Einführung in die Programmierung

Vorlesungsteil 10

Dynamische Datenstruktur Lineare Liste

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik

Sommersemester 2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

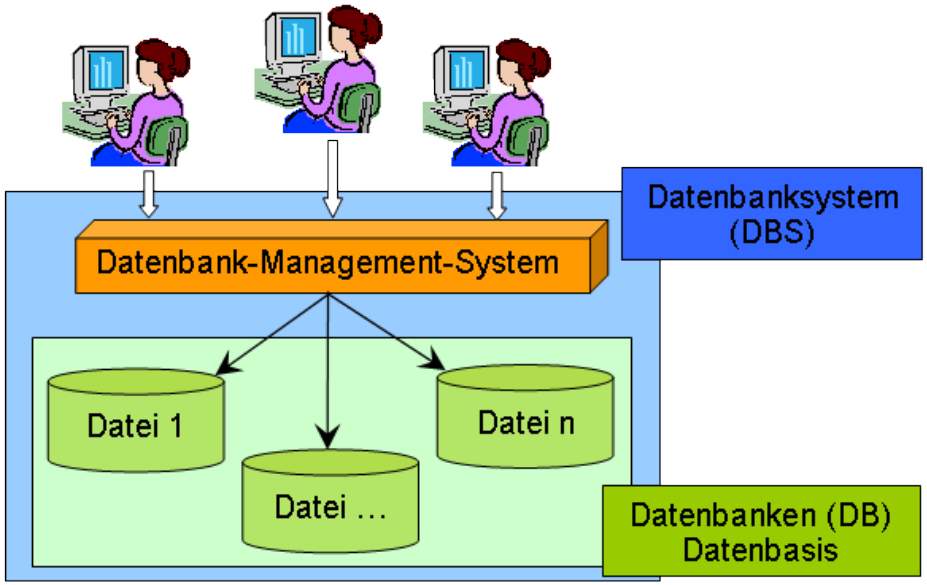
Datenbanken – eine Kernanwendung der Informatik

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.

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.

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.

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.

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 Java

1. Einführung und erste Schritte . . . . . .
. . Installation Java-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

2. Elementare Datentypen, Variablen, Arithmetik, Typecast . . . . . .
Java als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe

3. Imperative Kontrollstrukturen . . . . . .
. . . Befehlsfolgen, Verzweigungen, Schleifen und logische Ausdrücke programmieren

4. Methoden selbst programmieren . . . . . .
. . . . Methoden als wiederverwendbare Funktionen, Werteübernahme und -rückgabe

5. Rekursion . . . . . .
. . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel

6. Objektorientiert programmieren . . . . . .
. . . . . Klassen, Objekte, Attribute, Methoden, Sichtbarkeit, Vererbung, Polymorphie

7. Felder und Graphen . . . . . .
. . . . . effizientes Handling größerer Datenmengen und Beschreibung von Netzwerken

8. Sortieren . . . . . .
. . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf

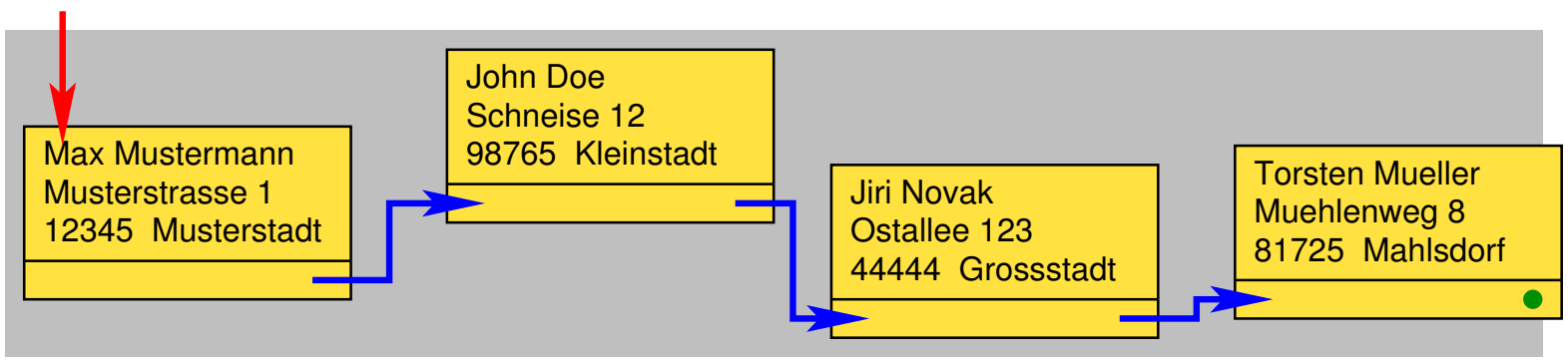
9. Zeichenketten, Dateiarbeit, Ausnahmen . . . . . .
. . . Texte analysieren, ver-/entschlüsseln, Dateien lesen/schreiben, Fehler behandeln

10. Dynamische Datenstruktur „Lineare Liste“ . . . . . .
. . . . . unsere selbstprogrammierte kleine Datenbank

11. Ausblick und weiterführende Konzepte . . . . . .

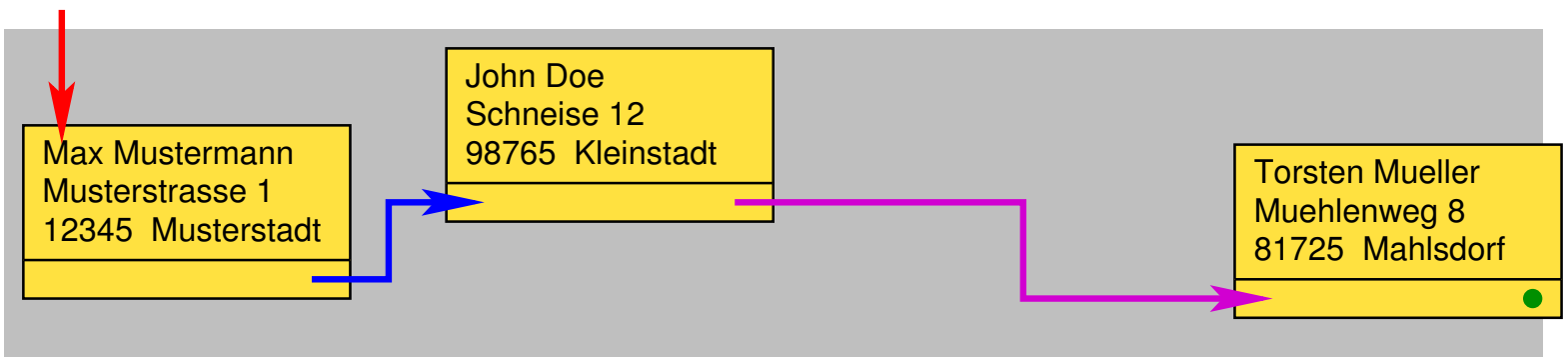
Dynamische Datenstruktur Lineare Liste

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

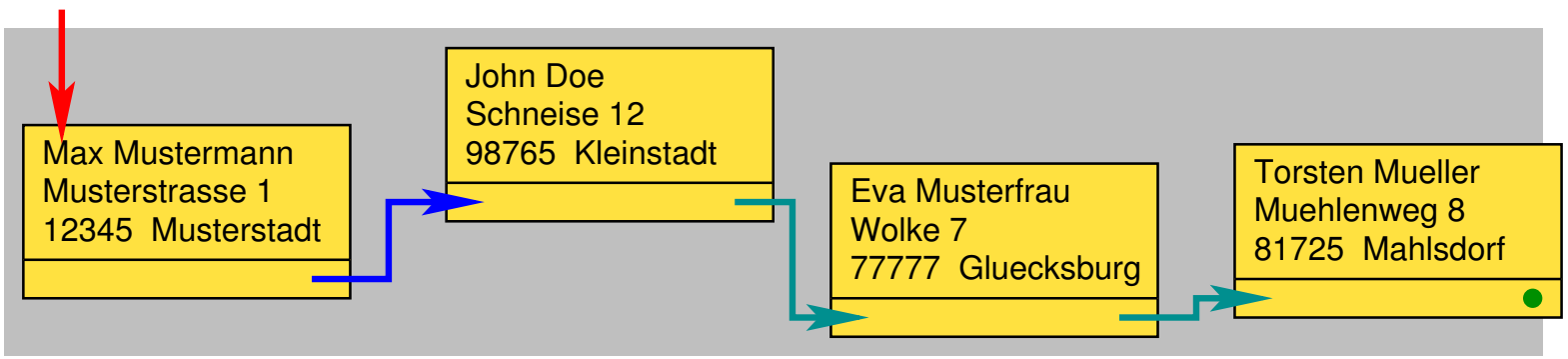
Eine **Lineare Liste** ist eine dynamische Datenstruktur, deren *Elemente (Datensätze)* beliebig im Speicher verteilt sein können. Jedes Element der Liste – bis auf das letzte – hat genau einen Nachfolger. Der *Zugriff* auf die Liste erfolgt üblicherweise über das erste Element. Jedes Element einer Liste 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.

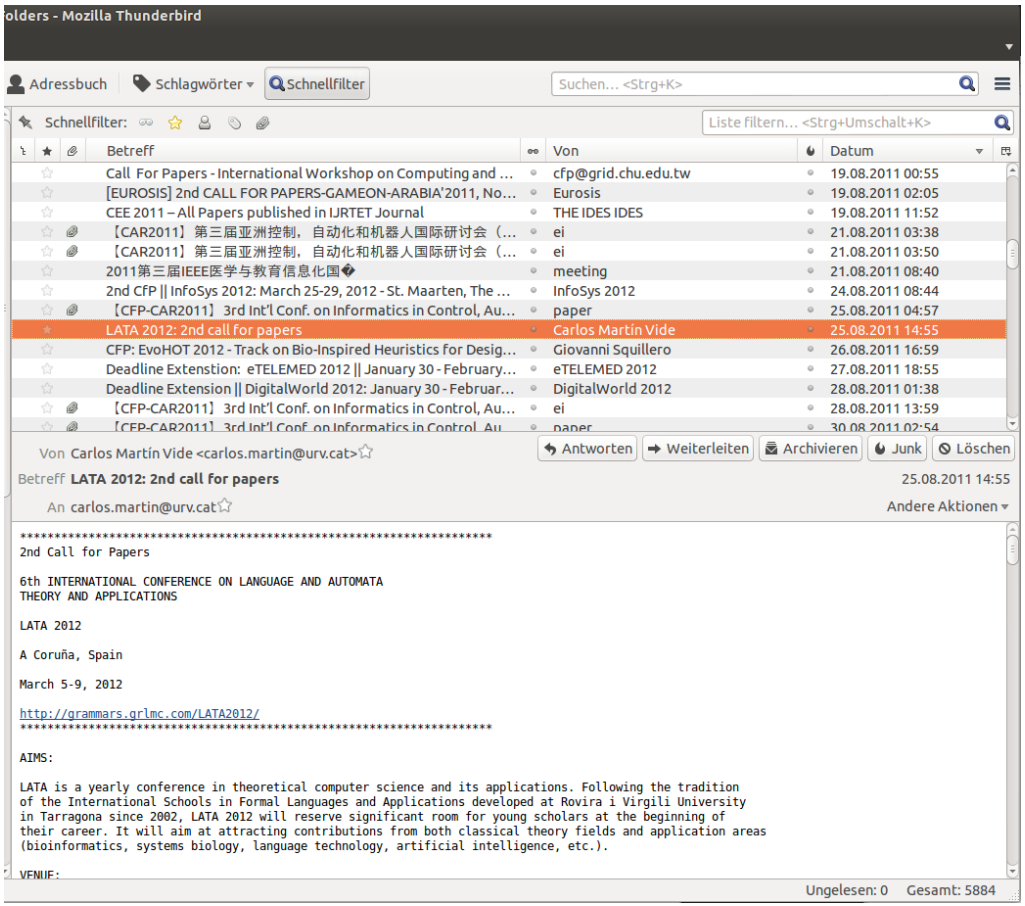
Dynamische Datenstruktur Lineare Liste

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.



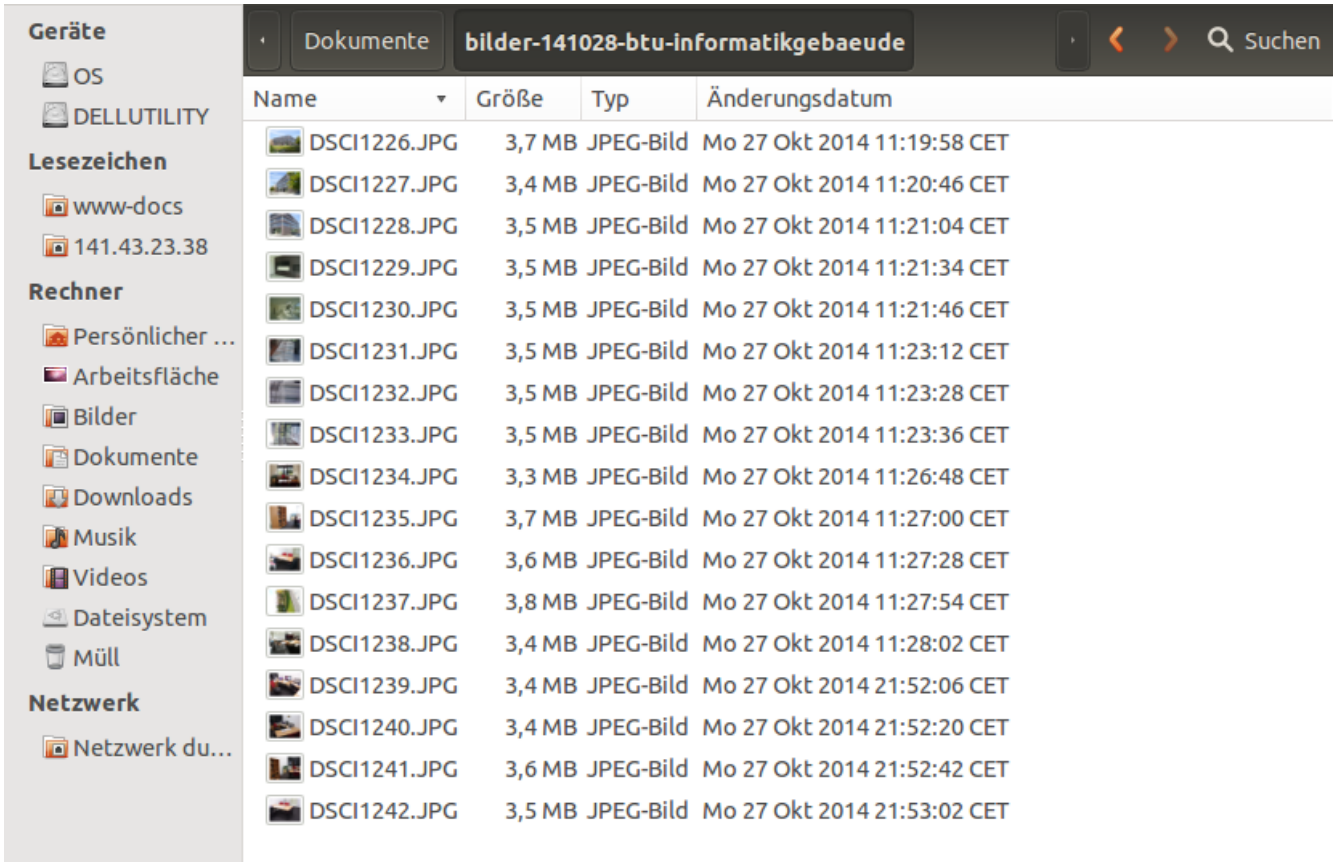
Beim Einfuegen eines neuen Datensatzes werden die betroffenen Vermerke ebenfalls aktualisiert.

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 Java als Objekt beschreiben

Beispiel: Uhrzeit aus Stunde, Minute, Sekunde und Zeitzone

```
public class Uhrzeit {
  int stunde;      // 0...23
  int minute;      // 0...59
  int sekunde;     // 0...59
  String zeitzone; // z.B. "GMT" oder "MESZ"

  // Konstruktor
  public Uhrzeit(int h, int min, int sek, String zz) {
    this.stunde = h;
    this.minute = min;
    this.sekunde = sek;
    this.zeitzone = zz;
  }
}

public class Weckzeit {
  public static void main(String[] args) {
    Uhrzeit myalarm = new Uhrzeit(7, 30, 0, "MESZ"); // Datensatz als Objekt anlegen
  }
}
```

- Ein *Datensatz* fasst mehrere Einzeldaten, die unterschiedlich getypt sein können, zusammen (hier: Stunde, Minute, Sekunde, Zeitzone).
- Eine *Klasse* ist zur Beschreibung einer Datensatzstruktur nutzbar.
- `Uhrzeit myalarm = new Uhrzeit(7, 30, 0, "MESZ");` legt Datensatz als Objekt `myalarm` der Klasse `Uhrzeit` an.

Datensatz löschen und Speicher freigeben

```
public class Uhrzeit {
  int stunde;      // 0...23
  int minute;      // 0...59
  int sekunde;     // 0...59
  String zeitzone; // z.B. "GMT" oder "MESZ"

  // Konstruktor
  public Uhrzeit(int h, int min, int sek, String zz) {
    this.stunde = h;
    this.minute = min;
    this.sekunde = sek;
    this.zeitzone = zz;
  }
}

public class Weckzeit2 {
  public static void main(String[] args) {
    Uhrzeit myalarm = new Uhrzeit(7, 30, 0, "MESZ"); // Datensatz als Objekt anlegen
    myalarm = null; // Datensatz myalarm loeschen
  }
}
```

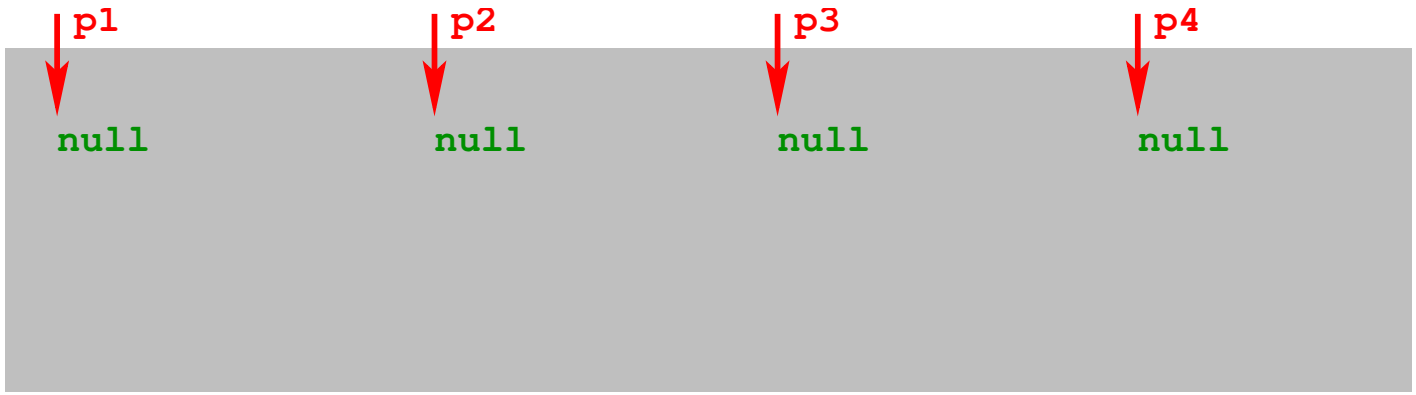
Der *Name eines Objekts* (hier: **`myalarm`**) symbolisiert die *Anfangsadresse* („Referenz“), ab der seine Daten im Speicher abgelegt sind. Zuweisen von **`null`** (*„keine Adresse“* oder *„Nullreferenz“*) signalisiert, dass das Objekt nicht länger im Speicher existieren soll. Die Java-Laufzeitumgebung löscht daraufhin das Objekt und gibt seinen Speicherplatz wieder frei: **`myalarm = null;`**

Datensätze je nach Bedarf anlegen und löschen

```
public class Weckzeit3 {
  public static void main(String[] args) {

    // Zugriffspunkte (Referenzen) auf vier Datensaetze bereitstellen
    Uhrzeit p1 = null;
    Uhrzeit p2 = null;
    Uhrzeit p3 = null;
    Uhrzeit p4 = null;

    p1 = new Uhrzeit( 7, 30,  0, "MESZ"); // Datensatz auf p1 anlegen
    p2 = new Uhrzeit(23, 59, 59, "MESZ"); // Datensatz auf p2 anlegen
    p3 = new Uhrzeit(18, 25,  0, "MESZ"); // Datensatz auf p3 anlegen
    p2 = null;                            // Datensatz auf p2 loeschen
    p2 = new Uhrzeit( 5, 45,  0, "GMT");  // Datensatz auf p2 anlegen
    p4 = new Uhrzeit(12, 12, 30, "MESZ"); // Datensatz auf p4 anlegen
  }
}
```

Objektnamen (hier: **`p1`**, **`p2`**, **`p3`** und **`p4`**), die als Platzhalter dienen, bieten *Zugangspunkte* (Referenzen) auf Datensätze, die dann während der Programmlaufzeit angelegt oder gelöscht werden können

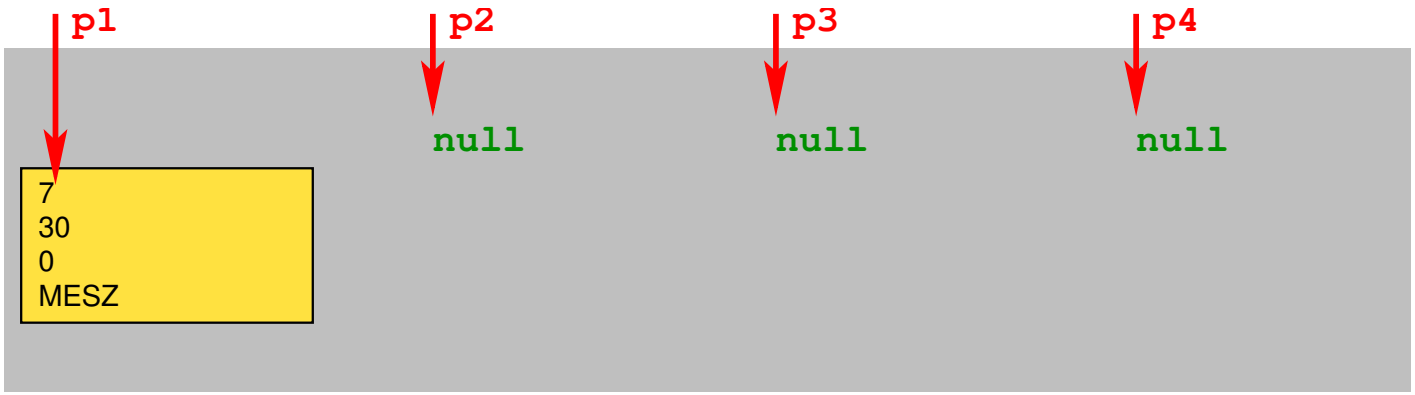
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

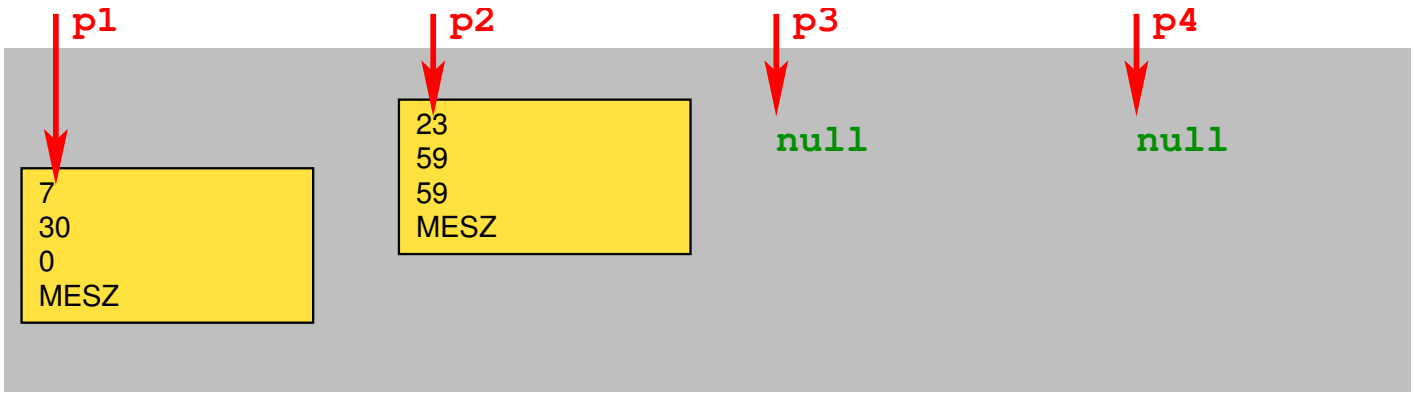
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

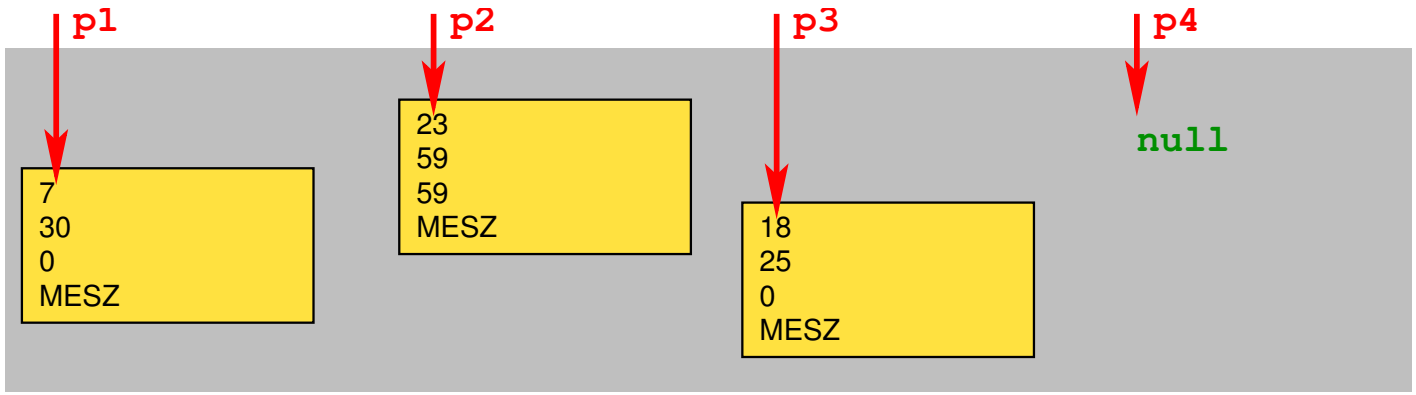
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

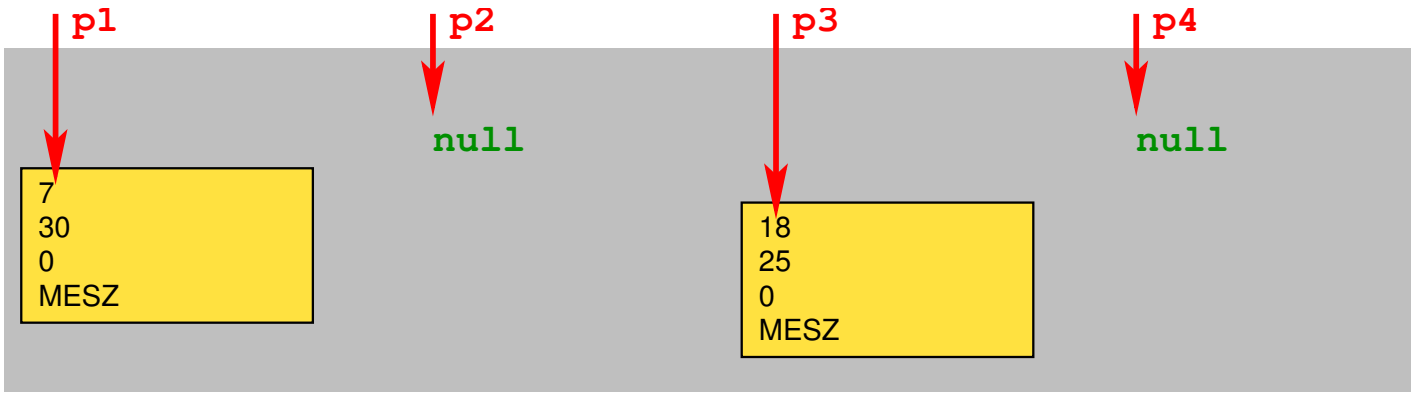
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

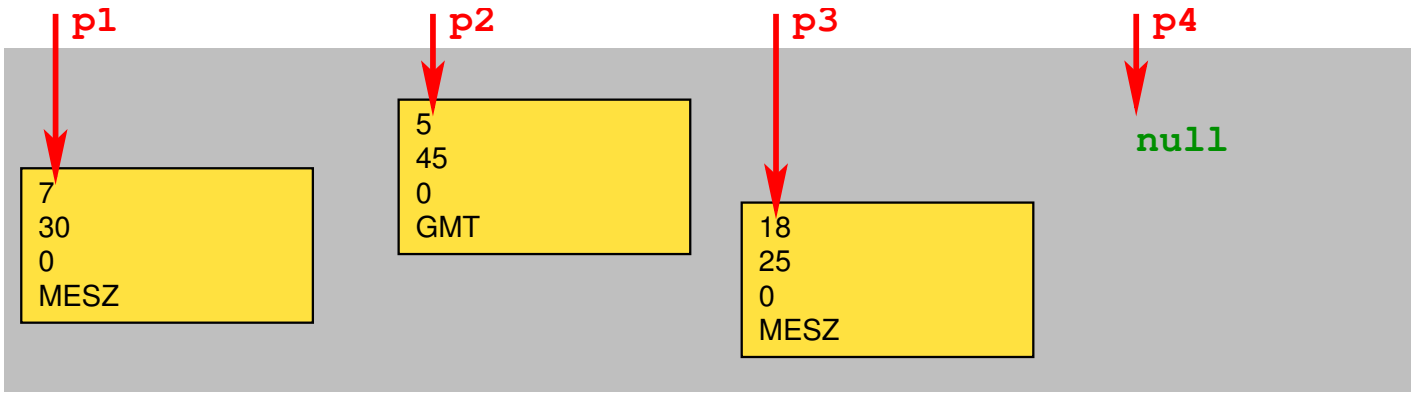
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

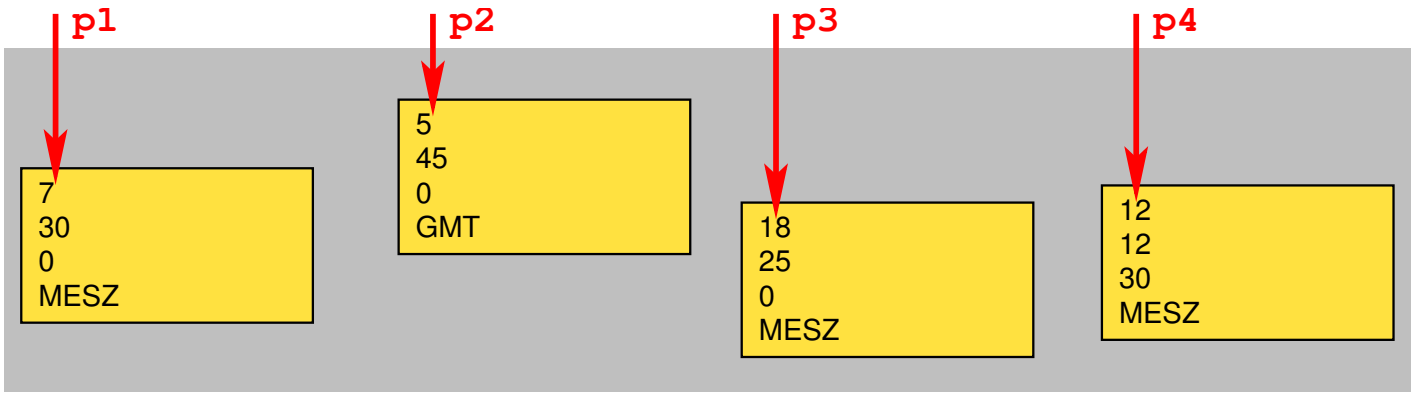
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

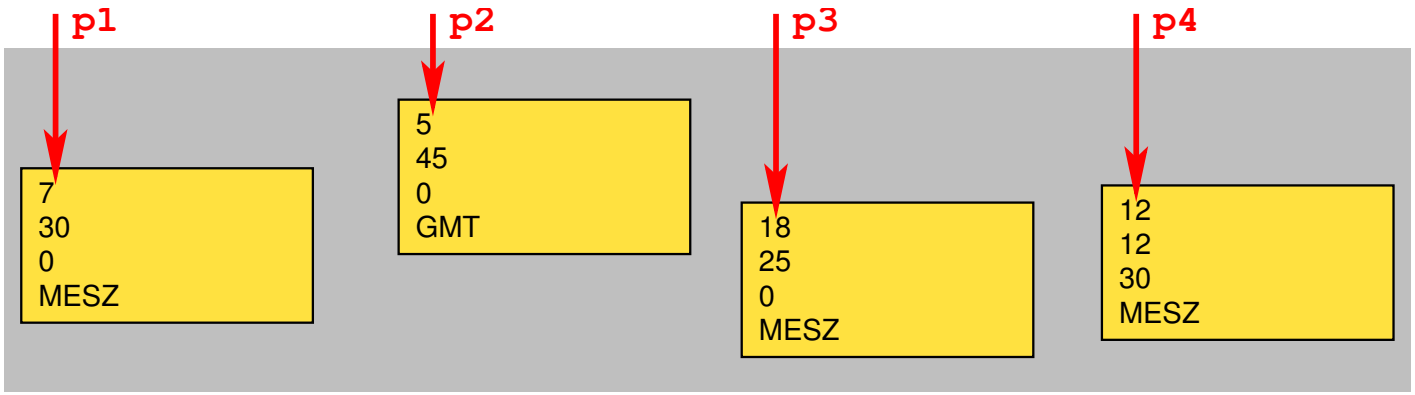
Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

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

Datensätze je nach Bedarf anlegen und löschen



Alle Datensaetze Objekte der Klasse `Uhrzeit`

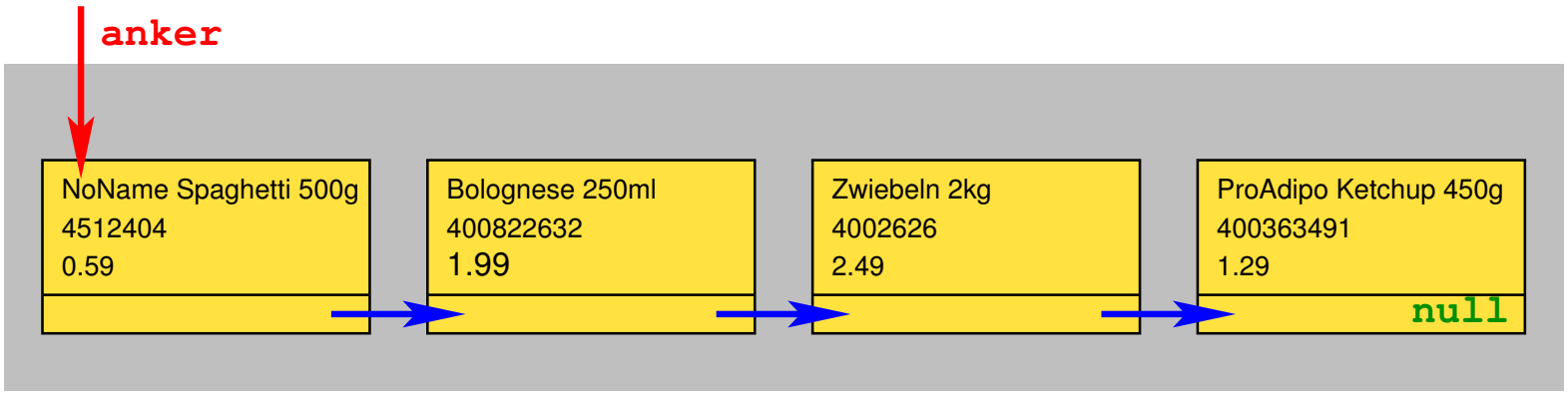
Mit mehreren Zugangspunkten, z.B. `p1`, `p2`, `p3` und `p4`, können wir entsprechend viele Datensätze während des Programmlaufs genau dann anlegen, wenn sie benötigt werden (`new`) und individuell freigeben (auf `null` setzen), sobald sie nicht mehr gebraucht werden.

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

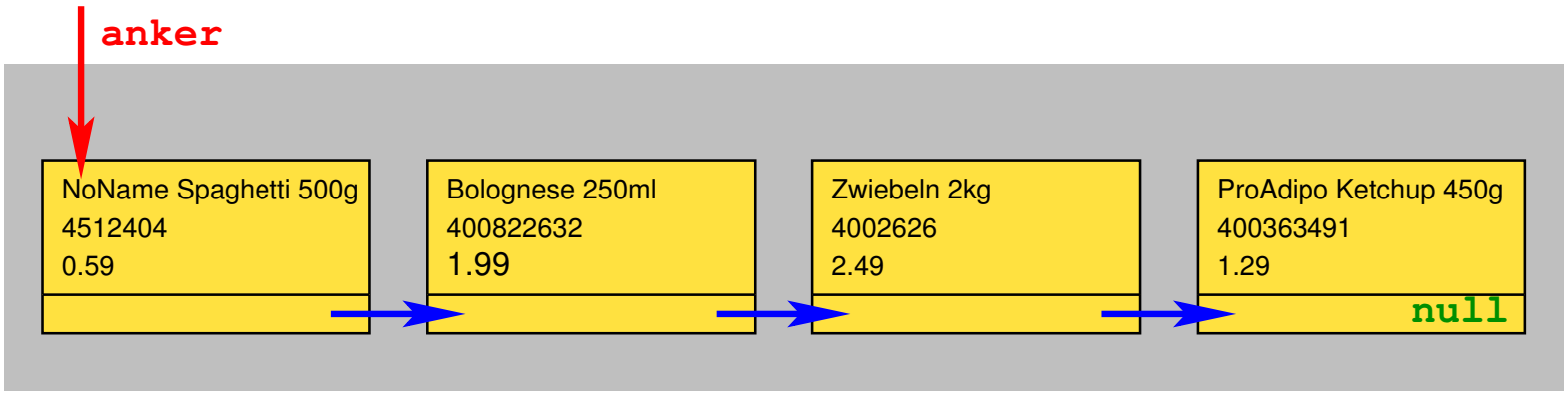
Jeder Artikel ist ein Listenelement. Artikel können hinzugefügt, gelöscht, angezeigt und zur Gesamtpreisberechnung ausgewertet werden.

Einfach verkettete Lineare Liste



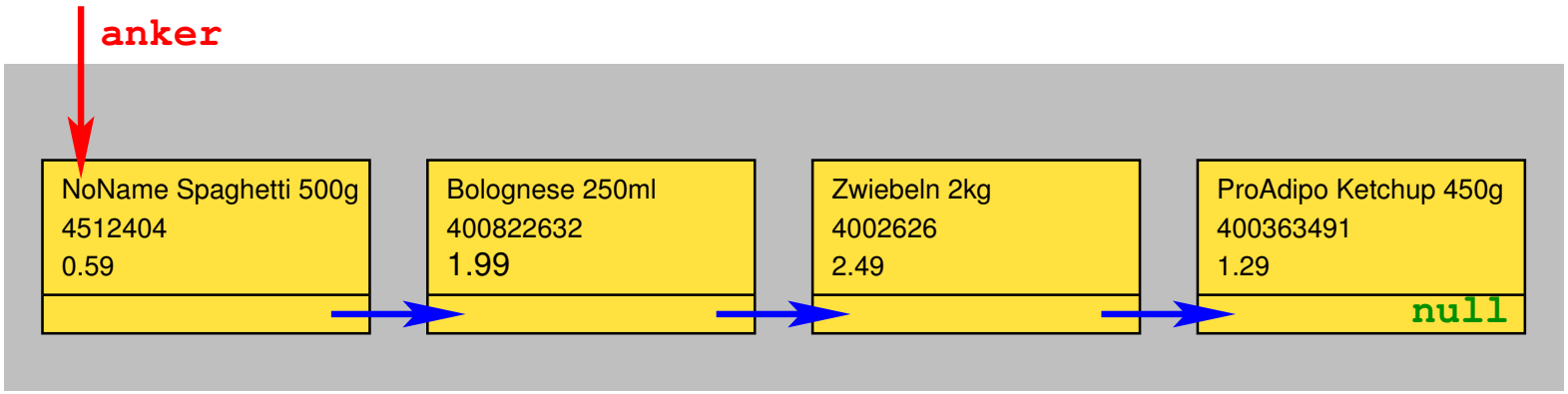
- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).

Einfach verkettete Lineare Liste



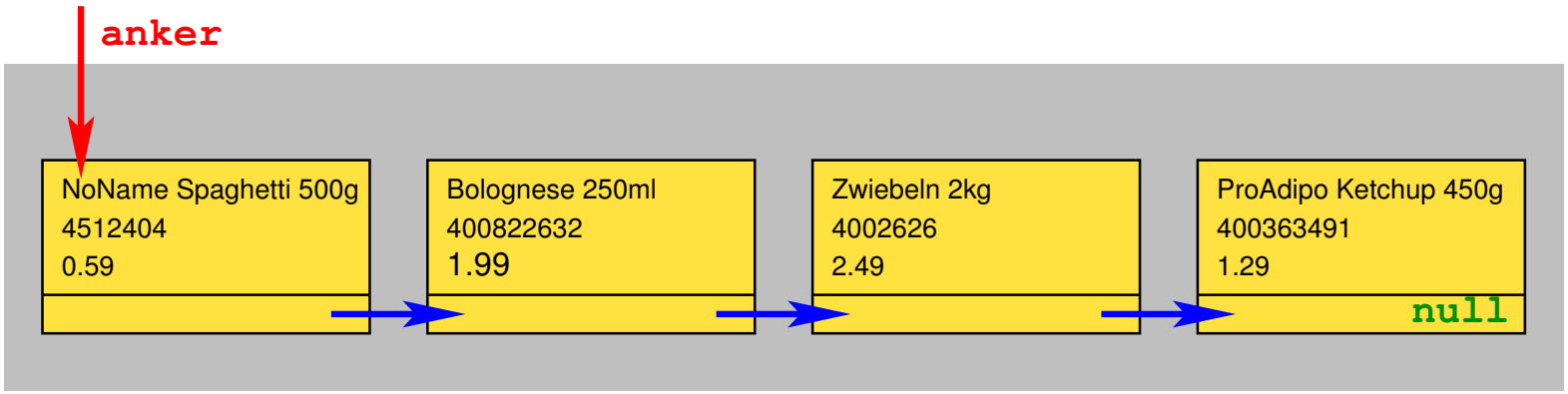
- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).
- Zusätzlich enthält jedes Listenelement eine Komponente, die *auf das nachfolgende Listenelement verweist* (seinen Zugangspunkt) bzw. beim letzten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

Einfach verkettete Lineare Liste



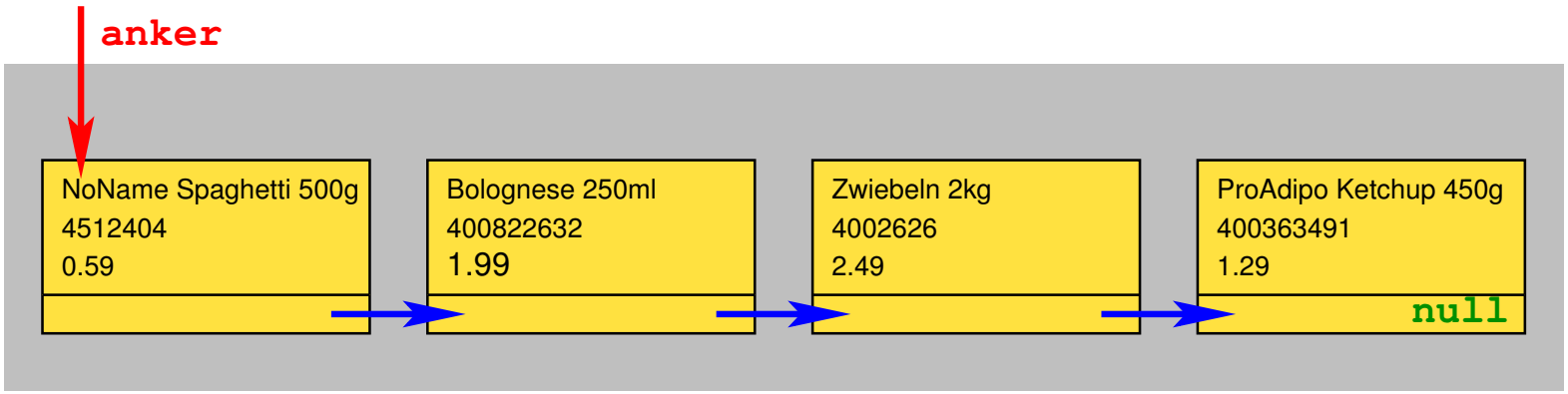
- Jedes *Listenelement* verkörpert einen *Datensatz*, hier: Artikel mit den Informationen *Produktname*, *Produktcode* und *Preis* („Informationsteil“).
- Zusätzlich enthält jedes Listenelement eine Komponente, die *auf das nachfolgende Listenelement verweist* (seinen Zugangspunkt) bzw. beim letzten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

Klasse **Artikel** beschreibt Listenelemente, Klasse **Warenkorb** führt Listenanker

```
public class Artikel {
  String produktname;           //Name eines Artikels
  long produktcode;             //z.B. EAN-Strichcode
  float preis;                  //Preis in Euro

  Artikel next;                 //Verweis auf Nachfolger

  Artikel(String n, long c, float s, Artikel x) {
    this.produktname = n;
    this.produktcode = c;
    this.preis = s;             //Konstruktor
    this.next = x;
  }
}
```

```
public class Warenkorb {
  private Artikel anker;        //Listenanker als Attribut

  Warenkorb() {                 //Konstruktor
    anker = null;               //legt leere Liste an
  }

  // ... Methoden, die Listenelemente anlegen, loeschen
}            // oder auswerten
```

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                                    400363491,
                                        1.29f,
                                        null);
```

Liste ist leer, Listenanker **anker** hat Wert **null**

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                    400363491,
                    1.29f,
                    null);
```

neues Listenelement als Objekt mittels **new** anlegen, dafür Speicherplatz ausfassen

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                    400363491,
                    1.29f,
                    null);
```

Produktname (Zeichenkette) wird in den neuen Datensatz eingetragen

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                    400363491,
                    1.29f,
                    null);
```

Produktcode (Ganzzahl vom Typ **`long`**) wird in den neuen Datensatz eingetragen

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                                        400363491,
                                            1.29f,
                                            null);
```

Preis (Gleitkommazahl vom Typ **`float`**) wird in den neuen Datensatz eingetragen

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                    400363491,
                    1.29f,
                    null);
```

Endekennung `null` wird in den neuen Datensatz eingetragen.

Erstes Element in leere Liste einfügen


```
anker = new Artikel("ProAdipo Ketchup 450g",
                    400363491,
                    1.29f,
                    null);
```

Listenanker **anker** als Zugangspunkt zuweisen. Einfügen damit abgeschlossen.

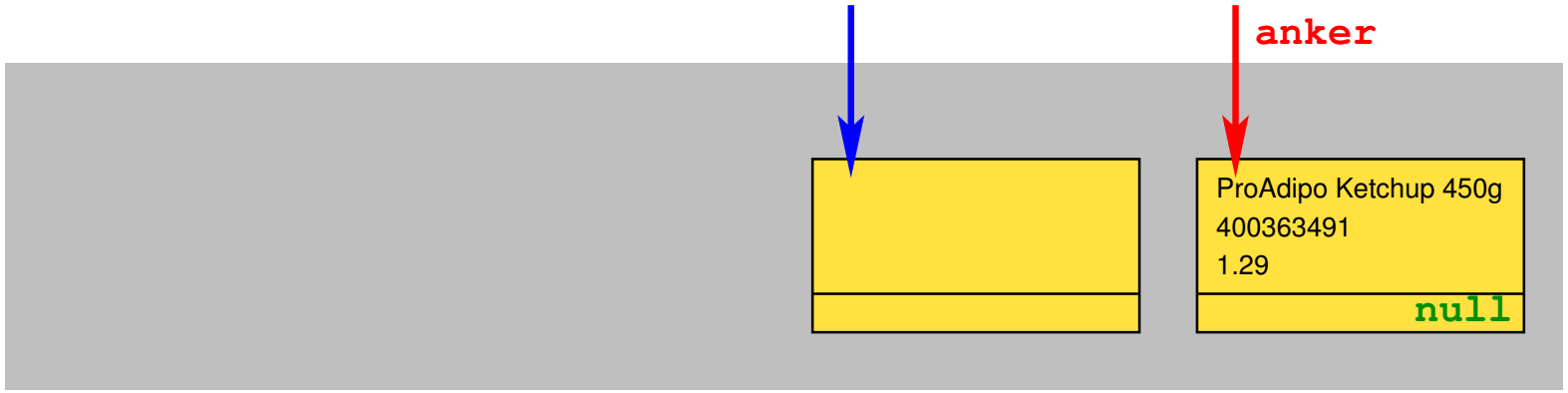
Neues Element am Listenanfang einfügen

Neues Element am Listenanfang einfügen


```
anker = new Artikel("Zwiebeln 2kg",
                    4002626,
                      2.49f,
                     anker);
```

Ähnliche Anweisungsfolge wie beim Einfügen des ersten Elements

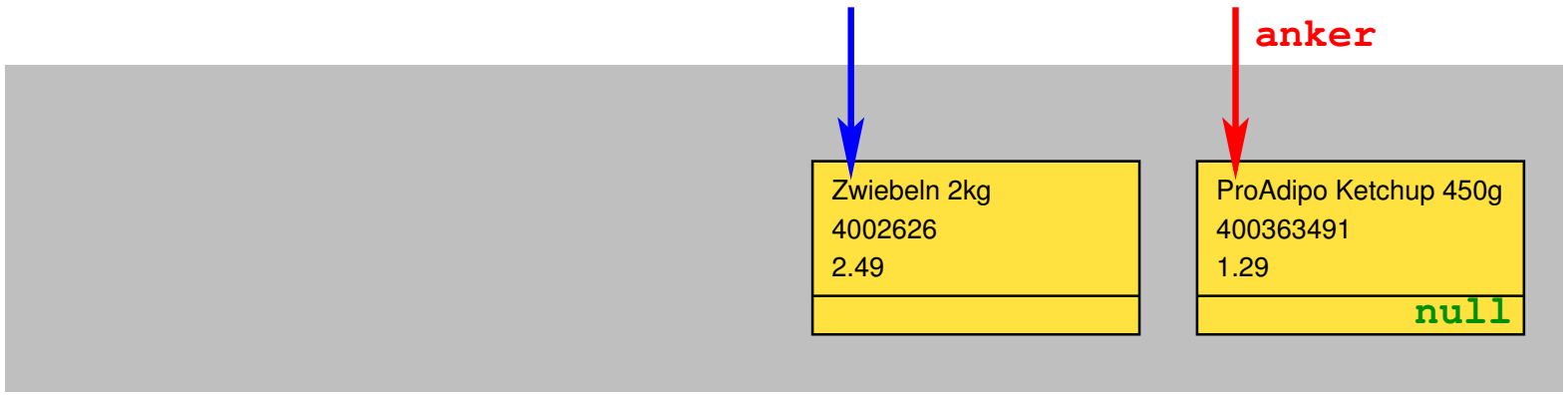
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Zwiebeln 2kg",
                    4002626,
                    2.49f,
                    anker);
```

neues Listenelement als Objekt mittels **new** anlegen, dafür Speicherplatz resevieren

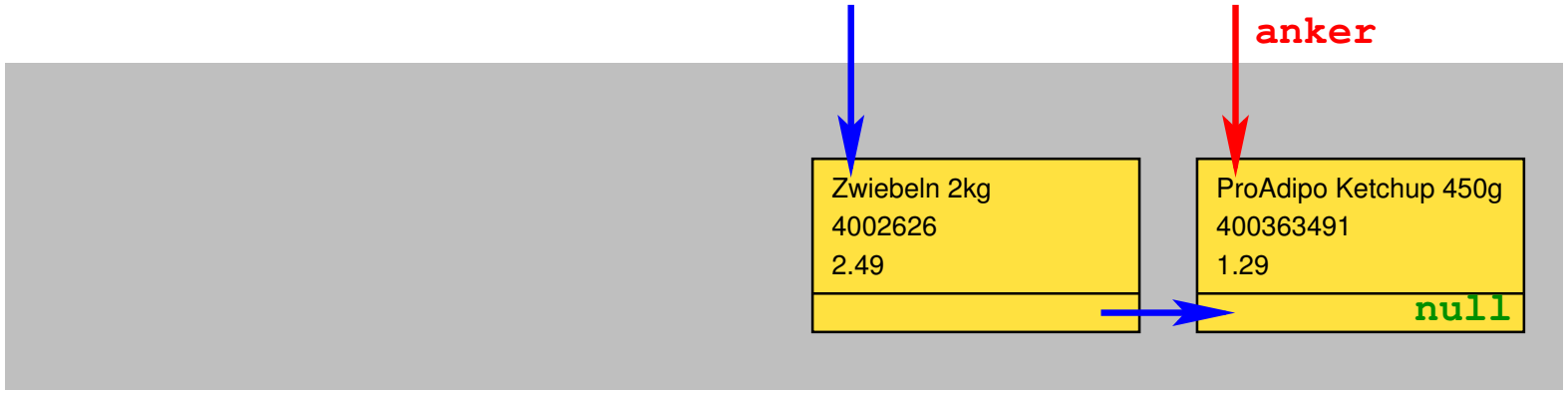
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Zwiebeln 2kg",
                    4002626,
                      2.49f,
                    anker);
```

Informationsteil des Datensatzes wird befüllt

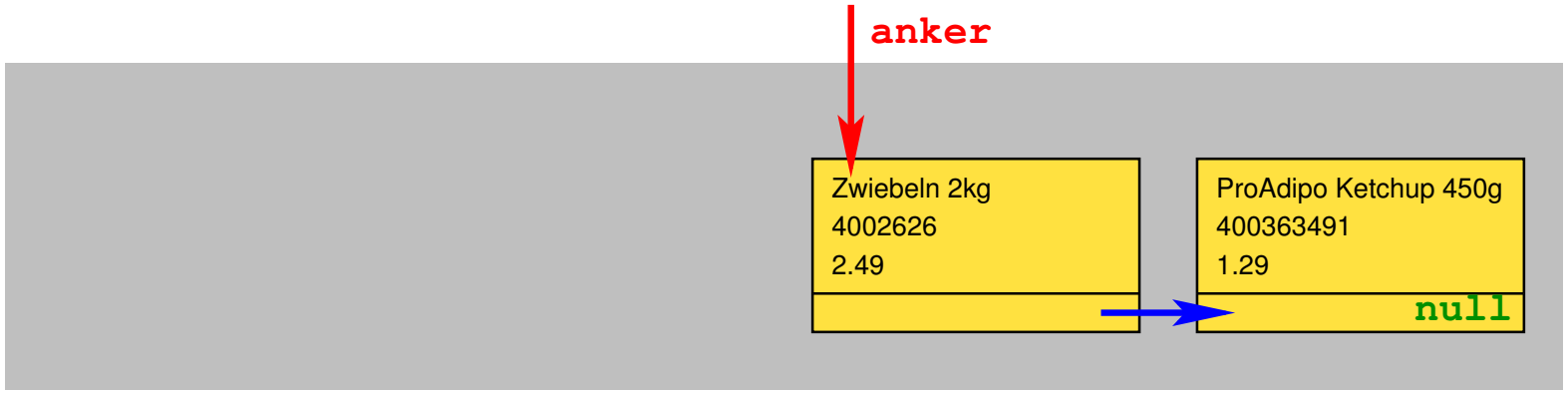
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Zwiebeln 2kg",
                    4002626,
                      2.49f,
                     anker);
```

Listenanker **anker** als Zugangspunkt auf das nunmehr nachfolgende Listenelement eintragen

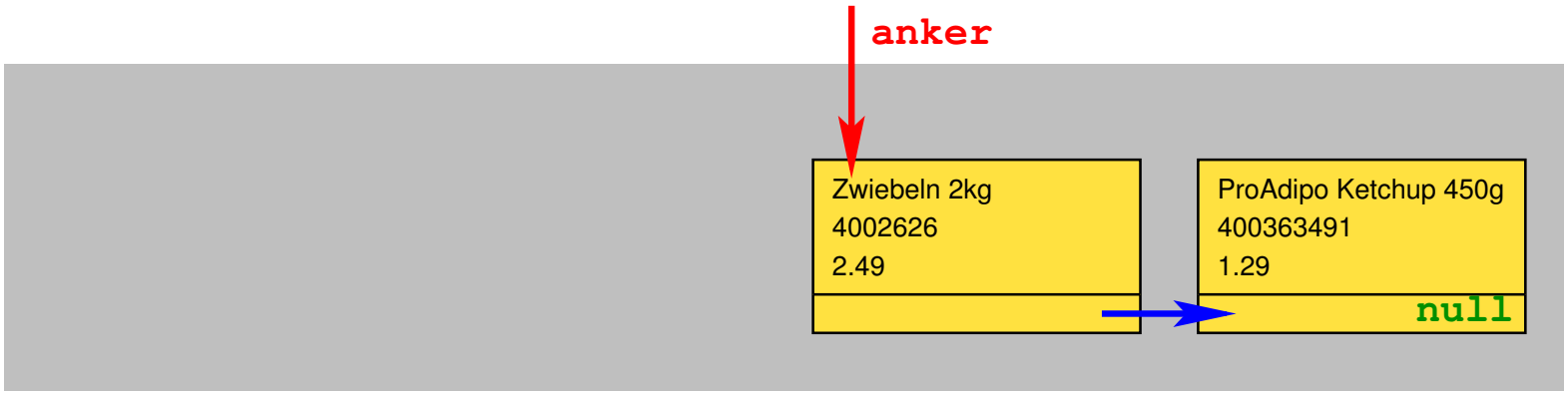
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Zwiebeln 2kg",
                    4002626,
                    2.49f,
                    anker);
```

Listenanker **anker** aktualisieren mit Zugangspunkt auf das nun führende Listenelement. Fertig.

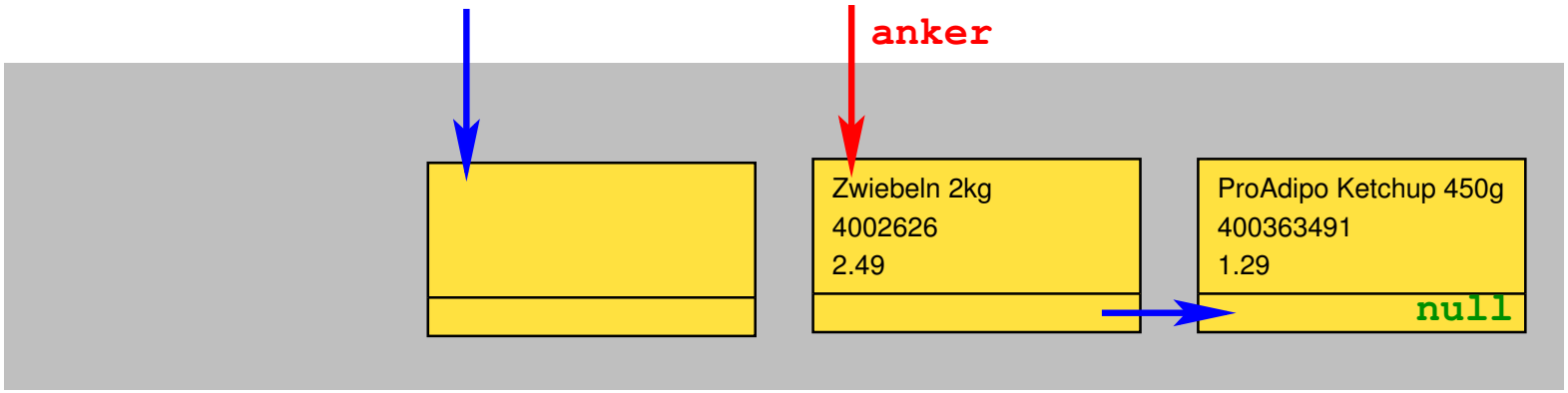
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Bolognese 250ml",
                    400822632,
                    1.99f,
                    anker);
```

Gleiche Anweisungsfolge wie beim Einfügen des vorherigen Elements

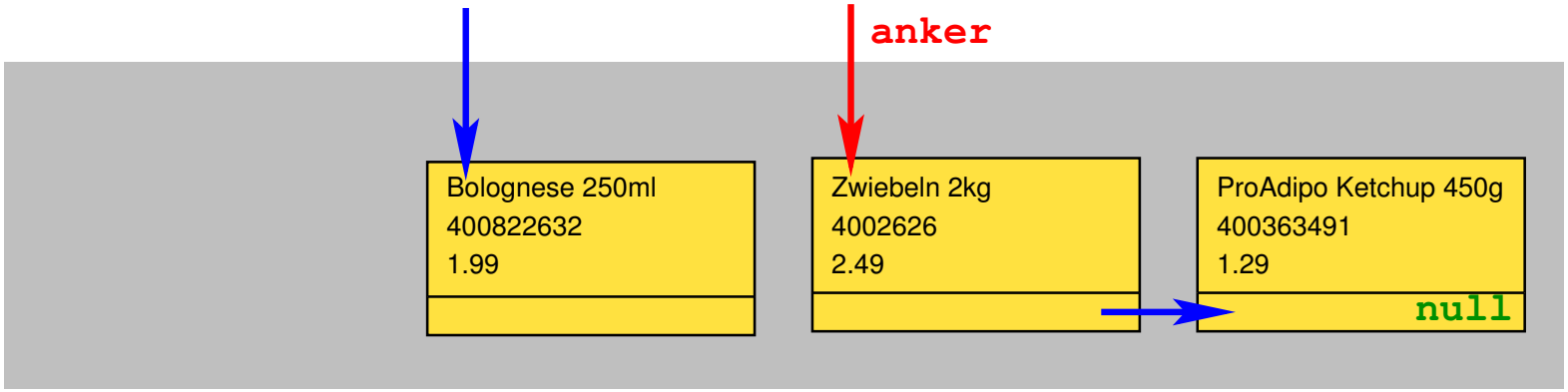
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Bolognese 250ml",
                    400822632,
                    1.99f,
                    anker);
```

neues Listenelement als Objekt mittels `new` anlegen, dafür Speicherplatz resevieren

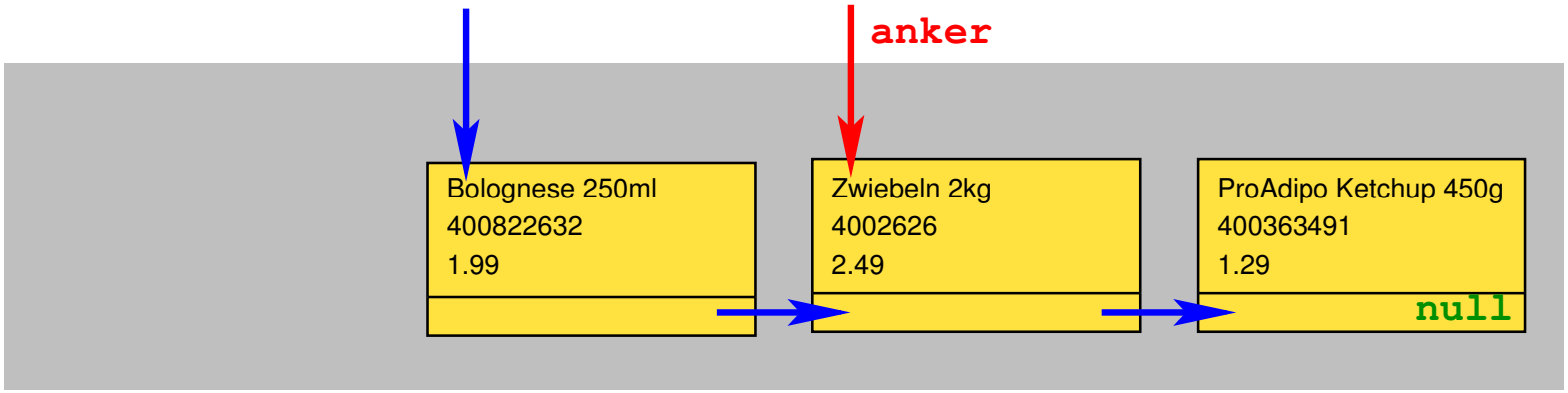
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Bolognese 250ml",
                    400822632,
                      1.99f,
                    anker);
```

Informationsteil des Datensatzes wird befüllt

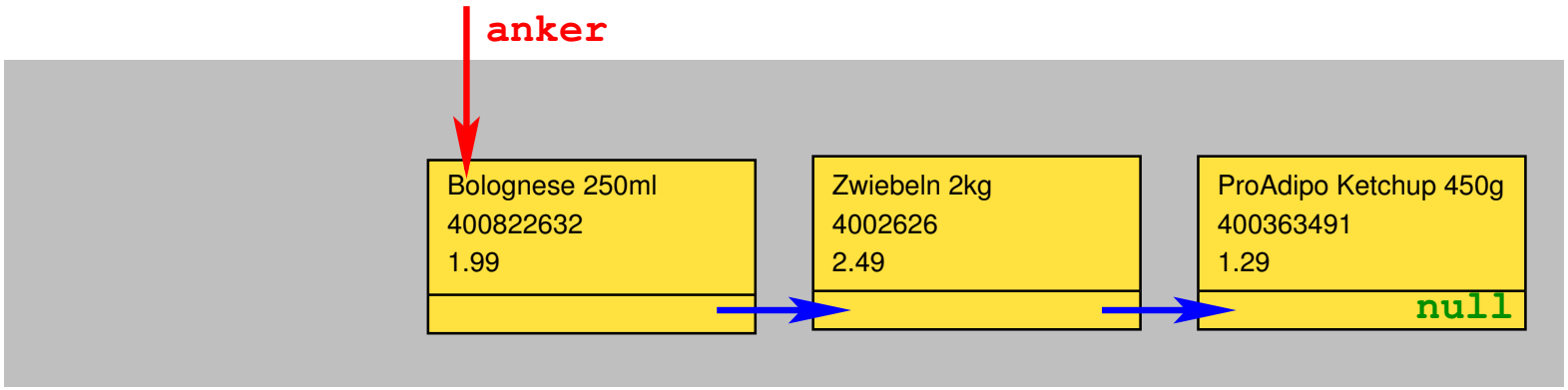
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Bolognese 250ml",
                    400822632,
                    1.99f,
                    anker);
```

Listenanker **anker** als Zugangspunkt auf das nunmehr nachfolgende Listenelement eintragen

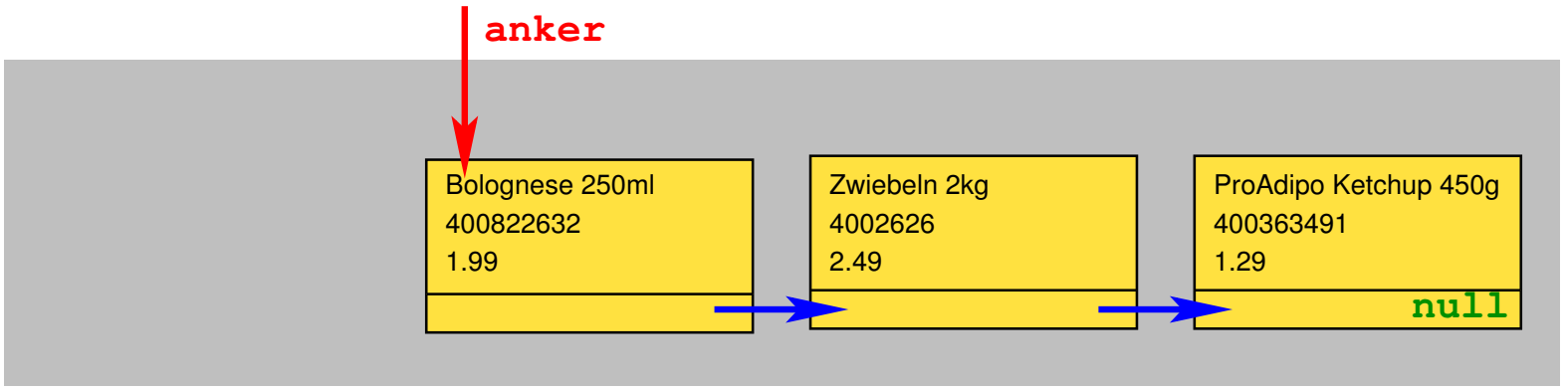
Neues Element am Listenanfang einfügen




```
anker = new Artikel("Bolognese 250ml",
                    400822632,
                    1.99f,
                    anker);
```

Listenanker **anker** aktualisieren mit Zugangspunkt auf das nun führende Listenelement. Fertig.

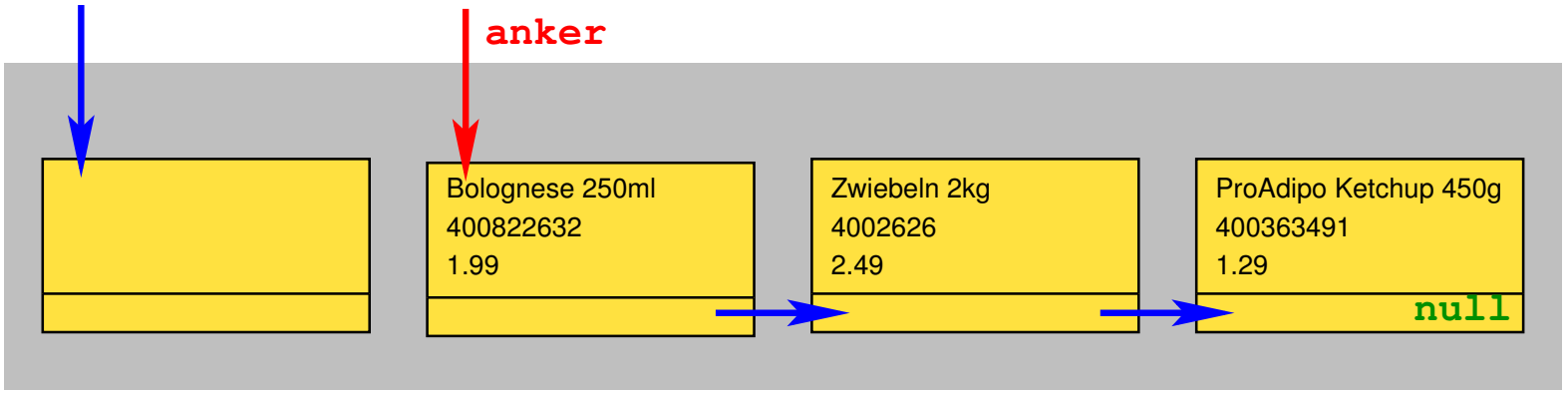
Neues Element am Listenanfang einfügen




```
anker = new Artikel("NoName Spaghetti 500g",
                                        4512404,
                                          0.59f,
                                         anker);
```

Gleiche Anweisungsfolge wie beim Einfügen des vorherigen Elements

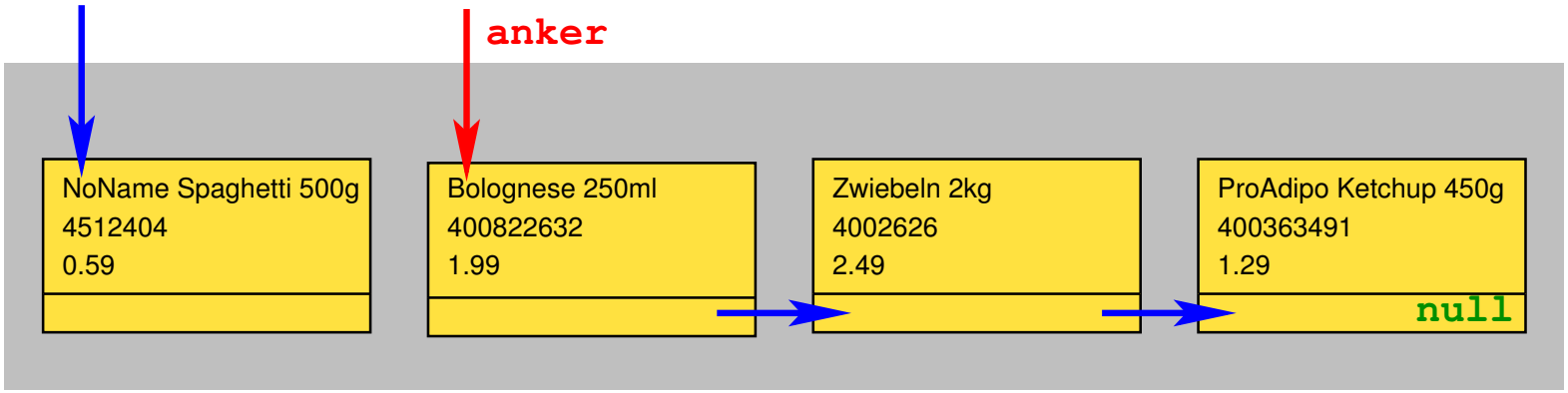
Neues Element am Listenanfang einfügen




```
anker = new Artikel("NoName Spaghetti 500g",
                                        4512404,
                                          0.59f,
                                         anker);
```

neues Listenelement als Objekt mittels **new** anlegen, dafür Speicherplatz resevieren

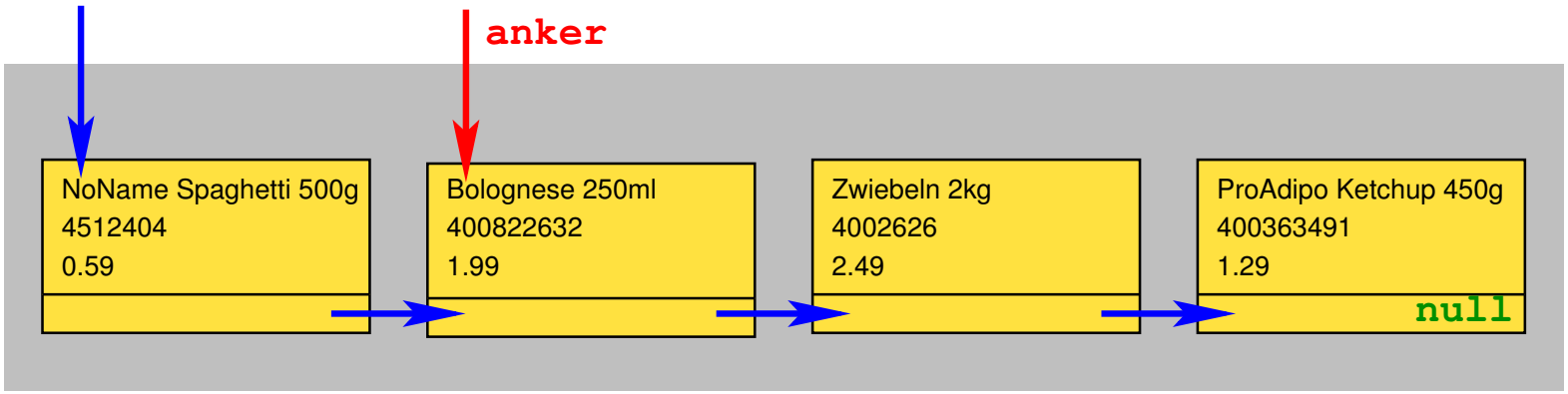
Neues Element am Listenanfang einfügen




```
anker = new Artikel("NoName Spaghetti 500g",
                                    4512404,
                                      0.59f,
                                    anker);
```

Informationsteil des Datensatzes wird befüllt

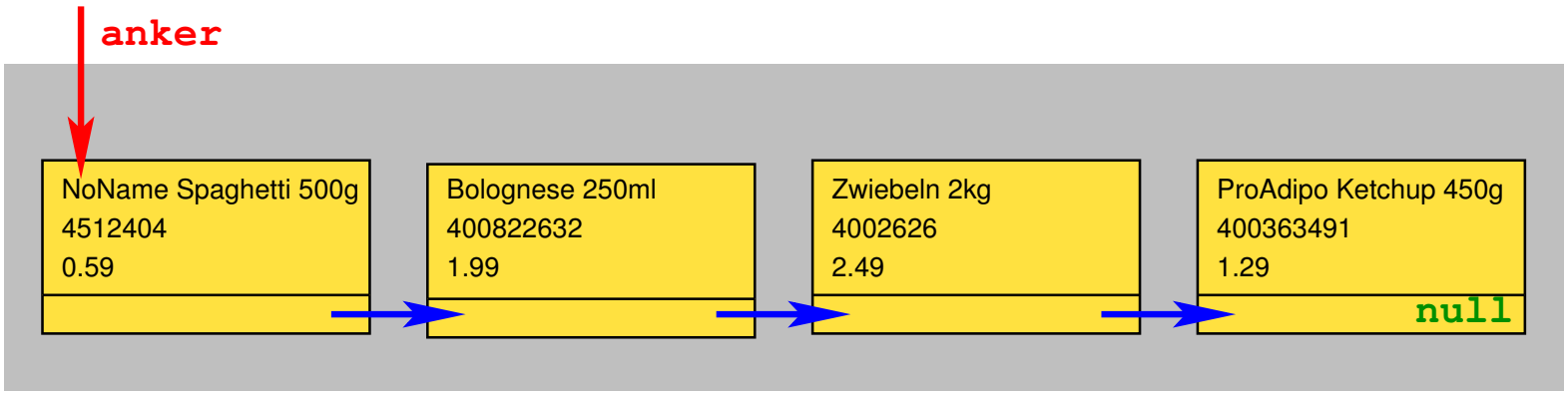
Neues Element am Listenanfang einfügen




```
anker = new Artikel("NoName Spaghetti 500g",
                                        4512404,
                                          0.59f,
                                         anker);
```

Listenanker **anker** als Zugangspunkt auf das nunmehr nachfolgende Listenelement eintragen

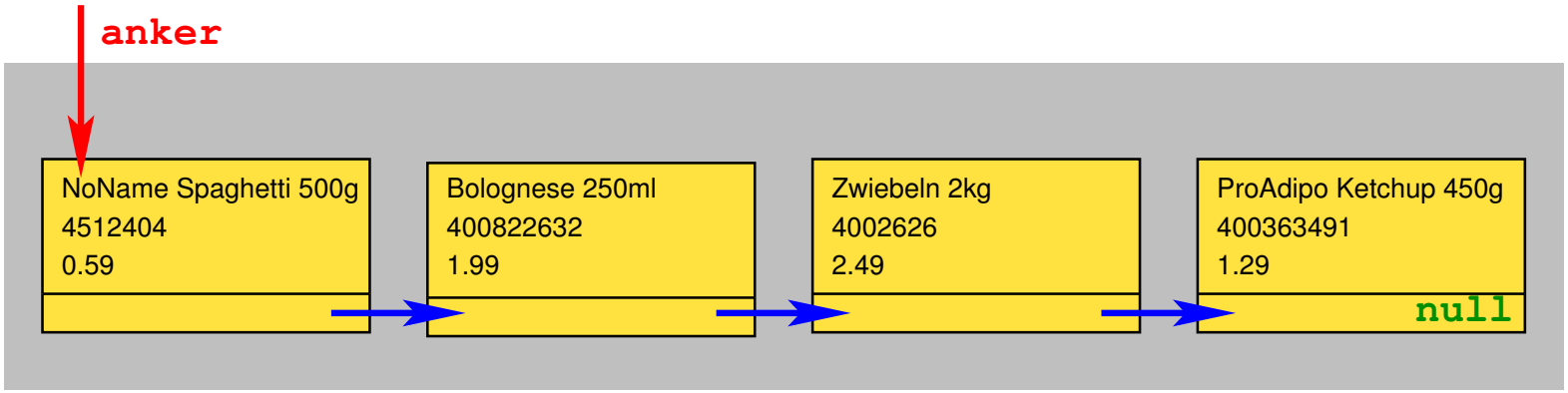
Neues Element am Listenanfang einfügen




```
anker = new Artikel("NoName Spaghetti 500g",
                                        4512404,
                                          0.59f,
                                         anker);
```

Listenanker **anker** aktualisieren mit Zugangspunkt auf das nun führende Listenelement.

Neues Element am Listenanfang einfügen

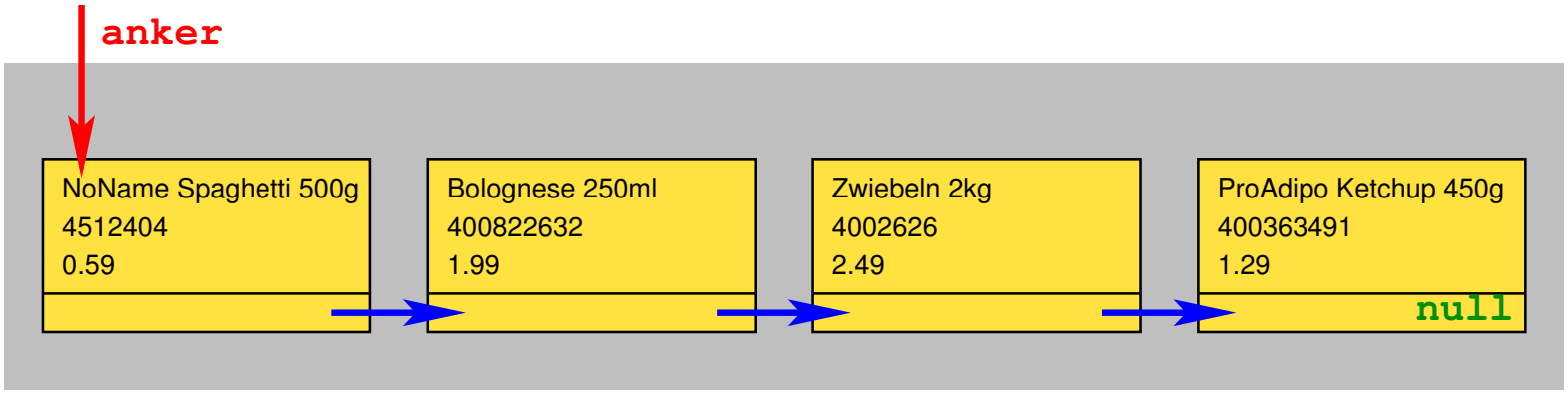



```
anker = new Artikel("NoName Spaghetti 500g",
                                        4512404,
                                          0.59f,
                                         anker);
```

Einfügen von vier Elementen jeweils am Listenanfang abgeschlossen.

Anzahl Elemente in der Liste bestimmen

Anzahl Elemente in der Liste bestimmen

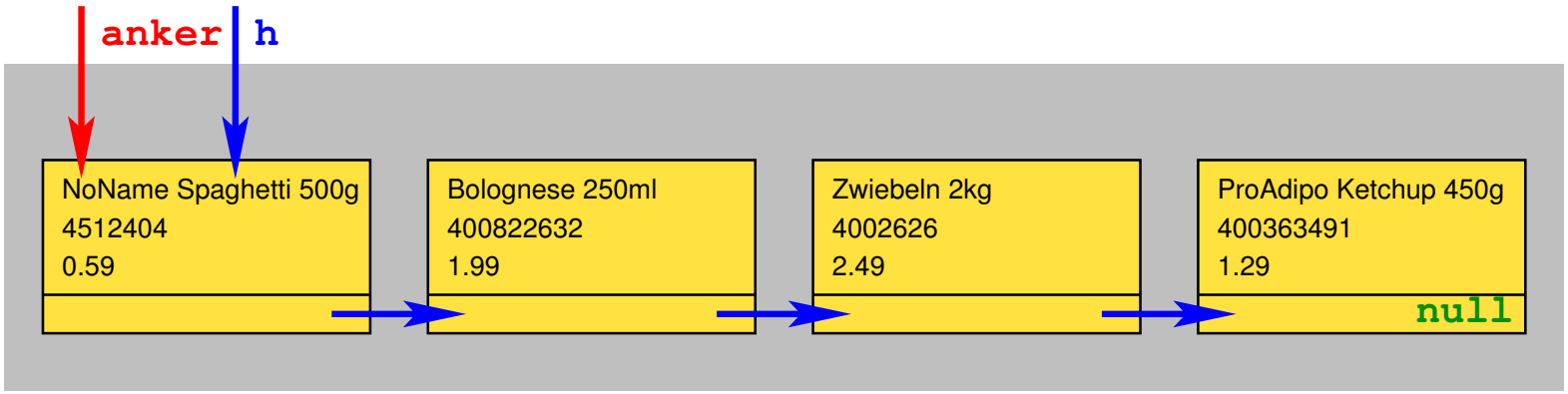



```
int i = 0;
Artikel h = anker;

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

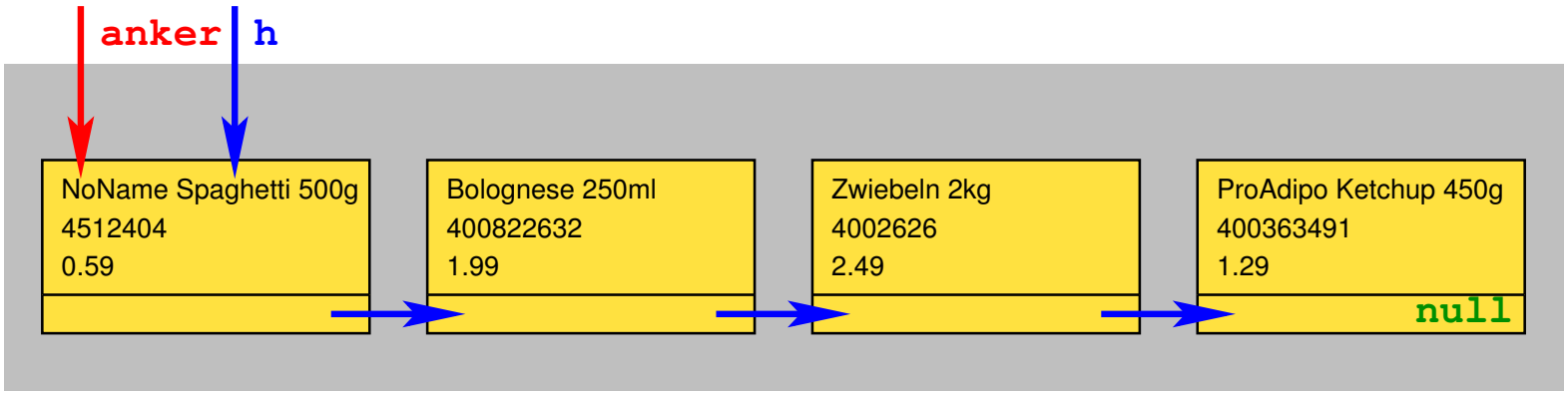



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 0

Hilfszugangspunkt `h` anlegen und am Listenanfang platzieren

Anzahl Elemente in der Liste bestimmen

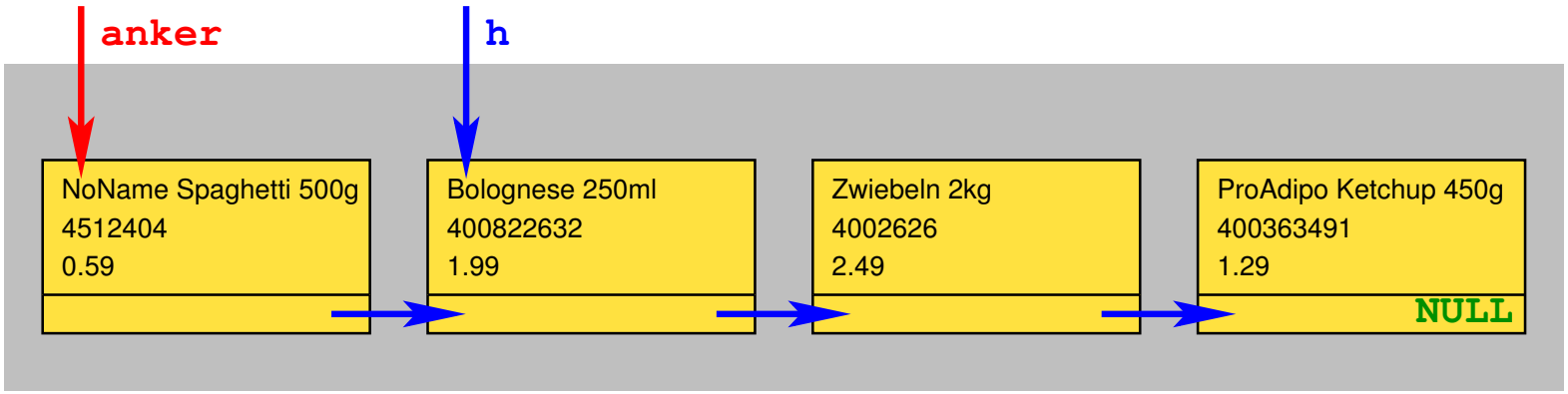



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 1

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

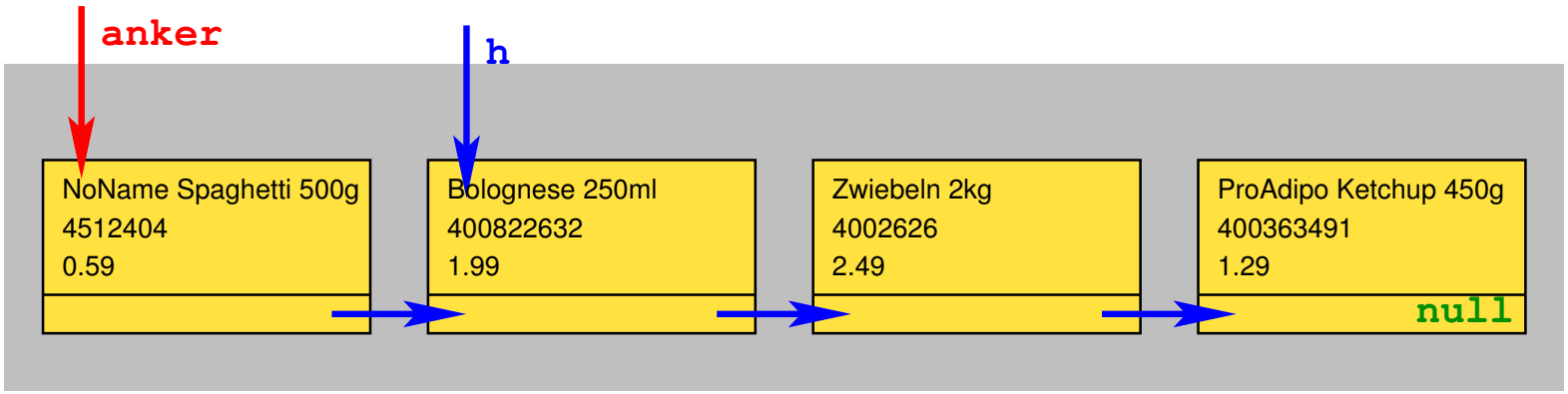



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 1

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen

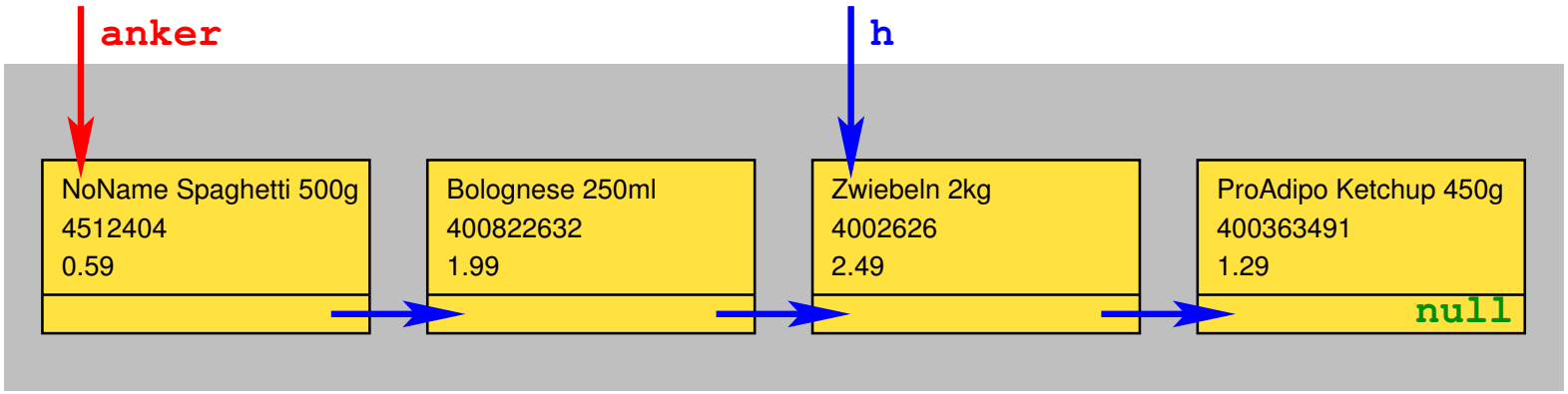



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 2

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

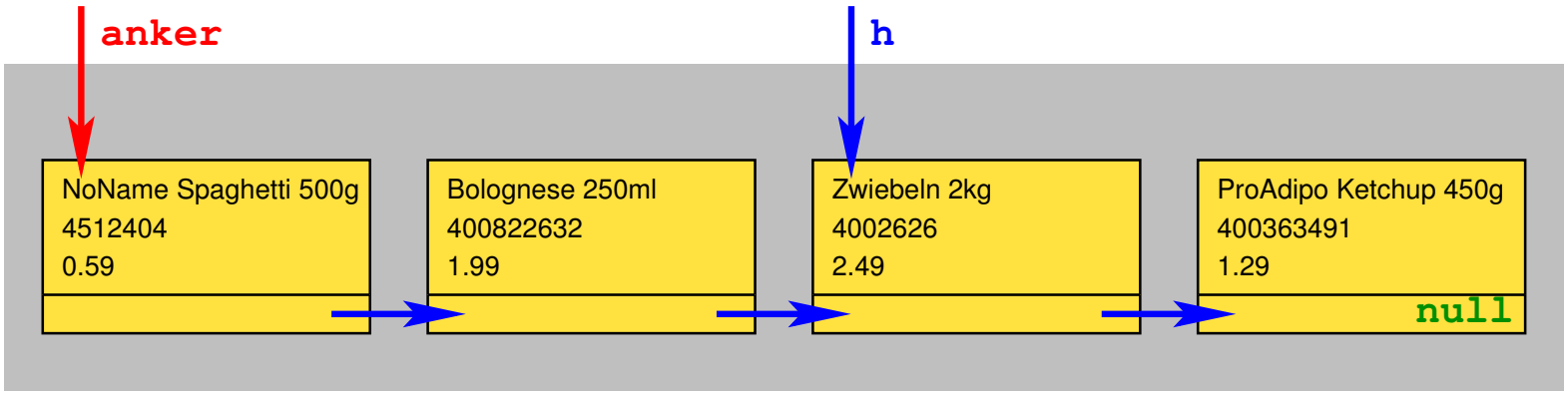



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 2

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen

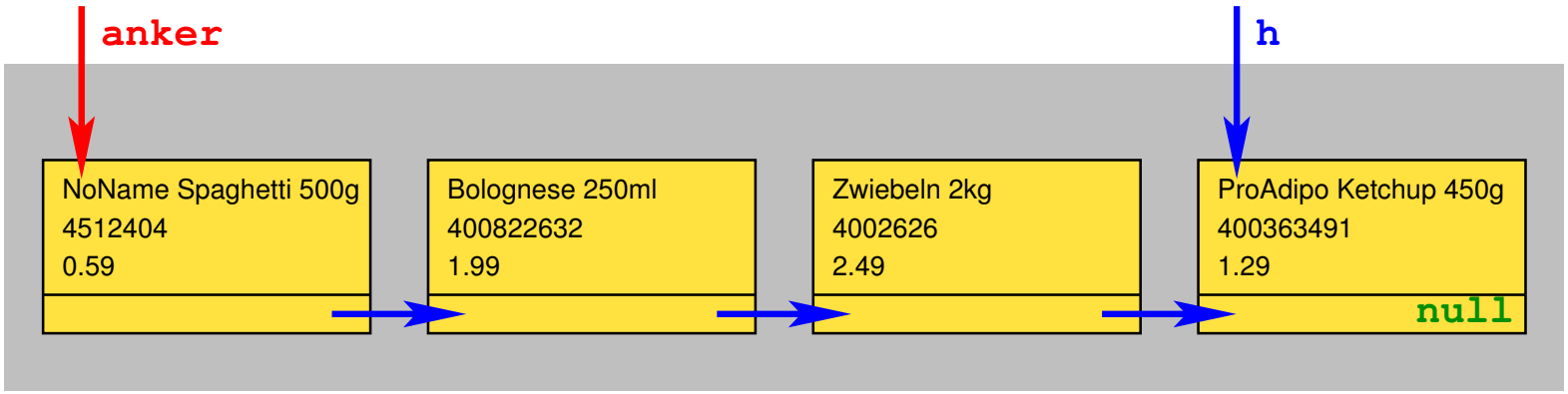



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 3

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

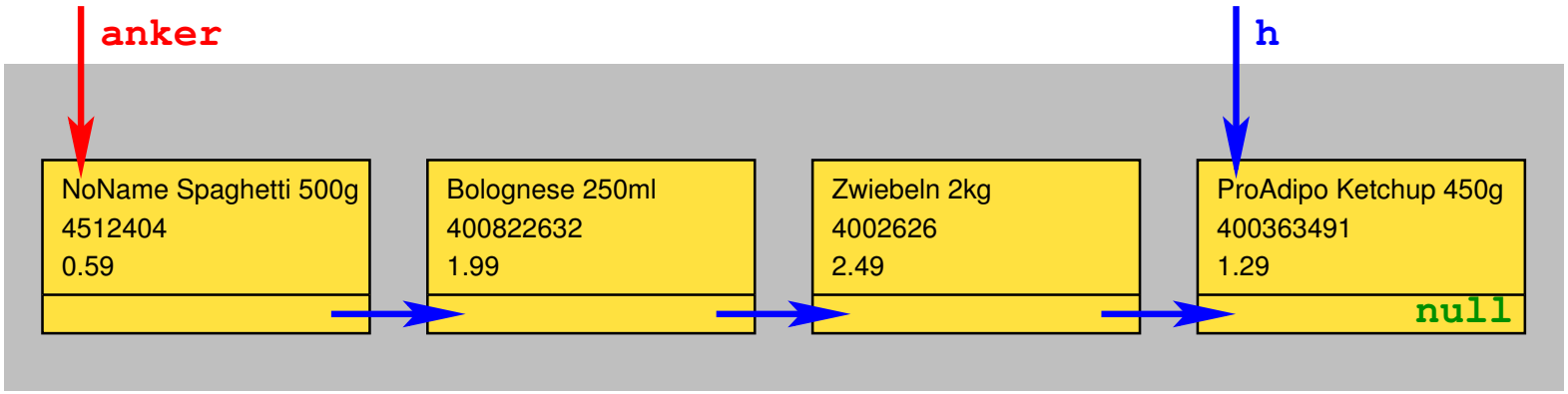



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 3

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen

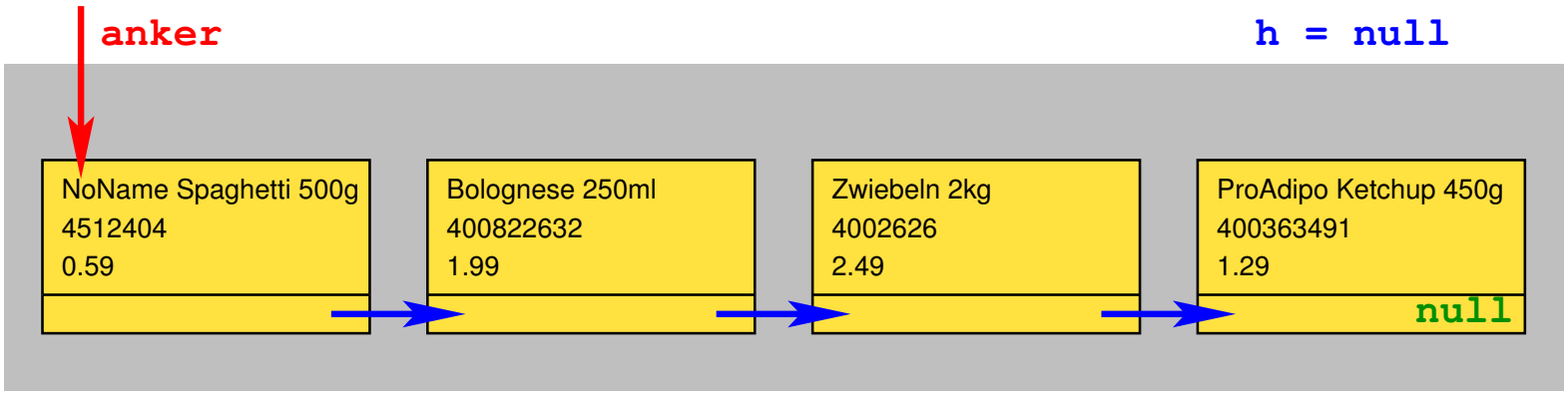



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 4

Zähler `i` inkrementieren

Anzahl Elemente in der Liste bestimmen

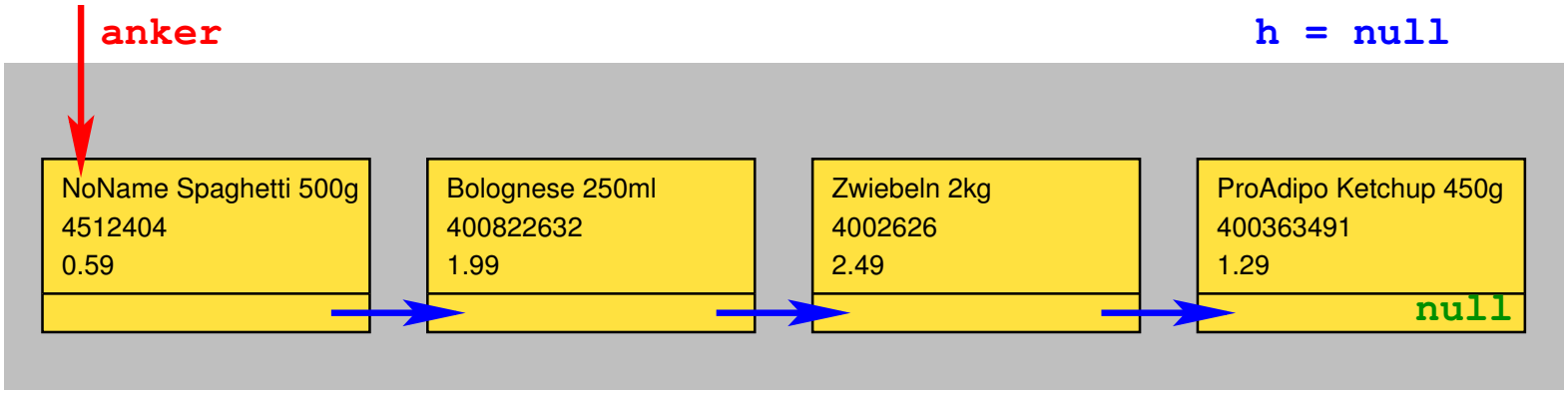



```
int i = 0;
Artikel h = anker;

while (h != null)
{
  i++;
  h = h.next;
}
```

i: 4

zum nächsten Listenelement laufen

Anzahl Elemente in der Liste bestimmen



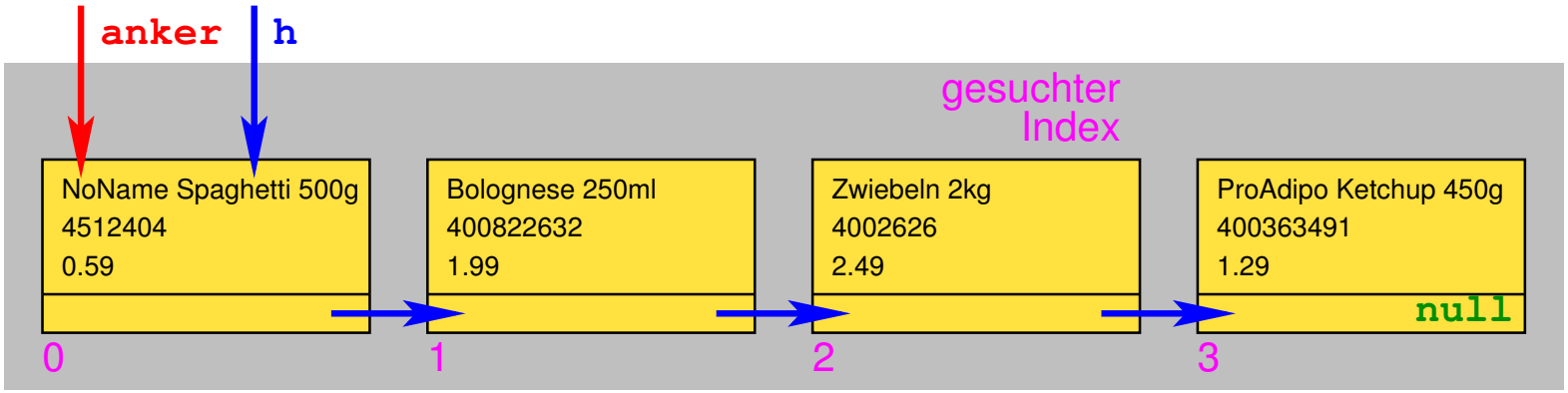

```
int i = 0;
Artikel h = anker;

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

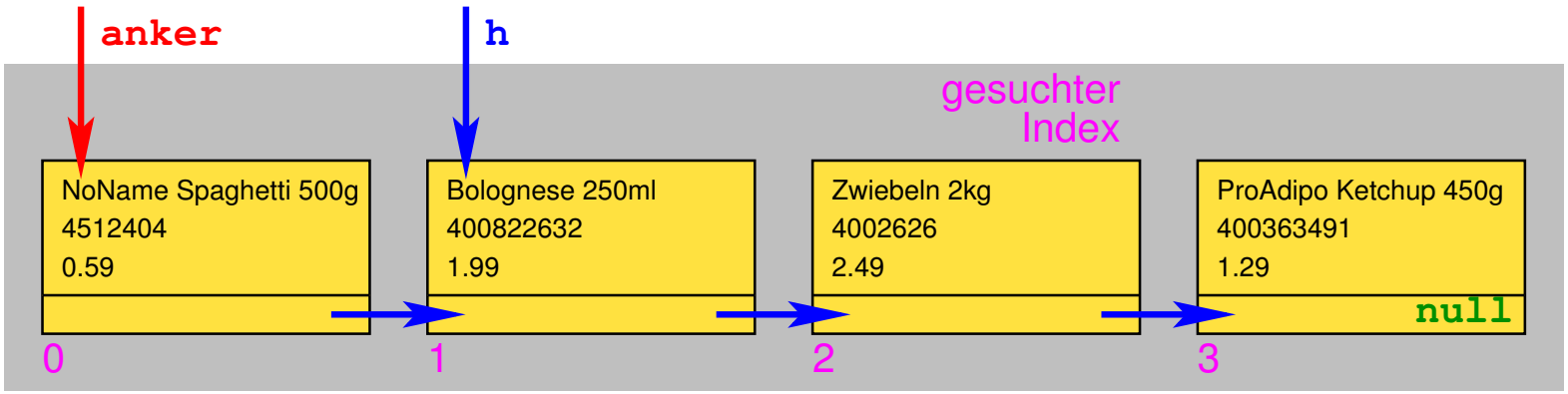
Listenelement suchen und auslesen („get“)




```
int i = 0;
Artikel h = anchor;

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

Listenelemente von 0 beginnend fortlaufend durchnummeriert *(Index)*

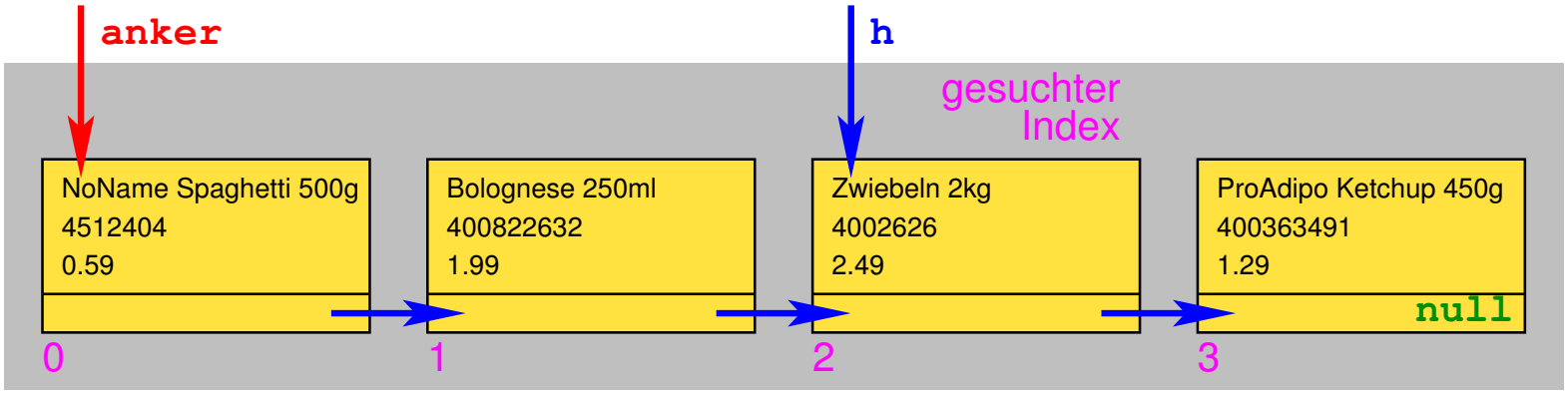
Listenelement suchen und auslesen („get“)




```
int i = 0;
Artikel h = anchor;

if ((index < 0) || (index >= anzahlArtikel())) { /* 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“)

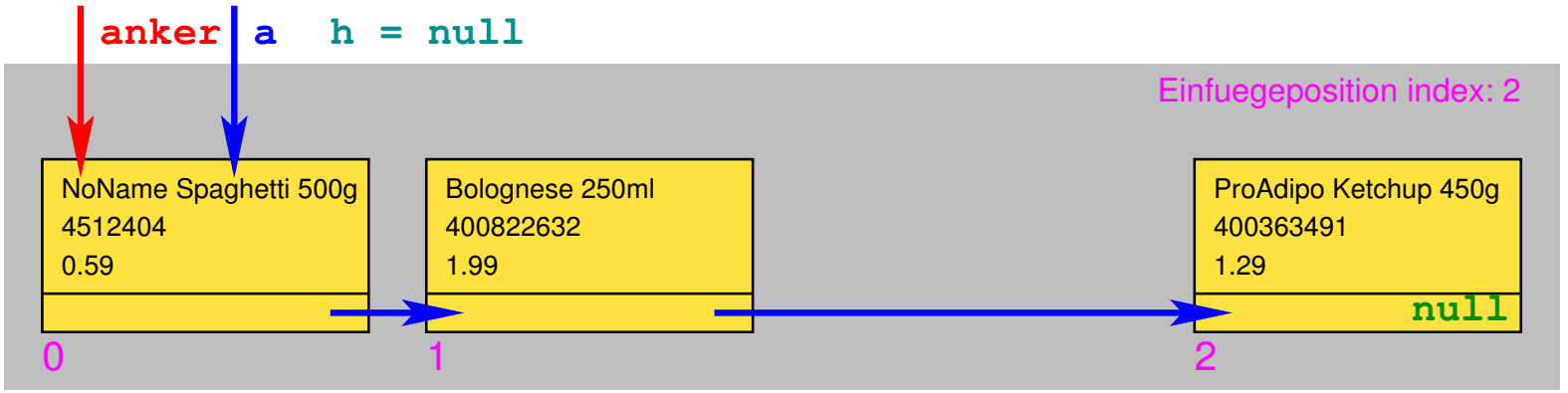



```
int i = 0;
Artikel h = anchor;

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

Bei Erreichen des gesuchten Index Daten des Informationsteils bereitstellen

Listenelement an gegebener Indexposition einfügen

Listenelement an gegebener Indexposition einfügen



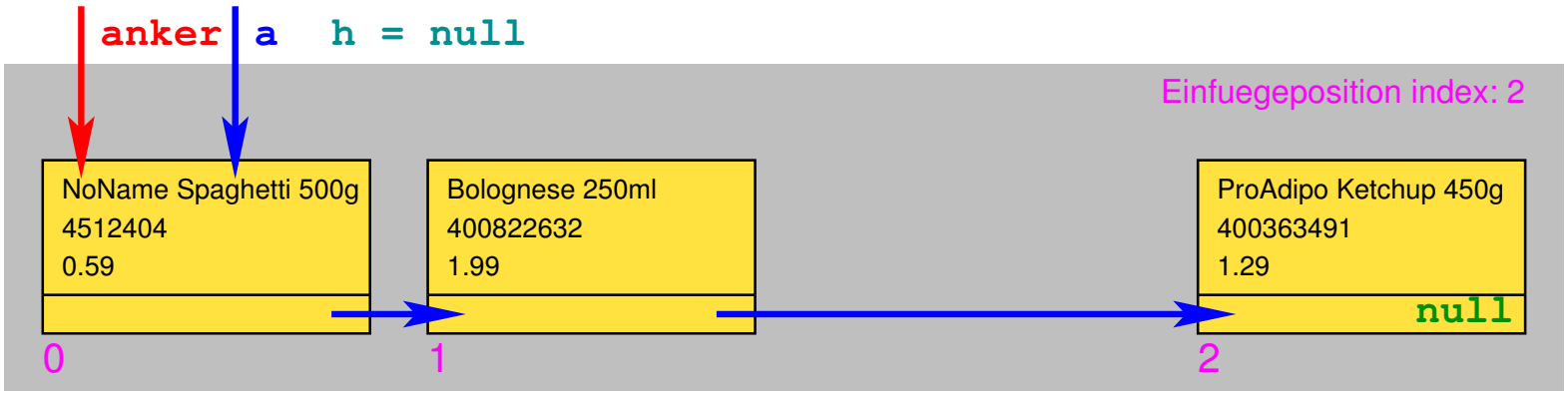

```
int i = 0;
Artikel a = anchor;
Artikel h = null;
```

```
if (index < 0) || (index >= anzahlArtikel())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
while (i < index - 1)
{
  i++;
  a = a.next;
}
h = new Artikel("Joghurt 200g", 44444, 0.39f, a.next);
a.next = h;
```

Hilfszugangspunkt **a** anlegen und auf Listenanfang setzen,

Hilfszugangspunkt **h** für neues Listenelement anlegen und auf **null** setzen

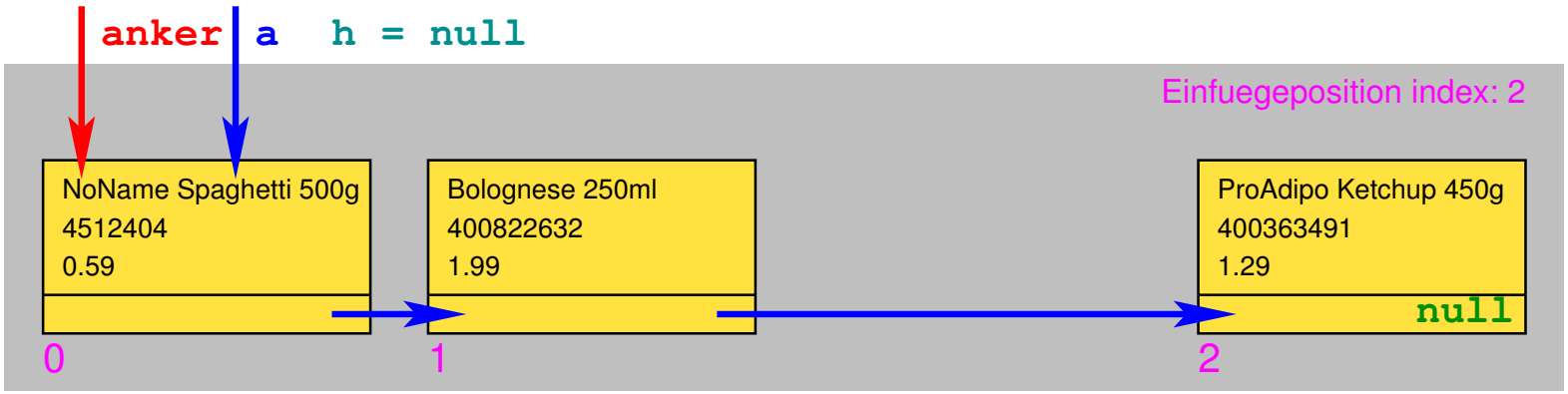
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
Artikel a = anchor;
Artikel h = null;
if (index < 0) || (index >= anzahlArtikel())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
while (i < index - 1)
{
  i++;
  a = a.next;
}
h = new Artikel("Joghurt 200g", 44444, 0.39f, a.next);
a.next = h;
```

Plausibilitätscheck: Indexposition zum Einfügen zulässig?

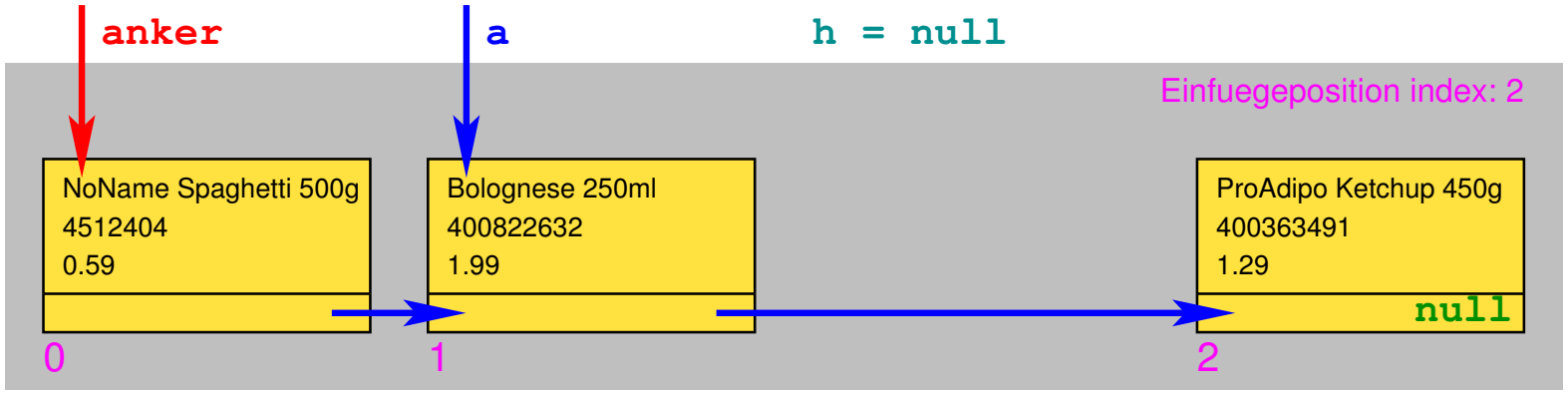
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
Artikel a = anchor;
Artikel h = null;

if (index < 0) || (index >= anzahlArtikel())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
while (i < index - 1)
{
  i++;
  a = a.next;
}
h = new Artikel("Joghurt 200g", 44444, 0.39f, a.next);
a.next = h;
```

Einfügen an Indexposition 0 entspricht Einfügen am Listenanfang

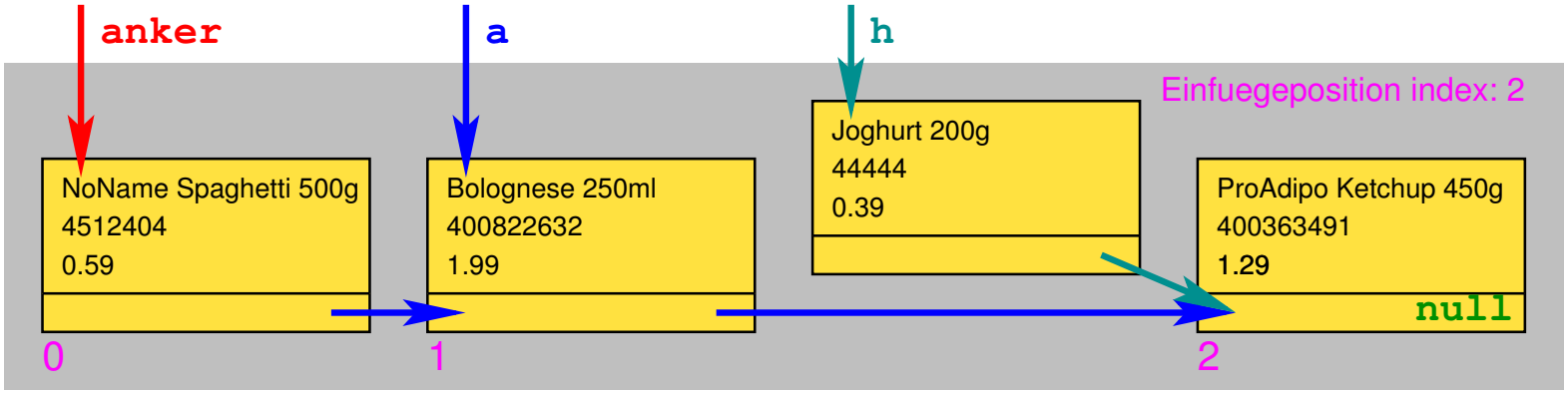
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
Artikel a = anchor;
Artikel h = null;

if (index < 0) || (index >= anzahlArtikel())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
while (i < index - 1)
{
  i++;
  a = a.next;
}
h = new Artikel("Joghurt 200g", 44444, 0.39f, a.next);
a.next = h;
```

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

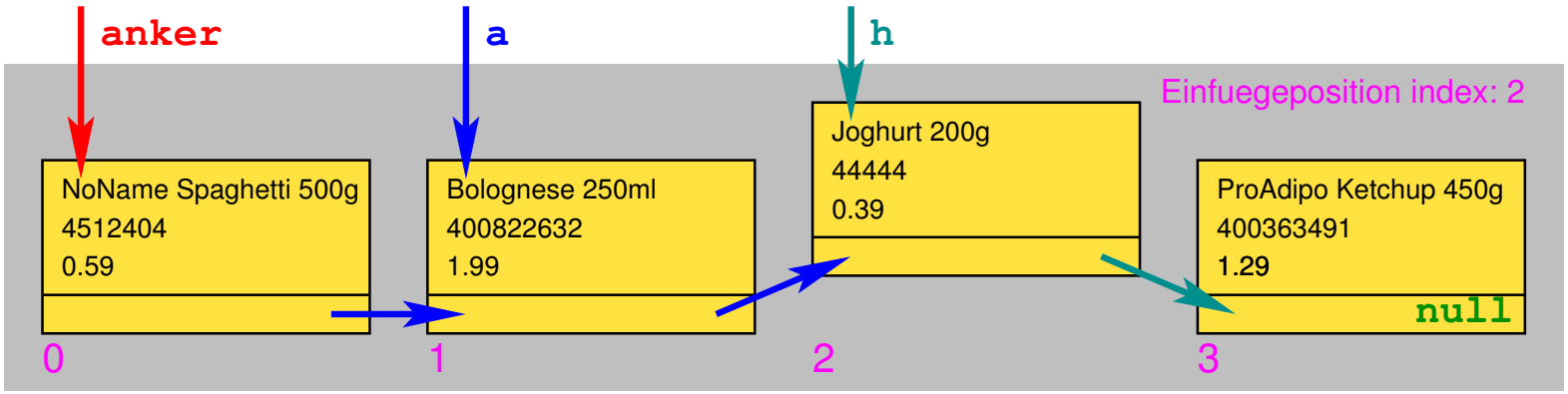
Listenelement an gegebener Indexposition einfügen




```
int i = 0;
Artikel a = anchor;
Artikel h = null;

if (index < 0) || (index >= anzahlArtikel())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
while (i < index - 1)
{
  i++;
  a = a.next;
}
h = new Artikel("Joghurt 200g", 44444, 0.39f, a.next);
a.next = h;
```

neues Listenelement als Objekt anlegen und befüllen einschl. Verkettung zum Nachfolger

Listenelement an gegebener Indexposition einfügen

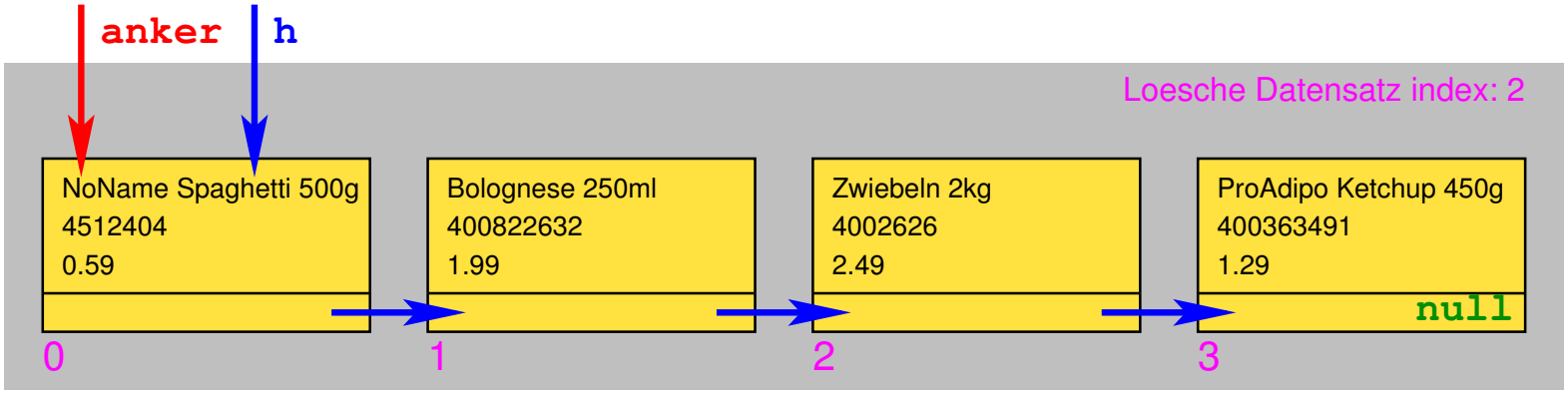



```
int i = 0;
Artikel a = anchor;
Artikel h = null;

if (index < 0) || (index >= anzahlArtikel())) { /* Fehler */ }
if (index == 0) { /* Einfuegen am Listenanfang */ }
while (i < index - 1)
{
  i++;
  a = a.next;
}
h = new Artikel("Joghurt 200g", 44444, 0.39f, a.next);
a.next = h;
```

Neues Listenelement an Vorgänger anketten, Einfügen fertig

Listenelement an gegebener Indexposition löschen

Listenelement an gegebener Indexposition löschen

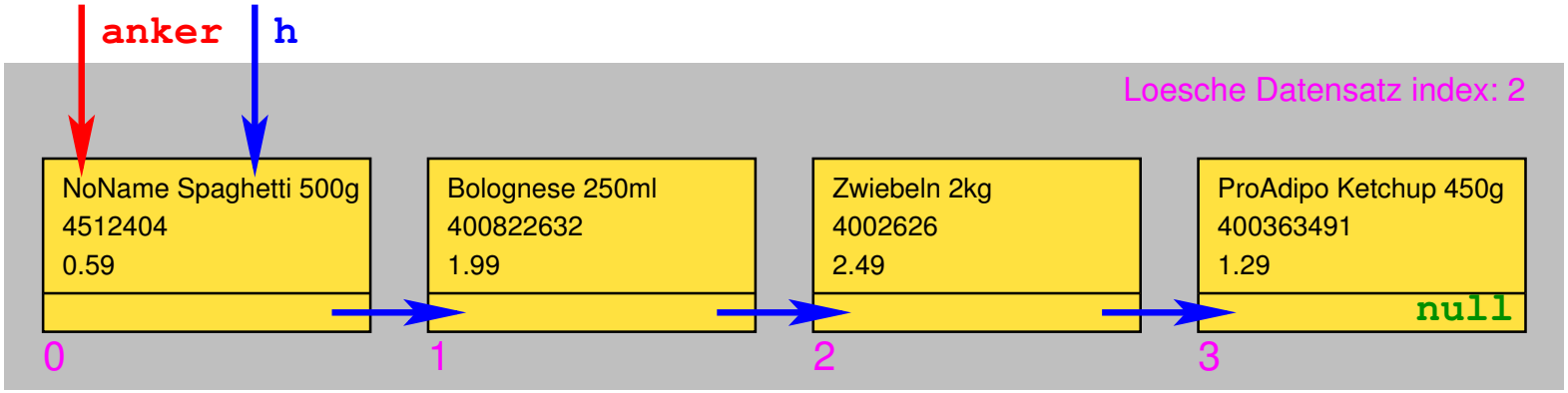



```
int i = 0;
Artikel h = anker;
```

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

Hilfszugangspunkt **h** anlegen und auf Listenanfang setzen

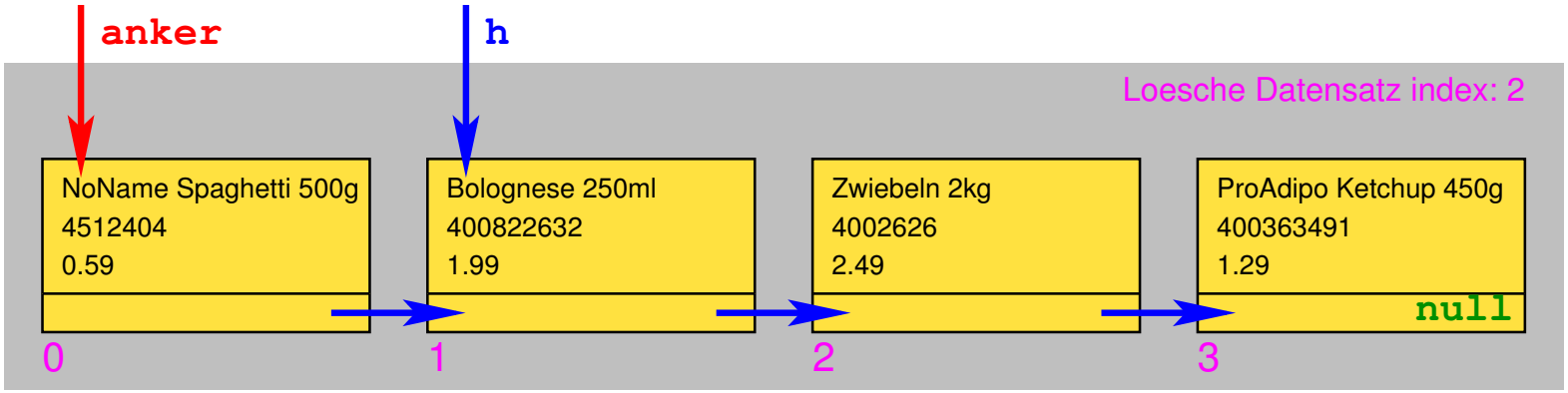
Listenelement an gegebener Indexposition löschen




```
int i = 0;
Artikel h = anker;
if ((index < 0) || (index >= anzahlArtikel())) { /* Fehler */}
if (index == 0) { /* Listenelement am Anfang loeschen */ }
while (i < index - 1)
{
  i++;
  h = h.next;
}
h.next = h.next.next;
```

Plausibilitätscheck: Indexposition zum Löschen zulässig?

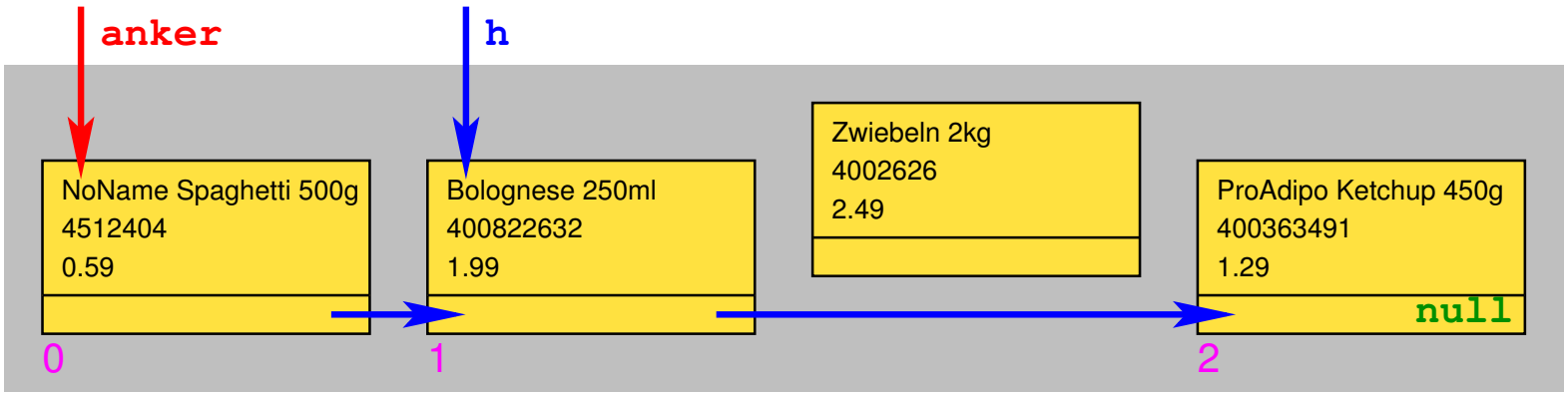
Listenelement an gegebener Indexposition löschen




```
int i = 0;
Artikel h = anker;

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

Liste durchlaufen, Zugangspunkt `h` unmittelbar vor zu löschendem Elem. platzieren

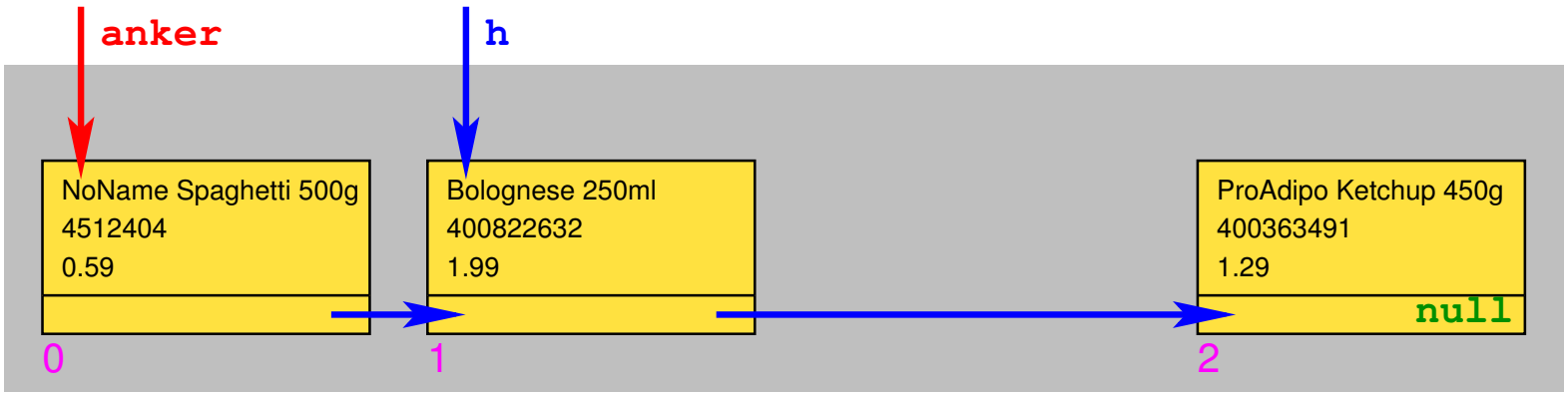
Listenelement an gegebener Indexposition löschen




```
int i = 0;
Artikel h = anker;

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

Zu löschendes Listenelement ausketten und Nachfolger neu setzen

Listenelement an gegebener Indexposition löschen

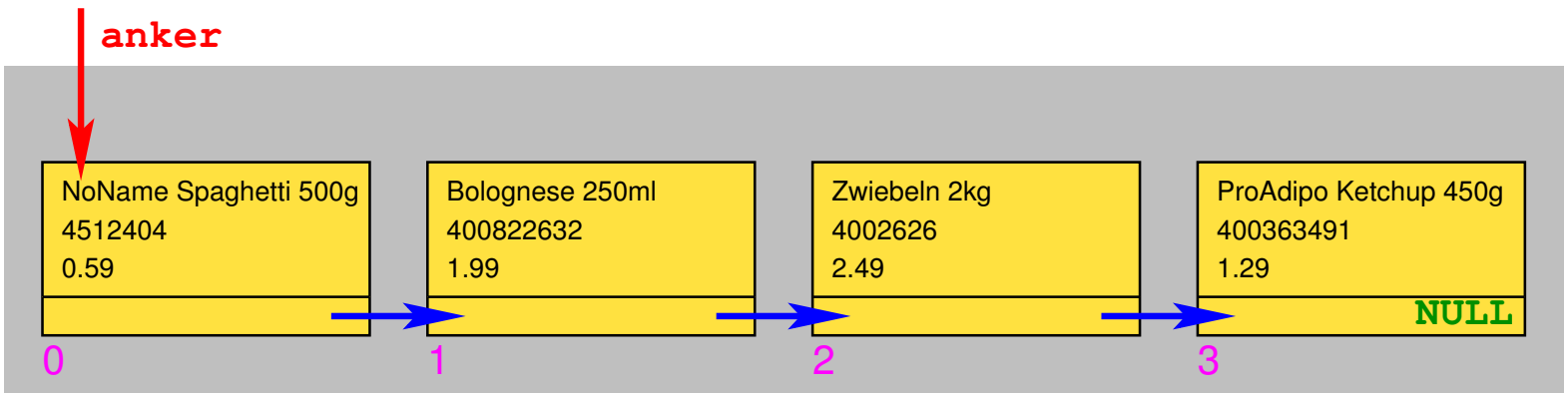



```
int i = 0;
Artikel h = anker;

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

Listenelement ohne Zugangspunkt ist aus Liste entfernt, sein Speicherplatz wird automatisch freigegeben. Fertig.

Listenelement am Listenanfang löschen (index 0)

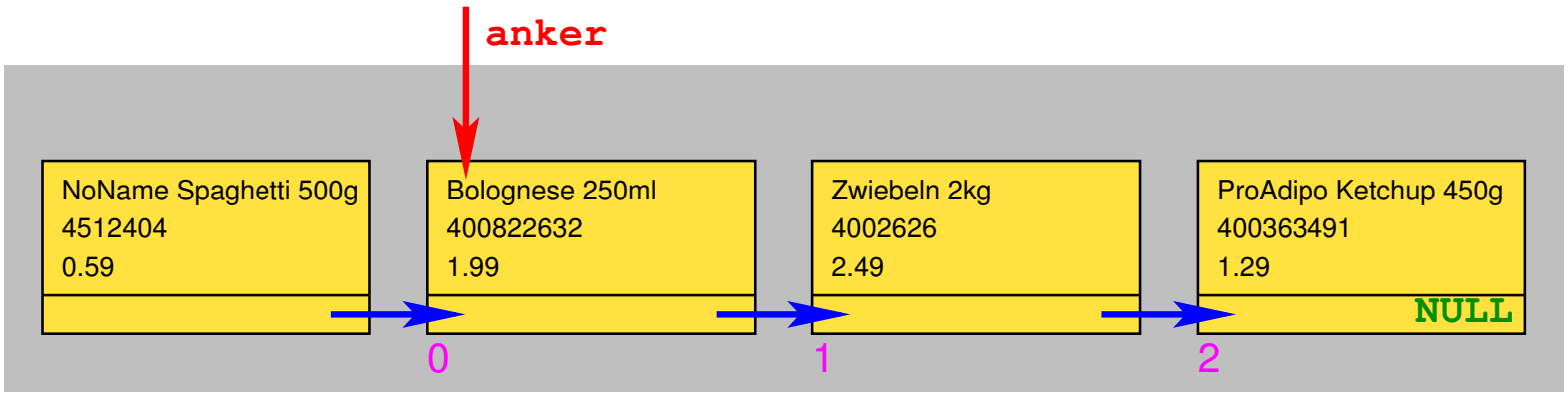
Listenelement am Listenanfang löschen




```
anker = anker.next;
```

Zugangspunkt Listenanker **anker** nutzen

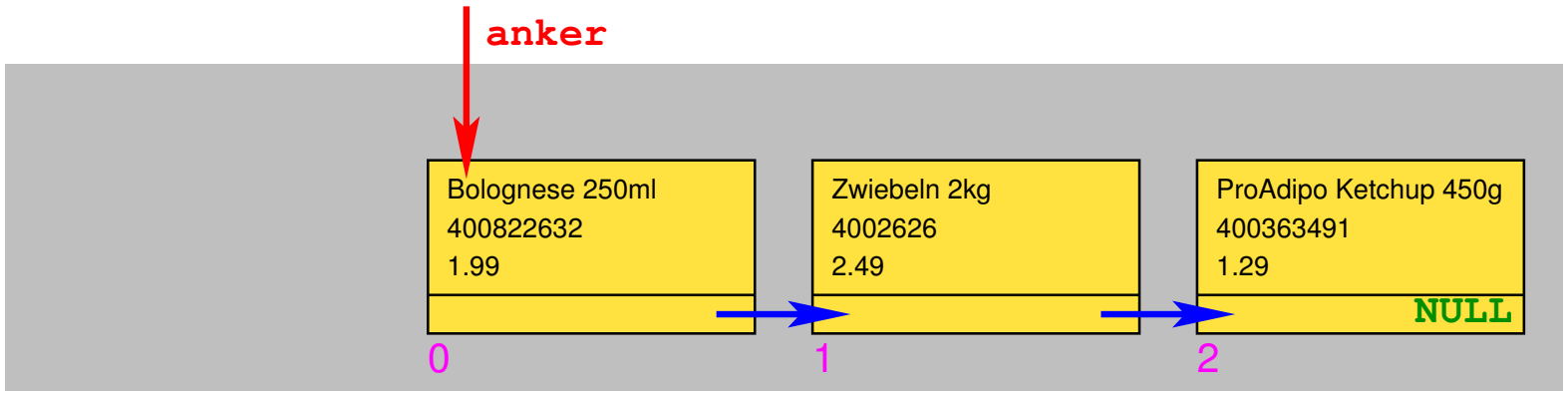
Listenelement am Listenanfang löschen




```
anker = anker.next;
```

Listenanker **anker** ein Element weiterschieben

Listenelement am Listenanfang löschen




```
anker = anker.next;
```

Listenelement ohne Zugangspunkt ist aus Liste entfernt, sein Speicherplatz wird automatisch freigegeben. Fertig.

Unsere kleine selbstprogrammierte Datenbank

MeinWarenkorb.java sowie **Warenkorb.java** und **Artikel.java**

```
import java.io.*;

public class Warenkorb {

  private Artikel anchor; //Listenanker als Attribut

  public Warenkorb() {  //Konstruktor
    anchor = null;      //Leerverweis setzen, da Warenkorb initial leer
  }


  /* ----- neuen Datensatz einfuegen am Listenanfang */

  public void fuegeArtikelHinzu(String name, long code, float stueckpreis) {
    if ((name.length() < 1) || (code < 1) || (stueckpreis < 0)) {
      return; //Plausibilitaetspruefung. Datensatz nur bei gueltigen Daten anlegen
    }
    if (anchor == null) {
      anchor = new Artikel(name, code, stueckpreis, null); //erster Eintrag
    } else {
      anchor = new Artikel(name, code, stueckpreis, anchor); //Folgeeintrag
    }
  }
```

Quelltext **warenkorb.zip** zum Download auf LV-Webseite

Nach Entpacken Compilieren mit **javac MeinWarenkorb.java**

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 anzeigen
- Artikel im Warenkorb in Datei speichern

Programmierte Methoden

Datenverwaltungsschicht

`fuegeArtikelHinzu` ..... neuen Datensatz am Listenanfang einfügen
`anzahlArtikel` ..... Anzahl Datensätze bestimmen
`fuegeArtikelHinzuAnPos` .... neuen Datensatz an geg. Position einfügen
`gibProduktname` ..... Produktname im Datensatz an geg. Position liefern
`gibProduktcode` ..... Produktcode im Datensatz an geg. Position liefern
`gibStueckpreis` ..... Stueckpreis im Datensatz an geg. Position liefern
`loescheArtikel` ..... Datensatz an geg. Position löschen
`speichereWarenkorb` ..... alle Datensätze in Datei schreiben

Anwendungskern

`insertsort` ..... Datensätze aufsteigend nach Preis sortieren

Nutzeroberfläche

`zeigeWarenkorbAn` ..... 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

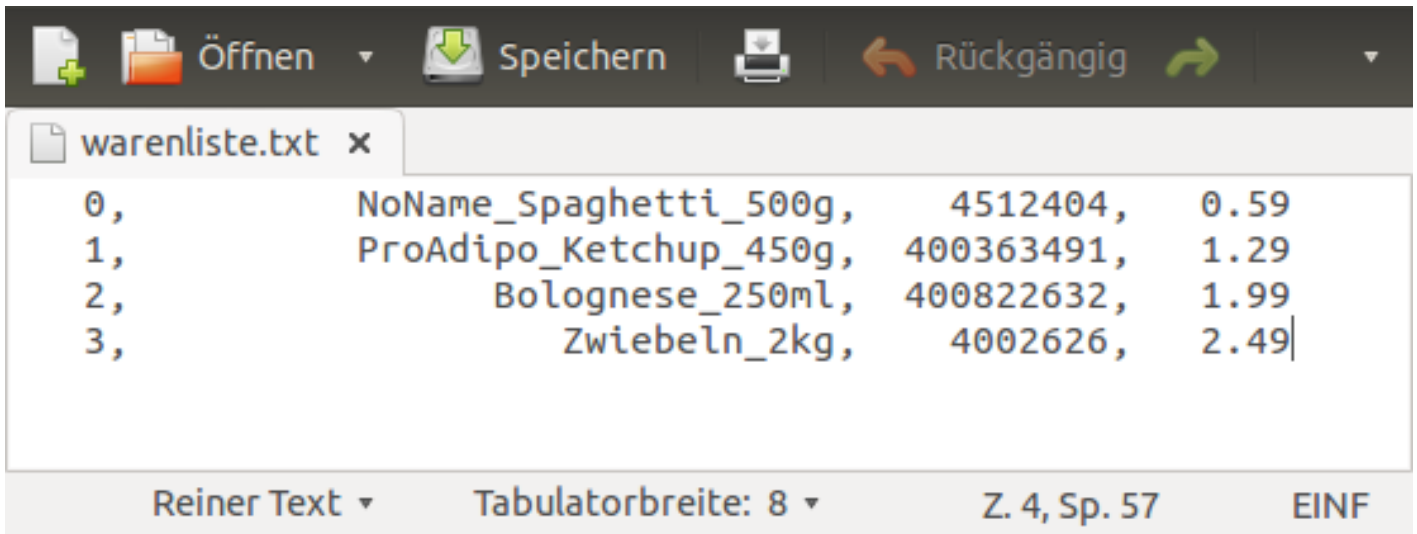
Nach dem Anlegen von vier Datensätzen

```
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

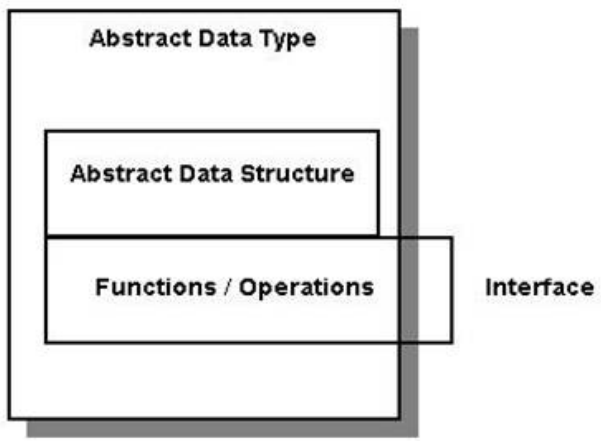


Insertionsort auf der Liste

```
public void insertsort() {
  String name;
  long code;
  float stueckpreis, stueckpreis2;
  int k = 1; /* Index, bis wohin schon sortiert */
  int r;

  while (k < anzahlArtikel()) {
    name = gibProduktname(k);
    code = gibProduktcode(k);
    stueckpreis = gibStueckpreis(k);
    loescheArtikel(k); //Datensatz mit index k aus Liste entfernen
    r = 0;
    do
    {
      stueckpreis2 = gibStueckpreis(r);
      r++;
    } while ((r-1 < k) && (stueckpreis > stueckpreis2));
    fuegeArtikelHinzuAnPosition(r-1, name, code, stueckpreis);
    k++;
  }
}
```

Abstrakter Datentyp

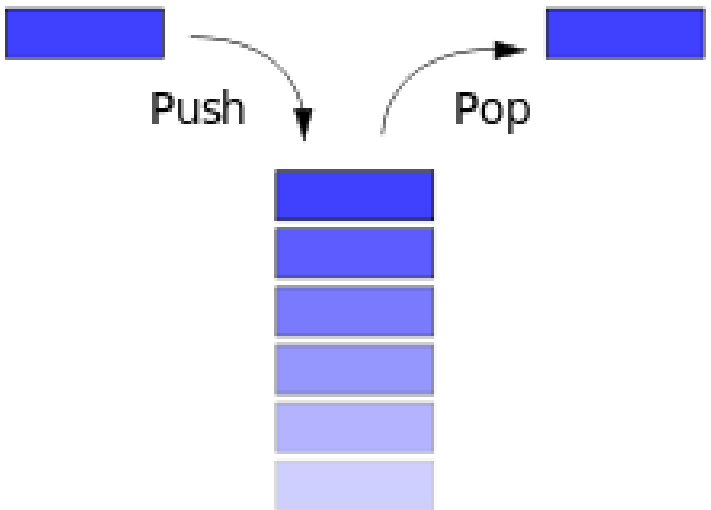
Als **abstrakten Datentyp** bezeichnet man eine *Datenstruktur* gemeinsam mit darauf wirkenden *Operationen*.



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

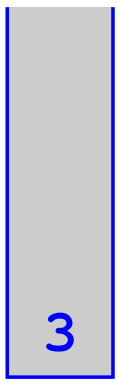
`isEmpty` ..... Test auf leeren Stack
`push` ..... Neues Element auf den Stack legen
`pop` ..... Oberstes Element auslesen und vom Stack nehmen

Stack zur Auswertung arithmetischer Terme

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

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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

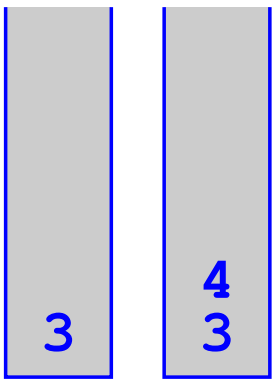


Stack zur Auswertung arithmetischer Terme

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

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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

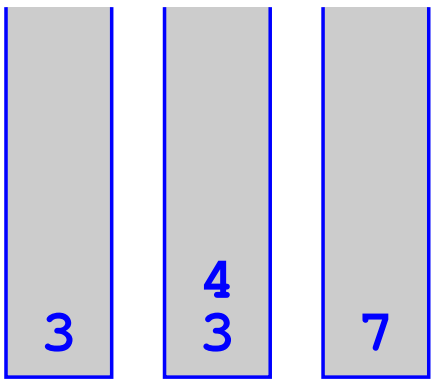


Stack zur Auswertung arithmetischer Terme

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

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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

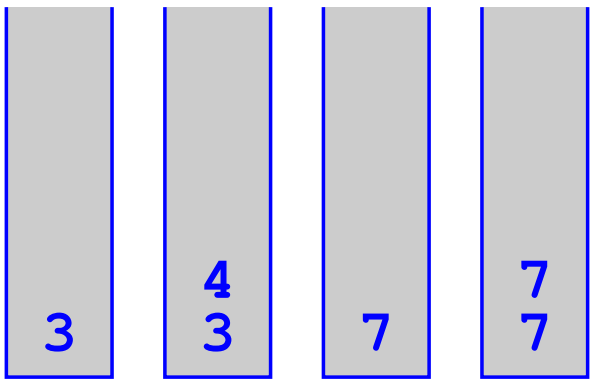


Stack zur Auswertung arithmetischer Terme

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

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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

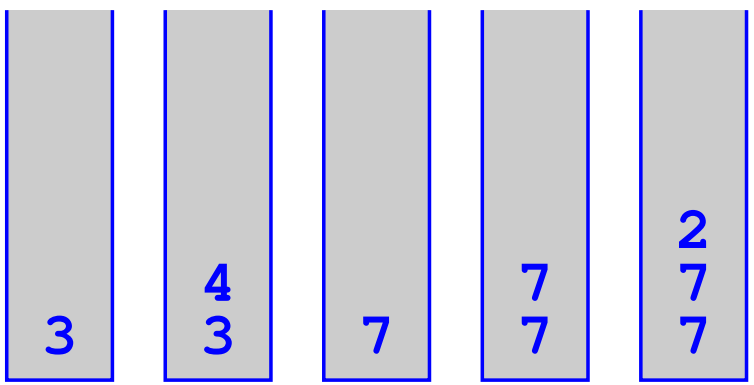


Stack zur Auswertung arithmetischer Terme

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

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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

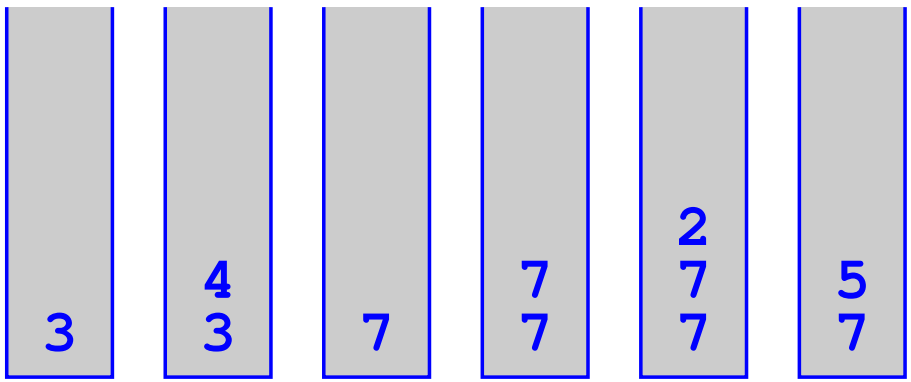


Stack zur Auswertung arithmetischer Terme

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

Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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

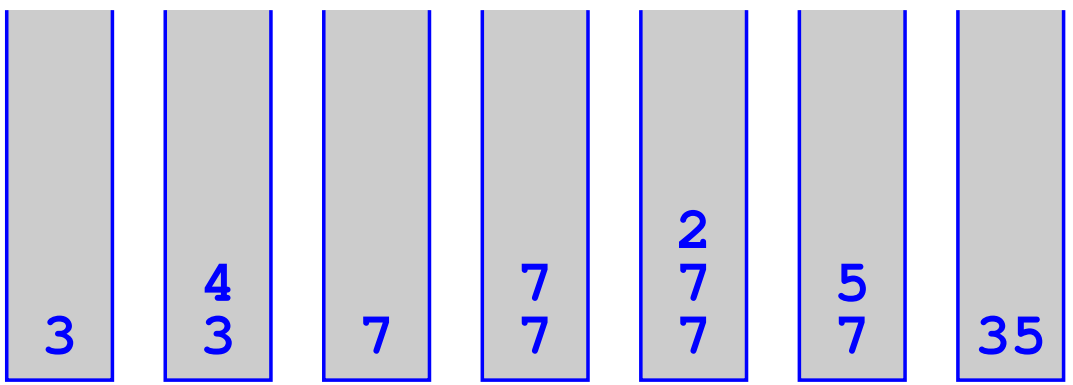


Stack zur Auswertung arithmetischer Terme

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

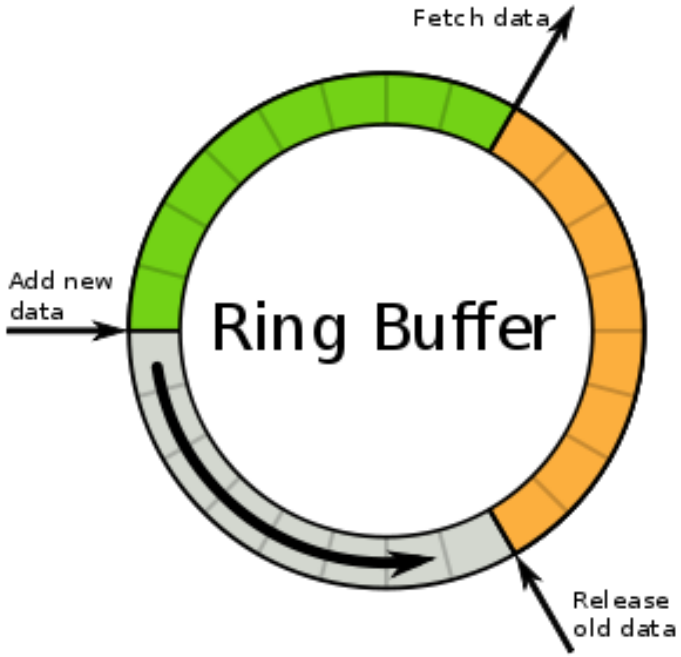
Umwandlung in
postfix–Ausdruck mit
Shunting–yard–Algorithmus

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



Ringpuffer

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

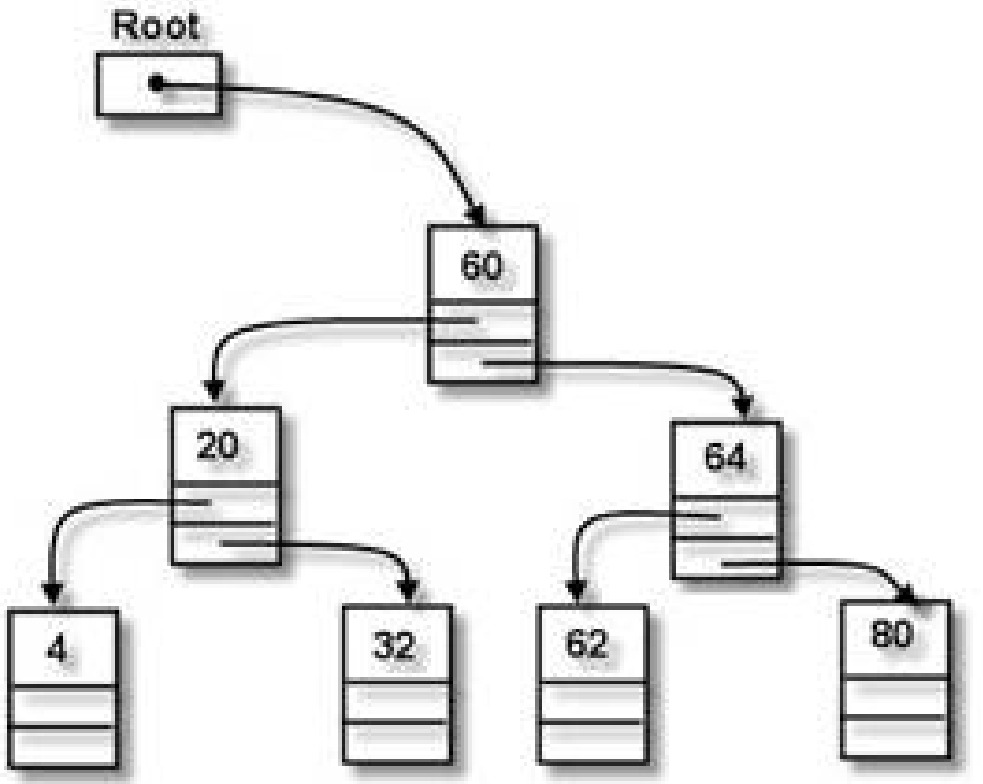


www.wikipedia.de

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)