Einführung in die Programmierung

Vorlesungsteil 10
Dynamische Datenstruktur Lineare Liste

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

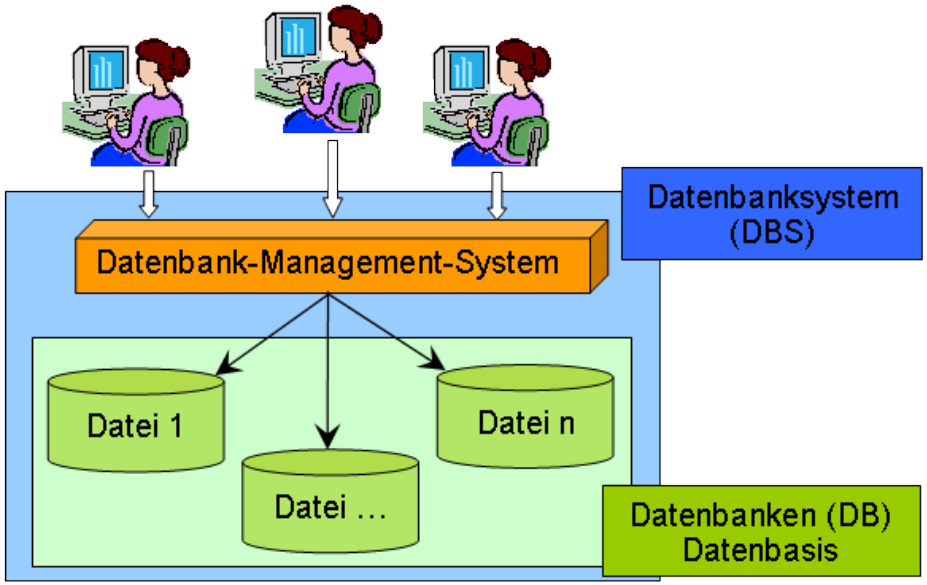
Datenbanken – eine Kernanwendung der Informatik

www.chip.de – Großrechner

- etwa 40% des Umsatzes der Softwarebranche in Deutschland über *Datenbanksysteme**
- zahlreiche *Internetdienste* (wie online-Shopping oder Cloud-Dienste) ohne Datenbanken nicht möglich
- Datenbank-Softwareentwicklung in Deutschland *finanziell am einträglichsten* im Vergleich zu anderen Programmiersparten*

* Studie und Gehaltsumfrage des c't-Magazins für Computertechnik, Heise-Verlag, 2013

Was zeichnet ein gutes Datenbanksystem aus?



www.info-wsf.de

- *Datenbanksystem* ist ein *Programm*, das idealerweise *permanent* fehlerfrei und ohne Abstürze *läuft*
- Während der Laufzeit können der oder die Nutzer:
  neue Datensätze anlegen, *Datensätze löschen*,
  nach Datensätzen suchen und *Datensätze auswerten*
- Datenbanksystem sichert Datenbestand regelmäßig in Dateien und hält den *Datenbestand* komplett oder auszugsweise *im Arbeitsspeicher* für schnelleren Zugriff

Herausforderung: Wie programmiert man das?

Anforderung: Während der Laufzeit des Programms werden neue Datensätze angelegt oder bestehende gelöscht.

Herausforderung: Wie programmiert man das?

Anforderung: Während der Laufzeit des Programms werden neue Datensätze angelegt oder bestehende gelöscht.

Zur Erfassung großer Datenmengen kennen wir bisher nur die Datenstruktur *Feld* (engl. array).

Herausforderung: Wie programmiert man das?

Anforderung: Während der Laufzeit des Programms werden neue Datensätze angelegt oder bestehende gelöscht.

Zur Erfassung großer Datenmengen kennen wir bisher nur die Datenstruktur *Feld* (engl. array).

Problem: Die *Feldgröße* (Anzahl Feldelemente) und mithin die Anzahl erfassbarer Datensätze lässt sich nach dem Anlegen des Feldes nicht mehr verändern, in C muss sie sogar schon *beim Compilieren des Programms bekannt* sein.

Herausforderung: Wie programmiert man das?

Anforderung: Während der Laufzeit des Programms werden neue Datensätze angelegt oder bestehende gelöscht.

Zur Erfassung großer Datenmengen kennen wir bisher nur die Datenstruktur *Feld* (engl. array).

> **Problem:** Die *Feldgröße* (Anzahl Feldelemente) und mithin die Anzahl erfassbarer Datensätze lässt sich nach dem Anlegen des Feldes nicht mehr verändern, in C muss sie sogar schon *beim Compilieren des Programms bekannt* sein.

Erste Idee: Ein *riesiges Feld anlegen* in der Hoffnung, dass seine Größe stets ausreicht.

Herausforderung: Wie programmiert man das?

Anforderung: Während der Laufzeit des Programms werden neue Datensätze angelegt oder bestehende gelöscht.

Zur Erfassung großer Datenmengen kennen wir bisher nur die Datenstruktur *Feld* (engl. array).

Problem: Die *Feldgröße* (Anzahl Feldelemente) und mithin die Anzahl erfassbarer Datensätze lässt sich nach dem Anlegen des Feldes nicht mehr verändern, in C muss sie sogar schon *beim Compilieren des Programms bekannt* sein.

Erste Idee: Ein *riesiges Feld anlegen* in der Hoffnung, dass seine Größe stets ausreicht.

Nachteile: Ist der Datenbestand klein, verschwendet man sehr viel Speicherplatz. Wächst der Datenbestand immer weiter, wird das statische Feld irgendwann zu klein.

Herausforderung: Wie programmiert man das?

Anforderung: Während der Laufzeit des Programms werden neue Datensätze angelegt oder bestehende gelöscht.

Zur Erfassung großer Datenmengen kennen wir bisher nur die Datenstruktur *Feld* (engl. array).

Problem: Die *Feldgröße* (Anzahl Feldelemente) und mithin die Anzahl erfassbarer Datensätze lässt sich nach dem Anlegen des Feldes nicht mehr verändern, in C muss sie sogar schon *beim Compilieren des Programms bekannt* sein.

Erste Idee: Ein *riesiges Feld anlegen* in der Hoffnung, dass seine Größe stets ausreicht.

Nachteile: Ist der Datenbestand klein, verschwendet man sehr viel Speicherplatz. Wächst der Datenbestand immer weiter, wird das statische Feld irgendwann zu klein.

„Wir brauchen eine flexible Datenstruktur, deren Größe (Anzahl enthaltene Datensätze) sich *dynamisch* den Nutzeranforderungen anpasst.“

Eine Datenstruktur nach Idee einer Schnipseljagd

oder der Schatzsuche

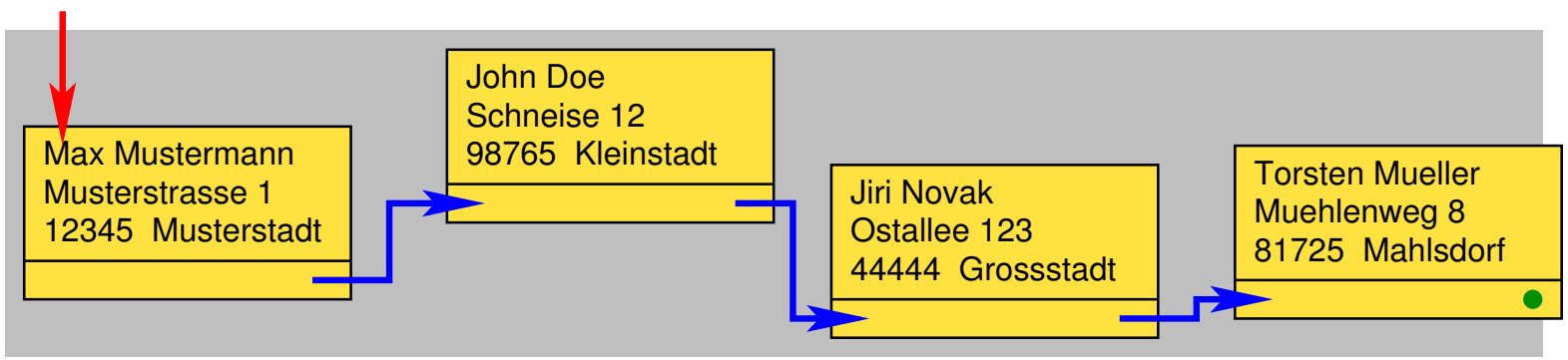
- Infoschnipsel (Datensätze) beliebig im Gelände (Speicher) verteilt
- Jagd beginnt an einem bekannten Startpunkt (hier: Infoschnipsel „A“)
- Auf jedem Infoschnipsel steht, wo sich nächster Infoschnipsel befindet
- Man läuft durch das Gelände von Infoschnipsel zu Infoschnipsel
- Beim letzten Infoschnipsel endet der Lauf bzw. liegt eine Belohnung

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

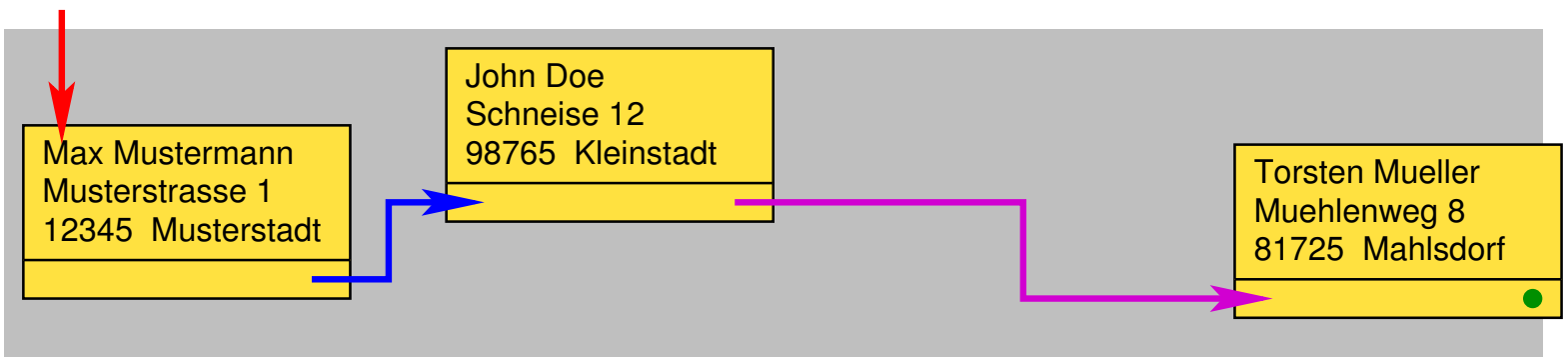
Dynamische Datenstruktur Lineare Liste

Eine **Lineare Liste** ist eine dynamische Datenstruktur, deren *Elemente (Datensätze)* beliebig im Speicher verteilt sein können. Jedes Element der Liste – bis auf das letzte – hat genau einen Nachfolger. Der *Zugriff* auf die Liste erfolgt üblicherweise über das erste Element. Jedes Element einer Liste enthält einen *Vermerk*, wo sich im Speicher das Nachfolgerelement befindet. Beim letzten Element wird stattdessen als Vermerk eingetragen, dass es *kein Nachfolgerelement* gibt.



Dynamische Datenstruktur Lineare Liste

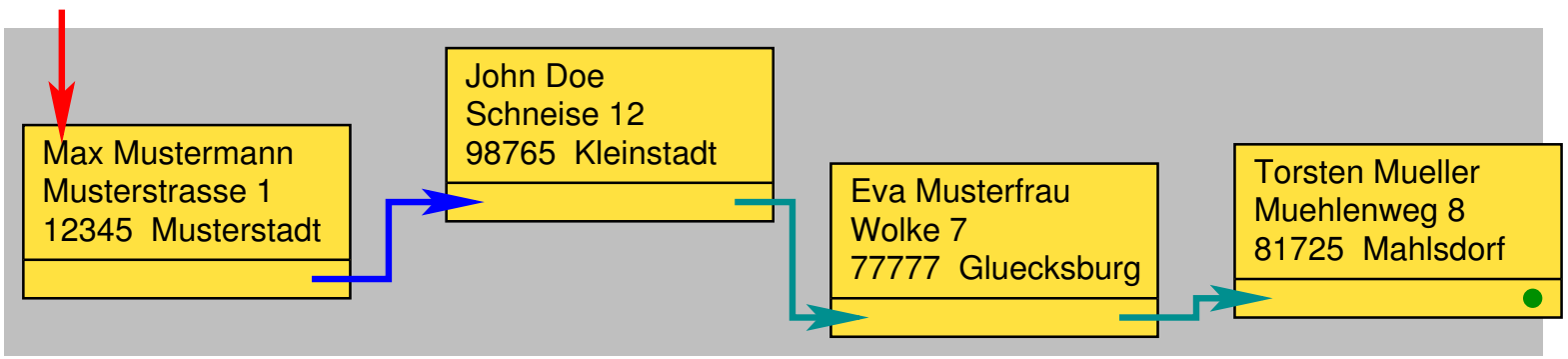
Eine **Lineare Liste** ist eine dynamische Datenstruktur, deren *Elemente (Datensätze)* beliebig im Speicher verteilt sein können. Jedes Element der Liste – bis auf das letzte – hat genau einen Nachfolger. Der *Zugriff* auf die Liste erfolgt üblicherweise über das erste Element. Jedes Element einer Liste enthält einen *Vermerk*, wo sich im Speicher das Nachfolgerelement befindet. Beim letzten Element wird stattdessen als Vermerk eingetragen, dass es *kein Nachfolgerelement* gibt.



Beim Loeschen eines Datensatzes werden betroffene Vermerke aktualisiert.

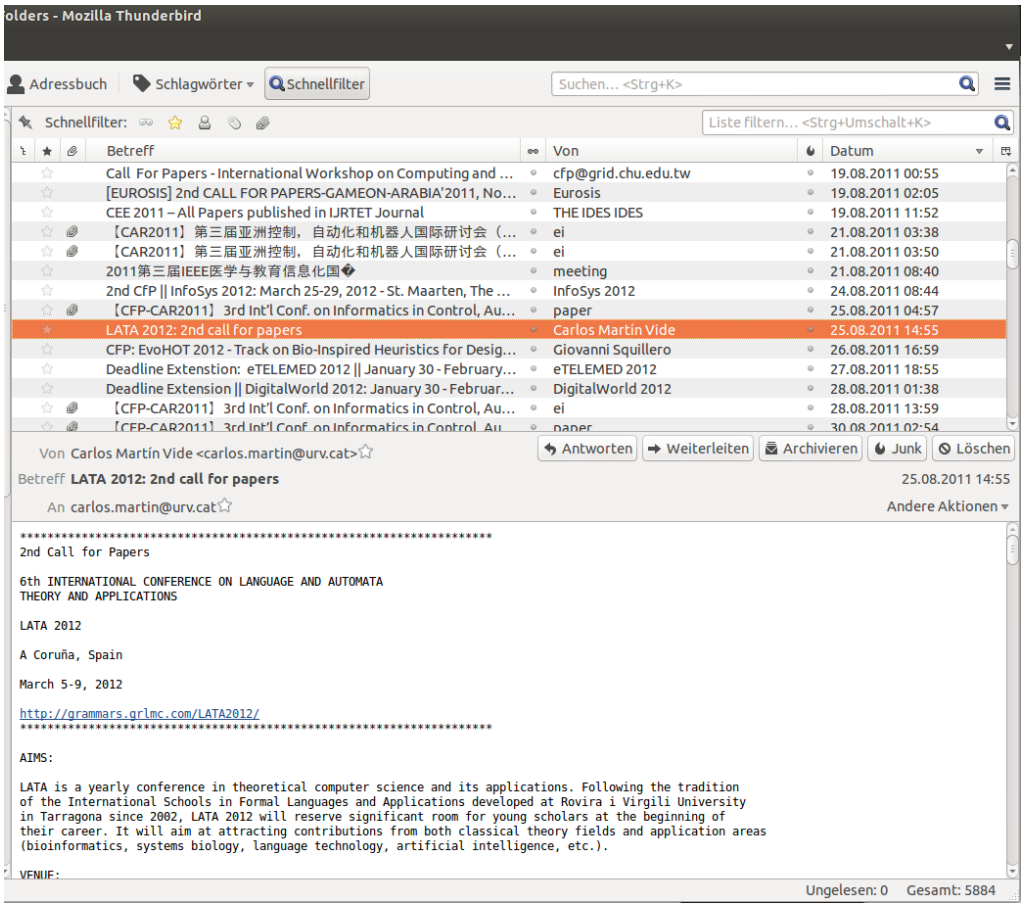
Dynamische Datenstruktur Lineare Liste

Eine **Lineare Liste** ist eine dynamische Datenstruktur, deren *Elemente (Datensätze)* beliebig im Speicher verteilt sein können. Jedes Element der Liste – bis auf das letzte – hat genau einen Nachfolger. Der *Zugriff* auf die Liste erfolgt üblicherweise über das erste Element. Jedes Element einer Liste enthält einen *Vermerk*, wo sich im Speicher das Nachfolgerelement befindet. Beim letzten Element wird stattdessen als Vermerk eingetragen, dass es *kein Nachfolgerelement* gibt.



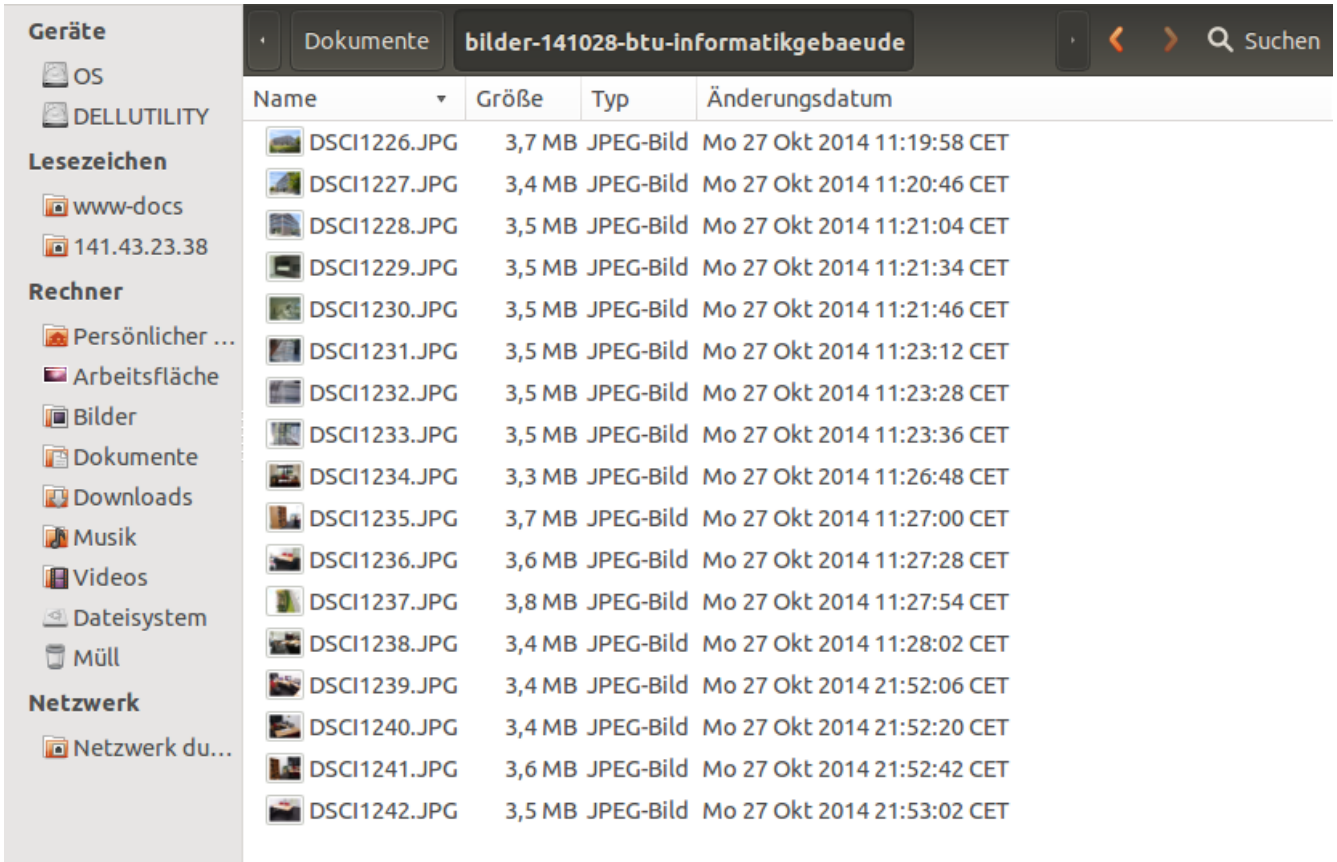
Beim Einfuegen eines neuen Datensatzes werden die betroffenen Vermerke ebenfalls aktualisiert.

Liste von E-Mails verwaltet vom E-Mail-Programm



Jede E-Mail ist ein Listenelement. E-Mails können angezeigt, sortiert, hinzugefügt und gelöscht werden.

Liste von Dateien in einem Verzeichnis



Jede Datei ist ein Listenelement. Dateien können angezeigt, sortiert, hinzugefügt und gelöscht werden.

Kassenzettel als Liste von Artikeln

Jeder Artikel ist ein Listenelement. Artikel können hinzugefügt, storniert (gelöscht), angezeigt und zur Gesamtpreisberechnung ausgewertet werden.

Elektronischer Warenkorb als Liste von Artikeln

Jeder Artikel ist ein Listenelement. Artikel können hinzugefügt, gelöscht, angezeigt und zur Gesamtpreisberechnung ausgewertet werden.

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Steckbrief Lineare Liste

- *dynamische Datenstruktur* (Größe kann variieren)
- Datensätze (Listenelemente) beliebig im Speicher *verteilt*
- während Programmlaufzeit können neue Datensätze angelegt, in die Liste *eingefügt* und bestehende Datensätze *gelöscht* werden
- *kein Direktzugriff* auf beliebigen Datensatz
- stattdessen Zugriff auf ersten Datensatz und dann von dort aus Datensatz für Datensatz *durch die Liste „hangeln“*
- Jeder Datensatz – bis auf den letzten – enthält *Anfangsadresse des nachfolgenden Datensatzes* im Speicher
- letzter Datensatz enthält stattdessen einen *Endevermerk*
- alle Datensätze besitzen *gleiches Datenformat*
- Lineare Liste bevorzugt für *kleine bis mittelgroße Datenbestände* (einige tausend Datensätze)

Einen Datensatz in C erfassen mittels `struct`

Beispiel: Uhrzeit aus Stunde, Minute und Sekunde (weckzeit.c)

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};                   //Semikolon am Ende nicht vergessen!


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         myalarm.stunde, myalarm.minute, myalarm.sekunde, myalarm.zeitzone);
  return 0;
}
```

```
Meine Weckzeit:  7 : 30 :  0 (MEZ)
```

- `struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};` legt Variable `myalarm` vom Typ `struct TUhrzeit` an.
- Komponenten werden in der Reihenfolge, wie sie in der Struktur definiert sind, mit Werten belegt.

Einen Datensatz in C erfassen mittels `struct`

Beispiel: Uhrzeit aus Stunde, Minute und Sekunde (weckzeit.c)

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};                   //Semikolon am Ende nicht vergessen!


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         myalarm.stunde, myalarm.minute, myalarm.sekunde, myalarm.zeitzone);
  return 0;
}
```

```
Meine Weckzeit:  7 : 30 :  0 (MEZ)
```

- Punktoperator `.` erlaubt lesenden wie schreibenden Zugriff auf die Komponenten.
- Beispielsweise ändert eine zusätzliche Programmzeile `myalarm.minute = 15;` den entsprechenden Eintrag.

Zugriff auf einen Datensatz per Zeiger (weckzeit2.c)

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  struct TUhrzeit *p = &myalarm; //Zeiger auf den Datensatz myalarm
                                 //p enthaelt Anfangsadresse des Datensatzes

  //Zugriff auf den Datensatz mittels Zeiger p
  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         (*p).stunde, (*p).minute, (*p).sekunde, (*p).zeitzone);
  return 0;
}
```

Zugriff auf einen Datensatz per Zeiger (weckzeit2.c)

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  struct TUhrzeit *p = &myalarm; //Zeiger auf den Datensatz myalarm
                                 //p enthaelt Anfangsadresse des Datensatzes

  //Zugriff auf den Datensatz mittels Zeiger p
  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         (*p).stunde, (*p).minute, (*p).sekunde, (*p).zeitzone);
  return 0;
}
```

p: Anfangsadresse des Datensatzes. Zugriff auf Komponenten wie **(*p).stunde** schreibaufwendig, denn runde Klammer darf nicht weggelassen werden, da ***** niedriger priorisiert als Punktoperator **.**

Zugriff auf einen Datensatz per Zeiger (weckzeit3.c)

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  struct TUhrzeit *p = &myalarm; //Zeiger auf den Datensatz myalarm
                                 //p enthaelt Anfangsadresse des Datensatzes

  //Zugriff auf den Datensatz mittels Zeiger p
  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         p->stunde, p->minute, p->sekunde, p->zeitzone);
  return 0;
}
```

Es gibt eine Kurzschreibweise für den Zugriff auf einen Datensatz per Zeiger. Statt `(*p).stunde` schreibt man `p->stunde` usw.

Speicherplatzbedarf eines Datensatzes: `sizeof`

(weckzeit4.c)

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  int c = sizeof(struct TUhrzeit);

  printf("Speicherplatzbedarf eines Datensatzes in Byte: %d\n", c);
  return 0;
}
```

`sizeof` ist ein Schlüsselwort und dient dazu, den Speicherplatzbedarf von Datenobjekten eines *Typs* in Byte zu bestimmen, z.B. **`sizeof(struct TUhrzeit)`** liefert 20 *Bytes*.

Speicher für Datensatz reservieren: `malloc` (weckzeit5.c)

```c
#include <stdio.h>
#include <stdlib.h> //fuer malloc
#include <string.h>

struct TUhrzeit
{
  int stunde;       //0...23
  int minute;       //0...59
  int sekunde;      //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};

int main(void)
{
  struct TUhrzeit *p = NULL;  //Zeiger anlegen

  p = malloc(sizeof(struct TUhrzeit)); //Speicher ausfassen;
                                       //Anfangsadresse in p bereitstellen
  if(p == NULL) {printf("Nicht genug Speicher\n"); return 1;}
  p->stunde = 7;  //Datensatz ueber Zeiger p erreichen und fuellen
  p->minute = 30;
  p->sekunde = 0;
  strcpy(p->zeitzone, "MEZ");
  //Datensatzinhalt ueber Zeiger p ausgeben
  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n", p->stunde, p->minute, p->sekunde, p->zeitzone);
  return 0;
}
```

`malloc` (*„memory allocation“*), verfügbar über **`stdlib.h`**, reserviert zusammenhängenden Speicherbereich. Anzahl Bytes als Parameter übergeben. Anfangsadresse des ausgefassten Speicherbereichs zurückgegeben. **`malloc`** zum *Anlegen eines neuen Datensatzes*.

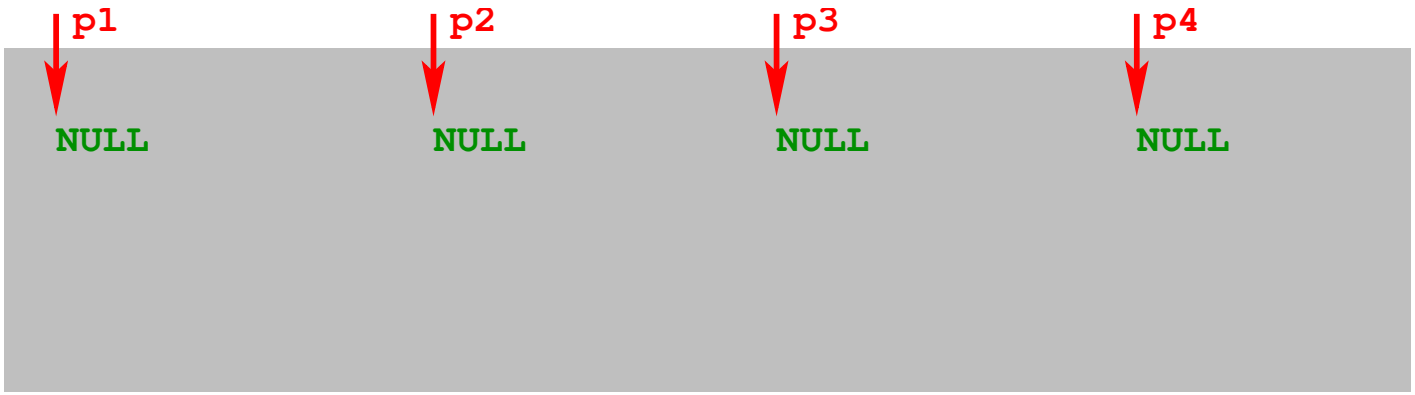
Datensatz löschen und Speicher freigeben: `free`

(weckzeit6.c)

```c
int main(void)
{
  struct TUhrzeit *p = NULL;  //Zeiger anlegen

  p = malloc(sizeof(struct TUhrzeit)); //Speicher ausfassen;
                                       //Anfangsadresse in p bereitstellen
  if(p == NULL) {printf("Nicht genug Speicher\n"); return 1;}
  p->stunde = 7;  //Datensatz ueber Zeiger p erreichen und fuellen
  p->minute = 30;
  p->sekunde = 0;
  strcpy(p->zeitzone, "MEZ");

  /* Datensatz ab Adresse p loeschen und Speicherbereich wieder freigeben */

  free(p);

  return 0;
}
```

`free(p)` löscht den Datensatz ab Adresse `p` und *gibt den Speicherplatz wieder frei*. Da `p` auf ein Datenobjekt bekannten Typs zeigt (hier: `struct TUhrzeit`), steht fest, wieviele Bytes freigegeben werden.

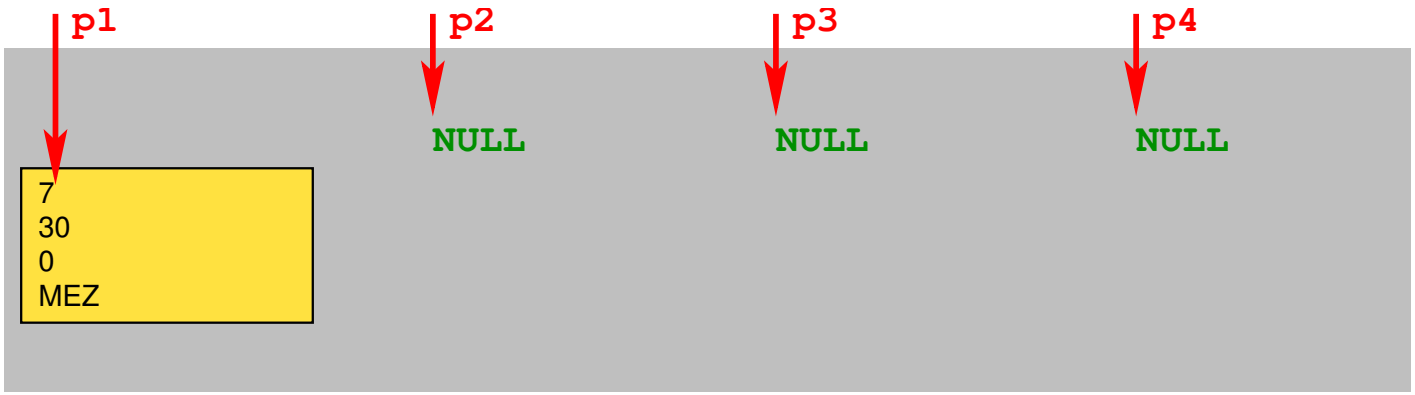
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

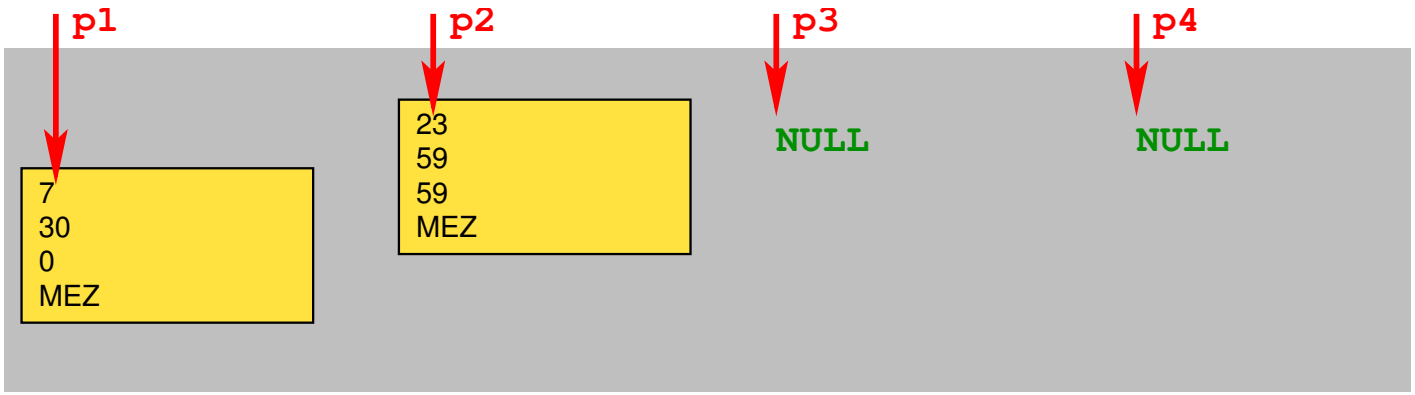
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

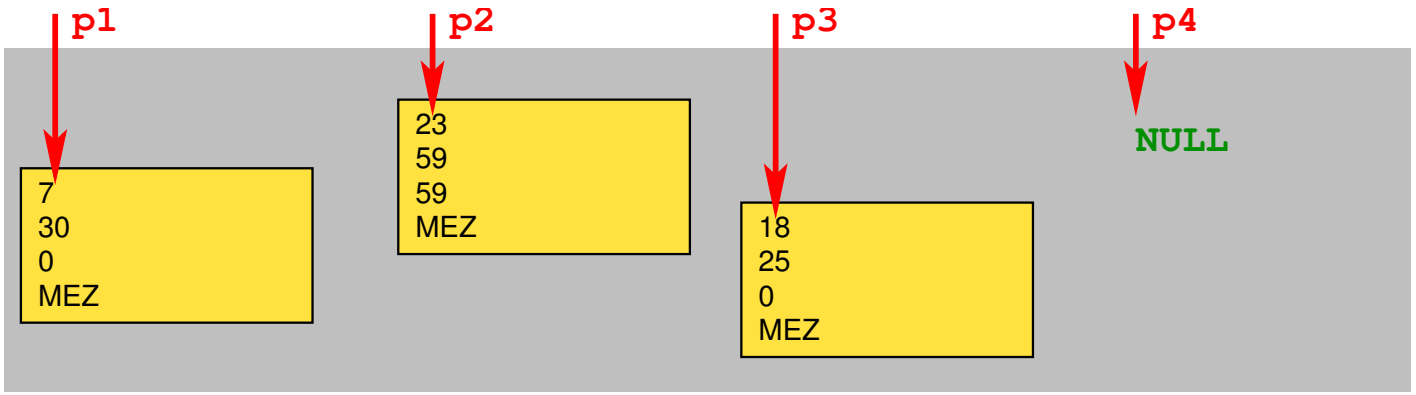
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

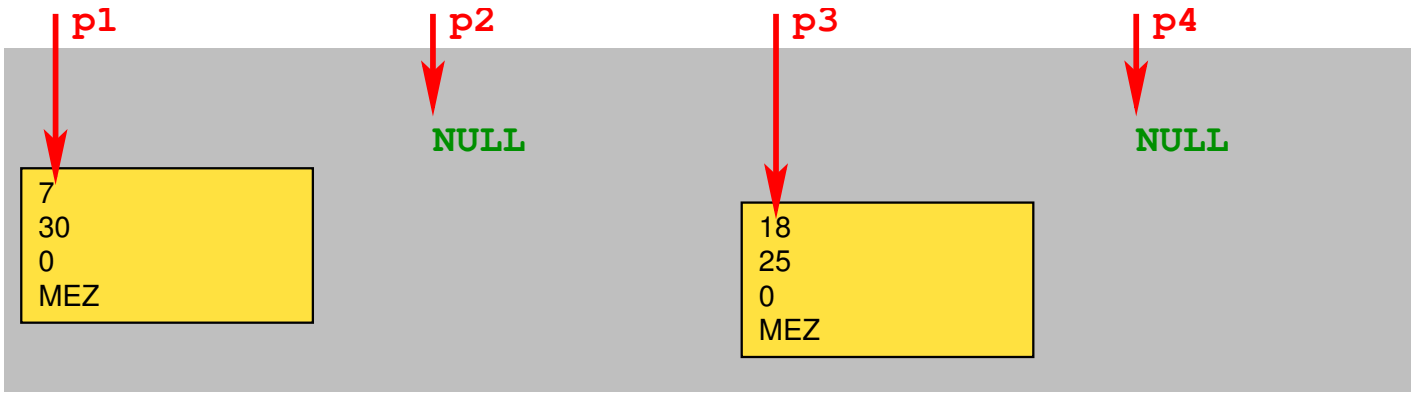
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. `p1`, `p2`, `p3` und `p4`, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (`malloc`) und individuell freigeben (`free`), sobald sie nicht mehr gebraucht werden.

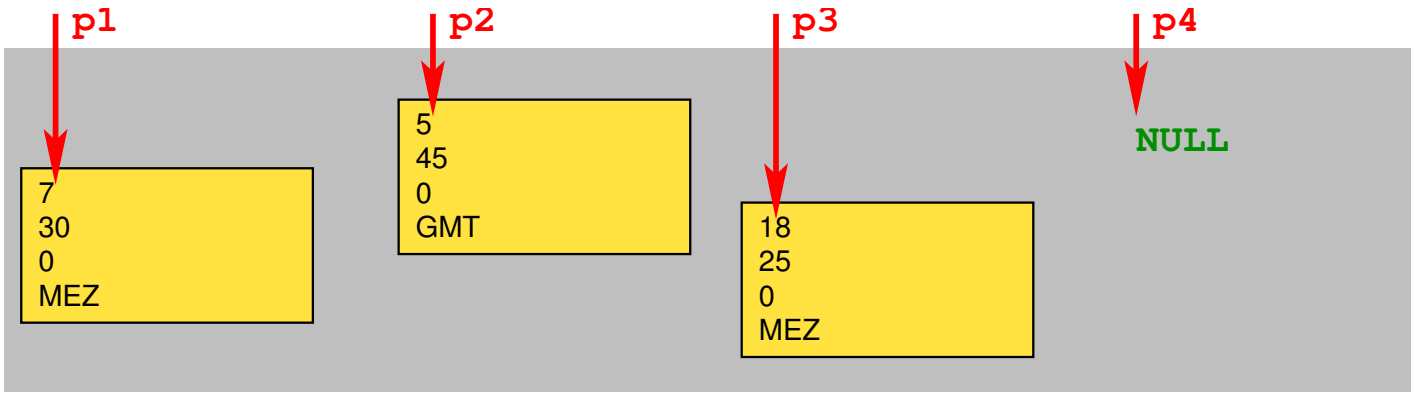
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

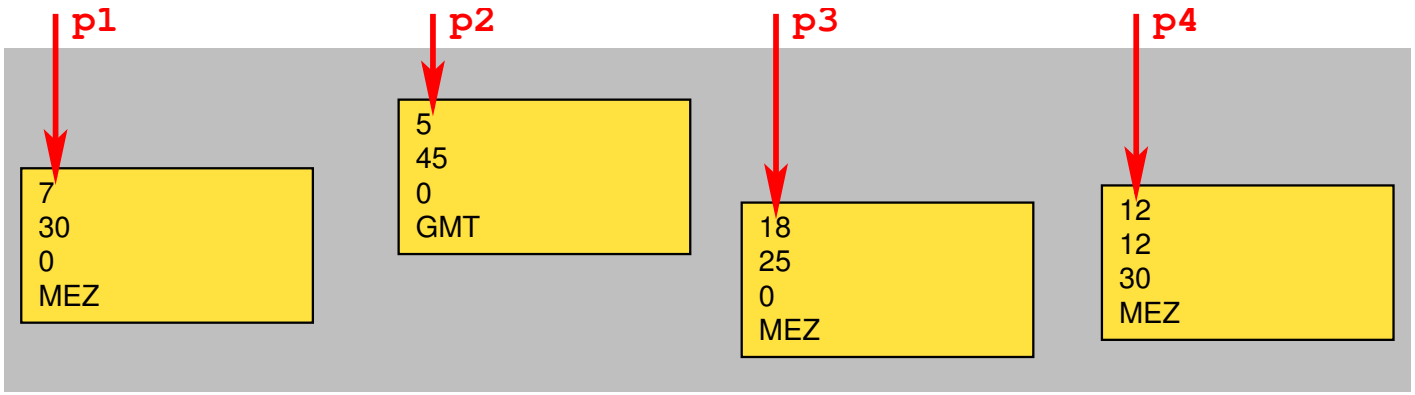
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

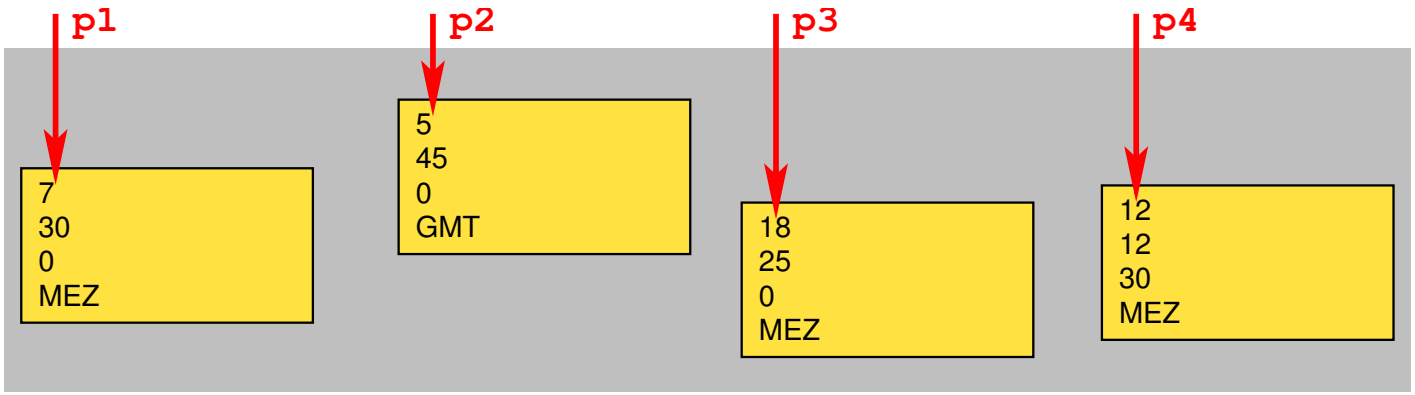
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze vom Typ `struct TUhrzeit`

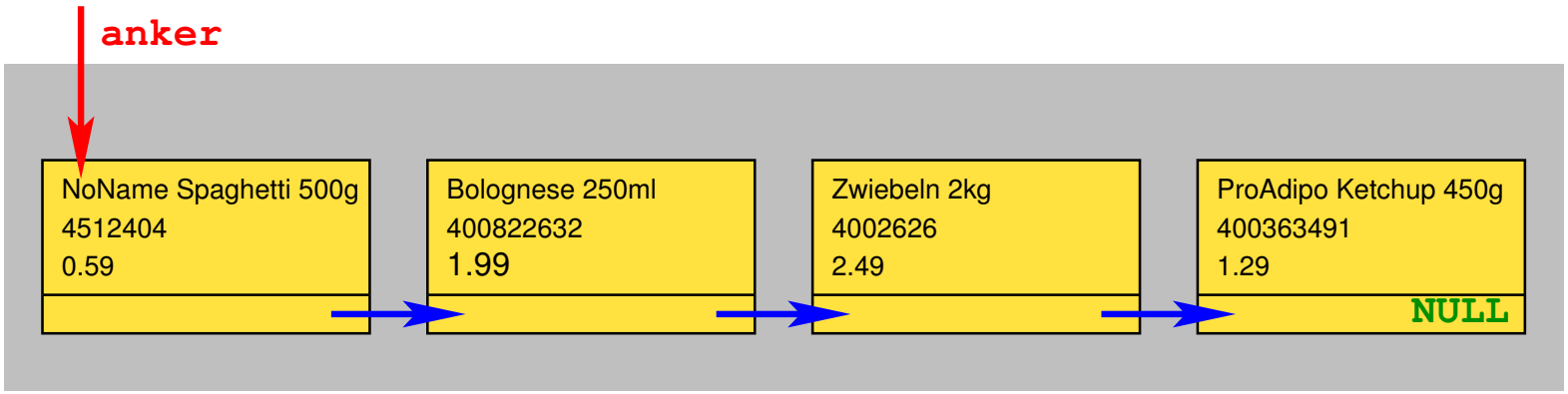
Mit mehreren Zeigern, z.B. **`p1`**, **`p2`**, **`p3`** und **`p4`**, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (**`malloc`**) und individuell freigeben (**`free`**), sobald sie nicht mehr gebraucht werden.

Was uns zur Implementierung einer *Linearen Liste* (nur) noch fehlt, ist die *Verkettung* der einzelnen Datensätze untereinander (Verweise auf *Nachfolger* bzw. *Endekennung*).

Beispiel: Warenkorb als Liste von Artikeln

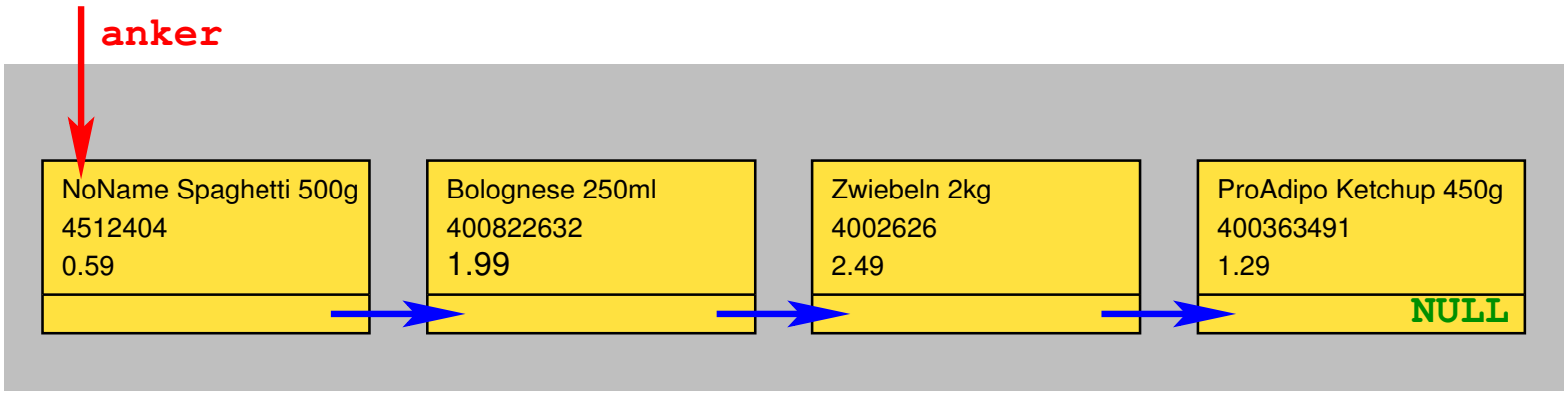
Jeder Artikel ist ein Listenelement. Artikel können hinzugefügt, gelöscht, angezeigt und zur Gesamtpreisberechnung ausgewertet werden.

Einfach verkettete Lineare Liste



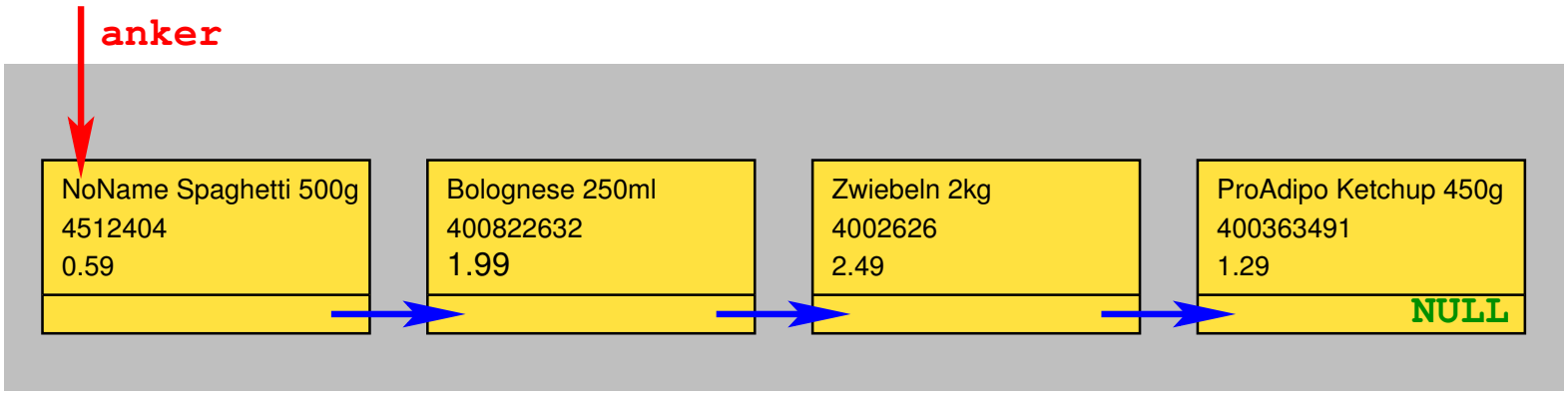
- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).

Einfach verkettete Lineare Liste



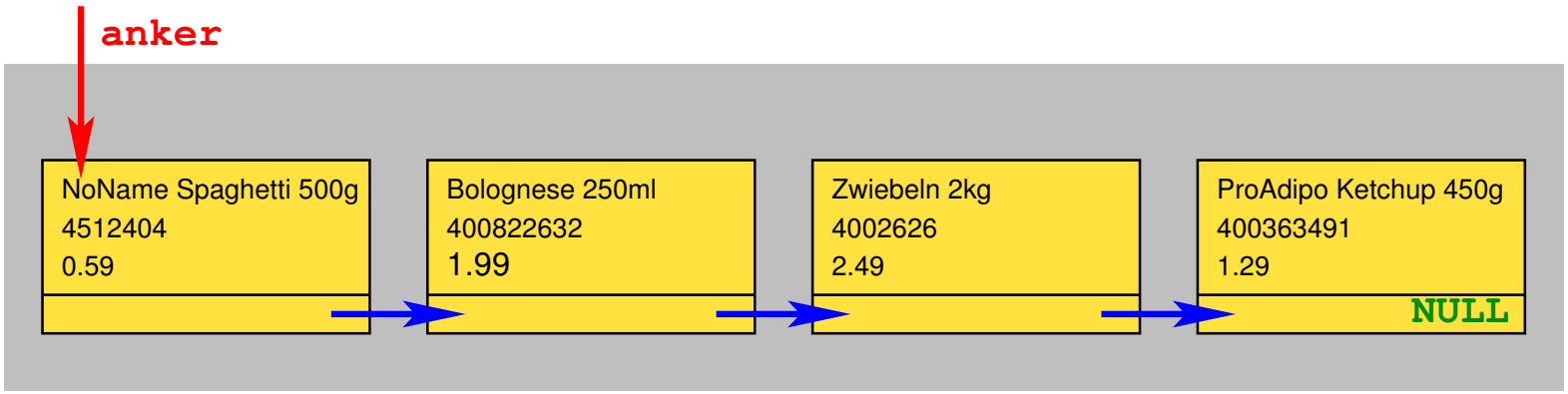
- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).
- Zusätzlich enthält jedes Listenelement eine Komponente, die *auf das nachfolgende Listenelement verweist* (Anfangsadresse im Speicher) bzw. beim letzten Listenelement eine *Endekennung* (**`NULL`**) trägt

Einfach verkettete Lineare Liste



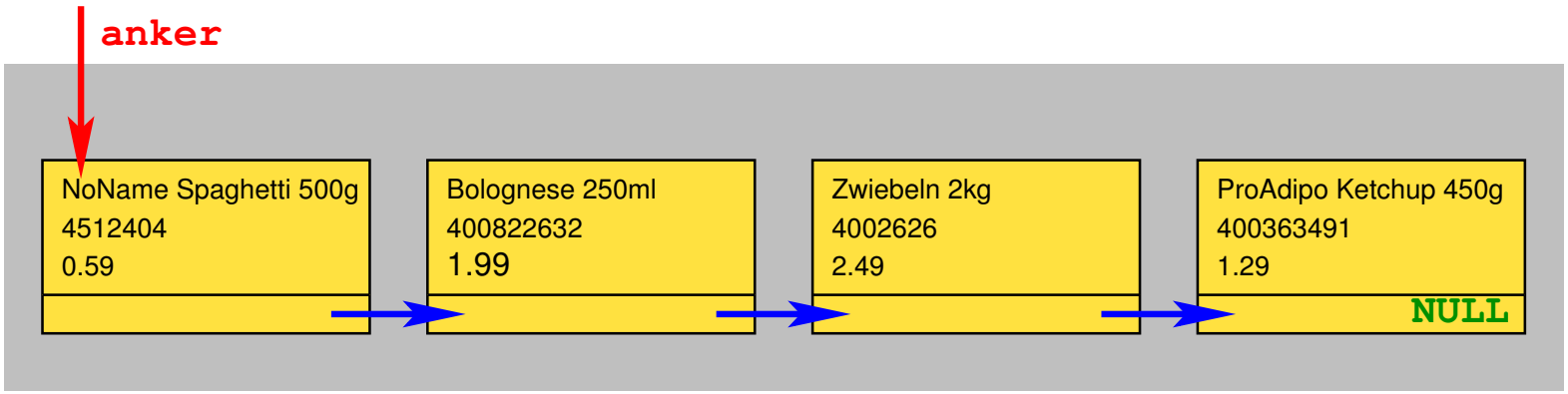
- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).
- Zusätzlich enthält jedes Listenelement eine Komponente, die *auf das nachfolgende Listenelement verweist* (Anfangsadresse im Speicher) bzw. beim letzten Listenelement eine *Endekennung* (**`NULL`**) trägt
- Liste ist über den (globalen) *Listenanker* zugänglich. Er verweist auf das erste Listenelement (Adresse des ersten Listenelements)

Einfach verkettete Lineare Liste



- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).
- Zusätzlich enthält jedes Listenelement eine Komponente, die *auf das nachfolgende Listenelement verweist* (Anfangsadresse im Speicher) bzw. beim letzten Listenelement eine *Endekennung* (`NULL`) trägt
- Liste ist über den (globalen) *Listenanker* zugänglich. Er verweist auf das erste Listenelement (Adresse des ersten Listenelements)
- Vom Listenanker aus kann die Liste zum Ende hin elementweise durchlaufen werden

Einfach verkettete Lineare Liste



- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).
- Zusätzlich enthält jedes Listenelement eine Komponente, die *auf das nachfolgende Listenelement verweist* (Anfangsadresse im Speicher) bzw. beim letzten Listenelement eine *Endekennung* (`NULL`) trägt
- Liste ist über den (globalen) *Listenanker* zugänglich. Er verweist auf das erste Listenelement (Adresse des ersten Listenelements)
- Vom Listenanker aus kann die Liste zum Ende hin elementweise durchlaufen werden
- Weitere Verkettungen zwischen den Listenelementen (z.B. zum Vorgänger) gibt es nicht, deshalb *einfach verkettete Lineare Liste*

Globale Datenstruktur für die Liste

Typbeschreibung für Listenelemente und globaler Listenanker

```
struct TWarenkorb
{
  char produktname[256];        //Name eines Artikels
  unsigned long produktcode;    //z.B. EAN-Strichcode
  float preis;                  //Preis in Euro

  struct TWarenkorb *next;      //Verweis auf Nachfolger
};
```

```
struct TWarenkorb *anker = NULL;  //globaler Listenanker
```

In die Typdefinition `struct TWarenkorb` wird als Komponente ein Zeiger `struct TWarenkorb *next;` aufgenommen, der die Anfangsadresse des nachfolgenden Listenelementes oder die Endekennung `NULL` enthält.

Der Listenanker `anker` wird als *globale Variable* vor den Listenfunktionen definiert.

Erstes Element in leere Liste einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

Liste ist leer, Listenanker `anker` hat Wert `NULL`

Erstes Element in leere Liste einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

Speicherplatz für ein neues Listenelement mittels **`malloc`** ausgefasst, Zeiger **`h`** enthält Anfangsadresse

Erstes Element in leere Liste einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

Im **anker** hinterlegte Adresse (Endekennung **NULL**) wird in das neue Listenelement eingetragen

Erstes Element in leere Liste einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

Produktcode (vorzeichenlose Ganzzahl) wird in den neuen Datensatz eingetragen, Zugriff über Zeiger `h`

Erstes Element in leere Liste einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

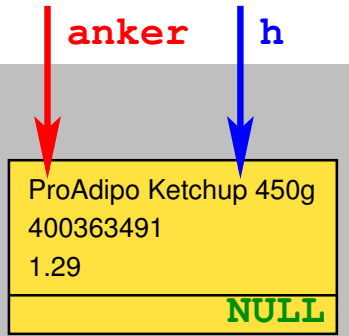
Produktname (Zeichenkette) wird in den neuen Datensatz eingetragen, Zugriff über Zeiger `h`

Erstes Element in leere Liste einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

Preis (Gleitkommazahl) wird in den neuen Datensatz eingetragen, Zugriff über Zeiger `h`

Erstes Element in leere Liste einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400363491;
strcpy(h->produktname, "ProAdipo Ketchup 450g");
h->preis = 1.29;
anker = h;
```

Listenanker **anker** wird auf Anfangsadresse des neuen Datensatzes, also auf **h**, gesetzt und zeigt jetzt auf das erste Listenelement. Einfügen damit abgeschlossen.

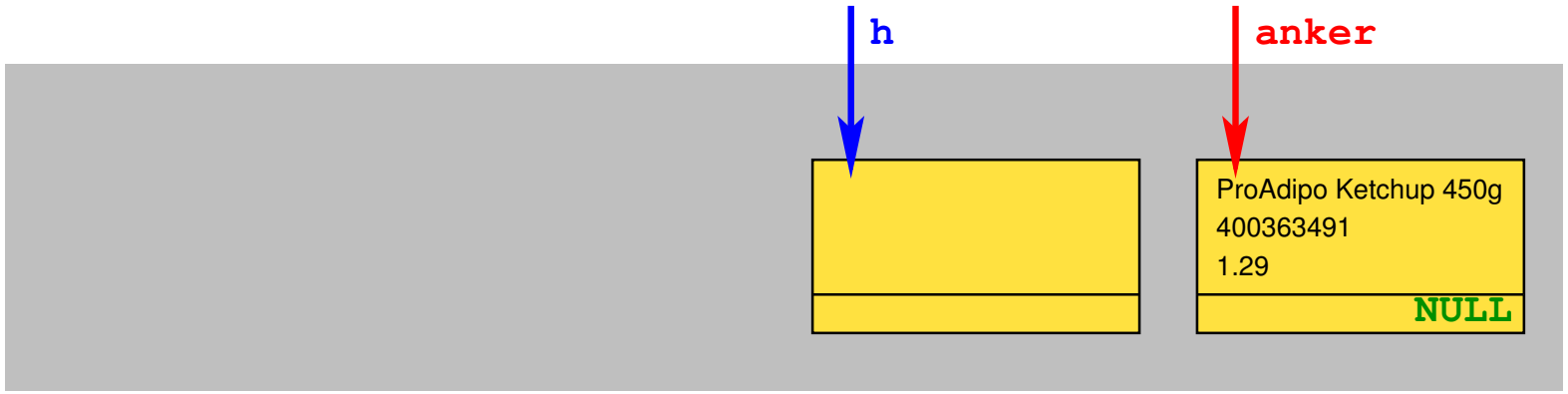
Neues Element am Listenanfang einfügen

Neues Element am Listenanfang einfügen


```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4002626;
strcpy(h->produktname, "Zwiebeln 2kg");
h->preis = 2.49;
anker = h;
```

Gleiche Anweisungsfolge wie beim Einfügen des ersten Elements

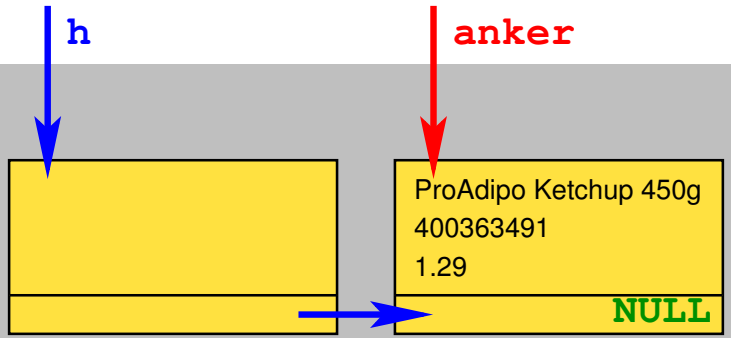
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4002626;
strcpy(h->produktname, "Zwiebeln 2kg");
h->preis = 2.49;
anker = h;
```

Speicherplatz für ein neues Listenelement mittels **`malloc`** ausgefasst, Zeiger **`h`** enthält Anfangsadresse

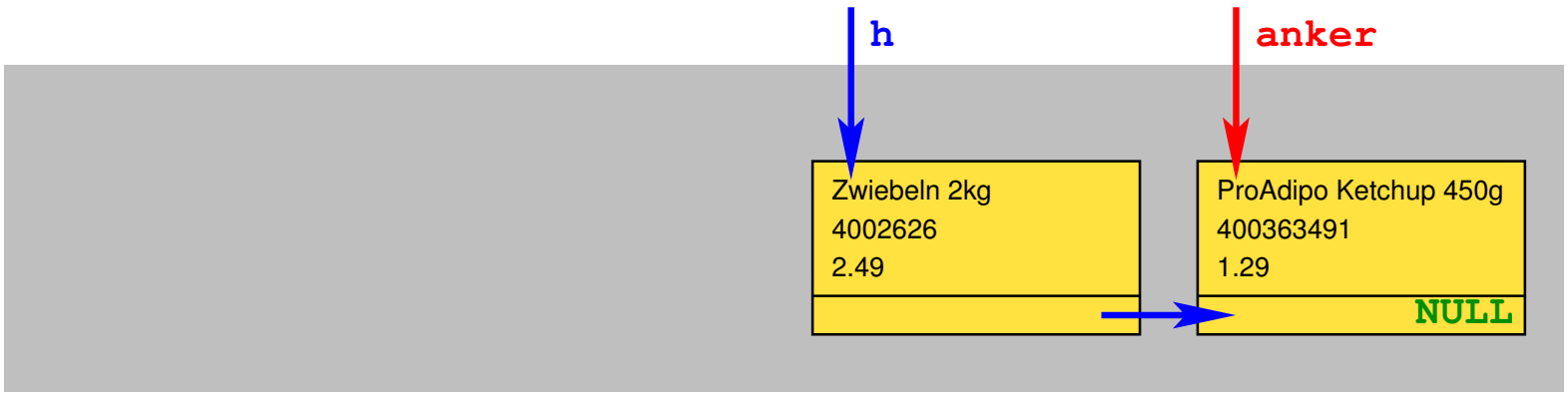
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4002626;
strcpy(h->produktname, "Zwiebeln 2kg");
h->preis = 2.49;
anker = h;
```

Im **anker** hinterlegte Adresse des nunmehr nachfolgenden Listenelements wird eingetragen

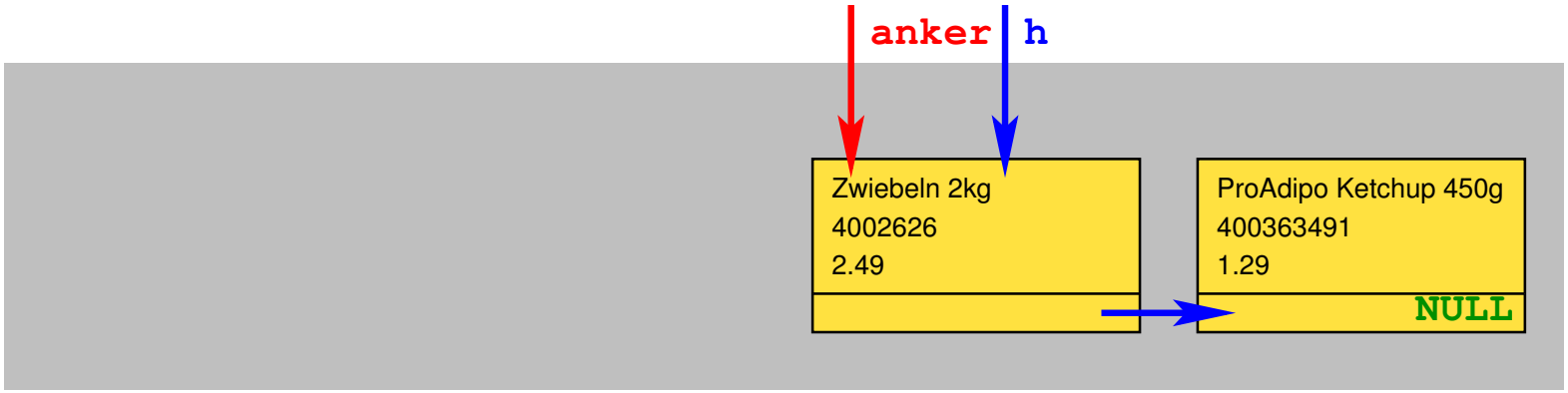
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4002626;
strcpy(h->produktname, "Zwiebeln 2kg");
h->preis = 2.49;
anker = h;
```

Informationsteil des Datensatzes wird befüllt

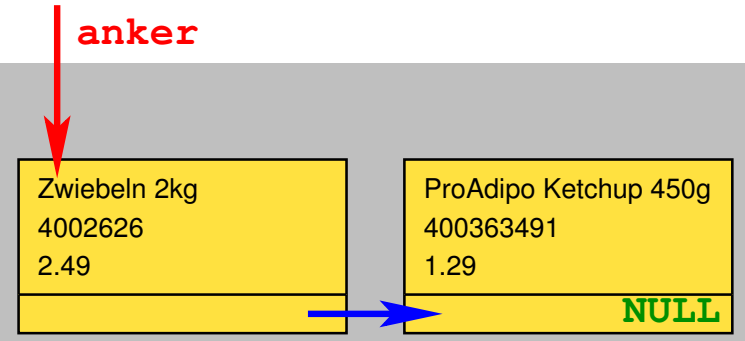
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4002626;
strcpy(h->produktname, "Zwiebeln 2kg");
h->preis = 2.49;
anker = h;
```

Listenanker **anker** wird auf Anfangsadresse des neuen Datensatzes, also auf **h**, gesetzt und zeigt jetzt auf das erste Listenelement. Fertig.

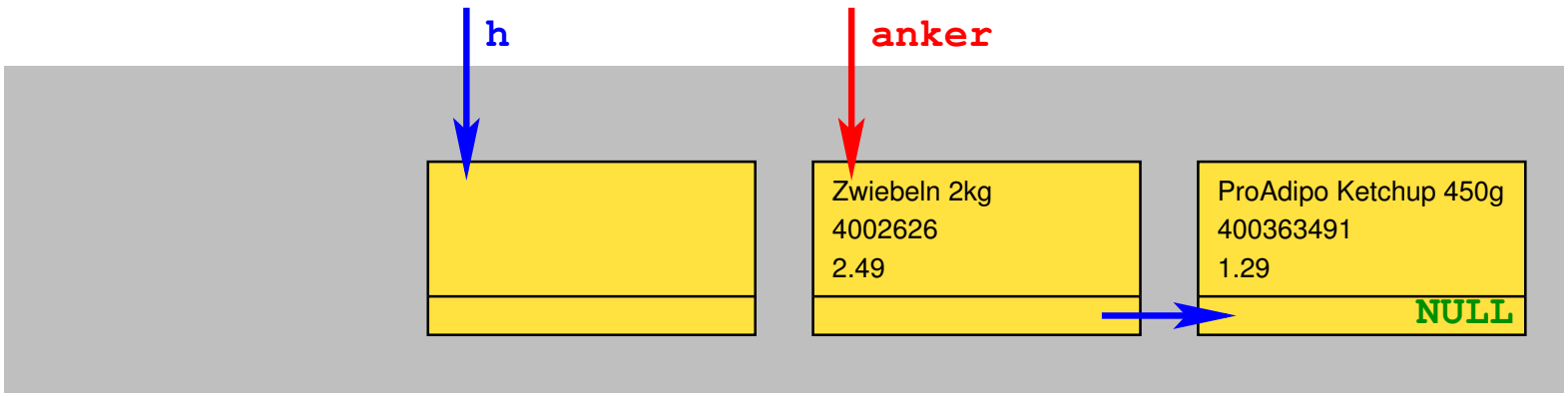
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400822632;
strcpy(h->produktname, "Bolognese 250ml");
h->preis = 1.99;
anker = h;
```

Gleiche Anweisungsfolge wie beim Einfügen des ersten Elements

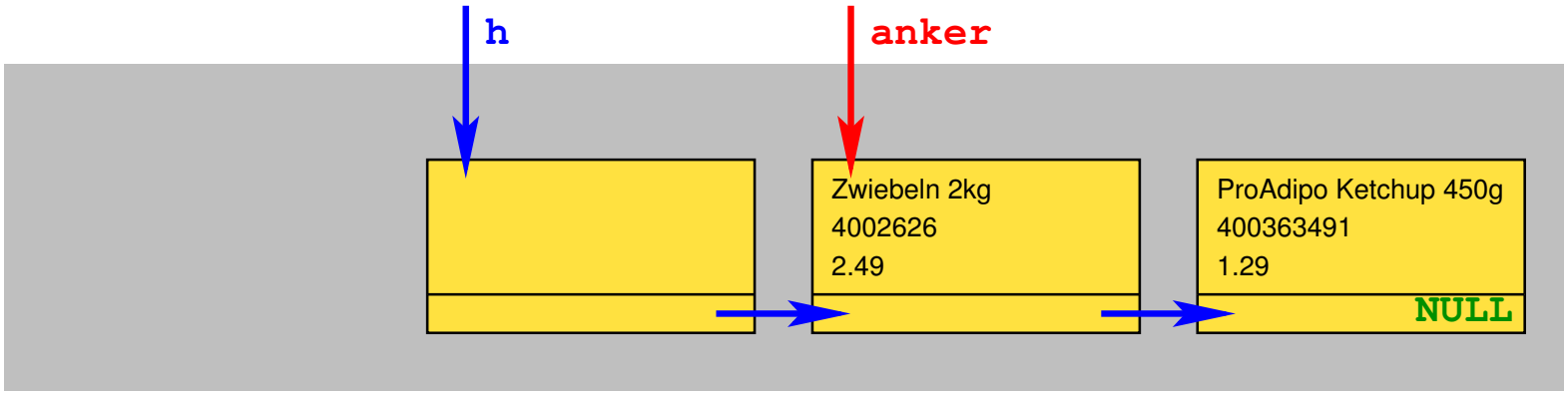
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400822632;
strcpy(h->produktname, "Bolognese 250ml");
h->preis = 1.99;
anker = h;
```

Speicherplatz für ein neues Listenelement mittels **malloc** ausgefasst, Zeiger **h** enthält Anfangsadresse

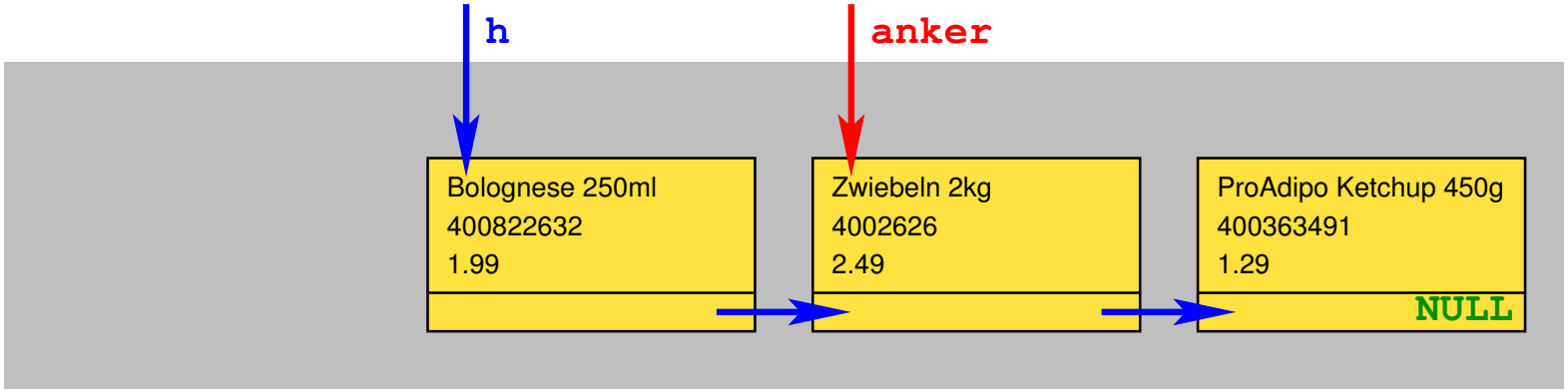
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400822632;
strcpy(h->produktname, "Bolognese 250ml");
h->preis = 1.99;
anker = h;
```

Im **anker** hinterlegte Adresse des nunmehr nachfolgenden Listenelements wird eingetragen

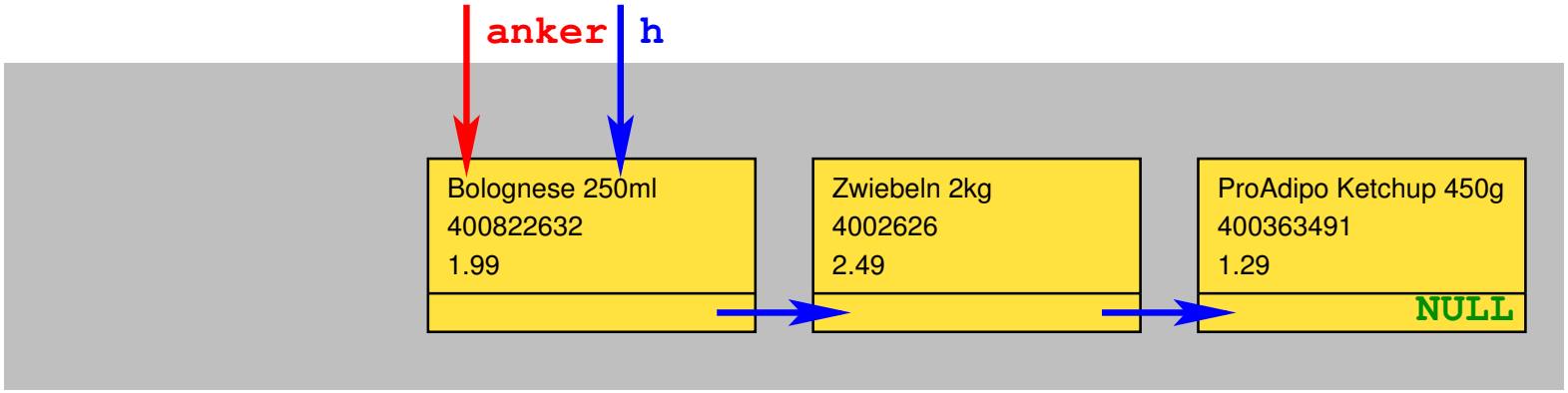
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400822632;
strcpy(h->produktname, "Bolognese 250ml");
h->preis = 1.99;
anker = h;
```

Informationsteil des Datensatzes wird befüllt

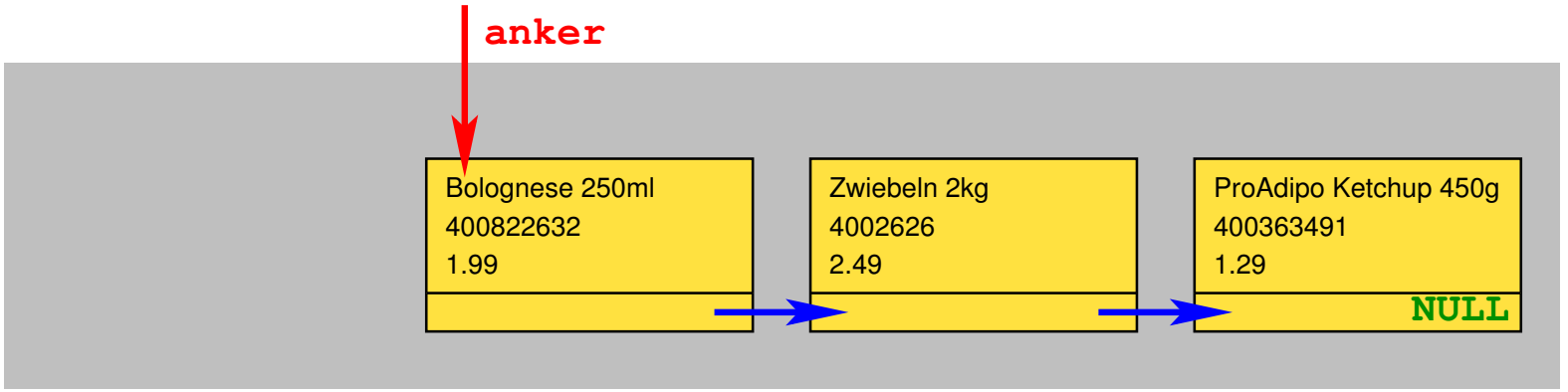
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 400822632;
strcpy(h->produktname, "Bolognese 250ml");
h->preis = 1.99;
anker = h;
```

Listenanker **anker** wird auf Anfangsadresse des neuen Datensatzes, also auf **h**, gesetzt und zeigt jetzt auf das erste Listenelement. Fertig.

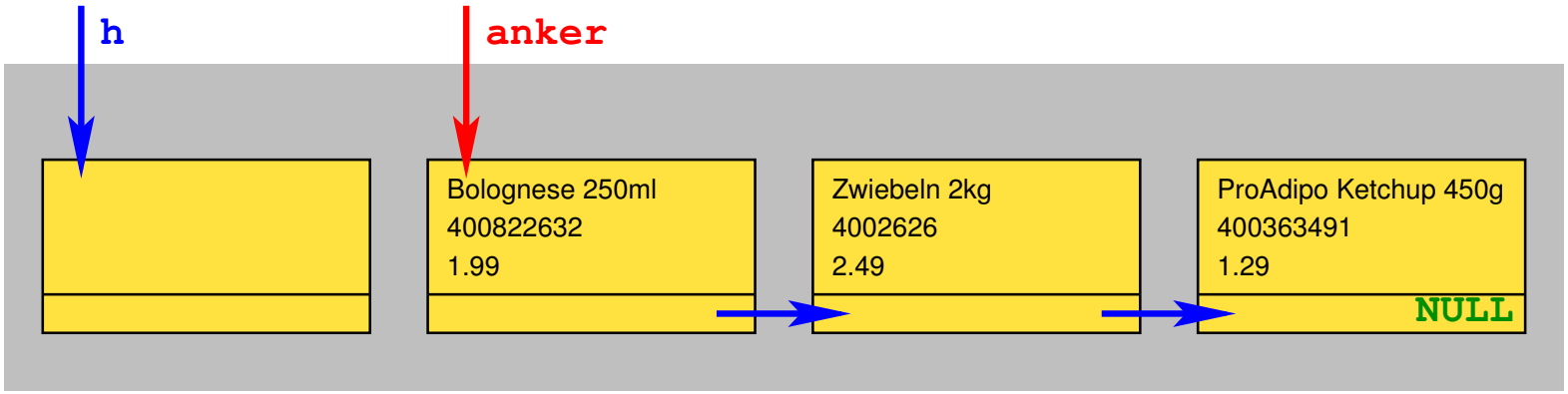
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4512404;
strcpy(h->produktname, "NoName Spaghetti 500g");
h->preis = 0.59;
anker = h;
```

Gleiche Anweisungsfolge wie beim Einfügen des ersten Elements

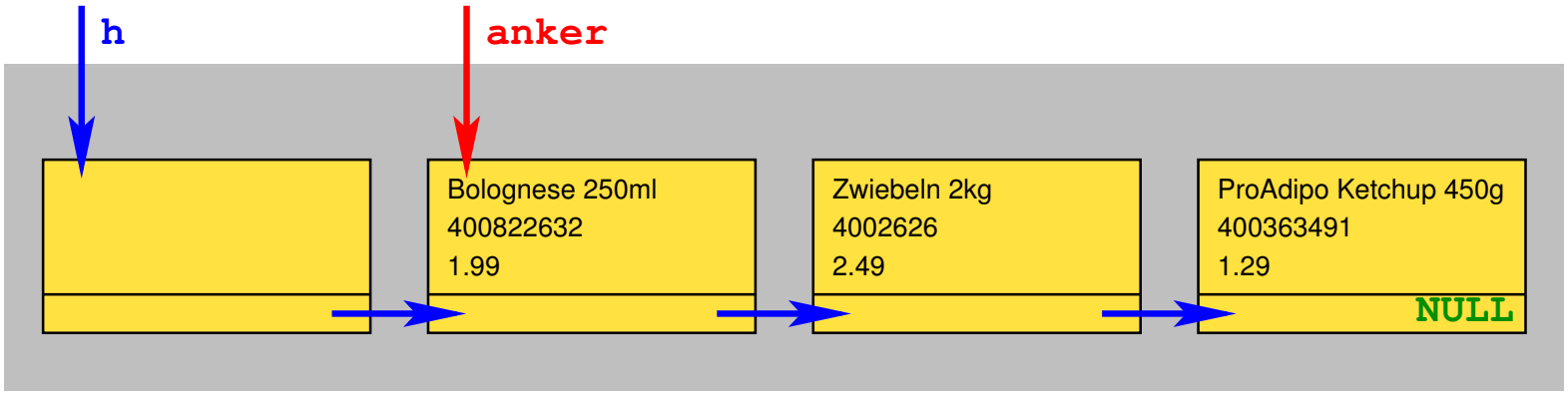
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4512404;
strcpy(h->produktname, "NoName Spaghetti 500g");
h->preis = 0.59;
anker = h;
```

Speicherplatz für ein neues Listenelement mittels **`malloc`** ausgefasst, Zeiger **`h`** enthält Anfangsadresse

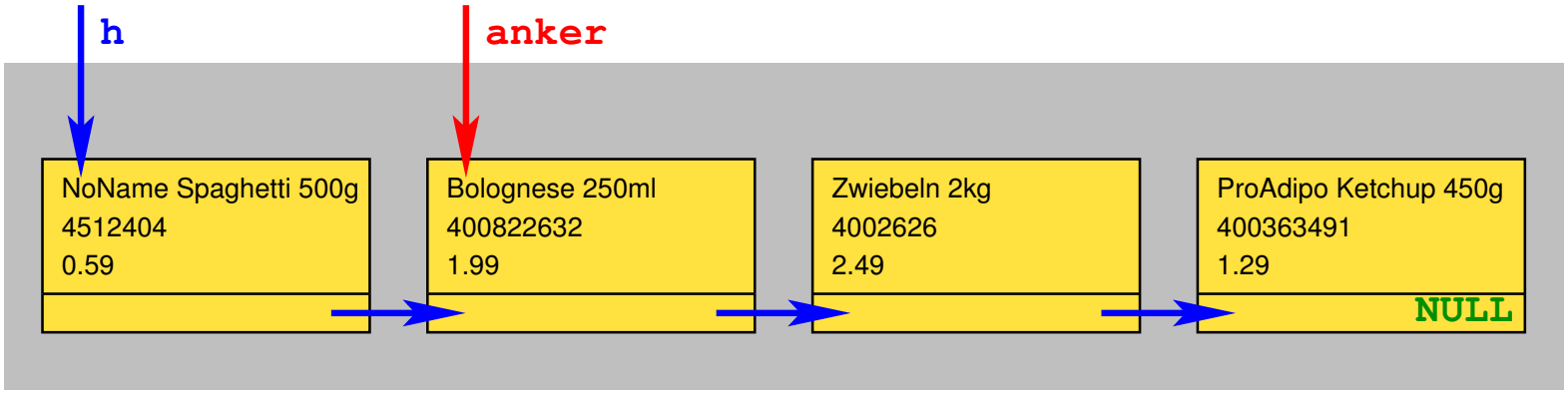
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4512404;
strcpy(h->produktname, "NoName Spaghetti 500g");
h->preis = 0.59;
anker = h;
```

Im **anker** hinterlegte Adresse des nunmehr nachfolgenden Listenelements wird eingetragen

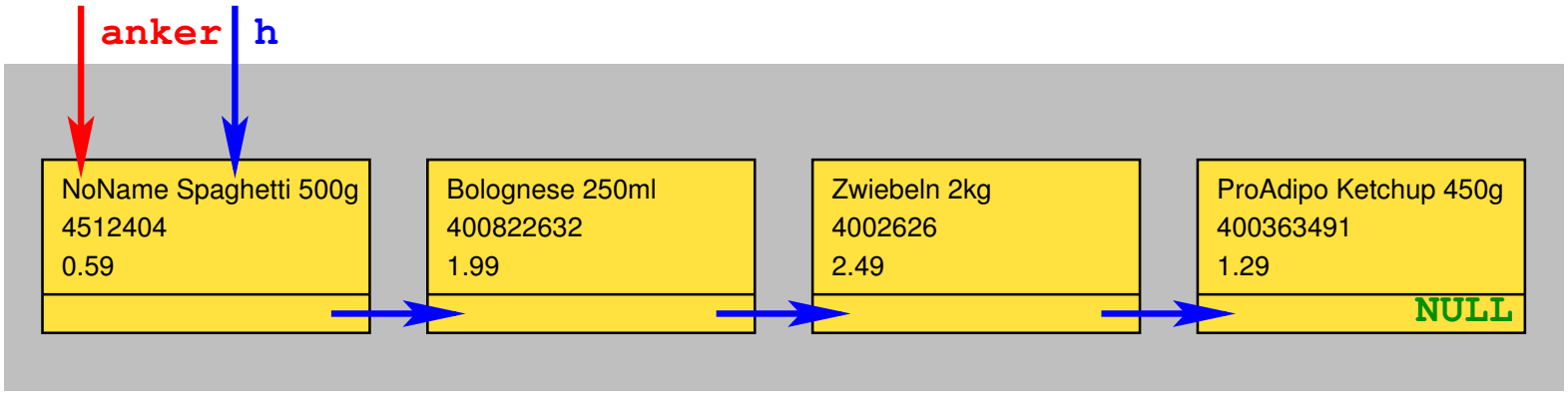
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4512404;
strcpy(h->produktname, "NoName Spaghetti 500g");
h->preis = 0.59;
anker = h;
```

Informationsteil des Datensatzes wird befüllt

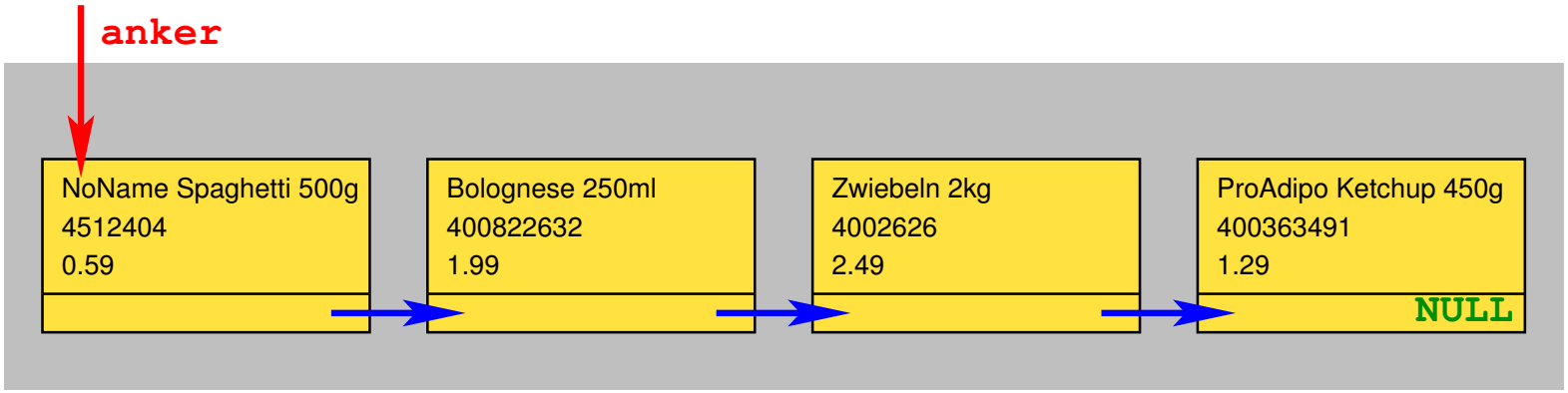
Neues Element am Listenanfang einfügen




```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4512404;
strcpy(h->produktname, "NoName Spaghetti 500g");
h->preis = 0.59;
anker = h;
```

Listenanker **anker** wird auf Anfangsadresse des neuen Datensatzes, also auf **h**, gesetzt und zeigt jetzt auf das erste Listenelement.

Neues Element am Listenanfang einfügen

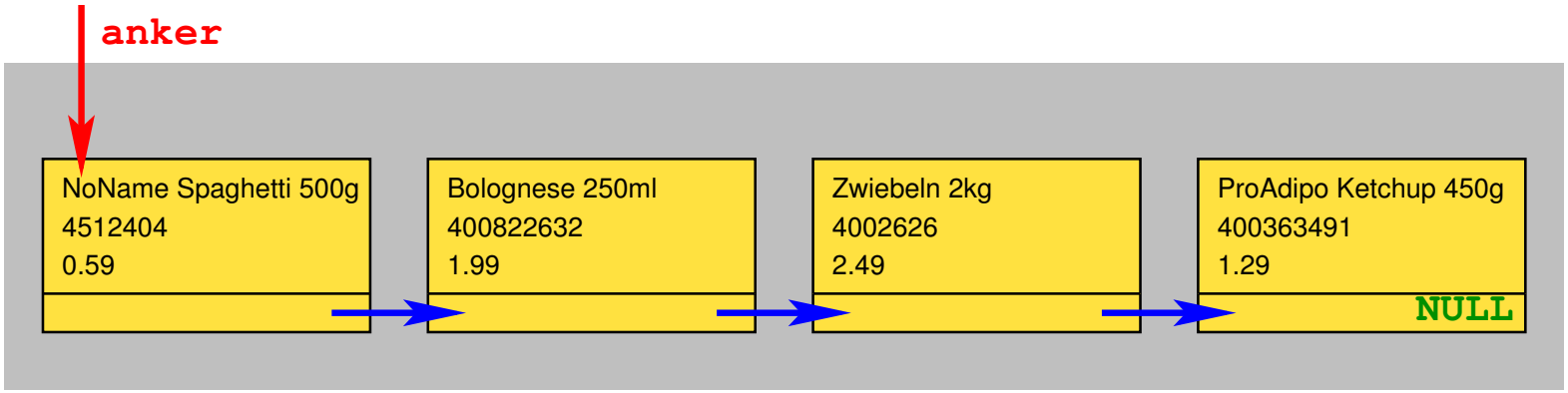



```
h = malloc(sizeof(struct TWarenkorb));
...
h->next = anker;
h->produktcode = 4512404;
strcpy(h->produktname, "NoName Spaghetti 500g");
h->preis = 0.59;
anker = h;
```

Einfügen von vier Elementen jeweils am Listenanfang abgeschlossen.

Anzahl Elemente in der Liste bestimmen

Anzahl Elemente in der Liste bestimmen

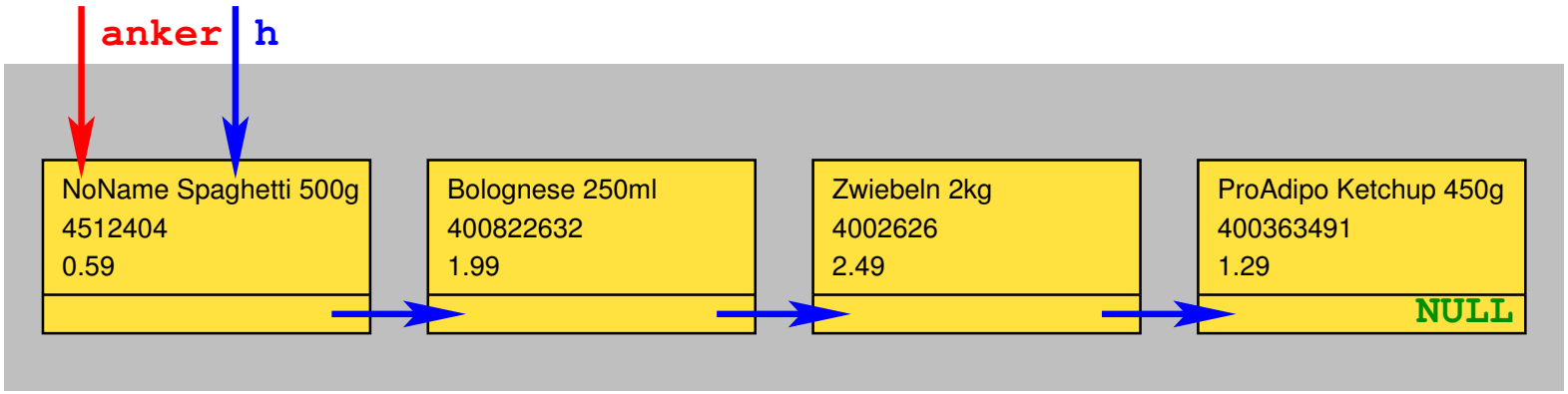



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 0

Vollständiges Ablaufen der Liste vom **Anker** aus. Zählvariable **i** mit jedem Element um eins hochzählen

Anzahl Elemente in der Liste bestimmen

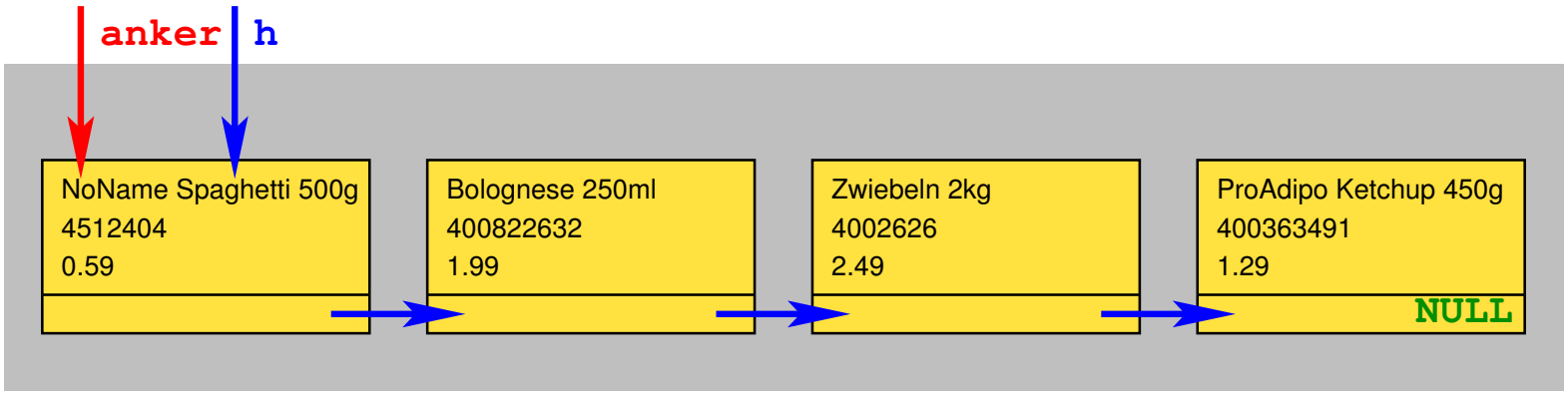



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 0

Hilfszeiger `h` am Listenanfang platzieren

Anzahl Elemente in der Liste bestimmen

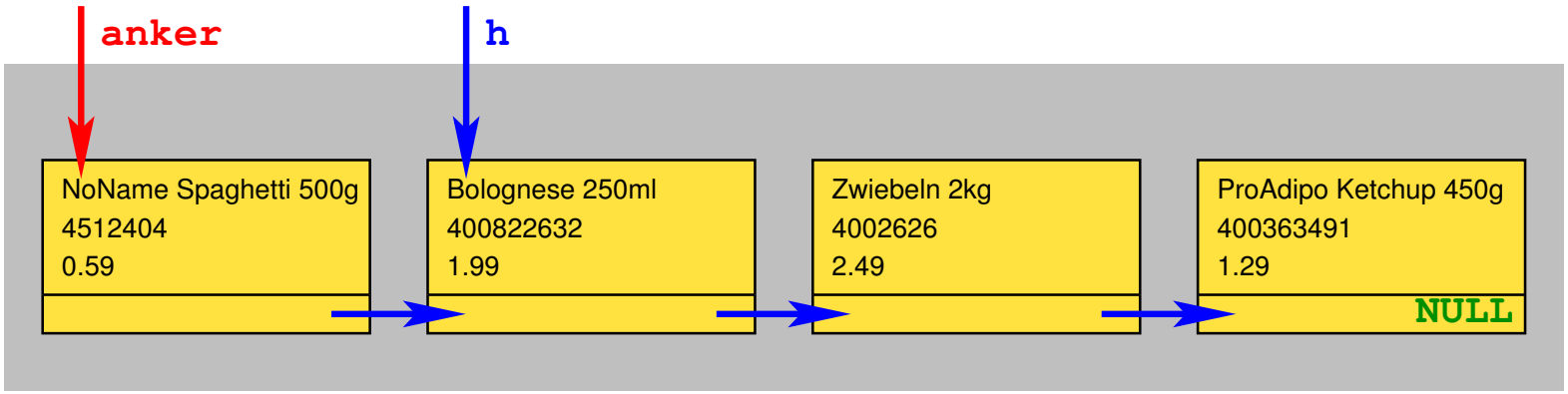



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 1

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

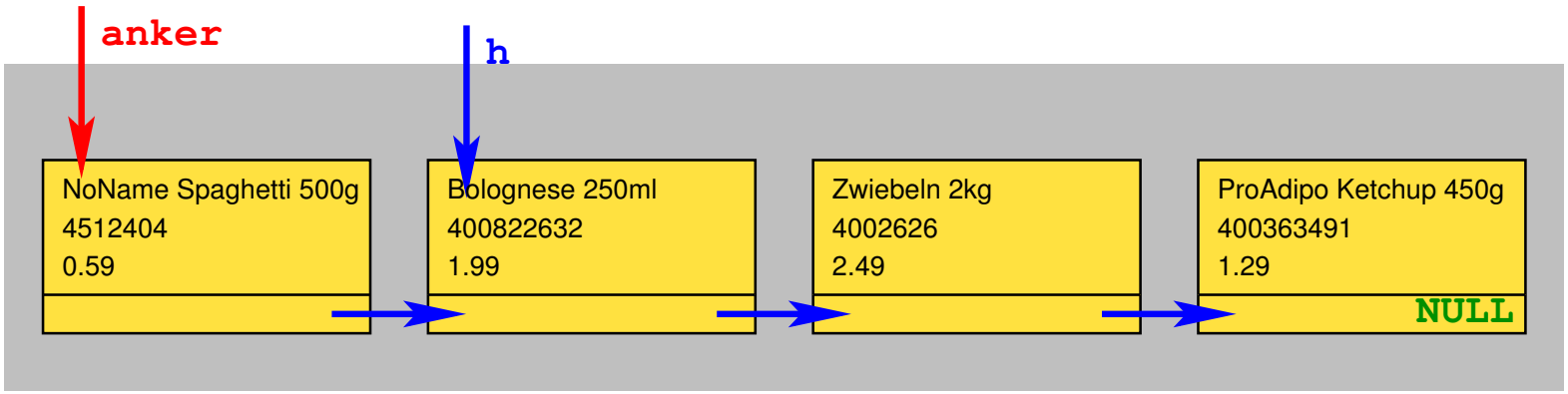



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 1

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen

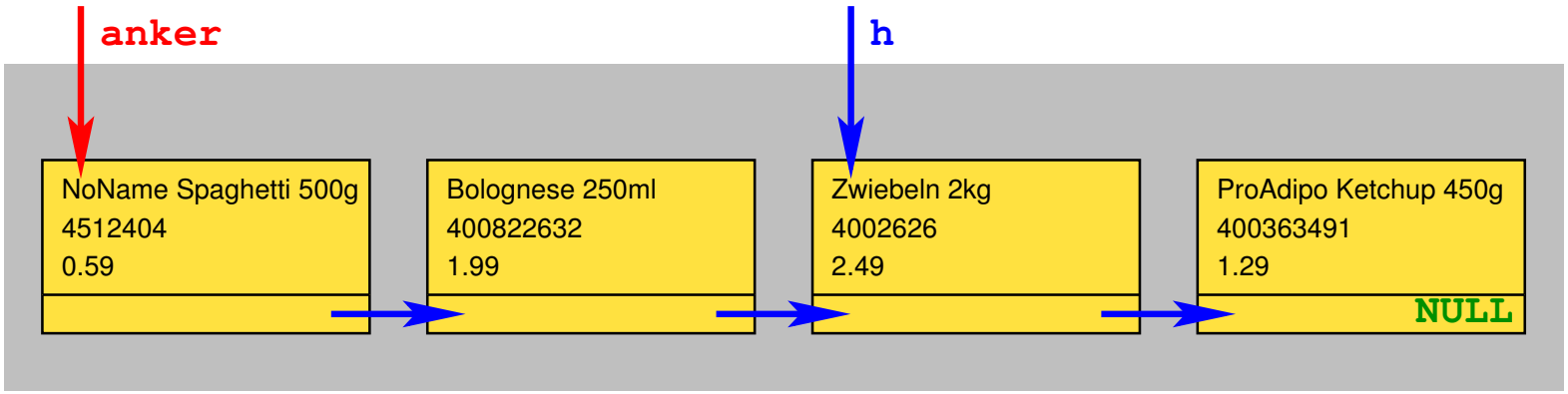



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 2

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

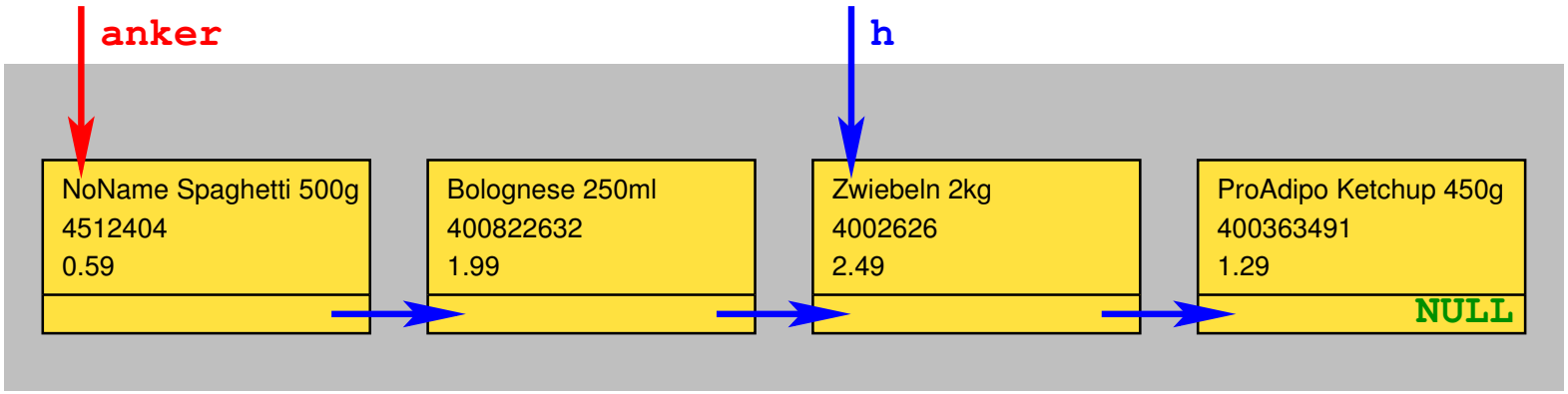



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 2

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen

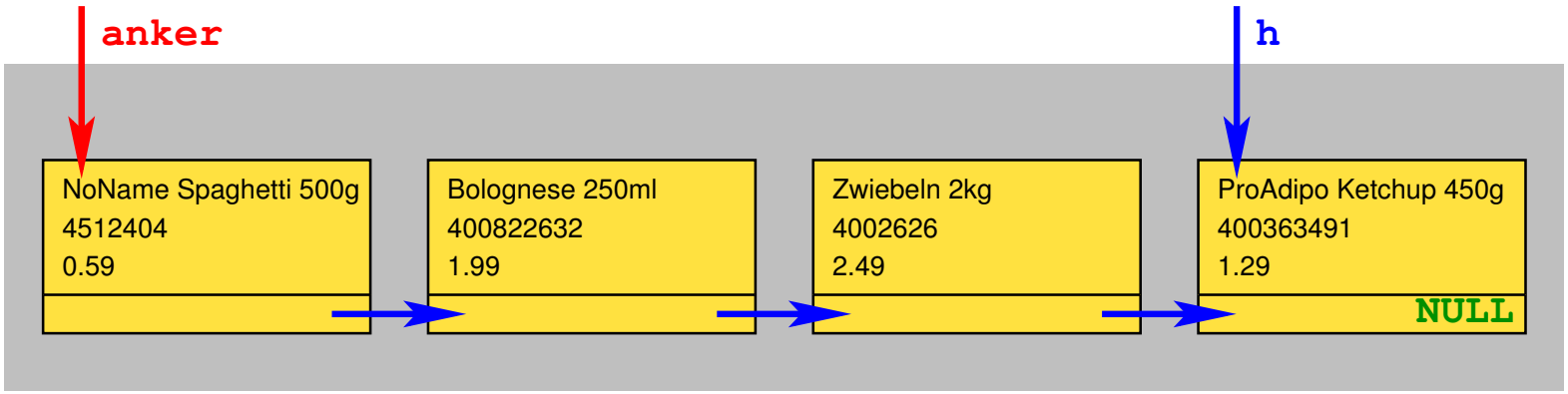



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 3

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

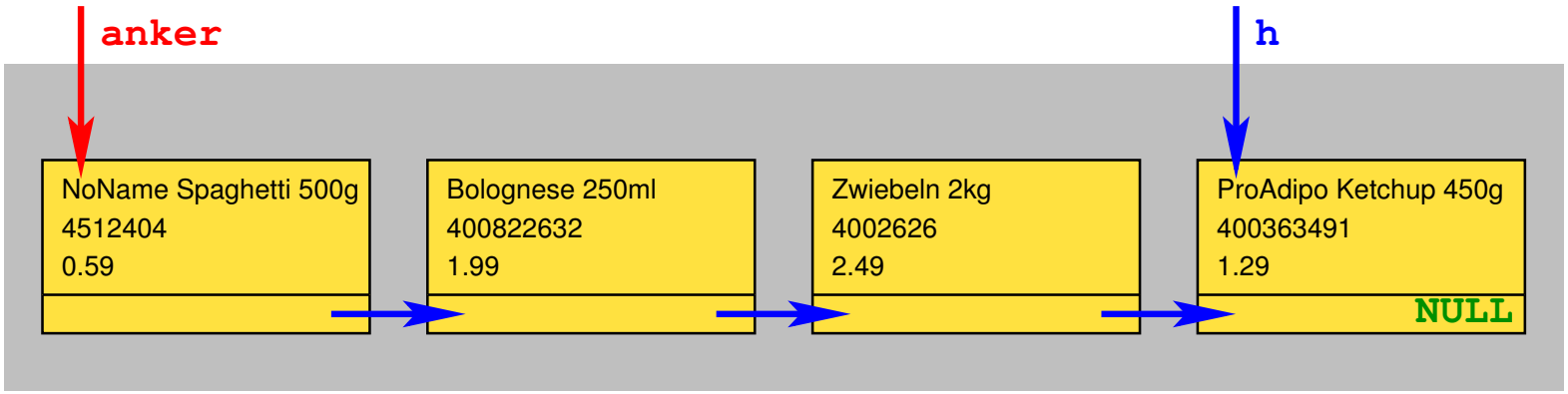



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 3

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen

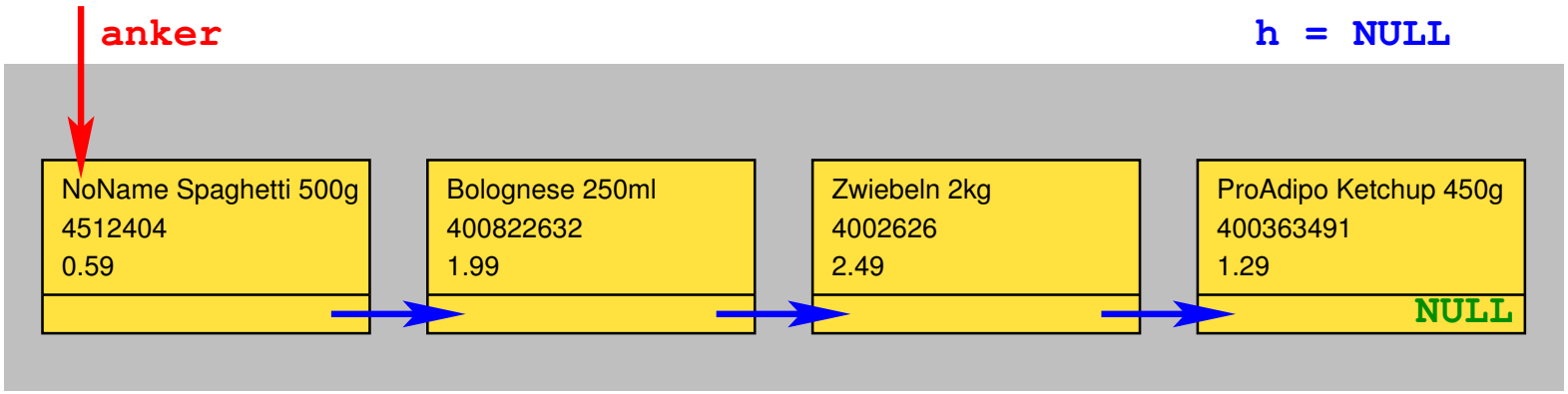



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 4

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

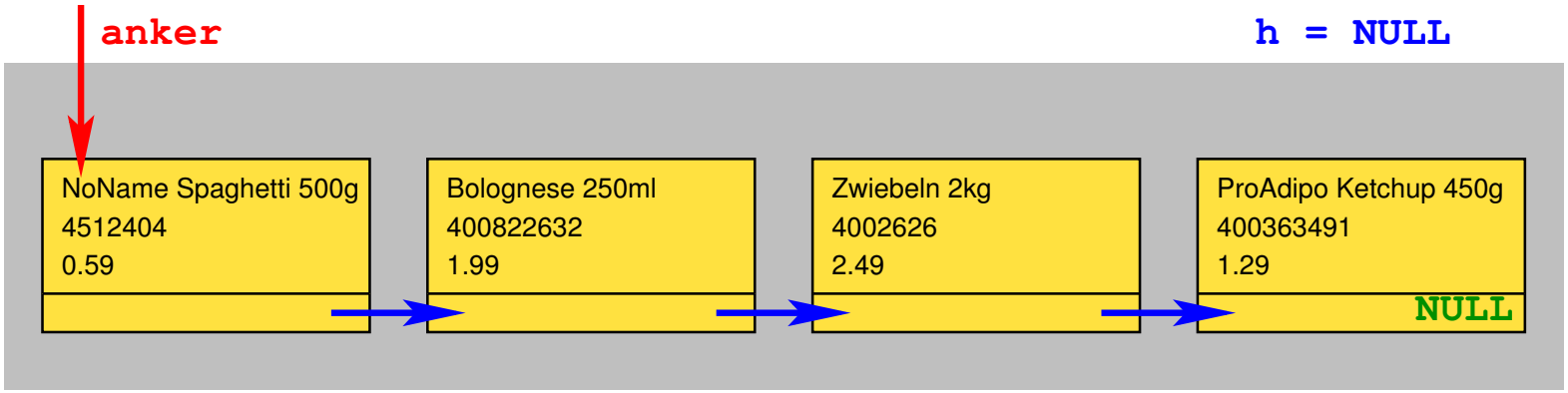



```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 4

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen



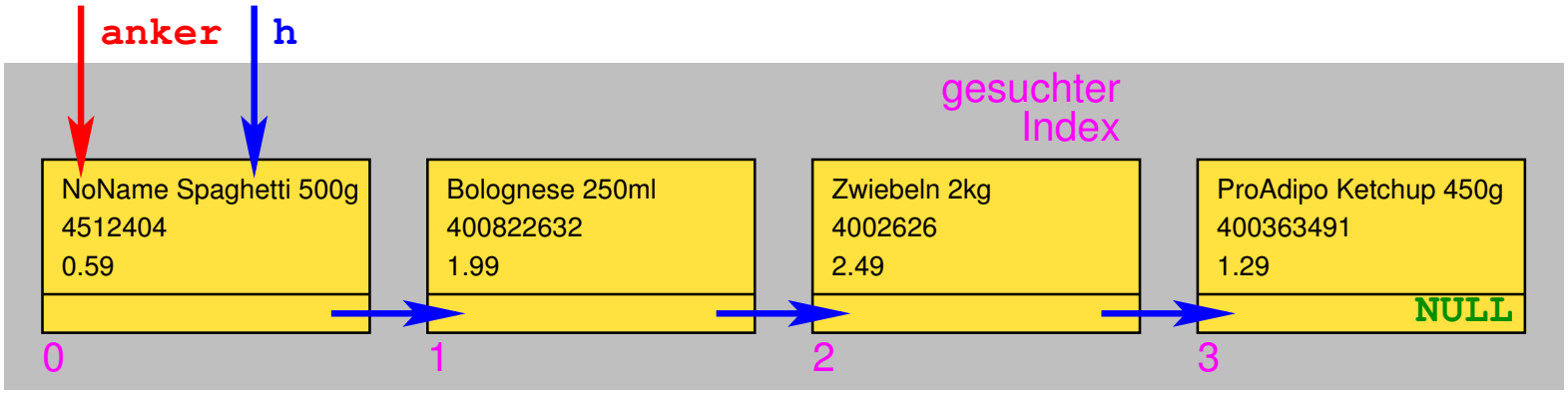

```
int i = 0;
struct TWarenkorb *h = anker;

while (h != NULL)
{
  i++;
  h = h->next;
}
```

i: 4

Listenende erreicht, **`while`**-Schleife wird verlassen. Wert von `i` entspricht Anzahl Listenelemente.

Listenelement suchen und auslesen

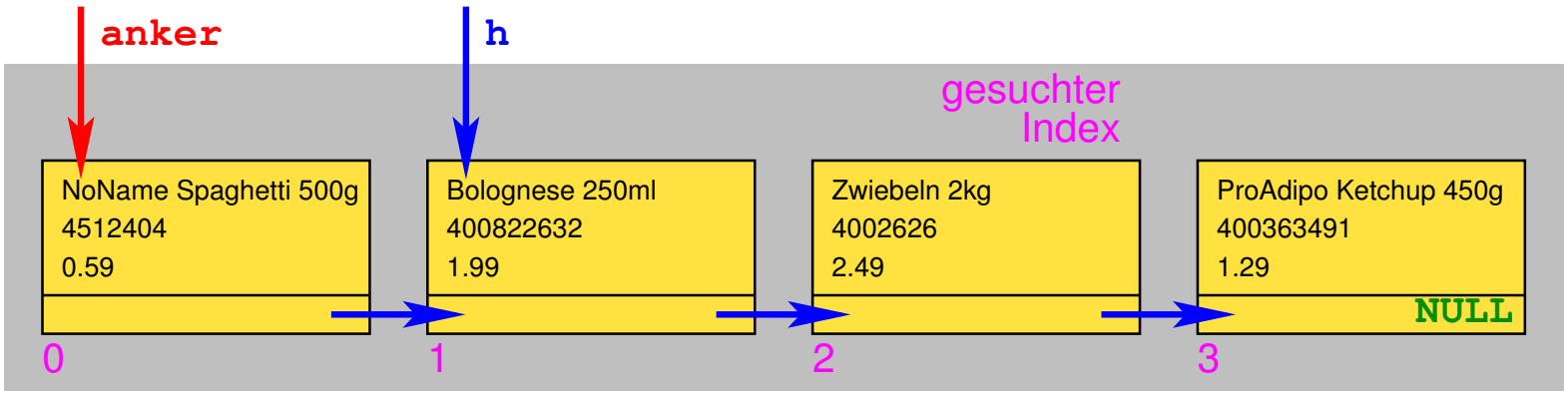
Listenelement suchen und auslesen („get“)




```
int i = 0;
struct TWarenkorb *h = anker;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
while (i < index)
{
  i++;
  h = h->next;
}
*name = h->produktname;
*code = h->produkcode;
*stueckpreis = h->preis;
```

Listenelemente von 0 beginnend fortlaufend durchnummeriert *(Index)*

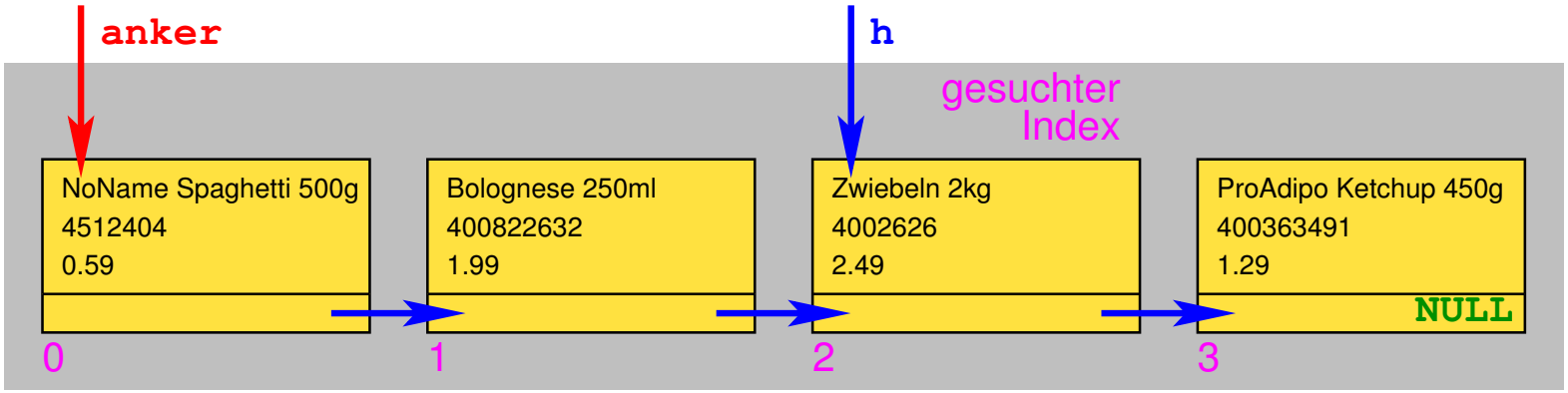
Listenelement suchen und auslesen („get“)




```
int i = 0;
struct TWarenkorb *h = anker;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
while (i < index)
{
  i++;
  h = h->next;
}
*name = h->produktname;
*code = h->produkcode;
*stueckpreis = h->preis;
```

Liste bis zum gesuchten Index elementweise durchlaufen

Listenelement suchen und auslesen („get“)

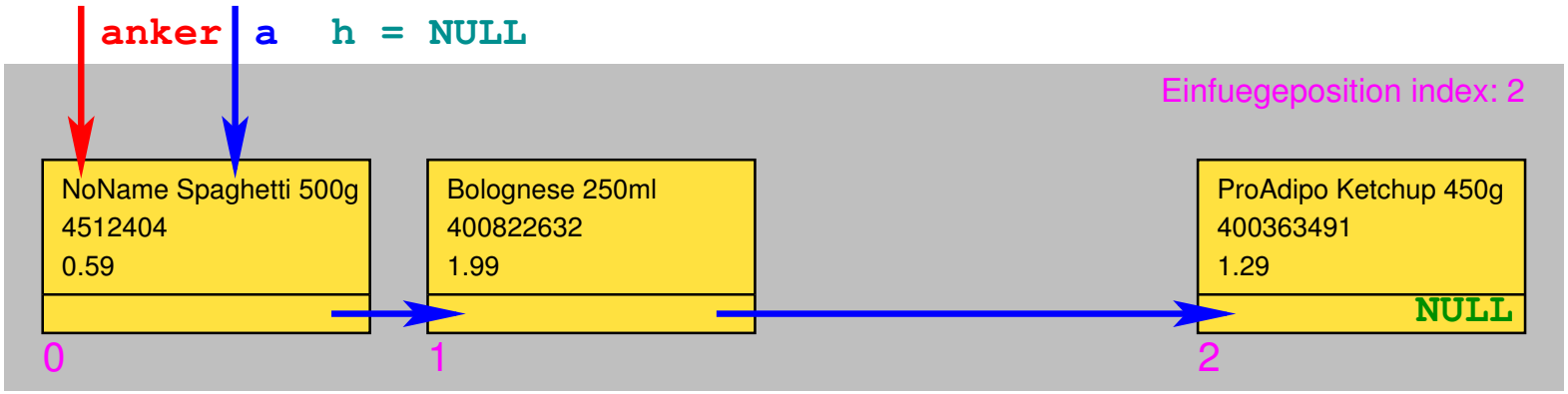



```
int i = 0;
struct TWarenkorb *h = anker;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
while (i < index)
{
  i++;
  h = h->next;
}
*name = h->produktname;
*code = h->produkcode;
*stueckpreis = h->preis;
```

Bei Erreichen des gesuchten Index Komponenten des Informationsteils bereitstellen

Listenelement an gegebener Indexposition einfügen

Listenelement an gegebener Indexposition einfügen

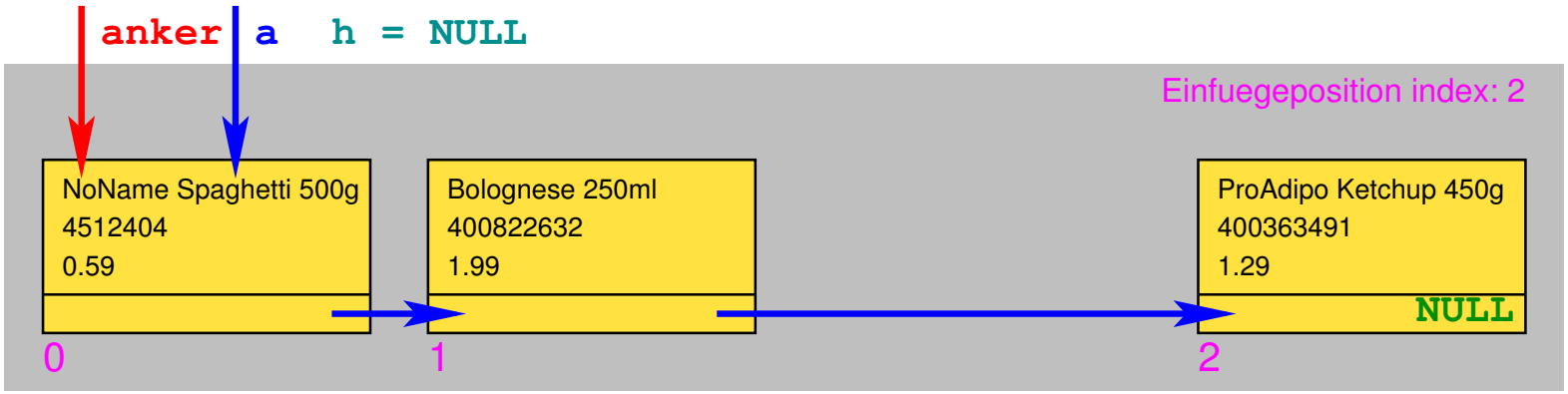



```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;
```

```
if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Hilfszeiger **a** auf Listenanfang, Hilfszeiger **h** für neues Listenelement

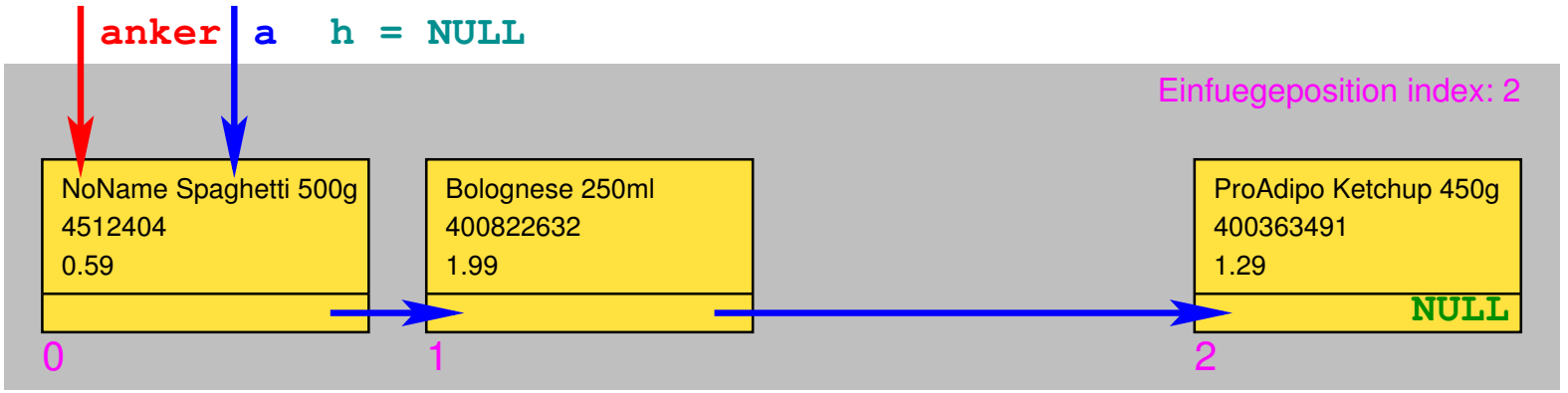
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;
if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Plausibilitätscheck: Indexposition zum Einfügen zulässig?

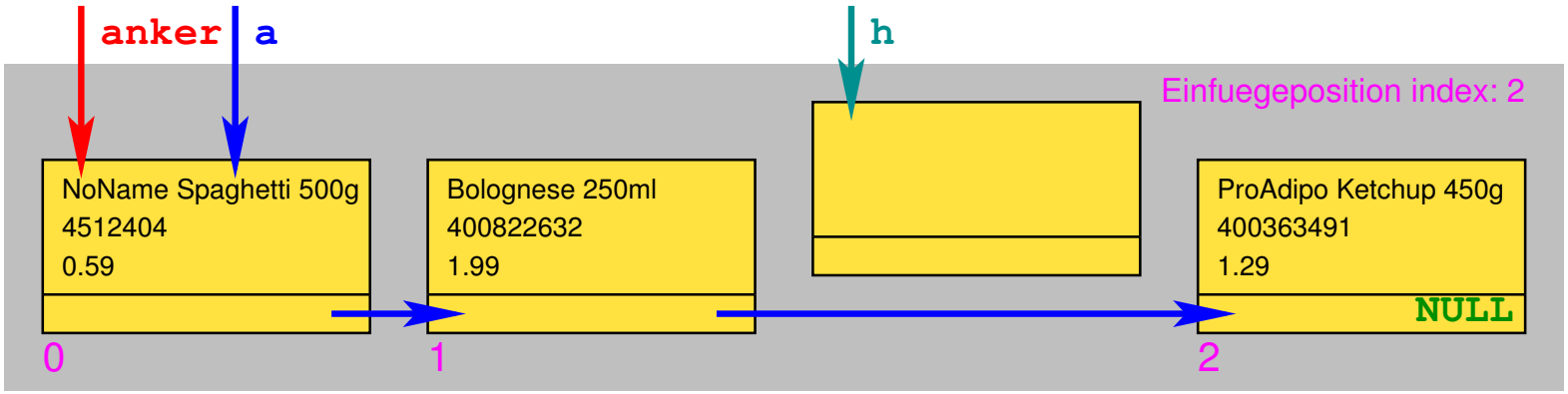
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;

if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Einfügen an Indexposition 0 entspricht Einfügen am Listenanfang

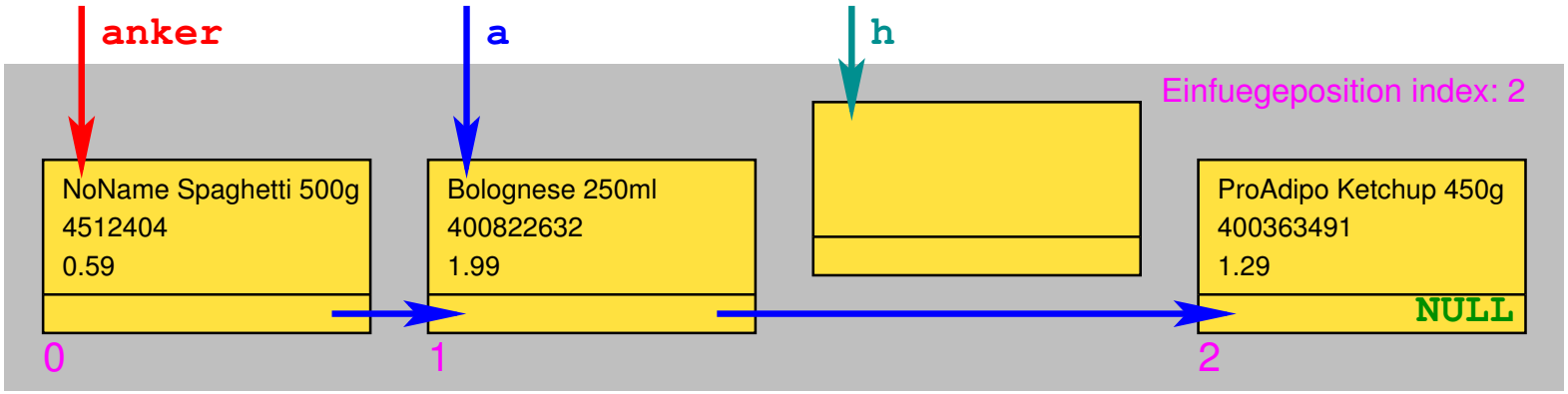
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;

if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Speicherplatz für neues Listenelement ausfassen

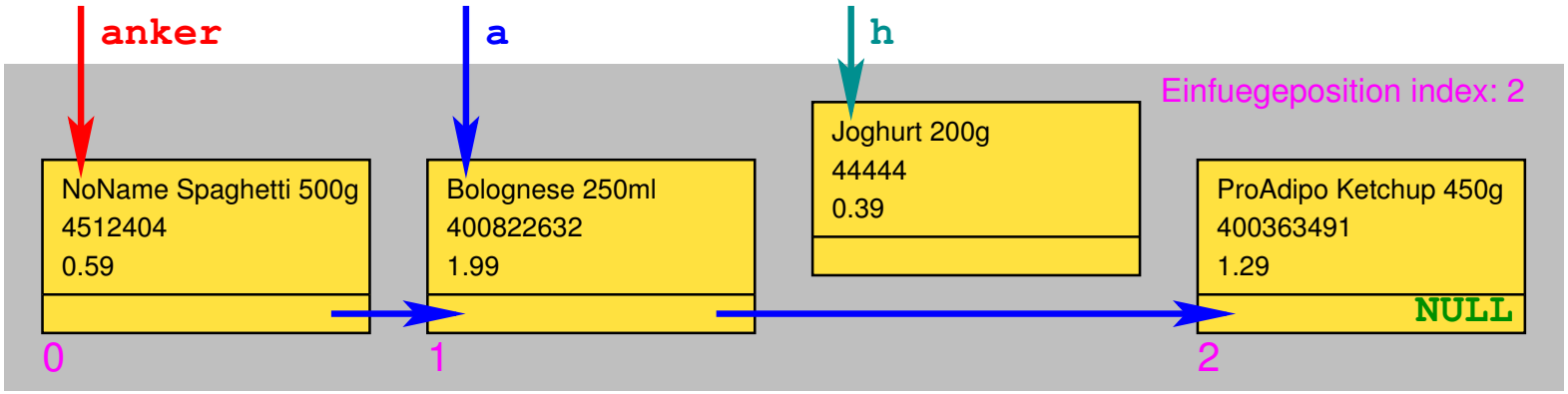
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;

if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Liste durchlaufen, Zeiger **a** unmittelbar vor einzufügendem Elem. platzieren

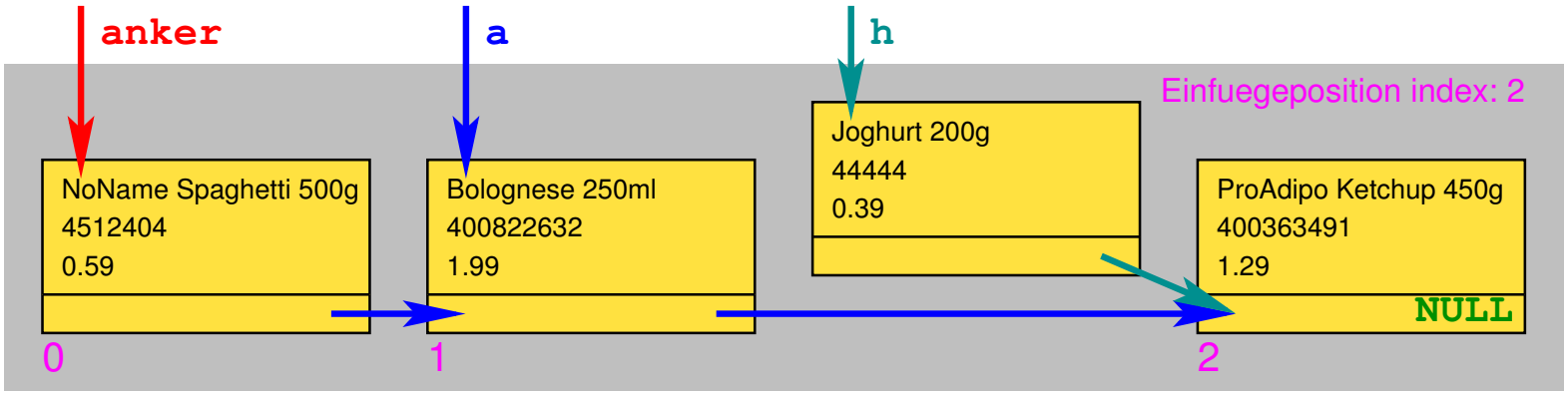
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;

if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Informationsteil des neuen Listenelements füllen

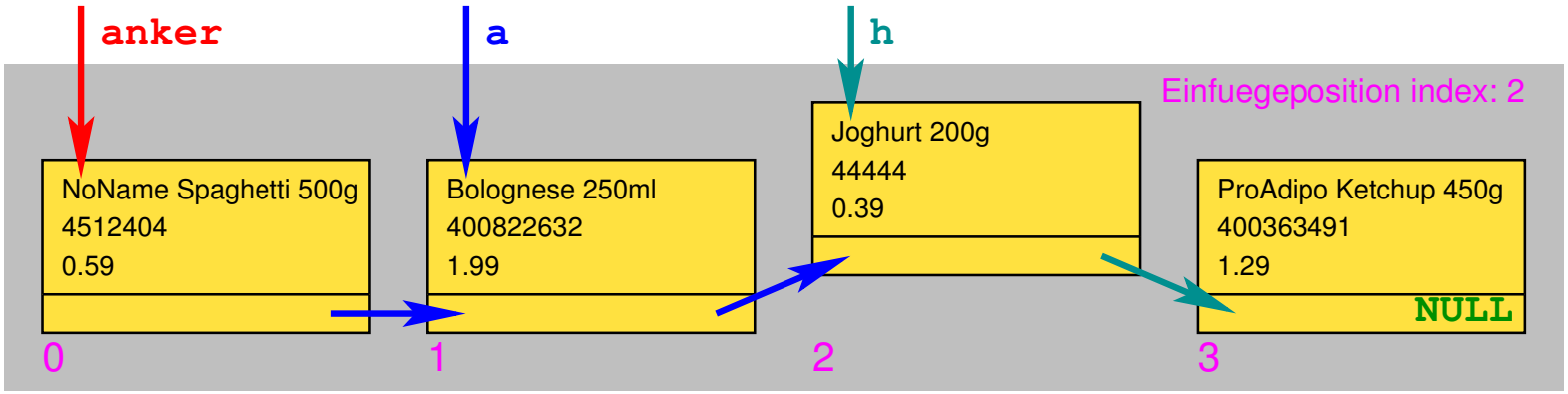
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;

if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Verkettung zum Nachfolger des neuen Listenelements setzen

Listenelement an gegebener Indexposition einfügen

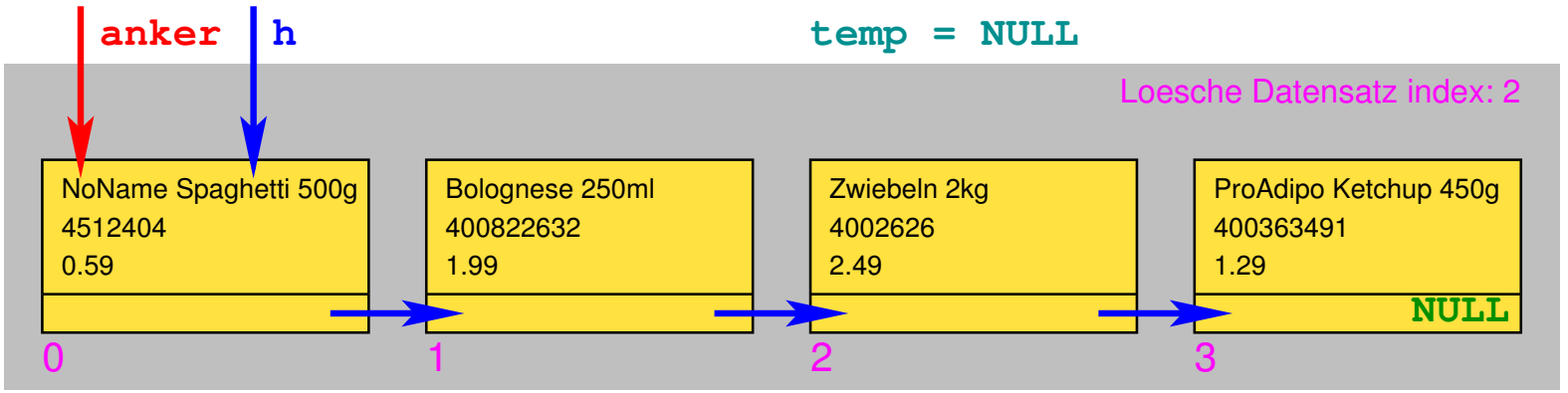



```
int i = 0;
struct TWarenkorb *a = anker;
struct TWarenkorb *h = NULL;

if (index < 0) || (index >= number_elems())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
h = malloc(sizeof(struct TWarenkorb));
if (h == NULL) { /* Nicht genug Speicher */ }
while (i < index - 1)
{
  i++;
  a = a->next;
}
h->produktcode = "44444";
strcpy(h->produktname, "Joghurt 200g");
h->preis = 0.39;
h->next = a->next;
a->next = h;
```

Neues Listenelement an Vorgänger anketten, Einfügen fertig

Listenelement an gegebener Indexposition löschen

Listenelement an gegebener Indexposition löschen

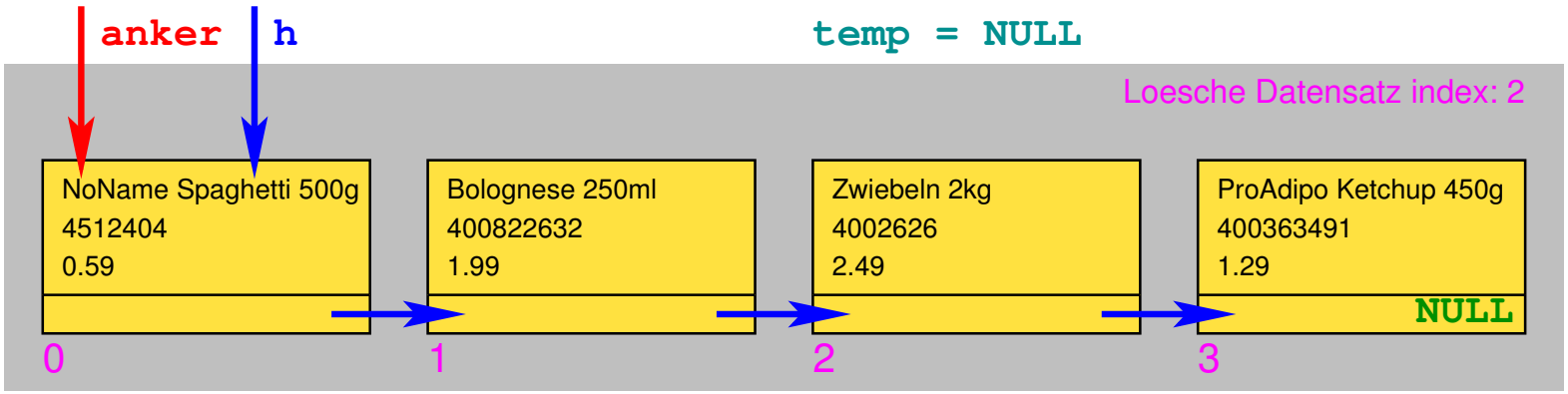



```
int i = 0;
struct TWarenkorb *h = anker;
struct TWarenkorb *temp = NULL;
```

```
if ((index < 0) || (index >= number_elems())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h->next;
}
temp = h->next;
h->next = h->next->next;
free(temp);
```

Hilfszeiger **h** auf Listenanfang, Hilfszeiger **temp** anlegen

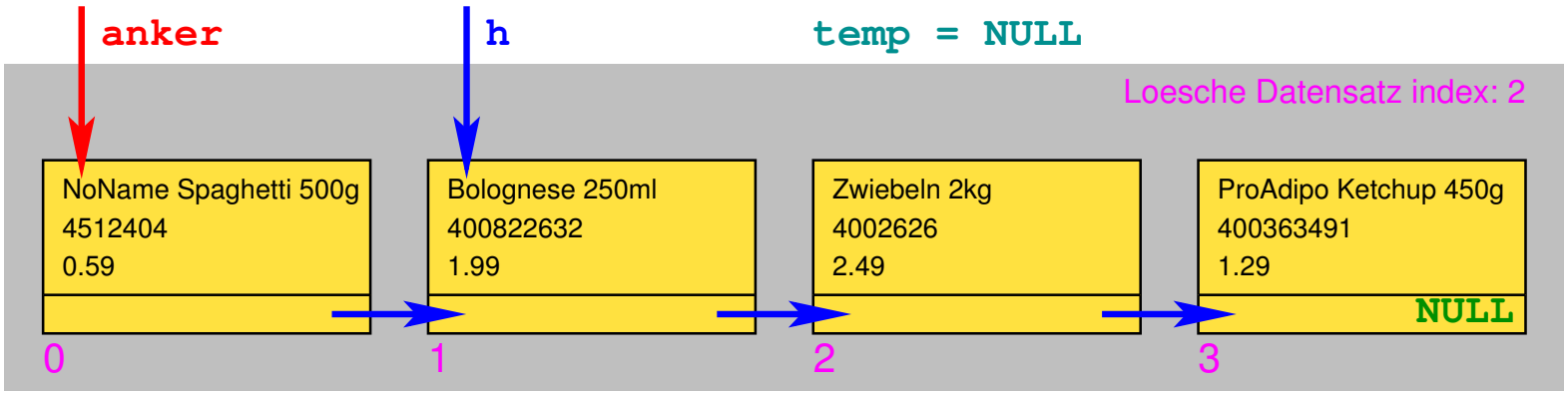
Listenelement an gegebener Indexposition löschen




```
int i = 0;
struct TWarenkorb *h = anker;
struct TWarenkorb *temp = NULL;
if ((index < 0) || (index >= number_elems())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h->next;
}
temp = h->next;
h->next = h->next->next;
free(temp);
```

Plausibilitätscheck: Indexposition zum Löschen zulässig?

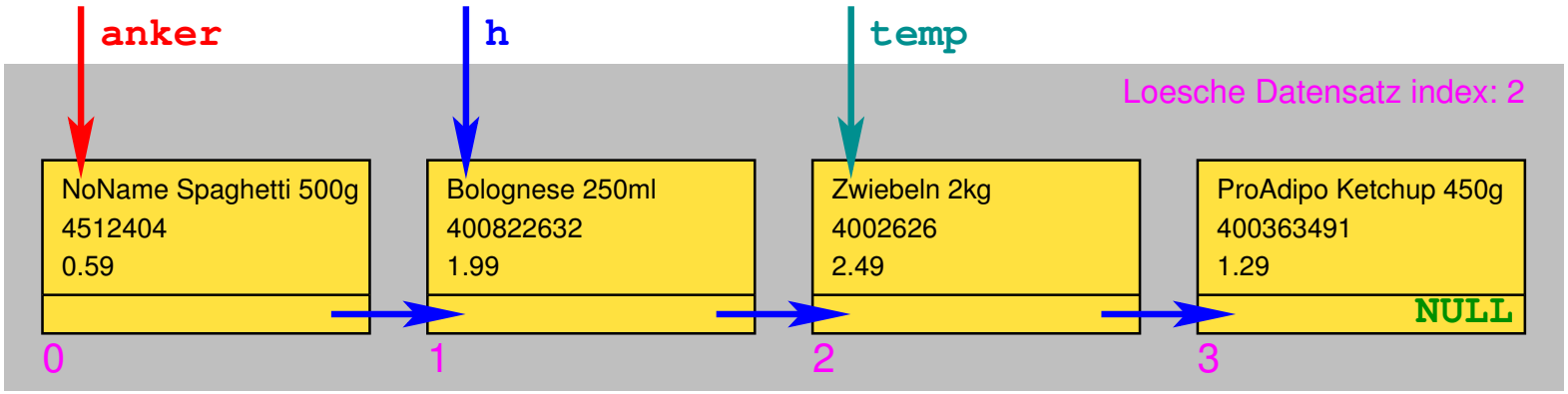
Listenelement an gegebener Indexposition löschen




```
int i = 0;
struct TWarenkorb *h = anker;
struct TWarenkorb *temp = NULL;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h->next;
}
temp = h->next;
h->next = h->next->next;
free(temp);
```

Liste durchlaufen, Zeiger **h** unmittelbar vor zu löschendem Elem. platzieren

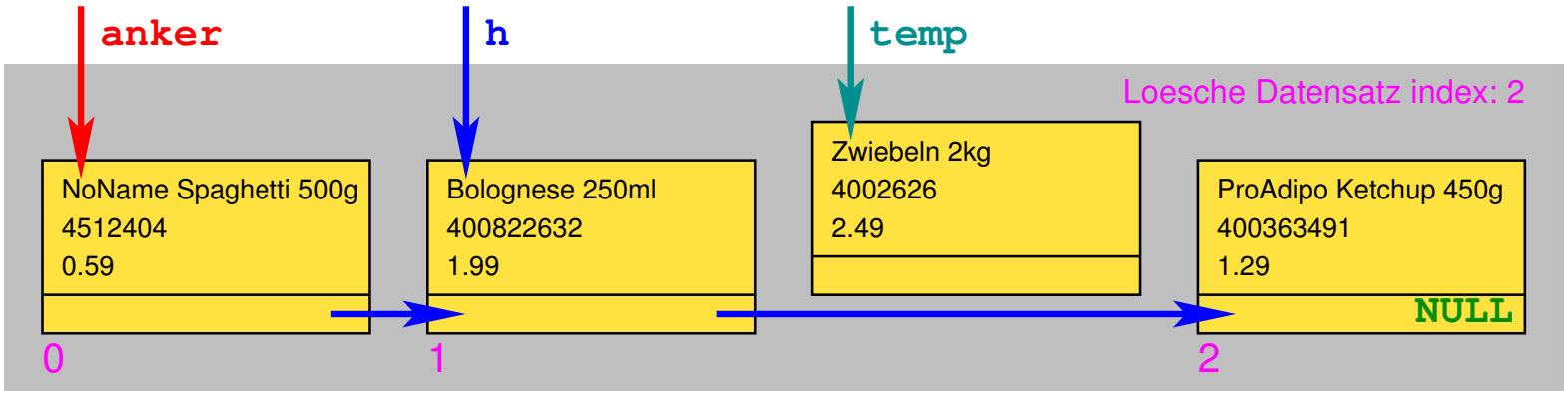
Listenelement an gegebener Indexposition löschen




```
int i = 0;
struct TWarenkorb *h = anker;
struct TWarenkorb *temp = NULL;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h->next;
}
temp = h->next;
h->next = h->next->next;
free(temp);
```

`temp` zeigt auf zu löschendes Listenelement

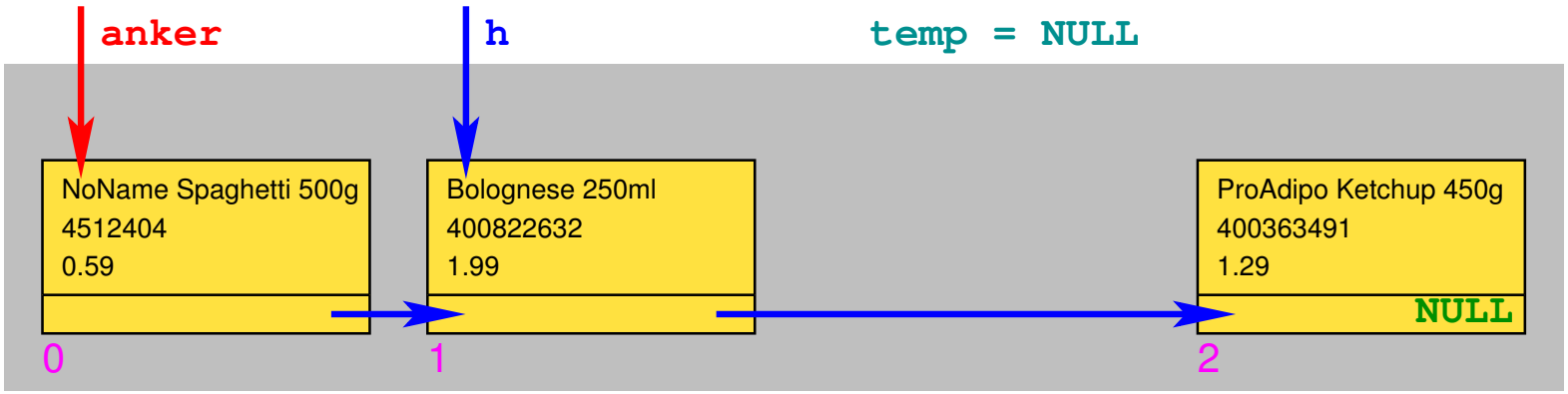
Listenelement an gegebener Indexposition löschen




```
int i = 0;
struct TWarenkorb *h = anker;
struct TWarenkorb *temp = NULL;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h->next;
}
temp = h->next;
h->next = h->next->next;
free(temp);
```

Zu löschendes Listenelement ausketten und Nachfolger neu setzen

Listenelement an gegebener Indexposition löschen

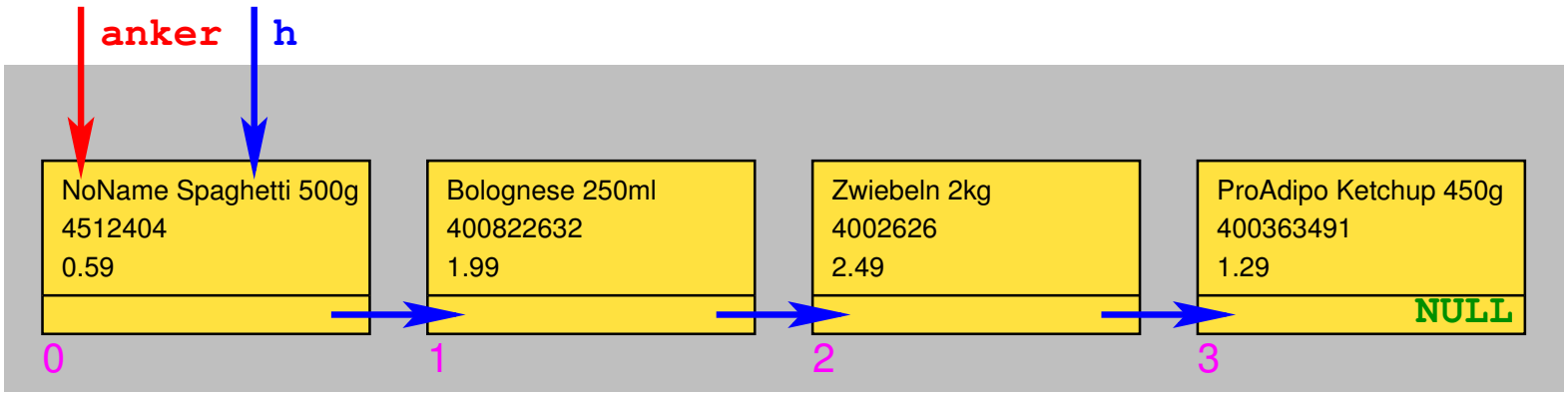



```
int i = 0;
struct TWarenkorb *h = anker;
struct TWarenkorb *temp = NULL;

if ((index < 0) || (index >= number_elems())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h->next;
}
temp = h->next;
h->next = h->next->next;
free(temp);
```

Speicherplatz von gelöschtem Listenelement freigeben. Fertig.

Listenelement am Listenanfang löschen (index 0)

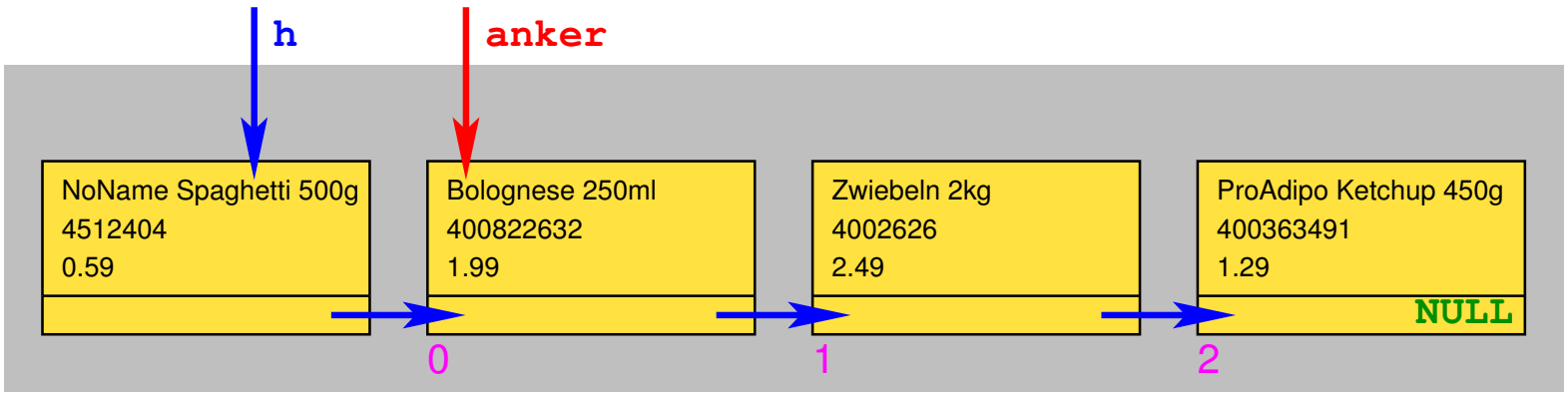
Listenelement am Listenanfang löschen




```
struct TWarenkorb *h = anker;
anker = h->next;
free(h);
```

Hilfszeiger **h** auf Listenanfang setzen

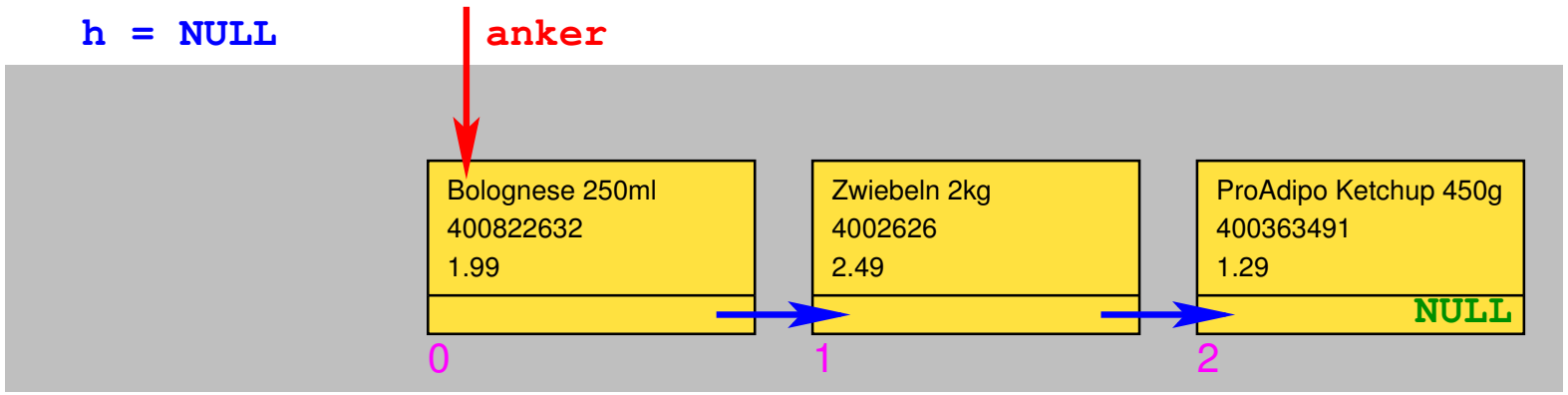
Listenelement am Listenanfang löschen




```
struct TWarenkorb *h = anker;
anker = h->next;
free(h);
```

Listenanker **anker** ein Element weiterschieben

Listenelement am Listenanfang löschen




```
struct TWarenkorb *h = anker;
anker = h->next;
free(h);
```

Speicherplatz von gelöschtem Listenelement freigeben. Fertig.

Unsere kleine selbstprogrammierte Datenbank

Quelltextbeispiel warenkorb.c

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

#define MAX_STRINGLAENGE 256

/* ----- globale Datenstruktur einfach verkettete lineare Liste */

struct TWarenkorb
{
  char produktname[MAX_STRINGLAENGE];  //Produktbezeichnung als Zeichenkette
  unsigned long produktcode;           //z.B. EAN-Strichcode
  float preis;                         //Stueckpreis in Euro
  struct TWarenkorb *next;             //Verweis auf das nachfolgende Listenelement
};

struct TWarenkorb *anker = NULL;
```

Gesamter Quelltext **`warenkorb.c`** zum Download auf LV-Webseite

Liste von Artikeln verwalten

```
Mein Warenkorb
  1: Neuen Datensatz anlegen. Produkt, Produktcode und Stueckpreis eingeben
  2: Bestehenden Datensatz loeschen
  3: Datensaetze aufsteigend nach Stueckpreis sortieren (mit Insertsort)
  4: Datensaetze anzeigen
  5: Datensaetze in Datei speichern
  6: Programmende

Auswahl:
```

- Neue Artikel in den Warenkorb aufnehmen
- Existierenden Artikel aus Warenkorb (Liste) löschen
- Artikel aufsteigend nach Preis sortieren (Insertionsort)
- Artikel im Warenkorb sowie den Gesamtpreis anzeigen
- Artikel im Warenkorb in Datei speichern

Programmierte Funktionen

Datenverwaltungsschicht

`insert` ..... neuen Datensatz am Listenanfang einfügen
`number_elems` ..... Anzahl Datensätze bestimmen
`insert_at` ..... neuen Datensatz an geg. Position einfügen
`get` ..... Infoteil des Datensatzes an geg. Position liefern
`delete` ..... Datensatz an geg. Position löschen
`save_list` ..... alle Datensätze in Datei schreiben

Anwendungskern

`total_price` ..... Gesamtpreis berechnen
`insertsort` ..... Datensätze aufsteigend nach Preis sortieren

Nutzeroberfläche

`display_list` ..... alle Datensätze anzeigen
`main` ..... Menü und Nutzereingaben

Arbeit mit dem Datenbanktool

Nach dem Anlegen von vier Datensätzen

```
Gesamte Liste:
Nr.   Produkt                          Produktcode  Stueckpreis
-------------------------------------------------------------
   0,          NoName_Spaghetti_500g,    4512404,   0.59
   1,                Bolognese_250ml,  400822632,   1.99
   2,                   Zwiebeln_2kg,    4002626,   2.49
   3,          ProAdipo_Ketchup_450g,  400363491,   1.29
```

Arbeit mit dem Datenbanktool

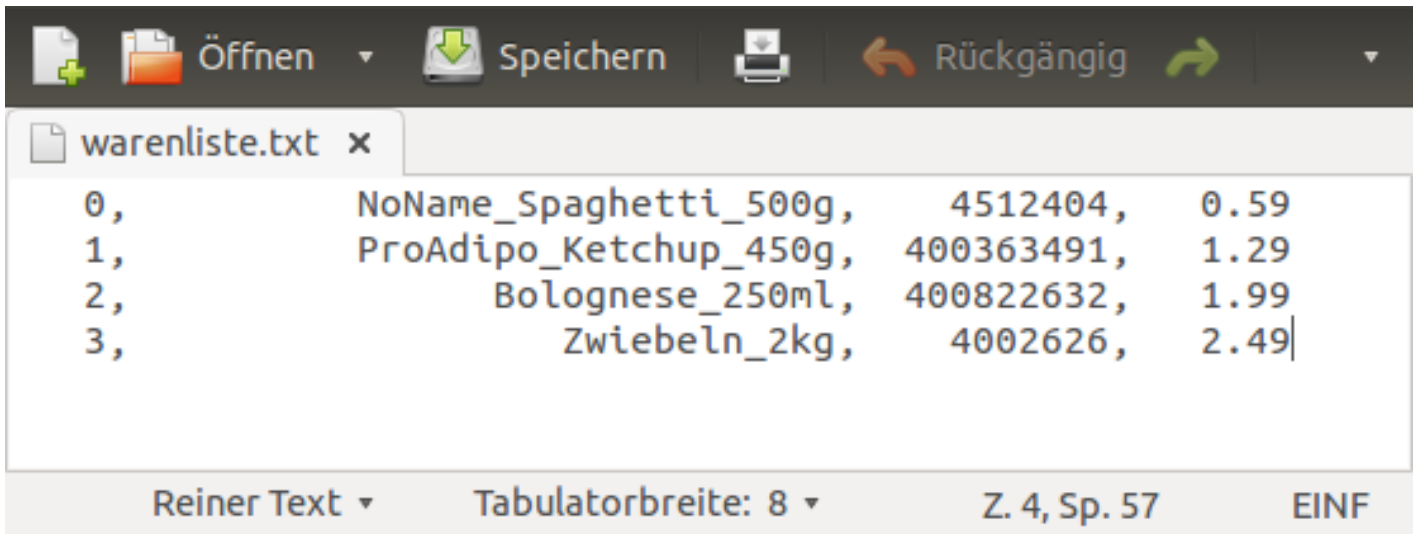
Nach dem Anlegen von vier Datensätzen

```
Gesamte Liste:
Nr.   Produkt                          Produktcode  Stueckpreis
-------------------------------------------------------------
   0,          NoName_Spaghetti_500g,    4512404,   0.59
   1,                Bolognese_250ml,  400822632,   1.99
   2,                   Zwiebeln_2kg,    4002626,   2.49
   3,          ProAdipo_Ketchup_450g,  400363491,   1.29
```

Nach dem Sortieren

```
Gesamte Liste:
Nr.   Produkt                          Produktcode  Stueckpreis
-------------------------------------------------------------
   0,          NoName_Spaghetti_500g,    4512404,   0.59
   1,          ProAdipo_Ketchup_450g,  400363491,   1.29
   2,                Bolognese_250ml,  400822632,   1.99
   3,                   Zwiebeln_2kg,    4002626,   2.49
```

Geschriebene Textdatei mit den Listeneinträgen

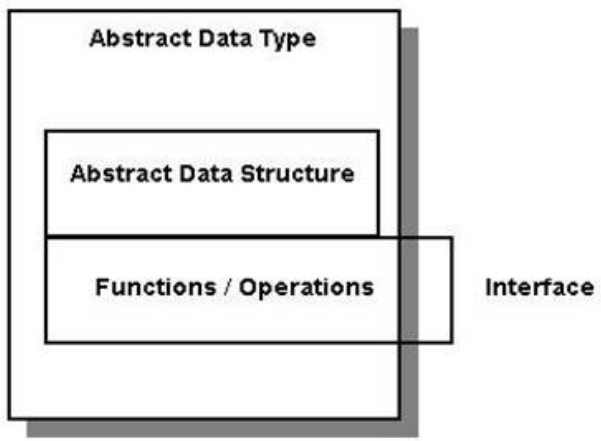


Insertionsort auf der Liste

```
/* ----- Liste aufsteigend nach Preis sortieren mit Insertionsort */

int insertsort(void)
{
  unsigned long code, code2;
  float  stueckpreis, stueckpreis2;
  char pn[MAX_STRINGLAENGE];
  char *name, *name2;
  int k = 1; /* Index, bis wohin schon sortiert */
  int r;

  while(k<number_elems())
  {
    get(k, &name, &code, &stueckpreis);
    strcpy(pn, name);
    delete(k);
    r = 0;
    do
    {
      get(r, &name2, &code2, &stueckpreis2);
      r++;
    } while ((r-1 < k) && (stueckpreis > stueckpreis2));
    insert_at(r-1, pn, code, stueckpreis);
    k++;
  }
  return 0;
}
```

Abstrakter Datentyp

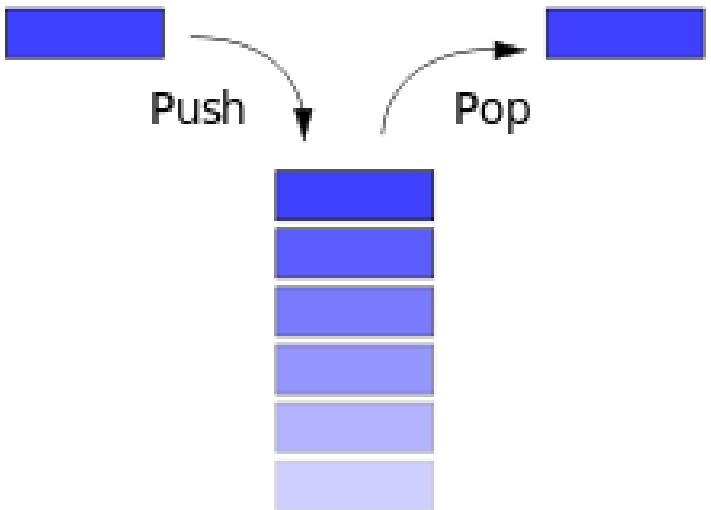
Als **abstrakten Datentyp** bezeichnet man eine *Datenstruktur* gemeinsam mit darauf wirkenden *Operationen*.



Eine *Lineare Liste* mit den Operationen ***number_elems***, ***insert***, ***get*** und ***delete*** (jeweils mit frei wählbarer Indexposition) ist ein Beispiel für einen abstrakten Datentypen.

Stack

Stapelspeicher nach dem Prinzip „Last In First Out (LIFO)“



www.wikipedia.de

Lineare Liste mit den Operationen

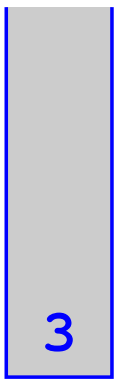
`isEmpty` ..... Test auf leeren Stack
`push` ..... Neues Element auf den Stack legen
`pop` ..... Oberstes Element auslesen und vom Stack nehmen

Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```

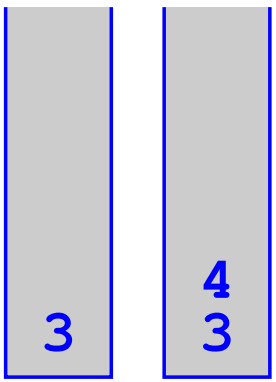


Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```

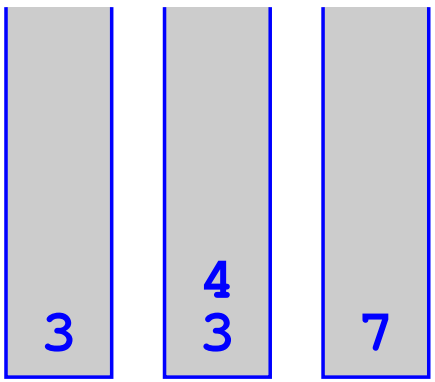


Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```

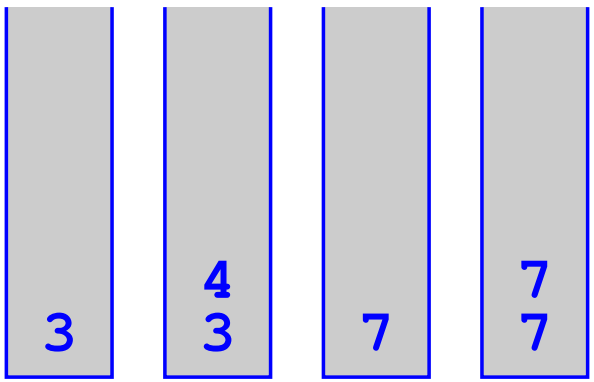


Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```

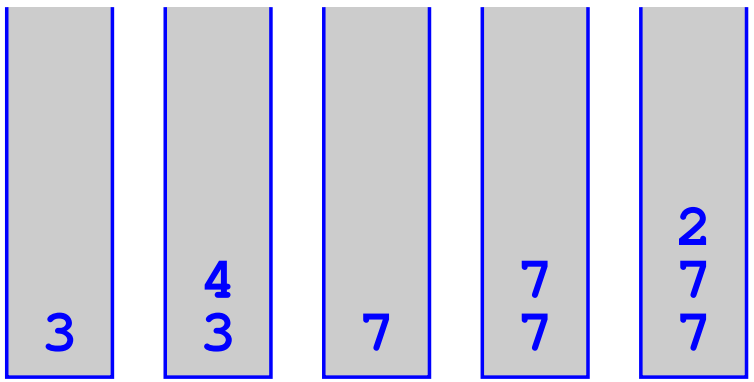


Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```

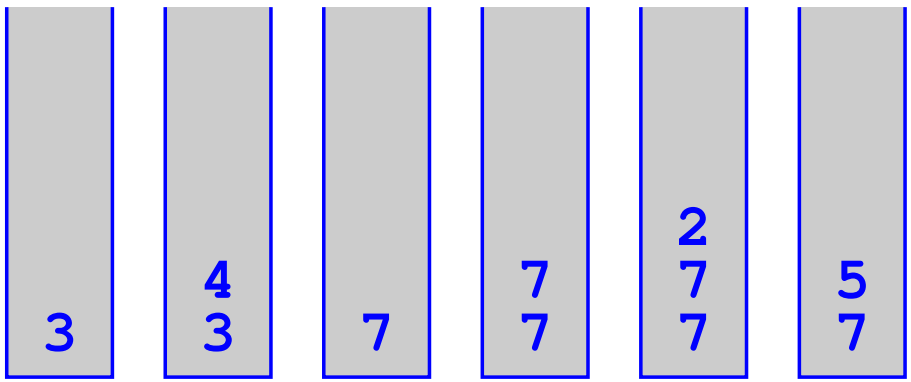


Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```

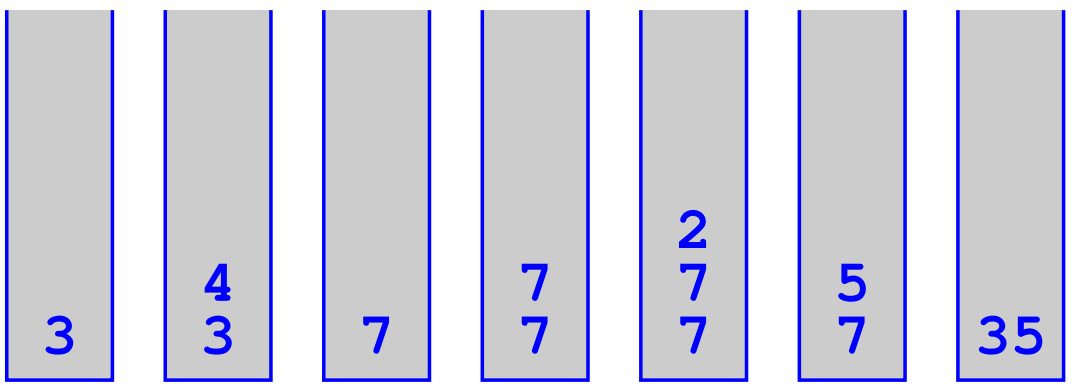


Stack zur Auswertung arithmetischer Terme

```
( 3 + 4 ) * ( 7 - 2 )
```

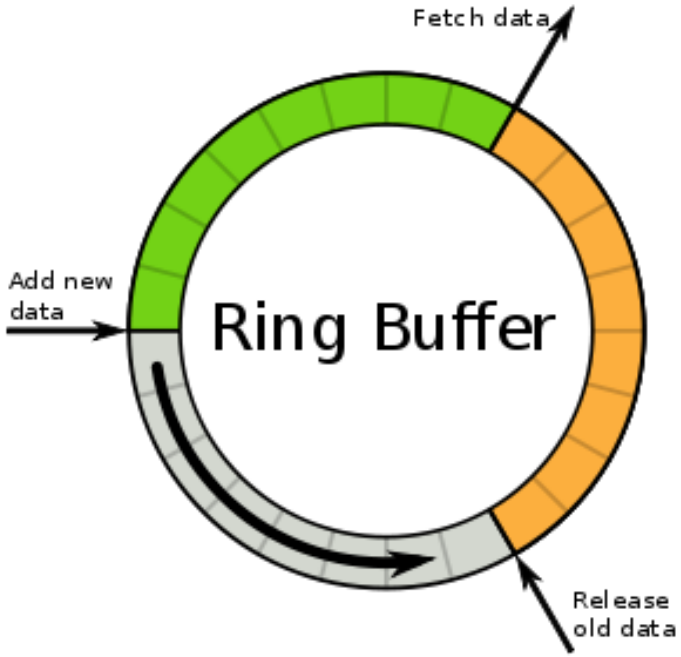
Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

```
3   4   +   7   2   -   *
```



Ringpuffer

eingesetzt als Warteschlange „First In First Out (FIFO)“

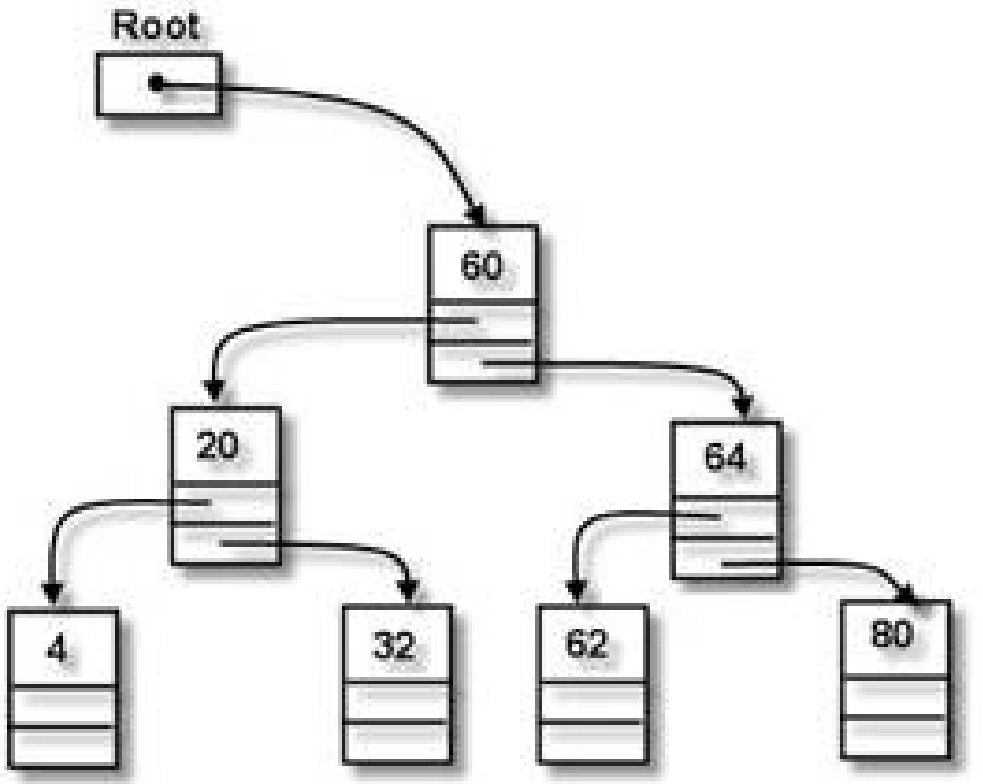


www.wikipedia.de

Zyklische Liste mit den Operationen

`isEmpty` ..... Test auf leeren Puffer
`enqueue` ..... Neues Element am Ende einfügen
`dequeue` ..... Führendes Element auslesen und entfernen

Binärer Baum



www.wikipedia.de

- Jedes Element hat bis zu zwei Nachfolger
- Lineare Liste ist Spezialfall eines binären Baumes
- Große Datenbestände vorteilhaft als balancierter binärer Suchbaum organisiert (sog. AVL-Baum)