



Technische
Universität
Braunschweig



Algorithmen und Datenstrukturen – Übung #4

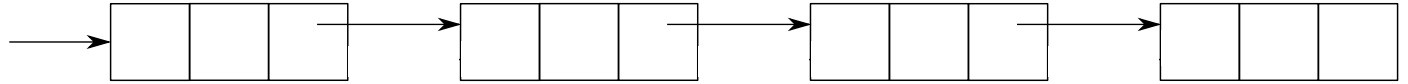
Dynamische Datenstrukturen

Ramin und Chek-Manh
11.01.2024

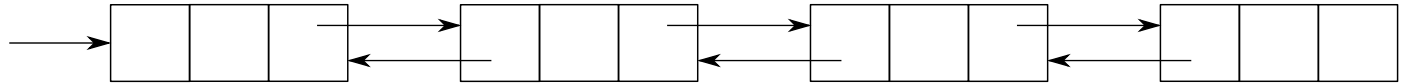
(Zyklisch) Verkettete Listen

Listen

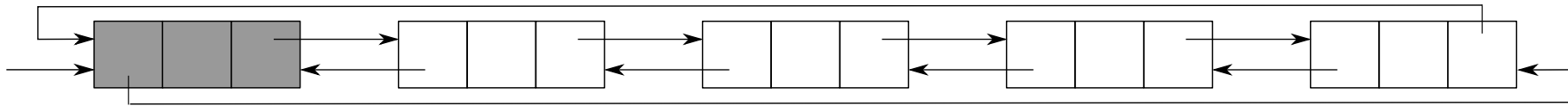
Einfach verkettet:



Doppelt verkettet:

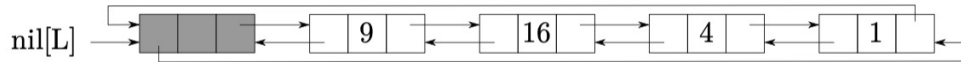
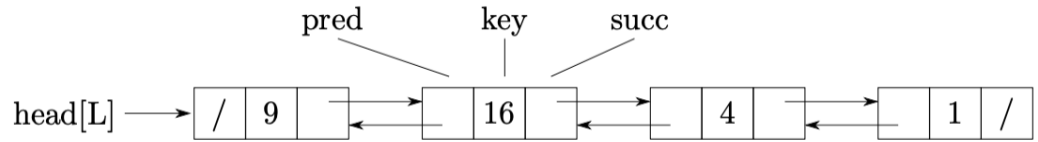


Zyklisch doppelt verkettet (mit Wächter):



Wächter

```
1: function LISTE-DELETE( $L, x$ )
2:   if pred[ $x$ ]  $\neq$  NIL then
3:     succ[pred[ $x$ ]]  $\leftarrow$  succ[ $x$ ]
4:   else
5:     head[ $L$ ]  $\leftarrow$  succ[ $x$ ]
6:   if succ[ $x$ ]  $\neq$  NIL then
7:     pred[succ[ $x$ ]]  $\leftarrow$  pred[ $x$ ]
```



```
1: function LIST-DELETE'( $L, x$ )
2:   succ[pred[ $x$ ]]  $\leftarrow$  succ[ $x$ ]
3:   pred[succ[ $x$ ]]  $\leftarrow$  pred[ $x$ ]
```

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 $O(\min(n, m))$.


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

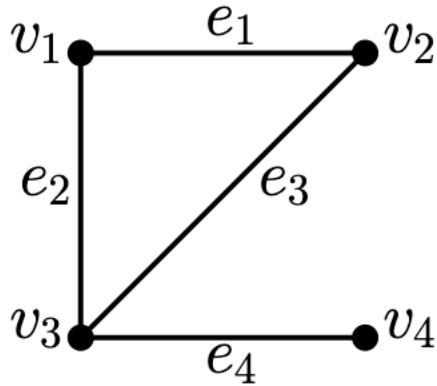
- 1: starte in einem Knoten v_0
(wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
- 2: $i \leftarrow 0$
- 3: **while** es gibt eine zu v_i inzidente, unbenutzte Kante **do**
- 4: wähle eine zu v_i inzidente, unbenutzte Kante $\{v_i, v_j\}$
- 5: laufe zum Nachbarknoten v_j
- 6: lösche $\{v_i,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



v_1 : v_2, v_3

v_2 : v_3, v_1

v_3 : v_1, v_2, v_4

v_4 : v_3

Laufzeit Hierholzer-Algorithmus

Benutze Adjazenzliste und Doppelt-Verkettete Listen.

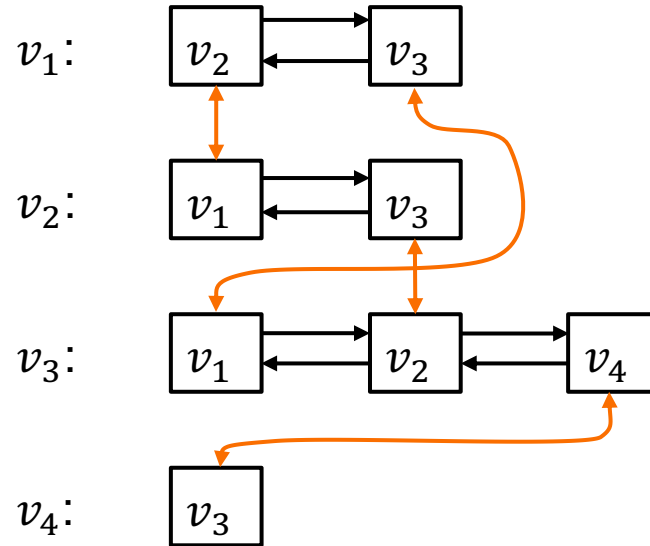
Verwende zusätzliche **Pointer** für gleiche Kante.

v_1 : v_2, v_3

v_2 : v_3, v_1

v_3 : v_1, v_2, v_4

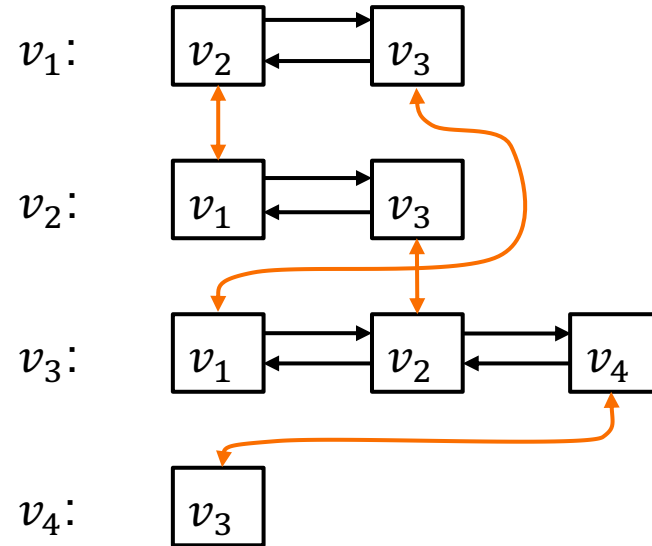
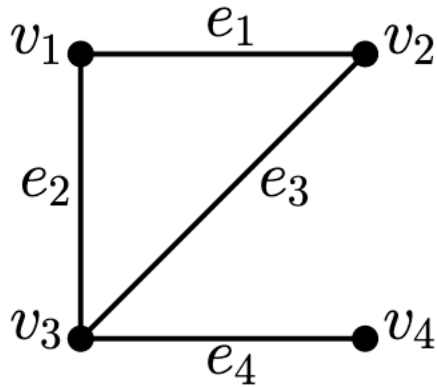
v_4 : v_3



Laufzeit Hierholzer-Algorithmus

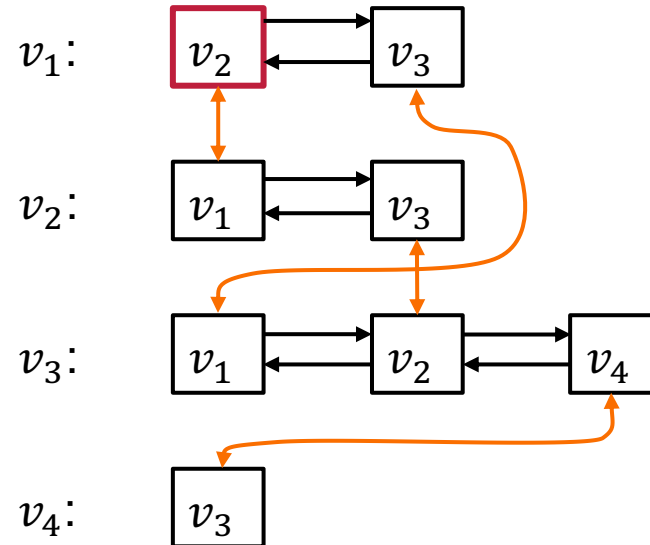
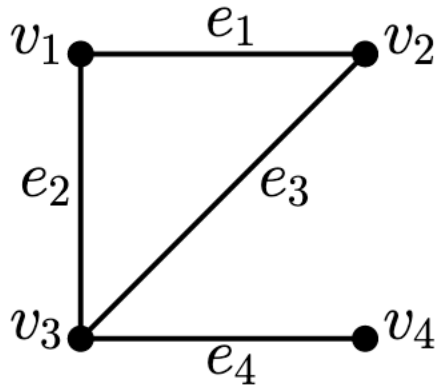
Benutze Adjazenzliste und Doppelt-Verkettete Listen.

Verwende zusätzliche **Pointer** für gleiche Kante.



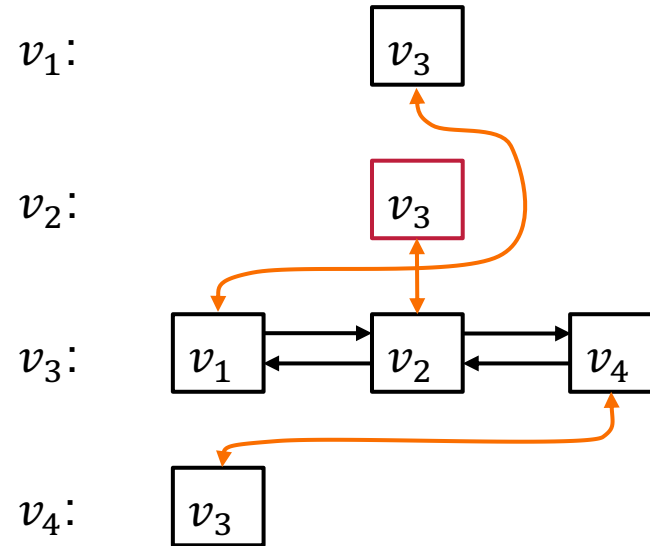
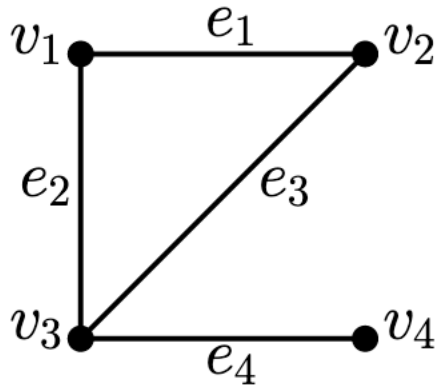
Laufzeit Hierholzer-Algorithmus

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



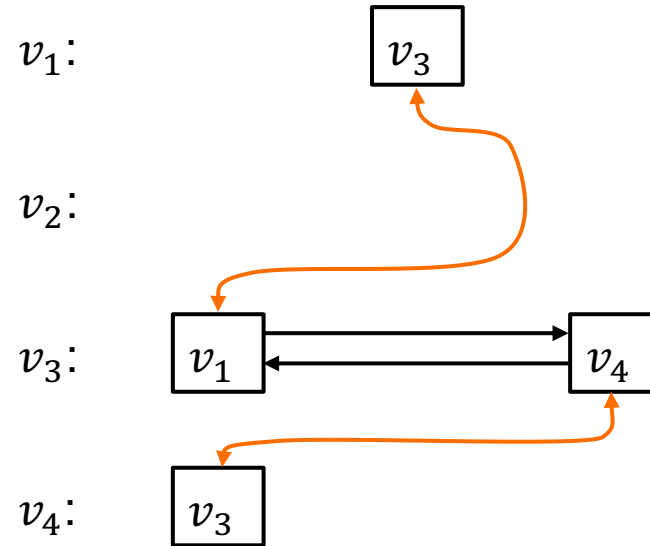
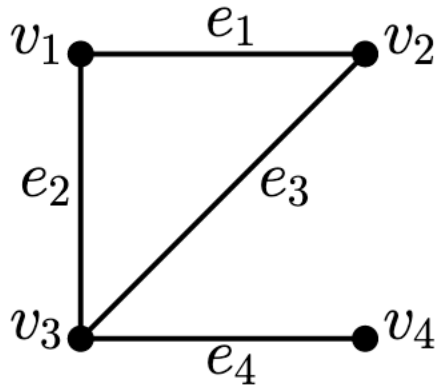
Laufzeit Hierholzer-Algorithmus

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



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) ¹
Laufzeit	$O(m)$	$O(m^2)$	$O(m(\log m)^3 \log \log m)$

1. <https://dl.acm.org/doi/pdf/10.1145/335305.335345>

Binäre Suchbäume

Bin. Suchbäume

Anstatt einen Nachfolger (Liste), benutze zwei:

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

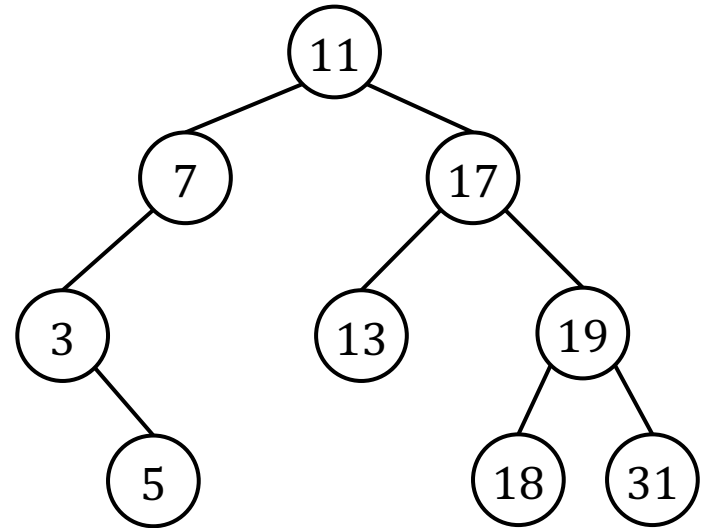
Verwalte zudem eine Totalordnung der Elemente:

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

Beispiel:

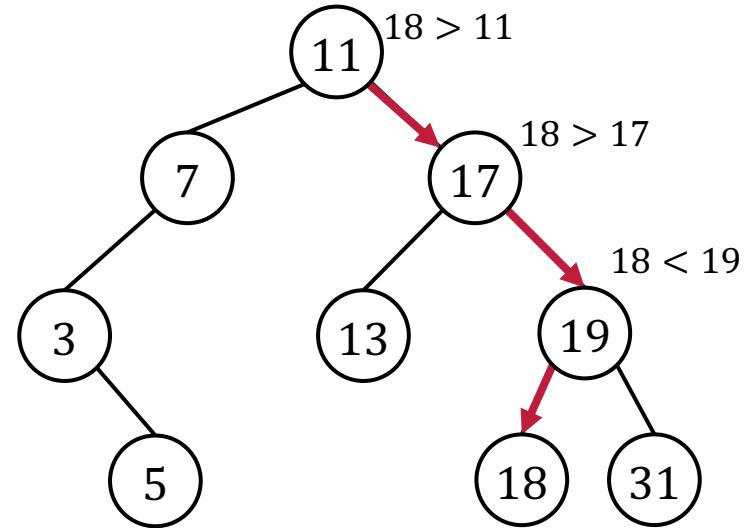
Füge folgende Sequenz von Zahlen in einen bin. Suchbaum ein:

11, 17, 13, 19, 7, 3, 31, 18, 5



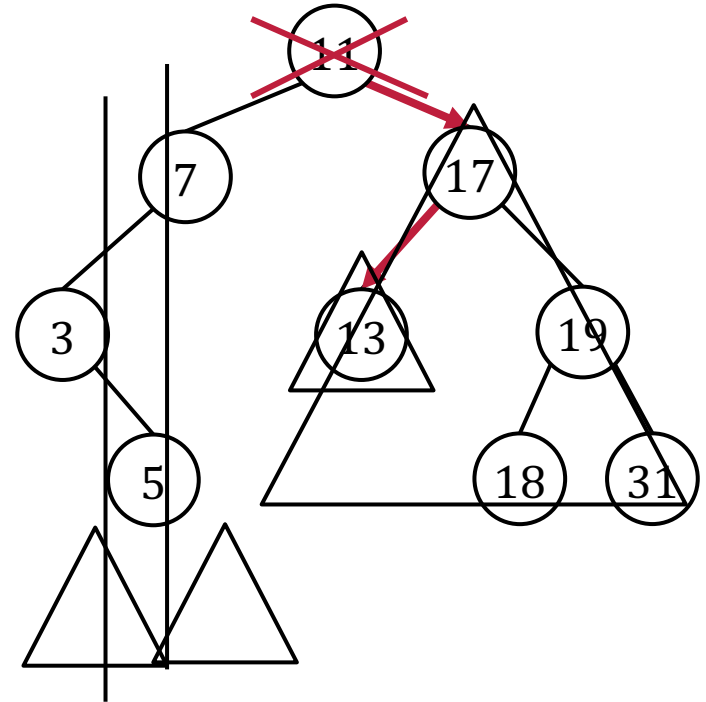
Operationen

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



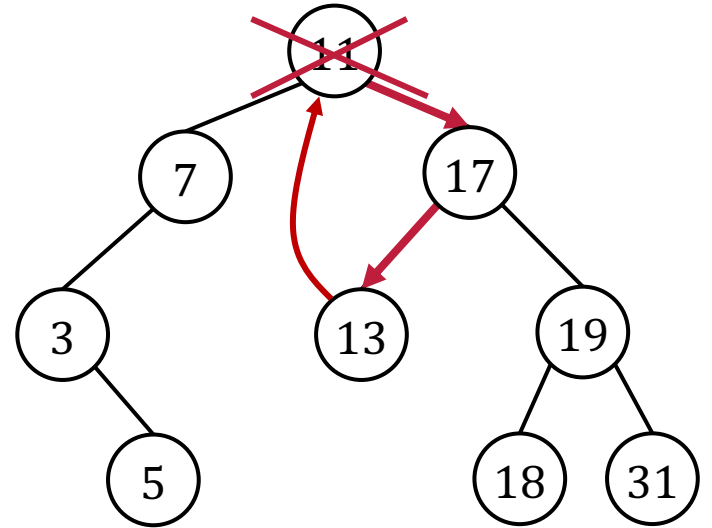
Operationen

- Insert (eben gesehen)
- Search
 - search(18)
- Vorgänger (pred)/Nachfolger (succ)
 - pred(5) =
 - pred(17) =
 - succ(11) =
 - succ(5) =
- Delete
 - delete (11)



Operationen

- Insert (eben gesehen)
- Search
 - `search(18)`
- Vorgänger (pred)/Nachfolger (succ)
 - `pred(5) = 3`
 - `pred(17) = 13`
 - `succ(11) = 13`
 - `succ(5) = 7`
- Delete
 - `delete(11)`



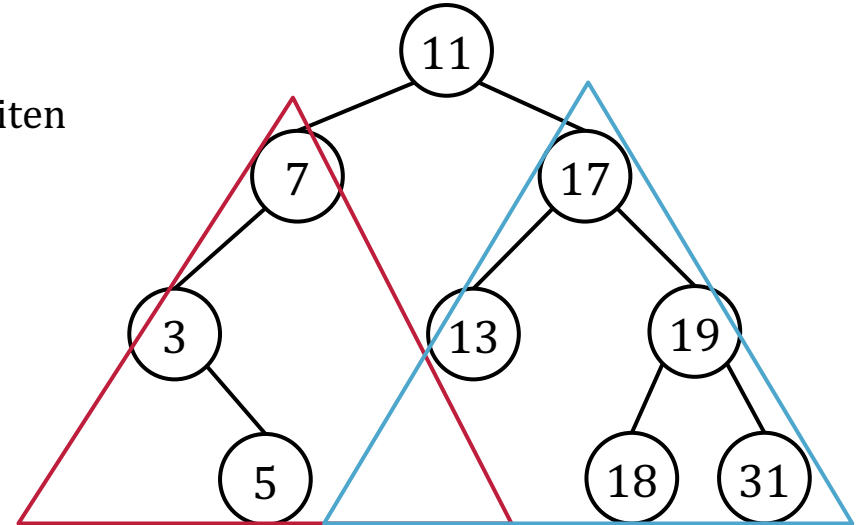
Traversierung

Wie durchläuft man einen binären Suchbaum?

Man unterscheidet unter anderem drei Möglichkeiten

- **Inorder** (Links, Wurzel, Rechts)
 - 3, 5, 7, 11, 13, 17, 18, 19, 31 →
- **Preorder** (Wurzel, Links, Rechts)
 - 11, 7, 3, 5, 17, 13, 19, 18, 31
- **Postorder** (Links, Rechts, Wurzel)
 - 5, 3, 7, 13, 18, 31, 19, 17, 11

$$h(W) = \max(h(L), h(R)) + 1$$



Traversierung

Dank der Baumstruktur kann man das ganz einfach als Algorithmus aufschreiben.

<pre>function INORDER(v) if ($v \neq \text{NIL}$) INORDER($l(v)$) print $S(v)$ INORDER($r(v)$)</pre>	<pre>function POSTORDER(v) if ($v \neq \text{NIL}$) POSTORDER($l(v)$) POSTORDER($r(v)$) print $S(v)$</pre>	<pre>function PREORDER(v) if ($v \neq \text{NIL}$) print $S(v)$ PREORDER($l(v)$) PREORDER($r(v)$)</pre>
Inorder	Postorder	Preorder

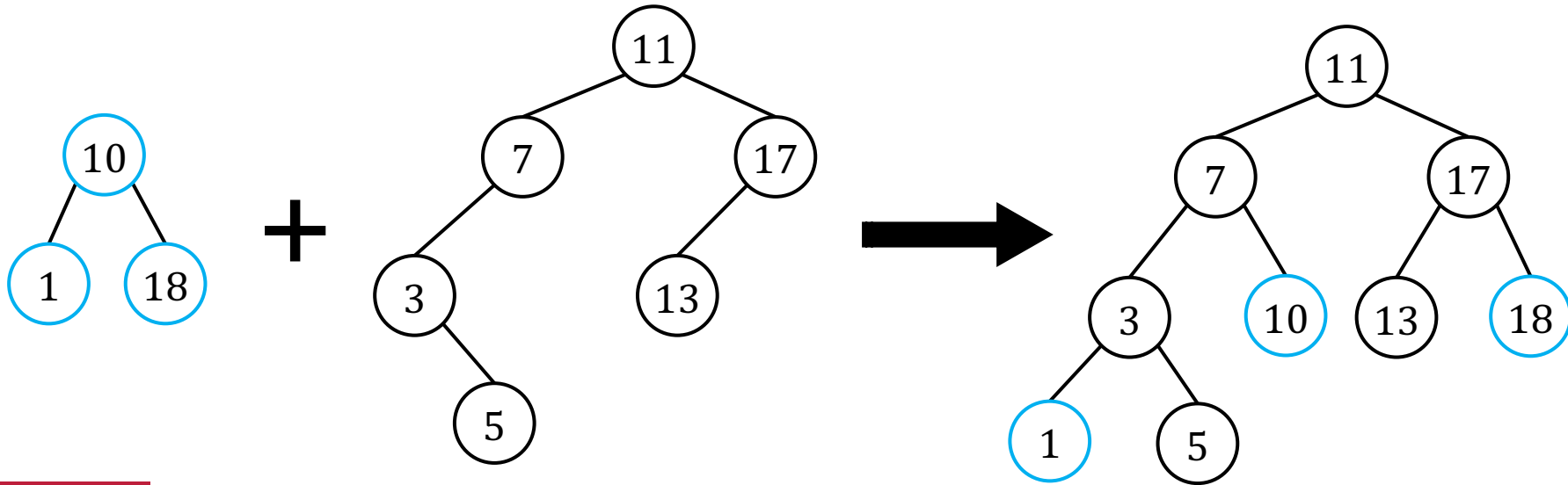
Verschmelzen von binären Suchbäumen

Merge – Das Problem

Gegeben: Zwei binäre Suchbäume mit n bzw. m Elementen.

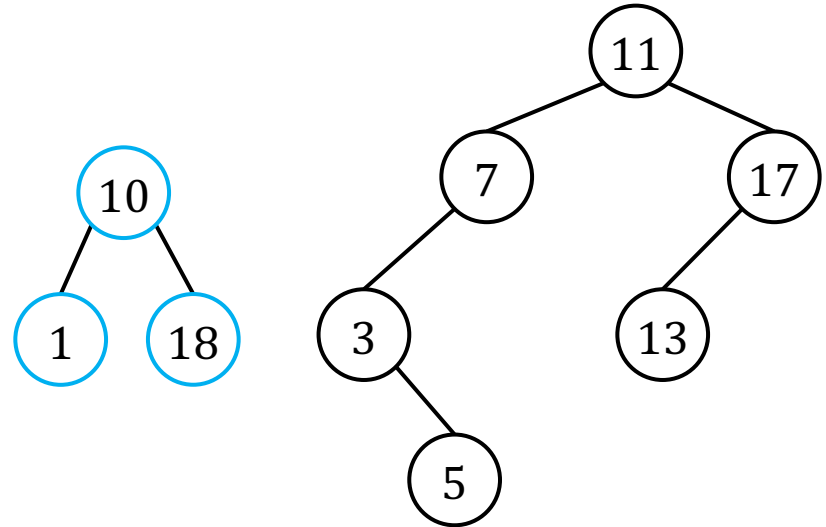
Aufgabe: Konstruiere daraus einen Suchbaum mit $n + m$ Elementen.

Wie (schnell) geht das?



Merge – Ideen und Schranken

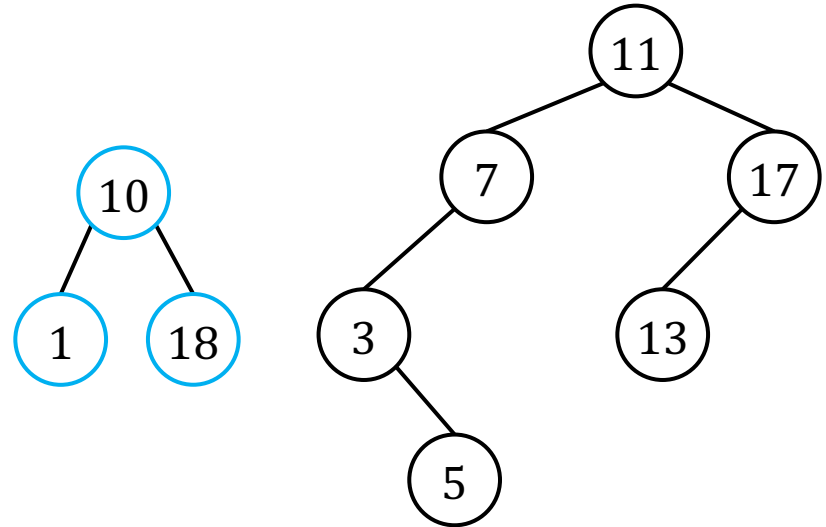
Idee



Merge – Ideen und Schranken

Idee

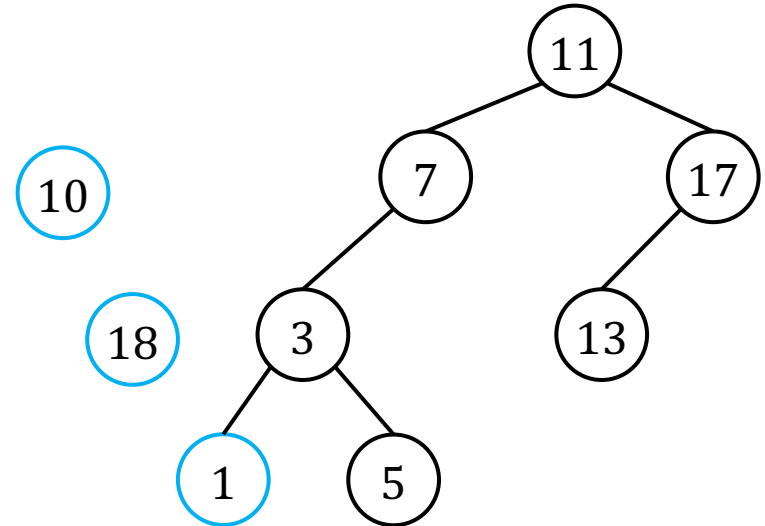
Für jeden Schlüssel S im ersten Suchbaum: Füge S in den zweiten Suchbaum ein.



Merge – Ideen und Schranken

Idee

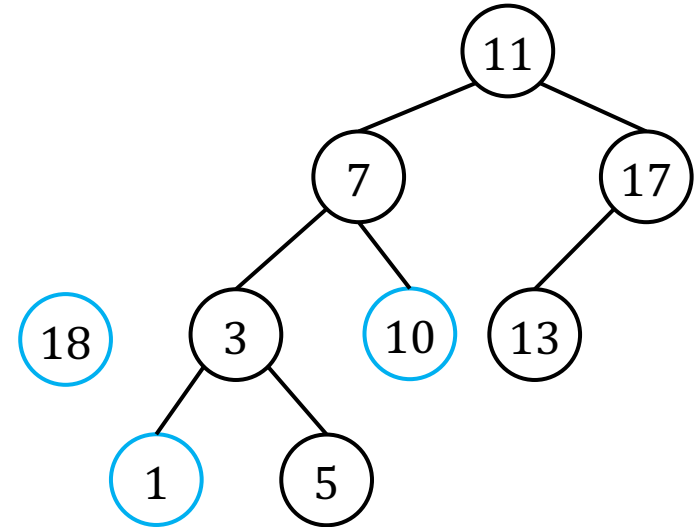
Für jeden Schlüssel S im ersten Suchbaum: Füge S in den zweiten Suchbaum ein.



Merge – Ideen und Schranken

Idee

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



Merge – Ideen und Schranken

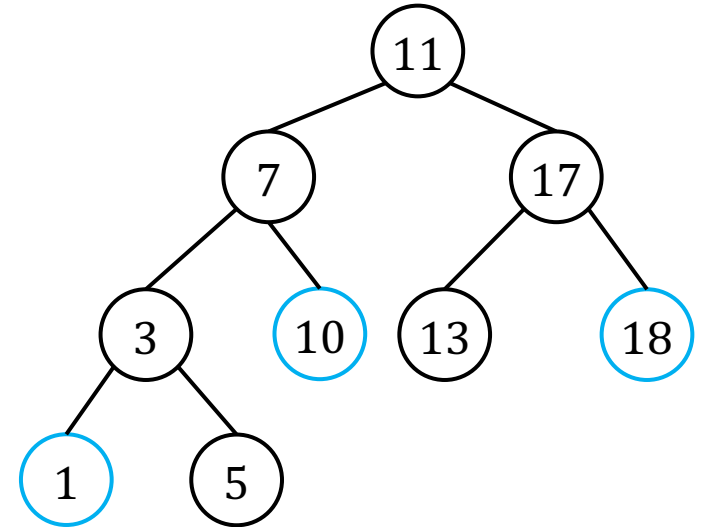
Idee

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

Laufzeit

Jeder Schlüssel n muss in den zweiten Suchbaum der Höhe h_m eingefügt werden.

Entsprechend erhalten wir $O(n \cdot h_m)$, sowie $O(n \cdot m)$ im Worst-Case.



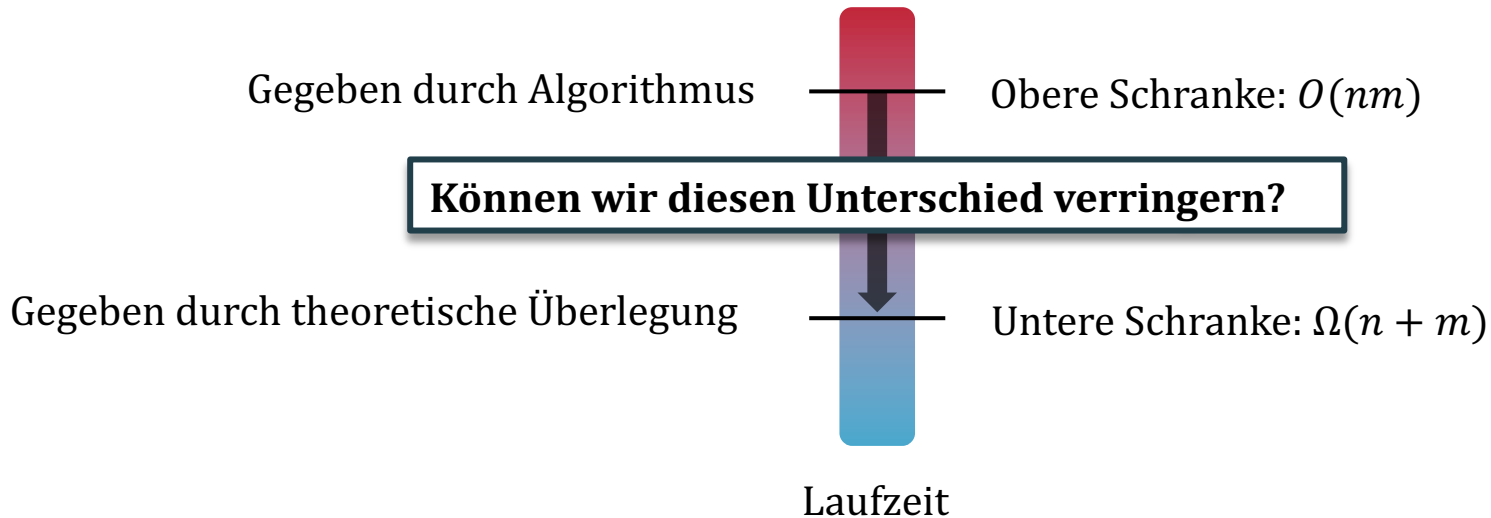
Ist das eine gute Laufzeit?

Merge – Ideen und Schranken

Ist das eine gute Laufzeit?

Wie lange braucht man mindestens?

Jeder Schlüssel muss mindestens einmal betrachtet werden. $\rightarrow \Omega(n + m)$



Ideen?

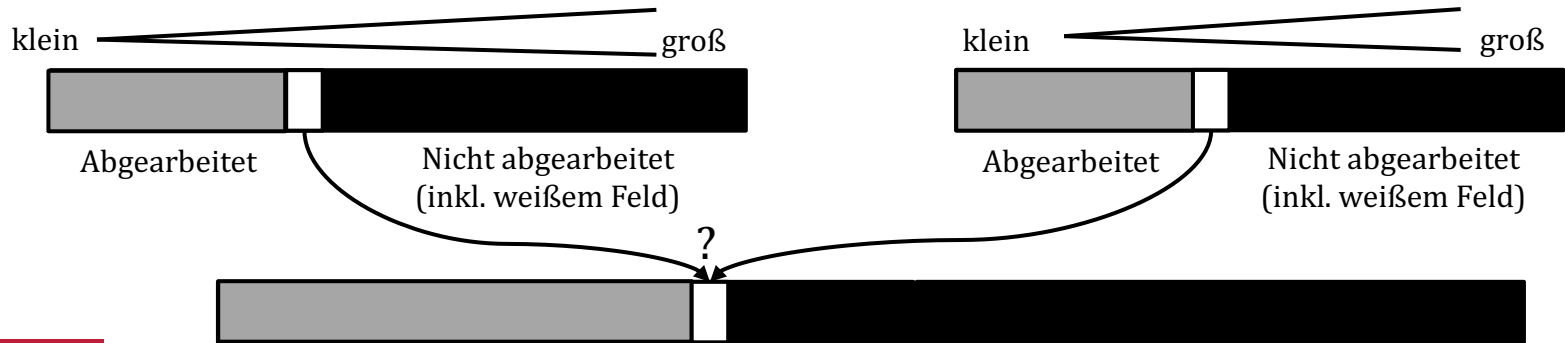
Merge – Alternative

Probieren wir folgende Strategie:

1. Transformiere beide Suchbäume in sortierte 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:

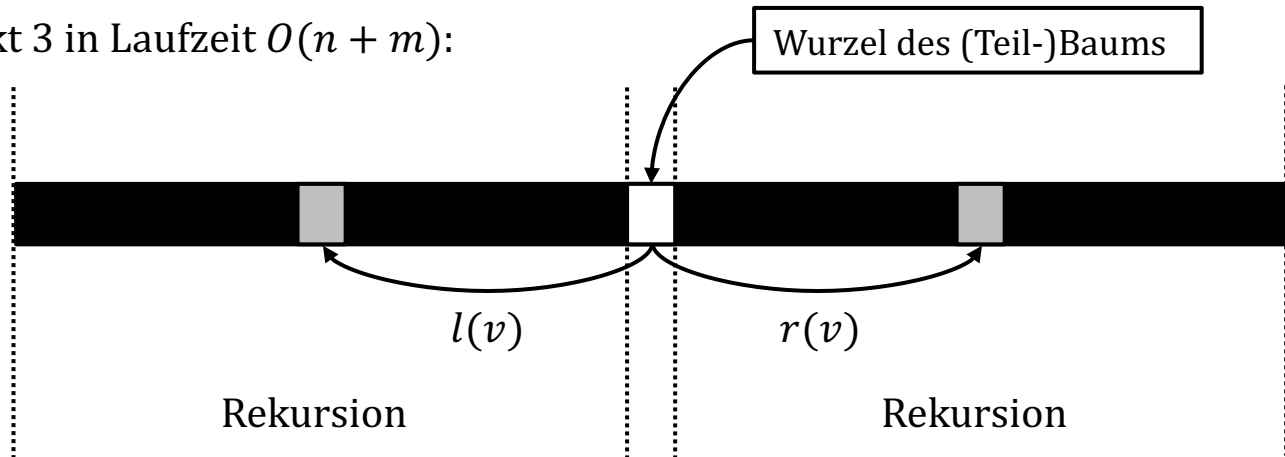


Merge – Alternative

Probieren wir folgende Strategie:

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

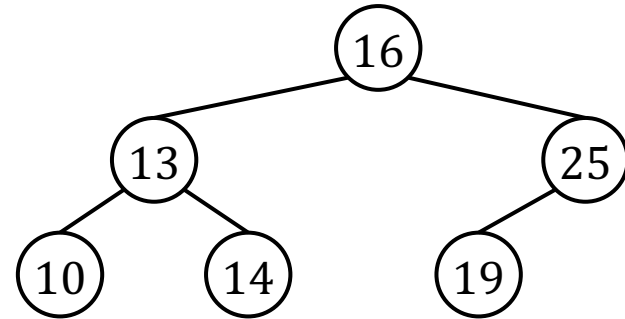
Für Punkt 3 in Laufzeit $O(n + m)$:



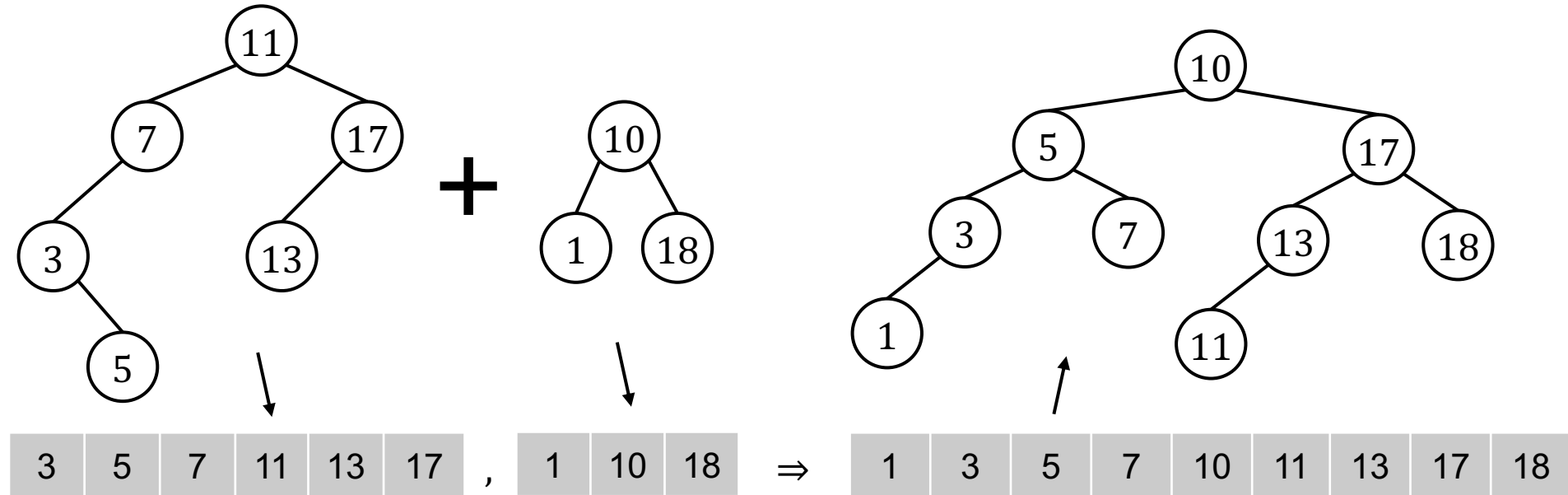
Einen binären Baum aus einem Array bauen

```
function BUILD_FROM_ARRAY( $A, p, r$ )  
  if ( $p > r$ )  
    return NIL  
   $q = \lfloor \frac{p+r}{2} \rfloor$   
   $root = \text{Tree}(A(q))$   
   $l[root] = \text{BUILD\_FROM\_ARRAY}(A, p, q - 1)$   
   $r[root] = \text{BUILD\_FROM\_ARRAY}(A, q + 1, r)$   
  return  $root$ 
```

$A = \{10, 13, 14, 16, 19, 25\}$



Merge - Beispiel



Zusammenfassung: Laufzeiten in binären Suchbäumen

Zusammenfassung: Laufzeiten in binären Suchbäumen

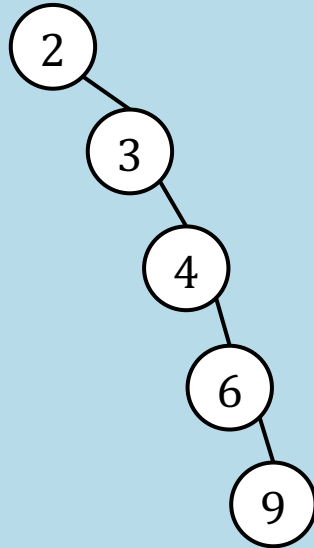
Operation	Laufzeit
Suchen	$O(h)$
Einfügen	$O(h)$
Löschen	$O(h)$
Traversierung	$O(n)$
Merge	$O(n + m)$

Balancierte Suchbäume - Motivation

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

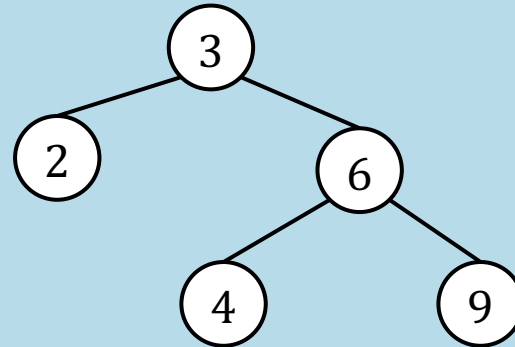
Konstruiere den folgenden Suchbaum durch Einfügen von 2, 3, 4, 6, 9:

Naive Konstruktion



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

Balancierter Suchbaum



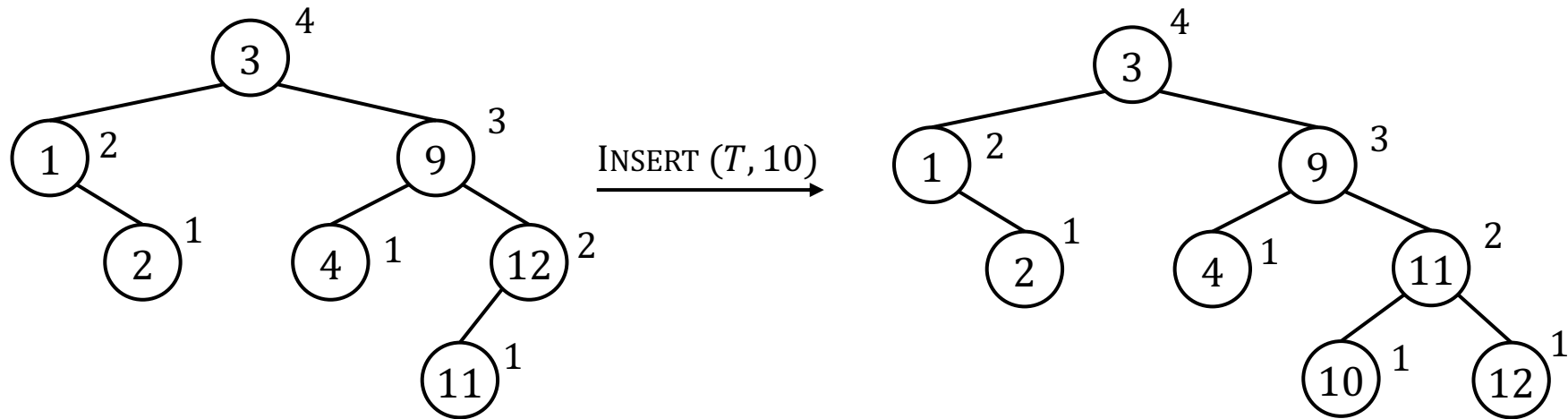
$h = \log(n)$:
Einfügen,
Suchen,
Entfernen in
 $O(\log(n))$

AVL-Bäume

AVL-Bäume – Definition

Ein AVL-Baum besitzt folgenden Eigenschaften:

- Er ist ein binärer Suchbaum.
- Höhe des linken und rechten Teilbaums jeden 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 *Insert* und *Delete*.

AVL-Bäume

Bei Insert und Delete stellen sich nun folgende Fragen:

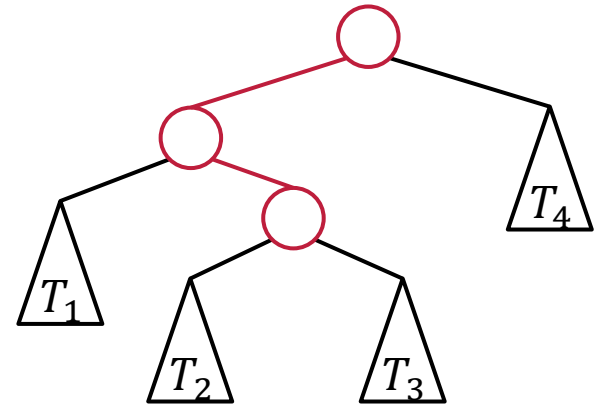
1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?

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?

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



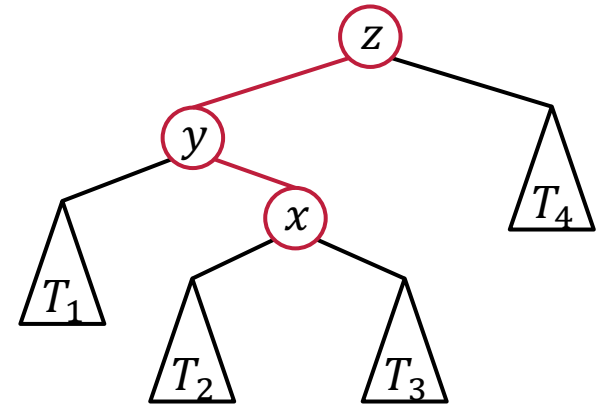
AVL-Bäume – Restructure

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?

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

Sortiere Elemente aufsteigend und rotiere entsprechend.



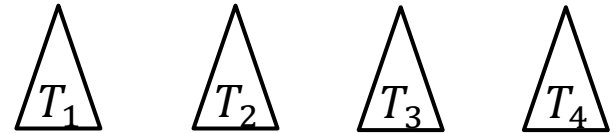
AVL-Bäume – Restructure

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?

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

Sortiere Elemente aufsteigend und rotiere entsprechend.



AVL-Bäume – Restructure

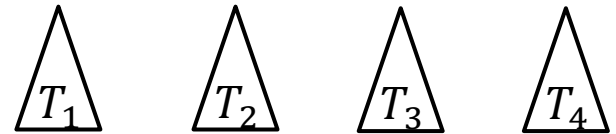
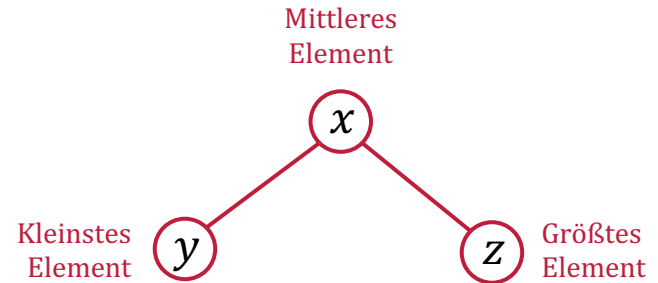
Bei Insert und Delete stellen sich nun folgende Fragen:

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

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

Sortiere Elemente aufsteigend und rotiere entsprechend.

Die Teilbäume T_1, \dots, T_4 werden wieder an den neuen,
rotierten Baum angehängt.



AVL-Bäume – Restructure

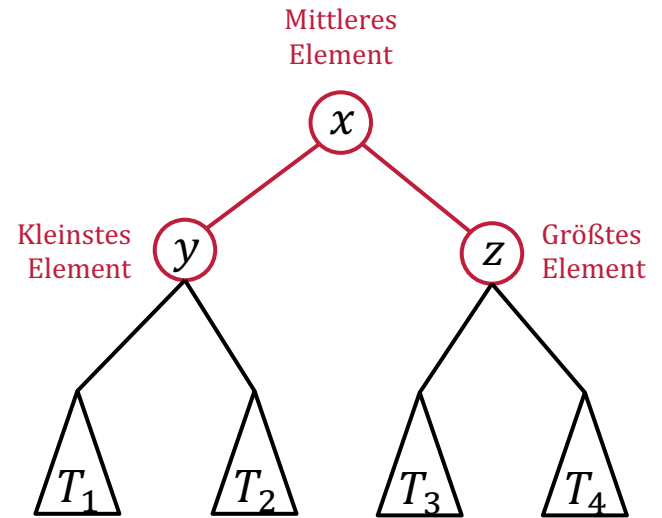
Bei Insert und Delete stellen sich nun folgende Fragen:

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

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

Sortiere Elemente aufsteigend und rotiere entsprechend.

Die Teilbäume T_1, \dots, T_4 werden wieder an den neuen,
rotierten Baum angehängt.



AVL-Bäume – Restructure

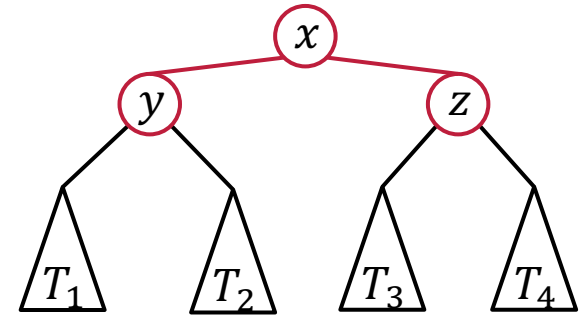
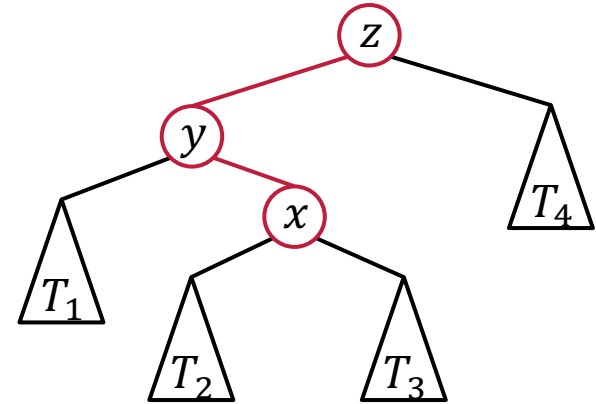
Bei Insert und Delete stellen sich nun folgende Fragen:

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

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

Sortiere Elemente aufsteigend und rotiere entsprechend.

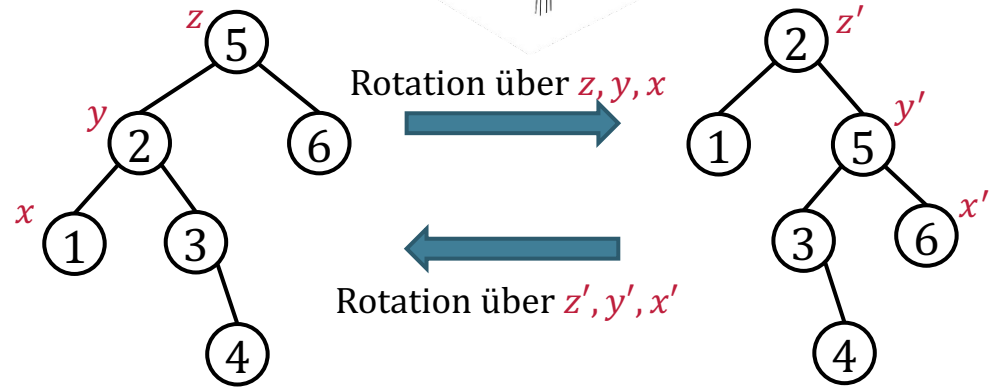
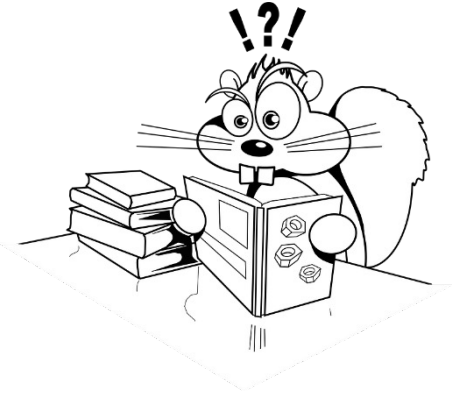
Die Teilbäume T_1, \dots, T_4 werden wieder an den neuen,
rotierten Baum angehängt.



AVL-Bäume – Restructure

Bei Insert und Delete stellen sich nun folgende Fragen:

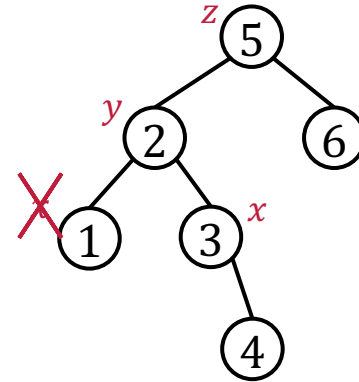
1. Welche Knoten werden unbalanciert?
2. Wie stellt man die Balance wieder her?
3. Welche Regeln sollte man berücksichtigen?



AVL-Bäume – Restructure

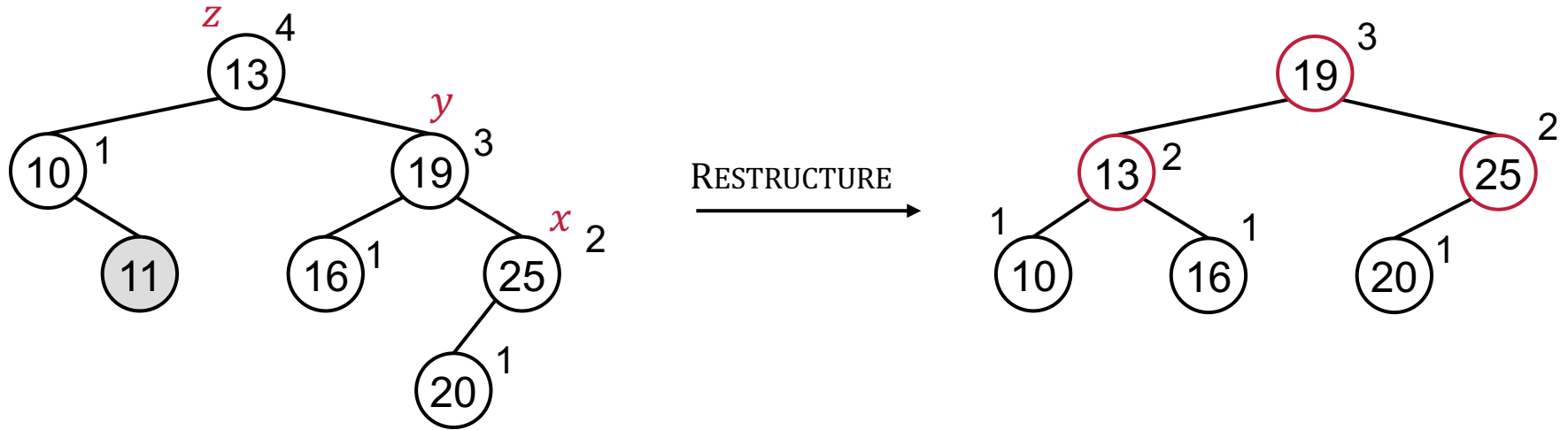
Bei Insert und Delete stellen sich nun folgende Fragen:

1. Welche Knoten werden unbalanciert?
 2. Wie stellt man die Balance wieder her?
 3. Welche Regeln sollte man berücksichtigen?
-
1. Starte bei tiefstem unbalanciertem Knoten: Das ist z .
 2. Wähle Kinder (x, y) nach deren Höhe aus.



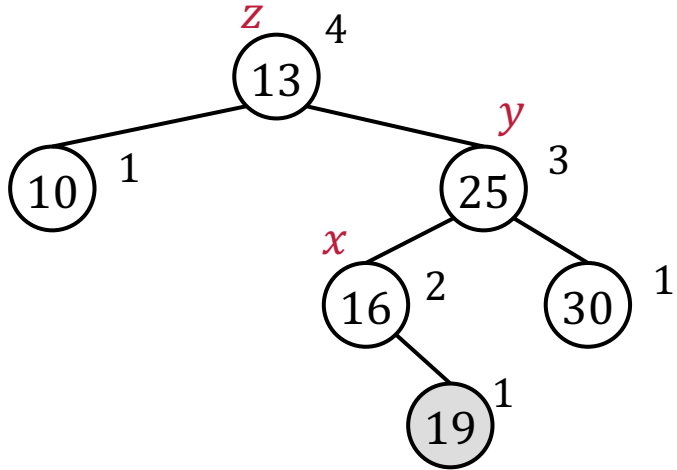
AVL-Bäume – Beispiele

DELETE($T, 11$)

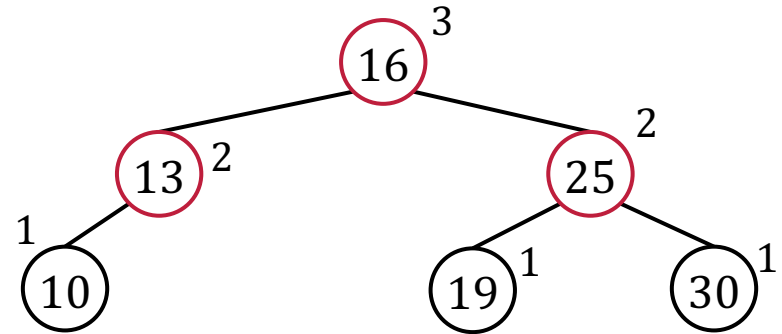


AVL-Bäume – Beispiele

INSERT($T, 19$)

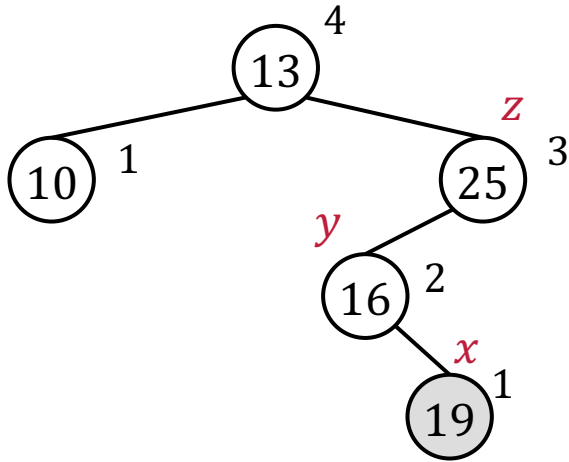


RESTRUCTURE

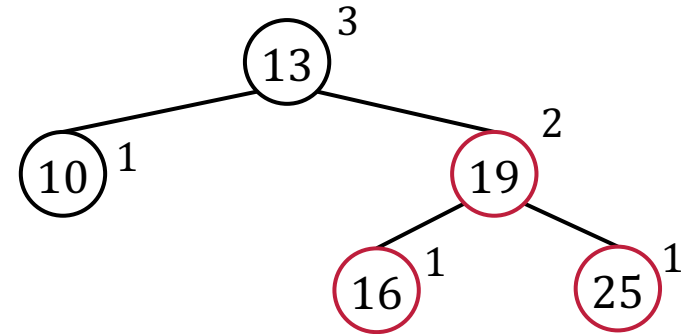


AVL-Bäume – Beispiele

INSERT($T, 19$)



RESTRUCTURE
→



AVL Restructure - Zusammenfassung

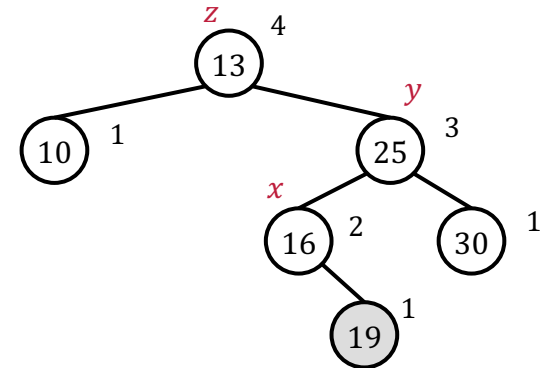
Ist nach INSERT oder DELETE die AVL-Eigenschaft verletzt, kann diese mit RESTRUCTURE wiederhergestellt werden.

Nach INSERT reicht ein RESTRUCTURE-Aufruf, um die Höhenbalanciertheit wiederherzustellen.

Nach DELETE können mehrere (genauer gesagt $O(\log n)$ viele) Aufrufe von RESTRUCTURE nötig sein.

RESTRUCTURE verlangt drei Knoten **x**, **y** und **z** als Argumente:

- Starte beim tiefsten unbalancierten Knoten: Das ist **z**.
- Das Kind von **z** mit größerer Höhe ist **y**.
- Das Kind von **y** mit größerer Höhe ist **x**.



Fragen?

Nächstes Mal:

Das Master-Theorem

Satz: Sei $T: \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion der Form

$$T(n) = \sum_{i=1}^m T(\alpha_i \cdot n) + \theta(n^k),$$

wobei $m \in \mathbb{N}$, $k \in \mathbb{R}$ und $0 < \alpha_i < 1$ für alle $i \in \{1, \dots, m\}$.
Dann gilt

$$T(n) \in \begin{cases} \theta(n^k), & \text{falls } \sum_{i=1}^m \alpha_i^k < 1 \\ \theta(n^k \log n), & \text{falls } \sum_{i=1}^m \alpha_i^k = 1 \\ \theta(n^c) \text{ mit } \sum_{i=1}^m \alpha_i^c = 1, & \text{falls } \sum_{i=1}^m \alpha_i^c > 1 \end{cases}$$



Arne Schmidt | Mergesort, Master-Theorem, Heapsort | Seite 13



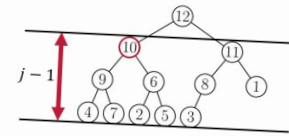
Algorithmen und Datenstrukturen - Übung #5

Mergesort, Master-Theorem, Heapsort

Ramin & Chek-Manh

Build-Max-Heap

```
1: function MAX-HEAPIFY(A, i)
2:   ℓ := links(i), r := rechts(i)
3:   if ℓ ≤ heap-größe[A] und A[ℓ] > A[i] then
4:     max := ℓ
5:   else
6:     max := i
7:   if r ≤ heap-größe[A] und A[r] > A[max] then
8:     max := r
9:   if max ≠ i then
10:    Vertausche A[max] und A[i]
11:    MAX-HEAPIFY(A, max)
```



Für Knoten auf Ebene $h - j$ benötigen wir maximal $O(j)$ Durchgänge.

Jeder Durchgang kostet $O(1)$



Arne Schmidt | Mergesort, Master-Theorem, Heapsort | Seite 28

... schon nächste Woche! 🤖

