Verkettete Datenstrukturen: Listen

Listen

Formal:

- Liste = endliche Folge von Elementen $[a_1, a_2, \ldots, a_n]$.
- Spezialfall: leere Liste [ ].
- Länge einer Liste = Anzahl der Elemente
(bei leerer Liste: 0).
- Anders als bei Mengen:
  - Listen sind geordnet (es kommt auf die Reihenfolge an),
  Wiederholungen von Elementen sind erlaubt.
- (Mengen können durch Listen implementiert werden).

Listen: wie repräsentieren?

Bisher: Array (oder in Java: Vector)

- Zugriff auf Elemente erfolgt über Index/Position.
Nachbarschaft zwischen Elementen ist implizit durch ihre Position definiert.
- Zugriff auf Element über Position: Zeitbedarf $O(1)$.
- Zugriff auf „nächstes“ Element: Zeitbedarf $O(1)$.
- Anhängen eines neuen Elements am Schluß: Zeitbedarf $O(1)$
(solange Platz vorhanden).
- Einfügen eines neuen Elements am Anfang oder zwischen zwei Elementen: Zeitbedarf $O(n)$
(nachfolgende Elemente müssen umkopiert werden).

Listen: wie repräsentieren?

Andere Möglichkeit der Repräsentation: verkettete Liste
(siehe Ende der letzten Vorlesungsstunde)

- Zugriff auf Elemente erfolgt sequentiell.
- Jedes Element besitzt eine Referenz auf seinen Nachfolger
(„weiß“, was sein Nachfolger ist)
$\rightsquigarrow$ einfach verkettete Liste
- oder sogar: Jedes Element besitzt Referenzen auf seinen Nachfolger und seinen Vorgänger
(„weiß“, was sein Nachfolger und sein Vorgänger ist)
$\rightsquigarrow$ doppelt verkettete Liste)

Listen: wie repräsentieren?

Verkettete Liste

- Zugriff auf „nächstes“ Element: Zeitbedarf $O(1)$.
- Zugriff auf Element über Position: Zeitbedarf $O(n)$.
  (drittes Element = Nachfolger des Nachfolgers des ersten Elements)
- Einfügen neuer Elemente: Zeitbedarf $O(1)$.

Einfach verkettete Listen

Grundlegender Baustein, aus dem eine verkettete Liste aufgebaut ist:

```
public class Element {
    Element nach; // Referenz auf Nachfolger
    int wert;
    // oder z.B.: Object wert
}
```

Elementsicht:

- `nach` ist eine Referenz auf das nächste Element der Liste.

Listensicht:

- `nach` ist eine Referenz auf den Rest der Liste.

Einfach verkettete Listen

Frage: was passiert am Listenende?

- Übliche Lösung: Beim letzten Element der Liste hat `nach` den Wert `null` (= „kein Objekt“).
- Das heißt: `null` repräsentiert die leere Liste.

Einfach verkettete Listen

Problem: Listenoperationen

- z.B.: vorne (oder hinten) an eine Liste ein Element anhängen
- In einer objektorientierten Sprache sollte der Aufruf so aussehen: `liste.haengeAn(34)`
- Aber: `null` ist kein Objekt.
  (Folglich kann man null keine Nachrichten schicken.)

Einfach verkettete Listen

Ausweg: ein weiterer Baustein: Listenkopf

```
// "Liste" und "Element" sind im gleichen Package
// (da wir in "Liste" auf die Variablen von "Element"
// zugreifen werden).
public class Liste {
    Element erstes;
    // entweder null (falls Liste leer)
    // oder Referenz auf erstes Element (sonst)

    // Methoden
    ...
}
```


Einfach verkettete Listen: Methoden

```
public boolean istLeer() {
    return erstes == null;
}
```


Einfach verkettete Listen: Methoden

```
public int ersteZahl() {
    if (istLeer()) {
        // hier sollte eigentlich eine Exception
        // ausgelöst werden (aber Exceptions wurden
        // noch nicht besprochen)
        return 0;
    } else {
        return erstes.wert;
    }
}
```


Einfach verkettete Listen: Methoden

```
public int laenge() {
    int i = 0;
    Element e = erstes;
    while (e != null) {
        e = e.nach;
        i++;
    }
    return i;
}
```

Einfach verkettete Listen: Methoden

```
public int anzahlVorkommen(int j) {
    int i = 0;
    Element e = erstes;
    while (e != null) {
        if (e.wert == j) {
            i++;
        }
        e = e.nach;
    }
    return i;
}
```


Einfach verkettete Listen: Methoden

```
public String toString() {
    Element e = erstes;
    String s = "[ ";
    while (e != null) {
        s += e.wert + " ";
        e = e.nach;
    }
    s += "]";
    return s;
}
```

ermöglicht folgenden Aufruf:

```
Liste liste = ...;
System.out.println("liste hat den Wert " + liste);
```


Einfach verkettete Listen: Methoden

```
public void haengeVorneAn(int j) {
    Element neu = new Element();
    neu.nach = erstes;
    neu.wert = j;
    erstes = neu;
}
```


Einfach verkettete Listen: Methoden

```
public void haengeHintenAn(int j) {
    Element neu = new Element();
    neu.wert = j;
    neu.nach = null; // eigentlich unnötig

    if (erstes == null) {
        erstes = neu;
    } else {
        Element e = erstes;
        while (e.nach != null) {
        // d.h.: solange e nicht auf das letzte Element zeigt
            e = e.nach;
        }
        e.nach = neu;
    }
}
```

Einfach verkettete Listen: Methoden

Die Schleife in `haengeHintenAn()` sieht anders aus als in `laenge()`. Warum?

Wir müssen hier die `nach`-Variable des letzten Listenelements verändern. Die Schleife

```
while (e != null) {...}
```

aus der Methode `laenge()` wird erst verlassen, wenn `e` das letzte Listenelement *überschritten* hat und nun den Wert `null` hat.

In diesem Moment haben wir aber keinen Zugriff mehr auf das letzte Listenelement.

Stattdessen müssen wir in der Schleife untersuchen, ob `e` gerade auf das letzte Listenelement zeigt.

Einfach verkettete Listen: Methoden

```
public void loescheVorn() {
    if (erstes == null) {
        // hier sollte eigentlich eine
        // Exception ausgelöst werden
    } else {
        erstes = erstes.nach;
    }
}
```


Einfach verkettete Listen: Methoden

```
public void loescheHinten() {
    if (erstes == null) {
        // hier sollte eigentlich eine Exception ausgelöst werden
    } else if (erstes.nach == null) {
        erstes = null;
    } else {
        Element e = erstes;
        while (e.nach.nach != null) {
        // d.h.: solange e nicht auf das VORletzte Element zeigt
            e = e.nach;
        }
        e.nach = null;
    }
}
```


Einfach verkettete Listen: Methoden

Diese Schleife ist noch komplizierter als in `haengeHintenAn()`, da wir hier die `nach`-Variable des *vorletzten* Listenelementes verändern müssen.

Aus diesem Grund müssen wir auch den Fall einer einelementigen Liste separat behandeln; in dieser gibt es nämlich kein vorletztes Element.

Einfach verkettete Listen: Methoden

```
public void loescheEinmal(int j) {
    if (erstes == null) {
        // nichts zu tun
    } else if (erstes.wert == j) {
        erstes = erstes.nach;
    } else {
        Element e = erstes;
        while (e.nach != null && e.nach.wert != j) {
            e = e.nach;
        }
        if (e.nach != null) {
            e.nach = e.nach.nach;
        }
    }
}
```


Einfach verkettete Listen: Methoden

```
public Liste teillisteAb(int j) {
    Liste l = new Liste();
    Element e = erstes;
    while (e != null && e.wert != j) {
        e = e.nach;
    }
    l.erstes = e;
    return l;
}
```


Einfach verkettete Listen: Methoden

Vorsicht: die Originalliste und die zurückgegebene Teilliste verwenden *dieselben* Elemente („sharing“).

Wenn aus einer der Listen etwas gelöscht wird, führt das zu *sehr* merkwürdigen Effekten.
(Insbesondere macht es einen Unterschied, ob das erste „gesharte“ Element gelöscht wird, oder irgend ein anderes Element. Warum?)

Ein möglicher Ausweg: explizites Kopieren der Teilliste.

Einfach verkettete Listen: Alternativen

Geht das alles auch einfacher?

Kann man sich zum Beispiel die Sonderfälle in `haengeHintenAn()`, `loescheHinten()` und `loescheEinmal()` sparen?

Einfach verkettete Listen: Alternativen

Idee: Wenn man die Variable in der Klasse `Liste` nicht `erstes` nennt, sondern `nach`, dann kann man die Klasse `Element` als Erweiterung der Klasse `Liste` auffassen:

```
public class Liste {
    Element nach;
    ...
}

public class Element extends Liste {
    int wert;
    // oder z.B.: Object wert
}
```


Einfach verkettete Listen: Alternativen

Die Laufvariable `e` in `haengeHintenAn()` kann dann als `Liste` anstatt als `Element` deklariert werden.

Vorteil: `e` kann nun über den Listenkopf *und* über die Elemente laufen:

```
public void haengeHintenAn(int j) {
    Element neu = new Element();
    neu.wert = j;
    neu.nach = null; // eigentlich unnötig

    Liste e = this;
    while (e.nach != null) {
    // d.h.: solange e nicht auf das letzte Element zeigt
            e = e.nach;
    }
    e.nach = neu;
}
```


Einfach verkettete Listen: Alternativen

Könnte man die Klassenhierarchie auch anders wählen?

Könnte man die Klassenhierarchie auch sinnvoller wählen?

Doppelt verkettete Listen

Geht das alles noch einfacher?

Teilweise ja, wenn man doppeltverkettete Listen verwendet:

```
public class Element {
    Element vor;  // Referenz auf Vorgänger
    Element nach; // Referenz auf Nachfolger
    int wert;
}
```

Doppelt verkettete Listen

Zusätzlicher Trick:

Die `vor`-Variable des ersten Elements und die `nach`-Variable des letzten Elements haben nicht den Wert `null`, sondern verweisen auf ein gemeinsames Dummy-Element.
Die Liste wird sozusagen zu einem Kreis geschlossen.
Der Listenkopf enthält eine Referenz auf das Dummy-Element.

Die Abfrage `nach == null` muß dann natürlich ersetzt werden durch die Abfrage, ob `nach` gleich dem Dummy-Element ist.

Doppelt verkettete Listen

Nachteile:

Elemente brauchen (etwas) mehr Speicherplatz.

Teillisten können nicht mehr mit Sharing erzeugt werden.
(Die Elemente *müssen* also jetzt kopiert werden.)

Vorteile:

Man kann jetzt auch das Element löschen, auf das man gerade zeigt.

Man kann in konstanter Zeit auf das letzte Element einer Liste zugreifen (statt in $O(n)$, wie bei einfach verketteten Listen).