Diskussion:

- Sei $V(n)$ die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge n benötigt.

  Dann gilt:

$$\begin{array}{rcl} V(1) & = & 0 \\ V(2n) & \leq & 2 \cdot V(n) + 2 \cdot n \end{array}$$

- Für $n = 2^k$, sind das dann nur $k \cdot n$ Vergleiche !!!

Achtung:

- Unsere Funktion `sort()` zerstört ihr Argument ?!
- Alle Listen-Knoten der Eingabe werden weiterverwendet :-)
- Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden (wie ?-)
- Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ impliementieren (wie ?-))

12 Abstrakte Datentypen

- Spezifiziere nur die Operationen!
- Verberge Details
  - der Datenstruktur;
  - der Implementierung der Operationen.

$\Longrightarrow$ Information Hiding

Sinn:

- Verhindern illegaler Zugriffe auf die Datenstruktur;
- Entkopplung von Teilproblemen für
  - Implementierung, aber auch
  - Fehlersuche und
  - Wartung;
- leichter Austausch von Implementierungen ($\uparrow$rapid prototyping).

12.1 Beispiel 1: 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.

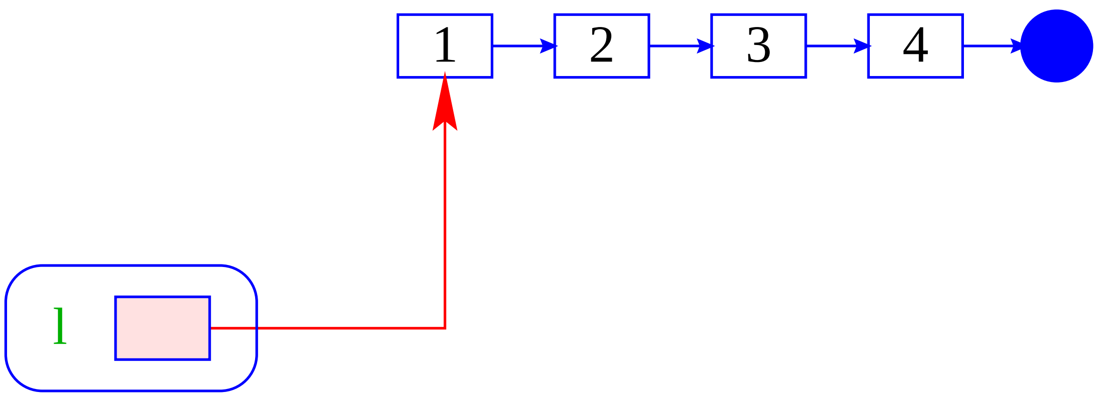
Friedrich Ludwig Bauer, TUM

Modellierung:

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

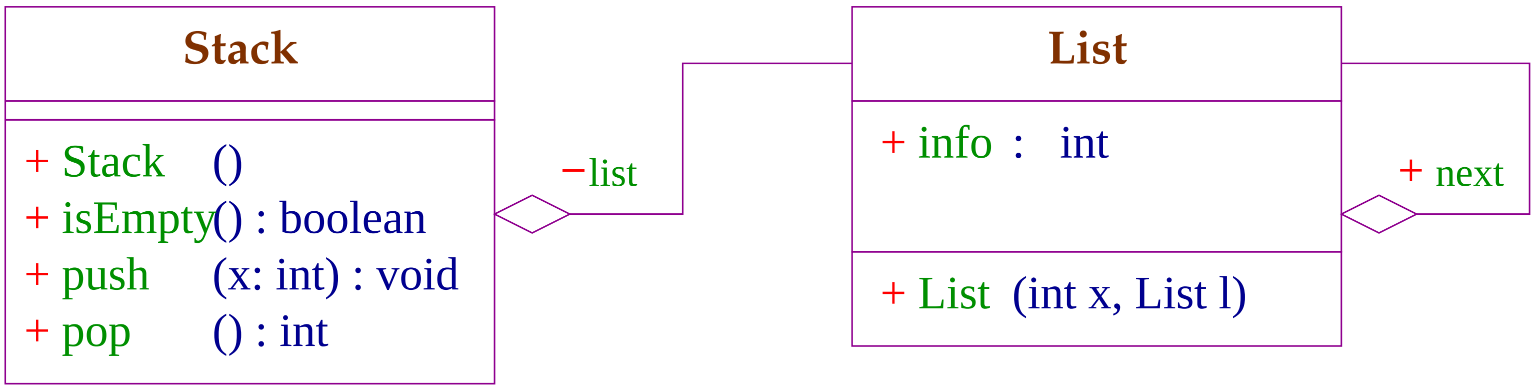
Erste Idee:

- Realisiere Keller mithilfe einer Liste!



- Das Attribut `l` 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 list;
   // Konstruktor:
   public Stack() {
      list = null;
      }
   // Objekt-Methoden:
   public boolean isEmpty() {
      return list==null;
      }
   ...
```

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

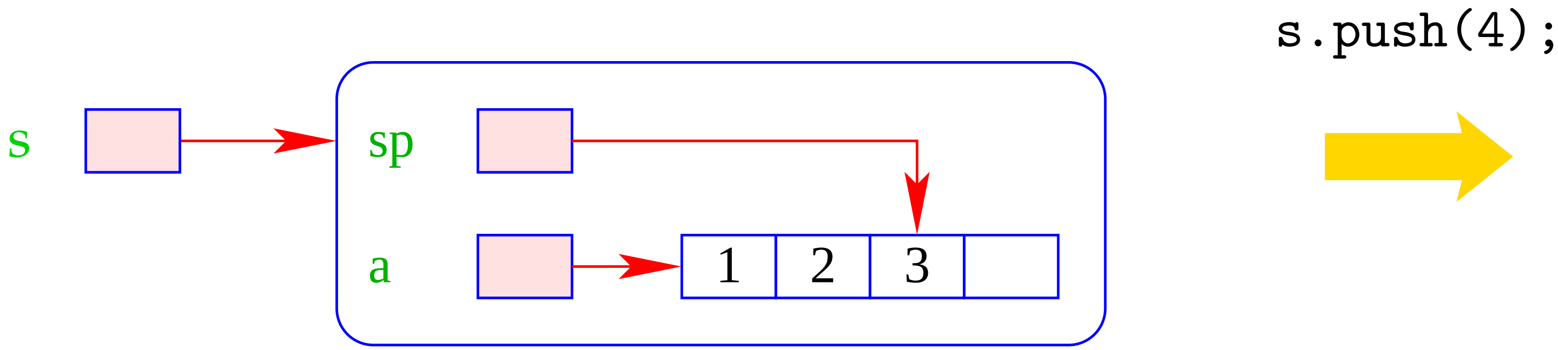
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von `List` aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;

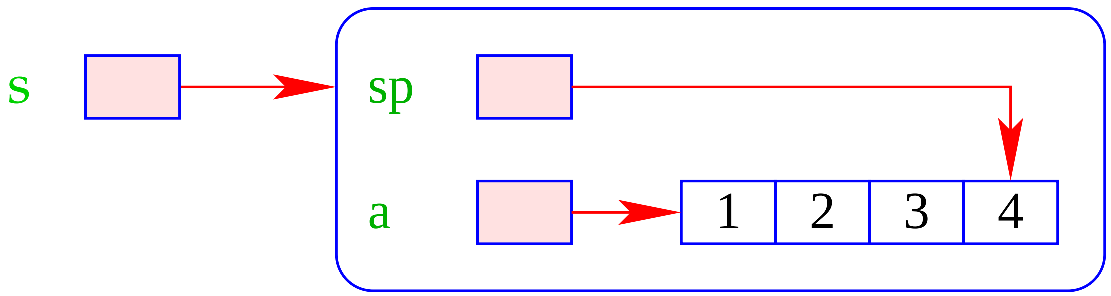
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

Zweite Idee:

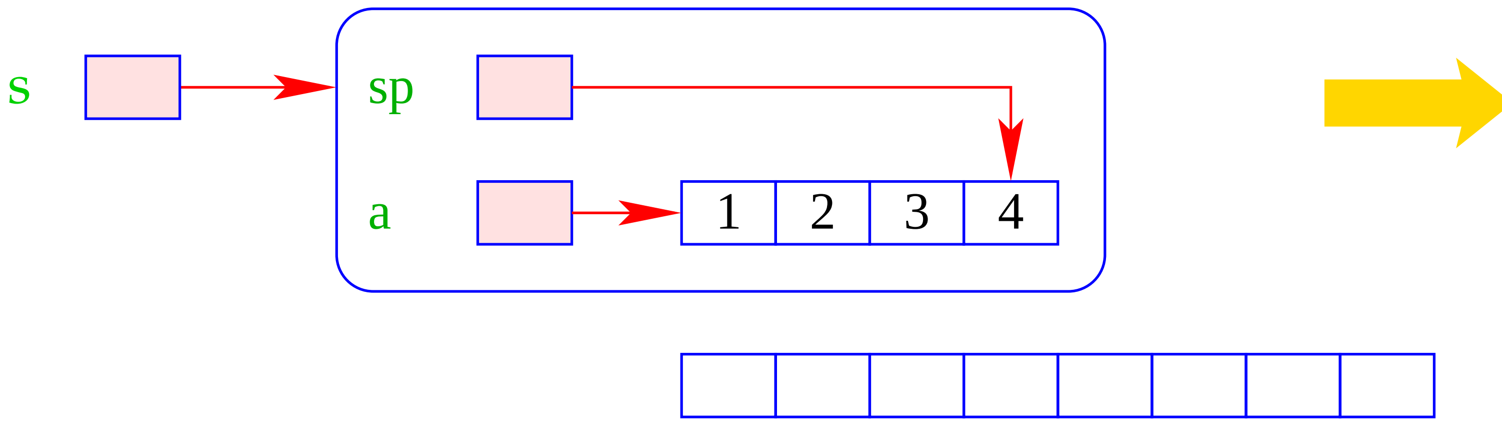
- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres :-)

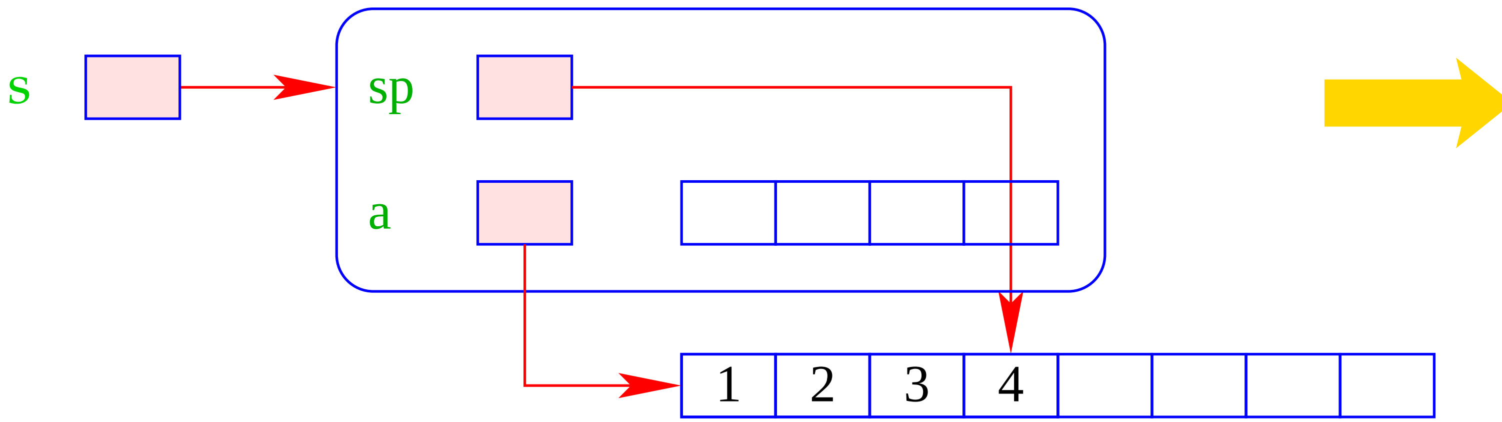


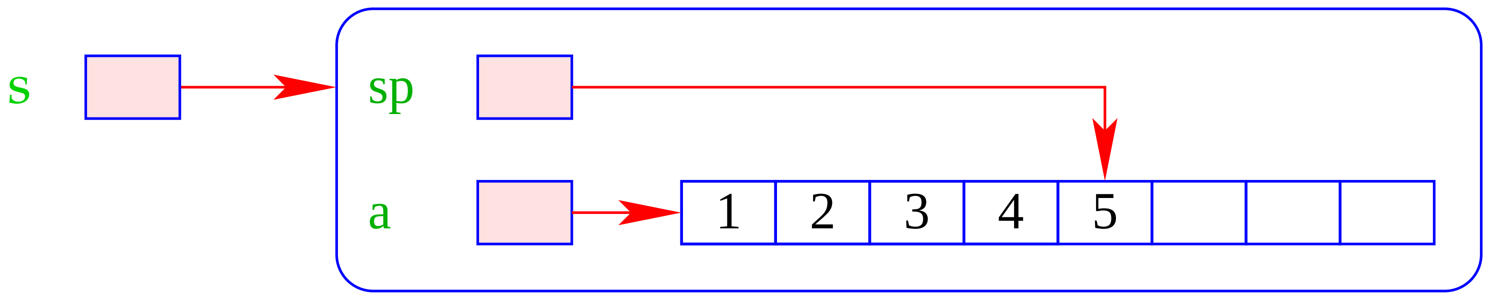
s
sp
a
1
2
3
s.push(4);



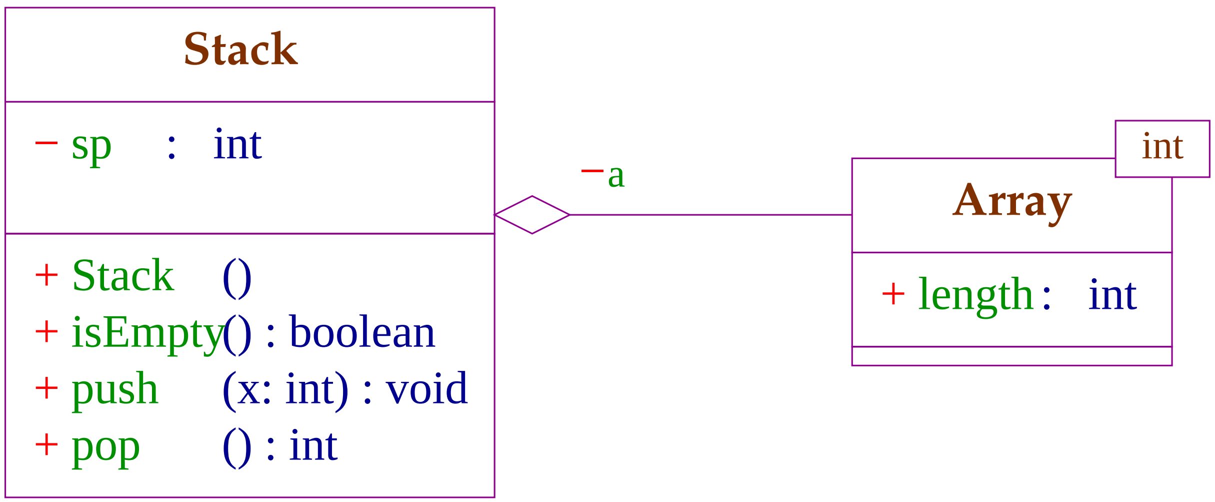
`s.push(5);`


s
sp
a
1
2
3
4


s
sp
a
1
2
3
4


s
sp
a
1
2
3
4
5

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);
      }
   ...
```

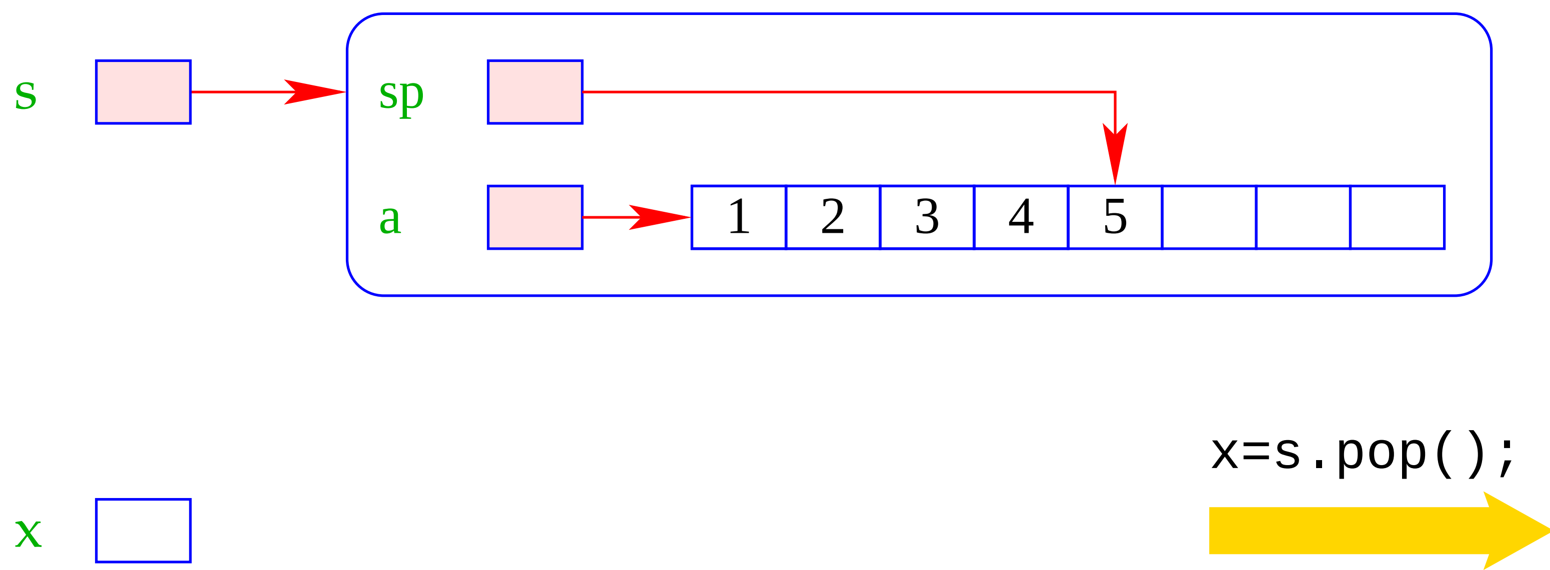
```
    public int pop() {
        return a[sp--];
        }
    public void push(int x) {
        ++sp;
        if (sp == a.length) {
            int[] b = new int[2*sp];
            for(int i=0; i<sp; ++i) b[i] = a[i];
            a = b;
            }
        a[sp] = x;
        }
    public toString()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

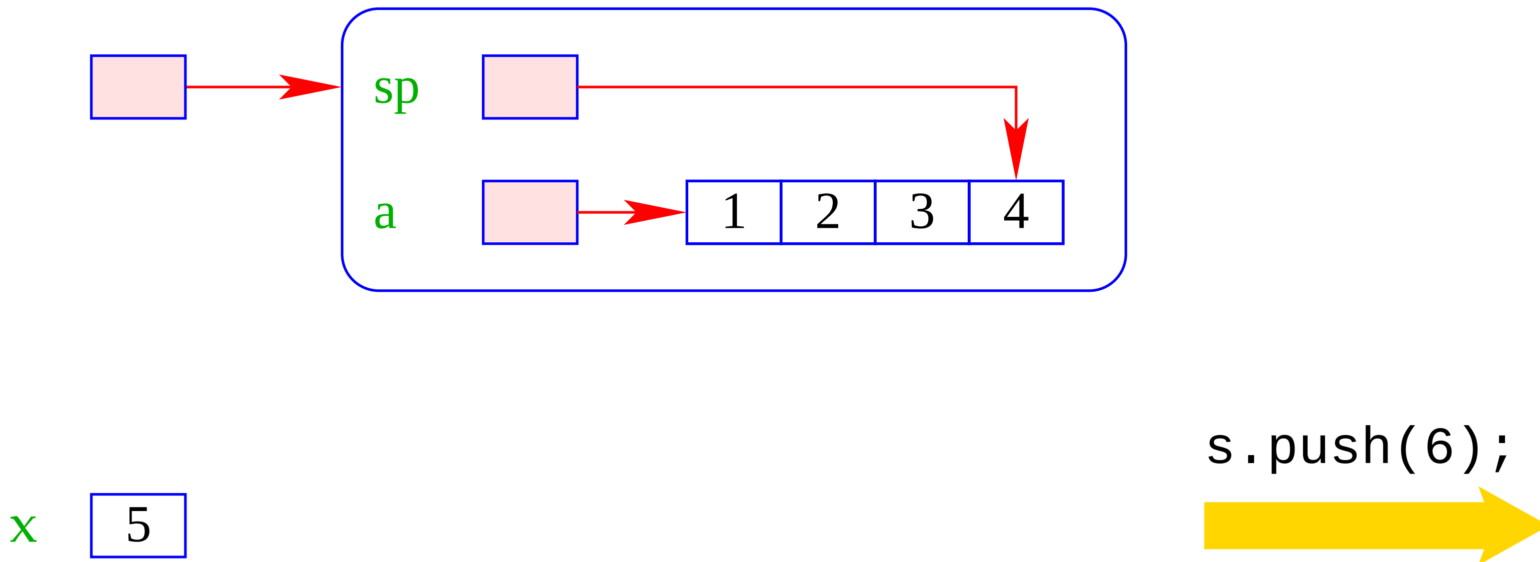
Nachteil:

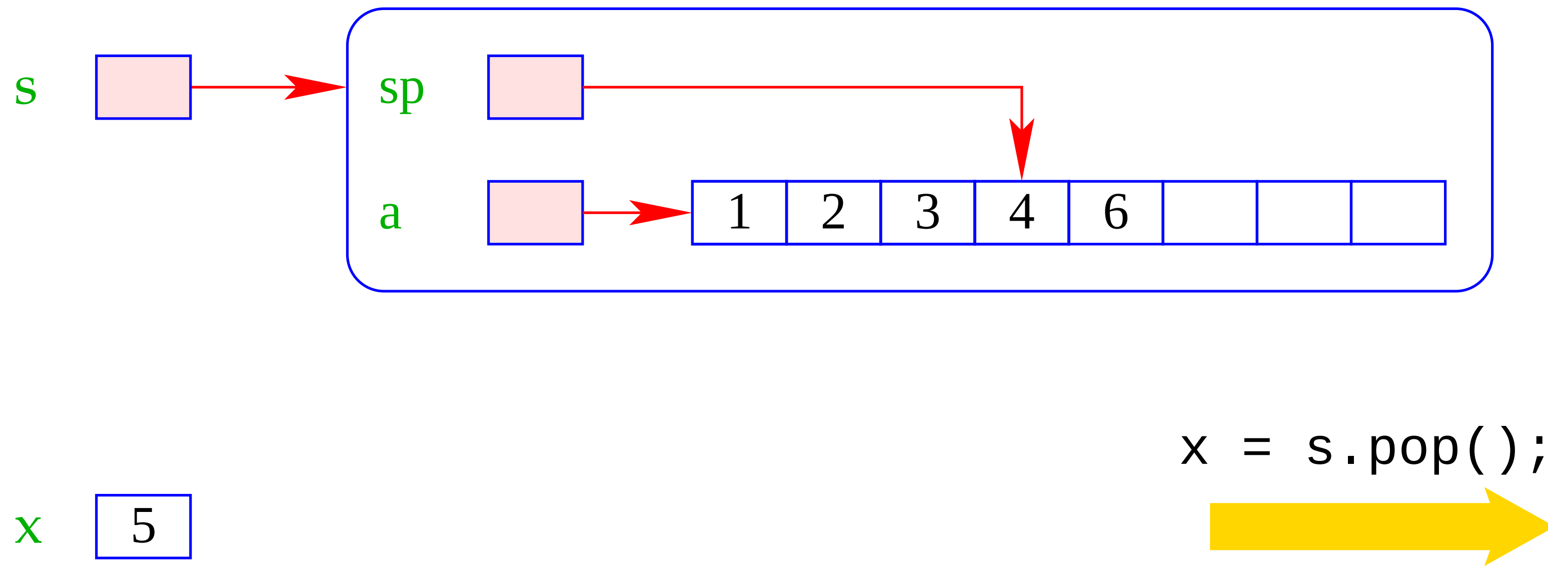
- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben :-(

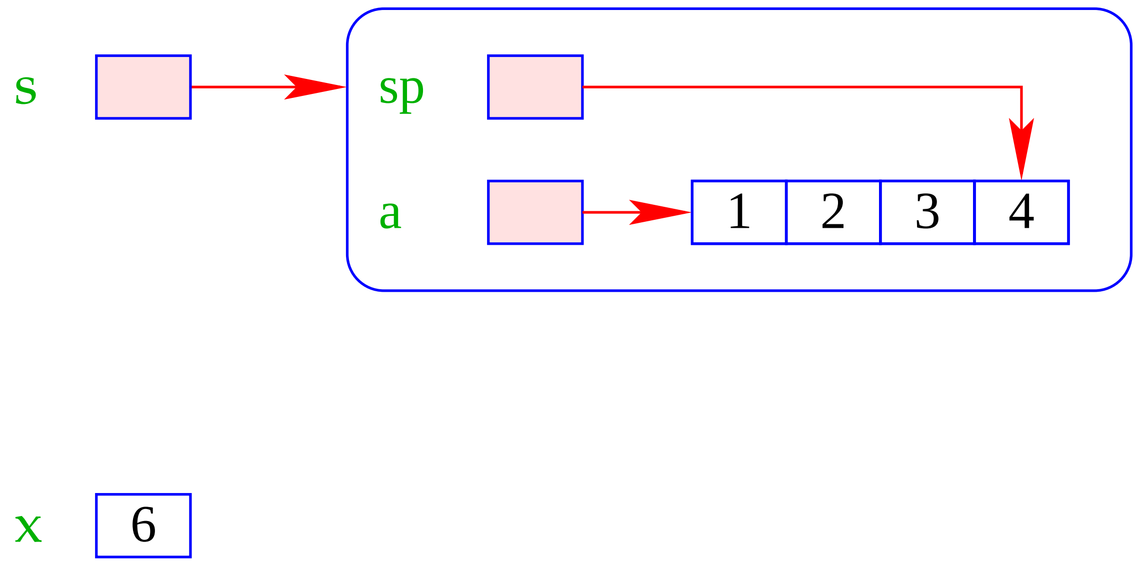
Erste Idee:

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

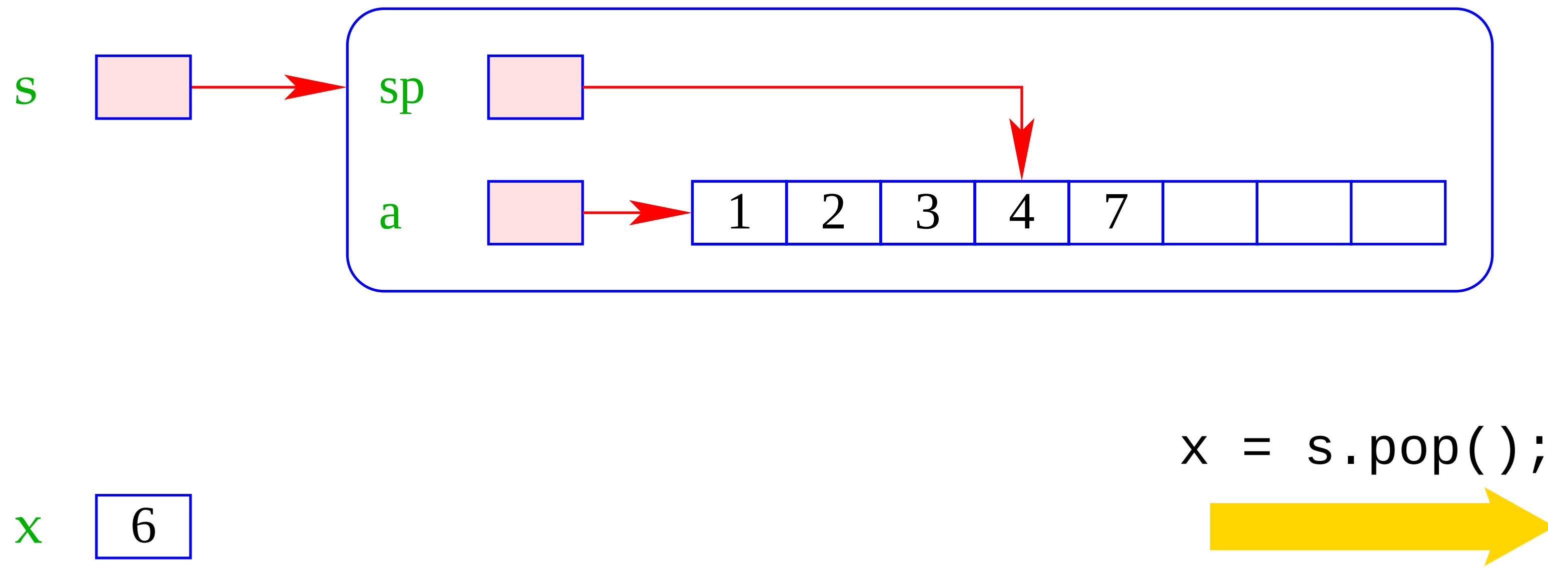

s
sp
a
1
2
3
4
5
x
x=s.pop();


sp
a
1
2
3
4
s.push(6);
x
5


s
sp
a
1
2
3
4
6
x
5
x = s.pop();



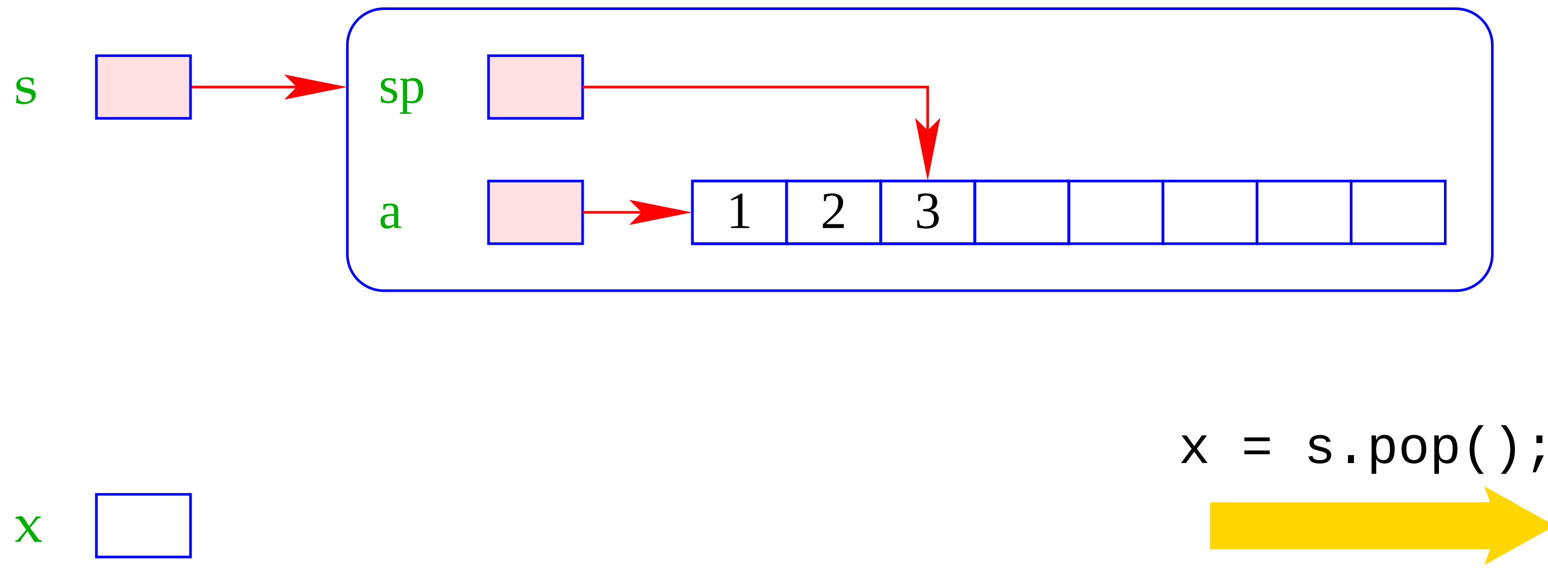
`s.push(7);`


s
sp
a
1
2
3
4
7
x
6
x = s.pop();

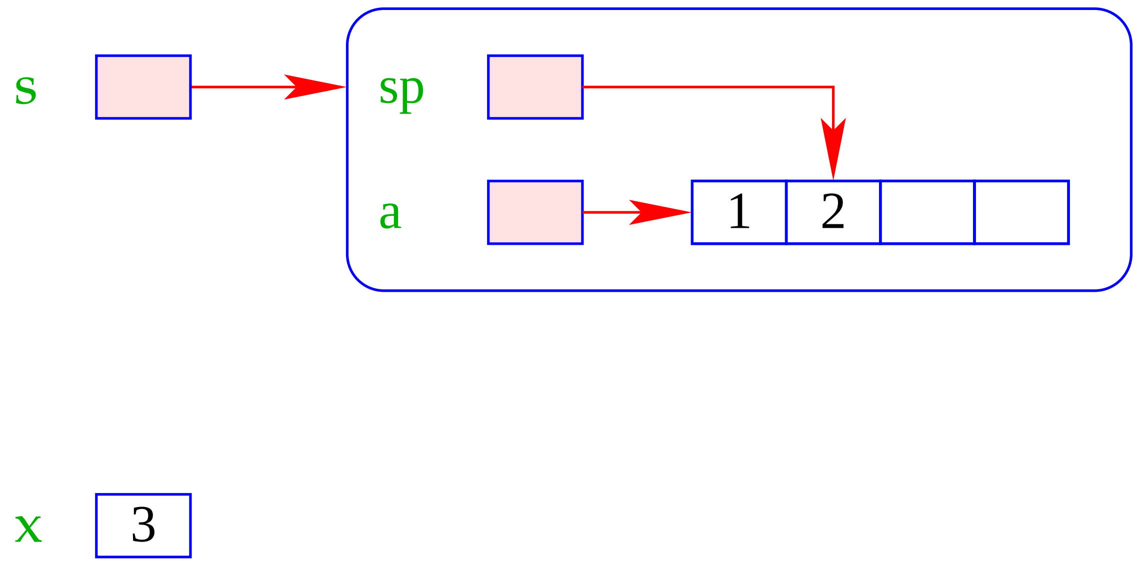
- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden :-(

Zweite Idee:

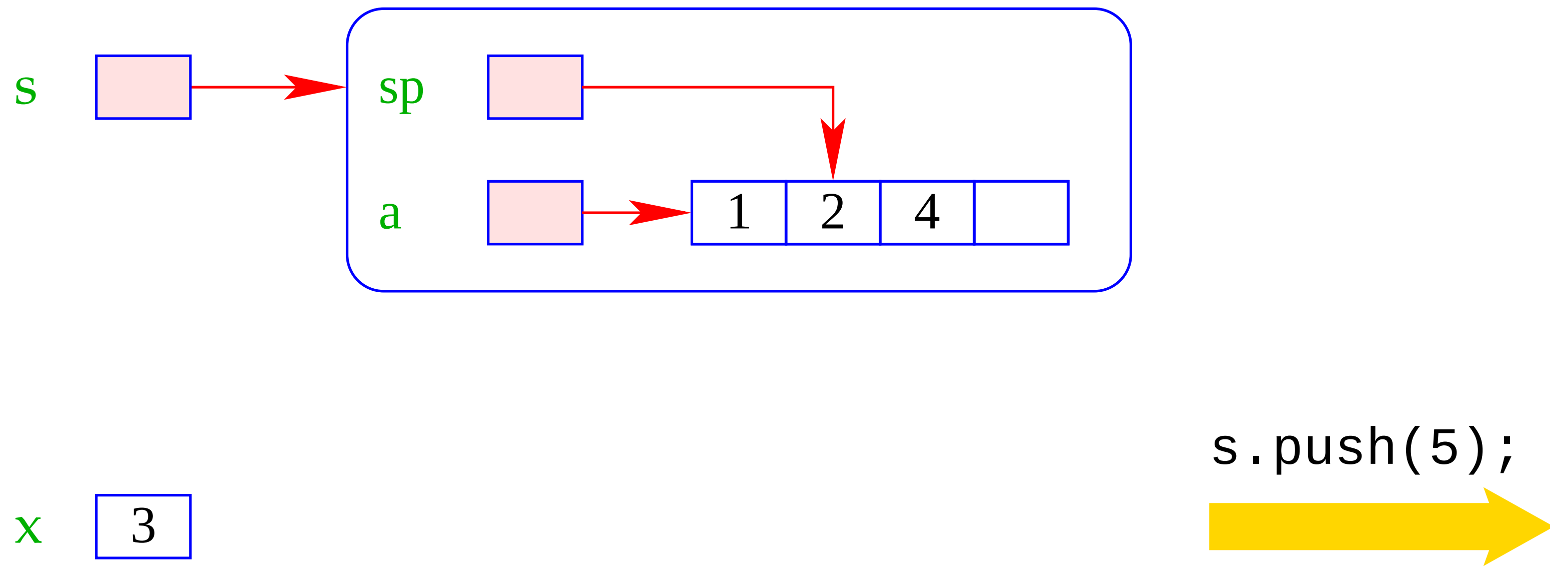
- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte !

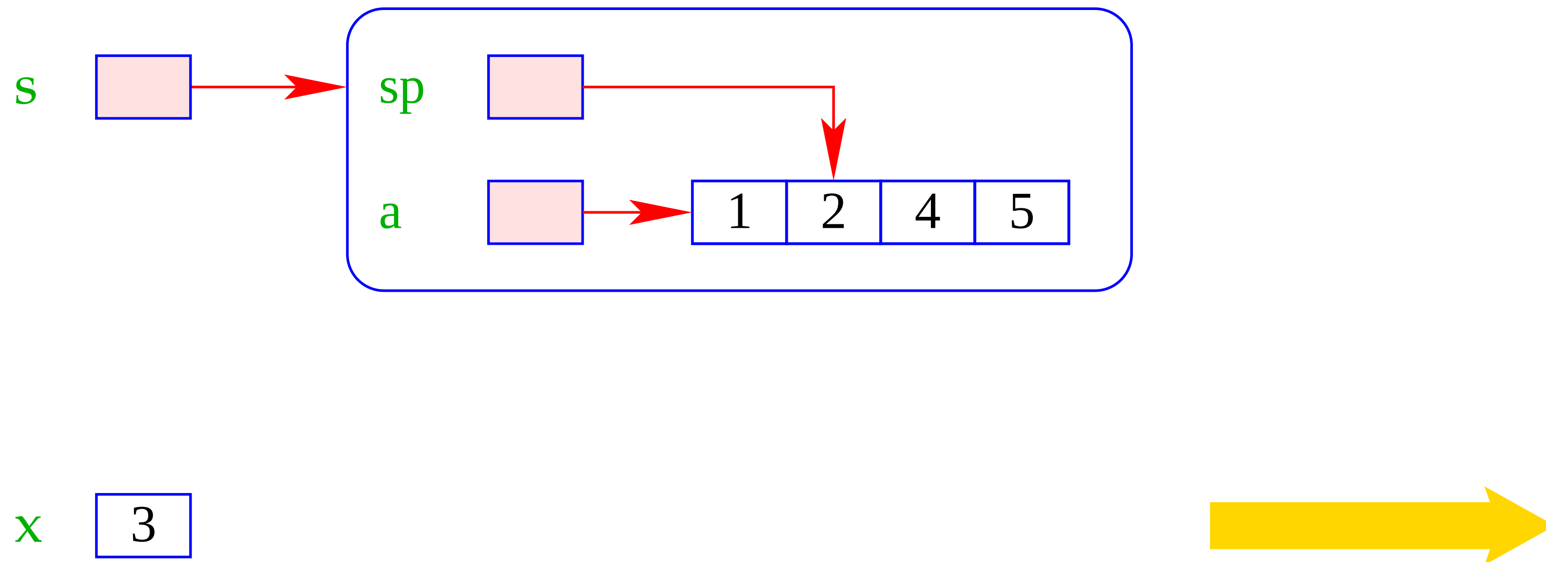



```
x = s.pop();
```



s.push(4);


s
sp
a
1
2
4
x
3
s.push(5);


s
sp
a
1
2
4
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;
   }
```

12.2 Beispiel 2: Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem FIF0-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.

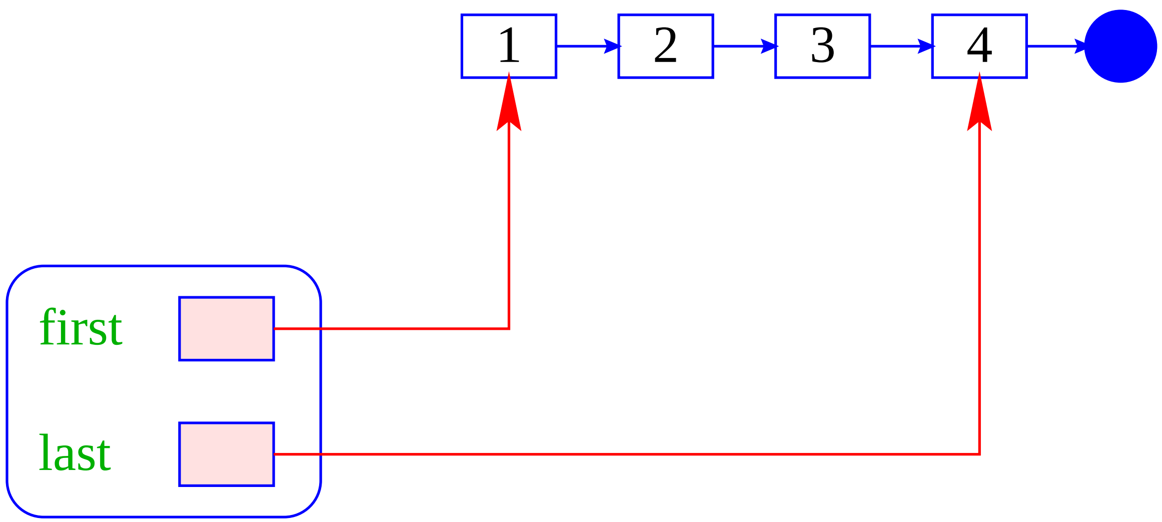
Weiterhin müssen wir eine leere Schlange anlegen können :-)

Modellierung:

Queue
+ Queue ()   + isEmpty() : boolean   + enqueue(x: int) : void   + dequeue() : int

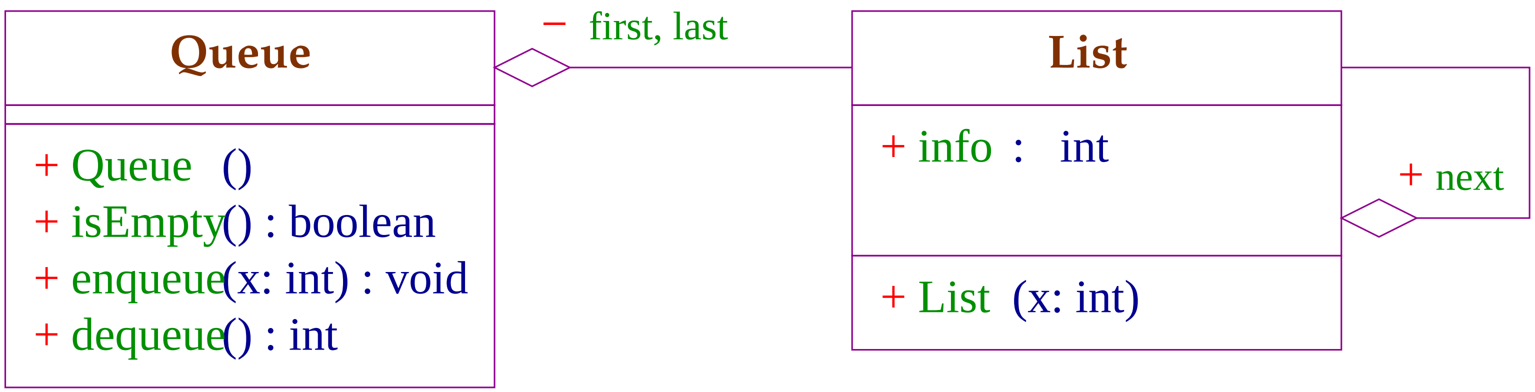
Erste Idee:

- 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

  $\Longrightarrow$ 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

  $\Longrightarrow$ 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.