PGdP

Überladen

Überschreiben

Generics

Interfaces

Anonyme Klassen

Lamda Ausdrücke

Polymorphie II

P-Aufgaben

Folien: go.tum.de/904005

Tobias Schamel

Überladen

Überladen von Methoden: Mehrere Methoden mit dem selben Namen aber unterschiedlicher Signatur:

```
void steerLeft() {
  this.degree = -45;
}
void steerLeft(double by) {
  this.degree -= by;
}
```

Es wird die Methode ausgeführt, zu der die Signatur des Aufrufes passt.

Überladen

Überladen von Konstrukoren: Mehrere Konstruktoren mit unterschiedlichen Parametern:

```
public Rational(int zähler, int nenner) {
  this.zähler = zähler;
  this.nenner = nenner;
}
//für Ganzzahlen
public Rational(int zähler) {
  this.zähler = zähler; this.nenner = 1;
}
```

Überschreiben

Überschreiben von Methoden: Mehrere Methoden mit dem selben Namen und der selben Signatur in unterschiedlichen Klassen (in der Hierarchie):

```
void beschleunigen(double wert) { //in Fahrzeug
  this.geschwindigkeit += wert;
}
void beschleunigen(double wert) { //in Auto
  if(this.tank > 0) this.geschwindigkeit += wert;
}
```

Siehe Polymorphie.

? super T

Alle Superklassen/Oberklassen

Generics

Generics sind 'Variablen für Datentypen'.

```
class ListElement<T> {
  ListElement next;
  T info;
}
```

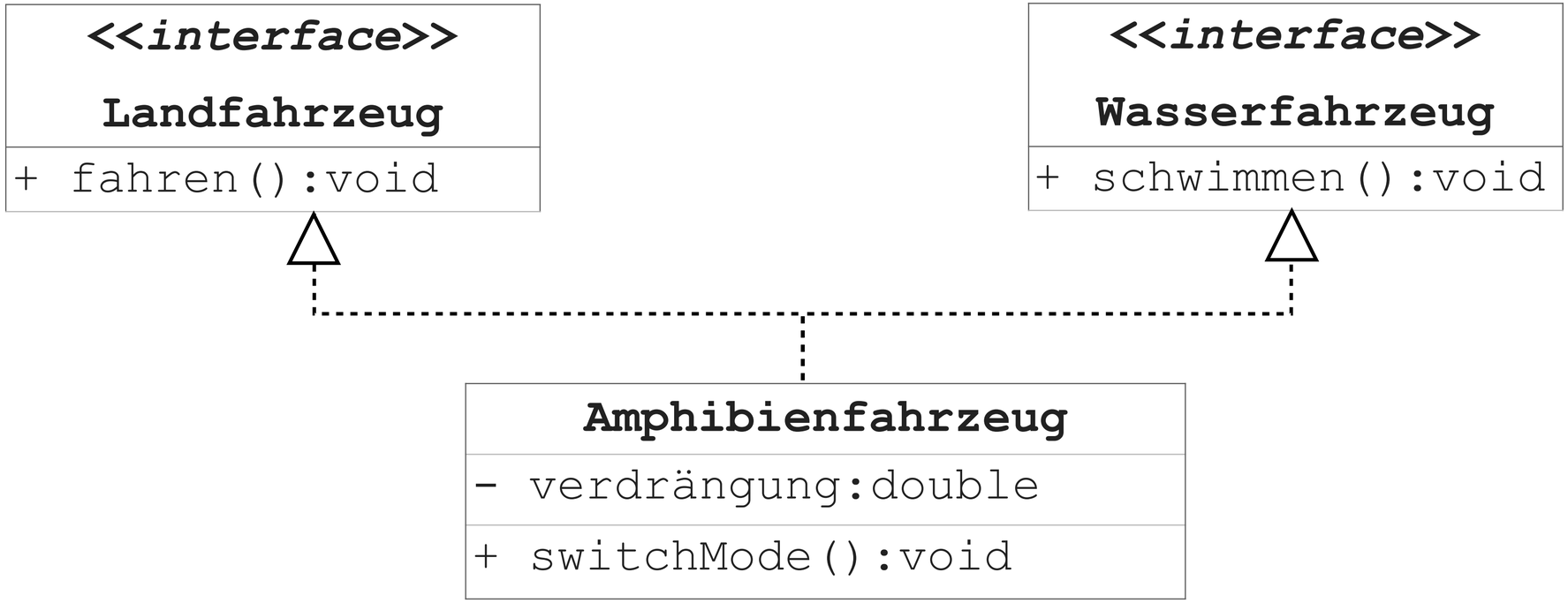
Datentyp kann zur Laufzeit festgelegt werden

Jede Unterklasse muss alle Typargumente in der Oberklassen/den Interfaces binden:

- A<S,T> extends B<T> ✓
- A<S> extends B<S,T> ✗

Interfaces

Interfaces stellen eine Möglichkeit dar, mehrere 'Oberklassen' festzulegen.



Interfaces

Interfaces stellen eine Möglichkeit dar, r
'Oberklassen' festzulegen.

Erzwingt konkrete Implementierung in implementierender Klasse.


```
public interface Wasserfahrzeug
  /* keine Attribute (nur
     final oder static) */
  //kein Konstruktor möglich > keine Instanziierung
  public abstract void schwimmen();
  //nur abstrakte Methoden?
}
```

```
public class Amphibienfahrzeug implements
Wasserfahrzeug, Landfahrzeug { /* ... */ }
```

Interfaces

Interfaces stellen eine Möglichkeit dar, r
'Oberklassen' festzulegen.

Seit Java 8 gibt es 'default' Methoden in Interfaces.

```
public interface Wasserfahrzeug
   /* keine Attribute (nur
      final oder static) */
   //kein Konstruktor möglich > keine Instanziierung
   public default void schwimmen() {
     write("Fahrzeug schwimmt!");
   }
}
public class Amphibienfahrzeug implements
Wasserfahrzeug, Landfahrzeug { /* ... */ }
```

Interfaces

Interfaces können wie statische Datentypen verwendet werden.

```
public static void main(String[] args) {
   Wasserfahrzeug w1 = new
    Amphibienfahrzeug();

   w1.schwimmen();
   w1.fahren(); //Compilerfehler
   ((Amphibienfahrzeug) w1).fahr
   ((Landfahrzeug) w1).fahren();
}
```

Statischer Typ bestimmt aufrufbare Methoden.

Interfaces

Interfaces vs. abstrakte Oberklassen

Eigenschaft	Interface	abstrakteKlasse
Polymorphie	✓	✓
Attribute	✗, nur `static` und `final`	✓
Methodenimplentierung	(✓)	✓
Interfaces implementieren	✓, mit `extends`	✓
von Klassen erben	✗	✓

Interfaces

Interfaces in der Java Standardbibliothek:

Iterable<T>, Comparable<T>, etc.

> Iterarable<T>: `iterator():Iterator<T>`

```
Iterable l1 = new
  java.util.ArrayList<Integer>();
 //fill list
 for(Integer current : l1)
   write(current);
```

ForEach Schleife

(braucht Iterable)

Anonyme Klassen

Selten oder gar nicht wiederverwendete Klassen werden häufig als anonyme Klassen geschrieben.

```
Tree<Penguin> tree = new Tree<Penguin>(
  new Comparator<Penguin>() {
    // Implementierung
    public int compareTo(Penguin o1, Penguin o2) {
      return o1.age - o2.age;
    }
  }
);
```

Lamda Ausdrücke

Lamda Ausdrücke können wie ein Methodenargument statt einer anonymen Klasse übergeben werden, falls klar ist, welche Methode & welche Parameter (> nur eine Methode).

```
Tree<Penguin> tree =
 new Tree<Penguin>((o1, o2) -> o1.age - o2.age);
```

Parameter — Methodenkörper, implizit auch return Wert

```
//für Konstruktor Tree<T>(Comparator comp)
```

Mehr dazu bei Streams.

Lamda Ausdrücke

Ein weiteres Beispiel:

```
interface Rechteck {
  public abstract double getArea(
   double w, double h);
}
public static void main(String[] args) {
  Rechteck r1 = ((w, h) -> {double area = w*h;
    return area;
  });
}
```

Polymorphie I

Eine Instanz einer Unterklasse kann auch im statischen Referenztyp der Oberklasse gespeichert werden.

```
Fahrzeug a1 = new Auto(0, 1000, 20);
Auto a2 = new Auto(0, 1200, 15);
```

Statischer Typ — Dynamischer Typ

Verfügbare Methoden & Attribute werden durch den statischen Typen eines Objekts bestimmt.

```
((Auto) a1).tanken(20);  //Compilerfehler
a2.tanken(20);
```

Polymorphie II

Welche Methode wird nach dem Überschreiben von der JVM ausgewählt? Aufruf von: $e_0 . f(e_1, \ldots, e_k)$.

1. Bestimmte die statischen Typen $s(e_0) = T_0, \ldots, s(e_k) = T_k$
2. Suche in den Oberklassen & in der Klasse von T_0 eine Methode, deren Signatur bestmöglich zu $T_1, \ldots, T_k$ passt
3. Der Compiler merkt sich die Methode bzw. deren Signatur $I_1, \ldots, I_k$
4. Zur Laufzeit wird eine passende Methode mit der vorgemerkten Signatur $I_1, \ldots, I_k$ in den Oberklassen und der Klasse des dynamischen Typen $d(e_0)$ ausgeführt

Polymorphie II

Beispiel: Polymorphie (immer Klausuraufgabe)

```
class A {
  void f(A a, A b) { write("A.f(AB)"; }
  static void g(A a, A b) { write("A.g(AB)"); }
}
class B extends A {
  void f(A a, A b) { write("B.f(AB)"); }
}
A a = new A(); A b = new B();
b.f(a, b);          b.g(a, b);
```

Stat. A, Dyn. B

Stat. A

Polymorphie II

Beispiel: Polymorphie (immer Klausuraufgabe)

```
class A {
   void f(A a, A b) { write("A.f(AB)"; }
}
class B extends A {
   void f(A a, A b) { write("B.f(AB)"; }
}
A a = new A(); A b = new B();
b.f(a, b);
```

Statisch: A
Dynamisch: B

Compiler wählt Methode mit dieser Signatur

Polymorphie II

Beispiel: Polymorphie (immer Klausurau

```
class A {
  void f(A a, A b) { write("A.f
}
class B extends A {
  void f(A a, A b) { write("B.f(AB)"); }
}
A a = new A(); A b = new B();
b.f(a, b);
```

Statisch: A
Dynamisch: B

Zur Laufzeit wird die am Besten zum dyn. Typ passende Methode mit der gewählten Signatur ausgeführt.

Andere Methode
⇒ dynamic Dispatch

P08.01

Polymorphie

Für jeden Aufruf geben wir an:

- Aufruf (Nummer & Zeilennummer)
- Statischer Typ des Objekts (auf dem aufgerufen wird)
- Signatur des Methodenaufrufs
- Zeilennummer aller möglichen Methoden
- Vom Compiler gewählte Methodensignatur (Zeilenummer)
- Begründung: eindeutig, speziellste Signatur
- Dynamischer Typ des Objekts (auf dem aufgerufen wird)
- Dynamisch gewählte Signatur (Zeilennummer)
- Begründung: static, kein Dispatch, statischer. Typ = dynamischer Typ
- Ausgabe der Methode

P08.01

Polymorphie

Für jeden Aufruf geben wir an:

- Aufruf (Nummer & Zeilennummer)
- Statischer Typ des Objekts (auf dem aufgerufen wird)
- Signatur des Methodenaufrufs
- Zeilennummer aller möglichen Methoden
- Vom Compiler gewählte Methodensignatur (Zeilenummer)
- Begründung: eindeutig, speziellste Signatur
- Dynamischer Typ des Objekts (auf dem aufgerufen wird)
- Dynamisch gewählte Signatur (Zeilennummer)
- Begründung: [static, kein Dispatch], statischer. Typ = dynamischer Typ
- Ausgabe der Methode

Dispatch: Auswahl einer dynamisch besser passenden Methode als die des statischen Typs.

P08.02

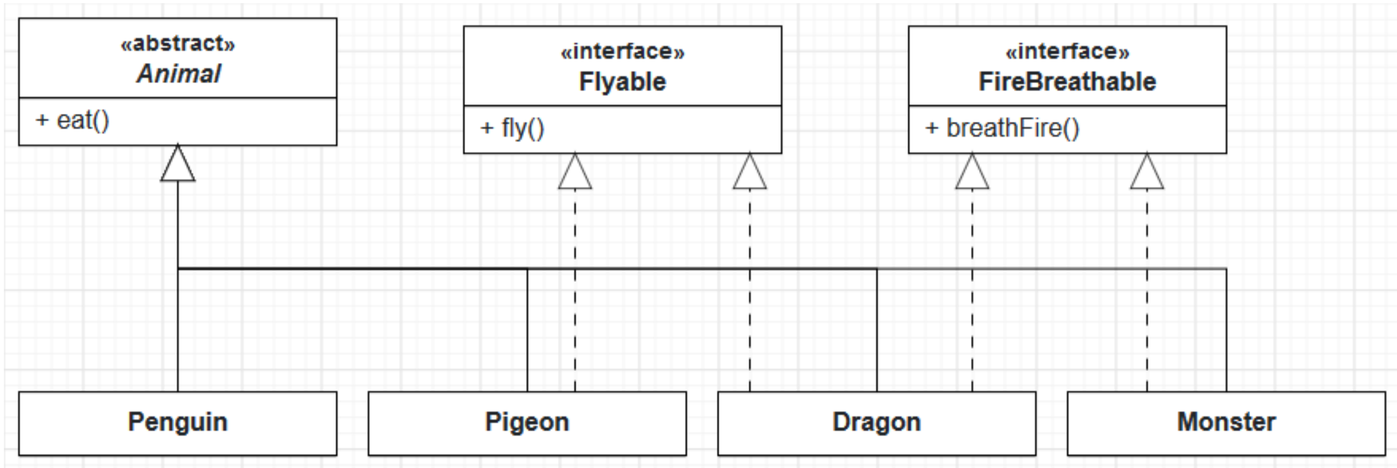
Fabelwesen

Penguin: eat

Pigeon: eat, fly

Dragon: eat, fly, breathFire

Monster: eat, breathFire

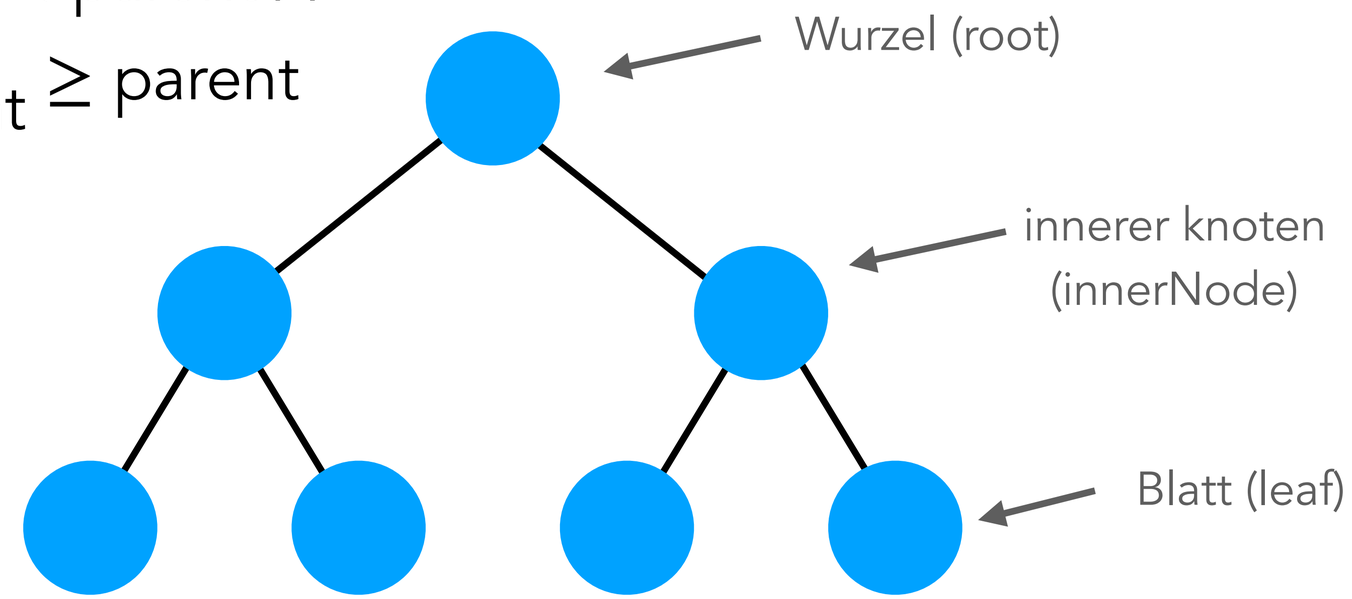


P08.03

Binärbäume

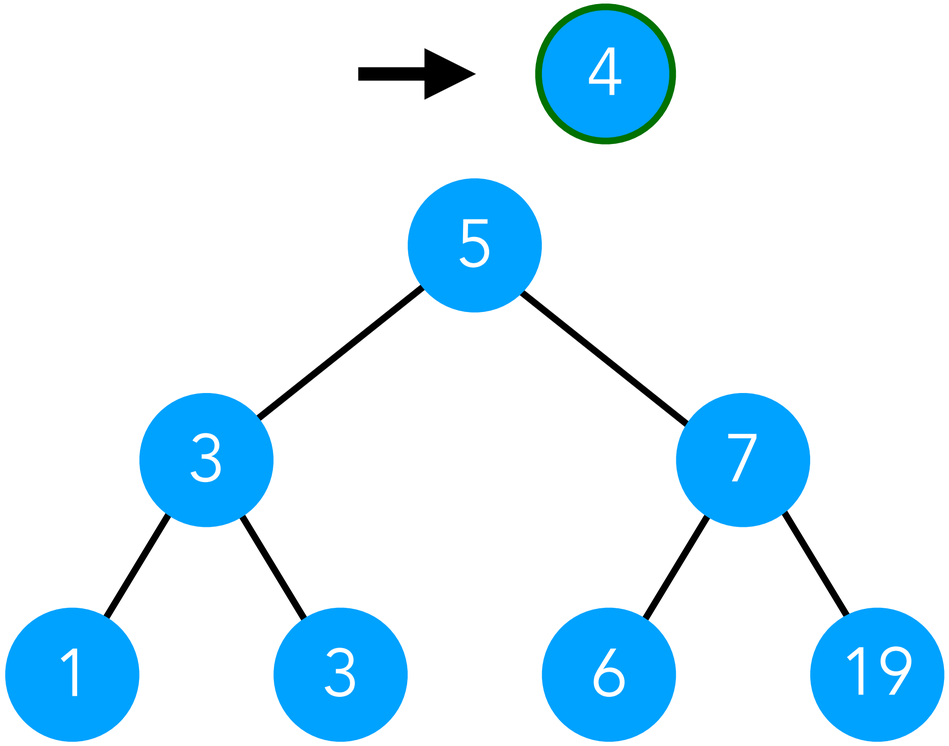
Invariante für sortierte Bäume

$$\text{parent}_{\text{left}} < \text{parent} \land$$
$$\text{parent}_{\text{right}} \geq \text{parent}$$



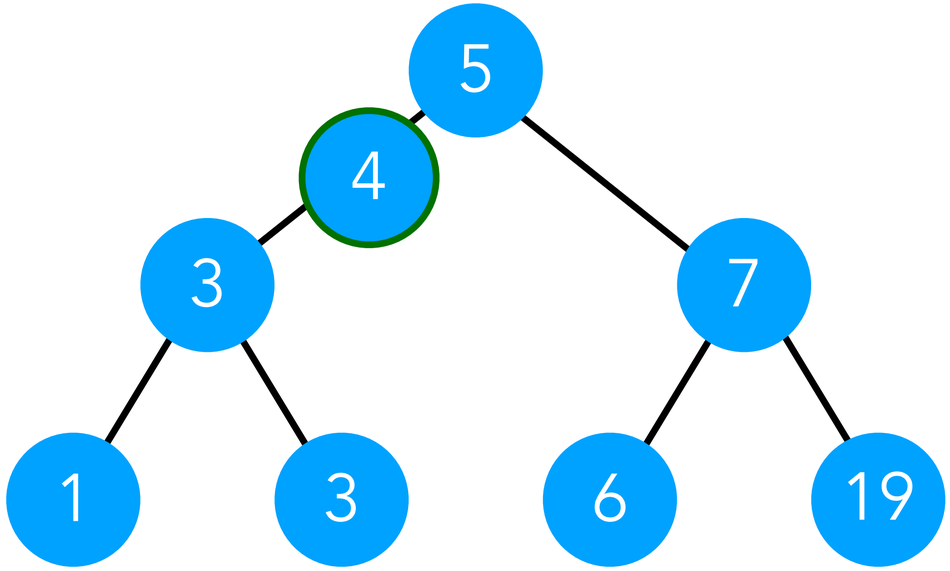
P08.03

Binärbäume, Einfügen



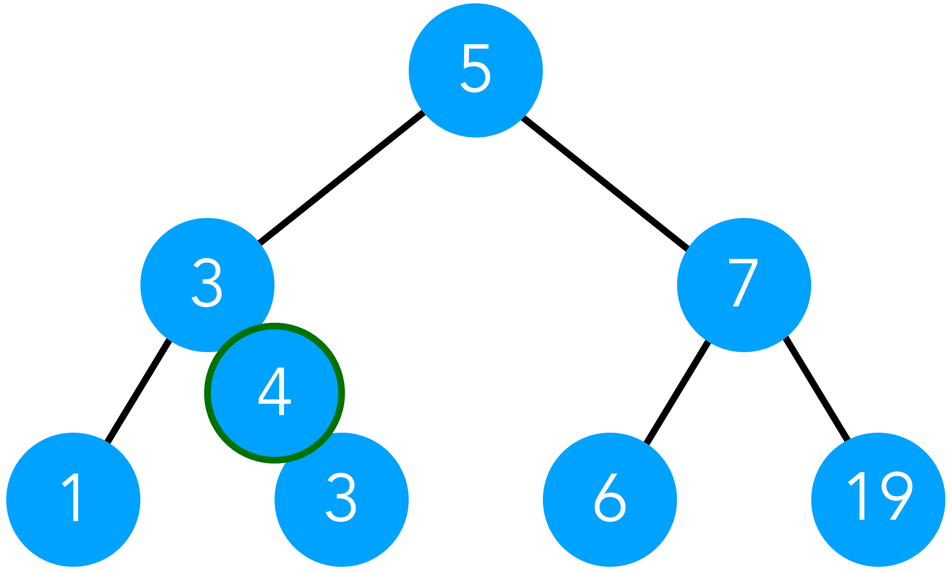
P08.03

Binärbäume, Einfügen



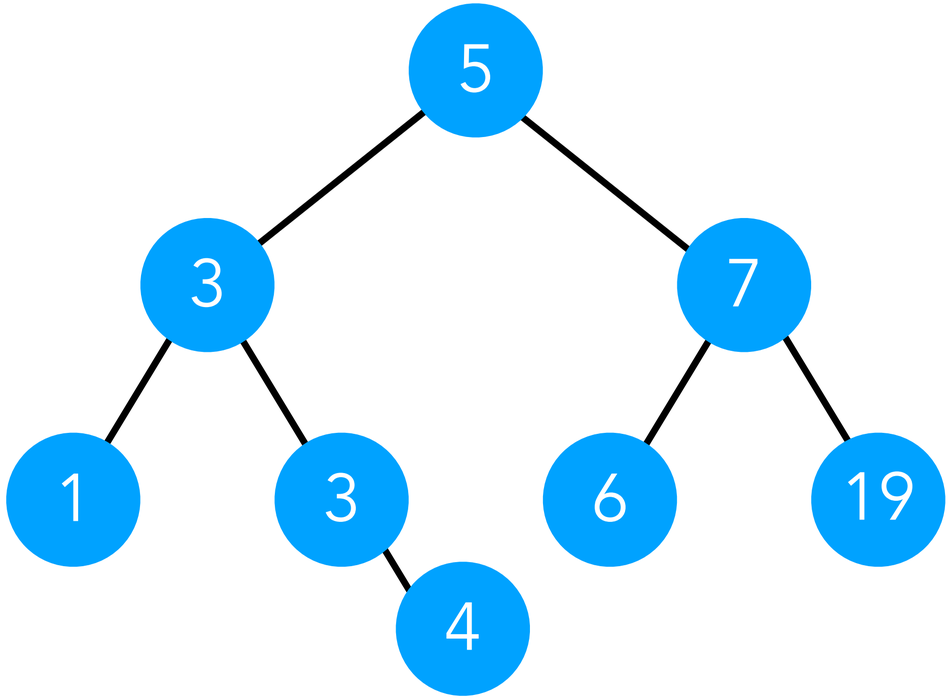
P08.03

Binärbäume, Einfügen



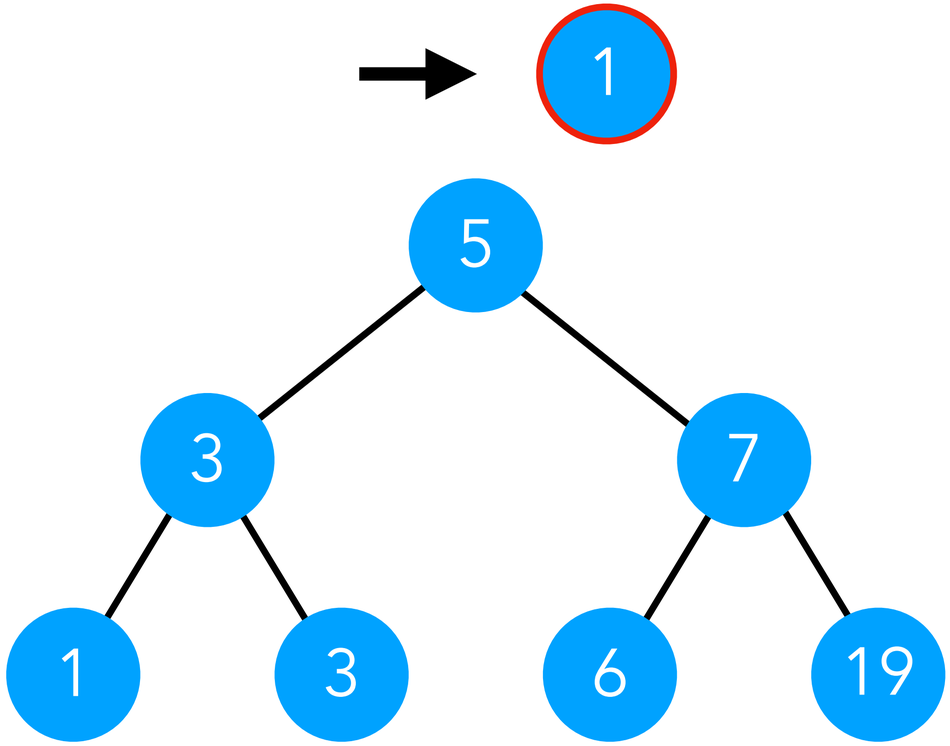
P08.03

Binärbäume, Einfügen



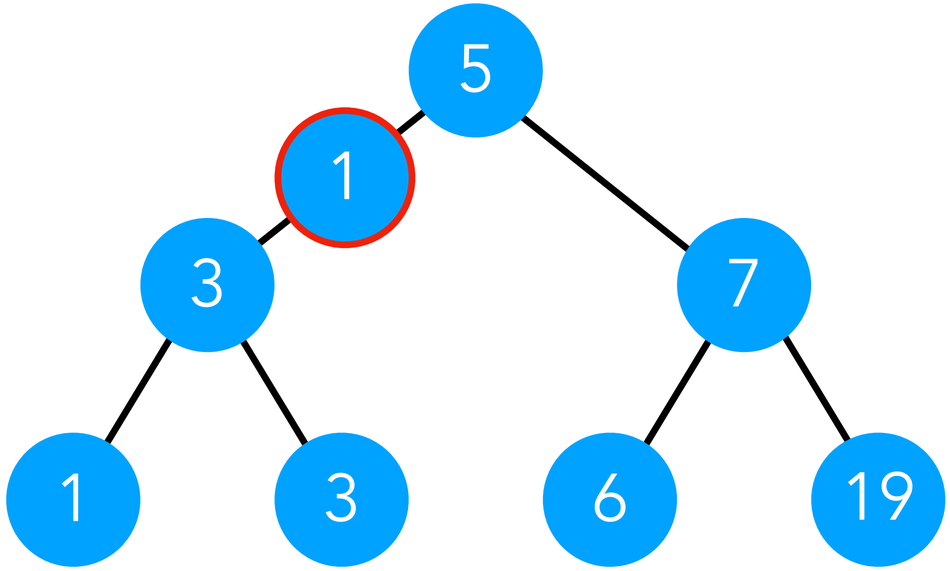
P08.03

Binärbäume, Löschen (1)



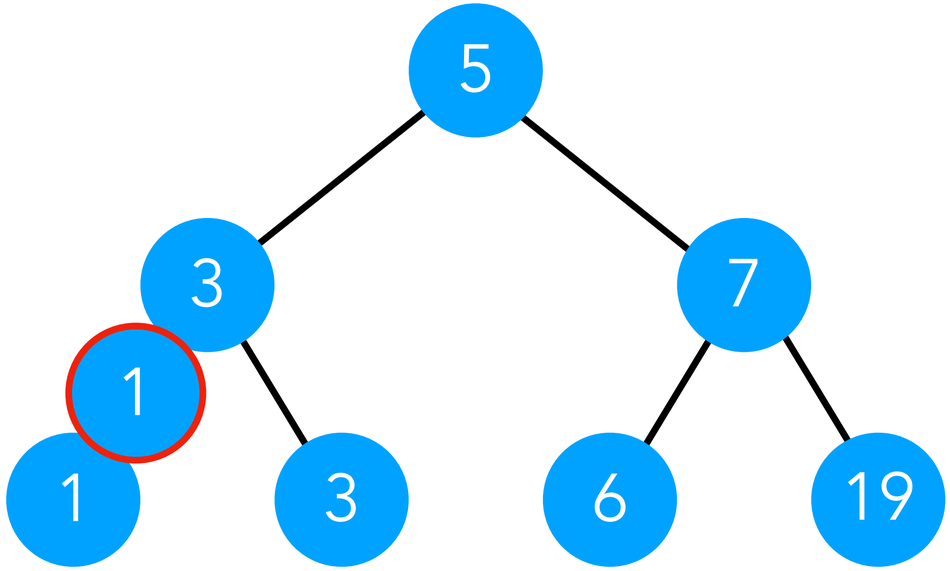
P08.03

Binärbäume, Löschen (1)



P08.03

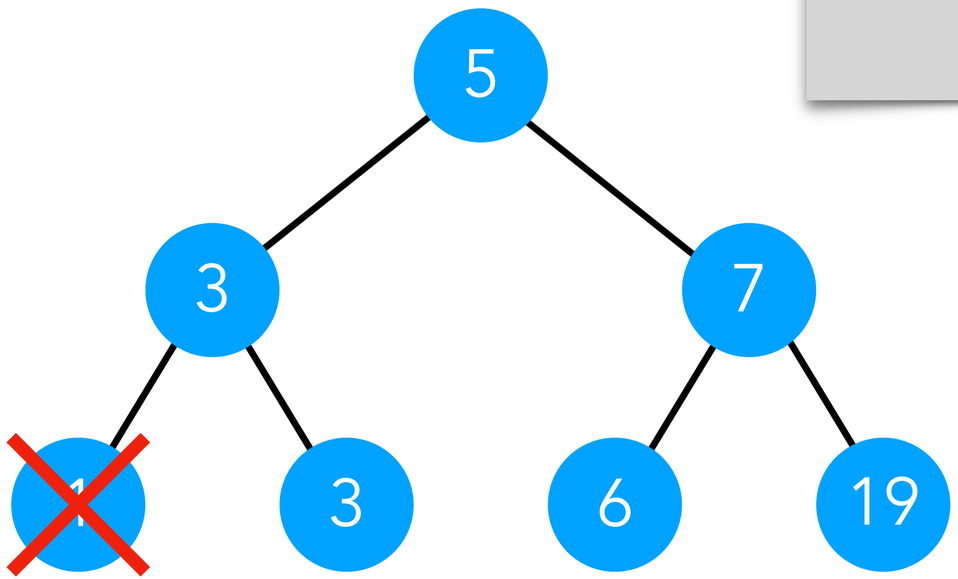
Binärbäume, Löschen (1)



P08.03

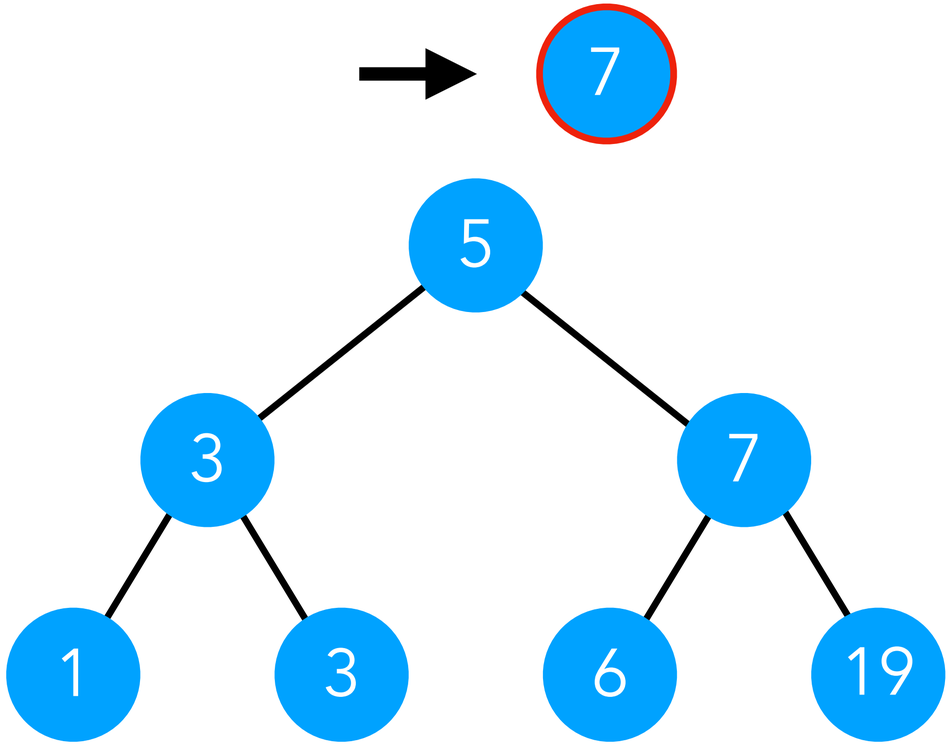
Binärbäume, Löschen (1)

Kann einfach gelöscht werden, da rechts nur ein Blatt ist; linker Teilbaum wird angehangen.



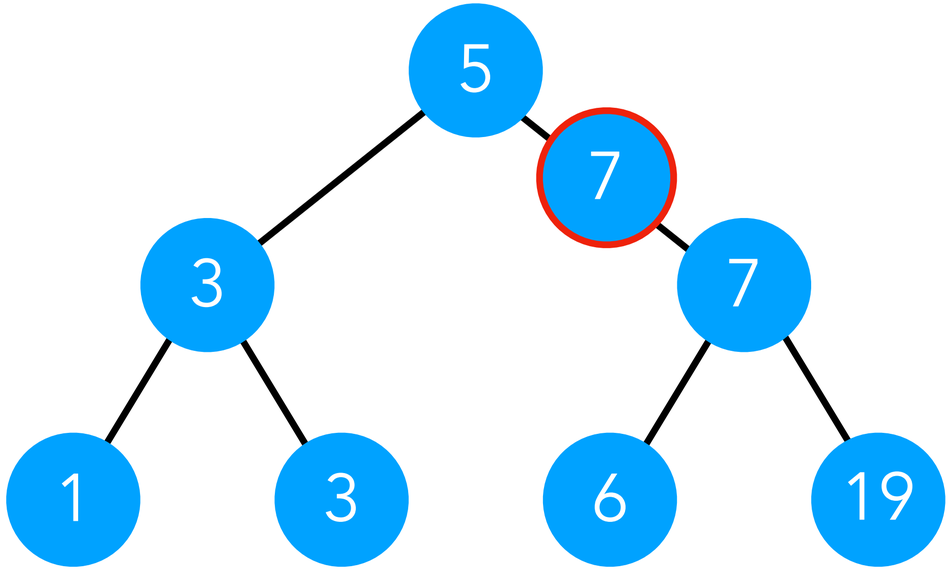
P08.03

Binärbäume, Löschen (2)



P08.03

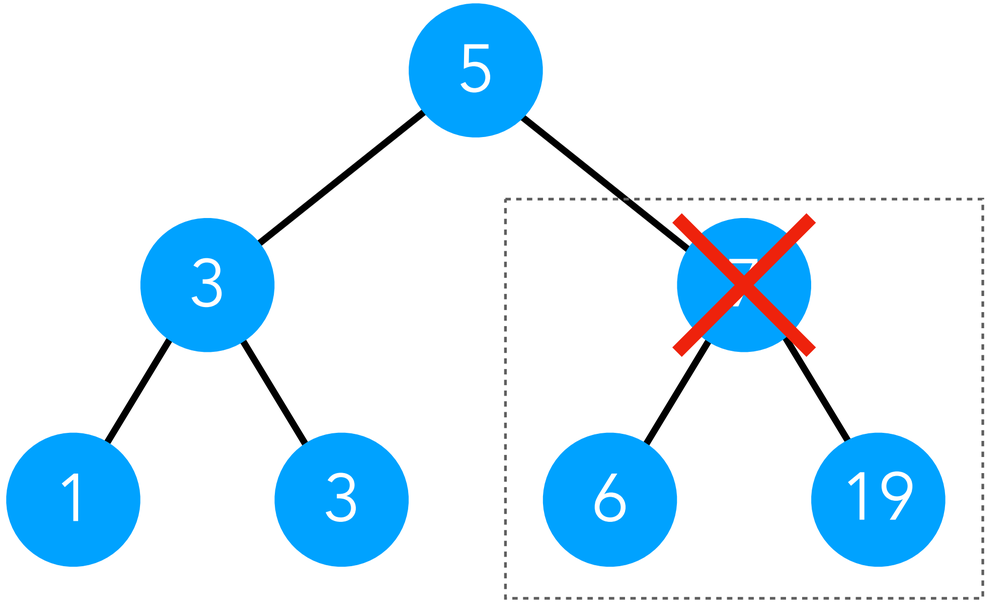
Binärbäume, Löschen (2)



P08.03

Binärbäume, Löschen (2)

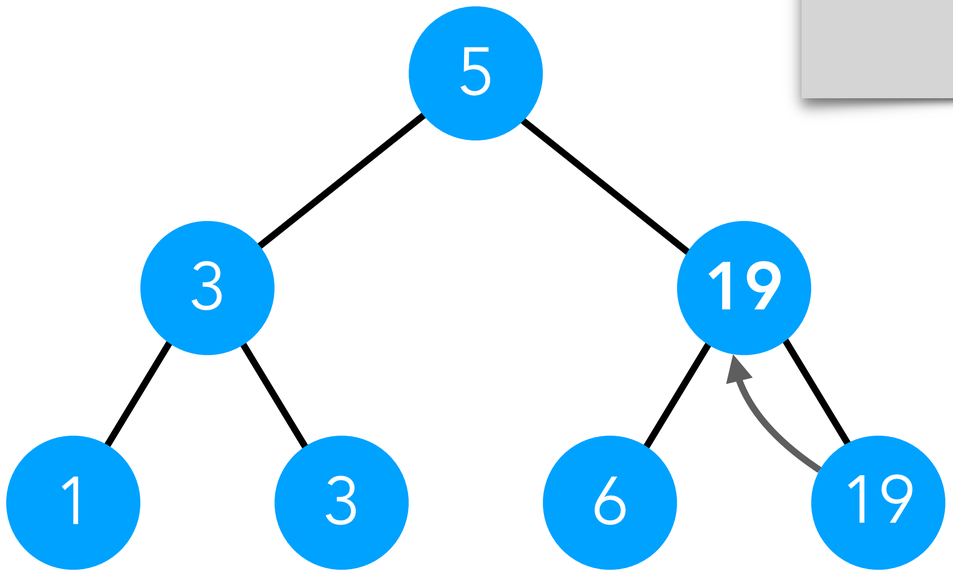
Einfach löschen bedeutet löschen des induzierten Teilbaums



P08.03

Binärbäume, Löschen (2)

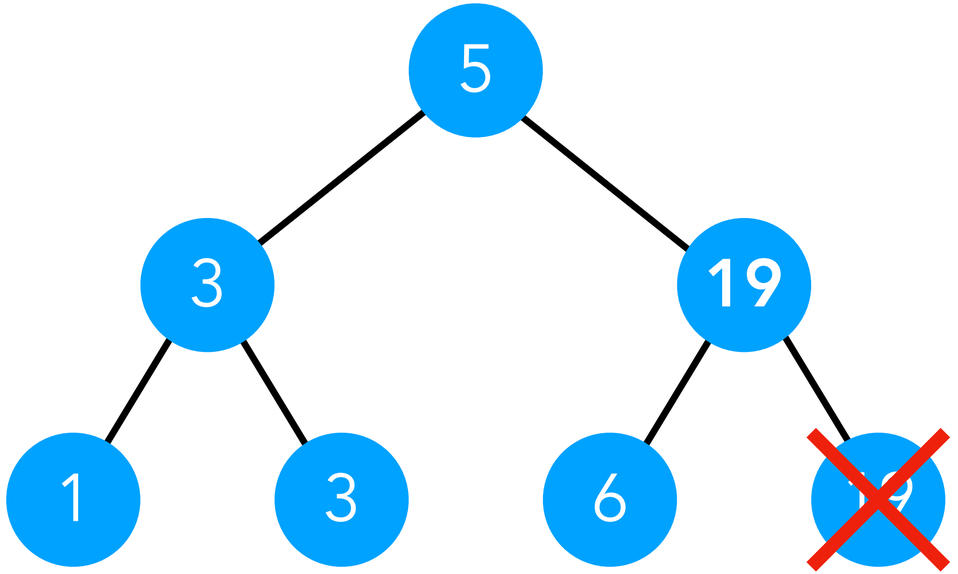
Kleinstes Element des rechten Teilbaums wird neue Wurzel des Teilbaums.



P08.03

Binärbäume, Löschen (2)

Kleinstes Element löschen.



P08.03

Binärbäume,

Kommentierte Version der Lösung auf der Website.