Script **generated by TTT**

Title: groh: profile1 (24.06.2016)

Date: Fri Jun 24 13:33:20 CEST 2016

Duration: 103:26 min

Pages: 51

Teil III.1: Datenstrukturen für Sequenzen

Teil III.1: Datenstrukturen für Sequenzen

basiert auf Elementen aus [3],[4] 19

Teil III.1: Datenstrukturen für Sequenzen

basiert auf Elementen aus [3],[4] 19

Operationen auf Sequenzen

Operation	liefert	Zustand nach Operation
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle[i]$	e_i (Referenz auf e_i)	unverändert
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle.\ \mathrm{set}(i, e)$	--	$\langle e_0, e_1, \ldots, e, \ldots, e_{n-1}\rangle$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{pushBack}(e)$	--	$\langle e_0, e_1, \ldots, e_{n-1}, e\rangle$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{popBack}()$	--	$\langle e_0, e_1, \ldots, e_{n-2}\rangle$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{size}()$	n	unverändert
...	...	...

Operationen auf Sequenzen

Operation	liefert	Zustand nach Operation
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle[i]$	e_i (Referenz auf e_i)	unverändert
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle.\ \mathrm{set}(i, e)$	--	$\langle e_0, e_1, \ldots, e, \ldots, e_{n-1}\rangle$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{pushBack}(e)$	--	$\langle e_0, e_1, \ldots, e_{n-1}, e\rangle$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{popBack}()$	--	$\langle e_0, e_1, \ldots, e_{n-2}\rangle$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{size}()$	n	unverändert
...	...	...

21

Statische Arrays

- in Java bspw.:
```
int[] someIntArray = new int[120];
Bicycle someBicycleArray =  new Bicycle[20];
```

- direkter Zugriff auf Elemente über Index
- Vorteile: direkter Zugriff über Index möglich, homogen im Speicher
- Nachteile: Einfügen, Löschen: schwierig,
  Erweitern: nicht möglich (→ dynamisches Array)
- typische Anwendung: lineare Algebra

Operation	Java	Kompl.
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle[i]$	`someArray[i]`	$O(1)$
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle.\ \mathrm{set}(i, e)$	`someArray[i] = e;` `if(lastFilledIndex < i) lastFilledIndex = i;`	$O(1)$ (1)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{pushBack}(e)$	`someArray[lastFilledIndex++] = e;`	$O(1)$ (2)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{popBack}()$	`lastFilledIndex--;`	$O(1)$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{size}()$	`someArray.length;`	$O(1)$
...		

(1),(2): Vorausgesetzt die Größe wird nicht überschritten;

Statische Arrays

- in Java bspw.:
```
int[] someIntArray = new int[120];
Bicycle someBicycleArray =  new Bicycle[20];
```

- direkter Zugriff auf Elemente über Index
- Vorteile: direkter Zugriff über Index möglich, homogen im Speicher
- Nachteile: Einfügen, Löschen: schwierig,
  Erweitern: nicht möglich (→ dynamisches Array)
- typische Anwendung: lineare Algebra

Operation	Java	Kompl.
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle[i]$	`someArray[i]`	$O(1)$
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle.\ \mathrm{set}(i, e)$	`someArray[i] = e;` `if(lastFilledIndex < i) lastFilledIndex = i;`	$O(1)$ (1)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{pushBack}(e)$	`someArray[lastFilledIndex++] = e;`	$O(1)$ (2)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{popBack}()$	`lastFilledIndex--;`	$O(1)$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle.\ \mathrm{size}()$	`someArray.length;`	$O(1)$
...		

(1),(2): Vorausgesetzt die Größe wird nicht überschritten;

22

Statische Arrays

- in Java bspw.:
  ```
  int[] someIntArray = new int[120];
  Bicycle someBicycleArray =  new Bicycle[20];
  ```
- direkter Zugriff auf Elemente über Index
- Vorteile: direkter Zugriff über Index möglich, homogen im Speicher
- Nachteile: Einfügen, Löschen: schwierig,
  Erweitern: nicht möglich (→ dynamisches Array)
- typische Anwendung: lineare Algebra

Operation	Java	Kompl.
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle$[i]	`someArray[i]`	$O(1)$
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle$.set(i, e)	`someArray[i] = e;` `if(lastFilledIndex < i) lastFilledIndex = i;`	$O(1)$ (1)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.pushBack(e)	`someArray[lastFilledIndex++] = e;`	$O(1)$ (2)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.popBack()	`lastFilledIndex--;`	$O(1)$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.size()	`someArray.length;`	$O(1)$
...		

(1),(2): Vorausgesetzt die Größe wird nicht überschritten;

Statische Arrays

- in Java bspw.:
  ```
  int[] someIntArray = new int[120];
  Bicycle someBicycleArray =  new Bicycle[20];
  ```
- direkter Zugriff auf Elemente über Index
- Vorteile: direkter Zugriff über Index möglich, homogen im Speicher
- Nachteile: Einfügen, Löschen: schwierig,
  Erweitern: nicht möglich (→ dynamisches Array)
- typische Anwendung: lineare Algebra

Operation	Java	Kompl.
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle$[i]	`someArray[i]`	$O(1)$
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle$.set(i, e)	`someArray[i] = e;` `if(lastFilledIndex < i) lastFilledIndex = i;`	$O(1)$ (1)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.pushBack(e)	`someArray[lastFilledIndex++] = e;`	$O(1)$ (2)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.popBack()	`lastFilledIndex--;`	$O(1)$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.size()	`someArray.length;`	$O(1)$
...		

(1),(2): Vorausgesetzt die Größe wird nicht überschritten;

22

Statische Arrays

- in Java bspw.:
  ```
  int[] someIntArray = new int[120];
  Bicycle someBicycleArray =  new Bicycle[20];
  ```
- direkter Zugriff auf Elemente über Index
- Vorteile: direkter Zugriff über Index möglich, homogen im Speicher
- Nachteile: Einfügen, Löschen: schwierig,
  Erweitern: nicht möglich (→ dynamisches Array)
- typische Anwendung: lineare Algebra

Operation	Java	Kompl.
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle$[i]	`someArray[i]`	$O(1)$
$\langle e_0, e_1, \ldots, e_i, \ldots, e_{n-1}\rangle$.set(i, e)	`someArray[i] = e;` `if(lastFilledIndex < i) lastFilledIndex = i;`	$O(1)$ (1)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.pushBack(e)	`someArray[lastFilledIndex++] = e;`	$O(1)$ (2)
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.popBack()	`lastFilledIndex--;`	$O(1)$
$\langle e_0, e_1, \ldots, e_{n-1}\rangle$.size()	`someArray.length;`	$O(1)$
...		

(1),(2): Vorausgesetzt die Größe wird nicht überschritten;

22

Dynamische Arrays

- in Java bspw.:
  ```
  Vector<ElementTyp> someVector = new Vector<ElementTyp>();
  ```
- immer wenn Array zu klein: generiere neues Array doppelter Größe und kopiere altes Array hinein
- immer wenn Array zu groß (lastIndexNotNull < n/4) generiere neues Array der halben Größe und kopiere altes Array hinein

„reallocate“

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}
```

```
void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

Dynamische Arrays

- in Java bspw.:
  `Vector<ElementTyp> someVector = new Vector<ElementTyp>();`

- immer wenn Array zu klein: generiere neues Array doppelter Größe und kopiere altes Array hinein
- immer wenn Array zu groß (lastIndexNotNull < n/4) generiere neues Array der halben Größe und kopiere altes Array hinein

} „reallocate“

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}

void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```


Dynamische Arrays

- in Java bspw.:
  `Vector<ElementTyp> someVector = new Vector<ElementTyp>();`

- immer wenn Array zu klein: generiere neues Array doppelter Größe und kopiere altes Array hinein
- immer wenn Array zu groß (lastIndexNotNull < n/4) generiere neues Array der halben Größe und kopiere altes Array hinein

} „reallocate“

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}

void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

23

Dynamische Arrays

- in Java bspw.:
  `Vector<ElementTyp> someVector = new Vector<ElementTyp>();`

- immer wenn Array zu klein: generiere neues Array doppelter Größe und kopiere altes Array hinein
- immer wenn Array zu groß (lastIndexNotNull < n/4) generiere neues Array der halben Größe und kopiere altes Array hinein

} „reallocate“

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}

void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

23

Dynamische Arrays

- in Java bspw.:
  `Vector<ElementTyp> someVector = new Vector<ElementTyp>();`

- immer wenn Array zu klein: generiere neues Array doppelter Größe und kopiere altes Array hinein
- immer wenn Array zu groß (lastIndexNotNull < n/4) generiere neues Array der halben Größe und kopiere altes Array hinein

} „reallocate“

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}

void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

23

Dynamische Arrays

- in Java bspw.:
  `Vector<ElementTyp> someVector = new Vector<ElementTyp>();`

- immer wenn Array zu klein: generiere neues Array doppelter Größe und kopiere altes Array hinein
- immer wenn Array zu groß (lastIndexNotNull < n/4) generiere neues Array der halben Größe und kopiere altes Array hinein

„reallocate“

innere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}
```

```
void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```


Dynamische Arrays

- Komplexität von reallocate: $\Theta(n)$ (es muss ja u.a. immer das gesamte Array kopiert werden) → Komplexität von pushBack und popBack auch $\Theta(n)$???

- Beobachtung: reallocate ist vergleichsweise selten notwendig. Bei genauer Betrachtung (“amortisierte Kosten”):
  Jede Folge von n pushback und popBack Operationen kann in O(n) ausgeführt warden → jede einzelne hat amortisierte Kosten von O(1)

innere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}
```

```
void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```


Asymptotische Komplexität: Landau Symbole

- Asymptotische (d.h. für großes n sich wesentlich ergebende) Zeit- und Platz-Komplexität ist nicht von Progammiersprache oder Maschinen-Charakteristika abhängig.

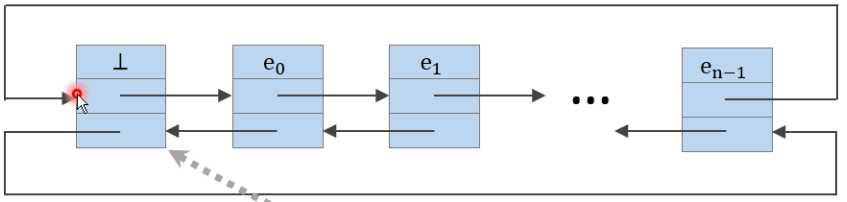
- → „wesentlich“: Asymptotische Komplexität mit Landau-Symbolen beschreiben (→ u.a. Verzicht auf konstante Faktoren):
  Für eine Funktion $f: \mathbb{N} \to \mathbb{R}$ bezeichnet

$$O(f) = \{\, g \mid \exists c > 0 : \exists n_0 > 0 :\ \forall n \geq n_o : g(n) \leq c\, f(n) \}$$

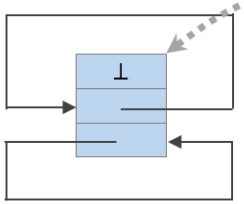
die Menge aller Funktionen $g: \mathbb{N} \to \mathbb{R}$, die asymptotisch ($\forall n \geq n_o$) wesentlich ($\exists c > 0$) höchstens so schnell wachsen wie f

Doppelt verkettete Listen

Alternative zu Array: **Doppelt verkettete Liste** (doubly linked list):



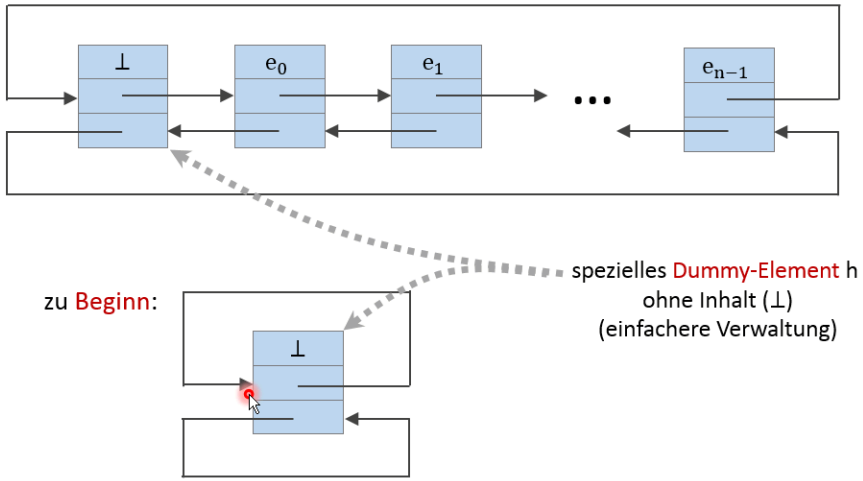
zu Beginn:



spezielles Dummy-Element h ohne Inhalt (⊥) (einfachere Verwaltung)

Doppelt verkettete Listen

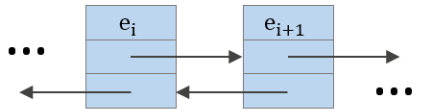
Alternative zu Array: **Doppelt verkettete Liste** (doubly linked list):



Doppelt verkettete Listen

- indirekter Zugriff über Vorgänger / Nachfolger
- Vorteile: Einfügen, Löschen, Erweitern: einfach (kein reallocate nötig)
- Nachteile: kein direkter Zugriff über Index,
  Elemente über Speicher verteilt.
- typische Anwendung: Realisierung von Stacks, Queues

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




```
class List<ElementType>{
   Item< ElementType> h; // initialisiert mit ⊥ und Referenzen auf sich selbst
  ... weitere Variablen und Methoden ...
}
```

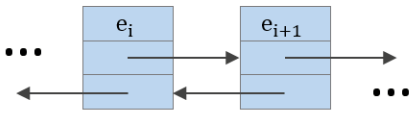
in Java Class Library bspw. `Linked List`

Invariante: `next.prev == prev.next == this`

Doppelt verkettete Listen

- indirekter Zugriff über Vorgänger / Nachfolger
- Vorteile: Einfügen, Löschen, Erweitern: einfach (kein reallocate nötig)
- Nachteile: kein direkter Zugriff über Index,
  Elemente über Speicher verteilt.
- typische Anwendung: Realisierung von Stacks, Queues

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




```
class List<ElementType>{
   Item< ElementType> h; // initialisiert mit ⊥ und Referenzen auf sich selbst
  ... weitere Variablen und Methoden ...
}
```

in Java Class Library bspw. `Linked List`

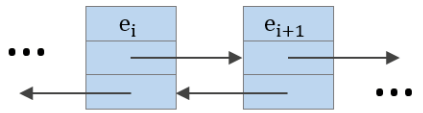
Invariante: `next.prev == prev.next == this`

26

Doppelt verkettete Listen

- indirekter Zugriff über Vorgänger / Nachfolger
- Vorteile: Einfügen, Löschen, Erweitern: einfach (kein reallocate nötig)
- Nachteile: kein direkter Zugriff über Index,
  Elemente über Speicher verteilt.
- typische Anwendung: Realisierung von Stacks, Queues

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




```
class List<ElementType>{
   Item< ElementType> h; // initialisiert mit ⊥ und Referenzen auf sich selbst
  ... weitere Variablen und Methoden ...
}
```

in Java Class Library bspw. `Linked List`

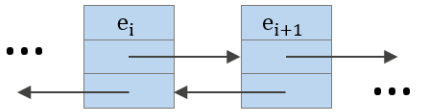
Invariante: `next.prev == prev.next == this`

26

Doppelt verkettete Listen

- indirekter Zugriff über Vorgänger / Nachfolger
- Vorteile: Einfügen, Löschen, Erweitern: einfach (kein reallocate nötig)
- Nachteile: kein direkter Zugriff über Index,
  Elemente über Speicher verteilt.
- typische Anwendung: Realisierung von Stacks, Queues

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




```
class List<ElementType>{
   Item< ElementType> h;  // initialisiert mit ⊥ und Referenzen auf sich selbst
  ... weitere Variablen und Methoden ...
}
```

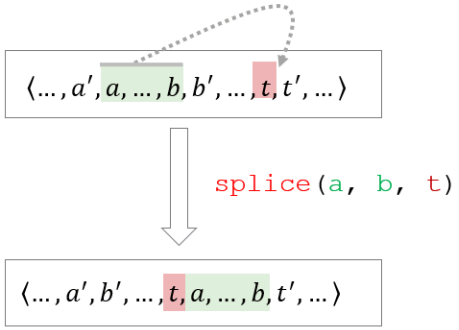
in Java Class Library bspw. `Linked List`

Invariante: `next.prev == prev.next == this`

Doppelt verkettete Listen - splice

Zentrale (statische) **Methode**:

```
splice(Item<ElementType> a,
       Item<ElementType> b,
       Item<ElementType> t):
```



Bedingungen:

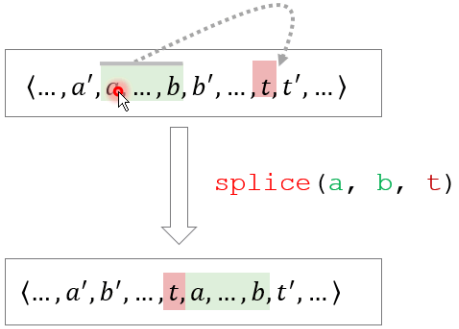
- $\langle a, \dots, b \rangle$ muss Teilsequenz sein; ($a = b$ erlaubt).
- b nicht vor a
- t nicht in $\langle a, \dots, b \rangle$

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen den Elementen selbt und Referenzen auf Elemente unterschieden

Doppelt verkettete Listen - splice

Zentrale (statische) **Methode**:

```
splice(Item<ElementType> a,
       Item<ElementType> b,
       Item<ElementType> t):
```



Bedingungen:

- $\langle a, \dots, b \rangle$ muss Teilsequenz sein; ($a = b$ erlaubt).
- b nicht vor a
- t nicht in $\langle a, \dots, b \rangle$

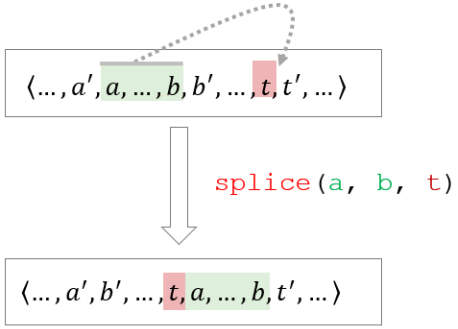
Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen den Elementen selbt und Referenzen auf Elemente unterschieden

27

Doppelt verkettete Listen - splice

Zentrale (statische) **Methode**:

```
splice(Item<ElementType> a,
       Item<ElementType> b,
       Item<ElementType> t):
```



Bedingungen:

- $\langle a, \dots, b \rangle$ muss Teilsequenz sein; ($a = b$ erlaubt).
- b nicht vor a
- t nicht in $\langle a, \dots, b \rangle$

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen den Elementen selbt und Referenzen auf Elemente unterschieden

27

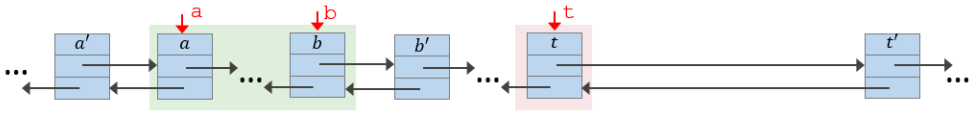
Doppelt verkettete Listen - splice

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen Items und Elementen unterschieden

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

 // schneide ⟨a, ..., b⟩ heraus:
   Item<ElementType> ap = a.prev;
   Item<ElementType> bn = b.next;
   ap.next = bn;
   bn.prev = ap;

 // füge ⟨a, ..., b⟩ hinter t ein:
   Item<ElementType> tn = t.next;
   b.next = tn;
   a.prev = t;
   t.next = a;
   tn.prev = b;
}
```



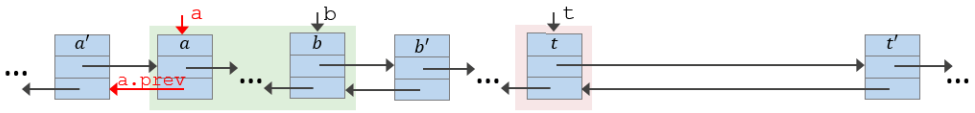
Doppelt verkettete Listen - splice

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen Items und Elementen unterschieden

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

 // schneide ⟨a, ..., b⟩ heraus:
   Item<ElementType> ap = a.prev;
   Item<ElementType> bn = b.next;
   ap.next = bn;
   bn.prev = ap;

 // füge ⟨a, ..., b⟩ hinter t ein:
   Item<ElementType> tn = t.next;
   b.next = tn;
   a.prev = t;
   t.next = a;
   tn.prev = b;
}
```



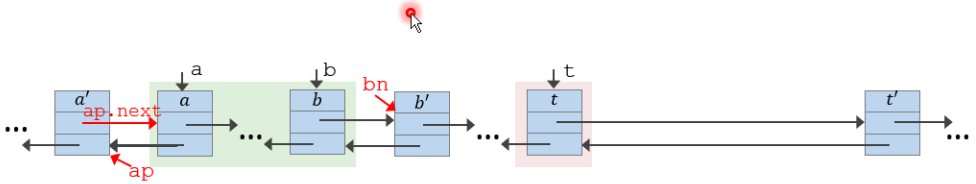
Doppelt verkettete Listen - splice

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen Items und Elementen unterschieden

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

 // schneide ⟨a, ..., b⟩ heraus:
   Item<ElementType> ap = a.prev;
   Item<ElementType> bn = b.next;
   ap.next = bn;
   bn.prev = ap;

 // füge ⟨a, ..., b⟩ hinter t ein:
   Item<ElementType> tn = t.next;
   b.next = tn;
   a.prev = t;
   t.next = a;
   tn.prev = b;
}
```



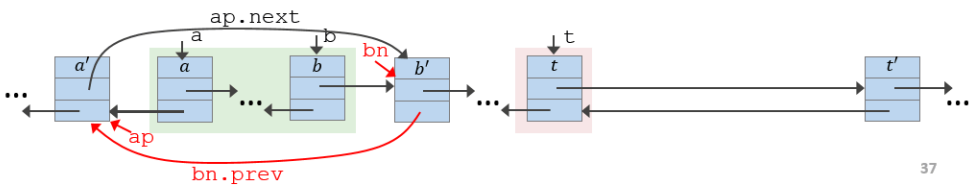
Doppelt verkettete Listen - splice

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen Items und Elementen unterschieden

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

 // schneide ⟨a, ..., b⟩ heraus:
   Item<ElementType> ap = a.prev;
   Item<ElementType> bn = b.next;
   ap.next = bn;
   bn.prev = ap;

 // füge ⟨a, ..., b⟩ hinter t ein:
   Item<ElementType> tn = t.next;
   b.next = tn;
   a.prev = t;
   t.next = a;
   tn.prev = b;
}
```



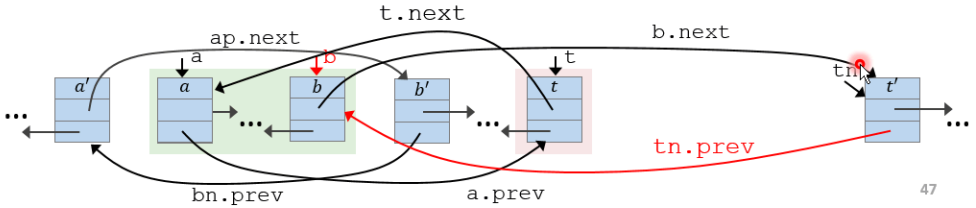
Doppelt verkettete Listen - splice

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen Items und Elementen unterschieden

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

 // schneide ⟨a, ..., b⟩ heraus:
   Item<ElementType> ap = a.prev;
   Item<ElementType> bn = b.next;
   ap.next = bn;
   bn.prev = ap;

 // füge ⟨a, ..., b⟩ hinter t ein:
   Item<ElementType> tn = t.next;
   b.next = tn;
   a.prev = t;
   t.next = a;
   tn.prev = b;
}
```



Doppelt verkettete Listen – weitere Methoden

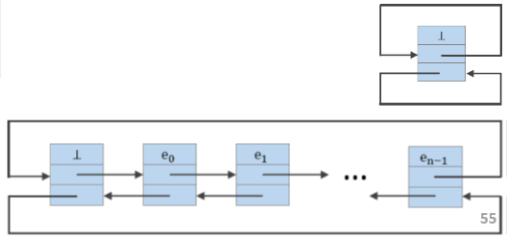
alle: Laufzeit: O(1)

```
Item<ElementType> head() {
   return h;
}
```

```
boolean isEmpty(){
   return (h.next == head());
}
```

```
Item<ElementType> first(){
   return h.next; // evtl. ⊥
}
```

```
Item<ElementType> last(){
   return h.prev; // evtl. ⊥
}
```



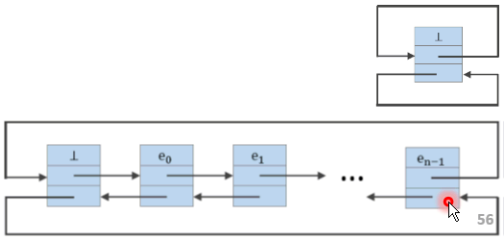
Doppelt verkettete Listen – weitere Methoden

alle: Laufzeit: O(1)

```
static moveAfter (Item<ElementType> b,
Item<ElementType> a){
   splice(b, b, a); // schiebe b hinter a
}
```

```
moveToFront (Item<ElementType> b){
   moveAfter(b, head()); // schiebe b ganz nach vorn
}
```

```
moveToBack (Item<ElementType> b){
   moveAfter(b, last()); // schiebe b ganz nach hinten
}
```



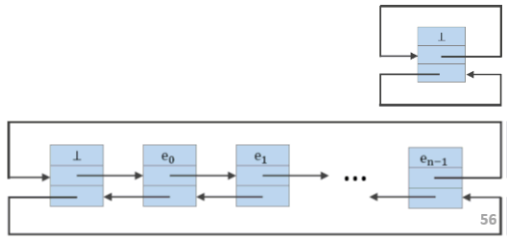
Doppelt verkettete Listen – weitere Methoden

alle: Laufzeit: O(1)

```
static moveAfter (Item<ElementType> b,
Item<ElementType> a){
   splice(b, b, a); // schiebe b hinter a
}
```

```
moveToFront (Item<ElementType> b){
   moveAfter(b, head()); // schiebe b ganz nach vorn
}
```

```
moveToBack (Item<ElementType> b){
   moveAfter(b, last()); // schiebe b ganz nach hinten
}
```



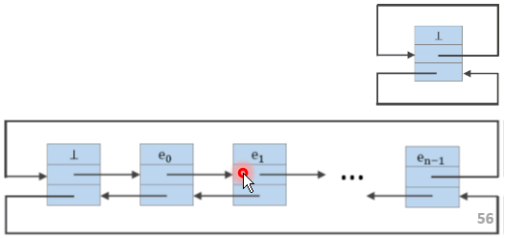
Doppelt verkettete Listen – weitere Methoden

```
static moveAfter (Item<ElementType> b,
Item<ElementType> a){
   splice(b, b, a); // schiebe b hinter a
}
```

alle: Laufzeit: O(1)

```
moveToFront (Item<ElementType> b){
   moveAfter(b, head()); // schiebe b ganz nach vorn
}
```

```
moveToBack (Item<ElementType> b){
   moveAfter(b, last()); // schiebe b ganz nach hinten
}
```



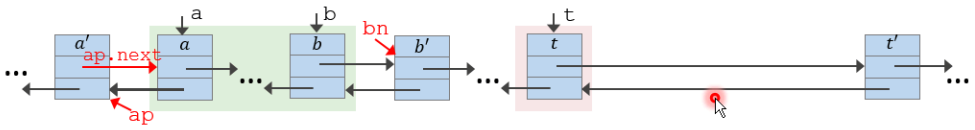
Doppelt verkettete Listen - splice

Zur besseren Verdeutlichung wird auf dieser Folie nicht zwischen Items und Elementen unterschieden

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

 // schneide ⟨a, ..., b⟩ heraus:
   Item<ElementType> ap = a.prev;
   Item<ElementType> bn = b.next;
   ap.next = bn;
   bn.prev = ap;

 // füge ⟨a, ..., b⟩ hinter t ein:
   Item<ElementType> tn = t.next;
   b.next = tn;
   a.prev = t;
   t.next = a;
   tn.prev = b;
}
```



Doppelt verkettete Listen – Einfügen und Löschen

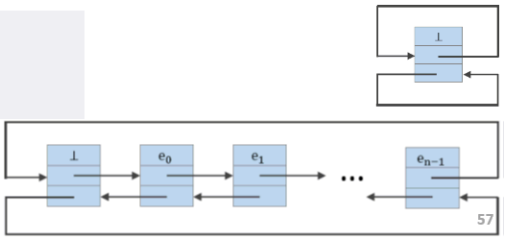
Löschen und **Einfügen** von Elementen:
mittels separater Liste freeList
→ bessere Laufzeit
(Speicherallokation teuer)

alle: Laufzeit: O(1)

```
static remove(Item<ElementType> b){
   moveAfter(b, freeList.head());
}
```

```
popFront(){
   remove(first());
}
```

```
popBack(){
   remove(last());
}
```

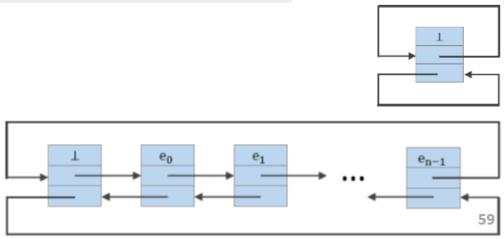


Doppelt verkettete Listen – Finden

Finden von Elementen
mittels **linearem Durchlaufen**
Trick: verwende Dummy-Element (als Wächter (Sentinel))

Laufzeit: O(n)

```
Item<ElementType> findNext(ElementType x,
Item<ElementType> from){
   h.e = x;
   while (from.e != x)
      from = from.next;
   h.e = ⊥;   //not always necessary
   return from;
}
```



Doppelt verkettete Listen – Finden

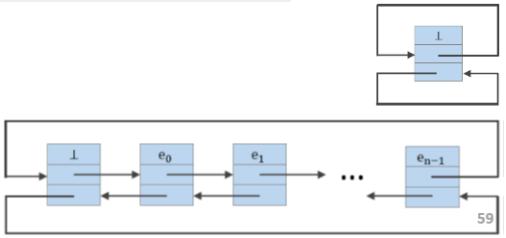
Finden von Elementen

mittels **linearem Durchlaufen**

Trick: verwende Dummy-Element (als Wächter (Sentinel))

Laufzeit: O(n)

```
Item<ElementType> findNext(ElementType x,
Item<ElementType> from){
   h.e = x;
   while (from.e != x)
      from = from.next;
   h.e = ⊥;   //not always necessary
   return from;
}
```



Doppelt verkettete Listen – Finden

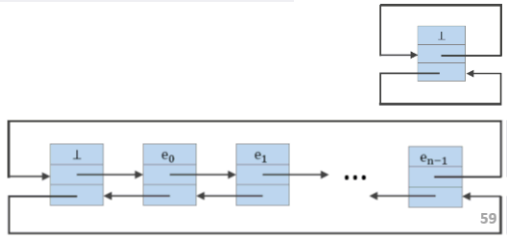
Finden von Elementen

mittels **linearem Durchlaufen**

Trick: verwende Dummy-Element (als Wächter (Sentinel))

Laufzeit: O(n)

```
Item<ElementType> findNext(ElementType x,
Item<ElementType> from){
   h.e = x;
   while (from.e != x)
      from = from.next;
   h.e = ⊥;   //not always necessary
   return from;
}
```



Doppelt verkettete Listen – Finden

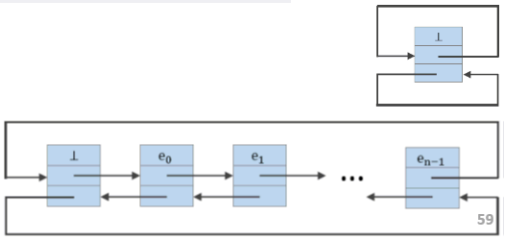
Finden von Elementen

mittels **linearem Durchlaufen**

Trick: verwende Dummy-Element (als Wächter (Sentinel))

Laufzeit: O(n)

```
Item<ElementType> findNext(ElementType x,
Item<ElementType> from){
   h.e = x;
   while (from.e != x)
      from = from.next;
   h.e = ⊥;   //not always necessary
   return from;
}
```



Doppelt verkettete Listen – Einfügen und Löschen

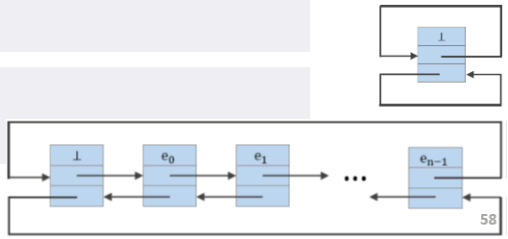
alle: Laufzeit: O(1)

```
static Item<ElementType> insertAfter(ElementType x,
Item<ElementType> a){
   checkFreeList(); // u.U. Speicher allokieren
   Item<ElementType> b = freeList.first();
   moveAfter(b, a);
   b.e = x;
   return b;
}
```

```
static Item<ElementType> insertBefore(ElementType x,
Item<ElementType> b){
   return insertAfter(x, b.prev);
}
```

```
pushFront(ElementType x){
   insertAfter(x, head());
}
```

```
pushBack(ElementType x){
   insertAfter(x, last());
}
```



Doppelt verkettete Listen – Einfügen und Löschen

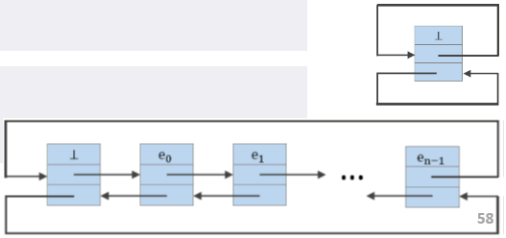
alle: Laufzeit: O(1)

```
static Item<ElementType> insertAfter(ElementType x,
Item<ElementType> a){
    checkFreeList(); // u.U. Speicher allokieren
    Item<ElementType> b = freeList.first();
    moveAfter(b, a);
    b.e = x;
    return b;
}

static Item<ElementType> insertBefore(ElementType x,
Item<ElementType> b){
   return insertAfter(x, b.prev);
}

pushFront(ElementType x){
   insertAfter(x, head());
}

pushBack(ElementType x){
   insertAfter(x, last());
}
```



Doppelt verkettete Listen – Einfügen und Löschen

alle: Laufzeit: O(1)

```
static Item<ElementType> insertAfter(ElementType x,
Item<ElementType> a){
    checkFreeList(); // u.U. Speicher allokieren
    Item<ElementType> b = freeList.first();
    moveAfter(b, a);
    b.e = x;
    return b;
}

static Item<ElementType> insertBefore(ElementType x,
Item<ElementType> b){
   return insertAfter(x, b.prev);
}

pushFront(ElementType x){
   insertAfter(x, head());
}

pushBack(ElementType x){
   insertAfter(x, last());
}
```


58

Doppelt verkettete Listen – Einfügen und Löschen

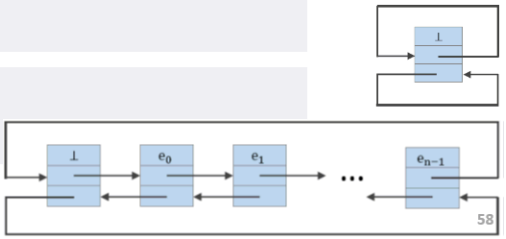
alle: Laufzeit: O(1)

```
static Item<ElementType> insertAfter(ElementType x,
Item<ElementType> a){
    checkFreeList(); // u.U. Speicher allokieren
    Item<ElementType> b = freeList.first();
    moveAfter(b, a);
    b.e = x;
    return b;
}

static Item<ElementType> insertBefore(ElementType x,
Item<ElementType> b){
   return insertAfter(x, b.prev);
}

pushFront(ElementType x){
   insertAfter(x, head());
}

pushBack(ElementType x){
   insertAfter(x, last());
}
```



58

Dynamische Arrays

- Komplexität von reallocate: $\Theta(n)$ (es muss ja u.a. immer das gesamte Array kopiert werden) → Komplexität von pushBack und popBack auch $\Theta(n)$???
- Beobachtung: reallocate ist vergleichsweise selten notwendig. Bei genauer Betrachtung (“amortisierte Kosten”):
  Jede Folge von n pushback und popBack Operationen kann in O(n) ausgeführt warden → jede einzelne hat amortisierte Kosten von O(1)

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}

void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

Dynamische Arrays

- Komplexität von reallocate: $\Theta(n)$ (es muss ja u.a. immer das gesamte Array kopiert werden) → Komplexität von pushBack und popBack auch $\Theta(n)$???
- Beobachtung: reallocate ist vergleichsweise selten notwendig. Bei genauer Betrachtung ("amortisierte Kosten"):
  Jede Folge von n pushback und popBack Operationen kann in O(n) ausgeführt warden → jede einzelne hat amortisierte Kosten von O(1)

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}
```

```
void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

24

Dynamische Arrays

- Komplexität von reallocate: $\Theta(n)$ (es muss ja u.a. immer das gesamte Array kopiert werden) → Komplexität von pushBack und popBack auch $\Theta(n)$???
- Beobachtung: reallocate ist vergleichsweise selten notwendig. Bei genauer Betrachtung ("amortisierte Kosten"):
  Jede Folge von n pushback und popBack Operationen kann in O(n) ausgeführt warden → jede einzelne hat amortisierte Kosten von O(1)

intere Realisierung in `Vector` (benutzt internes Array) ≈

```
void pushBack(ElementTyp e){
    if(lastFilledIndex ==n)
        reallocate(2*n);
    internalArray[lastFillexIndex]=e;
    lastFilledIndex++;
}
```

```
void popBack(){
   lastFilledIndex--;
   if(lastFilledIndex < n/4)
   reallocate(n/2);
}
```

```
void reallocate(int newSize){
    n = newSize;
    ElementTyp[] newInternalArray = new
       ElementTyp[newSize]();
    for (i=0; i<lastFilledIndex; i++){
       newInternalArray[i] = internalArray[i];
    }
    internalArray = newInternalArray;
}
```

24

Beispiele für abstrakte Datentypen

- $\mathbb{N}$: (Repräsentationen von) natürliche(n) Zahlen; Operationen: +, - , div, mod, …)
- Stack: Sequenz von Objekten beliebigen Typs, Operationen (i.w.): void push(e), e pop().

- Java-Klasse Bicycle: als abstrakter Datentyp aufgefasst: Menge aller Bicycle Objekte mit cadence, speed, gear als Attributen und den Methoden speedUp(), changeGear() etc.

Beispiele für abstrakte Datentypen

- $\mathbb{N}$: (Repräsentationen von) natürliche(n) Zahlen; Operationen: +, - , div, mod, …)
- Stack: Sequenz von Objekten beliebigen Typs, Operationen (i.w.): void push(e), e pop().

- Java-Klasse Bicycle: als abstrakter Datentyp aufgefasst: Menge aller Bicycle Objekte mit cadence, speed, gear als Attributen und den Methoden speedUp(), changeGear() etc.

17