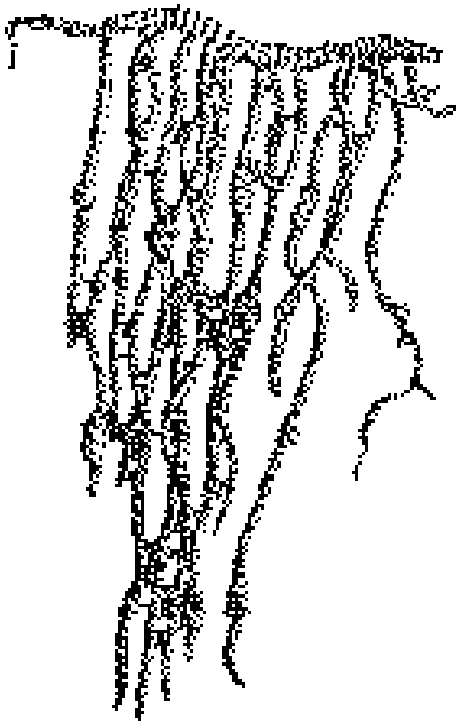
Arrays und Listen



Listen, Folgen, Sequenzen, Strings, Files, sind abstrakt gesehen alles linear angeordnete Elemente.

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

Wir schauen uns zwei Implementierungen näher an

1 Unbschränkte Felder — `vector`

Kennzeichnende Operation von Feldern: $[\cdot]$, Zugriff über Position.

Weitere Operationen:

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

Die fehlen in C-style (beschränkten) arrays!

Beispiel: Unix `sort` file

1.1 Implementierung

über beschränktes Array

Class *vector* **of** *Element*

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

n=0 : $\mathbb{N}$ // current size. **invariant** $n \leq w$

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

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

assert $0 \leq i < n$

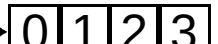
return $b[i]$

1.1.1 pushBack and popBack

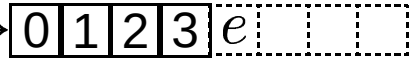
Constant β=2 : $\mathbb{R}_+$ // growth factor

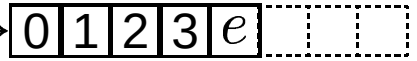
Constant α=4 : $\mathbb{R}_+$ // worst case memory blowup

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

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

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

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

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

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

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

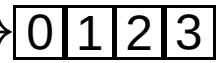
  $n{-}{-}$ // $b \rightarrow$ 

  if $w \geq \alpha n \wedge n > 0$ **then** // reduce waste of space

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

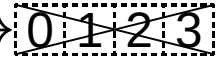
1.1.2 Reallocation

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 $b \rightarrow$ 

1.2 Analyse

Lemma 1. *m $pushBack$ und $popBack$ Operationen auf einem anfangs leeren $vector$ kosten $\mathcal{O}(m)$ Zeit*

wir zeigen, dass $\leq 3m$ Elemente in $reallocate$ kopiert werden.

Idee:

$pushBack$ zahlt 3 Einheiten “Versicherungsprämie”.

Reallocate zahlt aus den eingezahlten Prämien.

Invariante: nach reallocate sind noch n Einheiten auf dem Konto.

Beim ersten mal: $w : 1 \to 2, n = 2, \geq 6 \geq 2$ Einheiten auf dem Konto. OK.

$reallocate$ **während** $pushBack$

$\geq n/2$ übrig vom letzten Mal

$\geq n/2$ neue Elemente haben $3 \cdot n/2$ Einheiten eingezahlt

$\geq 2n$ auf dem Konto (vor dem Umkopieren)

n übrig (nachher) OK

$reallocate$ **während** $popBack$

n hat sich halbiert $\geq 2n$ übrig vom letzten Mal (vor dem Umkopieren)

n übrig (nachher) ■

1.3 Amortisierte Analyse

c_o dürfen als amortisierte Kosten der Operation o bezeichnet werde, falls die Summe der amortisierten Kosten aller ausgeführten Operationen mindestens die tatsächlichen Kosten sind.

Corollary 2. *$vector$ implementiert die Operation $[\cdot]$ in konstanter Zeit (nichtamortiesiert, schlechtester Fall) und die Operationen $pushBack$ and $popBack$ in amortisiert konstanter Zeit.*

2 Listen

Kennzeichnende Funktionsweise: Zugriff relativ zu anderen Folgenenlementen (Vorgänger/Nachfolger)

2.1 Doppelt verkettete Listen

low level Grundbaustein sind Items und deren Handles.

Type *Handle* = **Pointer to** *Item*

Class *Item* **of** *Element*

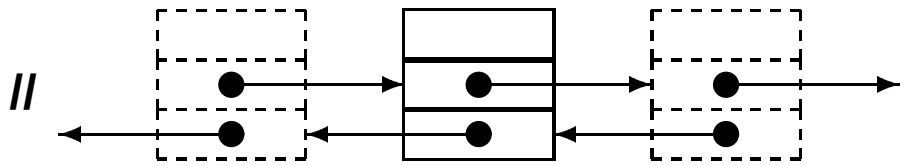
e : *Element*

next : *Handle*

prev : *Handle*

invariant $next \rightarrow prev = prev \rightarrow next = \textbf{this}$

// one link in a doubly linked list



$splice$: Eine f¨ur alles

// Remove $\langle a, \ldots, b \rangle$ from its current list and insert it after t

// $\ldots, a', a, \ldots, b, b', \ldots, t, t', \ldots \mapsto \ldots, a', b', \ldots, t, a, \ldots, b, t', \ldots$

Procedure *splice*$(a,b,t : Handle)$**assert** b *is not before* $a \wedge t \notin \langle a, \ldots, b \rangle$

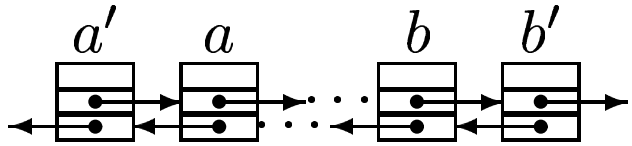
// Cut out $\langle a, \ldots, b \rangle$

$a' := a \rightarrow prev$

$b' := b \rightarrow next$

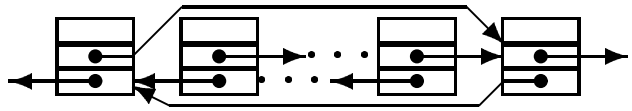
$a' \rightarrow next := b'$

$b' \rightarrow prev := a'$



//

//



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

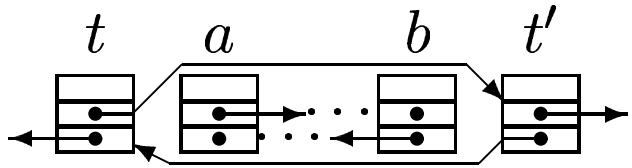
$t' := t \rightarrow next$

$b \rightarrow next := t'$

$a \rightarrow prev := t$

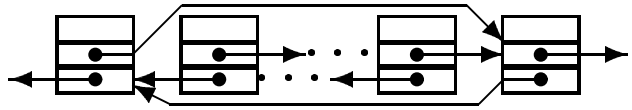
$t \rightarrow next := a$

$t' \rightarrow prev := b$



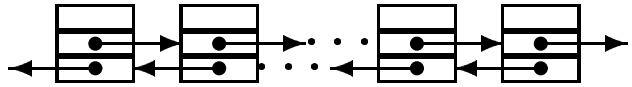
//

//



//

//

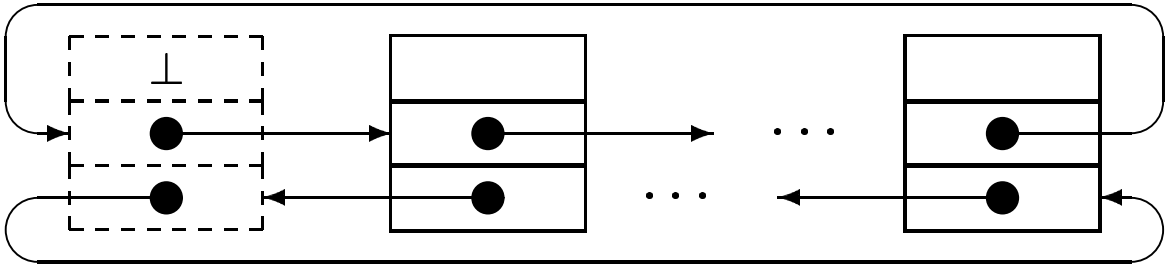


Alles hat ein Ende?

(nur die Lyoner hat keins)

Und unsere Listen auch nicht!

Die $Item$-Invariante lässt nur zyklische Strukturen zu.



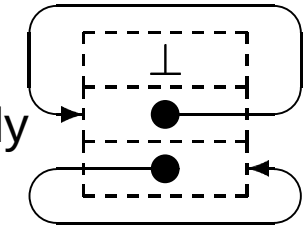
$Item$s verpackt in $List$en

Class *List* **of** *Element*

// Item h is the predecessor of the first element

// Item h is the successor of the last element.

$h = \begin{pmatrix} \perp \\ \mathbf{this} \\ \mathbf{this} \end{pmatrix}$: *Item* *//* empty list $\langle\rangle$ with dummy item only



// Simple access functions

Function *head*() : *Handle;* **return address of** *h*// Pos. before any proper element

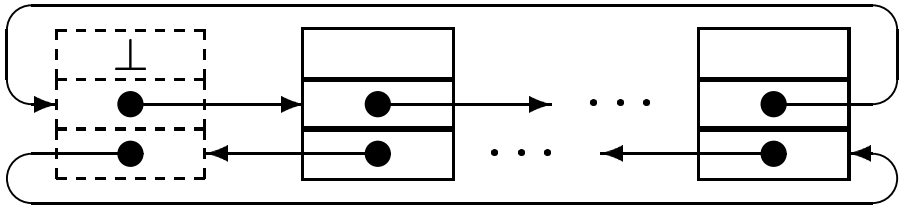
Function *isEmpty*() : **boolean** *;* **return** *h.next* = **this** *//* $\langle\rangle$?

Function *first*() : *Handle*

return *h.next*

Function *last*() : *Handle*

return *h.prev*



INFORMATIK

Elemente in einer Liste bewegen

// $(\langle \ldots, a, b, c \ldots, a', c', \ldots \rangle) \mapsto (\langle \ldots, a, c \ldots, a', b, c', \ldots \rangle)$

Procedure *moveAfter*$(b$,*a'* : $Handle)$ *splice*(b, b, a')

Procedure *moveToFront*$(b : Handle)$ *moveAfter*$(b, head())$

Procedure *moveToBack*$(b : Handle)$ *moveAfter*$(b, last())$

INFORMATIK

Einfügen und Löschen

// $\langle \ldots, a, b, c, \ldots \rangle \mapsto \langle \ldots, a, c, \ldots \rangle$

Procedure *remove*$(b : Handle)$ *moveAfter*(*b*, *freeList*.*head*$()$)

Procedure *popFront*$()$ *remove*$(first())$

Procedure *popBack*$()$ *remove*$(last())$

// $\langle \ldots, a, b, \ldots \rangle \mapsto \langle \ldots, a, e, b, \ldots \rangle$

Procedure *insertAfter*$($*x* : *Element*; *a* : *Handle*$)$

checkFreeList$()$ // make sure $freeList$ is nonempty.

moveAfter$($*freeList*.*first*$()$, *a*$)$

a$\rightarrow$*next*$\rightarrow$*e*=*x*

Procedure *insertBefore*$($*x* : *Element*; *b* : *Handle*$)$ *insertAfter(e, pred(b))*

Procedure *pushFront*$($*x* : *Element*$)$ *insertAfter(x, head*$())$

Procedure *pushBack*$($*x* : *Element*$)$ *insertAfter(x, last*$())$

Exkurs: Speicherverwaltung bei Listen u.ä.

Hier nur einige Bemerkungen

- C++ new und $delete$ für $Item$s nur wenn geeigneter STL allocators oder LEDA memory mangement verwendet wird
- Immer Blöcke für viele $Item$s allozieren (einige KByte)
- In konkreten Anwendungen kann oft die Gesamtzahl $Item$s vorausgesagt werden. Dann können alle $Item$s einmal als Array alloziert werden.

Manipulation ganzer Listen

// $(\langle a, \ldots, b\rangle, \langle c, \ldots, d\rangle) \mapsto (\langle a, \ldots, b, c, \ldots, d\rangle, \langle\rangle)$

Procedure *concat*$(o : List)$

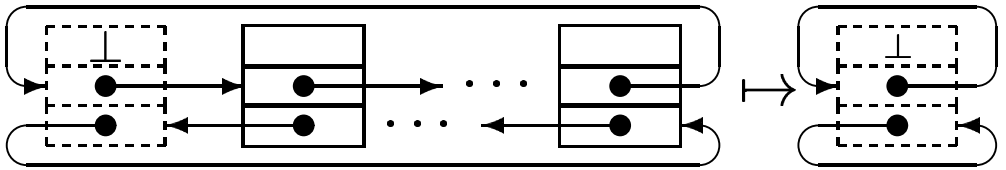
splice(o.first$()$*, o.last*$()$*, head*$()$*)*

// $\langle a, \ldots, b\rangle \mapsto \langle\rangle$

Procedure *makeEmpty*$()$

*freeList.concat(***this** *)*

//



Suchen mit Sentinels (Wächtern)

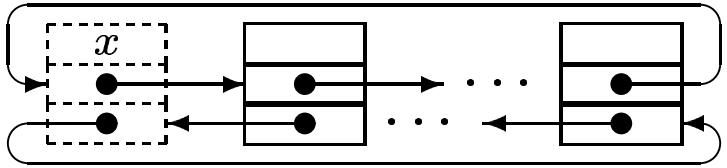
Function *findNext*(*x* : *Element; from* : *Handle*) : *Handle*

h.e = *x* // Sentinel

while *from*$\rightarrow e \neq x$ **do**

from := *from*$\rightarrow$*next*

return *from*



2.2 Singly Linked Lists

Man spart sich den Vorgängerzeiger

Aber weniger $\mathcal{O}(1)$ Operationen,

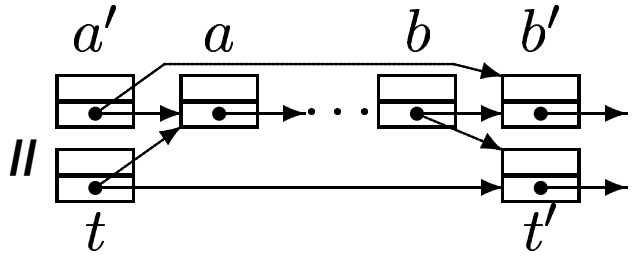
z.B. $remove \mapsto removeAfter$.

// $(\langle \ldots, a', a, \ldots, b, b' \ldots \rangle, \langle \ldots, t, t', \ldots \rangle) \mapsto$

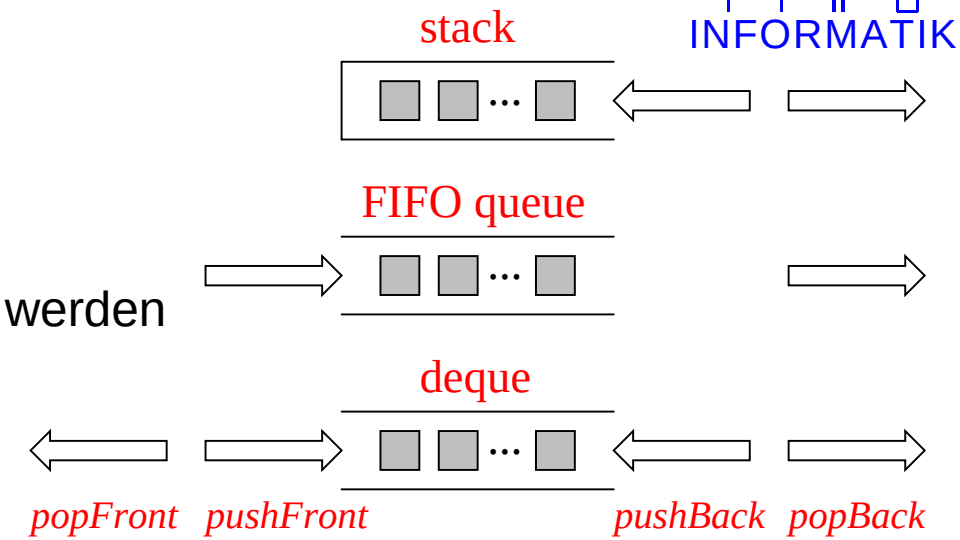
// $(\langle \ldots, a', b' \ldots \rangle, \langle \ldots, t, a, \ldots, b, t', \ldots \rangle)$

Procedure *splice*$(a'$,*b*,*t* : *SHandle*$)$

$$\begin{pmatrix} a' \to next \\ t \to next \\ b \to next \end{pmatrix} := \begin{pmatrix} b \to next \\ a' \to next \\ t \to next \end{pmatrix}$$



3 Stacks und Queues:



spezialisierte Folgen,

die nur an ihren Enden zugegriffen werden

Geht alles mit $List$en. Warum also die Aufregung?

- Fehlervermeidung
- flexiblere Implementierung (Platz, . . .)

beschränkter $stack \mapsto$ array

$stack \mapsto vector$

$stack$, ($FIFO$) $\mapsto$ singly linked list ($+$ Endezeiger)

3.1 Zyklische Arrays

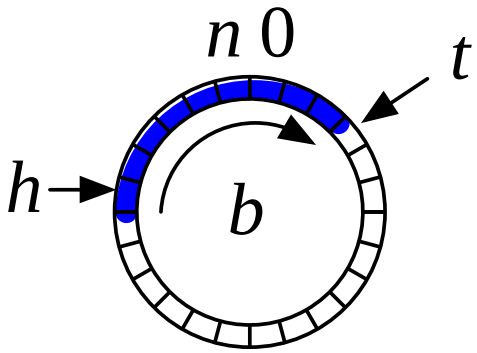
// FIFO with at most n elements

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

b : **Array** $[0..n]$ **of** $Element$ *//* $n+1$ entries!

h=*0* : $\mathbb{N}$ *//* index of first element

t=*0* : $\mathbb{N}$ *//* index of first free entry



Function *isEmpty*() : **boolean** *;* **return** $h = t$

Function *first*() : *Element;* **assert** $\neg$*isEmpty*$()$*;* **return** $b[h]$

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

Procedure *pushBack*$($*x* : *Element*$)$

assert *size*$() < n$

$b[t]$:= *x*

$t := (t + 1) \bmod (n + 1)$

Procedure *popFront*$()$ **assert** $\neg$*isEmpty*$()$*;* $h := (h + 1) \bmod (n + 1)$

INFORMATIK

4 Zusammenfassung

Operation	$List$	$SList$	$UArray$	$CArray$	explanation of '*'
$[\cdot]$	n	n	1	1	
$\|\cdot\|$	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	
$findNext, \ldots$	n	n	n^*	n^*	cache efficient