

9.1 Verkettete Liste: Konzept

9.2 Iteratoren

9.3 Implementierung in Java

9.4 Menge ohne Sortierung

9.5 Menge mit Sortierung

9.6 Sortieren durch Auswählen



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

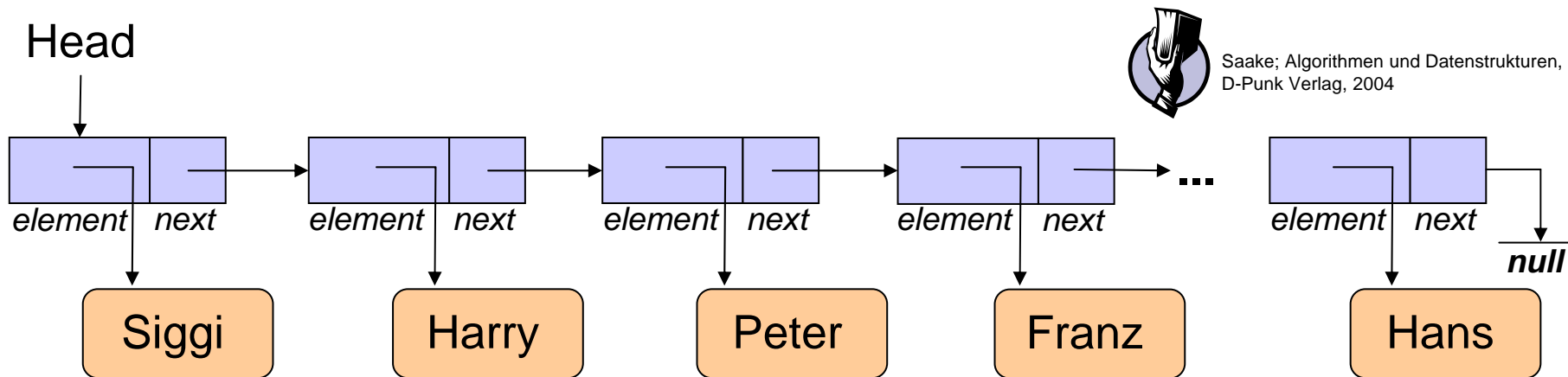
- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



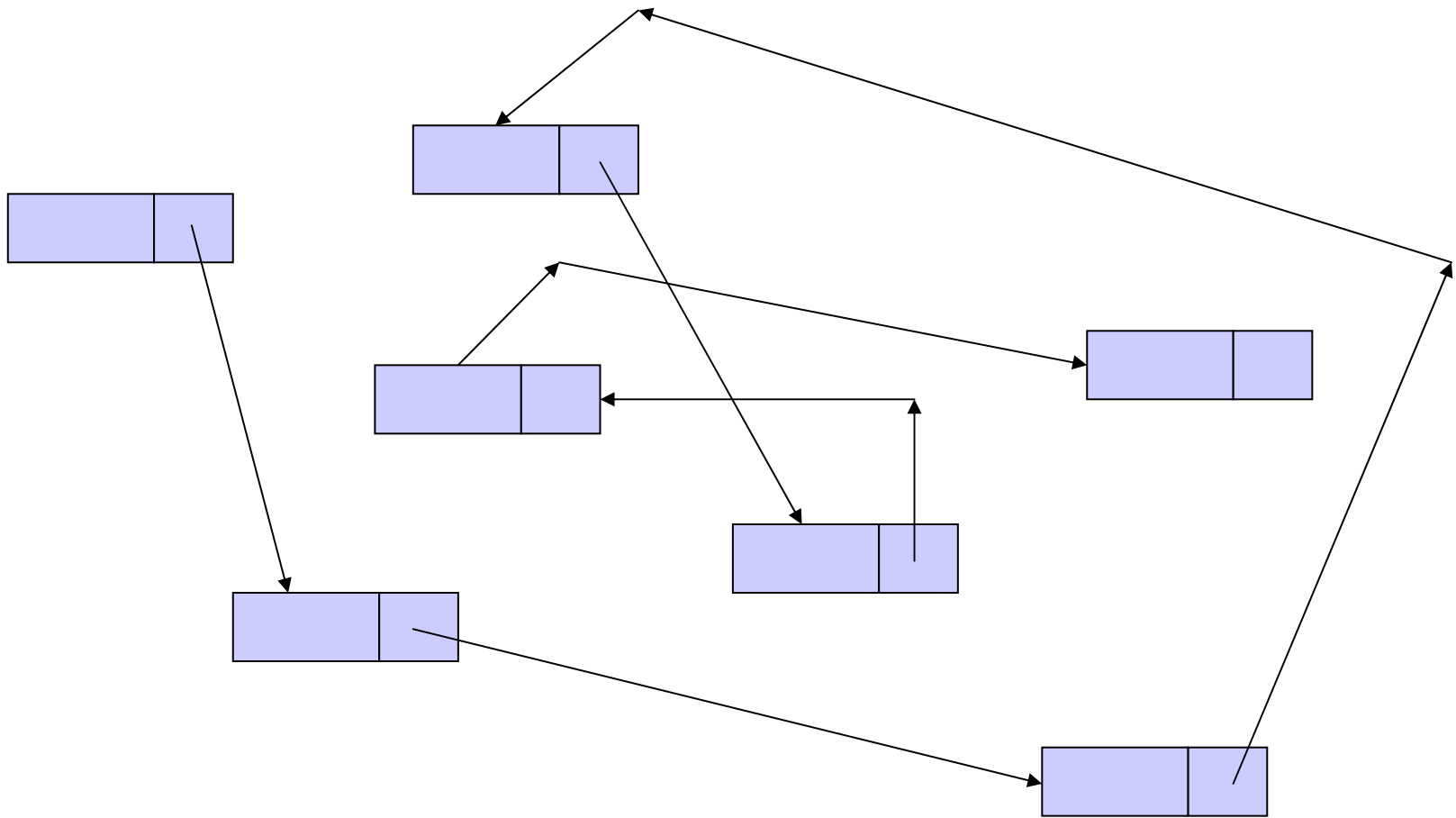
Typischer Vertreter einer dynamischen Datenstruktur

- Menge von Knoten, die untereinander verzeigert (verkettet) sind
- Knoten
 - Verweis auf das eigentlich zu speichernde Element (Objektreferenz in Java)
 - Verweis auf den nachfolgenden Knoten
 - Letzter Knoten verweist auf null
- Erster Knoten: Head
 - Alle anderen Knoten durch Navigation erreichbar

Veranschaulichung



Listengröße ist von vornherein unbekannt, d.h. es kann kein zusammenhängender Speicherplatz reserviert werden.



Einige Grundoperationen sollten in jedem Fall möglich sein.

- Erstellen einer neuen Liste
- Einfügen eines neuen Elements an beliebiger Stelle
- Löschen eines Elements an beliebiger Stelle
- Überprüfen, ob ein Element in einer Liste enthalten ist
- Überprüfen, ob eine Liste leer ist
- Die Länge einer Liste bestimmen

Eigenschaften

- Duplikate möglich
- Kontrollierte Reihenfolge von Elementen (über Parameter `index` vgl. Kap. 9.3)



Annahme

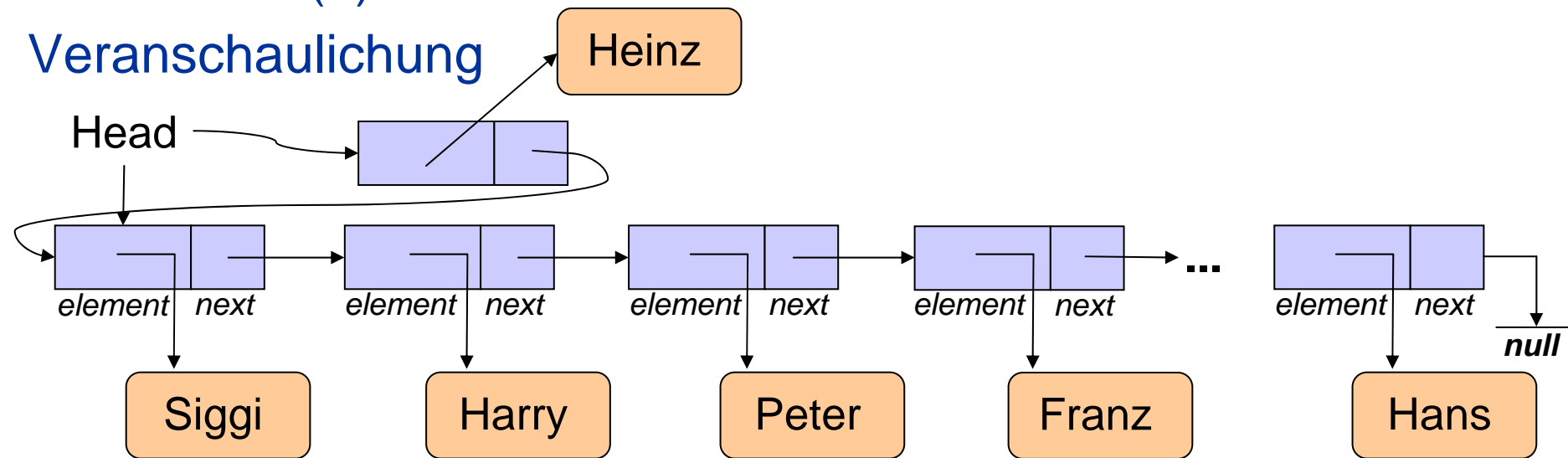
- Head zeigt auf das erste Element einer Liste

Vorgehen

- Erzeugen eines neuen Knotens, der das Element aufnimmt
- Zeigt auf das erste Element der Liste
- Head so ändern, dass es auf den neuen Knoten zeigt

Aufwand: $O(1)$

Veranschaulichung



Ähnlich: Löschen erster Knoten $O(1)$, Lesen erster Knoten $O(1)$

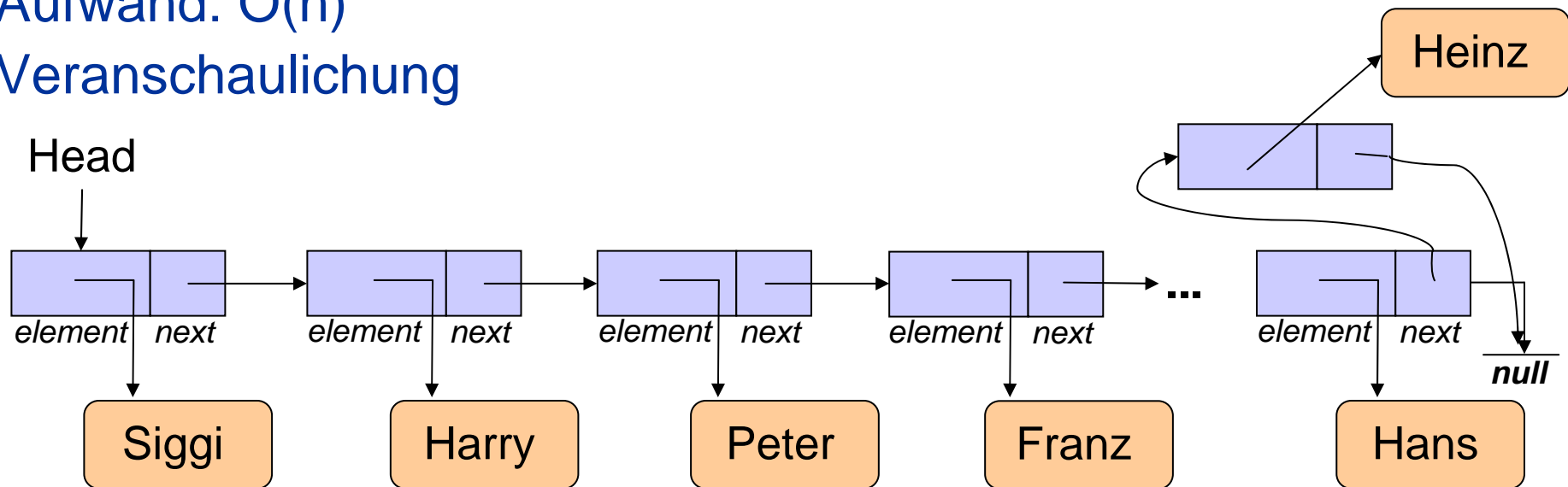


Vorgehen

- Erzeugen eines neuen Knotens der das Element aufnimmt
- Letzter Knoten muss durch Verfolgung der Zeiger ermittelt werden
- Letzter Knoten i.d.R. als Tail bezeichnet
- Zeiger des letzten Knotens ändern, so dass er auf den neuen Knoten zeigt

Aufwand: $O(n)$

Veranschaulichung



Ähnlich: Löschen letzter Knoten $O(n)$, Lesen letzter Knoten $O(n)$

Ziel

- Vereinfachung der Operationen am Listende

Verbesserung

- Knoten kennen nicht nur ihren Nachfolger sondern auch ihren Vorgänger

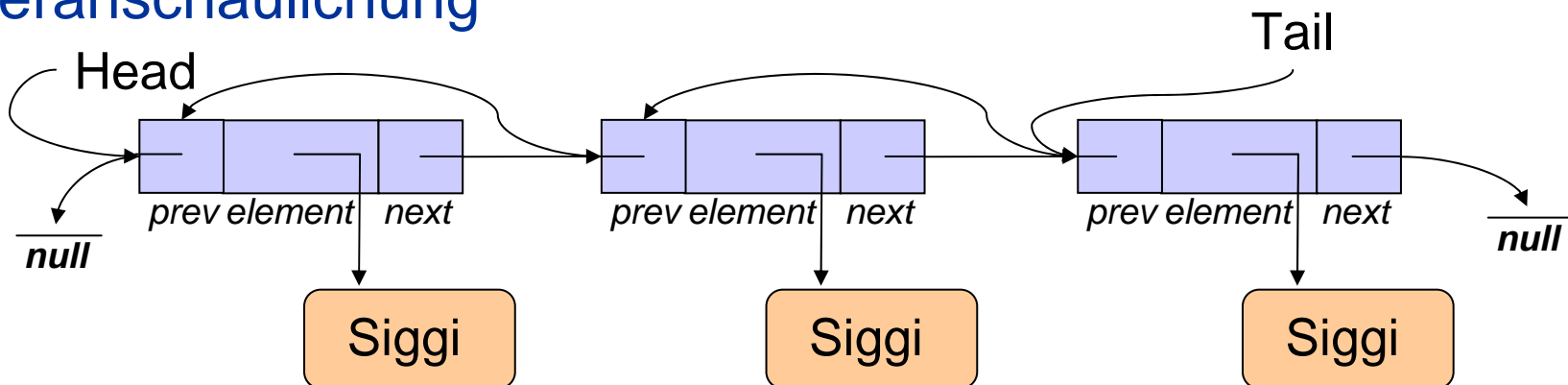
Erster Knoten: Head

Letzter Knoten: Tail

Letztes Element

- Über `getLast` über Tail mit konstantem Aufwand $O(1)$

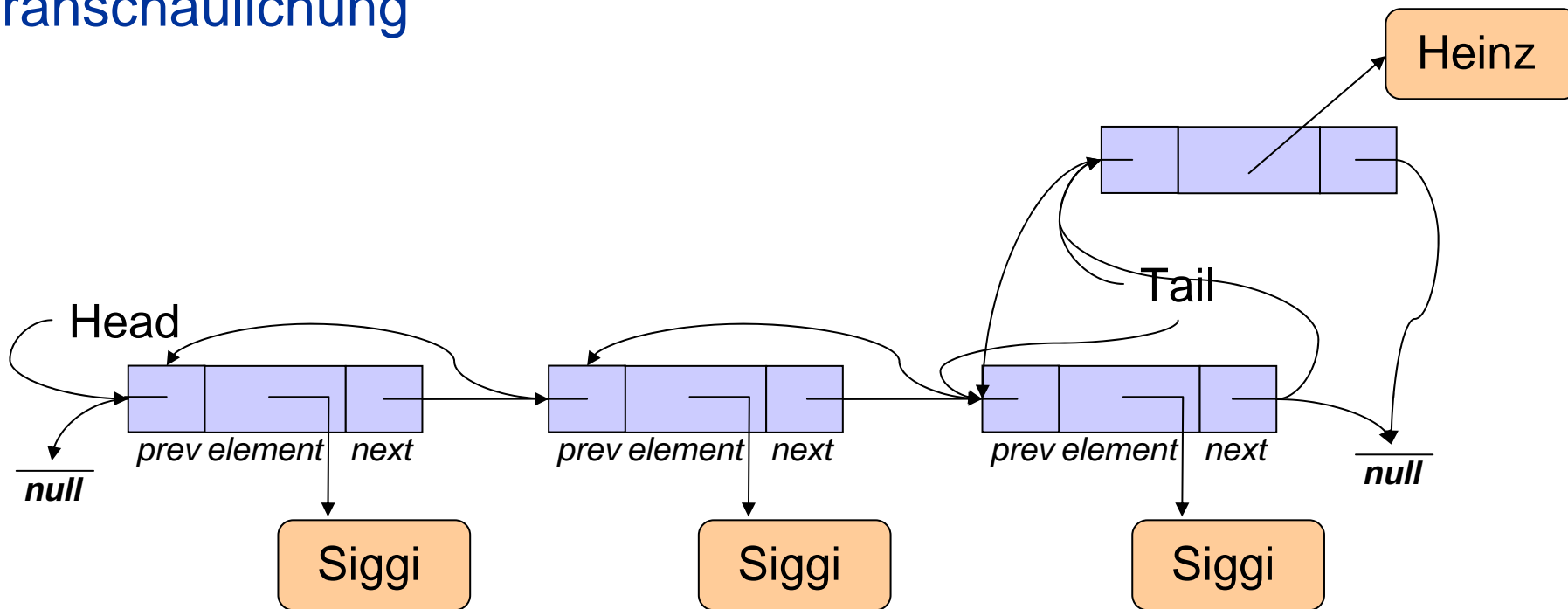
Veranschaulichung



Anhängen eines Elements am Ende

- Über Tail den bisher letzten Knoten ermitteln
- Dessen next-Zeiger auf neuen Knoten setzen
- Prev-Zeiger des neuen Knotens auf bisherigen letzten Knoten setzen
- Aufwand: $O(1)$

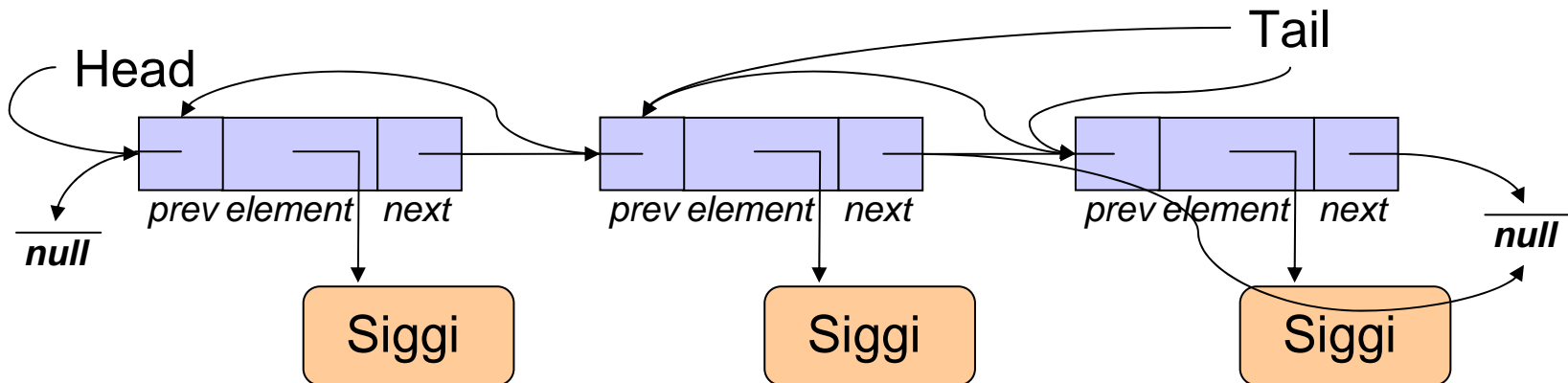
Veranschaulichung



Löschen des letzten Elements

- Über Tail den bisher letzten Knoten ermitteln
- Über prev-Zeiger dieses Knotens vorletzten Knoten ermitteln
- Dessen next-Zeiger auf null setzen
- Aufwand: $O(1)$

Veranschaulichung



Problemstellung

- Suche nach einer Möglichkeit, durch eine Liste zu navigieren
- Eine Liste soll gleichzeitig mehrmals traversiert werden können
- Deshalb muss Zustand der Traversierung unabhängig vom Listenobjekt sein

Gesucht

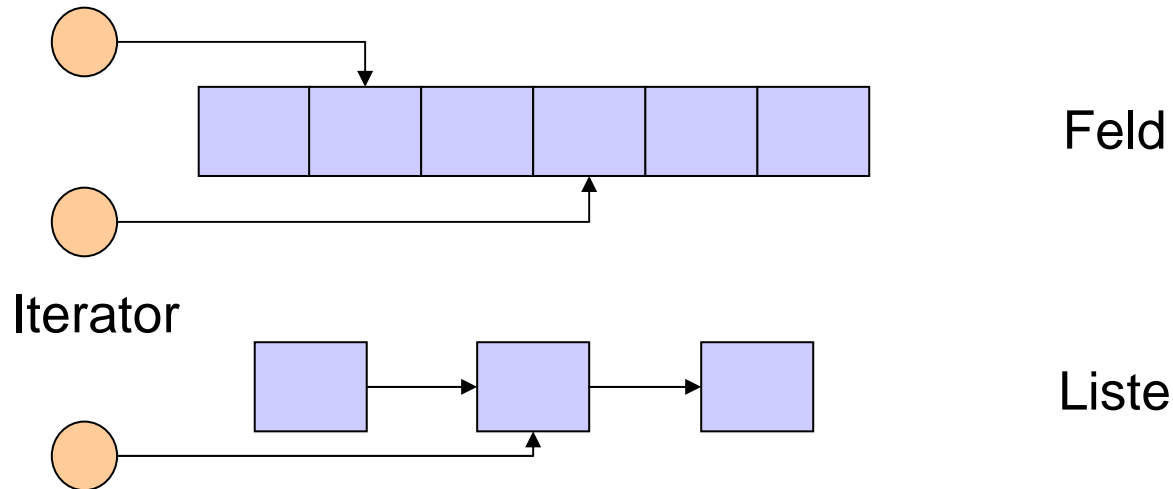
- Konzept zur einheitlichen Behandlung des Navigierens unabhängig von der internen Realisierung

Realisierung in Java

- Iterator
 - Ein Objekt genau für diesen Zweck
 - Verwaltet einen internen Zeiger auf die aktuelle Position in der zugrunde liegenden Datenstruktur
- Mehrere Iteratoren können gleichzeitig und unabhängig voneinander auf der gleichen Datenstruktur operieren



Veranschaulichung Iterator-Konzept



Die Schnittstelle `java.util.Iterator` definiert die folgenden Methoden

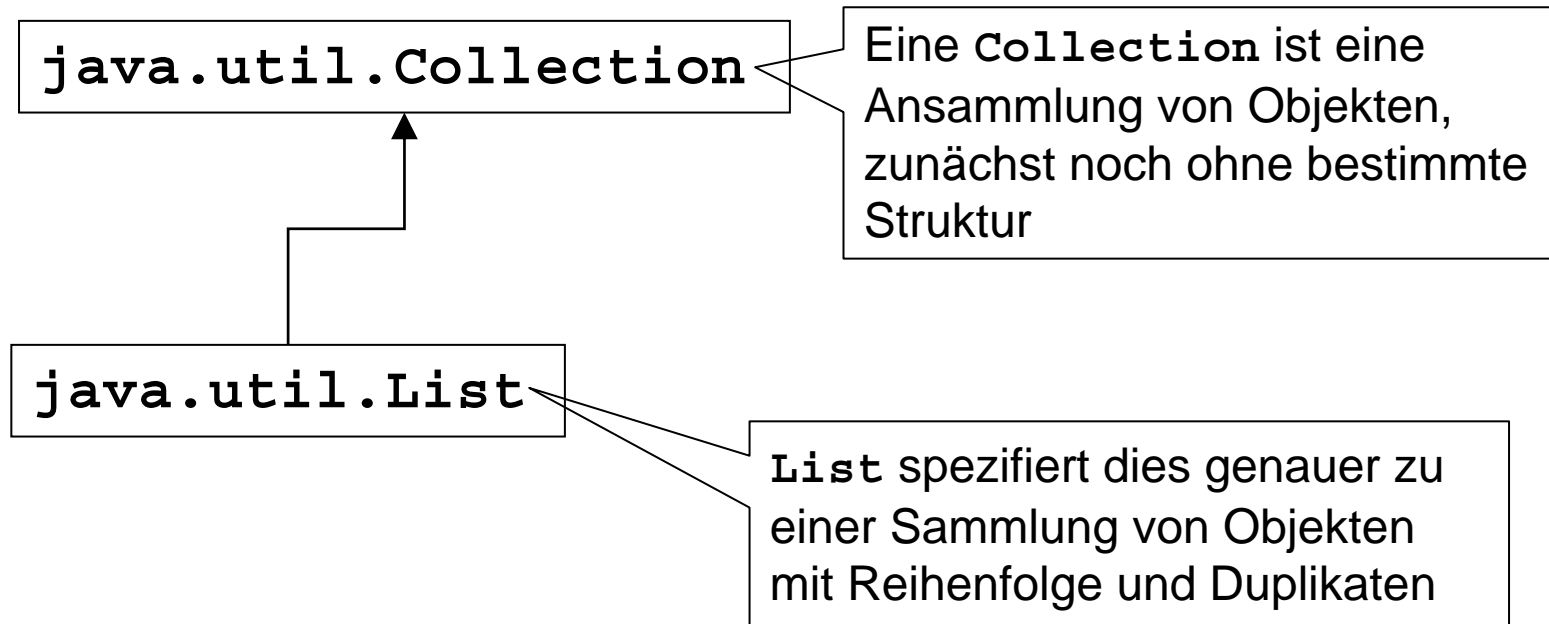
- `Object next()` Liefert das nächste Element in der Traversierung
- `Boolean hasNext()` prüft ob es noch ein nächstes Element gibt
- `void remove()` erlaubt, das aktuelle Element zu löschen

- mit `list.iterator()` erhält man ein (neues) Iteratorobjekt zu einer Liste `list`

Liste als Spezialfall einer dynamischen Menge

Java Collection Framework

- Ansammlung von Klassen, die dynamische Datenstrukturen bereitstellen
- Trennung von Schnittstellen und Implementierungen
 - Zu den Schnittstellen sind jeweils verschiedene Implementierungen verfügbar



```
// Element o ans Ende der Liste anhängen
```

```
boolean add (Object o);
```

```
// Alle Elemente aus Collection ans Ende der Liste anhängen
```

```
boolean addAll (Collection c);
```

```
// Entfernt das erste Vorkommnis von o
```

```
boolean remove(Object o);
```

```
// Entfernt alle Elemente aus Collection
```

```
boolean removeAll (Collection c);
```

```
// Alle Elemente aus der Liste entfernen
```

```
void clear();
```

```
// Überprüfe, ob Element o enthalten ist
```

```
boolean contains (Object o);
```



```
// Prüfen, ob eine Liste leer ist
```

```
boolean isEmpty();
```

```
// Liefert die Anzahl der Elemente der Liste
```

```
int size();
```

```
// Liefert einen Iterator für diese Liste zurück
```

```
Iterator iterator();
```

```
...
```

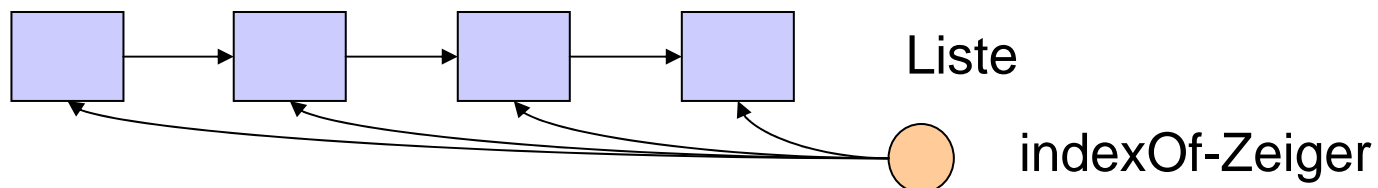


Keine konkrete Implementierung für Collection

- Ableiten weiterer Schnittstellen

Java-Lösung: Das **interface** `java.util.List` realisiert die abstrakte Idee einer Liste.

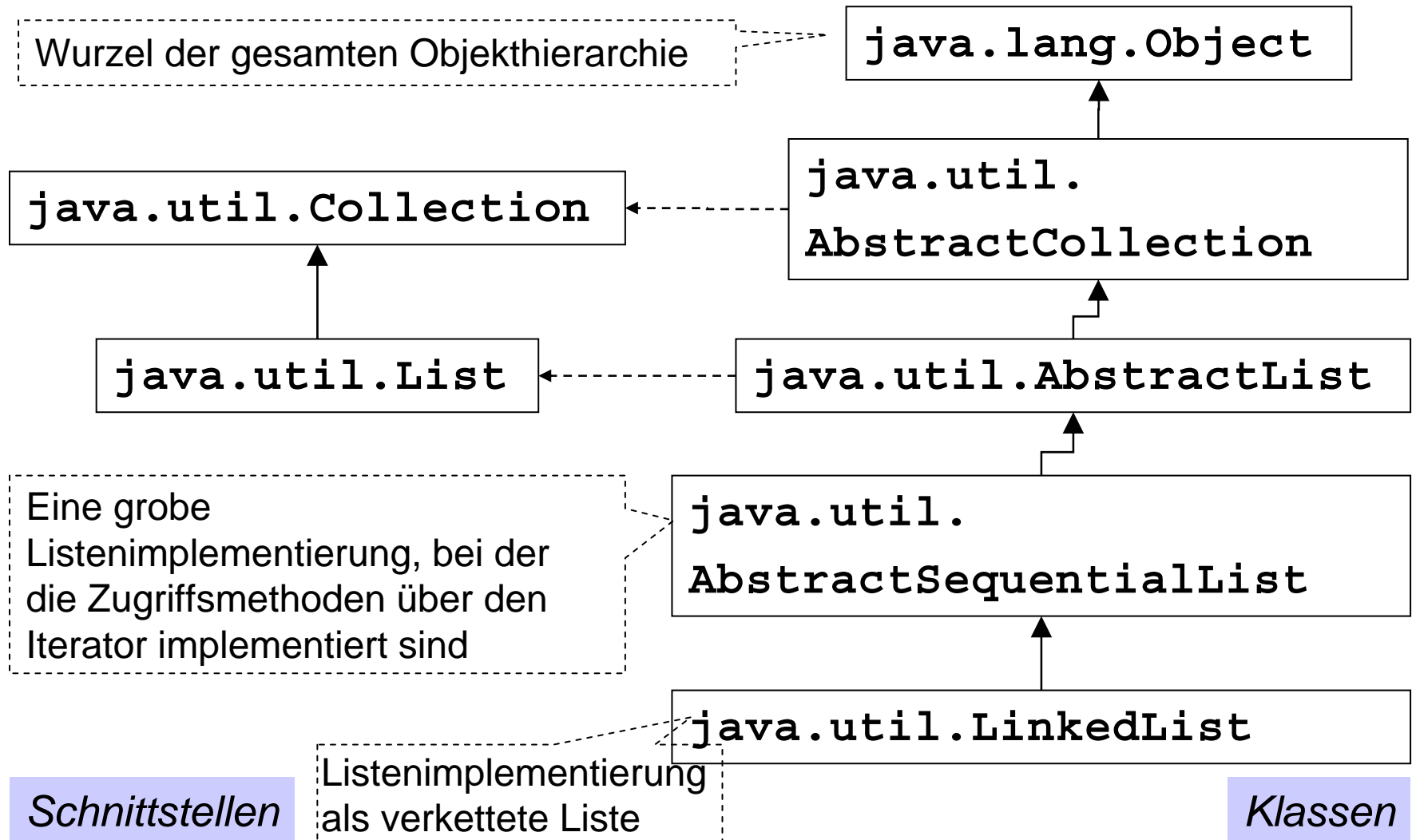
- Ein Interface implementiert noch keine konkreten Methoden.
- Es zeigt die Grundidee einer Liste auf
- Die konkrete Implementierung erfolgt erst später (→ z.B. als `java.util.LinkedList`)
- List definiert eine Schnittstelle für geordnete Mengen
 - Einfügereihenfolge bleibt erhalten
 - Positioniertes Auslesen bzw. Einfügen von Elementen ist möglich



```
// Element o an der Stelle index einfügen  
void add (int index, Object o);  
  
// Liefert das Element an der Stelle index  
Object get (int index);  
  
// Ermittelt die erste Position des Objekts o  
Object indexOf (Object o);  
  
// Entfernt das Element an der Stelle index  
Object remove (int index);  
  
// Ersetzt Element an Position index durch o  
// und gibt dieses zurück  
Object set (int index, Object o);  
  
...
```



Die Implementierung basiert auf dem Prinzip der Code-Wiederverwendung und Spezialisierung durch Unterklassen:



Implementierung

- Unsere Implementierung orientiert sich weitgehend an den Klassen aus `java.util.*`
- Einige Sonderfälle und selten genutzte Funktionen werden nicht betrachtet.
- Ausnahme-Behandlung weitgehend ausgeklammert

Vorteil:

- Kenntnis der Java-Klassenbibliothek wird vertieft
- Späterer Praxiseinsatz problemlos möglich
(Standard-Schnittstellen)



Die folgende Implementierung orientiert sich weitgehend an der Klasse `java.util.LinkedList`:

```
// Klasse für Einträge. Jeweils mit einem
// Objekteintrag und einem Verweis auf das nachfolgende
// Element
private static class Entry {
    Object element;
    Entry next;

    Entry(Object element, Entry next) {
        this.element = element;
        this.next = next;
    }
}
```



```
public class LinkedList extends AbstractSequentialList
    implements List
{
    // Kopfelement erzeugen, Größe auf 0
    private Entry header = new Entry(null, null);
    private int size = 0;

    // Konstruktor der Klasse
    public LinkedList() {
        header.next = header;
    }
}
```

 $O(1)$ 

```
// Hilfsmethode zum Einfügen von o nach Eintrag e
private Entry addAfter(Object o, Entry e) {
    Entry newEntry = new Entry(o, e.next);
    e.next = newEntry;
    size++;
    return newEntry;
}
```

 $O(1)$

```
// Hilfsmethode, die den Eintrag an der Stelle
// index liefert
private Entry entry(int index) {
    Entry e = header;
    for (int i = 0; i <= index; i++) { e = e.next; }
    return e;
}
```

 $O(n)$

```
// Hilfsmethode zum Entfernen eines Eintrags
private void remove(Entry e) {
    Entry p;
    for ( p = header; p.next != e;) { p = p.next; }
    p.next = e.next;
    size--;
}
```

 $O(n)$ 

// Eigentliche Funktion zum Hinzufügen von Objekten

```
public void add(int index, Object o) {  
    if (index==0) addAfter(o, head)  
    else  $O(1)$  addAfter(o, entry(index));  
}
```

$O(n)$

$O(n)$

// Objekt am Anfang einfügen

```
public boolean add(Object o) {  
    addAfter(o, header);  
    return true;  
}
```

$O(1)$

$O(1)$

// Liste leeren

```
public void clear() {  
    header.next = header;  
    size = 0;  
}
```

$O(1)$

// Überprüfen, ob Element o enthalten ist

```
public boolean contains(Object o) {  
    // o == null wird nicht behandelt  
    for (Entry e = header.next; e != header; e = e.next) {  
        if (o.equals(e.element)) return true;  
    }  
    return false;  
}
```

$O(n)$



// Liefert das Objekt an der Stelle index

```
public Object get(int index) {  
    return entry(index).element;  
}
```

$O(n)$

$O(n)$

// Entfernt das Objekt an der Stelle index

```
public Object remove(int index) {  
    Entry e = entry(index);  
    remove(e);  
    return e.element;  
}
```

$O(n)$

$O(n)$

$O(n)$

// Entfernt das erste Vorkommen eines Objekts

```
public boolean remove(Object o) {  
    ...  
}
```

$O(n)$



```
// Gebe Listenlänge zurück  
int size() { return size; }
```

 $O(1)$

```
// Überprüfe, ob Liste leer ist  
boolean isEmpty() { return size==0; }
```

 $O(1)$

```
// Konkrete Implementierung des Iterators in  
// LinkedList. Details werden hier nicht betrachtet.  
private class ListItr implements Iterator {  
    ...  
}  
  
// Liefert einen Iterator für die Liste  
public Iterator iterator() {  
    return new ListItr();  
}  
  
...  
}
```



Nunmehr: Betrachtung des Allgemeinfalls Mengen von Elementen

- Keine Duplikate
- Keine feste Reihenfolge von Elementen zugesichert

Zuerst: Definition einer Mengen-Schnittstelle mit Hilfe von Polymorphie

- Wir akzeptieren einen Typ als Parameter für die Klasse
- In Java so nicht möglich, aber mit:
 - ➔ GenericJava (<http://www.cis.unisa.edu.au/~pizza/gj/>)



Kapselung:

```
module Set (Set, CreateSet, single, insert, isElem, delete,  
            union, intersect, diff, isEmpty) where
```

Datentyp:

```
data Set  $\alpha$  = CreateSet | Add (Set  $\alpha$ )  $\alpha$ 
```

Signatur:

Zusätzlich: `equals` für Mengenvergleich, `containsAll` für Teilmengenbeziehung, `clear` für Leeren der Menge

```
single      ::  $\alpha \rightarrow \text{Set } \alpha$ 
```

```
add          insert      :: Eq  $\alpha \Rightarrow \text{Set } \alpha \rightarrow \alpha \rightarrow \text{Set } \alpha$ 
```

```
remove       delete     :: Eq  $\alpha \Rightarrow \text{Set } \alpha \rightarrow \alpha \rightarrow \text{Set } \alpha$ 
```

```
addAll       union      :: Eq  $\alpha \Rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha$ 
```

```
retainAll    intersect  :: Eq  $\alpha \Rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha$ 
```

```
removeAll    diff       :: Eq  $\alpha \Rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha$ 
```

```
contains     isElem     :: Eq  $\alpha \Rightarrow \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Bool}$     Assoziativer
```

```
isEmpty      isEmpty    :: Set  $\alpha \rightarrow \text{Bool}$                             Zugriff!
```

Navigierender Zugriff: `iterator`



```
public interface Set<A> extends Collection<A>
{
    // Liefert die Kardinalität der Menge
    int size();

    // Überprüft, ob die Menge leer ist
    boolean isEmpty();

    // Überprüft, ob die Menge das Element o enthält
    boolean contains(A o);

    // Liefert einen Iterator für die Menge zurück
    Iterator<A> iterator();

    // Fügt das angegebene Element zur Menge hinzu
    boolean add(A o);
}
```



```
// Entfernt das Element o aus der Menge
```

```
boolean remove(A o);
```

```
// Überprüft, ob die Menge alle Elemente aus c
```

```
// enthält - Teilmengenbeziehung
```

```
boolean containsAll(Collection<A> c);
```

```
// Fügt alle Elemente von c zur Menge hinzu -
```

```
// entspricht Vereinigung der Mengen
```

```
boolean addAll(Collection<A> c);
```

```
// Entfernt alle Elemente von c aus der Menge (sofern
```

```
// enthalten) - entspricht Differenz der Mengen
```

```
boolean removeAll(Collection<A> c);
```



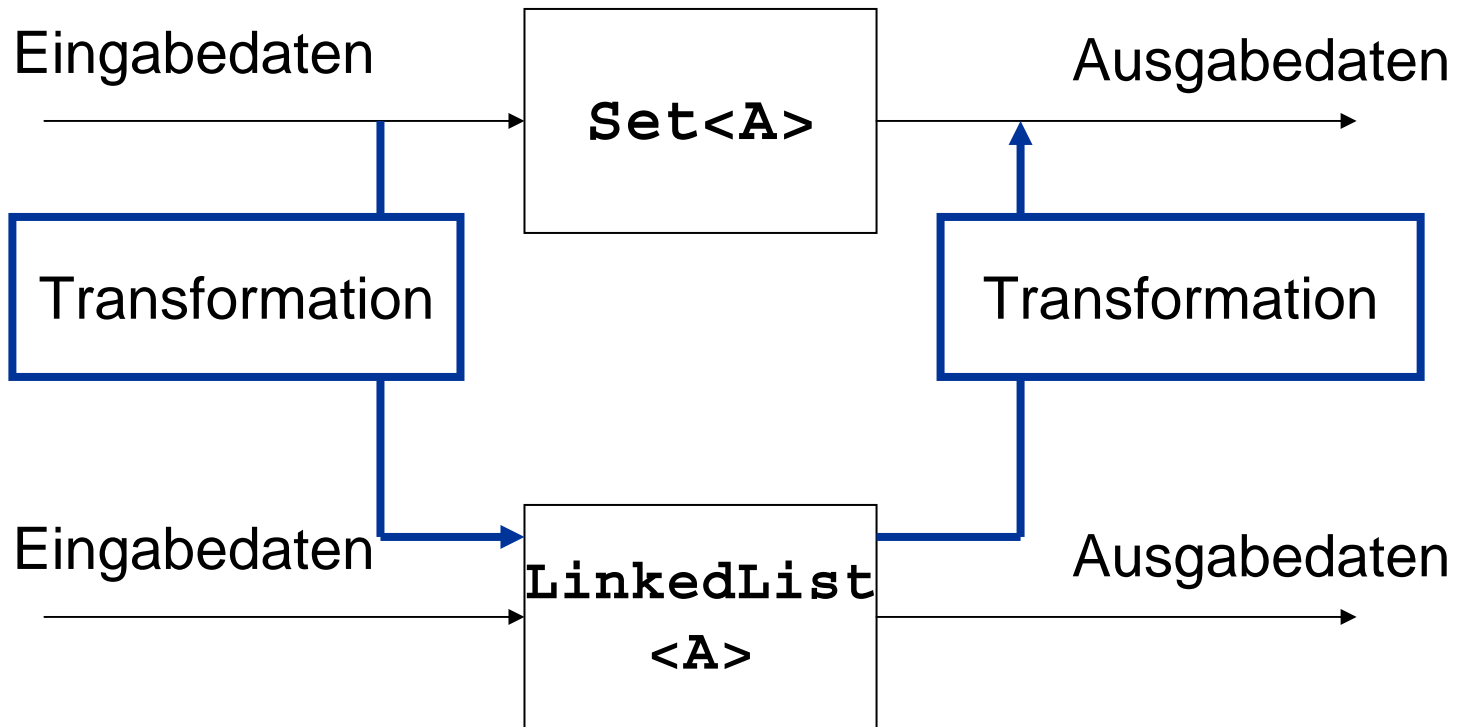
```
// Behält die Elemente der Menge, die auch in c
// enthalten sind - entspricht Schnittmenge der Mengen
boolean retainAll(Collection<A> c);

// Leert die Menge
void clear();

// Überprüft die Gleichheit zweier Mengen
boolean equals(Object o);
}
```



Verwende Klasse `LinkedList<A>` zur Implementierung von `Set<A>`



Fehlen einer Ordnung in der Menge:

- dadurch keine Beschränkungen,
- aber auch keine irgendwie geartete Nutzbarkeit einer Beschränkung.
- Suche nur erschöpfend durch sequentiellen Listendurchlauf.
- Einfügen eines neuen Elements kann freizügig gehandhabt werden, am einfachsten am Listenanfang. Jedoch: Problem bei der Duplikatvermeidung.
- Einfache Verkettung genügt für Löschen, da bei sequentiellm Suchdurchlauf Vorgänger bekannt ist.
- Universell einsetzbar für alle Mengenklassen.
- Besonders flexibel für Mengen unbeschränkter Größe.



```
public class SetAsList<A> extends AbstractSet<A>
    implements Set<A>
{
    // Zugrunde liegende Liste
    protected List<A> list;

    // Leere Liste wird erzeugt
    public SetAsList() {
        list = new LinkedList<A>();
    }

    // Liefert die Kardinalität der Menge
    public int size() {
        return list.size();
    }

    // Überprüft, ob die Menge leer ist
    public boolean isEmpty() {
        return list.isEmpty();
    }
}
```

 $O(1)$ $O(1)$ $O(1)$ $O(1)$ 

// Liefert einen Iterator für die Liste zurück

```
public Iterator<A> iterator() {  
    return list.iterator();  
}
```

$O(1)$

// Überprüft, ob die Menge das Element o enthält

```
public boolean contains(A o) {  
    return list.contains(o);  
}
```

$O(n)$

$O(n)$

// Fügt das angegebene Element zur Liste hinzu

```
public boolean add(A o) {  
    if (contains(o)) return false;  
    return list.add(o);  
}
```

$O(n)$

$O(1)$

$O(n)$

// Entfernt das Element o aus der Menge

```
public boolean remove(A o) {  
    return list.remove(o);  
}
```

$O(n)$

$O(n)$



// Überprüft, ob die Menge alle Elemente aus c
// enthält - Teilmengenbeziehung

```
public boolean containsAll(Collection<A> c) {
    Iterator<A> i = c.iterator();
    while (i.hasNext()) {
        if (!contains(i.next())) return false;
    }
    return true;
}
```

$O(n^2)$

$O(n)$ } *n-mal*

// Fügt alle Elemente von c zur Menge hinzu -
// Vereinigung

```
public boolean addAll(Collection<A> c) {
    Iterator<A> i = c.iterator();
    boolean setChanged = false;
    while (i.hasNext()) {
        setChanged = add(i.next()) || setChanged;
    }
    return setChanged;
}
```

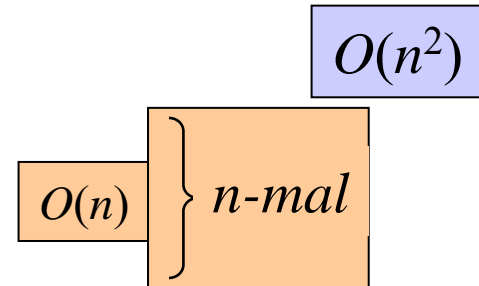
$O(n^2)$

$O(n)$ } *n-mal*



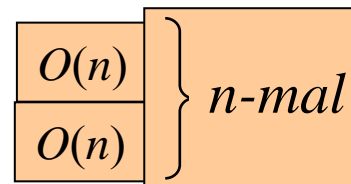
// Entfernt alle Elemente von c aus der Menge - Differenz

```
public boolean removeAll(Collection<A> c) {
    Iterator<A> i = c.iterator();
    boolean setChanged = false;
    while (i.hasNext()) {
        setChanged = remove(i.next()) || setChanged;
    }
    return setChanged;
}
```



// Behält die Elemente der Menge, die auch in c
// enthalten sind - Schnitt

```
public boolean retainAll(Collection<A> c) {
    Iterator<A> i = iterator();
    boolean setChanged = false;
    while (i.hasNext()) {
        A a = i.next();
        if (!c.contains(a)) {
            i.remove();
            setChanged = true;
        }
    }
    return setChanged;
}
```



$O(n^2)$



// Leert die Menge

```
public void clear(){  
    list.clear();  
}
```

$O(1)$

// Überprüft die Gleichheit zweier Mengen

```
public boolean equals(Object o) {  
    if (o == this) return true;  
    if (!(o instanceof Set)) return false;  
  
    Collection c = (Collection) o;  
    if (c.size() != size()) return false;  
    return containsAll(c);  
}  
}
```

$O(n^2)$

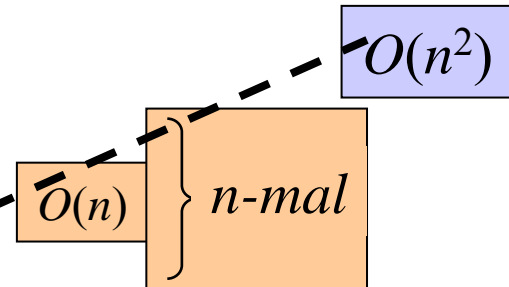


Wir erinnern uns

```
// Überprüft, ob die Menge alle Elemente aus c
// enthält - Teilmengenbeziehung
```

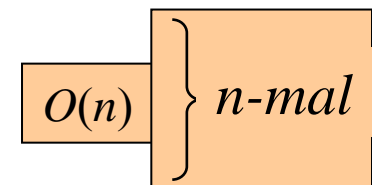
```
public boolean containsAll(Collection<A> c) {
    Iterator<A> i = c.iterator();
    while (i.hasNext()) {
        if (!contains(i.next())) return false;
    }
    return true;
}
```

Aufwand
reduzierbar?



```
// Fügt alle Elemente von c zur Menge hinzu -
// Vereinigung
```

```
public boolean addAll(Collection<A> c) {
    Iterator<A> i = c.iterator();
    boolean setChanged = false;
    while (i.hasNext()) {
        setChanged = add(i.next()) || setChanged;
    }
    return setChanged;
}
```



Gegeben: Grundmenge U , vollständige Ordnung \leq hierauf,
Mehrfachmenge M mit Elementen $e \in U$.

Gesucht: Anordnung der Elemente aus M gemäß \leq .

Formulierung:

- Gegeben: Liste $M = [e_0, e_1, \dots, e_{n-1}]$.
- Gesucht: Liste $L = [e_{j_0}, e_{j_1}, \dots, e_{j_{n-1}}]$
mit Nachbedingung $e_{j_0} \leq e_{j_1} \leq \dots \leq e_{j_{n-1}} \wedge \text{perm}(M, L)$.

L ist Permutation von M .



In der Praxis:

- Elemente i sind Objekte mit ausgezeichneten Feldern, deren Werte zusammengesetzt den (Sortier-) Schlüssel bilden.
- Oft erwünscht: Stabiles Sortieren: Elemente mit gleichem Schlüssel erscheinen in M und L in der gleichen Reihenfolge (Verschärfung der Nachbedingung!).

Kosten/Nutzen-Analyse:

- Sei m : Zahl der Suchvorgänge über die Lebensdauer der Menge
- Vorteil falls

$$T_s + m \cdot T' < m \cdot T$$

mit T_s : Sortieraufwand,

T' : Suchaufwand sortiert,

T : Suchaufwand unsortiert.



- Das Interface `java.lang.Comparable` definiert eine einzige Methode:

```
public abstract int compareTo(Object o);
```

- Alle Implementierungen sollten
 - Einen negativen Wert zurückgeben, wenn gemäß der gegebenen Ordnung dieses Objekt „kleiner“ als das übergebene Objekt ist:
 $\text{this} < o$
 - 0 zurückgeben, wenn die beiden Objekte „gleich groß“ sind (oder die relative Sortierung gleichgültig ist):
 $\text{this} = o$
 - Einen positiven Wert zurückgeben, wenn dieses Objekt „größer“ als das übergebene Objekt ist:
 $\text{this} > o$
- Wir gehen im Folgenden davon aus, dass Elemente von sortierten Listen dieses Interface implementieren!



Idee

- Vereinigung bei Sortierung gemäß Reißverschlussprinzip

Hinweis

- Im Folgenden wird statt `java.util.Iterator<A>` `java.util.ListIterator<A>` verwendet
- Dadurch kann u.a. die Methode `add(A o)` verwendet werden, welche ein Element `o` vor der aktuellen Position des Iterators einfügt.
- Außerdem verwenden wir eine Methode `previous()`, mit der man den Iterator auf das vorangegangene Traversierungselement zurücksetzen kann.

```
public boolean addAll(SortedList<A> s) {  
    // beide Mengen seien vorsortiert  
    ListIterator<A> i = listIterator();  
    ListIterator<A> j = s.listIterator();
```



```
// Die von i verschiedenen Elemente von j gehen nach i
while (i.hasNext() && j.hasNext()) {
    // Nimm Element von i und j
    A a1 = i.next();
    A a2 = j.next();
    // Gehe auf i solange weiter, bis ein Element
    // gefunden wird, das größer ist als a2 (aus j)
    while (((Comparable) a1).compareTo((Comparable) a2) <= 0
        && i.hasNext()) {
        a1 = i.next();
    }
    // Jetzt ist gerade a1 > a2 geworden oder i ist am Ende
    if (i.hasNext()) {
        // Wenn i nicht am Ende ist, muss man 1 Element zurück
        i.previous();
    }
    // Dann a2 einfügen
    i.add(a2);
}
// Die übrigen Element von j
// (die alle größer als die von i sind) am Ende einfügen
while (j.hasNext()) {
    i.add(j.next());
}
return s.size() > 0;
}
```

 $O(n)$

Dazu ist aber noch der Aufwand für das Sortieren hinzuzufügen!



Ziel:

- Betrachtung von Sortierverfahren, für die sich die verkettete Liste als Datenstruktur für die zu sortierende Liste besonders eignet.
- Dies ist der Fall, wenn das Verfahren die Elemente der Liste entlang ihrer Anordnung aufgreifen will.



Grundidee (aus Informatik I):

- Lösche nacheinander die Minima aus einer Liste und füge sie hinten an die anfangs leere Ergebnisliste an

selsort [] = []

selsort xs = kleinstes : selsort (ohne kleinstes xs)

where kleinstes = minimum xs

ohne x (y:ys) | x == y = ys

| otherwise = y : (ohne x ys)

selsort [10,9,8,-1,3,14] > [-1,3,8,9,10,14]

selsort xs ist korrekt:

- Induktionsanfang: selsort [] ist trivialerweise korrekt
- Induktionsannahme: selsort ist korrekt für Listen der Länge $0 \leq n-1$. Dann ist selsort auch korrekt für Listen der Länge n, wenn die Funktion ohne x xs für Listen der Länge n korrekt ist.
- Innere Induktion: Beweis der Korrektheit von ohne x xs



Verfahren: Lösche fortlaufend Maxima aus vorgegebener Liste M und stelle sie (anfangs leerer) Ergebnisliste L voran.

Problemlösung durch Induktion:

- Induktionsanfang: Leere Liste L und einelementige Liste L sind sortiert.
- Induktionsschritt: Sei $(n-1)$ -elementige Liste L sortiert. Nächstes Element aus M ist kleiner als alle Elemente in L und wird als erstes Element in L eingefügt. Daher ist n -elementige Liste L sortiert.
- Induktionsende: Wenn M leer, befinden sich alle Elemente in L .

Formulierung als rekursive Methode der Klasse `LinkedList`.



- Die Lösung soll die Liste M nicht zerstören.
- Wir wollen nicht mit jedem Rekursionsschritt eine neue (Teil-) Liste erstellen, daher brauchen wir wieder eine äußere Methode zur Einbettung der rekursiven Methode, in der insbesondere die Liste L vereinbart wird.



```
public LinkedList selsortRec() {
    // Kopie der Ausgangsliste
    LinkedList M = new LinkedList();
    M.addAll(this);
    // Erzeuge leere Ergebnisliste L
    LinkedList L = new LinkedList();
    {P:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0 \wedge L \text{ sortiert}$ }
    // Aufruf der Rekursion
    return selsortRec(M, L);
    {Q:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0 \wedge L \text{ sortiert} \wedge M \text{ leer}$ }
}
```

P

$\neg b$

Auf dem Weg zur Schleifeninvariante



```
LinkedList selsortRec(LinkedList M, LinkedList L)
{
    if (!M.isEmpty()) { // b
        int m = M.maximum();
        M.remove(m);
        L.add(m);
        return selsortRec(M, L);
        {I:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0 \wedge L \text{ sortiert}$ }
    }
    else return L;
}
```

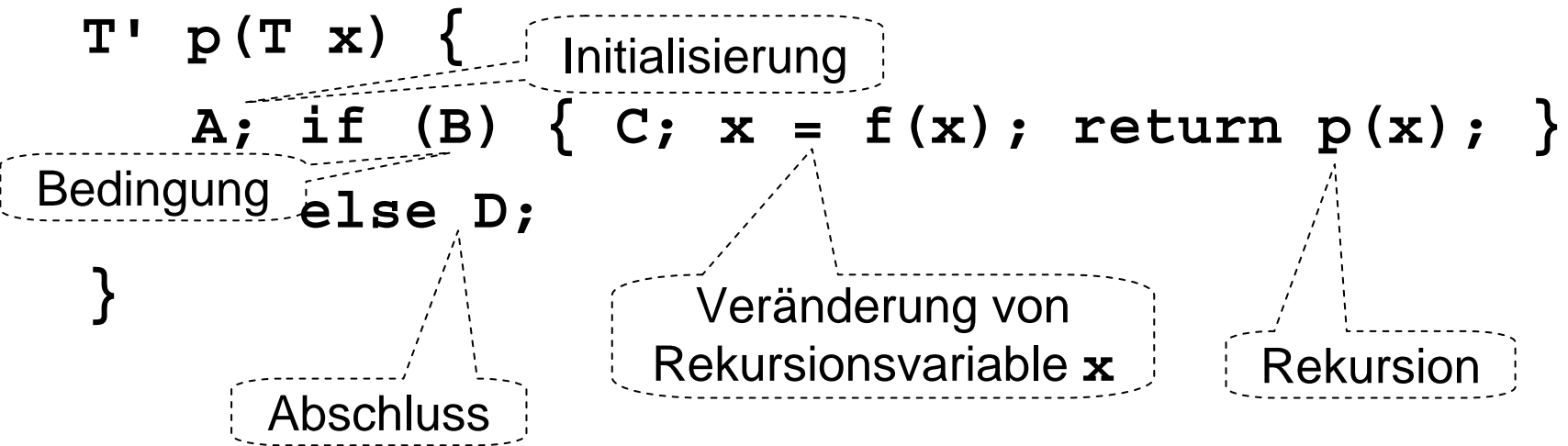
Schleifeninvariante



- Rekursive Lösungen stellen zwar elegante Formulierungen dar, aber ihre Berechnung ist häufig aufwändiger als eine äquivalente iterative Formulierung.
- Gesucht: Methodisches Überführen von Rekursionen in Iterationen.



Umformulierung der rekursiven Formulierung in folgendes Schema



Anmerkungen:

- Dieses Schema heißt rechtsrekursive Methode.
- Von geringerer Bedeutung sind linksrekursive Methoden, in deren Schema der rekursive Aufruf vor dem c-Teil liegt. Hier sind nämlich starke Einschränkungen zu beachten.



Die Innere Methode befindet sich bereits in diesem Schema!

```

LinkedList selsortRec(LinkedList M, L) {
    if (!M.isEmpty()) { // b
        B int m = M.maximum();
        M.remove(m);
        L.add(m);
        return selsortRec(M, L);
        {I:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
          $\wedge L$  sortiert}
    }
    else return L;
}

```

A leer

C ist hier leer!

Anweisung mit Veränderung der Rekursionsvariablen **M, L**

Rekursion

D



Nutze aus: Schema von zuvor ist äquivalent zu

```
T' p(T x) {  
    A; while (B) { C; x = f(x); A } D;  
}
```

Begründung: Vergleiche die beiden Kontrollflüsse!

```
T' p(T x) {  
    A; if (B) { C; x = f(x); return p(x); }  
    else D;  
}  
⇒  
T' p(T x) {  
    A; while (B) { C; x = f(x); A } D;  
}
```



Umformung:

```
LinkedList selsortRec(LinkedList M, L) {  
    {P:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0 \wedge L \text{ sortiert}$ }  
    while (!M.isEmpty()) { // b  
        int m = M.maximum();  
        M.remove(m);  
        L.add(m);  
    }  
    {Q:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0 \wedge L \text{ sortiert}$   
         $\wedge M \text{ leer}$ }  
    return L;  
}
```



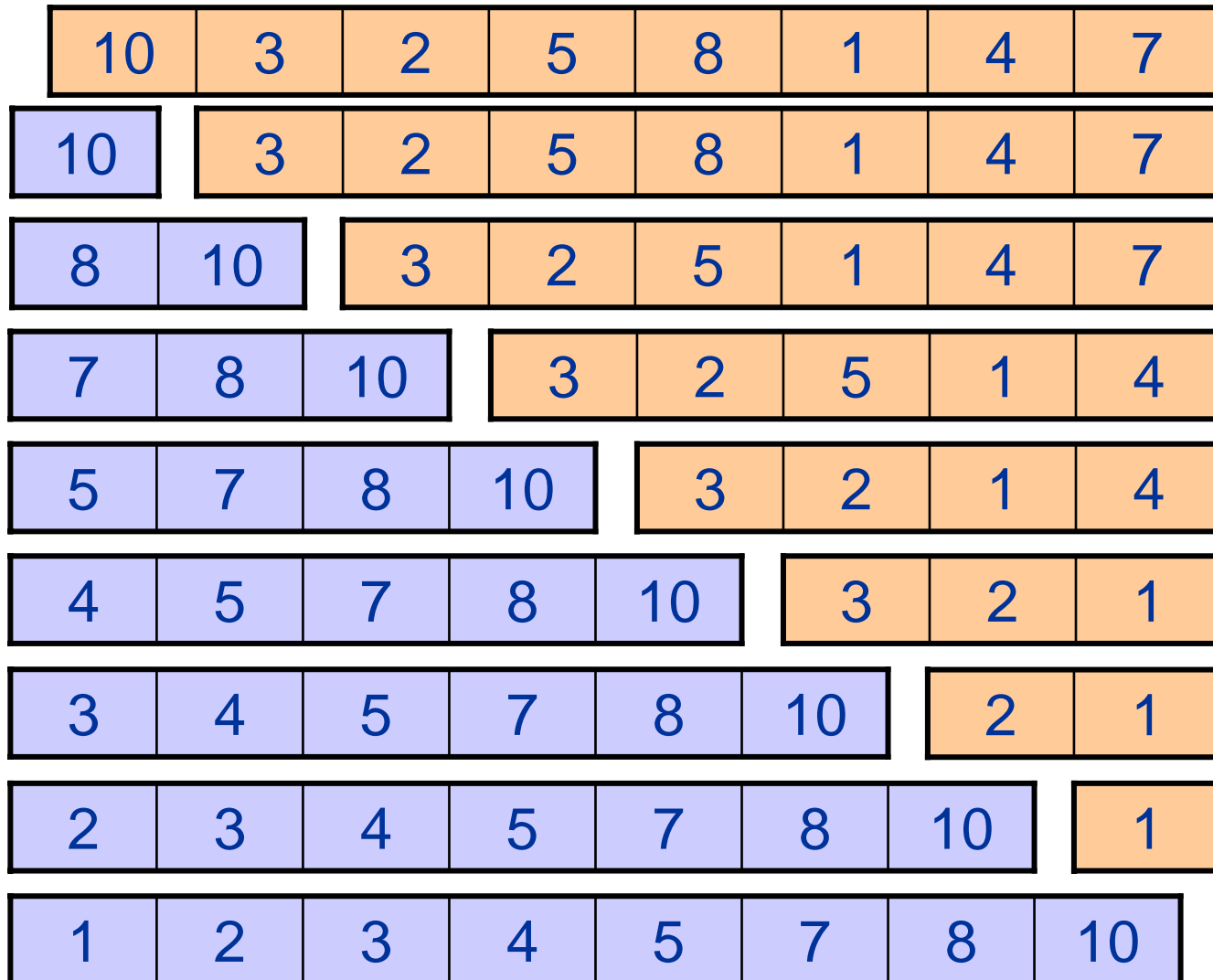
```
public LinkedList selsort() {
    // Kopie der Ausgangsliste this
    LinkedList M = new LinkedList();
    M.addAll(this);
    // Erzeuge leere Ergebnisliste L
    LinkedList L = new LinkedList();
    {P:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
       $\wedge L$  sortiert}
    while (!M.isEmpty()) {
        int m = M.maximum();
        M.remove(m);
        L.add(m);
    }
    {Q:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
       $\wedge L$  sortiert  $\wedge M$  leer}
    return L;
}
```

Teil der
Außen-
methode

Einbau der ehemals
rekursiven Lösung in
die Außenmethode
("offener Einbau")

Man schreibe erst die rekursive Lösung an und forme dann um in die äquivalente iterative Lösung. Auf diese Weise wird auch die Zusicherung der (partiellen) Korrektheit erleichtert.





```
public LinkedList selsort() {
    LinkedList M = new LinkedList();
    M.addAll(this);
    LinkedList L = new LinkedList();
    {P:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
         $\wedge L$  sortiert}
    while (!M.isEmpty()) {
        int m = M.maximum();
        M.remove(m);
        L.add(m);
    }
    {Q:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
         $\wedge L$  sortiert  $\wedge M$  leer}
    return L;
}
```

selsort ist korrekt:

- I gilt für die leere Liste L.
- Bei Verlassen der Schleife gilt I mit $M = \emptyset$:
 $L = M_0 \wedge L$ sortiert ("=" steht für Gleichheit von Mehrfachmengen).
- I ist Schleifeninvariante:
 Wiederherstellen von I durch `remove()` und `add()`
- Terminierung, da M monoton verkürzt wird.

selsort ist stabil

- sofern `maximum()` das in der Reihenfolge von M letzte Maximum nimmt und `remove()` auch dieses Element löscht (das ist in unserer Beispielimplementierung nicht der Fall, dort wird das erste genommen!).



```

public LinkedList selsort() {
    LinkedList M = new LinkedList();
    M.addAll(this);
    LinkedList L = new LinkedList();
    {P:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
         $\wedge L$  sortiert}
    while (!M.isEmpty()) {
        int m = M.maximum();
        M.remove(m);
        L.add(m);
    }
    {Q:  $\forall v \in M, v' \in L: v \leq v' \wedge M \cup L = M_0$ 
         $\wedge L$  sortiert  $\wedge M$  leer}
    return L;
}

```

im Mittel $n/2$ -mal

$O(1)$

$O(n)$

$O(n)$

n -mal

durch Merken der
Maximum-Position: $O(1)$

Daher $T_{\text{selsort}} = \Theta(n^2)$.