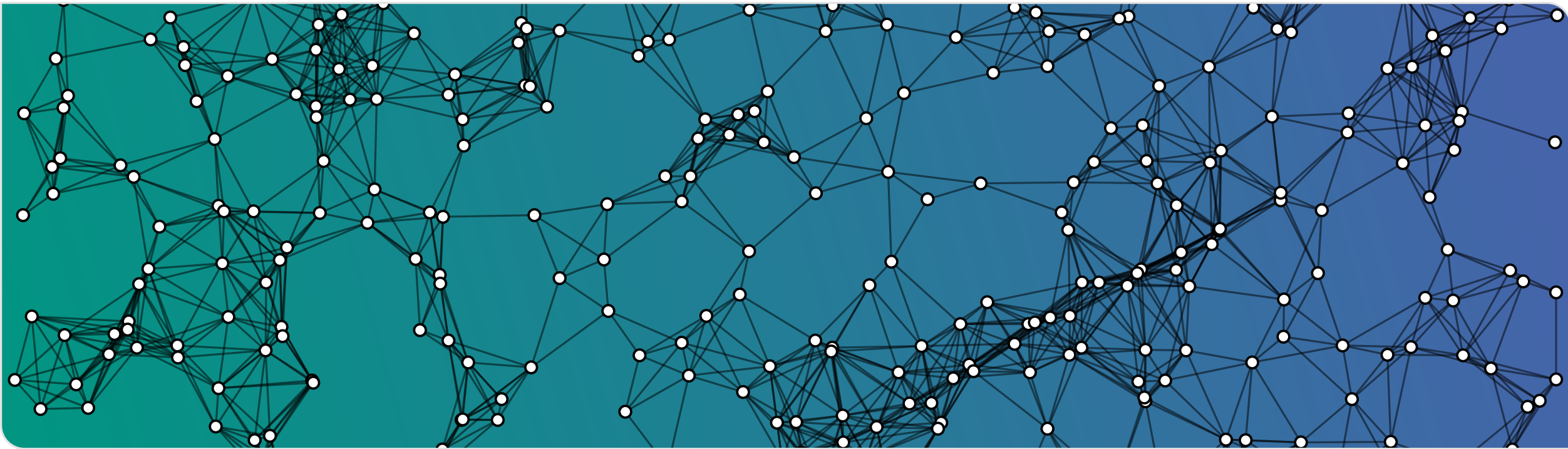


# Algorithmen 1

## Sortierte Folgen und Suchbäume





# Best-Fit Bin-Packing

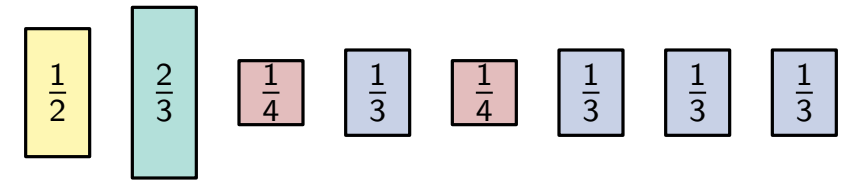
## Bin-Packing: Umzugskisten packen

- gegeben: Gegenstände unterschiedlicher Größe
- hier: Größe ist eine Zahl aus  $[0, 1]$
- in jede Kiste passen Gegenstände mit Gesamtgröße 1
- Ziel: packe alle Gegenstände in möglichst wenige Kisten

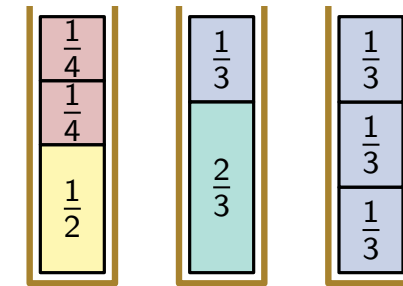
## Best-Fit Strategie

- pro Schritt: füge einen Gegenstand in eine Kiste ein
- Freiheitsgrad: Wahl der Kiste
- hier: möglichst volle Kiste, in die der Gegenstand passt
- Anmerkung: nicht optimal, aber brauchbare Approximation
- Ziel heute:  $O(\log n)$  pro Schritt

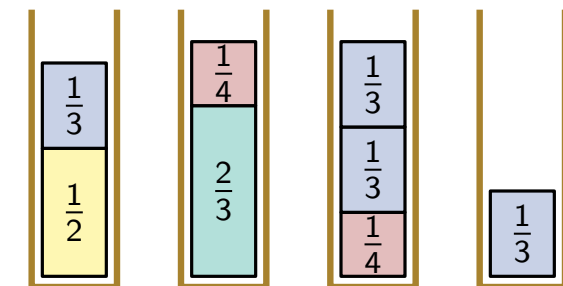
Warum eignen sich unsere bisherigen Datenstrukturen nicht?



## Optimale Lösung

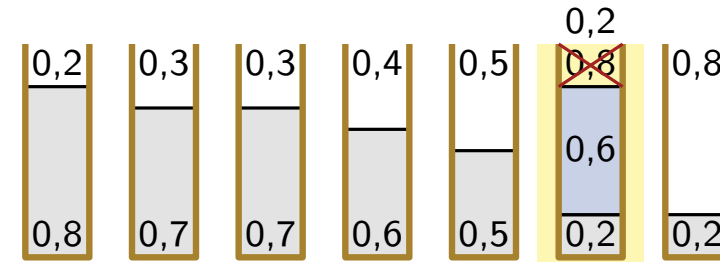


## Best-Fit Lösung

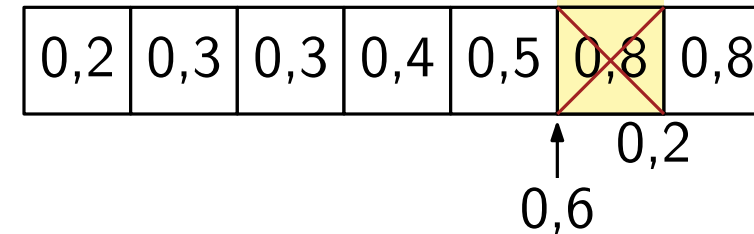


# Versuch 1: sortiertes Array

Füllgrad der Bins:



Array sortiert nach freier Kapazität:



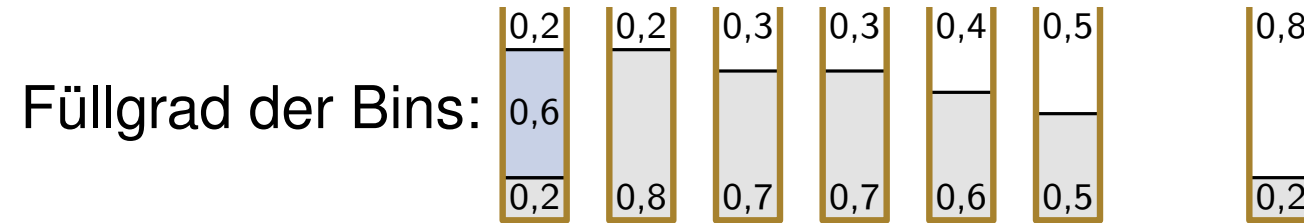
## Super: richtigen Bin finden

- Ziel: Gegenstand der Größe  $x$  einfügen
- binäre Suche nach  $x \rightarrow$  Bin mit kleinster freier Kapazität  $\geq x$
- Kosten:  $O(\log n)$

## Problem: Array sortiert halten

- Zielposition des Bins kann wieder leicht gefunden werden (binäre Suche)
- Element verschieben: ggf. müssen  $\Theta(n)$  Elemente angefasst werden

# Versuch 2: sortierte Liste



## Super: Einträge umhängen geht schnell

- nur ein paar Zeiger umhängen

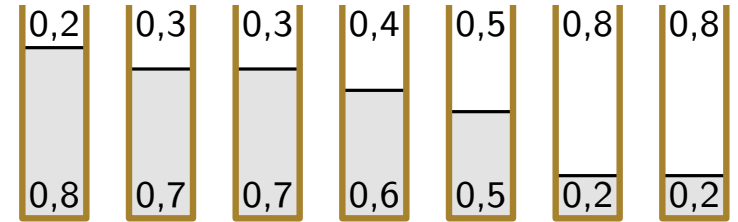
## Problem: Richtiges Bin finden

- keine binäre Suche, da kein wahlfreier Zugriff
- kostet ggf.  $\Theta(n)$

# Versuch 3: Hashing oder Heaps

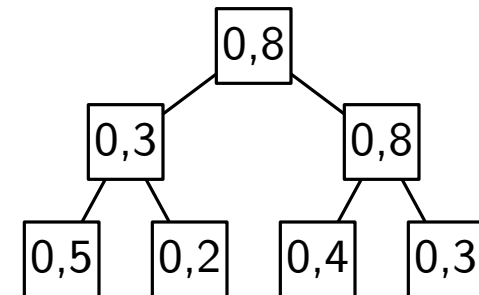
## Hashing

- kann nur exakte Anfragen
- wir können schnell rausfinden, ob es einen Bin gibt, wo der neue Gegenstand genau rein passt
- nächstkleinere freie Kapazität finden geht nicht



## Heaps

- Maximale freie Kapazität kann schnell gefunden werden
- darüber hinaus können wir nicht effizient suchen



# Ziel: dynamische sortierte Folgen

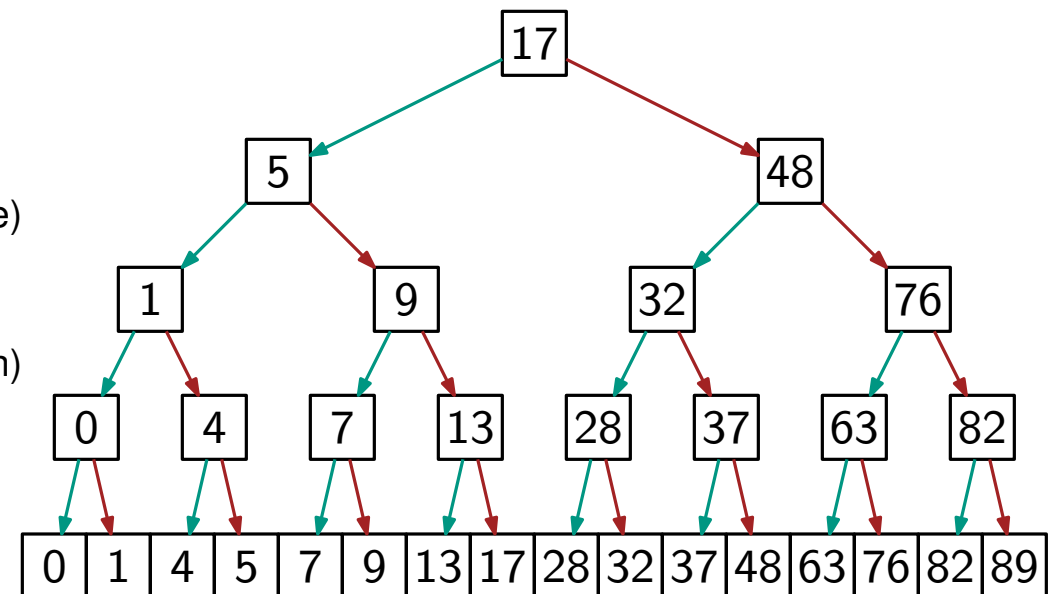
## Ziel: kombiniere Vorteile von Listen (Modifizierbarkeit) und Arrays (binäre Suche)

- Problem der Liste: kein wahlfreier Zugriff auf mittleres Element
- Beobachtung: wir fragen im Entscheidungsbaum immer die selben Elemente an
- wir brauchen nicht wirklich wahlfreien Zugriff
- wenn wir den Entscheidungsbaum speichern

## Plan: Suchbaum

- speichere sortierte Liste + Baum
- Suche:  $O(\log n)$  dank Baum (simuliert eine binäre Suche)
- Einfügen und Löschen:
  - in der Liste selbst schnell ( $O(1)$  nach dem Suchen)
  - Hoffnung: Baum kann man schnell updaten

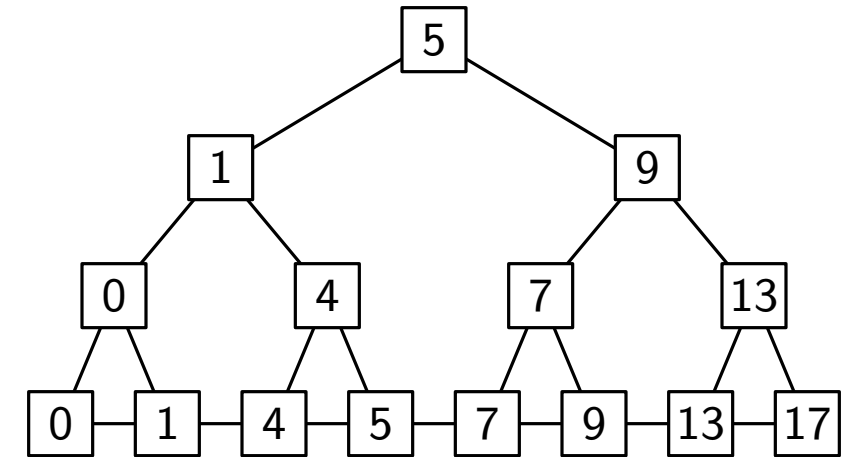
binäre Suche nach  $x = 25$   
 (bzw. nach nächst-größerem Eintrag, falls  $x$  nicht existiert)



# Suchbaum: Vorüberlegungen zum Update

## Binärer Suchbaum – Eigenschaften

- **binär:** jeder innere Knoten hat genau zwei Kinder
- **balanciert:** jedes Blatt hat gleiche Tiefe (Distanz zur Wurzel)
- **Suchbaum:** für Knoten mit Schlüssel  $k$  gilt:
  - Knoten in linkem Teilbaum:  $\leq k$
  - Knoten in rechtem Teilbaum:  $> k$



## Suchbaum-Eigenschaft

- essentiell, da Suche sonst nicht funktioniert → müssen wir beim Update sicherstellen

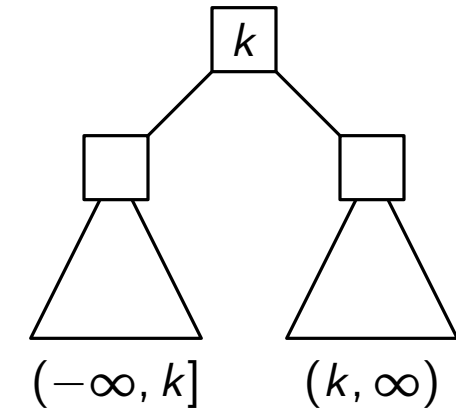
## Binär und balanciert

- sorgt für logarithmische Höhe → schnelle Suche
- etwas Unordnung ist ok: wir brauchen nicht beides exakt
- binär und halbwegs balanciert: rot-schwarz Baum
- halbwegs binär und perfekt balanciert:  $(a, b)$ -Baum → heute

# Nicht-Binär: Suchbaum mit höherem Verzweigungsgrad

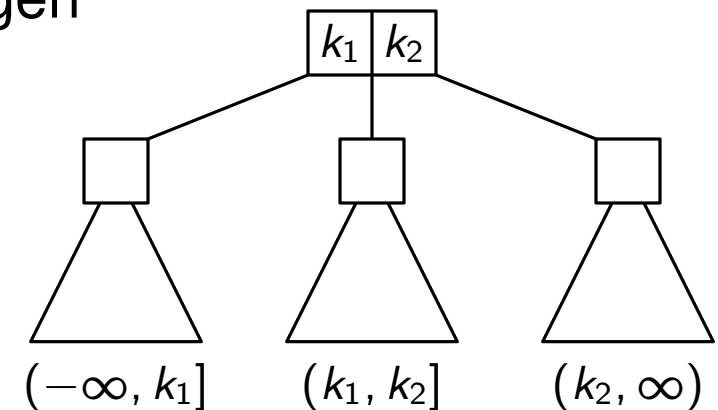
## Suche nach $x$ in binärem Suchbaum

- pro Knoten: in linken oder rechten Teilbaum absteigen
- Knoten speichert einen Schlüssel  $k$
- Fall 1:  $x \leq k \rightarrow$  linker Teilbaum
- Fall 2:  $k < x \rightarrow$  rechter Teilbaum



## Suche nach $x$ in Suchbaum mit 3 Kindern pro Knoten

- pro Knoten: in linken, mittleren oder rechten Teilbaum absteigen
- Knoten speichert zwei Schlüssel  $k_1$  und  $k_2$
- Fall 1:  $x \leq k_1 \rightarrow$  linker Teilbaum
- Fall 2:  $k_1 < x \leq k_2 \rightarrow$  mittlerer Teilbaum
- Fall 3:  $k_2 < x \rightarrow$  rechter Teilbaum



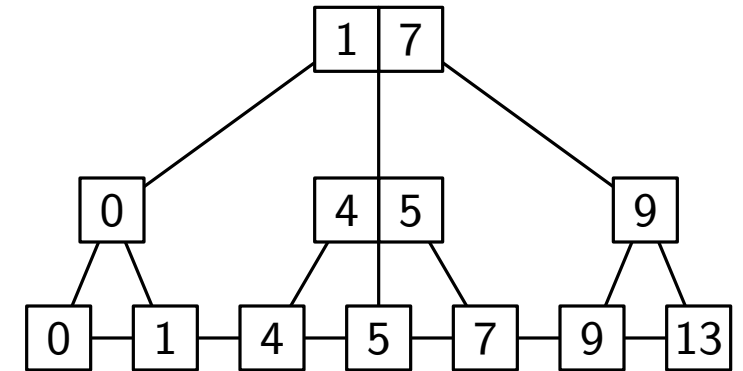
**noch mehr Kinder:** analog



# (2, 3)-Baum

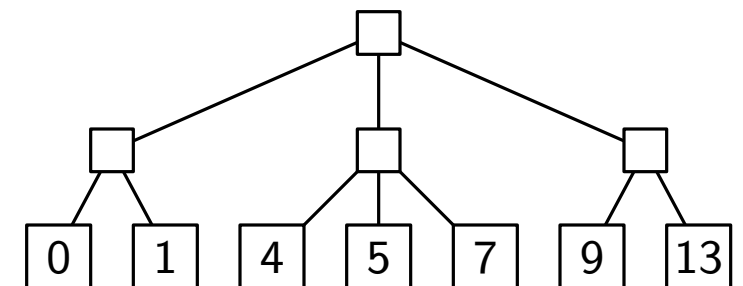
## Gewünschte Eigenschaften

- **fast binär:** jeder innere Knoten hat 2 oder 3 Kinder
- **balanciert:** jedes Blatt hat gleiche Tiefe (Distanz zur Wurzel)



## Beobachtungen & Anmerkungen

- Höhe ist logarithmisch:
  - die Anzahl Knoten pro Lage wachsen exponentiell (mindestens Faktor 2)
  - daher: nur logarithmisch viele Lagen
- Möglichkeit für 2 oder 3 Kinder gibt Flexibilität beim Einfügen und Löschen
- im Folgenden: vereinfachte Darstellung
  - Fokus auf die Baumstruktur
  - Schlüssel in den Knoten erstmal ignorieren  
(wenn die Baumstruktur passt, dann bekommt man die Schlüssel sicher auch noch hin)



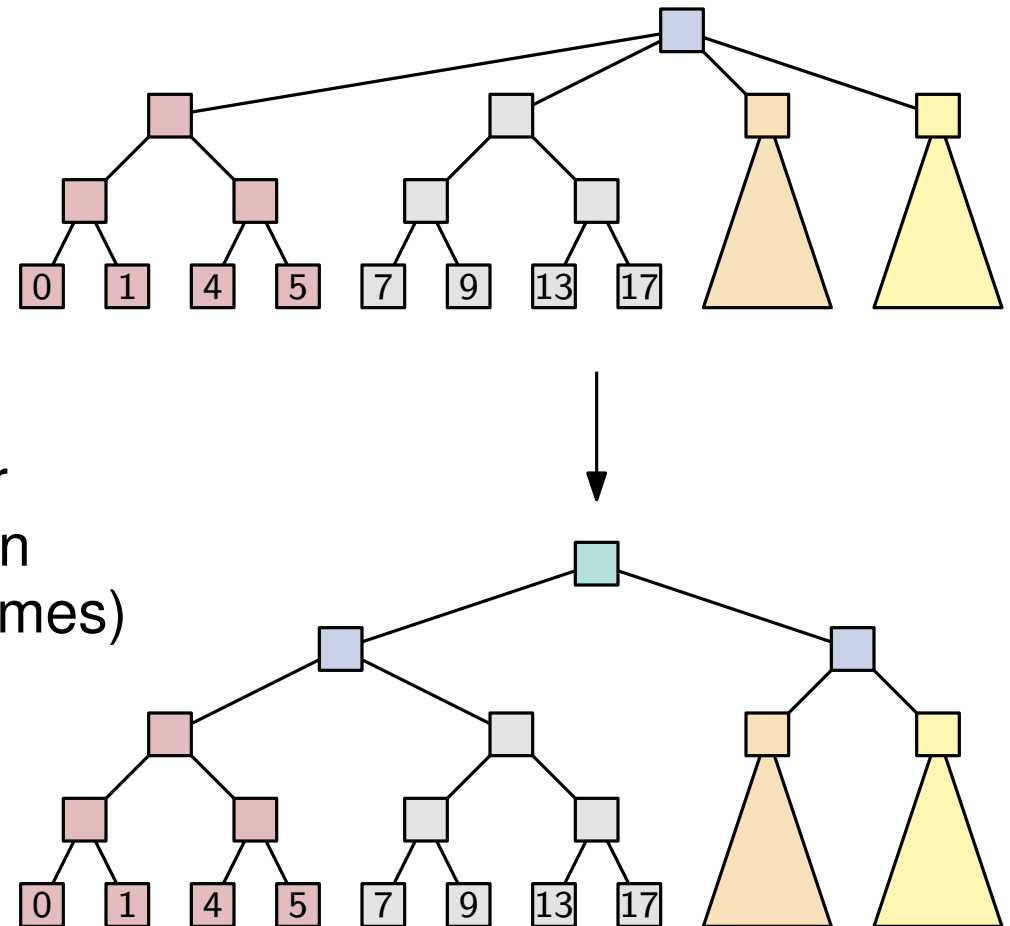
## (2, 3)-Baum: Einfügen am Beispiel

### Plan

- richtige Stelle zum Einfügen finden  
→ Elter  $b$  des Nachfolgers
- füge neues Blatt einfach als Kind von  $b$  ein
- Fall 1:  $b$  hat danach 3 Kinder → alles gut
- Fall 2:  $b$  hat danach 4 Kinder → aufspalten
  - erzeugt 2 Knoten mit je 2 Kindern
  - Elter  $a$  von  $b$  hat dann ein Kind mehr als vorher
  - verschiebt Problem Schritt für Schritt nach oben
  - kann nur  $O(\log n)$  oft passieren (Höhe des Baumes)

### Anmerkung

- Aufspaltung der Wurzel → neue Wurzel



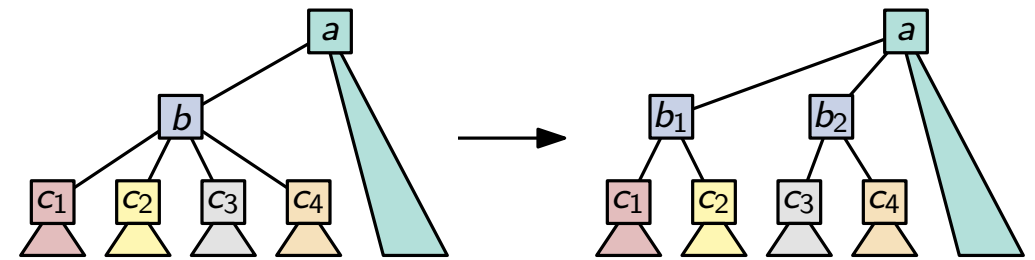
# (2, 3)-Baum: Einfügen

## Neues Blatt einfügen

- Schritt 1: erstmal einfach neues Blatt an der richtigen Stelle einfügen
- Schritt 2: Knoten mit 4 Kindern aufspalten

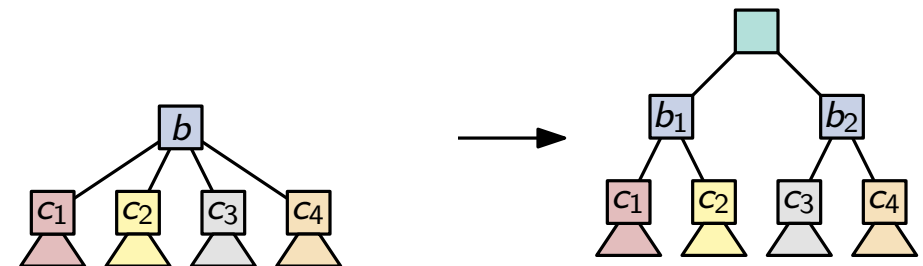
## Aufspalten

- ersetze  $b$  durch  $b_1$  mit Kindern  $c_1$  und  $c_2$ , sowie  $b_2$  mit Kindern  $c_3$  und  $c_4$
- $a$  hat ein Kind mehr  $\rightarrow$  ggf. rekursiv aufspalten



## Aufspalten (Wurzel)

- wie sonst: aufspalten von  $b$  in  $b_1$  und  $b_2$
- neue Wurzel mit Kindern  $b_1$  und  $b_2$



## Invarianten

- jedes Blatt hat die gleiche Tiefe
- höchstens ein Knoten hat nicht 2 oder 3 Kinder

(und dieser wandert mit jeder Aufspaltung nach oben)

# (2, 3)-Baum: Löschen

## Existierendes Blatt entfernen

- Schritt 1: erstmal entsprechendes Blatt einfach löschen
- Schritt 2: Knoten mit nur einem Kind aufräumen

## Geschwister mit 2 Kindern: Verschmelzen

- verschmelze mit Geschwisterknoten
- Elter hat ein Kind weniger → ggf. rekursiv aufräumen

## Geschwister mit 3 Kindern: Ausbalancieren

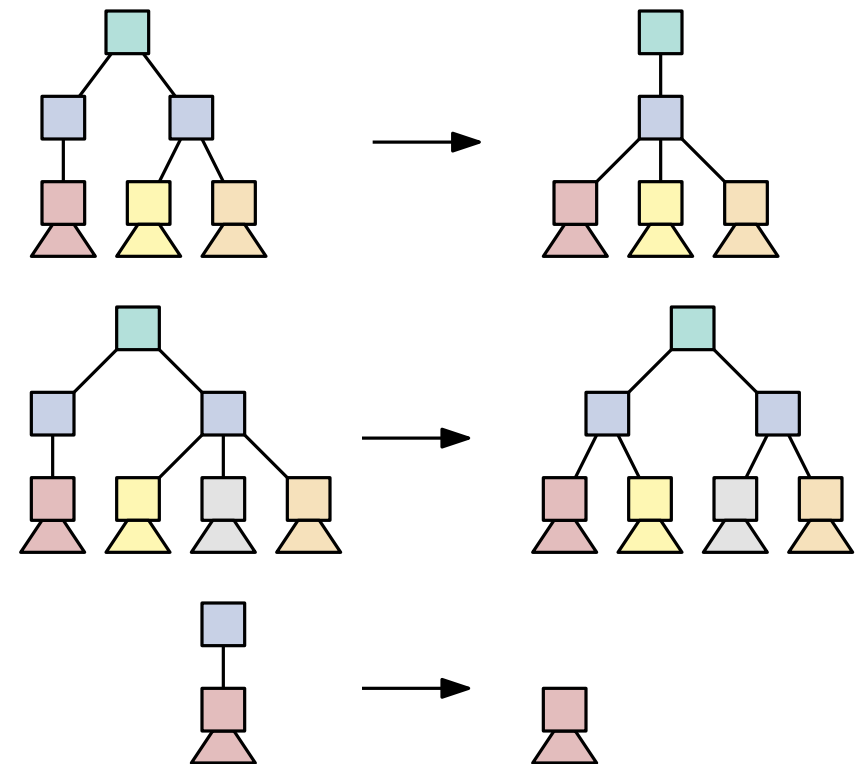
- verschiebe eins der Kinder (inklusive Teilbaum)

## Wurzel hat nur 1 Kind

- lösche die Wurzel (Höhe des Baums sinkt um 1)

## Invarianten

- jedes Blatt hat die gleiche Tiefe
- höchstens ein Knoten hat nicht 2 oder 3 Kinder

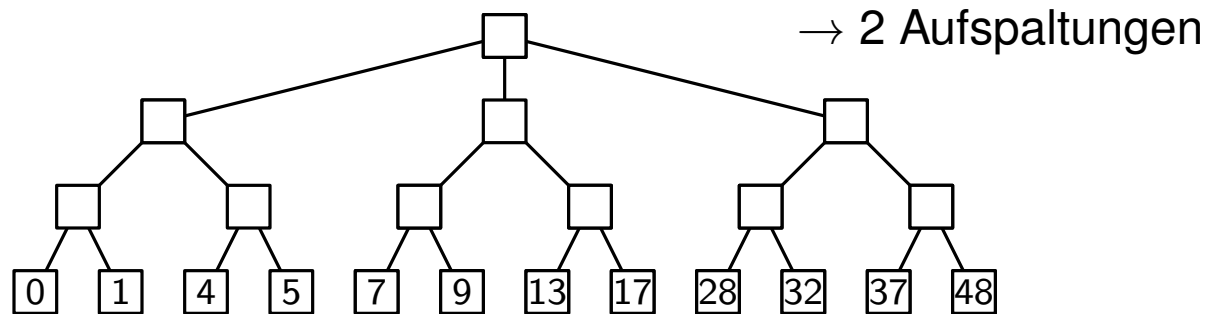


(und dieser wandert mit jeder Aufspaltung nach oben)

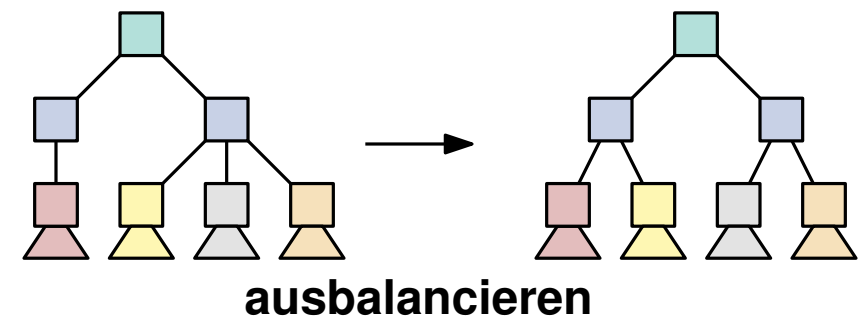
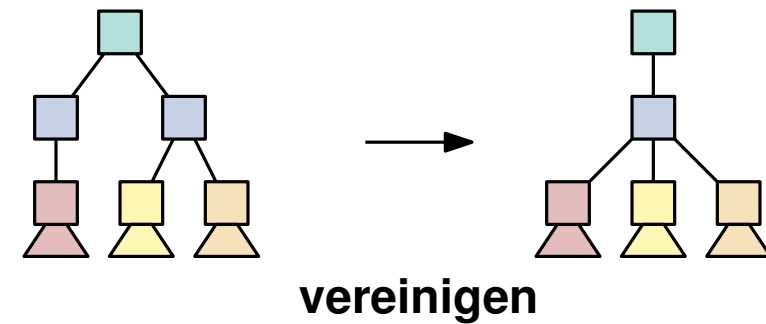
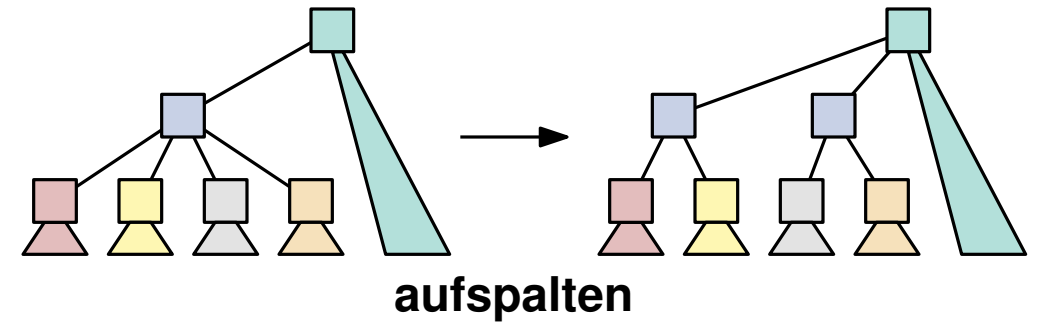
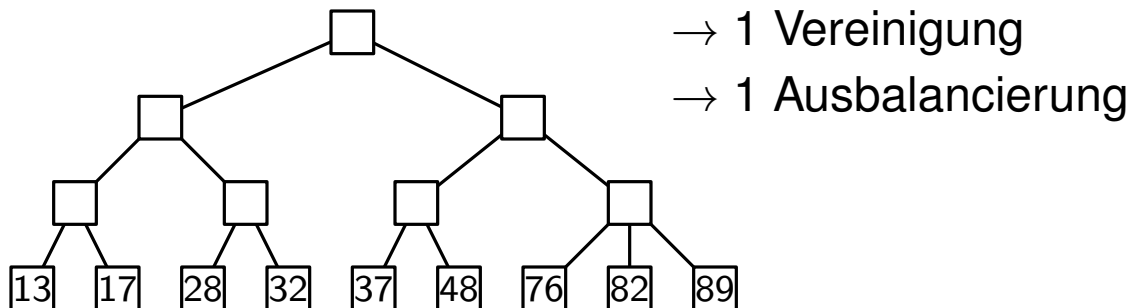


# Wie viel müssen wir Arbeiten?

Wie oft müssen wir aufspalten, wenn wir 1 einfügen?



Wie oft müssen wir vereinigen, wenn wir 63 löschen?  
Wie oft ausbalancieren?



# (2, 3)-Baum: Überblick

## Lemma

Ein (2, 3)-Baum hat logarithmische Tiefe.

## Erinnerung: (2, 3)-Baum

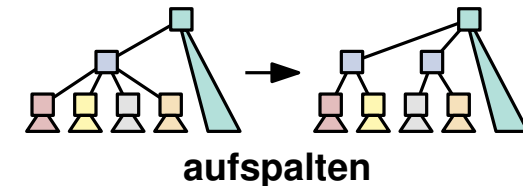
- **fast binär:** jeder innere Knoten hat 2 oder 3 Kinder
- **balanciert:** jedes Blatt hat gleiche Tiefe

## Lemma

Sei  $T$  ein Baum der die (2, 3)-Baum Eigenschaften erfüllt, abgesehen für einen Knoten mit 4 Kindern.  $T$  kann mit  $O(\log n)$  Aufspaltungen in einen (2, 3)-Baum überführt werden.

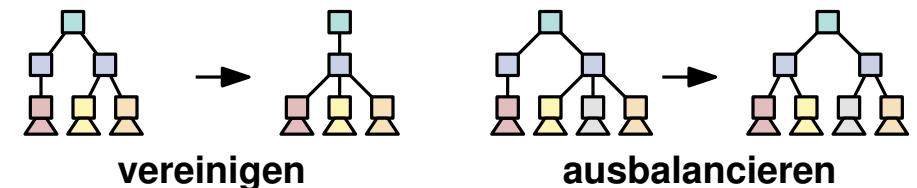
## Folgerung

Wir können in  $O(\log n)$  in einen (2, 3)-Baum einfügen.



## Analoge Folgerung

Wir können in  $O(\log n)$  in einem (2, 3)-Baum löschen.



**Achtung:** Wir müssen uns eigentlich noch um die Schlüssel in den Knoten kümmern!

# Suchbäume in der Wildnis

## Java

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeMap.html>

### Class `TreeMap<K,V>`

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the `natural ordering` of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed  $\log(n)$  time cost for the `containsKey`, `get`, `put` and `remove` operations.

#### Method Summary

Modifier and Type	Method	Description
<code>K</code>	<code>ceilingKey(K key)</code>	Returns the least key greater than or equal to the given key, or null if there is no such key.
<code>K</code>	<code>floorKey(K key)</code>	Returns the greatest key less than or equal to the given key, or null if there is no such key.
<code>V</code>	<code>get(Object key)</code>	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
<code>V</code>	<code>put(K key, V value)</code>	Associates the specified value with the specified key in this map.
<code>V</code>	<code>remove(Object key)</code>	Removes the mapping for this key from this <code>TreeMap</code> if present.

# Überblick Datenstrukturen

## Unbeschränkte Arrays

<b>pushBack</b> ( $x$ )	Element $x$ hinten einfügen
<b>popBack</b> ()	letztes Element löschen
<b>at</b> ( $i$ )	Zugriff auf Element mit Index $i$
<b>find</b> ( $x$ )	binäre Suche nach $x$ , wenn sortiert
<b>C++:</b> vector	<b>Java:</b> ArrayList

## Prioritätswarteschlange (Heap)

<b>push</b> ( $x, p$ )	Element $x$ mit Priorität $p$ einfügen
<b>popMin</b> ()	Element mit minimaler Priorität extrahieren
<b>decPrio</b> ( $a, p$ )	Priorität von Knoten $a$ auf $p$ verkleinern
<b>C++:</b> priority_queue	<b>Java:</b> PriorityQueue
(Prioritätsänderungen muss man sich ggf. selbst bauen: Lazy Evaluation)	

## Listen

<b>insertAfter</b> ( $a, x$ )	Element $x$ nach Knoten $a$ einfügen
<b>remove</b> ( $a$ )	Knoten $a$ löschen
<b>splice</b> ( $a, b, c$ )	Teillisten $\langle a, \dots, b \rangle$ hinter $c$ einfügen
<b>C++:</b> list	<b>Java:</b> LinkedList

## Sortierte Folge (Suchbaum)

<b>set</b> ( $k, v$ )	setze Wert für Schlüssel $k$ auf $v$
<b>find</b> ( $k$ )	Suche nach Schlüssel $k$
<b>remove</b> ( $k$ )	Eintrag mit Schlüssel $k$ löschen
<b>C++:</b> map	<b>Java:</b> TreeMap

## Hashtabelle

<b>set</b> ( $k, v$ )	setze Wert für Schlüssel $k$ auf $v$
<b>get</b> ( $k$ )	Zugriff auf Wert von Schlüssel $k$
<b>remove</b> ( $k$ )	Eintrag für Schlüssel $k$ löschen
<b>C++:</b> unordered_map	<b>Java:</b> HashMap

## Anmerkung

- hier sind jeweils nur die wichtigsten Operationen aufgeführt (die Datenstrukturen können meist noch mehr)
- Faustregel: wähle einfachste DS, die benötigte Operationen effizient kann



# Zusammenfassung

## Die eierlegende Wollmilchsau

- Suchbäume können im Prinzip alles, was das Herz begehrt
- einfügen, suchen, löschen, Min/Max extrahieren, Bereichsanfragen, jeweils in  $O(\log n)$

## Allgemeiner: $(a, b)$ -Bäume

- Knoten haben mindestens  $a$  und höchstens  $b$  Kinder
- funktioniert analog, wenn  $2 \leq a \leq (b + 1)/2$

## Lernziel

- wissen, wie Suchbäume funktionieren
- können für eine Anwendung die passende Datenstruktur wählen
- Fähigkeit zur Ausarbeitung der Details: Wissen über die Funktionsweise (Kenntnisstand heute; hohe Abstraktionsebene) → Implementierung (Pseudocode)  
(Pseudocode auswendig können ist nicht hilfreich)

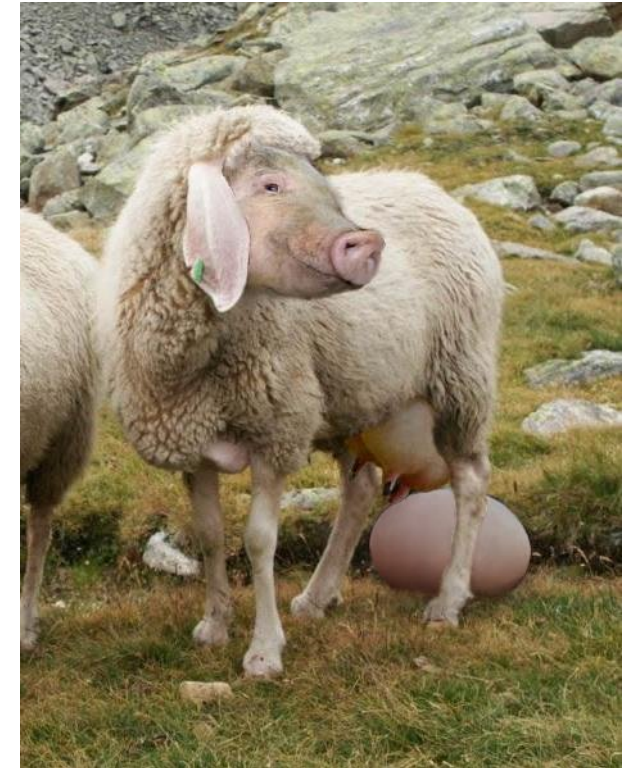


Bild: Wollmilchsau / Georg Mittenecker / Creative Commons