

7. Verkettete Strukturen: Listen

Java-Beispiele:

IntList.java

List.java

Stack1.java

Schwerpunkte

- Vergleich:
Arrays – verkettete Listen
- Listenarten
- Implementation:
 - Pascal (C, C++): über Datenstrukturen
→ Records (Structs) + Pointertyp
 - Java: über Klassen + Objektreferenzen
- Gängige Funktionen für Listen:
length, nthElement, addToEnd, ...
- Noch einmal 'Stack':
diesmal unbeschränkt
- Lokale Klassen

Verkettete Listen:

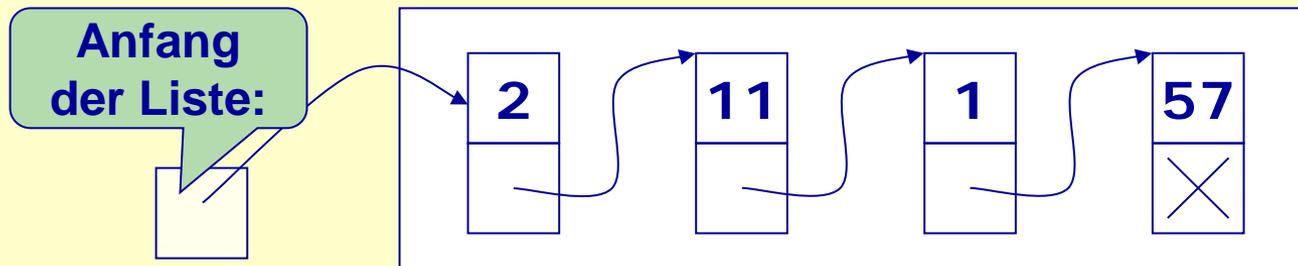
Überblick und Grundprinzip

Verkettete Listen

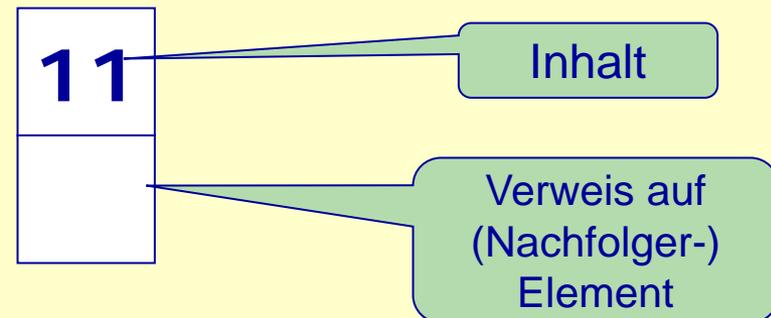
Verkettete Liste:

Folge von Elementen, die miteinander über Verweise (Zeiger) verkettet sind

Beispiel:
einfach
verkettete
Liste



Element (Zelle):



Verkettete Listen: Aufgabe

Grundaufgabe vieler Programme:

Information abspeichern und wiederfinden

Darstellung: Array, Liste, Baum ...

Arrays:

```
int[] a = new int[n];
```

Anzahl der Elemente n muss erst zur Laufzeit festgelegt werden.

Aber: einmal festgelegte Größe n kann nicht mehr verändert werden.

Probleme mit Arrays:

- n zu groß: Platz verschwendet
- n zu klein: u. U. Laufzeitprobleme (Überlauf)

Beispiele:

- Compiler: Anzahl der Bezeichner eines Programms?
- Unterschiedliche Wörter in Goethe, 'Faust I'?

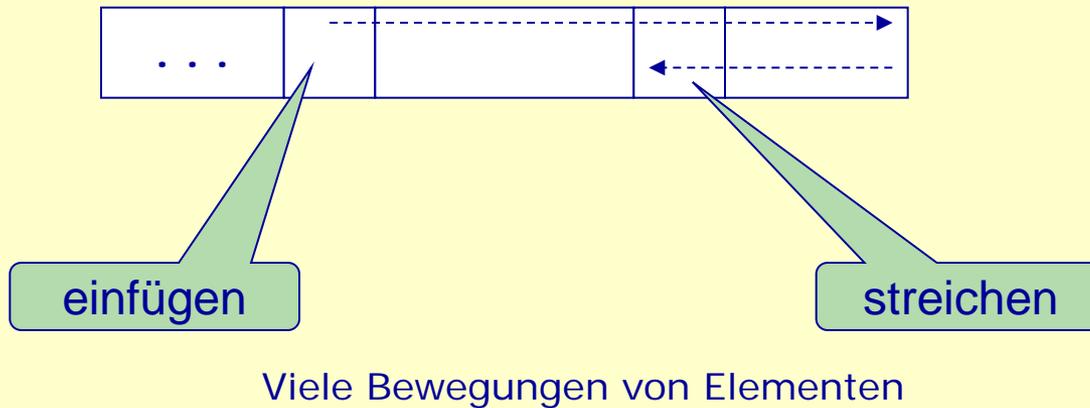
(Aufgabe: Wie häufig kommen einzelne Wörter vor?)

Verkettete Listen: Vorzüge

Verkettete Liste: sehr flexible Datenstruktur

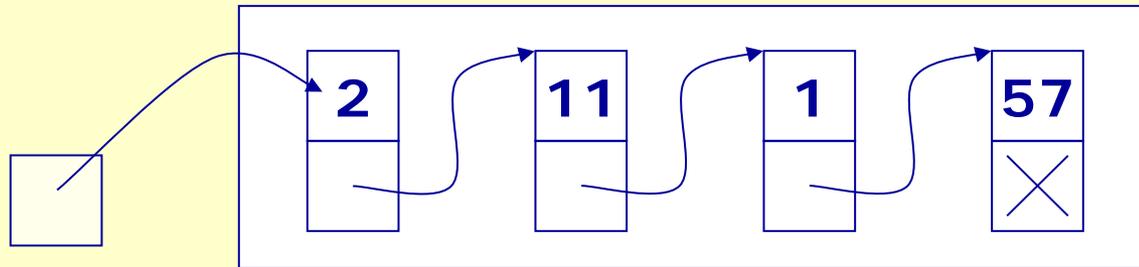
- Beliebige Anzahl von Elementen
(keine Größenbeschränkung)
- Einfügen, Streichen von Elementen effizient

Array:



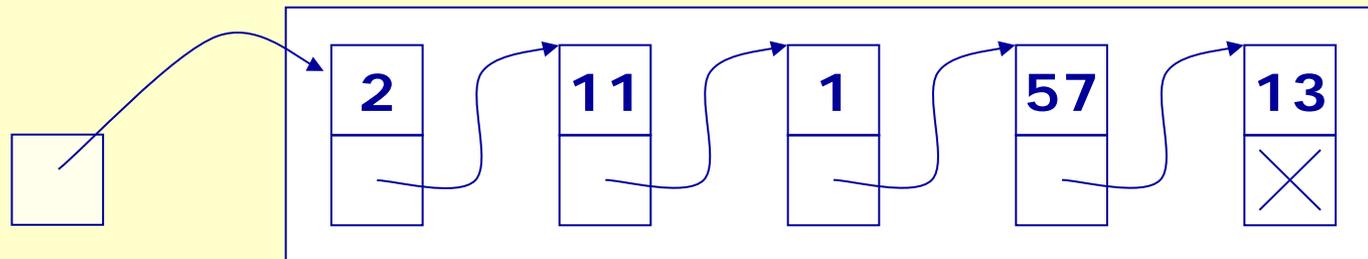
Verkettete Liste: Operationen

Beispiel:
einfach
verkettete
Liste



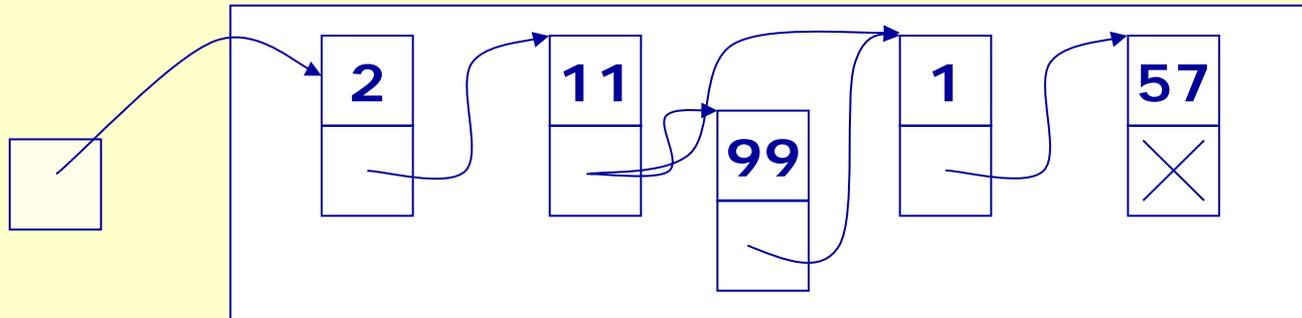
Neues Element anhängen:

- Speicherplatz für neue Zelle anfordern (dynamisch zur Laufzeit)
- Verweis auf diese Zelle einrichten → keine Längenbeschränkung



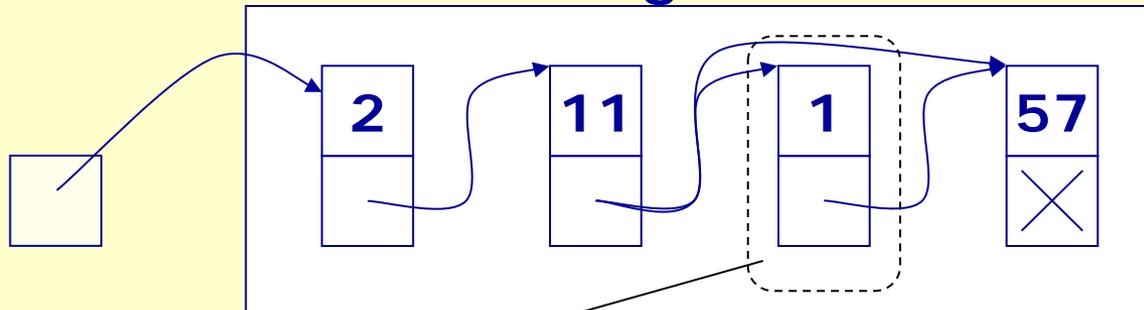
Liste: weitere Operationen

- Einfügen eines neuen Elements an beliebiger Stelle



(kein Verschieben von Daten)

- Streichen eines beliebigen Elements



- Nur logisch gestrichen
- Physisch noch im Speicher:
Garbage Collector (Java System) gibt Speicher zurück

Garbage Collector (Müllsammler)

**Automatische Freigabe von nicht mehr
benötigtem Speicherplatz *)
durch das Java-Laufzeitsystem**

*) Speicherplätze, die nicht über Variablen
des Programms (direkt oder über Verweislisten)
erreichbar sind

→ Java: Programmierer muss sich in Java nicht
um Freigabe kümmern

(Teil der JVM realisiert das: Garbage Collector)

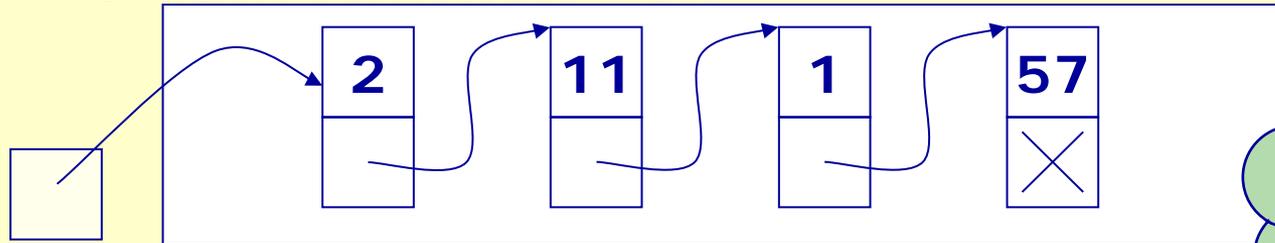
- anders als in C++, Pascal, Modula-2

(dort: explizite Anweisungen zur Freigabe
von Speicherplatz im Programm)

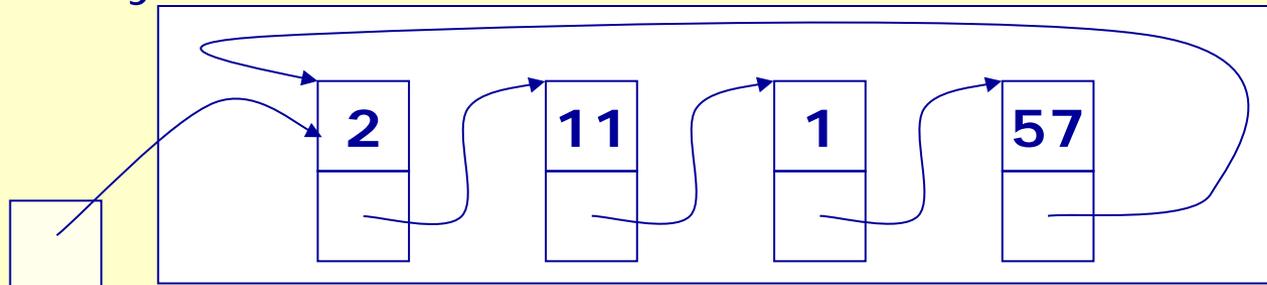
- Technik stammt von funktionalen Sprachen: Lisp

Listenarten

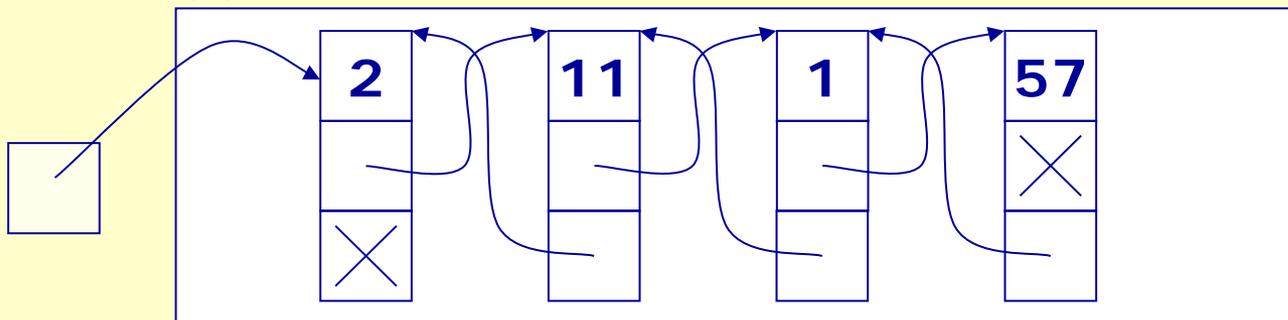
- einfach verkettete Liste



- zyklisch verkettete Liste



- doppelt verkettete Liste



- zyklisch doppelt verkettete Liste: analog

Wovon hängt die konkrete Auswahl ab ?

Implementation verketteter Listen

IntList.java
List.java

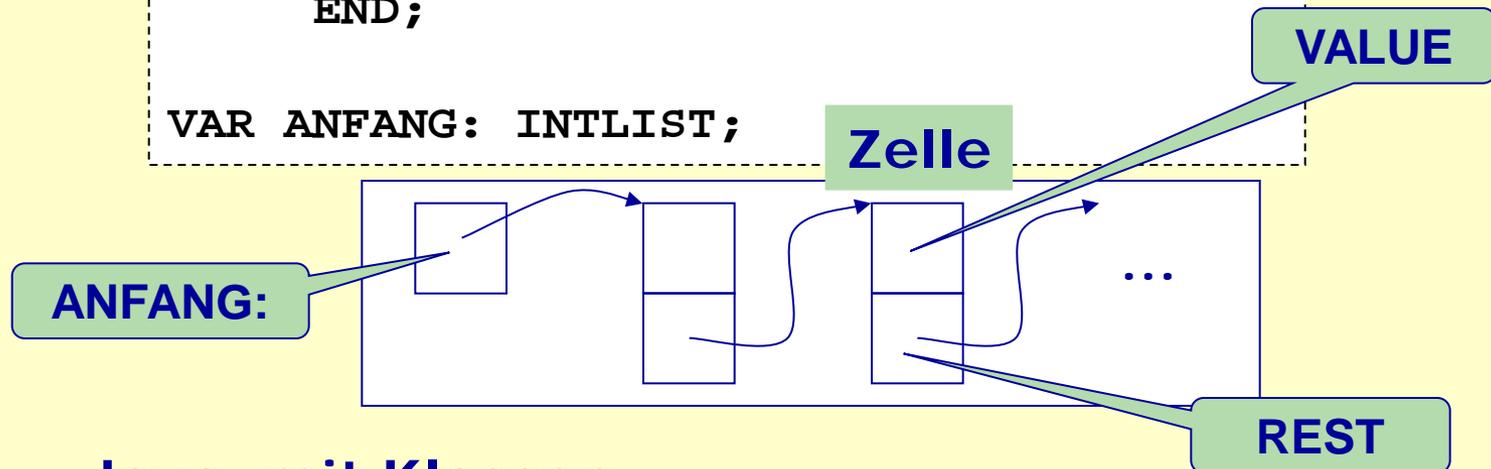
Implementation

(von nun an: nur einfach verkettete lineare Liste)

- **Pascal (C, C++): mit Datenstrukturen**
Record (Struct) + Pointertyp

```
TYPE INTLIST = @ ZELLE;  
  ZELLE =  
    RECORD  
      VALUE: INTEGER;  
      REST:  INTLIST  
    END;
```

```
VAR ANFANG: INTLIST;
```



- **Java: mit Klassen**
Java hat weder Records noch Pointer

Implementation in Java

IntList.java

```
class IntList {
```

```
    private int value ;  
    private IntList rest ;
```

Rekursive Definition:
Typ der Variablen = definierte Klasse

Wie Pascal-Record
(eine Zelle)

```
    public IntList (int v, IntList next) {  
        value = v;  
        rest = next;  
    }  
    public int getValue () { return value; }  
  
    public void setValue (int val) {  
        value = val;  
    }  
  
    public IntList getRest () {  
        return rest;  
    }  
    ...  
}
```

Erzeugung einer Liste

```
class IntList {  
    private int value ;  
    private IntList rest ;  
    public IntList (int v, IntList next) {  
        value = v;  
        rest = next;  
    } ...  
}
```

List.java
IntList.java

null Objekt:
leerer Verweis

in: List.java:

- 1 `IntList list = new IntList(57, null);`
- 2 `list = new IntList(1, list);`

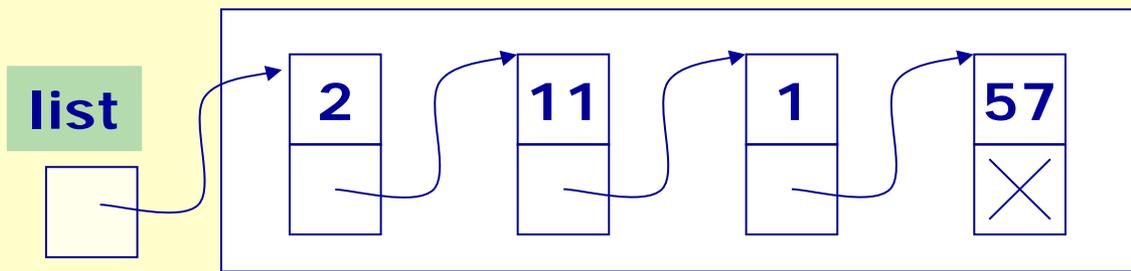


Erzeugung und Ausgabe einer Liste

```
class List {  
    public static void main (String[] args) {  
        IntList list = new IntList(57, null);  
        list = new IntList(1, list);  
        list = new IntList(11, list);  
        list = new IntList(2, list);  
  
        IntList temp;  
        for (temp = list; temp != null; temp = temp.getRest())  
            System.out.println(temp.getValue() + ", ");  
        System.out.println();  
  
        System.out.println(list.toString());  
    }  
}
```

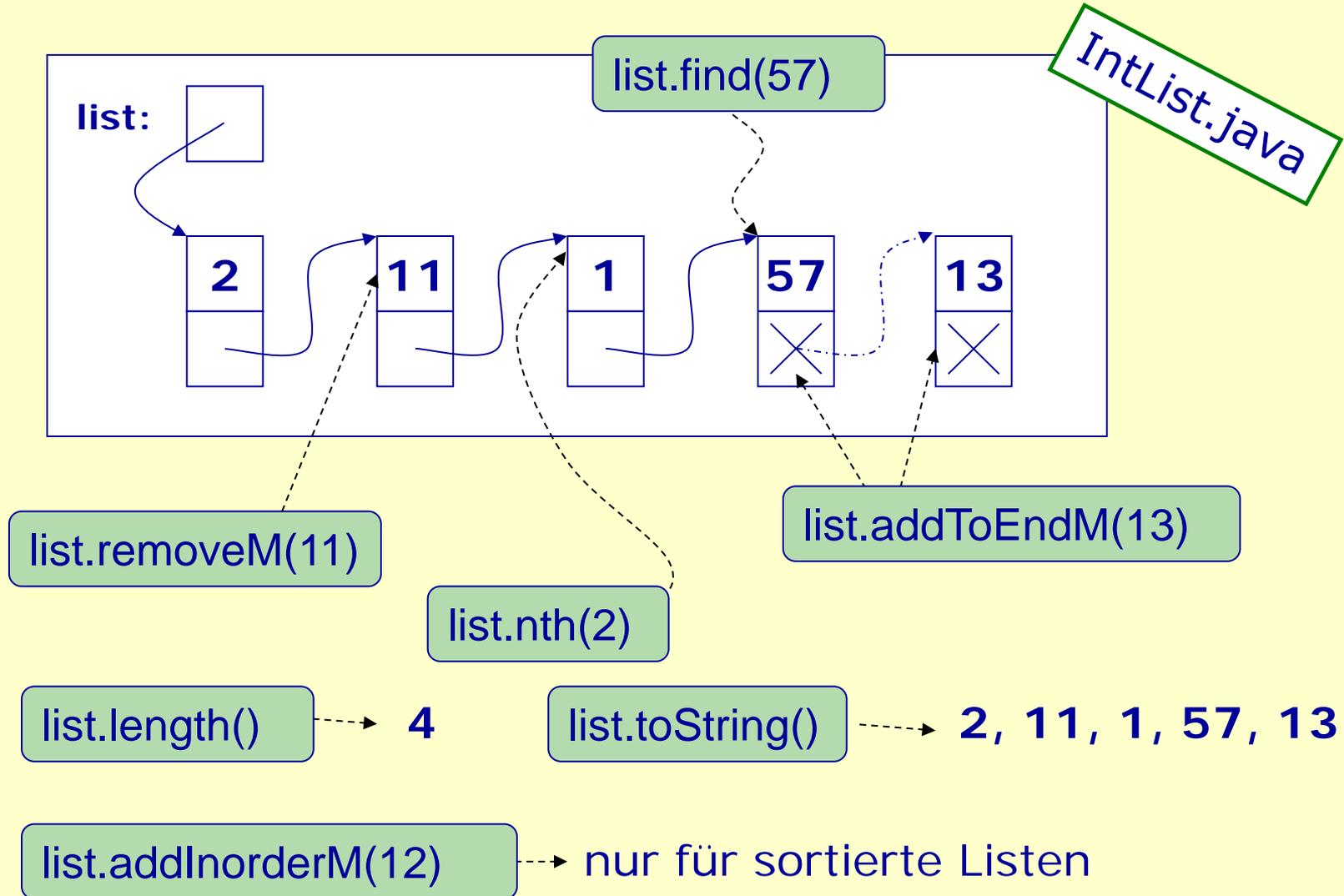
List.java

IntList.java



```
% java List  
2,  
11,  
1,  
57,  
  
2, 11, 1, 57
```

Listenfunktionen: Überblick



"M" im Methodennamen: Liste wird modifiziert

Implementation von Listenfunktionen

Listen als Datenstrukturen betrachtet als ...

- Rekursive Datenstruktur
(Liste = Listenkopf gefolgt von Restliste)
→ rekursiver Algorithmus natürlich
- Iterative Datenstruktur
(Liste = Folge von Elementen)
→ iterativer Algorithmus natürlich

VL

Aufgabe: iterative Implementation selbst

Wichtige Funktionen für Listen (1)

IntList.java

```
class IntList {
    private int value ;
    private IntList rest ;
    public IntList (int v, IntList next)...

    public int length () ...
}
```

- Länge der Liste

```
public int length () {
    if (rest == null)
        return 1;
    else
        return 1 + rest.length();
}
```

Wichtige Funktionen für Listen (2)

IntList.java

- Finde (Verweis auf) Zelle mit Inhalt 'key'

```
public IntList find (int key) {  
    if (value == key)  
        return this;  
    else if (rest == null)  
        return null;  
    else  
        return rest.find(key);  
}
```

Aktuelles Objekt:
Verweis auf aktuelle
Zelle

Nicht gefunden

Suche weiter

Binäre Suche
nicht möglich
→ $O(n) = \frac{1}{2} n$

Schwachstellen von Listen:
* Speicherbedarf
* Suchzeiten

Wichtige Funktionen für Listen (3)

- Finde Verweis auf die n-te Zelle

IntList.java

```
public IntList nth (int n) {
    if (n == 0)
        return this;
    else if (rest == null)
        return null;
    else
        return rest.nth(n-1);
}
```

Wichtige Funktionen für Listen (4)

- Neue Zelle an das Ende der Liste anhängen

IntList.java

```
public void addToEndM (int val) {  
    if (rest != null)  
        // a cell in the middle  
        rest.addToEndM(val);  
    else // the last cell  
        rest = new IntList(val, null);  
}
```

Verkettung der
Liste am Ende mit
neuer Zelle

Wichtige Funktionen für Listen (5)

IntList.java

- Füge Element entsprechend der Größe in eine (aufsteigend) sortierte Liste ein

→ Aufbau sortierter Listen: nur mit `addInOrderM()`

```
public IntList addInorderM (int n) {
    if (n < value)
        return new IntList (n, this);
    else if (n == value)
        return this;
    else if (rest == null) {
        rest = new IntList (n, null);
        return this;
    }
    else {
        rest = rest.addInorderM(n);
        return this;
    }
}
```

n kleinste Zahl:
vorn einfügen

n bereits vorhanden:
nicht einfügen

n größte Zahl:
hinten anfügen

sonst:
rekursiv in der Mitte einfügen

Nacharbeiten der Listen-Implementation

- **Durcharbeiten:**

- `IntList.java`
- `List.java`

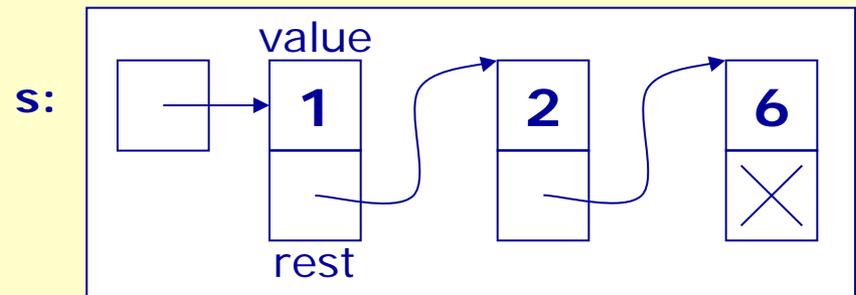
- **Verstehen:**

- `Bilder malen`

- **Aufgabe:**

- `Iterative Lösung selbst implementieren`

```
s.addInorderM (20);
```



(3. Fall: neue größte Zahl)

Stack beliebiger Größe

Stack1.java

Lokale Klassen

Noch einmal 'Stack': diesmal unbeschränkt

Stack1.java

```
class Stack1 {  
  
    private class Zelle {  
        Object inhalt;           // Inhalt  
        Zelle next;             // Verweis  
    }  
  
    // Verweis auf oberste Zelle  
    private Zelle top;  
  
    public Stack1() {  
        top = null;  
    }  
    public boolean isempty() {  
        return top == null;  
    }  
    ...  
}
```

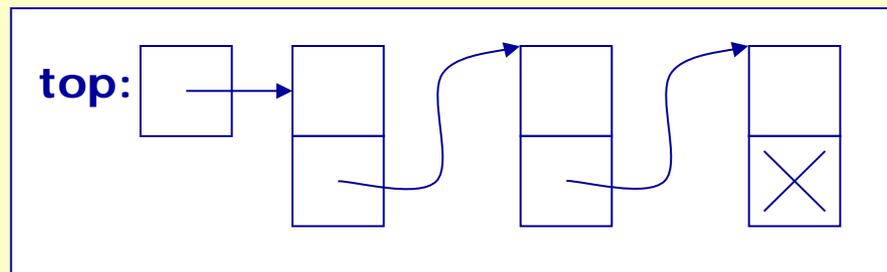
Lokale Klasse
(Datensammlung)

Klasse 'Stack': Daten durch Variable 'top' repräsentiert

```
class Stack {  
    private class Zelle {  
        Object inhalt; // Inhalt  
        Zelle next; // Verweis  
    }  
  
    // Verweis auf oberste Zelle  
    private Zelle top;  
  
    ...  
}
```

lokale Klasse

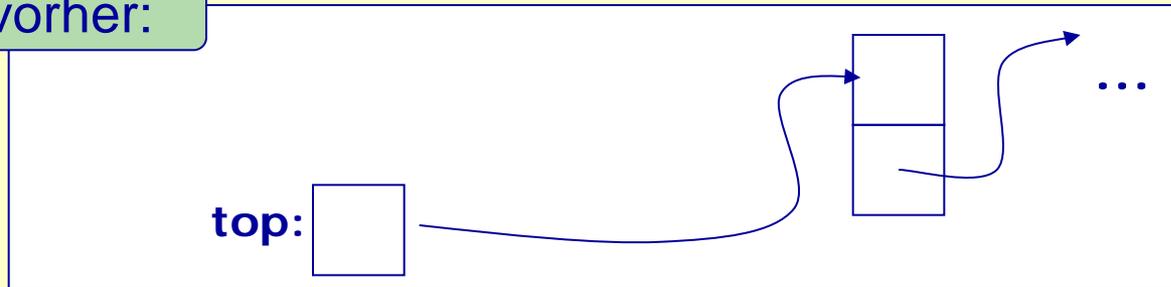
Daten des Stack,
Beginn der Liste



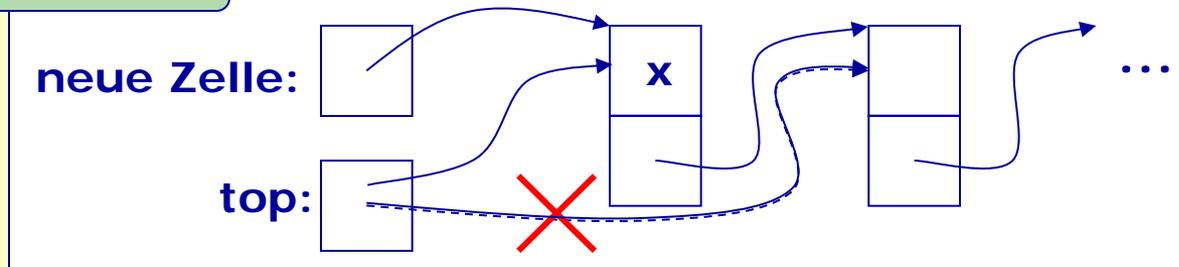
'push': Liste vorn verlängern

```
public void push (Object x) {  
    Zelle neueZelle = new Zelle();  
    neueZelle.inhalt = x;  
    neueZelle.next  = top;  
    top             = neueZelle;  
}
```

vorher:



nachher:

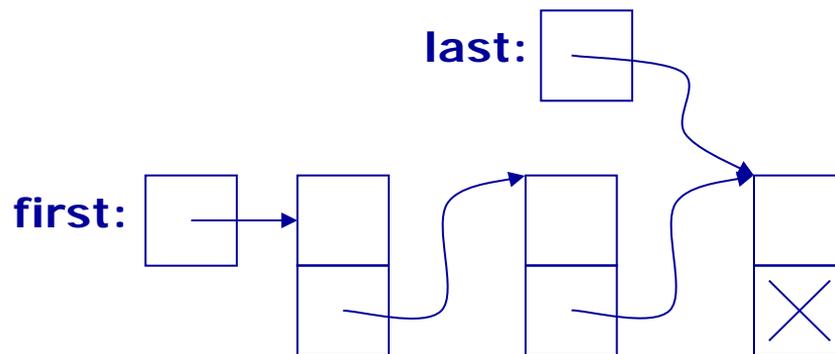


Klasse 'Queue' (Warteschlange):

Prinzip: vorn lesen, hinten anhängen: FIFO

```
class Queue {  
  
    private class Zelle {  
        Object inhalt; //Inhalt  
        Zelle next;    //Verweis  
    }  
  
    // 2 Verweise: erste / letzte Zelle  
  
    private Zelle first, last;
```

Idee?



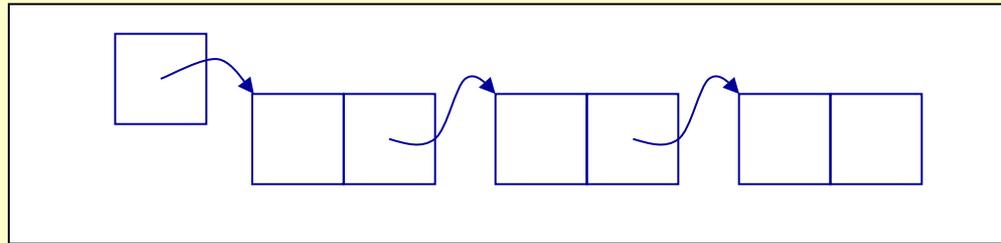
FIFO: First in, First out (Queue)
LIFO: Last in, First out (Stack)

Wo einfügen?
Wo streichen?

Bewertung: Verkettung in Java

- **Verkettete Strukturen:**

- Angelegenheit der Datenstrukturierung



- **Pascal, C, C++ u.v.a.:** mit Datenstrukturen

- Record-Typ + Pointer Typ

- **Java**

- Vermischung von Objektorientierung und Datenstrukturierung:
- Objekterzeugung zur Pointergenerierung genutzt

Missbrauch?

- Klasse: Realisierung ADT
- Grund: kein anderer Weg, da Portabilität oberstes Ziel von Java (Pointer: problematisch für Portabilität - Adressverwaltung)