Problem: Sortieren eines Arrays $A[1..n]$

Algorithmen bisher:

	Vergleiche	Bewegungen	Extra-Platz	Lokalität
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$	gut
Selection Sort	$O(n^2)$	$O(n)$	$O(1)$	gut
Insertion Sort	$O(n \log n)$	$O(n^2)$	$O(1)$	gut
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	gut

in der Regel am wichtigsten

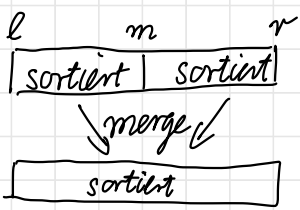
kann verbessert werden, ist aber kompliziert

Algorithmen springen nicht wild im Speicher hin und her

Erinnerung:

Merge Sort: teile Array
sortiere links
sortiere rechts
merge

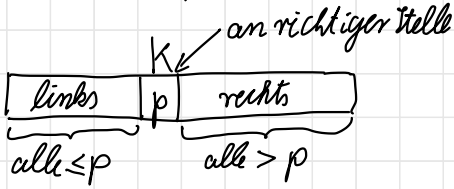
Arbeit + Extraplatz ist hier



Idee: Entscheide erst, welche Elemente nach links bzw rechts gehören.

Quicksort:

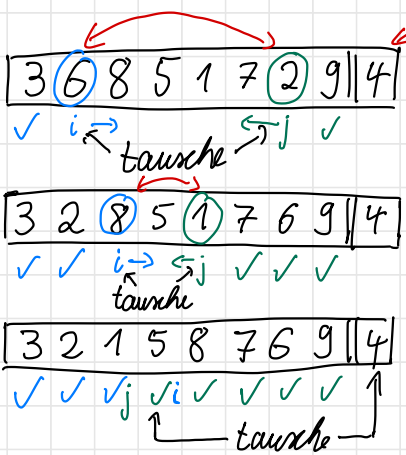
1) Wähle ein Element p ("Pivotelement", z.B. $p := A[n]$)
2) Finde korrekte Position für p
3) Schaffe alle Elemente $\leq p$ nach links und alle Elemente $> p$ nach rechts
4) Sortiere rekursiv linken und rechten Teil



Verschmelzen nicht notwendig

```
QuickSort(A, l, r)
    if l < r
        k = Aufteilen(A, l, r)   // räumt A[l...r] so um, dass
                                 //   A[i] ≤ A[k] für i = l...k-1
                                 //   A[i] > A[k] für i = k+1...r
        QuickSort(A, l, k-1)
        QuickSort(A, k+1, r)
```

Aufteilen: Wähle Pivot, z.B. $p = A[r]$



Bewege i nach rechts und j nach links, bis wir ein Paar zum Tauschen finden

Am Ende (wenn $i > j$) tausche Pivot an richtige Stelle $i = j+1$

```
Aufteilen(A, l, r)   // l < r
    i := l
    j := r - 1
    p := A[r]
    repeat
        while i < r and A[i] ≤ p do i := i + 1
        while j ≥ l and A[j] > p do j := j - 1
        if i < j: tausche A[i] mit A[j]
    until i > j
    tausche A[i] mit A[r]   // am Ende ist i die richtige Stelle für Pivot
    return i
```

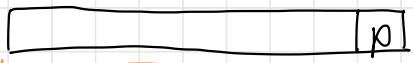
für den Extremfall, dass p das Maximum oder Minimum in $A[l..r]$ ist

Laufzeit von Aufteilen $\leq O(r-l) \leq O(n)$

Laufzeit: Rekursion hängt davon ab, wo Pivot landet

gut: 

$T(n) \leq 2 \cdot T(\frac{n}{2}) + cn \leq O(cn \log n)$

schlecht: 

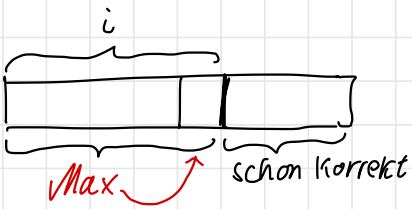
$T(n) = T(n-1) + cn \leq O(n^2)$

z.B. wenn A schon sortiert war.

Lösung: Wenn wir p zufällig wählen, sind wir im guten Fall.
↝ randomisierte Algorithmen: nächstes Semester

HEAPSORT

Erinnerung: SelectionSort

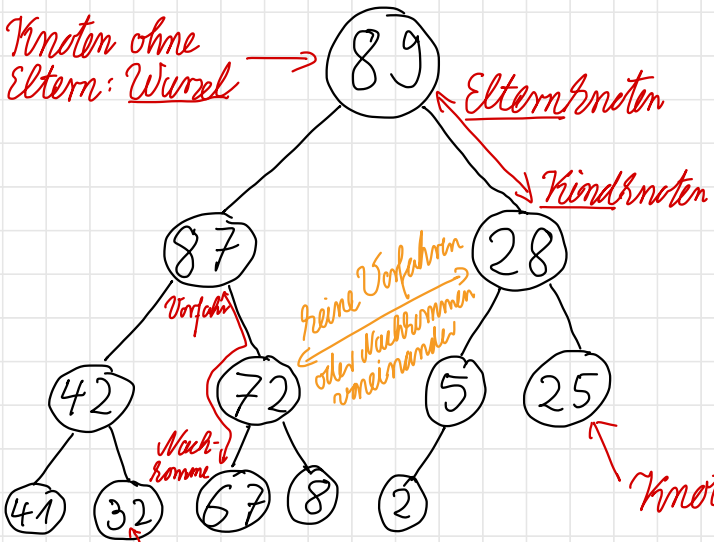
Invariante: 

Finden des Maximum kostete Zeit $O(i)$ ↝ Laufzeit $O(n^2)$

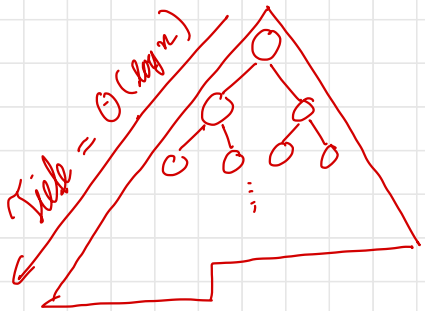
GEHT ES BESSER?

Idee: Arrangiere die Daten so, dass wir das Maximum schnell finden können → (Max-)Heap

(MAX-) HEAP: Anordnung der Daten in einem Baum



genauer: vollständiger Binärbaum:



Heapbedingung: Schlüssel (Knoten) ≥ Schlüssel (Kinder)

Induktion ⇝ Schlüssel (Vorfahr) ≥ Schlüssel (Nachkomme)

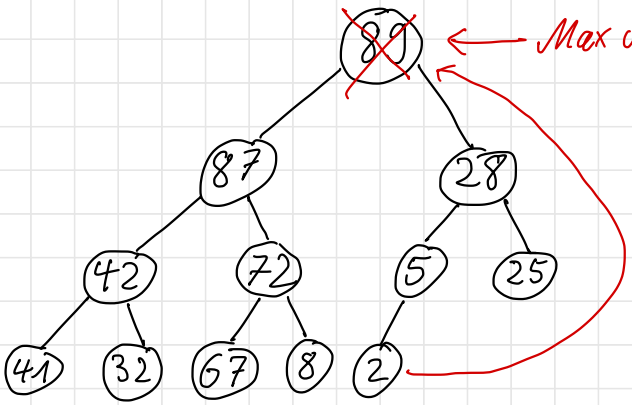
⇝ Maximum ist an der Wurzel

Zum Sortieren müssen wir zwei Operationen ausführen können:

1) Array in Heap umwandeln
2) Maximum im Heap finden und entfernen

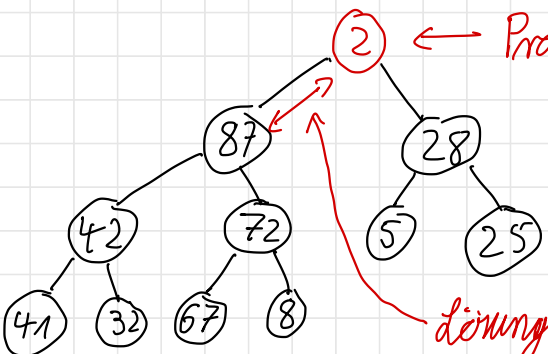
einfach

Maximum aus Heap löschen (ExtractMax)



← Max ist an der Wurzel

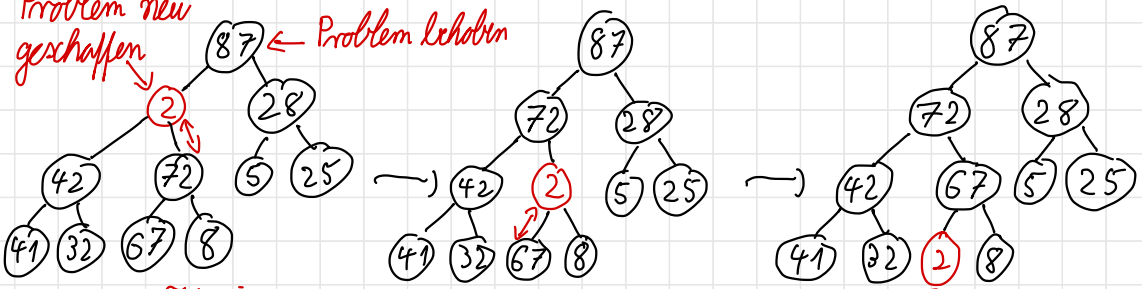
Um Baumstruktur wiederherzustellen: Ersetze Wurzel durch letztes Blatt



← Problem: Dieser Knoten erfüllt (als einziger!) die Heapbedingung nicht mehr

Lösung: Tausche den Knoten mit dem größeren Kind

Problem neu geschaffen



← Problem behoben

Lösung: „Versickere Knoten"

Problem gelöst, sobald der Knoten ein Blatt ist

Laufzeit: $O(\log n)$

Daten in Heap umwandeln, Variante 1:

- Starte mit leerem Heap
- Füge Elemente nacheinander ein

insert (H, p) // füge einen neuen Knoten mit Schlüssel p in H ein

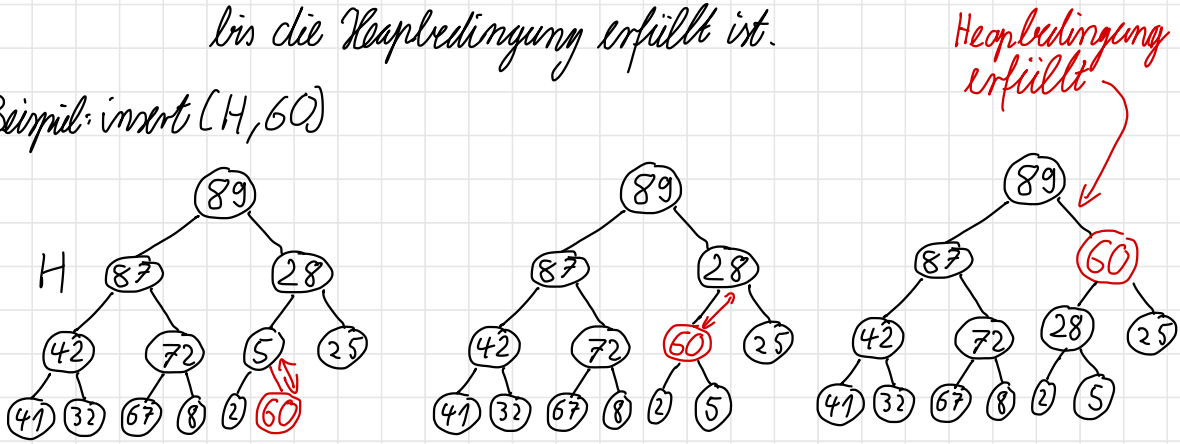
Erzeuge neuen Knoten v mit Schlüssel p // Annahme: Wir kennen die Zahl Knoten in H

an nächster freier Stelle

Vertausche v so lange mit Elternknoten,

bis die Heapbedingung erfüllt ist.

Beispiel: insert (H, 60)



Laufzeit: $O(\log n)$ pro insert

HeapSort (A)

H = empty Heap

for i = 1 ... n : insert (H, A[i])

for i = n ... 1 : A[i] ← ExtractMax (H)

Laufzeit: $n \times$ insert $\rightsquigarrow O(n \cdot \log n)$

$n \times$ ExtractMax $\rightsquigarrow O(n \cdot \log n)$

} insgesamt $O(n \log n)$

Daten in Heap umwandeln, Variante 2 (optional)

Input: Array $A[1..n]$

Ziel: Erzeuge Heap mit Schlüsseln $A[1], ..., A[n]$

CreateHeap(A)

Erzeuge vollständigen Binärbaum mit Schlüsseln $A[1], ..., A[n]$
und Tiefe $T := \lfloor \log_2 n \rfloor$ (ohne Heapbedingung)

for $t = T ... 0$:

für alle Knoten v in Tiefe t: Versickere v

Invariante: Nach Runden $T ... t$ erfüllen alle Knoten in Tiefe $\geq t$ die Heap-Bedingung.

Beweis: Induktion ("Übung")

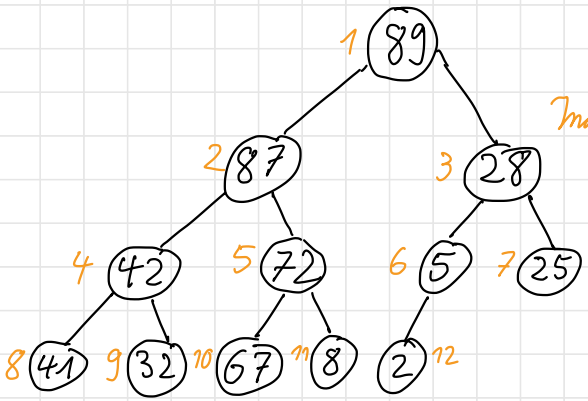
Laufzeit: $O(n \cdot \log n)$

↑ n Knoten ↑ Laufzeit für Versickern

Genauere Analyse zeigt: Laufzeit ist sogar $O(n)$ (Skript)

Darstellung eines Binärbaums im Speicher

Wir speichern die Knoten in einem Array: erst Level 1, dann Level 2, ...



Index 1 2 3 4 5 6 7 8 9 10 11 12

89	87	28	42	72	5	25	41	32	67	8	2

Beobachtung: Die Kinder von $A[k]$ stehen in $A[2k]$, $A[2k+1]$
(solange $2k, 2k+1 \leq n$)

Beweis durch Induktion nach k. (Übung)

Alle Heap-Operationen können in dem Array durchgeführt werden

Beispiel: Testen der Heap-Bedingung für Knoten $A[k]$:

```
SatisfiesHeapCondition(k)
    if 2k > n return true   // A[k] ist ein Blatt, Heapbedingung automatisch erfüllt
    if 2k = n and A[2k] ≤ A[k] return true   // nur ein Kind
    if 2k < n and A[2k] ≤ A[k] and A[2k+1] ≤ A[k] return true
    return false
```

vollständiger Pseudocode von HeapSort mit Array: siehe Skript

Vorteile / Nachteile von HeapSort

+ Laufzeit $O(n \log n)$

+ in place

- schlechte Lokalität (springt viel umher)

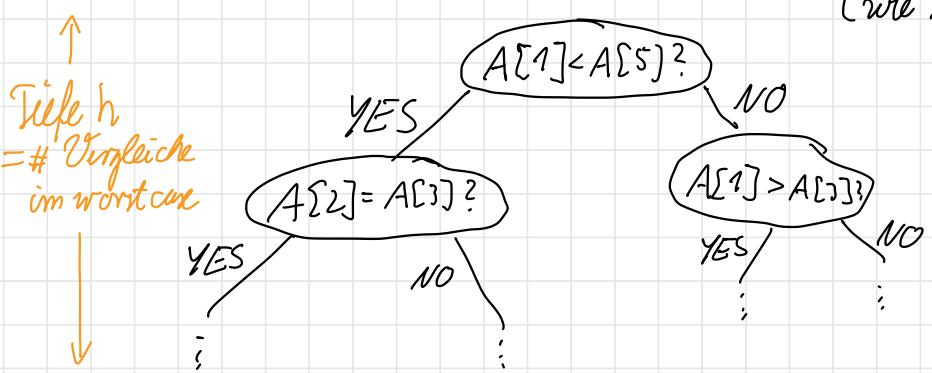
HeapSort wird in der Praxis etwas seltener verwendet als MergeSort und QuickSort

HeapSort, MergeSort und (randomisiertes) QuickSort haben Laufzeit $O(n \log n)$

GEHT ES BESSER?

Antwort: NEIN (Annahme: vergleichsbasiertes Sortieren)

Beweis: Betrachte Entscheidungsbaum für einen beliebigen Algorithmus (wie bei Suchen)



Es muss gelten: # Knoten ≥ # mögliche Outputs = $n!$

Umordnungen von A, z.B.

A(1)	A(2)	A(3)	...	A(n)
A(2)	A(1)	A(3)	...	A(n)
A(3)	A(1)	A(2)	...	A(n)
⋮				

$\rightsquigarrow 2^{h+1} \geq$ # Knoten $\geq n!$

$\rightsquigarrow h \geq \log_2(n!) - 1 \geq \Omega(n \log n)$

DATENSTRUKTUREN

Heap: geschickte Organisation von Daten erlaubt effizienteren Zugriff auf Maximum

Was, wenn wir an etwas anderem interessiert sind?
~> verschiedene Datenstrukturen

Abstrakte Datentypen (ADT)

beschreiben die Ziele: Was wollen wir mit den Daten tun?

ADT: Objekte + Operationen

Bei uns: Objekte = Schlüssel $\in \mathbb{N}$

In der Praxis hängt an einem Objekt oft ein Datensatz, bzw. ein Pointer, der angibt, wo im Speicher die Daten stehen

Beispiel: Studierendendatensätze, Schlüssel = Matrikelnummer

Datenstruktur = Implementierung eines ADT

Wie realisieren wir einen ADT im Speicher?

1. ADT Liste (Crash course, Details später in E Prog)

enthält Objekte / Schlüssel in fester Reihenfolge

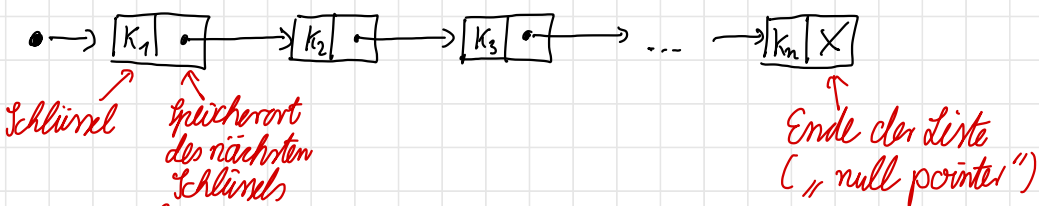
Operationen (Auswahl):

- insert(K, L) // fügt Objekt mit Schlüssel K am Ende der Liste ein
- get(i, L) // gibt i-ten Schlüssel aus
- delete(o, L) // lösche Objekt o aus Liste
- insertAfter(o, K, L) // füge Objekt mit Schlüssel K hinter o ein

Datenstrukturen für ADT Liste

Array (falls maximale Länge bekannt)

verkettete Liste (linked list)



doppelt verkettete Liste: Zeiger auf Nachfolger und Vorgänger

	insert	get	delete	insertAfter
Array	O(1)	O(1)	O(n)	O(n)
einfach verkettete Liste	O(n)	O(n)	O(n)	O(1)
doppelt verkettete Liste	O(1)	O(n)	O(1)	O(1)

sofern wir Speicherort von Objekt o kennen