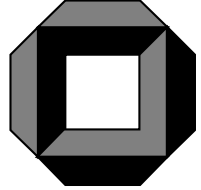
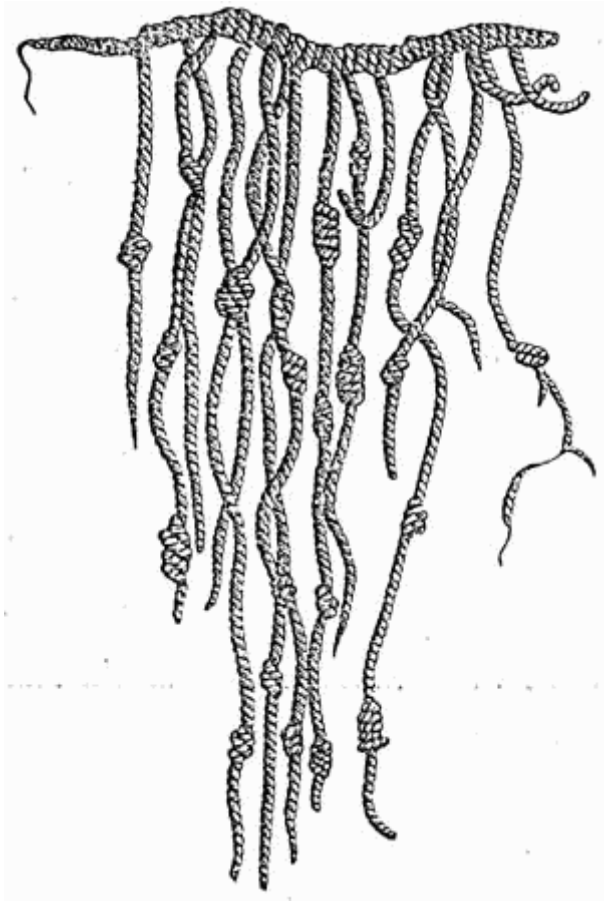
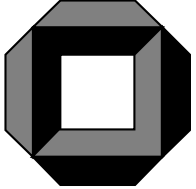




# 2 Folgen als Felder und Listen



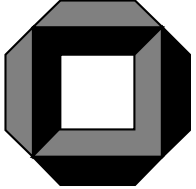




# Beschränkte Felder (Bounded Arrays)

Eingebaute Datenstruktur.

Größe muss von Anfang an bekannt sein





# Unbeschränke Felder

z.B. `std::vector`

pushBack: Element anhängen

popBack: Letztes Element löschen

Idee: verdopple wenn der Platz ausgeht.

Hhalbiere wenn Platz verschwendet wird

Wenn man das richtig macht, brauchen

$n$ pushBack/popBack Operationen Zeit $O(n)$

Algorithmese: pushBack/popBack haben konstante amortisierte Komplexität

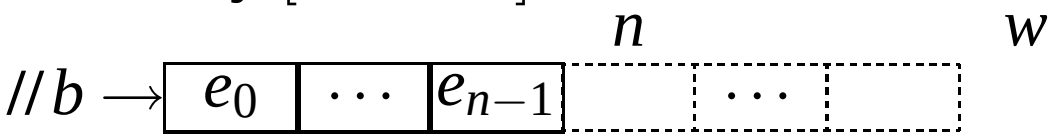
**Class** UArray **of** Element

$w$=1 : $\mathbb{N}$ // allocated size

$n$=0 : $\mathbb{N}$ // current size.

**invariant** $n \leq w < \alpha n$ or $n = 0$ and $w \leq 2$

$b$ : **Array** $[0..w-1]$ **of** Element



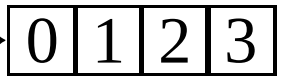

**Operator** $[i : \mathbb{N}]$ : Element

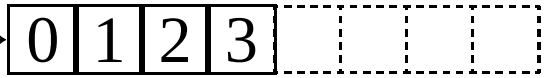
**assert** $0 \leq i < n$

**return** $b[i]$

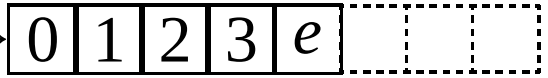
**Function** size : $\mathbb{N}$ **return** $n$

**Procedure** pushBack$(e$ : Element$)$ // Example for $n = w = 4$:

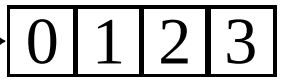
**if** $n = w$ **then** // $b \rightarrow$ 

reallocate$(2n)$ // $b \rightarrow$ 

$b[n] := e$ // $b \rightarrow$ 

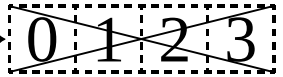
$n{+}{+}$ // $b \rightarrow$ 

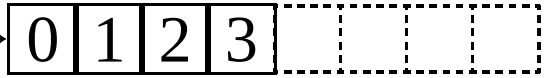
**Procedure** reallocate$(w'$ : $\mathbb{N})$ // Example for $w = 4$, $w' = 8$:

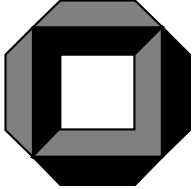
$w := w'$ // $b \rightarrow$ 

$b' :=$ **allocate Array** $[0..w'-1]$ **of** Element // $b' \rightarrow$ 

$(b'[0],\ldots,b'[n-1]) := (b[0],\ldots,b[n-1])$ // $b' \rightarrow$ 

**dispose** $b$ // $b \rightarrow$ 

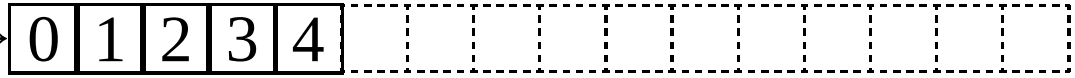
$b := b'$ // pointer assignment $b \rightarrow$ 

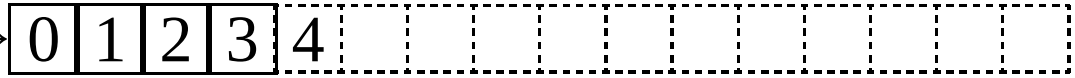




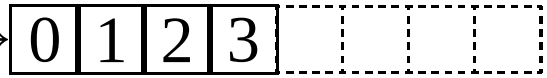
# Kürzen

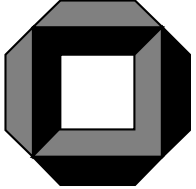
**Procedure** popBack // Example for $n = 5$, $w = 16$:

**assert** $n > 0$ // $b \rightarrow$ 


$n{--}$ // $b \rightarrow$ 


**if** $4n \leq w \wedge n > 0$ **then** // reduce waste of space

reallocate$(2n)$ // $b \rightarrow$ 





# Amortisierte Komplexität unbeschr. Felder

Sei $u$ ein anfangs leeres, unbeschränktes Feld.

Jede Operationenfolge $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$

von pushBack oder popBack Operationen auf $u$

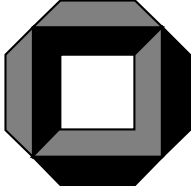
wird in Zeit $O(m)$ ausgeführt.

Sprechweise:

pushBack und popBack haben amortisiert konstante Ausführungszeit

—

$$O\left(\overbrace{m}^{\#\text{Ops}} / \underbrace{m}_{\text{Gesamtzeit}}\right) = O(1) \quad .$$



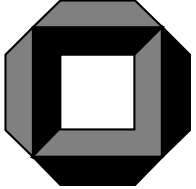


# Beweis: Konto-Methode (oder Versicherung)

| Operation | Kosten | | Typ |
|---|---|---|---|
| pushBack | ○○ | (2 Token) | einzahlen |
| popBack | ○ | (1 Token) | einzahlen |
| reallocate$(2n)$ | $n \times$ ○ | ($n$ Token) | abheben |

Zu zeigen: keine Überziehungen

Erster Aufruf von reallocate: kein Problem





# Beweis: Konto-Methode (oder Versicherung)

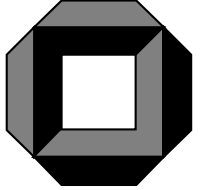
| Operation | Kosten | | Typ |
|---|---|---|---|
| pushBack | ○○ | (2 Token) | einzahlen |
| popBack | ○ | (1 Token) | einzahlen |
| reallocate$(2n)$ | $n \times$ ○ | ($n$ Token) | abheben |

Weitere Aufrufe von reallocate:

rauf: reallocate$(2n)$ $\underbrace{\overset{\geq n \times \text{pushBack}}{\rightsquigarrow}}_{\geq n \times ○○}$ reallocate$(4n)$

runter: reallocate$(2n)$ $\underbrace{\overset{\geq n/2 \times \text{popBack}}{\rightsquigarrow}}_{\geq n/2 \times ○}$ reallocate$(n)$

□





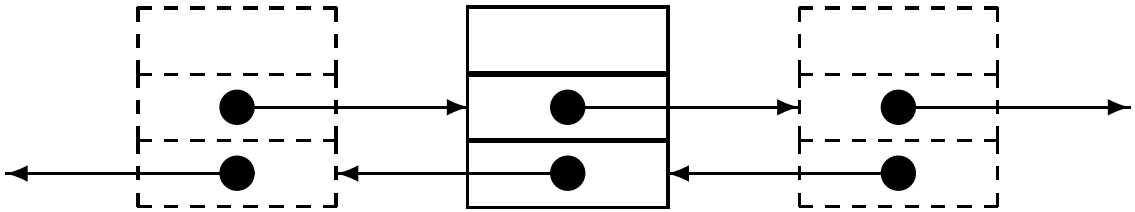
# Doppelt verkettete Listen

**Class** Item **of** Element // one link in a doubly linked list

e : Element

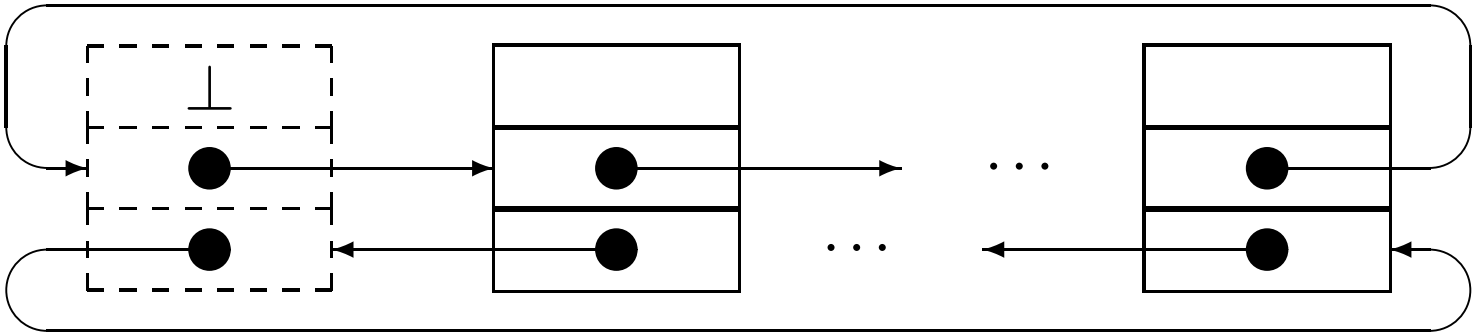
next : Handle //

prev : Handle



**invariant** next→prev=prev→next=**this**

Trick: dummy header



**Procedure** splice(a,b,t : Handle)

**assert** b is not before $a \wedge t \notin \langle a, \ldots, b \rangle$

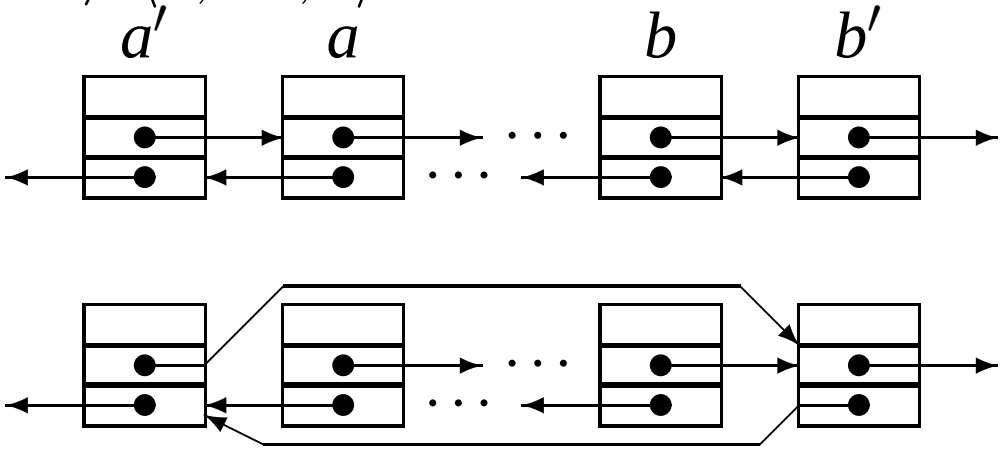
// Cut out $\langle a, \ldots, b \rangle$

$a'$ := a→prev

$b'$ := b→next

$a'$ →next := $b'$ //

$b'$ →prev := $a'$ //




// insert $\langle a, \ldots, b \rangle$ after $t$

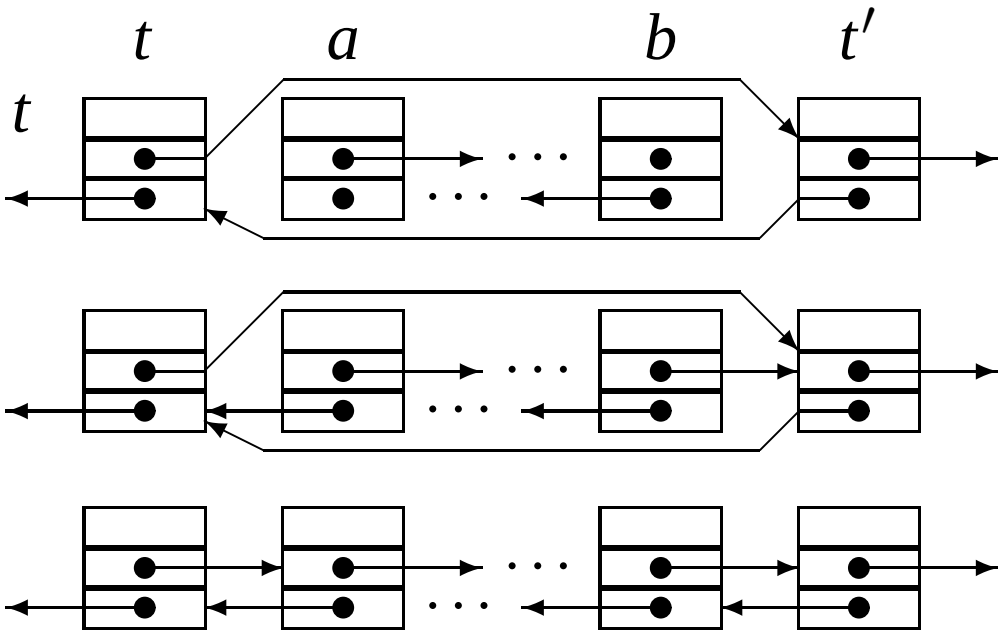
$t'$ := t→next //

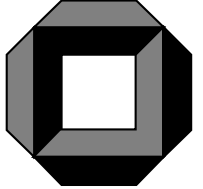
$b$→next := t' //

$a$→prev := $t$ //

$t$ →next := a //

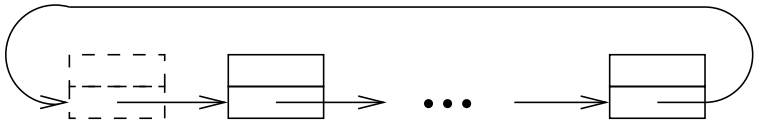
$t'$ →prev := b //





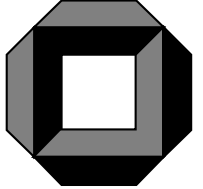


# Einfach verkettete Listen



Vergleich mit doppelt verketteten Listen

- Weniger Speicherplatz
- Platz ist oft auch Zeit
- Eingeschränkter, z.B. kein delete
- Merkwürdige Benutzerschnittstelle, z.B. deleteAfter





# Speicherverwaltung für Listen

- kann leicht 90 % der Zeit kosten!
- Lieber Elemente zwischen (Free)lists herschieben als echte `mallocs`
- Alloziere viele Items gleichzeitig
- Am Ende alles freigeben?
- Speichere „parasitär“. z.B. Graphen:

  Knotenarray. Jeder Knoten speichert ein ListItem

  $\rightsquigarrow$ Partition der Knoten kann als verkettete Listen gespeichert werden

  $\rightsquigarrow$ MST, shortest Path

Challenge: garbage collection, viele Datentypen

$\rightsquigarrow$ auch ein Software Engineering Problem hier nicht

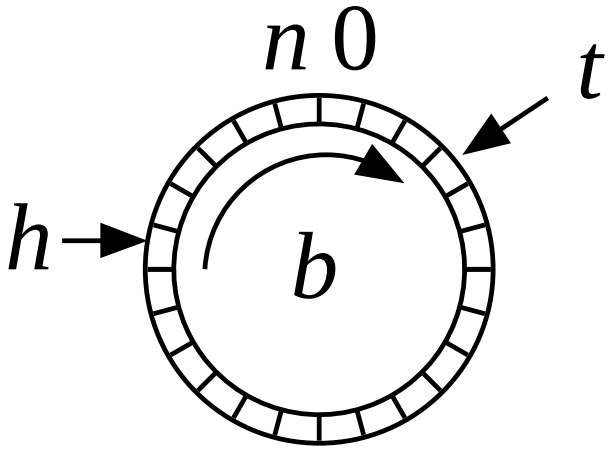
**Class** BoundedFIFO($n : \mathbb{N}$) **of** Element

$b$ : **Array** $[0..n]$ **of** Element

h$=0$ : $\mathbb{N}$

t$=0$ : $\mathbb{N}$

**Function** isEmpty : $\{0,1\}$; **return** $h = t$




**Function** first : Element; **assert** ¬isEmpty; **return** $b[h]$

**Function** size : $\mathbb{N}$; **return** $(t-h+n+1)$ **mod** $(n+1)$

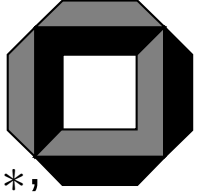
**Procedure** pushBack(x : Element)

**assert** size$< n$

$b[t]$ := x

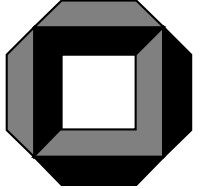
$t := (t+1)$ **mod** $(n+1)$

**Procedure** popFront **assert** ¬isEmpty; $h := (h+1)$ **mod** $(n+1)$





| Operation | List | SList | UArray | CArray | explanation '*' |
|---|---|---|---|---|---|
| $[\cdot]$ | $n$ | $n$ | 1 | 1 | |
| $\lvert\cdot\rvert$ | $1^*$ | $1^*$ | 1 | 1 | not with inter-list splice |
| first | 1 | 1 | 1 | 1 | |
| last | 1 | 1 | 1 | 1 | |
| insert | 1 | $1^*$ | $n$ | $n$ | insertAfter only |
| remove | 1 | $1^*$ | $n$ | $n$ | removeAfter only |
| pushBack | 1 | 1 | $1^*$ | $1^*$ | amortized |
| pushFront | 1 | 1 | $n$ | $1^*$ | amortized |
| popBack | 1 | $n$ | $1^*$ | $1^*$ | amortized |
| popFront | 1 | 1 | $n$ | $1^*$ | amortized |
| concat | 1 | 1 | $n$ | $n$ | |
| splice | 1 | 1 | $n$ | $n$ | |
| findNext,... | $n$ | $n$ | $n^*$ | $n^*$ | cache-efficient |





# Externe Stapel

Datei mit Blöcken

2 interne Puffer

push: Falls Platz, in Puffer.

Sonst schreibe Puffer zwei in die Datei (push auf Blockebene)

pop: Falls vorhanden, pop aus Puffer.

Sonst lese Puffer eins aus der Datei (pop auf Blockebene)

Analyse: amortisiert $O(1/B)$ I/Os pro Operation

Aufgabe 1: Beweis.

Aufgabe 2: effiziente Implementierung ohne überflüssiges Kopieren