

Algorithmen und Datenstrukturen

Sommersemester 2020

Stefan Näher

Universität Trier

naeher@uni-trier.de

Vorlesung 4

30. April 2020

Kapitel I: Grundlagen

- Maschinenmodell
- Pseudo-Code
- Laufzeitanalyse
- $\mathcal{O}$ -Notation,
- Rekursive Algorithmen
- Grundlegende Datenstrukturen

Ein **(statische) Schlange oder Queue** wird beschrieben durch:

Maximalgröße n ;

Datentyp D der Queue-Elemente (hier meistens Integers)

Feld $A[1..n + 1]$

Index des ersten und letzten Schlangen-Elements: **first** und **last**

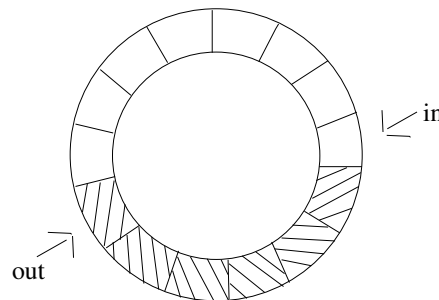
Achtung:

Das Feld A muss zyklisch betrachtet werden $\leadsto$ Übung.

Schlangen arbeiten nach dem FIFO-Prinzip: First-In / First-Out.

Schlangen: statische Implementierung

Wir verwenden wieder Felder. Diese werden aber zyklisch interpretiert als **Ring-puffer**.



Beachte: Schlangen arbeiten nach dem FIFO-Prinzip (First In / First Out).

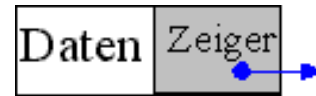
Schlangen-Operationen

S.clear() S.empty() S.append(x) S.pop() S.first()	Implementierung (Übung)
---	-------------------------

Listen

Man unterscheidet
einfach verkettete und **doppelt** verkettete Listen.

Einfach verkettete Listen

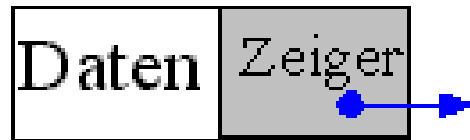


Die einzelnen **Listenelemente** unterteilen sich in:
Speicherplatz für die Daten (Werte) sowie
einen **Verweis** (Zeiger, Referenz, Pointer) auf das nächste Listenelement.



Der **Kopf** (Start) ist ein Verweis auf das erste Listenelement.
Das letzte Listenelement enthält einen **leeren Verweis** (null).

Listenelemente



Beschreibung durch eine Struktur (C) oder Klasse (Java)

```
class ListElem {  
    int val;           // Daten zu diesem Element  
    ListElem next;    // Referenz auf nächstes Element  
} ;
```

ListElem p (Referenz- oder Pointervariable)

Allokation eines Listenelements im Speicher der RAM

```
p ← new ListElem;
```

Zugriff auf Daten

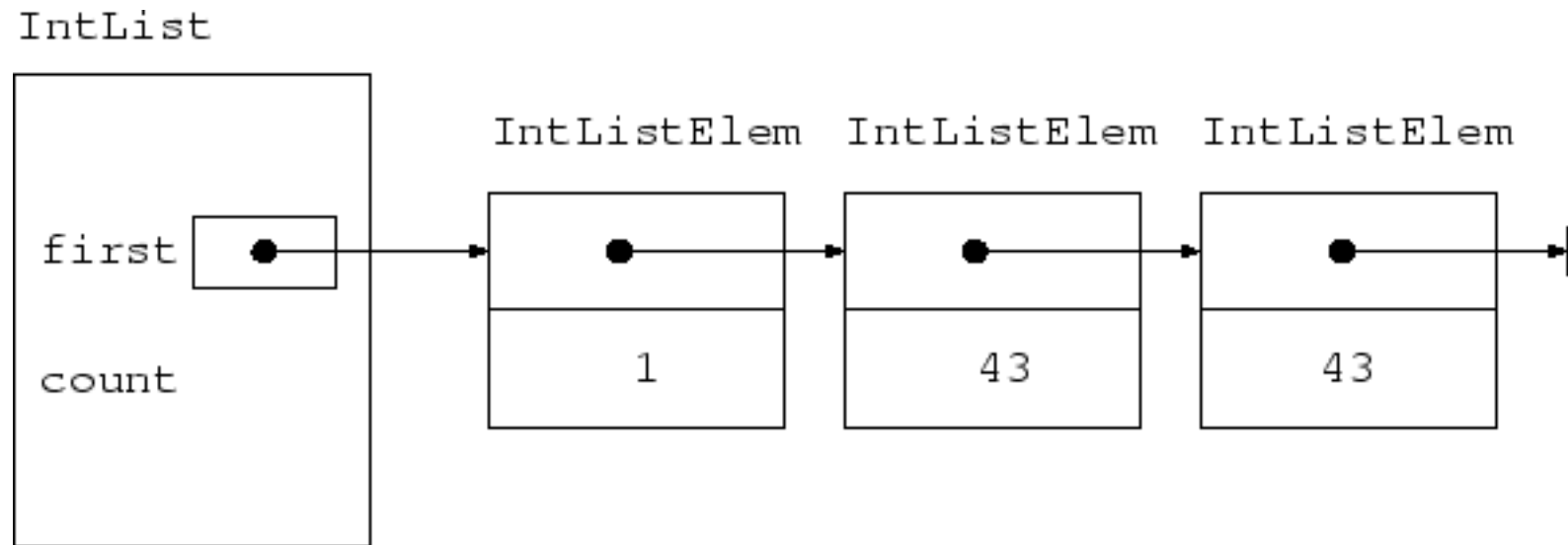
```
p.val ← 17;
```

```
p.next ← null;
```

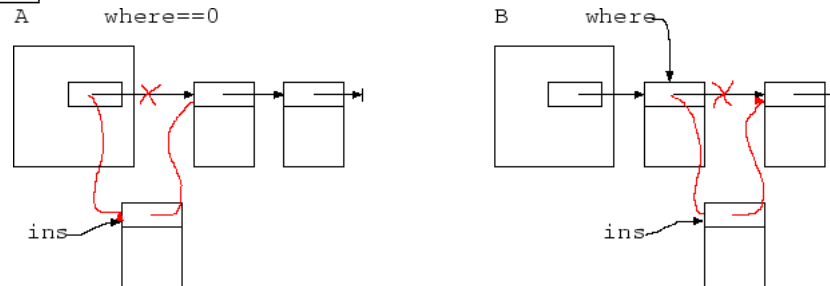
```
q ← new ListElem;
```

```
p.next ← q;
```

Liste



Einfügen in Liste



Vorne Einfügen (push)

$p \leftarrow \text{new ListElem};$

$p.\text{val} \leftarrow x$

$p.\text{next} \leftarrow \text{kopf};$

$\text{kopf} \leftarrow p;$

Nach einem Element q einfügen

$p \leftarrow \text{new ListElem};$

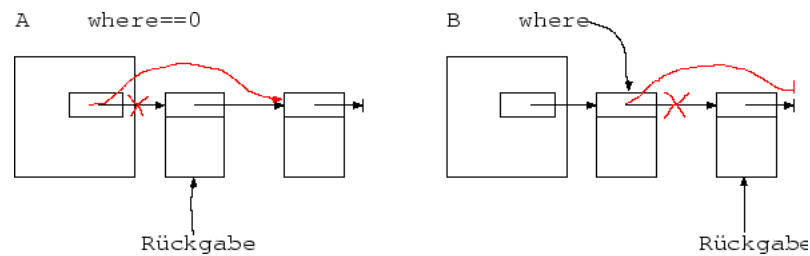
$p.\text{val} \leftarrow x$

$p.\text{next} \leftarrow q.\text{next};$

$q.\text{next} \leftarrow p;$

Laufzeit: $\mathcal{O}(1)$

Löschen aus Liste



Entfernen des ersten Listenelements

$\text{kopf} \leftarrow \text{kopf.next}$

Entfernen des Listenelements nach Position q

$q.\text{next} \leftarrow q.\text{next.next}$

Laufzeit: $\mathcal{O}(1)$

Iteration über alle Elemente

$p \leftarrow \text{kopf};$

while $p \neq \text{null}$ **do**

$x \leftarrow p.\text{val};$

 // do something with x

 PRINT(x);

$p \leftarrow p.\text{next};$

od

Laufzeit: $\mathcal{O}(n)$

Anwendung: Linear Suche nach einem Wert in der Liste.

Was (einfach verkettete) Listen noch können

Zur Verwaltung von Listen wollen wir folgende Operationen vorsehen

- Erzeugen einer leeren Liste.
- Einfügen von Elementen an beliebiger Stelle.
- Entfernen von (beliebigen) Elementen.
- Durchsuchen der Liste.

Dynamische Keller

Verwenden Sie eine einfach verkettete Liste zur Darstellung von Stacks beliebiger Größe.

~> Übung.

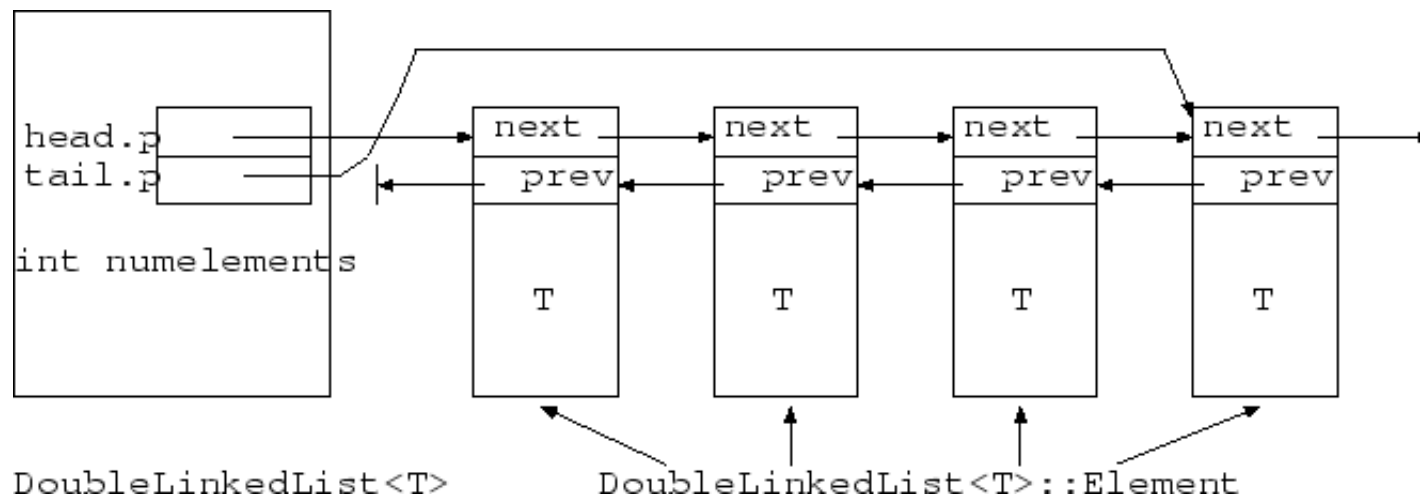
Wie kann man die Operationen clear, push, top, pop, empty realisieren ?

Dynamische Schlangen

Hier brauchen wir eine Variante die außer dem Listenkopf (Verweis auf das erste Element) auch einen Verweis auf das letzte Element speichert.

Übung: Realisieren Sie die Queue-Operationen `clear`, `empty`, `append`, `pop`, `top`.

Doppelt verkettete Listen



Details werden in der Übung behandelt.

Kapitel II: Sortieren

Allgemeine Sortiervverfahren

- **Heapsort** (Sortieren durch Max-Auswahl)
- **Quicksort** (Sortieren durch Teilen)
- **Mergesort** (Sortieren durch Mischen)

Spezielle Sortiervverfahren

- **Counting Sort** (Sortieren durch Zählen)
- **Bucket Sort** (Sortieren durch Verteilung)

Heapsort: Sortieren durch Max-Auswahl

... hatten wir bereits in der Übung betrachtet.

Grundidee: Suche n -mal nach Maximum im noch unsortierten Feld

Frage: Muss die Maximumsuche “unbedingt” eine lineare sein und somit Linearzeit benötigen ?

NEIN! Wenn wir die Daten geeignet halten (in einem in einem Feld gespeicherten speziellen Baum (Heap)...

Heaps

Ein **Heap** (Haufen) ist ein Baum, dessen Knoten mit Zahlen $\text{num}(v)$ beschriftet sind, so dass für jeden Knoten v , der nicht die Wurzel ist, gilt:

$$\text{num}(v) \leq \text{num}(\text{parent}(v))$$

Folgerung: Die größte Zahl in einem Heap steht an der Wurzel.
Daher heißt so ein Heap auch **Max-Heap**.

Analog: **Min-Heap**

Wichtig: “Regelmäßige” Heaps.

Binärbäume

Ein **Binärbaum** ist ein Baum, bei dem jeder Knoten null oder zwei Kinder hat (vielleicht mit einer Ausnahme).

Knoten mit null Kindern heißen auch **Blätter**.

Die Länge des Pfades von der Wurzel zu einem Knoten v heißt **Tiefe** von v .

Ein Binärbaum heißt **ausgeglichen**, wenn es eine Zahl k gibt mit:

- (1) Alle Blätter haben Tiefe k oder $k + 1$.
- (2) Auf Tiefe $k + 1$ stehen die Blätter so weit links wie möglich.

Skizze ...

Heaps als Feld

Heaps lassen sich statisch in einem Feld abspeichern.

Für einen ausgeglichenen Heap mit n Knoten benötigen wir ein Feld $A[1..n]$.

Dabei gilt:

- (1) $2i$ und $2i + 1$ sind die Indizes des linken bzw. rechten Kindes von i .
- (2) $\lfloor i/2 \rfloor$ ist der Index des Elternteils von i .

Heap-Eigenschaft

Wird in dieser Weise ein Heap in $A[1..n]$ abgespeichert, so gilt:

$$\forall i \in \{2, \dots, n\} : A[i] \leq A[\lfloor i/2 \rfloor].$$

Umgekehrt lässt sich jedes Feld mit dieser Heap-Eigenschaft als ausgeglichener Heap interpretieren.