

Algorithmen und Datenstrukturen

Sommersemester 2020

Stefan Näher

Universität Trier

naeher@uni-trier.de

Vorlesung 5

6. Mai 2020

Kapitel II: Sortieren

Allgemeine Sortierverfahren

- **Heapsort** (Sortieren durch Max-Auswahl)
- **Quicksort** (Sortieren durch Teilen)
- **Mergesort** (Sortieren durch Mischen)

Spezielle Sortierverfahren

- **Counting Sort** (Sortieren durch Zählen)
- **Bucket Sort** (Sortieren durch Verteilung)

Heapsort: Sortieren durch Max-Auswahl

Grundidee: Suche für $r = n, n - 1, \dots, 2$ jeweils nach Maximum in $A[1..r]$ und vertausche es mit letzter Position.

$r \leftarrow n;$

while $r > 1$ **do**

 // finde Position j des Maximums in $A[1..r]$

$j = 1;$

for $i = 2$ **to** r **do**

if $A[i] > A[j]$ **then** $j \leftarrow i$; **fi**

od

$A[j] \longleftrightarrow A[r];$ // vertausche Max mit letztem Element

$r \leftarrow r - 1;$

od

Laufzeit

$n - 1$ mal lineare Suche in Feld der Länge $n, n - 1, \dots, 2$

Laufzeit: $\mathcal{O}(n^2)$ **WARUM ?**

Frage: Muss die Maximumsuche “unbedingt” eine lineare sein und somit Linearzeit benötigen ?

NEIN! Wenn wir die Daten geeignet halten (in einem in einem Feld gespeicherten speziellen Baum (**Heap**)). . .

Heaps

Ein **Heap** (Haufen) ist ein Baum, dessen Knoten mit Zahlen $\text{num}(v)$ beschriftet sind, so dass für jeden Knoten v , der nicht die Wurzel ist, gilt:

$$\text{num}(v) \leq \text{num}(\text{parent}(v))$$

Folgerung: Die größte Zahl in einem Heap steht an der Wurzel.
Daher heißt so ein Heap auch **Max-Heap**.

Analog: **Min-Heap**

Wichtig: “Regelmäßige” Heaps.

Binärbäume

Ein **Binärbaum** ist ein Baum, bei dem jeder Knoten null oder zwei Kinder hat (vielleicht mit einer Ausnahme).

Knoten mit null Kindern heißen auch **Blätter**.

Die Länge des Pfades von der Wurzel zu einem Knoten v heißt **Tiefe** von v .

Ein Binärbaum heißt **ausgeglichen**, wenn es eine Zahl k gibt mit:

- (1) Alle Blätter haben Tiefe k oder $k + 1$.
- (2) Auf Tiefe $k + 1$ stehen die Blätter so weit links wie möglich.

Skizze ...

Heaps als Feld

Heaps lassen sich statisch in einem Feld abspeichern.

Für einen ausgeglichenen Heap mit n Knoten benötigen wir ein Feld $A[1..n]$.

Dabei gilt:

- (1) $2i$ und $2i + 1$ sind die Indizes des linken bzw. rechten Kindes von i .
- (2) $\lfloor i/2 \rfloor$ ist der Index des Elternteils von i .

Heap-Eigenschaft

Wird in dieser Weise ein Heap in $A[1..n]$ abgespeichert, so gilt:

$$\forall i \in \{2, \dots, n\} : A[i] \leq A[\lfloor i/2 \rfloor].$$

Umgekehrt lässt sich jedes Feld mit dieser Heap-Eigenschaft als ausgeglichener Heap interpretieren.

Heapsort — Grundidee

Heapsort ist eine Methode zum Sortieren von Feldern.

Da das Ausgangsfeld im Allg. keine Heap-Eigenschaft besitzt, zerfällt der Algorithmus in zwei Phasen:

1. **Aufbauphase**: Das Eingabefeld A wird in einen Heap verwandelt.

2. **Auswahlphase** oder **Selektionsphase**:

Sortieren durch Auswahl wird mit Hilfe der Heap-Struktur implementiert.

Dabei muss darauf geachtet werden, dass “zwischendurch” die Heap-Eigenschaft erhalten bleibt.

Die Aufbauphase von Heapsort

konstruiert von unten nach oben (bottom-up) Heaps der Höhe $2, 3, \dots$ durch Heruntersinkenlassen von Werten.

Beobachtung: $\lfloor n/2 \rfloor$ ist der erste Knoten (von hinten) der Kinder besitzt (d.h. kein Blatt ist).

```
for  $i = \lfloor n/2 \rfloor$  downto 1 do  
    SINK( $i, n$ );  
od
```

SINK(i, n) soll $A[i]$ im Heap $A[1..n]$ hinuntersinken lassen.

Dieses Vorgehen heißt auch **Versenken**, **Sinkenlassen** oder **Versickern** oder **Heapify**.

SINK(i,n)

```
// lasse  $A[i]$  im Heap  $A[1..n]$  heruntersinken
 $x \leftarrow A[i]$ ;
 $j \leftarrow 2 \cdot i$ ; // linkes Kind von  $A[i]$ 
while  $j \leq n$  do
    if  $j < n \wedge A[j + 1] > A[j]$  then
         $j \leftarrow j + 1$  // gehe zum rechten Kind, falls größer
    fi
    if  $x \geq A[j]$  then break ; fi ; // Abbruch
     $A[i] \leftarrow A[j]$ ;
     $i \leftarrow j$ ;
     $j \leftarrow 2 \cdot i$  // weiter runtersteigen
od
 $A[i] \leftarrow x$ 
```

Ein Beispiel auf Feldebene

```

1 2 3 4 5 6 7 8
H E A P S O R T
      ^           ^
H E A T S O R P
      ^           ^ ^
H E R T S O A P
      ^   ^ ^
H T R E S O A P
      ^           ^
H T R P S O A E
^ ^ ^
T H R P S O A E
      ^   ^ ^
T S R P H O A E

```

Wir beginnen links von der Mitte, d. h. bei P:
 sein Nachfolger ist T. Da $T > P$ ist, tauschen wir beide.
 Wir fahren mit dem A fort. Seine Nachfolger sind
 O und R. Es gilt sowohl $O > A$ als auch $R > A$.
 $R > O$, also tauschen wir R und A.

Dann vergleichen wir E mit seinen Nachfolgern T und S.
 Es gilt $T > S$ und $T > E$. Deshalb müssen wir T und E vertauschen.

Wir müssen nun E weiter sinken lassen, denn der neue
 Nachfolger von E ist P, und $P > E$.

Nun vergleichen wir H mit seinen
 Nachfolgern T und R. $T > R$ und $T > H$.

Wir lassen H weiter sinken.
 $S > P$ und $S > H$.

Wir haben das Array nun in einen Max-Heap überführt.

Bemerkungen zur Korrektheit

Nach jedem Schleifendurchlauf der **Aufbauphase** gilt:
alle Binärbäume, die von $A[i]$ bis $A[n]$ als Wurzeln induziert werden, sind Heaps.

Diese Eigenschaft wird (induktiv) beim Aufruf von SINK benutzt, denn die einzige *Konfliktstelle* stellt das neu hinzukommende Elternteil dar.

Durch Tauschen an eine geeignete Stelle wird gewährleistet, dass das Maximum (des Teilbaums) an der Wurzel steht und auch (noch) die Heapeigenschaft für die (anderen) Teilbäume gewahrt ist.

Selektionsphase

Das eigentliche Sortieren durch Maximumsauswahl.

Beachte: Das Maximum steht nun in $A[1]$ (da A ein Heap ist).

$r \leftarrow n;$

while $r > 1$ **do**

 // $A[1]$ ist maximal, da $A[1..r]$ ein Heap ist.

$A[1] \longleftrightarrow A[r];$

$r \leftarrow r - 1;$

 SINK($A, 1, r$); // verwandelt $A[1..r]$ in einen Heap

od

Selektionsphase am Beispiel (rechts vom Strich sortiert)

```

1 2 3 4 5 6 7 8
T S R P H O A E
^                               ^

```

Wir vertauschen das erste Element mit dem letzten.

```

E S R P H O A | T
^ ^ ^

```

Das T ist nun an der korrekten Position.

Nun müssen wir das E sinken lassen. $S > R$ und $S > E$
 $P > H$ und $P > E$.

```

S E R P H O A | T
  ^   ^ ^

```

```

S P R E H O A | T
^                               ^

```

Wir lassen E nicht weiter sinken, denn T liegt bereits jenseits des Heaps im fertig sortierten Bereich.

Wir haben also wieder einen korrekten Heap und können S und A vertauschen, womit auch das S sortiert ist.

```

A P R E H O | S T
^ ^ ^

```

Jetzt müssen wir das A sinken lassen.

$R > P$ und $R > A$.

```

R P A E H O | S T
    ^       ^

```

Da das S bereits korrekt liegt, vergleichen wir nur A und O. $O > A$.

```

R P O E H A | S T
^             ^

```

Die Heap-Eigenschaft für das linke Teilarray ist wieder erfüllt. Wir vertauschen R und A.

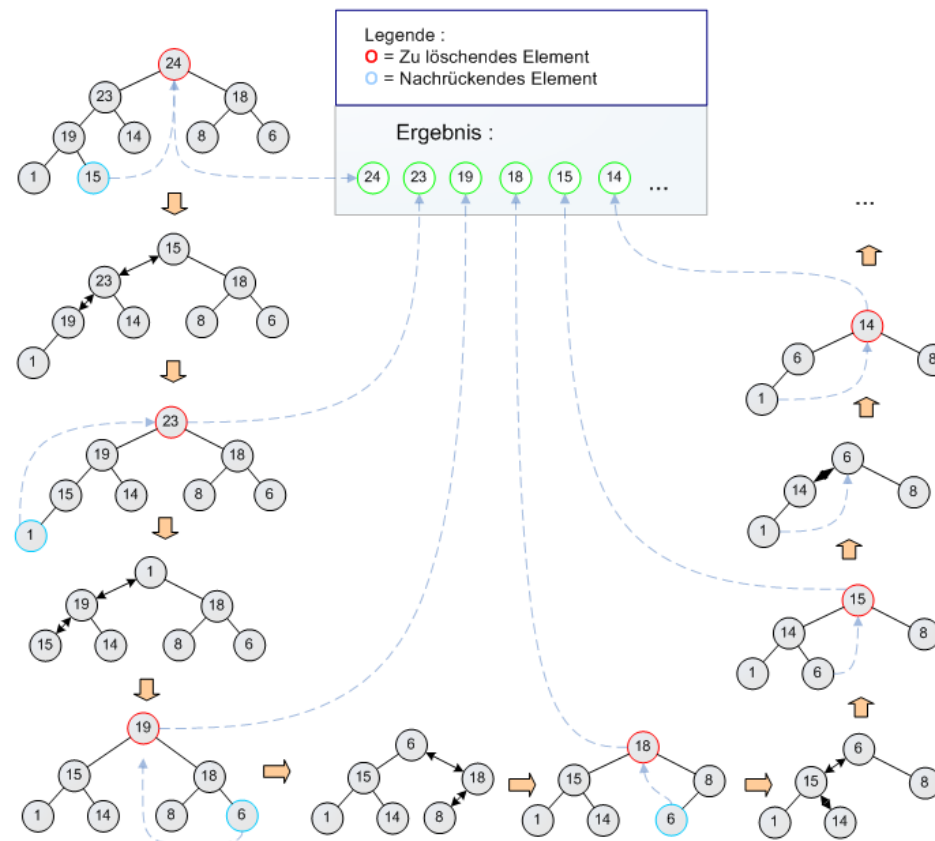
Selektionsphase am Beispiel (Forts.) Rechts vom Strich sortiert

A P O E H R S T ^ ^ ^	Wir lassen A sinken. P>O und P>A.
P A O E H R S T ^ ^ ^	Wir lassen A weiter sinken. H>E und H>A.
P H O E A R S T ^ ^	Wir vertauschen P und A.
A H O E P R S T ^ ^ ^	Wir lassen A sinken. O>H und O>A.
O H A E P R S T ^ ^	Wir vertauschen O und E.
E H A O P R S T ^ ^ ^	Wir lassen E sinken. H>A und H>E.
H E A O P R S T ^ ^	Wir vertauschen H und A.
A E H O P R S T ^ ^	Wir lassen A sinken. E>A.
E A H O P R S T ^ ^	Wir vertauschen E und A.
A E H O P R S T	Das Array ist jetzt fertig sortiert.

Ein Baumbeispiel

Zu sortierende Folge : 23,1,6,19,14,18,8,24,15

Heapsort :



Laufzeitanalyse: Satz: Heapsort arbeitet in $\mathcal{O}(n \log n)$.

Es sei $h(i)$ die Höhe des von Knoten i in A induzierten Teilbaums, kurz auch **Höhe von i** genannt.

Der Aufwand für $\text{SINK}(i, n)$ lässt sich also mit $\mathcal{O}(h(i))$ abschätzen.

Die Anzahl der Knoten auf Höhe h ist höchstens $\frac{n}{2^h}$.

Dies folgt durch einen leichten Induktionsbeweis.

$$\sum_{i=1}^{n/2} h(i) \leq \sum_{h=1}^{\lceil \log_2(n) \rceil} h \cdot \frac{n}{2^h} \leq \sum_{h=1}^{\infty} n \cdot \frac{h}{2^h} \leq n \cdot 2$$

$\leadsto$ Aufwand für die Aufbauphase: $\mathcal{O}(n)$.

Da $h \leq \lceil \log_2(n) \rceil$, folgt: Aufwand für Selektionsphase: $\mathcal{O}(n \cdot \log n)$.

Varianten & Geschichte

Heapsort ist ein 1964 von Robert W. Floyd und J. W. J. Williams entwickeltes, relativ schnelles Sortierverfahren. Es handelt sich um eine Verbesserung von Selectionsort.

BottomUp-Heapsort

ist ein Sortieralgorithmus, der u.a. 1990 von Ingo Wegener vorgestellt wurde und im Durchschnitt besser als Quicksort arbeitet, falls man Vergleichsoperationen hinreichend stark gewichtet. Im Durchschnittsfall benötigt BottomUp-Heapsort nur $n \log_2(n) + \mathcal{O}(n)$ Schlüsselvergleiche, selbst im schlimmsten Fall nur $n \log_2(n) + \mathcal{O}(n \log \log(n))$. Es wird ausgenutzt, dass “meist” in die Nähe der Blattebene abgesenkt werden muss.