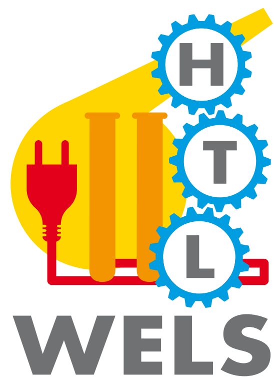
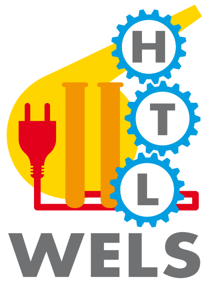
DYNAMISCHE DATENSTRUKTUREN IN C

DI Thomas Helml

SEW 3
SJ 2019/20

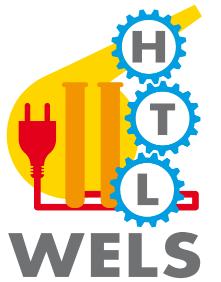


INHALTSVERZEICHNIS



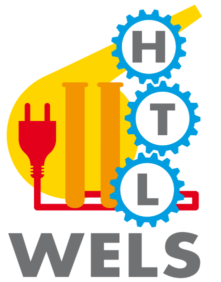
- Dynamischer Speicher
  - Reservieren
  - Freigeben
  - Vergrößern + Verkleinern
- Einfach verkettete Listen
- Doppelt verkettete Listen
- Ringstruktur
- Bäume

MOTIVATION



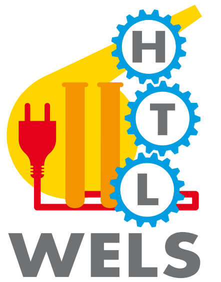
- Nachteil statischer Speicher (Arrays)
  - Man weiß vorher nicht, wie viel Speicher benötigt wird
  - Verschwendung von Speicherplatz
  - Gültigkeit des Arrays: Anweisungsblock
- Lösung:
  - Dynamische Speicherverwaltung (zur Laufzeit)

SPEICHER RESERVIEREN



- Heap (Halde)
  - Speicher, der zur Laufzeit reserviert werden kann
  - man erhält immer zusammenhängenden Bereich
- Reservierung von Speicher mittels (`stdlib.h`):
  - `malloc()`
  - `calloc()`

SPEICHER RESERVIEREN

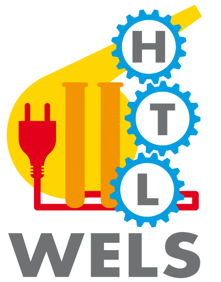


- Syntax:

```
void *malloc(size_t size);
```

- `malloc()` (memory allocation) liefert `size` Bytes zusammenhängenden Speicher
- Return-Wert:
  - `NULL` bei Fehler, sonst
  - Zeiger auf Anfangsadresse des res. Speicherblocks

SPEICHER RESERVIEREN



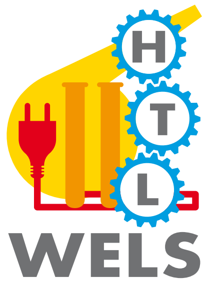
- Beispiel: Platz für 100 int reservieren

```
int *iptr;
iptr = malloc (400); // 400 Byte Speicher reservieren
...
```

- Vorsicht! int-kann unterschiedl. Größe haben
- Besser:

```
iptr = malloc (100*sizeof(int));
iptr = malloc (100*sizeof(*iptr));
```

SPEICHER RESERVIEREN



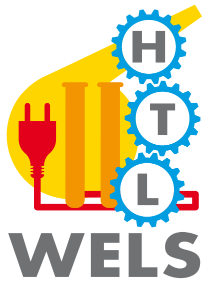
- Syntax `calloc` (core allocation):

```
void *calloc(size_t n, size_t size);
```

- Parameter:
  - `n`: Anzahl an zu reservierenden Objekte
  - `size`: Größe eines Objekts
- `calloc()` initialisiert den Speicher mit 0!
- Beispiel:

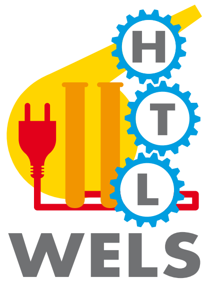
```
iptr = calloc (100, sizeof(int));
```

SPEICHER FREIGEBEN



- ACHTUNG:
  - Jeder Speicherblock der reserviert wurde, muss auch freigegeben werden!
  - Freigabe erfolgt anders wie in Java NICHT automatisch
  - Fehlende Freigabe führt zu Speicherlöchern – sogenannten Memory Leaks

SPEICHERBLOCK FREIGEBEN

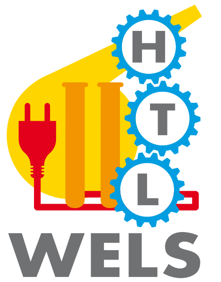


- Syntax:

```
#include <stdlib.h>
void free(void *ptr);
```

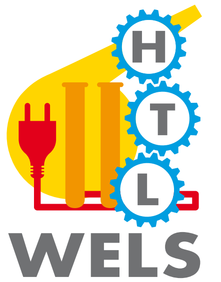
- der Speicherbereich, der mit `malloc/calloc` allokiert wurde, wird freigegbn
- ist `ptr` ein `NULL`-Pointer, passiert nichts
- ist `ptr` kein/ein falscher Zeiger => undefiniertes Verhalten
- `ptr` sollte nach Freigabe wieder auf `NULL` gesetzt werden

SPEICHER FREIGEBEN / BEISPIEL



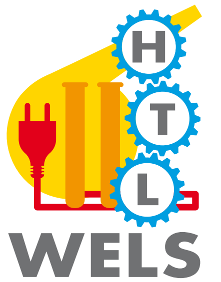

```
#include <stdio.h>
#include <stdlib.h>

// reserviert einen Speicherbereich für n-INT Werte
// und gibt einen Zeiger darauf zurück
int *createArray(unsigned int n) {
    int *iptr = NULL;
    int i = 0;

    iptr = malloc (n*(sizeof(int)));
    if (iptr!=NULL)
        for (i=0; i<n; i++)
            iptr[i] = i*i; // *(iptr+i) = ...;
    return iptr;
}
```

SPEICHER FREIGEBEN / BEISPIEL




```
int main () {
    int *arr = NULL;
    unsigned int val=0, i=0;

    printf ("Wie viele int-Elemente benötigen Sie? "); fflush(stdout);
    scanf("%u", &val);

    arr = meinArray(val);
    if (arr==NULL) {
        printf ("nicht genügend speicher");
        return -1;
    }
    printf ("Ausgabe der Elemente\n");
    for (i=0;i<val;i++) printf ("arr[%d] = %d\n", i, arr[i]);
    free(arr);
    arr=NULL;
    return 0;
}
```

SPEICHERBLOCK VERGRÖẞERN/VERKLEINERN

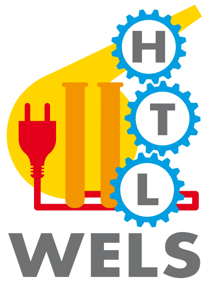


- Größenänderung mit `realloc()` möglich
- Syntax

```
void *realloc (void *ptr, size_t size);
```

- der durch `ptr` adressierte Speicherbereich
  - wird freigeben
  - der ursprüngliche Block bleibt erhalten
  - falls möglich wird der neue Block hinten angehängt
  - sonst wird der gesamte Block umkopiert
- Rückgabewert:
  - Im Fehlerfall: NULL
  - Sonst wird ein Zeiger auf den Speicherblock mit `size` Byte Größe

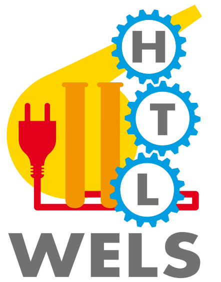
SPEICHERBLOCK VERGRÖßERN/VERKLEINERN



- Wird für `ptr` ein `NULL` Zeiger verwendet, so funktioniert `realloc()` wie `malloc()`
- Folgende Aufrufe sind somit ident:

```
ptr = malloc(100*sizeof(int));
ptr = realloc(NULL, 100*sizeof(int));
```

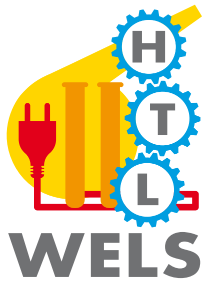
SPEICHERBLOCK VERGRÖßERN/VERKLEINERN



- Verkleinern des Speichers:
  - für `size` kleinere Größe als ursprünglich angenommen:

```
// Speicher für 100 int-Elemente reservieren
ptr = malloc(100*sizeof(int));
...
// Speicher auf 50 int-Elemente verkleinern
ptr = realloc(ptr, 50*sizeof(int));
```

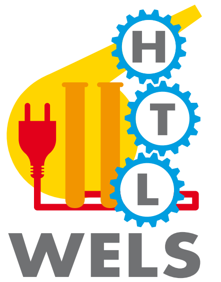
SPEICHERBLOCK VERGRÖẞERN/VERKLEINERN



- Vergrößern des Speichers:
  - für `size` muss Größe angegeben werden, die das alte `size` beinhaltet
  - Folgendes Beispiel ist falsch!

```
int block = 256;
ptr = malloc(block * sizeof(int));
...
// Hier wird kein neuer Speicher reserviert
// es wird nur Speicher für 256 int-Element reserviert
ptr = realloc(ptr, block*sizeof(int));
```

SPEICHERBLOCK VERGRÖßERN/VERKLEINERN



- Vergrößern des Speichers:
  - Korrektes Beispiel:

```
int block = 256;
ptr = malloc(block * sizeof(int));
...
block += block;
// Speicher für 512 int-Elemente reservieren
ptr = realloc(ptr, block*sizeof(int));
```

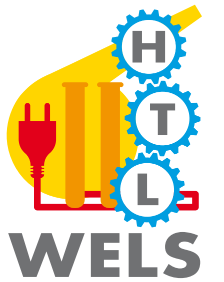
EINFACH VERKETTETE LISTEN

- Beispiel: Struktur für „Namensliste“ (WH)

```
#define MAX_LEN 255


typedef struct data {
    char name[MAX_LEN];
    char vorname[MAX_LEN];
}DATA;
```

- damit lässt sich 1 Datensatz speichern
- Wie kann ich mehrere speichern?

EINFACH VERKETTETE LISTEN



- Möglichkeit 1:
- `DATA dataArr[MAX];`

- Nachteil:
  - Limitierung!
  - Fixe Größe

EINFACH VERKETTETE LISTEN

- Möglichkeit 2:

```
DATA *d = NULL;
d = malloc (sizeof(DATA));
```

  - Speicherplatz wird dynamisch reserviert
  - ABER nur für 1 Datensatz
  - Zeiger muss man sich merken!

```
DATA *d[MAX];
d[i] = malloc (sizeof(DATA));
```

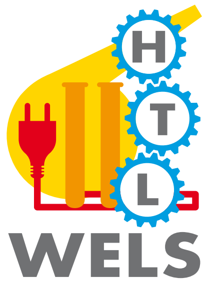
  - Nachteil: kompliziert + limitiert!

EINFACH VERKETTETE LISTEN

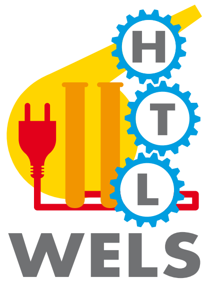
- Annahme:
  - Reihe von Strukturvariablen dynamisch erzeugen
  - Wie können wir uns alle Zeiger merken?
- Lösung: Verketten
- In der Struktur Zeiger auf nächste Struktur

```
typedef struct data {
   char name[MAX];
   char vorname[MAX];
   struct data *next;
}DATA;
```

EINFACH VERKETTETE LISTEN

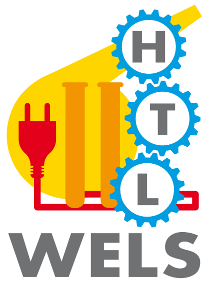


EINFACH VERKETTETE LISTEN



- Anlegen eines neuen Elements

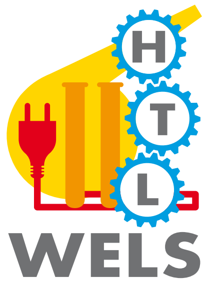
EINFACH VERKETTETE LISTEN



- Anlegen eines neuen Elements

```
DATA *first = NULL;
```

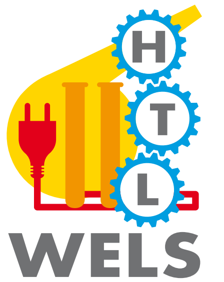
EINFACH VERKETTETE LISTEN



- Anlegen eines neuen Elements

```
DATA *first = NULL;

first = malloc (sizeof(DATA));
```

EINFACH VERKETTETE LISTEN



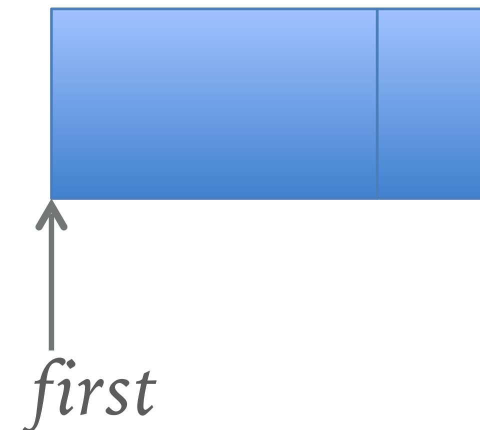
- Anlegen eines neuen Elements

```
DATA *first = NULL;

first = malloc (sizeof(DATA));
```

EINFACH VERKETTETE LISTEN

➤ Anlegen eines neuen Elements

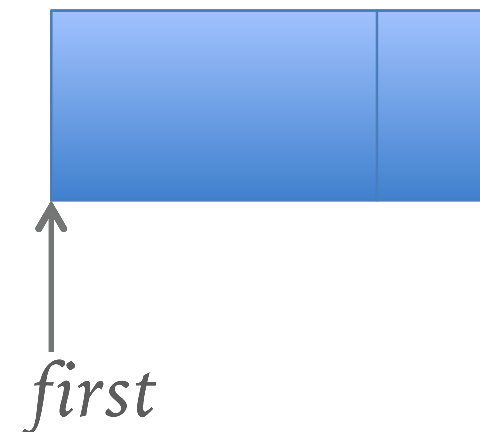
```
DATA *first = NULL;

first = malloc (sizeof(DATA));
```



EINFACH VERKETTETE LISTEN

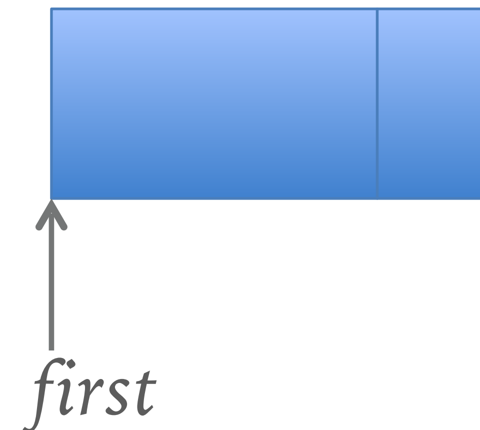
➤ Anlegen eines neuen Elements

```
DATA *first = NULL;

first = malloc (sizeof(DATA));

if (first!=NULL)
```



- Anlegen eines neuen Elements

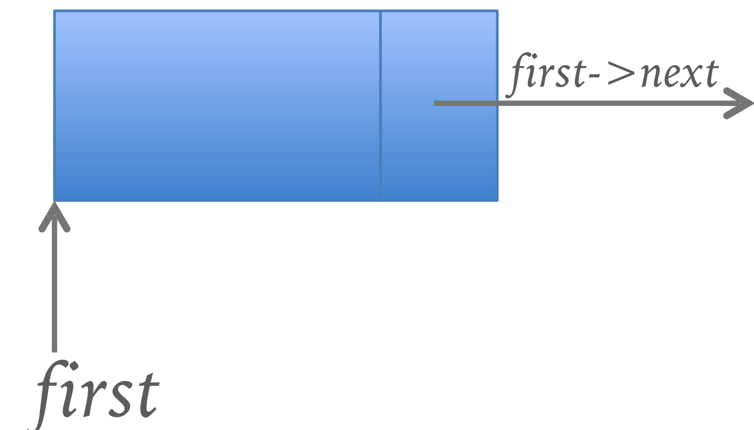
```
DATA *first = NULL;
first = malloc (sizeof(DATA));
if (first!=NULL)
   first->next = NULL;
```



EINFACH VERKETTETE LISTEN

- Anlegen eines neuen Elements

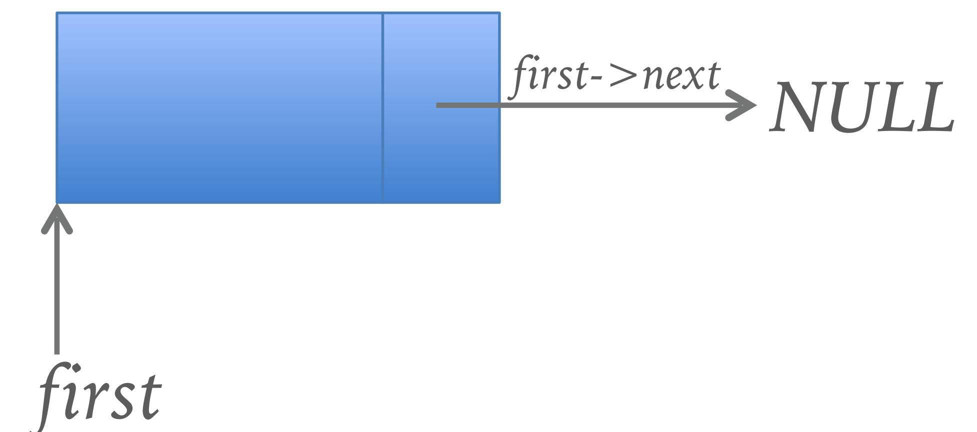
```
DATA *first = NULL;
first = malloc (sizeof(DATA));
if (first!=NULL)
   first->next = NULL;
```



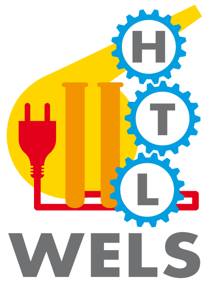
EINFACH VERKETTETE LISTEN

- Anlegen eines neuen Elements

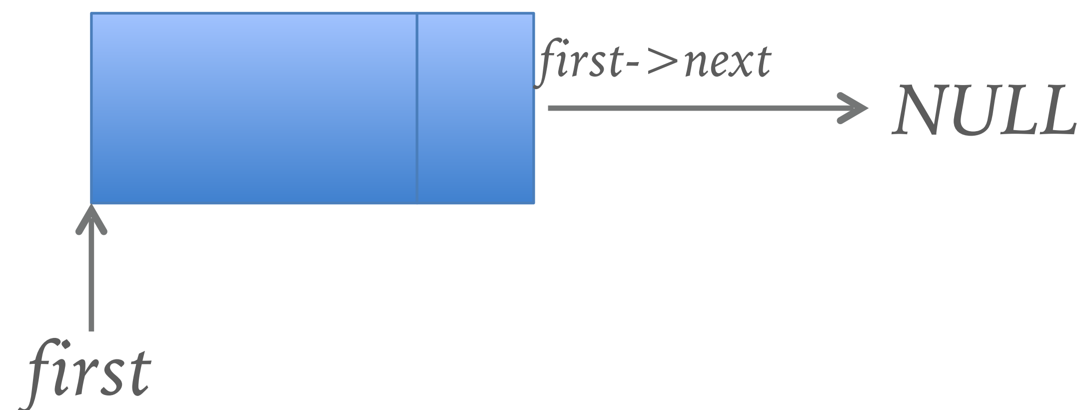
```
DATA *first = NULL;
first = malloc (sizeof(DATA));
if (first!=NULL)
   first->next = NULL;
```



VERKETTETE LISTEN

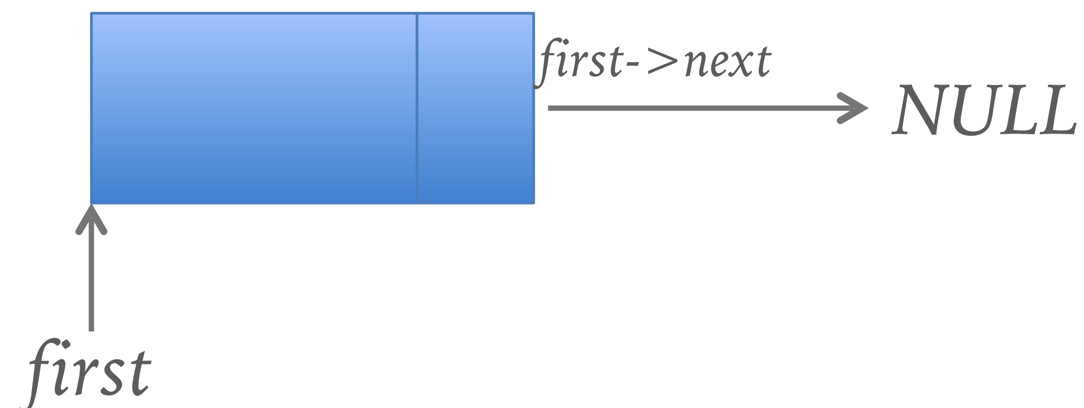


- Anlegen 2. Elements + Anhängen an 1. Element



- Anlegen 2. Elements + Anhängen an 1. Element

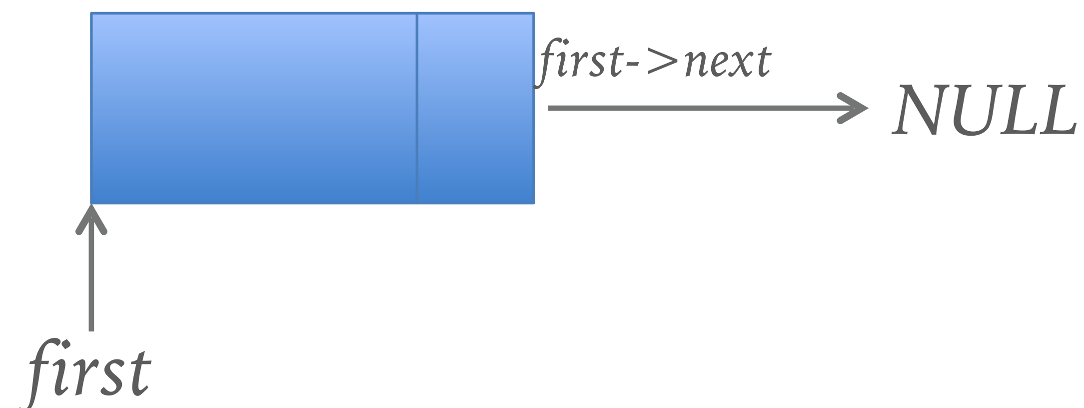
```
DATA *help = NULL;
```



VERKETTETE LISTEN

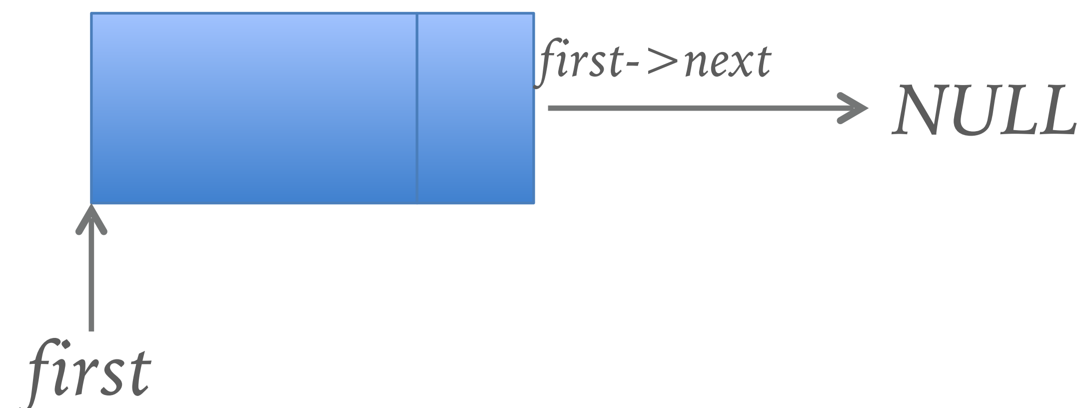
- Anlegen 2. Elements + Anhängen an 1. Element

```
DATA *help = NULL;
```



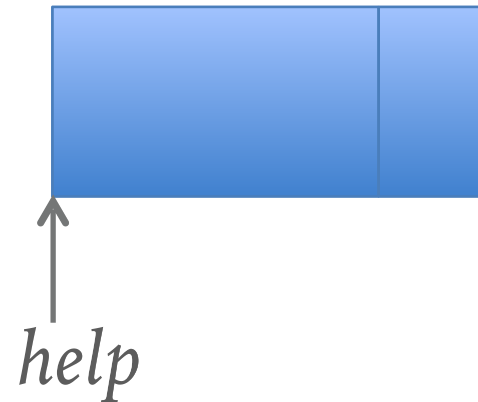
- Anlegen 2. Elements + Anhängen an 1. Element

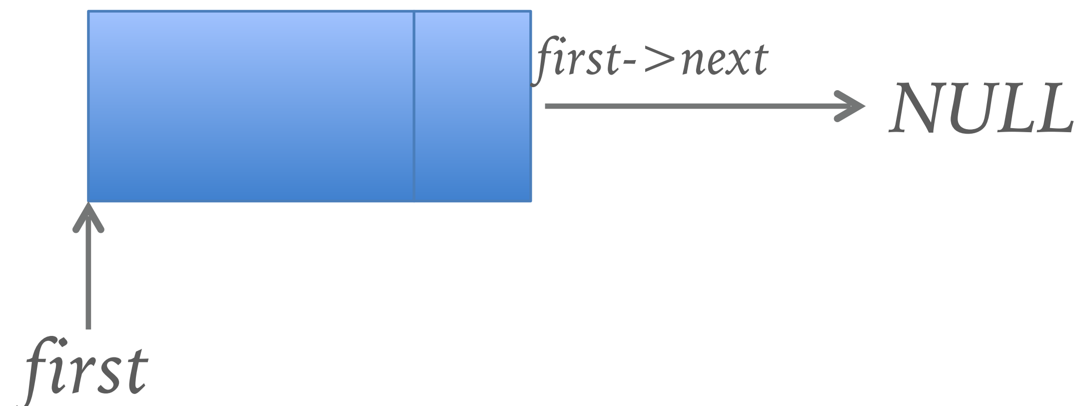
```
DATA *help = NULL;


help = malloc (sizeof(DATA));
```



- Anlegen 2. Elements + Anhängen an 1. Element

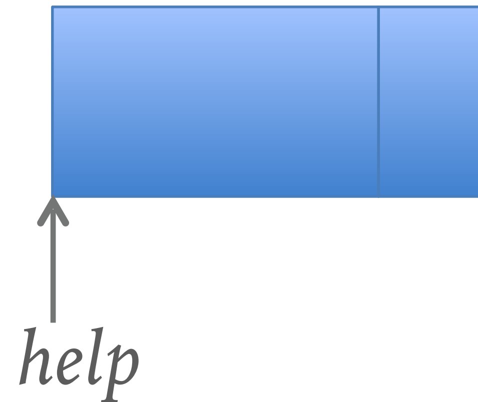
```
DATA *help = NULL;

help = malloc (sizeof(DATA));
```

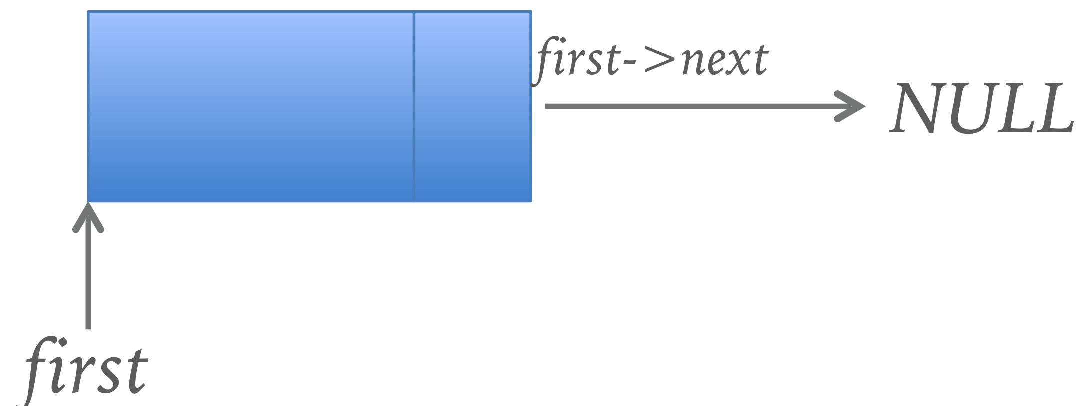




- Anlegen 2. Elements + Anhängen an 1. Element

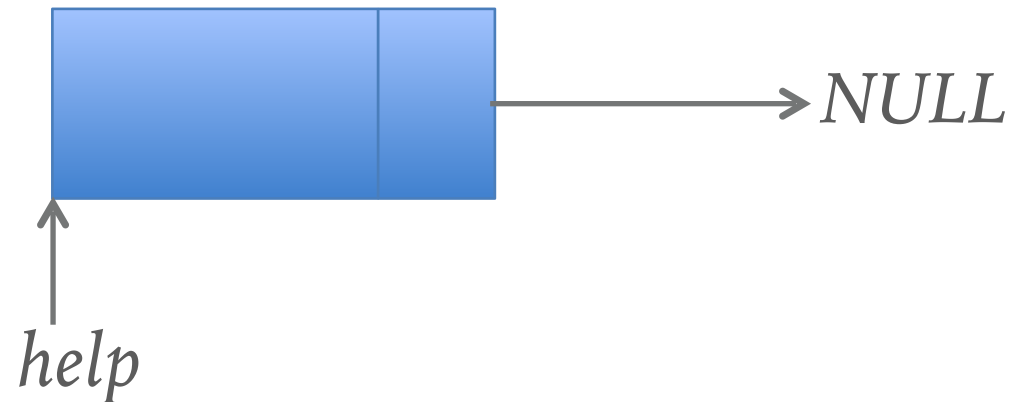
```
DATA *help = NULL;


help = malloc (sizeof(DATA));
if (help!=NULL) {
```

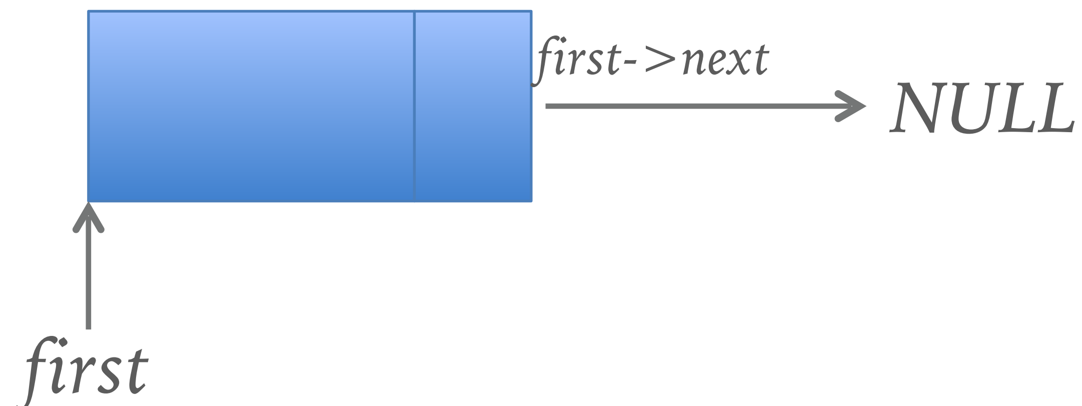




- Anlegen 2. Elements + Anhängen an 1. Element

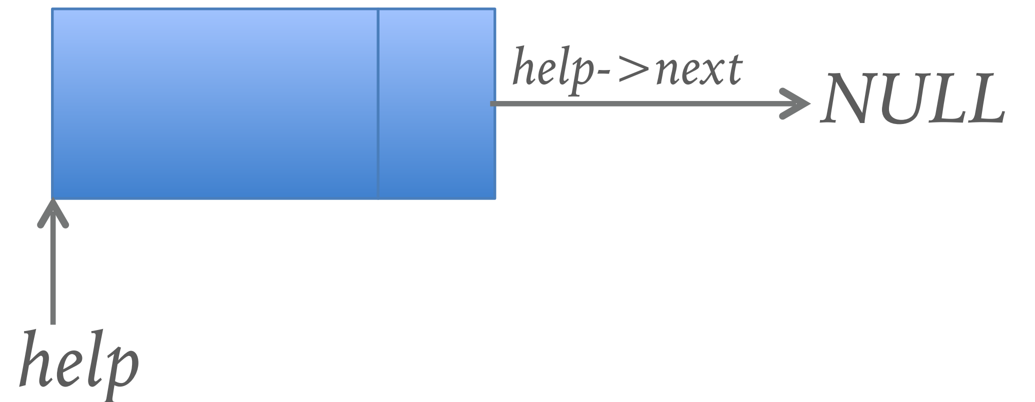
```
DATA *help = NULL;


help = malloc (sizeof(DATA));
if (help!=NULL) {
   help->next = NULL;
```

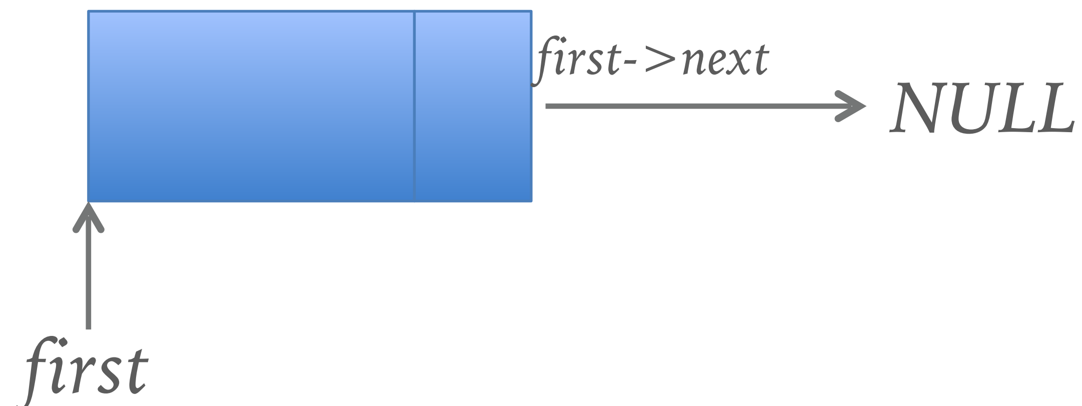




- Anlegen 2. Elements + Anhängen an 1. Element

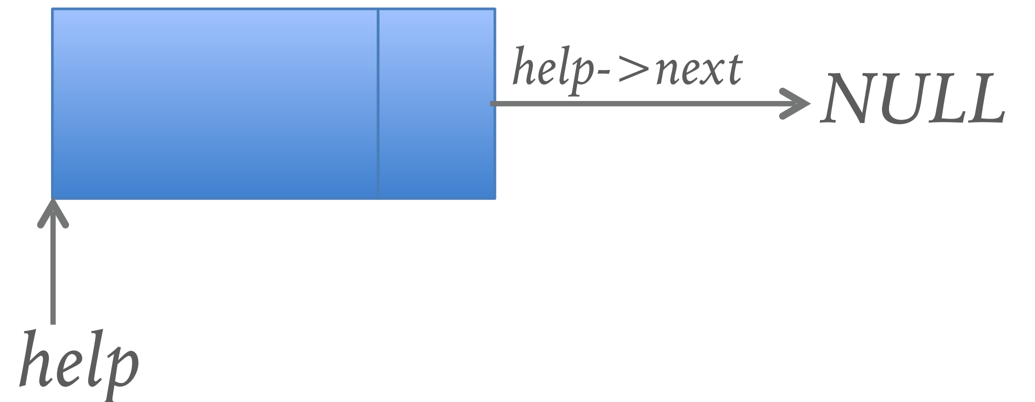
```
DATA *help = NULL;


help = malloc (sizeof(DATA));
if (help!=NULL) {
   help->next = NULL;
```

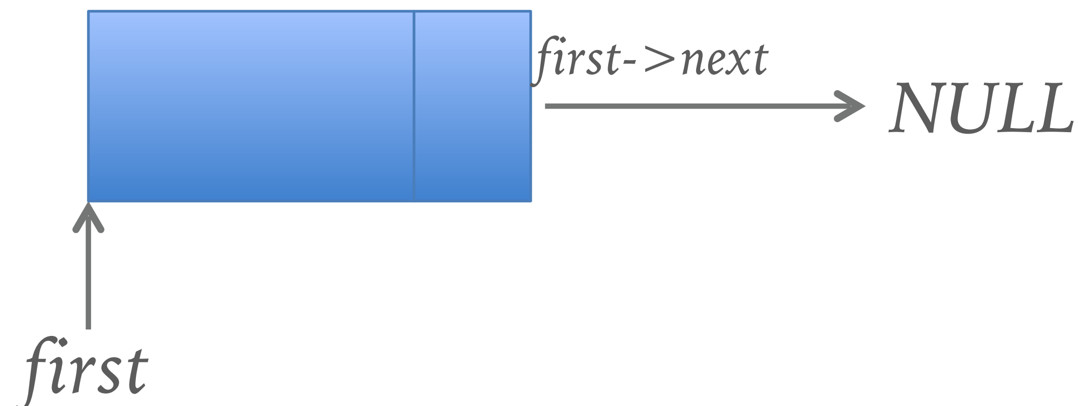




- Anlegen 2. Elements + Anhängen an 1. Element

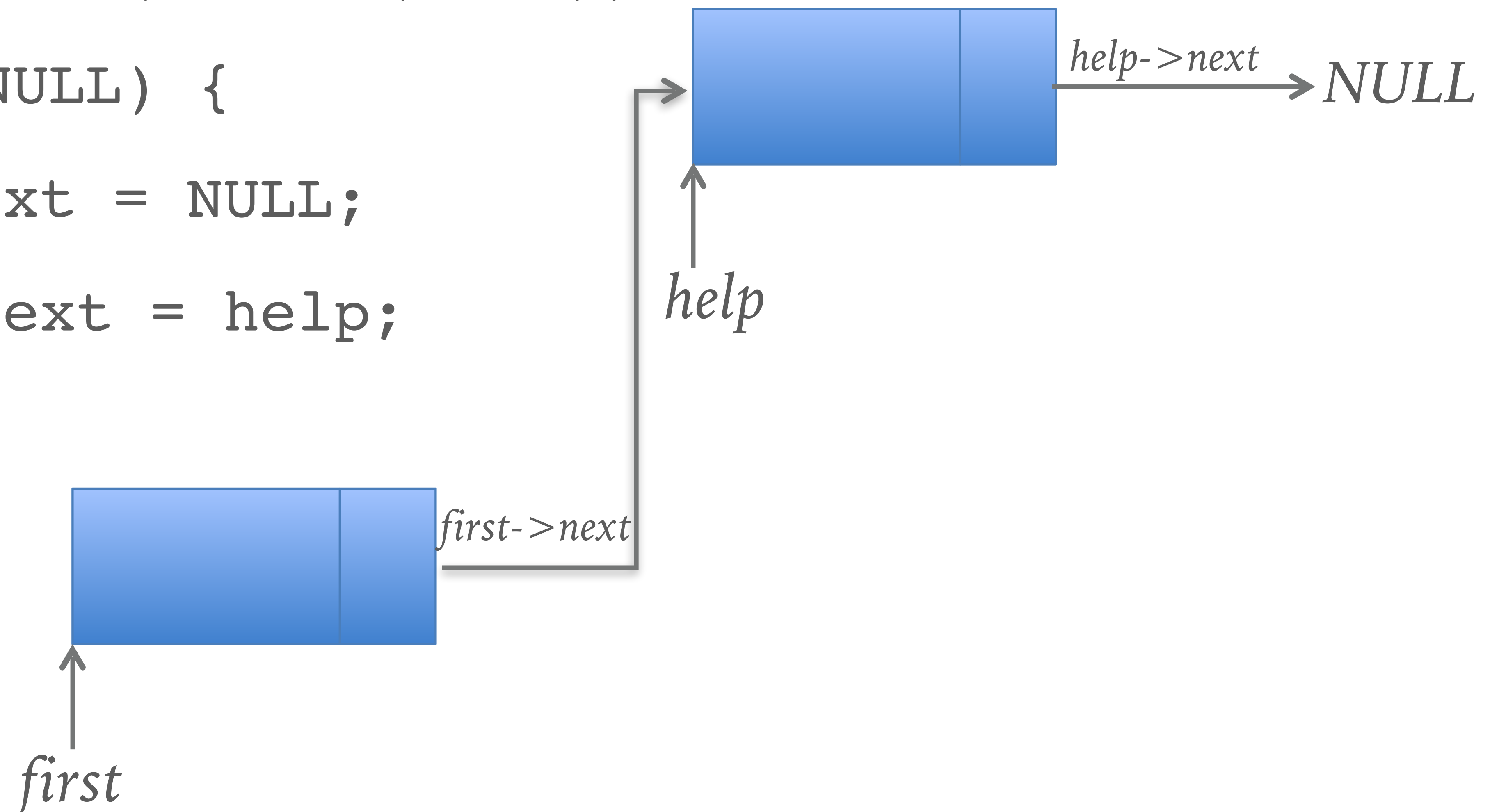
```
DATA *help = NULL;


help = malloc (sizeof(DATA));
if (help!=NULL) {
    help->next = NULL;
    first->next = help;
```





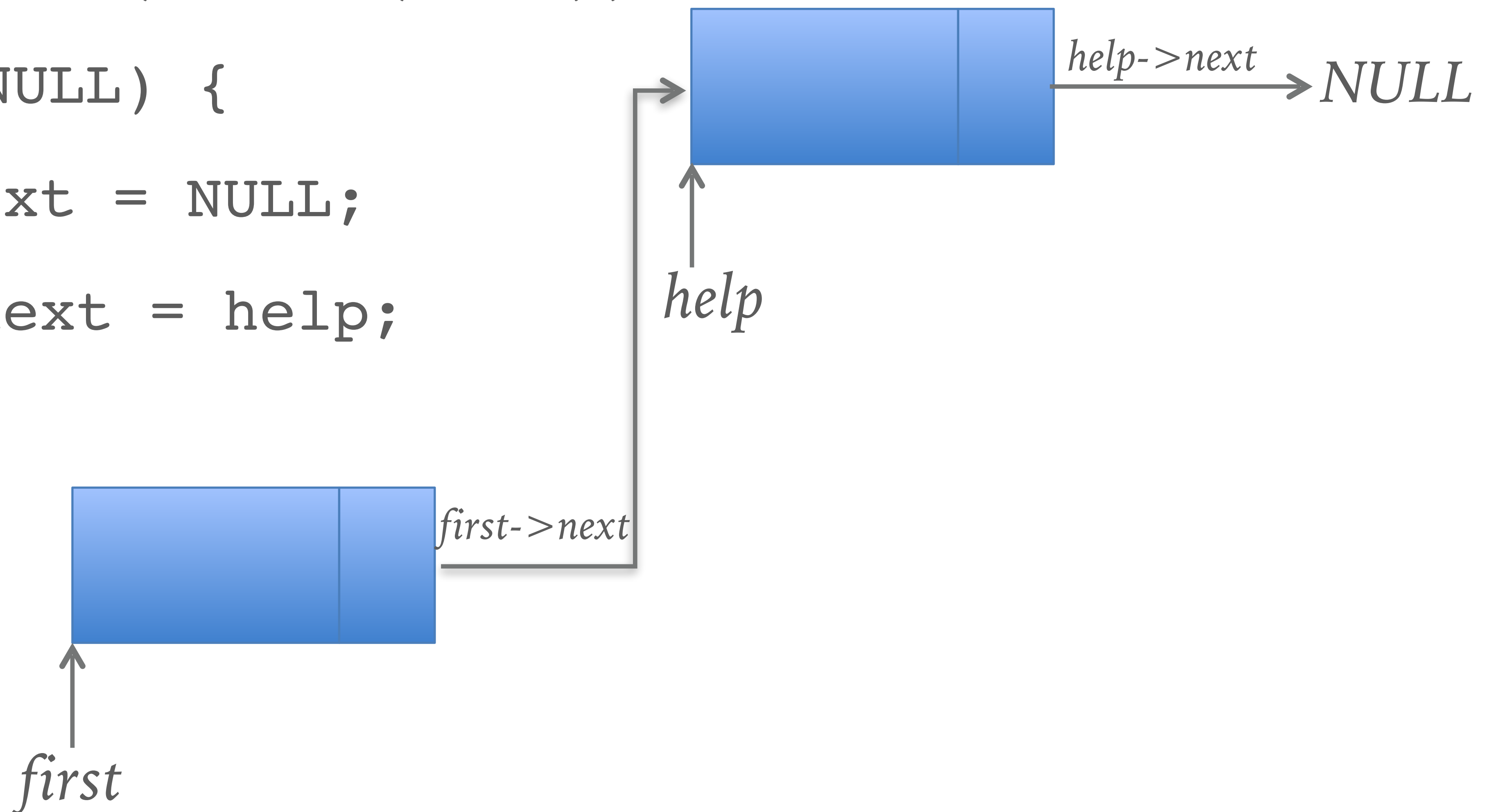
- Anlegen 2. Elements + Anhängen an 1. Element

```
DATA *help = NULL;


help = malloc (sizeof(DATA));
if (help!=NULL) {
   help->next = NULL;
   first->next = help;
```

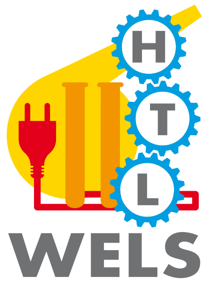


- Anlegen 2. Elements + Anhängen an 1. Element

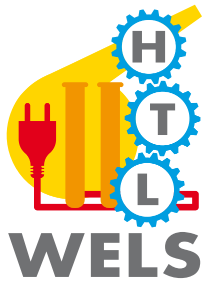
```
DATA *help = NULL;


help = malloc (sizeof(DATA));
if (help!=NULL) {
   help->next = NULL;
   first->next = help;
}
```



VERKETTETE LISTEN

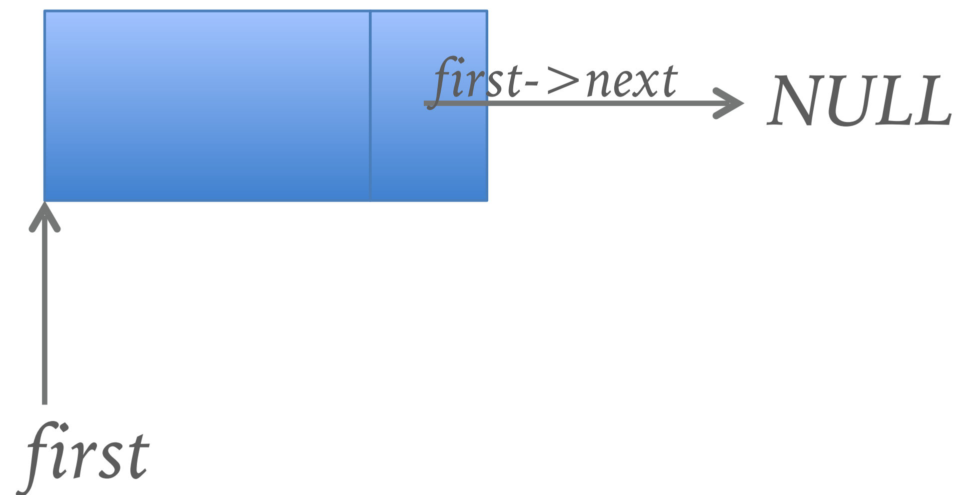


VERKETTETE LISTEN



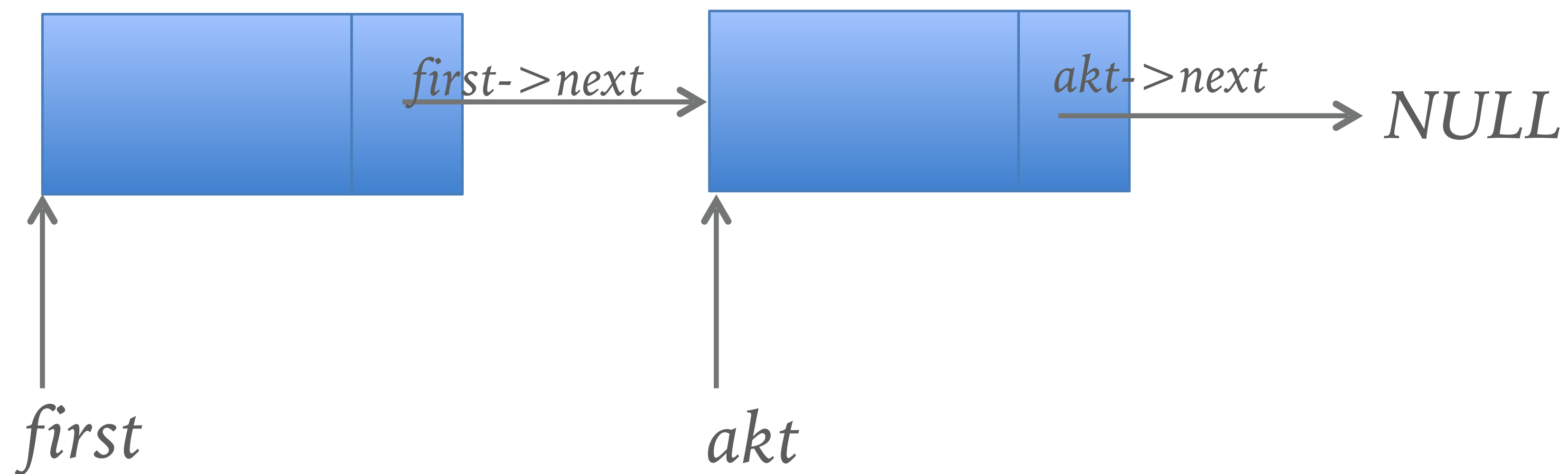
- Für jedes weitere Element:
  - Zeiger auf aktuelles (und letztes) Element merken

- Für jedes weitere Element:
  - Zeiger auf aktuelles (und letztes) Element merken



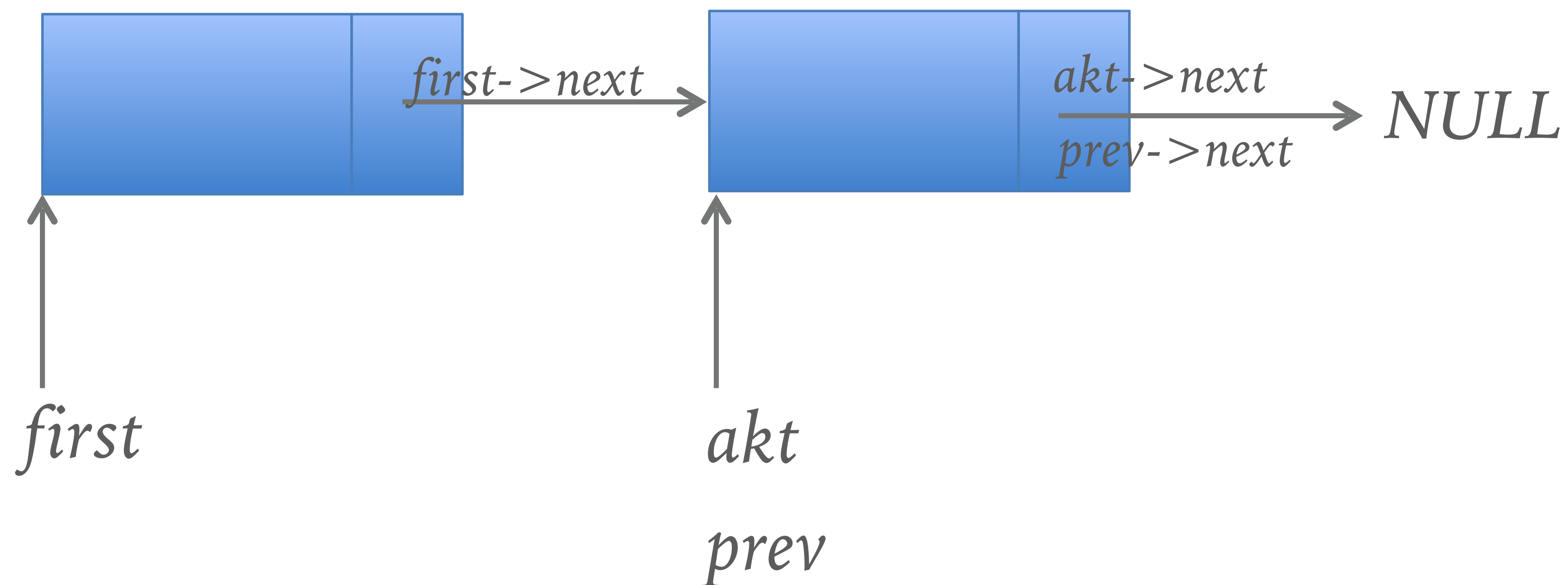
VERKETTETE LISTEN

- Für jedes weitere Element:
  - Zeiger auf aktuelles (und letztes) Element merken



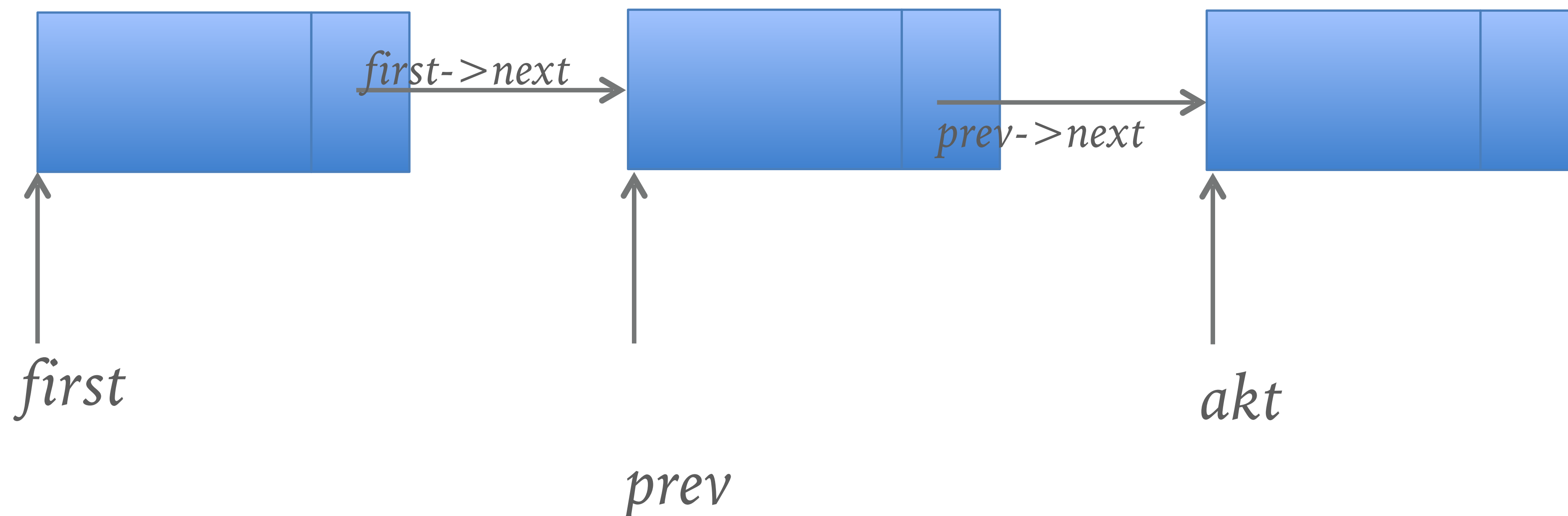
VERKETTETE LISTEN

- Für jedes weitere Element:
  - Zeiger auf aktuelles (und letztes) Element merken



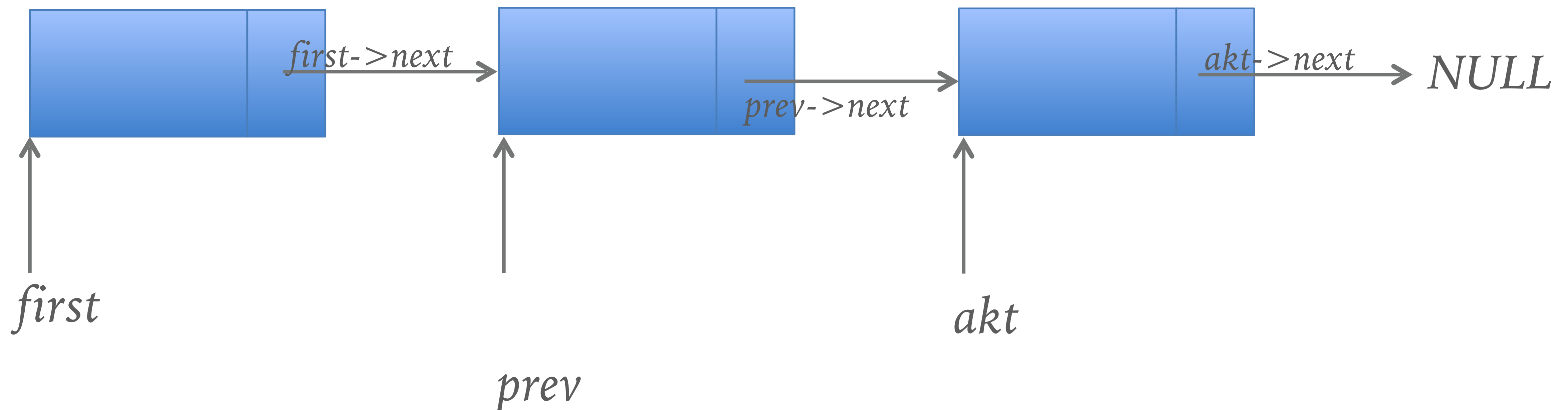
VERKETTETE LISTEN

- Für jedes weitere Element:
  - Zeiger auf aktuelles (und letztes) Element merken

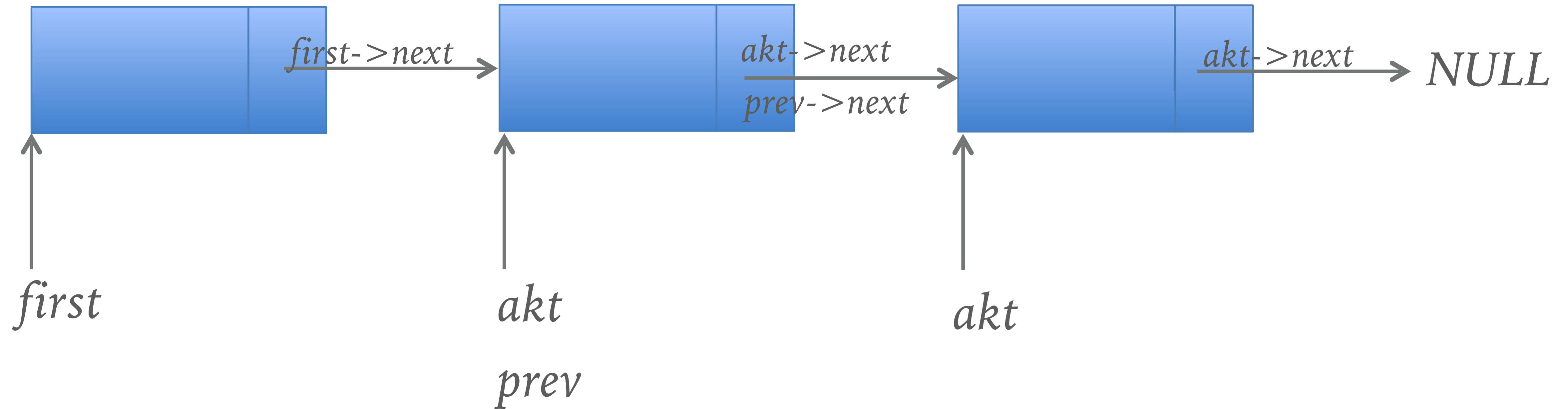


VERKETTETE LISTEN

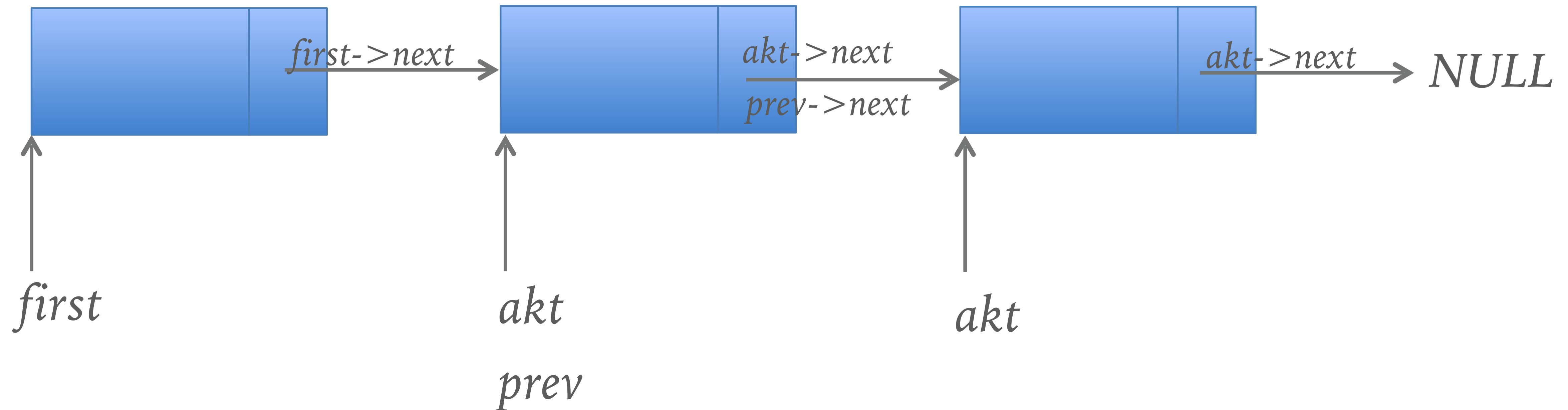
- Für jedes weitere Element:
  - Zeiger auf aktuelles (und letztes) Element merken



EINFACH VERKETTETE LISTEN

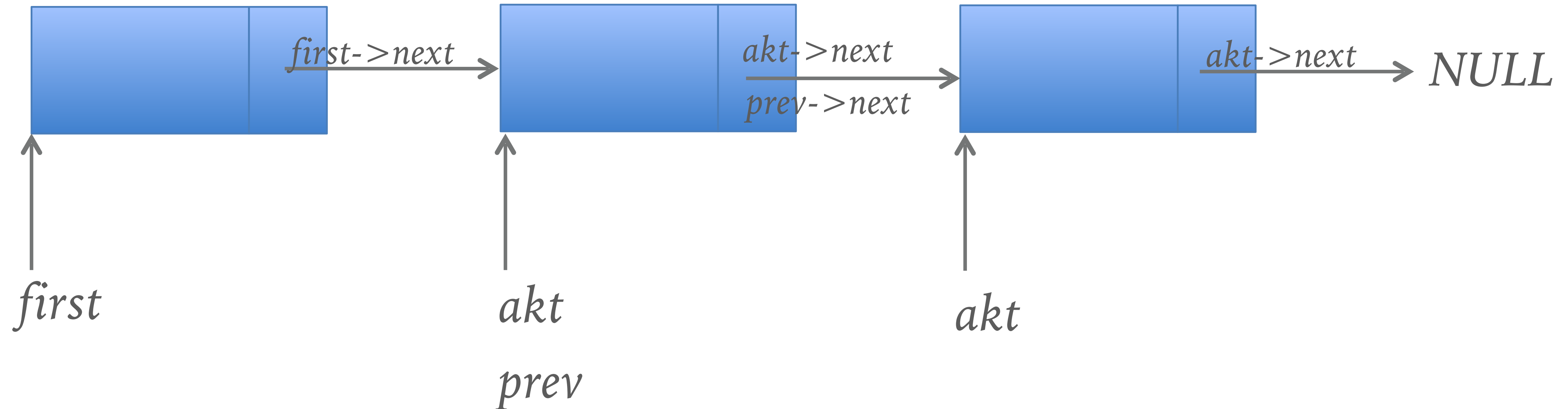


EINFACH VERKETTETE LISTEN



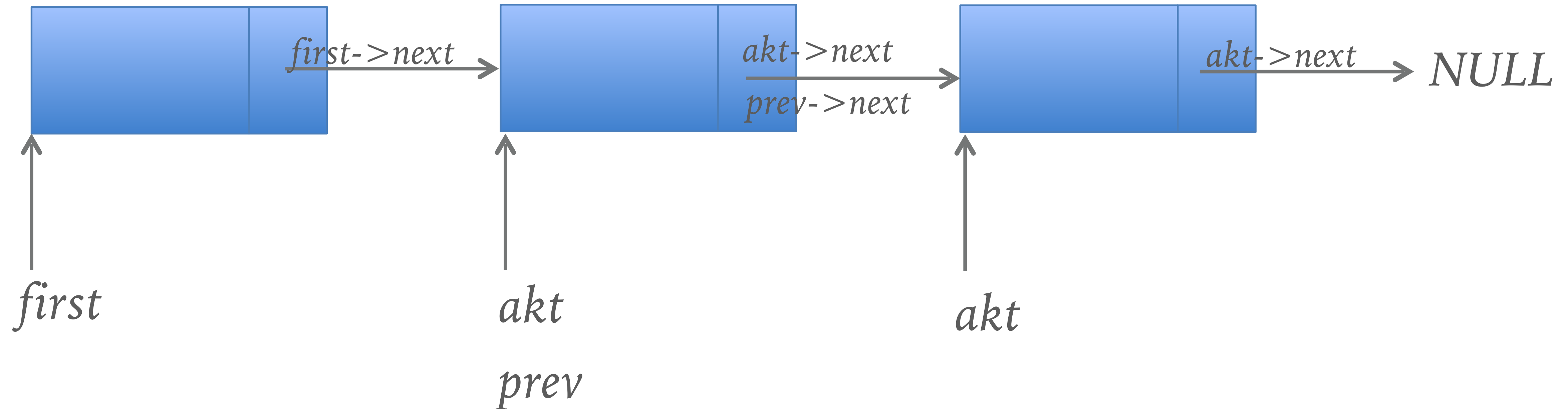
- `first` zeigt immer auf erstes Element
  - Verlust von `first` -> Kette verloren

EINFACH VERKETTETE LISTEN



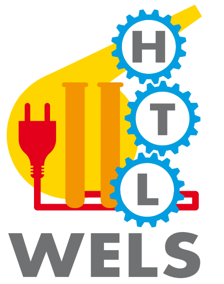
- `first` zeigt immer auf erstes Element
  - Verlust von `first` -> Kette verloren
- `prev` zeigt auf vorletztes Element

EINFACH VERKETTETE LISTEN

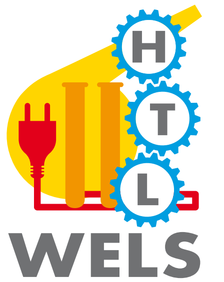


- `first` zeigt immer auf erstes Element
  - Verlust von `first` -> Kette verloren
- `prev` zeigt auf vorletztes Element
- `akt` zeigt auf aktuelles/zuletzt hinzugefügtes Element

EINFACH VERKETTETE LISTEN



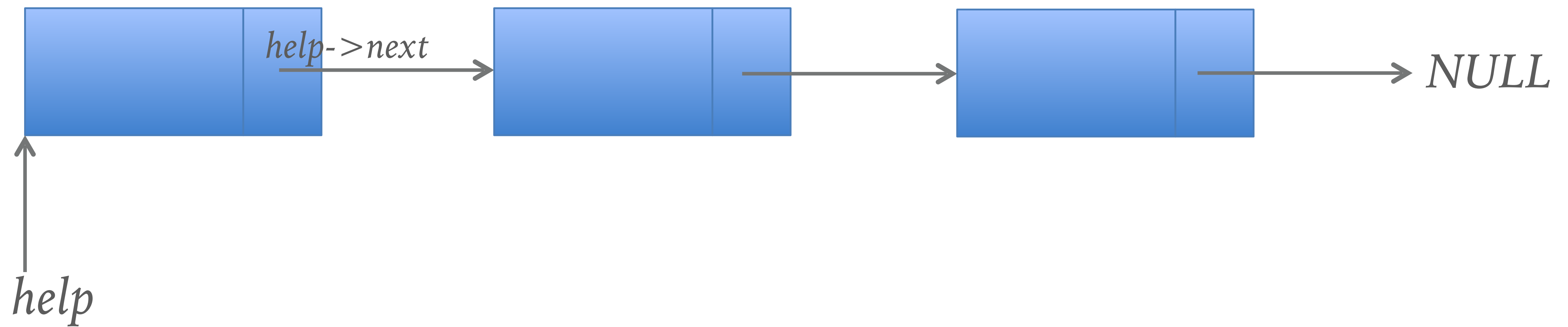
EINFACH VERKETTETE LISTEN



- Hinzufügen eines Elements

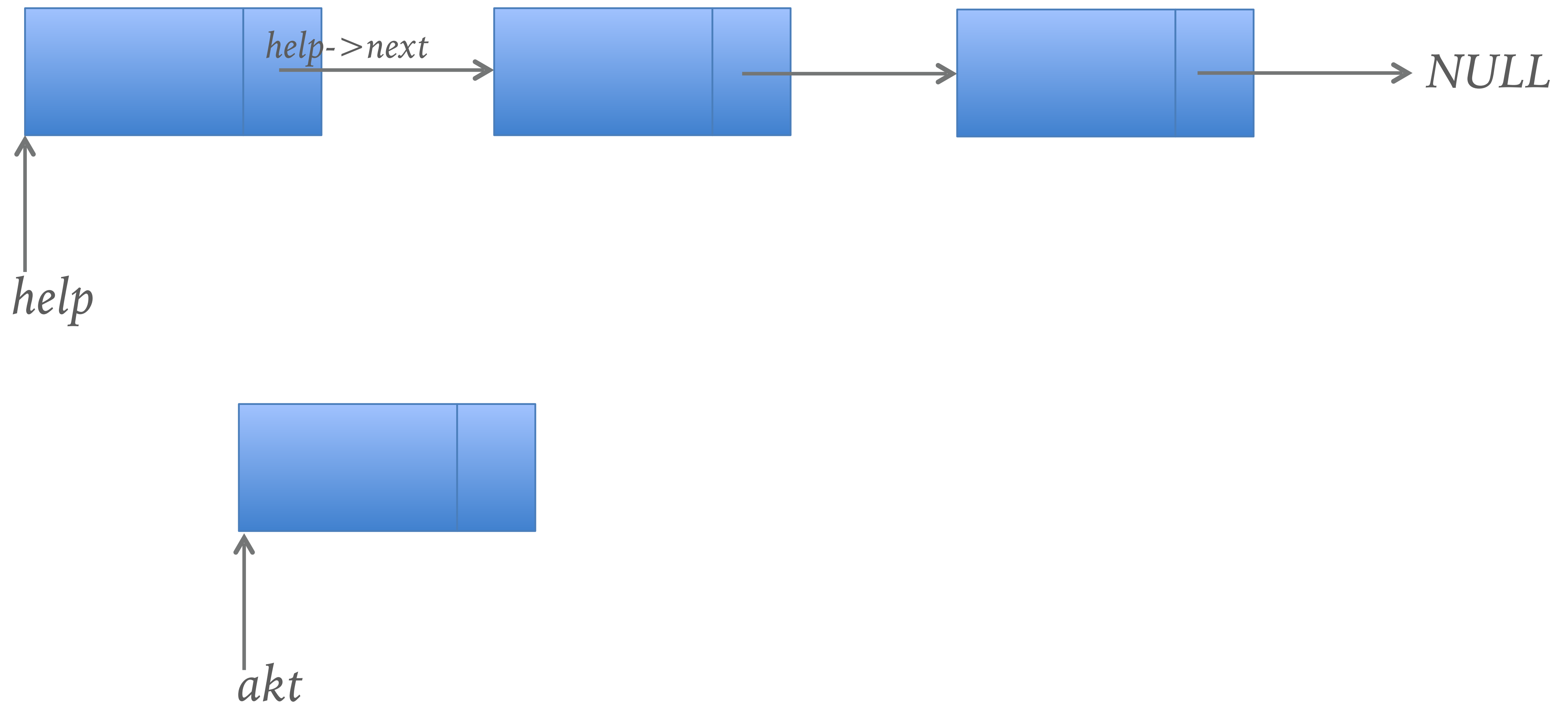
EINFACH VERKETTETE LISTEN

- Hinzufügen eines Elements



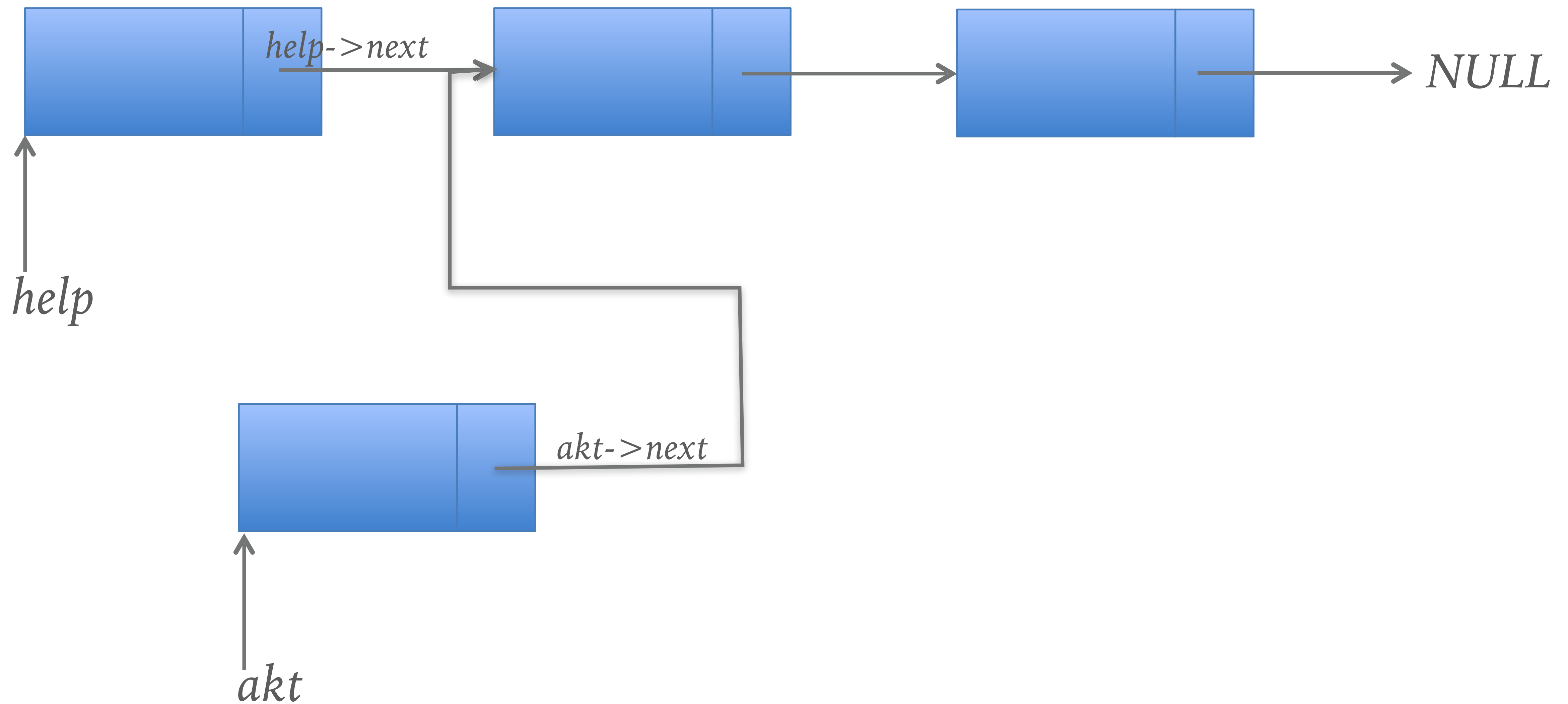
EINFACH VERKETTETE LISTEN

- Hinzufügen eines Elements

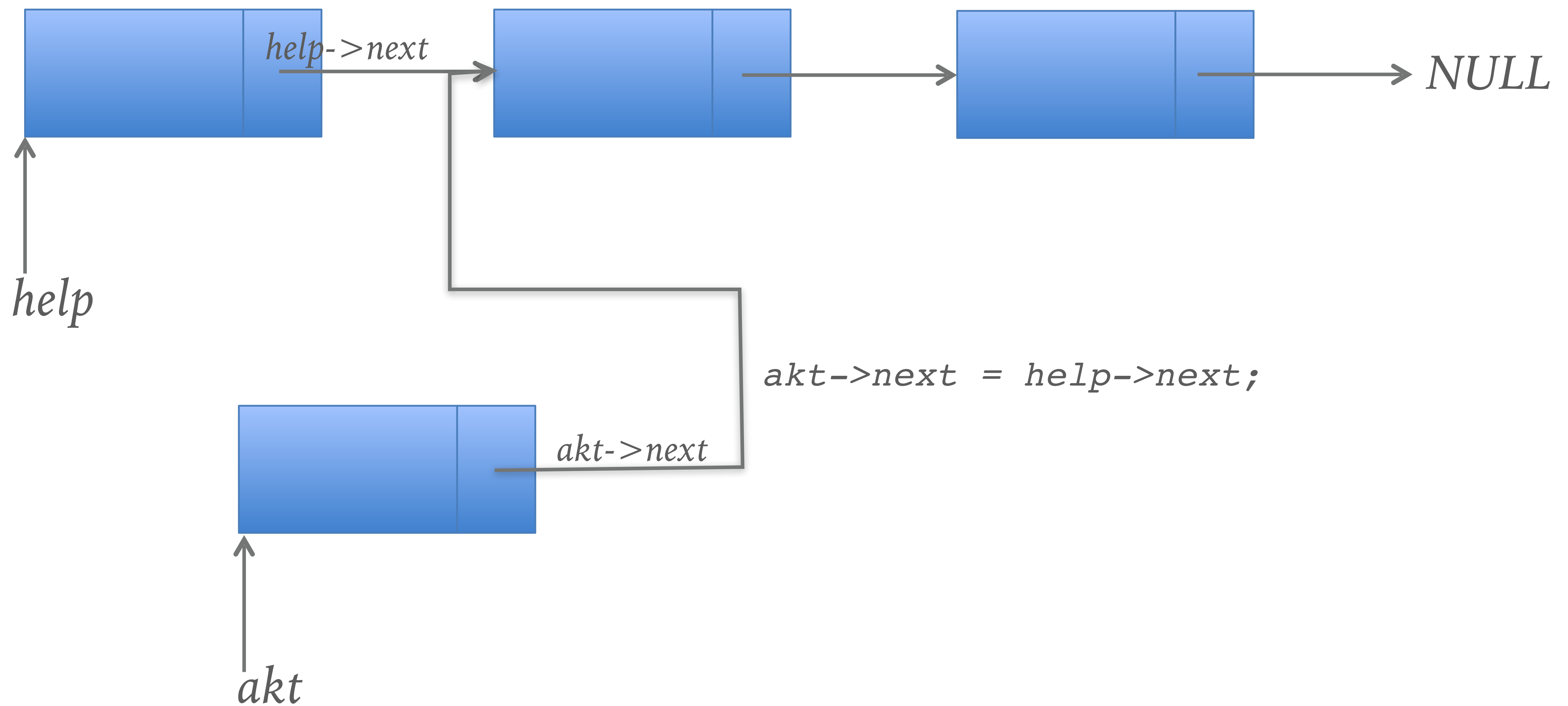


EINFACH VERKETTETE LISTEN

➤ Hinzufügen eines Elements

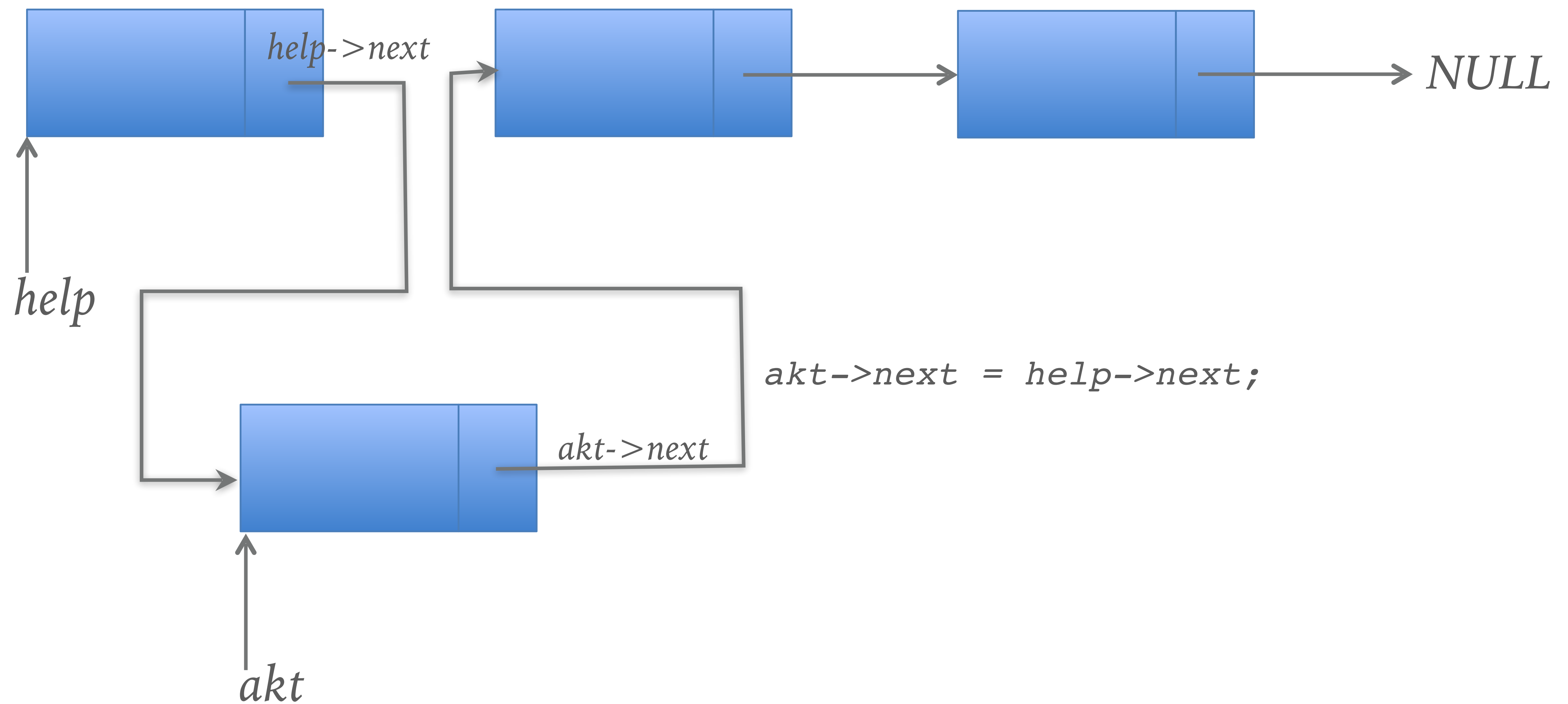


- Hinzufügen eines Elements

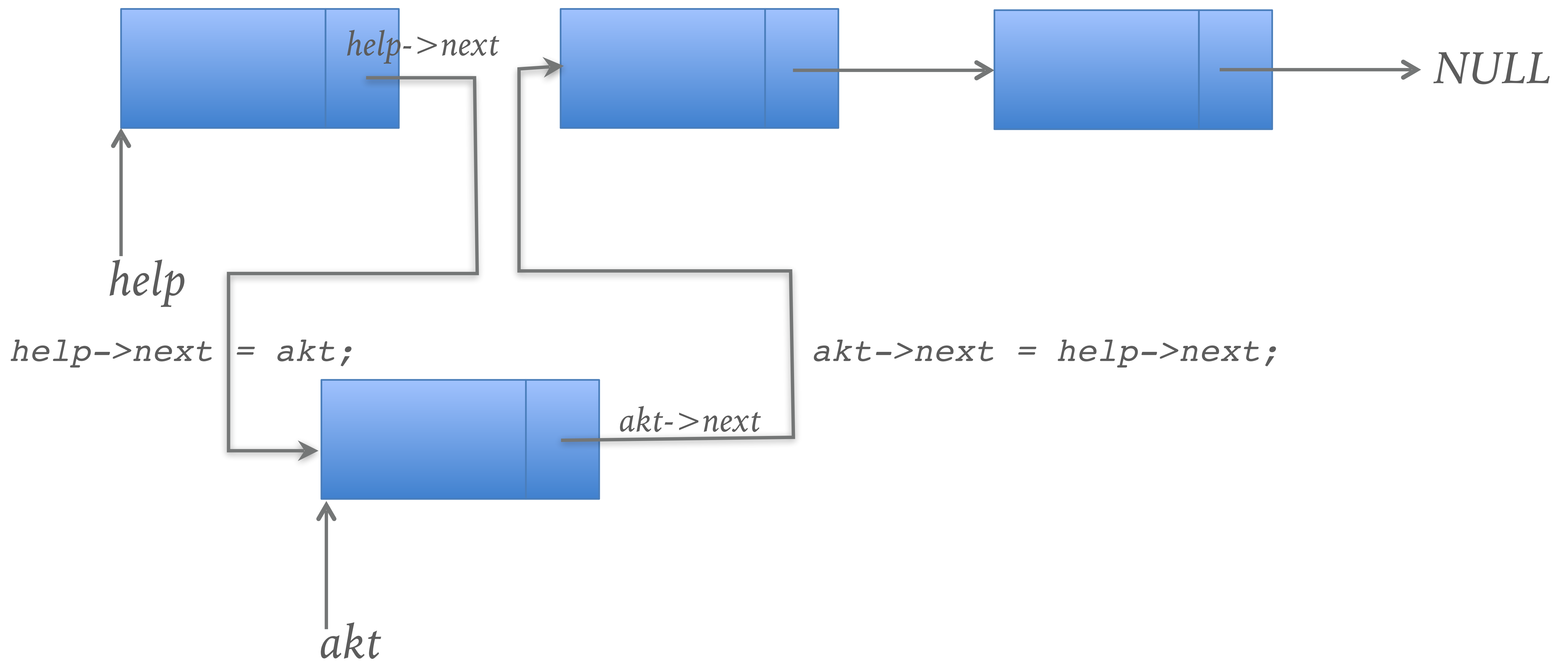


EINFACH VERKETTETE LISTEN

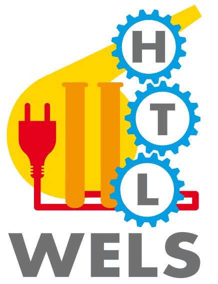
- Hinzufügen eines Elements



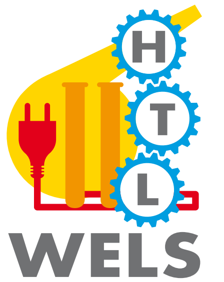
- Hinzufügen eines Elements



EINFACH VERKETTETE LISTEN



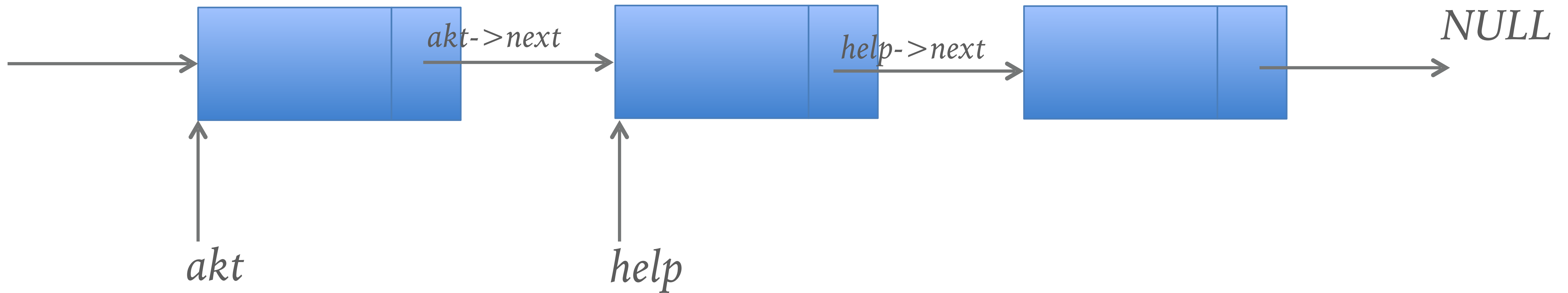
EINFACH VERKETTETE LISTEN



- Löschen eines Elements

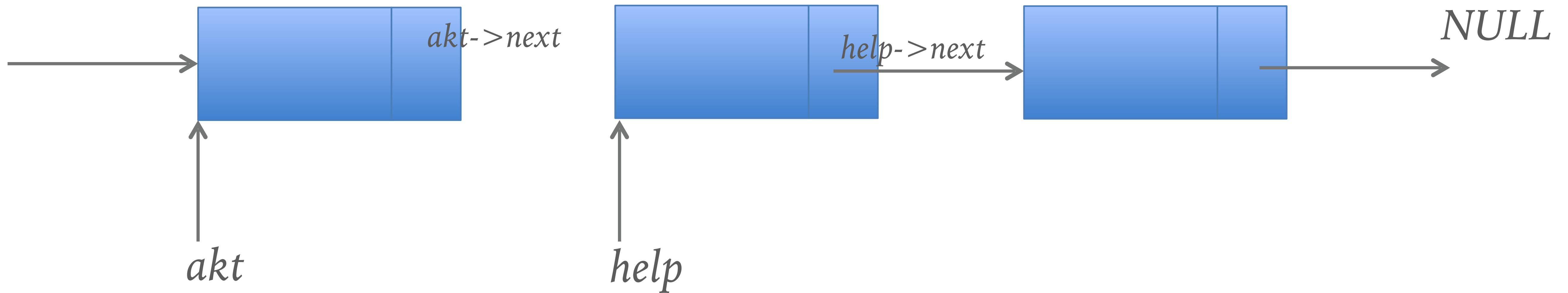
EINFACH VERKETTETE LISTEN

➤ Löschen eines Elements



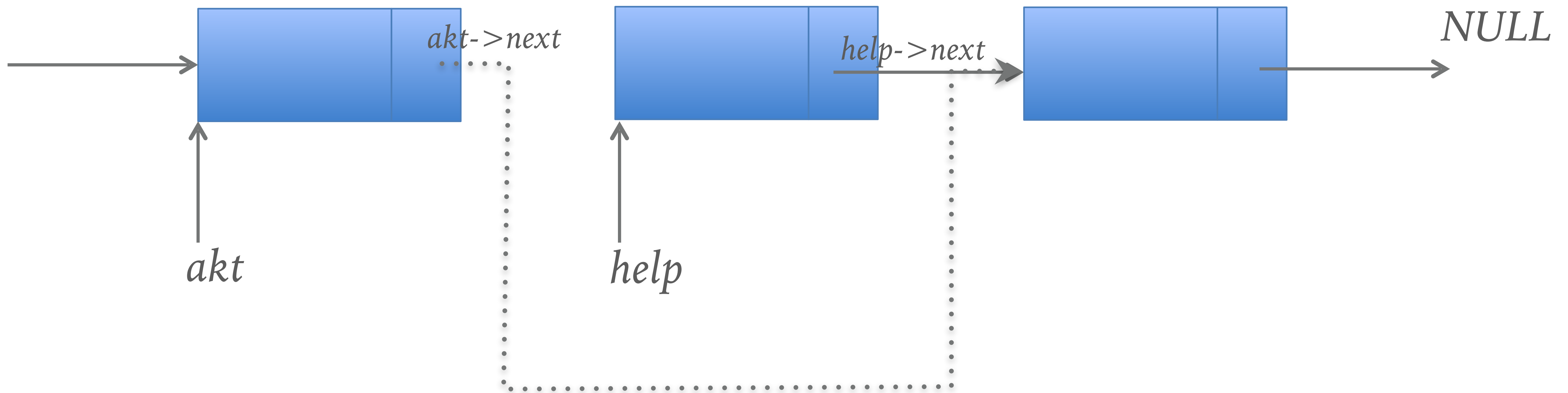
EINFACH VERKETTETE LISTEN

➤ Löschen eines Elements



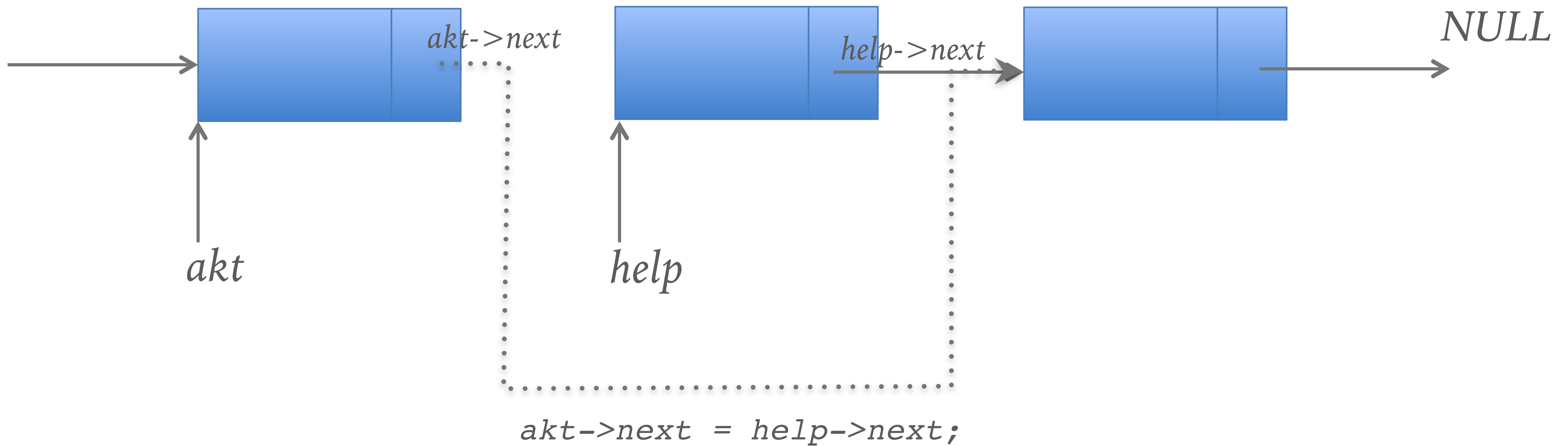
EINFACH VERKETTETE LISTEN

- Löschen eines Elements

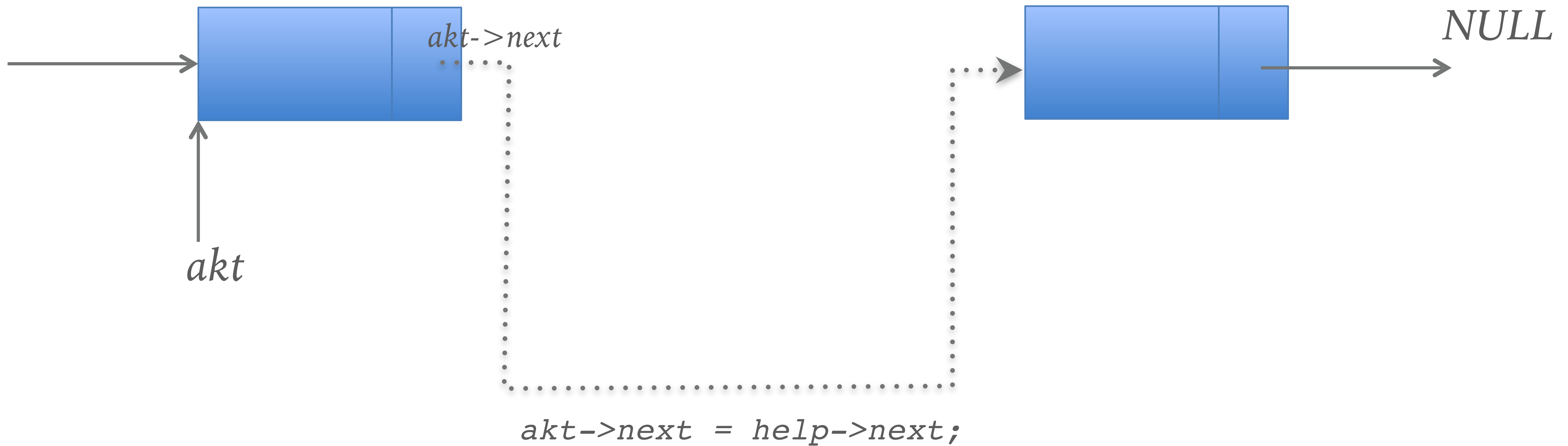


EINFACH VERKETTETE LISTEN

➤ Löschen eines Elements

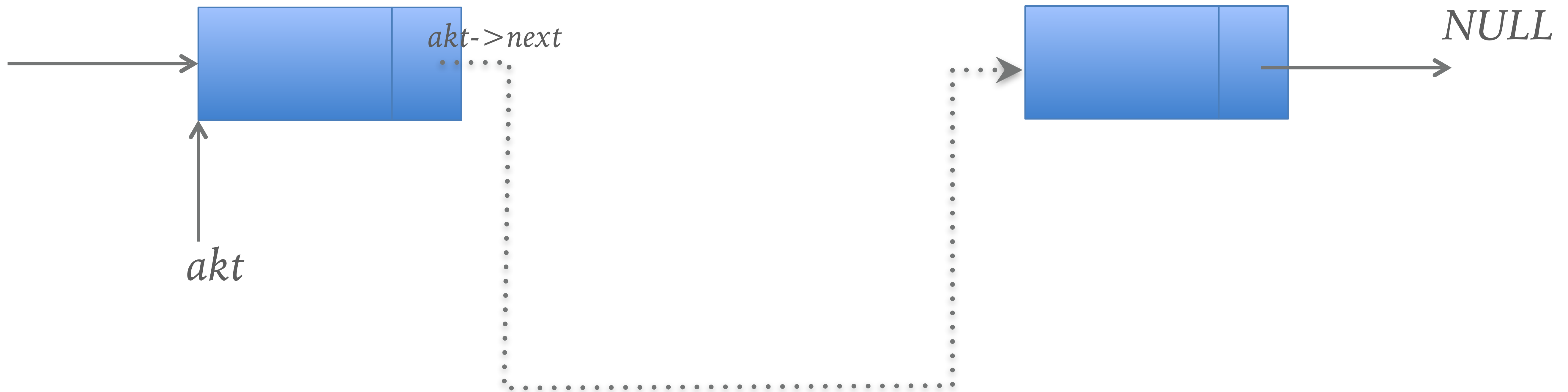


- Löschen eines Elements



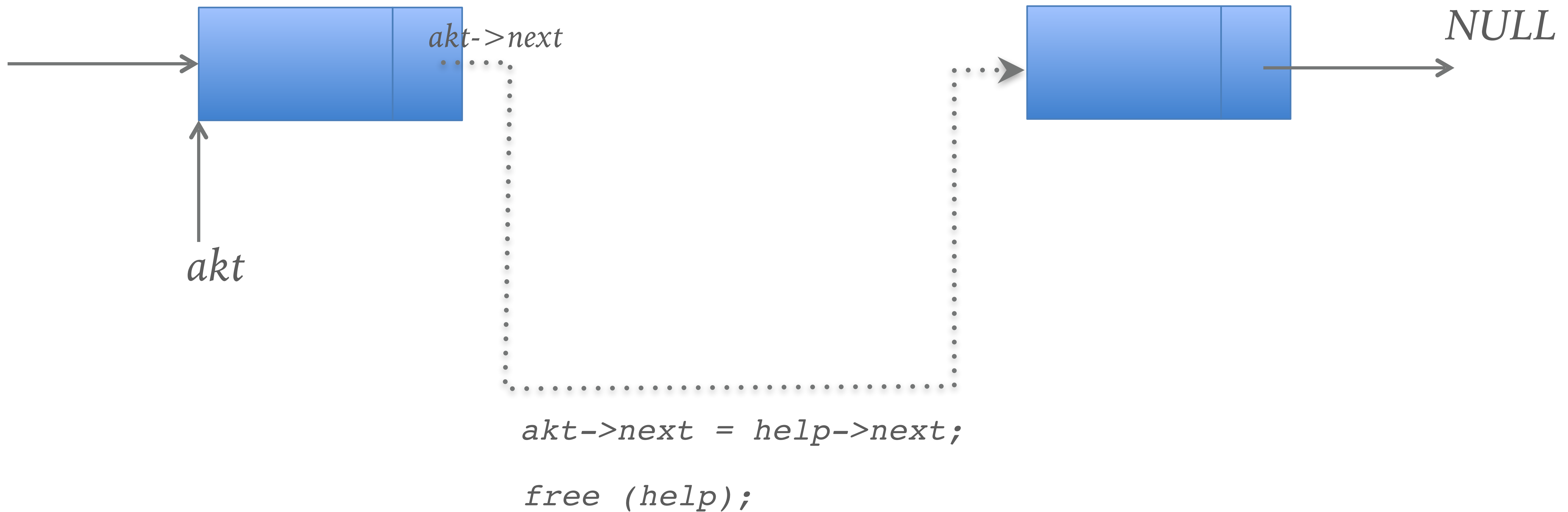
EINFACH VERKETTETE LISTEN

➤ Löschen eines Elements



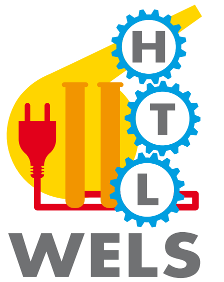
EINFACH VERKETTETE LISTEN

- Löschen eines Elements



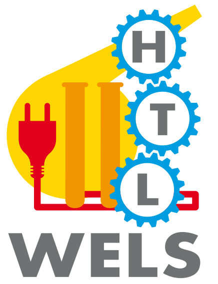
- *VORSICHT beim Löschen des ersten Elements!*

TIP ZUR EINFACHEREN PROGRAMMIERUNG

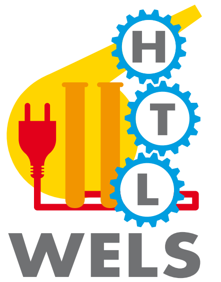


- Ein Tip zur Vereinfachung:
  - Lege zu Beginn ein `first` Element an (global?)
  - Lösche das `first` Element erst beim Programmende
  - in `first` sind keine Daten

DOPPELT VERKETTETE LISTEN

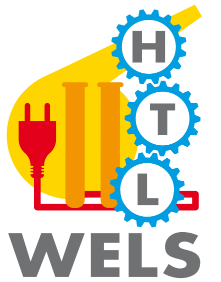


DOPPELT VERKETTETE LISTEN



- Vorteil:
    - schnelleres Navigieren durch Liste

DOPPELT VERKETTETE LISTEN



- Vorteil:
    - schnelleres Navigieren durch Liste

```
typedef struct data {
     char name[MAX_LEN];
     char vorname[MAX_LEN];
     struct data *next;
     struct data *prev;
}DATA;
```

DOPPELT VERKETTETE LISTEN

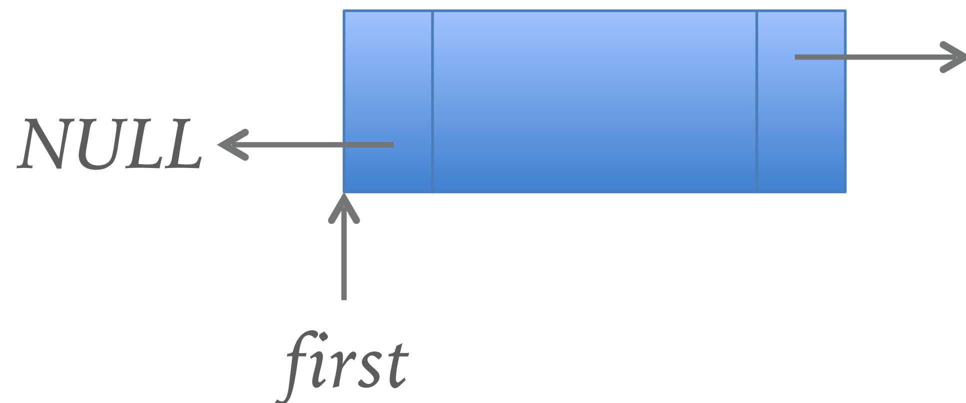
- Vorteil:
  - schnelleres Navigieren durch Liste

```
typedef struct data {
    char name[MAX_LEN];
    char vorname[MAX_LEN];
    struct data *next;
    struct data *prev;
}DATA;
```

DOPPELT VERKETTETE LISTEN

- Vorteil:
  - schnelleres Navigieren durch Liste

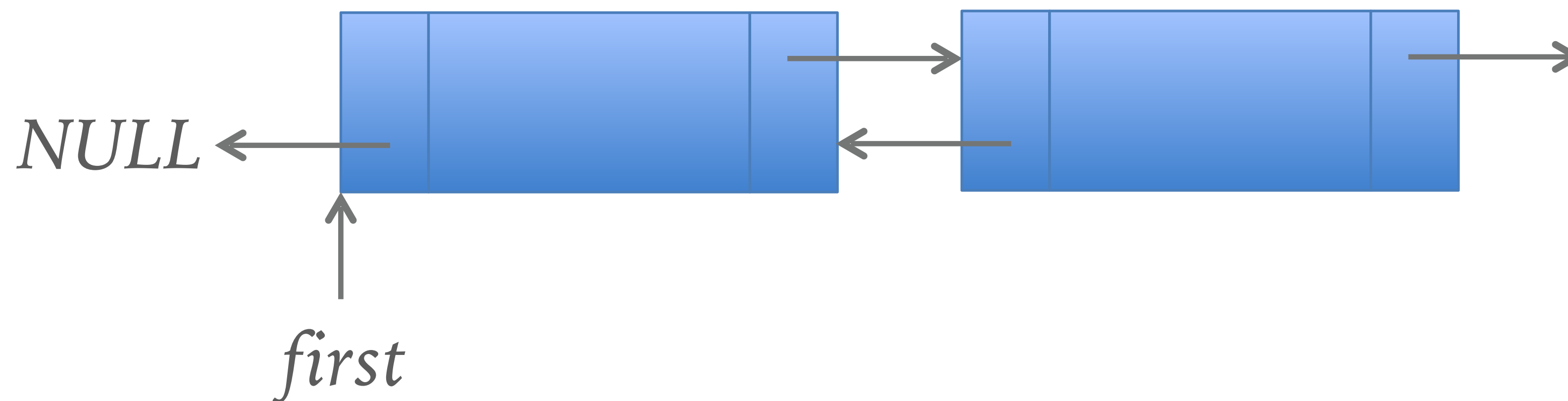
```c
typedef struct data {
    char name[MAX_LEN];
    char vorname[MAX_LEN];
    struct data *next;
    struct data *prev;
}DATA;
```



DOPPELT VERKETTETE LISTEN

- Vorteil:
  - schnelleres Navigieren durch Liste

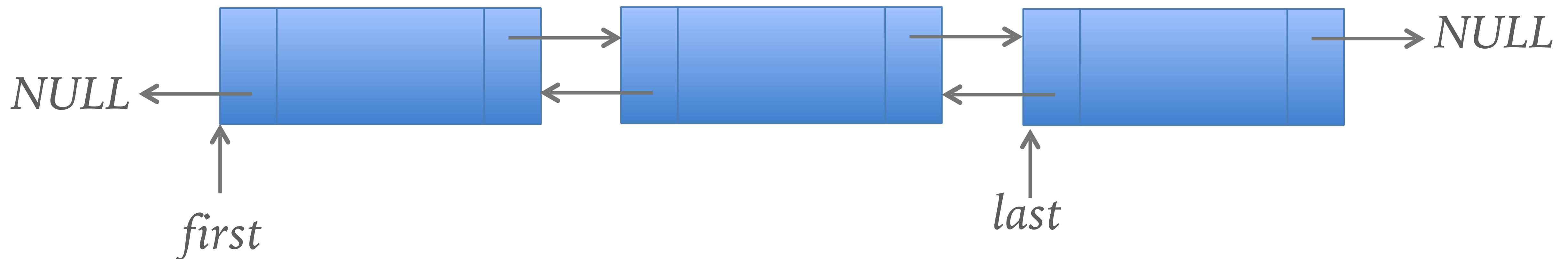
```
typedef struct data {
    char name[MAX_LEN];
    char vorname[MAX_LEN];
    struct data *next;
    struct data *prev;
}DATA;
```



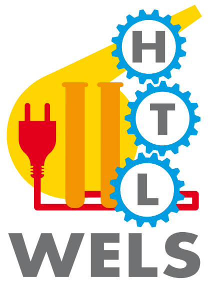
DOPPELT VERKETTETE LISTEN

- Vorteil:
  - schnelleres Navigieren durch Liste

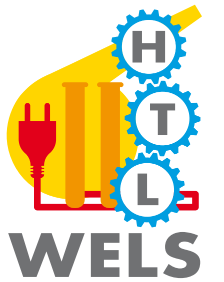
```
typedef struct data {
    char name[MAX_LEN];
    char vorname[MAX_LEN];
    struct data *next;
    struct data *prev;
}DATA;
```



DOPPELT VERKETTETE LISTEN



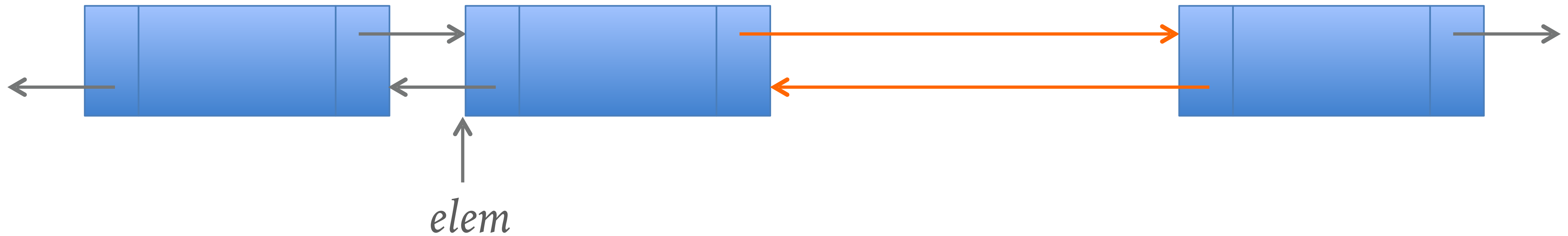
DOPPELT VERKETTETE LISTEN



- Einfügen von Elementen

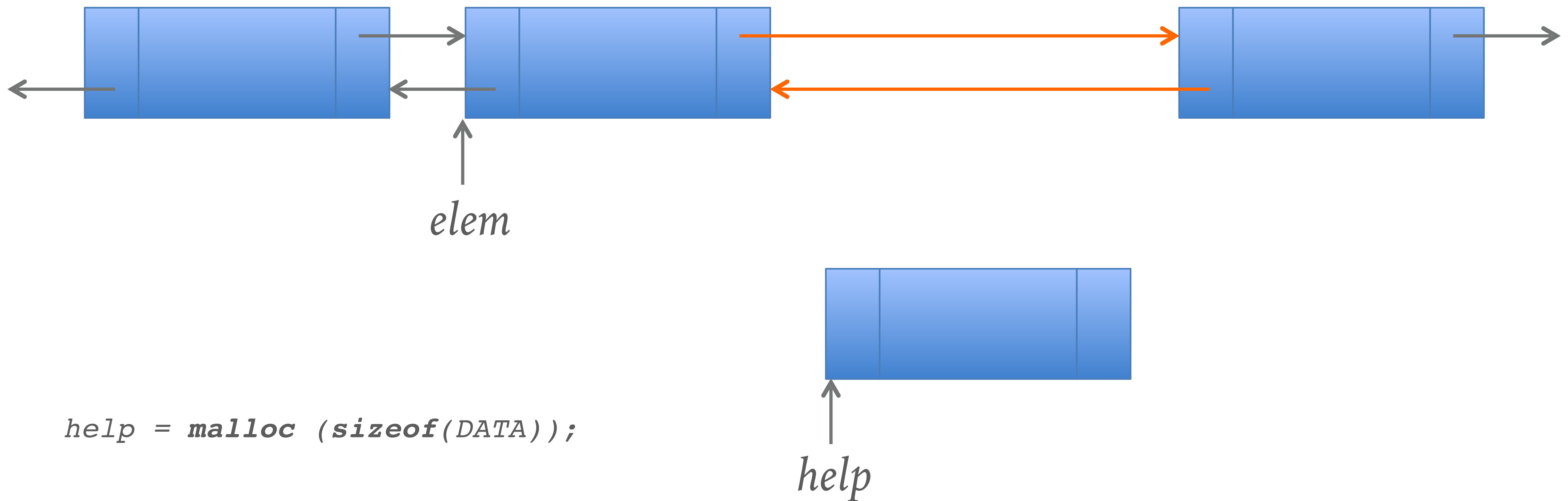
DOPPELT VERKETTETE LISTEN

- Einfügen von Elementen



DOPPELT VERKETTETE LISTEN

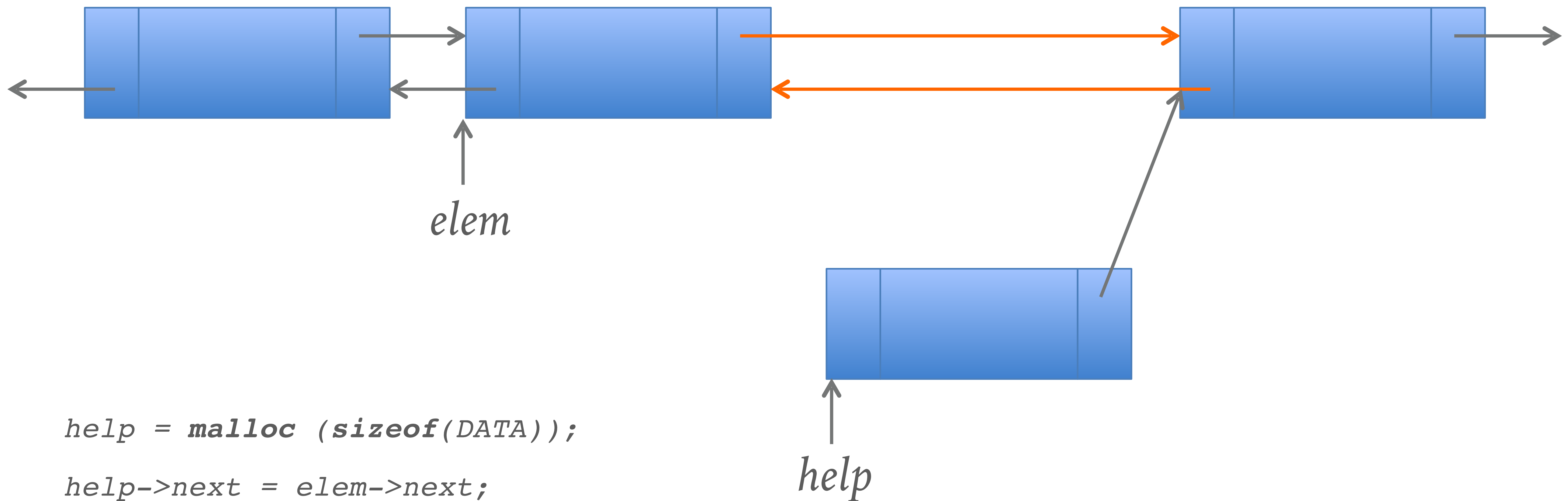
➤ Einfügen von Elementen




```
help = malloc (sizeof(DATA));
```

DOPPELT VERKETTETE LISTEN

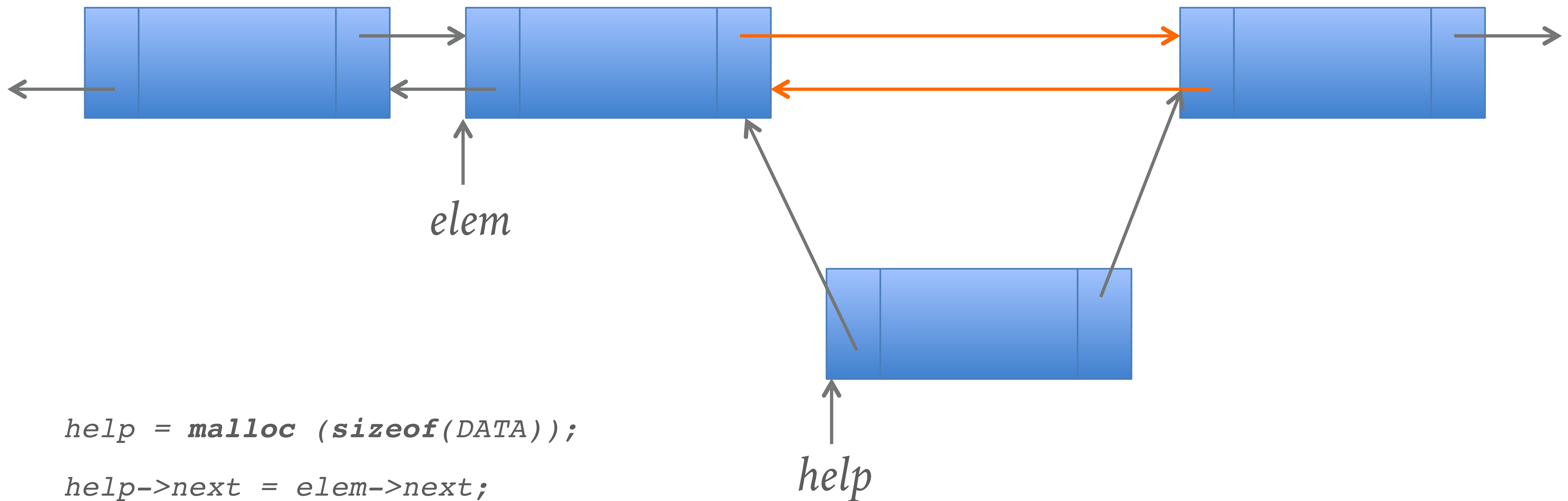
- Einfügen von Elementen




```
help = malloc (sizeof(DATA));
help->next = elem->next;
```

DOPPELT VERKETTETE LISTEN

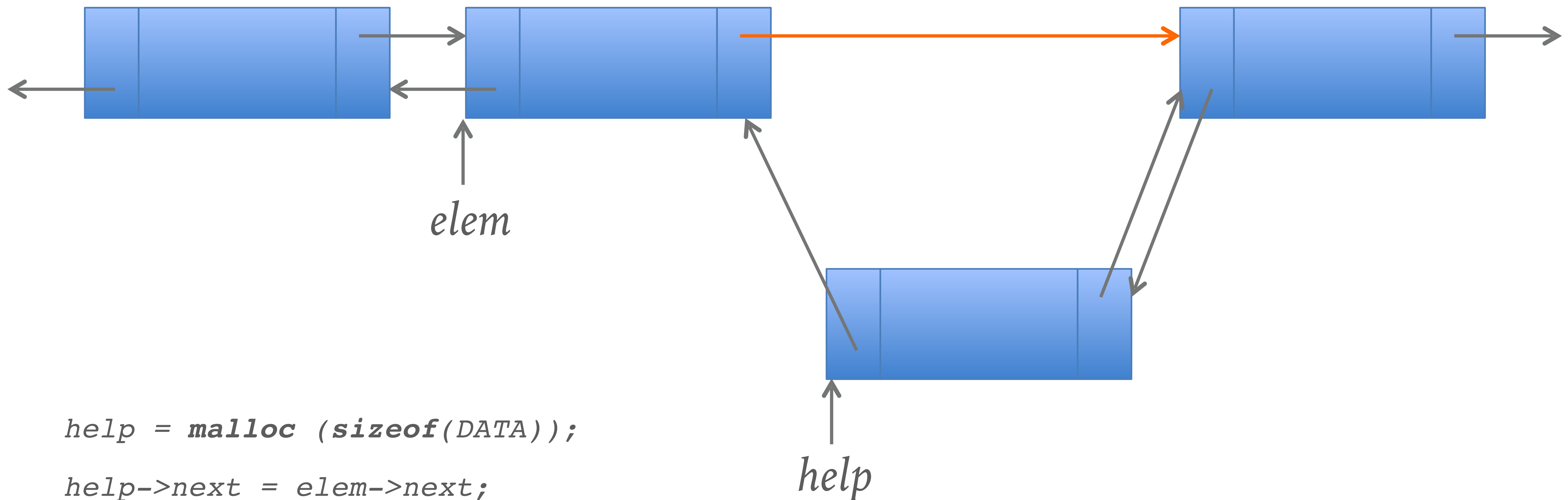
➤ Einfügen von Elementen




```
help = malloc (sizeof(DATA));
help->next = elem->next;
help->prev = elem;
```

DOPPELT VERKETTETE LISTEN

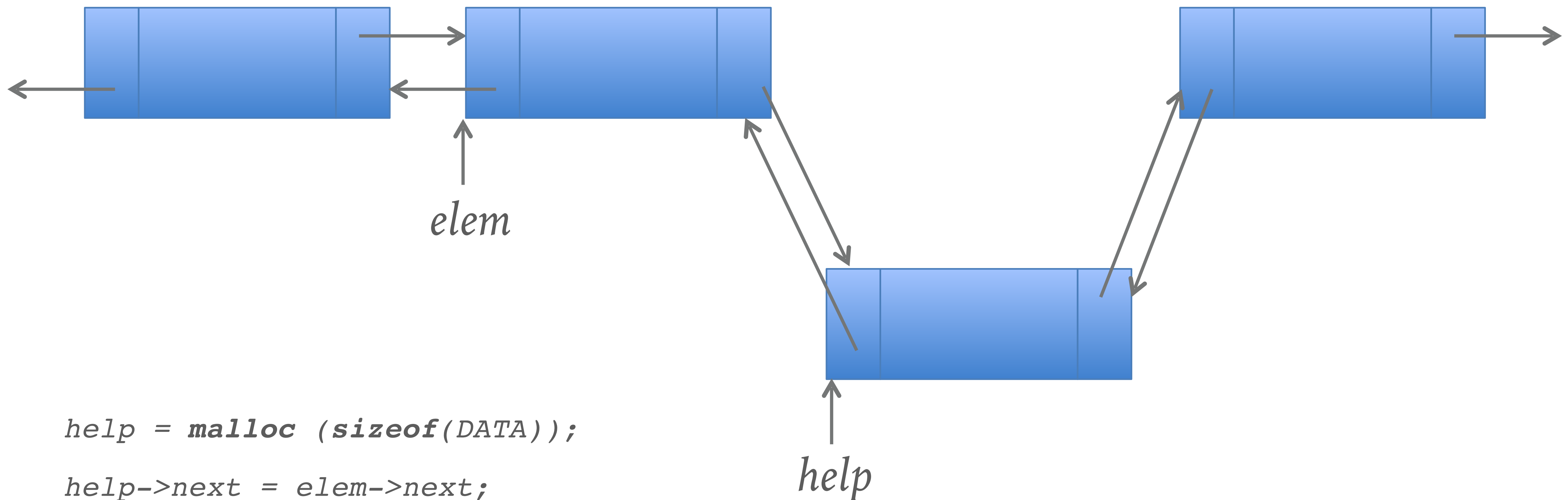
➤ Einfügen von Elementen




```
help = malloc (sizeof(DATA));
help->next = elem->next;
help->prev = elem;
elem->next->prev = help;
```

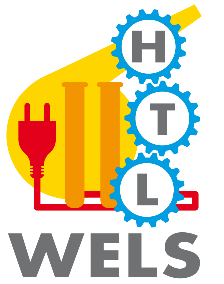
DOPPELT VERKETTETE LISTEN

➤ Einfügen von Elementen

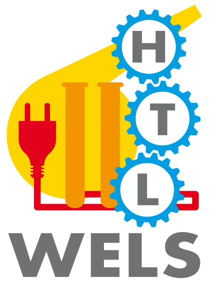



```
help = malloc (sizeof(DATA));
help->next = elem->next;
help->prev = elem;
elem->next->prev = help;
elem->next = help;
```

DOPPELT VERKETTETE LISTEN



DOPPELT VERKETTETE LISTEN



- Löschen von Elementen

DOPPELT VERKETTETE LISTEN

➤ Löschen von Elementen

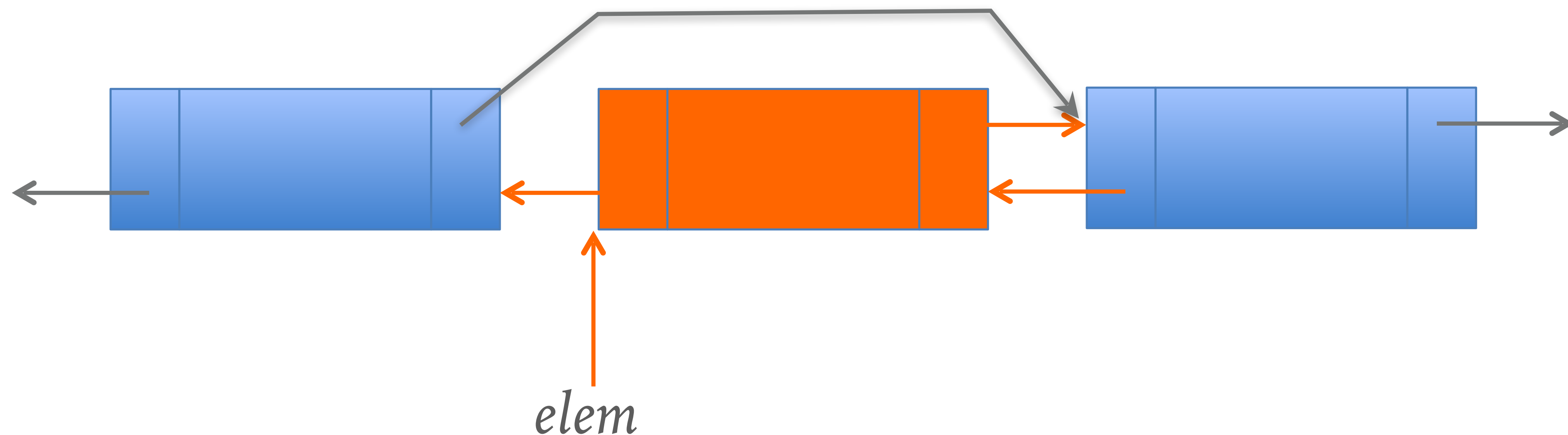
DOPPELT VERKETTETE LISTEN

➤ Löschen von Elementen

```
elem->prev->next=elem->next;
```

DOPPELT VERKETTETE LISTEN

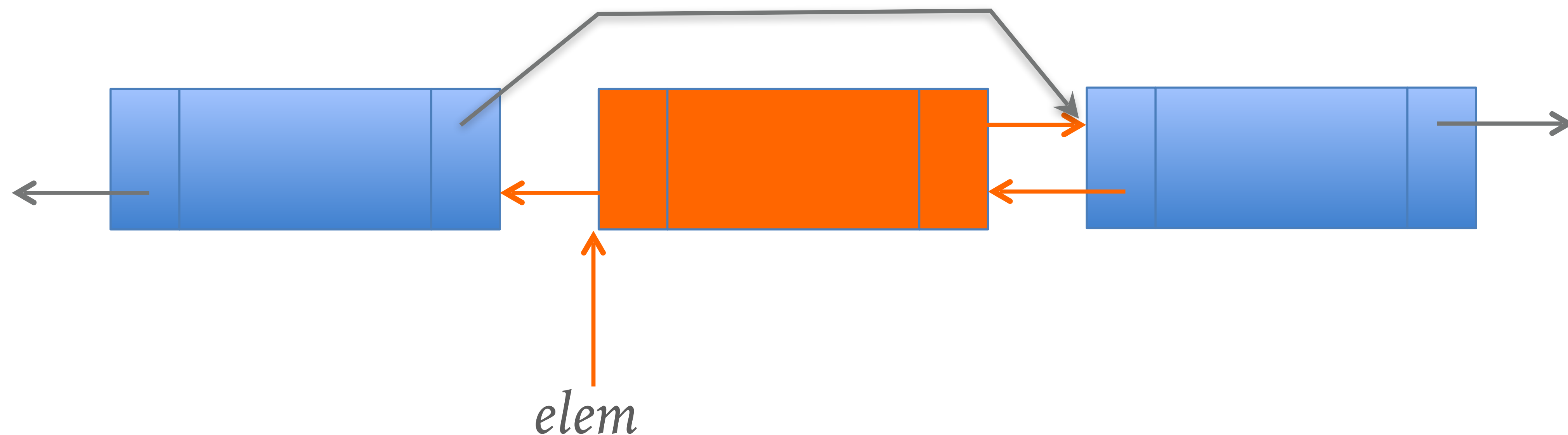
- Löschen von Elementen




```
elem->prev->next=elem->next;
```

DOPPELT VERKETTETE LISTEN

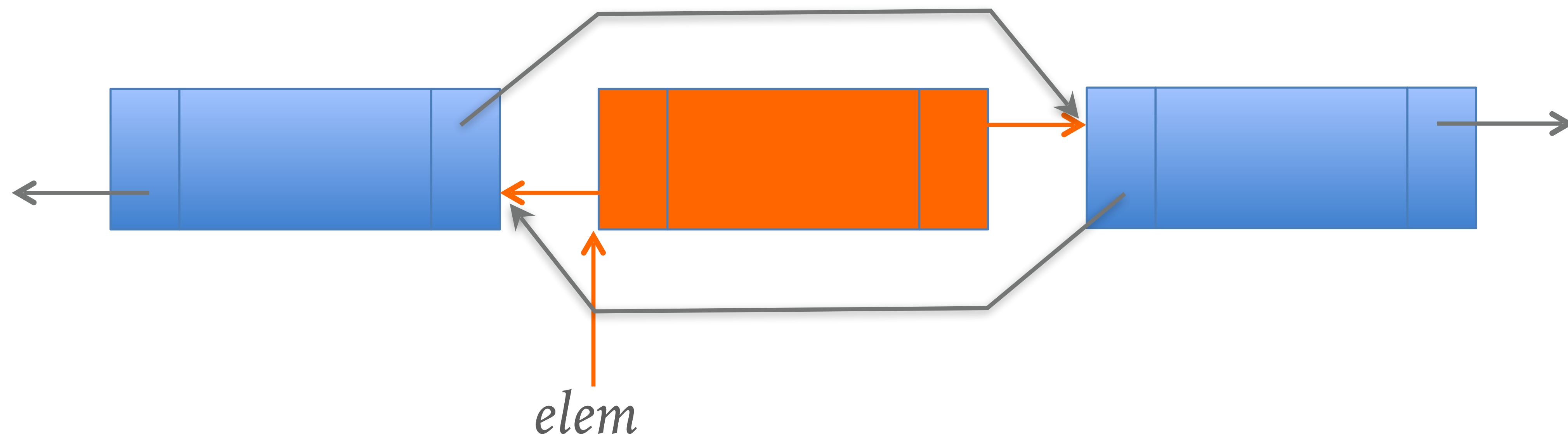
➤ Löschen von Elementen




```
elem->prev->next=elem->next;

elem->next->prev = elem->prev;
```

DOPPELT VERKETTETE LISTEN

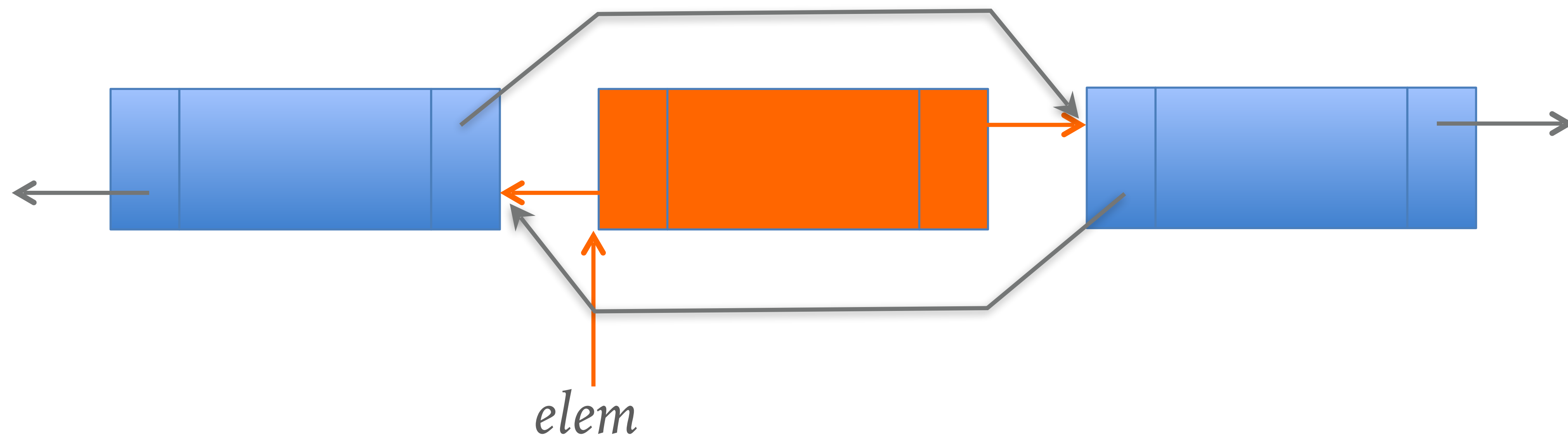
- Löschen von Elementen




```
elem->prev->next=elem->next;
elem->next->prev = elem->prev;
```

DOPPELT VERKETTETE LISTEN

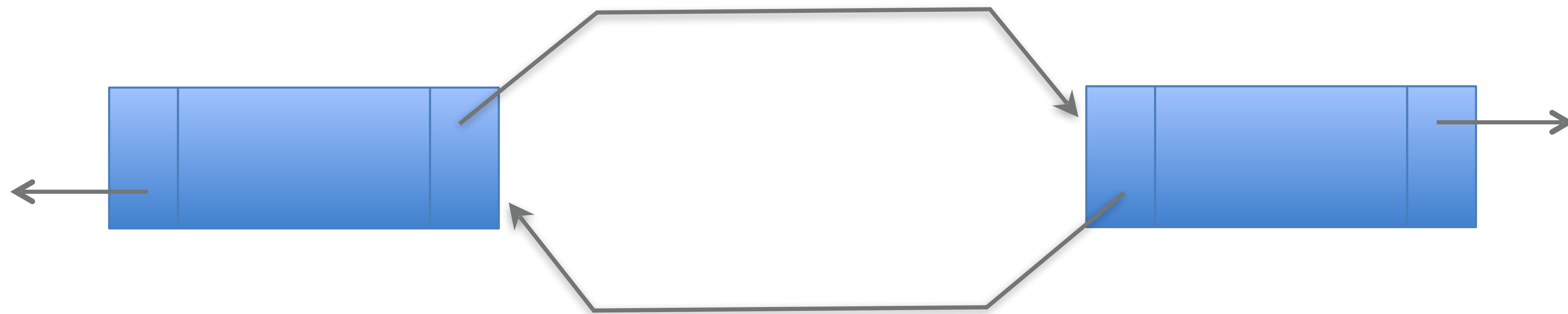
➤ Löschen von Elementen




```
elem->prev->next=elem->next;
elem->next->prev = elem->prev;
free (elem);
```

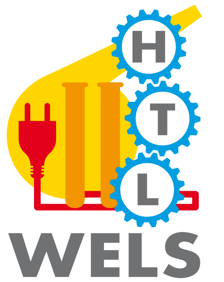
DOPPELT VERKETTETE LISTEN

- Löschen von Elementen



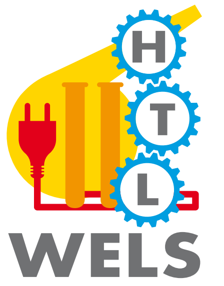
```
elem->prev->next=elem->next;
elem->next->prev = elem->prev;
free (elem);
```

RINGSTRUKTUR

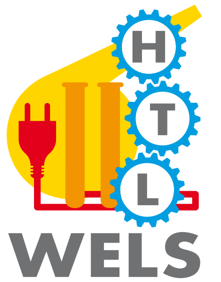


- Spezialfall einer verketteten Liste
- Letztes Element ist mit ersten verbunden
- gibt keinen Startknoten
- ein Knoten muss immer bekannt sein
- Achtung bei erstem/letztem Knoten

RINGSTRUKTUR

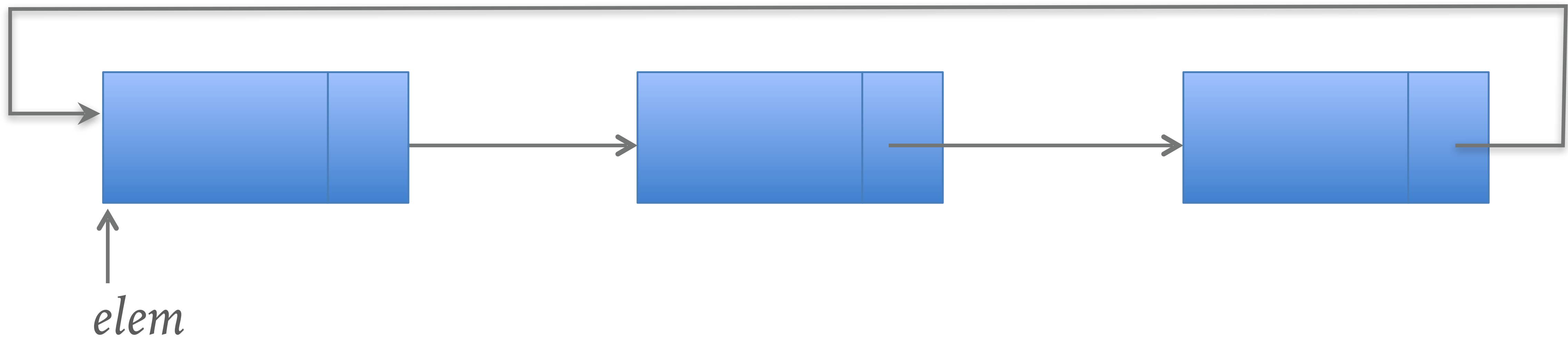


RINGSTRUKTUR

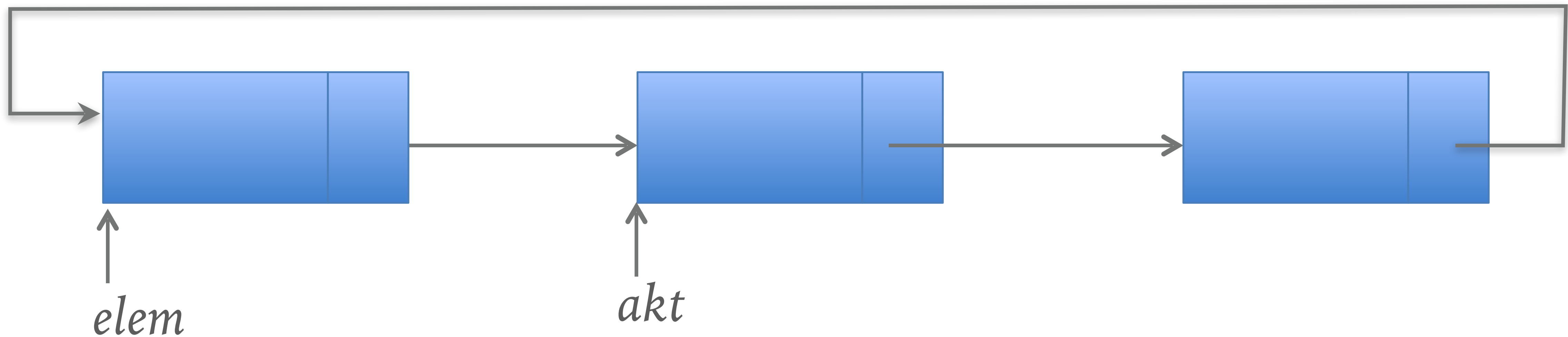


- Element löschen (einfach verkettet)

- Element löschen (einfach verkettet)



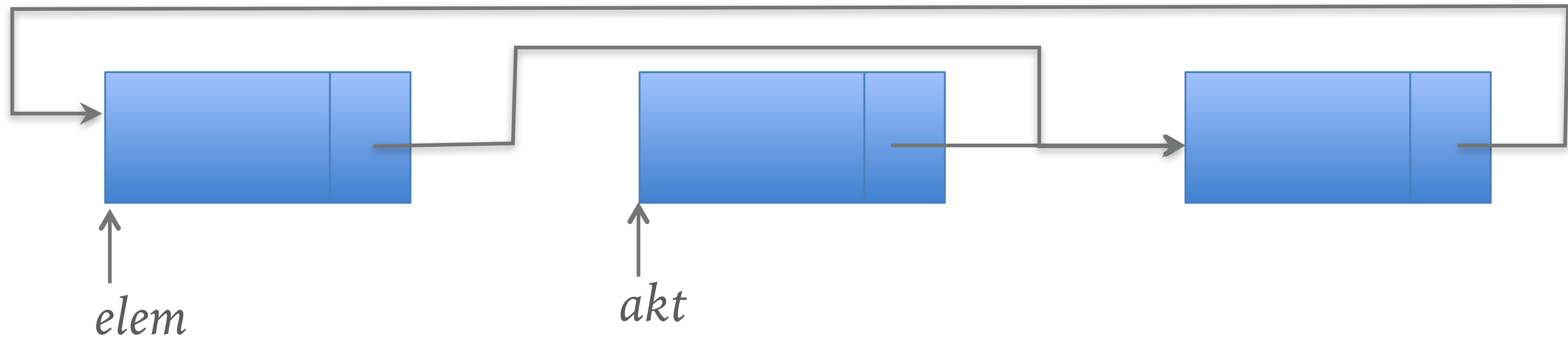
- Element löschen (einfach verkettet)




```
akt=elem->next;
```

RINGSTRUKTUR

- Element löschen (einfach verkettet)



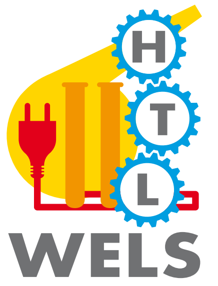

```
akt=elem->next;
elem->next = akt->next; // oder: elem->next->next;
```

RINGSTRUKTUR

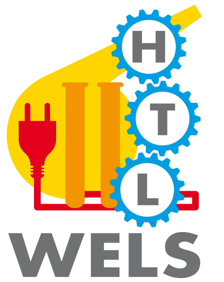
- Element löschen (einfach verkettet)

```
akt=elem->next;
elem->next = akt->next; // oder: elem->next->next;
free(akt);
```

RINGSTRUKTUR

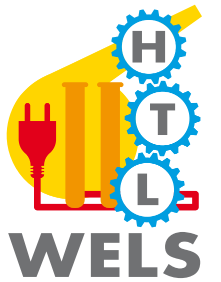


RINGSTRUKTUR



- Element erzeugen (doppelt verkettet)

RINGSTRUKTUR



- Element erzeugen (doppelt verkettet)

```
elem=malloc(sizeof(DATA));
```

RINGSTRUKTUR

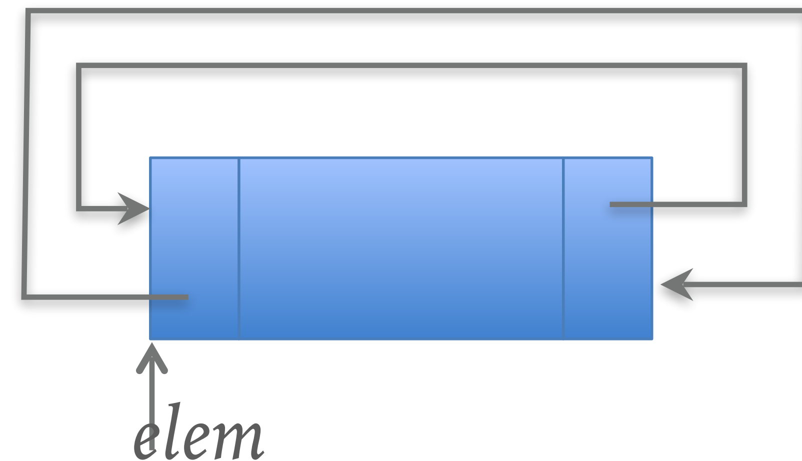
- Element erzeugen (doppelt verkettet)

```
elem=malloc(sizeof(DATA);

elem->next = elem;
```

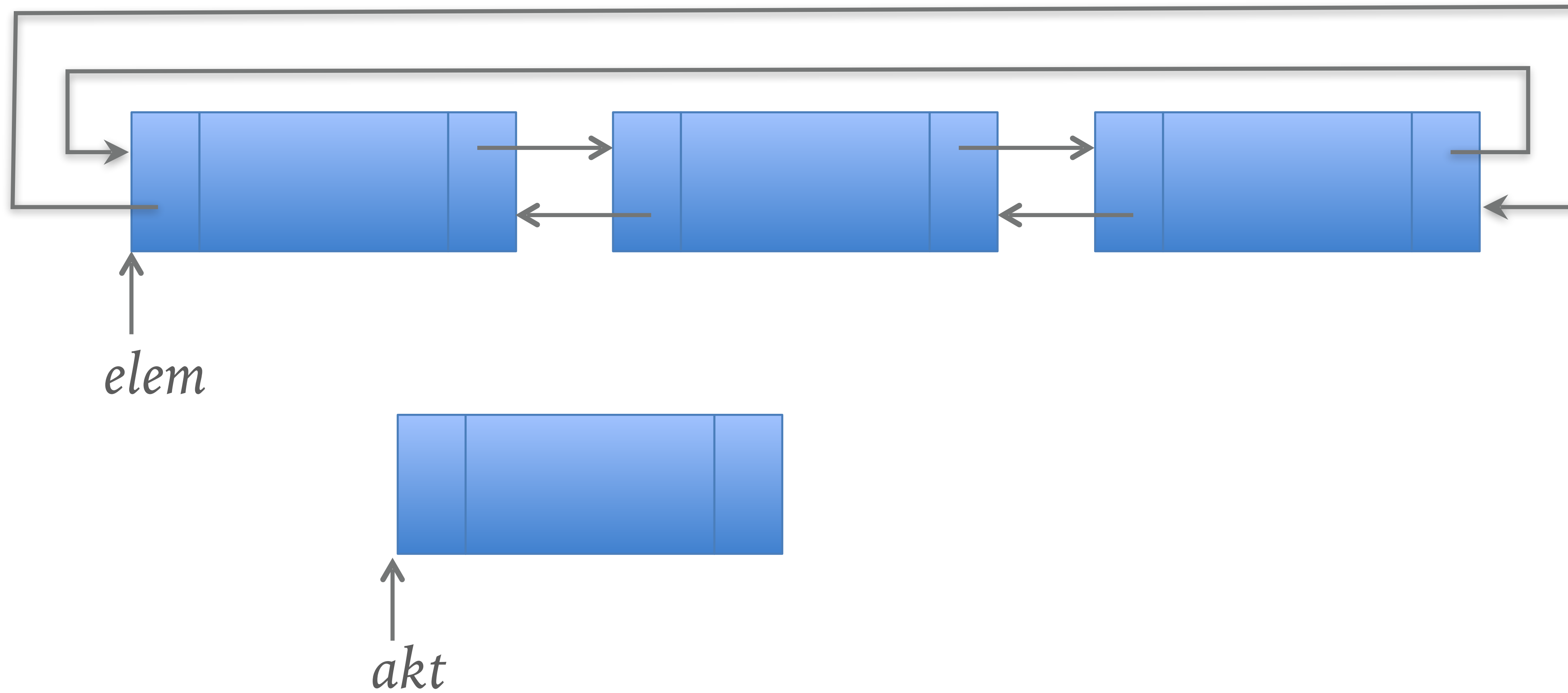
RINGSTRUKTUR

- Element erzeugen (doppelt verkettet)

```
elem=malloc(sizeof(DATA);
elem->next = elem;
elem->pref = elem;
```

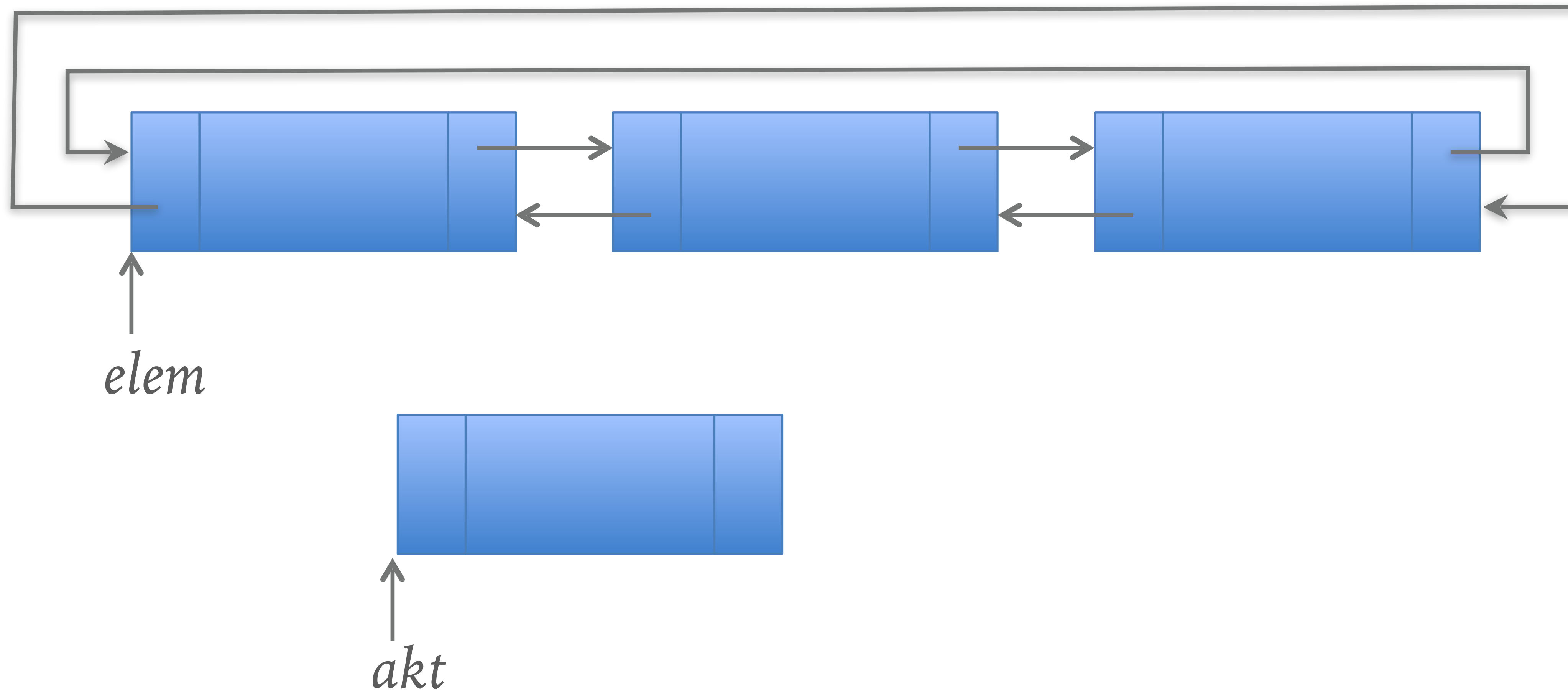


RINGSTRUKTUR



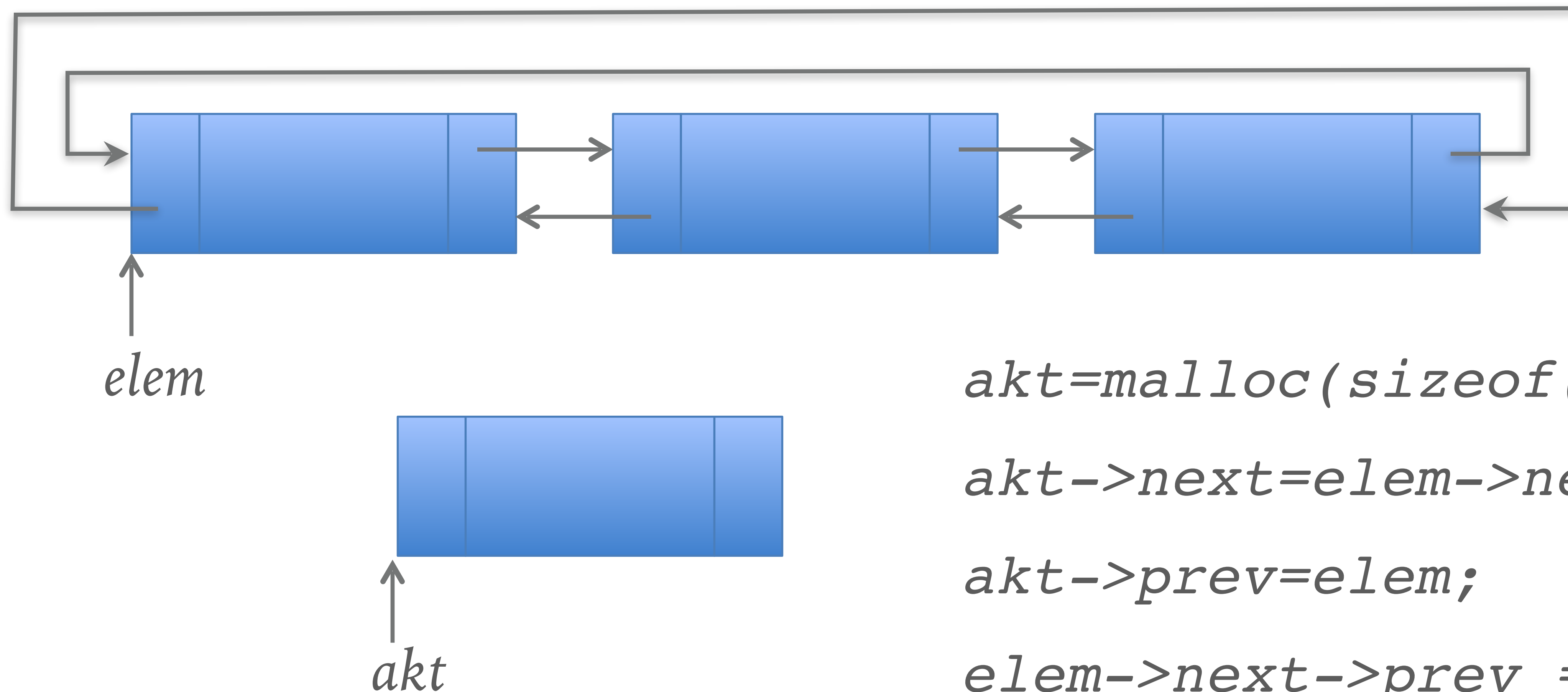
RINGSTRUKTUR

- Element einfügen (doppelt verkettet)



RINGSTRUKTUR

- Element einfügen (doppelt verkettet)




```
akt=malloc(sizeof(DATA));
akt->next=elem->next;
akt->prev=elem;
elem->next->prev = akt;
elem->next = akt;
```