

11.2 Keller (Stacks)

Operationen:

`boolean isEmpty()` : testet auf Leerheit;
`int pop()` : liefert oberstes Element;
`void push(int x)` : legt x oben auf dem Keller ab;
`String toString()` : liefert eine String-Darstellung.

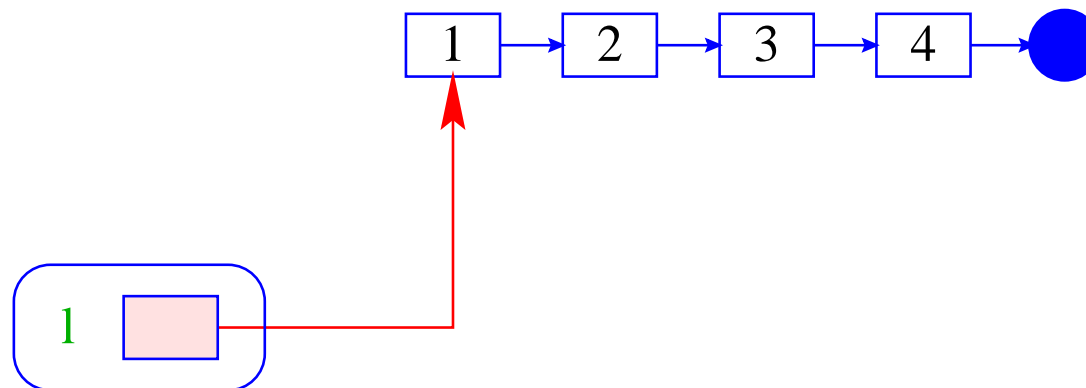
Weiterhin müssen wir einen leeren Keller anlegen können.

Modellierung:

Stack	
+ Stack	()
+ isEmpty	() : boolean
+ push	(x: int) : void
+ pop	() : int

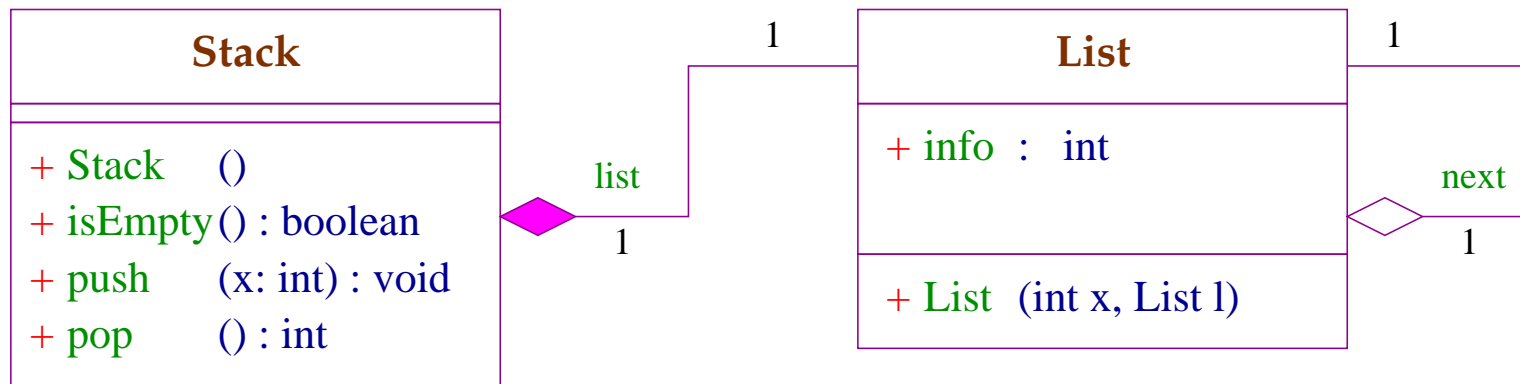
Erste Idee:

- Realisiere Keller mithilfe einer Liste!



- Das Attribut 1 zeigt auf das oberste Element.

Modellierung:



Die **gefüllte Raute** besagt, dass die Liste nur von Stack aus zugreifbar ist :-)

Implementierung:

```
public class Stack {
    private List l;
    // Konstruktor:
    public Stack() {
        l = null;
    }
    // Objekt-Methoden:
    public boolean isEmpty() {
        return l==null;
    }
    ...
}
```

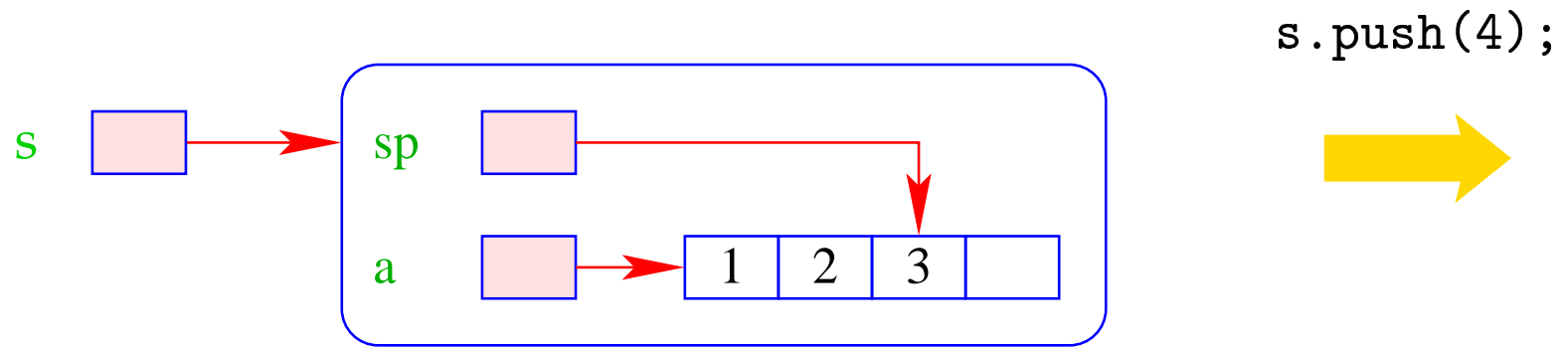
```
public int pop() {
    int result = l.info;
    l = l.next;
    return result;
}
public void push(int a) {
    l = new List(a,l);
}
public String toString() {
    return List.toString(l);
}
} // end of class Stack
```

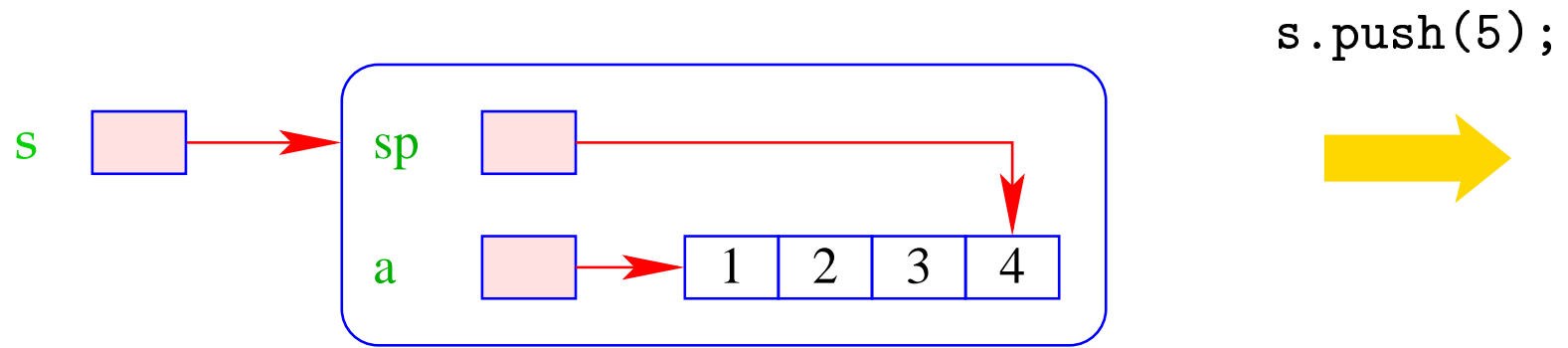
- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von List aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;
 - ⇒ führt zu schlechtem ↑Cache-Verhalten des Programms!

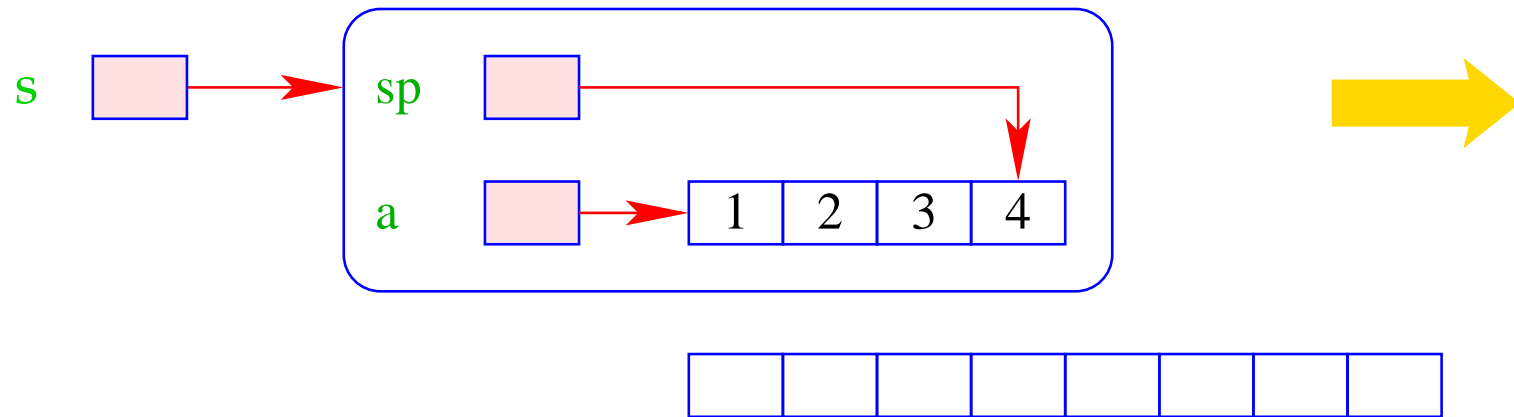
- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von List aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;
 - ⇒ führt zu schlechtem ↑Cache-Verhalten des Programms!

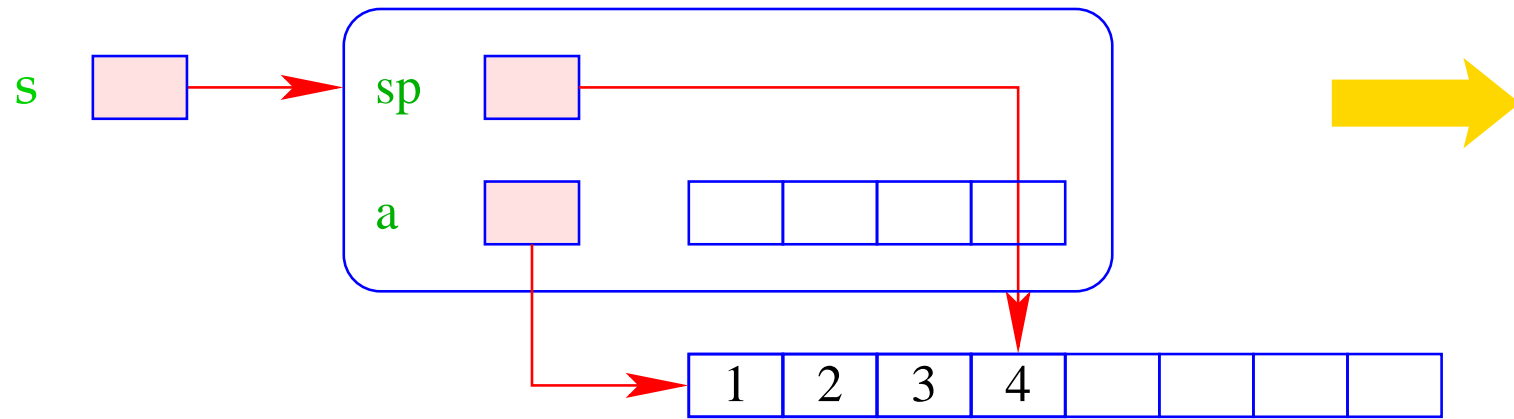
Zweite Idee:

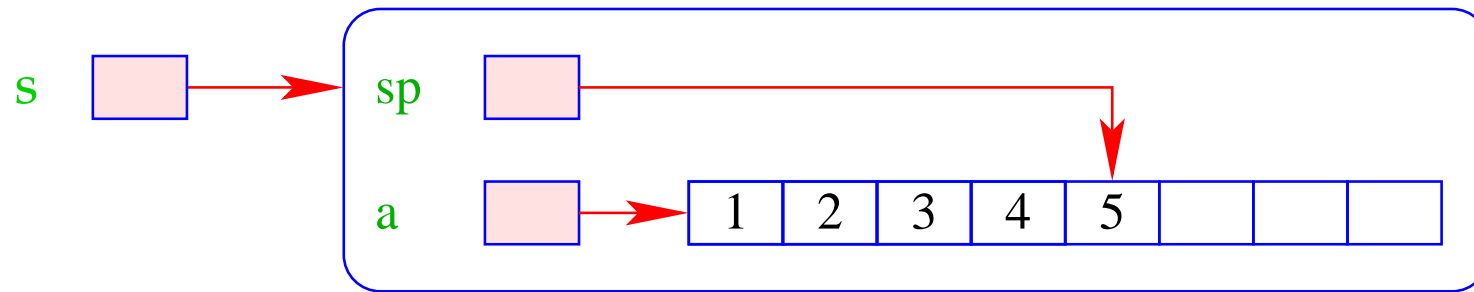
- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres :-)



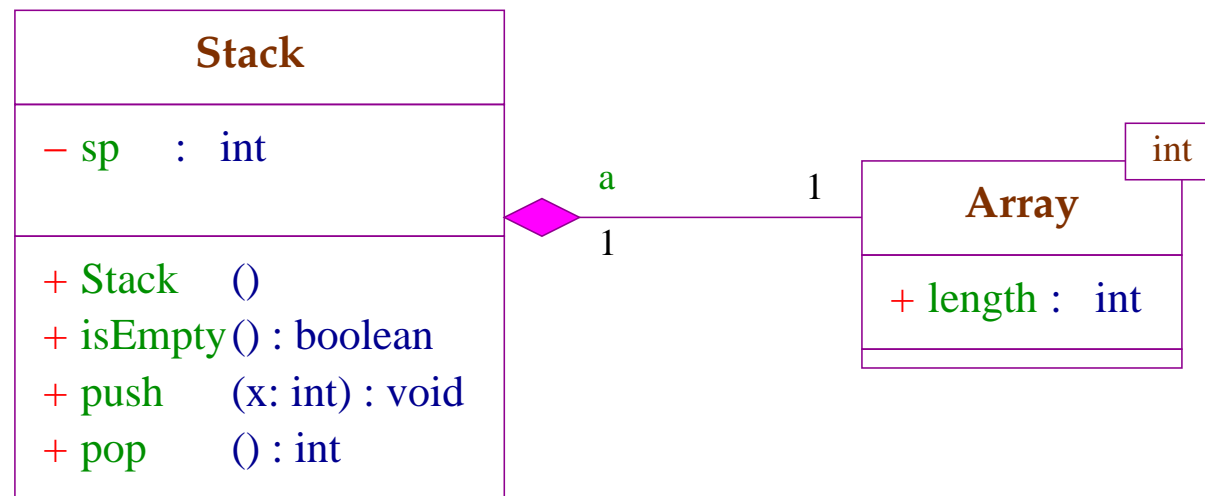








Modellierung:



Implementierung:

```
public class Stack {
    private int sp;
    private int[] a;
    // Konstruktoren:
    public Stack() {
        sp = -1; a = new int[4];
    }
    // Objekt-Methoden:
    public boolean isEmpty() {
        return (sp<0);
    }
    ...
}
```

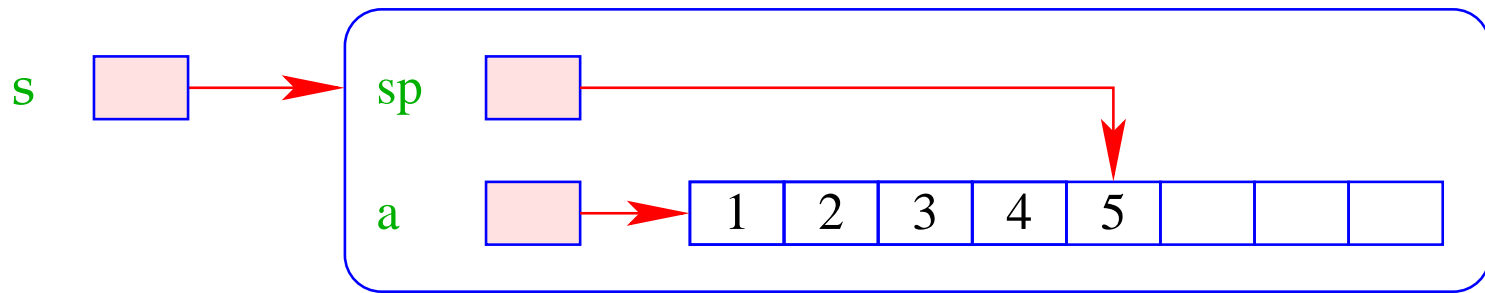
```
public int pop() {
    return a[sp--];
}
public void push(int x) {
    ++sp;
    if (sp == a.length) {
        int[] b = new int[2*sp];
        for(int i=0; i<sp; ++i) b[i] = a[i];
        a = b;
    }
    a[sp] = x;
}
public toString() {...}
} // end of class Stack
```


Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben :-)

Erste Idee:

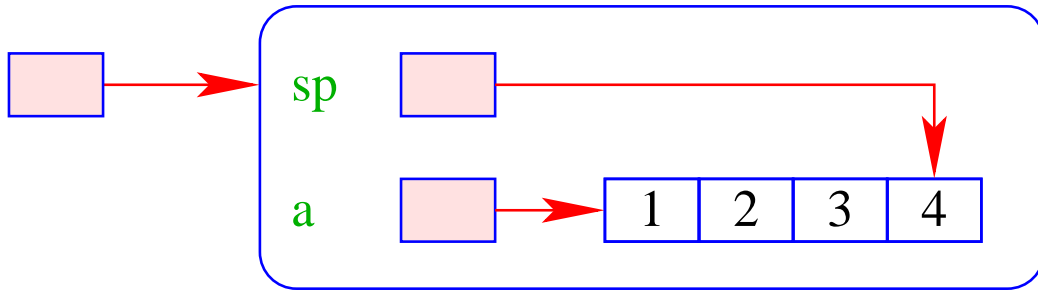
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei ...



`x`

```
x=s.pop();
```

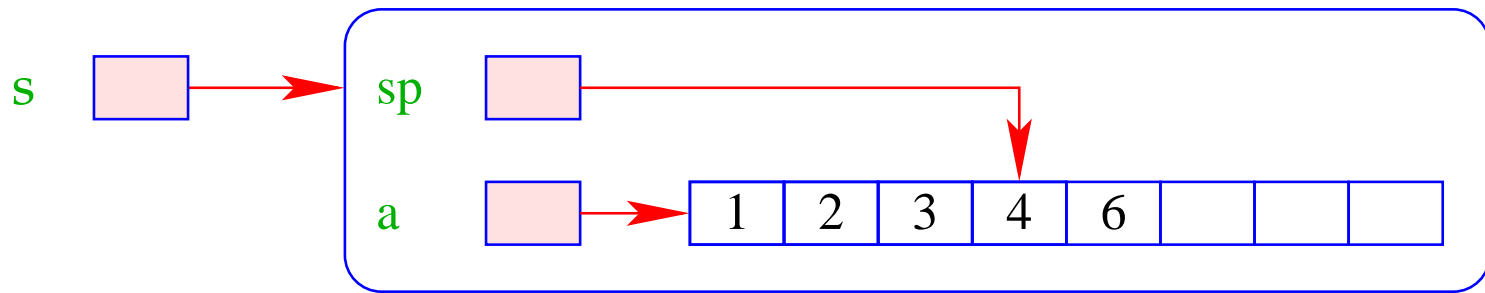




x 5

`s.push(6);`

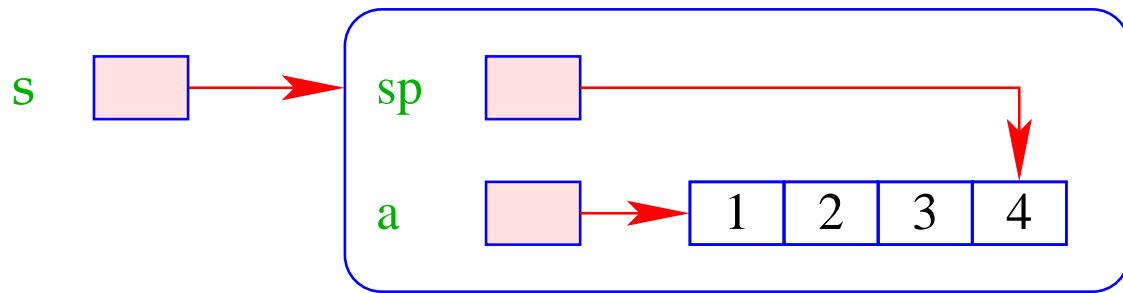




`x` 5

```
x = s.pop();
```

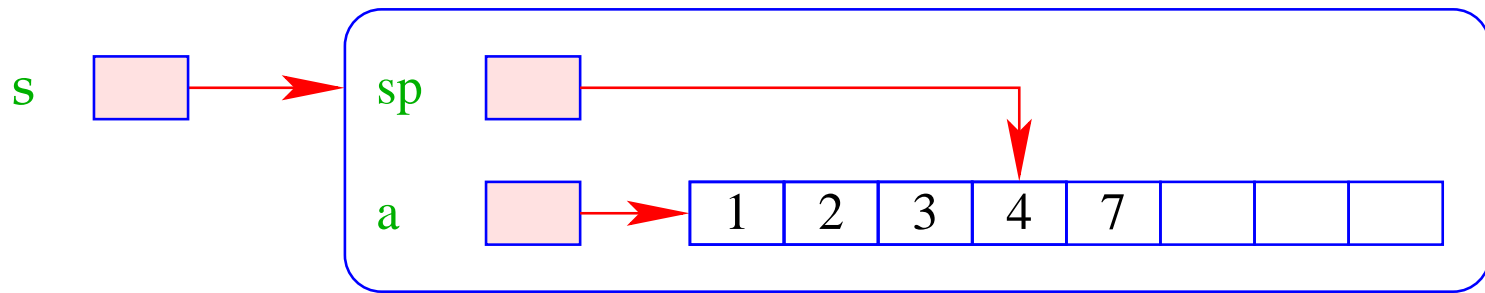




`x` 6

`s.push(7);`





`x` 6

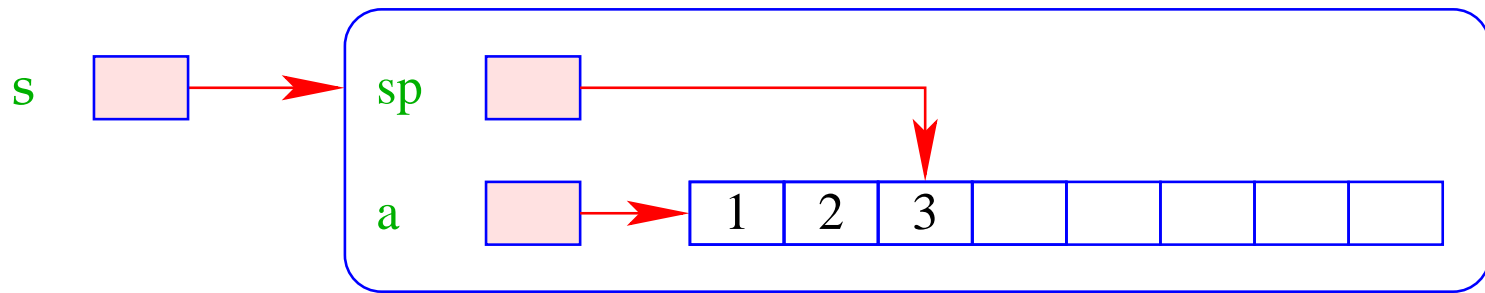
```
x = s.pop();
```



- Im schlimmsten Fall müssen bei **jeder** Operation sämtliche Elemente kopiert werden :-)

Zweite Idee:

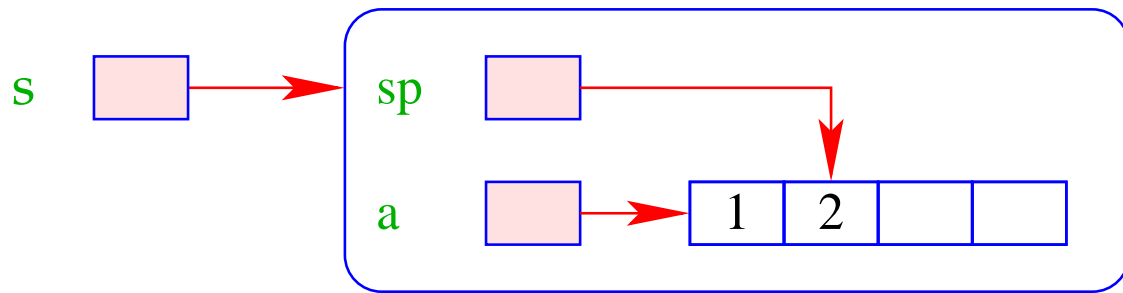
- Wir geben erst frei, wenn der Pegel auf **ein Viertel** fällt – und dann auch nur die Hälfte !



`x`

```
x = s.pop();
```

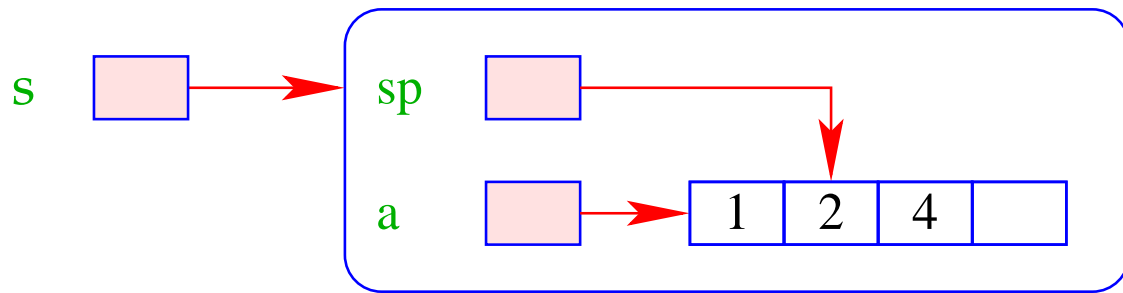




`x` `3`

`s.push(4);`

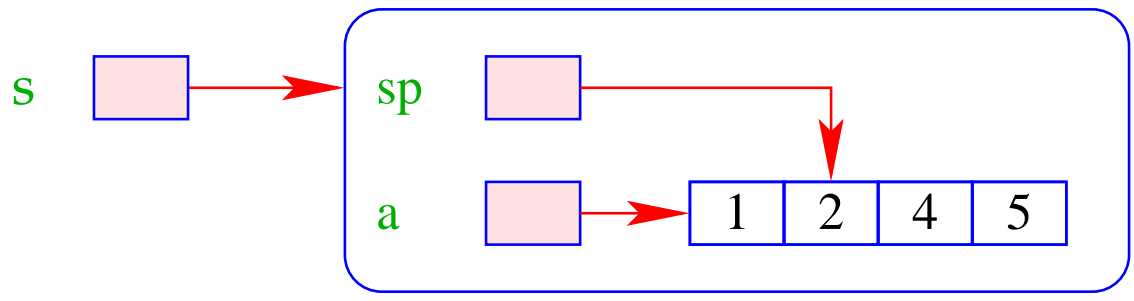




`x` 3

`s.push(5);`





x 3



- Vor jedem Kopieren werden **mindestens** halb so viele Operationen ausgeführt, wie Elemente kopiert werden :-)
- Gemittelt über die gesamte Folge von Operationen werden pro Operation maximal zwei Zahlen kopiert ↑ **amortisierte Aufwandsanalyse**.

```
public int pop() {
    int result = a[sp];
    if (sp == a.length/4 && sp>=2) {
        int[] b = new int[2*sp];
        for(int i=0; i < sp; ++i)
            b[i] = a[i];
        a = b;
    }
    sp--;
    return result;
}
```

11.3 Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem **FIFO**-Prinzip (First-In-First-Out).

Operationen:

`boolean isEmpty()` : testet auf Leerheit;
`int dequeue()` : liefert erstes Element;
`void enqueue(int x)` : reiht x in die Schlange ein;
`String toString()` : liefert eine String-Darstellung.

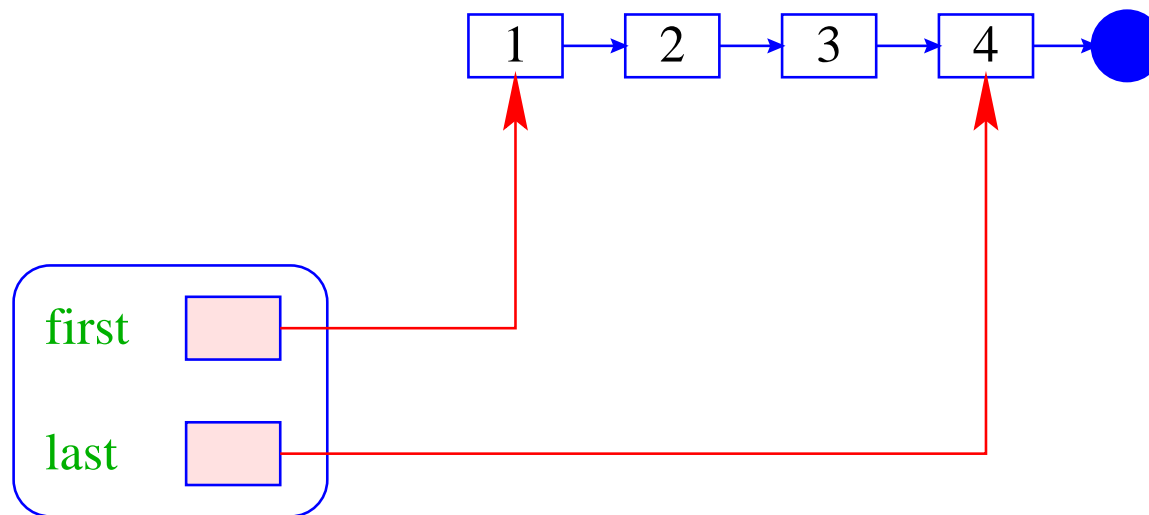
Weiterhin müssen wir eine leere Schlange anlegen können :-)

Modellierung:

Queue
<ul style="list-style-type: none">+ Queue ()+ isEmpty () : boolean+ enqueue (x: int) : void+ dequeue () : int

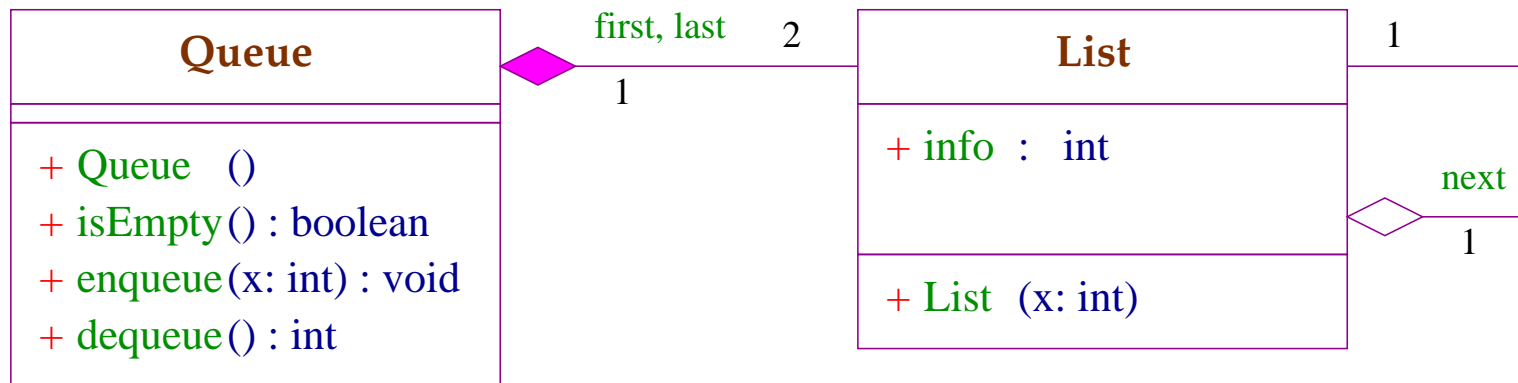
Erste Idee:

- Realisiere Schlange mithilfe einer Liste :



- `first` zeigt auf das nächste zu entnehmende Element;
- `last` zeigt auf das Element, hinter dem eingefügt wird.

Modellierung:



Objekte der Klasse Queue enthalten **zwei** Verweise auf Objekte der Klasse List :-)

Implementierung:

```
public class Queue {
    private List first, last;
    // Konstruktor:
    public Queue () {
        first = last = null;
    }
    // Objekt-Methoden:
    public boolean isEmpty() {
        return first==null;
    }
    ...
}
```

```
public int dequeue () {
    int result = first.info;
    if (last == first) last = null;
    first = first.next;
    return result;
}

public void enqueue (int x) {
    if (first == null) first = last = new List(x);
    else { last.next = new List(x); last = last.next; }
}

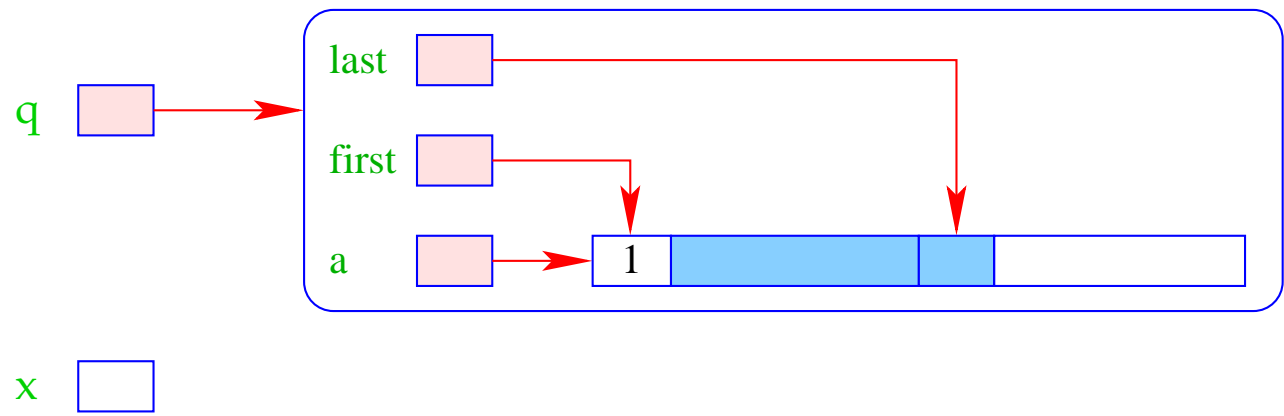
public String toString() {
    return List.toString(first);
}
} // end of class Queue
```

- Die Implementierung ist wieder sehr einfach :-)
- ... nutzt ebenfalls kaum Features von List aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut
 - ⇒ führt zu schlechtem ↑Cache-Verhalten des Programms :-)

- Die Implementierung ist wieder sehr einfach :-)
- ... nutzt ebenfalls kaum Features von List aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut
 - ⇒ führt zu schlechtem ↑Cache-Verhalten des Programms :-)

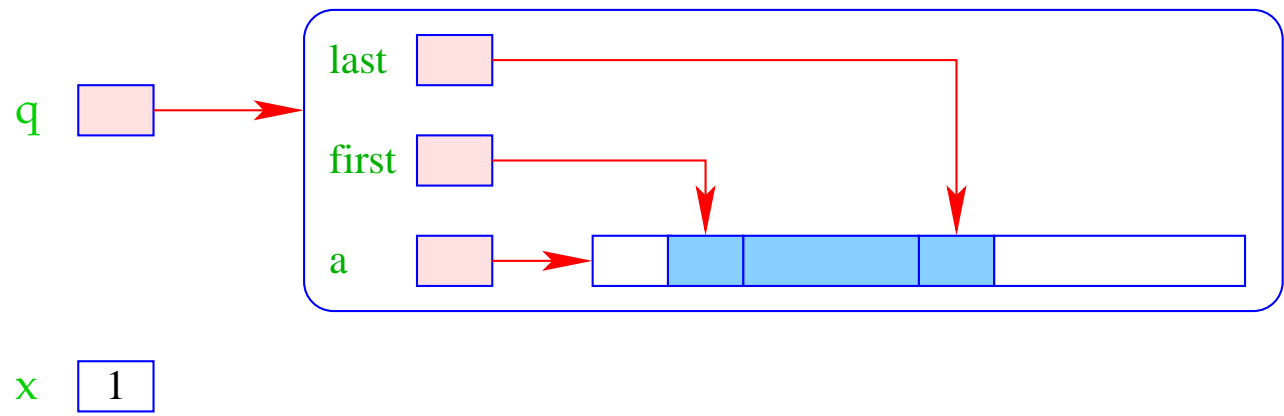
Zweite Idee:

- Realisiere die Schlange mithilfe eines Felds und **zweier** Pointer, die auf das erste bzw. letzte Element der Schlange zeigen.
- Läuft das Feld über, ersetzen wir es durch ein größeres.



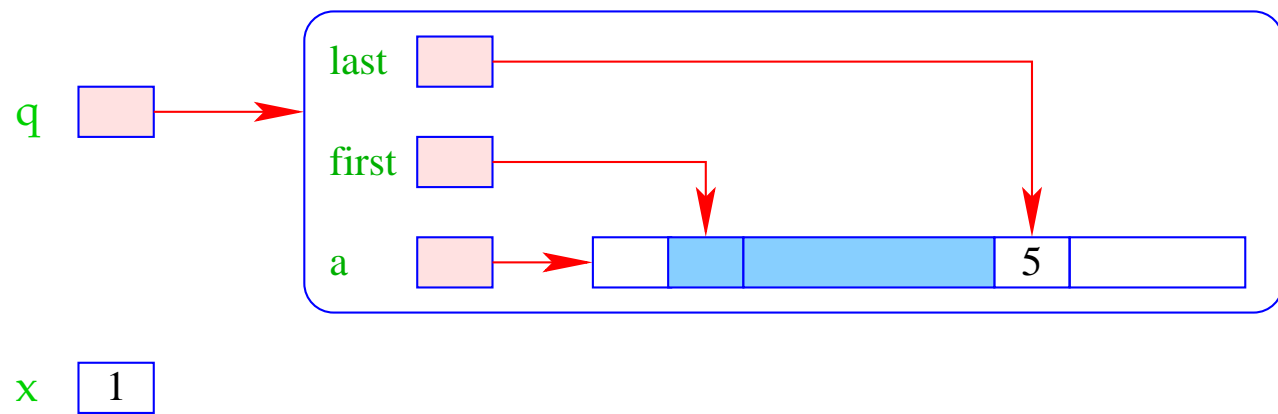
`x = q.dequeue();`

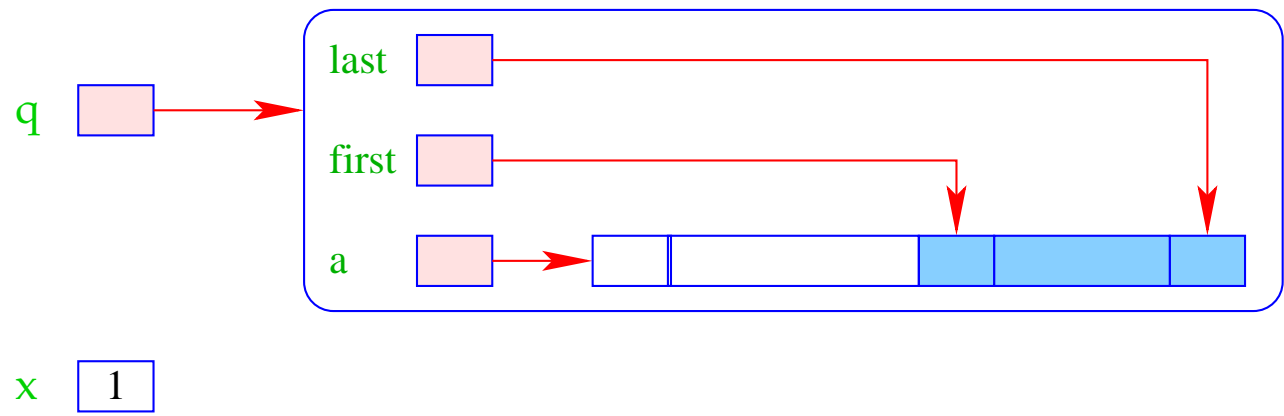




```
x = q.enqueue(5);
```

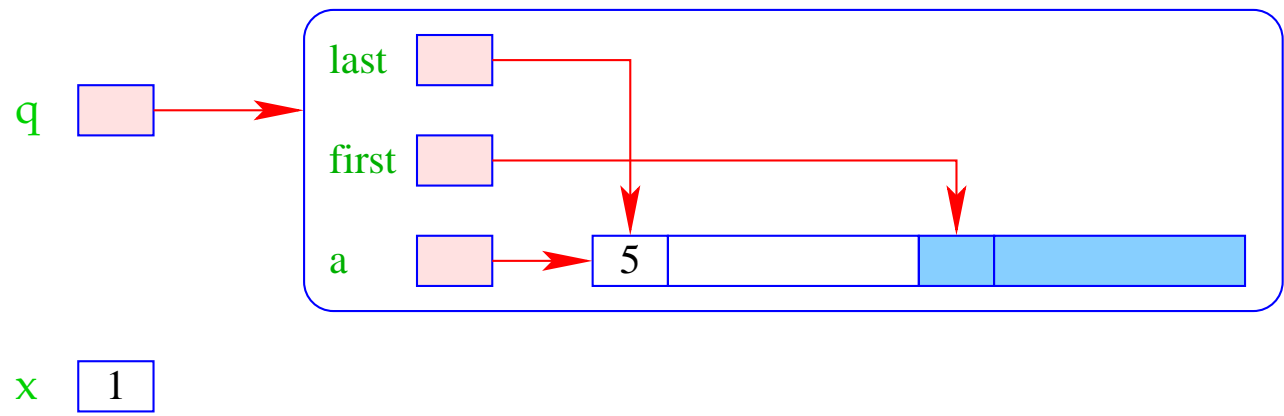




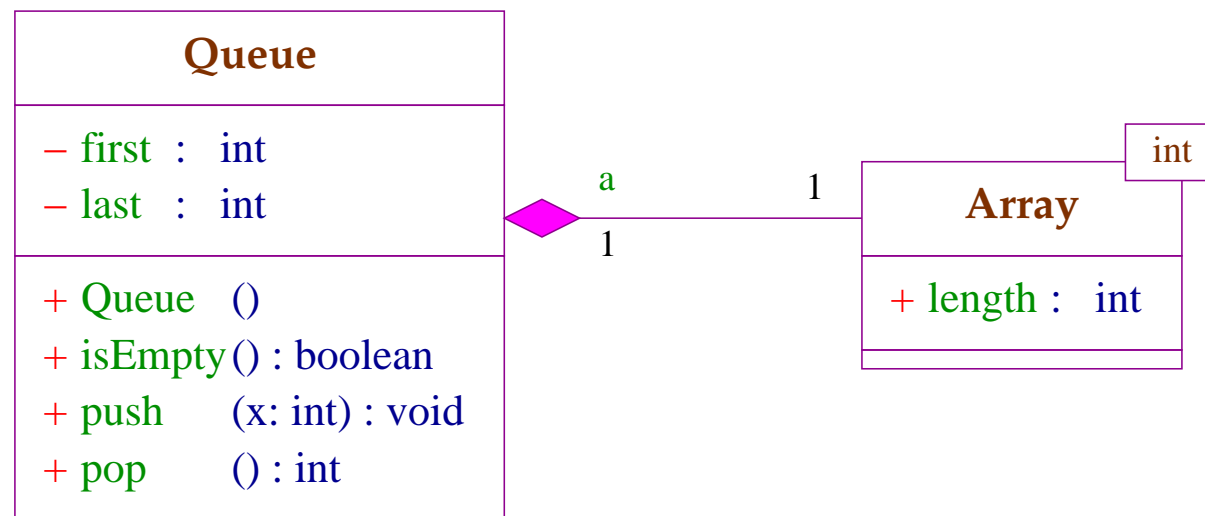


```
x = q.enqueue(5);
```





Modellierung:



Implementierung:

```
public class Queue {
    private int first, last;
    private int[] a;
    // Konstruktor:
    public Queue () {
        first = last = -1;
        a = new int[4];
    }
    // Objekt-Methoden:
    public boolean isEmpty() { return first==-1; }
    public String toString() {...}
    ...
}
```

Implementierung von `enqueue()`:

- Falls die Schlange leer war, muss `first` und `last` auf 0 gesetzt werden.
- Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- In diesem Fall legen wir ein Feld doppelter Größe an.
Die Elemente `a[first]`, `a[first+1]`, ..., `a[a.length-1]`,
`a[0]`, `a[1]`, ..., `a[first-1]` kopieren wir nach `b[0]`, ...,
`b[a.length-1]`.
- Dann setzen wir `first = 0`; , `last = a.length` und `a = b`;
- Nun kann `x` an der Stelle `a[last]` abgelegt werden.

```

public void enqueue (int x) {
    if (first==-1) {
        first = last = 0;
    } else {
        int n = a.length;
        last = (last+1)%n;
        if (last == first) {
            int[] b = new int[2*n];
            for (int i=0; i<n; ++i) {
                b[i] = a[(first+i)%n];
            } // end for
            first = 0; last = n; a = b;
        } // end if and else
        a[last] = x;
    }
}

```

Implementierung von dequeue() :

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden `first` und `last` auf `-1` gesetzt.
- Andernfalls wird `first` um 1 (modulo der Länge von `a`) inkrementiert ...

```
public int dequeue () {
    int result = a.first;
    if (first == last) first = last = -1;
    else first = (first+1) % a.length;
    return result;
}
```

Diskussion:

- In dieser Implementierung von `dequeue()` wird der Platz für die Schlange nie verkleinert ...
- Fällt die Anzahl der Elemente in der Schlange unter ein Viertel der Länge des Felds `a`, können wir aber (wie bei Kellern) das Feld durch ein halb so großes ersetzen :-)

Achtung:

Die Elemente in der Schlange müssen aber jetzt nicht mehr nur am Anfang von `a` liegen !!!

Zusammenfassung:

- Der Datentyp List ist nicht sehr **abstract**, dafür extrem flexibel
 \implies gut geeignet für **rapid prototyping**.
- Für die **nützlichen** (eher **:-**) abstrakten Datentypen Stack und Queue lieferten wir zwei Implementierungen:

Technik	Vorteil	Nachteil
List	einfach	nicht-lokal
int[]	lokal	etwas komplexer

- **Achtung:** oft werden bei diesen Datentypen noch weitere Operationen zur Verfügung gestellt **:-)**