Kap. 3: Sortieren (3)

Professor Dr. Petra Mutzel
Lehrstuhl für Algorithm Engineering, LS11
Fakultät für Informatik, TU Dortmund

6. VO DAP2 SS 2009 30. April 2009

Überblick

- Quick-Sort
- Analyse von Quick-Sort
- Quick-Sort Varianten

- Heap-Sort

Termine für Übungstests

- **1. Übungstest:**
- Di **19. Mai**, in der Vorlesung im AudiMax, Beginn: ab 12:15 Uhr, ab 12 Uhr anwesend sein wegen Sitzordnung

- **2. Übungstest:**
- Di **16. Juni**, in der VO im AudiMax, Beginn: ab 12:15 Uhr, ab 12 Uhr anwesend sein wegen Sitzordnung

Motivation

„Warum soll ich hier bleiben?"

Wir lernen Quick-Sort

„Was ist daran denn besonders?"

Quick-Sort gehört zu den „Top 10" wichtigsten Algorithmen

Analyse von QuickSort: Average

- Grundannahmen
  - alle Schlüssel sind verschieden
  - o.B.d.A. $A[i] \in \{1,2,\ldots n\}$
  - alle Permutationen sind gleich wahrscheinlich

Jede Zahl $k \in \{1,2,\ldots,n\}$ tritt mit gleicher Wahrscheinlichkeit $1/n$ an Position n auf

- Pivotelement k erzeugt zwei Folgen der Längen k-1 und n-k
- Diese Folgen sind wieder zufällig: gilt

Analyse von QuickSort: Average

- Teilt man sämtliche Folgen mit n Elementen mit dem Pivotelement (an der letzten Stelle), so erhält man sämtliche Folgen der Länge k-1 und n-k.

Rekursionsgleichung der Laufzeitfunktionen:

$$T(n) \leq \begin{cases} a, & \text{für } n = 1 \\ \frac{1}{n}\sum_{k=1}^{n}(T(k-1)+T(n-k))+bn & \text{für } n \geq 2 \end{cases}$$

wobei die Summe über alle n möglichen Aufteilungen läuft und bn der Aufteilungsaufwand für eine Folge der Länge n ist.

Analyse von QuickSort: Average

Einsetzen von T(0)=0:

$$T(n) \leq \begin{cases} a, & \text{für } n = 1 \\ \frac{2}{n}\sum_{k=1}^{n-1} T(k) + bn & \text{für } n \geq 2 \end{cases}$$

Lösung der Rekursionsgleichung:

$T(n)=\Theta(n \log n)$

Beweis von Average-Case

- Wir zeigen: $T(n)=O(n \log n)$
- $T(n)=\Omega(n \log n)$ ähnlich (o.Bw.)

Wir zeigen durch vollständige Induktion:
es existieren $c, n_0>0$, so dass für alle $n \geq n_0$:
$T(n) \leq cn \log n$

Wir wählen: $n_0=2$.

Induktionsanfang: $n=2$: $T(2)=T(1)+bn=a+2b$

somit ist $T(2) \leq c2 \log 2$ g.d.w. $c \geq (a+2b)/2=a/2+b$

Beweis von Average-Case ff

Induktionsschluss: Sei nun $n \geq 3$, Beh. gilt für alle kleineren Werte:

$$\begin{aligned} T(n) &\leq \frac{2}{n}\sum_{k=1}^{n-1} T(k) + bn \\ &\leq \frac{2}{n}\sum_{k=1}^{n-1} (ck \log k) + bn \\ &\leq \ldots \leq cn \log n - \frac{c}{4}n - \frac{c}{2} + bn \\ &\leq cn \log n \end{aligned}$$

Diese Ungleichung gilt, falls $c \geq 4b$

Wähle insgesamt $c=\max\{a/2+b, 4b\}$

Eigenschaften von QuickSort

- **Anzahl der Schlüsselvergleiche:**
  $C_{best}(n)=C_{avg}(n)=\Theta(n \log n)$; $C_{worst}(n)=\Theta(n^2)$
- **Eigenschaften:**
  - in situ? $\Theta(n)$ zusätzlichen Speicher
  - adaptiv?
  - stabil?

- sehr gut in der Praxis
- es existieren viele Varianten

Quicksort - Varianten

- Randomisierter Quicksort
- Quicksort mit $\log(n)$ Speicher

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

Randomisierter Quick-Sort

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

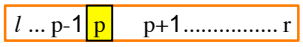
Aber: Natürlich kann der Worst-Case noch eintreten!

Man berechnet eine **Erwartete Laufzeit**
= Erwartungswert der Laufzeit ≈
durchschnittliche Laufzeit → Average-Case

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

Quick-Sort mit log(n) Speicher

Bisher: O(n) Speicher



Idee: Lange Teile sofort bearbeiten
D.h.: Rekursiver Aufruf nur für kleinere Teile.
→ diese Teile sind ≤ Hälfte der betrachteten Länge
→ Tiefe der rekursiven Aufrufe: O(log n)

Quick-Sort mit log(n) Speicher

Idee: Lange Teile sofort bearbeiten

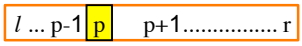
bisher

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


Quick-Sort mit log(n) Speicher



Zwischenschritt

```
(1) procedure QS(ref A,l,r)
(2) while l < r do {
(3)     p := 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 := 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: $O(n^2)$ einfach ☺
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!

Übersicht

Heap-Sort
Einführung in Datenstruktur Heap
Analyse von Heap-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)

Heap: eine neue Datenstruktur!

Speicherung…

…weiterhin als Array (in-situ!)

Interpretation…

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

Definition Heap

Eine Folge $H=<k_1,k_2,\ldots,k_n>$ von Schlüsseln ist ein **(Max-) Heap**, wenn für jede Position i die folgende **Heap-Eigenschaft** erfüllt ist:

Falls $2i \le n$, so gilt $k_i \ge k_{2i}$ und falls $2i+1 \le n$ so gilt $k_i \ge k_{2i+1}$.

Beispiel: $F=<8,6,7,3,4,5,2,1>$ (Positionen 1 2 3 4 5 6 7 8)

Ähnlich sind **Min-Heaps** definiert. Dort gilt:

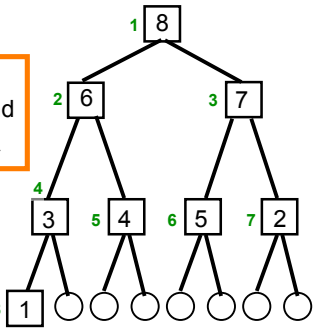
Falls $2i \le n$, so gilt $k_i \le k_{2i}$ und falls $2i+1 \le n$ so gilt $k_i \le k_{2i+1}$.

Heap: Interpretation

$F=<8,6,7,3,4,5,2,1>$

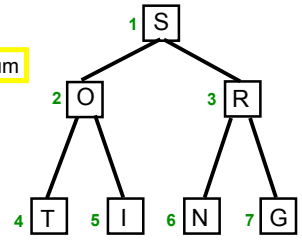
Heap-Eigenschaft:
Falls $2i \le n$, so gilt $k_i \ge k_{2i}$ und falls $2i+1 \le n$ so gilt $k_i \ge k_{2i+1}$.

Heap–Eigenschaft:
➔Schlüssel der Elternknoten ist größer gleich denen seiner Kinder



Heap: Interpretation

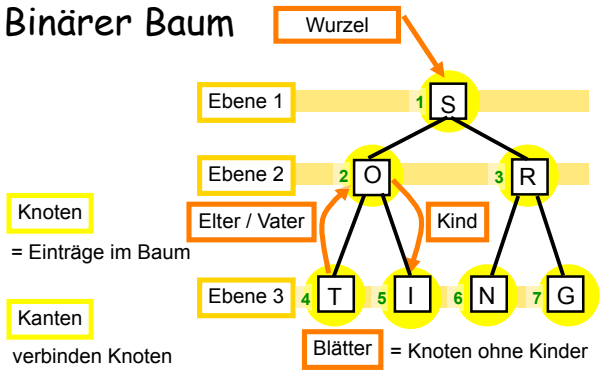
Interpretation als Binärer Baum



Speicherung als Array

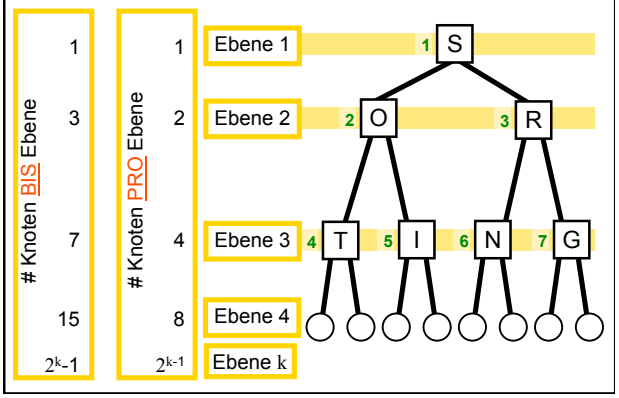
1	2	3	4	5	6	7
S	O	R	T	I	N	G

Binärer Baum

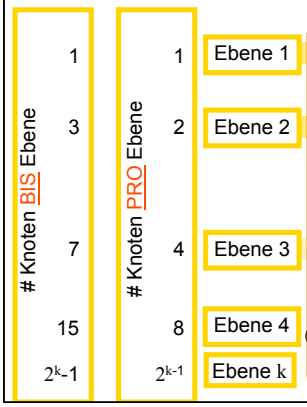


Binärer Baum: maximal 2 Kinder pro Knoten

Tiefe/Höhe des Baums



Tiefe/Höhe des Baums



Tiefe/Höhe eines Baums = Anzahl der Ebenen - 1

Gegebene Höhe *h*:

$n_{max} = 2^{h+1}-1$

$n_{min} = 2^h$

Maximale Höhe bei n Elementen?

$n = 2^h \rightarrow h_{max} = \log_2 n$

→Höhe des Heaps: $\lfloor \log n \rfloor$

Heap-Eigenschaft

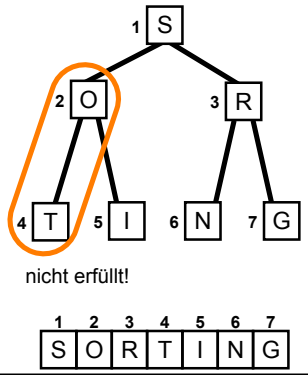
Heap–Eigenschaft:
➔Schlüssel der Elternknoten ist größer gleich denen seiner Kinder

Array A[1…n]

∀ Indizes i:

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

falls diese Indizes existieren



Idee von Heap-Sort

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

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

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

CreateHeap

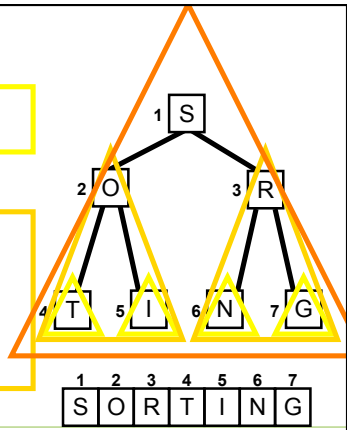
Beobachtung

Die Heapbedingung ist an den Blättern erfüllt.

Algorithmus

Betrachte Knoten über den Blättern → repariere Teilheap

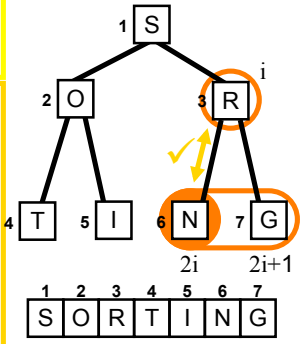
Bearbeite Baum von unten nach oben → Teilbäume sind immer Heaps



CreateHeap

```
procedure CreateHeap () {
  for i := ⌊n/2⌋ … 1 {
    SiftDown (i, n)
} }
```

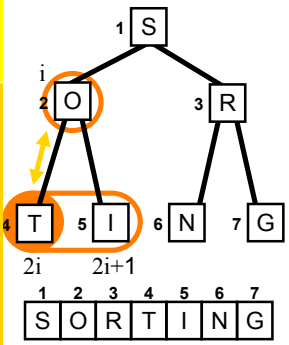
```
procedure SIFTDOWN (i, m) {
  while Knoten i hat Kinder ≤ m
  {
    j := Kind ≤ m mit größerem
  Wert
    if A[i] < A[j] then {
      vertausche A[i] und A[j]
      i := j
    } else return
  } }
```



CreateHeap

```
procedure CREATEHEAP () {
  for i := ⌊n/2⌋ … 1 {
    SIFTDOWN (i, n)
} }

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

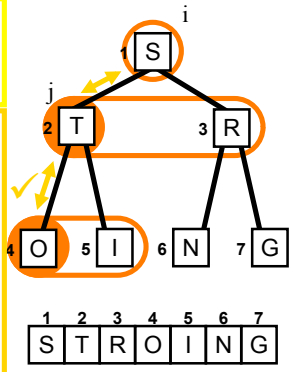


CreateHeap

➔Heap-Erstellung beendet

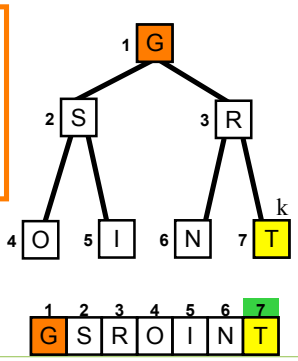
```
procedure CREATEHEAP () {
  for i := ⌊n/2⌋ … 1 {
    SIFTDOWN (i, n)
} }

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



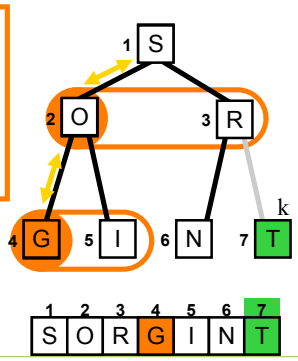
HeapSort

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



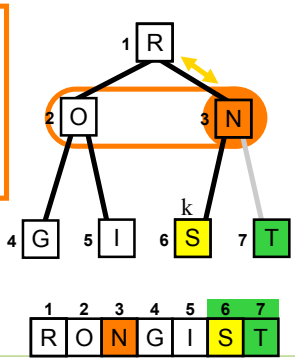
HeapSort

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



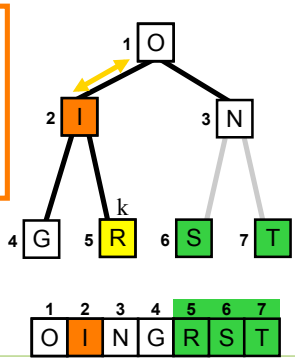
HeapSort

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



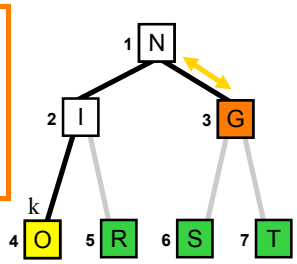
HeapSort

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



HeapSort

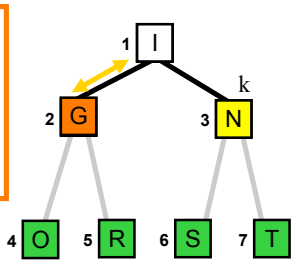
```
procedure HEAPSORT () {
   CREATEHEAP ()
   for k := n … 2 {
      vertausche A[1] und A[k]
      SIFTDOWN (1, k-1)
} }
```



1	2	3	4	5	6	7
N	I	G	O	R	S	T

HeapSort

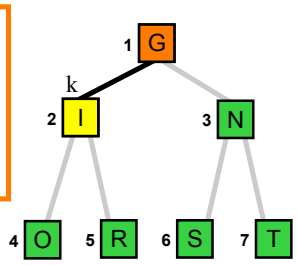
```
procedure HEAPSORT () {
   CREATEHEAP ()
   for k := n … 2 {
      vertausche A[1] und A[k]
      SIFTDOWN (1, k-1)
} }
```



1	2	3	4	5	6	7
I	G	N	O	R	S	T

HeapSort

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



1	2	3	4	5	6	7
G	I	N	O	R	S	T