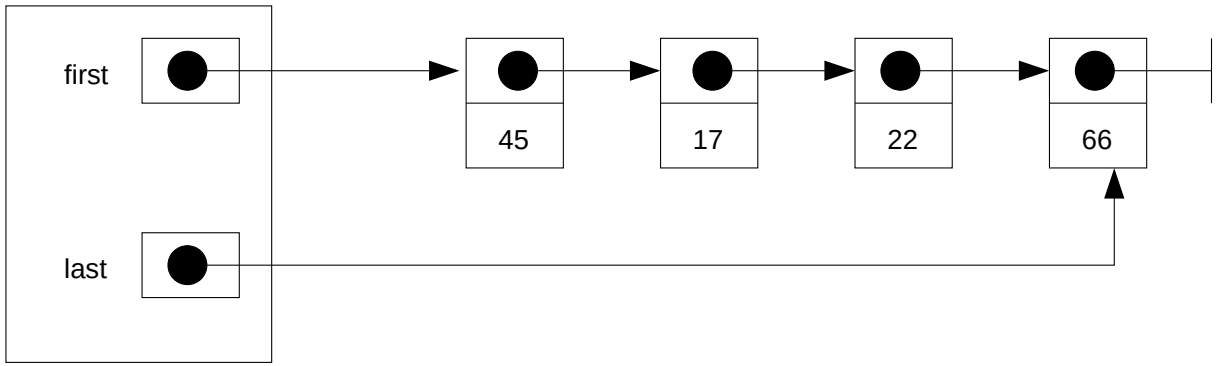
Einfach verkettete Liste

- Für jedes einzelne Element der Liste wird ein Hilfsobjekt erzeugt.
- Jedes Hilfsobjekt enthält zwei Instanzvariablen:
  - den zu speichernden Wert bzw. einen Verweis auf das zu speichernde Objekt und
  - einen Verweis auf das nächste Hilfsobjekt in der Liste.
- Das die Liste repräsentierende Objekt enthält wiederum zwei Verweise:
  - Einen Verweis auf das Hilfsobjekt zum ersten Listenelement und
  - einen Verweis auf das Hilfsobjekt zum letzten Listenelement.

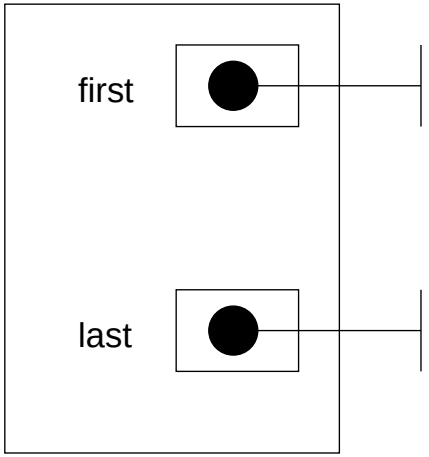


Implementierungsansatz: Verkettete Liste

- `first` und `last` enthalten zu Anfang die Nullreferenz (`null`), dies entspricht einer leeren Warteschlange.

```
public class Warteschlange<T> {
    ...
    private class Item {
        T    value;
        Item next;
    }

    private Item first = null;
    private Item last = null;
    ...
}
```



- `void enqueue(T elem)`

  Wir legen ein Hilfsobjekt an.

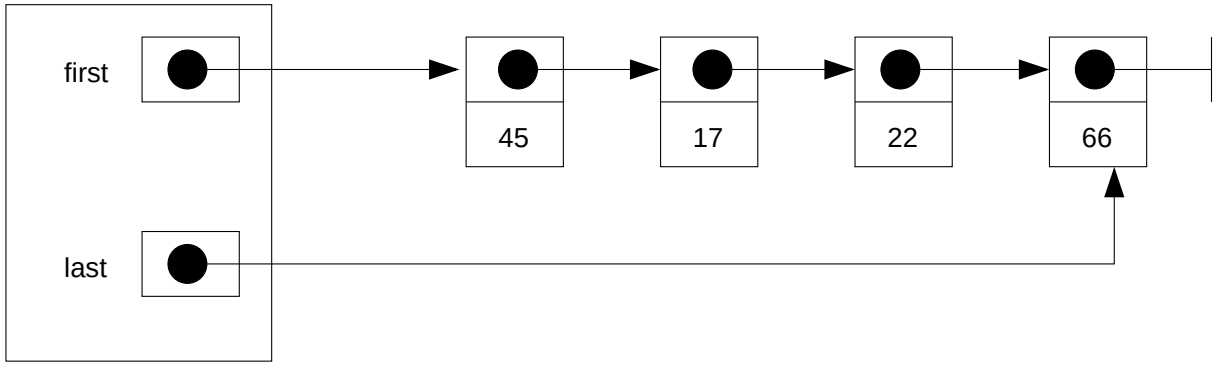
  Wenn die Warteschlange bisher leer ist, lassen wir `first` und `last` auf das Hilfsobjekt verweisen.

  Andernfalls müssen wir das neue Hilfsobjekt hinten an die Liste hängen.

```
Item item = new Item();
item.next = null;
item.value = elem;

if ( last == null ) {
    first = last = item;
}
else {
    last.next = item;
    last = item;
}
```

Beispiel: `enqueue(45); enqueue(17); enqueue(22); enqueue(66);`



- `T front()`

  Im Normalfall (`first != null`) geben wir das Element des ersten Hilfsobjektes zurück:

  ```
  return first.value;
  ```

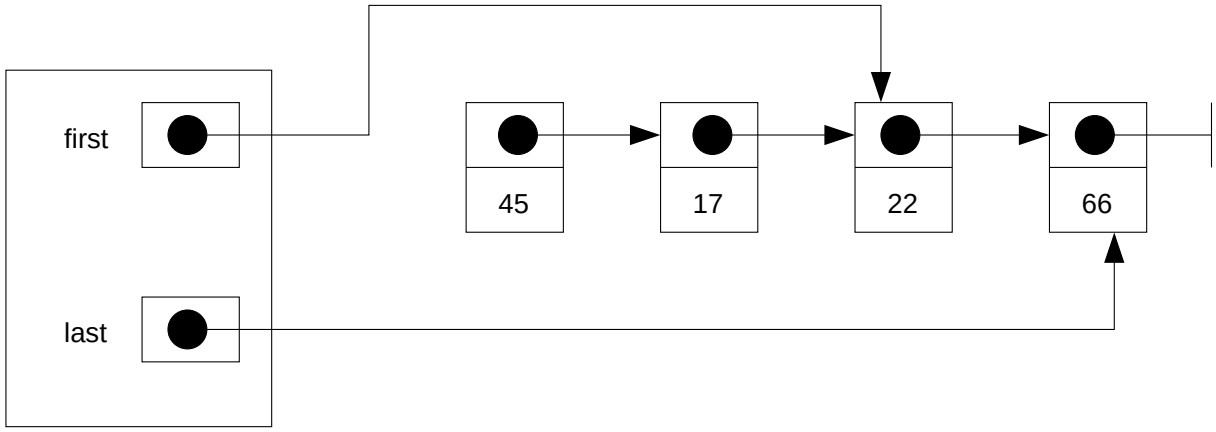
- `void dequeue()`

  Vorbedingung: Keine leere Liste

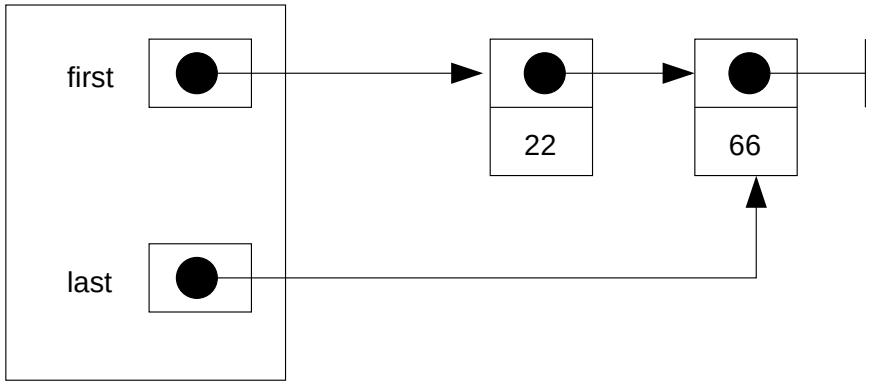
  Wir müssen nur `first` den Nachfolger von `first` zuweisen und dabei aufpassen, dass eine leere Warteschlange entstehen kann.

```
first = first.next;
if ( first == null ) {
    last = null;
}
```

  Beispiel: `dequeue(); dequeue();`



Man beachte, dass die nicht mehr referenzierten Hilfsobjekte später vom Garbage Collector gelöscht werden, so dass dann im Speicher der folgende Zustand entsteht:



Logisch besteht zwischen den beiden Zuständen kein Unterschied.

Effizienz mit verketteter Liste

- Alle Operationen können in Zeit $O(1)$ ausgeführt werden.
- Es gibt keine Kapazitätsbeschränkung mehr. Die Größe der Liste und nur noch durch den zur Verfügung stehenden Arbeitsspeicher beschränkt.

☞ dynamische Datenstruktur

Exkurs: Innere Klassen in Java

- Es gibt in Java die Möglichkeit, eine Klasse innerhalb einer anderen Klasse zu definieren. Solche Klassen heißen *innere Klassen*.
- Klassen die nicht innerhalb einer anderen Klasse definiert sind, sind sogenannte *Top-Level-Klassen*.
- Warum innere Klassen?
  - engere Bindung an die umgebende Klasse
  - Alternative zum Paketsystem
- Hier keine vollständige Behandlung von inneren Klassen, Details siehe Java Lehrbücher

Arten von inneren Klassen

- Statische innere Klassen
- Elementklassen
- Lokale Klassen
- Anonyme innere Klassen

Statische Innere Klassen

```
public class Aussen {
    ...
    static class Innen {
        ...
    }
    ...
}
```

- `Innen` ist die innere statische Klasse. Sie kann aufgebaut sein, wie eine übliche Klasse und innerhalb der inneren Klasse hat man Zugriff auf alle `static`-Elemente der äußeren Klasse.
- Statische innere Klassen sind eigenständige Klassen. Eine Instanz der inneren statische Klasse ist unabhängig von den Instanzen der äußeren Klasse.

- Erzeugung einer Instanz der inneren Klasse:
  - Innerhalb von `Aussen`:
    ```
    new Innen(...);
    ```
  - Außerhalb von `Aussen`:
    ```
    new Aussen.Innen(...);
    ```
- Die innere Klasse kann mit Modifikatoren für die Sichtbarkeit versehen werden (`public`, `private`).
- Statische innere Klassen bieten sich als Alternative zum Paketsystem an.

Elementklassen

```
public class Aussen {
    ...
    class Innen {
        ...
    }
    ...
}
```

- Eine Instanz einer Elementklasse ist immer mit einer Instanz der äußeren Klasse verbunden, d.h. zu einem `Innen`-Objekt gibt es stets genau ein `Aussen`-Objekt.
- Elementklassen stellen somit Hilfsobjekte für Instanzen der äußeren Klasse bereit, siehe verkettete Liste.
- In der inneren Klasse können keine `static`-Elemente deklariert werden.
- Von der inneren Klasse aus ist ein Zugriff auf alle Elemente der äußeren Klasse möglich, auch `private`-Elemente.

- Modifikatoren für die Sichtbarkeit der inneren Klasse sind möglich.
- Erzeugung einer Instanz der inneren Klasse:
  - geschieht typischerweise innerhalb einer Instanzmethode der äußeren Klasse. Hierdurch entsteht implizit die Zuordnung der Instanz der inneren Klasse zu `this`.
  - Ansonsten:
    ```
    Aussen aussen = new Aussen(...);
    aussen.new Innen(...);
    ```
    Zuordnung der inneren Instanz zu dem durch `aussen` referenzierten Objekt.
- Innerhalb der inneren Klasse bezeichnet `this` die Instanz von `Innen`.
- Wie kommt man an das umgebende Objekt? `Aussen.this`

Lokale Klassen

- Klassendefinitionen innerhalb von Blöcken, z.B. lokal innerhalb einer Methode.
- Keine Modifikatoren für die Sichtbarkeit erlaubt.
- Zugriff auf Methoden der äußeren Klasse, sowie auf Konstanten der umgebenden Methode oder Klasse.
- Keine große praktische Bedeutung.

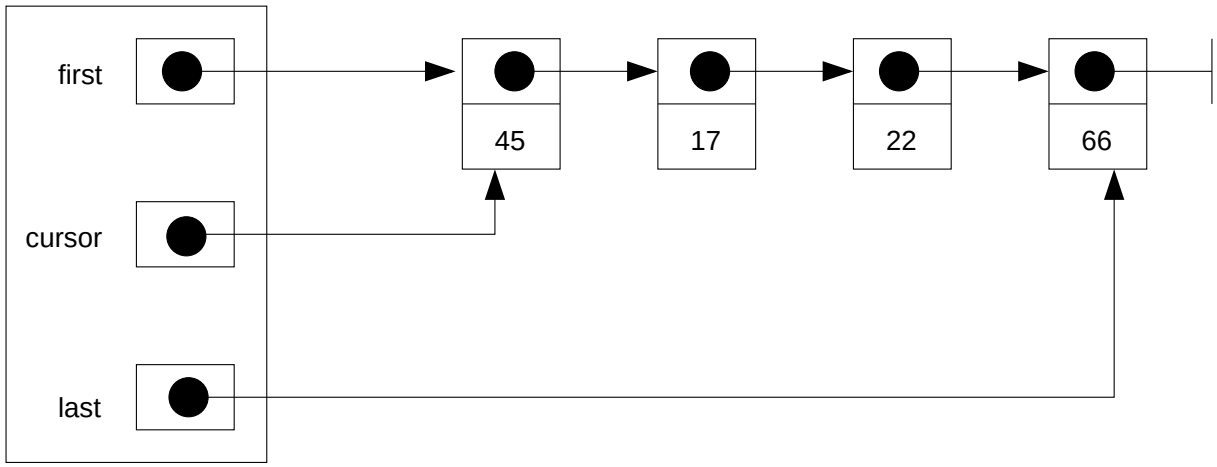
Anonyme innere Klasse

- Objekterzeugung und Klassendefinition in einem
- Praktische Bedeutung bei der lokalen Definition von Objekten für die Implementierung von einfachen Schnittstellen.
- Siehe `ActionListener`-Beispiel in Kapitel 2 (auf der Homepage)

Weitere Listenoperationen

- Für Listen gibt es viele weitere sinnvolle Operationen.
- Beispielsweise könnte man den sequentiellen schrittweisen Durchlauf durch eine Liste unterstützen.
- In der Instanzvariablen `cursor` merken wir uns die aktuelle Position in der Liste.
- `void reset()`
  Setzt den Cursor auf den Anfang der Liste.

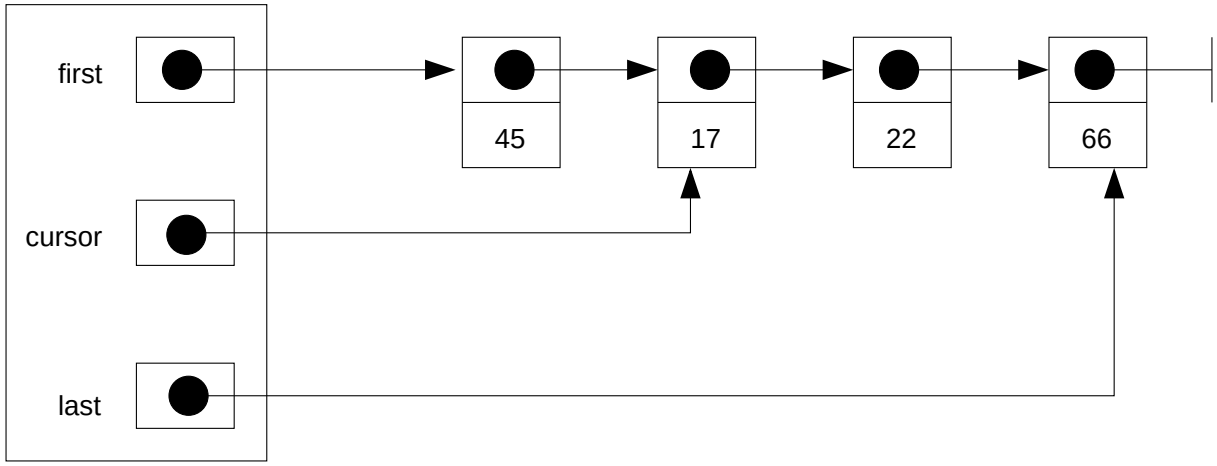
  ```
  cursor = first;
  ```



- `T next()`

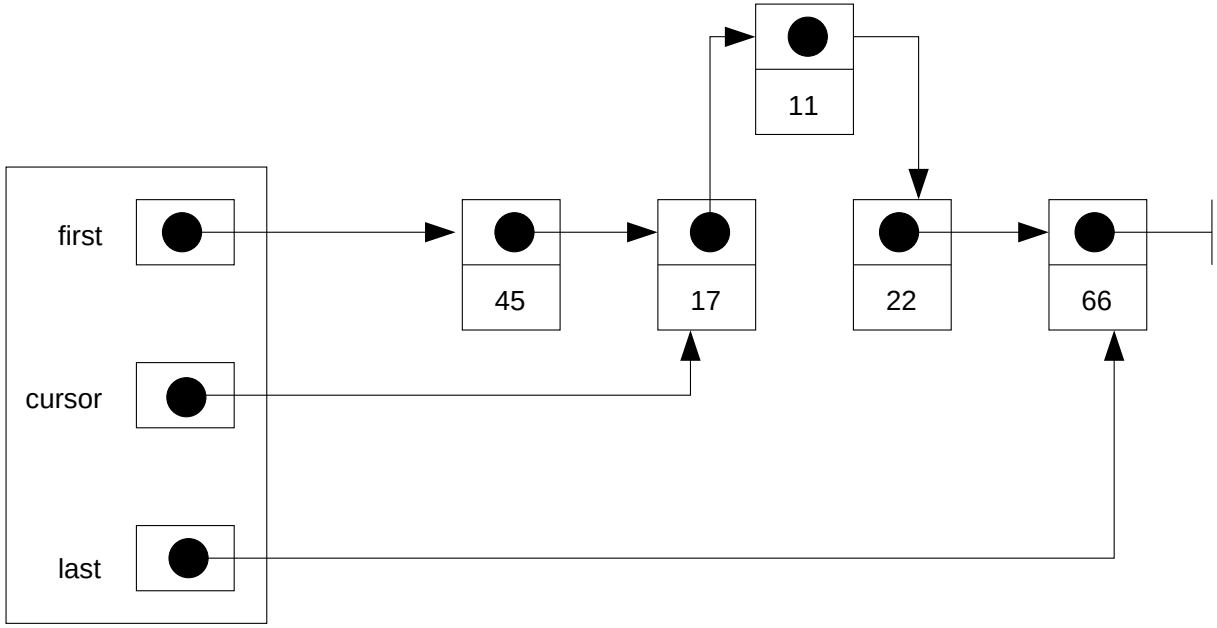
  Setzt den Cursor eine Position weiter und liefert das entsprechende Listenelement zurück.

```
cursor = cursor.next;
return cursor.value;
```



- `void insert(T elem)`
  Fügt hinter dem Cursor ein Listenelement ein.

```
Item item = new Item();
item.value = elem;
item.next = cursor.next;
cursor.next = item;
```

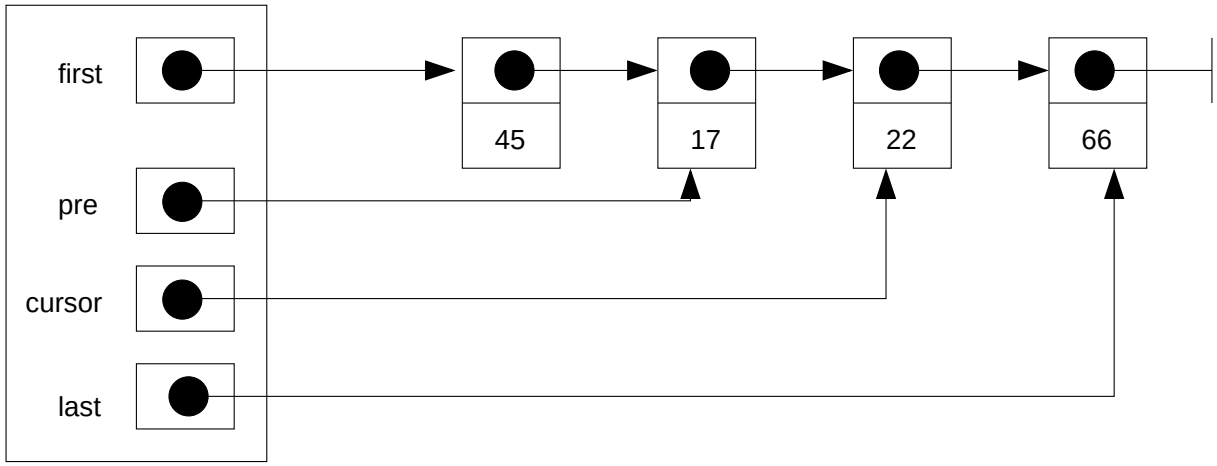


Zeitaufwand Einfügen: $O(1)$

- `void remove()`

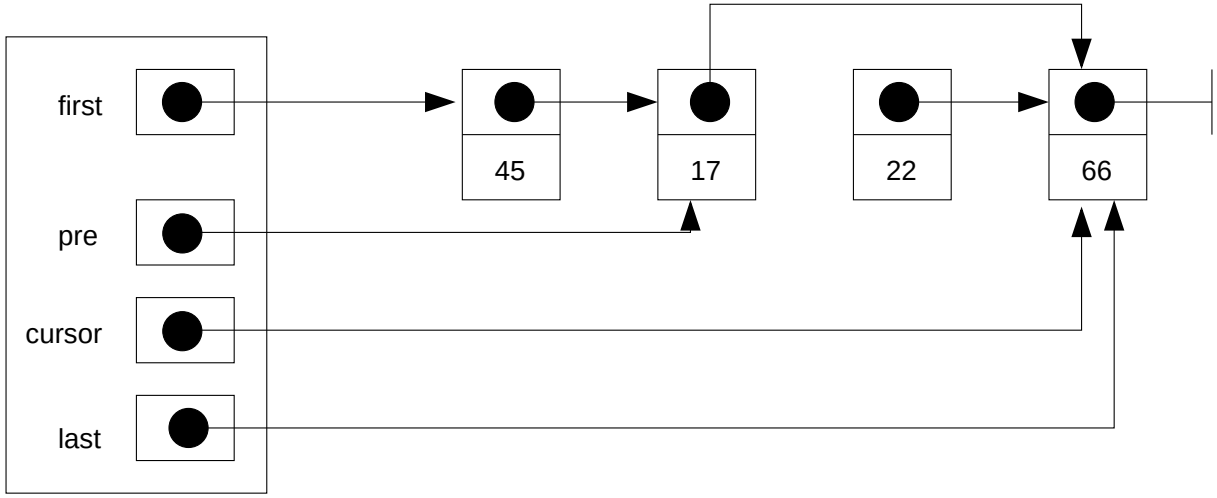
  Löscht das Cursorlement.

  Hierfür benötigt man einen Verweis auf das Element vor dem Cursor. Dies kann man durch einen Durchlauf durch die Liste ermitteln (Zeitaufwand $O(n)$) oder man sieht einen weitere Instanzvariable `pre` vor, die immer auf das Element vor dem Cursor verweist.



  `pre` muss dann natürlich bei den anderen Operationen entsprechend angepasst werden.

```
pre.next = cursor.next;
cursor = cursor.next;
```

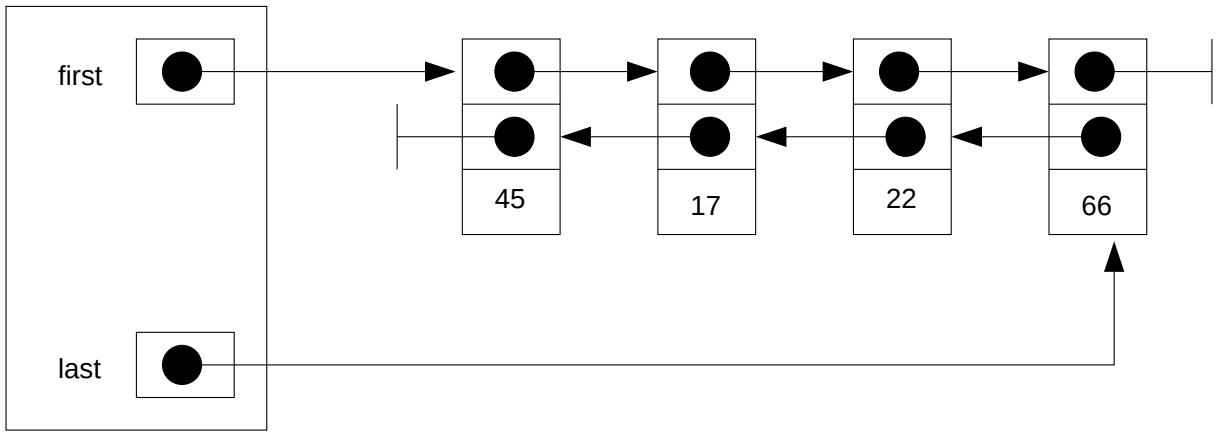


Damit ist auch Löschen in $O(1)$ möglich.

Das gelöschte Element wird später vom Garbage Collector entsorgt.

Doppelt verkettete Liste

In einer *doppelt verketteten Liste* verweist jedes Listenelement nicht nur auf den Nachfolger sondern auch auf den Vorgänger.



Dies erleichert z.B. das Löschen des aktuellen Elements (kein `pre` notwendig).

- `void remove()`

```
Item pred = cursor.prev;  // Vorgaengerelement
Item succ = cursor.next;  // Nachfolgerelement

if (pred == null) {       // Das erste Element wird geloescht
    first = succ;
}
else {
    pred.next = succ;
}

if (succ == null) {       // Das letzte Element wird geloescht
    last = pred;
}
else {
    succ.prev = pred;
}
```

- `void insert(T elem)`

  Hinter dem Cursor ein neues Element einfügen.

```
Item item = new Item();   // neues Element
Item succ = cursor.next;  // Nachfolgerelement

item.value = elem;
item.next = succ;
item.prev = cursor;

cursor.next = item;
if (succ == null) {       // Einfuegen hinter dem letzten Element
    last = item;
}
else {
    succ.prev = item;
}
```

  Analog ist natürlich auch ein Einfügen vor dem Cursor möglich.

Wahlfreier Zugriff

- Alle bisher vorgestellten Listenoperationen sind mit doppelt verketteten Listen in Zeit $O(1)$ implementierbar.
- Für die folgende Operation gilt dies nicht:

  `T get(int i)`

  Liefert das Element an der `i`-ten Stelle der Liste.

  Für eine Implementierung mit verketteten Listen beträgt der Zeitaufwand $O(n)$, eine Implementierung mit Feldern hätte dagegen den Zeitaufwand $O(1)$.
- Und wenn man nun `get()` sehr häufig benötigt?

  ☞ z.B. geschickte Größenanpassung des Feldes führt zu amortisierter Zeit $O(1)$ beim Einfügen

Dynamische Größenanpassung (1)

- Angenommen, es würde für eine Anwendung ausreichen, wenn eine Liste die Operationen `enqueue()` und `get()` unterstützt.
- Bei Implementierung mit Feld: Problem bei `enqueue()`, wenn das Feld für die Listenelemente vollständig gefüllt ist.
- Wir müssten dann ein größeres Feld erzeugen und alle Elemente in das neue Feld kopieren.

  Zeitaufwand: $O(n)$
- Angenommen, wir verdoppeln immer die Größe, wenn das Feld vollständig gefüllt ist. Welcher Gesamtaufwand entsteht dann?

Dynamische Größenanpassung (2)

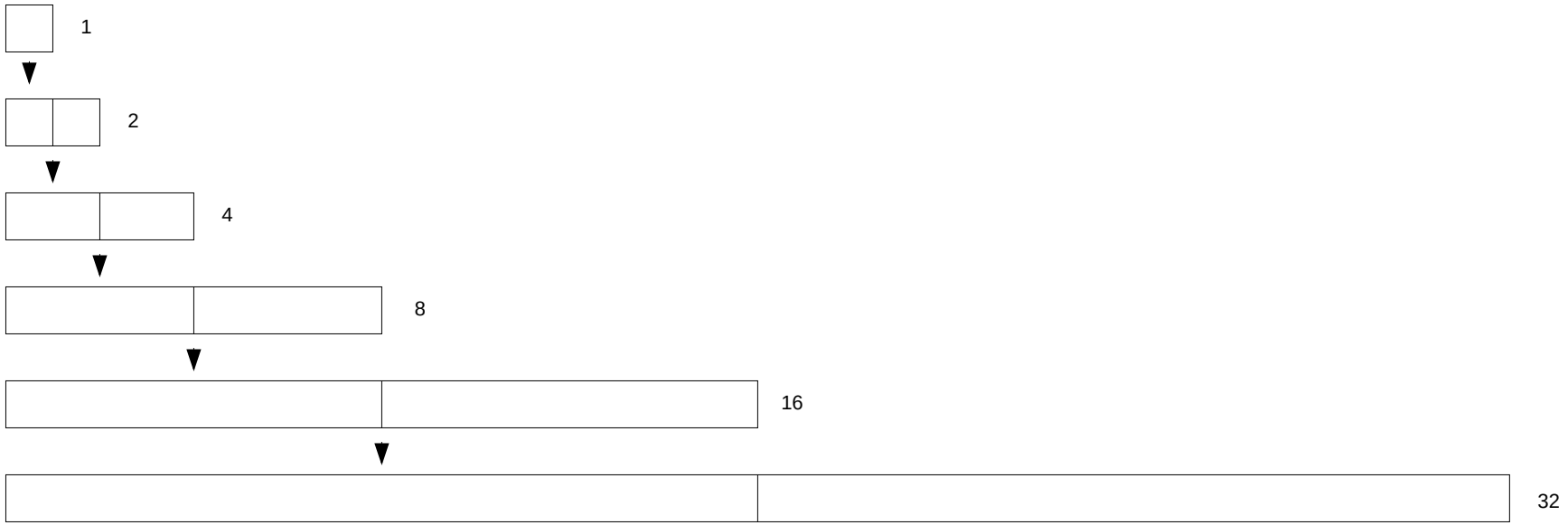
```
public class Liste<T> {
    private T[] feld = (T[]) new Object[1];
    private int size = 0;

    public void enqueue(T elem) {
        T[] neuFeld;

        if (size >= feld.length) {
            neuFeld = (T[]) new Object[2*feld.length];
            for (int i=0 ; i<feld.length ; i++) {
                neuFeld[i] = feld[i];
            }
            feld = neuFeld;
        }

        feld[size] = elem;
        size++;
    }
    ...
}
```

Dynamische Größenanpassung (3)



☞ Der Gesamtaufwand für `enqueue()` entspricht der Gesamtlänge aller erzeugten Felder.

Dynamische Größenanpassung (4)

Es sei n die Anzahl der eingefügten Elemente (Länge der Liste). O.B.d.A. sei n eine Zweierpotzen, also $n = 2^k$. Dann gilt für die Gesamtlänge:

$$\begin{aligned} \text{Gesamtlänge} &= n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^k} \\ &= n\,(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k}) \\ &= n \sum_{i=0}^{k} \left(\frac{1}{2}\right)^k \end{aligned}$$

Mit der Formel

$$\sum_{i=0}^{k} x^k = \frac{1 - x^{k+1}}{1 - x}$$

können wir die Gesamtlänge abschätzen:

$$\begin{aligned}
\text{Gesamtlänge} &= n \sum_{i=0}^{k} \left(\frac{1}{2}\right)^k \\
&= n \frac{1 - \left(\frac{1}{2}\right)^{k+1}}{\frac{1}{2}} \\
&= 2n \left(1 - \left(\frac{1}{2}\right)^{k+1}\right) \\
&\leq 2n
\end{aligned}$$

Fazit:

- ☞ Der Gesamtzeitaufwand für das Einfügen von n Elementen beträgt $O(n)$.
- ☞ Damit benötigt eine einzelne Einfügeoperation im Mittel Zeit $O(1)$.

Amortisierte Laufzeitanalyse

- Die durchgeführte Analyse ist eine sogenannte *amortisierte Laufzeitanalyse*.
- Bei der amortisierten Laufzeitanalyse betrachtet man die Kosten (Zeitaufwand) von Folgen von Operationen, nicht nur einer einzelnen Operation.
- Dividiert man den Zeitaufwand durch die Anzahl der Operationen, erhält man den durchschnittlichen Zeitaufwand pro Operation (Aggregat-Methode).
- Man beachte:
  - Der durchschnittliche Zeitaufwand für ein einzelnes `enqueue()` ist nur $O(1)$, obwohl
  - der Zeitaufwand für ein einzelnes `enqueue()` im Worst-Case $O(n)$ beträgt.
  - Der Zeitaufwand für n-faches `enqueue()` beträgt ebenfalls nur $O(n)$.

Listen im Java-API

- Generische Klassen und Schnittstellen für Listen finden sich im Paket `java.util`.
- Generische Schnittstelle für Listen: `List<T>`
- Feldbasierte Implementierung: `ArrayList<T>`

  `ArrayList` nutzt eine dynamische Größenerweiterung des Feldes. Zitat aus der API-Dokumentation:

  > Resizable-array implementation of the List interface. (...) The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time.

- Doppelt verkettete Liste: `LinkedList<T>`

  Ein kleiner Trick senkt etwas den Aufwand bei `get()`, hat aber keinen Einfluss auf die Größenordnung:

  > Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.