

# Inhalt

- 4 Datenstruktur Verkettete Liste
  - Einfach verkettete Liste (EVL)
  - EVL - Implementierung
  - Durchlauf durch verkettete Listen
  - Varianten verketteter Listen

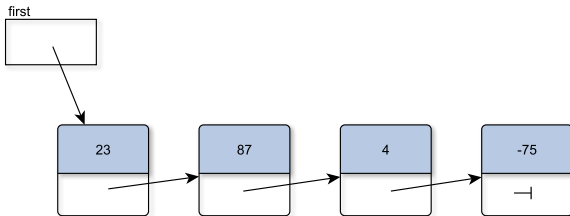
# Verkettete Strukturen

Alternativer Ansatz (zur Verwaltung einer „großen“ Anzahl an Datensätzen) **ohne Verwendung von Arrays**:

- ▶ Es wird eine „Kette (Liste)“ von Objekten aufgebaut: die Objekte bestehen jeweils aus zwei Komponenten:
  - einer Referenz auf den eigentlichen Datensatz
  - einer Referenz auf das nächste Kettenglied (Listenelement)
- ▶ Benötigt eine **innere Klasse**, die diese Kettenglieder (Listenelemente) kapselt.
- ▶ Die Liste verlängert/verkürzt sich **bei jedem** Einfügen bzw Entfernen eines Elementes.
- ▶ Die Datenstruktur zur Darstellung der „Menge“ benötigt (nur) eine Referenz auf das **erste** Listenelement.

# Schematische Darstellung

häufig hilfreich, manchmal unabdingbar (!)



wichtig: unterscheide zwischen

- ▶ Referenz auf Listenelement ( first )
- ▶ Listenelement: „Doppelpack“ aus (Referenz auf) Datensatz und Referenz auf nächstes Element

im gezeigten Beispiel: statt Referenzen auf Datensätze sind int-Konstanten eingetragen

# Inhalt

- 4 Datenstruktur Verkettete Liste
  - Einfach verkettete Liste (EVL)
  - EVL - Implementierung
  - Durchlauf durch verkettete Listen
  - Varianten verketteter Listen

## EVL<T> in „Pseudo-UML“

- ▶ definiert eine Referenz auf den Listen „Anfang“
- ▶ definiert eine „innere“ Klasse zur Darstellung der Listenelemente
- ▶ definiert Methoden zum Einfügen, Entfernen, Zugriff auf Elemente
- ▶ (später auch noch weitere Methoden)

EVL<T>
ListenElem first
InnerClass ListenElem
size(): int isEmpty(): boolean get(): T insert(T): void remove(): void

# Die innere Klasse

- ▶ kapselt die Implementierung eines Listenelements mit seinen beiden Attributen `value` und `next`
- ▶ kann **innerhalb** der Klasse `EVL<T>` definiert werden, weil sie nur hier gebraucht wird

```
class ListElem {  
    T value;  
    ListElem next;  
  
    ListElem (T v) {  
        value = v;  
        next = null;  
    }  
}
```

## size, isEmpty und get

- ▶ size() liefert die **Anzahl** der Listenelemente
- ▶ isEmpty() prüft, ob die Liste leer ist
- ▶ get() liefert den **Wert** des ersten Listenelements, löst eine Exception aus, falls die Liste leer ist

```
public int size() { ... }
```

```
public boolean isEmpty() { ... }
```

```
public T get() throws NoSuchElementException {  
    if (first == null)  
        throw new NoSuchElementException("Liste leer");  
    return first.value;  
}
```

# Implementierung von insert

- ▶ `insert(T v)` erzeugt ein neues Listenelement mit **Wert** `v` und fügt es **vorne** in die Liste ein
- ▶ das bisherige erste Element wird dann das zweite in der Liste (also Nachfolger des neuen Elements)

```
public void insert(T v) {  
    ListenElem neu = new ListenElem(v);  
    neu.next = first;  
    first = neu;  
    size++;  
}
```



# Implementierung von remove

- ▶ `remove()` entfernt das **erste** Element der Liste
- ▶ das bisherige zweite Element wird dann zum ersten
- ▶ falls die Liste (vorher) leer ist, hat `remove()` keine Wirkung

```
public void remove() {  
    if (first == null)  
        return;  
    first = first.next;  
    size--;  
}
```

## Beispiel: Implementierung von delete

- ▶ `delete(T v)` entfernt das Element `e`, dessen Eintrag  $= v$  ist
  - falls mehrere, dann das erste vorkommende Element mit Eintrag `v`
  - wenn es kein solches Element gibt, dann keine Wirkung
- ▶ dazu notwendig: ein Element mit Eintrag `v` in der Liste finden
- ▶ dann: „Umketten“ der `next`-Referenz im Vorgänger von `e` auf den Nachfolger von `e`
- ▶ zu überlegen: wann ist Fallunterscheidung erforderlich?
  - Liste leer
  - Liste enthält kein Element mit Eintrag `v`
  - `e` ist erstes Element der Liste
  - `e` ist innerhalb der Liste
  - `e` ist letztes Element der Liste
  - ... ?

→ Code-Beispiel; [Speicherbilder malen!](#)

# Inhalt

- 4 Datenstruktur Verkettete Liste
  - Einfach verkettete Liste (EVL)
  - EVL - Implementierung
  - **Durchlauf durch verkettete Listen**
  - Varianten verketteter Listen

# Durchlaufen einer verketteten Liste

## Durchlauf

- ▶ darf die Struktur (Verkettung der Listenelemente) nicht ändern
- ▶ benötigt ein Referenzelement, das sukzessive auf die verschiedenen Listenelemente „zeigt“

Beispiel: Iterative Lösung für toString()

```
public String toString() {  
    String s = "";  
  
    ListElem tmp = first;  
    while (tmp != null) {  
        s += tmp.value + " ";  
        tmp = tmp.next;  
    }  
  
    return s;  
}
```

# Rekursion und Verkettung

**Rekursion:** beliebt bei „Pseudocode“-Darstellung von Algorithmen, die auf verketteten Strukturen arbeiten.

In der Praxis eher vermeiden, wenn es sich vermeiden lässt ;)

Prinzip:

- ▶ Rekursive Methode benötigt ein **ListenElement** als Eingabeparameter
- ▶ ... und wird rekursiv für das **nächste ListenElement** aufgerufen.
- ▶ Definiere zum „Einstieg“ eine (nicht-rekursive) parameterlose Methode, die die rekursive Methode für das **first-Element** aufruft.

# Bsp: Rekursives Durchlaufen einer verketteten Liste

Beispiel: Rekursive Lösung für toString()

```
// nicht-rekursive Einstiegsmethode
public String toString() {
    return rekToString(first);
}

// rekursive Methode
private String rekToString(ListenElem e) {
    if (e == null)
        return "";
    else
        return rekToString(e.next) + " " + e.value;
}
```

## Beispiel: contains(T e)

contains(T e) : Methode, die prüft, ob das Element e (das heißt: ein ListElem mit value e) in der Liste enthalten ist

- ▶ sehr unkompliziert rekursiv zu formulieren
- ▶ besser: iterativ zu implementieren

# Zusammenfassung EVL

## ► Laufzeit der Operationen:

- `size()`,
- `isEmpty()`,
- `get()`,
- `insert(T e)`,
- `remove()`

arbeiten in Zeit  $\mathcal{O}(1)$

- ## ►
- `contains(T e)`
  - `delete(T e)`

benötigen Zeit  $\mathcal{O}(n)$

## ► Bemerkung:

`insert(T e)` fügt das neue Element „vorne“ in die Liste ein, dh die Liste enthält die Elemente in **umgekehrter** Einfüge-Reihenfolge

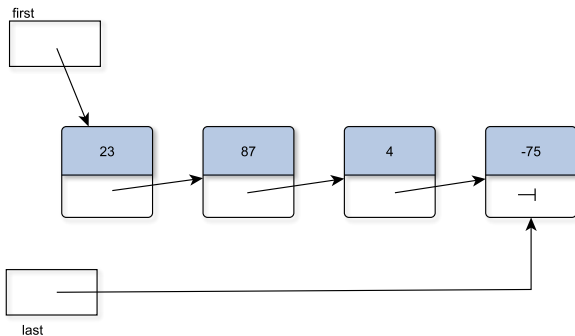


# Inhalt

- 4 Datenstruktur Verkettete Liste
  - Einfach verkettete Liste (EVL)
  - EVL - Implementierung
  - Durchlauf durch verkettete Listen
  - Varianten verketteter Listen

# EVL mit last-Zeiger - EVLL

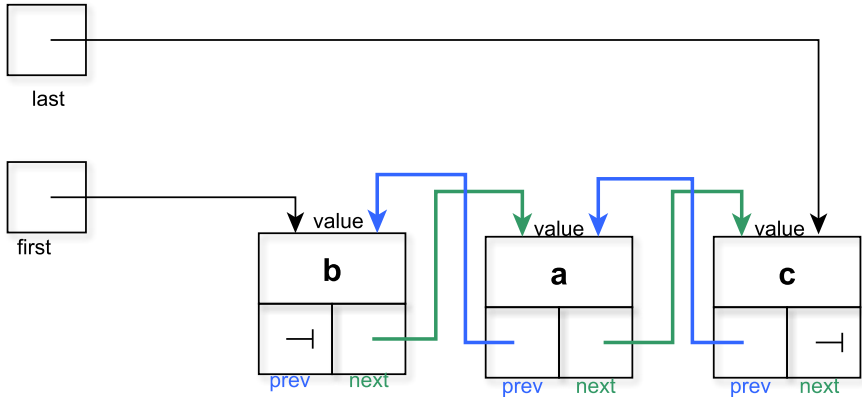
- ▶ wie EVL mit zusätzlichem Instanzattribut ListElem last
- ▶ ermöglicht eine Operation `append(T e)`, die ein Element an das **Ende** der Liste anfügt in Zeit  $\mathcal{O}(1)$



# Doppelt verkettete Liste - DVL

- ▶ Instanzattribute `ListenElem first` und `ListenElem last` wie bei EVLL
- ▶ `ListenElemente` haben neben dem `value` nun **zwei Referenzen** :
  - `next` verweist auf das **nachfolgende** Listenelement
  - `prev` verweist auf das **vorhergehende** Listenelement
- ▶ ermöglicht Operationen in Zeit  $\mathcal{O}(1)$ 
  - `insert(T e)`: vorne einfügen
  - `append(T e)`: hinten anhängen
  - `removeFirst()`: erstes entfernen
  - `removeLast()`: letztes entfernen
- ▶ ermöglicht zusätzlich in Zeit  $\mathcal{O}(pos)$  für  $0 \leq pos < size$ :
  - Vereinbarung für `pos`: `first` hat Nummer 0, `last` hat Nummer  $(size - 1)$
  - `get(int pos)`: **liefert** das `pos`-te Element
  - `set(int pos, T e)`: **ändert** das `pos`-te Element
  - `insert(int pos, T e)`: **fügt** das Element `e` an Position `pos` **ein**
  - `remove(int pos)`: **entfernt** das `pos`-te Element

# Beispiel: Speicherbild DVL



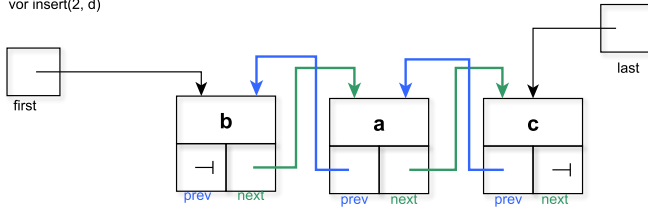
## Beispiel insert(2, d)

Einfügen eines neuen Elementes mit  $\text{value} = d$  an Position 2  
(dh: hinter Element mit pos-Nr 1)

- ▶ Erzeugen eines Listen-Elementes:  
`ListenElem neu = new ListenElem(d);`
- ▶ Erzeugen eines Listen-Elementes zum Durchlaufen der Liste:  
`ListenElem csr = first ;`
- ▶ in einer Schleife (wie oft?)  
`csr = csr.next;`
- ▶ „Umhängen“ der rot markierten Referenzen (siehe nächste Folie)
- ▶ wobei vorher(!) Sonderfälle zu beachten sind  
zB  $\text{pos} = 0$ ,  $\text{pos} = \text{size} \dots$

# Beispiel Speicherbild für insert(2, d)

vor insert(2, d)



nach insert(2, d)

