

Datenstrukturen und Algorithmen

Vorlesung 3

Überblick

- Vorige Woche:
 - Arrays
 - Iteratoren
- Heute betrachten wir:
 - ADT Bag & SortedBag
 - ADT Set & SortedSet
 - ADT Map & SortedMap
 - ADT Matrix
 - ADT Queue

ADT Bag / MultiSet

(Interface und Implementierung besprochen im Seminar 1)

- Domäne:

$$\mathcal{B} = \{b \mid b \text{ ist ein Bag mit Elementen vom Typ TElem}\}$$

- ADT Bag ist ein Container, in welcher Elemente nicht eindeutig sind und keine Positionen haben (ein Element kann vielfach enthalten sein)

ADT Bag - Interface

- $\text{init}(b)$
 - **pre:** true
 - **post:** $b \in \mathcal{B}$, b ist ein leeres Bag
- $\text{add}(b, e)$
 - **pre:** $b \in \mathcal{B}$, $e \in \text{TElem}$
 - **post:** $b' \in \mathcal{B}$, $b' = b \cup \{e\}$ (TElem e wird in dem Bag eingefügt)
- $\text{remove}(b, e)$
 - **pre:** $b \in \mathcal{B}$, $e \in \text{TElem}$
 - **post:** $b' \in \mathcal{B}$, $b' = b \setminus \{e\}$ (**ein Vorkommen**/occurrence des Elementes e wurde aus dem Bag entfernt). Falls e nicht in b enthalten war, dann bleibt b unverändert

$$\text{remove} \leftarrow \begin{cases} \text{true}, & \text{falls ein Element entfernt wurde } (\text{size}(b') < \text{size}(b)) \\ \text{false}, & \text{falls } e \text{ nicht in } b \text{ enthalten war } (\text{size}(b') = \text{size}(b)) \end{cases}$$

ADT Bag - Interface

- $\text{search}(b, e)$
 - **pre:** $b \in \mathcal{B}, e \in T\text{Elem}$
 - **post:** $\text{search} \leftarrow \begin{cases} \text{wahr,} & \text{falls } e \in b \\ \text{falsch,} & \text{ansonsten} \end{cases}$
- $\text{size}(b)$
 - **pre:** $b \in \mathcal{B}$
 - **post:** $\text{size} \leftarrow \text{Anzahl der Elemente aus } b$
- $\text{nrOccurrences}(b, e)$
 - **pre:** $b \in \mathcal{B}, e \in T\text{Elem}$
 - **post:** $\text{nrOccurrences} \leftarrow \text{Vorkommen des Elementes } e \text{ in } b$

ADT Bag - Interface

- **isEmpty(b)**
 - **descr:** überprüft ob das Bag leer ist
 - **pre:** $b \in \mathcal{B}$
 - **post:** $isEmpty \leftarrow \begin{cases} \text{wahr, falls } b \text{ keine Elemente enthält} \\ \text{falsch, ansonsten} \end{cases}$
- **destroy(b)**
 - **pre:** $b \in \mathcal{B}$
 - **post:** b wurde zerstört
- **iterator(b, i)**
 - **pre:** $b \in \mathcal{B}$
 - **post:** $i \in \mathcal{I}$, i ist ein Iterator für b

ADT Bag – Repräsentierung A

- **Ein dynamisches Array, wo die Elemente vielfach vorkommen können** (Beispiel im Seminar 1)
- Zum Beispiel, wenn das Bag folgende Elemente enthält: 1, 3, 2, 6, 2, 5, 2, dann kann man diese in einem Array folgendermaßen speichern:

1	3	2	6	2	5
---	---	---	---	---	---

- Lösche das Element 6:

1	3	2	5	2	
---	---	---	---	---	--

ADT Bag – Repräsentierung B

- Ein dynamisches Array von Paaren der Form (*Element*, *Frequenz*), wobei die Elemente eindeutig sind, und zusätzlich speichert man für jedes Element die Frequenz
- Zum Beispiel, wenn das Bag folgende Elemente enthält: 1, 2, 5, 2, 3, 3, 1, 2, 1, 2, 6 dann kann man diese in einem Array folgendermaßen speichern:

(1, 3)	(2, 4)	(5, 1)	(3, 2)	(6, 1)		
--------	--------	--------	--------	--------	--	--

- oder

elems	1	2	5	3	6		
freq	3	4	1	2	1		

ADT Bag – Repräsentierung B

elems	1	2	5	3	6		
freq	3	4	1	2	1		

- Füge das Element 2 ein:

elems	1	2	5	3	6		
freq	3	5	1	2	1		

- Füge das Element -5 ein:

elems	1	2	5	3	6	-5	
freq	3	5	1	2	1	1	

ADT Bag – Repräsentierung B

elems	1	2	5	3	6	-5	
freq	3	5	1	2	1	1	

- Lösche das Element 5:

elems	1	2	-5	3	6		
freq	3	5	1	2	1		

- Lösche das Element 2:

elems	1	2	-5	3	6		
freq	3	4	1	2	1		

ADT Bag – Repräsentierung C

- **Ein dynamisches Array von Frequenzen**, folgendermaßen aufgebaut:
 - Man berechnet das aktuelle Werteintervall $[a,b]$ (funktioniert nur wenn die Werte ganze Zahlen sind)
 - Das Werteintervall wird auf das Intervall $[1,x]$ übersetzt, so dass $x = b - a + 1$
 - In dem Array speichert man:
 - Auf Position 1 die Frequenz des Wertes a (der minimale Wert)
 - Auf Position 2 die Frequenz des Wertes $a+1$
 - ...
 - Auf Position $x-1$ die Frequenz des Wertes $b-1$
 - Auf Position x die Frequenz des Wertes b (der maximale Wert)
- Wenn ein Wert eingefügt wird, welcher sich außerhalb des Intervalls $[a,b]$ befindet, dann muss das Werteintervall und folglich auch das Array von Frequenzen aktualisiert werden

ADT Bag – Repräsentierung C

- Nehmen wir als Beispiel ein Bag, das folgende Elemente enthält:

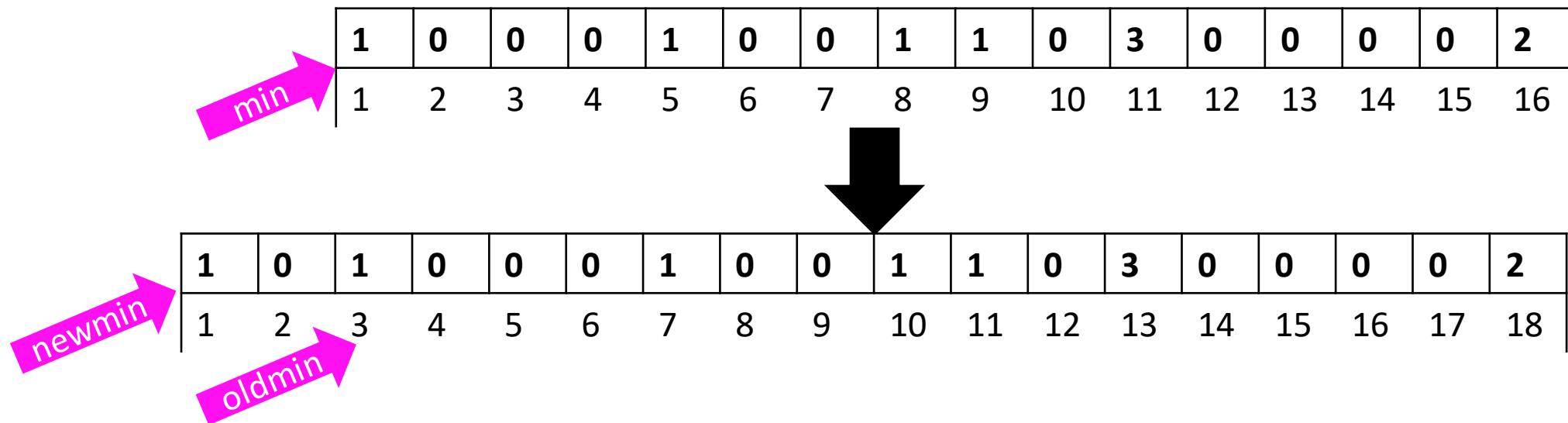
5, 10, -1, 2, 3, 10, 5, 5, -5

- Das Werteintervall $[-5,10]$ wird in das Intervall $[1,16]$ übersetzt, d.h. :
 - auf der Position 1 speichert man die Frequenz des Wertes -5 (minimaler Wert), auf der Position 2 die Frequenz des Wertes -4, ..., auf der Position 16 die Frequenz des Wertes 10
- Es wird in dem Array folgendermaßen gespeichert:
 - Bemerkung: Die eigentlichen Werte werden nicht im Array gespeichert. Welche Werte braucht man?

Freq	1	0	0	0	1	0	0	1	1	0	3	0	0	0	0	2
index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Wert	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10

ADT Bag – Repräsentierung C

- Wenn man das Element -7 einfügt, dann ändert sich das Werteintervall: [-7,10]
- Das heißt Position 1 in dem Array entspricht jetzt dem Wert -7 (alle Werte müssen mit zwei Positionen nach rechts verschoben werden)
- Bemerkung: wenn *min* der minimale Wert ist, dann ist die **Position** eines Elementes *e* genau ***e-min+1***



ADT Bag – Repräsentierung D

- Ein dynamisches Array von eindeutigen Elementen (E) und ein dynamisches Array von Positionen (P) in E für die Elemente des Bags
- Das Array E enthält also die Menge aller Werte des Bags
- Das Array P hat folgende Bedeutung:
 - $P[1] = \mathbf{a}$, i.e. das erste Element des Bags ist $E[\mathbf{a}]$
 - $P[2] = \mathbf{b}$, i.e. das nächste Element des Bags ist $E[\mathbf{b}]$
 - ...
- Nehmen wir als Beispiel ein Bag, das folgende Elemente enthält:
5, 10, -1, 2, 3, 10, 5, 5, -5
- Es wird folgendermaßen repräsentiert:
 - $E = [5, 10, -1, 2, 3, -5]$
 - $P = [1, 2, 3, 4, 5, 2, 1, 1, 6]$

ADT Bag – Repräsentierung D

- Füge das Element 2 ein:

E:

5	10	-1	2	3	-5			
1	2	3	4	5	6	7	8	9

P:

1	2	3	4	5	2	1	1	6	4		
1	2	3	4	5	6	7	8	9	10	11	12

E:

5	10	-1	2	3	-5			
1	2	3	4	5	6	7	8	9

P:

1	2	3	4	5	2	1	1	6			
1	2	3	4	5	6	7	8	9	10	11	12

- Füge das Element 6 ein:

E:

5	10	-1	2	3	-5	6		
1	2	3	4	5	6	7	8	9

P:

1	2	3	4	5	2	1	1	6	4	7	
1	2	3	4	5	6	7	8	9	10	11	12

ADT Bag – Repräsentierung D

- Lösche das Element 2:

E:

5	10	-1	2	3	-5	6		
1	2	3	4	5	6	7	8	9

P:

1	2	3	7	5	2	1	1	6	4		
1	2	3	4	5	6	7	8	9	10	11	12

E:

5	10	-1	2	3	-5	6		
1	2	3	4	5	6	7	8	9

P:

1	2	3	4	5	2	1	1	6	4	7	
1	2	3	4	5	6	7	8	9	10	11	12

- Lösche das Element -1:

E:

5	10	6	2	3	-5			
1	2	3	4	5	6	7	8	9

Aufpassen!

P:

1	2	4	3	5	2	1	1	6			
1	2	3	4	5	6	7	8	9	10	11	12

ADT Bag – Repräsentierung

- Bemerkung. Für Repräsentierung A und B können auch andere Datenstrukturen benutzt werden.
- Repräsentierung C und D sind spezifisch für ein dynamisches Array.

ADT Sorted Bag

- In einem Bag können die Elemente basierend auf einer Ordnungsrelation sortiert werden → *SortedBag*
- In diesem Fall enthält das SortedSet Elemente vom Typ *TComp* anstatt Elemente vom Typ *TElem*

ADT Sorted Bag

- Welche Operationen muss man ändern?
- Die einzigen Änderungen zu dem Interface sind bei der *init* Operation, wo man auch die Relation als Parameter hat
- Domäne: $SB = \{sb \mid sb \text{ ist ein Sorted Bag mit Elementen vom Typ TComp}\}$
- *init*(sb, rel)
 - **desc:** erstellt einen leeren Sorted Bag, wo die Elemente basierend auf der Relation *rel* geordnet werden
 - **pre:** *rel* ∈ Relation
 - **post:** *sb* ∈ *SB*, *sb* ist ein leeres SortedBag mit *rel* als Relation

ADT Sorted Bag

- Die Relation bestimmt wie die Elemente sortiert werden (z.B. steigende Reihenfolge, alphabetische Reihenfolge, usw.)
- Die Relation wird als Funktion mit zwei Parametern definiert (die zwei Elemente, die verglichen werden), welche *true* zurückgibt, falls die Elemente in der richtigen Reihenfolge sind, oder *false*, falls die Elemente nicht in der richtigen Reihenfolge sind.

ADT Sorted Bag - Iterator

- Für einen sortierten Bag muss der Iterator die Elemente in der Reihenfolge gegeben von der Relation durchlaufen
- Damit die Iterator Operationen Komplexität $\Theta(1)$ haben, sollen die Elemente sortiert gespeichert werden

Bag/SortedBag - Repräsentierung

- Um ADT Bag (oder ADT SortedBag) zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
 - (dynamisches) Array
 - Verkettete Liste
 - Hashtabellen
 - (balancierte) Binärbäume – für sortierte Bags
 - Skip Listen – für sortierte Bags

Bag/SortedBag - Repräsentierung

- Unabhängig von der Datenstruktur gibt es zwei Möglichkeiten, die Elemente zu speichern:
 - Alle Elemente separat zu speichern (in der entsprechenden Reihenfolge)
 - Man speichert eindeutige Elemente (in der entsprechenden Reihenfolge) und die Frequenzen

Aufgabe

- Um Wahlbetrug zu vermeiden (jede Person darf ein einziges Mal wählen) braucht man eine Anwendung um die Personalnummer der Personen zu speichern, die schon gewählt haben.
- Welche Eigenschaften sollte der Container haben?
 - Die Elemente sollen eindeutig sein
 - Die Reihenfolge der Elemente ist nicht wichtig
- Ein Container, dessen Elemente eindeutig sind, wobei die Reihenfolge der Elemente keine Rolle spielt (die Elemente haben keine entsprechenden Positionen) ist **ADT Set**

ADT Set

- Es gibt keine Operationen basierend auf Positionen
- Die Elemente sind nicht unbedingt in derselben Reihenfolge gespeichert, in der sie eingefügt wurden
- Domäne für ADT Set:
 $\mathcal{S} = \{s \mid s \text{ ist ein Set mit Elementen vom Typ TElem}\}$

Set – Interface

- **init(s)**
 - **descr:** erstellt einen leeren Set
 - **pre:** wahr
 - **post:** $s \in \mathcal{S}$, s ist einen leeren Set
- **add(s, e)**
 - **descr:** fügt ein neues Element zu dem Set ein
 - **pre:** $s \in \mathcal{S}$, $e \in TElem$
 - **post:** $s' \in \mathcal{S}$, $s' = s \cup \{e\}$ (TElem e wird in dem Set eingefügt nur **falls er noch nicht in dem Set enthalten** war, ansonsten bleibt s unverändert)
$$add \leftarrow \begin{cases} true, & \text{falls ein Element eingefügt wurde} \\ false, & \text{falls } e \text{ schon in } b \text{ enthalten war} \end{cases}$$

Set – Interface

- **remove**(s, e)
 - **descr**: löscht ein Element aus dem Set
 - **pre**: $s \in S, e \in T\text{Elem}$
 - **post**: $s' \in S, s' = s \setminus \{e\}$ (Falls e nicht in s enthalten war, dann bleibt s unverändert)
$$\text{remove} \leftarrow \begin{cases} \text{true, falls ein Element entfernt wurde} \\ \text{false, fall } e \text{ nicht in } b \text{ enthalten war} \end{cases}$$
- **search**(s, e)
 - **descr**: sucht ob ein Element in dem Set enthalten ist
 - **pre**: $s \in S, e \in T\text{Elem}$
 - **post**: $\text{search} \leftarrow \begin{cases} \text{wahr, falls } e \in s \\ \text{falsch, ansonsten} \end{cases}$

Set – Interface

- **size(s)**
 - **descr:** gibt die Anzahl der Element aus dem Set zurück
 - **pre:** $s \in \mathcal{S}$
 - **post:** $\text{size} \leftarrow \text{Anzahl der Elemente aus } s$
- **isEmpty(s)**
 - **descr:** überprüft ob das Set leer ist
 - **pre:** $s \in \mathcal{S}$
 - **post:** $\text{isEmpty} \leftarrow \begin{cases} \text{wahr, falls } s \text{ keine Elemente enthält} \\ \text{falsch, ansonsten} \end{cases}$

Set – Interface

- `iterator(s, i)`
 - **descr**: gibt ein Iterator für ein Set zurück
 - **pre**: $s \in \mathcal{S}$
 - **post**: $i \in \mathcal{I}$, i ist ein Iterator für s
- `destroy(s)`
 - **descr**: zerstört ein Set
 - **pre**: $s \in \mathcal{S}$
 - **post**: s wurde zerstört

Set – Interface

- Andere mögliche Operationen (spezifisch für mathematische Mengen):
 - Vereinigung zweier Mengen
 - Durchschnitt zweier Mengen
 - Differenz zweier Mengen

ADT Set – Repräsentierung A

- **Ein dynamisches Array, wo die Elemente eindeutig sind**
- Zum Beispiel, wenn das Set folgende Elemente enthält: 1, 3, 2, 6, 5 dann kann man diese in einem Array folgendermaßen speichern:

1	3	2	6	5
----------	----------	----------	----------	----------

ADT Set – Repräsentierung B

- **Ein dynamisches Array von Boolean Werten (Bitarray)**
 - Man berechnet das aktuelle Werteintervall $[a, b]$
 - Das Werteintervall wird in das Intervall $[1, x]$ übersetzt, so dass $x = b - a + 1$
 - In dem Array speichert man:
 - Auf Position 1 True, falls der Wert a (das minimale Wert) zu dem Set gehört, False ansonsten
 - Auf Position 2 True, falls der Wert $a+1$ zu dem Set gehört, False ansonsten
 - ...
 - Auf Position $x-1$ True, falls der Wert $b-1$ zu dem Set gehört, False ansonsten
 - Auf Position x True, falls der Wert b zu dem Set gehört, False ansonsten
- Wenn ein Wert eingefügt wird, welcher sich außerhalb des Intervalls $[a, b]$ befindet, dann muss das Werteintervall und folglich auch das Bitarray aktualisiert werden

ADT Set – Repräsentierung B

- Nehmen wir als Beispiel ein Set, das folgende Elemente enthält:

5, 10, -1, 2, 3, -5

- Das Werteintervall $[-5, 10]$ wird in das Intervall $[1, 16]$ übersetzt, i.e. :
 - auf der Position 1 speichert man ob der Wert -5 (minimale Wert) zu dem Set gehört, auf der Position 2 ob der Wert -4 zu dem Set gehört, ..., auf der Position 16 ob der Wert 10 zu dem Set gehört
- Es wird in einem Boolean Array folgendermaßen gespeichert:

True	False	False	False	True	False	False	True	True	False	True	False	False	False	False	True
------	-------	-------	-------	------	-------	-------	------	------	-------	------	-------	-------	-------	-------	------

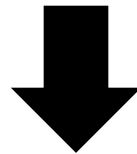
- Oder in einem Bitarray:

1	0	0	0	1	0	0	1	1	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ADT Set – Repräsentierung B

- Wenn man das Element -7 einfügt, dann ändert sich das Werteintervall: [-7,10]
- Das heißt Position 1 in dem Array entspricht jetzt dem Wert -7 (alle Werte müssen mit zwei Positionen nach rechts verschoben werden)
- Das Bitarray muss dann folgendermaßen geändert werden:

Bit	1	0	0	0	1	0	0	1	1	0	1	0	0	0	0	1
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Wert	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10



1	0	1	0	0	0	1	0	0	1	1	0	1	0	0	0	0	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10

SortedSet

- In einem Set können die Elemente basierend auf einer Ordnungsrelation sortiert werden → *SortedSet*
- In diesem Fall enthält das SortedSet Elemente vom Typ *TComp* anstatt Elemente vom Typ *TElem*
- Die einzigen Änderungen zu dem Interface sind bei der *init* Operation, wo man auch die Relation als Parameter hat
- Für einen sortierten Set muss der Iterator die Elemente in der Reihenfolge gegeben von der Relation durchlaufen

Set/SortedSet - Repräsentierung

- Um ADT Set (oder ADT SortedSet) zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
 - (dynamisches) Array:
 - Array von Elementen oder
 - Bitarrays
 - Verkettete Liste
 - Hashtabellen
 - (balancierte) Binärbäume – für sortierte Sets
 - Skip Listen – für sortierte Sets

Betrachte folgende Aufgabe:

- Wir haben einen Text und wollen das Wort finden, das am häufigsten in diesem Text vorkommt.
- Welche Eigenschaften sollte der Container haben?
 - Man braucht Paare der Form Schlüssel (Wort) – Wert (Anzahl der Vorkommen)
 - Die Schlüssel sollen eindeutig sein
 - Die Reihenfolge der Schlüssel ist nicht wichtig
- Ein Container, dessen Elemente Paare der Form $\langle key, value \rangle$ sind und wo die Schlüssel eindeutig sind, wobei die Reihenfolge der Schlüssel keine Rolle spielt ist **ADT Map** (Dictionary)

ADT Map

- Jeder Schlüssel hat genau einen zugehörigen Wert (falls es mehrere Werte gibt, redet man von *MultiMap*)
- Man kann auf den Wert nur mit Hilfe des Schlüssels zugreifen (es gibt keine Positionen in einer Map)
- Wenn man ein Map implementiert, dann braucht man eine Datenstruktur, in der es einfach ist einen Schlüssel zu finden
- Python: Dictionaries

ADT Map

- Domäne von ADT Map:

$\mathcal{M} = \{m \mid m \text{ ist eine Map mit Elementen } e = (k, v), \text{ wobei } k \in T\text{Key} \text{ und } v \in T\text{Value}\}$

Map – Interface

- **init(m)**
 - **descr:** erstellt eine leere Map
 - **pre:** wahr
 - **post:** $m \in \mathcal{M}$, m ist eine leere Map
- **destroy(m)**
 - **descr:** zerstört eine Map
 - **pre:** $m \in \mathcal{M}$
 - **post:** m wurde zerstört

Map – Interface

- **add(m, k, v)**
 - **descr:** fügt ein neues Schlüssel-Wert Paar zu der Map ein (die Operation kann auch *put* genannt werden)
 - **pre:** $m \in \mathcal{M}, k \in TKey, v \in TValue$
 - **post:** $m' \in \mathcal{M}, m' = m \cup \langle k, v \rangle$
- Was passiert wenn es schon ein Paar $\langle k, v \rangle$ gibt?
- Eine Möglichkeit wäre:
 - Falls der Schlüssel k schon in der Map enthalten ist, dann ersetzt man den Wert mit dem neuen Wert und der alte Wert wird zurückgegeben
 - Falls der Schlüssel k nicht in der Map enthalten ist, dann fügt man ein neues Paar ein und *NIL* wird zurückgegeben
 - $$\text{add} \leftarrow \begin{cases} v', & \text{falls } \exists \langle k, v' \rangle \in m \text{ und } m' \in \mathcal{M}, m' = m \setminus \langle k, v' \rangle \cup \langle k, v \rangle \\ o_{TValue}, & \text{falls der Schlüssel } k \text{ nicht in } m \text{ enthalten war} \end{cases}$$

Map – Interface

- `remove(m, k)`
 - **descr:** löscht ein Paar mit einem gegebenen Schlüssel aus der Map
 - **pre:** $m \in \mathcal{M}, k \in T\text{Key}$
 - **post:**

$$\text{remove} \leftarrow \begin{cases} v', \text{ falls } \exists \langle k, v' \rangle \in m \text{ und } m' \in \mathcal{M}, m' = m \setminus \langle k, v' \rangle \\ o_{T\text{Value}}, \text{ ansonsten} \end{cases}$$

Map – Interface

- `search(m, k)`
 - **descr:** such den Wert, der dem gegebenen Schlüssel entspricht
 - **pre:** $m \in \mathcal{M}, k \in TKey$
 - **post:**

$$\text{search} \leftarrow \begin{cases} v', & \text{falls } \exists \langle k, v' \rangle \in m \\ 0_{TV\text{value}}, & \text{ansonsten} \end{cases}$$

Map – Interface

- `iterator(m, it)`
 - **descr**: gibt ein Iterator für eine Map zurück
 - **pre**: $m \in \mathcal{M}$
 - **post**: $it \in \mathcal{I}$, it ist in Iterator für m
- `size(m)`
 - **descr**: gibt die Anzahl der Paare in der Map zurück
 - **pre**: $m \in \mathcal{M}$
 - **post**: $size \leftarrow$ Anzahl der Paare in m

Map – Interface

- **isEmpty(m)**
 - **descr:** überprüft ob die Map leer ist
 - **pre:** $m \in M$
 - **post:** $isEmpty \leftarrow \begin{cases} \text{wahr, falls } m \text{ keine Elemente enthält} \\ \text{falsch, ansonsten} \end{cases}$
- **keys(m, s)**
 - **descr:** gibt die Menge der Schlüssel aus der Map zurück
 - **pre:** $m \in M$
 - **post:** $s \in S$, s ist ein Set, der alle Schlüssel aus m enthält

Map – Interface

- `values(m, b)`
 - **descr**: gibt ein Bag von Werten aus der Map zurück
 - **pre**: $m \in \mathcal{M}$
 - **post**: $b \in \mathcal{B}$, b ist ein Bag, der alle Werte aus m enthält
- `pairs(m, s)`
 - **descr**: gibt die Menge der Paare aus der Map zurück
 - **pre**: $m \in \mathcal{M}$
 - **post**: $s \in \mathcal{S}$, s ist ein Set, der alle Paare aus m enthält

Sorted Map

- Man kann für die Schlüssel in der Map eine Ordnungsrelation definieren, dann benutzt man *TComp* anstatt *TKey*
- Die einzigen Änderungen zu dem Interface sind bei der *init* Operation, wo man auch die Relation als Parameter hat
- Für eine sortierte Map muss der Iterator die Paare in der Reihenfolge gegeben von der Relation durchlaufen. In diesem Fall geben die Operationen *keys* und *pairs* SortedSets zurück.

MultiMap / Sorted MultiMap

- Für einen Schlüssel kann es mehrere zugehörige Werte geben
- Operationen, die sich ändern:
 - Die Löschoperation braucht jetzt auch den Schlüssel und auch den Wert um das Paar löschen zu können
 - Wenn wir nach einem Schlüssel suchen, dann wird eine Liste von Werten zurückgegeben

Map/SortedMap - Repräsentierung

- Um ADT Map (oder ADT SortedMap) zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
 - (dynamisches) Array
 - Verkettete Liste
 - Hashtabellen
 - (balancierte) Binärbäume – für sortierte Maps
 - Skip Listen – für sortierte Maps

ADT Matrix

- Eine *Matrix* ist ein Container, der ein zweidimensionales Array enthält
- Jedes Element hat eine eindeutige Position, bestimmt von zwei Indexen: Zeile und Spalte
- Domäne $Mat = \{mat \mid mat \text{ ist eine Matrix mit Elementen vom Typ TElem}\}$
- Die Operationen einer Matrix unterscheiden sich von den Operationen der meisten Containers, da man in einer Matrix kein Element einfügen oder löschen kann. Man kann nur den Wert eines Elementes ändern.

ADT Matrix - Interface

- `init(mat, nrL, nrC)`
 - **descr:** erstellt eine neue Matrix mit nrL Zeilen und nrC Spalten
 - **pre:** $nrL \in \mathbb{N}^*$ und $nrC \in \mathbb{N}^*$
 - **post:** $mat \in MAT$, mat ist eine Matrix mit nrL Zeilen und nrC Spalten
 - **throws:** ein Exception falls nrL oder nrC negativ oder Null sind

ADT Matrix - Interface

- **nrLines(mat)**
 - **descr:** gibt die Anzahl der Zeilen aus der Matrix zurück
 - **pre:** $mat \in MAT$
 - **post:** $nrLines \leftarrow$ die Anzahl der Zeilen aus mat
- **nrColumns(mat)**
 - **descr:** gibt die Anzahl der Spalten aus der Matrix zurück
 - **pre:** $mat \in MAT$
 - **post:** $nrColumns \leftarrow$ die Anzahl der Spalten aus mat

ADT Matrix - Interface

- `element(mat, i, j)`
 - **descr:** gibt das Element von Zeile i und Spalte j zurück
 - **pre:** $mat \in MAT, 1 \leq i \leq nrLines, 1 \leq j \leq nrColumns$
 - **post:** $element \leftarrow$ das Element von Zeile i und Spalte j aus mat
 - **throws:** ein Exception falls die Position (i,j) nicht gültig ist

ADT Matrix - Interface

- `modify(mat, i, j, val)`
 - **descr:** ändert den Wert des Elementes von Zeile i und Spalte j
 - **pre:** $mat \in MAT, 1 \leq i \leq nrLines, 1 \leq j \leq nrColumns, val \in TElem$
 - **post:** der Wert von Position (i,j) wird auf val gesetzt und $modify \leftarrow$ gibt den alten Wert von Position (i,j) zurück
 - **throws:** ein Exception falls die Position (i,j) nicht gültig ist

Matrix – Operationen

- Andere mögliche Operationen:
 - Gebe die Position eines Elementes zurück
 - Erstelle einen Iterator, der die Elemente nach Spalten iteriert
 - Erstelle einen Iterator, der die Elemente nach Zeilen iteriert
 - Mathematische Operationen...
 - Usw.

Matrix – Repräsentierung

- Meistens benutzt man für eine Matrix eine sequentielle Repräsentierung (die Zeilen der Matrix werden im Speicherplatz auf aufeinanderfolgende Blöcke gespeichert)
- Falls die Matrix viele 0-Werte (oder 0_{TElem}) enthält, dann redet man von **schwachbesetzte** oder **dünnbesetzte** Matrix (**sparse**) → in diesem Fall ist es effizienter nur die Elemente unterschiedlich von 0 zu speichern

Schwachbesetzte Matrix – Beispiel

- Von den 20 Elemente sind nur 7 Elemente unterschiedlich von 0:

$$\begin{bmatrix} 0 & -2 & 0 & -7 & 0 \\ -6 & 0 & 0 & 0 & 0 \\ 0 & -9 & -8 & 0 & -5 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix}$$

Schwachbesetzte Matrix – Repräsentierung A

- Man kann **Tupeln der Form (Zeile, Spalte, Wert)** speichern, für die **Werte unterschiedlich von 0** (oder 0_{TElem})
- Für die Effizienz speichert man die Elemente lexikografisch sortiert nach (Zeile, Spalte)
- Die Tupel können in folgende Datenstrukturen gespeichert werden:
 - (dynamisches) Array
 - Verkettete Listen
 - (balancierte) Binärbäume

Schwachbesetzte Matrix – Operationen

- Die Operationen der schwachbesetzten Matrix sind gleich mit den Operationen der regulären Matrix.
- Die schwierigste Operation ist *modify*, weil man 4 Fälle unterscheiden muss basierend auf dem aktuellen Wert von Zeile i und Spalte j (*old_value*) und den neuen Wert (*new_value*):
 - $old_value = 0$ und $new_value = 0 \Rightarrow$ tu nichts
 - $old_value = 0$ und $new_value \neq 0 \Rightarrow$ füge einen neuen Tupel/Knoten ein mit *new_value*
 - $old_value \neq 0$ und $new_value = 0 \Rightarrow$ lösche den Tupel/Knoten mit *old_value*
 - $old_value \neq 0$ und $new_value \neq 0 \Rightarrow$ aktualisiere den Wert aus dem Tupel/Knoten mit *new_value*

Schwachbesetzte Matrix – Repräsentierung A Beispiel

$$\begin{bmatrix} 0 & -2 & 0 & -7 & 0 \\ -6 & 0 & 0 & 0 & 0 \\ 0 & -9 & -8 & 0 & -5 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix}$$

- Für das vorige Beispiel speichert man folgende Tupeln:

Zeile	1	1	2	3	3	3	4
Spalte	2	4	1	2	3	5	5
Wert	-2	-7	-6	-9	-8	-5	-2

- Man muss auch die Dimensionen der Matrix speichern (es kann sein, dass die letzten Zeilen oder Spalten nur 0-Werte enthalten)

Schwachbesetzte Matrix – Repräsentierung A

Zeile	1	1	2	3	3	3	4
Spalte	2	4	1	2	3	5	5
Wert	-2	-7	-6	-9	-8	-5	-2

- Ändern Sie den Wert von Position (1,4) auf 0

Zeile	1	2	3	3	3	4
Spalte	2	1	2	3	5	5
Wert	-2	-6	-9	-8	-5	-2

- Ändern Sie den Wert von Position (3,4) auf 5

Zeile	1	2	3	3	3	3	4
Spalte	2	1	2	3	4	5	5
Wert	-2	-6	-9	-8	5	-5	-2

Schwachbesetzte Matrix – Repräsentierung B

- In dem vorigen Beispiel merken wir, dass es viele aufeinanderfolgende Elemente gibt mit demselben Wert für die Zeile. Dieses Array kann folgenderweise komprimiert werden:
 - Man behaltet die Arrays *Spalte* und *Wert*
 - Für die Zeilen hat man ein Array mit $\text{nrRows}+1$ Elemente, in dem man an Position i die Position aus dem Array *Spalte* speichert, an der die Folge der Elemente von Zeile i beginnt
 - Also, die **Elemente von Zeile i** befinden sich in den Arrays *Spalte* und *Wert* zwischen den Positionen [**Zeile $[i]$** , **Zeile $[i+1]$**)
- Diese Repräsentierung heißt **Compressed Row Storage (CRS)** oder compressed sparse line representation
- Bem. Mit dieser Repräsentierung werden die Elemente zeilenweise gespeichert (erst Elemente der ersten Zeile, dann Elemente der zweiten Zeile, usw.)

Schwachbesetzte Matrix – Repräsentierung B Beispiel

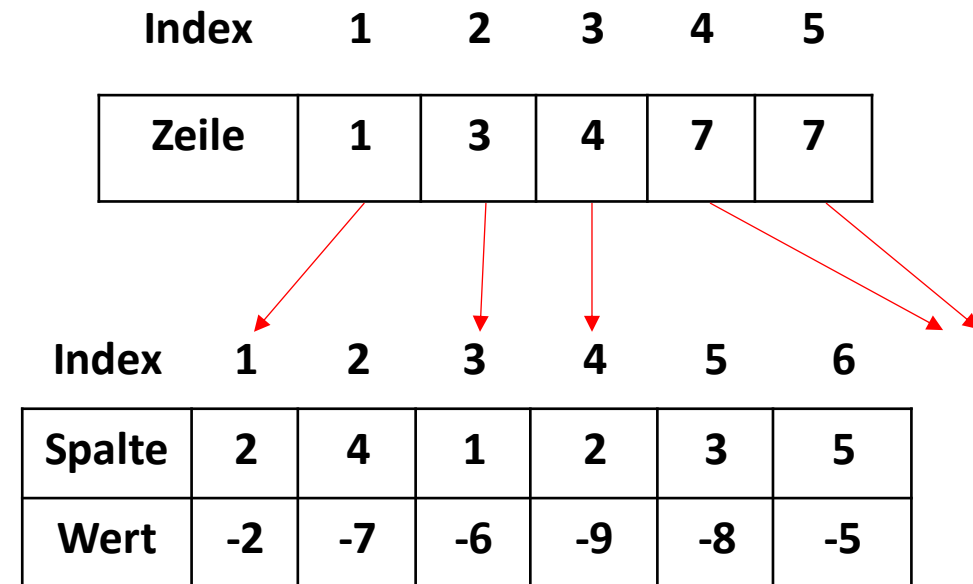
- Matrix:

$$\begin{bmatrix} 0 & -2 & 0 & -7 & 0 \\ -6 & 0 & 0 & 0 & 0 \\ 0 & -9 & -8 & 0 & -5 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- Man braucht folgende Arrays:

- Zeile – mit nrRows+1 Elemente
- Spalte und Wert – die Länge dieser Arrays wird von der Anzahl der Elementen unterschiedlich von 0 aus dem Matrix gegeben

- Repräsentierung:



Schwachbesetzte Matrix – Repräsentierung B Beispiel

- Ändern Sie den Wert von Position (3,3) auf 0
 - Wir suchen das Element von Position (3,3)
 - Elemente von Zeile 3 werden zwischen Position 4 und 6 gespeichert
 - Wir finden das gesuchte Element auf Position 5 in den Arrays Spalte und Wert
 - Auf 0 setzen heißt, dass es aus *Spalte* und *Wert* gelöscht wird
 - In dem Array Zeile werden nur die Werte entsprechend geändert

Index	1	2	3	4	5
Zeile	1	3	4	7	7

Index	1	2	3	4	5	6
Spalte	2	4	1	2	3	5
Wert	-2	-7	-6	-9	-8	-5

Index	1	2	3	4	5
Zeile	1	3	4	6	6

Index	1	2	3	4	5
Spalte	2	4	1	2	5
Wert	-2	-7	-6	-9	-5

Schwachbesetzte Matrix – Repräsentierung B Beispiel

- Ändern Sie den Wert von Position (2,2) auf 10
 - Wir suchen das Element von Position (2,2)
 - Elemente von Zeile 2 werden auf Position 3 gespeichert
 - Da wir Spalte 2 auf diese Position nicht finden, heißt es, dass das Element aktuell den Wert 0 hat
 - Auf 10 setzen heißt, dass man ein neues Element in *Spalte* und *Wert* einfügen muss
 - In dem Array *Zeile* werden nur die Werte entsprechend geändert

Index	1	2	3	4	5
Zeile	1	3	4	6	6

Index	1	2	3	4	5
Spalte	2	4	1	2	5
Wert	-2	-7	-6	-9	-5

Index	1	2	3	4	5
Zeile	1	3	5	7	7

Index	1	2	3	4	5	6
Spalte	2	4	1	2	2	5
Wert	-2	-7	-6	10	-9	-5

Schwachbesetzte Matrix – Repräsentierung B

- Die Bedeutung der drei Arrays: Spalte, Zeile, Wert ist also folgende:
 - Die Elemente von der Zeile i ($i = 1, 2, \dots, \text{nrRows}$) befinden sich auf den Spalten $\text{Spalte}[\text{Zeile}[i]]$, $\text{Spalte}[\text{Zeile}[i]+1]$, ... , $\text{Spalte}[\text{Zeile}[i+1]-1]$und haben die Werte
 $\text{Werte}[\text{Zeile}[i]]$, $\text{Werte}[\text{Zeile}[i]+1]$, ... , $\text{Werte}[\text{Zeile}[i+1]-1]$
- Falls eine Zeile i nur 0-Werte enthält, dann ist $\text{Zeile}[i] = \text{Zeile}[i+1]$, d.h. keine Werte in den Arrays *Spalte* und *Wert* entsprechen der Zeile i

Schwachbesetzte Matrix – Repräsentierung C

- Ähnlich kann man die Repräsentierung **Compressed Column Storage (CCS)** definieren
- Man benutzt drei Arrays: Zeile, Spalte, Wert mit der folgenden Bedeutung:
 - Die Elemente von der Spalte j ($j = 1, 2, \dots, \text{nrCol}$) befinden sich auf den Zeilen $\text{Zeile}[\text{Spalte}[j]]$, $\text{Zeile}[\text{Spalte}[j]+1]$, ... , $\text{Zeile}[\text{Spalte}[j+1]-1]$

und haben die Werte

$\text{Werte}[\text{Spalte}[j]]$, $\text{Werte}[\text{Spalte}[j]+1]$, ... , $\text{Werte}[\text{Spalte}[j+1]-1]$

Schwachbesetzte Matrix – Repräsentierung C Beispiel

- Matrix:

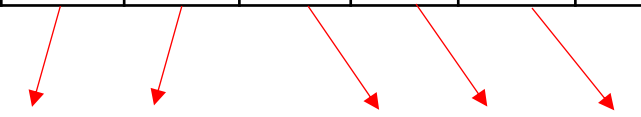
$$\begin{bmatrix} 0 & -2 & 0 & -7 & 0 \\ -6 & 0 & 0 & 0 & 0 \\ 0 & -9 & -8 & 0 & -5 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix}$$

- Man braucht folgende Arrays:

- Spalte – mit nrCol+1 Elemente
- Zeile und Wert – die Länge dieser Arrays wird von der Anzahl der Elementen unterschiedlich von 0 aus dem Matrix gegeben

- Repräsentierung:

Index	1	2	3	4	5	6
Spalte	1	2	4	5	6	8



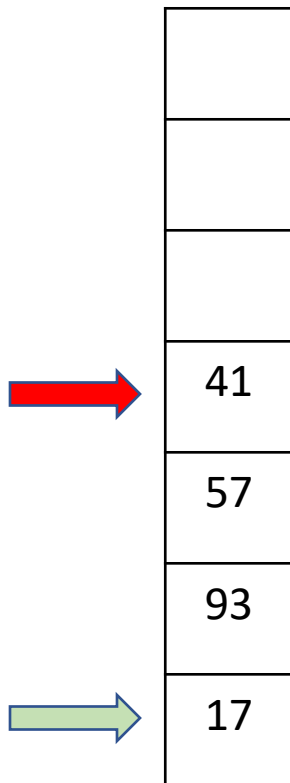
Index	1	2	3	4	5	6	7
Zeile	2	1	3	3	1	3	4
Wert	-6	-2	-9	-8	-7	-5	-2

ADT Queue (Schlange)

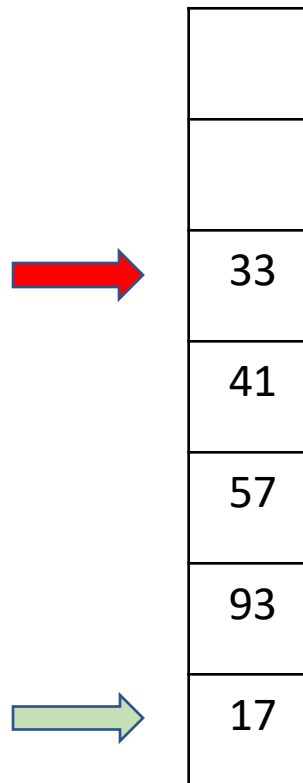
- ADT Schlange ist ein Container, in welchem die Elemente nur an einem Ende (dem Ende der Schlange) eingefügt (push/enqueue) und nur am anderen Ende (dem Kopf der Schlange) entfernt (pop/dequeue) werden können.
- Es gilt also das FIFO Prinzip (First-in-First-out)

ADT Queue - Beispiel

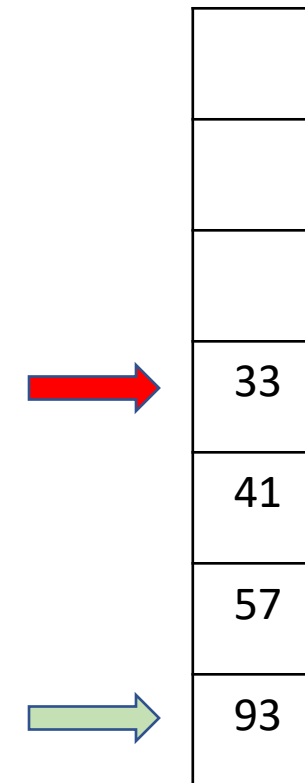
- Man fängt von der folgenden Schlange an (grüner Pfeil zeigt den Kopf, roter Pfeil zeigt das Ende)



- Man fügt den Wert 33 ein (push)

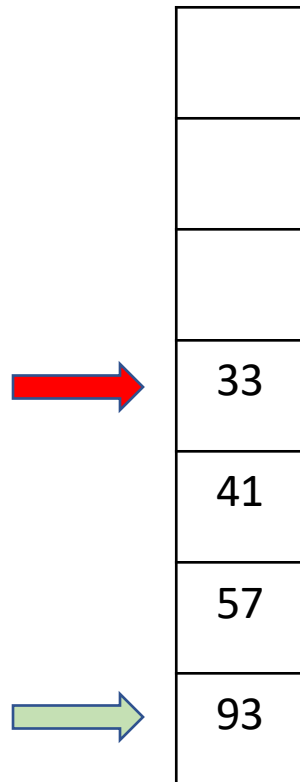


- Man entfernt ein Element (pop)

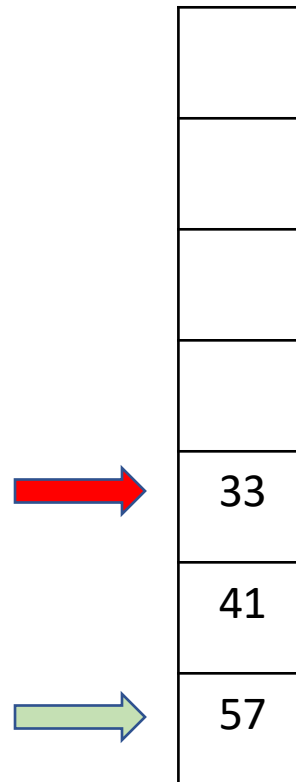


ADT Queue - Beispiel

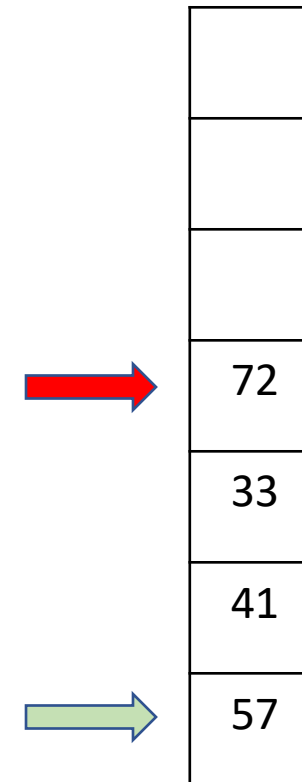
- Man fängt von der folgenden Schlange an (grüner Pfeil zeigt den Kopf, roter Pfeil zeigt das Ende)



- Man entfernt ein Element (pop)



- Man fügt 72 ein (push)



ADT Queue - Interface

- Domäne von ADT Schlange:

$$Q = \{q \mid q \text{ ist eine Schlange mit Elementen vom Typ } TElem\}$$

ADT Queue - Interface

- **init(q)**
 - **descr:** erstellt eine leere Schlange
 - **pre:** wahr
 - **post:** $q \in \mathcal{Q}$, q ist eine leere Schlange
- **destroy(q)**
 - **descr:** zerstört eine Schlange
 - **pre:** $q \in \mathcal{Q}$
 - **post:** q wurde zerstört

ADT Queue - Interface

- **push**(q , e)
 - **descr**: fügt ein neues Element am Ende der Schlange ein
 - **pre**: $q \in \mathcal{Q}$, $e \in TElem$
 - **post**: $q' \in \mathcal{Q}$, $q' = q \oplus e$, e ist das Element am Ende der Schlange
- **pop**(q)
 - **descr**: liefert das Element vom Anfang der Schlange und entfernt es von der Schlange
 - **pre**: $q \in \mathcal{Q}$
 - **post**: $pop \leftarrow e$, $e \in TElem$, e ist das Element am Anfang der Schlange q , $q' \in \mathcal{Q}$, $q' = q \ominus e$
 - **throws**: ein Underflow Error, falls die Schlange leer ist

ADT Queue - Interface

- **top(q)**
 - **descr:** liefert das Element vom Anfang der Schlange (die Schlange wird aber nicht geändert)
 - **pre:** $q \in Q$
 - **post:** $top \leftarrow e, e \in TElem, e$ ist das Element am Anfang der Schlange q
 - **throws:** ein Underflow Error, falls die Schlange leer ist
- **isEmpty(q)**
 - **descr:** überprüft ob die Schlange leer ist
 - **pre:** $q \in Q$
 - **post:** $isEmpty \leftarrow \begin{cases} \text{wahr, falls } q \text{ keine Elemente enthält} \\ \text{falsch, ansonsten} \end{cases}$

ADT Queue - Interface

- **Bemerkung!**

Die Schlange kann nicht iteriert werden!

Deshalb gibt es auch keine *iterator* Operation.

- Man kann nicht auf ein beliebiges Element zugreifen (z.B. das dritte Element), Zugriff ist nur auf das erste Element erlaubt.

ADT Queue – Repräsentierung

- Um ADT Schlange zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
 - Statische Arrays
 - Dynamische Arrays
 - Einfach verkettete Listen
 - Doppelt verkettete Listen
- Wo sollte man den Anfang und das Ende der Schlange für jede Repräsentierung speichern?

ADT Queue – Repräsentierung auf Arrays

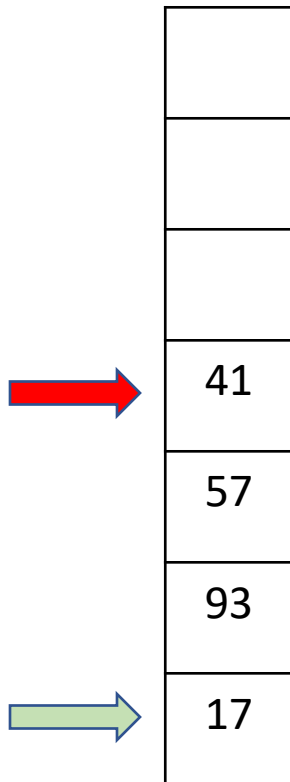
- Wo sollte man den Anfang und das Ende der Schlange in dem Array speichern?
- Theoretisch gibt es zwei Möglichkeiten:
 - Der Anfang der Schlange am Anfang des Arrays und das Ende der Schlange am Ende des Arrays speichern
 - Der Anfang der Schlange am Ende des Arrays und das Ende der Schlange am Anfang des Arrays speichern
- In beiden Fällen hat eine der Operationen *push* oder *pop* Komplexität $\Theta(n)$

ADT Queue – Repräsentierung auf Arrays

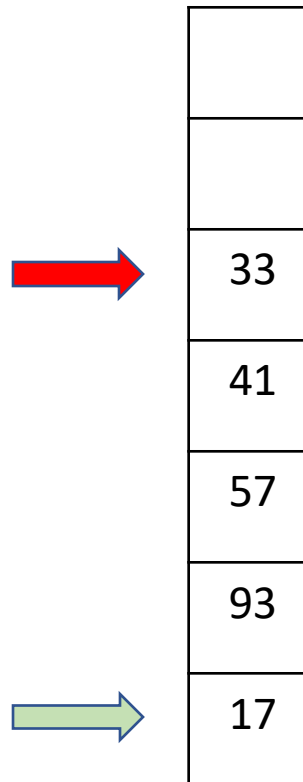
- Man kann die Komplexität der Operationen verbessern, wenn man nicht unbedingt den Anfang oder das Ende der Schlange auf Position 1 in dem Array speichert

ADT Queue – Repräsentierung auf Arrays

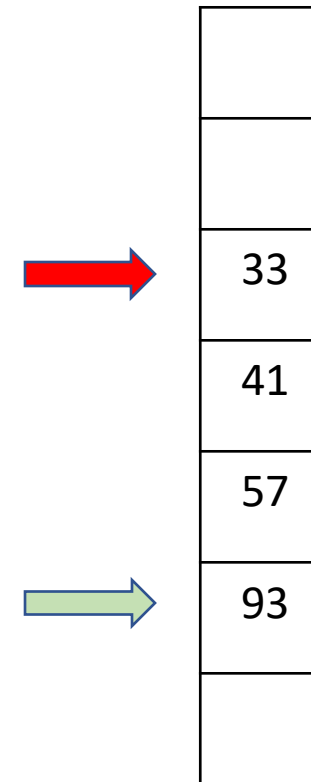
- Man fängt von der folgenden Schlange an (grüner Pfeil zeigt den Kopf, roter Pfeil zeigt das Ende)



- Man fügt den Wert 33 ein (push)

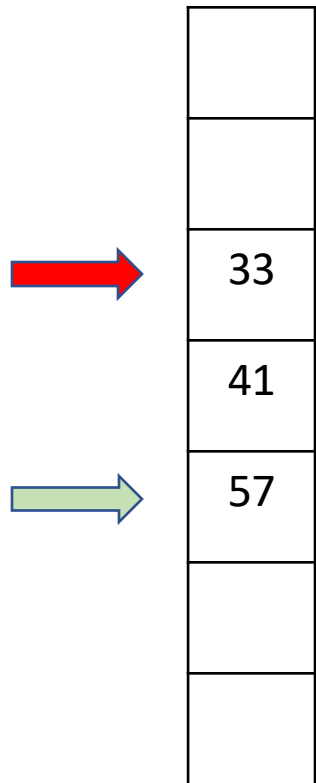


- Man entfernt ein Element (pop)

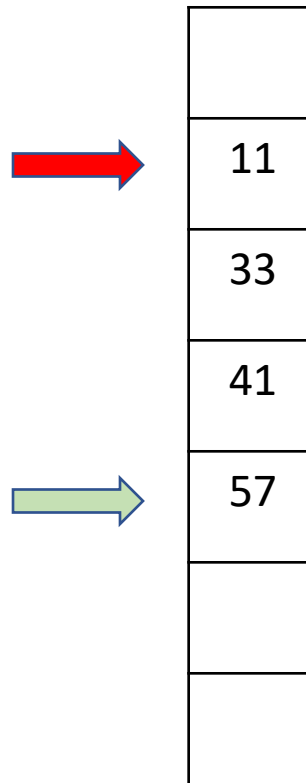


ADT Queue – Repräsentierung auf Arrays

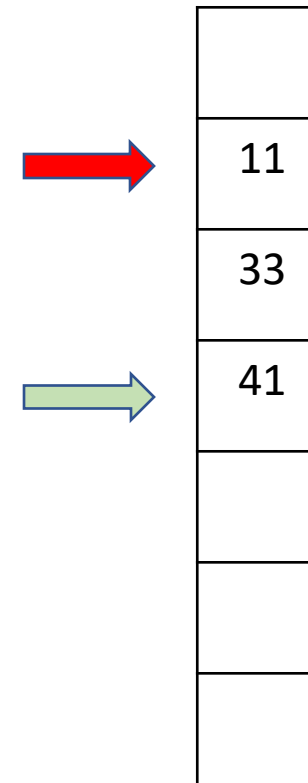
- Man entfernt ein neues Element



- Man fügt den Wert 11 ein (push)

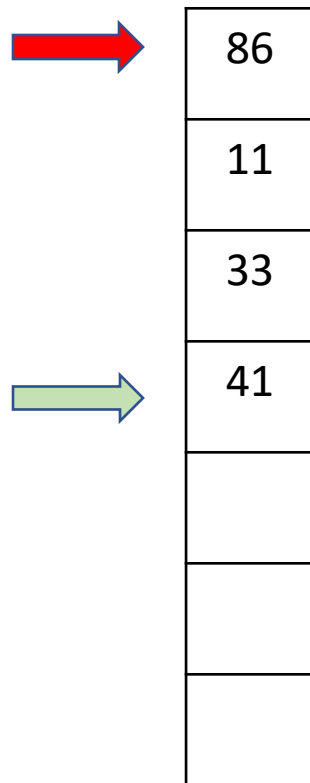


- Man entfernt ein Element (pop)

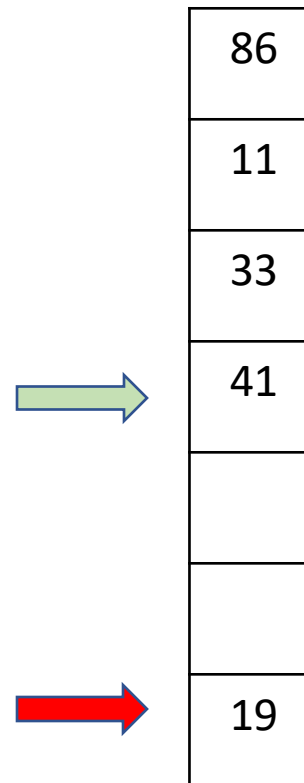


ADT Queue – Repräsentierung auf Arrays

- Man fügt 86 ein



- Man fügt 19 ein



Das wird als zirkuläres Array bezeichnet.

ADT Queue – Repräsentierung auf zirkulären Arrays

- Wie kann man eine Schlange auf einem zirkulären Array repräsentieren?

Queue:

capacity: Integer
front: Integer
rear: Integer
elems: TElem[]

- Manchmal wird auch die Länge der Schlange gespeichert

ADT Queue – Repräsentierung auf zirkulären Arrays - *init*

- Man benutzt den Wert -1 für *front* und *rear*, falls die Schlange leer ist

subalgorithm init(q) **is:**

q.capacity \leftarrow INIT_CAPACITY //eine Konstante

q.front \leftarrow -1

q.rear \leftarrow -1

@allokiere Speicherplatz für die Elemente des Arrays

end-subalgorithm

- Komplexität: $\Theta(1)$

ADT Queue – Repräsentierung auf zirkulären Arrays - *isEmpty*

- Wie überprüft man ob die Schlange leer ist?

```
function isEmpty(q) is:  
    if q.front = -1 then  
        isEmpty ← True  
    else  
        isEmpty ← False  
    end-if  
end-function
```

- Komplexität: $\Theta(1)$

ADT Queue – Repräsentierung auf zirkulären Arrays - *top*

- Was sollte die *top* Operation tun?

```
function top(q) is:  
    if q.front  $\neq$  -1 then  
        top  $\leftarrow$  q.elems[q.front]  
    else  
        @error – die Schlange ist leer  
    end-if  
end-function
```

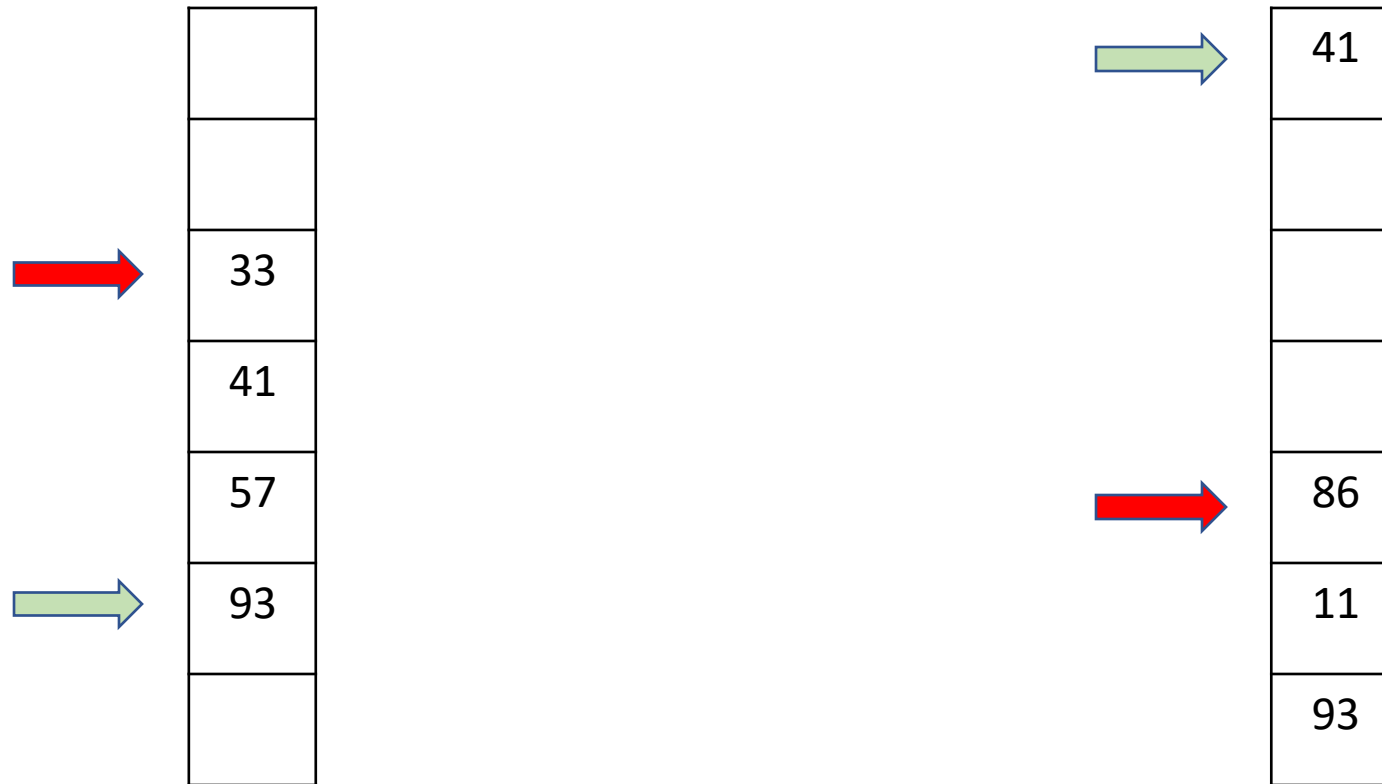
- Komplexität: $\Theta(1)$

ADT Queue – Repräsentierung auf zirkulären Arrays - *pop*

- Was sollte die *pop* Operation tun?

ADT Queue – Repräsentierung auf zirkulären Arrays - *pop*

- Es gibt zwei Situationen, die bei der Schlange vorkommen können:

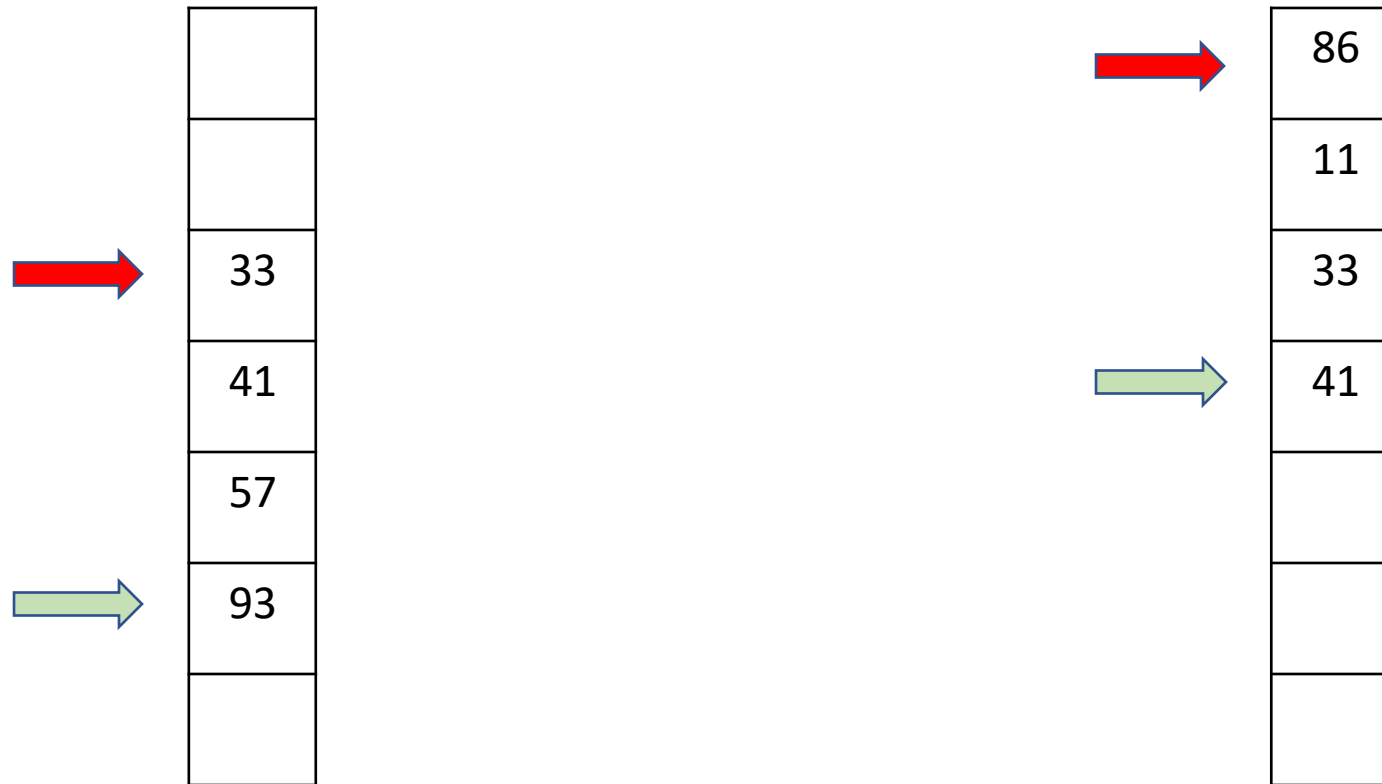


ADT Queue – Repräsentierung auf zirkulären Arrays - *push*

- Was sollte die *push* Operation tun?

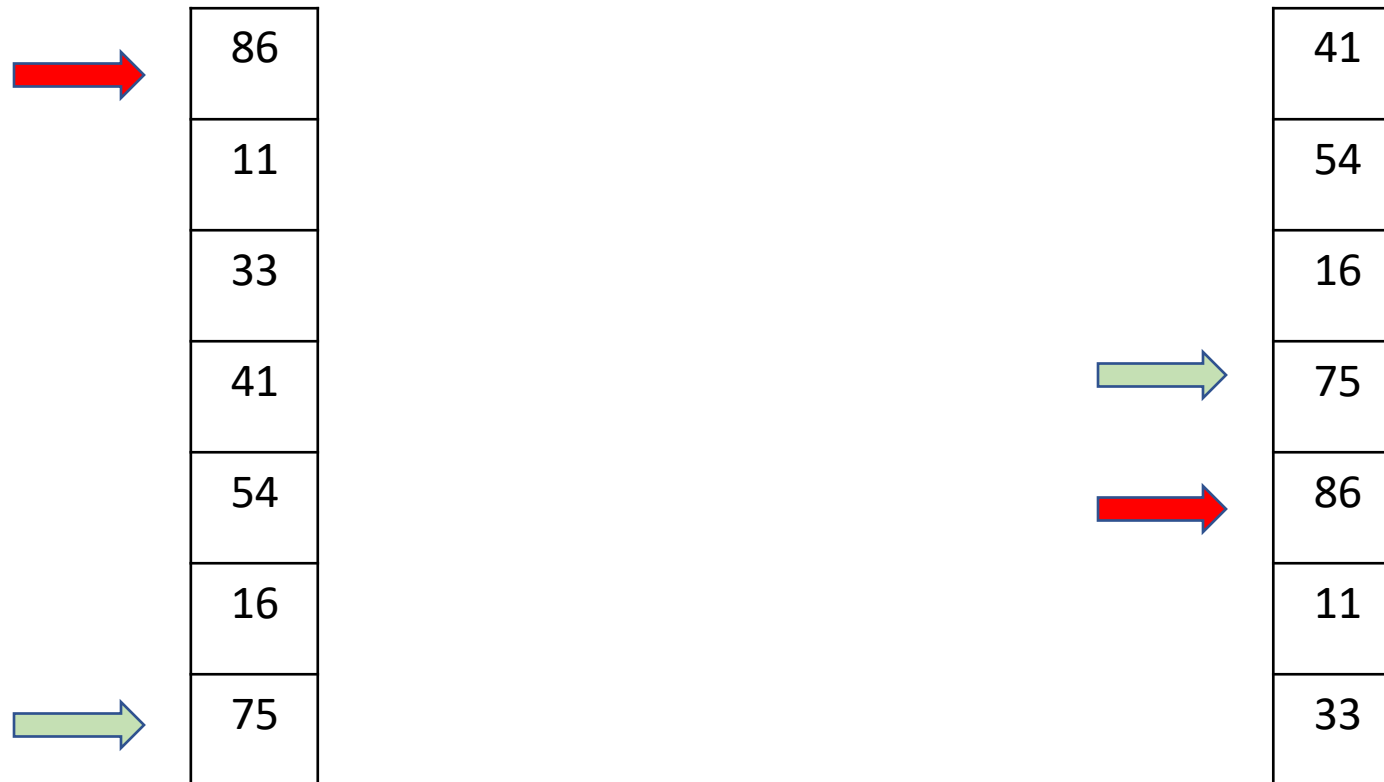
ADT Queue – Repräsentierung auf zirkulären Arrays - *push*

- Es gibt zwei Situationen, die bei der Schlange vorkommen können:



ADT Queue – Repräsentierung auf zirkulären Arrays - *push*

- Bei *push* muss man überprüfen ob die Schlange voll ist



- In beiden Fällen wurden die Elemente in folgender Reihenfolge eingefügt:
75, 16, 54, 41, 33, 11, 86

ADT Queue – Repräsentierung auf zirkulären Arrays - *push*

- Falls die Repräsentierung ein dynamisches Array benutzt, dann muss man die Kapazität vergrößern, falls die Schlange voll ist: es wird mehr Platz für das Array allokiert und die alten Elemente werden in dem neuen Array kopiert
- Wie kann man überprüfen ob die Schlange voll ist?
- Wenn die alten Elemente kopiert werden, dann können diese in dem Array reorganisiert werden