

# Datenstrukturen und Algorithmen

Vorlesung 2

# Überblick

- Vorige Woche:
  - Organisatorisches
  - Abstrakter Datentyp und Datenstrukturen
  - Algorithmenanalyse
- Heute betrachten wir:
  - Arrays
  - Iteratoren

# Arrays

- Ein Array ist eine einfache Datenstruktur, die eine bestimmte Anzahl von Elementen hat, die **aufeinanderfolgende Speicherplätze** besetzen
- Arrays werden oft benutzt, um komplexere Datenstrukturen zu repräsentieren
- Bei dem Erstellen eines Arrays muss die maximale Anzahl der Elemente angegeben werden (Kapazität des Arrays)

# Arrays

- Der Speicherplatz eines Arrays berechnet man als die Kapazität multipliziert mit der Größe eines Elements
- Größe der Elemente hängt vom Typ ab:
  - Boolean – 1 Byte
  - Integer – 2 Bytes (kann abhängig von Programmiersprache, Betriebssystem, usw. variieren)
- Der Array selber wird als die Adresse des ersten Elements gespeichert

# Arrays

- Zum Beispiel, ein Array mit Boolean Elemente:

```
Address of array: 00B5FBD4
Address of 0th elemet: 00B5FBD4
Address of 1th elemet: 00B5FBD5
Address of 2th elemet: 00B5FBD6
Address of 3th elemet: 00B5FBD7
Address of 4th elemet: 00B5FBD8
Address of 5th elemet: 00B5FBD9
Address of 6th elemet: 00B5FBDA
Address of 7th elemet: 00B5FBDB
Address of 8th elemet: 00B5FBDC
Address of 9th elemet: 00B5FBDD
...
Press any key to continue . . .
```

- Boolean Werte besetzen 1 Byte

# Arrays

- Vorteil:
  - Auf jedes Element kann in konstanter Zeit ( $\Theta(1)$ ) zugegriffen werden, da die Adresse des Elementes berechnet werden kann
  - Adresse des  $i$ -ten Elementes = Array Adresse +  $i$  \* Elementgröße
- Nachteil:
  - Ein Array ist eine statische Struktur, man muss die Anzahl der Elemente von Anfang an bestimmen
    - Man kann Elemente hinzufügen oder löschen, aber die Anzahl der Slots (und dadurch der reservierte Speicherplatz) bleibt gleich
    - Wenn die Kapazität des Array zu klein ist, können wir keine neue Elemente hinzufügen
    - Wenn die Kapazität zu groß ist, dann bleibt viel Speicherplatz frei

# Dynamische Arrays

- Es gibt Arrays deren Größe variable ist → dynamische Arrays (oder dynamische Vektoren)
- Für dynamische Arrays besetzen Elemente immer noch aufeinander folgende Speicherplätze, aber diesmal muss die Länge des Arrays gemerkt werden (diese kann nicht vom Programm herausgefunden werden)

# Dynamische Arrays

- Um ein dynamisches Array zu speichern, braucht man, im Allgemeinen folgende Felder:
  - cap – die Kapazität des Arrays (Anzahl von Slots)
  - len – die eigentliche Anzahl der Elemente aus dem Array
  - elems – der eigentliche Array mit den allokierten Speicherbereich (uninitialisiert, falls es kein Element auf einem Slot gibt)

## DynamischesArray:

cap: Integer

len: Integer

elems: TElem[]



# Dynamische Arrays - resize

- Wenn die Länge des Arrays, *len*, gleich mit der Kapazität, *cap*, ist, dann ist das Array voll
- Sobald ein Element eingefügt werden soll, dass nicht mehr in das Array passen würde, allokiert man ein doppelt so großes Array und man kopiert die Elemente (zusätzliche Allokation)
- In Programmiersprachen ohne garbage collection muss man das alte Array auch deallokieren
- Optional kann die Größe des Arrays auch nach einer Löschoperation geändert werden, falls das Array zu leer wird (Speicherbereich kürzen)

# Dynamisches Array – DS vs. ADT

- Dynamisches Array ist eine Datenstruktur:
  - Es beschreibt wie die Daten gespeichert werden (aufeinanderfolgende Speicherplätze) und wie auf ein Element zugegriffen werden kann
  - Es kann als Repräsentierung genutzt werden um unterschiedliche abstrakte Datentypen zu implementieren.
- Aber, in den meisten Programmiersprachen kann man Dynamisches Array auch als Container benutzen.
  - Dynamisches Array ist nicht wirklich ein ADT, da es eine einzige mögliche Implementierung gibt, aber wir können es auch als ADT betrachten und wir werden das entsprechende Interface besprechen.

# ADT Dynamisches Array

- **Domäne** des ADT *DynamischesArray*:

$DA = \{da \mid da = (cap, len, e_1e_2e_3...e_{len}), cap, len \in \mathbb{N}, len \leq cap, e_i \text{ ist vom Typ TElem} \}$

- Das **Interface** des ADT enthält alle mögliche Operationen mit Spezifikation, Vor- und Nachbedingungen

# ADT Dynamisches Array - Interface

- `init(da, cp)`
  - **Beschreibung:** erstellt ein neues, leeres *DynamischesArray* mit der Anfangskapazität *cp* (Konstruktor)
  - **pre:**  $cp \in \mathbb{N}$
  - **post:**  $da \in DA, da.cap = cp, da.len = 0$
  - **throws:** ein Exception, falls *cp* negativ oder 0 ist
- `destroy(da)`
  - **Beschreibung:** löst ein *DynamischesArray* auf (Destruktor)
  - **pre:**  $da \in DA$
  - **post:** *da* wurde aufgelöst (der Speicherplatz des dynamische Arrays wurde deallokiert)

# ADT Dynamisches Array - Interface

- `size(da)`
  - **Beschreibung:** gibt die Länge (Anzahl der Elemente) des *Dynamisches Arrays* zurück
  - **pre:**  $da \in DA$
  - **post:**  $size \leftarrow$  Länge von  $da$  (Anzahl der Elemente)

# ADT Dynamisches Array - Interface

- getElement(da, i, e)
  - **Beschreibung:** gibt das Element des *DynamischesArray* an einer bestimmten Position aus
  - **pre:**  $da \in DA, 1 \leq i \leq da.len$
  - **post:**  $e \in TElem, e = da.e_i$  (Element an der Position  $i$ )
  - **throws:** ein Exception, falls  $i$  keine gültige Position ist

# ADT Dynamisches Array - Interface

- `setElement(da, i, e)`
  - **Beschreibung:** ändert den Wert des *Dynamisches Arrays* an einer bestimmten Position
  - **pre:**  $da \in DA, 1 \leq i \leq da.len, e \in TElem$
  - **post:**  $setElement \leftarrow da. e_i$  (alter Wert),  $da' \in DA, da'.e_i = e$  (Element an der Position  $i$  hat jetzt den Wert  $e$ )
  - **throws:** ein Exception, falls  $i$  keine gültige Position ist

# ADT Dynamisches Array - Interface

- `addToEnd(da, e)`
  - **Beschreibung:** fügt ein Element am Ende des *DynamischenArrays* ein. Falls das Array voll ist, dann wird die Kapazität vergrößert.
  - **pre:**  $da \in DA, e \in TElem$
  - **post:**  $da' \in DA, da'.len = da.len + 1,$   
 $da'.e_{da'.len} = e$   
(falls  $da.cap = da.len \Rightarrow da'.cap \leftarrow da.cap * 2$ )



# ADT Dynamisches Array - Interface

- **addToPosition**(da, i, e)
  - **Beschreibung:** fügt ein Element an einer bestimmten Position in das *DynamischeArray* ein. Falls das Array voll ist, dann wird die Kapazität vergrößert.
  - **pre:**  $da \in DA, 1 \leq i \leq da.len+1, e \in TElem$
  - **post:**  $da' \in DA, da'.len = da.len + 1,$   
 $da'.e_j = da.e_{j-1} \quad \forall j = da'.len, da'.len - 1, \dots, i+1$   
 $da'.e_i = e$   
(falls  $da.cap = da.len \Rightarrow da'.cap \leftarrow da.cap * 2$ )
  - **throws:** ein Exception, falls  $i$  keine gültige Position ist ( $da.len + 1$  ist eine gültige Position beim Einfügen)

# ADT Dynamisches Array - Interface

- `deleteFromEnd(da, e)`
  - **Beschreibung:** löscht ein Element vom Ende des *Dynamischen Arrays* und gibt das Element zurück
  - **pre:**  $da \in DA, da.len > 0$
  - **post:**  $e \in TElem, e = da.e_{da.len}$   
 $da' \in DA, da'.len = da.len - 1$
  - **throws:** ein Exception, falls *da* leer ist

# ADT Dynamisches Array - Interface

- `deleteFromPosition(da, i, e)`
  - **Beschreibung:** löscht ein Element an einer bestimmten Position des *Dynamischen Arrays* und gibt das Element zurück
  - **pre:**  $da \in DA, 1 \leq i \leq da.len$
  - **post:**  $e \in TElem, e = da.e_i$   
 $da' \in DA, da'.len = da.len - 1,$   
 $da'.e_j = da.e_{j+1} \quad \forall i \leq j \leq da'.len$   
 $da'.e_j = da.e_j \quad \forall 1 \leq j \leq i$
  - **throws:** ein Exception, falls  $i$  keine gültige Position ist

# ADT Dynamisches Array - Interface

- `Iterator(da, it)`
  - **Beschreibung:** gibt ein Iterator für das *DynamischeArray* zurück
  - **pre:**  $da \in DA$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  ist ein Iterator für  $da$ , wobei das aktuelle Element zu dem ersten Element aus  $da$  zeigt, oder, falls  $da$  leer ist, dann ist  $it$  ungültig.

# ADT Dynamisches Array - Interface

- Andere mögliche Operationen:
  - Lösche alle Elemente des *DynamischenArrays*
  - Überprüfe ob das *DynamischeArray* leer ist
  - Lösche ein Element (gegeben als Wert und nicht Position)
  - Überprüfe ob ein Element zu dem *DynamischenArray* gehört
  - Usw.

# ADT Dynamisches Array - Implementierung

- Wir besprechen die Implementierung zweier Operationen:
  - *addToEnd*
  - *addToPosition*
- Für das dynamische Array benutzen wir die vorher erwähnte Struktur:

DynamischesArray:

cap: Integer

len: Integer

elems: TElem[]

# ADT Dynamisches Array - addToEnd

Füge das Element 34 am Ende des dynamischen Arrays

|           |           |           |           |           |           |   |   |   |    |
|-----------|-----------|-----------|-----------|-----------|-----------|---|---|---|----|
| <b>51</b> | <b>32</b> | <b>19</b> | <b>31</b> | <b>47</b> | <b>23</b> |   |   |   |    |
| 1         | 2         | 3         | 4         | 5         | 6         | 7 | 8 | 9 | 10 |

- Kapazität (cap): 10
- Länge (len): 6

|           |           |           |           |           |           |           |   |   |    |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|---|---|----|
| <b>51</b> | <b>32</b> | <b>19</b> | <b>31</b> | <b>47</b> | <b>23</b> | <b>34</b> |   |   |    |
| 1         | 2         | 3         | 4         | 5         | 6         | 7         | 8 | 9 | 10 |

- Kapazität (cap): 10
- Länge (len): **7**

# ADT Dynamisches Array - addToEnd

|           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|
| <b>51</b> | <b>32</b> | <b>19</b> | <b>31</b> | <b>47</b> |
| 1         | 2         | 3         | 4         | 5         |

Füge das Element 34 am Ende des dynamischen Arrays

- Kapazität (cap): 5
- Länge (len): 5

|           |           |           |           |           |  |  |  |  |  |
|-----------|-----------|-----------|-----------|-----------|--|--|--|--|--|
| <b>51</b> | <b>32</b> | <b>19</b> | <b>31</b> | <b>47</b> |  |  |  |  |  |
| 1         | 2         | 3         | 4         | 5         |  |  |  |  |  |

|           |           |           |           |           |           |   |   |   |    |
|-----------|-----------|-----------|-----------|-----------|-----------|---|---|---|----|
| <b>51</b> | <b>32</b> | <b>19</b> | <b>31</b> | <b>47</b> | <b>34</b> |   |   |   |    |
| 1         | 2         | 3         | 4         | 5         | 6         | 7 | 8 | 9 | 10 |

- Kapazität (cap): **10**
- Länge (len): **6**



# ADT Dynamisches Array - addToEnd

**subalgorithm** addToEnd (da, e) **is:**

**if** da.len = da.cap **then**

//das dynamische Array ist voll und die Kapazität muss vergrößert werden.

da.cap  $\leftarrow$  da.cap \* 2

newElems  $\leftarrow$  @ ein Array mit da.cap neue Slots

//wir kopieren die existierenden Elemente in newElems

**for** index  $\leftarrow$  1, da.len **execute**

newElems[index]  $\leftarrow$  da.elems[index]

**end-for**

//das alte dynamische Array wird mit den neuen (größeren) Array ersetzt

//in manchen Programmiersprachen muss das alte Speicherplatz deallokiert/ frei gegeben werden

da.elems  $\leftarrow$  newElems

**end-if**

//an diesem Schritt gibt es bestimmt Platz für das neue Element

da.len  $\leftarrow$  da.len + 1

da.elems[da.len]  $\leftarrow$  e


**end-subalgorithm**

# ADT Dynamisches Array - addToPosition

Füge das Element 34 an Position 3 im dynamischen Array

|           |           |           |           |           |           |   |   |   |    |
|-----------|-----------|-----------|-----------|-----------|-----------|---|---|---|----|
| <b>51</b> | <b>32</b> | <b>19</b> | <b>31</b> | <b>47</b> | <b>23</b> |   |   |   |    |
| 1         | 2         | 3         | 4         | 5         | 6         | 7 | 8 | 9 | 10 |

- Kapazität (cap): 10
- Länge (len): 6



|           |           |           |           |           |           |           |   |   |    |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|---|---|----|
| <b>51</b> | <b>32</b> | <b>34</b> | <b>19</b> | <b>31</b> | <b>47</b> | <b>23</b> |   |   |    |
| 1         | 2         | 3         | 4         | 5         | 6         | 7         | 8 | 9 | 10 |

- Kapazität (cap): 10
- Länge (len): **7**

**subalgorithm** addToPosition (da, i, e) **is**:

**if**  $i > 0$  **and**  $i \leq \text{da.len} + 1$  **then**

**if**  $\text{da.len} = \text{da.cap}$  **then**

//das dynamische Array ist voll und die Kapazität muss vergrößert werden.

$\text{da.cap} \leftarrow \text{da.cap} * 2$

$\text{newElems} \leftarrow @$  ein Array mit  $\text{da.cap}$  neue Slots

**for**  $\text{index} \leftarrow 1, \text{da.len}$  **execute**

$\text{newElems}[\text{index}] \leftarrow \text{da.elems}[\text{index}]$

**end-for**

$\text{da.elems} \leftarrow \text{newElems}$

**end-if**

// an diesem Schritt gibt es bestimmt Platz für das neue Element

$\text{da.len} \leftarrow \text{da.len} + 1$

//die Elemente werden nach rechts verschoben um die Position  $i$  zu leeren

**for**  $\text{index} \leftarrow \text{da.len}, i+1, -1$  **execute**

$\text{da.elems}[\text{index}] \leftarrow \text{da.elems}[\text{index}-1]$

**end-for**

$\text{da.elems}[i] \leftarrow e$

**else**

$@$ throw exception

**end-if**

**end-subalgorithm**

# ADT Dynamisches Array

- Welche ist die Komplexität der Operationen `addToEnd` und `addToPosition`?

# ADT Dynamisches Array

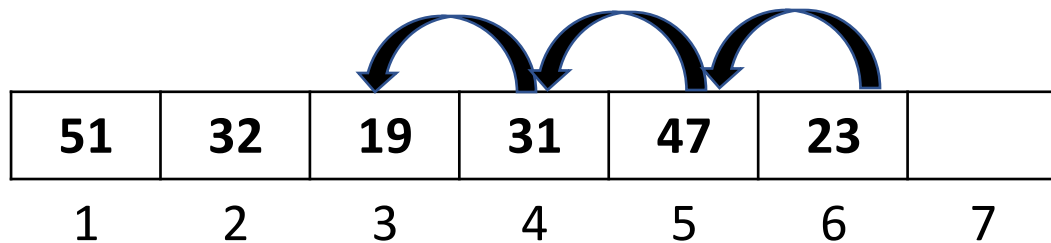
- Bemerkungen:
  - Falls die Kapazität des dynamischen Arrays 0 sein kann, dann kann man bei der Vergrößerungsoperation folgende Formel benutzen:
$$da.cap \leftarrow da.cap * 2 + 1$$
  - Nach einer Vergrößerungsoperation (resize) werden die Elemente des Arrays immer noch aufeinanderfolgenden Speicherplätze besetzen, aber nicht dieselben wie vor der Operation.

# ADT Dynamisches Array

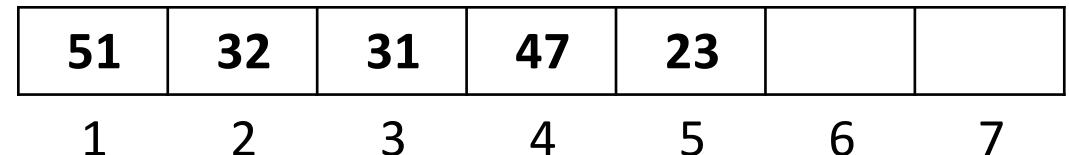
- Es ist nicht verpflichtend die Kapazität zu verdoppeln, aber es ist wichtig die **neue Kapazität als Produkt** der alten Kapazität mit einer Konstante zu definieren (anstatt die Kapazität z.B. mit 1 zu inkrementieren)
- Resize in Programmiersprachen:
  - Microsoft Visual C++ - mit 1.5 multipliziert
  - Java – mit 1.5 multipliziert
  - Python – mit ungefähr 1.125 multipliziert
  - C# - mit 2 multipliziert

# ADT Dynamisches Array - Löschoperationen

- Es gibt zwei Löschoperationen:
  - Das Element am Ende des Arrays zu löschen
  - Das Element an der Position  $i$  zu löschen: in diesem Fall müssen die Elemente nach der Position  $i$  nach links verschoben werden
- Beispiel: lösche das Element an der Position 3



- Kapazität (cap): 7
- Länge (len): 6



- Kapazität (cap): 7
- Länge (len): 5

# ADT Dynamisches Array – Komplexitäten der Operationen

- Da die Repräsentierung des dynamischen Array klar ist, können wir auch die Komplexitäten der Operationen bestimmen:

| Operation          | Komplexität             |
|--------------------|-------------------------|
| size               | $\Theta(1)$             |
| getElement         | $\Theta(1)$             |
| setElement         | $\Theta(1)$             |
| iterator           | $\Theta(1)$             |
| addToEnd           | $\Theta(1)$ amortisiert |
| addToPosition      | $O(n)$                  |
| deleteFromEnd      | $\Theta(1)$             |
| deleteFromPosition | $O(n)$                  |



# Amortisierte Analyse für Verdoppelungsstrategie

- Eine Datenstruktur wird häufig nicht nur für eine Operation, sondern für eine Reihe von  $n$  Operationen genutzt
- Betrachtet man die Kosten der einzelnen Operationen, so ergeben sich unter Umständen sehr hohe Worst-Case Kosten
- Betrachtet man jedoch die gesamte Folge, so könnte es sein, dass diese höheren Kosten nur in bestimmten Situationen eintreten (beim Schritt wo die Verdoppelung stattfindet)
- Dadurch ergeben sich insgesamt niedrigere Kosten, als bei der Verwendung der Worst-Case Laufzeit

# Amortisierte Analyse

- **Amortisierte Kosten:** benötigt man im schlechtesten Fall  $T(N)$  Zeit für  $N$  Operationen auf einer Datenstruktur, so benötigt jede Operation die amortisierte Kosten  $\frac{T(N)}{N}$
- **Wichtig!** *Amortisierte Kosten* müssen klar von *Average-Case Kosten* unterschieden werden!
- Bei *Average-Case Kosten* geht eine *Wahrscheinlichkeitsverteilung* ein, und man betrachtet die erwartete Laufzeit.
- Bei den *amortisierten Kosten* handelt es sich um eine tatsächliche *obere Schranke*, d.h.  $N$  beliebige Operationen auf der Datenstruktur benötigen höchstens  $T(N)$  Zeit. Die amortisierte Kosten für jede einzelne Operation sind dann die entstandenen *durchschnittlichen Kosten* für jede Operation.

# Amortisierte Analyse - *addToEnd*

- Worst-Case Kosten:
  - Für eine Operation, wo die Verdoppelungsoperation stattfindet, muss das alte Array in die ersten Felder des neuen Arrays kopiert werden  $\Rightarrow O(n)$
  - D.h. für  $n$  Aufrufe der Operation *addToEnd* sind die Kosten  $O(n^2)$
- Amortisierte Kosten:
  - Wir berechnen die Kosten für  $n$  Aufrufe und teilen diese durch  $n$  um die durchschnittliche Kosten zu berechnen

# Amortisierte Analyse - *addToEnd*

- Die Verdoppelungsoperation ist nicht oft nötig wenn wir  $n$  Aufrufe der Operationen betrachten
- Man muss die Kapazität bei der  $i$ -ten Operation verdoppeln, falls  $i-1$  eine Potenz von 2 ist
- D.h. die Kosten der Operation  $i$  sind:

$$c_i = \begin{cases} i, & \text{wenn } i - 1 \text{ eine Potenz von 2 ist} \\ 1, & \text{ansonsten} \end{cases}$$

# Amortisierte Analyse - *addToEnd*

- Die Kosten für  $n$  Operationen sind:

$$\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lceil \log_2 n \rceil} 2^j < n + 2n = 3n$$

- Da die gesamte Kosten für  $n$  Operationen  $3n$  sind, es folgt dass die amortisierte Kosten für eine Operation 3 sind, also konstant ( $\Theta(1)$ )

# Amortisierte Analyse - *addToEnd*

- Zusammenfassung - *addToEnd*:
  - Für die *addToEnd* Operation sind die Worst-Case Kosten  $O(n)$ , aber die amortisierte Kosten  $\Theta(1)$  (für die Verdoppelungsstrategie!)
  - Falls man eine andere Vergrößerungsstrategie benutzt, können die amortisierte Kosten unterschiedlich sein (wenn man die alte Kapazität nur mit 1 inkrementiert dann gelten die amortisierte Kosten nicht!).
- Für die *addToPosition* Operation sind sowohl die Worst-Case Kosten als auch die amortisierte Kosten  $O(n)$  – auch wenn die Kapazität nicht vergrößert werden muss, müssen die Elemente verschoben werden

# Amortisierte Analyse

- Bei der *addToEnd* Operation kann man von amortisierten Kosten reden, weil der schlimmste Fall selten vorkommt
- Wenn ihr bei einem Algorithmus nicht sicher seid ob man amortisierte Kosten berechnen kann oder nicht, dann kann man folgende Frage stellen:  
    **"Kann der schlimmste Fall bei zwei aufeinanderfolgenden Aufrufen vorkommen?"**
- Falls die Antwort „Ja“ ist, dann kann man nicht von amortisierten Kosten reden.
- Falls die Antwort „Nein“ ist, dann ist man nicht unbedingt sicher, dass der Algorithmus amortisierte Kosten hat, aber es ist auch nicht ausgeschlossen.

# Amortisierte Analyse

- Damit man nicht ein sehr leeres Dynamisches Array hat, kann man das Array nach einer Löschoperation verkleinern, falls es „zu leer“ ist
- Wie leer soll das Array sein damit man es verkleinert? Welche Strategie wäre besser?
  - Warte bis das Array halb leer ist ( $\text{da.len} \cong \text{da.cap}/2$ ) und halbiere die Kapazität
  - Warte bis das Array ein Viertel voll ist ( $\text{da.len} \cong \text{da.cap}/4$ ) und halbiere die Kapazität



# Sequentielle Repräsentierung

- Merkmale:
  - Generelle Einfüge- und Löschoperationen –  $O(n)$
  - Einfügeoperation am Ende -  $\Theta(1)$  amortisiert
  - Löschoperation vom Ende -  $\Theta(1)$
  - Ineffiziente Speicherplatz Verwaltung
  - Direkter Zugriff zu den Elementen -  $\Theta(1)$

# Iterator

- Ein Iterator dient dazu, über alle Elemente einer beliebigen Sammlung (Container) durchzulaufen.
- Behälter/Containers haben unterschiedliche Repräsentierungen, mit unterschiedlichen Datenstrukturen. Iteratoren bieten ein **generisches** Framework, um beliebige Datenstrukturen zu durchlaufen ohne Berücksichtigung einer speziellen Art der Darstellung.
- Jeder Container, den man durchlaufen kann, muss in der Schnittstelle eine Operation *iterator* haben, welche ein Iterator für den Container zurückgibt

# Iterator

- Ein Iterator enthält normalerweise:
  - Eine **Referenz zu dem Container**, den er durchläuft
  - Eine **Referenz zu dem *aktuellen* Element** (current element) aus dem Container
- Zum Iterieren der Elemente des Containers verschiebt man die Referenz des aktuellen Elementes von einem Element zu dem nächsten bis der Iterator *ungültig* wird
- Die Repräsentierung des aktuellen Elementes hängt von der Datenstruktur ab, die bei der Implementierung des Containers benutzt wurde
- Falls sich die Datenstruktur/Implementierung des Containers ändert, dann muss auch die Repräsentierung/Implementierung des Iterators geändert werden.

# Iterator - Domäne

- **Domäne** des Iterators:

$I = \{it \mid it \text{ ist ein Iterator für den Container mit Elementen vom Typ TElem}\}$

# Iterator - Interface

- `init(it, c)`
  - **Beschreibung:** erstellt einen neuen Iterator für einen Container
  - **pre:**  $c$  ist ein Container
  - **post:**  $it \in \mathcal{I}$  und  $it$  zeigt auf das erste Element in  $c$  falls  $c$  nicht leer ist, ansonsten ist  $it$  nicht gültig

# Iterator - Interface

- `getCurrent(it)`
  - **Beschreibung:** gibt das aktuelle Element aus dem Iterator zurück
  - **pre:**  $it \in \mathcal{I}$ ,  $it$  ist gültig
  - **post:** ,  $getCurrent \leftarrow e$ ,  $e \in TElem$ ,  $e$  ist das aktuelle Element aus  $it$

# Iterator - Interface

- `next(it)`
  - **Beschreibung:** das aktuelle Element wird mit dem nächsten Element aus dem Container ersetzt oder der Iterator wird ungültig falls es keine Elemente mehr gibt
  - **pre:**  $it \in I$ ,  $it$  ist gültig
  - **post:**  $it' \in I$ , das aktuelle Element aus  $it'$  zeigt auf das nächste Element aus dem Container oder  $it'$  wird ungültig falls es keine Elemente mehr gibt

# Iterator - Interface

- `valid(it)`
  - **Beschreibung:** überprüft ob der Iterator gültig ist oder nicht
  - **pre:**  $it \in I$
  - **post:**  
 $valid \rightarrow \begin{cases} True, & \text{wenn der Iterator auf ein gültiges Element zeigt} \\ False, & \text{ansonsten} \end{cases}$



# Iterator - Interface

- `first(it)`
  - **Beschreibung:** setzt das aktuelle Element aus dem Iterator auf das erste Element
  - **pre:**  $it \in I$
  - **post:**  $it' \in I$ , das aktuelle Element aus  $it'$  zeigt auf das erste Element aus dem Container falls dieser nicht leer ist, oder  $it'$  ist ungültig ansonsten

# Arten von Iteratoren

- Das bisherige Interface stellt den einfachsten Iterator dar: *unidirektional* und *read-only*
- Ein ***unidirektionaler*** Iterator kann einen Container nur in einer Richtung durchlaufen (meistens vorwärts, aber es gibt auch reverse-Iteratoren)
- Ein ***bidirektionaler*** Iterator kann den Container in beide Richtungen durchlaufen: außer der Operation *next* gibt es auch eine Operation *previous*

# Arten von Iteratoren

- Ein ***random access*** Iterator kann mit mehreren Schritten auf einmal verschoben werden (anstatt mit einem einzigen Schritt nach vorne oder nach hinten)
- Ein ***read-only*** Iterator kann den Container durchlaufen, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich (es kann die Elemente nicht ändern)
- Ein ***read-write*** Iterator kann für das Hinzufügen neuer Elemente in den Container oder Löschen von Elementen aus dem Container benutzt werden

# Iterator

- Da die Schnittstelle eines Iterators nicht abhängig von der Repräsentierung des Containers ist, können die Elemente des Containers folgendermaßen ausgedruckt werden:

**subalgorithm** printContainer(c) is:

//pre: c ist ein Container

//post: die Elemente aus c wurden ausgedruckt

//man erstellt ein Iterator mit der *iterator* Operation aus dem Containers

iterator(c, it)

**while** valid(it) **execute**

//das aktuelle Element aus dem Iterator wird zurückgegeben

elem ← getCurrent(it)

**print** elem

//der Iterator rückt zu dem nächsten Element

next(it)

**end-while**

**end-subalgorithm**

# Iterator für dynamisches Array

- Wie kann man ein Iterator für ein dynamisches Array definieren?
- Wie kann das *aktuelle Element* aus dem Iterator repräsentiert werden?
- Im Falle eines dynamischen Arrays ist es am einfachsten die Position des aktuellen Elementes zu speichern

IteratorDA:

da: DynamischesArray

current: Integer

# Iterator für dynamisches Array - init

**subalgorithm** init(it, da) is:

*//it ist ein IteratorDA, da ist ein DynamischesArray*

it.da  $\leftarrow$  da

it.current  $\leftarrow$  1

**end-subalgorithm**

- Komplexität:  $\Theta(1)$

# Iterator für dynamisches Array - getCurrent

```
subalgorithm getCurrent(it) is:  
    getCurrent ← it.da.elems[it.current]  
end-subalgorithm
```

- Komplexität:  $\Theta(1)$

# Iterator für dynamisches Array - next

```
subalgorithm next(it) is:  
    it.current  $\leftarrow$  it.current + 1  
end-subalgorithm
```

- Komplexität:  $\Theta(1)$



# Iterator für dynamisches Array - valid

```
subalgorithm valid(it) is:  
  if it.current  $\leq$  it.da.len then  
    valid  $\leftarrow$  True  
  else  
    valid  $\leftarrow$  False  
  end-if  
end-subalgorithm
```

- Komplexität:  $\Theta(1)$

# Iterator für dynamisches Array - first

**subalgorithm** first(it) is:

it.current  $\leftarrow$  1

**end-subalgorithm**

- Komplexität:  $\Theta(1)$

# Iterator für dynamisches Array

- Methoden für das Ausdrucken der Elemente eines dynamischen Arrays:
  - Mithilfe eines Iterators (wie wir vorher gesehen haben)
  - Mithilfe der Indexe der Elemente

**subalgorithm** printDA(da) is:

//pre: da ist ein Dynamisches Array

//post: die Elemente von da wurden ausgedruckt

**for**  $i \leftarrow 1, \text{size}(\text{da})$  **execute**

        elem  $\leftarrow$  getElement(da, i)

**print** elem

**end-for**

**end-subalgorithm**

# Iterator für dynamisches Array

- Für ein dynamisches Array haben beide Algorithmen, die die Elemente ausdrucken, die Komplexität  $\Theta(n)$
- Für andere Datenstrukturen/ Containers braucht man einen Iterator, weil:
  - Es keine Indexe gibt für die Elemente
  - Die Zeitkomplexität kleiner ist, wenn man ein Iterator benutzt um alle Elemente zu durchlaufen

# Iterator

- Für alle Containers, die iteriert werden können, besprechen wir die Implementierung des Iterators bezüglich der Repräsentierung des Containers
- Warum sind Iteratoren so wichtig?
  - uniforme Iterierung der Elemente eines Containers (unabhängig von dem Container und von der Repräsentierung)!!
  - für viele Containers bietet der Iterator die einzige Möglichkeit den Container zu durchlaufen

# Iterator

- Wenn man in einem Container auf die Positionen der Elemente verzichtet, dann kann man manche Operationen optimieren:
  - Man kann Datenstrukturen benutzen, die eine gute Komplexität für die Operationen haben, aber wo die Positionen der Elemente keine Rolle spielen und wo es schwer ist die Positionen zu erzwingen (z.B. Hashtabellen)
- Auch wenn der Container Positionen für die Elemente hat, kann es effizienter sein den Container mit dem Iterator zu durchlaufen anstatt mit Hilfe der Positionen (Position inkrementieren und Element zurückgeben)