

# **Einführung in die Programmierung**

## **Einfache Datenstrukturen**

# Einfache Datenstrukturen

- ☐ Arrays
- ☐ Verkettete Listen
- ☐ Doppeltverkettete Listen
- ☐ Stack (Stapel)
- ☐ Queue (Warteschlangen)
  
- ☐ Warum: Grundlagen für alle weiteren Algorithmen

# Arrays

□ Arrays: Speicherung größerer Mengen oder Folgen gleichartiger Elemente.

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.



length

# Problem: Unbekannte Anzahl an Elementen

## ❑ Probleme:

- Die Anzahl der Elemente ist oft vorher nicht bekannt
- Die Anzahl ändert sich während des Programmdurchlaufs
- ...

## ❑ Folgerung:

- Brauchen dynamischere Möglichkeit der Speicherverwaltung

## ❑ Möglichkeiten:

- Adhoc mittels malloc
- Oder mittels **dynamischer Datenstrukturen**

# Dynamische Datenstrukturen

# Dynamische Datenstrukturen

## □ Beispiele von dynamischen Datenstrukturen:

- Liste
- Baum
- Graph
- ...

## □ Eigenschaften

- Erweiterbar
- Schnelles Einfügen
- Löschen möglich

# Wie dynamisch sind Arrays?

❑ Arrays: Speicherung größerer Mengen oder Folgen gleichartiger Elemente.

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.

❑ Allerdings:

- Löschen: Lücken im Array.
- Einfügen: i. A. neues Array und Daten müssen kopiert werden.



length



❑ Arrays: Speicherung größerer Mengen oder Folgen gleichartiger Elemente.

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.

❑ Allerdings:

- Löschen: Lücken im Array.
- Einfügen: i. A. neues Array und Daten müssen kopiert werden.



length

□ Arrays: Speicherung größerer Mengen oder Folgen gleichartiger Elemente.

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.

□ Allerdings:

- Löschen: Lücken im Array.
- Einfügen: i. A. neues Array und Daten müssen kopiert werden.



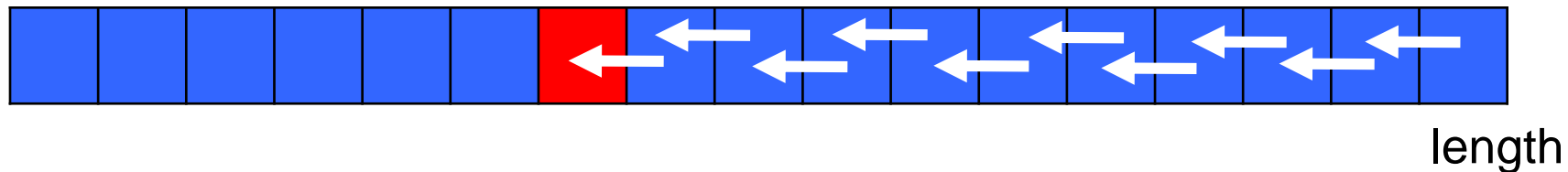
length

□ Arrays: Speicherung größerer Mengen oder Folgen gleichartiger Elemente.

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.

□ Allerdings:

- Löschen: Lücken im Array.
- Einfügen: i. A. neues Array und Daten müssen kopiert werden.



❑ Arrays: Speicherung größerer Mengen oder Folgen gleichartiger Elemente.

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.

❑ Allerdings:

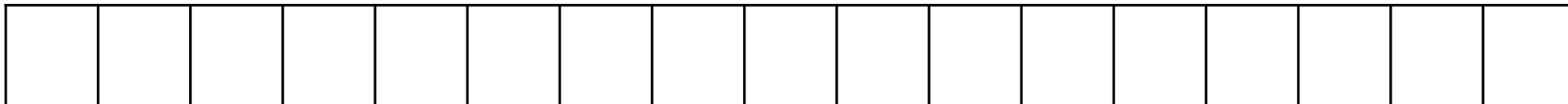
- Löschen: Lücken im Array.
- Einfügen: i. A. neues Array und Daten müssen kopiert werden.



length

# Dynamic Array

- ❑ Dynamic Array: Erweiterung des klassischen Arrays für effizientes Löschen / Einfügen am Ende.
  - Unterscheidung zwischen capacity und length:
    - capacity: Größe im Speicher.



capacity

# Dynamic Array

- ❑ Dynamic Array: Erweiterung des klassischen Arrays für effizientes Löschen / Einfügen am Ende.
  - Unterscheidung zwischen capacity und length:
    - capacity: Größe im Speicher.
    - length: Anzahl der tatsächlich verwendeten Elemente.



length

capacity

# Dynamic Array

- ❑ Dynamic Array: Erweiterung des klassischen Arrays für effizientes Löschen / Einfügen am Ende.
  - Vorteile des klassischen Arrays bleiben erhalten.
  - Einfügen und Löschen am Ende sind  $\approx O(1)$ .
  - ABER: Einfügen und Löschen an einer beliebigen Stelle immer noch teuer.



length

capacity

# Einfache Datenstrukturen

- ☐ Arrays
- ☐ Verkettete Listen
- ☐ Doppeltverkettete Listen
- ☐ Stack (Stapel)
- ☐ Queue (Warteschlangen)
  
- ☐ Warum: Grundlagen für alle weiteren Algorithmen

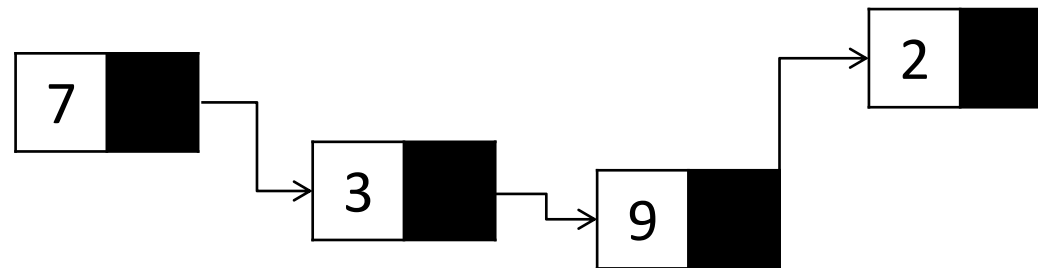


# Verkette Listen

# Verkettete Liste

## □ Verkettete Liste:

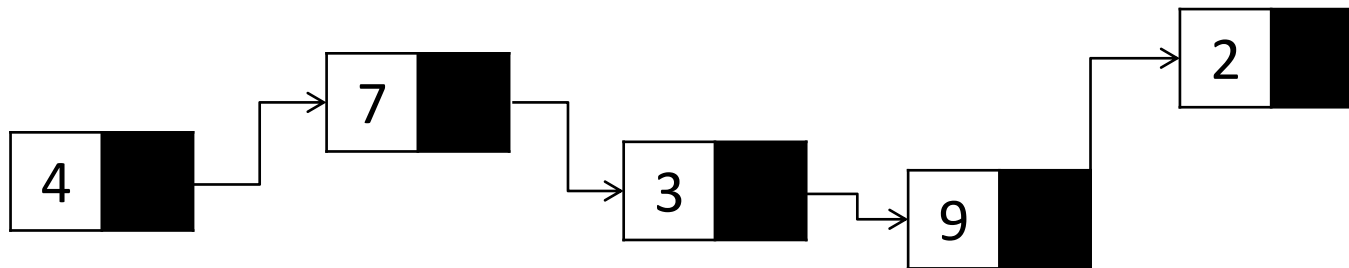
- Elemente bestehen aus Inhalt und Nachfolger.
- Jedes Element "verweist" auf seinen Nachfolger.



# Verkettete Liste

## □ Verkettete Liste:

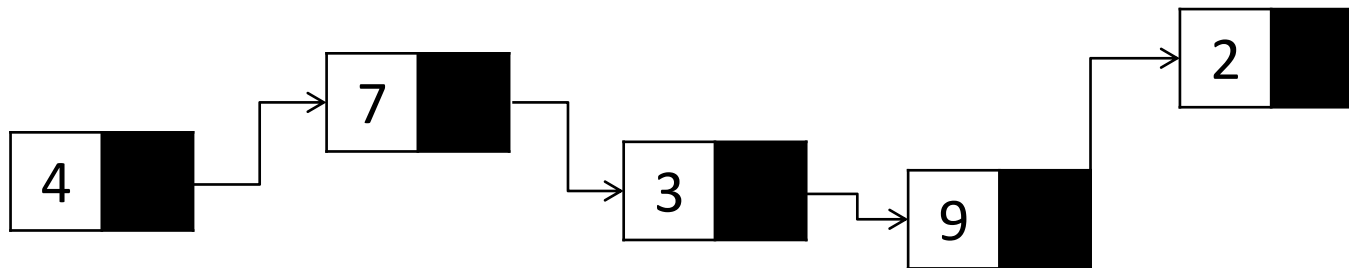
- Jedes Element "verweist" auf seinen Nachfolger.
- Elemente können beliebig eingefügt.



# Verkettete Liste

## □ Verkettete Liste:

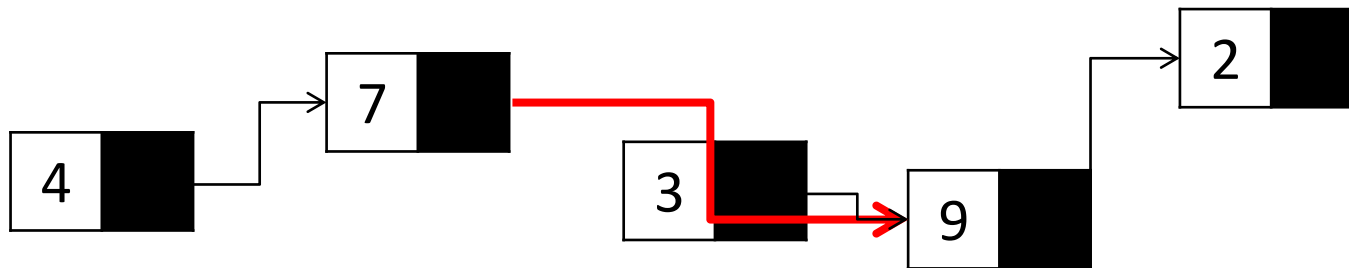
- Jedes Element "verweist" auf seinen Nachfolger.
- Elemente können beliebig eingefügt und gelöscht werden.



# Verkettete Liste

## □ Verkettete Liste:

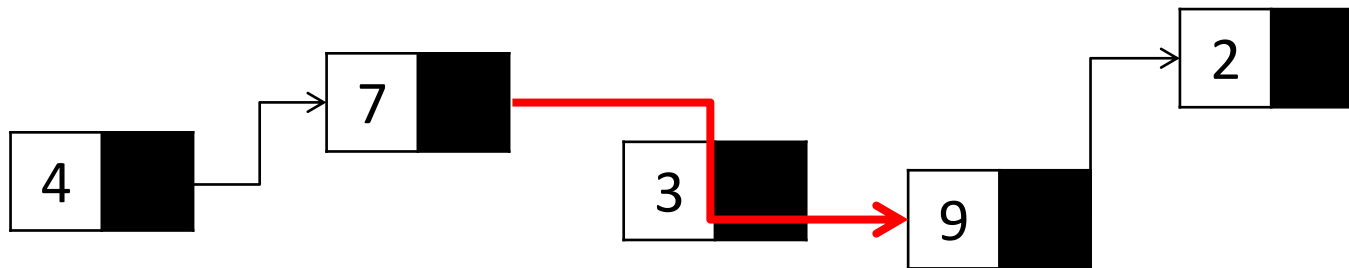
- Jedes Element "verweist" auf seinen Nachfolger.
- Elemente können beliebig eingefügt und gelöscht werden.



# Verkettete Liste

## □ Verkettete Liste:

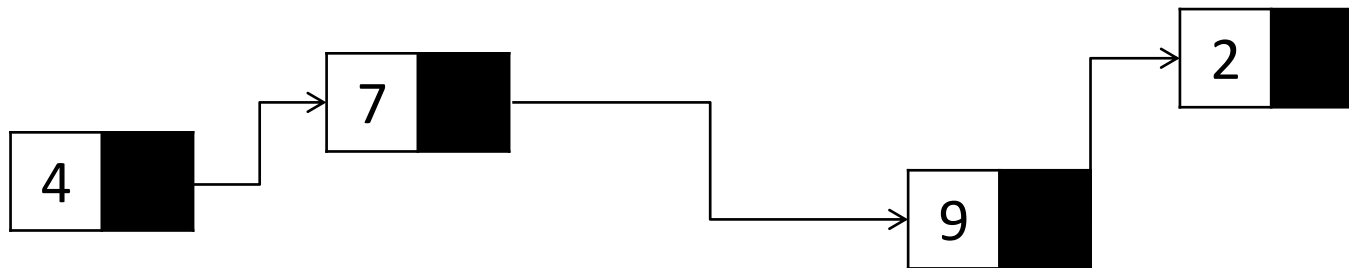
- Jedes Element "verweist" auf seinen Nachfolger.
- Elemente können beliebig eingefügt und gelöscht werden.



# Verkettete Liste

## □ Verkettete Liste:

- Jedes Element "verweist" auf seinen Nachfolger.
- Elemente können beliebig eingefügt und gelöscht werden.



- Dabei wird „nur“ der Verweis auf den Nachfolger umgesetzt.
- Und der Speicher des gelöschten Elements freigegeben.

# Verkettete Liste: Implementierung



# Verkettete Liste: Implementierung

□ Die Implementierung von verketteten Listen benötigt:

➤ **Elemente** bestehend aus

- Inhalt
- Verweis auf Nachfolger

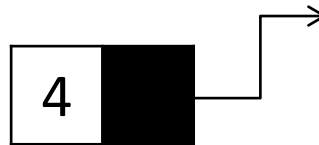
➤ Und die **Logik** zum

- Einfügen
- Durchlaufen und Ausgeben
- Löschen

# Verkettete Liste Elemente

□ Listenelemente sind ein **zusammengesetzter Datentyp**

- Datenwert
- Verweis auf das nächste Element
- ...



# **Ausflug: Zusammengesetzte Datentypen in C**

# Zusammengesetzte Datentypen

- ❑ Zusammenfassung von zusammengehörigen Daten in eigenen Datentyp **struct**

- ❑ Beispiel Produkte mit Namen und Preis:

```
struct produkt {  
    char name[255];  
    float preis;  
};  
struct produkt beispiel;
```

- ❑ Zugriff durch Selektor **.** (**Punkt**)

```
beispiel.preis = 0.79;  
strncpy(beispiel.name, "Apfel", 255);  
printf("Ware %s mit Preis %f\n",  
        beispiel.name, beispiel.preis);
```

# Zusammengesetzte Datentypen

- ❑ Zusammenfassung von zusammengehörigen Daten in eigenen Datentyp **struct**

- ❑ Beispiel Produkte mit Namen und Preis:

```
struct produkt{  
    char name[255];  
    float preis;  
};  
struct produkt beispiel;  
// Array vom Typ struct produkt  
struct produkt warenkorb[100];
```

- ❑ D.h. Structs sind ganz normale Datentypen bei denen wir Arrays, Pointer, Adressen, etc haben.

# Zusammengesetzte Datentypen

- ❑ Zusammenfassung von zusammengehörigen Daten in eigenen Datentyp **struct**

- ❑ Beispiel Produkte mit Namen und Preis:

```
struct produkt{
    char name[255];
    float preis;
};
struct produkt beispiel;
// Array vom Typ struct produkt
struct produkt warenkorb[100];
// Pointer = Adresse
struct produkt * ware = warenkorb;
```

- ❑ D.h. Structs sind ganz normale Datentypen bei denen wir Arrays, Pointer, Adressen etc haben.

# Zusammengesetzte Datentypen

- ❑ Zusammenfassung von zusammengehörigen Daten in eigenen Datentyp `struct`

- ❑ Beispiel Produkte mit Namen und Preis:

```
struct produkt{  
    char name[255];  
    float preis;  
};  
struct produkt beispiel;  
struct produkt warenkorb[100];  
struct produkt * ware = warenkorb;
```

- ❑ Bei Pointern durch Selektor `->`

```
printf(„Produkt %s mit Preis %f\n“,  
       ware->name, ware->preis);
```

❑ Oft sinnvoll, eigene Typen zu definieren mittels **typedef**

❑ Beispiel:

```
typedef struct produkt {  
    char name[255];  
    float preis;  
} ware;  
// Variablendeklaration  
ware produkt1, produkt2;
```

❑ Warum? ⇒ Bessere Lesbarkeit, bessere Dokumentation!  
Weniger Tipparbeit



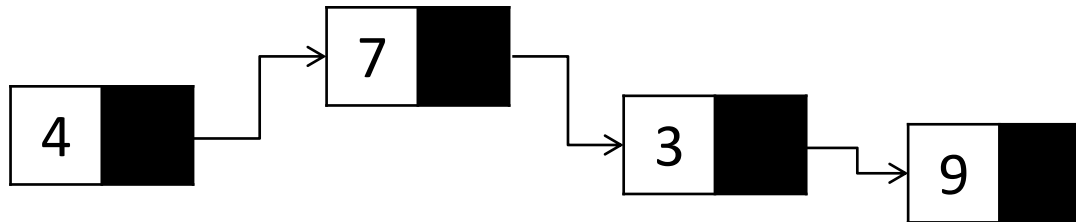
**Ende Ausflug:**  
**Zusammengesetzte Datentypen**

**Weiter mit:**  
**Implementierung verketteter Listen**

# Verkettete Liste

## □ Verkettete Liste:

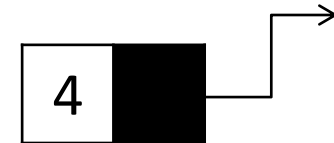
- Jedes Element "verweist" auf seinen Nachfolger.



# Verkettete Liste – Elemente

❑ Listenelemente sind ein **zusammengesetzter Datentyp**

- Datenwert
- Verweis auf das nächste Element



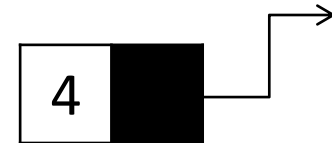
❑ Einfach verkettete Liste

```
/* Datentyp für einfach verkettete Liste */  
struct simple_list {  
    int value;                // Daten  
    struct simple_list *next; // Nachfolger  
};
```

# Verkettete Liste – Elemente

□ Listenelemente sind ein **zusammengesetzter Datentyp**

- Datenwert
- Verweis auf das nächste Element



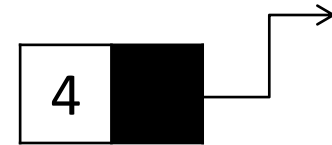
□ Einfach verkettete Liste

```
/* Datentyp für einfach verkettete Liste */  
typedef struct simple_list {  
    int value;                // Daten  
    struct simple_list *next; // Nachfolger  
} slist;
```

# Verkettete Liste – Elemente

□ Listenelemente sind ein **zusammengesetzter Datentyp**

- Datenwert
- Verweis auf das nächste Element



□ Struct Definition oft mit Namenskonvention `_typename`

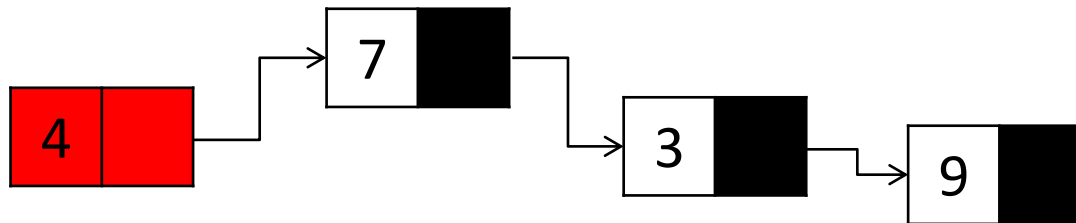
```
/* Datentyp für einfach verkettete Liste */  
typedef struct _slist {  
    int value;           // Daten  
    struct _slist *next; // Nachfolger  
} slist;
```

# Verkettete Liste – Wurzel

## □ Verkettete Liste:

➤ Jedes Element “verweist” auf seinen Nachfolger.

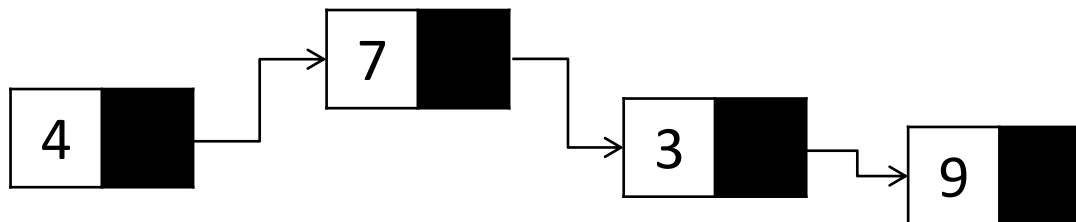
□ Erstes Element wird oft „**Wurzel**“ (engl. „root“), „**Anker**“ oder „Kopf“ der Liste genannt



# Verkettete Liste – Durchlaufen

## □ Durchlaufen der Liste:

- Anfang an der Wurzel
- Solange wie Nachfolger existiert
  - Gib den Wert aus
  - Gehe zum Nachfolger



# Verkettete Liste – Durchlaufen

```
❑ typedef struct _slist {  
    int value;           // Daten  
    struct _slist *next; // Nachfolger  
} slist;
```

❑ Durchlaufen der Liste i.d.R. von der Wurzel der Liste aus

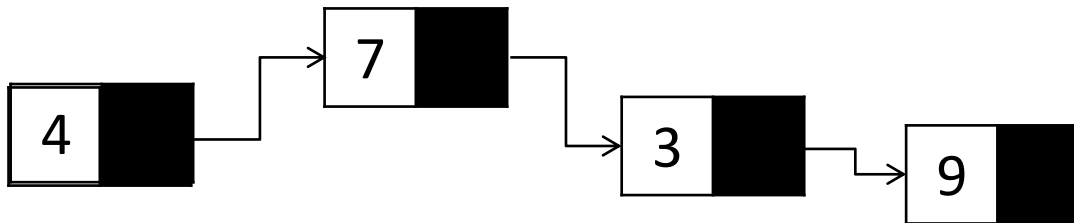
```
slist *root, *tmp;  
tmp = root;           // Anfang an der Wurzel  
while(tmp != NULL) {  // Solange Nachfolger  
    printf("%d\n", tmp->value); // Wert  
    tmp = tmp->next      // Nächstes Element  
}
```



# Verkettete Liste – Einfügen

## □ Einfügen:

- Neuen Knoten anlegen
- Knoten initialisieren
- Knoten einfügen (Hier am Anfang)



# Verkettete Liste – Einfügen

□ Objekte vom Type Liste werden zur Laufzeit:

➤ Alloziert

```
slist *insert(slist *list_pointer, int value){  
    slist * new;  
    new = (slist *) calloc(1, sizeof(slist));
```

➤ Besetzt bzw. initialisiert

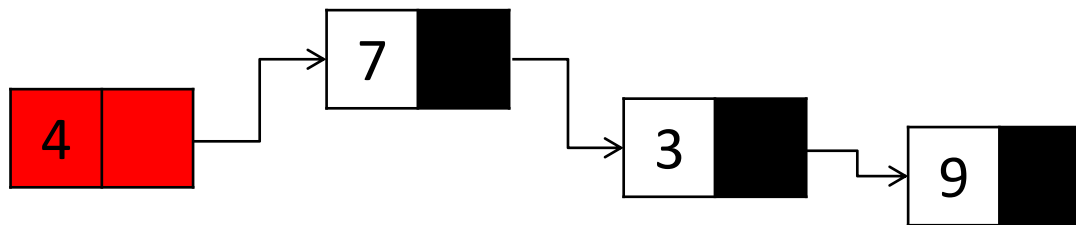
```
    new->value = value;  
    new->next = NULL;
```

➤ Und in die Liste eingehängt

```
    if(list_pointer)  
        new->next = list_pointer;  
    return(new) ;  
}
```

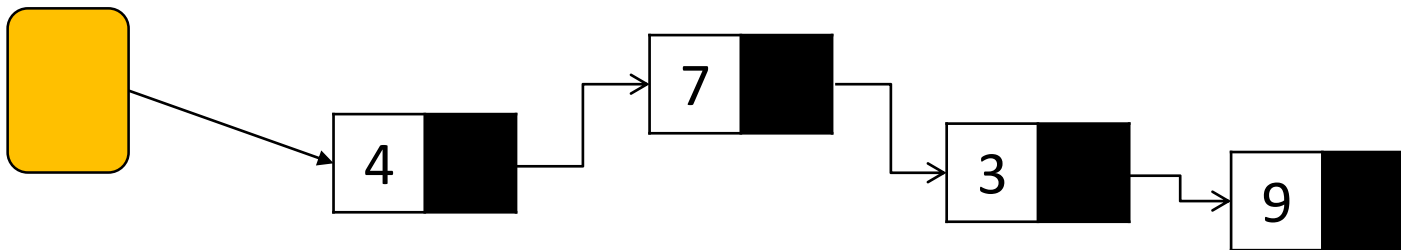
# Verkettete Liste mit extra Wurzel

- ❑ Erstes Element wird oft „Wurzel“ (engl. „root“), „Anker“ oder „Kopf“ der Liste genannt
- ❑ Allerdings braucht man häufig weitere Informationen
  - Anzahl (#) Listenelemente
  - Letztes Element der Liste



# Verkettete Liste mit Wurzel

- ❑ Erstes Element wird oft „Wurzel“ (engl. „root“), „Anker“ oder „Kopf“ der Liste genannt
- ❑ Allerdings braucht man häufig weitere Informationen
  - Anzahl (#) Listenelemente
  - Letztes Element der Liste
- ❑ Lösung
  - **Separate Wurzel** für Verwaltungsinformationen



# Verkettete Liste mit Wurzelement

## □ Datentyp für die Wurzel

```
typedef struct _list {  
    int count;        // Anzahl Listeneinträge  
    slist *first;     // Erstes Element der Liste  
} list;
```

## □ Funktion zum Initialisieren

```
list * warenliste = calloc(1, sizeof(list));  
void init_list(list *list_pointer) {  
    list_pointer->first = NULL;  
    list_pointer->count = 0;  
}  
init_list(warenliste);
```

# Verkettete Liste mit Wurzelement

- Datentyp für die Wurzel

```
typedef struct _list {  
    int count;        // Anzahl Listeneinträge  
    slist *first;     // Erstes Element der Liste  
} list;
```

- Funktion zum Einfügen

```
list * warenliste = calloc(1, sizeof(list));  
void list_insert(list *list_pointer, int value){  
    slist * new = (slist *) calloc(1, sizeof(slist));  
    new->value = value;  
    new->next = list_pointer->first;  
    list_pointer->first = new;  
    list_pointer->count++;  
}  
list_insert(warenliste, 100);
```

# Verkettete Liste mit Wurzelement

- Datentyp für die Wurzel

```
typedef struct _list {  
    int count;        // Anzahl Listeneinträge  
    slist *first;     // Erstes Element der Liste  
} list;
```

- Funktion zum Ausgeben

```
list * warenliste = calloc(1, sizeof(list));  
void list_print(list *list_pointer) {  
    slist *tmp = list_pointer->first;  
    while(tmp) {  
        printf("cur: %d ", tmp->value);  
        tmp = tmp->next;  
    } printf("\n");  
    list_print(warenliste);  
}
```

# Verkettete Liste – Operationen

## □ Typische Operationen

- Initialisieren
- Einfügen
- Suchen entspricht Durchlaufen
- Löschen – gleich
- Invertieren – gleich
- Sortieren – später
- Konkatenieren
- ....



# Linked List Implementierungen und Laufzeiten

□ Zahlreiche Variationsmöglichkeiten bei der Implementierung.

➤ "Beste" Variante hängt vom Problem ab.

□ Minimale Implementierung:

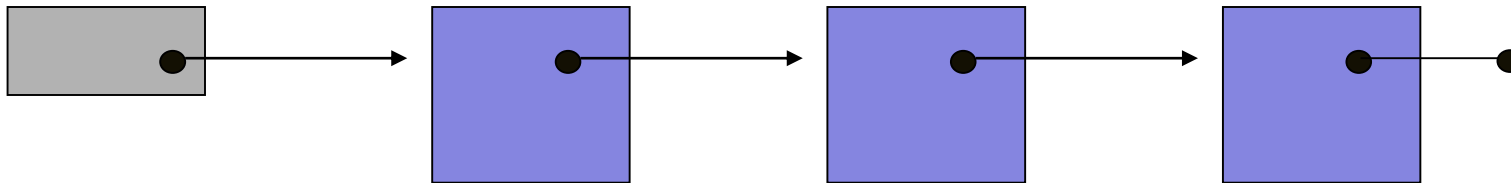
➤ head (Kopfzeiger): Zeigt auf das erste Element der Liste.

➤ Methoden / Funktionen:

- Find element // Suchen
- Insert element // Einfügen
- Delete element // Löschen

# Wiederholung Listenoperationen: Suchen

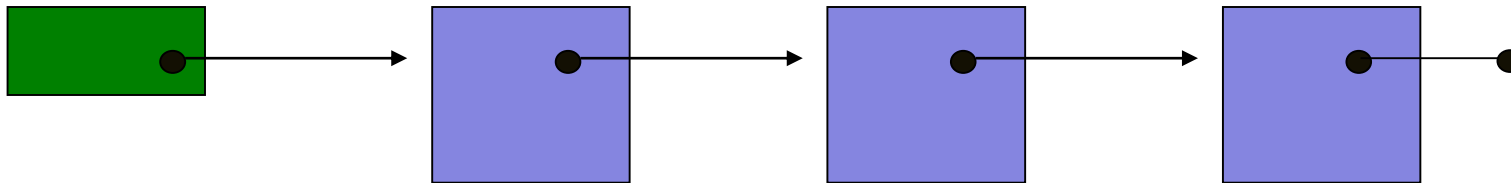
□ Suchen eines Objektes in der Liste



□ Durchlaufen der Liste

# Wiederholung Listenoperationen: Suchen

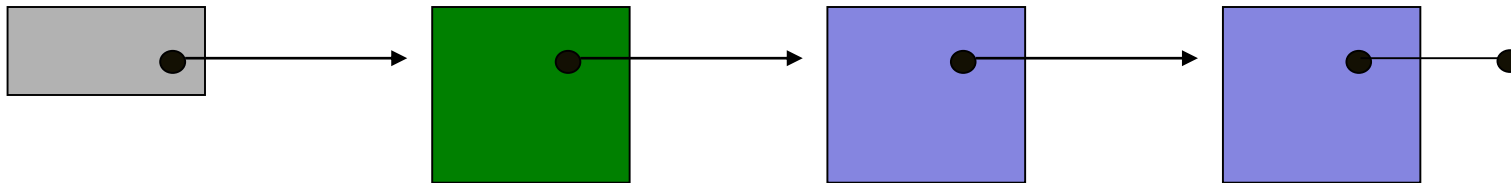
□ Suchen eines Objektes in der Liste



□ Durchlaufen der Liste

# Wiederholung Listenoperationen: Suchen

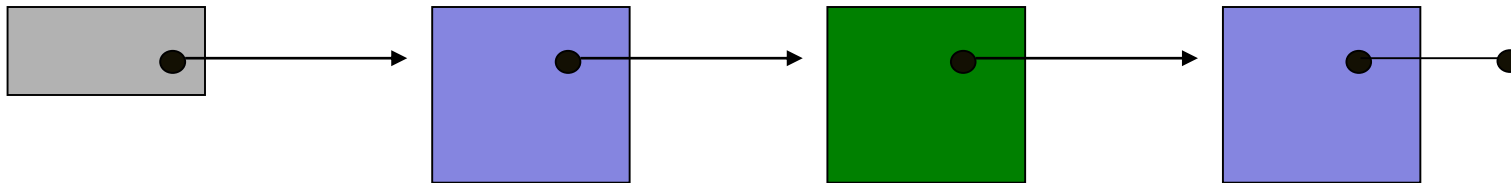
□ Suchen eines Objektes in der Liste



□ Durchlaufen der Liste

# Listenoperationen: Suchen Laufzeit

## □ Suchen eines Objektes in der Liste



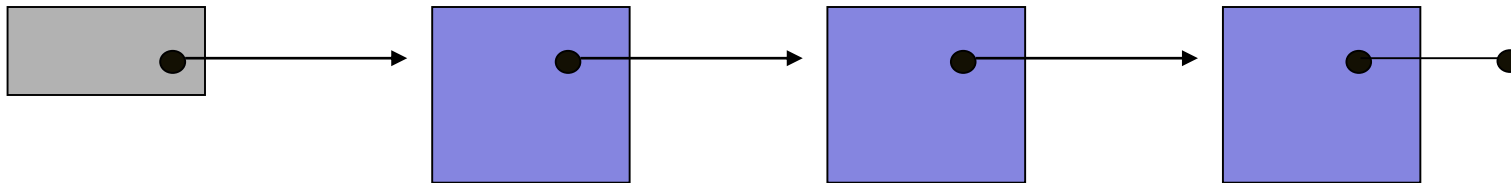
## □ Durchlaufen der Liste

- Bis das gewünschte Element gefunden wurde
- Oder das Ende der Liste erreicht ist

## □ Laufzeit: $O(n)$

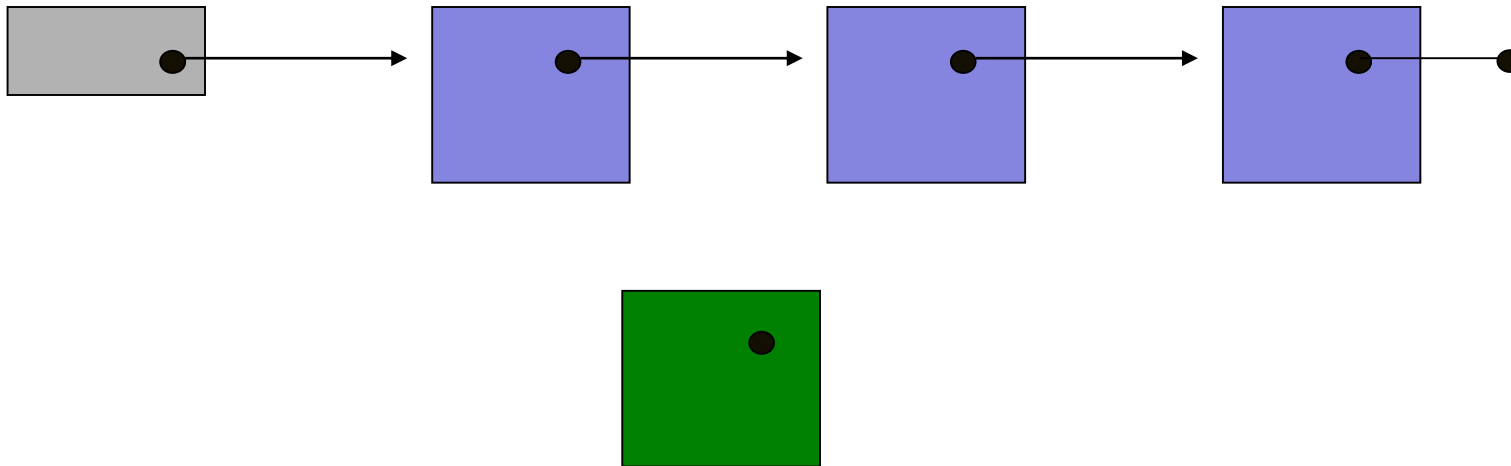
# Wiederholung Listenoperationen: Einfügen

□ Einfügen eines Objektes in die Liste:



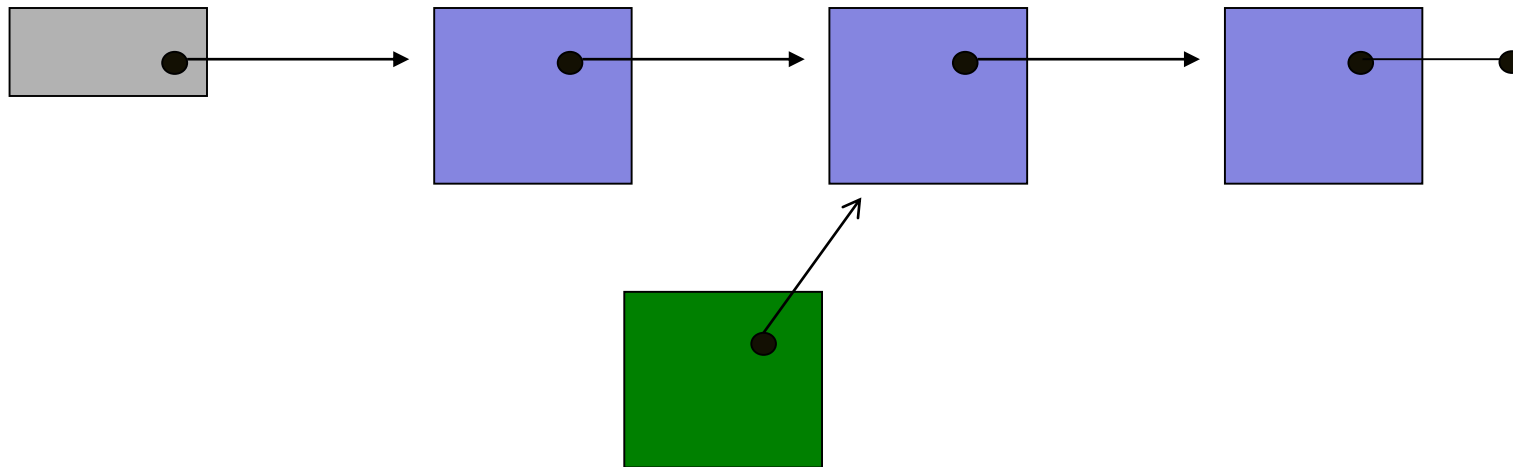
# Wiederholung Listenoperationen: Einfügen

□ Einfügen eines Objektes in die Liste:



# Wiederholung Listenoperationen: Einfügen

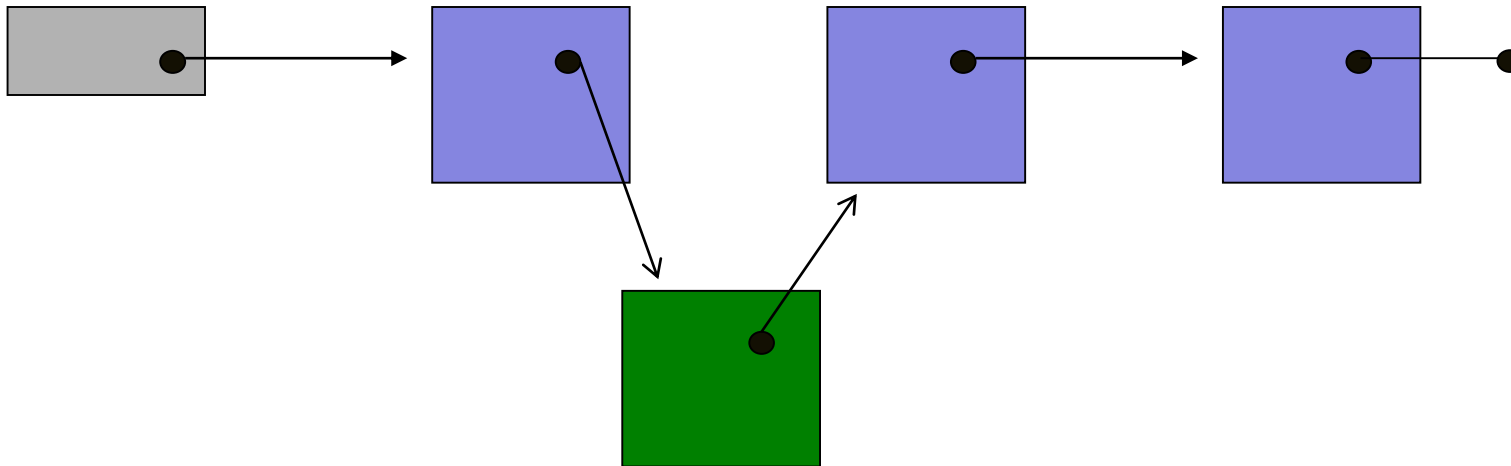
□ Einfügen eines Objektes in die Liste:





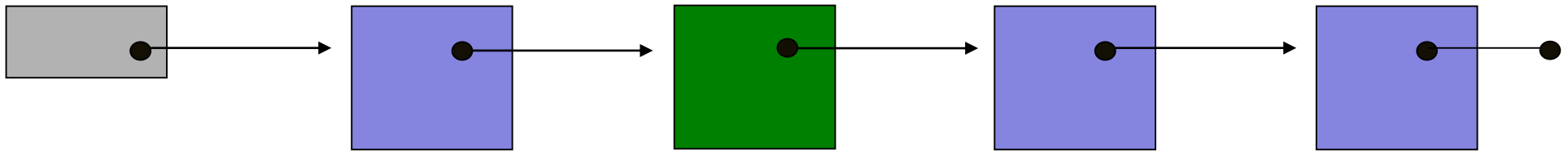
# Wiederholung Listenoperationen: Einfügen

□ Einfügen eines Objektes in die Liste:



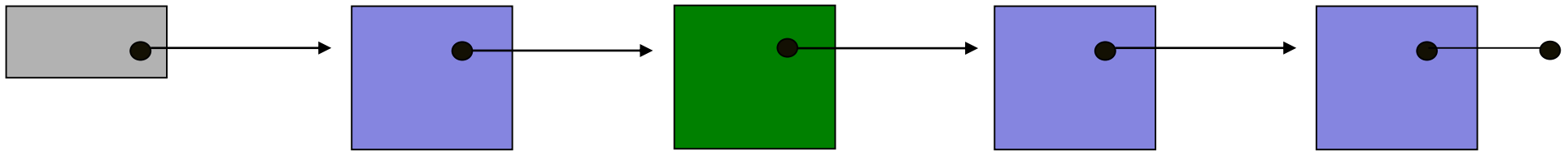
# Wiederholung Listenoperationen: Einfügen

□ Einfügen eines Objektes in die Liste:



# Listenoperationen: Einfügen Laufzeit

□ Einfügen eines Objektes in die Liste:



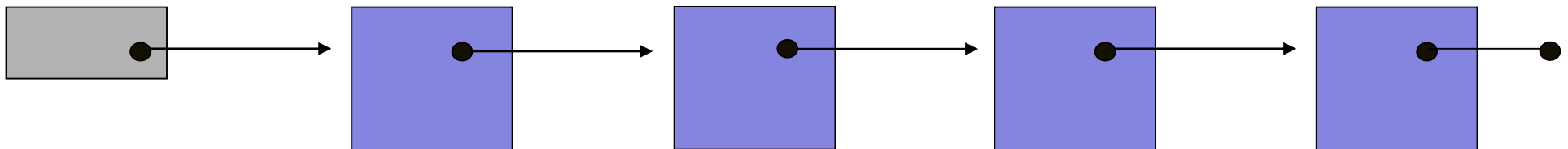
□ Elemente werden nur „eingehängt“.

□ Einfügen „hinter“ bekannter Stelle ist  $O(1)$ .  
Position ist durch Pointer bekannt.

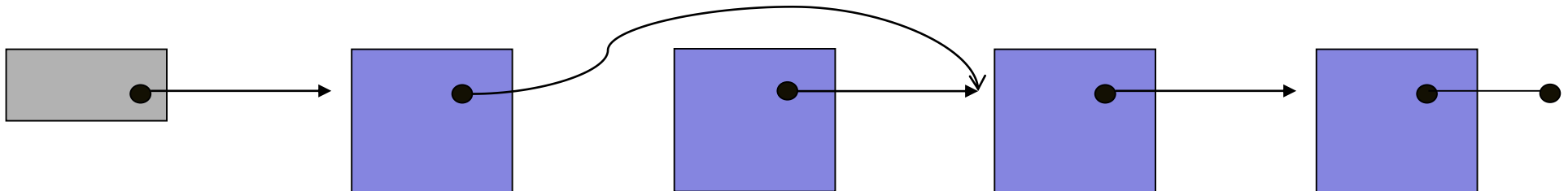
□ Einfügen an unbekannter Stelle ist  $O(n)$ .

➤ Warum? Position muss erst gesucht werden: Kosten  $O(n)$

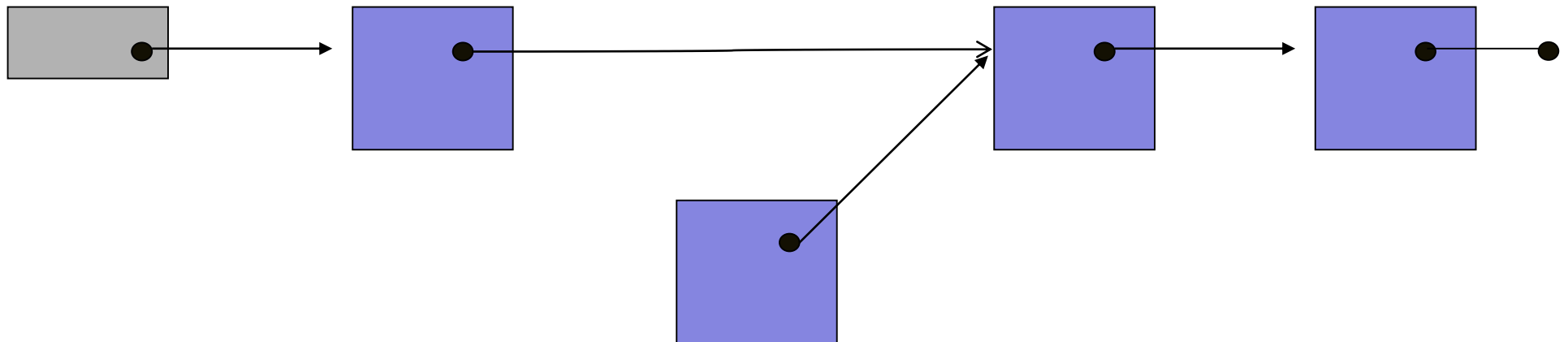
- Beim Entfernen wird ein Element, dessen Position bekannt ist, aus der Liste entfernt



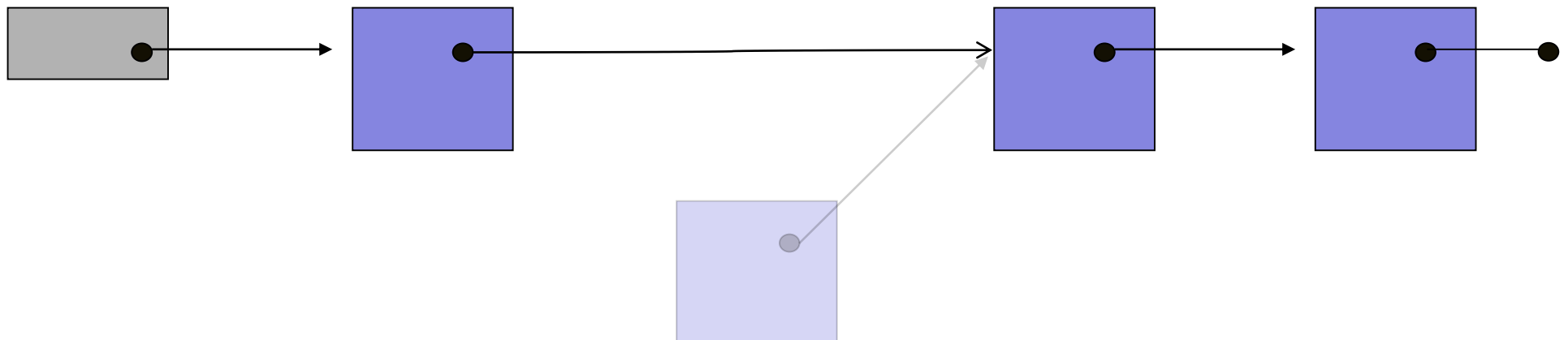
- Beim Entfernen wird ein Element, dessen Position bekannt ist, aus der Liste entfernt



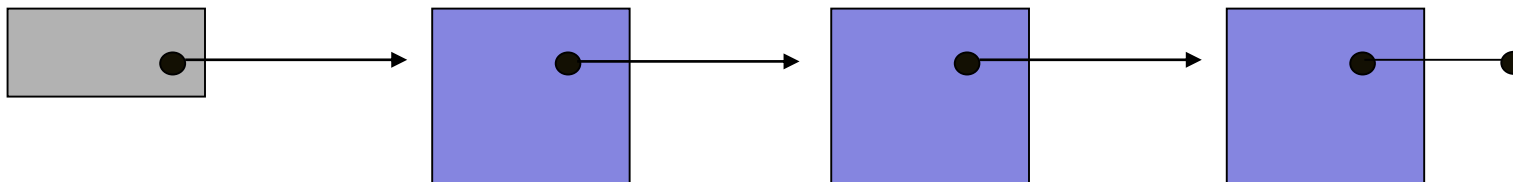
- Beim Entfernen wird ein Element, dessen Position bekannt ist, aus der Liste entfernt



- Beim Entfernen wird ein Element, dessen Position bekannt ist, aus der Liste entfernt



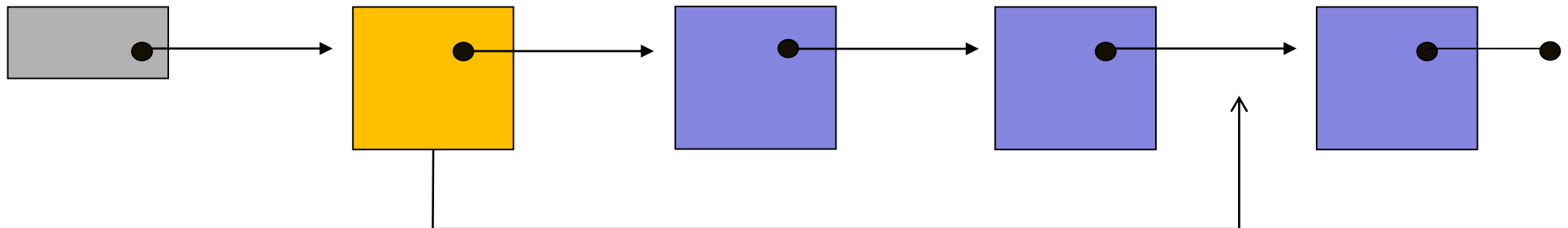
- Beim Entfernen wird ein Element, dessen Position bekannt ist, aus der Liste entfernt
- Laufzeit Entfernen bei bekannter Position:  $O(1)$
- Laufzeit Entfernen bei unbekannter Position:  $O(n)$





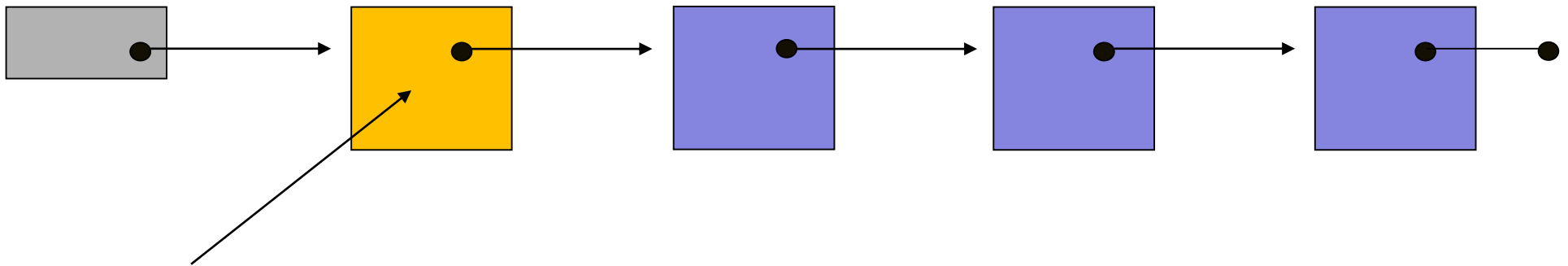
# Versetzen eines Elements

- ❑ Um bei Arrays die Reihenfolge der Elemente zu ändern, müssen alle Elemente dazwischen verschoben werden.
- ❑ Bei verketteten Listen werden Elemente einfach versetzt



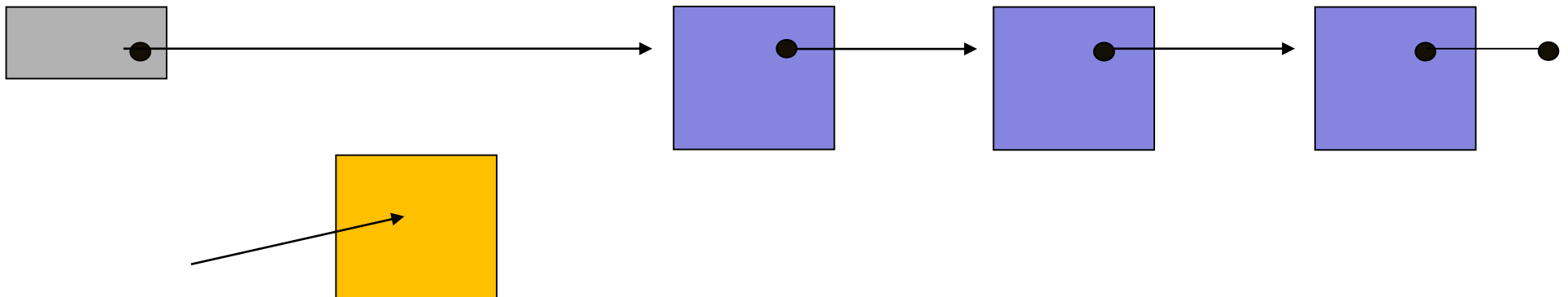
# Versetzen eines Elements

- ❑ Um bei Arrays die Reihenfolge der Elemente zu ändern, müssen alle Elemente dazwischen verschoben werden.
- ❑ Bei verketteten Listen werden Elemente einfach versetzt



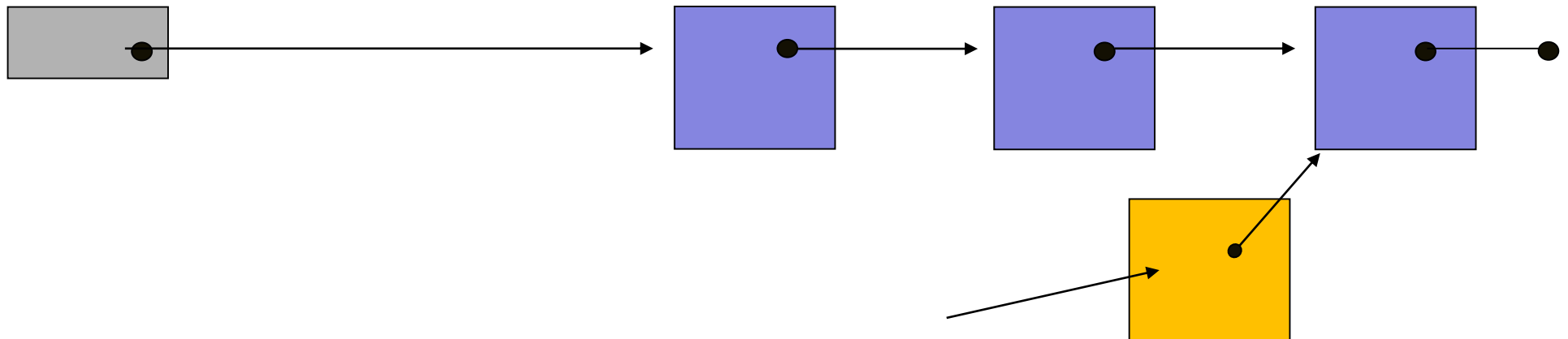
# Versetzen eines Elements

- ❑ Um bei Arrays die Reihenfolge der Elemente zu ändern, müssen alle Elemente dazwischen verschoben werden.
- ❑ Bei verketteten Listen werden Elemente einfach versetzt



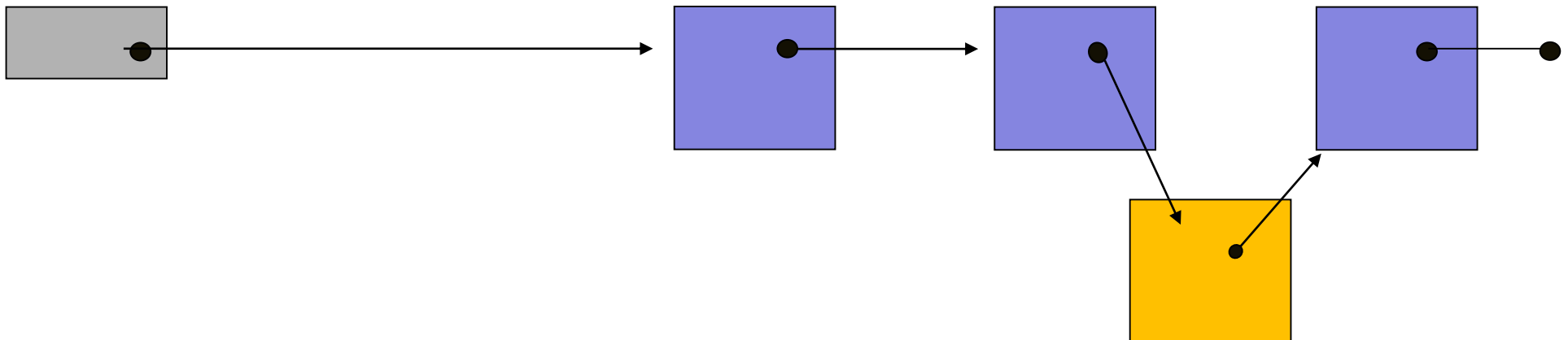
# Versetzen eines Elements

- ❑ Um bei Arrays die Reihenfolge der Elemente zu ändern, müssen alle Elemente dazwischen verschoben werden.
- ❑ Bei verketteten Listen werden Elemente einfach versetzt



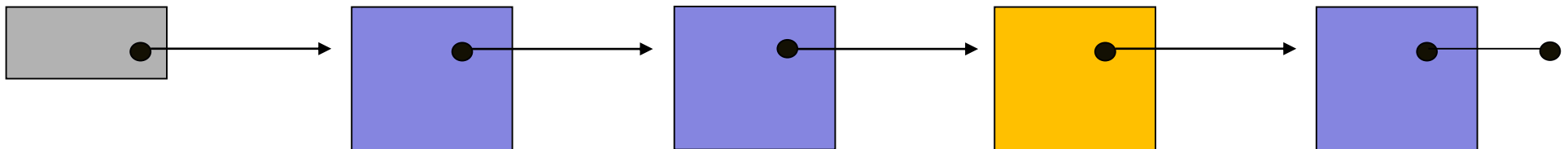
# Versetzen eines Elements

- ❑ Um bei Arrays die Reihenfolge der Elemente zu ändern, müssen alle Elemente dazwischen verschoben werden.
- ❑ Bei verketteten Listen werden Elemente einfach versetzt



# Versetzen eines Elements Laufzeit

- ❑ Um bei Arrays die Reihenfolge der Elemente zu ändern, müssen alle Elemente dazwischen verschoben werden.
- ❑ Bei verketteten Listen werden Elemente einfach versetzt
- ❑ **Laufzeit bei bekannten (Pointern an) Positionen:  $O(1)$**   
(statt  $O(k)$ ,  $k$  Anzahl der Stellen, bei Arrays)



# Linked Lists

## Implementierungsvarianten

# Linked List Implementierung Wiederholung

- ❑ Zahlreiche Variationsmöglichkeiten bei der Implementierung.
  - “Beste” Variante hängt vom Problem ab.
- ❑ Minimale Implementierung:
  - head (Kopfzeiger): Zeigt auf das erste Element der Liste.
  - Methoden / Funktionen:
    - Find element.
    - Insert element.
    - Delete element.

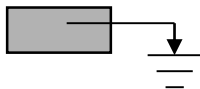


# Implementierungsvarianten

## Einfacher Kopfzeiger

- ❑ head zeigt auf das erste Element.
- ❑ Das Ende der Folge ist durch einen leeren Zeiger (Wert = null) gekennzeichnet.
  - Die leere Folge wird dann durch einen leeren Kopfzeiger repräsentiert.
- ❑ Leere Folge:

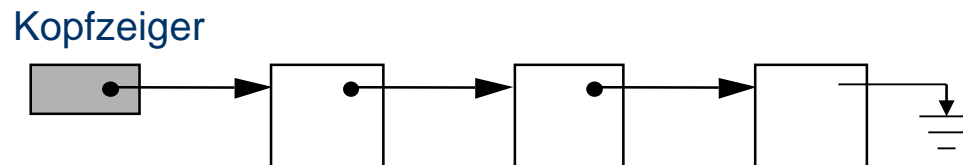
Kopfzeiger



# Implementierungsvarianten

## Einfacher Kopfzeiger

- ❑ head zeigt auf das erste Element.
- ❑ Das Ende der Folge ist durch einen leeren Zeiger (Wert = null) gekennzeichnet.
  - Die leere Folge wird dann durch einen leeren Kopfzeiger repräsentiert.
- ❑ Nichtleere Folge:



# Implementierungsvarianten

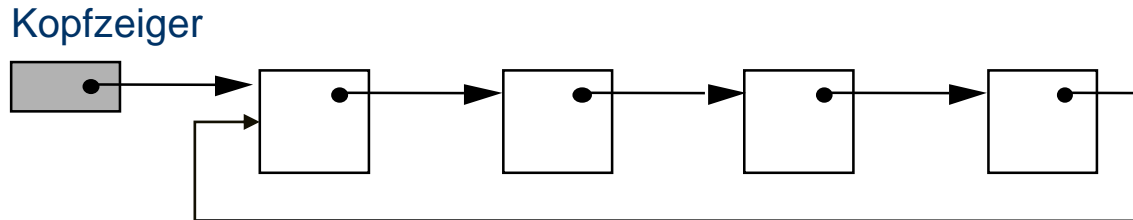
## Einfacher Kopfzeiger

- ❑ head zeigt auf das erste Element.
- ❑ Das Ende der Folge ist durch einen leeren Zeiger (Wert = null) gekennzeichnet.
  - Die leere Folge wird dann durch einen leeren Kopfzeiger repräsentiert.
- ❑ Hinweis zur Implementierung:
  - head muss immer auf das erste Element zeigen (Beachte z.B. insert und delete)

# Implementierungsvarianten

## Zyklische Verkettung

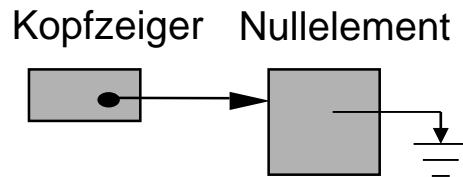
- ❑ Man kann die Zeigerkette schließen
  - Das erleichtert das Ablaufen in vielen Fällen.
- ❑ Nichtleere Liste:



# Implementierungsmöglichkeiten

## Kopfzeiger mit Nullelement (dummy)

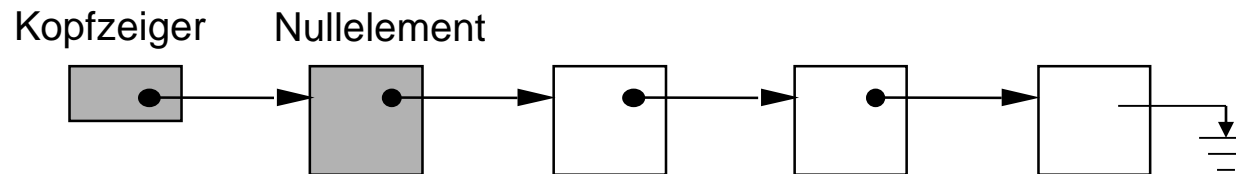
- Neben dem Kopfzeiger wird ein Nullelement verwendet, das dynamisch erzeugt ist und sich daher nicht von den anderen Elementen unterscheidet.
- Leere Folge:



# Implementierungsmöglichkeiten

## Kopfzeiger mit Nullelement (dummy)

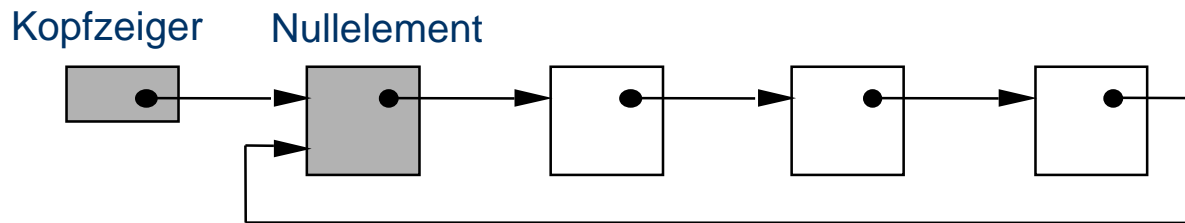
- Neben dem Kopfzeiger wird ein Nullelement verwendet, das dynamisch erzeugt ist und sich daher nicht von den anderen Elementen unterscheidet.
- Nicht leere Folge:



# Implementierungsmöglichkeiten

## Kopfzeiger mit Nullelement (dummy), zyklisch verkettet

- ❑ Neben dem Kopfzeiger wird ein Nullelement verwendet, jedoch mit zyklischer Verkettung.
- ❑ Nicht leere Folge



- ❑ Auf diese Weise lassen sich viele Listenoperationen recht kompakt und elegant formulieren.

# Einfache Datenstrukturen

- ☐ Arrays
  - ☐ Verkettete Listen
  - ☐ **Doppelt verkettete Listen**
  - ☐ Stack (Stapel)
  - ☐ Queue (Warteschlangen)
- 
- ☐ Warum: Grundlagen für alle weiteren Algorithmen

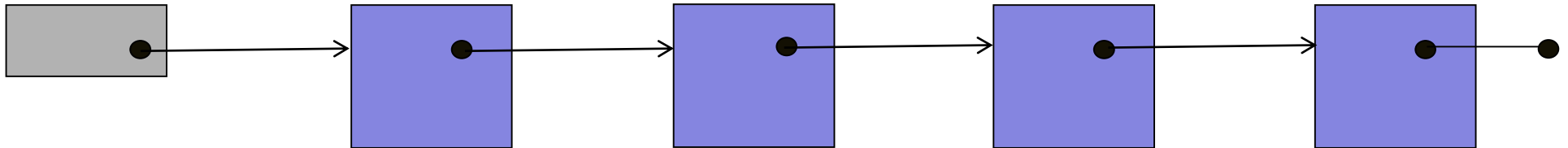


# Doppelt Verkettete Liste

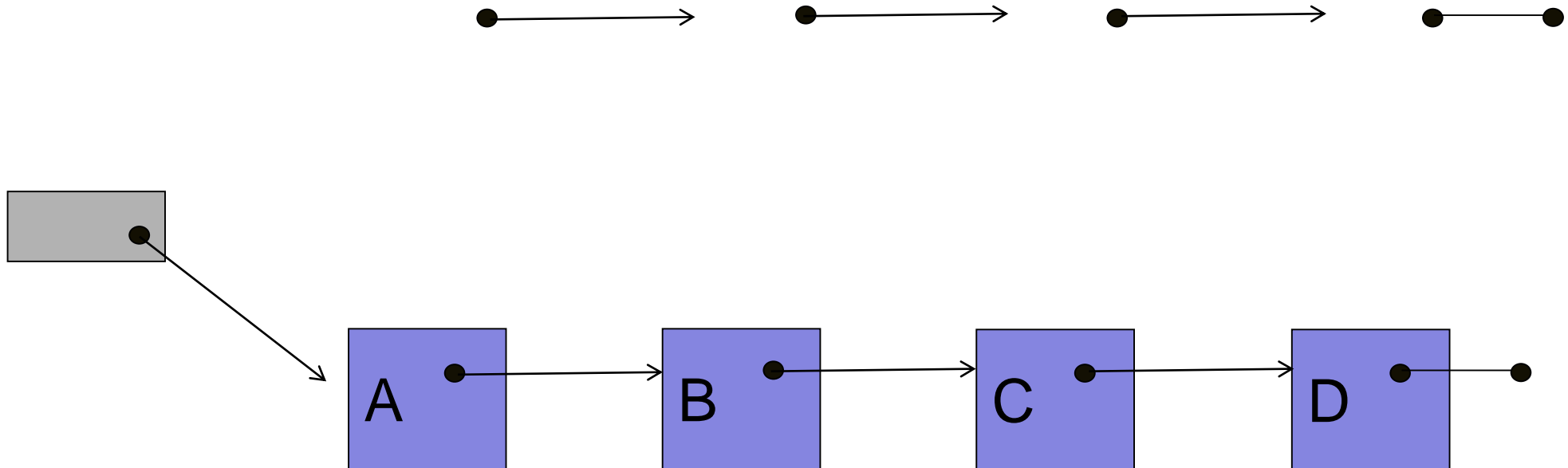
Fortgeschrittene Methoden einer Linked List

- ❑ Einfach verkettete Folgen können nicht in umgekehrter Richtung durchlaufen werden.
- ❑ Dazu kann bei Bedarf eine Operation Invertieren angeboten werden.
  - Zweimaliges Invertieren muss wieder die ursprüngliche Folge ergeben.

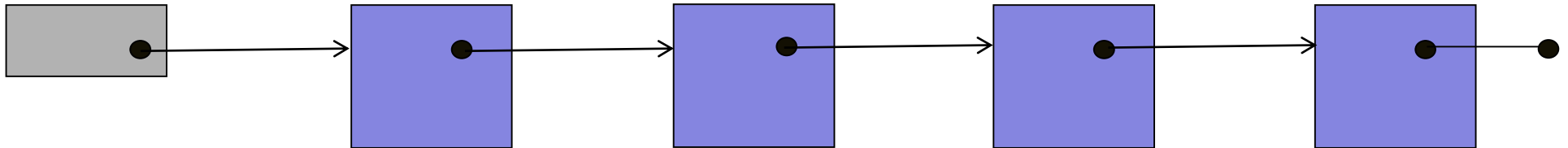
# Invertieren



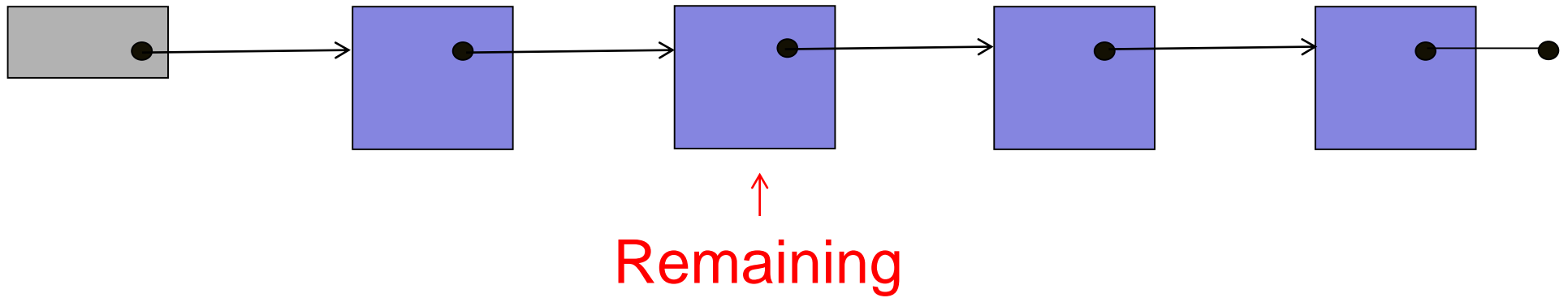
# Invertieren



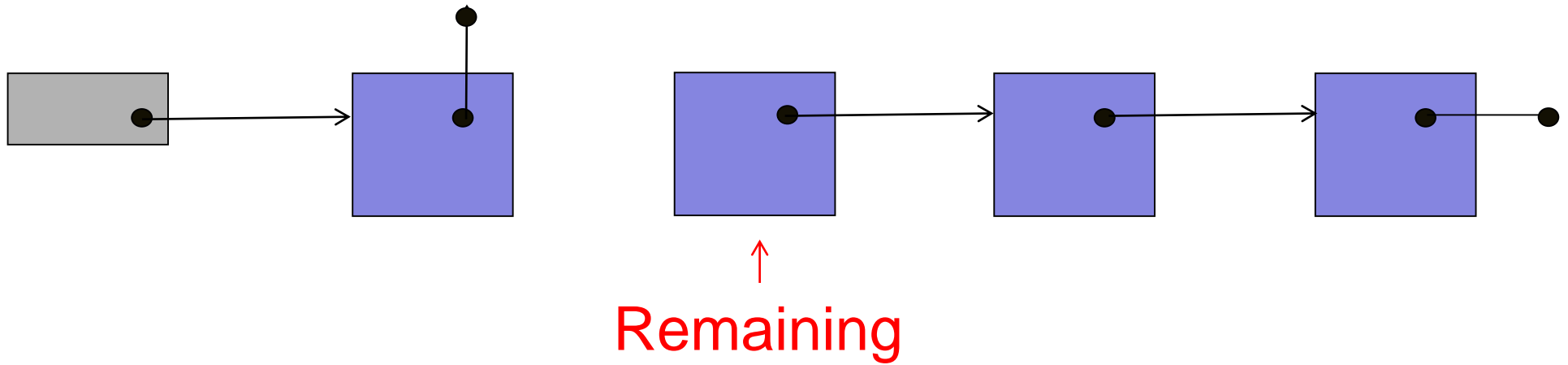
# Invertieren



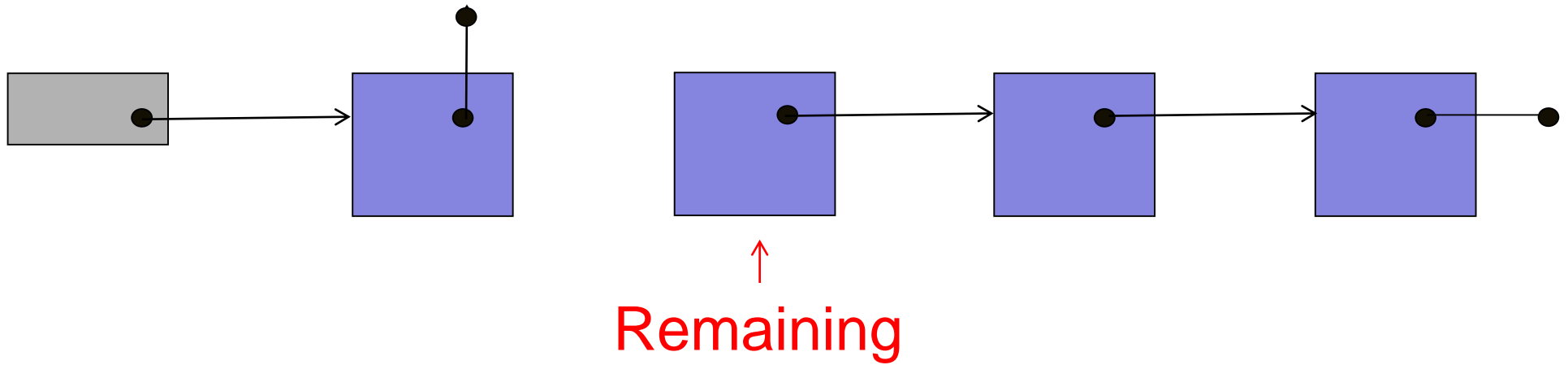
# Invertieren



# Invertieren

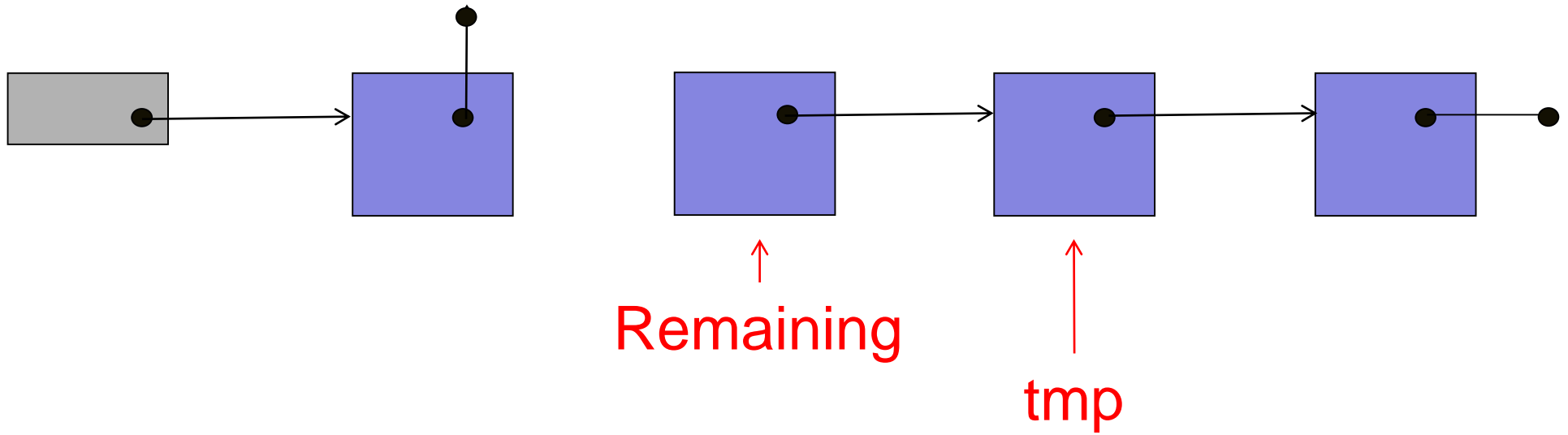


# Invertieren

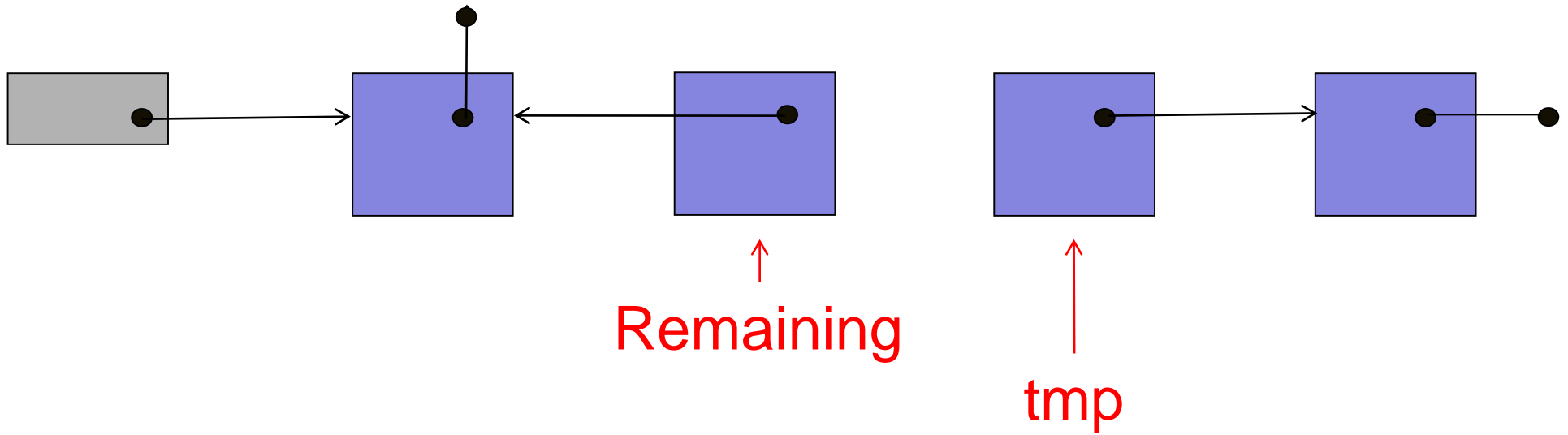




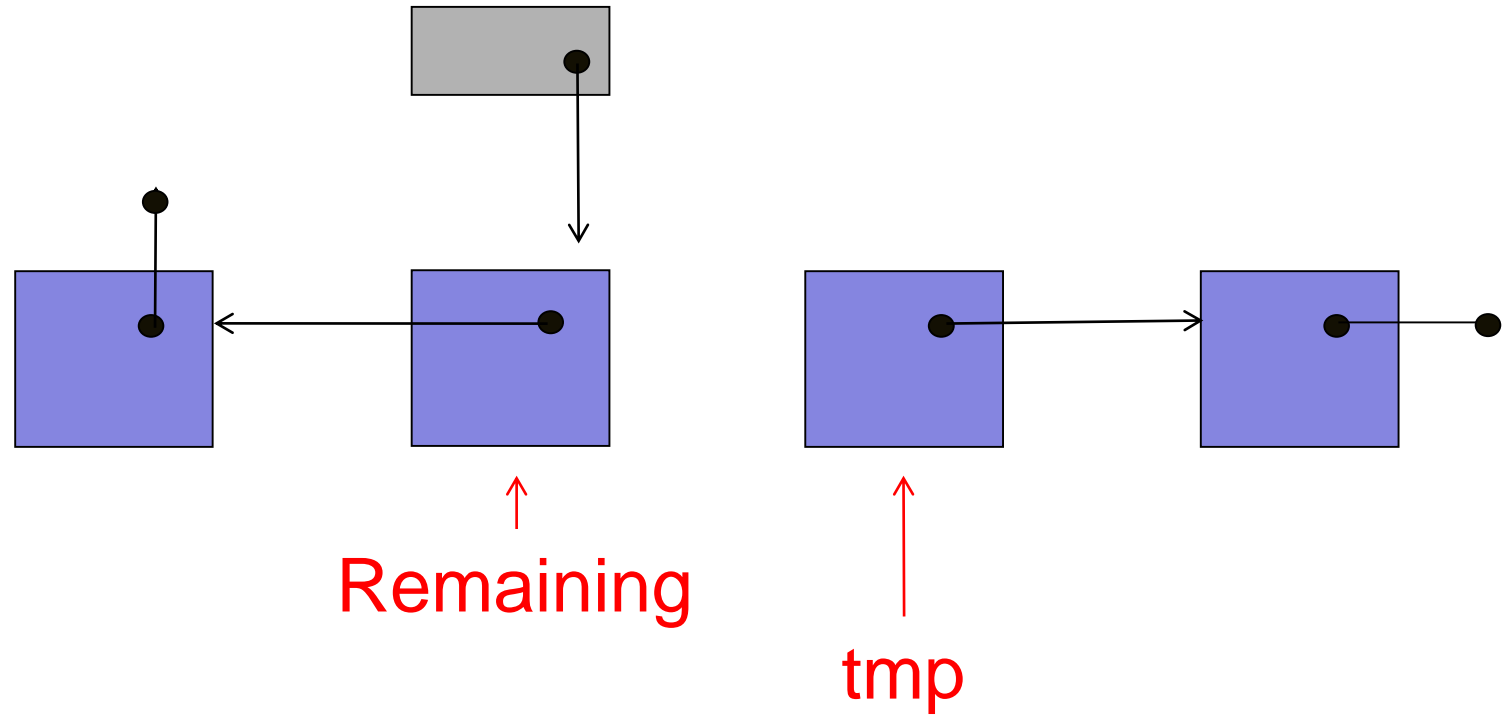
# Invertieren



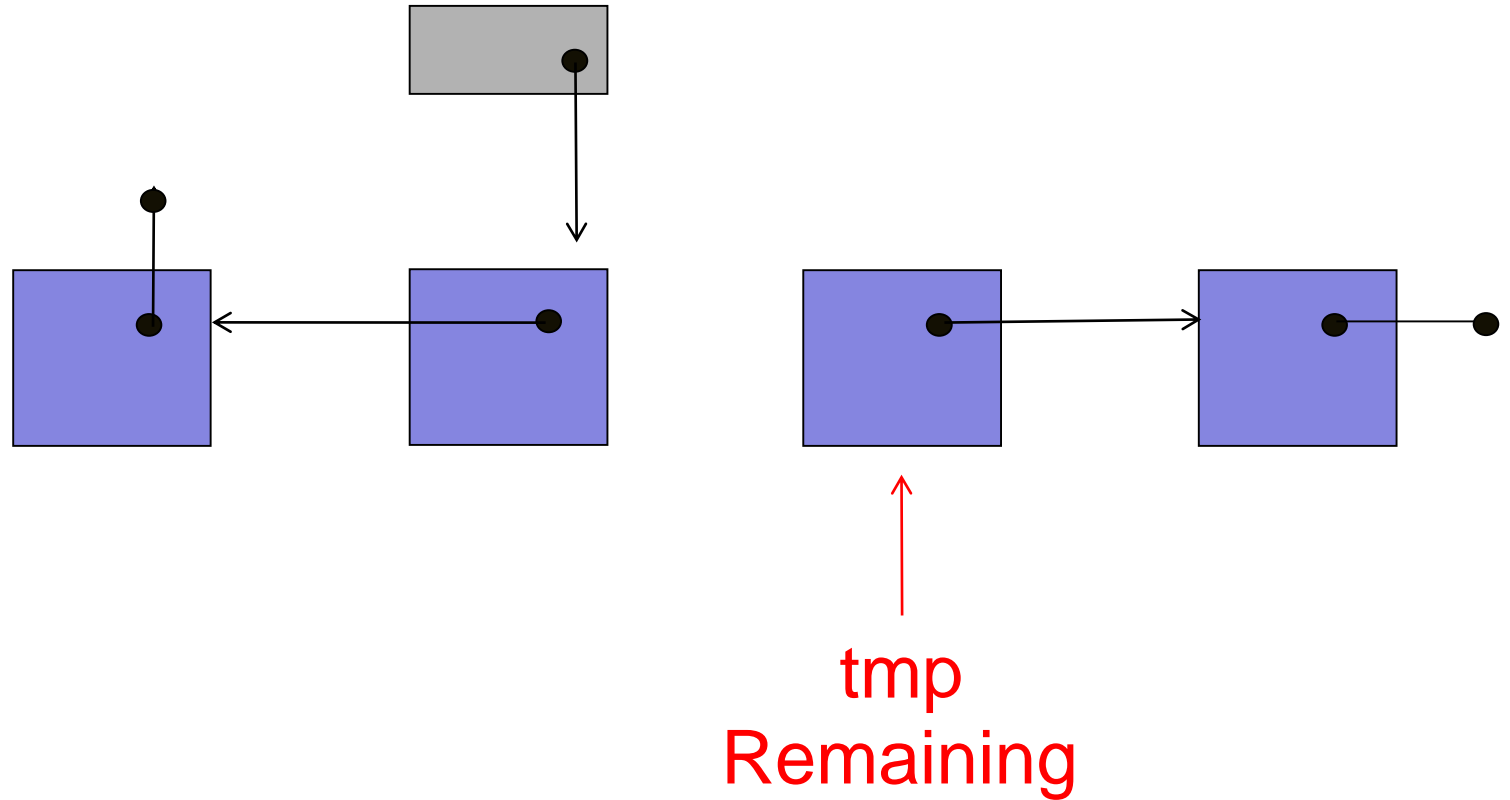
# Invertieren



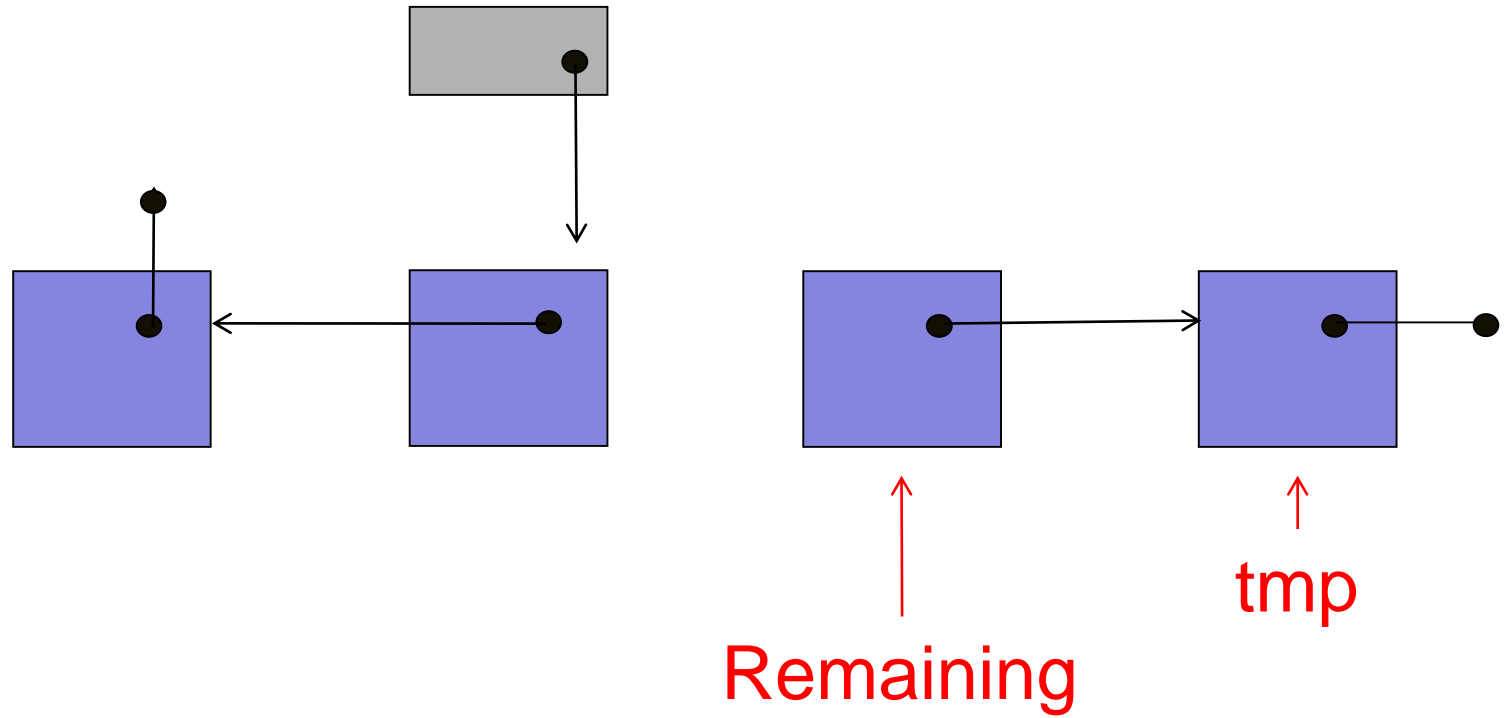
# Invertieren



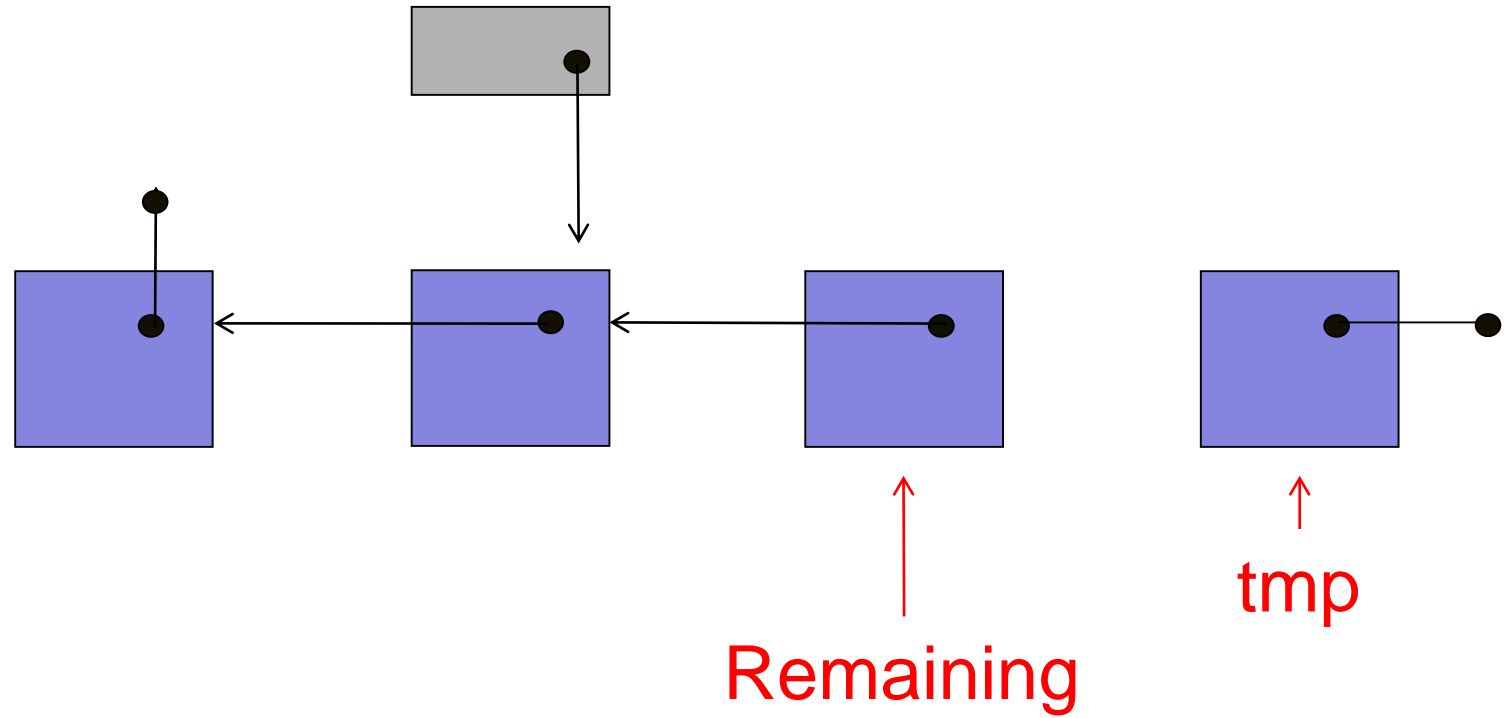
# Invertieren



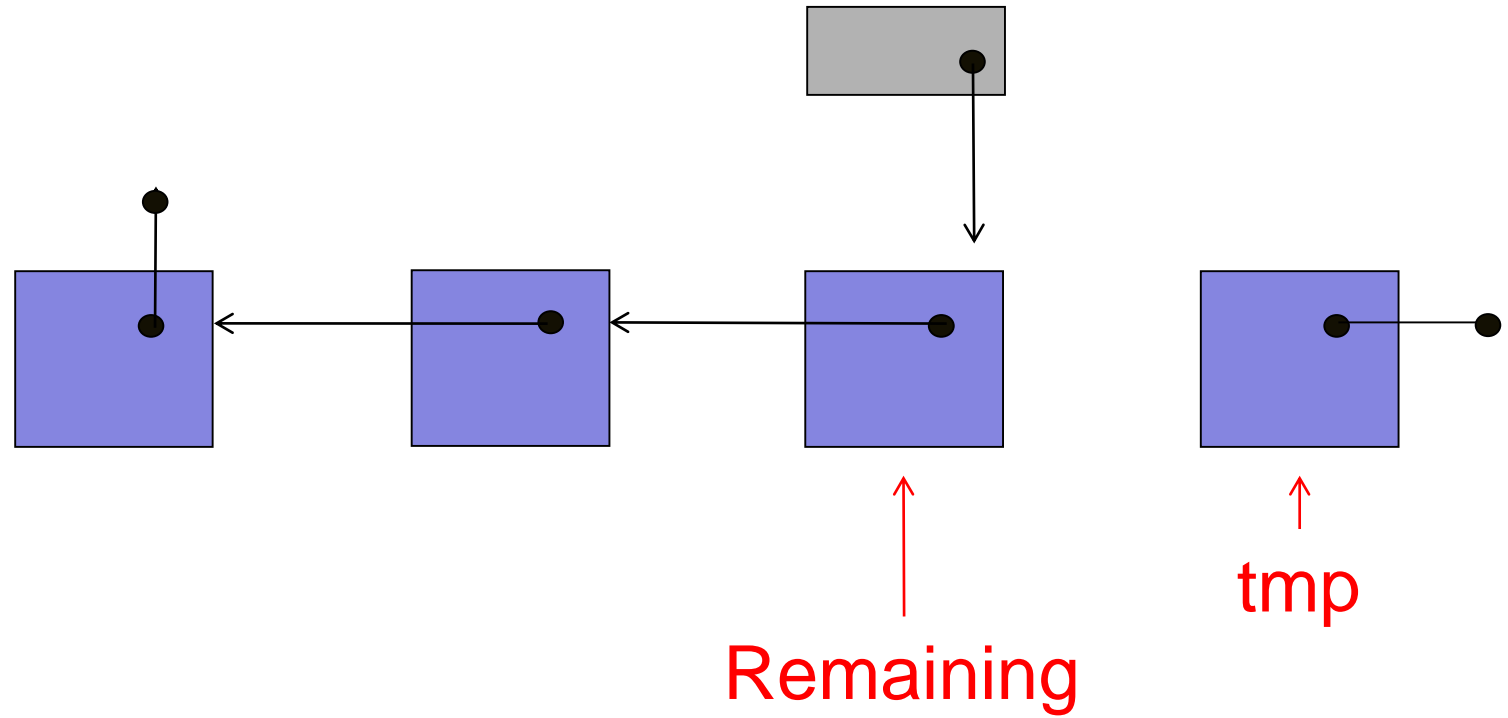
# Invertieren



# Invertieren



# Invertieren



# Double Linked List

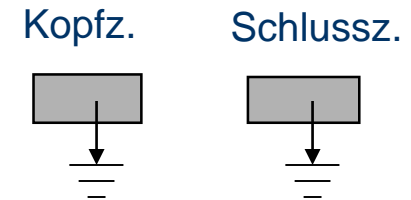
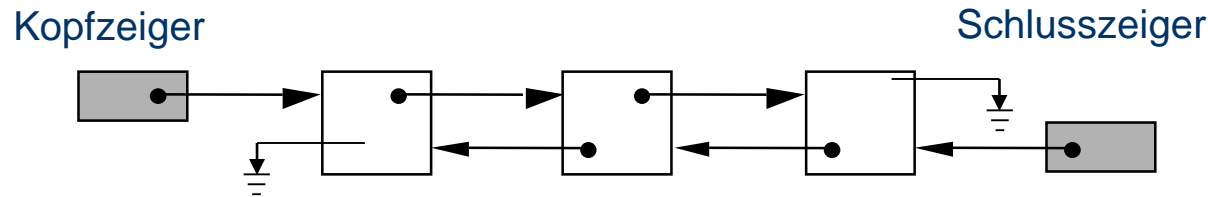
- ❑ Soll eine Folge in beiden Richtungen effizient abgelaufen werden können, so ist eine Doppelverzeigerung erforderlich.



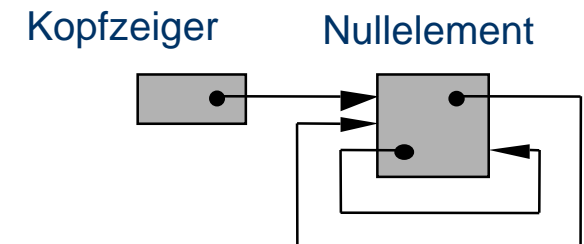
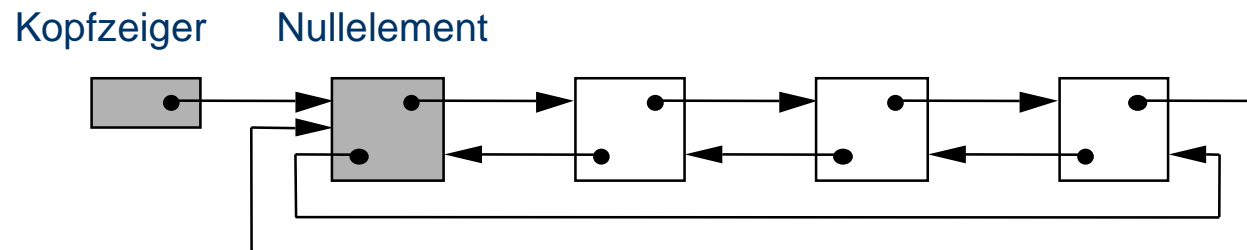
# Double Linked List

## Implementierungsvarianten

### ➤ Kopfzeiger ohne Nullelement, ohne Zyklus



### ➤ Kopfzeiger auf Nullelement, zyklisch verkettet



# Double Linked List

## □ Merkmale:

- Teilweise schnellere Laufzeit, z.B. Wenn Pointer auf Listenposition vorhanden: delete, insert in  $O(1)$
- Durchlaufrichtung frei wählbar (invertieren nicht notwendig)
- Aber: Erhöhter Speicherbedarf für Sekundärdaten
  - Nachfolger und Vorgänger
  - next und previous Pointer

# Einfache Datenstrukturen

- ☐ Arrays
- ☐ Verkettete Listen
- ☐ Doppelt verkettete Listen
- ☐ Stack (Stapel)
- ☐ Queue (Warteschlangen)
  
- ☐ Warum: Grundlagen für alle weiteren Algorithmen

# Vergleich der Eigenschaften

	Array	Dynamic Array	Linked List
Element Access	$O(1)$	$O(1)$	$O(n)$
Insert at begin	$O(n)$	$O(n)$	$O(1)$
Insert at end	$O(1)$	$O(1)$	$O(1)$
Insert at known position	$O(n)$	$O(n)$	$O(1)$
Extra space	0	$O(n)$	$O(n)$

After: [http://en.wikipedia.org/wiki/Linked\\_list](http://en.wikipedia.org/wiki/Linked_list)

# Einfache Datenstrukturen

- ☐ Arrays
  - ☐ Verkettete Listen
  - ☐ Doppelt verkettete Listen
  - ☐ **Stack (Stapel)**
  - ☐ Queue (Warteschlangen)
- 
- ☐ Warum: Grundlagen für alle weiteren Algorithmen

# Stacks und Queues Bauen auf Linked List auf....

# Stapel (stack)

- ❑ Stack: Datenstruktur, welche effizientes Entfernen in der umgekehrten Einfügereihenfolge ermöglicht.
  - Last-In-First-Out  
First-In-Last-Out (FILO) Datenstruktur.
  - Methoden: **push/pop**
- ❑ Einsatzbereiche:
  - Tellerstapeln
  - Browser Historie
  - Funktionsaufrufe! U.a. Bei der Rekursion
  - Kartenstapel

# Stapel (stack)

- ❑ Stack: Datenstruktur, welche effizientes Entfernen in der umgekehrten Einfügereihenfolge ermöglicht.
  - Last-In-First-Out  
First-In-Last-Out (FILO) Datenstruktur.
  - Methoden: **push/pop**
- ❑ Die Implementierung kann als Liste erfolgen:
  - **push** fügt am Kopf ein neues Element ein
  - **pop** entnimmt das als letztes eingefügte Element
  - Beide Operationen können in  $O(1)$  durchgeführt werden.
  - head ist gleichzeitig "top of stack".

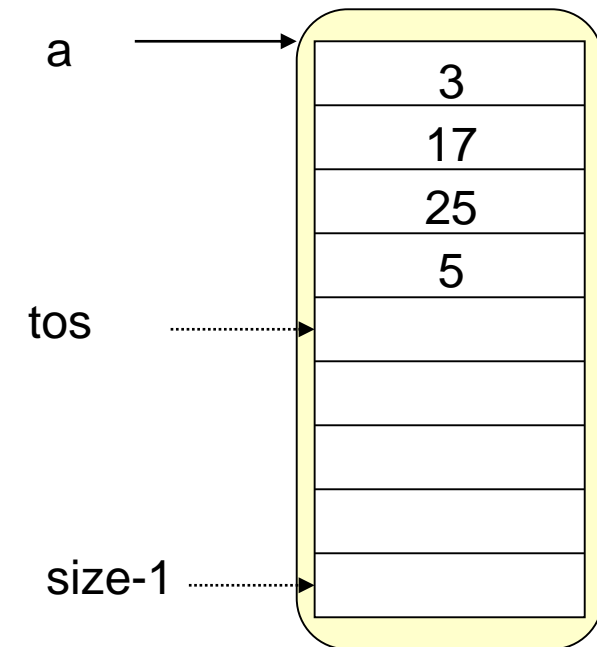


# Stapel (stack)

## Implementierung als Array

- ❑ Auch in einem Array können push und pop mit Laufzeit  $O(1)$  implementiert werden.
- ❑ Nachteil: Max. Stackgröße ist fest, d.h. der Stapel kann "überlaufen" (stack overflow).
  - Zugriff auf Element kann jedoch effizienter sein, als mit Linked List.

Implementierung:  
(nach außen nicht sichtbar)

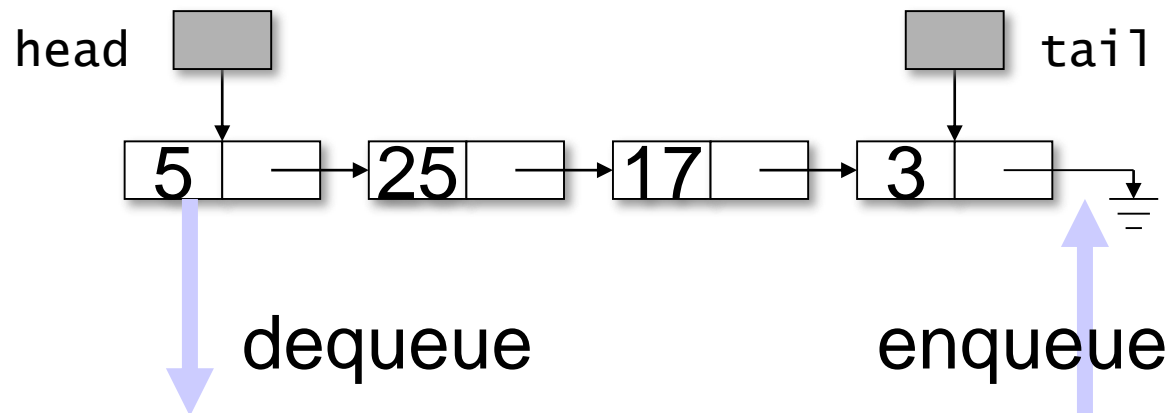


# Warteschlange (queue)

- ❑ Queue: Datenstruktur, welche effizientes Entfernen in der Einfügereihenfolge ermöglicht.
  - First-In-First-Out (FIFO) Datenstruktur.
  - Zugriff durch Methoden `enqueue` / `dequeue` realisiert.
- ❑ Einsatzbereiche:
  - Mensa
  - Verwaltung
  - Druckjobs
  - Wursttheke
  - ...

# Warteschlange (queue)

- ❑ Queue: Datenstruktur, welche effizientes Entfernen in der Einfügereihenfolge ermöglicht.
  - First-In-First-Out (FIFO) Datenstruktur.
  - Zugriff durch Methoden `enqueue` / `dequeue` realisiert.
- ❑ Implementierung durch Linked List möglich:

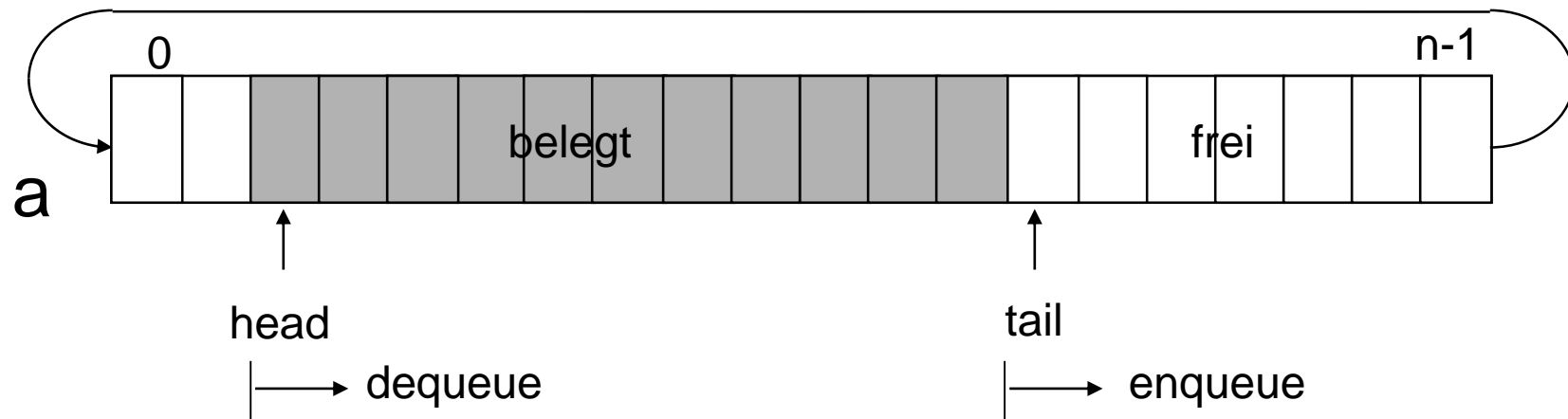


# Warteschlange (queue)

- ❑ Queue: Datenstruktur, welche effizientes Entfernen in der Einfügereihenfolge ermöglicht.
  - First-In-First-Out (FIFO) Datenstruktur.
  - Zugriff durch Methoden `enqueue` / `dequeue` realisiert.
- ❑ Implementierung durch Linked List möglich:
  - Auch hier gelingt die Implementierung von `enqueue` und `dequeue` mit konstanter Laufzeit  $O(1)$ .

# Warteschlange (queue)

- ❑ Array-Implementierung der Warteschlange üblicherweise als Ringpuffer (ring buffer) realisiert.
- ❑ Es werden zwei Zeiger (Indizes) mitgeführt, die auf den Anfang bzw. das Ende der Warteschlange zeigen.



# Warteschlange (queue)

- ❑ Array-Implementierung der Warteschlange üblicherweise als Ringpuffer (ring buffer) realisiert.
- ❑ Es werden zwei Zeiger (Indizes) mitgeführt, die auf den Anfang bzw. das Ende der Warteschlange zeigen.
  - Nachteil: Max. Größe ist fest.
  - Zugriff auf Elemente kann effizienter sein als mit Linked Lists.