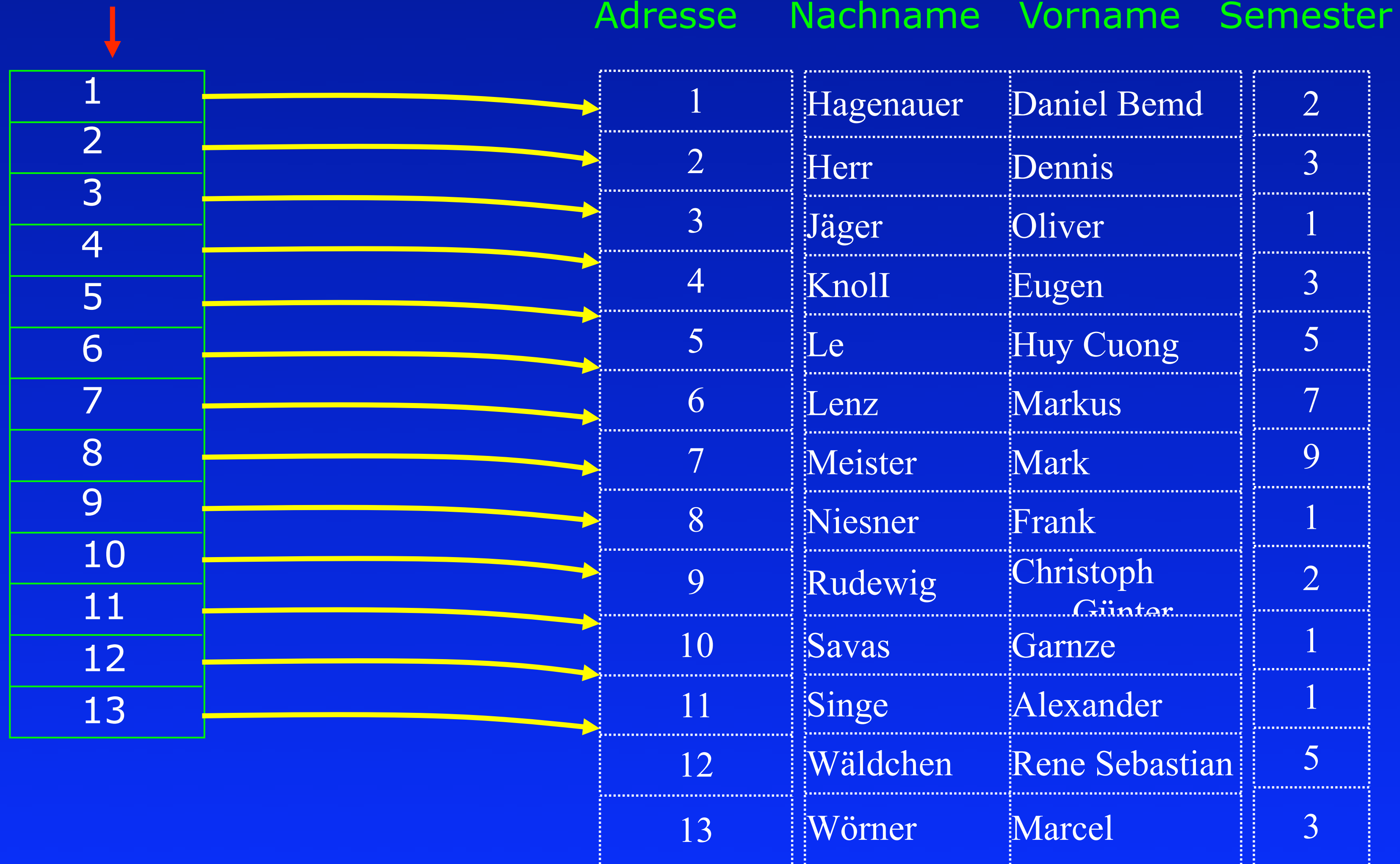
```
def counting_sort(A, k):
        size = len(A)
        B = [0 for i in range(0, size)]
        C = [0 for i in range(0, k+1)]

        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)

Datenbank

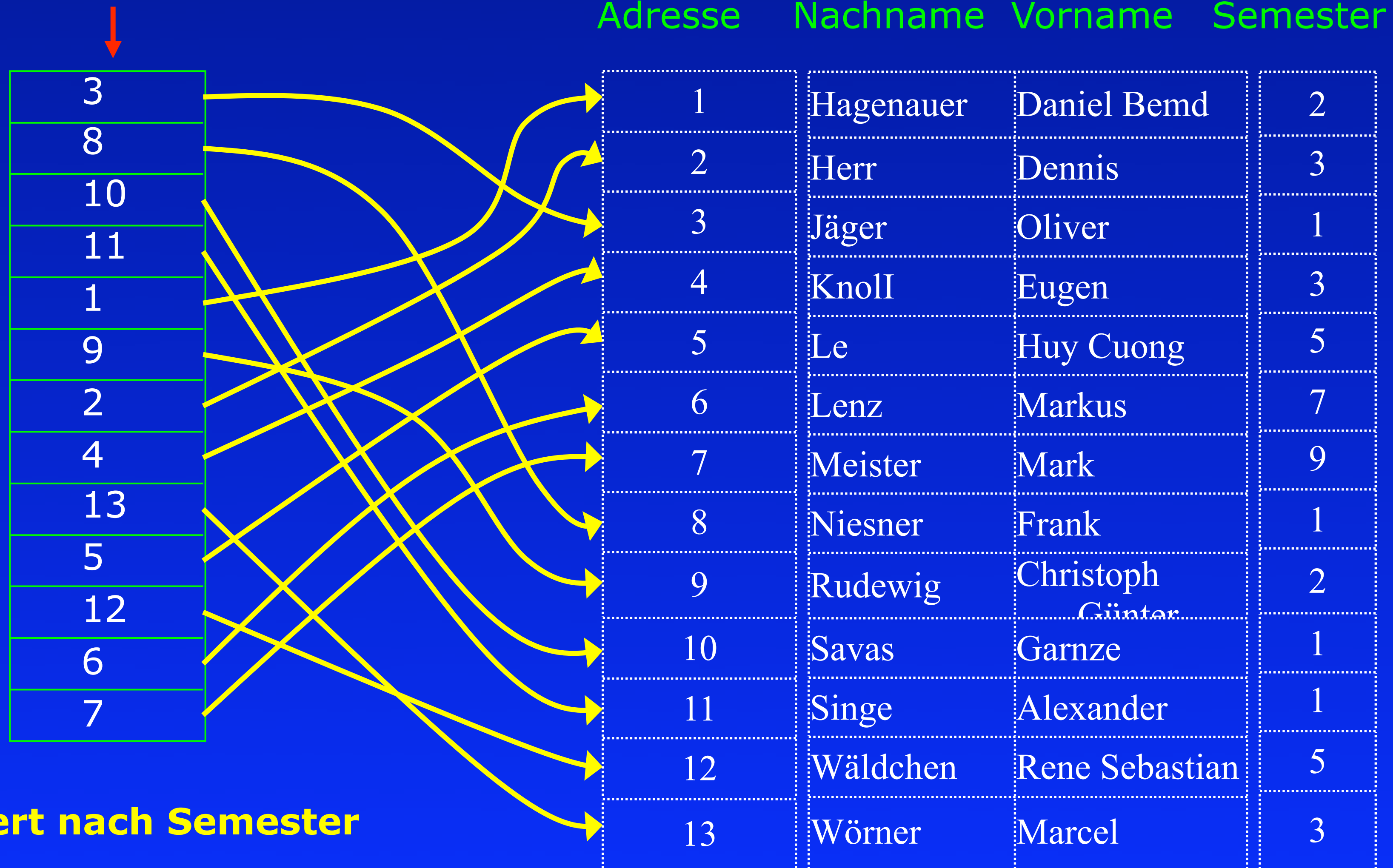


M. Esponda

Counting sort

Im Array sind
nur die Adressen

Datenbank



M. Esponda

Counting sort

Im Array sind
nur die Adressen

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	Christoph Günter	2
10	Savas	Garnze	1
11	Singe	Alexander	1
12	Wäldchen	Rene Sebastian	5
13	Wörner	Marcel	3

Sortiert nach Semester

M. Esponda

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**

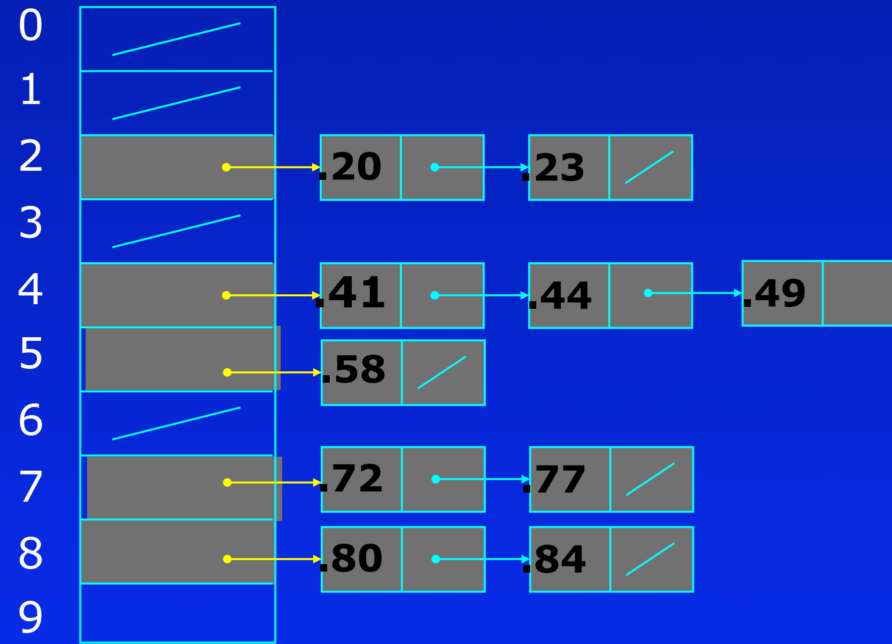
  zusätzlicher Speicherplatz (**O(n)**) wird benötigt
- linearer Aufwand **O(n)**
- Grundidee ist den Wertebereich [0,1) in **m** kleinere Wertebereiche zu teilen und *Buckets* dafür zu definieren.
- Die Zahlen werden in den dazugehörigen *Buckets* verteilt und innerhalb diesen sortiert.
- Zum Schluss 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

B



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
```