Stack & Queue

Beispiel: Stack und Queue

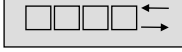
Stack (Kellerspeicher)

push(x); legt x auf den Stack
x = pop(); entfernt und liefert oberstes Stackelement

```
push( 3);   [3]
push( 4);   [3][4]

x =  pop(); [3]        // x == 4
y =  pop();            // y == 4
```

LIFO- Datenstruktur
(last in - first out)



Queue (Puffer, Schlange)

put(x); fügt x hinten an die Queue an
x = get(); entfernt und liefert vorderstes Queueelement

```
put( 3);    [3]
put( 4);    [3][4]

x = get();  [4]        // x == 3
y = get();             // y == 4
```

FIFO- Datenstruktur
(first in - first out)

Klasse Stack

```
class Stack {
   int[] data;
   int top;
   Stack (int size) {
      data = new int[ size];    top = -1;
   }
   void push (int x) {
      if (top >= data. length -1)
          Out. println("-- overflow");
      else
          data[++ top] = x;
   }
   int pop () {
      if (top <= -1) {
          Out. println("-- underflow"); return 0;
      } else
          return data[ top--];
   }
}
```

Benutzung

```
Stack s = new Stack( 10);
s. push( 3);
s. push( 6);
int x = s. pop() + s. pop();    // x == 9
```

Klasse Queue

```
class Queue {
   int[] data;
   int head, tail;
   Queue (int size) {
      data = new int[ size]; head = 0; tail = 0;
   }
   void put (int x) {
      if (( tail+ 1) % data. length == head)
          Out. println("-- overflow");
      else {
          data[ tail] = x;
          tail = (tail+ 1) % data. length;
       }
   }
   int get () {
      if (head == tail) {
          Out. println("-- underflow"); return 0;
      } else
          int x = data[ head];
          head = (head+ 1) % data. length;
          return x;
   }
}
```

Benutzung

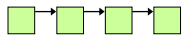
```
Queue q = new Queue( 10);
q. put( 3);
q. put( 6);
int x = q. get();      // x == 3
int y = q. get();      // y == 6
```

Dynamische Datenstrukturen

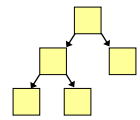
Warum "dynamisch"

- Elemente werden zur Laufzeit (dynamisch) mit *new* angelegt.
- Datenstruktur kann dynamisch wachsen und schrumpfen.

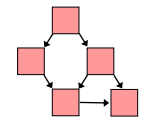
Die wichtigsten dynamischen Datenstrukturen



Liste



Baum

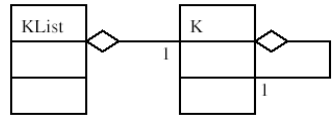


Graph

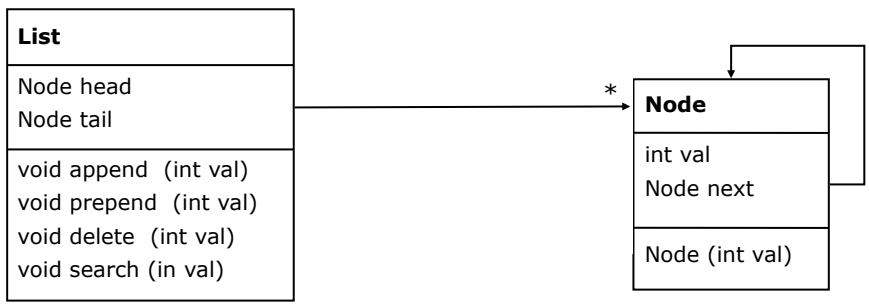
Container-Klassen für Listen

Containerklassen sind für Listen sinnvoll.

- Enthalten Referenz auf Kopf und Ende der Liste.
- Sowie die Methoden.

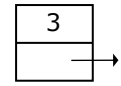


Liste von Knoten



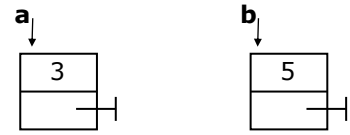
Verknüpfen von Knoten

```
class Node {
    int val;
    Node next;
    Node( int v) {val = v;}
}
```



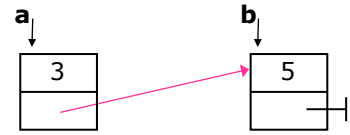
Erzeugen

```
Node a = new Node( 3);
Node b = new Node( 5);
```



Verknüpfen

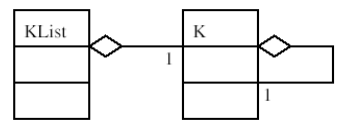
```
a. next = b;
```



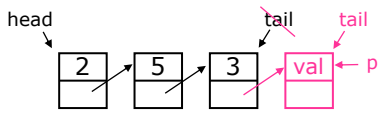
Container-Klassen für Listen

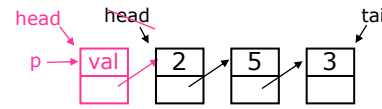
Containerklassen sind für Listen sinnvoll.

- Enthalten Referenz auf Kopf und Ende der Liste.
- Sowie die Methoden.



Unsortierte Liste

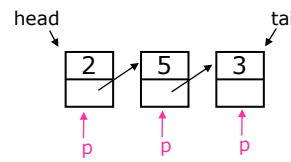


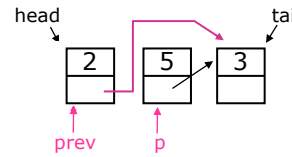



```
class List {
    Node head = null, tail = null;
    void append (int val) { // Einfügen am Listenende
        Node p = new Node( val);
        if (head == null) head = p; else tail. next = p;
        tail = p;
    }
    void prepend (int val) { // Einfügen am Listenanfang
        Node p = new Node( val);
        p. next = head; head = p;
    }
}
```

Unsortierte Liste

```
class List {
    Node head = null, tail = null;
    ...
    Node search (int val) { // Suchen eines Werts
        Node p = head;
        while (p != null && p. val != val) p = p. next;
        // p == null || p. val == val
        return p;
    }
    void delete (int val) { // Löschen eines Werts
        Node p = head, prev = null;
        while (p != null && p. val != val) {
          prev = p;   p = p. next;
        }
        // p == null || p. val == val
        if (p != null)     // p. val == val
            if (p == head)  head = p. next
            else        prev. next = p. next;
            }
        }
}
```





Listen versus Arrays

Listen

- **Vorteile**
  - Einfügen neuer Elemente leicht möglich.
  - Löschen leicht möglich.
- **Nachteile**
  - „Durchhangeln" durch viele Elemente der Liste, um ein Spezielles zu erreichen.

    Etwa das „Fünfte in der Liste"
  - Zusätzlicher Speicherbedarf.

    Eine Referenz pro Listenelement.

Arrays

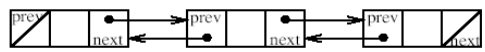
- **Nachteile**
  - Einfügen neuer Elemente erfordert „umkopieren".
  - Löschen von Elementen erfordert ebenfalls ein „umkopieren".
- **Vorteile**
  - Wahlfreier Zugriff.
  - Speicherbedarf nur für Daten.

    Nur insgesamt pro Array noch Speicher-Bedarf für ein length-Feld.

Doppelt verkettete Listen

Doppelt verkettete Listen bestehen aus Listenzellen mit zwei Zeigern

- Ein Zeiger prev auf die vorherige Listenzelle,
- Ein Zeiger next auf die nächste Listenzelle



Doppelt verkettete Listen

Vorteile doppelt verketteter Listen:

- I. a. Einfügen sehr viel schneller möglich.
  - Kein Durchlaufen durch die ganze Liste.

Nachteile doppelt verketteter Listen:

- Höherer Speicherbedarf.
  - Zwei Referenzen pro Listenzelle statt nur einer
- Mehr Aufwand bei Listenmanipulation.
  - Zwei Referenzen sind zu ändern statt nur einer