Verkettete Listen

- Verwendung von Listen in Java
- Das Prinzip des Iterators
- Implementierung von einfach verketteten Listen
- Implementierung von doppelt verketteten Listen

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 (wie in OCAML) 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.

Die Klasse `LinkedList`

Die Klasse `java.util.LinkedList` implementiert verkettete Listen.

Unter anderem gibt es die folgenden Methoden:

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

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

Die Schnittstelle `ListIterator`

Um auf Positionen innerhalb einer Liste zuzugreifen, gibt es die Schnittstelle `ListIterator`. Sie bietet u.a. folgende Methoden an:

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

`Object next()` liefert das Element beim Positionszeiger zurück. Fehlerhaft, falls nicht `hasNext()`.

In `LinkedList` gibt es noch die Methode

```
ListIterator listIterator()
```

Sie liefert zu einer Liste einen zugehörigen Iterator, der auf das erste Element verweist.

Anwendungsbeispiel

```
import java.util.*;

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

        ListIterator iterator = staff.listIterator(); // |DHRT
        iterator.next(); // D|HRT
        iterator.next(); // DH|RT
        iterator.add("Juliet"); // DHJ|RT
        iterator.add("Nina"); // DHJN|RT
        iterator.next(); // DHJNR|T
        iterator.remove(); // DHJN|T
```

Anwendungsbeispiel

```
        iterator = staff.listIterator();
        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 Methode `remove` ist nur zulässig, wenn vorher `next` aufgerufen wurde; dann wird das von `next` zurückgegebene Element aus der Liste entfernt (“ausgespleißt”).
- Es gibt auch noch die Methoden `hasPrevious, previous`, die den Positionszeiger nach vorne bewegen. Die Methode `remove` darf auch nach einem Aufruf von `previous` aufgerufen werden und entfernt dann das von `previous` zurückgegebene Element aus der Liste.

Implementierung von `LinkedList`

Die Klasse `LinkedList` 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 {
    public Object data;
    public Link next;
}

public class LinkedList {
    protected Link first;

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

Implementierung von `LinkedList`

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

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

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

Implementierung von `LinkedList`

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

class LinkedListIterator extends LinkedList
                         implements ListIterator {
    private Link position;
    private LinkedList list;


    public LinkedListIterator(LinkedList l) {
        position = l.first;
```

Implementierung von `LinkedList`

```
        list = l;
    }

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

    /**
       Vorbedingung: hasNext()
    */
    public Object next() {
        Object obj = position.data;
        position = position.next;
        return obj;
    }
    /* Hier fehlen noch Dummymethoden */
}
```

Erklärung

- Ein `Link` besteht aus einem Datum und einem Verweis auf ein (das nächste) Link.
- Eine Liste ist einfach ein Verweis auf ein `Link`.
- Die leere Liste wird durch `null` repräsentiert.
- `LinkedListIterator` ist Unterklasse von `LinkedList`, damit die protected Instanzvariablen sichtbar sind.

  Das braucht man insbesondere, wenn man auch noch `add` implementieren möchte.
- Eine Alternative besteht darin, `LinkedListIterator` zu einer *inneren Klasse* von `LinkedList` zu machen.

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 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.

Implementierung

```
import java.util.*;

class Link {
    public Object data;
    public Link next;
    public Link prev;
}

public class LinkedList {
    protected Link first;
    protected Link last;

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

Implementierung

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

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

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

Implementierung

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

Implementierung

```
class LinkedListIterator extends LinkedList
    implements ListIterator {
    private Link forward;
    private Link backward;
    private LinkedList list;
    private Link lastReturned;

    public LinkedListIterator(LinkedList l) {
        forward = l.first;
        backward = null;
        list = l;
        lastReturned = null;
    }

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

Implementierung

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

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

public void add(Object obj) {
    lastReturned = null;
    if (backward == null) {
        list.addFirst(obj);
        backward = list.first;
```

Implementierung

```
        } else if (!hasNext()) {
            list.addLast(obj);
            backward = backward.next;
        } else {
            Link newLink = new Link();
            newLink.data = obj;
            newLink.next = forward;
            newLink.prev = backward;
            backward.next = newLink;
            forward.prev = newLink;
            backward = newLink;
        }
    }
}
```

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`, siehe Doku.)
- Es gibt in der Literatur zahlreiche Varianten.
- Listen können (wie schon in Info I) 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.

Übung

Was wird gedruckt?:

```
LinkedList staff = new  LinkedList();
ListIterator 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());
```