

- Konstruktion der Partition: eigentliche Kunst / Arbeit bei Quicksort

1. Wahl eines Elementes W im Array (heißt **Pivot-Element**)
2. Suchen eines i von links mit $A[i] > W$
3. Suchen eines j von rechts mit $A[j] \leq W$
4. Vertauschen von $A[i]$ und $A[j]$
5. Wiederholung der Schritte bis $i \geq j-1$ gilt
6. W "dazwischen" speichern

- Resultat: Partition $A=A_1WA_2$

$\underbrace{\quad A_1 \quad}_{\leq W} \quad W \quad \underbrace{\quad A_2 \quad}_{> W}$

G. Zachmann Informatik 2 - SS 06 Sortieren 39

Algo-Animation

Partitioning in Quicksort

- How do we partition in-place efficiently?
 - Partition element = rightmost element.
 - Scan from left for larger element.
 - Scan from right for smaller element.
 - Exchange.
 - Repeat until pointers cross.

Q U I C K S O R T I S C O O L

■ partition element
 ■ unpartitioned
 ■ left
■ partitioned
 ■ right

Courtesy Kevin Wayne & Robert Sedgwick
Quicksort - Partition - Slide 1

G. Zachmann Informatik 2 - SS 06 Sortieren 40

Programm

```

def quicksort( A ):
    recQuicksort( A, 0, len(A)-1 )

def recQuicksort( A, links, rechts ):
    if rechts <= links :
        return

    # find Pivot and partition array in-place
    pivot = partition( A, links, rechts )

    # sort smaller array slices
    recQuicksort( A, links, pivot-1 )
    recQuicksort( A, pivot+1, rechts )
  
```

G. Zachmann Informatik 2 - SS 06 Sortieren 41

```

def partition( A, links, rechts ):
    i, j = links, rechts-1
    pivot = rechts

    while i < j:
        # find elem > pivot from left
        while A[i] <= A[pivot] and i < rechts:
            i += 1
        # find elem < pivot from right
        while A[j] > A[pivot] and j > links:
            j -= 1
        if i < j:
            # swap mis-placed elems
            A[i], A[j] = A[j], A[i]

    # put pivot at its right place and return its pos
    A[i], A[pivot] = A[pivot], A[i]
    return i
  
```

G. Zachmann Informatik 2 - SS 06 Sortieren 42

Partition bei der Arbeit

Pivot

a n e x a m p l e Ausgangssituation, Pivot rechts

a n e x a m p l e Suche mit **while**-Schleifen

i j

a a e x n m p l e falsche Elemente tauschen

a a e x n m p l e Suche mit **while**-Schleifen

i j

Abbruch der Suchvorgänge, Zeiger treffen sich

a a e e n m p l x Pivot richtig einsortieren

Pivot

G. Zachmann Informatik 2 - SS 06 Sortieren 43

Quicksort bei der Arbeit

LO = nach der ersten Iteration, RU = fertig.
X-Achse = Index, Y-Achse = Wert in diesem Array-Element

G. Zachmann Informatik 2 - SS 06 Sortieren 44

Korrektheit der Partitionierung

- Ann.: wähle das **letzte Element** A_r im Teil-Array $A_{l..r}$ als **Pivot**
- Bei der Partitionierung wird das Array in vier Abschnitte, die auch leer sein können, eingeteilt:
 - $A_{l..i}$ → Einträge dieses Abschnitts sind $\leq pivot$
 - $A_{j..r-1}$ → Einträge dieses Abschnitts sind $> pivot$
 - $A_r = pivot$
 - $A_{i..j}$ → Status bzgl. $pivot$ ist unbekannt
- Ist eine Schleifeninvariante

G. Zachmann Informatik 2 - SS 06 Sortieren 45

- Initialisierung:** vor der ersten Iteration gilt:
 - $A_{l..i}$ und $A_{j..r-1}$ sind leer – Bedingungen 1 und 2 sind (trivial) erfüllt
 - r ist der Index des Pivots – Bedingung 3 ist erfüllt

```

i, j = l, r-1
p = A[r]
while i < j:
    # find elem > pivot from left
    while A[i] <= p and i < r:
        i += 1
    # find elem < pivot from right
    while A[j] > p and j > l:
        j -= 1
    # swap mis-placed elems
    if i < j:
        A[i], A[j] = A[j], A[i]
[...]
```

G. Zachmann Informatik 2 - SS 06 Sortieren 46

- Erhaltung der Invariante:
 - Nach erster while-Schleife gilt: $A[i] > p$ oder $i=r$
 - Nach zweiter while-Schleife gilt: $A[j] \leq p$ oder $j=1$
 - Vor if gilt: falls $i < j$, dann ist $A[i] > p \geq A[j]$
 - was dann durch den if-Body "repariert" wird
 - Nach if gilt wieder Schleifeinvariante


```

i, j = l, r-1
p = A[r]
while i < j:
    # find elem > pivot from left
    while A[i] <= p and i < r:
        i += 1
    # find elem < pivot from right
    while A[j] > p and j > 1:
        j -= 1
    # swap mis-placed elems
    if i < j:
        A[i], A[j] = A[j], A[i]
[...]
```

G. Zachmann Informatik 2 - SS 06 Sortieren 47

- Beendigung:
 - nach while-Schleife gilt:

$$i \geq j \wedge (A_i > A_r \vee i = r)$$
 - d.h.
 - $A_{l,i} \leq pivot$
 - $A_{i+1,r} > pivot$
 - $A_r = pivot$
 - der vierte Bereich, $A_{l,j}$, ist leer, oder man weiß $A_j \geq pivot$
 - Die letzte Zeile vertauscht A_r und A_i :
 - $Pivot$ wird vom Ende des Feldes zwischen die beiden Teil-Arrays geschoben
 - damit hat man $A_{l,i} \leq pivot$ und $A_{i+1,r} > pivot$
 - Also wird die Partitionierung korrekt ausgeführt

```

i, j = l, r-1
p = A[r]
while i < j:
    [...]
A[i], A[r] = A[r], A[i]
return i
```

G. Zachmann Informatik 2 - SS 06 Sortieren 49

Laufzeit des Algorithmus

- Die Laufzeit von Quicksort hängt davon ab, ob die Partitionen ausgeglichen sind oder nicht.
- Worst-Case
 - tritt auf, wenn jeder Aufruf zu am wenigsten ausgewogenen Partitionen führt
 - eine Partition ist am wenigsten ausgewogen, wenn
 - das Unterproblem 1 die Größe $n-1$ und das Unterproblem 2 die Größe 0, oder umgekehrt, hat
 - $pivot \geq$ alle Elemente $A_{l,r-1}$ oder $pivot <$ alle Elemente $A_{l,r-1}$
 - jeder Aufruf ist am wenigsten ausgewogen, wenn
 - das Array $A_{l,n}$ sortiert oder umgekehrt sortiert ist

G. Zachmann Informatik 2 - SS 06 Sortieren 50

Worst-Case-Partitionen

- Laufzeit für Worst-Case-Partitionen bei jedem Rekursionsschritt

$$\begin{aligned}
 T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n) \\
 &= T(n-1) + \Theta(n) \\
 &= \sum_{k=1}^n \Theta(k) = \Theta\left(\sum_{k=1}^n k\right) \\
 &\in \Theta(n^2)
 \end{aligned}$$

Rekursionsbaum für Worst-Case-Partitionen

G. Zachmann Informatik 2 - SS 06 Sortieren 51

Laufzeit bei Best-Case-Partitionierung

- Größe jedes Unterproblems $\leq \frac{n}{2}$
genauer: ein Unterproblem hat die Größe $\lfloor \frac{n}{2} \rfloor$, das andere die Größe $\lceil \frac{n}{2} \rceil - 1$
- Laufzeit:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \text{PartitionTime}(n)$$

$$= 2T\left(\frac{n}{2}\right) + cn$$
- Gesamt: $T(n) \in \Theta(n \log(n))$

Rekursionsbaum für Best-Case-Partition

Gesamt: $c \cdot n \log n$

G. Zachmann Informatik 2 - SS 06 Sortieren 52

Auswahl des Pivot-Elementes

- Pivot =
 - "central point or pin on which a mechanism turns", oder
 - "a person or thing that plays a central part or role in an activity"
- optimal wäre ein Element, das A in zwei genau gleich große Teile partitioniert (Median)
- exakte Suche macht Laufzeitvorteil von Quicksort wieder kaputt
- üblich: Inspektion von drei Elementen
 - $A[l]$, $A[r]$, $A[\text{mid}]$ mit $\text{mid} = (l+r)/2$
 - wähle davon den Median (wertmäßig das mittlere der drei)
 - nennt man dann "median-of-three quicksort"
- Alternative: zufälligen Index als Pivot-Element
 - Technik: "Randomisierung"

G. Zachmann Informatik 2 - SS 06 Sortieren 53

- Beispiel, wenn man nur $A[\text{mid}]$ als Vergleichselement nimmt:

```

SORTIERBEISPIEL
SORTIER B EISPIEL
B ORTIERSEISPIEL

```

 - schlechtest mögliche Partitionierung
- A_2 weiter sortieren:

```

ORTIERSEISPIEL
ORTIER S EISPIEL
ORLIEREEIIP S ST

```
- Beispiel, wenn mittleres Element von $A[l]$, $A[r]$, $A[\text{mid}]$ als Pivot-Element verwendet wird:

```

SORTIERBEISPIEL
BEIIIEE L RTSPROS

```

G. Zachmann Informatik 2 - SS 06 Sortieren 54

Programm für Median-of-3-Quicksort

```

def median( A, i, j, k ):
    if A[i] <= A[j]:
        if A[j] <= A[k]:
            return i,j,k
        else:
            if A[i] <= A[k]:
                return i,k,j
            else:
                return k,i,j
    else:
        if A[i] <= A[k]:
            return j,i,k
        else:
            if A[j] <= A[k]:
                return j,k,i
            else:
                return k,j,i

```

G. Zachmann Informatik 2 - SS 06 Sortieren 55

```

def median_pivot( A, links, rechts ):
    middle = (links+rechts) / 2
    l,m,r = median( A, links, middle, rechts )
    A[l], A[m], A[r] = A[links], A[middle], A[rechts]
    return m

def median_quicksort( A, links, rechts ):
    if rechts <= links :
        return

    # find Pivot and partition array in-place
    pivot = median_pivot( A, links, rechts )
    pivot = partition( A, links+1, pivot, rechts-1 )

    # sort smaller array slices
    median_quicksort( A, links, pivot-1 )
    median_quicksort( A, pivot+1, rechts )

```

G. Zachmann Informatik 2 - SS 06 Sortieren 56

Weitere Optimierungen von Quicksort

- Beobachtung:
 - Arrays auf den unteren Levels der Rekursion sind "klein" und "fast" sortiert
 - Idee: verwende dafür Algo, der auf "fast" sortierten Arrays schneller ist → Insertionsort
- Was tun gegen quadratische Laufzeit?
 - Zähle Rekursionstiefe mit
 - Schalte auf anderen Algo um, falls Tiefe größer $c \cdot \log(n)$ wird
 - Typ.: wähle $c=2$, schalte um auf Heapsort (später)

G. Zachmann Informatik 2 - SS 06 Sortieren 57

State-of-the-Art für Quicksort

- Untere Schranke:
 - $C_{\max}(n) > C_{\text{av}}(n) \geq \lceil \log(n!) \rceil - 1 \approx n \log n - 1,4427n$
- Ziel: $C_{\max}(n)$ bzw. $C_{\text{av}}(n) \leq n \log n + cn$ für kleines c
- Quicksort-Verfahren
 - QUICKSORT (Hoare 1962)
 - $C_{\text{av}}(n) \approx 1,386n \log n - 2,846n + O(\log n)$
 - CLEVER-QUICKSORT (Hoare 1962)
 - $C_{\text{av}}(n) \approx 1,188n \log n - 2,255n + O(\log n)$
 - QUICK-HEAPSORT (Cantone & Cincotti 2000)
 - $C_{\text{av}}(n) = n \log n + 3n + o(n)$
 - QUICK-WEAK-HEAPSORT
 - $C_{\text{av}}(n) = n \log n + 0,2n + o(n)$

G. Zachmann Informatik 2 - SS 06 Sortieren 58

Quicksort = Beispiel für Divide-and-Conquer

- Anmerkung: Quicksort ist ein klassisches Divide-and-Conquer-Verfahren
 - Divide = partition
 - Conquer = rekursiver Aufruf (kleineres Arrays sortieren)
 - Combine (merge) = keine Arbeit hier nötig, da die Teilarrays *in-situ* sortiert wurden

G. Zachmann Informatik 2 - SS 06 Sortieren 59

Heap

- Definition **Heap** :
ist ein **vollständiger** Baum mit einer Ordnung \geq , für den gilt, daß jeder Vater \geq seinen beiden Söhnen ist, d.h.,

$$\forall v : \text{left}(v) \leq v \wedge \text{right}(v) \leq v$$

Form :



und

Ordnung: Entlang jedes Pfades von einem Knoten zur Wurzel sind die Knoteninhalte aufsteigend sortiert.

- Spezielle Eigenschaft der Wurzel: kleinstes Element
- Achtung: keine Ordnung zwischen $\text{left}(v)$, $\text{right}(v)$!
- Obige Def \rightarrow sog. "Max-Heap"; analog "Min-Heap"

G. Zachmann Informatik 2 - SS 06 Sortieren 60

Erinnerung

- Array betrachtet als vollständiger Baum
 - physikalisch – lineares Array
 - logisch – Binärbaum, gefüllt auf allen Stufen (außer der niedrigsten)
- Abbildung von Array-Elementen auf Knoten (und umgekehrt) :
 - Wurzel $\leftrightarrow A[1]$
 - links[j] $\leftrightarrow A[2j]$
 - rechts[j] $\leftrightarrow A[2j+1]$
 - Vater[j] $\leftrightarrow A[\lfloor j/2 \rfloor]$

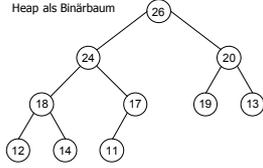
G. Zachmann Informatik 2 - SS 06 Sortieren 61

Beispiel

Heap als Array

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Heap als Binärbaum



- Höhe eines Heaps: $\lfloor \lg(n) \rfloor$
- letzte Zeile wird von links nach rechts aufgefüllt

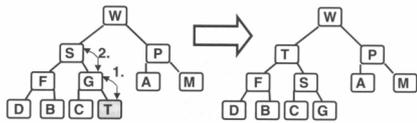
G. Zachmann Informatik 2 - SS 06 Sortieren 62

- Einfügen eines Knotens:
 - nur eine mögliche Position, wenn Baum vollständig bleiben soll
 - aber im allg. wird Heap-Eigenschaft verletzt
 - Wiederherstellen mit **UpHeap** (Algorithmus ähnlich zu Bubblesort):
vergleiche Sohn und Vater und vertausche gegebenenfalls



G. Zachmann Informatik 2 - SS 06 Sortieren 63

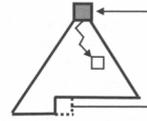
- Beispiel:



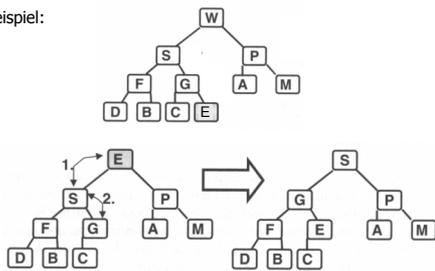
- Aufwand: $O(\log N)$

- Löschen der Wurzel:

- Ersetzen durch das am weitesten rechts stehende Element der untersten Schicht (Erhaltung der Formeigenschaft des Heaps)
- Zustand jetzt: beide Teilbäume unter der Wurzel sind Heaps, aber gesamter Baum i.A. nicht mehr
- Wiederherstellen der Ordnungseigenschaft mit **DownHeap**:
Vertauschen des Vaters mit dem größeren der beiden Söhne, bis endgültiger Platz gefunden wurde



- Beispiel:



- Aufwand: **UpHeap** und **DownHeap** sind beide $O(\log N)$

- Heap implementiert eine Verallgemeinerung des FIFO-Prinzips:
Priority-Queue (p-Queue)

- Daten werden entsprechend ihres Wertes, der Priorität, einsortiert
- Daten werden nur vorne an der Wurzel (höchste Priorität) entfernt