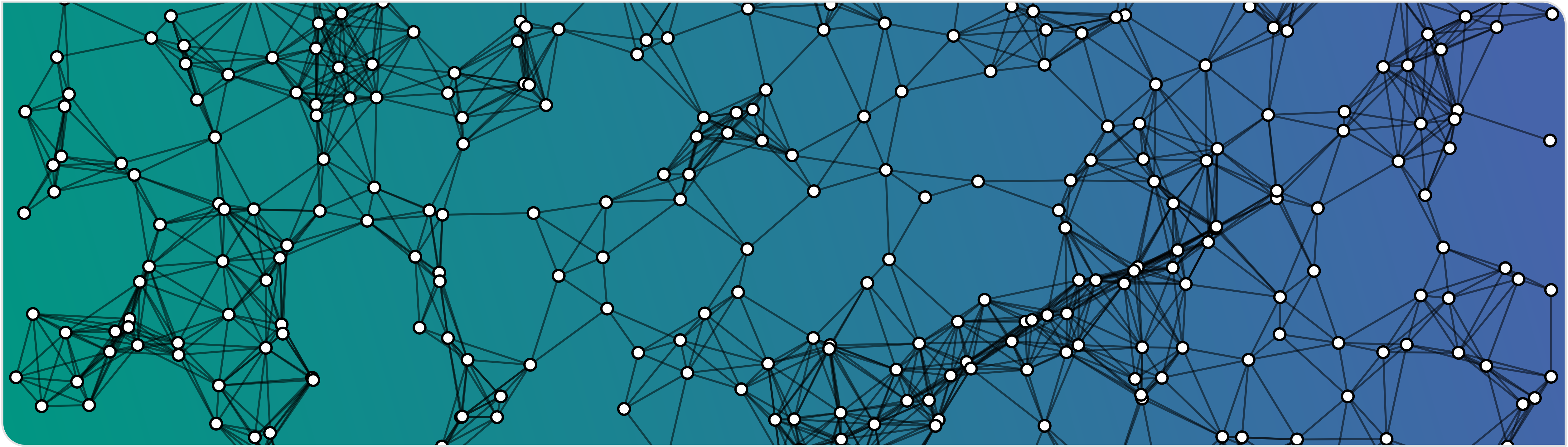
Karlsruher Institut für Technologie

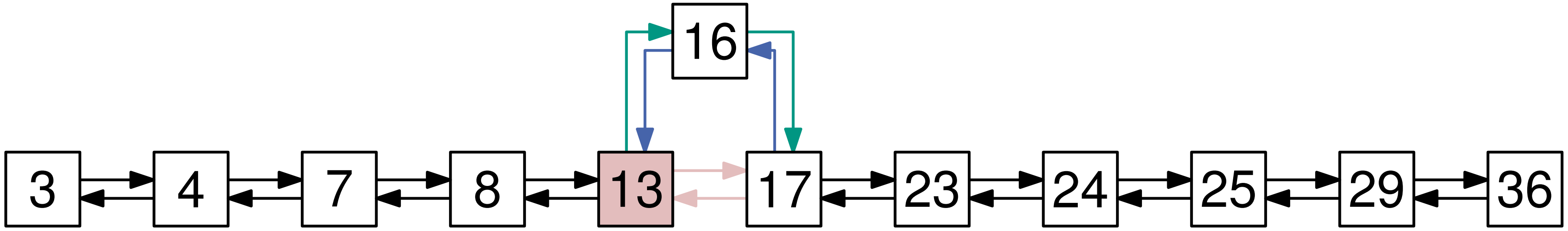
# Algorithmen 1

## Übung 5
## Suchbäume & Suche in Graphen



# Listen und Binäre Suche

- In Suchbäumen wollen wir die Vorteile von Listen und der binären Suche vereinen
  - Listen erlauben schnelles Einfügen
  - das erfordert Zugang zum Vorgänger Element



# Listen und Binäre Suche

Karlsruher Institut für Technologie

- In Suchbäumen wollen wir die Vorteile von Listen und der binären Suche vereinen
  - Listen erlauben schnelles Einfügen
  - das erfordert Zugang zum Vorgänger Element
  - Listen erlauben einfaches Löschen
  - das erfordert Zugang zum zu löschenden Element

In Listen können wir schnell umsortieren, wenn wir die Elemente schon in der Hand haben.

Wie können wir in Listen **schnell** suchen?

- Finde Elemente mit linearer Suche. Beispiel: Finde $13$.

$O(n)$

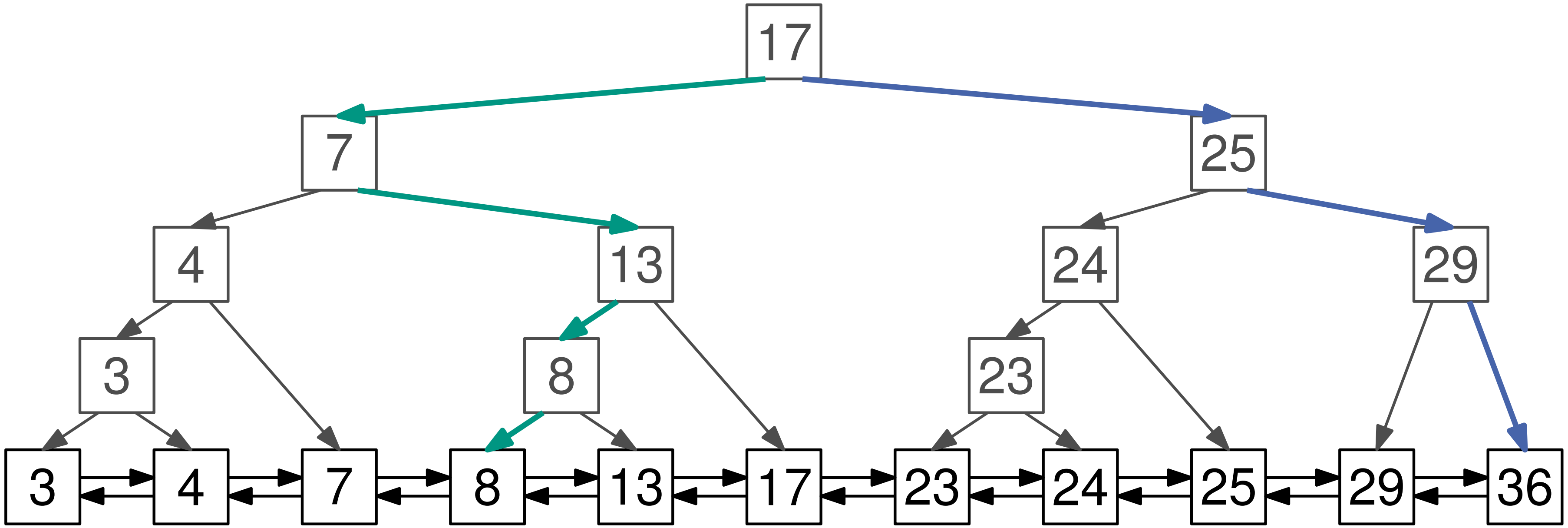
$$\boxed{3} \leftrightarrows \boxed{4} \leftrightarrows \boxed{7} \leftrightarrows \boxed{8} \leftrightarrows \boxed{13} \leftrightarrows \boxed{17} \leftrightarrows \boxed{23} \leftrightarrows \boxed{24} \leftrightarrows \boxed{25} \leftrightarrows \boxed{29} \leftrightarrows \boxed{36}$$

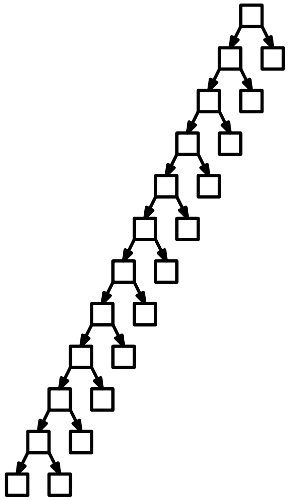
# Listen und Binäre Suche

- In Suchbäumen wollen wir die Vorteile von Listen und der binären Suche vereinen
  - Speed-Up der Suche durch Simulation von binärer Suche wie in Arrays
  - Beispiel: Binäre Suche nach 8

Aktueller Suchbereich Mittleres Element

Größe des Suchbereichs halbiert sich in jedem Schritt $\rightarrow O(\log(n))$



  - Idee: Speichere alle möglichen Suchen als Baum ab
  - Problem(?): Blätter hängen in unterschiedlicher Tiefe

Kann das beliebig schlimm werden?

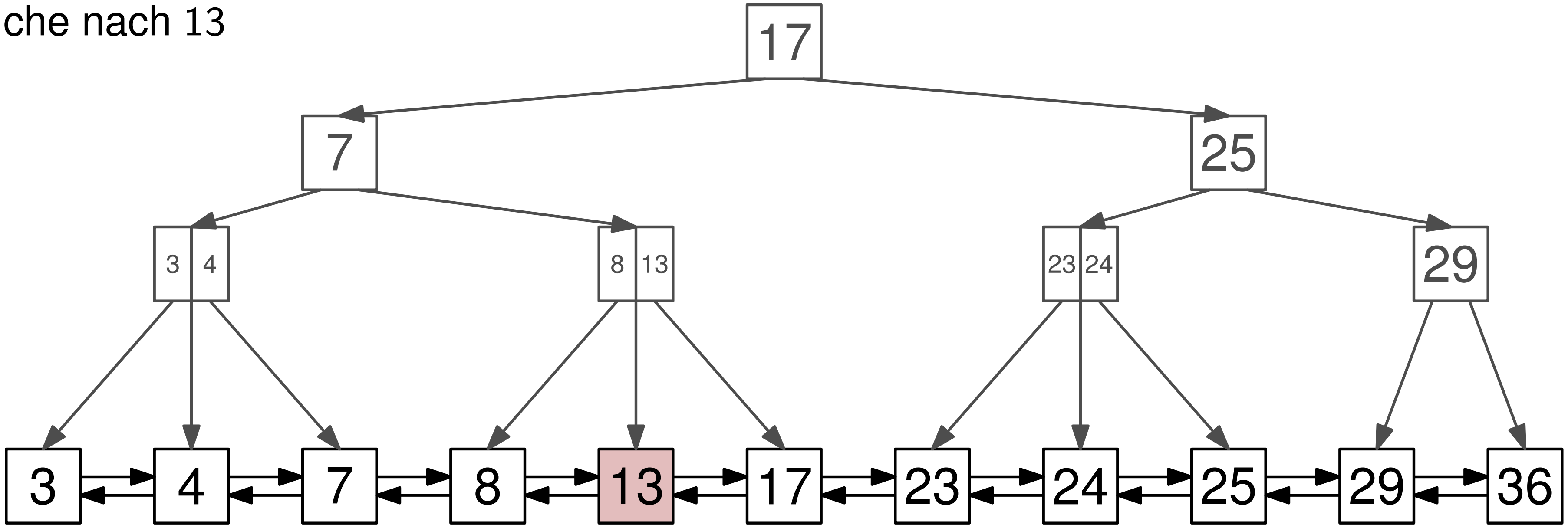
Nicht beim initialen Aufbau...

... aber beim Einfügen / Löschen?

# $(2, 3)$-Bäume

- Idee: **fast** binär und dafür balanciert
  - jeder Knoten hat 2 oder 3 Kinder
  - dadurch können wir dafür sorgen, dass alle Blätter die gleiche Tiefe haben

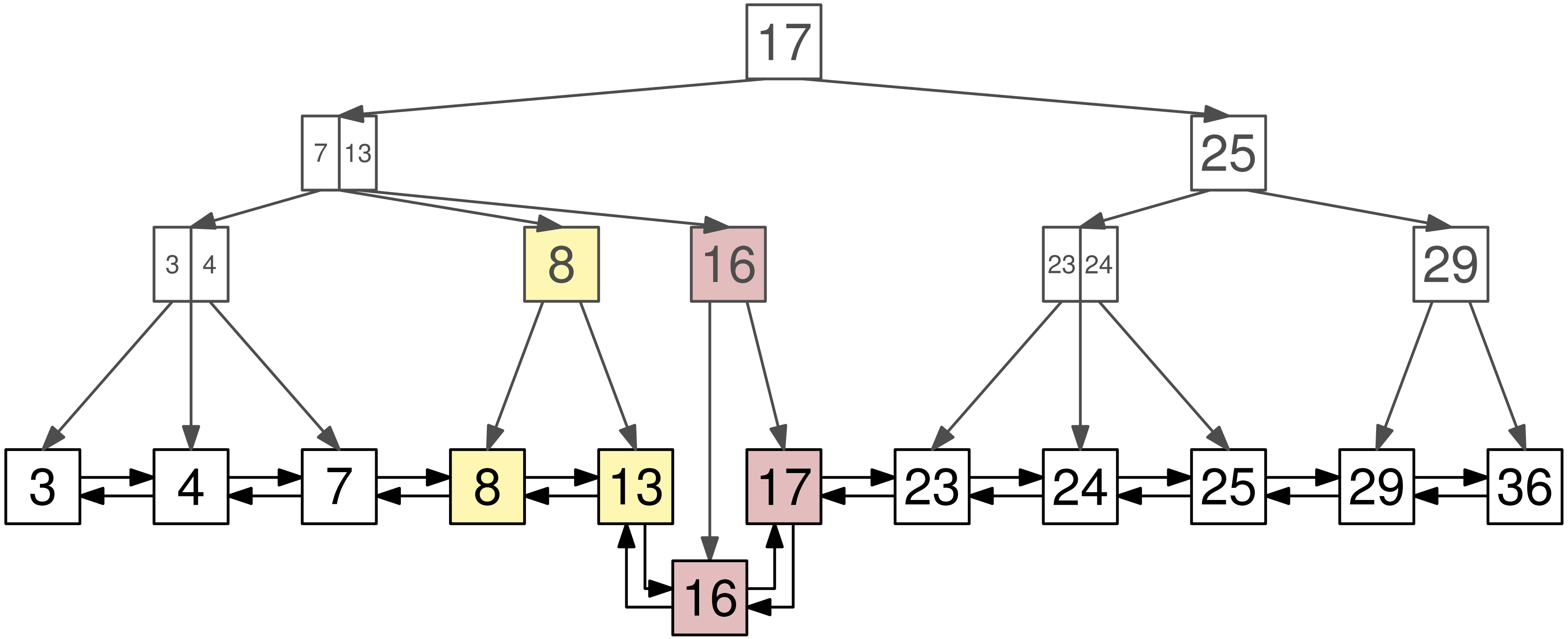
Beispiel: Suche nach 13




- Knoten halten ggf. zwei Schlüssel $k_1, k_2$. Abstieg: Links ($x \leq k_1$), Mitte ($k_1 < x \leq k_2$), Rechts ($k_2 < x$)

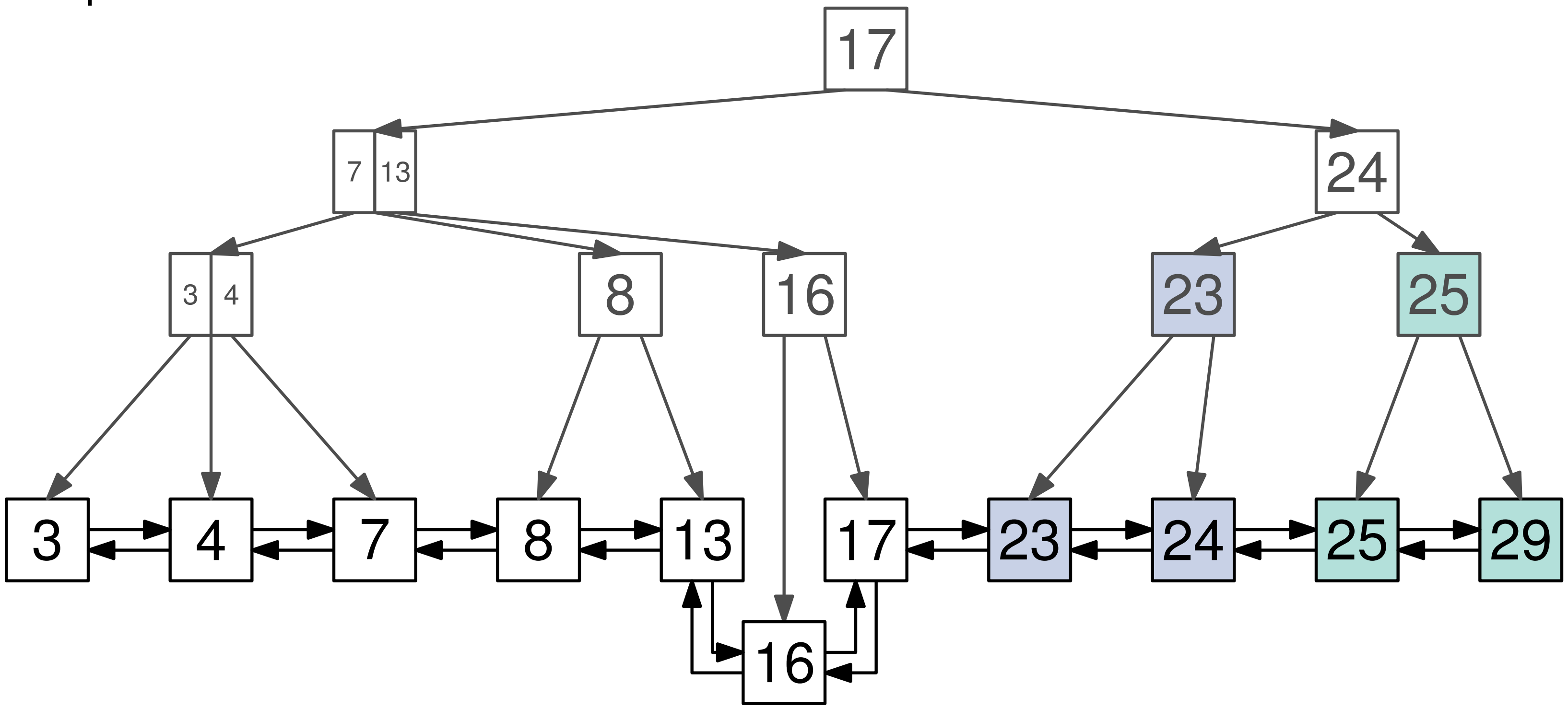
# $(2, 3)$-Bäume – Einfügen & Löschen

- Einfügen: Neues Blatt an Elter vom Nachfolger anhängen und danach (rekursiv) aufspalten wenn nötig   Beispiel: 16 Einfügen



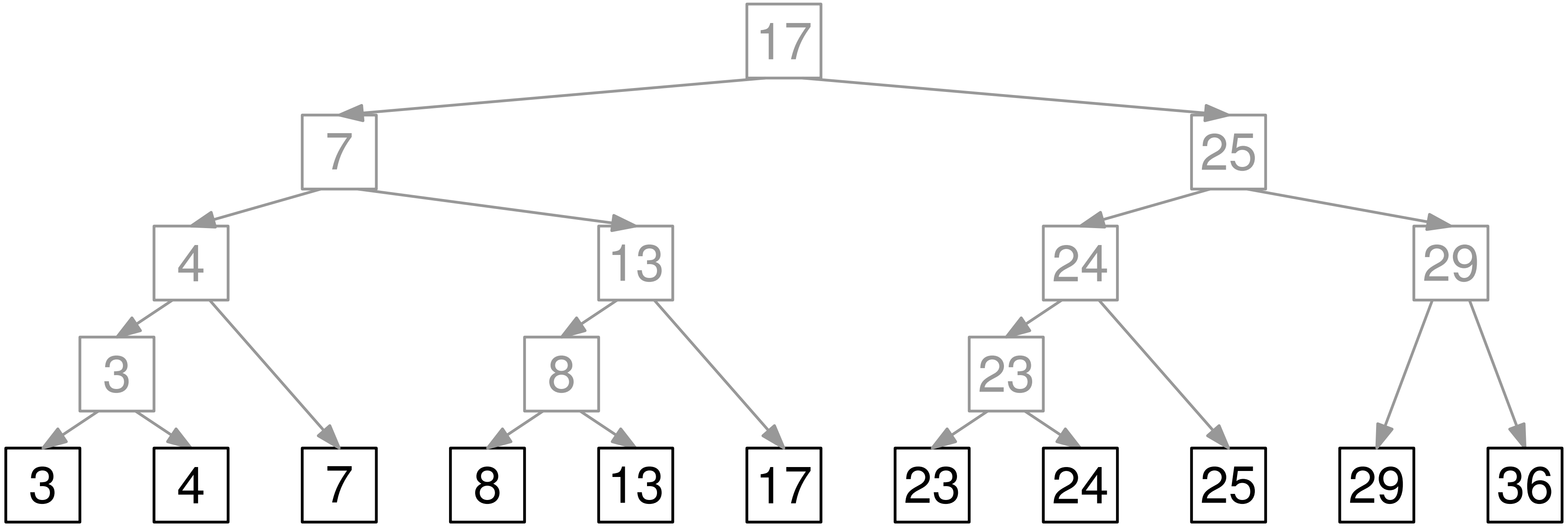
# $(2, 3)$-Bäume – Einfügen & Löschen

- Einfügen: Neues Blatt an Elter vom Nachfolger anhängen und danach (rekursiv) aufspalten wenn nötig  Beispiel: 16 Einfügen
- Löschen: Blatt einfach abschneiden und danach (rekursiv) verschmelzen oder ausbalancieren  Beispiel: 36 Löschen



# Rot-Schwarz-Bäume

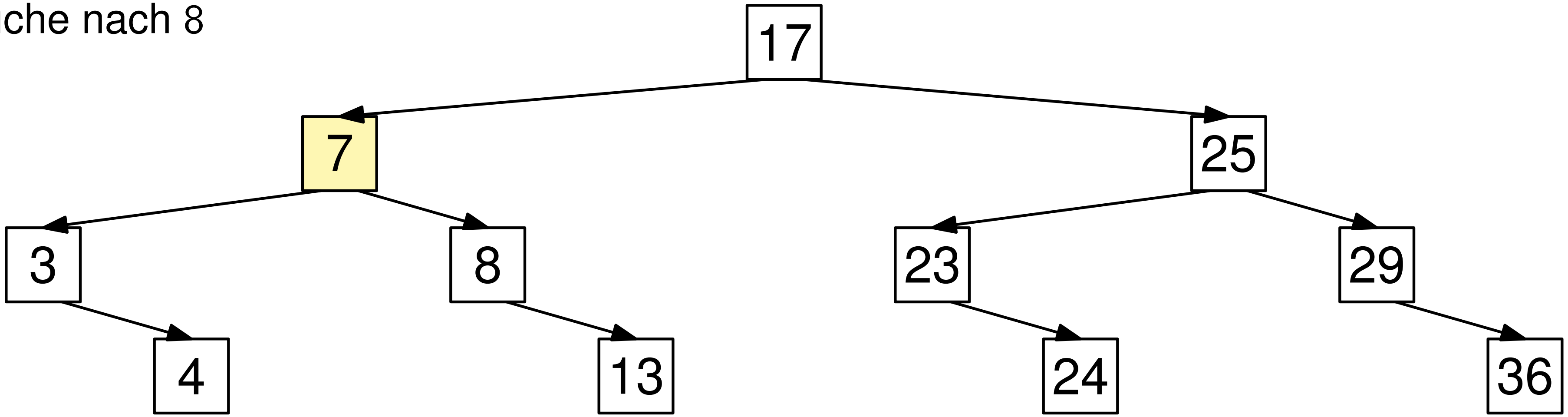
- Idee: Suchbaum, bei dem wir weniger Wert auf Balance legen, dafür bitte immer binär.
- Wichtiger Unterschied: Daten nicht nur in den Blättern



# Rot-Schwarz-Bäume

- Idee: Suchbaum, bei dem wir weniger Wert auf Balance legen, dafür bitte immer binär.
- Wichtiger Unterschied: Daten nicht nur in den Blättern

Beispiel: Suche nach 8

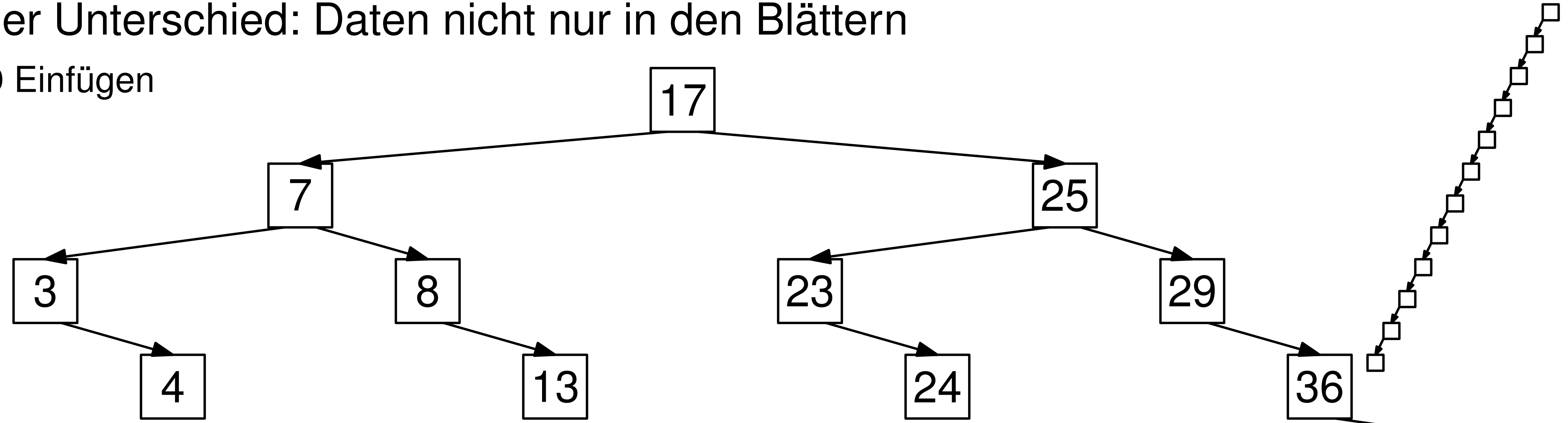


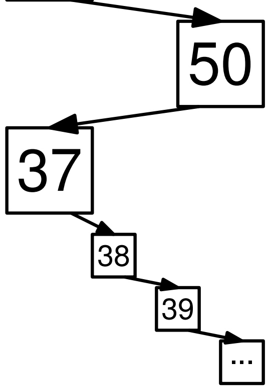

- Suche nach $x$: Vergleiche $x$ mit Schlüssel $k$
  - $x < k$: Abstieg in linken Teilbaum
  - $x = k$: Element gefunden
  - $x > k$: Abstieg in rechten Teilbaum

# Rot-Schwarz-Bäume

- Idee: Suchbaum, bei dem wir weniger Wert auf Balance legen, dafür bitte immer binär.
- Wichtiger Unterschied: Daten nicht nur in den Blättern

Beispiel: 39 Einfügen



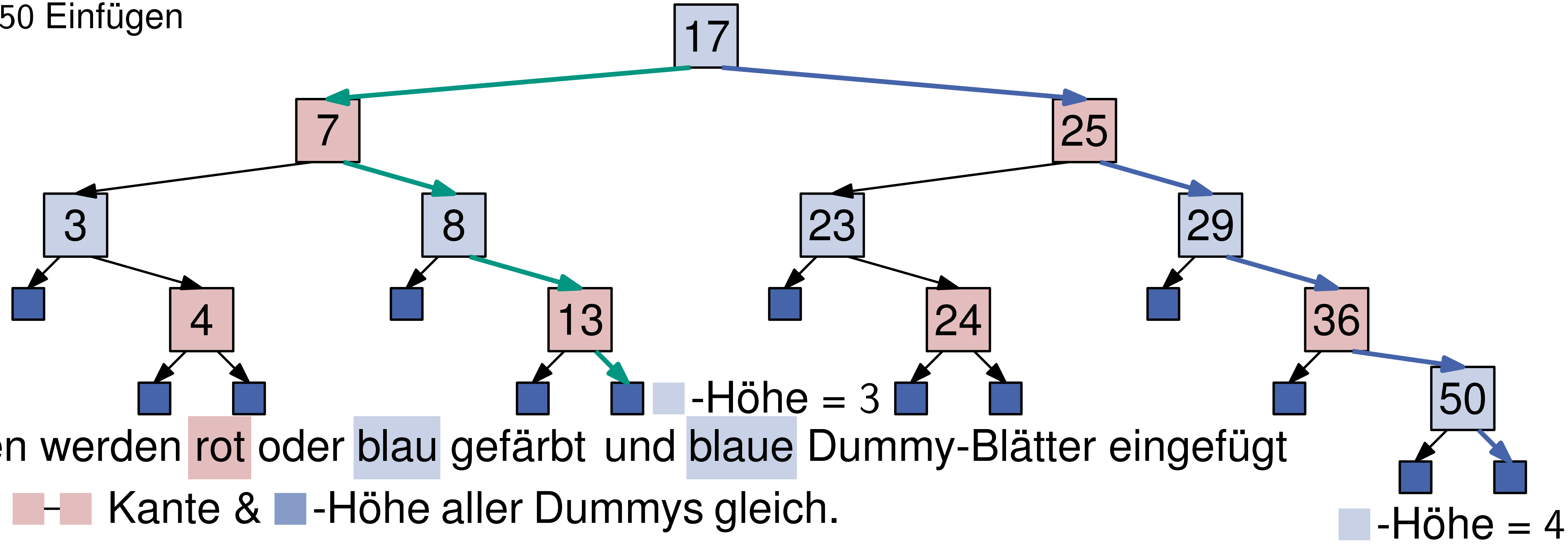

- Einfügen: Füge an richtigen noch nicht existierenden Abstieg ein
- Problem: Zu wenig Balance $\rightarrow$ Operationen nicht mehr in $O(\log(n))$
  - Baum darf nicht zu tief werden!

# Rot-Schwarz-Bäume

Karlsruher Institut für Technologie

- Idee: Suchbaum, bei dem wir nicht so viel wert auf Balance legen, dafür bitte immer binär.
- Kriterium: Das tiefste Blatt hängt höchstens **doppelt** so tief wie das flachste Blatt.

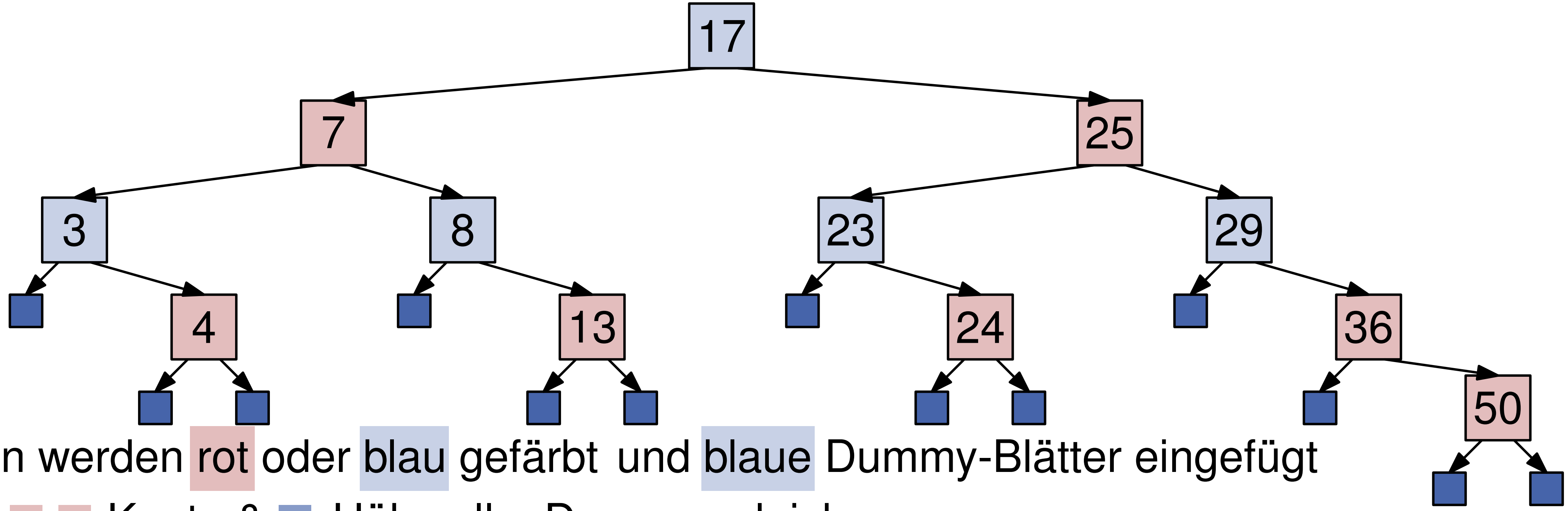
Beispiel: 50 Einfügen




- Knoten werden rot oder blau gefärbt und blaue Dummy-Blätter eingefügt
- Keine – Kante & -Höhe aller Dummys gleich.

# Rot-Schwarz-Bäume

- Idee: Suchbaum, bei dem wir nicht so viel wert auf Balance legen, dafür bitte immer binär.
- Kriterium: Das tiefste Blatt hängt höchstens **doppelt** so tief wie das flachste Blatt.



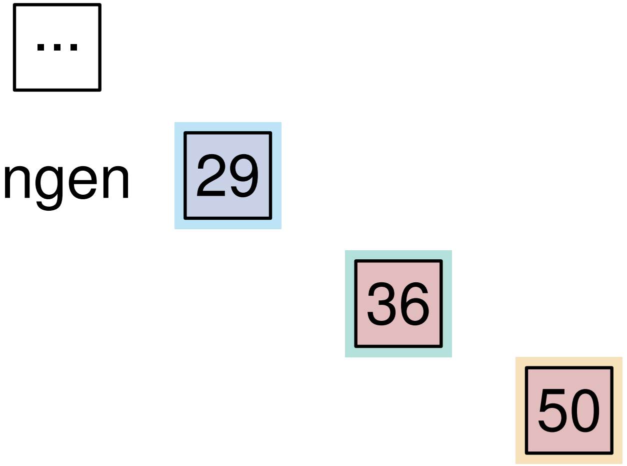

- Knoten werden rot oder blau gefärbt und blaue Dummy-Blätter eingefügt
- Keine – Kante & -Höhe aller Dummys gleich.
- Wir können 50 nicht einfügen ohne die Kriterien zu verletzen
- Es muss ausbalanciert werden!

# Rot-Schwarz-Bäume – Einfügen

keine – Kante
Blätter gleiche -Höhe
Suchbaumeigenschaft

- Idee: Neues Element als rotes Blatt an richtiger Stelle einfügen und gegebenenfalls ausbalancieren.
- Fall 1: Elter ist rot
  - dann ist Großelter blau (Warum?)
  - Fall 1a: Eltergeschwister ist blau

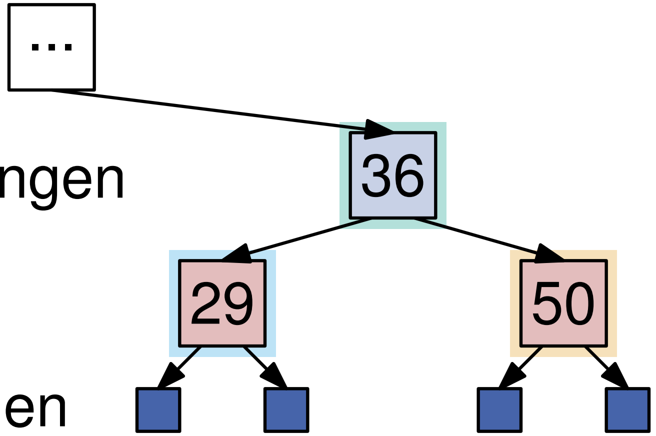
    Drei-Knoten-Umstrukturierung
    - Kind, Elter, und Großelter in die richtige Reihenfolge bringen

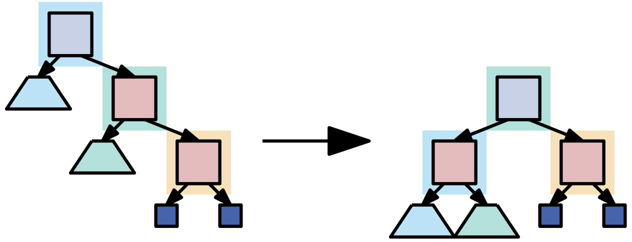


# Rot-Schwarz-Bäume – Einfügen

- Idee: Neues Element als rotes Blatt an richtiger Stelle einfügen und gegebenenfalls ausbalancieren.
- Fall 1: Elter ist rot
  - dann ist Großelter blau (Warum?)
  - Fall 1a: Eltergeschwister ist blau
    Drei-Knoten-Umstrukturierung
    - Kind, Elter, und Großelter in die richtige Reihenfolge bringen
    - Mittlerer Knoten wird Elter der anderen beiden
    - Mittlerer Knoten wird blau, die anderen rot
    - Teilbäume die unter den Knoten hingen korrekt umhängen

keine –  Kante
Blätter gleiche  -Höhe
Suchbaumeigenschaft



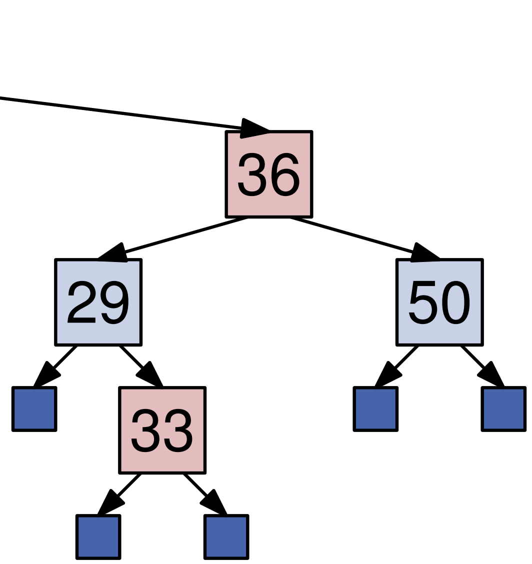



# Rot-Schwarz-Bäume – Einfügen

keine – Kante
Blätter gleiche -Höhe
Suchbaumeigenschaft

- Idee: Neues Element als rotes Blatt an richtiger Stelle einfügen und gegebenenfalls ausbalancieren.
- Fall 1: Elter ist rot
  - dann ist Großelter blau (Warum?)
  - Fall 1a: Eltergeschwister ist blau

    Drei-Knoten-Umstrukturierung
    - Kind, Elter, und Großelter in die richtige Reihenfolge bringen
    - Mittlerer Knoten wird Elter der anderen beiden
    - Mittlerer Knoten wird blau, die anderen rot
    - Teilbäume die unter den Knoten hingen korrekt umhängen
  - Fall 1b: Eltergeschwister ist rot
    - Umfärben: Elter und Eltergeschwister blau, Großelter rot



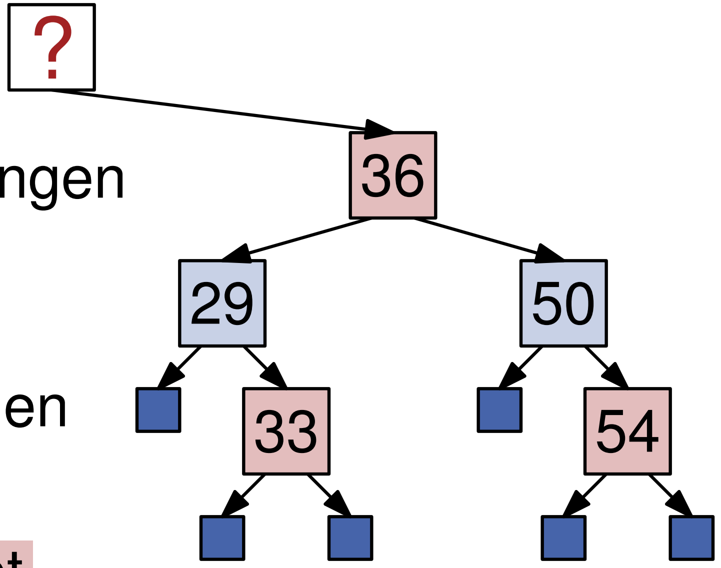
# Rot-Schwarz-Bäume – Einfügen

keine –  Kante
Blätter gleiche -Höhe
Suchbaumeigenschaft

- Idee: Neues Element als rotes Blatt an richtiger Stelle einfügen und gegebenenfalls ausbalancieren.
- Fall 1: Elter ist rot
  - dann ist Großelter blau (Warum?)
  - Fall 1a: Eltergeschwister ist blau

    Drei-Knoten-Umstrukturierung
    - Kind, Elter, und Großelter in die richtige Reihenfolge bringen
    - Mittlerer Knoten wird Elter der anderen beiden
    - Mittlerer Knoten wird blau, die anderen rot
    - Teilbäume die unter den Knoten hingen korrekt umhängen
  - Fall 1b: Eltergeschwister ist rot
    - Umfärben: Elter und Eltergeschwister blau, Großelter rot

    Sind wir jetzt fertig? -Höhe und Suchbaumeigenschaft erhalten, ggf. – Kante erzeugt! **Rekursion**
- Fall 2: Elter ist blau $\rightarrow$ fertig



# Rot-Schwarz-Bäume – Einfügen

- Diverse Fallunterscheidungen (Welche Eltergeschwister existieren? Wie werden zugehörige Teilbäume umgehangen? etc.)
- Laufzeit
  - Blatt an richtiger Stelle anhängen $O(\textit{Höhe des Baumes})$
  - Umstrukturieren **oder** Umfärben
    - Umstrukturieren (höchstens 1x) $O(1)$
    - Umfärben (höchstens $O(\textit{Höhe des Baumes})$) $O(\textit{Höhe des Baumes})$

$O(\textit{Höhe des Baumes})$

- Löschen auch in $O(\textit{Höhe des Baumes})$

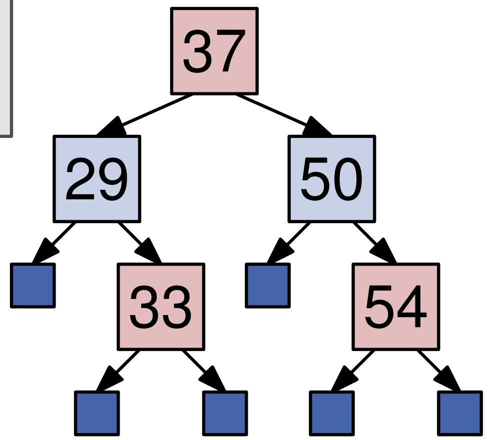
noch umständlicher

- Aber was ist denn nun die Höhe des Baumes?
- Hat das mit den Farben überhaupt funktioniert?

# Rot-Schwarz-Bäume – Höhe

- Beobachtung 1: Der Teilbaum eines Knotens in einem Rot-Schwarz-Baum ist wieder ein Rot-Schwarz-Baum.

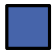
keine ■–■ Kante
Blätter gleiche ■-Höhe
Suchbaumeigenschaft

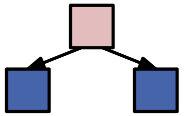



> **Lemma** *Rot-Schwarz-Baum – Anzahl Knoten*
> Sei $T$ ein Rot-Schwarz-Baum mit ■-Höhe $h$. Dann hat $T$ mindestens $n \geq 2^h - 1$ Knoten.

## Beweis über Induktion

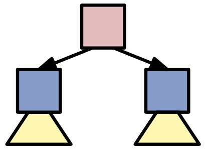
- Anfang: $h = 1$



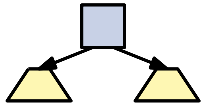


$n = 1 \geq 2^1 - 1 = 1$ $\quad$ $n = 3 \geq 2^1 - 1 = 1$

- Schritt: $h + 1$



Teilbäume mit ■-Höhe $h$



$$n \geq 2 \cdot (2^h - 1) + 2 + 1 = 2^{h+1} + 1$$

$$n \geq 2 \cdot (2^h - 1) + 1 = 2^{h+1} - 1$$

# Rot-Schwarz-Bäume – Höhe

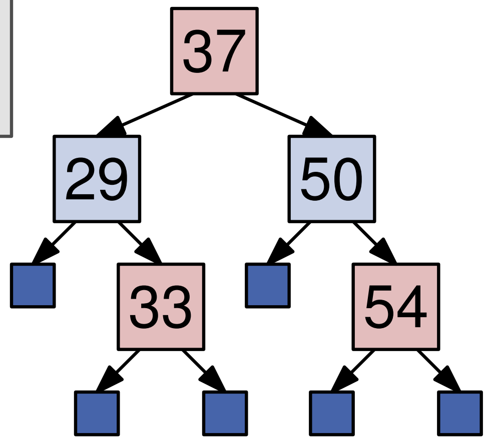
- Beobachtung 1: Der Teilbaum eines Knotens in einem Rot-Schwarz-Baum ist wieder ein Rot-Schwarz-Baum.

keine – Kante
Blätter gleiche -Höhe
Suchbaumeigenschaft

**Lemma** *Rot-Schwarz-Baum – Anzahl Knoten*
Sei $T$ ein Rot-Schwarz-Baum mit -Höhe $h$. Dann hat $T$ mindestens $n \geq 2^h - 1$ Knoten.

$2^h - 1 \leq n \rightarrow 2^h \leq n + 1 \rightarrow h \leq \log(n+1)$

- Beobachtung 2: Auf jedem Wurzel-Dummy-Pfad ist mindestens die Hälfte der Knoten blau. (Dummy ist blau & keine – Kante)
  - Für die Höhe $H$ des Baumes gilt: $H \leq 2 \cdot h \leq 2 \cdot \log(n+1)$



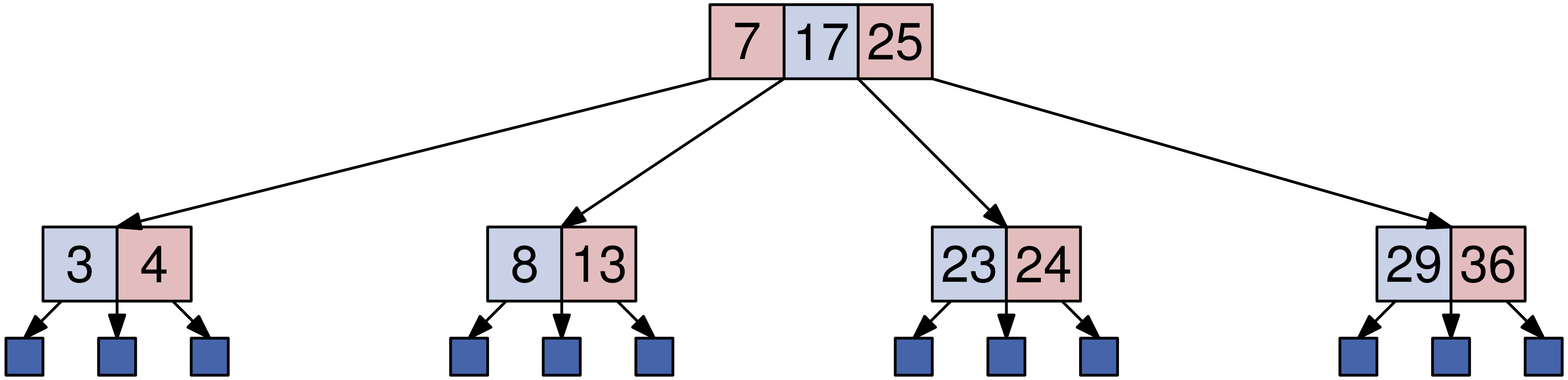

**Lemma** *Rot-Schwarz-Baum – Höhe*
Sei $T$ ein Rot-Schwarz-Baum mit $n$ Knoten. Dann hat $T$ eine Höhe von $H \leq 2\log(n+1)$.

# Suchbäume – Höhe

- Wir wünschen uns binäre und balancierte Suchbäume
- je eins der Kriterien kann etwas aufgeweicht werden
  - $(a, b)$-Bäume: **fast** binär, dafür perfekt balanciert
  - Rot-Schwarz-Bäume: binär, dafür **ausreichend** balanciert
- in beiden Fällen konnten wir logarithmische Höhe erreichen

Zufall?

- Ziehe im Rot-Schwarz-Baum die roten Knoten zu ihren Eltern



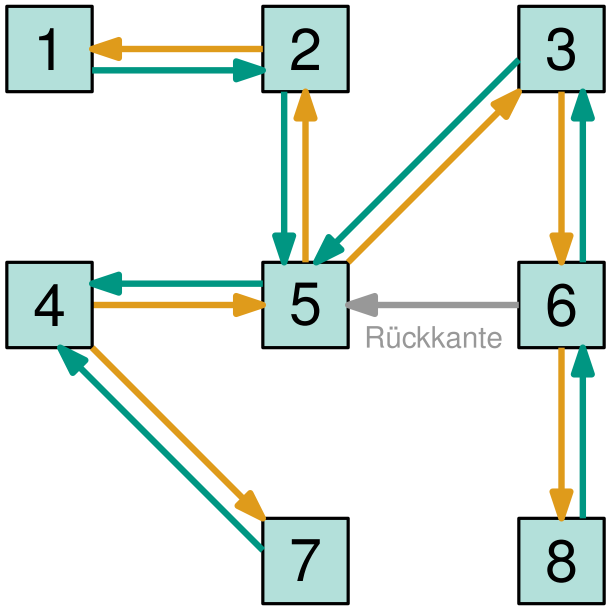

- Rot-Schwarz-Bäume haben quasi die gleiche Struktur wie $(2, 4)$-Bäume!

Aber irgendwie ist uns die Listenstruktur in den Blättern verloren gegangen...

# Tiefensuche

- Starte bei einem Knoten $s$
- Gehe in jedem Schritt zu einem ~~neuen~~ ungesehenen Nachbarn
- Sackgasse: (kein neuer Nachbar) $\rightarrow$ Backtracking: zurück zum Vorgänger
- Während einer Tiefensuche kann sich ein Knoten in 3 Zuständen befinden
  - ungesehen
  - gesehen
  - fertig

Beispiel: Startknoten 4

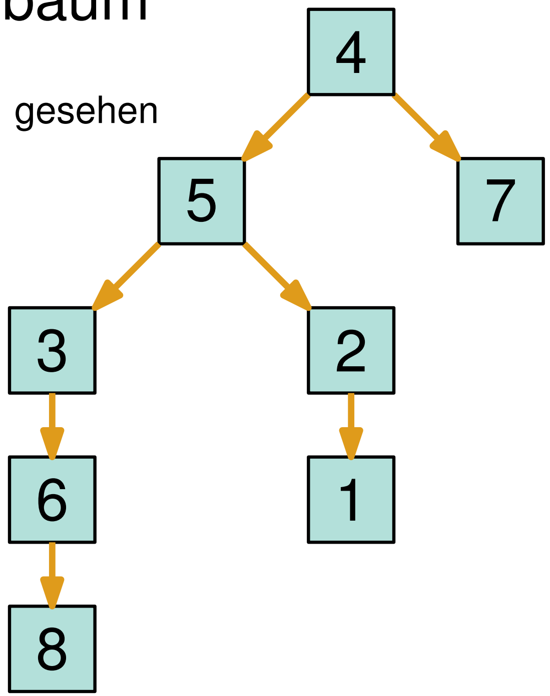



- Freiheitsgrad: Knotenreihenfolge

- Tiefensuchbaum

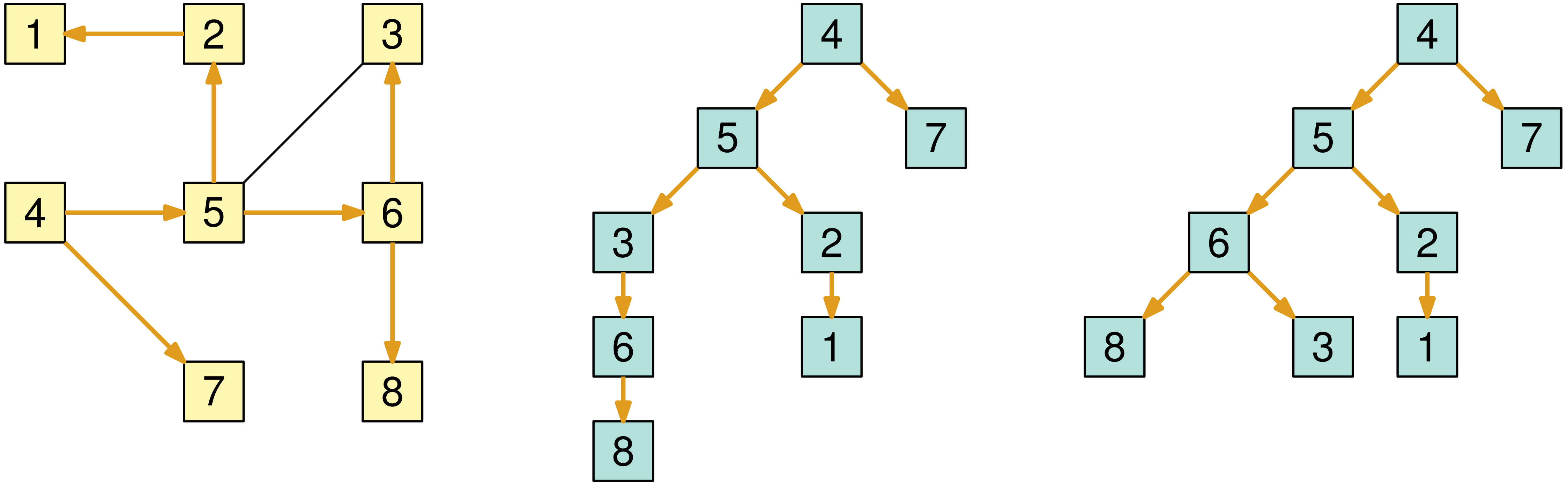
Warum?

- jeder Knoten 1x gesehen (außer Wurzel)
  $\rightarrow n - 1$ Kanten
- keine Rückkanten
  $\rightarrow$ keine Kreise



# Tiefensuche – Knotenreihenfolge

- Freiheitsgrad: Knotenreihenfolge

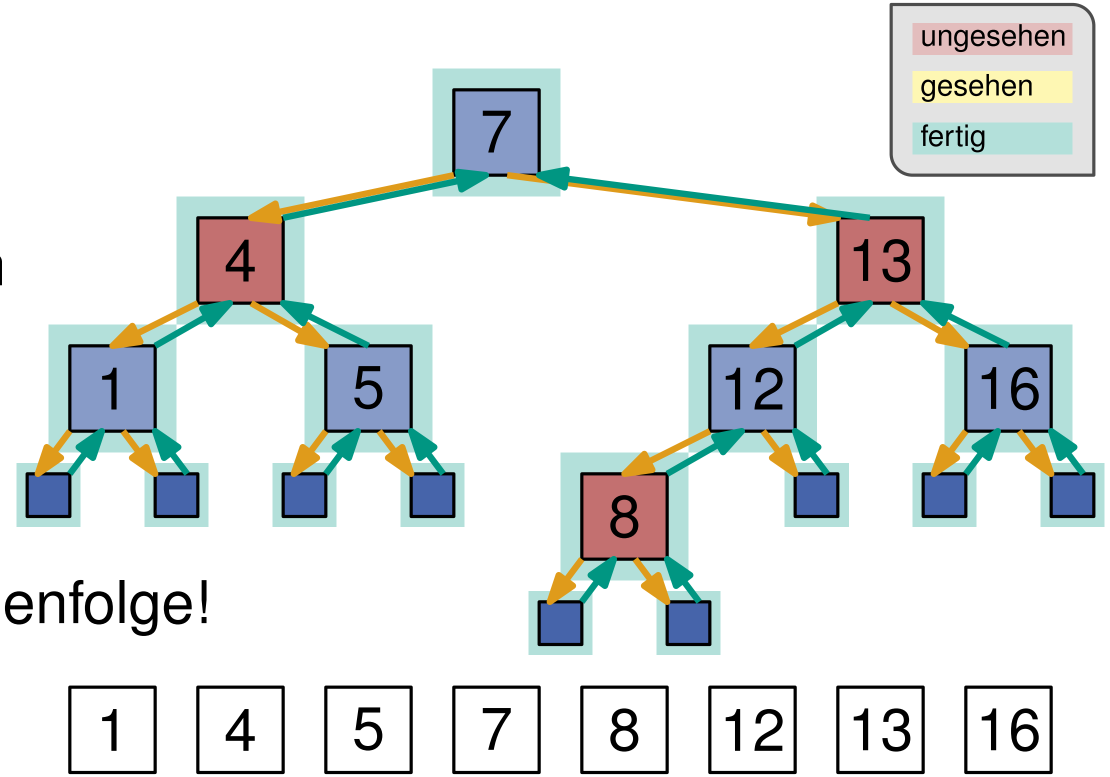



- Die Knotenreihenfolge bestimmt die Struktur des Tiefensuchbaums.
- Wir können uns die Reihenfolge zu Nutze machen...

# Rot-Schwarz-Bäume – Traversieren

Tiefensuche: **In-Order-Traversierung**

- Knotenreihenfolge: Kleineres Kind zuerst
- Startknoten: Wurzel
- Linker Teilbaum fertig? $\rightarrow$ Schlüssel ausgeben



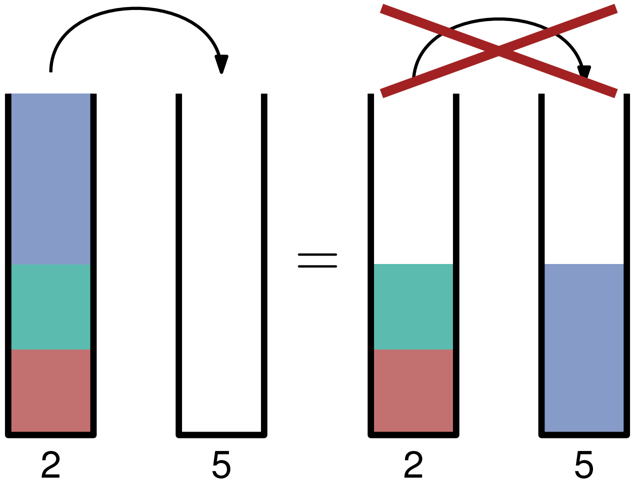

Wir erhalten die Schlüssel in aufsteigender Reihenfolge!

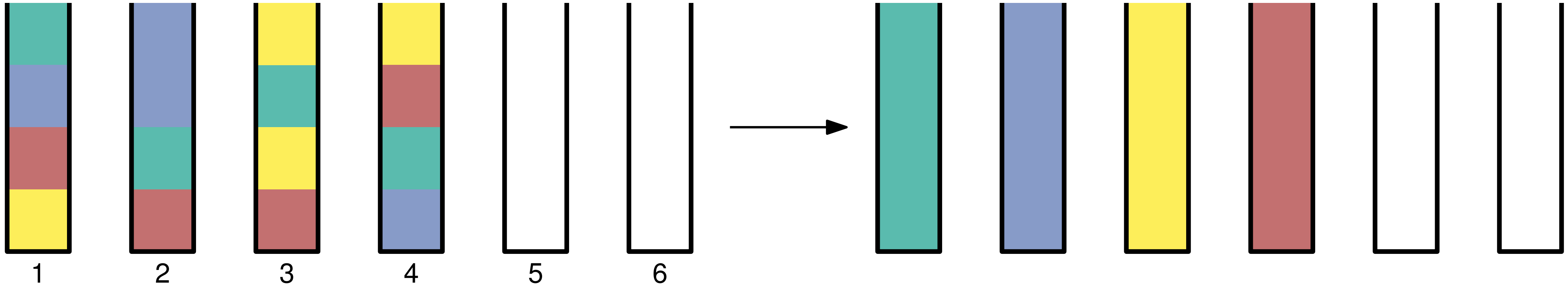
Laufzeit?

- Jede Kante wird genau zwei Mal abgelaufen
- Der Baum hat $n - 1$ Kanten
- Die sortierte Folge kann in Linearzeit aus dem Baum abgelesen werden.

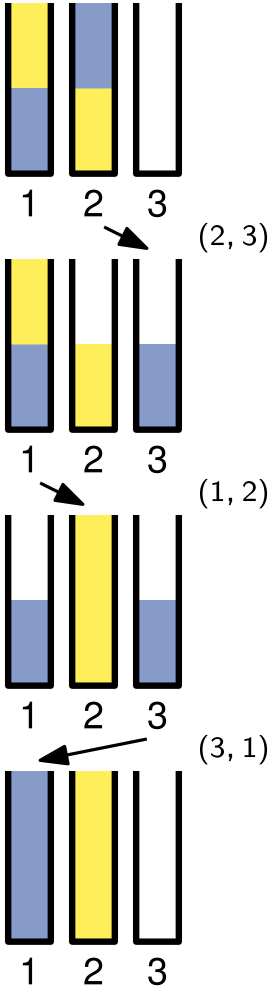
# Suche in Graphen – Ein Spiel

- wir haben eine Menge von Gläsern
- darin sind (alkoholfreie) Cocktails bestehend aus verschiedenen Säften
- **Ziel**: Entmischen (jeder Saft für sich in einem Glas)
  - Gesucht: Abfolge von erlaubten Umfüll-Operationen
- Regeln
  - ein Glas hält $\leq 4$ Einheiten, es gibt $4$ Einheiten pro Saft
  - ein Saft wird immer in Gänze umgefüllt
  - keinen Saft auf einen anderen kippen



# Cocktails – Beispiel & Modellierung

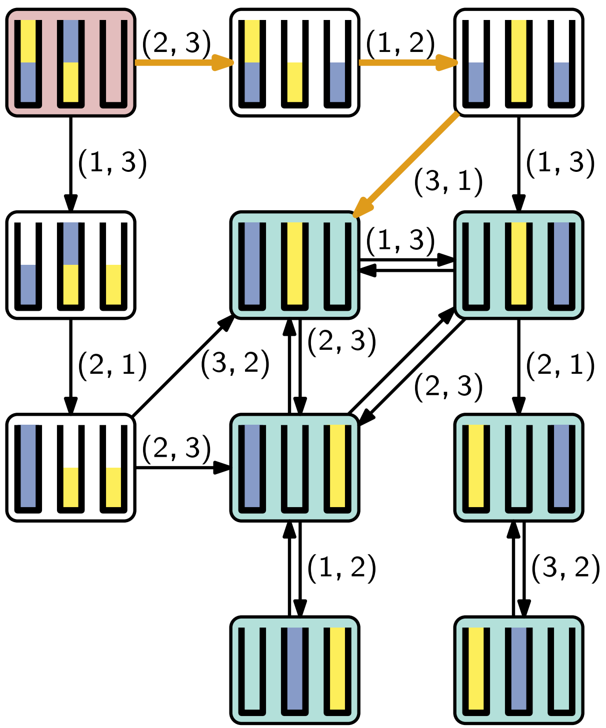



- Wie lösen wir dieses Problem algorithmisch?
  - Beginne mit Startkonfiguration
  - Führe irgendeine erlaubte Umfüll-Operation durch
  - ... wiederholt ...
  - ... bis entmischte Konfiguration erreicht
- Modellierung als Graph
  - eine Konfiguration ist ein Knoten
  - Kanten zwischen Konfigurationen die man mit erlaubten Operationen ineinander überführen kann
  - Eine Lösung ist ein Pfad von der Startkonfiguration zu einer entmischten Zielkonfiguration

*Wir wissen schon, wie man Wege in Graphen sucht!*

... aber es gibt da ein Problem...

Wie verwaltet man das sinnvoll?

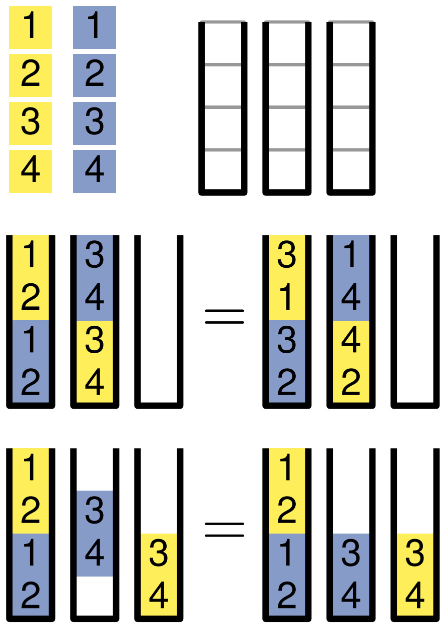


# Cocktails – Das Problem

- $k$ Säfte, $\ell$ Gläser
  - $4k$ Safteinheiten, $4\ell$ Glaseinheiten
  - $\binom{4\ell}{4k}$ Möglichkeiten Safteinheiten auf Gläser aufzuteilen
  - $k \cdot 4!$ Möglichkeiten Safteinheiten gleicher Säfte zu vertauschen
  - Schwerkraft: Pro Glas $2^4 - 5$ Füllungen mit Luftblasen

$$\approx \frac{\binom{4\ell}{4k}}{k \cdot 4! \cdot k \cdot (2^4 - 5)} \approx \frac{\ell^{4k}}{k! \cdot k^2} \text{ Konfigurationen}$$

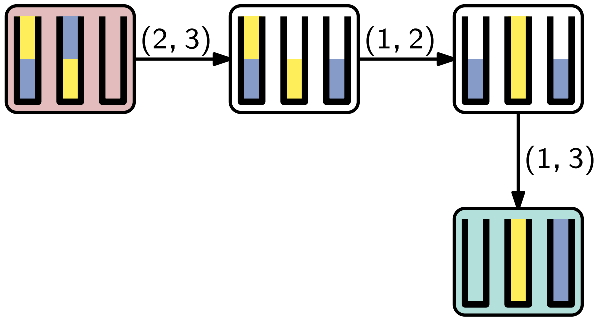
$$\binom{\ell}{k} \approx \frac{\ell^k}{k!}$$



- Die Anzahl möglicher Konfigurationen ist **exponentiell** in der Anzahl Säfte und die Basis ist durch die Anzahl Gläser bestimmt.
  - Anzahl Knoten im Graph viel zu hoch
  - Graph aufbauen benötigt zu viel Zeit / Speicher
  - Und dann müssen wir in dem Graphen auch noch Pfade finden...

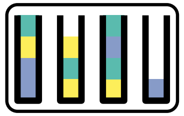
# Cocktails – Die Lösung

- Idee: Graph aufbauen während wir ihn explorieren
  - Startknoten erzeugen
  - eine erlaubte Umfüll-Operationen bestimmen und zugehörige Kante / Knoten in den Graph einfügen
  - Testen ob neuer Knoten eine Zielkonfiguration repräsentiert

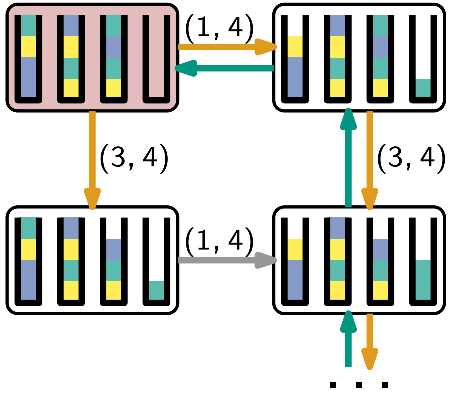



*Dieses Verfahren entspricht einer Tiefensuche im Konfigurationsgraphen!*

- Sackgasse: In einer Konfiguration kann nicht mehr umgefüllt werden



- Woher wissen wir, wann eine Kante die wir in den Graphen einfügen eine Rückkante ist?




- Verwalte Hashtabelle von bereits gesehenen Konfigurationen!

Breitensuche?

- Wie sieht eine Breitensuche in dem Graphen aus?
- Warum ist eine Breitensuche ggf. nicht so gut geeignet wie eine Tiefensuche?

# Zusammenfassung

**Suchbäume**

- Einfügen, Suchen, Löschen in logarithmischer Zeit
- Beschränkte Höhe des Suchbaums durch Ausgleich von Binarität und Balanciertheit
- $(2, 3)$-Bäume vs. Rot-Schwarz-Bäume

**Tiefensuche**

- Freiheitsgrad: Reihenfolge der Knoten
- In-Order-Traversierung zum Auslesen von Rot-Schwarz-Bäumen

**Modellierung als Graphproblem**

- Mittels Tiefensuche den Graph möglicher Konfigurationen Schritt für Schritt generieren und durchsuchen
- Das Funktioniert auch mit Konfigurationen anderer Spiele
  - Schach, Dame, Sudoku, ...
- Graphen als Framework für Spiele-Solver!