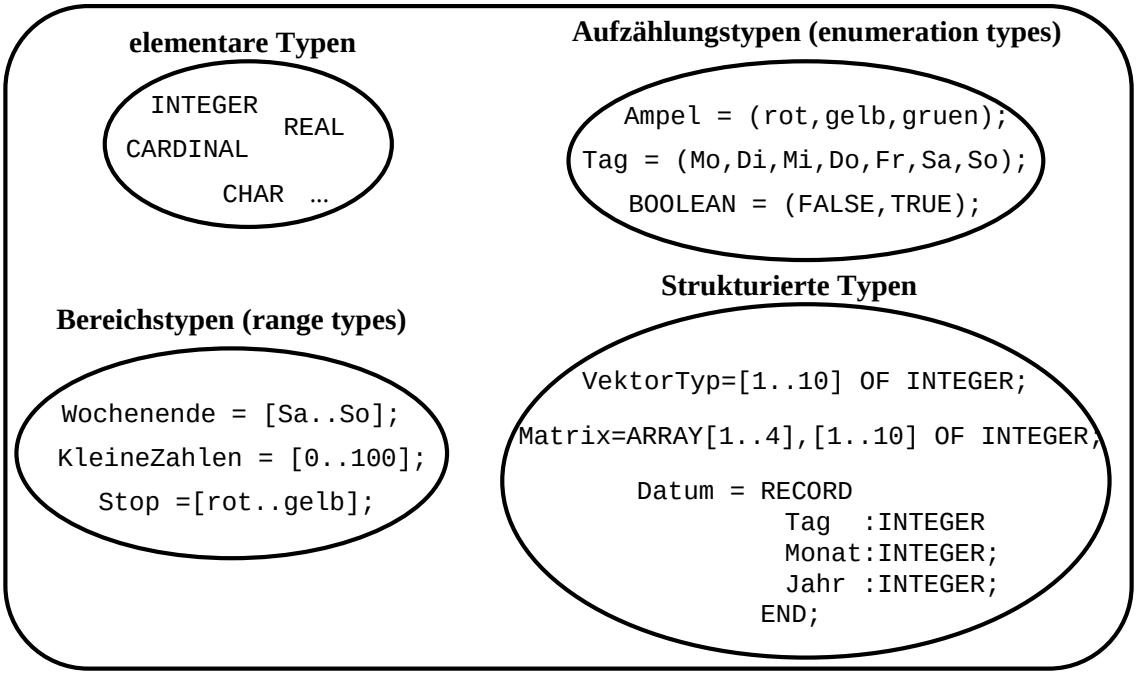
Strukturierte Datentypen - Überblick

Modula-2-Typen und benutzerdefinierte Typen



All diese Typen können bei der Definition strukturierter Datentypen verwendet werden.

Dynamische Datenstrukturen

Zeiger (Pointer) und dynamische Datenstrukturen

Mit Feldern (Array) und Verbunden (Record) ist es möglich komplexe Datenstrukturen zu definieren.

Allerdings sind diese Strukturen **statisch** (d.h. von fester Größe).

Zur Definition von Datenstrukturen, bei denen zur Laufzeit nicht nur die zugeordneten Werte, sondern auch der **Aufbau und die Größe variabel** sein sollen, braucht man **dynamische Strukturen**.

Beispiele: **Listen, Bäume** oder **Graphen**

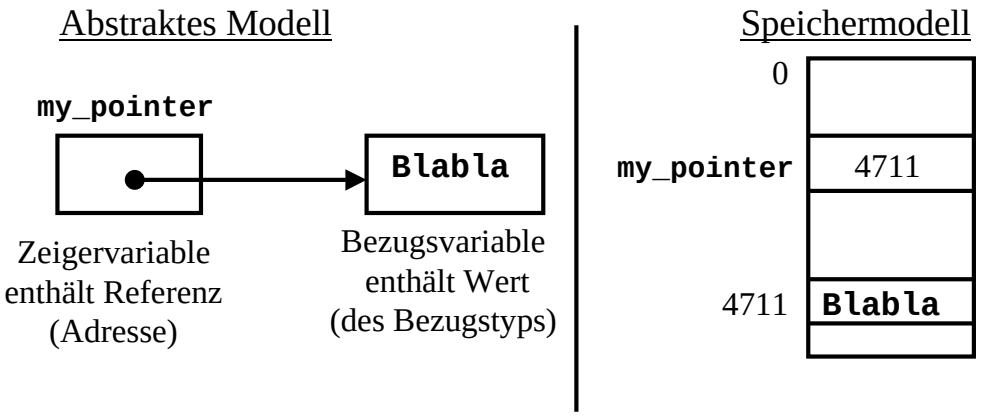
Modula-2 bietet ein einfaches Werkzeug zur Definition beliebiger dynamischer Datenstrukturen: den **Zeiger (Pointer)**

Zeiger (Pointer)

Was sind Zeiger ?

Der Wert (Inhalt) einer Zeigervariablen dient als **Referenz** auf eine Variable seines Bezugstyps (referenced type).
Ein Zeiger (Pointer) "**zeigt**" auf eine andere Variable.

Man kann den Wert einer Zeigervariable auch als **Anfangsadresse** eines bestimmten Speicherbereiches auffassen.



Zeiger (Pointer)

Formale Definition für einen Zeigertyp in Modula-2:

```
TYPE PointerType   = POINTER TO Type;
```

Beispiel:

```
TYPE string_pointer = POINTER TO string;
```

- **"Type"** stellt den **Bezugstyp**, also den Typ der referenzierten Variable (referenced variable), dar.
- Ein Zeiger (Pointer) kann in Modula-2 **nicht** auf Variablen beliebigen (unterschiedlichen) Typs zeigen.
  Dies soll die Programmsicherheit erhöhen, da nichttypisierte Zeigervariablen schwer lokalisierbare Fehler zur Folge haben können.

Zeiger (Pointer)

Wie werden Zeiger verwaltet ?

Beispiel:

Mit dem Befehl

```
VAR my_pointer : string_pointer;
```

wird eine **Zeigervariable** vom Typ `string_pointer` deklariert.

Die zugehörige **Bezugsvariable** (referenced variable) muss allerdings zur **Laufzeit** (also innerhalb des Programms) explizit erzeugt werden. Hierfür steht die Standardprozedur `NEW(...)` zur Verfügung:

```
NEW(my_pointer);
```

Dieser Befehl hat zur Folge, dass der entsprechende **Speicherplatz** für die benötigte **Bezugsvariable** zur Laufzeit **allokiert** wird. Zusätzlich wird in `my_pointer` die **Adresse** auf diesen Speicherplatz eingetragen.

```
DISPOSE(my_pointer);
```

gibt den allokierten Speicherbereich wieder frei, er wird **deallokiert**.

Zeiger (Pointer)

Die Operationen `NEW( )` bzw. `DISPOSE( )` werden in Aufrufe der Funktionen `ALLOCATE( )` bzw. `DEALLOCATE( )` übersetzt:

`NEW (p)` → `ALLOCATE(p,SIZE(XT))`

Mit `ALLOCATE(p,SIZE(XT))` wird für eine Variable des Typs `XT` ein Speicherbereich der Größe `SIZE(XT)` angefordert, und dessen **Anfangsadresse** wird in `p` hinterlegt.

`DISPOSE(p)` → `DEALLOCATE(p,SIZE(XT))`

`DEALLOCATE(p,SIZE(XT))` bewirkt, dass der Speicherbereich mit der Länge `SIZE(XT)` ab der Adresse `p` freigegeben wird.
Da Pointer immer einem bestimmten Typ `XT` zugeordnet sind, kann man direkt aus dem Pointertyp die Länge `SIZE(XT)` bestimmen.

Zeiger (Pointer)

Hierzu muss man allerdings das Modul **`Storage`** importieren, da die Funktionen **`ALLOCATE( )`** und **`DEALLOCATE( )`** dort definiert sind.

```
FROM Storage IMPORT ALLOCATE,DEALLOCATE;
```


Zeiger (Pointer)

Die Variablen, auf die **`my_pointer`** zeigt, ist **anonym** (hat also keinen Namen) und kann **nur indirekt** über den Zeiger **`my_pointer`** angesprochen werden!

Es ist **nicht** möglich, die Bezugsvariable **ohne** den zugehörigen Zeiger anzusprechen.

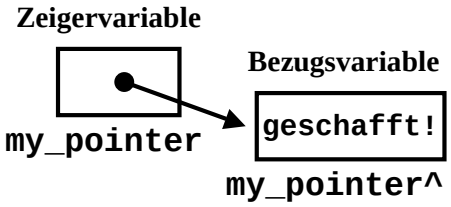
Der Wert (Inhalt) der Bezugsvariablen kann über den sog. **Dereferenzierungs-Operator** (dereferencing operator) **`^`** angesprochen werden.

Zeiger (Pointer)

Beispiel:

Die Bezugsvariable von **`my_pointer`** wird also mit **`my_pointer^`** angesprochen.

```
MODULE Pointer_example;
…
TYPE string_pointer = Pointer TO string;
VAR my_pointer:string_pointer
BEGIN
   NEW(my_pointer);
   my_pointer^:='geschafft!';
END Pointer_example.
```



Zeiger (Pointer)

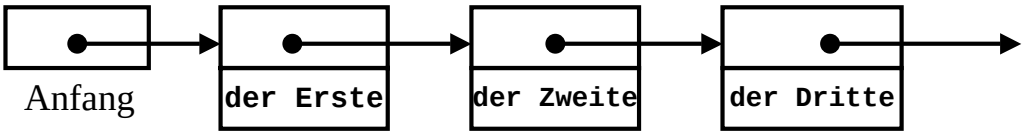
Frage: Warum sind Zeiger so wichtig ?

- Man kann mit Zeigern auf Variablen zeigen, die wiederum auch Zeiger enthalten.

  Das ermöglicht den Aufbau **dynamischer Strukturen**!
- Analog zu den rekursiven Prozeduren, kann man mit Zeigern **rekursive Datenstrukturen** konstruieren.
- Beispiele hierfür sind **Listen** oder **Bäume**.

Lineare Liste

Beispiel: Listenerzeugung mit Zeigern



Deklaration einer linearen Liste

```
TYPE ListPointer = POINTER TO ListNode;
TYPE ListNode = RECORD
                    next:ListPointer;
                    Data:string;
                END
 VAR Anfang,p,Listend : ListPointer;
```


Lineare Liste

Frage: Wie beende ich eine Liste oder eine andere dynamische rekursive Datenstruktur?

- Wir benötigen ein Mittel, um einen Pointer auf "nichts" zeigen lassen zu können, damit die rekursive Definition der Datenstruktur terminiert.
  (termination of data structure)
- Für Zeigervariablen existiert ein spezieller Wert:
  `NIL` (**N**ot **I**n **L**ist)
- **`NIL`** zeigt auf **kein** Objekt.
  Damit ist es offensichtlich, dass der Ausdruck **`p^`** **nicht** ausgewertet werden darf, falls **`p=NIL`** ist.

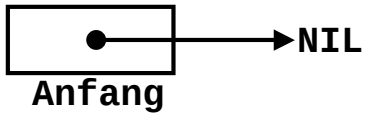
Lineare Liste

Wie muss ein Programm zur Erzeugung einer Liste aussehen?

- Zunächst müssen wir die Liste **initialisieren**.
  Das bedeutet, dass wir die Variable **`Anfang`** auf **`NIL`** setzen müssen, damit sie nicht auf eine undefinierte Speicherposition zeigt:

```
Anfang := NIL;
```

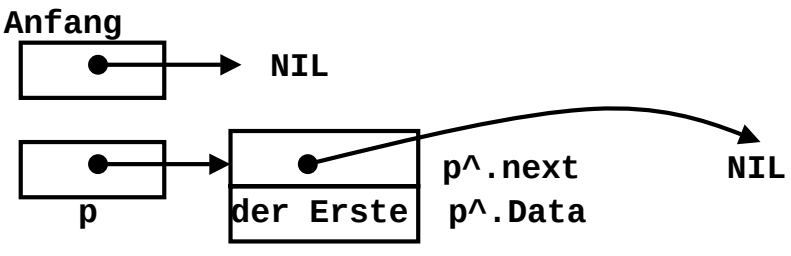
  Damit haben wir die **leere Liste** erzeugt.



Lineare Liste

Jetzt erzeugen wir das erste Element:

- **`new(p);`**
- Da der Zeiger **`p^.next`** auf eine undefinierte Speicherposition zeigen könnte, wird er auf **`NIL`** gesetzt:
  `p^.next := NIL;`
- Wir setzen die gewünschten Daten in unser neues Element:
  `p^.Data := "der Erste";`

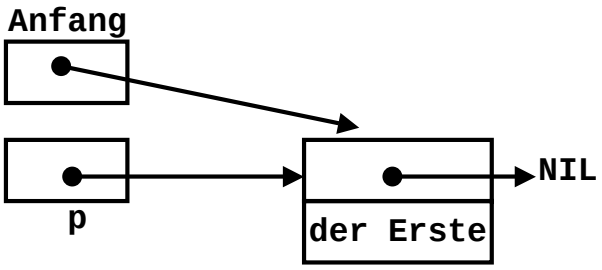


Lineare Liste

Jetzt müssen wir das neue Element in die leere Liste **einfügen**:

- Dazu setzen wir den Zeiger **`Anfang`** so,
  dass er auf das Element zeigt, auf welches auch **`p`** zeigt:

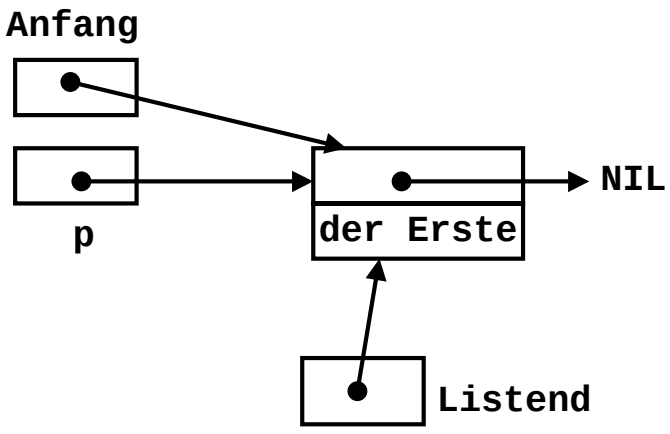
```
Anfang := p;
```



Lineare Liste

Wir "merken" uns das aktuelle **Ende** der Liste mit einem **Hilfszeiger**, den wir z.B. **`Listend`** nennen.

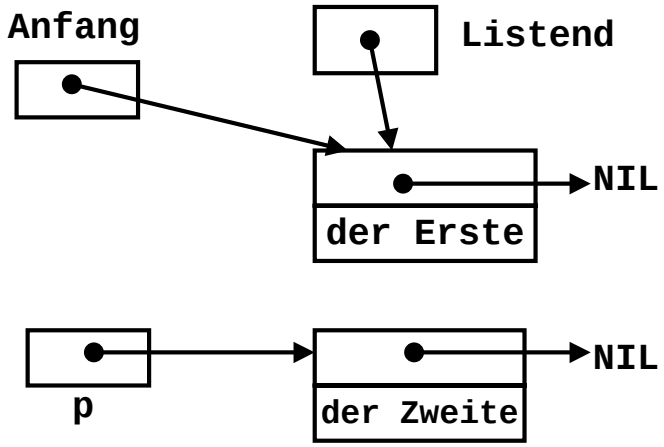
```
Listend := p;
```



Lineare Liste

Wir erzeugen einen weiteren Knoten:

```
new(p);
p^.next := NIL;
p^.DATA := "der Zweite";
```

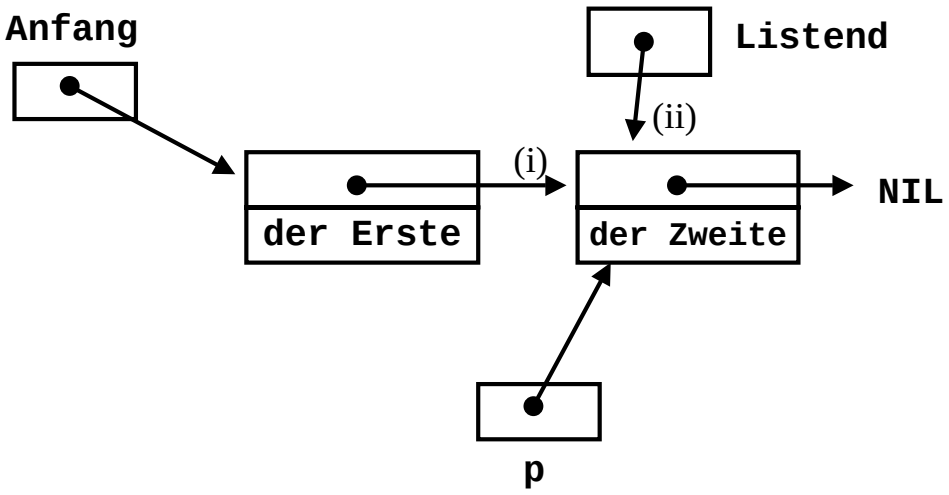


Lineare Liste

Wir fügen auch diesen Knoten am Ende der Liste ein und **aktualisieren** unseren Zeiger auf das Listenende:

(i) `Listend^.next := p;`

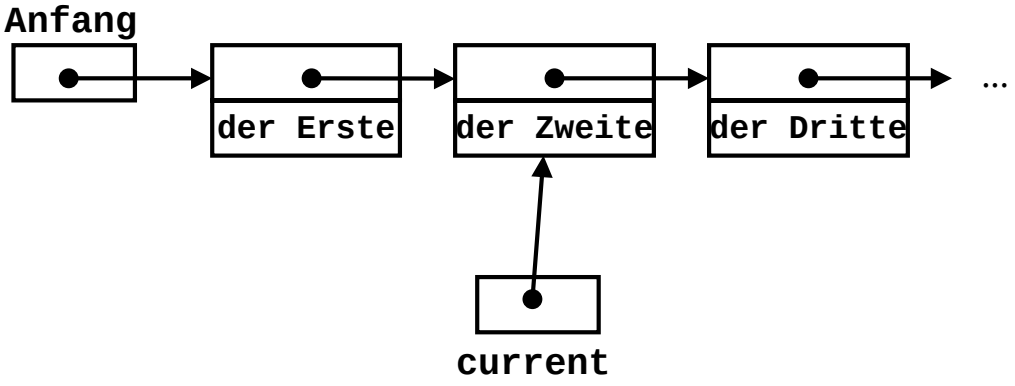
(ii) `Listend := Listend^.next;`



Lineare Liste

Frage: Wie durchsucht man eine lineare Liste?

- Um eine lineare Liste durchsuchen zu können, brauchen wir eine **Laufvariable** (hier **`current`**), die angibt, an welcher Listenposition wir uns aktuell befinden.



Lineare Liste

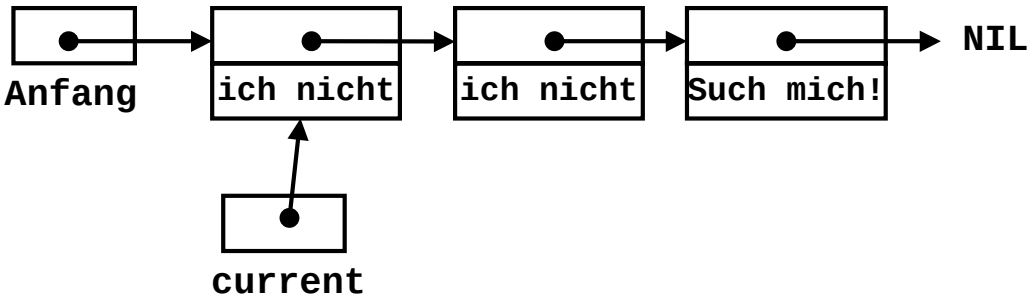
Diese Laufvariable wird als **Zeiger des Listentyps** deklariert,

```
VAR current : ListPointer;
```

und auf den Listenanfang gesetzt (Initialisierung).

```
current := Anfang;
```

Lineare Liste



Danach muss man in einer **Schleife** die Liste **durchlaufen**, bei jedem Schritt die Suchbedingung **überprüfen** und die Laufvariable **`current`** gegebenenfalls ein Listenelement **weiterbewegen**.

```
WHILE (current^.Data<>"Such mich!") AND
       (current^.Data<>NIL) (* Auf Listenende achten *)
  DO  current:=current^.next
END;
```


Lineare Liste

- Man kann also eine Funktion schreiben, die eine Liste durchsucht.
- Weiterhin ist es möglich, die Elemente an jedem Punkt der Liste einfügen oder löschen zu können. Dazu kann man die Such-Funktion verwenden.
- Bei der Implementierung von Einfüge- bzw. Löschoperationen muss man aufpassen, dass man die Liste wieder korrekt zusammenfügt.

Lineare Liste

Beispiel:

Löschen des Listenelements mit dem Inhalt "**`der Zweite`**"

Zunächst werden zwei Hilfszeiger **lauf1** und **lauf2** deklariert.

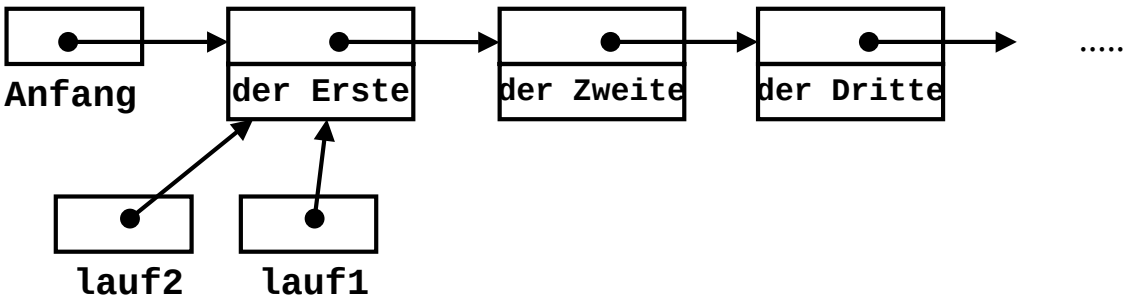
```
VAR lauf1, lauf2 : Listpointer;
```

Ziel ist es, **`lauf2`** auf das zu **löschende Element** und **`lauf1`** auf dessen **Vorgänger** zeigen zu lassen.

Lineare Liste

Beide Zeiger werden auf den Listenanfang gesetzt.

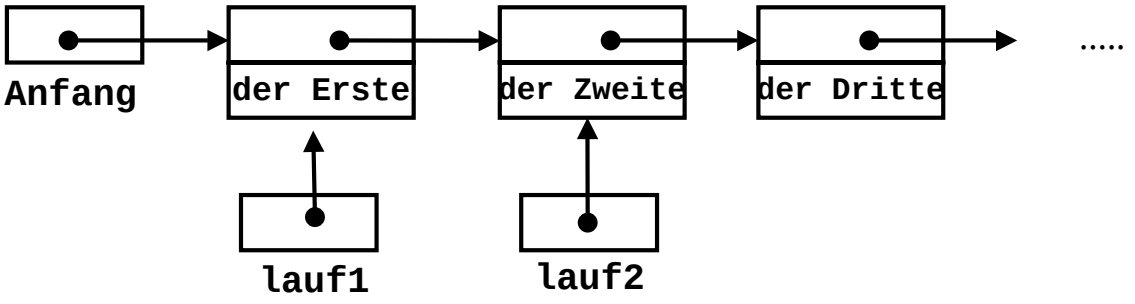
```
lauf1:=Anfang;
lauf2:=Anfang;
```



Lineare Liste

Danach wird das entsprechende Element gesucht:

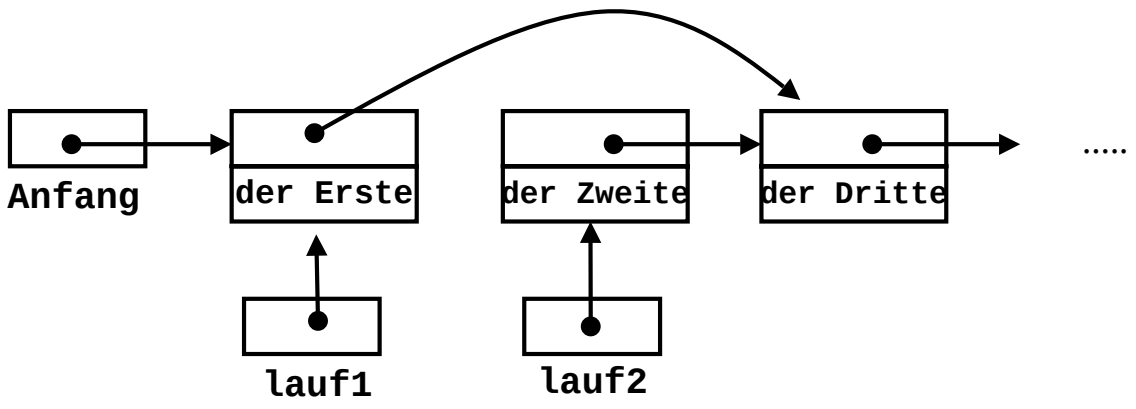
```
REPEAT
    lauf1 := lauf2;
    lauf2 := lauf2^.next;
UNTIL lauf2^.Data = "der Zweite";
```



Lineare Liste

Jetzt wird das Element aus der Kette heraus genommen:

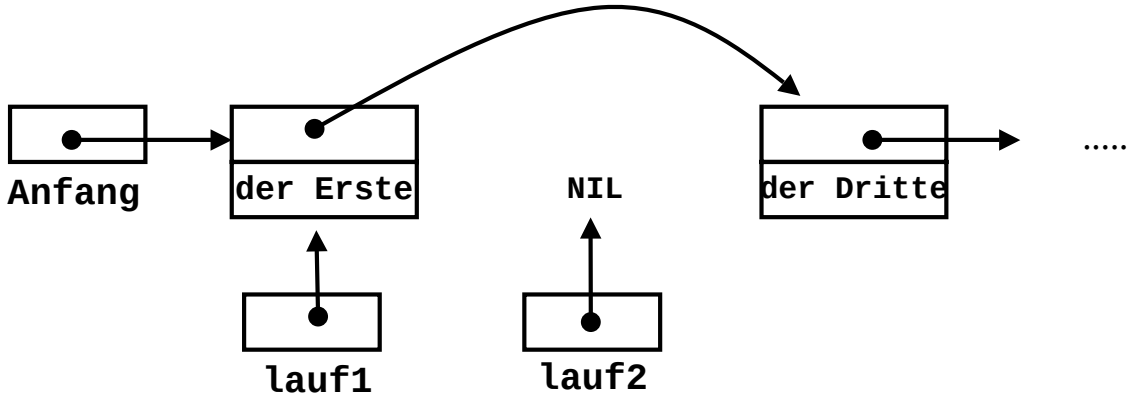
```
lauf1^.next := lauf2^.next;
```



Lineare Liste

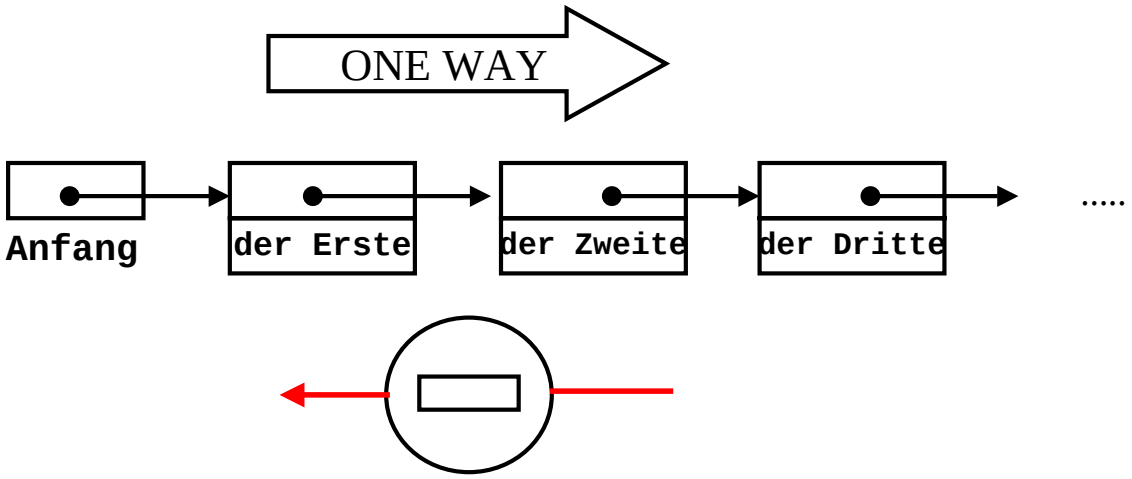
Das Element kann nun auch physikalisch gelöscht werden.
(Sein Speicherplatz wird wieder deallokiert (freigegeben).

```
DISPOSE (lauf2);
```



Lineare Liste

Problem: Lineare Listen können immer nur
in einer Richtung durchlaufen werden.



Stapel (stack)

Eine oft genutzte Datenstruktur, die mit einer linearen Liste realisiert werden kann, ist der **Stapel (stack)**, der manchmal auch als **Keller** bezeichnet wird.

Beim Stapel sind nur die folgenden beiden Operationen erlaubt:

- **`pop`** = hole das **oberste Element** vom Stapel
- **`push(x)`** = lege das Element `x` **oben** auf den Stapel

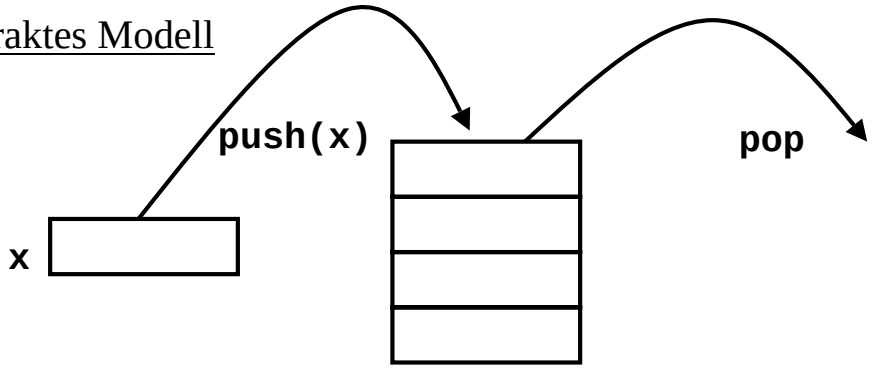
Beim Stapel kann man immer nur Zugriff auf das **oberste** Element.

Um ein Element aus einem Stapel zu **löschen**, muss man zuvor alle Elemente, die "über" dem gesuchten Element liegen, entfernen.

Stapel (stack)

Beispiel eines Stapels:

Abstraktes Modell



Ein Stapel arbeiten nach dem **LIFO**-Prinzip (**Last In First Out**).