Technische Universität Braunschweig

Algorithmen und Datenstrukturen – Übung #5

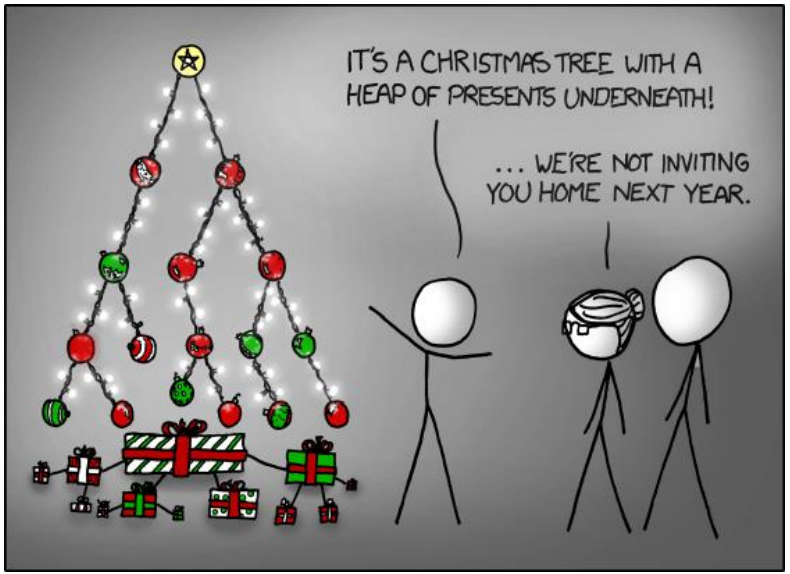
Dynamische Datenstrukturen

Matthias Konitzny, Arne Schmidt

17.12.2020

Heute

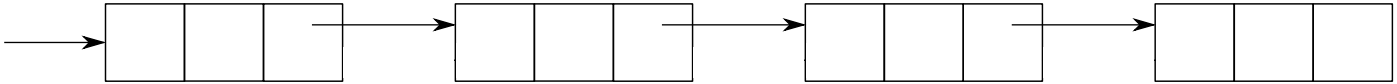
- Verkettete Listen
- Binäre Suchbäume
  - Operationen
  - Verschmelzen
- AVL-Bäume



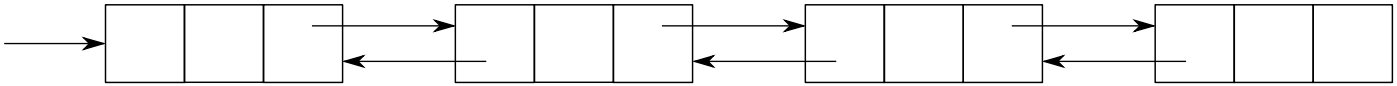
(Zyklisch) Verkettete Listen

Listen

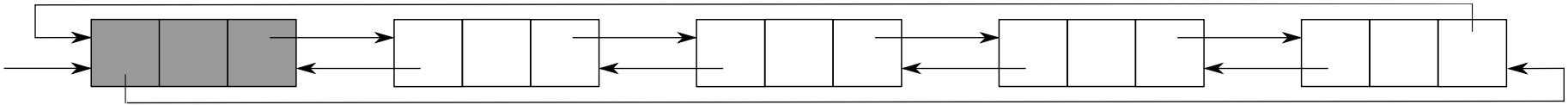
Einfach verkettet:



Doppelt verkettet:



Zyklisch doppelt verkettet (mit Wächter):



Laufzeiten in Listen

Operation	Einfach	Doppelt	Zyklisch
Suchen	$O(n)$	$O(n)$	$O(n)$
Einfügen	$O(1)$	$O(1)$	$O(1)$
Löschen	$O(n)$	$O(1)$	$O(1)$
Merge*	$O(1)$**	$O(1)$**	$O(1)$

*: Verschmelze zwei Listen der Größe n und m.
**: Sofern Adresse des letzten Elements bekannt. Andernfalls $O(\max(n, m))$.

Laufzeit Hierholzer-Algorithmus

Eingabe: Zusammenhängender Graph G mit höchstens 2 ungeraden Knoten

Ausgabe: Ein Eulerweg, bzw. eine Eulertour in G

1: starte in einem Knoten v
   (wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
2: verwende Algorithmus 2.7, um einen Weg W von v aus zu bestimmen
3: **while** es existieren unbenutzte Kanten **do**
4: wähle einen Knoten w aus W mit positivem Grad im Restgraphen
5: verwende Algorithmus 2.7, um einen Weg W' von w aus zu bestimmen
6: verschmelze W und W'

Algorithmus 2.8: Hierholzers Algorithmus zum Finden eines Eulerweges oder einer Eulertour

○ $O(n+m)$ □ $O(m)$? △ $O(n+m)$? ☆ $O(|W|+|W'|)$?

Laufzeit Hierholzer-Algorithmus

Eingabe: Graph G

Ausgabe: Ein Weg in G

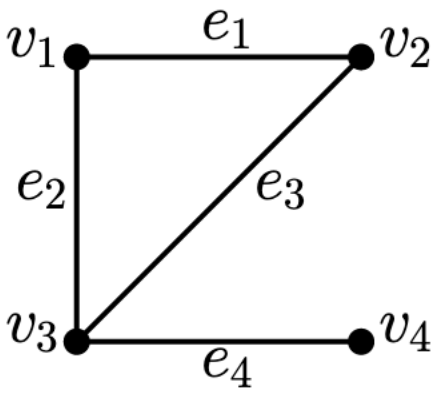
1: starte in einem Knoten v_0
   (wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
2: $i \leftarrow 0$
3: **while** es gibt eine zu v_i inzidente, unbenutzte Kante **do**
4: wähle eine zu v_i inzidente, unbenutzte Kante $\{v_i, v_j\}$
5: laufe zum Nachbarknoten v_j
6: lösche $\{v_i, v_j\}$ aus der Menge der unbenutzten Kanten
7: $v_{i+1} \leftarrow v_j$
8: $i \leftarrow i + 1$

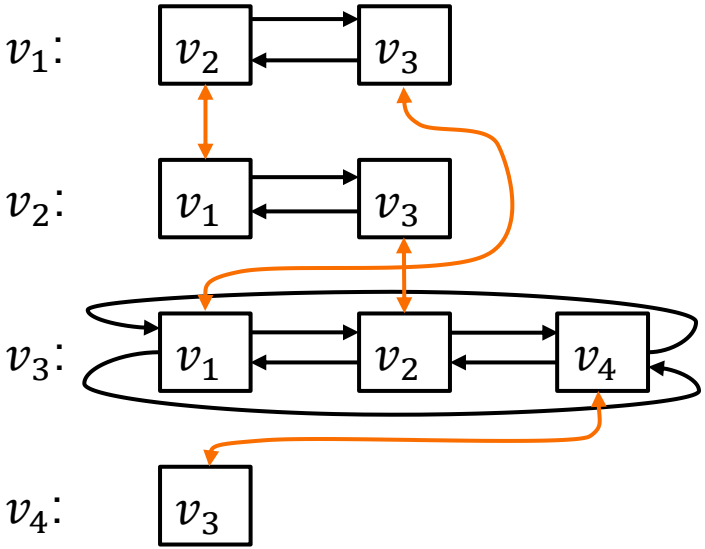
Algorithmus 2.7: Algorithmus zum Finden eines Weges in einem Graphen

Können wir einen Weg W in $O(|W|)$ Zeit bestimmen? Ideen?

Laufzeit Hierholzer-Algorithmus

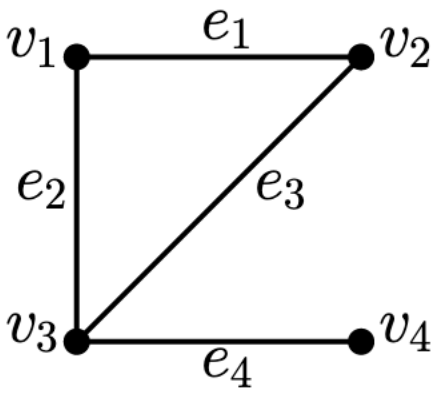
Benutze Adjazenzliste und Doppelt-Verkettete Listen.
Verwende zusätzliche Pointer für gleiche Kante.

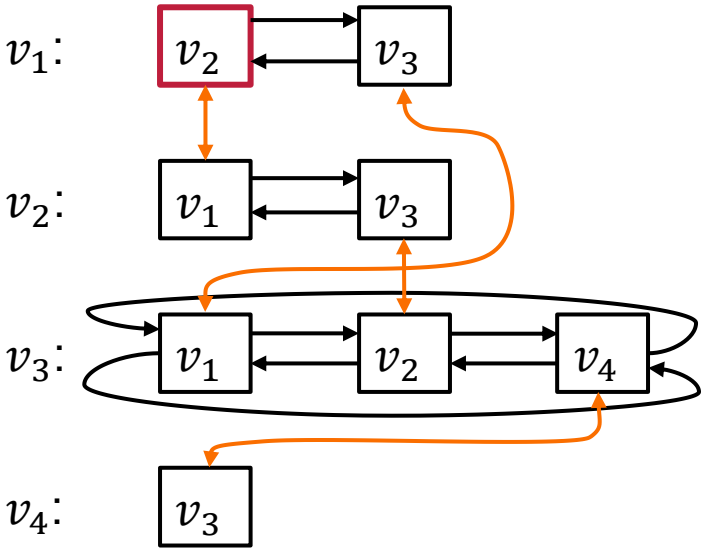




Laufzeit Hierholzer-Algorithmus

Starte bei v_1 und gehe zum nächsten Knoten.
Lösche die Kante aus der Adjazenzliste.
Alle Operationen kosten $O(1)$ Zeit.





Laufzeit Hierholzer-Algorithmus

Eingabe: Zusammenhängender Graph G mit höchstens 2 ungeraden Knoten

Ausgabe: Ein Eulerweg, bzw. eine Eulertour in G

1: starte in einem Knoten v
   (wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
2: verwende Algorithmus 2.7, um einen Weg W von v aus zu bestimmen
3: **while** es existieren unbenutzte Kanten **do**
4: wähle einen Knoten w aus W mit positivem Grad im Restgraphen
5: verwende Algorithmus 2.7, um einen Weg W' von w aus zu bestimmen
6: verschmelze W und W'

Algorithmus 2.8: Hierholzers Algorithmus zum Finden eines Eulerweges oder einer Eulertour

○ $O(n+m)$ □ $O(|W|)$ △ $O(n+m)$? ☆ $O(|W|+|W'|)$?

Laufzeit Hierholzer-Algorithmus

Nutze für die Eulertour/den Eulerweg auch eine zyklische doppelt-verkettete Liste.

Damit:

- Kosten für das Verschmelzen: $O(1)$
- Nächsten Startknoten suchen: $O(m)$ über alle Iterationen

Damit ist die Laufzeit also:

$$O(n+m) + O(m) + \sum_{\text{Wege } W} O(|W|) = O(n+m) + O(m) + O(m) = O(m)$$

Ersten Startknoten suchen

Weitere Startknoten suchen

Finden und Verschmelzen aller Wege

Da Graph zusammenhängend (also $n \leq m$)

Vergleich zu Fleury

Algorithmus	Hierholzer	Fleury	Fleury (mit Optimierungen)
Laufzeit	$O(m)$	$O(m^2)$	$O(m(\log m)^3 \log\log m)$

Binäre Suchbäume

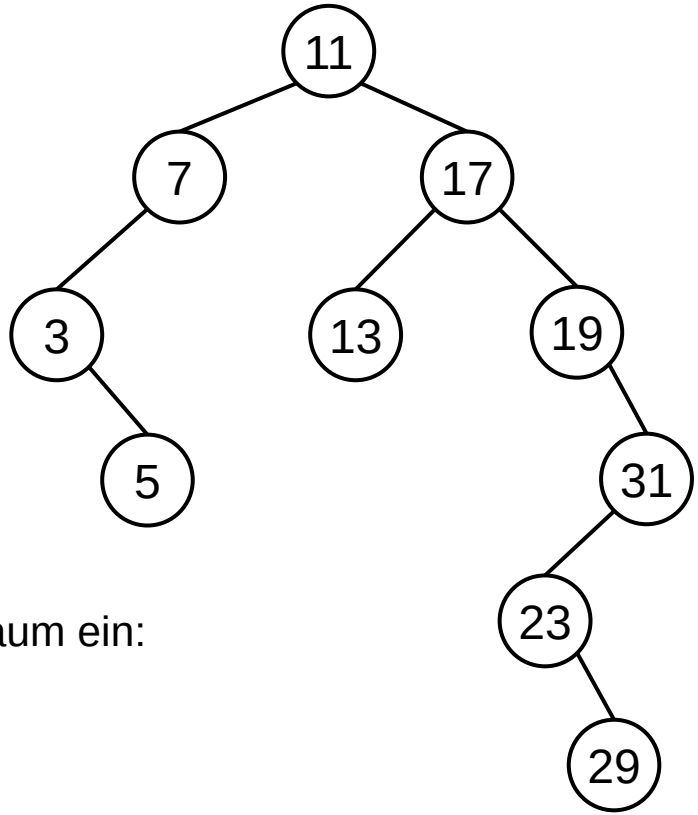
Bin. Suchbäume

Anstatt einen Nachfolger (Liste), benutze zwei:

- Ein linkes Kind ($l[v]$)
- Ein rechtes Kind ($r[v]$)

Verwalte zudem eine Totalordnung der Elemente:

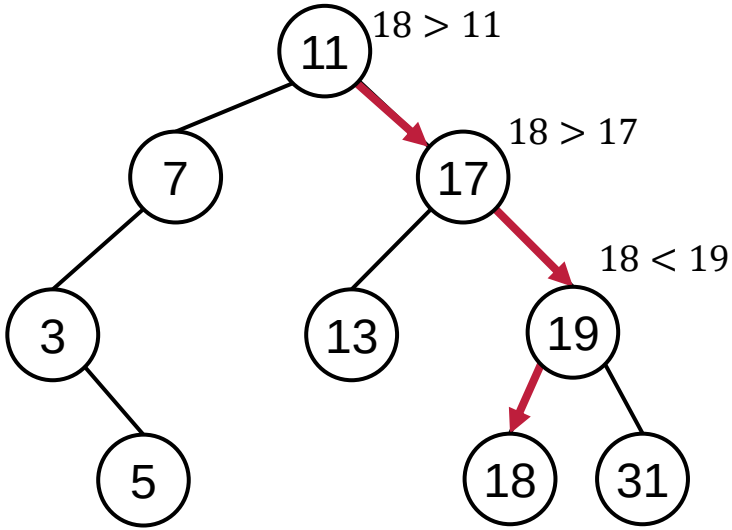
- Schlüssel im linken Teilbaum: kleiner
- Schlüssel im rechten Teilbaum: größer



Beispiel:
Füge folgende Sequenz von Zahlen in einen bin. Suchbaum ein:
11, 17, 13, 19, 7, 3, 31, 23, 5, 29

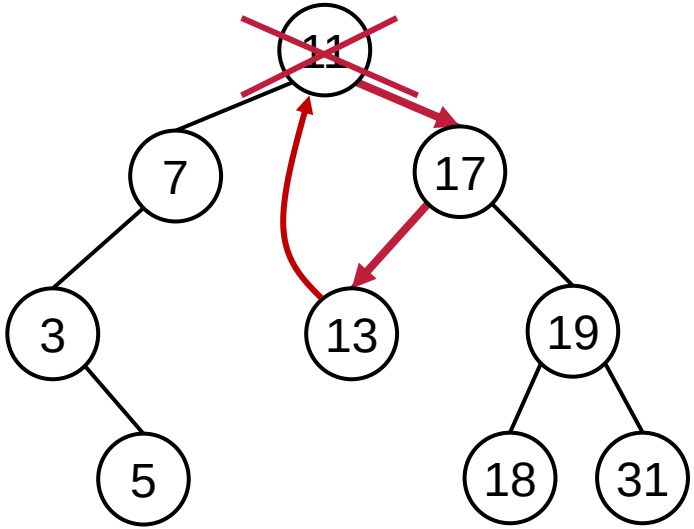
Operationen

- Insert (eben gesehen)
- Search
  - Search(18)



Operationen

- Insert (eben gesehen)
- Search
  - Search(18)
- Vorgänger (pred)/Nachfolger (succ)
  - $\mathrm{pred}(5) =$
  - $\mathrm{pred}(17) =$
  - $\mathrm{succ}(11) =$
  - $\mathrm{succ}(5) =$
- Delete
  - Delete(11)

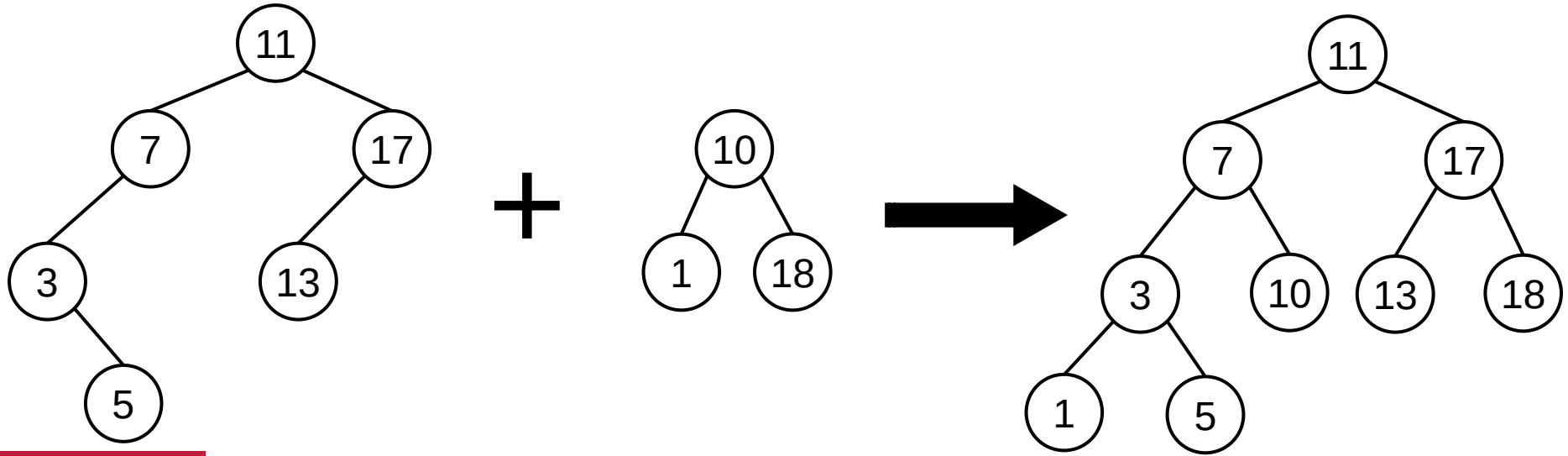


Verschmelzen von Binären Suchbäumen

Merge – Das Problem

Gegeben: Zwei binäre Suchbäume mit n bzw. m Elementen.
Aufgabe: Konstruiere daraus einen Suchbaum mit $n + m$ Elementen.

Wie schnell geht das?



Merge – Ideen und Schranken

Erste Möglichkeit:

Für jeden Schlüssel S im ersten Suchbaum: Füge S in den zweiten Suchbaum ein.

Laufzeit:

$O(nh_m)$, wobei h_m die Höhe des zweiten Suchbaums ist.

Worst-Case: $O(nm)$

Wie lange braucht man mindestens?

Man muss jeden Schlüssel mindestens einmal betrachten.

Also: $\Omega(n + m)$

Können wir diesen Unterschied zwischen oberer und unterer Schranke schließen?

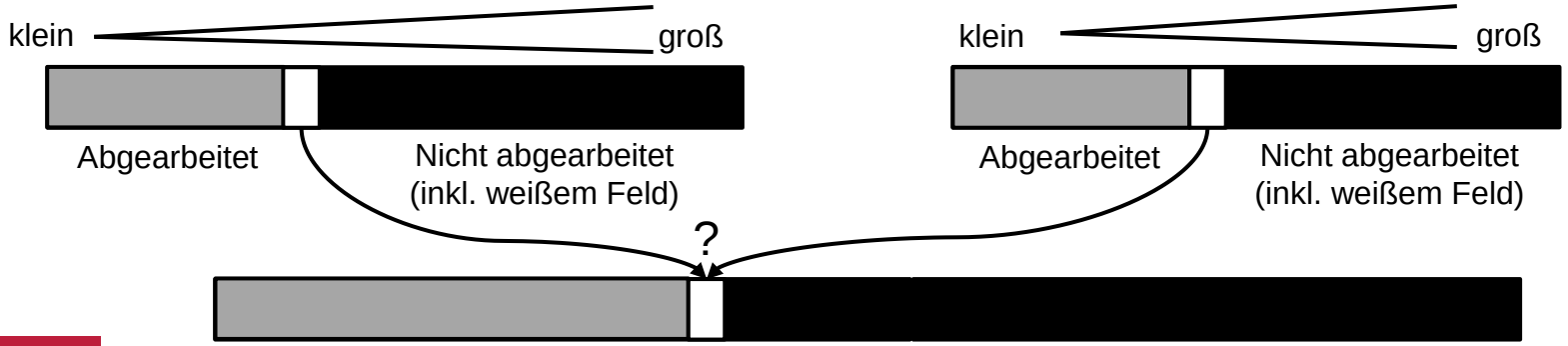
Merge – Alternative

Probieren wir folgende Strategie:

1. Transformiere beide Suchbäume in sortiere Arrays (durch inorder Traversierung).
2. Verschmelze beide Arrays in ein sortiertes Array.
3. Konstruiere aus dem sortierten Array einen Suchbaum.

Punkt 1 benötigt offensichtlich $O(n + m)$ Zeit.

Für Punkt 2 benötigen wir $O(n + m)$ Zeit:

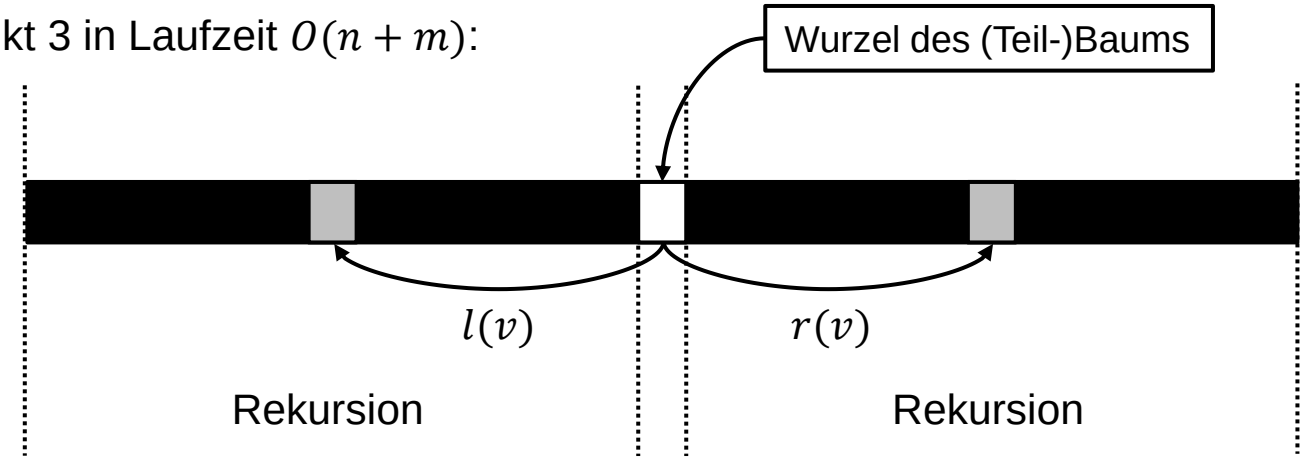


Merge – Alternative

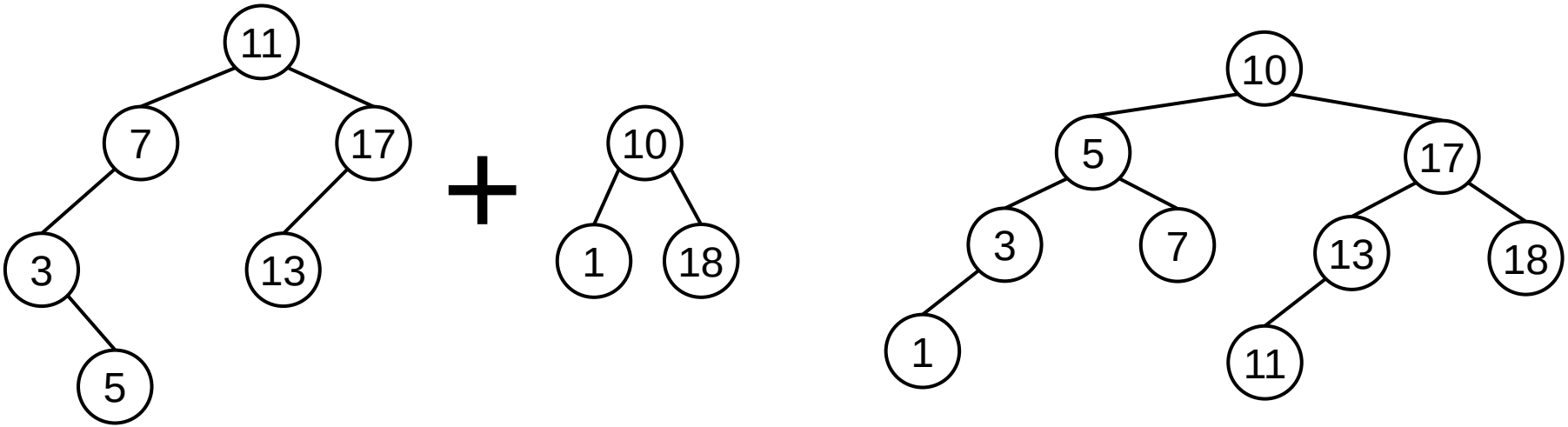
Probieren wir folgende Strategie:

1. Transformiere beide Suchbäume in sortiere Arrays (durch inorder Traversierung).
2. Verschmelze beide Arrays in ein sortiertes Array.
3. Konstruiere aus dem sortierten Array einen Suchbaum.

Für Punkt 3 in Laufzeit $O(n+m)$:



Merge - Beispiel



3	5	7	11	13	17	,	1	10	18	⇒	1	3	5	7	10	11	13	17	18

Laufzeiten der Operationen

Operation	Einfach
Suchen	$O(h)$
Einfügen	$O(h)$
Löschen	$O(h)$
Traversierung	$O(n)$
Merge	$O(n+m)$

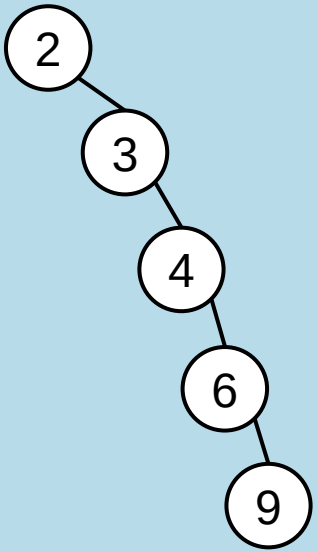
AVL-Bäume

Balancierte Suchbäume - Motivation

Wir wissen: Einfügen, Entfernen und Suchen in binären Suchbäumen geht in $O(h)$.

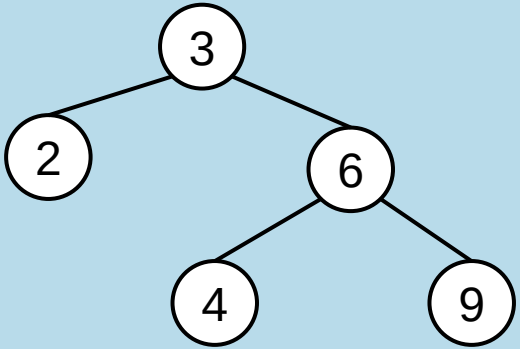
Konstruiere den folgenden Suchbaum durch Einfügen von 2, 3, 4, 6, 9:

Naive Konstruktion



Problem: Da $h = n$ ist, liegen Einfügen, Entfernen und Suchen in $O(n)$

Balancierter Suchbaum

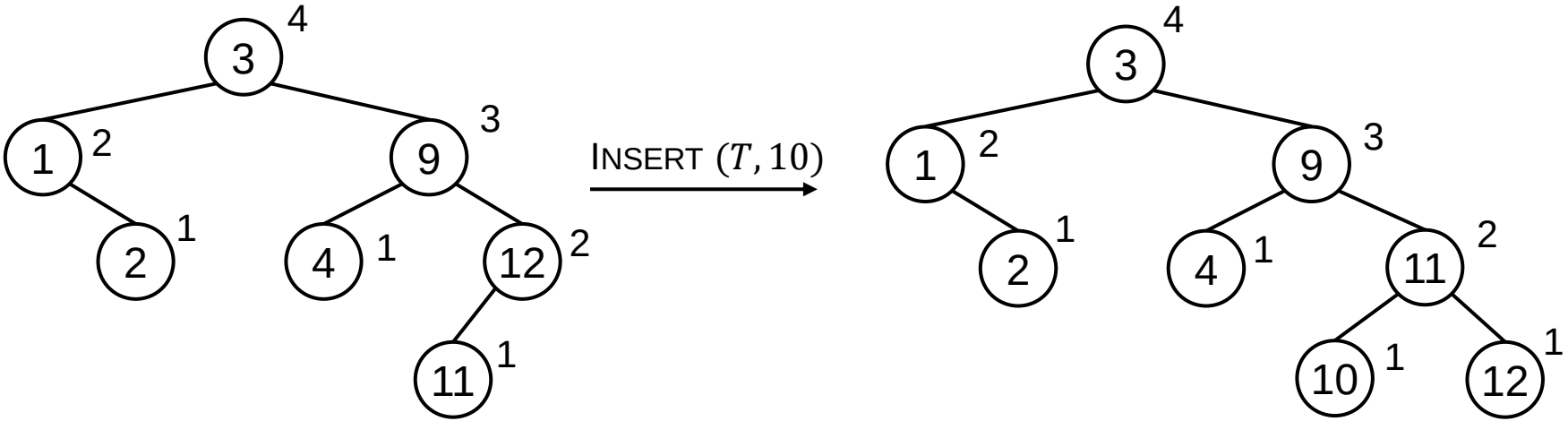


$h = \log(n)$: Einfügen, Suchen, Entfernen in $O(\log(n))$

AVL-Bäume – Definition

Ein AVL-Baum besitzt folgenden Eigenschaften:

- Er ist ein binärer Suchbaum.
- Höhe des linken und rechten Teilbaums eines Knotens unterscheidet sich um maximal 1.



AVL-Bäume – Operationen

Operationen für binäre Suchbäume funktionieren auch für AVL-Bäume, d.h. wir können:

- Insert,
- Delete,
- Minimum/Maximum,
- Predecessor/Successor,
- …

ausführen.

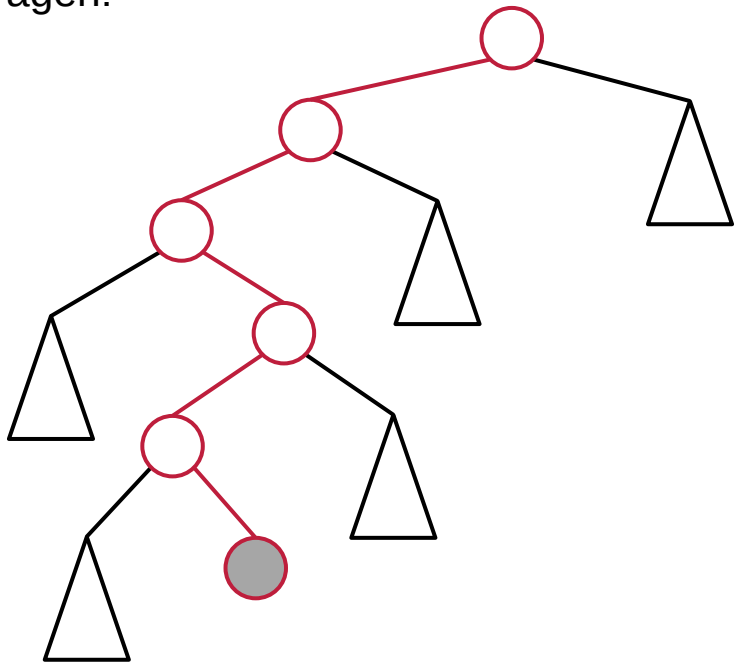
Um die Balancierung zu erhalten, müssen nur Operationen verändert werden, die die Struktur des Baumes verändern.
Das sind nur *Insert* und *Delete.*

AVL-Bäume – Restructure

Bei Insert und Delete stellen sich nun folgende Fragen:

1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?

Zu 1.: Nur Knoten, die auf dem Pfad von der Wurzel zum eingefügten/gelöschten Knoten liegen können unbalanciert werden.

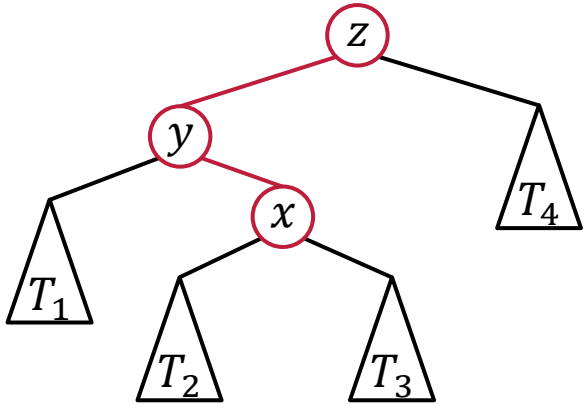


AVL-Bäume – Restructure

Bei Insert und Delete stellen sich nun folgende Fragen:

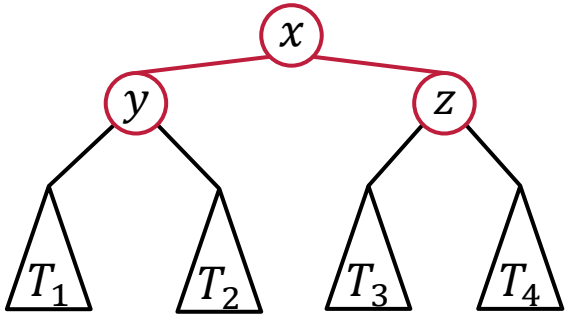
1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?

Zu 2.: Betrachte den unbalancierten Knoten z,
sein Kind y und dessen Kind x.



Sortiere Elemente aufsteigend und rotiere entsprechend:

1. $x \leq y \leq z$
2. $y \leq x \leq z$
3. $z \leq x \leq y$
4. $z \leq y \leq x$



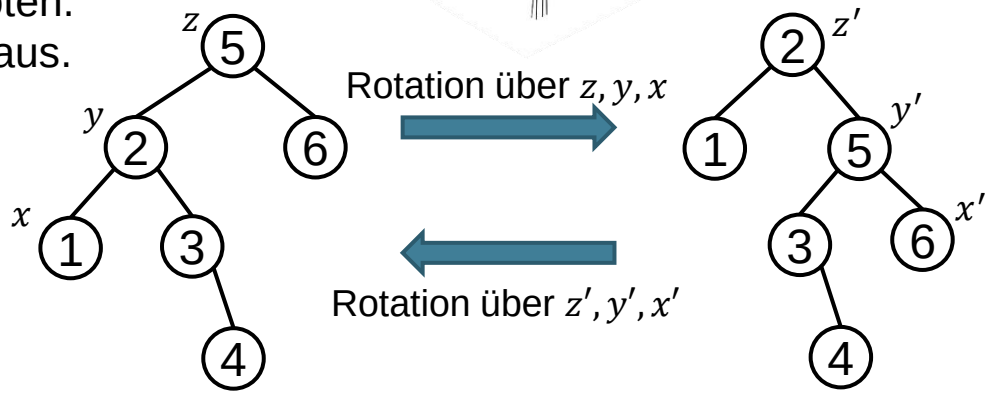
AVL-Bäume – Restructure

Bei Insert und Delete stellen sich nun folgende Fragen:

1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?

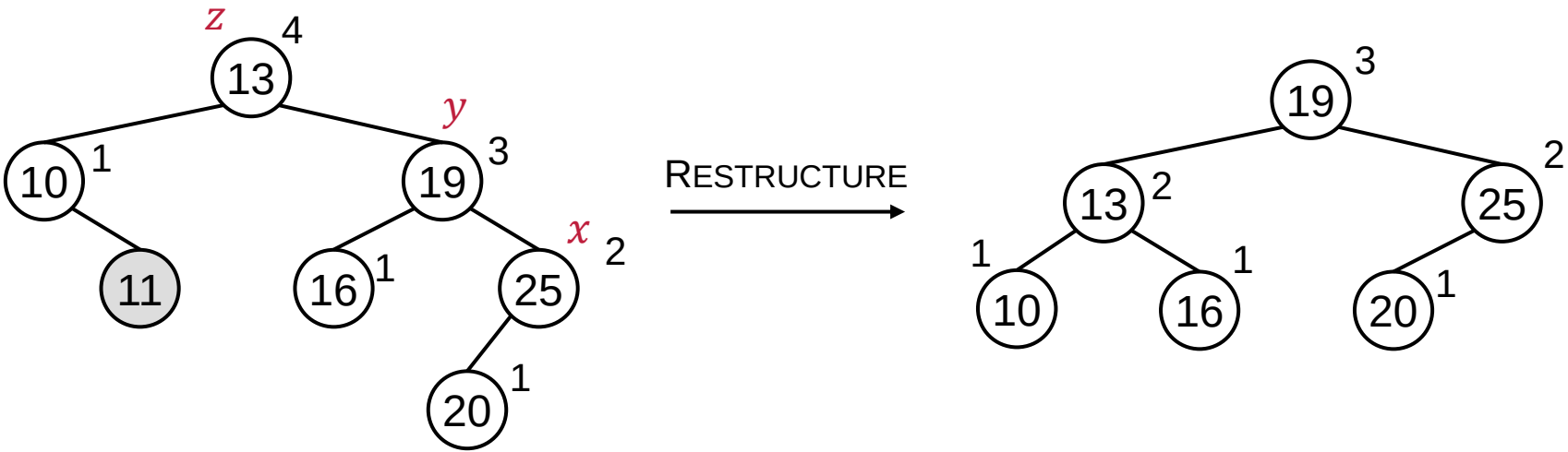
Zu 3.:

- Starte bei tiefstem unbalancierten Knoten.
- Wähle Kinder (x, y) nach deren Höhe aus.



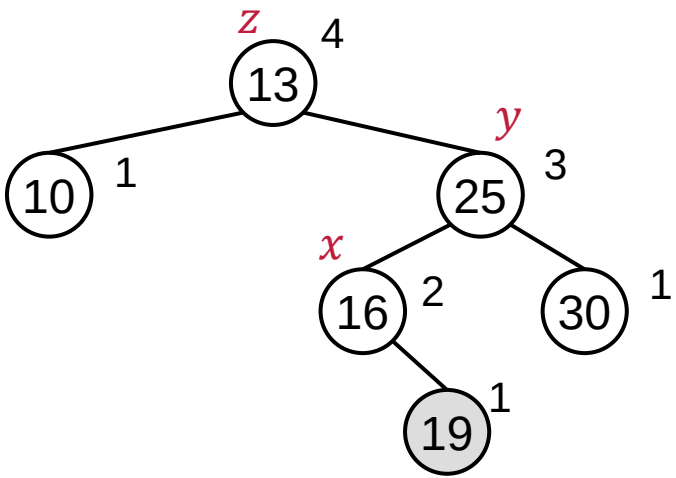
AVL-Bäume – Beispiele

DELETE$(T, 11)$

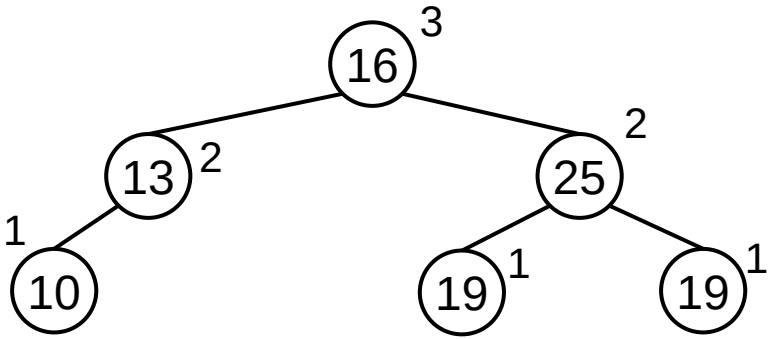


AVL-Bäume – Beispiele

INSERT$(T, 19)$

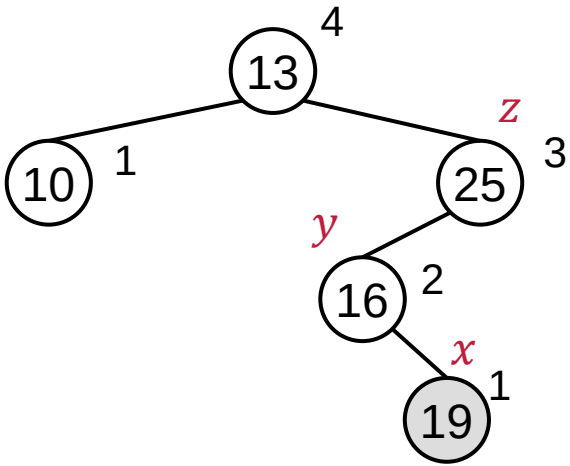


RESTRUCTURE



AVL-Bäume – Beispiele

INSERT$(T, 19)$



RESTRUCTURE