

# Sortieren: Heap-Sort

Markus Chimani

Lehrstuhl für Algorithm Engineering, LS11

6. VO

20. April 2006

# Übersicht

Nachtrag Quick-Sort:

- Randomisierter QS
- $O(\log n)$  Speicher

Heap-Sort

# Randomisierter Quick-Sort

## Ziel

Laufzeit unabhängig von der „Qualität“ der Eingabefolge (vorsortiert, etc...)

## Idee

Wähle Pivotelement zufällig aus!

## Einfachste Umsetzung

„Neues“ Partitionieren:

- (1)  $z :=$  Zufallszahl zwischen  $l$  und  $r$
- (2) Vertausche  $A[z]$  mit  $A[r]$
- (3) normales Partitionieren

# Randomisiertes Quick-Sort

Randomisierte Algorithmen:

Es lassen sich keine Best-Case und Worst-Case  
Beispiele mehr konstruieren!

Aber: Natürlich kann der Worst-Case noch eintreten!

Man berechnet eine

Erwartete Laufzeit

= Erwartungswert der Laufzeit  $\approx$  durchschnittliche  
Laufzeit  $\rightarrow$  Average-Case

$$E[ T(n) ] = O(n \log n)$$

# Quick-Sort mit $\log(n)$ Speicher

Bisher:  $O(n)$  Speicher

$O(\log n)$  Speicher?

→ Lange Teile sofort  
bearbeiten



```
(1) procedure QS(ref A,l,r)
(2) if  $l < r$  then {
(3)    $p := \text{Partition}(A,l,r)$ 
(4)   QuickSort(A,l,p-1)
(5)   QuickSort(A,p+1,r)
(6) }
```

Zwischenschritt

```
(1) procedure QS(ref A,l,r)
(2) while  $l < r$  do {
(3)    $p := \text{Partition}(A,l,r)$ 
(4)   QuickSort(A,l,p-1)
(5)    $l := p+1$ 
(6) }
```

# Quick-Sort mit $\log(n)$ Speicher

Rekursiver Aufruf nur für kleinere Teile.

→ diese Teile sind  $\leq$  Hälfte der betrachteten Länge

→ Tiefe der rekursiven Aufrufe:  $O(\log n)$



Zwischenschritt

(1) **procedure** QS(ref A,l,r)

(2) **while**  $l < r$  **do** {

(3)  $p := \text{Partition}(A,l,r)$

(4) QuickSort(A,l,p-1)

(5)  $l := p+1$

(6) }



(1) **procedure** QS(ref A,l,r)

(2) **while**  $l < r$  **do** {

(3)  $p := \text{Partition}(A,l,r)$

(4) **if**  $p-l < r-p$  **then** {

(5) QuickSort(A,l,p-1)

(6)  $l := p+1$

(7) } **else** {

(8) QuickSort(A,p+1,r)

(9)  $r := p-1$

(10) } }

# Was bisher geschah...

Insertion-Sort: einfach 😊,  $O(n^2)$  ⚡

Selection-Sort:  $\Theta(n^2)$  ⚡

Merge-Sort:  $O(n \log n)$  😊, nicht in-situ ⚡

Quick-Sort: praxis schnell 😊,  $O(n^2)$  ⚡

jetzt:

**Heap-Sort:**  $O(n \log n)$  😊, in-situ 😊

Auf Basis von Selection-Sort!

# Selection-Sort...

## GUT

In jedem Schritt wird ein Schlüssel an seine richtige Position gesetzt

## SCHLECHT

Suchen des richtigen Schlüssels:  $\Theta(n)$

## IDEE

Den unsortierten Teil „vorsortieren“ →  
Schnelles Finden des nächsten Schlüssels

# Heap (=Haufen)

Endlich wieder eine neue Datenstruktur!

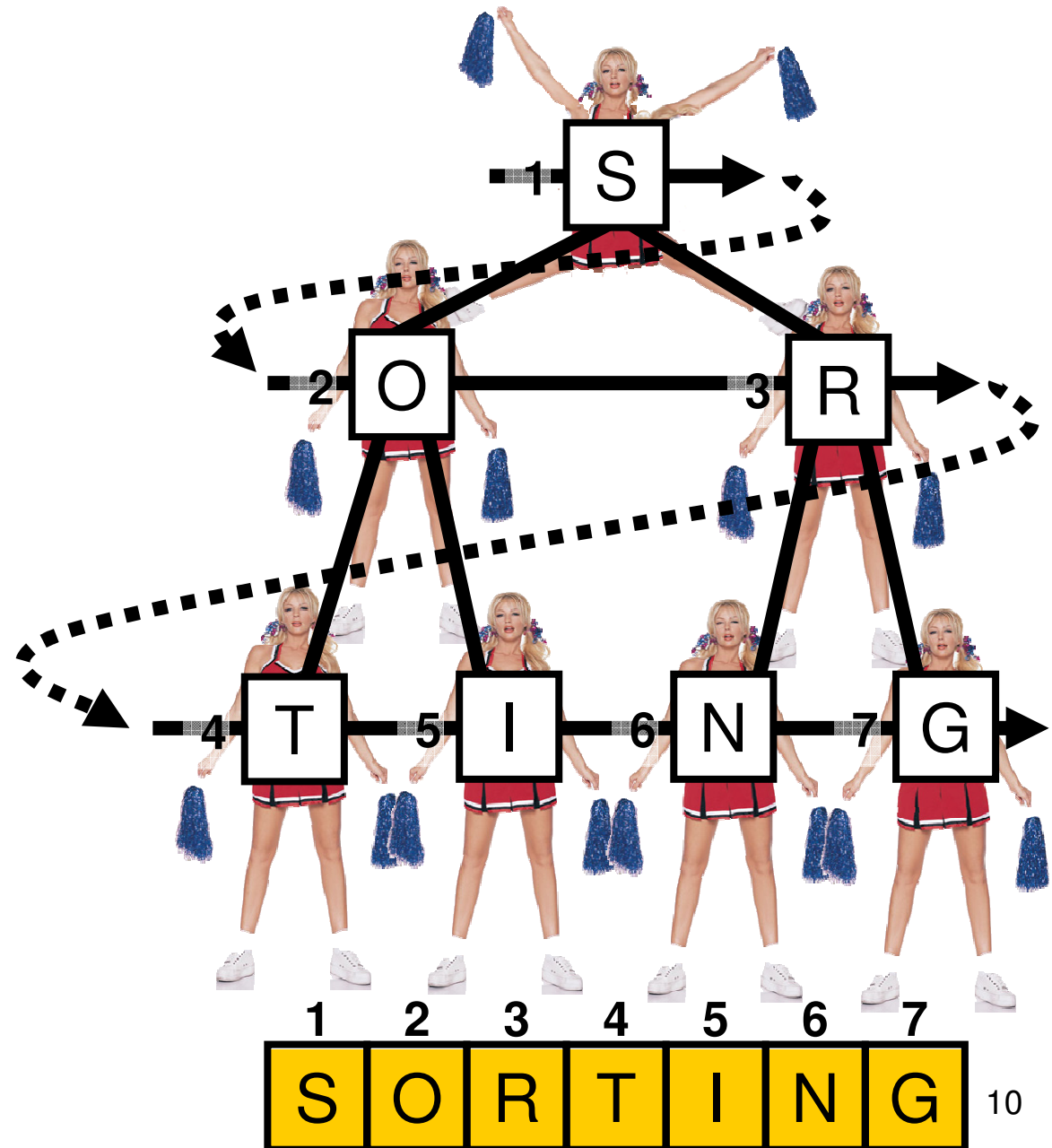
Speicherung...

...weiterhin als Array (in-situ!)

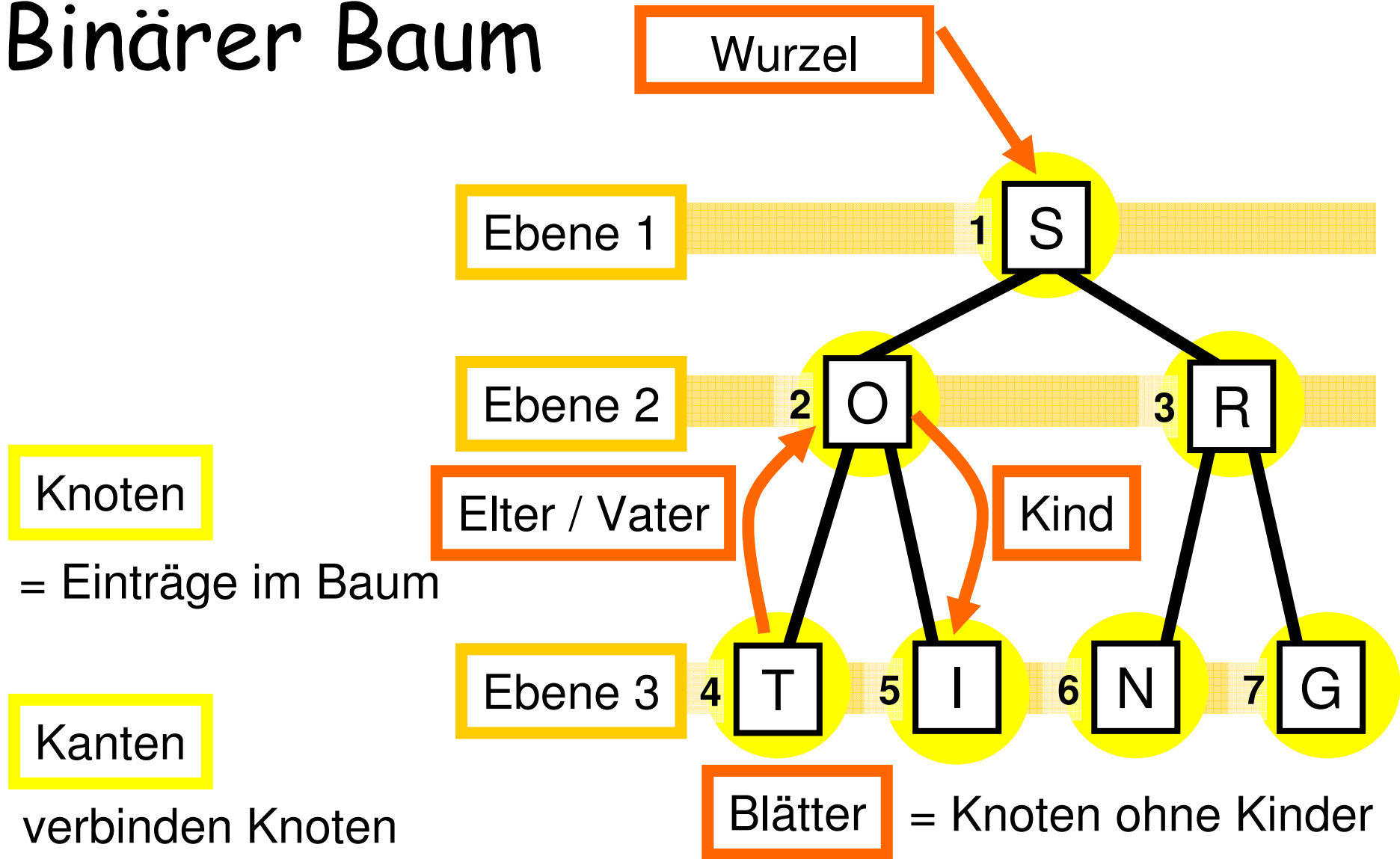
Interpretation...

...als binärer Baum,  
an dessen Wurzel immer das größte  
Element stehen soll

# Heap: Interpretation

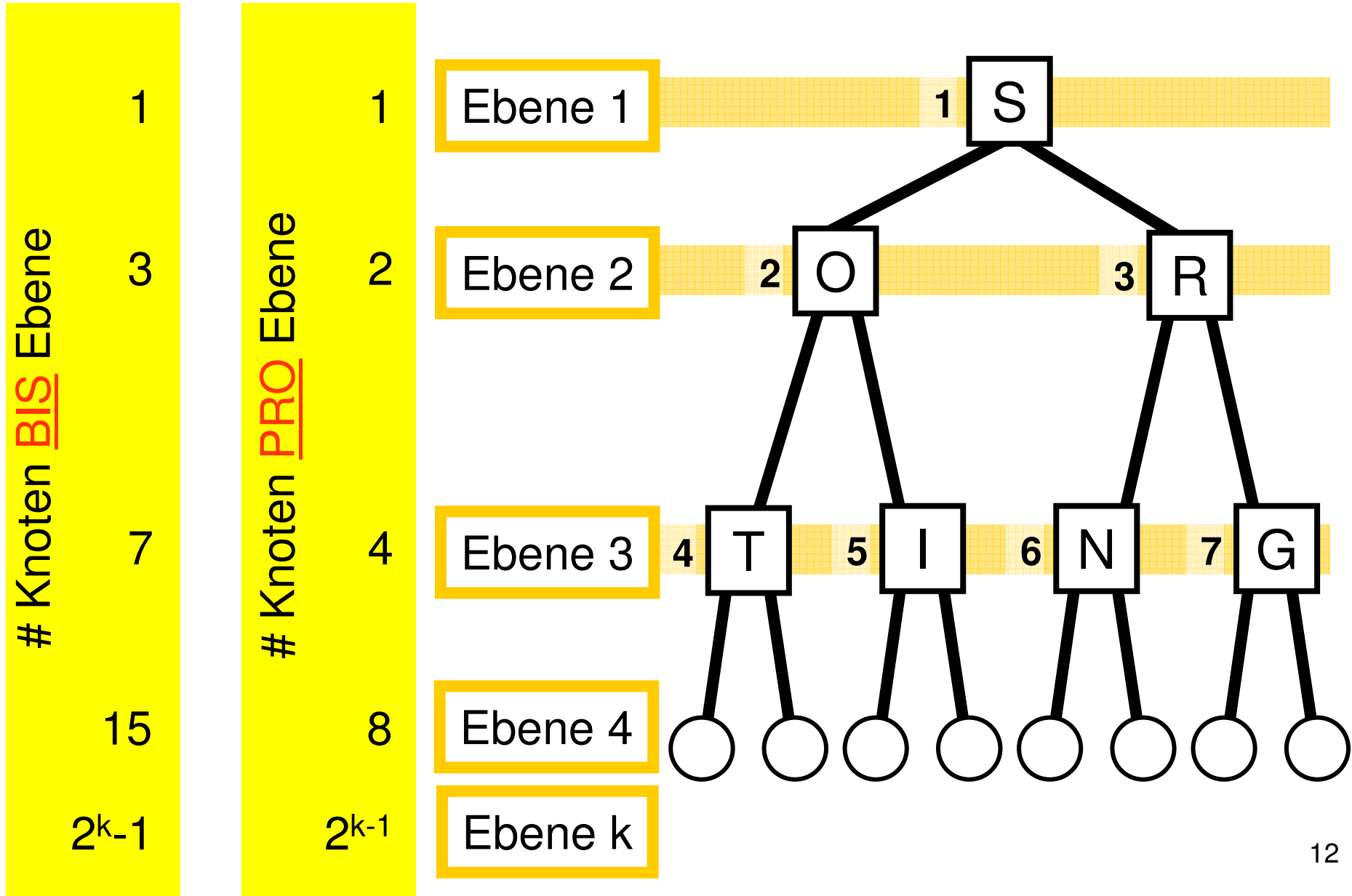


# Binärer Baum

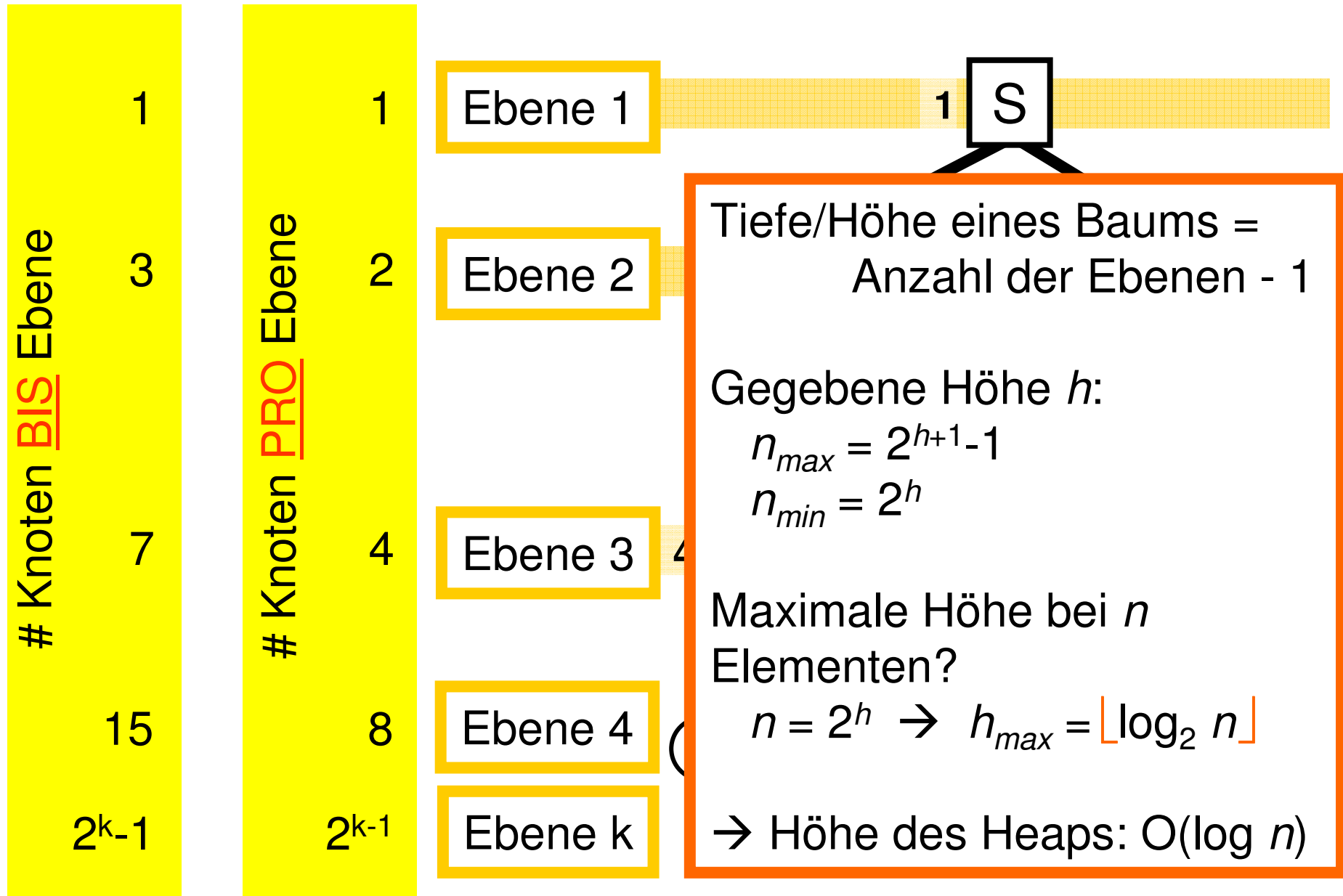


Binärer Baum: maximal 2 Kinder pro Knoten

# Tiefe/Höhe des Baums



# Tiefe/Höhe des Baums



# Heap-Eigenschaft

MaxHeap =  
 Jeder Vaterknoten ist  
 größer als seine Kinder

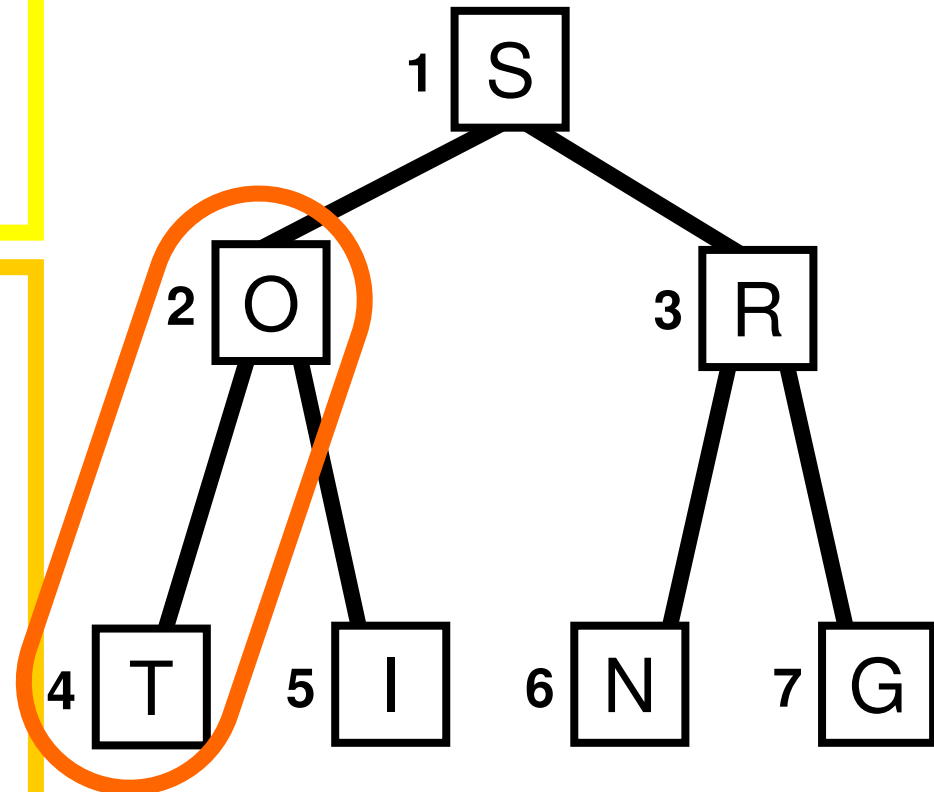
Array  $A[1\dots n]$

$\forall$  Indizes  $i$ :

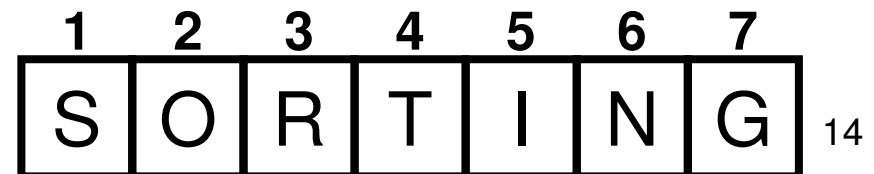
$A[i] \geq A[2i]$  und

$A[i] \geq A[2i+1]$

falls diese Indizes  
 existieren



nicht erfüllt!



# Also was ist zu tun?

Elemente im unsortierte Array so verschieben, dass Heap-Eigenschaft erfüllt ist (*CreateHeap*)

Größtes Element finden ist einfach →  
Wurzel = erstes Element im Array

Wurzel aus Heap entfernen →  
Heap-Eigenschaften wiederherstellen  
(*Sifting*)

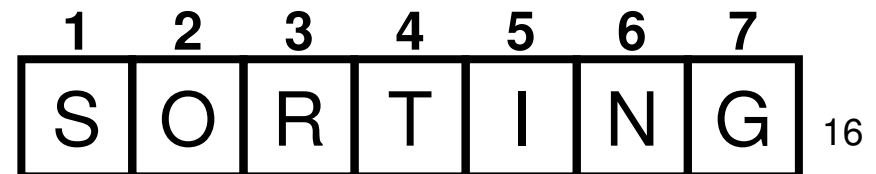
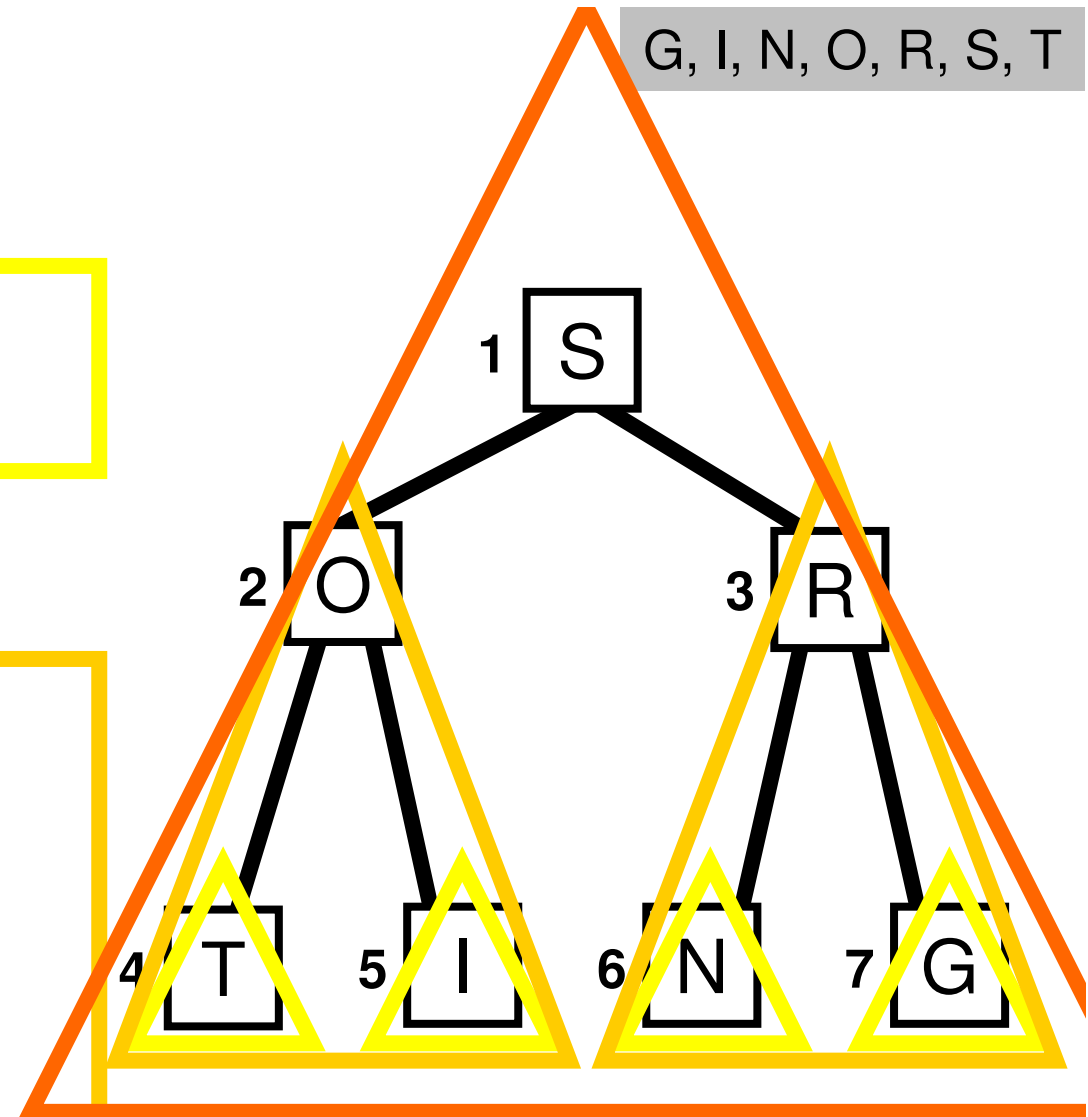
# CreateHeap

## Beobachtung

Die Heapbedingung ist an den Blättern erfüllt.

## Algorithmus

Betrachte Knoten über den Blättern → repariere Teilheap  
Bearbeite Baum von unten nach oben → Teilbäume sind immer Heaps



# CreateHeap

```

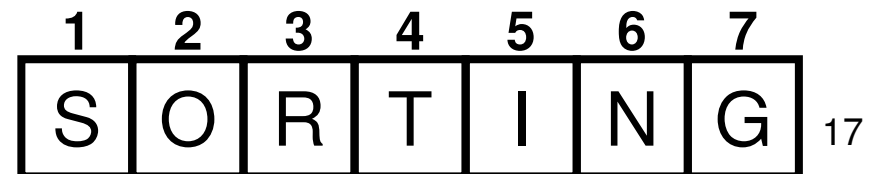
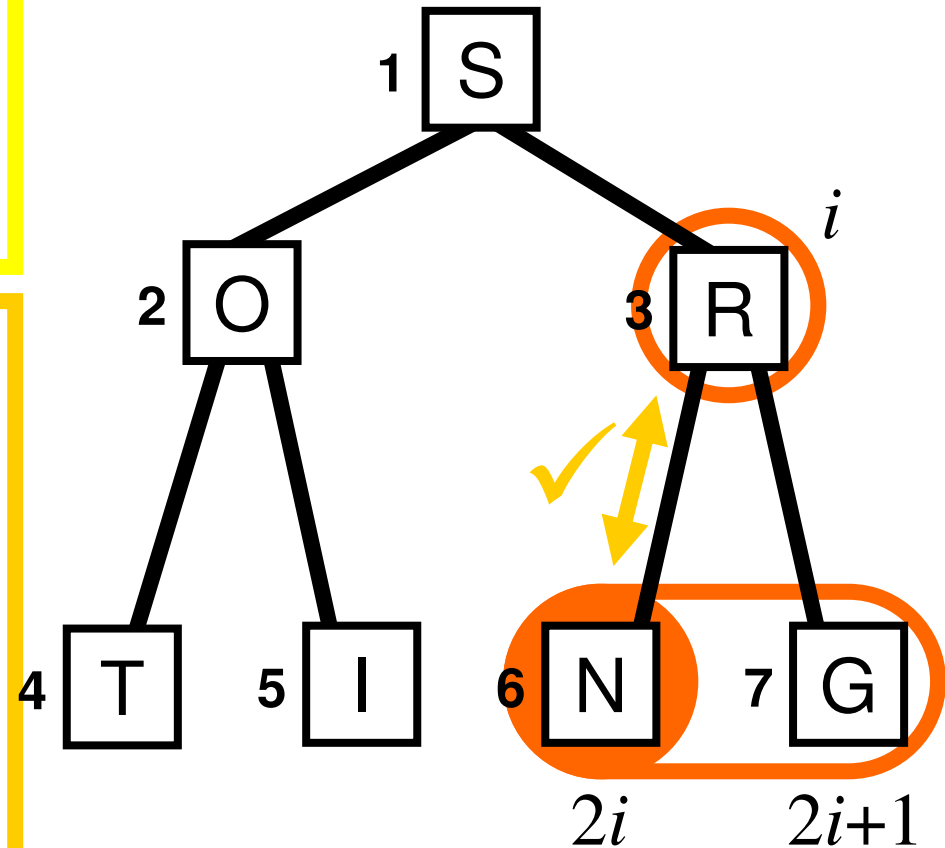
procedure CREATEHEAP () {
  for  $i := \lfloor n/2 \rfloor \dots 1$  {
    SIFTDOWN ( $i, n$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# CreateHeap

```

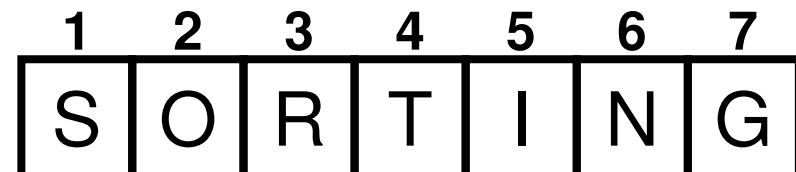
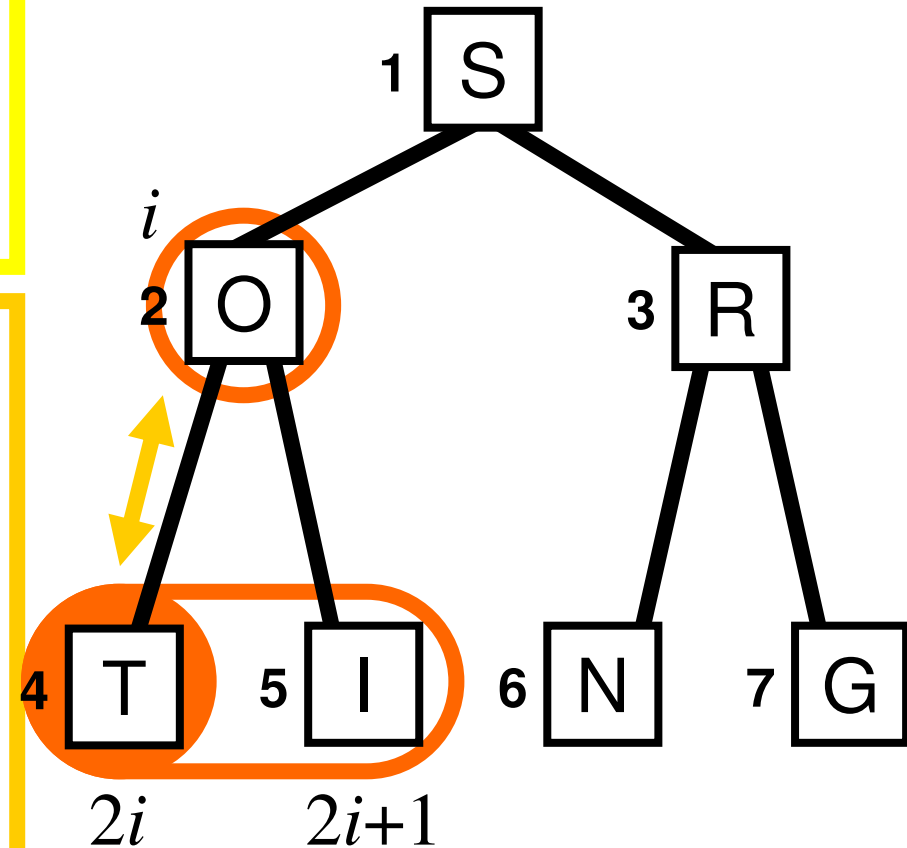
procedure CREATEHEAP () {
  for  $i := \lfloor n/2 \rfloor \dots 1$  {
    SIFTDOWN ( $i, n$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# CreateHeap

```

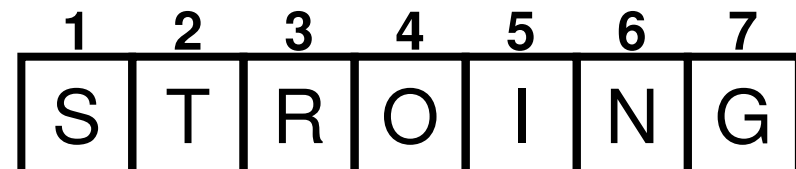
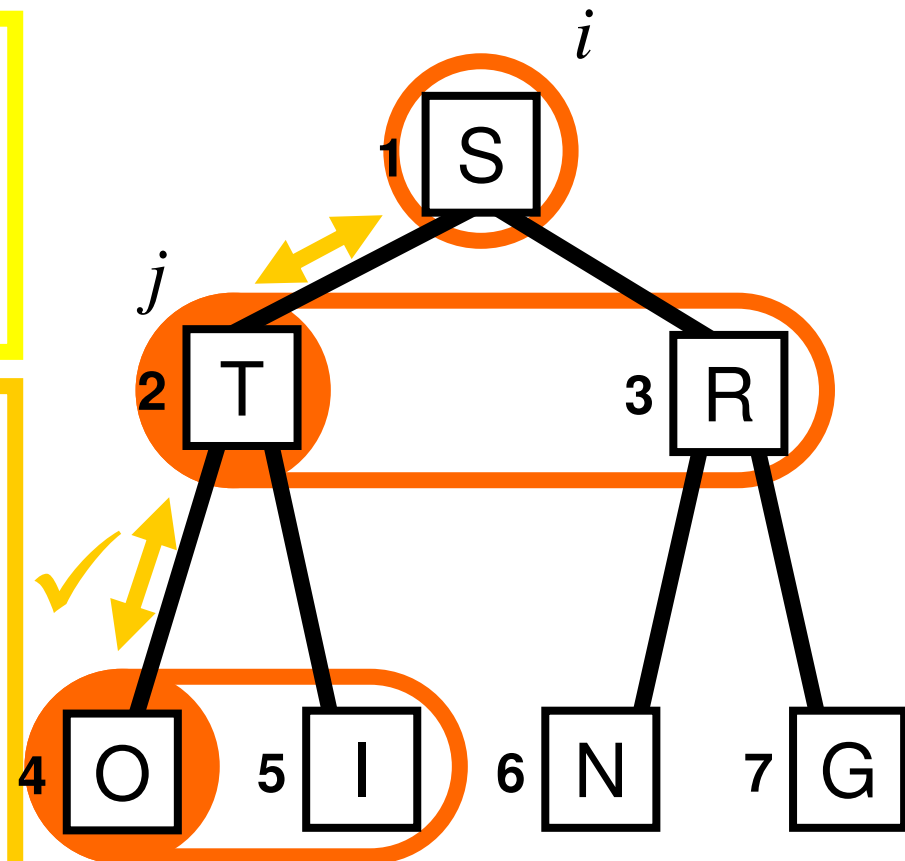
procedure CREATEHEAP () {
  for  $i := \lfloor n/2 \rfloor \dots 1$  {
    SIFTDOWN ( $i, n$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# HeapSort

```

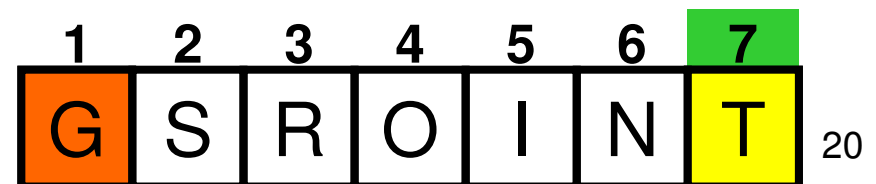
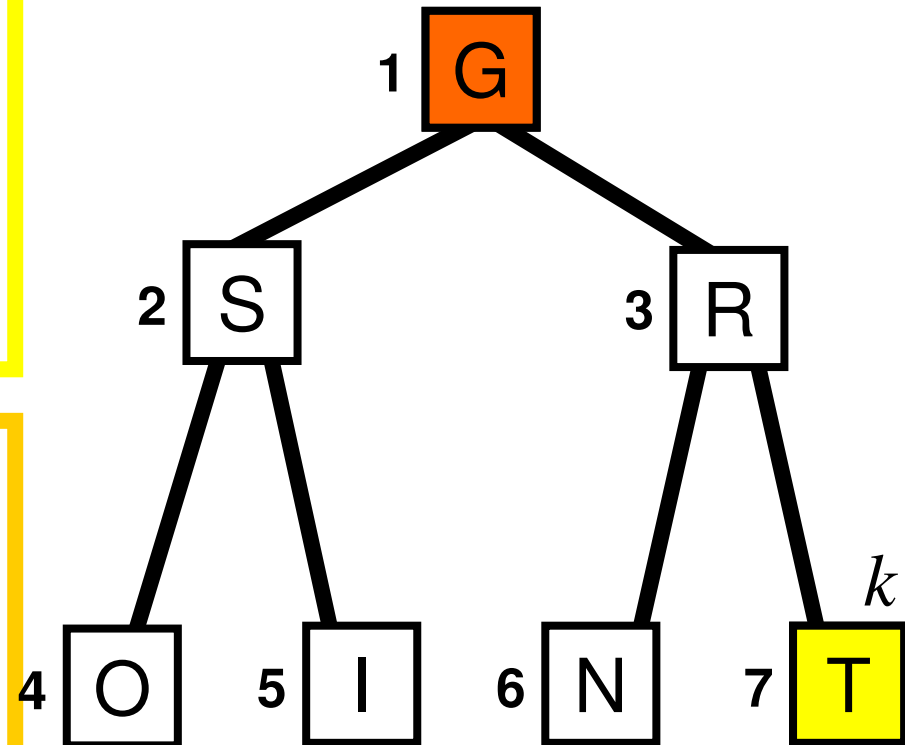
procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    SIFTDOWN (1,  $k-1$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# HeapSort

```

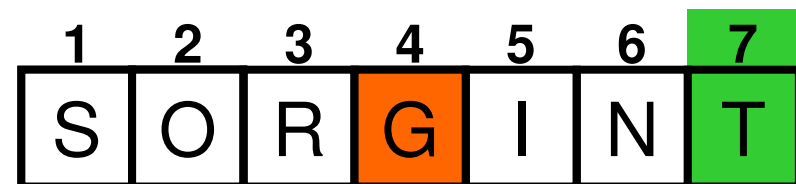
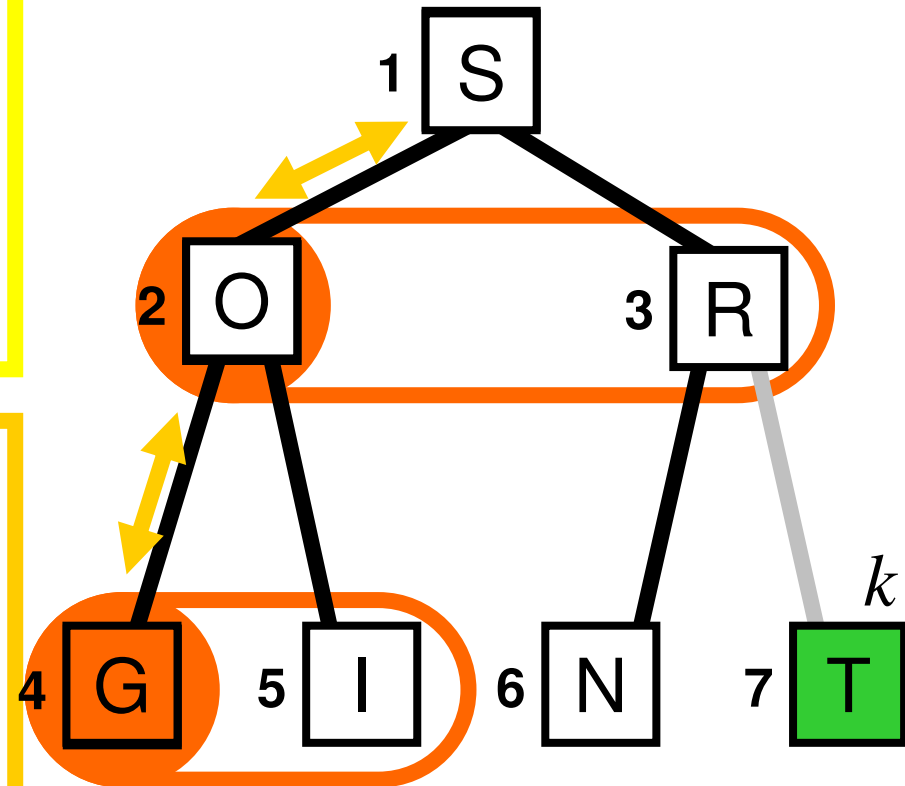
procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    SIFTDOWN (1,  $k-1$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# HeapSort

```

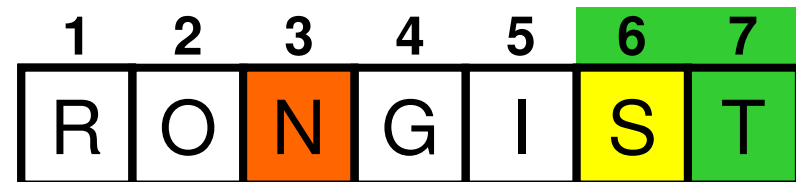
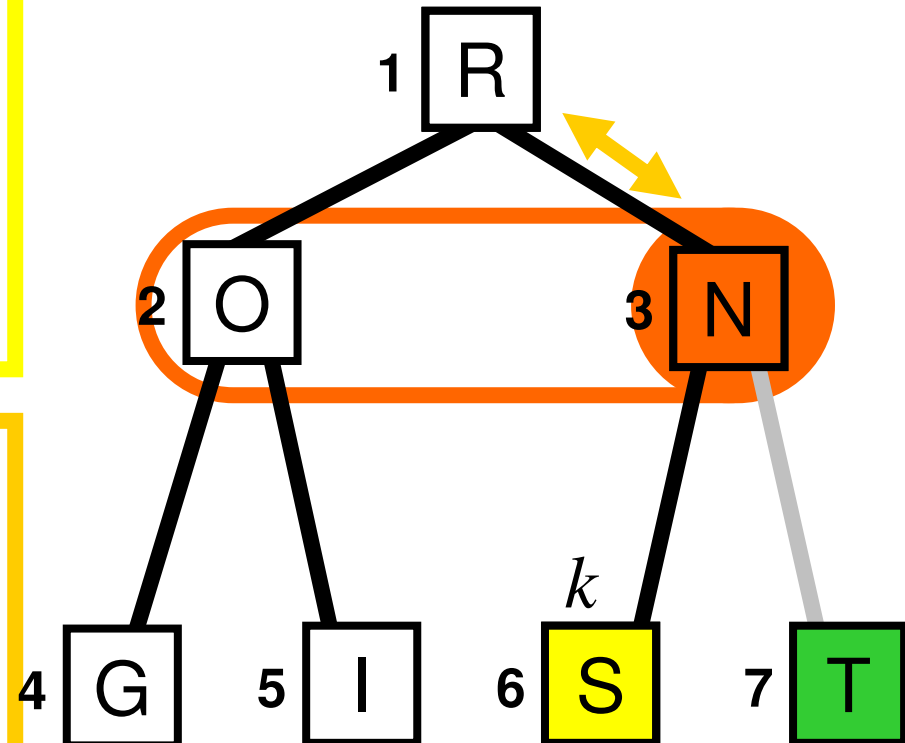
procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    SIFTDOWN (1,  $k-1$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# HeapSort

```

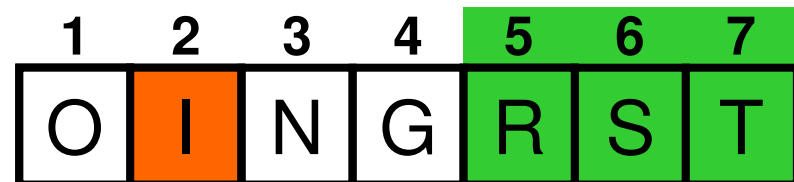
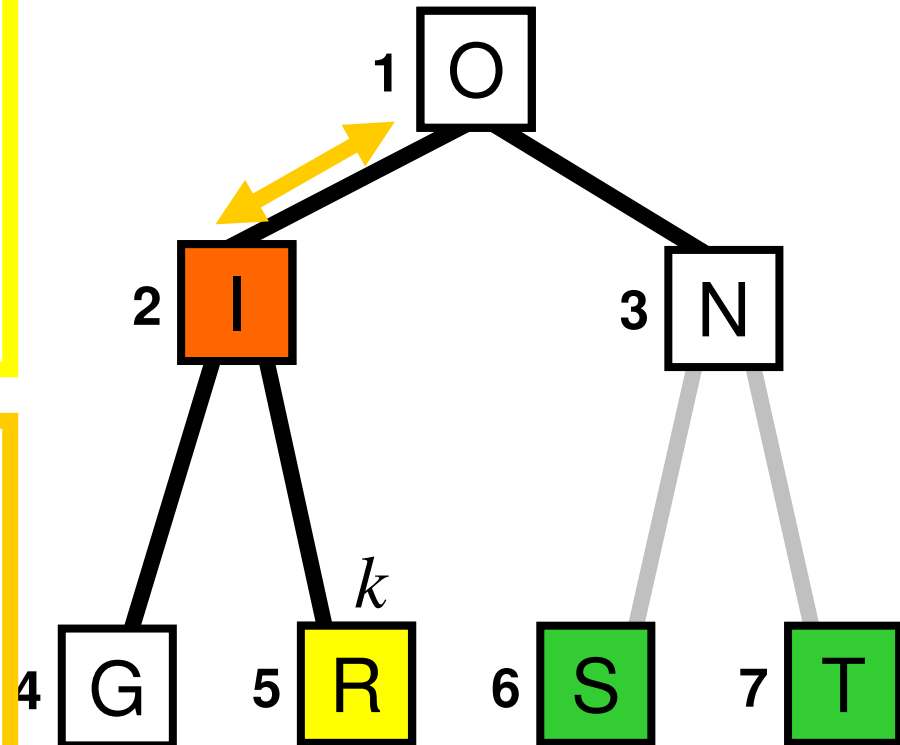
procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    RSIFTDOWN (1,  $k-1$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



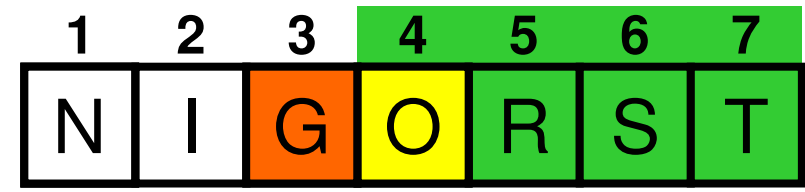
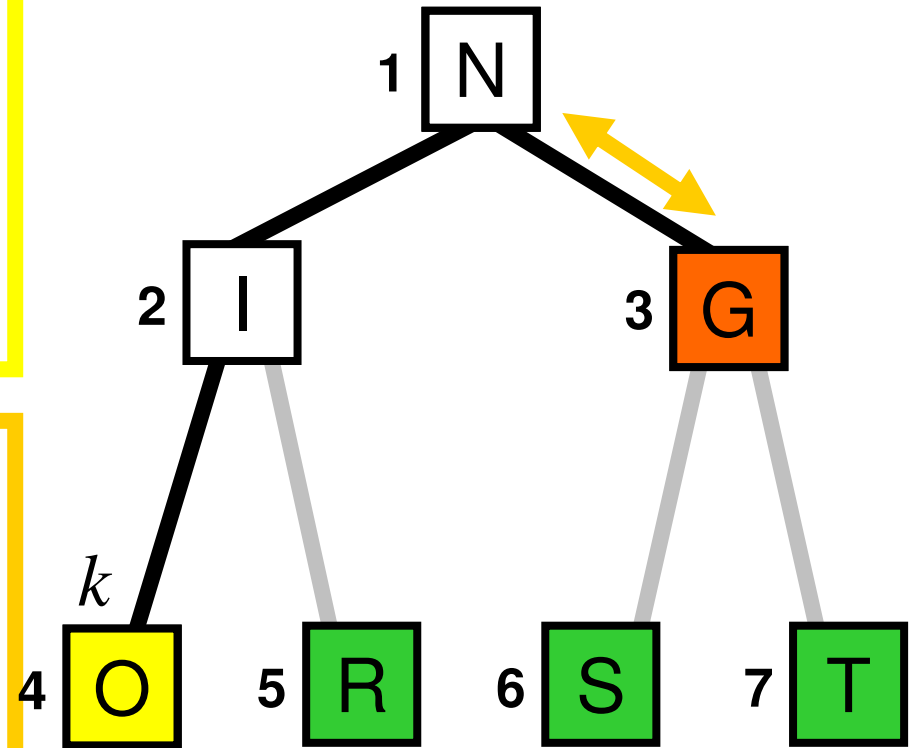
# HeapSort

```

procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    SIFTDOWN (1,  $k-1$ )
  }
}
    
```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}
    
```



# HeapSort

```

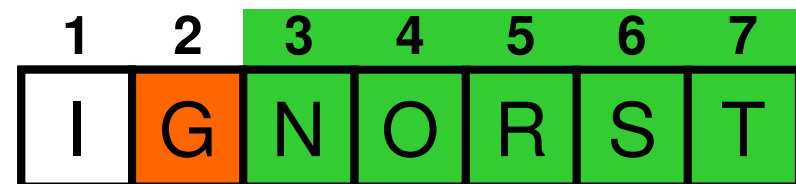
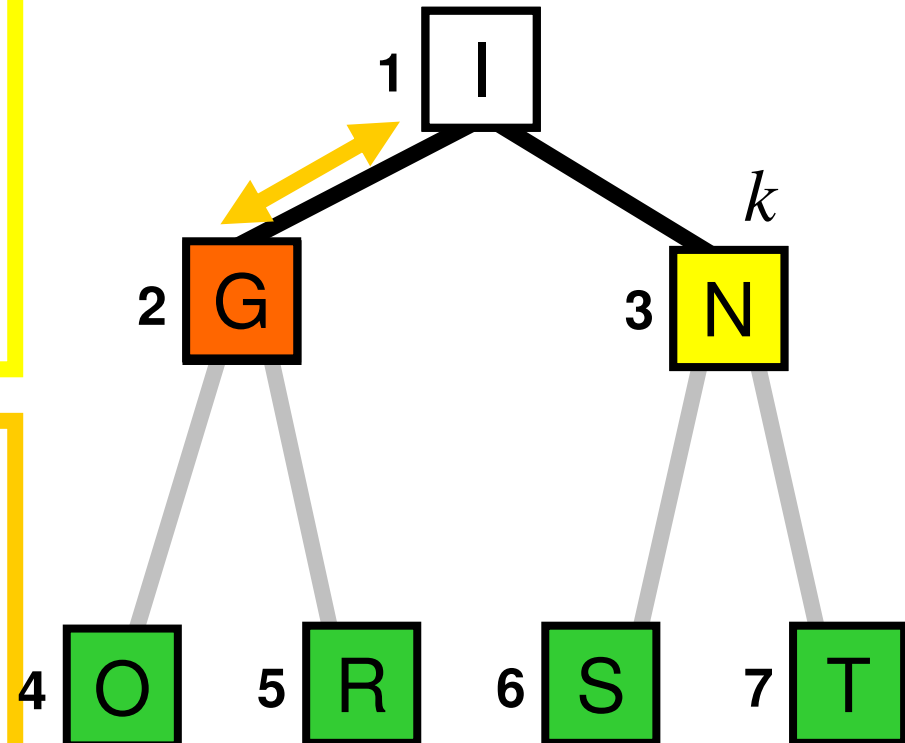
procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    SIFTDOWN (1,  $k-1$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

```



# HeapSort

```

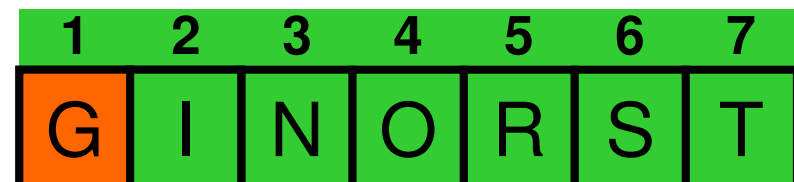
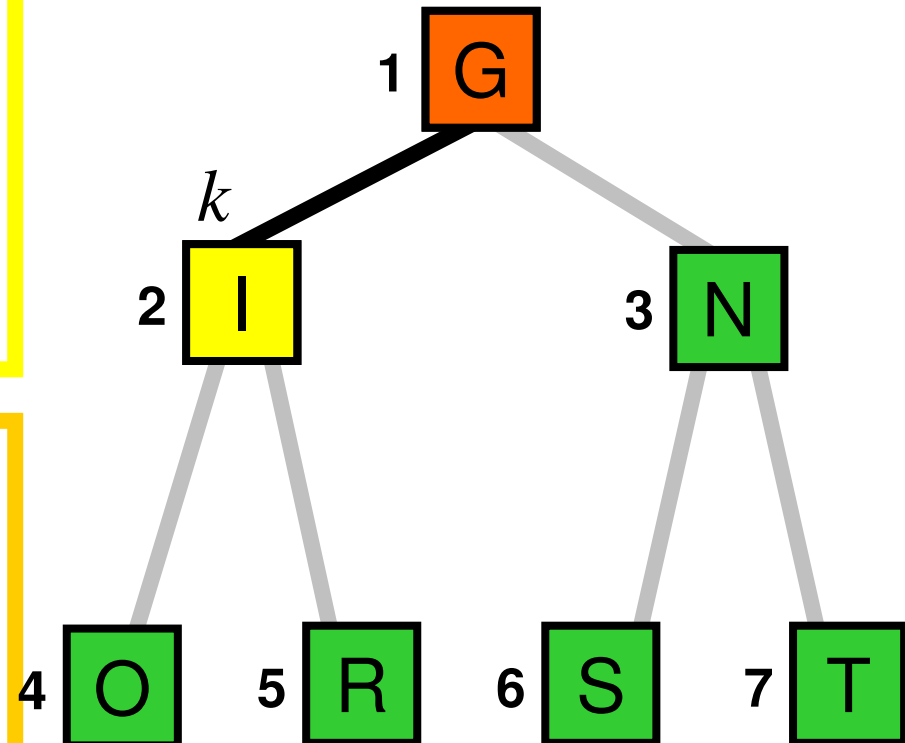
procedure HEAPSORT () {
  CREATEHEAP ()
  for  $k := n \dots 2$  {
    vertausche  $A[1]$  und  $A[k]$ 
    RSIFTDOWN (1,  $k-1$ )
  }
}

```

```

procedure SIFTDOWN ( $i, m$ ) {
  while Knoten  $i$  hat Kinder  $\leq m$  {
     $j :=$  Kind  $\leq m$  mit größerem Wert
    if  $A[i] < A[j]$  then {
      vertausche  $A[i]$  und  $A[j]$ 
       $i := j$ 
    } else return
  }
}

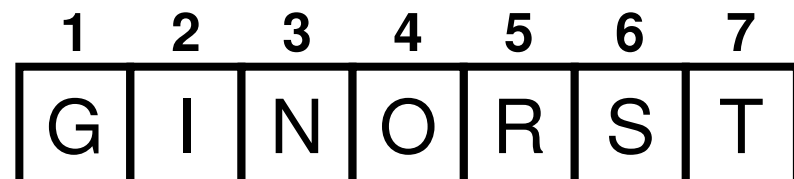
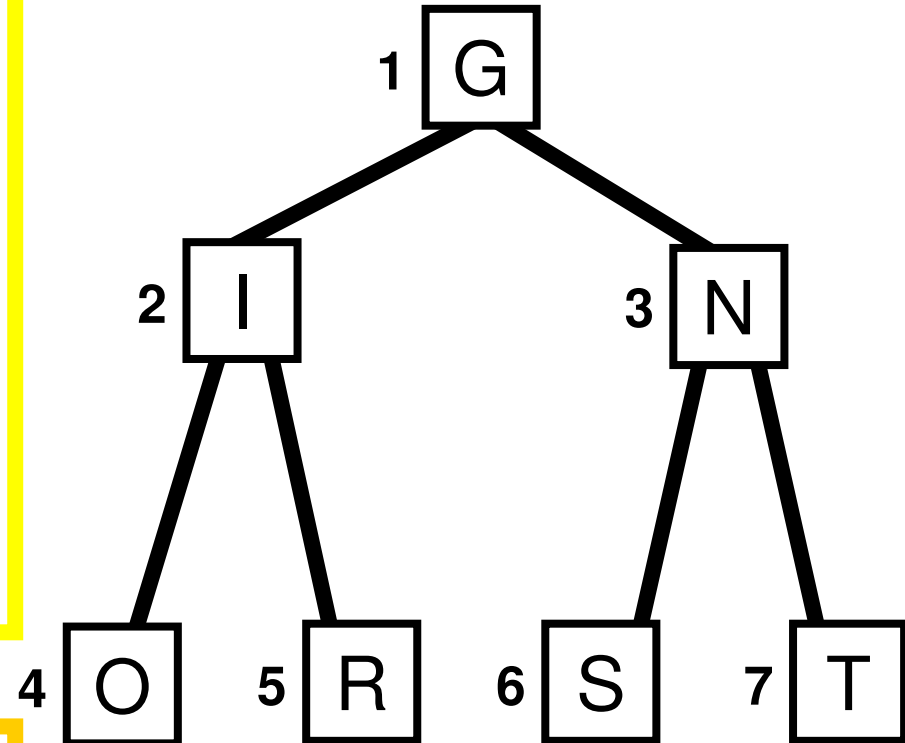
```



# Analyse SiftDown

```
procedure SIFTDOWN (i, m) {  
  while Knoten i hat Kinder  $\leq m$  {  
    j := Kind  $\leq m$  mit größerem Wert  
    if  $A[i] < A[j]$  then {  
      vertausche  $A[i]$  und  $A[j]$   
      i := j  
    } else return  
  }  
}
```

$O(\text{Baumtiefe}) = O(\log n)$



# Analyse CreateHeap

```
procedure CREATEHEAP () {  
  for  $i := \lfloor n/2 \rfloor \dots 1$  {  
    SIFTDOWN ( $i, n$ )  
  }  
}
```

Schleife:

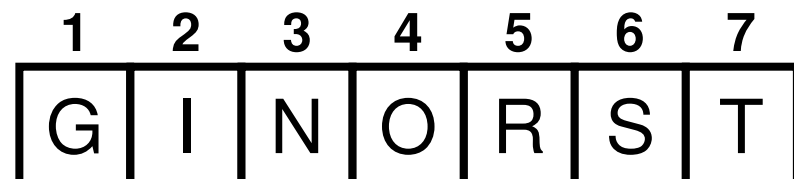
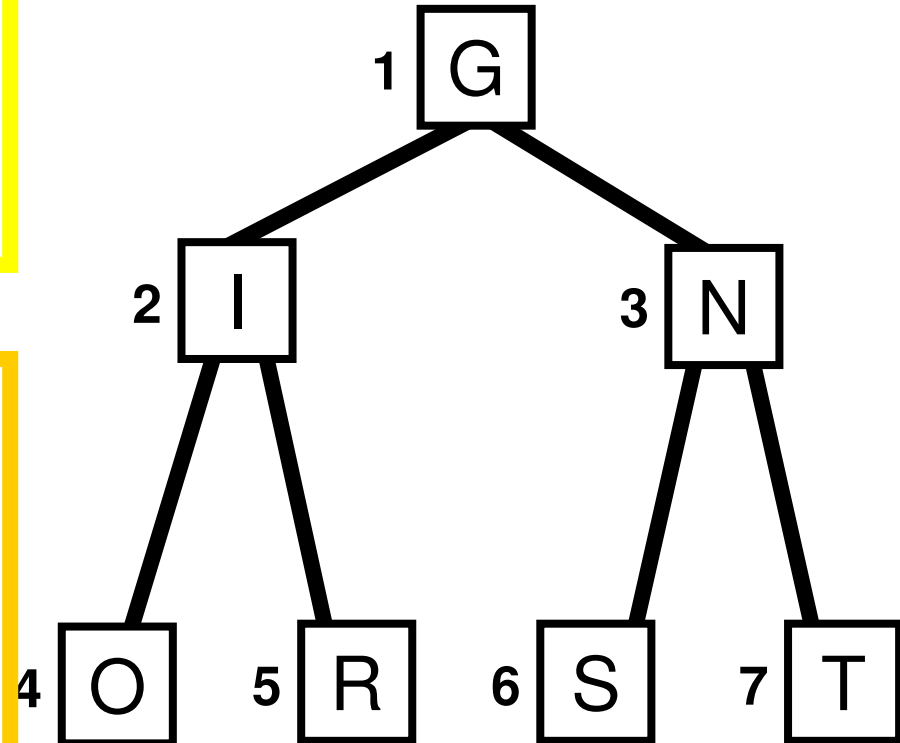
$O(n)$  mal

Pro Schleife:

$O(\log n)$

$\rightarrow O(n \log n)$

Es gilt sogar:  $O(n)$



# Analyse HeapSort

```
procedure HEAPSORT () {  
  CREATEHEAP()  
  for  $k := n \dots 2$  {  
    vertausche  $A[1]$  und  $A[k]$   
    SIFTDOWN (1,  $k-1$ )  
  } }  
}
```

CreateHeap:

$O(n \log n)$

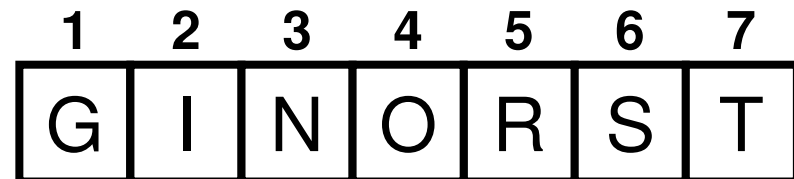
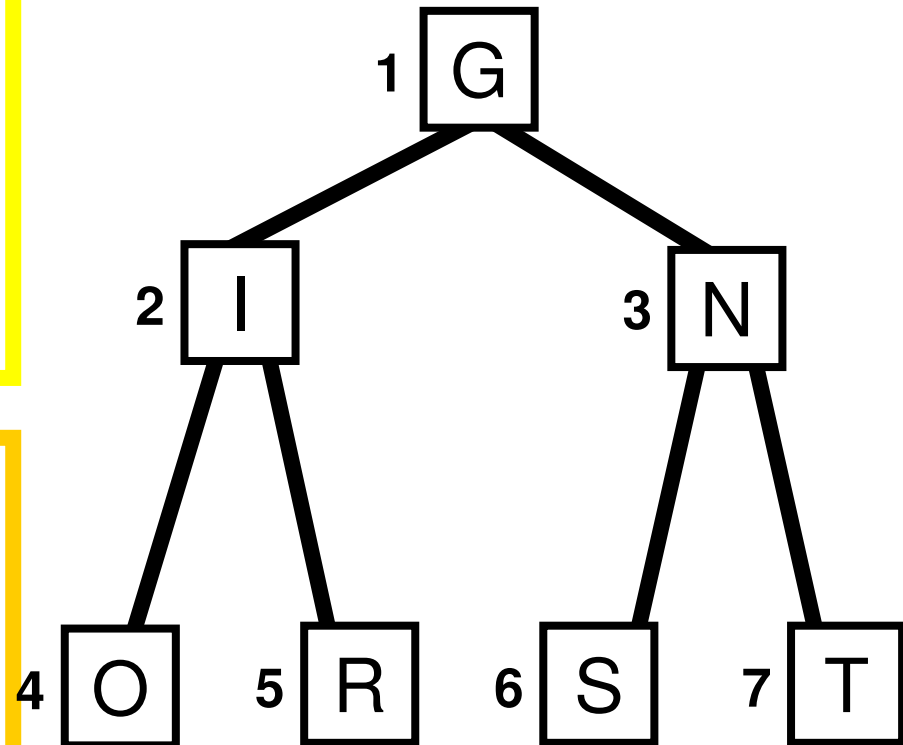
Schleife:

$O(n)$  mal

Pro Schleife:

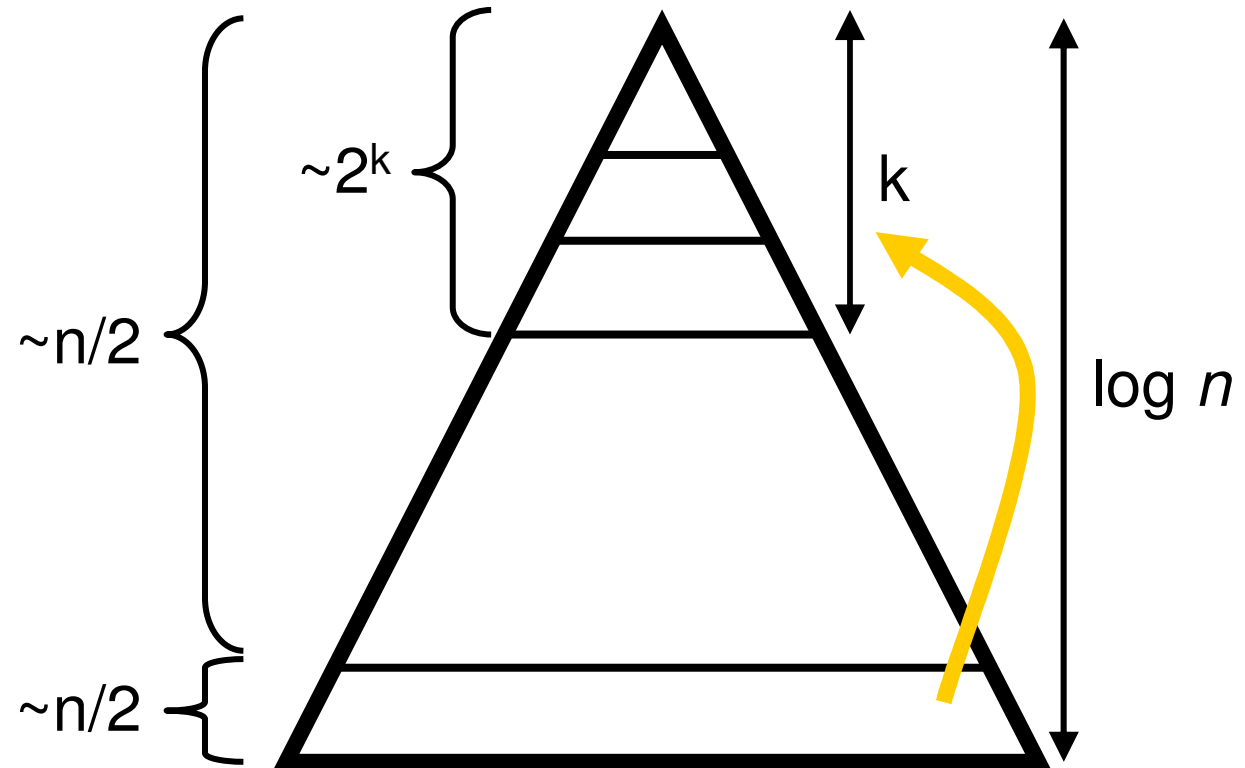
$O(\log n)$

→  $O(n \log n)$



# Best Case?

$$\Omega(n \log n)$$






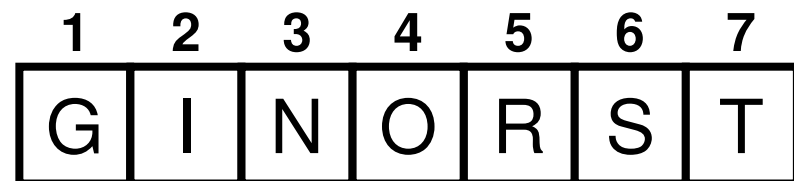
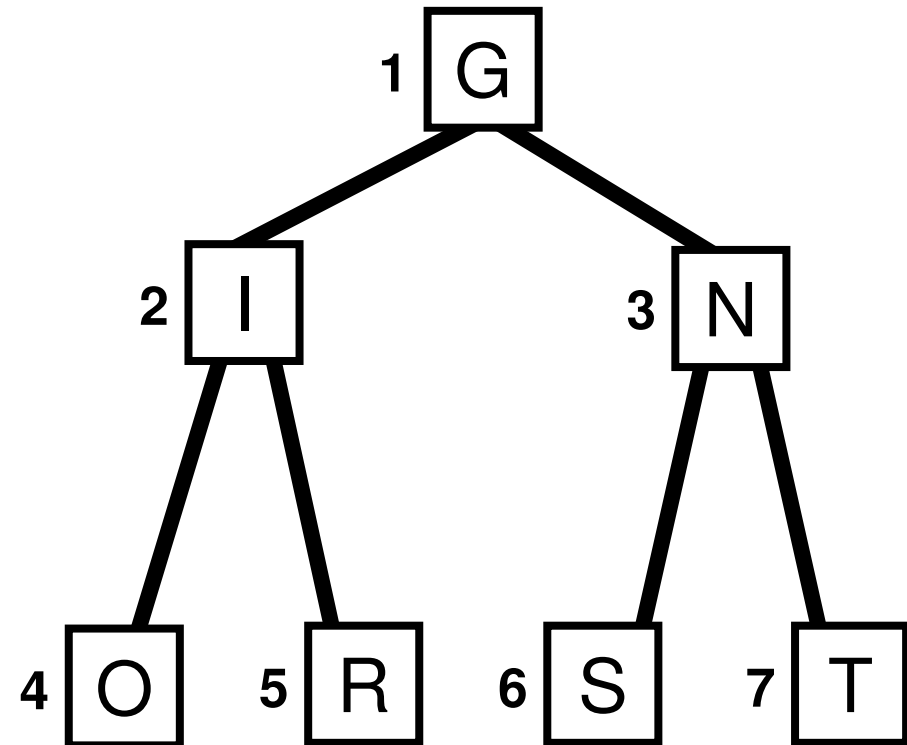
## Intuition

Annahme: linear

- SiftDown nur konstant ( $k$ ) viele Schritte
- zu wenig Platz für alle Blätter! ⚡

# Eigenschaften

in-situ?	
stabil?	
adaptiv?	

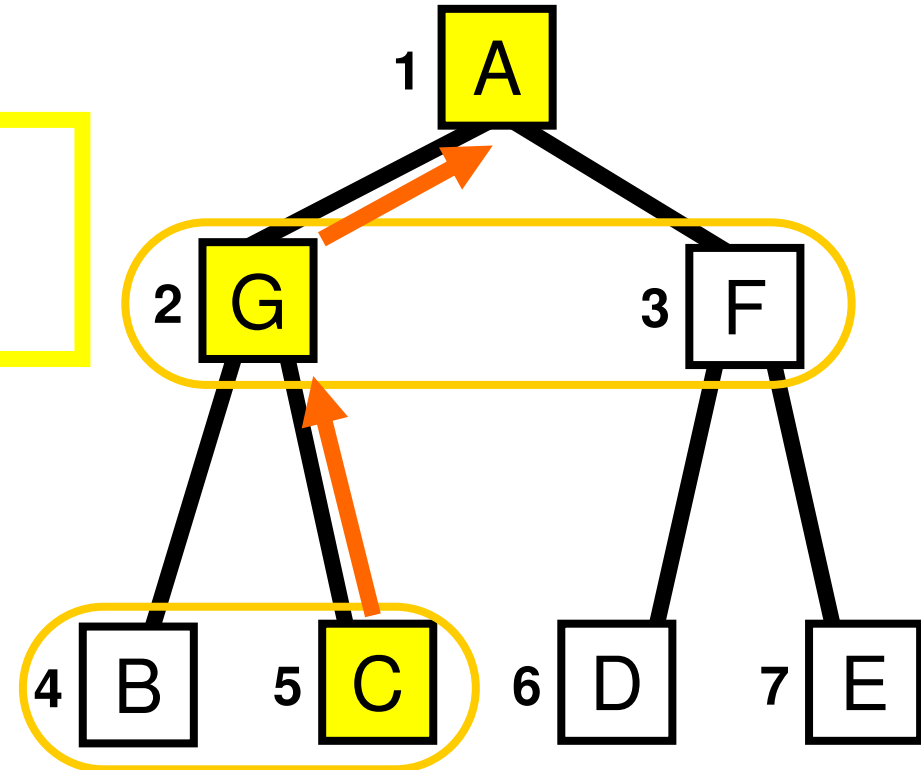


# Alternative Sift-Methoden

## SiftDown (bisher)

Versickern von Wurzel bis zu Blatt (2 Vergl./Ebene)

u.v.m.



## SiftUp

Pfad von Wurzel bis zu Blatt (1 Vergl./Ebene),  
„Aufwärts sickern“ von Blatt zu Wurzel (1 Vergl./Ebene)

Ende

