2. Schnittstellen

Lernziele:

- Das Konzept Schnittstelle in der objektorientierten Programmierung kennen und verstehen,
- implementierte Schnittstellen nutzen können,
- Schnittstellen implementieren können und
- Schnittstellen definieren können.
- Insgesamt: Das Konzept Schnittstelle bei der Softwareentwicklung angemessen einsetzen können.

Charakterisierung von Schnittstellen

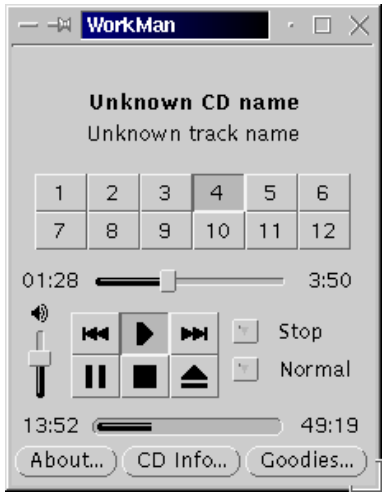
Eine *Schnittstelle (Interface)* ist eine Spezifikation eines Typs in Form eines Typnamens und einer Menge von abstrakten Methoden.

- Wir können Schnittstellen anschaulich zunächst als rein abstrakte Klassen auffassen, d.h. alle Methoden dieser Klasse sind abstrakt.

Was bringt uns das?

- Vollständige Trennung von Spezifikation und Implementierung
- Gänzlich verschiedene Implementierungen der gleichen Spezifikation
- Austauschbarkeit

Beispiel: CD-Spieler als Exemplare eines abstrakten Konzepts



Spezifikation

- ☞ einheitliche Signatur
  für die Operationen play, pause, stop, eject, etc.
- ☞ vereinbartes Verhalten

Datenabstraktion

- ☞ Trennung von Verhalten und Implementierung
- ☞ Verstecken der Implementierung

Schnittstelle: Syntax

Definition einer Schnittstelle:

```
[public] interface InterfaceName {
   Konstantendeklarationen
   Methodenköpfe
}
```

Beispiel für eine öffentliche Schnittstelle:

```
public interface Playable {
   void play();
   void pause();
   void stop();
}
```

Beispiel: Funktionen

Wir möchten mathematische Funktionen $f : D \longrightarrow \mathbb{R}$ mit $D \subseteq \mathbb{R}$ als abstraktes Konzept mit den folgenden Operationen modellieren:

- Test, ob ein $x \in \mathbb{R}$ zum Definitionsbereich von f gehört: Gilt $x \in D$?
- Berechnung von $f(x)$ für $x \in D$.
- Ermittlung einer textuellen Repräsentation für die Funktion $f(x)$.

```
public interface Funktion {
   boolean istImDefBereich(double x);
   double  wert(double x);
   String  gibBeschreibung();
}
```

Beispiel: Wörterbuch (Dictionary)

Ein *Wörterbuch (Dictionary)* bezeichnet in der Informatik einen abstrakten Datentyp, der eine Menge von Schlüssel-Wert-Paaren verwaltet, so dass jeder Schlüsselwert höchstens einmal in der Menge auftritt.

Beispiel: Schlüssel: KFZ-Kennzeichen, Wert: Informationen über das KFZ

Klassische Operationen für Wörterbücher:

- **put:** Füge ein Schlüssel-Wert-Paar in das Wörterbuch ein. Sollte der Schlüssel im Wörterbuch schon existieren, wird der mit dem Schlüssel verknüpfte Wert ersetzt.
- **get:** Ermittle den Wert zu einem Schlüssel.
- **remove:** Lösche einen Schlüssel und den verknüpften Wert aus dem Wörterbuch.

Wörterbuch als Java-Interface

```
public interface Dictionary {
   void put(String key, String info);
   String get(String key);
   void remove(String key);
}
```

Bemerkungen:

- In der Programmierung werden Wörterbücher häufig als `Map` bezeichnet.
- Das Java SDK enthält Schnittstellendefinitionen und Implementierungen für Maps. Mehr hierzu in Kapitel 4.
- Wie man Wörterbücher effizient implementiert, ist eine zentrale Frage im zweiten Teil der Vorlesung.

Eigenschaften von Schnittstellendefinitionen

- Im Kopf wird statt `class` das Schlüsselwort `interface` verwendet.
- Alle Methoden in einem Interface sind abstrakt:
  - es sind keine Methodenrümpfe zugelassen,
  - das Schlüsselwort `abstract` wird nicht angegeben.
- Interfaces enthalten keine Konstruktoren.
- Alle Methoden sind implizit `public`:
  - die Sichtbarkeit muss nicht angegeben werden.
- Als Datenfelder können nur öffentliche Konstanten deklariert werden (obsolet):
  - die Festlegung als Konstante durch `public`, `static` und `final` muss nicht erfolgen.

Sichtbarkeit für Interfaces: nur `public` oder default ist möglich

Schnittstellen und Datentypen (1)

- Wenn die Schnittstellendefinition in einer Datei mit der Endung `.java` liegt, können wir die Schnittstellendefinition wie eine Klasse übersetzen.
- Analog zu einer Klasse entsteht durch die Kompilierung auch eine `.class`-Datei.
- Ebenso wie bei Klassen wird durch die Schnittstelle auch ein Datentyp erzeugt.
- Da Schnittstellen aber abstrakt sind, können zu Schnittstellen keine Objekte instanziiert werden, d.h.

  `new` *`Schnittstelle`*`(...);`

  ist nicht möglich (Ausnahme: Anonyme Klassen).
- Analog zu abstrakten Klassen kann aber auch eine Schnittstelle als Datentyp für Variablen oder Parameter verwendet werden.

Implementierung von Interfaces

- Klassen können Interfaces implementieren.
- Hierzu muss eine Klasse folgendes leisten:
  1. Für jede Methode des Interface muss die Klasse eine Methode mit passender Signatur anbieten.
  2. Im Kopf der Klasse müssen wir explizit angegeben, dass unsere Klasse das Interface implementiert. Hierzu nutzen wir das Schlüsselwort `implements`.

Syntax:

```
[public] class Klasse [extends Oberklasse] implements Schnittstelle {
   ...
}
```

Beispiel: ☞ Implementierung der Schnittstelle `Funktion`

Implementierung mehrerer Interfaces

- Wenn wir `extends` zusammen mit `implements` nutzen, muss `extends` vor `implements` stehen.
- Es ist möglich, dass eine Klasse mehr als eine Schnittstelle implementiert.
- In diesem Fall geben wir alle implementierten Schnittstellen als Kommaliste hinter `implements` an.

```
public class Klasse extends Oberklasse
                    implements Schnittstelle1, Schnittstelle2, ...
{
   ...
}
```

Schnittstellen und Datentypen (2)

- Ein Interface definiert genau wie eine Klasse einen Datentyp.
- Klassen, die ein Interface implementieren, definieren Subtypen des Interface-Typs.
- Somit können Variablen von einem Interface-Typ deklariert werden, obwohl keine Objekte dieses Typs existieren können, sondern nur Objekte der Subtypen.
- Wenn die Klasse `Parabel` die Schnittstelle `Funktion` implementiert:

```
Funktion f = new Parabel( 1.0, 0.0, 0.0 ); // f(x) = x^2
```

- Auch möglich:

```
Funktion[] f = new Funktion[10];
```

- Damit haben wir wieder Variablen- und Methodenpolymorphie.

Beispiel: ☞ Funktionen

Aspekte von Schnittstellen: Implementierte Schnittstelle nutzen

Im Java SDK ist die Schnittstelle `Comparable` definiert (ungefähr so):

```
public interface Comparable {
    int compareTo(Object o);
}
```

Vereinbartes Verhalten: Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

`Comparable` wird implementiert von u.a.: `Integer`, `String`, etc.

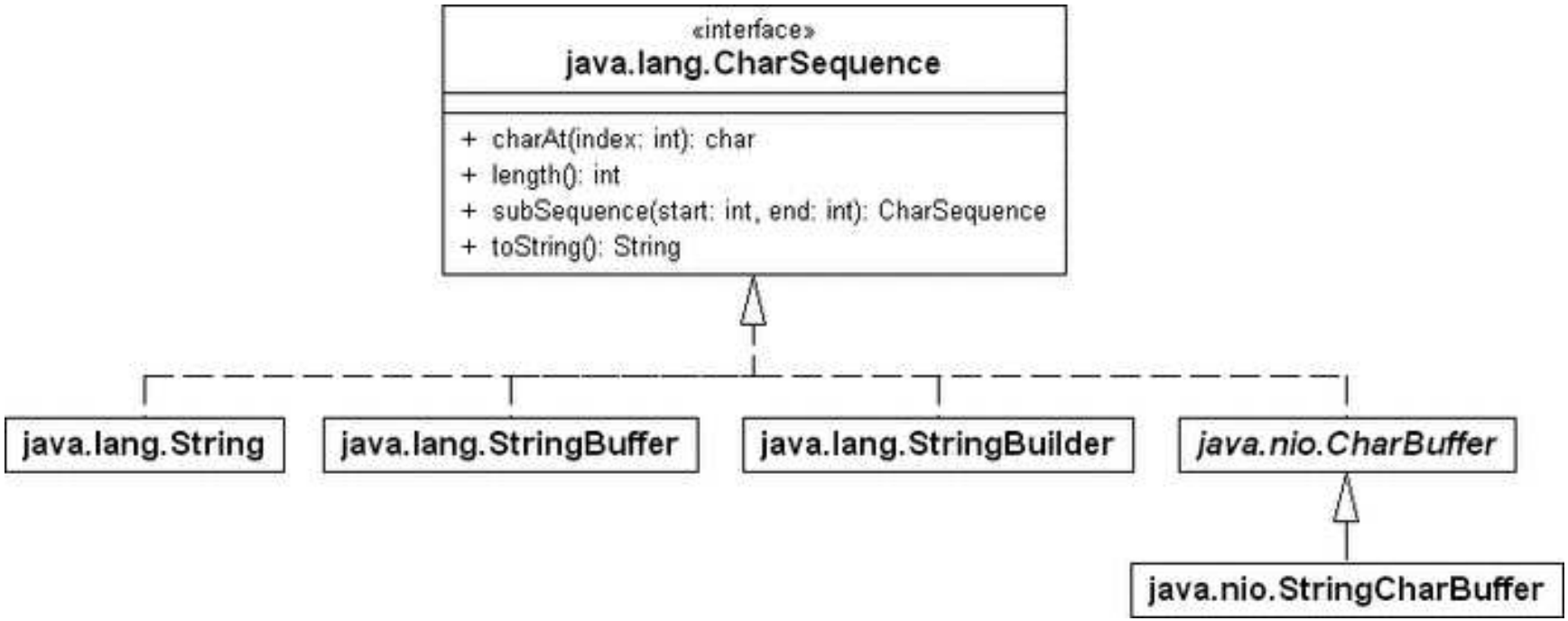
Damit können wir eine generische Minimumsbildung implementieren, die für alle Datentypen funktioniert, die `Comparable` implementieren:

```
public class GenerischesMin {
    public static Comparable min(Comparable[] c) {
        Comparable minval = c[0];

        for (int i=1 ; i<c.length ; i++) {
            if ( c[i].compareTo(minval) < 0 ) {
                minval = c[i];
            }
        }
        return minval;
    }
}
```

Beispiel zur Nutzung: ☞ siehe Homepage

In den Übungen: Implementierungen von CharSequence nutzen



Siehe: C. Ullenboom, Java ist auch eine Insel

Aspekte von Schnittstellen: Schnittstelle implementieren

- Click auf einen Button in GUI
- asynchron
- Wie im Programm Reaktion definieren?

Typischer Ansatz:

- Definierte Schnittstelle für die Benachrichtigung von Button-Clicks
- Objekte, die über Button-Clicks informiert werden möchten, implementieren die Schnittstelle und registrieren sich beim Button.
- Wird der Button geklickt, informiert er die registrierten Objekte über die Methode der Schnittstelle.

Schnittstelle `java.awt.event.ActionListener`:

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

- Klasse `javax.swing.JButton` implementiert Buttons
- In `JButton` Methode `addActionListener()` um Objekt zu registrieren, das über Clicks informiert werden möchte.
- Implementierung von `ActionListener` für das entsprechende Objekt.

Beispiel: ☞ siehe Homepage

- Entspricht dem Observer Pattern (mehr zu Patterns in Software-Engineering I, 3. Semester)
- Paradigma: Inversion of Control

Aspekte von Schnittstellen: Schnittstellen definieren

- Siehe Beispiel zu Funktionen.
- Weitere Nutzungen: Verkettung von Funktionen, generische Wertetabelle

Beispiele: ☞ siehe Homepage

Interfaces als Spezifikation

- Interfaces erlauben es, die Funktionalität (Schnittstelle) vollständig von der Implementierung zu trennen.
- Gute Beispiele hierfür findet man im Java SDK.
- Im Paket `java.util` findet man beispielsweise das Interface `List` sowie die Klassen `ArrayList` und `LinkedList`.
- Das Interface `List` definiert die volle Funktionalität einer Liste, ohne eine Implementierung festzulegen.
- Die Klassen `ArrayList` und `LinkedList` stellen verschiedene Implementierungen dieser Schnittstelle dar.
- Die beiden Implementierungen unterscheiden sich erheblich in der Effizienz einiger Methoden.
- Welche Implementierung für eine gegebene Anwendung besser ist, ist im voraus oft schwer festzulegen.

- Das Interface `List` macht aber solch eine Listenanwendung flexibel.
- Wenn wir stets `List` für Datentypen bei den entsprechenden Variablen und Parametern verwenden, funktioniert die Anwendung unabhängig von der aktuell gewählten Implementierung für Listen.
- Lediglich bei der Erzeugung einer Liste müssen wir eine Implementierung festlegen. Beispiel:

  ```
  private List meineListe = new ArrayList();
  ```

- So könnten wir in der Anwendung eine `LinkedList` verwenden, indem wir einfach an dieser einen Stelle `ArrayList` durch `LinkedList` ersetzen.

Wann abstrakte Klassen, wann Schnittstellen?

- Abstrakte Klassen bieten die Wiederverwendung von Quelltext, erlauben aber nur einfache Vererbung.

  ☞ Abstrakte Klassen bei Generalisierung von Implementierungen für verwandte Klassen
- Schnittstellen bieten keine Wiederverwendung von Quelltext, erlauben aber die generische Nutzung unabhängig von der Vererbungshierarchie

  ☞ Schnittstellen für die vollständige Trennung von Funktionalität und Implementierung

Typumwandlungen (Upcasting)

Implizite Upcasts finden für Schnittstellen in folgenden Fällen statt:

- Von Klasse K nach Schnittstelle S, wenn die Klasse K die Schnittstelle S implementiert.

```
interface S { ... }
class K implements S { ... }

S s = new K();
```

- Von `null` zu jeder Klasse, Schnittstelle oder Feld.

```
class K { ... }
interface S { ... }

K k = null;
S s = null;
```

```
K[] kfeld = null;
S[] sfeld = null;
```

- Von Schnittstelle V nach Schnittstelle S, wenn V Unterschnittstelle von S ist.

```
interface S { ... }
interface V extends S { ... }
class K implements V { ... }

V v = new K();
S s = v;
```

- Von Schnittstelle oder Feld zu `Object`.
- Von Feld R[ ] mit R als Komponententyp nach Feld T[ ] mit Typ T als Komponententyp, wenn es einen Upcast von R nach T gibt.

Typumwandlungen (Downcasting)

- Downcasting ist nur explizit möglich.
- Wird zur Laufzeit auf Korrektheit geprüft.
- `ClassCastException` falls Downcasting fehlerhaft
- Die Verwendung von `instanceof` ist auch mit Schnittstellentypen möglich.

Schnittstellen und Vererbung

- Schnittstellen besitzen analog zu Klassen die Möglichkeit, mit dem Schlüsselwort `extends` eine schon vorhandene Schnittstelle zu erweitern.
- ```
  public interface NachrichtenQuelle {
      public int SPORT    = 0;
      public int POLITIK  = 1;
      public int KULTUR   = 2;
      public int ANZEIGEN = 3;
      public int GESAMT   = 4;

      public void anmelden(NachrichtenEmpfaenger empf, int typ);
      public void sendeNachricht(String nachricht);
  }
  ```
  ```

- ```
  public interface Vermittler extends NachrichtenQuelle {
      public void empfangeNachricht(String nachricht);
  }
  ```
- Wirkungsweise: Die Definition der Unterschnittstelle erbt die Definitionen der Oberschnittstelle.

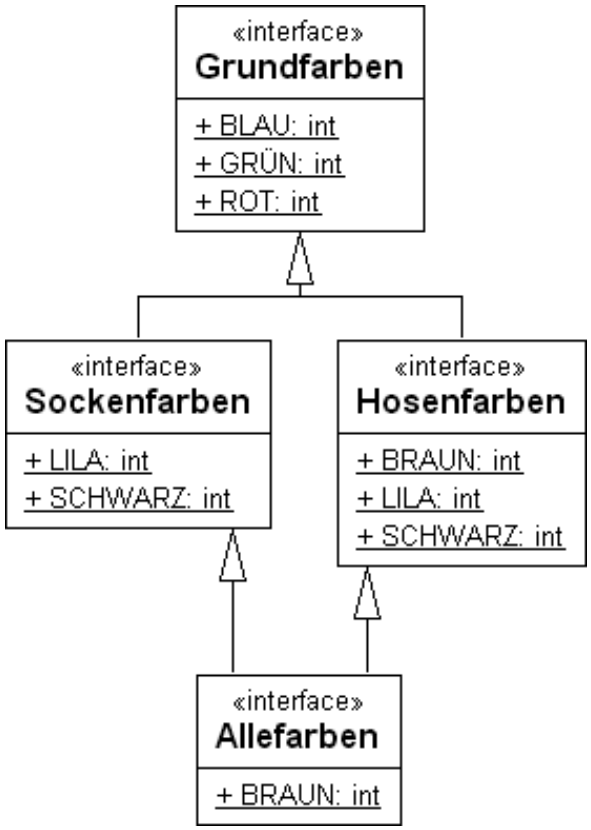
  Im Beispiel: Die Schnittstelle `Vermittler` umfasst alle Definitionen von `NachrichtenQuelle` plus die Methode `empfangeNachricht()`.
  ```

Mehrfachvererbung bei Schnittstellen

- Im Gegensatz zur Einfachvererbung von Klassen ist in Java bei Schnittstellen eine Mehrfachvererbung erlaubt.
- Damit kann eine Schnittstelle gleichzeitig mehrere andere Schnittstellen erweitern.
- ```
  public interface Schnittstelle
                   extends Oberschnittstelle1, Oberschnittstelle2, ... {
      ...
  }
  ```
- Die Mehrfachvererbung bei Schnittstellen hat allerdings keine große Bedeutung. Die Möglichkeit, dass eine Klasse mehrere Schnittstellen implementiert, ist von erheblich größerer Bedeutung.
  ```

Mehrdeutigkeiten bei Mehrfachvererbung

```
public interface Grundfarben {
   int ROT = 1; int GRUEN = 2;
   int BLAU = 3;
}
public interface Sockenfarben
   extends Grundfarben {
   int SCHWARZ = 10; int LILA = 11;
}
public interface Hosenfarben
   extends Grundfarben {
   int LILA = 11; int SCHWARZ = 20;
   int BRAUN = 21;
}
public interface Allefarben
   extends Sockenfarben, Hosenfarben {
   int BRAUN = 30;
}
```



- Konstanten werden vererbt: `Allefarben.ROT` ist definiert
- Konstanten dürfen überschrieben werden: `Allefarben.BRAUN` überschreibt `Hosenfarben.BRAUN`
- Bei der Nutzung der Konstanten dürfen keine Mehrdeutigkeiten auftreten: `Allefarben.SCHWARZ` ist nicht definiert
- Auch keine potentiellen Mehrdeutigkeiten: `Allefarben.LILA` ist nicht definiert

Markierungsschnittstelle (Marker Interface)

- Eine *Markierungsschnittstelle* dient dazu, die Nutzung bestimmter Methode zu erlauben.
- Solch eine Markierungsschnittstelle enthält in Ihrer Definition keine Methoden, da die Methoden schon implementiert sind.
- Implementierende Klassen geben über eine Markierungsschnittstelle an, dass sie eine bestimmte Funktionalität unterstützen.
- Beispiel: `java.lang.Cloneable`

Cloneable als Beispiel für ein Marker Interface

- Die Klasse `Object` definiert die Methoden `clone()`, die eine identische Kopie eines Objektes anlegt.

  > This method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment;
  > the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.

- Da alle Klassen implizit von `Object` abgeleitet sind, verfügt somit jedes Objekt über eine Methode `clone()`.
- Die Nutzung ist aber nicht ohne weiteres möglich, denn:

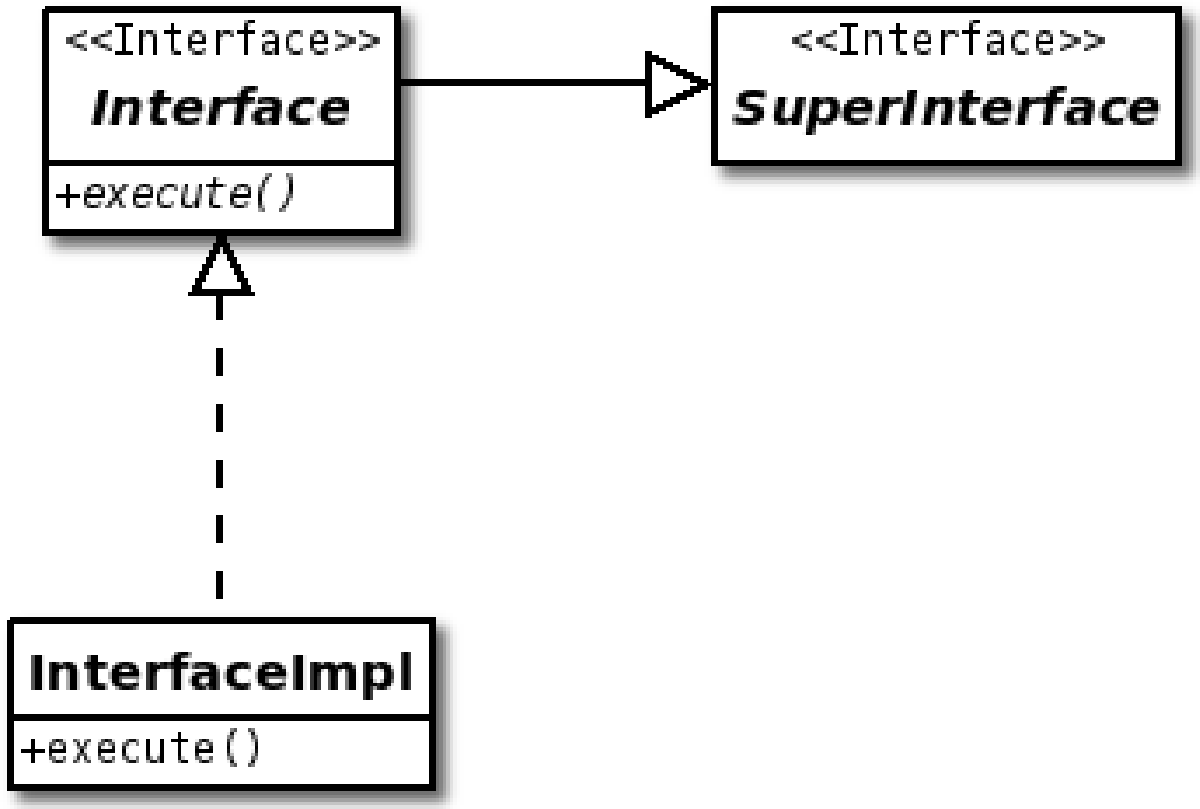
  > If the class of this object does not implement the interface `Cloneable`, then a `CloneNotSupportedException` is thrown.

- Eine Klasse, die das Kopieren von Objekten erlauben möchte, macht dies durch `implements Clonable` deutlich.

  ```
  class KopierenMoeglich implements Clonable { ... }
  ```

- Eine Implementierung von `clone()` ist aber nicht notwendig.
- `clone()` kann aber überschrieben werden, z.B. um "tiefes Kopieren" zu realisieren. Die Kopie sollte dabei zunächst durch `super.clone()` erzeugt werden.

Interfaces umd UML



3. Exceptions

Hintergrund: Programmieren auf der Basis von Verträgen

Kundenklasse
Methodenaufruf • Verpflichtung zur Einhaltung der Vorbedingung

Lieferantenklasse
Methodendefinition • Vorbedingung • Invarianten • Verpflichtung zur Einhaltung der Nachbedingung

☞ Ausnahme/Exception: Lieferantenklasse kann ihren Vertrag nicht erfüllen.

Beispiel:

```
public static void main(String[] args)
{
   int a = Integer.parseInt(args[0]);
   int b = Integer.parseInt(args[1]);

   System.out.println(a + "/" + b + "=" + (a/b));
}
```

- `java ExceptionTest`
  ☞ ArrayIndexOutOfBoundsException
- `java ExceptionTest 4 0`
  ☞ ArithmeticException
- `java Exception 4 a`
  ☞ NumberFormatException

Exception

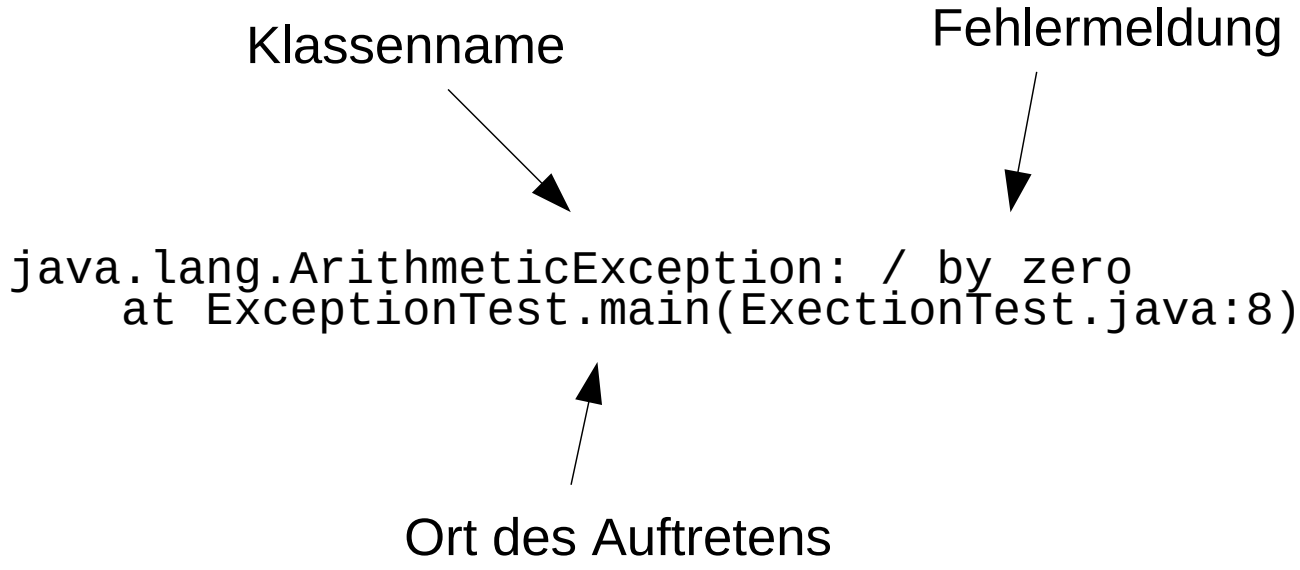
- Eine *Exception* ist ein Objekt, das Informationen über einen Programmfehler enthält.
- Eine Exception wird ausgelöst, um zu signalisieren, dass ein Fehler aufgetreten ist.

Vorteile von Exceptions:

- Für einen Klienten/Kunden ist es (fast) unmöglich, eine aufgetretene Exception zu ignorieren und einfach weiterzuarbeiten.
- Wenn der Kunde die Exception nicht behandelt, dann wird die laufende Anwendung beendet.

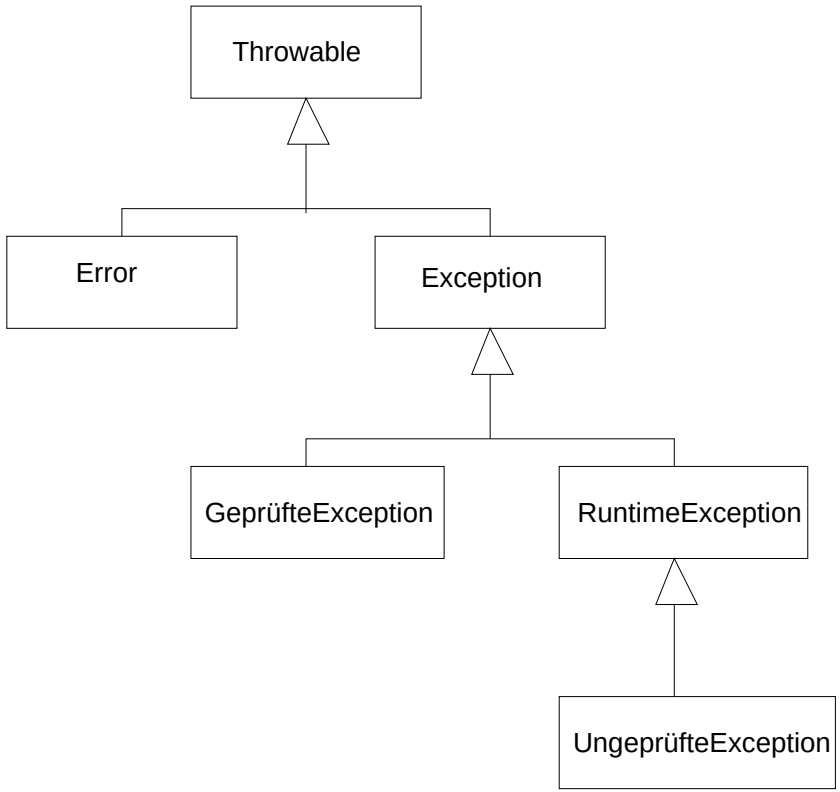
Die Klasse java.lang.Exception

- Exceptions sind Instanzen der Klasse `java.lang.Exception`
- bzw. einer Unterklasse von `java.lang.Exception`.
- `java.lang.Exception` definiert u.a. die folgenden Methoden:
  - `getMessage()`: Rückgabe der Fehlermeldung
  - `printStackTrace()`: Erzeugt die Ausgabe, die wir auf der Konsole sehen.



Exception-Klassen

- Eine Exception ist immer eine Instanz aus einer speziellen Vererbungshierarchie.
- Wir können selbst neue Exception-Typen definieren, indem wir Subklassen dieser Hierarchie erzeugen.
- Subklassen von `java.lang.Error` sind für Fehler des Laufzeitsystems vorgesehen.
- Für neue Exceptions erzeugen wir Subklassen von `java.lang.Exception`.
- Das Paket `java.lang` stellt bereits eine Reihe von vordefinierten Subklassen bereit.



Arten von Exceptions

- Java unterscheidet zwischen *geprüften* und *ungeprüften Exceptions*.
- `RuntimeException` und alle Subklassen der Klasse `RuntimeException` definieren ungeprüfte Exceptions.
- Alle anderen Subklassen von `Exception` definieren geprüfte Exceptions

Ungeprüfte Exceptions

- Dies sind Exceptions, bei deren Verwendung der Compiler keine zusätzlichen Überprüfungen vornimmt.
- Genauer: Der Compiler prüft nicht, ob sich der Klient um diese Exceptions kümmert.
- Ungeprüfte Exceptions sind für Fälle gedacht, die normalerweise nicht auftreten sollten, z.B. ausgelöst durch Programmfehler.

Beispiele:

```
ArrayIndexOutOfBoundsException
NullPointerException
NumberFormatException
```

Geprüfte Exceptions

- Geprüfte Exceptions sind Exception-Typen, bei deren Verwendung der Compiler zusätzliche Überprüfungen durchführt und einfordert.
- Ein Klient muss sich um geprüfte Exceptions kümmern.
- Geprüfte Exceptions sind für die Fälle gedacht, bei denen ein Klient damit rechnen sollte, dass eine Operation fehlschlagen kann. Hier sollte der Klient gezwungen werden, sich um den Fehler zu kümmern.

Beispiele:

`java.io.IOException`, `java.io.FileNotFoundException`

Faustregeln

Verwende ungeprüfte Exceptions ...

- ☞ ... in Situationen, die zu einem Programmabbruch führen sollten.
- ☞ ... bei Fehlern, die vermeidbar gewesen wären.

Verwende geprüfte Exceptions ...

- ☞ ... bei Fehlern, die vom Klienten behandelt werden können.
- ☞ ... für Fälle, die außerhalb des Einflusses des Programmierers liegen.

Auswirkungen einer Exception

- Wenn eine Exception ausgelöst wird, wird die Ausführung der auslösenden Methode sofort beendet.
- In diesem Fall muss kein Wert von der Methode zurückgeliefert werden, auch wenn die Methode nicht als `void` deklariert wurde.
- Die Anweisung, die die auslösende Methode aufgerufen hat, konnte nicht korrekt abgearbeitet werden.
- Es besteht die Möglichkeit, solche Exceptions zu *fangen*.

Kontrollfluss bei Exceptions

```
Klasse obj = new Klasse();
try {
   . . .
   obj.method();
   . . .
}
catch ( MeineException e ) {
   /* spez. F.-Behandlung */
}
catch ( Exception e ) {
   /* allg. F.-Behandlung */
}
finally {
   /* Aufräumarbeiten */
}
...
```

```
public class Klasse
{
   public void method()
          throws MeineException
   {
      . . .
      /* Fehlererkennung */
      throw new MeineException("...");
      . . .
   }
}
```

Die Konstrukte try, catch, finally, throw und throws

`try`
: Definiert einen Block, innerhalb dessen Exceptions auftreten können.

`catch`
: Definiert einen Block, der die Fehlerbehandlung für eine Ausnahme durchführt.

`finally`
: Definiert einen Block, der Aufräumarbeiten durchführt.

`throw`
: Erzeugt eine Ausnahme.

`throws`
: Deklariert eine Ausnahme für eine Methode.

Geprüfte Exceptions: throws

- Der Compiler fordert, dass eine Methode, die eine geprüfte Exception auslösen (oder weiterreichen) kann, dies im Methodenkopf angibt.
- Syntaxbeispiel:

```
public void speichereInDatei(String dateiname)
   throws IOException
```

- Die Angabe von ungeprüften Exception-Typen ist erlaubt aber nicht erforderlich.
- Hinter `throws` können durch Komma getrennt auch mehrere Exception-Typen angegeben werden.

```
public void kopiereDatei(String quelldatei, String zieldatei)
   throws FileNotFoundException, IOException
```

Exception-Handler

- Ein *Exception-Handler* ist ein Programmabschnitt, der Anweisungen schützt, in denen eine Exception auftreten könnte.
- Der Exception-Handler definiert Anweisungen zur Meldung oder zum Wiederaufsetzen nach einer aufgetretenen Exception.

Syntax:

```
try {
   geschützte Anweisungen
}
catch (ExceptionTyp e) {
   Anweisungen für die Behandlung
}
finally {
   Anweisungen für alle Fälle
}
```

Zwang zur Behandlung von Exceptions

- Eine Anweisung innerhalb der Methode `caller()` ruft eine Methode `method()` auf, die eine geprüfte Exception `E` auslösen kann.
- Dann muss für `caller()` folgendes gelten:
  Entweder in `caller()` wird `E` durch einen Exception-Handler behandelt
  oder `caller()` macht mittels `throws` deutlich, dass die Exception `E` weitergereicht wird.

```
public class Klasse {
    void method() throws SomeException { ... }
}
```

Falsch:

```
public class AndereKlasse {
    void caller() {
        Klasse k = new Klasse();
        k.method();
    }
}
```

Richtig:

```
public class AndereKlasse {
    void caller() throws SomeException {
        Klasse k = new Klasse();
        k.method();
    }
}
```

oder ...

... in `caller()` einen Exception-Handler für `SomeException` verwenden:

```
public class AndereKlasse {
    void caller() {
        Klasse k = new Klasse();
        try {
            k.method();
        }
        catch (SomeException e) {
            ...
        }
        ...
    }
}
```

Propagierung von Exceptions

- Eine ausgelöste Exception `E` wird entlang der Aufrufhierarchie propagiert, bis ein geeigneter Exception-Handler gefunden wurde.
- Ein Exception-Handler ist geeignet, wenn er eine `catch`-Klausel hat, deren Exception-Typ gleich `E` ist oder ein Obertyp davon.
- Die `catch`-Klauseln werden daraufhin in der Reihenfolge geprüft, wie sie im Quelltext angegeben sind.
- Die erste passende `catch`-Klausel wird genommen!

Auswahl des `catch`-Blocks

☞ Die erste passende `catch`-Klausel wird genommen!

falsch:

```
try {
   ...
}
catch (Exception e) {
   ...
}
catch (SpezielleException se) {
   ...
}
```

richtig:

```
try {
   ...
}
catch (SpezielleException se) {
   ...
}
catch (Exception e) {
   ...
}
```

Aufräumen mit `finally`

- Nach einer oder mehreren `catch`-Klauseln kann optional eine `finally`-Klausel folgen.
- Die Anweisungen im Block zu `finally` werden immer ausgeführt:
  - Wenn keine Exception ausgelöst wurde,
  - wenn eine Exception ausgelöst und durch eine `catch`-Klausel behandelt wird und auch
  - wenn eine Exception ausgelöst wird, für die keine passende `catch`-Klausel vorhanden ist (d.h. die Exception wird weitergereicht).
- Typische Anwendung: Freigabe von Ressourcen unabhängig vom Auftreten eines Fehlers.

Auslösen von Exceptions: throw

- Exceptions werden mit Hilfe der `throw`-Klausel ausgelöst.
- Hinter `throw` gibt man ein Exception-Objekt an.
- Typischerweise erzeugt man hierzu ein neues Exception-Objekt mittels `new`.
- Als Konstruktorparameter ist eine Fehlermeldung zulässig.

Beispiel:

```
throw new Exception("Parameterwert ungueltig!");
```

Maßgeschneiderte Exceptions

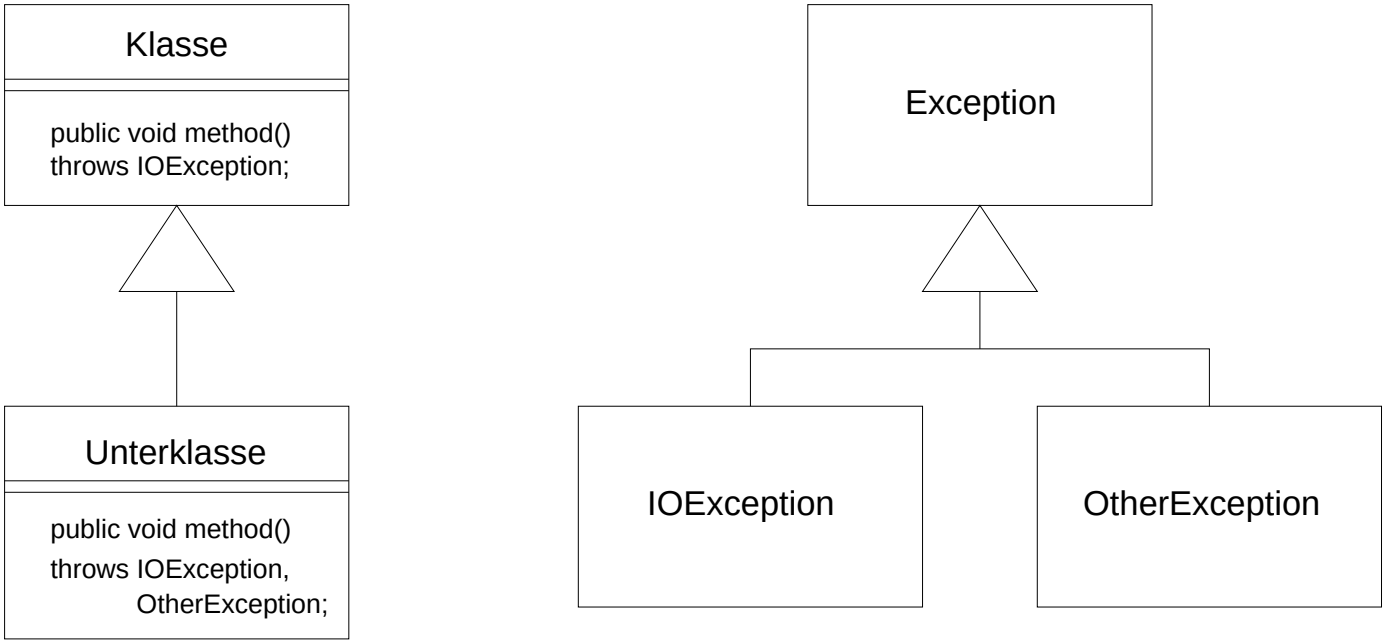
- Für eigene geprüfte Exceptions: Klasse definieren, die abgeleitet von `Exception` ist.
- Ansonsten gelten die üblichen Regeln der Vererbung.

```
public class MyException extends Exception {
   ...
   public MyException() {
      super();
   }

   public MyException(String msg) {
      super(msg);
   }
   ...
}
```

Exceptions und Vererbung

Das folgende Design ist in Java (vernüftigerweise) nicht erlaubt! Warum?



Exceptions und Vererbung (2)

- Durch die `throws`-Klausel werden Exceptions Bestandteil des Vertrages zwischen Klient und Lieferant.
- Sie regeln, was im Falle einer Vertragsverletzung passiert.

☞ Prinzip der Ersetzbarkeit: Spezialisierte Methoden dürfen keine neuen geprüften Exceptions definieren.

- Konsequenz: Das Design der vorangegangenen Folie ist in Java nicht erlaubt!
- Korrekte Lösung: `method()` von `Unterklasse` darf
  - `IOException` auslösen,
  - eine Spezialisierung von `IOException` auslösen oder
  - keine Exception auslösen.
- Und wenn `method()` von `Unterklasse` andere Methoden benutzt, die andere geprüfte Exceptions als `IOException` auslösen können?

- Dann muss `method()` von `Unterklasse` diese Exceptions fangen!
- Eine Umsetzung auf `IOException` ist dabei erlaubt.

```
public class Unterklasse {
    ...
    public void method() throws IOException {
        ...
        try {
            this.otherMethod();
        }
        catch(OtherException e) {
            throw new IOException(...);
        }
        ...
    }
}
```

Schnittstellen und Exceptions

Zur Schnittstellendefinition gehört auch die Deklaration von möglichen Exceptions.

Folgende Konstruktion ist nicht erlaubt:

```
interface Schnittstelle {
   void methode();
}

public class Klasse implements Schnittstelle {
   public void methode() throws Exception { ... }
}
```

☞ Widerspruch zum Prinzip der Ersetzbarkeit.

Lösung: Die Exception muss in der Schnittstelle deklariert werden!

```
interface Schnittstelle {
   void methode() throws Exception;
}
```

- Damit dürfte die Implementierung von `methode()` `Exception` oder eine beliebige Unterklasse von `Exception` auslösen.
- Nicht gestattet: allgemeinere Exceptions oder weitere Exceptions

Und wenn die Definition von `Schnittstelle` nicht angepasst werden kann (z.B. weil `Schnittstelle` aus einer Bibliothek stammt)?

- Dann darf die implementierende Klasse für `methode()` keine geprüften Exceptions auslösen oder weiterreichen!
- Innerhalb von `methode()` müssen alle möglichen geprüften Exception gefangen werden!

Assertions

- Seit Java 1.4 gibt es in Java das Konzept der *Zusicherung (assertion)*.
- Eine Zusicherung ist ein Ausdruck, mit dem Einschränkungen definiert werden, welche die erlaubten Zustände oder das Verhalten von Objekten betreffen.
- Zusicherungen gehören eigentlich zur Spezifikation bzw. Modellierung.
- Zusicherungen im Quelltext prüfen, ob die Spezifikation verletzt ist (Korrektheit).
- Verletzte Zusicherungen lösen in Java ein Objekt vom Typ `AssertionError` aus (Untertyp von `Error`).
- Eine verletzte Zusicherung heißt: Das Programm bzw. die Klasse erfüllt nicht die geforderte Spezifikation.
- Man beachte: `Error` statt `Exception`!

Zusicherungen im Quelltext

- Die Überprüfung erfolgt mit dem Schlüsselwort `assert`:

  ```
  assert boolscher-Ausdruck;
  ```

- Liefert der boolesche Ausdruck den Wert `true`, wird das Programm fortgesetzt.
- Ansonsten wird ein `AssertionError` ausgelöst.
- Optional kann für eine Assertion ein Fehlertext angegeben werden:

  ```
  assert boolscher-Ausdruck : String-Ausdruck;
  ```

Zusicherungen compilieren und aktivieren

- Sie benötigen eine Java-Version $\geq$ 1.4.
- Damit Kompatibilität zu älteren Versionen gewahrt ist, müssen Assertions für den Compiler durch die Option `-source` aktiviert werden:

```
javac -source 1.4 Klasse.java
```

- Assertions sind auch in der virtuellen Maschine standardmäßig nicht aktiviert. Mit Hilfe der Option `-ea` werden die Assertions aktiviert.

```
java -ea Klasse
```

4. Parametrisierbare Klassen

Java now supports *generics*, the most significant change to the language since the addition of inner classes in Java 1.2 — some would say the most significant change to the language ever.

M. Naftalin, P. Wadler, *Java Generics*, O'Reilly, 2006.

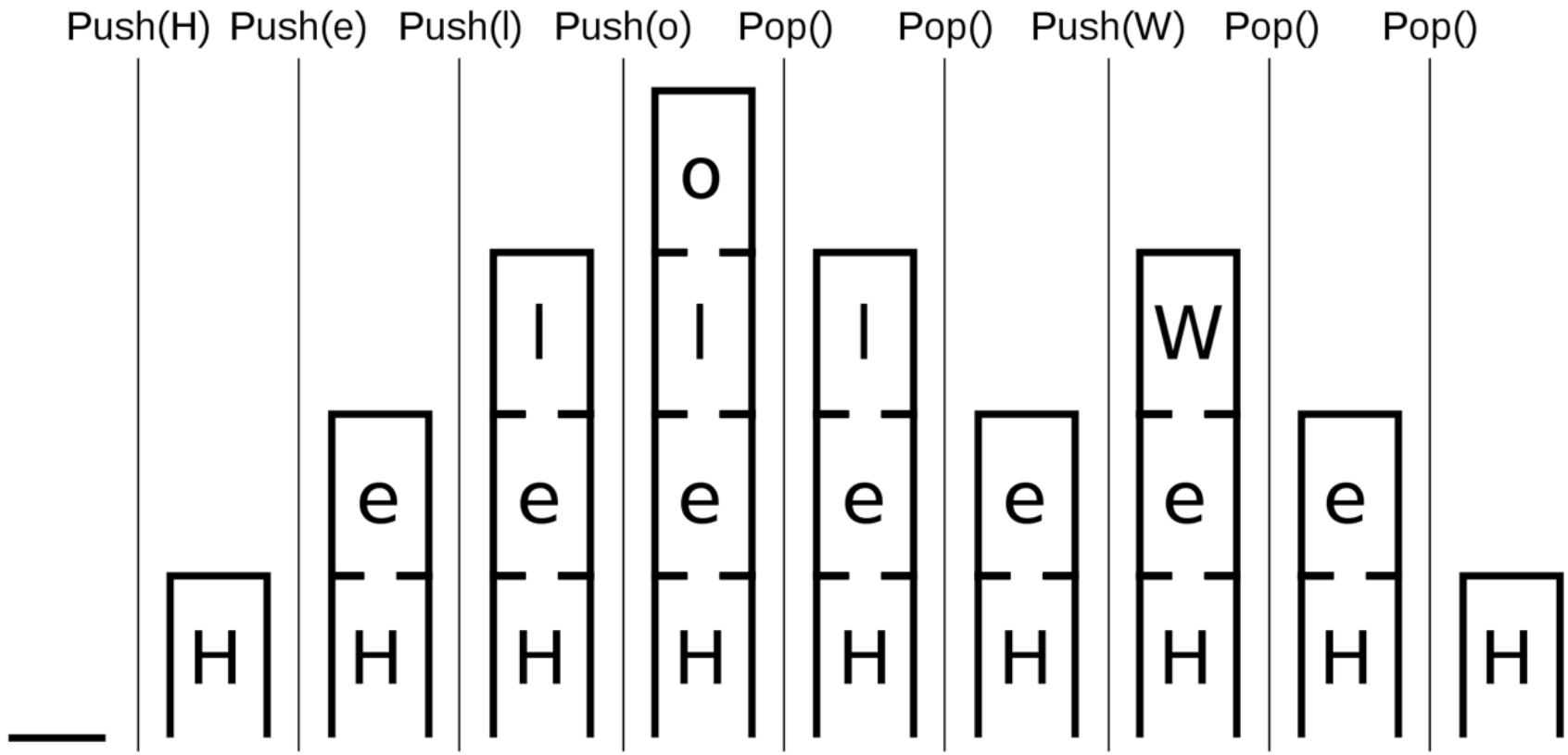
Datenstruktur Stapel (Stack)

Wichtige abstrakte Datentypen bzw. Datenstrukturen der Informatik sind unabhängig von einem Basis- bzw. Komponententyp.

Definition 4.1. Es sei T ein beliebiger Datentyp. Ein *Stapel (Stack)* ist eine Datenstruktur, die folgende Operationen unterstützt:

- `void push(T elem)`
  Legt ein Element $elem$ vom Typ T auf dem Stapel ab.
- `T pop()`
  Entfernt das oberste Element vom Stapel und liefert es als Ergebnis zurück.
- `T peek()`
  Liefert das oberste Element des Stapels, ohne es vom Stapel zu entfernen.

Stapel (Stack)



Naiver Implementierungsansatz

- Für jeden Komponententyp wird der Stapel separat implementiert.
- Die Anweisungen zur Implementierung eines Stapels sind nahezu unabhängig vom Komponententyp T.
- Vergleicht man die Implementierungen eines Stapels z.B. für `int` und `String`, so unterscheiden sich die Implementierungen nur im angegebenen Datentyp.

☞ Beispiel

Ursprünglicher generischer Implementierungsansatz: `Object`

Allgemeine Objektreferenz mittels `Object`.

```
public class Stapel
{
   ...
   void push(Object element) { ... }
   Object pop() { ... }
   Object peek() { ... }
   ...
}
```

☞ Vorgehensweise bis Java 1.4

Generische Objektreferenz mittels `Object` (1)

Probleme:

- Downcasting notwendig nach `pop()` und `peek()`

```
T t = (T) stack.pop();
```

- Keine Typüberprüfung zur Übersetzungszeit

```
Stack stack = new Stack();
stack.push(new Integer(4711)); // wird vom Compiler trotz
stack.push("hallo!");          // verschiedener Typen akzeptiert
```

- Evtl. Typfehler zur Laufzeit

```
stack.push("hallo!");
Integer i = (Integer) stack.pop(); // ClassCastException
```

Generische Objektreferenz mittels `Object` (2)

Und wenn wir doch für jeden benötigten Komponententyp eine eigene Implementierung bereitstellen?

- hoher Aufwand
  - Mehrfache Verwendung des fast gleichen Quelltextes
  - Mehrfaches Testen notwendig
  - entdeckte Fehler müssen mehrfach korrigiert werden
- geringe Abstraktion

Typvariablen

- Seit Java 5 sind *Typvariablen* bei einer Klassendefinition möglich.
- Die damit realisierte Klasse entspricht einem *generischen Datentyp*.
- In der objektorientierten Programmierung bezeichnen wir dies auch als *parametrische Polymorphie*.
- In Java bezeichnet man diese Möglichkeit als *Generics*.

Wirkung:

- Typvariablen ermöglichen es, bei der Definition komplexer Datentypen von den zu Grunde liegenden Basistypen zu abstrahieren.
- Erst bei der Instanziierung muss für die Typvariablen ein konkreter Typ angegeben werden.

Typvariablen: Syntax

```
public class Stapel<T>
{
    ...
    void push(T element) { ... }
    T pop() { ... }
    T peek() { ... }
    ...
}
```

- Hier eine Typvariable T, angegeben in spitzen Klammern hinter dem Klassennamen
- Die Typvariable T steht innerhalb der Klassendefinition für einen beliebigen Referenztyp.
- Innerhalb des Quelltextes einer generischen Klasse kann die Typvariable (fast) überall dort stehen, wo ein Datentyp verlangt ist.

Instanziierung generischer Typen (1)

Bei der Instanziierung eines generischen Typs müssen wir die Typvariable durch einen konkreten Typ ersetzen.

```
Stapel<String> sstapel = new Stapel<String>();
Stapel<Double> dstapel = new Stapel<Double>();
```

Die Stapel sind jetzt typsicher:

```
dstapel.push("text");   // hier meckert der Compiler
```

Kein Downcast notwendig:

```
dstapel.push(new Double(Math.PI));
Double d = dstapel.peek();          // auch ohne Downcast kein Problem
```

Instanziierung generischer Typen (2)

Leider wäre die folgende Deklaration nicht erlaubt:

```
Stapel<int> istapel = new Stapel<int>();
```

Grund: Typvariablen dürfen nur durch Referenztypen instanziiert werden.

Wie können wir aber dann einen Integer-Stapel erzeugen? Verwendung von Wrapper-Klassen: `Integer`, `Double`, etc.

```
Stapel<Integer> istapel = new Stapel<Integer>();
```

Exkurs: Wrapper-Klassen (Hüllklassen)

- Instanzen von *Wrapper-Klassen (Hüllklassen)* haben die Aufgabe, einen primitiven Wert als Objekt zu repräsentieren.
- Es gibt für jeden einfachen Datentyp eine zugehörige Wrapper-Klasse, z.B. `int/Integer`, `double/Double`, `char/Character`.
- `Integer i = new Integer(4711);`
- Wie Strings sind Instanzen von Wrapper-Klassen grundsätzlich *unveränderlich (immutable)*.

Exkurs: Boxing und Unboxing

- Die Repräsentation eines einfachen Wertes als Objekt mit Hilfe einer Wrapper-Klasse bezeichnen wir auch als *Boxing*.

```
Integer io = new Integer(4711); // Boxing
```

- Der Zugriff auf den einfachen Wert nennen wir *Unboxing*.

```
int ival = io.intValue(); // Unboxing
```

Exkurs: Autoboxing (1)

- Die manuelle Ausführung von Boxing und Unboxing ist oft unhandlich.

```
Stapel<Integer> istapel = new Stapel<Integer>();
istapel.push( new Integer(4711) );
int iv = istapel.peek().intValue();
```

- Die automatische Umwandlung von Werten einfacher Datentypen in Instanzen einer Wrapper-Klasse und umgekehrt wird als *Autoboxing* bezeichnet.
- Java beherrscht Autoboxing seit Java 5.

```
int i     = 4711;
Integer j = i;          // automatisches Boxing
int k     = j;          // automatisches Unboxing
```

Exkurs: Autoboxing (2)

Damit ist natürlich auch möglich:

```
Stapel<Integer> istapel = new Stapel<Integer>();
istapel.push(4711);          // Boxing
int iv = istapel.pop();      // Unboxing
```

Vorsicht bei Vergleichen mit == und Autoboxing!

Typanpassungen

Ein instanziierter generischer Typ lässt sich durch Typanpassung auf eine allgemeine Form bringen:

```
Stapel<Integer> istapel = new Stapel<Integer>();
Stapel          stapel  = (Stapel) istapel;
```

Jetzt findet für `stapel` keine Typprüfung mehr statt. Daher würde

```
stapel.push("No Integer");
```

keinen Fehler zur Übersetzungszeit liefern.

Typvariablen in Methoden

Typvariablen können auch auf Methodendeklarationen beschränkt sein:

```
public class Util
{
   public static <T> T zufall(T o1, T o2)
   {
      return Math.random() < 0.5 ? o1 : o2;
   }
}
```

Die Angabe von <T> beim Klassennamen entfällt und verschiebt sich auf die Methodendefinition.

☞ Typinferenz (Schlussfolgerungen über Datentypen), siehe Beispiel

Mehrere Typvariablen

- Bei der Definition einer generischen Klasse können auch mehrere Typvariablen verwendet werden.
- Beispiel: Eine generische Klasse für die Repräsentation einer Funktion

$$f : T \longrightarrow U$$

  mit endlichem Definitionsbereich T und Wertebereich U.
- entspricht einer funktionalen endlichen Relation $T \times U$.

```
public class Zuordnung <T,U> {
    ...
    void put(T t, U u) { ... }
    U get(T t) { ... }
    ...
}
```

- put: ordnet einem Wert $t \in T$ den Funktionswert $u \in U$ zu
- get: liefert zu einem Wert $t \in T$ den zugehörigen Funktionswert $f(t) \in U$
- Beispielhafte Nutzung:

```
Zuordnung<Integer,String> z = new Zuordnung<Integer,String>();
                          // Definition der Zuordnungen:
z.put(1,"eins");          // 1 --> "eins"
z.put(2,"zwei");          // 2 --> "zwei"

String s2 = z.get(2);     // liefert "zwei"
```

Schachtelung von Generischen Typen

- Bei der Instanziierung von generischen Klassen kann als Typ auch eine instanziierte generische Klasse verwendet werden.
- Beispiel: Ein Stapel von String-Listen:

```
Stapel<ArrayList<String>> stapelVonListen
                              = new Stapel<ArrayList<String>>();
```

Die new-Operation erzeugt natürlich wiederum nur den Stapel, aber nicht die im Stapel ablegbaren Listen. Diese müssen separat erzeugt werden.

```
// neue Liste erzeugen und auf Stapel ablegen
stapelVonListen.push(new ArrayList<String>());

// Liefert oberste Liste des Stapels
ArrayList<String> list = stapelVonListen.peek();
```

Typvariablen und Schnittstellen

- Typvariablen dürfen auch bei der Definition von Schnittstellen verwendet werden. Damit enststehen generische Schnittstellen.
- Syntax für Typvariable analog zu Klassen:

```
public interface List<T> {
   void add(T elem);  // Element hinzufuegen
   int size();        // Anzahl Listenelemente bestimmen
   T get(int i);      // Liefert das i-te Element
   ...
}
```

Die Schnittstelle Comparable

- `Comparable<T>` (aus dem Paket `java.lang`) ist eine generische Schnittstelle für die Vergleichbarkeit von Objekten eines Typs T.
- Definition mit Generics (vgl. Folie 56):

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- Es sei `this` das Objekt, für das die Methode `compareTo()` aufgerufen wird. Dann soll `compareTo()` liefern:
  - $== 0$, wenn `this` und `o` gleich sind,
  - > 0, wenn `this` größer als `o` ist und
  - < 0, wenn `this` kleiner als `o` ist.

- Einige wichtige Klassen des Java-API implementieren `Comparable<T>`, z.B. die Wrapper-Klassen.
- D.h. die Klasse `String` implementiert `Comparable<String>`, `Integer` implementiert `Comparable<Integer>`, usw.
- Vgl. Übungsaufgabe 1 (a) auf Aufgabenblatt 3: Mit Verwendung generischer Typen

```
public class Person implements Comparable<Person> {
   ...
   public int compareTo(Person o) {
      ...
   }
   ...
}
```

- Mit Hilfe von `Comparable<T>` können generische Sortierverfahren implementiert werden.
  ☞ Übungsaufgabe
- Eine ähnliche Schnittstelle ist `Comparator<T>` aus dem Paket `java.util`.

Die Schnittstelle Comparator

Problem der Schnittstelle `Comparable`:

- Sie muss von der Klasse implementiert werden, deren Instanzen man vergleichen möchte.
- Damit ist in der Klasse nur ein Sortierkriterium definierbar.
- Beispiel: Übungsaufgabe 1, Blatt 3: `Person`-Instanzen können nur nach Namen aber nicht z.B. nach PLZ sortiert werden.

Lösung:

- Klassenlogik und Vergleichslogik trennen.
- Komparatoren definieren, die einen Vergleich von Instanzen einer Klasse K nach eigener Logik durchführen.
- Jeder Komparator kann die Instanzen von K auf andere Weise vergleichen.

Die Schnittstelle Comparator (2)

Generische Schnittstelle `Comparator<T>` aus dem Paket `java.util`:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object o);
}
```

`compare(T o1, T o2)` folgt der üblichen Logik und liefert:

- $== 0$, wenn o1 gleich o2 ist,
- > 0, wenn o1 größer als o2 ist und
- < 0, wenn o1 kleiner als o2 ist.

Die Schnittstelle Comparator (3)

Man beachte:

- Die Methode `compare()` wird nicht auf einer Instanz der Klasse `T` aufgerufen, sondern auf einer Komparator-Instanz, die `Comparator<T>` implementiert.
- Wir können nun ganz unterschiedliche Klassen für unterschiedliche Arten des Vergleich implementieren.

Beispiel:

- Für einen Vergleich von `Person`-Instanzen nach Namen:

```
public class NameKomparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        ...
    }
    ...
}
```

- Für einen Vergleich von `Person`-Instanzen nach Postleitzahlen:

```
public class PLZKomparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        ...
    }
    ...
}
```

- Das Java SDK bietet uns generische Sortiermethoden, wobei die Sortierung über einen Komparator gesteuert werden kann (z.B. in der Klasse `java.util.Arrays`, ungefähr so):

```
public class Arrays {
    ...
    public static <T> void sort(T[] a, Comparator<T> c) { ... }
    ...
}
```

- Dies können wir nun wie folgt nutzen:

```
public class Test {
    public static void main(String[] args) {
        Person[] p = new Person[...];
        p[0] = ... // Feld fuellen;
        ...
        Comparator<Person> c1 = new NameKomparator();
        Comparator<Person> c2 = new PLZKomparator();

        Arrays.sort(p,c1);   // sortiert Feld p nach Namen
        Arrays.sort(p,c2);   // sortiert Feld p nach Postleitzahlen
    }
}
```

Die Schnittstellen Iterable und Iterator

- `java.lang.Iterable`:

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

- `java.lang.Iterator`:

```
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

- Diese Schnittstellen dienen dazu, auf die Elemente einer Kollektion iterativ zuzugreifen.
- Alle Kollektionen des Java Collection Frameworks implementieren `Iterable`, z.B. `ArrayList`.
- Auch die Schnittstelle `List` ist von `Iterable` abgeleitet.

Nutzung von Iterator

```
List<String> list = new ArrayList<String>();
list.add("eins"); list.add("zwei"); list.add("drei");

Iterator<String> iter = list.iterator();
while ( iter.hasNext() ) {            // Elemente der Liste sequentiell ausgeben
    System.out.println(iter.next());
}
```

Foreach

Java bietet für den iterativen lesenden Zugriff auf eine Kollektion eine spezielle Form der `for`-Schleife:

```
for (Datentyp Schleifenvariable : Kollektion )
    Anweisung
```

Voraussetzungen:

- *`Kollektion`* muss die Schnittstelle `Iterable` implementieren.
- *`Datentyp`* muss zum Basisdatentyp von *`Kollektion`* passen.

Wirkung:

- Es wird implizit ein Iterator erzeugt, mit dem die Kollektion vollständig durchlaufen wird.
- Jedes Element der Kollektion (vgl. `next()` von `Iterator`) wird dabei genau einmal an die Schleifenvariable gebunden.

Beispiele:

```
List<String> list = new ArrayList<String>();
list.add("eins"); list.add("zwei"); list.add("drei");

for (String s : list) {
    System.out.println(s);
}
```

Die Schleife wird dabei implizit ausgeführt als:

```
for (Iterator<String> iter=list.iterator() ; iter.hasNext() ; ) {
    String s = iter.next();
    System.out.println(s);
}
```

Auch für Felder kann diese Form der Schleife genutzt werden:

```
public static double sumArray(double[] a) {
    double s = 0.0;
    for (double x : a) {
        s += x;
    }
    return s;
}
```

Realisierungsmöglichkeiten für parametrische Polymorphie

- **heterogen**
  Für jeden konkreten Typ wird individueller Code erzeugt (type expansion).
  Diesen Ansatz verfolgt z.B. C++ (templates directory).
  Vorteile: Typinstanziierung zur Laufzeit bekannt, mehr Optimierungsmöglichkeiten
  Nachteil: Erzeugter Objektcode wird aufgebläht (code bloat).
- **homogen**
  Kein individueller Bytecode, `Object` statt der Typvariablen, Typanpassungen für einen konkreten Typ, Typüberprüfungen nur zur Übersetzungszeit (type erasure).
  Realisierung für Java.
  Vorteile: schlanker Bytecode, einfachere Verwendung
  Nachteil: Typinstanziierung zur Laufzeit nicht bekannt.

Typlöschung (type erasure)

Zur Laufzeit ist der konkrete Typ der Typvariablen für ein Objekt nicht bekannt. Dies hat Konsequenzen für `instanceof`.

Folgendes liefert einen Übersetzungsfehler:

```
Stapel<String> stapel = new Stapel<String>();

if ( stapel instanceof Stapel<String> ) ... // Compiler-Fehler
if ( stapel instanceof Stapel ) ...         // OK
```

Grund: Zur Laufzeit liegt nur die Typinformation `Stapel` vor.

```
Stapel<String> stapel1 = new Stapel<String>();
Stapel<Integer> stapel2 = new Stapel<Integer>();
System.out.println(stapel1.getClass() == stapel2.getClass()); // true!
```

Raw-Type

Eine Instanziierung mit einem konkreten Typ muss nicht unbedingt stattfinden. Ohne Instanziierung gilt implizit `Object` als konkreter Typ.

```
Stapel stapel = new Stapel<String>();
Stapel ostapel = new Stapel();
```

Solche Typen heissen *Raw-Type*.

Raw-Typen können wir wie parametrisierten Typen verwendet werden, es findet aber keine Typüberprüfung bzgl. des Komponentenyps statt.

Eventuell gibt der Compiler Warnungen aus.

☞ Vorsicht vor Typproblemen zur Laufzeit.

Generische Felder

- Konsequenz der Typlöschung: Es können keine Felder mit instanziierten Typvariablen angelegt werden. So liefert z.B. die folgende Anweisung einen Übersetzungsfehler:

```
Stapel<String>[] stapelFeld = new Stapel<String>[10];
```

- Stattdessen kann für die Erzeugung des Feldes ein Raw-Type als Komponententyp verwendet werden. Die Zuweisung an eine Variable mit instanziiertem Typ ist möglich, liefert aber je nach Compiler eine Warnung:

```
Stapel<String>[] stapelFeld = new Stapel[10];
```

- Zugriffe auf das Feld sind wieder typsicher:

```
stapelFeld[0] = new Stapel<String>();  // OK
stapelFeld[1] = new Stapel<Integer>(); // Fehler bei Compilierung
```

- Bei der Implementierung eines generischen Typs kann die Typvariable auch nicht als Komponententyp für ein Feld verwendet werden!

- Folgendes liefert einen Übersetzungsfehler:

```
public class Stapel<T> {
    private T[] stapel = new T[4];
    ...
}
```

- Das Problem ist wieder nur die Felderzeugung, nicht die Variablendeklaration!
- Alternative: `Object` als Basistyp mit Downcast:

```
private T[] stapel = (T[]) new Object[4];
```

oder Verzicht auf Felder in eigenen generischen Klassen und stattdessen Verwendung der generischen Klassen des Java-API (z.B. `ArrayList<T>` statt `T[]`).

Kovarianz bei Kollektionen

- In Java verhalten sich Felder kovariant.
- Wenn `Fussballer` eine Unterklasse von `Sportler` ist, dann ist `Fussballer[]` ein Untertyp von `Sportler[]`.
- Daher ist folgenden problemlos möglich:

```
Fussballer[] fussballer = new Fussballer[11];
Sportler[] sportler = fussballer;
```

- In der objektorientierten Programmierung bezeichnet man dies als *Kovarianz*: Ein Aspekt (hier: Feldbildung) wird gleichartig zur Vererbungsrichtung (Fussballer ist Unterklasse von Sportler) eingesetzt.
- Kovarianz stellt bei lesendem Zugriff kein Problem dar und garantiert für diesen Fall Substituierbarkeit.

- Beispiel:

```
class Sportler { ...  public void gibAus() { ... } ... }
class Fussballer extends Sportler { ... }

public static void gebeSportlerAus(Sportler[] sportler) {
    for (int i=0 ; i<sportler.length ; i++)
        sportler[i].gibAus();
}

public static void irgendeineMethode() {
   ...
   Fussballer[] fussballer = new Fussballer[11];
   fussballer[0] = new Fussballer(...);
   ...
   gebeSportlerAus(fussballer);    // impliziter Up-Cast
   ...
}
```

Generics und Vererbung (1)

- Generics sind untereinander nicht kovariant sondern invariant!
- D.h. die Typhierarchie der Typvariablen überträgt sich nicht auf den generischen Typ.
- Beispiel: `ArrayList<Fussballer>` ist kein Untertyp von `ArrayList<Sportler>`.
- Daher liefert der Compiler für folgende Anweisungen einen Fehler:

```
ArrayList<Sportler> sportler = new ArrayList<Fussballer>();
```

- Begründung: Kovarianz ist bei schreibendem Zugriff auf eine Kollektion problematisch.

Problem der Kovarianz

- Wegen der Kovarianz bei Feldern liefert der Java-Compiler für die folgenden Anweisungen keinen Fehler bei der Übersetzung:

```
class Sportler { ... }
class Fussballer extends Sportler { ... }
class Handballer extends Sportler { ... }
...
Sportler[] sportler = new Fussballer[11];   // impliziter Up-Cast
sportler[0] = new Handballer();             // ArrayStoreException
```

- Zur Laufzeit würde aber eine `ArrayStoreException` ausgelöst, denn eine `Handballer`-Instanz kann natürlich nicht in ein Feld von `Fussballern` eingefügt werden.
- Durch den impliziten Upcast nach `Sportler[]` geht die Typinformation für das `Fussballer`-Feld verloren.

Keine Kovarianz bei Generics

- Im Gegensatz zu Feldern verhalten sich Generics nicht kovariant.
- Dies verhindert den Laufzeitfehler auf der vorangegangenen Folie:

```
class Sportler { ... }
class Fussballer extends Sportler { ... }
class Handballer extends Sportler { ... }
...
                                          // Compiler meldet Fehler
ArrayList<Sportler> sportler = new ArrayList<Fussballer>();
sportler.add(new Handballer());       // deshalb kann es hier nicht zu ei
                                          // Laufzeitfehler kommen
```

Generics und Vererbung (2)

- Wir haben gesehen, dass Kovarianz bei schreibendem Zugriff problematisch sein kann!
- Dagegen wäre folgendes vom Prinzip her unproblematisch:

```
class Mensch { ... }
class Sportler extends Mensch { ... }
class Handballer extends Sportler { ... }
...
Sportler[] sportler = new Mensch[5];
sportler[0] = new Handballer();
sportler[1] = new Sportler();
...
ArrayList<Sportler> sportler = new ArrayList<Mensch>();
sportler.add(new Handballer());
sportler.add(new Sportler());
```

würde aber nicht vom Compiler akzeptiert, weder für Felder noch für Generics.

- *Kontravarianz*: Verwendung eines Aspektes entgegen der Vererbungsrichtung
- Bei schreibendem Zugriff ist Kontravarianz typsicher.

☞ Statt festeingebauter Kovarianz bieten Generics in Java durch Typeinschränkungen und Wildcards die Möglichkeit, je nach Situation Kovarianz, Kontravarianz oder Invarianz zu definieren.

Typeinschränkungen bei Generics

Die zugelassenen konkreten Datentypen für eine Typvariable können mit `extends` auf Untertypen eines Obertyps eingeschränkt werden.

Syntax:

```
<T extends O>
```

Für die Typvariable T sind dann O und alle Untertypen von O zulässig.

Beispiel: Ein typsicheres generisches `max`:

```
public static <T extends Comparable<T>> T max(T o1, T o2)
{
   return o1.compareTo(o2) >= 0 ? o1 : o2;
}
```

- ☞ **extends** auch für Untertypen von Schnittstellen!
- ☞ Das ist besser als

```
public static <T> Comparable<T> max(Comparable<T> o1, Comparable<T> o2)
```

Warum?

Typeinschränkung mit `super`

- Mit `super` statt `extends` kann eine Einschränkung auf Obertypen statt auf Untertypen erfolgen.
- `<T super O>`
- Damit dürfen für die Typvariable T O oder Obertypen von O eingesetzt werden.
- Diese Einschränkung wird z.B genutzt, wenn eine mit T parametrisierte generische Datenstruktur manipuliert wird (Kontravarianz).

Kombinationen von Obertypen

- Soll die Typvariable T zu mehreren Obertypen passen, lassen sich Kombinationen bilden.
- Man beachte: Für T kann nur eine Oberklasse angegeben werden, da es in Java nur einfache Vererbung bei Klassen gibt.
  Weitere Obertypen müssen Schnittstellen sein.
- Beispiel: `<T extends O & I1 & I2 & I3>`
  Bedeutung: Für die Typvariable T sind alle Typen erlaubt, die Untertypen der Klasse O und der Schnittstellen I_1, I_2 und I_3 sind (also diese Schnittstellen implementieren).
  genauer: T muss von O abgeleitet sein, und I_1, I_2 und I_3 implementieren.

Generische Variablen: Wildcards

Folgendes funktioniert nicht (weil sich Generics nicht kovariant verhalten):

```
class Sportler { ... }
class Fussballer extends Sportler { ... }
class Handballer extends Sportler { ... }
class Biertrinker { ... }
...
ArrayList<Sportler> sportler;
ArrayList<Fussballer> fussballer = new ArrayList<Fussballer>();
ArrayList<Handballer> handballer = new ArrayList<Handballer>();
sportler = fussballer;     // Compiler liefert Fehler!
```

- Es ist jetzt aber möglich, statt konkreter Typangaben Wildcards anzugeben und
- diese mit zusätzlichen Einschränkungen (`extends, super`) zu versehen.

Generische Variablen: Wildcards (2)

Beispiel:

```
ArrayList<? extends Sportler> sportler;
ArrayList<Fussballer> fussballer = new ArrayList<Fussballer>();
ArrayList<Handballer> handballer = new ArrayList<Handballer>();
ArrayList<Biertrinker> biertrinker = new ArrayList<Biertrinker>();
sportler = fussballer; sportler = handballer;     // OK!
sportler = biertrinker;                           // Fehler!
```

Generische Variablen: Wildcards (3)

Wildcard mit `extends` liefert uns also Kovarianz: Für

```
public static void gebeSportlerAus(ArrayList<? extends Sportler> sportler) {
    for (Sportler s : sportler) {
        s.gibAus();
    }
}
```

ist

```
ArrayList<Fussballer> fussballer = new ArrayList<Fussballer>();
fussballer.add(new Fussballer(...)); ...
gebeSportlerAus(fussballer);
```

kein Problem mehr.

Generische Variablen: Wildcards (4)

Wildcard mit super liefert uns Kontravarianz: Für

```
public static void packEinenFussballerDrauf(Stapel<? super Fussballer> stapel) {
    stapel.push(new Fussballer());
}
```

ist

```
Stapel<Sportler> sportler = new Stapel<Sportler>();
packEinenFussballerDrauf(stapel);
```

kein Problem.

Generische Variablen: Wildcards (5)

- Kovarianz und Kontravarianz können natürlich auch zusammen auftreten.
- Beispiel: Eine Methode, die alle Elemente einer Liste `src` in eine Liste `dest` kopiert:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    ...
}
```

- Kovarianz für die Liste `src`, aus der gelesen wird.
- Kontravarianz für die Liste `dest`, in die geschrieben wird.
- Nutzung:

```
List<Fussballer>  fussballer  = new ArrayList<Fussballer>();
List<Fussballer>  fussballer2 = new ArrayList<Fussballer>();
List<Sportler>    sportler    = new LinkedList<Sportler>();
List<Biertrinker> biertrinker = new ArrayList<Biertrinker>();
```

```
copy(fussballer2, fussballer);   // OK fuer T=Fussballer
copy(sportler, fussballer);      // OK fuer T=Sportler oder T=Fussballer
copy(fussballer, sportler);      // Fehler
copy(sportler, biertrinker);     // Fehler
copy(biertrinker, sportler);     // Fehler
```

The Get and Put Principle:

Use an `extends` wildcard when you only *get* values out of a structure, use a `super` wildcard when you only *put* values into a structure, and don't use a wildcard when you *both* get and put.

M. Naftalin, P. Wadler, *Java Generics*, O'Reilly, 2006.

5. Listen

- Wichtige abstrakte Datentypen in Zusammenhang mit Listen kennen,
- Implementierungsansätze für Warteschlangen verstehen, sowie deren Vor- und Nachteile benennen können,
- den Aufbau von verketteten Listen und die zugehörigen Algorithmen implementieren können,
- die Effizienz von Listenoperationen einschätzen können und
- den Begriff der amortisierten Laufzeitanalyse kennen und einordnen können.

Warteschlange

Definition 5.1. Es sei T ein beliebiger Datentyp. Eine *Warteschlange (Queue)* ist eine Datenstruktur, die die folgenden Operationen unterstützt:

- `void enqueue(T elem)`
  Fügt ein Element `elem` an das Ende der Warteschlange an.
- `void dequeue()`
  Entfernt das erste Element der Warteschlange.
- `T front()`
  Liefert das Erste Element der Warteschlange, ohne es zu entfernen.

Beispiel: Warteschlange

enqueue(4711); enqueue(7643); enqueue(1234);

4711	7643	1234

dequeue(); enqueue(1999); enqueue(2812); enqueue(32168);

7643	1234	1999	2812	32168

dequeue(); dequeue(); dequeue(); enqueue(1683);

2812	32168	1683

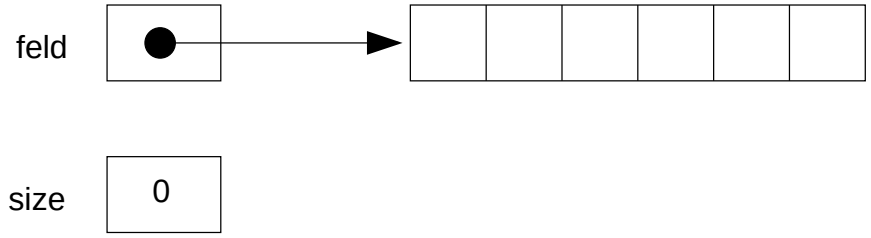
Bemerkungen: Warteschlange

- Die Warteschlange arbeitet nach dem *FIFO-Prinzip (first in, first out)*.
- Im Gegensatz dazu ein Stack: LIFO (last in, first out)
- Die generische Schnittstelle `Queue` (aus dem Paket `java.util`) entspricht der Spezifikation einer Warteschlange.
- Es gibt in `java.util` verschiedene Klassen, die die Schnittstelle `Queue` implementieren.
- Wir interessieren uns für Implementierungsansätze und deren Eigenschaften (Effizienz der Operationen).

Implementierungsansatz: Feld

- Wir speichern die Elemente der Warteschlange in einem Feld fester Größe. In einer weiteren Variablen (`size`) merken wir uns, wie viele Feldelemente belegt sind.

```
public class Warteschlange<T> {
    private final int FIXED_MAX_SIZE = ...;
    ...
    private T[] feld = (T[]) new Object[FIXED_MAX_SIZE];
    private int size = 0;
    ...
}
```



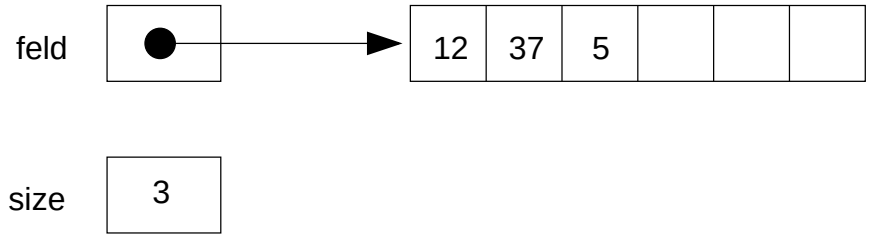
- `void enqueue(T elem)`

  Wir legen `elem` an der ersten freien Stelle im Feld ab und inkrementieren `size`.

  ```
  feld[size] = elem;
  size++;
  ```

  Dies geht natürlich nur, wenn die Kapazität des Feldes noch nicht erschöpft ist, d.h. `size < FIXED_MAX_SIZE` gilt.

  Beispiel: `enqueue(12); enqueue(37); enqueue(5);`



- T front()

  Im Normalfall (`size > 0`) geben wir einfach das erste Element des Feldes zurück:

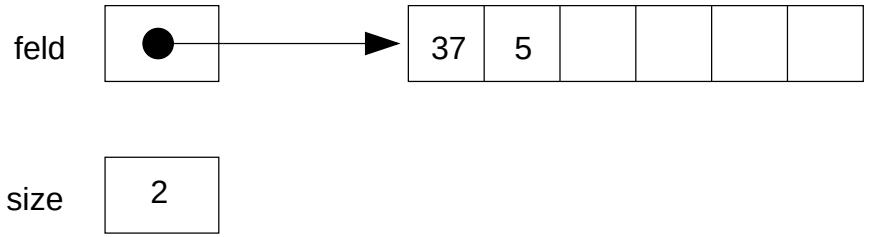
  ```
  return feld[0];
  ```

- `void dequeue()`

  Wir verschieben alle Elemente ab der zweiten Position im Feld um eine Position nach vorne und dekrementieren `size`.

```
for(int i=1 ; i<size ; i++) {
   feld[i-1] = feld[i];
}
size--;
```

  Beispiel: `dequeue();`

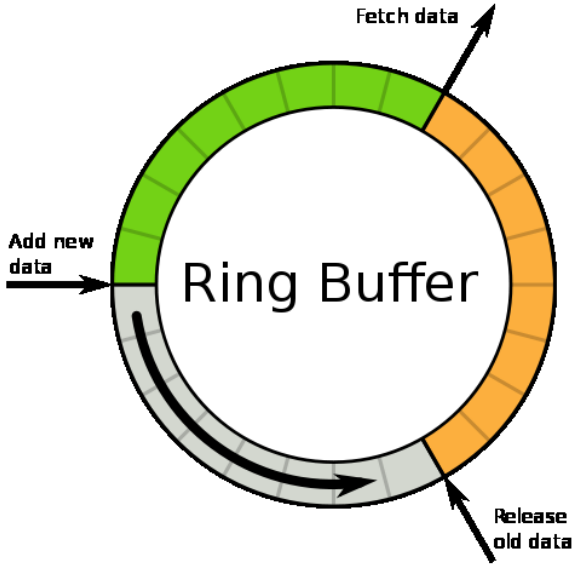


Effizienz

- Es sei n die Anzahl der Elemente in der Warteschlange.
- Zeitaufwand für `enqueue()` und `front()`: $O(1)$
- Zeitaufwand für `dequeue()`: $O(n)$
- Speicherplatzverbrauch: $O(n)$
- Probleme:
  - Bei `size == FIXED_MAX_SIZE` ist kein `enqueue()` mehr möglich.
  - Alternative: Neues größeres Feld anlegen und Inhalte kopieren, dann aber Zeitaufwand nicht mehr $O(1)$ sondern $O(n)$ für `enqueue()`.

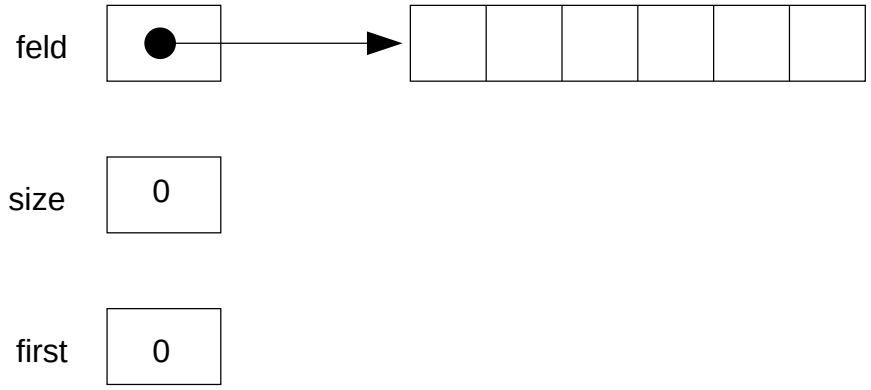
Implementierungsansatz: Ringpuffer

- Bisher: Hoher Zeitaufwand durch das Verschieben der Elemente bei `dequeue()`.
- Besser: Statt die Elemente zu verschieben verschiebt man den Anfangsindex. Diesen merkt man sich in einer weiteren Instanzvariablen.
- Entspricht logisch einem sogenannten *Ring-Puffer*.



- In einer zusätzlichen Variablen (`first`) merken wir uns, bei welcher Feldposition die Warteschlange beginnt.

```
public class Warteschlange<T> {
    public final int FIXE_MAX_SIZE = ...;
    ...
    private T[] feld = (T[]) new Object[FIXED_MAX_SIZE];
    private int size = 0;
    private int first = 0;
    ...
}
```



- `void enqueue(T elem)`

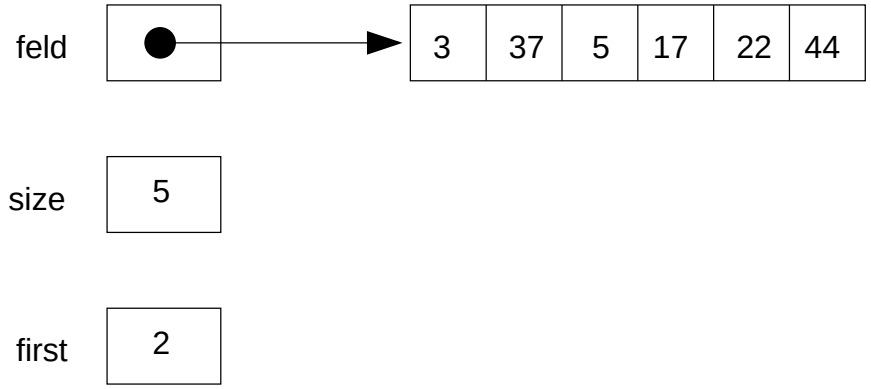
  Man beachte, dass die Warteschlange nicht unbedingt bei Index 0 beginnt! Bei Überlauf wird wieder ab vorne wieder aufgefüllt.

  ```
  feld[(first+size) % FIXED_MAX_SIZE] = elem;
  size++;
  ```

  Dies geht wiederum nur, wenn die Kapazität des Feldes noch nicht erschöpft ist.

  Beispiel:

  ```
  enqueue(12); enqueue(37); enqueue(5); dequeue(); dequeue(); enqueue(17);
  enqueue(22); enqueue(44); enqueue(3);
  ```



Erstes Element der Warteschlange bei Index `first`.
Letztes Element der Warteschlange bei Index `(first+size-1)%FIXED_MAX_SIZE`.
Das Element 37 gehört nicht zur Warteschlange!

- T front()
  Falls `size > 0`: Das erste Element der Warteschlange liegt in der Feldkomponente mit Index `first`:

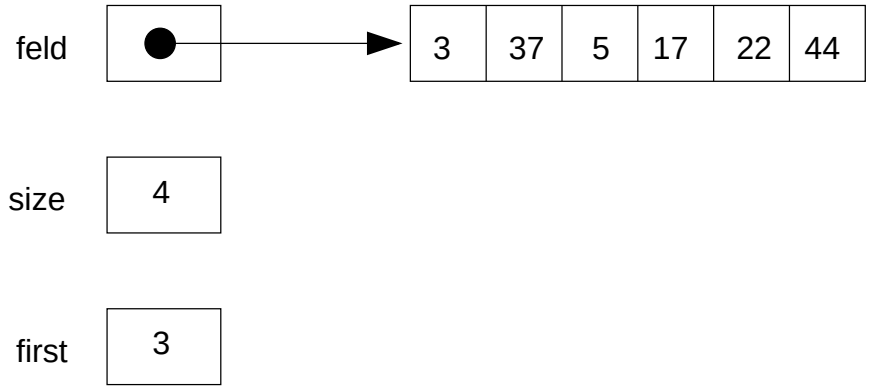
```
return feld[first];
```

- `void dequeue()`

  Wir inkrementieren `first`, müssen dabei aber einen möglichen Überlauf berücksichtigen, und dekrementieren `size`.

```
first = (first+1) % FIXED_MAX_SIZE;
size--;
```

  Beispiel: `dequeue();`



  Beginn der Warteschlange jetzt bei Element 17, Ende bei 3, 37 und 5 gehören nicht dazu.

Effizienz

- Zeitaufwand für alle Operationen: $O(1)$
- Es bleibt das Problem der beschränkten Kapazität, damit liegt keine dynamische Datenstruktur vor.

  Als *dynamische Datenstruktur* bezeichnet man Datenstrukturen, die eine flexible Menge an Arbeitsspeicher reservieren.
- Die Erzeugung eines neuen größeren Feldes führt zu einem Zeitaufwand von $O(n)$ für `enqueue()`.

Deque

Eine *Deque (Double-Ended Queue)* erlaubt das Einfügen, Löschen und den Zugriff sowohl am Anfang als auch am Ende der Elementliste.

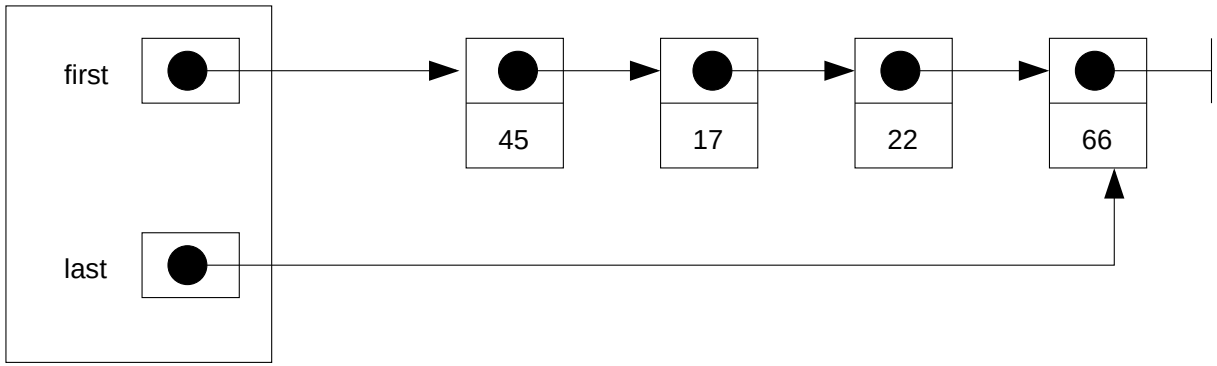
- `void insert(T elem)`
  Fügt ein Element `elem` am Anfang der Elementliste ein.
- `void append(T elem)`
  Fügt ein Element `elem` am Ende der Elementliste ein (entspricht `enqueue()`).
- `void deleteFront()`
  Entfernt das erste Element der Elementliste (entspricht `dequeue()`).
- `void deleteRear()`
  Entfernt das letzte Element der Elementliste.
- `T front()`
  Liefert das erste Element.

- `T rear()`
  Liefert das letzte Element.

Prinzipiell sind die bisher gezeigten Implementierungsansätze auch für eine Deque geeignet, mit den schon vorgestellten Vor- und Nachteilen.

Einfach verkettete Liste

- Für jedes einzelne Element der Liste wird ein Hilfsobjekt erzeugt.
- Jedes Hilfsobjekt enthält zwei Instanzvariablen:
  - den zu speichernden Wert bzw. einen Verweis auf das zu speichernde Objekt und
  - einen Verweis auf das nächste Hilfsobjekt in der Liste.
- Das die Liste repräsentierende Objekt enthält wiederum zwei Verweise:
  - Einen Verweis auf das Hilfsobjekt zum ersten Listenelement und
  - einen Verweis auf das Hilfsobjekt zum letzten Listenelement.

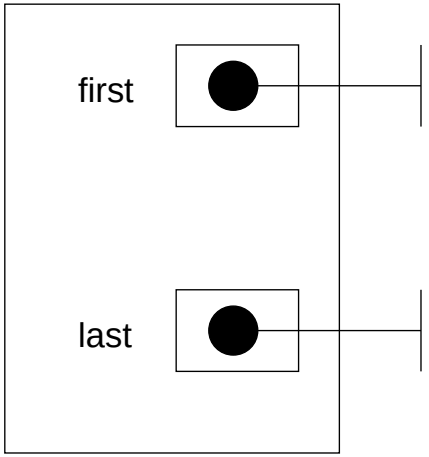


Implementierungsansatz: Verkettete Liste

- `first` und `last` enthalten zu Anfang die Nullreferenz (`null`), dies entspricht einer leeren Warteschlange.

```
public class Warteschlange<T> {
    ...
    private class Item {
        T    value;
        Item next;
    }

    private Item first = null;
    private Item last = null;
    ...
}
```



- `void enqueue(T elem)`

  Wir legen ein Hilfsobjekt an.

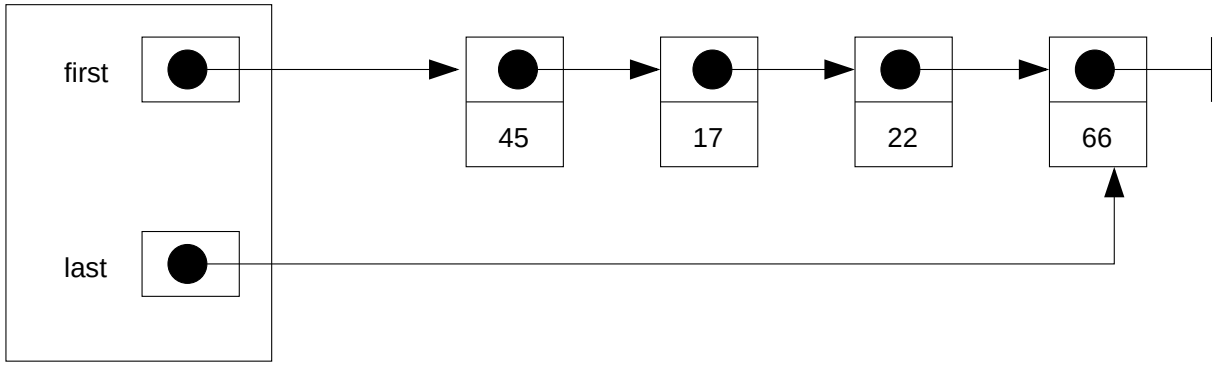
  Wenn die Warteschlange bisher leer ist, lassen wir `first` und `last` auf das Hilfsobjekt verweisen.

  Andernfalls müssen wir das neue Hilfsobjekt hinten an die Liste hängen.

```
Item item = new Item();
item.next = null;
item.value = elem;

if ( last == null ) {
    first = last = item;
}
else {
    last.next = item;
    last = item;
}
```

Beispiel: `enqueue(45); enqueue(17); enqueue(22); enqueue(66);`



- `T front()`

  Im Normalfall (`first != null`) geben wir das Element des ersten Hilfsobjektes zurück:

  ```
  return first.value;
  ```

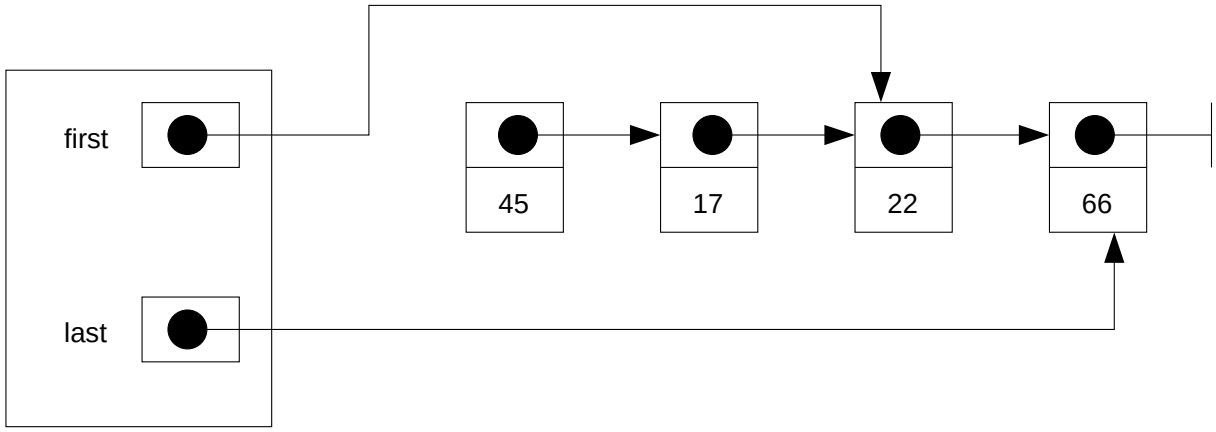
- `void dequeue()`

  Vorbedingung: Keine leere Liste

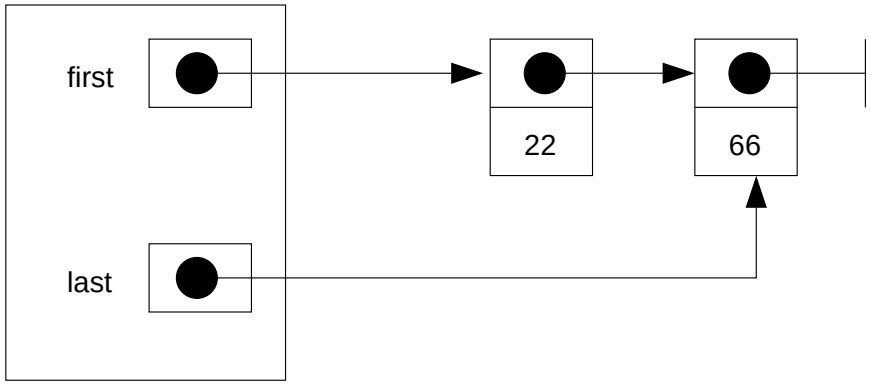
  Wir müssen nur `first` den Nachfolger von `first` zuweisen und dabei aufpassen, dass eine leere Warteschlange entstehen kann.

```
first = first.next;
if ( first == null ) {
    last = null;
}
```

  Beispiel: `dequeue(); dequeue();`



Man beachte, dass die nicht mehr referenzierten Hilfsobjekte später vom Garbage Collector gelöscht werden, so dass dann im Speicher der folgende Zustand entsteht:



Logisch besteht zwischen den beiden Zuständen kein Unterschied.

Effizienz mit verketteter Liste

- Alle Operationen können in Zeit $O(1)$ ausgeführt werden.
- Es gibt keine Kapazitätsbeschränkung mehr. Die Größe der Liste und nur noch durch den zur Verfügung stehenden Arbeitsspeicher beschränkt.

☞ dynamische Datenstruktur

Exkurs: Innere Klassen in Java

- Es gibt in Java die Möglichkeit, eine Klasse innerhalb einer anderen Klasse zu definieren. Solche Klassen heißen *innere Klassen*.
- Klassen die nicht innerhalb einer anderen Klasse definiert sind, sind sogenannte *Top-Level-Klassen*.
- Warum innere Klassen?
  - engere Bindung an die umgebende Klasse
  - Alternative zum Paketsystem
- Hier keine vollständige Behandlung von inneren Klassen, Details siehe Java Lehrbücher

Arten von inneren Klassen

- Statische innere Klassen
- Elementklassen
- Lokale Klassen
- Anonyme innere Klassen

Statische Innere Klassen

```
public class Aussen {
    ...
    static class Innen {
        ...
    }
    ...
}
```

- `Innen` ist die innere statische Klasse. Sie kann aufgebaut sein, wie eine übliche Klasse und innerhalb der inneren Klasse hat man Zugriff auf alle `static`-Elemente der äußeren Klasse.
- Statische innere Klassen sind eigenständige Klassen. Eine Instanz der inneren statische Klasse ist unabhängig von den Instanzen der äußeren Klasse.

- Erzeugung einer Instanz der inneren Klasse:
  - Innerhalb von `Aussen`:
    `new Innen(...);`
  - Außerhalb von `Aussen`:
    `new Aussen.Innen(...);`
- Die innere Klasse kann mit Modifikatoren für die Sichtbarkeit versehen werden (`public`, `private`).
- Statische innere Klassen bieten sich als Alternative zum Paketsystem an.

Elementklassen

```
public class Aussen {
    ...
    class Innen {
        ...
    }
    ...
}
```

- Eine Instanz einer Elementklasse ist immer mit einer Instanz der äußeren Klasse verbunden, d.h. zu einem `Innen`-Objekt gibt es stets genau ein `Aussen`-Objekt.
- Elementklassen stellen somit Hilfsobjekte für Instanzen der äußeren Klasse bereit, siehe verkettete Liste.
- In der inneren Klasse können keine `static`-Elemente deklariert werden.
- Von der inneren Klasse aus ist ein Zugriff auf alle Elemente der äußeren Klasse möglich, auch `private`-Elemente.

- Modifikatoren für die Sichtbarkeit der inneren Klasse sind möglich.
- Erzeugung einer Instanz der inneren Klasse:
  - geschieht typischerweise innerhalb einer Instanzmethode der äußeren Klasse. Hierdurch entsteht implizit die Zuordnung der Instanz der inneren Klasse zu `this`.
  - Ansonsten:

    ```
    Aussen aussen = new Aussen(...);
    aussen.new Innen(...);
    ```

    Zuordnung der inneren Instanz zu dem durch `aussen` referenzierten Objekt.
- Innerhalb der inneren Klasse bezeichnet `this` die Instanz von `Innen`.
- Wie kommt man an das umgebende Objekt? `Aussen.this`

Lokale Klassen

- Klassendefinitionen innerhalb von Blöcken, z.B. lokal innerhalb einer Methode.
- Keine Modifikatoren für die Sichtbarkeit erlaubt.
- Zugriff auf Methoden der äußeren Klasse, sowie auf Konstanten der umgebenden Methode oder Klasse.
- Keine große praktische Bedeutung.

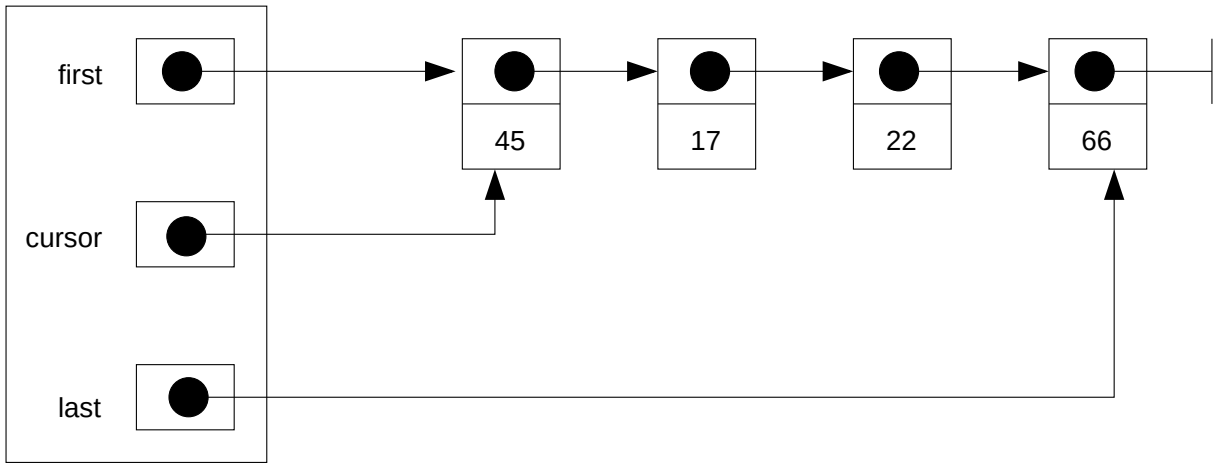
Anonyme innere Klasse

- Objekterzeugung und Klassendefinition in einem
- Praktische Bedeutung bei der lokalen Definition von Objekten für die Implementierung von einfachen Schnittstellen.
- Siehe `ActionListener`-Beispiel in Kapitel 3 (auf der Homepage)

Weitere Listenoperationen

- Für Listen gibt es viele weitere sinnvolle Operationen.
- Beispielsweise könnte man den sequentiellen schrittweisen Durchlauf durch eine Liste unterstützen.
- In der Instanzvariablen `cursor` merken wir uns die aktuelle Position in der Liste.
- `void reset()`
  Setzt den Cursor auf den Anfang der Liste.

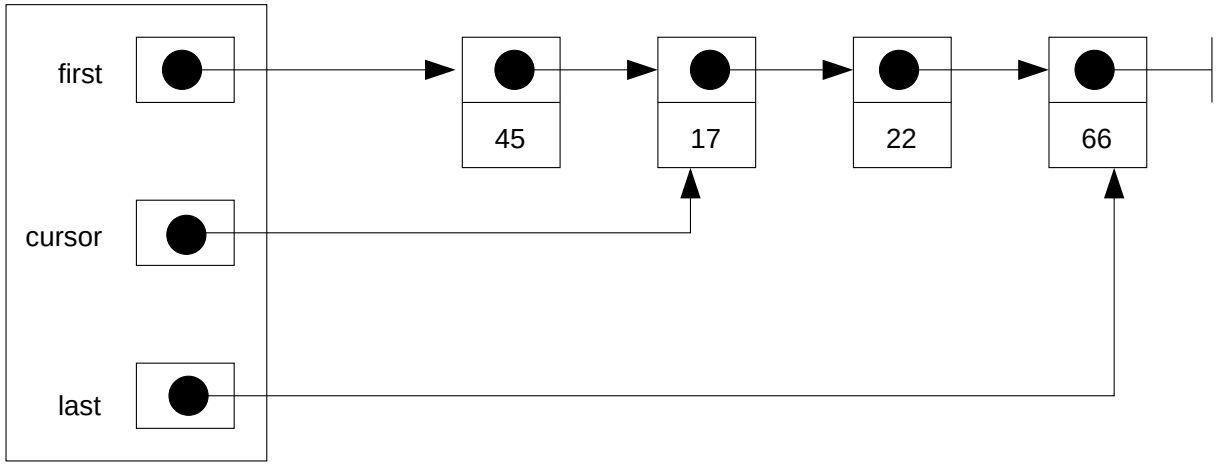
  ```
  cursor = first;
  ```



- `T next()`

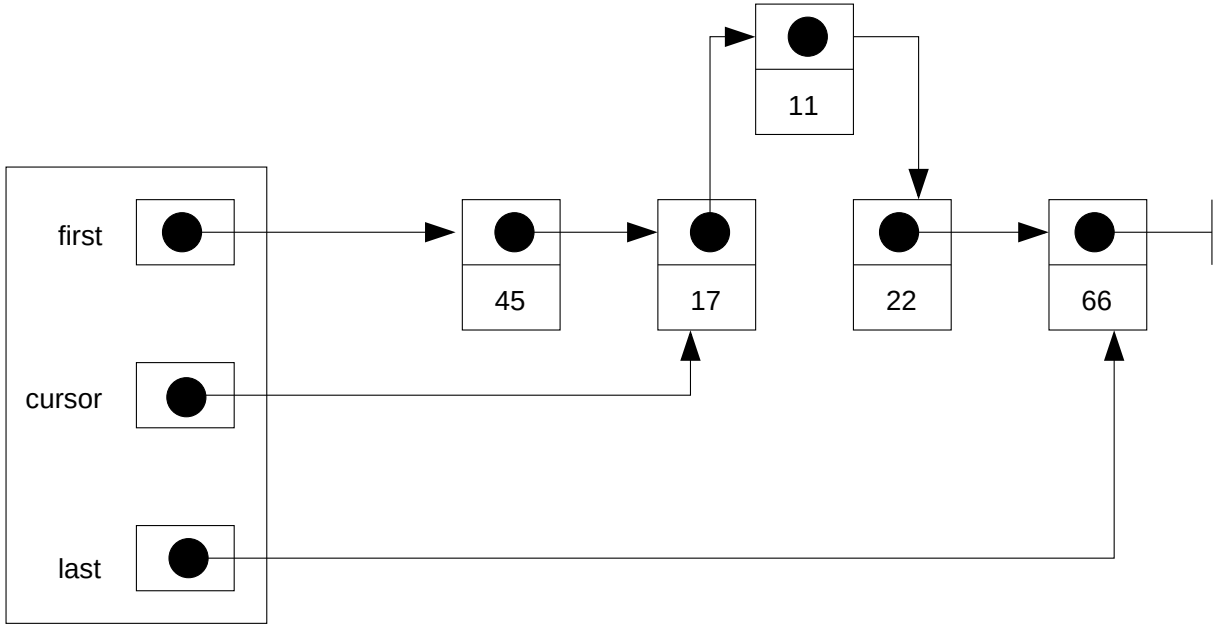
  Setzt den Cursor eine Position weiter und liefert das entsprechende Listenelement zurück.

```
cursor = cursor.next;
return cursor.value;
```



- `void insert(T elem)`
  Fügt hinter dem Cursor ein Listenelement ein.

```
Item item = new Item();
item.value = elem;
item.next = cursor.next;
cursor.next = item;
```

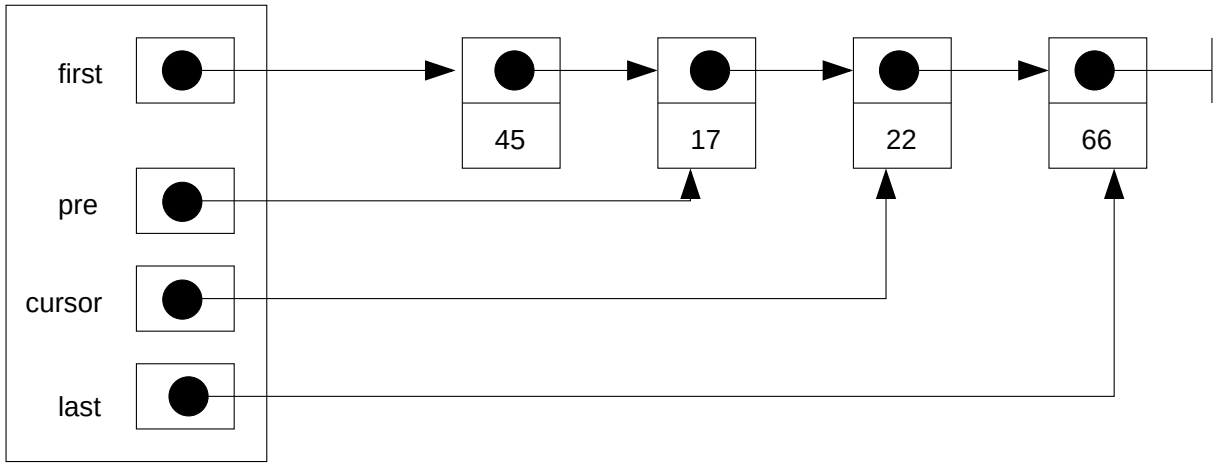


Zeitaufwand Einfügen: $O(1)$

- `void remove()`

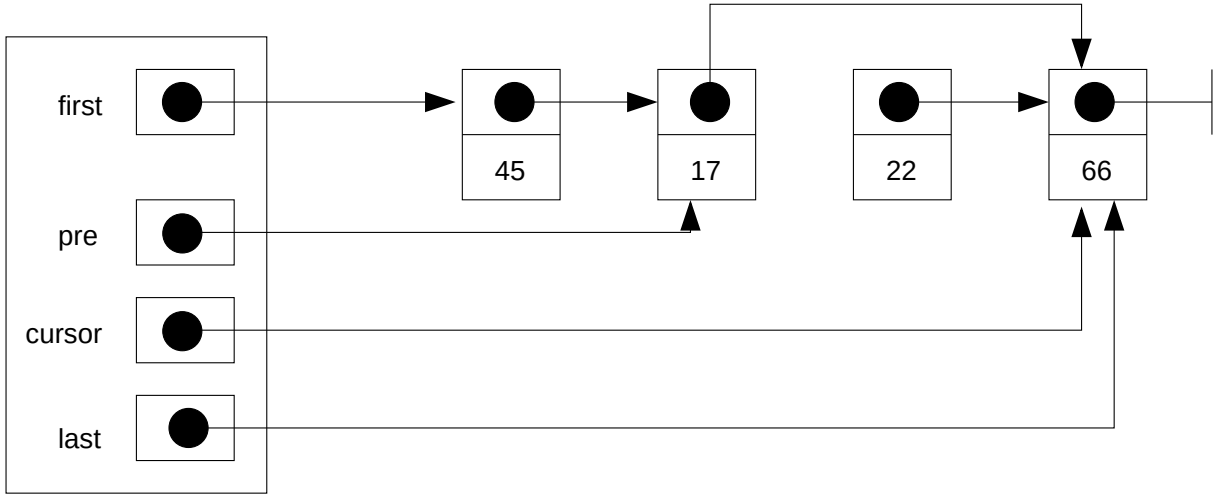
  Löscht das Cursorlement.

  Hierfür benötigt man einen Verweis auf das Element vor dem Cursor. Dies kann man durch einen Durchlauf durch die Liste ermitteln (Zeitaufwand $O(n)$) oder man sieht einen weitere Instanzvariable `pre` vor, die immer auf das Element vor dem Cursor verweist.



  `pre` muss dann natürlich bei den anderen Operationen entsprechend angepasst werden.

```
pre.next = cursor.next;
cursor = cursor.next;
```

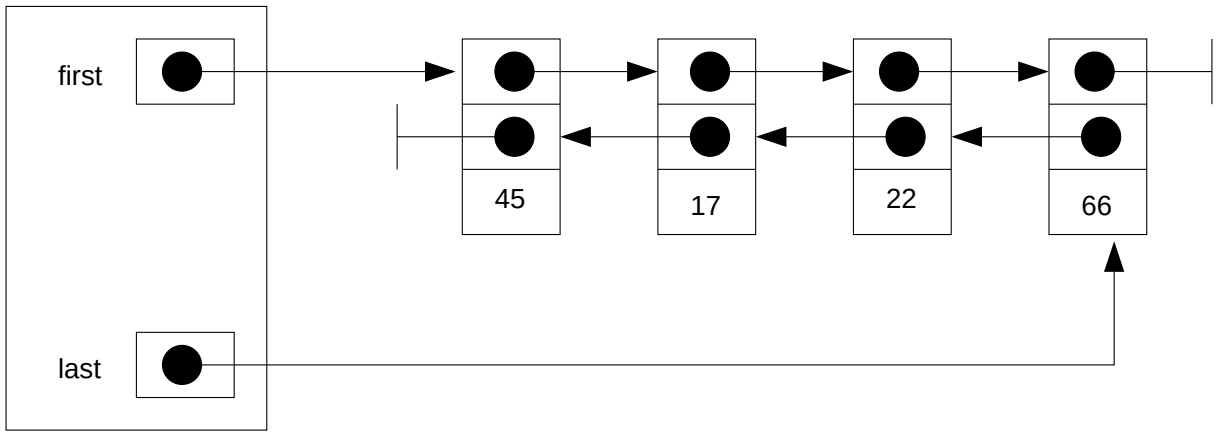


Damit ist auch Löschen in $O(1)$ möglich.

Das gelöschte Element wird später vom Garbage Collector entsorgt.

Doppelt verkettete Liste

In einer *doppelt verketteten Liste* verweist jedes Listenelement nicht nur auf den Nachfolger sondern auch auf den Vorgänger.



Dies erleichert z.B. das Löschen des aktuellen Elements (kein `pre` notwendig).

- `void remove()`

```
Item pred = cursor.prev;  // Vorgaengerelement
Item succ = cursor.next;  // Nachfolgerelement

if (pred == null) {       // Das erste Element wird geloescht
    first = succ;
}
else {
    pred.next = succ;
}

if (succ == null) {       // Das letzte Element wird geloescht
    last = pred;
}
else {
    succ.prev = pred;
}
```

- `void insert(T elem)`

  Hinter dem Cursor ein neues Element einfügen.

```
Item item = new Item();   // neues Element
Item succ = cursor.next;  // Nachfolgerelement

item.value = elem;
item.next = succ;
item.prev = cursor;

cursor.next = item;
if (succ == null) {       // Einfuegen hinter dem letzten Element
    last = item;
}
else {
    succ.prev = item;
}
```

  Analog ist natürlich auch ein Einfügen vor dem Cursor möglich.

Wahlfreier Zugriff

- Alle bisher vorgestellten Listenoperationen sind mit doppelt verketteten Listen in Zeit $O(1)$ implementierbar.
- Für die folgende Operation gilt dies nicht:

  `T get(int i)`

  Liefert das Element an der `i`-ten Stelle der Liste.

  Für eine Implementierung mit verketteten Listen beträgt der Zeitaufwand $O(n)$, eine Implementierung mit Feldern hätte dagegen den Zeitaufwand $O(1)$.
- Und wenn man nun `get()` sehr häufig benötigt?

  ☞ z.B. geschickte Größenanpassung des Feldes führt zu amortisierter Zeit $O(1)$ beim Einfügen

Dynamische Größenanpassung (1)

- Angenommen, es würde für eine Anwendung ausreichen, wenn eine Liste die Operationen `enqueue()` und `get()` unterstützt.
- Bei Implementierung mit Feld: Problem bei `enqueue()`, wenn das Feld für die Listenelemente vollständig gefüllt ist.
- Wir müssten dann ein größeres Feld erzeugen und alle Elemente in das neue Feld kopieren.

  Zeitaufwand: $O(n)$
- Angenommen, wir verdoppeln immer die Größe, wenn das Feld vollständig gefüllt ist. Welcher Gesamtaufwand entsteht dann?

Dynamische Größenanpassung (2)

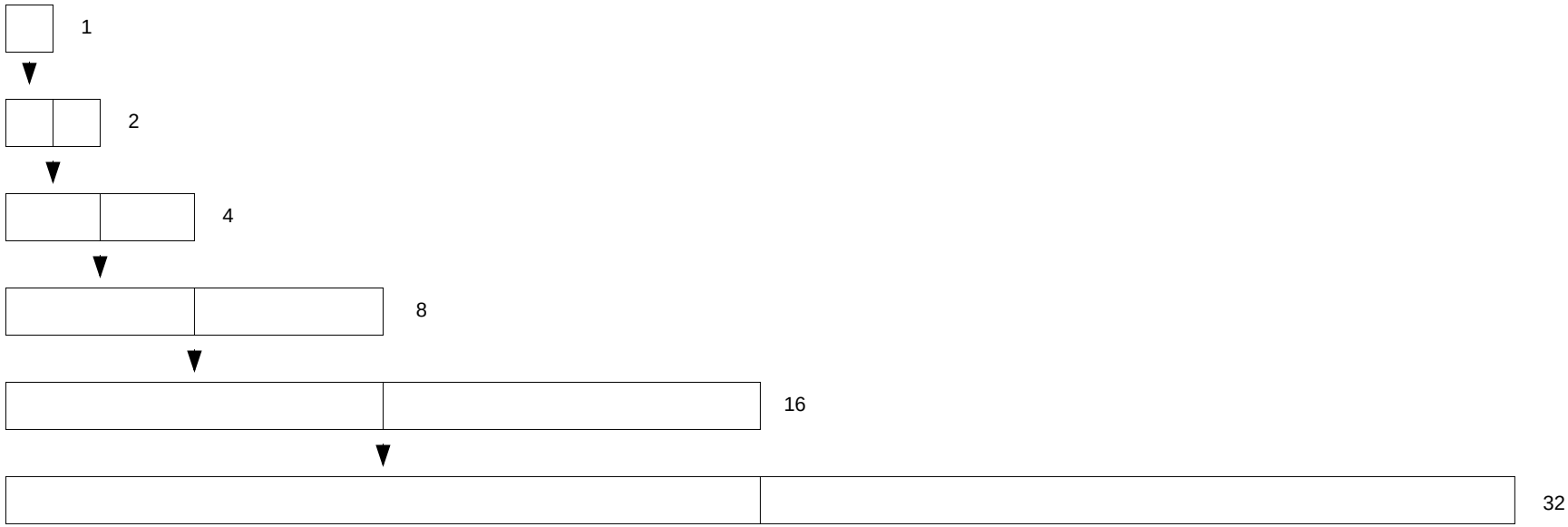
```
public class Liste<T> {
    private T[] feld = (T[]) new Object[1];
    private int size = 0;

    public void enqueue(T elem) {
        T[] neuFeld;

        if (size >= feld.length) {
            neuFeld = (T[]) new Object[2*feld.length];
            for (int i=0 ; i<feld.length ; i++) {
                neuFeld[i] = feld[i];
            }
            feld = neuFeld;
        }

        feld[size] = elem;
        size++;
    }
    ...
}
```

Dynamische Größenanpassung (3)



☞ Der Gesamtaufwand für `enqueue()` entspricht der Gesamtlänge aller erzeugten Felder.

Dynamische Größenanpassung (4)

Es sei n die Anzahl der eingefügten Elemente (Länge der Liste). O.B.d.A. sei n eine Zweierpotzen, also $n = 2^k$. Dann gilt für die Gesamtlänge:

$$\begin{aligned} \text{Gesamtlänge} &= n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^k} \\ &= n\,(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k}) \\ &= n \sum_{i=0}^{k} \left(\frac{1}{2}\right)^k \end{aligned}$$

Mit der Formel

$$\sum_{i=0}^{k} x^k = \frac{1 - x^{k+1}}{1 - x}$$

können wir die Gesamtlänge abschätzen:

$$\begin{aligned}
\text{Gesamtlänge} &= n \sum_{i=0}^{k} \left(\frac{1}{2}\right)^k \\
&= n \frac{1 - \left(\frac{1}{2}\right)^{k+1}}{\frac{1}{2}} \\
&= 2n \left(1 - \left(\frac{1}{2}\right)^{k+1}\right) \\
&\leq 2n
\end{aligned}$$

Fazit:

- ☞ Der Gesamtzeitaufwand für das Einfügen von n Elementen beträgt $O(n)$.
- ☞ Damit benötigt eine einzelne Einfügeoperation im Mittel Zeit $O(1)$.

Amortisierte Laufzeitanalyse

- Die durchgeführte Analyse ist eine sogenannte *amortisierte Laufzeitanalyse*.
- Bei der amortisierten Laufzeitanalyse betrachtet man die Kosten (Zeitaufwand) von Folgen von Operationen, nicht nur einer einzelnen Operation.
- Dividiert man den Zeitaufwand durch die Anzahl der Operationen, erhält man den durchschnittlichen Zeitaufwand pro Operation (Aggregat-Methode).
- Man beachte:
  - Der durchschnittliche Zeitaufwand für ein einzelnes `enqueue()` ist nur $O(1)$, obwohl
  - der Zeitaufwand für ein einzelnes `enqueue()` im Worst-Case $O(n)$ beträgt.
  - Der Zeitaufwand für n-faches `enqueue()` beträgt ebenfalls nur $O(n)$.

Listen im Java-API

- Generische Klassen und Schnittstellen für Listen finden sich im Paket `java.util`.
- Generische Schnittstelle für Listen: `List<T>`
- Feldbasierte Implementierung: `ArrayList<T>`

  `ArrayList` nutzt eine dynamische Größenerweiterung des Feldes. Zitat aus der API-Dokumentation:

  > Resizable-array implementation of the List interface. (...) The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time.

- Doppelt verkettete Liste: `LinkedList<T>`

  Ein kleiner Trick senkt etwas den Aufwand bei `get()`, hat aber keinen Einfluss auf die Größenordnung:

  > Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

6. Bäume

Lernziele:

- Definition und Eigenschaften binärer Bäume kennen,
- Traversierungsalgorithmen für binäre Bäume implementieren können,
- die Bedeutung von Suchbäumen für die effiziente Implementierung verschiedenen abstrakter Datentypen erläutern können und
- Ausgleichstechniken für binäre Bäume verstehen und einsetzen können.

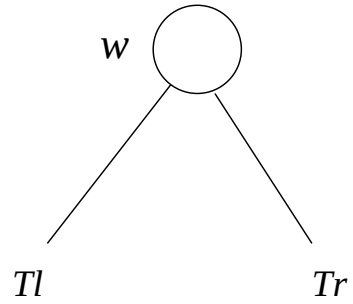
☞ Wir beschränken uns in diesem Kapitel auf Binärbäume.

☞ Datenstrukturen für die effiziente Suche im Hauptspeicher.

Binärbäume

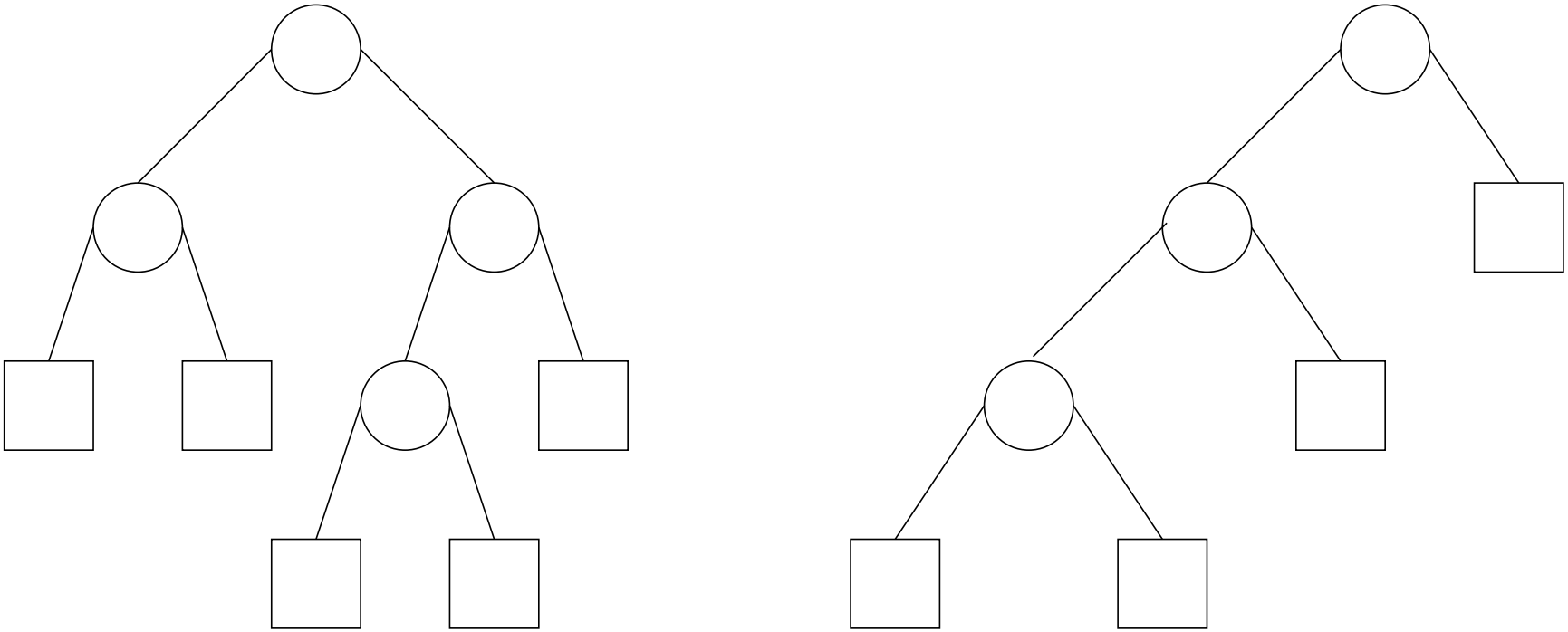
Definition 6.1. Ein *Binärbaum* ist wie folgt definiert:

- Der *leere Baum*, bezeichnet durch $\square$, ist ein Binärbaum.
- Wenn T_l und T_r Binärbäume sind, dann ist auch



ein Binärbaum. Der Knoten w ist die *Wurzel* des Binärbaums, T_l der linke und T_r der rechte Unterbaum von w.

Beispiel 6.1. Zwei binäre Bäume:



Knotenanzahl binärer Bäume

Definition 6.2. Die Knoten ◯ bezeichnet man auch als *innere Knoten*, die leeren Bäume □ als *externe Knoten*.

Lemma 6.1. Ein binärer Baum mit n inneren Knoten hat genau $n + 1$ externe Knoten.

Beweis: Vollständige Induktion über die Anzahl der inneren Knoten

Induktionsanfang: $n = 0$

z.Z.: "Ein binärer Baum mit 0 inneren Knoten hat genau einen externen Knoten."

Es gibt nur einen binären Baum, der keinen inneren Knoten hat: der leere Baum. Der leere Baum hat genau einen externen Knoten. Also gilt die Aussage für $n = 0$.

Induktionsschritt: $n \to n + 1$

z.Z. "Wenn die Aussage für alle binären Bäume mit höchsten n inneren Knoten gilt, dann gilt sie auch für alle Bäume mit $n + 1$ inneren Knoten."

Induktionsvoraussetzung (Wenn-Teil): Für $0 \leq m \leq n$ gilt: Ein binärer Baum mit m inneren Knoten hat $m + 1$ externe Knoten.

Induktionsbehauptung (Dann-Teil): Ein beliebiger binärer Baum mit $n + 1$ inneren Knoten hat $n + 2$ externe Knoten.

Es sei T ein beliebiger binärer Baum mit $n + 1$ inneren Knoten. Es sei w die Wurzel von T, T_l sei der linke Unterbaum von w und T_r der rechte Unterbaum.

n_l sei die Anzahl der inneren Knoten von T_l, n_r sei die Anzahl der inneren Knoten von T_r.

Es gilt: $n_l + n_r = n$, denn T_l und T_r enthalten alle der $n + 1$ inneren Knoten von T bis auf die Wurzel.

Aus der Induktionsvoraussetzung folgt: T_l enthält $n_l + 1$ externe Knoten, T_r enthält $n_r + 1$ externe Knoten.

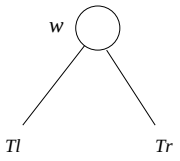
Alle externen Knoten von T sind in T_l oder T_r. Also hat T

$$(n_l + 1) + (n_r + 1) = (n_l + n_r) + 2 = n + 2$$

externe Knoten.

Höhe binärer Bäume

Definition 6.3. Die Höhe $h(T)$ eines binären Baums T ist wie folgt definiert:

- Für den leeren Baum gilt: $h(\square) = 0$
- Für einen nicht-leeren Baum mit $T =$ 

gilt:

$$h(T) = 1 + \max\{h(T_l), h(T_r)\}$$

Lemma 6.2.

(a) Ein binärer Baum der Höhe h hat höchstens 2^h externe und höchstens $2^h - 1$ innere Knoten.

(b) Für jeden Binärbaum T mit n inneren Knoten gilt

$$h(T) \geq \log_2(n + 1)$$

Beweis: (a) Vollständige Induktion über die Höhe h.

$h = 0$: Nur der leere Baum hat die Höhe 0 und er hat $2^0 = 1$ externe Knoten.

$h \to h + 1$: Es sei T ein binärer Baum mit Höhe $h + 1$, T_l sei der linke und T_r sei der rechte Unterbaum der Wurzel von T.

Die Höhe von T_l und T_r ist höchsten h. Nach Induktionsvoraussetzung enthalten T_l und T_r jeweils höchstens 2^h externe Knoten.

T enthält somit höchstens $2^h + 2^h = 2 \cdot 2^h = 2^{h+1}$ externe Knoten.

Die Formel für die Anzahl der internen Knoten folgt aus Lemma 6.1.

(b) Aus der Formel für die inneren Knoten folgt:

$$\begin{aligned} n \leq 2^{h(T)} - 1 \quad &\Leftrightarrow \quad n + 1 \leq 2^{h(T)} \\ &\Leftrightarrow \quad \log_2(n + 1) \leq h(T) \end{aligned}$$

Traversierungsalgorithmen

- Sehr häufig müssen alle Knoten eines Baumes aufgesucht und bearbeitet werden.
- Verfahren hierfür werden auch *Traversierungs-* oder *Durchlaufalgorithmen* genannt.
- Die folgenden Traversierungsalgorithmen für Binärbäume sind von besonderer Bedeutung.
- Diese Verfahren basieren direkt auf der rekursiven Definition eines Binärbaums.

Preorder-Traversierung

Algorithmus 6.1. [Preorder-Traversierung] Bei der *Preorder-Traversierung* wird ein Binärbaum T wie folgt durchlaufen:

- Falls T leer ist, tue nichts,
- ansonsten führe die folgenden Schritte in der genannten Reihenfolge aus:
  - Behandle die Wurzel w von T,
  - durchlaufe T_l in Preorder und
  - durchlaufe T_r in Preorder.

Inorder-Traversierung

Algorithmus 6.2. [Inorder-Traversierung] Bei der *Inorder-Traversierung* wird ein Binärbaum T wie folgt durchlaufen:

- Falls T leer ist, tue nichts,
- ansonsten führe die folgenden Schritte in der genannten Reihenfolge aus:
  - Durchlaufe den linken Unterbaum T_l in Inorder,
  - behandle die Wurzel w von T und
  - durchlaufe den rechten Unterbaum T_r in Inorder.

Postorder-Traversierung

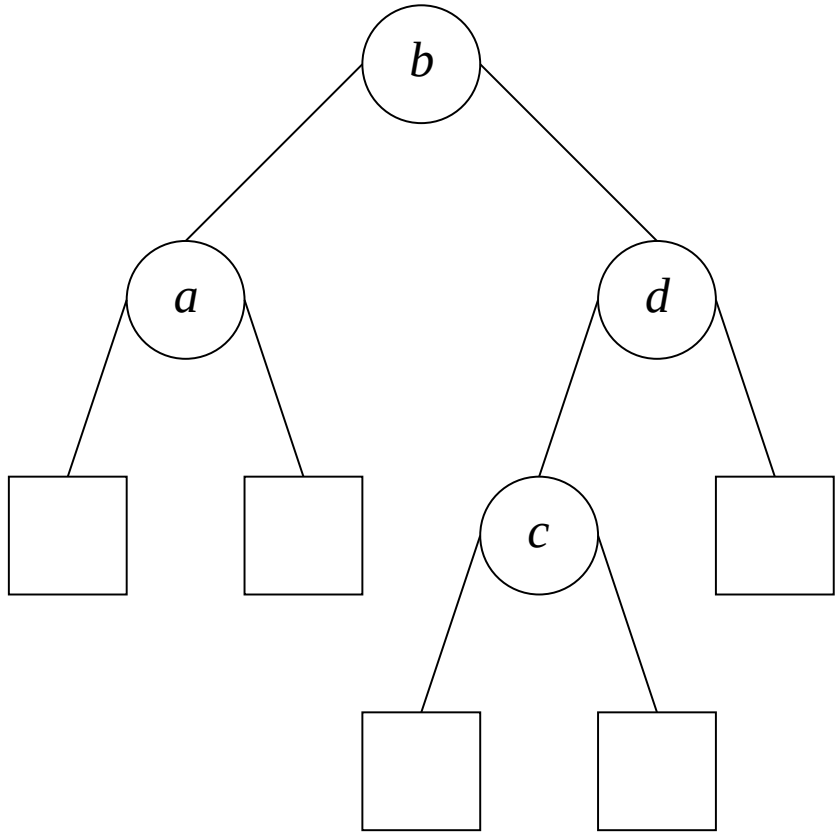
Algorithmus 6.3. [Postorder-Traversierung] Bei der *Postorder-Traversierung* wird ein Binärbaum T wie folgt durchlaufen:

- Falls T leer ist, tue nichts,
- ansonsten führe die folgenden Schritte in der genannten Reihenfolge aus:
  - Durchlaufe den linken Unterbaum T_l in Postorder,
  - durchlaufe den rechten Unterbaum T_r in Postorder und
  - behandle die Wurzel w von T.

Beispiel Traversierungsalgorithmen

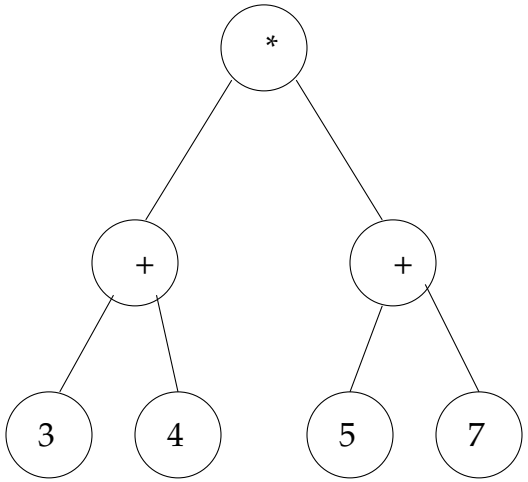
Beispiel 6.2.

- Preorder: *b a d c*
- Inorder: *a b c d*
- Postorder: *a c d b*



Repräsentation arithmetischer Ausdrücke

Beispiel 6.3. Repräsentation des arithmetischen Ausdrucks $(3+4)*(5+7)$ als Binärbaum:



Inorder (nicht eindeutig): $3 + 4 * 5 + 7$

Postorder (UPN, eindeutig): $3\,4 + 5\,7 + *$

Bemerkungen zu den Traversierungsalgorithmen

- Preorder bzw. Postorder liefern eine injektive Abbildung von binären Strukturbäumen für arithmetische Ausdrücke in Strings.
- Ergebnis von Preorder bzw. Postorder ist dann die sog. klammerfreie Präfix- bzw. Postfix-Notation (Umgekehrte Polnische Notation, UPN).
- Inorder dagegen ist nicht injektiv. Die Infixnotation wird erst durch Klammern eindeutig.

Abstrakter Datentyp Menge

Es sei T ein beliebiger Datentyp. Eine abstrakter Datentyp *Menge* (von Elementen aus T) unterstützt die folgenden Operationen:

- `void insert(T e)`
  Fügt ein Element e vom Typ T in eine Menge M ein.
  $M := M \cup \{e\}$
- `void remove(T e)`
  Löscht das Element e aus einer Menge M.
  $M := M \setminus \{e\}$
- `boolean contains(T e)`
  Prüft, on e sich in einer Menge M befindet.
  Ist $e \in M$ wahr?

Abstrakter Datentyp Wörterbuch

Definition 6.4. Es seien S und T beliebige Datentypen. Ein *Wörterbuch (Dictionary, Map)* ist eine Datenstruktur, die die folgenden Operationen unterstützt:

- `void put(S k, T v)`
  Hinterlegt im Wörterbuch zum Schlüssel k den assoziierten Wert v.
- `T get(S k)`
  Liefert den im Wörterbuch hinterlegten assoziierten Wert für Schlüssel k.
- `void remove(S k)`
  Löscht aus dem Wörterbuch den Schlüssel k und seinen assoziierten Wert.

☞ Ein Wörterbuch entspricht einer partiellen Funktion $f : S \longrightarrow T$.

Effizienz bekannter Implementierungsansätze

Menge bzw. Wörterbuch als verkettete Liste:

- `insert()` bzw. `put()` hat Zeitaufwand $O(1)$
- `remove()`, sowie `contains()` bzw. `get()` haben den Zeitaufwand $O(n)$ (für n Elemente), da die Liste sequentiell durchsucht werden muss.

Menge bzw. Wörterbuch als sortiertes Feld:

- `contains()` bzw. `get()` haben Zeitaufwand $O(\log n)$ bei Implementierung mittels binärer Suche (siehe "Einführung in die Programmierung", Folien 415 bis 419).
- Alle anderen Operationen haben i.A. den Zeitaufwand $O(n)$.

☞ Implementierung möglich mit allen Operationen besser als $O(n)$?

Suchbäume

Definition 6.5. Ein binärer Suchbaum T ist ein binärer Baum mit den folgenden Eigenschaften:

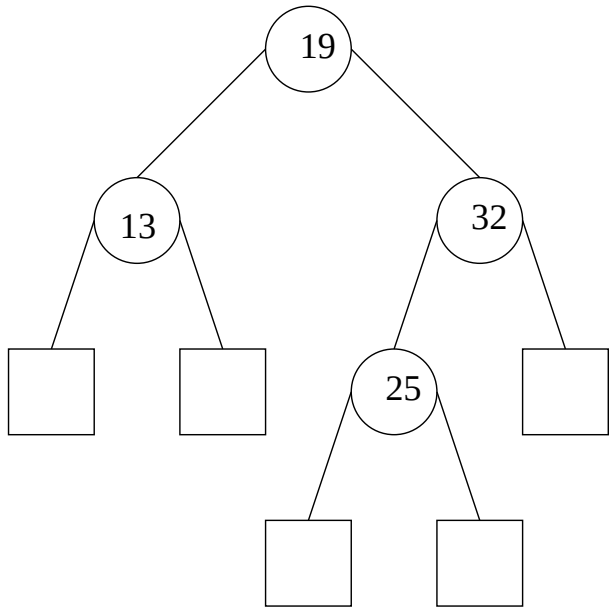
- Jeder innere Knoten w enthält bzw. speichert einen Wert $val(w)$.
- Für alle innere Knoten w und für alle Werte v im linken Teilbaum zu einem Knoten w gilt:
$$v < val(w)$$
- Für alle innere Knoten w und für alle Werte v im rechten Teilbaum zu einem Knoten w gilt:
$$v > val(w)$$

Im Folgenden sagen wir kurz Suchbaum statt binärer Suchbaum.

Beispiel Suchbaum

- Der Baum rechts hat die Eigenschaften eines Suchbaums.
- Die in den inneren Knoten enthaltenen Werte sind bspw. die Elemente einer Menge oder die Schlüssel eines Wörterbuchs.
- Die externen Knoten entsprechen Intervallen ohne Wert bzw. Schlüssel.
- Durch die Verzweigung muss nicht die ganze Menge nach einem Schlüssel durchsucht werden.
- Im Idealfall Suche (`get()` bzw. `contains()` in $O(\log n)$) möglich.

Im Folgenden verzichten wir auf die Darstellung der externen Knoten.



Suchbaum in Java

```
public class SearchTree<T extends Comparable<T>> {

    private class Node {
        T    value;     // Wert am Knoten
        Node left;      // linker Unterbaum
        Node right;     // rechter Unterbaum
    }
    ...
    private Node root;   // Wurzelknoten
}
```

Suchen

Am Beispiel der Methode `boolean contains(T elem)` für Mengen:

```
Node node = root;
int  diff;

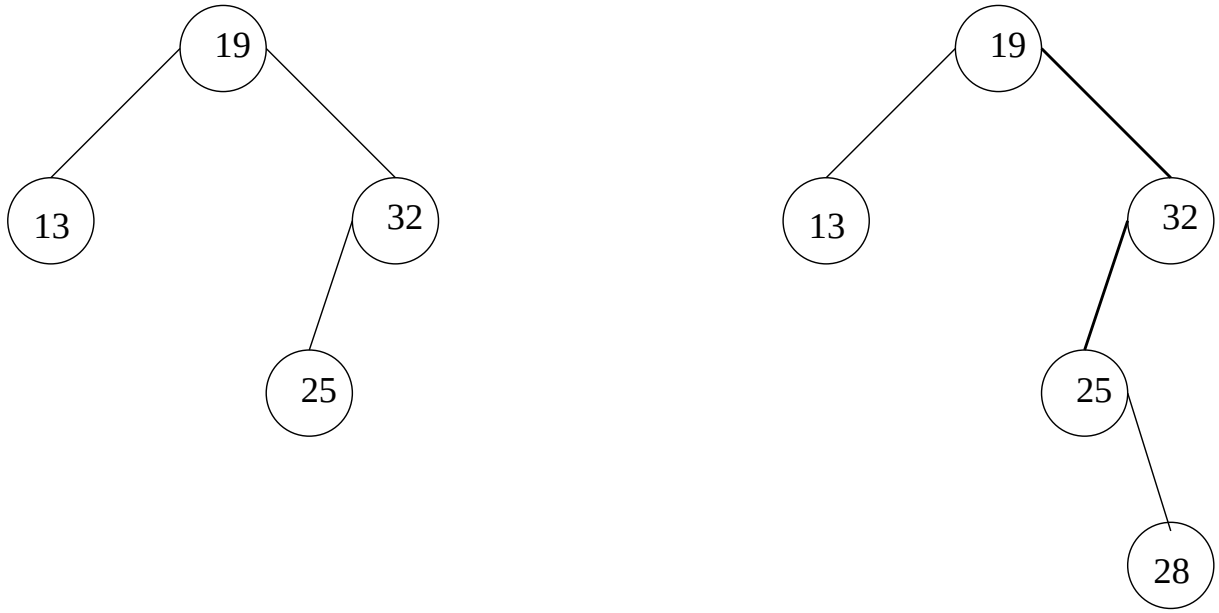
while (node != null) {
    diff = node.value.compareTo(elem);
    if (diff == 0) {              // Element gefunden
        return true;
    }
    else if (diff>0) {            // node.value > elem,
        node = node.left;         // dann liegt elem im linken Unterbaum
    }
    else {                        // node.value < elem,
        node = node.right;        // dann liegt elem im rechten Unterbaum
    }
}
return false;
```

Einfügen: Logik

- Wir gehen davon aus, dass wir einen neuen Wert $\mathtt{x}$ einfügen möchten, der noch nicht im Suchbaum enthalten ist.
- Wie beim Suchen muss man den Baum durchlaufen, bis man an einem Knoten w nicht mehr weiterkommt (leerer Unterbaum).
- Wir legen einen neuen Knoten $\mathtt{r}$ mit $\mathtt{val}(\mathtt{r}) = \mathtt{x}$ an und setzen $\mathtt{r}$ als Unterbaum von w ein.

Einfügen: Veranschaulichung

Einfügen von 28:



Einfügen: Java

Methode `void insert(T elem)`:

```
Node node = root;    // Hilfsknoten fuer Suche und Einfuegen
Node father = null;  // Vater von node bei Suche
int diff = 0;

while (node!=null) {
    father = node;
    diff = node.value.compareTo(elem);
    if (diff==0) {   // elem schon in Suchbaum vorhanden
        return;
    }
    else if (diff>0) {
        node = node.left;
    }
    else {
        node = node.right;
    }
```

```
}

// neuen Knoten anlegen und initialisieren
node = new Node();
node.left = node.right = null;
node.value = elem;

if (father==null) {     // Baum war bisher leer
    root = node;        // dann ist node jetzt die Wurzel
}
else {                  // ansonsten wird node Unterbaum von father
    if (diff>0) {       // father.value > elem => neuer linker Unterbaum
        father.left = node;
    }
    else {              // father.value < elem => neuer rechter Unterbaum
        father.right = node;
    }
}
```

Löschen: Logik

- Wir möchten den Wert x löschen.
- Suche Knoten w mit zu löschendem Wert x. Es sei T_l der linke und T_r der rechte Unterbaum von w.
- Möglichkeiten:

(a) w ist ein Blatt (d.h. T_l und T_r sind leer):
Dann lösche w.

(b 1) T_l ist nicht leer, T_r ist leer:
Dann ersetze w durch die Wurzel von T_l.

(b 2) T_l ist leer, T_r ist nicht leer:
Dann ersetze w durch die Wurzel von T_r.

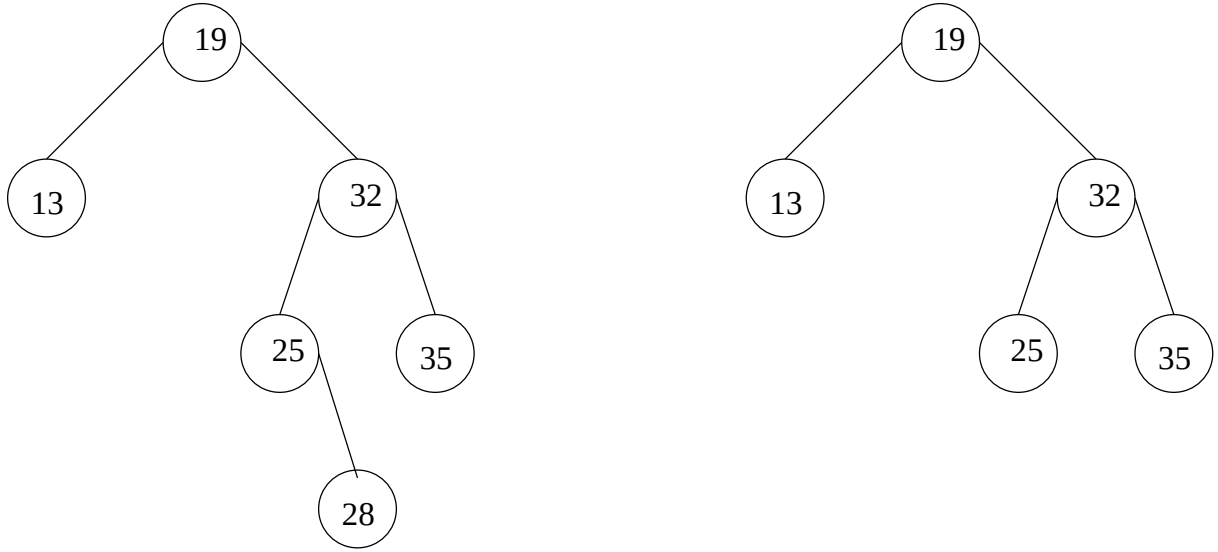
(c) Weder T_l noch T_r ist leer:
Dann bestimme den Knoten w' mit dem größten Wert x' in T_l. Setze den Wert x' in w ein und Lösche w'.

Löschen: Bemerkungen

- Die Fälle (b 1) und (b 2) sind symmetrisch.
- In Fall (c) kann man statt des größten Wertes in T_l auch den kleinsten Wert in T_r bestimmen.
- In Fall (c) kann für den zu löschenden Knoten w' nicht wieder der Fall (c) vorliegen.

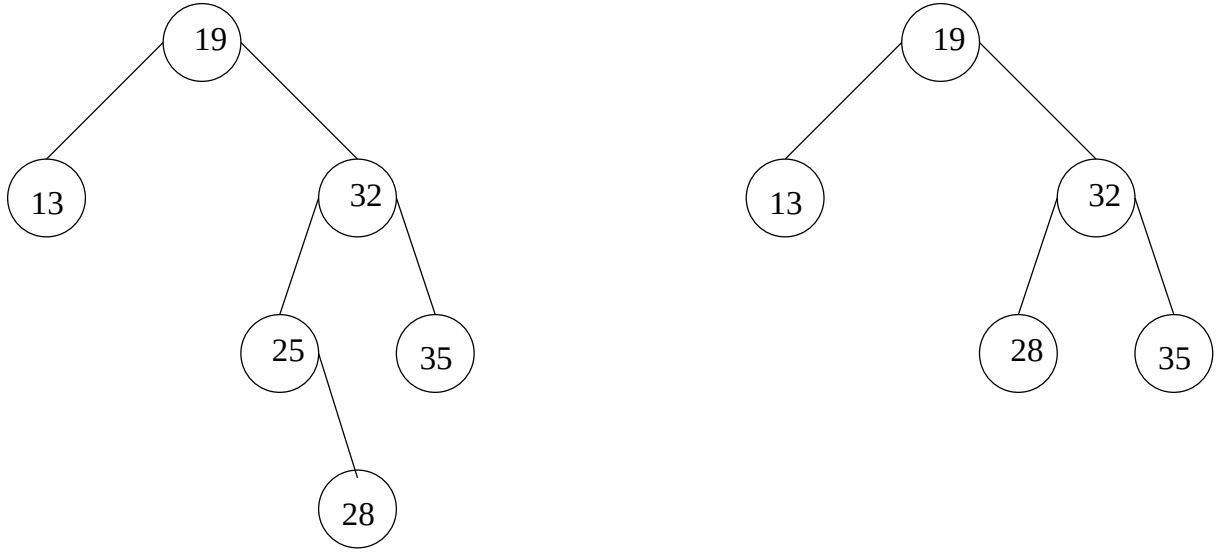
Löschen: Veranschaulichung (1)

Löschen von 28 (Fall (a)):



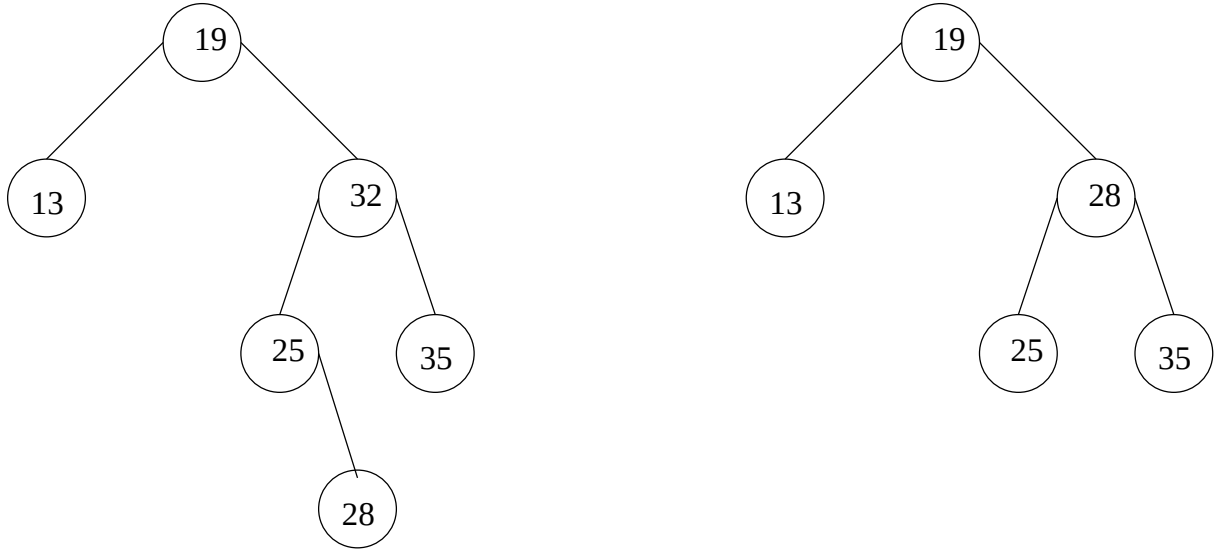
Löschen: Veranschaulichung (2)

Löschen von 25 (Fall (b 2)):



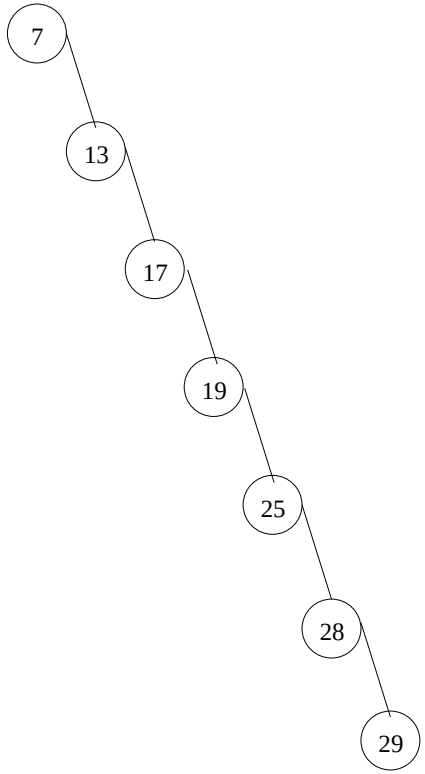
Löschen: Veranschaulichung (3)

Löschen von 32 (Fall (c)):

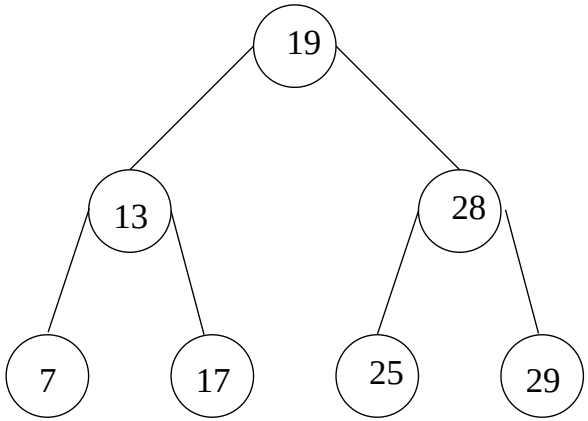


Ausgeglichener und unausgeglichener Baum

Unausgeglichener Baum:



Ausgeglichener Baum:



Effizienz

- Der Aufwand jeder Operation ist proportional zur Höhe des Baumes.
- Beim Einfügen und Löschen muss stets ein Pfad von der Wurzel zu einem Blatt durchlaufen werden.
- Beim Suchen muss im Worst-Case ein Pfad von der Wurzel zu einem Blatt durchlaufen werden.
- Wenn für jeden Knoten eines Baums gilt, dass sich die Höhe des linken und des rechten Teilbaums nicht oder nur gering unterscheiden, dann nennen wir den Baum *ausgeglichen*. Ein Baum, der nicht ausgeglichen ist, wird auch als *entartet* bezeichnet.
- Die Höhe von ausgeglichenen Bäumen ist $O(\log n)$.
- Konsequenz: Für ausgeglichene Bäume können alle Operationen in Zeit $O(\log n)$ ausgeführt werden.

AVL-Bäume

Ziele:

- Logarithmischer Zeitaufwand für die Suche (auch im Worst-Case). Hierzu sind nicht-entartete Bäume notwendig.
- Logarithmischer Zeitaufwand für Änderungen.

AVL-Bäume: Definition

- AVL-Bäume gehen zurück auf die russischen Mathematiker G. M. Adelson-Velsky und J. M. Landis (1962).

Definition 6.6. Ein Binärbaum heißt *AVL-Baum*, wenn für jeden Knoten w das sogenannte *AVL-Kriterium* gilt:

$$|h(T_l(w)) - h(T_r(w))| \leq 1$$

d.h. an jedem Knoten w unterscheiden sich die Höhe des linken Unterbaums $T_l(w)$ und die Höhe des rechten Unterbaums $T_r(w)$ von w höchstens um eins.

AVL-Bäume: Maximale Höhe

Satz 6.3. *Es sei* T *ein AVL-Baum mit* n *Knoten. Dann gilt:*

$$h(T) < 1.5 \log_2(n + 1.5)$$

Beweis:

- Es sei n_h die minimale Anzahl an Knoten in einem AVL-Baum der Höhe h. Dann gilt:
  $n_0 = 0, n_1 = 1, n_2 = 2$
  $n_h = n_{h-1} + n_{h-2} + 1$
- Wir betrachten: $\tilde{n}_h := n_h + 1$. Dann gilt:
  - $\tilde{n}_h = \tilde{n}_{h-1} + \tilde{n}_{h-2}$
  - $\tilde{n}_0 = 1, \tilde{n}_1 = 2$
- F_h sei die h-te Fibonacci-Zahl. Dann gilt: $F_h = \tilde{n}_{h-2}$

- Es gilt:

$$F_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right)$$

- Weiterhin gilt für die Fibonacci-Zahlen die folgende Ungleichung:

$$F_h + 1/2 > \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h$$

- Wir definieren: $\alpha := \frac{1+\sqrt{5}}{2}$. Mit $n_h = \tilde{n}_h - 1 = F_{h+2} - 1$ folgt

$$n_h + \frac{3}{2} \geq \frac{1}{\sqrt{5}} \alpha^{h+2}$$

Daraus folgt:

$$\alpha^{h+2} \leq \sqrt{5} \left(n_h + \frac{3}{2} \right) \quad \Leftrightarrow \quad h + 2 \leq \frac{\log_2(\sqrt{5}(n_h + \frac{3}{2}))}{\log_2 \alpha}$$

$$\Leftrightarrow \quad h \leq \frac{\log_2(n_h + \frac{3}{2})}{\log_2 \alpha} + \frac{\log_2 \sqrt{5}}{\log_2 \alpha} - 2$$

$$\Leftrightarrow \quad h \leq 1.4405 \log_2(n_h + \frac{3}{2}) + c$$

mit

$$1.4405 = 1/\log_2\alpha \text{ und } c = \frac{\log_2 \sqrt{5}}{\log_2 \alpha} - 2$$

Bemerkungen:

- Die Höhe eines AVL-Baums wächst auch im ungünstigsten Fall nur mit dem Logarithmus der Gesamtknotenanzahl, also sehr langsam.
- Ein AVL-Baum mit $n = 10^6$ Einträgen hat auch im ungünstigsten Fall eine Höhe ≤ 30.
- Das langsame Anwachsen der Baumhöhe garantiert, dass die Operationen `contains()` bzw. `get()` stets effizient ausgeführt werden können.
- Die entscheidende Frage ist, ob auch die Operationen `insert()` und `remove()` mit Hilfe eines AVL-Baums effizient ausgeführt werden können.

Einfügen und Rebalancierung (1)

- Wir wollen AVL-Bäume als Suchbäume nutzen.
- Somit können wir ein neues Element in einen AVL-Baum wie in einen gewöhnlichen Suchbaum einfügen.
- Der entstehende Suchbaum braucht aber kein AVL-Baum mehr zu sein.
- Fazit: Nach dem Einfügen eines Elements kann das AVL-Kriterium an verschiedenen Knoten verletzt sein.

Einfügen und Rebalancierung (2)

☞ Diejenigen Knoten, für die nach dem Einfügen das AVL-Kriterium verletzt ist, können nur auf dem Pfad liegen, der von der Wurzel zu dem neuen Knoten führt.

☞ Die sind höchstens h und somit höchstens $O(\log n)$ viele Knoten.

Rebalancierung:

- Bei den Knoten, an denen das AVL-Kriterium verletzt ist, wird der AVL-Baum durch Rebalancierung so transformiert, dass
  (a) wieder überall das AVL-Kriterium erfüllt ist und
  (b) der Baum weiterhin sortiert ist, also die Bedingungen für einen Suchbaum erfüllt.

Einfügen und Rebalancierung (3)

- Im folgenden wird ein AVL-Unterbaum $\mathfrak{U}$ betrachtet, in den eingefügt wird.
- $\mathfrak{U}$ besteht aus der Wurzel w und den Unterbäumen $\mathfrak{A}$ (links) und $\mathfrak{B}$ (rechts).

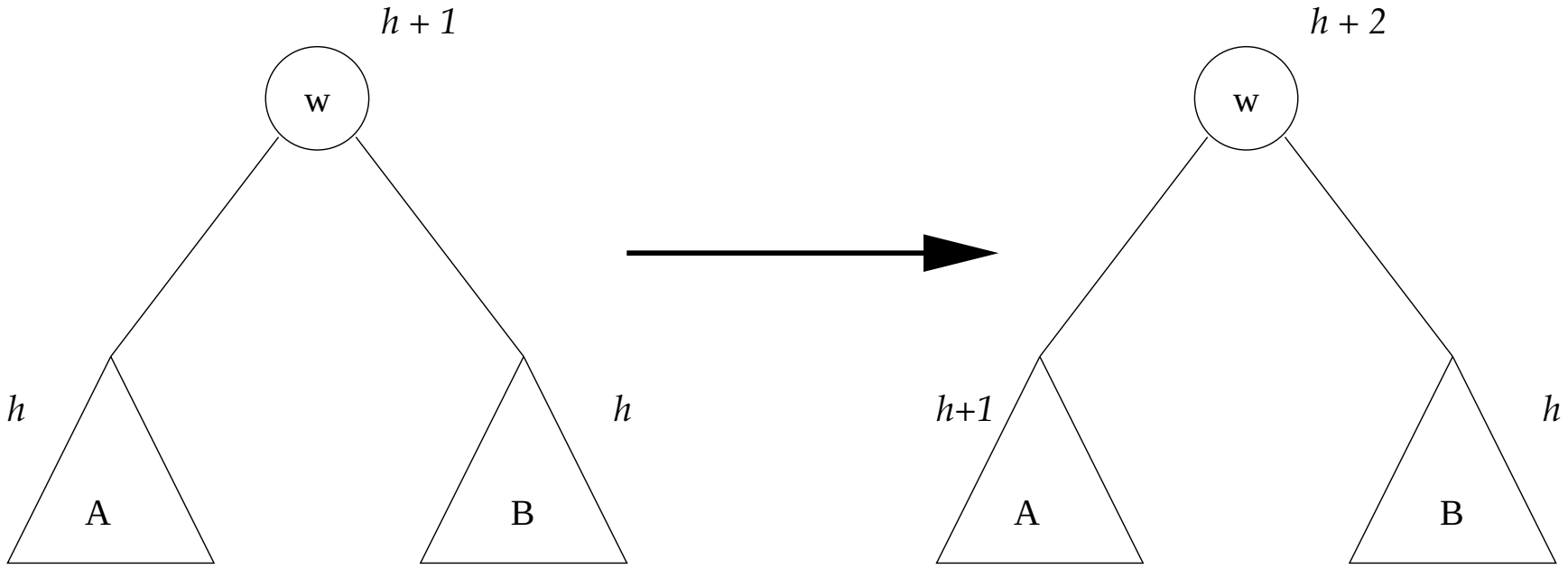
Insgesamt kann man beim Einfügen die folgenden Situationen unterscheiden:

Fall 1:

- $\mathfrak{A}$ und $\mathfrak{B}$ haben die gleiche Höhe $\mathfrak{h}$.
- Durch Einfügen in $\mathfrak{A}$ ändert sich die Höhe von $\mathfrak{A}$.

$\Rightarrow$ Das AVL-Kriterium bleibt am Knoten w erfüllt.

Veranschaulichung:



Bemerkungen:

- Der enstandene Unterbaum ist wieder ein AVL-Baum, d.h. an w ist keine Rebalancierung notwendig.
- Da sich aber die Höhe des Unterbaums $\mathfrak{U}$ geändert hat, kann sich diese Höhenänderung zur Wurzel hin fortpflanzen.
- An einem Vorfahr von w könnte demnach das AVL-Kriterium verletzt sein. Dort liegt dann eine der Situationen von Fall 3 vor (s.u.).

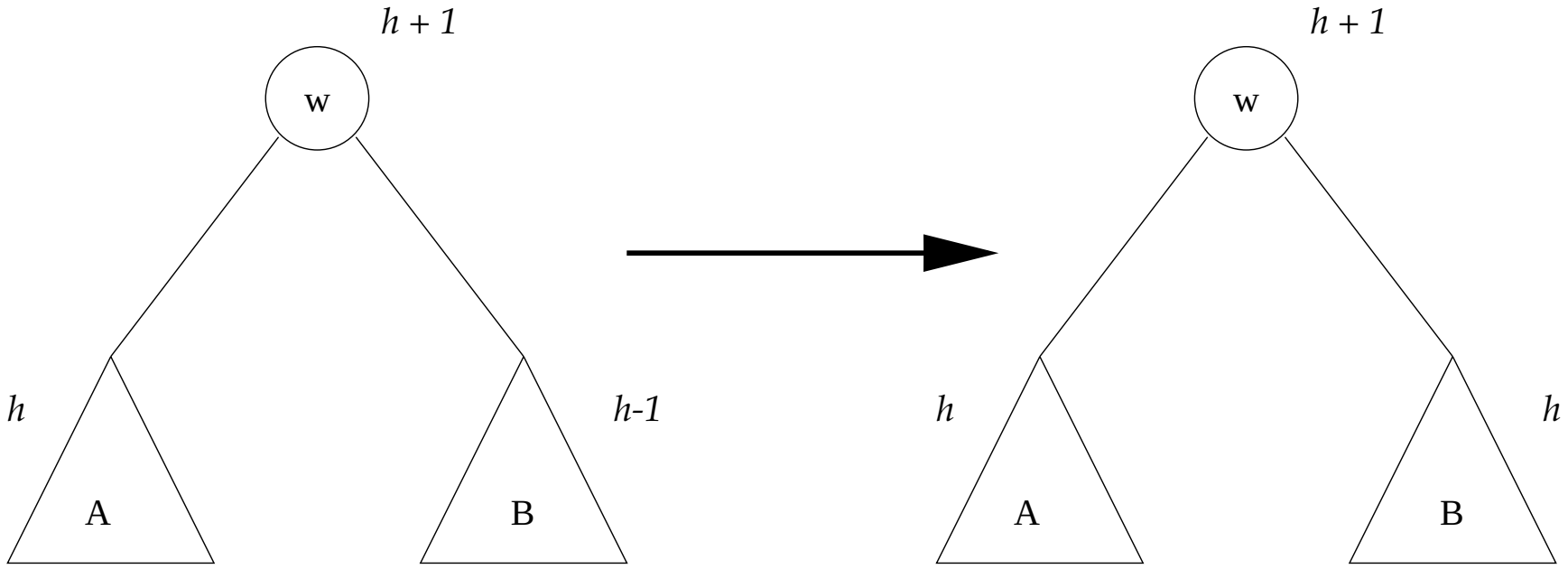
Einfügen und Rebalancierung (4)

Fall 2:

- B ist niedriger als A.
- Durch Einfügen in B ändert sich die Höhe von B.

$\Rightarrow$ Das AVL-Kriterium bleibt am Knoten w erfüllt.

Veranschaulichung:



Bemerkungen:

- Der enstandene Unterbaum ist wieder ein AVL-Baum, d.h. an w ist keine Rebalancierung notwendig.
- Da Für den Unterbaum $\mathfrak{U}$ keine Höhenveränderung stattgefunden hat, wird an w die Änderung absorbiert,
- d.h. an keinem Vorfahr von w wird eine Rebalancierung notwendig sein.

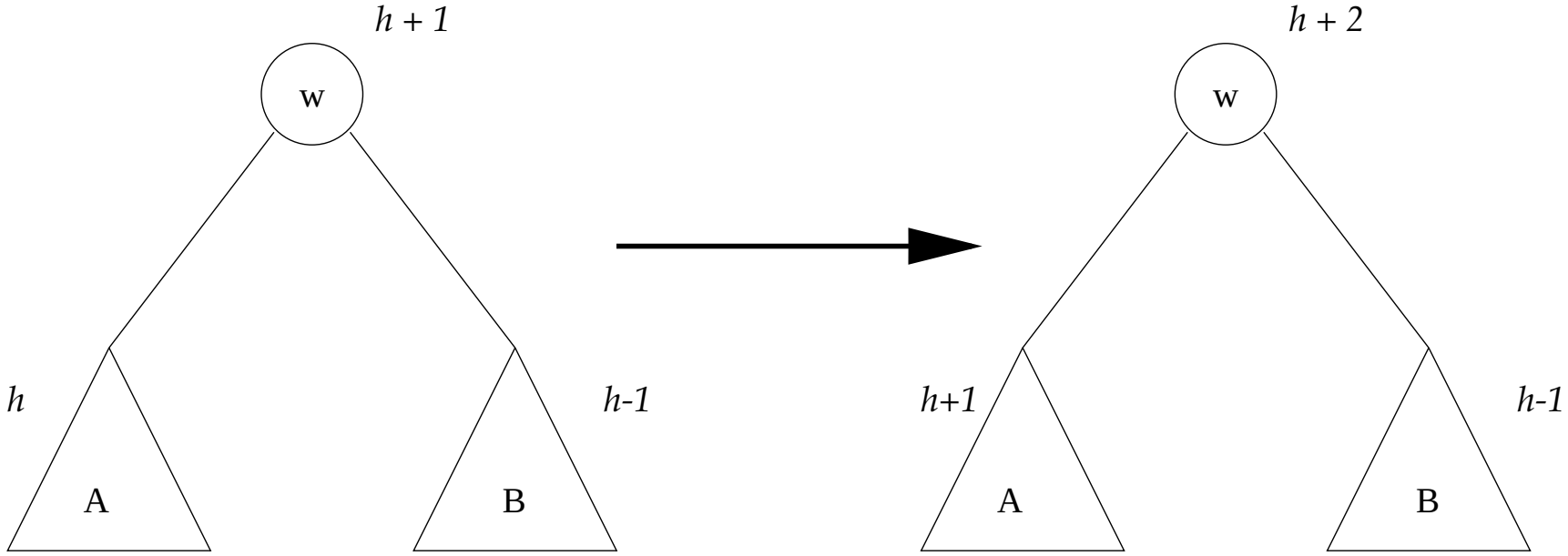
Einfügen und Rebalancierung (5)

Fall 3:

- B ist niedriger als A.
- Durch Einfügen in A ändert sich die Höhe von A.

$\Rightarrow$ Das AVL-Kriterium wird verletzt, d.h. der so entstandene Baum muss rebalanciert werden.

Veranschaulichung:

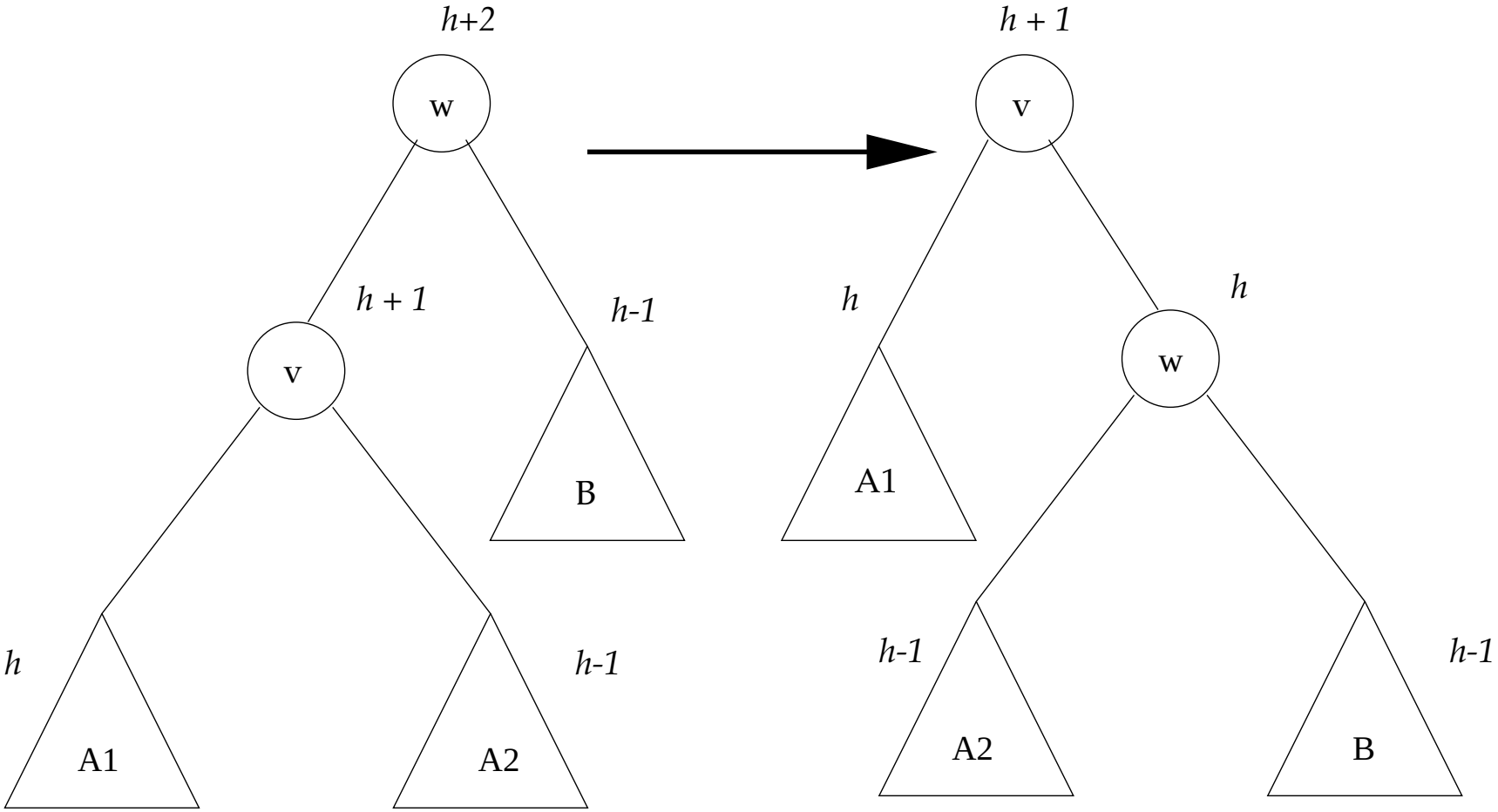


Hier können nun mehrere Unterfälle unterschieden werden.

Fall 3a:

- Im Unterbaum A ist der linke Unterbaum der höhere,
- d.h. die Höhenveränderung enstand durch Einfügen in den linken Unterbaum von A.
- In diesem Fall wird die Rebalancierung durch eine sogenannte *LL-Rotation* vorgenommen.

Veranschaulichung:



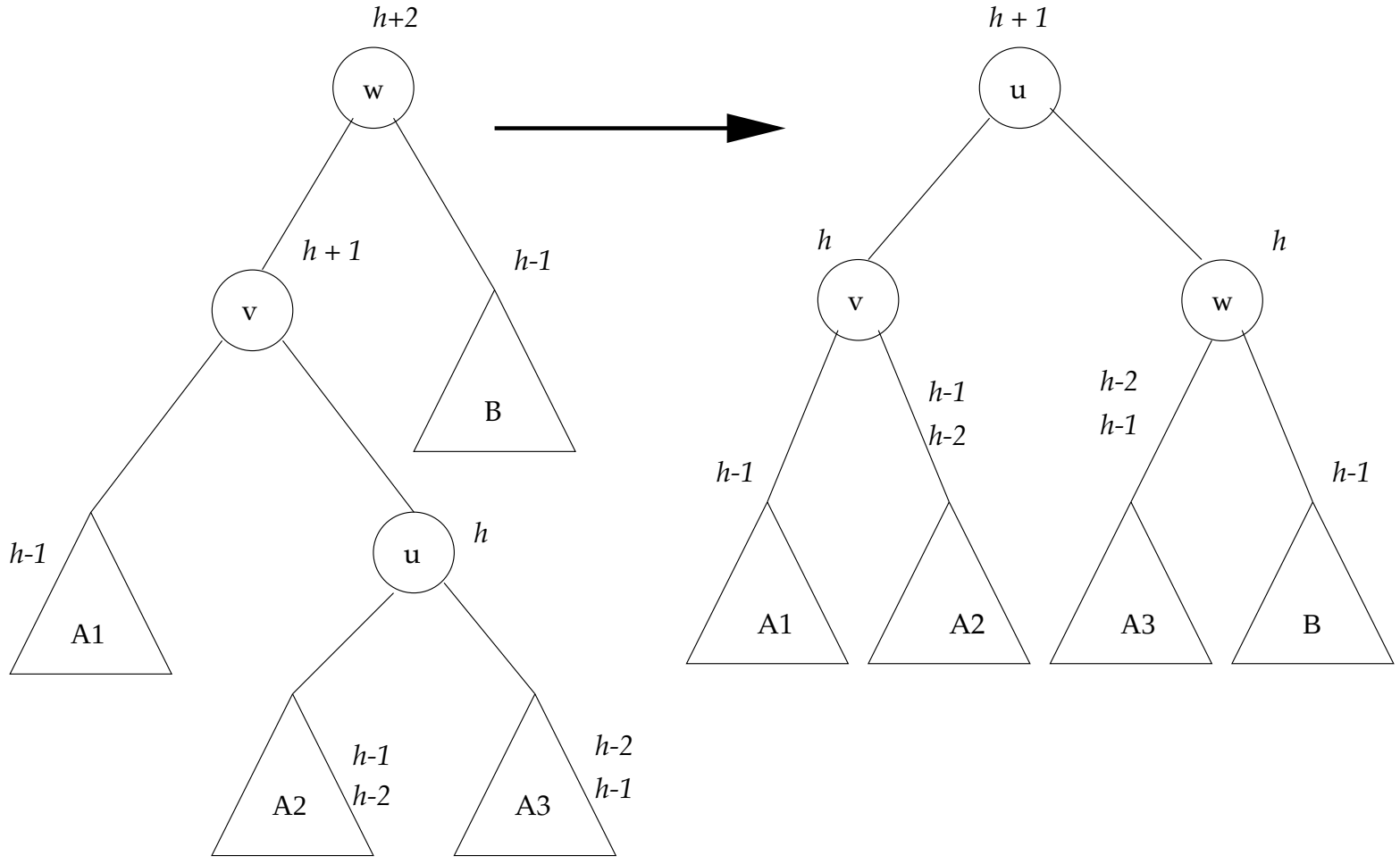
Bemerkungen:

- Nach der LL-Rotation erfüllt der Unterbaum das AVL-Kriterium.
- Die Sortierreihenfolge bleibt bei der LL-Rotation erhalten.
- Der Unterbaum U hat nach der LL-Rotation die gleiche Höhe wie vor der Einfügung.
- Damit wird durch die LL-Rotation die Änderung absorbiert, d.h. auf höherer Ebene ist keine Rebalancierung mehr notwendig.

Fall 3b:

- Im Unterbaum A ist der rechte Unterbaum der höhere,
- d.h. die Höhenveränderung enstand durch Einfügen in den rechten Unterbaum von A.
- In diesem Fall wird die Rebalancierung durch eine sogenannte *LR-Rotation* vorgenommen.

Veranschaulichung:



Bemerkungen:

- Nach der LR-Rotation erfüllt der Unterbaum das AVL-Kriterium.
- Die Sortierreihenfolge bleibt bei der LR-Rotation erhalten.
- Der Unterbaum hat nach der LR-Rotation die gleiche Höhe wie vor der Einfügung.
- Damit wird durch die LR-Rotation die Änderung absorbiert, d.h. auf höherer Ebene ist keine Rebalancierung mehr notwendig.

Einfügen und Rebalancierung: Fazit

- Das AVL-Kriterium kann durch Einfügen zerstört werden, aber durch einfache Umformungen kann es wieder hergestellt werden.
- Der dabei erforderliche Aufwand ist höchstens proportional zur Höhe des Baumes, also $O(\log n)$.
- Es wurden nur die Fälle betrachtet, in denen gilt $h(A) \geq h(B)$. Die anderen Fälle sind zu den behandelten symmetrisch.
- Bei dem zu Fall 3a symmetrischen Fall spricht man von einer *RR-Rotation*.
- Bei dem zu Fall 3b symmetrischen Fall spricht man von einer *RL-Rotation*.
- LL- und RR-Rotation werden auch als *einfache* Rotationen bezeichnet, LR- und RL-Rotation als *doppelte Rotationen*.

Löschen und Rebalancierung (1)

- Zunächst nutzt man den Löschalgorithmus von binären Bäumen.
- Es wird also immer ein Knoten gelöscht, der mindestens einen leeren Unterbaum hat.
- Dadurch kann sich die Höhe eines Unterbaums verringern und somit das AVL-Kriterium verletzt werden.
- Dieses muss anschließend durch Rebalancierung wieder hergestellt werden.
- Hier ergeben sich ähnliche Fälle wie beim Einfügen.

Löschen und Rebalancierung (2)

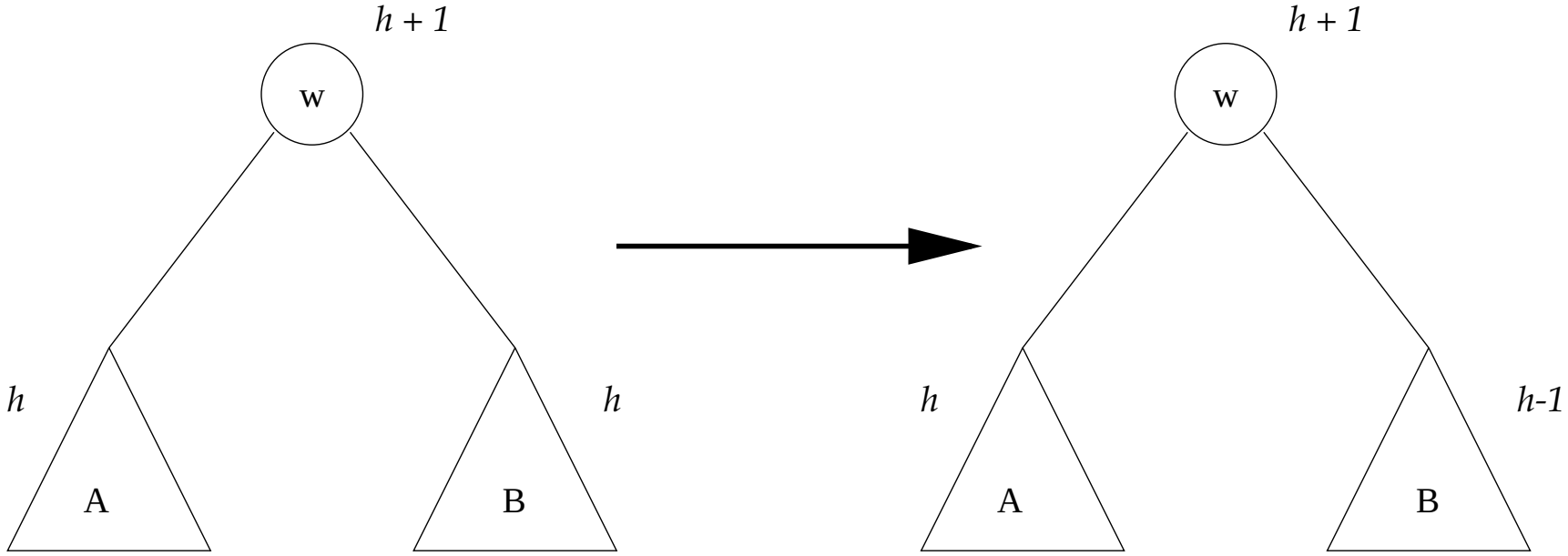
Fall 1:

- A und B haben die gleiche Höhe h.
- Durch Löschen in A (oder B) ändert sich die Höhe von A (bzw. B).

$\Rightarrow$ Das AVL-Kriterium bleibt am Knoten w erfüllt.

- Die Höhenänderung pflanzt sich nicht fort.

Veranschaulichung:



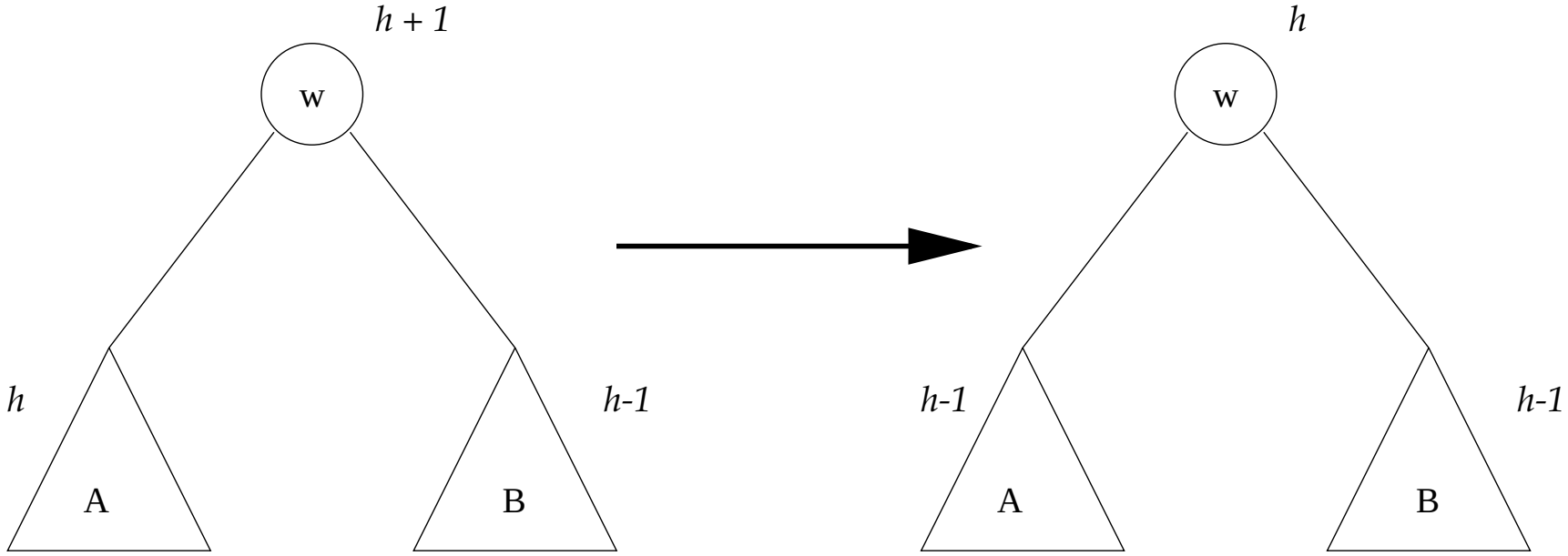
Löschen und Rebalancierung (3)

Fall 2:

- B ist niedriger als A.
- Durch Löschen in A ändert sich die Höhe von A.

$\Rightarrow$ Das AVL-Kriterium bleibt am Knoten w erfüllt.

Veranschaulichung:



- Die Höhe des Unterbaums U hat sich verringert.
- Die Höhenveränderung kann sich somit zur Wurzel hin fortpflanzen.
- An einem der Vorgnger von w kann eine Rebalancierung erforderlich werden.

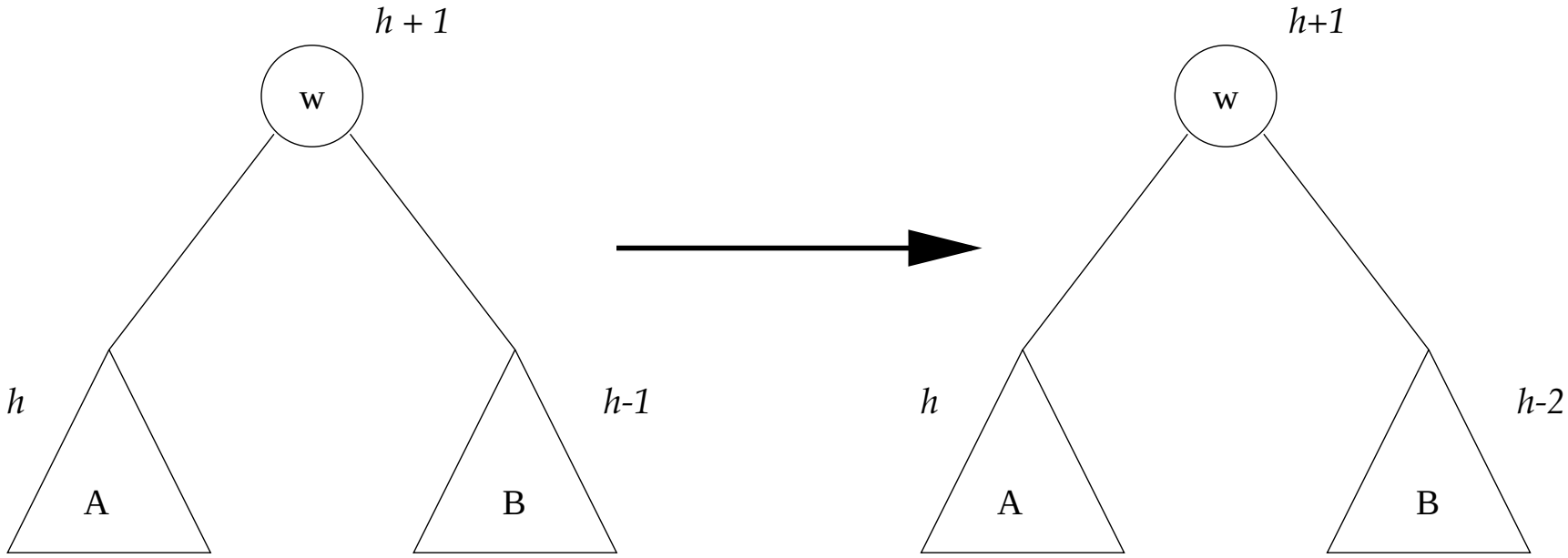
Löschen und Rebalancierung (4)

Fall 3:

- B ist niedriger als A.
- Durch Löschen in B ändert sich die Höhe von B.

$\Rightarrow$ Das AVL-Kriterium ist damit am Knoten w verletzt.

Veranschaulichung:

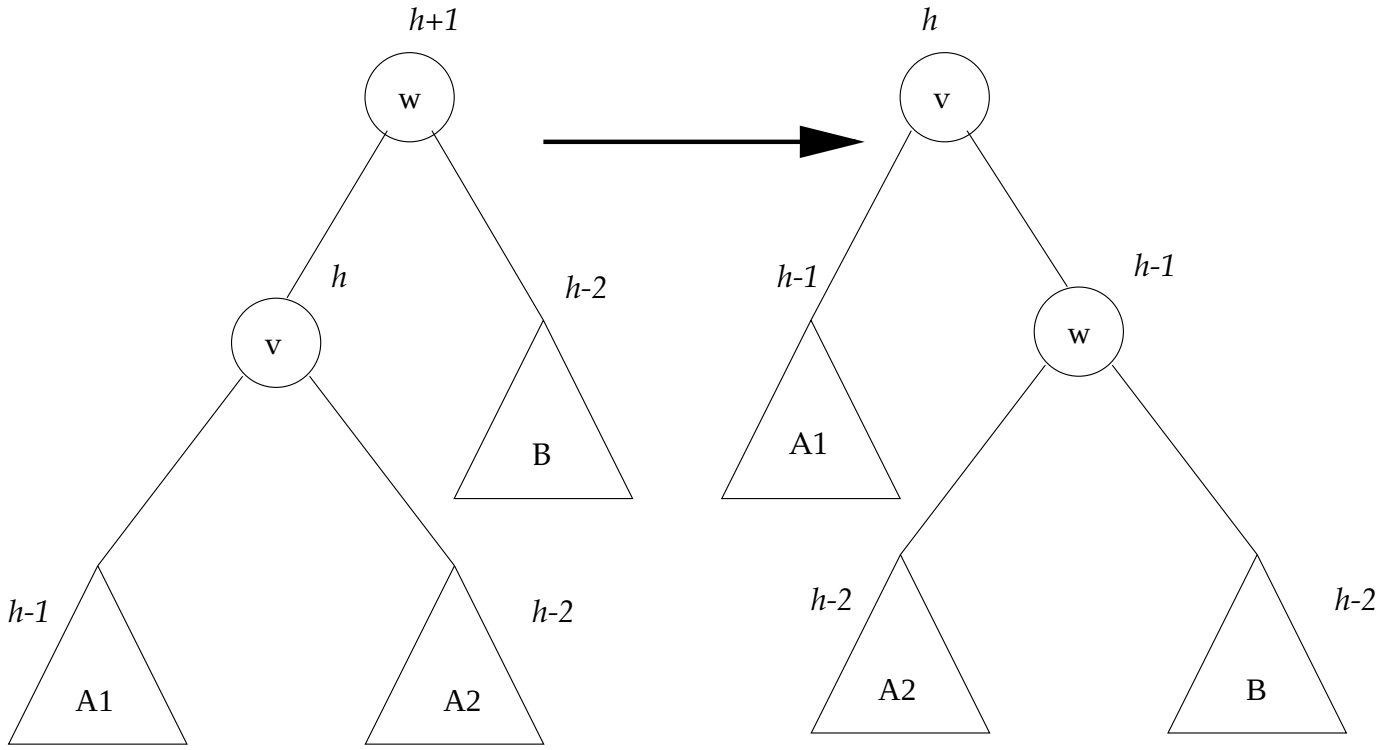


- Analog zum Einfügen können hier verschiedene Unterfälle unterschieden werden.

Fall 3a:

- Im Unterbaum A ist der linke Unterbaum der höhere.
- Ausgleich durch eine LL-Rotation.

Veranschaulichung:



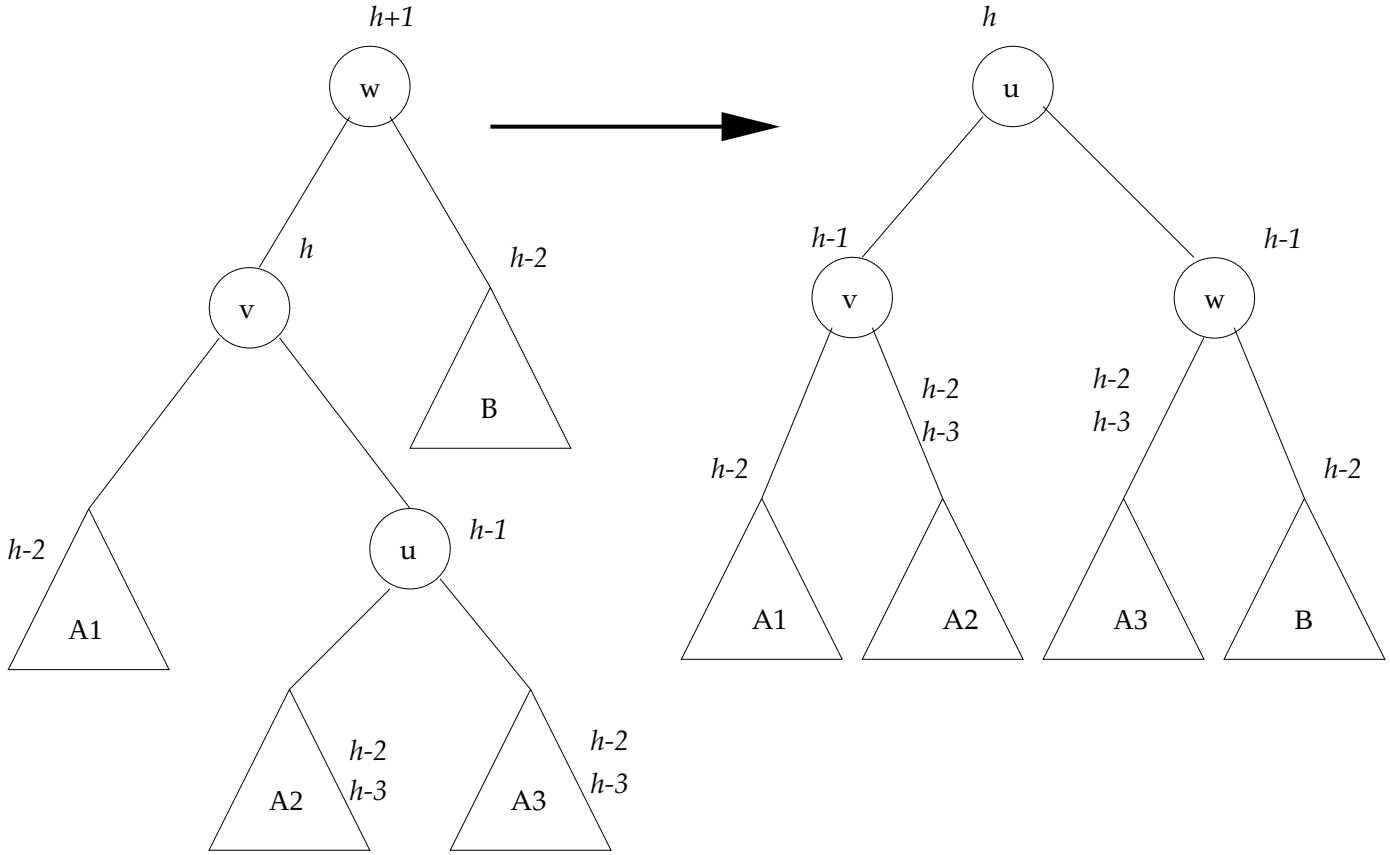
Bemerkungen:

- Durch die LL-Rotation hat sich die Höhe des Unterbaums $\mathfrak{U}$ verringert.
- Die Höhenänderung kann sich zur Wurzel hin fortpflanzen.
- Eventuell sind bei Vorfahren von w weitere Rotationen notwendig.

Fall 3b:

- Im Unterbaum A ist der rechte Unterbaum der höhere.
- Ausgleich durch eine LR-Rotation.

Veranschaulichung:



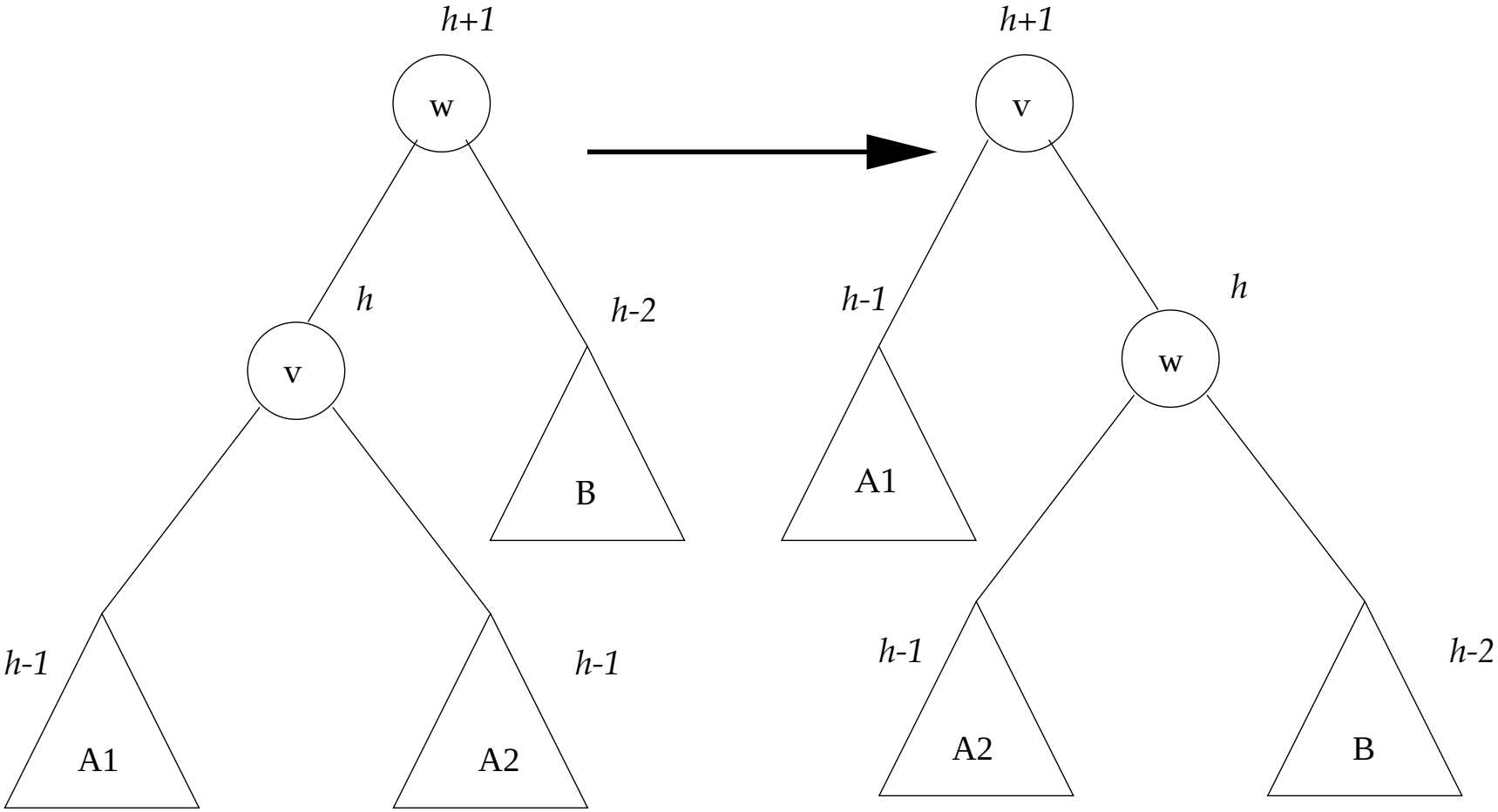
Bemerkungen:

- Durch die LR-Rotation hat sich die Höhe des Unterbaums $\mathfrak{U}$ verringert.
- Die Höhenänderung kann sich zur Wurzel hin fortpflanzen.
- Eventuell sind bei Vorfahren von w weitere Rotationen notwendig.

Fall 3c:

- Im Unterbaum A sind der rechte und der linke Unterbaum gleich hoch.
- Dieser Fall kann nur beim Löschen auftreten.
- Ausgleich durch eine LL-Rotation.

Veranschaulichung:



Bemerkungen:

- Durch die Rotation wird die Höhe des Unterbaums $\mathfrak{U}$ nicht verändert.
- Damit wird durch die LL-Rotation die Änderung absorbiert, d.h. auf höherer Ebene ist keine Rebalancierung mehr notwendig.

Löschen: Fazit

- Es wurden wiederum nur die Fälle mit $h(A) \geq h(B)$ betrachtet.
- Die anderen Fälle sind zu den hier behandelten Fällen symmetrisch.

- Durch Rotationen kann das AVL-Kriterium nach dem Löschen wieder hergestellt werden.
- Aufwand: $O(\log n)$, da
  - jede Rotation in $O(1)$ durchgeführt werden kann und
  - maximal $O(\log n)$ Rotationen notwendig sind.

Fazit für AVL-Bäume

- Mit AVL-Bäumen können geordnete Mengen effizient verwaltet werden.
- Suchen, Einfügen und Löschen ist in Zeit $O(\log n)$ möglich.
- AVL-Bäume haben für die Darstellung von geordneten Mengen im Arbeitsspeicher eine große praktische Bedeutung.
- Für Peripheriespeicher sind sie aber weniger geeignet.

7. Sortieren

Lernziele:

- Die wichtigsten Sortierverfahren kennen und einsetzen können,
- Aufwand und weitere Eigenschaften der Sortierverfahren kennen,
- das Problemlösungsparadigma Teile-und-herrsche verstehen und zur Konstruktion von Algorithmen verwenden können und
- Algorithmen für Probleme kennen, die mit Sortieren verwandt sind.

Einfache Sortierverfahren

Siehe "Einführung in die Programmierung":

- Sortieren durch Auswählen (S. 422–424)
- Sortieren durch Einfügen (S. 425–427)

Weiteres bekanntes Verfahren: Sortieren durch Vertauschen: Bubblesort (s.u.)

Bemerkung:

- Um die Darstellung zu vereinfachen, werden alle Sortierverfahren am Beispiel der aufsteigenden Sortierung von **int**-Werten (**int**-Feld) vorgestellt.
- Abstraktere Konzepte für den Vergleich haben wir in den vorangegangenen Kapiteln kennengelernt.

Bubblesort: Grundidee

- Gegeben ist ein zu sortierendes **int**-Feld a.
- In jeder Iteration durchlaufen wir das Feld von links nach rechts.
- Wenn `a[i] > a[i+1]` gilt, dann vertauschen wir die beiden Werte.
- Wenn in einer Iteration keine Vertauschung mehr notwendig ist, ist das Feld sortiert.

```
Beginn:                  5 4 2 3 7 8 6
Nach der 1. Iteration: 4 2 3 5 7 6 8
Nach der 2. Iteration: 2 3 4 5 6 7 8
Nach der 3. Iteration: 2 3 4 5 6 7 8
```

Bubblesort: Bemerkungen

- Nach der ersten Iteration ist garantiert, dass das größte Element ganz rechts steht (Index `length-1`).
- Spätestens nach der zweiten Iteration befindet sich das zweitgrößte Element an der zweiten Stelle von rechts.
- Somit ist das Feld spätestens nach `length` Iterationen sortiert.

Optimierung:

- Wenn in einer Iteration die letzte Vertauschung an Position `i` (mit `i+1`) stattgefunden hat, dann enthält das Feld ab Position `i+1` die `length-i-1` größten Werte des Feldes.
- Konsequenz: Wir merken uns die Position der letzen Vertauschung und gehen in einer Iteration nur noch bis zu dieser Stelle.

Bubblesort: Algorithmus

```
static void bubblesort(int[] a) {
    int lastSwap=a.length-1;       // Position des letzten Tausches
    int iright;                    // rechte Grenze fuer Iteration

    while (lastSwap > 0) {         // solange in voriger Iteration Vertauschung
        iright = lastSwap;         // lege rechte Grenze fuer Iteration fest
        lastSwap = 0;              // noch gab es keine Vertauschung
        for(int i=0 ; i<iright ; i++) {
            if ( a[i] > a[i+1] ) { // wenn Tausch notwendig
                swap(a,i,i+1);     // dann tausche
                lastSwap = i;      // und merke Position
            }
        }
    }
}
```

Bubblesort: Aufwand im Best und Worst Case

Das Feld a habe die Länge n.

- Best Case:
  Das Feld a ist schon aufsteigend sortiert. Dann ist in der ersten Iteration keine Vertauschung notwendig und der Algorithmus terminiert.
  Aufwand: $O(n)$
- Worst Case:
  Das Feld a ist absteigend sortiert. Dann sind n Iterationen notwendig mit $n-1, n-2, \ldots$ Vertauschungen.
  Aufwand: $O(n^2)$

Bubblesort: Aufwand im Mittel (Average Case)

- Wenn zu Beginn der Sortierung für `i < j` gilt `a[i] > a[j]` (wir bezeichnen dies als Fehlstand), dann müssen die beiden Elemente `a[i]` und `a[j]` irgendwann einmal getauscht werden.
- Wie viele Fehlstände hat eine zufällig generierte Folge der Länge n im Mittel?
- Für das erste Element wird es unter den Elementen 2 bis n im Mittel $\frac{1}{2}(n-1)$ Fehlstände geben, für das zweite Element im Mittel $\frac{1}{2}(n-2)$, usw.
- Wir haben also im Mittel

$$\frac{1}{2}\sum_{i=1}^{n-1}(n-i) = \frac{1}{2}\sum_{i=1}^{n-1} i = \frac{1}{2}\cdot\frac{n(n-1)}{2} = O(n^2)$$

  Fehlstände.
- Damit sind im Mittel $O(n^2)$ Vertauschungen notwendig und der Aufwand beträgt auch im Mittel $O(n^2)$.

Zusammenfassung: Einfache Sortierverfahren

- Alle bisher betrachteten Verfahren haben im Mittel eine Laufzeit von $O(n^2)$.
- Ist Sortieren nicht schneller möglich als in Zeit $O(n^2)$?
- Doch!
- Und im Prinzip wissen wir auch schon wie! ☞ ausgeglichene Bäume

Sortieren mit Hilfe von ausgeglichenen Bäumen

Erste Idee: Wir bauen mit den n zu sortierenden Werten einen AVL-Baum auf.

Konsequenzen:

- Jede Einfügeoperation hat Aufwand $O(\log n)$.
- Gesamtaufwand für den Aufbau des Baums daher: $O(n \log n)$
- Eine Inorder-Traversierung des AVL-Baums liefert die sortierte Reihenfolge in Zeit $O(n)$.
- Gesamtaufwand Sortierung: $O(n \log n)$

Kleinere Probleme:

- relativ hoher konstanter Faktor, zur Erinnerung: Höhe von AVL-Bäumen bis zu 1.5 mal minimale Höhe, "aufwendige" Ausgleichsalgorithmen
- keine In-Place-Sortierung (in situ): Es wird zusätzlich $O(n)$ Speicher benötigt.

Heap

Definition 7.1. Es sei T ein binärer Baum, wobei an jedem inneren Knoten w ein Wert $val(w)$ gespeichert wird. Für einen Knoten w bezeichne w_l den linken und w_r den rechten Sohn (Wurzel des linken bzw. rechten Unterbaums).

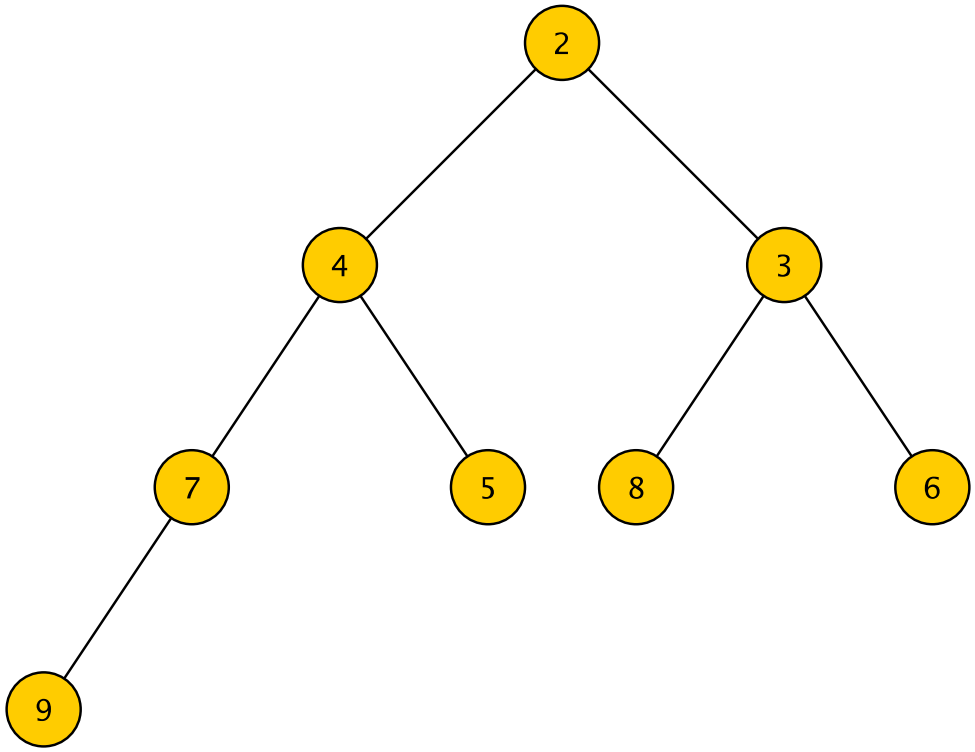
T ist ein *Heap*, gdw. für alle Knoten w gilt:

- w_l nicht leer $\Rightarrow val(w) \leq val(w_l)$ und
- w_r nicht leer $\Rightarrow val(w) \leq val(w_r)$

Anschaulich: Der Wert an einem Knoten ist stets kleiner oder gleich den Werten an den Söhnen.

Wir sprechen in diesem Fall von einem Min-Heap, gilt $\geq$ dann liegt ein Max-Heap vor.

Beispiel: Min-Heap



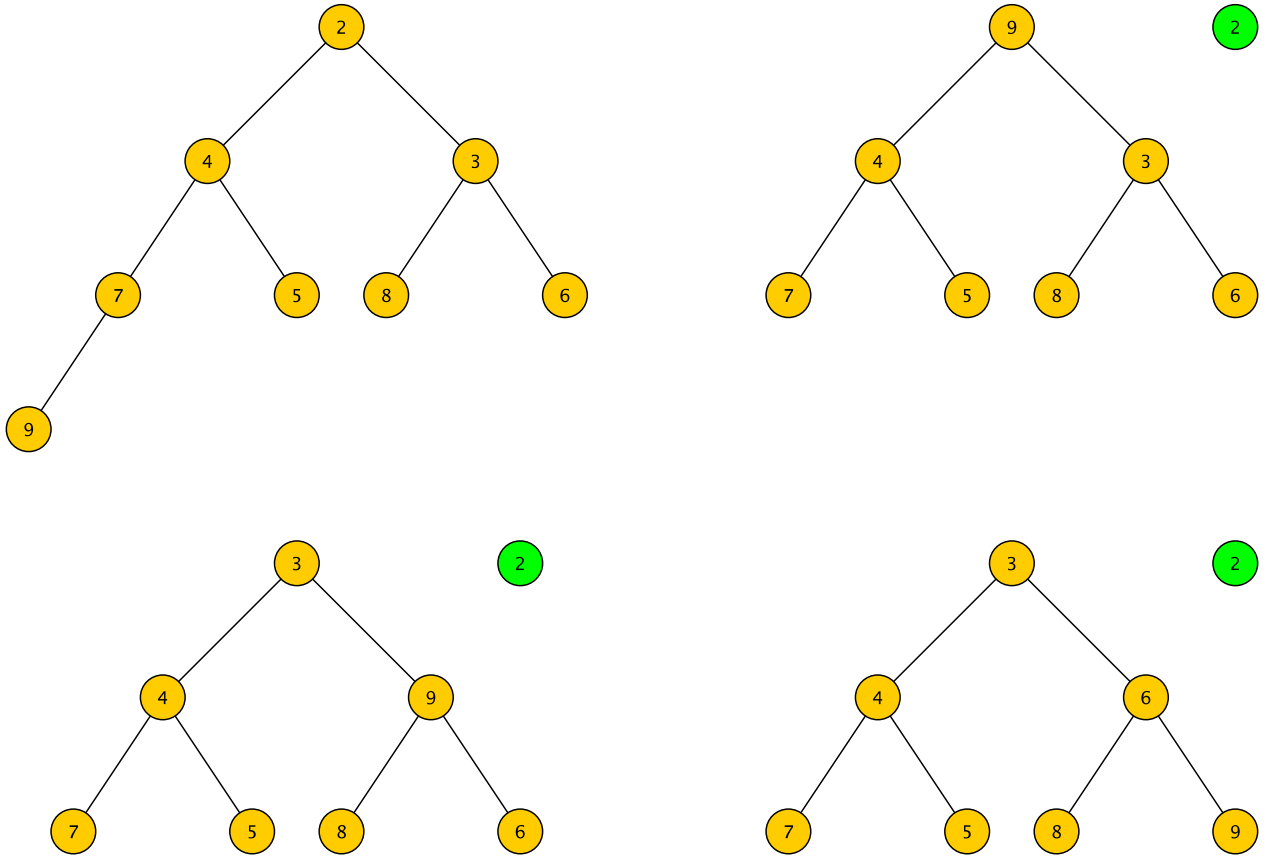
Sortieren eines Heap

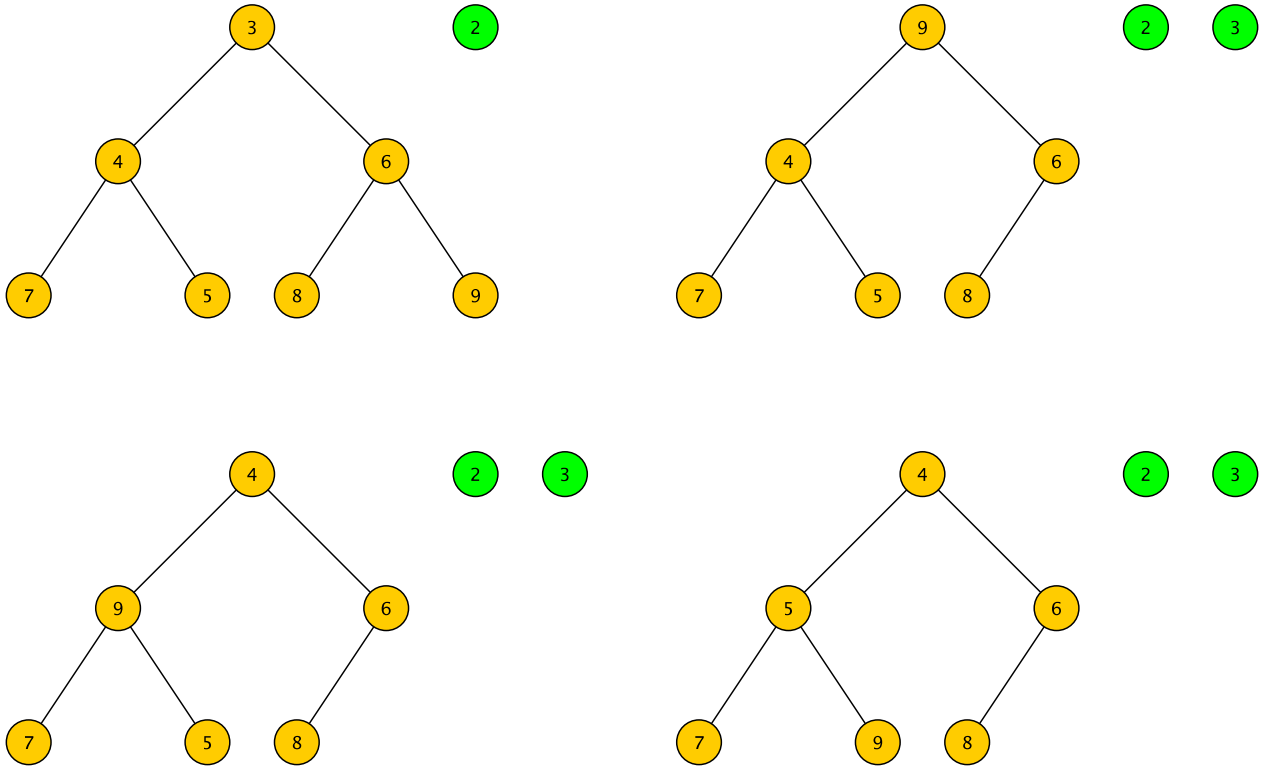
- An der Wurzel steht (bei einem Min-Heap) der kleinste Wert.
- Wir nehmen den Wert der Wurzel, nehmen ihn in die sortierte Folge auf und löschen ihn aus dem Heap.
- Um die Lücke an der Wurzel zu füllen, verschieben wir den am weitesten rechts stehenden Wert der untersten Ebene in die Wurzel und löschen den dazugehörigen Knoten.

  Hierdurch kann die Heap-Definition an der Wurzel verletzt werden.
- Sollte die Heap-Definition verletzt sein, vertauschen wir den Wert der Wurzel mit dem kleineren der beiden Söhne.

  Damit ist an der Wurzel die Heapbedingung wieder erfüllt.
- Falls jetzt am ausgetauschten Sohn die Heap-Bedingung verletzt ist, verfahren wir dort wie an der Wurzel.
- Wir setzen dieses Verfahren fort, bis keine Verletzung mehr vorliegt. Dies ist spätestens dann der Fall, wenn wir die unterste Ebene des Heap erreichen.

Beispiel: Sortieren eines Heap





✎ Beispiel weiter an Tafel

Vollständiger Binärbaum

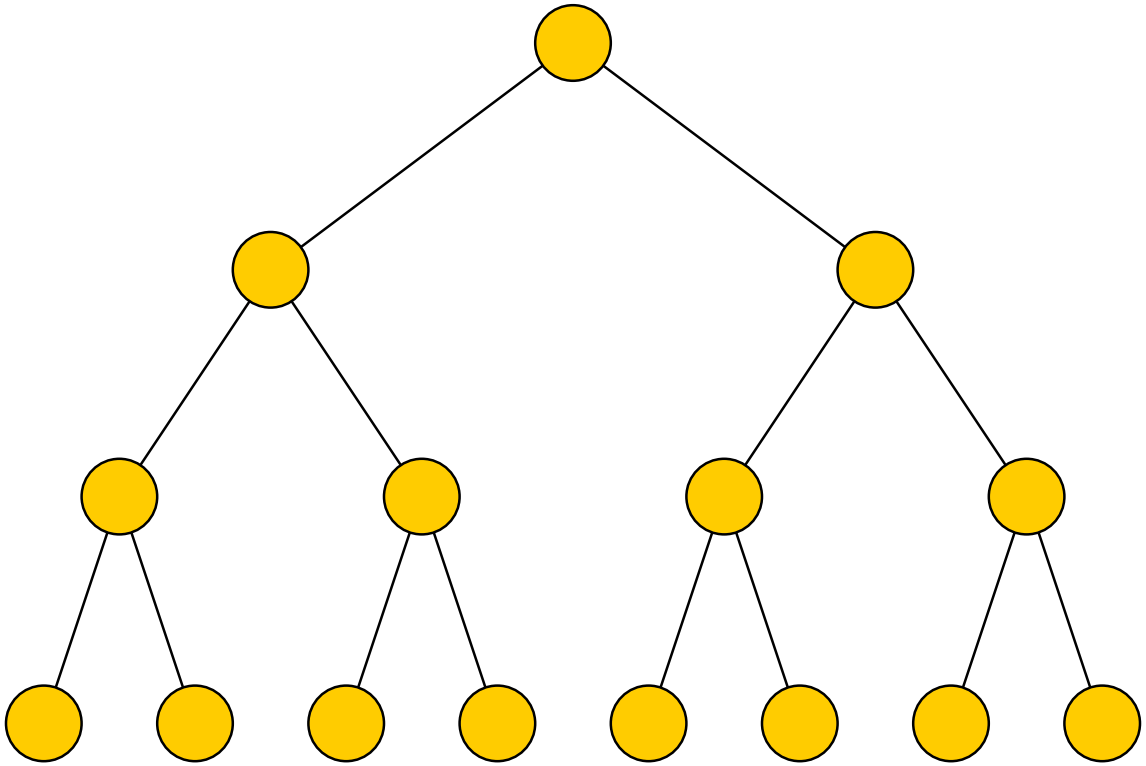
Definition 7.2. Ein binärer Baum der Höhe h heißt *vollständig*, wenn er $2^h - 1$ (innere) Knoten hat.

Bemerkung: vgl. Lemma 6.2:

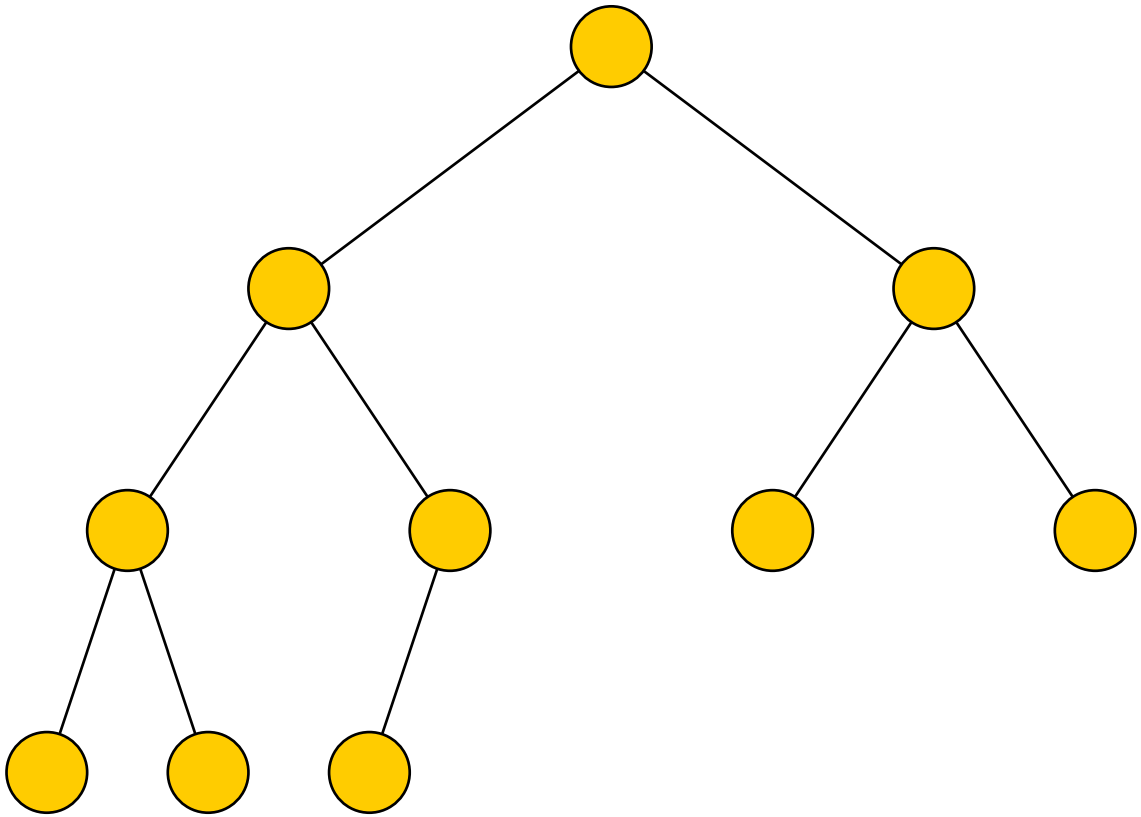
- Ein vollständiger Baum hat für seine Höhe maximal viele Knoten n. Für n ist die Höhe minimal.
- Es gilt: $n = 2^h - 1$.
- Daraus folgt: $h = \log_2(n + 1)$

Definition 7.3. Ein binärer Baum der Höhe h mit $2^{h-1} \leq n \leq 2^h - 1$ Knoten, bei dem die Knoten der Ebene h alle möglichst weit links stehen, heißt *links vollständiger binärer Baum*.

Beispiel: Vollständiger Binärbaum der Höhe 4



Beispiel: Links vollständiger Baum

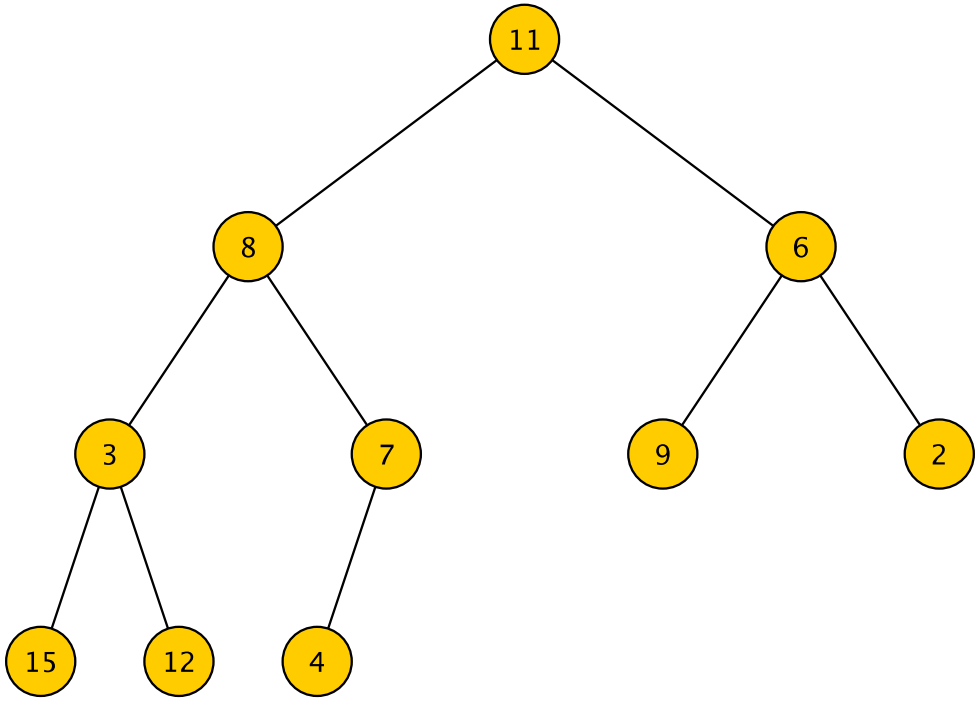


Konstruktion eines Heap

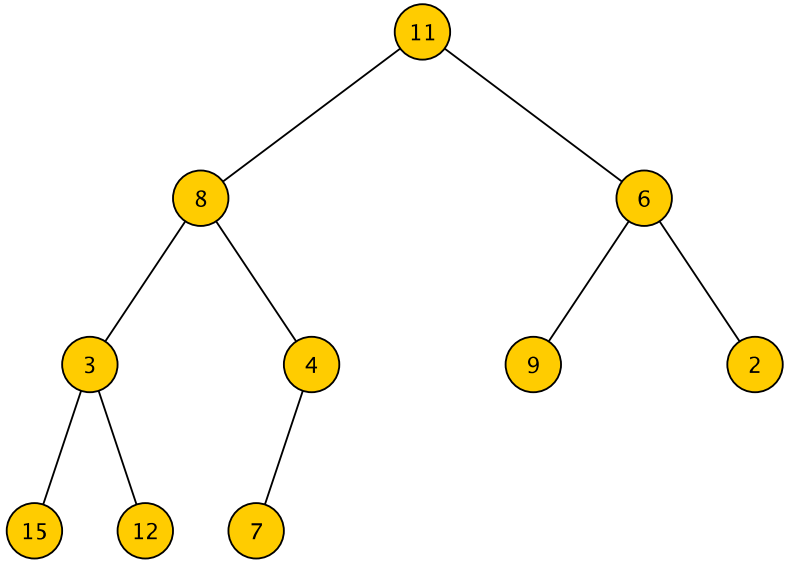
- Wir bauen einen links vollständigen Binärbaum mit n Knoten auf und
- platzieren die zu sortierenden Werte in den Knoten von oben nach unten und in jeder Ebene von links nach rechts.
- Wir überprüfen die Heap-Definition von unten nach oben und in jeder Ebene von rechts nach links.
- Wenn die Heap-Definition an einem Knoten verletzt ist, dann tauschen wir den Wert am Knoten mit dem kleineren der beiden Söhne. Wenn notwendig, setzen wir dieses Verfahren am ausgetauschten Sohn fort (vgl. Folie 293).

Beispiel: Konstruktion eines Heap

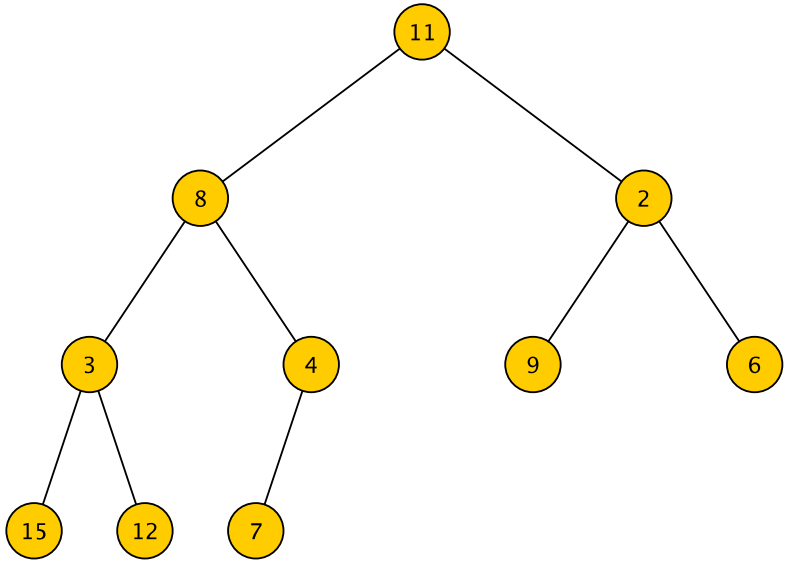
links vollständiger Binärbaum zur Folge 11 8 6 3 7 9 2 15 12 4



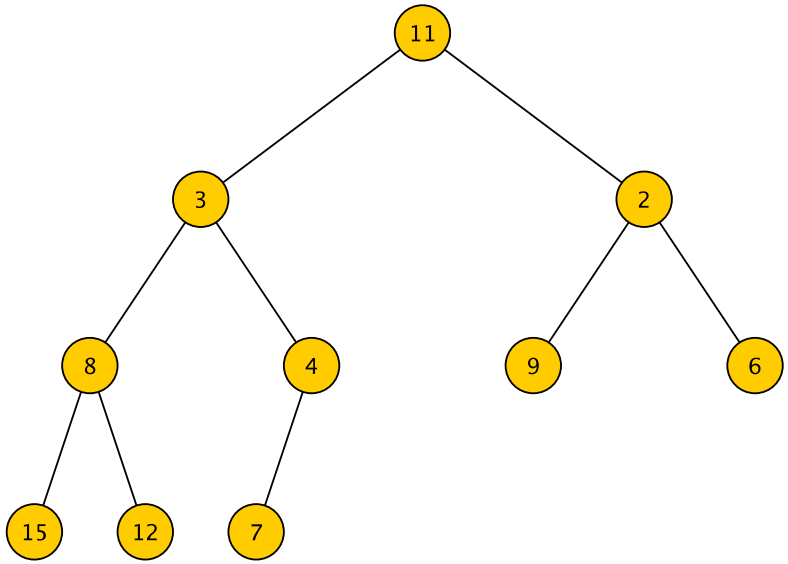
4 12 15 2 9 ok, Vertauschung bei 7



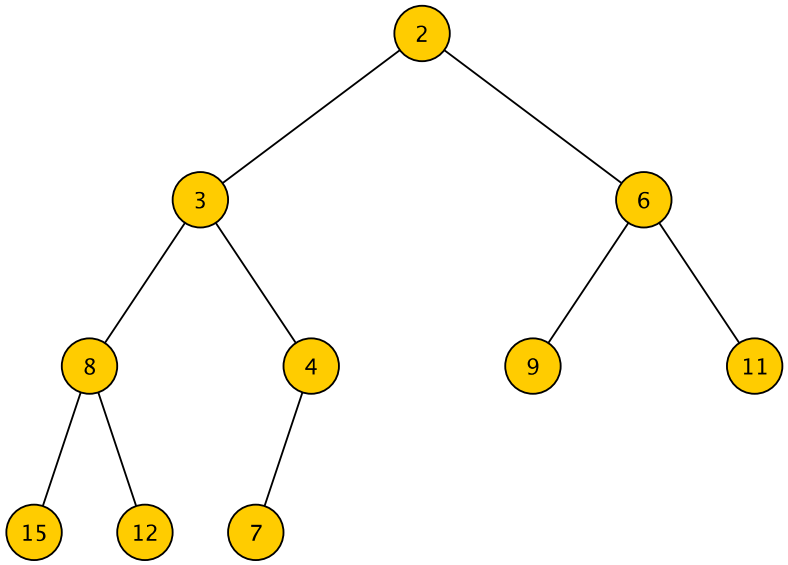
3 ok, Vertauschung bei 6



Vertauschung bei 8



Vertauschung bei 11 mit weiterer Vertauschung



Damit ist der Heap fertig!

Heapsort

(1) Heap konstruieren (Folie 299)

(2) Heap sortieren (Folie 293)

Aufwand im Worst Case:

- Aufbau des links vollständigen Binärbaums: $O(n)$
- Konstruktion des Heap aus dem links vollständigen Baum: $O(n \log n)$, weil für jeden Knoten höchsten $h = O(\log n)$ Vertauschungen anfallen.
- Sortieren des Heap: $O(n \log n)$, weil nach jedem "Abpflücken" der Wurzel höchstens $h = O(\log n)$ Vertauschungen anfallen.
- Gesamtaufwand: $O(n \log n)$

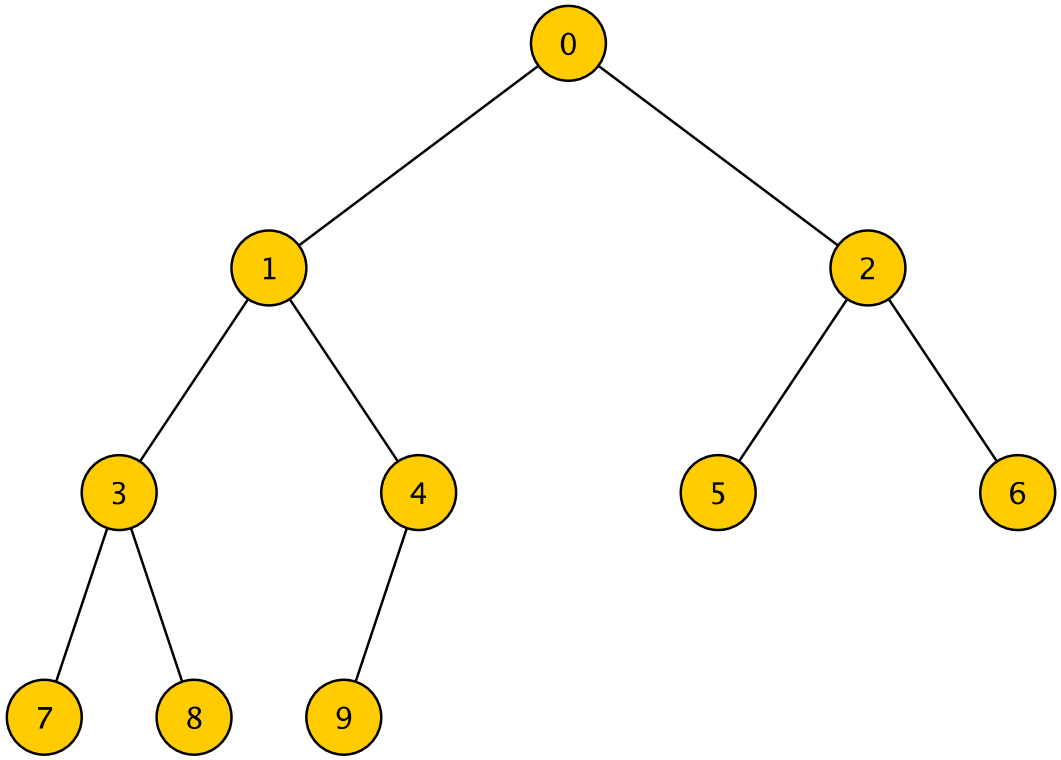
Abbildung eines Heap auf ein Feld

Wir können einen vollständigen Binärbaum mit n Knoten wie folgt auf ein Feld mit Index von 0 bis $n-1$ abbilden:

- Die Wurzel liegt bei Index 0.
- Der Knoten mit Index i hat seine Söhne bei Index $2i+1$ (linker Sohn) und $2i+2$ (rechter Sohn).
- Die Söhne existieren genau dann, wenn $2i+1 \leq n-1$ bzw. $2i+2 \leq n-1$ gilt.

Beispiel: Feldabbildung eines links vollständigen Binärbaums

$n = 10$

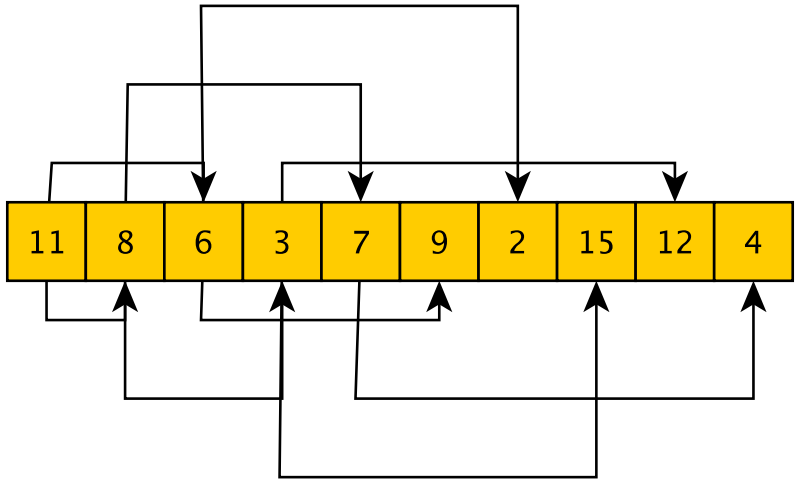


In-Place Heapsort

- Heap-Konstruktion: Stelle im Feld von rechts nach links die Heap-Definition für einen Max-Heap her!
- Heap-Sortierung: analog zu Folie 293:
  - Nehme in jeder Iteration das Maximum aus dem Heap (Index 0),
  - verschiebe den am weitesten rechts stehenden Wert der untersten Ebene in die Wurzel,
  - platziere das soeben gelöschte Maximum an die Stelle, wo der am weitesten rechts stehende Wert der untersten Ebene stand und
  - stelle Heap-Definition (für Max-Heap) wieder her.

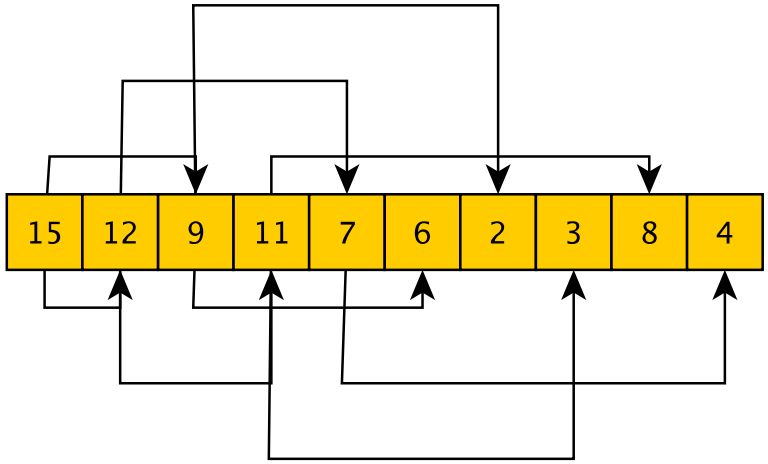
Beispiel: In-Place Heapsort

Zu sortierende Folge als Feld:

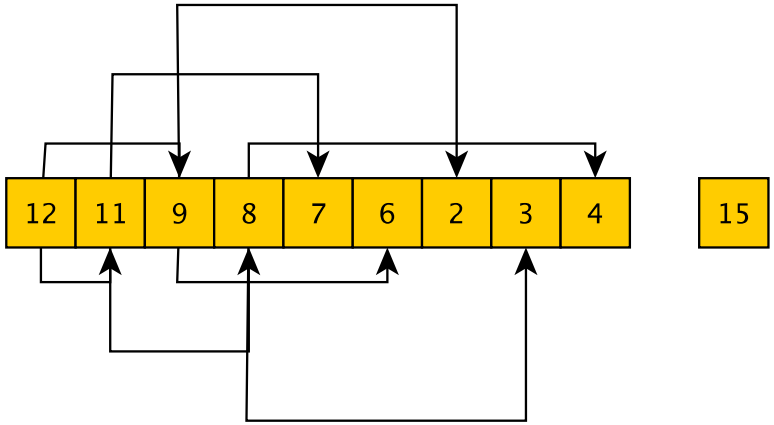


Die unteren Pfeile verweisen auf die linken Söhne, die oberen auf die rechten Söhne (nur zur Veranschaulichung).

Die Konstruktion des Max-Heap liefert:



Wir ziehen die 4 auf die Wurzel. Die 15 wird im Feld der 4 abgelegt, dies gehört jetzt aber nicht mehr zum Heap. Ausgehend von der 4 in der Wurzel stellen wir durch Vertauschungen wieder einen Heap her.



✎ weiter an der Tafel

Fazit Heapsort

- effizient auch im Worst Case: Zeit $O(n \log n)$
- In-Place Implementierung möglich: $O(1)$ zusätzlicher Speicher

Paradigma: Teile-und-Herrsche

Teile-und-herrsche ist ein allgemeines Prinzip zur Konstruktion von Algorithmen.

Andere Bezeichnungen: Divide-and-conquer, divide et impera

- Zerlege das gegebene Problem in mehrere getrennte Teilprobleme,
  - löse diese einzeln und
  - setze die Lösung des ursprünglichen Problems aus den Teillösungen zusammen.
- Wende diese Technik auf jedes der Teilprobleme an, dann deren Teilprobleme usw., bis die Teilprobleme klein genug sind, um sie explizit zu lösen.
- Strebe an, dass jedes Teilproblem von derselben Art ist, wie das ursprüngliche Problem.

☞ Rekursion

Mergesort

Zerlegung: Zerlege die zu sortierende Folge in zwei möglichst gleich große Teilfolgen und sortiere die Teile einzeln.

Beispiel: 11 8 6 3 7 9 2 15 12 4 wird in zwei Folgen zerlegt: 11 8 6 3 7 und
9 2 15 12 4

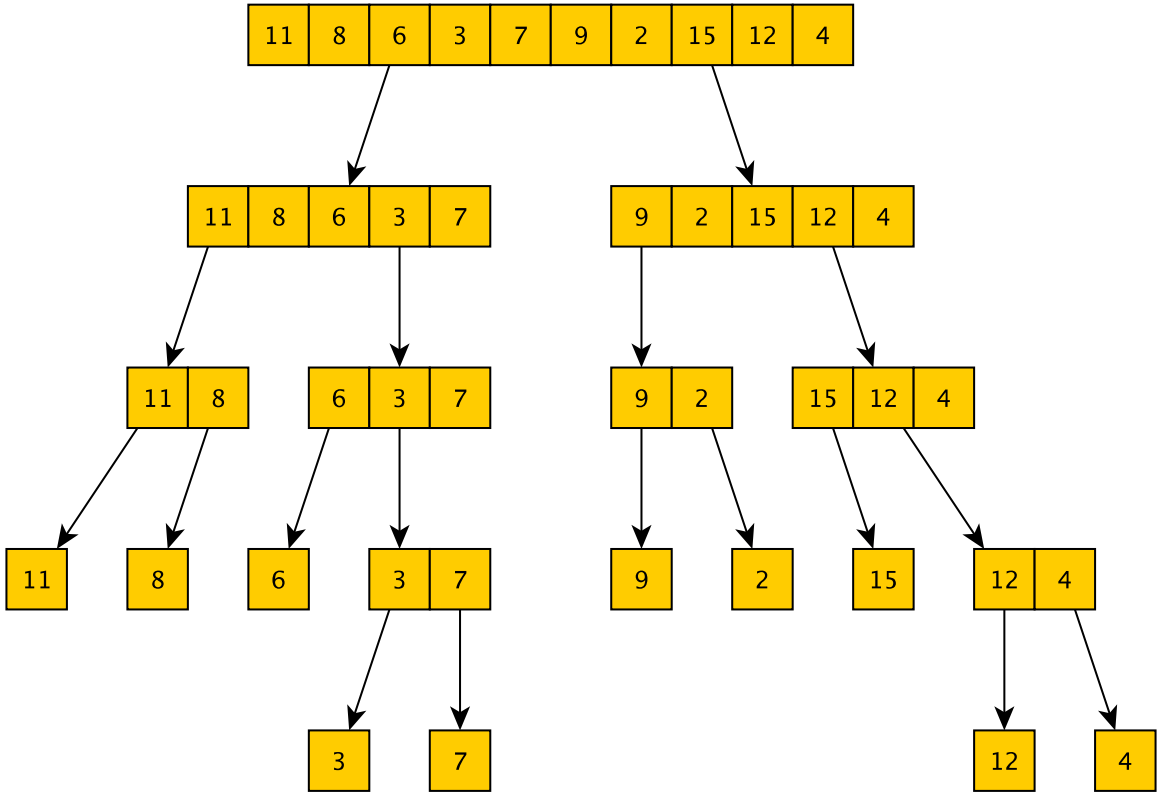
Zusammensetzung: Mische die sortierten Teilfolgen, so dass eine sortierte Gesamtfolge entsteht.

Beispiel: Die sortierten Teilfolgen lauten: 3 6 7 8 11 und 2 4 9 12 15

Sortierte Mischung ergibt 2 3 4 6 7 8 9 11 12 15

Explizite Lösung: Teilfolgen der Länge eins sind bereits sortiert.

Zerlegung


11 8 6 3 7 9 2 15 12 4
11 8 6 3 7
9 2 15 12 4
11 8
6 3 7
9 2
15 12 4
11
8
6
3 7
9
2
15
12 4
3
7
12
4

Algorithmus Zerlegung

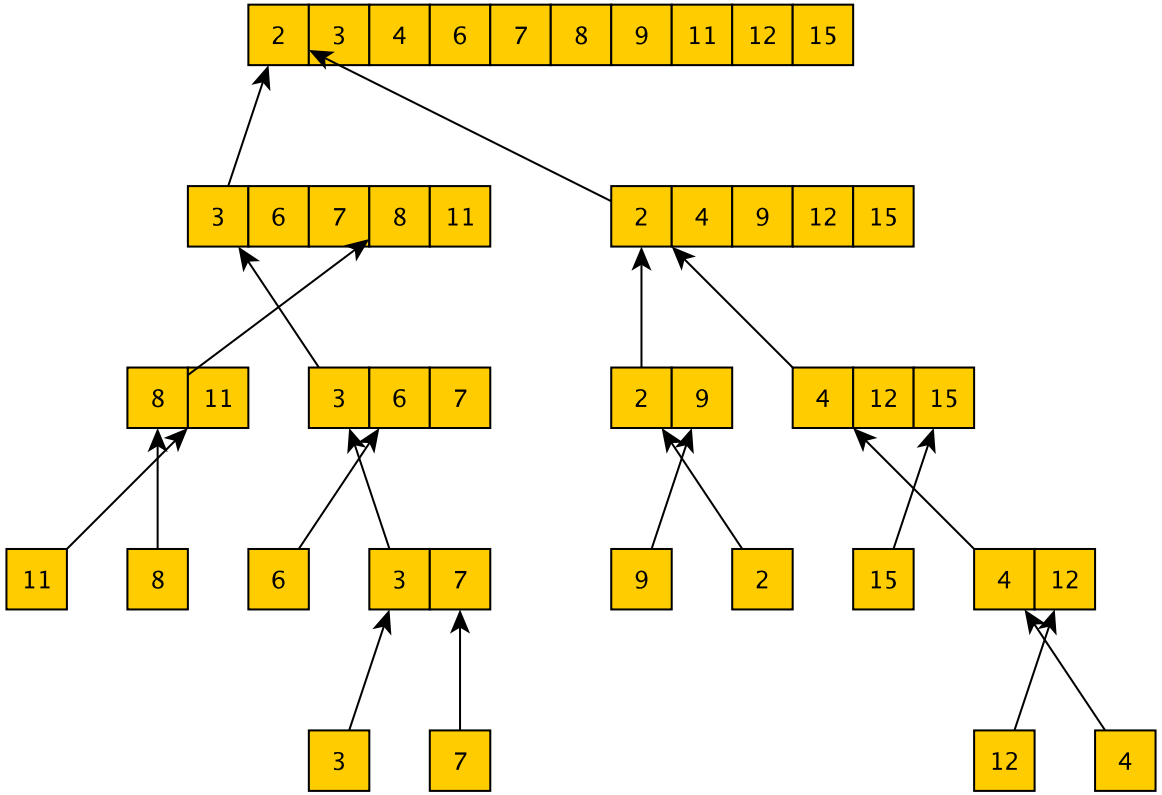
```
/*
 * Sortiere den Bereich des Feldes a zwischen il und ir
 * il ist das erste Element am linken Rand des zu sortierenden Bereichs
 * ir ist das erste Element am rechten Rand, das nicht mehr zum Bereich gehoert
 */
static void mergesort(int[] a, int il, int ir) {
    int imid = (il+ir)/2;          // mittleres Element

    if (ir-il<=1) {                // Bereich ist schon sortiert, wenn nicht mehr als
        return;                    // ein Wert zu sortieren ist.
    }

    mergesort(a, il, imid);        // sortiere linken Teil
    mergesort(a, imid, ir);        // sortiere rechten Teil

    /* jetzt mischen, siehe unten */
}
```

Mischen


2 3 4 6 7 8 9 11 12 15
3 6 7 8 11
2 4 9 12 15
8 11
3 6 7
2 9
4 12 15
11
8
6
3 7
9
2
15
4 12
3
7
12
4

Algorithmus Mischen

```
static void mergesort(int[] a, int il, int ir) {
    int imid = (il+ir)/2;

    /* zerlegen, siehe oben */

    int[] atmp = new int[ir-il];      // Hilfsfeld, um sortierte Mischung zu speichern
    int   itmp = 0;                   // Index fuer Hilfsfeld
    int   jl = il;                    // Index fuer linken sortierten Teil von a
    int   jr = imid;                  // Index fuer rechten sortierten Teil von a

    while (jl < imid && jr < ir) {    // solange es in den sortierten Teilen noch Werte gibt
        if (a[jl] <= a[jr]) {         // nehme den kleineren aus dem linken Teil
            atmp[itmp] = a[jl];       // uebertrage ihn nach atmp
            jl++;                     // und gehe im linken Teil zum naechsten Wert
        }
        else {                        // nehme kleineren aus rechtem Teil
            atmp[itmp] = a[jr];       // uebertrage ihn nach atmp
            jr++;                     // und gehe im rechten Teil zum naechsten Wert
```

```
        }
        itmp++;                          // auf jeden Fall ist in atmp jetzt ein Wert mehr
    }

    // jetzt ist im linken oder im rechten sortierten Teil
    // kein Wert mehr
    while (jl < imid) {                  // uebertrage Reste vom linken Teil nach atmp
        atmp[itmp] = a[jl];
        jl++; itmp++;
    }
    while (jr < ir) {                    // uebertrage Reste vom rechten Teil nach atmp
        atmp[itmp] = a[jr];
        jr++; itmp++;
    }

    // jetzt liegt die sortierte Mischung des linken und rechten Teils in atmp
    // kopiere die Inhalte von atmp nach a
    for (int i=0 ; i<atmp.length ; i++) {
       a[il+i] = atmp[i];
    }
}
```

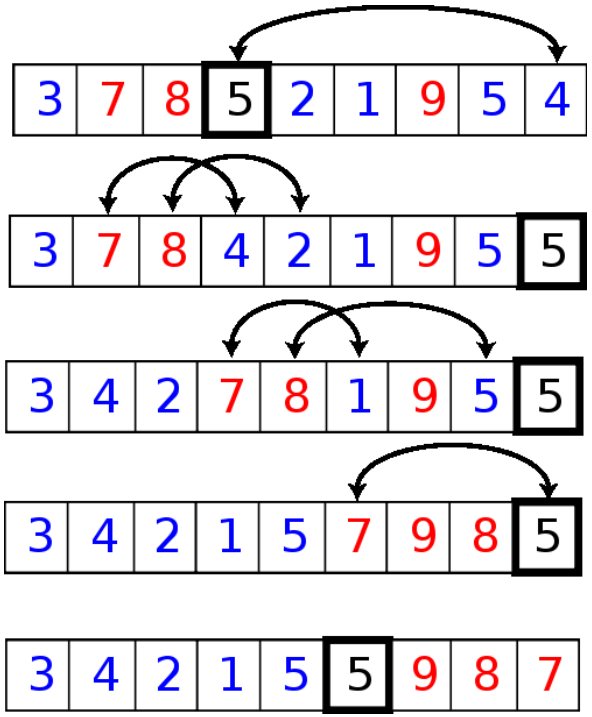
Eigenschaften von Mergesort

- Gesamtaufwand: Zeit $O(n \log n)$
  - Mischen zweier Teilfolgen der Länge n_l bzw. n_r in Zeit $O(n_l + n_r)$
  - Zerlegung produziert ausgeglichenen Baum der Höhe $O(\log n)$ aus Teilproblemen
  - Auf jeder Ebene Aufwand von $O(n)$ für das Mischen der Teilfolgen
- Keine In-Place Sortierung
- Mergesort ist *stabil*, d.h. gleiche Werte behalten in der sortierten Folge ihre relative Reihenfolge.

Quicksort

- Eines der bekanntesten Sortierverfahren ist Quicksort.
- basiert ebenfalls auf Teile und herrsche
- stammt von C. A. R. Hoare aus dem Jahre 1960
- Idee:
  - Wähle ein Pivotelement aus.
  - Bringe alle Werte, die kleiner als das Pivotelement sind, auf die linke Seite des Pivotelements und alle größeren Werte auf die rechte Seite.
  - Sortiere (rekursiv) die linke und rechte Seite.

Quicksort In-Place



- Wähle Pivotelement (hier 5) und vertausche Pivotelement mit dem Element am rechten Rand.
- Von links nach rechts bis vor das Pivotlement:
  - Wenn Element $\leq$ Pivotelement, dann tausche nach links.
  - Zielposition für einen Tausch ist zunächst der linke Rand, aber für jedes Element $\leq$ Pivotelement rückt die Zielposition um eins nach rechts.
- Tausche Pivotelement an die richtige Position.
- Sortiere (rekursiv) linken (blau) und rechten Teil (rot).

Quicksort: Algorithmus

```
private static void quicksort(int[] a, int il, int ir) {
    int imid = (il+ir)/2;      // Index Pivotelement
    int pivot;                 // Wert des Pivotelements
    int ipneu;                 // Index fuer Tausch und Endposition Pivotelement

    if (ir-il<=1) {            // nicht mehr als ein Element zu sortieren?
        return;                // ja, dann sind wir fertig.
    }

    pivot = a[imid];
    swap(a,imid,ir-1);         // vertausche Pivotelement mit dem rechten Rand

    ipneu = il;                // Ziel fuer Tauschvorgaenge liegt zunaechst am linken Rand
    for (int i=il; i < ir-1; i++) {
        if (a[i]<=pivot) {     // Ein Element gefunden, dass vom Pivotelement stehen muss?
            swap(a,ipneu,i);   // ja, dann tausche nach links
            ipneu++;           // Index fuer naechsten Tausch erhoehen
        }
```

```
    }
    swap(a,ir-1,ipneu);         // Pivotelement an die richtige Position platzieren

    quicksort(a,il,ipneu);      // alles links vom Pivotelement sortieren
    quicksort(a,ipneu+1,ir);    // alles rechts vom Pivotlelement sortieren
}
```

Quicksort: Eigenschaften

- Worst Case: Das Pivotelement ist z.B. stets das kleinste Element. Dann ist die linke Teilfolge leer und die rechte Teilfolge enthält nur ein Element weniger.
  Aufwand im Worst Case: $O(n^2)$
- Average Case: Das Pivotelement teilt die Folge in ungefähr gleich große Teilfolgen.
  Aufwand im Average Case: $O(n \log n)$
- Vergleich zu Mergesort:
  - Bei Mergesort fällt der Aufwand beim Zusammensetzen an, bei Quicksort bei der Zerlegung.
  - Bei gleichlangen Folgen ist die Zerlegung bei Quicksort effizienter als das Zusammensetzen bei Mergesort.
  - Dafür zerlegt Mergesort stets optimal, Quicksort nicht.
- In-Place Sortierung
- Quicksort ist nicht stabil.

8. Hashing

Lernziele:

- Hashverfahren verstehen und einsetzen können,
- Vor- und Nachteile von Hashing gegenüber Suchbäumen benennen können,
- verschiedene Verfahren zur Auflösung von Kollisionen kennen, deren Funktionsweise nachvollziehen und erläutern können und
- Speicherverfahren, die auf Hashing beruhen, in Java implementieren können.

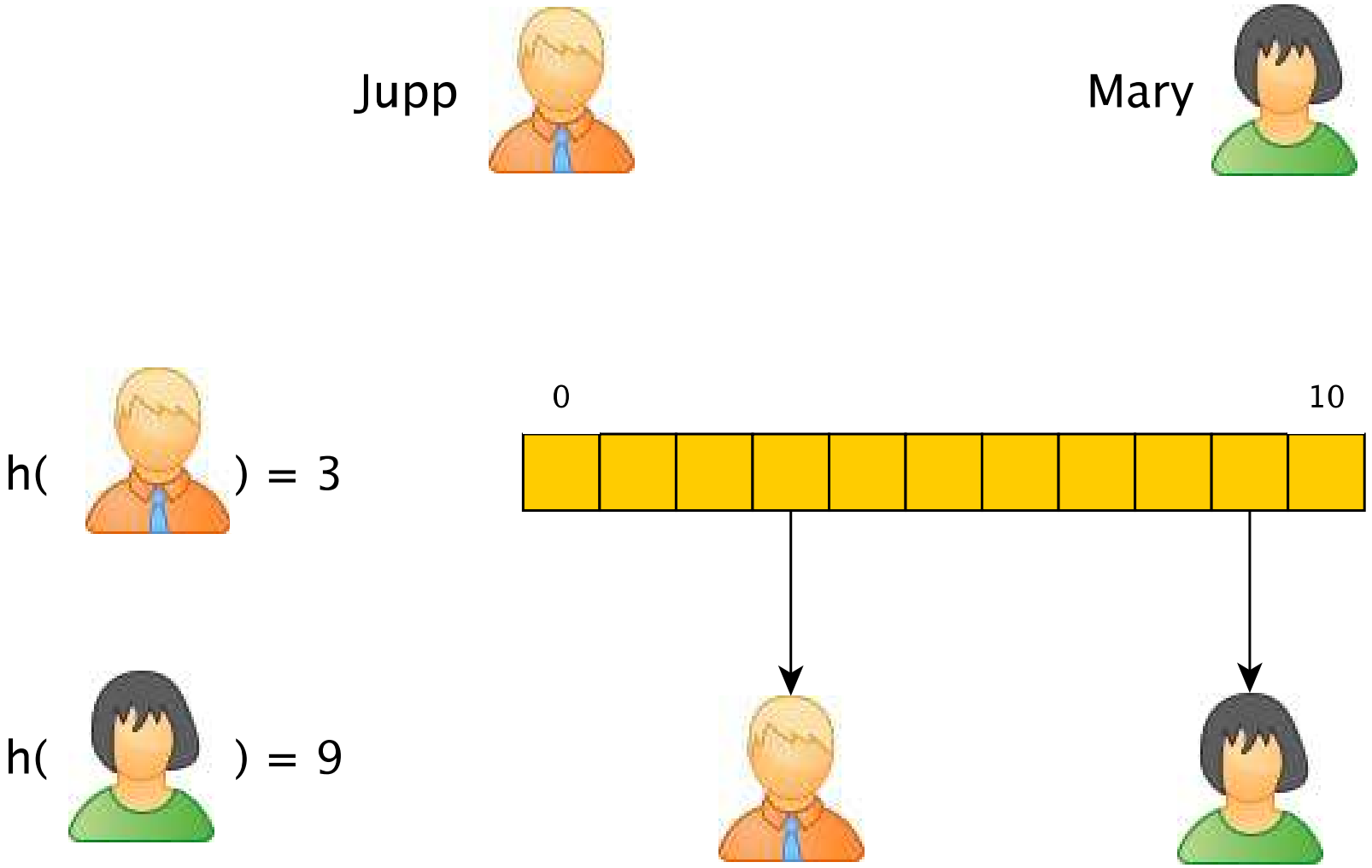
Idee des Hashing

- Hashing dient der Verwaltung einer Menge von Objekten vom Typ T (vgl. Folie 224).
- Objekte werden in einem "normalen" Feld mit direktem Zugriff gespeichert.
- Eine sogenannte Hashfunktion ermöglicht den direkten Zugriff auf ein Objekt.

☞ Statt einer ausgereiften Datenstruktur wie bei Bäumen benötigen wir hier eine ausgereifte Funktion zur Adressberechnung.

Grundprinzip

- Die Speicherung der Objekte erfolgt in einem Feld der Länge N, der sogenannten *Hashtabelle*.
- Eine Hashfunktion $h : T \to \{0, \ldots, N-1\}$ bestimmt für ein Objekt vom Typ T die Position des Objekts im Feld.
- Die Hashfunktion muss für eine "gute" Verteilung der zu speichernden Objekte in dem Feld sorgen.


Jupp
Mary
h( ) = 3
h( ) = 9
0
10

Kollisionen

Weil

- der Datentyp T (Menge der möglichen Objekte) in der Regel deutlich mehr als N Elemente enthält und
- die zu speichernden Objekte vorher unbekannt sind,

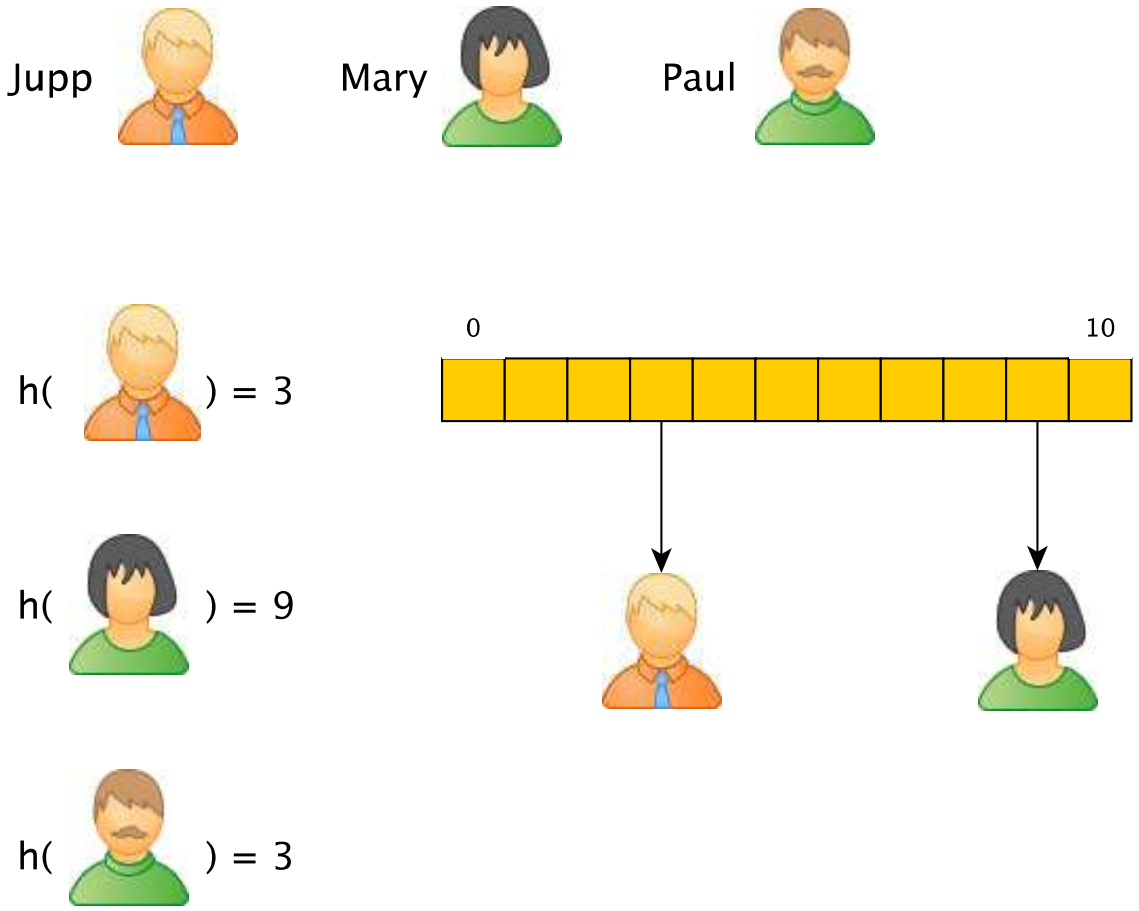
kann es zwangsläufig passieren, dass

- für zwei Objekte o_1 und o_2 mit $o_1 \neq o_2$
- $h(o_1) = h(o_2)$

gilt.

Solche eine Situation nennen wir *Kollision*.

Wohin mit Paul?



Anforderungen

- Die Hashfunktion sollte die zu speichernden Objekte möglichst gleichmäßig auf die Hashtabelle verteilen. So wird auch die Anzahl der Kollisionen gering gehalten.
- Wenn keine gleichmäßige Verteilung erfolgt, entartet das Hashverfahren, d.h. es kommt zu einer hohen Anzahl an Kollisionen.
- Da typischerweise $|T| > N$ gilt, lassen sich Kollisionen in der Regel nicht vermeiden.

Zentrale Fragen beim Hashing:

1. Was ist eine geeignete Hashfunktion?
2. Wie gehen wir mit Kollisionen um?

Anforderungen an eine Hashfunktion

- Effizienz der Berechnung
  Idealerweise in Zeit $O(1)$, insbesondere unabhängig von der Anzahl der Werte in der Hashtabelle
- Surjektivität
  $h(T) = \{0, \ldots, N-1\}$, so dass keine Plätze in der Hashtabelle leer bleiben.
- Kollisionsvermeidung bzw. Gleichverteilung
  Für zwei Objekte o_1 und o_2 sollte die Wahrscheinlichkeit, dass $h(o_1) = h(o_2)$ gilt, möglichst klein sein.
- Unstetigkeit
  Ähnliche Objekte sollten nicht zu ähnlichen Hashwerten führen.
- Effizienz des Speicherung
  Der Hashwert $h(o)$ eines Objektes $o \in T$ sollte möglichst kompakt gespeichert werden können.

Hashfunktionen für numerische Werte

Für nichtnegative ganze Zahlen wird oft der direkte Integer-Wert i in Verbindung mit dem Modulo-Operator als Hashfunktion genutzt.

$$h(i) = i \bmod N$$

- N (und damit die Größe der Hashtabelle) sollte eine große Primzahl sein.
- Für $2|N$ (d.h. 2 ist Teiler von N) wäre bspw. die Parität eine invariante Eigenschaft:
  - Wenn o_1 und o_2 die gleiche Parität hätten, dann auch $h(o_1)$ und $h(o_2)$.
  - Wenn es sich bei den Werten i z.B. um künstlich generierte Schlüssel handelt, die stets die gleiche Parität aufweisen, würde die Hälfte der Hashtabelle nicht belegt.
- Für $N = 2^k$ wäre $h(i)$ gleich dem Wert der letzten k Bits von i.
  - Problematisch z.B. bei der Verwendung von ungleich verteilten Prüfziffern.

- Vorsicht bei negativen Zahlen: Der `mod`-Operator entspricht in Java (und auch anderen Sprachen) nicht der mathematischen Definition.

  Mathematik: $-2 \bmod 3 = 1$, denn es gilt $-2 = (-1) \cdot 3 + 1$.
  Java: $-2 \% 3 = -2$, denn $-(2 \bmod 3) = -2$
- Für das Hashing von Fließpunktzahlen kann man z.B. Mantisse und Exponent addieren (oder auf andere Weise kombinieren).
- Da für die zu hashenden Integer-Werte meistens keine Gleichverteilung vorliegt, ist der `div`-Operator zur Konstruktion von Hashfunktionen ungeeignet.

  Beispiel: Matrikelnummern der Hochschule Bonn-Rhein-Sieg.

Hashfunktionen für Strings (1)

- ASCII-Werte der einzelnen Zeichen eines Strings nutzen. Für ein Zeichen $c \in \texttt{char}$ bezeichne $\texttt{ascii}(c)$ den ASCII-Wert des Zeichens c.
- Es sei $s = s[0]s[1] \cdots s[n-1]$ ein String der Länge n.
- Eine (meistens zu) einfache Hashfunktion:

$$h(s) = \left(\sum_{i=0}^{n-1} \texttt{ascii}(s[i]) \right) \mod N$$

Probleme:

- $h(s)$ liefert für alle Strings, die aus den gleichen Zeichen bestehen (sogenannte Anagramme), gleiche Hashwerte:
  - algorithmus und logarithmus

  - lager und regal
  - _regierung und genug_irre
- Bei kurzen Strings und großer Hashtabelle werden die hinteren Bereiche der Hashtabelle erst gar nicht erreicht. Es sei $N = 10000$:

$$\begin{aligned} h(\text{jupp}) &= 106 + 117 + 112 + 112 = 447 \\ h(\text{mary}) &= 109 + 97 + 114 + 121 = 441 \\ h(\text{paul}) &= 112 + 97 + 117 + 108 = 434 \end{aligned}$$

  Für Namen zwischen 3 und 10 Zeichen würde sich alles zwischen $3 \cdot \texttt{ascii}(\text{a}) = 291$ und $10 \cdot \texttt{ascii}(\text{z}) = 1220$ ballen.

Hashfunktionen für Strings (2)

Häufig nutzt man für ein festes $x \in \mathbb{N}$ die Hashfunktion:

$$h(s) = \left(\sum_{i=0}^{n-1} \texttt{ascii}(s[i]) \cdot x^{n-i-1} \right) \mod N$$

- Polynom vom Grad $n-1$ mit den ASCII-Werten der Zeichen $s[i]$ als Koeffizienten, ausgewertet an der Stelle x.
- Für $x = 1$ ist dies die Hashfunktion von Folie 333.
- Für $x = 2$ ergibt sich

$$\begin{aligned} h(\text{jupp}) &= 106 \cdot 8 + 117 \cdot 4 + 112 \cdot 2 + 112 = 1652 \\ h(\text{mary}) &= 109 \cdot 8 + 97 \cdot 4 + 114 \cdot 2 + 121 = 1609 \\ h(\text{paul}) &= 112 \cdot 8 + 97 \cdot 4 + 117 \cdot 2 + 108 = 1626 \end{aligned}$$

was schon etwas stärker streut.

- Beliebt ist $x = 2^8$, auch wegen effizienter Berechnung durch Bitoperationen (s.u.):

$$\begin{aligned} h(\text{jupp}) &= (106 \cdot 2^{24} + 117 \cdot 2^{16} + 112 \cdot 2^8 + 112) \bmod 10000 = 1392 \\ h(\text{mary}) &= (109 \cdot 2^{24} + 97 \cdot 2^{16} + 114 \cdot 2^8 + 121) \bmod 10000 = 2841 \\ h(\text{paul}) &= (112 \cdot 2^{24} + 97 \cdot 2^{16} + 117 \cdot 2^8 + 108) \bmod 10000 = 5244 \end{aligned}$$

  Aber wegen der Zweierpotenz ist dies auch nicht ganz unproblematisch.

- Java nutzt zur eingebauten Berechnung von Hashwerten (Methode `hashCode()`) für Strings $x = 31 = 2^5 - 1$. Dies lässt sich ebenfalls effizient berechnen (s.u.).

Effiziente Berechnungen beim String-Hashing

Der Term

$$\sum_{i=0}^{n-1} \texttt{ascii}(s[i]) \cdot x^{n-i-1}$$

der Hashfunktion wird nicht via Definition berechnet, sondern über das sogenannte *Horner-Schema*.

Auswertung nach dem Horner-Schema:

- Gegeben sei das Polynom

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Dies ist identisch zu der folgenden Form

$$f(x) = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0$$

- Beispiel:

$$\begin{aligned} f(x) &= 4x^3 + 5x^2 + x + 2 \\ &= ((4x + 5)x + 1)x + 2 \end{aligned}$$

- Vorteil: Es müssen keine Potenzen gebildet werden, nur $n - 1$ statt $2n - 1$ Multiplikationen
- Java-Methode zur Polynomauswertung:

```
public static double f(double[] a, double x) {
    double fx = 0.0;                // Funktionswert f(x) an der Stelle x

    for (int i=a.length-1 ; i>=0 ; i--) {
        fx = fx * x + a[i];
    }
    return fx;
}
```

Weitere Optimierungen

- Für z.B. $x = 256 = 2^8$ oder $x = 31 = 2^5 - 1$ kann die Berechnung durch Bitoperationen (siehe EidP, Folien 119–124) noch effizienter erfolgen.
- Die Multiplikation mit 2^8 entspricht einer Verschiebung aller Bits um acht Positionen nach links. Operator <<
- Nach der Verschiebung haben die acht niedrigstwertigen Bits alle den Wert 0.
- Eine Addition mit einer Zahl $0 \leq a \leq 255$ entspricht dann einer bitweisen Oder-Verknüpfung. Operator |

```
public static int f(char[] s) {
    int hash = 0;

    for (int i=0 ; i<s.length ; i++) {
        hash = (hash << 8) | s[i];
    }
    return hash;
}
```

- Die Multiplikation eines Wertes i mit 31 lässt sich durch eine Linksverschiebung um 5 Bits und eine Subtraktion von i ausdrücken.

```
i = (i << 5) - i;
```

- Manche Prozessoren benötigen für die gesamte Berechnung nur eine Instruktion (z.B. ARM).
- Insgesamt:

```
public static int f(char[] s) {
    int hash = 0;

    for (int i=0 ; i<s.length ; i++) {
        hash = (hash << 5) - hash + s[i];
    }
    return hash;
}
```

Hashfunktionen für sonstige Objekte

Für Objekt o vom Typ T bestehend aus Teilobjekten o_1 bis o_k mit Typen $T_1, \ldots, T_k$:

- Nutze existierende Hashfunktionen h_i für die Typen T_i und
- aggregiere die einzelnen Hashwerte auf angemessene Weise, z.B. durch Summenbildung.

$$h(o) = \left(\sum_{i=1}^{k} h_i(o_i) \right) \mod N$$

Ansätze zur Behandlung von Kollisionen

Verkettung: Objekte mit gleichem Hashwert werden in einer verketteten Liste gespeichert.

- Die Hashtabelle enthält also nicht die eigentlichen Objekte sondern Verweise auf Listen, die die Objekte enthalten.

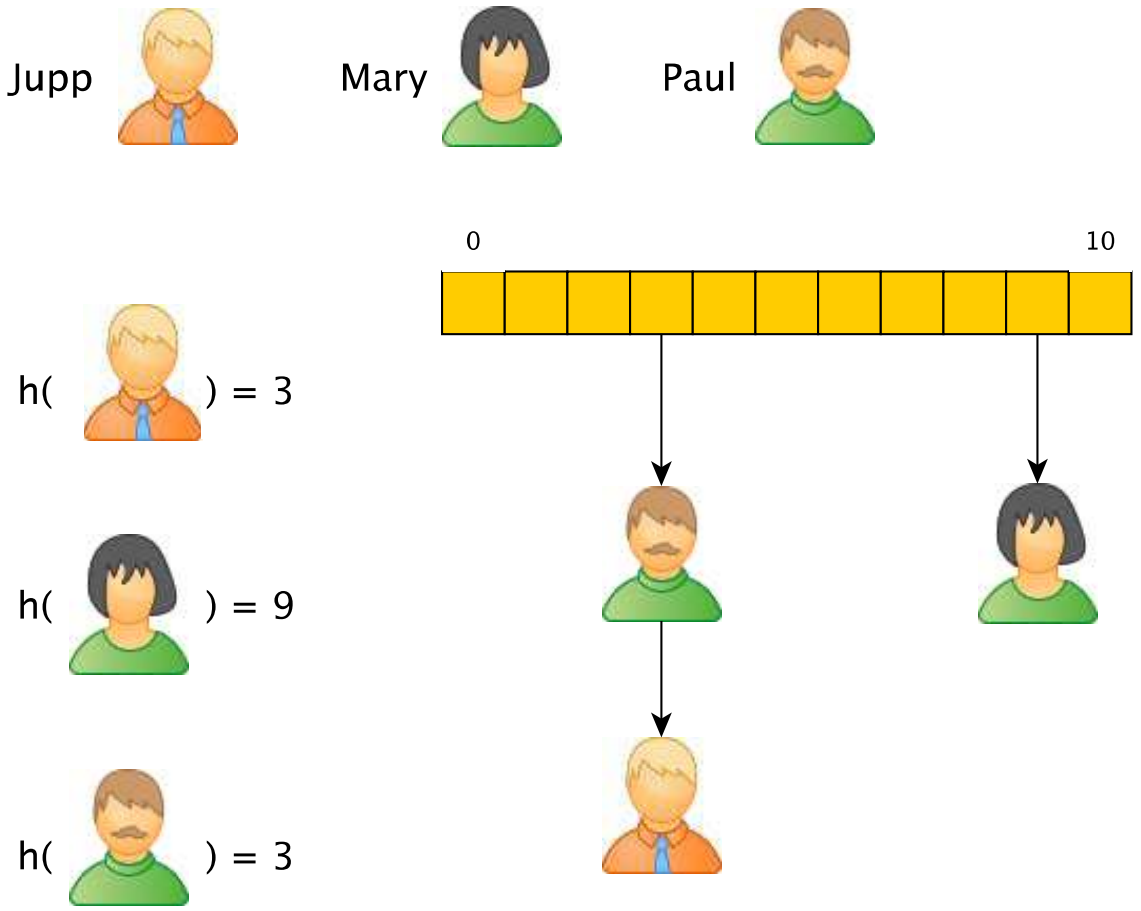
Offene Adressierung: Es wird nach einer alternativen Position in der Hashtabelle gesucht.

- Beim Sondieren geschieht dies durch eine vordefinierte Folge von Offsets relativ zur eigentlichen Position.
- Beim mehrfachen Hashen werden weitere Hashfunktionen für eine alternative Positionsbestimmung genutzt.

Verkettung

- Unter einem Überläufer versteht man ein Objekt, das zu einer Kollision führt.
- Bei der Verkettung wird das Objekt an einer Position der Hashtabelle zusammen mit allen Überläufern in einer verketteten Liste gespeichert.
- Die Verkettung von Überläufern kann zu einer linearen Liste "entarten", wenn die Mehrzahl der Elemente durch die Hashfunktion auf dieselbe Position abgebildet wird.
- Zur Verhinderung der Entartung kann statt einer Liste daher auch ein ausgeglichener Suchbaum verwendet werden.

Dahin mit Paul!



Hashing mit Verkettung in Java

```
public class HashSet<T> {

    private static final int N = ...;

    private class Item {       // Element der verketteten Liste besteht aus
        T    value;            // Verweis auf gespeichertes Objekt und
        Item next;             // Verweis auf naechstes Element der Liste
    }

    private Item[] hashTable = new Item[N];     // die Hashtabelle

    public void insert(T elem) {
        int  i = elem.hashCode() % N;     // setzt hashCode() >= 0 voraus
        Item item = new Item();

        item.value = elem;                // neues Objekt wird das erste
        item.next = hashTable[i];         // in der verketteten Liste
        hashTable[i] = item;
```

```
    }

    public boolean contains(T elem) {
        Item item = hashTable[elem.hashCode() % N];  // setzt hashCode() >= 0 voraus
                                                     // nach Element suchen
        while (item != null && !item.value.equals(elem)) {
            item = item.next;
        }
        return item != null;          // wenn gefunden, dann ist item != null
    }

    public void remove(T elem) {
        int  i = elem.hashCode() % N;
        Item item = hashTable[i];          // setzt hashCode() >= 0 voraus
        Item pre = null;                   // Vorgaenger in Liste
                                                     // nach Element suchen
        while (item != null && !item.value.equals(elem)) {
            pre = item;
            item = item.next;
        }
        if (item != null) {                // nur wenn gefunden
            if (pre == null) {             // wenn erstes Element in Liste
```

```
                hashTable[i] = item.next; // dann Eintrag in Hashtabelle mit Nachfolger belegen
            }
            else {                        // ansonsten
                pre.next = item.next;     // Vorgaenger auf Nachfolger zeigen lassen
            }
        }
    }
}
```

Sondieren

- Kommt es zu einer Kollision, dann suchen wir für den Überläufer nach einer anderen, noch unbesetzten Position in der Hashtabelle gesucht.
- Das Suchen nach solch einer Position bezeichnen wir als *Sondieren*.
- Für das Sondieren nutzt man eine Offset-Folge $\text{offset}(i)$ für $i = 1, 2, \ldots$
- Mit der Offset-Folge wird für die Positionen

$$(h(o) + \text{offset}(i)) \bmod N$$

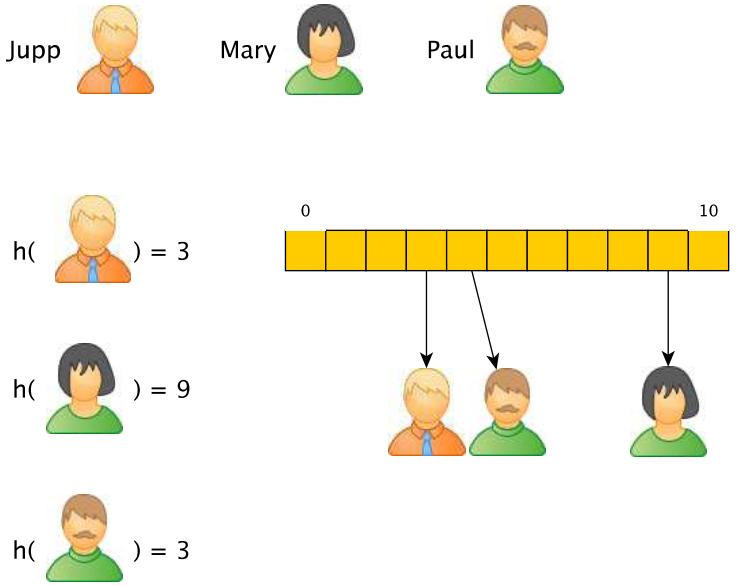
  geprüft, ob sie noch frei sind.
- Beim kleinsten i mit einem freien Platz wird das Objekt o in die Hashtabelle eingefügt.

Lineares Sondieren

- Beim Linearen Sondieren gilt

$$\text{offset}(i) = i$$

  Es wird also linear nach dem nächsten freien Platz gesucht.
- Problem: Dadurch kann es leicht zu Ballungen um belegte Plätze herum kommen.



Beispiel: Lineares Sondieren

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

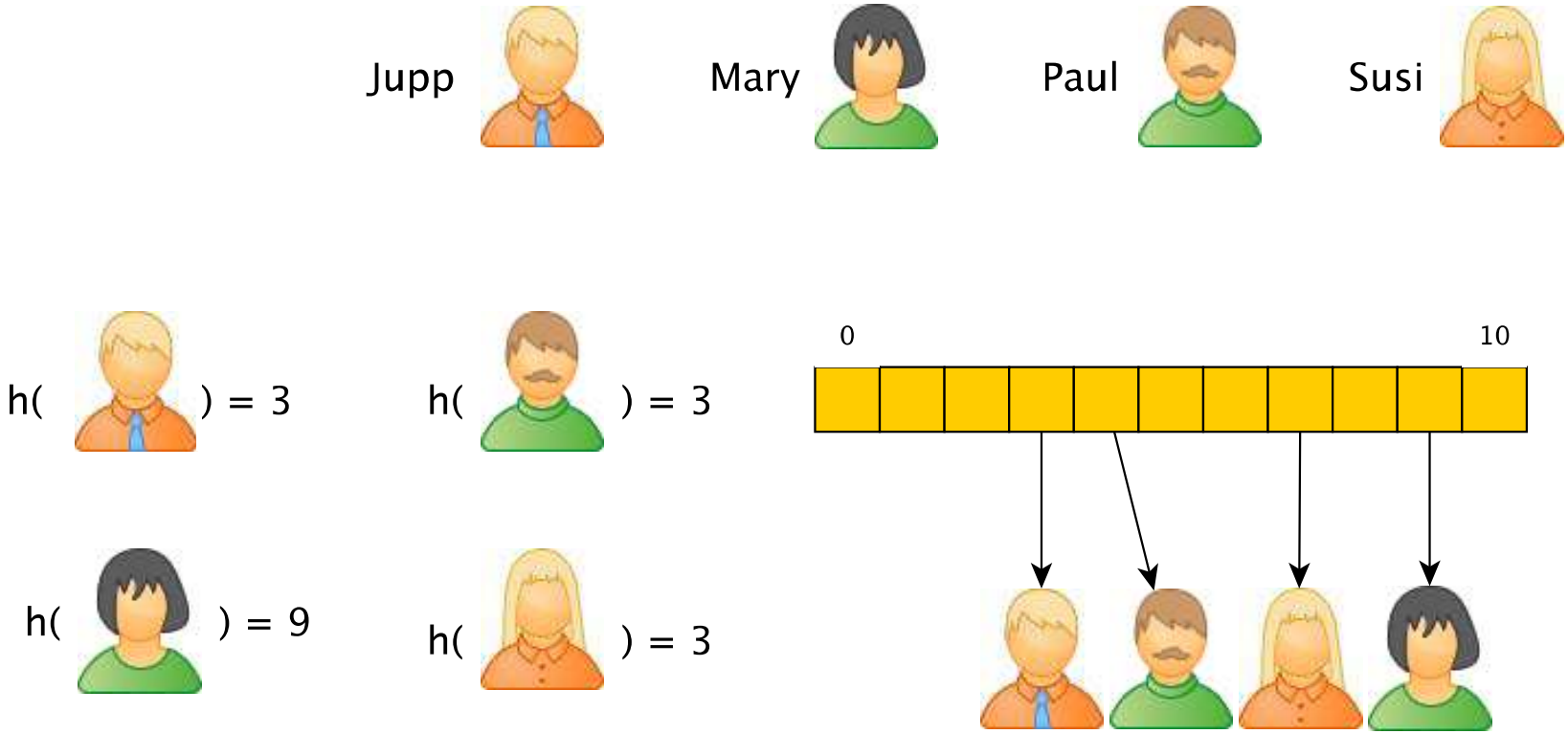
Quadratisches Sondieren

- Beim Quadratischen Sondieren gilt

$$\text{offset}(i) = i^2$$

- Vorteil: Ballungen um belegte Plätze sind weniger wahrscheinlich.
- Nachteil: Beim Sondieren werden u.U. nicht alle Plätze der Hashtabelle berücksichtigt.
  - Ist bspw. N eine Quadratzahl, so werden beim quadratischen Sondieren u.U. nur $\sqrt{N} - 1$ alternative Positionen der Hashtabelle untersucht. Beispiel: 16
  - Wenn N eine Primzahl ist, kann aber garantiert werden, dass mindestens $N/2$ Positionen untersucht werden (s.u.).

Dahin mit Paul und Susi!



Beispiel: Quadratisches Sondieren

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Satz zum quadratischen Sondieren

Satz 8.1. *Es sein* N *die Größe einer Hashtabelle.*

Wenn N *eine Primzahl ist, dann werden beim quadratischen Sondieren mindesten* $N/2$ *alternative Positionen geprüft.*

Beweis: Es sei j die kleinste natürliche Zahl, für die beim quadratischen Sondieren eine schon untersuchte Position wieder untersucht wird. Es sei i diejenige natürliche Zahl, die zur ersten Untersuchung dieser Position gehört.

Dann gilt:

$$\begin{aligned} \text{offset}(i) \bmod N = \text{offset}(j) \bmod N \quad &\Leftrightarrow \quad i^2 \bmod N = j^2 \bmod N \\ &\Leftrightarrow \quad j^2 - i^2 \bmod N = 0 \\ &\Leftrightarrow \quad (j-i) \cdot (j+i) \bmod N = 0 \end{aligned}$$

- Daraus folgt, dass $(j - i) \cdot (j + i)$ gleich N oder ein Vielfaches von N sein muss, d.h.

$$\exists c \in \mathbb{N} : (j - i) \cdot (j + i) = c \cdot N$$

- Es gilt $j - i < N$, denn es gibt nur $N - 1$ alternative Positionen.
- Somit muss, da N eine Primzahl ist, $(j + i)$ gleich N oder ein Vielfaches von N sein.
- Wegen $j > i$ folgt hieraus $j > N/2$.

Diskussion Sondieren

Das Sondieren verkompliziert Such-, Einfüge- und Löschoperationen:

- Beim Suchen nach Objekt o: Befindet sich an Position $i = h(o)$ ein Objekt $o' \neq o$, dann wurde o mittels Sondieren u.U. an einer anderen Position platziert. Wir müssen bei der Suche also ebenfalls Sondieren.

  Bemerkung: Wenn der Eintrag an der Stelle i der Hashtabelle leer ist, dann kann o nicht enthalten sein.
- Beim Löschen von Objekt o: Ein einfaches Löschen könnte die Suchkette für andere Elemente zerstören.

  Deshalb: nicht Löschen, sondern nur als gelöscht markieren und Platz beim Einfügen wiederverwenden.

☞ Sondierung bzw. offene Adressierung nur dann sinnvoll, wenn Löschen selten ist.

Hashing in Java: Die Methode `hashCode()`

- In Java verfügt jedes Objekt über eine Methode `hashCode()` zur Berechnung eines Hash-Wertes.
- Durch Bitverschiebung oder einen Überlauf bei den Integer-Operationen können auch negative Hashwerte als Resultat der Methode `hashCode()` entstehen (vgl. Hashfunktionen für Strings).
- Auch die Modulo-Operation zur Abbildung auf die Hashtabelle wird nicht durch `hashCode()` durchgeführt.
- In der Klasse `Object` wird der Hashwert auf Basis der Speicheradresse ermittelt.
- Für die Definition eigener Hashfunktionen überschreibt man in den betreffenden Klassen die Methode `hashCode()`.

Hashing in Java: Collection Classes

Alles im Paket `java.util`:

- Klasse `Hashtable<K,V>`
  - Bildet mit Hilfe einer Hashtabelle Schlüssel vom Typ `K` auf Objekte vom Typ `V` ab. Interface für solch eine Art der Abbildung: `Map<K,V>`
  - Der Typ `K` muss die Methoden `hashCode()` und `equals()` geeignet implementieren.
  - `null` ist nicht erlaubt, sowohl als Schlüssel als auch als Objekt.
  - Die Hashtabelle wird automatisch vergrößert, wenn sie zu voll wird. Größe und Vergrößerung können durch Parameter gesteuert werden.
  - Ist `synchronized`, d.h. Parallelverarbeitung wird unterstützt.
- Klasse `HashMap<K,V>`
  - Leistet vom Prinzip das gleiche wie `Hashtable`.
  - Erlaubt im Gegensatz zu `Hashtable` aber `null` als Schlüssel bzw. als Objekt.
  - Unterstützt keine Parallelverarbeitung.

- Klasse `HashSet<E>`
  - Realisierung einer Menge von Objekten des Typs `E` mit Hilfe einer Hashtabelle.
  - Implementiert die Schnittstelle `Set<E>`.
  - Basiert auf `HashMap`.