Algorithmen und Datenstrukturen

TECHNISCHE UNIVERSITÄT DARMSTADT

SYSTEMS

Stefan Roth, SS 2025

03

Grundlegende Datenstrukturen

Folien beruhen auf der Veranstaltung von Prof. Marc Fischlin und Christian Janson aus dem SS 2024

Stacks

zur Erinnerung

Abstrakte Datentypen (ADTs) und Datenstrukturen

näher an der Anwendung

Abstrakter Datentyp („was“)

Übergang fließend; ADTs werden daher oft auch als Datenstruktur bezeichnet

Datenstruktur („wie“)

näher „an der Maschine“

Beispiel:

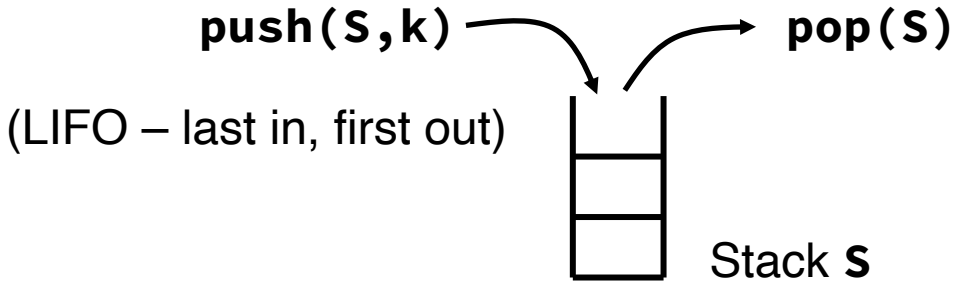
Stack mit Operationen
`isEmpty,pop,push`

Stack-Operationen
als Array
oder verkettete Liste

Abstrakter Datentyp Stack

Bemerkung: auch Schreibweise **`S.push(k)`** oder einfach **`pop()`**

`new(S)`	-	erzeugt neuen (leeren) Stack namens **`S`**
`isEmpty(S)`	-	gibt an, ob Stack **`S`** leer
`pop(S)`	-	gibt oberstes Element vom Stack **`S`** zurück und löscht es vom Stack (bzw. Fehlermeldung, wenn Stack leer)
`push(S,k)`	-	schreibt **`k`** als neues oberstes Element auf Stack **`S`** (bzw. Fehlermeldung, wenn Stack voll)



Formale Erfassung z.B.
per algebraischer Spezifikation

Algebraische Spezifikation von Stacks

Stack mit Operationen **`new,isEmpty,push,pop`** muss folgende Regeln erfüllen:

(1) Wenn **`new(S)`**, dann unmittelbar **`isEmpty(S)`**, ergibt Ergebnis **`true`**.

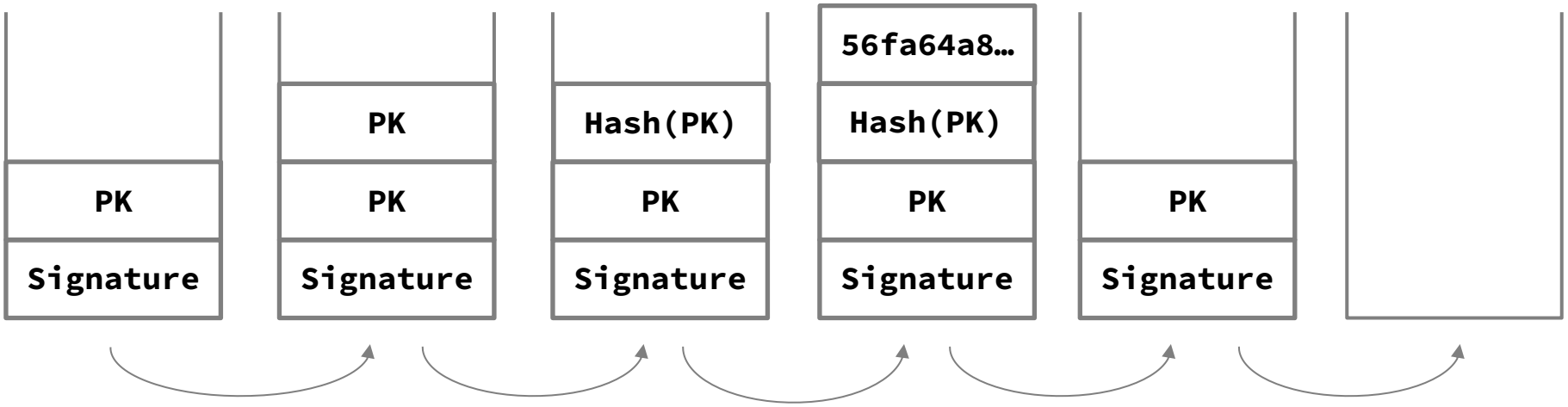
(2) Wenn **`push(S,k)`** und keine Fehlermeldung, dann unmittelbar **`pop(S)`**, ergibt Ergebnis **`k`**.

(3) …

Fokus dieses Teils der Vorlesung liegt auf Entwurf der Datenstrukturen; alle Lösungen erfüllen „natürliche“ Forderungen an solche Operationen.

Beispiel: Bitcoin

```
scriptPubKey:
OP_DUP OP_HASH160 56fa64a8bd7852d2c58095fa9a2fcd52d2c580b65d35549d
OP_EQUALVERIFY OP_CHECKSIG
```



entspricht
```
k=pop()
push(k)
push(k)
```

entspricht
```
k=pop()
//h=Hash(k)
push(h)
```

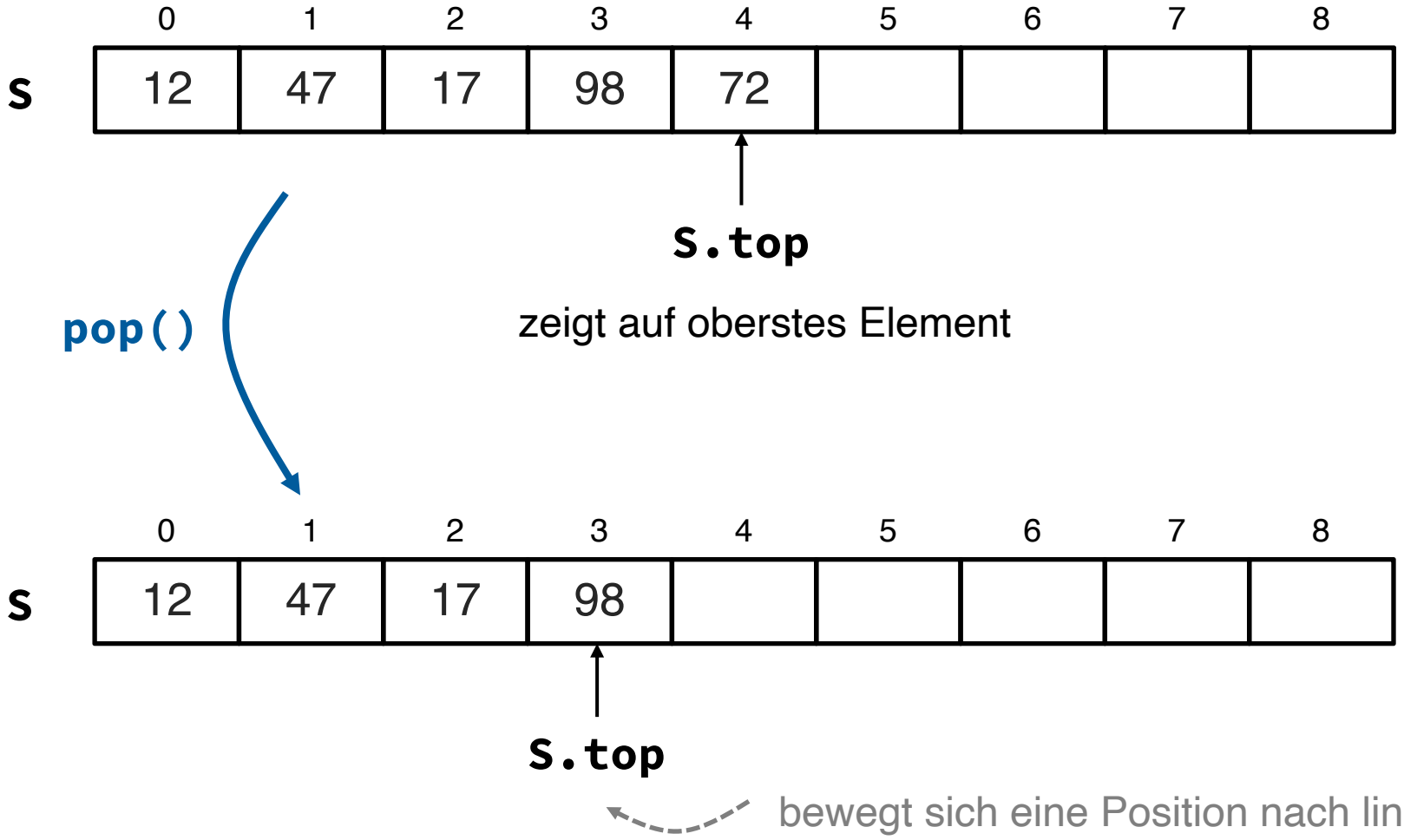
entspricht
```
push(56fa…)
```

entspricht
```
k=pop()
h=pop()
//k==h?
```

entspricht
```
k=pop()
h=pop()
//k,h gültig?
```

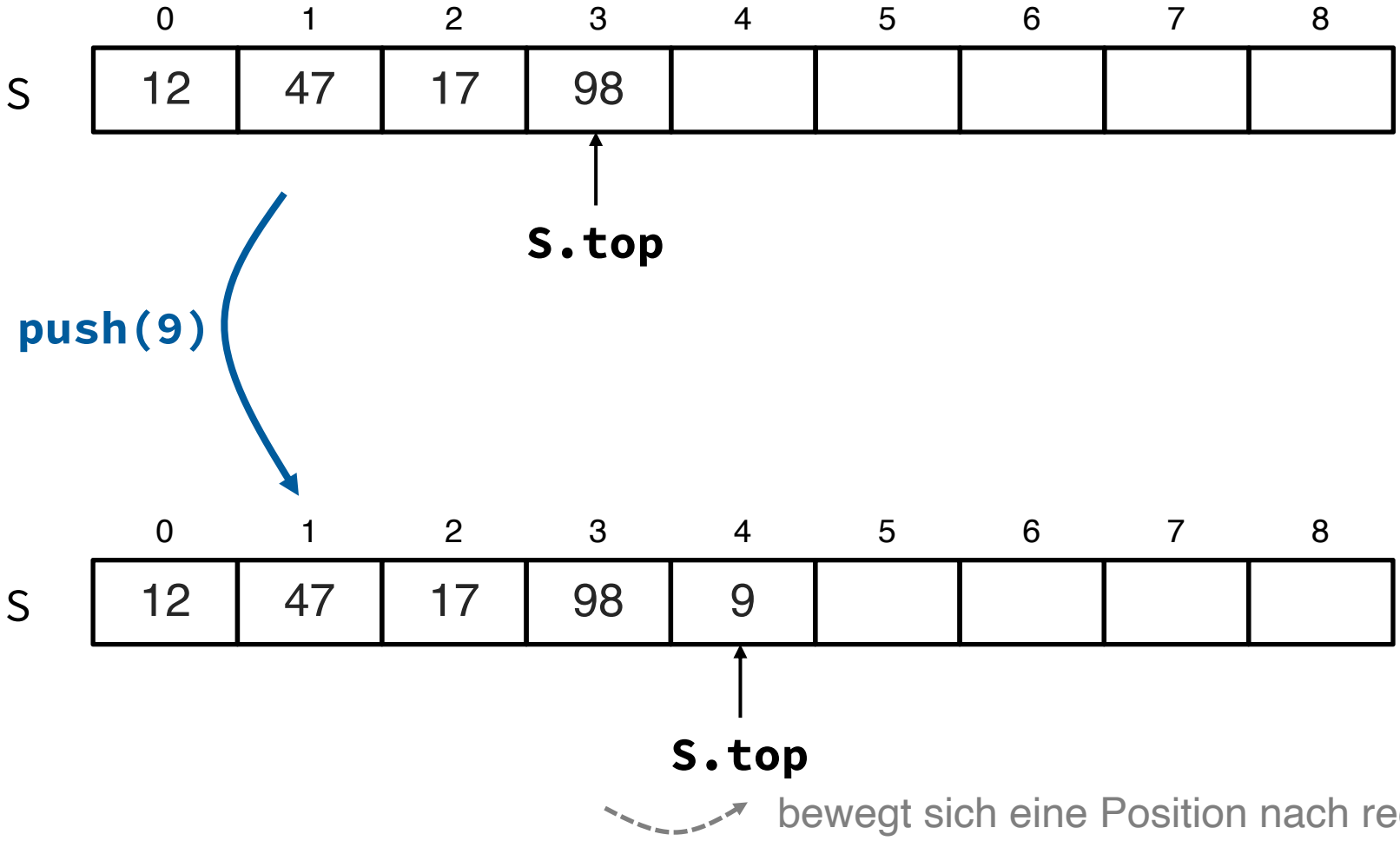
Stacks als Array (I)

Annahme: maximale Größe MAX des Stacks vorher bekannt



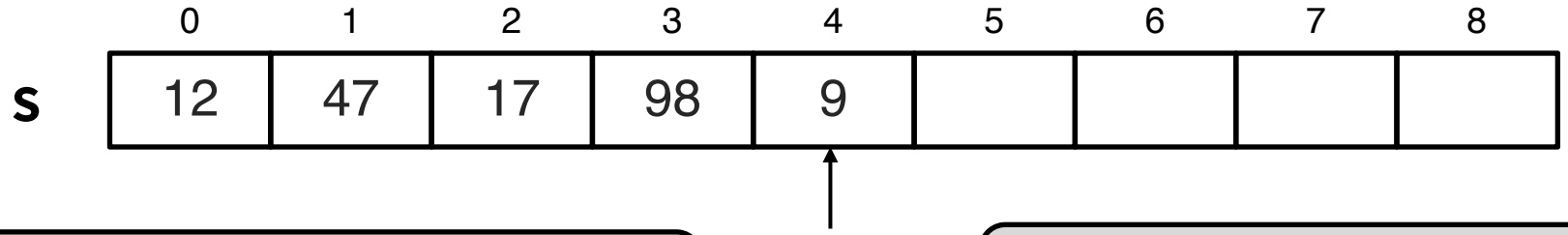
Stacks als Array (II)

Annahme: maximale Größe MAX
des Stacks vorher bekannt



Stacks als Array: Algorithmen

Annahme: maximale Größe MAX des Stacks vorher bekannt



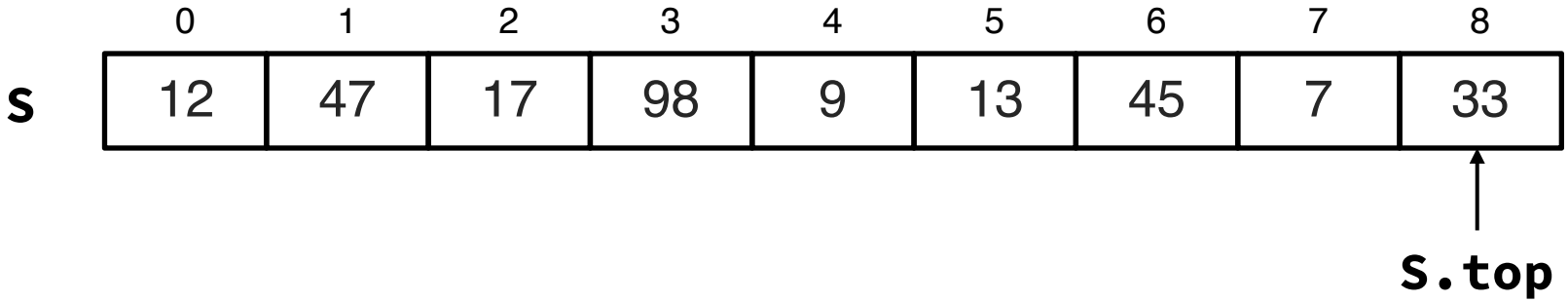

```
new(S)

1  S.A[]=ALLOCATE(MAX);
2  S.top=-1;
```

```
isEmpty(S)

1  IF S.top<0 THEN
2     return true
3  ELSE
4     return false;
```

```
pop(S)

1  IF isEmpty(S) THEN
2     error ‘underflow’
3  ELSE
4     S.top=S.top-1;
5     return S.A[S.top+1];
```

```
push(S,k)

1  IF S.top==MAX-1 THEN
2     error ‘overflow’
3  ELSE
4     S.top=S.top+1;
5     S.A[S.top]=k;
```

Stacks mit variabler Größe



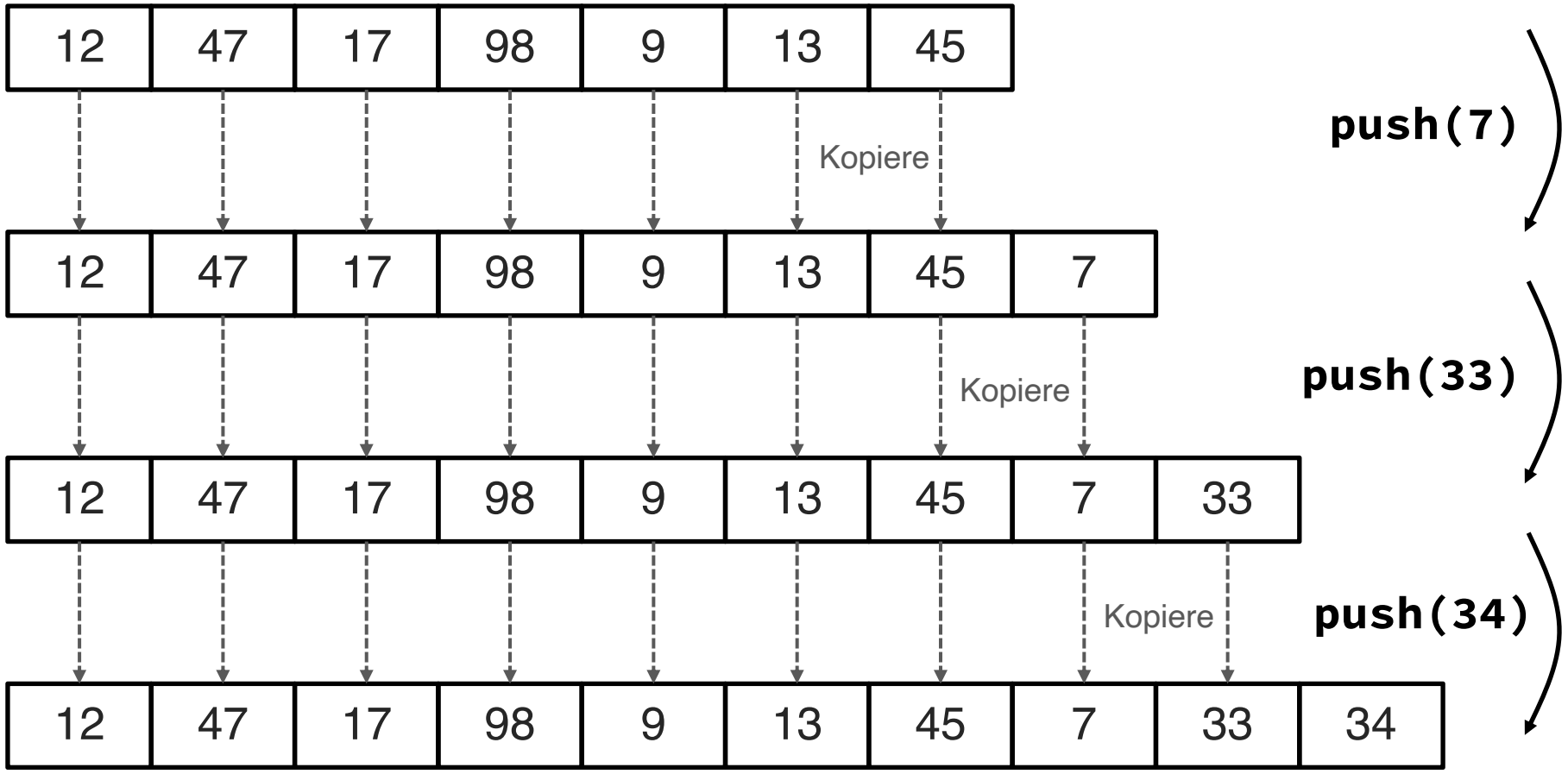
push(14)

Kopiere entweder in größeres, zusammenhängendes Array um,

oder verteile auf viele Arrays (→ Datenstruktur verkette Listen)

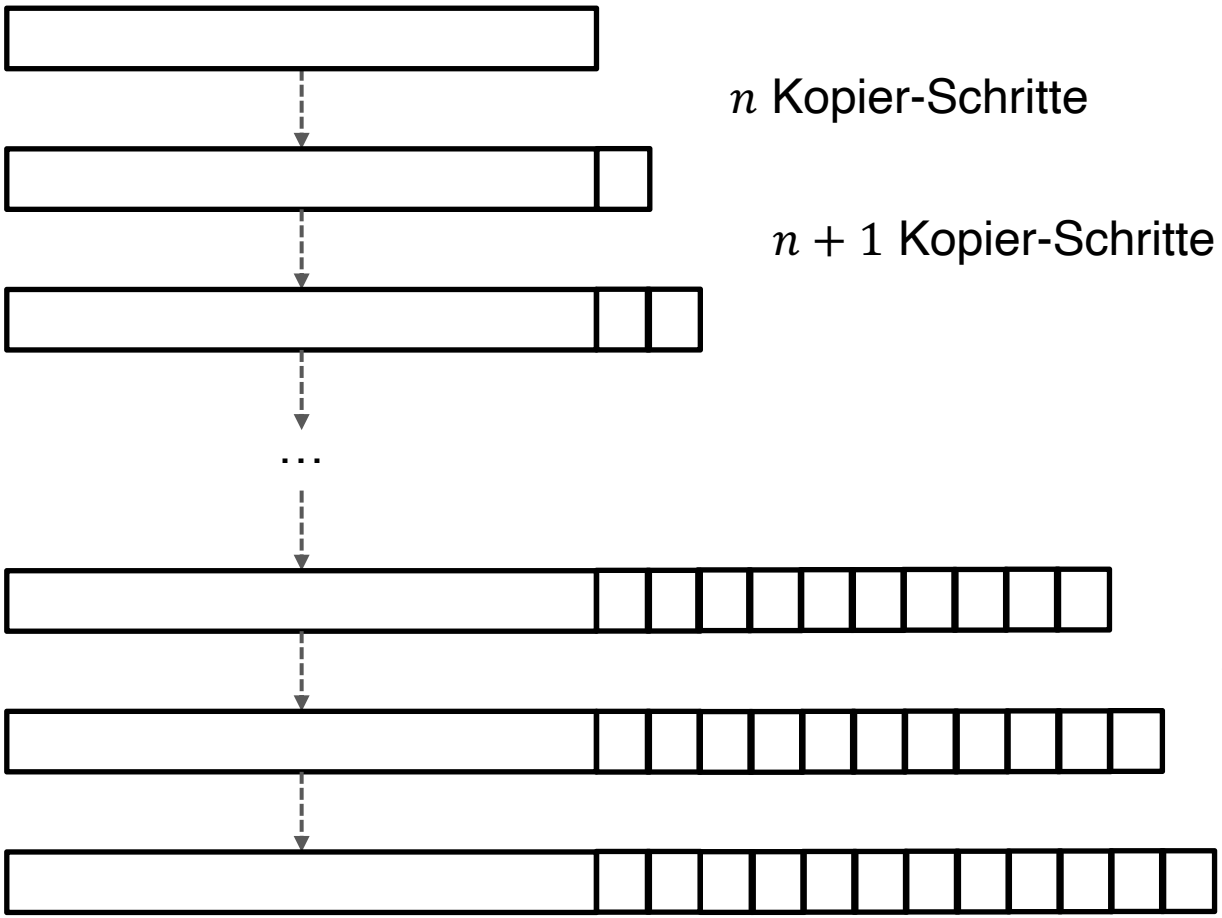
Einfache Feldarbeit

Erzeuge bei Bedarf neues Array mit zusätzlichem Eintrag und kopiere aktuellen Stack um



Laufzeit

$n = MAX$ Elemente in Liste und n weitere **push**-Befehle



n Kopier-Schritte

$n + 1$ Kopier-Schritte

$2n - 2$ Kopier-Schritte

$2n - 1$ Kopier-Schritte

In Summe also
$n^2 + \sum_{i=0}^{n-1} i = \Omega(n^2)$
Kopier-Schritte

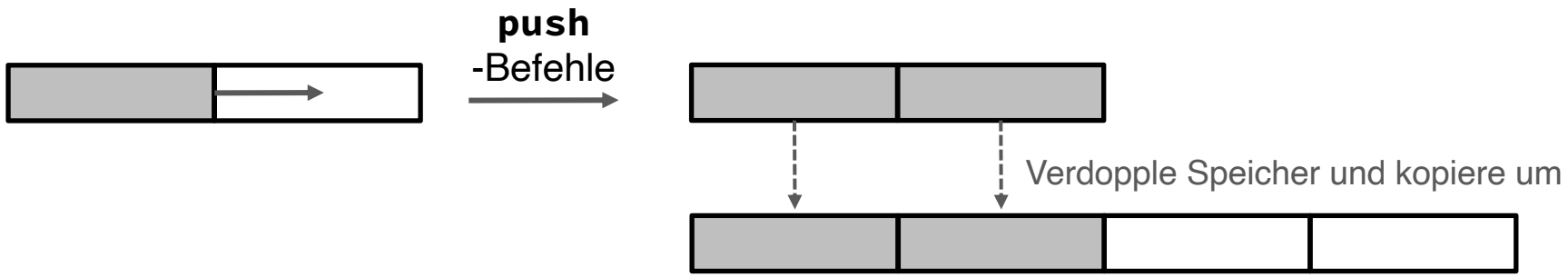
Somit durchschnittlich
$\Omega(n)$ Kopier-Schritte
pro **push**-Befehl!

Verbesserung?

Triviale Lösung: reserviere „unendlich“ viel Speicher

Gesucht: Lösung, die maximal jeweils $\mathcal{O}(\#Elemente)$ Zellen benötigt

Idee: Wenn Grenze erreicht, verdoppele Speicher und kopiere um



Schrumpfe und kopiere um, sofern weniger als ein Viertel belegt

Feldarbeit: Algorithmen

RESIZE(S,m)
reserviert neuen Speicher der Größe **m**,
kopiert **S.A** um, und lässt **S.A** auf neuen Speicher zeigen

```
new(S)

1  S.A[]=ALLOCATE(1);
2  S.top=-1;
3  S.memsize=1;
```

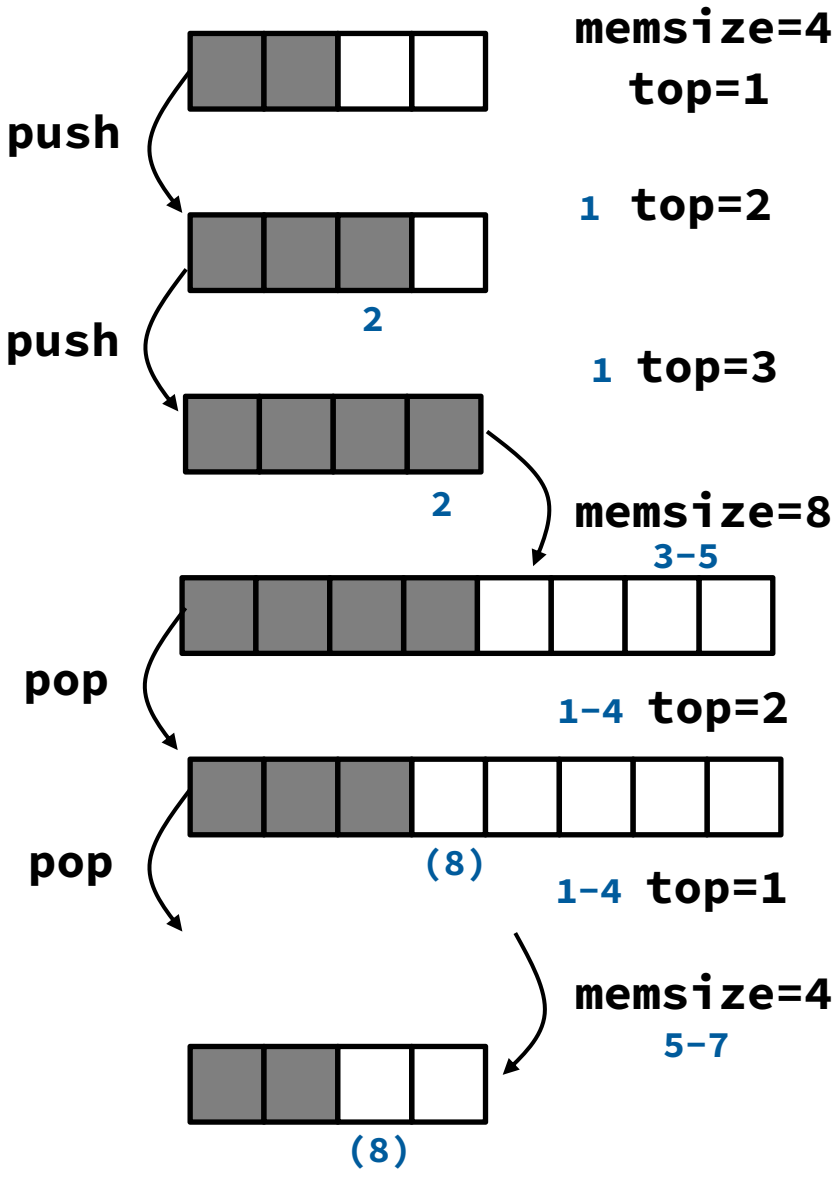
```
isEmpty(S)

1  IF S.top<0 THEN
2     return true
3  ELSE
4     return false;
```

```
pop(S)

1  IF isEmpty(S) THEN
2    error ‘underflow‘
3  ELSE
4    S.top=S.top-1;
5    IF 4*(S.top+1)==S.memsize THEN
6       S.memsize=S.memsize/2;
7       RESIZE(S,S.memsize);
8    return S.A[S.top+1];
```

```
push(S,k)

1  S.top=S.top+1;
2  S.A[S.top]=k;
3  IF S.top+1==S.memsize THEN
4    S.memsize=2*S.memsize;
5    RESIZE(S,S.memsize);
```

```
push(S,k)

1  S.top=S.top+1;
2  S.A[S.top]=k;
3  IF S.top+1==S.memsize THEN
4     S.memsize=2*S.memsize;
5     RESIZE(S,S.memsize);
```

```
pop(S)

1  IF isEmpty(S) THEN
2     error ‘underflow‘
3  ELSE
4     S.top=S.top-1;
5     IF 4*(S.top+1)==S.memsize THEN
6        S.memsize=S.memsize/2;
7        RESIZE(S,S.memsize);
8     return S.A[S.top+1];
```

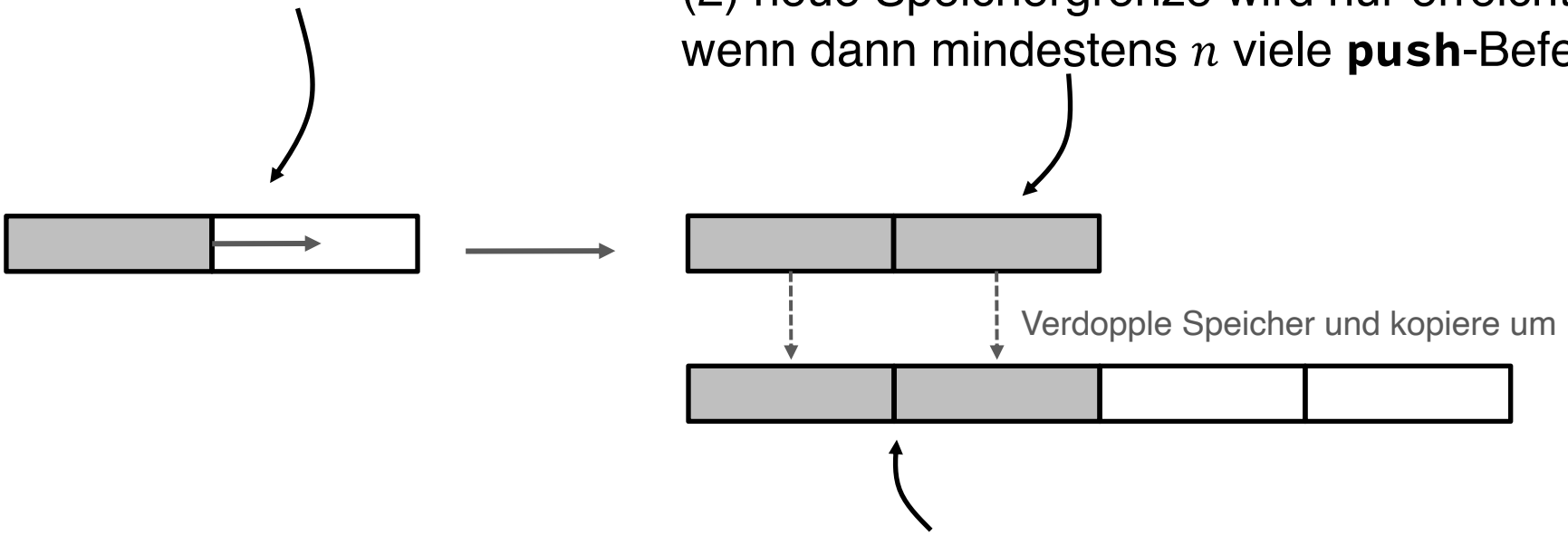


Analyse Laufzeit

Vergrößern (gilt analog für Verkleinern)

(1) n Elemente (unmittelbar nach letzter Vergrößerung)

(2) neue Speichergrenze wird nur erreicht, wenn dann mindestens n viele **push**-Befehle



(3) Umkopieren kostet dann $\mathcal{O}(n)$ Schritte

Im Durchschnitt für jeden der mindestens n Befehle $\Theta(1)$ Umkopierschritte!

Stacks in Java

Class Stack in Java 13 bis 24

Quelle: Wikipedia

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

```
public class Stack<E>
extends Vector<E>
```

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vec`

`capacityIncrement` gibt an, wieviel **`Vector`** wachsen soll, wenn zu viele Elemente (default = 2)

Fields

Modifier and Type	Field	Description
`protected int`	`capacityIncrement`	The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
`protected int`	`elementCount`	The number of valid components in this `Vector` object.
`protected Object[]`	`elementData`	The array buffer into which the components of the vector are stored.

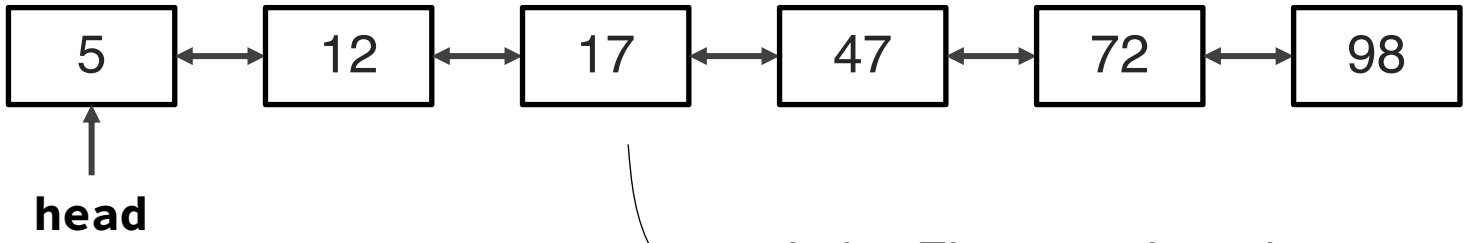
Geben Sie eine „schlechte“ Eingabe für die Java-Klasse **`Stack`** an, bei der wegen des fehlenden Schrumpfens viel Speicher verschwendet wird.

Stellen Sie die Operation **`clear`** für einen Stack in Pseudocode dar, die den Stack leert.

Verkettete Listen

Datenstruktur Verkettete Listen

(doppelt) verkettete Liste

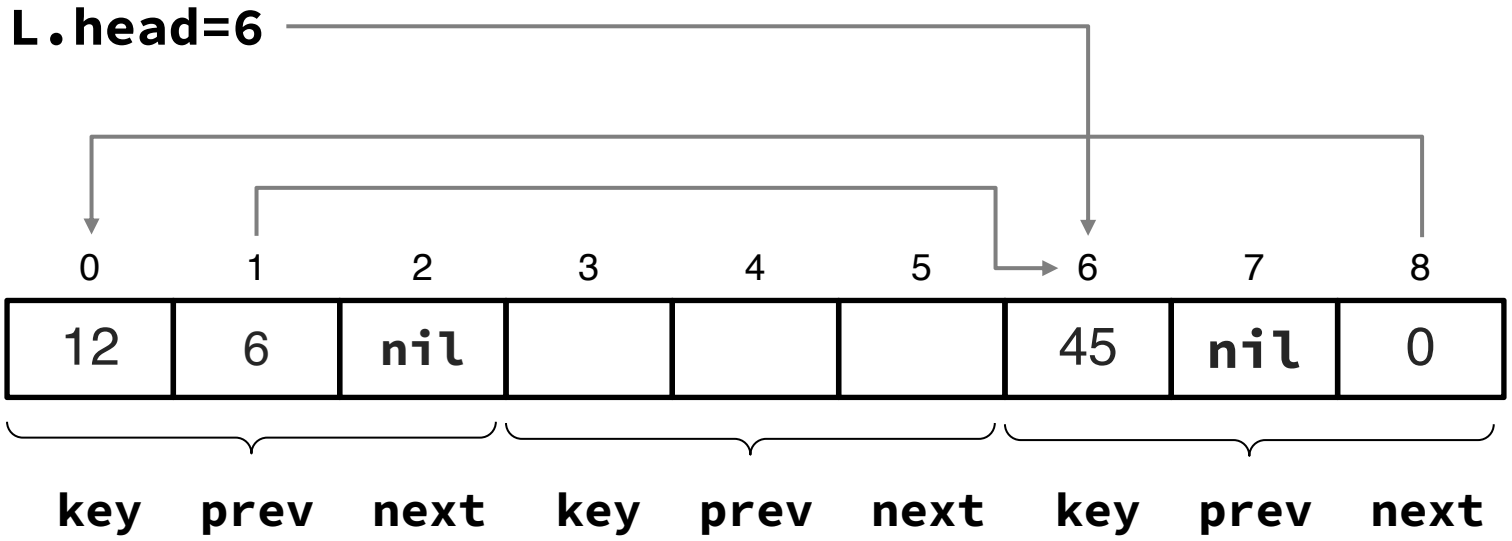


zeigt auf erstes Element (bzw. ist **`nil`** für leere Liste)

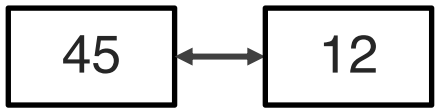
Jedes Element **`x`** besteht aus:
`key` – Wert
`prev` – Zeiger auf Vorgänger (bzw. **`nil`**)
`next` – Zeiger auf Nachfolger (bzw. **`nil`**)

evtl. schon als Datenstruktur implementiert, oder aber…

Datenstruktur Verkettete Listen durch Arrays



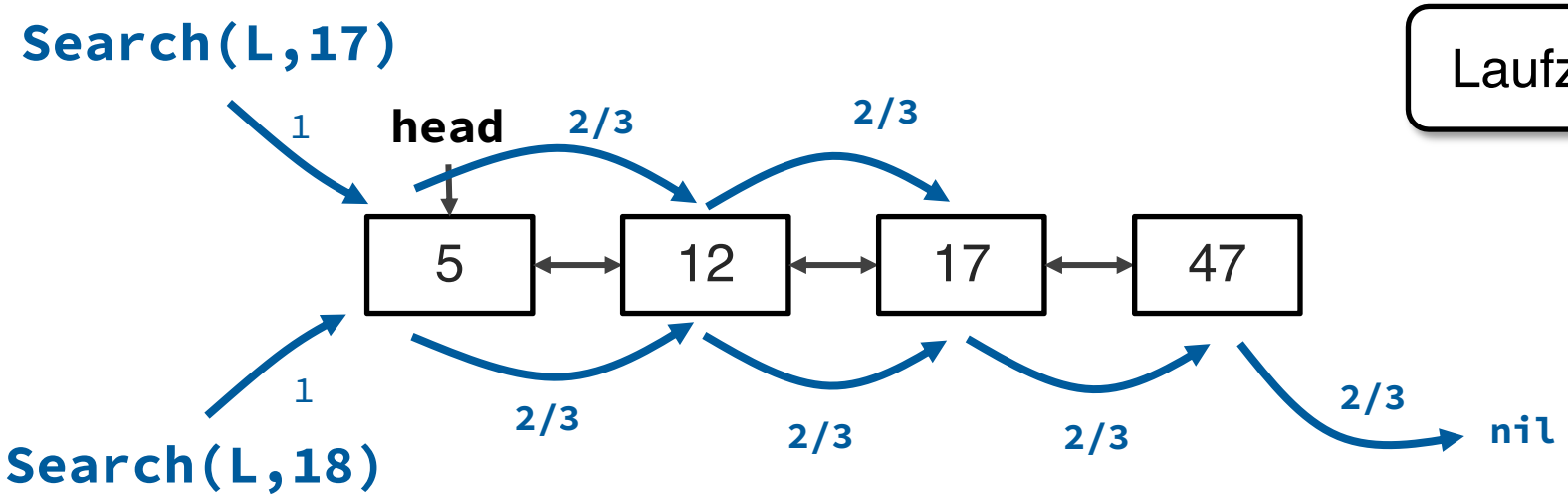
entspricht doppelt verketteter Liste



Elementare Operationen auf verketteten Listen

```
search(L,k)   //returns pointer to k in L (or nil)

1  current=L.head;

2  WHILE current != nil AND current.key != k DO
3     current=current.next;
4  return current;
```

short circuit evaluation (wie in Java)



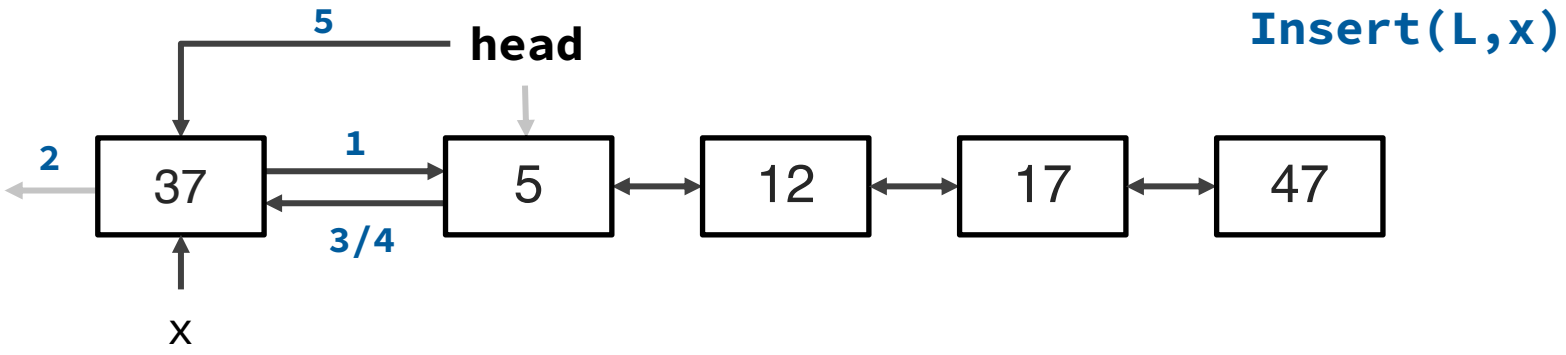
Laufzeit= $\Theta(n)$

Elementare Operationen auf verketteten Listen

```
insert(L,x)   //inserts element x in L

1  x.next=L.head;
2  x.prev=nil;
3  IF L.head != nil THEN
4     L.head.prev=x;
5  L.head=x;
```

call-by-reference
bzw. call-by-value
für Objekte wie in Java

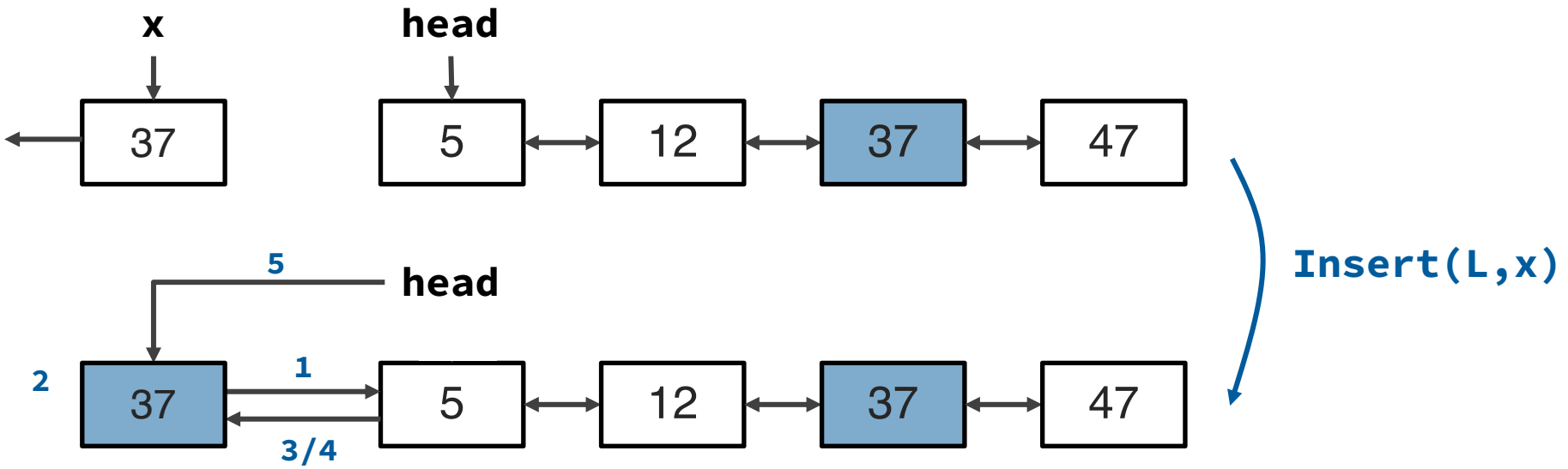
Laufzeit= $\Theta(1)$



Elementare Operationen auf verketteten Listen

Achtung: Einfüge-Operation prüft nicht, ob Wert bereits in Liste

Wenn zuerst Suche nach Wert, dann wiederum Laufzeit $\Omega(n)$!

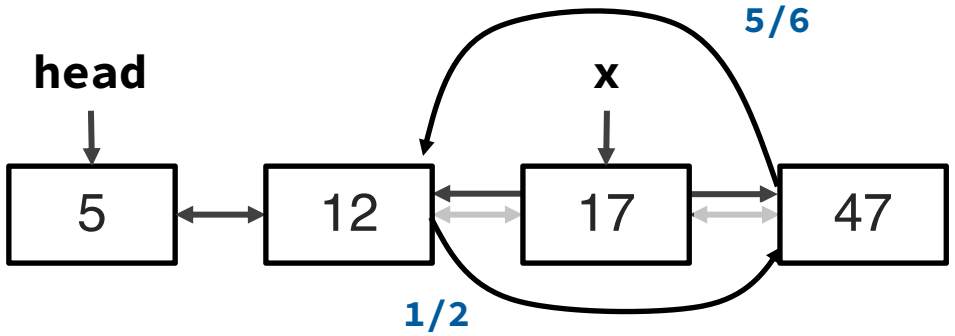


Elementare Operationen auf verketteten Listen

```
delete(L,x)   //deletes element x from L

1  IF x.prev != nil THEN
2       x.prev.next=x.next
3  ELSE
4       L.head=x.next;
5  IF x.next != nil THEN
6       x.next.prev=x.prev;
```

Laufzeit= $\Theta(1)$

Achtung: Löschen eines Wertes **k** kostet Zeit $\Omega(n)$

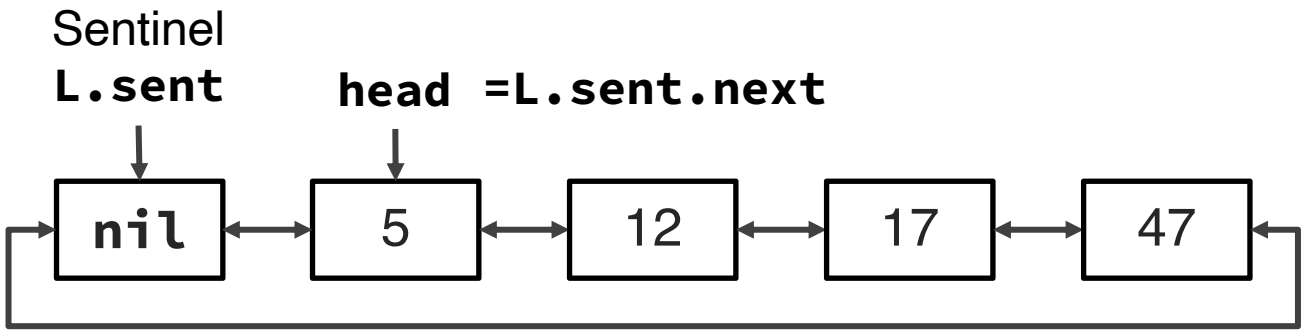


Delete(L,x)

Vereinfachung per Wächter/Sentinels

```
delete(L,x)   //deletes element x from L

1  IF x.prev != nil THEN
2      x.prev.next=x.next
3  ELSE
4      L.head=x.next;
5  IF x.next != nil THEN
6      x.next.prev=x.prev;
```

Ziel:
eliminiere die
Spezialfälle für
Listenanfang/-ende



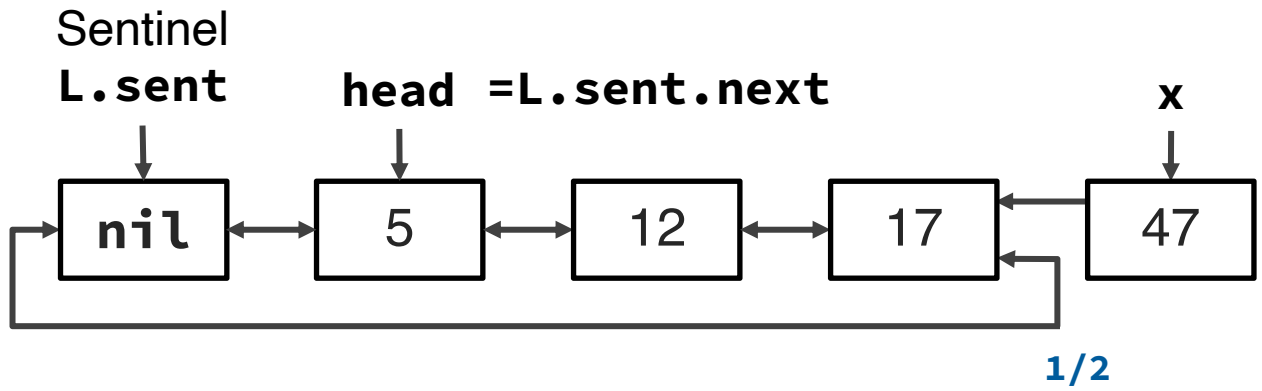
Sentinel ist „von außen“ nicht sichtbar

Leere Liste besteht nur aus Sentinel

Löschen mit Sentinels

```
deleteSent(L,x)
       // deletes x from L with sentinel

1  x.prev.next=x.next;
2  x.next.prev=x.prev;
```

Andere Operationen wie Einfügen und Löschen müssen auch angepasst werden



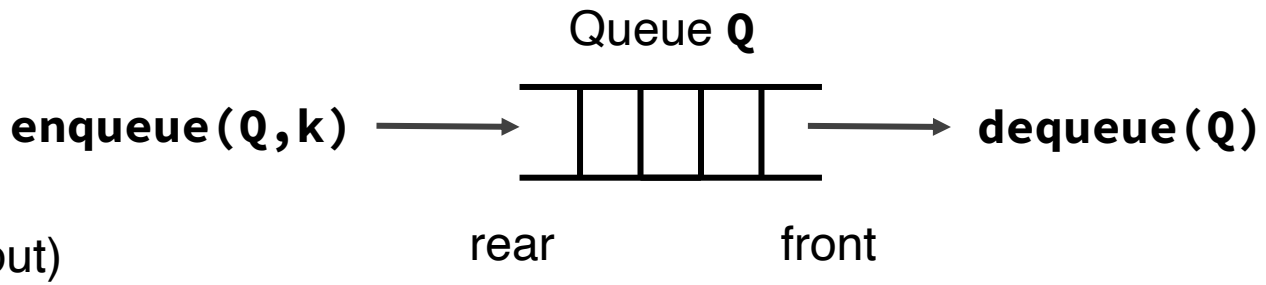
Delete(L,x)

1/2

Queues

Abstrakter Datentyp Queue

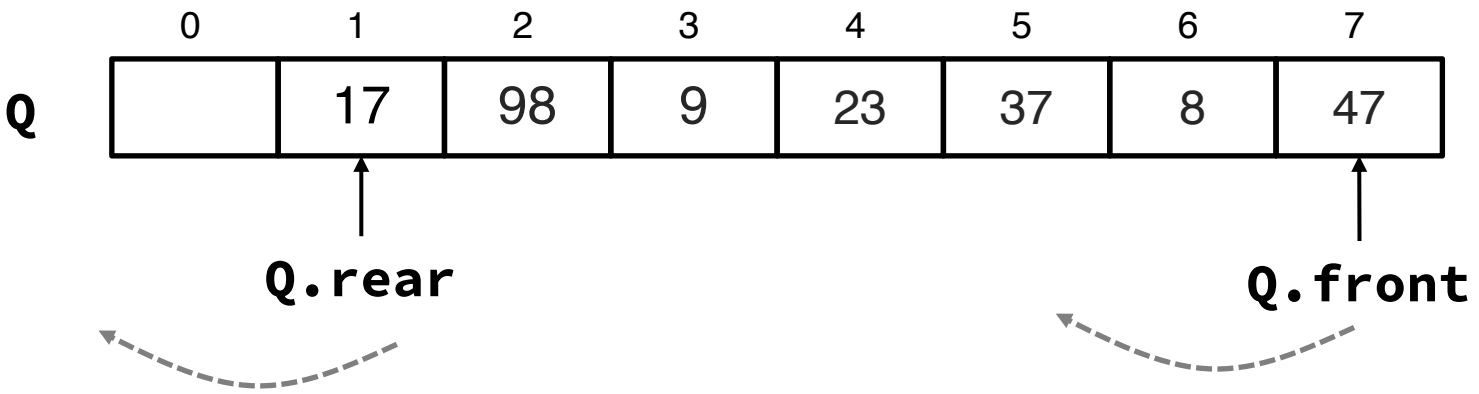
`new(Q)`	-	erzeugt neue (leere) Queue namens **`Q`**
`isEmpty(Q)`	-	gibt an, ob Queue **`Q`** leer
`dequeue(Q)`	-	gibt vorderstes Element der Queue **`Q`** zurück und löscht es aus Queue (bzw. Fehlermeldung, wenn Queue leer)
`enqueue(Q,k)`	-	schreibt **`k`** als neues hinterstes Element auf **`Q`** (bzw. Fehlermeldung, wenn Queue voll)



(FIFO – first in, first out)

Queues als Array? (I)

Wo ist **`front`**, wo **`rear`**?



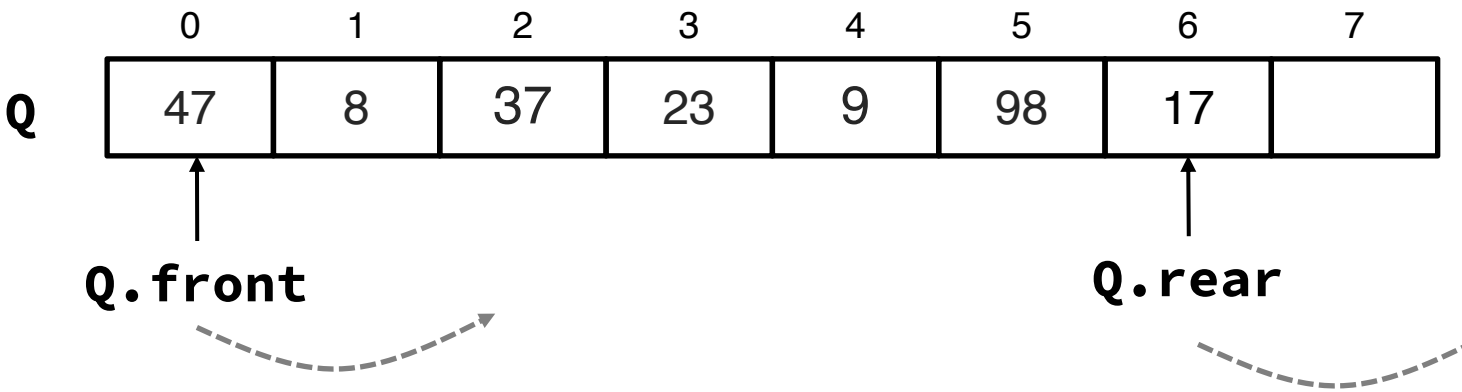
zu Beginn auch rechts,
wandert dann nach links
beim Einfügen

zu Beginn ganz rechts,
wandert dann nach links
beim Auswerfen

Problem:
Selbst wenn maximale Anzahl Elemente, die gleichzeitig in der Queue sind, vorher bekannt, kann **`Q.rear`** die Array-Grenze links erreichen

Queues als Array? (II)

Wo ist **`front`**, wo **`rear`**?



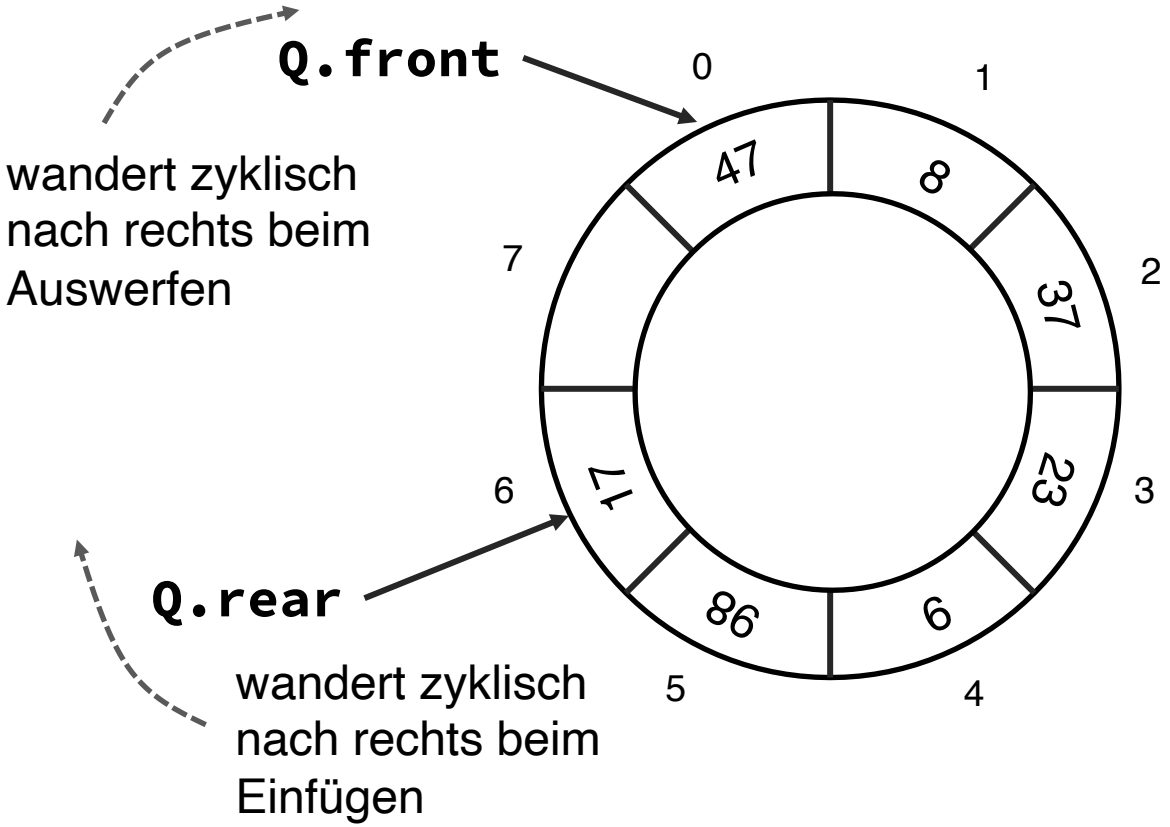
zu Beginn ganz links,
wandert dann nach rechts
beim Auswerfen

zu Beginn auch links,
wandert dann nach rechts
beim Einfügen

Problem:
Selbst wenn Array nach rechts unendlich lang,
wird Speicher links von **`Q.front`** verschwendet

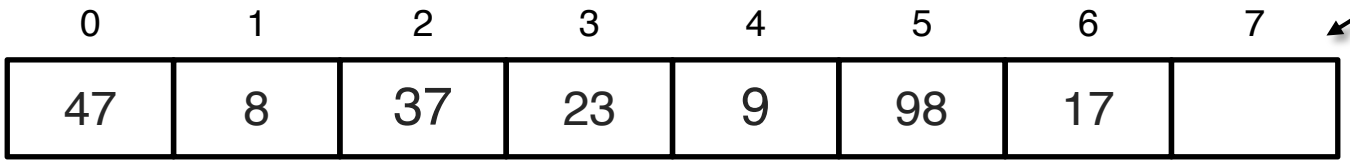
Queues als (virtuelles) zyklisches Array

MAX Elemente gleichzeitig in Queue



nur virtuell

Zeiger muss beim Wandern nach rechts von 7 auf 0 springen (und links von 0 auf 7)



real

Modulo-Operator

Modulo-Operator $x \; mod \; n$ für $n > 0$

Der Modulo-Operator $x \; mod \; n$ bildet eine ganze Zahl x auf die Zahl y zwischen 0 und $n - 1$ ab, so dass $y = x - i \cdot n$ für eine ganze Zahl i.

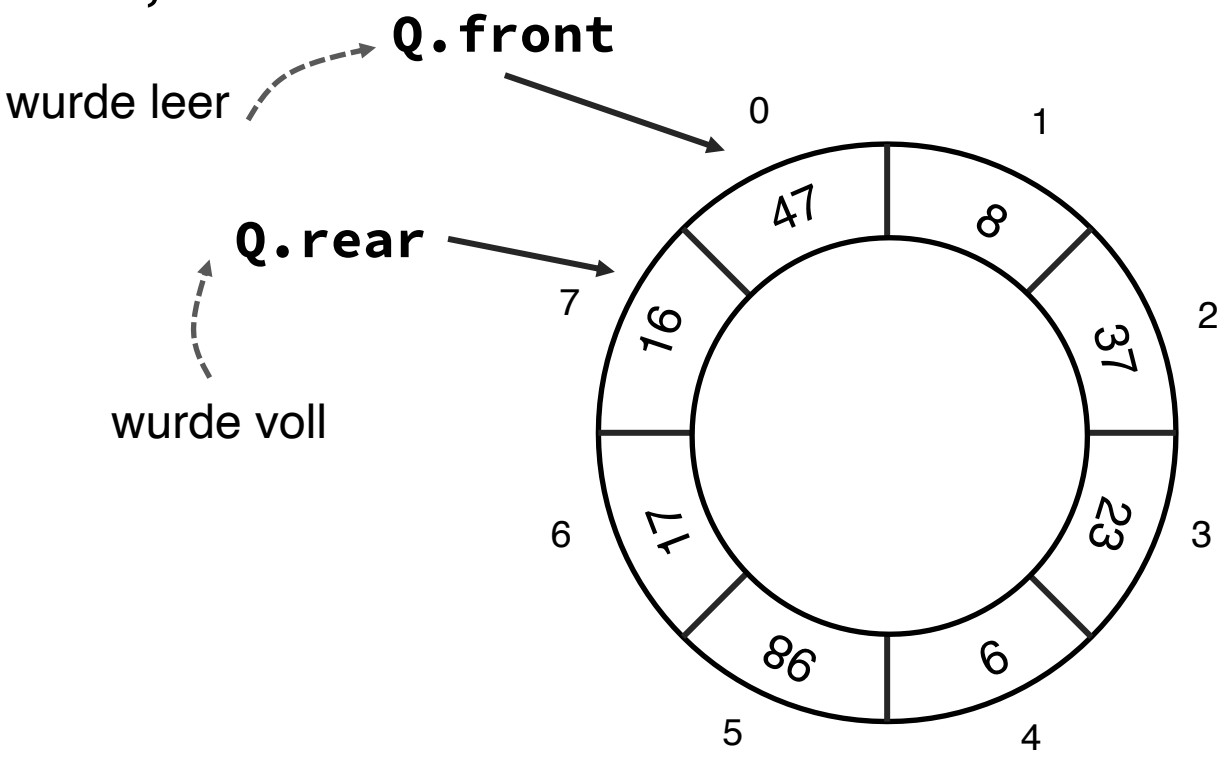
Beispiele:

$15 \bmod 5 = 0,$	weil $0 = 15 - 3 \cdot 5$
$(6 + 7) \; mod \; 8 = 5,$	weil $5 = 13 - 1 \cdot 8$
$-4 \; mod \; 7 = 3,$	weil $3 = -4 + 1 \cdot 7$

Achtung: In Java ist der %-Operator als Divisionsrest für negative Zahlen x anders definiert, dort ist z.B. $-4 \; \% \; 7 = -4$. Man kann dies unter Beachtung eventueller Überläufe abbilden durch $x \; mod \; n = ((x \; \% \; n) + n) \; \% \; n$.

Ein Bit, bitte

MAX Elemente
gleichzeitig in Queue



Ist das eine leere Schlange oder eine volle Schlange?

Speichere diese Information in Boolean **`empty`**
(alternativ: reserviere ein Element des Arrays als „Abstandshalter“)

Queues als zyklisches Array: Algorithmen

Q leer, wenn
front==rear+1 mod MAX
und **empty==true**

```
new(Q)

1  Q.A[]=ALLOCATE(MAX);
2  Q.front=0;
3  Q.rear=-1;
4  Q.empty=true;
```

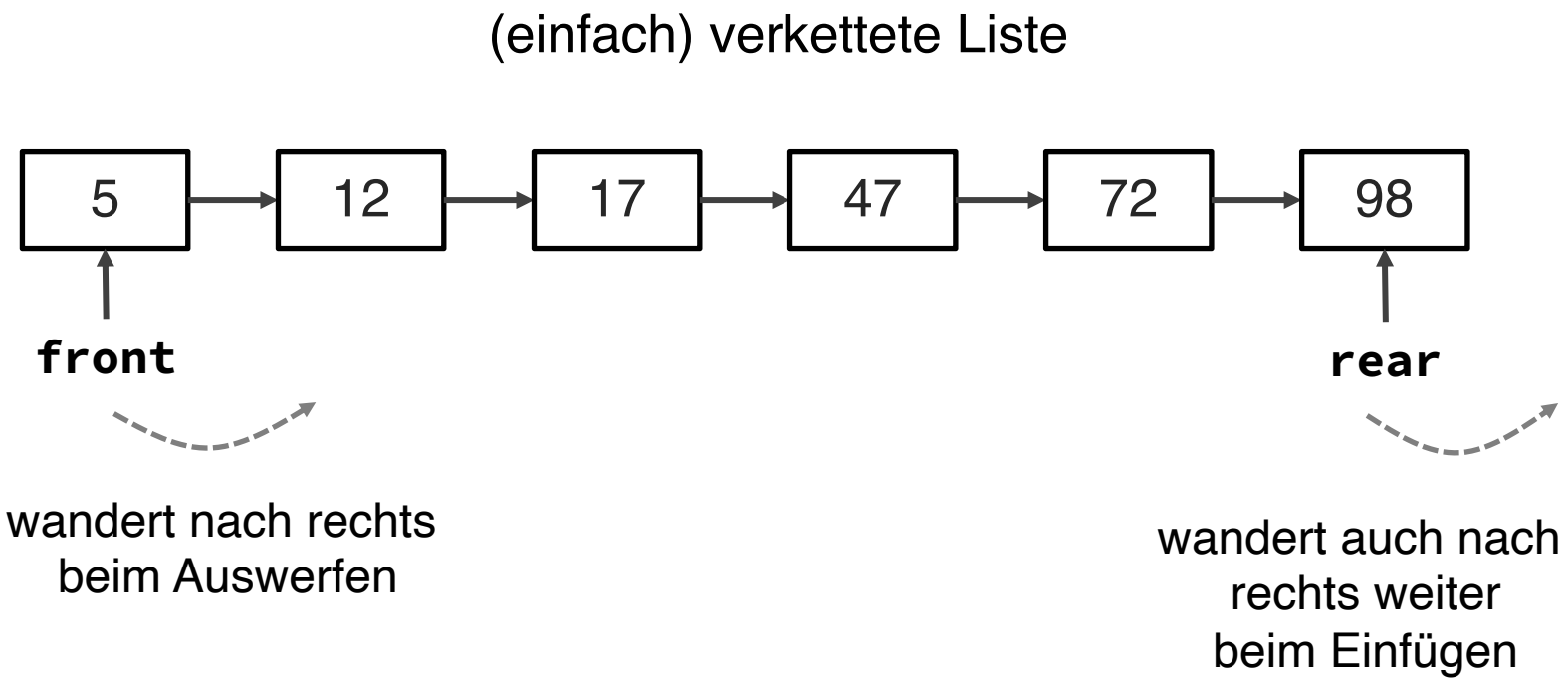
Q voll, wenn
front==rear+1 mod MAX
und **empty==false**

```
isEmpty(Q)

1  return Q.empty;
```

```
dequeue(Q)

1  IF isEmpty(Q) THEN
2    error 'underflow'
3  ELSE
4    Q.front=Q.front+1 mod MAX;
5    IF Q.front==Q.rear+1 mod MAX
6       THEN Q.empty=true;
7    return Q.A[Q.front-1 mod MAX];
```

```
enqueue(Q,k)

1  IF Q.front==Q.rear+1 mod MAX
                  AND !Q.empty THEN
2    error 'overflow'
3  ELSE
4    Q.rear=Q.rear+1 mod MAX;
5    Q.A[Q.rear]=k;
6    Q.empty=false;
```

Queues durch einfach (!) verkettete Listen



Queues durch Liste: Algorithmen

```
new(Q)

1  Q.front=nil;
2  Q.rear=nil;
```

```
isEmpty(Q)

1  IF Q.front==nil THEN
2     return true
3  ELSE
4     return false;
```

```
dequeue(Q)

1  IF isEmpty(Q) THEN
2     error 'underflow'
3  ELSE
4     x=Q.front;
5     Q.front=Q.front.next;
6     return x;
```

```
enqueue(Q,x)

1  IF isEmpty(Q) THEN
2      Q.front=x;
3  ELSE
4      Q.rear.next=x;
5  x.next=nil;
6  Q.rear=x;
```

Anzahl Operationen

*Worst Case

Stack

Operation	Laufzeit*
Push	$\Theta(1)$
Pop	$\Theta(1)$

Queue

Operation	Laufzeit*
Enqueue	$\Theta(1)$
Dequeue	$\Theta(1)$

Verkettete Liste

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(1)$
Suchen	$\Theta(n)$

Laufzeit Löschen eines Wertes $\Omega(n)$

Geben Sie die Suchoperation für einen Wert **k** bei einer Liste mit Sentinels an.

Wie kann man mit Hilfe zweier Stacks eine Queue implementieren?

Wie kann man mit Hilfe zweier Queues einen Stack implementieren?