Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen

Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

Sortieren: Fortgeschrittene Sortierverfahren

- Sortieren durch Auswahl (selection sort)
  $\leadsto$ Heapsort

- Quicksort

Sortieren durch Auswahl

. . . hatten wir bereits früher betrachtet.

Grundidee: Suche n-mal nach Maximum (oder Minimum) im noch unsortierten Feld

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 Binärbaum). . .

Heaps (Haufen)

Ein *Heap* ist ein Baum, dessen Knoten mit Zahlen $\zeta(v)$ beschriftet sind, sodass für jeden Knoten v, der nicht die Wurzel ist, gilt: $\zeta(v) \leq \zeta(\text{Vater}(v))$.

Folgerung: Die größte Zahl in einem Heap steht an der Wurzel.
Daher heißt so ein Heap auch manchmal *Max-Heap*.
Wichtig: “Regelmäßige” Heaps.

Beispiel (Tafel).

Binärbäume I

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 an der Tafel.
Lemma: Es sei B ein Binärbaum. B ist ausgeglichen gdw. alle von beliebigen Knoten x von B als Wurzel induzierten Teilbäume ausgeglichen sind.

Binärbäume II

Erinnerung: Binärbäume lassen sich statisch in einem Feld abspeichern.

Für einen ausgeglichenen Binärbaum mit n Knoten benötigen wir dabei genau 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.

Folgerung: Wird in dieser Weise ein Heap in $A[1..n]$ abgespeichert, so gilt:
$\forall i \in \{2, \ldots, n\} : A[i] \leq A[\lfloor i/2 \rfloor]$. (*Heap-Eigenschaft*)

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.

Aufbauphase

Algorithm 1 Aufbauphase von Heapsort

Input(s): an array $A : \mathbb{R}[1..n]$

Output(s): a heap array $A : \mathbb{R}[1..n]$

for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**
  Versenke(A, i, n).

Naja: Wir haben das Problem auf “Versenke” verschoben.

Diese Prozedur soll $A[i]$ im “Baum” $A[1..n]$ hinuntersinken lassen.

Dieses Vorgehen selbst heißt *Versenken*, *Sinkenlassen* oder *Versickern*.

Algorithm 2 Versenke

Input(s): an array $A : \mathbb{R}[1..n]$, indices $i \leq r$.

Output(s): a heap array $A : \mathbb{R}[1..n]$

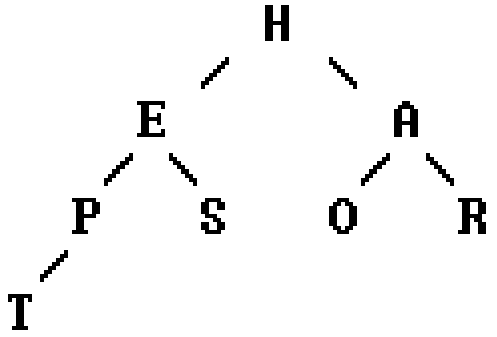
```
k ← A[i]; j ← 2i; {Linkes Kind von A[i]}
while (j ≤ r) do
    if (j < r) then
        if A[j] < A[j + 1] then
            j + + {Gehe zum rechten Kind, falls Zahl größer}
    if (k < A[j]) then
        A[i] ← A[j]; i ← j; j ← 2i {Runtersteigen}
    else
        j ← r + 1 {Abbruch von while}
end while
A[i] ← k
```

Ein Beispiel auf Feldebene

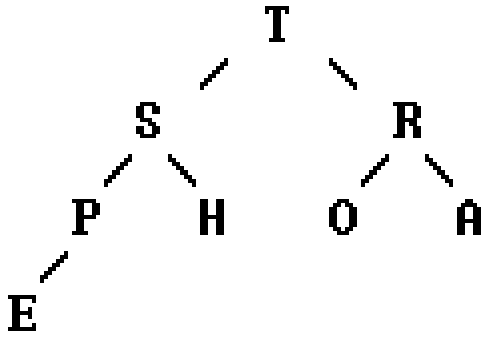
```
1 2 3 4 5 6 7 8
H E A P S O R T    Wir beginnen links von der Mitte, d. h. bei P:
      ^       ^    sein Nachfolger ist T. Da T>P ist, tauschen wir beide.
H E A T S O R P    Wir fahren mit dem A fort. Seine Nachfolger sind
    ^     ^ ^      O und R. Es gilt sowohl O>A als auch R>A.
                   R>O, also tauschen wir R und A.
H E R T S O A P    Dann vergleichen wir E mit seinen Nachfolgern T und S.
  ^   ^ ^          Es gilt T>S und T>E. Deshalb müssen wir T und E vertauschen
H T R E S O A P    Wir müssen nun E weiter versickern, denn der neue
      ^       ^    Nachfolger von E ist P, und P>E.
H T R P S O A E    Nun vergleichen wir H mit seinen
^ ^ ^              Nachfolgern T und R. T>R und T>H.
T H R P S O A E    Wir versickern das H weiter.
  ^   ^ ^          S>P und S>H.
T S R P H O A E    Wir haben das Array nun in einen Max-Heap überführt.
```

Felder und Bäume—zum Vergleich

Das Feld `HEAPSORT` als Baum:



Das Feld `TSRPHOAE` als Max-Heap:



Bemerkungen zur Korrektheit

Nach jedem Schleifendurchlauf von “Aufbauphase” gilt:
alle Binärbäume, die von $A[i]$ bis $A[n]$ als Wurzeln induziert werden, sind Heaps.
(Beachte dazu auch voriges Lemma.)

Diese Eigenschaft wird (induktiv) beim Aufruf von “Versenke” 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.

Algorithm 3 Selektionsphase

Input(s): an array $A : \mathbb{R}[1..n]$ satisfying the heap property
Output(s): a sorted array $A : \mathbb{R}[1..n]$

$r \leftarrow n$;
while $(r > 1)$ **do**
  {$A[1..r]$ is a heap; $A[r+1..n]$ is sorted}
  swap$(A[1], A[r])$;
  $r \leftarrow r - 1$;
  Versenke$(A, 1, r)$
  {$A[1..r-1]$ is a heap; $A[r..n]$ is sorted}
end while

Lemma: Die blauen Zusicherungen sind korrekt (und damit Heapsort).

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 versickern. S>R und S>E
S E R P H O A|T     P>H und P>E.
  ^   ^ ^
S P R E H O A|T     Wir versickern E nicht weiter, 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 versickern.
^ ^ ^               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.
```

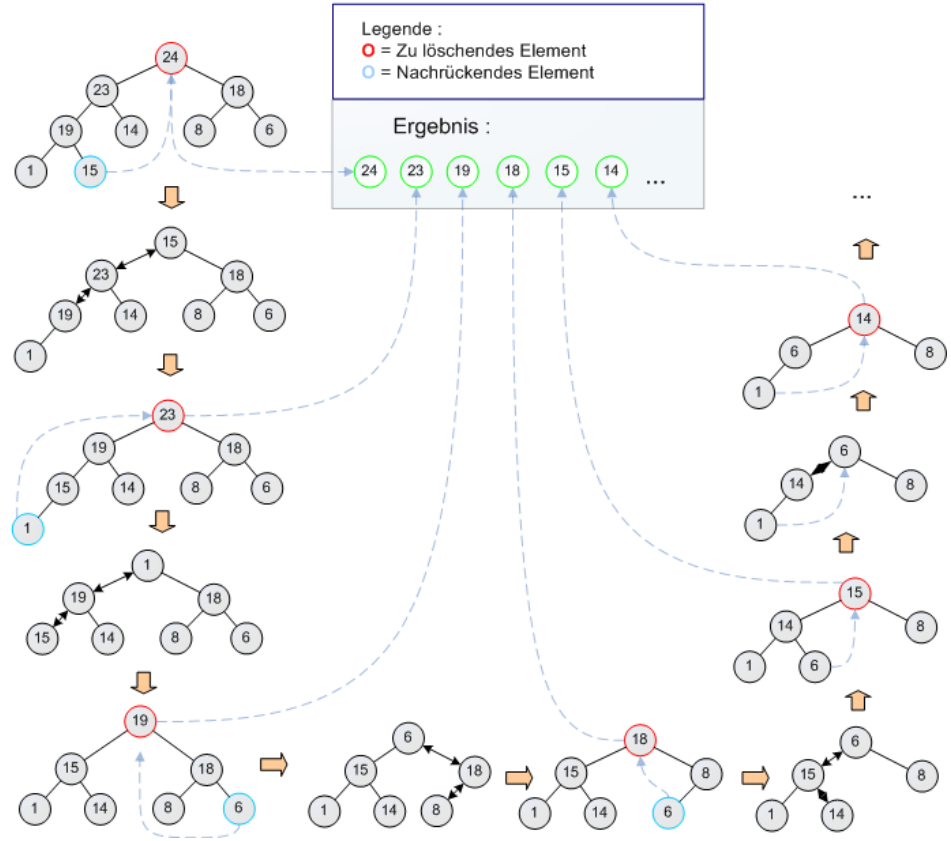
Selektionsphase am Beispiel (Forts.) Rechts vom Strich sortiert

```
A P O E H|R S T     Wir versickern A.
^ ^ ^               P>O und P>A.
P A O E H|R S T     Wir versickern A weiter. 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 versickern A.
^ ^ ^               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 versickern E.
^ ^ ^               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 versickern A. 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 Versenke(A, i, r) 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.

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

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

Ein Summationstrick

Wie berechnet man:

$$\sum_{h=0}^{\infty} hx^h ?$$

Bekannt:

(1) $\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$ für $x \in [0,1)$.

(2) Ableitung $\frac{dx^h}{dx} = hx^{h-1}$

(3) “Summe der Ableitungen gleich Ableitung der Summe” (unter gewissen Bed.)

$$\sum_{h=0}^{\infty} hx^h = x \sum_{h=0}^{\infty} hx^{h-1} = x \cdot \sum_{h=0}^{\infty} \frac{dx^h}{dx} = x \cdot \frac{d \sum_{h=0}^{\infty} x^h}{dx} = \frac{x}{(1-x)^2}$$

Varianten & Histörchen

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.

Smoothsort: Normales Heapsort sortiert bereits weitgehend vorsortierte Felder nicht schneller als andere. Die größten Elemente müssen immer erst ganz nach vorn an die Spitze des Heaps wandern, bevor sie wieder nach hinten kopiert werden. Smoothsort ändert das. Die Originalarbeit von E.W. Dijkstra finden Sie unter: `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF`.