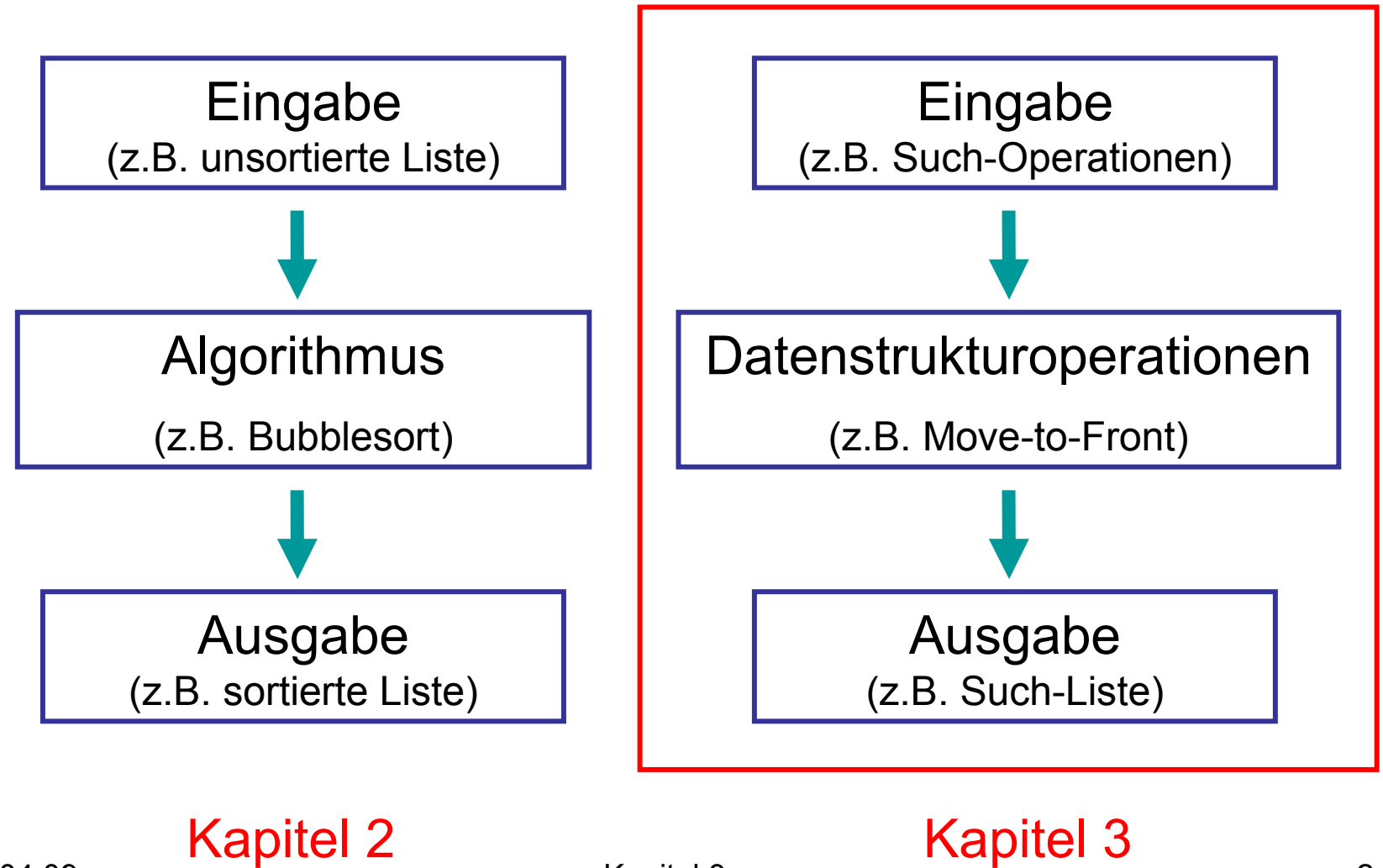
Grundlagen der Algorithmen und Datenstrukturen

Kapitel 2

Christian Scheideler + Helmut Seidl

SS 2009

Grundlegende Laufzeitanalyse



Übersicht

Thema: Repräsentation von Sequenzen als Felder und verkettete Listen

- Was ist eine Sequenz?
- Repräsentation als Feld und amortisierte Analyse
- Repräsentation als verkettete Liste
- Stapel (Stacks) und Schlangen (Queues)

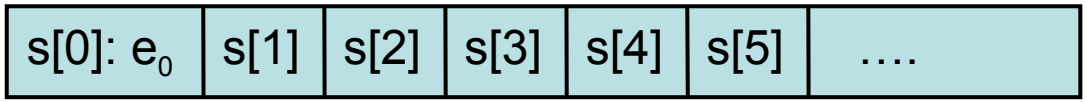
Sequenzen

Sequenz:

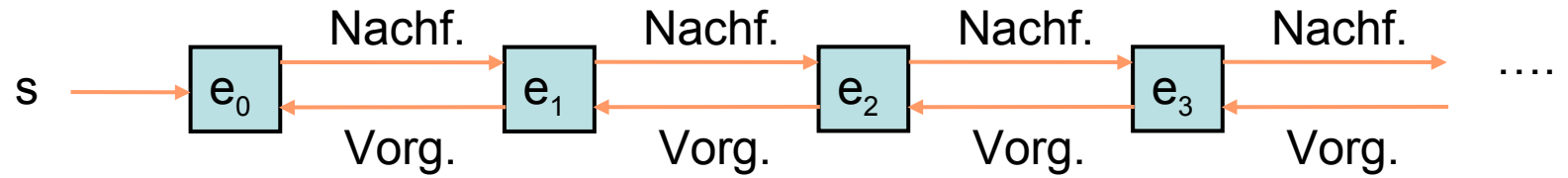
$$s = \langle e_0, \ldots, e_{n-1} \rangle$$

Arten, auf Element zuzugreifen:

- Feldrepräsentation: direkter Zugriff über s[i]



- Listenrepräsentation: indirekter Zugriff über Nachfolger und/oder Vorgänger (Schnitzeljagd durch Speicher)

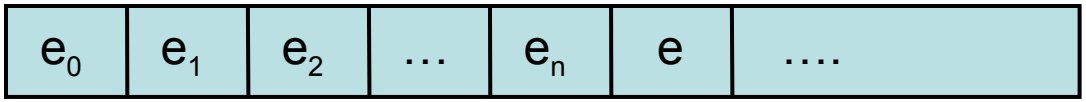


Sequenz als Feld

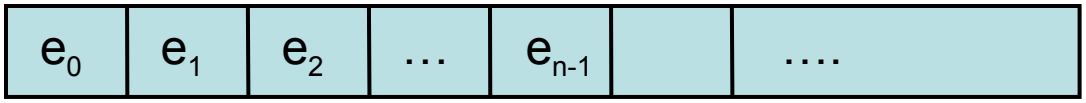
Operationen:

$\langle e_0,\ldots,e_n\rangle$.get(i) = e_i (analog set(i,e))

- $\langle e_0,\ldots,e_n\rangle$.pushBack(e) = $\langle e_0,\ldots,e_n,e\rangle$



- $\langle e_0,\ldots,e_n\rangle$.popBack() = $\langle e_0,\ldots,e_{n-1}\rangle$

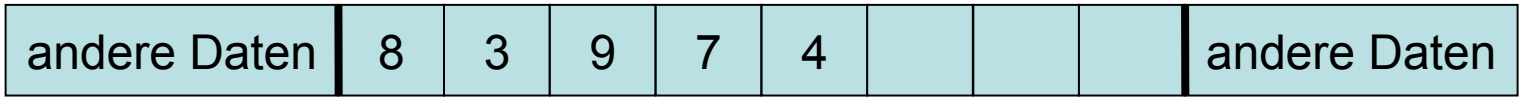


- ($\langle e_0,\ldots,e_{n-1}\rangle$).size() = n

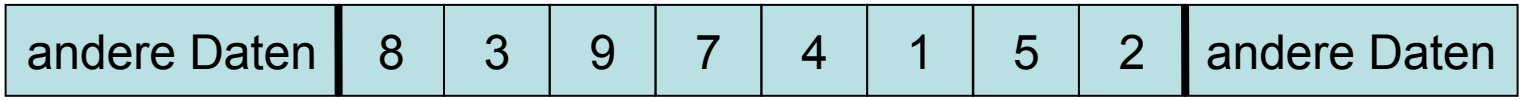
Sequenz als Feld

Problem: s sei Feld mit 8 Einträgen

s-Feld



- pushback(1), pushback(5), pushback(2)



- pushback(6): voll!!

Sequenz als Feld

Problem:

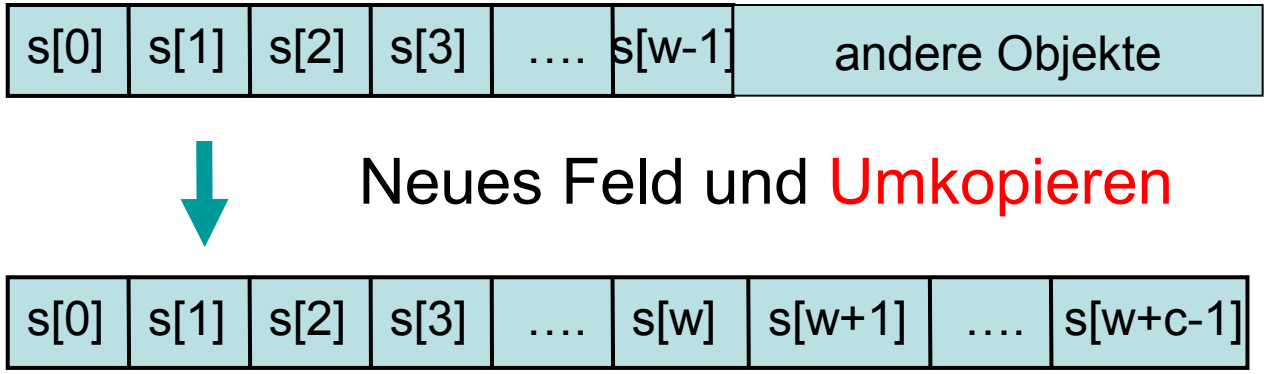
- Im vornherein nicht bekannt, wieviele Elemente das Feld enthalten wird
- Nur Anlegen von statischen Feldern möglich
  (s = new Elem [w])

Lösung: Datenstruktur für dynamisches Feld

3.1 Dynamisches Feld

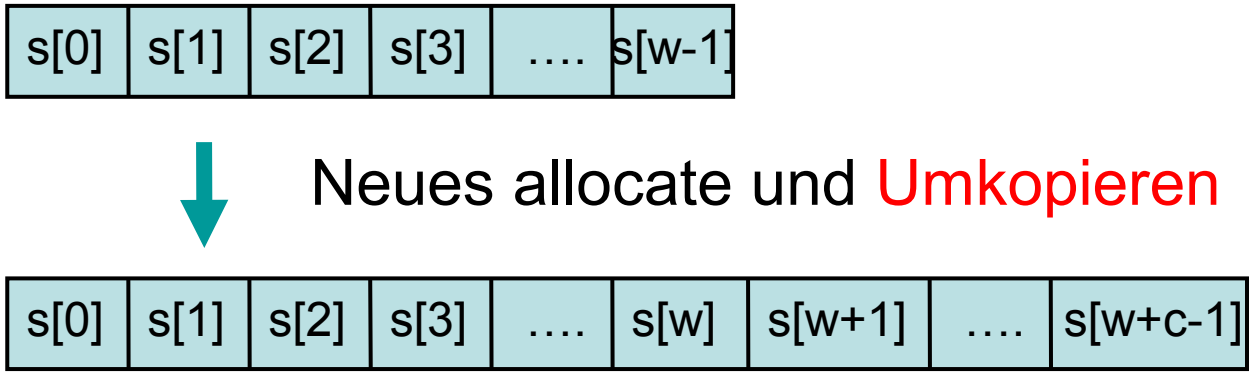
Erste Idee:

- Jedesmal, wenn Feld s nicht mehr ausreicht (n>w-1), generiere neues Feld der Größe w+c für ein festes c.



Dynamisches Feld

Zeitaufwand für Erweiterung ist O(w):



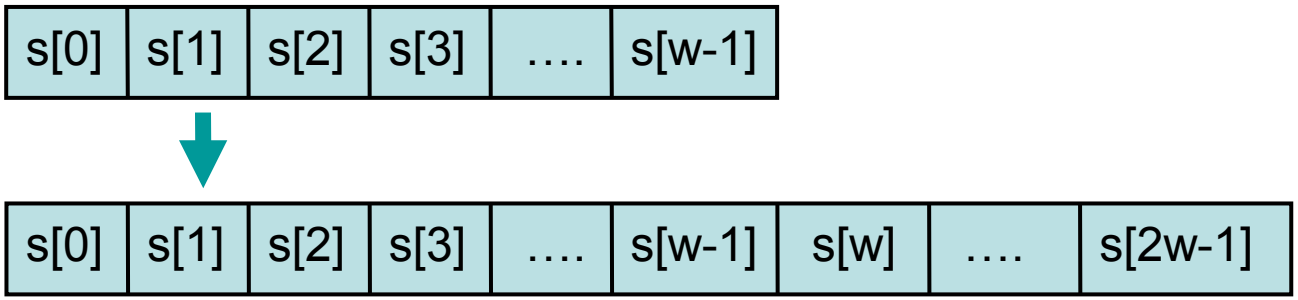
Zeitaufwand für n pushBack Operationen:

- Aufwand von O(w) je c Operationen
- Gesamtaufwand: $O(\sum_{i=1}^{n/c} c \cdot i) = O(n^2)$

Dynamisches Feld

Bessere Idee:

- Jedesmal, wenn Feld s nicht mehr ausreicht (n>w-1), generiere neues Feld der doppelten Größe 2w.



- Jedesmal, wenn Feld s zu groß ist (n<w/4), generiere neues Feld der halben Größe w/2.

Dynamisches Feld

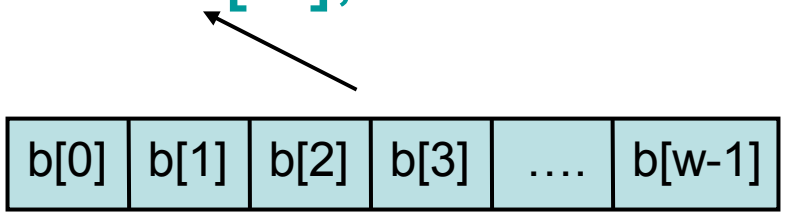
Implementierung als Klasse UArray mit

- Methode Element get (int i)
- Methode int size()
- Methode void pushBack(Elem e)
- Methode void popBack()
- Methode void increase(int w)

Dynamisches Feld

Variablen in Klasse UArray:

- $\beta = 2;$ Wachstumsfaktor
- $\alpha = 4;$ max. Speicheroverhead
- w=1; momentane Feldgröße
- n=0; momentane # Elemente
- b = new Elem [w];

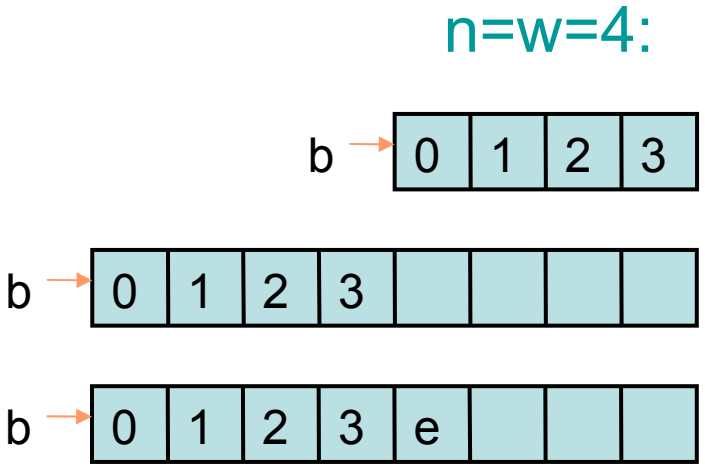


Dynamisches Feld

```
Elem get (int i) {
    assert (0<=i && i<n);
    return b[i];
}
int size() {
    return n;
}
```

Dynamisches Feld

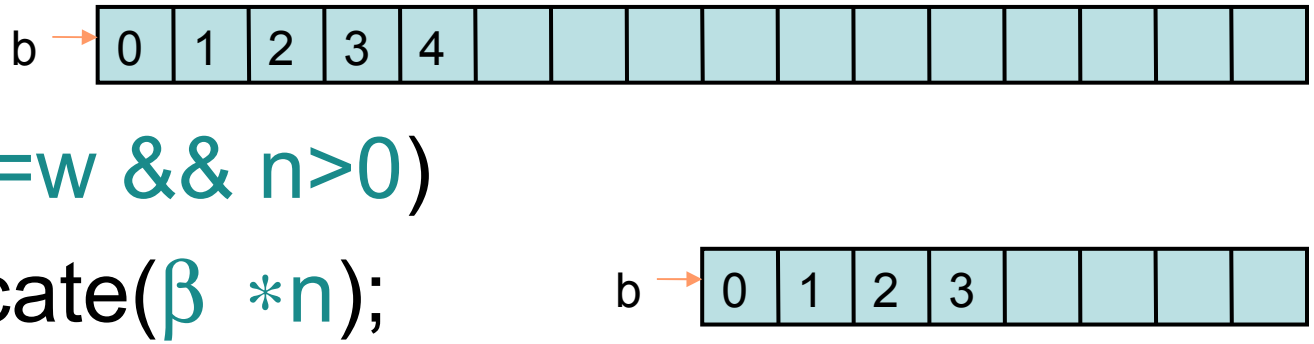
```
void pushBack(Elem e) {
    if (n==w)
        reallocate(β ∗n);
    b[n]=e;
    n=n+1;
}
```



Dynamisches Feld

```
void popBack() {
    assert (n>0);
    n=n-1;
    if (α ∗n<=w && n>0)
        reallocate(β ∗n);
}
```

n=5, w=16:



Dynamisches Feld

```
void reallocate (int w) {
    this.w=w;
    Elem [] b0 = new Elem [w];
    for (i=0; i<n; i++)
        b0[i]=b[i];
    b=b0;
}
```

} Umkopieren

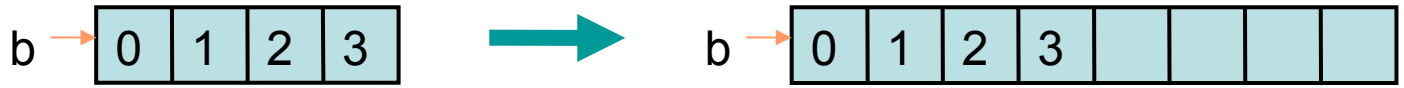
Dynamisches Feld

Lemma 3.1: Betrachte ein anfangs leeres dynamisches Feld s. Jede Folge $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ von pushBack und popBack Operationen kann auf s in Zeit O(n) bearbeitet werden.

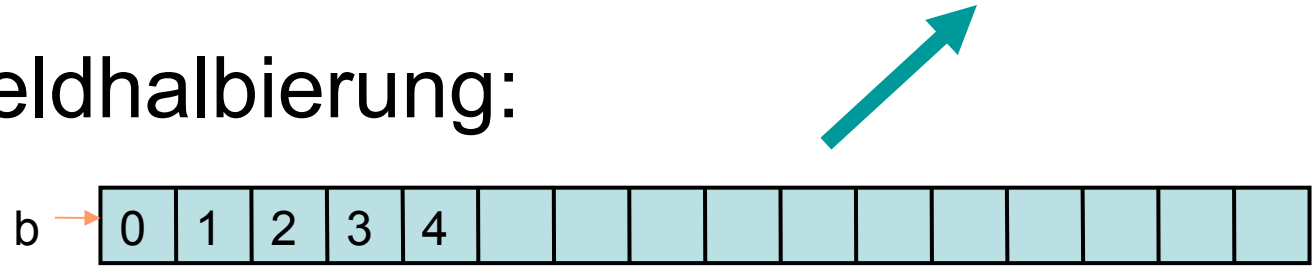
- Erste Idee: Laufzeit $O(n^2)$
- Nur durchschnittlich konstante Laufzeit pro Operation
  (Fachbegriff für „durchschnittlich“: amortisiert)

Dynamisches Feld - Analyse

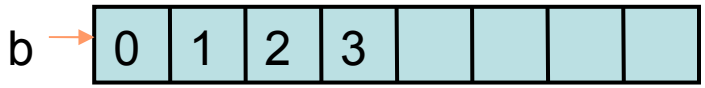
- Feldverdopplung:



- Feldhalbierung:



- Von



  - Nächste Verdopplung: ≥ n pushBack Ops
  - Nächste Halbierung: ≥ n/2 popBack Ops

Dynamisches Feld - Analyse

- Von 

  - Nächste Verdopplung: ≥ n pushBack Ops
  - Nächste Halbierung: ≥ n/2 popBack Ops

- Idee: verrechne reallocate-Kosten mit pushBack/popBack Kosten (ohne realloc)
  - Kosten für pushBack/popBack: O(1)
  - Kosten für reallocate($\beta * n$): O(n)

Dynamisches Feld - Analyse

Idee:

verrechne reallocate-Kosten mit pushBack / popBack Kosten

Kontenmethode:

Günstige Operationen zahlen Tokens ein.

Teure Operationen entnehmen Tokens.

Tokenkonto darf nie negativ werden!

Dynamisches Feld - Analyse

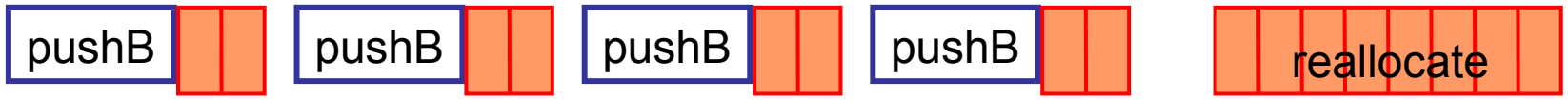
Kontenmethode:

- Günstige Operationen zahlen Tokens ein
  $\longrightarrow$ pro pushBack 2 Tokens
  $\longrightarrow$ pro popBack 1 Token
- Teure Operationen entnehmen Tokens
  $\longrightarrow$ pro reallocate($\beta * n$) $-n$ Tokens
- Tokenkonto darf nie negativ werden!
  $\longrightarrow$ erfüllt über Zeugenargument

Dynamisches Feld - Analyse

Tokenlaufzeit:

- Ausführung von push/popBack kostet 1 Token
  $\rightarrow$ Tokenkosten für pushBack: 1+2 = 3
  $\rightarrow$ Tokenkosten für popBack: 1+1 = 2
- Ausführung von reallocate($\beta * n$) kostet n Tokens
  $\rightarrow$ Tokenkosten für reallocate($\beta * n$): n-n=0



Gesamtlaufzeit = O(Summe der Tokenlaufzeiten)

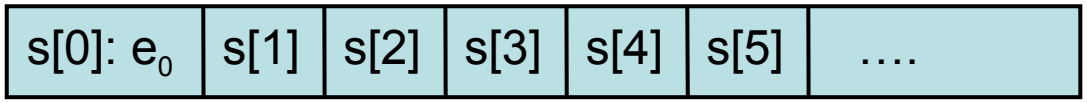
Sequenzen

Sequenz:

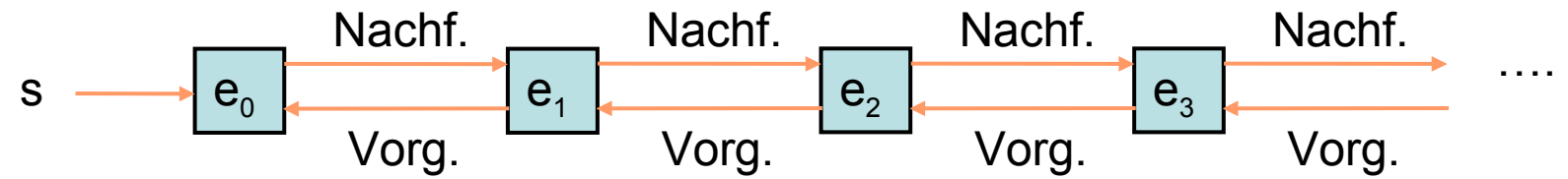
$s = \langle e_0, \ldots, e_{n-1} \rangle$

Arten, auf Element zuzugreifen:

- Feldrepräsentation: direkter Zugriff über s[i]

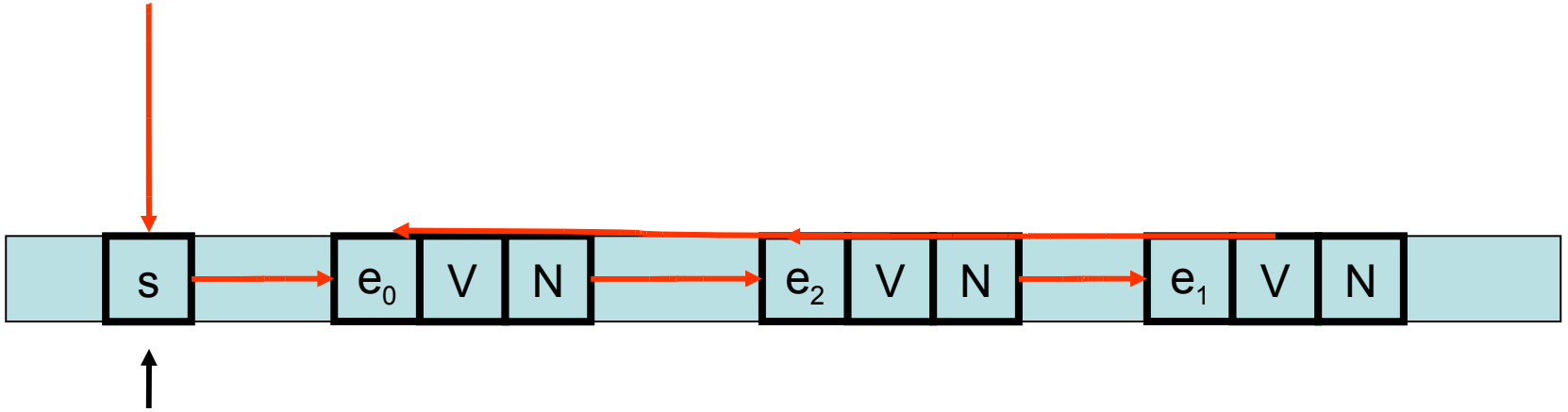


- Listenrepräsentation: indirekter Zugriff über Nachfolger und/oder Vorgänger



Doppelt verkettete Liste

Interne Speicherung (3 Elemente):

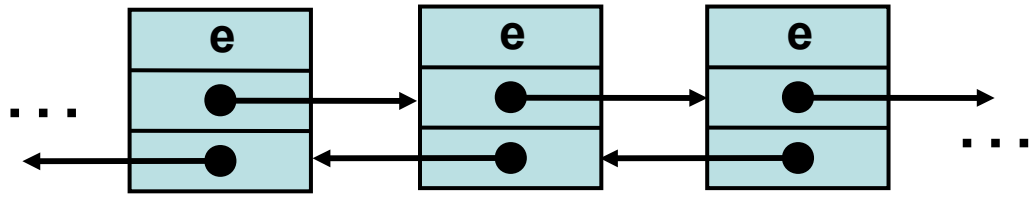


Variable s speichert Startpunkt der Liste

Wie Schnitzeljagd im Speicher.

Doppelt verkettete Liste

```
class Item<Elem> {
    Elem e;
    Item<Elem> next;
    Item<Elem> prev;
}
```



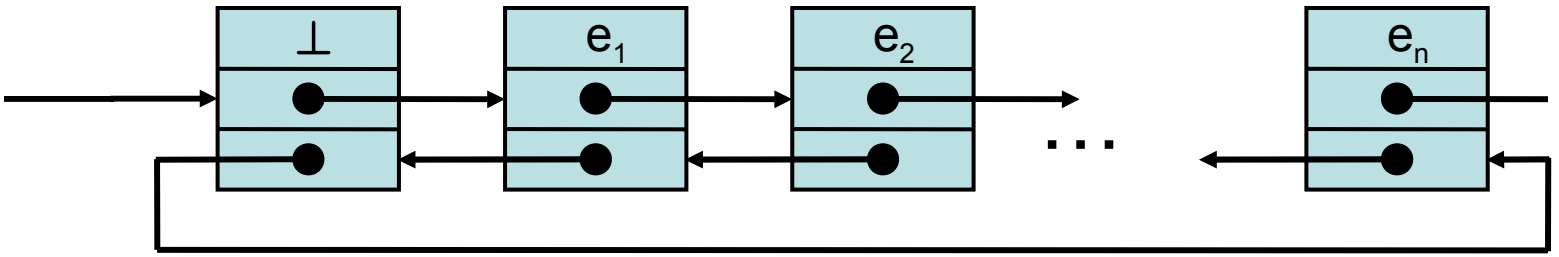

```
class List<Elem> {
    Item<Elem> h;
    …weitere Variablen und Methoden…
```

Invariante: (this: Zeiger auf aktuelles Element)
next.prev == prev.next == this

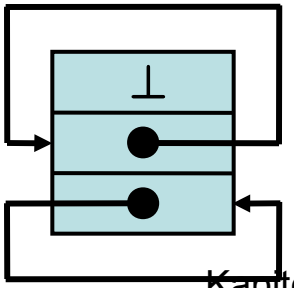
Doppelt verkettete Liste

Einfache Verwaltung:

durch „Dummy“-Element mit Inhalt null:



Anfangs:



Doppelt verkettete Liste

Zentrale statische Methode: splice

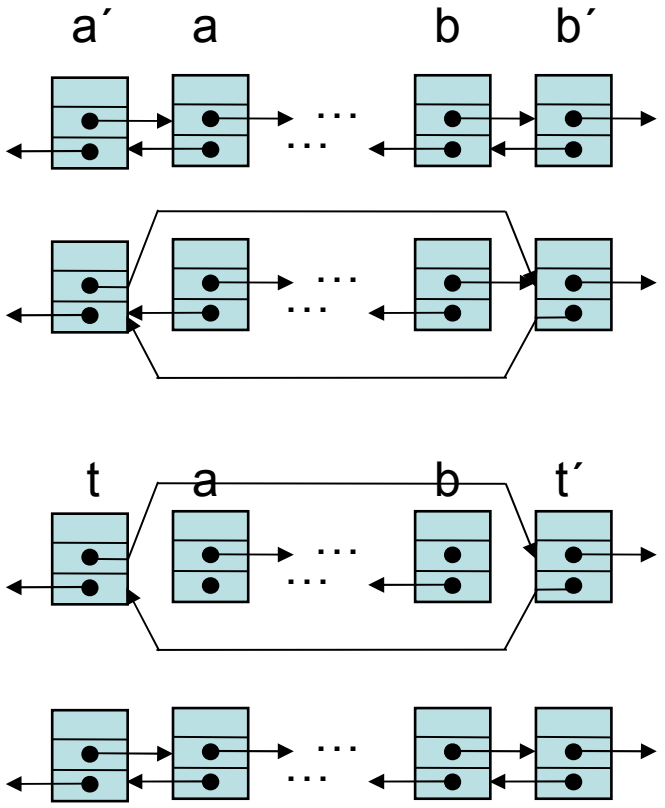
- splice entfernt <a,…,b> aus Sequenz und fügt sie hinter einem t an.
- Bedingung: <a,…,b> muss eine Teilsequenz sein, b ist nicht vor a, und t darf nicht in <a,…,b> stehen.

Für $<e_1,\ldots,a',a,\ldots,b,b',\ldots,t,t'\ldots,e_n>$ liefert

splice (a, b, t) = $<e_1,\ldots,a',b',\ldots,t,a,\ldots,b,t',\ldots,e_n>$

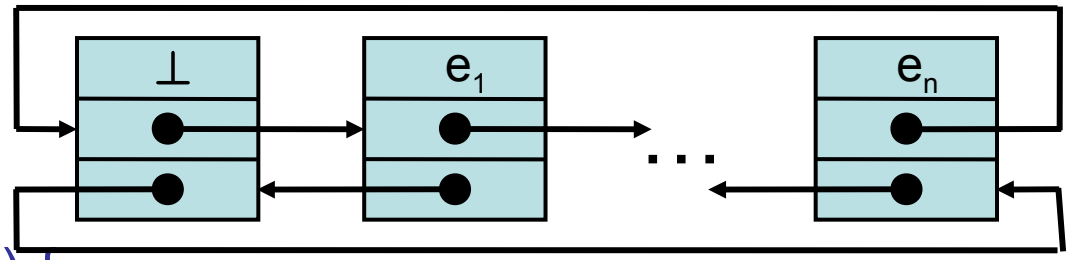
Doppelt verkettete Liste

```
static void splice(Item<Elem> a, Item<Elem> b,Item<Elem> t) {

// schneide <a,…,b> heraus
  Item<Elem> a' = a.prev;
  Item<Elem> b' = b.next;
  a'.next = b´;
  b'.prev = a';

// füge <a,…,b> hinter t ein
  Item<Elem> t' = t.next;
  b.next = t';
  a.prev = t;
  t.next = a;
  t'.prev = b;
}
```



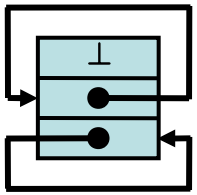
Doppelt verkettete Liste

h: Item mit null



Methoden:

```
Item<Elem> head() {
    return h;
}
boolean isEmpty() {
    return h.next == h;
}
Item<Elem> first() {
    return h.next;
}
Item<Elem> last() {
    return h.prev;
}
```

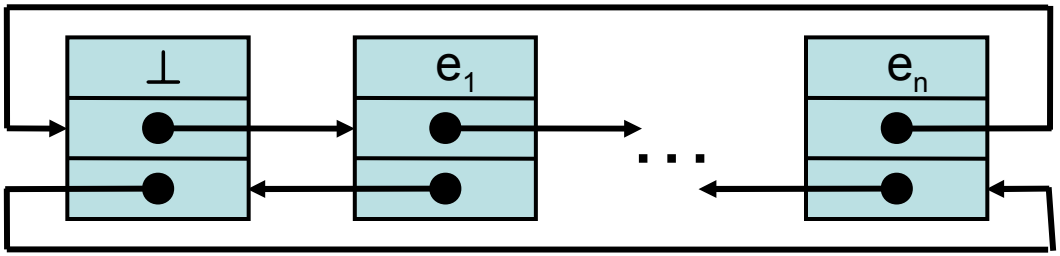


Kann evtl. auf ⊥-Element zeigen!

Kann evtl. auf ⊥-Element zeigen!

Doppelt verkettete Liste

h: Item mit null



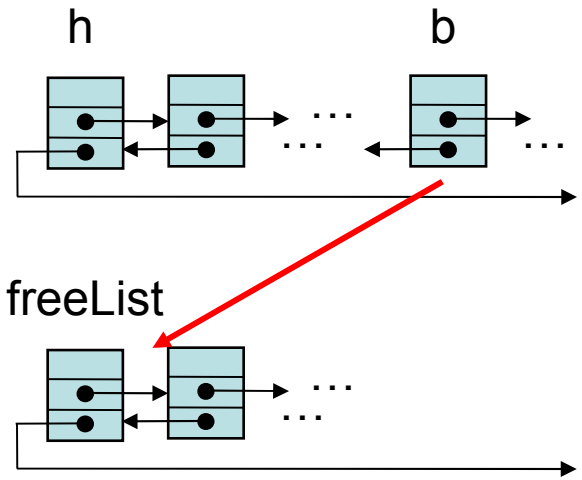
Methoden:

```
static void moveAfter(Item<Elem> b, Item<Elem> a) {
    splice(b,b,a);
}                                          // schiebe b nach a
void moveToFront(Item<Elem> b) {
    moveAfter(b,head());
}                                          // schiebe b nach vorn
                                           // analog moveToEnd
```

Doppelt verkettete Liste

Löschen und Einfügen von Elementen:
mittels extra Liste freeList

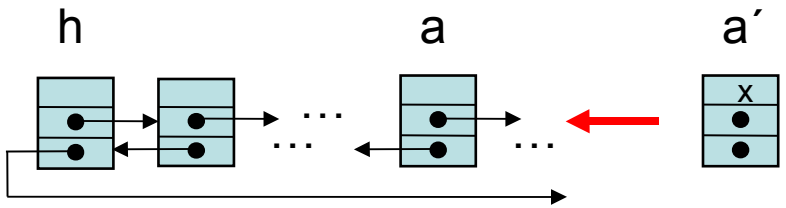
```
static void remove(Item<Elem b) {
   moveAfter(b, freeList.head());
}
void popFront() {
   remove(first());
}
void popBack() {
   remove(last());
}
```



freeList: gut für Laufzeit, da Speicherallokation teuer

Doppelt verkettete Liste

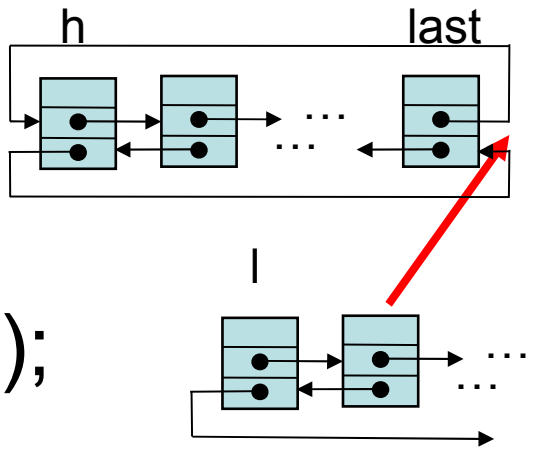
```
static Item<Elem> insertAfter(Elem x, Item<Elem> a){
    Item<Elem> a';
    if (freeList.isEmpty())  a' = new Item<Elem>();
    else a' = freeList.first();
    moveAfter(a', a);
    a'.e = x;
    return a';
}
static Item<Elem> insertBefore(Elem e, Item<Elem> b) {
    return insertAfter(e, b.prev);
}
void pushFront(Elem e) {
insertAfter(e, head());
}
void pushBack(Elem e)  { insertAfter(e. last()): }
```



Doppelt verkettete Liste

Manipulation ganzer Listen:

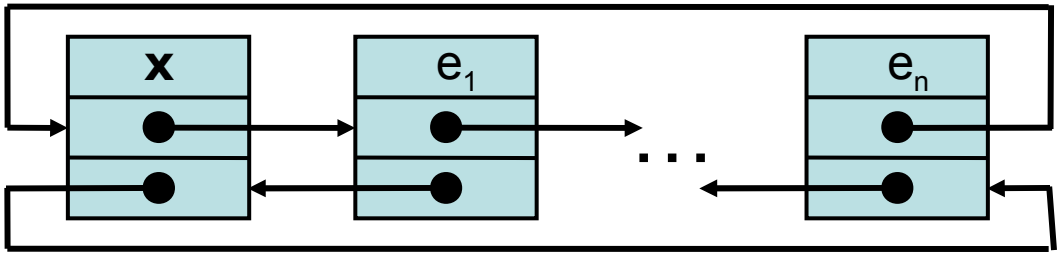
```
void concat(List<Elem> l) {
   if (! l.isEmpty())
     splice(l.first(), l.last(), last());
}
```



Doppelt verkettete Liste

Suche nach einem Element:

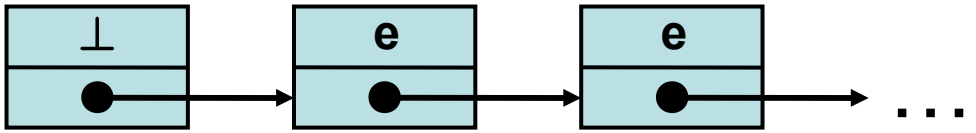
Trick: verwende „Dummy“-Element




```
Item<Elem> findNext (Elem x, Item<Elem> from) {
    h.e = x;
    while (from.e != x)
        from = from.next;
    h.e = null; return from;
}
```

Einfach verkettete Liste

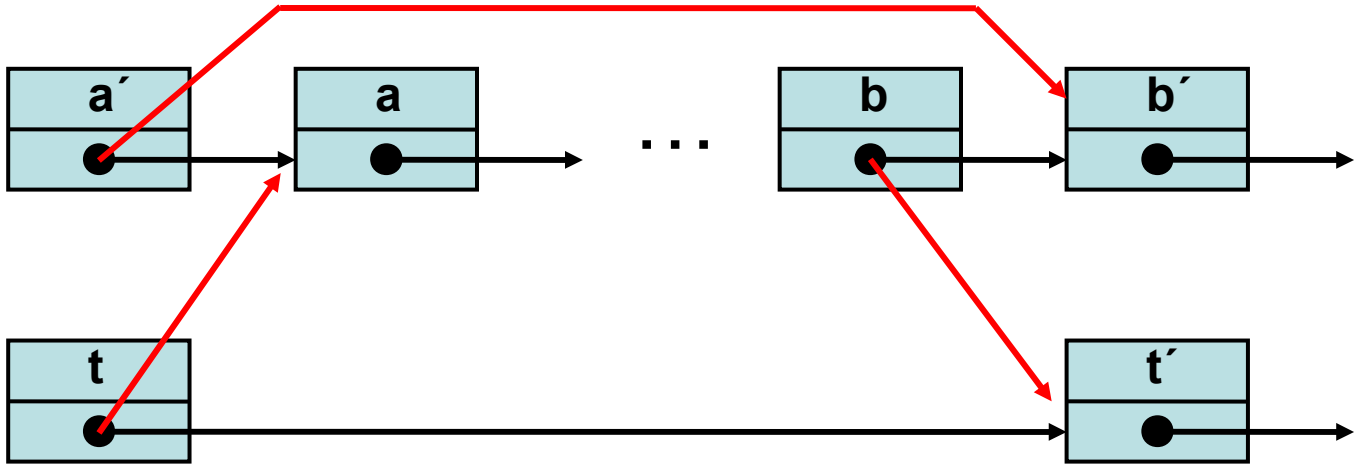
```
class Sitem <Elem> {
    Elem e;
    Sitem<Elem> next;
}
```




```
class Slist<Elem> {
    Sitem<Elem> h;
    …weitere Variablen und Methoden…
}
```

Einfach verkettete Liste

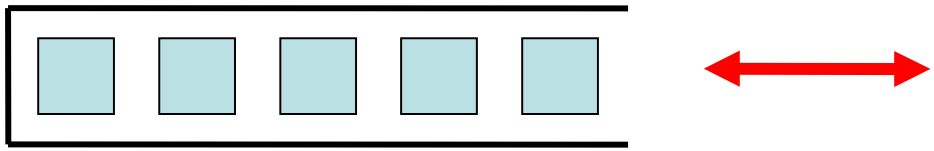
```
static void splice(Sitem a', Sitem b, Sitem t) {
    Sitem a = a'.next;
    a'.next = b.next;
    b.next = t.next;
    t.next = a;
}
```



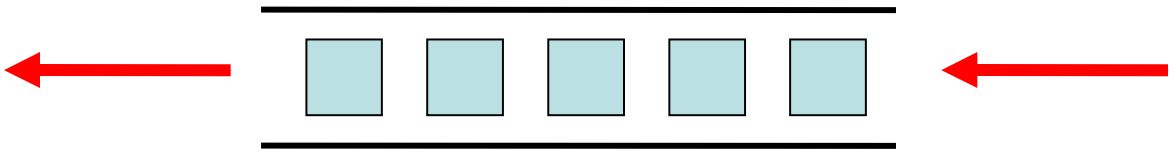
Stacks und Queues

Grundlegende Datenstrukturen für Sequenzen:

- Stack



- FIFO-Queue:



Stack

Methoden:

- pushBack:
  $\langle e_0,\ldots,e_n\rangle$.pushBack(e) = $\langle e_0,\ldots,e_n,e\rangle$
- popBack:
  $\langle e_0,\ldots,e_n\rangle$.popBack() = $\langle e_0,\ldots,e_{n-1}\rangle$
- last:
  $\langle e_0,\ldots,e_n\rangle$.last() = e_n

Implementierungen auf vorherigen Folien.

FIFO-Queue

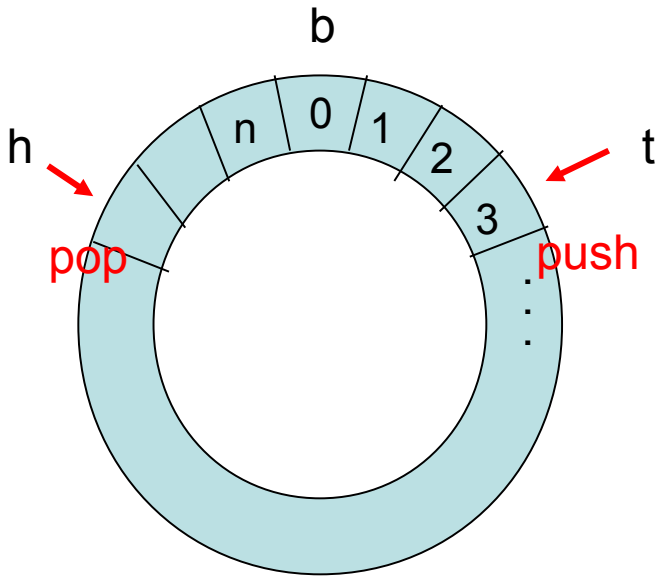
Methoden:

- pushBack:
  $\langle e_0,\ldots,e_n\rangle$.pushBack($e$) = $\langle e_0,\ldots,e_n,e\rangle$
- popFront:
  $\langle e_0,\ldots,e_n\rangle$.popFront() = $\langle e_1,\ldots,e_n\rangle$
- first:
  $\langle e_0,\ldots,e_n\rangle$.first() = e_0

Implementierungen auf vorherigen Folien

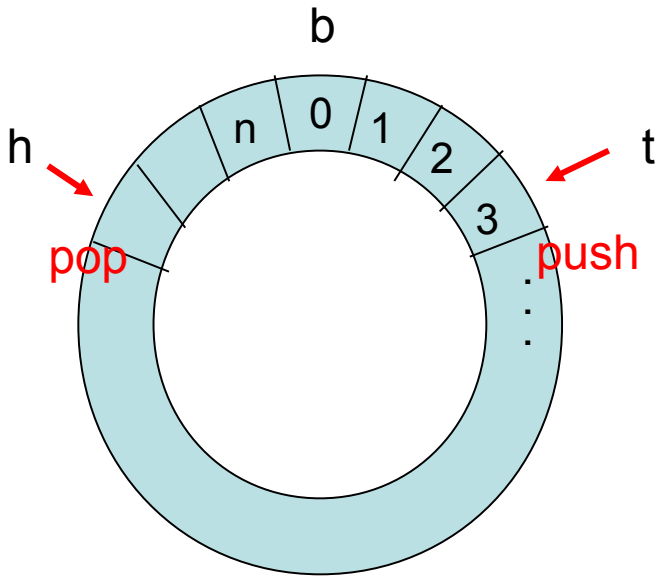
Beschränkte FIFO-Queue

```
class BoundedFIFO <Elem> {
    Elem[] b;
    int h=0;     // Index des ersten Elements
    int t=0;     // Index des ersten freien Eintrags
}
boolean isEmpty() {
    return h == t;
}
Elem first() {
    return b[h];
}
int size() {
    return (t-h+n+1) % (n+1);
}
```



Beschränkte FIFO-Queue

```
class BoundedFIFO<Elem> {
    Elem[] b;
    int h=0;      // Index des ersten Elements
    int t=0;      // Index des ersten freien Eintrags
...
void pushBack(Elem x) {
    assert (size() < n);
    b[t] = x;
    t = (t+1) % (n+1);
}
void popFront() {
    assert (!isEmpty());
    h = (h+1) % (n+1);
}
```



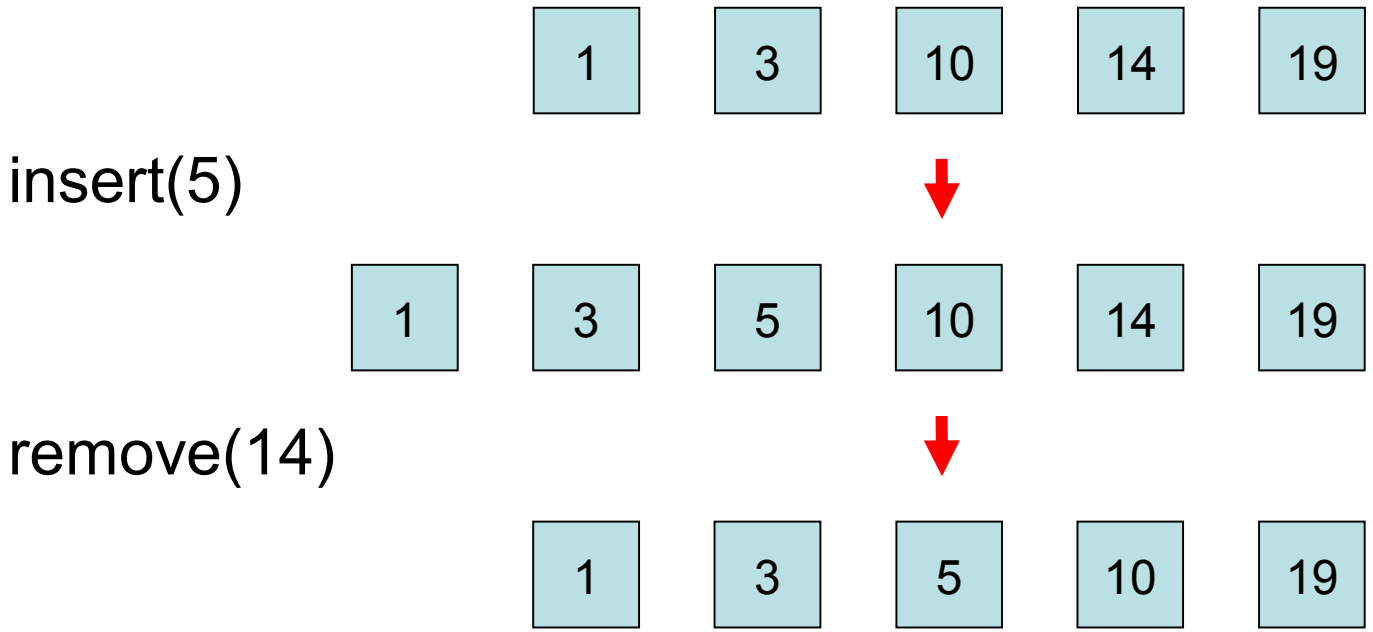
Fazit

- Listen sind sehr flexibel, wenn es darum geht, Elemente in der Mitte einzufügen
- Felder können in konstanter Zeit auf jedes Element zugreifen
- Listen haben kein Reallokationsproblem bei unbeschränkten Größen

$\longrightarrow$ beide Datenstrukturen einfach, aber oft nicht wirklich zufriedenstellend

Beispiel: Sortierte Sequenz

Problem: bewahre nach jeder Einfügung und Löschung eine sortierte Sequenz

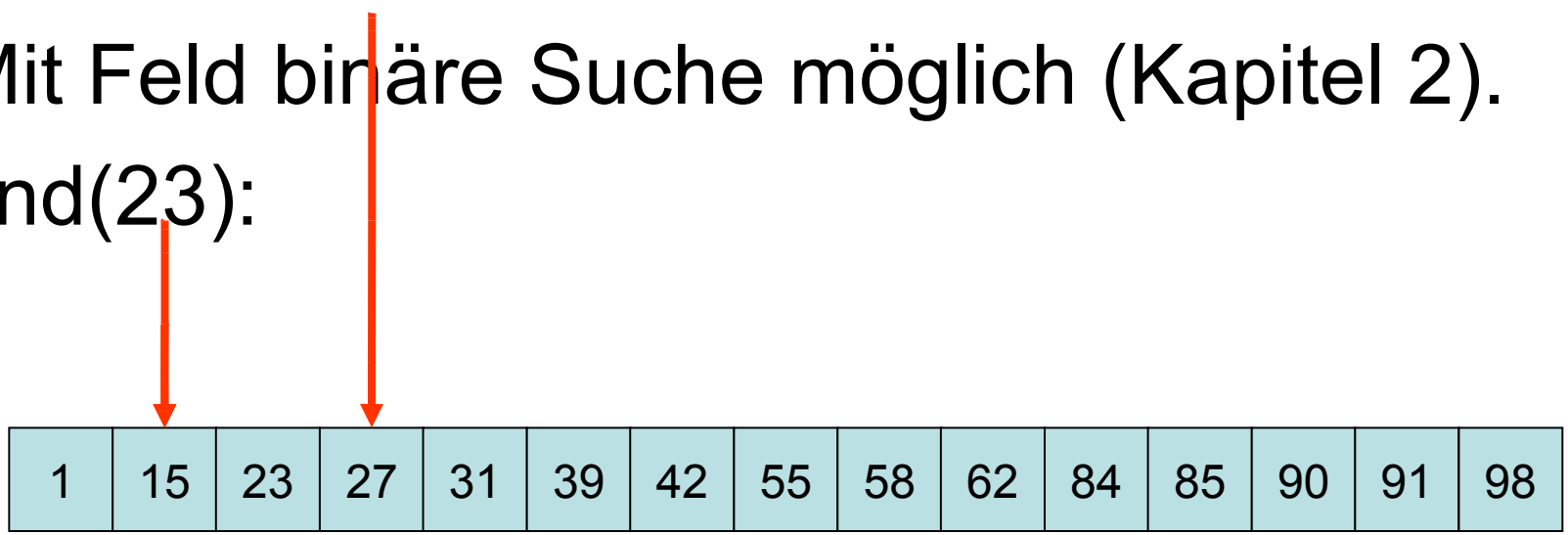


Beispiel: Sortierte Sequenz

Warum sortierte Sequenz?

Mit Feld binäre Suche möglich (Kapitel 2).

find(23):



In n-elementiger Folge kann jedes Element in max. log n Schritten gefunden.

Beispiel: Sortierte Sequenz

S: sortierte Sequenz

Jedes Element e identifiziert über key(e).

Operationen:

- $\langle e_0,\ldots,e_n\rangle$.insert(e) = $\langle e_0,\ldots,e_i,e,e_{i+1},\ldots,e_n\rangle$ für das i mit $key(e_i) < key(e) < key(e_{i+1})$
- $\langle e_0,\ldots,e_n\rangle$.remove(k) = $\langle e_0,\ldots,e_{i-1},e_{i+1},\ldots,e_n\rangle$ für das i mit $key(e_i)=k$
- $\langle e_0,\ldots,e_n\rangle$.find(k) = e_i für das i mit $key(e_i)=k$

Beispiel: Sortierte Sequenz

- Realisierung als Liste:
  - insert, remove und find auf Sequenz der Länge n kosten im worst case $\Theta(n)$ Zeit
- Realisierung als Feld:
  - insert und remove kosten im worst case $\Theta(n)$ Zeit
  - find kann so realisiert werden, dass es im worst case nur $O(\log n)$ Zeit benötigt ($\rightarrow$ binäre Suche!)

Beispiel: Sortierte Sequenz

Kann man insert und remove besser mit einem Feld realisieren?

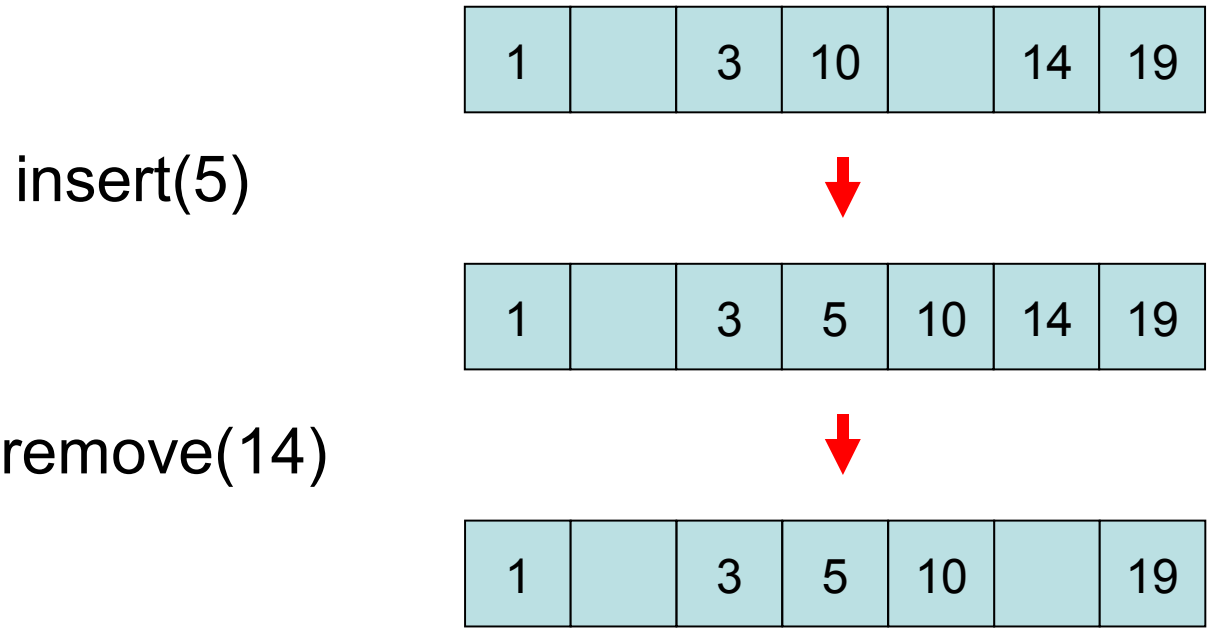
- folge Beispiel der Bibliothek!

- verwende Hashtabellen (Kapitel 4)

Beispiel: Sortierte Sequenz

Bibliotheksprinzip: lass Lücken!

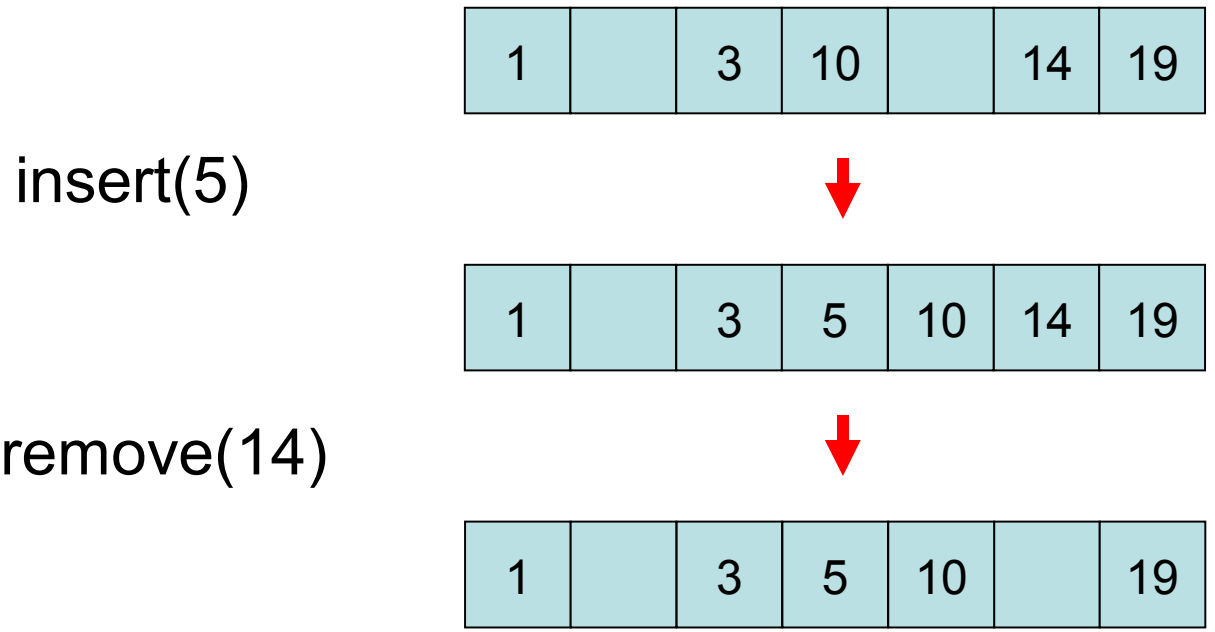
Angewandt auf sortiertes Feld:



Beispiel: Sortierte Sequenz

Durch geschickte Verteilung der Lücken:

amortierte Kosten für insert und remove $\Theta(\log^2 n)$



Beispiel: Sortierte Sequenz

Insert, delete und find noch viel effizienter umsetzbar, wie wir sehen werden!

Nächstes Kapitel

- Wörterbuchproblem
- Hashing
- Statische Wörterbücher
- Dynamische Wörterbücher