

# Definition Feld

## Definition Feld

Ein **Feld**  $F$  ist eine Folge von  $n$  Datenelementen  $(d_i)_{i=1,\dots,n}$ ,

$$F = d_1, d_2, \dots, d_n$$

mit  $n \in \mathbb{N}_0$ .

Die Datenelemente  $d_i$  sind beliebige Datentypen (z.B. primitive).

## Beispiele:

- ▶  $F$  sind die natürlichen Zahlen von 1 bis 10, aufsteigend geordnet:

$$F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

- ▶ Ist  $n = 0$ , so ist das Feld leer.

# Definition Feld

## Operationen

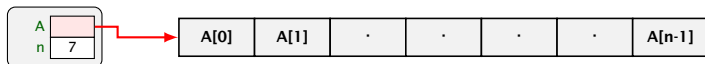
- ▶ `F.initialize()`: initialisiere leeres Feld `A`
- ▶ `F.elementAt(i)`: Zugriff auf `i`-tes Element von `A`.  
`i` muss zwischen `1` und `F.size()` liegen.
- ▶ `F.insert(d, i)`: füge Element `d` an Position `i` in Feld `A` ein.  
`i` muss zwischen `1` und `F.size()+1` liegen
- ▶ `F.erase(i)`: entferne `i`-tes Element aus Feld `A`.
- ▶ `F.size()`: gibt die Anzahl der Elemente des Feldes zurück

Ein abstrakter Datentyp wird im wesentlichen durch die auf ihn anwendbaren Operationen definiert.

# Feld als Array

## Repräsentation von Feld durch Array der Länge $n$

- ▶ Datenelemente werden in Array gespeichert
- ▶ einfacher Zugriff über index-operator ( $A[i]$ )
- ▶ Hinzufügen/Löschen schwierig...



**Achtung:** Indizierung des Arrays startet bei 0!

$A[i]$  enthält Element  $i+1$  des Feldes

# Eigenschaften von Arrays

Feld  $F$  mit Länge  $n$  als Array

## Vorteile:

- ▶ direkter Zugriff auf Elemente in konstanter Zeit mittels  $A[i]$
- ▶ sequentielles Durchlaufen sehr einfach

## Nachteile:

- ▶ Verlängern des Feldes aufwendig
- ▶ Hinzufügen und Löschen von Elementen aufwendig

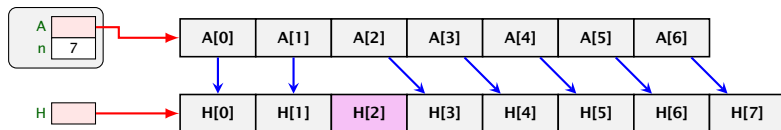
# Hinzufügen eines Elementes

**Gegeben:** Feld  $F$ , Länge  $n$ , via Array implementiert

**Gewünscht:** Feld  $F$ , zusätzliches Element  $d_i$  an Position  $i$ ;  
Elemente an Positionen  $\geq i$  werden auf nächsthöhere Position  
verschoben

- ▶ neuen Speicher der Größe  $n+1$  reservieren
- ▶ altes Array in neuen Speicher kopieren

**Beispiel:**  $F.insert(12, 3)$  (einfügen an Position 3)



## Implementierung: Feld via Array

```
3 class Feld {
4     int n;
5     int* A;
6
7     public:
8     Feld() {
9         n = 0;
10        A = new int[n];
11    }
```

## Implementierung: Feld via Array

```
13 void insert(int d, int i) {
14     int* H = new int[n+1];
15     for (int j=0; j<i-1; j++) {
16         H[j] = A[j];
17     }
18     H[i-1] = d;
19     for (int j=i-1; j<n; j++) {
20         H[j+1] = A[j];
21     }
22     delete[] A;
23     A = H;
24     n++;
25 }
```

## Implementierung: Feld via Array

```
27 void erase(int i) {
28     int* H = new int[n-1];
29     for (int j=0; j<i-1; j++) {
30         H[j] = A[j];
31     }
32     for (int j=i; j<n; j++) {
33         H[j-1] = A[j];
34     }
35     delete[] A;
36     A = H;
37     n--;
38 }
```



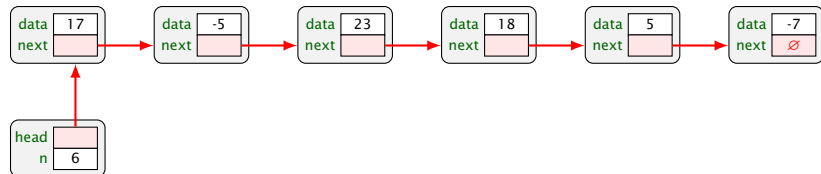
## Implementierung: Feld via Array

```
40     int elementAt(int i) {
41         return A[i-1];
42     }
43
44     int size() {
45         return n;
46     }
47 };
```

# Feld als einfach verkettete Liste

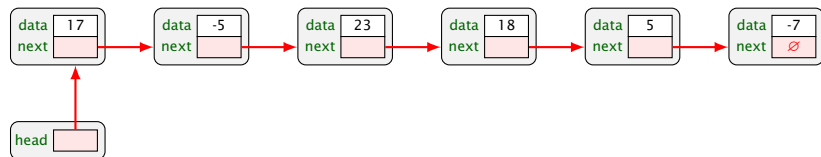
## Repräsentation von Feld als verkettete Liste

- ▶ dynamische Anzahl von Datenelementen
- ▶ in linearer Reihenfolge gespeichert (nicht notwendigerweise zusammenhängend!)
- ▶ mit Referenzen oder Zeigern verkettet



auf Englisch: *linked list*

# Verkettete Liste



Folge von miteinander verbundenen Elementen

jedes Element besteht aus

- ▶ **data**: Wert des Feldes an Position  $i$
- ▶ **next**: Referenz auf das nächste Element

**head** ist Referenz auf erstes Element der Liste

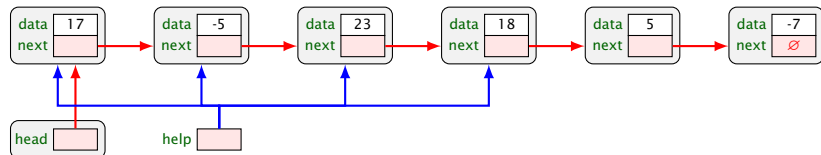
letztes Element hat keinen Nachfolger

- ▶ symbolisiert durch **null**-Referenz

# Verketteter Liste – Zugriff auf Element

## Zugriff auf Element $i$ :

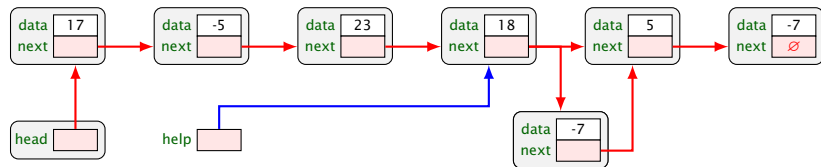
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum  $i$ -ten Element



# Verketteter Liste – Einfügen nach Referenz

## Einfügen nach Element i:

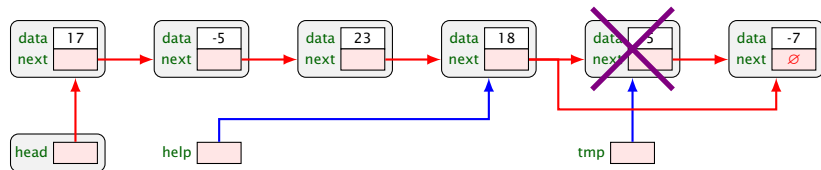
- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i-1**-ten Element
- ▶ Referenzen umsetzen



# Verketteter Liste – Löschen nach Referenz

## Einfügen nach Element i:

- ▶ beginne bei **head**-Referenz
- ▶ “vorhangeln” entlang **next**-Referenzen bis zum **i-1**-ten Element
- ▶ Referenzen umsetzen + Speicher freigeben



## Implementierung: Feld via Liste

```
4 struct Node {
5     int data;
6     Node *next;
7
8     Node(int d, Node* n) {
9         data = d;
10        next = n;
11    }
12 };
```

## Implementierung: Feld via Liste

```
14 class Feld {
15     int n;
16     Node *head;
17 public:
18
19     Feld() { // erzeuge leeres Feld
20         n = 0;
21         head = NULL;
22     }
23
24     int size() { return n; }
25
26     int elementAt(int i) {
27         Node* h = head;
28         while (i-- > 1)
29             h = h->next;
30         return h->data;
31     }
```



## Implementierung: Feld via Liste

```
33     void insert(int d, int i) {
34         Node* tmp = new Node(d, NULL);
35         n++;
36         if (i == 1) {
37             tmp->next = head;
38             head      = tmp;
39             return;
40         }
41         Node* h = head;
42         i--;
43         while (i-- > 1)
44             h = h->next;
45         tmp->next = h->next;
46         h->next  = tmp;
47     }
```

## Implementierung: Feld via Liste

```
49     void erase(int i) {
50         n--;
51         if (i == 1) {
52             Node* tmp = head;
53             head = head->next;
54             delete tmp;
55             return;
56         }
57         Node* h = head;
58         i--;
59         while (i-- > 1)
60             h = h->next;
61         Node* tmp = h->next;
62         h->next = h->next->next;
63         delete tmp;
64     }
65 }; // end class
```

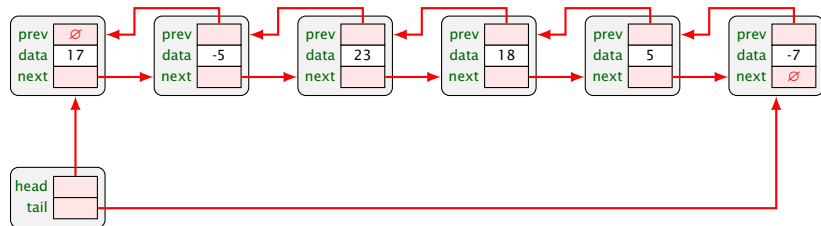
# Gegenüberstellung Array und verkettete Liste

Array	Verkettete Liste
⊕ Direkter Zugriff auf i-tes Element	⊖ Zugriff auf i-tes Element erfordert i Iterationen
⊕ sequentielles Durchlaufen sehr einfach	⊕ sequentielles Durchlaufen sehr einfach
⊖ statische Länge, kann Speicher verschwenden	⊕ dynamische Länge
	⊖ zusätzlicher Speicher für Zeiger benötigt
⊖ Einfügen/Löschen erfordert erheblich Kopieraufwand	⊕ Einfügen/Löschen einfach

# Feld als doppelt verkettete Liste

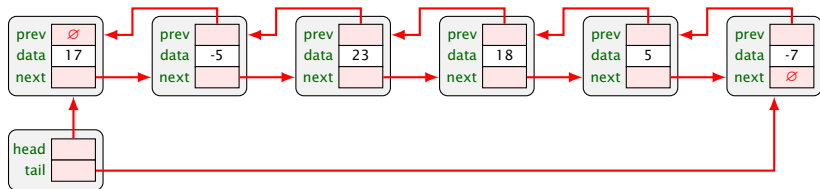
## Repräsentation von Feld A als doppelt verkettete Liste

- ▶ verkettete Liste
- ▶ jedes Element mit Referenzen **doppelt** verkettet



auf Englisch: *doubly linked list*

# Doppelt verkettete Liste



Folge von miteinander verbundenen Elementen

Jedes Element besteht aus

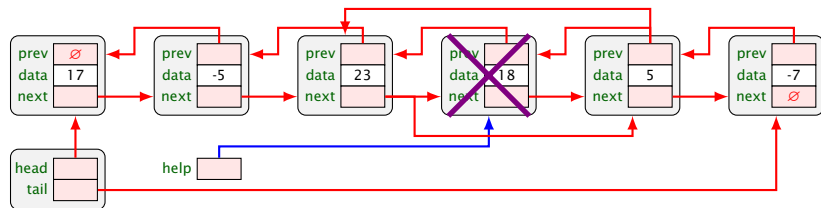
- ▶ **data**: Wert des Feldes
- ▶ **next**: Referenz auf das nächste Element
- ▶ **prev**: Referenz auf das vorherige Element

**head/tail** sind Referenzen auf erstes/letztes Element; diese haben keinen Nachfolger bzw. keinen Vorgänger.

# Operationen auf doppelt verketteter Liste

## Löschen von Element i:

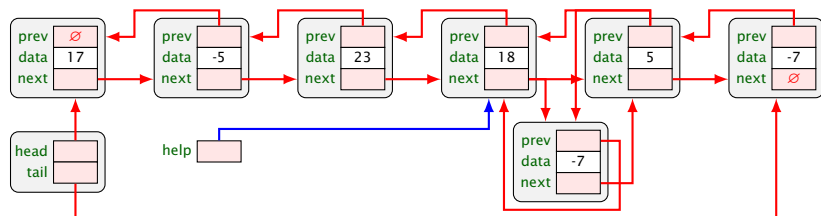
- ▶ Zugriff auf Element  $i$
- ▶ umhängen von Referenzen
- ▶ Speicherplatz freigeben



# Operationen auf doppelt verketteter Liste

## Einfügen von Element an Stelle $i$ :

- ▶ Zugriff auf Element  $i-1$
- ▶ umhängen von Referenzen



# Eigenschaften doppelt verkettete Liste

## Doppelt verkettete Liste

### Vorteile:

- ▶ Durchlauf in beiden Richtungen möglich
- ▶ Einfügen/Löschen potentiell einfacher, da man sich Vorgänger nicht extra merken muss

### Nachteile:

- ▶ zusätzlicher Speicher erforderlich für zwei Referenzen
- ▶ Referenzverwaltung komplizierter und fehleranfällig



# Zusammenfassung Felder

Ein **Feld** A kann repräsentiert werden als:

- ▶ **Array**
- ▶ **verkettete Liste** (linked list)
- ▶ **doppelt verkettete Liste** (doubly linked list)

**Eigenschaften:**

- ▶ einfach und flexibel
- ▶ aber manche Operationen aufwendig

# Definition Abstrakter Datentyp

**Abstrakter Datentyp (englisch: abstract data type, ADT)** Ein **abstrakter Datentyp** ist ein mathematisches **Modell** für bestimmte Datenstrukturen mit vergleichbarem Verhalten.

Ein abstrakter Datentyp wird **indirekt** definiert über

- ▶ mögliche **Operationen** auf ihm sowie
- ▶ mathematische Bedingungen (oder: constraints) über die **Auswirkungen der Operationen** (u.U. auch die Kosten der Operationen).

# Beispiel abstrakter Datentyp: abstrakte Variable

Abstrakte Variable  $V$  ist eine veränderliche Dateneinheit mit zwei Operationen

- ▶  $\text{load}(V)$  liefert einen Wert
- ▶  $\text{store}(V, x)$  wobei  $x$  ein Wert ist

und der Bedingung

- ▶  $\text{load}(V)$  liefert immer den Wert  $x$  der letzten Operation  $\text{store}(V, x)$

# Beispiel abstrakter Datentyp: abstrakte Liste (Teil 1)

Abstrakte Liste  $L$  ist ein Datentyp

mit Operationen

- ▶  $\text{pushFront}(L, x)$  liefert eine Liste
- ▶  $\text{front}(L)$  liefert ein Element
- ▶  $\text{rest}(L)$  liefert eine Liste

und den Bedingungen

- ▶ ist  $x$  Element,  $L$  Liste, dann liefert  $\text{front}(\text{pushFront}(L, x))$  das Element  $x$ .
- ▶ ist  $x$  Element,  $L$  Liste, dann liefert  $\text{rest}(\text{pushFront}(L, x))$  die Liste  $L$ .

## Beispiel abstrakter Datentyp: abstrakte Liste (Teil 2)

Abstrakte Liste  $L$ . Weitere Operationen sind

- ▶ `isEmpty(L)` liefert `true` oder `false`
- ▶ `initialize()` liefert eine Listeninstanz

mit den Bedingungen

- ▶ `initialize() ≠ L` für jede Liste  $L$  (d.h. jede neue Liste ist separat von alten Listen)
- ▶ `isEmpty(initialize()) == true` (d.h. eine neue Liste ist leer)
- ▶ `isEmpty(pushFront(L, x)) == false` (d.h. eine Liste ist nach einem `pushFront` nicht leer)

# Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein **Stack** ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem **Last In – First Out (LIFO)** Prinzip mit den Eigenschaften

- ▶ löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ nur das **letzte Element** darf manipuliert werden.

## Operationen auf Stacks:

- ▶ **push**: legt ein Element auf den Stack (einfügen)
- ▶ **pop**: entfernt das letzte Element vom Stack (löschen)
- ▶ **top**: liefert das letzte Stack-Element
- ▶ **isEmpty**: liefert **true** falls Stack leer
- ▶ **initialize**: Stack erzeugen und in Anfangszustand (leer) setzen

## Definition Stack

**Stack (oder deutsch: Stapel, Keller)** Ein **Stack** ist ein abstrakter Datentyp. Er beschreibt eine spezielle Listenstruktur nach dem **Last In – First Out (LIFO)** Prinzip mit den Eigenschaften

- ▶ löschen, einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ nur das **letzte Element** darf manipuliert werden.

## Definition Stack (exakter)

Stack  $S$  ist ein abstrakter Datentyp mit Operationen

- ▶  $\text{pop}(S)$  liefert einen Wert
- ▶  $\text{push}(S, x)$  wobei  $x$  ein Wert

mit der Bedingung

- ▶ ist  $x$  Wert und  $V$  Variable, dann ist die Sequenz  $\text{push}(S, x); V = \text{pop}(S);$  äquivalent zu  $V = x;$

sowie der Operation

- ▶  $\text{top}(S)$  liefert einen Wert

mit der Bedingung

- ▶ ist  $x$  Wert und  $V$  Variable, dann ist die Sequenz  $\text{push}(S, x); V = \text{top}(S);$  äquivalent zu  $\text{push}(S, x); V = x;$



## Definition Stack (exakter, Teil 2)

Stack  $S$ . Weitere Operationen sind

- ▶ `isEmpty(S)` liefert `true` oder `false`
- ▶ `initialize()` liefert eine Stackinstanz

mit den Bedingungen

- ▶ `initialize() ≠ S` für jeden Stack  $S$  (d.h. jeder neue Stack ist separat von alten Stacks)
- ▶ `isEmpty(initialize()) == true` (d.h. ein neuer Stack ist leer)
- ▶ `isEmpty(push(S, x)) == false` (d.h. ein Stack nach push ist nicht leer)

# Anwendungsbeispiele Stack

Call-Stack bei Funktionsaufrufen

Einfache Vorwärts- / Rückwärts Funktion in Software

- ▶ z.B. im Internet-Browser

Syntaxanalyse eines Programms

- ▶ z.B. zur Erkennung von Syntax-Fehlern durch Compiler

Auswertung arithmetischer Ausdrücke

# Auswertung arithmetischer Ausdrücke

Gegeben sei ein vollständig geklammerter, einfacher arithmetischer Ausdruck mit Bestandteilen Zahl, +, \*, =

**Beispiel:**  $(3 * (4 + 5)) =$

## Schema:

- ▶ arbeite Ausdruck von links nach rechts ab, speichere jedes Zeichen ausser ) und = in Stack S
- ▶ bei ) werte die 3 obersten Elemente von S aus, dann entferne die passende Klammer ( vom Stack S und speichere Ergebnis in Stack S
- ▶ bei = steht das Ergebnis im obersten Stack-Element von S

# Beispiel: Auswertung arithmetischer Ausdrücke

39
3
-
14
11
(
+
7
.
4
1
+
3
(
28
(
4.3 Stacks
(

# Implementation Stack

## Stack ist abstrakter Datentyp.

- ▶ Implementation ist nicht festgelegt
- ▶ nur Operationen und Bedingungen sind festgelegt

Stack kann auf viele Arten implementiert werden, zum Beispiel als:

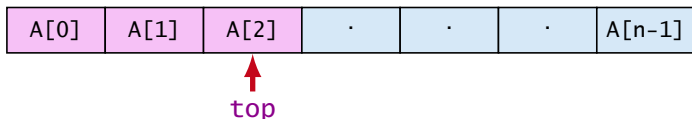
- ▶ Array
- ▶ verkettete Liste

# Implementation Stack via Array

Stack-Elemente im Array (Länge  $n$ ) speichern

oberstes Stack-Element merken mittels Variable  $top$

falls Stack **leer** ist  $top == -1$

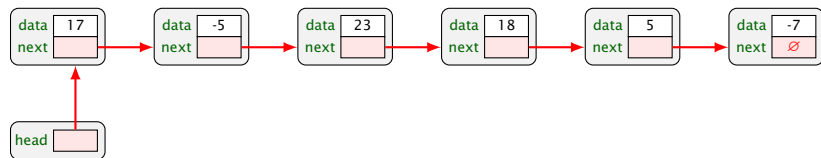


- ▶  $push(x)$  inkrementiert  $top$  und speichert  $x$  in  $A[top]$
- ▶  $pop()$  liefert  $A[top]$  zurück und dekrementiert  $top$
- ▶  $top()$  liefert  $A[top]$  zurück

# Implementation Stack als verkettete Liste

Stack-Elemente speichern in verketteter Liste

oberstes Stack-Element wird durch **head**-Referenz markiert



- ▶ **push(x)** fügt Element an erster Position ein
- ▶ **pop()** liefert Element an erster Position zurück und entfernt es
- ▶ **top()** liefert Element an erster Position zurück

# Zusammenfassung Stack

Stack ist **abstrakter Datentyp** als Metapher für einen Stapel

- ▶ wesentliche Operationen: **push, pop**

Implementation als **Array**

- ▶ fixe Größe (entweder Speicher verschwendet oder zu klein)
- ▶ push, pop sehr effizient

Implementation als **verkettete Liste**

- ▶ dynamische Größe, aber Platz für Zeiger “verschwendet”
- ▶ push, pop effizient
- ▶ eventuell nicht cache-effizient

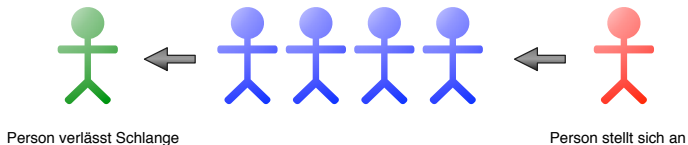


# Definition Queue

## Queue (oder deutsch: Warteschlange)

Eine **Queue** ist ein abstrakter Datentyp. Sie beschreibt eine spezielle Listenstruktur nach dem **First In – First Out (FIFO)** Prinzip mit den Eigenschaften

- ▶ einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ entfernen ist nur **am Anfang** der Liste erlaubt.



# Definition Queue

## Queue (oder deutsch: Warteschlange)

Eine **Queue** ist ein abstrakter Datentyp. Sie beschreibt eine spezielle Listenstruktur nach dem **First In – First Out (FIFO)** Prinzip mit den Eigenschaften

- ▶ einfügen ist nur **am Ende** der Liste erlaubt,
- ▶ entfernen ist nur **am Anfang** der Liste erlaubt.

## Operationen auf Queues:

- ▶ **enqueue**: fügt ein Element am Ende der Schlange hinzu
- ▶ **dequeue**: entfernt das erste Element der Schlange
- ▶ **isEmpty**: liefert **true** falls Queue leer
- ▶ **initialize**: Queue erzeugen und in Anfangszustand (leer) setzen

## Definition Queue (exakter)

Queue  $Q$  ist ein abstrakter Datentyp mit Operationen

- ▶ `dequeue(Q)` liefert einen Wert
- ▶ `enqueue(Q, x)` wobei  $x$  ein Wert
- ▶ `isEmpty(Q)` liefert `true` oder `false`
- ▶ `initialize` liefert eine Queue Instanz

und mit Bedingungen

- ▶ ist  $x$  Wert,  $V$  Variable,  $Q$  leere Queue, dann ist Sequenz `enqueue(Q, x); V=dequeue(Q);` äquivalent zu `V=x.`
- ▶ sind  $x, y$  Werte,  $V$  Variable und  $Q$  Queue, dann ist Sequenz `enqueue(Q, x); enqueue(Q, y); V=dequeue(Q)` äquivalent zu `enqueue(Q, x); V=dequeue(Q); enqueue(Q, y)`
- ▶ `initialize() ≠ Q` für jede Queue  $Q$
- ▶ `isEmpty(initialize()) == true`
- ▶ `isEmpty(enqueue(Q, x)) == false`

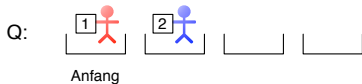
# Beispiel: Queue



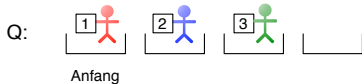
`Q = initialize();`



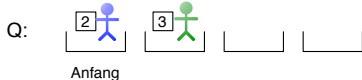
`enqueue(1);`



`enqueue(2);`



`enqueue(3);`



`dequeue();`



`dequeue();`

# Anwendungsbeispiele Queue

- ▶ Druckerwarteschlange
- ▶ Playlist von iTunes (oder ähnlichem Musikprogramm)
- ▶ Kundenaufträge bei Webshops
- ▶ Warteschlange für Prozesse im Betriebssystem (Multitasking)

# Anwendungsbeispiel Stack und Queue

## Palindrom

Ein Palindrom ist eine Zeichenkette, die von vorn und von hinten gelesen gleich bleibt.

**Beispiel:** Reittier

Erkennung ob Zeichenkette ein Palindrom ist

- ▶ ein **Stack** kann die Reihenfolge der Zeichen umkehren
- ▶ eine **Queue** behält die Reihenfolge der Zeichen

# Palindromerkennung

```
1 Input: Zeichenkette str mit Laenge n
2 Output: true falls str Palindrom; sonst false
3
4 Stack S;
5 Queue Q;
6
7 i = 0;
8 while (i<n)
9     S.push(str[i]);
10    Q.enqueue(str[i]);
11    i++;
12 i = 0;
13 while (i<n)
14     s = S.pop();
15     q = Q.dequeue();
16     if (s != q) return false;
17     i++;
18 return true;
```

# Implementation Queue

Auch Queue ist abstrakter Datentyp.

- ▶ Implementation ist nicht festgelegt
- ▶ nur Operationen und Bedingungen sind festgelegt

Queue kann auf viele Arten implementiert werden, zum Beispiel als:

- ▶ verkettete Liste
- ▶ Array

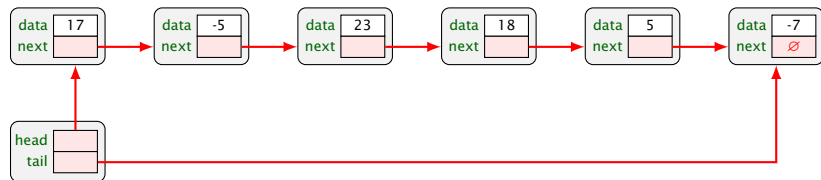


# Implementation Queue als verkettete Liste

Queue-Elemente speichern in verketteter Liste

Anfang der Queue wird durch **head**-Referenz markiert

Ende der Queue wird durch extra **tail**-Referenz markiert



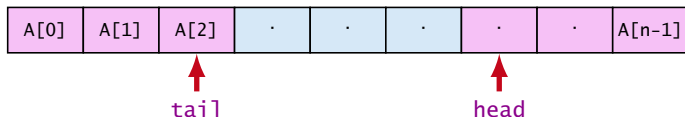
- ▶ **enqueue(x)** fügt Element bei **head**-Referenz ein
- ▶ **dequeue()** liefert Element bei **tail**-Referenz zurück und entfernt es

# Implementation Queue via Array

Queueelemente in Array (Länge  $n$ ) speichern

Anfang der Queue wird durch Index  $head$  markiert

Ende der Queue wird durch Index  $tail$  markiert



- ▶  $enqueue(x)$  fügt Element bei Index  $(tail+1)\%n$  ein
- ▶  $dequeue$  liefert Element bei Index  $head$  zurück und entfernt es durch Inkrement von  $head$  ( $head=(head+1)\%n$ )

## Implementation Queue als zwei Stacks

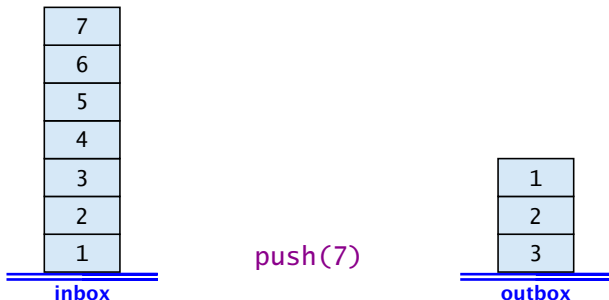
Queue  $Q$  kann mittels zwei Stacks implementiert werden

erster Stack  $inbox$  wird für  $enqueue$  benutzt:

- ▶  $Q.enqueue(x)$  resultiert in  $inbox.push(x)$

zweiter Stack  $outbox$  wird für  $dequeue$  benutzt:

- ▶ falls  $outbox$  leer, kopiere alle Elemente von  $inbox$  zu  $outbox$ :  $outbox.push( inbox.pop() )$
- ▶  $Q.dequeue()$  liefert  $outbox.pop()$  zurück



# Zusammenfassung Queue

Queue ist abstrakter Datentyp als Metapher für eine Warteschlange

- ▶ wesentliche Operationen: **enqueue, dequeue**

Implementation als verkettete Liste

- ▶ dynamische Größe, aber Platz für Referenzen “verschwendet”
- ▶ enqueue, dequeue effizient
- ▶ nicht cache-effizient

Implementation als Array

- ▶ fixe Größe (entweder Speicher verschwendet oder zu klein)
- ▶ enqueue, dequeue sehr effizient