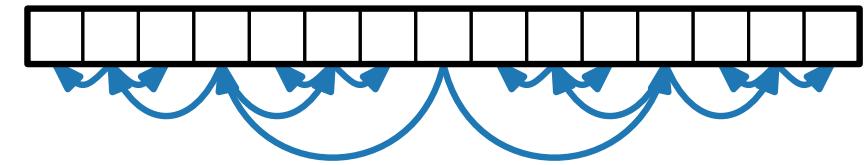
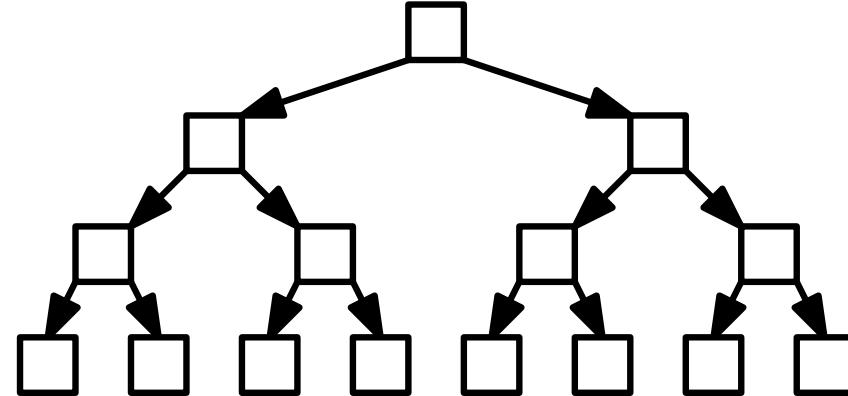




Algorithmen und Datenstrukturen

Vorlesung 13: Binäre Suchbäume



Zwischentest II: 12.12.2024, 8:15–10:00

- Zufallsexperimente, (Indikator-) Zufallsvariable, Erwartungswert
- (Randomisiertes) QuickSort
- Untere Schranke für WC-Laufzeit von vergleichsbasierten Sortierverfahren
- Linearzeit-Sortierverfahren
- Auswahlproblem (Median): randomisiert & deterministisch
- Elementare Datenstrukturen
- Hashing
- Binäre Suchbäume (Rot-Schwarz-Bäume noch nicht)

Dynamische Menge



Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M , wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation

`ptr INSERT(key k, value v)`
`DELETE(ptr x)`

`ptr SEARCH(key k)` Wörterbuch

`ptr MINIMUM()`

`ptr MAXIMUM()`

`ptr PREDECESSOR(ptr x)`

`ptr SUCCESSOR(ptr x)`

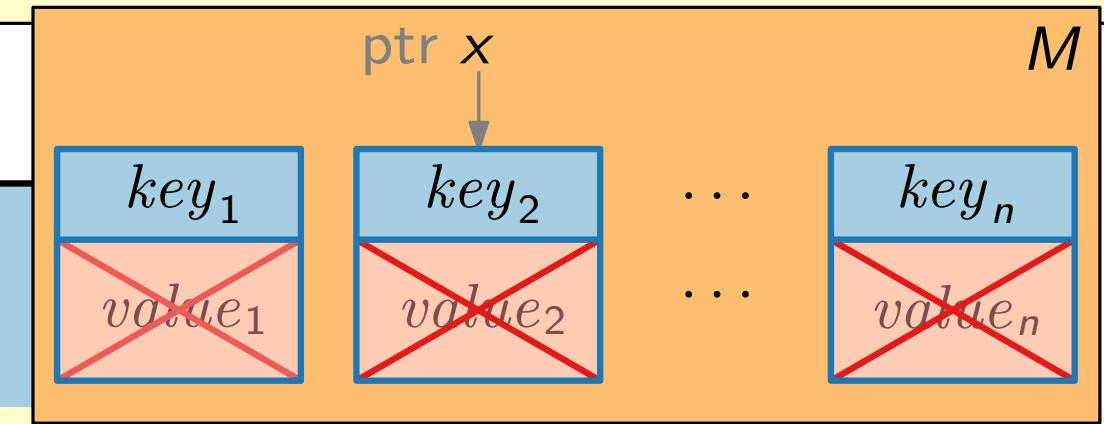
Pointer

Funktionalität

Veränderungen

Anfragen

`ptr x`

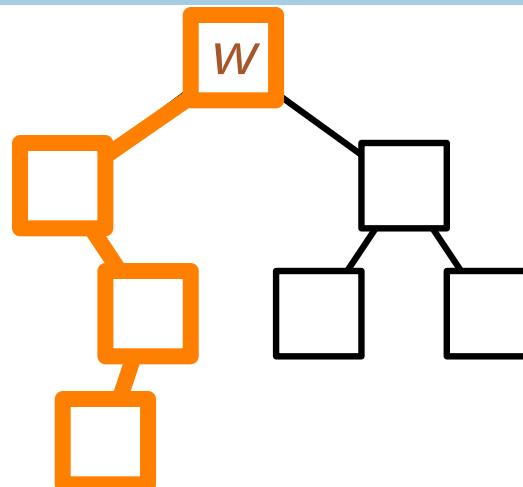


Vereinfachung

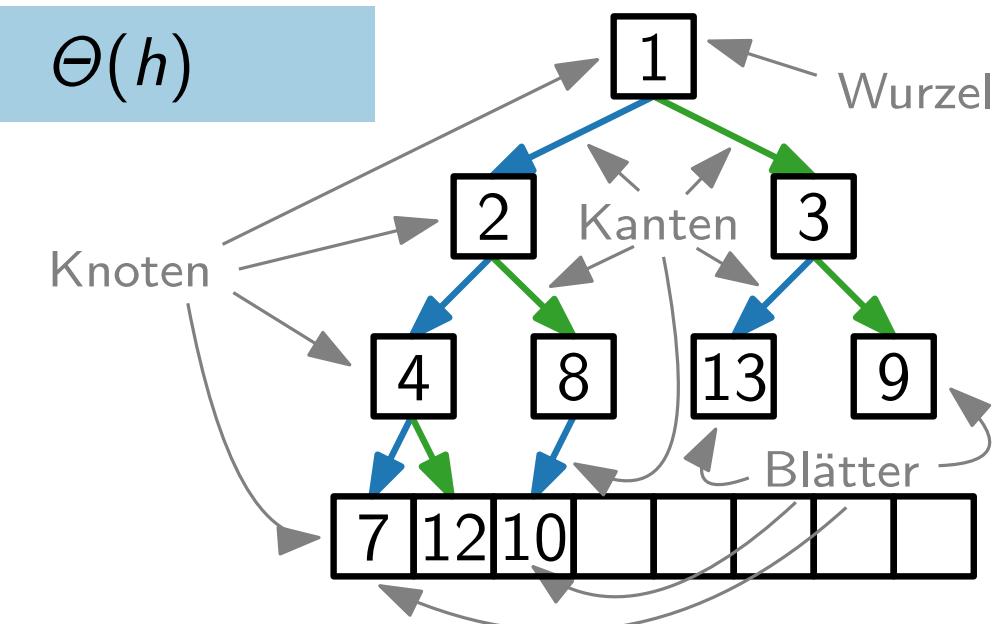
Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	–	–
MINHEAP	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



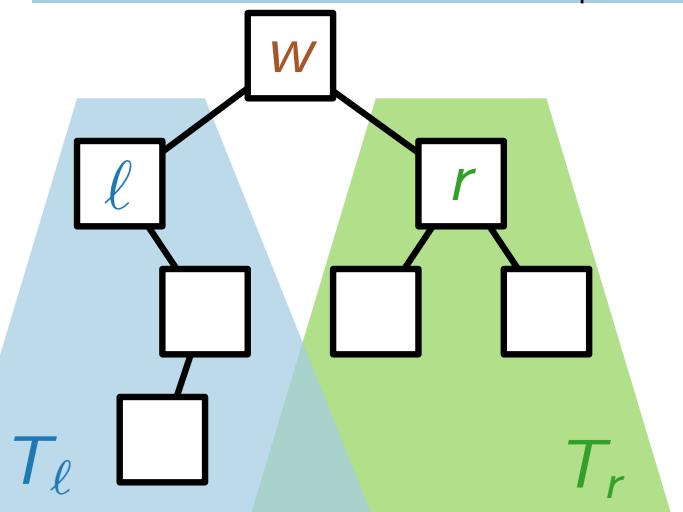
$h(T) =$ Höhe des Baums T
 $=$ Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad



Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

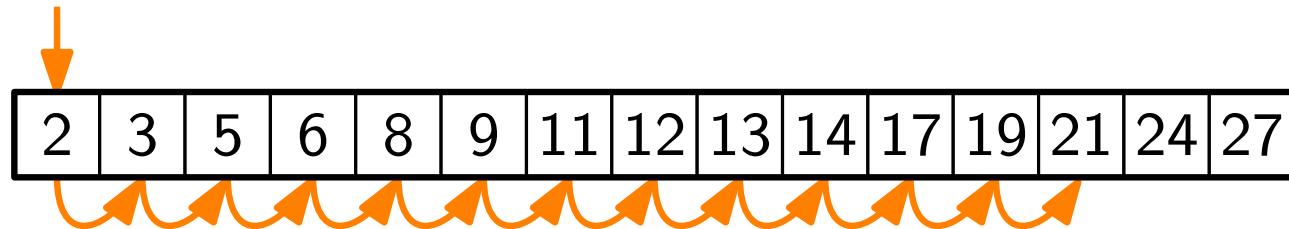
* unter bestimmten Annahmen.



$h(T) =$ Höhe des Baums T
 $=$ Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad

$$= \begin{cases} 0 & \text{falls Baum = Blatt} \\ 1 + \max\{h(T_l), h(T_r)\} & \text{sonst.} \end{cases}$$

Suche im sortierten Feld



SEARCH(21)

hier

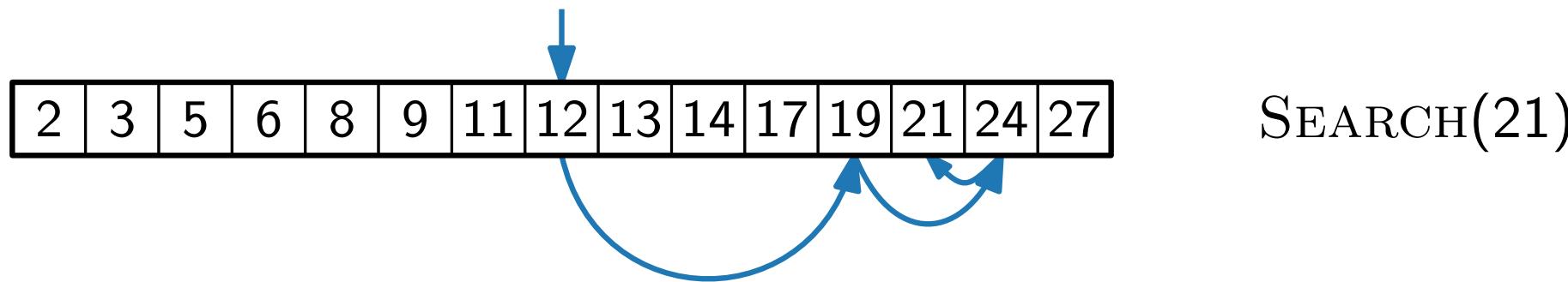
im Worst Case

Lineare Suche: 13

n

Schritte

Suche im sortierten Feld



Lineare Suche:	hier 13	im Worst Case n	Schritte
-----------------------	------------	----------------------	----------

Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte
----------------------	---	-------------------------------	----------

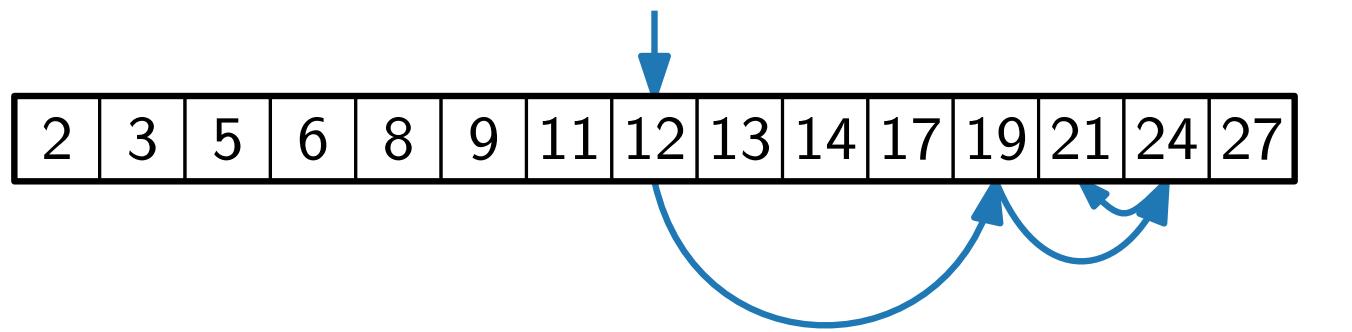
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

groß: Wie oft muss ich n halbieren, bis ich bei 1 bin?

$$\text{genau: } T(n) \leq T(\lfloor n/2 \rfloor) + 1 \text{ und } T(1) = 1$$

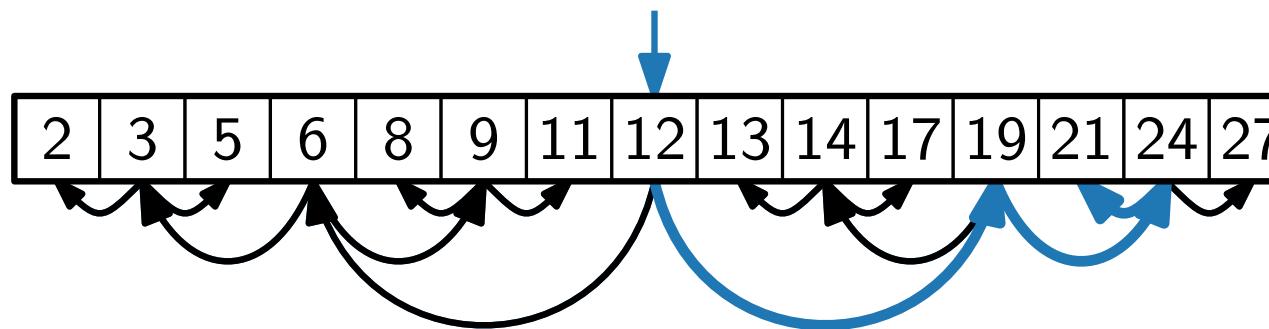
$$\begin{aligned} &\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}_{\lfloor \log_2 n \rfloor} \\ &= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2(n + 1) \rceil \end{aligned}$$

Suche im sortierten Feld

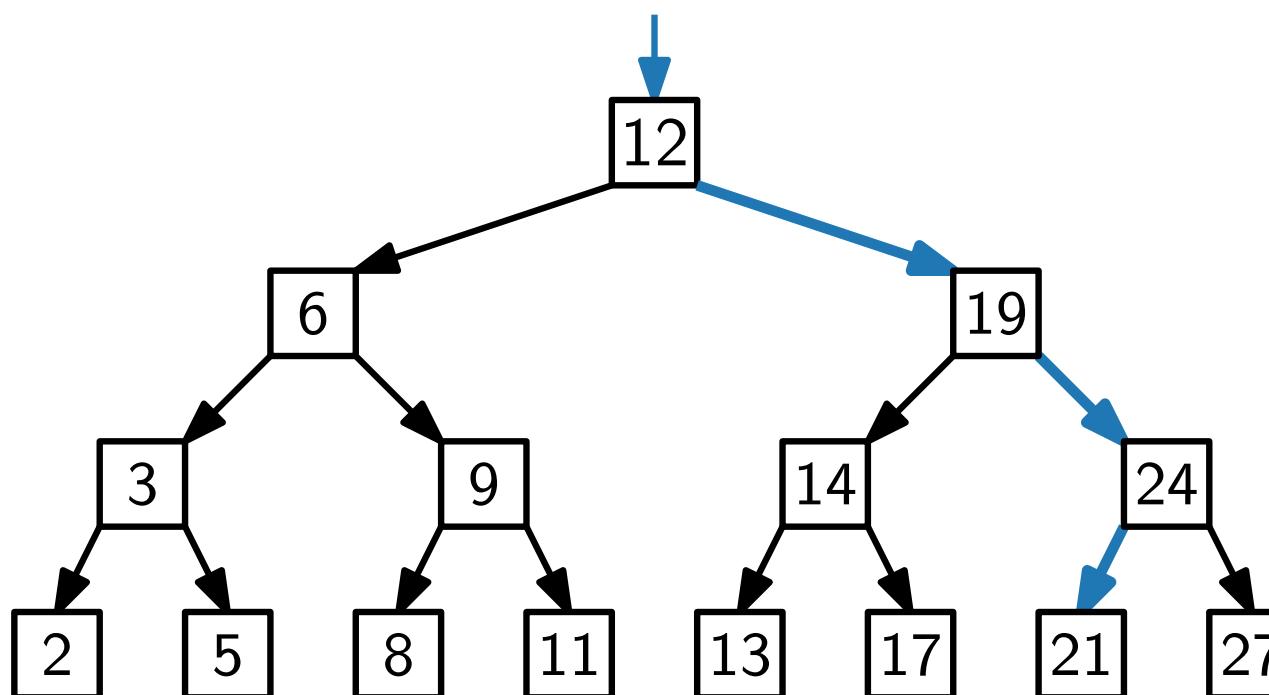


	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

Suche im sortierten Feld



SEARCH(21)



Binärer Suchbaum

Jeder Knoten hat höchstens zwei Kinder.

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

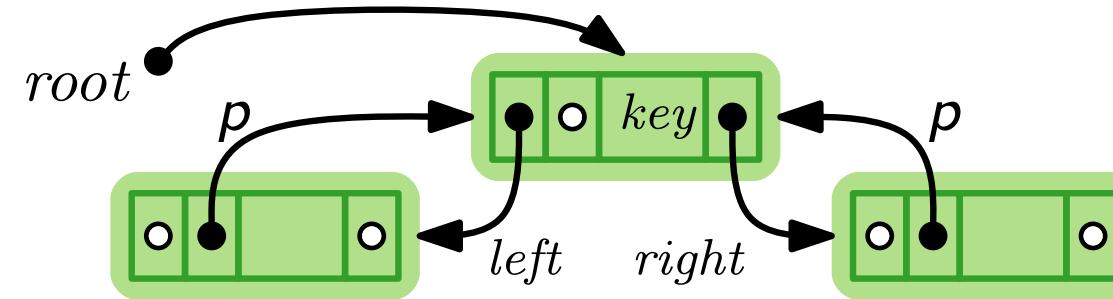
Binärer-Suchbaum-Eigenschaft.

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
rechten

Für jeden Knoten v gilt:

$$\begin{matrix} \leq \\ \geq \end{matrix}$$

Binärer Suchbaum



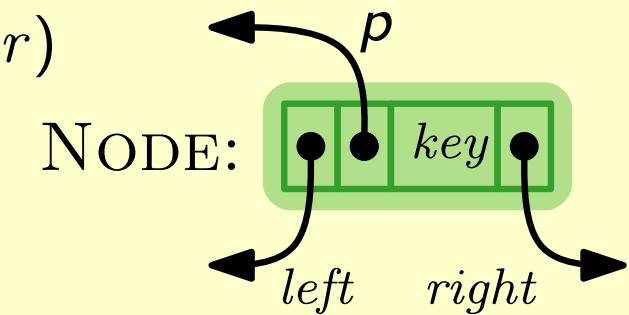
Operation

BINSEARCHTREE()

Implementierung

$root = \text{nil}$

NODE(key k , NODE par)
 $key = k$
 $p = par$
 $right = left = \text{nil}$



Node SEARCH(key k)

Node INSERT(key k)

DELETE(Node x)

Node MINIMUM()

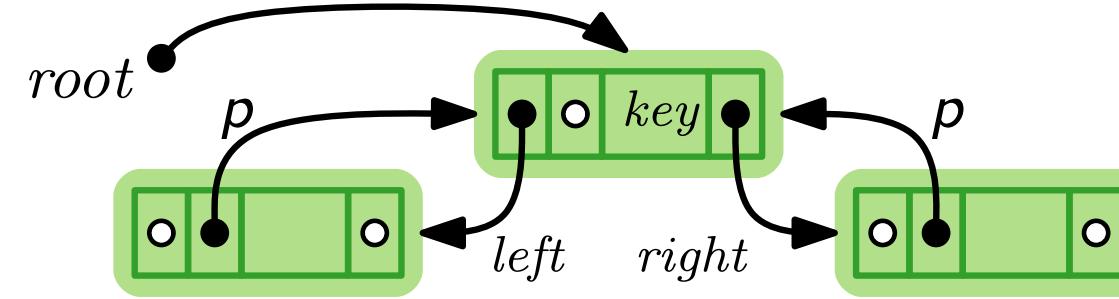
Node MAXIMUM()

Node PREDECESSOR(Node x)

Node SUCCESSOR(Node x)

TODO

Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

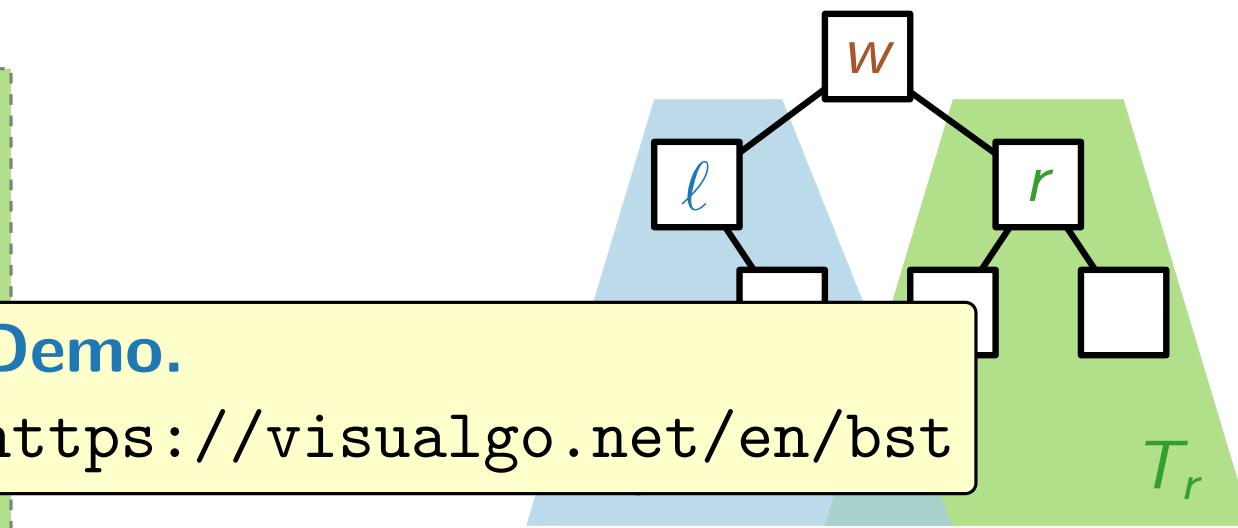
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ nil then
    INORDERTREEWALK(x.left)
    gib x.key aus
    INORDERTREEWALK(x.right)
```

Demo.

<https://visualgo.net/en/bst>



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $\text{root} = \text{nil}$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. rekursive Def. der Höhe

Also werden *ihre* Schlüssel sortiert ausgegeben.

Binärer-Suchbaum-Eigenschaft ⇒

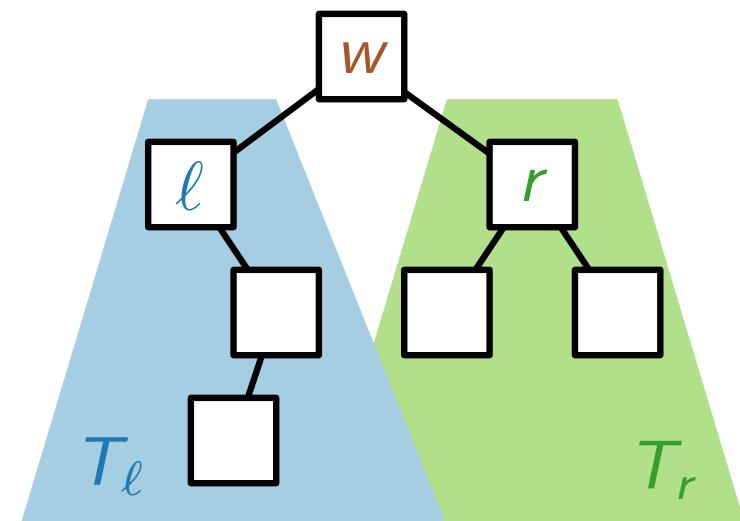
Ausgabe (sortierte Schlüssel von T_{links} , dann $\text{root}.key$,
dann sortierte Schlüssel von T_{rechts}) ist sortiert. ✓

Code:

INORDERTREEWALK(Node $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
  
```



Laufzeit

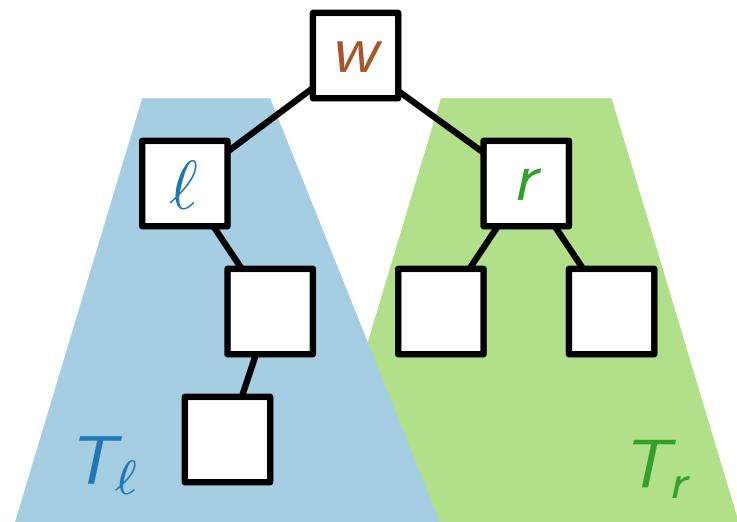
Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

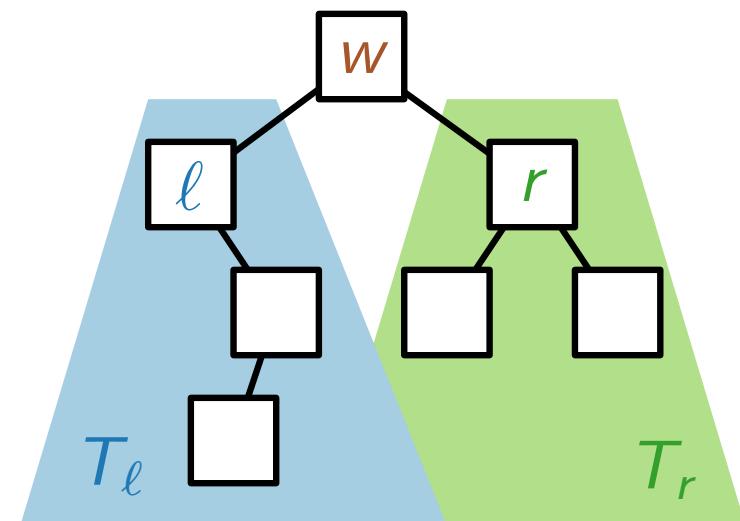
Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

Beweis durch
Induktion

Code:

INORDERTREEWALK(Node $x = \text{root}$)

```
if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

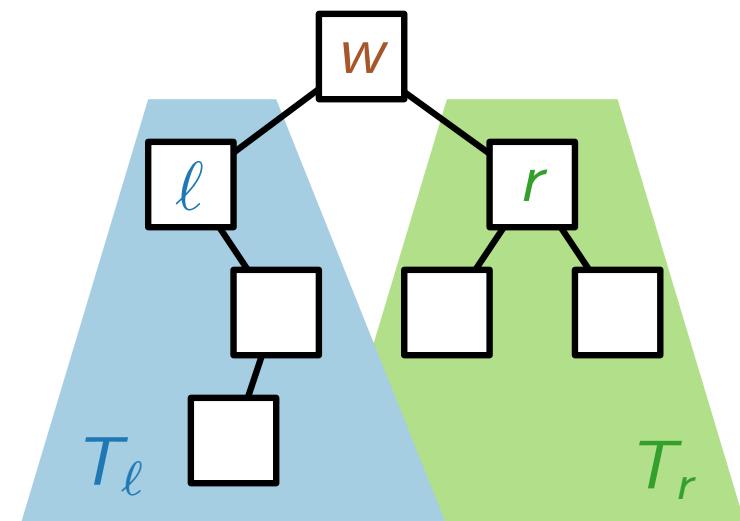
Für Bäume gilt: #Kanten = #Knoten - 1 = n - 1

$$\Rightarrow T(n) = c_1 \cdot (n - 1) + c_2 \cdot n \in \mathcal{O}(n).$$

Code:

```
INORDERTREEWALK(Node x = root)
```

```
if x ≠ nil then
    INORDERTREEWALK(x.left)
    gib x.key aus
    INORDERTREEWALK(x.right)
```



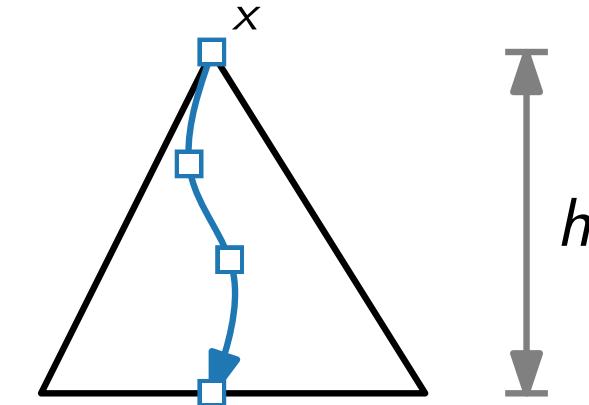
Suche

rekursiv

```
Node SEARCH(key k, Node x = root)
  if x == nil or x.key == k then
    return x
  if k < x.key then
    return SEARCH(k, x.left)
  else
    return SEARCH(k, x.right)
```

iterativ

```
while x ≠ nil and x.key ≠ k do
  if k < x.key then
    x = x.left
  else
    x = x.right
return x
```



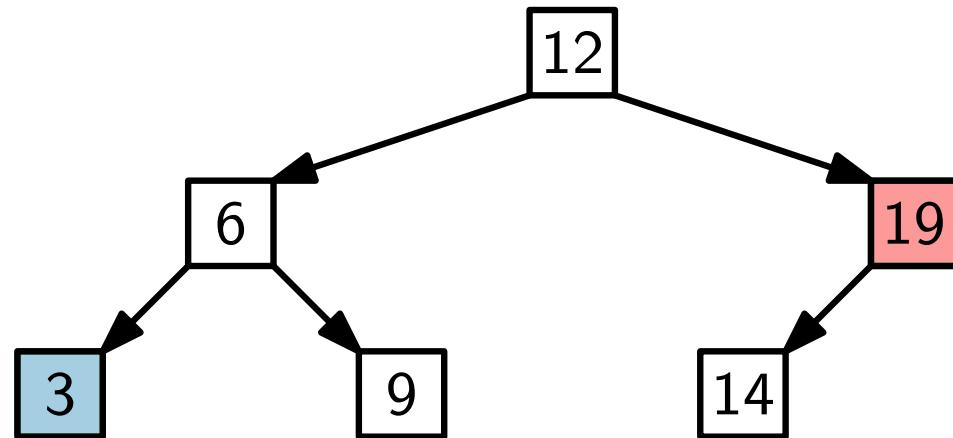
Laufzeit: $\mathcal{O}(h)$

Laufzeit: $\mathcal{O}(h)$

Trotzdem schneller, da
keine Verwaltung der
rekursiven
Methodenaufrufe.

Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Antwort: Min steht ganz links, Max ganz rechts!

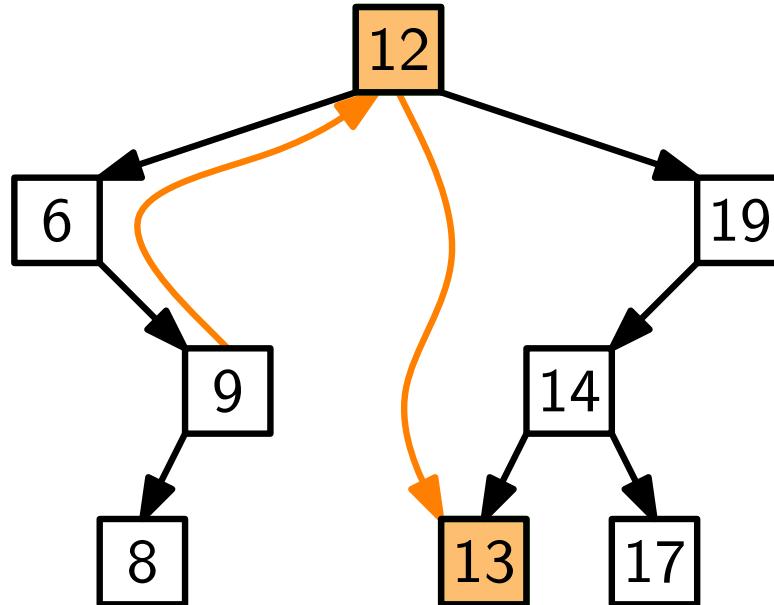
iterativ

```
Node MINIMUM(Node x = root)
  if x == nil then return nil
  while x.left ≠ nil do
    x = x.left
  return x
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel}$
 unter allen y mit $y.key > x.key$.
 $= \arg \min_y \{y.key \mid y.key > x.key\}.$



$$\text{Nachfolger}(19) = \text{nil}$$

$$\text{Nachfolger}(12) = 13 = \text{MINIMUM}(12.\text{right})$$

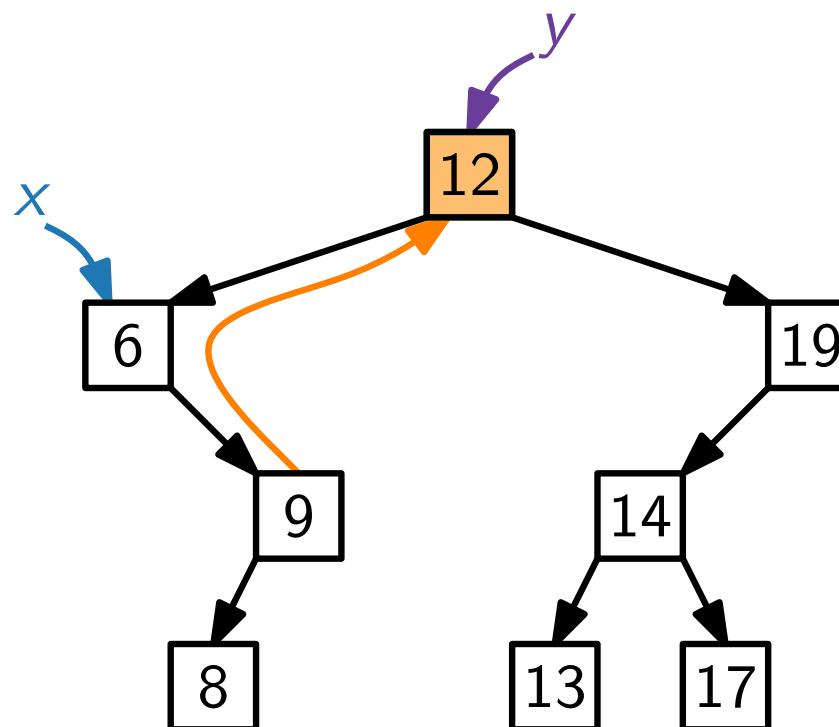
$$\text{Nachfolger}(9) = ?$$

9 hat kein rechtes Kind; 9 = $\text{MAXIMUM}(12.\text{left})$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel}$
 unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Tipp: Probieren Sie auch
z.B. $\text{SUCCESSOR}(19)!$

```

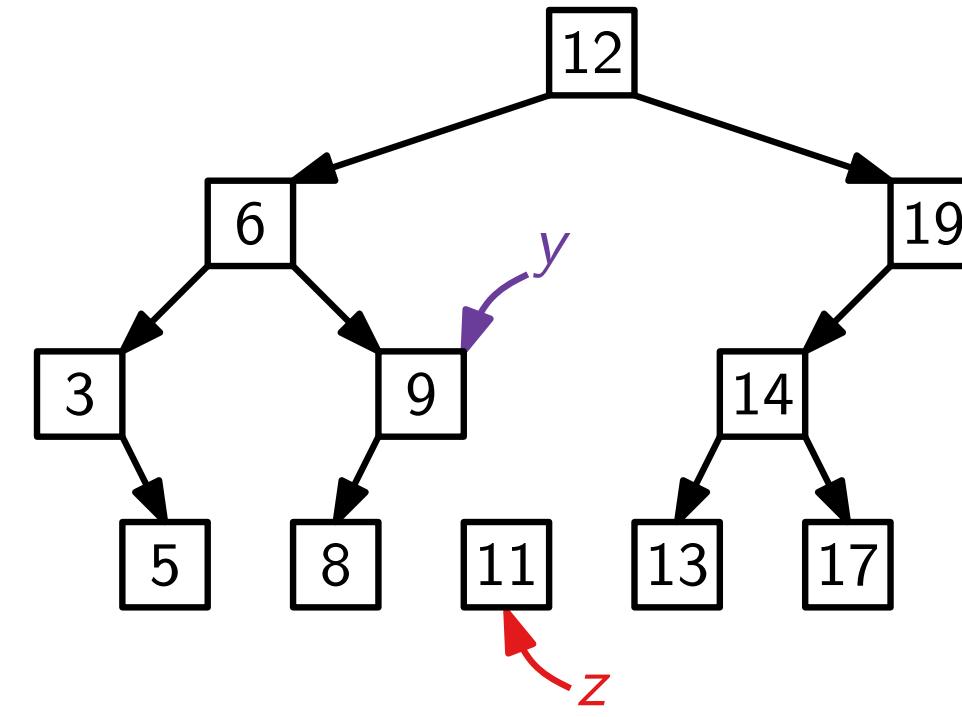
NODE SUCCESSOR(NODE x)
  if x.right ≠ nil then
    return MINIMUM(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
  return y
  
```

Einfügen

```

Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else
            x = x.right
    z = new Node(k, y)

```



INSERT(11)

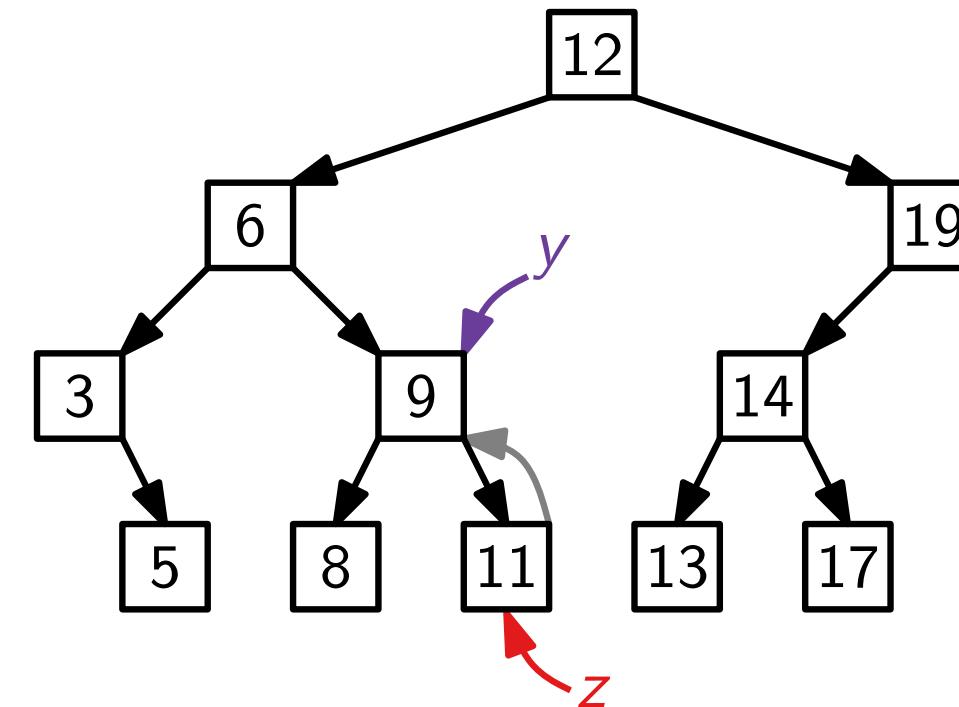
$x == \text{nil}$

Einfügen

```
Node INSERT(key k)
```

```

 $y = \text{nil}$ 
 $x = \text{root}$ 
while  $x \neq \text{nil}$  do
     $y = x$ 
    if  $k < x.\text{key}$  then
         $x = x.\text{left}$ 
    else  $x = x.\text{right}$ 
 $z = \text{new Node}(k, y)$ 
if  $y == \text{nil}$  then  $\text{root} = z$ 
else
    if  $k < y.\text{key}$  then  $y.\text{left} = z$ 
    else  $y.\text{right} = z$ 
return  $z$ 
```



INSERT(11)
 $x == \text{nil}$

Löschen

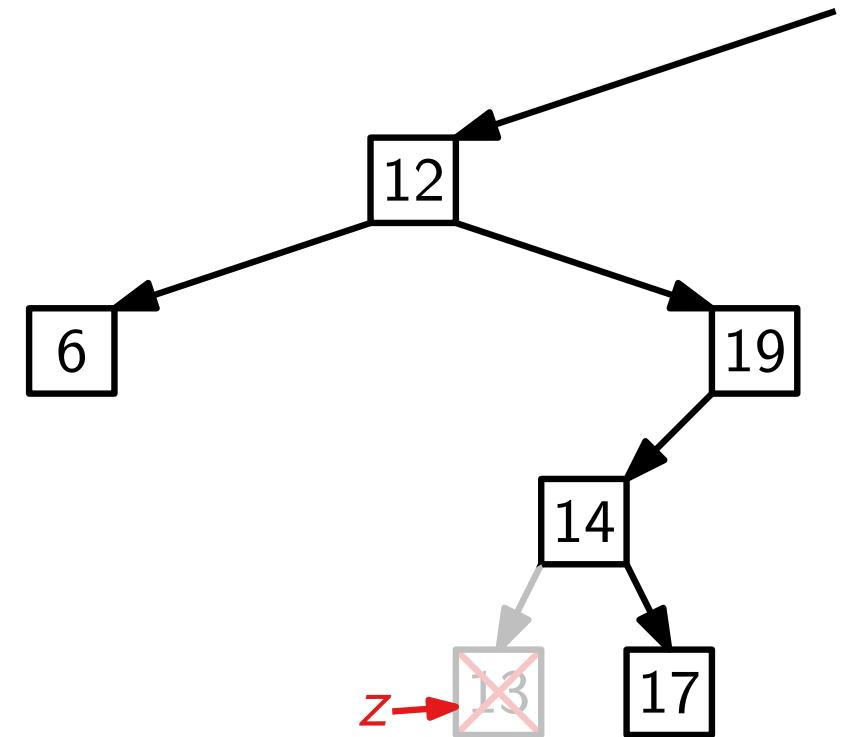
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

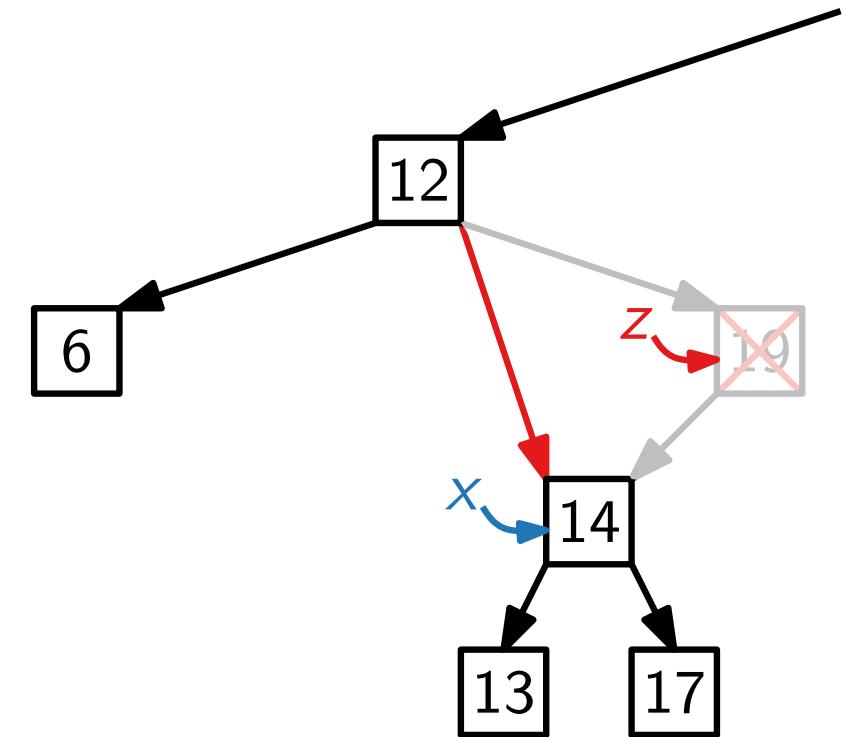
1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

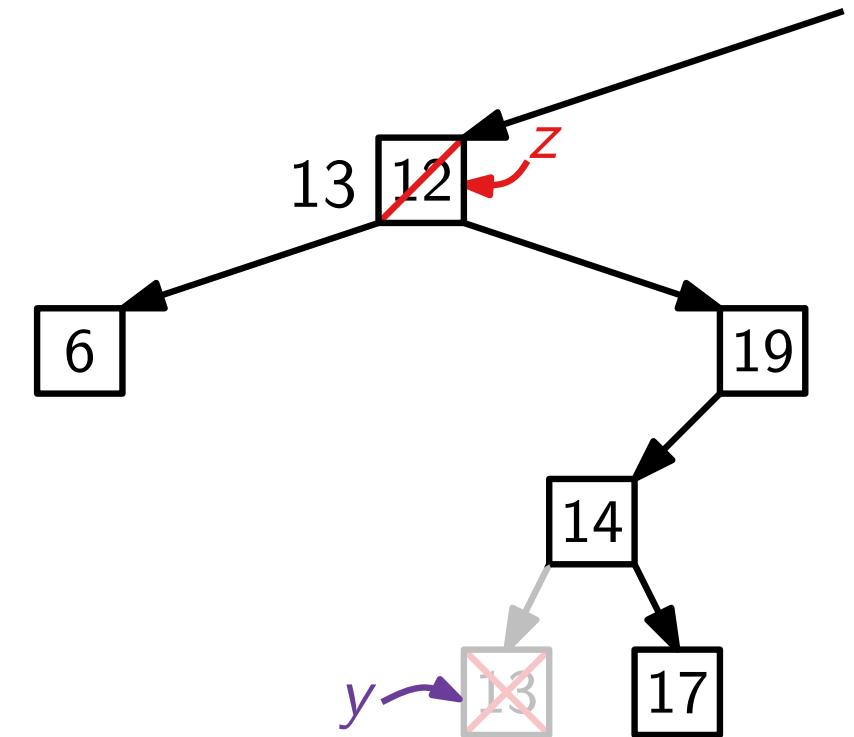
Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$ und $z.key = y.key$. Lösche y . (Fall 1 oder 2!)



Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume **balancieren** $\Rightarrow h \in \mathcal{O}(\log n)$

