

8. Elementare Datenstrukturen

Abstrakte Datentypen Stapel, Warteschlange,
Implementationsvarianten der verketteten Liste, [Ottman/Widmayer,
Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2]

Abstrakte Datentypen

Wir erinnern uns⁵ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

⁵hoffentlich

Abstrakte Datentypen

Wir erinnern uns⁵ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**(x, S): Legt Element x auf den Stapel S .

⁵hoffentlich

Abstrakte Datentypen

Wir erinnern uns⁵ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**(x, S): Legt Element x auf den Stapel S .
- **pop**(S): Entfernt und liefert oberstes Element von S , oder **null**.

⁵hoffentlich

Abstrakte Datentypen

Wir erinnern uns⁵ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**(x, S): Legt Element x auf den Stapel S .
- **pop**(S): Entfernt und liefert oberstes Element von S , oder **null**.
- **top**(S): Liefert oberstes Element von S , oder **null**.

⁵hoffentlich

Abstrakte Datentypen

Wir erinnern uns⁵ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**(x, S): Legt Element x auf den Stapel S .
- **pop**(S): Entfernt und liefert oberstes Element von S , oder **null**.
- **top**(S): Liefert oberstes Element von S , oder **null**.
- **isEmpty**(S): Liefert **true** wenn Stack leer, sonst **false**.

⁵hoffentlich

Abstrakte Datentypen

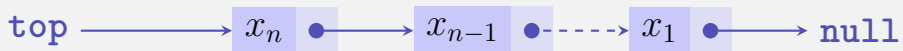
Wir erinnern uns⁵ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- **push**(x, S): Legt Element x auf den Stapel S .
- **pop**(S): Entfernt und liefert oberstes Element von S , oder **null**.
- **top**(S): Liefert oberstes Element von S , oder **null**.
- **isEmpty**(S): Liefert **true** wenn Stack leer, sonst **false**.
- **emptyStack**(): Liefert einen leeren Stack.

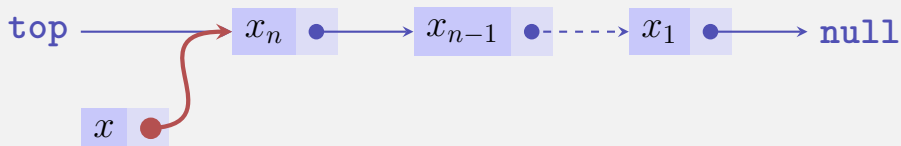
⁵hoffentlich

Implementation Push



`push(x, S):`

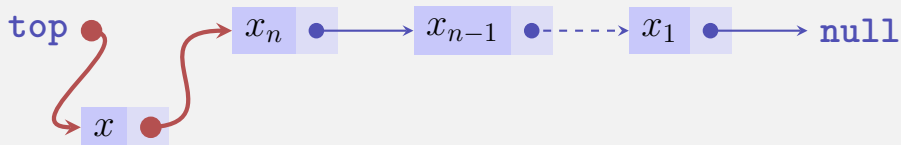
Implementation Push



`push`(x, S):

- 1 Erzeuge neues Listenelement mit x und Zeiger auf den Wert von `top`.

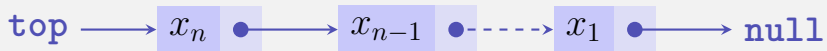
Implementation Push



`push`(x, S):

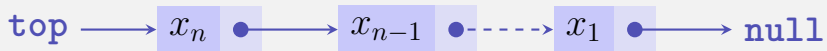
- 1 Erzeuge neues Listenelement mit x und Zeiger auf den Wert von `top`.
- 2 Setze `top` auf den Knoten mit x .

Implementation Pop



`pop(S)`:

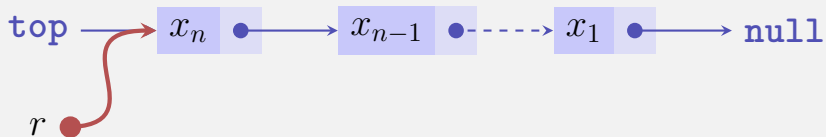
Implementation Pop



`pop(S)`:

- 1 Ist `top=null`, dann gib `null` zurück

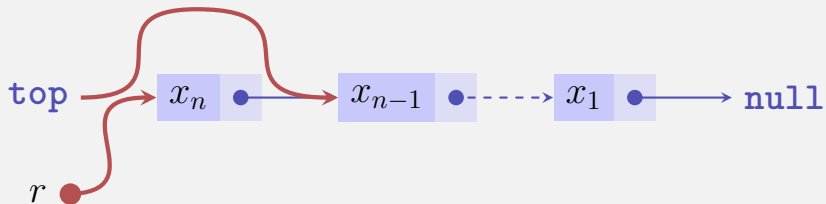
Implementation Pop



$\text{pop}(S)$:

- 1 Ist $\text{top}=\text{null}$, dann gib null zurück
- 2 Andernfalls merke Zeiger p von top in r .

Implementation Pop



`pop(S)`:

- 1 Ist `top=null`, dann gib `null` zurück
- 2 Andernfalls merke Zeiger p von `top` in r .
- 3 Setze `top` auf $p.next$ und gib r zurück

Analyse

Jede der Operationen `push`, `pop`, `top` und `isEmpty` auf dem Stack ist in $\mathcal{O}(1)$ Schritten ausführbar.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **enqueue**(x, Q): fügt x am Ende der Schlange an.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)
- **head**(Q): liefert das Objekt am Beginn der Schlange zurück (**null** sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

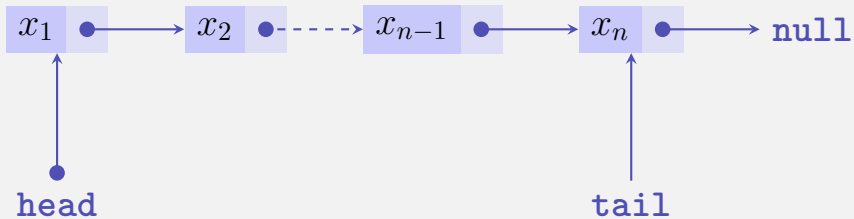
- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)
- **head**(Q): liefert das Objekt am Beginn der Schlange zurück (**null** sonst.)
- **isEmpty**(Q): liefert **true** wenn Queue leer, sonst **false**.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

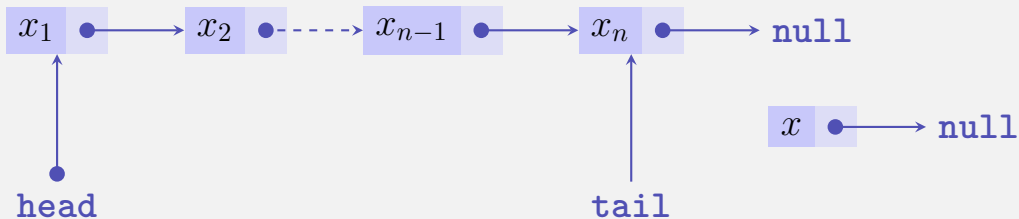
- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)
- **head**(Q): liefert das Objekt am Beginn der Schlange zurück (**null** sonst.)
- **isEmpty**(Q): liefert **true** wenn Queue leer, sonst **false**.
- **emptyQueue**(): liefert leere Queue zurück.

Implementation Queue



`enqueue(x, S):`

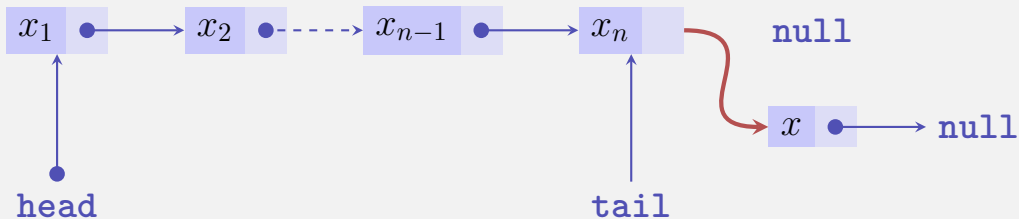
Implementation Queue



`enqueue`(x, S):

- 1 Erzeuge neues Listenelement mit x und Zeiger auf `null`.

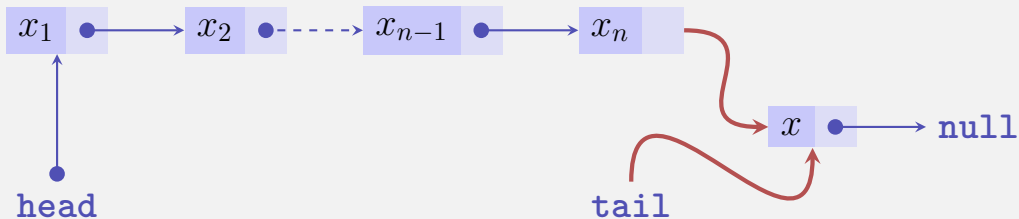
Implementation Queue



`enqueue`(x, S):

- 1 Erzeuge neues Listenelement mit x und Zeiger auf `null`.
- 2 Wenn `tail` \neq `null`, setze `tail.next` auf den Knoten mit x .

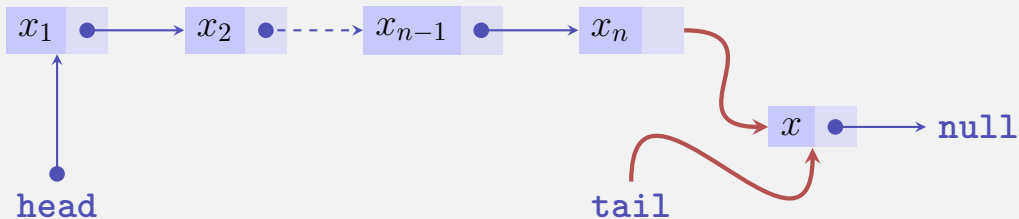
Implementation Queue



`enqueue`(x, S):

- 1 Erzeuge neues Listenelement mit x und Zeiger auf `null`.
- 2 Wenn `tail` \neq `null`, setze `tail.next` auf den Knoten mit x .
- 3 Setze `tail` auf den Knoten mit x .

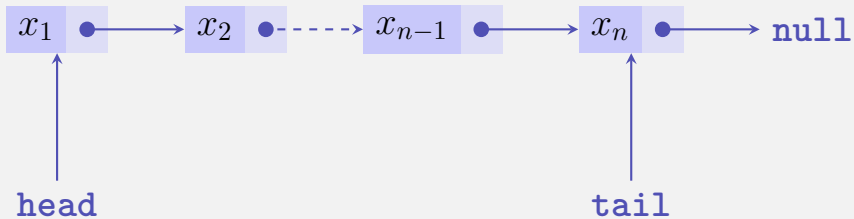
Implementation Queue



`enqueue`(x, S):

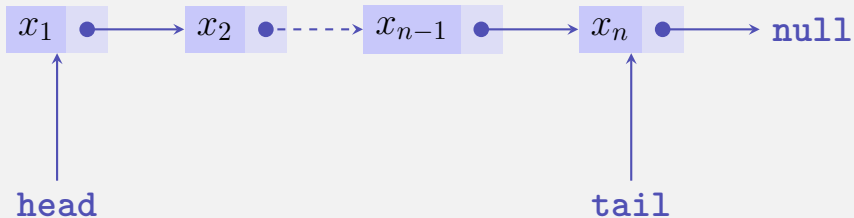
- 1 Erzeuge neues Listenelement mit x und Zeiger auf `null`.
- 2 Wenn `tail` \neq `null`, setze `tail.next` auf den Knoten mit x .
- 3 Setze `tail` auf den Knoten mit x .
- 4 Ist `head` = `null`, dann setze `head` auf `tail`.

Invarianten!



Mit dieser Implementation gilt

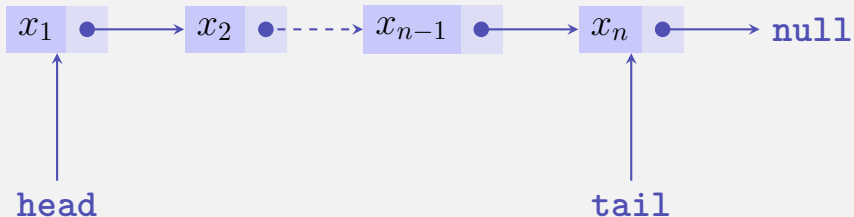
Invarianten!



Mit dieser Implementation gilt

- entweder `head = tail = null`,

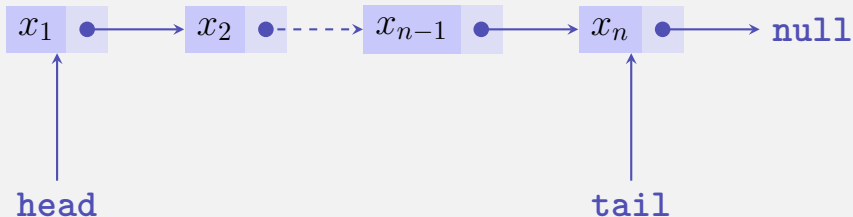
Invarianten!



Mit dieser Implementation gilt

- entweder `head = tail = null`,
- oder `head = tail \neq null` und `head.next = null`

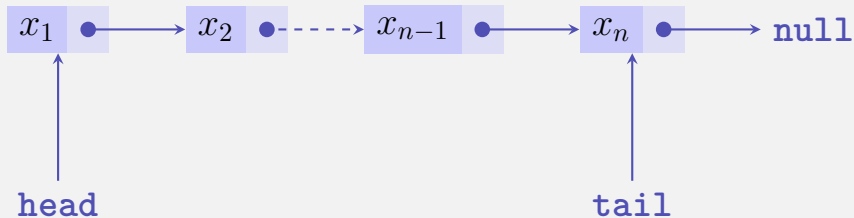
Invarianten!



Mit dieser Implementation gilt

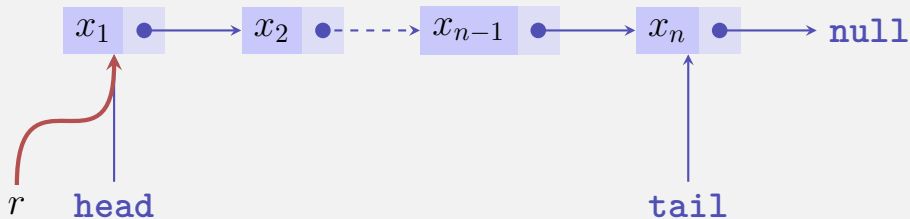
- entweder `head = tail = null`,
- oder `head = tail \neq null` und `head.next = null`
- oder `head \neq null` und `tail \neq null` und `head \neq tail` und `head.next \neq null`.

Implementation Queue



`dequeue(S):`

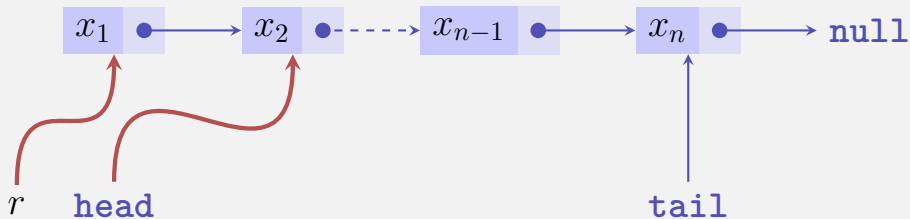
Implementation Queue



`dequeue(S)`:

- 1 Merke Zeiger von `head` in r . Wenn $r = null$, gib r zurück.

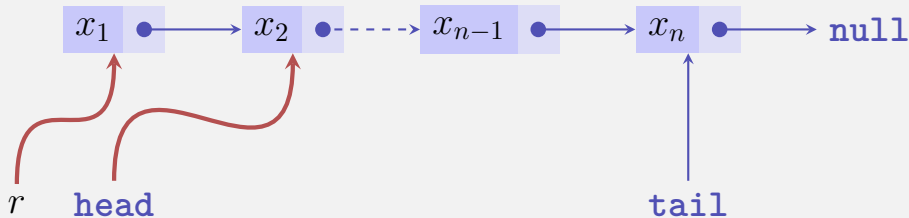
Implementation Queue



`dequeue(S)`:

- 1 Merke Zeiger von `head` in r . Wenn $r = \text{null}$, gib r zurück.
- 2 Setze den Zeiger von `head` auf `head.next`.

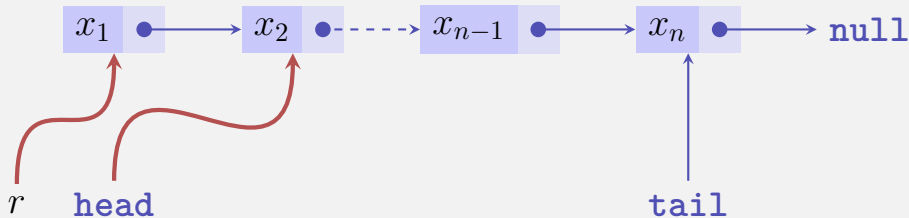
Implementation Queue



`dequeue(S)`:

- 1 Merke Zeiger von `head` in r . Wenn $r = \text{null}$, gib r zurück.
- 2 Setze den Zeiger von `head` auf `head.next`.
- 3 Ist nun `head = null`, dann setze `tail` auf `null`.

Implementation Queue



`dequeue(S)`:

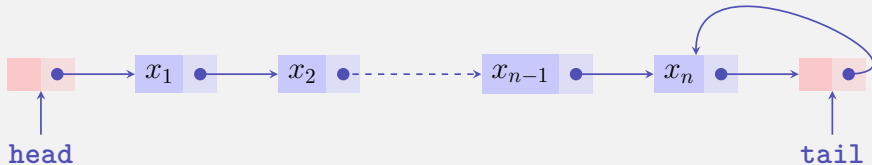
- 1 Merke Zeiger von `head` in `r`. Wenn `r = null`, gib `r` zurück.
- 2 Setze den Zeiger von `head` auf `head.next`.
- 3 Ist nun `head = null`, dann setze `tail` auf `null`.
- 4 Gib den Wert von `r` zurück.

Analyse

Jede der Operationen `enqueue`, `dequeue`, `head` und `isEmpty` auf der Queue ist in $\mathcal{O}(1)$ Schritten ausführbar.

Implementationsvarianten verketteter Listen

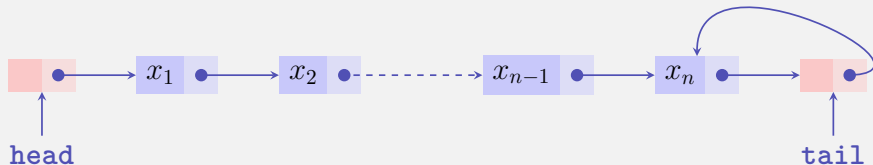
Liste mit Dummy-Elementen (Sentinels).



Vorteil: Weniger Spezialfälle!

Implementationsvarianten verketteter Listen

Liste mit Dummy-Elementen (Sentinels).



Vorteil: Weniger Spezialfälle!

Variante davon: genauso, dabei Zeiger auf ein Element immer einfach indirekt gespeichert. (Bsp: Zeiger auf x_3 zeigt auf x_2 .)

Implementationsvarianten verketteter Listen

Doppelt verkettete Liste



Übersicht

	enqueue	delete	search	concat
(A)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = Einfach verkettet

(B) = Einfach verkettet, mit Dummyelement am Anfang und Ende

(C) = Einfach verkettet, mit einfach indirekter Elementadressierung

(D) = Doppelt verkettet