Algorithmen und Datenstrukturen

Vorlesung 6

Binäre Suchbäume I

UNI FREIBURG

Fabian Kuhn
Algorithmen und Komplexität

Abstrakte Datentypen : Dictionary

UNI FREIBURG

Dictionary: (auch: Maps, assoziative Arrays)

- Verwaltet eine Menge von Elementen, wo bei jedes Element durch einen eindeutigen Schlüssel (key) repräsentiert wird

Operationen:

- *create* : erzeugt einen leeren Dictionary
- *D.insert(key, value)* : fügt neues *(key,value)*-Paar hinzu
  - falls schon ein Eintrag für *key* besteht, wird er ersetzt
- *D.find(key)* : gibt Eintrag zu Schlüssel *key* zurück
  - falls ein Eintrag vorhanden (gibt sonst einen Default-Wert zurück)
- *D.delete(key)* : löscht Eintrag zu Schlüssel *key*

Mit Hashtabellen in (amortisiert) konstanter Zeit!

Abstrakte Datentypen : Dictionary

UNI FREIBURG

Dictionary:

Weitere mögliche Operationen:

- *D.minimum()* : gibt kleinsten *key* in der Datenstruktur zurück
- *D.maximum()* : gibt grössten *key* in der Datenstruktur zurück
- *D.successor(key)* : gibt nächstgrösseren *key* zurück
- *D.predecessor(key)* : gibt nächstkleineren *key* zurück
- *D.getRange(k1, k2)* : gibt alle Einträge mit Schlüsseln im Intervall *[k1,k2]* zurück

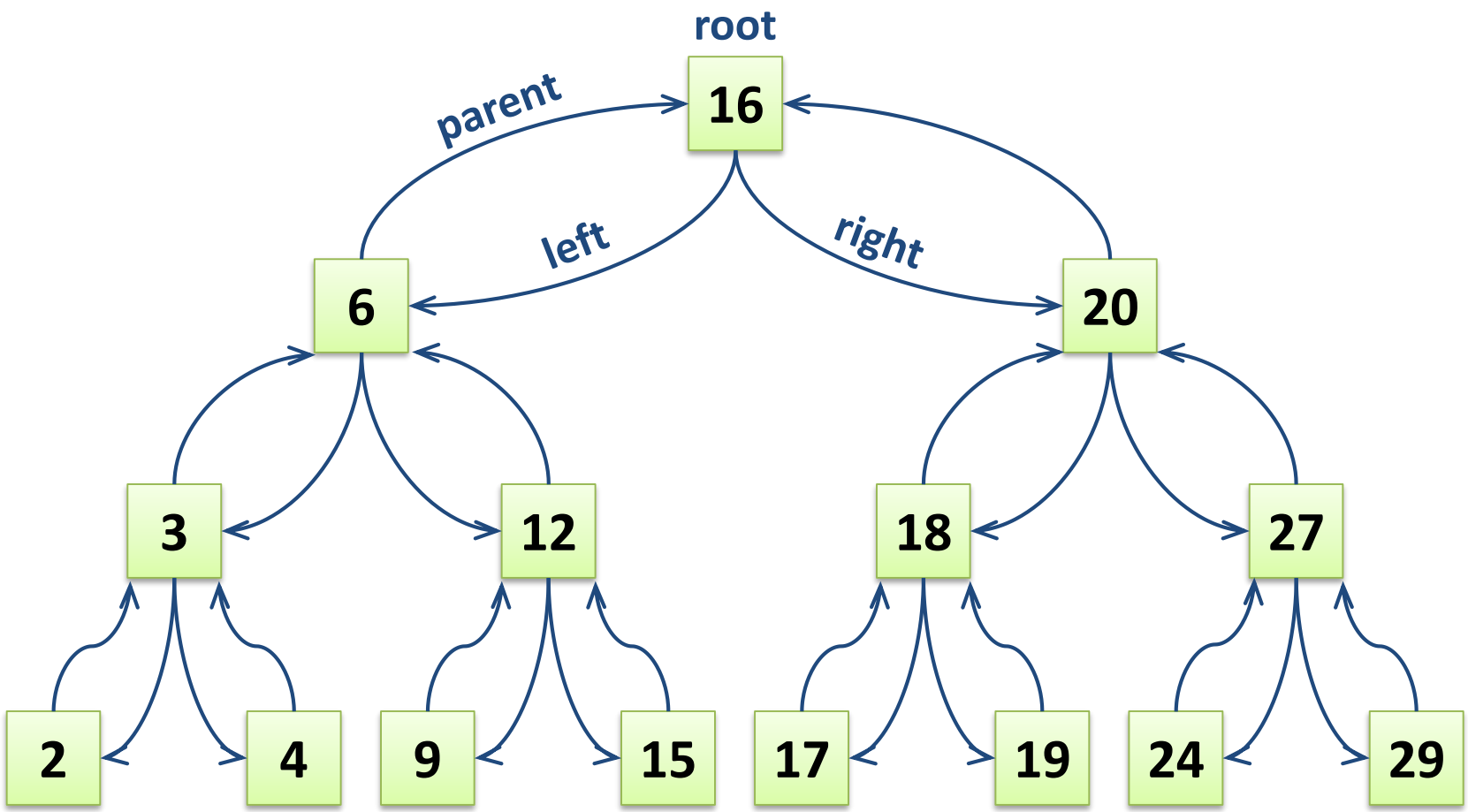
Diese Operationen können mit einer Hashtabelle nicht effizient implementiert werden.

Binäre Suchbäume: Idee

Suche nach Schlüssel 19:

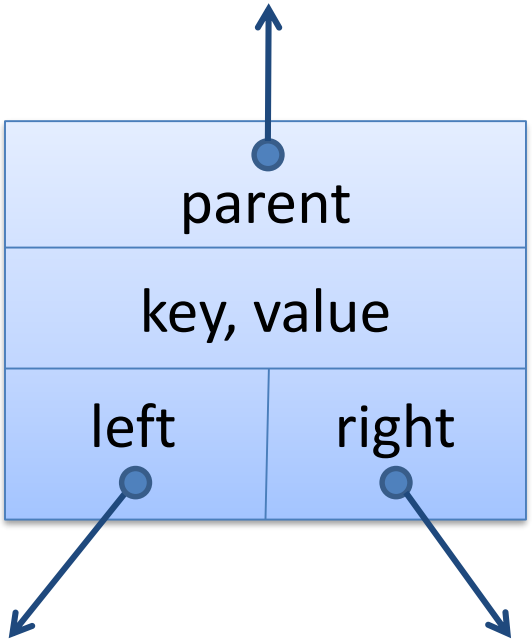
Binäre Suchbäume: Idee

- Benutze den Suchbaum der binären Suche als Datenstruktur



Binärer Suchbaum : Elemente

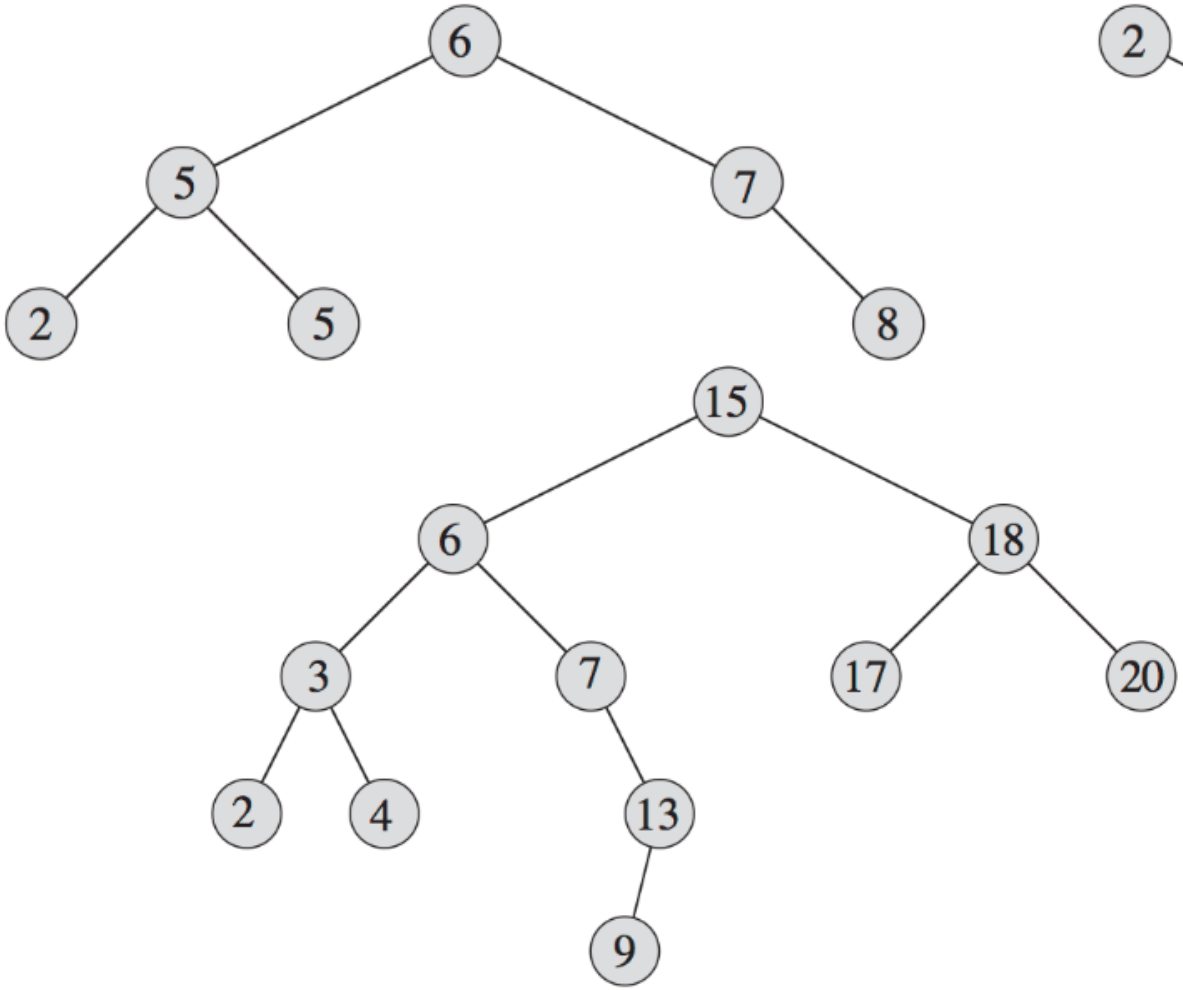
TreeElement:

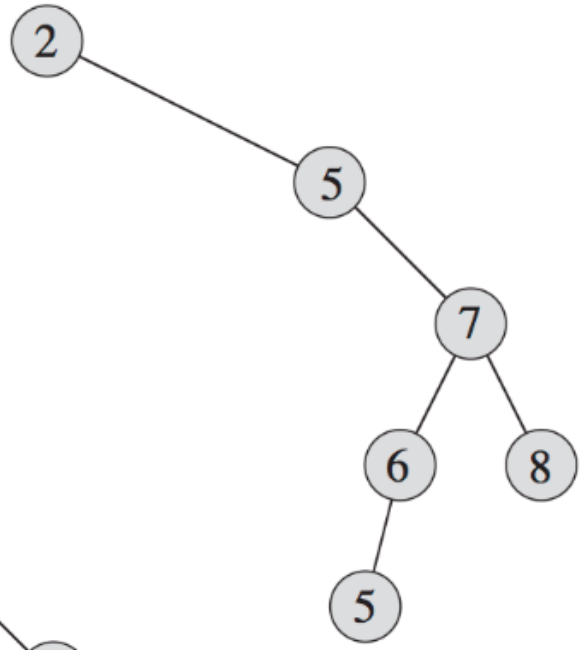


Implementierung: gleich wie bei den Listen-Elementen

Binäre Suchbäume

- Binäre Suchbäume müssen nicht immer so schön symmetrisch sein…



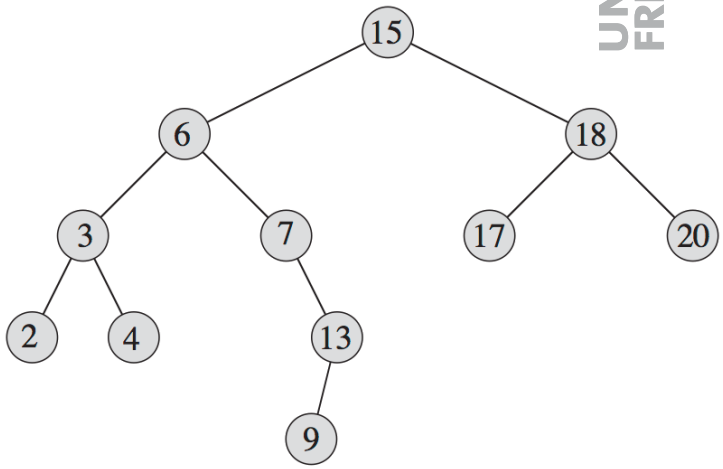


Quelle: [CLRS]

Suche in einem binären Suchbaum

Suche nach Schlüssel x

- Verwende binäre Suche
  - Deshalb heisst es binärer Suchbaum …



Laufzeit: O(Tiefe des Baums)

```
current = root
while current is not None and current.key != x:
    if current.key > x:
        current = current.left
    else:
        current = current.right
```

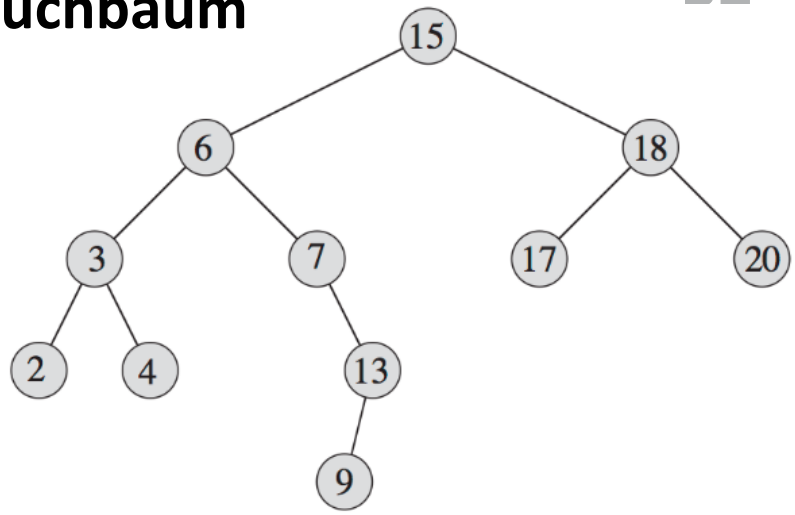
Am Schluss:

- Schlüssel x nicht im Baum : `current == None`
- Schlüssel x gefunden : `current.key == x`

Suche Minimum / Maximum

Finde kleinstes Element in einem bin. Suchbaum

- Alle kleineren Element sind immer im linken Teilbaum

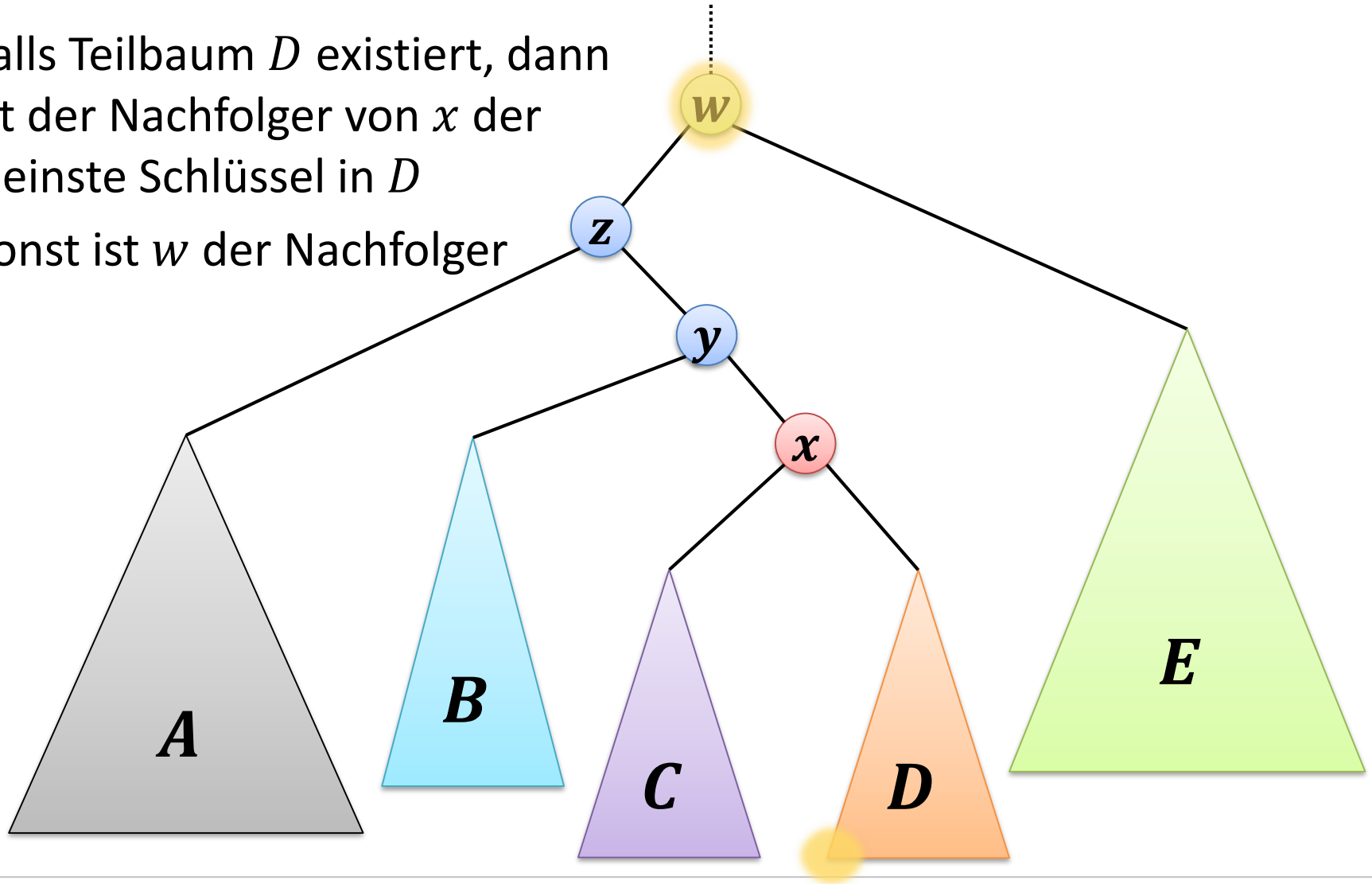



```
current = root
while current.left is not None:
    current = current.left
```

Suche Nachfolger

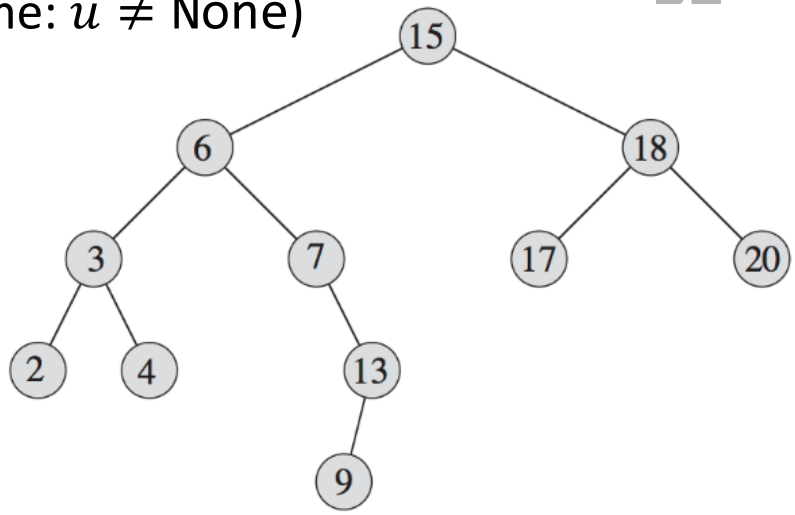
Reihenfolge: $A < z < B < y < C < x < D < w < E$

- Falls Teilbaum D existiert, dann ist der Nachfolger von x der kleinste Schlüssel in D
- Sonst ist w der Nachfolger



Suche Nachfolger

Finde Nachfolger eines Knoten u (Annahme: $u \neq$ None)

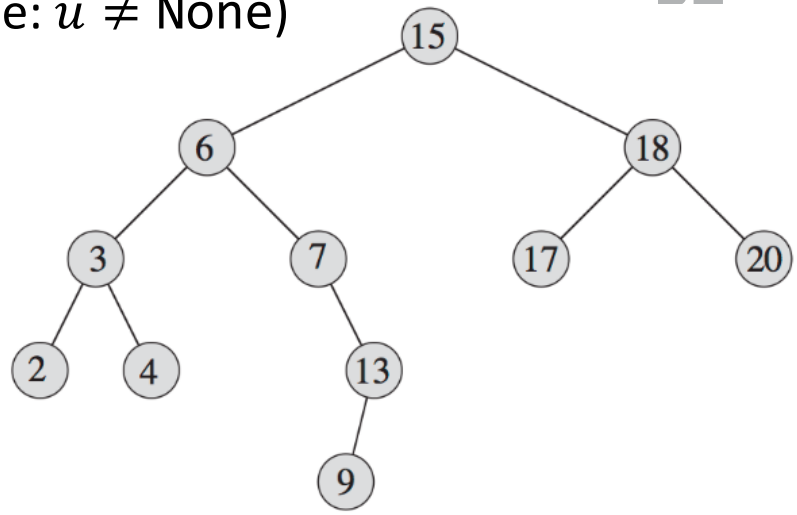



```
if u.right is not None:
    # min in right subtree
    current = u.right
    while current.left is not None:
        current = current.left
    return current
else
    # find first node towards root s.t. u is in left subtree
    current = u
    parent = current.parent
    while parent is not None and current == parent.right:
        current = parent
        parent = current.parent
    return parent
```

Laufzeit: O(Tiefe des Baums)

Suche Vorgänger

Finde Vorgänger eines Knoten u (Annahme: $u \neq$ None)



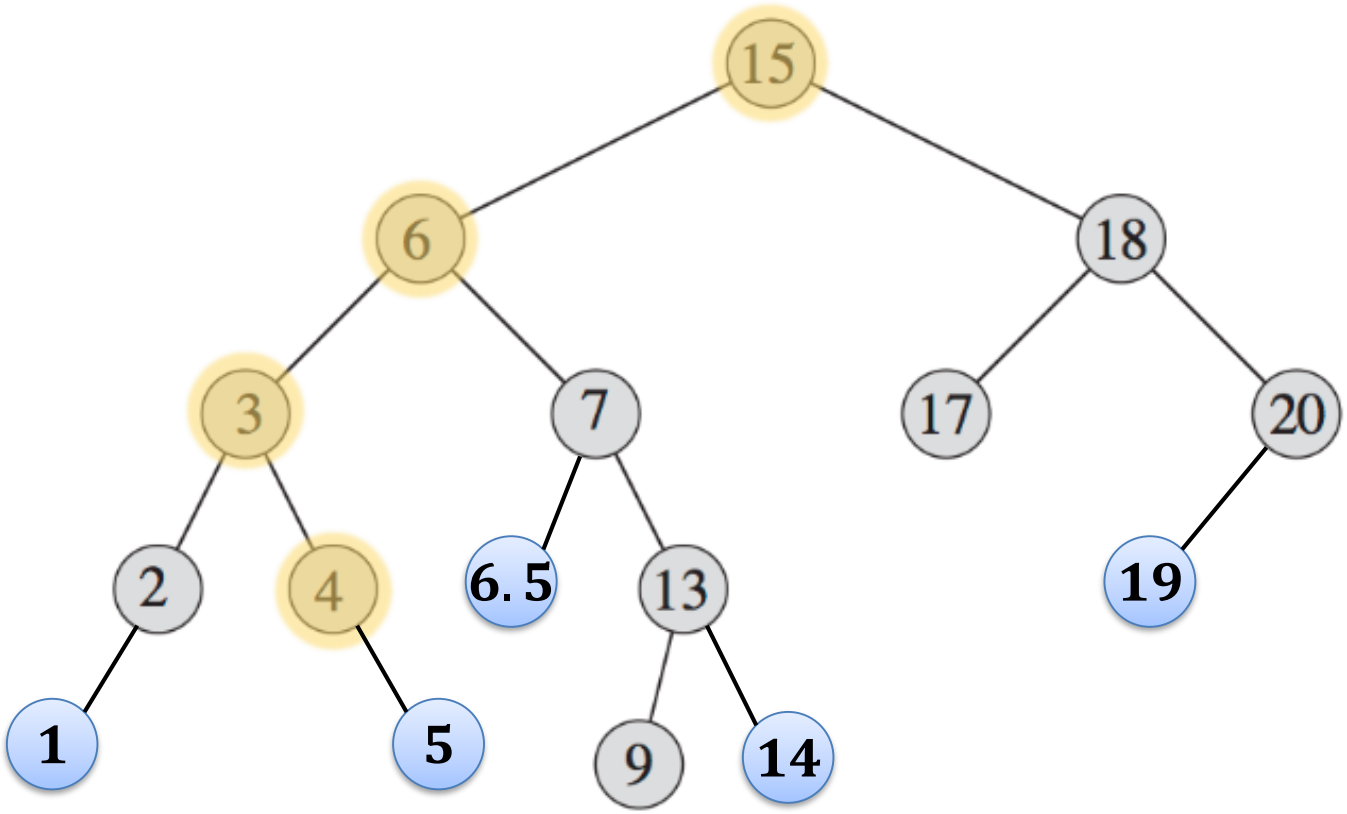

```
if u.left is not None:
    # max in left subtree
    current = u.left
    while current.right is not None:
        current = current.right
    return current
else
    # find first node towards root s.t. u is in right subtree
    current = u
    parent = current.parent
    while parent is not None and current == parent.left:
        current = parent
        parent = current.parent
    return parent
```

Laufzeit: O(Tiefe des Baums)

Einfügen eines Schlüssels

UNI FREIBURG

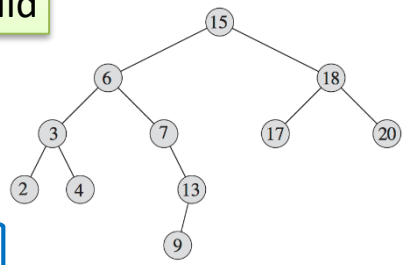
Füge Schlüssel 5, 1, 14, 6.5, 19 ein…



Laufzeit: O(Tiefe des Baums)

Einfügen eines Schlüssels x mit Wert a

Schlüssel | Wert | Parent | Left Child | Right Child



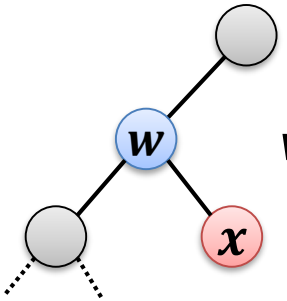

```
if root is None:
   root = new TreeElement(x, a, None, None, None)
else:
   current = root; parent = None
   while current is not None and current.key != x:
      parent = current
      if x < current.key:
         current = current.left
      else:
         current = current.right
   if current is None:   (Schlüssel x ist nicht im Baum enthalten)
      if x < parent.key:
         parent.left = new TreeElement(x, a, parent, None, None)
      else:
         parent.right = new TreeElement(x, a, parent, None, None)
   else:
      current.value = a  (Schlüssel x ist schon enthalten, ersetze den Wert)
```

Binäre Suche

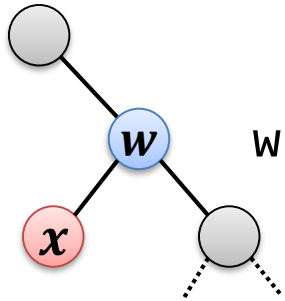
Löschen eines Schlüssels I

Lösche Schlüssel x, einfache Fälle:

- Schlüssel x ist in einem Blatt u des Baums
  - Blatt = Knoten hat keine Kinder

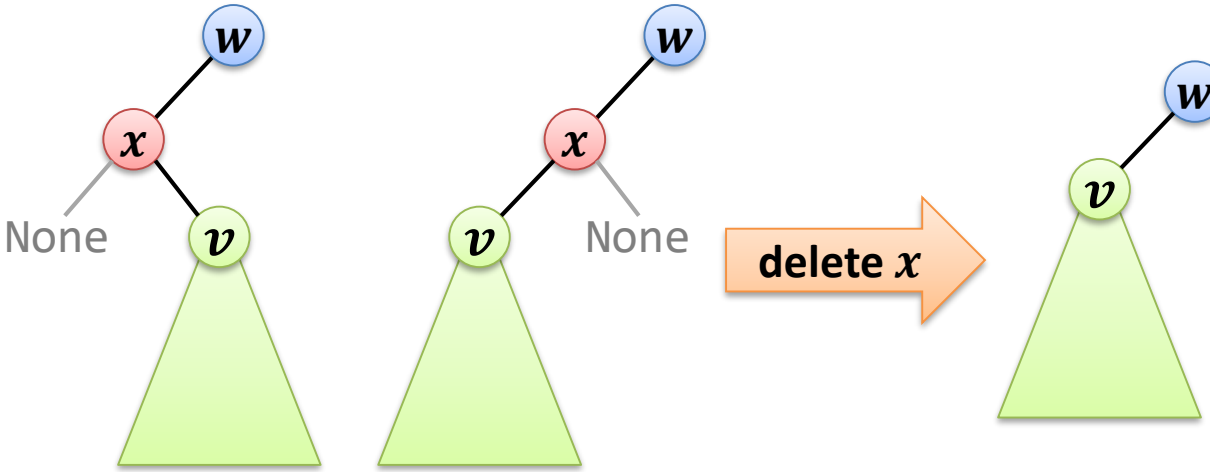


`w.right = None`



`w.left = None`

- Knoten mit Schlüssel x hat nur 1 Kind



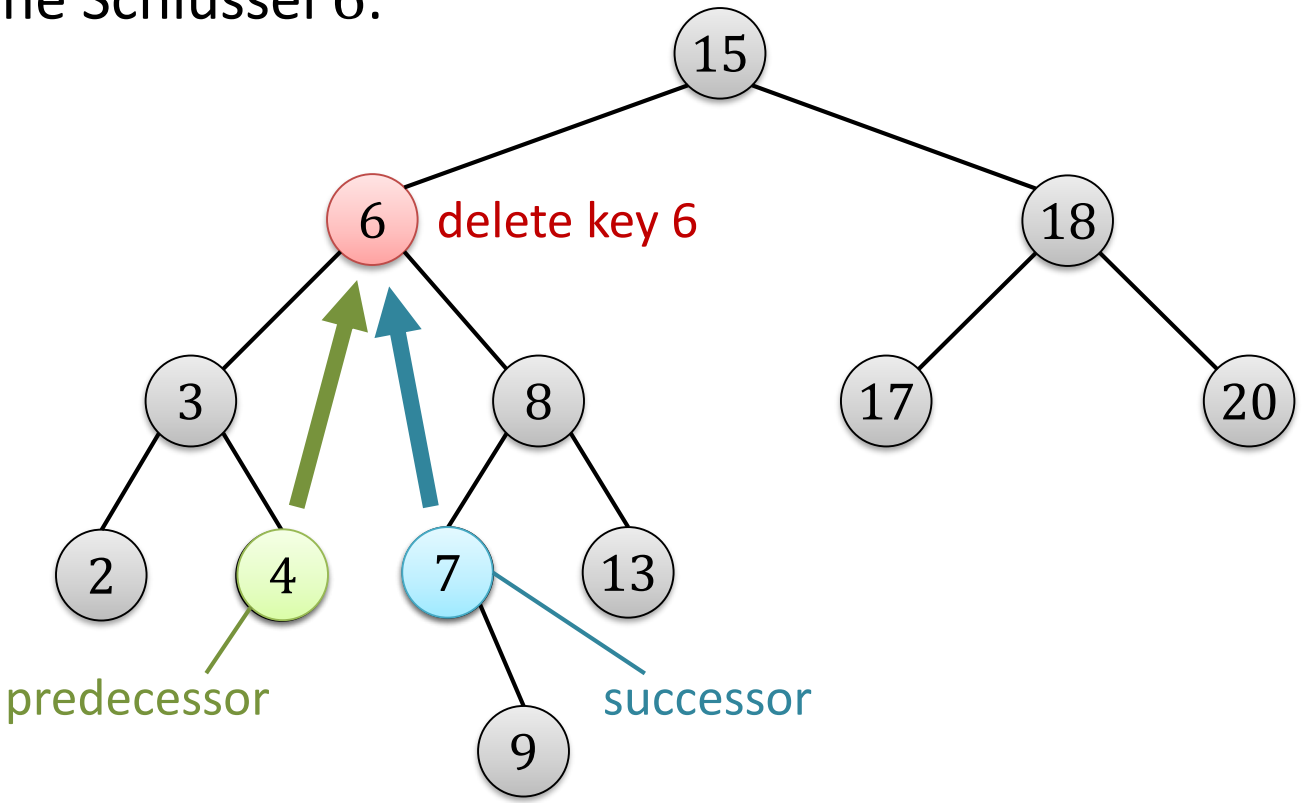
`w.left = v`

Case, where x is the right child of w is symmetric.

Löschen eines Schlüssels II

Lösche Schlüssel x, Knoten hat zwei Kinder:

- Lösche Schlüssel 6:



Löschen eines Schlüssels III

Lösche Schlüssel x, Knoten hat zwei Kinder:

- Vorgänger ist grösster Knoten im linken Teilbaum
  - Vorgänger hat kein rechtes Kind
- Nachfolger ist kleinster Knoten im rechten Teilbaum
  - Nachfolger hat kein linkes Kind

- Schreibe Schlüssel und Daten des Vorgängers (oder alternativ Nachfolgers) in den Knoten von x
- Lösche Vorgänger/Nachfolger-Knoten
  - Vorgänger/Nachfolger ist entweder ein Blatt oder hat nur ein Kind

Löschen eines Schlüssels IV

UNI FREIBURG

Lösche Schlüssel x:

1. Finde Knoten u mit $u.key = x$
   - wie üblich mit binärer Suche
2. Falls u nicht 2 Kinder hat, lösche Knoten u
   - Annahme: v ist Parent von u, u ist linkes Kind von v (anderer Fall analog)
   - Fall u ein Blatt ist, wird $v.left = \text{None}$
   - Falls u ein Kind w hat, wird $v.left = w$
3. Falls u zwei Kinder hat, dann bestimme Vorgängerknoten v
   - Funktioniert auch mit Nachfolgerknoten
4. Setze $u.key = v.key$ und $u.data = v.data$
5. Lösche Knoten v (gleich, wie oben u gelöscht wird)
   - Knoten v hat höchstens 1 Kind!

Laufzeit: $O(\text{Tiefe des Baums})$

Laufzeiten Binärer Suchbaum

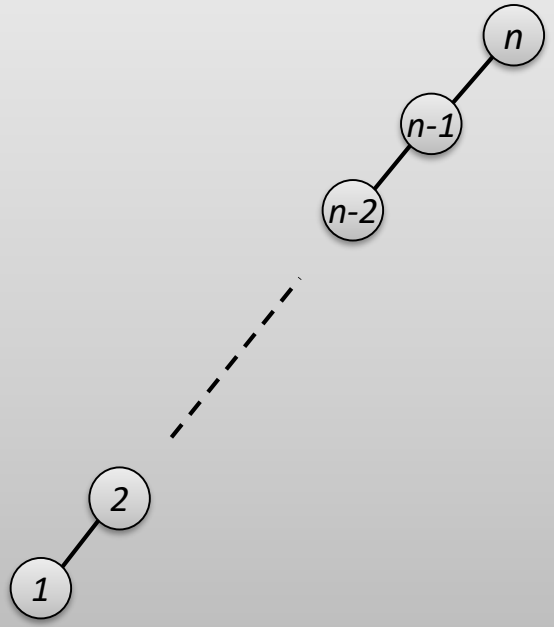
Die Operationen

find, min, max, predecessor, successor, insert, delete

haben alle **Laufzeit O(Tiefe des Baums)**.

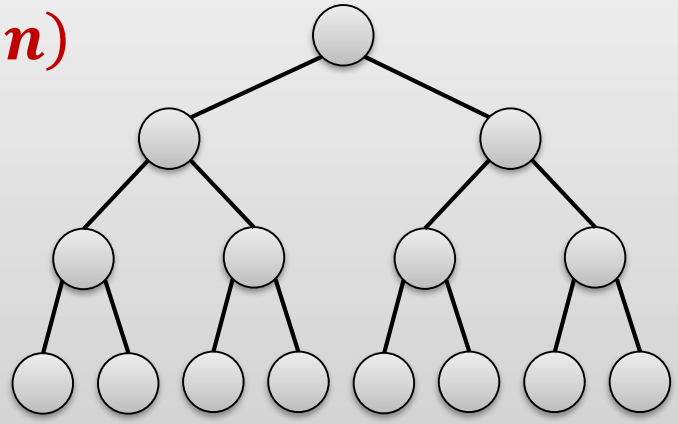
Was ist die Tiefe eines binären Suchbaums?

Worst Case: $\Theta(n)$



Best Case: $\Theta(\log n)$

- Max. #Knoten in Tiefe k ist 2^k
- Tiefe $\geq \lfloor \log_2 n \rfloor$



Average Case: $\Theta(\log n)$

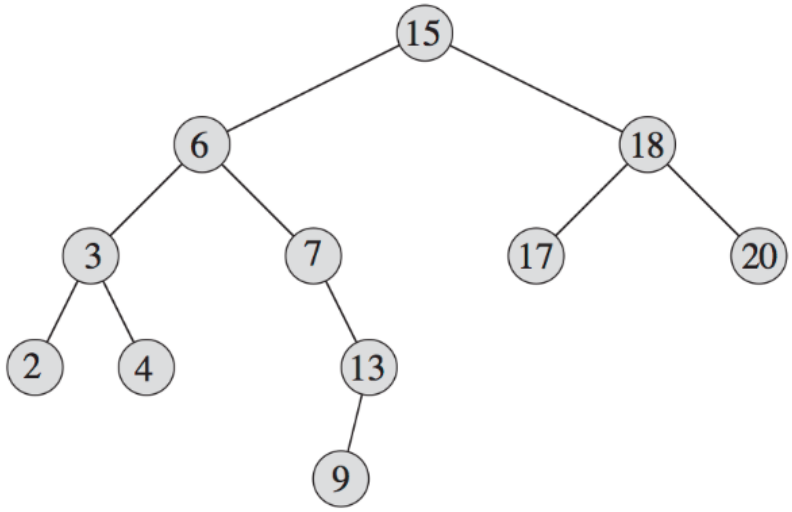
- Falls die Schlüssel in zufälliger Reihenfolge eingefügt werden, ist die Tiefe $O(\log n)$
- Typischer Fall?

Sortieren mit binärem Suchbaum

1. Füge alle Elemente in einen binären Suchbaum ein
2. Lese die Elemente in sortierter Reihenfolge aus
   - Einfachste Lösung: suche und entferne das Minimum
   - Oder besser: suche Minimum und dann $n-1$ Mal *getSuccessor*

Bessere Lösung: Auslesen aller Elemente:

- Rekursiv:
  1. Lese linken Teilbaum aus (rekursiv)
  2. Lese Wurzel aus
  3. Lese rechten Teilbaum aus (rekursiv)



Laufzeit bei Tiefe $O(\log n)$:

- Einfügen: $O(n \cdot \log n)$
- Auslesen: $O(n)$

Auslesen eines Teils der Elemente

Gegeben: Schlüssel $x_{\min}$ und $x_{\max}$ $(x_{\min} \leq x_{max})$

Ziel: Gebe **alle Schlüssel x** mit $x_{\min} \leq x \leq x_{\max}$ aus.

Bearbeite Teilbaum von u

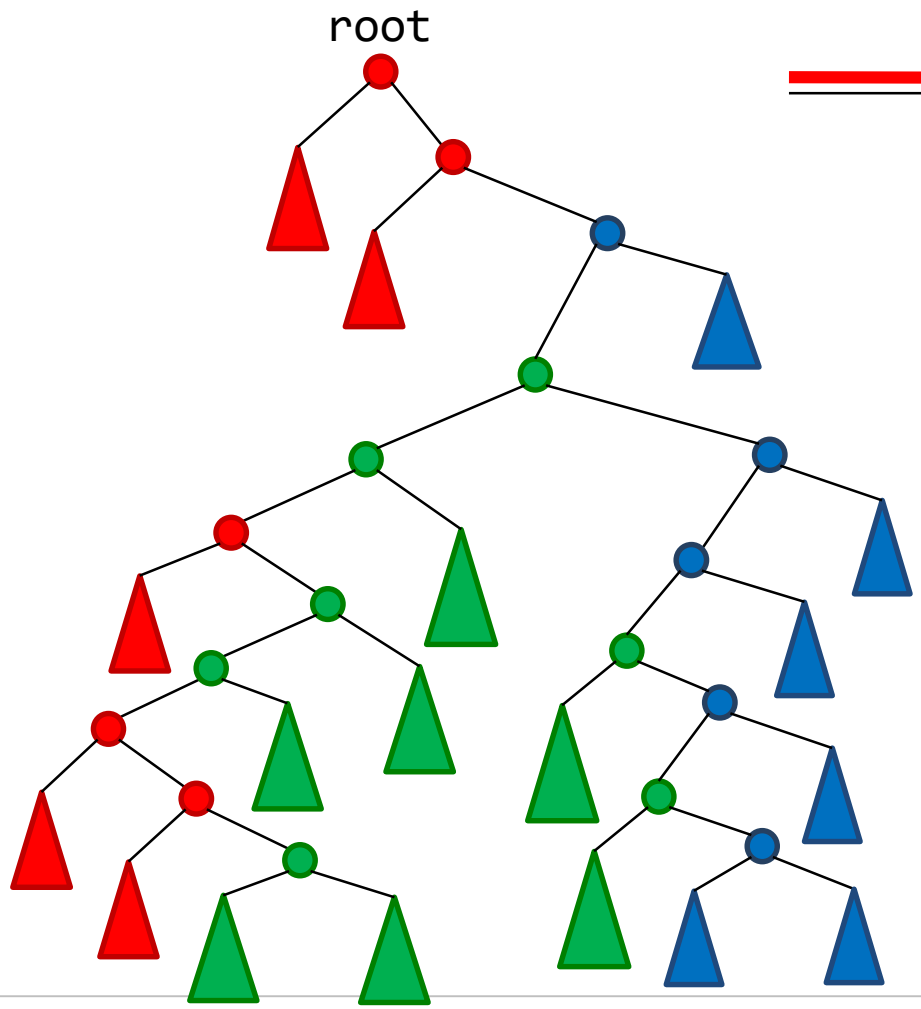

```
getrange(u, xmin, xmax):
    if u is not None:
        if u.key > xmin:
            getrange(u.left, xmin, xmax)
        if (xmin <= u.key) and (u.key <= xmax):
            add u to output
        if u.key < xmax:
            getrange(u.right, xmin, xmax)
```

- **Annahme:** #Schlüssel im Bereich $[x_{\min}, x_{\max}]$ ist k
- **Laufzeit:** ganz sicher $O(n)$ und ganz sicher auch $\Omega(k + \text{Tiefe})$

Auslesen eines Teils der Elemente

Gegeben: Schlüssel $x_{\min}$ und $x_{\max}$ $(x_{\min} \leq x_{max})$

Ziel: Gebe **alle Schlüssel x** mit $x_{\min} \leq x \leq x_{\max}$ aus.



Anzahl besuchte Knoten:

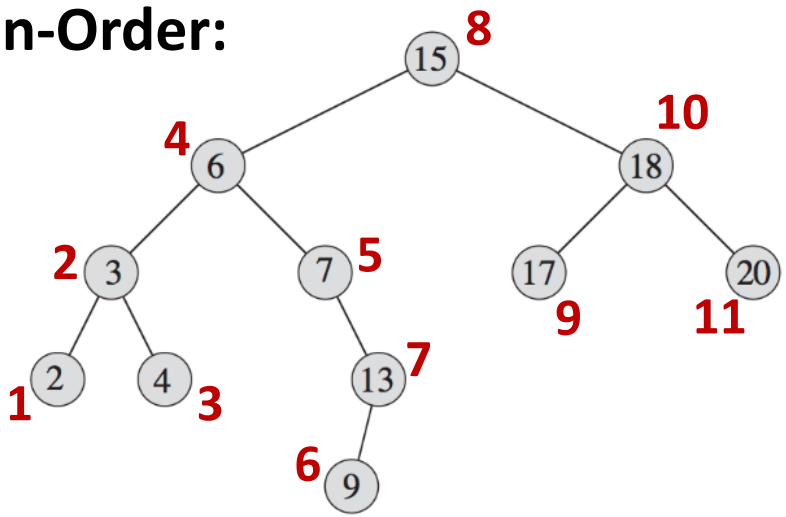
- #grün $= k$
- #rot $\leq$ Tiefe
- #blau $\leq$ Tiefe

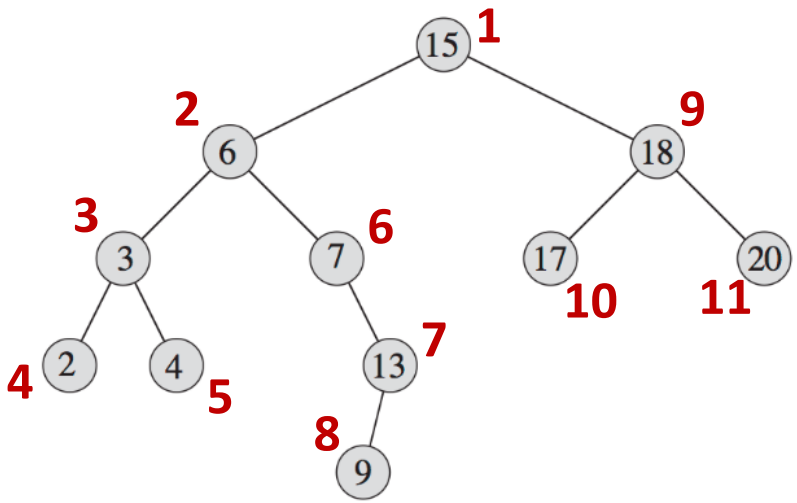
Laufzeit: $O(k + \text{Tiefe})$

Traversieren eines binären Suchbaums

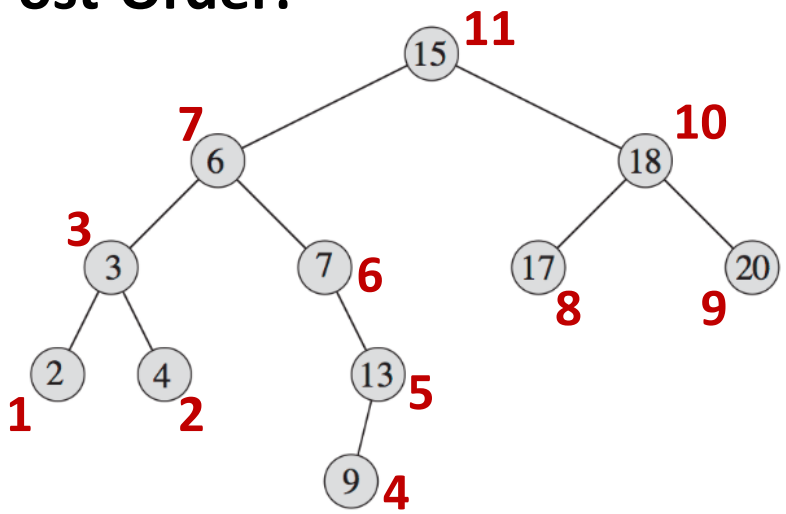
Ziel: Besuche alle Knoten eines binären Suchbaums einmal

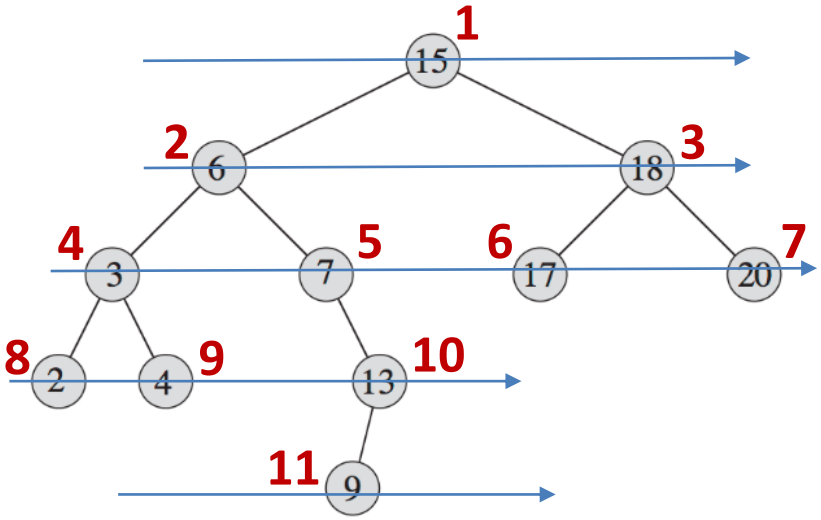
In-Order:





Post-Order:





Traversieren eines binären Suchbaums

UNI FREIBURG

Tiefensuche (Depth First Search / DFS Traversal)

Pre-Order: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20

In-Order: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

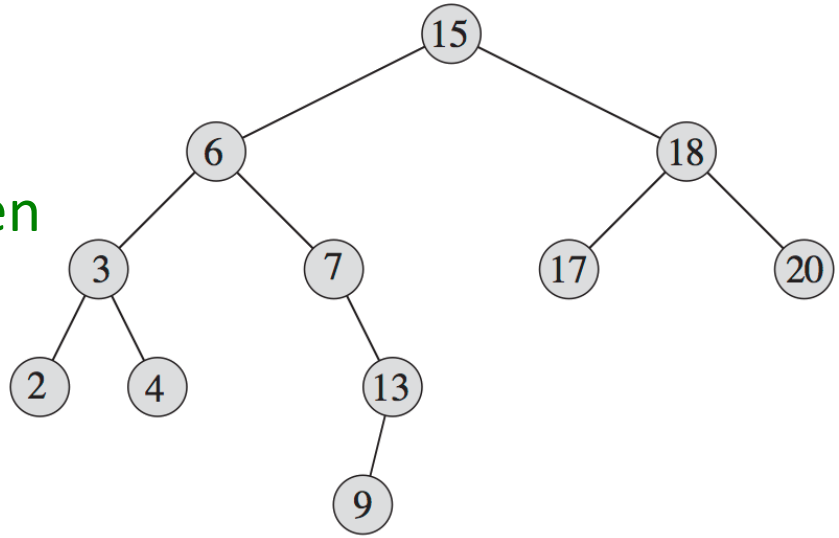
Post-Order: 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

rekursiv

Breitensuche (Breadth First Search / BFS Traversal)

Level-Order: 15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9

- Geht nicht auf die gleiche Art
  $\Longrightarrow$ werden wir gleich noch anschauen



Tiefensuche / DFS Traversierung

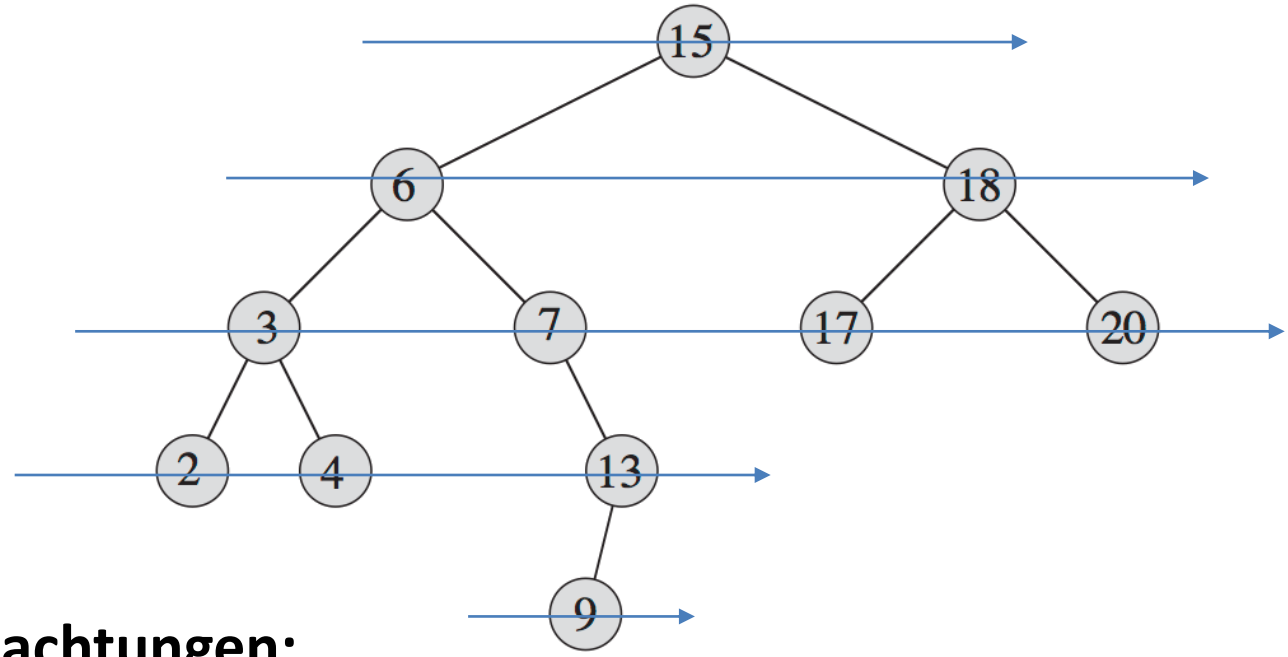
```
preorder(node):
    if node is not None:
        visit(node)
        preorder(node.left)
        preorder(node.right)

inorder(node):
    if node is not None:
        inorder(node.left)
        visit(node)
        inorder(node.right)

postorder(node):
    if node is not None:
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

Breitensuche (BFS Traversierung)

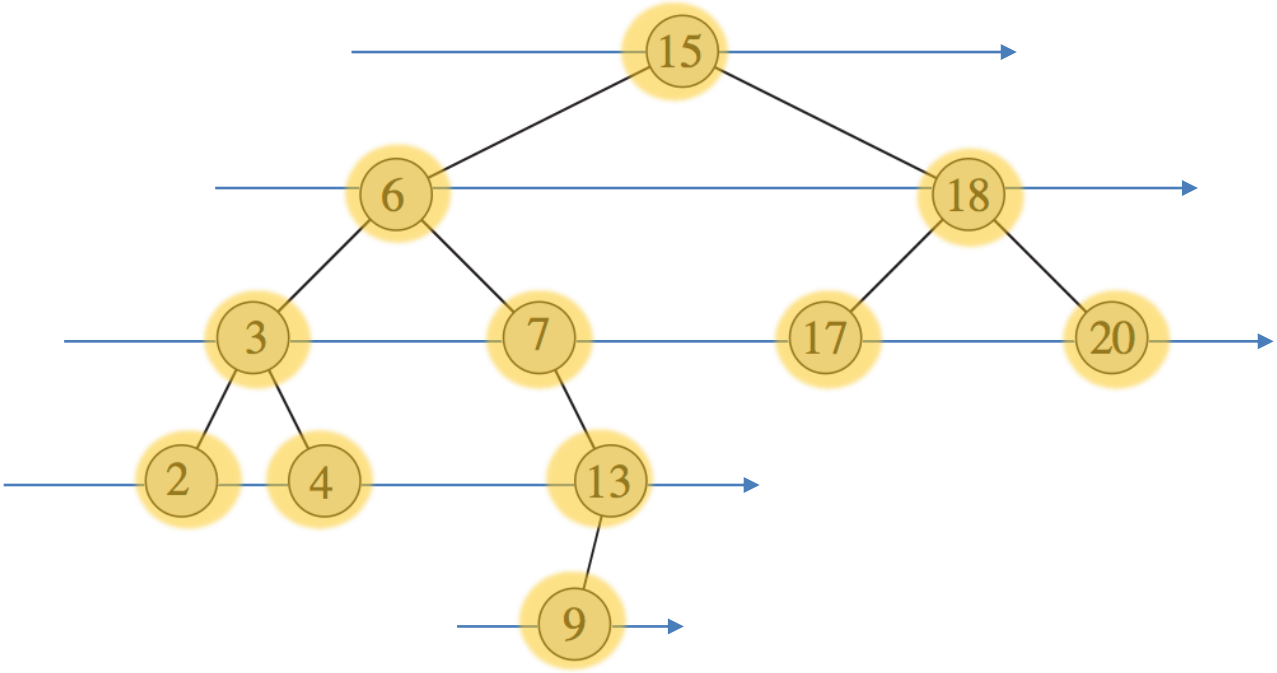
- Funktioniert nicht so einfach rekursiv wie die Tiefensuche



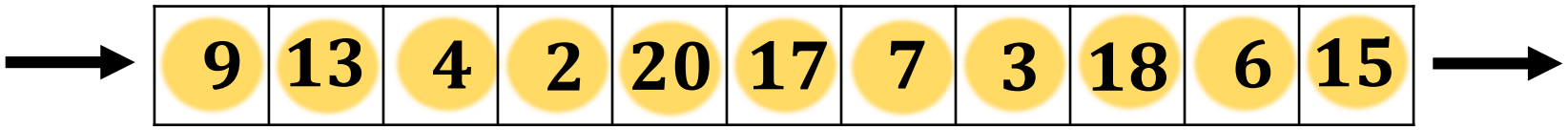
- **Beobachtungen:**
  - Die Wurzel eines Teilbaums wird jeweils vor seinen Kindern besucht
  - Falls Knoten u vor Knoten v besucht wird, dann warden die Kinder von u auch vor den Kindern von v besucht
  - **Idee:** Benutze eine FIFO Queue: wenn u besucht wird, dann werden die Kinder von u in die FIFO Queue eingefügt.

Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche



FIFO Queue:



Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche
- Lösung mit einer Warteschlange:
  - Wenn ein Knoten besucht wird, werden seine Kinder in die Queue eingereiht

```
BFS-Traversal:
    Q = new Queue()
    Q.enqueue(root)
    while not Q.empty():
        node = Q.dequeue()
        visit(node)
        if node.left is not None:
            Q.enqueue(node.left)
        if node.right is not None:
            Q.enqueue(node.right)
```

Analyse Baum-Traversierung

UNI FREIBURG

Tiefensuche:

- Jeder Knoten wird genau einmal besucht
- Kosten pro Knoten: $O(1)$
- **Gesamtzeit** für DFS Traversierung: $\boldsymbol{O(n)}$

Breitensuche:

- Jeder Knoten wird genau einmal besucht
  - Kosten pro Knoten ist linear in der Anzahl Kinder, d.h. $O(1)$ im Binärbaum
  - Jeder Knoten wird genau einmal in die FIFO-Queue eingefügt
- Kosten pro Knoten: $O(1)$
- **Gesamtzeit** für BFS Traversierung: $\boldsymbol{O(n)}$

Anwendungen Tiefensuche I

UNI FREIBURG

In-Order Traversierung:

- Besucht die Elemente eines binären Suchbaums in sortierter Reihenfolge
- Sortieren:
  1. Einfügen aller Elemente
  2. In-Order Traversierung

- Beobachtung: Reihenfolge hängt nur von der Menge der Elemente (Schlüssel) ab, nicht aber von der Struktur des Baums

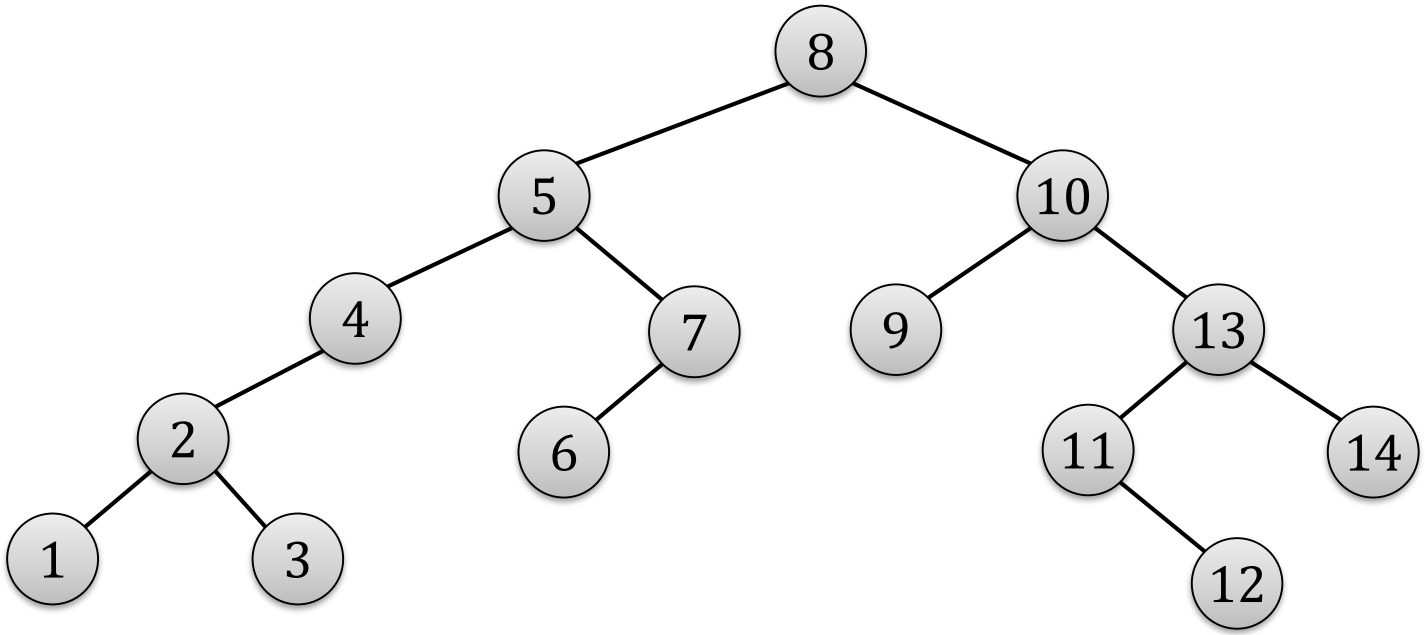
Anwendungen Tiefensuche II

Pre-Order Traversierung:

- Aus der Pre-Order-Reihenfolge lässt sich der Baum in eindeutiger (und effizienter) Weise rekonstruieren
- Geeignet, um den Baum z.B. in einer Datei zu speichern

Beispiel: 8, 5, 4, 2, 1, 3, 7, 6, 10, 9, 13, 11, 12, 14

linker Teilbaum | rechter Teilbaum



Anwendungen Tiefensuche III

UNI FREIBURG

Post-Order Traversierung:

- Löschen eines ganzen binären Suchbaums
- Zuerst muss der Speicher der Teilbäume freigegeben werden, dann kommt die Wurzel

```
delete-tree(node)
    if (node != None)
        delete-tree(node.left)
        delete-tree(node.right)
        delete node
```

Effizienz eines binären Suchbaums

UNI FREIBURG

Worst-Case Laufzeit der Operationen
find, min, max, predecessor, successor, insert, delete:

$$O(\text{Tiefe des Baums})$$

- Im **besten Fall** ist die Tiefe $\log_2 n$
  - Definition Tiefe: Länge des längsten Pfades von der Wurzel zu einem Blatt
- Im **schlechtesten Fall** ist die Tiefe $n - 1$

- Im **durchschnittlichen Fall** ist die Tiefe $O(\log n)$
  - Durchschnittlich heisst hier bei zufälliger Einfügereihenfolge

Nächste Vorlesung: Wie kann man garantieren, dass die Tiefe in einem binären Suchbaum immer Tiefe $O(\log n)$ ist?