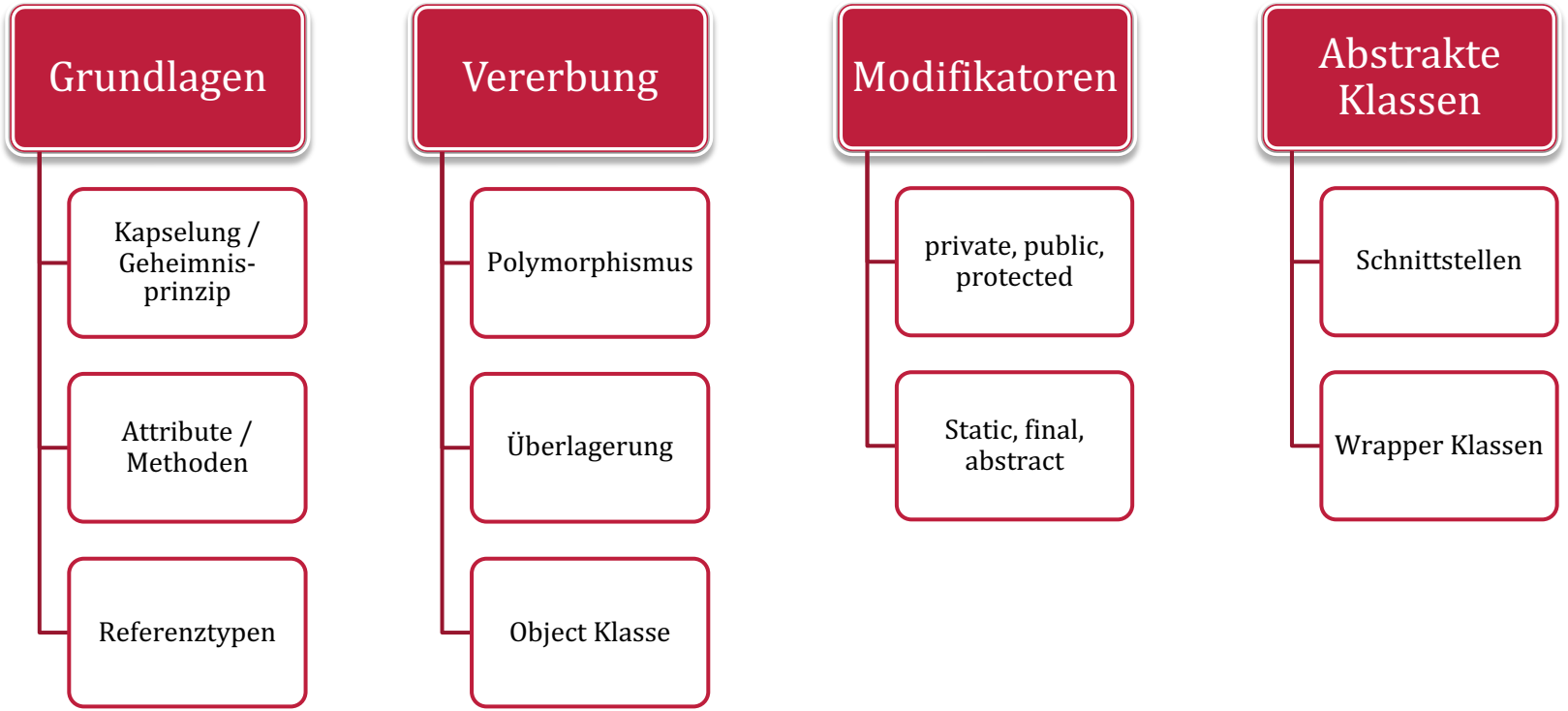
Technische Universität Braunschweig

Programmieren 1 – Vorlesung #8

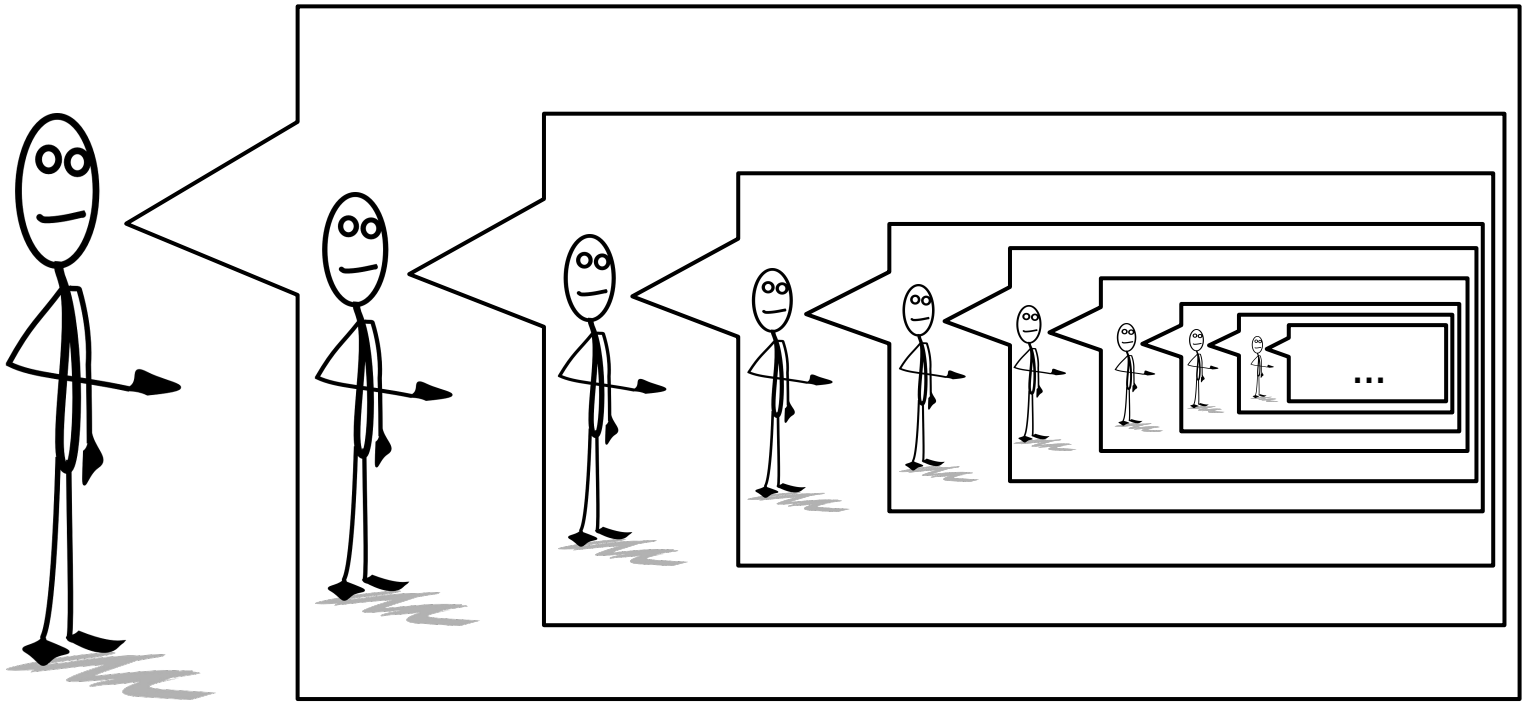
Arne Schmidt

Wiederholung

Letztes Kapitel



Heute: Rekursion



Kapitel 5 – Rekursion

Beispiel 1: Fibonacci-Zahlen

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Zur Berechnung der Fibonacci-Zahlen wird üblicherweise eine **rekursive Funktion** angegeben, also eine Funktion, die **sich selbst wieder aufruft**.

Für die n-te Fibonacci-Zahl ergibt sich

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Beispiel 2: Iterierte Quersumme

Die Quersumme einer Zahl ist die Summe der einzelnen Ziffern. Die iterative Quersumme wiederholt diesen Vorgang bis eine einstellige Zahl übrig bleibt.

Beispiel:
6 → 6

47 → 11 → 2

34914 → 21 → 3

```
static int iterativ(int n) {
   int s = n;

   // So lange s mehrstellig ist
   while (s > 9) {
      int i = s;
      s = 0;

      // Berechne Quersumme von i in s
      do {
         s = s + i % 10;
         i = i / 10;
      } while (i > 0);
   }
   return s;
}
```

Iterativ

```
static int rekursiv(int n) {
   if (n < 10){
      return n;
   }
   return rekursiv(rekursiv(n/10)+ n%10);
}
```

Rekursiv

Iterative Quersumme (alternative)

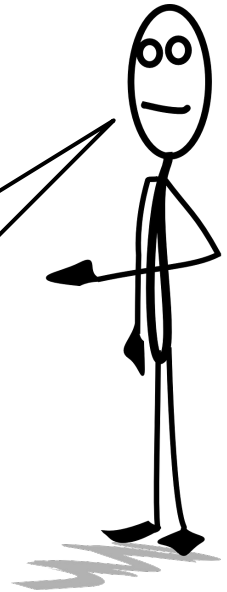
Rekursiv geht das mit dem ternären Operator noch kürzer:

```
static int rekursiv(int n) {
   return n < 10 ? n : rekursiv(rekursiv(n/10)+ n%10);
}
```

Iterativ benötigen wir 9 Zeilen Code, rekursiv nur 1 Zeile Code.

Rekursion ist also eine mächtige Programmiertechnik.

Wir schauen uns heute im Detail rekursive Funktionen an und welche Probleme dabei entstehen können.



Kapitel 5.1 – (Rekursive) Funktionen

(rekursive) Funktionen

Eine **Funktion** ordnet jedem Element x aus einer Definitionsmenge D ein Element y einer Zielmenge Z zu. In vielen Fällen wird eine Funktion $f\colon D \to Z$ durch einen **Ausdruck** beschrieben.

Beispiel max: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ wird definiert über den Ausdruck

$$\max(a, b) := \begin{cases} a, & \text{falls } a > b \\ b, & \text{falls } b > a \end{cases}$$

Eine Funktion wird **rekursiv** definiert, wenn der Ausdruck die Funktion selbst aufruft.

Beispiel $f\colon \mathbb{N} \to \mathbb{N}$ wird rekursiv definiert über den Ausdruck

$$f(n) := \begin{cases} 1, & \text{falls } n \leq 1 \\ f(3n+1), & \text{falls } n > 1 \ ungerade \\ f\left(\left\lfloor \frac{n}{2} \right\rfloor\right), & \text{falls } n > 1 \ gerade \end{cases}$$

Auswertung für rekursive Funktionen

Beispiel $f: \mathbb{N} \rightarrow \mathbb{N}$ wird rekursiv definiert über den Ausdruck

$$f(n) := \begin{cases} 1, & \text{falls } n \leq 1 \\ f(3n+1), & \text{falls } n > 1 \; ungerade \\ f\left(\left\lfloor \frac{n}{2} \right\rfloor\right), & \text{falls } n > 1 \; gerade \end{cases}$$

Funktionsdefinitionen können als **Ersetzungssysteme** gesehen werden. Funktionswerte lassen sich aus dieser Sicht durch wiederholtes Einsetzen berechnen.

Die **Auswertung** von $f(3)$ ergibt
$f(3) \rightarrow f(10) \rightarrow f(5) \rightarrow f(16) \rightarrow f(4) \rightarrow f(2) \rightarrow f(1) \rightarrow 1$

Terminiert $f(n)$ für beliebige n? (Collatz-Problem: Lothar Collatz 1937)

Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

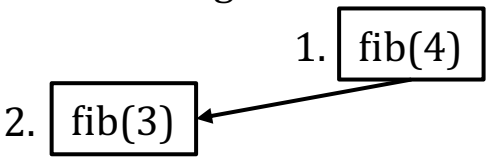
1. fib(4)

Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

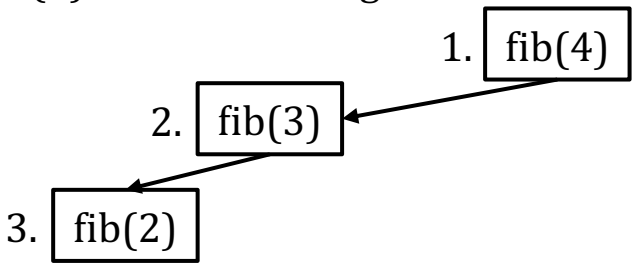


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

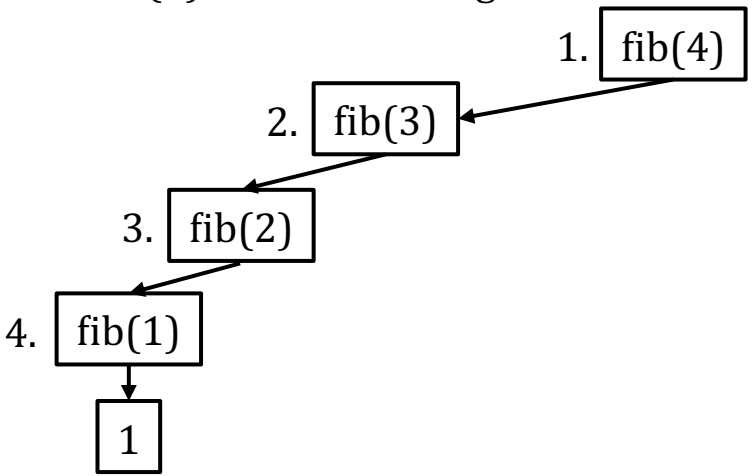


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

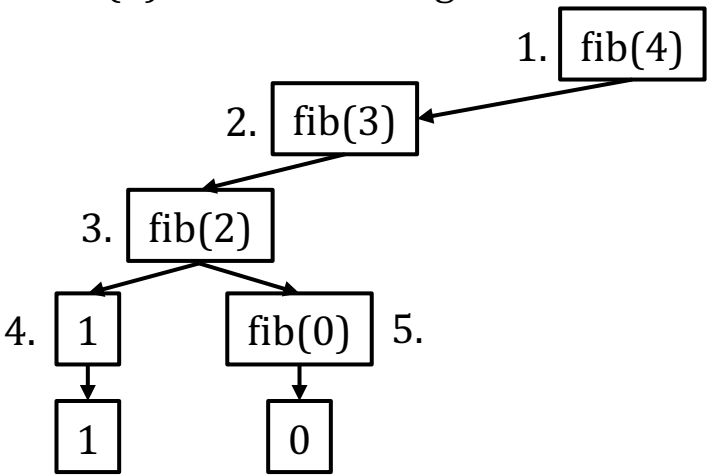


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

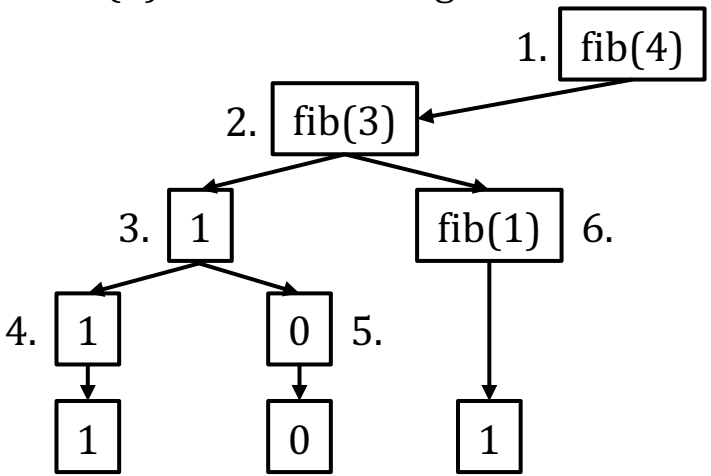


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

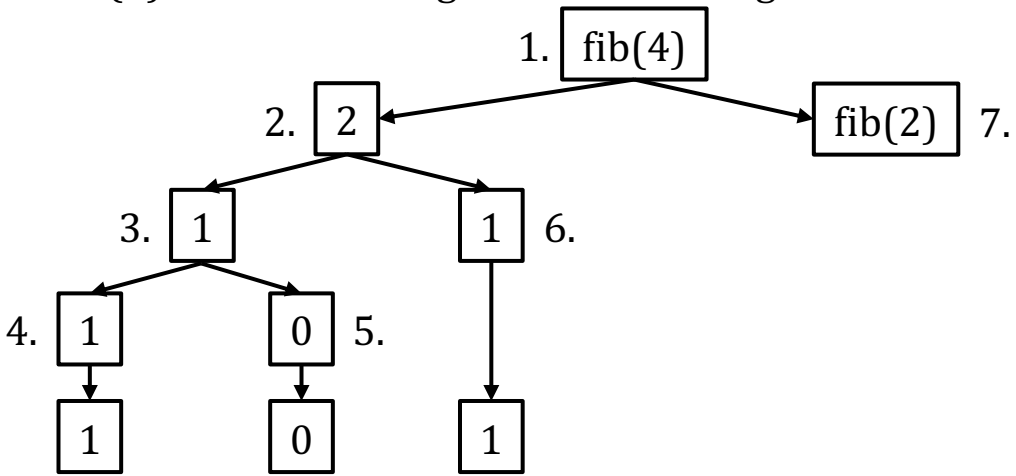


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

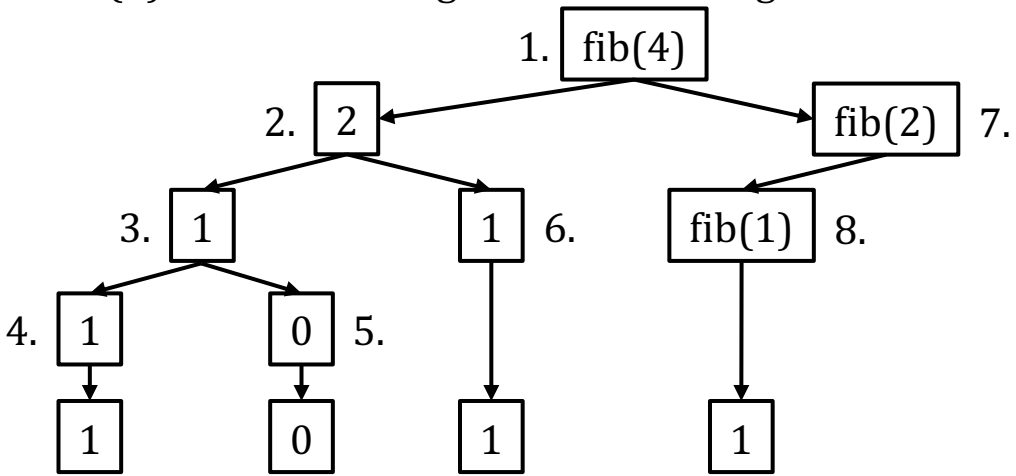


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

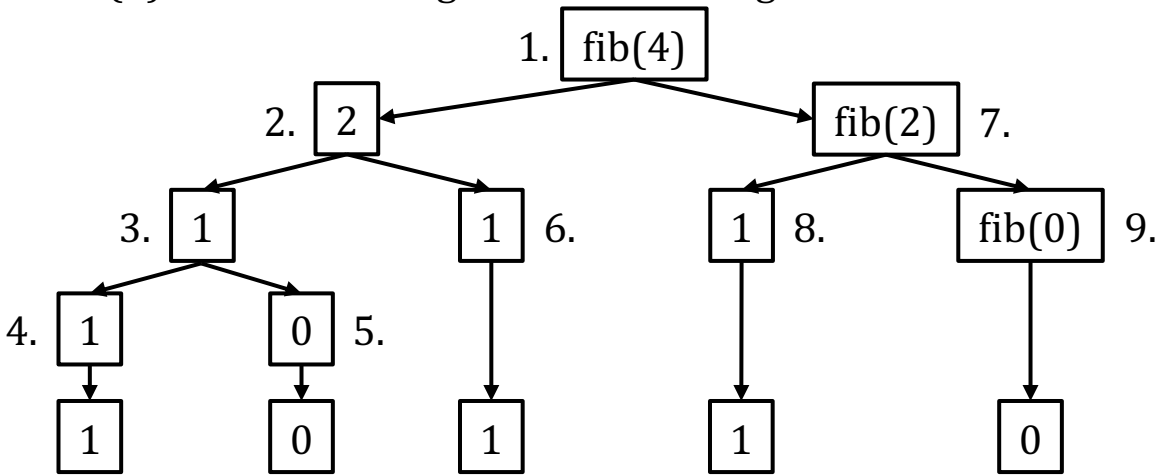


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

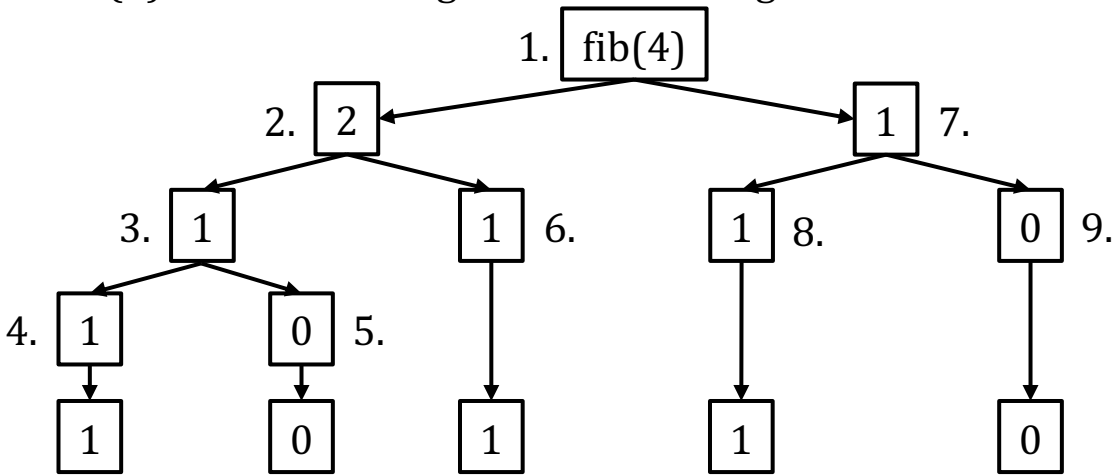


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

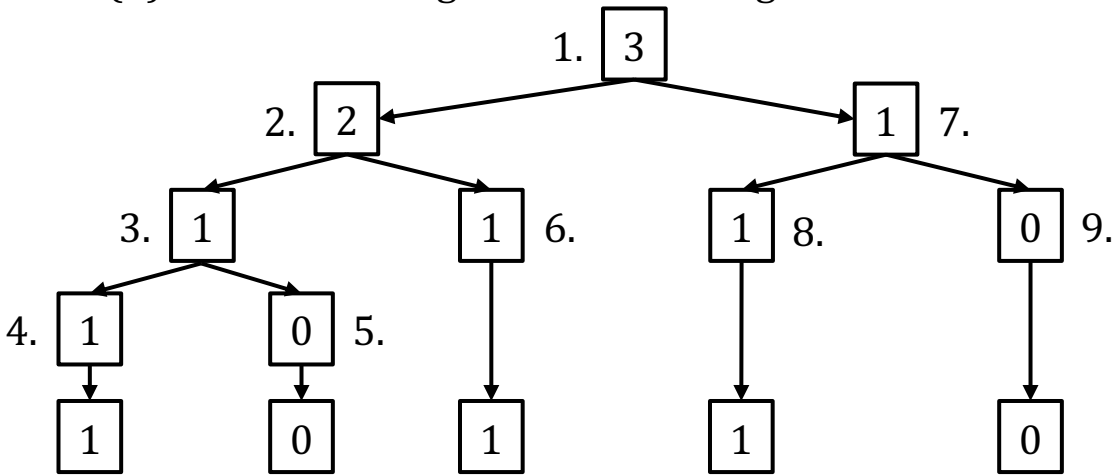


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:

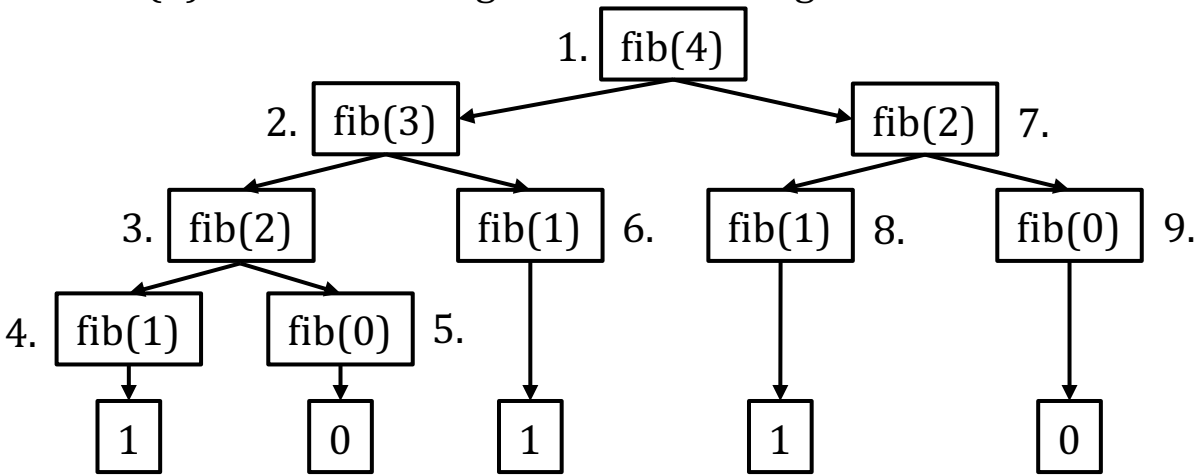


Verzweigungen

Betrachte erneut die Fibonacci-Zahlen:

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Für den Aufruf fib(4) können wir folgende Auswertung festhalten:



Formen von Rekursion

Für einen Funktionsaufruf ist eine **Verzweigung** ein rekursiver Aufruf. Der **Grad der Verzweigung** (oder Verzweigungsgrad) ist die maximale Anzahl an Verzweigungen, die ein Funktionsaufruf bilden kann.

Damit können wir verschiedene Formen von Rekursion definieren.

Lineare Rekursion:
Besitzt einen Verzweigungsgrad von 1.

Baumrekursion:
Besitzt mindestens 2 getrennte Aufrufe.

Verschränkt oder **wechselseitig:**
Funktion wird über andere Funktion erneut aufgerufen

Endrekursion:
Besitzt einen Verzweigungsgrad von 1; Rekursion ist die letzte Operation.

Verschachtelte Rekursion:
Besitzt Aufrufe, deren Ergebnis in einem weiteren Aufruf benutzt wird.

Formen von Rekursion – Beispiele I

$$f(n) := \begin{cases} 1, & \text{falls } n \leq 1 \\ f(3n+1), & \text{falls } n > 1 \text{ ungerade} \\ f\left(\left\lfloor \frac{n}{2} \right\rfloor\right), & \text{falls } n > 1 \text{ gerade} \end{cases}$$

Grad der Zweigung: 1
Verzweigung taucht nur am Ende auf.

➔ Endrekursion

Formen von Rekursion – Beispiele I

$$fac(n) := \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot fac(n-1), & \text{falls } n > 1 \end{cases}$$

Grad der Zweigung: 1
Verzweigung taucht nicht alleine am Ende auf, sondern muss noch mit n multipliziert werden.

➔ Lineare Rekursion

Formen von Rekursion – Beispiele I

$$fib(n) := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ fib(n-1) + fib(n-2), & \text{sonst} \end{cases}$$

Grad der Zweigung: 2
Verzweigungen sind nicht verschachtelt.

➔ Baumrekursion

Formen von Rekursion – Beispiele I

$$A(n,m) := \begin{cases} m+1, & \text{falls } n = 0 \\ A(n-1,1), & \text{falls } m = 0 \\ A\big(n-1, A(n,m-1)\big), & \text{sonst} \end{cases}$$

Auch bekannt als Ackermann-Funktion

Grad der Zweigung: 2
Verzweigungen im letzten Fall sind verschachtelt.

➔ Verschachtelte Rekursion

Kapitel 5.2 – Memoisation

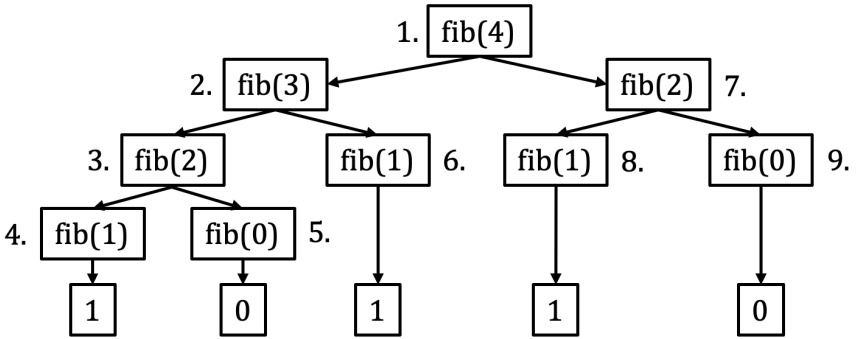
Fibonacci-Zahlen in Java

```
class Fibonacci {
   static long fib(long n) {
      if ((n == 0) || (n == 1)) {
         return n;
      } else {
         return fib(n - 1) + fib(n - 2);
      }
   }

   public static void main(String[] args) {
      for (int i = 0; i <= 50; i++) {
         System.out.println(i + ": " + fib(i));
      }
   }
}
```

Was kann hier ein Problem sein?

Dieses Programm benötigt eine Ewigkeit!



Die *Laufzeit* ist exponentiell, weil wir Arbeit wiederholen. → Rufe bereits berechnete Werte ab!

Memoization

```
class FibonacciDyn {
   private static long[] fib;
   public static long fib(int n) {
      if (n == 0 || n == 1) {return n; }
      fib = new long[n + 1];

      fib[0] = 0; fib[1] = 1; // schlecht formatierte Init.
      for (int i = 2; i <= n; i++) {fib[i] = -1;} // des Arrays
      return f(n); // Begin der eigentlichen Rekursion
   }
   private static long f(int n) {
      if (fib[n] >= 0) {
         return fib[n];
      } else {
         return fib[n] = f(n - 1) + f(n - 2);
      }
   }
}
```

Anstatt Werte immer wieder neu zu berechnen, speichere diese in einer geeigneten Datenstruktur. Dieser Vorgang wird "**memoisieren**" genannt.

Am Beispiel der Fibonacci-Zahlen reicht uns ein Array.

Solange ein Wert unbekannt ist, besitzt das Array an der Stelle eine -1.

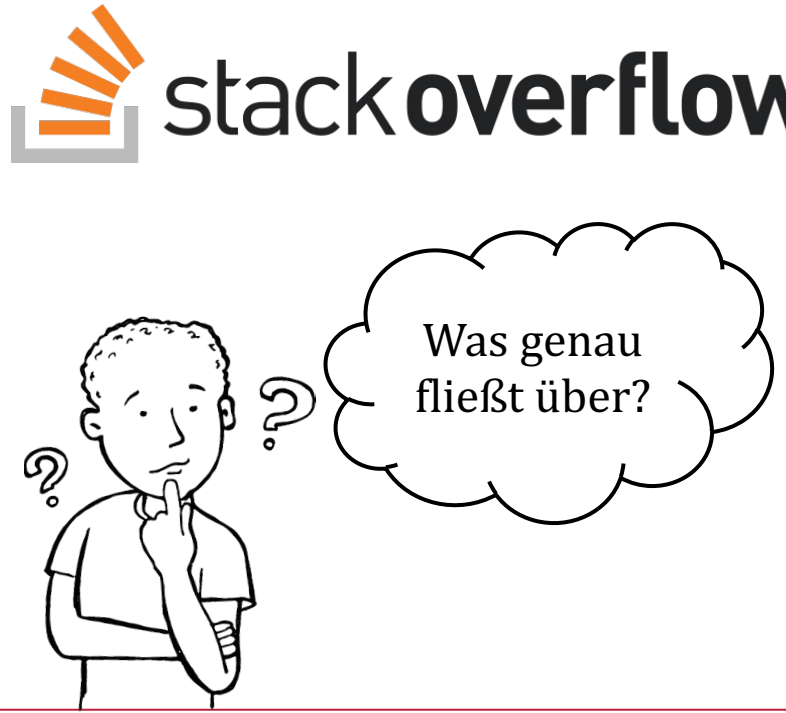
Bevor wir rekursiv starten, prüfen wir immer, ob der Wert schon berechnet wurde.

Kapitel 5.3 – Rekursionstiefe

Summe der natürlichen Zahlen

```
class Summe {
   static long sum(long n) {
      if (n == 0){
         return 0;
      }
      return n + sum(n-1);
   }

   public static void main(String[] args) {
      System.out.println("200.000: " + sum(200_000));
   }
}
```



Methodenaufrufen

Kapitel 5.3.1 – Was passiert bei Rekursion?

Was passiert bei einem Methodenaufruf?

Zunächst muss man sich ein paar Fragen stellen!

Woher weiß das Programm, wohin es wieder springen muss, wenn eine Methode terminiert?
Man könnte den Springpunkt speichern.

Und wenn in der Methode eine Methode aufgerufen wird?
Dann muss ich die Reihenfolge der Springpunkte speichern!

Also:
Wie und wo werden die Punkte gespeichert?
Braucht man noch mehr Informationen?

Was muss man in einer Methode wissen?

Jede Methode verfügt über:
- Parameter
- Lokale Variablen

Diese darf man natürlich nicht vergessen!

Speichere also folgende Informationen bei einem Methodenaufruf ab:
Parameter, lokale Variablen und return-Adresse

Wir nennen diese Informationen einen **stack frame.** Dies entspricht einem Abbild der Methode, welches auf einem **Stapel** gespeichert wird.

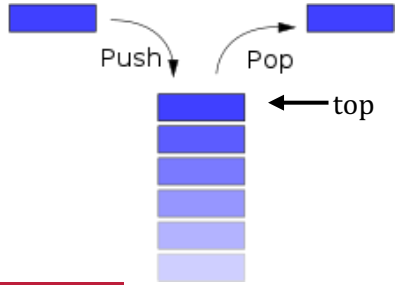
Kapitel 5.3.2 – Stapel

Stapel

Ein **Stapel** (engl. **Stack**) ist eine einfache Datenstruktur, welche Daten nach dem LIFO-Prinzip (last in, first out) verwaltet.

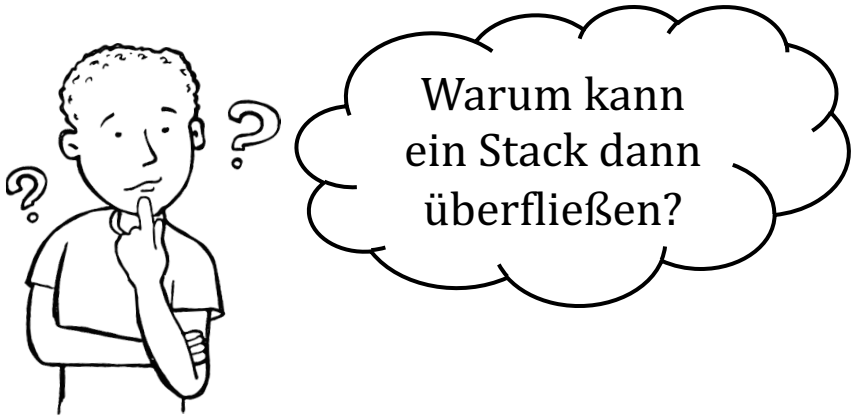
Ein Stapel unterstützt in der Regel folgende Operatoren:

- **push(Object o)**: Fügt Objekt o dem Stapel hinzu.
- **top()**: Gibt das oberste (zuletzt hinzugefügte) Objekt zurück.
- **pop()**: Gibt das oberste Objekt zurück und löscht es vom Stapel
- **empty()**: Prüft, ob der Stapel leer ist.



Größe von (Call) Stacks

Generell kann die Größe von Stapeln dynamisch angelegt werden, d.h. sie hängt allein von den gespeicherten Daten ab und kann prinzipiell unendlich viele Daten speichern.



Programmiersprachen limitieren die Größe des **call stacks**, welcher die stack frames enthält. Damit kann es nur begrenzt viele Methodenaufrufe geben!

Und nun?


Wie kann ich dann rekursive Funktionen implementieren, die eine hohe Rekursionstiefe besitzen?

Entrekursivierung - Idee

Eine Option ist es, selbst einen Stack zu verwalten und die Rekursion durch eine while-Schleife und mehreren if-Verzweigungen oder switch-cases zu ersetzen.

Lineare Rekursionen lassen sich sogar einfacher mit Hilfe von Arrays und einer for-Schleife entrekursivieren.

```
//Ein Stack, der stack frames enthalten wird, sowie eine Zahl (obj.num)
Stack stack = new Stack();
stack.push(init stack frame)
while (!stack.empty()){
    switch stack.top().num{
    case 0:
        //Alles vor dem ersten rekursiven Aufruf
        //Push stack frame und Zahl 1
        break;
    case 1:
        //Alles nach dem ersten rekursiven Aufruf
        //Push stack frame und Zahl 2
        break;
    //...
    case default:
        //Alles nach dem letzten rekursiven Aufruf
        stack.pop();
    }
}
```

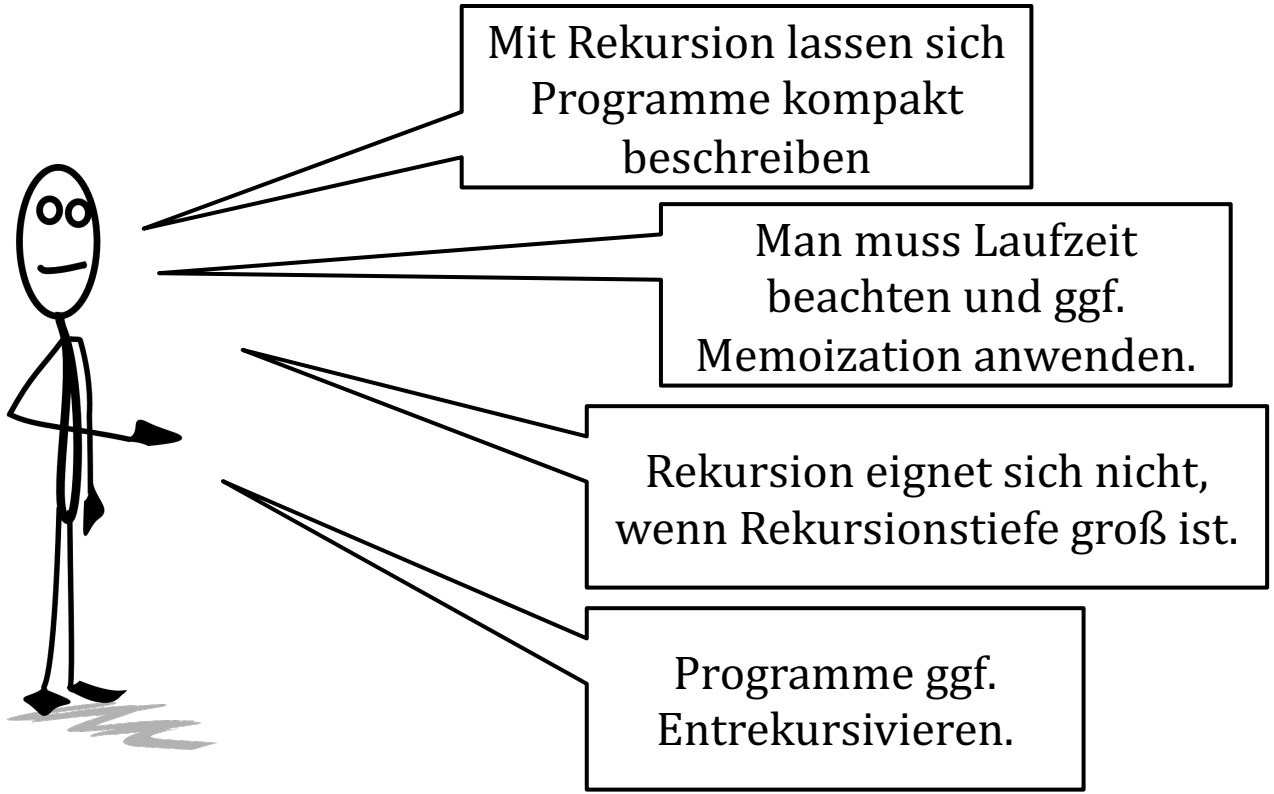
Beispiel Entrekursivierung

```
public static int f(int a, int b){
    if (b == 0){
        return a;
    }
    return f(b, a mod b);
}
```

```
public static int f_iter(int a, int b){
    while (true){
        if (b == 0){
            return a;
        }

        int temp = a mod b;
        a = b;
        b = temp;
    }
}
```

Zusammenfassung



Nächste Woche: Exceptions

```
public class Main{

   public static void main(String[] args) {
      String[] A = {"Dieses", "Array", "Hat", "Insgesamt", "Sechs", "Felder"};
      A[6] = "!";
   }
}
```

```
Exception in thread "main" java.lang
    .ArrayIndexOutOfBoundsException: Index 6 out of
    bounds for length 6at Main.main(Main.java:6)
```