XIII: 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 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<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()
```

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

Die Schnittstelle `ListIterator<E>`

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

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

`E next()` liefert das Element beim Positionszeiger zurück. Fehlerhaft, falls `hasNext()` = false.

In `LinkedList<E>` gibt es noch die Methode

```
ListIterator<E> 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<String> staff = new  LinkedList<String>();
        staff.addFirst("Tom");
        staff.addFirst("Romeo");
        staff.addFirst("Harry");
        staff.addFirst("Dick");

        ListIterator<String> iterator = staff.listIterator(); // |
        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<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;
    }
```

Implementierung von `LinkedList<E>`

```
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;
```

Implementierung von `LinkedList<E>`

```
        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;
```

Implementierung von `LinkedList<E>`

```
    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 Dummymethoden */
}
```

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 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<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;
    }
```

Implementierung

```
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;
}
```

Implementierung

```
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;
```

Implementierung

```
            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;
```

Implementierung

```
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>();
```

Implementierung

```
            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;
    }
```

Implementierung

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

public int nextIndex() {return 0;}
public E previous()
{
    lastReturned = backward;
    forward = backward;
    backward = backward.prev;
    return forward.data;
}

public int previousIndex() {return 0;}
```

Implementierung

```
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;
```

Implementierung

```
            lastReturned = null;
        }
    }

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

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 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());
```

XIV: Hashtabellen und Suchbäume

- Die Schnittstelle `Set<E>` der Mengen
- Die Schnittstelle `Map<K,V>` der endlichen Abbildungen
- Hashtabellen
- Binäre Suchbäume
- Balancierte Bäume (AVL-Bäume)
- B-Bäume

Mengen

Grundlegende Operationen auf einer Menge:

- Element hinzufügen
- Element entfernen
- Test auf Elementschaft
- Alle Elemente ausgeben (in beliebiger Ordnung)

In Java gibt es die Schnittstelle `Set<E>`, die diese und andere Methoden definiert.

Sie wird durch die Klassen `HashSet<E>` und `TreeSet<E>` implementiert.

`E` repräsentiert jeweils den Typ der Elemente.

Anwendungsbeispiel

```
import java.util.*;
import javax.swing.JOptionPane;

public class SetTest
{
    public static void main(String[] args) {
        Set<String> names = new HashSet<String>();

        boolean done = false;
        while (!done) {
            String input = JOptionPane.showInputDialog(
                                "Add Name, Cancel when done.");
            if (input == null)
                done = true;
            else {
                names.add(input);
                print(names);}}
```

Anwendungsbeispiel

```
        done = false;
        while (!done) {
            String input = JOptionPane.showInputDialog(
                                "Remove Name, Cancel when done.");
            if (input == null)
                done = true;
            else {
                names.remove(input);
                print(names);
            }
        }
        System.exit(0);
    }
```

Anwendungsbeispiel

```
    private static void print(Set<String> s) {
        Iterator<String> iter = s.iterator();
        System.out.print("{");
        while (iter.hasNext()) {
            System.out.print(iter.next());
            System.out.print(" ");
        }
        System.out.println("}");
    }
}
```

NB: Außer beim Konstruktor verwenden wir die *Schnittstelle* `Set<E>`, nicht die *Klasse* `HashSet<E>`. So müssen wir nur eine Zeile ändern um auf `TreeSet<E>` umzusteigen.

Abbildungen

Eine endliche Abbildung (engl. (finite) *map*) ist eine Zuordnung von *Schlüsseln* (*keys*) zu *Werten* (*values*).

- Zu einer Abbildung können Bindungen hinzugefügt oder entfernt werden.
- Zu jedem Schlüssel in der Definitionsmenge der Abbildung kann der zugehörge Wert ausgegeben werden.
- Zu jedem Schlüssel gibt es nur einen Wert. Fügt man eine neue Bindung mit demselben Schlüssel hinzu, so wird der alte Wert überschrieben.

Die Schnittstelle `Map<K,V>` stellt diese und andere Funktionen zur Verfügung. Sie wird implementiert u.a. durch die Klassen `HashMap<K,V>` und `TreeMap<K,V>`.

Der Typ `K` reprässentiert die Schlüssel; der Typ `V` repräsentiert die Werte.

Anwendungsbeispiel

```
import java.util.*;
import javax.swing.JOptionPane;
import java.awt.Color;

public class MapTest
{
    public static void main(String[] args) {
        Map<String,Color> favoriteColors = new HashMap<String,Colo
        favoriteColors.put("Juliet", Color.pink);
        favoriteColors.put("Romeo", Color.green);
        favoriteColors.put("Adam", Color.blue);
        favoriteColors.put("Eve", Color.pink);
        print(favoriteColors);
        favoriteColors.put("Adam", Color.yellow);
        favoriteColors.remove("Romeo");
        print(favoriteColors);
    }
```

Anwendungsbeispiel

```
    private static void print(Map<String,Color> m) {
        Set<String> keySet = m.keySet();
        Iterator<String> iter = keySet.iterator();
        while (iter.hasNext()) {
            String key = iter.next();
            String value = m.get(key);
            System.out.println(key + "->" + value);
        }
    }
}
```

Hashwerte

Die Klasse `Object` enthält eine Methode

```
int hashCode();
```

Sie liefert zu jedem Objekt einen Integer, den *Hashcode* oder *Hashwert*.

Die Spezifikation von `hashCode` besagt, dass zwei im Sinne von `equals` gleiche Objekte denselben Hashwert haben.

Es ist aber erlaubt, dass zwei verschiedene Objekte denselben Hashwert haben. Das ist kein Wunder, denn es gibt ja "nur" 2^{32} `int` Werte.

Allerdings sorgt eine gute Implementierung von `hashCode` dafür, dass die Hashwerte möglichst breit gestreut (*to hash* = fein hacken) werden. Bei "zufälliger" Wahl eines Objekts einer festen Klasse sollen alle Hashwerte "gleich wahrscheinlich" sein.

Implementierung von `Set` als Hashtabelle

Eine Möglichkeit, eine Menge zu implementieren, besteht darin, ein Array `s` einer bestimmten Größe `SIZE` vorzusehen und ein Element `x` im Fach `x.hashCode() % SIZE` abzulegen.

Das geht eine Weile gut, funktioniert aber nicht, wenn wir zwei Elemente ablegen möchten, deren Hashwerte gleich modulo `SIZE` sind.

In diesem Falle liegt eine *Kollision* vor.

Um Kollisionen zu begegnen kann man in jedem Fach eine verkettete Liste von Objekten vorsehen.

Für `get` und `put` muss man zunächst das Fach bestimmen und dann die dort befindliche Liste linear durchsuchen.

Sind Kollisionen selten, so bleiben diese Listen recht kurz und der Aufwand hält sich in Grenzen.

Genaue stochastische Analyse erfolgt in “Effiziente Algorithmen”.

Implementierung von `Map` als Hashtabelle

Praktisch dieselbe Datenstruktur kann man auch für Abbildungen verwenden:

Die Bindung $k \mapsto x$ wird im Fach `k.hashCode() % SIZE` abgelegt.

Dadurch ist sichergestellt, dass zu jedem Schlüssel nur ein Eintrag vorhanden ist.

Programmiersprachliche Realisierung

```
public class BinTree {

    public BinTree left;
    public BinTree right;
    public Object root;

    public BinTree(Object root, Tree left, Tree right) {
        this.left = left;
        this.root = root;
        this.right = right;
    }
```