12. Rekursion

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration, Lindenmayer Systeme

Lernziele

- Sie verstehen, wie eine Lösung eines rekursives Problems in Java umgesetzt werden kann.
- Sie wissen, wie Methoden in einem **Aufrufstapel** abgearbeitet werden.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich rekursiv definierbar.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich rekursiv definierbar.
- Das heisst, die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

Rekursion in Java: Genauso!

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

```
// POST: return value is n!
public static int fac (int n) {
 if (n <= 1)
    return 1;
  else
    return n * fac (n-1);
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife…

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife…
- …nur noch schlechter ("verbrennt" Zeit **und** Speicher)

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife…
- …nur noch schlechter (“verbrennt” Zeit **und** Speicher)

```
private static void f() {
  f(); // f() -> f() -> ... stack overflow
}
```

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n):`
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument $< n$ aufgerufen.

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)`:
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument $< n$ aufgerufen.

„n wird mit jedem Aufruf kleiner."

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {

  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Aufruf von **`fac(4)`**

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {
 // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {
 // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Auswertung des Rückgabeausdrucks

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {
 // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Rekursiver Aufruf mit Argument $n - 1 == 3$

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {
 // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {
  // n = 3
    if (n <= 1) return 1;
    return n * fac(n-1); // n > 1
}
```

Es gibt jetzt zwei n. Das von `fac(4)` und das von `fac(3)`

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
public static int fac (int n) {

  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es wird mit dem n des aktuellen Aufrufs gearbeitet: $n = 3$

Initialisierung des formalen Arguments

Der Aufrufstapel

```
Out.println(fac(4))
```

Der Aufrufstapel

Bei jedem Methodenaufruf:

`fac(4)` ↑

`Out.println(fac(4))`

Der Aufrufstapel

Bei jedem Methodenaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel

$n = 4$

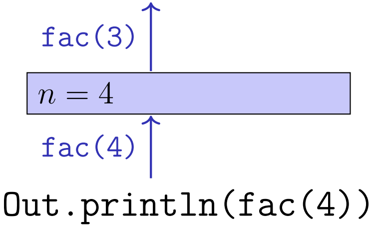
`fac(4)`

`Out.println(fac(4))`

Der Aufrufstapel

Bei jedem Methodenaufruf:

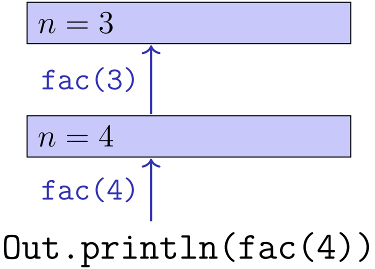
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Methodenaufruf:

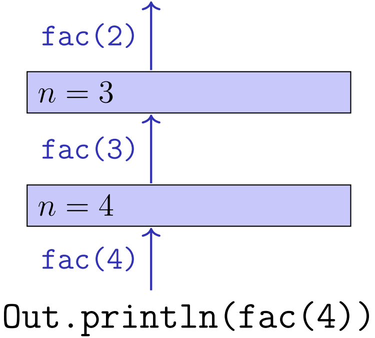
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Methodenaufruf:

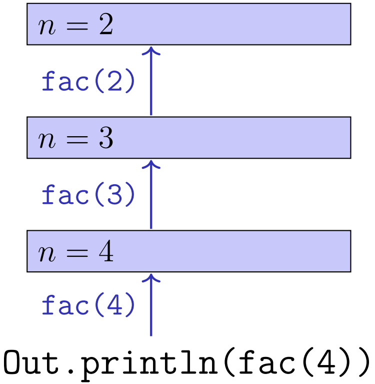
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Methodenaufruf:

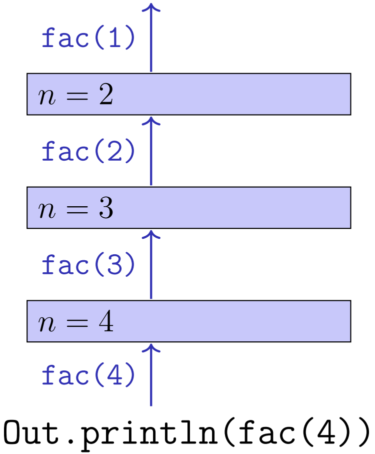
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Methodenaufruf:

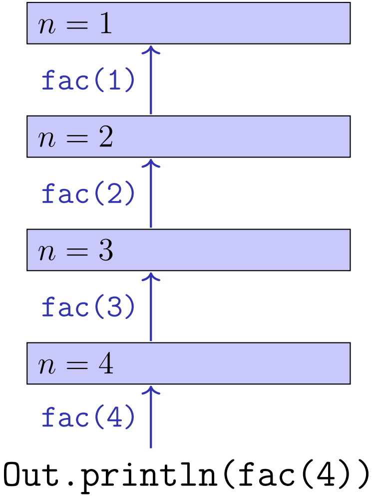
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Methodenaufruf:

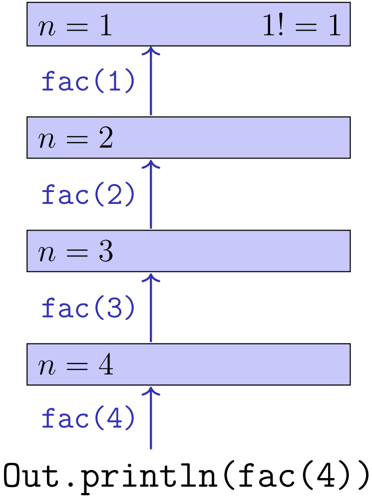
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Methodenaufruf:

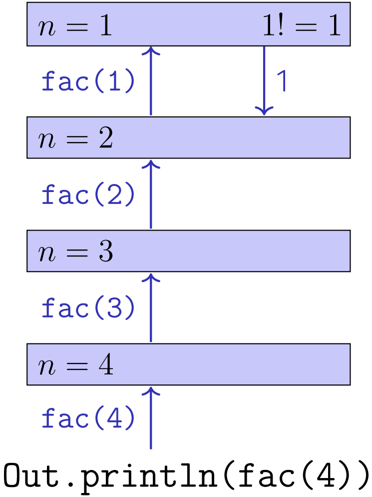
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet



Der Aufrufstapel

Bei jedem Methodenaufruf:

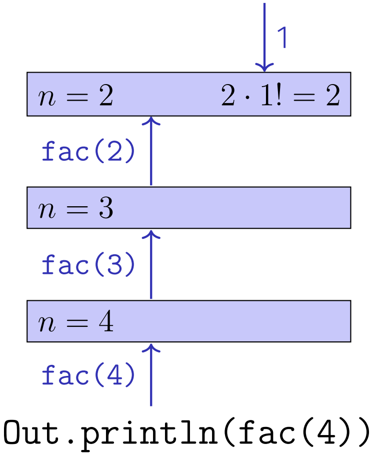
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

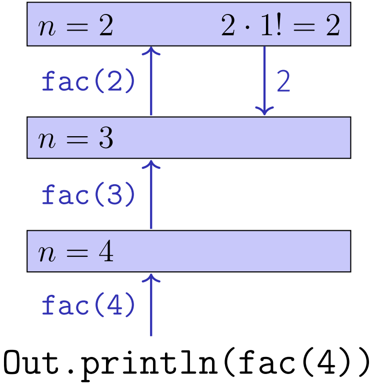
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

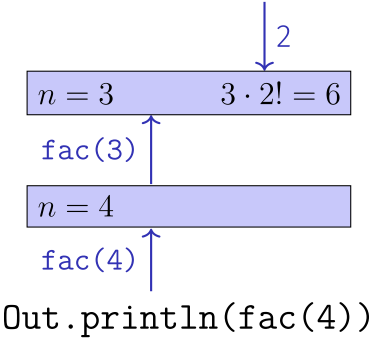
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

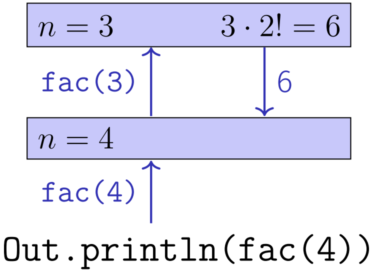
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

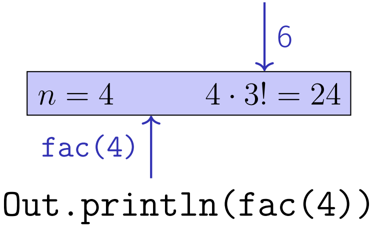
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

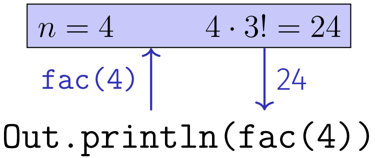
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Methodenaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht

24

```
Out.println(fac(4))
```

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\operatorname{gcd}(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion:

$$\operatorname{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \operatorname{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in Java

$$\gcd(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```java
public static int gcd (int a, int b) {
if (b == 0)
    return a;
  else
    return gcd (b, a % b);
}
```

Euklidischer Algorithmus in Java

$$\mathrm{gcd}(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \mathrm{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```java
public static int gcd (int a, int b) {
if (b == 0)
    return a;
  else
    return gcd (b, a % b);
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

Fibonacci-Zahlen in Zürich

Fibonacci-Zahlen in Java

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
public static int fib (int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib (n-1) + fib (n-2); // n > 1
}
```

Fibonacci-Zahlen in Java

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
public static int fib (int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib (n-1) + fib (n-2); // n > 1
}
```

Korrektheit und Terminierung sind klar.

Fibonacci-Zahlen in Java

Laufzeit

`fib(50)` dauert „ewig", denn es berechnet
F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal, F_{43} 21-mal … F_1 ca. 10^9 mal (!)

```
public static int fib (int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib (n-1) + fib (n-2); // n > 1
}
```

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!
- Berechne die nächste Zahl als Summe von `a` und `b`!

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  int a = 1;  // F_1
  int b = 1;  // F_2
  for (int i = 3; i <= n; ++i){
    int a_old = a;  // F_i-2
    a = b;                         // F_i-1
    b += a_old;                    // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(\underbrace{F_{i-2}}_{a}, \underbrace{F_{i-1}}_{b}) \longrightarrow (F_{i-1}, F_i)$$

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  int a = 1;  // F_1
  int b = 1;  // F_2
  for (int i = 3; i <= n; ++i){
    int a_old = a;  // F_i-2
    a = b;                          // F_i-1
    b += a_old;                     // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a

b

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  int a = 1;  // F_1
  int b = 1;  // F_2
  for (int i = 3; i <= n; ++i){
    int a_old = a;  // F_i-2
    a = b;                          // F_i-1
    b += a_old;                     // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (\underbrace{F_{i-1}}_{\text{a}}, \underbrace{F_i}_{\text{b}})$$

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  int a = 1;  // F_1
  int b = 1;  // F_2
  for (int i = 3; i <= n; ++i){
    int a_old = a;  // F_i-2
    a = b;                          // F_i-1
    b += a_old;                     // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (\underset{a}{F_{i-1}}, \underset{b}{F_i})$$

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  int a = 1;  // F_1
  int b = 1;  // F_2
  for (int i = 3; i <= n; ++i){
    int a_old = a;  // F_i-2
    a = b;                        // F_i-1
    b += a_old;                   // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

sehr schnell auch bei **`fib(50)`**

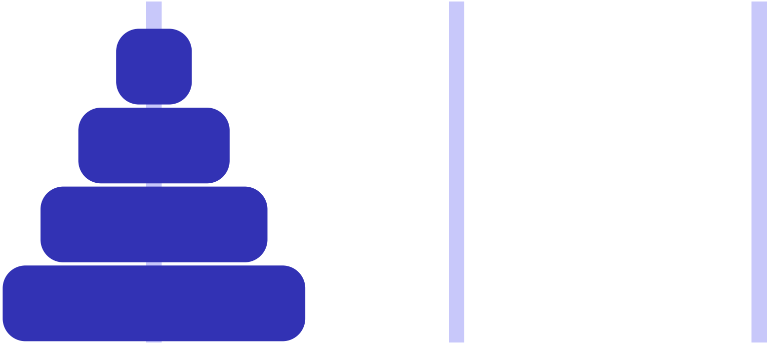
$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a b

Die Macht der Rekursion

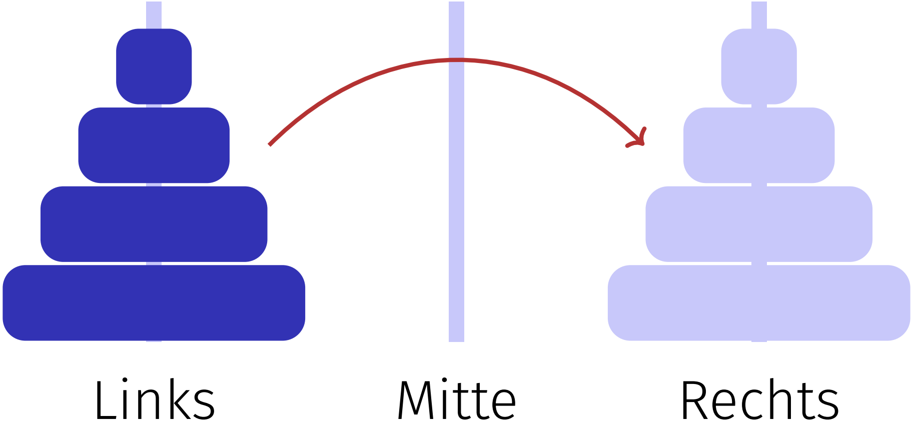
- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich einfacher lösbar.
- Beispiele: *Die Türme von Hanoi*, das n-Damen-Problem, Parsen von Ausdrücken, Sudoku-Löser, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren) $\rightarrow$ Informatik II

Experiment: Die Türme von Hanoi

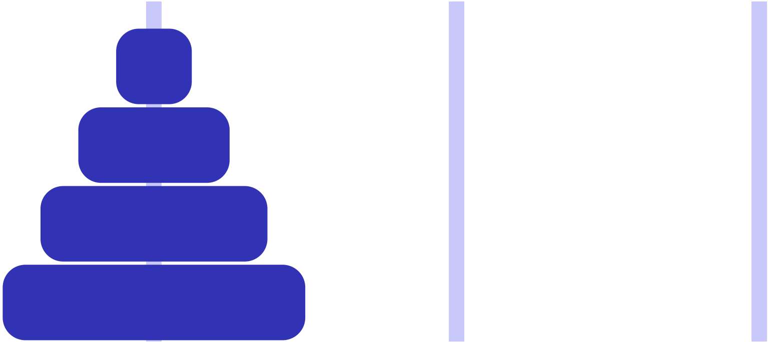


Links Mitte Rechts

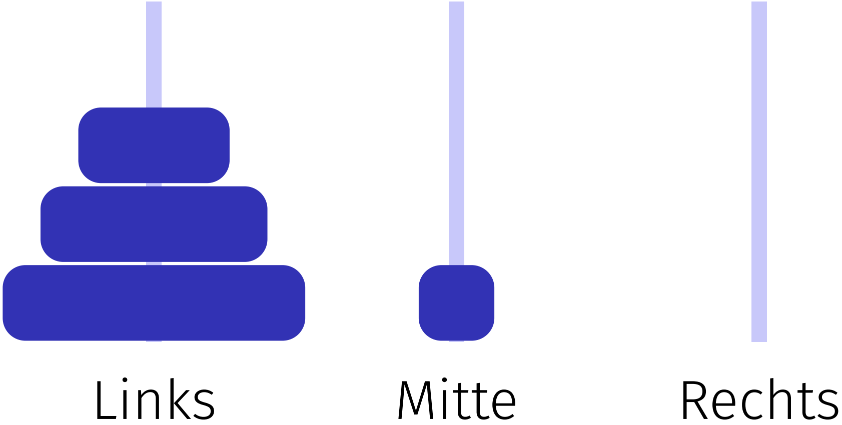
Experiment: Die Türme von Hanoi



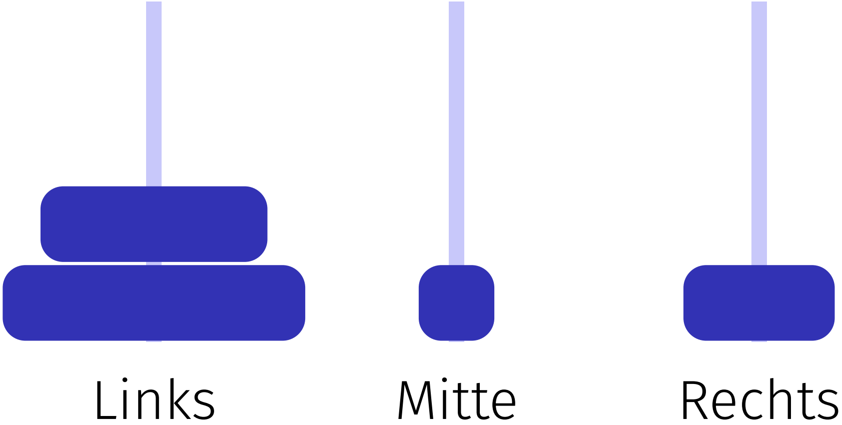
Die Türme von Hanoi - So gehts!



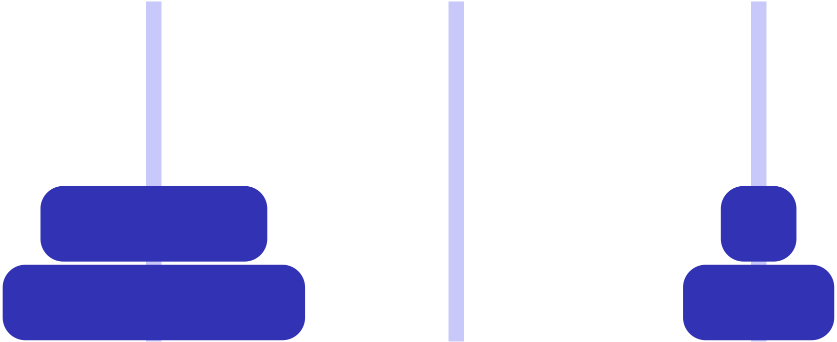
Die Türme von Hanoi - So gehts!



Die Türme von Hanoi - So gehts!

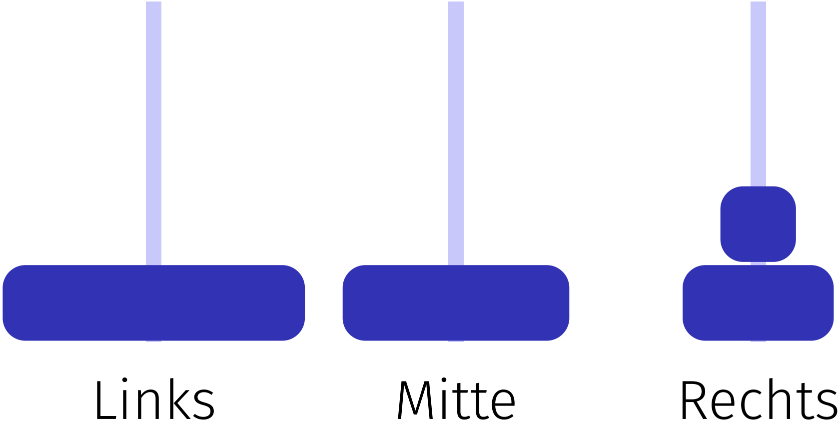


Die Türme von Hanoi - So gehts!

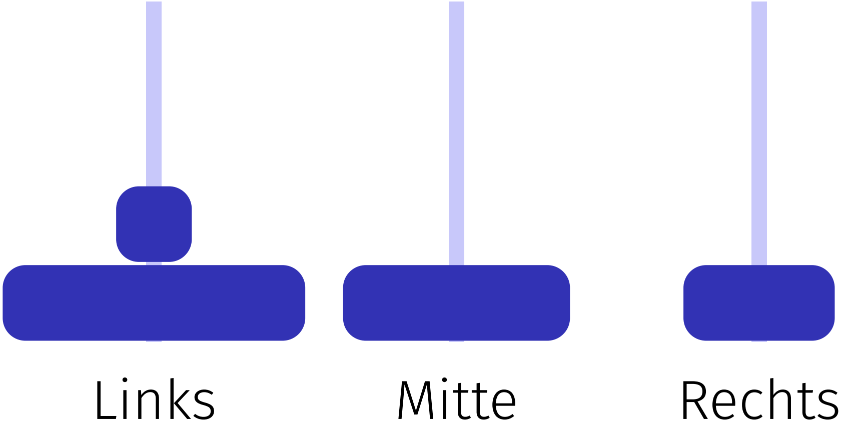


Links Mitte Rechts

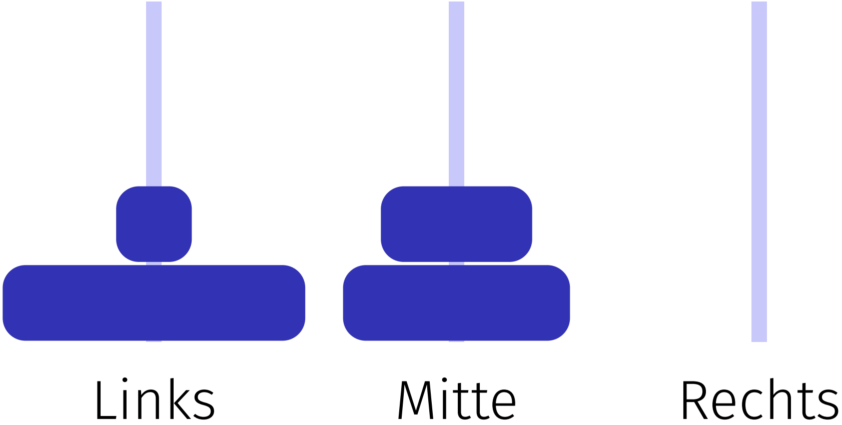
Die Türme von Hanoi - So gehts!



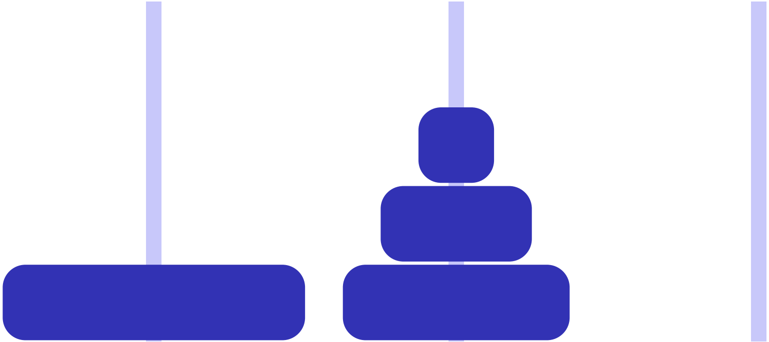
Die Türme von Hanoi - So gehts!



Die Türme von Hanoi - So gehts!



Die Türme von Hanoi - So gehts!

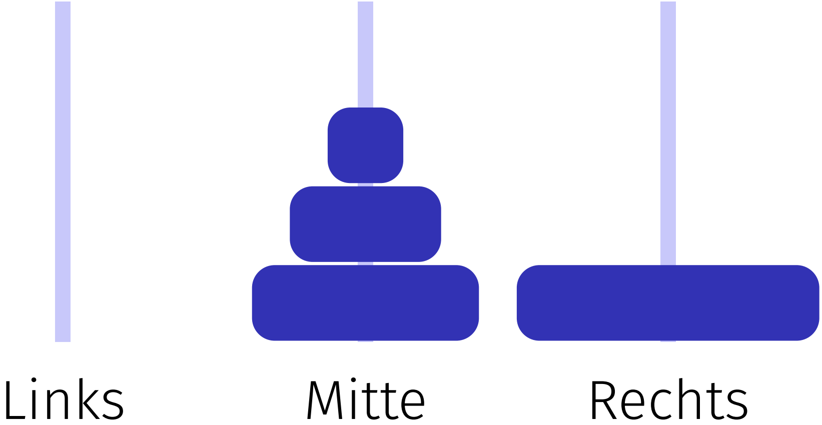


Links

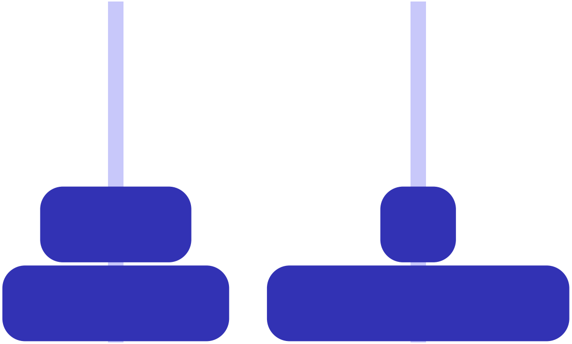
Mitte

Rechts

Die Türme von Hanoi - So gehts!

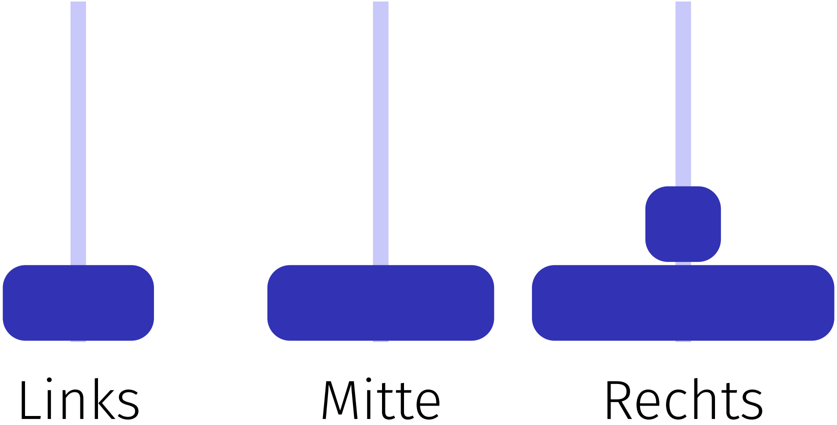


Die Türme von Hanoi - So gehts!

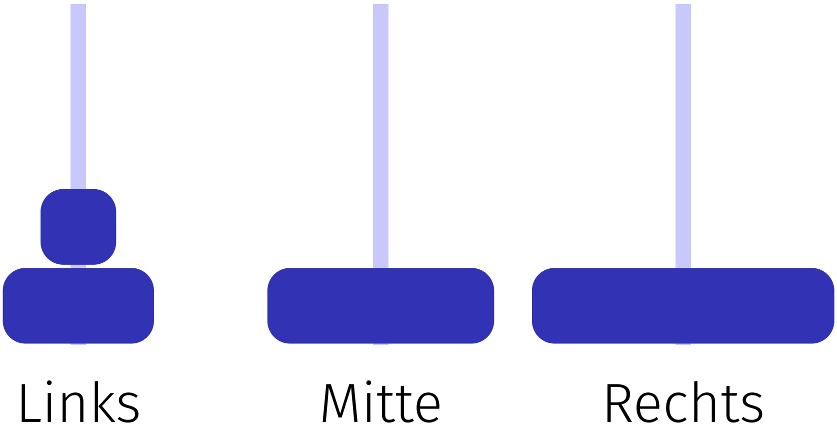


Links Mitte Rechts

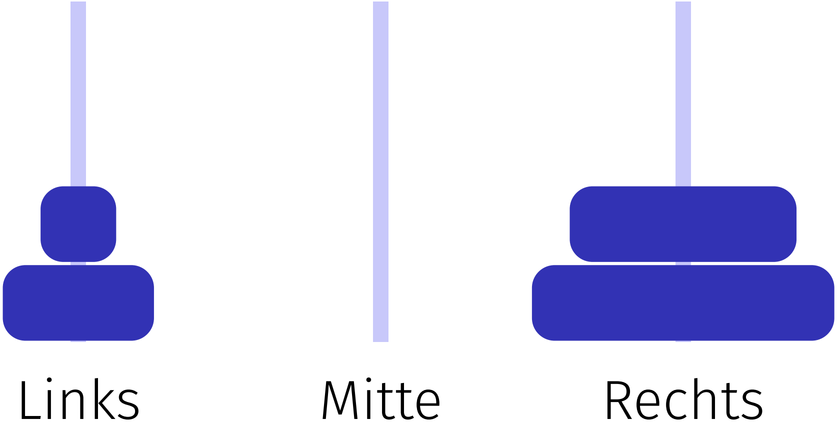
Die Türme von Hanoi - So gehts!



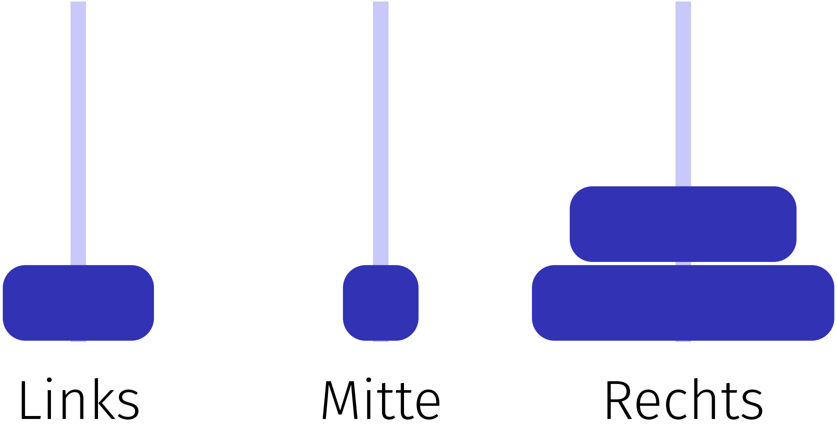
Die Türme von Hanoi - So gehts!



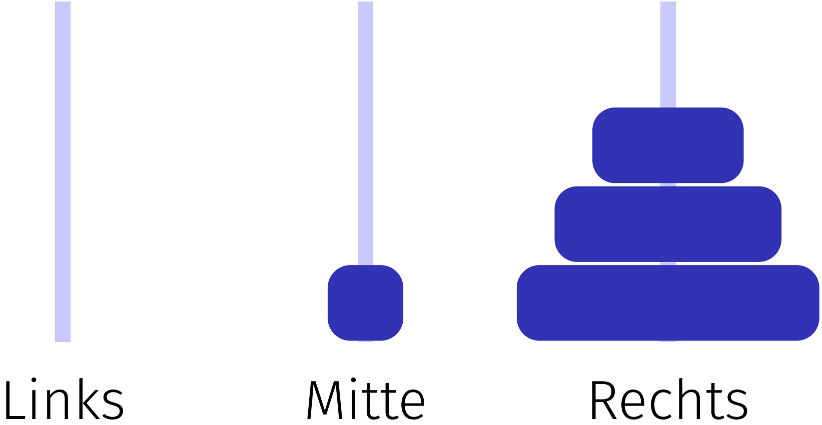
Die Türme von Hanoi - So gehts!



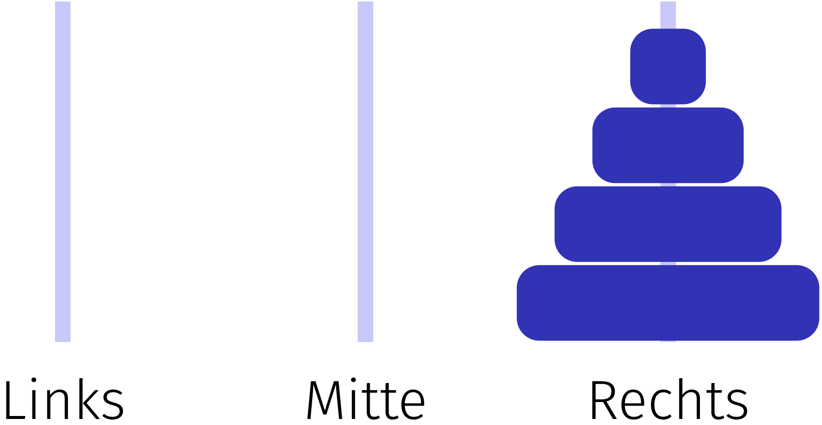
Die Türme von Hanoi - So gehts!



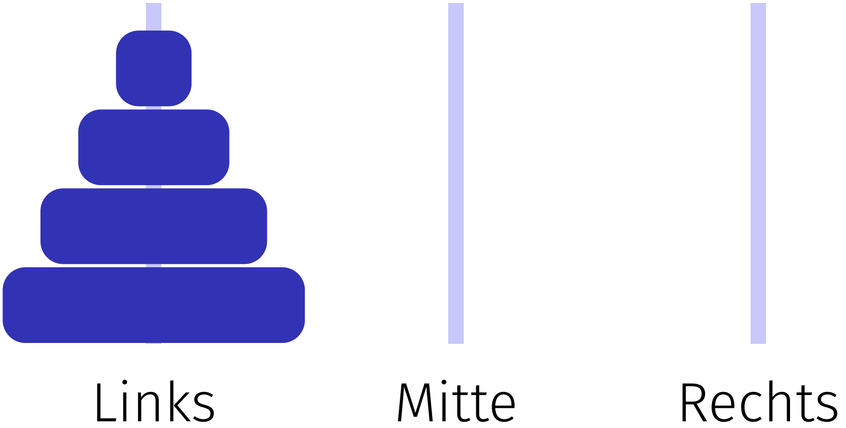
Die Türme von Hanoi - So gehts!



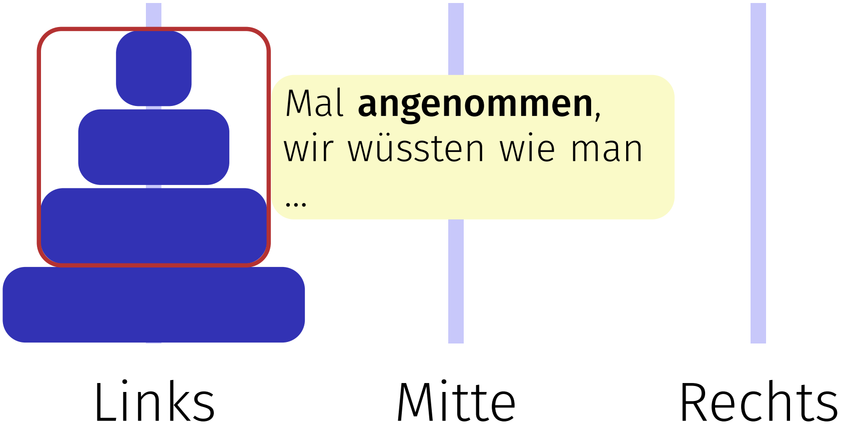
Die Türme von Hanoi - So gehts!



Die Türme von Hanoi - Rekursiv



Die Türme von Hanoi - Rekursiv

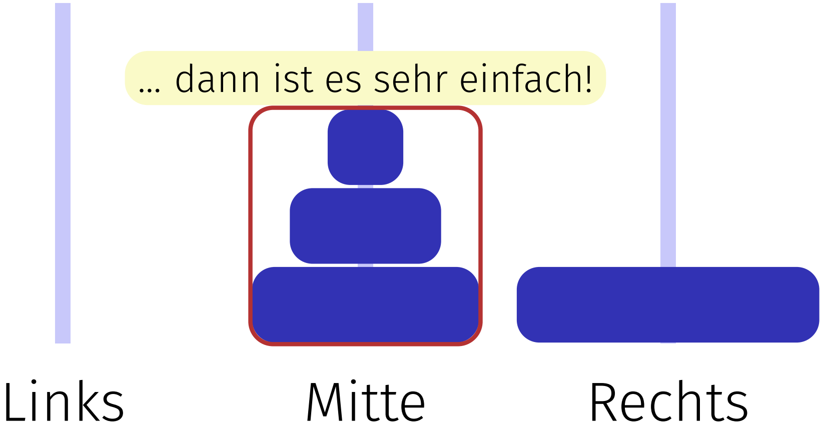


Die Türme von Hanoi - Rekursiv

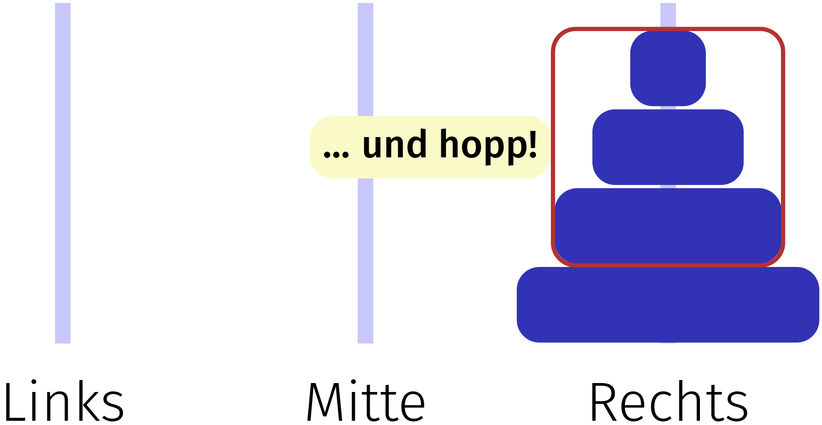
… **drei** Scheiben von einem Stapel zum anderen bringt …

Links Mitte Rechts

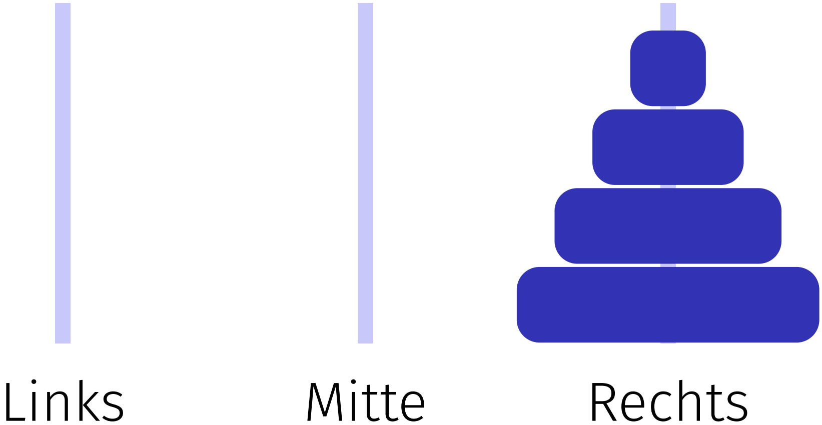
Die Türme von Hanoi - Rekursiv



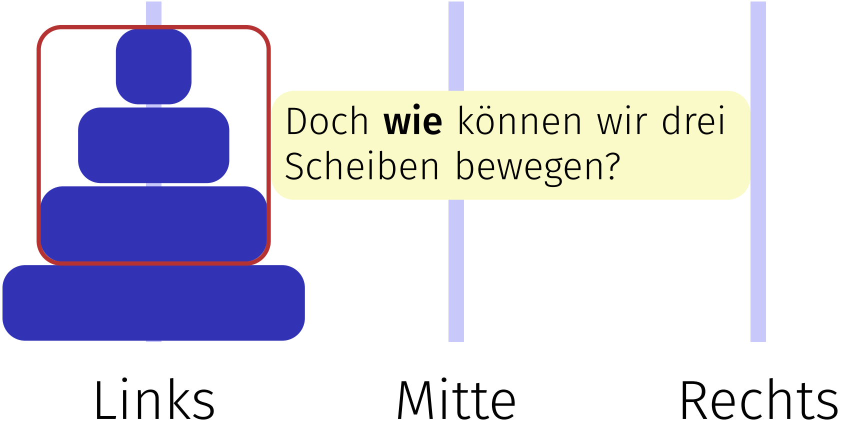
Die Türme von Hanoi - Rekursiv



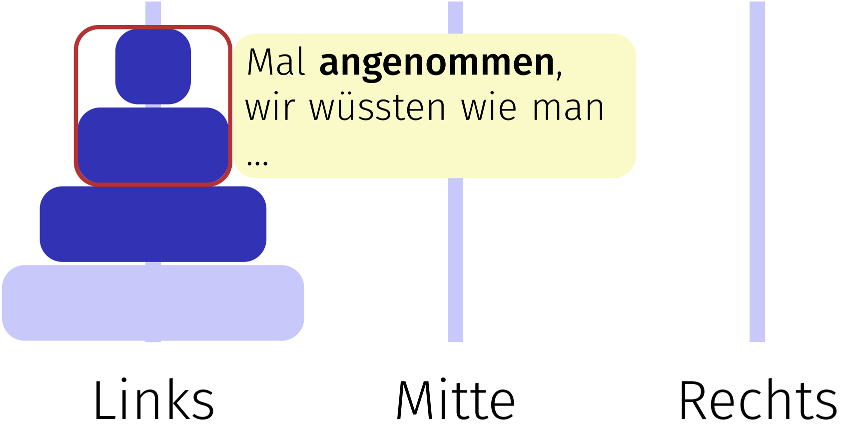
Die Türme von Hanoi - Rekursiv



Die Türme von Hanoi - Rekursiv

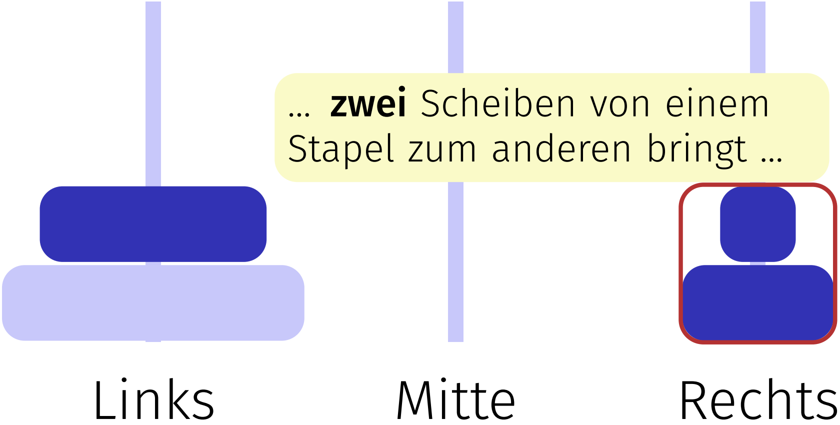


Die Türme von Hanoi - Rekursiv

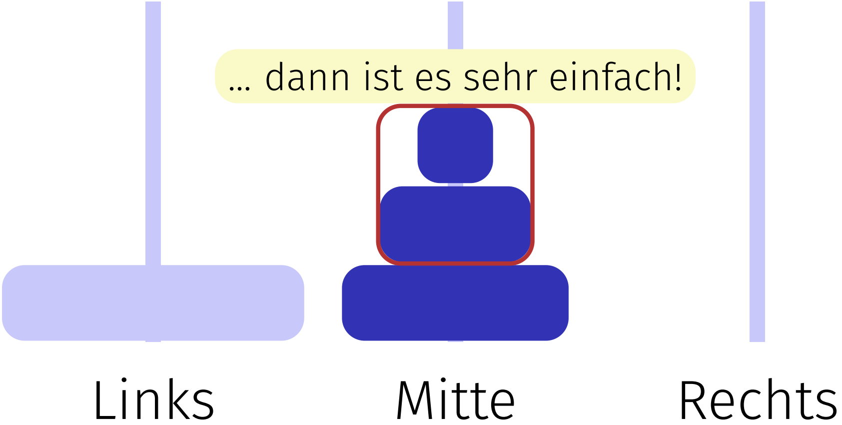


Die Türme von Hanoi - Rekursiv

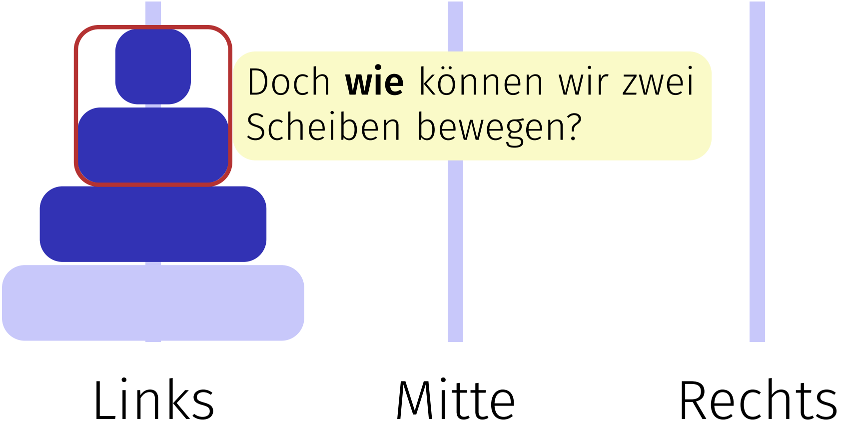
… **zwei** Scheiben von einem Stapel zum anderen bringt …



Die Türme von Hanoi - Rekursiv



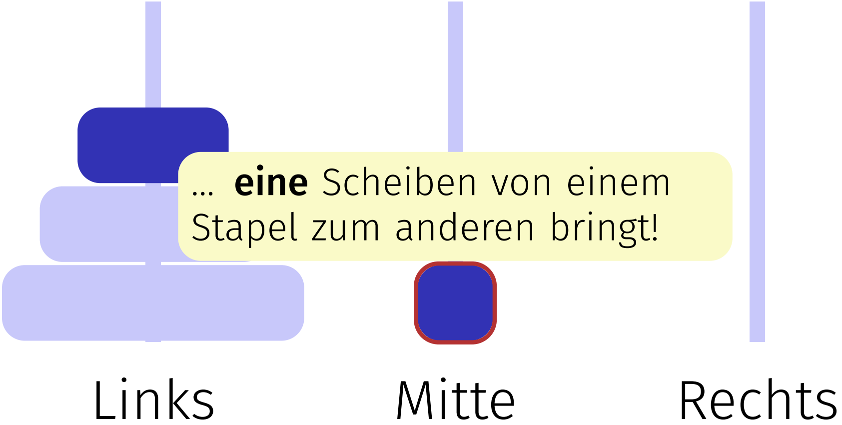
Die Türme von Hanoi - Rekursiv



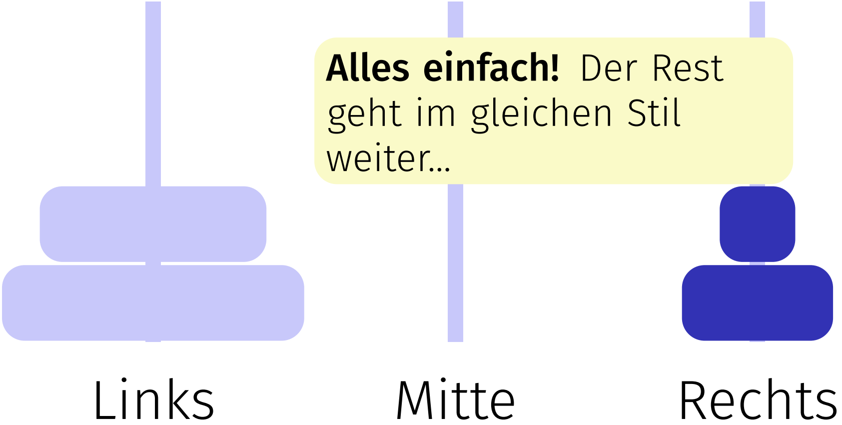
Die Türme von Hanoi - Rekursiv

Die Türme von Hanoi - Rekursiv

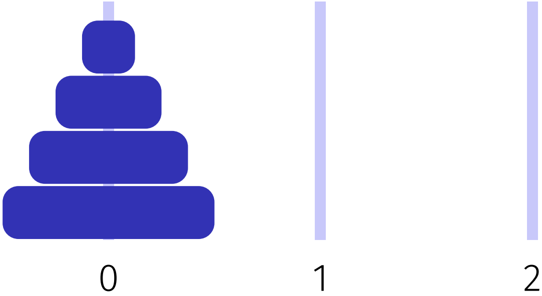
… **eine** Scheiben von einem Stapel zum anderen bringt!



Die Türme von Hanoi - Rekursiv



Die Türme von Hanoi - Code



Bewege 4 Scheiben von 0 nach 2 mit Hilfsstapel 1:

```
move(4, 0, 1, 2);
```

Die Türme von Hanoi - Code

```
move(4, 0, 1, 2);
```

==

1. Bewege 3 Scheiben von 0 nach 1 mit Hilfsstapel 2:
   `move(3, 0, 2, 1);`
2. Bewege 1 Scheibe von 0 nach 2
   `move(1, 0, 1, 2);`
3. Bewege 3 Scheiben von 1 nach 2 mit Hilfsstapel 0
   `move(3, 1, 0, 2);`

Die Türme von Hanoi - Code

```
public static void move(int n, int source, int aux, int d
  if (n==1){
    Out.println("move " + source + "->" + dest);
  } else {
    move(n-1, source, dest, aux);
    move(1, source, aux, dest);
    move(n-1, aux, source, dest);
  }
}
```