Einführung in die Programmierung mit Java

Teil 14: Verkettete Listen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

18. Januar 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 14: Verkettete Listen

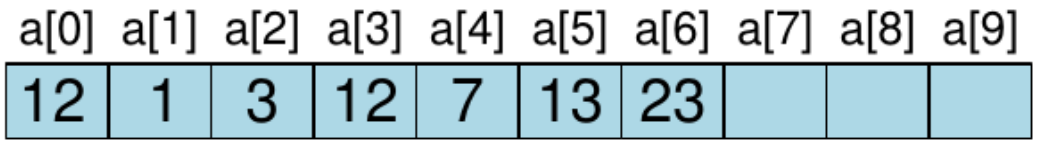
ArrayList

Zur Verwaltung einer gewissen Anzahl Elemente gleichen Typs haben wir bisher Arrays bzw. `ArrayList` verwendet:

```
public class ArrayList<E> implements List<E>{
  boolean isEmpty();                  // Array leer?
  int size();                         // Elemente im Array

  E get(int index);                   // Element bei Index lesen
  E set(int index, E element);        // Element bei Index ersetzen

  boolean add(E e);                   // Am Ende einfügen
  void add(int index, E e);           // Bei Index einfügen

  boolean remove(Object o);           // Erstes Element entfernen
  E remove( int index );              // Element bei Index löschen
  ...
}
```

Implementierung: ArrayList

`ArrayList<E>` verwendet intern klassische Arrays:



Geeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(1)$

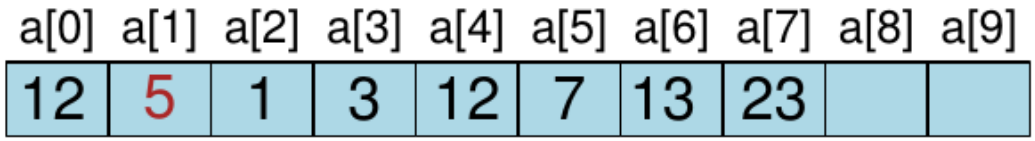
Ungeeignet, falls

- Anzahl der Elemente sich oft ändert, also Aufrufe von `a.add(e)` oder `a.remove(e)` sollten selten sein. $O(n)$

Hinzufügen eines Elementes benötigt Umkopieren aller folgenden Elemente. Die Programmbibliothek macht dies zwar automatisch für uns, aber es kostet den Anwender Rechenzeit!

Implementierung: ArrayList

`ArrayList<E>` verwendet intern klassische Arrays:



Geeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(1)$

Ungeeignet, falls

- Anzahl der Elemente sich oft ändert, also Aufrufe von `a.add(e)` oder `a.remove(e)` sollten selten sein. $O(n)$

Hinzufügen eines Elementes benötigt Umkopieren aller folgenden Elemente. Die Programmbibliothek macht dies zwar automatisch für uns, aber es kostet den Anwender Rechenzeit!

Interface List

Verallgemeinerung durch Interface `List<E>` für *geordnete Folgen* von Elementen gleichen Typs, möglicherweise mit Duplikaten.

```
public interface List<E> extends Collection<E>{
  boolean isEmpty();              // Liste leer?
  int size();                     // Anzahl Elemente in Liste

  E get(int index);               // Element bei Index lesen
  E set(int index, E element);    // Element bei Index ersetzen

  boolean add(E e);               // Am Ende einfügen
  void add(int index, E e);       // Bei Index einfügen

  boolean remove(Object o);       // Erstes Element entfernen
  E remove( int index );          // Element bei Index löschen
  ...
}
```

Interface List

Verallgemeinerung durch Interface `List<E>` für *geordnete Folgen* von Elementen gleichen Typs, möglicherweise mit Duplikaten.

```
public interface List<E> extends Collection<E>{
  boolean isEmpty();              // Liste leer?
  int size();                     // Anzahl Elemente in Liste

  E get(int index);               // Element bei Index lesen
  E se                                                    tzen

  bool
  void

  bool                                                    en
  E re                                                    hen
  ...
}
```

Tipp: Verwende für Variablen/Parameter immer `List<E>` anstatt `ArrayList<E>`, dann kann man nachträglich leicht eine andere Implementierung des Interfaces verwenden:

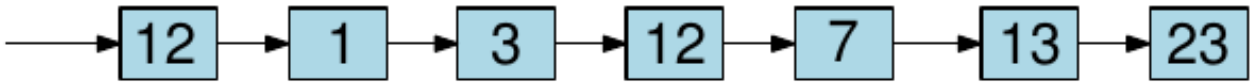
```
List<String> mylist = new ArrayList<>();
```

anstatt

```
ArrayList<String> mylist = new ArrayList<>();
```

Implementierung: LinkedList

`LinkedList<E>` implementiert `List<E>` als Kette von Objekten



Geeignet, falls

- Anzahl der Elemente sich oft ändert, also oft Aufrufe von `a.add(e)` oder `a.remove(e)` $O(1)$
- Wenn meistens sowieso *alle* Elemente der Reihe nach verwendet werden, z.B. mit Schleifen. $O(n)$

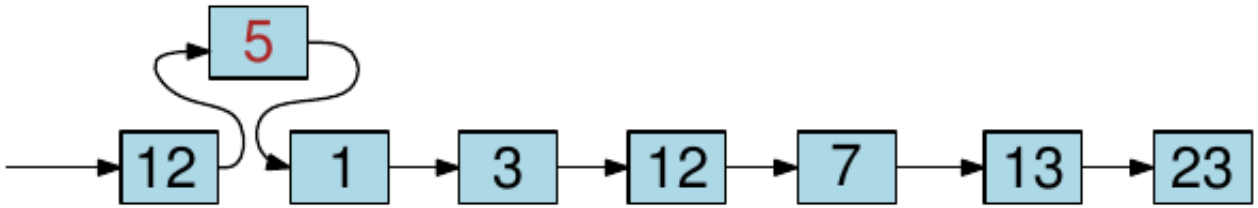
Ungeeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(n)$

Verändern der Elementanzahl benötigt nur das Umsetzen von Zeigern, aber wenn man ein spezielles Element sucht, muss man sich durchhangeln.

Implementierung: LinkedList

`LinkedList<E>` implementiert `List<E>` als Kette von Objekten



Geeignet, falls

- Anzahl der Elemente sich oft ändert, also oft Aufrufe von `a.add(e)` oder `a.remove(e)` $O(1)$
- Wenn meistens sowieso *alle* Elemente der Reihe nach verwendet werden, z.B. mit Schleifen. $O(n)$

Ungeeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(n)$

Verändern der Elementanzahl benötigt nur das Umsetzen von Zeigern, aber wenn man ein spezielles Element sucht, muss man sich durchhangeln.

Verkettete Listen

- Eine verkettete Liste besteht (wie eine Kette) aus einzelnen Gliedern.
- Jedes Glied enthält ein Datum, sowie einen Verweis auf das nächste Glied, eventuell einen zusätzlichen Verweis auf das vorhergehende Glied.
- Verkettete Listen dienen zur Verwaltung von Daten variabler Anzahl, auf die in der Regel sequentiell zugegriffen wird.
- Sie erlauben das Einfügen eines Elements an beliebiger Stelle in *konstanter Zeit* $O(1)$.

Einfügen in ArrayList: $O(n)$

Die Klasse LinkedList<E>

Die Klasse `java.util.LinkedList<E>` implementiert verkettete Listen.

Hierbei ist `E` ähnlich wie bei `ArrayList<E>` ein Typparameter.

Unter anderem gibt es die folgenden Methoden:

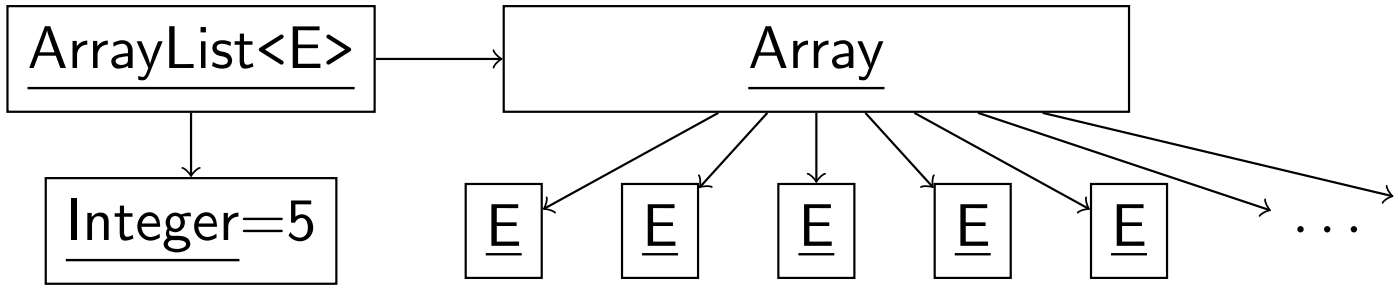
```
    void addFirst(E obj)
    void addLast(E obj)
    E    getFirst()
    E    getLast()
    E    removeFirst()
    E    removeLast()
```

Gleich zum Uebungsblatt 13

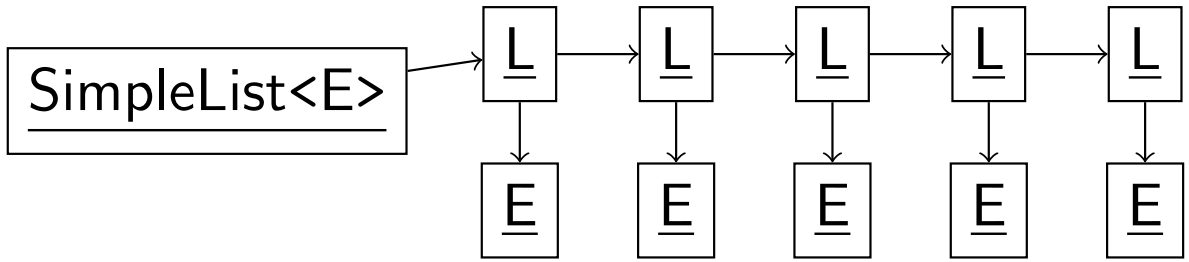
Weitere Methoden wie Einfügen an beliebiger Stelle werden über einen **Iterator** bereitgestellt.

Verweis-Strukturen

Array



Einfach verkettete Liste

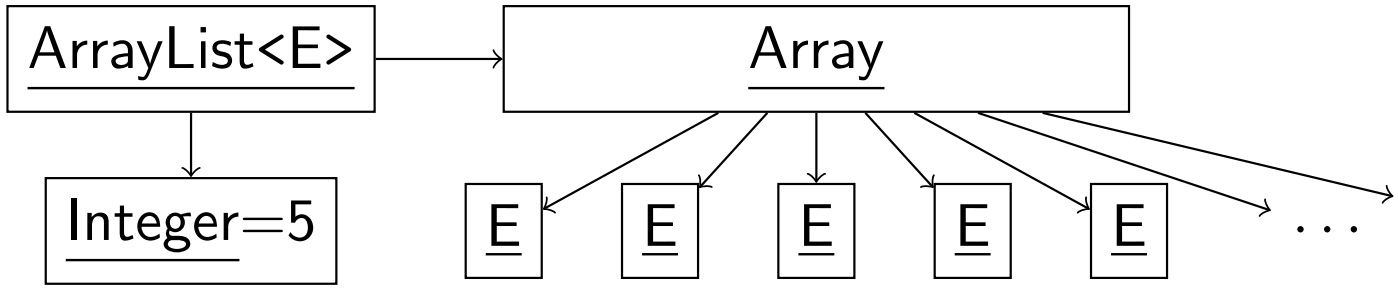


wobei
L = `Link<E>`

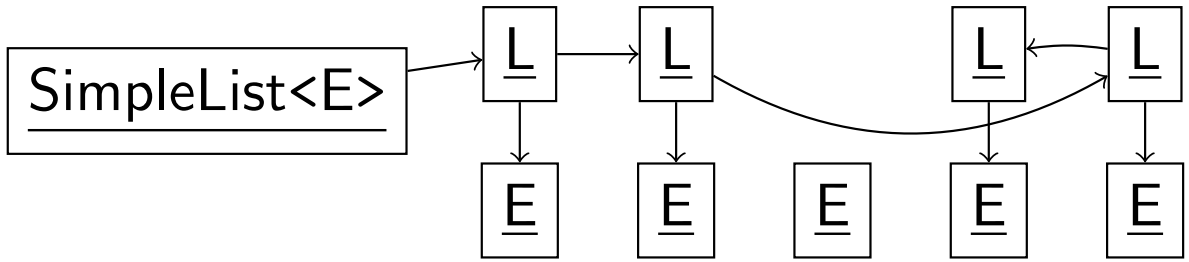
- `ArrayList` hat ein Array, darin liegen alle Verweise nacheinander im Speicher. Ein zu kleines Array muss komplett ausgetauscht werden.
- `SimpleList` speichert Verweise in privaten `Link`-Objekten, welche irgendwo im Speicher liegen und leicht ausgetauscht werden können.
- Die `E`-Objekte bleiben immer unverändert irgendwo im Heap.

Verweis-Strukturen

Array



Einfach verkettete Liste



wobei
L = `Link<E>`

- `ArrayList` hat ein Array, darin liegen alle Verweise nacheinander im Speicher. Ein zu kleines Array muss komplett ausgetauscht werden.
- `SimpleList` speichert Verweise in privaten `Link`-Objekten, welche irgendwo im Speicher liegen und leicht ausgetauscht werden können.
- Die `E`-Objekte bleiben immer unverändert irgendwo im Heap.

Schnittstelle ListIterator<E>

Für Zugriffe innerhalb der Liste bietet bietet `LinkedList<E>` die Methode `ListIterator<E> listIterator()` welche einen Iterator liefert, der auf das erste Element zeigt.

Die Schnittstelle `ListIterator<E>` bietet u.a. folgende Methoden:

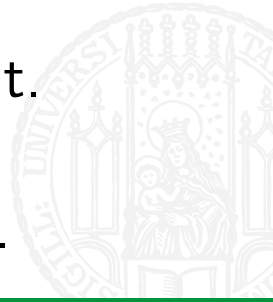
`boolean hasNext()`
: gibt an, ob am Positionszeiger noch ein Element vorhanden ist.

`E next()`
: liefert das Element am Positionszeiger zurück.
: Fehler, falls `hasNext()` = false.

`void add(E e)`
: fügt ein Element vor dem Positionszeiger ein.

`void remove()`
: entfernt das Letzte, von `next()` zurückgegebene, Element.

`void set(E e)`
: ersetzt das Letzte, von `next()` zurückgegebene, Element.



Anwendungsbeispiel

```
import java.util.*;

public class ListTest {
  public static void main(String[] args) {
    LinkedList<String> staff = new  LinkedList<String>();
    staff.addFirst("Tom");
    staff.addFirst("Romeo");
    staff.addFirst("Harry");
    staff.addFirst("Dick");

    ListIterator<String> iterator = staff.listIterator();
                                    // →DHRT
    String s = iterator.next();     // D→HRT      s=="Dick"
    iterator.next();                // DH→RT
    iterator.add("Juliet");         // DHJ→RT
    iterator.add("Nina");           // DHJN→RT
```

```
    ⋮

    iterator.add("Juliet");          // DHJ→RT
    iterator.add("Nina");            // DHJN→RT
    iterator.next();                 // DHJNR→T
    iterator.remove();               // DHJN→T
    iterator = staff.listIterator(); // neuer Iterator
    while (iterator.hasNext())
      System.out.println(iterator.next());
    }
}
```

Ausgabe:

⋮

```
    iterator.add("Juliet");          // DHJ→RT
    iterator.add("Nina");            // DHJN→RT
    iterator.next();                 // DHJNR→T
    iterator.remove();               // DHJN→T
    iterator = staff.listIterator(); // neuer Iterator
    while (iterator.hasNext())
      System.out.println(iterator.next());
    }
}
```

Ausgabe:

```
Dick
Harry
Juliet
Nina
Tom
```

Erklärung

- Die Methode `add` fügt ein neues Element unmittelbar vor dem Positionszeiger ein.
- Die Methoden `remove` und `set` sind nur zulässig, wenn direkt vorher `next` aufgerufen wurde; dann wird das von `next` zurückgegebene Element aus der Liste entfernt ("ausgespleißt") (bei `remove`) oder ersetzt (bei `set`).
- Es gibt auch noch die Methoden `hasPrevious, previous`, die den Positionszeiger nach vorne bewegen.
  Methoden `remove` und `set` dürfen auch nach einem Aufruf von `previous` aufgerufen werden und betreffen dann das von `previous` zurückgegebene Element.

Iteratoren-Schleifen

Mit einem Iterator kann man bequem über eine List laufen:

```
Iterator<E> iter = list.listIterator();
while(iter.hasNext()){
    E elem = iter.next();
    // Code zur Bearbeitung von elem
}
```

Bemerkung Alternativ kann auch die bereits behandelte vereinfachte `for`-Schleife verwendet werden:

```
for (E elem : list) { // Code zur Bearbeitung von elem }
```

Solche **For-Each-Schleifen** sind nicht nur für Listen, sondern für alle Typen erlaubt, welche das Interface `Iterable<E>` implementieren.

Iteratoren-Schleifen

Mit einem Iterator kann man bequem über eine List laufen:

Schleifen mit `for (E elem : list)`:

Vorteile Kurzer, leicht verständlicher Code; Index Fehler nicht möglich.

Nachteile Liste darf innerhalb der Schleife nicht verändert werden.

Bemerkung Alternativ kann auch die bereits behandelte vereinfachte `for`-Schleife verwendet werden:

```
for (E elem : list) { // Code zur Bearbeitung von elem }
```

Solche **For-Each-Schleifen** sind nicht nur für Listen, sondern für alle Typen erlaubt, welche das Interface `Iterable<E>` implementieren.

Fallstricke Iteratoren

- Während der Verwendung eines Iteratoren darf die Datenstruktur, über die iteriert wird, nicht verändert werden. Ausnahme `ConcurrentModificationException` wird sonst geworfen. Dabei ist es unerheblich, welcher Thread die Modifikation durchführt.
  Ausnahme: Veränderungen durch den Iterator selbst, z.B. mit `remove` oder `set`
- Manche Implementierung unterstützen das Interface nicht vollständig: `UnsupportedOperationException` wird dann bei Aufrufen von `remove` oder `set` geworfen.
- Aufruf von `remove` oder `set` ohne vorher `next` oder `previous` aufgerufen zu haben, liefert Ausnahme `IllegalStateException`.

Implementierung von LinkedList<E>

Die Klasse `LinkedList<E>` ist bereits implementiert.
Wir wollen sehen, wie das gemacht ist.
Braucht man nur die Methoden `addFirst`, `getFirst`, `next`, `hasNext`, so kann man *einfach* verkettete Listen verwenden:

```
import java.util.*;
class Link<E> {
    E data;
    Link<E> next;
}

public class LinkedList<E> {
    private Link<E> first;

    public LinkedList<E>() {
        first= null;
    }
```

```
Link<E> getFirstLink(){return first;}

public ListIterator<E> listIterator() {
    return new LinkedListIterator<E>(this);
}

public E getFirst() {
    if (first==null)
        throw new NoSuchElementException();
    else return first.data;
}

public void addFirst(E obj) {
    Link<E> newLink = new Link<E>();
    newLink.data = obj;
    newLink.next = first;
    first = newLink;
}
```

```
public E removeFirst() {
        if (first==null)
            throw new NoSuchElementException();
        E obj = first.data;
        first = first.next;
        return obj;
    }
}

class LinkedListIterator<E> implements ListIterator<E> {
    private Link<E> position;
    private LinkedList<E> list;


    public LinkedListIterator(LinkedList<E> l) {
        position = l.getFirstLink();
        list = l;
    }
```

```
public boolean hasNext() {
        return position != null;
    }

    /** Vorbedingung: hasNext() */
    public E next() {
        E obj = position.data;
        position = position.next;
        return obj;
    }
    /* Hier fehlen noch weitere Methoden
     * des Interfaces ListIterator<E>    */
}
```

Erklärung

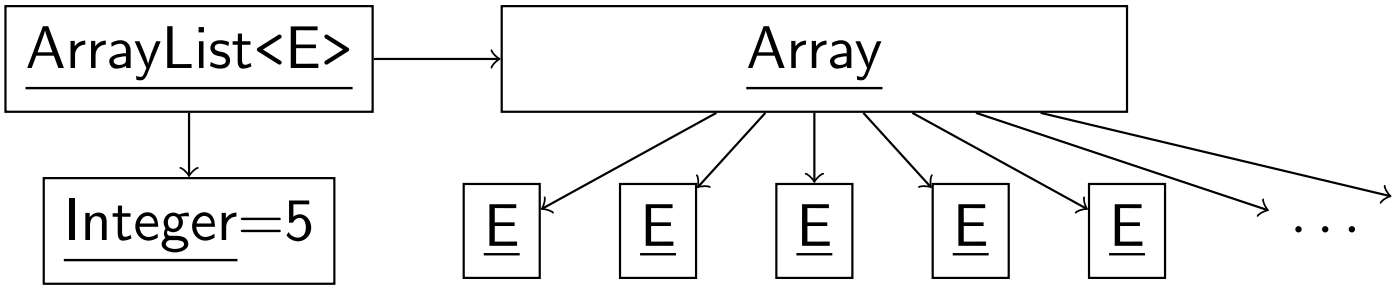
- Ein `Link<E>` besteht aus einem Datum und einem Verweis auf ein (das nächste) Link.
- Eine Liste ist einfach ein Verweis auf ein `Link<E>`.
- Die leere Liste wird durch `null` repräsentiert.
- Die Klassen `Link` und `LinkedListIterator` werden von außen nicht benötigt. Man kann sie daher auch als innere Klassen realisieren. Dann ist zudem der Zugriff auf `first` in `LinkedListIterator` einfacher.

Doppelt verkettete Listen

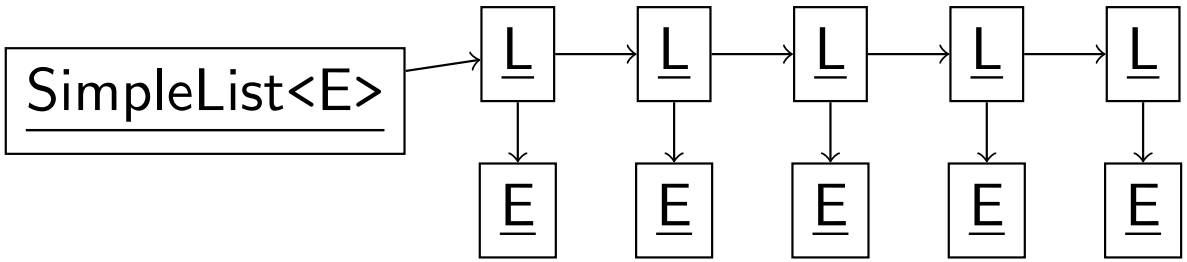
- Will man auch die Methoden `addLast`, `getLast`, `add`, `remove`, `hasPrevious`, `previous` implementieren, so muss man die Möglichkeit haben, in einer Liste rückwärts zu gehen.
- Dazu gibt man jedem Link auch noch einen Verweis auf das vorhergehende Link mit. Man muss natürlich all diese Verweise in den Methoden konsistent halten.
- Eine Liste besteht nun aus zwei Verweisen: einem auf das erste Link-Objekt und einen auf das Letzte. Die leere Liste wird durch zwei Nullreferenzen repräsentiert.
- Der Iterator wird nunmehr auch durch zwei Verweise (genannt `forward, backward`) implementiert.

Verweis-Strukturen

Array

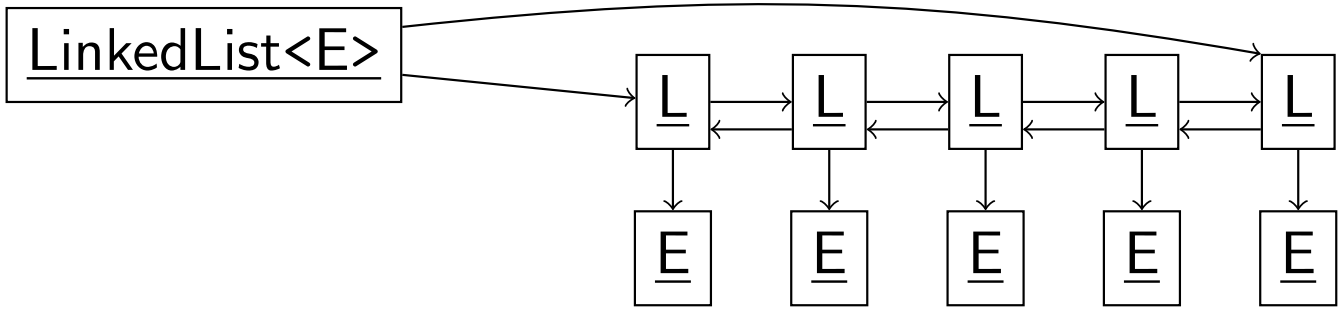


Einfach verkettete Liste



wobei
L = `Link<E>`

Doppelt verkettete Liste



Implementierung

```
import java.util.*;

class Link<E> {
    E data;
    Link<E> next;
    Link<E> prev;
}
public class LinkedList<E> {
    private Link<E> first;
    private Link<E> last;

    Link<E> getFirstLink(){return first;}

    public LinkedList() {
        first= null;
        last = null;
    }
```

```
public ListIterator<E> listIterator() {
    return new LinkedListIterator<E>(this);
}

public E getFirst() {
    if (first==null)
        throw new NoSuchElementException();
    else return first.data;
}

public E getLast() {
    if (first==null)
        throw new NoSuchElementException();
    else return last.data;
}
```

```
public void addFirst(E obj) {
    Link<E> newLink = new Link<E>();
    newLink.data = obj;
    newLink.next = first;
    newLink.prev = null;
    if (first == null) {
        first = newLink;
        last = newLink;
    } else {
        first.prev = newLink;
        first = newLink;
    }
}
```

```
    public void addLast(E obj) {
        Link<E> newLink = new Link<E>();
        newLink.data = obj;
        newLink.next = null;
        newLink.prev = last;
        if (first == null) {
            first = newLink;
            last = newLink;
        } else {
            last.next = newLink;
            last = newLink;
        }
    }
}
```

```
class LinkedListIterator<E> //extends LinkedList<E>
    implements ListIterator<E> {
    private Link<E> forward;
    private Link<E> backward;
    private LinkedList<E> list;
    private Link<E> lastReturned;

    public LinkedListIterator(LinkedList<E> l) {
        forward = l.getFirstLink;
        backward = null;
        list = l;
        lastReturned = null;
    }
```

```
public void add(E obj) {
    lastReturned = null;
    if (backward == null) {
        list.addFirst(obj);
        backward = list.getFirstLink();
    } else if (!hasNext()) {
        list.addLast(obj);
        backward = backward.next;
    } else {
        Link<E> newLink = new Link<E>();
        newLink.data = obj;
        newLink.next = forward;
        newLink.prev = backward;
        backward.next = newLink;
        forward.prev = newLink;
        backward = newLink;
    }
}
```

```
public boolean hasNext() {
    return forward != null;
}

public boolean hasPrevious() {
    return backward!= null;
}

public E next() {
    lastReturned = forward;
    backward = forward;
    forward = forward.next;
    return backward.data;
}
```

```
public E previous() {
    lastReturned = backward;
    forward = backward;
    backward = backward.prev;
    return forward.data;
}

public void set(E obj) {
    lastReturned.data = obj;
}

public int nextIndex()
{ throw new UnsupportedOperationException(); }

public int previousIndex()
{ throw new UnsupportedOperationException(); }
```

```
        public void remove() {
            if (lastReturned == null)
                throw new IllegalStateException();
            else {
                if (lastReturned.prev == null)
                    list.removeFirst();
                else if (lastReturned.next == null)
                    list.removeLast();
                else {
                    lastReturned.prev.next = lastReturned.next;
                    lastReturned.next.prev = lastReturned.prev;
                }
                if (lastReturned == backward)
                    backward = lastReturned.prev;
                else
                    forward = lastReturned.next;
                lastReturned = null;
            }
        }
    }
```

Bemerkungen

- Der Fall einer leeren Liste ist jeweils gesondert zu behandeln.
- Es wurden nicht alle erforderlichen Methoden implementiert; insbesondere nicht “remove”.
- Die Instanzvariable `lastReturned` verweist auf das Glied, das von `next`, bzw. `prev` als letztes zurückgegeben wurde, Es ist `null` falls der letzte Aufruf nicht `next` oder `previous` war. Man braucht sie zur Implementierung von `remove` und `set`.
- Es gibt in der Literatur zahlreiche Varianten.
- Listen können zur Realisierung von Stacks und Queues verwendet werden.

Vergleich Listen, Arrays

Sowohl Listen, als auch Arrays speichern Folgen von Daten.

- Listen sind besonders geeignet, falls Element an bestimmter Stelle eingefügt oder entfernt werden müssen.
- Listen haben keine feste Größe, sondern können beliebig erweitert werden.
- Arrays sind besonders geeignet, wenn der Zugriff auf Elemente über Positions**zahlen** (Indices) erfolgt. Bei Listen verursachen solche Operationen einen Aufwand, der proportional zum Index ist.

Fazit

- Verändert sich die Anzahl der Elemente oft, oder werden die Elemente immer der Reihe nach bearbeitet, dann `LinkedList`;
- Ändert sich die Anzahl der Elemente kaum (oder steht diese vorab fest), und werden die Elemente möglicherweise in willkürlicher Reihenfolge benutzt, dann `ArrayList`.

Übung

Was wird gedruckt?:

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Dick");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while(iterator.hasNext())
    System.out.println(iterator.next());
```