

Kap. 3 Sortieren

3.1.5 HeapSort
3.1.6 Priority Queues

Vorlesung am Do 7.5.
entfällt wegen FVV
um 14 Uhr



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

7. VO DAP2 SS 2009 5. Mai 2009

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?“

Heap als Datenstruktur

„Warum soll mich das interessieren?“

Dies wird später immer wieder benötigt...

Überblick

- Wiederholung HeapSort
- Analyse HeapSort
- ADT Priority Queue

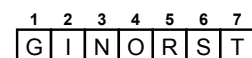
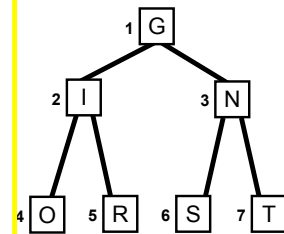
Wdhlg: Animation von Heap-Sort



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  
  }  
}
```

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



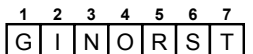
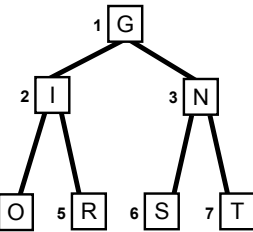
Analyse CreateHeap

```

procedure CREATEHEAP () {
  for i := [n/2] ... 1 {
    SIFTDOWN (i, n)
  }
}
    
```

Schleife:
 $O(n)$ mal
 Pro Schleife:
 $O(\log n)$
 $\rightarrow O(n \log n)$

Es gilt sogar: $O(n)$ (jetzt Beweis)



Für Beweis: Wdhlg. Höhe des Baums

# Knoten BIS Ebene	1	1	Ebene 1	Tiefe/Höhe eines Baums = Anzahl der Ebenen - 1
	3	2	Ebene 2	
	7	4	Ebene 3	
	15	8	Ebene 4	
	$2^k - 1$	2^{k-1}	Ebene k	

Gegebene Höhe h :
 $n_{\max} = 2^{h+1} - 1$
 $n_{\min} = 2^h$

Maximale Höhe bei n Elementen?
 $n = 2^h \rightarrow h_{\max} = \log_2 n$
 \rightarrow Höhe des Heaps: $[O(\log n)]$

Analyse CreateHeap

Lemma: CREATEHEAP() läuft in Zeit $O(n)$ für n Elemente.

Beweis: Sei k die Anzahl der Ebenen des Binärbaums, d.h. $2^{k-1} \leq n \leq 2^k - 1$.

- Auf Ebene j sind $\leq 2^{j-1}$ Schlüssel
- Anzahl Vergleiche für SIFTDOWN() von Ebene j ist im worst case proportional zu $k-j$
- Insgesamt:

$$T(n) = \sum_{j=1}^{k-1} 2^{j-1} (k-j) = \sum_{j=1}^{k-1} j 2^{k-j-1} = 2^{k-1} \sum_{j=1}^{k-1} \frac{j}{2^j} \leq 2^{k-1} \cdot 2 \leq 2n$$

(denn $\sum_{j=1}^{\infty} \frac{j}{2^j} = 2$)

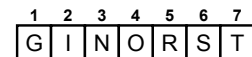
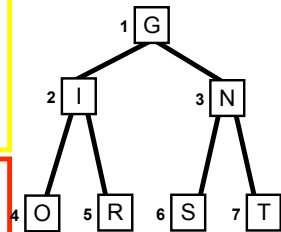
Analyse HeapSort

```

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

CreateHeap: $O(n)$
 Schleife: $O(n)$ mal
 Pro Schleife: $O(\log n)$

Insgesamt: $\rightarrow O(n \log n)$



Best Case Analyse

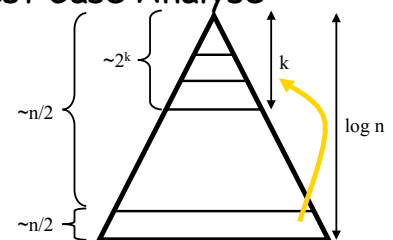
Intuition

Annahme: linear? SIFTDOWN() nur konstant (k) viele Schritte?

- Bei Beendigung von SIFTDOWN() muss $A[i] \geq A[j]$
- $A[i]$ wurde aber gerade erst von unten getauscht
- um "oben" auf dem Heap zu bleiben muss es groß sein
- Kann es sein, dass alle großen Elemente oben auf dem Heap bleiben können?
- Nein, denn "oben" ist zu wenig Platz (die meisten Elemente sind unten)

Best Case Analyse

$\Omega(n \log n)$



Annahme: linear?

- \rightarrow SIFTDOWN() nur konstant (k) viele Schritte
- \rightarrow zu wenig Platz für alle Blätter!

Analyse HeapSort

- Anzahl der Schlüsselvergleiche:**

$$C_{\text{worst}}(n) = C_{\text{best}}(n) = C_{\text{avg}}(n) = \Theta(n \log n)$$

- Anzahl der Datenbewegungen:**

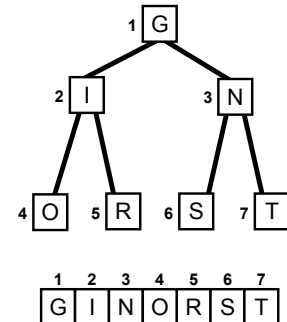
$$C_{\text{worst}}(n) = C_{\text{best}}(n) = C_{\text{avg}}(n) = \Theta(n \log n)$$

Eigenschaften

in-situ? 😊

stabil? ⚡

adaptiv? ⚡



3.1.6 Der ADT Priority Queue

„Prioritätswarteschlange, (PQ)“

- Wertebereich:** Multi-Menge von Paaren (p,v) aus $P \times V$, wobei P eine linear geordnete Menge von Prioritäten und V eine beliebige Menge von Werten (value) bezeichnet.

Operationen des ADT Priority Queue

- Allgemeine Operationen:**

- Einfügen eines Elements (INSERT)
- Löschen eines Elements an Stelle pos (DELETE)
- ISEMPY
- Gib Priorität von Element an pos zurück (PRIORITY)
- Minimum Ausgabe (ACCESSMIN)
- Minimum Extrahieren (d.h. Minimum ausgeben und dann entfernen (EXTRACTMIN)
- DECREASEPRIORITY: Verringere Priorität eines Elements an Position pos

Realisierung des ADT Priority Queue

- mittels eines Heaps (MinHeap) in Array**

- Operationen (HIER speziell):**

- Initialisieren
- Minimum Ausgabe (AccessMin): Wurzel
- Minimum Entfernen (DeleteMin): kopiere letztes Element an Wurzel, SIFTDOWN(1)
- Einfügen eines Elements Insert(p,v): füge hinten ein, Aufruf von SIFTUP(n)
- DecreasePriority(pos,p): ändere prio, SIFTUP(pos)

Repräsentation einer Min-Heap PQ

HeapElement

- prioType priority // Priorität des Elements
- valueType value //Wert (value)
- int index //Index des Elements im Feld

Repräsentation einer MinHeap PQ

- int heapSize
- HeapElement A[1..n]

Initialisierung: heapSize=0

Repräsentation einer Min-Heap PQ

```
Function AccessMin(): HeapElement  
return A[1]
```

```
Procedure DeleteMin()  
A[heapSize].index = 1  
kopiere A[heapSize] nach A[1]  
heapSize = heapSize - 1  
SIFTDOWN(1, heapSize)
```

```
Procedure DecreasePriority(pos, p)  
pos.priority = p //prüfe vorher, ob p wirklich kleiner  
SIFTUP(pos.index)
```

INSERT bei Min-Heap PQ

```
Function INSERT(p, v): HeapElement  
var HeapElement x  
heapSize = heapSize + 1  
x = new HeapElement  
x.priority = p  
x.value = v  
x.index = heapSize  
A[heapSize] = x  
SIFTUP(heapSize)  
return x
```

DELETE bei Min-Heap PQ

```
Function DELETE(pos)  
A[heapSize].index = pos.index  
Tausche A[heapSize] und A[pos.index]  
heapSize = heapSize - 1  
i = pos.index  
SIFTUP[i] //es ist entweder NUR SIFTUP  
SIFTDOWN[i] //oder SIFTDOWN notwendig  
delete pos
```

SIFTDOWN bei MinHeap PQ

```
Procedure SIFTDOWN (i, m) {  
  while Knoten i hat Kinder  $\leq m$  {  
    j := Kind  $\leq m$  mit kleinerem Wert  
    if A[i].priority > A[j].priority then {  
      A[j].index = i // weiß immer, an welcher Stelle  
      A[i].index = j // Element im Heap ist  
      vertausche A[i] und A[j]  
      i := j  
    } else return  
  } }
```

SIFTUP bei MinHeap PQ

```
Procedure SIFTUP (i) {  
  var int parent  
  parent = i/2 //abgerundet  
  while parent > 0 {  
    if A[parent].priority > A[i].priority then {  
      A[parent].index = i  
      A[i].index = parent  
      vertausche A[i] und A[parent]  
      i = parent  
      parent = i/2  
    } else return  
  } }
```

Analyse des ADT Priority Queue

• mittels eines Heaps (MinHeap) in Array

• Operationen:

- Initialisieren: $O(1)$
- Minimum Suchen (AccessMin): $O(1)$
- Minimum Entfernen (DeleteMin): $O(\log n)$
- Einfügen eines Elements Insert(p, v): $O(\log n)$
- DecreasePriority(pos, p): $O(\log n)$
- weitere Op: Delete(pos): s. Skript

Ende Heap-Sort

Bedeutung der asymptotischen Laufzeit

	Algorithmus	Implementierung	Geschwindigkeit: ops/sec
schneller Computer	InsertionSort	$2n^2$	10000 Mio
langsamer Computer	MergeSort	$50n \log n$	100 Mio

- Sortieren von 1 Mio. Zahlen:

- InsertSort: $2(10^6)^2 \text{ops} / 10^{10} \text{ops/sec} = 200 \text{ sec}$
- MergeSort: $50 \cdot 10^6 \log 10^6 \text{ops} / 10^8 \text{ops/sec} \approx 10 \text{ sec}$
- gleiche Rechner: **200 Sekunden vs. 0,1 Sekunde**

• Wie Computer sind auch Algorithmen Technologie!

	Algorithmus	Implementierung	Geschwindigkeit: ops/sec
schneller Computer	InsertionSort	$2n^2$	10000 Mio
langsamer Computer	MergeSort	$50n \log n$	100 Mio

- Sortieren von 10 Mio. Zahlen:

- InsertSort: $2(10^7)^2 \text{ops} / 10^{10} \text{ops/sec} = 20000 \text{ sec} \approx 5,5 \text{ Std.}$
- MergeSort: $50 \cdot 10^7 \log 10^7 \text{ops} / 10^7 \text{ops/sec} \approx 120 \text{ sec}$
- gleiche Rechner: **5,5 Stunden vs. 1,2 Sekunden**