Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

# Algorithmen und Datenstrukturen

Wintersemester 2022/23
13. Vorlesung

## Binäre Suchbäume

*Prof. Dr. Alexander Wolff*

*Lehrstuhl für Informatik I*

# Dynamische Menge

verwaltet Elemente einer sich ändernden Menge $M$

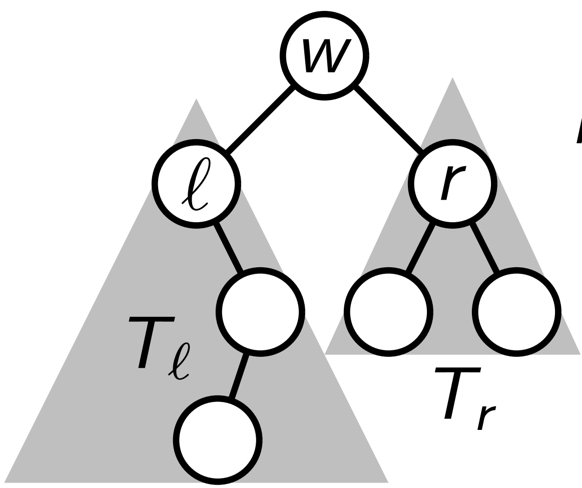
| **Abstrakter Datentyp** | *Funktionalität* | |
|---|---|---|
| ptr Insert(key $k$, info $i$) | Änderungen | Wörterbuch |
| Delete(ptr $x$) | | |
| ptr Search(key $k$) | Anfragen | |
| ptr Minimum() | | |
| ptr Maximum() | | |
| ptr Predecessor(ptr $x$) | | |
| ptr Successor(ptr $x$) | | |

**Implementierung:** je nachdem...

# Implementierung

$^{\star}$) *unter bestimmten Annahmen.*

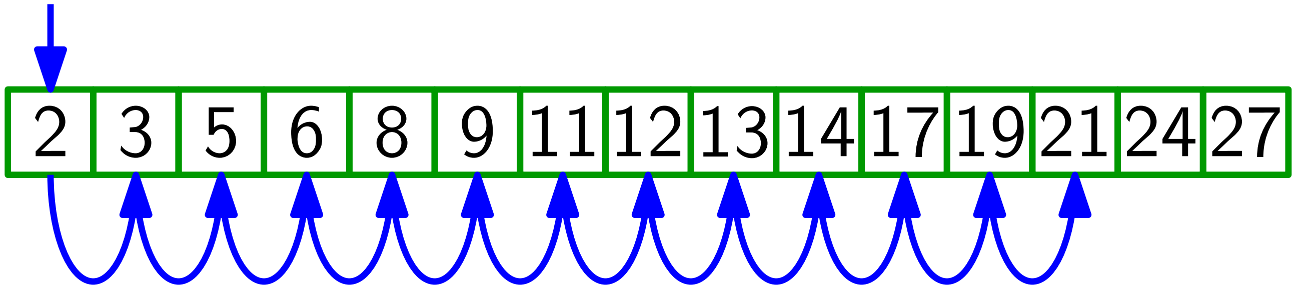
| | Search | Ins/Del | Min/Max | Pred/Succ |
|---|---|---|---|---|
| unsortierte Liste | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| unsortiertes Feld | $\Theta(n)$ | $\Theta(1)/\Theta(n)$ | $\Theta(1)^{\oplus}$ | $\Theta(n)$ |
| sortiertes Feld | **?** | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| Hashtabelle | $\Theta(1)^{\star}$ | $\Theta(1)^{\star}$ | – | – |
| Binärer Suchbaum | $\Theta(h)$ | $\Theta(h)$ | $\Theta(h)$ | $\Theta(h)$ |




$$
\begin{aligned}
h(T) &= \text{Höhe des Baums } T \\
&= \text{Anz. Kanten auf längstem Wurzel-Blatt-Pfad} \\
&= \begin{cases} 0 & \text{falls Baum} = \text{Blatt} \\ 1 + \max\{h(T_\ell), h(T_r)\} & \text{sonst.} \end{cases}
\end{aligned}
$$

$^{\oplus}$) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

# Suche im sortierten Feld




*Suche 21!*

| | *hier* | *im Worst Case* | |
|---|---|---|---|
| Lineare Suche: | 13 | $n$ | Schritte |

# Suche im sortierten Feld

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 8 | 9 | 11 | 12 | 13 | 14 | 17 | 19 | 21 | 24 | 27 |

*Suche 21!*

| | *hier* | *im Worst Case* | |
|---|---|---|---|
| Lineare Suche: | 13 | $n$ | Schritte |
| *Binäre Suche:* | 4 | **?** | Schritte* |

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

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

$$\le T(\lfloor n/4 \rfloor) + 1 + 1 \le \cdots \le T(1) + \underbrace{1 + \cdots + 1}_{\lfloor \log_2 n \rfloor}$$

*Übung!* $= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2 (n+1) \rceil$

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

# Suche im sortierten Feld

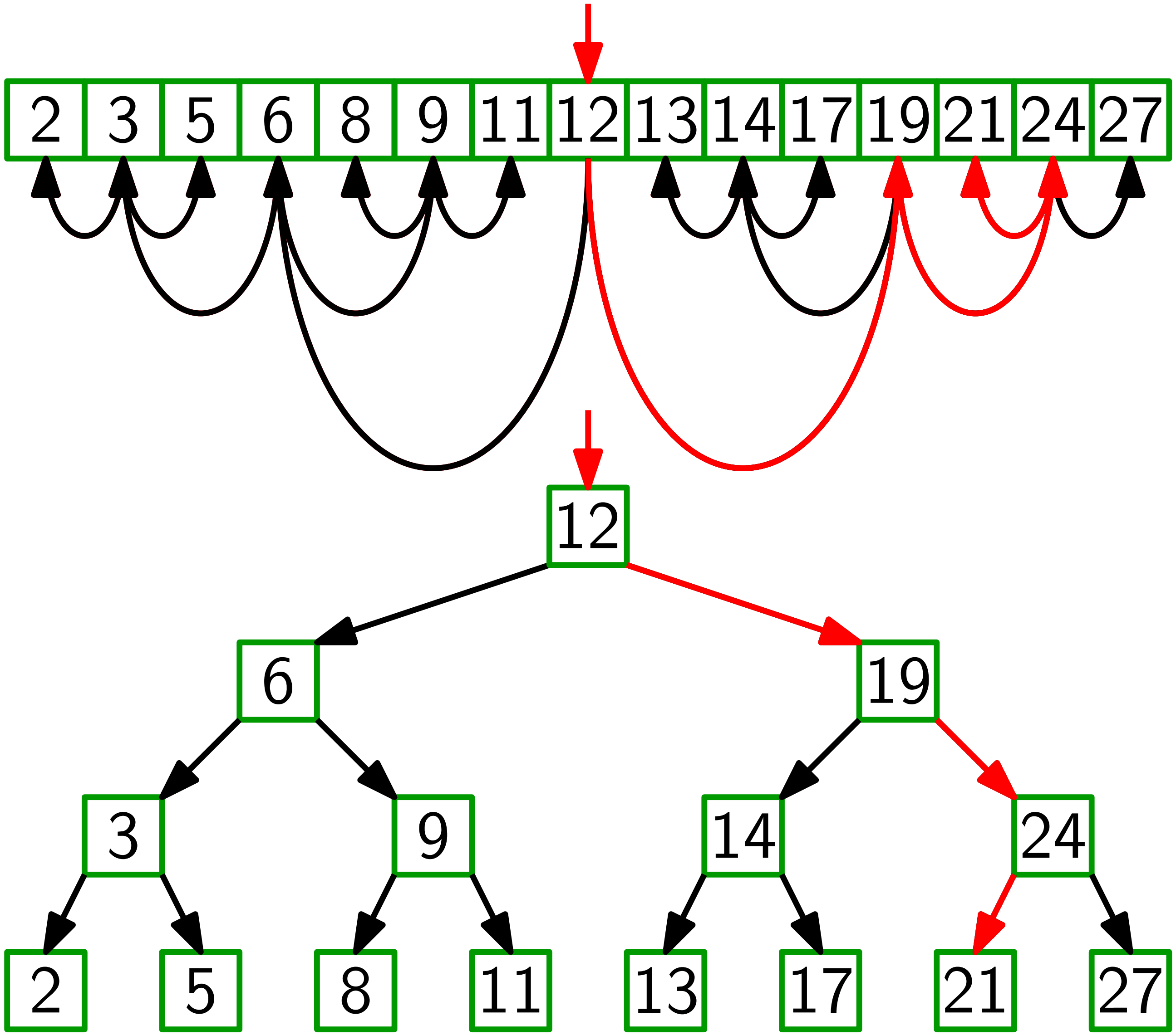
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 8 | 9 | 11 | 12 | 13 | 14 | 17 | 19 | 21 | 24 | 27 |

*Suche 21!*

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

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

# Suche im sortierten Feld

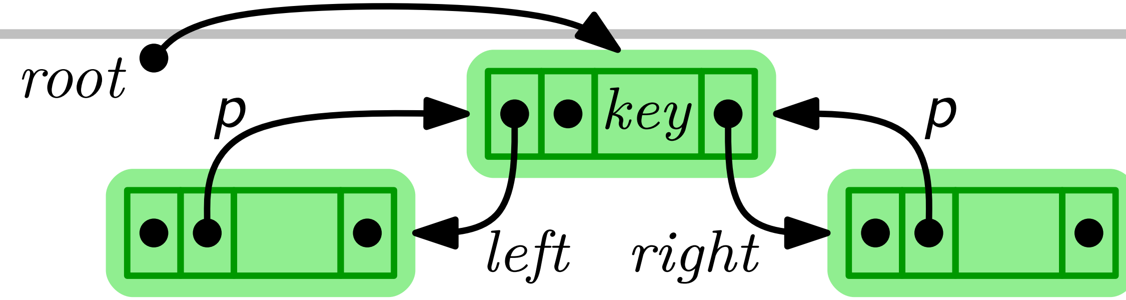



*Suche 21!*

## *Binärer Suchbaum*

*Binärer-Suchbaum-Eigenschaft:* Für jeden Knoten $v$ gilt:
alle Knoten im linken Teilbaum von $v$ haben Schlüssel $\leq v.key$
rechten $\geq$

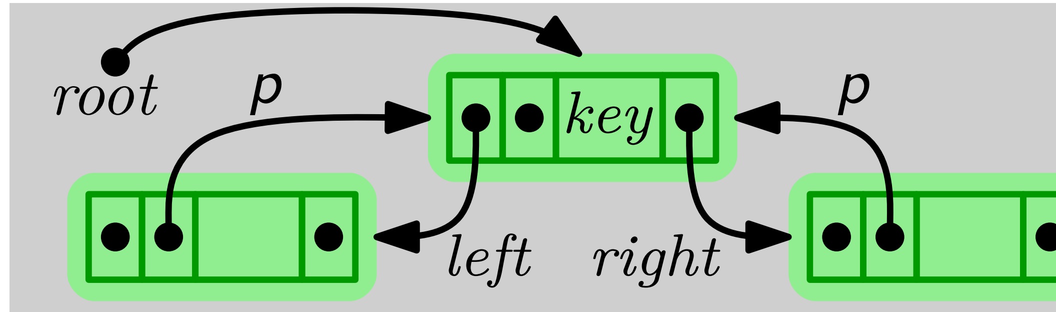
# Bin. Suchbaum




| **Abs. Datentyp** | **Implementierung** | | |
|---|---|---|---|
| BinSearchTree() | $root = nil$ | Node(key $k$, Node $par$)<br>$key = k$<br>$p = par$<br>$right = left = nil$<br>Node $root$ | Node<br>key $key$<br>Node $left$<br>Node $right$<br>Node $p$ |
| Node Search(key $k$) | | | |
| Node Insert(key $k$)<br>Delete(Node $x$) | | | |
| Node Minimum()<br>Node Maximum() | | | |
| Node Predecessor(Nd. $x$)<br>Node Successor(Node $x$) | | | |

To do!

# 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** $x \neq nil$ **then**
    InorderTreeWalk($x.left$)
    gib $x.key$ aus
    InorderTreeWalk($x.right$)

# Korrektheit

**zu zeigen:** Schlüssel werden in sortierter Rf. ausgegeben.
Induktion über die Baumhöhe $h$.

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien $T_{\text{links}}$ und $T_{\text{rechts}}$ li. & re. Teilbaum der Wurzel.
$T_{\text{links}}$ und $T_{\text{rechts}}$ haben Höhe $< h$. *[rekursive Def. der Höhe!]*
Also werden *ihre* Schlüssel sortiert ausgegeben.

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

**Code:**

```
InorderTreeWalk(Node x = root)
  if x ≠ 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) $T(n) \leq c \cdot n \; - \mathbf{1}$

**Code:**

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

# Laufzeit

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

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

*oder:* Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. von Schritten aus.

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

Übung: zeig's mit Induktion!

**Code:**

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

# Laufzeit

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

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n \; - \mathbf{1}$

*oder:* Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. von Schritten aus.

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

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

**Code:**

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

# Vorlesungsumfrage – jetzt!

Bitte suchen Sie die Email, die Sie von EvaSys bekommen haben und klicken Sie dort auf den Link zur Umfrage.

**4. Additional remarks**

4.1 What did you especially like in this course?

4.2 From your perspective, what could be improved? What do you criticize?

**Zum Beispiel:**
– Folien
– Übungsaufgaben
– Roter Faden?
– Beweise?
– Reaktion auf Fragen?
– Buch zur Vorlesung?
– Zwischentests
– Aufgaben in der VL
– . . .

All questions are optional. Should a question not be applicable to a course, you can leave the answer open.

## 1 Course as a whole

1.1 Please rate the course as a whole.

very good ○ ○ ○ ○ ○ insufficient

## 2 Lecture

2.1 The lecturer is well prepared and presents the material in a way that is easy to understand.

completely agree ○ ○ ○ ○ ○ do not agree at all

2.2 The supplied lecture materials (writings on the blackboard, presentation slides, videos, additional literature) are well prepared and improve my understanding of the course contents.

completely agree ○ ○ ○ ○ ○ do not agree at all

2.3 Portions of the lecture were held online (live stream, recorded lecture) and helped improve the lecture.

completely agree ○ ○ ○ ○ ○ do not agree at all | ○ no online parts

## 3 Exercises

3.1 The tutor is well prepared and presents the material in a way that is easy to understand.

completely agree ○ ○ ○ ○ ○ do not agree at all | ○ no exercises

3.2 The exercises are comprehensible and improve my understanding of the course contents.

completely agree ○ ○ ○ ○ ○ do not agree at all | ○ no exercises

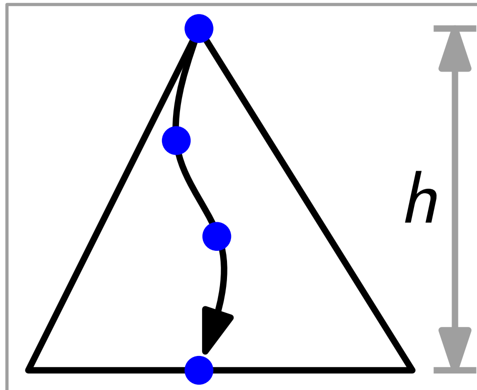
# Suche

**Aufgabe:** Schreiben Sie Pseudocode für die rekursive Methode

Node Search(key $k$, Node $x = root$)

rekursiv

**if** $x == nil$ **or** $x.key == k$ **then**
  **return** $x$
**if** $k < x.key$ **then**
  **return** Search($k, x.left$)
**else** **return** Search($k, x.right$)




Laufzeit: $O(h)$

iterativ

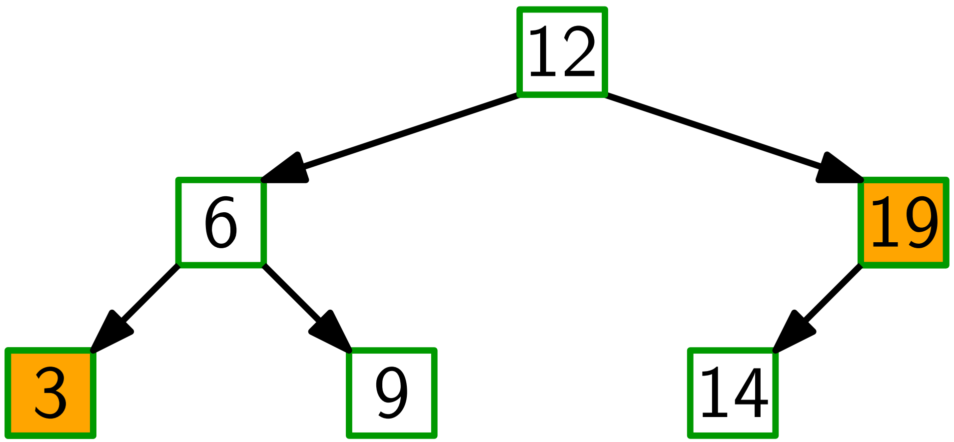
**while** $x \neq nil$ **and** $x.key \neq k$ **do**
  **if** $k < x.key$ **then**
    $x = x.left$
  **else** $x = x.right$
**return** $x$

Laufzeit: $O(h)$

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!

**Aufgabe:** Schreiben Sie für binäre Suchbäume die Methode

Node Minimum(Node $x = root$) — *iterativ!*

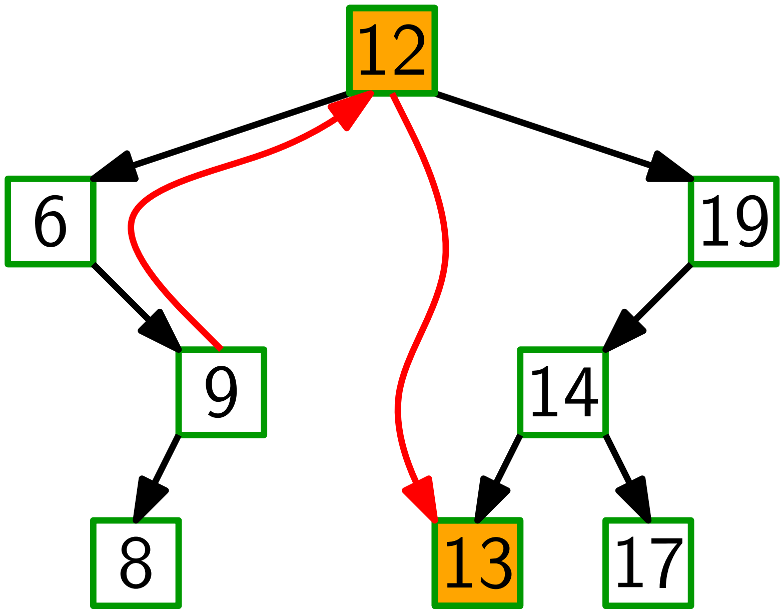
**if** $x == nil$ **then return** $nil$
**while** $x.left \neq nil$ **do**
  $x = x.left$
**return** $x$

# Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

**Erinnerung:** Nachfolger($x$) $=$ Knoten mit kleinstem Schlüssel unter allen $y$ mit $y.key > x.key$.

$= \arg\min_y \{y.key \mid y.key > x.key\}$.




Nachfolger(19) $:= nil$

Nachfolger(12) $=$ ?

Nachfolger(9) $=$ ?

13 $==$ Minimum(„12.$right$“)

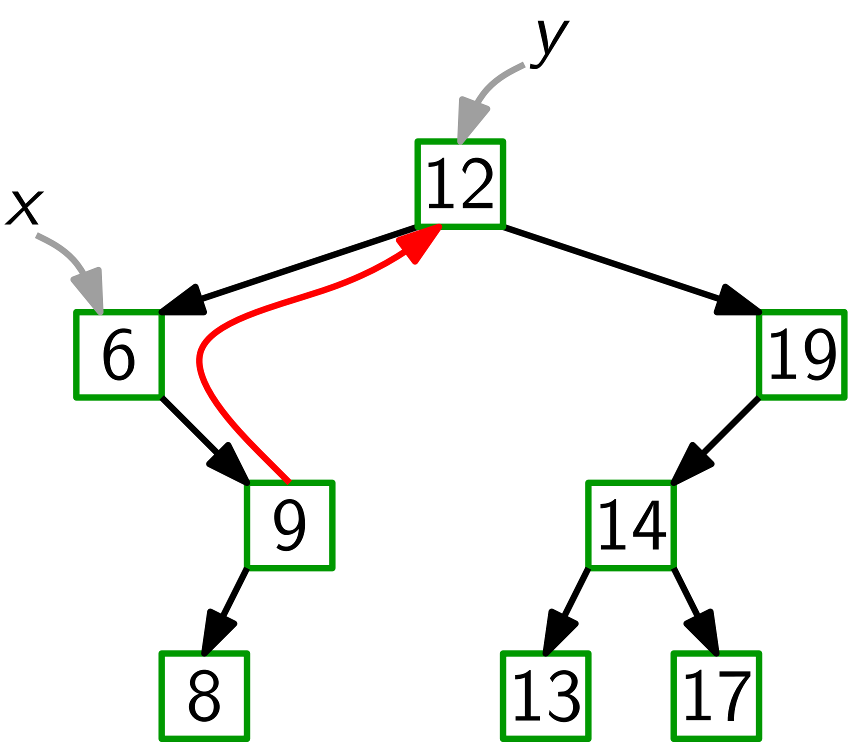
9 hat kein rechtes Kind; 9 $==$ Maximum(„12.$left$“)

# Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

**Erinnerung:** Nachfolger($x$) $=$ Knoten mit kleinstem Schlüssel unter allen $y$ mit $y.key > x.key$.

$= \arg\min_y \{y.key \mid y.key > x.key\}$.




*Tipp:* Probieren Sie auch z.B. Successor( "19" )!

Node Successor(Node $x$)

**if** $x.right \neq nil$ **then**
  **return** Minimum($x.right$)

$y = x.p$

**while** $y \neq nil$ **and** $x == y.right$ **do**
  $x = y$
  $y = y.p$

**return** $y$

# Einfügen

Node Insert(key $k$)

$y = nil$

$x = root$

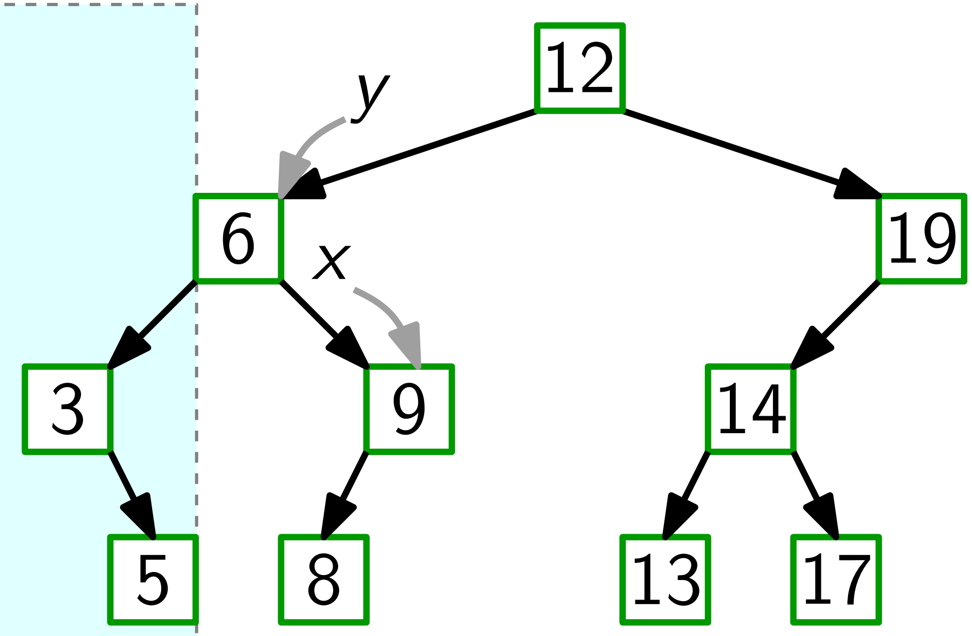
**while** $x \neq nil$ **do**

$y = x$

**if** $k < x.key$ **then**

$x = x.left$

**else** $x = x.right$




Insert(11)

# Einfügen

Node Insert(key $k$)

$y = nil$

$x = root$

**while** $x \neq nil$ **do**

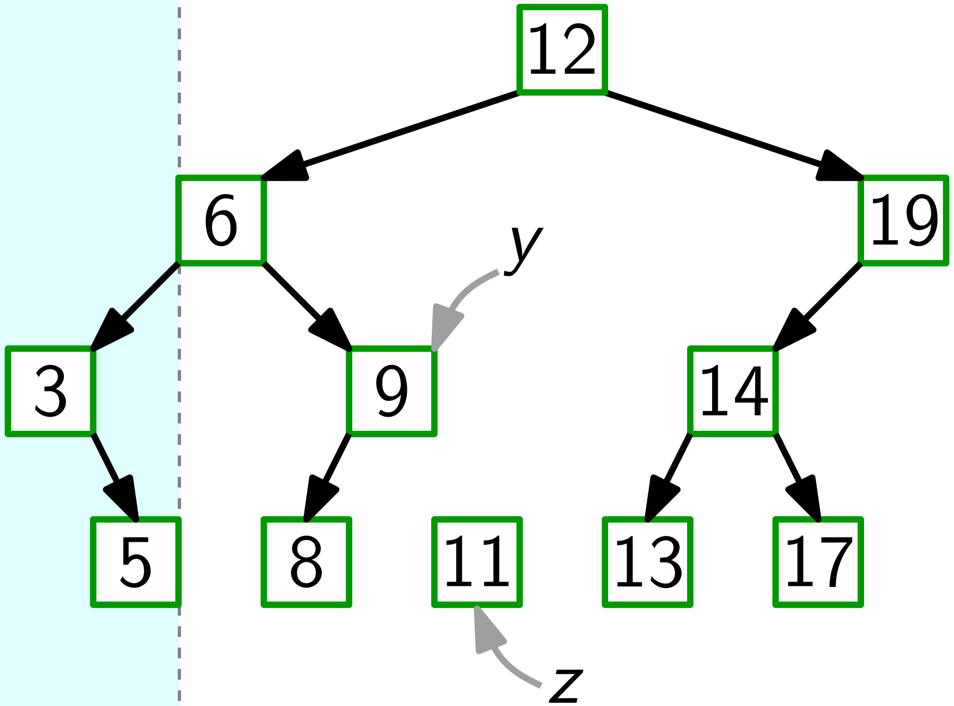
  $y = x$

  **if** $k < x.key$ **then**

    $x = x.left$

  **else** $x = x.right$

$z =$ new Node$(k, y)$

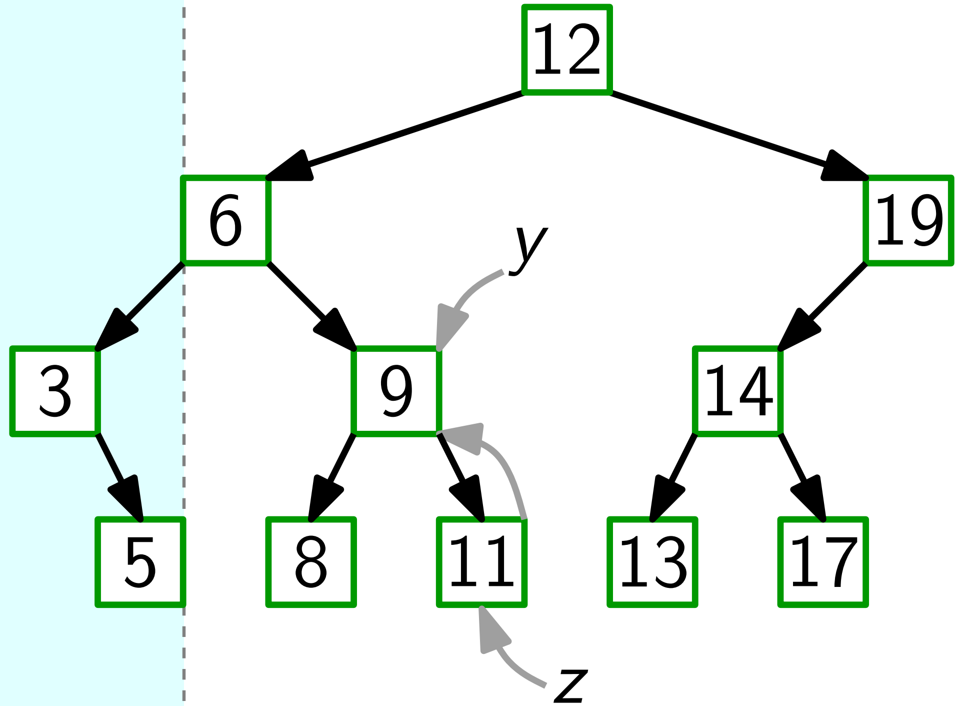



Insert(11)

$x == nil$

# 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)
    if y == nil then root = z
    else
        if k < y.key then y.left = z
        else              y.right = z
    return z
```

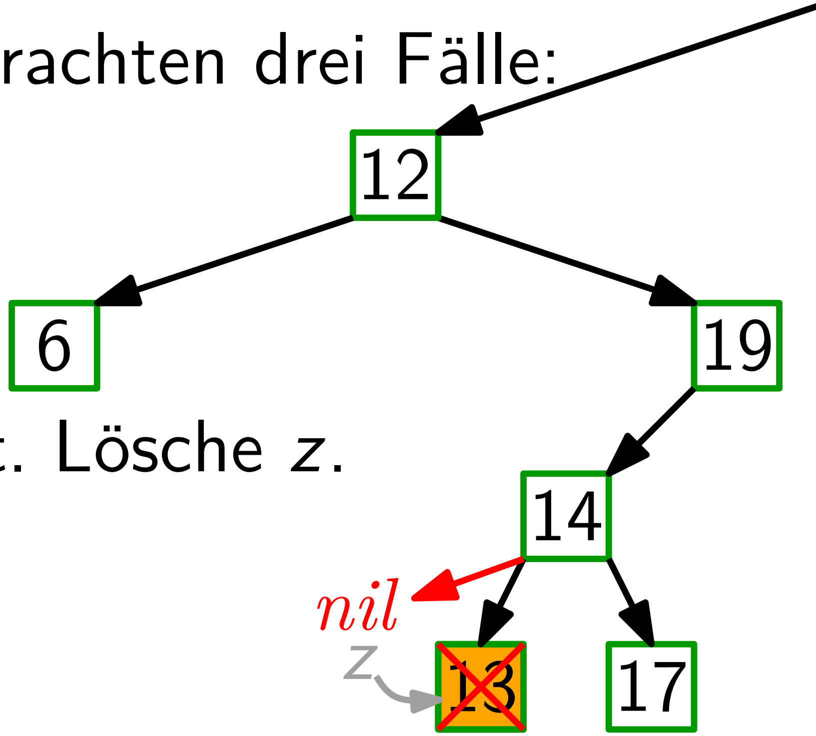



Insert(11)

$x == nil$

# Löschen

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




**1.** $z$ hat keine Kinder.

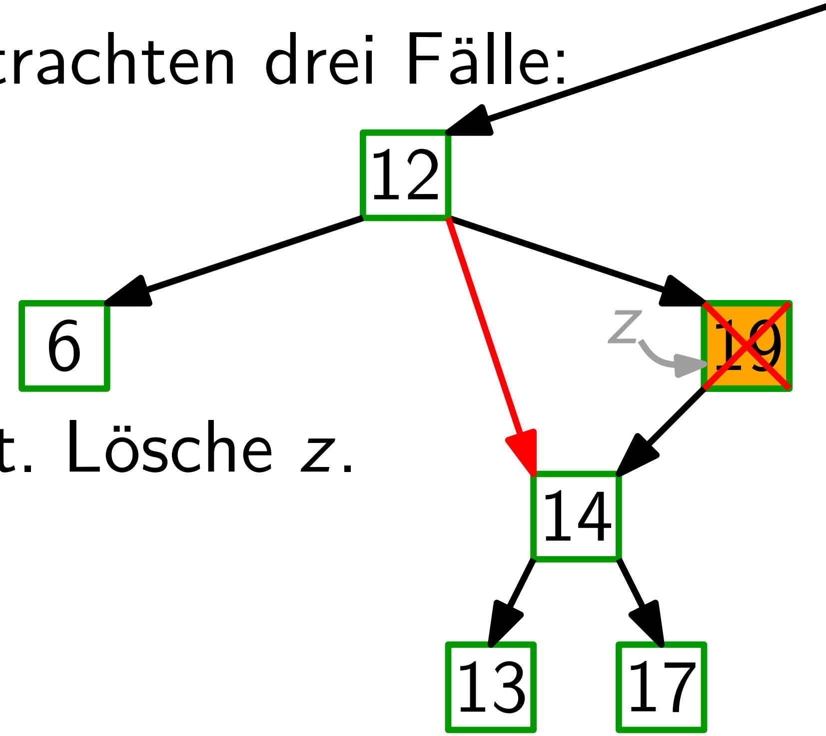
Falls $z$ linkes Kind von $z.p$ ist,
setze $z.p.\mathit{left} = \mathit{nil}$; sonst umgekehrt. Lösche $z$.

**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 $z$.

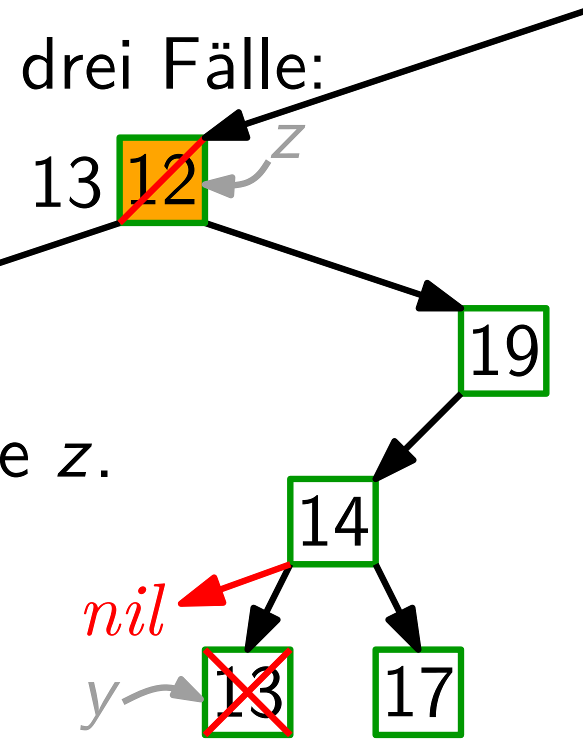
**2.** $z$ hat ein Kind $x$.

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 $O(h)$ Zeit, wobei $h$ die momentane Höhe des Baums ist.

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

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

**Achtung:** Die Vorlesung am Do, 8.12., fällt aus!