Informatik II
Sortieralgorithmen

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Motivation

- Preprocessing fürs Suchen
- Sind für kommerzielle Anwendungen häufig die Programmteile, die die meiste Rechenzeit verbrauchen
- Viele raffinierte Methoden wurden im Laufe der Zeit entwickelt, von denen wir ein paar kennenlernen wollen

Die Sortieraufgabe

- Eingabe: Datensätze (records) aus einem File, der Form
  - Key und *satellite/payload* data

Sortierschlüssel	Inhalt

- Sortierschlüssel kann aus einem oder mehreren Feldern des Datensatzes bestehen (z.B. Nachname + Vorname)
- Bedingung: auf den Keys muß eine totale Ordnungsrelation $\preceq$ definiert sein, d.h., es gilt
  - Trichotomie: für alle Keys a,b gilt genau eine Relation

$$a \prec b, \quad a = b, \quad a \succ b$$

  - Transitivität:

$$\forall a, b : a \prec b \ \wedge b \prec c \ \Rightarrow \ a \prec c$$

- Aufgabe: bestimme eine Permutation $\Pi = (p_1, \ldots, p_n)$ für die Records, so daß die Keys in nicht-fallender Ordnung sind:

$$K_{p_1} \preceq \ldots \preceq K_{p_n}$$

- Implementierung üblicherweise als Klasse mit eingebautem Vergleichsoperator:

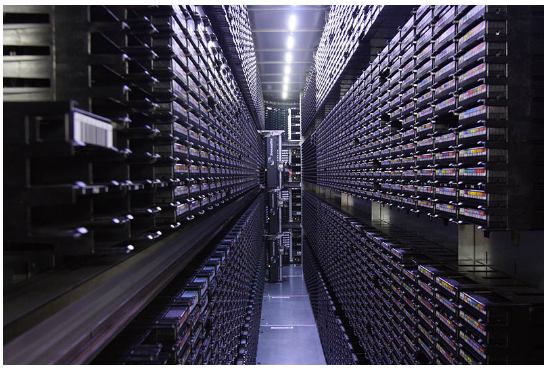
```
class MyData:
    def __init__( self, key, value ):
        self.key = key
        self.value = value
    def __cmp__( self, other ):
        if self.key < other.key:
            return -1
        elif self.key > other.key:
            return 1
        else:
            return 0
a = MyData(…)
b = MyData(…)
if a < b:
   ...
```

Klassifikation / Kriterien von Sortierverfahren

- Interne Sortierverfahren:
  - Alle Datensätze befinden sich im Hauptspeicher
  - Es besteht random access auf den gesamten Datenbestand
  - Bekannte Verfahren: Bubblesort, Insertionsort, Selectionsort, Quicksort, Heapsort
- Externe Sortierverfahren:
  - Die Datensätze befinden sich in einem Hintergrundspeicher (Festplatte, Magnetband, etc.) und können nur sequentiell verarbeitet werden
  - Bekanntes Verfahren: Mergesort

Exkurs: Tape Libraries

- Wird auch heute noch für Datenarchive gerne verwendet
- Beispiel (Deutsches Klimarechenzentrum, Stand 2010):
  - 8 robots per library
  - 500 TeraByte disk cache
  - Total capacity: 60 PetaByte
  - Projected fill rate: 10 PetaByte/year



Fortsetzung Klassifikation

- Vergleichsbasiert (comparison sort): zulässige Operationen auf den Daten sind nur Vergleich und Umkopieren
- Zahlenbasiert: man darf/kann auf den Keys auch rechnen
  - Diese Unterscheidung ist analog zu der bei den Suchalgorithmen
- Stabil (stable): Gleiche Keys haben nach dem Sortieren die selbe relative Lage zueinander wie vorher
- Array-basiert vs. Listen-basiert: Können Datensätze beliebig im Speicher angeordnet sein (Liste), oder müssen sie hintereinander im Speicher liegen (Array)
- In-Place (in situ): Algorithmus braucht nur konstanten zusätzlichen Speicher (z.B. Zähler; aber keine Hilfsarrays o.ä.)
  - Konsequenz: Ausgabe-Array = Eingabe-Array

Erster Sortier-Algorithmus: Bubblesort

- Die Idee des Algo:
  - Vergleiche von links nach rechts jeweils zwei Nachbarelemente und vertausche deren Inhalt, falls sie in der falschen Reihenfolge stehen;
  - Wiederhole dies, bis alle Elemente richtig sortiert sind;
  - Analogie: die kleinsten Elemente steigen wie Luftblasen zu ihrer richtigen Position auf (je nachdem, wierum man sortiert)

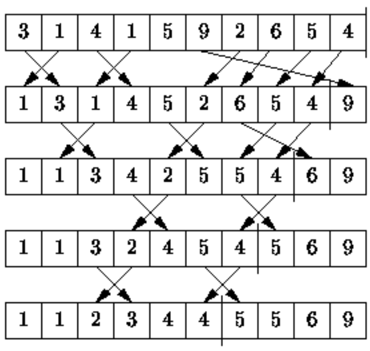
Effiziente Python-Implementierung

```
def bubblesort( a ):
    for k in ...:
        for i in range( 0, len(a)-1 ):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

```
def bubblesort( a ):
    for k in range( 0, len(a)-1 ):
        for i in range( 0, len(a)-1 ):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

```
def bubblesort( a ):
    for k in range( len(a)-1, 0, -1 ):
        for i in range(0,k):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```


- Beispiel:



Korrektheits-Beweis

- Schleifeninvariante:
  - Nach dem 1. Durchlauf befindet sich das größte Element an der richtigen Stelle
  - Nach dem 2. Durchlauf auch das 2.-größte, etc.
  - Nach dem *i*-ten Durchlauf befinden sich die *i* größten Elemente an der richtigen Position (und damit in der richtigen Reihenfolge)
- Nach spätestens *N*-1 Durchgängen ist das Array sortiert
- Da bei jedem Durchlauf auch andere Elemente ihre Position verbessern, ist häufig der Vorgang bereits nach weniger als *N*-1 Durchgängen beendet

- Beispiel

0														14	
S	O	R	T	I	E	R	B	E	I	S	P	I	E	L	Original-Array
O	R	S	I	E	R	B	E	I	S	P	I	E	L	T	nach 1. BubbleUp
O	R	I	E	R	B	E	I	S	P	I	E	L	S	T	nach 2. BubbleUp
O	I	E	R	B	E	I	R	P	I	E	L	S	S	T	
I	E	O	B	E	I	R	P	I	E	L	R	S	S	T	
E	I	B	E	I	O	P	I	E	L	R	R	S	S	T	... etc. ...
E	B	E	I	I	O	I	E	L	P	R	R	S	S	T	
B	E	E	I	I	I	E	L	O	P	R	R	S	S	T	
B	E	E	I	I	E	I	L	O	P	R	R	S	S	T	
B	E	E	I	E	I	I	L	O	P	R	R	S	S	T	nach 10. BubbleUp
B	E	E	E	I	I	I	L	O	P	R	R	S	S	T	Sortiert !

- Kleine Optimierung: Test auf vorzeitiges Ende

```
def bubblesort( a ):
    for k in range (len(a)-1, 0, -1):
        sorted = true
        for i in range (0,k):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
                sorted = false
        if sorted:
            break
```


Aufwand von Bubblesort

- Laufzeitberechnung für den *worst case*:

```
def bubblesort( a ):
    k = len(a)-1
    while k >= 0:
        for i in range (0,k):
            if a[i]>a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

O(1) O(k) $T(n)$

$$T(n) \in \sum_{k=1}^{n} O(k) = O\Big(\sum_{k=1}^{n} k\Big) = O\Big(\frac{1}{2}n(n+1)\Big) = O(n^2)$$

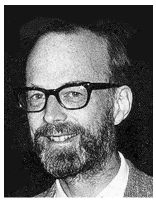
- Für den *best case* (für den Code mit "early exit"): $T(n) \in O(n)$
  - Beweis: Übungsaufgabe
- Im *average case* (o.Bew.): $T(n) \in O(n^2)$

Weitere "einfache" Sortierverfahren

- Insertion Sort, Selection Sort, u.a.

http://www.youtube.com/watch?v=INHF_5RIxTE

Quicksort

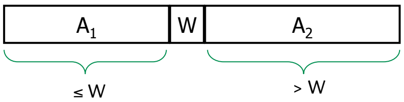


C. A. R. Hoare, 1960

- C.A.R. Hoare, britischer Informatiker, erfand 1960 Quicksort
- Bis dahin dachte man, man müsse die einfachen Sortieralgorithmen durch raffinierte Assembler-Programmierung beschleunigen
- Quicksort zeigt, daß es sinnvoller ist, nach besseren Algorithmen zu suchen
- Einer der schnellsten bekannten allgemeinen Sortierverfahren
- Idee:
  - Vorgegebenes Sortierproblem in kleinere Teilprobleme zerlegen
  - Teilprobleme rekursiv sortieren
  - Allgemeines Algorithmen-Prinzip: divide and conquer (divide et impera)

Der Algorithmus

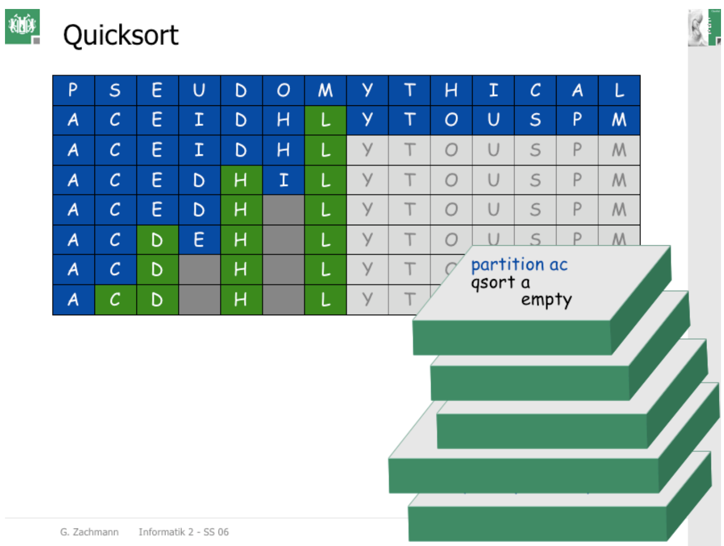
1. Wähle irgend einen Wert W des Arrays A
2. Konstruiere Partitionierung des Arrays mit folgenden Eigenschaften:



- A_1 und A_2 sind noch unsortiert!

3. Wenn man jetzt A_1 und A_2 sortiert, ist das Problem gelöst
4. A_1 und A_2 sortiert man natürlich wieder mit ... Quicksort

Algo-Animation



- Konstruktion der Partition ist die eigentliche Kunst / Arbeit bei Quicksort!

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

- Resultat:

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

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
partitioned
left
right

Courtesy Kevin Wayne & Robert Sedgewisk

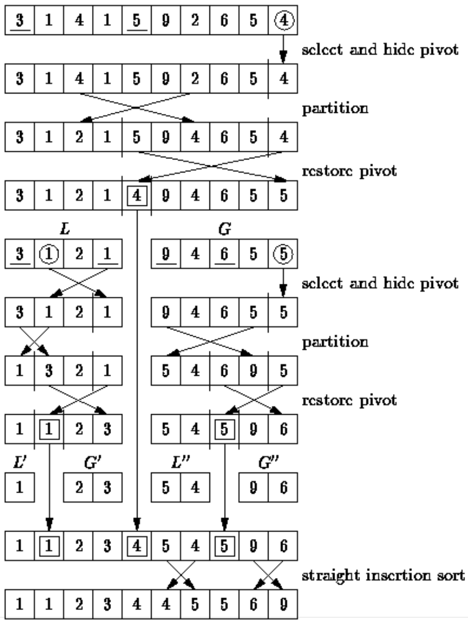
Python-Implementierung

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

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

    # 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 )
```

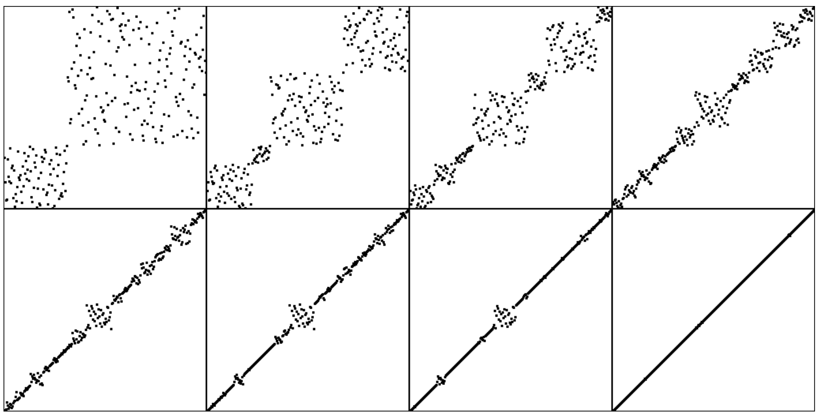


```
def partition( A, links, rechts ):
    pivot = rechts            # choose right-most as pivot
    i, j = links, rechts-1

    while i < j:              # quit when i,j "cross over"
        # 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
```

Beispiel-Durchlauf von Quicksort



Visualisierung von Quicksort



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

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:
  1. $A_{l..i-1}$ → Einträge dieses Abschnitts sind ≤ pivot
  2. $A_{j+1..r-1}$ → Einträge dieses Abschnitts sind > pivot
  3. A_r = pivot
  4. $A_{i..j}$ → Status bzgl. *pivot* ist unbekannt
- Ist eine Schleifeninvariante

- Initialisierung: vor der ersten Iteration gilt:
  - $A_{l..i-1}$ und $A_{j+1..r-1}$ 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]
[…]
```

- Erhaltung der Invariante (am Ende des Schleifenrumpfes):
  - Nach erster **while**-Schleife gilt: **A[i]** > **p** oder **i=r**
  - Nach zweiter **while**-Schleife gilt: **A[j]** ≤ **p** oder **j=l**
  - Vor **if** gilt: falls **i<j**, dann ist **A[i]** > **p** ≥ **A[j]**
    - was dann durch den **if**-Body "repariert" wird
  - Nach **if** gilt wieder Schleifeninvariante

```
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]
[…]
```


- Beendigung:
  - nach while-Schleife gilt:

$$i \geq j \ \wedge \ (A_i > A_r \vee i = r)$$

  - d.h.
    - $A_{l..i-1} \leq pivot$
    - $A_{i+1..r-1} > pivot$
    - $A_r = pivot$
    - der vierte Bereich , $A_{i..j}$, ist leer

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

- Die letzte Zeile vertauscht A_i und A_r:
  - *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

Laufzeit des Algorithmus

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

- 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

$$\begin{array}{c} \downarrow \\ n-1 \\ \downarrow \\ n-2 \\ \downarrow \\ n-3 \\ \vdots \\ 2 \\ \downarrow \\ 1 \end{array}$$

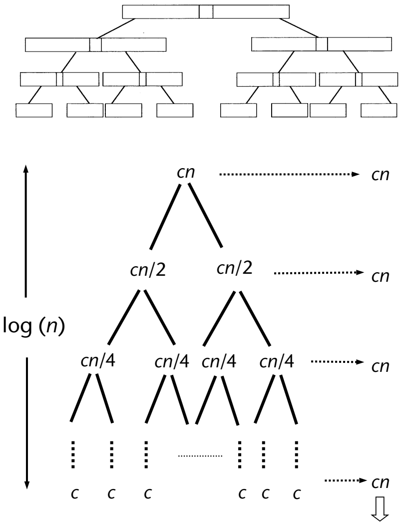
Laufzeit bei Best-Case-Partitionierung

- 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:

$$T(n) \leq 2T(\tfrac{n}{2}) + \text{PartitionTime}(n) = 2T(\tfrac{n}{2}) + cn$$

- Ann.: $T(1) = c$
- Also: $T(n) \in \Theta(n \log(n))$

Rekursionsbaum für Best-Case-Partition



Gesamt: $c \cdot n \log n$

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 ist: Inspektion von drei Elementen
  - A[li], A[re], A[mid] mit mid=(li+re)/2
  - wähle davon den Median (wertmäßig das mittlere der drei)
  - nennt man dann *"median-of-three quicksort"*
- Alternative: zufälligen Index als Pivot-Element
  - Diese Technik heißt: "Randomisierung"

- Beispiel, wenn man nur A[mid] als Vergleichelement 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[li], A[re], A[mid] als Pivot-Element verwendet wird:

```
SORTIERBEISPIEL
BEIIIEE L RTSPROS
```


Programm für Median-of-3-Quicksort

```
# Liefert Indizes a,b,c (= Permutation von i,j,k)
# so dass A[a] <= A[b] <= A[c]
def median( A, i, j, k ):
    if A[i] <= A[j]:
        if A[j] <= A[k]:
            return i,j,k
        else:                             # i,k < j
            if A[i] <= A[k]:
                return i,k,j
            else:
                return k,i,j
    else:                                 # j < i
        if A[i] <= A[k]:
            return j,i,k
        else:                             # j,k < i
            if A[j] <= A[k]:
                return j,k,i
            else:
                return k,j,i
```

```
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 )
```


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 *log(n) wird
  - Typischerweise: wähle c=2, schalte um auf Heapsort (später)

State-of-the-Art für Quicksort

- Untere Schranke für average case:

$$C_{av}(n) \geq \lceil \log(n!) \rceil - 1 \approx n \log n - 1,4427n$$

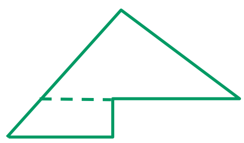
- Ziel: $C_{av}(n) \leq n \log n + cn$ für möglichst kleines c
- Quicksort-Verfahren:
  - QUICKSORT (Hoare 1962)
    $C_{av}(n) \approx 1,386 n \log n - 2,846 n + O(\log n)$
  - CLEVER-QUICKSORT (Hoare 1962)
    $C_{av}(n) \approx 1,188 n \log n - 2,255 n + O(\log n)$
  - QUICK-HEAPSORT (Cantone & Cincotti 2000)
    $C_{av}(n) = n \log n + 3n + o(n)$
  - QUICK-WEAK-HEAPSORT
    $C_{av}(n) = n \log n + 0,2n + o(n)$

Der Heap

- Definition Heap :
  ist ein vollständiger Baum mit einer Ordnung ≤, für den gilt, daß jeder Vater ≤ seiner beiden Söhnen ist, d.h.,

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

- Form:



- Eigenschaft: entlang jedes Pfades von der Wurzel zu einem Knoten sind die Knoten aufsteigend sortiert.
- Spezielle Eigenschaft der Wurzel: kleinstes Element
- Achtung: keine Ordnung zwischen `left(v)` und `right(v)`!
- Obige Definition ist ein sog. "Min-Heap" (analog "Max-Heap")

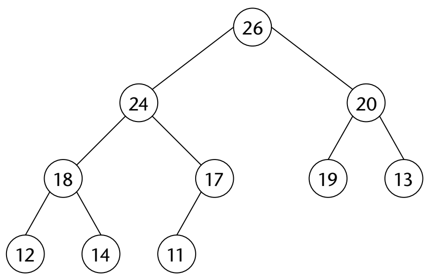
Erinnerung

- 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[i] $\leftrightarrow A[\,2i\,]$
  - rechts[i] $\leftrightarrow A[\,2i+1\,]$
  - Vater[i] $\leftrightarrow A[\,\lfloor i/2 \rfloor\,]$

Beispiel

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

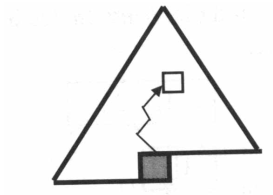
Max-Heap als Array



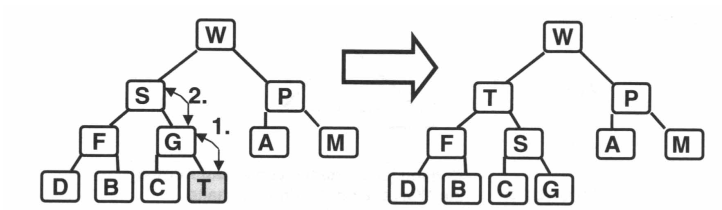
Max-Heap als Binärbaum

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

- Einfügen eines Knotens:
  - Nur eine mögliche Position, wenn der 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

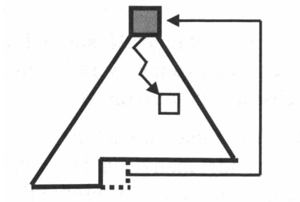


- Beispiel:

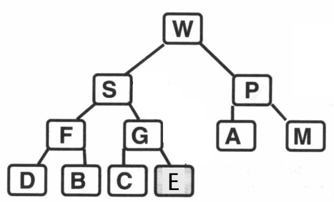


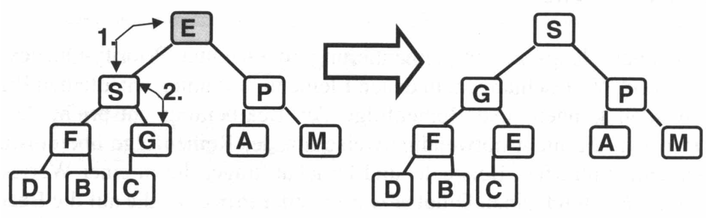
- Aufwand: O(log *N*)

- 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 immer noch Heaps, aber der gesamte Baum i.A. nicht mehr
  - Wiederherstellen der Ordnungseigenschaft mit **DownHeap**: Vertauschen des Vaters mit dem kleineren 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: die Priority-Queue (*p-queue*)
  - Daten werden nur vorne an der Wurzel (höchste Priorität) entfernt (wie bei Queue)
  - Aber Daten werden entsprechend ihres Wertes, der Priorität, einsortiert