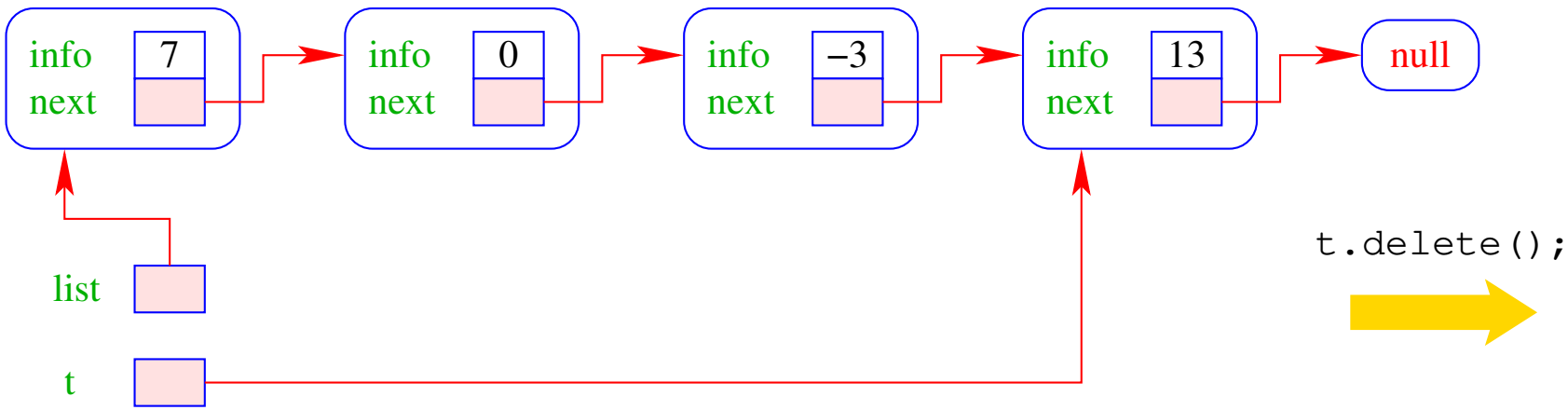
```
// Objekt-Methoden:
   public void insert(int x) {
      next = new List(x,next);
      }
   public void delete() {
      if (next != null)
         next = next.next;
      }
   public String toString() {
      String result = "["+info;
      for(List t=next; t!=null; t=t.next)
         result = result+", "+t.info;
      return result+"]";
      }
   ...
```

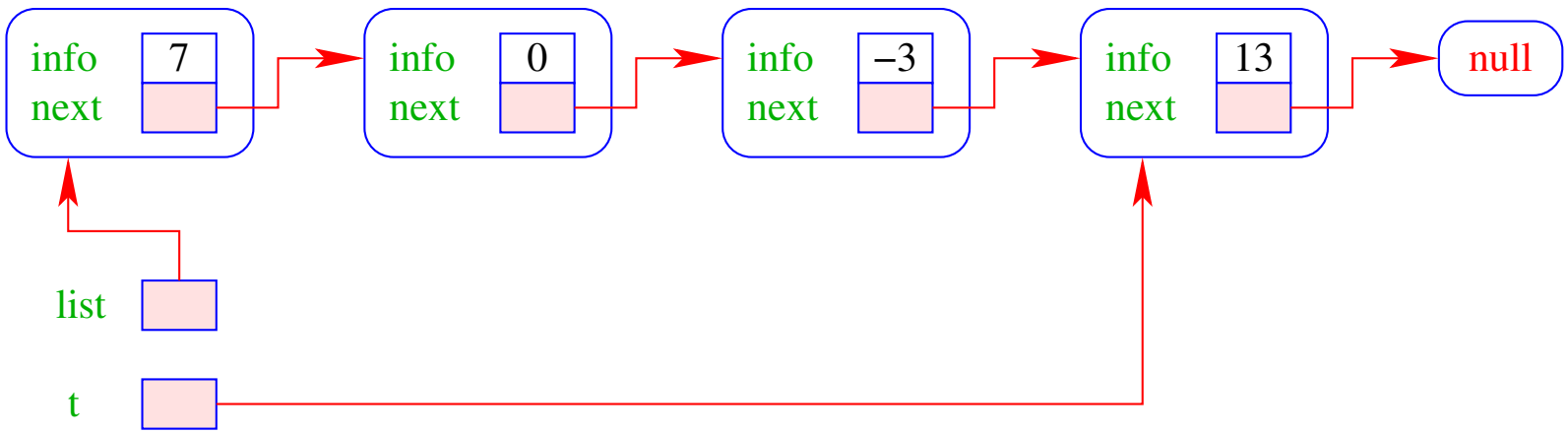
- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar $\Longrightarrow$ sehr flexibel, sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

Achtung:

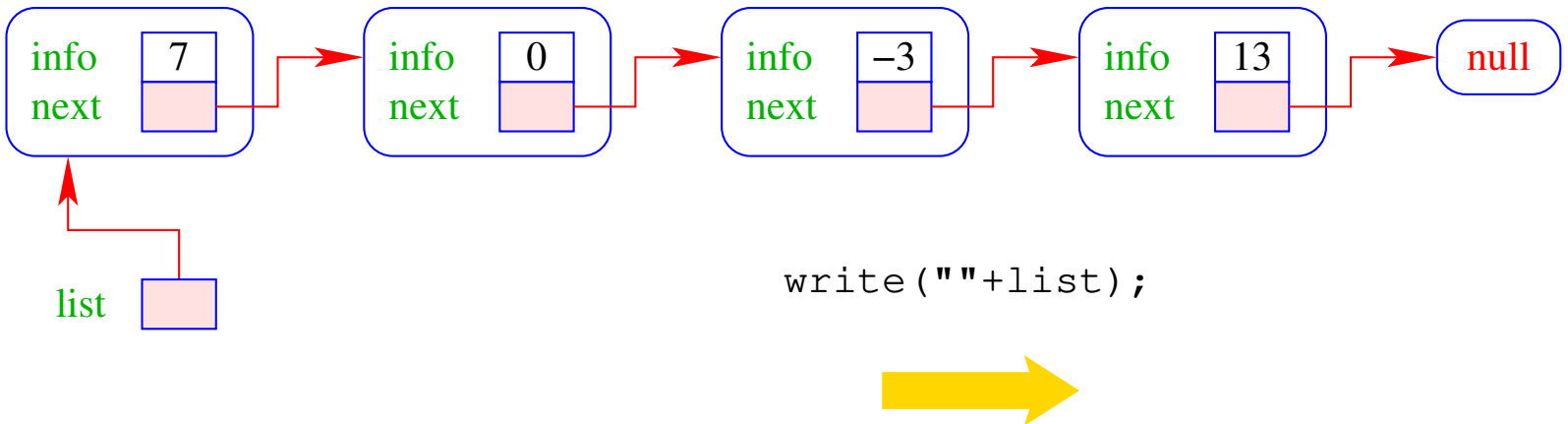
Wenn `delete()` mit dem letzten Element der Liste aufgerufen wird, zeigt `next` auf `null`.

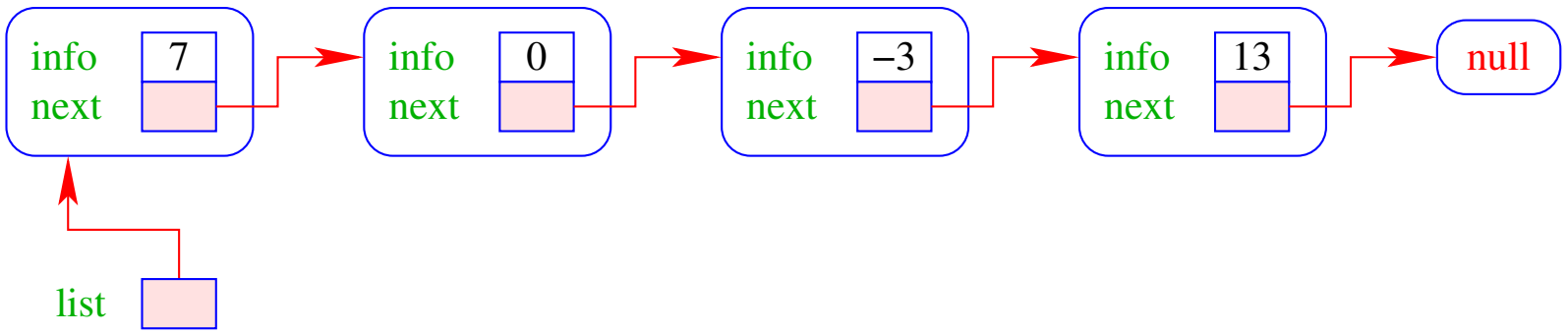
$\Longrightarrow$ Wir tun dann nix.


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t
t.delete();

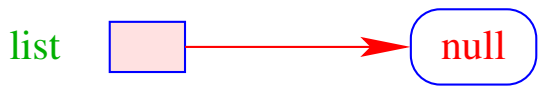

info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t

- Weil Objekt-Methoden nur für von `null` verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels `toString()` als `String` dargestellt werden.
- Der Konkatenations-Operator "+" ist so schlau, vor Aufruf von `toString()` zu überprüfen, ob ein `null`-Objekt vorliegt. Ist das der Fall, wird "null" ausgegeben.
- Wollen wir eine andere Darstellung, benötigen wir eine Klassen-Methode `String toString(List l)`.

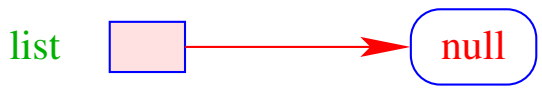

info
next
7
info
next
0
info
next
−3
info
next
13
null
list
write(""+list);




```
"[7, 0, -3, 13]"
```




```
write(""+list);
```




```
"null"
```

```
// Klassen-Methoden:
   public static boolean isEmpty(List l) {
      if (l == null)
         return true;
      else
         return false;
      }
   public static String toString(List l) {
      if (l == null)
         return "[]";
      else
         return l.toString();
      }
   ...
```

```
public static List arrayToList(int[] a) {
   List result = null;
   for(int i = a.length-1; i>=0; --i)
      result = new List(a[i],result);
   return result;
   }
public int[] listToArray() {
   List t = this;
   int n = length();
   int[] a = new int[n];
   for(int i = 0; i < n; ++i) {
      a[i] = t.info;
      t = t.next;
      }
   return a;
   }
...
```

- Damit das erste Element der Ergebnis-Liste `a[0]` enthält, beginnt die Iteration in `arrayToList()` beim größten Element.
- `listToArray()` ist als Objekt-Methode realisiert und funktioniert darum nur für nicht-leere Listen :-)
- Um eine Liste in ein Feld umzuwandeln, benötigen wir seine Länge.

```
    private int length() {
       int result = 1;
       for(List t = next; t!=null; t=t.next)
          result++;
       return result;
       }
} // end of class List
```

- Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- Damit `length()` auch für `null` funktioniert, hätten wir analog zu toString() auch noch eine Klassen-Methode `int length(List l)` definieren können.
- Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode `static int [] listToArray (List l)` zu definieren, die auch für leere Listen definiert ist.

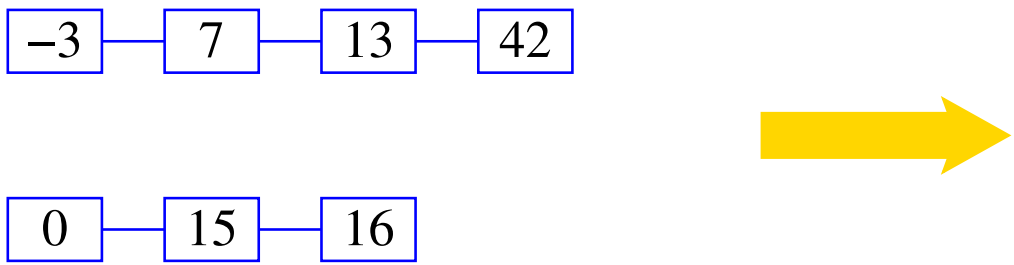
Anwendung: Mergesort – Sortieren durch Mischen

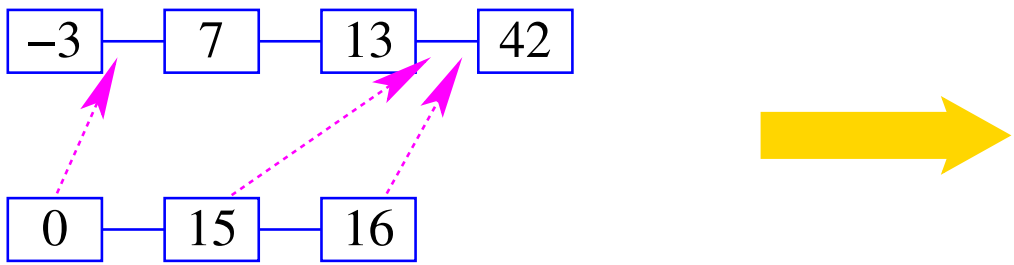
John von Neumann (1945)

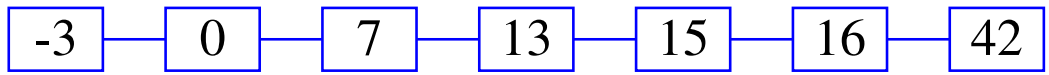
Mischen:

Eingabe: zwei sortierte Listen;

Ausgabe: eine gemeinsame sortierte Liste.

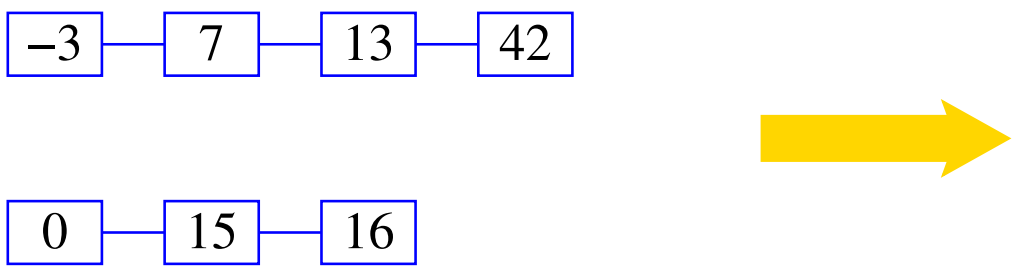

−3
7
13
42
0
15
16

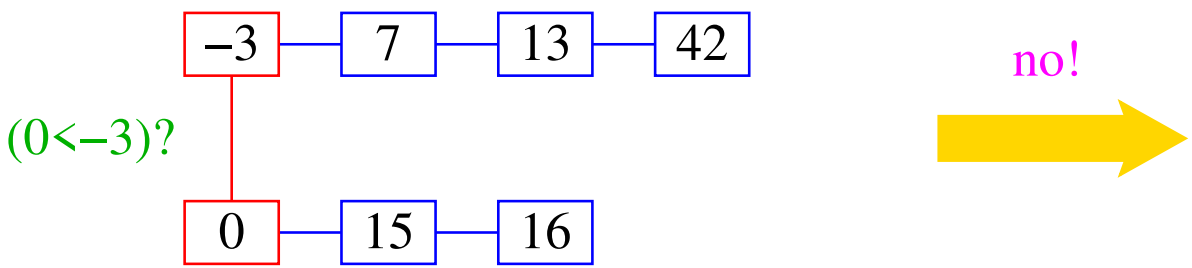

−3
7
13
42
0
15
16

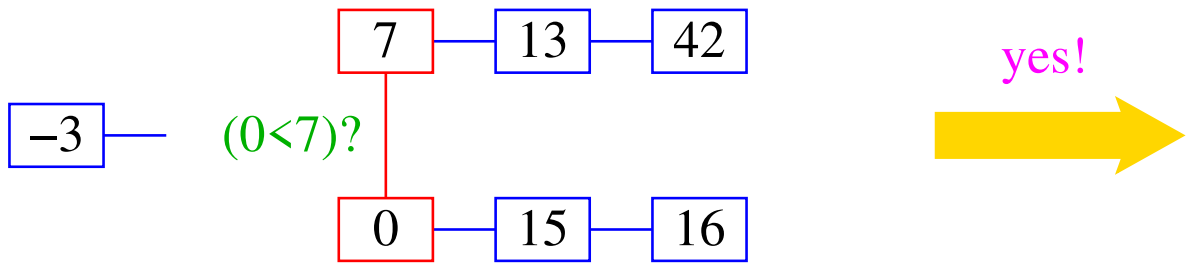

-3
0
7
13
15
16
42

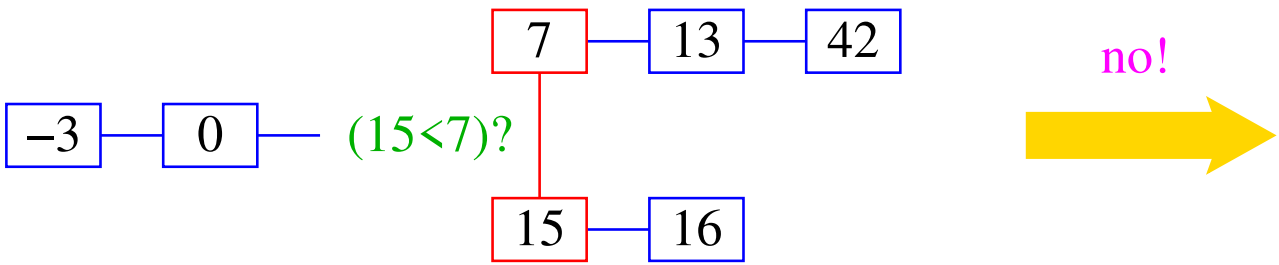
Idee:

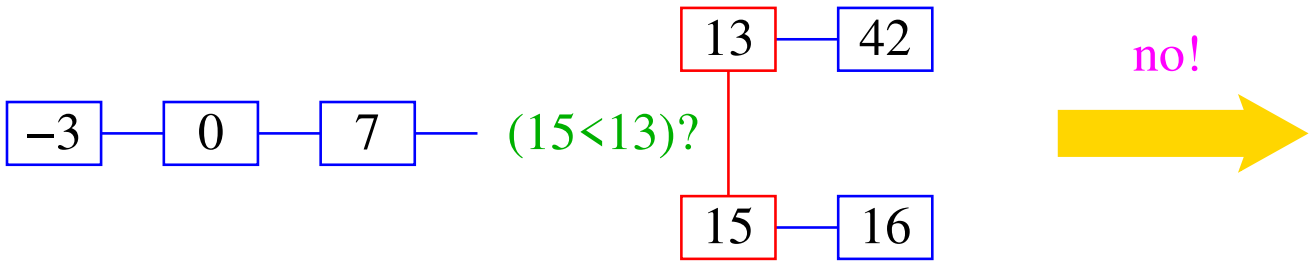
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Input-Listen.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche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

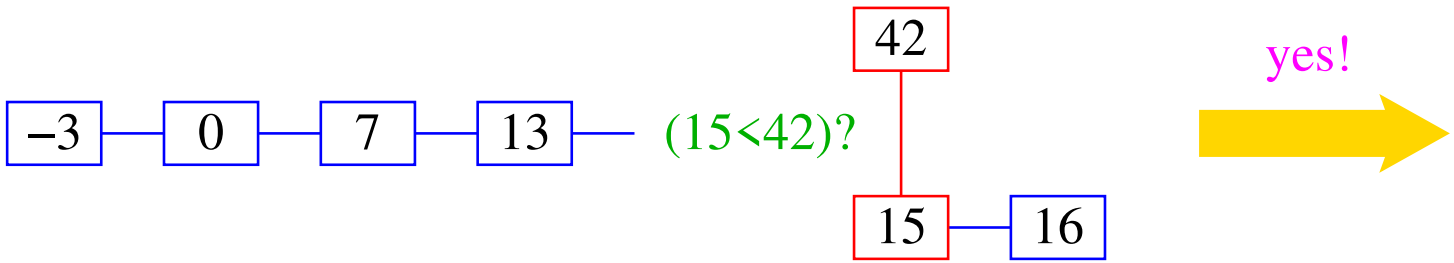

−3
7
13
42
0
15
16

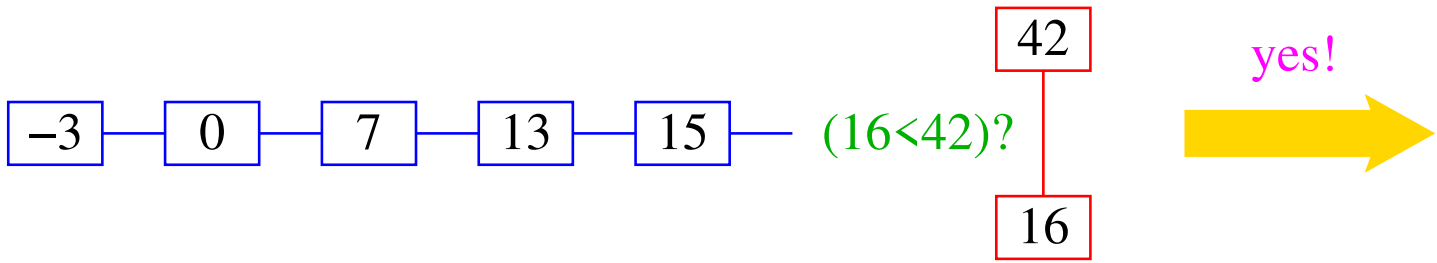

−3
7
13
42
(0<−3)?
0
15
16
no!

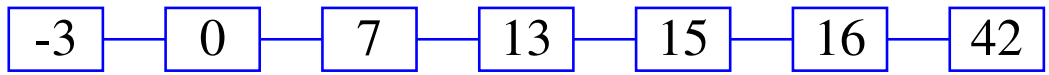

7
13
42
−3
(0<7)?
yes!
0
15
16


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

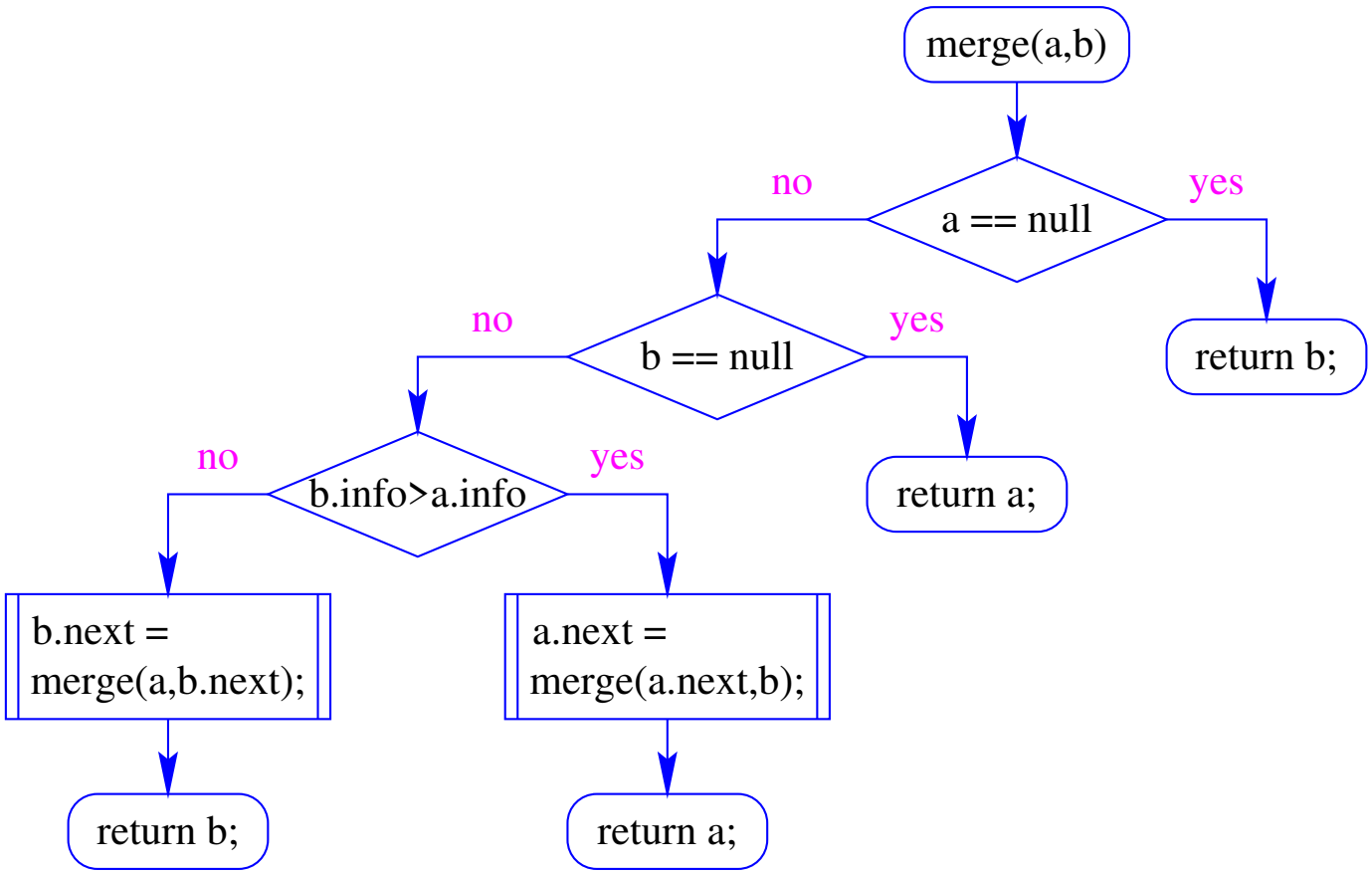

−3
0
7
13
15
(16<42)?
42
16
yes!


-3
0
7
13
15
16
42

Rekursive Implementierung:

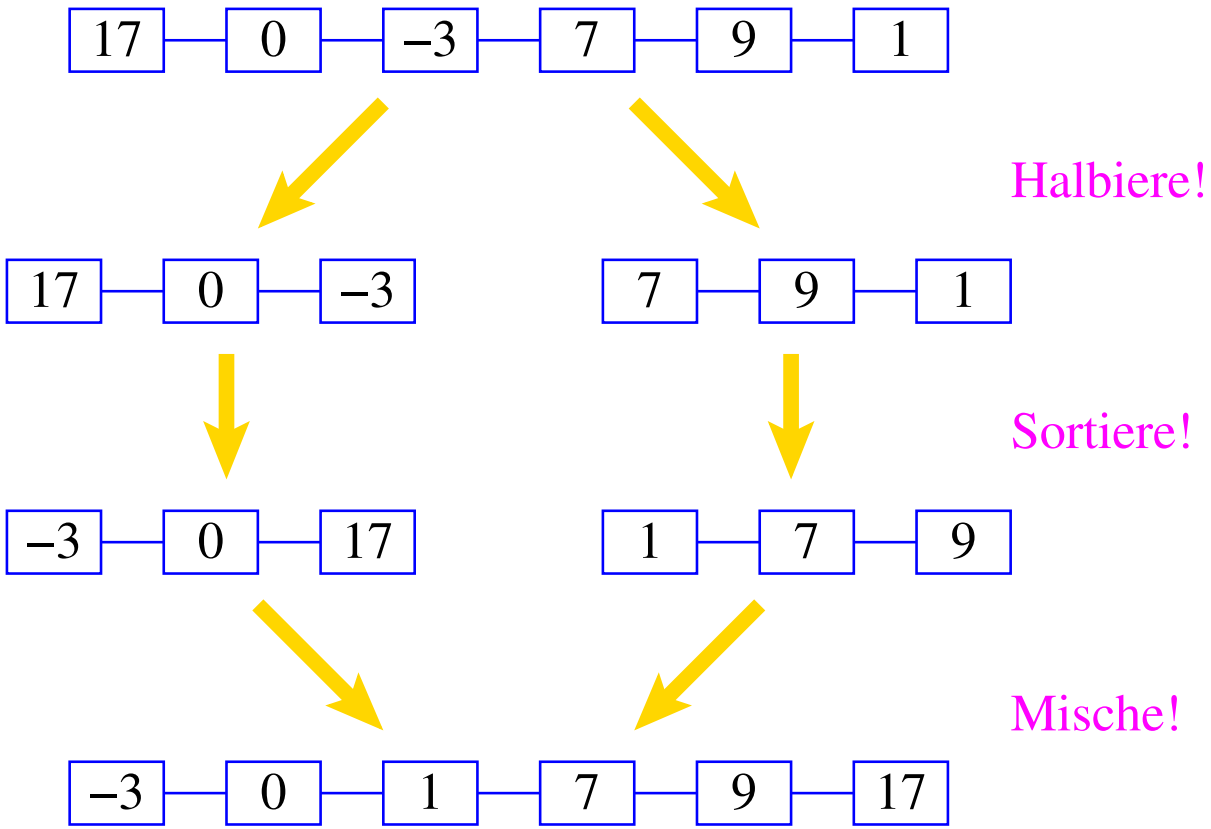
- Falls eine der beiden Listen `a` und `b` leer ist, geben wir die andere aus :-)
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente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

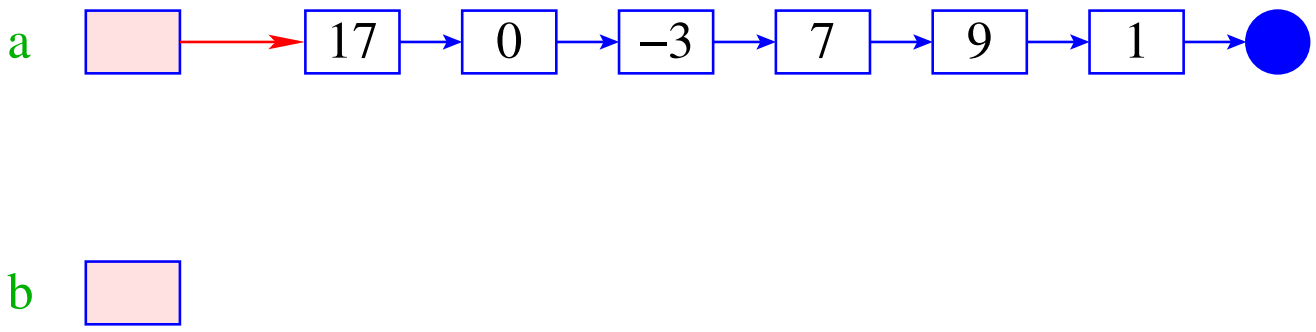
Sortieren durch Mischen:

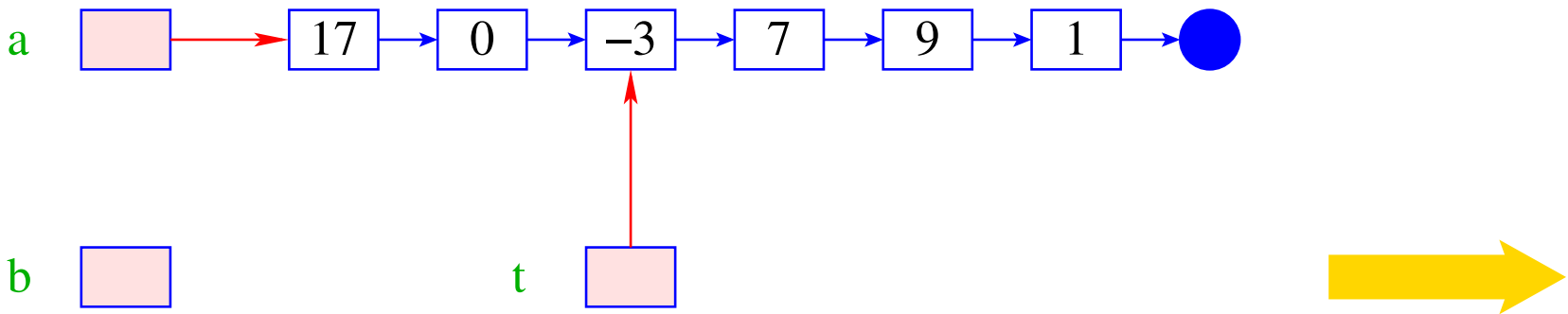
- Teile zu sortierende Liste in zwei Teil-Listen;
- sortiere jede Hälfte für sich;
- mische die Ergebnisse!

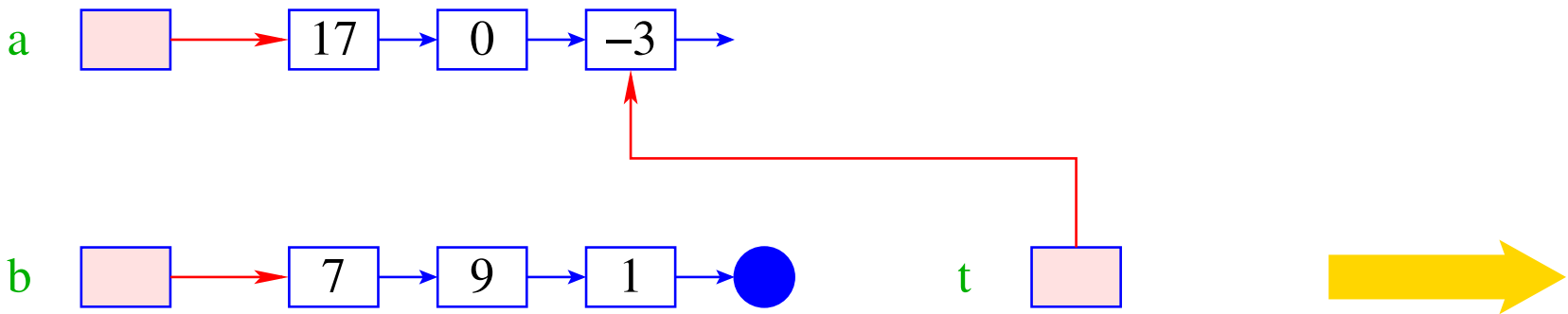

17 0 −3 7 9 1
Halbiere!
17 0 −3
7 9 1
Sortiere!
−3 0 17
1 7 9
Mische!
−3 0 1 7 9 17

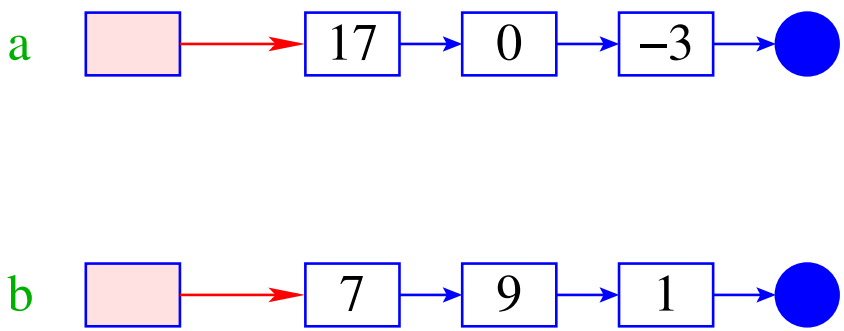
```
public static List sort(List a) {
   if (a == null || a.next == null)
      return a;
   List b = a.half(); // Halbiere!
   a = sort(a);
   b = sort(b);
   return merge(a,b);
   }
```

```
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