

Algorithmen & Datenstrukturen

Herbst 2018

Vorlesung 7

Problem: Sortieren eines Arrays $A[1], \dots, A[n]$

Algorithmen bisher:

↑
keys (Schlüssel)

	Vergleiche	Bewegungen
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n)$
Insertion Sort	$O(n \log n)$	$O(n^2)$

Alle diese sind "inplace", d.h. brauchen keinen Extraspeicher: Resultat ist in A .

Selection Sort noch einmal, diesmal von rechts nach links:



Selection Sort (A)

for $i = n \dots 2$

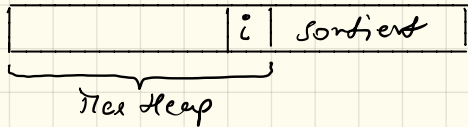
→ $j = \text{Index des Maximum in } A[1 \dots i]$
Tausche $A[i]$ und $A[j]$

das kostet $O(n^2)$ da jedes Max $O(i)$ kostet

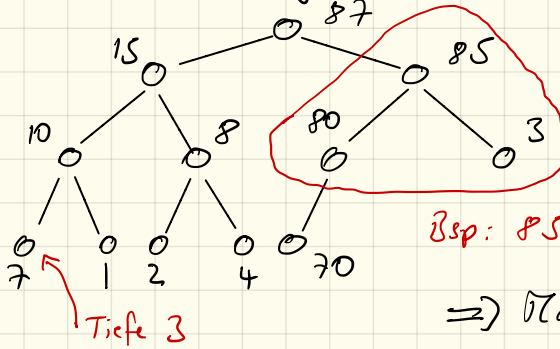
Gesucht: Datenstruktur die das Finden des Max billiger macht.

Offnung: Max in $O(\log i) \Rightarrow$ Algorithmen $O(n \log n)$!

Lösung: Max-Heap



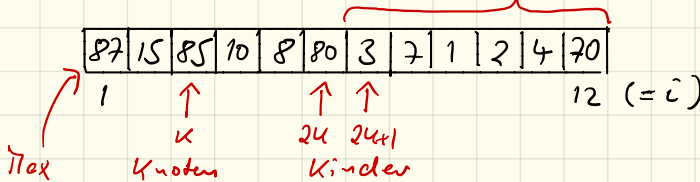
Visualisierung als Baum: (Beispiel $i=12$)



Heapbedingung:
Für alle Knoten:
Schlüssel Knoten
 $\geq$ Schlüssel Kinder
Bsp: $85 \geq 80, 3$

$\Rightarrow$ Maximum ist Wurzel

Gespeichert im Array: keine Kinder



Max-Heapen: $O(n)$!

Heapbedingung (Array)

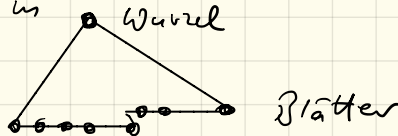
$$\forall k \in \{1, \dots, n\}: \quad 2k \leq n \Rightarrow A[2k] \leq A[k]$$

$$2k+1 \leq n \Rightarrow A[2k+1] \leq A[k]$$

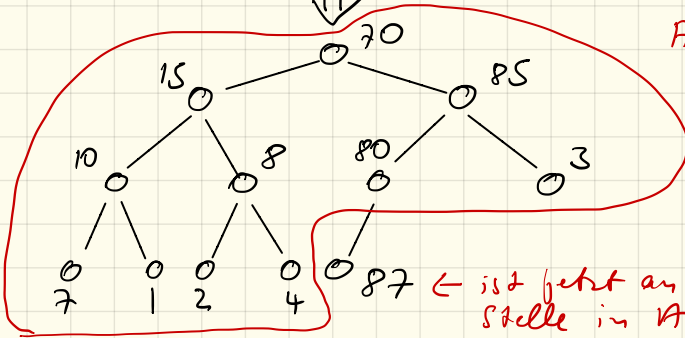
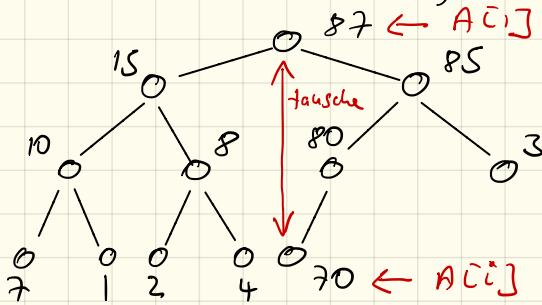
Weitere Eigenschaften des Heaps:

- grösste Tiefe: $\lfloor \log_2(n) \rfloor$
- Anzahl Blätter: $\lceil n/2 \rceil$ (die Hälfte)

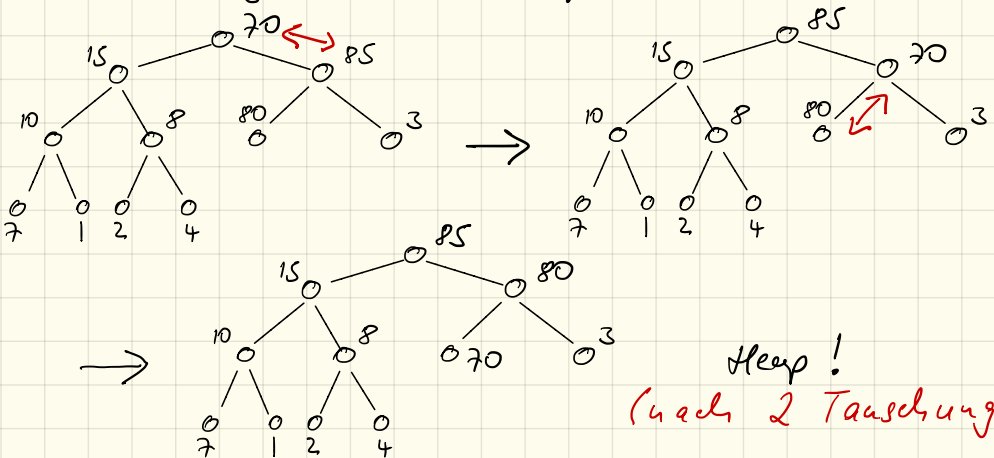
$\Rightarrow$ Voller Binarbaum



Zurück zum Sortieren, i -te Iteration:



Also: stelle Heap Bedingung wieder her
 "versichere neue Wurzel durch Tauschen
 mit grösserem Kind, bis Kinder kleiner"



Versichern: $O(\log(i))$ Vergleiche
 $O(\log(i))$ Bewegungen

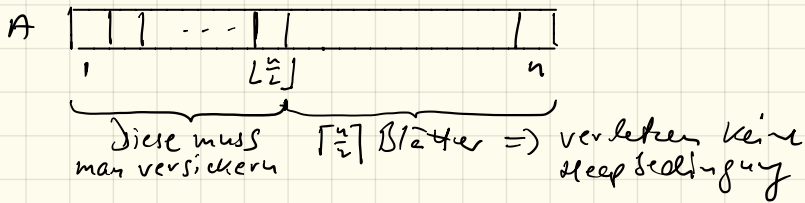
also gesamt $\sum_{i=1}^n c \cdot \log(i) \leq O(n \log n)$

↑
hätten wir letztes
Mal schon

Pseudocode:

RestoreHeapCondition(A, n, i) // versichere Element n
im Skript in $A[1..i]$

Was noch fehlt: A am Anfang in Heap verwandeln



HeapSort(A)

for $i = \lfloor \frac{n}{2} \rfloor \dots 1$

RestoreHeapCondition(A, i, n)

for $i = n \dots 2$

Vertausche $A[1]$ und $A[i]$

RestoreHeapCondition($A, 1, i-1$)

} Erzeuge Heap

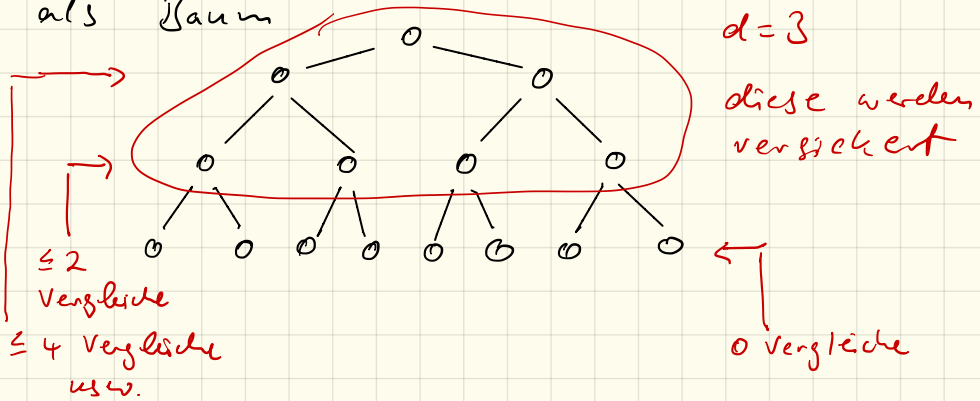
} $O(n \log n)$ s.o.

Analyse Erzeuge Heap:

$n/2$ Aufrufe, jeder $\leq O(\log n) \Rightarrow O(n \log n)$

Es gilt sogar $O(n)$: siehe Skript oder
nächste Seite

Wir nehmen an $n = 2^d - 1$ und nutzen A als Baum



Anzahl Vergleiche für "Erzeuge Heap" höchstens

$$0 \cdot 2^d + 2 \cdot 2^{d-1} + 4 \cdot 2^{d-2} + 6 \cdot 2^{d-3} + \dots + 2d \cdot 2^{d-d}$$

$$= 2 \sum_{i=0}^{d-1} i \cdot 2^i = 2d(2^d - 1) - 2 \sum_{i=0}^{d-1} i \cdot 2^i$$

Das folgende habe ich nicht gezeigt:

Benutze: $\sum_{i=0}^{d-1} i x^i = \frac{x - dx^d + (d-1)x^{d+1}}{(1-x)^2}, x \neq 1$

woher kommt das?

Nimm $\sum_{i=0}^{d-1} x^i = (1-x^d)/(1-x)$ und leite auf beiden Seiten ab

$$= 2d(2^d - 1) - 2(2 - d2^d + (d-1)2^{d+1})$$

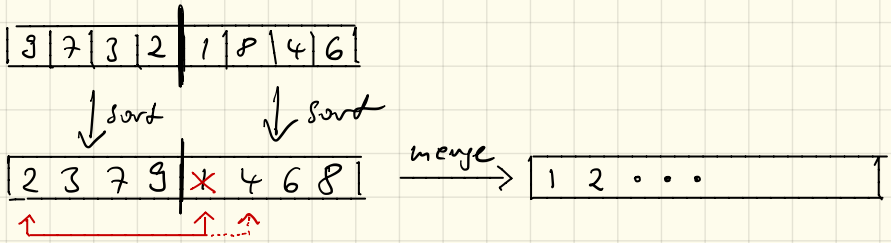
$$= 4 \cdot 2^d - 2d - 4 \leq O(n)$$

Pro 2 Vergleiche ≤ 1 Vertauschung $\Rightarrow O(n)$ Vertauschungen

HeapSort: $+ O(n \log n)$
 $+ \text{Inplace (Extraplatz } O(1))$
 $- \text{schlechte Lokalität}$
 $(= \text{es wird viel hin- und her gehüpft})$

Algorithmus 5: Mergesort

Idee: Divide-and-Conquer



das nächstkleinste
 ist immer eines der
 beiden ersten (lasse 2 Zeiger wandern)

Mergesort(A, left, right) // Anfang: left=1, right=n
 if left < right

middle = $\lfloor (left + right) / 2 \rfloor$

Mergesort(A, left, middle)

Mergesort(A, middle+1, right)

Merge(A, left, middle, right)

Den Algorithmus
 kann man
 iterativ
 implementieren:
 Straight
 Merge Sort
 im Skript

Beispiel: $[3, 7, 3, 2, 1, 8, 4, 6]$ merge
 $[7, 9, 2, 3, 1, 8, 4, 6]$ merge
 $[2, 3, 7, 9, 1, 4, 6, 8]$ merge
 $1, 2, 3, 4, 6, 7, 8, 9$

Merge (A, l, m, r)

$i = l$ // Index in A links

$j = m+1$ // Index in A rechts

$k = l$ // Index in B

while $i \leq m$ and $j \leq r$

if $A[i] < A[j]$

$B[k] = A[i]$

$i = i+1$

$k = k+1$

else

$B[k] = A[j]$

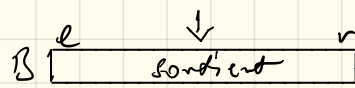
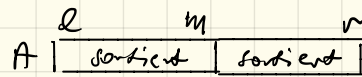
$j = j+1$

$k = k+1$

übernimm Rest links

" " rechts

kopiere B zurück nach A



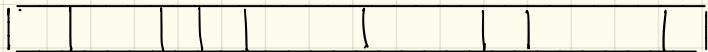
braucht Hilfsarray

} nur ein Fall tritt ein

Laufzeit: Vergleiche $T(n) = 2T(\frac{n}{2}) + cn \leq O(n \log n)$
 Bewegungen " $O(n \log n)$
 Extraplatz $O(n)$

Variante: Natural Mergesort (siehe Skript)

1.) Finde sortierte Teilstücke:



2.) Merge

usw.

schon sortiert

Algorithmus 6: Quicksort

Mergesort

teile Array
sortiere links
sortiere rechts
verschmelze

↪ Ansatz + Extraplate ist hier

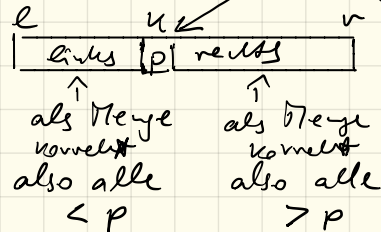
Invariante:

l	m	r
sortierte linke Hälfte		sortierte rechte Hälfte

Idee: schiebe die Array ins Aufteilen

- 1.) Aufteilen und Array
- 2.) Rekursion links und rechts
- 3.) Verschmelzen nicht notwendig

Invariante an richtiger Stelle



Quicksort(A, l, r)

if $l < r$

$k = \text{Aufteilen}(A, l, r)$ // wählt ein Element p

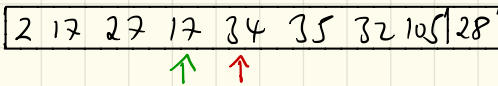
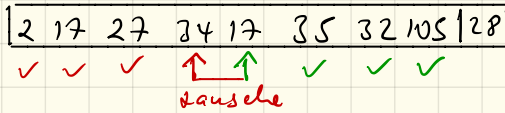
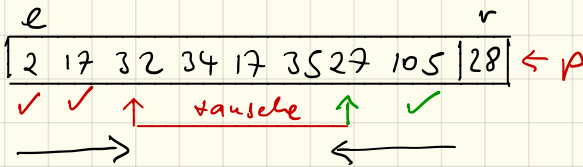
Quicksort($A, l, k-1$)

Quicksort($A, k+1, r$)

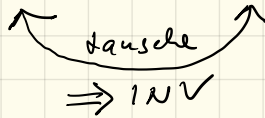
hat es an richtiger
Stelle k index
es INV herstellt

$p = \text{Pivotelement, z.B. } p = A[r]$

Aufteilen:



stop wenn
grün $\leq$ rot:



Aufteilen(A, l, r) // $l < r$

$i = l$

$j = r - 1$

$p = A[r]$

repeat

while $i < r$ and $A[i] < p$: $i = i + 1$

while $j > l$ and $A[j] > p$: $j = j - 1$

if $i < j$: tausche $A[i], A[j]$

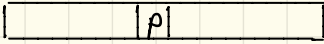
until $i \geq j$

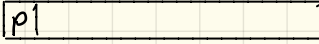
tausche $A[i], A[r]$

return i

// jetzt ist Pivot am
richtigen Platz

Laufzeit: hängt davon ab wo Pivot landet

gut: 

schlecht: 

$$\text{gut: } T(n) = 2T\left(\frac{n}{2}\right) + cn \leq O(n \log n)$$

$$\text{schlecht: } T(n) = T(n-1) + cn \leq O(n^2)$$

Trotzdem wird Quicksort viel verwendet
Warum?

Worst-case ist selten!

z.B. best/worst case alternierend $\Rightarrow O(n \log n)$

Quicksort ist $O(n \log n)$ im average case
(neue Analyse, machen wir nicht)

Unglücklicherweise: schon sortiert ist worst case

Häufig wird das Pivotelement zufällig
gewählt (randomisiertes Quicksort)

Randomisierte Algorithmen: nächstes Semester

Komplexität vergleichsbasiertes Sortieren

geht es mit weniger als $\Theta(n \log n)$ Vergleichen?

Idee: Jeder Algorithmus entspricht einem Entscheidungsbaum (ähnlich Suche)

Höhe (oder Tiefe)

0

$a_i < a_j$

ja

nein

1

$a_i < a_e$

ja

nein

ja

$a_i < a_v$

ja

nein

2

... etc. ...

was i und j ist hängt von Algorithmus ab, aber jeder fängt irgendwo an

Blätter $\leftrightarrow$ mögliche Sortierungen ($n!$ viele)

Laufzeit (worst case) $\leftrightarrow$ Höhe Baum h

Baum der Höhe h hat $\leq 2^h$ Blätter, also muss

$$2^h \geq n!$$

$$\Leftrightarrow h \geq \log_2(n!) \geq \Omega(n \log n)$$

haben wir letztes Mal

Also Komplexität vergleichsbasiertes Sortieren ist $\Theta(n \log n)$.

Vorlesung GeSS 1:	Algorithmen	Datenstrukturen
Entwurf	✓	heute
Analyse	✓	heute

Datenstrukturen für abstrakte Datentypen (ADTs)

ADT: Objekte
Operationen

Beispiel: Objekte = Schlüssel $\in \mathbb{N}$

Beispiel: Studentenlegen Setze
Schlüssel = Matrikelnummer

Datenstruktur =
Implementierung eines ADTs
Ziel: Effizienz

Welche ADTs haben wir bisher gesehen?

ADT	Datenstruktur
Stack (Stach)	
Schlange (Queue)	
Priority Queue	
Graph	Heap
	Adjazenzliste
	Adjazenzmatrix