Komplexität eines Algorithmus, Grössenordnung, Landau-Symbole, Beispiel einer Komplexitätsberechnung (Mergesort)

7. KOMPLEXITÄT

Komplexität eines Algorithmus

Algorithmen verbrauchen Ressourcen

- Rechenzeit
- Speicher
- (Energie)

Wichtiges Ziel:

Ressourcenverbrauch minimieren

Beispiel einer Frage:

Wie hoch ist der Ressourcenverbrauch beim Berechnen des Medians mit Mergesort?

Präzisierte Frage:

Resourcenverbrauch im besten Fall, im schlechtesten Fall, im Durchschnitt?

Quelle: Vorlesung Informatik II D-ITET, Prof. Mattern

Begriffe

Betrachten Probleme mit einem gewissen ***Problemumfang n***

- Anzahl Eingabewerte, Anzahl der Bits der Eingabe

Aufwand (Zeit / Speicherplatz) typischerweise **von n abhängig**

- Wird daher als Funktion $f(n)$ angegeben

Beim Zeitaufwand wird i.A. **von konstanten Faktoren abstrahiert**

- z.B. $f(n) = n \log n$ statt (genauer) $3 + 7\, n \log n$
- Beim Speicherverbrauch wird oft nicht von Konstanten abstrahiert

Fälle

Aufwand eines Algorithmus i.A. nicht nur von Problemgrösse n, sondern von konkreten Eingabewerten abhängig, daher Unterscheidung:

- günstigster Fall („best case“)
- mittlerer Fall („average case“)
- ungünstigster Fall („worst case“)

Asymptotischer Aufwand

Oft ist man nur am asymptotischen Aufwand (für $n \to \infty$) als Funktion von n interessiert

Achtung: für „kleine“ n kann ein Algorithmus mit schlechterem asymptotischen Aufwand besser sein!

Komplexität eines Problems = geringstmöglicher Aufwand, der mit dem dafür besten Lösungsalgorithmus erreicht werden kann

Manche Probleme sind inhärent aufwändig / schwierig / „komplex“

Komplexitätsgrössenordnung

Zweck: Angabe der Grössenordnung der Komplexität eines Algorithmus als Funktion der Eingabegrösse

- Zeitkomplexität meist Anzahl "Schritte"
- Von unwesentlichen Konstanten wird abstrahiert

Schreibweise

- $O(1)$ konstant
- $O(\log n)$ logarithmisch
- $O(n)$ linear
- $O(n^2)$ quadratisch
- $O(n^k), k > 0$ polynomial
- $O(c^n)$ exponentiell

"gut"

"böse"

Man sagt
"Grössenordnung x"
"O von x"
"Gross-O von x"

O(x)

Gross-O Notation

Wir sagen ein Algorithmus f *ist* $O(g)$ (oder "von der Ordnung g") wenn für die exakte Komplexität von f gilt:

Es gibt eine Konstante c und eine positive ganze Zahl n_0, so dass

$$f(n) \leq c \cdot g(n) \quad \text{für alle } n \geq n_0$$

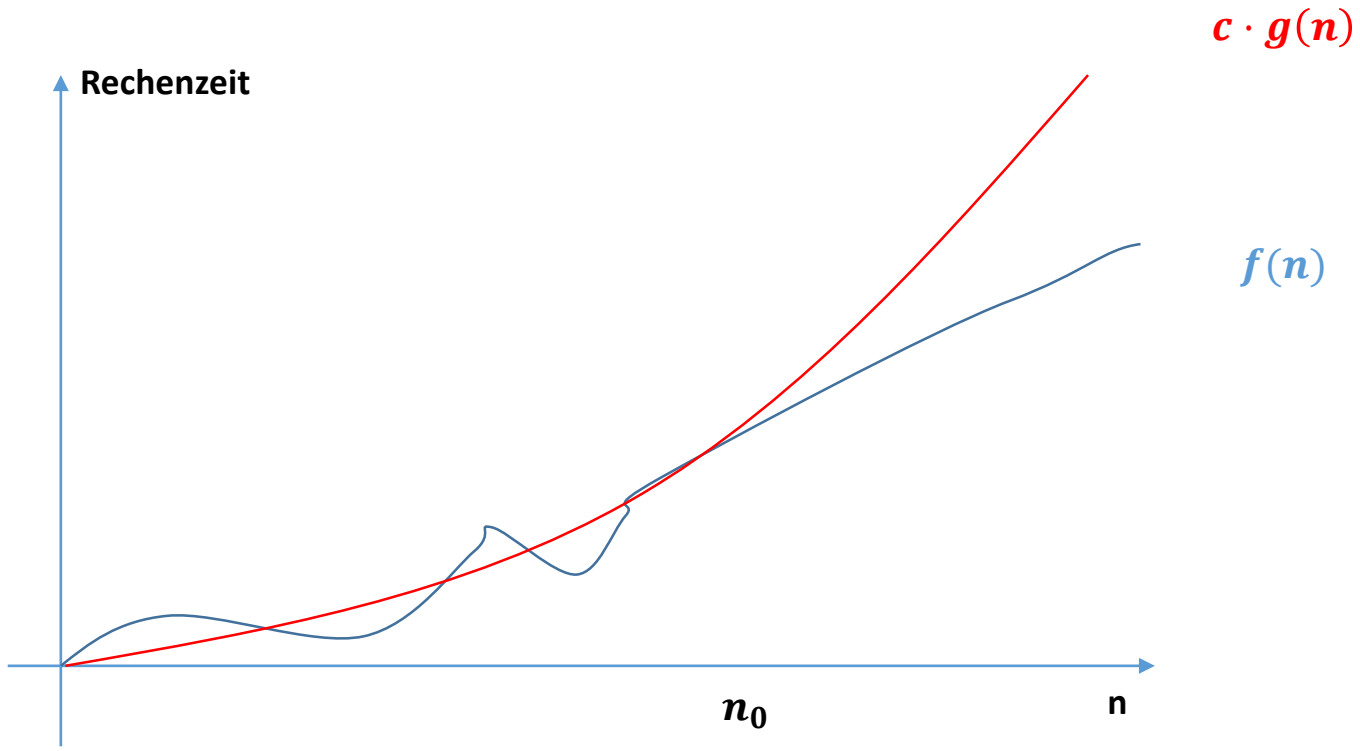
gilt.

Gross-O Notation

Es gibt eine Konstante $\boldsymbol{c}$ und eine positive ganze Zahl $\boldsymbol{n_0}$, so dass

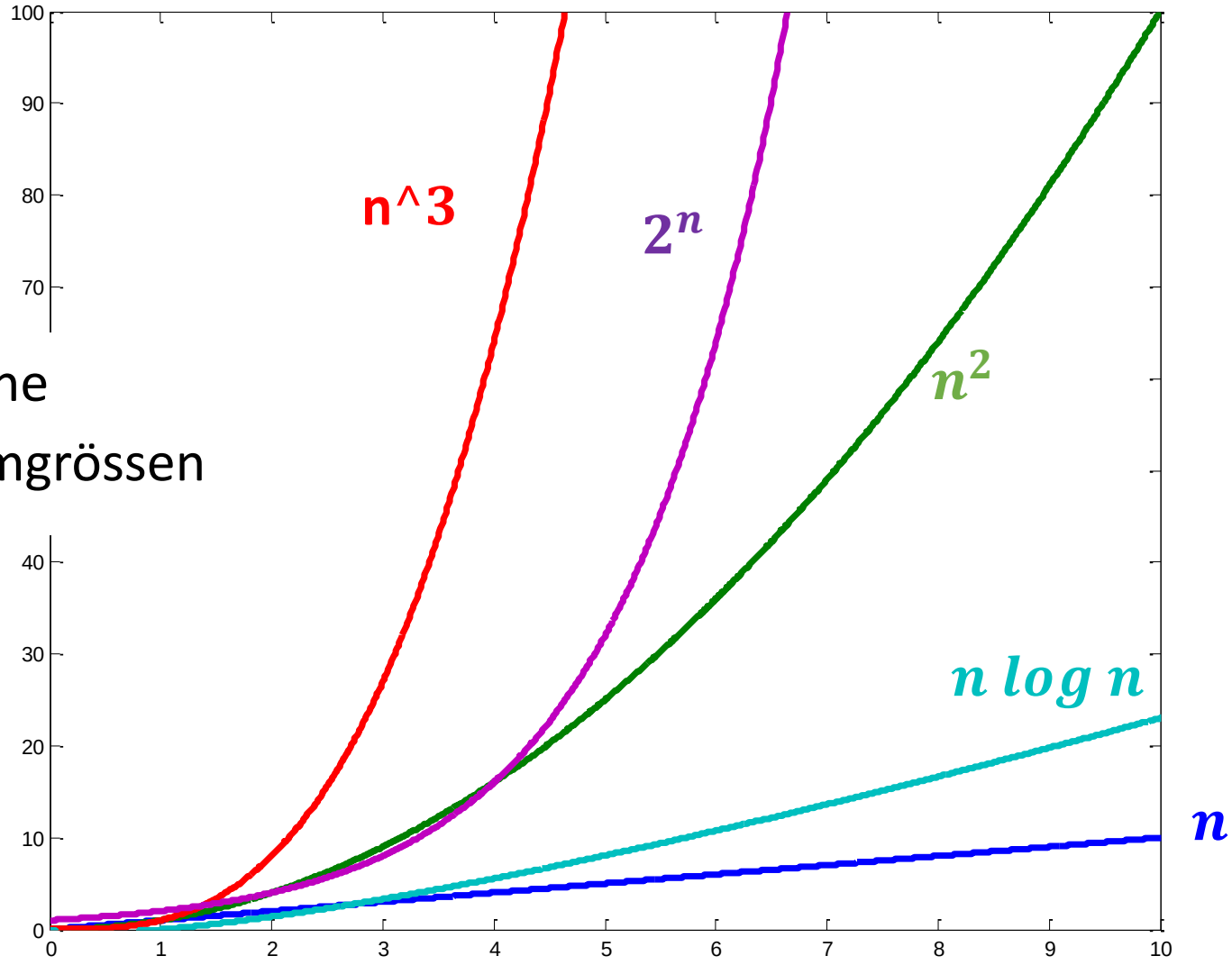
$$\boldsymbol{f(n) \leq c \cdot g(n)}$$
für alle $\boldsymbol{n \geq n_0}$

ist eine asymptotische Aussage. Charakterisiert nicht das Verhalten für kleine Problemgrössen.

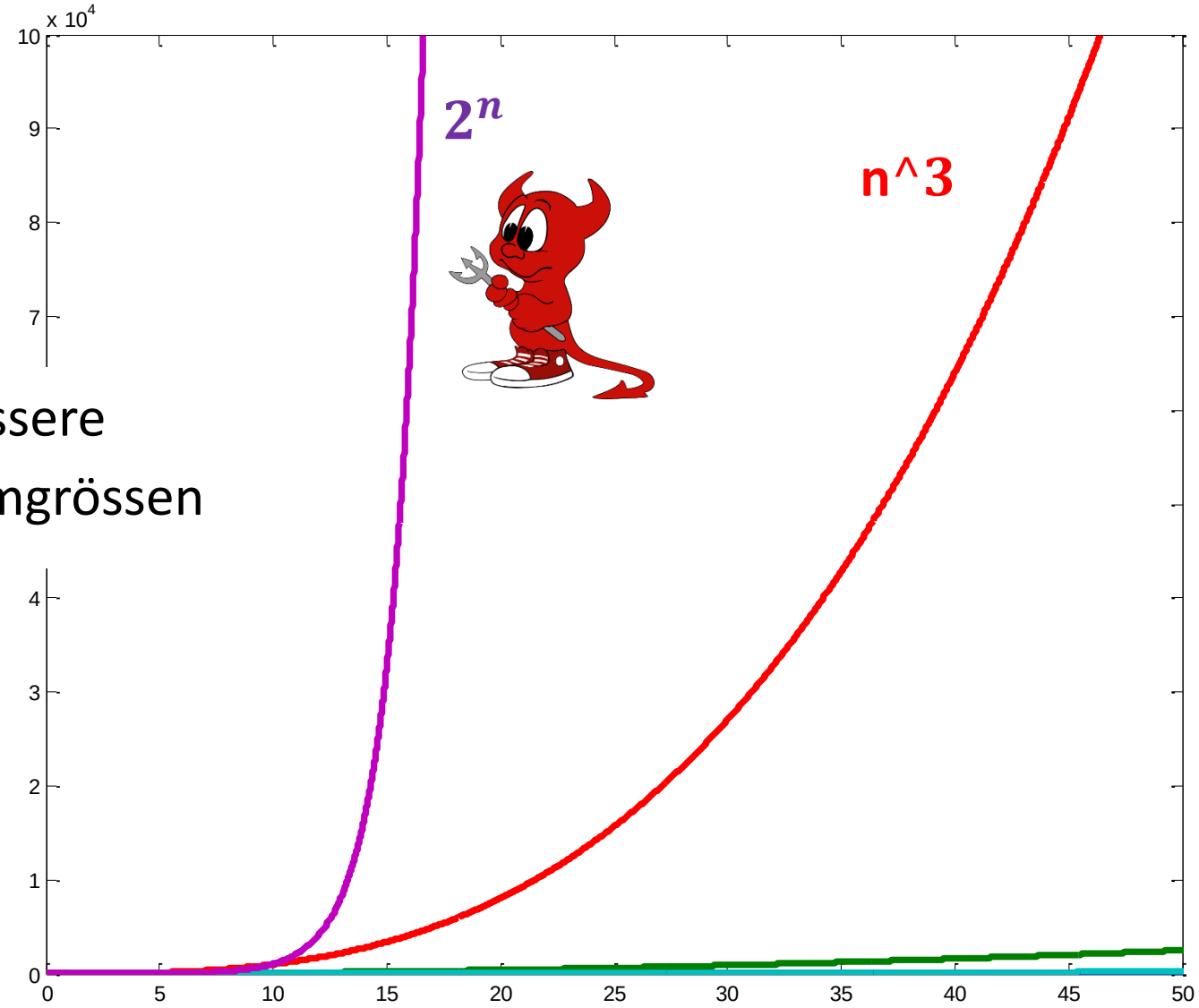


Laufzeiten und Problemgrössen

für kleine Problemgrössen



Laufzeiten und Problemgrössen



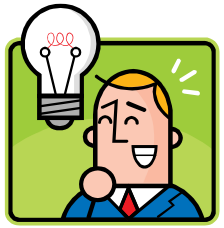
für grössere Problemgrössen

Zeitbedarf bei vergrössertem Problemumfang

Beispiel: Bei $1\mu s$ für n=1

	n=1	**n=100**	**n=10000**	**n=10^6**
$\log n$	$1\ \mu s$	$7\ \mu s$	$13\ \mu s$	$20\ \mu s$
n	$1\ \mu s$	$100\ \mu s$	$0.01\ s$	$1\ s$
n^2	$1\ \mu s$	$0.01\ s$	$1.7\ min$	$11.5\ Tage$
2^n	$1\ \mu s$	10^{14} Jahrh.	$\sim \infty$	$\sim \infty$

Eine gute Strategie?



… dann kaufe ich mir eben eine bessere Maschine

Wenn ich heute ein Problem der Grössenordnung N lösen kann, dann kann ich mit einer zehn mal (!) so schnellen Maschine …

f(n)	**alter → neuer Problemumfang**
$\log n$	$N \rightarrow N^{10}$
n	$N \rightarrow 10\ N$
$n \log n$	$N \rightarrow \text{fast } 10\ N\ (\rightarrow 10N\ f\ddot{u}r\ N \rightarrow \infty)$
n^2	$N \rightarrow \sqrt{10}\ N \approx 3.16\ N$
n^3	$N \rightarrow 10^{\frac{1}{3}} N \approx 2.15\ N$
2^n	$N \rightarrow N + \log_2 10 \approx N + 3.3$

Komplexitätsklassen formeller*

Eigentlich bezeichnet das *Landau-Symbol*

$$O(f) = \{h \mid \exists c > 0, \exists n_0 \in N: \forall n > n_0: h(n) \leq cf(n)\}$$

eine Klasse von Funktionen.

Man schreibt zwar manchmal salopp $g = O(f)$, meint jedoch g $\in O(f)$

Es gibt auch Beschränkung von unten

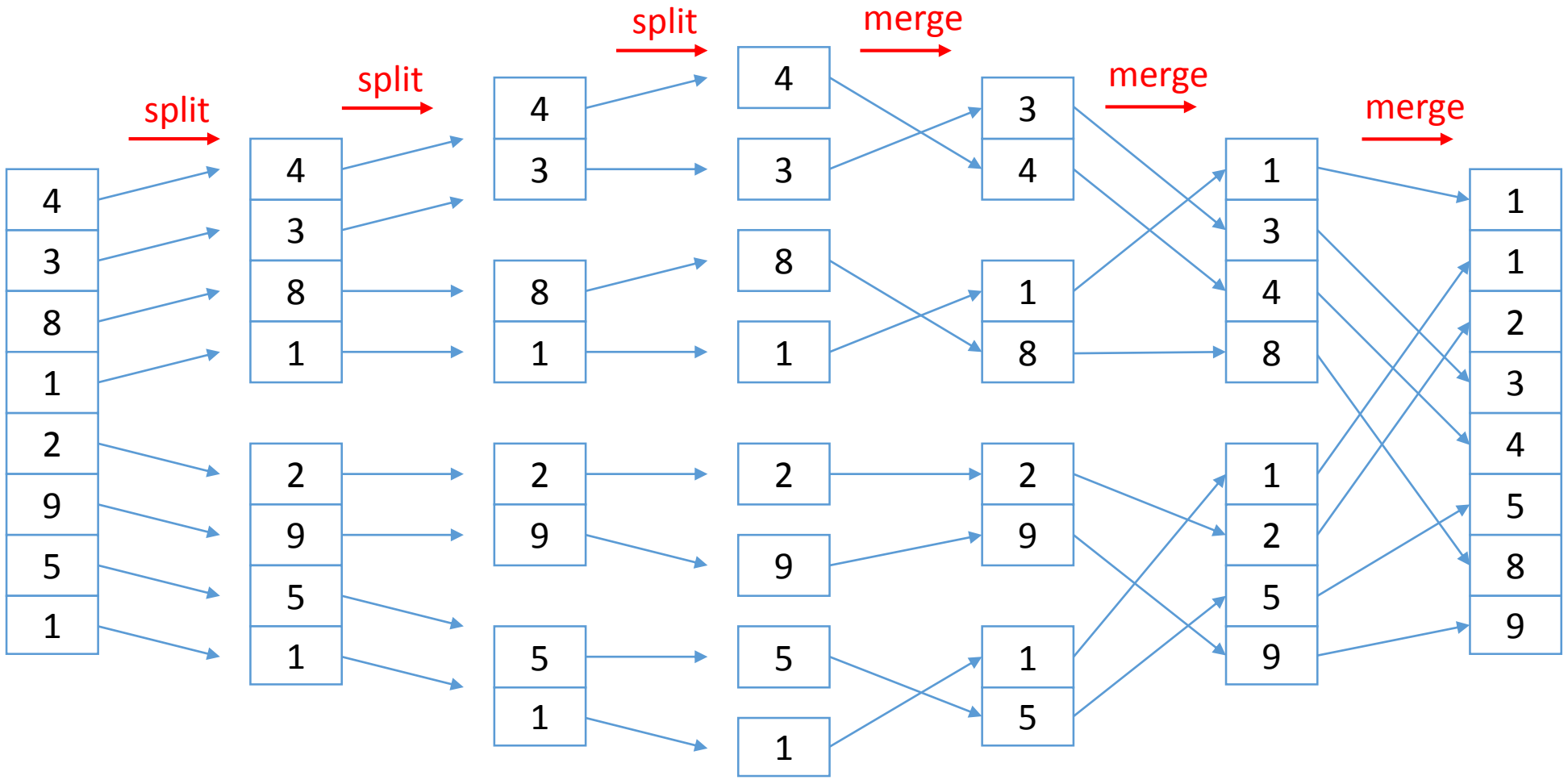
$$\Omega(f) = \{h | \exists c > 0, \text{für unendlich viele } n: h(n) \geq cf(n)\}$$

und "beides"

$$\Theta(f) = \Omega(f) \cap O(f)$$

Beispiel einer Komplexitätsberechnung

Mergesort, vereinfachende Annahme: $n = 2^d$



Beispiel einer Komplexitätsberechnung

Aufwand für Merge

$$T(n) = n + 2 \cdot T\left(\frac{n}{2}\right), \text{ wenn } n > 1$$

$$T(1) = 1$$

Beispiel einer Komplexitätsberechnung

Komplexitätsberechnung Rekursion → Iteration

$$
\begin{aligned}
T(2^d) &= 2^d + 2 \cdot T(2^{d-1}) \\
&= 2^d + 2 \cdot \left(2^{d-1} + 2\, T\left(2^{d-2}\right)\right) \\
&= 2^d + 2 \cdot \left(2^{d-1} + 2 \cdot (2^{d-2} + 2 \cdot T\left(2^{d-3}\right)\right) \\
&= 2^{0} \cdot 2^d + 2^1 \cdot 2^{d-1} + 2^2 \cdot 2^{d-2} + \cdots + 2^{d-1} \cdot 2 + \; 2^d T(1) \\
&= 2^d \cdot d + 2^d = n \log_2 n + n \\
&= O(n \log n)
\end{aligned}
$$

Bäume, Binäre Suchbäume, Heap, Effizienter Online-Median

8. DYNAMISCHE DATENSTRUKTUREN II

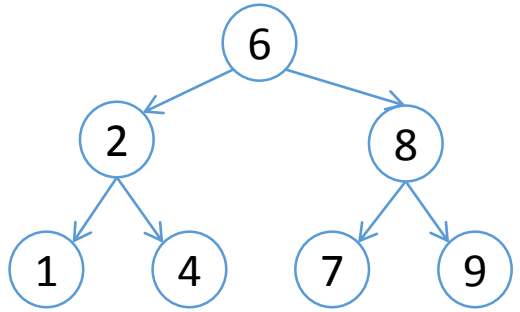
Bäume – Motivation

Erinnerung an die Fallstudie Online-Statistik

- Median konnte nicht effizient berechnet werden
- Verlinkte Listen schaffen Abhilfe bzgl. effizienterem sortierten Einfügen von Elementen. Laufzeit für die Suche eines Elementes wird jedoch nicht verringert

Suchbäume können verwendet werden, um indizierte Elemente effizient zu suchen

- Was nützt uns das für den Median?
- Antwort etwas später



Literatur: Ottmann, Widmayer, Algorithmen und Datenstrukturen, Kapitel 5

Bäume – Motivation

Bäume sind

- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter*, azyklischer Graph.

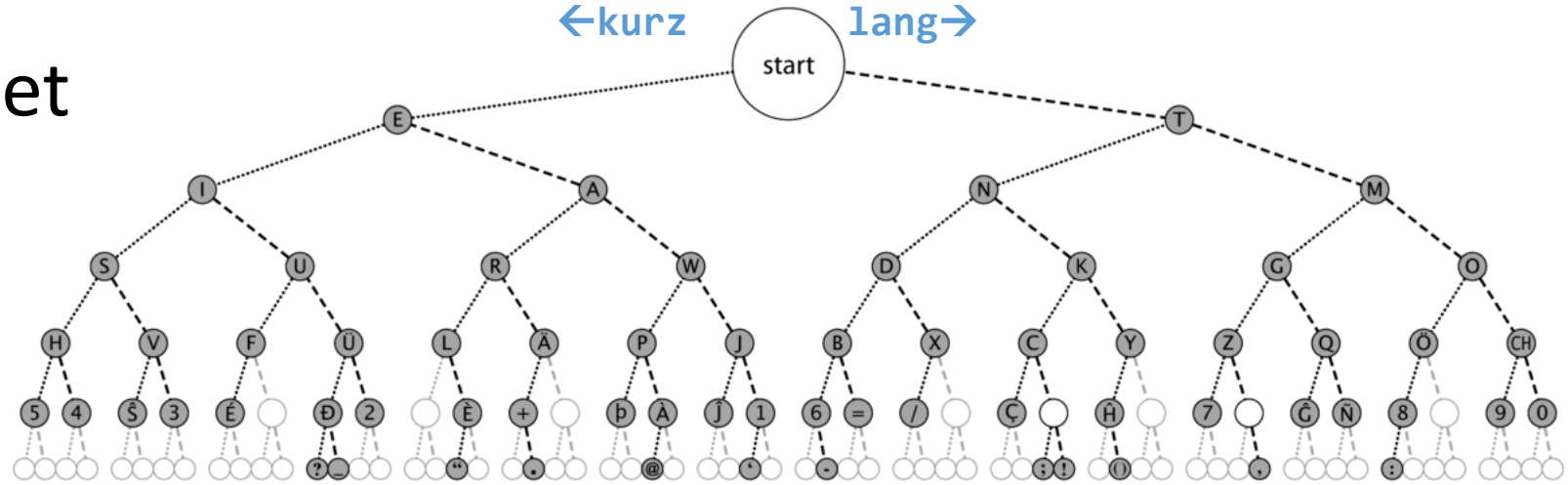
Verwendung

- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffmann Code
- **Suchbäume**: ermöglichen effizientes Suchen eines Elementes

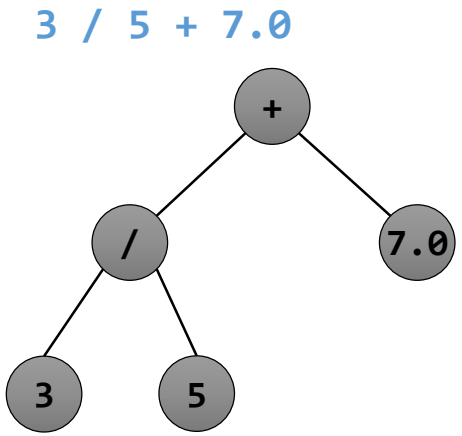
*manche Autoren betrachten auch ungerichtete Bäume (wir nicht)

Beispiele

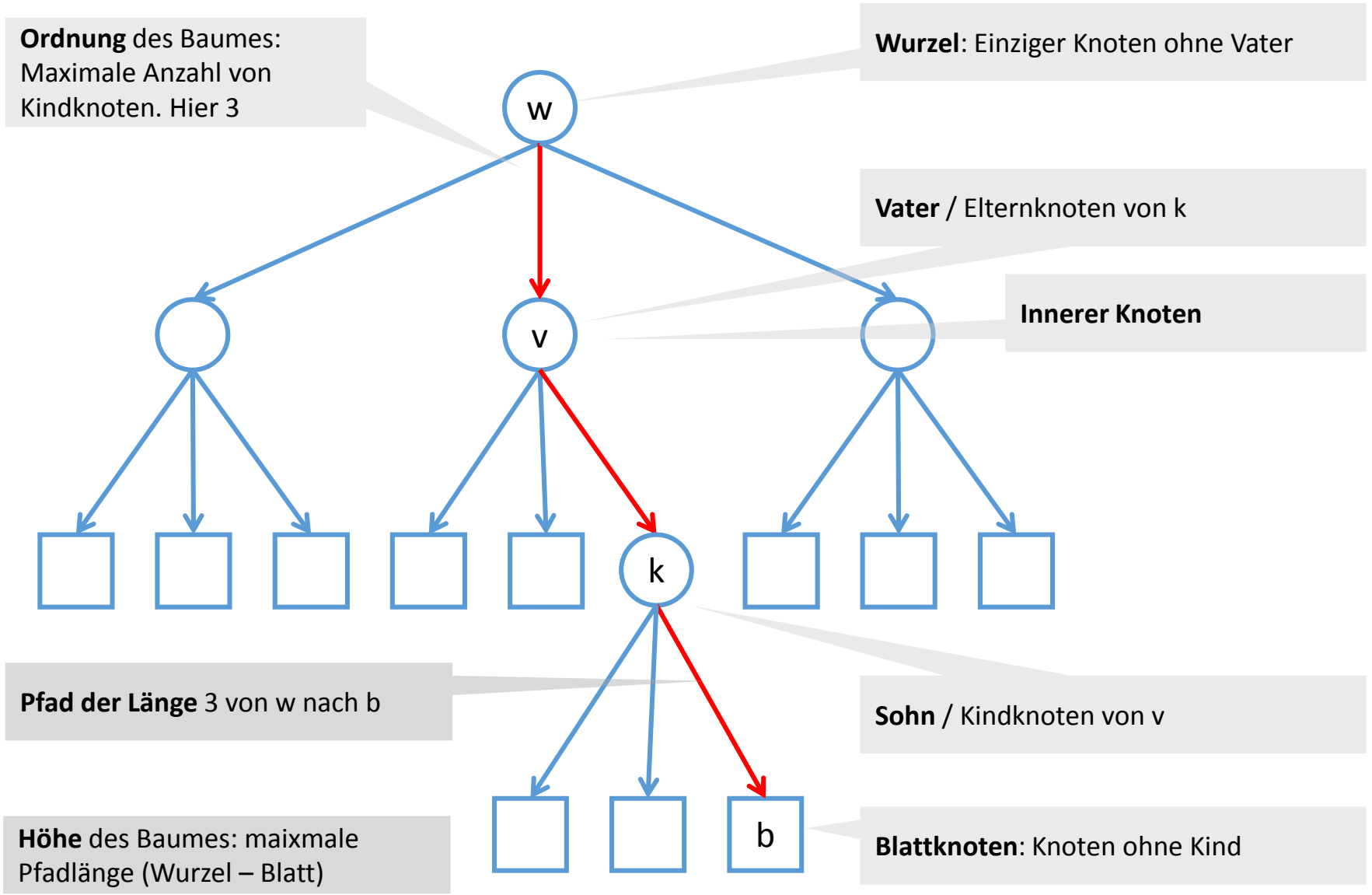
Morsealphabet



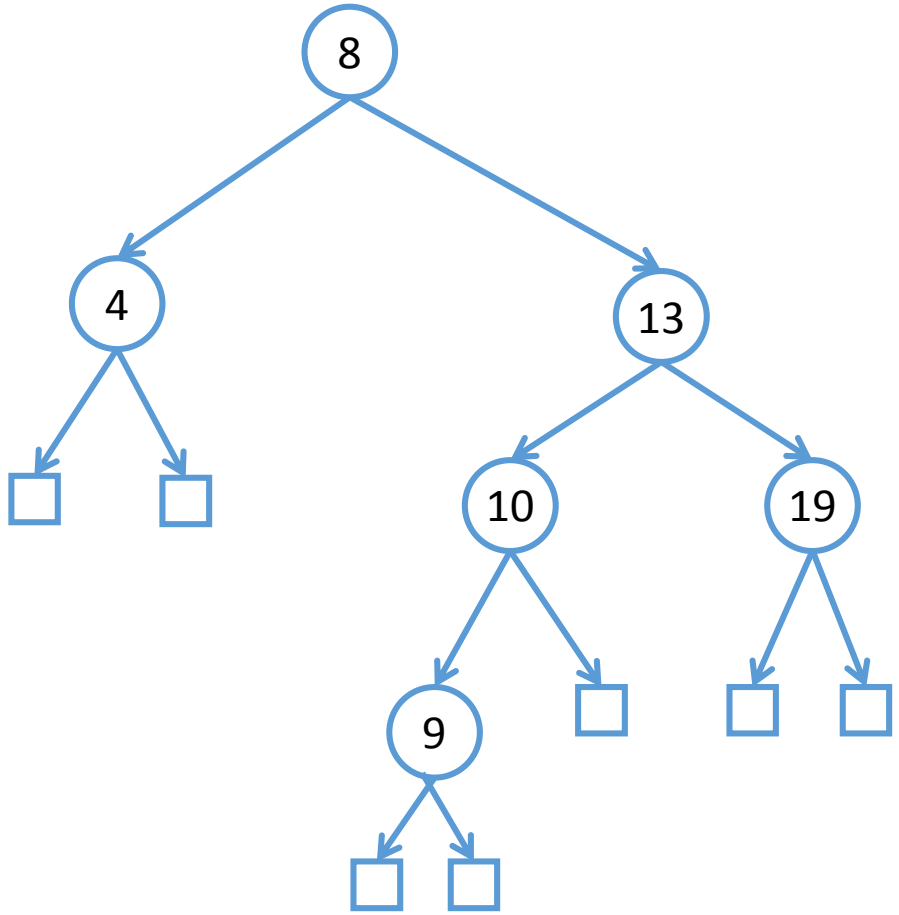
Ausdrucksbaum



Bäume: Nomenklatur



Binärer Suchbaum



$(-\infty, \mathbf{4}), (\mathbf{4}, \mathbf{8}), (\mathbf{8}, \mathbf{9}), (\mathbf{9}, \mathbf{10}), (\mathbf{10}, \mathbf{13},), (\mathbf{13}, \mathbf{19}), (\mathbf{19}, \infty)$

Baum der Ordnung 2

Knoten beherbergen paarweise verschiedene Schlüssel (und potentiell andere Nutzdaten)

Schlüssel im linken Teilbaum kleiner als am Knoten

Schlüssel im rechten Teilbaum grösser als am Knoten

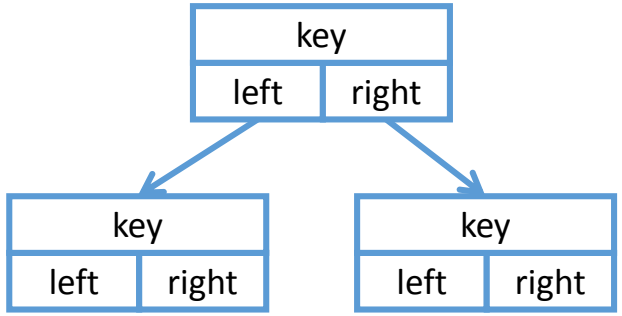
(Leere*) Blätter repräsentieren Schlüsselintervalle

*ausser bei Blattsuchbäumen, betrachten wir hier nicht

Datenstruktur Suchknoten

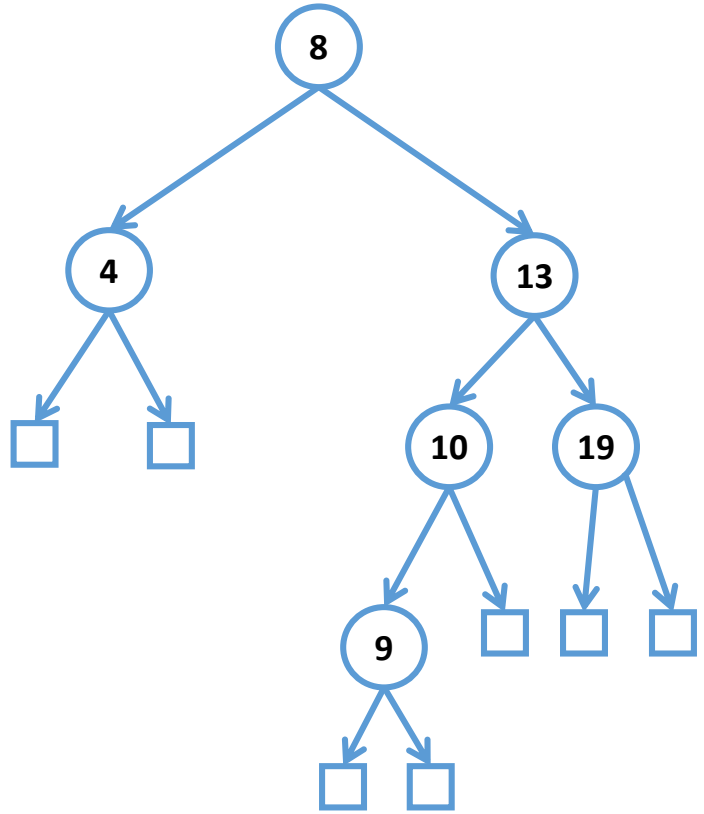
Ähnlich wie bei den Listen besteht ein Baum aus verketteten / verflochtenen Instanzen einer Datenstruktur Knoten (SearchNode)

```
public class SearchNode {
	int key;		// Schlüssel
	SearchNode left;	// linker Teilbaum
	SearchNode right;	// rechter Teilbaum

	// Konstruktor: Knoten ohne Nachfolger
	SearchNode(int k){
		key = k;
		left = right = null;
	}
}
```



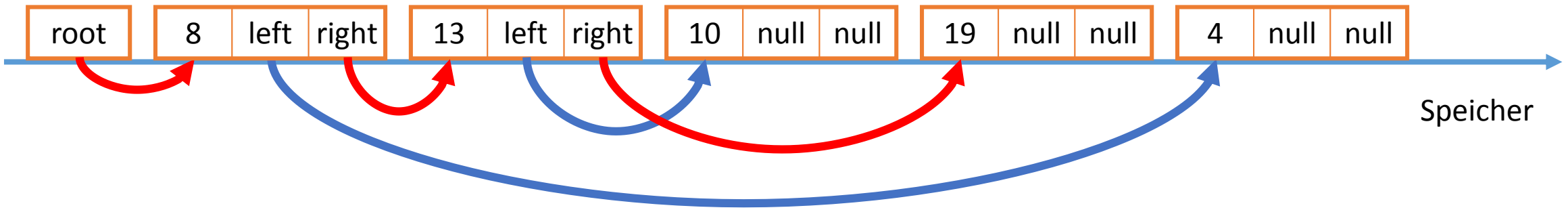
Datenstruktur Suchbaum

```
public class SearchTree {
    SearchNode root;// Wurzelknoten

    // Konstruktor: leerer Suchbaum
    SearchTree() {
        root = null;
    }

    // Gibt Knoten mit Schlüssel k zurück. Wenn nicht existiert: null.
    public SearchNode Search (int k){
        SearchNode n = root;
        while (n != null && n.key != k)
        {
            if (k < n.key) n = n.left;
            else n = n.right;
        }
        return n;
    }

    ... // Einfügen, Löschen
}
```



Sieht effizient aus.
Stimmt das (immer)?

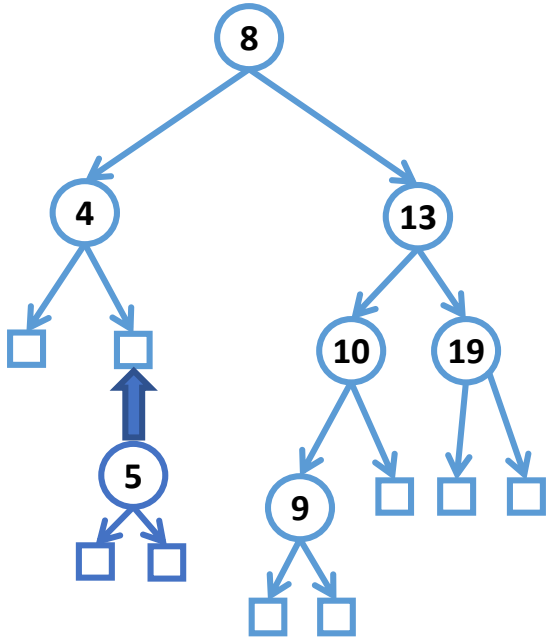
Baum Im Speicher



Knoten Einfügen

```
// Fügt Knoten mit Schlüssel k ein. Gibt erzeugten Knoten zurück.
// null, wenn Knoten mit Schlüssel k bereits existiert
public SearchNode Insert (int k) {
    if (root == null)
        return root = new SearchNode(k);
    SearchNode t = root;
    while (true){
        if (k == t.key)
            return null; // schon vorhanden
        if (k < t.key){
            if (t.left == null)
                return t.left = new SearchNode(k);
            else
                t = t.left;
        }
        else { // k > t.key
            if (t.right == null)
                return t.right = new SearchNode(k);
            else
                t = t.right;
        }
    }
}
```

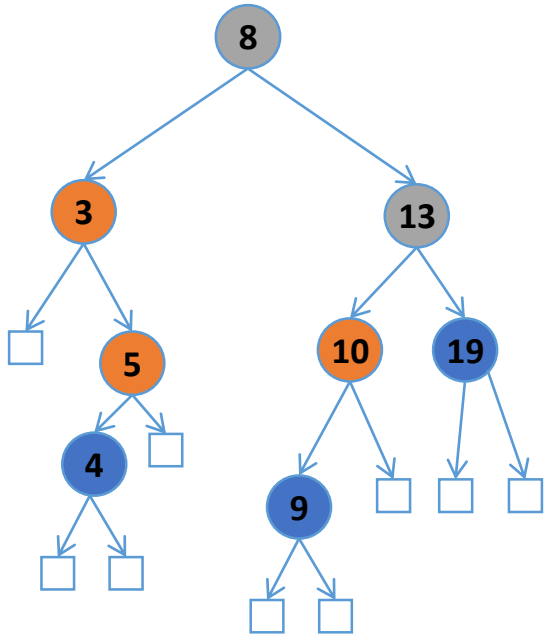
Traversiere Baum bis zum passenden Intervall-Blatt, ersetze Intervallblatt mit Knoten



Knoten Löschen

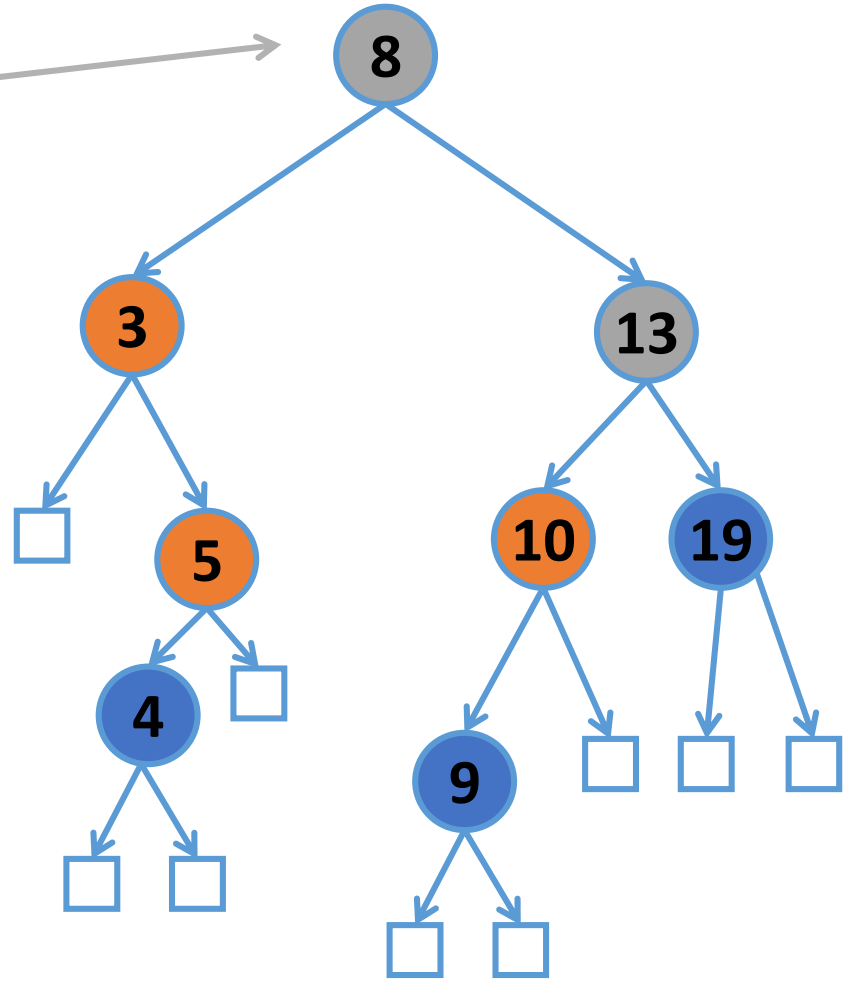
Drei Fälle

1. Knoten hat keine Kinder
   - Knoten entfernen
2. Knoten hat nur ein Kind
   - Knoten durch Kind ersetzen
3. Knoten hat zwei Kinder
   - Knoten durch einen *symmetrischen Nachfolger* ersetzen
   - Symmetrischer Nachfolger:
     Knoten im rechten (linken) Teilbaum, welcher am weitesten links (rechts) steht.
   - Korrespondiert mit dem kleinsten (grössten) Schlüssel, welcher gerade noch grösser (kleiner) als der Schlüssel des zu entfernenden Knotens ist
   - Symmetrischer Nachfolger hat maximal ein Kind



Knoten Löschen

```
public boolean Delete (int k) {
    SearchNode n = root;
    if (n != null && n.key == k) {
        root = SymmetricDesc(root);
        return true;
    }
    else {
        while (n != null) {
            if (n.left != null && k == n.left.key) {
                n.left = SymmetricDesc(n.left);
                return true;
                }
            else if (n.right != null && k == n.right.key) {
                n.right = SymmetricDesc(n.right);
                return true;
                }
            else if (k < n.key)
                n = n.left;
            else
                n = n.right;
        }
        return false;
    }
}
```



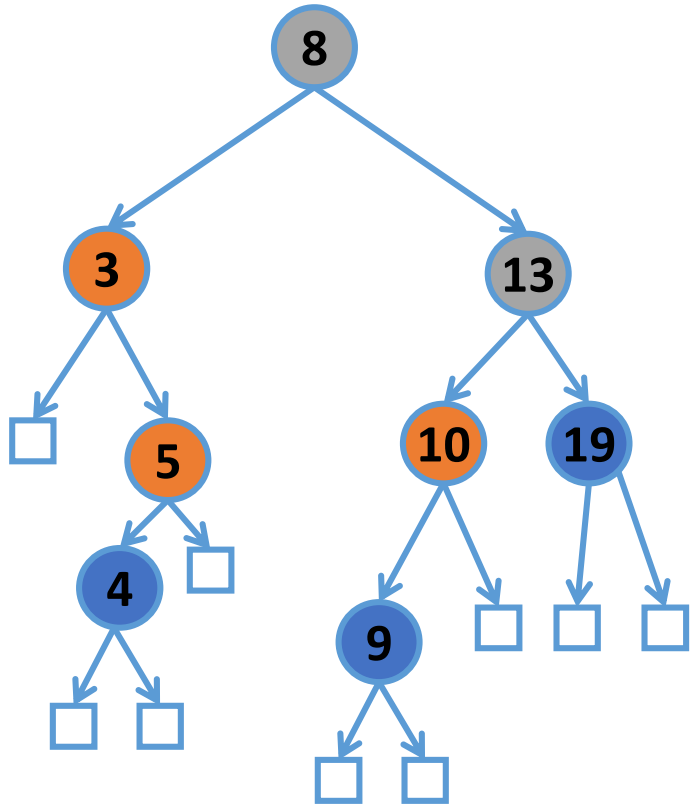
Symmetrischer Nachfolger

```
public SearchNode SymmetricDesc(SearchNode node){
    if (node.left == null)
        return node.right;
    if (node.right == null)
        return node.left;
    SearchNode n = node;
    SearchNode parent = null;
    n = n.right;
    while (n.left != null) {
        parent = n;
        n = n.left;
    }
    if (parent != null) {
        parent.left = n.right;
        n.left = node.left;
        n.right = node.right;
    }
    else
        n.left = node.left;
    return n;
}
```

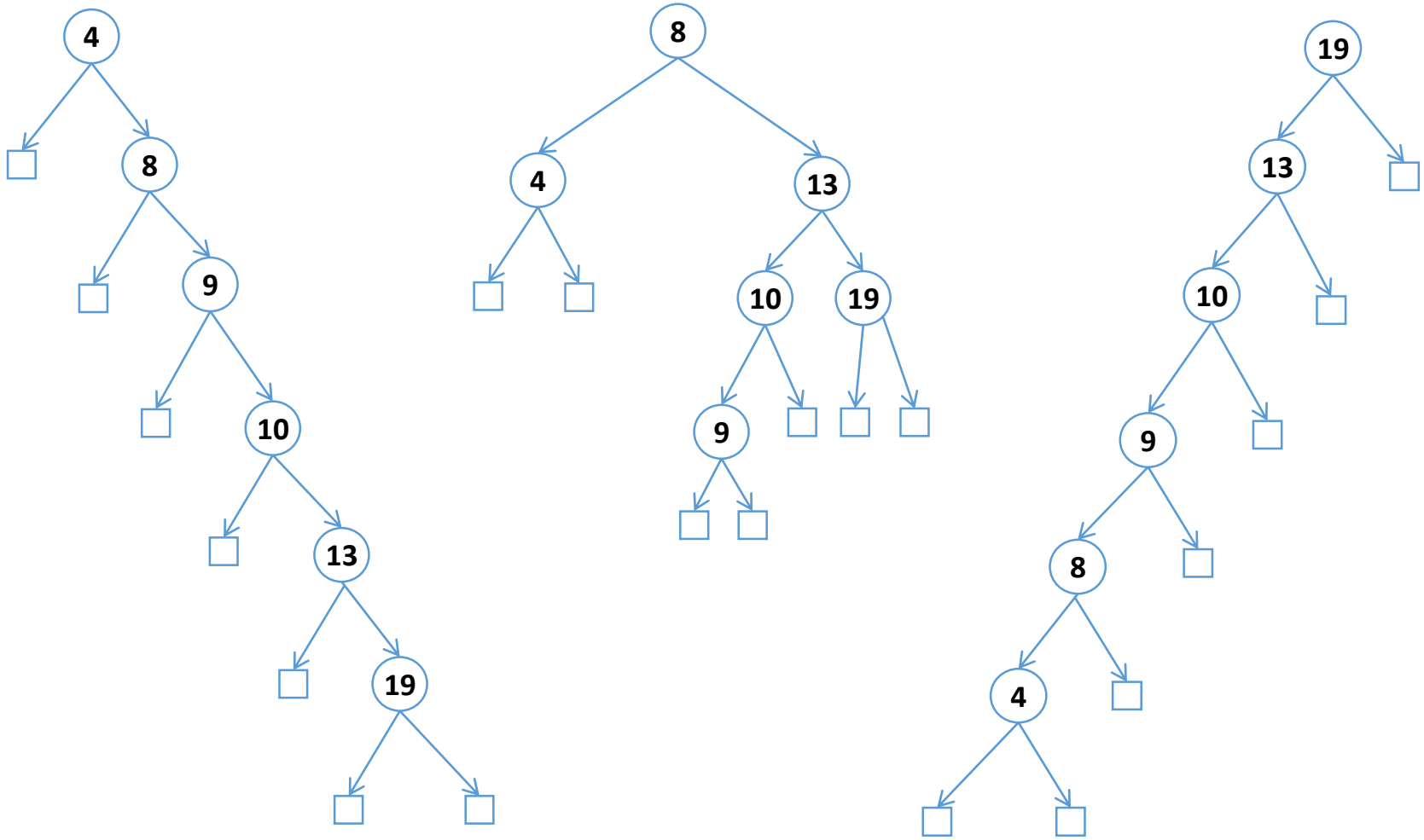
Wenn der Knoten nur ein Kind hat, gib dieses zurück.
Wenn der Knoten kein Kind hat, gib null zurück

Suche den symmetrischen Nachfolger von node

Entferne den symmetrischen Nachfolger:
ersetze ihn durch sein rechtes Kind!
Ersetze Kinder des gelöschten Knotens durch die Kinder von node.



Degenerierte Bäume



Binäre Bäume können, je nach Updateoperationen, zu linearen Listen degenerieren! Folgerung?

Balancierte Bäume

Komplexität von Suchen, Einfügen und Löschen eines Knoten in binären Suchbäumen *im Mittel* $O(\log_2 n)$

Worst case: $O(n)$ bei degeneriertem Baum

Verhinderung der Degenerierung: künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes: worst case auch $O(\log_2 n)$

Balancieren: garantiere, dass ein Baum mit n Knoten stets eine Höhe von $\mathrm{O}(\log n)$ hat.

Z.B. AVL Bäume / Rot-Schwarz-Bäume etc. Wir behandeln das nicht.