

## 8. Natürliche Suchbäume

---

[Ottman/Widmayer, Kap. 5.1, Cormen et al, Kap. 12.1 - 12.3]

Bäume sind

- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

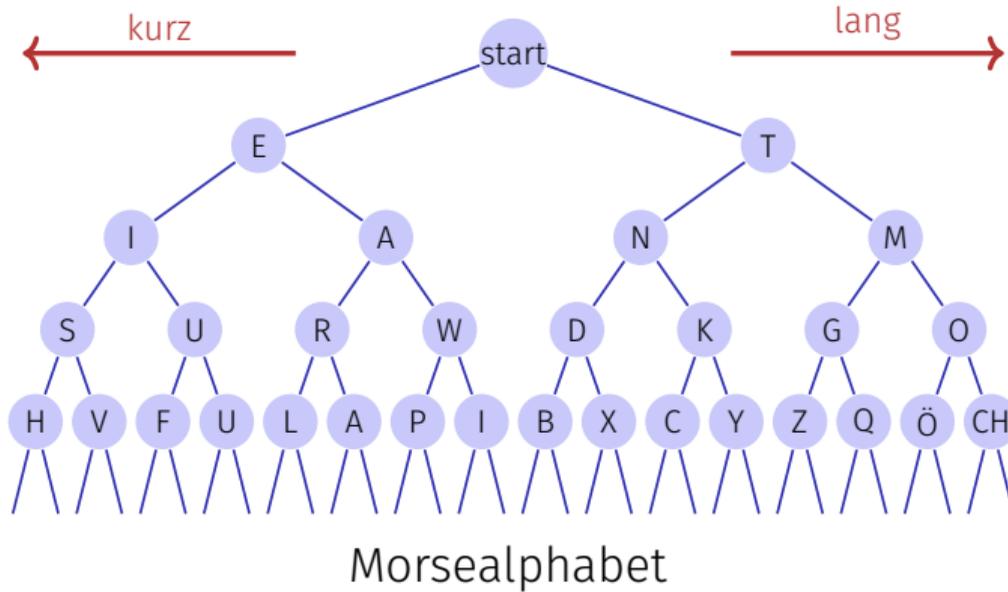
# Bäume

## Verwendung

- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffman Code
- Suchbäume: ermöglichen effizientes Suchen eines Elementes

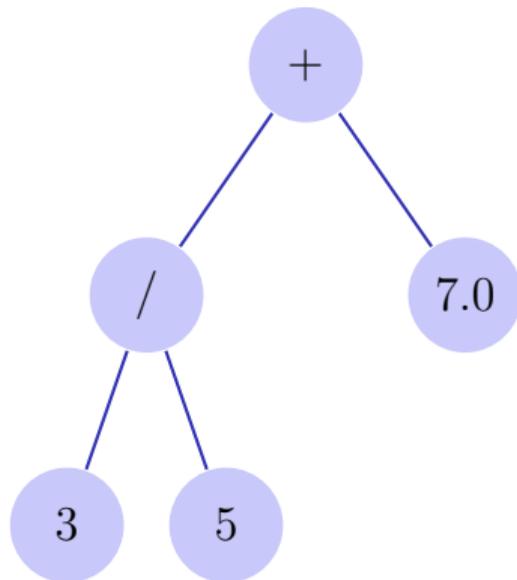


# Beispiele



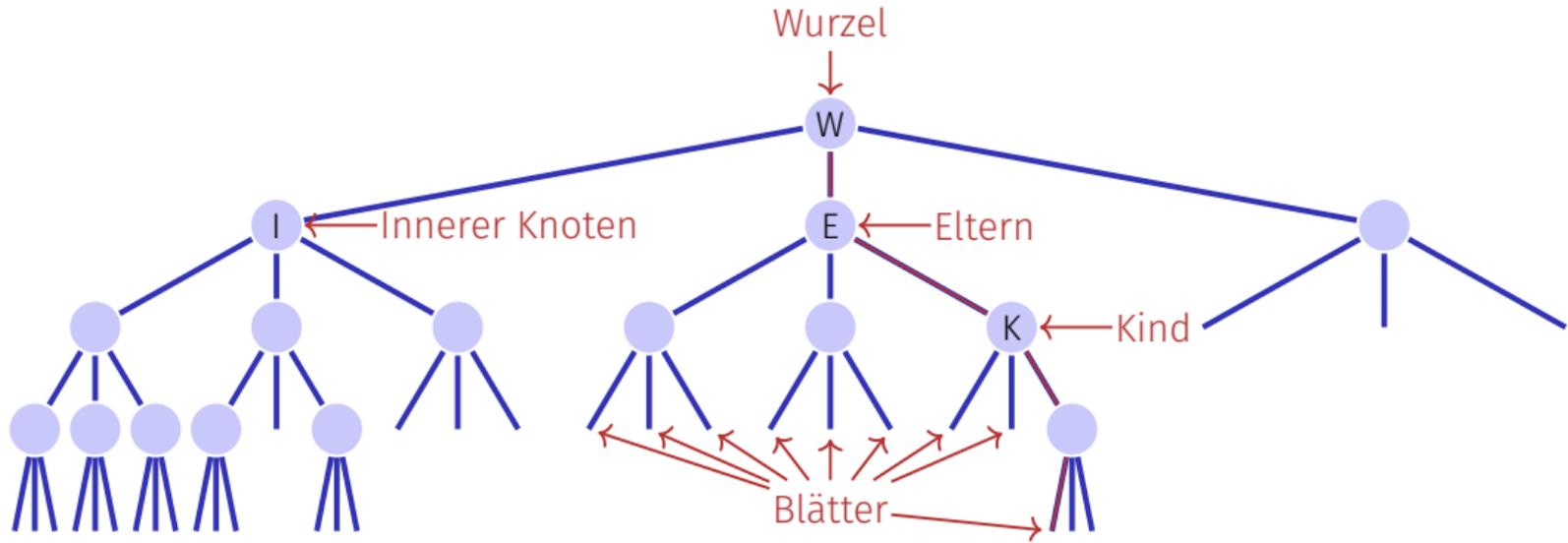
# Beispiele

$3/5 + 7.0$



Ausdrucksbaum

# Nomenklatur



- Ordnung des Baumes: Maximale Anzahl Kindknoten, hier: 3
- Höhe des Baumes: maximale Pfadlänge Wurzel – Blatt (hier: 4)

# Binäre Bäume

Ein binärer Baum ist

- entweder ein Blatt, d.h. ein leerer Baum,
- oder ein innerer Knoten mit zwei Bäumen  $T_l$  (linker Teilbaum) und  $T_r$  (rechter Teilbaum) als linken und rechten Nachfolger.

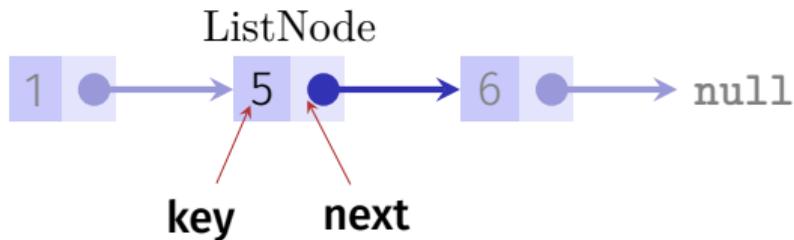
In jedem inneren Knoten  $v$  wird gespeichert

- ein Schlüssel  $v.\mathbf{key}$  und
- zwei Zeiger  $v.\mathbf{left}$  und  $v.\mathbf{right}$  auf die Wurzeln der linken und rechten Teilbäume.



Ein Blatt wird durch den **null**-Zeiger repräsentiert

# Listknoten in Python



```
class ListNode:
    # entries key, next implicit via constructor

    def __init__(self, key , next = None):
        """Constructor that takes a key and, optionally, next."""
        self.key = key
        self.next = next
}
```

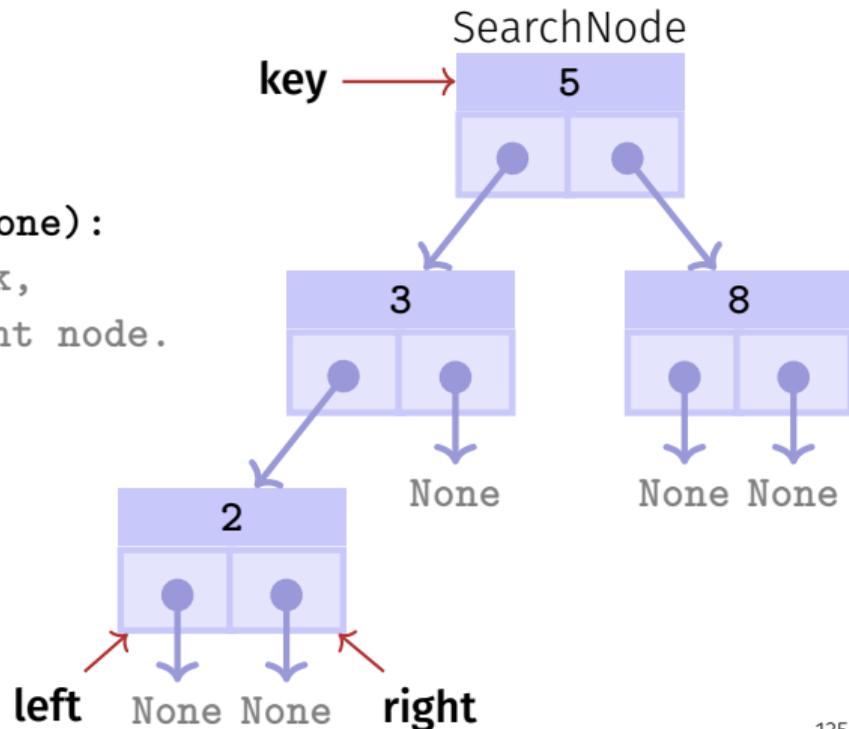
# Jetzt: Baumknoten in Python

```
class SearchNode:
# implicit entries key, left, right

def __init__(self, k, l=None, r=None):
# Constructor that takes a key k,
# and optionally a left and right node.
self.key = k
self.left, self.right = l, r
```

# Jetzt: Baumknoten in Python

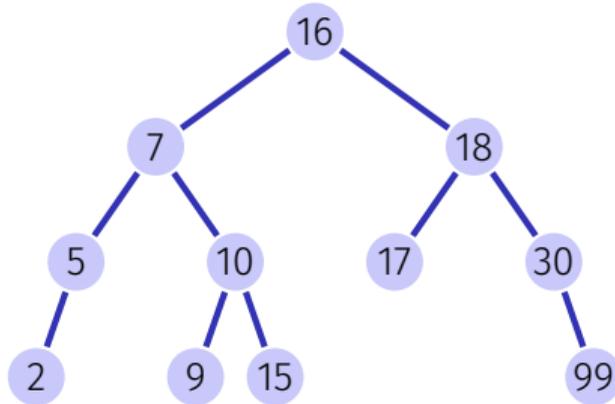
```
class SearchNode:  
    # implicit entries key, left, right  
  
    def __init__(self, k, l=None, r=None):  
        # Constructor that takes a key k,  
        # and optionally a left and right node.  
        self.key = k  
        self.left, self.right = l, r
```



# Binärer Suchbaum

Ein **binärer Suchbaum** ist ein binärer Baum, der die **Suchbaumeigenschaft** erfüllt:

- Jeder Knoten  $v$  speichert einen Schlüssel
- Schlüssel im linken Teilbaum  $v.\mathbf{left}$  kleiner als  $v.\mathbf{key}$
- Schlüssel im rechten Teilbaum  $v.\mathbf{right}$  grösser als  $v.\mathbf{key}$



# Suchen

**Input:** Binärer Suchbaum mit Wurzel  $r$ ,  
Schlüssel  $k$

**Output:** Knoten  $v$  mit  $v.key = k$  oder **null**

$v \leftarrow r$

**while**  $v \neq \text{null}$  **do**

**if**  $k = v.key$  **then**

        | **return**  $v$

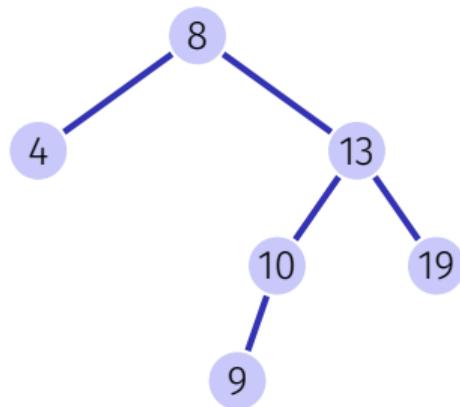
**else if**  $k < v.key$  **then**

        |  $v \leftarrow v.left$

**else**

        |  $v \leftarrow v.right$

**return null**



# Suchen

**Input:** Binärer Suchbaum mit Wurzel  $r$ ,  
Schlüssel  $k$

**Output:** Knoten  $v$  mit  $v.key = k$  oder **null**

$v \leftarrow r$

**while**  $v \neq \text{null}$  **do**

**if**  $k = v.key$  **then**

        | **return**  $v$

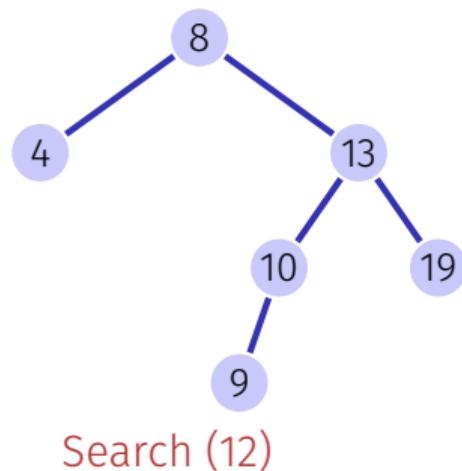
**else if**  $k < v.key$  **then**

        |  $v \leftarrow v.left$

**else**

        |  $v \leftarrow v.right$

**return null**



# Suchen

**Input:** Binärer Suchbaum mit Wurzel  $r$ ,  
Schlüssel  $k$

**Output:** Knoten  $v$  mit  $v.key = k$  oder **null**

$v \leftarrow r$

**while**  $v \neq \text{null}$  **do**

**if**  $k = v.key$  **then**

        | **return**  $v$

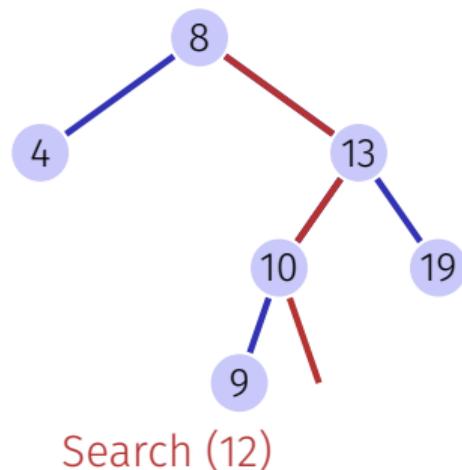
**else if**  $k < v.key$  **then**

        |  $v \leftarrow v.left$

**else**

        |  $v \leftarrow v.right$

**return null**



# Suchen

**Input:** Binärer Suchbaum mit Wurzel  $r$ ,  
Schlüssel  $k$

**Output:** Knoten  $v$  mit  $v.key = k$  oder **null**

$v \leftarrow r$

**while**  $v \neq \text{null}$  **do**

**if**  $k = v.key$  **then**

        | **return**  $v$

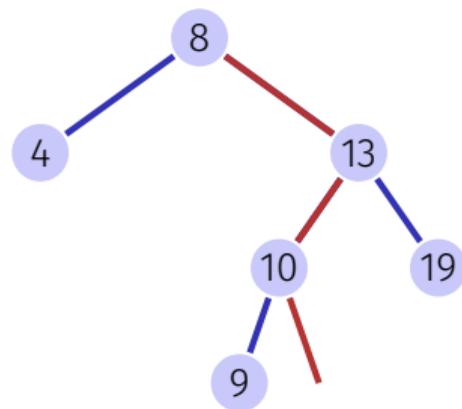
**else if**  $k < v.key$  **then**

        |  $v \leftarrow v.left$

**else**

        |  $v \leftarrow v.right$

**return null**



Search (12) → **null**

# Suchen in Python

```
def findNode(root, key):  
    n = root  
    while n != None and n.key != key:  
        if key < n.key:  
            n = n.left  
        else:  
            n = n.right  
    return n
```

# Höhe eines Baumes

Die Höhe  $h(T)$  eines binären Baumes  $T$  mit Wurzel  $r$  ist gegeben als

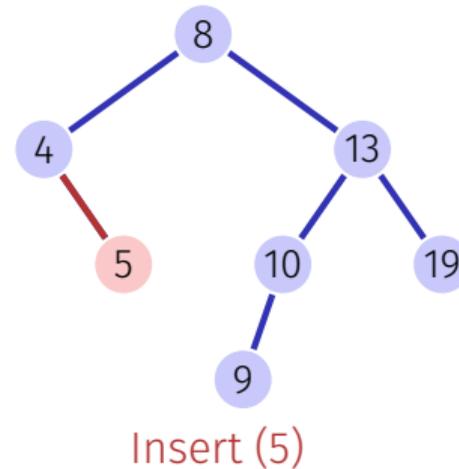
$$h(r) = \begin{cases} 0 & \text{falls } r = \mathbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist somit im schlechtesten Fall  $\mathcal{O}(h(T))$

# Einfügen eines Schlüssels

Einfügen des Schlüssels  $k$

- Suche nach  $k$ .
- Wenn erfolgreich: z.B. Fehlerausgabe
- Wenn erfolglos: Einfügen des Schlüssels am erreichten Blatt.



# Knoten Einfügen in Python

```
def addNode(root, key):
    n = root
    if n == None:
        root = Node(key)
    while n.key != key:
        if key < n.key:
            if n.left == None:
                n.left = Node(key)
            n = n.left
        else:
            if n.right == None:
                n.right = Node(key)
            n = n.right
    return root
```

# Tree in Python

```
class Tree:
    def __init__(self):
        self.root = None

    def find(self, key):
        return findNode(self.root, key)

    def has(self, key):
        return self.find(key) != None

    def add(self, key):
        self.root = addNode(self.root, key)

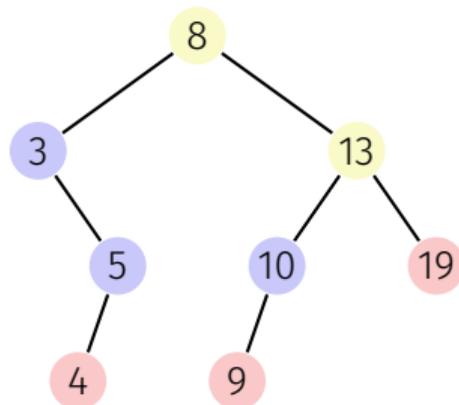
# ....
```

# Knoten entfernen

Drei Fälle möglich

- Knoten hat keine Kinder
- Knoten hat ein Kind
- Knoten hat zwei Kinder

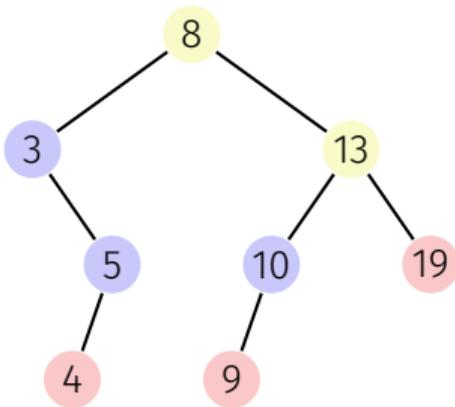
[Blätter zählen hier nicht]



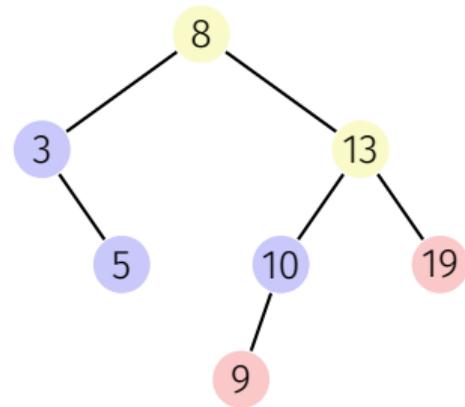
# Knoten entfernen

Knoten hat keine Kinder

Einfacher Fall: Knoten durch Blatt ersetzen.



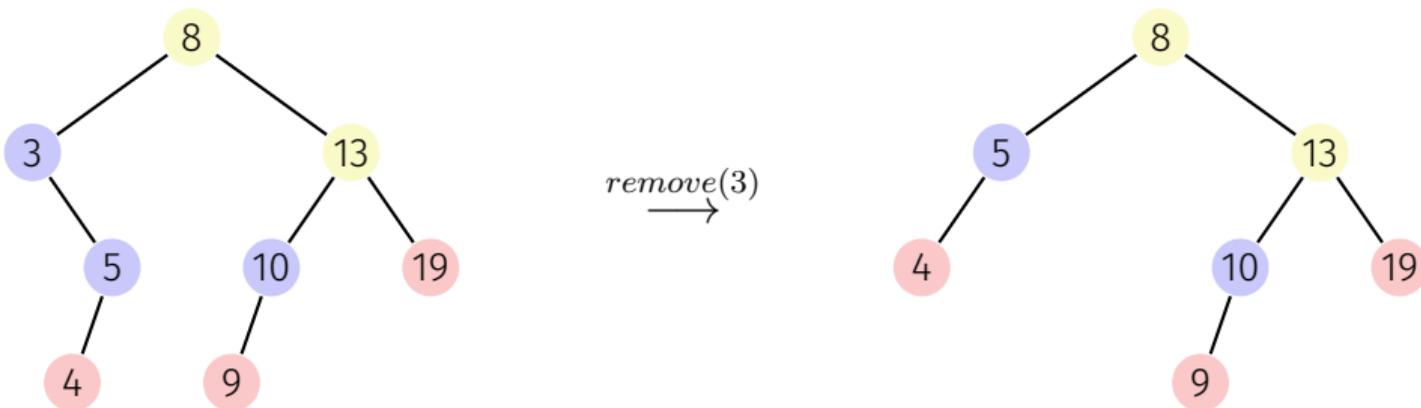
*remove*(4)  
→



# Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



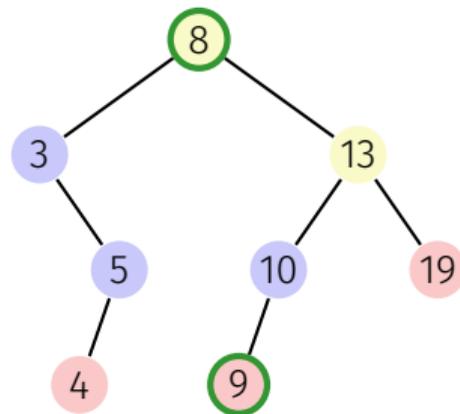
# Knoten entfernen

Knoten  $v$  hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum  $v.\text{right}$  (der **symmetrische Nachfolger** von  $v$ )

- ist kleiner als alle Schlüssel in  $v.\text{right}$
- ist grösser als alle Schlüssel in  $v.\text{left}$
- und hat kein linkes Kind.

Lösung: ersetze  $v$  durch seinen symmetrischen Nachfolger

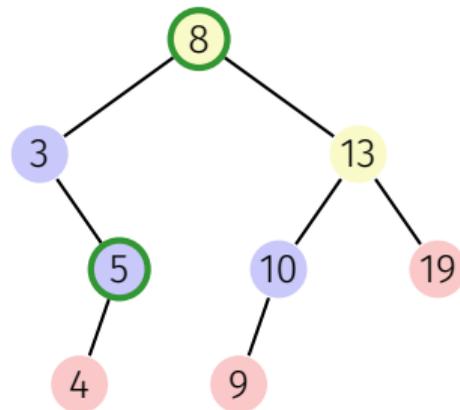


# Aus Symmetriegründen...

Knoten  $v$  hat zwei Kinder

Auch möglich: ersetze  $v$  durch seinen symmetrischen Vorgänger

Implementation: der Teufel steckt im Detail!



# Algorithmus SymmetricSuccessor( $v$ )

**Input:** Knoten  $v$  eines binären Suchbaumes

**Output:** Symmetrischer Nachfolger von  $v$

$w \leftarrow v.\text{right}$

$x \leftarrow w.\text{left}$

**while**  $x \neq \text{null}$  **do**

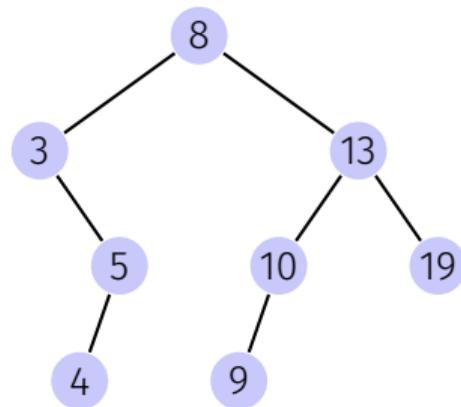
$w \leftarrow x$

$x \leftarrow x.\text{left}$

**return**  $w$

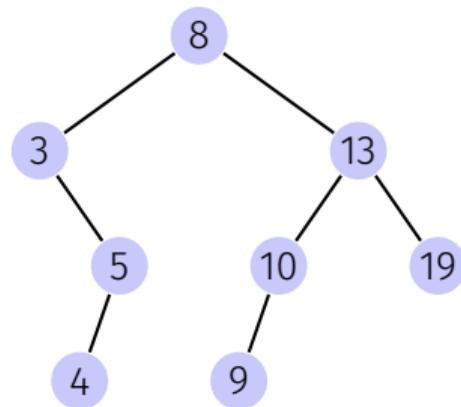
# Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .



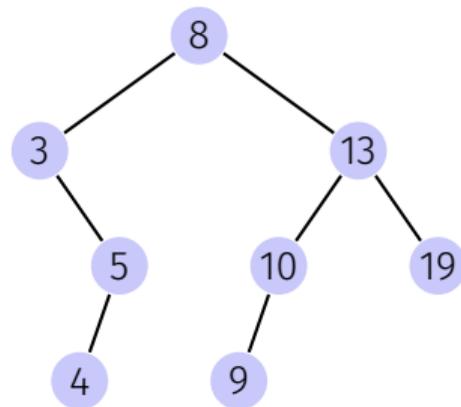
# Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19



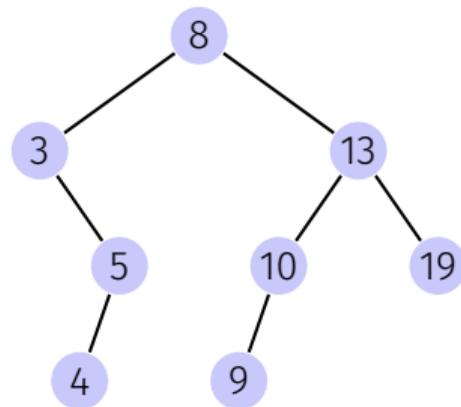
# Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder):  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ , dann  $v$ .



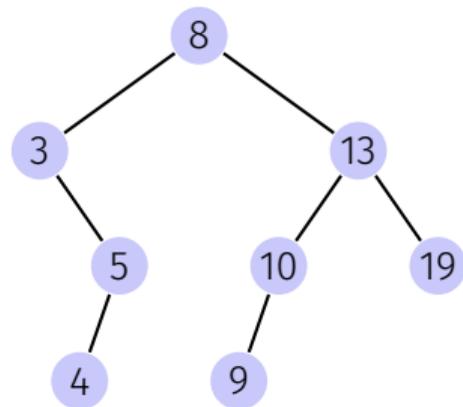
# Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder):  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ , dann  $v$ .  
4, 5, 3, 9, 10, 19, 13, 8



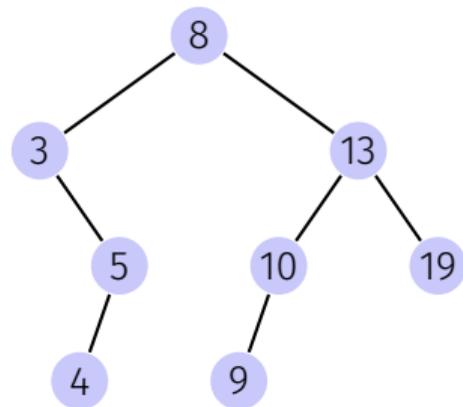
# Traversierungsarten

- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder):  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ , dann  $v$ .  
4, 5, 3, 9, 10, 19, 13, 8
- Symmetrische Reihenfolge (inorder):  $T_{\text{left}}(v)$ , dann  $v$ , dann  $T_{\text{right}}(v)$ .

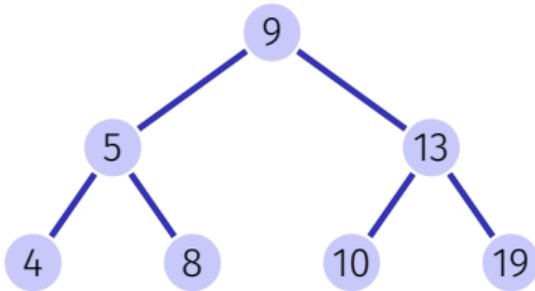


# Traversierungsarten

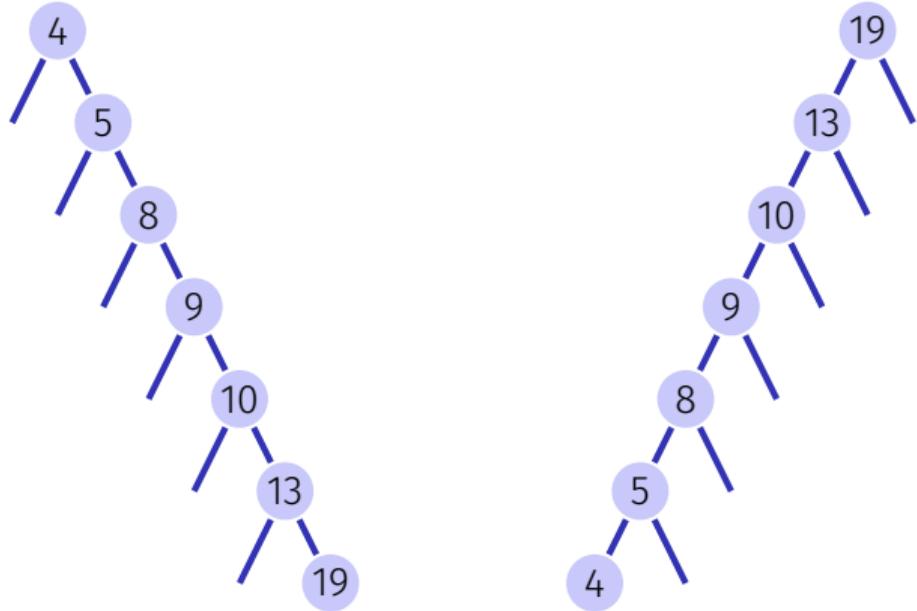
- Hauptreihenfolge (preorder):  $v$ , dann  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ .  
8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder):  $T_{\text{left}}(v)$ , dann  $T_{\text{right}}(v)$ , dann  $v$ .  
4, 5, 3, 9, 10, 19, 13, 8
- Symmetrische Reihenfolge (inorder):  $T_{\text{left}}(v)$ , dann  $v$ , dann  $T_{\text{right}}(v)$ .  
3, 4, 5, 8, 9, 10, 13, 19



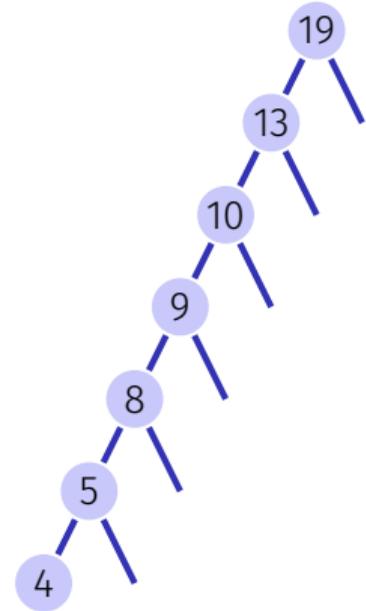
# Degenerierte Suchbäume



Insert 9,5,13,4,8,10,19  
bestmöglich  
balanciert



Insert 4,5,8,9,10,13,19  
Lineare Liste



Insert 19,13,10,9,8,5,4  
Lineare Liste

Ein Suchbaum, welcher aus einer zufälligen Sequenz von Zahlen erstellt wird hat erwartete Pfadlänge von  $\mathcal{O}(\log n)$ .

Achtung: das gilt nur für Einfügeoperation. Wird der Baum zufällig durch Einfügen und Entfernen gebildet, ist die erwartete Pfadlänge  $\mathcal{O}(\sqrt{n})$ .

*Balancierte* Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine  $\mathcal{O}(\log n)$  Worst-Case-Garantie.

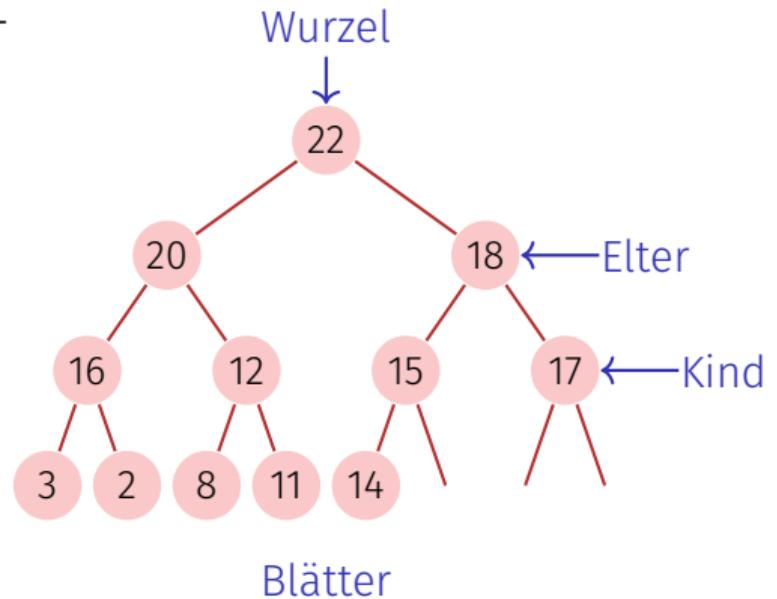
## 9. Heaps

---

Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

# [Max-]Heap\*

Binärer Baum mit folgenden Eigenschaften

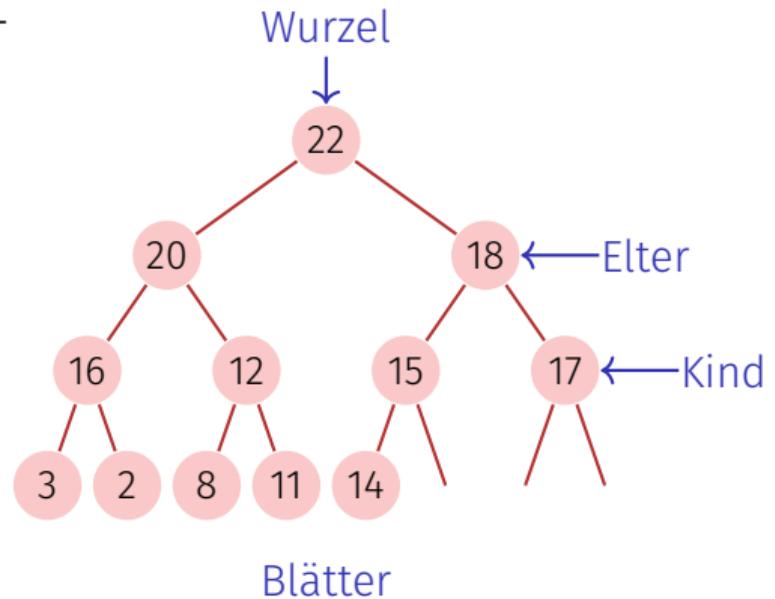


\*Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

# [Max-]Heap\*

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene

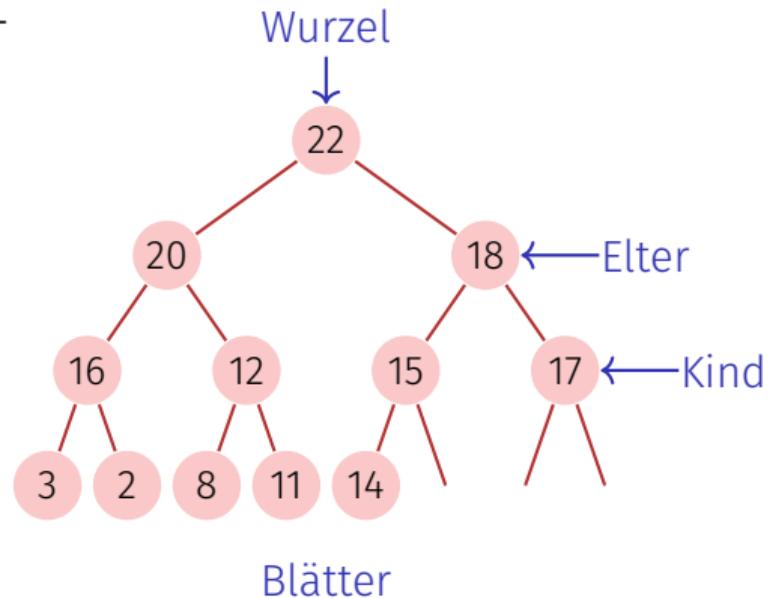


\*Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

# [Max-]Heap\*

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.



\*Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

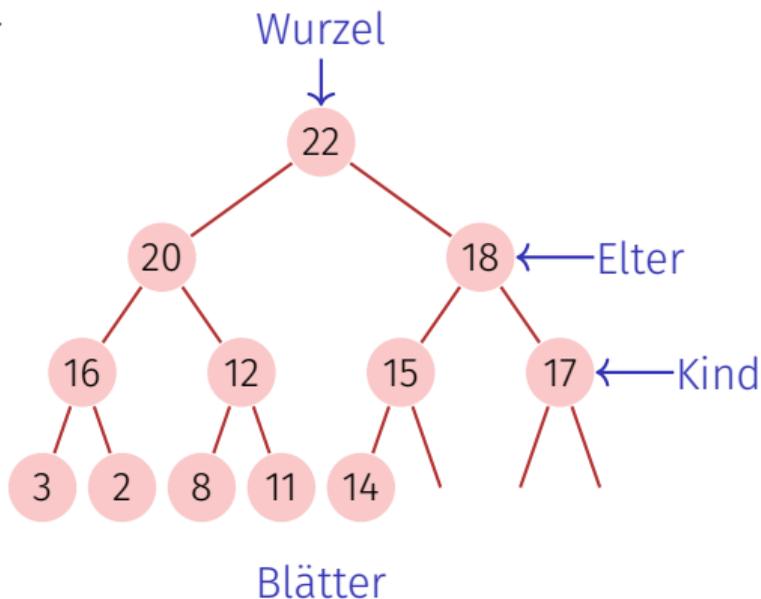
# [Max-]Heap\*

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.

3. **Heap-Bedingung:**

Max-(Min-)Heap: Schlüssel eines Kindes kleiner (größer) als der des Elternknotens

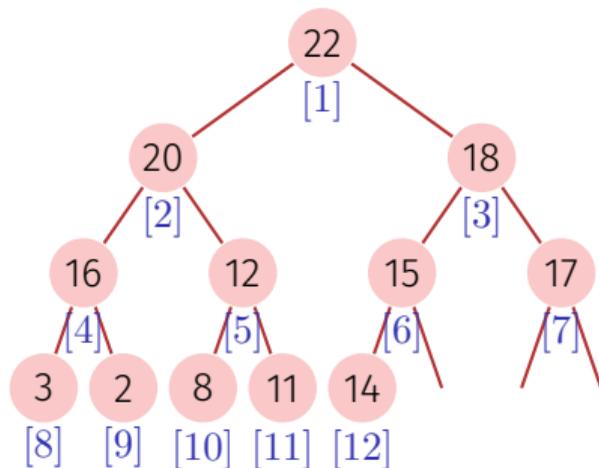
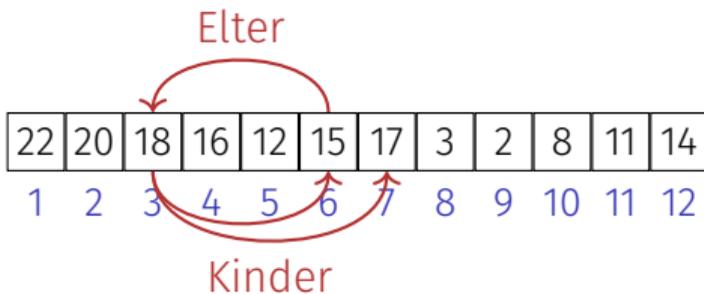


\*Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

# Heap als Array

Baum  $\rightarrow$  Array:

- $\text{Kinder}(i) = \{2i, 2i + 1\}$
- $\text{Elter}(i) = \lfloor i/2 \rfloor$



Abhängig von Startindex!<sup>4</sup>

<sup>4</sup>Für Arrays, die bei 0 beginnen:  $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$ ,  $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

# Höhe eines Heaps

Welche Höhe  $H(n)$  hat ein Heap mit  $n$  Knoten? Auf der  $i$ -ten Ebene eines Binären Baumes befinden sich höchstens  $2^i$  Knoten. Bis auf die letzte Ebene sind alle Ebenen eines Heaps aufgefüllt.

$$H(n) = \min\{h \in \mathbb{N} : \sum_{i=0}^{h-1} 2^i \geq n\}$$

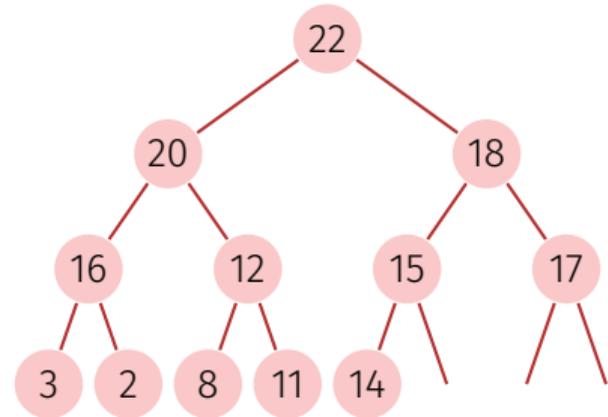
Mit  $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ :

$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

also

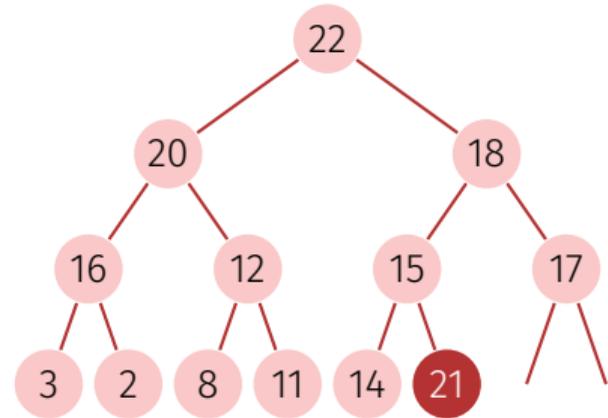
$$H(n) = \lceil \log_2(n + 1) \rceil.$$

# Einfügen



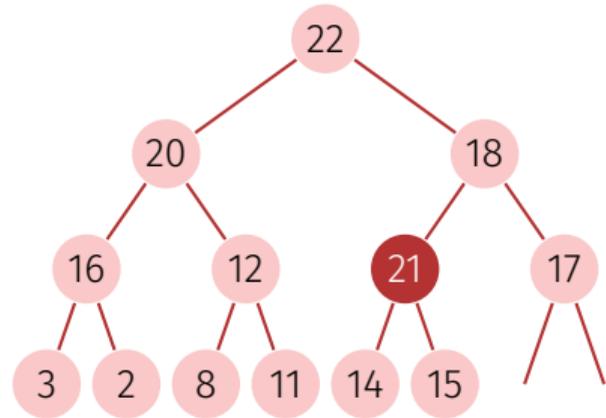
# Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.



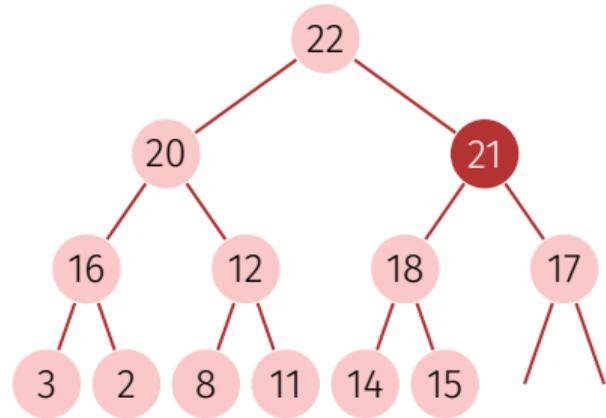
# Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



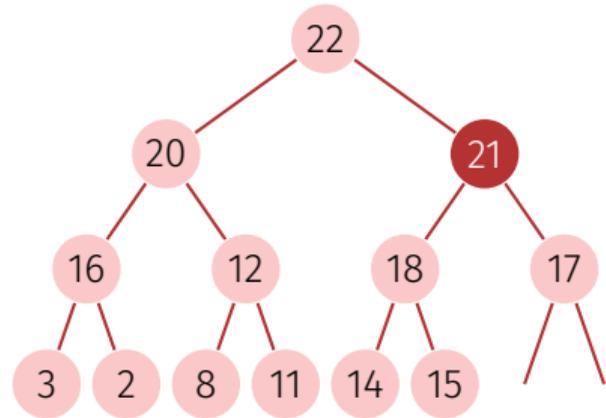
# Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



# Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall:  $\mathcal{O}(\log n)$



# Algorithmus Aufsteigen( $A, m$ )

**Input:** Array  $A$  mit mindestens  $m$  Elementen und Max-Heap-Struktur auf  $A[1, \dots, m-1]$

**Output:** Array  $A$  mit Max-Heap-Struktur auf  $A[1, \dots, m]$ .

$v \leftarrow A[m]$  // Wert

$c \leftarrow m$  // derzeitiger Knoten (child)

$p \leftarrow \lfloor c/2 \rfloor$  // Elternknoten (parent)

**while**  $c > 1$  and  $v > A[p]$  **do**

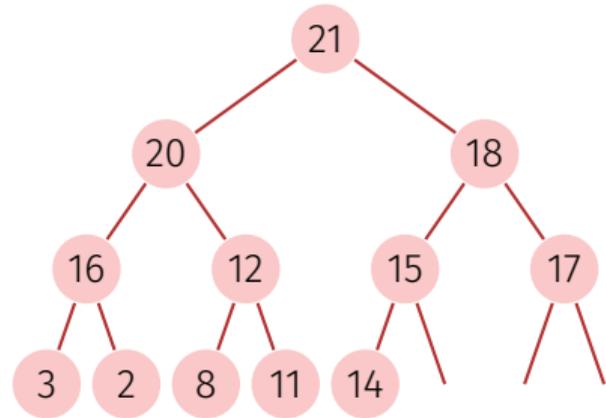
$A[c] \leftarrow A[p]$  // Wert Elternknoten  $\rightarrow$  derzeitiger Knoten

$c \leftarrow p$  // Elternknoten  $\rightarrow$  derzeitiger Knoten

$p \leftarrow \lfloor c/2 \rfloor$

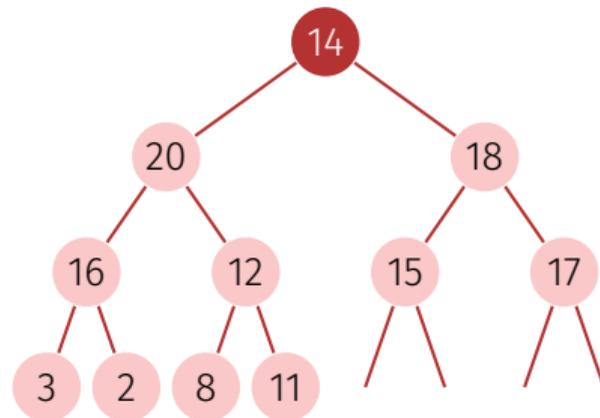
$A[c] \leftarrow v$  // Wert  $\rightarrow$  Wurzel des (Teil-)Baumes

# Maximum entfernen



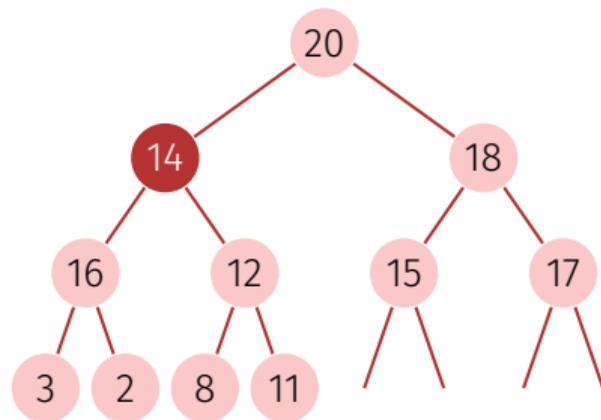
# Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



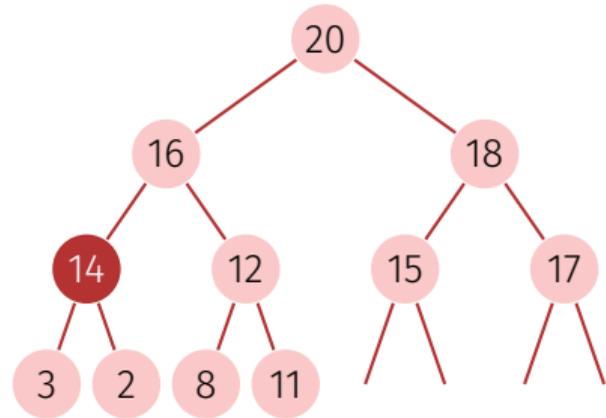
# Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her:  
Sukzessives Absinken (in Richtung des grösseren Kindes).



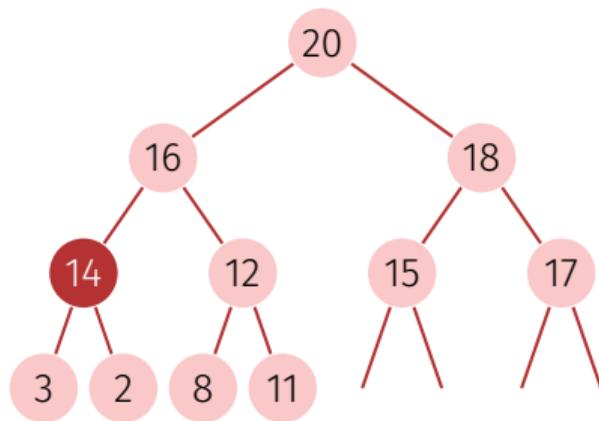
# Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



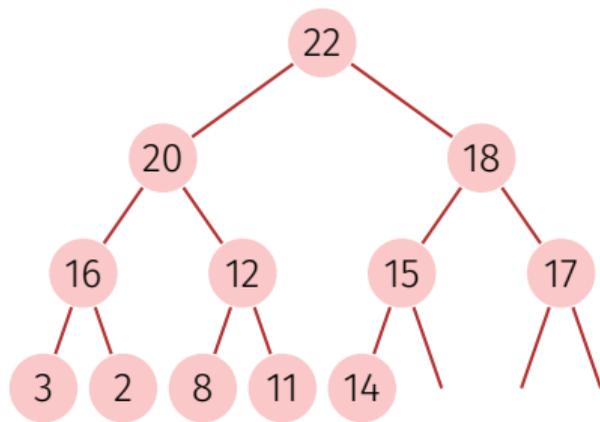
# Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her:  
Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall:  
 $\mathcal{O}(\log n)$



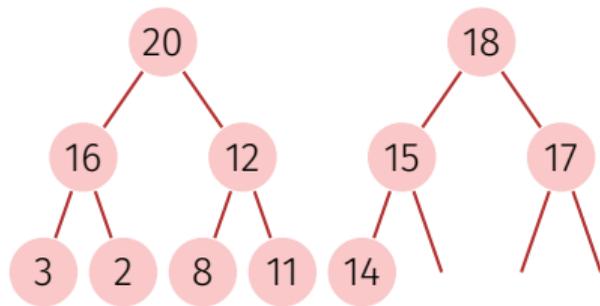
# Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



# Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



# Algorithmus Versickern( $A, i, m$ )

**Input:** Array  $A$  mit Heapstruktur für die Kinder von  $i$ . Letztes Element  $m$ .

**Output:** Array  $A$  mit Heapstruktur für  $i$  mit letztem Element  $m$ .

**while**  $2i \leq m$  **do**

$j \leftarrow 2i$ ; //  $j$  linkes Kind

**if**  $j < m$  and  $A[j] < A[j + 1]$  **then**

$j \leftarrow j + 1$ ; //  $j$  rechtes Kind mit grösserem Schlüssel

**if**  $A[i] < A[j]$  **then**

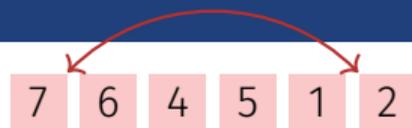
        swap( $A[i], A[j]$ )

$i \leftarrow j$ ; // weiter versickern

**else**

$i \leftarrow m$ ; // versickern beendet

# Heap Sortieren



$A[1, \dots, n]$  ist Heap.

Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1);$
- $n \leftarrow n - 1$

# Heap Sortieren

Tauschen  $\Rightarrow$

7	6	4	5	1	2
2	6	4	5	1	7

$A[1, \dots, n]$  ist Heap.

Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1);$
- $n \leftarrow n - 1$

# Heap Sortieren

$A[1, \dots, n]$  ist Heap.

Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1);$
- $n \leftarrow n - 1$

Tauschen  $\Rightarrow$

7	6	4	5	1	2
---	---	---	---	---	---

2	6	4	5	1	7
---	---	---	---	---	---

Versickern  $\Rightarrow$

6	5	4	2	1	7
---	---	---	---	---	---

# Heap Sortieren

$A[1, \dots, n]$  ist Heap.

Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1);$
- $n \leftarrow n - 1$

Tauschen  $\Rightarrow$

7	6	4	5	1	2
---	---	---	---	---	---

2	6	4	5	1	7
---	---	---	---	---	---

Versickern  $\Rightarrow$

6	5	4	2	1	7
---	---	---	---	---	---

Tauschen  $\Rightarrow$

1	5	4	2	6	7
---	---	---	---	---	---

# Heap Sortieren

$A[1, \dots, n]$  ist Heap.

Solange  $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{Versickere}(A, 1, n - 1);$
- $n \leftarrow n - 1$

Tauschen  $\Rightarrow$

7	6	4	5	1	2
---	---	---	---	---	---

Versickern  $\Rightarrow$

2	6	4	5	1	7
---	---	---	---	---	---

Tauschen  $\Rightarrow$

6	5	4	2	1	7
---	---	---	---	---	---

Versickern  $\Rightarrow$

1	5	4	2	6	7
---	---	---	---	---	---

Tauschen  $\Rightarrow$

5	4	2	1	6	7
---	---	---	---	---	---

Versickern  $\Rightarrow$

1	4	2	5	6	7
---	---	---	---	---	---

Tauschen  $\Rightarrow$

4	1	2	5	6	7
---	---	---	---	---	---

Versickern  $\Rightarrow$

2	1	4	5	6	7
---	---	---	---	---	---

Tauschen  $\Rightarrow$

2	1	4	5	6	7
---	---	---	---	---	---

1	2	4	5	6	7
---	---	---	---	---	---

# Heap erstellen

**Beobachtung:** Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

**Folgerung:**

# Heap erstellen

**Beobachtung:** Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

**Folgerung:** Induktion von unten!

# Algorithmus HeapSort( $A, n$ )

**Input:** Array  $A$  der Länge  $n$ .

**Output:**  $A$  sortiert.

// Heap Bauen.

**for**  $i \leftarrow n/2$  **downto** 1 **do**

└ Versickere( $A, i, n$ );

// Nun ist  $A$  ein Heap.

**for**  $i \leftarrow n$  **downto** 2 **do**

└ swap( $A[1], A[i]$ )

└ Versickere( $A, 1, i - 1$ )

// Nun ist  $A$  sortiert.

# Analyse: Sortieren eines Heaps

Versickere durchläuft maximal  $\log n$  Knoten. An jedem Knoten 2 Schlüsselvergleiche.  $\Rightarrow$  Heap Sortieren kostet im schlechtesten Fall  $2n \log n$  Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch  $\mathcal{O}(n \log n)$ .

# Analyse: Heap bauen

Aufrufe an Versickern:  $n/2$ .

Also Anzahl Vergleiche und Bewegungen  $v(n) \in \mathcal{O}(n \log n)$ .

Versickerpfade sind aber im Mittel viel kürzer:

Wir verwenden, dass  $h(n) = \lceil \log_2 n + 1 \rceil = \lfloor \log_2 n \rfloor + 1$  für  $n > 0$

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level } l} \cdot \underbrace{(\lfloor \log_2 n \rfloor + 1 - l - 1)}_{\text{Höhe Heaps auf Level } l} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k \\ &= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n) \end{aligned}$$

mit  $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$  ( $0 < x < 1$ ) und  $s(\frac{1}{2}) = 2$