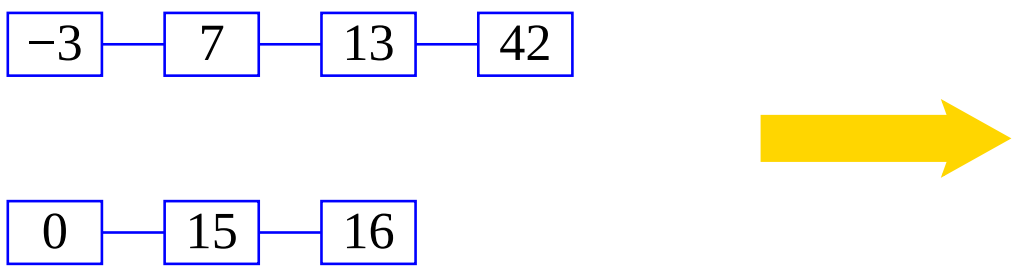
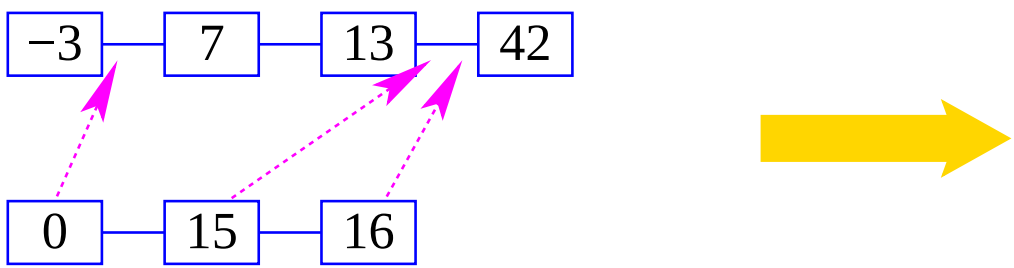
Anwendung: Mergesort – Sortieren durch Mischen

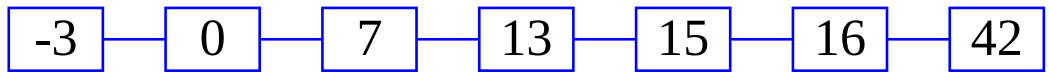
Mischen:

Eingabe: zwei sortierte Listen;

Ausgabe: eine gemeinsame sortierte Liste.

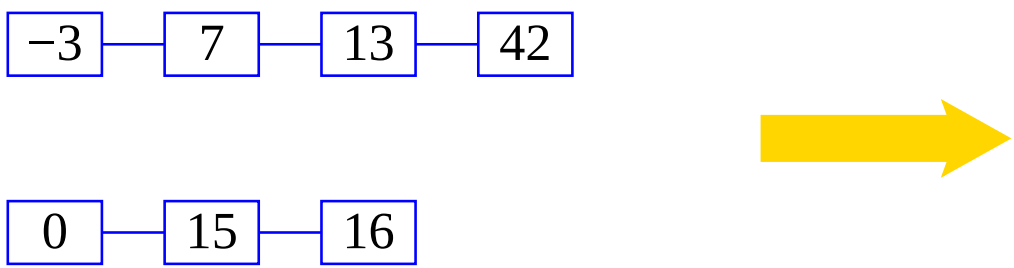

−3
7
13
42
0
15
16

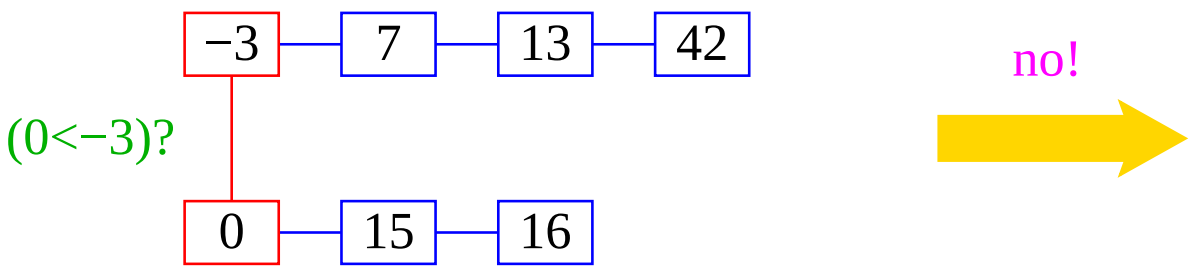

−3
7
13
42
0
15
16

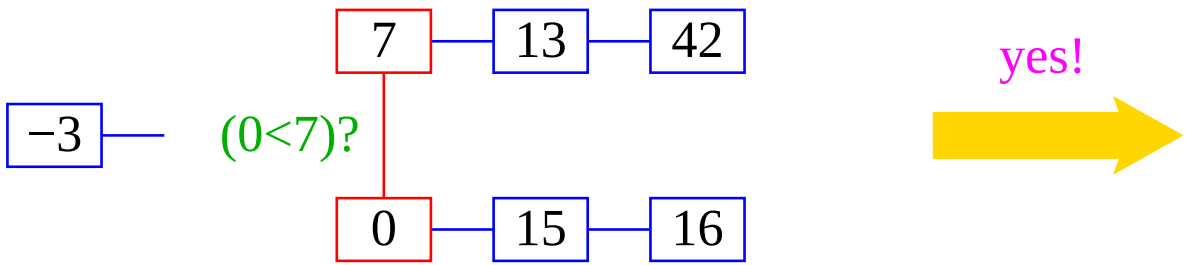

-3
0
7
13
15
16
42

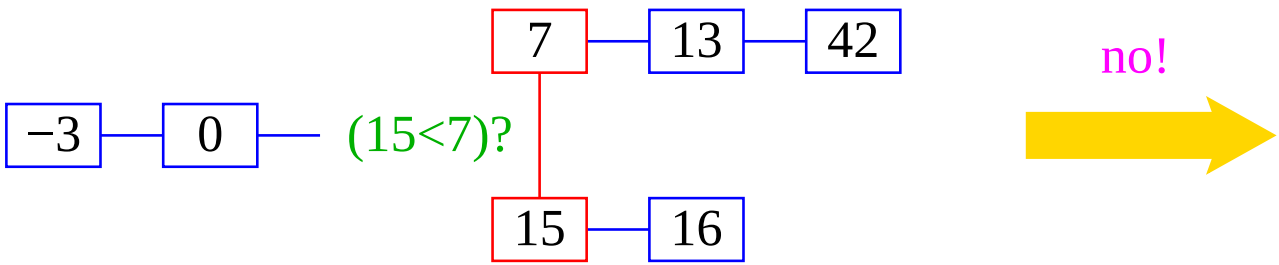
Idee:

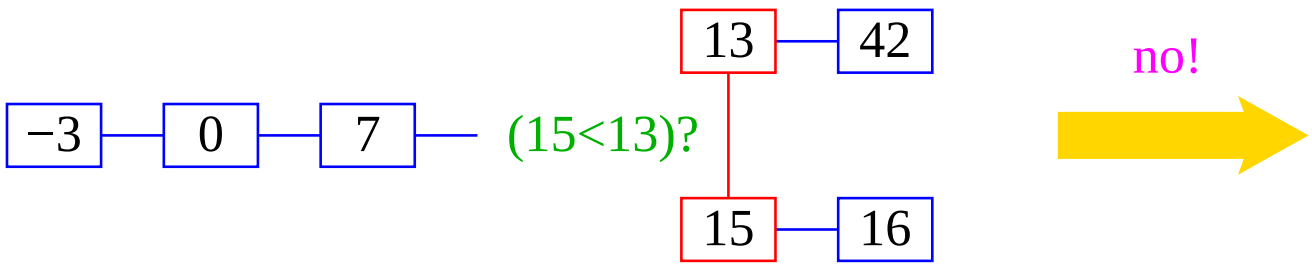
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Input-Listen.
- Falls die n die Länge der längeren Liste ist, sind offenbar maximal nur $n - 1$ Vergleiche nötig :-)

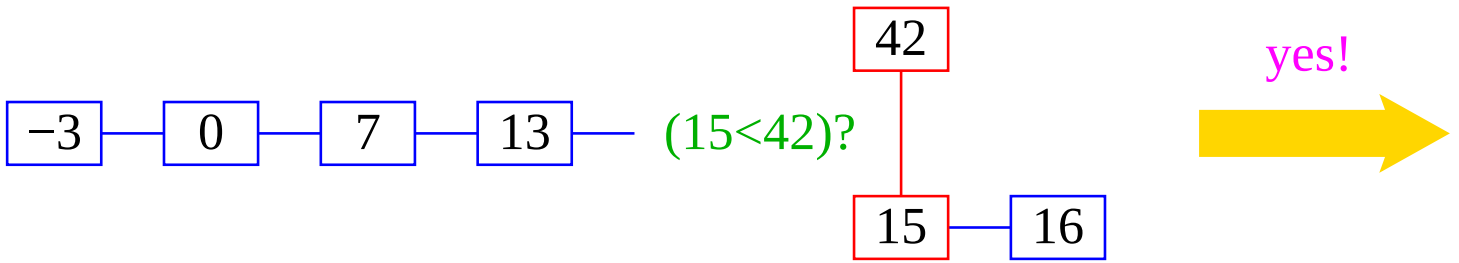

−3
7
13
42
0
15
16

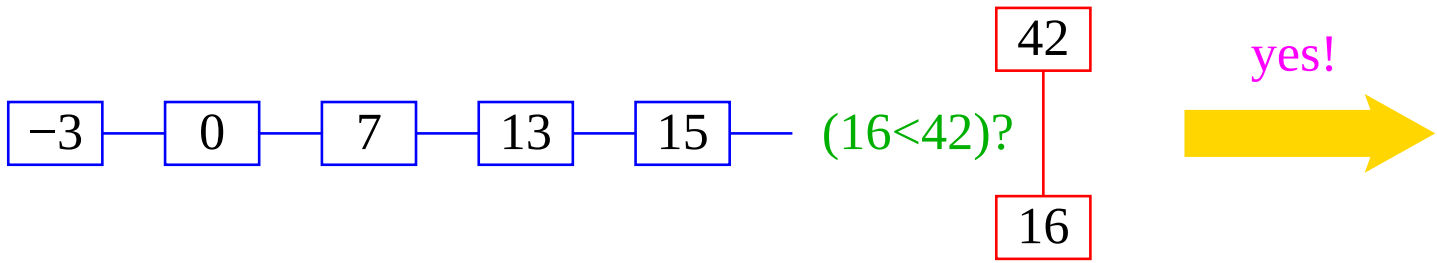

−3
7
13
42
(0<−3)?
0
15
16
no!

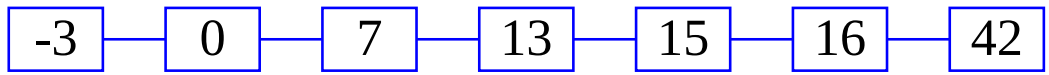

7
13
42
−3
(0<7)?
yes!
0
15
16


7
13
42
−3
0
(15<7)?
15
16
no!


13
42
no!
−3
0
7
(15<13)?
15
16


42
yes!
−3
0
7
13
(15<42)?
15
16

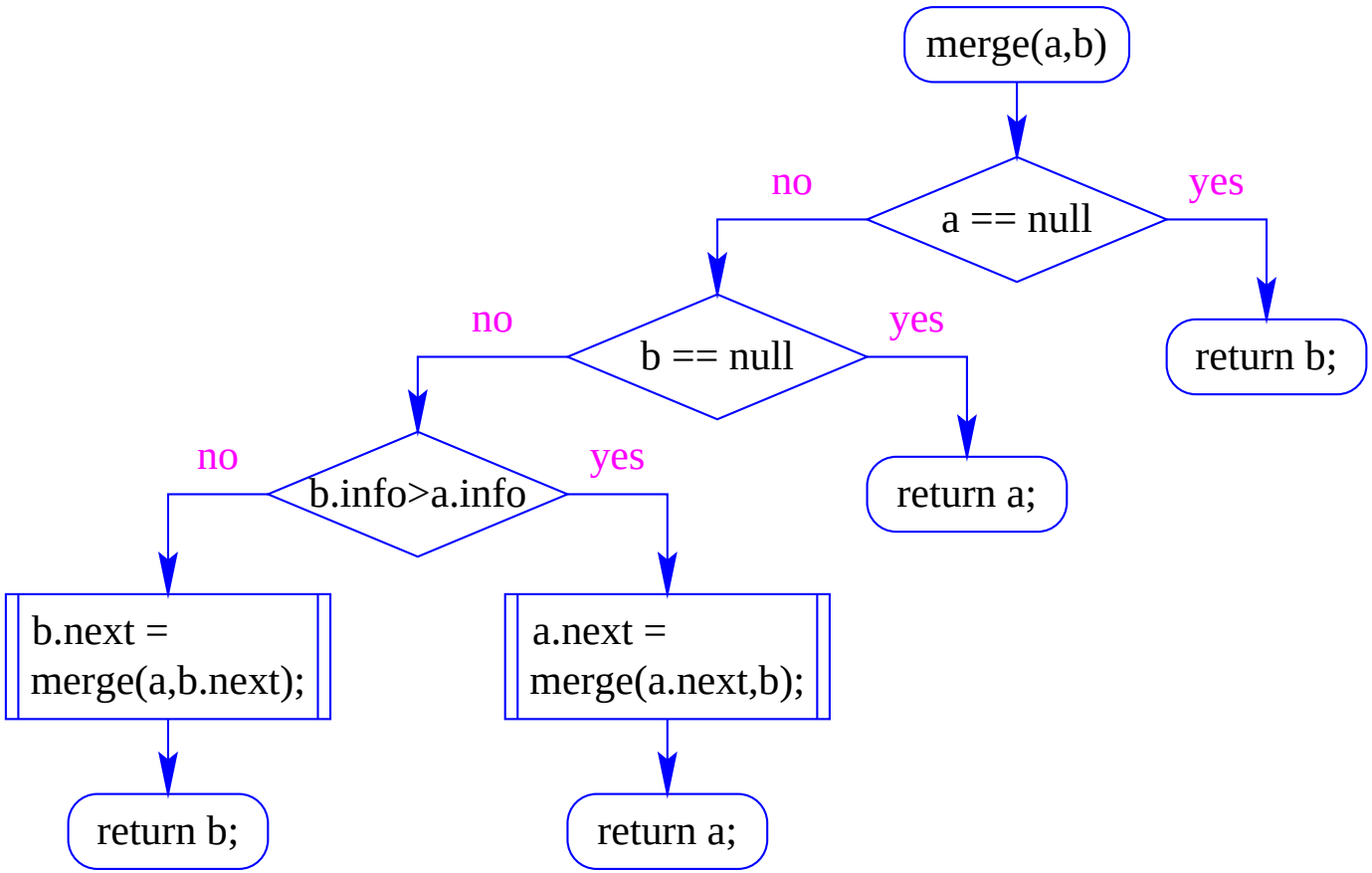

−3
0
7
13
15
(16<42)?
42
16
yes!


-3
0
7
13
15
16
42

Rekursive Implementierung:

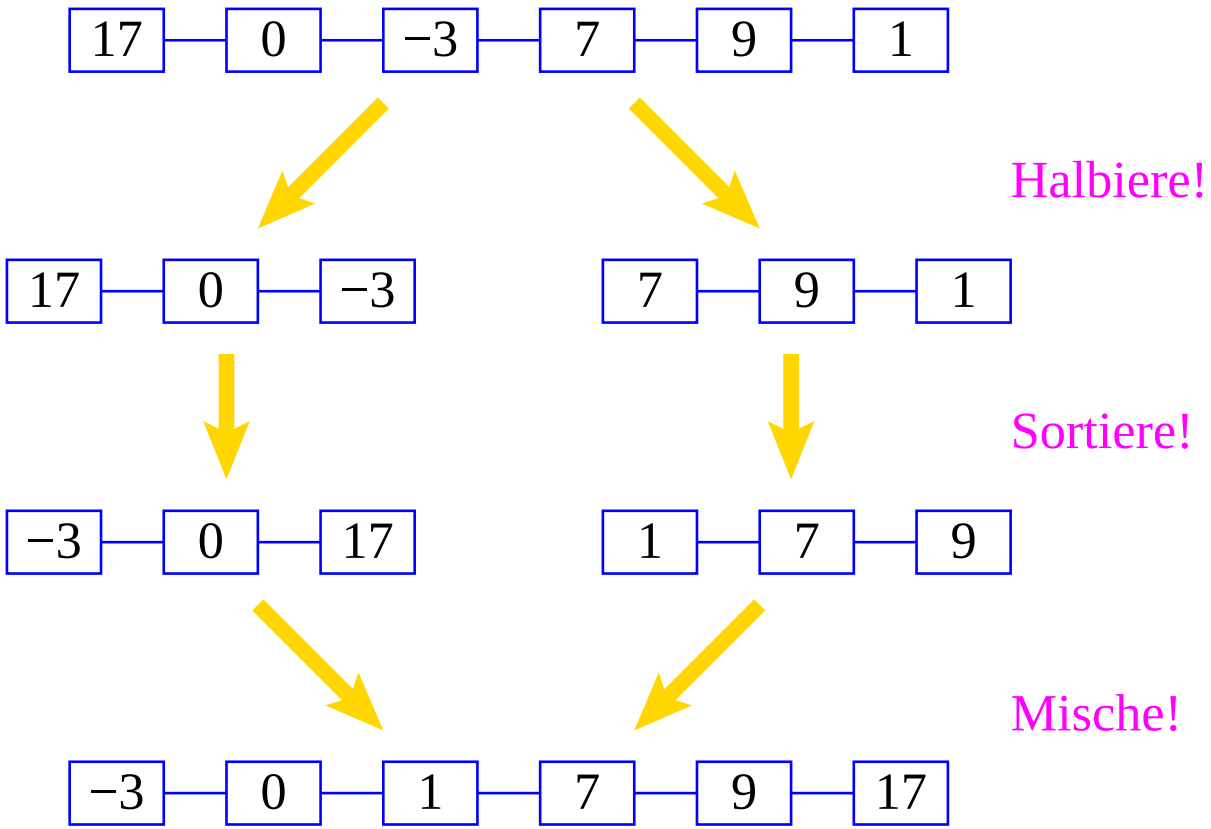
- Falls eine der beiden Listen `a` und `b` leer ist, geben wir die andere aus :-)
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten ...

```
public static List merge(List a, List b) {
   if (b == null)
      return a;
   if (a == null)
      return b;
   if (b.info > a.info) {
      a.next = merge(a.next, b);
      return a;
   } else {
      b.next = merge(a, b.next);
      return b;
      }
   }
```


merge(a,b)
a == null
no
yes
return b;
b == null
no
yes
return a;
b.info>a.info
no
yes
b.next =
merge(a,b.next);
a.next =
merge(a.next,b);
return b;
return a;

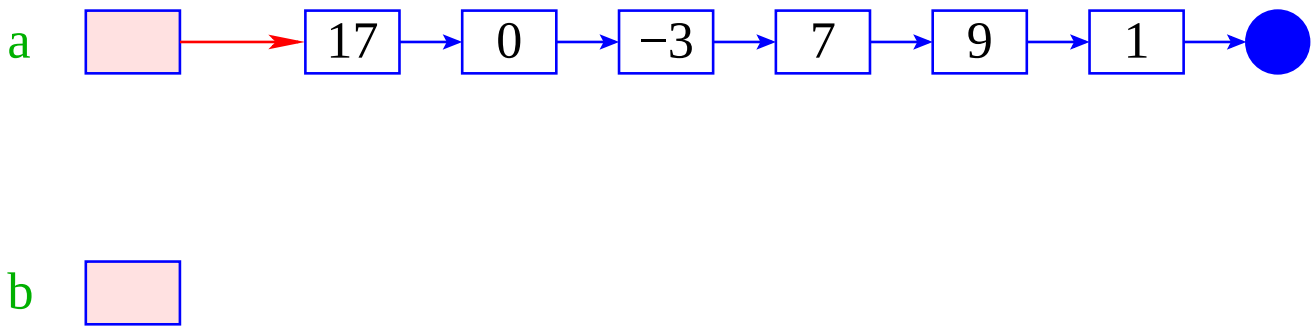
Sortieren durch Mischen:

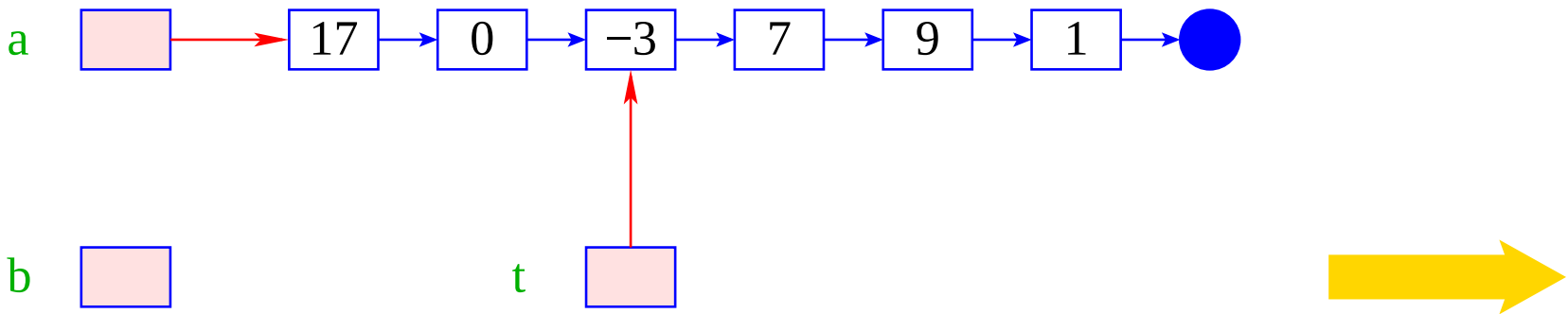
- Teile zu sortierende Liste in zwei Teil-Listen;
- sortiere jede Hälfte für sich;
- mische die Ergebnisse!

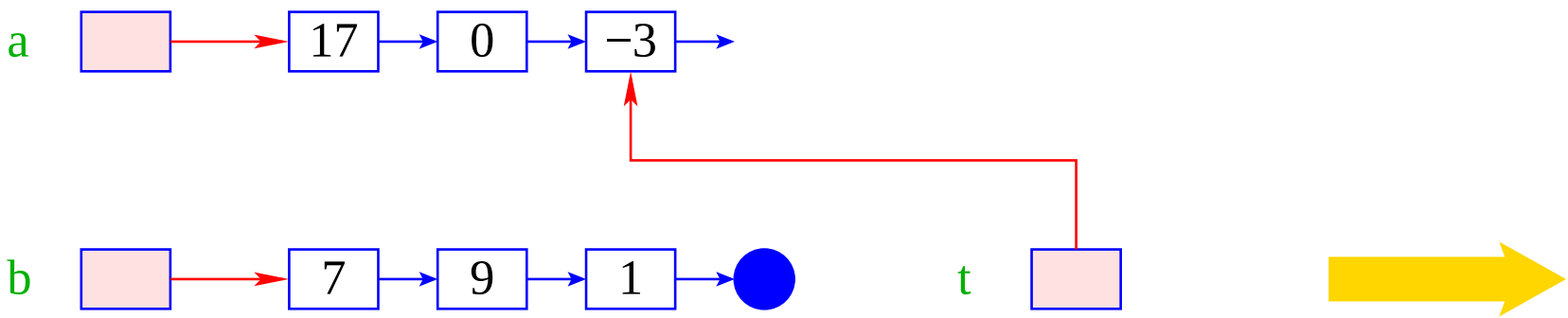

17 0 −3 7 9 1
Halbiere!
17 0 −3
7 9 1
Sortiere!
−3 0 17
1 7 9
Mische!
−3 0 1 7 9 17

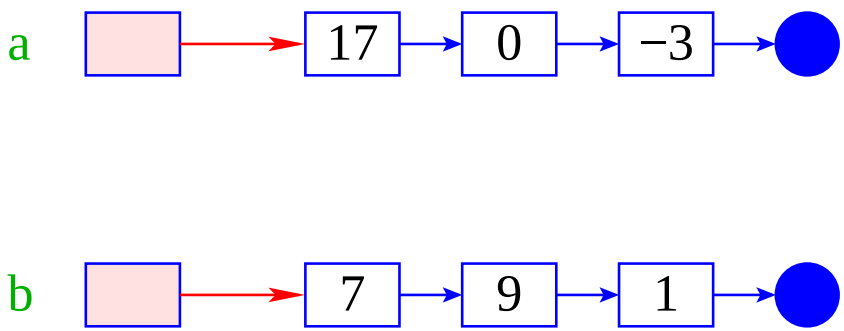
```
public static List sort(List a) {
   if (a == null || a.next == null)
      return a;
   List b = a.half(); // Halbiere!
   a = sort(a);
   b = sort(b);
   return merge(a,b);
   }
```

```
public List half() {
   int n = length();
   List t = this;
   for(int i=0; i<n/2-1; i++)
      t = t.next;
   List result = t.next;
   t.next = null;
   return result;
   }
```


a
17
0
−3
7
9
1
b
b = a.half();


a
17
0
−3
7
9
1
b
t


a
17
0
−3
b
7
9
1
t


a
17
0
−3
b
7
9
1

Diskussion:

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

  Dann gilt:

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

- 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 ?-))

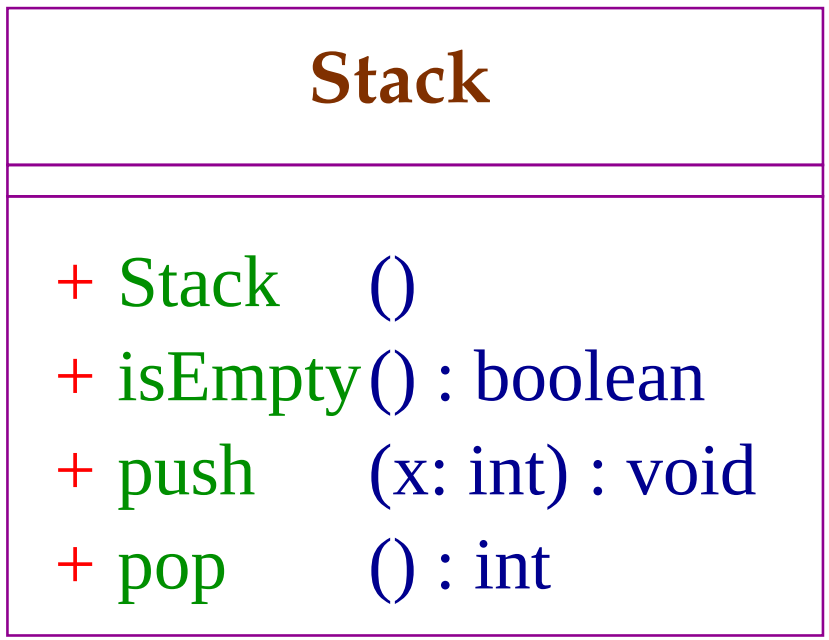
11.2 Keller (Stacks)

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.

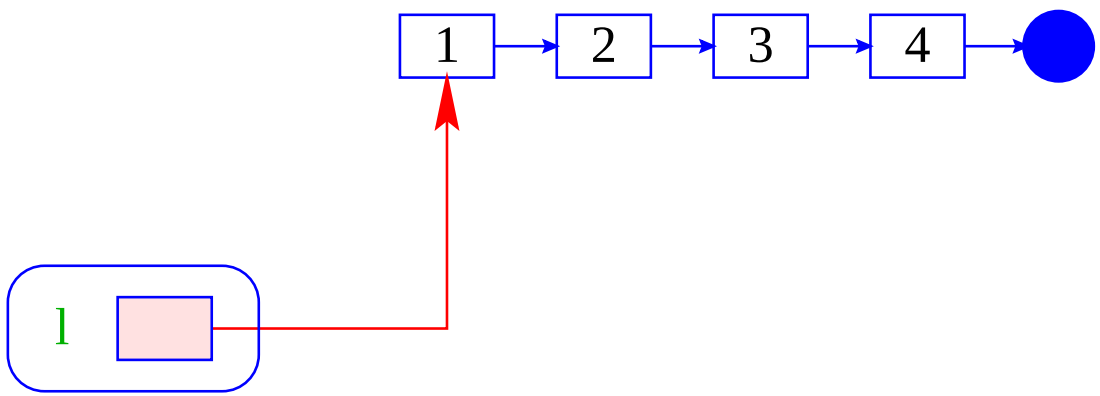
Weiterhin müssen wir einen leeren Keller anlegen können.

Modellierung:



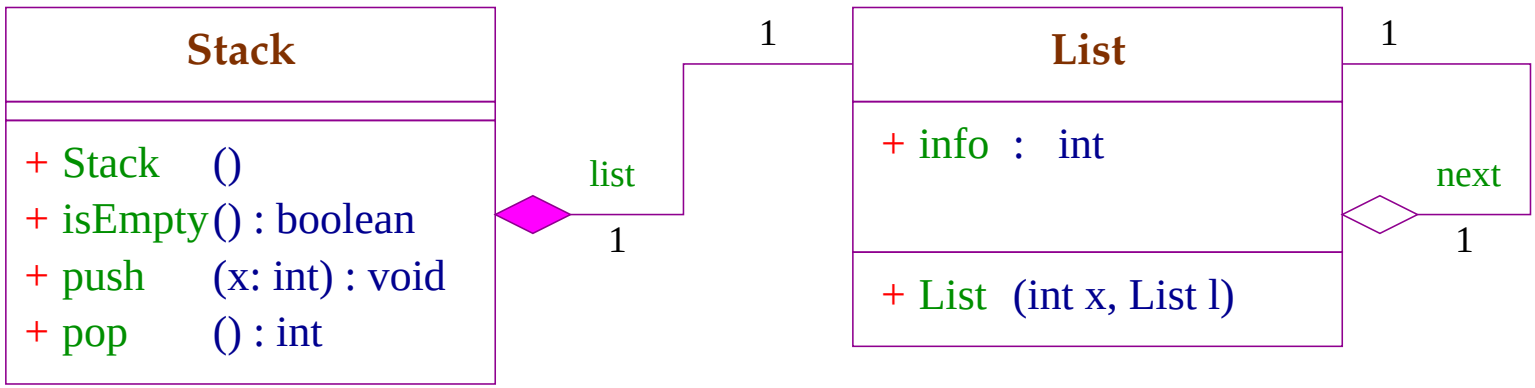
Erste Idee:

- 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;

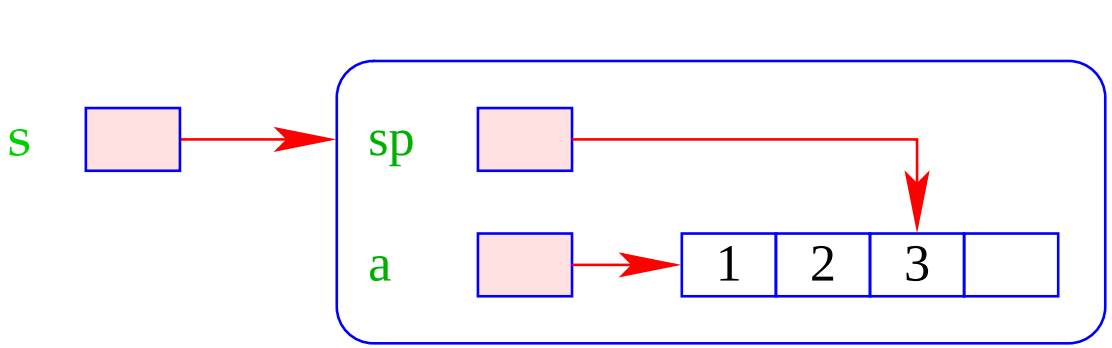
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

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

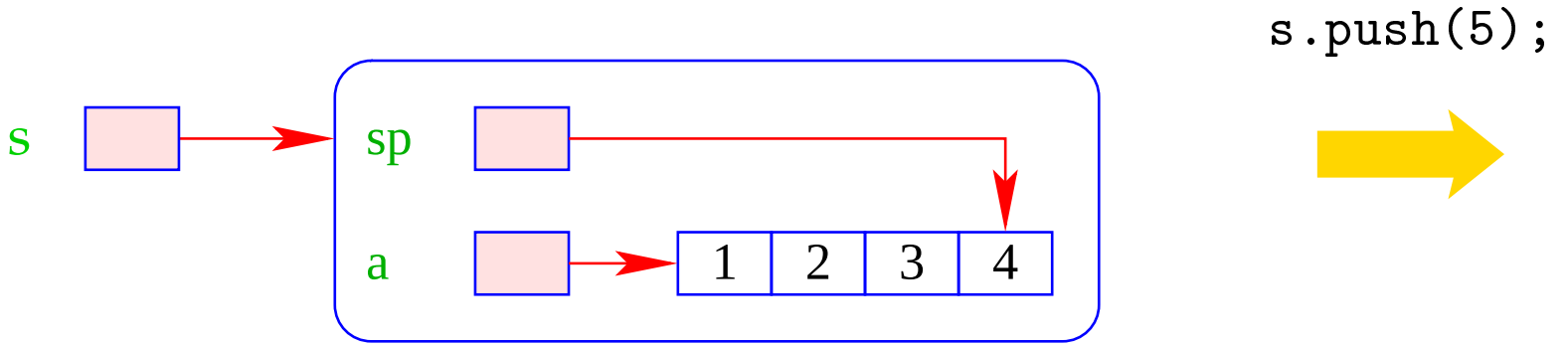
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

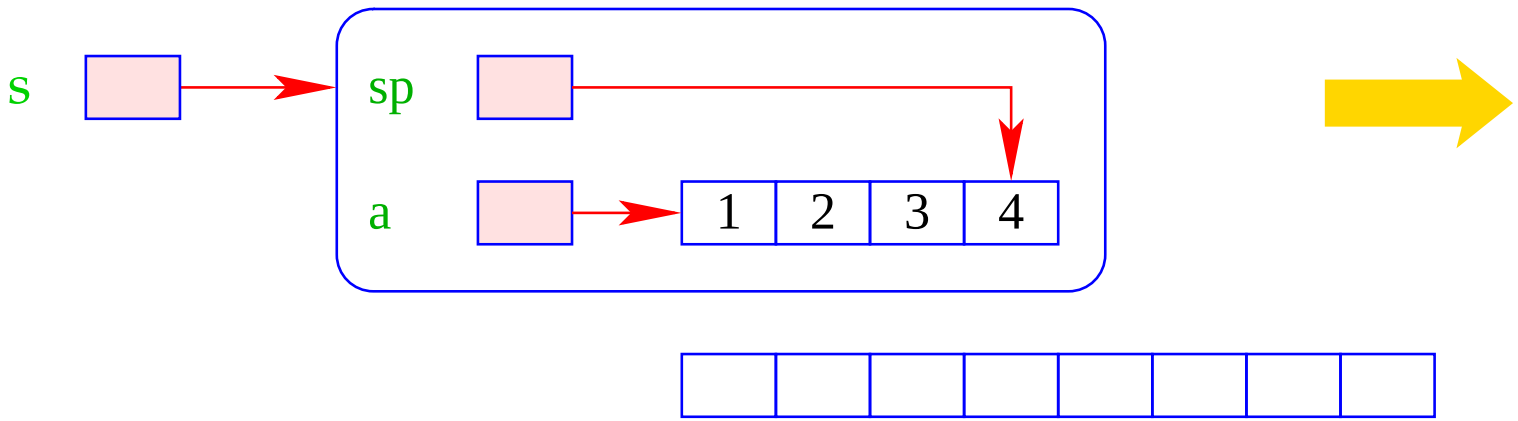
Zweite Idee:

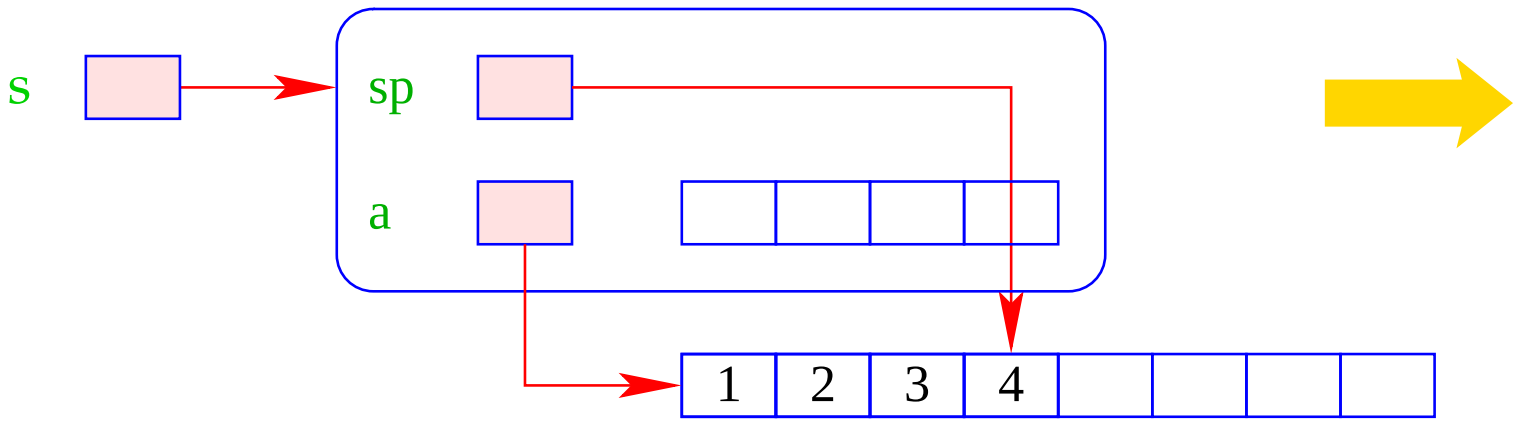
- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres :-)

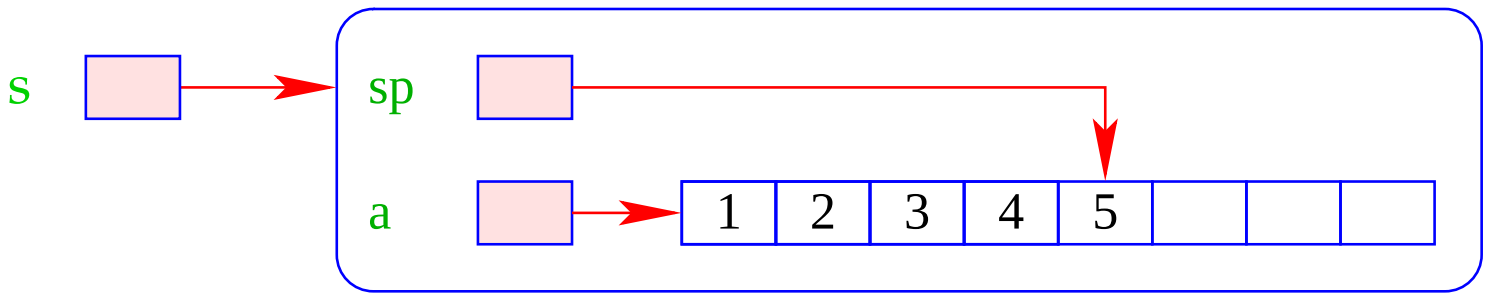



```
s.push(4);
```

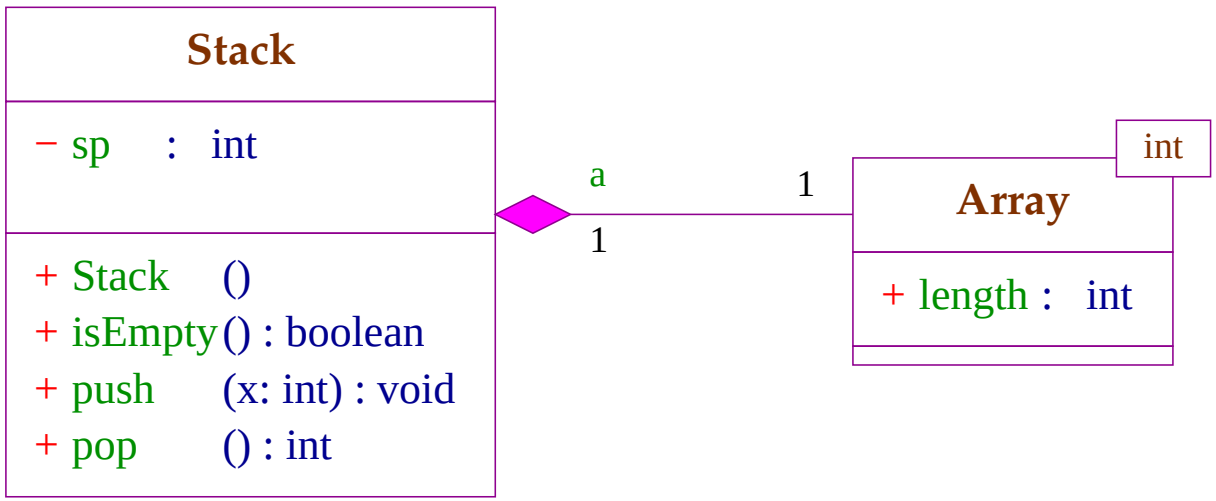

s
sp
a
1
2
3
4
s.push(5);


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);
      }
   ...
```

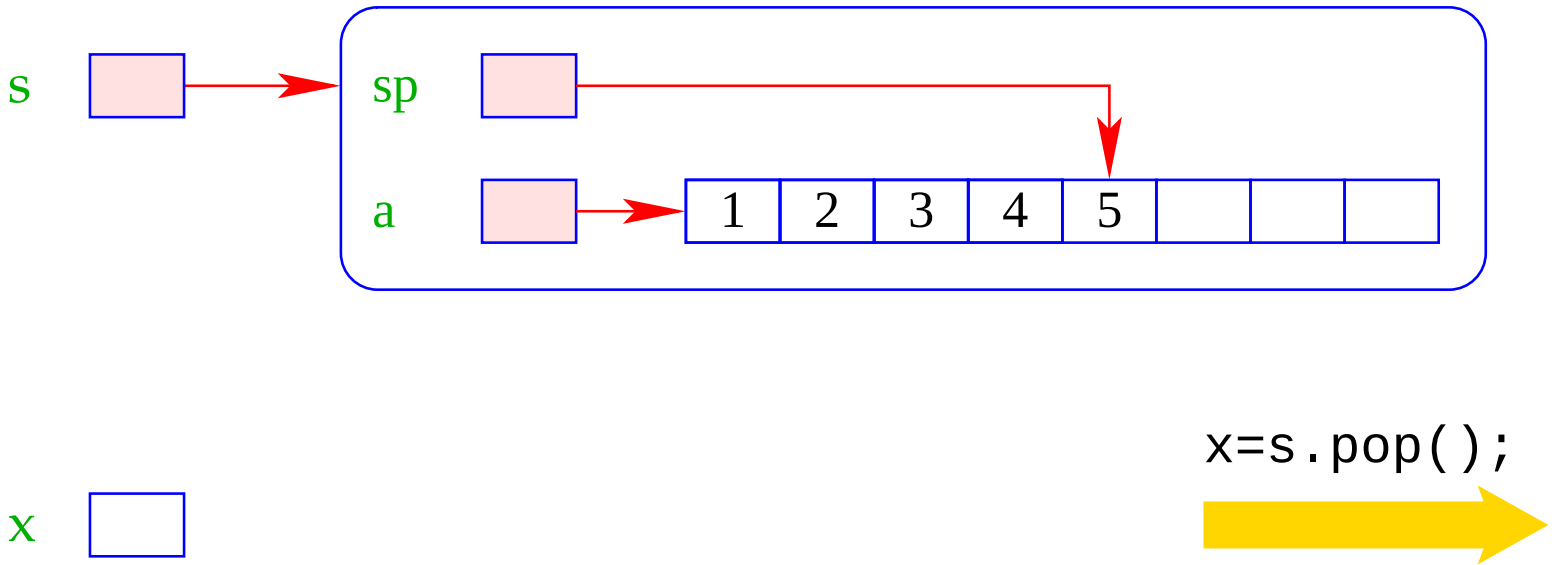
```
   public int pop() {
      return a[sp--];
      }
   public void push(int x) {
      ++sp;
      if (sp == a.length) {
         int[] b = new int[2*sp];
         for(int i=0; i<sp; ++i) b[i] = a[i];
         a = b;
         }
      a[sp] = x;
      }
   public toString() {...}
} // end of class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

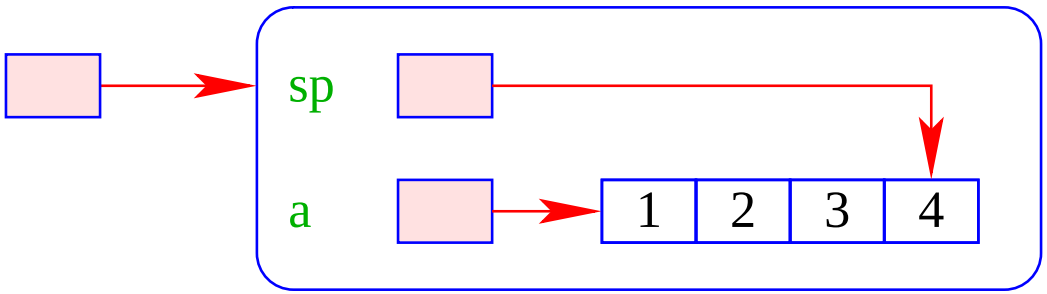
Nachteil:

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

Erste Idee:

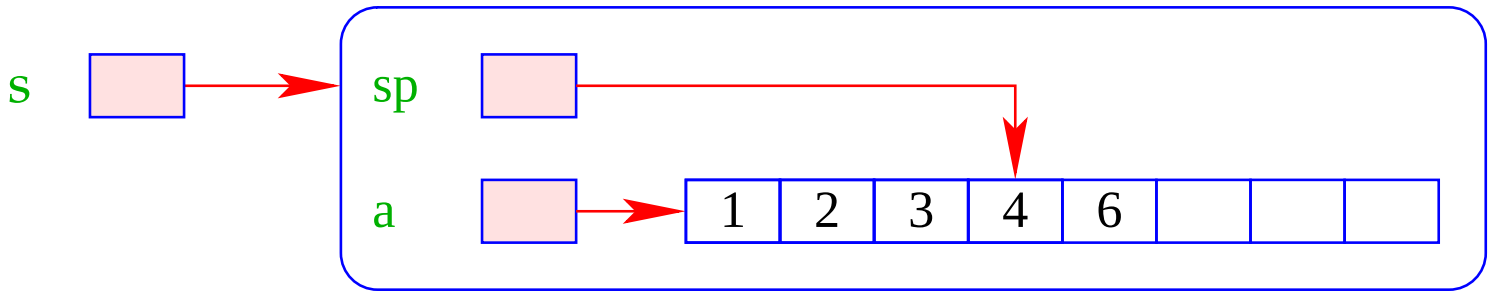
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei ...


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



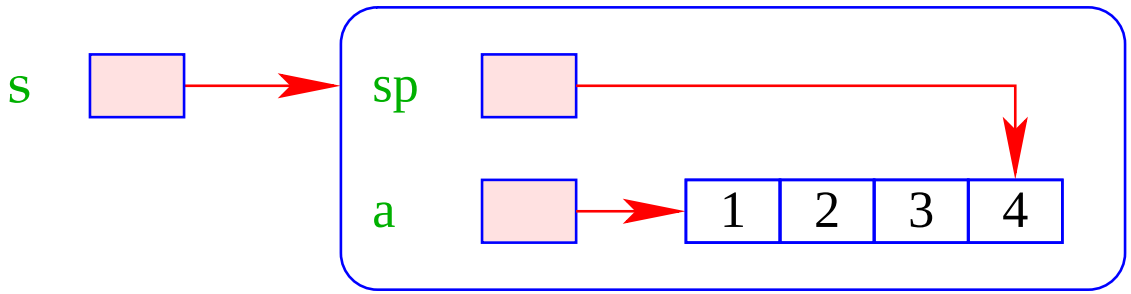
x 5

```
s.push(6);
```



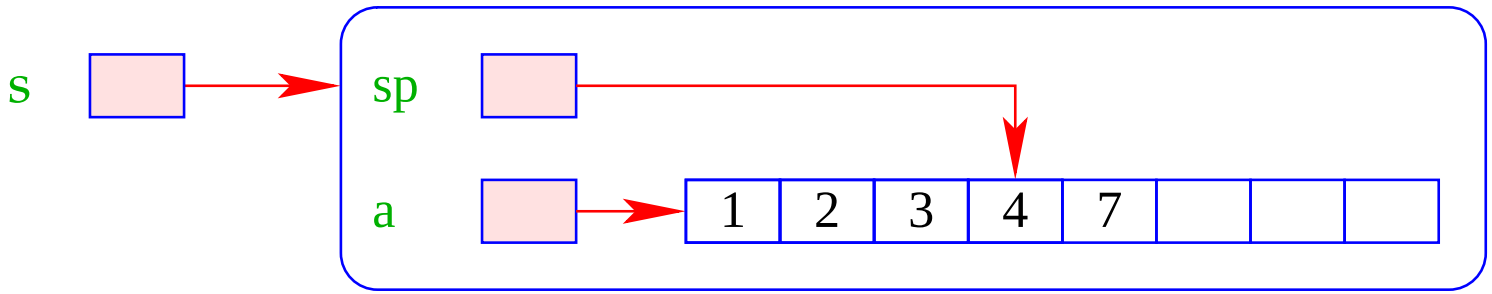

```
x = s.pop();
```

x 5



x 6

```
s.push(7);
```



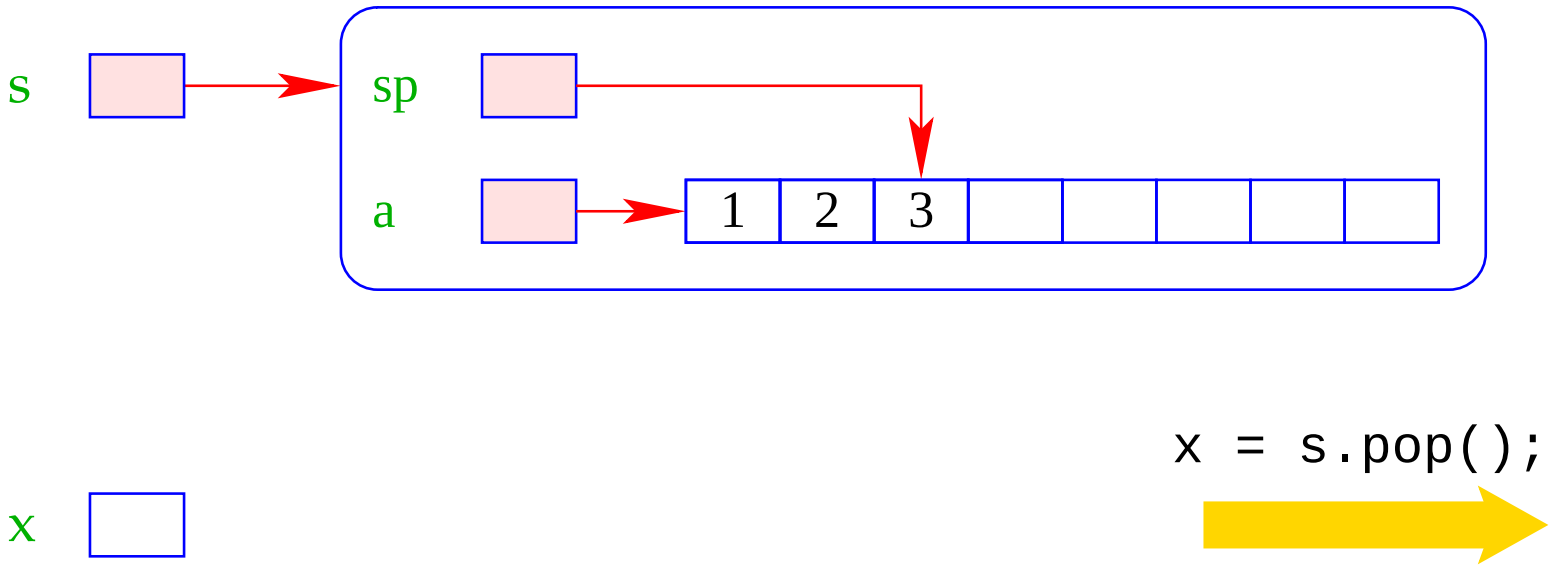

```
x = s.pop();
```

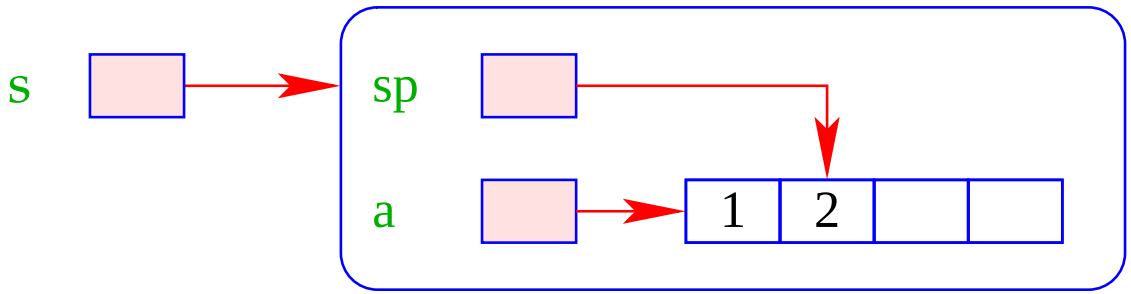

x 6

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

Zweite Idee:

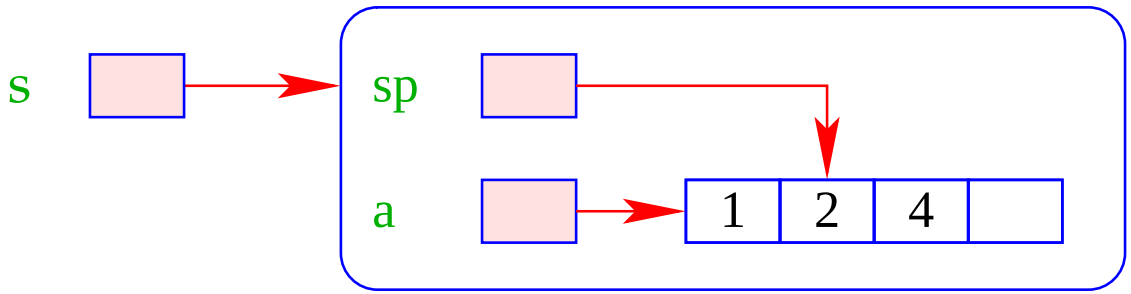
- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte !


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



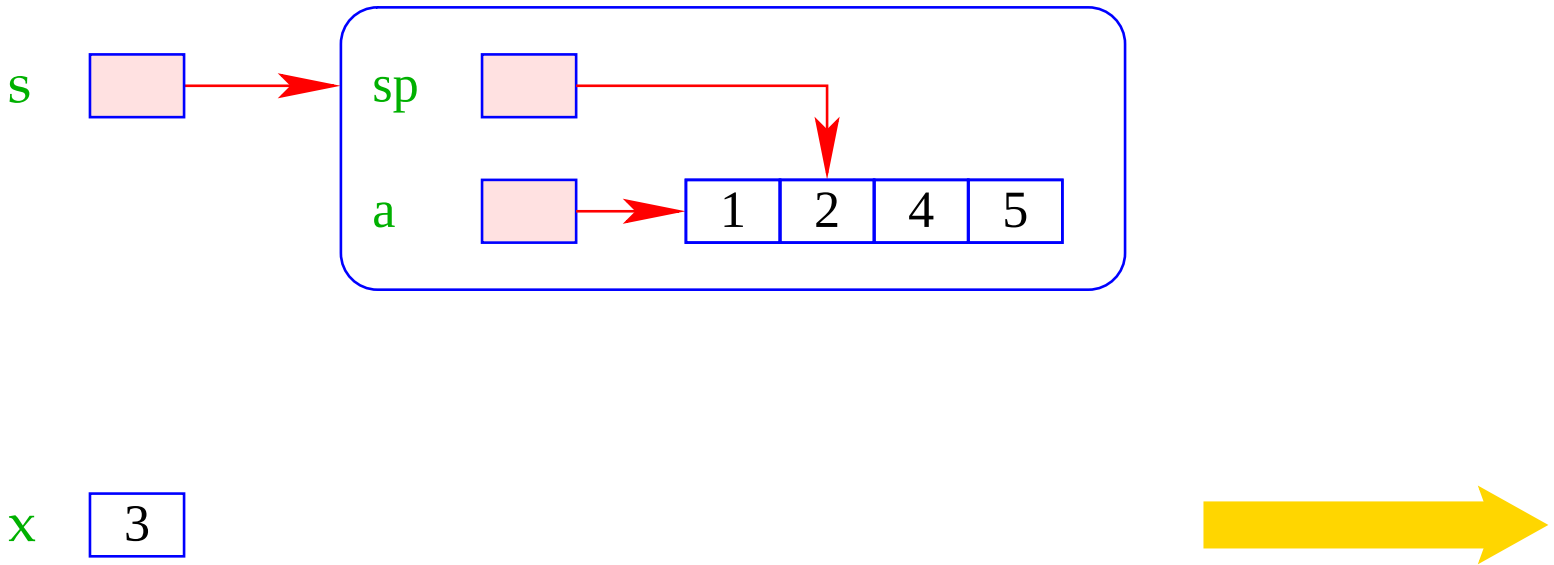
x 3

```
s.push(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;
   }
```