

Algorithmen und Datenstrukturen

Wintersemester 2018/19
1. Vorlesung

Kapitel 1: Sortieren

Das Problem

Gegeben: eine Folge $A = \langle a_1, a_2, \dots, a_n \rangle$ von n Zahlen

Umordnung

Algorithmus

Gesucht:

eine Permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ von A

so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Ausgabe

Beachte:

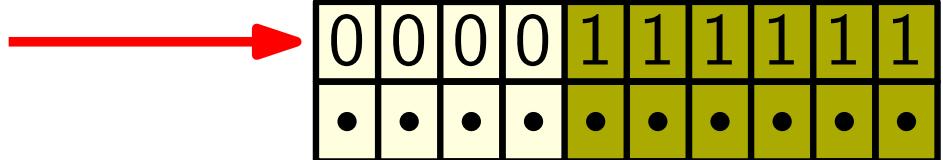
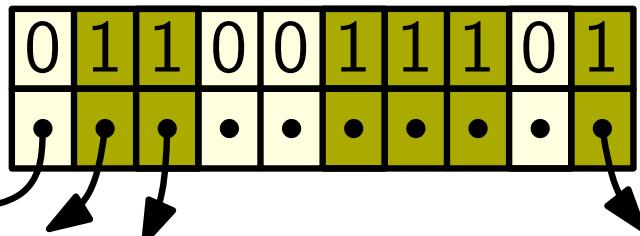
Computerinterne Zahlendarstellung hier unwichtig!

Wichtig:

- Je zwei Zahlen lassen sich vergleichen.
- Ein Vergleich dauert „konstante Zeit“, d.h. die Dauer ist unabhängig von n .



Noch was:

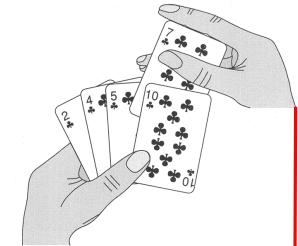


Frage an alle Erstis

Wie sortieren Sie?

Eine Lösung

InsertionSort



- Linke Hand anfangs leer. Alle Karten liegen auf dem Tisch.
- Rechte Hand nimmt Karten nacheinander auf und steckt sie (von rechts kommend) an die richtige Position zwischen die Karten in der linken Hand.
- Linke Hand hält immer eine sortierte Reihenfolge aufrecht.

Invariante!



Korrektheit: am Ende sind alle Karten in der linken Hand – und zwar *sortiert!*

Ein inkrementeller Algorithmus

// In Pseudocode

```
IncrementalAlg( array of ... A )
```

↓

Name des Alg. Eingabe Variable

Typ der Eingabe (hier ein Feld von ...)

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

berechne Lösung für $A[1]$ // Initialisierung

for $j = 2$ **to** $A.length$ **do** // Schleifenkopf

Zuweisungsopeator

- Anzahl der Elemente des Feldes A
- in manchen Sprachen $j := 2$
 - in manchen Büchern $j \leftarrow 2$
 - in Java $j = 2$

Ein inkrementeller Algorithmus

// In Pseudocode

IncrementalAlg(array of ... A)

berechne Lösung für $A[1]$ // Initialisierung

for $j = 2$ **to** $A.length$ **do** // Schleifenkopf

 // Schleifenkörper; wird $(A.length - 1)$ -mal durchlaufen

 berechne Lösung für $A[1..j]$ mithilfe der für $A[1..j - 1]$

return Lösung // Ergebnisrückgabe

Teilarray von A mit den Elementen $A[1], A[2], \dots, A[j]$

Ein inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A  
berechne Lösung für A[1] // nix zu tun: A[1..1] ist sortiert  
for j = 2 to A.length do  
    // berechne Lösung für A[1..j] mithilfe der für A[1..j - 1]  
    ... kommt noch ...  
return Lösung // nicht nötig – das aufrufende Programm  
hat Zugriff auf das sortierte Feld A
```

Ein inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
```

~~berechne Lösung für $A[1]$~~ // nix zu tun: $A[1..1]$ ist sortiert

for $j = 2$ **to** $A.length$ **do**

// berechne Lösung für $A[1..j]$ mithilfe der für $A[1..j - 1]$

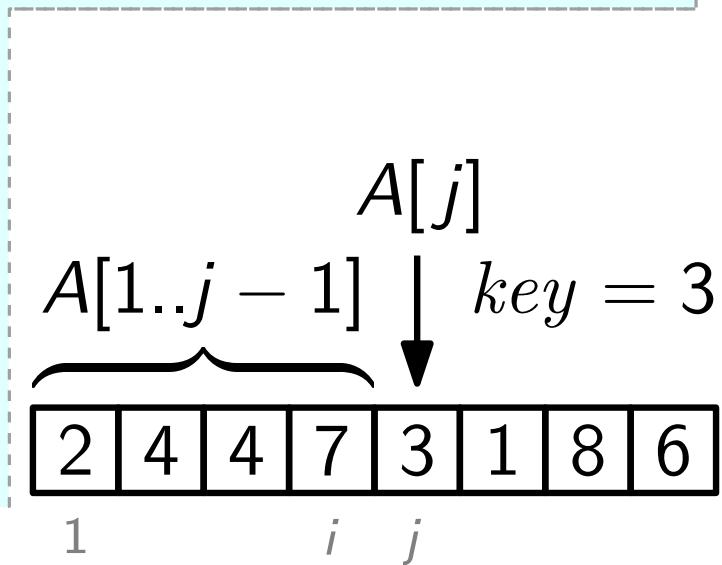
// hier: füge $A[j]$ in die sortierte Folge $A[1..j - 1]$ ein

$key = A[j]$

$i = j - 1$

while $i > 0$ **and** $A[i] > key$ **do**

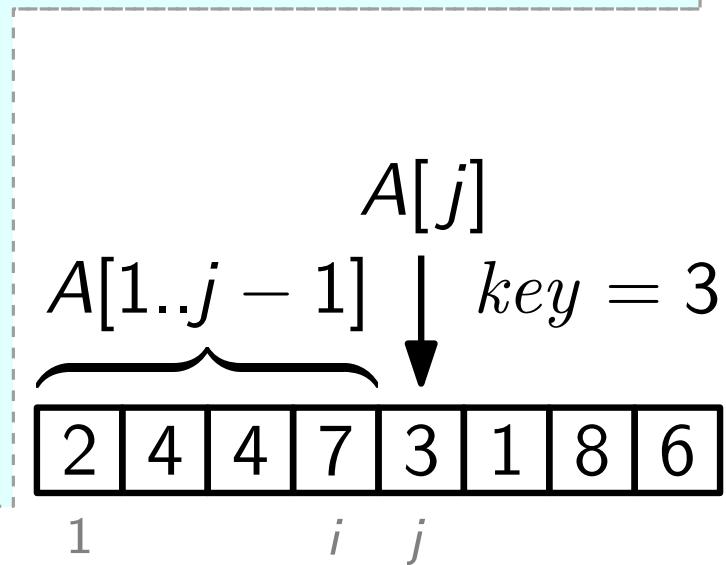
Wie verschieben wir die Einträge größer key nach rechts?



Ein inkrementeller Algorithmus

InsertionSort

```
IncrementalAlg( array of int A ) // Schreiben wir künftig so: int[] A
berechne Lösung für A[1] // nix zu tun: A[1..1] ist sortiert
for j = 2 to A.length do
    // berechne Lösung für A[1..j] mithilfe der für A[1..j - 1]
    // hier: füge A[j] in die sortierte Folge A[1..j - 1] ein
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```



Fertig?

Nicht ganz...

Wir interessieren uns heute (und im Rest dieser Vorlesung) für folgende zentrale Fragen:

- Ist der Algorithmus korrekt?
- Welche Laufzeit hat der Algorithmus?
- Wie viel Speicherplatz benötigt der Algorithmus?

Korrektheit beweisen

```
InsertionSort(int[] A)
```

```
for j = 2 to A.length do
```

```
    key = A[j]
```

```
    i = j - 1
```

```
    while i > 0 and A[i] > key do
```

```
        A[i + 1] = A[i]
```

```
        i = i - 1
```

```
    A[i + 1] = key
```

Hier enthält $A[1..j - 1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Idee der *Schleifeninvariante*:

Wo? am Beginn jeder Iteration der for-Schleife...

Was? **WANTED:** Bedingung, die

- an dieser Stelle immer erfüllt ist und
- bei Abbruch der Schleife Korrektheit liefert

Korrektheit beweisen

```
InsertionSort(int[] A)
```

```
  for  $j = 2$  to  $A.length$  do
```

```
    key =  $A[j]$ 
```

```
    i =  $j - 1$ 
```

```
    while  $i > 0$  and  $A[i] > key$  do
```

```
         $A[i + 1] = A[i]$ 
```

```
        i =  $i - 1$ 
```

```
     $A[i + 1] = key$ 
```

Hier enthält $A[1..j - 1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Schleifeninvariante

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:

$A[1..j - 1] = A[1..1]$ ist unverändert und „sortiert“.

Korrektheit beweisen

```
InsertionSort(int[] A)
```

```
for j = 2 to A.length do
```

```
    key = A[j]
```

```
    i = j - 1
```

```
    while i > 0 and A[i] > key do
```

```
        A[i + 1] = A[i]
```

```
        i = i - 1
```

```
    A[i + 1] = key
```

Hier enthält $A[1..j - 1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Schleifeninvariante

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ 2.) Aufrechterhaltung

Zeige: Wenn die Invariante vor dem j . Schleifendurchlauf erfüllt ist, dann auch vor dem $j + 1$.

Hier: Eigentlich: Invariante für while-Schleife aufstellen und beweisen!

Korrektheit beweisen

```
InsertionSort(int[] A)
```

```
for j = 2 to A.length do
```

```
    key = A[j]
```

```
    i = j - 1
```

```
    while i > 0 and A[i] > key do
```

```
        A[i + 1] = A[i]
```

```
        i = i - 1
```

```
    A[i + 1] = key
```

Hier enthält $A[1..j - 1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Schleifeninvariante

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ 2.) Aufrechterhaltung ✓

Zeige: Wenn die Invariante vor dem j . Schleifendurchlauf erfüllt ist, dann auch vor dem $j + 1$.

Hier: Beob.: Elemente werden so lange nach rechts geschoben wie nötig. key wird korrekt eingefügt.

Korrektheit beweisen

```
InsertionSort(int[] A)
```

```
for j = 2 to A.length do
```

```
    key = A[j]
```

```
    i = j - 1
```

```
    while i > 0 and A[i] > key do
```

```
        A[i + 1] = A[i]
```

```
        i = i - 1
```

```
    A[i + 1] = key
```

Hier enthält $A[1..j - 1]$ dieselben Elemente wie zu Beginn des Algorithmus – jedoch sortiert.

Schleifeninvariante

Beweis nach Schema „F“: Wir brauchen noch drei Zutaten...

1.) Initialisierung ✓ 2.) Aufrechterhaltung ✓ 3.) Terminierung ✓

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung ist $j > A.length$.
D.h. $j = A.length + 1$. Einsetzen in Inv. \Rightarrow korrekt!

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

if $k < 0$ **then error**(...)

$f = 1$

$j = 2$

while $j \leq k$ **do**

$f = f \cdot j$

$j = j + 1$

return f

Schleifeninvariante:
 $f = (j - 1)!$

1.) Initialisierung ✓

Zeige: Invariante ist beim 1. Schleifendurchlauf erfüllt.

Hier: klar, denn für $j = 2$ gilt:

$$f = (2 - 1)! = 1! = 1$$

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

```
if  $k < 0$  then error(...)
```

 $f = 1$
 $j = 2$

```
while  $j \leq k$  do
```

 $f = f \cdot j$
 $j = j + 1$

```
return  $f$ 
```

Schleifeninvariante:
 $f = (j - 1)!$



1.) Initialisierung ✓ 2.) Aufrechterhaltung ✓

Zeige: Wenn die Invariante vor dem j . Schleifendurchlauf erfüllt ist, dann auch vor dem $j + 1$.

Hier: Vor dem j . Durchlauf gilt INV, d.h. $f = (j - 1)!$
 Dann wird f mit j multipliziert $\Rightarrow f = j!$
 Dann wird j um 1 erhöht $\Rightarrow f = (j - 1)! \Rightarrow \text{INV}$

Noch ein Beispiel: Fakultät berechnen

Factorial(int k)

```
if  $k < 0$  then error(...)
```

 $f = 1$
 $j = 2$
while $j \leq k$ **do**
 $f = f \cdot j$
 $j = j + 1$
return f

Schleifeninvariante:
 $f = (j - 1)!$



- 1.) Initialisierung ✓
- 2.) Aufrechterhaltung ✓
- 3.) Terminierung ✓

Zeige: Zusammengenommen ergeben Invariante und verletzte Schleifenbedingung die Korrektheit.

Hier: Verletzte Schleifenbedingung: $j > k$, also $j = k + 1$. Einsetzen von „ $j = k + 1$ “ in INV liefert $f = k!$

Noch ein Beispiel: Fakultät berechnen

```
Factorial(int k)
```

```
  if  $k < 0$  then error(...)
```

```
   $f = 1$ 
```

```
   $j = 2$ 
```

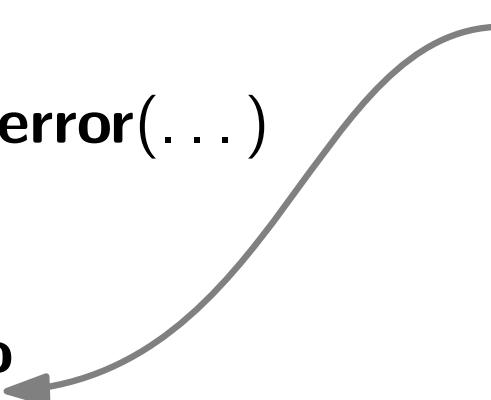
```
  while  $j \leq k$  do
```

```
     $f = f \cdot j$ 
```

```
     $j = j + 1$ 
```

```
  return  $f$ 
```

Schleifeninvariante:
 $f = (j - 1)!$

- 
- 1.) Initialisierung ✓
 - 2.) Aufrechterhaltung ✓
 - 3.) Terminierung ✓

Der Algorithmus Factorial(int) terminiert und liefert das korrekte Ergebnis.

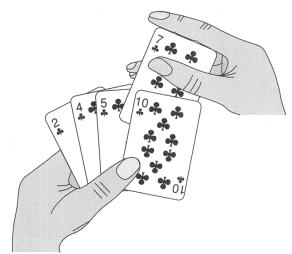
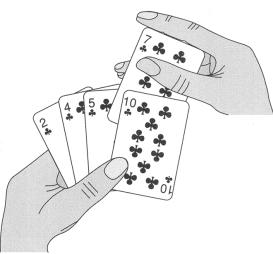
Selbstkontrolle

- Programmieren Sie *InsertionSort* in Java!
 - Lesen Sie Kapitel 1 und Anhang A des Buchs von Cormen et al. durch und machen Sie dazu so viel Übungsaufgaben wie möglich!
 - Bringt Fragen in die Übung mit!
 - Bleiben Sie von Anfang an am Ball!
 - Schreiben Sie sich in die Vorlesung ein:
 - wuecampus2.uni-wuerzburg.de
 - <https://www-sbhome1.zv.uni-wuerzburg.de>
- Zählen Sie Vergleiche für verschiedene Eingaben.
- 

Algorithmen und Datenstrukturen

Wintersemester 2018/19
2. Vorlesung

Sortieren mit anderen Mitteln



Teile und herrsche

Idee:

- teile den Kartenstapel in zwei ungefähr gleichgroße Teile,
- sortiere die Teile (z.B. durch verschiedene Personen) und
- füge die Teilstapel zu einem sortierten Stapel zusammen.

Allgemein:

Teile...

eine Instanz in kleinere Instanzen *dieselben* Problems. ↗ Aufruf einer Funktion durch sich selbst

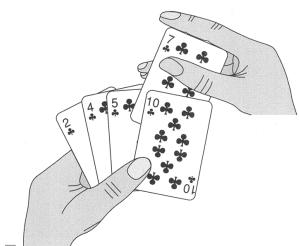
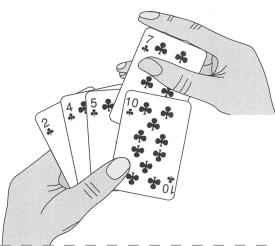
Herrsche...

durch **rekursives** Lösen von Teilinstanzen – nur falls diese sehr klein sind, löse sie direkt.

Kombiniere... die Teillösungen zu einer Lösung der ursprünglichen Instanz.

*) Abb. aus [Cormen et al. „Introduction to Algorithms“, MIT Press]

Teile und herrsche



MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

Defaultwerte –

Dadurch wird die Funktion

$\text{MergeSort}(A) \equiv$

$\text{MergeSort}(A, 1, A.length)$

definiert.

Allgemein:

Teile...

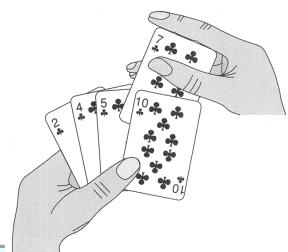
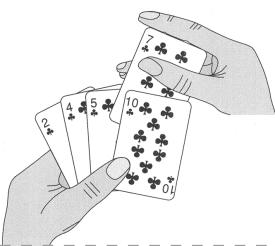
eine Instanz in kleinere Instanzen *dieselben* Problems.

Herrsche...

durch *rekursives Lösen* von Teilinstanzen – nur falls diese sehr klein sind, löse sie direkt.

Kombiniere... die Teillösungen zu einer Lösung der ursprünglichen Instanz.

*) Abb. aus [Cormen et al. „Introduction to Algorithms“, MIT Press]



Teile und herrsche

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$

} teile

MergeSort(A, ℓ, m)

} herrsche

MergeSort($A, m + 1, r$)

} kombiniere

Merge(A, ℓ, m, r)

To do!

Allgemein:

Teile...

eine Instanz in kleinere Instanzen *dieselben* Problems.

Herrse...

durch *rekursives Lösen* von Teilinstanzen – nur falls diese sehr klein sind, lös sie direkt.

Kombiniere... die Teillösungen zu einer Lösung der ursprünglichen Instanz.

*) Abb. aus [Cormen et al. „Introduction to Algorithms“, MIT Press]

Kombiniere

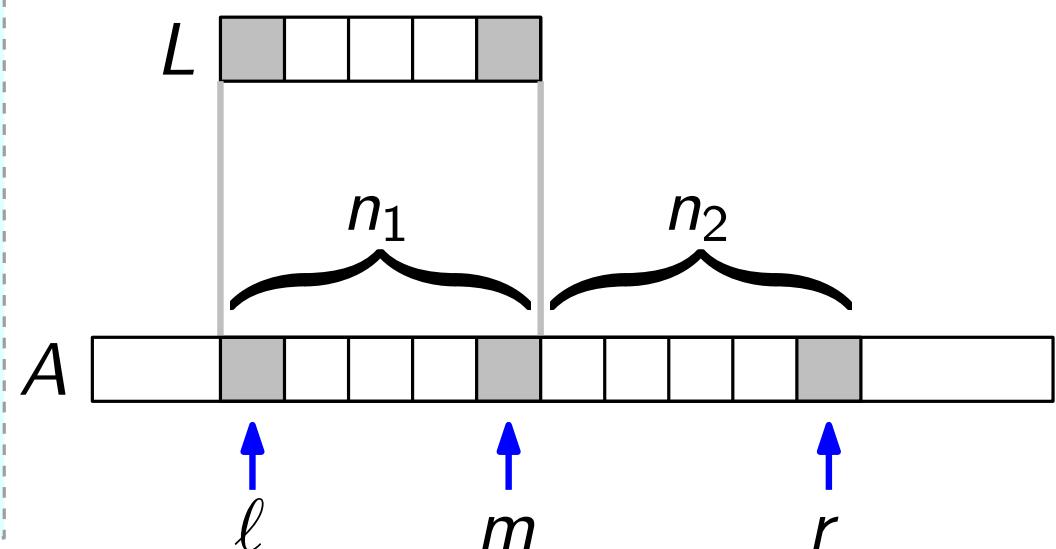
Merge(int[] A, int ℓ , int m , int r)

$$n_1 = m - \ell + 1; \quad n_2 = r - m$$

$L = \text{new int}[1..n_1 + 1]; \quad R = \text{new int}[1..n_2 + 1]$

$L[1..n_1] = A[\ell..m]$

for $i = 1$ to n_1 **do**
 └ $L[i] = A[(\ell - 1) + i]$



Kombiniere

Merge(int[] A, int ℓ , int m , int r)

$$n_1 = m - \ell + 1; \quad n_2 = r - m$$

$L = \text{new int}[1..n_1 + 1]; \quad R = \text{new int}[1..n_2 + 1]$

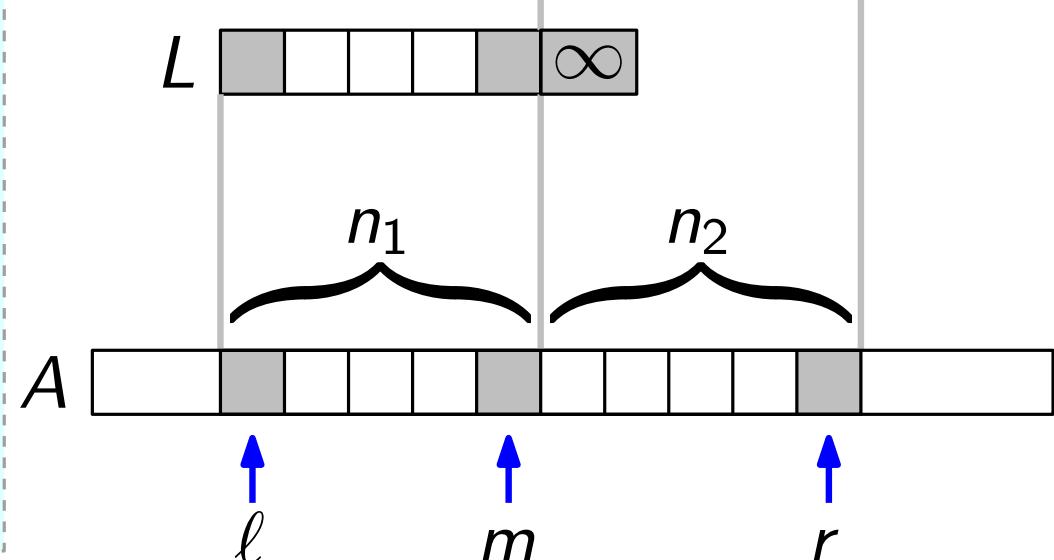
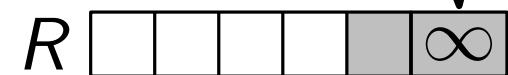
$L[1..n_1] = A[\ell..m]$

$R[1..n_2] = A[m + 1..r]$

$L[n_1 + 1] = R[n_2 + 1] = \infty$



Stopper (engl. sentinel)



Kombiniere

Merge(int[] A, int ℓ , int m , int r)

$$n_1 = m - \ell + 1; \quad n_2 = r - m$$

$L = \text{new int}[1..n_1 + 1]; \quad R = \text{new int}[1..n_2 + 1]$

$L[1..n_1] = A[\ell..m]$

$R[1..n_2] = A[m + 1..r]$

$L[n_1 + 1] = R[n_2 + 1] = \infty$

$i = j = 1$

for $k = \ell$ **to** r **do**

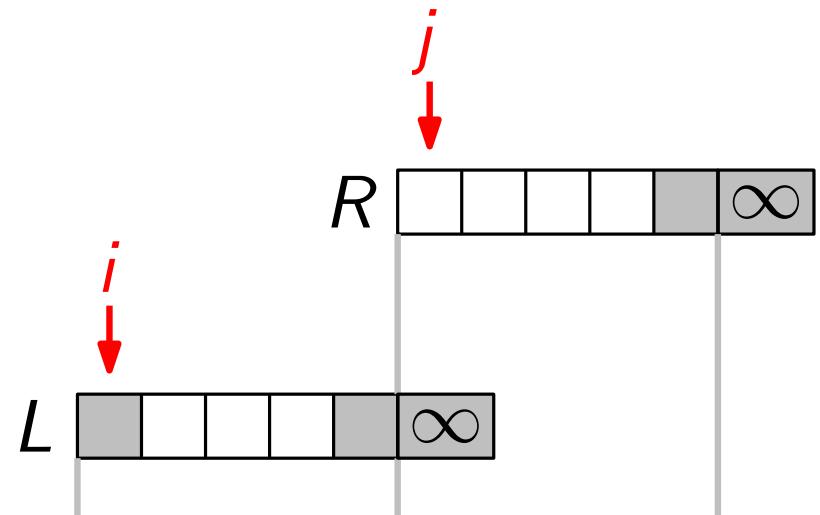
Aufgabe:

Schließen Sie Ihre Bücher und Ihren Browser!

Schreiben Sie mit Ihrer NachbarIn den Rest der Routine!

Benutzen Sie dazu die beiden neuen Felder L und R .

Sie haben **5 Minuten**.



Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k - 1]$ enthält die $k - \ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

Merge(int[] A, int ℓ , int m , int r)
 $n_1 = m - \ell + 1; n_2 = r - m$
lege $L[1..n_1 + 1]$ und $R[1..n_2 + 1]$ an
 $L[1..n_1] = A[\ell..m]$
 $R[1..n_2] = A[m + 1..r]$
 $L[n_1 + 1] = R[n_2 + 1] = \infty$
 $i = j = 1$
for $k = \ell$ **to** r **do**
 if $L[i] \leq R[j]$ **then**
 | $A[k] = L[i]$
 | $i = i + 1$
 else
 | $A[k] = R[j]$
 | $j = j + 1$

1. Initialisierung ✓

- Da beim ersten Schleifendurchlauf $k = \ell$ gilt, enthält $A[\ell..k - 1] = \langle \rangle$ die 0 kleinsten Ele. von $L \cup R$.
- Da $i = j = 1$, sind $L[i]$ und $R[j]$ die kleinsten noch nicht kopierten Ele.

Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k-1]$ enthält die $k-\ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```
Merge(int[] A, int  $\ell$ , int  $m$ , int  $r$ )
 $n_1 = m - \ell + 1; n_2 = r - m$ 
lege  $L[1..n_1 + 1]$  und  $R[1..n_2 + 1]$  an
 $L[1..n_1] = A[\ell..m]$ 
 $R[1..n_2] = A[m + 1..r]$ 
 $L[n_1 + 1] = R[n_2 + 1] = \infty$ 
 $i = j = 1$ 
for  $k = \ell$  to  $r$  do
    if  $L[i] \leq R[j]$  then // Fall (a)
         $A[k] = L[i]$ 
         $i = i + 1$ 
    else // Fall (b)
         $A[k] = R[j]$ 
         $j = j + 1$ 
```

1. Initialisierung

- Zwei Fälle: (a) $L[i] \leq R[j]$,

2. Aufrechterhaltung

- Nun gilt: – $A[\ell..k]$ enthält die kleinsten $k-\ell+1$ Elém. sortiert

(dank INV)

(Fall (b) symmetrisch.)

- Nun gilt: – $A[\ell..k]$ enthält die kleinsten $k-\ell+1$ Elém. sortiert

Betrachte Fall (a).

(dank INV)

– $L[i+1]$ ist kleinstes noch nicht kopiertes Elém. in L .

erhöhe $i \Rightarrow$

$L[i]$ ist kleinstes noch nicht kopiertes Elém. in L .]

erhöhe $k \Rightarrow$

$A[\ell..k-1]$ enthält die kleinsten $k-\ell$ Elém. sortiert]

INV

Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k - 1]$ enthält die $k - \ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```
Merge(int[] A, int  $\ell$ , int  $m$ , int  $r$ )
 $n_1 = m - \ell + 1; n_2 = r - m$ 
lege  $L[1..n_1 + 1]$  und  $R[1..n_2 + 1]$  an
 $L[1..n_1] = A[\ell..m]$ 
 $R[1..n_2] = A[m + 1..r]$ 
 $L[n_1 + 1] = R[n_2 + 1] = \infty$ 
 $i = j = 1$ 
for  $k = \ell$  to  $r$  do
    if  $L[i] \leq R[j]$  then
         $A[k] = L[i]$ 
         $i = i + 1$ 
    else
         $A[k] = R[j]$ 
         $j = j + 1$ 
```

1. Initialisierung ✓

2. Aufrechterhaltung ✓

3. Terminierung ✓

- Nach Abbruch der for-Schleife gilt $k = r + 1$.

$\Rightarrow A[\ell..k - 1] = A[\ell..r]$ enthält die $r - \ell + 1$ kleinsten Elem. von $L \cup R$ sortiert.

- $|L \cup R| = n_1 + n_2 + 2 = r - \ell + 3$, d.h. $A[\ell..r]$ korrekt sort.

+2 Stopper

Korrektheit von Merge

... nach Schema „F“!

0. Schleifeninvariante

- $A[\ell..k - 1]$ enthält die $k - \ell$ kleinsten Elemente von $L \cup R$ sortiert.
- $L[i]$ und $R[j]$ sind die kleinsten Elemente in L bzw. R , die noch nicht in A kopiert wurden.

```
Merge(int[] A, int ℓ, int m, int r)
  n1 = m - ℓ + 1; n2 = r - m
  lege L[1..n1 + 1] und R[1..n2 + 1] an
  L[1..n1] = A[ℓ..m]
  R[1..n2] = A[m + 1..r]
  L[n1 + 1] = R[n2 + 1] = ∞
  i = j = 1
  for k = ℓ to r do
    if L[i] ≤ R[j] then
      | A[k] = L[i]
      | i = i + 1
    else
      | A[k] = R[j]
      | j = j + 1
```

1. Initialisierung ✓

Also ist Merge korrekt!

Laufzeit?

Und MergeSort?

2. Aufrechterhaltung ✓

Merge macht genau $r - \ell + 1$ Vergleiche.

Korrekt? Effizient?

3. Terminierung ✓

q.e.d.

MergeSort – ein Beispiel

`MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)`

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$

`MergeSort(A, ℓ , m)`

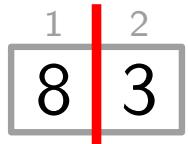
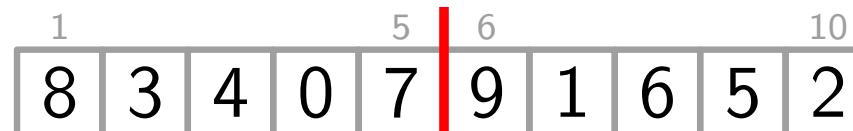
`MergeSort(A, $m + 1$, r)`

`Merge(A, ℓ , m , r)`

} teile

} herrsche

} kombiniere



MergeSort – ein Beispiel

`MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)`

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$

 MergeSort(A, ℓ, m)

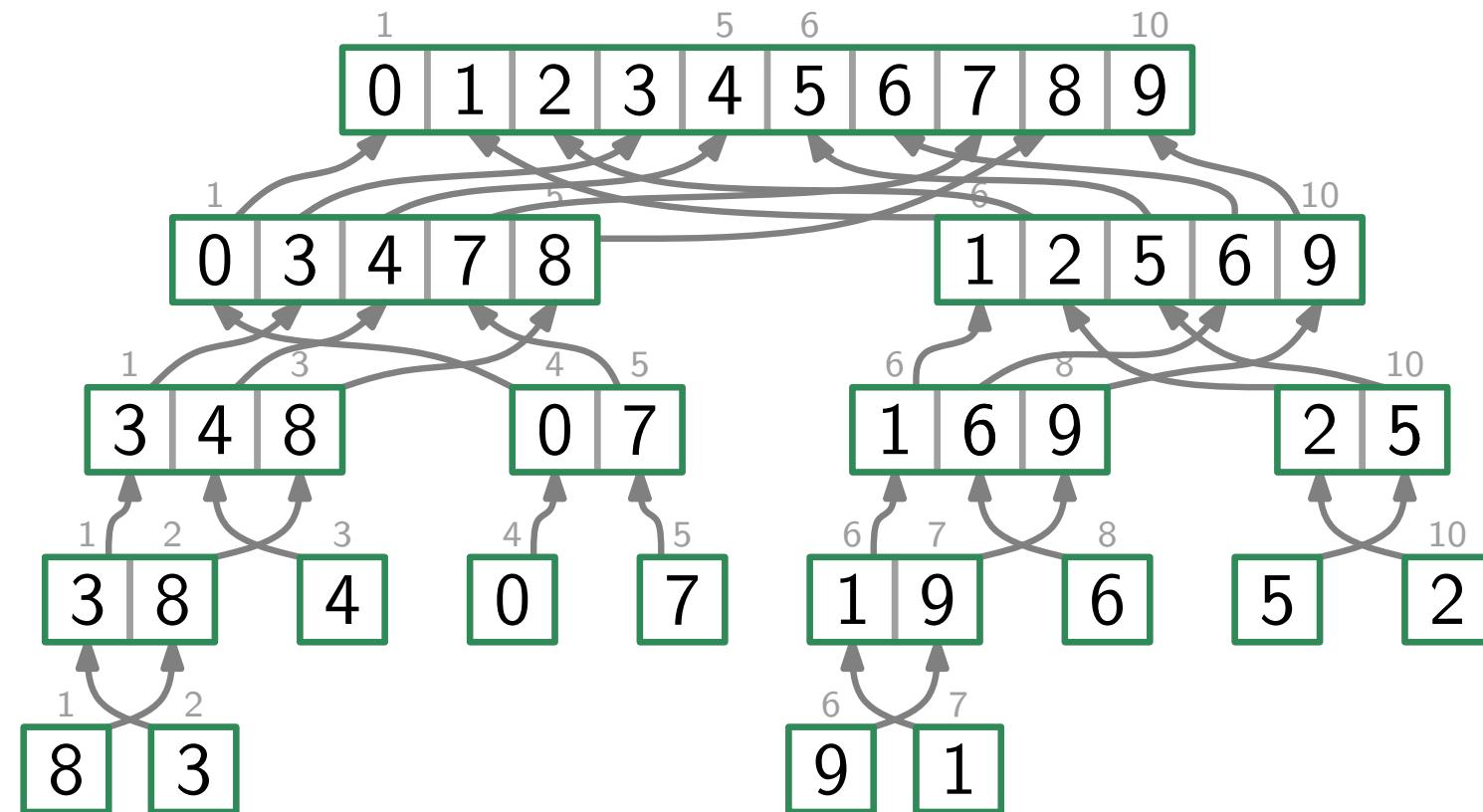
 MergeSort($A, m + 1, r$)

 Merge(A, ℓ, m, r)

} teile

} herrsche

} kombiniere



MergeSort – ein Beispiel

`MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)`

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$

 MergeSort(A, ℓ, m)

 MergeSort($A, m + 1, r$)

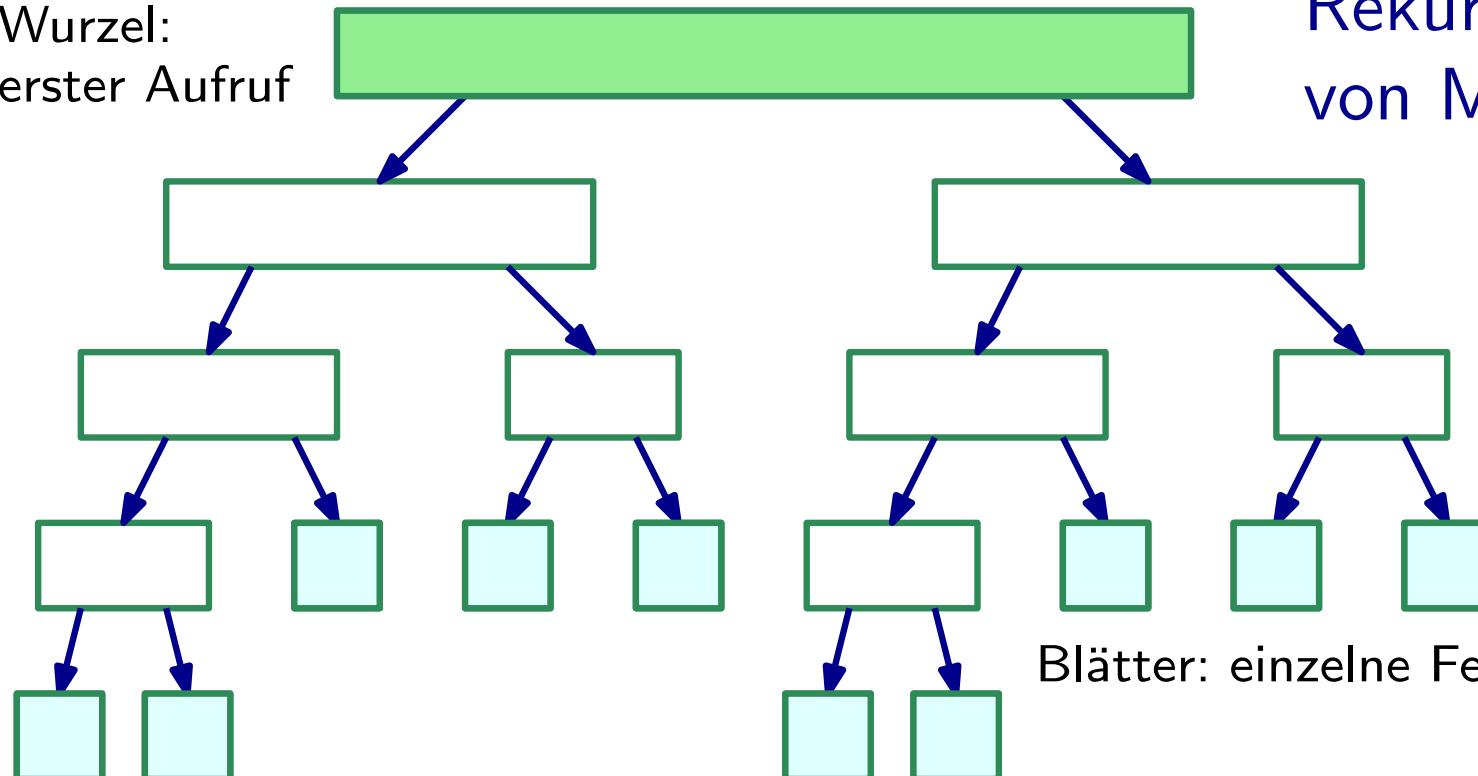
 Merge(A, ℓ, m, r)

} teile

} herrsche

} kombiniere

Wurzel:
erster Aufruf



Rekursionsbaum
von MergeSort:

Baum der
rekursiven
Aufrufe

Blätter: einzelne Feldelemente

Korrektheit von Mergesort

MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$

MergeSort(A, ℓ, m)

MergeSort($A, m + 1, r$)

Merge(A, ℓ, m, r)

} teile

} herrsche

} kombiniere

Korrekt? Welche Beweistechnik? Hm, MergeSort ist *rekursiv*...

Vollständige Induktion über $n = r - \ell + 1$ ($= A[\ell..r].length$):

$n = 1$: *Induktionsanfang*

Dann ist $\ell = r$.

\Rightarrow if-Block wird nicht betreten.

D.h. nichts passiert.

OK, da $A[\ell..\ell]$ schon sortiert.



Korrektheit von Mergesort

`MergeSort(int[] A, int $\ell = 1$, int $r = A.length$)`

if $\ell < r$ **then**

$m = \lfloor (\ell + r)/2 \rfloor$

`MergeSort(A, ℓ, m)`

`MergeSort(A, $m + 1, r$)`

`Merge(A, ℓ, m, r)`

} teile

} herrsche

} kombiniere

$n > 1$: Induktionsschritt

Induktionsannahme: MergeSort korrekt für Felder d. Länge $< n$.

Wegen $n > 1$ ist $\ell < r$. \Rightarrow if-Block wird betreten.

Nach Wahl von m gilt $\ell \leq m < r$.

$\Rightarrow A[\ell..m]$ und $A[m + 1..r]$ sind kürzer als $A[\ell..r]$.

\Rightarrow I.A. `MergeSort(A, ℓ, m)` ist korrekt und
 $\quad\quad\quad$ `MergeSort(A, $m + 1, r$)` ist korrekt. } `MergeSort(A, ℓ, r)`
 ist korrekt, d.h. MS

Schon bewiesen: `Merge` ist korrekt. } für Felder d. Länge n . \square

Übersicht

Techniken für Korrektheitsbeweise

- iterative Algorithmen (à la InsertionSort)

Schleifeninvariante (Schema „F“)

- rekursive Algorithmen (à la MergeSort)

Induktion

Algorithmen und Datenstrukturen

Wintersemester 2018/19
3. Vorlesung

Laufzeitanalyse

Recap: Diskutieren Sie mit Ihrer NachbarIn!

1. Was sind die zwei (oder drei?) entscheidensten Fragen, die wir uns über einen Algorithmus stellen?
2. Warum eigentlich interessieren wir uns fürs Sortieren?
3. Welche Entwurfstechniken für Algorithmen kennen wir schon?
4. Wie beweisen wir die Korrektheit
 - a) einer Schleife?
 - b) eines inkrementellen Algorithmus?
 - c) eines rekursiven Algorithmus?

Heute schon implementiert?
Zum Beispiel
– InsertionSort,
– MergeSort, ...
Probieren Sie's selbst!

Laufzeit analysieren: InsertionSort

```
InsertionSort(int[] A)
```

```
for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Zwei Konventionen:

- 1) $n :=$ Größe der Eingabe
= hier $A.length$
- 2) Wir zählen nur **Vergleiche!**
(zwischen Elementen der Eingabe)

Gesucht: Maß für die Laufzeit, das nur von n abhängt.

Problem: Tatsächliche Laufzeit hängt stark von Eingabe ab.

Lösung(?): Betrachte Extremfälle!

Bester Fall: A vorsortiert $\Rightarrow n - 1$ Vergleiche

Schlechtester Fall: A absteigend sortiert

$$\Rightarrow 1 + 2 + \cdots + (n - 1) = \frac{n^2 - n}{2} \text{ Vgl.}$$

Laufzeit von MergeSort

```
MergeSort(int[] A, int ℓ = 1, int r = A.length)
```

if $\ell < r$ **then**

$$m = \lfloor (\ell + r)/2 \rfloor$$

MergeSort(A, ℓ, m)

MergeSort($A, m + 1, r$)

Merge(A, ℓ, m, r)

} teile

} herrsche

} kombiniere

Korrekt? ✓

Effizient? ✓

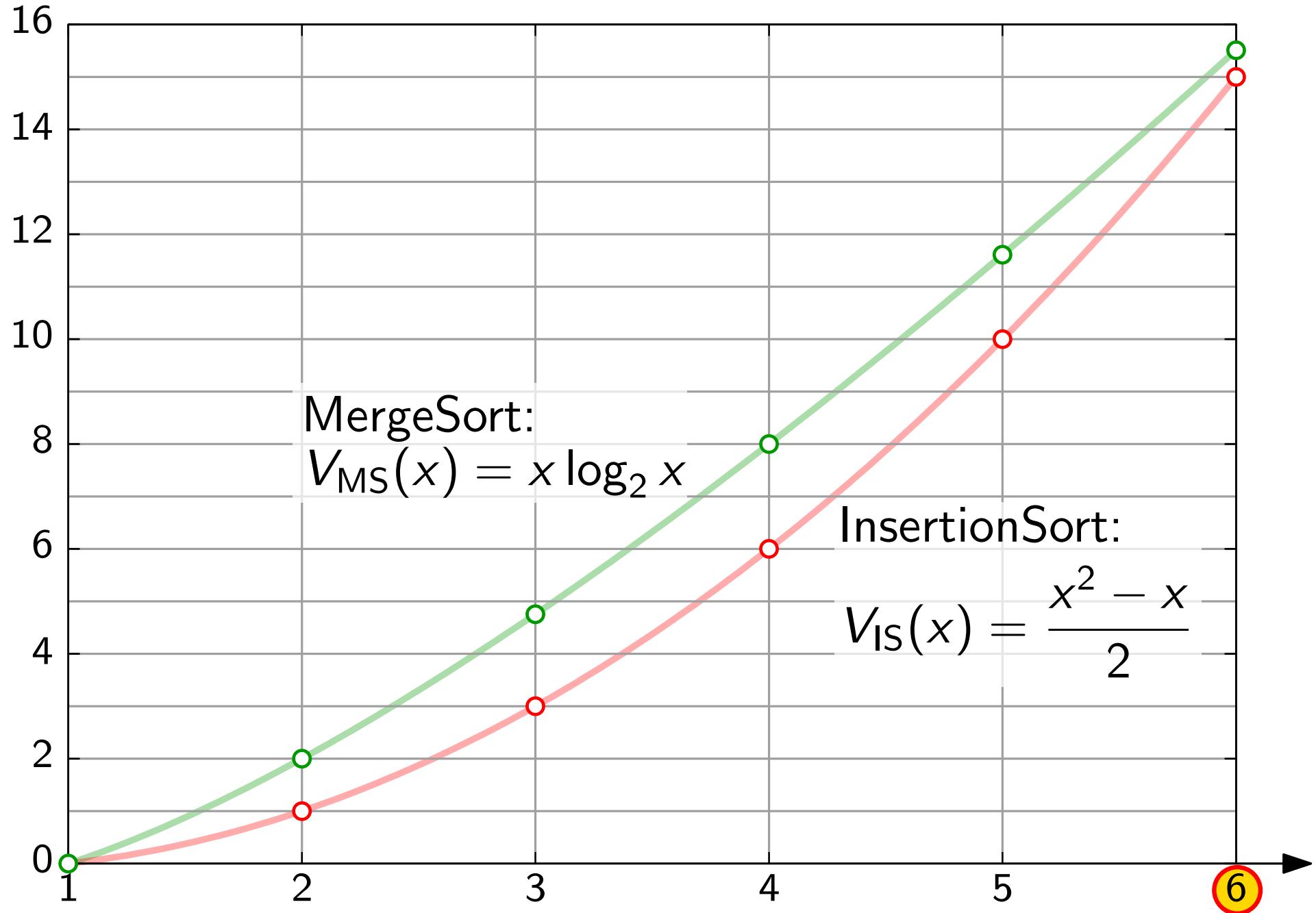
Sei $V_{\text{MS}}(n)$ die maximale Anzahl von Vergleichen, die MergeSort zum Sortieren von n Zahlen benötigt.

Dann gilt

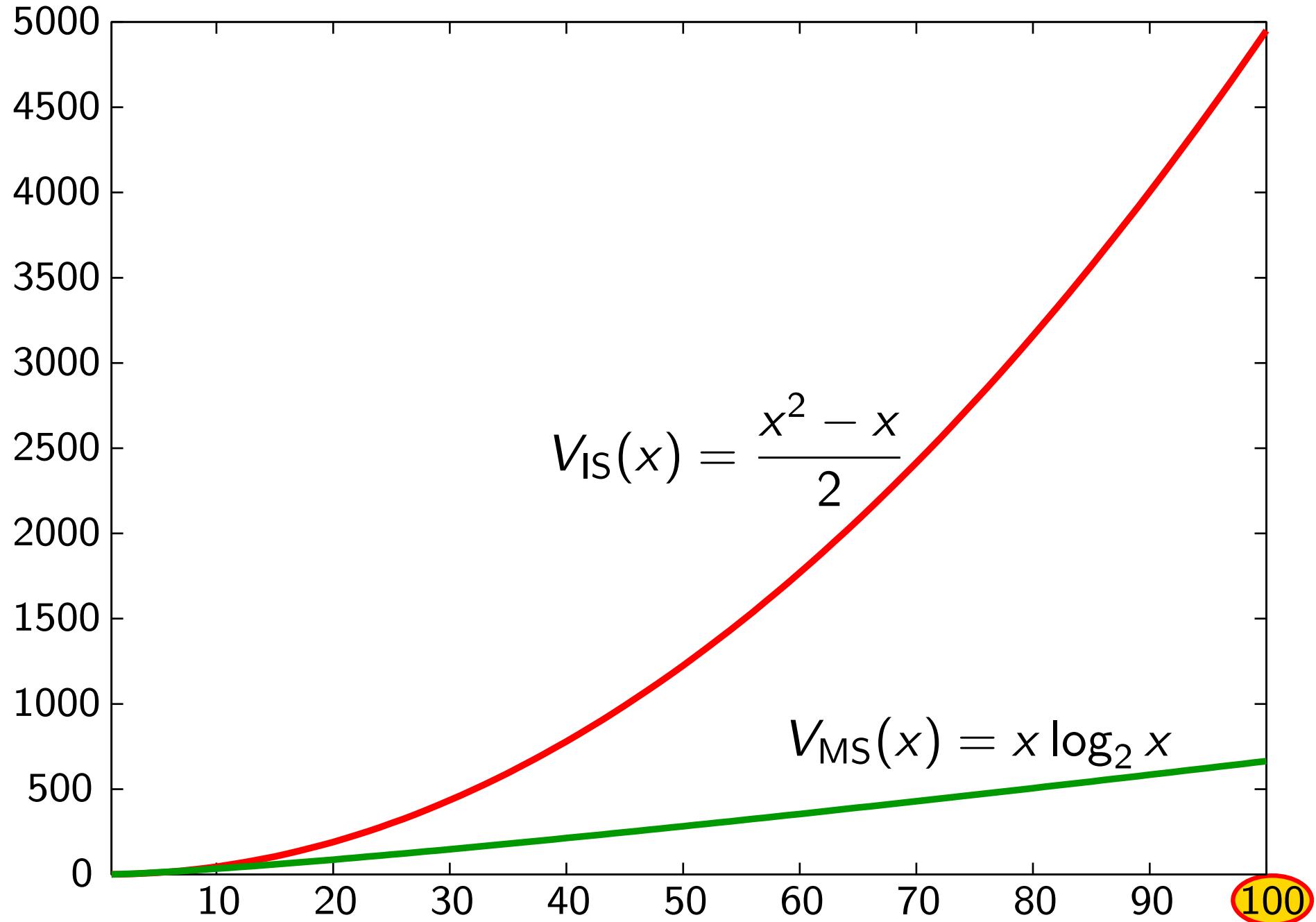
$$V_{\text{MS}}(n) = \begin{cases} 0 & \text{falls } n = 1, \\ 2V_{\text{MS}}(n/2) + n & \text{sonst.} \end{cases} = n \log_2 n$$

falls n
Zweierpotenz

Vergleich InsertionSort vs. MergeSort



Vergleich InsertionSort vs. MergeSort



Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g “

$$O(g) = \left\{ f: \mathbb{N} \rightarrow \mathbb{R} \mid \begin{array}{l} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass für alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die *höchstens* so schnell wachsen wie g .

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behaupt.: $f \in O(n^2)$; m.a.W. f wächst höchstens quadratisch.

Beweis. *Wähle* positive c und n_0 ,
so dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot n^2$.
 $f(n) = 2n^2 + 4n - 20 \leq 6n^2 \Rightarrow$ wähle $c = 6$.

da $4n \leq 4n^2$

Welches n_0 ? Aussage gilt für jedes $n \geq 0$. Nimm z.B. $n_0 = 1$.



Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g “

$$O(g) = \left\{ f: \mathbb{N} \rightarrow \mathbb{R} \mid \begin{array}{l} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass f\"ur alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die *h\"ochstens* so schnell wachsen wie g .

Beispiel. $f(n) = 2n^2 + 4n - 20$

negiere! (\neg)

Behaupt.: $f \notin O(n)$; m.a.W. f w\"achst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$, so dass $f(n) > c \cdot n$.

Ein Klassifikationsschema für Funktionen (I)

Definition.

Sei $g: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion. Dann ist „Groß-Oh von g “

$$O(g) = \left\{ f: \mathbb{N} \rightarrow \mathbb{R} \mid \begin{array}{l} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass f\"ur alle } n \geq n_0 \text{ gilt:} \\ f(n) \leq c \cdot g(n) \end{array} \right\}$$

die Klasse der Fkt., die *h\"ochstens* so schnell wachsen wie g .

Beispiel. $f(n) = 2n^2 + 4n - 20$

Behaupt.: $f \notin O(n)$; m.a.W. f w\"achst schneller als linear.

Beweis. Zeige: für alle pos. Konst. c und n_0 gibt es ein $n \geq n_0$,
so dass $f(n) > c \cdot n$.

Also: bestimme n in Abh. von c und n_0 , so dass
 $n \geq n_0$ und $f(n) = 2n^2 + 4n - 20 > c \cdot n$.

Fortsetzung des Beweises $f \notin O(n)$

Bestimme n in Abh. von c und n_0 , so dass $n \geq n_0$ und

$$f(n) = 2n^2 + 4n - 20 > c \cdot n.$$

Problem: Die „-20“ stört.

Aber wenn $n \geq 5$, dann gilt $4n - 20 \geq 0$.

D.h. wenn $n \geq 5$ und $2n^2 > cn$, dann $f(n) > cn$.

$$\begin{array}{c} \uparrow \\ n \geq 5 \end{array} \iff \boxed{n > c/2}$$

Wie wär's mit $n = c$?

Gut, aber wir müssen sicherstellen,
dass auch $n \geq 5$ und $n \geq n_0$ gilt.

Also nehmen wir $n = \lceil \max(c, 5, n_0) \rceil$.

Für dieses n gilt $n \geq n_0$ und $f(n) > cn$. Also gilt $f \notin O(n)$. \square

Ein Klassifikationsschema für Funktionen (II)

Definition.

Sei $g: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion. Dann ist „Groß-Omega von g “

$$\Omega(g) = \left\{ f: \mathbb{N} \rightarrow \mathbb{R} \mid \begin{array}{l} \text{es gibt positive Konstanten } c \text{ und } n_0, \\ \text{so dass f\"ur alle } n \geq n_0 \text{ gilt:} \\ c \cdot g(n) \leq f(n) \end{array} \right\}$$

die Klasse der Fkt., die *mindestens* so schnell wachsen wie g .

Beispiel. $f(n) = 2n^2 + 4n - 20$

Bewiesen: $f \notin O(n)$, $f \in O(n^2)$, $f \in O(n^3)$

Entsprechend: $f \in \Omega(n)$, $f \in \Omega(n^2)$, $f \notin \Omega(n^3)$

Zusammen:

$$f \in \Theta(n^2)$$

d.h. es gibt pos. Konst. c_1, c_2, n_0 , so dass f\"ur alle $n \geq n_0$ gilt:

$$c_1 \cdot n^2 \leq f(n) \leq c_2 \cdot n^2.$$

Das Klassifikationsschema – *intuitiv*

$f \in O(n^2)$	bedeutet f wächst <i>höchstens</i>	quadratisch.
$f \in \Omega(n^2)$	<i>mindestens</i>	
$f \in \Theta(n^2)$	„genau“	
$f \in o(n^2)$	<i>echt langsamer als</i>	
$f \in \omega(n^2)$	<i>echt schneller als</i>	



Genaue Definition für „klein-Oh“ und „klein-Omega“ s. Kap. 3, [CLRS].

Übung.

Gegeben folgende Funktionen $\mathbb{N} \rightarrow \mathbb{R}$ mit $n \mapsto \dots$:

$$n^2, \log_2 n, \sqrt{n \log_2 n}, 1,01^n, n^{\log_3 4}, \log_2(n \cdot 2^n), 4^{\log_3 n}.$$

Sortieren Sie nach Geschwindigkeit des *asymptotischen Wachstums*, also so, dass danach gilt: $O(\dots) \subseteq O(\dots) \subseteq \dots \subseteq O(\dots)$.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
4. Vorlesung

Laufzeitanalyse – Beispiele

Analyse von Aktienkursen



Wichtig:
Es genügt *nicht*
Minimum und
Maximum zu
suchen!

Problem. Gegeben: Folge $A[1..n]$ von Aktienkursen in Euro.

Gesucht: Paar (i, j) mit $1 \leq i < j \leq n$,
so dass $A[j] - A[i]$ maximal.

Verkaufskurs $\overbrace{A[j] - A[i]}$ Einkaufskurs
 Profit pro Aktie

Analyse von Aktienkursen

Problem: Gegeben: Folge $A[1..n]$ von ganzen Zahlen
Gesucht: Paar (i, j) mit $1 \leq i < j \leq n$,
 so dass $A[j] - A[i]$ maximal.

MAX-
DIFF

Lösung: per „roher Gewalt“

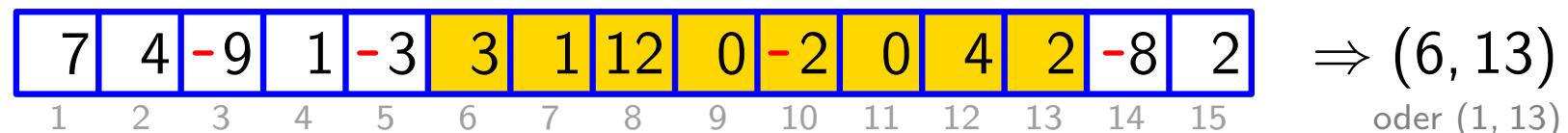
- für alle erlaubten Paare (i, j) berechne $A[j] - A[i]$
- gib Maximum zurück

Laufzeit \approx Anzahl erlaubter Paare =
 $= (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n^2 - n}{2} \in \Theta(n^2)$

Ein ähnliches Problem

Problem: Gegeben: Folge $A[1..n]$ von ganzen Zahlen
 Gesucht: Paar (i, j) mit $1 \leq i \leq j \leq n$,
 so dass $\sum_{k=i}^j A[k]$ maximal.

} MAX-SUM



Lösung: per „roher Gewalt“

Übung:
Schreiben Sie
Pseudocode!

- für alle erlaubten Paare (i, j) berechne $\sum_{k=i}^j A[k]$
- gib Maximum zurück

Laufzeit \approx Anzahl der Additionen

Obere Schranke dafür:

Untere Schranke (Anz. Paare)

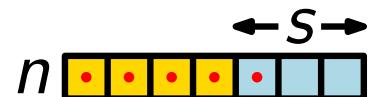
$$\begin{array}{c} O(n^2) \cdot O(n) = O(n^3) \\ \downarrow \quad \downarrow \\ \end{array} = \Omega(n^2)$$

Wo ist die Wahrheit?

Genauere Analyse

Laufzeit \approx Anzahl der Additionen des Rohe-Gewalt-Algos:

- für alle erlaubten Paare (i, j) berechne $\sum_{k=i}^j A[k]$
- gib Maximum zurück



Beob.

- Anz. der Summen mit s Summanden ist $n - s + 1$.
- s Summanden benötigen $s - 1$ Additionen.

$$\Rightarrow \text{Anz. Add.} = \sum_{s=1}^n (n - s + 1) \cdot (s - 1)$$

$$= n \cdot 0 + (n - 1) \cdot 1 + (n - 2) \cdot 2 + \dots + 2 \cdot (n - 2) + 1 \cdot (n - 1) \quad \leftarrow$$

$$= \dots + \underbrace{\frac{3n}{4} \cdot \frac{n}{4} + \dots + \frac{n}{2} \cdot \frac{n}{2} + \dots + \frac{n}{4} \cdot \frac{3n}{4}}_{\frac{n}{2} + 1 \text{ Terme der Größe mindestens } \frac{n}{4} \cdot \frac{n}{4}} + \dots \in \Omega(n^3)$$

$\frac{n}{2} + 1$ Terme der Größe mindestens $\frac{n}{4} \cdot \frac{n}{4}$

Übung:

Berechnen Sie diese Summe genau und beweisen Sie Ihr Ergebnis per Induktion!

\Rightarrow Der Rohe-Gewalt-Alg. läuft in $O(n^3) \cap \Omega(n^3) = \Theta(n^3)$ Zeit.

Can we do better?

Eine schnellere Lösung

Problem: Gegeben: Folge $A[1..n]$ von ganzen Zahlen

Gesucht: Paar (i, j) mit $1 \leq i \leq j \leq n$,

so dass $S_{ij} = \sum_{k=i}^j A[k]$ maximal.

Idee:

Für $i = 1, \dots, n$

berechne $S_{ii}, S_{i,i+1}, S_{i,i+2}, S_{i,i+3}, \dots, S_{i,n}$

$$= A[i] + A[i+1] + A[i+2] + A[i+3] + \dots + A[n]$$

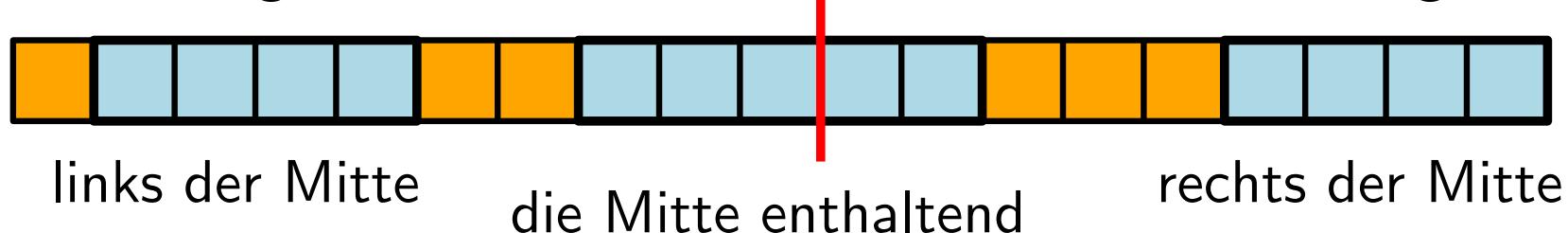
Wie? $A[i]$ $\underbrace{A[i+1] \quad A[i+2] \quad A[i+3] \quad \dots \quad A[n]}$

$n - i$ Additionen

Insgesamt $\sum_{i=1}^n (n - i) = \sum_{j=n-1}^0 j = \sum_{j=1}^{n-1} j \in \Theta(n^2)$ Add.

Eine noch schnellere Lösung?

Idee: Drei Möglichkeiten, wo maximale Teilsumme liegt:



Nimm Entwurfstechnik *Teile & Herrsche!*

- *teile*: in zwei ungefähr gleichgroße Hälften
- *herrsche*: durch rekursive Aufrufe für li. u. re. Hälfte
- *kombiniere*: kontrolliere **alle** Teilsummen, die die Mitte enthalten

Davon gibt's $\frac{n}{2} \cdot \frac{n}{2} \in \Theta(n^2)$

Einsicht: Wenn die *maximale* Teilsumme die Mitte enthält, dann muss ihr linker Teil (bis zur Mitte) maximal sein *und* dann muss ihr rechter Teil (ab der Mitte) maximal sein.
 ⇒ Wir können li. u. re. Teil *unabhängig* von einander berechnen!

Teile & Herrsche

`MaxTeilfeld(int[] A, int beginn = 1, int ende = A.length)`

```

if beginn == ende then
|   return (beginn, ende, A[beginn]) herrsche (in kleinen Teilinstanzen)
else

```

$mitte = \lfloor (beginn + ende)/2 \rfloor$	<i>teile</i>	<i>herrsche</i>
$(L\text{-}beginn, L\text{-}ende, L\text{-}summe) = MaxTeilfeld(A, beginn, mitte)$		
$(R\text{-}beginn, R\text{-}ende, R\text{-}summe) = MaxTeilfeld(A, mitte + 1, ende)$		
$(M\text{-}beginn, M\text{-}ende, M\text{-}summe) =$		
	MaxMittleresTeilfeld(A, beginn, mitte, ende)	←

return (Tripel mit größter Summe) *kombiniere*

Laufzeit: $T_{MT}(1) = \Theta(1)$

für $n > 1$: $T_{MT}(n) = T_{MT}(\lfloor n/2 \rfloor) + T_{MT}(\lceil n/2 \rceil) + T_{MMT}(n)$
 $\approx 2 \cdot T_{MT}(n/2) + T_{MMT}(n)$

$T_{MMT}(n) = ?$

Kombiniere

MaxMittleresTeilfeld(int[] A, int beginn, int mitte, int ende)

$L\text{-summe} = -\infty$

$summe = 0$

for $i = mitte$ **downto** $beginn$ **do**

Vervollständigen Sie
den Algorithmus!

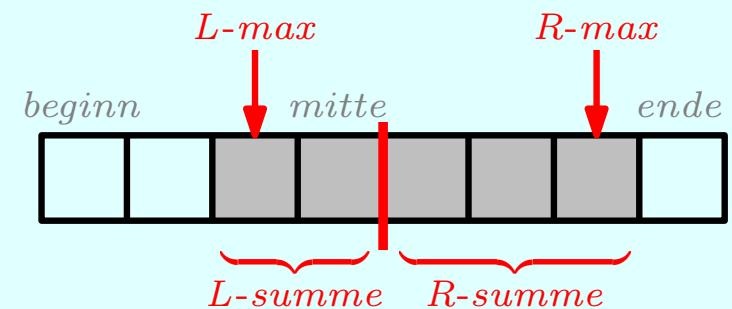
$R\text{-summe} = -\infty$

$summe = 0$

for $i = mitte + 1$ **to** $ende$ **do**

└ // analog zu oben

return ($L\text{-max}$, $R\text{-max}$, $L\text{-summe} + R\text{-summe}$)



Kombiniere

MaxMittleresTeilfeld(int[] A, int beginn, int mitte, int ende)

$L\text{-summe} = -\infty$

$summe = 0$

for $i = mitte$ **downto** $beginn$ **do**

$summe = summe + A[i]$

if $summe > L\text{-summe}$ **then**

$L\text{-summe} = summe$

$L\text{-max} = i$

$R\text{-summe} = -\infty$

$summe = 0$

for $i = mitte + 1$ **to** $ende$ **do**

// analog zu oben 

return ($L\text{-max}, R\text{-max}, L\text{-summe} + R\text{-summe}$)

Korrektheit? 

Schleifeninvariante:

$summe = S_{i,mitte}$ und

$L\text{-summe} =$

$\max_{i \leq k \leq mitte} S_{k,mitte}$

Laufzeit? 

$\mathbf{:=}_{\text{hier}}$ Anz. Additionen

$mitte - beginn + 1$

$ende - mitte$

$\underline{ende - beginn + 1}$

$= n$

Putting Things Together

Laufzeit von MaxTeilfeld:

$$T_{\text{MT}}(1) = \Theta(1)$$

für $n > 1$: $T_{\text{MT}}(n) \approx 2 \cdot T_{\text{MT}}(n/2) + T_{\text{MMT}}(n)$

$$= 2 \cdot T_{\text{MT}}(n/2) + n$$

$$= V_{\text{MS}}(n) = O(n \log_2 n)$$

Warum die Einschränkung wegfällt, sehen wir noch...

[für $n = \text{Zweierpotenz}$]

Denn für $a, b \geq 2$ gilt:
 $\Theta(\log_a n) = \Theta(\log_b n)$.



Denkaufgaben:

- Lösen Sie MaxTeilfeld in $O(n)$ – also in linearer – Zeit!
- Was hat MaxTeilfeld mit Aktienkursanalyse (vom Anfang der VL) zu tun?

Und wenn...? $T(n) = 2 \cdot T(n/2) + 4n$ (und $T(1) = \Theta(1)$)

Gilt dann auch $T(n) = O(n \log n)$?

Algorithmen und Datenstrukturen

Wintersemester 2018/19
5. Vorlesung

Rekursionsgleichungen

Lösen von Rekursions(un)gleichungen

Frage: Gilt für $T(n) = 2 \cdot T(n/2) + 4n$ (mit $T(1) = 0$)
 auch $T(n) \in O(n \log n)$?

Behauptung: Es gibt ein $c > 0$, so dass $T(n) \leq cn \log_2 n$.

Beweis. Durch Induktion über n . Ind.-Anfang: $T(1) \leq 0$ ✓
 Induktionsannahme: $T(k) \leq ck \log_2 k$ gilt für alle $k < n$.
 Wir wissen: $T(n) = 2T(n/2) + 4n$

Substitutionsmethode:

1. Lösung von
Rekursion raten
2. Mit Induktion
beweisen

$$\begin{aligned}
 &\stackrel{\leq}{=} 2c \frac{n}{2} \log_2 \frac{n}{2} + 4n \quad (\text{wegen IA}) \\
 &= cn \cdot (\log_2 n - \log_2 2) + 4n \\
 &= cn \log_2 n - cn + 4n \\
 &= cn \log_2 n + (4 - c)n \\
 &\leq cn \log_2 n \quad \text{falls } c \geq 4.
 \end{aligned}$$

⇒ Behauptung wahr (es gibt ein $c > 0...$) ⇒ $T(n) \in O(n \log n)$!

I) Substitutionsmethode

Noch'n Beispiel: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

Behauptung: $T(n) \in O(n)$ (mit $T(1) = 0$)

Also zeigen wir: $T(n) \leq cn$ für eine Konstante $c > 0$.

~~Beweis~~ Induktion über n .

IA: $T(k) \leq ck$ für alle $k < n$

Wissen: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$
 $\leq c \cdot \lfloor n/2 \rfloor + c \cdot \lceil n/2 \rceil + 1$ wg. IA

$\leq c \cdot (\lfloor n/2 \rfloor + \lceil n/2 \rceil) + 1$

$\leq c \cdot n + 1$



Noch'n Versuch

Noch'n Beispiel: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

Behauptung: $T(n) \in O(n)$ (mit $T(1) = 0$)

Also zeigen wir: $T(n) \leq cn + 1$ für eine Konstante $c > 0$.

~~Beweis~~ Induktion über n .

IA: $T(k) \leq ck + 1$ für alle $k < n$.

Wissen: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

$$\leq (c \cdot \lfloor n/2 \rfloor + 1) + (c \cdot \lceil n/2 \rceil + 1) + 1$$

$$\leq c \cdot n + 3$$



Nicht verzagen!

Selbes Beispiel: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

Behauptung: $T(n) \in O(n)$ ✓ (mit $T(1) = 0$)

Nun probieren wir: $T(n) \leq cn - d$ für Konstanten $c, d > 0$. ✓
 D.h. wir machen unsere Aussage schärfer!!

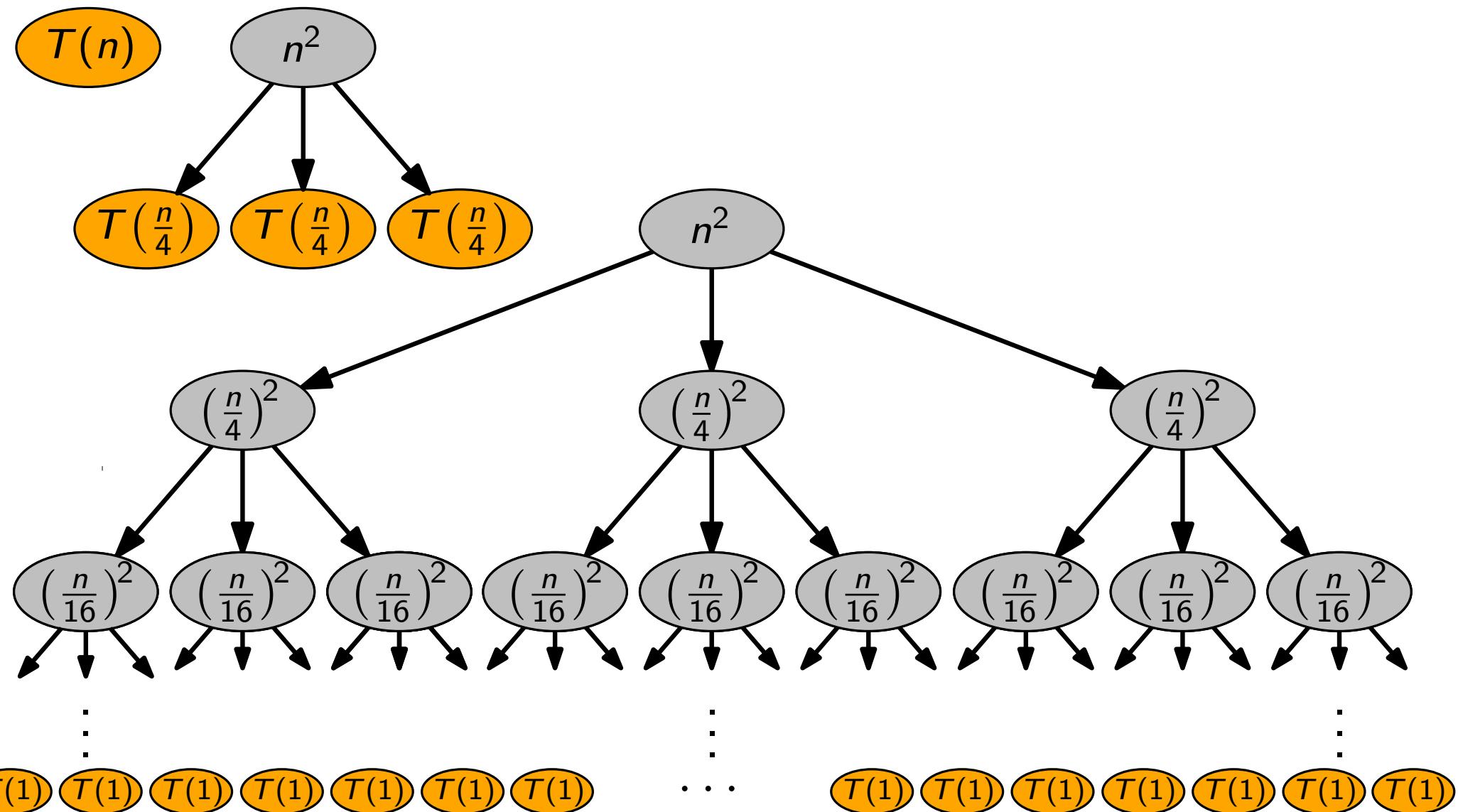
Beweis. Induktion über n .

IA: $T(k) \leq ck - d$ für alle $k < n$.

Wissen:
$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &\leq c \cdot (\lfloor n/2 \rfloor + \lceil n/2 \rceil) - d - d + 1 \\ &\leq cn - d + (1 - d) \\ &\leq cn - d \quad \text{falls } d \geq 1. \end{aligned}$$
 ✓

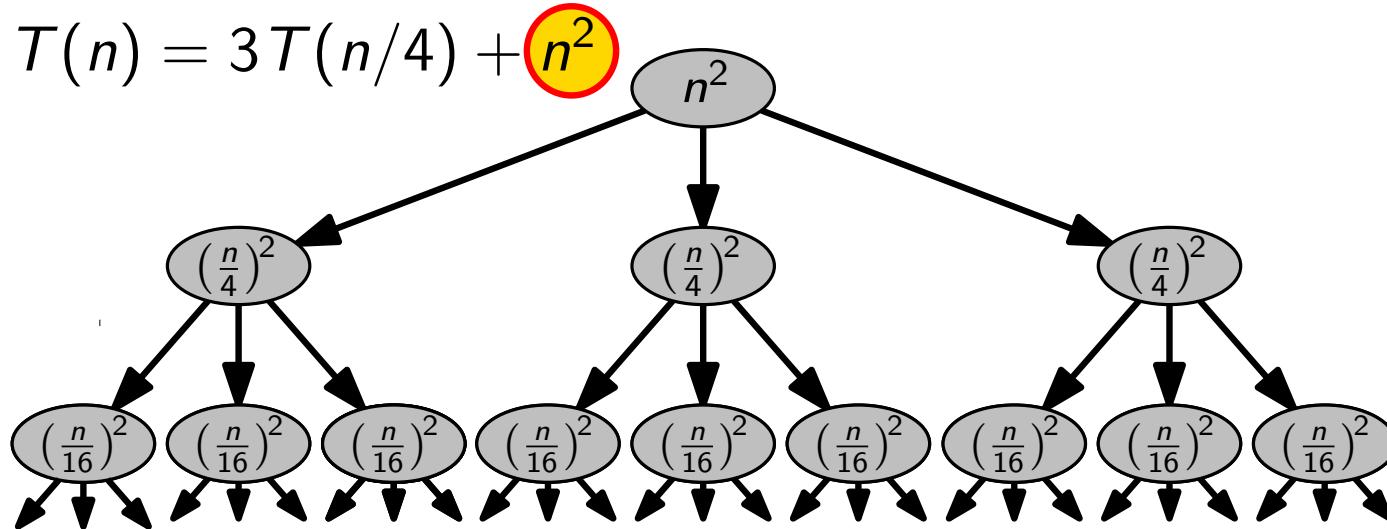
II) Rekursionsbaummethode

Beispiel: $T(n) = 3T(n/4) + n^2$ (*) (mit $T(1) = 1$)



II) Rekursionsbaummethode

$$T(n) = 3T(n/4) + n^2$$



lfd. Nr. Ebene	Anz. Knoten	Beitrag (Ebene)
0	$3^0 = 1$	n^2
1	3^1	$\frac{3}{16} n^2$
2	3^2	$\frac{3^2}{16^2} n^2$
:	:	:
i	3^i	$\frac{3^i}{16^i} n^2$
:	:	:
$\log_4 n$	$3^{\log_4 n}$	$n^{\log_4 3}$

vorausgesetzt
 $T(1) = 1$

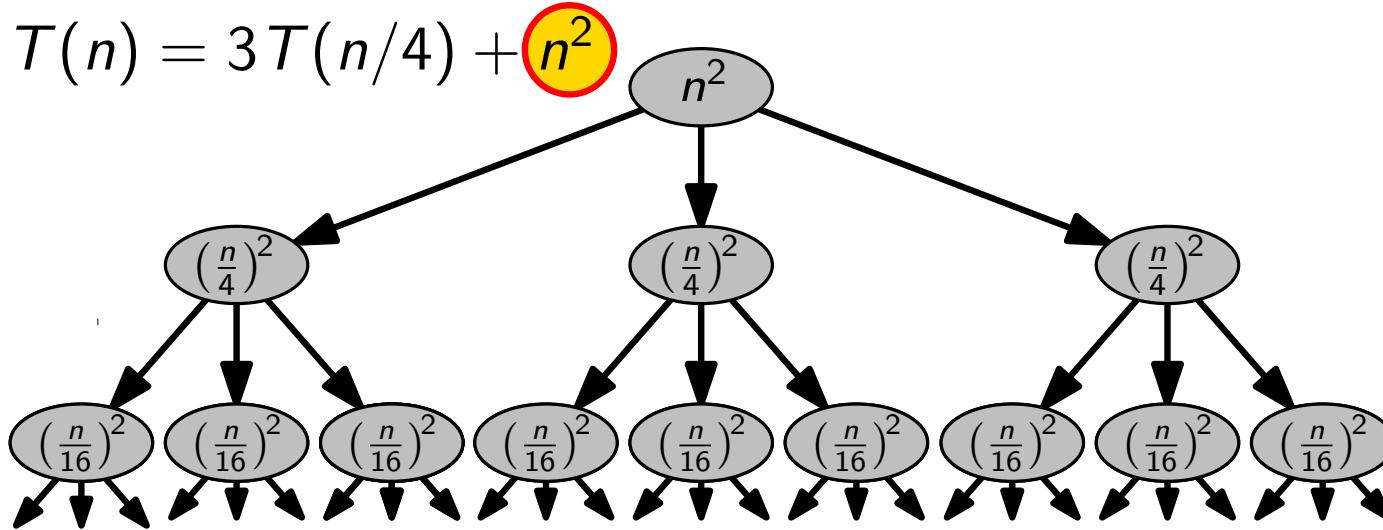
unterste Ebene andere Ebenen $(\log_4 n) - 1$

$\Rightarrow T(n) = n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)-1} \left(\frac{3}{16}\right)^i n^2 \leq n^{0,793} + n^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i = n^{0,793} + \frac{16}{13} n^2$

$\Rightarrow T \in \Theta(n^2)!$

II) Rekursionsbaummethode

$$T(n) = 3T(n/4) + n^2$$



Berechnen Sie mit der
Rekursionsbaummethode
 $T(n) = 2T(n/2) + n \log_2 n$,
wobei $T(1) = 0$.

$T(1) T(1) T(1) T(1) T(1) T(1) \dots$

$T(1) T(1) T(1) T(1) T(1) T(1)$

unterste Ebene

andere Ebenen

$$(\log_4 n) - 1$$

0. Summand schon 1 n^2 !

geometr. Reihe!!!

$$\Rightarrow T(n) = n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)-1} \left(\frac{3}{16}\right)^i n^2 \leq n^{0,793} + n^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i = n^{0,793} + \frac{16}{13} n^2$$

$\Rightarrow T \in \Theta(n^2)!$

Ifd.Nr. Ebene	Anz. Knoten	Beitrag (Ebene)
0	$3^0 = 1$	n^2
1	3^1	$\frac{3}{16} n^2$
2	3^2	$\frac{3^2}{16^2} n^2$
:	:	:
i	3^i	$\frac{3^i}{16^i} n^2$
:	:	:
$\log_4 n$	$3^{\log_4 n}$	$n^{\log_4 3}$
	$=$	
	$n^{\log_4 3}$	vorausgesetzt $T(1) = 1$

III) Meistermethode

Nichts ist praktischer als eine gute Theorie...

Achtung!

Die Methode kann man nur anwenden bei Rekursionen der Art

$$T(n) = aT(n/b) + f(n)$$

wobei $a \geq 1$, $b > 1$ Konst. und $f: \mathbb{N} \rightarrow \mathbb{R}$ asymptotisch positiv...

...und auch da nicht in allen Fällen!

III) Meistermethode

Satz: Seien $a \geq 1$, $b > 1$ Konstanten und $f, T: \mathbb{N} \rightarrow \mathbb{R}$ mit

$$T(n) = aT(n/b) + f(n),$$

wobei n/b sowohl für $\lfloor n/b \rfloor$ als auch $\lceil n/b \rceil$ stehen kann.

Dann gilt

$$T \in \begin{cases} \Theta(n^{\log_b a}) & \text{falls } f \in O(n^{(\log_b a)-\varepsilon}) \text{ für ein } \varepsilon > 0. \\ \Theta(n^{\log_b a} \log n) & \text{falls } f \in \Theta(n^{\log_b a}). \\ \Theta(f) & \text{falls } f \in \Omega(n^{(\log_b a)+\varepsilon}) \text{ für ein } \varepsilon > 0 \\ & \text{und die Regularitätsbedingung gilt.} \end{cases}$$

Definition: Die *Regularitätsbedingung* ist erfüllt, falls

$$af(n/b) \leq cf(n)$$

für ein $c < 1$ und für alle großen n .

III) Meistermethode

$$T(n) = aT(n/b) + f(n)$$

$$T \in \begin{cases} \Theta(n^{\log_b a}) & \text{falls } f \in O(n^{(\log_b a)-\varepsilon}) \text{ für ein } \varepsilon > 0. \\ \Theta(n^{\log_b a} \log n) & \text{falls } f \in \Theta(n^{\log_b a}). \\ \Theta(f) & \text{falls } f \in \Omega(n^{(\log_b a)+\varepsilon}) \text{ für ein } \varepsilon > 0 \\ & \text{und die Regularitätsbedingung gilt.} \end{cases}$$

Beispiel: $T(n) = 3T(n/4) + n^2$

$$\Rightarrow a = 3 (\geq 1), b = 4 (> 1) \text{ und } f: n \mapsto n^2.$$

$$\Rightarrow f \in \overset{\Omega}{?}(n^{(\log_4 3) \pm \varepsilon}), \text{ z.B. für } \varepsilon = 1, \text{ da } \log_4 3 < 1.$$

Das ist Fall 3! $\Rightarrow T \in \Theta(f) = \Theta(n^2)$ \square

Also müssen wir die Regularitätsbedingung testen:

$$3f(n/4) = \frac{3}{16}n^2 \leq c \cdot f(n) = cn^2, \text{ z.B. für } c = \frac{3}{16}.$$

Wichtig: Unser c muss *echt* < 1 sein!



Üben! Hausaufgaben!

Übersicht

- *Substitutionsmethode*

Für „Genies“: Lösung raten, dann per Induktion beweisen!

- *Rekursionsbaummethode*

Etwas umständlich, funktioniert aber immer!

- *Meistermethode*

Funktioniert nur bei Rekursionsgleichungen der Art
 $T(n) = aT(n/b) + f(n)$ (und auch da nicht immer).

Achtung: Viele verstehen die Bedeutung von ε in den Bedingungen der Fälle 1 & 3 nicht richtig!

Beispiel: $T(n) = 2T(n/2) + n \log_2 n$

Also können wir die Meistermethode hier nicht verwenden!

$$\Rightarrow n^{\log_b a} = n^{\log_2 2} = n^1$$

$$\text{aber } f(n) = n \log_2 n \notin \Omega(n^{1+\varepsilon}) !!$$

Grund: $\log n$ wächst langsamer als n^ε , für jedes $\varepsilon > 0$.

PS: Wie könnte man das beweisen?

Algorithmen und Datenstrukturen

Wintersemester 2018/19
6. Vorlesung

Prioritäten setzen

Heute: Wir „bauen“ eine Datenstruktur

Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung

wie wird die gewünschte Funktionalität realisiert:
– wie sind die Daten gespeichert (Feld, Liste, ...)?
– welche Algorithmen implementieren die Operationen?

Anwendung: Prozesssteuerung

Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anforderung:

- Prozesse (mit ihrer Priorität) einfügen
- Prozess mit höchster Priorität finden/löschen
- Priorität von Prozessen erhöhen

modelliere

Abstrakter Datentyp: **Prioritätsschlange**

verwaltet Elemente einer Menge M ,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M ,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

<i>Operation</i>	<i>Funktionalität</i>
Insert(element x)	$M = M \cup \{x\}$
element FindMax()	liefere $x \in M$ mit $x.key = \max\{y.key \mid y \in M\}$
element ExtractMax()	$x = \text{FindMax}(); M = M \setminus \{x\};$ liefere x
IncreaseKey (element x , priorität p)	$x.key = p$

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

W-C-Laufzeiten
meiner Implement.*

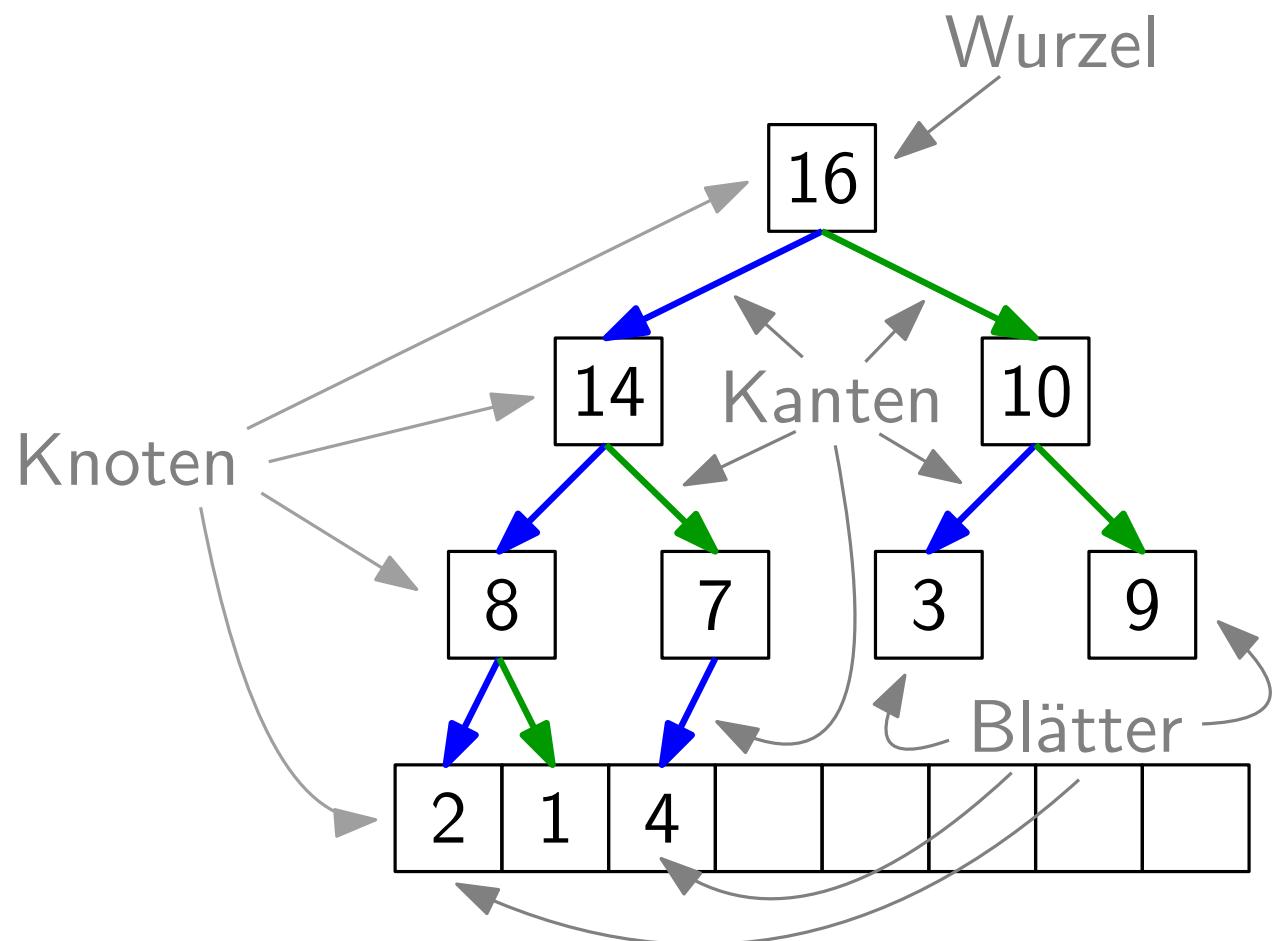
heute:
Implementierung
als Heap (Haufen)

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
heute: Implementierung als Heap (Haufen)	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

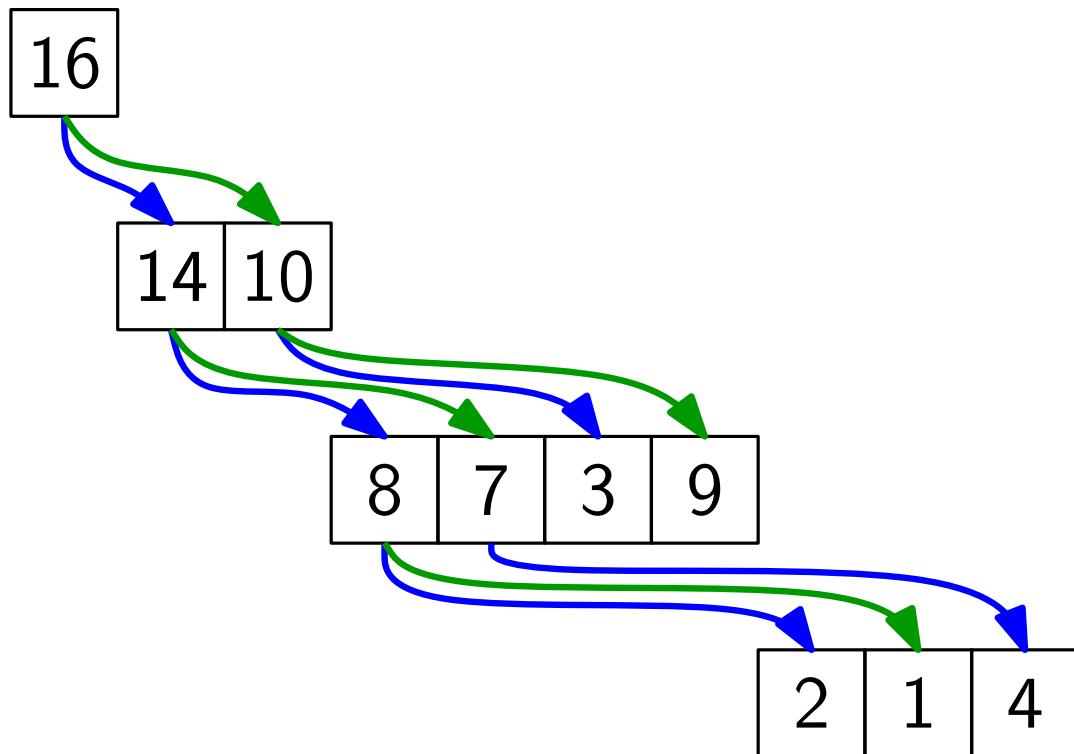
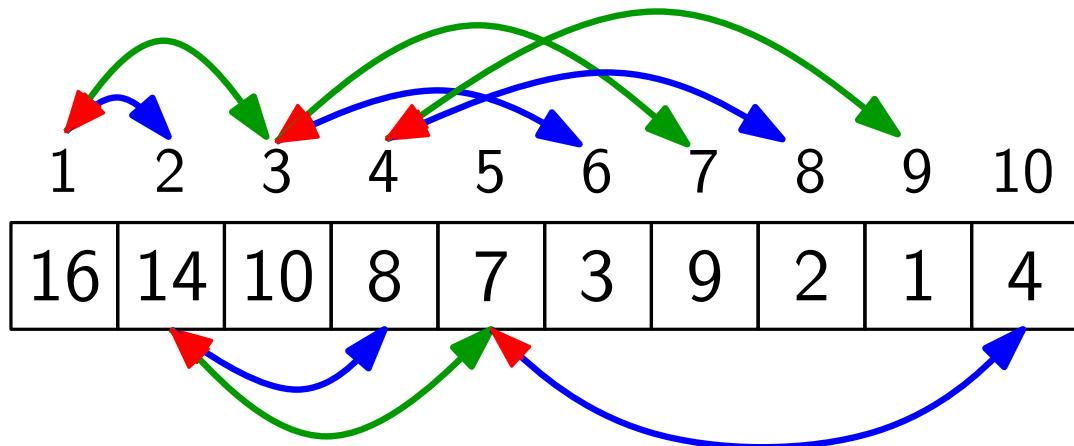
- * { – Daten werden in einem Feld gespeichert.
 – Neue Elemente werden hinten angehängt (unsortiert).
 – Maximum wird immer aufrechterhalten.
 – Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Achtung: Das Feld bekommt bei einer naiven Implementierung durch ExtractMin im Laufe der Zeit Lücken. Wie kann man das verhindern, ohne Elemente zu verschieben?

Bäume, gut gepackt



Bäume, gut gepackt



sehr schnelle Rechenoperationen!

Pfeile implementieren:

`left(index i)`

return

$2i$

`right(index i)`

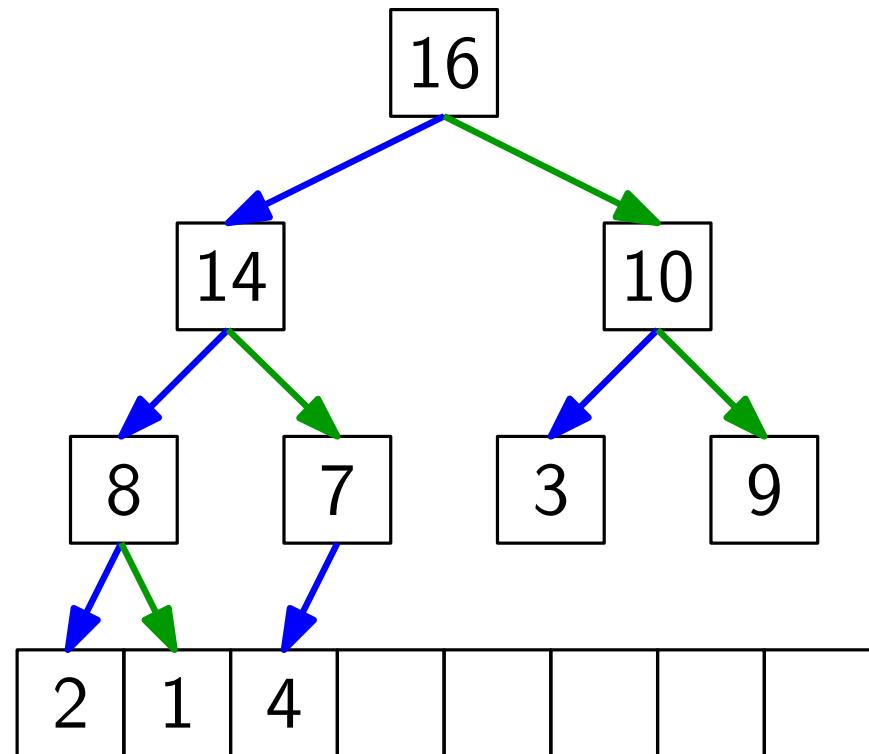
return

$2i + 1$

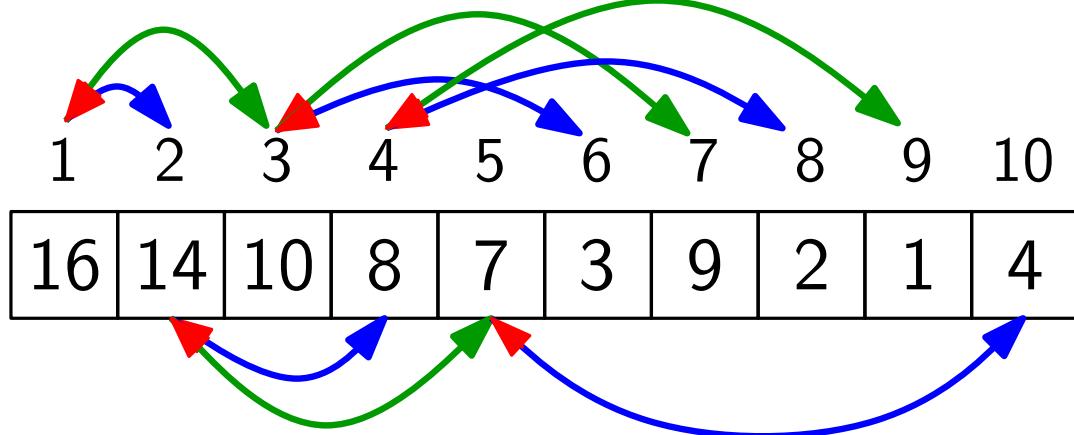
`parent(index i)`

return

$\lfloor i/2 \rfloor$



Bäume, gut gepackt



Definition:

Ein *Heap* ist ein Feld, das einem **binären** Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und
- die *Heap-Eigenschaft* gilt.

sehr schnelle Rechenoperationen!

Pfeile implementieren:

`left(index i)`

return

$2i$

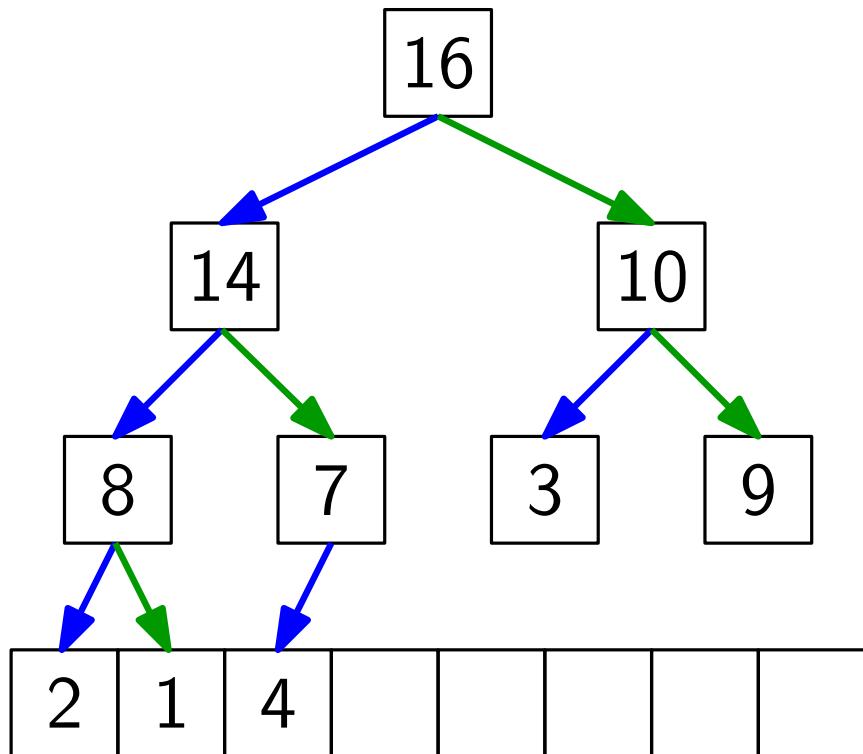
`right(index i)`

return

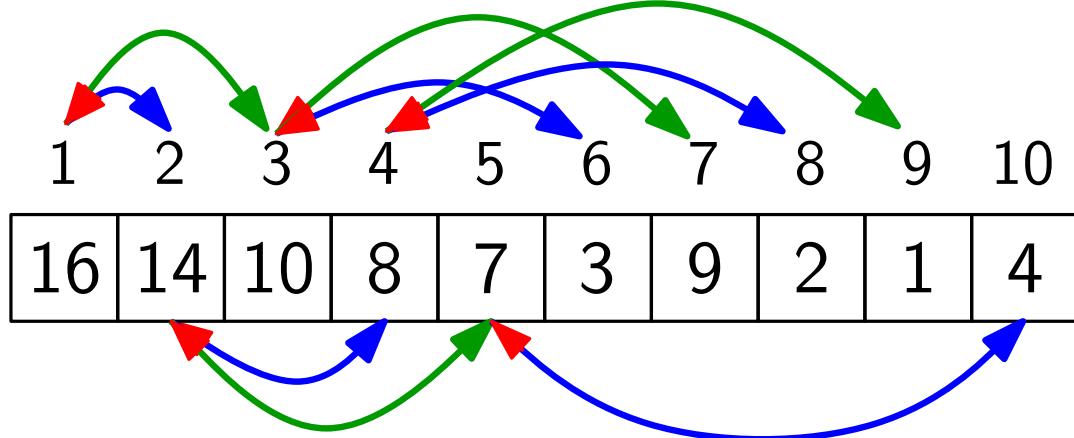
$2i + 1$

`parent(index i)` **return**

$\lfloor i/2 \rfloor$



Bäume, gut gepackt



Definition:

Ein Heap hat die

Max-Heap-Eigenschaft,

wenn für jeden Knoten $i > 1$ gilt:

$A[\text{parent}(i)] \geq A[i]$.

So ein Heap heißt *Max-Heap*.

sehr schnelle Rechenoperationen!

Pfeile implementieren:

`left(index i)`

return

$2i$

`right(index i)`

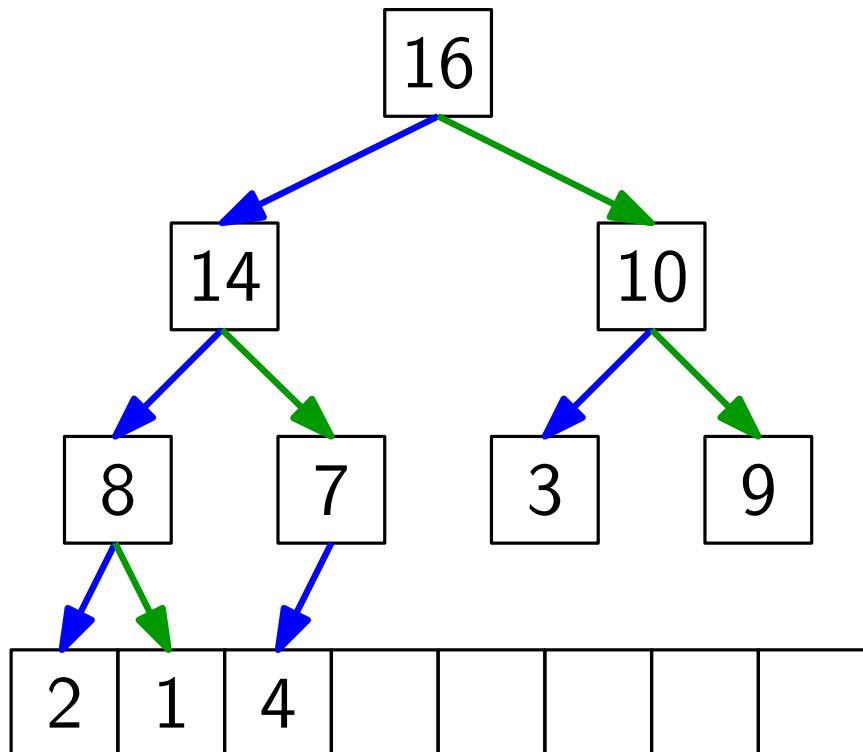
return

$2i + 1$

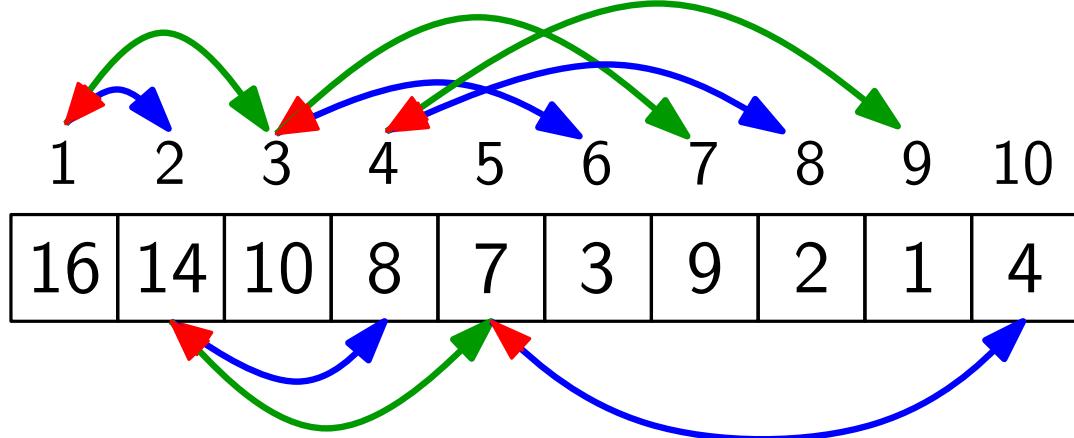
`parent(index i)`

return

$\lfloor i/2 \rfloor$



Bäume, gut gepackt



Definition:

Ein Heap hat die

~~Min~~ Max-Heap-Eigenschaft,

wenn für jeden Knoten $i > 1$ gilt:

$A[\text{parent}(i)] \geq A[i]$.

\leq

So ein Heap heißt ~~Max-Heap~~.

sehr schnelle Rechenoperationen!

Pfeile implementieren:

`left(index i)`

return

$2i$

`right(index i)`

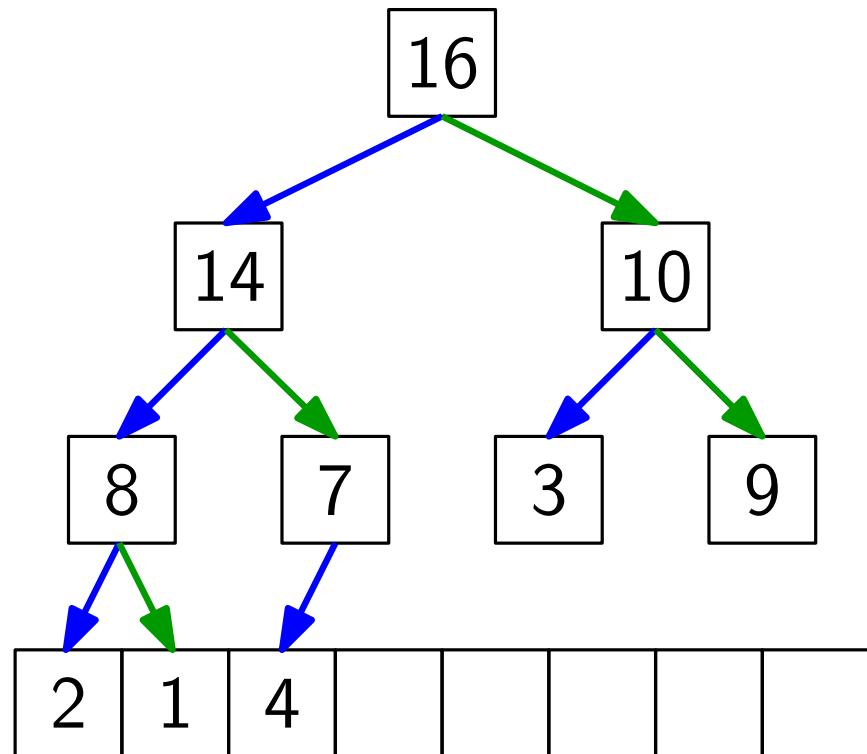
return

$2i + 1$

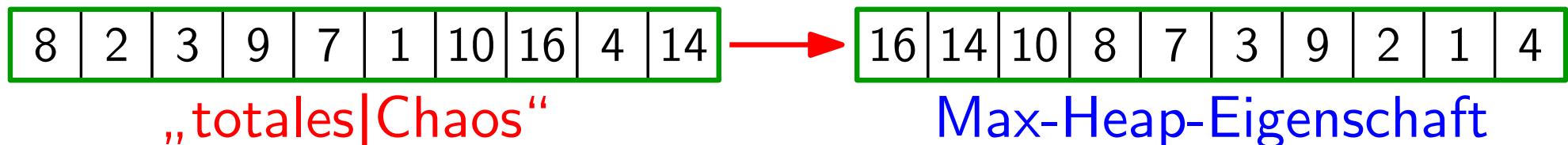
`parent(index i)`

return

$\lfloor i/2 \rfloor$

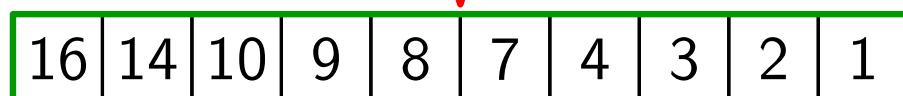


Baustelle



Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

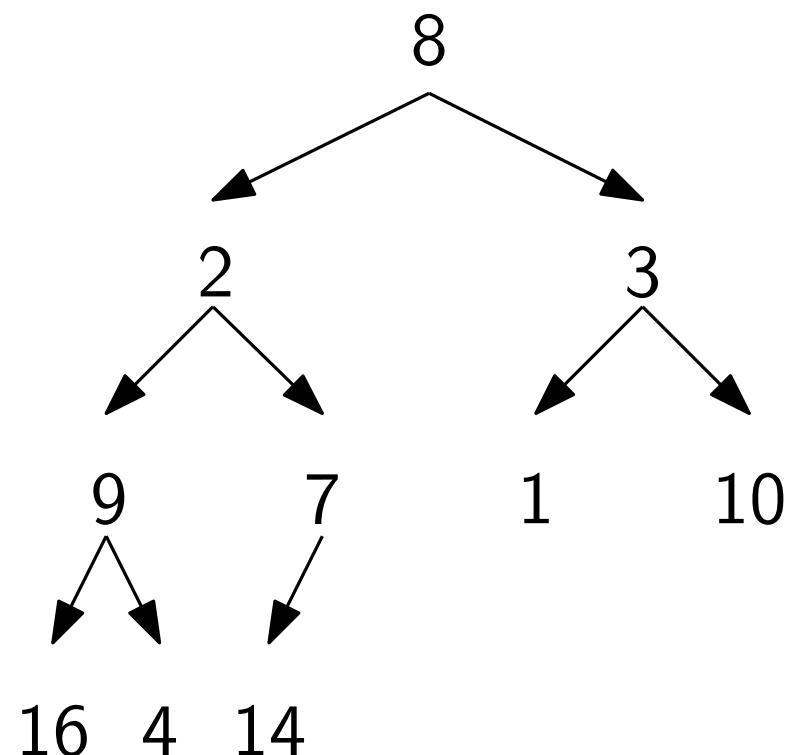
Nimm MergeSort!



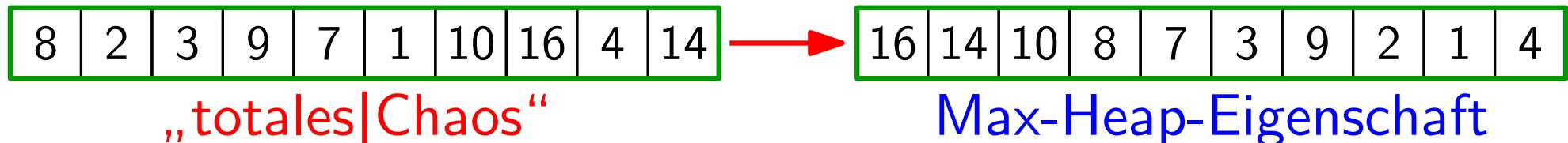
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...

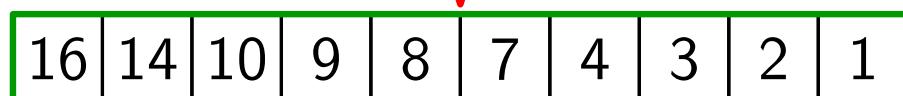


Baustelle



Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

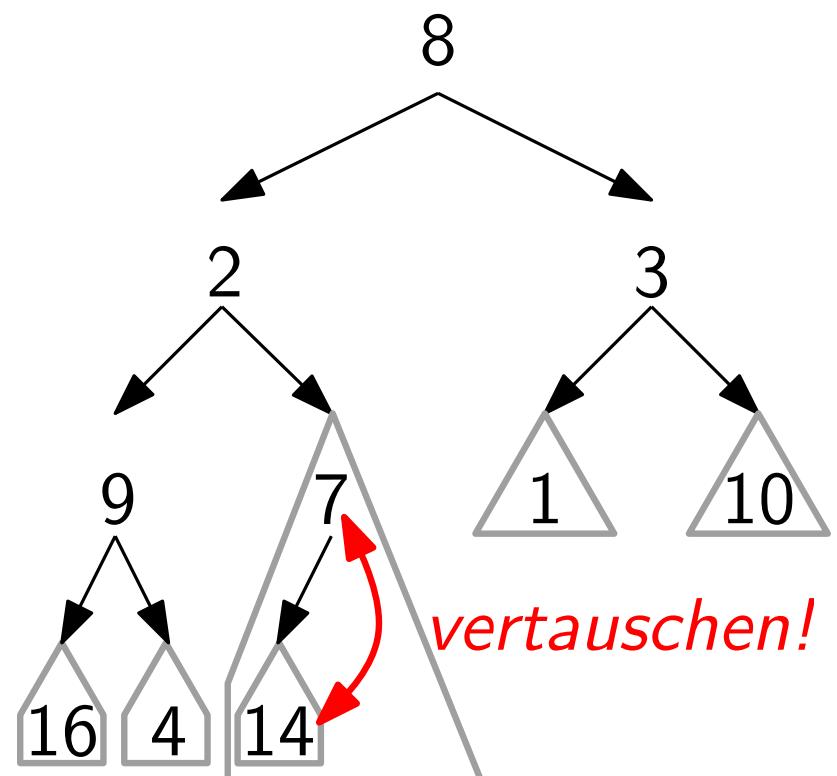


Absteigende Sortierung

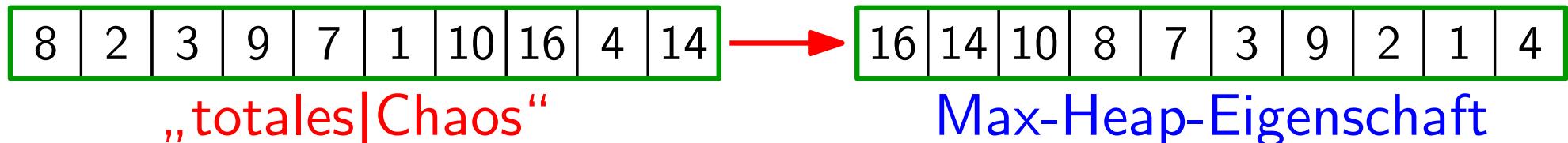
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...

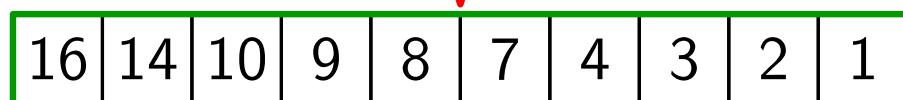


Baustelle



Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!



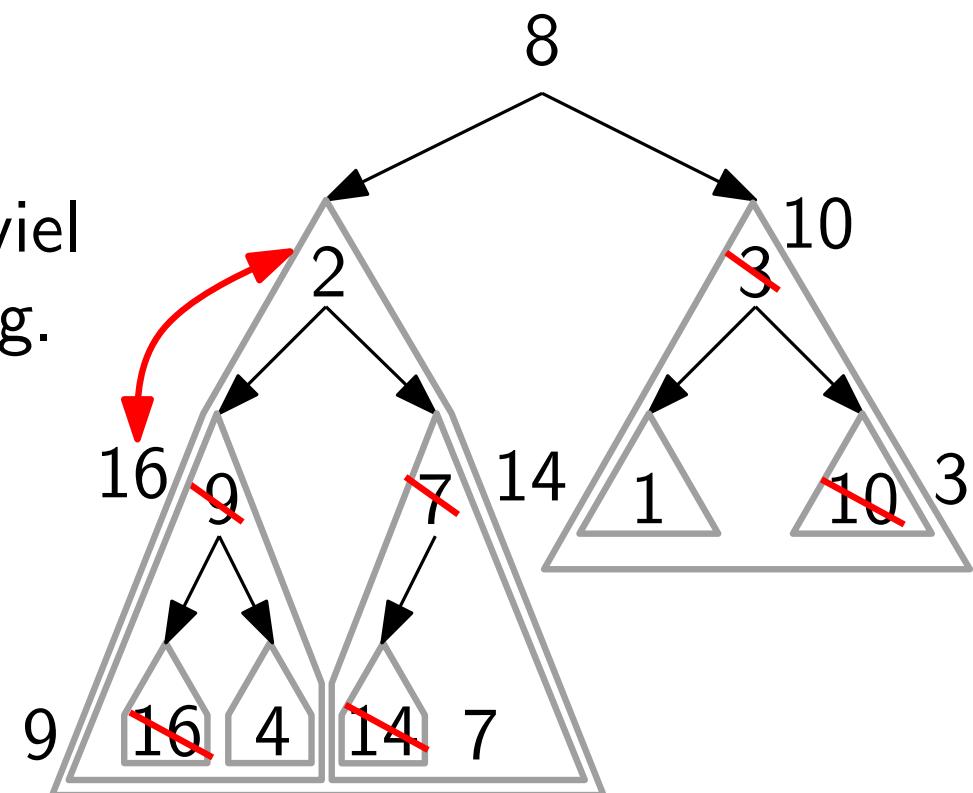
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

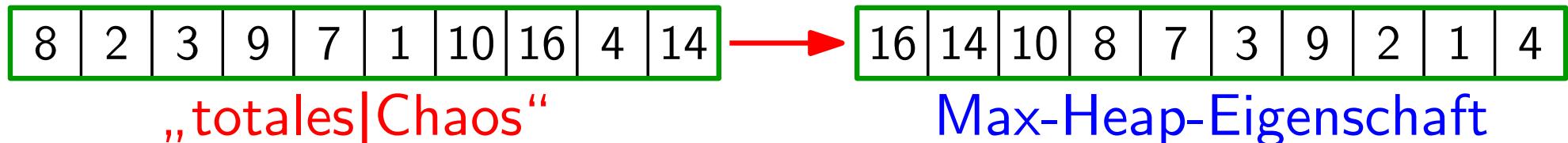
Idee: Nutze Baumstruktur!

Arbeite *bottom-up*:

Erst die Blätter...



Baustelle



Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

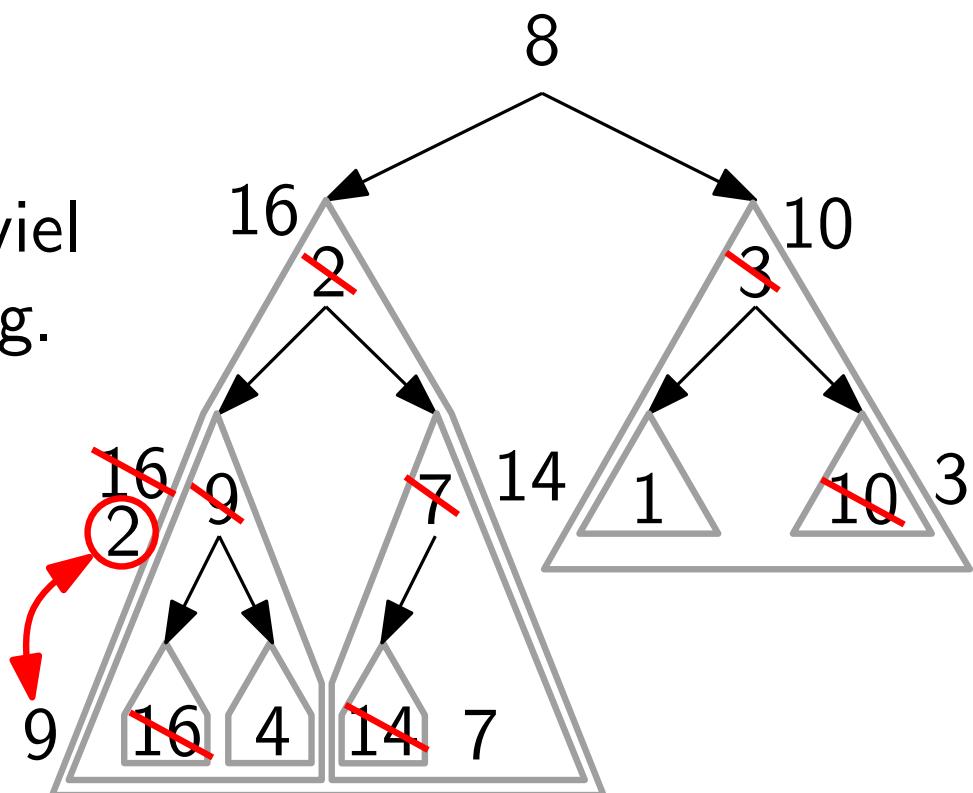
Nimm MergeSort!



Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14
---	---	---	---	---	---	----	----	---	----

„totales Chaos“

16	14	10	8	7	3	9	2	1	4
----	----	----	---	---	---	---	---	---	---

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

16	14	10	9	8	7	4	3	2	1
----	----	----	---	---	---	---	---	---	---

Absteigende Sortierung

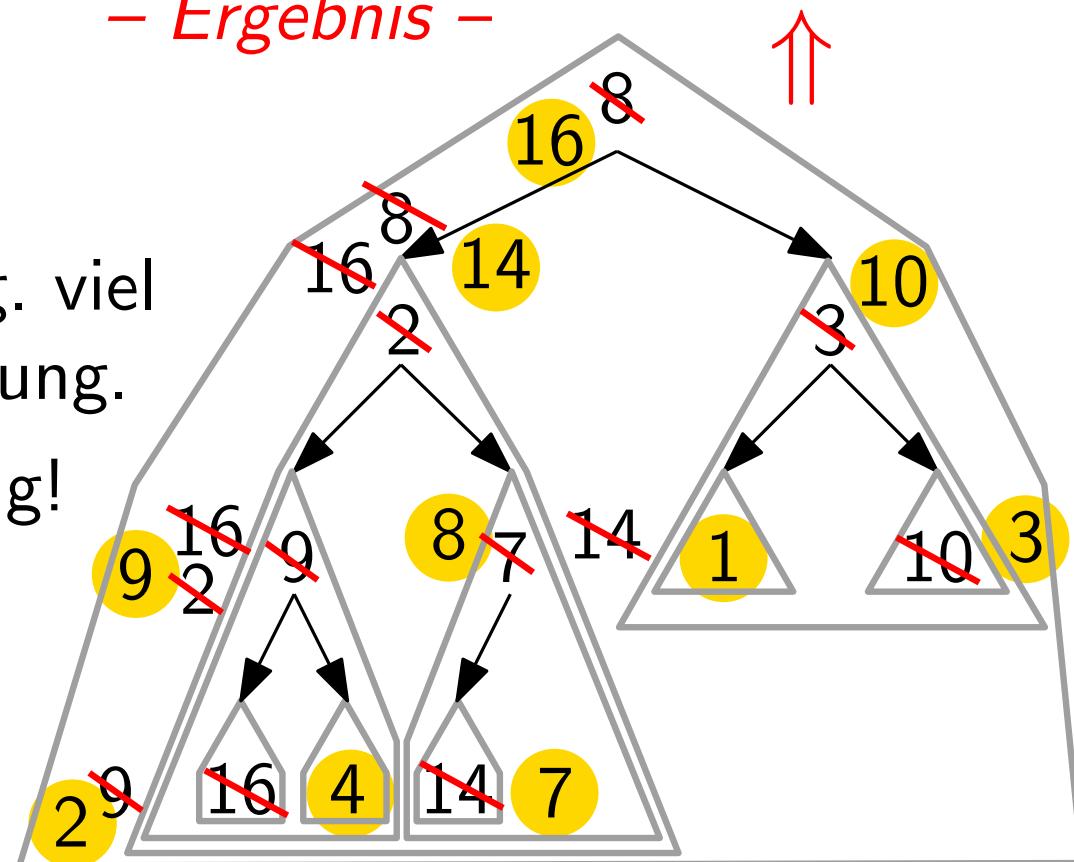
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...

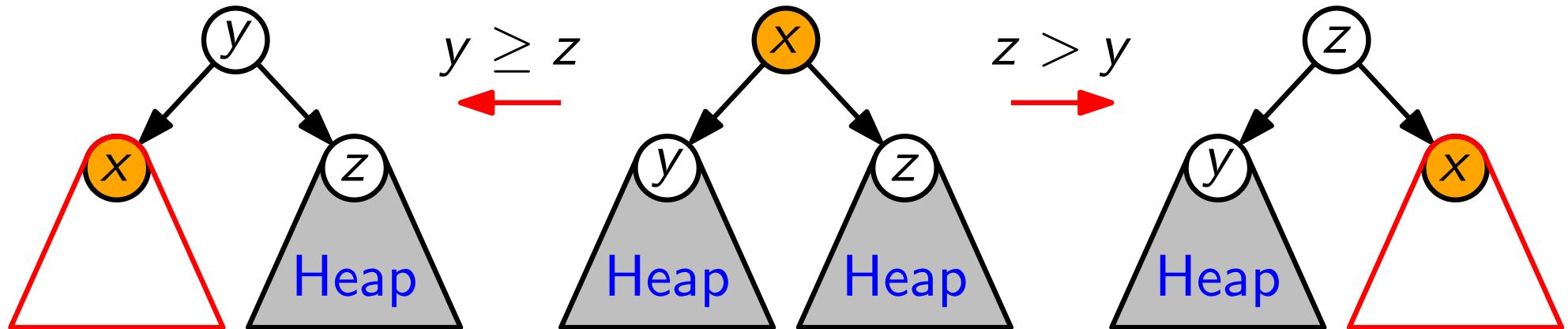
--	--	--	--	--	--	--	--	--	--

– Ergebnis –



Elementaroperation

„Versickere“ x , falls x zu klein, d.h. falls $x < \max(y, z)$



```
MaxHeapify(int A[], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*
Laufzeit? $T_{\text{MH}}(n, i)$

- := Anzahl der Swaps
- = Länge d. Weges v. $A[i]$
- ≤ Höhe von i im Teilheap mit Wurzel i
- = Höhe dieses Teilheaps

Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\text{MH}}(n, i) \leq$ Höhe des Teilheaps mit Wurzel i

Globale Strategie: *bottom-up*

BuildMaxHeap(int A[]])

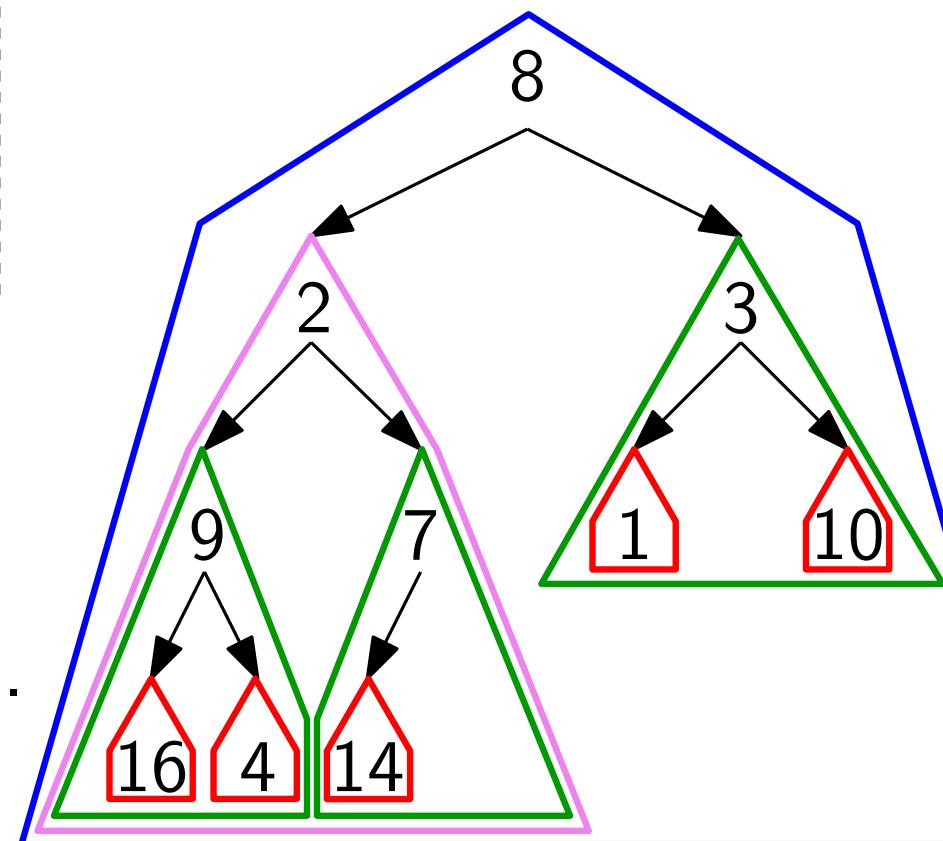
```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1 do
    MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\text{BMH}}(n) =$

$$\begin{aligned} &= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\text{MH}}(n, i) \\ &\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \dots \end{aligned}$$

$$= n \sum_{i=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^{i+1} \cdot i = ?$$



Forts. Laufzeitanalyse

$$T_{\text{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\frac{1}{2}\right)^{i+1} \stackrel{?}{=} \frac{n}{4} \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\frac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad (\text{falls } |x| < 1)$$

ableiten!

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$$

ableiten!

Quotientenregel:

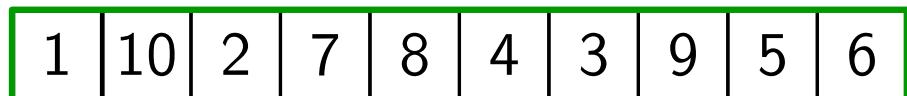
$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

$$\Rightarrow T_{\text{BMH}}(n) \leq \frac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^{i-1} = \frac{n}{4} \cdot \frac{1}{\left(\frac{1}{2}\right)^2} = n$$

Satz. Ein Heap von n Elementen kann in $\Theta(n)$ Zeit berechnet werden.

Übung Heap-Aufbau

Aufgabe: Bauen Sie einen Heap mit BuildMaxHeap!



BuildMaxHeap(int A[])

$A.heap\text{-size} = A.length$

for $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**
 └ MaxHeapify(A, i)

```
MaxHeapify(int A[], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

$O(\quad)$

return $A[1]$

Laufzeiten?

ExtractMax()

$O(\quad)$

if $A.\text{heap-size} < 1$ **then**
error "Heap underflow"

$max = A[1]$

$A[1] = A[A.\text{heap-size}]$

$A.\text{heap-size} --$

MaxHeapify($A, 1$)

return max

IncreaseKey(index i , prio. p) $O(\quad)$

if $p < A[i]$ **then error** "prio. too small"

$A[i] = p$

while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$

 swap($A, i, \text{parent}(i)$)

$i = \text{parent}(i)$

Insert(priorität p)

$O(\quad)$

$A.\text{heap-size} ++$

if $A.\text{heap-size} > A.\text{length}$ **then error...**

$A[A.\text{heap-size}] = -\infty$

IncreaseKey($A.\text{heap-size}, p$)

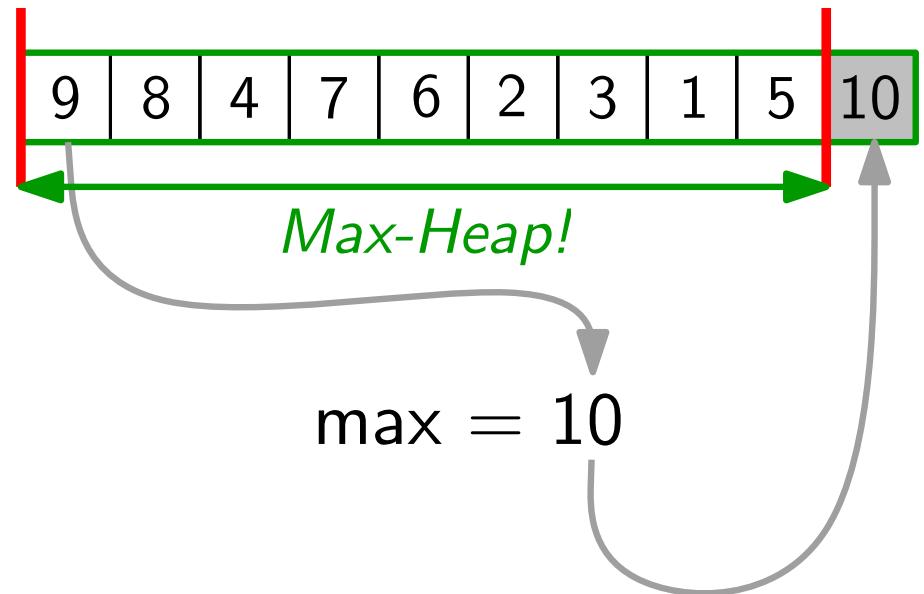
Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

HeapSort(A)

Schreiben Sie den Pseudocode.
Verwenden Sie BuildMaxHeap
und ExtractMax.



Vom Heap zur Sortierung

Idee:

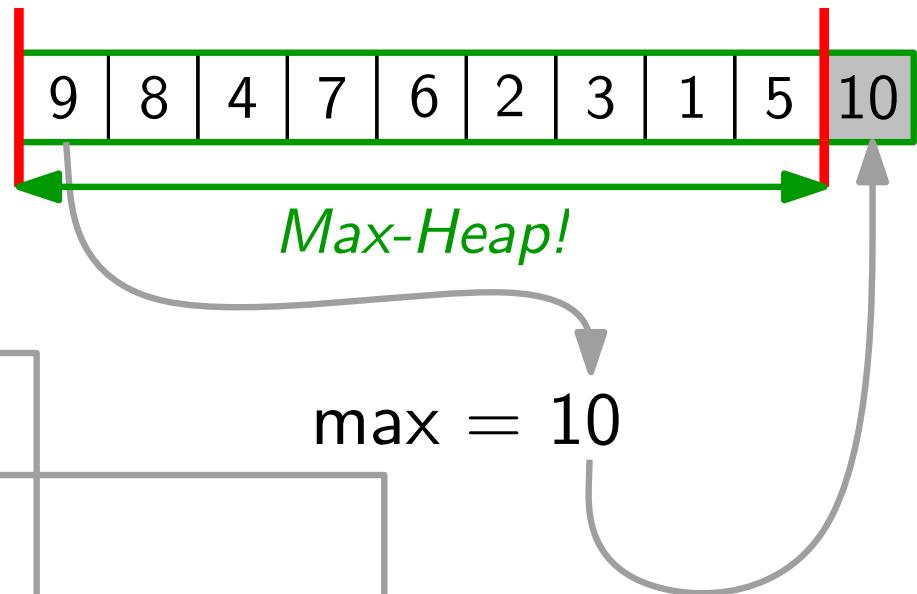
- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

```
HeapSort(A)
```

```
BuildMaxHeap(A)
```

```
for  $i = A.length$  downto 2 do
```

```
     $A[i] = \text{ExtractMax}()$ 
```



Laufzeit: $T_{HS}(n) \in O(n) + (n - 1) \cdot O(\log n) = O(n \log n)$

Satz.

HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ <i>Warum?</i>	$\Theta(n \log n)$	$\Theta(n \log n)$	
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
in situ ¹ (<i>in place</i>)	✓	✗	✓	
stabil ²	✓	✓	✗	

¹) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

²) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
7. Vorlesung

Zufall!

Guten Morgen!

Tipps für unseren ersten Test am Do, 22. November:

- Lesen Sie die Definitionen der Klassen O , Ω und Θ gaaaanz genau – bis Sie sie *restlos* verstehen! Besonders Beweise der Art $f \notin O(g)$ machen erfahrungsgemäß Schwierigkeiten.
- Lesen Sie alle Vorlesungsfolien (bis einschließlich Di, 6.11.) und Kap. 1–4 & 6 im Buch [CLRS]!
- Machen Sie möglichst viele Übungsaufgaben in Kap. 3, 4, 6 [CLRS]!
- Programmieren Sie – z.B. Pseudocode aus der Vorlesung!
- Stellen Sie Fragen – Kommilitonen, Tutoren, Erklärhiwis, mir!
- Haben Sie schon das/ein Buch?

Tipp: Das Buch

„Algorithmen & Datenstrukturen: Die Grundwerkzeuge“
von Dietzfelbinger, Mehlhorn und Sanders (Springer, 2014)
kann man im Uninetz kostenlos von der Unibib herunterladen.

Lesen!!

Was ganz (?) anderes:

- Erinnern Sie sich an die Linearzeitlösung für MaxSum?
- Beweisen Sie ihre Korrektheit mit einer Schleifeninvarianten!

Lösen der Übungsaufgaben

- Geben Sie auf Ihren Lösungen immer die Namen aller (≤ 3) Autoren an – nur die bekommen Punkte!
- Spezialfall: Keine Namen – **keine Punkte!**
- Geben Sie immer die Nummer Ihrer Übungsgruppe an – sonst gibt's ebenfalls keine Punkte!
- Lösen Sie Aufgaben möglichst nur mit Mitgliedern *Ihrer* Übungsgruppe. Nur so haben Sie die Lösungen bei der Besprechung vor sich liegen.
- Wenn Sie nicht immer *alle* Aufgaben lösen können – nicht verzweifeln! Wichtig ist, dass Sie's probiert haben!

Inhaltsverzeichnis

- Ein Zufallsexperiment
- InsertionSort: erwartete bzw. Durchschnittslaufzeit
- Das Geburtstagsparadoxon

Ein Experiment

Ein Franke und ein Münchner gehen (unabhängig voneinander) n mal in verschiedenen Restaurants essen und benoten nach jedem Besuch ihr Essen.

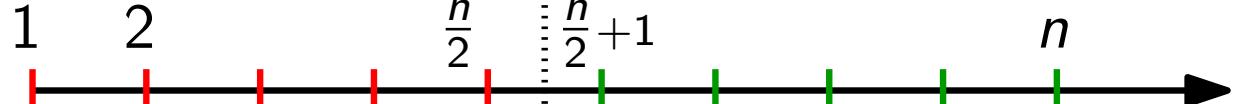
Der Franke ist zufrieden, wenn er überdurchschnittlich gut isst, d. h. wenn seine aktuelle Note über dem **Mittel** liegt.

Der Münchner ist zufrieden, wenn er besser isst als er jemals vorher gegessen hat.

Wer is(s)t zufriedener?

$$\frac{\sum_{i=1}^n i}{n} = \frac{n+1}{2}$$

Das Kleingedruckte:



Die Reihenfolge der Restaurants ist *zufällig*. Beide Gourmets müssen ihr Essen mit einer Zahl zwischen 1 (= sehr schlecht) und n (= sehr gut) bewerten und dürfen jede Zahl nur einmal vergeben. Die Zahl n sei gerade. Der Münchner ist beim ersten Essen zufrieden.

Modellierung

Ein Ergebnis unseres Experiments entspricht einer Permutation der Zahlen $1, 2, \dots, n$.

Sei S_n die Menge all dieser Permutationen. $\Rightarrow |S_n| = n!$

Ergebnismenge Ω ↗ ↘ *Beobachtungsmenge Ω'*

Sei $M: S_n \rightarrow \{1, \dots, n\}$ eine *Zufallsvariable*, die angibt, wie oft der Münchener zufrieden ist.

Uns interessiert der erwartete Wert von M ,
kurz: der *Erwartungswert $E[M]$* von M .

Definition: $E[M] = \sum_{i \in \Omega'} i \cdot \Pr[M = i]$ (Def. für *diskrete ZV*)

„gewichtetes Mittel“ der Werte in Ω'

Es gilt: $\sum_{i \in \Omega'} \Pr[M = i] = 1$

Problem: Was ist $\Pr[M = 7]??$

Ein Trick

Definition: $\mathbf{E}[M] = \sum_{i \in \Omega'} i \cdot \mathbf{Pr}[M = i]$

Führe *Indikator-Zufallsvariable* ein (für $i = 1, \dots, n$):

Sei $M_i = \begin{cases} 1, & \text{falls Münchener nach dem } i. \text{ Essen zufrieden,} \\ 0 & \text{sonst.} \end{cases}$

$$\Rightarrow \mathbf{E}[M_i] \stackrel{\text{laut Def.}}{=} 0 \cdot \mathbf{Pr}[M_i = 0] + 1 \cdot \mathbf{Pr}[M_i = 1] = \mathbf{Pr}[M_i = 1]$$

Beispiel: Zahlenfolge = (7, 2, 8, 5, 9, 1, 4, 6, 3)

$\mathbf{Pr}[M_i = 1] =$ WK, dass $i.$ Zahl die bisher größte ist

$$= \frac{\text{Anz. der „guten“ Ergebnisse}}{\text{Anz. aller Ergebnisse}}$$

Voraussetzung:
Alle Ergebnisse sind gleich wahrscheinlich!

$$= \frac{\text{Anz. Perm., bei denen } i. \text{ Zahl am größten}}{\text{Anz. aller Permutationen von } i \text{ Zahlen}} = \frac{(i-1)!}{i!} = \frac{1}{i}$$

Ein Trick

Definition: $\mathbf{E}[M] = \sum_{i \in \Omega'} i \cdot \mathbf{Pr}[M = i]$

Führe *Indikator-Zufallsvariable* ein (für $i = 1, \dots, n$):

Sei $M_i = \begin{cases} 1, & \text{falls Münchener nach dem } i. \text{ Essen zufrieden,} \\ 0 & \text{sonst.} \end{cases}$

$$\Rightarrow \mathbf{E}[M_i] \stackrel{\text{laut Def.}}{=} 0 \cdot \mathbf{Pr}[M_i = 0] + 1 \cdot \mathbf{Pr}[M_i = 1] = \mathbf{Pr}[M_i = 1]$$

Beispiel: Zahlenfolge = (7, 2, 8, 5, 9, 1, 4, 6, 3)

$\mathbf{Pr}[F_i = 1] =$ WK, dass $i.$ Zahl größer als $\frac{n+1}{2}$ ist

$$= \frac{\text{Anz. der „guten“ Ergebnisse}}{\text{Anz. aller Ergebnisse}}$$

$$= \frac{\text{Anz. Perm., bei denen } i. \text{ Zahl} > n/2}{\text{Anz. aller Permutationen von } n \text{ Zahlen}} = \frac{(n-1)! \cdot \frac{n}{2}}{n!} = \frac{1}{2}$$

Zurück zum Erwartungswert

$$M_i = \begin{cases} 1, & \text{falls Münchner nach dem } i. \text{ Essen zufrieden,} \\ 0 & \text{sonst.} \end{cases}$$

$\Rightarrow M = \sum_{i=1}^n M_i$ (Anz. Male, die Münchener zufrieden ist.)

$$\Rightarrow \mathbf{E}[M] = \mathbf{E}[\sum_i M_i] = \sum_i \mathbf{E}[M_i] = \sum_i \mathbf{Pr}[M_i = 1]$$

Linearität des Erwartungswerts *Indikatorvariable*

$$= \sum_{i=1}^n \frac{1}{i} \leq 1 + \int_1^n \frac{1}{x} dx = 1 + \ln n$$

Entsprechend...

$$\mathbf{E}[F] = \sum_{i=1}^n \mathbf{Pr}[F_i = 1] = \sum_{i=1}^n \frac{1}{2} = \frac{n}{2}$$

M.a.W.: man kann erwarten,
dass der Franke *exponentiell zufriedener* ist als der Münchner!
;-)

Average-Case-Laufzeit von InsertionSort

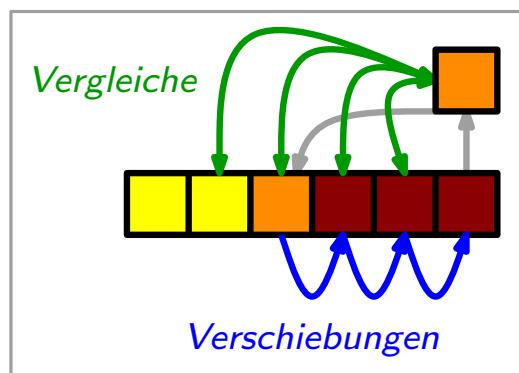
Beob. Der „durchschnittliche Fall“ ist i. A. schwer fassbar.

Hier: Was ist die durchschnittliche Laufzeit über alle Permutationen der Eingabe $A[1..n]$ (für festes n)?

Einfacher: Was ist die durchschnittliche Laufzeit über alle Permutationen der Eingabe $\langle 1, 2, \dots, n \rangle$?

Wissen: $n - 1 \leq V_{IS}(n) \leq n(n - 1)/2$

Beob. Statt Vergleiche können wir auch die Anzahl der Verschiebungen T_{IS} zählen, denn



wenn wir ein Element einfügen
(innere Schleife von InsertionSort), gilt:

$$\# \text{ Verschiebungen} \leq \# \text{ Vergleiche} \leq \# \text{ Verschiebungen} + 1$$

d.h. insg. gilt: $T_{IS} \leq V_{IS} \leq T_{IS} + (n - 1)$.

Average-Case-Laufzeit vs. erwartete Laufzeit

Beob. Statt der durchschnittlichen Laufzeit über alle Permutationen können wir auch die *erwartete* Laufzeit einer zufälligen Permutation betrachten.

Warum?

Betrachte Definition:

$$\mathbf{E}[T] = \sum_{i=0}^{n^2} i \cdot \mathbf{Pr}[T = i]$$

hier: # Verschiebungen
(d.h. Laufzeit)

Anteil der Permutationen,
die i Verschiebungen verursachen.

Erwartete Laufzeit von InsertionSort

$T :=$ Zufallsvariable für die Anzahl von Verschiebungen, die IS benötigt, um eine zufällige Permutation $A[1..n]$ von $\langle 1, 2, \dots, n \rangle$ zu sortieren

Was wäre eine gute Indikatorvariable um T auszudrücken?

$$T_{ij} := \begin{cases} 1, & \text{falls } A[i] > A[j] \\ 0 & \text{sonst.} \end{cases} \text{ für } 1 \leq i < j \leq n.$$

$$\Rightarrow T_j = \sum_{i=1}^{j-1} T_{ij} = \text{Anz. Pos., um die } A[j] \text{ n. li. verschoben wird.}$$

$$\Rightarrow T = \sum_{j=2}^n T_j = \sum \sum T_{ij} \quad \text{und} \quad \mathbf{E}[T] = \sum \sum \mathbf{E}[T_{ij}]$$

Aber was ist $\mathbf{E}[T_{ij}]$? Laut Def. $\mathbf{E}[T_{ij}] = \Pr[T_{ij} = 1] = \frac{1}{2}$

$$\Rightarrow \mathbf{E}[T] = \sum_{j=2}^n \sum_{i=1}^{j-1} \mathbf{E}[T_{ij}] = \sum_{j=2}^n \frac{j-1}{2} = \frac{1}{2} \sum_{j=1}^{n-1} j = \frac{n(n-1)}{4} \quad \square$$

Zusammenfassung InsertionSort

Satz. [alt]

Im *besten Fall* benötigt InsertionSort $n - 1 \in \Theta(n)$ Vergleiche und 0 Verschiebungen.

Im *schlechtesten Fall* benötigt InsertionSort $n(n - 1)/2 \in \Theta(n^2)$ Vergleiche/Verschiebungen.

Satz. [neu]

Im *Durchschnitt* benötigt InsertionSort $n(n - 1)/4 \in \Theta(n^2)$ Verschiebungen und zwischen $n(n - 1)/4$ und $n(n - 1)/4 + (n - 1)$, d.h. $\Theta(n^2)$, Vergleiche.

Kurz: Bei InsertionSort gilt

Average Case =_{asymptotisch} Worst Case!

Geburtstagswahrscheinlichkeiten

Frage: Wie groß ist die *Wahrscheinlichkeit*, dass mindestens zwei Leute hier im Hörsaal am gleichen Tag Geburtstag haben? [siehe Abschnitt 5.4, CLRS]

Frage': Wie groß ist der *Erwartungswert* für die Anzahl X von Pärchen hier im Hörsaal, die am gleichen Tag Geburtstag haben?

Was wäre eine gute Indikatorvariable um X auszudrücken?

$$X_{ij} := \begin{cases} 1 & \text{falls } i \text{ und } j \text{ gleichen Geburtstag haben} \\ 0 & \text{sonst.} \end{cases} \quad \left. \begin{array}{l} \text{für} \\ 1 \leq i < j \leq k; \\ k = \text{Anz. Leute} \end{array} \right\}$$

Dann gilt $X = \sum_{1 \leq i < j \leq k} X_{ij} = \sum_{i=1}^{k-1} \sum_{j=i+1}^k X_{ij}.$

Geburtstagserwartungen

Es gilt: $X_{ij} := \begin{cases} 1 & \text{falls } G_i = G_j \\ 0 & \text{sonst.} \end{cases}$ und $X = \sum_{1 \leq i < j \leq k} X_{ij}$.

Annahme: Alle n Tage sind gleich wahrscheinlich Geburtstage.

$$\Rightarrow \mathbf{E}[X_{ij}] = \Pr[X_{ij} = 1] = \sum_{t=1}^n \Pr[G_i = G_j = t] = \frac{n}{n^2} = 1/n.$$

Ereignisse schließen sich gegenseitig aus!

(Geht auch einfacher!)

Gesucht: $\mathbf{E}[X] = \mathbf{E} \left[\sum_{1 \leq i < j \leq k} X_{ij} \right] = \sum_{1 \leq i < j \leq k} \mathbf{E}[X_{ij}] =$

Linearität des Erwartungswerts!

$$= \binom{k}{2} \cdot \frac{1}{n} = \frac{k(k-1)}{2n} \geq 1 \Leftrightarrow k(k-1) \geq 2n.$$

Für ein Jahr mit $n = 365$ Tagen braucht man also nur $k \geq 28$ Personen um ein Pärchen mit gleichem Geburtstag erwarten zu können. \square

Algorithmen und Datenstrukturen

Wintersemester 2018/19
8. Vorlesung

Sortieren – mit dem Würfel!

Anmelden zum 1. Kurztest

- Anmeldung im WueCampus-Kurs unter „Teilnahme am 1. Kurztest am 22.11.2018“
- **Deadline: Di, 20.11., 13:00 Uhr**
- Wer sich nicht anmeldet, kann nicht mitschreiben :-(

Und noch einmal: Sortieren!

Zur Erinnerung: MergeSort...

- + gute Worst-Case-Laufzeit (durch Teile-und-Herrsche)
- kein in-situ-Verfahren (benötigt extra Felder beim Mergen)

Ziel: Teile-&-Herrsche-Verfahren, das trotzdem in situ sortiert!

Sortiere ein Teilstück $A[\ell..r]$ wie folgt: **QuickSort(int[] A, int ℓ, r)**

Teile:

*Partition(A, ℓ, r)
[liefert m zurück]*

Bestimme einen Index $m \in \{\ell, \dots, r\}$ und teile $A[\ell..r]$ so in $A[\ell..m - 1]$ und $A[m + 1..r]$ auf, dass alle Elemente im ersten Teilstück kleiner gleich $A[m]$ sind und alle im zweiten größer als $A[m]$.

Herrsche:

durch rekursives Sortieren der beiden Teilstücke.

Kombiniere: —

Schreiben Sie QuickSort in Pseudocode unter Verwendung von Partition(A, ℓ, r)!

QuickSort

`QuickSort($A, \ell = 1, r = A.length$)`

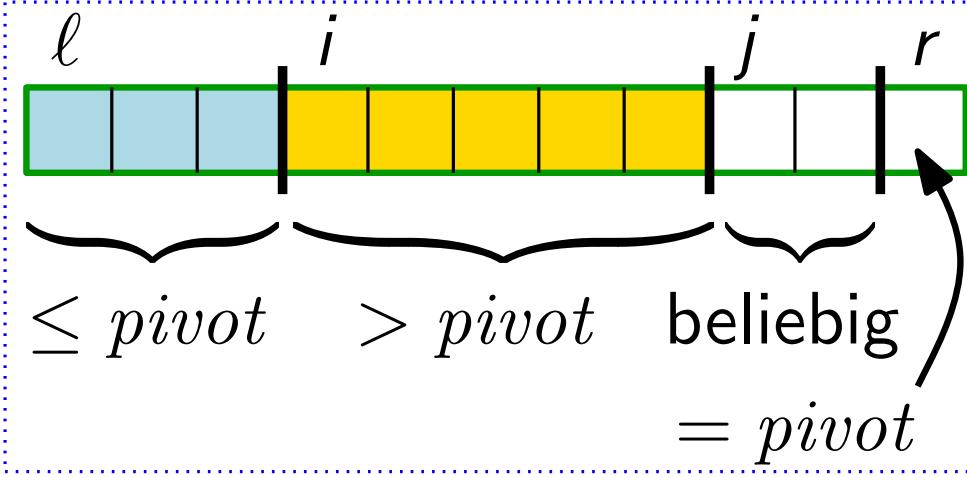
```

if  $\ell < r$  then
     $m = \text{Partition}(A, \ell, r)$ 
    QuickSort( $A, \ell, m - 1$ )
    QuickSort( $A, m + 1, r$ )

```

Schleifeninvariante:

- (i) Für $k = \ell, \dots, i - 1$ gilt $A[k] \leq pivot$.
- (ii) Für $k = i, \dots, j - 1$ gilt $A[k] > pivot$.
- (iii) $A[r] = pivot$.
- (iv) $A[\ell..j-1]$ enthält die gleichen Elemente wie zu Beginn.



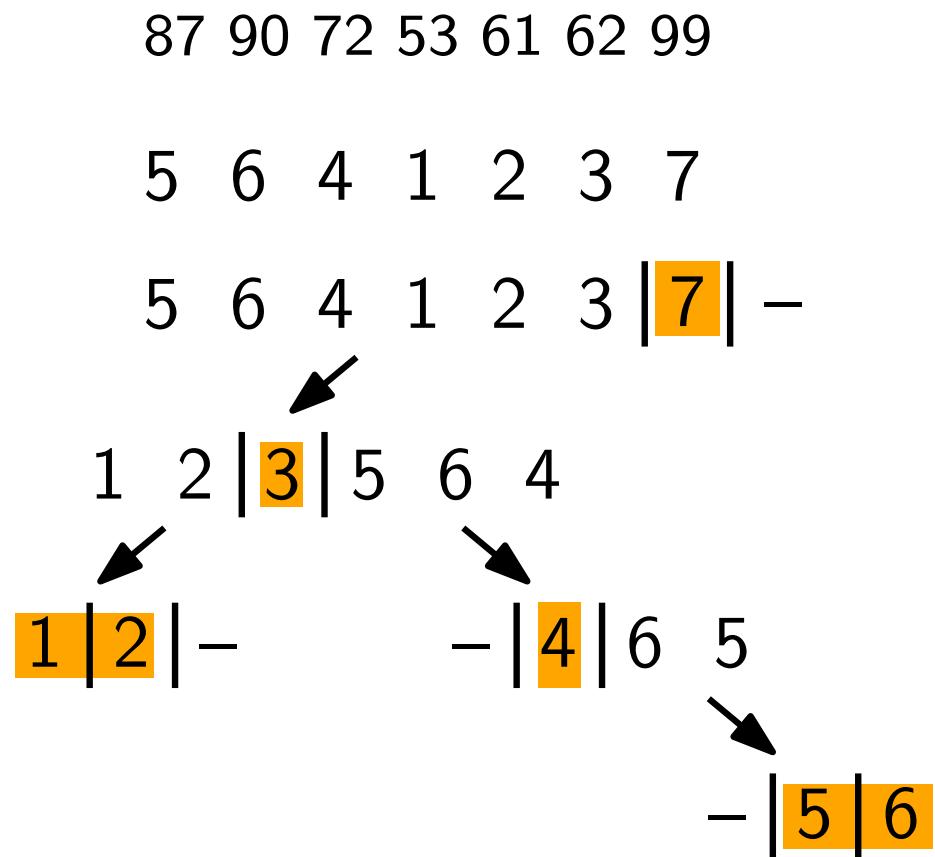
`int Partition(int[] A, int ℓ , int r)`

```

    pivot = A[r]
    i =  $\ell$ 
    for  $j = \ell$  to  $r - 1$  do
        if  $A[j] \leq pivot$  then
            Swap(A, i, j)
            i = i + 1
    Swap(A, i, r)
    return i

```

Ein Beispiel



QuickSort($A, \ell = 1, r = \dots$)

```

if  $\ell < r$  then
     $m = \text{Partition}(A, \ell, r)$ 
    QuickSort( $A, \ell, m - 1$ )
    QuickSort( $A, m + 1, r$ )
  
```

int Partition(A, ℓ, r)

```

pivot =  $A[r]$ 
 $i = \ell$ 
for  $j = \ell$  to  $r - 1$  do
    if  $A[j] \leq \text{pivot}$  then
        Swap( $A, i, j$ )
         $i = i + 1$ 

```

```

Swap( $A, i, r$ )
return  $i$ 

```

Laufzeit

Zähle Anzahl der Vergleiche!

Beob. Partition benötigt *immer* $r - \ell$ Vergleiche.

Wovon hängt dann die Laufzeit ab?

$$T_{QS}(n) = T_{QS}(m-1) + T_{QS}(n-m) + n-1$$

1. Extremfall: m immer erstes Element

$$\begin{aligned} T_{QS}(n) &= T_{QS}(0) + T_{QS}(n-1) + n-1 \\ &= (T_{QS}(n-2) + n-2) + n-1 \\ &\vdots \\ &= T_{QS}(1) + 1 + 2 + \cdots + n-2 + n-1 \\ &\in \Theta(n^2) \end{aligned}$$

2. Extremfall: m immer mittleres Element

$$T_{QS}(n) \approx 2T_{QS}(n/2) + n - 1 \in \Theta(n \log n)$$

siehe MergeSort

QuickSort($A, \ell = 1, r = \dots$)

```

if  $\ell < r$  then
     $m = \text{Partition}(A, \ell, r)$ 
    QuickSort( $A, \ell, m - 1$ )
    QuickSort( $A, m + 1, r$ )
  
```

int Partition(A, ℓ, r)

```

    pivot =  $A[r]$ 
    i =  $\ell$ 
    for  $j = \ell$  to  $r - 1$  do
        if  $A[j] \leq \text{pivot}$  then
            Swap( $A, i, j$ )
            i = i + 1
    
```

Swap(A, i, r)
return i

Wo ist die Wahrheit?

M.a.W. was passiert im Durchschnittsfall (*average case*)?

Vgl. InsertionSort:

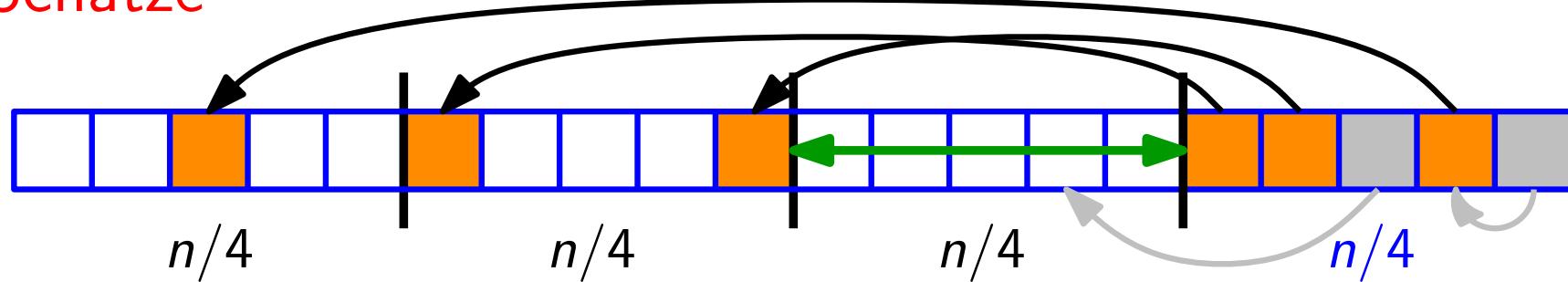
Bester Fall = $n - 1 \in \Theta(n)$	Vergleiche
Schlechterer Fall = $\binom{n}{2} \in \Theta(n^2)$	Vergleiche
Durchschnittsfall = $\Theta(n^2)$	

Mittle die Laufzeit über alle Permutationen der Eingabe!

Schwierig...

Statt dessen:

~~Berechne erwartete Laufzeit $E[T_{IS}]$ einer zufälligen Permutation~~
Schätze ab!



$$E[T_{IS}] \geq E[\text{Aufwand für letzte } \frac{n}{4} \text{ Elém.}] \geq \frac{n}{4} \cdot \frac{1}{2} \cdot \frac{n}{4} \in \Omega(n^2)$$

Zurück zu QuickSort

Idee: Steck Zufall in den Algorithmus!

Seien z_1, z_2, \dots, z_n die Elemente von A in sortierter Reihenfolge.

Wann vergleicht Alg. z_i und z_j ?

* höchstens ein Mal:
wenn eins von beiden *pivot* ist.

Definiere Indikator-Zufallsvariable:

$$V_{ij} = \begin{cases} 1, & \text{falls Alg. } z_i \text{ und } z_j \text{ vergleicht,} \\ 0 & \text{sonst.} \end{cases}$$

Sei V ZV für Gesamtanz. von Vgl.

Dann gilt $V = \sum_{1 \leq i < j \leq n} V_{ij}$.

$$\Rightarrow E[V] = \sum_{1 \leq i < j \leq n} E[V_{ij}]$$

RandomizedPartition(A, ℓ, r)

$k = \text{Random}(\ell, r)$ Liefert Zufallszahl
 $\in \{\ell, \dots, r\}$.

Swap(A, r, k)

return Partition(A, ℓ, r)

Partition(A, ℓ, r)

$pivot = A[r]$

$i = \ell$

for $j = \ell$ **to** $r - 1$ **do**

if $A[j] \leq pivot$ **then**

 Swap(A, i, j)

$i = i + 1$

Swap(A, i, r)

return i

First come, first serve

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \boxed{?}$$

Betrachte die Menge $Z_{ij} := \{z_i, z_{i+1}, \dots, z_j\}$.

Sei z^* die erste Zahl in Z_{ij} , die Pivot wird.

Es gilt: Alg. vergleicht z_i und $z_j \Leftrightarrow z^* = z_i$ oder $z^* = z_j$.

$$\begin{aligned} \Rightarrow \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] &= \Pr[z^* = z_i \text{ oder } z^* = z_j] \\ &\stackrel{i \neq j}{=} \Pr[z^* = z_i] + \Pr[z^* = z_j] \\ &= \frac{1}{|Z_{ij}|} + \frac{1}{|Z_{ij}|} \\ &= \frac{2}{j - i + 1} \end{aligned}$$

Auf zum letzten Gefecht . . .

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \frac{2}{j - i + 1}$$

Wir wissen:

$$E[V] = \sum_{1 \leq i < j \leq n} E[V_{ij}] = \sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1}$$

$$\begin{aligned} &= \sum_{i=1}^{n-1} \left(\sum_{j=i+1}^n \frac{2}{j - i + 1} \right) \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} \end{aligned}$$

Trick: ersetze $j - i$ durch k !

Auf zum letzten Gefecht . . .

$$E[V_{ij}] = \Pr[\text{Alg. vergleicht } z_i \text{ und } z_j] = \frac{2}{j - i + 1}$$

Wir wissen:

$$E[V] = \sum_{1 \leq i < j \leq n} E[V_{ij}] = \sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \quad \text{Trick: ersetze } j - i \text{ durch } k!$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \in O(n \log n)$$

Satz: RandomizedQuickSort sortiert n Zahlen in $O(n \log n)$ erwarteter Zeit.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	QuickSort
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Avg.-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
in situ ¹ (<i>in place</i>)	✓	✗	✓	(✓)*
stabil ²	✓	✓	✗	✗

¹) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

²) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

*) QuickSort muss für jeden rekursiven Aufruf die Variable m zwischenspeichern. Dafür wird im worst case $\Omega(n)$ zusätzlicher Speicherplatz benötigt. Mit Tricks kann man dieses Problem umgehen und so QuickSort in-situ machen.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
9. Vorlesung

Sortieren in Linearzeit

Reminder

Melden Sie sich unbedingt unter WueCampus an zur

Teilnahme am 1. Kurztest am 22.11.

... sonst können Sie nicht teilnehmen!

Anmeldefrist: Di, 20.11., 13:00 Uhr

... aber besser gleich *jetzt!*

Reminder II

Auslandsaufenthalte für Studierende der Informatik

Infoveranstaltung, heute (15.11.), SE 10 (ehem. TB Physik)

- 16:00 Partnerschaften und Fördermöglichkeiten der Uni Würzburg
(Susanne Holzheimer, International Students Office)
- 16:20 Das TASSEP-Programm
(Prof. Dr. Sebastian von Mammen)
- 16:30 Praktika und Abschlussarbeiten im Ausland
(Prof. Dr. Tobias Hoßfeld und Kathrin Borchert)
- 16:40 Partnerschaften in der Informatik: Konkrete Beispiele
(Prof. Dr. Alexander Wolff)
- 17:00 Einzelgespräche

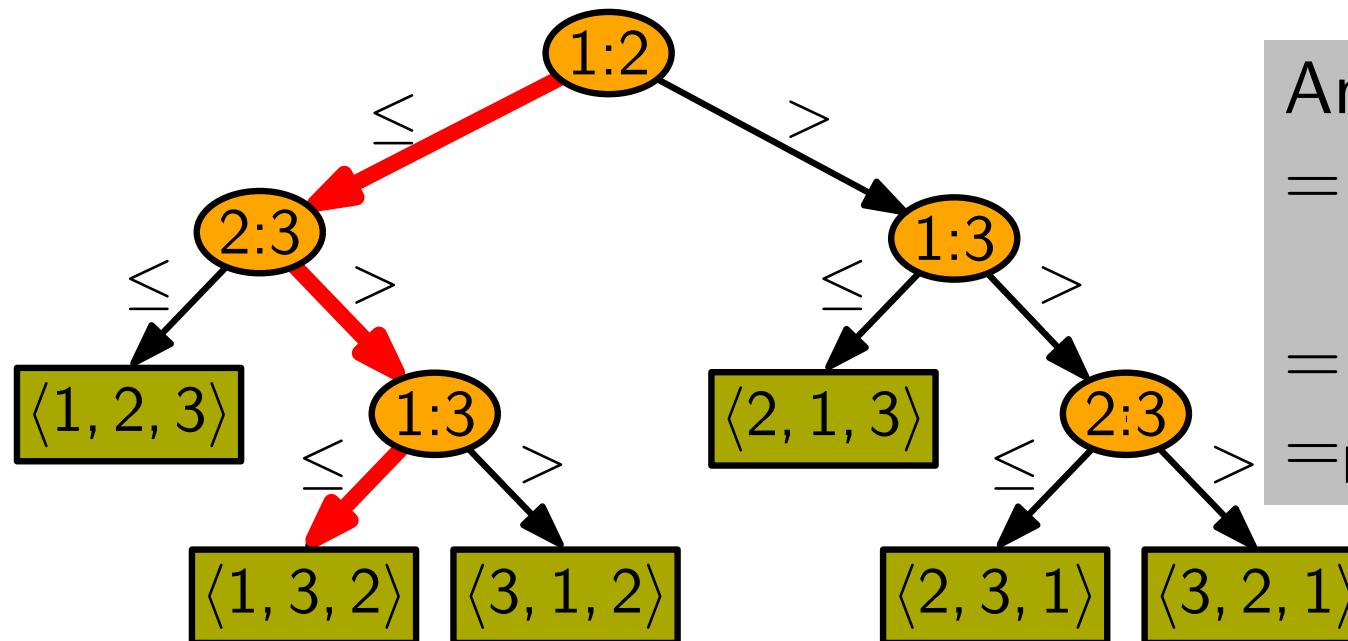
Sortieren durch Vergleichen

Eingabefolge $\langle a_1, a_2, \dots, a_n \rangle$ Sortieralg.
Schlüsselvergleiche Ausgabe: sortierte Eingabe

Für festes n ist ein vergleichsbasierter Sortieralg. charakterisiert durch seinen Entscheidungsbaum:



- innere Knoten = Vergleiche (o.B.d.A. immer \leq , z.B. „ $a_1 \leq a_2?$ “)
 - Blätter = sortierte Permutationen der Eingabe
 - Kanten = Ergebnisse ($\leq / >$) eines Vergleichs



Anz. Vgl. im *worst case*
= Länge eines *längsten*
Wurzel-Blatt-Pfads
=: Höhe des Baums
= hier 3

Entscheidungsbaum für InsertionSort und $n = 3$ [CLRS]

Eine untere Schranke

Frage: Wie viele Vergleiche braucht *jeder* vergleichsbasierte Sortieralg. im worst case um n verschiedene Objekte zu sortieren?

M.a.W. Gegeben

- ein beliebiger vergleichsbasierter Sortieralgorithmus,
- eine Zahl n von verschiedenen Objekten, die man sortieren soll, welche Höhe hat der Entscheidungsbaum *mindestens*?

Beob.: Die Höhe ist eine Funktion der Blätteranzahl.

Anz. Blätter = Anz. Permutationen von n Obj. = $n!$

Höhe Binärbaum mit B Blättern $\geq \lceil \log_2 B \rceil$

$$\text{Höhe Entscheidungsbaum} \geq \log_2 n! = \sum_{i=1}^n \log_2 i$$

$$\geq \int_1^n \log_2 x \, dx = \frac{1}{\ln 2} \int_1^n \ln x \, dx = \frac{1}{\ln 2} \int_1^n 1 \cdot \ln x \, dx$$

$$= \frac{1}{\ln 2} \left(x \cdot \ln x \Big|_1^n - \int_1^n x \cdot \frac{1}{x} \, dx \right) = \frac{(n \ln n - 0) - (n - 1)}{\ln 2}$$

$$\in \Omega(n \log n)$$



$$\int u' v = uv - \int u v'$$

Resultat

Satz. Jeder vergleichsbasierte Sortieralg. benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche um n Objekte zu sortieren.

Korollar. MergeSort und HeapSort sind asymptotisch worst-case optimal.

Wir durchbrechen die Schallmauer



- SpaghettiSort sortiert Spaghetti nach Länge ;-)
- CountingSort sortiert Zahlen in $\{0, \dots, k\}$
- RadixSort sortiert s -stellige b -adische Zahlen
- BucketSort sortiert gleichverteilte zufällige Zahlen



aus:
www.marions-kochbuch.de

CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

Variable:

A	Eingabefeld	C Rechenfeld
B	Ausgabefeld	k begrenzt das <i>Universum</i> : $\{0, \dots, k\}$

- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x



- 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$



CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

Variable:

A	Eingabefeld	C Rechenfeld
B	Ausgabefeld	k begrenzt das <i>Universum</i> : $\{0, \dots, k\}$

- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x



CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

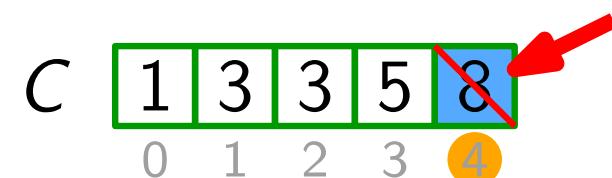
Variable:

A Eingabefeld B Ausgabefeld	C Rechenfeld k begrenzt das <i>Universum</i> : $\{0, \dots, k\}$
------------------------------------	---

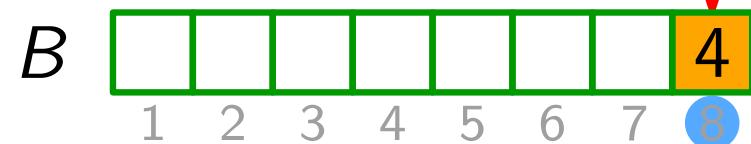
- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x



- 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$



- 2) Schreibe jedes x in A direkt an die richtige Position in B



CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

Variable:

A Eingabefeld	C Rechenfeld
B Ausgabefeld	k begrenzt das <i>Universum</i> : $\{0, \dots, k\}$

- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x



- 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$



- 2) Schreibe jedes x in A direkt an die richtige Position in B



CountingSort ist *stabil!*

CountingSort

- Plan:**
- 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x
 - 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$
 - 2) Schreibe jedes x in A direkt an die richtige Position in B

CountingSort(int[] A, int[] B, int k)

Eingabefeld
Ausgabefeld
beschränkt Universum $\{0, \dots, k\}$

sei $C[0..k] = \langle 0, 0, \dots, 0 \rangle$ ein neues Feld

for $j = 1$ **to** $A.length$ **do** [] // (1a)

// $C[i]$ enthält jetzt die Anz. der Elem. gleich i in A

for $i = 1$ **to** k **do** [] // (1b)

// $C[i]$ enthält jetzt die Anz. der Elem. $\leq i$ in A

for $j = A.length$ **downto** 1 **do**

[] // (2)

Aufgabe:

Fülle die Felder mit Code,
der obige Idee umsetzt!

CountingSort

Laufzeit:
 $O(n + k)$

- Plan:**
- 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x
 - 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$
 - 2) Schreibe jedes x in A direkt an die richtige Position in B

CountingSort(int[] A, int[] B, int k)

Eingabefeld
Ausgabefeld
beschränkt Universum $\{0, \dots, k\}$

sei $C[0..k] = \langle 0, 0, \dots, 0 \rangle$ ein neues Feld

for $j = 1$ **to** $A.length$ **do** $C[A[j]] = C[A[j]] + 1$ // (1a)
// $C[i]$ enthält jetzt die Anz. der Elem. gleich i in A

for $i = 1$ **to** k **do** $C[i] = C[i] + C[i - 1]$ // (1b)
// $C[i]$ enthält jetzt die Anz. der Elem. $\leq i$ in A

for $j = A.length$ **downto** 1 **do**

$B[C[A[j]]] = A[j]$ // (2)

$C[A[j]] = C[A[j]] - 1$

RadixSort

(Jahr, Monat, Tag)

Frage: Gegeben Liste von Menschen mit deren Geburtstagen.
Wie würden Sie die Liste nach Alter sortieren?

Drei (?) Lösungen:

- Geburtstage in Anz. Tage seit 1.1.1970 umrechnen, dann vergleichsbasiertes Sortierverfahren verwenden.
- Spezielle Vergleichsroutine schreiben und in vergleichsbasiertes Sortierverfahren einbauen.
- Liste $3 \times$ sortieren: je $1 \times$ nach Jahr, Monat, Tag.

Aber in welcher Reihenfolge??

RadixSort(A, s)

Anz. Stellen (hier: 3)

Laufzeit?

for $i = 1$ to s **do** [1 = Index der *niederwertigsten* (!) Stelle]
 ↳ sortiere A *stabil* nach der i -ten Stelle

z.B. mit CountingSort

Beispiel

Sortiere $A = \langle 25, 13, 31, 23, 11, 37, 15 \rangle$:

Gemäß RadixSort erst nach Einern, dann (stabil) nach Zehnern.

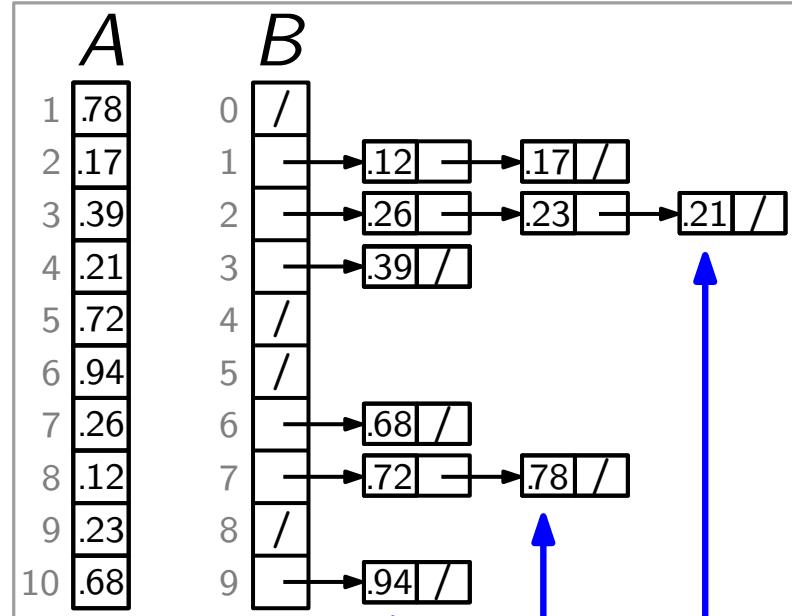
RadixSort(A, s)

for $i = 1$ **to** s **do**

 └ sortiere A *stabil* nach der i -ten Stelle

BucketSort

[CLRS]



(c) www.seafish.org

„Eimerinhalt“: Verkettete Liste von Elementen aus *A*.

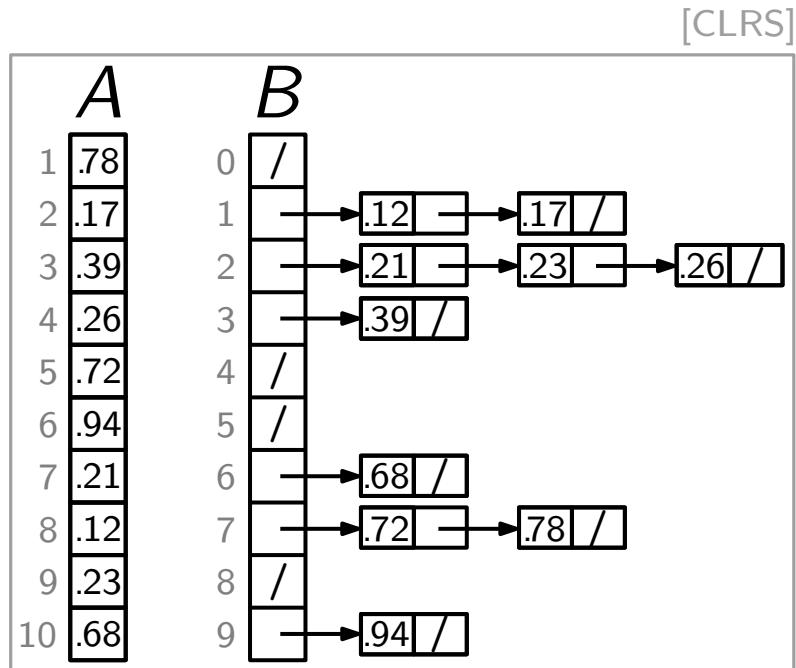
Hilfsfeld *B*[0..*n* – 1];
jeder Eintrag entspricht einem „Eimer“ der Weite $1/n$

Eingabefeld *A*[1..*n*] enthält Zahlen,
zufällig und gleichverteilt aus [0, 1) gezogen

[Im Bsp. auf 2 Nach-
kommastellen gerundet!]

BucketSort

BucketSort(Feld A von Zahlen in $[0, 1)$)



$$n = A.length$$

lege Feld $B[0..n - 1]$ von Listen an

for $j = 1$ **to** n **do**

 └ füge $A[j]$ in Liste $B[\lfloor n \cdot A[j] \rfloor]$ ein

for $i = 0$ **to** $n - 1$ **do**

 └ sortiere Liste $B[i] = [\frac{i}{n}, \frac{i+1}{n}) \cap A$

hänge $B[0], \dots, B[n - 1]$ aneinander

kopiere das Ergebnis nach $A[1..n]$

Korrektheit?

2 Fälle:

- $A[i]$ und $A[j]$ in der gleichen Liste

- $A[i]$ und $A[j]$ in verschiedenen Listen

Laufzeit?

- *erwartet*, hängt von den zufälligen Zahlen in A ab
- hängt vom Sortieralgorithmus in Zeile 6 ab;
wir nehmen InsertionSort: schnell auf kurzen Listen!

Erwartete Laufzeit von BucketSort

$$T_{\text{BS}}(n) = \Theta(n) + \sum_{i=0}^{n-1} T_{\text{IS}}(n_i) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

$$\begin{aligned} E[T_{\text{BS}}(n)] &= E[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) = \Theta(n) \end{aligned}$$

Behauptung: $E[n_i^2] \leq 2 - \frac{1}{n}$

fest!

Beweis. Def. Indikator-ZV $X_j := 1$, falls $A[j]$ in Eimer i fällt.

$$\Rightarrow n_i = \sum_{j=1}^n X_j \quad E[X_j] = \Pr[X_j = 1] = 1/n$$

$$\begin{aligned} \Rightarrow n_i^2 &= \left(\sum_{j=1}^n X_j\right)^2 = \sum_{j=1}^n \sum_{k=1}^n X_j X_k \\ &= \sum_{j=1}^n X_j^2 + \sum_{j=1}^n \sum_{k \neq j} X_j X_k \end{aligned}$$

Erwartete Laufzeit von BucketSort

Es gilt $n_i^2 = \sum_{j=1}^n X_j^2 + \sum_{j=1}^n \sum_{k \neq j} X_j X_k$

Behauptung:
 $E[n_i^2] \leq 2 - \frac{1}{n}$

$$\Rightarrow E[n_i^2] = \sum_{j=1}^n E[X_j^2] + \sum_{j=1}^n \sum_{k \neq j} E[X_j X_k]$$

Behandle die beiden Typen von Erwartungswerten getrennt:

$$\begin{aligned} E[X_j^2] &= 1 \cdot \Pr[X_j^2 = 1] + 0 \cdot \Pr[X_j^2 = 0] && \text{unabhängig von } j! \\ &= 1 \cdot \Pr[X_j = 1] + 0 \cdot \Pr[X_j = 0] = 1 \cdot \frac{1}{n} + 0 = \frac{1}{n} \end{aligned}$$

$$\begin{aligned} E[X_j X_k] &= E[X_j] \cdot E[X_k] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} && \text{unabh. von } j \text{ und } k! \\ &&& \text{für } j \neq k \text{ sind } X_j \text{ und } X_k \text{ unabhängig} \end{aligned}$$

Fasse die Zwischenergebnisse zusammen:

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^n E[X_j^2] + \sum_{j=1}^n \sum_{k \neq j} E[X_j X_k] \\ &= n \cdot \frac{1}{n} + n \cdot (n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n} \end{aligned}$$

□

Zusammenfassung

- Jedes *vergleichsbasierte* Sortierverfahren braucht im schlechtesten Fall $\Omega(n \log n)$ Vergleiche für n Zahlen.
- **CountingSort** sortiert Zahlen in $\{0, \dots, k\}$ (*stabil!*)
Laufzeit für n Zahlen: $O(n + k)$
- **RadixSort** sortiert s -stellige b -adische Zahlen
Laufzeit für n Zahlen: $O(s \cdot (n + b))$
- **BucketSort** sortiert gleichverteilte zufällige Zahlen
erwartete Laufzeit für n Zahlen: $O(n)$

Bem. Die Idee mit den (gleichgroßen) Eimern ist natürlich nicht nur auf Zufallszahlen beschränkt, aber hier lässt sie sich hübsch analysieren.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
10. Vorlesung

Das Auswahlproblem

„Kleine Vorlesungsevaluierung“: Ergebnisse

Was läuft gut?

Angemessenes Tempo	11
Gute Erklärungen	79
Es wird auf Fragen eingegangen	32
Gute Folien	61
Gute Vorlesungsstruktur	20
Viele anschauliche Beispiele	16
Studierende werden miteinbezogen	24
Kompetenter Dozent	1
Gute Buchempfehlung	1
Zwischentests	2
Gute Übungsaufgaben	4
Engagierter Dozent	7
Erklärungen an der Tafel	3
Umfangreiche Informationen	1
Nutzung des Mikrofons	21
Gute Lernatmosphäre in der VL	5
Donnerstags ab 8:30 Uhr	2
IPE-Grafiken	1
Dozent spricht frei	1

„Kleine Vorlesungsevaluierung“: Ergebnisse

Was sollte verbessert werden?

Zu langsam	2
Tempo zu schnell	23
Viel zu schneller Einstieg	1
Anfangs zu schnell, mittlerweile ok	1
Teilweise sehr abstrakter Inhalt	4
Zu viele Vorkenntnisse nötig	20
Übungen zu schwer, da Vorlesungsstoff zum Teil zur Bearbeitung unzureichend	9
Zu viel Stoff pro Vorlesungseinheit/zu schnelle Abarbeitung einzelner Themen	6
Folien werden bei vielen Änderungen unübersichtlich. Folien ab & zu „aufräumen“	9
Eine kurze Pause in der Mitte der Vorlesung	2
Antworten von Studierenden nochmal kurz erläutern	1
Vorkurs für ADS	1
Kurze Wh. der letzten Vorlesung am Anfang oder wenn Stoff aufgegriffen wird	5
Mehr Erklärungen in den Folien um diese daheim nachzuarbeiten	5
Pseudocode anfangs genauer erklären	1
Mehr Beispiele	4
Eindeutige Definitionen der Grundlagen	2
Keine Zeit für Diskussionen	1
Praktische Anwendungen des Stoffs erläutern	4
Weniger Sortieralgorithmen	1
Mehr PABS-Aufgaben	1

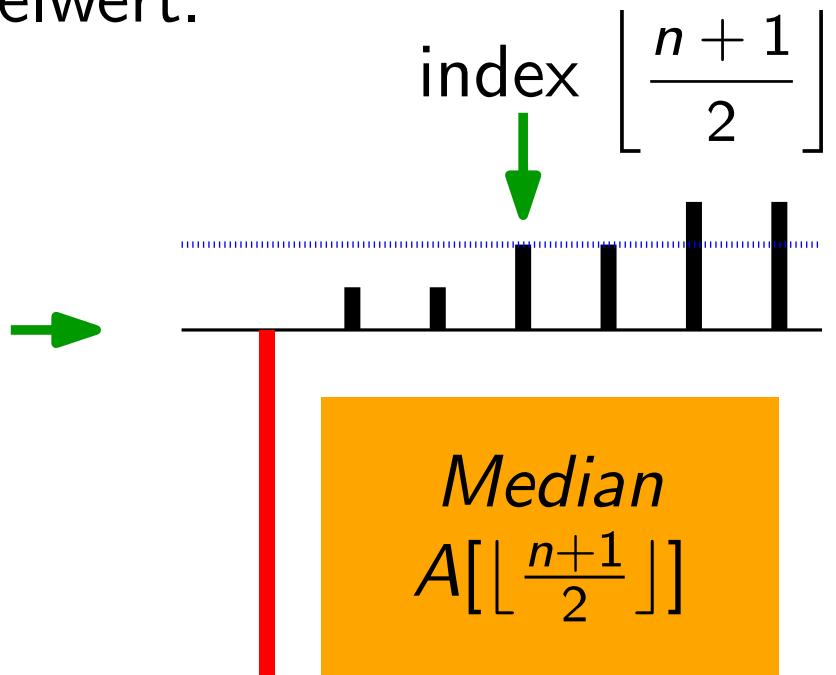
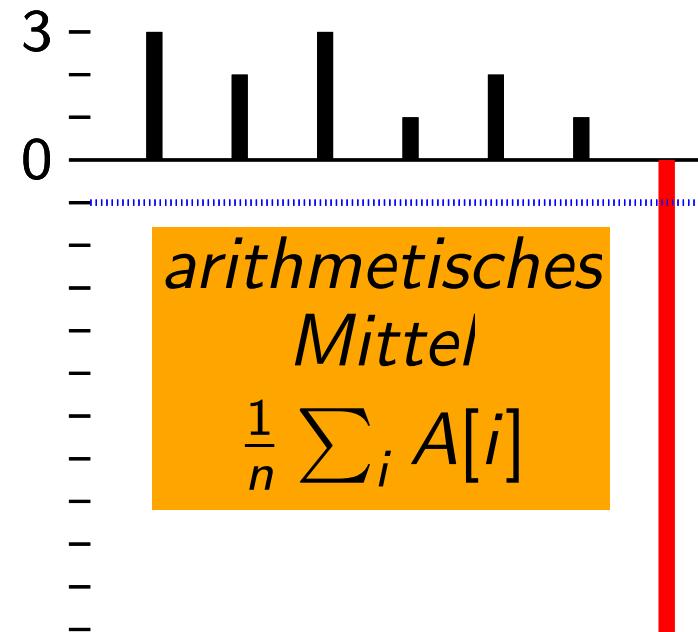
„Kleine Vorlesungsevaluierung“: Ergebnisse

Übungsaufgaben schwer verständlich.	3
Kritik an fehlender Mitarbeit weglassen.	1
Es fehlt richtiges Skript mit zusätzlichen Beispielen & ausführlichen Erklärungen.	7
Mehr Zeit für Denkaufgaben.	4
Es wird nicht lang genug auf Stoff eingegangen (z.B. O-Notation).	1
Undeutlich wann die Regularitätsbedingung der Meistermethode gebraucht wird.	3
Mehr Zeit für Grundlagen	1
Anfangs evtl mal Java-Code statt Pseudocode präsentieren	1
Begriffe werden teilweise nicht erklärt	1
Die Vorlesung ist scheiße, weil der Dozent Arrays mit Index 1 anfängt.	9
$A[l - r]$ statt $A[l..r]$	1
Skript morgens online stellen, so dass man es für Notizen ausdrucken kann.	4
Übungslösungen evtl. online fehlt oder zumindest beim Übungsleiter	1
Übungsleiter ist etwas penibel	1
Besseres Bestimmen der Wahrscheinlichkeit der Indikator-Zufallsvariablen	1
Deutsche Version von Carmen online öffentlich machen	1
Repititorium nicht nur vor Nachklausur sondern (auch?) vor Erstklausur.	1
Skripte mit Buch verbinden.	1
8 Uhr ist zu früh	1
Mehr Tafeleinsatz	1
Algorithmen ein bisschen näher erklären: z.B. was ist $A[i]$? z.B. an einem Bild auch bei HeapSort (wie 8. Vorlesung).	1

Analyse von Messreihen

Problem: Gegeben eine Reihe von n Messwerten $A[1..n]$, finde einen „guten“ Mittelwert.

Beispiel:



Beob.:

Der Median ist stabiler gegen Ausreißer als das arithmetische Mittel.

Das Auswahlproblem

Aufgabe: Gegeben ein Feld $A[1..n]$,
finde das i -kleinste Element von A .

Lösung: Sortiere und gib $A[i]$ zurück!

Worst-Case-Laufzeit: $\Theta(n \log n)$

[wenn man nichts über die
Verteilung der Zahlen weiß]

Geht das besser?

Spezialfälle

$i = \lfloor \frac{n+1}{2} \rfloor$: Median



Geht das auch in linearer Zeit??

$i = 1$: Minimum

$i = n$: Maximum } Laufzeit $\Theta(n)$

} Geht beides zusammen mit weniger als $2(n - 1)$ Vergleichen?

Minimum(int[] A)

$min = A[1]$

for $i = 2$ **to** $A.length$ **do**

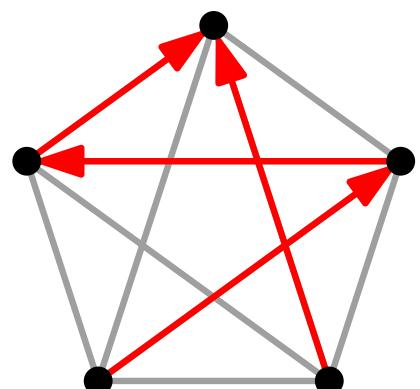
 └ **if** $min > A[i]$ **then** $min = A[i]$

return min

Anzahl Vergleiche = $n - 1$

Ist das *optimal*?

Betrachte ein K.O.-Turnier.



Bis ein Gewinner feststeht, muss jeder – außer dem Gewinner – mindestens einmal verlieren.

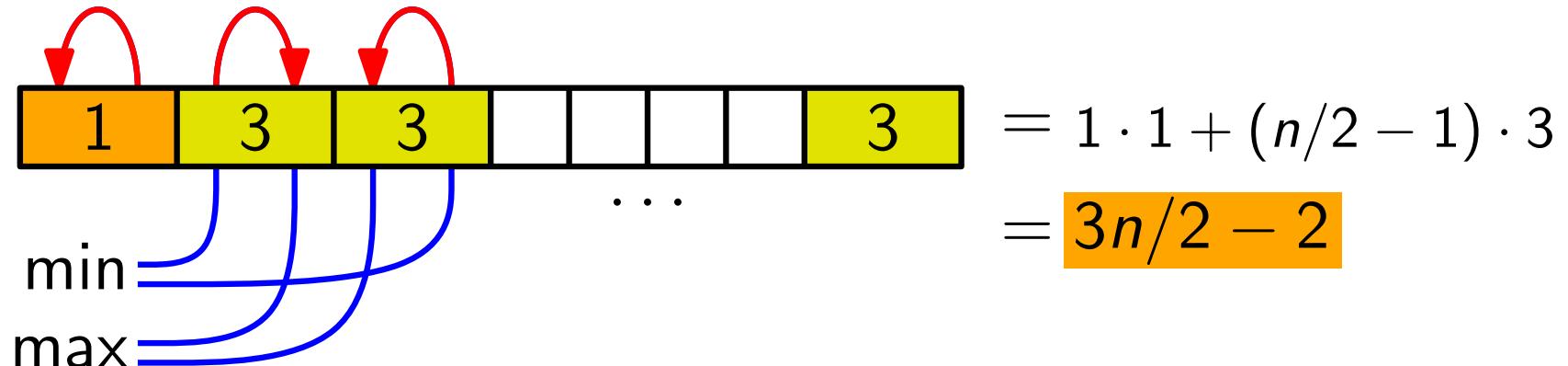
Also sind $n - 1$ Vergleiche optimal.

Eine Randbemerkung...

Def. Sei $V_{\min\max}(n)$ die Anz. der Vgl., die man braucht um Minimum *und* Maximum von n Zahlen zu bestimmen.

Klar: $V_{\text{minmax}}(n) \leq 2 \cdot V_{\text{min}}(n) = 2(n - 1)$

Frage: Geht es auch mit weniger Vergleichen? (n gerade)



Ist das *optimal*?

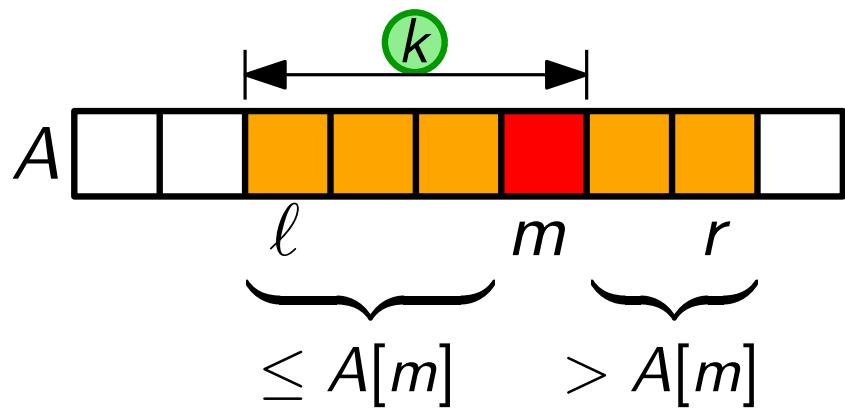
Auswahl per Teile & Herrsche

Zur Erinnerung...

Randomized
QuickSort(int[] A, int ℓ, r)

```

if  $\ell < r$  then
     $m = \text{Partition}(A, \ell, r)$ 
    QuickSort( $A, \ell, m - 1$ )
    QuickSort( $A, m + 1, r$ )
  
```



Finde i -.kleinstes Element in $A[\ell..r]$!

RandomizedSelect(int[] A, int ℓ, r, i)

```

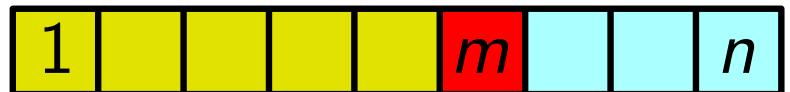
if  $\ell == r$  then return  $A[\ell]$ 
 $m = \text{RandomizedPartition}(A, \ell, r)$ 
 $k = m - \ell + 1$  //  $A[m]$  ist  $k$ -kleinstes El.  
von  $A[\ell..r]$ 
if  $i == k$  then
    return  $A[m]$ 
else
    if  $i < k$  then
        return RSelect( $A, \ell, m-1, i$ )
    else
        return RSelect( $A, m+1, r, i-k$ )
  
```

Ist Ihnen klar warum?

Laufzeit-Analyse

Anz. Vergl. von RandomizedSelect ist ZV; hängt von n und i ab.

Trick: Geh davon aus, dass das gesuchte i . Element immer im *größeren* Teilstück liegt.



\Rightarrow resultierende Zufallsvariable $V(n)$ ist

- obere Schranke für tatsächliche Anzahl von Vergleichen
- unabhängig von i

$$V(n) = \underbrace{V_{\text{Part}}(n)}_{= n-1} + \left\{ \begin{array}{ll} V(n-1) & \text{falls } m = 1 \\ V(n-2) & \text{falls } m = 2 \\ \dots & \\ V(\lfloor \frac{n}{2} \rfloor) & \text{falls } m = \lfloor \frac{n}{2} \rfloor + 1 \\ \dots & \\ V(n-2) & \text{falls } m = n-1 \\ V(n-1) & \text{falls } m = n \end{array} \right\}$$

Alle Fälle gleich
wahrscheinlich!
vorausgesetzt
alle Elem. sind
verschieden!

$$\Rightarrow E[V(n)] \leq n-1 + 2 \cdot \frac{1}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[V(k)] \leq c \cdot n \quad (\text{für ein } c > 0)$$

Substitutionsmethode

Wir schreiben $f(n)$ für $E[V(n)]$.

Dann gilt $f(n) \leq n + \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} f(k)$

Wir wollen prüfen, ob es ein $c > 0$ gibt, so dass $f(n) \leq cn$.

Also: $f(n) \leq n + \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} c \cdot k$ [laut Annahme]

Aufgabe:

Bestimmen Sie ein c ,
so dass $f(n) \leq cn!$
(Ignorieren Sie das
Abrunden $\lfloor \dots \rfloor$.)

Bem.: Wir sind *nicht* an $\sum_{k=1}^{n/2} f(k)$
interessiert – siehe letzte Folie.
Die Indizes sind wichtig!

$$E[V(n)] \leq n-1 + 2 \cdot \frac{1}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[V(k)]$$

Substitutionsmethode

Wir schreiben $f(n)$ für $E[V(n)]$.

Dann gilt $f(n) \leq n + \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} f(k)$

Wir wollen prüfen, ob es ein $c > 0$ gibt, so dass $f(n) \leq cn$.

$$\begin{aligned}
 \text{Also: } f(n) &\leq n + \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} c \cdot k \quad [\text{laut Annahme}] \\
 &= n + \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) \\
 &= n + \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{\lfloor n/2 \rfloor (\lfloor n/2 \rfloor - 1)}{2} \right) \\
 &\leq n + \frac{c}{n} (n(n-1) - (n/2 - 1)(n/2 - 2)) \\
 &\leq n + c \cdot \frac{3n+2}{4} = cn - \left(c \cdot \frac{n-2}{4} - n \right) \stackrel{?}{\geq} 0 \\
 &\leq cn \quad \text{falls } c \geq \frac{4n}{n-2} = \frac{4}{1-2/n} \xrightarrow[n \rightarrow \infty]{} 4^+
 \end{aligned}$$

Für jedes $\varepsilon > 0$ gilt:

$$E[V(n)] \leq n-1 + 2 \cdot \frac{1}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[V(k)] \leq \overbrace{(4+\varepsilon)n}^{c :=} \begin{cases} \text{falls } \\ n \geq \frac{8}{\varepsilon} + 2 \end{cases}$$

Ergebnis und Diskussion

Satz. Das Auswahlproblem kann in erwartet linearer Zeit gelöst werden.

Genauer: Für jedes $\varepsilon > 0$ gilt, dass man in einer Folge von $n \geq \frac{8}{\varepsilon} + 2$ Zahlen die i -kleinste Zahl ($1 \leq i \leq n$) mit **erwartet** $(4 + \varepsilon)n$ Vergleichen finden kann.

Frage: Geht das auch *deterministisch*, d.h. ohne Zufall?

M.a.W.: Kann man das Auswahlproblem auch im *schlechtesten Fall* in linearer Zeit lösen?

Vorbereitung

Wir verwenden wieder Teile-und-Herrsche – aber diesmal mit einer **garantiert guten** Aufteilung in Teilfelder.

d.h. *balanciert*:

jede Seite sollte $\geq \gamma n$ Elem. enthalten, für ein festes $0 < \gamma \leq \frac{1}{2}$.

Partition'(A, ℓ , r , pivot)

~~$pivot = A[r]$~~

$i = \ell - 1$

for $j = \ell$ **to** $r > 1$ **do**

if $A[j] \leq pivot$ **then**

$i = i + 1$

 Swap(A, i , j)

 Swap(A, $i + 1$, r)

return $i + 1$

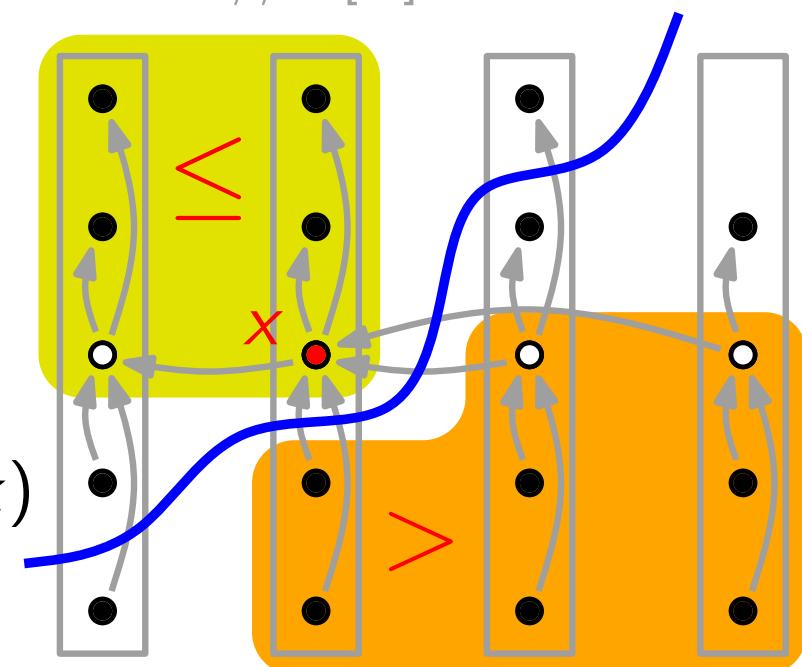
Wir gehen für die Analyse wieder davon aus, dass alle Elemente verschieden sind.

Select: deterministisch

Select(A, ℓ, r, i)

1. Teile die n Elem. der Eingabe in $\lfloor n/5 \rfloor$ 5er-Gruppen und eine Gruppe mit den restlichen ($n \bmod 5$) Elem.
2. Sortiere jede der $\lceil n/5 \rceil$ Gruppen und bestimme ihren Median.
3. Bestimme *rekursiv* den Median x der Gruppen-Mediane.
4. $m = \text{Partition}'(A, \ell, r, x); k = m - \ell + 1$ // $A[m]$ k -kleinstes El.
5. if $i == k$ then return $\underbrace{A[m]}_x$
 else
 - if $i < k$ then
 | return Select($A, \ell, m - 1, i$)
 - else
 | return Select($A, m + 1, r, i - k$)

$$\text{Anzahl}(\bullet) \geq 3 \left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right) \geq \frac{3n}{10} - 6$$

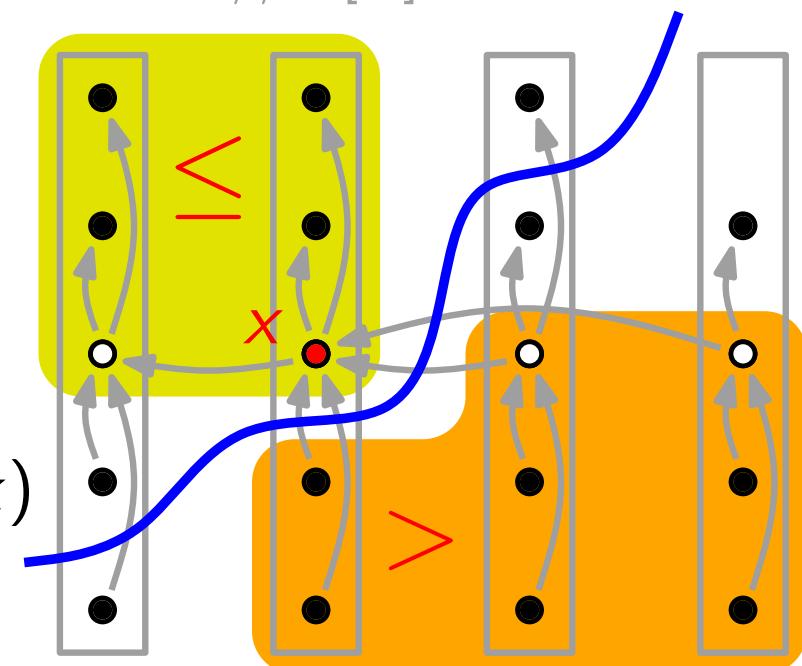


Select: deterministisch

Select(A, ℓ, r, i)

1. Teile die n Elem. der Eingabe in $\lfloor n/5 \rfloor$ 5er-Gruppen und eine Gruppe mit den restlichen ($n \bmod 5$) Elem.
2. Sortiere jede der $\lceil n/5 \rceil$ Gruppen und bestimme ihren Median.
3. Bestimme *rekursiv* den Median x der Gruppen-Mediane.
4. $m = \text{Partition}'(A, \ell, r, x); k = m - \ell + 1$ // $A[m]$ k -kleinstes El.
5. if $i == k$ then return $\underbrace{A[m]}_x$
 else
 - if $i < k$ then
 | return Select($A, \ell, \underbrace{m-1}_{\leq 7n/10 + 6 \text{ Elem.}}, i$)
 - else
 | return Select($A, \overbrace{m+1, r}^{>}, i - k$)

$$\text{Anzahl}(\bullet) \geq 3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$$



Laufzeit-Analyse

Beob. Es genügt wieder, Vergleiche zu zählen!

Partition': $\approx 1n$, Sortieren: $\approx \frac{n}{5} \cdot V_{IS}(5) = 2n$ Vgl.

Ansatz:

$$V(n) \leq \begin{cases} \overbrace{V(\lceil n/5 \rceil)}^{\text{Schritt 3}} + \underbrace{V(7n/10 + 6)}_{\text{Schritt 5}} + 3n & \text{falls } n \geq n_0, \\ O(1) & \text{sonst.} \end{cases}$$

Laufzeit-Analyse

Beob. Es genügt wieder, Vergleiche zu zählen!

Partition': $\approx 1n$, Sortieren: $\approx \frac{n}{5} \cdot V_{IS}(5) = 2n$ Vgl.

Ansatz:

$$V(n) \leq \begin{cases} V(\lceil n/5 \rceil) + V(7n/10 + 6) + 3n & \text{falls } n \geq n_0, \\ O(1) & \text{sonst.} \end{cases}$$

Behauptung:

Es gibt $c, n_0 > 0$, so dass für alle $n \geq n_0$ gilt: $V(n) \leq cn$.

$$\begin{aligned} \Rightarrow V(n) &\leq c \cdot (n/5 + 1) + c \cdot (7n/10 + 6) + 3n && \stackrel{?}{\geq} 0 \\ &= c \cdot (9n/10 + 7) + 3n = cn - (c \cdot (n/10 - 7) - 3n) \end{aligned}$$

falls $c \geq \frac{3n}{n/10-7} = \frac{30}{1-70/n} \xrightarrow{n \rightarrow \infty}$

Laufzeit-Analyse

Beob. Es genügt wieder, Vergleiche zu zählen!

Partition': $\approx 1n$, Sortieren: $\approx \frac{n}{5} \cdot V_{IS}(5) = 2n$ Vgl.

Ansatz:

$$V(n) \leq \begin{cases} V(\lceil n/5 \rceil) + V(7n/10 + 6) + 3n & \text{falls } n \geq n_0, \\ O(1) & \text{sonst.} \end{cases}$$

Behauptung:

Es gibt $c, n_0 > 0$, so dass für alle $n \geq n_0$ gilt: $V(n) \leq cn$.

$$\begin{aligned} \Rightarrow V(n) &\leq c \cdot (n/5 + 1) + c \cdot (7n/10 + 6) + 3n \\ &= c \cdot (9n/10 + 7) + 3n = cn - (c \cdot (n/10 - 7) - 3n) \end{aligned}$$

$$\text{falls } c \geq \frac{3n}{n/10-7} = \frac{30}{1-70/n} \xrightarrow{n \rightarrow \infty} 30^+ \quad \text{bzw. } n \geq \frac{70c}{c-30}.$$

$$\Rightarrow \text{für jedes } \varepsilon > 0 \text{ und } n \geq \frac{2100}{\varepsilon} + 70 \text{ gilt: } V(n) \leq \underbrace{(30 + \varepsilon)}_c \cdot n$$

Laufzeit-Analyse

Kann man das verbessern?

Beob. Es genügt wieder, Vergleiche zu zählen!

Partition': $\approx 1n$, Sortieren: $\approx \frac{n}{5} \cdot V_{IS}(5) = 2n$ Vgl.

Ansatz:

$$V(n) \leq \begin{cases} V(\lceil n/5 \rceil) + V(7n/10 + 6) + 3n & \text{falls } n \geq n_0, \\ O(1) & \text{sonst.} \end{cases}$$

Behauptung:

Es gibt $c, n_0 > 0$, so dass für alle $n \geq n_0$ gilt: $V(n) \leq cn$.

$$\Rightarrow V(n) \leq c \cdot (n/5 + 1) + c \cdot (7n/10 + 6) + 3n$$

$$= c \cdot (9n/10 + 7) + 3n = cn - (c \cdot (n/10 - 7) - 3n)$$

$$\text{falls } c \geq \frac{3n}{n/10-7} = \frac{30}{1-70/n} \xrightarrow{n \rightarrow \infty} 30^+ \quad \text{bzw. } n \geq \frac{70c}{c-30}.$$

$$\Rightarrow \text{für jedes } \varepsilon > 0 \text{ und } n \geq \frac{2100}{\varepsilon} + 70 \text{ gilt: } V(n) \leq (30 + \varepsilon) \cdot n$$

Hausaufgabe!

Ergebnis und Diskussion

Satz: Das Auswahlproblem kann auch im schlechtesten Fall in linearer Zeit gelöst werden.

Genauer: Für jedes $\varepsilon > 0$ gilt, dass man in einer Folge von $n \geq 2100/\varepsilon + 70$ Zahlen die i -kleinste Zahl mit höchstens $(30 + \varepsilon)n$ Vergleichen finden kann.

Literatur: *Randomized Algorithms* [Motwani+Raghavan, Cambridge U Press, '95]
Algorithmen und Zufall [Vorlesungsskript, Jochen Geiger, Uni KL]

- Der Algorithmus LazySelect [Floyd & Rivest, 1975] löst das Auswahlproblem mit WK $1 - O(1/\sqrt[4]{n})$ mit $\frac{3}{2}n + o(n)$ Vgl.
- Die besten deterministischen Auswahl-Algorithmen (*sehr kompliziert!*) benötigen $3n$ Vergleiche im schlechtesten Fall.
- *Jeder* deterministische Auswahl-Alg. benötigt im schlechtesten Fall mindestens $2n$ Vergleiche.



Algorithmen und Datenstrukturen

Wintersemester 2018/19
11. Vorlesung

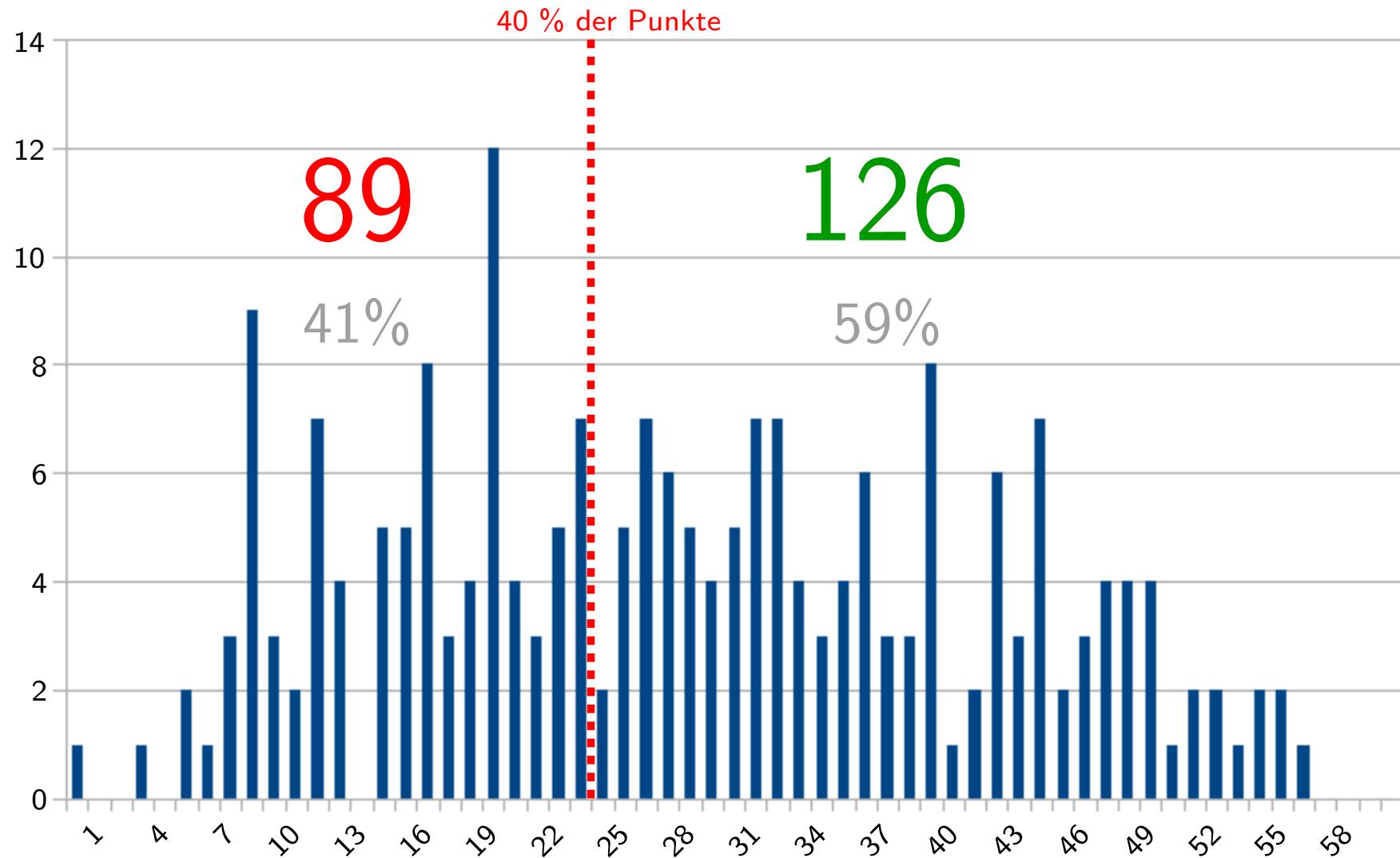
Elementare Datenstrukturen:
Stapel + Schlange + Liste

Johannes Zink

~~Prof. Dr. Alexander Wolff~~

Lehrstuhl für Informatik I

Ergebnisse 1. Kurztest WS 2018/19



Mitgeschrieben: 215

Mittelwert: 28,2 Punkte (47 % von 60 Punkten)

Median: 27 Punkte (45 % von 60 Punkten)

Wie gut wurden die Aufgaben gelöst?

1. Heaps	58 %
2. Algorithmen	52 %
3. BackwardsSort	38 %
4. Meistermethode	49 %
5. Groß-O-Notation	37 %
6. Logarithmische Laufzeitklassen	41 %
7. Pseudocode	39 %

Zur Erinnerung

Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung

wie wird die gewünschte Funktionalität realisiert:
– wie sind die Daten gespeichert (Feld, Liste, ...)?
– welche Algorithmen implementieren die Operationen?

Beispiel

Prioritätsschlange:

verwaltet Elemente einer Menge M , wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Abstrakter Datentyp

$O(1)$ stellt folgende Operationen bereit: $O(n)$
Insert, FindMax, ExtractMax, IncreaseKey

Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

Beispiel

Prioritätsschlange:

verwaltet Elemente einer Menge M , wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

$O(\log n)$

$O(1)$



Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

Teil III [CLRS]

Dynamische Menge:



verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<pre>ptr Insert(key k, info i)</pre>	<ul style="list-style-type: none"> • lege neuen Datensatz (k, i) an • $M = M \cup \{(k, i)\}$ • gib Zeiger auf (k, i) zurück <div style="text-align: center;"> <p style="margin-left: 150px;">M</p> </div>

Teil III [CLRS]

Dynamische Menge:



verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code>	<ul style="list-style-type: none"> • $M = M \setminus \{(x.key, x.info)\}$ <div style="text-align: center;"> <p>The diagram shows a sequence of elements arranged horizontally. Each element is represented by a green-bordered box divided into two parts: 'key' at the top and 'info' at the bottom. The elements are labeled $key_1, info_1$, $key_2, info_2$, followed by ellipses, and ending with $key_n, info_n$. A large red oval, labeled M above it, encircles all the elements except the last one, $key_n, info_n$. This visualizes the set M after the deletion of element x.</p> </div>

Teil III [CLRS]

Dynamische Menge:



verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code>	<ul style="list-style-type: none">• falls vorhanden, gib Zeiger p mit $p.key = k$ zurück• sonst gib Zeiger nil zurück

Teil III [CLRS]

Dynamische Menge:



verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<pre>ptr Insert(key k, info i) Delete(ptr x) ptr Search(key k) ptr Minimum() ptr Maximum() ptr Predecessor(ptr x) ptr Successor(ptr x)</pre>	<ul style="list-style-type: none"> • sei $M' = \{(k, i) \in M \mid k < x.key\}$ • falls $M' = \emptyset$, gib <i>nil</i> zurück, • sonst gib Zeiger auf (k^*, i^*) zurück, wobei $k^* = \max_{(k,i) \in M'} k$

Teil III [CLRS]

Dynamische Menge:



verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	<i>Funktionalität</i>	
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code>	$\left. \begin{array}{l} \text{ptr Insert(key } k, \text{ info } i) \\ \text{Delete(ptr } x) \\ \text{ptr Search(key } k) \end{array} \right\}$ Änderungen	$\left. \begin{array}{l} \text{ptr Insert(key } k, \text{ info } i) \\ \text{Delete(ptr } x) \\ \text{ptr Search(key } k) \end{array} \right\}$ Änderungen $\left. \begin{array}{l} \text{ptr Minimum()} \\ \text{ptr Maximum()} \\ \text{ptr Predecessor(ptr } x) \\ \text{ptr Successor(ptr } x) \end{array} \right\}$ Anfragen Wörterbuch
<code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	$\left. \begin{array}{l} \text{ptr Minimum()} \\ \text{ptr Maximum()} \\ \text{ptr Predecessor(ptr } x) \\ \text{ptr Successor(ptr } x) \end{array} \right\}$ Anfragen	

Implementierung: je nachdem... Drei Beispiele!

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

boolean Empty()

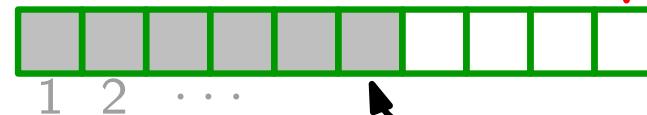
Push(key k)

key Pop()

key Top()

Implementierung

Größe?



key[] A
int top

if $top == 0$ **then return** true
else return false

$top = top + 1$

$A[top] = k$

if Empty() **then error** „underflow“
else

$top = top - 1$

return $A[top + 1]$

if Empty() **then** ... **else return** $A[top]$

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stapel(int n)

boolean Empty()

Push(key k)

key Pop()

key Top()

Implementierung

$A = \text{new}^* \text{key}[1..n]$
 $top = 0$

if $top == 0$ **then return** true
else return false

$top = top + 1$ **{ if** $top > A.length$ **then**
 $A[top] = k$ **{ error** „overflow“}

if Empty() **then error** „underflow“
else

$top = top - 1$
 return $A[top + 1]$

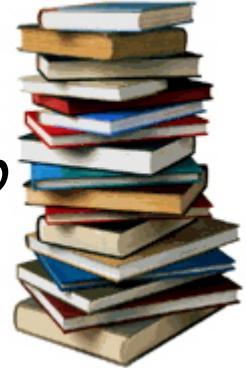
Laufzeiten?

Alle * $O(1)$,
d.h. konstant.

if Empty() **then** ... **else return** $A[top]$

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stapel(int *n*)

boolean Empty()

Push(key *k*)

key Pop()

key Top()

Implementierung

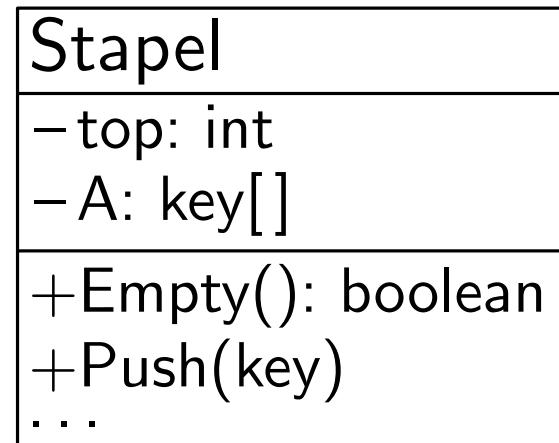
} Konstruktor

Attribute {

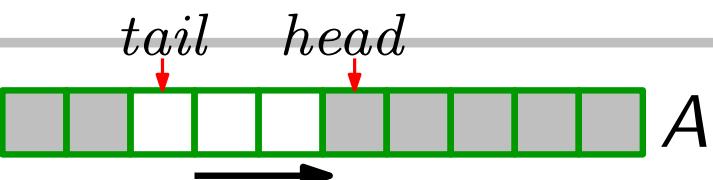
key[] A
int top

} Methoden

Aufgabe:
Fertigen Sie ein
UML-Diagramm für
die Klasse Stapel an!



II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp

Queue(*int n*)

Aufgabe: Fangen Sie underflow & overflow ab!

boolean Empty()

Enqueue(key *k*)

*stelle neues Element
an den Schwanz der
Schlange an*

key Dequeue()

*entnimm Element am
Kopf der Schlange*

Implementierung

A = new^{} key[1..n]
tail = head = 1*

key[] *A*
int *tail*
int *head*

if *head == tail then return true*
else return false

A[tail] = k

if *tail == A.length then tail = 1*
else *tail = tail + 1*

k = A[head]

if *head == A.length then head = 1*

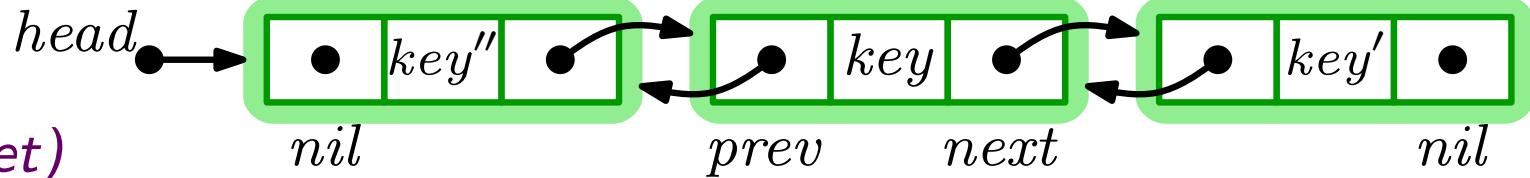
else *head = head + 1*

return *k*

Laufzeiten?
Alle^{*} *O(1)*.

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

ptr Insert(key k)

Implementierung

$head = nil$

$x = head$

while $x \neq nil$ **and** $x.key \neq k$ **do**
 ↳ $x = x.next$
return x

$x = \text{new Item}()$

$x.key = k; x.prev = nil; x.next = head$

if $head \neq nil$ **then** $head.prev = x$
 $head = x$; **return** x

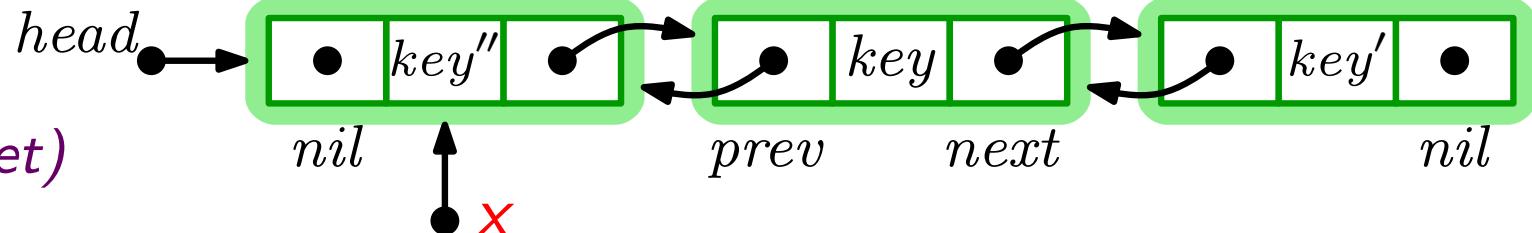
Item

key	key
ptr	prev
ptr	next

ptr	head
-----	------

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

Hausaufgabe:

Benutzen Sie
Stopper!

ptr Insert(key k)

Aufgabe:

Implementieren Sie
Delete(ptr x)

Implementierung

head = nil

Item(key k, ptr p)

key = k
next = p
prev = nil

Item

key key
ptr prev
ptr next

ptr head

x = head

while *x ≠ nil and x.key ≠ k* **do**

 └ *x = x.next*

return *x*

x = new Item(>k, head)

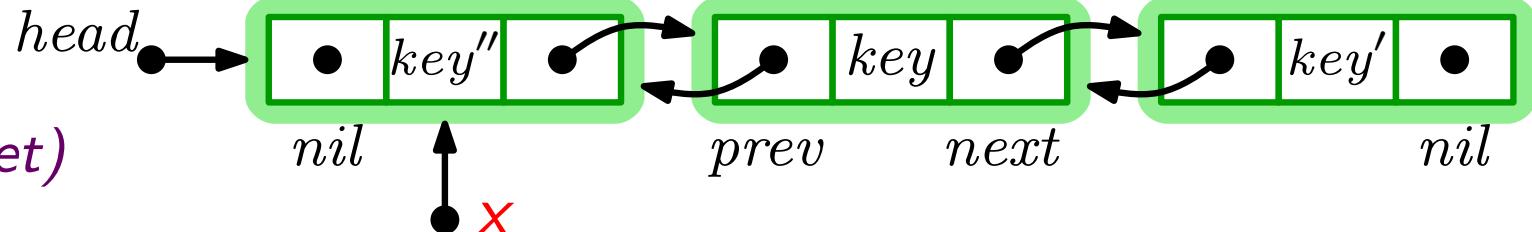
~~*x.key = k, x.prev = nil; x.next = head*~~

if *head ≠ nil* **then** *head.prev = x*

head = x; return x

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

$O(1)$

Laufzeiten?

ptr Search(key k)

$O(n)$

ptr Insert(key k)

$O(1)$

Delete(ptr x)

Implementierung

$head = nil$

Item(key k, ptr p)

$key = k$
 $next = p$
 $prev = nil$

Item

key
ptr
prev
ptr
next

ptr head

$x = head$

while $x \neq nil$ **and** $x.key \neq k$ **do**
 ↳ $x = x.next$

return x

$x = \text{new Item}(k, head)$

~~$x.key = k, x.prev = nil, x.next = head$~~

if $head \neq nil$ **then** $head.prev = x$
 $head = x$; **return** x

Von Pseudocode zu Javacode: (1) Item

```

public class Item {
    private Object key;
    private Item prev;
    private Item next;
}

public Item(Object k, Item p) {
    key = k;
    next = p;
    prev = null;
}

public void setPrev(Item p) { prev = p; }
public void setNext(Item p) { next = p; }
public Item getPrev() { return prev; }
public Item getNext() { return next; }
public Object getKey() { return key; }
}

```

Item(key k, ptr p)
 $key = k$
 $next = p$
 $prev = \text{nil}$

Item

key key
 ptr $prev$
 ptr $next$

setter- und getter- Methoden

Von Pseudocode zu Javacode: (2) List

```
public class List {
```

```
    private Item head;
```

```
    public List() {
        head = null;
    }
```

ptr head

List()
head = nil

```
    public Item insert(Object k) {
```

```
        Item x = new Item(k, head);
```

```
        if (head != null) {
            head.setPrev(x);
        }
```

```
        head = x;
        return x;
    }
```

ptr Insert(key k)

$x = \text{new Item}(k, head)$

if $head \neq \text{nil}$ **then**
 $\quad head.prev = x$

$head = x$

return x

```
    public Item getHead() { return head; }
```

Von Pseudocode zu Javacode: (2) List

```
ptr Search(key k)
    x = head
    while x ≠ nil and x.key ≠ k do
        x = x.next
    return x
```



```
public Item search(Object k) {
    Item x = head;
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

Von Pseudocode zu Javacode: (2) List

```
Delete(ptr x)
```

```
    if x.prev ≠ nil then x.prev.next = x.next  
    else head = x.next  
    if x.next ≠ nil then x.next.prev = x.prev
```



```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
             it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        } Was wird hier ausgegeben?  
    }  
}
```

Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
             it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        }  
  
        Item it = myList.search(new Integer(16));  
        myList.delete(it);  
    }  
}
```

Die Liste enthaelt:
16
10
Fehler!

Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

Listentest.java

```
myList.insert(new Integer(16));  
...  
Item it = myList.search(new Integer(16));  
myList.delete(it);
```

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

Warum "Fehler"?

```

public Item search(Object k) {
    Item x = head;      !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}

public void delete(Item x) {
    if (x == null) System.out.println("Fehler!");
    Item prev = x.getPrev();
    Item next = x.getNext();
    if (prev != null) prev.setNext(next);
    else head = next;
    if (next != null) next.setPrev(prev);
}

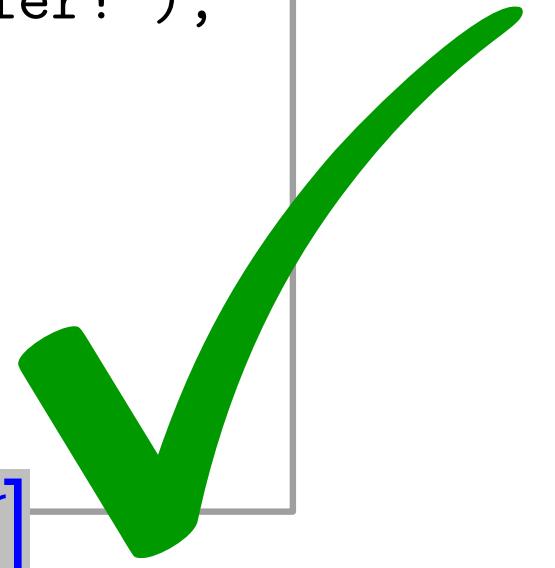
```

Item.java

Listentest.java

myList.insert(new Integer(16));
gleiche Zahlen, aber verschiedene Objekte!
 Item it = myList.search(new Integer(16));
 myList.delete(it);

[Unschön: Klasse Item muss public sein, so dass Anwender und Bibliotheksklasse List darüber kommunizieren können.]



Algorithmen und Datenstrukturen

Wintersemester 2018/19
12. Vorlesung

Hashing

Übungen

- Begründen Sie grundsätzlich alle Behauptungen – außer die Aufgabe verlangt explizit keine Begründung.
- Pseudocode allein genügt *nie!*
Algorithmen immer (auch) mit Worten erklären.
Verweisen Sie dabei auf Zeilen Ihres Pseudocodes.
- Bitte geben Sie auf Ihren Lösungen immer die **Nummer** Ihrer Übungsgruppe an!
- Kommentieren Sie Ihren Java-Code *beim Programmieren!*
Das hilft Fehler zu vermeiden. Außerdem verstehen Sie Ihren Code auch noch beim nächsten Mal, wenn Sie draufschauen.

What's the problem?

Wörterbuch

Spezialfall einer dynamischen Menge

Anwendung: im Compiler Symboltabelle für Schlüsselwörter

Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, Delete, Search

Implementierung

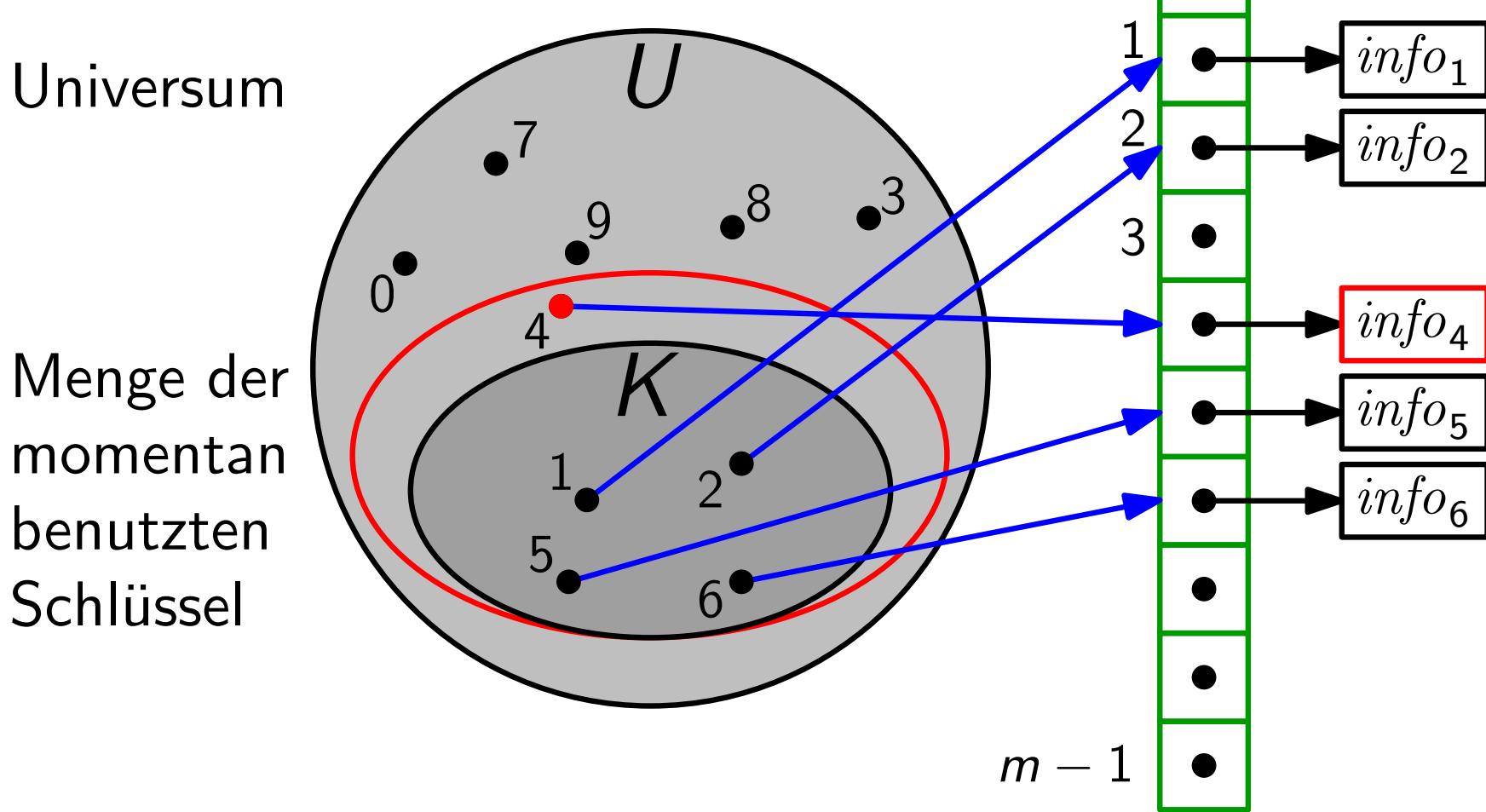
heute: Hashing (engl. to hash = zerhacken, kleinschneiden)

Suchzeit: – im schlechtesten Fall $\Theta(n)$,

– erwartet $O(1)$ unter akzeptablen Annahmen

Direkte Adressierung

- Annahmen:**
- Schlüssel aus kleinem *Universum* $U = \{0, \dots, m-1\}$
 - Schlüssel paarweise verschieden (dyn. Menge!)



Direkte Adressierung

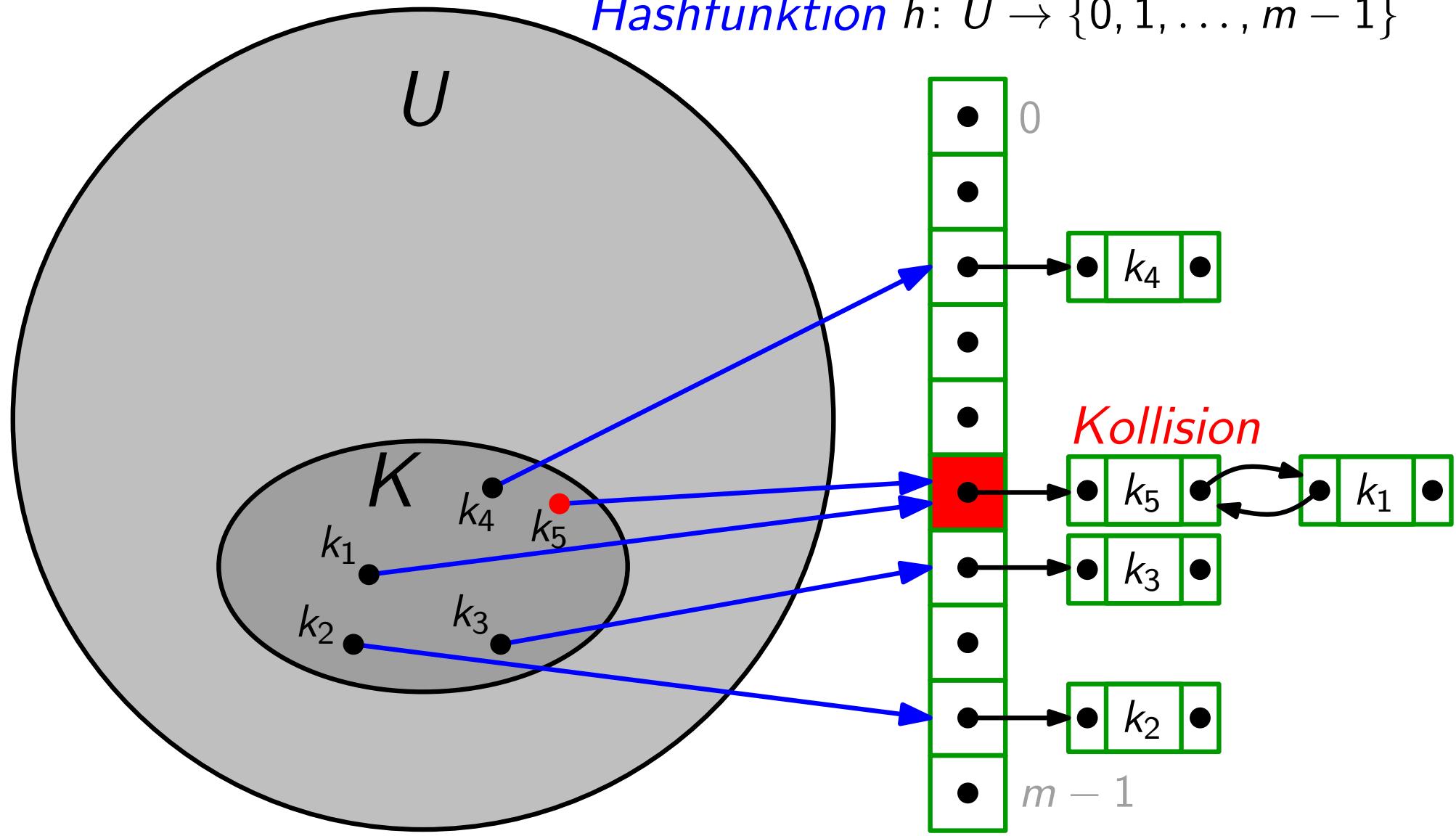
Annahmen: – Schlüssel aus kleinem *Universum* $U = \{0, \dots, m-1\}$
 – Schlüssel paarweise verschieden (dyn. Menge!)

Abs. Datentyp	Implementierung
HashDA(int m)	$T = \text{new } \text{ptr}[0..m-1]$ ptr[] T for $j = 0$ to $m-1$ do $T[j] = \text{nil}$ $\text{// } T[j] = \text{Zeiger auf } j. \text{ Datensatz}$
ptr Insert(key k , info i)	$\text{// lege neuen Datensatz an}$ $\text{// und initialisiere ihn mit } i$ $T[k] = \text{new info}(i)$
Delete(key k)	<ul style="list-style-type: none"> • Speicher freigeben, auf den $T[k]$ zeigt • $T[k] = \text{nil}$
ptr Search(key k)	return $T[k]$ Laufzeiten? Ins, Del, Search $O(1)$ im schlechtesten Fall

Hashing mit Verkettung

Annahme: großes Universum U , d.h. $|U| \gg |K|$

Hashfunktion $h: U \rightarrow \{0, 1, \dots, m - 1\}$



Hashing mit Verkettung

Voraussetzungen: $|U| \gg |K|$, Zugriff auf Hashfunktion h

Abs. Datentyp

```
HashChaining(  
    int m)
```

```
ptr Insert(key k)
```

```
Delete(ptr x)
```

```
ptr Search(key k)
```

Implementierung

```
T = new ptr[0..m - 1]
```

```
ptr[] T
```

```
for j = 0 to m - 1 do T[j] = List()
```

Aufgabe:

Schreiben Sie Insert, Delete & Search.
Verwenden Sie Methoden der DS List!

Hashing mit Verkettung

Voraussetzungen: $|U| \gg |K|$, Zugriff auf Hashfunktion h

Abs. Datentyp

```
HashChaining(  
    int m)
```

```
ptr Insert(key k)
```

```
Delete(ptr x)
```

```
ptr Search(key k)
```

Implementierung

```
T = new ptr[0..m - 1]           ptr[] T  
for j = 0 to m - 1 do T[j] = List()
```

```
return T[h(k)].Insert(k)
```

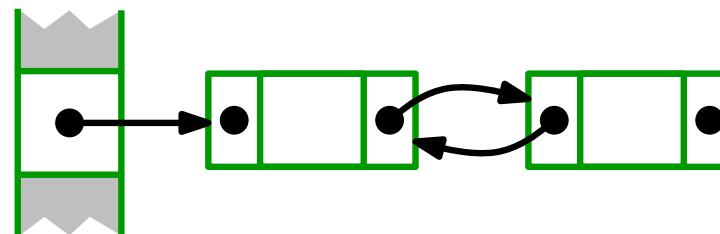
```
T[h(x.key)].Delete(x)
```

```
return T[h(k)].Search(k)
```

Analyse

Definition: Die *Auslastung* α einer Hashtabelle sei n/m , also der Quotient der Anzahl der gespeicherten Elemente und der Tabellengröße.

Bemerkung: α ist die durchschnittliche Länge einer *Kette*.



Laufzeit: $\Theta(n)$ im schlimmsten Fall: z.B. $h(k) = 0 \quad \forall k \in K$.

Annahme: *Einfaches uniformes Hashing*: jedes Element von U wird mit gleicher WK in jeden der m Einträge der Tabelle gehasht – unabhängig von anderen Elementen.
D.h. $\Pr[h(k) = i] = 1/m$

Suche

- Fälle:**
- 1) erfolglose Suche
 - 2) erfolgreiche Suche

Notation: $n_j = T[j].length$ für $j = 0, 1, \dots, m - 1$.

Dann gilt: $n = n_0 + n_1 + \dots + n_{m-1}$.

$$\mathbf{E}[n_j] = n/m = \alpha$$

Satz. Unter der Annahme des einfachen uniformen Hashings durchsucht eine *erfolglose* Suche erwartet α Elemente.

Beweis. Wenn die Suche nach einem Schlüssel k erfolglos ist, muss $T[h(k)]$ komplett durchsucht werden.

$$\mathbf{E}[T[h(k)].length] = \mathbf{E}[n_{h(k)}] = \alpha.$$

Erfolgreiche Suche

Satz. Unter der Annahme des einfachen uniformen Hashings durchsucht eine *erfolgreiche* Suche erwartet höchstens $1 + \alpha/2$ Elemente.

Beweis. Noch 'ne Annahme:

Jedes der n Elemente in T ist mit gleicher WK das gesuchte Element x .

$$\# \text{ durchsuchte Elem.} = \# \text{ Elem. } \underset{\text{räumlich}}{\text{vor}} x \text{ in } T[h(x)] + 1$$

$$X = \# \text{ Elem., die } \underset{\text{zeitlich}}{\text{nach}} x \text{ in } T[h(x)] \text{ eingefügt wurden} + 1$$

Sei x_1, x_2, \dots, x_n die Folge der Schlüssel in der Reihenfolge des Einfügens.

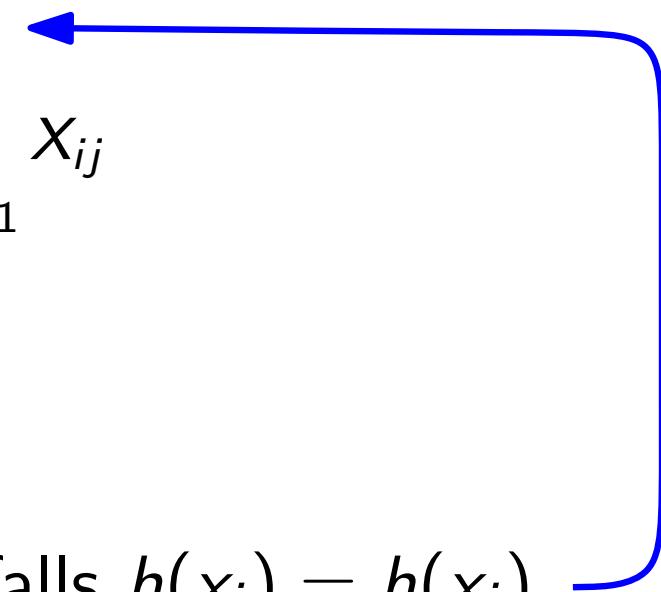
Definiere Indikator-ZV: $X_{ij} = 1$ falls $h(x_i) = h(x_j)$.

Klar: $\mathbf{E}[X_{ij}] = \Pr[X_{ij} = 1] = 1/m$ [einf. unif. Hashing!]

Erfolgreiche Suche

$X = \# \text{ Elem., die nach } x \text{ in } T[h(x)] \text{ eingefügt wurden} + 1$

$$\mathbf{E}[X] = \frac{1}{n} \sum_{i=1}^n \mathbf{E}[1 + \# \text{ Elem., die nach } x_i \text{ in } T[h(x_i)] \text{ eingefügt wurden}]$$

$$= \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{n} \sum_{i=1}^n \underbrace{\mathbf{E} [\# \text{ Elem. } x_j \text{ mit } j > i \text{ und } h(x_i) = h(x_j)]}_{\sum_{j=i+1}^n X_{ij}}$$


Definiere Indikator-ZV: $X_{ij} = 1$ falls $h(x_i) = h(x_j)$.

Erfolgreiche Suche

$X = \# \text{ Elem., die nach } x \text{ in } T[h(x)] \text{ eingefügt wurden} + 1$

$$\mathbf{E}[X] = \frac{1}{n} \sum_{i=1}^n \mathbf{E}[1 + \# \text{ Elem., die nach } x_i \text{ in } T[h(x_i)] \text{ eingefügt wurden}]$$

$$= \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{n} \sum_{i=1}^n \mathbf{E} [\underbrace{\# \text{ Elem. } x_j \text{ mit } j > i \text{ und } h(x_i) = h(x_j)}_{\# \text{ Elem. } x_j \text{ mit } j > i \text{ und } h(x_i) = h(x_j)}]$$

$$= 1 + \frac{1}{n} \sum_{i=1}^n \mathbf{E} \left[\sum_{j=i+1}^n X_{ij} \right]$$

$$= 1 + \frac{1}{n} \sum_{i=1}^n \sum_{j=i+1}^n E[X_{ij}] = 1 + \frac{1}{nm} \sum_{i=1}^n \sum_{j=i+1}^n 1$$

$n - i$

$$= 1 + \frac{1}{nm} \left(n^2 - \frac{n^2 + n}{2} \right) = 1 + \frac{n^2 - n}{2nm} = 1 + \frac{n-1}{2m} < 1 + \frac{\alpha}{2}$$



Zusammenfassung Ergebnisse

Satz. Unter der Annahme des einfachen uniformen Hashings durchsucht beim Hashing mit Verkettung eine

- *erfolgreiche* Suche erwartet höch. $1 + \alpha/2$ Elemente
- *erfolglose* Suche erwartet α Elemente.

Und *Einfügen*? Und *Löschen*?

Satz. Unter der Annahme des einfachen uniformen Hashings laufen *alle Wörterbuch-Operationen* in (erwartet) konstanter Zeit, falls $n = O(m)$.

Was ist eine gute Hashfunktion?

1. so „zufällig“ wie möglich – um der Annahme des einfachen uniformen Hashings möglichst nahe zu kommen.

- Hashfunktion sollte die Schlüssel aus dem Universum U möglichst gleichmäßig über die m Plätze der Hashtabelle verteilen.
- Hashfunktion sollte Muster in der Schlüsselmenge K gut auflösen.

Beispiel: $U = \text{Zeichenketten}$, $K = \text{Wörter der dt. Sprache}$

h : nimm die ersten drei Buchstaben \rightarrow Zahl

schlecht: – viele Wörter fangen mit „sch“ an

\Rightarrow selber Hashwert

– andere Buchst. haben keinen Einfluss

2. einfach zu berechnen!

Annahme: Alle Schlüssel sind (natürliche) Zahlen.

Suche also Hashfunktionen: $\mathbb{N} \rightarrow \{0, \dots, m-1\}$

Rechtfertigung:

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	,	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	-
48	0	64	@	80	P	96	'	112	p		

American
Standard
Code of
Information
Interchange

Zum Beispiel:

$$\text{AW} \rightarrow (65, 87)_{10} = (1000001, 1010111)_2 \rightarrow 1000001\ 1010111_2 \\ = 65 \cdot 128 + 87 = 8407_{10}$$

Divisionsmethode

Hashfunktion $h: \mathbb{N} \rightarrow \{0, \dots, m-1\}$

$$k \mapsto k \bmod m$$

Beispiel: $h(k) = k \bmod 1024$

$$h(1026) = 2$$

$$h(2050) = 2$$

10 niedrigwertigste Stellen

$$1026_{10} = 01\overbrace{00000000}^{10 \text{ niedrigwertigste Stellen}}10_2$$

$$2050_{10} = 10000000010_2$$

D.h. die 2 höherwertigsten Stellen werden von h ignoriert

Moral: vermeide $m = \text{Zweierpotenz}$

Strategie: wähle für m eine Primzahl, entfernt von Zweierpotenz



löst Muster gut auf

Multiplikationsmethode

Hashfunktion $h: \mathbb{N} \rightarrow \{0, \dots, m-1\}$

$$k \mapsto \lfloor m \cdot (\underbrace{kA \bmod 1}_{\text{gebrochener Anteil von } kA}) \rfloor, \text{ wobei } 0 < A < 1.$$

gebrochener Anteil von kA
d.h. $kA - \lfloor kA \rfloor$

- Verschiedene Werte von A „funktionieren“ verschieden gut.

gut: z.B. $A \approx \frac{\sqrt{5}-1}{2}$ [Knuth: The Art of Computer Programming III, '73]

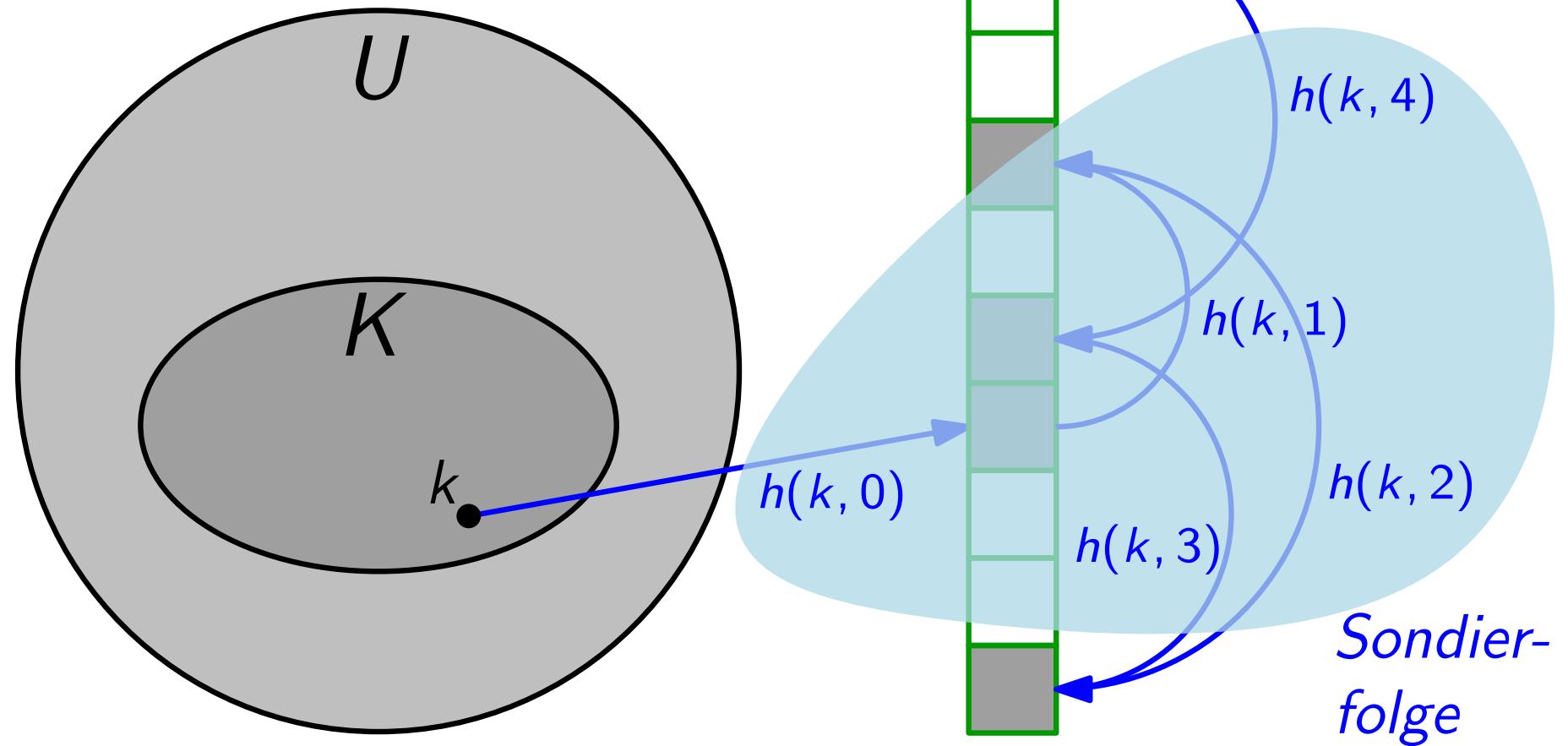
- Vorteil ggü. Divisionsmethode: Wahl von m relativ beliebig.
Insbesondere $m = \text{Zweierpotenz}$ möglich.
 \Rightarrow schnell berechenbar (in Java verschiebt `a << s` die Dualzahlendarstellung von a um s Stellen nach links)

Hashing mit offener Adressierung

Alle Elemente werden direkt in der Hashtabelle gespeichert.

⇒ Tabelle kann volllaufen $\Rightarrow \alpha \leq 1$

Strategie zur Kollisionsauflösung:



Hashing mit offener Adressierung

Abs. Datentyp

HashOA(*int m*)

int Insert(key k)

int Search(key k)

Implementierung

$T = \text{new key}[0..m - 1]$

key[] T

for $j = 0$ **to** $m - 1$ **do** $T[j] = -1$

$i = 0$

repeat

$j = h(k, i)$

if $T[j] == -1$ **then**

$T[j] = k$

return j

else $i = i + 1$

until $i == m$

error “table overflow”

Aufgabe:

Schreiben Sie Search mit repeat-Schleife!

Hashing mit offener Adressierung

Abs. Datentyp

HashOA(int m)

int ~~Insert(key k)~~
Search

...und Delete()?

Umständlich!
Dann lieber
Hashing mit
Verkettung!

int Search(key k)

Implementierung

$T = \text{new key}[0..m - 1]$

key[] T

for $j = 0$ to $m - 1$ do $T[j] = -1$

$i = 0$

repeat

$j = h(k, i)$

if $T[j] == -1$ **then**

~~$T[j] = k$~~

return j

else $i = i + 1$

until $i == m$ **or** $T[j] == -1$

error "table overflow"
return -1

Aufgabe:
Schreiben Sie Search mit repeat-Schleife!

Berechnung von Sondierfolgen

Hashfkt. für offene Adr. $h: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$
 $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ heißt *Sondierfolge* (für k).

Voraussetzungen:

- eine Sondierfolge ist eine Permutation von $\langle 0, 1, \dots, m-1 \rangle$
(Sonst durchläuft die Folge nicht alle Tabelleneinträge genau $1\times!$)
- Existenz von „gewöhnlicher“ Hashfkt $h_0: U \rightarrow \{0, \dots, m-1\}$

Verschiedene Typen von Sondierfolgen:

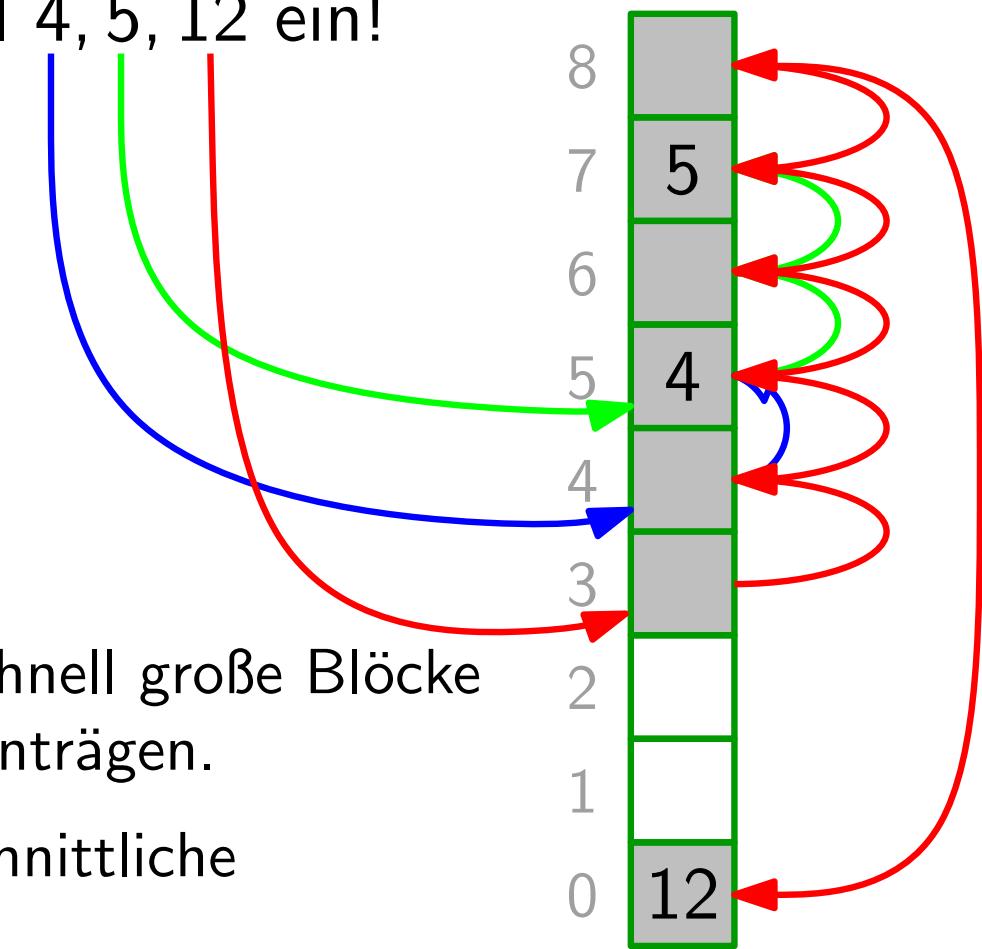
- Lineares Sondieren:
$$h(k, i) = (h_0(k) + i) \bmod m$$
- Quadratisches Sondieren:
$$h(k, i) = (h_0(k) + c_1 i + c_2 i^2) \bmod m$$
- Doppeltes Hashing:
$$h(k, i) = (h_0(k) + i \cdot h_1(k)) \bmod m$$

Lineares Sondieren

Hashfunktion $h(k, i) = (h_0(k) + i) \bmod m$

Beispiel: $h_0(k) = k \bmod 9$ und $m = 9$

Füge Schlüssel 4, 5, 12 ein!



Problem:

primäres
Clustering

Es bilden sich schnell große Blöcke von besetzten Einträgen.

⇒ hohe durchschnittliche Suchzeit!

Quadratisches Sondieren

Hashfunktion $h(k, i) = (h_0(k) + c_1 i + c_2 i^2) \bmod m$

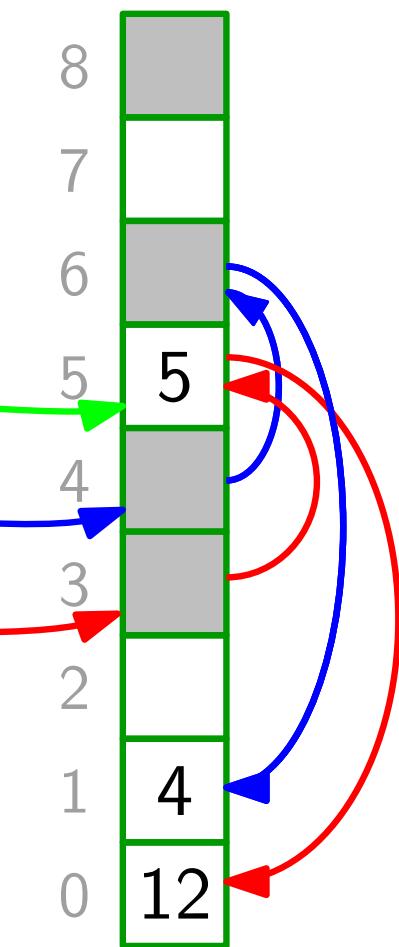
Beispiel: $h_0(k) = k \bmod 9$ und $m = 9$ und $c_1 = c_2 = 1$

Füge Schlüssel 4, 5, 12 ein!

Problem: Die Größen m , c_1 und c_2 müssen zu *einander passen*, sonst besucht nicht jede Sondierfolge alle Tabelleneinträge.

Problem: Falls $h_0(k) = h_0(k')$, so haben k und k' dieselbe Sondierfolge!

⇒ hohe Suchzeit bei schlechter Hilfshashfkt. h_0 !



sekundäres Clustering

Doppeltes Hashing

Hashfunktion $h(k, i) = (h_0(k) + i \cdot h_1(k)) \bmod m$

- Vorteile:**
- Sondierfolge hängt zweifach vom Schlüssel k ab!
 - potentiell m^2 verschiedene Sondierfolgen möglich
(bei linearem & quadratischem Sondieren nur m .)

Frage: Was muss gelten, damit eine Sondierfolge alle Tabelleneinträge durchläuft?

Antwort: $k' = h_1(k)$ und m müssen *teilerfremd* sein ,
d.h. $\text{ggT}(k', m) = 1$. [$\text{ggT}(a, b) =_{\text{Def.}} \max\{t : t|a \text{ und } t|b\}$]

Also: z.B. $m =$ Zweierpotenz und h_1 immer ungerade.
oder $m =$ prim und $0 < h_1(k) < m$ für alle k .

Uniformes Hashing

[kein neues Hashverfahren, sondern eine (idealisierte) Annahme...]

Annahme: Die Sondierfolge jedes Schlüssels ist gleich wahrscheinlich eine der $m!$ Permutationen von $\langle 0, 1, \dots, m - 1 \rangle$.

Satz: Unter der Annahme von uniformem Hashing ist die erwartete Anz. der versuchten Tabellenzugriffe bei offener Adressierung und

- erfolgloser Suche $\leq \frac{1}{1 - \alpha}$
- erfolgreicher Suche $\leq \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$

d.h. Suche dauert erwartet $O(1)$ Zeit, falls α konst.

Zusammenfassung Hashing

mit Verkettung

- + funktioniert für $n \in O(m)$

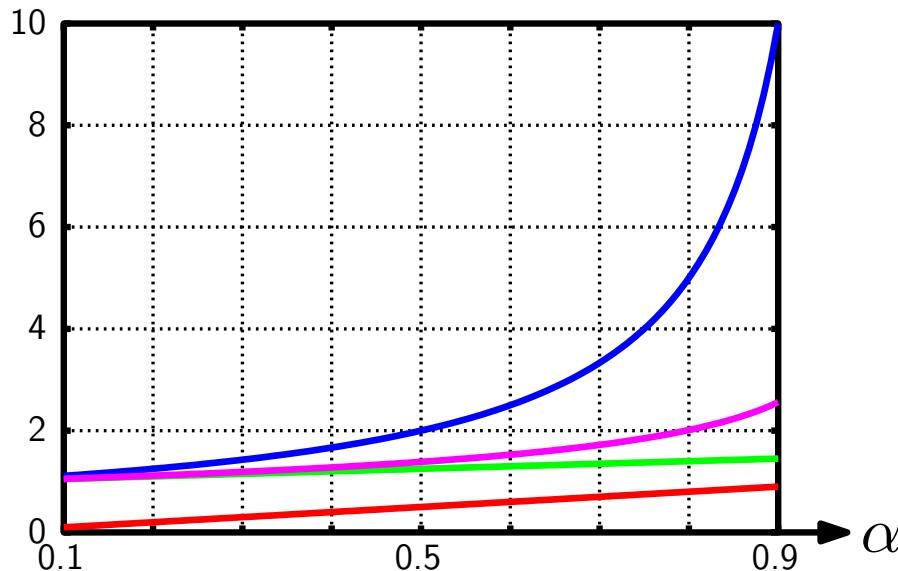
- + gute erwartete Suchzeit:

erfolglos: α [= n/m]

erfolgreich: $1 + \frac{\alpha}{2}$

[Modell: einfaches uniformes Hashing]

- Listenoperationen langsam



mit offener Adressierung

- funktioniert nur für $n \leq m$

- langsam, wenn $n \approx m$

Sondiermethoden:

- lineares Sondieren
- quadratisches Sondieren
- doppeltes Hashing

- + gute erwartete Suchzeit:

erfolglos: $\frac{1}{1-\alpha}$

erfolgreich: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

[Modell: uniformes Hashing]

Algorithmen und Datenstrukturen

Wintersemester 2018/19
13. Vorlesung

Binäre Suchbäume

Zwischentest II: Do, 20. Dez, 8:30 – 10:00

- Zufallsexperimente, (Indikator-) Zufallsvariable, Erwartungswert
- (Randomisiertes) QuickSort
- Untere Schranke für WC-Laufzeit von vergleichsbasierten Sortierverfahren
- Linearzeit-Sortierverfahren
- Auswahlproblem (Median)
- Elementare Datenstrukturen
- Hashing
- Binäre Suchbäume

Anmeldung: ab sofort, aber nur bis zum 18.12.

Dynamische Menge

verwaltet Elemente einer
sich ändernden Menge M



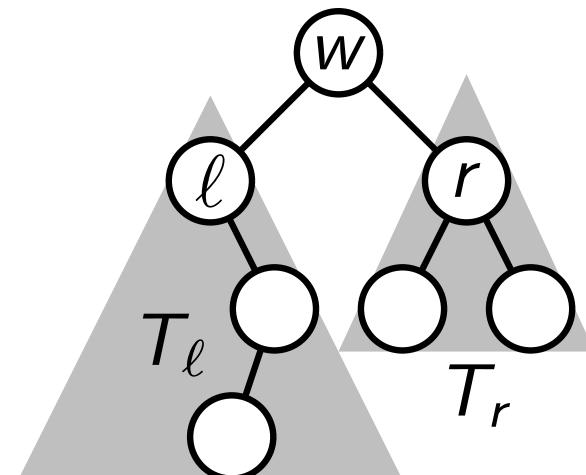
Abstrakter Datentyp	<i>Funktionalität</i>	
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code>	$\left. \begin{array}{l} \text{ptr Insert(key } k, \text{info } i) \\ \text{Delete(ptr } x) \\ \text{ptr Search(key } k) \end{array} \right\}$ Änderungen	$\left. \begin{array}{l} \text{ptr Insert(key } k, \text{info } i) \\ \text{Delete(ptr } x) \\ \text{ptr Search(key } k) \end{array} \right\}$ Änderungen
<code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	$\left. \begin{array}{l} \text{ptr Minimum()} \\ \text{ptr Maximum()} \\ \text{ptr Predecessor(ptr } x) \\ \text{ptr Successor(ptr } x) \end{array} \right\}$ Anfragen	$\left. \begin{array}{l} \text{ptr Minimum()} \\ \text{ptr Maximum()} \\ \text{ptr Predecessor(ptr } x) \\ \text{ptr Successor(ptr } x) \end{array} \right\}$ Anfragen

Implementierung: je nachdem...

Implementierung

\star) unter bestimmten Annahmen.

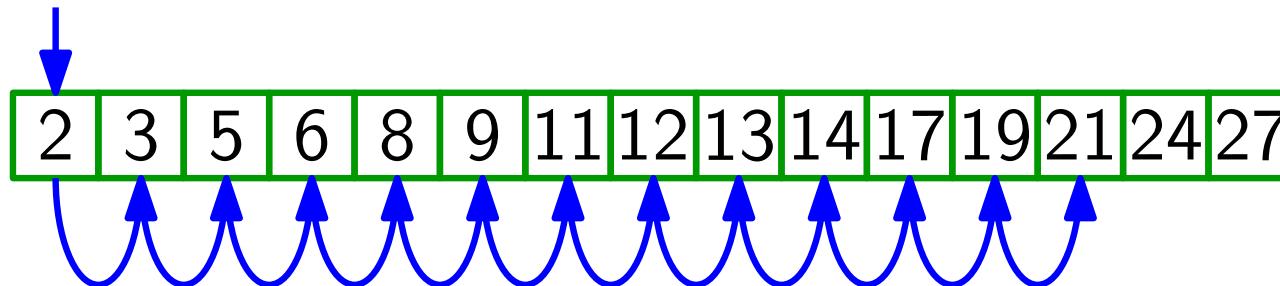
	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^\oplus$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$



$$\begin{aligned}
 h(T) &= \text{Höhe des Baums } T \\
 &= \text{Anz. Kanten auf längstem Wurzel-Blatt-Pfad} \\
 &= \begin{cases} 0 & \text{falls Baum = Blatt} \\ 1 + \max\{h(T_\ell), h(T_r)\} & \text{sonst.} \end{cases}
 \end{aligned}$$

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

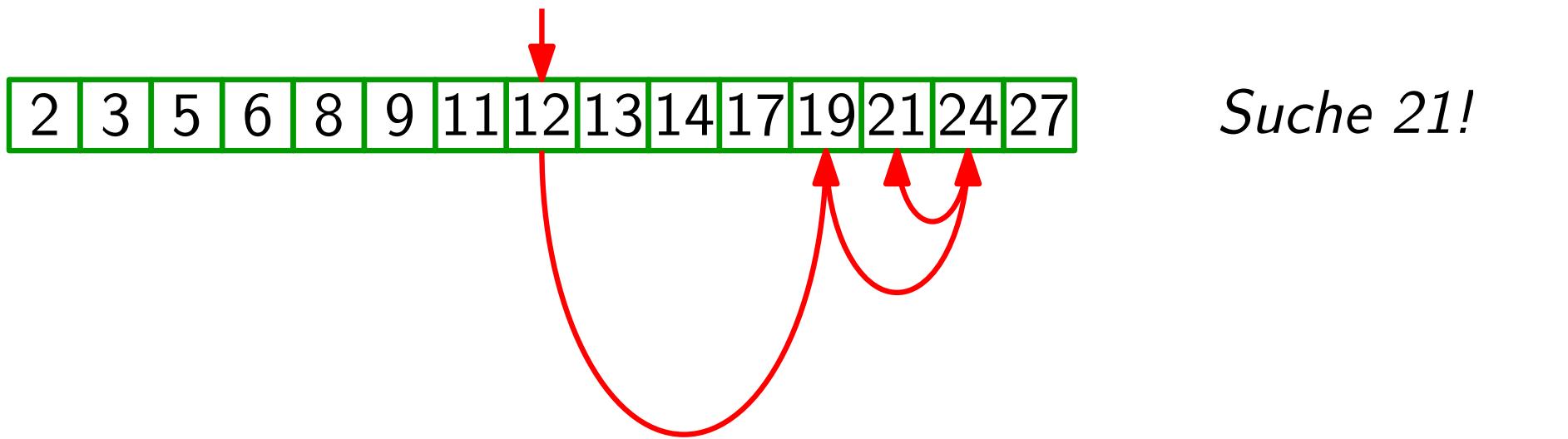
Suche im sortierten Feld



Suche 21!

Lineare Suche: 13 hier im Worst Case
 n Schritte

Suche im sortierten Feld



Lineare Suche: hier im Worst Case
 Schritte

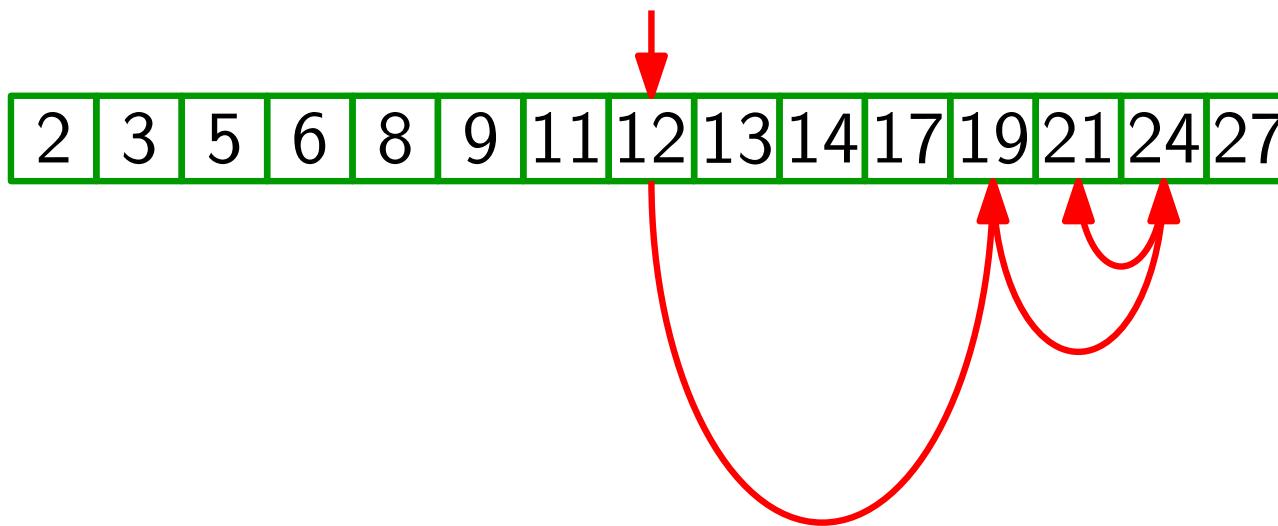
Binäre Suche: 4 ?
 Schritte*

groß: Wie oft muss ich n halbieren, bis ich bei 1 bin?

$$\begin{aligned}
 \text{genau: } T(n) &\leq T(\lfloor n/2 \rfloor) + 1 \text{ und } T(1) = 1 \\
 &\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}_{\lfloor \log_2 n \rfloor} \\
 \textcolor{red}{Übung!} \quad &= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil
 \end{aligned}$$

*) Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

Suche im sortierten Feld

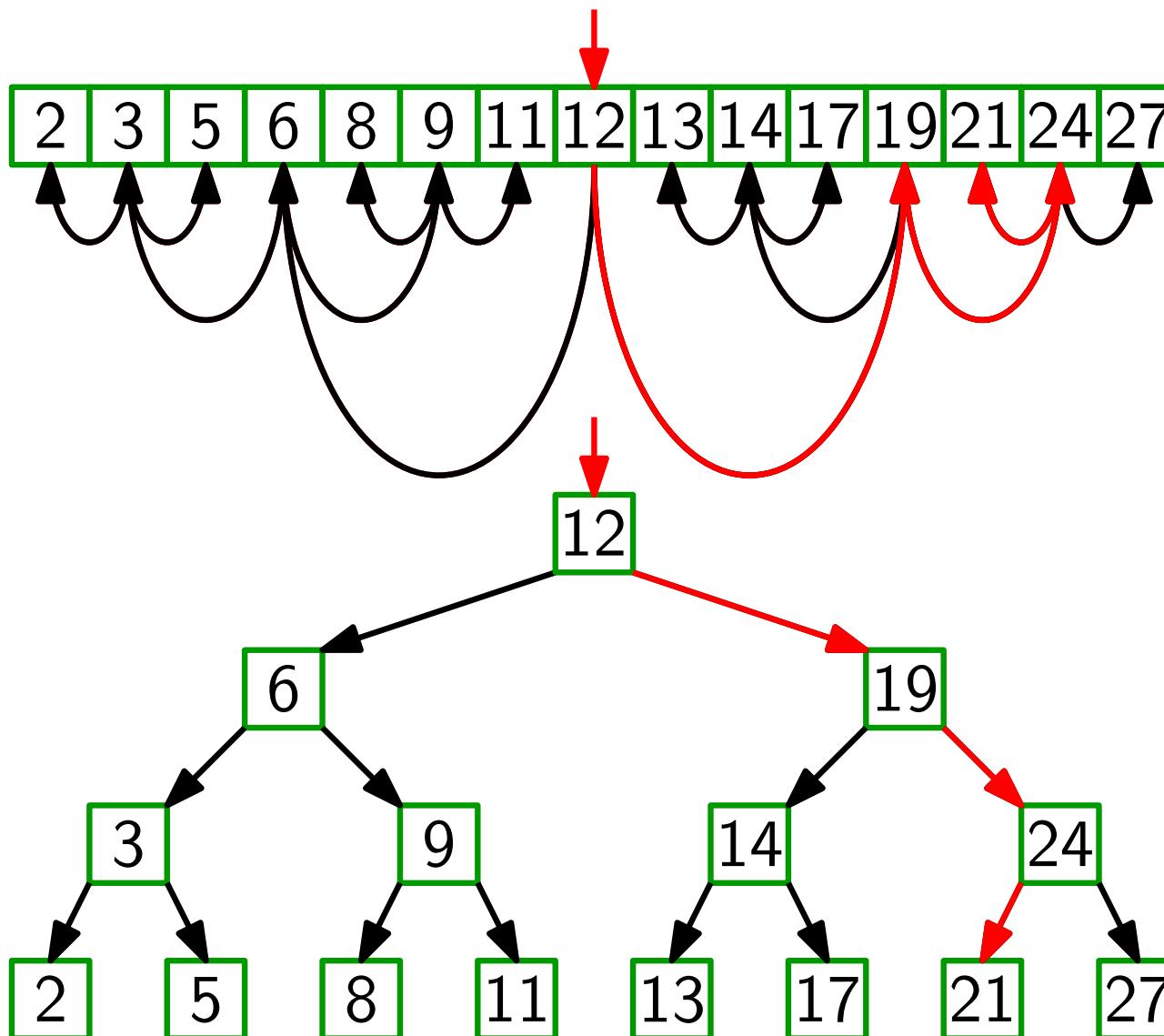


Suche 21!

	<i>hier</i>	<i>im Worst Case</i>		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte*	20

*) Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

Suche im sortierten Feld



Suche 21!

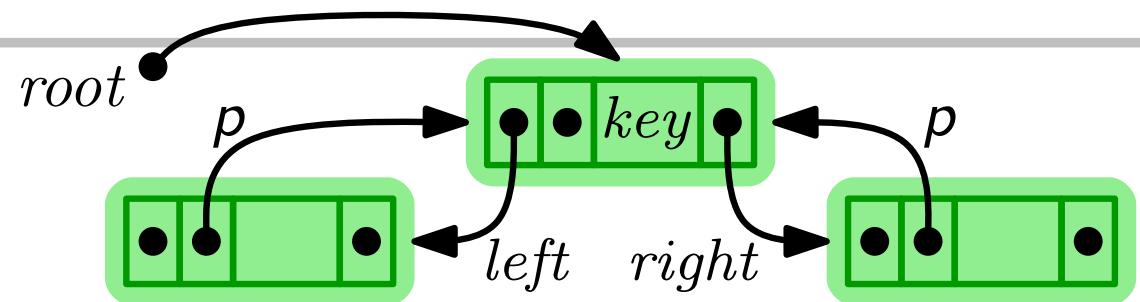
*Binärer
Suchbaum*

Binärer-Suchbaum-Eigenschaft:

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
rechten

Für jeden Knoten v gilt:
 $\geq \leq$

Bin. Suchbaum



Abs. Datentyp

BinSearchTree()

Node Search(key k)

Node Insert(key k)

Delete(Node x)

Node Minimum()

Node Maximum()

Node Predecessor(Nd. x)

Node Successor(Node x)

Implementierung

root = nil

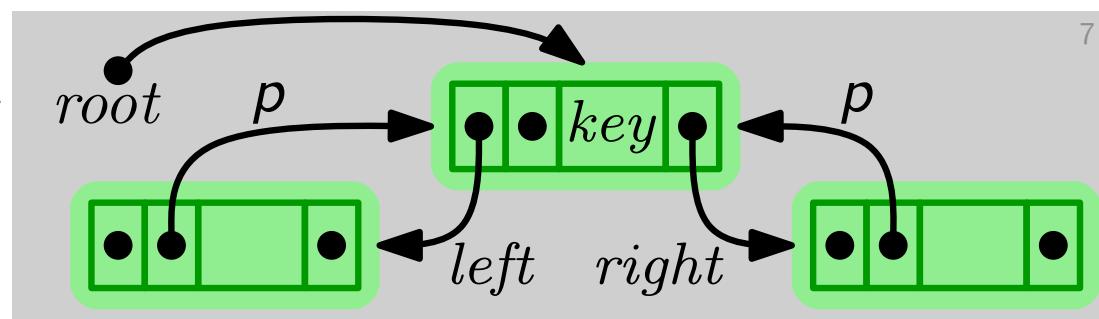
Node(key k, Node par)
key = k
p = par
right = left = nil

Node root

Node
key key
Node left
Node right
Node p

TO α !

Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

```
InorderTreeWalk(Node x = root)
```

```
if x ≠ nil then
    InorderTreeWalk(x.left)
    gib x.key aus
    InorderTreeWalk(x.right)
```

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $\text{root} = \text{nil}$



$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. [rekursive Def. der Höhe!]

Also werden *ihre* Schlüssel sortiert ausgegeben.

Binärer-Suchbaum-Eigenschaft \Rightarrow

Ausgabe (sortierte Schlüssel von T_{links} , dann $\text{root}.key$, dann sortierte Schlüssel von T_{rechts}) ist sortiert.



Code:

InorderTreeWalk(Node $x = \text{root}$)

if $x \neq \text{nil}$ **then**

InorderTreeWalk($x.\text{left}$)

gib $x.\text{key}$ aus

InorderTreeWalk($x.\text{right}$)

Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$


Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

Code:

```
InorderTreeWalk(Node x = root)
  if x ≠ nil then
    InorderTreeWalk(x.left)
    gib x.key aus
    InorderTreeWalk(x.right)
```

Laufzeit

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

Übung: zeig's
mit Induktion!

Code:

InorderTreeWalk(Node $x = root$)

```

if  $x \neq nil$  then
    InorderTreeWalk( $x.left$ )
    gib  $x.key$  aus
    InorderTreeWalk( $x.right$ )
  
```

Laufzeit

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. von Schritten aus.

Für Bäume gilt: $\#\text{Kanten} = \#\text{Knoten} - 1 = n - 1$

$\Rightarrow T(n) = c_1 \cdot (n - 1) + c_2 \cdot n \in O(n).$

Code:

```
InorderTreeWalk(Node x = root)
```

```
if x ≠ nil then
```

```
    InorderTreeWalk(x.left)
```

```
    gib x.key aus
```

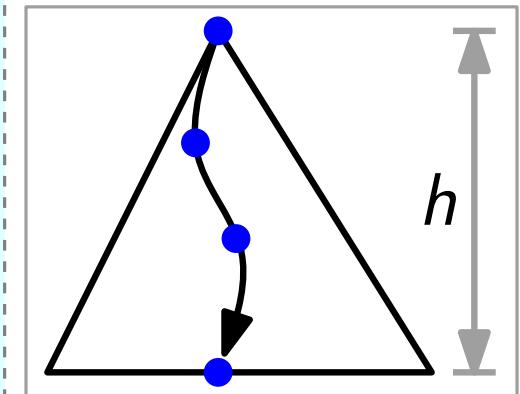
```
    InorderTreeWalk(x.right)
```

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

rekursiv

```
Node Search(key k, Node x = root)
  if x == nil or x.key == k then
    ↘ return x
  if k < x.key then
    ↘ return Search(k, x.left)
  else return Search(k, x.right)
```



Laufzeit: $O(h)$

iterativ

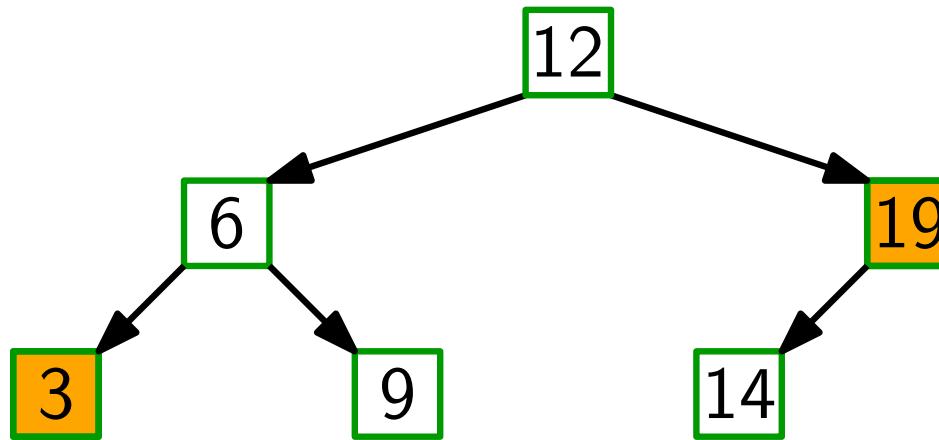
```
while x ≠ nil and x.key ≠ k do
  if k < x.key then
    ↘ x = x.left
  else x = x.right
return x
```

Laufzeit: $O(h)$

Trotzdem schneller,
da keine Verwaltung
der rekursiven
Methodenaufrufe.

Minimum & Maximum

Frage: Was folgt aus der Binäre-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Antwort: Min steht ganz links, Max ganz rechts!

Aufgabe: Schreiben Sie für binäre Suchbäume die Methode

`Node Minimum(Node $x = root$) — iterativ!`

if $x == nil$ **then return** nil

while $x.left \neq nil$ **do**

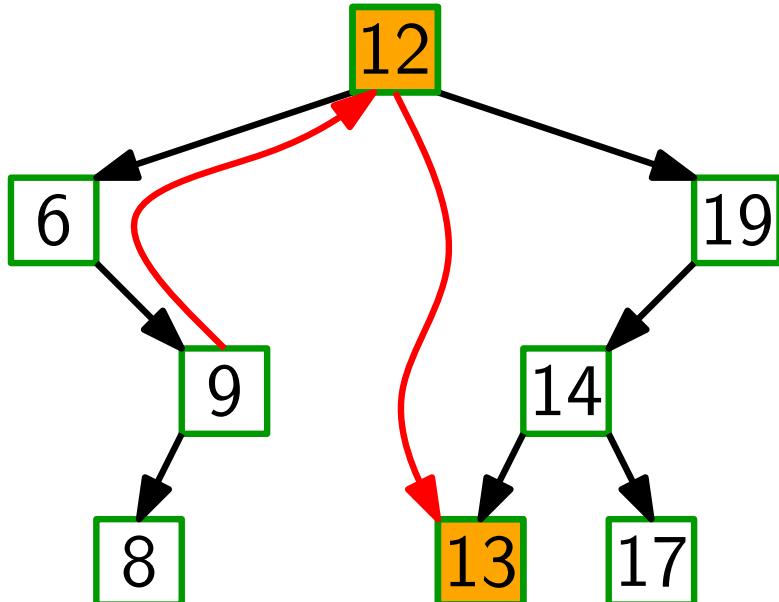
$x = x.left$

return x

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{Nachfolger}(x) = \text{Knoten mit kleinstem Schlüssel}$
 unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



$\text{Nachfolger}(19) := \text{nil}$

$\text{Nachfolger}(12) = ?$

$\text{Nachfolger}(9) = ?$

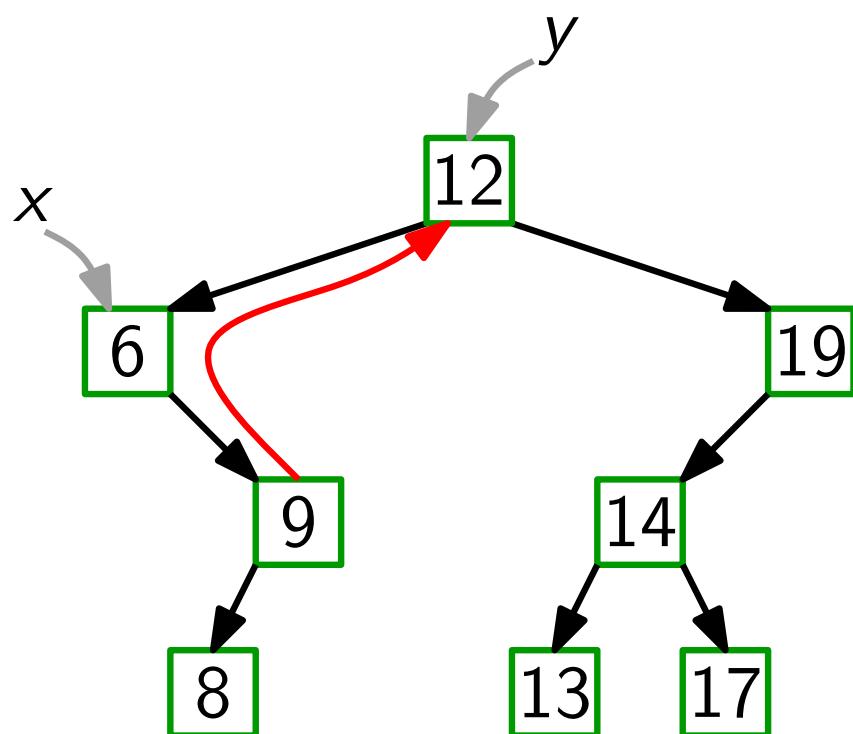
$13 == \text{Minimum}(\text{,,}12.\text{right}\text{,,})$

9 hat kein rechtes Kind; $9 == \text{Maximum}(\text{,,}12.\text{left}\text{,,})$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: Nachfolger(x) = Knoten mit kleinstem Schlüssel unter allen y mit $y.key > x.key$.
 $= \arg \min_y \{y.key \mid y.key > x.key\}.$



Tipp: Probieren Sie auch z.B. Successor("19")!

Node Successor(Node x)

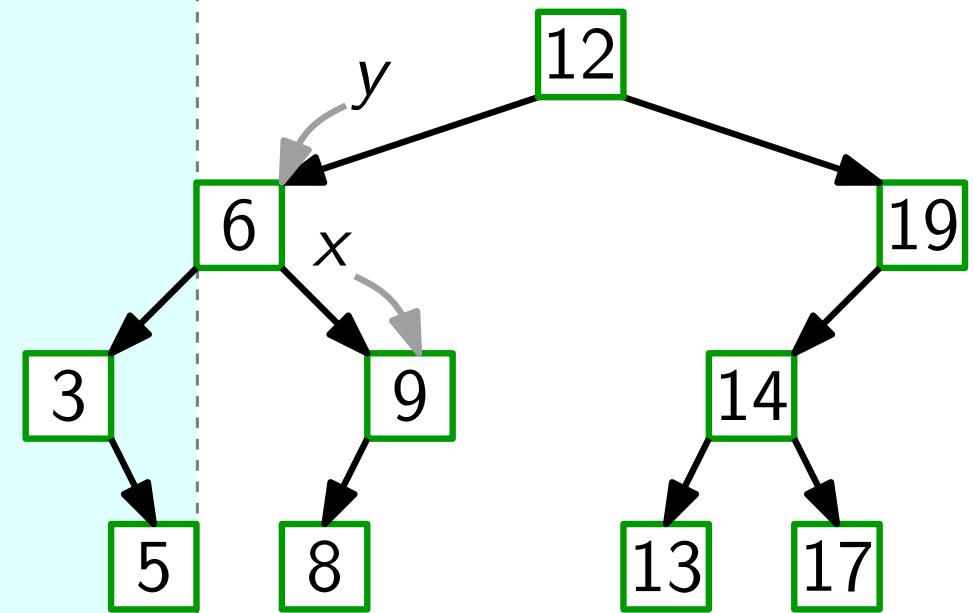
```

if  $x.right \neq \text{nil}$  then
  return Minimum( $x.right$ )
 $y = x.p$ 
while  $y \neq \text{nil}$  and  $x == y.right$  do
   $x = y$ 
   $y = y.p$ 
return  $y$ 
  
```

Einfügen

```

Node Insert(key k)
  y = nil
  x = root
while x ≠ nil do
  y = x
  if k < x.key then
    x = x.left
  else x = x.right
  
```



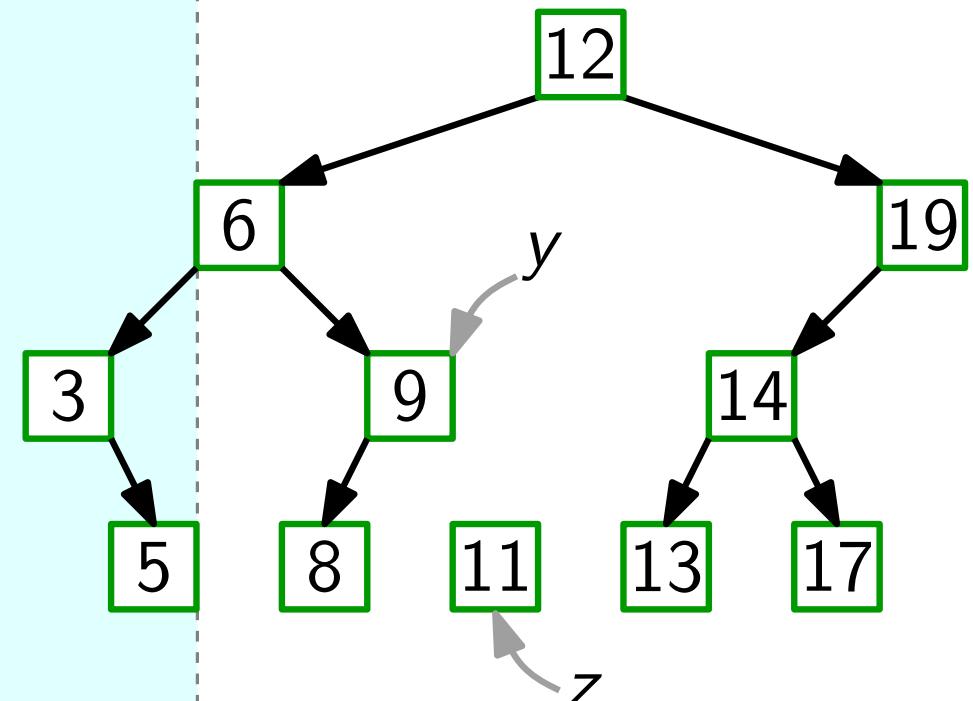
Insert(11)

Einfügen

```

Node Insert(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)

```



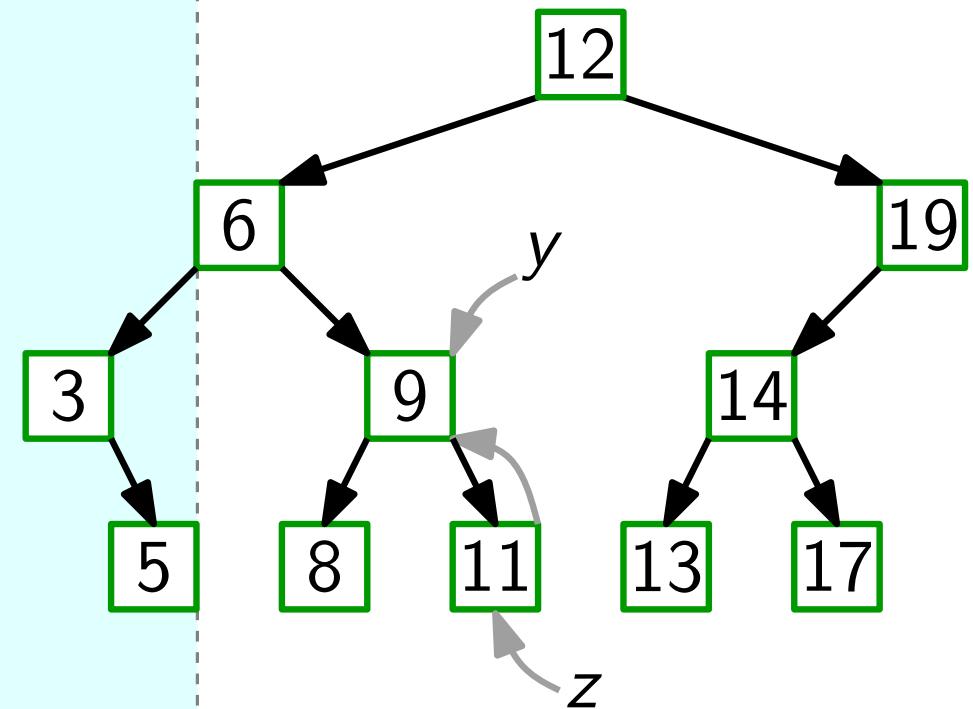
Insert(11)
 $x == \text{nil}$

Einfügen

```

Node Insert(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  z = new Node(k, y)
  if y == nil then root = z
  else
    if k < y.key then y.left = z
    else y.right = z
  return z

```



Insert(11)
x == nil

Löschen

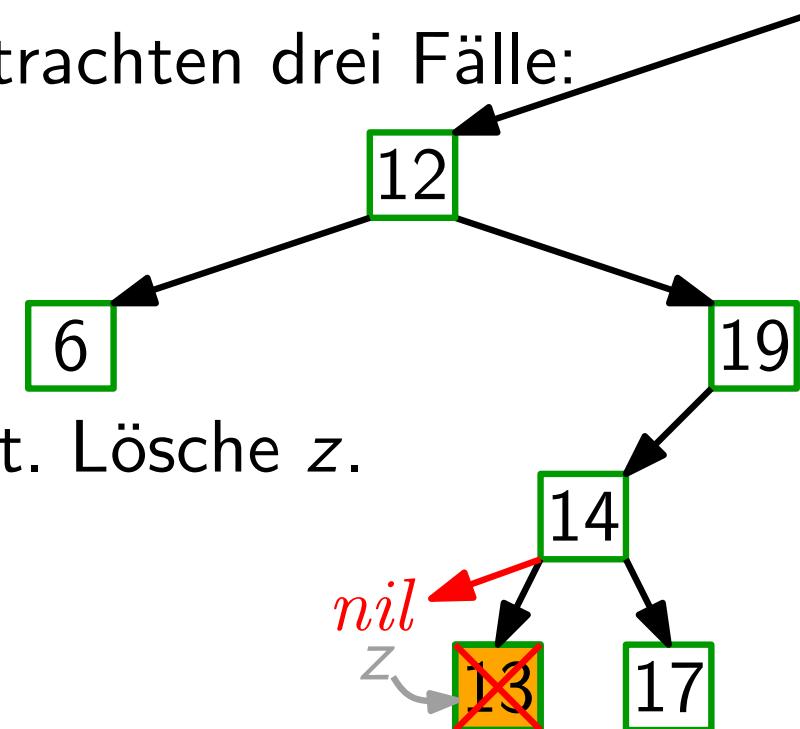
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

- 1.** z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

- 2.** z hat ein Kind x .

- 3.** z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

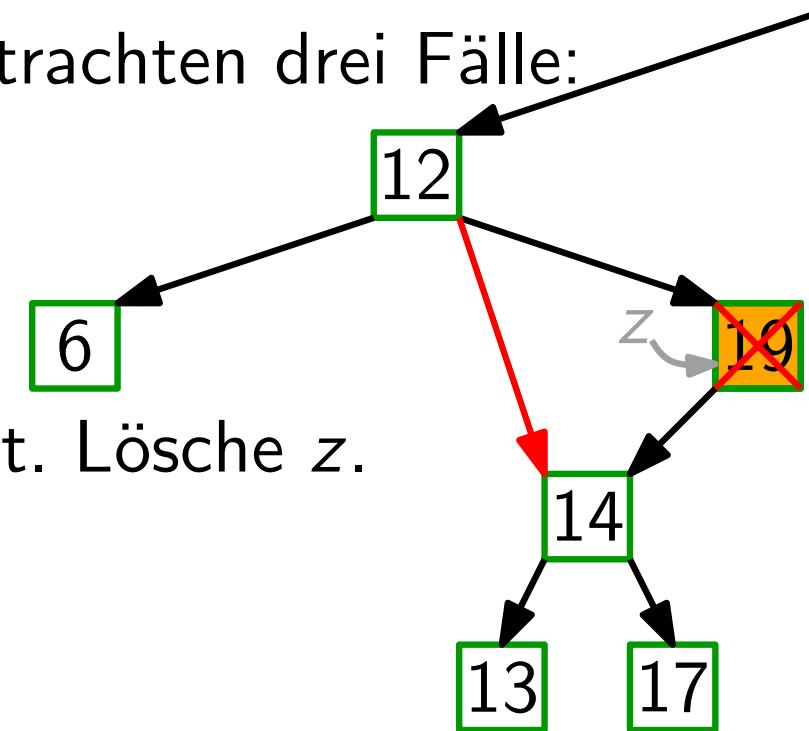
1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschenende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,

setze $z.p.left = nil$; sonst umgekehrt. Lösche z.

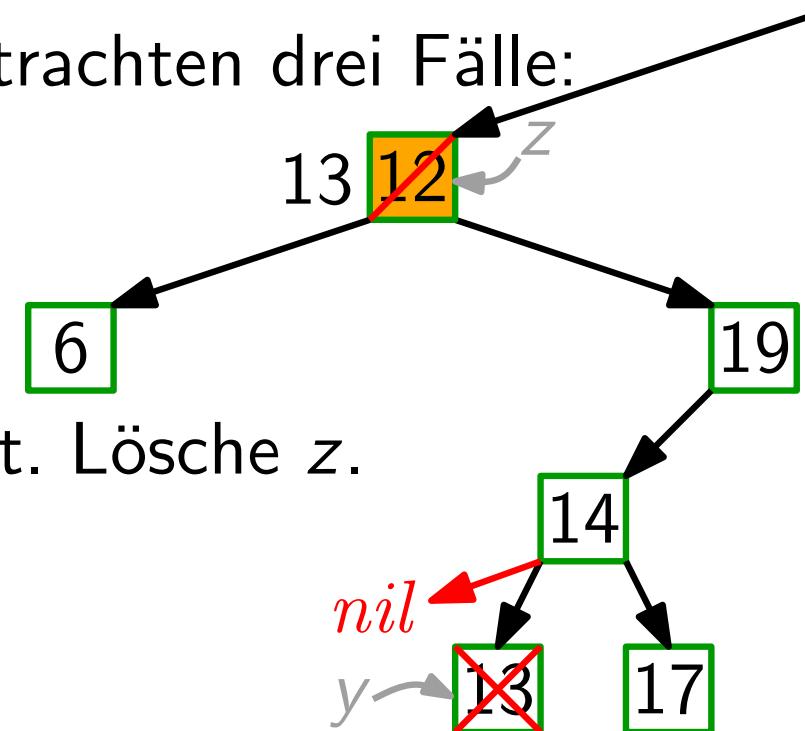
2. z hat ein Kind x.

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .

Setze $x.p = z.p$. Lösche z .

- 3.** z hat zwei Kinder.

Setze $y = \text{Successor}(z)$ und $z.key = y.key$. Lösche y . (Fall 1 oder 2!)



Zusammenfassung

- Satz.** Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $O(h)$ Zeit, wobei h die momentane Höhe des Baums ist.
- Aber:** Im schlechtesten Fall gilt $h \in \Theta(n)$.
- Ziel:** Suchbäume *balancieren* $\Rightarrow h \in O(\log n)$



Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Effiziente Algorithmen und
wissensbasierte Systeme



Algorithmen und Datenstrukturen

Wintersemester 2018/19
14. Vorlesung

Rot-Schwarz-Bäume

Dynamische Menge

verwaltet Elemente einer
sich ändernden Menge M



Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code> <code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	<p>} Änderungen</p> <p>} Anfragen</p>

Implementierung: je nachdem...

Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $O(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

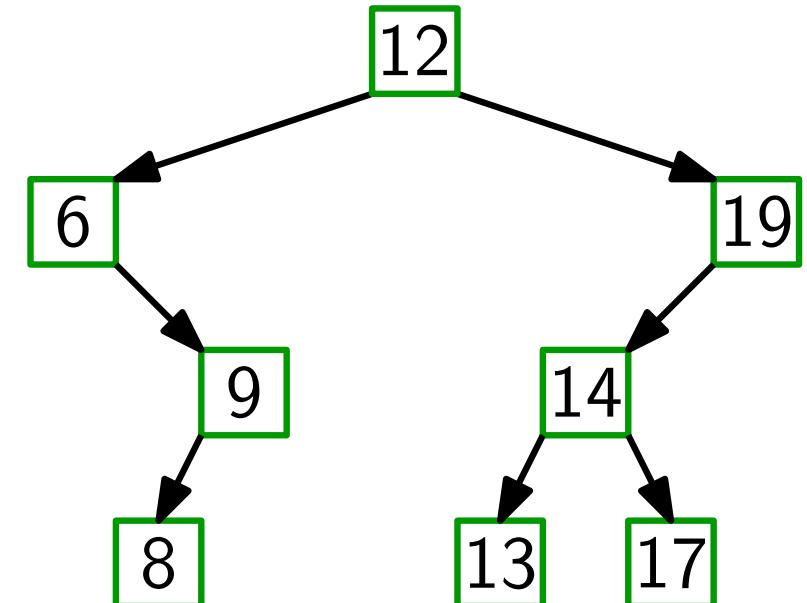
Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$

Binärer-Suchbaum-Eigenschaft:

Für jeden Knoten v gilt:

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
 alle Knoten im rechten Teilbaum von v haben Schlüssel $\geq v.key$



Balanciermethoden

Beispiele

nach **Gewicht**

$BB[\alpha]$ -Bäume

für jeden Knoten ist das Gewicht (= Anzahl der Knoten) von linkem u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

AVL-Bäume*

für jeden Knoten ist die Höhe von linkem und rechtem Teilbaum ungefähr gleich.

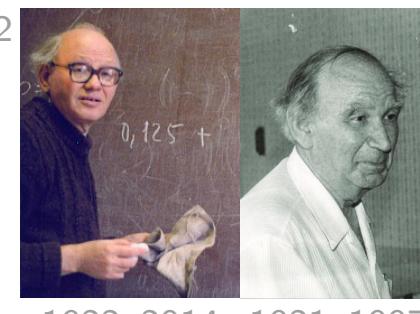


*) Georgi M. Adelson-Velski & Jewgeni M. Landis, Doklady Akademii Nauk SSSR, 1962

nach **Grad**

(2, 3)-Bäume

alle Blätter haben dieselbe Tiefe, aber innere Knoten können verschieden viele Kinder haben.



1922–2014 1921–1997

nach **Knotenfarbe**

Rot-Schwarz-Bäume

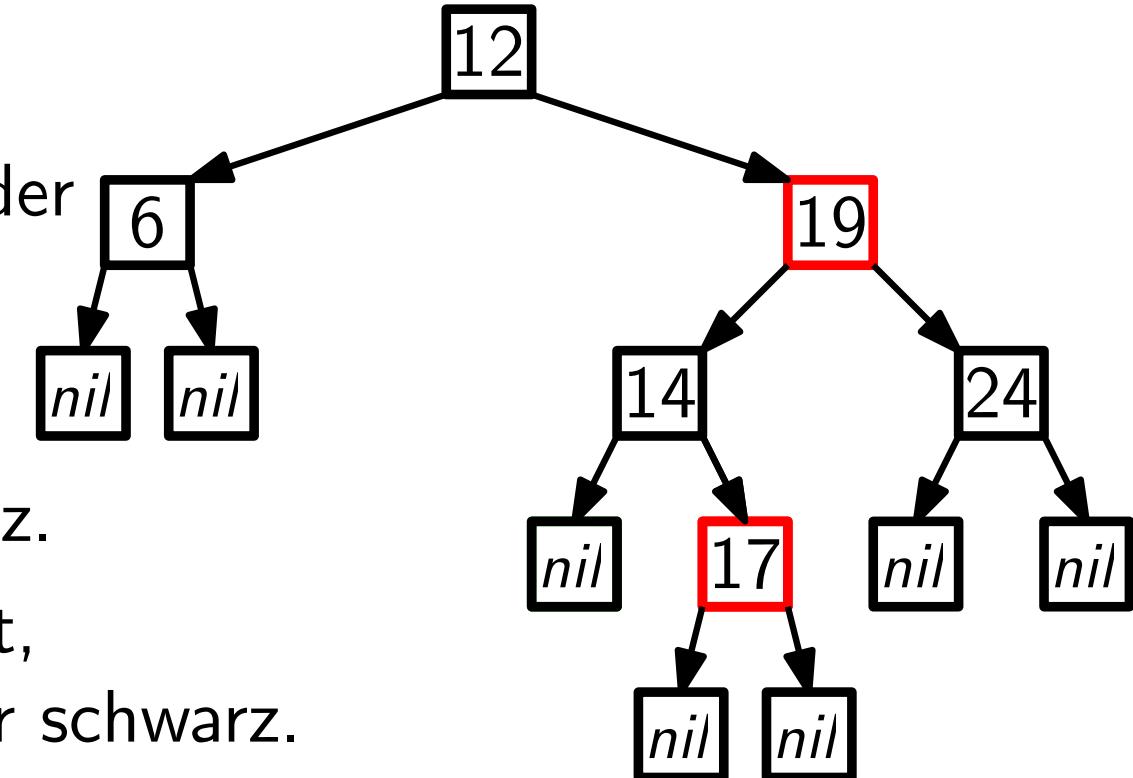
jeder Knoten ist entw. „gut“ oder „schlecht“; der Anteil schlechter Knoten darf in keinem Teilbaum zu groß sein.

Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Aus (E4) folgt: Auf keinem Wurzel-Blatt-Pfad folgen zwei rote Knoten direkt aufeinander.



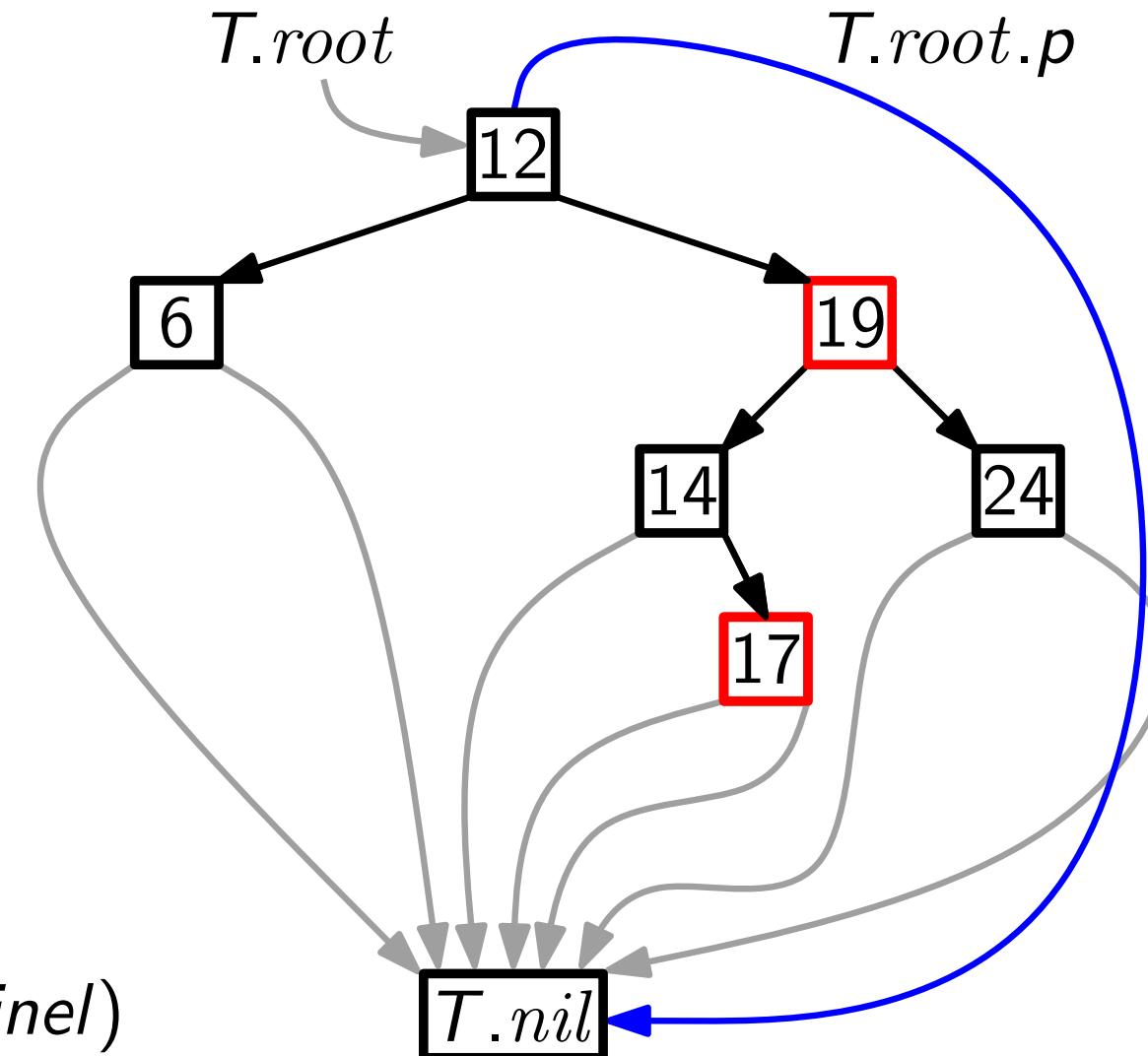
Technisches Detail

<i>Node</i>
<i>Key key</i>
<i>Node left</i>
<i>Node right</i>
<i>Node p</i>

<i>RBNode</i>
<i>Color color</i>

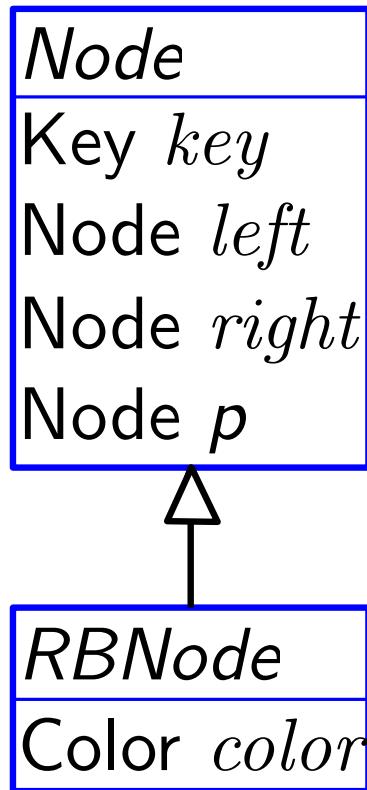
$T.root$, $T.nil$

Dummy-Knoten (engl. *sentinel*)

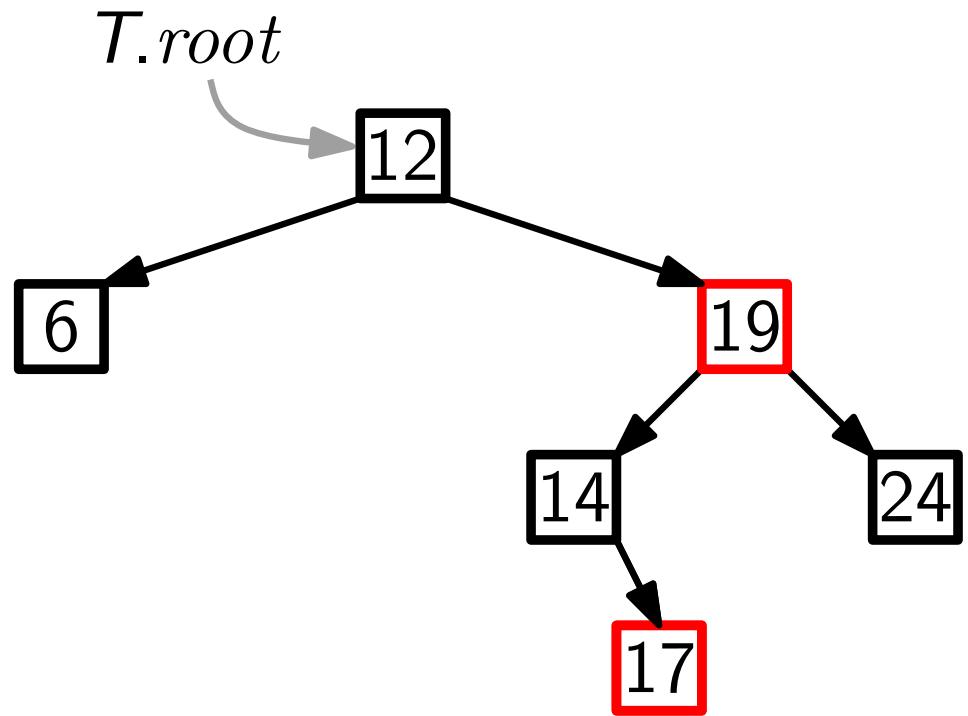


Zweck: um Baum-Operationen prägnanter aufzuschreiben zu können. (Wir zeichnen den Dummy-Knoten i.A. nicht.)

Technisches Detail

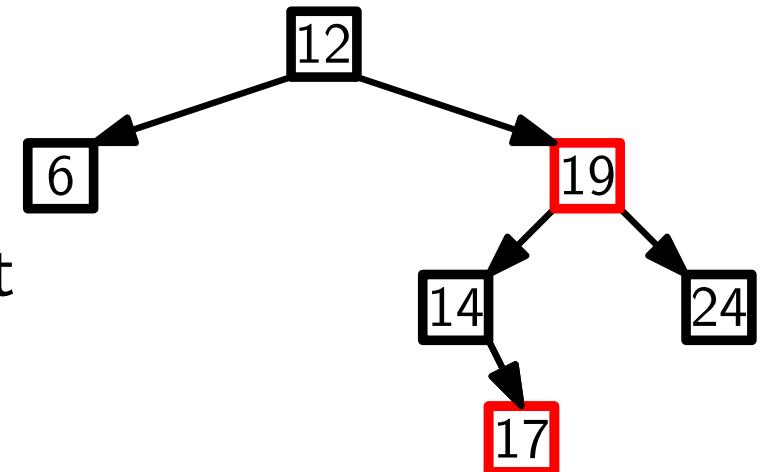


T.root, T.nil



Zweck: um Baum-Operationen prägnanter aufzuschreiben zu können. (Wir zeichnen den Dummy-Knoten i.A. nicht.)

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

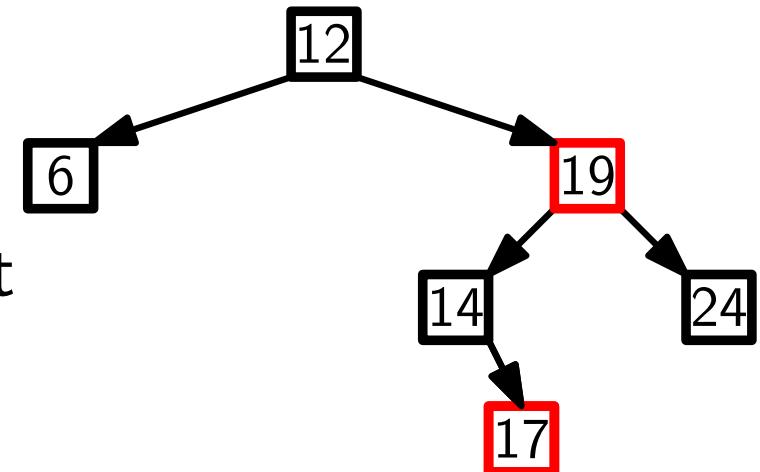
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition! Die *Höhe* $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten (ohne v) auf dem längsten Pfad zu einem Blatt (inkl. Blatt) in B_v .

Beispiel: „12“ hat Höhe

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition: Die **Schwarz-Höhe** $s\text{Höhe}(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** **längsten** Pfad zu einem Blatt (inkl. Blatt) in B_v .

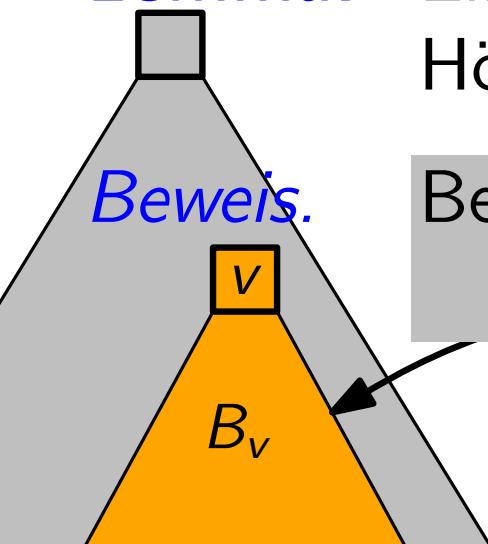
wohl-definiert
wg. (E5)

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: \forall Knoten $\Rightarrow s\text{Höhe}(v) \leq \text{Höhe}(v) \leq 2 \cdot s\text{Höhe}(v)$.

Höhe $\in \Theta(\log n)!!$

Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.



Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

Höhe $\in \Theta(\log n)!!$

Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis. Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{s\text{Höhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über Höhe(v).

Höhe(v) = 0. Dann $B_v = B.\text{nil}$ und $s\text{Höhe}(v) = 0$.
 B_v hat $0 = 2^0 - 1$ innere Knoten. 

Höhe(v) > 0. Beide Kinder von v haben Höhe < Höhe(v).
 \Rightarrow können Ind.-Annahme anwenden.

\Rightarrow # innere Knoten von B_v ist mind.

$$2 \cdot (2^{s\text{Höhe}(v)-1} - 1) + 1 = 2^{s\text{Höhe}(v)} - 1.$$

$s\text{Höhe}$ der Kinder von v ist mind.

 Anz. innerer Knoten unter
einem Kind von v 

Höhe $\in \Theta(\log n)!!$

Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis. Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v := B.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(B)}_n \geq 2^{\text{sHöhe}(B)} - 1.$$

$$\Rightarrow \text{sHöhe}(B) \leq \log_2(n + 1)$$

Wegen R-S-Eig. (E4) gilt: Höhe(B) $\leq 2 \cdot \text{sHöhe}(B)$.

$$\Rightarrow \text{Höhe}(B) \leq 2 \log_2(n + 1) \quad \square$$

Also: Rot-Schwarz-Bäume sind *balanciert!* Fertig?!
 Nee: Insert & Delete können R-S-Eig. *verletzen!*

Einfügen

Node Insert(key k)

$y = \text{nil}$

$x = \text{root}$

while $x \neq \text{nil}$ **do**

$y = x$

if $k < x.\text{key}$ **then**

$x = x.\text{left}$

else $x = x.\text{right}$

$z = \text{new Node}(k, y)$

if $y == \text{nil}$ **then** $\text{root} = z$

else

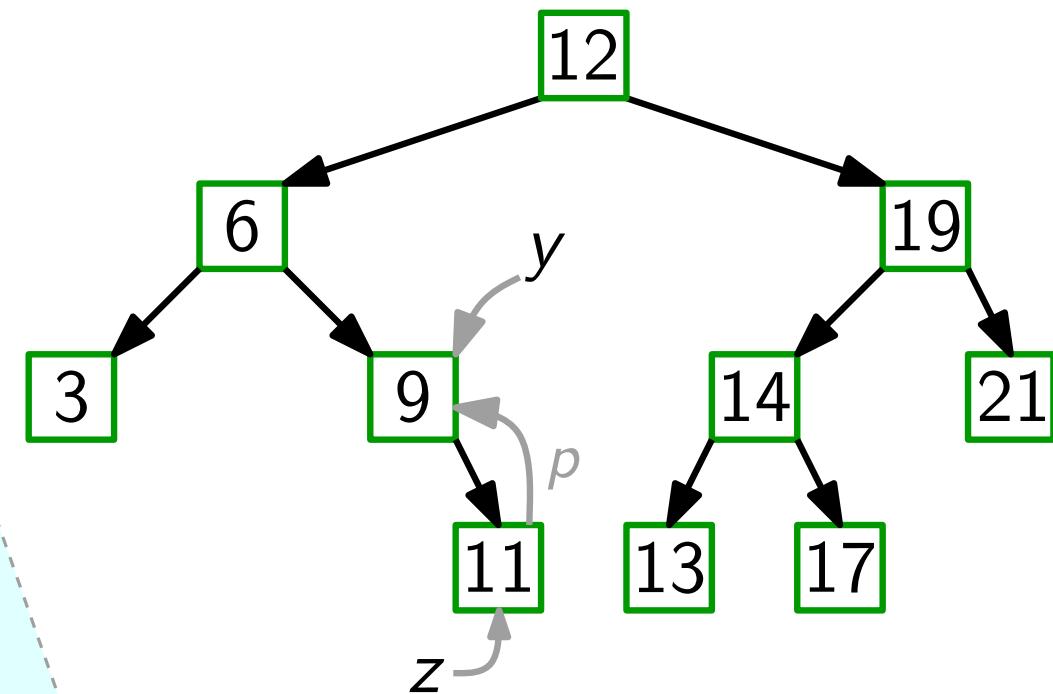
if $k < y.\text{key}$ **then** $y.\text{left} = z$

else

return z

Insert(11)

$x == \text{nil}$



Node(Key k, Node par)
 $\text{key} = k$
 $p = \text{par}$
 $\text{right} = \text{left} = \text{nil}$

Einfügen

Laufzeit? (ohne RBInsertFixup) $O(h) = O(\log n)$

RB .. RB

Node Insert(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

RB

$z = \text{new Node}(k, y, red)$

if $y == T.nil$ **then** $root = z$

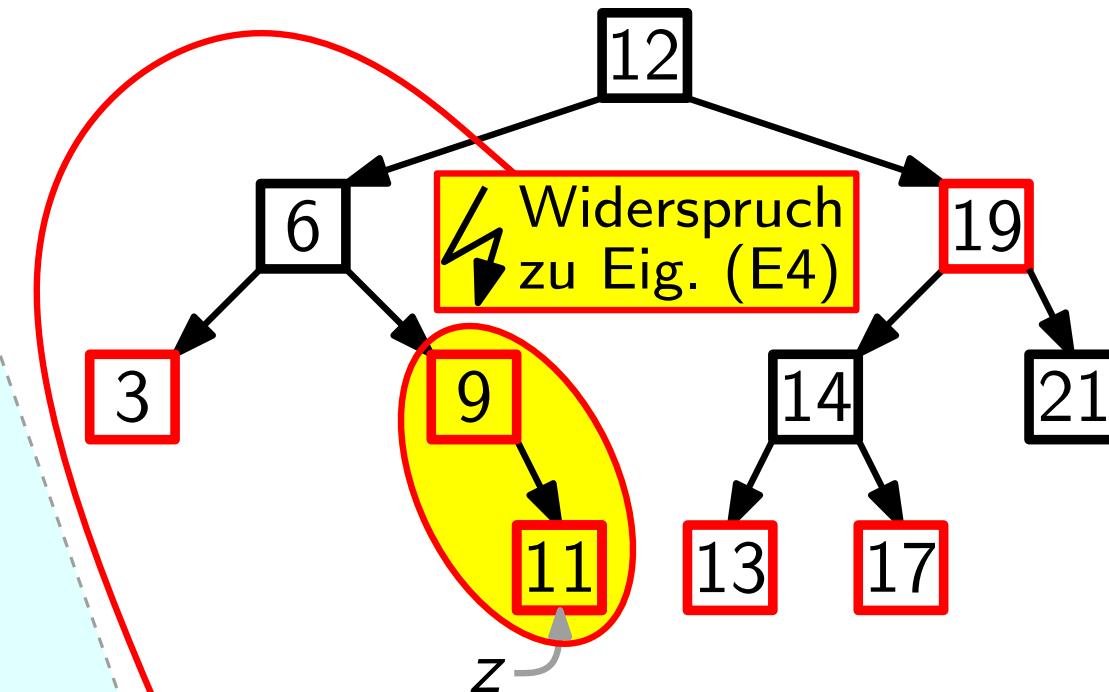
else

if $k < y.key$ **then** $y.left = z$

else

RBInsertFixup(z)

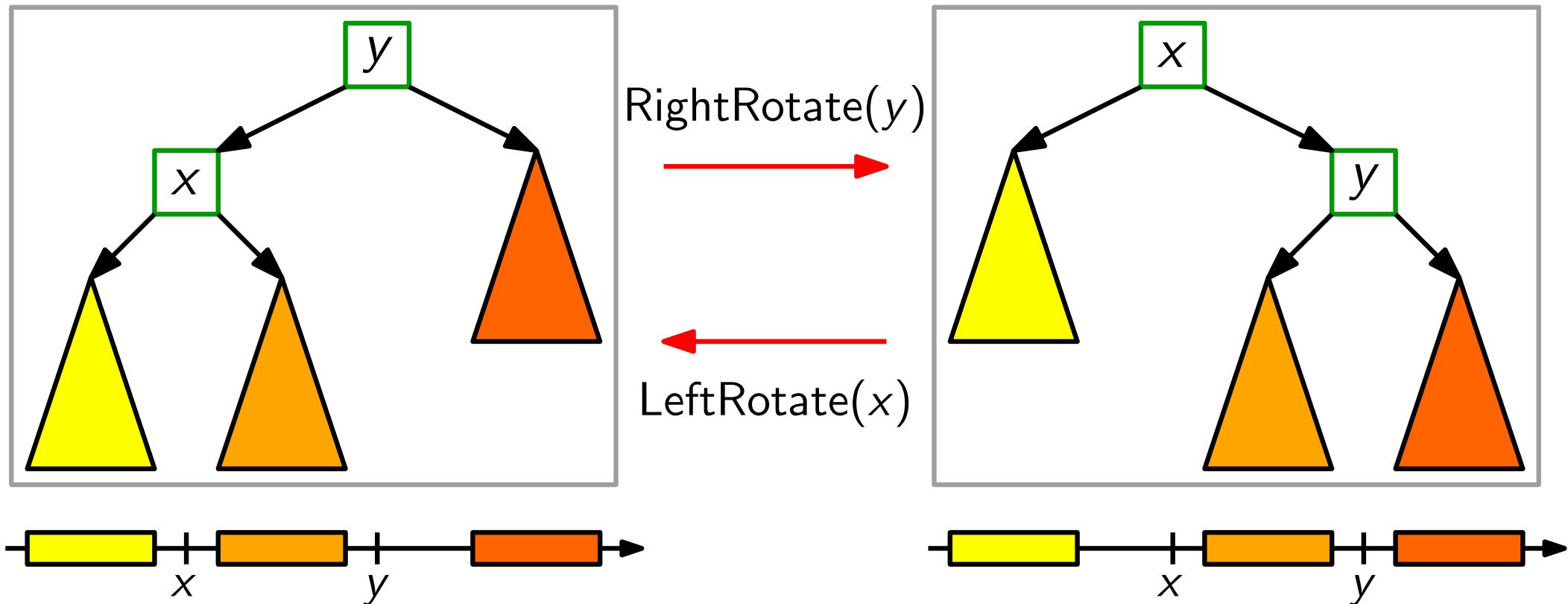
return z



Node(Key k, Node par)
 $key = k$
 $p = par$
 $right = left = T.nil$

RBNode(..., Color c)
 $\text{super}(k, par)$
 $color = c$

Exkurs: Rotationen



Also: Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

Aufgabe: Schreiben Sie Pseudocode für LeftRotate(x)!

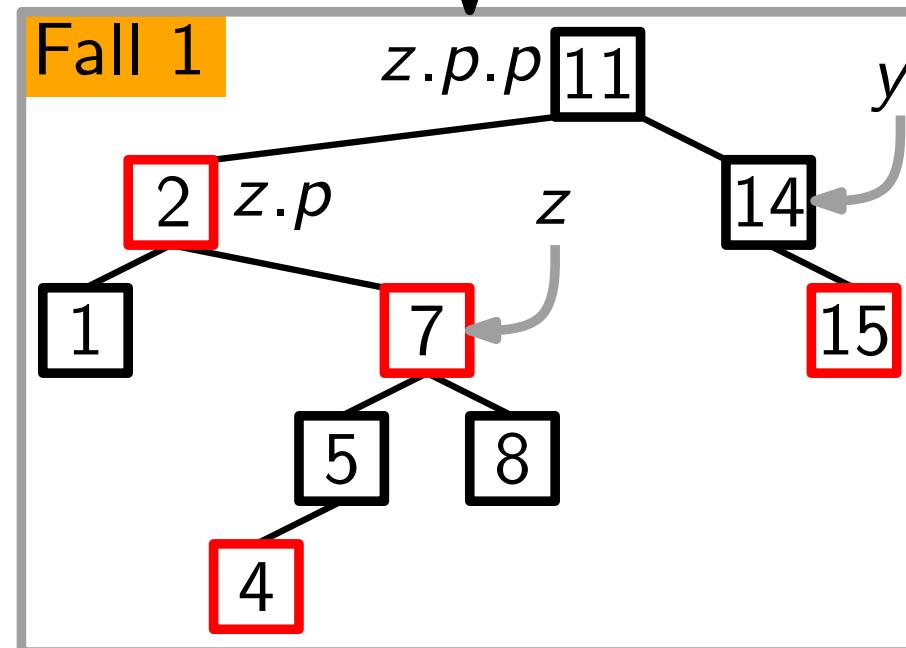
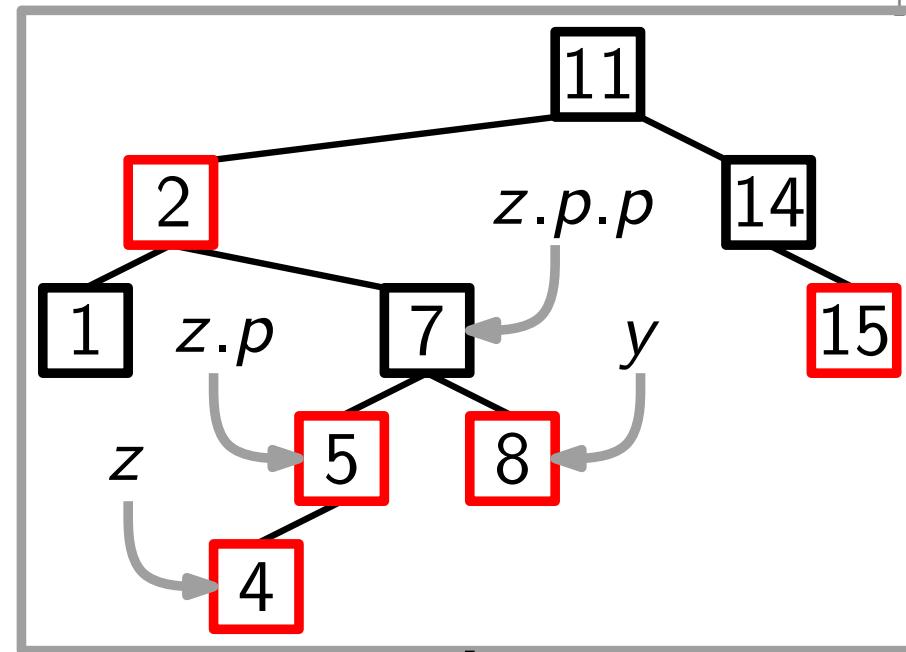
Laufzeit: $O(1)$.



RBIInsertFixup(Node z)

```

while  $z.p.color == \text{red}$  do
    if  $z.p == z.p.p.left$  then
         $y = z.p.p.right$  // Tante von z
        if  $y.color == \text{red}$  then
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
             $y.color = \text{black}$ 
             $z = z.p.p$ 
        else
            if  $z == z.p.right$  then
                 $z = z.p$ 
                LeftRotate( $z$ )
                 $z.p.color = \text{black}$ 
                 $z.p.p.color = \text{red}$ 
                RightRotate( $z.p.p$ )
            else ... // wie oben, aber re. & li. vertauscht
             $root.color = \text{black}$ 
    
```



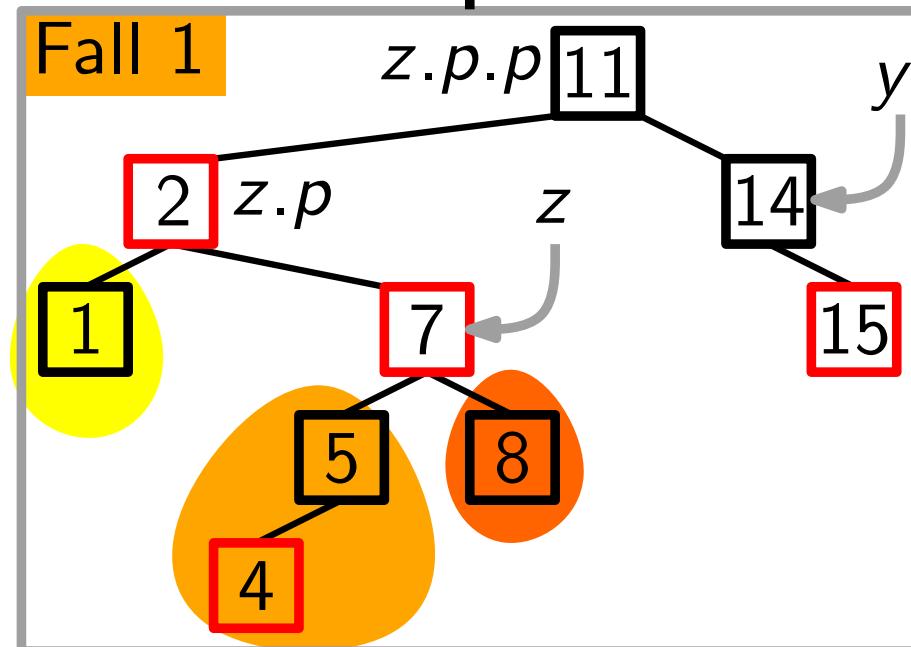
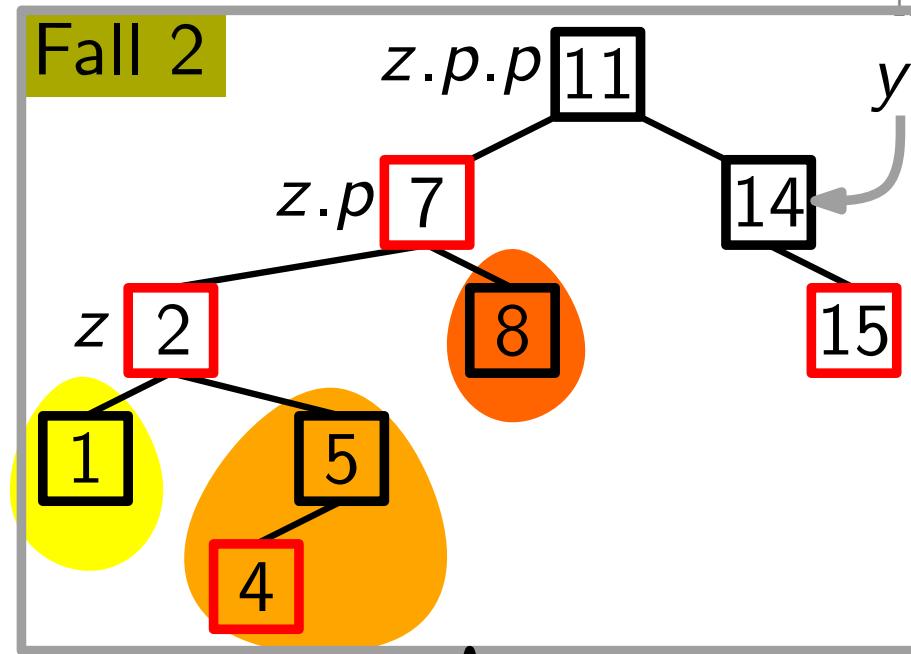
li. vertauscht

RBIInsertFixup(Node z)

```

while  $z.p.color == \text{red}$  do
    if  $z.p == z.p.p.left$  then
         $y = z.p.p.right$  // Tante von z
        if  $y.color == \text{red}$  then
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
             $y.color = \text{black}$ 
             $z = z.p.p$ 
        else
            if  $z == z.p.right$  then
                 $z = z.p$ 
                LeftRotate( $z$ )
                 $z.p.color = \text{black}$ 
                 $z.p.p.color = \text{red}$ 
                RightRotate( $z.p.p$ )
            else ... // wie oben, aber re. & li. vertauscht
     $root.color = \text{black}$ 

```



li. vertauscht

RBIInsertFixup(Node z)

```
while  $z.p.color == \text{red}$  do
```

```
    if  $z.p == z.p.p.left$  then
```

```
         $y = z.p.p.right$  // Tante von z
```

```
        if  $y.color == \text{red}$  then
```

```
             $z.p.color = \text{black}$ 
```

```
             $z.p.p.color = \text{red}$ 
```

```
             $y.color = \text{black}$ 
```

```
             $z = z.p.p$ 
```

```
        else
```

```
            if  $z == z.p.right$  then
```

```
                 $z = z.p$ 
```

```
                LeftRotate( $z$ )
```

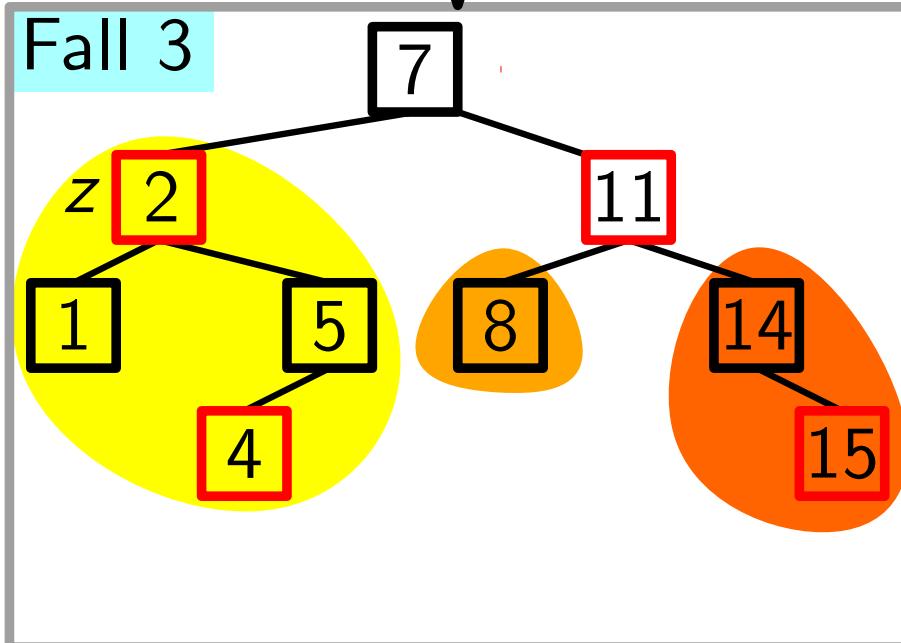
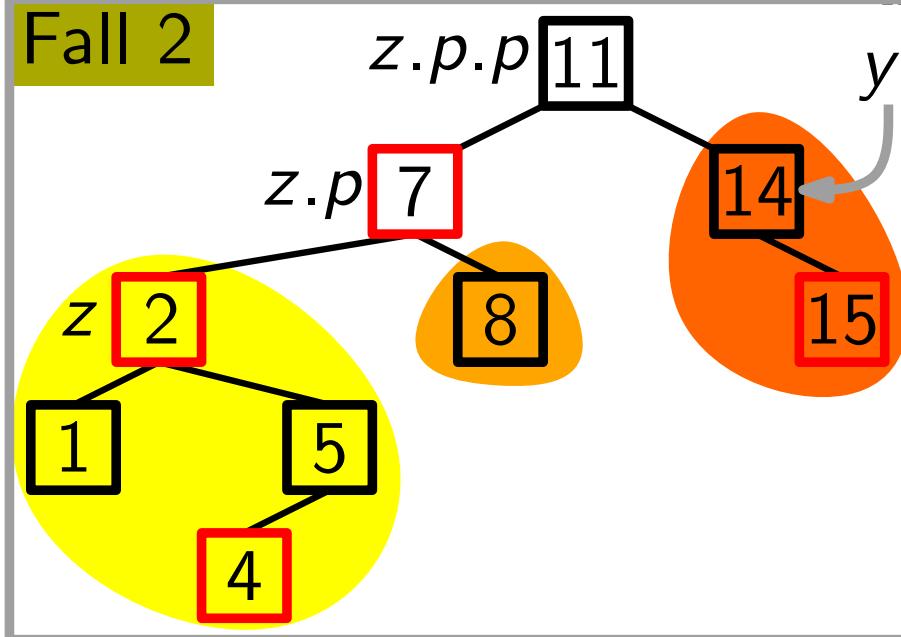
```
                 $z.p.color = \text{black}$ 
```

```
                 $z.p.p.color = \text{red}$ 
```

```
                RightRotate( $z.p.p$ )
```

```
        else ... // wie oben, aber re. &
```

```
root.color = black
```



// wie oben, aber re. & li. vertauscht

Korrektheit

Zu zeigen: RBInsertFixup stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder (E2) oder (E4).
 - Falls (E2) verletzt ist, dann weil $z = \text{root}$ und z rot ist.
 - Falls (E4) verletzt ist, dann weil z und $z.p$ rot sind.

Zeige:

- Initialisierung
- Aufrechterhaltung
- Terminierung

Viel Arbeit! Siehe [CLRS, Kapitel 13.3].

Laufzeit RBInsertFixup

```

while  $z.p.color == \text{red}$  do
    if  $z.p == z.p.p.left$  then
         $y = z.p.p.right$ 
        if  $y.color == \text{red}$  then
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
             $y.color = \text{black}$ 
             $z = z.p.p$ 
        }  $O(1)$ 
    else
        if  $z == z.p.right$  then
             $z = z.p$ 
            LeftRotate( $z$ )
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
            RightRotate( $z.p.p$ )
        }  $O(1)$ 
    }  $O(1)$ 
else ... // wie oben, aber re. & li. vertauscht
root.color = black

```

Insgesamt:

- Fall 1 $O(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

$O(\log n)$ Umfärbungen und ≤ 2 Rotationen

Klettert Baum zwei Ebenen nach oben.

Führt zum Abbruch der while-Schleife.

Löschen in (farblosen) binären Suchbäumen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

- 1.** z hat kein li. Kind.

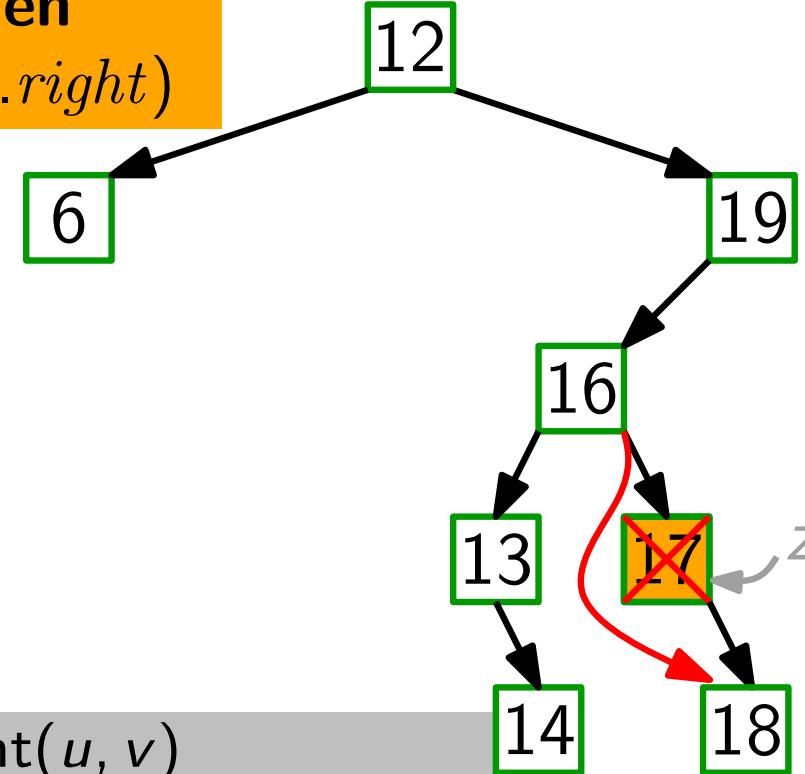
```
if  $z.left == \text{nil}$  then  
    Transplant( $z, z.right$ )
```

Setze $z.right$ an die Stelle von z .

Lösche z .

- 2.** z hat kein re. Kind.

- 3.** z hat zwei Kinder.



Transplant(u, v)

```
if  $u.p == \text{nil}$  then  $\text{root} = v$   
else  
    if  $u == u.p.left$  then  
         $u.p.left = v$   
    else  $u.p.right = v$ 
```

if $v \neq \text{nil}$ then $v.p = u.p$

Setze v
an die
Stelle
von u .

Löschen in (farblosen) binären Suchbäumen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind.

```
if  $z.left == nil$  then  
    Transplant( $z, z.right$ )
```

Setze $z.right$ an die Stelle von z .

Lösche z .

2. z hat kein re. Kind.

```
else if  $z.right == nil$  then  
    Transplant( $z, z.left$ )
```

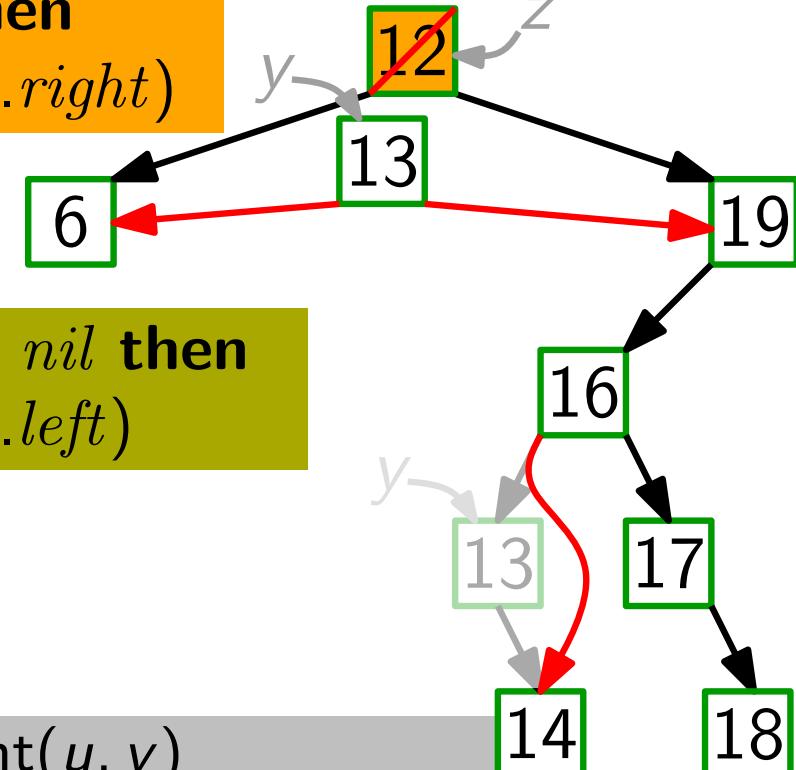
symmetrisch!

3. z hat zwei Kinder.

Setze $y = \text{Successor}(z)$

Falls $y.p \neq z$, setze $y.right$ an die Stelle von y .

Setze y an die Stelle von z



Transplant(u, v)

```
if  $u.p == nil$  then  $root = v$   
else  
    if  $u == u.p.left$  then  
         $u.p.left = v$   
    else  $u.p.right = v$ 
```

```
if  $v \neq nil$  then  $v.p = u.p$ 
```

Setze v an die Stelle von u .

Löschen (Übersicht)

Delete(Node z)

```
if  $z.left == nil$  then // kein linkes Kind
```

```
  | Transplant( $z, z.right$ )
```

```
else
```

```
  if  $z.right == nil$  then // kein rechtes Kind
```

```
    | Transplant( $z, z.left$ )
```

```
  else // zwei Kinder
```

```
     $y = \text{Successor}(z)$ 
```

```
    if  $y.p \neq z$  then
```

```
      | Transplant( $y, y.right$ )
```

```
      |  $y.right = z.right$ 
```

```
      | |  $y.right.p = y$ 
```

```
      | Transplant( $z, y$ )
```

```
      |  $y.left = z.left$ 
```

```
      | |  $y.left.p = y$ 
```

```

RBDelete(Node z)
y = z; origcolor = y.color
if z.left == T.nil then
    x = z.right
    RBTransplant(z, z.right)
else
    if z.right == T.nil then
        x = z.left
        RBTransplant(z, z.left)
    else
        y = Successor(z)
        origcolor = y.color
        x = y.right
        if y.p == z then x.p = y
        else
            RBTransplant(y, y.right)
            y.right = z.right
            y.right.p = y
        RBTransplant(z, y)
        y.left = z.left
        y.left.p = y; y.color = z.color
    if origcolor == black then RBDeleteFixup(x)

```

- y** zeigt auf den Knoten, der entweder gelöscht oder verschoben wird.
- x** zeigt auf den Knoten, der die Stelle von **y** einnimmt – das ist entweder das einzige Kind von **y** oder $T.nil$.
- Falls **y** ursprünglich *rot* war, bleiben alle R-S-Eig. erhalten:
- Keine Schwarzhöhe hat sich verändert.
 - Keine zwei roten Knoten sind Nachbarn geworden.
 - y rot $\Rightarrow y \neq$ Wurzel \Rightarrow Wurzel bleibt schwarz.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten x zählt eine schwarze Einheit extra
(E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

Ziel: Schiebe die überzählige schwarze Einheit nach oben, bis:

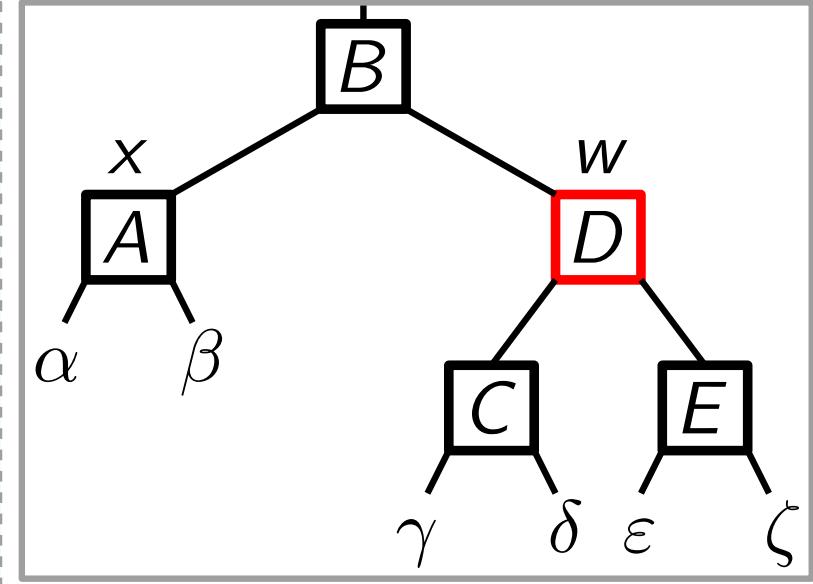
- x ist rot-schwarz \Rightarrow mach x schwarz.
- x ist Wurzel \Rightarrow schwarze Extra-Einheit verfällt.
- Problem wird lokal durch Umfärben & Rotieren gelöst.

RBDeleteFixup(RBNode x)

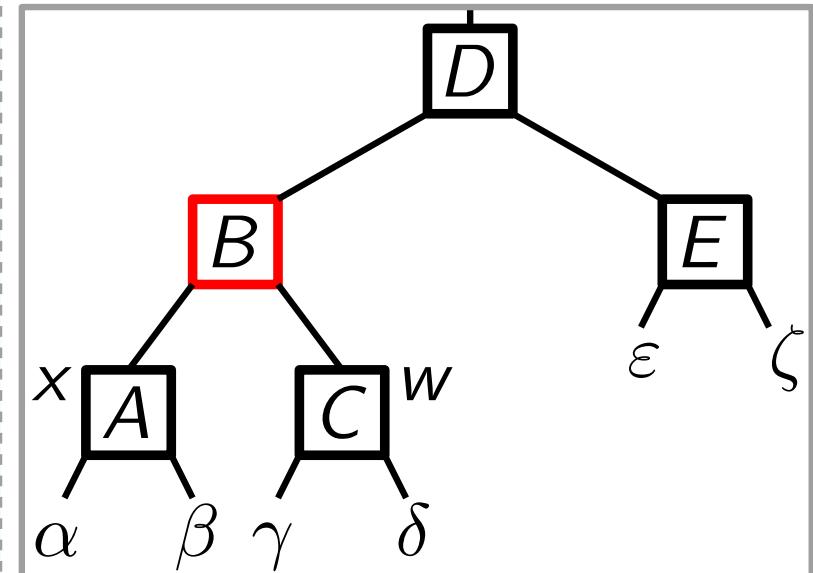
```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
    if  $x == x.p.\text{left}$  then
         $w = x.p.\text{right}$  // Schwester von  $x$ 
        if  $w.\text{color} == \text{red}$  then
             $w.\text{color} = \text{black}$ 
             $x.p.\text{color} = \text{red}$ 
            LeftRotate( $x.p$ )
             $w = x.p.\text{right}$ 
        if  $w.\text{left}.\text{color} == \text{black}$  and
             $w.\text{right}.\text{color} == \text{black}$  then
                 $w.\text{color} = \text{red}$ 
                 $x = x.p$ 
        else // kommt gleich!!
        else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
         $x.\text{color} = \text{black}$ 
    
```

Ziel:
 $w \rightarrow$ schwarz
ohne R-S-Eig.
zu verletzen.



Fall 1



RBDeleteFixup(RBNode x)

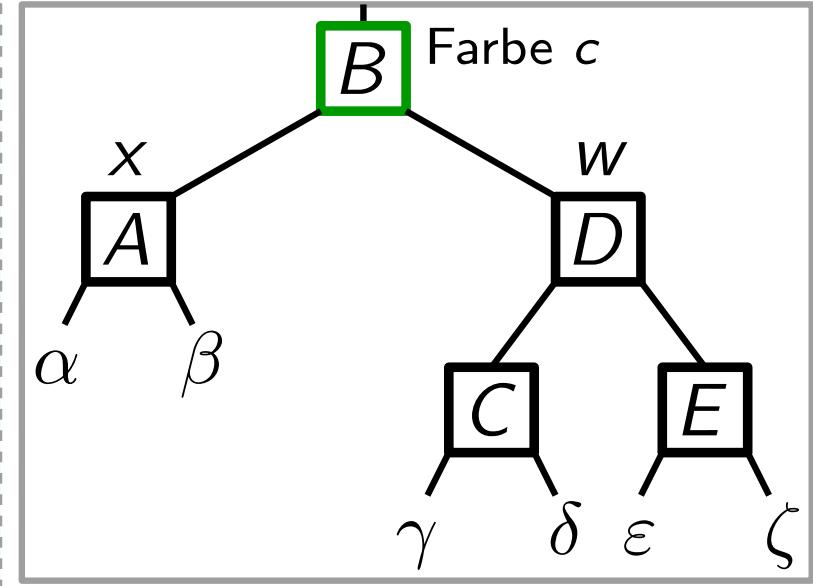
```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
    if  $x == x.p.\text{left}$  then
         $w = x.p.\text{right}$  // Schwester von x
        if  $w.\text{color} == \text{red}$  then
             $w.\text{color} = \text{black}$ 
             $x.p.\text{color} = \text{red}$ 
            LeftRotate( $x.p$ )
             $w = x.p.\text{right}$ 
        if  $w.\text{left.color} == \text{black}$  and
             $w.\text{right.color} == \text{black}$  then
                 $w.\text{color} = \text{red}$ 
                 $x = x.p$ 
        else // kommt gleich!!
    else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
 $x.\text{color} = \text{black}$ 

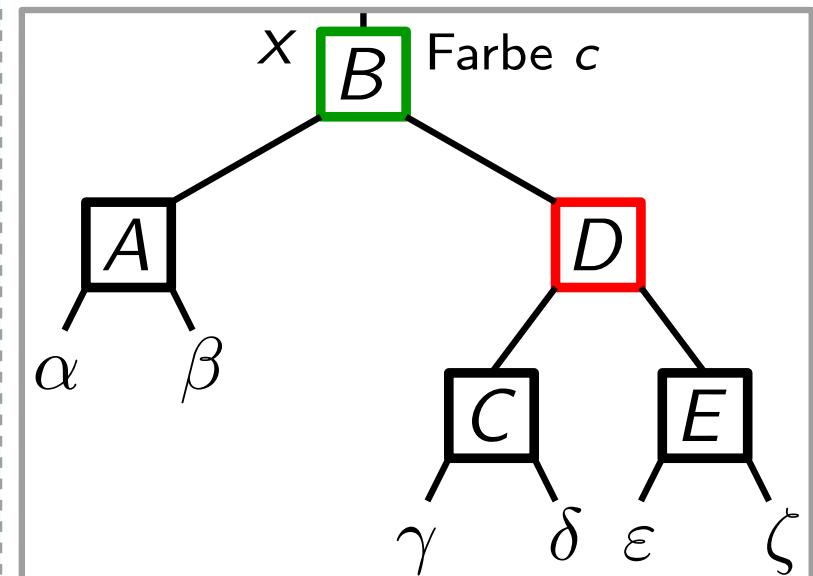
```

Ziel:
 $w \rightarrow$ schwarz
ohne R-S-Eig.
zu verletzen.

Schw. Einheit
raufschreiben.



Fall 2



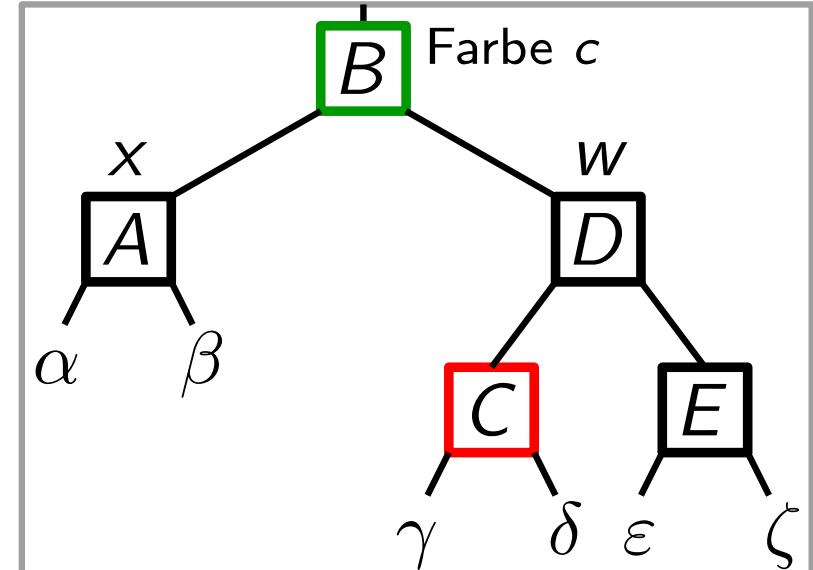
Bem.: Anz. der schw. Knoten (inkl. Extra-Einh. bei x) bleibt auf allen Pfaden gleich!

RBDeleteFixup (Forts.)

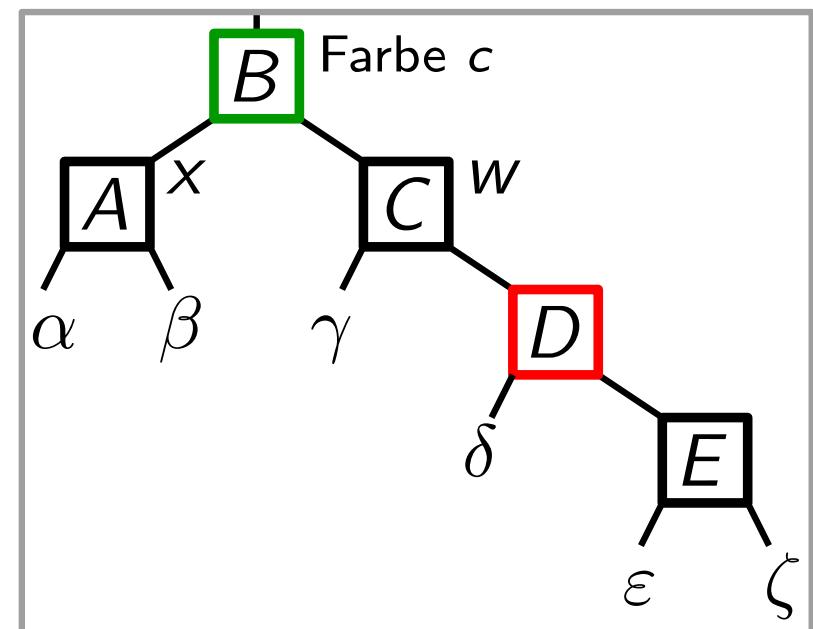
else

```
if w.right.color == black then
    w.left.color = black
    w.color = red
    RightRotate(w)
    w = x.p.right
```

```
w.color = x.p.color
x.p.color = black
w.right.color = black
LeftRotate(x.p)
x = root
```



Fall 3



RBDeleteFixup (Forts.)

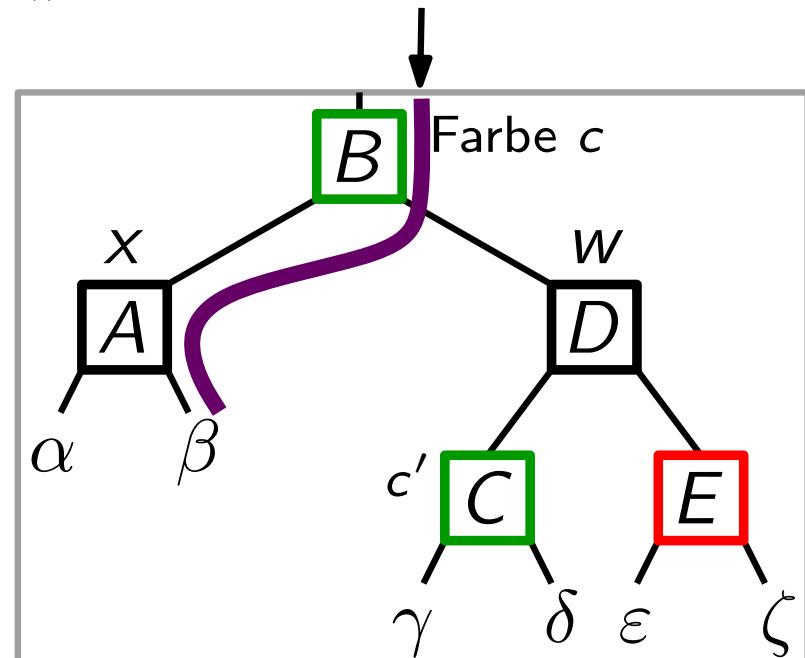
else

```
if w.right.color == black then
    w.left.color = black
    w.color = red
    RightRotate(w)
    w = x.p.right
```

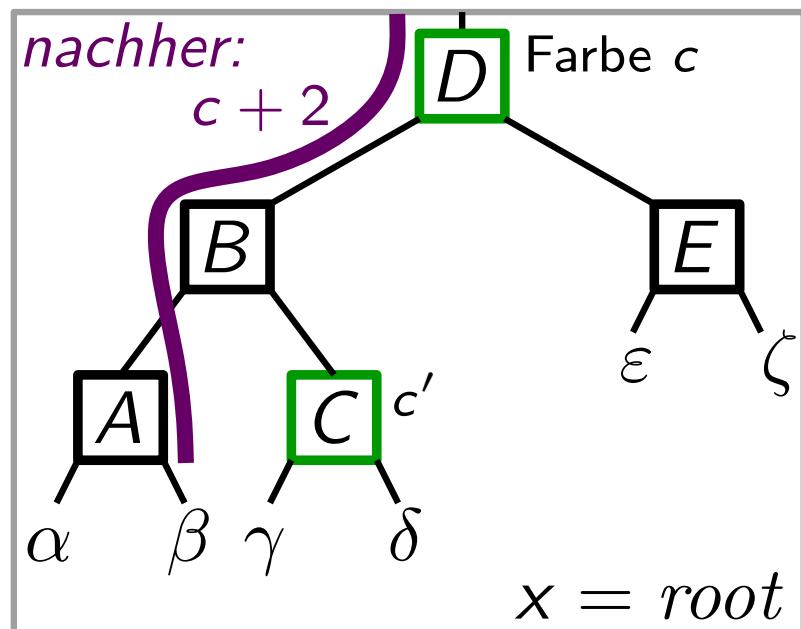
```
w.color = x.p.color
x.p.color = black
w.right.color = black
LeftRotate(x.p)
x = root
```

Bem.: Anz. der schwarzen Knoten
 (inkl. der Extra-Einheit bei x)
 bleibt auf allen Pfaden gleich!

vorher:
 # schwarze Einheiten = $c + 2$



Fall 4



RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

$w.color = x.p.color$

$x.p.color = black$

$w.right.color = black$

LeftRotate($x.p$)

$x = root$

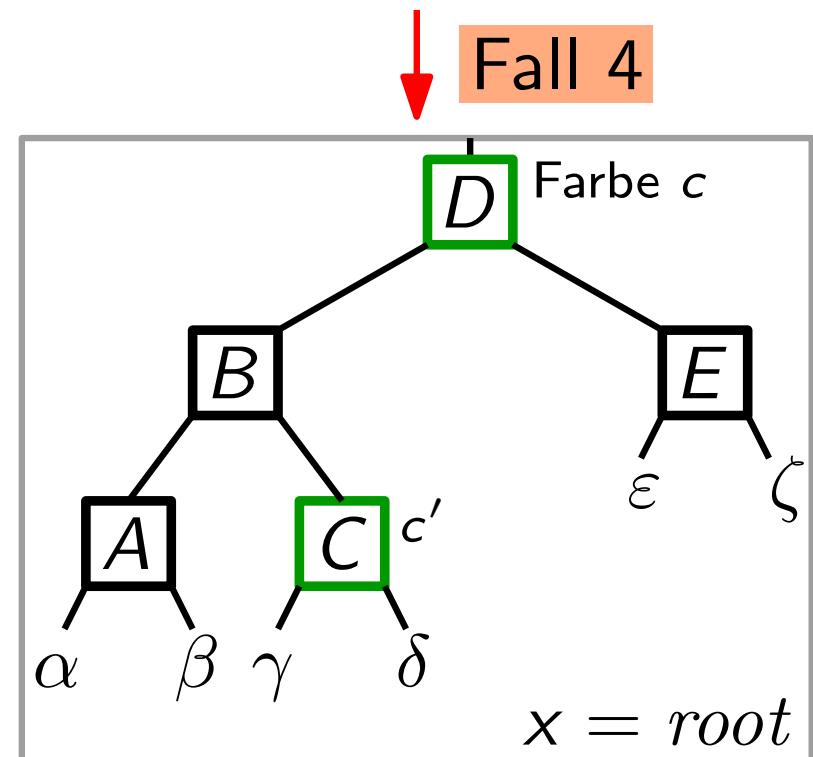
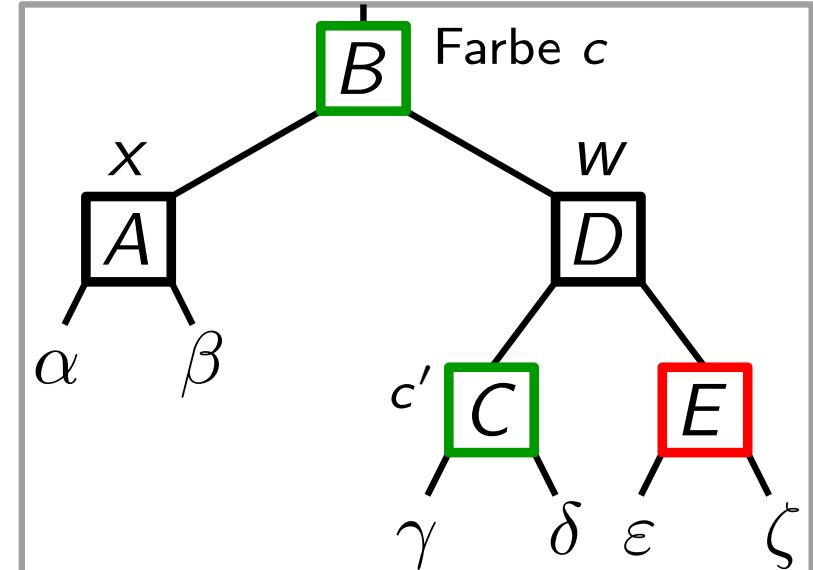
Laufzeit?

Fall 1: 1 Rotation + $O(1)$

Fall 2: $O(h)$ Umfärbungen

Fall 3: 1 Rotation + $O(1)$

Fall 4: 1 Rotation + $O(1)$



Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$



Laufzeit RBDelete $\in O(\log n)$

Satz.

Rot-Schwarz-Bäume implementieren alle dynamische-Menge-Operationen in $O(\log n)$ Zeit, wobei n die momentane Anz. der Schlüssel ist.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
15. Vorlesung

Augmentieren von Datenstrukturen

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel
- Hashtabelle
- Heap
- binärer Suchbaum

Allerdings gibt es viele Situationen, wo keine davon **genau** passt.

Herangehensweise: *Augmentieren* von Datenstrukturen,
d.h. wir verändern Datenstrukturen,
indem wir extra Information
hinzufügen und aufrechterhalten.

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
 - Liste
2. Welche Extrainformation aufrechterhalten?
 - Summe der Elemente (*sum*)
 - Anzahl der Elemente (*size*)
3. Aufwand zur Aufrechterhaltung der Extrainformation?
 - konstanter Aufwand beim Einfügen und Löschen
4. Implementiere neue Operationen!

`getMean()`

`return sum/size`

Ähnlich für Standardabweichung $\sqrt{\frac{1}{n} \sum_{i=1}^n (a_i - \bar{a})^2}$.

Probieren Sie's!

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

D.h. wir wollen zu jedem Zeitpunkt effizient

- das i -kleinste Element (z.B. den Median): Elem `Select(int i)`
- den *Rang* eines Elements: int `Rank(Elem e)`

in einer dynamischen Menge bestimmen können.

Fahrplan: 1. Welche Ausgangsdatenstruktur?

2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung?

4. Implementiere neue Operationen!

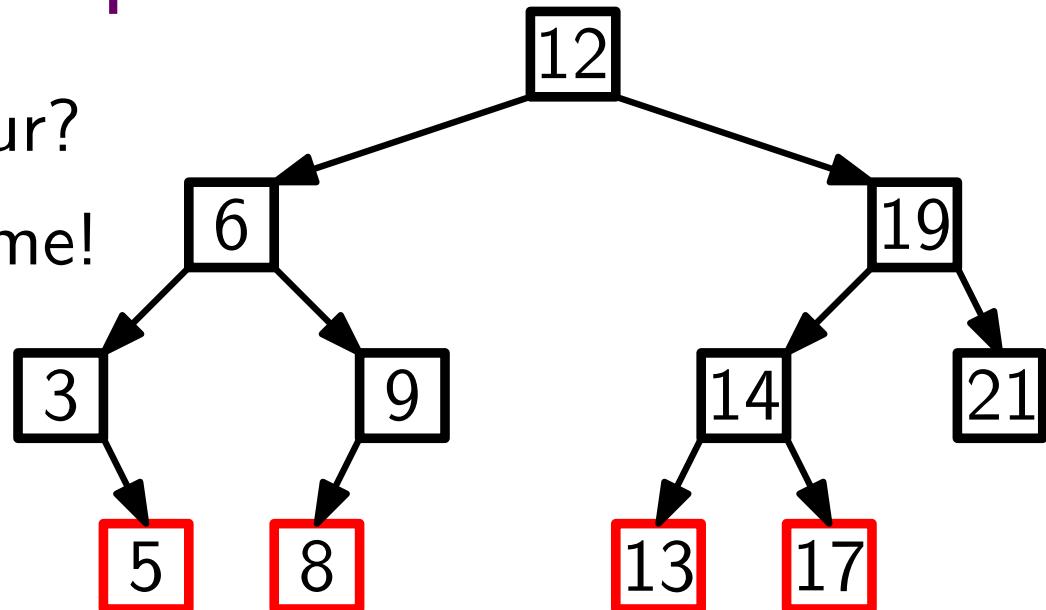
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere neue Operationen!

Select(int i):

```

 $v = \text{Minimum}()$ 
while  $v \neq \text{nil}$  and  $i > 1$  do
|    $v = \text{Successor}(v)$ 
|    $i = i - 1$ 
return  $v$ 
  
```

Rank(Node v):

```

 $j = 0$ 
while  $v \neq \text{nil}$  do
|    $v = \text{Predecessor}(v)$ 
|    $j = j + 1$ 
return  $j$ 
  
```

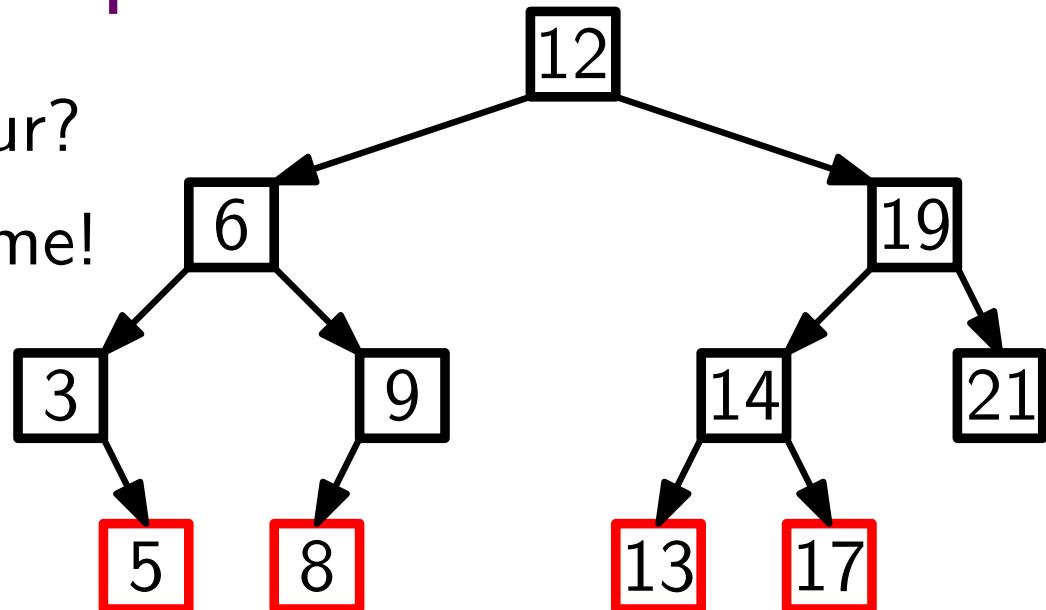
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich?

Select(int i): $O(i \cdot h)$

```

 $v = \text{Minimum}()$ 
while  $v \neq \text{nil}$  and  $i > 1$  do
|    $v = \text{Successor}(v)$ 
|    $i = i - 1$ 
|
return  $v$ 
  
```

Rank(Node v): $O(\text{rank} \cdot h)$

```

 $j = 0$ 
while  $v \neq \text{nil}$  do
|    $v = \text{Predecessor}(v)$ 
|    $j = j + 1$ 
|
return  $j$ 
  
```

Das dynamische Auswahlproblem

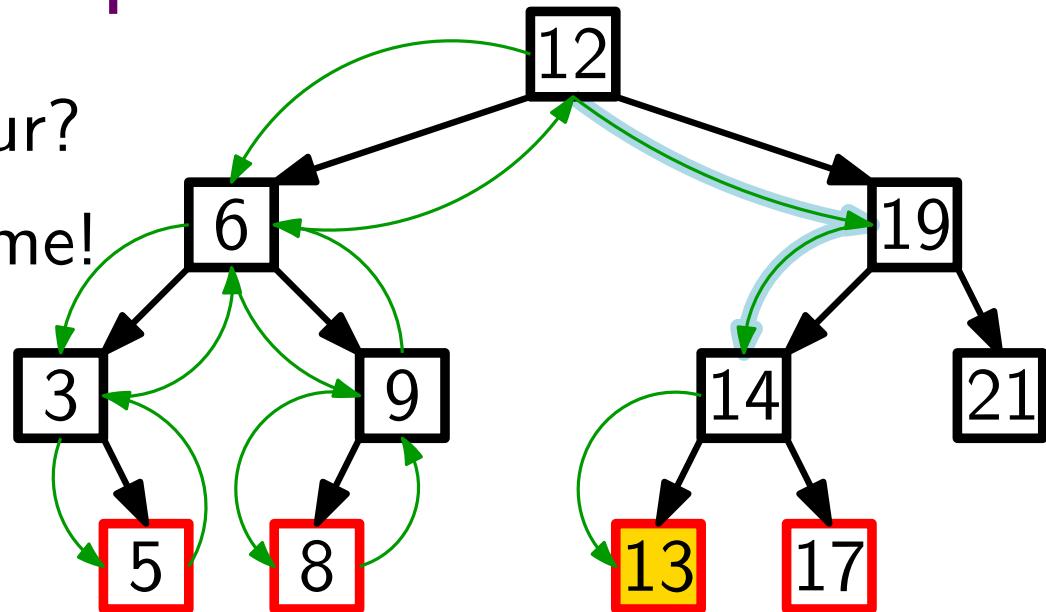
Select(7)

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

Select(int i): $O(\boxed{i} + h)$

```

 $v = \text{Minimum}()$ 
while  $v \neq \text{nil}$  and  $i > 1$  do
     $v = \text{Successor}(v)$ 
     $i = i - 1$ 
return  $v$ 
  
```

Rank(Node v): $O(\text{rank } v + h)$

```

 $j = 0$ 
while  $v \neq \text{nil}$  do
     $v = \text{Predecessor}(v)$ 
     $j = j + 1$ 
return  $j$ 
  
```

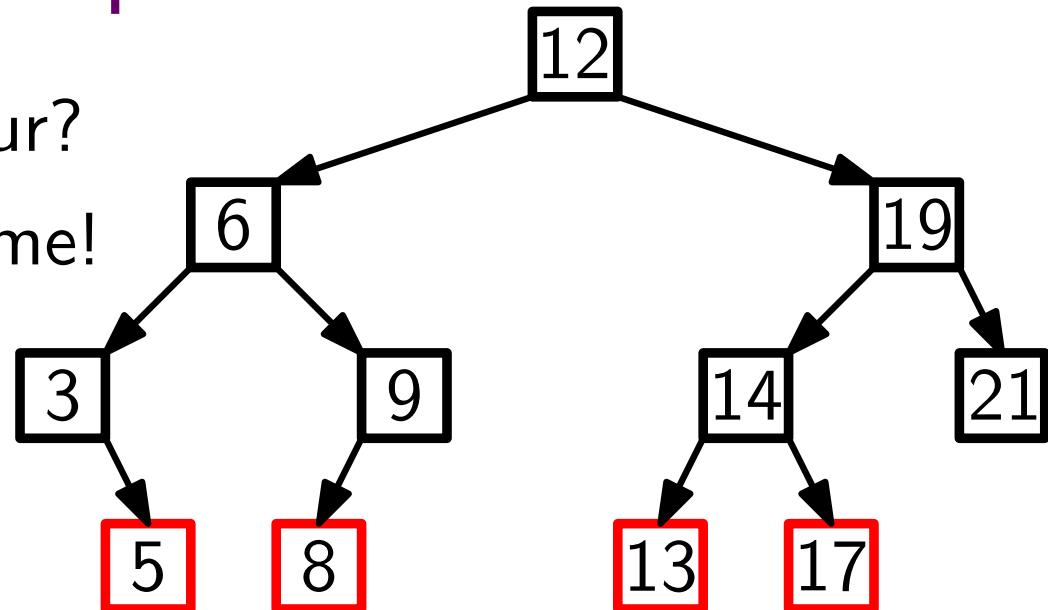
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

Select(int i): $O(i + h)$

Rank(Node v): $O(rank + h)$

Problem: Wenn $i \in \Theta(n)$ – z.B. beim Median –, dann ist die Laufzeit linear (wie im statischen Fall!).



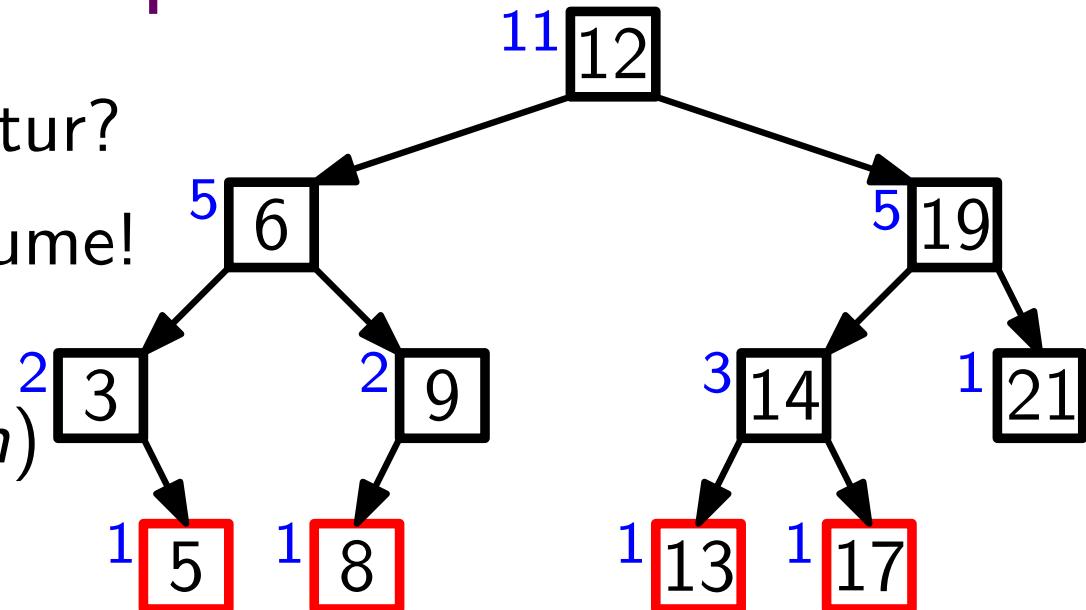
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume: für jeden Knoten v , speichere $v.size$

4. **Select**(Node $v = root$, int i): $O(h)$

```

r = v.left.size + 1
if i == r then return v
else
  if i < r then
    | return Select(v.left, i)
  else
    | return Select(v.right, i - r)
  
```

Rank(Node v):

```

r = v.left.size + 1
u = v
while u != root do
  if u == u.p.right then
    | r = r + u.p.left.size + 1
  u = u.p
return r
  
```

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife —
ist r der Rang von v im Teilbaum mit Wurzel u .

1.) Initialisierung

Vor 1. Iteration gilt $u = v \Rightarrow u\text{-Rang}(v) = v.left.size + 1$. ✓

Rank(Node v):

```

 $r = v.left.size + 1$ 
 $u = v$ 
while  $u \neq root$  do
    if  $u == u.p.right$  then
         $r = r + u.p.left.size + 1$ 
     $u = u.p$ 
return  $r$  (vorausgesetzt, dass  $T.nil.size = 0$ )

```

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife — ist r der Rang von v im Teilbaum mit Wurzel u .

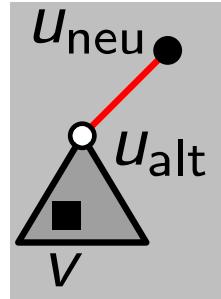
1.) Initialisierung ✓

u-Rang von v

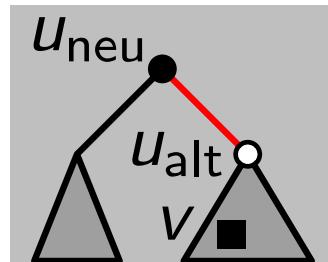
2.) Aufrechterhaltung ✓

Annahme: Inv. galt zu Beginn der aktuellen Iteration.

Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.
⇒ u -Rang von v bleibt gleich.



2. Fall: u war rechtes Kind.
⇒ u -Rang von v erhöht sich um Größe des li. Teilbaums von u plus 1 (für u selbst).

Rank(Node v):

$r = v.left.size + 1$

$u = v$

while $u \neq root$ **do**

if $u == u.p.right$ **then**

$r = r + u.p.left.size + 1$

$u = u.p$

return r (vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife — ist r der Rang von v im Teilbaum mit Wurzel u .

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

Bei Schleifenabbruch: $u = \text{root}$.

$\Rightarrow r = u\text{-Rang}(v) = \text{Rang}(v)$.

Zusammenfassung:

Die Methode Rank() liefert wie gewünscht den Rang des übergebenen Knotens.

```

Rank(Node v):
    r = v.left.size + 1
    u = v
    while u ≠ root do
        if u == u.p.right then
            r = r + u.p.left.size + 1
        u = u.p
    return r
    (vorausgesetzt, dass T.nil.size = 0)

```

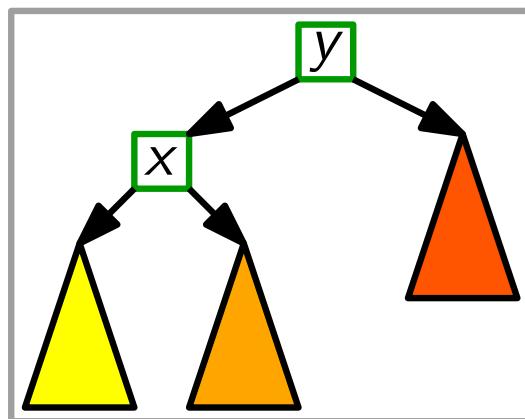
3. Aufwand zur Aufrechterhaltung der Extrainformation?

`RBInsert()` geht in zwei Phasen vor:

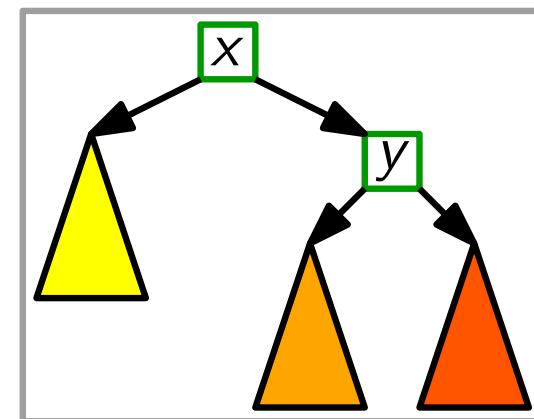
Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $O(h)$ { Für alle Knoten v auf dem Weg von der Wurzel zu z :
Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 Rotationen:



RightRotate(y)
Laufzeit $O(1)$
LeftRotate(x)



Welche Befehle müssen wir an `RightRotate(Node y)` anhängen, damit nach der Rotation alle `size`-Einträge wieder stimmen?

$$x.size = y.size$$

$$y.size = y.left.size + y.right.size + 1$$

(vorausgesetzt, dass $T.nil.size = 0$)

`RBDelete()` kann man analog „upgraden“.

Ergebnis

Satz. Das dynamische Auswahlproblem kann man so lösen, dass `Select()` und `Rank()` sowie alle gewöhnlichen Operationen für dynamische Mengen in einer Menge von n Elementen in $O(\log n)$ Zeit laufen.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v , $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) berechnen.



Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Beweisidee. Im Prinzip wie im Spezialfall $f \equiv size$.

Allerdings ist es im Prinzip möglich, dass sich die Veränderungen von einem gewissen veränderten Knoten bis in die Wurzel hochpropagieren. [Details Kapitel 14.2, CLRS]

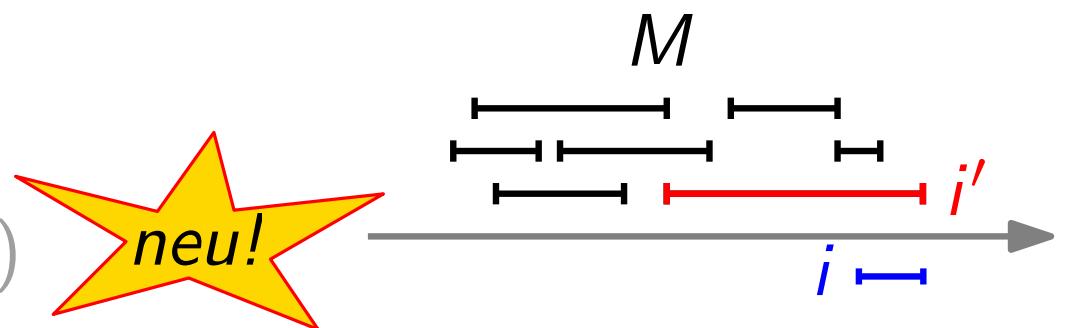
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element $\text{Insert}(\text{Interval } i)$
- Delete(Element e)
- Element $\text{Search}(\text{Interval } i)$



liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst *nil*.

Bitte lesen Sie's und
stellen Sie Fragen...

Algorithmen und Datenstrukturen

Wintersemester 2018/19
16. Vorlesung

Amortisierte Analyse

Special Guest: Fabian Feitsch

Einstiegsbeispiel: Hash-Tabellen

Frage: Wie groß macht man eine Hash-Tabelle?

Ziel: So groß wie nötig, so klein wie möglich...

Verhindere, dass die Tabelle überläuft
oder dass Operationen ineffizient werden.

Problem: Was tun, wenn man die maximale Anzahl zu speichernder Elemente vorab nicht kennt?

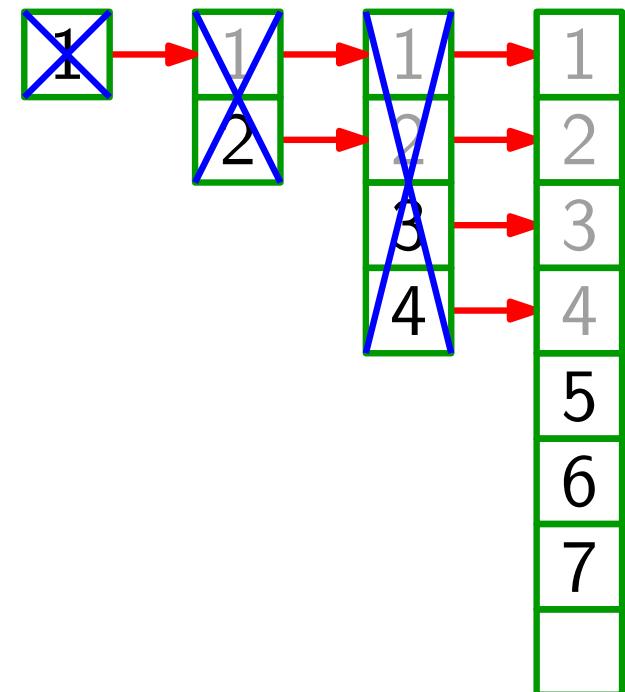
Lösung: *Dynamische Tabellen!*

Dynamische Tabellen

Idee:

- Wenn Tabelle voll, fordere doppelt so große Tabelle an (mit `new`).
- Kopiere alle Einträge von alter in neue Tabelle.
- Gib Speicher für alte Tabelle frei.

Insert(1)
Insert(2)
Insert(3)
Insert(4)
Insert(5)
Insert(6)
Insert(7)
...



Analyse: Welche Laufzeit benötigen n Einfügeoperationen im schlimmsten Fall?

Antwort:

- Tabelle wird genau ($\lceil \log_2 n \rceil$)-mal kopiert.
- Im schlimmsten (letzten!) Fall ist der Aufwand $\Theta(n)$.

Also ist der Gesamtaufwand ~~$\Theta(n \log n)$~~ , genauer $\Theta(n)$.
Let's see why...

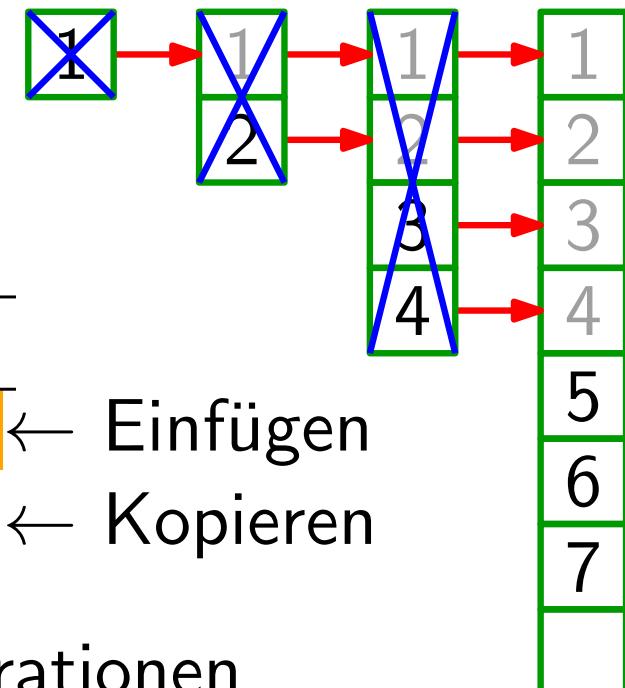
Lüge!

Genauere Abschätzung: Aggregationsmethode

Für $i = 1, \dots, n$ sei

c_i = Kosten fürs i -te Einfügen.

i	1	2	3	4	5	6	7	8	9
$size_i$	1	2	4	4	8	8	8	16	
c_i	1	1	1	1	1	1	1	1	
		1	2		4			8	



Also betragen die Kosten für n Einfügeoperationen

$$\sum_{i=1}^n c_i = n + \sum_{j=0}^{\lfloor \log_2(n-1) \rfloor} 2^j \leq n + \frac{2^{\log_2(n-1)+1} - 1}{2 - 1}$$

$$= n + 2(n - 1) - 1$$

$$< 3n \in \Theta(n)$$

$$\sum_{j=0}^k q^j = \frac{q^{k+1} - 1}{q - 1} \quad \begin{bmatrix} \text{endl.} \\ \text{geom.} \\ \text{Reihe} \end{bmatrix}$$

D.h. die durchschnittlichen („amortisierten“) Kosten sind $\Theta(1)$.

Amortisierte Analyse...

...bedeutet zu zeigen, dass die Operationen einer gegebenen Folge kleine durchschnittliche Kosten haben –
auch wenn einzelne Operationen in der Folge teuer sind!

Auch *randomisierte Analyse* kann man als Durchschnittsbildung (über alle Ereignisse, gewichtet nach Wahrscheinlichkeit) betrachten.

Bei amortisierter Analyse geht es jedoch um die durchschnittliche Laufzeit *im schlechtesten Fall* – nicht im Erwartungswert!

Wir betrachten 3 verschiedene Typen von amortisierter Analyse:

- Aggregationsmethode ✓
- Buchhaltermethode
- Potentialmethode

Buchhaltermethode

- Tendentiell genauer als die Aggregationsmethode
- Verbindet mit jeder Operation op_i amortisierte Kosten \hat{c}_i , die oft nicht mit den tatsächlichen Kosten c_i übereinstimmen.
 $\hat{c}_i > c_i \Rightarrow$ Wir legen etwas beiseite.
 $c_i > \hat{c}_i \Rightarrow$ Wir bezahlen teure Operationen mit vorher Beiseitegelegtem.
- Damit's klappt: wir dürfen nie in die Miesen kommen –

Guthaben $\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$ darf nicht negativ werden!

Dann gilt $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$.

D.h. amortisierte Kosten sind obere Schranke für tatsächliche Kosten!

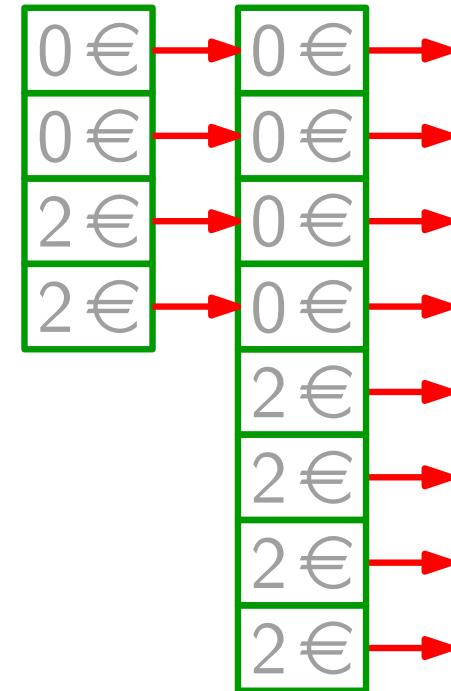
Buchhaltermethode für dynamische Tabellen

Jede Einfügeoperation op_i bezahlt $\hat{c}_i = 3\text{€}$:

- 1€ fürs tatsächliche Einfügen
- 2€ für die nächste Tabellenvergrößerung

Wir verknüpfen die Teilguthaben mit konkreten Objekten der Datenstruktur.

Damit wird deutlich, dass die DS nie Miese macht.



Also sind amortisierte Kosten obere Schranke für tatsächliche Kosten!

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i = 3n = \Theta(n)$$

D.h. die (tats.) Kosten für n Einfügeoperationen betragen $\Theta(n)$.

Buchhaltermethode: noch'n Beispiel



Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abs. Datentyp

```
boolean Empty()  
Push(key k)  
key Pop()  
key Top()
```

```
Multipop(int k)
```



Implementierung

```
while not Empty() and k > 0 do  
    Pop()  
    k = k - 1
```

Buchhaltermethode: Stapel mit Multipop

Betrachte Folge von Push-, Pop- und Multipop-Operationen.

Operation;	tatsächliche Kosten c_i	amortisierte Kosten \hat{c}_i
Push	1	2
Pop	1	0
$\text{Multipop}(k_i)$	$\min\{k_i, \text{size}_i\}$	0

Geht das gut??? – Ja! D.h. Folge von n Op. dauert $\Theta(n)$ Zeit.

Zeige: Amortisierte Kosten „bezahlen“ immer für die echten!

- Jede Push-Operation legt einen Teller auf den Stapel.
Dafür bezahlt sie 1 € und legt noch 1 € auf den Teller.
- Jede (Multi-)Pop-Operation wird mit den Euros auf den Tellern, die sie wegnimmt, komplett bezahlt.

Potentialmethode

Idee: Betrachte Bankguthaben (siehe Buchhaltermethode)
als physikalische Größe,
die den augenblicklichen Zustand der DS beschreibt.

Datenstruktur $D_0 \xrightarrow{op_1} D_1 \xrightarrow{op_2} \dots \xrightarrow{op_n} D_n$

Wähle Potential $\Phi: D_i \rightarrow \mathbb{R}$. O.B.d.A. $\Phi(D_0) = 0$

Ziel: Bank macht keine Miesen.

Also fordern wir $\Phi(D_i) \geq 0$ für $i = 1, \dots, n$.

amortisierte Kosten ↘ *echte Kosten* ↗ *Potentialdifferenz*
Def. $\hat{c}_i = c_i + \Delta\Phi(D_i)$, wobei $\Delta\Phi(D_i) = \Phi(D_i) - \Phi(D_{i-1})$

Folge: $\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1}))$ *teleskopierende Summe*



$$= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^n c_i$$

D.h. amortisierte Kosten „bezahlen“ für echte Kosten.

Potentialmethode: Stapel mit Multipop

To do: Definiere Potentialfunktion –
in Abhängigkeit vom aktuellen Zustand des Stapels!

Idee: Nimm $\Phi(D_i) = \text{size}_i$, also aktuelle Stapelgröße.
 $\Rightarrow \Phi(D_0) = 0$ und $\Phi(D_1), \dots, \Phi(D_n) \geq 0$. 

Prüfe: Falls die i -te Operation eine Push-Operation ist:
 $\Rightarrow \Delta\Phi(D_i) = +1$ und $\hat{c}_i = c_i + \Delta\Phi D_i = 1 + 1 = 2$

Falls die i -te Operation eine Multipop-Operation ist:
 $\Rightarrow \Delta\Phi(D_i) = -\min\{k_i, \text{size}_i\}$
 $c_i = +\min\{k_i, \text{size}_i\}$

$$\hat{c}_i = c_i + \Delta\Phi D_i = 0, \text{ dito mit Pop } (k_i = 1).$$

Also: Amortisierte Kosten pro Operation $O(1)$.
 \Rightarrow Echte Kosten für n Oper. im worst case $O(n)$.

Was
sind die
amort.
Kosten?

Zusammenfassung

Zeige mit **amortisierter Analyse**, dass die Operationen einer gegebenen Folge kleine durchschnittliche Kosten haben – *auch wenn einzelne Operationen in der Folge teuer sind!*

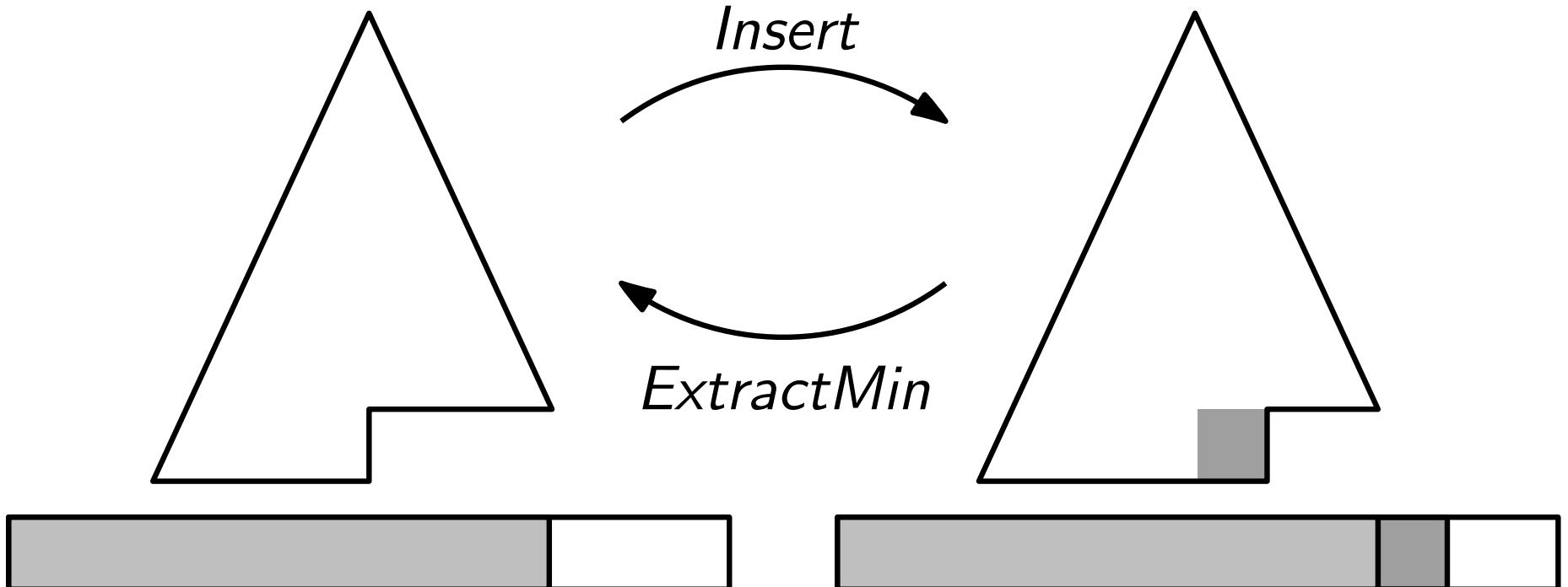
Drei Typen von amortisierter Analyse:

- Aggregationsmethode 
Summiere tatsächliche Kosten (oder obere Schranken dafür) auf.
- Buchhaltermethode 
Verbinde Extrakosten mit konkreten Objekten der DS und bezahle damit teure Operationen.
- Potentialmethode 
Definiere Potential der gesamten DS, so dass mit der Potentialdifferenz teure Operationen bezahlt werden können.

Übungsaufgaben zur amortisierten Analyse (I)

Gegeben sei ein gewöhnlicher MinHeap, dessen Methoden *Insert* und *ExtractMin* im schlechtesten Fall $O(\log n)$ Zeit brauchen.

Zeigen Sie mit der **Potentialmethode**, dass *Insert* amortisiert $O(\log n)$ Zeit und *ExtractMin* amortisiert $O(1)$ Zeit benötigt.



Übungsaufgaben zur amortisierten Analyse (II)

Entwerfen Sie eine Datenstruktur zum Verwalten einer dynamischen Menge von Zahlen. Die DS soll 2 Methoden haben:

- *Insert* zum Einfügen einer Zahl und
- *DeleteLargerHalf* zum Löschen aller Zahlen aus der Datenstruktur, die größer oder gleich dem aktuellen Median der Zahlenmenge sind.

Beide Methoden sollen **amortisiert $O(1)$** Zeit benötigen.

Tipp: Verwenden Sie eine Liste!

1. Beschreiben Sie Ihren Entwurf der Datenstruktur einschließlich der beiden Methoden in Worten.
2. Analysieren Sie mithilfe der **Buchhaltermethode**. Geben Sie die amortisierten Kosten, die Sie mit *Insert* und *DeleteLargerHalf* verbinden, exakt an.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
17. Vorlesung

Nächstes Paar

2. Zwischentest

Bitte melden Sie sich sofort an, falls Sie das noch nicht getan haben.

Die Abstimmung endet heute (Di, 18.12.) um 14:00 Uhr.

Problem:

- Gegeben: Menge P von n Punkten in der Ebene, jeder Punkt $p \in P$ als (x_p, y_p) .
Finde: Punktpaar $\{p, q\} \subseteq P$ mit kleinstem (euklidischen) Abstand.
- **Def.** Euklidischer Abstand von p und q ist $d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$.

Lösung:**Laufzeit:** $\Theta(n^2)$

- Gehe durch alle $\binom{n}{2}$ Punktpaare und berechne ihren Abstand.
- Gib ein Paar mit kleinstem Abstand zurück.

Mach's besser!

Entwurfsparadigma: – inkrementell?

– randomisiert?

– Teile und Herrsche?!

Spezialfall:



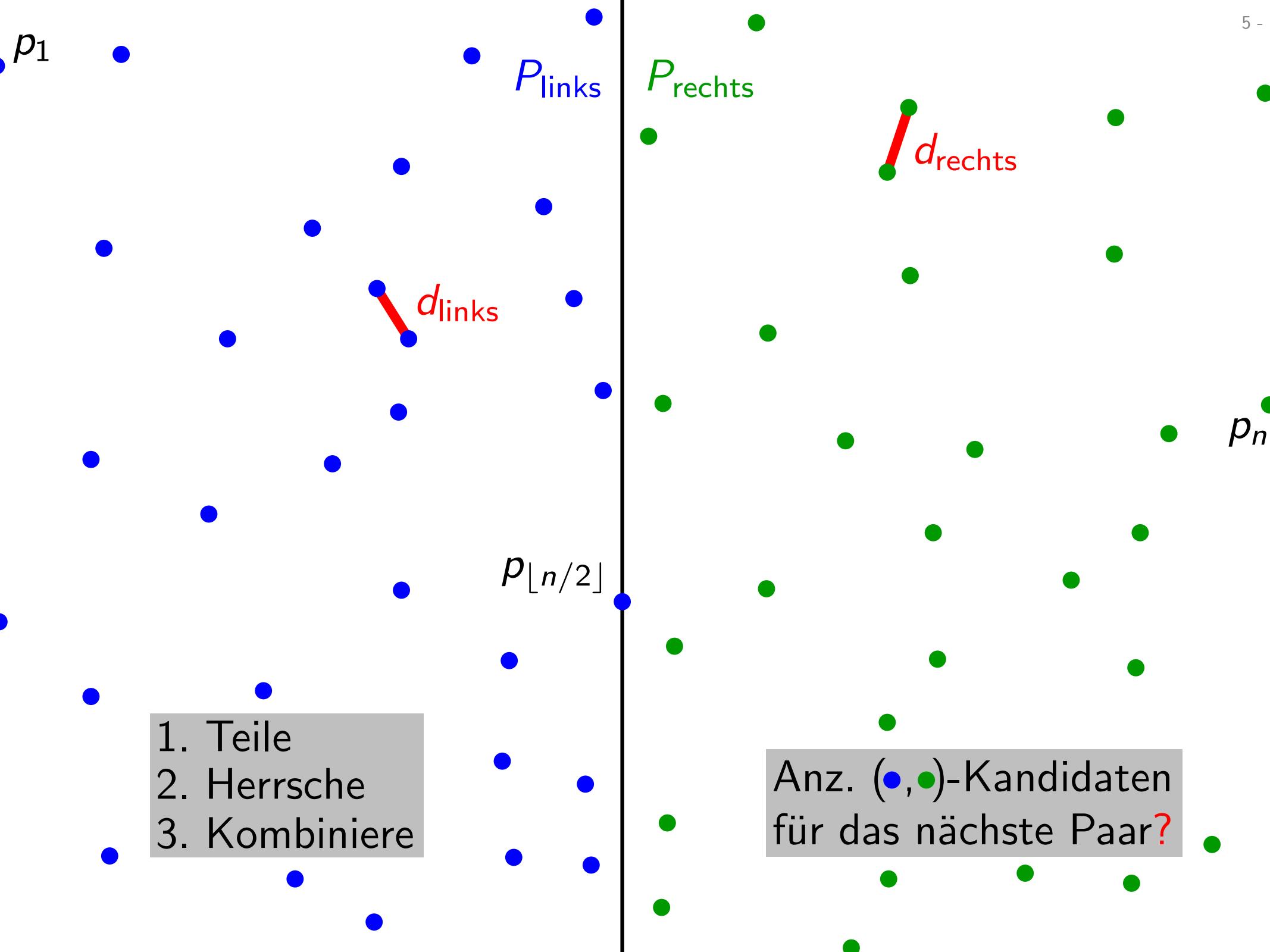
Lösung:

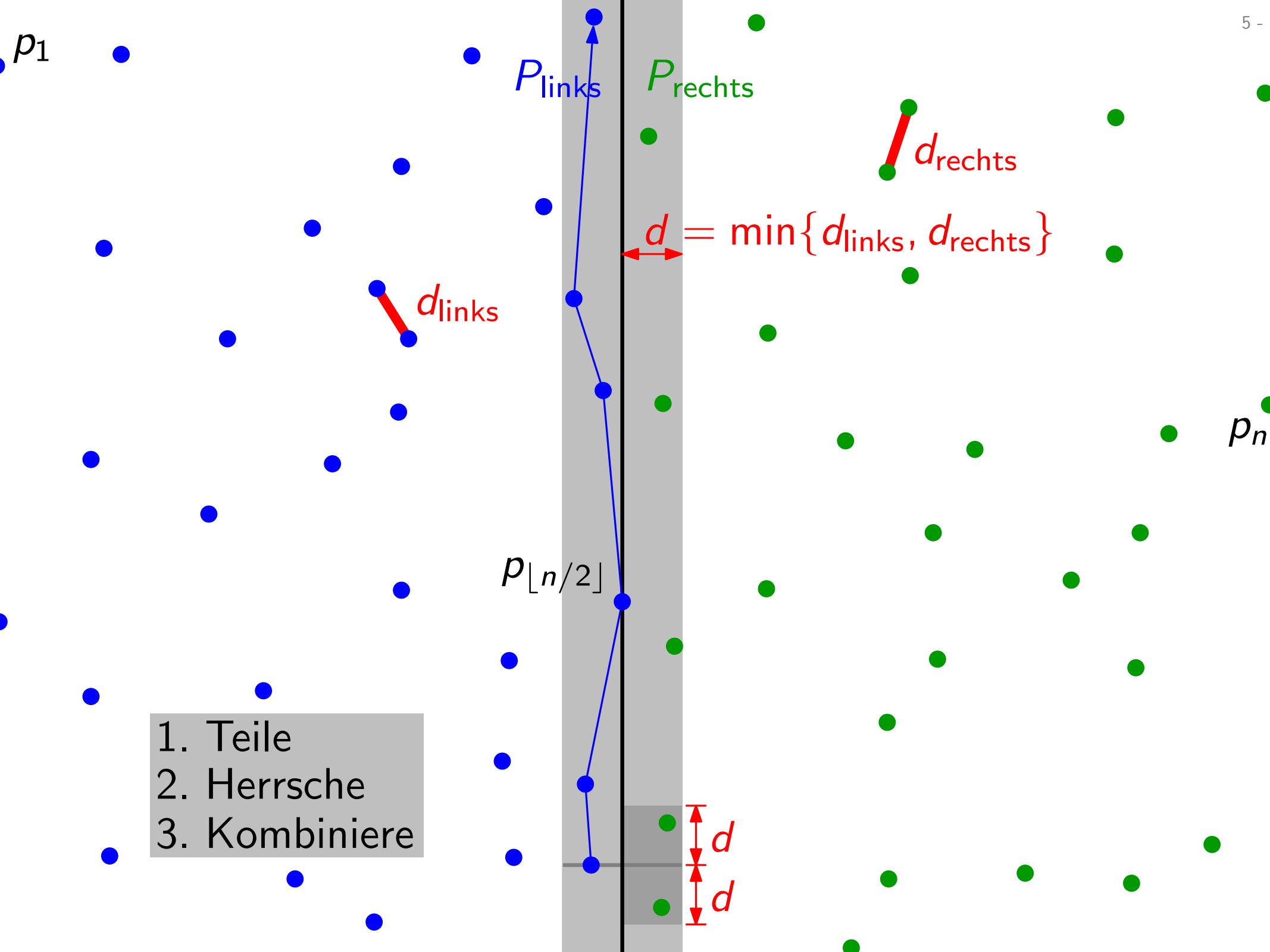
- Sortiere (nach x-Koordinate).
- Berechne Abstände aller *aufeinanderfolgender* Punktpaare.

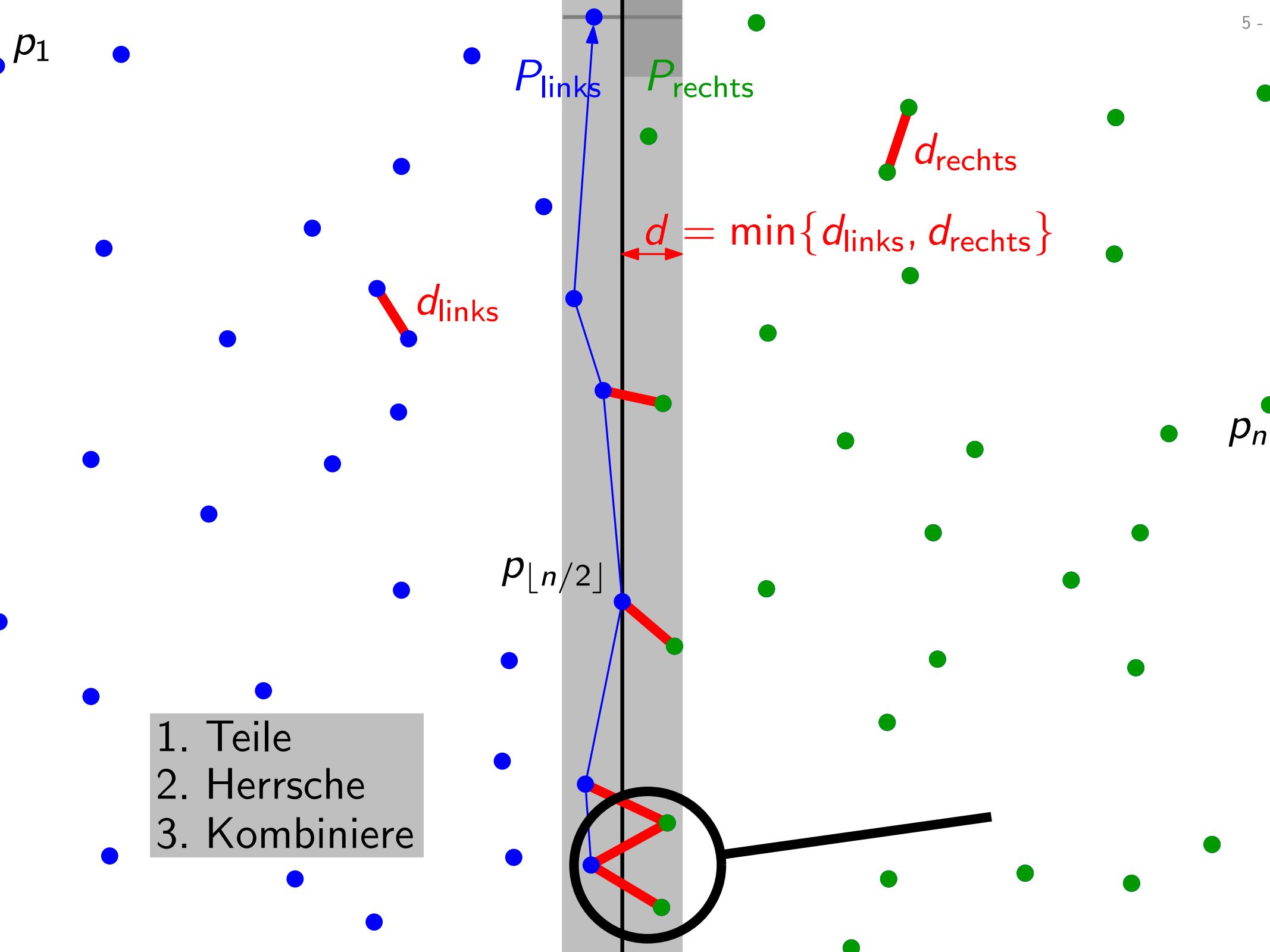
- Bestimme das Minimum dieser Abstände.

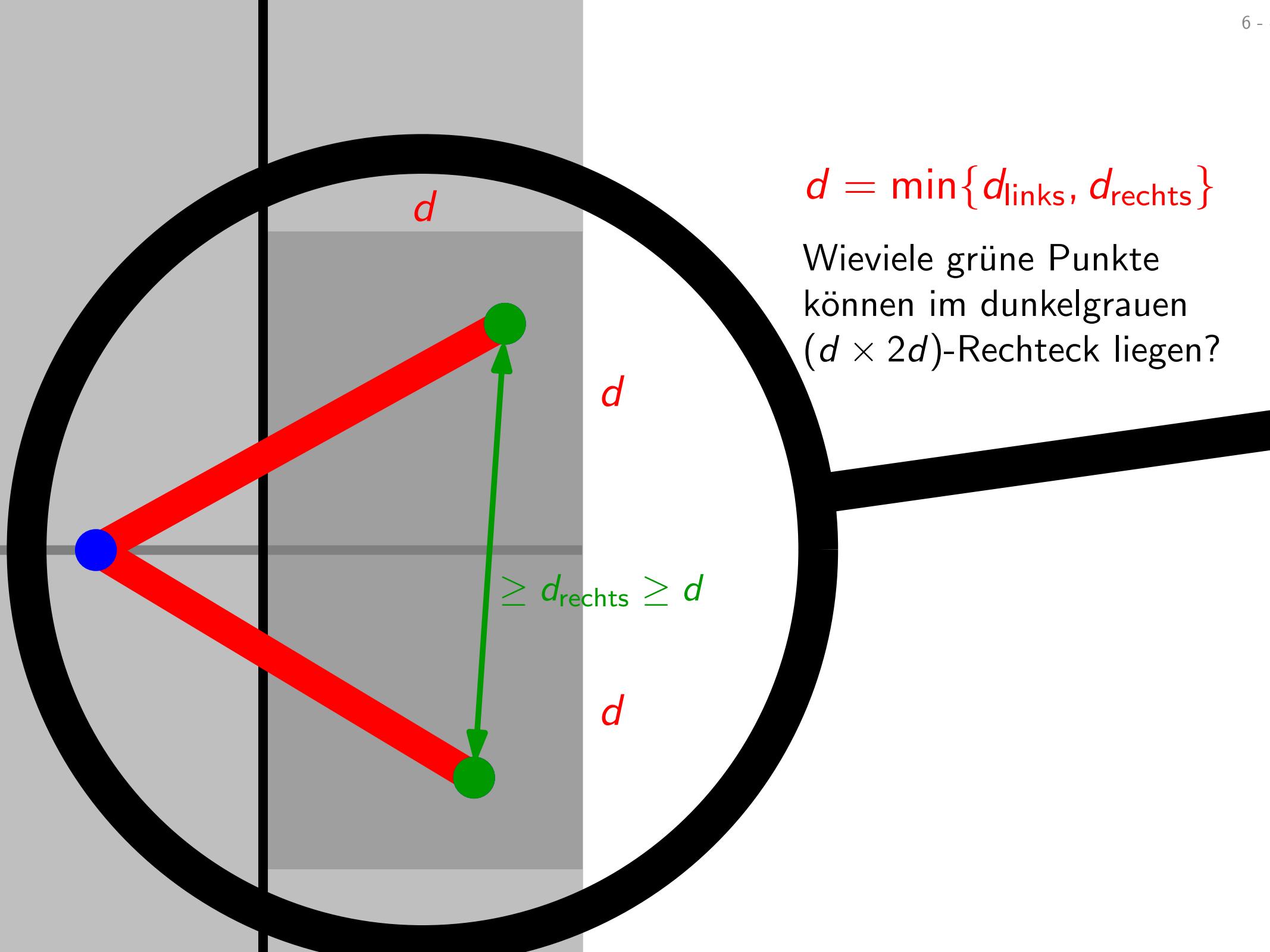
Strukturelle Einsicht:

Kandidatenmenge der Größe $n - 1$,
die gesuchtes Objekt enthält.



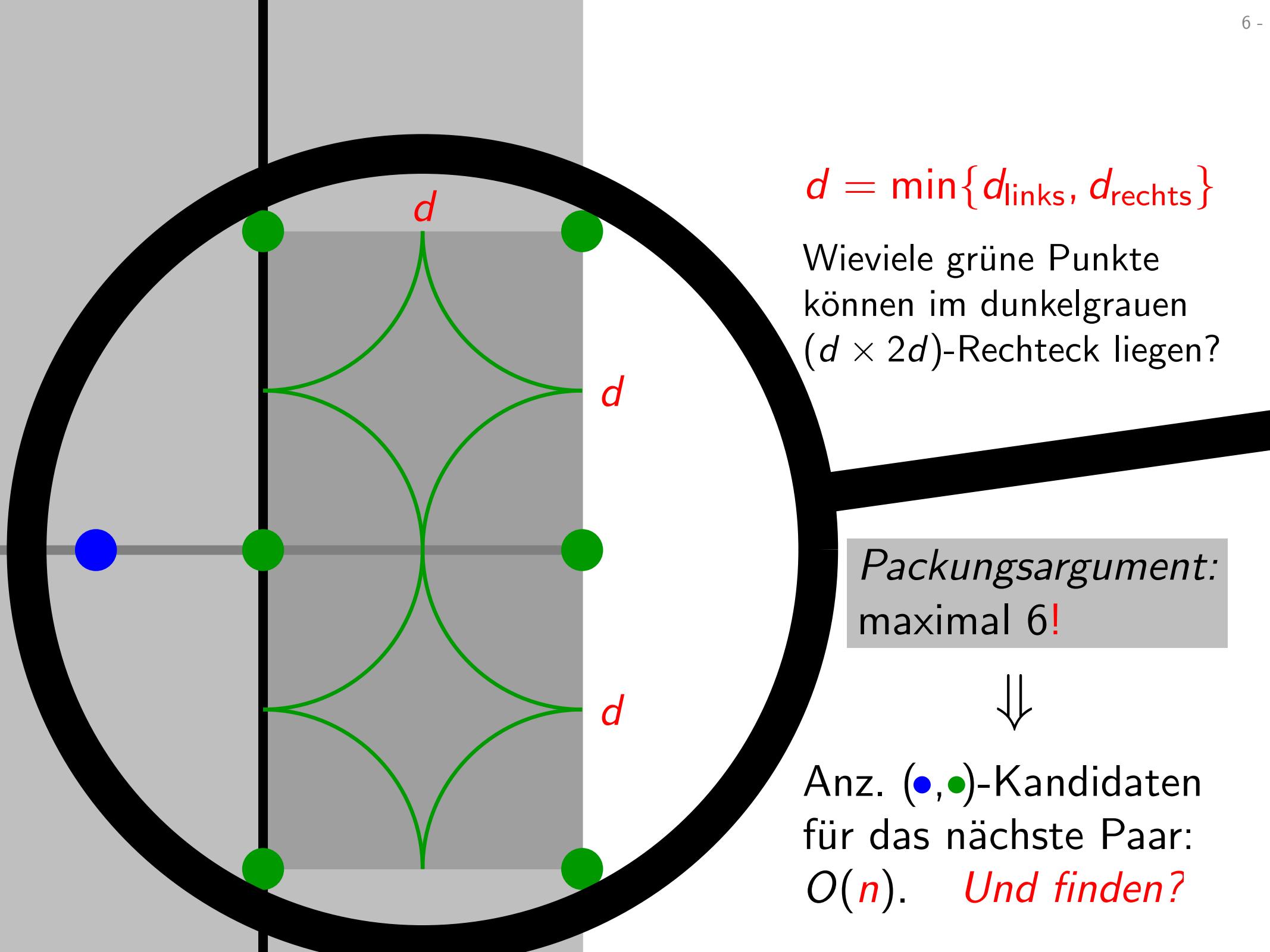






$$d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$$

Wieviele grüne Punkte können im dunkelgrauen $(d \times 2d)$ -Rechteck liegen?



$$d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$$

Wieviele grüne Punkte können im dunkelgrauen $(d \times 2d)$ -Rechteck liegen?

*Packungsargument:
maximal 6!*



Anz. (\bullet, \bullet) -Kandidaten für das nächste Paar:
 $O(n)$. *Und finden?*

Algorithmus

$$T(n) = \begin{cases} \text{Laufzeit des rekursiven Teils,} \\ \text{d.h. ohne Vorverarbeitung (1.)} \end{cases}$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$

2. Teile: $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$, $P_{\text{rechts}} = P \setminus P_{\text{links}}$

3. Herrsche:

bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}
 d_{rechts} P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$
- sortiere P_{links} und P_{rechts} nach y-Koordinate
- gehe „gleichzeitig“ durch P_{links} und P_{rechts} :
 für jeden Punkt p in P_{links} gehe in P_{rechts} bis y-Koord. $y_p + d$; halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)
- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

Algorithmus

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n)$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$

2. Teile: $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$, $P_{\text{rechts}} = P \setminus P_{\text{links}}$

3. Herrsche:

bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}

d_{rechts}

P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$

$O(1)$

- sortiere P_{links} und P_{rechts} nach y-Koordinate

$O(n \log n)$

- gehe „gleichzeitig“ durch P_{links} und P_{rechts} :

für jeden Punkt p in P_{links} gehe in P_{rechts} bis y-Koord. $y_p + d$; halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)

$O(n)$

- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

Laufzeit

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n)$$

Also $T(n) \approx 2T(n/2) + O(n \log n)$

Rekursionsgleichung mit Master-Theorem lösen?

Bestimme Parameter für das Theorem:

$$a = b = 2, f(n) = O(n \log n).$$

Betrachte $n^{\log_b a} = n^{\log_2 2} = n^1$.

Gilt $f \in \left\{ \begin{array}{ll} O(n^{1-\varepsilon}) & \text{für ein } \varepsilon > 0 \\ \Theta(n^1) & \\ \Omega(n^{1+\varepsilon}) & \text{für ein } \varepsilon > 0 \end{array} \right\}$?

Nein, $f: n \mapsto O(n \log n)$ passt in keinen der drei Fälle.



Die Rekursionsbaummethode liefert... $T(n) = O(n \log^2 n)$.

Noch besser?

$$T(n) \approx 2T(n/2) + O(n \log n) = O(n \log^2 n)$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$

2. Teile: P in $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$ und $P_{\text{rechts}} = P \setminus P_{\text{links}}$

3. Herrsche:

bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}
 $?$! d_{rechts} P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$
- sortiere P_{links} und P_{rechts} nach y-Koordinate

- gehe „gleichzeitig“ durch P_{links} und P_{rechts} :
 - für jeden Punkt p in P_{links} gehe in P_{rechts} bis y-Koord. $y_p + d$;
 - halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)
- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

Noch besser!

$$T(n) \approx 2T(n/2) + O(n \log n) = O(n \log^{\otimes} n)$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$
und $P' = P$ nach y-Koordinate $\rightarrow p'_1, \dots, p'_n$ mit $y'_1 \leq \dots \leq y'_n$

2. Teile: P in $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$ und $P_{\text{rechts}} = P \setminus P_{\text{links}}$
 P' in P'_{links} und P'_{rechts} (sortiert nach y-Koordinate)

3. Herrsche:
bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}
 d_{rechts} P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$
- gehe „gleichzeitig“ durch P'_{links} und P'_{rechts} :
für jeden Punkt p in P'_{links} gehe in P'_{rechts} bis y-Koord. $y_p + d$; halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)
- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P'_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

$O(n)$

Zusammenfassung

1. Vorverarbeitung (2× Sortieren)	$O(n \log n)$	
2. Teilen	$O(n)$	
3. Herrschen	$2T(n/2)$	
4. Kombinieren	$O(n)$	
<hr/>		
Gesamtlaufzeit	$O(n \log n)$	

Speicherplatzbedarf?

$O(n)$, wenn P' *in situ* in P'_{links} und P'_{rechts} zerlegt wird.

Ist die Laufzeit $O(n \log n)$ optimal?

Def. *Element-Uniqueness-Problem (für natürliche Zahlen)*

Gegeben eine Folge a_1, \dots, a_n von n Zahlen,
kommt jede Zahl nur einmal vor, d.h. $a_i \neq a_j$ für $i \neq j$?

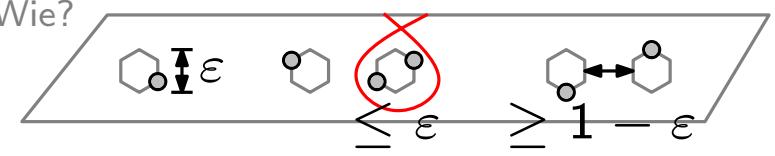
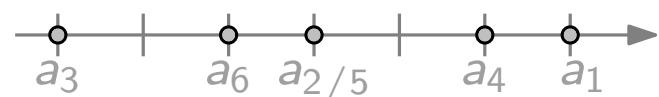
Satz. Das Element-Uniqueness-Problem kann nicht schneller als in $\Omega(n \log n)$ Zeit gelöst werden –
wenn man als Rechenmodell das sogenannte
algebraische Entscheidungsbaummodell zugrunde legt.

Was bedeutet das für das Problem *Nächstes Paar*?

Angenommen wir könnten Nächstes Paar in $\textcolor{red}{o}(n \log n)$ Zeit lösen – dann auch Element Uniqueness! ↗

Wie? Teste, ob das nächste Paar Abstand 0 hat!

Genaugenommen muss man die Zahlen a_1, \dots, a_n in eine Menge von (paarweise verschiedenen!) Punkten der Ebene transformieren, aber auch das geht! – Wie?

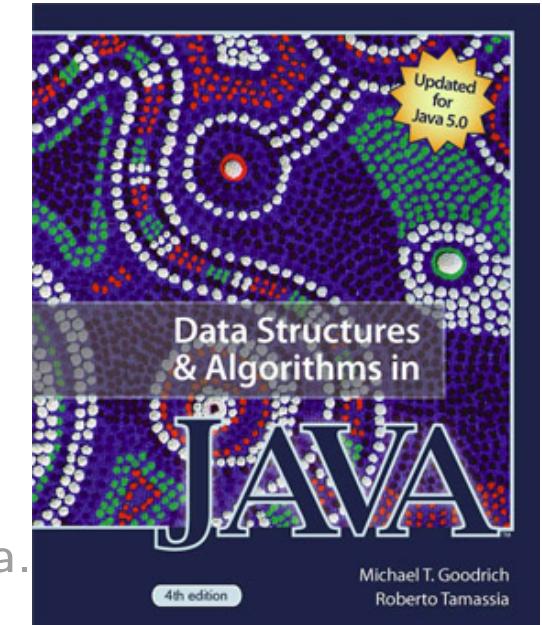


Das heißt. . .

- Satz.** Das Problem Nächstes Paar kann nicht schneller als in $\Omega(n \log n)$ Zeit gelöst werden, wenn man als Rechenmodell das algebraische Entscheidungsbaummodell zugrunde legt.
- Kor.** Unser $O(n \log n)$ -Zeit-Algorithmus für das Problem Nächstes Paar ist asymptotisch optimal, wenn man. . . .

Учиться, учиться и учиться

- Implementieren Sie die einfache Brute-Force-Lösung in Java.
- Implementieren Sie einen einfachen Teile-und-Herrsche-Algorithmus, der im Herrsche-Schritt *alle* (quadratisch vielen) (●, ●)-Kandidaten testet. *(Ist der schneller als der Brute-Force-Alg.?)*
- Implementieren Sie den hier vorgestellten Teile-und-Herrsche-Algorithmus, der in $O(n \log^2 n)$ Zeit läuft!
- Implementieren Sie den hier vorgestellten Teile-und-Herrsche-Algorithmus, der in $O(n \log n)$ Zeit läuft!



Goodrich & Tamassia:
Data Structures & Algorithms in Java.
Wiley, 4. Aufl., 2005 (5. Aufl., 2010)

Algorithmen & Datenstrukturen

Lernziele: In dieser Veranstaltung haben Sie schon gelernt...

- die Effizienz von Algorithmen zu messen und miteinander zu vergleichen,
- grundlegende Algorithmen und Datenstrukturen in Java zu implementieren,
- selbst Algorithmen und Datenstrukturen zu entwerfen sowie
- deren Korrektheit und Effizienz zu beweisen.

Inhalt: • Grundlagen und Analysetechniken

- Sortierverfahren

- Java

- Datenstrukturen

- Graphenalgorithmen (kürzeste Wege, min. Spannbäume)
- Systematisches Probieren (dynamisches Progr., Greedy-Alg.)

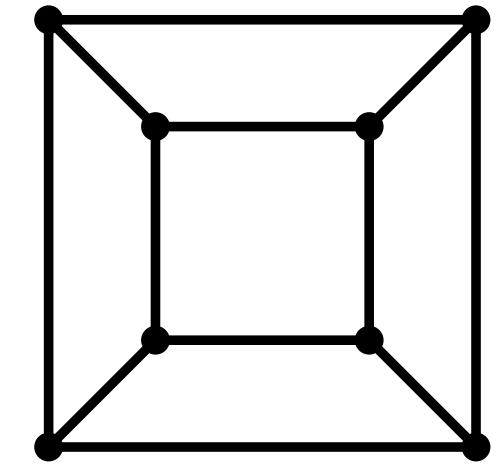
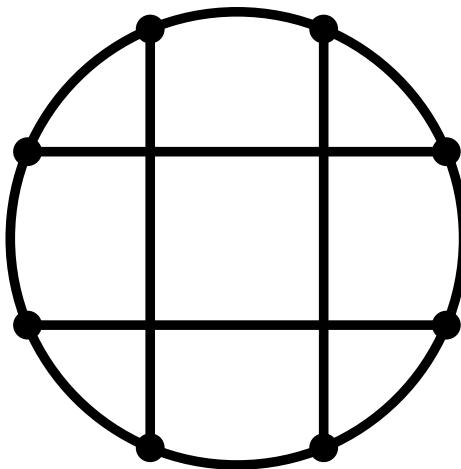
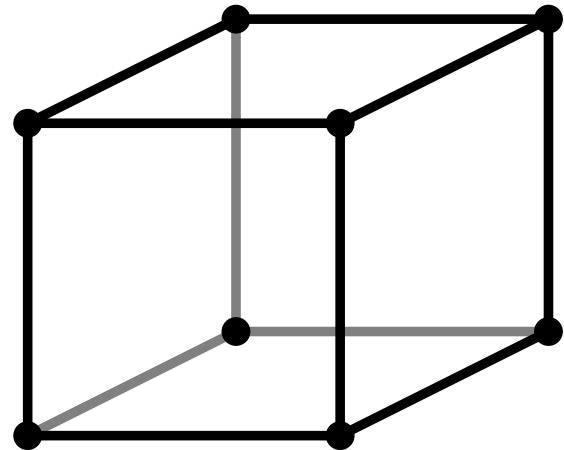
Todo

Algorithmen und Datenstrukturen

Wintersemester 2018/19
18. Vorlesung

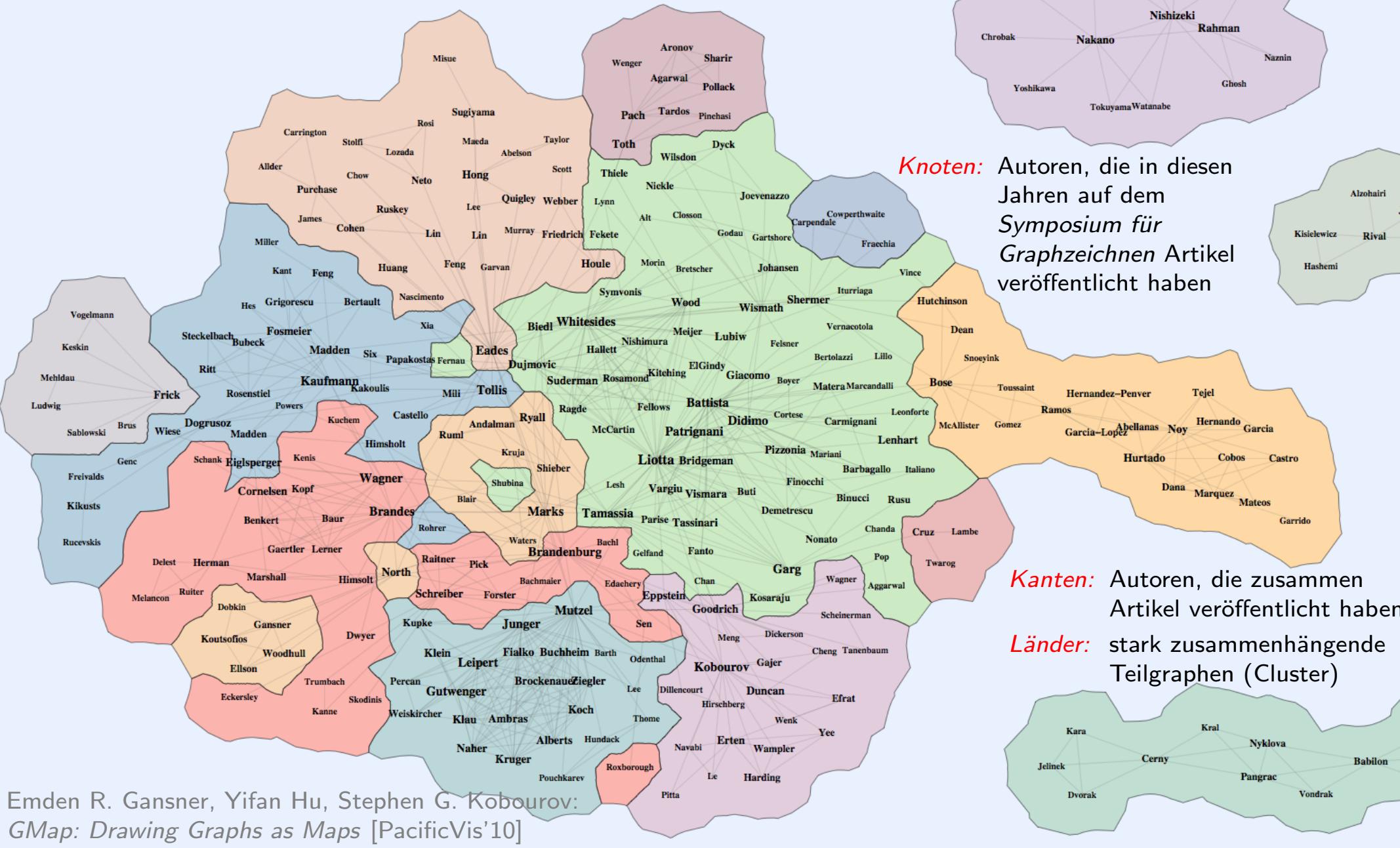
Graphen:
Repräsentation und Durchlaufstrategien

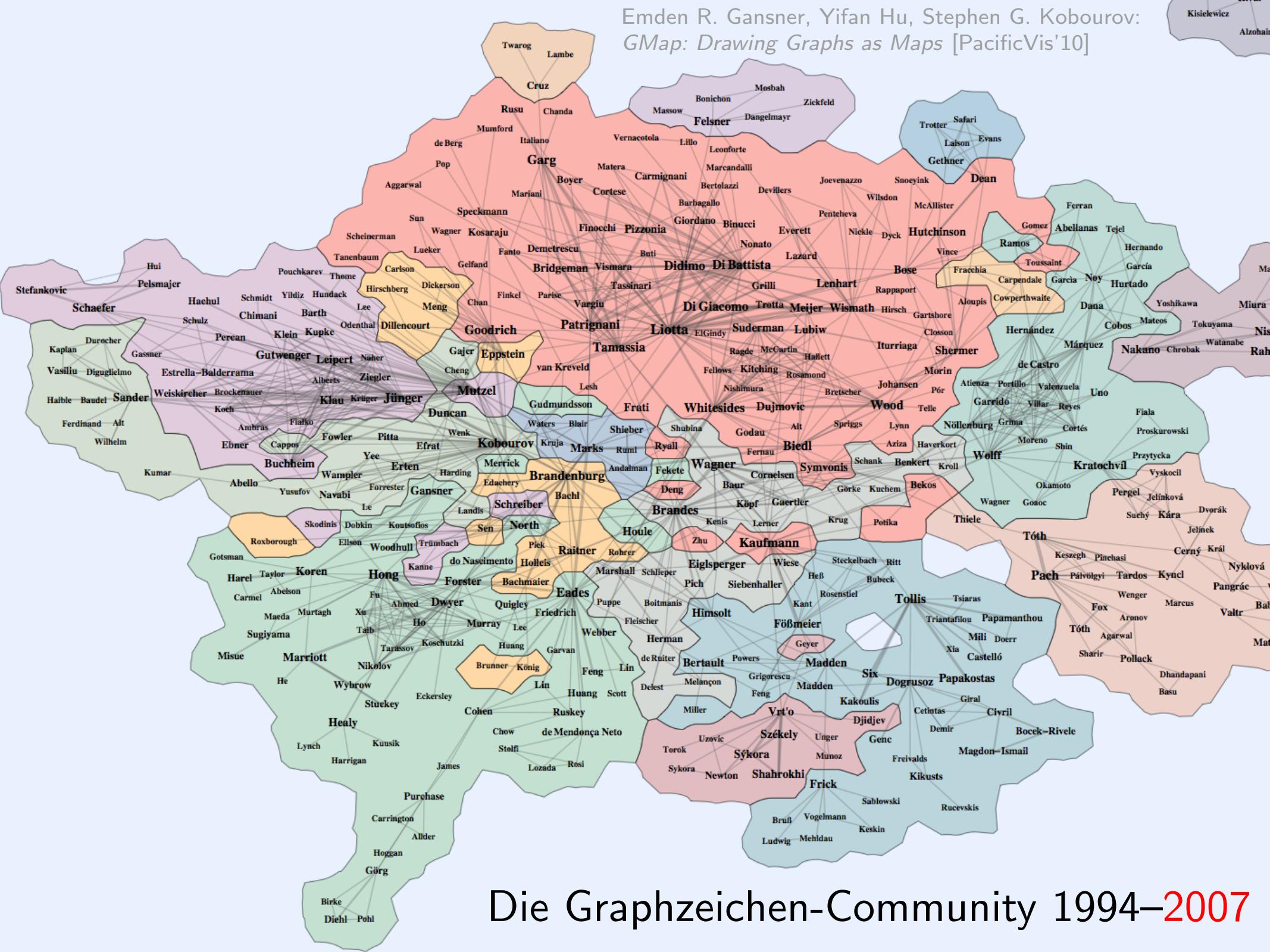
Was ist das?



Ein (und derselbe) *Graph*; der dreidimensionale Hyperwürfel.

Die Graphzeichen-Community 1994–2004





Die Graphzeichen-Community 1994–2007

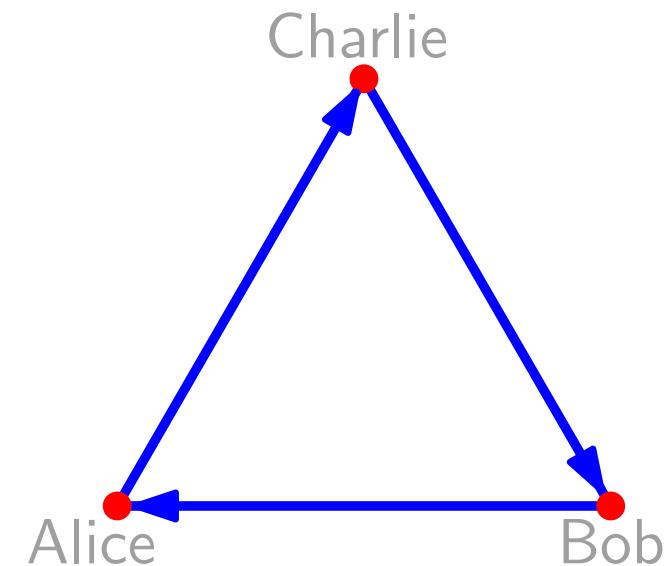
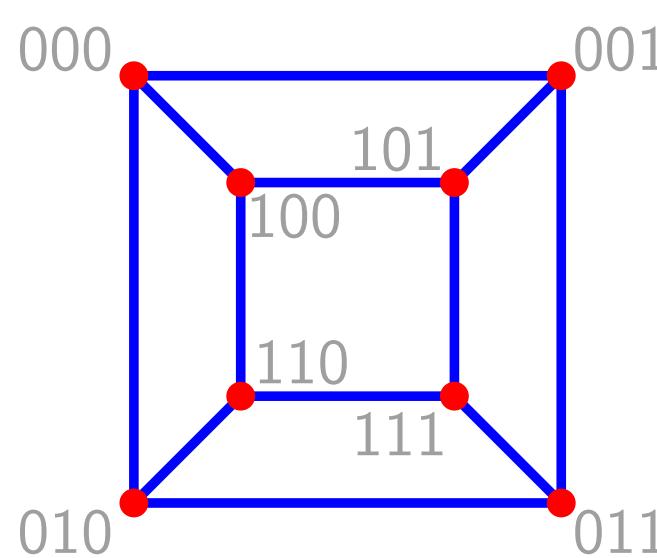
F: Was ist ein Graph?

A₁: Ein (ungerichteter) Graph ist ein Paar (V, E) , wobei

- V Knotenmenge und
- $E \subseteq \binom{V}{2} = \{\{u, v\} \subseteq V \mid u \neq v\}$ Kantenmenge.

$$V = \{000, 001, \dots, 111\}$$

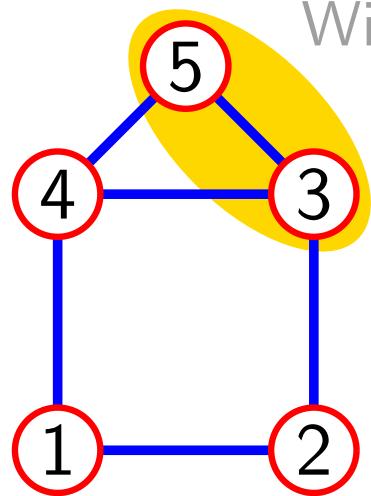
$$\{u, v\} \in E \Leftrightarrow ?$$



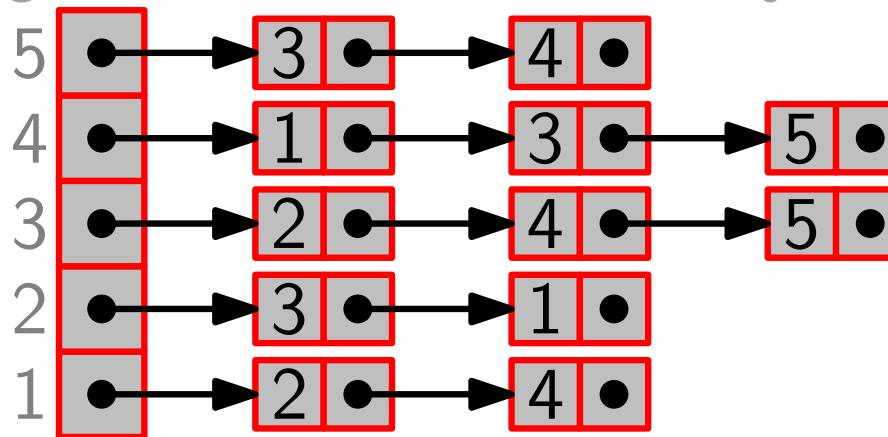
A₂: Ein gerichteter Graph ist ein Paar (V, E) , wobei

- V Knotenmenge und
- $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$ Kantenmenge.

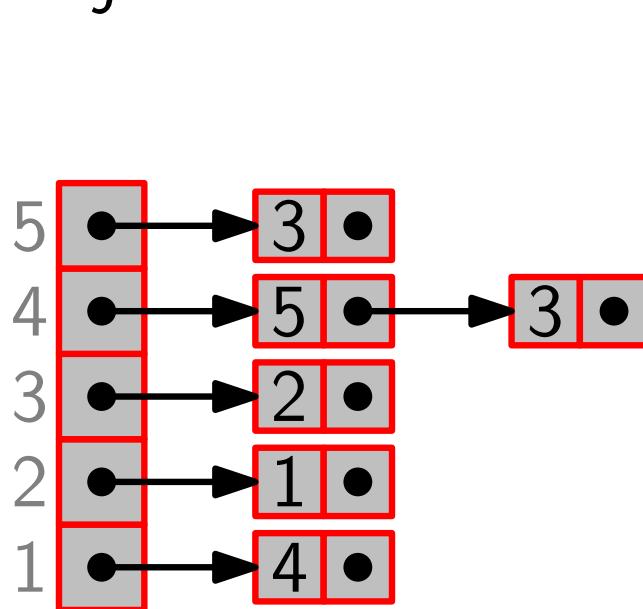
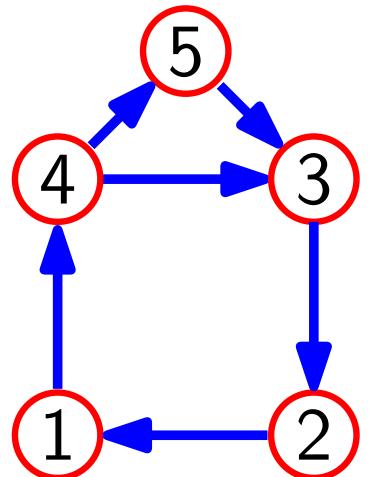
F: Wie repräsentiere ich einen Graphen?



Wir sagen: Knoten 3 und 5 sind *adjazent*.



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	0
3	0	1	0	1	1
4	1	0	1	0	1
5	0	0	1	1	0



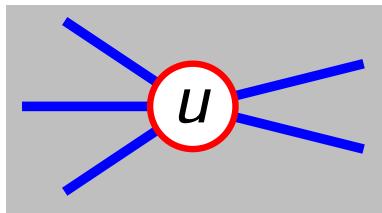
	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	0	0

$$\text{Adj}[i] = \{j \in V \mid (i, j) \in E\}$$

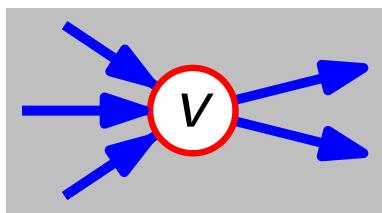
$$a_{ij} = 1 \Leftrightarrow (i, j) \in E$$

Grad eines Knotens

Def.



$$\deg u = |\text{Adj}[u]|$$



$$\text{outdeg } v = |\text{Adj}[v]|$$

$$\text{indeg } v = |\{u \in V : (u, v) \in E\}|$$

Beob.

Sei $G = (V, E)$ ein ungerichteter Graph.

Dann ist die Summe aller Knotengrade $= 2 \cdot |E|$.

Beweis.

Technik des *zweifachen Abzählens*:

Zähle alle Knoten-Kanten-Inzidenzen.

Eine Kante ist *inzident* zu ihren Endknoten.

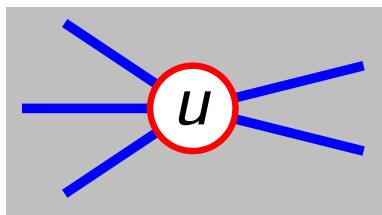
Ein Knoten ist *inzident* zu allen Kanten, deren Endknoten er ist.

Aus Sicht der Knoten: $\sum_{v \in V} \deg v$

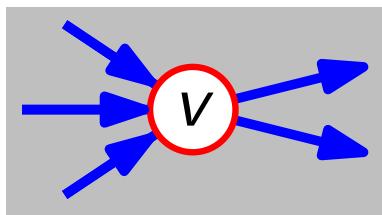
Aus Sicht der Kanten: $2 \cdot |E|$ also gleich!

Grad eines Knotens

Def.



$$\deg u = |\text{Adj}[u]|$$



$$\text{outdeg } v = |\text{Adj}[v]|$$

$$\text{indeg } v = |\{u \in V : (u, v) \in E\}|$$

Beob.

Sei $G = (V, E)$ ein ungerichteter Graph.

Dann ist die Summe aller Knotengrade $= 2 \cdot |E|$.

Sätzle.

Die Anzahl der Knoten ungeraden Grades ist gerade.

Beweis.

$$2 \cdot |E| = \sum_{v \in V} \deg v = \sum_{v \in V_{\text{ger}}} \deg v + \sum_{v \in V_{\text{ung}}} \deg v$$

gerade! *gerade!* *gerade!* \Rightarrow *gerade!*

$$\sum_{v \in V_{\text{ung}}} \deg v \text{ gerade } \Rightarrow |V_{\text{ung}}| \text{ ist gerade!}$$

□

Rundlaufstrategien für ungerichtete Graphen

1. Durchlaufe einen Graphen auf einem Kreis,
so dass jede Kante genau einmal durchlaufen wird.

Charakterisierung: Bei welchen Graphen geht das (nicht)?

Konstruktion: Wie (und in welcher Zeit) finde ich
einen solchen Rundlauf, falls er existiert?

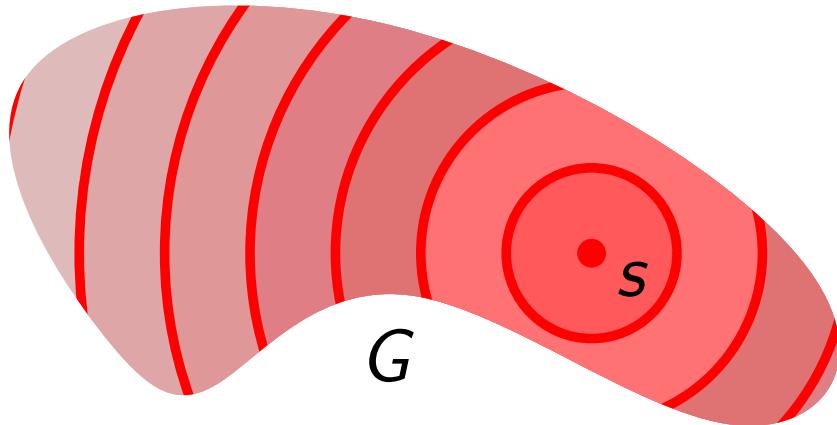
2. Durchlaufe einen Graphen auf einem Kreis,
so dass jeder **Knoten** genau einmal durchlaufen wird.

Charakterisierung: Bei welchen Graphen geht das (nicht)?

Konstruktion: Wie (und in welcher Zeit) finde ich
einen solchen Rundlauf, falls er existiert?

F: Wie durchlaufe ich einen Graphen?

Ideen?



1. wellenförmige Ausbreitung ab einem gegebenen Startknoten s
Breitensuche (breadth-first search, BFS)
2. vom Startknoten s möglichst schnell weit weg
Tiefensuche (depth-first search, DFS)

Breitensuche

BFS(Graph G , Vertex s)

Initialize(G, s)

$Q = \text{new Queue}()$

$Q.\text{Enqueue}(s)$

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{Dequeue}()$

foreach $v \in \text{Adj}[u]$ **do**

if $v.\text{color} == \text{white}$ **then**

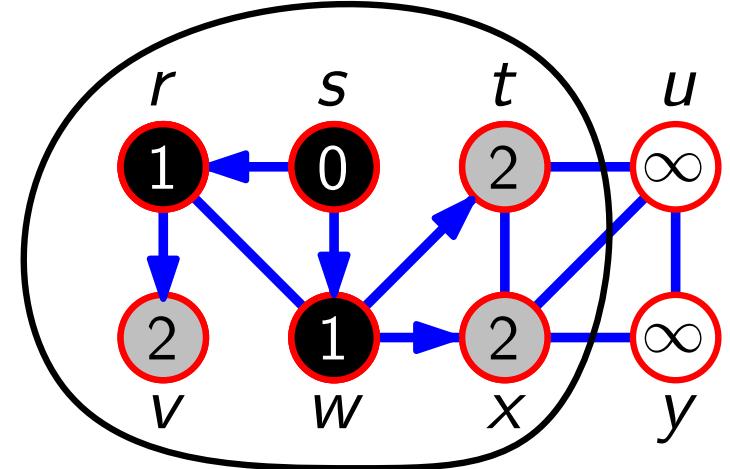
$v.\text{color} = \text{gray}$

$v.d = u.d + 1$

$v.\pi = u$

$Q.\text{Enqueue}(v)$

$u.\text{color} = \text{black}$



$w \quad r \quad t \quad x \quad v \quad \quad \quad usw.$

Initialize(Graph G , Vertex s)

foreach $u \in V$ **do**

$u.\text{color} = \text{white}$

$u.d = \infty$

$u.\pi = \text{nil}$

$s.\text{color} = \text{gray}$

$s.d = 0$

Laufzeit?

Initialize	En-/Dequeues	Adjazenzlisten (foreach-Schleifen)
$O(V)$	$+ O(V)$	$+ O(E) = O(V + E)$
		[Beob. über Knotengrade!]

Korrektheit von BFS – Vorbereitung

Definition. Sei $G = (V, E)$ (un)gerichteter Graph, $u, v \in V$.

$\delta(u, v) :=$ Länge eines kürzesten u - v -Wegs,
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Ziel:

Zeige, dass nach $\text{BFS}(G, s)$ für alle $v \in V$ gilt:

$$v.d = \delta(s, v).$$

berechneter Abstand von s *tatsächlicher Abstand von s*

Lemma 1.

(Eigenschaft kürzester Wege)

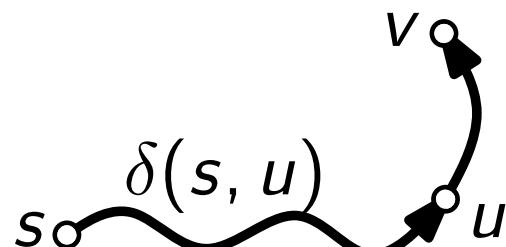
Sei $G = (V, E)$ ein (un)gerichteter Graph, $s \in V$.

Dann gilt für jede Kante $(u, v) \in E$:

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Beweis.

1. Fall: u ist von s erreichbar (d.h. \exists s - u -Weg)



Dieser s - v -Weg hat Länge $\delta(s, u) + 1$.



Kürzester s - v -Weg hat Länge $\leq \delta(s, u) + 1$.

Korrektheit von BFS – Fortsetzung

Lemma 1. Sei $s \in V$. Dann gilt für jede Kante $(u, v) \in E$:
 $\delta(s, v) \leq \delta(s, u) + 1$.



Lemma 2. Sei $G = (V, E)$ ein (un)gerichteter Graph, $s \in V$. Nach $\text{BFS}(G, s)$ gilt für alle $v \in V$: $v.d \geq \delta(s, v)$.

Beweis. Induktion über die Anz. k von Enqueue-Oper.

```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while not Q.Empty() do
    u = Q.Dequeue()
    foreach v ∈ Adj[u] do
      if v.color == white then
        v.color = gray
        v.d = u.d + 1
        v.π = u
        Q.Enqueue(v)
    u.color = black
```

$k = 1$: Situation nach $Q.\text{Enqueue}(s)$:

- $s.d = 0 = \delta(s, s)$
- für alle $v \in V \setminus \{s\}$ gilt $v.d = \infty \geq \delta(s, v)$

$k > 1$: Situation nach $Q.\text{Enqueue}(v)$:

v war gerade noch weiß und ist benachbart zu u .
 $v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$

Induktionsannahme für u Lemma 1
 $(u.d$ wurde gesetzt, als Anz. Enqueue-Oper. $< k)$

Jetzt ist v grau. $\Rightarrow v.d$ ändert sich nicht mehr.



Korrektheit von BFS – Fortsetzung

Lemma 2. Sei $G = (V, E)$ ein (un)gerichteter Graph, $s \in V$. Nach $\text{BFS}(G, s)$ gilt für alle $v \in V$: $v.d \geq \delta(s, v)$.



Lemma 3. Sei $Q = \langle v_1, v_2, \dots, v_r \rangle$ während BFS. Dann gilt:

- (A) $v_r.d \leq v_1.d + 1$ und
- (B) $v_i.d \leq v_{i+1}.d$ für $i = 1, \dots, r - 1$.

Also d -Werte der Knoten in Q z.B. $\langle 3, 3, 4, 4, 4 \rangle$.

Korollar. Angenommen u wird früher als v in Q eingefügt, dann gilt $u.d \leq v.d$, wenn v in Q eingefügt wird.

Beweis. Folgt aus Lemma 3 und der Tatsache, dass jeder Knoten $\leq 1 \times$ einen endlichen d -Wert bekommt.

Korrektheit von BFS – Hauptsatz

Satz. Sei G ein (un)gerichteter Graph, s ein Knoten von G . Nach $\text{BFS}(G, s)$ gilt:

- (i) Für alle Knoten $v \in V$ gilt $v.d = \delta(s, v)$.
- (ii) Jeder von s erreichbare Knoten wird entdeckt.
- (iii) Für jeden von s erreichbaren Knoten $v \neq s$ gilt:
es gibt einen kürzesten s - v -Weg, der aus einem
kürzesten s - v . π -Weg und der Kante $(v.\pi, v)$ besteht.

Beweis. (i) \Rightarrow (ii), (iii). Es genügt also (i) zu zeigen.

Lemma 2 $\Rightarrow v.d \geq \delta(s, v)$. Noch z.z.: $v.d \leq \delta(s, v)$.

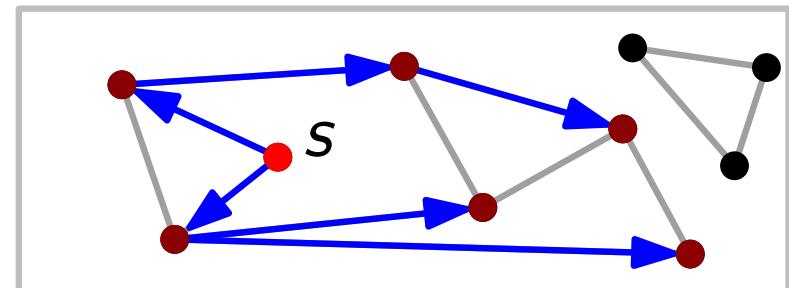
Widerspruchsbeweis mit Wahl des „kleinsten Schurken“.

Siehe Kapitel 22.2 [CLRS].

BFS-Bäume

Betrachte den *Vorgänger-Graphen* $G_\pi = (V_\pi, E_\pi)$ von G :

- $V_\pi = \{v \in V : v.\pi \neq \text{nil}\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V_\pi \setminus \{s\}\}$



Klar: G_π ist ein Baum (da zshg. und $|E_\pi| = |V_\pi| - 1$).

Beh.: G_π ist ein *Kürzeste-Wege-Baum* (oder *BFS-Baum*), d.h.

- $V_\pi = \{v \in V : v$ erreichbar von $s\}$
- für alle $v \in V_\pi$ enthält G_π einen eindeutigen Weg von s nach v , der ein kürzester s - v -Weg ist.

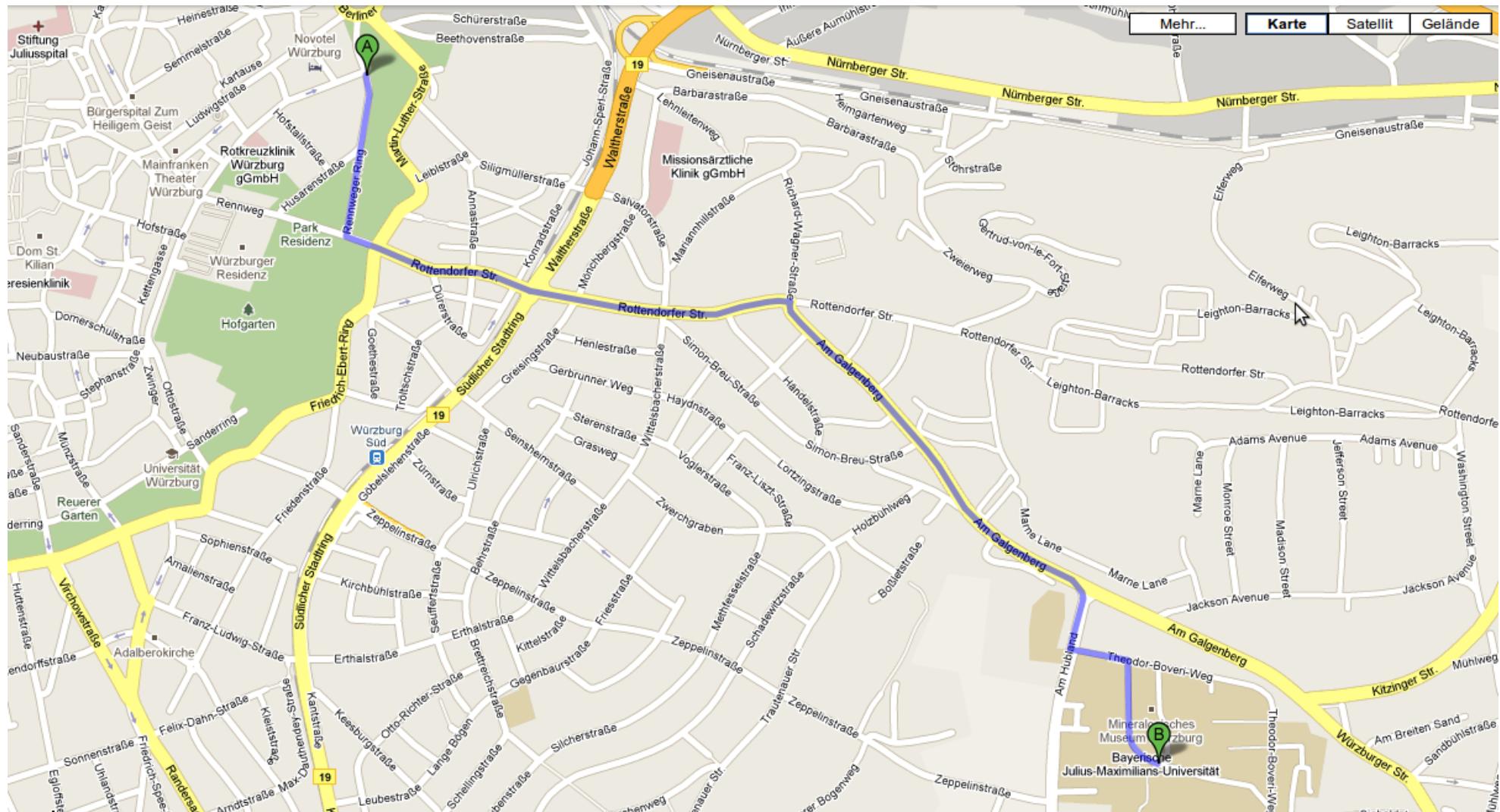
Bew.: Folgt aus (ii) und (iii) im Hauptsatz. □

Algorithmen und Datenstrukturen

Wintersemester 2018/19
19. Vorlesung

Kürzeste Wege & Dijkstras Algorithmus

Wozu kürzeste Wege?



Modellierung des Problems Routenplanung

Straßenkreuzung → Knoten

Straßenabschnitt → zwei entgegengerichtete Kanten

Einbahnstraßenabschnitt → in Fahrtrichtung gerichtete Kante

Fahrtzeit für Abschnitt e → Kantengewicht $w(e) \geq 0$

Straßennetz → gerichteter, gewichteter und zusammenhängender Graph $G = (V, E)$

Start → Knoten $s \in V$

Ziel → Knoten $t \in V$

Start-Ziel-Route → s - t -Weg, d.h. Folge von Kanten $(s, v_1), (v_1, v_2), \dots, (v_k, t)$ in G

Wozu kürzeste Wege?



Wozu kürzeste Wege? (II)

DB BAHN

Kontakt | Hilfe | Sitemap a a+ a++ Frage oder Suchbegriff eingeben ... Suchen

Startseite | Angebotsberatung | **Fahrplan & Buchung** | Services | BahnCard | Urlaub Meine Bahn

Suche Auswahl Ticket&Reservierung Zahlung Buchung Bestätigung → Neue Anfrage

Reisedaten 1 Erwachsener, 2. Klasse

Hinfahrt von Würzburg Hbf Di, 12.01.10 10:23 Abfahrt Ankunft → weitere Angaben ändern Aktualisieren

nach Mathematisches Institut, Würzburg

Kurzfristige Fahrplanänderungen
Informieren Sie sich hier über aktuelle Verkehrsmeldungen.
→ Weitere Informationen

Ihre Hinfahrtmöglichkeiten - sortiert nach **Abfahrt** Druckansicht

Bahnhof/Haltestelle	Datum	Zeit	Dauer	Umst.	Produkte	Normalpreis
		↑ Früher				
Würzburg Busbahnhof Mathematisches Institut, Würzburg	Di, 12.01.10	ab 10:29	0:16	0	Bus	Preisauskunft nicht möglich
	Di, 12.01.10	an 10:45				→ Rückfahrt hinzufügen
Würzburg Busbahnhof Mathematisches Institut, Würzburg	Di, 12.01.10	ab 10:49	0:16	0	Bus	Preisauskunft nicht möglich
	Di, 12.01.10	an 11:05				→ Rückfahrt hinzufügen
Würzburg Busbahnhof Mathematisches Institut, Würzburg	Di, 12.01.10	ab 11:09	0:16	0	Bus	Preisauskunft nicht möglich
	Di, 12.01.10	an 11:25				→ Rückfahrt hinzufügen
Details für alle anzeigen		↓ Später				

MobilCheck **UmweltMobilCheck**

Zurück

Was ist das Problem?

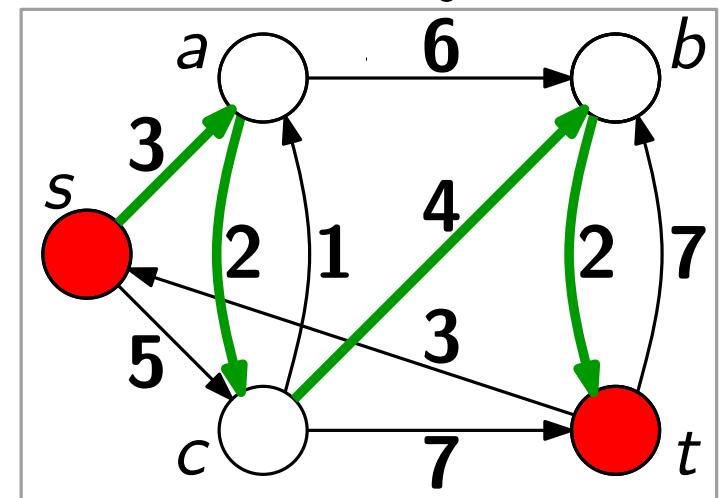
Eingabe:

- gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen **Kantengewichten** $w: E \rightarrow \mathbb{Q}_0^+$,
- Knoten s und t

Ausgabe:

- kürzester $s-t$ -Weg W in G , d.h. $\sum_{e \in W} w(e)$ minimal.

Darstellung durch Vorgänger-Zeiger π : für jeden Knoten v sei $\pi(v) \in V \cup \{nil\}$ Vorgänger von v auf kürzestem $s-v$ -Weg.



Was ist das Problem?

Eingabe:

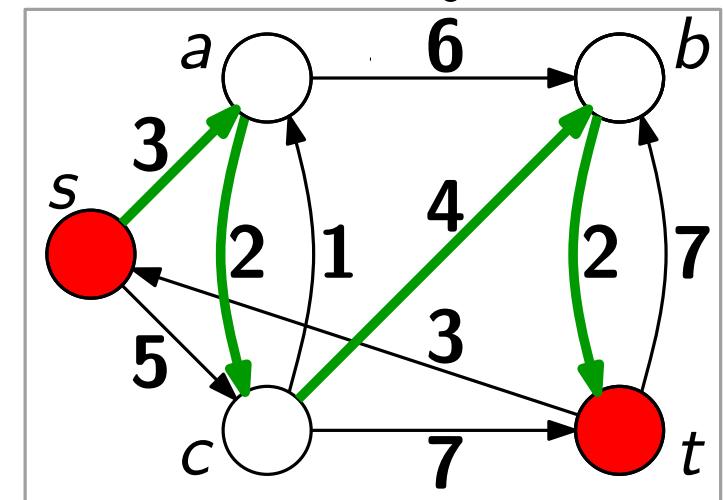
- gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen **Kantengewichten** $w: E \rightarrow \mathbb{Q}_0^+$,
- Knoten s und t

Ausgabe:

- kürzester $\cancel{s-t}$ -Wege W_t in G , $\underbrace{\text{für alle } t \in V}_{\text{d.h. } \sum_{e \in W} w(e) \text{ minimal.}}$

Darstellung durch Vorgänger-Zeiger π : für jeden Knoten v sei $\pi(v) \in V \cup \{nil\}$ Vorgänger von v auf kürzestem $s-v$ -Weg.

Nebenbemerkung: Analoges Berechnungsverfahren?



Dijkstra – BFS mit Gewichten

Dijkstra(WeightedGraph $G = (V, E; w)$, Vertex s)

Initialize(G, s) // Gewichtung

$Q = \text{new PriorityQueue}(V, d)$

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{ExtractMin}()$

foreach $v \in \text{Adj}[u]$ **do**

 Relax($u, v; w$)

$u.\text{color} = \text{black}$

Relax($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

$v.\text{color} = \text{gray}$

$v.d = u.d + w(u, v)$

$v.\pi = u$

$Q.\text{DecreaseKey}(v, v.d)$

BFS(Graph G , Vertex s)

Initialize(G, s)

$Q = \text{new Queue}()$

$Q.\text{Enqueue}(s)$

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{Dequeue}()$

foreach $v \in \text{Adj}[u]$ **do**

if $v.\text{color} == \text{white}$ **then**

$v.\text{color} = \text{gray}$

$v.d = u.d + 1$

$v.\pi = u$

$Q.\text{Enqueue}(v)$

$u.\text{color} = \text{black}$

Dijkstra – ein Beispiel

Dijkstra(WeightedGraph G , Vertex s)

Initialize(G, s)

$Q = \text{new PriorityQueue}(V, d)$

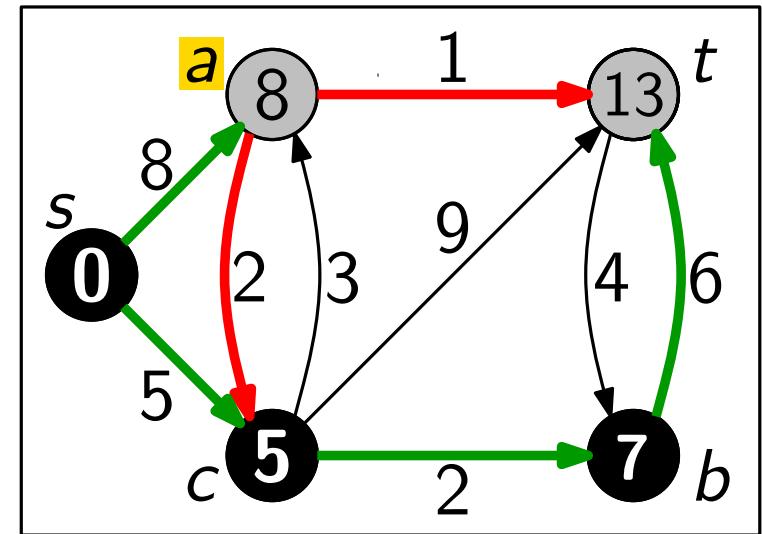
while not $Q.\text{Empty}()$ **do**

$u = Q.\text{ExtractMin}()$

foreach $v \in \text{Adj}[u]$ **do**

 └ Relax($u, v; w$)

 └ $u.\text{color} = \text{black}$



Relax($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

$v.\text{color} = \text{gray}$

$v.d = u.d + w(u, v)$

$v.\pi = u$

$Q.\text{DecreaseKey}(v, v.d)$

Initialize(Graph G , Vertex s)

foreach $u \in V$ **do**

$u.\text{color} = \text{white}$

$u.d = \infty$

$u.\pi = \text{nil}$

$s.\text{color} = \text{gray}$

$s.d = 0$

Dijkstra – ein Beispiel

Dijkstra(WeightedGraph G , Vertex s)

Initialize(G, s)

$Q = \text{new PriorityQueue}(V, d)$

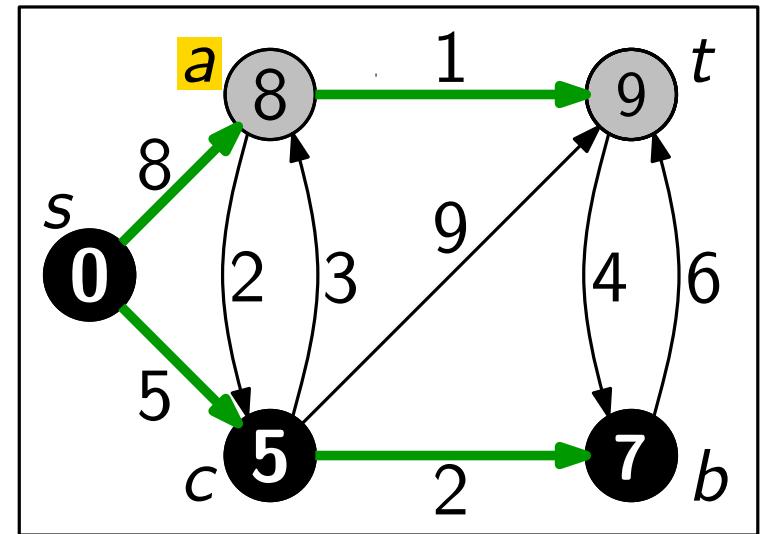
while not $Q.\text{Empty}()$ **do**

$u = Q.\text{ExtractMin}()$

foreach $v \in \text{Adj}[u]$ **do**

 └ Relax($u, v; w$)

 └ $u.\text{color} = \text{black}$



Relax($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

$v.\text{color} = \text{gray}$

$v.d = u.d + w(u, v)$

$v.\pi = u$

$Q.\text{DecreaseKey}(v, v.d)$

Initialize(Graph G , Vertex s)

foreach $u \in V$ **do**

$u.\text{color} = \text{white}$

$u.d = \infty$

$u.\pi = \text{nil}$

$s.\text{color} = \text{gray}$

$s.d = 0$

Dijkstra – ein Beispiel

Dijkstra(WeightedGraph G , Vertex s)

 Initialize(G, s)

$Q = \text{new PriorityQueue}(V, d)$

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{ExtractMin}()$

foreach $v \in \text{Adj}[u]$ **do**

 └ Relax($u, v; w$)

 └ $u.\text{color} = \text{black}$

Relax($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

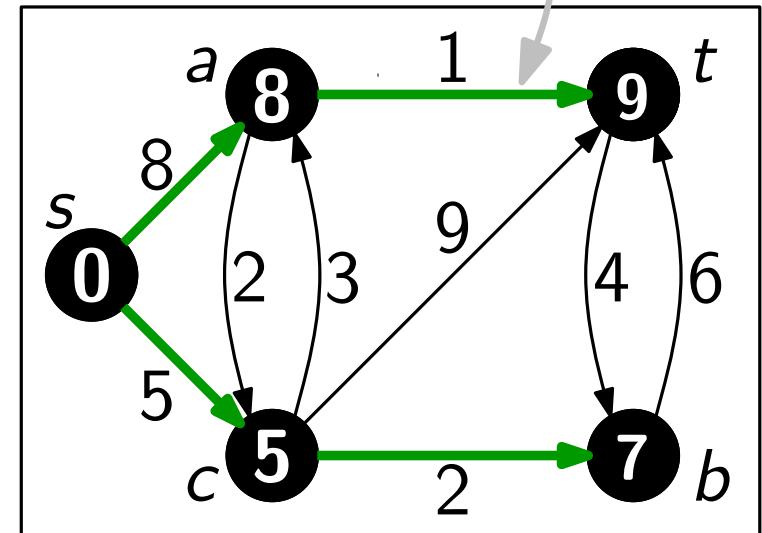
$v.\text{color} = \text{gray}$

$v.d = u.d + w(u, v)$

$v.\pi = u$

$Q.\text{DecreaseKey}(v, v.d)$

Kürzester-Wege-Baum
mit Wurzel s



Initialize(Graph G , Vertex s)

foreach $u \in V$ **do**

$u.\text{color} = \text{white}$

$u.d = \infty$

$u.\pi = \text{nil}$

$s.\text{color} = \text{gray}$

$s.d = 0$

Dijkstra – die Laufzeit

Dijkstra(WeightedGraph G , Vertex s)

Abk. für $O(|V|)$

Initialize(G, s)

$Q = \text{new PriorityQueue}(V, d)$

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{ExtractMin}()$

foreach $v \in \text{Adj}[u]$ **do**

 Relax($u, v; w$)

$u.\text{color} = \text{black}$

$O(V)$ Zeit

genau $|V|$ mal

Relax($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

$v.\text{color} = \text{gray}$

$v.d = u.d + w(u, v)$

$v.\pi = u$

$Q.\text{DecreaseKey}(v, v.d)$

Wie oft wird Relax aufgerufen?

Für jeden Knoten $u \in V$ genau $|\text{Adj}[u]| = \deg u^{(\text{out}-)}$ mal,

also insg. $\Theta(E)$ mal.

Also wird DecreaseKey $O(E)$ mal aufgerufen.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijstras Alg. in $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(V , E)$
als unsortiertes Feld	$O(n)$	$O(1)^*$	$O(V^2 + E)$
als Heap	$O(\log n)$	$O(\log n)^{**}$	$O((E + V) \log V)$
als Fibonacci-Heap	$O(\log n)$ <i>amortisiert</i>	$O(1)$ <i>amortisiert</i>	$O(E + V \log V)$ <i>im Worst-Case!</i>

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz (\rightarrow Direktzugriff)
 **) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Korollar. In einem Graphen $G = (V, E; w)$ mit $w: E \rightarrow \mathbb{Q}_{\geq 0}$ kann man in $O(E + V \log V)$ Zeit die kürzesten Wege von einem zu allen Knoten berechnen (SSSP-Problem).

Dijkstra – die Korrektheit

siehe [CLRS], Kapitel 24.3., Satz 24.6:
Korrektheisbeweis mittels Schleifeninvariante.

oder

MIT-Vorlesungsmitschnitt von Erik Demaine:
http://videolectures.net/mit6046jf05_demaine_lec17

Wozu kürzeste Wege? (III) – SMSen

G**H**I

M**N**O

D**E**F

M**N**O

P**Q**R**S**

M**N**O

A**B**C

T**U**V

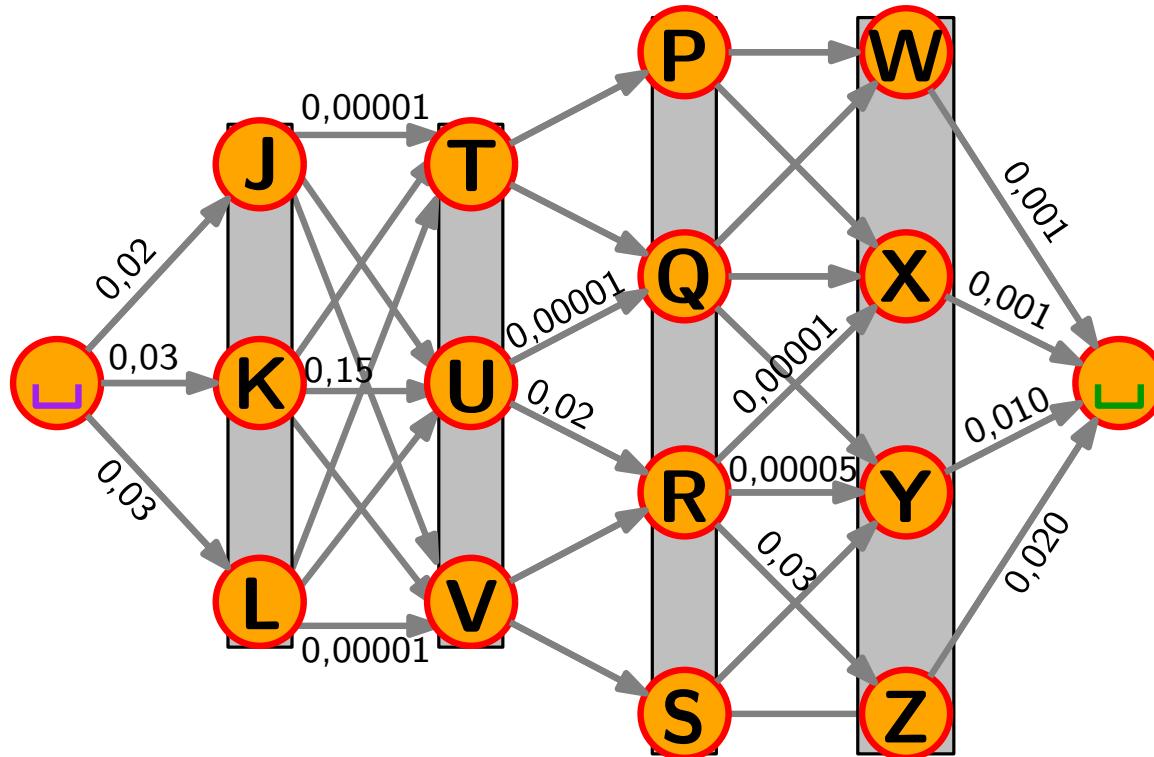
G**H**I

J**K**L

10:21 für T9



Modellierung – SMSen



- Graph:** Knoten $\hat{=}$ Buchstaben
 Kanten $\hat{=}$ aufeinanderfolgende Buchst.
 Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten
- Gesucht:** Weg P von $\textcolor{violet}{\square}$ nach $\textcolor{green}{\square}$ mit *größter WK* ($= \prod_{e \in P} w(e)$)
- Lösung:** *dynamisches Programmieren... [kommt noch!]*

Literatur

- **A note on two problems in connexion with graphs.**

Edsger Wybe Dijkstra:

Numerische Mathematik (1)

1959, S. 269–271.

Lesen Sie
mal rein!

- **Das Geheimnis des kürzesten Weges.**

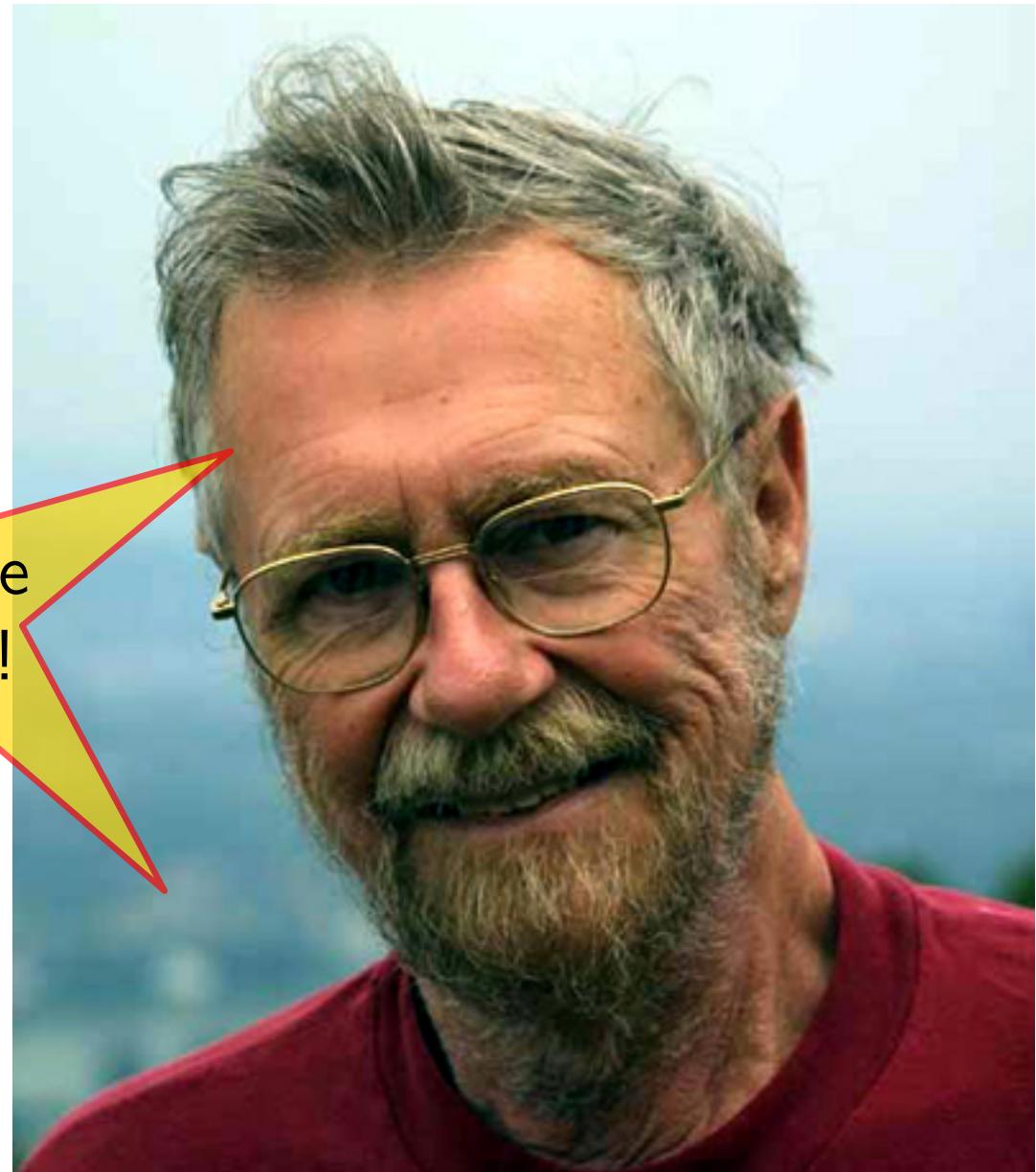
Ein mathematisches Abenteuer.

Peter Gritzmann und

René Brandenberg:

Springer-Verlag, 3. Aufl., 2005.

Beide Werke sind über die UB
frei zugänglich und über unsere
WueCampus-Seite verlinkt!



Edsger Wybe Dijkstra
* 1930 in Rotterdam
† 2002 in Nuenen, Niederlande

Kürzeste Wege nach Dijkstra

nicht-neg. Kantengew.	Dijkstra	$O(E + V \log V)$	✓
ungewichteter Graph	Breitensuche	$O(E + V)$	✓
azyklischer Graph	topol. Sortieren	$O(E + V)$	Nächstes Mal! ✓
negative Kantengew.	Bellman-Ford	$O(EV)$	Vorlesung Alg. Graphentheorie (?)
für alle Knotenpaare + negative Kantengew.	$V \times$ Dijkstra Floyd-Warshall Johnson	$O(V(E + V \log V))$ $O(V^3)$ $O(V(E + V \log V))$	✓
k kürzeste $s-t$ -Wege	Eppstein	$O(k + E + V \log V)$	

Algorithmen und Datenstrukturen

Wintersemester 2018/19
20. Vorlesung

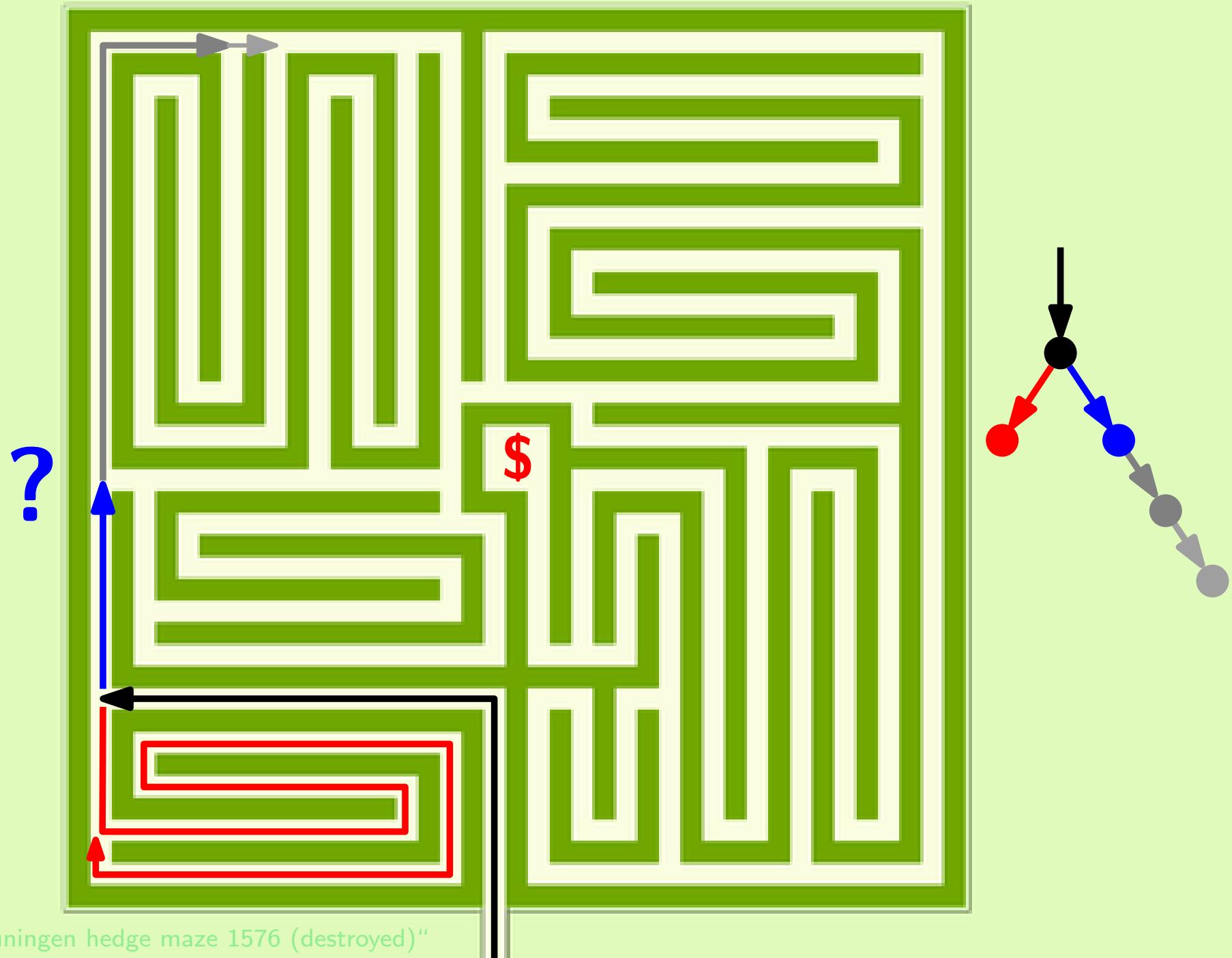
Tiefensuche und topologische Sortierung

Themen für den 3. Kurztest (Do, 24.01.18)

- Rot-Schwarz-Bäume (R-S-Eigenschaften, Höhe)
- Augmentieren von Datenstrukturen
- Amortisierte Analyse
- Nächstes Paar (Teile und Herrsche)
- Graphen und Breitensuche

Anmeldung

Ab sofort bis Di, 22.1., 13:00 Uhr.



„Maze-01 Grüningen hedge maze 1576 (destroyed)“

von RTH – Eigenes Werk. Lizenziert unter CC BY-SA 3.0 über Wikimedia Commons

Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe:

- Besuchsintervalle ($u.d / u.f$)



- Klassifizierung der Graphkanten:

- Baumkanten (Kanten von G_π)

Kanten des DFS-Waldes (entgegen π gerichtet)

- Rückwärtskanten (R)

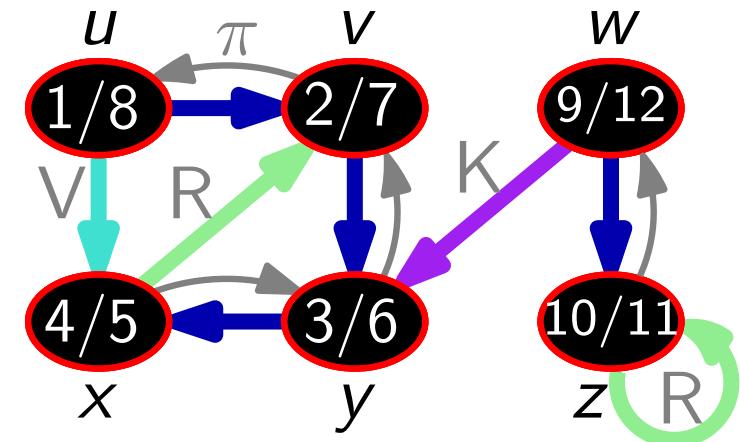
Nicht-Baumkanten zu einem Vorgängerknoten

- Vorwärtskanten (V)

Nicht-Baumkanten zu einem Nachfolgerknoten

- Kreuzkanten (K)

Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.



Farbe Zielknoten:

weiss

grau

schwarz und
start.d < ziel.d

schwarz und
start.d > ziel.d

Tiefensuche – Pseudocode

```

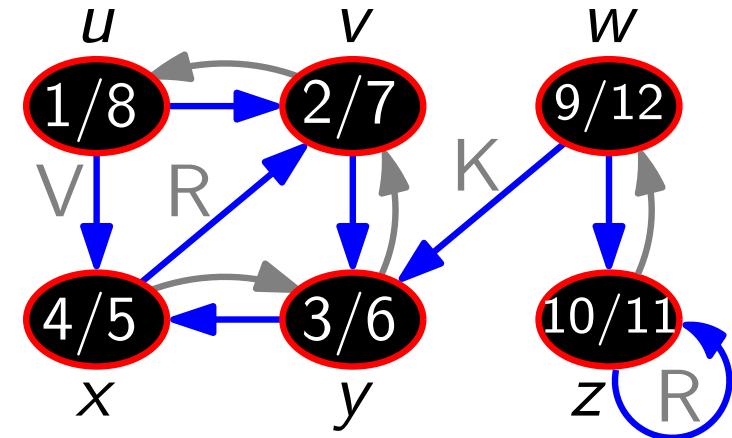
DFS(Graph  $G = (V, E)$ )
foreach  $u \in V$  do
     $u.color = \text{white}$ 
     $u.\pi = \text{nil}$ 
 $time = 0$  // globale Variable!
foreach  $u \in V$  do
    if  $u.color == \text{white}$  then DFSVisit( $G, u$ )

```

```

DFSVisit(Graph  $G$ , Vertex  $u$ )
 $time = time + 1$ 
 $u.d = time; u.color = \text{gray}$ 
foreach  $v \in \text{Adj}[u]$  do
    if  $v.color == \text{white}$  then
         $v.\pi = u$ ; DFSVisit( $G, v$ )
 $time = time + 1$ 
 $u.f = time; u.color = \text{black}$ 

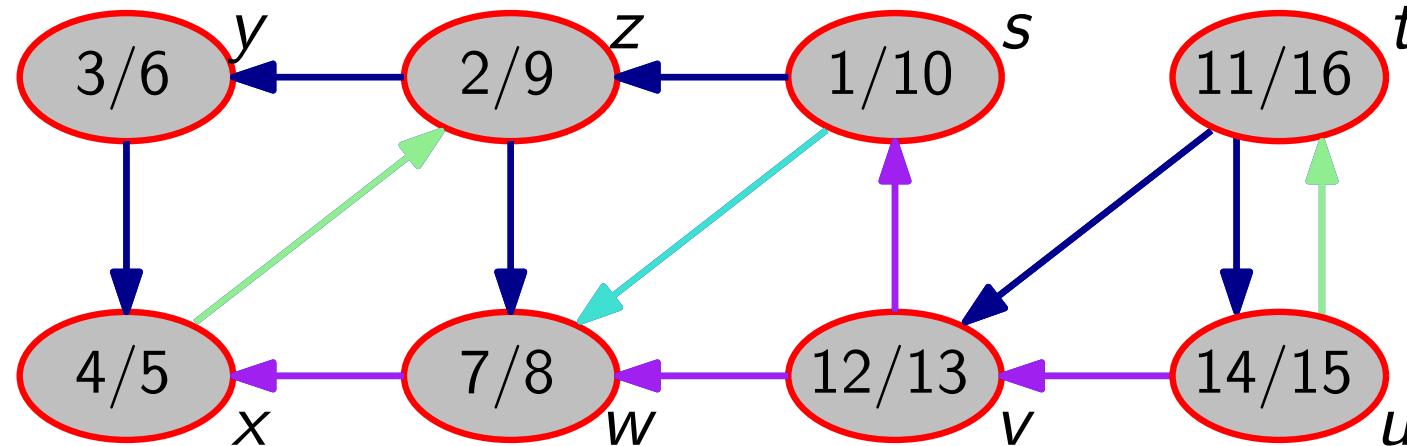
```



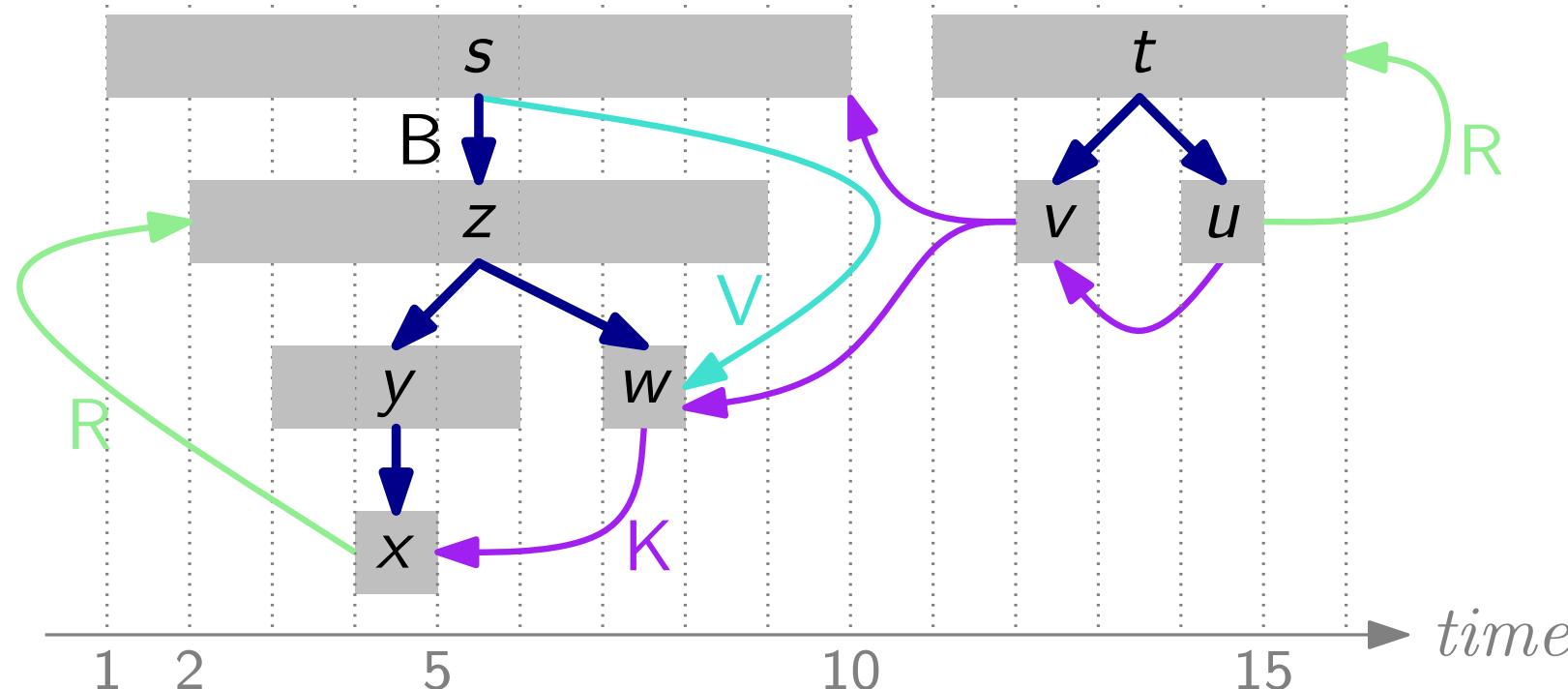
**Laufzeit
von DFS?**

- DFSVisit wird nur für weiße Knoten aufgerufen.
 - In DFSVisit wird der neue Knoten sofort grau gefärbt.
⇒ DFSVisit wird für jeden Knoten genau 1× aufgerufen.
 - DFS ohne if $O(V)$ Zeit
DFSVisit ohne Rek. $O(\deg u)$
- | | |
|------------|-----------------|
| DFS gesamt | $O(V + E)$ Zeit |
|------------|-----------------|

Tiefensuche – Eigenschaften



$$\left(s \left(z \left(y \left(x \; x \right) y \right) \left(w \; w \right) z \right) s \right) \left(t \left(v \; v \right) \left(u \; u \right) t \right)$$



Tiefensuche – Analyse

Satz.

(Klammerntheorem)

Nach $\text{DFS}(G)$ gilt für $\{u, v\} \in \binom{V}{2}$ genau eine der Bedingungen

- (i) Besuchsintervalle disjunkt und
Baumkanten enthalten weder u - v - noch v - u -Weg.
- (ii) $[u.d, u.f] \subset [v.d, v.f]$ und Baumkanten enthalten v - u -Weg.
- (iii) Wie (ii), nur umgekehrt.

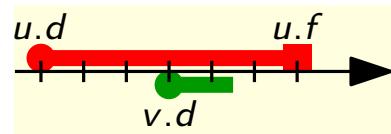
```

DFSVisit(Graph G, Vertex u)
  time = time + 1
  u.d = time; u.color = gray
  foreach v ∈ Adj[u] do
    if v.color == white then
      v.π = u; DFSVisit(G, v)
  time = time + 1
  u.f = time; u.color = black

```

Beweis. Wir betrachten zwei Fälle.

1. Fall: $u.d < v.d$.



A) $v.d < u.f$, d.h. v wurde entdeckt, als u noch grau war.
 $\Rightarrow v$ ist *Nachfolger* von u , d.h. es gibt einen u - v -Weg.

Wegen $u.d < v.d$ gilt: v wurde später als u entdeckt.

\Rightarrow alle Kanten, die v verlassen, sind erforscht;
 v wird schwarz, bevor DFS zu u zurückkehrt und u schwarz macht $\Rightarrow [v.d, v.f] \subset [u.d, u.f]$, d.h. (ii)



Tiefensuche – Analyse

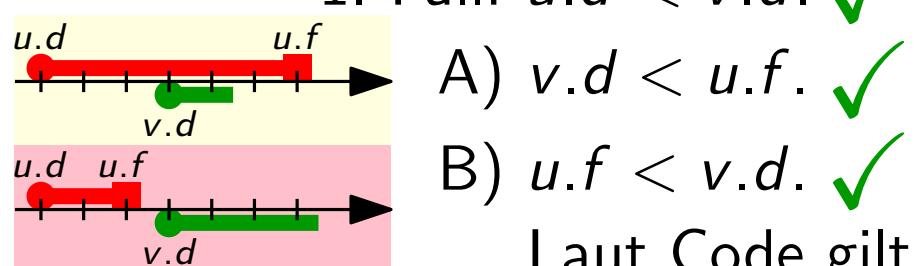
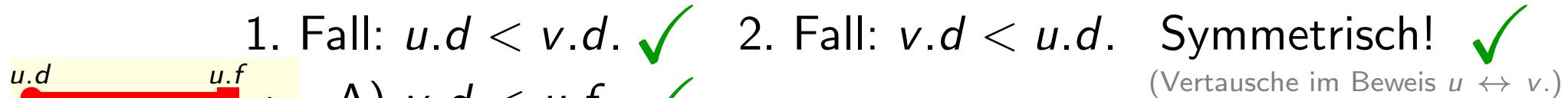
Satz.

(Klammerntheorem)

Nach $\text{DFS}(G)$ gilt für $\{u, v\} \in \binom{V}{2}$ genau eine der Bedingungen

- (i) Besuchsintervalle disjunkt und
Baumkanten enthalten weder u - v - noch v - u -Weg.
- (ii) $[u.d, u.f] \subset [v.d, v.f]$ und Baumkanten enthalten v - u -Weg.
- (iii) Wie (ii), nur umgekehrt.

Beweis. Wir betrachten zwei Fälle.



Laut Code gilt außerdem $u.d < u.f < v.d < v.f$

$$\Rightarrow [u.d, u.f] \cap [v.d, v.f] = \emptyset$$

(i)

⇒ Keiner der beiden Knoten wurde entdeckt, während
der andere noch grau war, d.h. keiner Nachf. des anderen. ↑

```

DFSVisit(Graph G, Vertex u)
  time = time + 1
  u.d = time; u.color = gray
  foreach v ∈ Adj[u] do
    if v.color == white then
      v.π = u; DFSVisit(G, v)
  time = time + 1
  u.f = time; u.color = black

```

Tiefensuche in ungerichteten Graphen

Satz. G ungerichtet

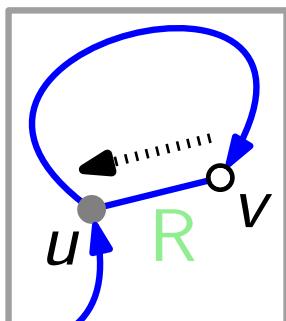
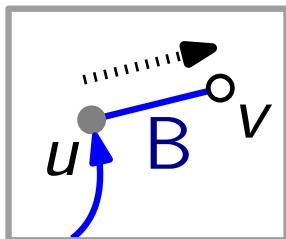
$\Rightarrow G$ hat nur Baum- und Rückwärtskanten.

Beweis. Sei uv (kurz für $\{u, v\}$) eine beliebige Kante von G .

O.B.d.A. gilt $u.d < v.d$.

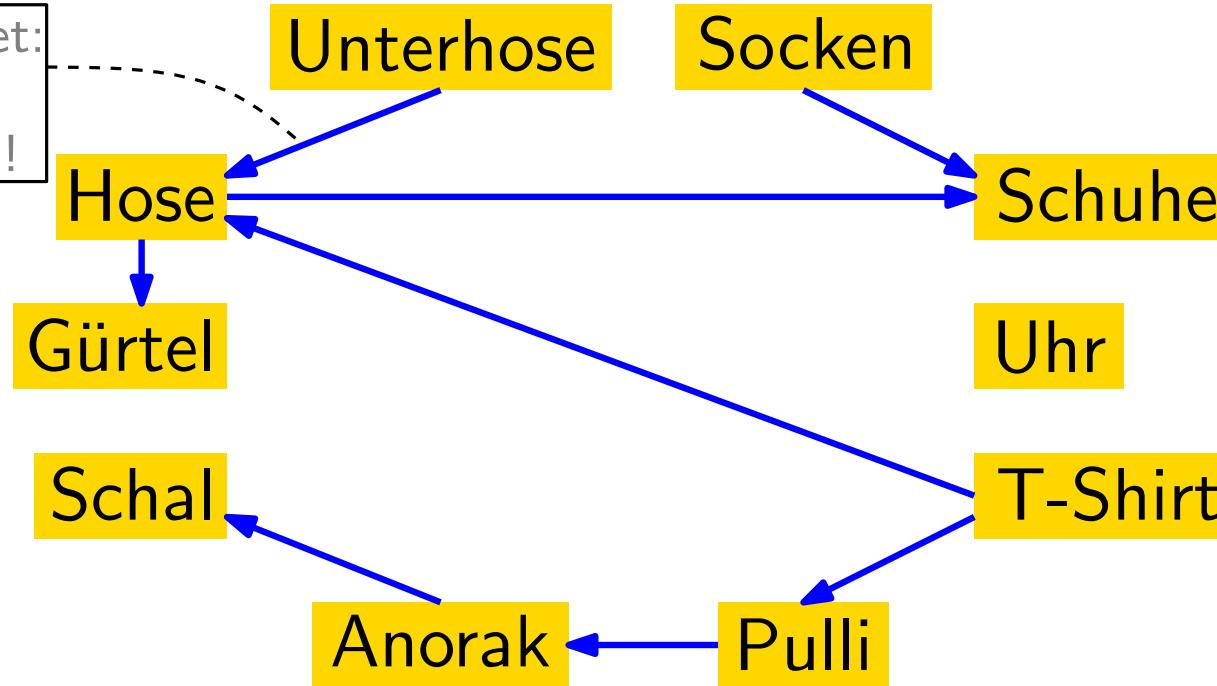
Dann entdeckt DFS v und färbt v schwarz, bevor u schwarz gefärbt wird (da $v \in \text{Adj}[u]$).

- Falls DFS uv zum ersten Mal von u nach v überschreitet, ist v zu diesem Zeitpunkt *weiss*. Dann ist uv Baumkante.
- Andernfalls wird uv zum ersten Mal von v nach u überschritten. Dann ist uv R-Kante, da u dann schon (und immer noch) *grau* ist.



Ablaufplanung

Kante bedeutet:
Unterhose *vor*
Hose anziehen!

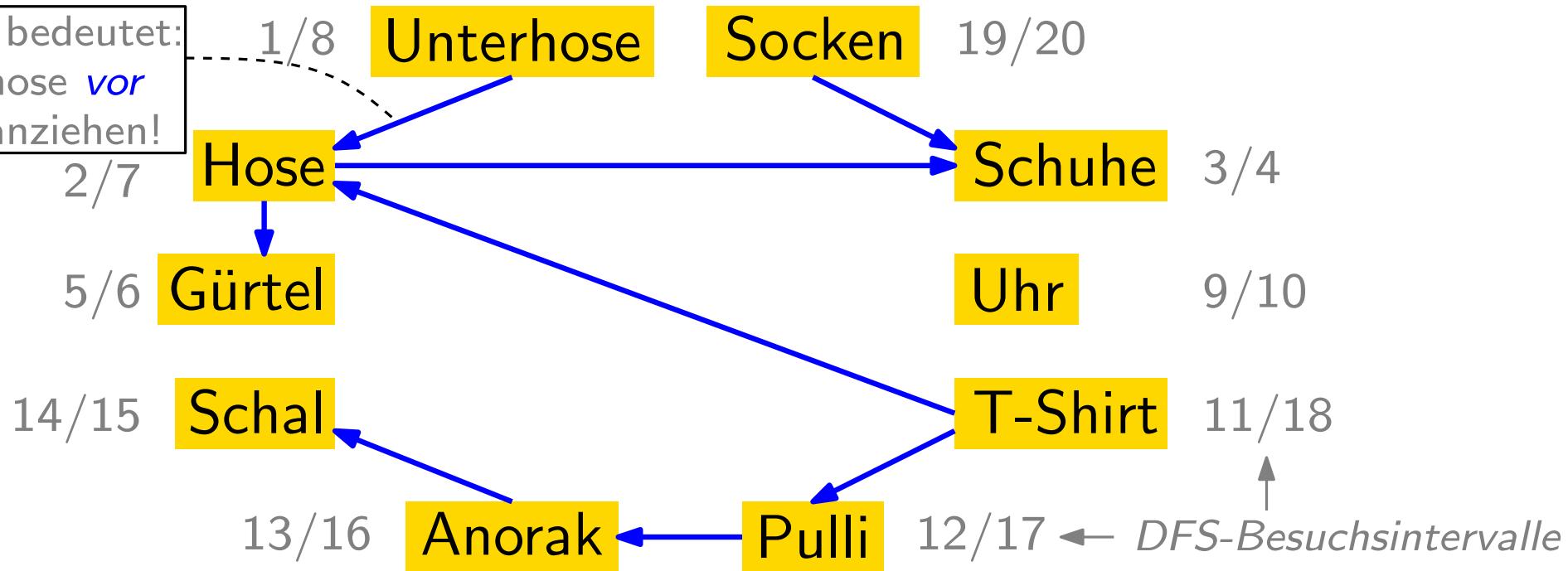


Aufgabe: Finde Ablaufplan –
d.h. Reihenfolge der Knoten, so dass alle Einschränkungen erfüllt sind (z.B. T-Shirt vor Pulli).

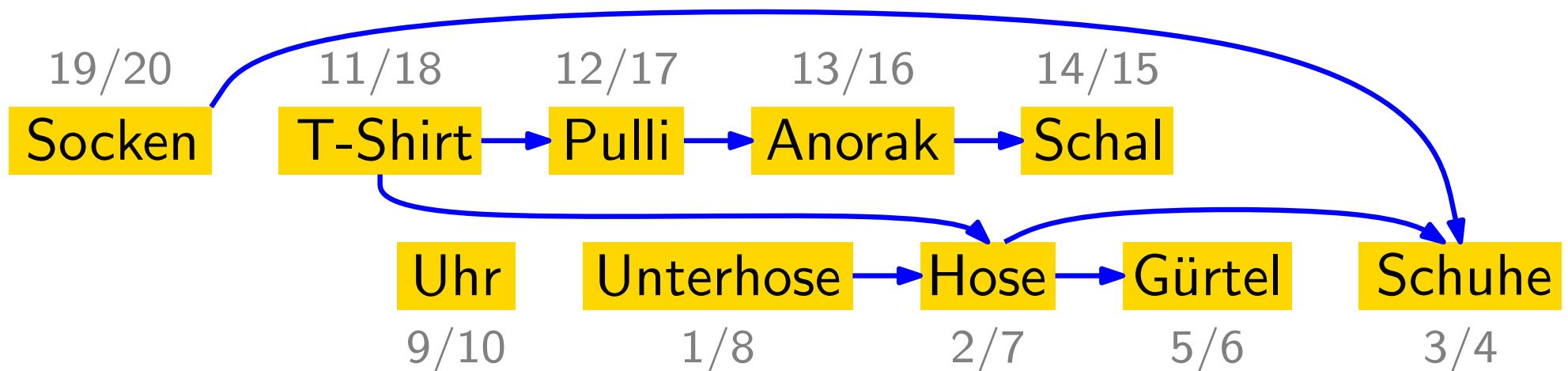
Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v .

Ablaufplanung

Kante bedeutet:
Unterhose *vor*
Hose anziehen!



Idee: Nutze Tiefensuche! \Rightarrow Alle Kanten sind nach rechts gerichtet.
Sortiere Knoten nach absteigenden f -Zeiten.



Topologisch sortieren

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v .

TopologicalSort(DirectedGraph G)

$L = \text{new List}()$

$\text{DFS}(G)$ mit folgender Änderung:

Wenn ein Knoten schwarz gefärbt wird,
häng ihn vorne an die Liste L an.

return L

Laufzeit?

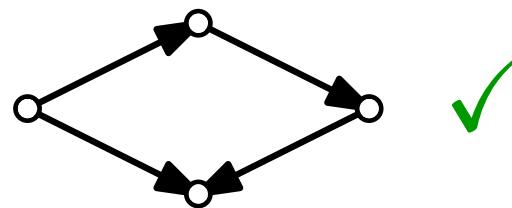
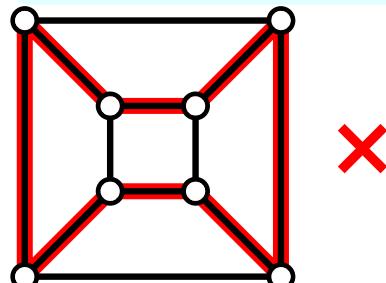
$O(V + E)$

Korrekt?

Wann funktioniert's?

Def.

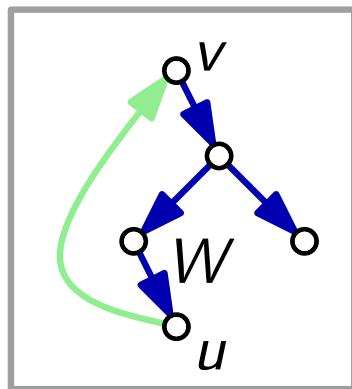
Ein (gerichteter) Graph ist *kreisfrei*,
wenn er keinen (gerichteten) Kreis enthält.



Kreisfrei \Leftrightarrow keine R-Kanten

Lem. Ein gerichteter Graph G ist kreisfrei
 $\Leftrightarrow \text{DFS}(G)$ liefert keine Rückwärtskanten.

Beweis. „ \Rightarrow “ Sei G kreisfrei.



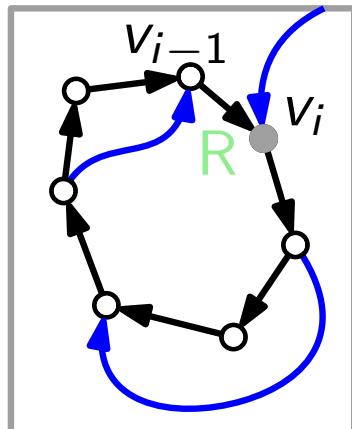
Angenommen $\text{DFS}(G)$ liefert R-Kante (u, v) .

Dann ist u Nachfolger von v im DFS-Wald.

D.h. G enthält einen gerichteten $v-u$ -Weg W .

Aber dann ist $W \oplus (u, v)$ ein gerichteter Kreis. ↗

„ \Leftarrow “ $\text{DFS}(G)$ liefere keine R-Kanten.



Ang. G enthält trotzdem Kreis $C = \langle v_1, \dots, v_k \rangle$.

Sei v_i der 1. Knoten in C , den $\text{DFS}(G)$ erreicht.

Es gibt einen Weg von v_i nach v_{i-1} in G .

\Rightarrow DFS gelangt zu v_{i-1} , solange v_i grau ist.

$\Rightarrow (v_{i-1}, v_i)$ ist R-Kante. ↗

□

Korrektheit von TopologicalSort

Satz. Sei G ein gerichteter kreisfreier Graph. Dann liefert $\text{TopologicalSort}(G)$ eine topologische Sortierung von G .

Beweis. Sei $L = \langle v_n, v_{n-1}, \dots, v_1 \rangle = \text{TopologicalSort}(G)$.

Dann gilt $v_1.f < v_2.f < \dots < v_n.f$.

Sei (u, v) Kante von G . Zu zeigen: $v.f < u.f$

Welche Farbe hat v , wenn DFS (u, v) überschreitet?

- | | | |
|--|--|--|
|  | v grau $\Rightarrow (u, v)$ ist R-Kante | ⚡ Widerspruch zu Lemma: G kreisfrei! |
|  | v weiß $\Rightarrow v$ Nachfolger von $u \Rightarrow v.f < u.f$ | ✓ |
|  | v schwarz $\Rightarrow u.f$ noch nicht gesetzt, $v.f$ gesetzt
$\Rightarrow v.f < u.f$ | ✓ |

□

Vergleich Durchlaufstrategien für Graphen

	Breitensuche	Tiefensuche
Laufzeit	$O(V + E)$	$O(V + E)$
Ergebnis	BFS-Baum, d.h. kürzeste Wege	d - und f -Werte, z.B. für top. Sortierung
Datenstruktur	Schlange	Rekursion bzw. Stapel
Vorgehen	nicht-lokal	lokal

Algorithmen und Datenstrukturen

Wintersemester 2018/19
21. Vorlesung

Minimale Spannbäume

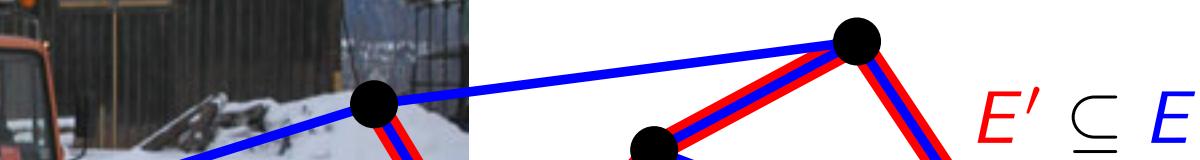
Motivation

*) Kantengewichte $w: E \rightarrow \mathbb{R}_{>0}$
**) $w(E') := \sum_{e \in E'} w(e)$

Gegeben: zusammenhängendes Straßennetz $G = (V, E; w^*)$, das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

- (1) man von jeder Stadt in G' zu jeder anderen kommen kann („ G' spannt G auf“) und
- (2) die „Schneeräumkosten“ $w(E')^{**}$ minimal sind unter allen Teilnetzen, die (1) erfüllen.



$$E' \subseteq E$$
$$G = (V, E; w)$$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow G'$ ist ein Wald.

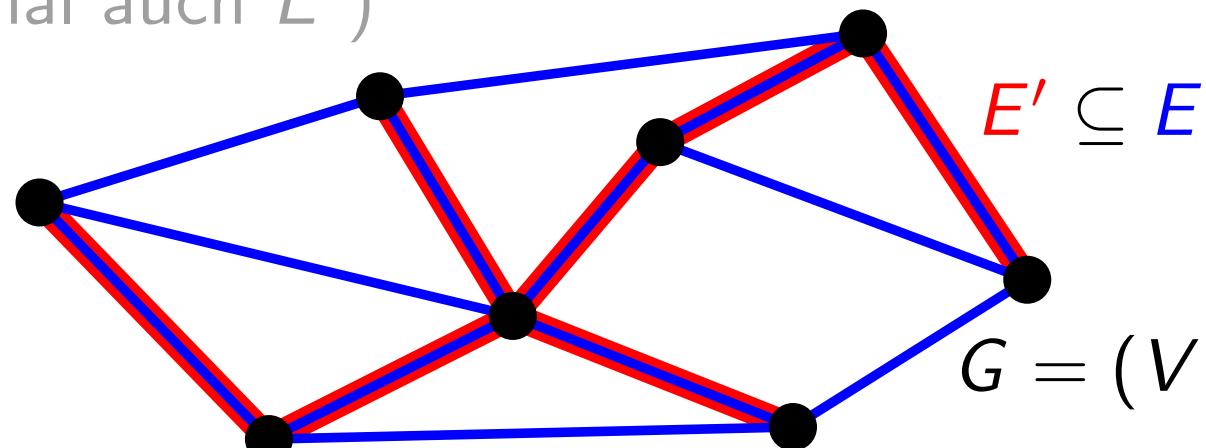
G' „erbt“ Zusammenhang von $G \Rightarrow G'$ Baum.

G' spannt G auf $\Rightarrow G'$ ist Spannbaum von G .

G' hat minimales Gewicht unter allen Spannbäumen von G .

Wir nennen G' kurz *minimalen Spannbaum* von G .

(manchmal auch E')



$$G = (V, E; w)$$

Beob. $|E'| = |V| - 1$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Generischer Min.-Spannbaum-Algorithmus

GenericMST(UndirectedConnectedGraph G , EdgeWeights w)

$A = \emptyset$

while $|A| < |V| - 1$ **do**

// Invariante: A ist Teilmenge eines min. Spannbaums von G

finde Kante uv , die **sicher** für A ist

$A = A \cup \{uv\}$

return A



Wir sagen uv ist *sicher* für A , falls Invariante für $A \cup \{uv\}$ gilt.

Beob. Dies ist ein sogenannter *Greedy-Algorithmus!*

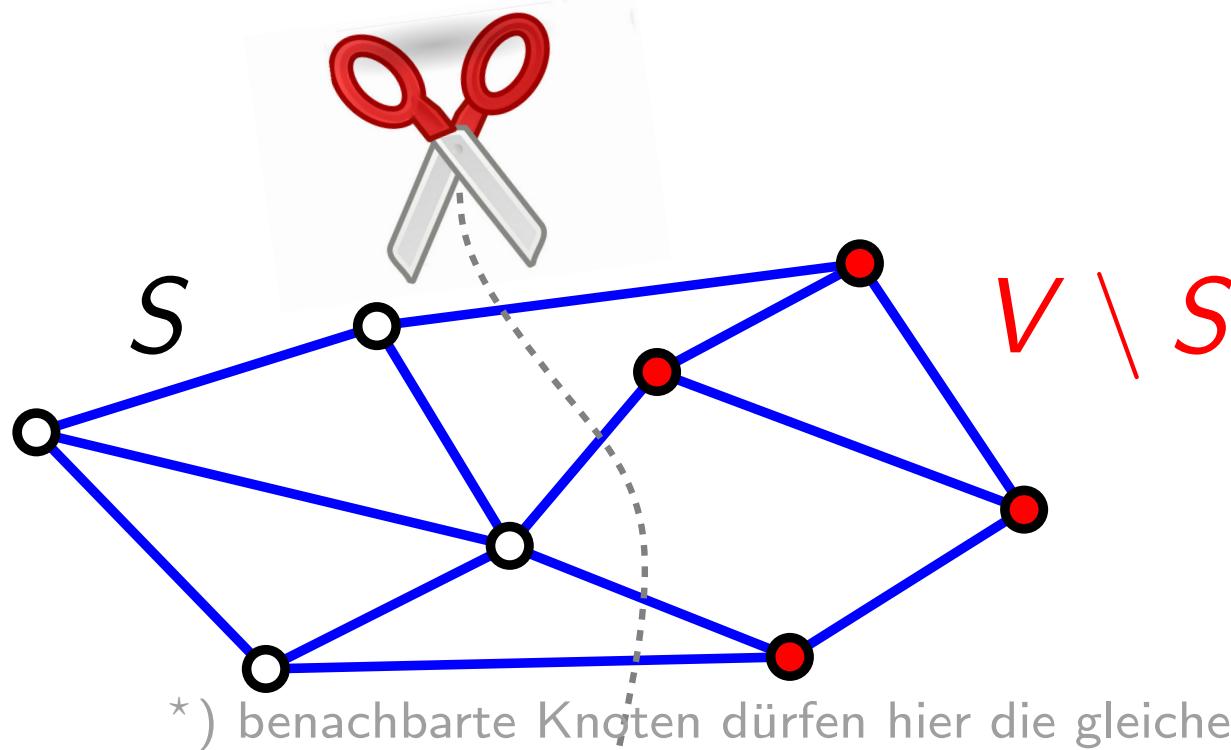
Frage: Gibt's überhaupt immer eine sichere Kante?

Antwort: Ja! – *Per Induktion!*

Frage: Aber wie findet man eine – ohne schon einen minimalen Spannbaum zu kennen?

Schnitte und leichte Kanten

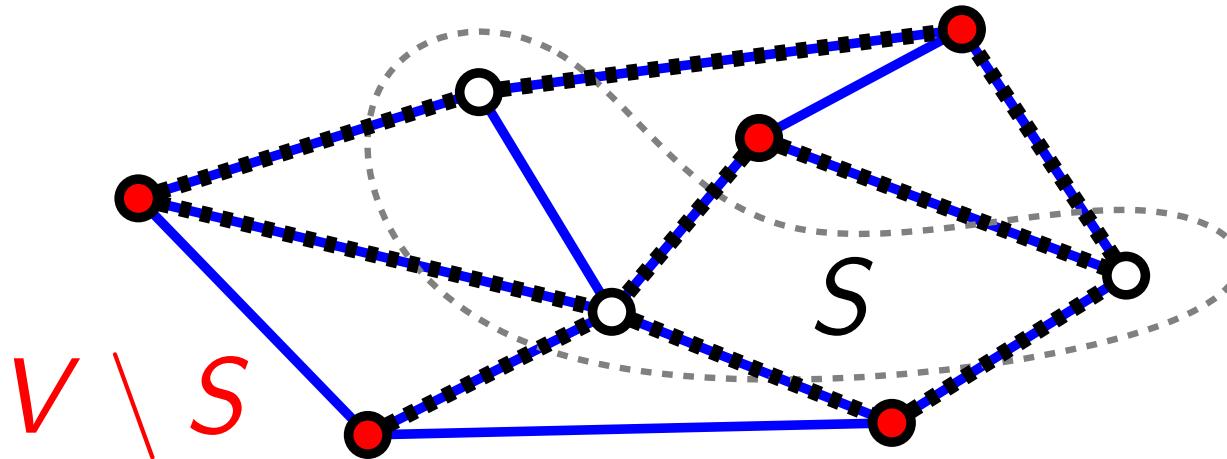
Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung^{*}) von V .



Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung^{*}) von V .

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

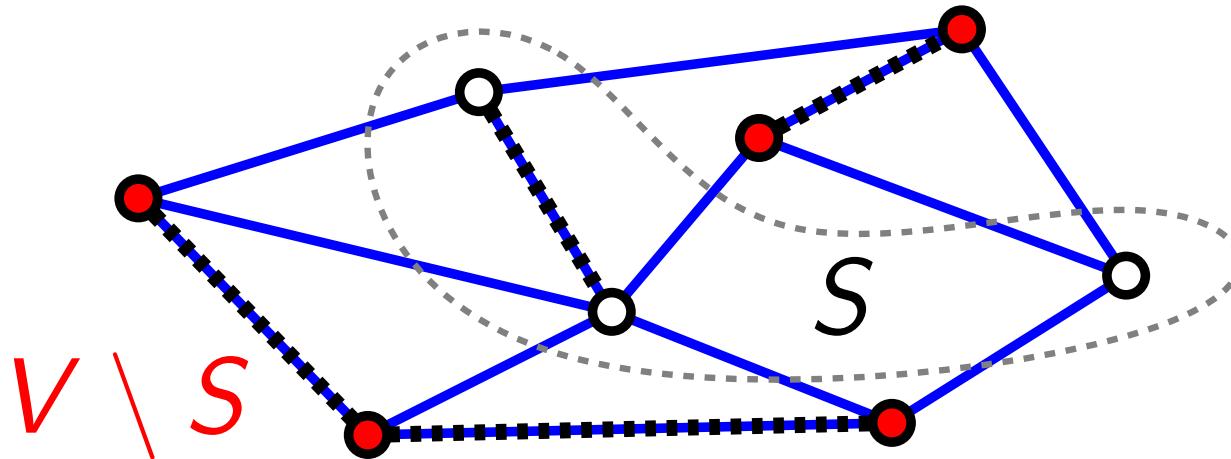


Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung^{*}) von V .

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt *respektiert* eine Kantenmenge A , wenn keine Kante in A den Schnitt kreuzt.



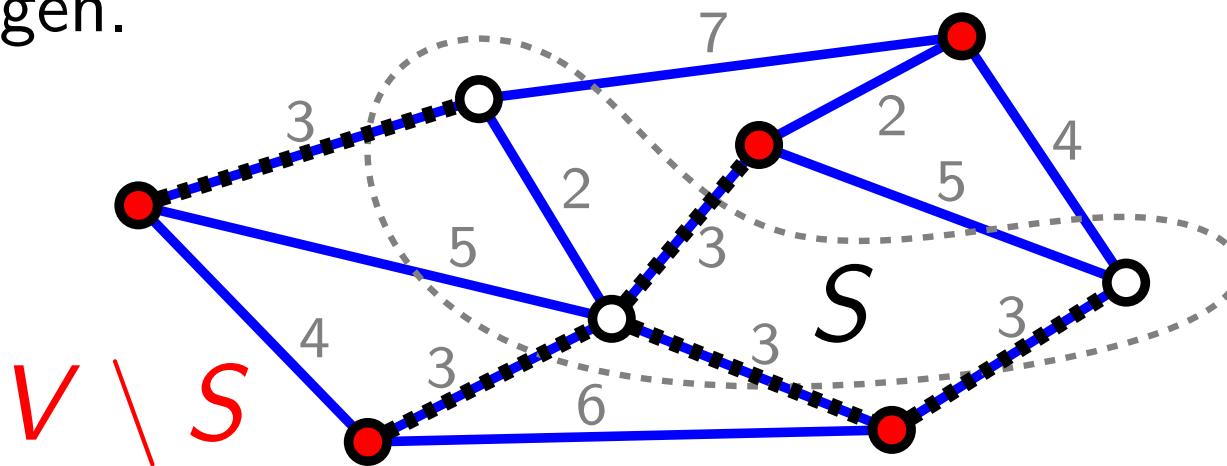
Schnitte und leichte Kanten

Def. Ein **Schnitt** $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung^{*}) von V .

Eine Kante e **kreuzt** $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt **respektiert** eine Kantenmenge A , wenn keine Kante in A den Schnitt kreuzt.

Eine Kante e , die einen Schnitt kreuzt, ist **leicht**, wenn alle Kanten, die den Schnitt kreuzen, mindestens $w(e)$ wiegen.



Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.

Sei T Kantenmenge eines min. Spannbaums von G .

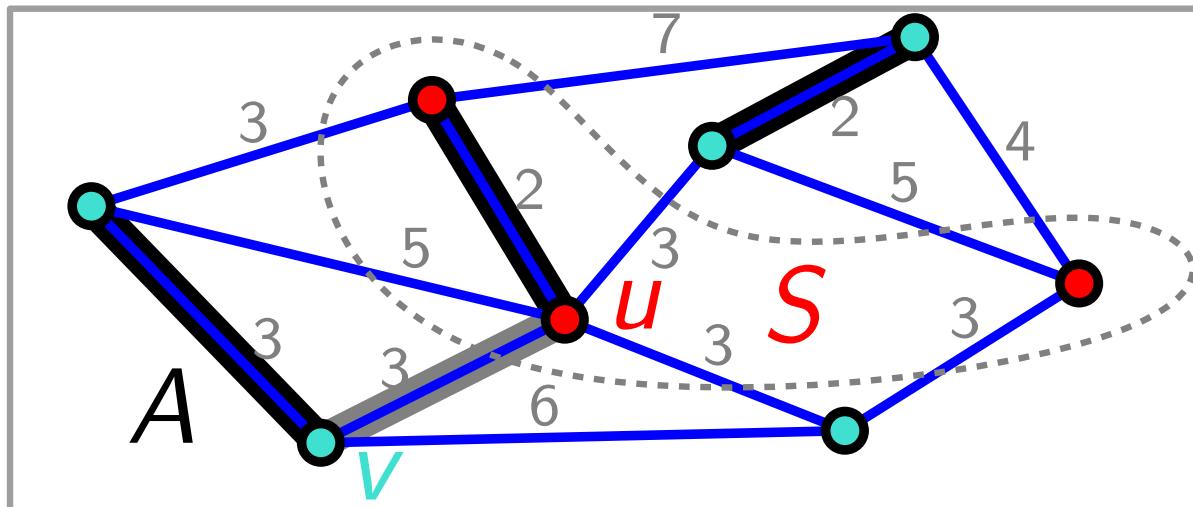
Sei A Teilmenge von T .

Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.

Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.

Dann ist uv sicher für A ,

d.h. G hat einen min. Spannbaum, der $A \cup \{uv\}$ enthält.



Beweis

Satz. ... Dann ist uv sicher für A .

Beweis. Zeige: G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.

Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u - v -Pfad in T .

$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)

\Rightarrow Kreis enthält zweite Kante xy , die $(S, V \setminus S)$ kreuzt.

$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G .

$$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0, \text{ da } uv \text{ leicht bzgl. } (S, V \setminus S)}$$

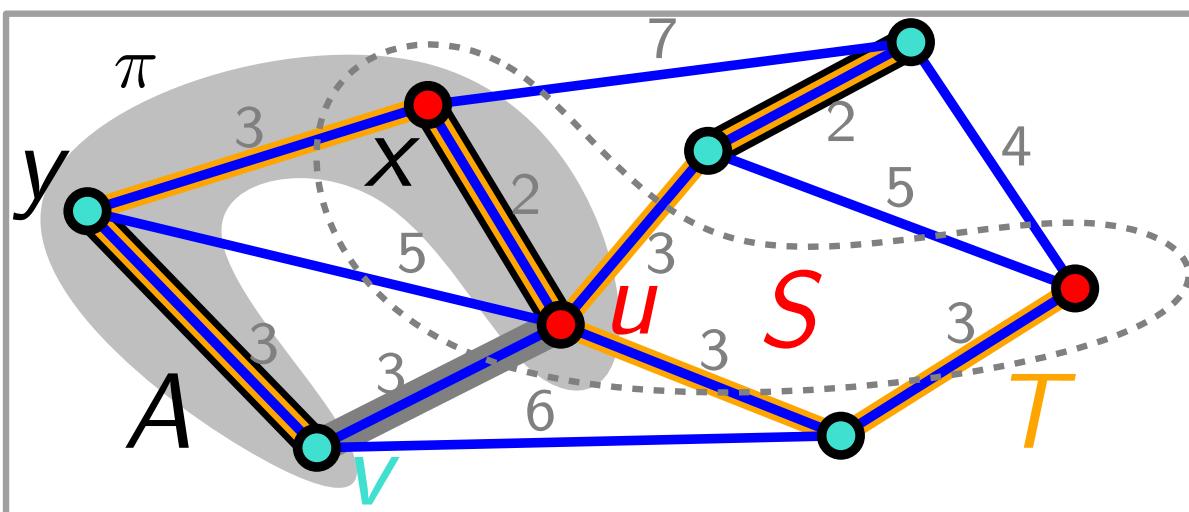
≤ 0 , da uv leicht bzgl. $(S, V \setminus S)$

$\Rightarrow T'$ ist *minimaler* Spannbaum von G .

Und: $A \cup \{uv\} \subseteq T'$.

$\Rightarrow uv$ ist sicher für A .

□



Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
 Sei T Kantenmenge eines min. Spannbaums von G .
 Sei A Teilmenge von T .
 Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
 Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
 Dann ist uv sicher für A .

GenericMST(UndirectedConnectedGraph G , EdgeWeights w)

$A = \emptyset$

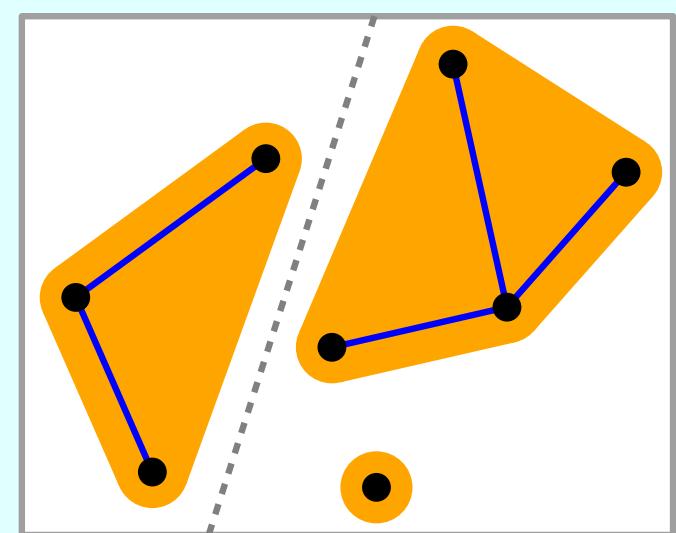
while $|A| < |V| - 1$ **do**

// INV: $A \subseteq$ min. Spannbaum von G

finde Kante uv , die *sicher* für A ist

$A = A \cup \{uv\}$

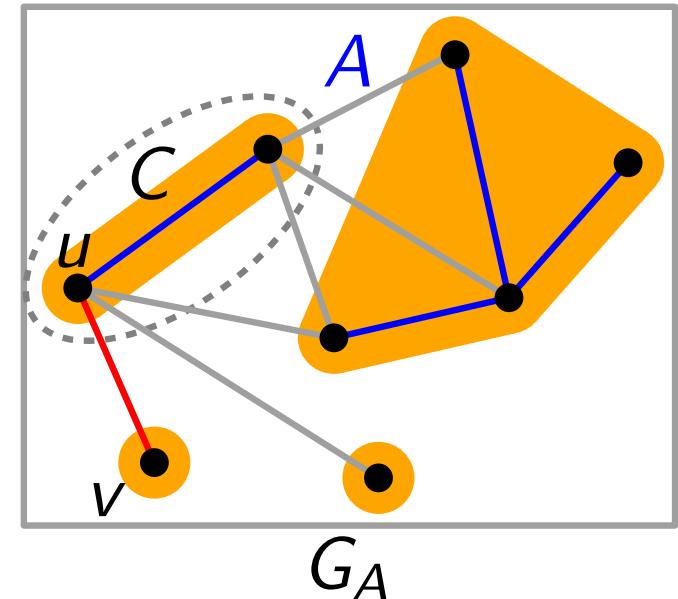
return A



Zusammenhangskomponenten

Def.

Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



Korollar.

$G = (V, E)$ wie gehabt.

$A \subseteq E$ in einem min. Spannbaum von G enthalten.

$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.

uv leicht bzgl. $(V_C, V \setminus V_C)$

Dann gilt: uv ist sicher für A .

Der Algorithmus von Jarník-Prim (1930/1957)

~~JarníkPrimMST~~ ↗ Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

Initialize(G, s)

$Q = \text{new PriorityQueue}(V, d)$ // Gewichtung

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{ExtractMin}()$

foreach $v \in \text{Adj}[u]$ **do**

 Relax'($u, v; w$)

$v \in Q$ and ...

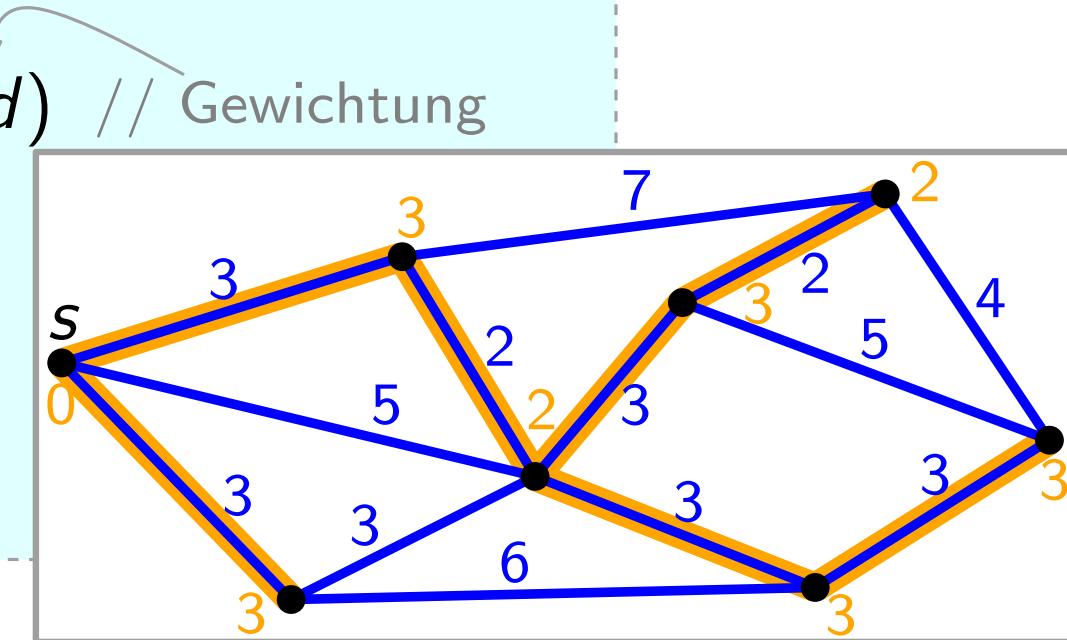
Relax'($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

$v.d = u.d + w(u, v)$

$v.\pi = u$

$Q.\text{DecreaseKey}(v, v.d)$



Korrektheit? ✓

Laufzeit?

$O(|E| \cdot \text{DecreaseKey} + |V| \cdot \text{ExtractMin})$

$\Rightarrow O((E + V) \log V)$ [Heap/RS-Baum]

$\Rightarrow O(E + V \log V)$ [Fibonacci-Heap]

Folgt aus Korollar:
 $A = \{ \{u, u.\pi\} : u \notin Q \}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^*, V \setminus Q^*)$,
wobei $Q^* = Q \cup \{u\}$.

Einschub: halbdynamische Mengen

(wachsen nur,
schrumpfen nicht)

Die halbdyn. Mengen zerlegen immer eine Grundmenge X .

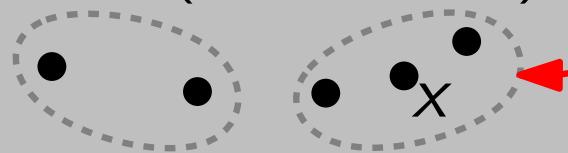
Drei Operationen:

MakeSet(Element x)



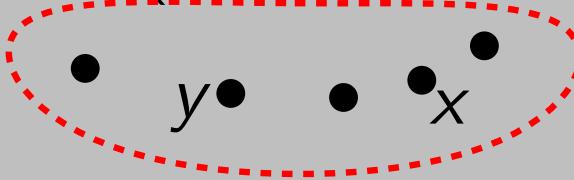
legt die Menge $\{x\}$ an.

FindSet(Element x)



liefert (Zeiger auf) die Menge zurück,
die momentan x enthält.

Union(Elem. x , Elem. y)



vereinigt die Mengen,
die momentan x und y enthalten.

Satz.

Eine Folge von m MakeSet-, Union- und FindSet-Oper., von denen n MakeSet-Oper. sind, benötigt $O(m \cdot \alpha(n))$ Zeit, wobei $\alpha(n) \leq 4$ für alle $n \leq 10^{80}$. Insbesondere $\alpha(n) \ll \log_{10} n$ für $n > 1$.

funktionales Inverses der extrem schnellwachsenden Ackermann-Funktion $A(n, n)$

Der Algorithmus von Kruskal

KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)

$A = \emptyset$

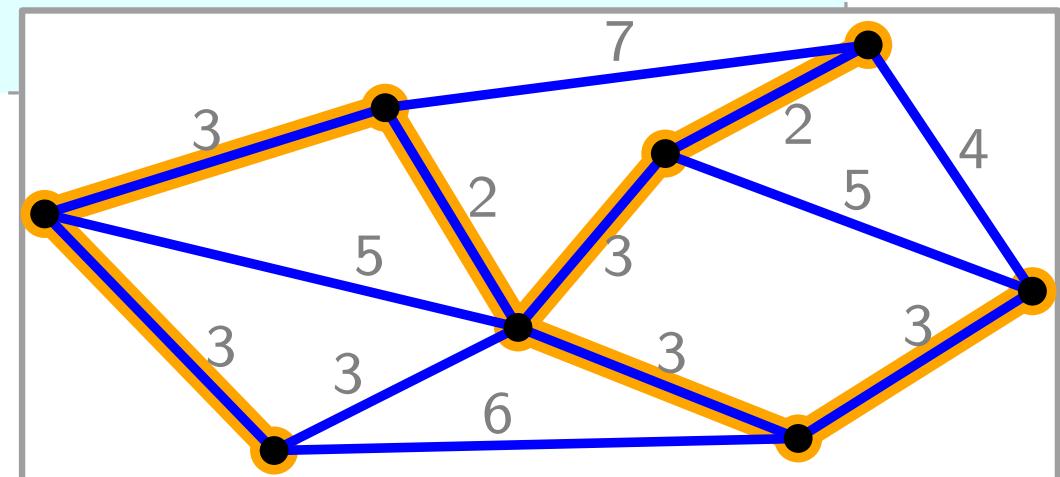
foreach $v \in V$ **do**
 └ MakeSet(v)

Sortiere E nicht-absteigend nach Gewicht w

foreach $uv \in E$ **do**
 └ **if** $\text{FindSet}(u) \neq \text{FindSet}(v)$ **then**
 └ $A = A \cup \{uv\}$
 └ Union(u, v)

Laufzeit?

$|V| \cdot \text{MakeSet} + (|V| - 1) \cdot \text{Union}$
 $+ 2|E| \cdot \text{FindSet} + \text{Sort}(E)$
 $\in O(E \log V + E \log E)$
 $= O(E \log V)$! Warum ??



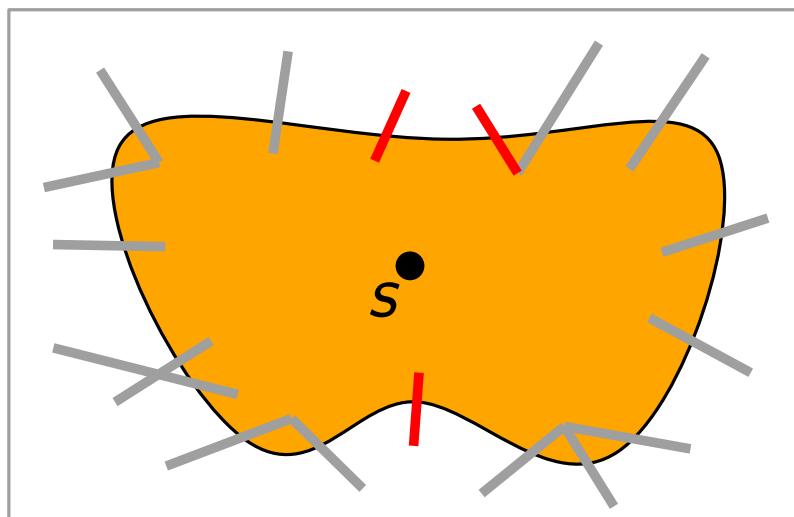
Weil $O(\log E) \subseteq O(\log V^2) = O(\log V)$. Ah...

Übersicht: Algo. für min. Spannbäume



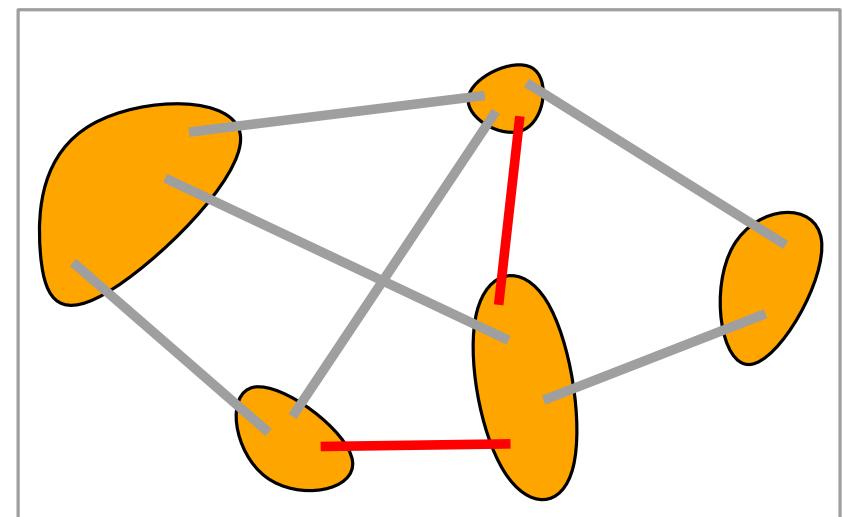
Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus
- aktuelle Kantenmenge zusammenhängend
- Laufzeit $O(E + V \log V)$



Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht
- nach Einfügen der i . Kante gibt es $n - i$ Zshgskomp.
- Laufzeit $O(E \log V)$



Algorithmen und Datenstrukturen

Wintersemester 2018/19
22. Vorlesung

Dynamisches Programmieren

Themen für den 3. Kurztest (Do, 24.01.19)

- Rot-Schwarz-Bäume (R-S-Eigenschaften, Höhe)
- Augmentieren von Datenstrukturen
- Amortisierte Analyse
- Nächstes Paar (Teile und Herrsche)
- Graphen und Breitensuche

Anmeldung

Bis **HEUTE**, Di, 22.1., 13:00 Uhr.

Entwurfstechniken

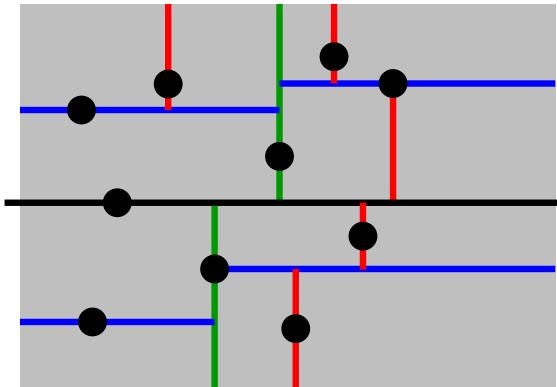
- Inkrementell
- Rekursiv
- Teile und Herrsche
- Randomisiert



- Dynamisches Programmieren

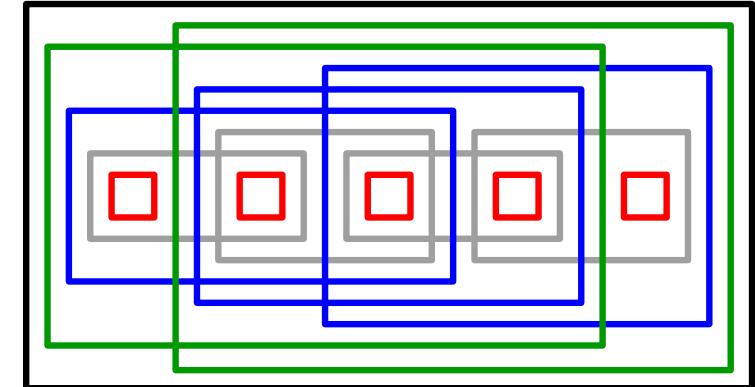
meint hier das Arbeiten mit einer Tabelle,
nicht das Schreiben eines Computerprogramms.

Vergleich



Teile und Herrsche

- zerlegt Instanz rekursiv in *disjunkte* Teilinstanzen
- top-down
- eher für Entscheidungs- oder Berechnungsprobleme



Dynamisches Programmieren

- zerlegt Instanz in *überlappende* Teilinstanzen, d.h. Teilinstanzen haben z.T. dieselben Teilteilinstanzen. Lösungen von Teilinstanzen werden zwischengespeichert, nicht neu berechnet.
- meist bottom-up
- meist für Optimierungsprobleme

Fahrplan

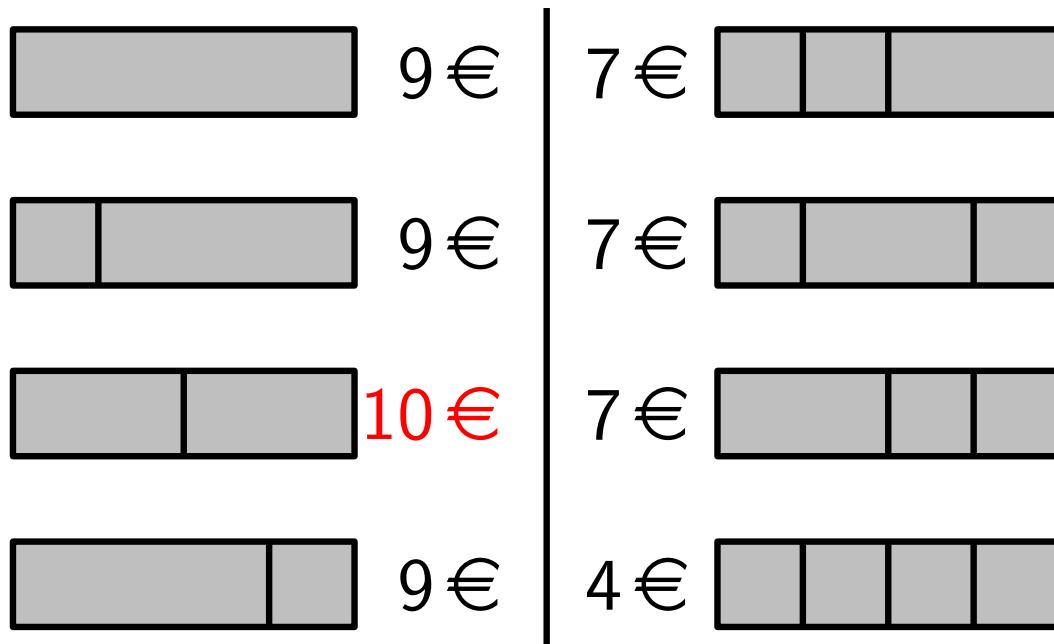
1. Struktur einer optimalen Lösung charakterisieren
2. Wert einer optimalen Lösung rekursiv definieren
3. Wert einer optimalen Lösung berechnen (meist bottom-up)
4. Optimale Lösung aus berechneter Information konstruieren

Ein Beispiel

Zerlegungsproblem

Wir haben einen Stab der Länge n und kennen die Preise p_1, p_2, \dots, p_n für Stäbe der Längen $1, 2, \dots, n$.

Durch welche Zerlegung unseres Stabs können wir unseren Ertrag maximieren?



Länge i	1	2	3	4
Preis p_i [in €]	1	5	8	9

Ein erster Versuch

Wir haben einen Stab der Länge n und kennen die Preise p_1, p_2, \dots, p_n für Stäbe der Längen $1, 2, \dots, n$.

Beispiel: $n = 4$

Länge i [in m]	1	2	3	4
Preis p_i [in €]	1	5	8	9
Quotient [€/m]	1	$2\frac{1}{2}$	$2\frac{2}{3}$	$2\frac{1}{4}$

Welche Stabzerlegung maximiert den Ertrag?

Ein ADSler schlägt vor:

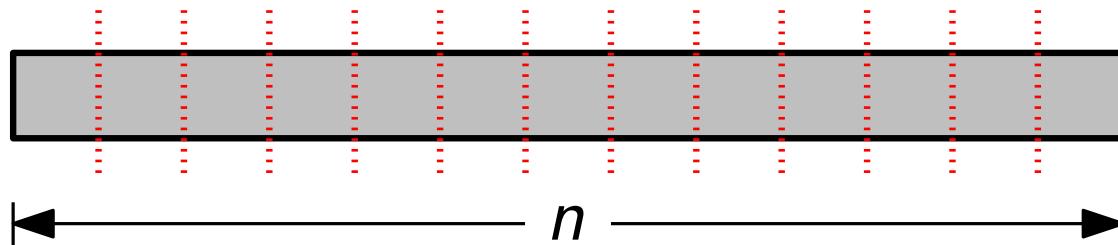
- Berechne für $i = 1, \dots, n$ den Preis pro Meter $q_i = p_i/i$.
- Zerlege Stab in möglichst viele Stücke der Länge i mit q_i max.
- Streiche alle Stablängen $\geq i$ aus der Tabelle und wiederhole den Prozess mit dem Stabrest (falls > 0).

Funktioniert dieser Greedy-Algorithmus?  Ja? Beweisen!

 Nein? Gegenbeispiel!

Rohe Gewalt

Frage: Wie viele Möglichkeiten gibt es einen Stab der Länge n zu zerlegen?



Antw.: Können $n - 1$ mal entscheiden: *schneiden* oder *nicht*.
 $\Rightarrow 2^{n-1}$ verschiedene^{*} Zerlegungen

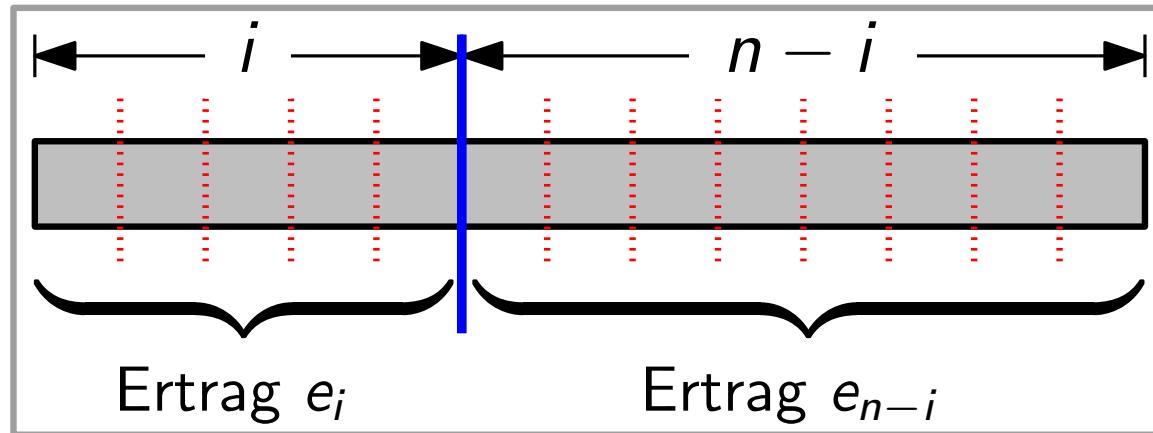
Also können wir es uns nicht leisten alle Zerlegungen durchzugehen und für jede ihren Ertrag zu berechnen.

^{*}) Genauer: die gesuchte Anzahl ist die Anzahl $p(n)$ der *Partitionen* der Zahl n , die angibt, auf wie viele Arten man n als Summe von natürlichen Zahlen schreiben kann. Es gilt $p(n) \approx e^{\pi \sqrt{2n/3}} / (4n\sqrt{3}) \in \Theta^*((13,00195...)^{\sqrt{n}})$.

1. Struktur einer optimalen Lösung charakterisieren

Def. Für $i = 1, \dots, n$

sei e_i der maximale Ertrag für einen Stab der Länge i .



Phänomen
der *optimalen*
Teilstruktur!

Beob. Ein Schnitt zerlegt das Problem in *unabh.* Teilprobleme.

2. Wert einer optimalen Lösung rekursiv definieren

Wissen nicht, welcher Schnitt in einer opt. Lösung vorkommt.

Also probieren wir einfach alle möglichen Schnitte aus:

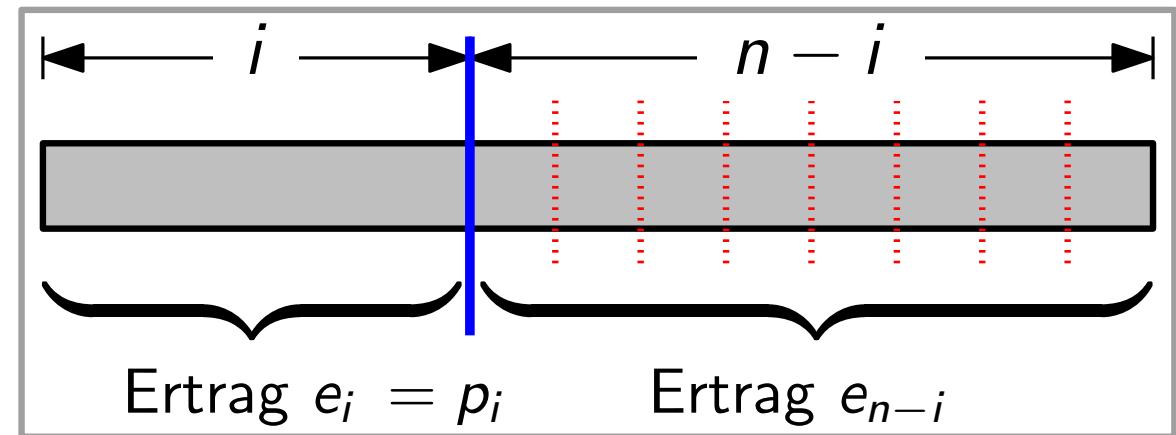
$$e_n = \max\{ p_n, e_1 + e_{n-1}, e_2 + e_{n-2}, \dots, e_{n-1} + e_1 \}$$

2. Wert einer optimalen Lösung rekursiv definieren

$$e_n = \max\{ p_n, e_1 + e_{n-1}, e_2 + e_{n-2}, \dots, e_{n-1} + e_1 \}$$

Kleine Verbesserung:

Verbiete weitere Schnitte im linken Teilstück!



Also gilt:

$$\begin{aligned} e_n &= \max\{ p_n, p_1 + e_{n-1}, p_2 + e_{n-2}, \dots, p_{n-1} + e_1 \} \\ &= \max_{1 \leq i \leq n} \{ p_i + e_{n-i} \}, \quad \text{wobei } e_0 := 0. \end{aligned}$$

Vorteil: Wert einer opt. Lösung ist Summe aus einer Zahl der Eingabe und *einem* Wert einer opt. Teillösung.

3. Wert einer optimalen Lösung berechnen: *top-down*

Wir wissen: $e_n = \max_{1 \leq i \leq n} \{p_i + e_{n-i}\}$, wobei $e_0 := 0$.

StangenZerlegung(int[] p , int $n = p.length$)

```

if  $n == 0$  then return 0
 $q = -\infty$ 
for  $i = 1$  to  $n$  do
     $q = \max\{q, p[i] + \text{StangenZerlegung}(p, n - i)\}$ 
return  $q$ 

```

Laufzeit: Sei $A(n)$ die Gesamtzahl
von Aufrufen von $\text{StangenZerlegung}(p, \cdot)$
beim Ausführen von $\text{StangenZerlegung}(p, n)$

$$\Rightarrow A(0) = 1$$

und $A(n) = 1 + \sum_{i=1}^n A(n-i) = 1 + \sum_{j=0}^{n-1} A(j) \stackrel{\text{Beweis?}}{=} 2^n$



3. Wert einer optimalen Lösung berechnen: mit Tabelle

Zeit-Speicher-Tausch (engl. *time-memory trade-off*)

```
MemoStangenZerlegung(int[] p, int n = p.length)
```

```
    e = new int[0..n]
```

```
    e[0] = 0
```

```
    for i = 1 to n do
```

```
        e[i] = -∞
```

```
    return HauptStangenZerlegung(p, n, e)
```

```
HauptStangenZerlegung(int[] p, int n, int[] e)
```

```
if e[n] ≥ 0 or n == 0 then return e[n]
```

```
q = -∞
```

```
for i = 1 to n do
```

```
    q = max{q, p[i] + HauptStangenZerlegung(p, n - i, e)}
```

```
e[n] = q; return q
```

3. Wert einer optimalen Lösung berechnen: bottom-up

BottomUpStangenZerlegung(int[] p, int n)

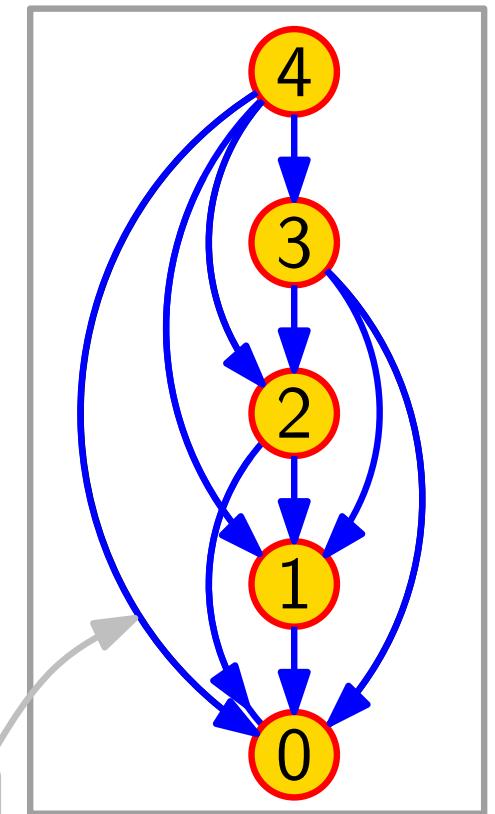
```

e = new int[0..n]
e[0] = 0
for j = 1 to n do
    q = -∞
    for i = 1 to j do
        q = max{q, p[i] + e[j - i]}
    e[j] = q
return q

```

Neu: *kein* rekursiver Aufruf!

Kante (j, i) bedeutet:
Teilinstanz j benutzt Wert einer opt. Lösung von Teilinstanz i .



Graph der Teilinstanzen

Beob. Größe des Graphen (d.h. in erster Linie Anz. Kanten) ist proportional zur Laufzeit des DP (Anz. Additionen).

Satz. BottUpSzerl() und MemoSzerl() laufen in $O(n^2)$ Zeit.

4. Optimale Lösung aus berechneter Info. konstruieren

GibZerlegungAus(int[] p , int n)

$\ell = \text{new int}[0..n]$; $e = \text{new int}[0..n]$

ErweiterteBottomUpZerlegung(p, e, ℓ, n)

while $n > 0$ **do** // gib wiederholt Länge des 1. Teilstücks aus
 ┌ print $\ell[n]$; $n = n - \ell[n]$

ErweiterteBottomUpZerlegung(int[] p , int[] e , int[] ℓ , int n)

$e[0] = 0$

for $j = 1$ **to** n **do**

$q = -\infty$

for $i = 1$ **to** j **do**

if $q < p[i] + e[j - i]$ **then**

$q = p[i] + e[j - i]$

$\ell[j] = i$

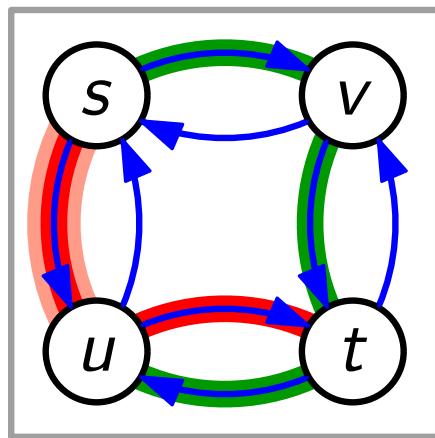
// merke Länge des 1. Teilstücks

$e[j] = q$

Längste Wege

Gegeben: ungewichteter gerichteter Graph $G = (V, E)$ mit $s, t \in V$, $s \neq t$ und t von s erreichbar.

Gesucht: ein längster einfacher $s-t$ -Weg,
d.h. eine Folge $\langle s = v_0, v_1, \dots, v_k = t \rangle$ mit
 $v_0 v_1, \dots, v_{k-1} v_k \in E$, $v_i \neq v_j$ (für $i \neq j$) und k maximal.



$\langle s, u, t \rangle$ ist ein längster einfacher $s-t$ -Weg.

Aber:

$\langle s, u \rangle$ ist *kein* längster einfacher $s-u$ -Weg;

$\langle s, v, t, u \rangle$ ist ein längster einfacher $s-u$ -Weg!

Fahrplan

1. Struktur einer optimalen Lösung charakterisieren
2. Wert einer optimalen Lösung rekursiv definieren
3. Wert einer optimalen Lösung berechnen (meist bottom-up)



*) Es ist NP-schwer für (G, s, t, k) zu entscheiden, ob G einen einfachen $s-t$ -Weg der Länge k enthält. (Vgl. Hamilton-Weg!)

Längste Wege in azyklischen Graphen

Gegeben: gewichteter gerichteter *kreisfreier* Graph $G = (V, E; w)$
mit $s, t \in V$, $s \neq t$ und t von s erreichbar.

Gesucht: ein längster $s-t$ -Weg.

Beob₁ In kreisfreien Graphen sind alle Wege einfach.

Beob₂ Dieses Problem hat optimale Teilstruktur, denn:
Ein längster $s-t$ -Weg π gehe durch u , d.h.

$$\pi = s \xrightarrow{\pi_{su}} u \xrightarrow{\pi_{ut}} t.$$

Dann gilt:

π_{su} ist längster $s-u$ -Weg; π_{ut} ist längster $u-t$ -Weg –
sonst wäre π kein längster $s-t$ -Weg.

Außerdem gilt $V(\pi_{su}) \cap V(\pi_{ut}) = \{u\}$;
sonst gäbe es einen Kreis!

Algorithmus nach Fahrplan

1. Struktur einer optimalen Lösung charakterisieren 

2. Wert einer optimalen Lösung rekursiv definieren

$$d_v = \max_{uv \in E} d_u + w(u, v) \quad // \text{Länge eines längsten } s-v\text{-Wegs}$$

so!

3. Wert einer optimalen Lösung berechnen (hier bottom-up)

- G topologisch sortieren
- d -Werte initialisieren: $d_s = 0$ und $d_v = -\infty$ für alle $v \neq s$
- for-Schleife durch Knoten v.l.n.r. d -Werte berechnen

Übrigens: kürzeste Wege in kreisfreien Graphen gehen analog
(mit min statt max und $+\infty$ statt $-\infty$)

Und jetzt?

Im Buch [CLRS] werden weitere, praxisrelevante Probleme mit dynamischem Programmieren gelöst:

- Ketten von Matrixmultiplikationen
- Längste gemeinsame Teilfolge (in Zeichenketten)
- Optimale binäre Suchbäume

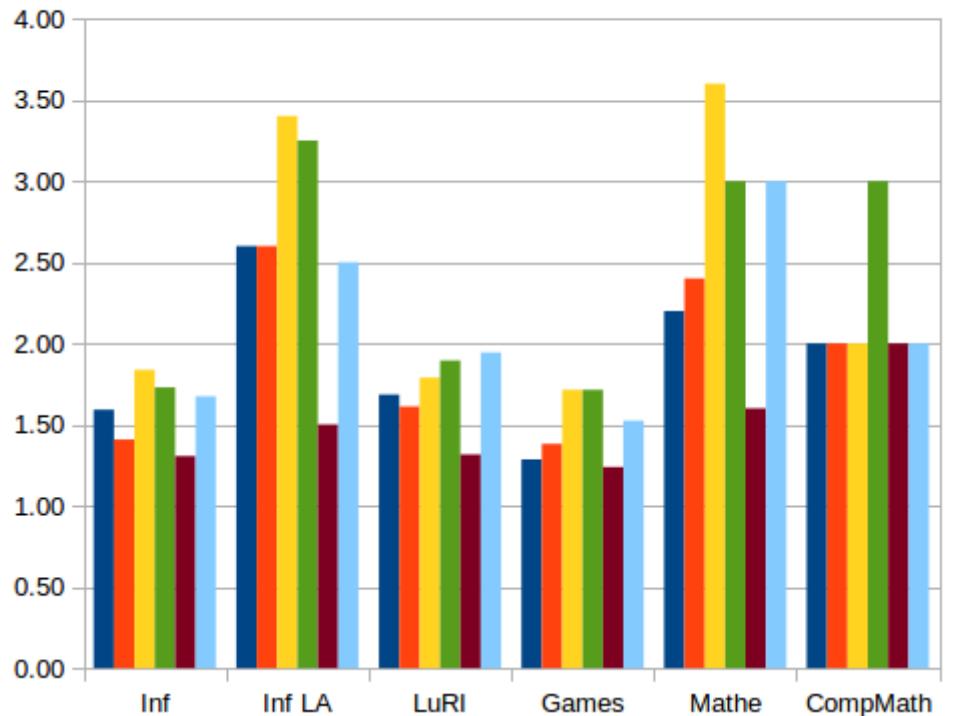
Lesen Sie Kapitel 15.2–5 !!!

Algorithmen und Datenstrukturen

Wintersemester 2018/19
23. Vorlesung

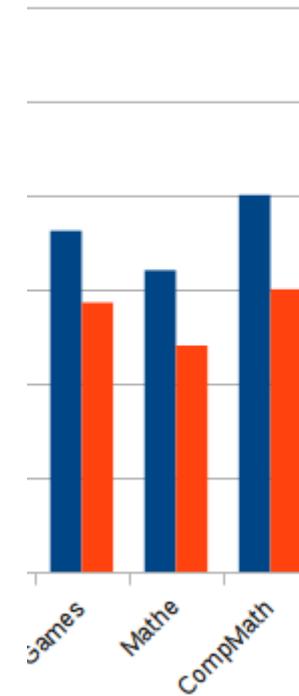
Greedy- und Approximationsalgorithmen

Bewertung der Übungsleiter

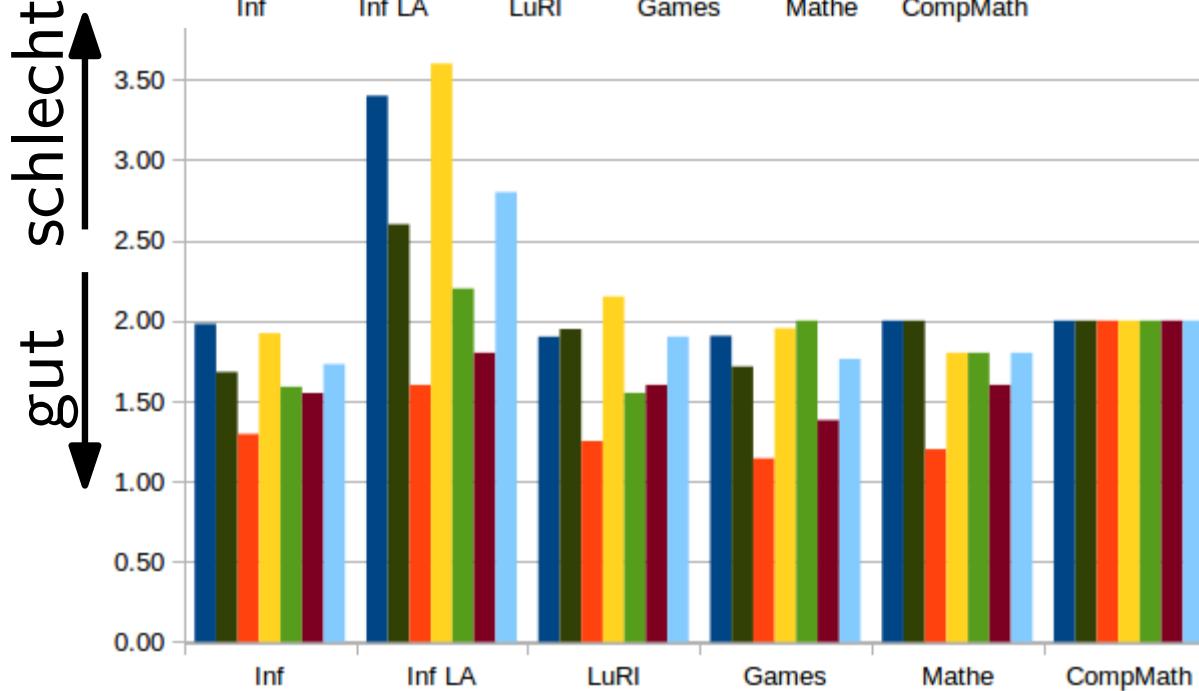


- Übungsleiter sind um gutes Verhältnis bemüht
- ausreichend vorbereitet
- Stoff verständlich
- roter Faden
- geht auf Fragen ein
- Antworten helfen

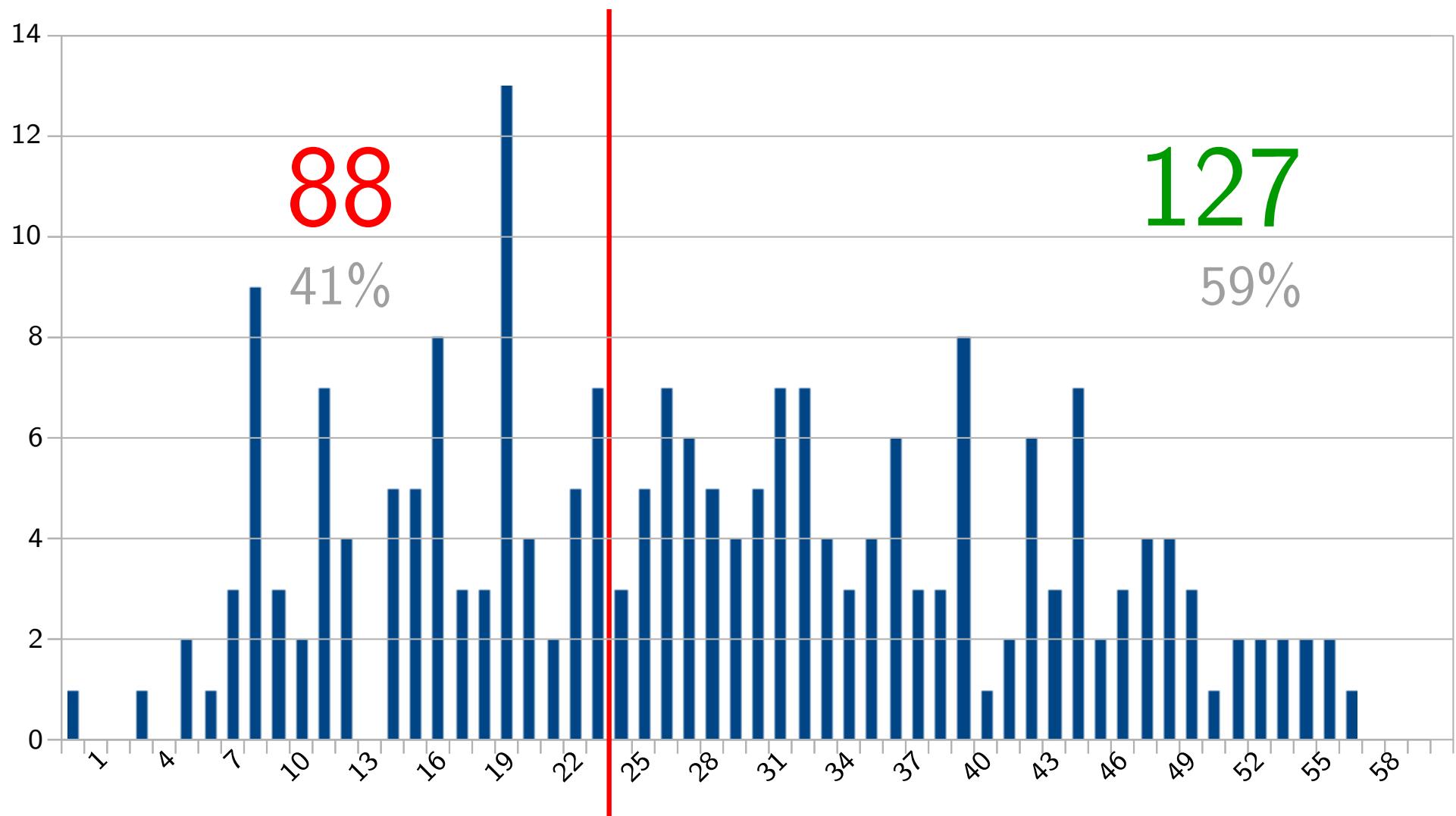
Tempo



■ Vorlesungstempo
■ Übungstempo

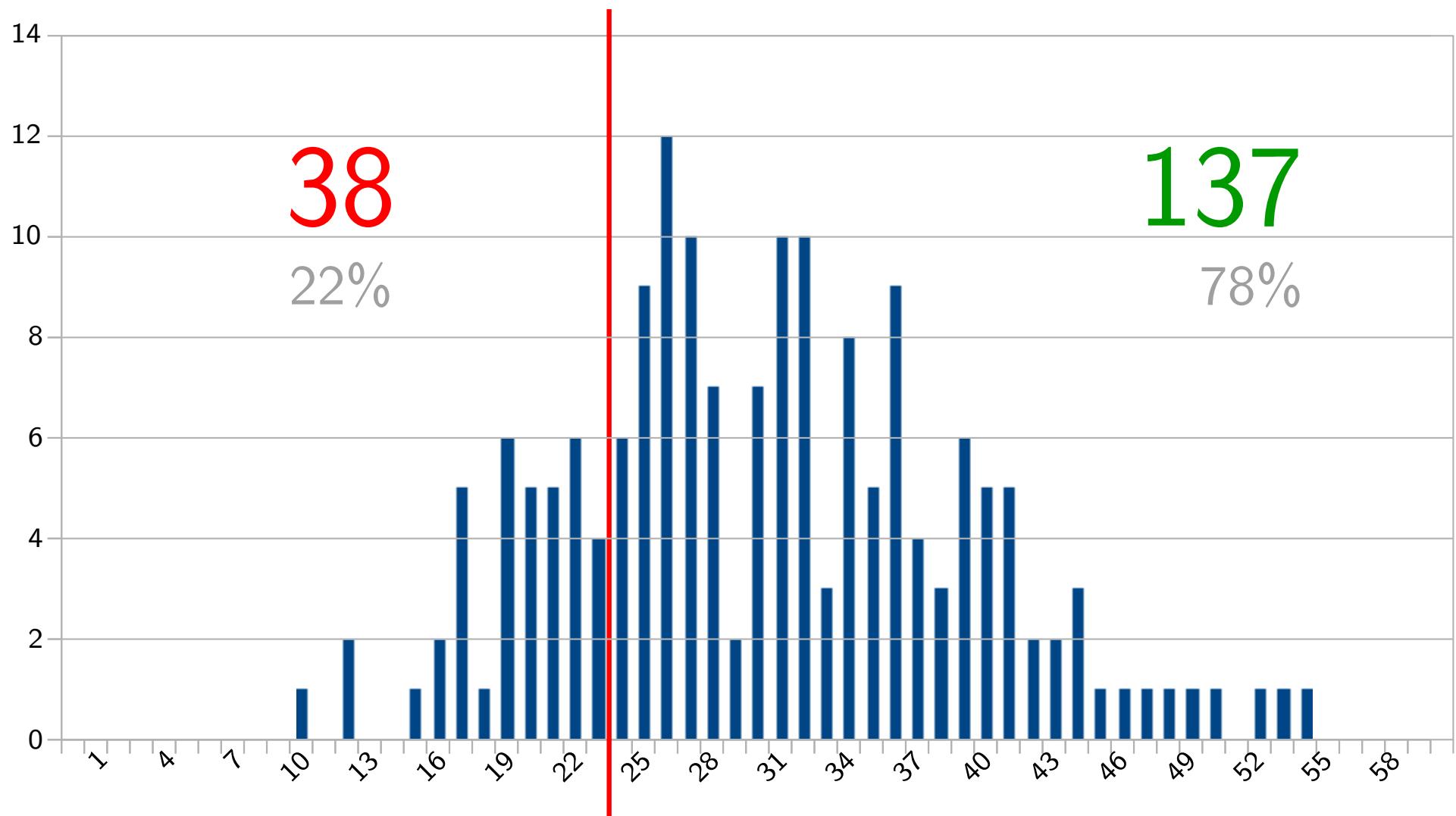


Ergebnisse des 1. Kurztests



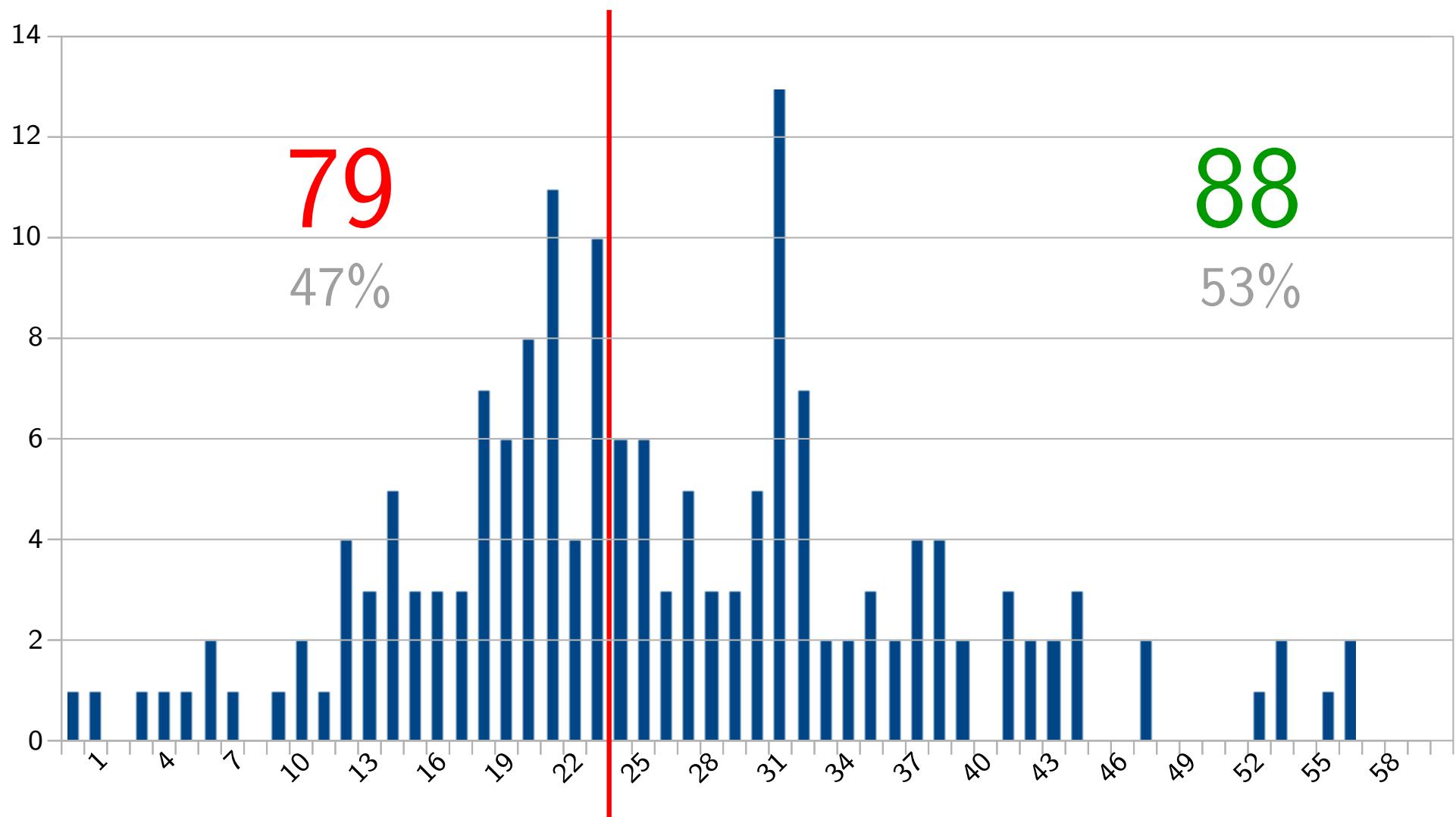
$n = 215$; Durchschnitt 28,3; Median 27

Ergebnisse des 2. Kurztests



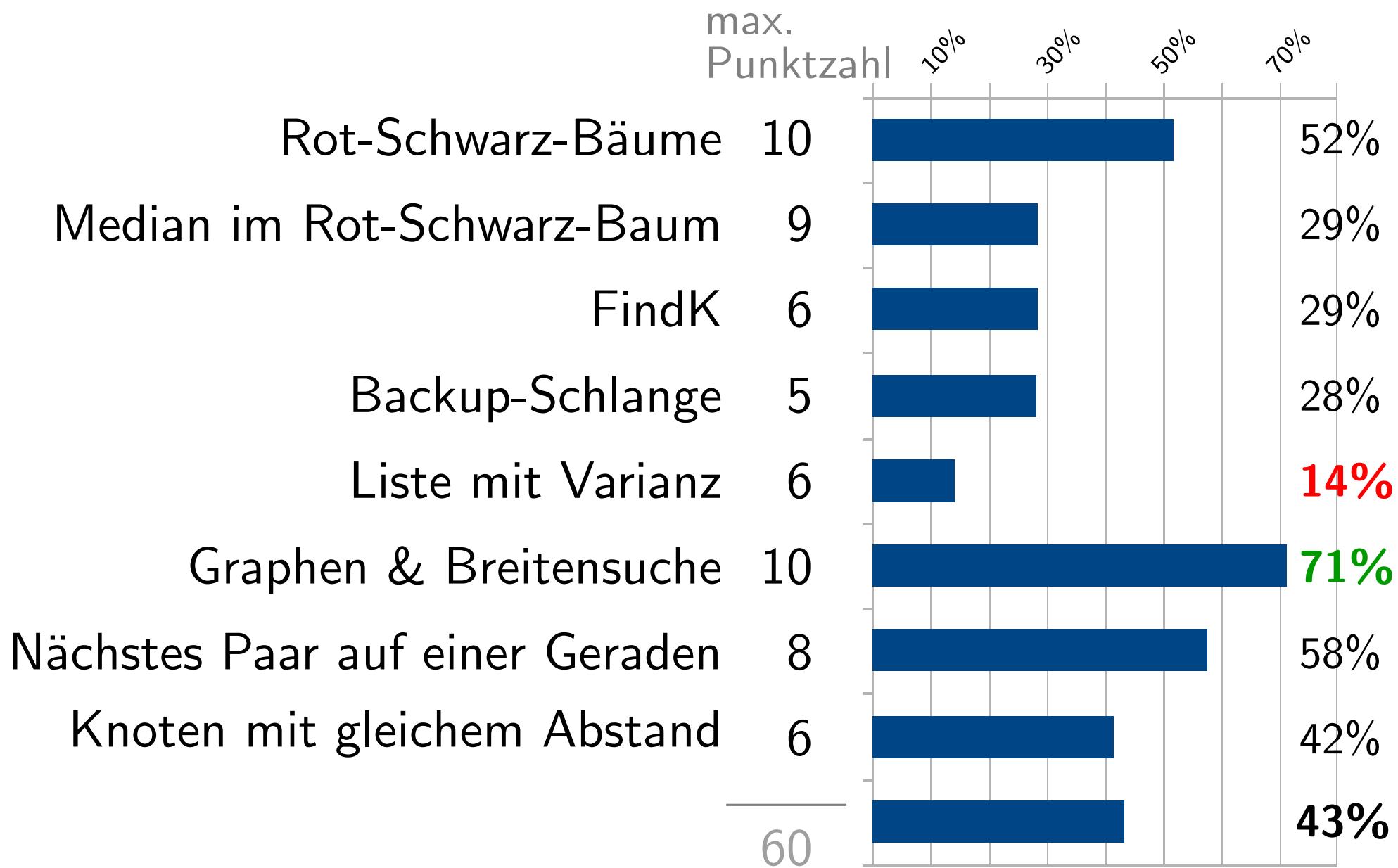
$n = 175$; Durchschnitt 30,5; Median 27

Ergebnisse des 3. Kurztests



$n = 167$; Durchschnitt 26,0; Median 24

Kurztest 3: Abschneiden nach Aufgaben



Vorbereitung auf die Klausur am Di, 12.2.

Tipp: Schreiben Sie sich alle Fragen auf, die Ihnen bei der Vorbereitung in den Kopf kommen!

- Nutzen Sie die Übungen zum Fragen.
- Kommen Sie nach der Vorlesung zu mir.
- Nutzen Sie meine Sprechstunde (Mi, 13–14 Uhr)

Auch in der letzten Vorlesung (Do, 7.2.) können Sie Fragen stellen. Ich bringe alle Folien mit.

Die vorletzte Vorlesung (Di, 5.2.) fällt aus, da beide großen HS für die Klausur GADS für Wilnfs & MCS benötigt werden.

In WueStudy bis Di, 31.1., 23:59 Uhr zur Klausur **anmelden!**
Wer sich nicht anmeldet, kann die Klausur NICHT mitschreiben!

ADS-Repetitorium

- Mo., 1.4., bis Fr, 5.4. (5 Tage, SR 10)
- Bereitet auf den zweiten Klausurtermin (Di, 9.4.) vor
- Täglich 9:30–16:00 Uhr: Abwechselnd Vorlesung / Übung
- Leitung: Michael Kreuzer und Diana Sieper
- 2008 haben 15 HörerInnen an der 2. Klausur teilgenommen.

	ADS-Repetitorium	<i>bestanden</i>
regelmäßig teilgenommen	7	4
nicht oder unregelmäßig	8	1

Operations Research

Optimierung für Wirtschaftsabläufe:

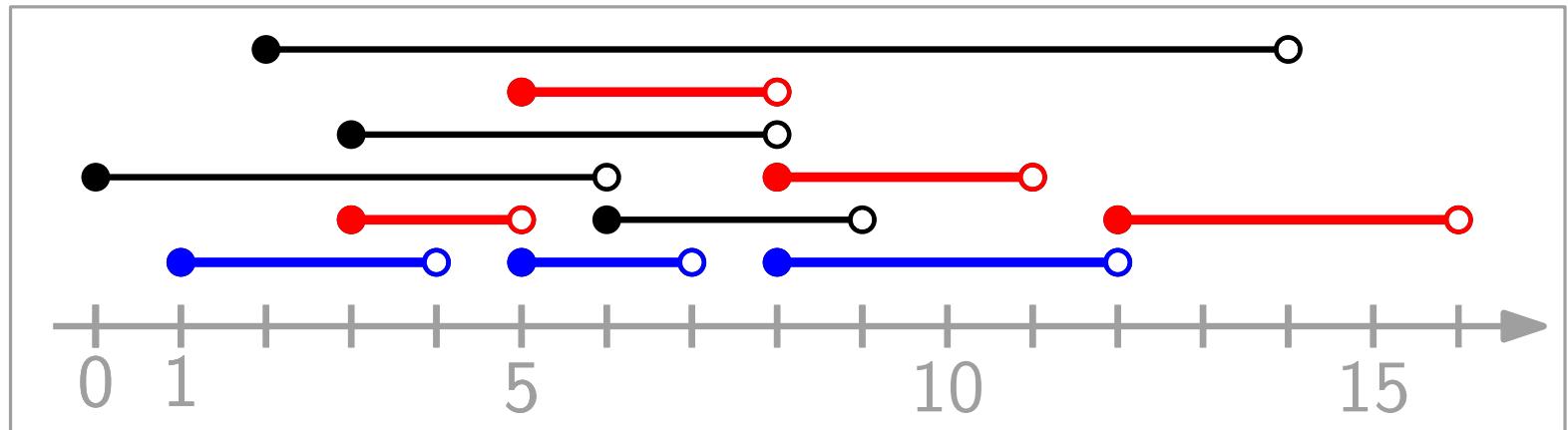
- Standortplanung
- Ablaufplanung
- Flottenmanagement
- Pack- und Zuschnittprobleme
- ...

Werkzeuge:

Statistik, Optimierung, Wahrscheinlichkeitstheorie, Spieltheorie,
Graphentheorie, mathematische Programmierung, Simulation...

Ein einfaches Problem der Ablaufplanung

Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von *Aktivitäten*, wobei für $i = 1, \dots, n$ gilt $a_i = [s_i, e_i]$.



a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

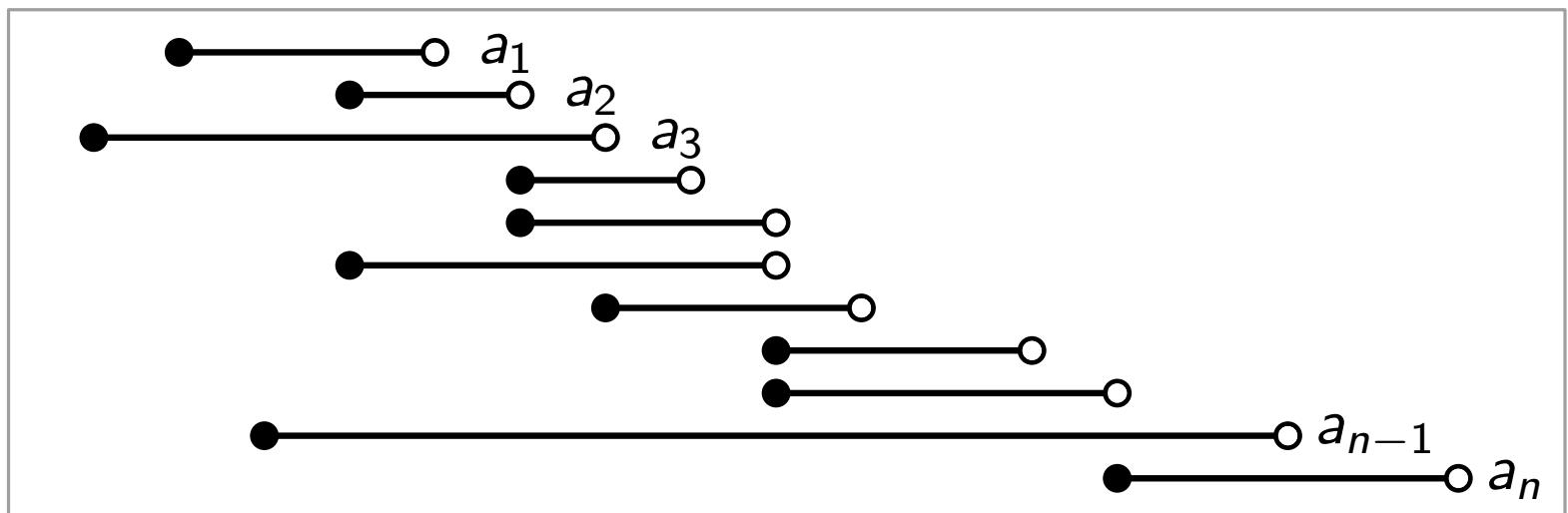
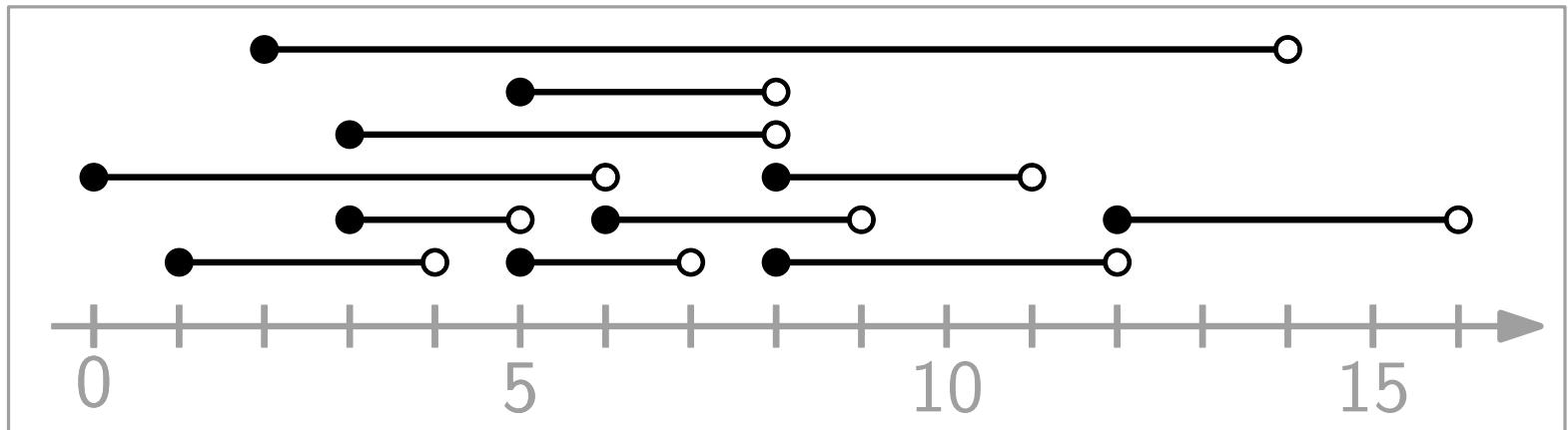
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

Grund: Aktivitäten (à 1€), die gleiche Ressource benutzen

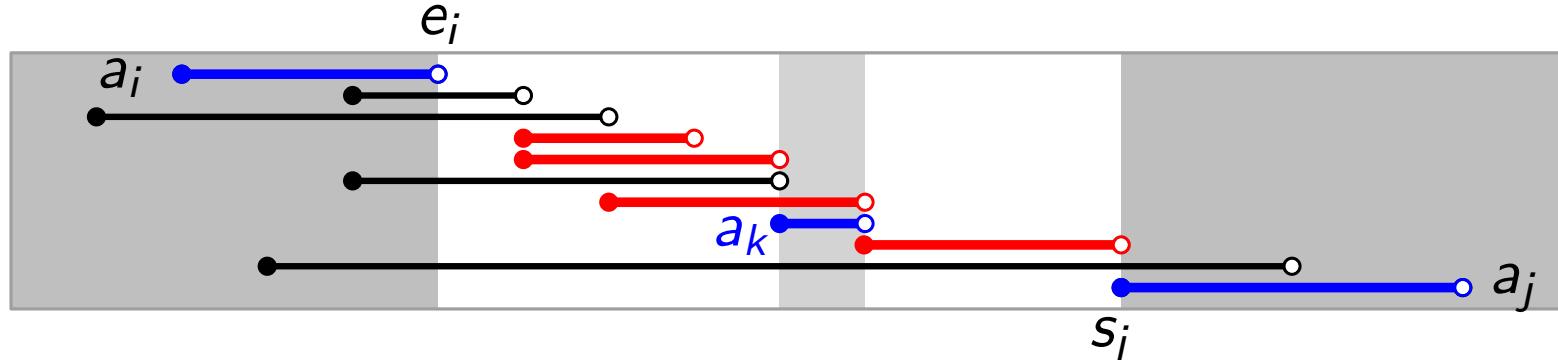
Ein kleiner technischer Trick

Wir nummerieren (für den Rest der Vorlesung) die Aktivitäten so, dass für die Endtermine gilt $e_1 \leq e_2 \leq \dots \leq e_n$.



Dynamisches Programmieren?

1. Struktur einer optimalen Lösung charakterisieren



Sei $A_{ij} \subset A$ die Menge aller Aktivitäten in $[e_i, s_j]$, d.h. „zwischen“ a_i und a_j .

Ang. a_i und a_j sind in einer opt. Lösung $L \subseteq A$ enthalten, dann ist $L \cap A_{ij}$ eine opt. Lösung für A_{ij} .

⇒ optimale Substruktur!

Beweis? Austauschargument!

2. Wert einer optimalen Lösung rekursiv definieren

Sei c_{ij} die Kardinalität einer opt. Lösung für A_{ij} .

Dann gilt: $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$

Dynamisches Programmieren?

3. Wert einer optimalen Lösung berechnen

Setze $e_0 = -\infty$ und $s_{n+1} = +\infty$. Dann ist $A = A_{0,n+1}$.

Berechne $c_{0,n+1}$, die Kardinalität einer opt. Lösung für A .

(a) *top-down*

`TopDownDP(int[]s, int[]e, int i, int j)`

→ liefert c_{ij}

(b) *bottom-up*

`BottomUpDP(int[]s, int[]e)`

→ liefert $c_{0,n+1}$

Siehe Folie

„Zurück zum dynamischen Programmieren“

Laufzeit? $O(n^3)$...

2. Wert einer optimalen Lösung rekursiv definieren

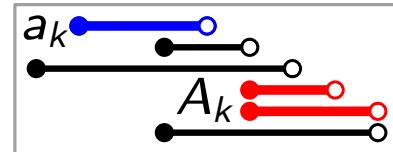
Sei c_{ij} die Kardinalität einer opt. Lösung für A_{ij} .

Dann gilt: $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$

Darf's auch etwas einfacher sein?

Idee: Sei L opt. Lösung für A . – Welche Aktivität hat gute Chancen die erste („linkste“) in L zu sein?

Intuition: Die Aktivität a_1 mit frühester Endzeit – weil a_1 die gemeinsame Ressource am wenigsten einschränkt.



Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

Sei L_k eine optimale Lösung von A_k .

Falls Intuition korrekt, dann ist $\{a_1\} \cup L_1$ optimal.

Satz.

optimale
Teilstruktur!

Sei $A_k \neq \emptyset$.

Sei a_m Aktivität mit frühester Endzeit in A_k .

\Rightarrow es gibt eine opt. Lösung von A_k , die a_m enthält.

Beweis.

Austauschargument!

Greedy – rekursiv

```
GreedyRecursive(int[]s, int[]e)
```

$e_0 = -\infty \quad // \Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain(s, e, 0)

```
GreedyRecursiveMain(int[]s, int[]e, int k) // best. Lsg. für  $A_k$ 
```

$m = k + 1; \quad n = s.length$

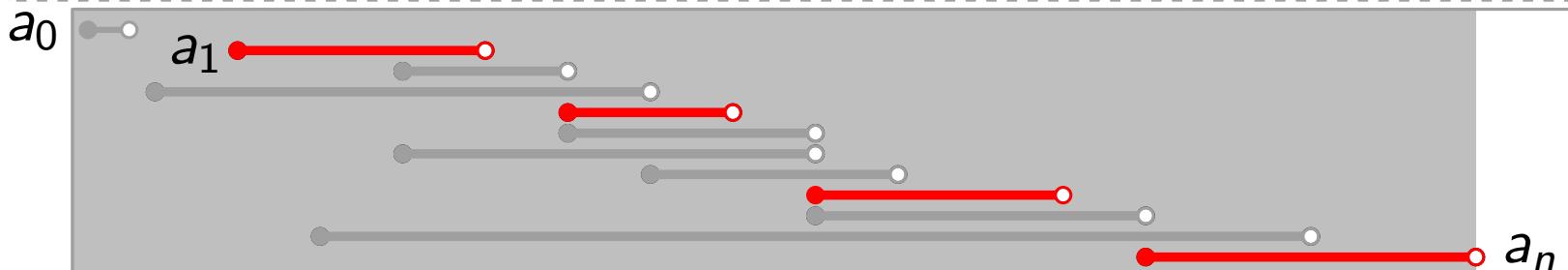
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ and $s[m] < e[k]$ **do**

 └ $m = m + 1$

if $m > n$ **then return** \emptyset

else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

```
GreedyRecursive(int[]s, int[]e)
```

$e_0 = -\infty \quad // \Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain(s, e, 0)

```
GreedyRecursiveMain(int[]s, int[]e, int k) // best. Lsg. für  $A_k$ 
```

$m = k + 1; \quad n = s.length$

// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ and $s[m] < e[k]$ **do**

 └ $m = m + 1$

if $m > n$ **then return** \emptyset

else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

Laufzeit? Wie oft wird m inkrementiert?

Insgesamt, über alle rekursiven Aufrufe, n Mal.

D.h. GreedyRecursive läuft (ohne Sortieren) in $\Theta(n)$ Zeit.

Greedy – iterativ

```
GreedyIterative(int[]s, int[]e)
```

$n = s.length$

if $n = 0$ **then return** \emptyset

$L = \{a_1\}$

$k = 1$ // höchster Index in L

for $m = 2$ **to** n **do**

if $s[m] \geq e[k]$ **then**

$L = L \cup \{a_m\}$

$k = m$

return L

Laufzeit? GreedyIterative läuft ebenfalls in $\Theta(n)$ Zeit.

Bemerkung: GreedyIterative berechnet dieselbe optimale Lösung wie GreedyRecursive – die „linkteste“.

Die Greedy-Strategie

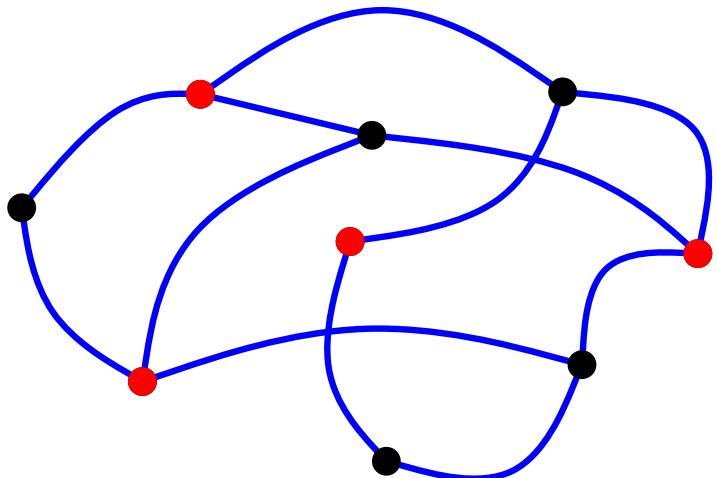
1. Teste, ob das Problem optimale Teilstruktur aufweist.
2. Entwickle eine rekursive Lösung
3. Zeige, dass bei einer Greedy-Entscheidung nur *ein* Teilproblem bleibt
4. Beweise, dass die Greedy-Wahl „sicher“ ist (vgl. Kruskal!)
5. Entwickle einen rekursiven Greedy-Algorithmus
6. Konvertiere den rekursiven in einen iterativen Algorithmus

Food for Thought

1. Welches allgemeinere Ablaufproblem könnte das DP lösen – aber nicht der GA?

Zur Erinnerung: Das DP berechnet $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$.

2. Problem *größte unabhängige Menge (guM)* in Graphen:



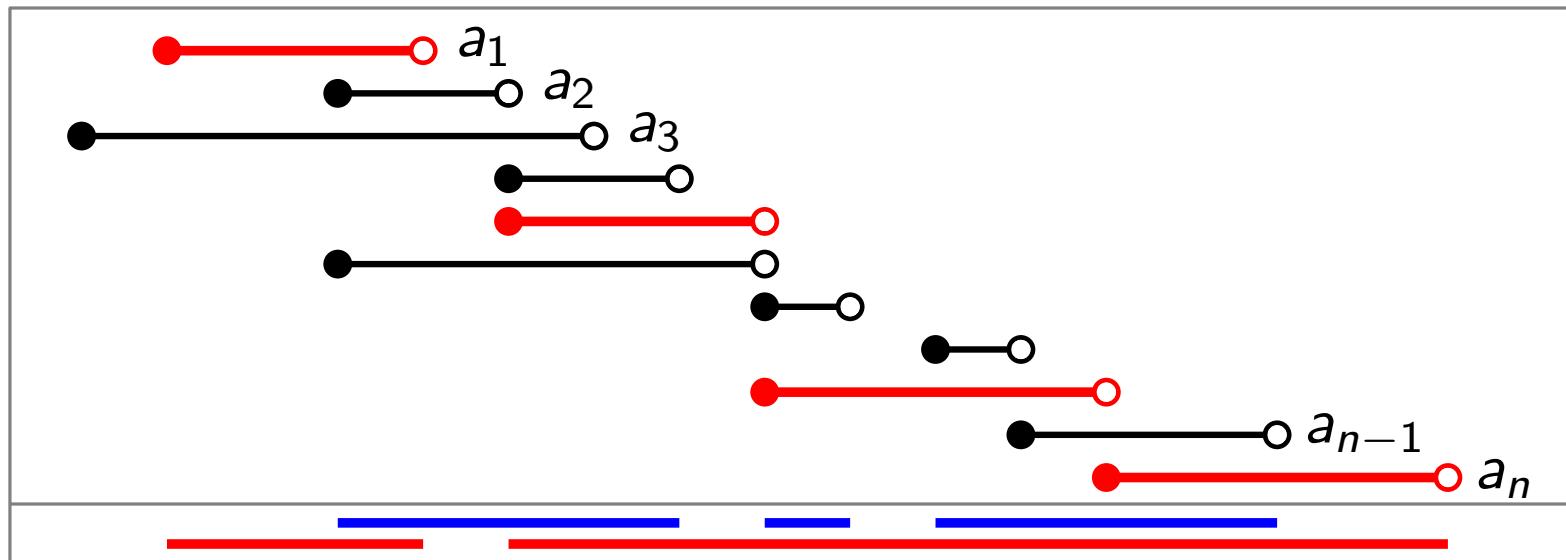
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

- Was hat guM mit unserem Ablaufplanungsproblem zu tun?
- Kann man guM mit DP oder GA lösen?

Ein ähnliches Problem der Ablaufplanung

Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von halboffenen Intervallen, mit $a_i = [s_i, e_i)$ für $i = 1, \dots, n$.

Für die Endpunkte gelte $e_1 \leq e_2 \leq \dots \leq e_n$.



Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

Grund: Intervalle $\hat{=}$ Prozesse, die die gleiche Ressource nutzen; der Gesamtertrag ist proportional zur Auslastung.

Greedy?

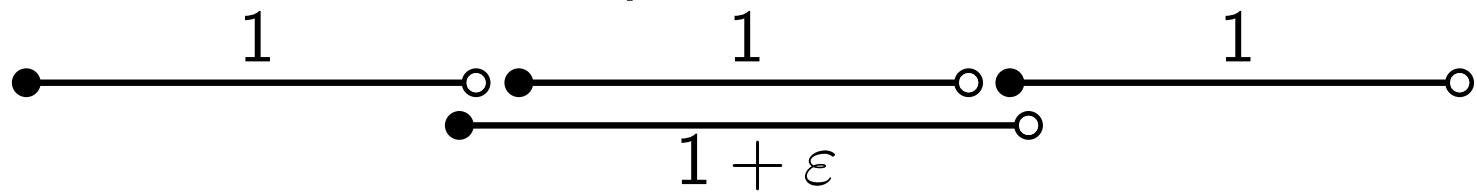
1. Versuch: Nimm Aktivität mit frühestem Endtermin,
streiche dazu inkompatible Aktivitäten und iteriere.

Gegenbsp.:



2. Versuch: Nimm längste Aktivität,
streiche dazu inkompatible Aktivitäten und iteriere.

Gegenbsp.:



Aufgabe: Können Sie den 2. GA in $O(n \log n)$ Zeit implementieren?

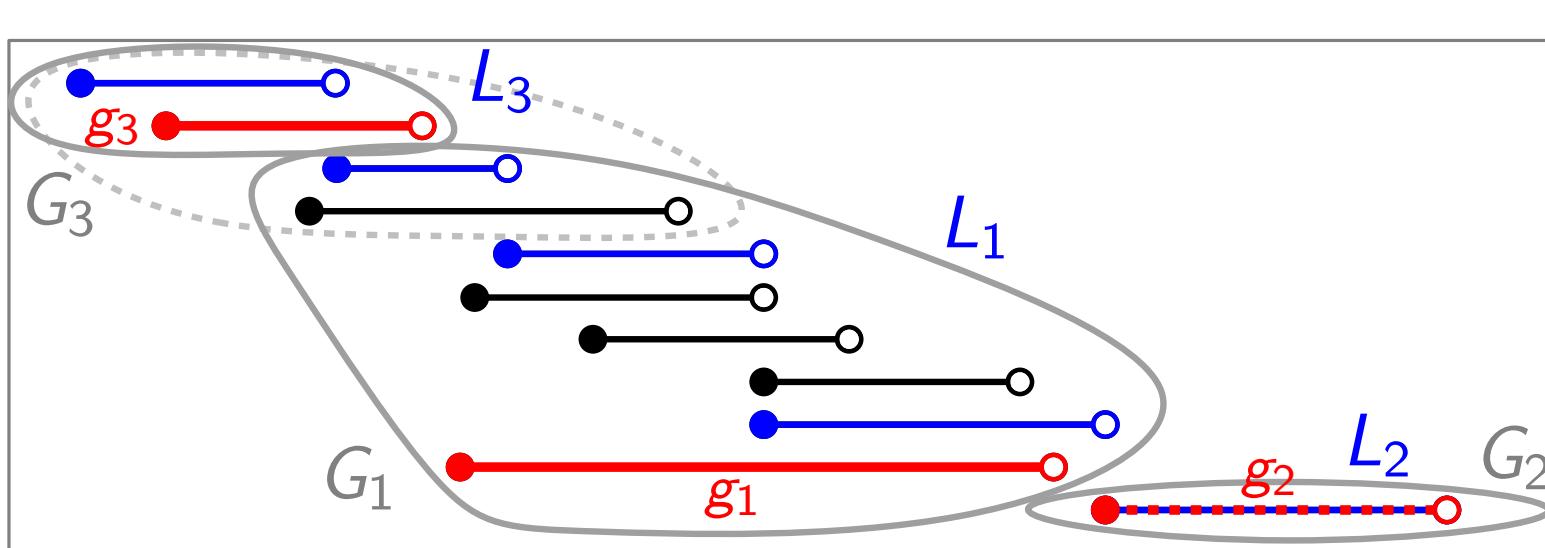
Diskutieren Sie mit Ihrer NachbarIn!

Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

Sei $G = \{g_1, g_2, \dots, g_k\} \subseteq A$ die Greedy-Lösung (*in dieser Rf.*).

Für $i = 1, \dots, k$ sei $G_i = \{a \in A \mid a \cap g_i \neq \emptyset\} \setminus (G_1 \cup \dots \cup G_{i-1})$



und sei
 $L_i = L \cap G_i$.

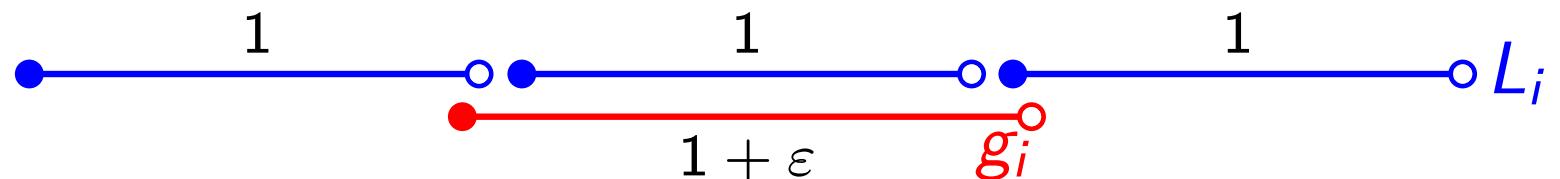
Dann gilt $A = G_1 \cup G_2 \cup \dots \cup G_k$ und $L = L_1 \cup L_2 \cup \dots \cup L_k$.

„ \subseteq “: GA wählt so lange Intervalle aus, bis es keine mehr gibt.

„ \supseteq “: klar, da $G_1 \subseteq A$, $G_2 \subseteq A$, ..., $G_k \subseteq A$

Wie gut/schlecht ist der 2. GA?

Behauptung: Für $i = 1, \dots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



Beweis.

- (a) g_i ist nach Wahl ein längstes Intervall in G ;
- (b) jedes $a \in L_i$ schneidet g_i ;
- (c) Intervalle in L_i sind paarweise disjunkt

$$\Rightarrow \text{OPT} = \ell(L) = \sum_{i=1}^k \ell(L_i) < 3 \sum_{i=1}^k \ell(g_i) = 3\ell(G)$$

$$\Rightarrow \ell(G) > \text{OPT}/3$$

\Rightarrow 2. GA liefert *immer* mind. $1/3$ der maximalen Gesamtlänge.

Also ist der 2. GA ein **Faktor-(1/3)-Approximationsalgorithmus**.

Approxi... hä?

„All exact science is dominated by the idea of approximation.“

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

z.B. Ablaufplanung

Sei ζ die *Zielfunktion* von Π : Lösung $\mapsto \mathbb{Q}_{\geq 0}$.

$$\zeta = \ell$$

Sei γ eine Zahl ≤ 1 .

$$\gamma = 1/3$$

Ein Algorithmus \mathcal{A} heißt γ -*Approximation*, wenn

- \mathcal{A} für jede Instanz I von Π eine Lösung $\mathcal{A}(I)$ berechnet, so dass

$$\frac{\zeta(\mathcal{A}(I))}{\text{OPT}(I)} \geq \gamma$$

$\zeta(\text{optimale Lösung})$

$\text{OPT}(I)$

Größe der Instanz I

- die Laufzeit von \mathcal{A} polynomiell in $|I|$ ist.

$$O(n \log n)$$

1/3-Approx.
liefert Menge von
Aktivitäten, deren
Gesamtlänge
mindestens 1/3
der maximal mög-
lichen Länge ist.

Zurück zum dynamischen Programmieren

BottomUpDPWeighted(int[]s, int[]e)

$n = s.length$

$c = \text{new int}[0..n][1..n + 1]$ // c_{ij} = Wert einer opt. Lsg. für A_{ij}

for $d = 1$ **to** $n - 1$ **do** // d = „Distanz“ zwischen j und i

for $i = 0$ **to** $(n + 1) - d$ **do**

$j = i + d$

if a_i und a_j kompatibel **then**

$c = 0$

for $k = i + 1$ **to** $j - 1$ **do**

$c' = c[i][k] + \ell(a_k) + c[k][j]$

if $c' > c$ **then** $c = c'$

$c[i][j] = c$

else $c[i][j] = 0$

return $c[0, n + 1]$

NEU! Im ungewichteten Fall stand hier eine Eins.

// falls $a_i \cap a_j = \emptyset$

Laufzeit? $O(n^3)$

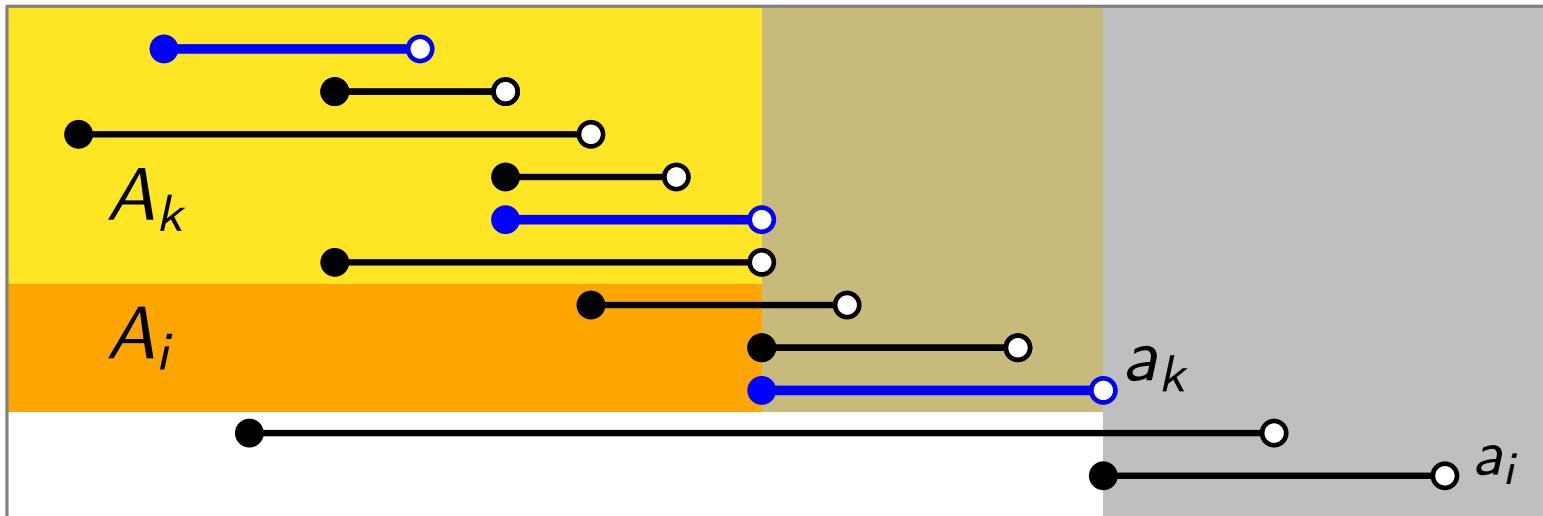
Aber warum verzweigen wir hier zweimal?

2. Wert einer optimalen Lösung rekursiv definieren

$$c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$$

Dynamisches Programmieren, aber einfacher

Für $i = 1, \dots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A , die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:

- einem letzten Intervall a_k und
- einer optimalen Lösung für A_k .

} optimale Teilstruktur!

Also gilt für den Wert c_i einer optimalen Lösung für A_i :

$$c_i = \max_{a_k \in A_i} c_k + l(a_k)$$

Dynamisches Programmieren, aber einfacher

Also gilt für den Wert c_i einer optimalen Lösung für A_i :

$$c_i = \max_{a_k \in A_i} c_k + \ell(a_k)$$

Erinnern wir uns...

c_{n+1} ist der Wert der optimalen Lösung für $A_{n+1} = A$.

Also genügt es c_1, \dots, c_{n+1} zu berechnen.

Laufzeit? $O(n^2)$

Work out the details!

Resultate:

- Der 2. Greedy-Alg. findet in $O(n \log n)$ Zeit eine Lösung, die *mindestens 1/3 des maximalen Ertrags* garantiert.
- Unser neues DP findet in $O(n^2)$ Zeit eine Lösung mit *maximalem Ertrag*. *Trade-Off zwischen Zeit und Qualität!*

Algorithmen und Datenstrukturen

Wintersemester 2018/19
24. Vorlesung

Der Handlungsreisende

Das Problem

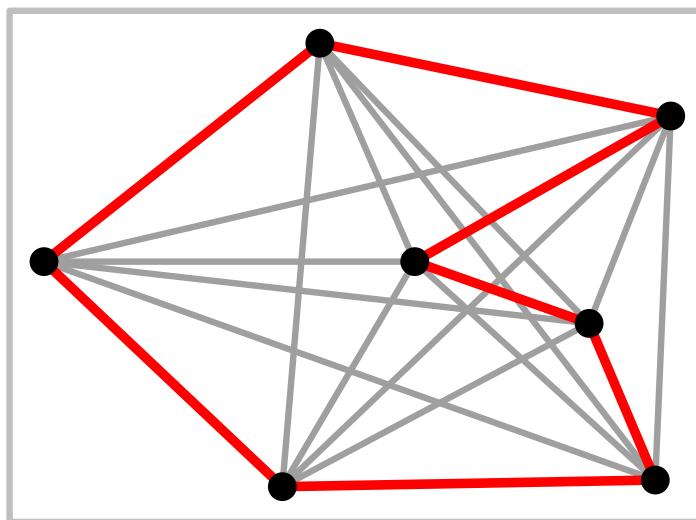
Definition. *Traveling Salesperson Problem (TSP)*

Gegeben: unger. vollständiger Graph $G = (V, E)$
mit Kantenkosten $c: E \rightarrow \mathbb{R}_{\geq 0}$

Gesucht: Hamiltonkreis K in G mit minimalen
Kosten $c(K) := \sum_{e \in K} c(e)$.
(Ein HK besucht jeden Knoten genau 1×.)

Beispiel.

$c \equiv d_{\text{Eukl.}}$



Problem.

- TSP ist NP-schwer
- und schwer zu approximieren.

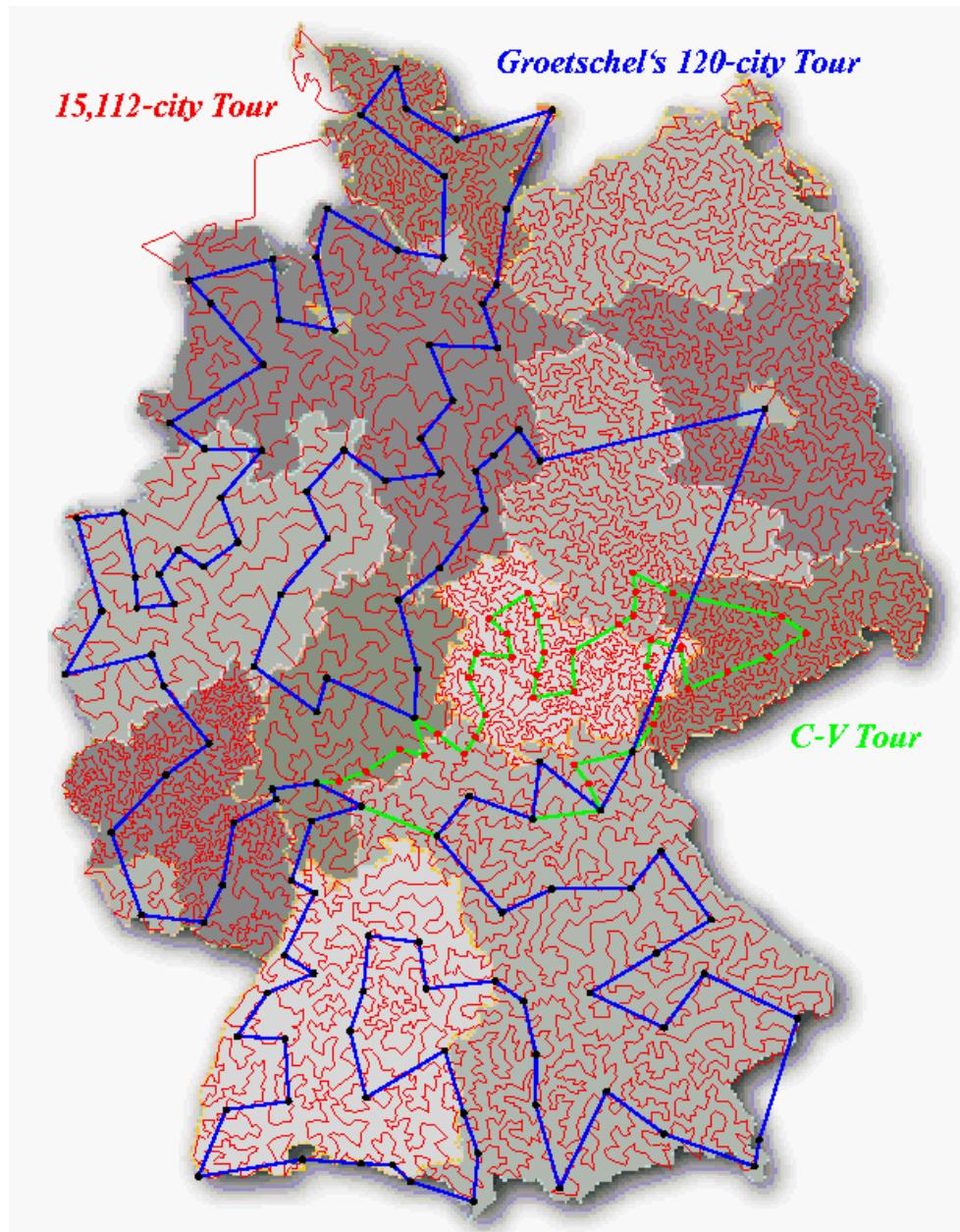


Etwas Geschichte

Der Handlungsreisende – wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein.
Von einem alten Commis-Voyageur [1832]

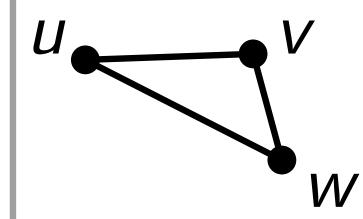
Rekord I:
optimale 120-Städte-Tour
[Groetschel, 1977]

Rekord II:
optimale 15.112-Städte-Tour
[Applegate, Bixby, Chvátal, Cook
2001]



Was tun? – Mach das Problem leichter!

Problem: *Metrisches Traveling Salesperson Problem (Δ -TSP)*



Gegeben: unger. vollständiger Graph $G = (V, E)$

mit Kantenkosten $c: E \rightarrow \mathbb{R}_{\geq 0}$,

die die Dreiecksungleichung erfüllen,

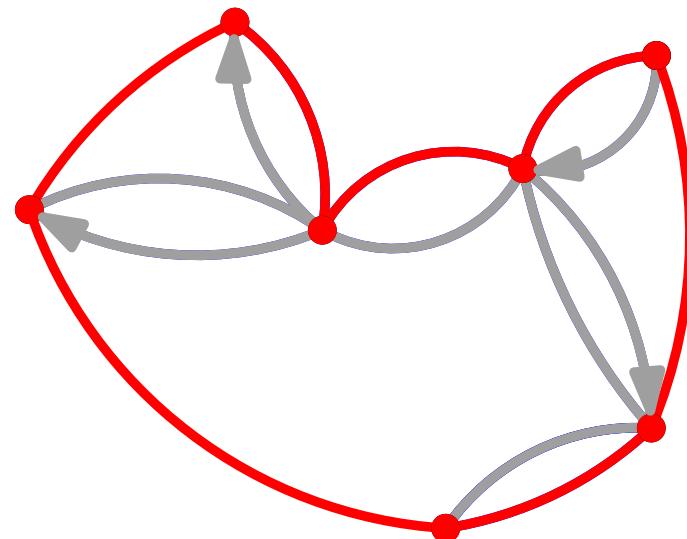
d.h. $\forall u, v, w \in V: c(u, w) \leq c(u, v) + c(v, w)$.

Gesucht: Hamiltonkreis in G mit minimalen Kosten.

Satz.

Es gibt eine 2-Approximation für Δ -TSP.

Beweis.



Algorithmus:

Berechne min. Spannbaum **MSB**.

Verdopple MSB \Rightarrow ergibt Kreis!

Durchlaufe den **Kreis**.

Überspringe besuchte Knoten.

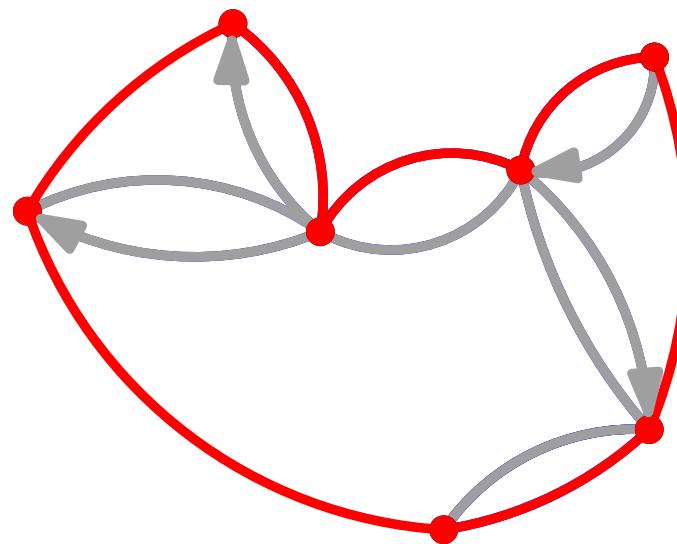
Füge „Abkürzungen“ ein.

Analyse

Satz.

Es gibt eine 2-Approximation für Δ -TSP.

Beweis.



1. Algorithmus

Berechne MSB von G .

Verdopple MSB \Rightarrow ergibt Kreis!

Durchlaufe den Kreis.

Überspringe besuchte Knoten.

Füge „Abkürzungen“ ein.

2. Analyse

$$c(\text{ALG}) \leq c(\text{Kreis}) = 2 \cdot c(\text{MSB}) \leq 2 \cdot \text{OPT}$$

Dreiecksungleichung

Optimale TSP-Tour minus eine Kante ist
(i.A. nicht minimaler) Spannbaum!!

Die „Kunst“ der unteren Schranke: $c(\text{min. Spannbaum}) \leq c(\text{TSP-Tour})$

Exakte Berechnung: Brute Force

Algorithmus: • Für jede Permutation σ von $\langle 1, 2, \dots, n \rangle$:

Berechne die Kosten der Tour durch die Knoten v_1, \dots, v_n in dieser Reihenfolge:

$$c(\sigma) = \sum_{i=1}^{n-1} c(v_{\sigma(i)} v_{\sigma(i+1)}) + c(v_{\sigma(n)} v_{\sigma(1)})$$

• Gib die kürzeste Tour zurück.

Laufzeit: Anzahl Permutationen von n Objekten: $n!$

Hält man den 1. Knoten fest, so bleiben „nur“ $(n - 1)!$ Permutationen.

Berechnung einer Tourlänge $c(\sigma)$: $O(n)$ Zeit.

Berechnung der nächsten Permutation: ???

Ang. ??? = $O(n)$, dann ist die Laufzeit $O(n!)$.

Speicher: $O(n)$ für die aktuelle Permutation.

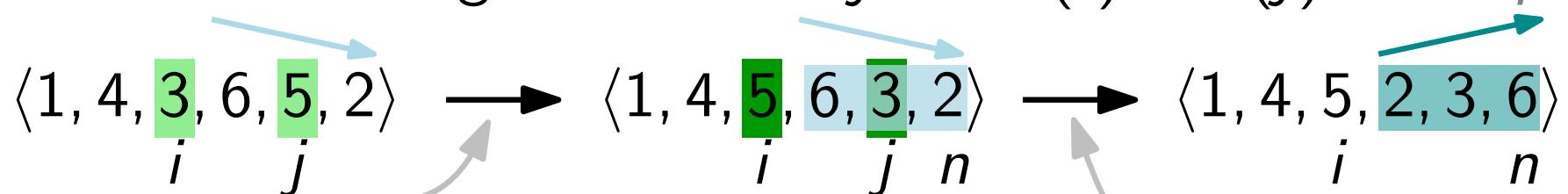
Wie iteriert man durch alle Permutationen?

Z.B. in lexikografischer Ordnung:

$\langle 1, 2, 3, 4, 5, 6 \rangle, \langle 1, 2, 3, 4, 6, 5 \rangle, \langle 1, 2, 3, 5, 4, 6 \rangle, \dots, \langle 6, 5, 4, 3, 2, 1 \rangle$.

Für gegebene Permutation σ , finde Nachfolger in $O(n)$ Zeit:

- Bestimme größten Index $i \in \{1, \dots, n-1\}$ mit $\sigma(i) < \sigma(i+1)$.
- Falls nicht existiert, fertig (σ = letzte Permutation).
- Sonst bestimme größten Index j mit $\sigma(i) < \sigma(j)$. *Beispiel:*



- Vertausche $\sigma(i)$ und $\sigma(j)$.
- Kehre die Teilfolge $\langle \sigma(i+1), \sigma(i+2), \dots, \sigma(n) \rangle$ um.

Wie groß ist $n!$?

$$\underbrace{n/2 \cdot n/2 \cdot \dots \cdot n/2}_{n/2 \text{ mal}} \leq n! = 1 \cdot 2 \cdot \dots \cdot n \leq n \cdot n \cdot \dots \cdot n = n^n$$

$$\Rightarrow 2^{n/2 \log_2 n/2} \leq n! \leq n^n = (2^{\log_2 n})^n = 2^{n \log_2 n}$$

$$\Rightarrow n! \in 2^{\Theta(n \log n)}$$

Exakte Berechnung: Schneller per DP!

Wir beginnen alle Rundtouren im Knoten v_1 .

Für eine Knotenmenge $W \subseteq V \setminus \{v_1\}$ mit $v_i \in W$ definieren wir
 $\text{OPT}[W, v_i] :=$ optimale (kürzeste) Länge eines v_1-v_i -Wegs
durch alle Knoten in W .

Schritt 2 für DP: *Definiere Wert einer opt. Lösung rekursiv!*

Dann gilt für $W = \{v_i\}$:

$$\text{OPT}[W, v_i] = c(v_1, v_i)$$

Und für W mit $\{v_i\} \subsetneq W$:

$$\text{OPT}[W, v_i] = \min_{v_j \in W \setminus \{v_i\}} \text{OPT}[W \setminus \{v_i\}, v_j] + c(v_j, v_i)$$

$$\Rightarrow \text{OPT} = \min_{k \neq 1} \text{OPT}[\{v_2, v_3, \dots, v_n\}, v_k] + c(v_k, v_1)$$

Der Algorithmus von Held-Karp

Schritt 3 für DP: Berechne Wert einer opt. Lsg. (hier: bot.-up)!

HeldKarp(Knotenmenge V , Abstände $c: V \times V \rightarrow \mathbb{R}_{\geq 0}$)

```

for  $i = 2$  to  $n$  do
     $\text{OPT}[\{v_i\}, v_i] = c(v_1, v_i)$ 
for  $j = 2$  to  $n - 1$  do
    foreach  $W \subseteq \{v_2, \dots, v_n\}$  mit  $|W| = j$  do
        foreach  $v_i \in W$  do
             $\text{OPT}[W, v_i] = \min_{v_j \in W \setminus \{v_i\}} \text{OPT}[W \setminus \{v_i\}, v_j] + c(v_j, v_i)$ 
return  $\min_{k \neq 1} \text{OPT}[\{v_2, v_3, \dots, v_n\}, v_k] + c(v_k, v_1)$ 
```

Laufzeit: Berechnung von $\text{OPT}[W, v_i]$: $O(n)$ Zeit

Wie viele Paare (W, v_i) mit $v_i \in W$ gibt's? $\leq n \cdot 2^{n-1}$

\Rightarrow Gesamlaufzeit $\in O(n^2 \cdot 2^n)$ **Speicher:** $O(n \cdot 2^n)$

Vergleich

	Brute Force	Held-Karp
Laufzeit	$2^{\Theta(n \log n)}$	$O(n^2 \cdot 2^n)$
Speicher	$O(n)$	$O(n \cdot 2^n)$

Der Algorithmus von Held und Karp verringert also die Laufzeit zu Kosten des Speicherplatzverbrauchs.

Das bezeichnet man als Laufzeit-Speicherplatz- *Trade-Off*.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
25. Vorlesung

Leichte Kreise in Graphen

Kürzeste Kreise

- Gewichteter und ungewichteter Fall
- Gerichteter und ungerichteter Fall

Z.B. für einen ungewichteter und ungerichteter Graphen G :

Für einen Knoten v liefert $\text{BFS}(G, v)$ – bis zur ersten Nicht-Baumkante – einen kürzesten Kreis C_v durch v .

Der kürzeste der Kreise in der Menge $\{C_v \mid v \in V\}$ ist ein kürzester Kreis in G .

Laufzeit: $O(VE)$

Minimales durchschnittliches Kantengewicht

Sei $G = (V, E)$ ein gerichteter Graph mit beliebigen Kantengewichten $w: E \rightarrow \mathbb{R}$. Sei $n = |V|$.

Für einen gerichteten Kreis $C = \langle e_1, e_2, \dots, e_k \rangle$ sei

$$\mu(C) = \frac{1}{k} \sum_{i=1}^k w(e_i)$$

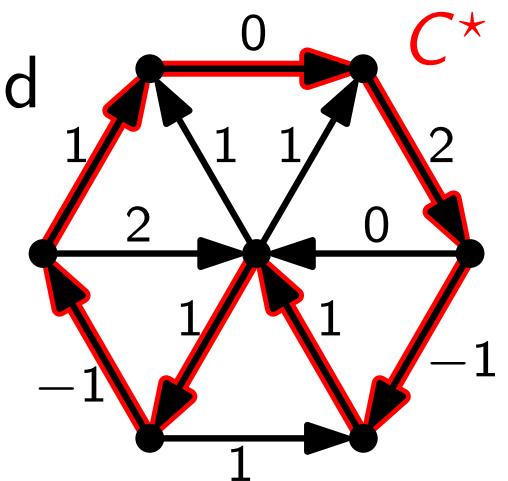
sein *durchschnittliches* Kantengewicht.

$$\mu^* = \mu(C^*) = \frac{3}{7}$$

Sei \mathcal{C} die Menge aller gerichteter Kreise in G und

$$\mu^* = \min_{C \in \mathcal{C}} \mu(C)$$

das minimale durchschnittliche Kantengewicht eines Kreises (*minimum mean cycle weight*).



Rohe Gewalt

Wir suchen also einen Kreis C^* mit $\mu(C^*) = \mu^*$, d.h. einen Kreis mit minimalem durchschnittlichem Kantengewicht.

MinMeanCycleBruteForce(DirectedWeightedGraph G, w)

$$\mu_{\min} = \infty$$

foreach $C = \langle e_1, e_2, \dots, e_k \rangle \in \mathcal{C}$ **do**

$$\mu = \frac{1}{k} \sum_{i=1}^k w(e_i)$$

if $\mu < \mu_{\min}$ **then**

$$\mu_{\min} = \mu$$

$$C' = C$$

return C'

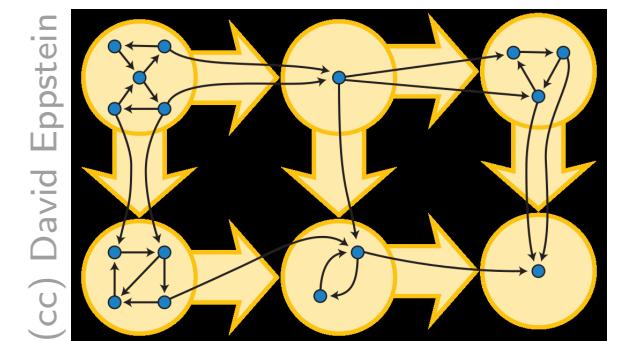
Laufzeit? *Mindestens exponentiell in $|V|$:-(
höchstens exponentiell in $|E|$*

Vorbereitungen

Wir nehmen an, dass G stark zusammenhängend ist, d.h. es gibt für jedes Knotenpaar (u, v) einen gerichteten u - v -Weg.

Ansonsten zerlegen wir G in seine starken Zusammenhangskomponenten (wie?*) und betrachten jede separat.

Sei s ein beliebiger Knoten von G .



Sei $\delta(s, v)$ das Gewicht eines kürzesten (leichtesten) s - v -Wegs.

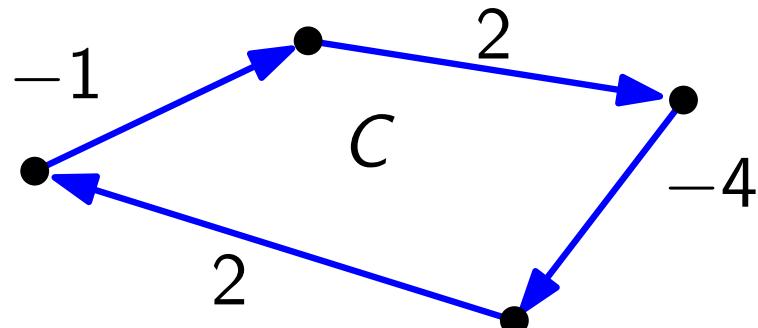
Für $k = 0, \dots, n - 1$ sei $\delta_k(s, v)$ das Gewicht eines kürzesten s - v -Wegs, der aus *genau k* Kanten besteht (sonst ∞).

*) Im Prinzip durch ein oder zwei Tiefensuchen (siehe https://en.wikipedia.org/wiki/Strongly_connected_component)

Schritt I

Zeige: Falls $\mu^* = 0$, dann gilt:

1. G hat keinen Kreis mit negativem Gewicht und ✓
2. $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ für jeden Knoten v . ✓



Beweis.

1. Angenommen es gäbe einen Kreis C mit $w(C) < 0$.
 $\Rightarrow \mu(C) < 0 \Rightarrow \mu^* < 0$ 
2. Betrachte s - v -Weg π mit $k > n - 1$ Kanten.
 $\Rightarrow \pi$ enthält Kreis C . Aber $w(C) \geq 0$. $\Rightarrow w(\pi \setminus C) \leq w(\pi)$
 \Rightarrow Es gibt einen kürzesten s - v -Weg mit $\leq n - 1$ Kanten.

Schritt II

Falls $\mu^* = 0$, dann gilt:

- G hat keinen Kreis mit negativem Gewicht und
- $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ für jeden Knoten v . (*)

Zeige: Falls $\mu^* = 0$, dann gilt für jeden Knoten v

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0.$$

Beweis: Nach Def. von δ gilt: $\delta_n(s, v) \geq \delta(s, v)$

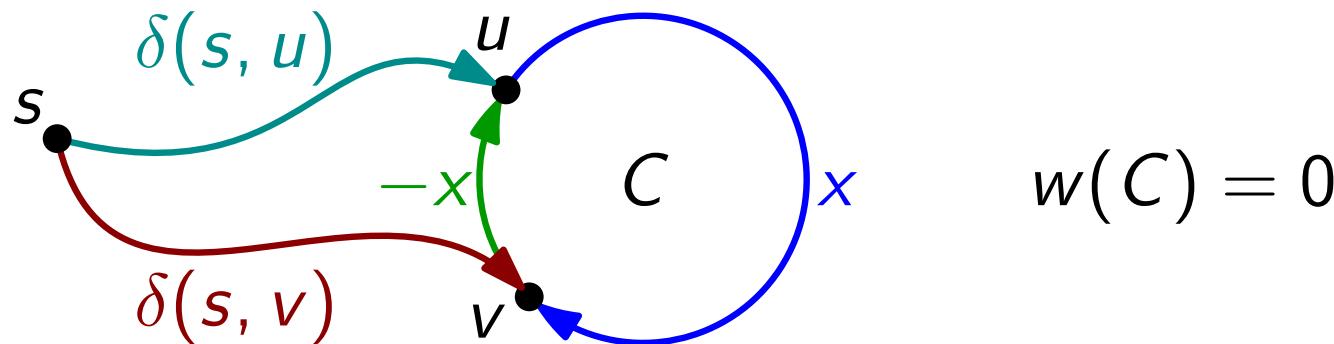
Wegen (*) gilt: $\delta(s, v) = \delta_k(s, v)$ für ein $k \in \{0, \dots, n-1\}$

Also gilt $\delta_n(s, v) \geq \delta_k(s, v)$ für ein $k \in \{0, \dots, n-1\}$

$\Rightarrow \max_{0 \leq k \leq n-1} \delta_n(s, v) - \delta_k(s, v) \geq 0 \Rightarrow$ Beh. $n - k > 0$ 

Schritt III

Sei C ein Kreis mit Gewicht 0. Seien u, v Knoten auf C .
 Sei x das Gewicht des Wegs von u nach v auf C .



Zeige: $\delta(s, v) = \delta(s, u) + x$.

Klar: $\delta(s, v) \leq \delta(s, u) + x$.

Aber warum kann es keinen kürzeren Weg von s nach v geben?

Angenommen, es gälte $\delta(s, v) < \delta(s, u) + x$.

Dann gäbe es einen Weg von s über v nach u der Länge...

$\delta(s, v) - x < (\delta(s, u) + x) - x = \delta(s, u)$ ⚡ zur Def. von δ .



Schritt IV

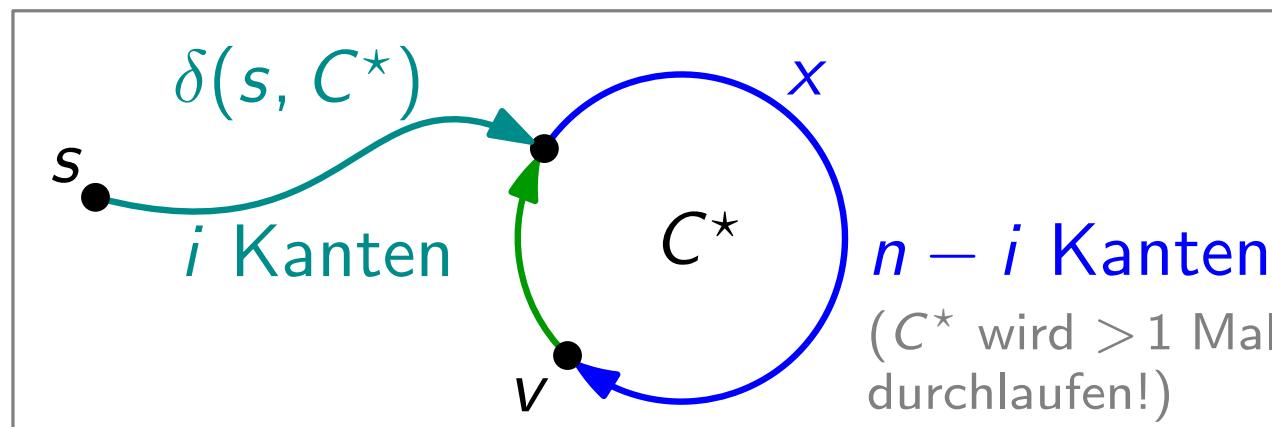
Falls $\mu^* = 0$, dann gilt für jeden Knoten v

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0.$$

Zeige:

Falls $\mu^* = 0$, dann gibt es einen Knoten v auf dem Kreis C^* , so dass

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$



Schritt III

$$\Rightarrow \delta_n(s, v) = \delta(s, v).$$

$$\Rightarrow \delta_n(s, v) \leq \delta_k(s, v).$$

Aber für welches k gilt
 $\delta_n(s, v) = \delta_k(s, v)$?

Schritt V

Falls $\mu^* = 0$, dann gilt für jeden Knoten v

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0.$$

Falls $\mu^* = 0$, dann gibt es einen Knoten v auf dem Kreis C^* , so dass

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

Zeige:

Falls $\mu^* = 0$, dann

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

Klar...

Schritt VI

Falls $\mu^* = 0$, dann

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

Zeige:

Falls wir eine Konstante t zum Gewicht jeder Kante von G addieren, dann steigt μ^* um t .

Schritt VI

Falls $\mu^* = 0$, dann

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

Zeige:

Falls wir eine Konstante t zum Gewicht jeder Kante von G addieren, dann steigt μ^* um t .

Zeige damit, dass

$$\mu^* \stackrel{?}{=} \min_{v \in V} \max_{0 \leq k \leq n-1} \underbrace{\frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}}_{\text{ }}$$

Zeige: Steigt auch um t , wenn alle Gew. um t erhöht werden.

Schritt VI

Falls $\mu^* = 0$, dann

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

Zeige:

Falls wir eine Konstante t zum Gewicht jeder Kante von G addieren, dann steigt μ^* um t .

$$+ \frac{nt - kt}{n - k} = +t$$

Zeige damit, dass

$$\mu^* \stackrel{?}{=} \min_{v \in V} \max_{0 \leq k \leq n-1} \underbrace{\frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}}_{\substack{+nt \\ +kt}}.$$

Zeige: Steigt auch um t , wenn alle Gew. um t erhöht werden.



Schritt VI

Falls $\mu^* = 0$, dann

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0. \quad \left. \right\} (**)$$

Zeige:

Falls wir eine Konstante t zum Gewicht jeder Kante von G addieren, dann steigt μ^* um t .

$$+ \frac{nt - kt}{n - k} = +t$$

Zeige damit, dass

$$\alpha(t) := \boxed{\mu^*} \stackrel{?}{=} \boxed{\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}} =: \beta(t)$$

Zeige: Steigt *auch* um t , wenn alle Gew. um t erhöht werden.

Also: α und β sind *lineare* Fkt. in t mit $\alpha(-\mu^*) = \beta(-\mu^*)$ und Steigung 1 $\Rightarrow \alpha \equiv \beta$. (**) ✓



Schritt VII

Es gilt

$$\mu^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}. \quad (***)$$



[Karp, 1978]

Satz. Ein Kreis C^* mit kleinstem durchschnittlichen Kantengewicht ($\mu(C^*) = \mu^*$) lässt sich in $O(VE)$ Zeit berechnen.

Gib einen Algorithmus an, der μ^* in $O(VE)$ Zeit berechnet:

- Setze $\delta_0(s, s) = 0$ und, für $v \in V \setminus \{s\}$, setze $\delta_0(s, v) = \infty$.
- Für $k = 1, \dots, n-1$ und $v \in V$, berechne in $O(\text{indeg } v)$ Zeit

$$\delta_k(s, v) = \min_{uv \in E} \delta_{k-1}(s, u) + w(u, v).$$

Dies benötigt insg. $O(VE)$ Zeit.

- Berechne μ^* nach $(***)$ in $O(V^2)$ Zeit. □