Sortierverfahren Heapsort

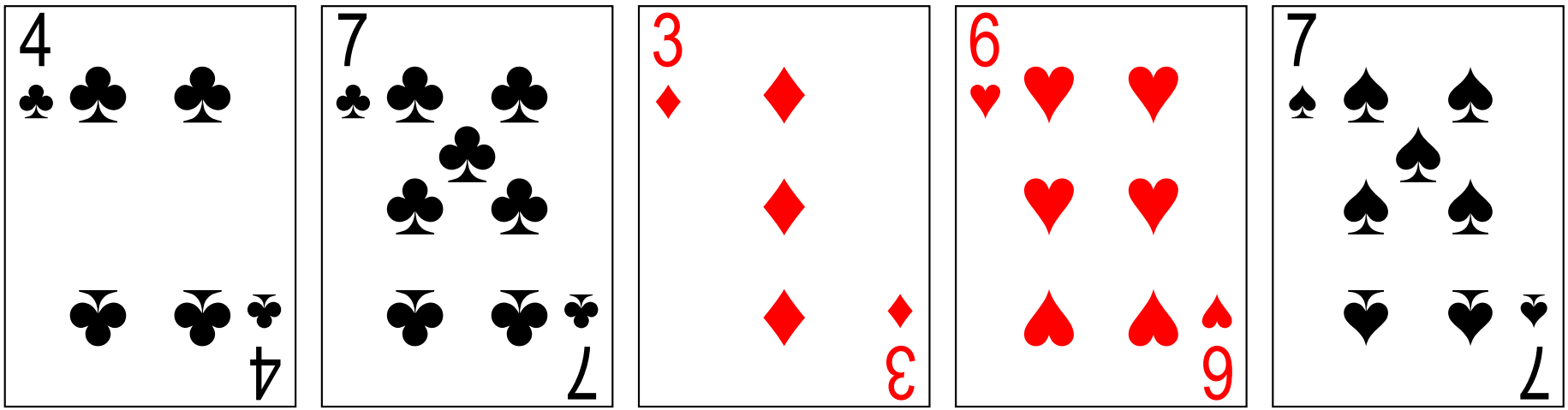
Dr.-Ing. Carsten Schmidt

22. März 2016

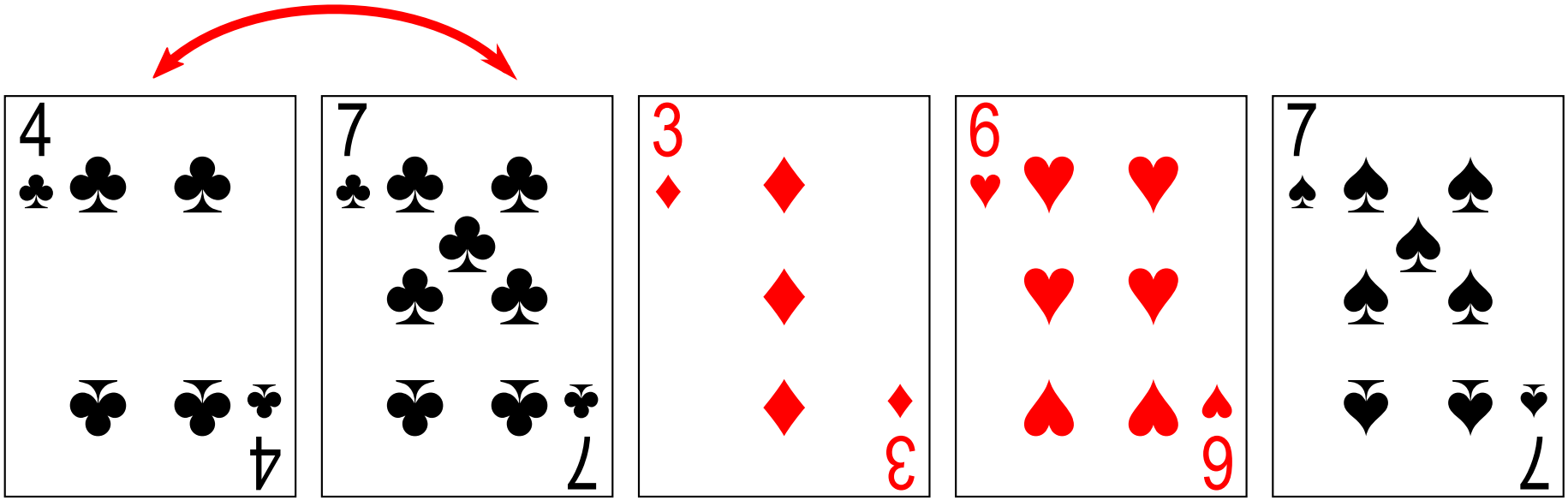
Inhalt

Inhalt

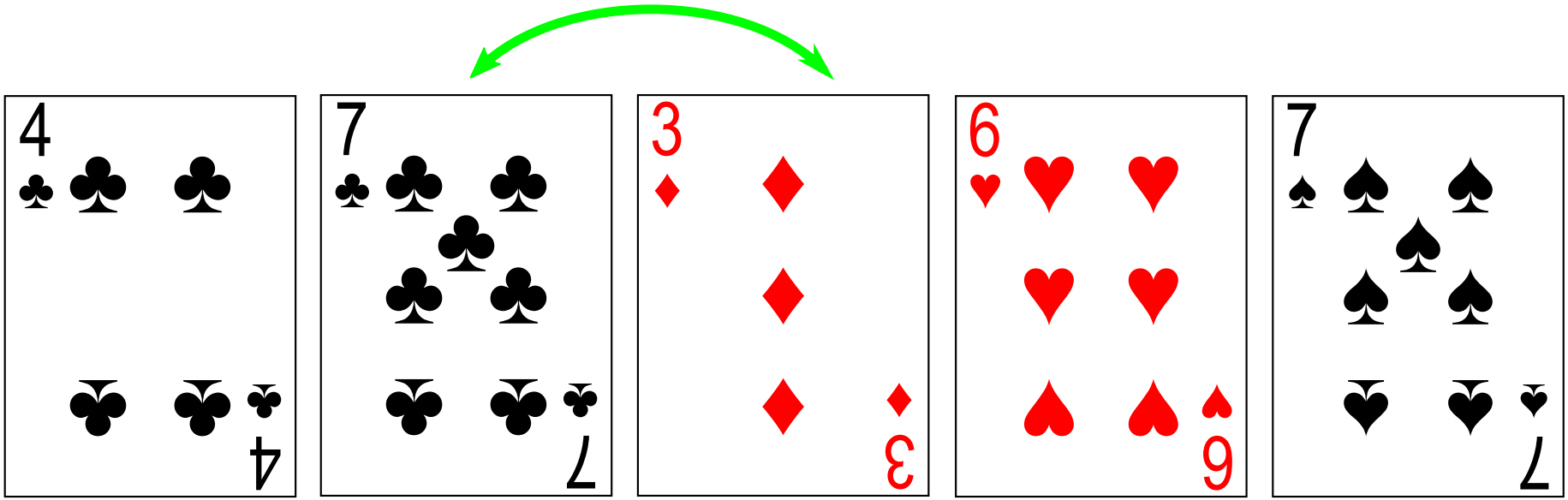
Beispiel: Manuelles Sortieren



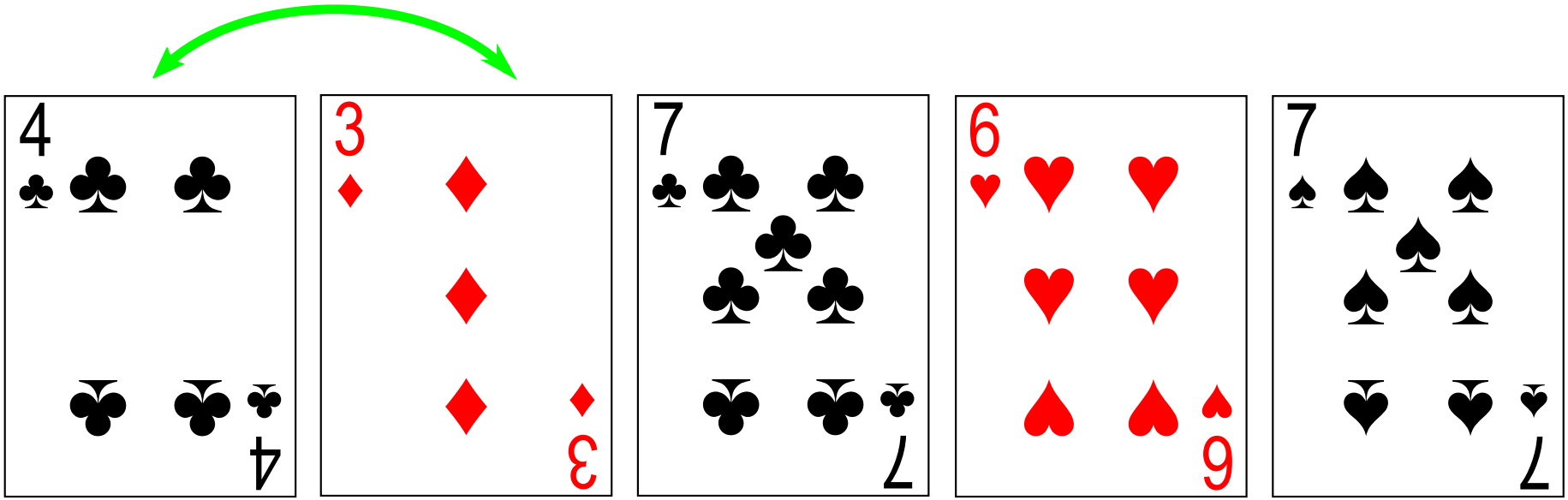
Beispiel: Manuelles Sortieren



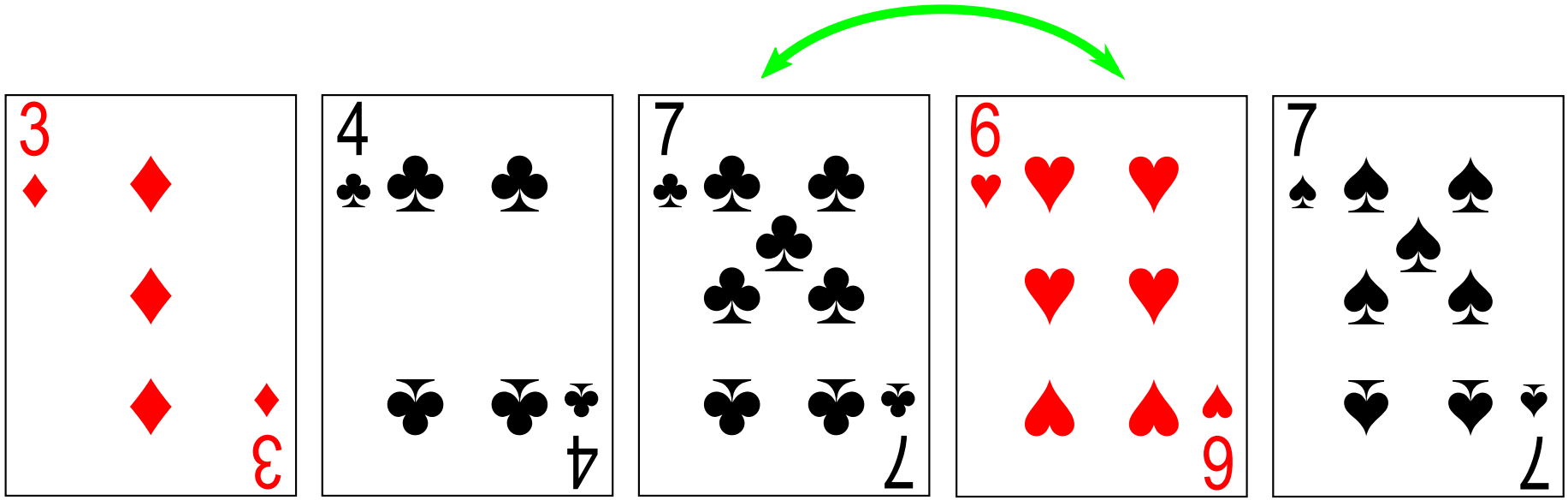
Beispiel: Manuelles Sortieren



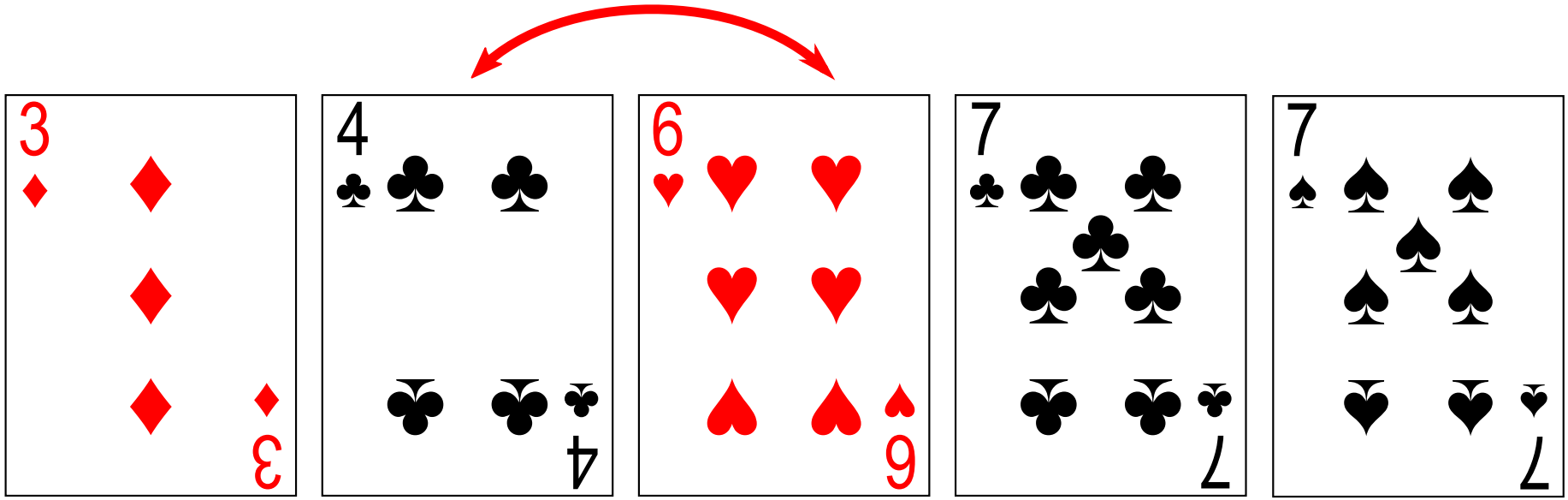
Beispiel: Manuelles Sortieren



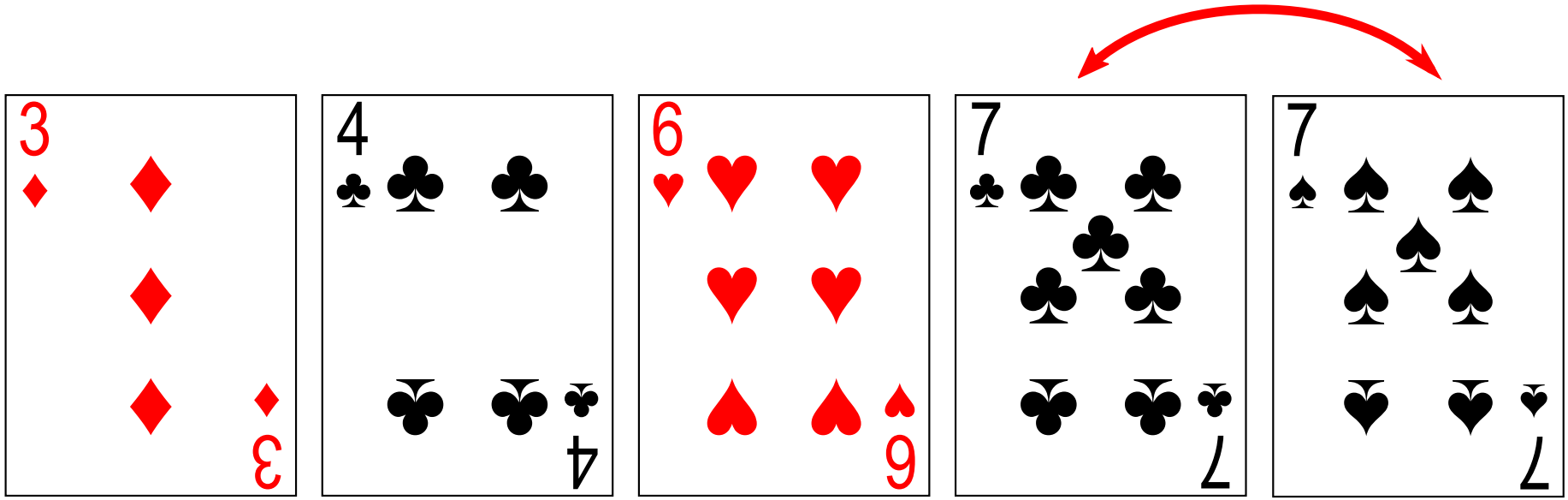
Beispiel: Manuelles Sortieren



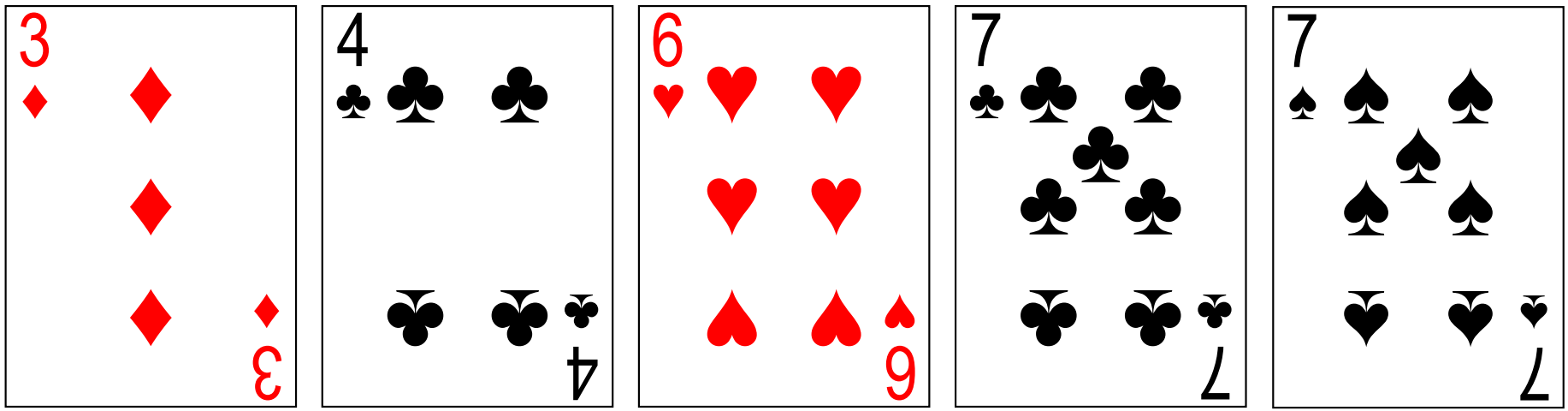
Beispiel: Manuelles Sortieren



Beispiel: Manuelles Sortieren



Beispiel: Manuelles Sortieren



Sortierverfahren

Ziel

Ordnen der Elemente einer Sequenz in einer **definierten Reihenfolge**.

Sortierverfahren

Ziel

Ordnen der Elemente einer Sequenz in einer **definierten Reihenfolge**.

Operationen

- **Bewertung** der Relation zweier Elemente zueinander.
- **Vertauschung** zweier Elemente.

Sortierverfahren

Ziel

Ordnen der Elemente einer Sequenz in einer **definierten Reihenfolge**.

Operationen

- **Bewertung** der Relation zweier Elemente zueinander.
- **Vertauschung** zweier Elemente.

Implementierung als Computerprogramm

- Üblicherweise Sortierung von Zahlenwerten (z.B. “Double” oder “**Integer**”) oder Objekten (z.B. “String”).
- Sequenzierung der Daten als **Feld** oder Liste.

Implementierung: Sequenzierung der Daten

Listing 1: Deklaration der Daten

```
const int N = ...;

int input[N];
```

Implementierung: Sequenzierung der Daten

Listing 1: Deklaration der Daten

```
const int N = ...;

int input[N];
```

Listing 2: Zugriff auf einzelne Elemente

```
input[0] = 7;

const int tmp = input[i];

input[N-1] = x;
```

Implementierung: Elementare Operationen

Implementierung: Elementare Operationen

Listing 3: Vergleich zweier Elemente

```
int less(const int *data, const int i, const int j)
{

}
```

Implementierung: Elementare Operationen

Listing 3: Vergleich zweier Elemente

```
int less(const int *data, const int i, const int j)
{
  return (int)(data[i] < data[j]);
}
```

Implementierung: Elementare Operationen

Listing 3: Vergleich zweier Elemente

```
int less(const int *data, const int i, const int j)
{
  return (int)(data[i] < data[j]);
}
```

Listing 4: Vertauschen zweier Elemente

```
void exch(int *data, const int i, const int j)
{



}
```

Implementierung: Elementare Operationen

Listing 3: Vergleich zweier Elemente

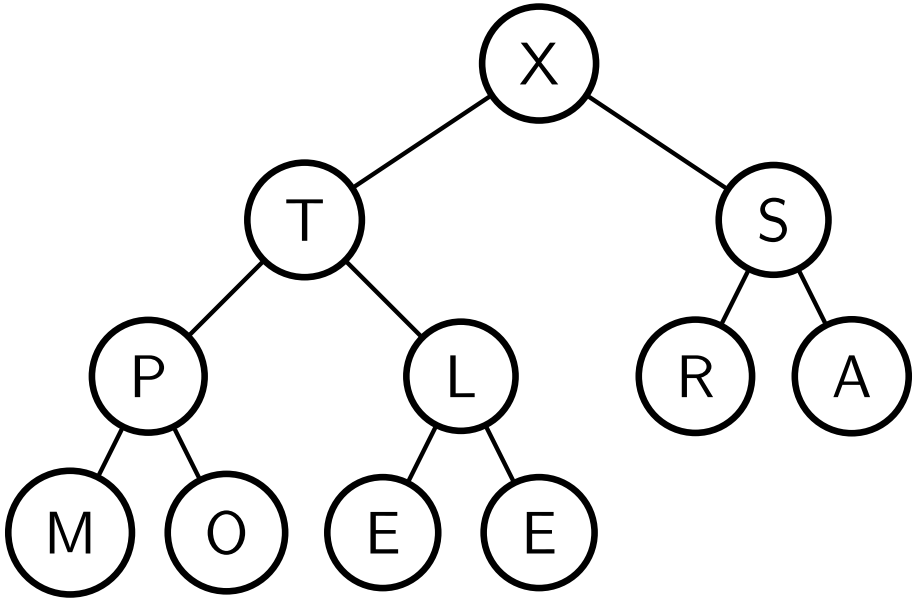
```
int less(const int *data, const int i, const int j)
{
  return (int)(data[i] < data[j]);
}
```

Listing 4: Vertauschen zweier Elemente

```
void exch(int *data, const int i, const int j)
{
  const int tmp = data[i];
  data[i] = data[j];
  data[j] = tmp;
}
```

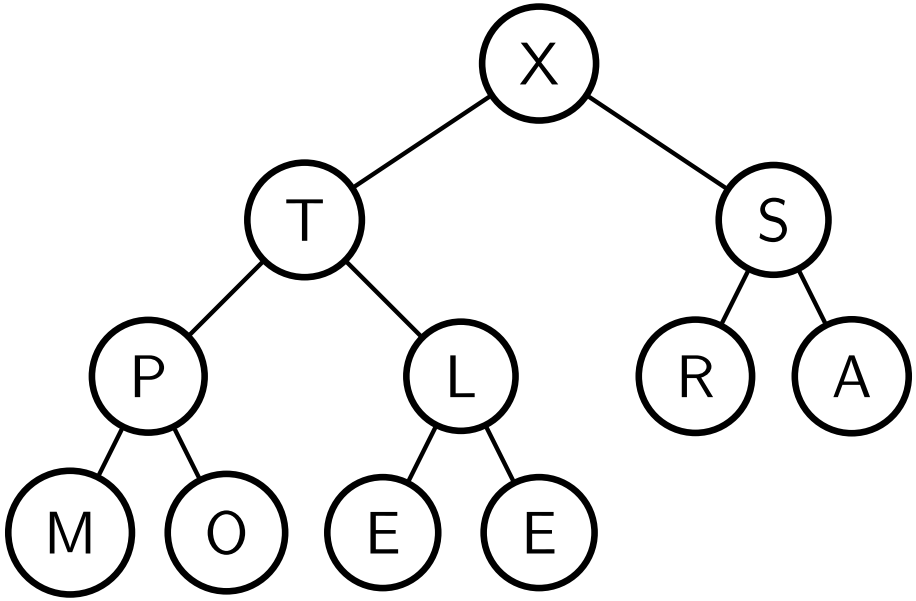
Inhalt

Binärer Heap – Konzept



Eigenschaften: *binary maximum heap*

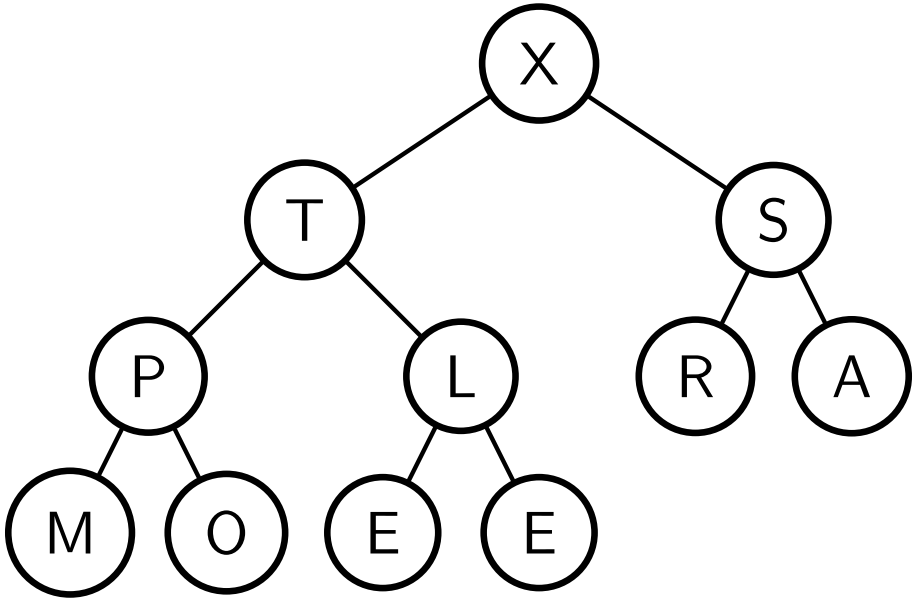
Binärer Heap – Konzept



Eigenschaften: *binary maximum heap*

- In einem **geordneten** *heap* ist der Wert eines Knotens **größer** oder **gleich** dem Wert seiner Kinder.

Binärer Heap – Konzept

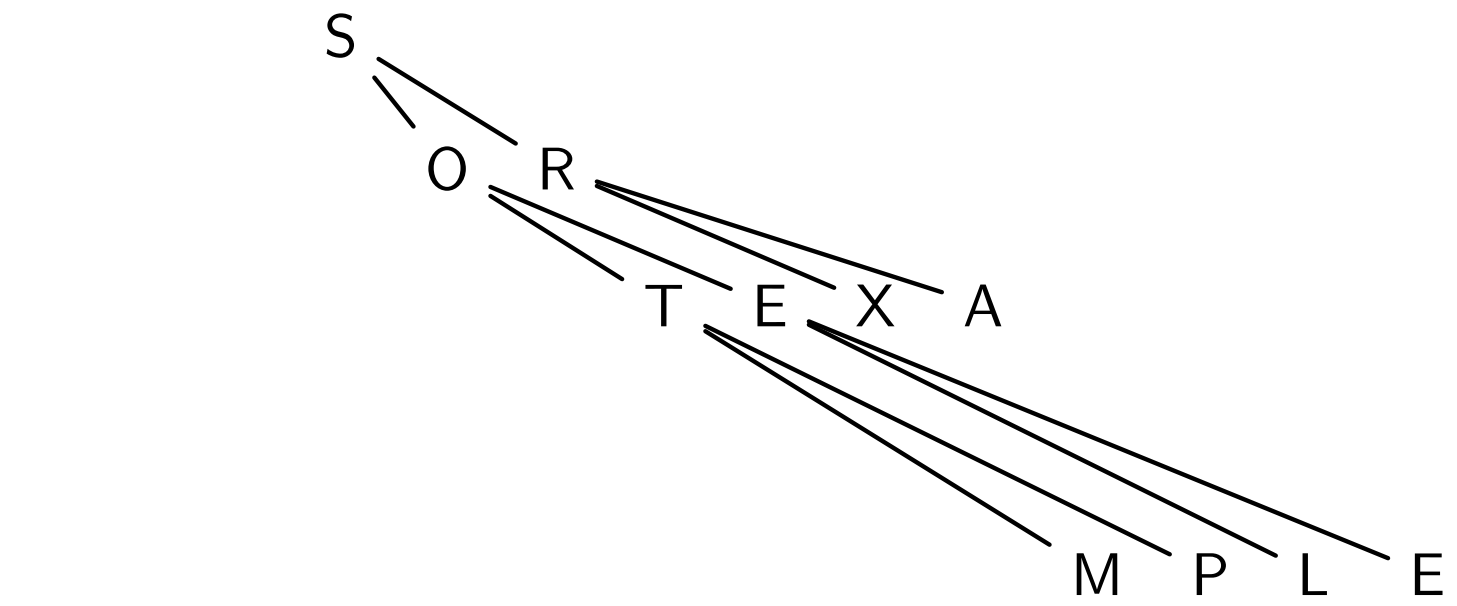


Eigenschaften: *binary maximum heap*

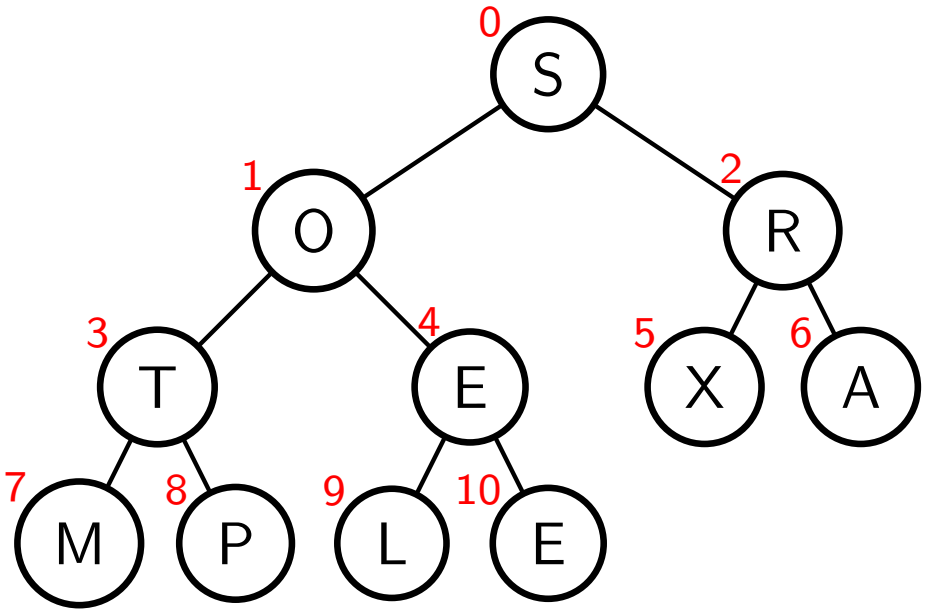
- In einem **geordneten** *heap* ist der Wert eines Knotens **größer** oder **gleich** dem Wert seiner Kinder.
- Die **Wurzel** besitzt den **größten** Wert.

Implementierung – *complete binary heap* 1

i	0	1	2	3	4	5	6	7	8	9	10
input[i]	S	O	R	T	E	X	A	M	P	L	E

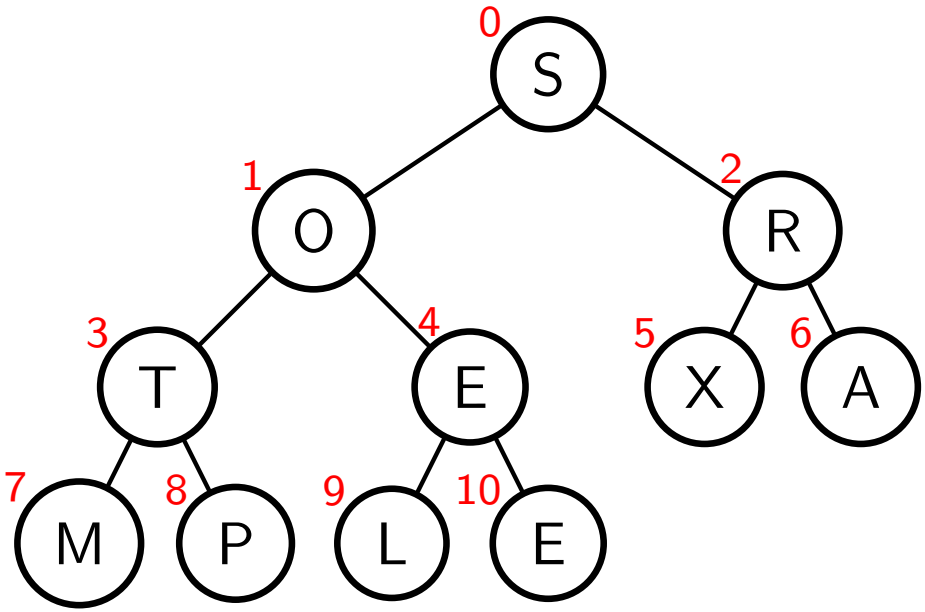


Implementierung – *complete binary heap* 2



Eigenschaften: *complete binary heap*

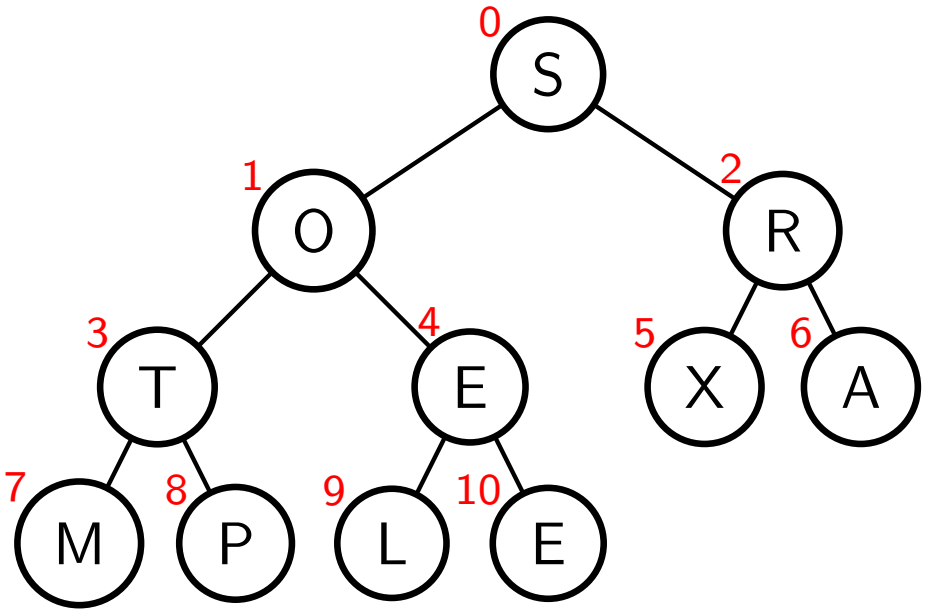
Implementierung – *complete binary heap* 2



Eigenschaften: *complete binary heap*

- Die **Ebenen** eines vollständigen, binären Heaps sind **fortlaufend** in einem Feld abgelegt.

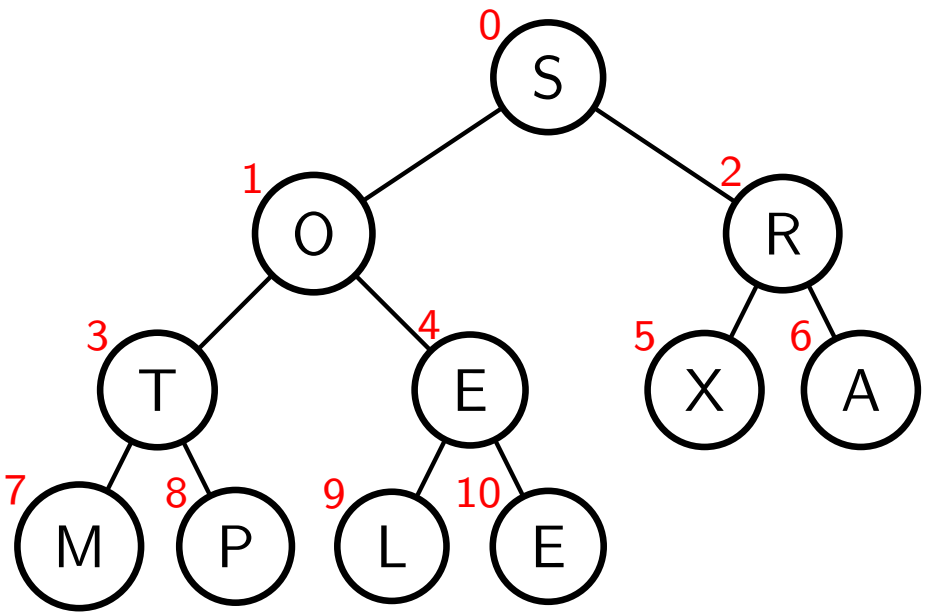
Implementierung – *complete binary heap* 2



Eigenschaften: *complete binary heap*

- Die **Ebenen** eines vollständigen, binären Heaps sind **fortlaufend** in einem Feld abgelegt.
- Die **Kinder** des Elementes k befinden sich an den Feldindizes $\mathbf{2k+1}$ und $\mathbf{2k+2}$.

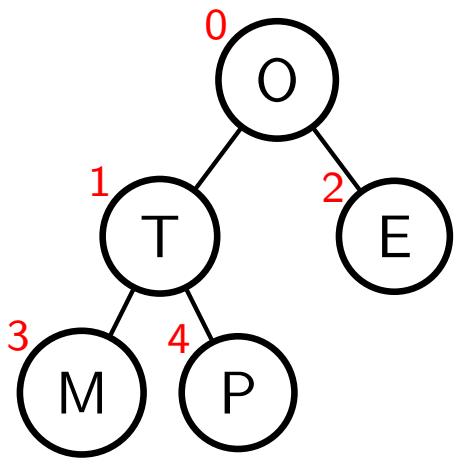
Implementierung – *complete binary heap* 2



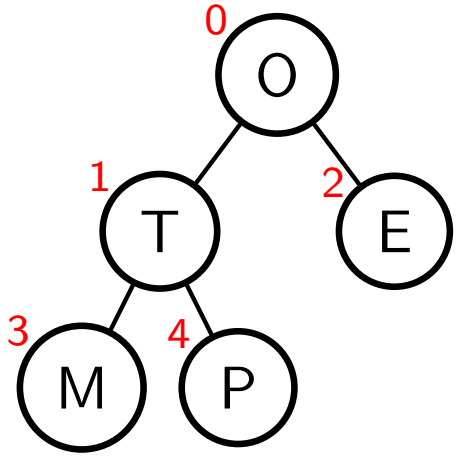
Eigenschaften: *complete binary heap*

- Die **Ebenen** eines vollständigen, binären Heaps sind **fortlaufend** in einem Feld abgelegt.
- Die **Kinder** des Elementes k befinden sich an den Feldindizes $\mathbf{2k+1}$ und $\mathbf{2k+2}$.
- Die **Höhe** eines binären Heaps von N Elementen ist $\lfloor \mathbf{\log_2 N} \rfloor$.

Herstellen der Heap-Ordnung



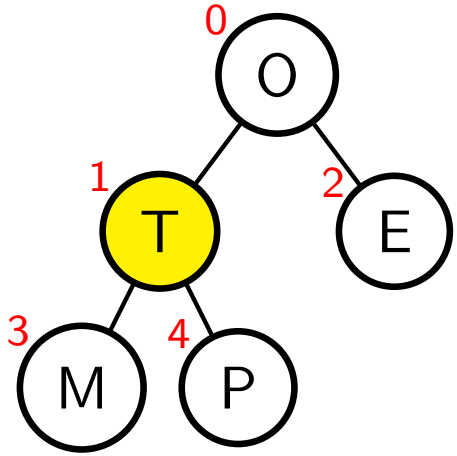
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.

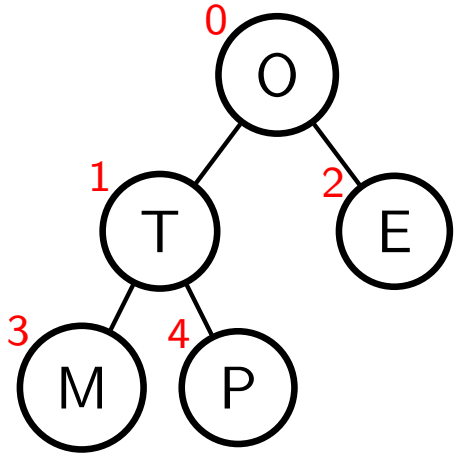
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.

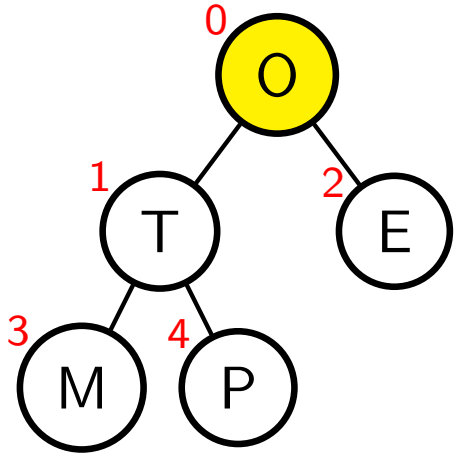
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.

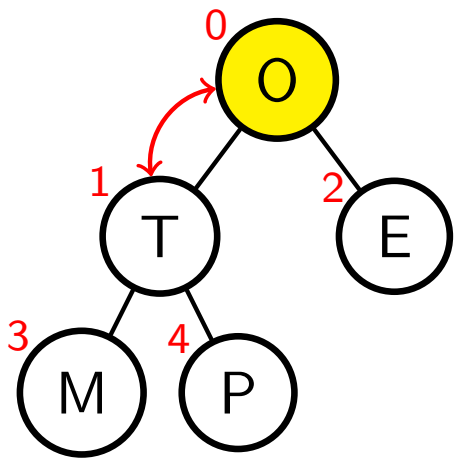
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.

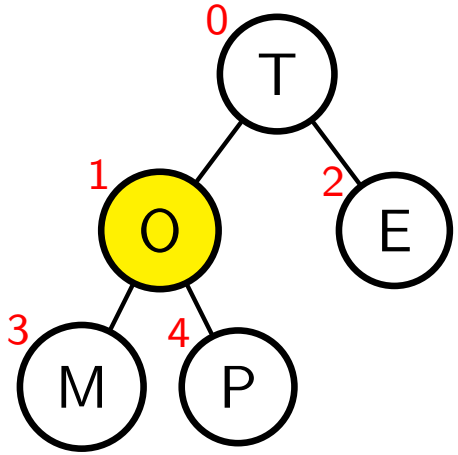
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.
- Zum **Absenken** eines Elementes wird die Relation zum **größeren** der beiden Kinder bewertet.

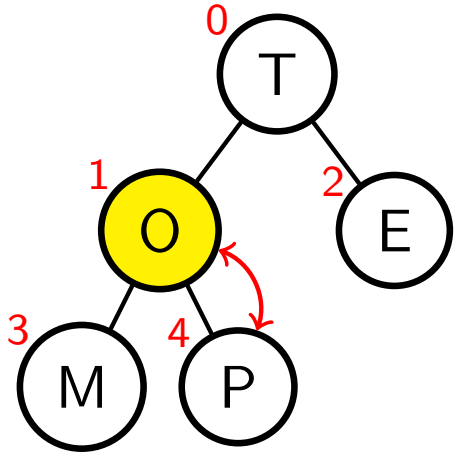
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.
- Zum **Absenken** eines Elementes wird die Relation zum **größeren** der beiden Kinder bewertet.

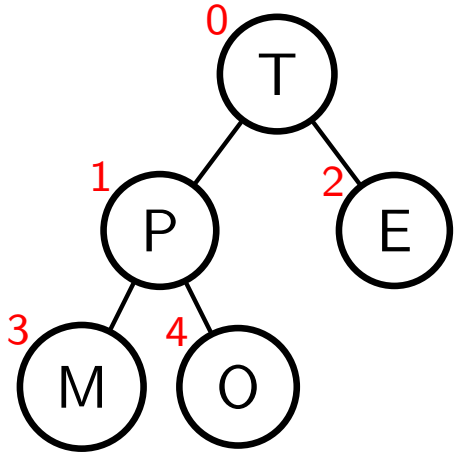
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.
- Zum **Absenken** eines Elementes wird die Relation zum **größeren** der beiden Kinder bewertet.

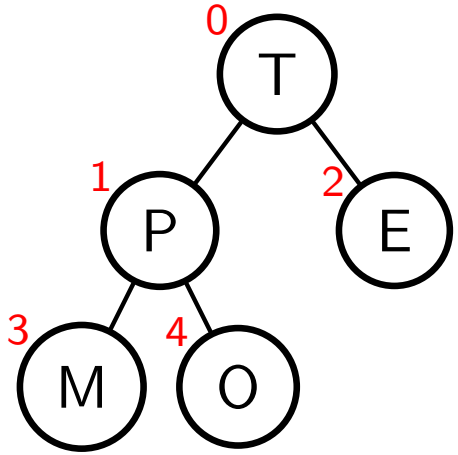
Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.
- Zum **Absenken** eines Elementes wird die Relation zum **größeren** der beiden Kinder bewertet.

Herstellen der Heap-Ordnung



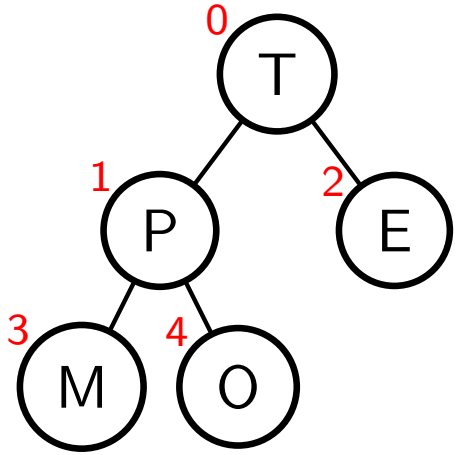
Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.
- Zum **Absenken** eines Elementes wird die Relation zum **größeren** der beiden Kinder bewertet.

Listing 5: Herstellen der Heap-Ordnung

```
for(int k = N/2-1; k >= 0; k--) {

}
```

Herstellen der Heap-Ordnung



Eigenschaften: *reheapify*

- Die **Verarbeitung** der Elemente erfolgt ***top-down***.
- Ausgehend von Position $\frac{N}{2} - 1$ erfolgt die Verarbeitung in **absteigender** Reihenfolge.
- Zum **Absenken** eines Elementes wird die Relation zum **größeren** der beiden Kinder bewertet.

Listing 5: Herstellen der Heap-Ordnung

```
for(int k = N/2-1; k >= 0; k--) {
  sink(data, k, N);
}
```

Implementierung: *top-down reheapify* (“*sink*”)

Listing 6: Herstellen der Heap-Ordnung

```
void sink(int *data, int k, const int N)
{




}
```

Implementierung: *top-down reheapify* (“*sink*”)

Listing 6: Herstellen der Heap-Ordnung

```
void sink(int *data, int k, const int N)
{
  while( 2*k+1 < N ) {
    int child = 2*k+1;









  }
}
```

Implementierung: *top-down reheapify* (“*sink*”)

Listing 6: Herstellen der Heap-Ordnung

```
void sink(int *data, int k, const int N)
{
  while( 2*k+1 < N ) {
    int child = 2*k+1;
    if( child+1 < N  &&  less(data, child, child+1) ) {
      child++;
    }




  }
}
```

Implementierung: *top-down reheapify* ("*sink*")

Listing 6: Herstellen der Heap-Ordnung

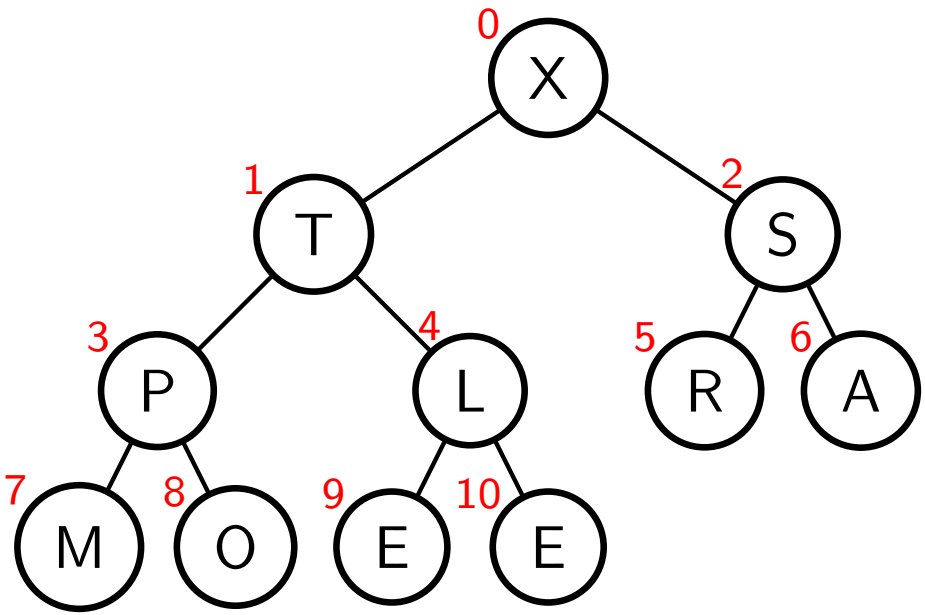
```
void sink(int *data, int k, const int N)
{
  while( 2*k+1 < N ) {
    int child = 2*k+1;
    if( child+1 < N  &&  less(data, child, child+1) ) {
      child++;
    }
    if( !less(data, k, child) ) {
      break;
    }


  }
}
```

Implementierung: *top-down reheapify* (“*sink*”)

Listing 6: Herstellen der Heap-Ordnung

```
void sink(int *data, int k, const int N)
{
  while( 2*k+1 < N ) {
    int child = 2*k+1;
    if( child+1 < N  &&  less(data, child, child+1) ) {
      child++;
    }
    if( !less(data, k, child) ) {
      break;
    }
    exch(data, k, child);
    k = child;
  }
}
```

Ergebnis: vollständiger, binärer Heap



Inhalt

Heapsort

Hintergrund

- Williams (Heapsort) & Floyd (*in-place* Variante), 1964

Heapsort

Hintergrund

- Williams (Heapsort) & Floyd (*in-place* Variante), 1964
- nutze Eigenschaften des binären Heaps
  - → Position des Maximums bekannt
  - → geringer Aufwand zur Korrektur einer Fehlstelle

Heapsort

Hintergrund

- Williams (Heapsort) & Floyd (*in-place* Variante), 1964
- nutze Eigenschaften des binären Heaps
  - → Position des Maximums bekannt
  - → geringer Aufwand zur Korrektur einer Fehlstelle
- Algorithmus gliedert sich in zwei Phasen
  1. Herstellen der Heap-Ordnung
  2. Sortierphase

Heapsort

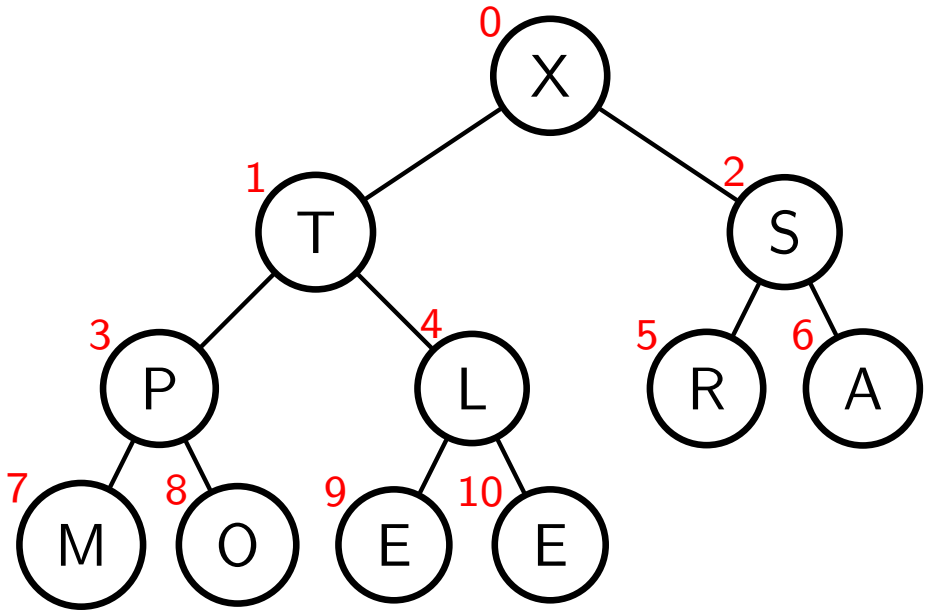
Hintergrund

- Williams (Heapsort) & Floyd (*in-place* Variante), 1964
- nutze Eigenschaften des binären Heaps
  - → Position des Maximums bekannt
  - → geringer Aufwand zur Korrektur einer Fehlstelle
- Algorithmus gliedert sich in zwei Phasen
  1. Herstellen der Heap-Ordnung
  2. Sortierphase

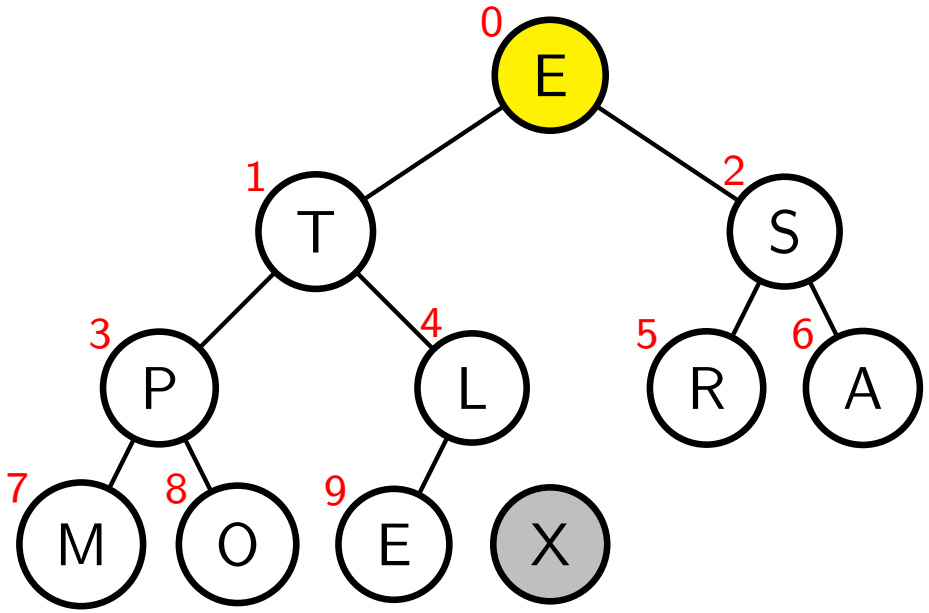
Eigenschaften

- Komplexität: $\mathcal{O}(N \log N)$
- nicht stabil
- *in-place*

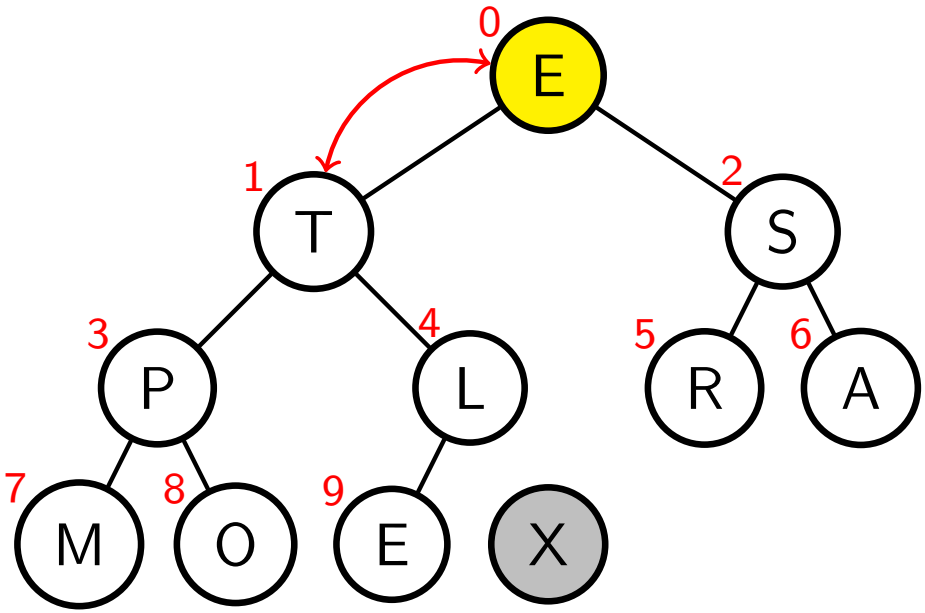
Heapsort – Sortierphase (1. Durchlauf)



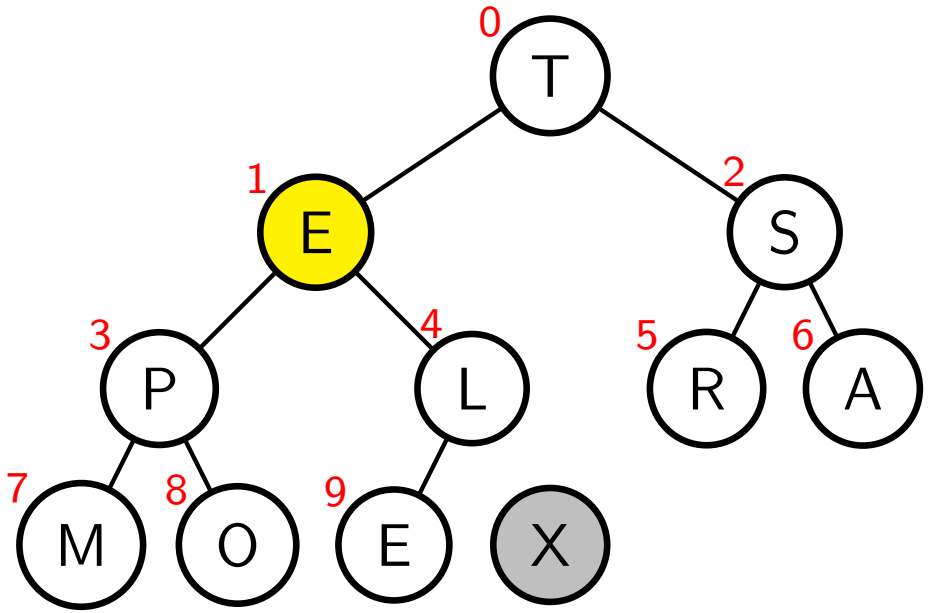
Heapsort – Sortierphase (1. Durchlauf)



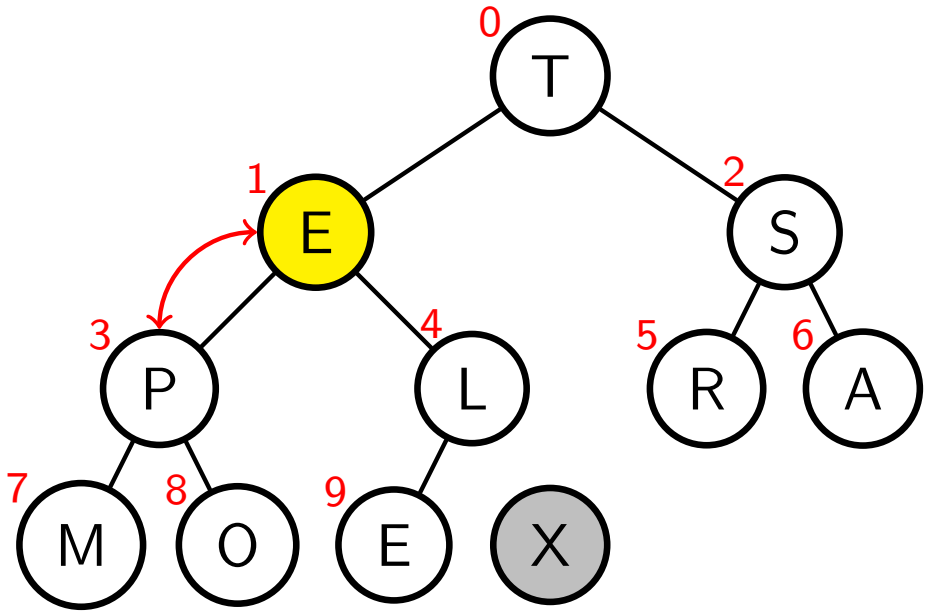
Heapsort – Sortierphase (1. Durchlauf)



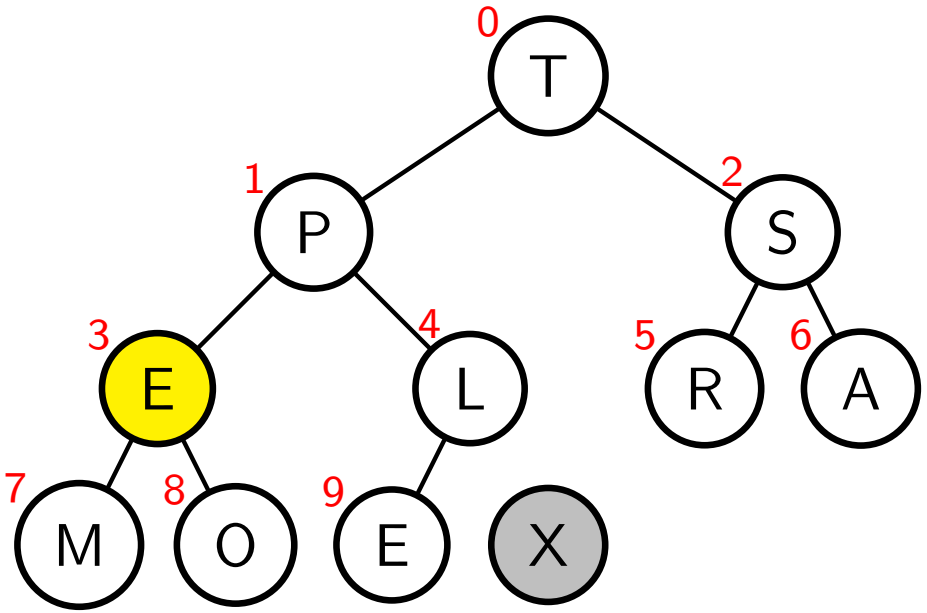
Heapsort – Sortierphase (1. Durchlauf)



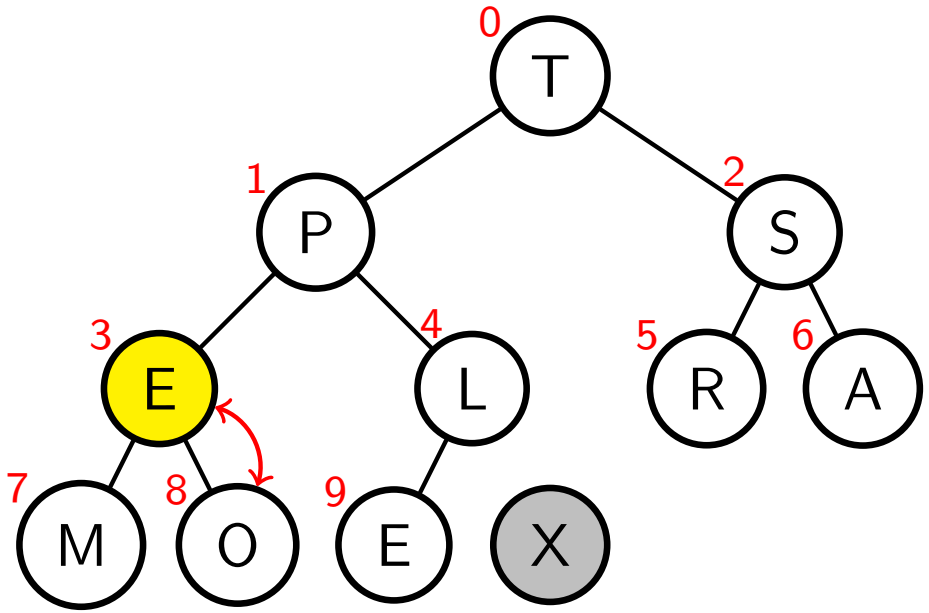
Heapsort – Sortierphase (1. Durchlauf)



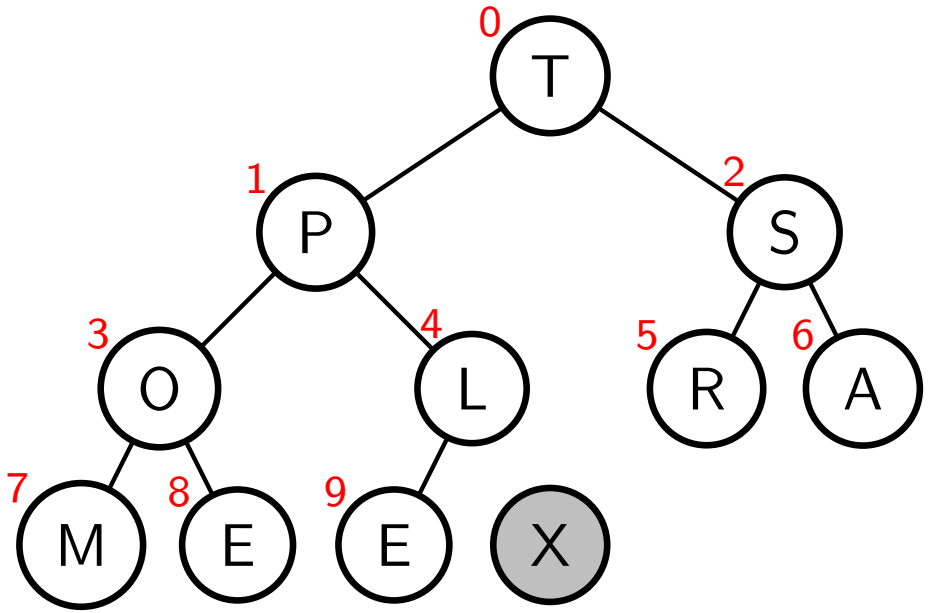
Heapsort – Sortierphase (1. Durchlauf)



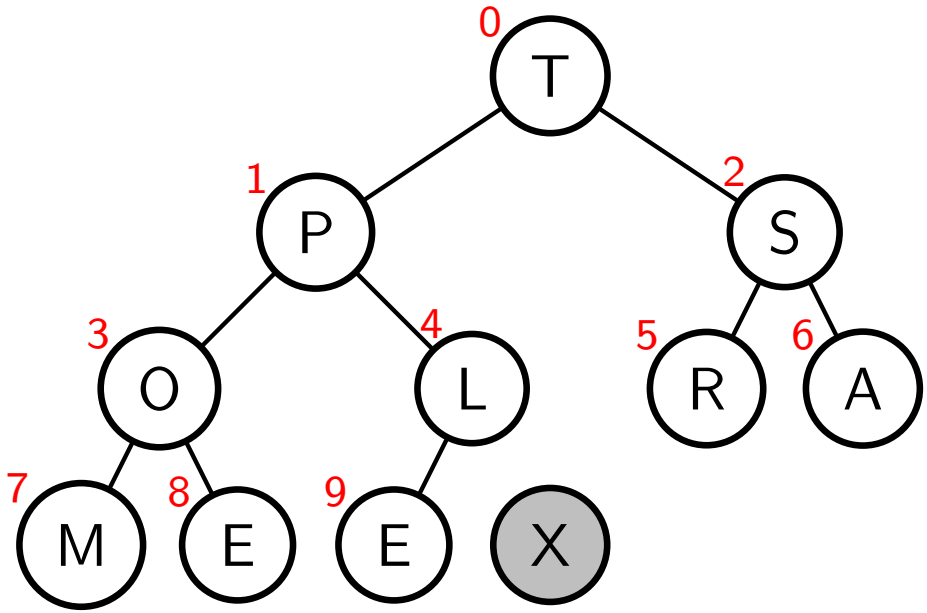
Heapsort – Sortierphase (1. Durchlauf)



Heapsort – Sortierphase (1. Durchlauf)



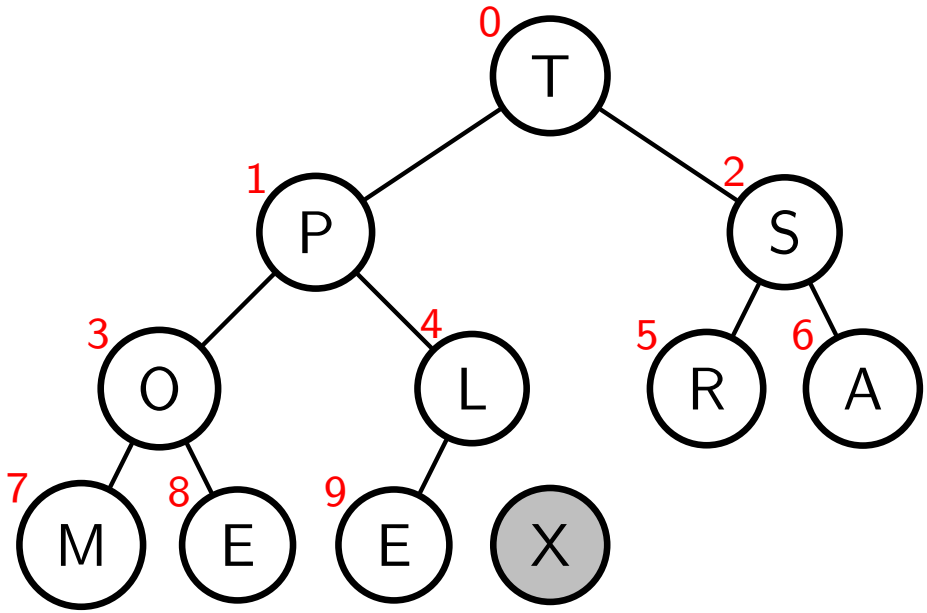
Heapsort – Sortierphase (1. Durchlauf)



Operationen der Sortierphase

- **Entfernen** des **Maximums**

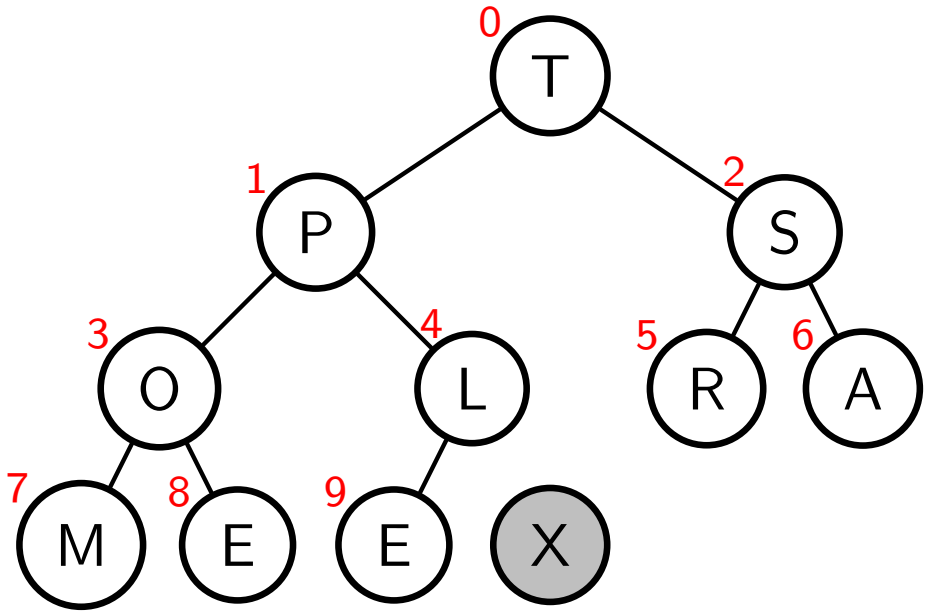
Heapsort – Sortierphase (1. Durchlauf)



Operationen der Sortierphase

- **Entfernen** des **Maximums**
- **Wiederherstellen** der **Heap-Ordnung**

Heapsort – Sortierphase (1. Durchlauf)



Operationen der Sortierphase

- **Entfernen** des **Maximums**
- **Wiederherstellen** der **Heap-Ordnung**

$\Rightarrow$ Die **Komplexität** eines Durchlaufes ist $\mathcal{O}(\mathbf{log}\,N)$.

Implementierung: Heapsort

Implementierung: Heapsort

Listing 7: Heapsort

```
void heapsort(int *data, int N)
{









}
```

Implementierung: Heapsort

Listing 7: Heapsort

```
void heapsort(int *data, int N)
{
  for(int k = N/2-1; k >= 0; k--) {
    sink(data, k, N);
  }



}
```

Implementierung: Heapsort

Listing 7: Heapsort

```
void heapsort(int *data, int N)
{
  for(int k = N/2-1; k >= 0; k--) {
    sink(data, k, N);
  }
  while( N > 1 ) {


  }
}
```

Implementierung: Heapsort

Listing 7: Heapsort

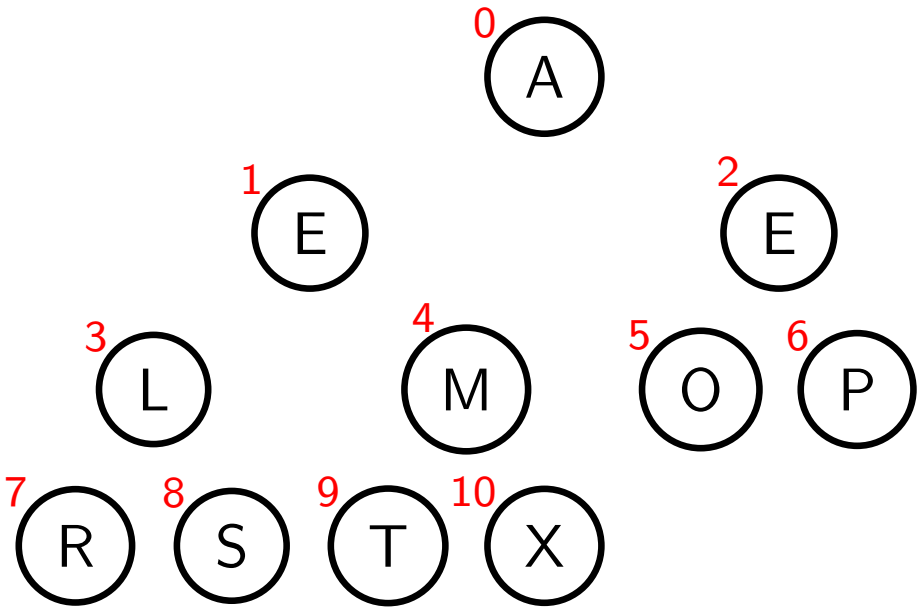
```
void heapsort(int *data, int N)
{
  for(int k = N/2-1; k >= 0; k--) {
    sink(data, k, N);
  }
  while( N > 1 ) {
    exch(data, 0, N-1);

  }
}
```

Implementierung: Heapsort

Listing 7: Heapsort

```
void heapsort(int *data, int N)
{
  for(int k = N/2-1; k >= 0; k--) {
    sink(data, k, N);
  }
  while( N > 1 ) {
    exch(data, 0, N-1);
    sink(data, 0, --N);
  }
}
```

Ergebnis: sortierte Sequenz



Zusammenfassung

Verfahren	Komplexität	stabil	*in-place*	Anmerkung
Selectionsort	$\mathcal{O}(N^2)$	nein	ja	
Insertionsort	$\mathcal{O}(N^2)$	ja	ja	
Heapsort	$\mathcal{O}(N \log N)$	nein	ja	
Quicksort	$\mathcal{O}(N \log N)$	nein	ja	parallelisierbar
Mergesort	$\mathcal{O}(N \log N)$	ja	nein	parallelisierbar

Ende

Literatur

- *Algorithms*, 4th ed., R. Sedgewick & K. Wayne

Visualisierung & Implementierung

- `https://github.com/CaSchmidt/csheapsort`

Vielen Dank!