Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 9 (16.5.2018)

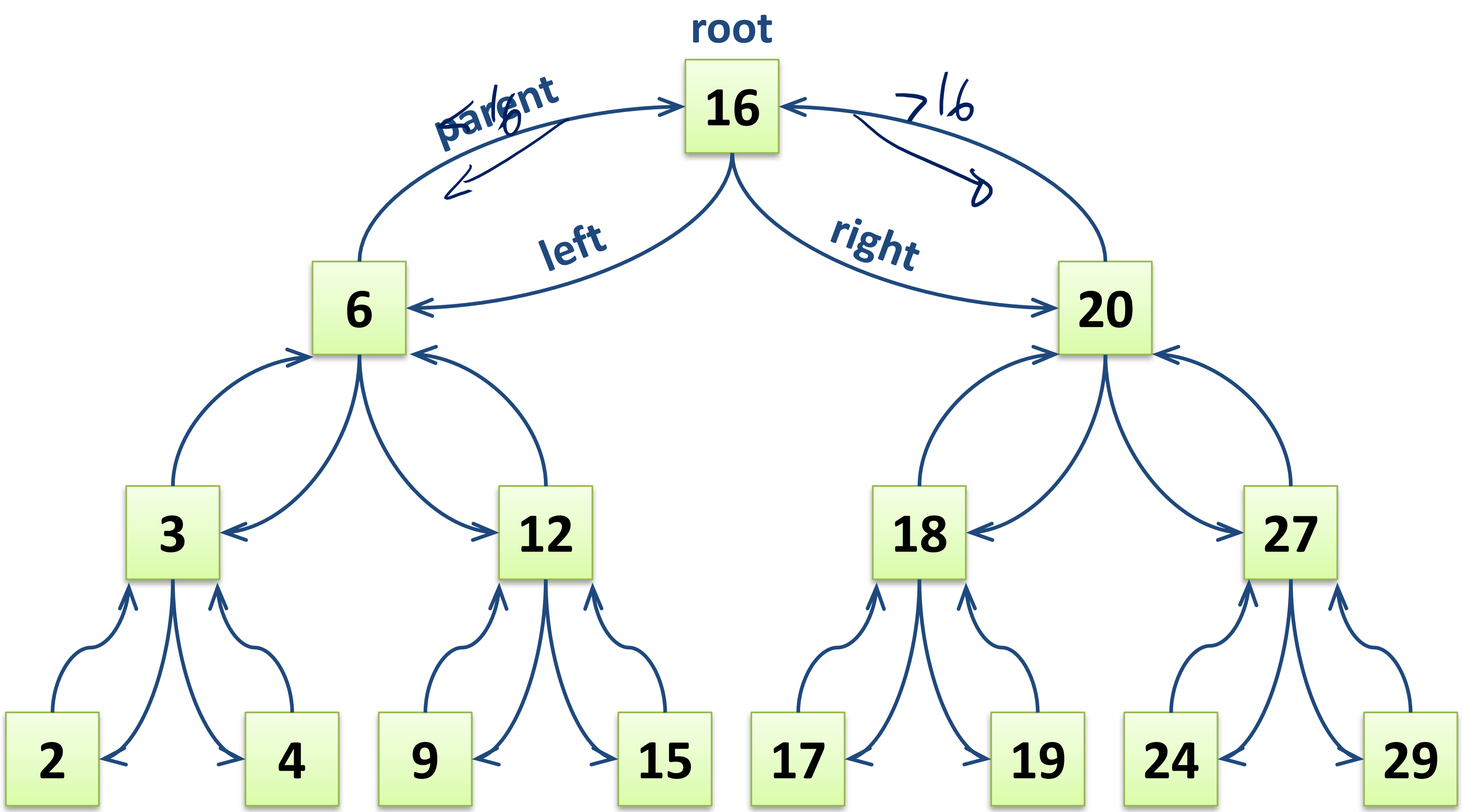
Binäre Suchbäume I

Fabian Kuhn
Algorithmen und Komplexität

UNI FREIBURG

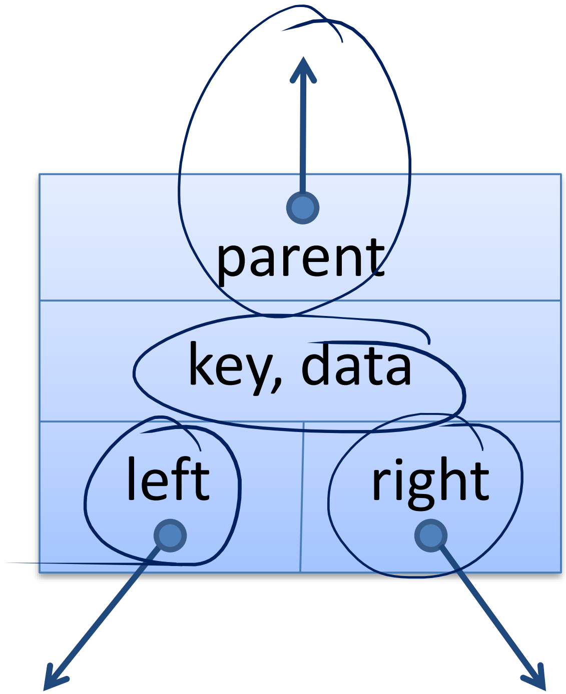
Binäre Suchbäume

- Benutze den Suchbaum der binären Suche als Datenstruktur



Binärer Suchbaum : Elemente

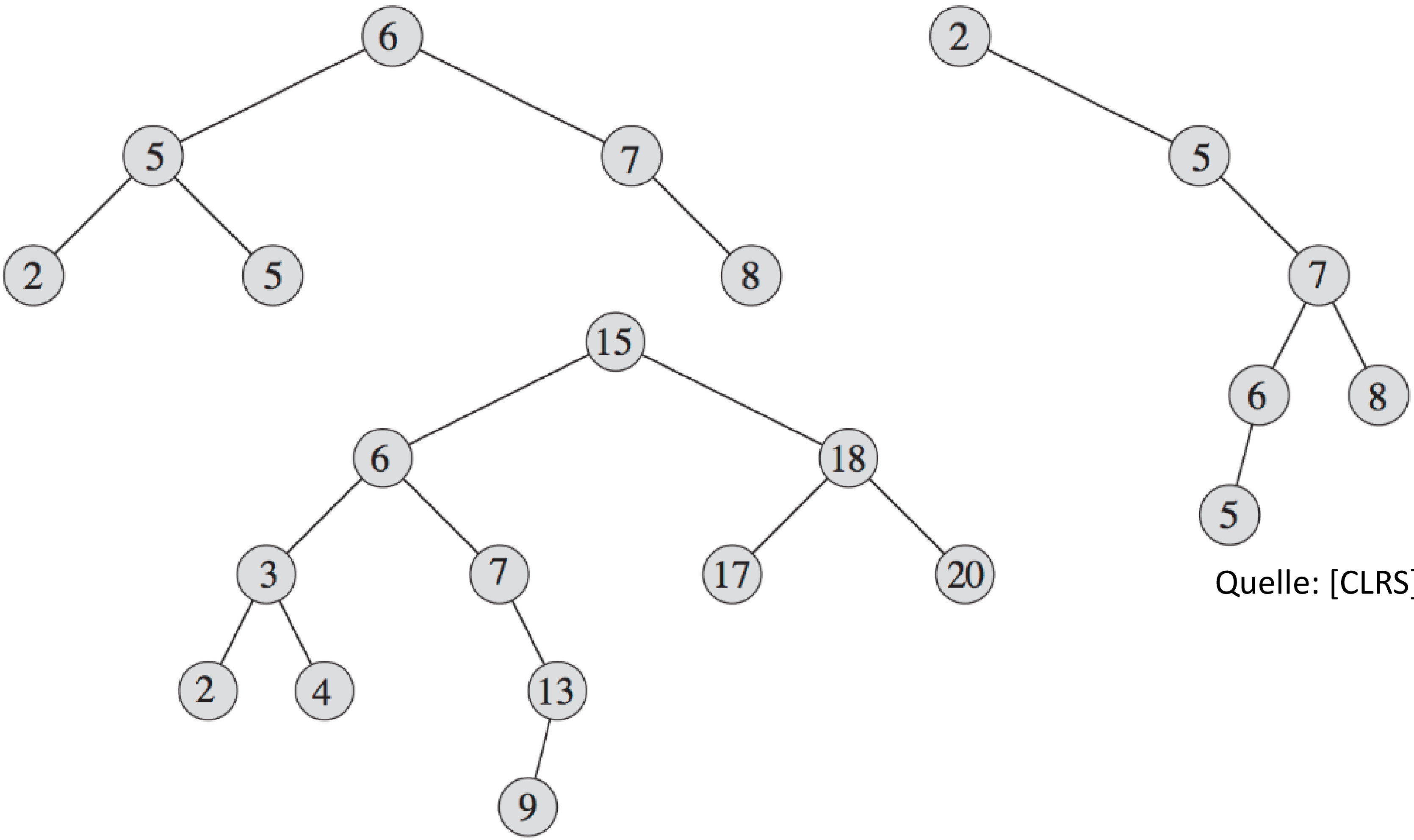
TreeElement:



Implementierung: gleich wie bei den Listen-Elementen

Binäre Suchbäume

- Binäre Suchbäume müssen nicht immer so schön symmetrisch sein…

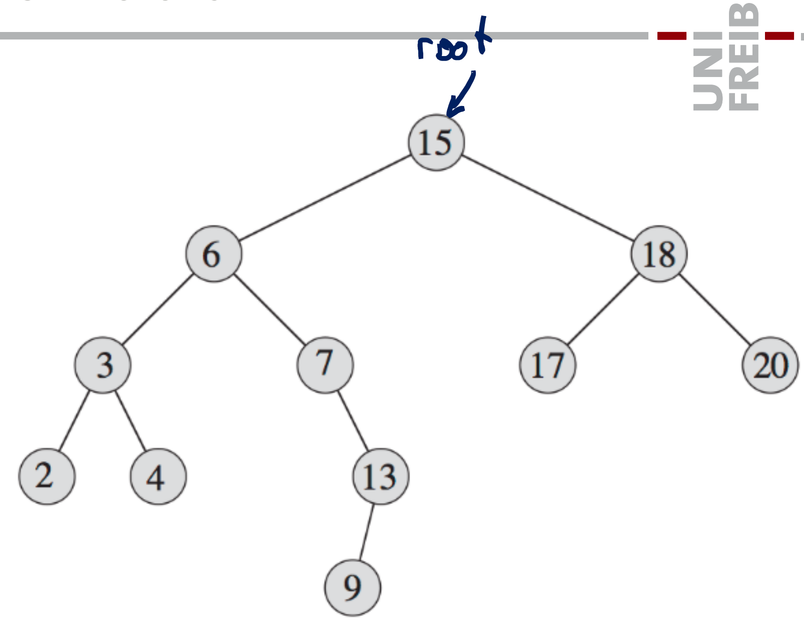


Quelle: [CLRS]

Suche in einem binären Suchbaum

Suche nach Schlüssel x

- Benutze binäre Suche
  (darum heisst's binärer Suchbaum…)



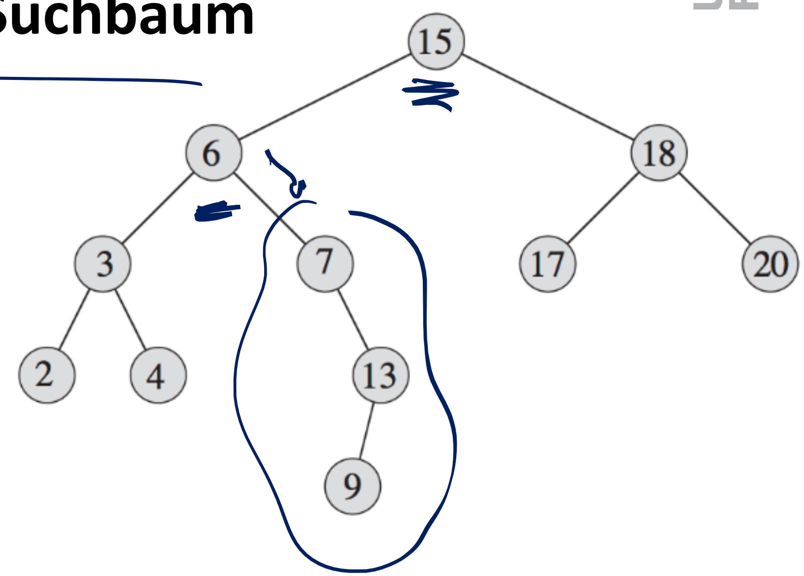

```
current = root
while current is not None and current.key != x:
    if current.key > x:
        current = current.left
    else:
        current = current.right
```

current = None $\to$ x ist nicht im Baum

current.key = x

Suche Minimum / Maximum

Finde kleinstes Element in einem bin. Suchbaum



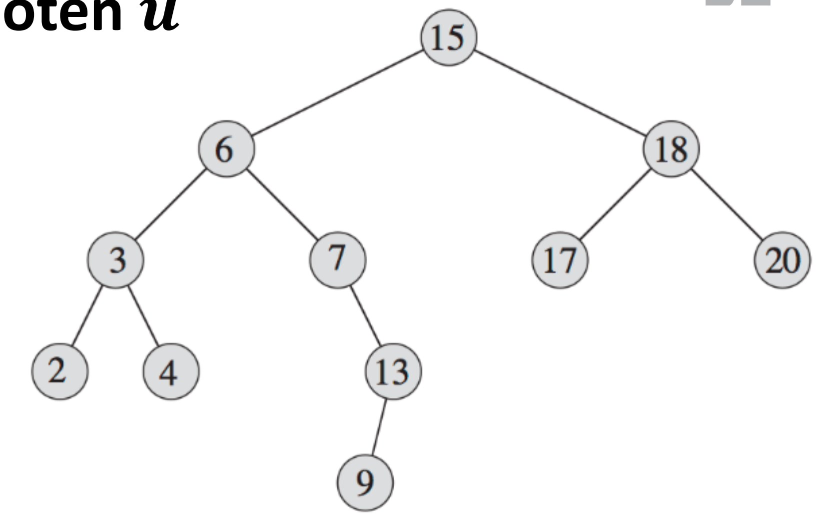
```
current = root
while current.left is not None:
    current = current.left
```

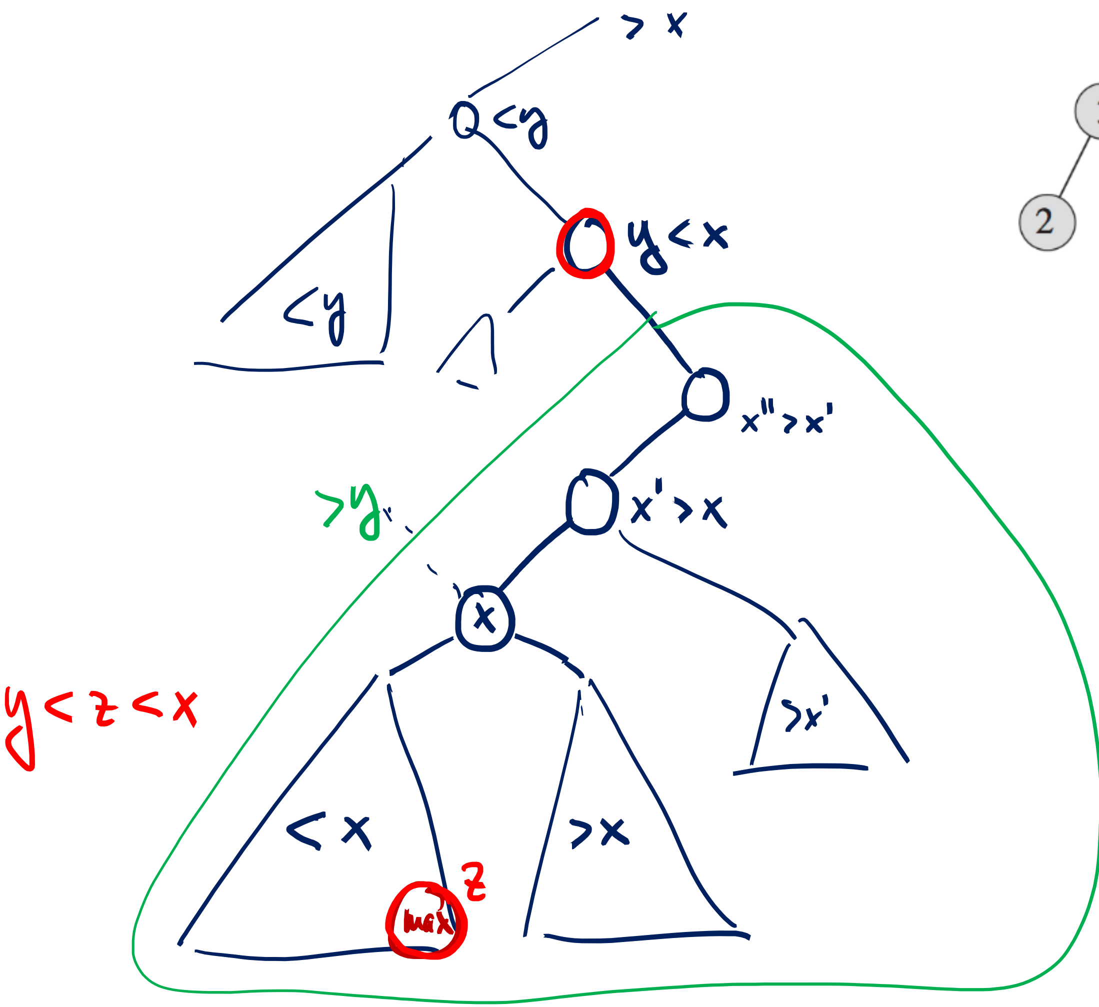
Laufzeit: $O(\text{Tiefe})$

Suche Vorgänger / Nachfolger

von x

Finde Vorgänger / Nachfolger eines Knoten u





falls l. Teilbaum nicht leer:

max. im l. Teilbaum

sonst:

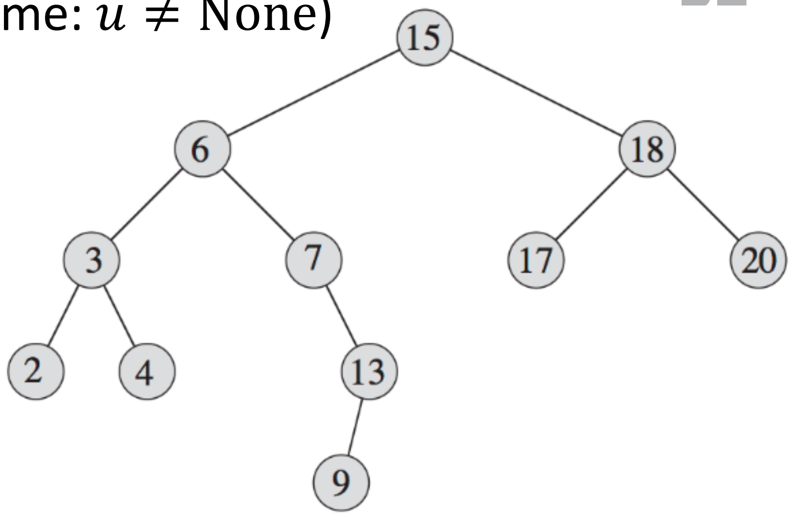
gehe Richtung root bis

current.parent.right = current

return current.parent

Suche Vorgänger / Nachfolger

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

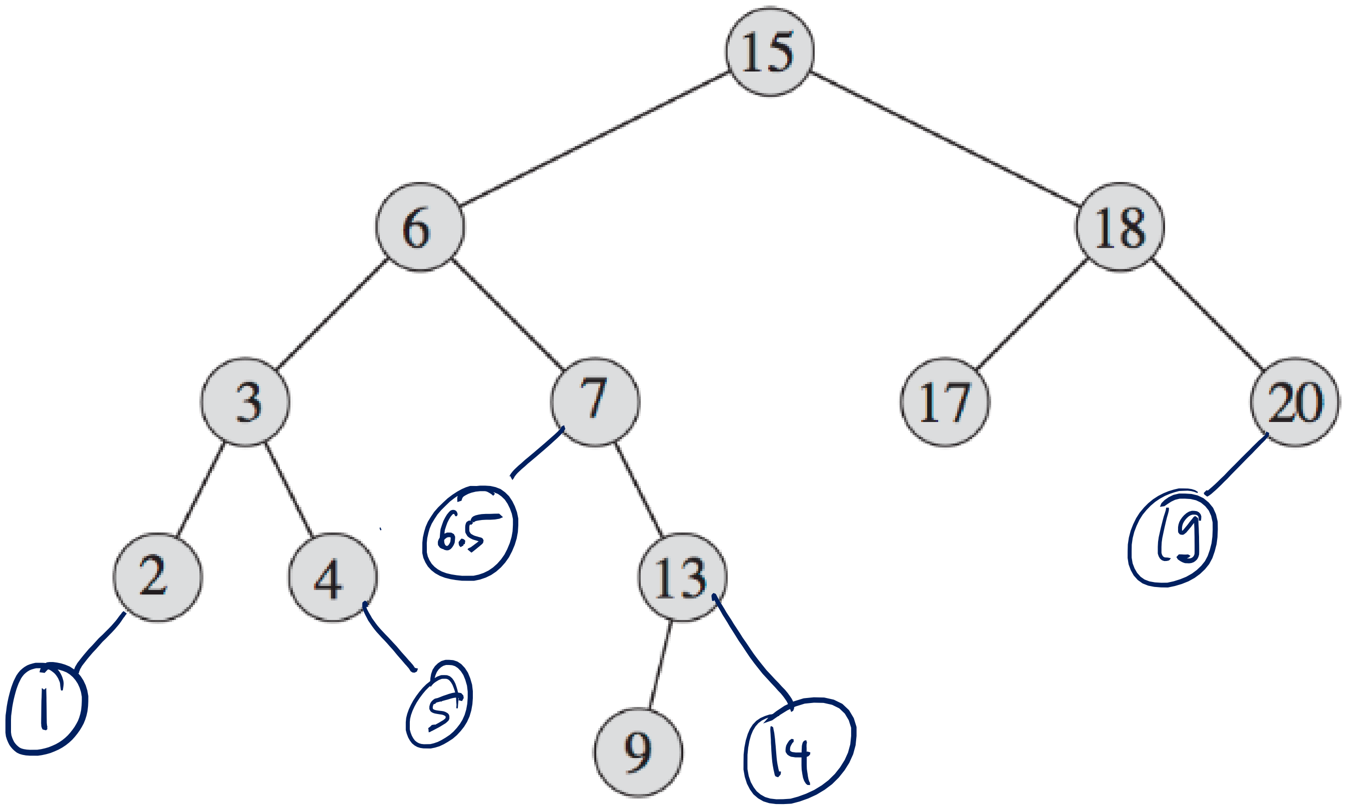


```
if u.right is not None:
    // min in right subtree
    current = u.right
    while current.left is not None:
        current = current.left
    return current
else
    // find first pred. s.t. u is in left subtree
    current = u
    parent = current.parent
    while parent is not None and parent.left != current:
        current = parent
        parent = current.parent
    return parent
```

Laufzeit: O(Tiefe)

Einfügen eines Schlüssels

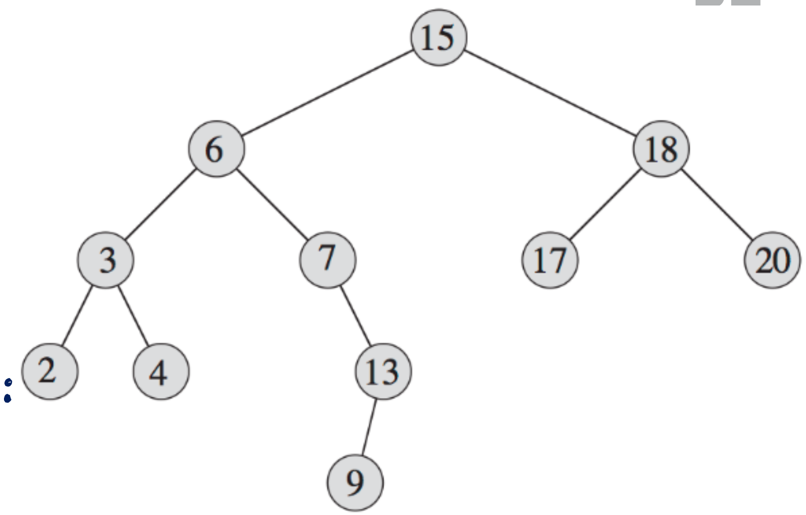
Füge Schlüssel 1, 5, 14, 6.5, 19 ein…



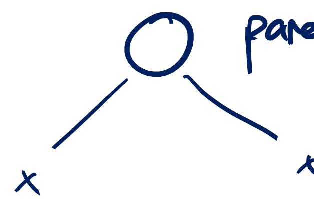
Einfügen eines Schlüssels

(parent = None)

Füge Schlüssel x ein



```
                          parent    l      r
if root is None:
    root = new TreeElm(x, None, None, None)
else:
    current = root
    while current is not None and current.key != x:
        parent = curret
        if x < current.key :
            current = current.left
        else:
            current = current.right
    if current is None:
        if x < parent.key:
            parent.left = new TreeElm(x, parent, None, None)
        else:
            parent.right = new        "
```

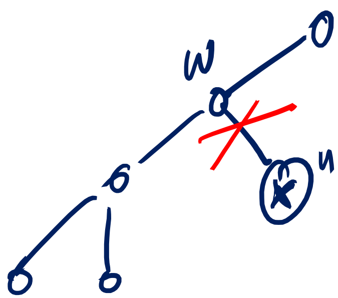


parent

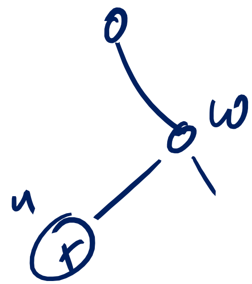
Löschen eines Schlüssels I

Lösche Schlüssel x, einfache Fälle:

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

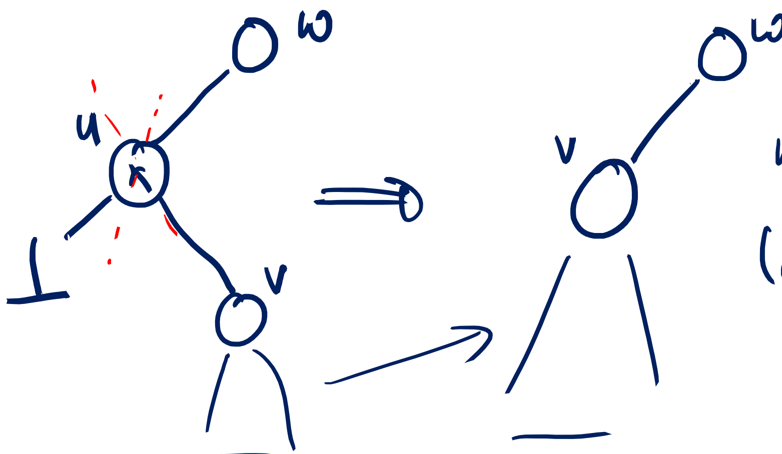


w.right = None
(delete u)

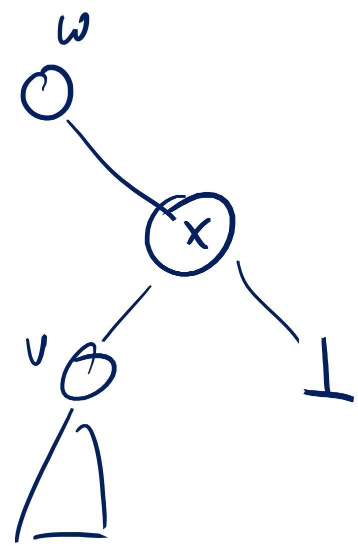


w.left = None
(delete u)

- Knoten mit Schlüssel x hat nur 1 Kind



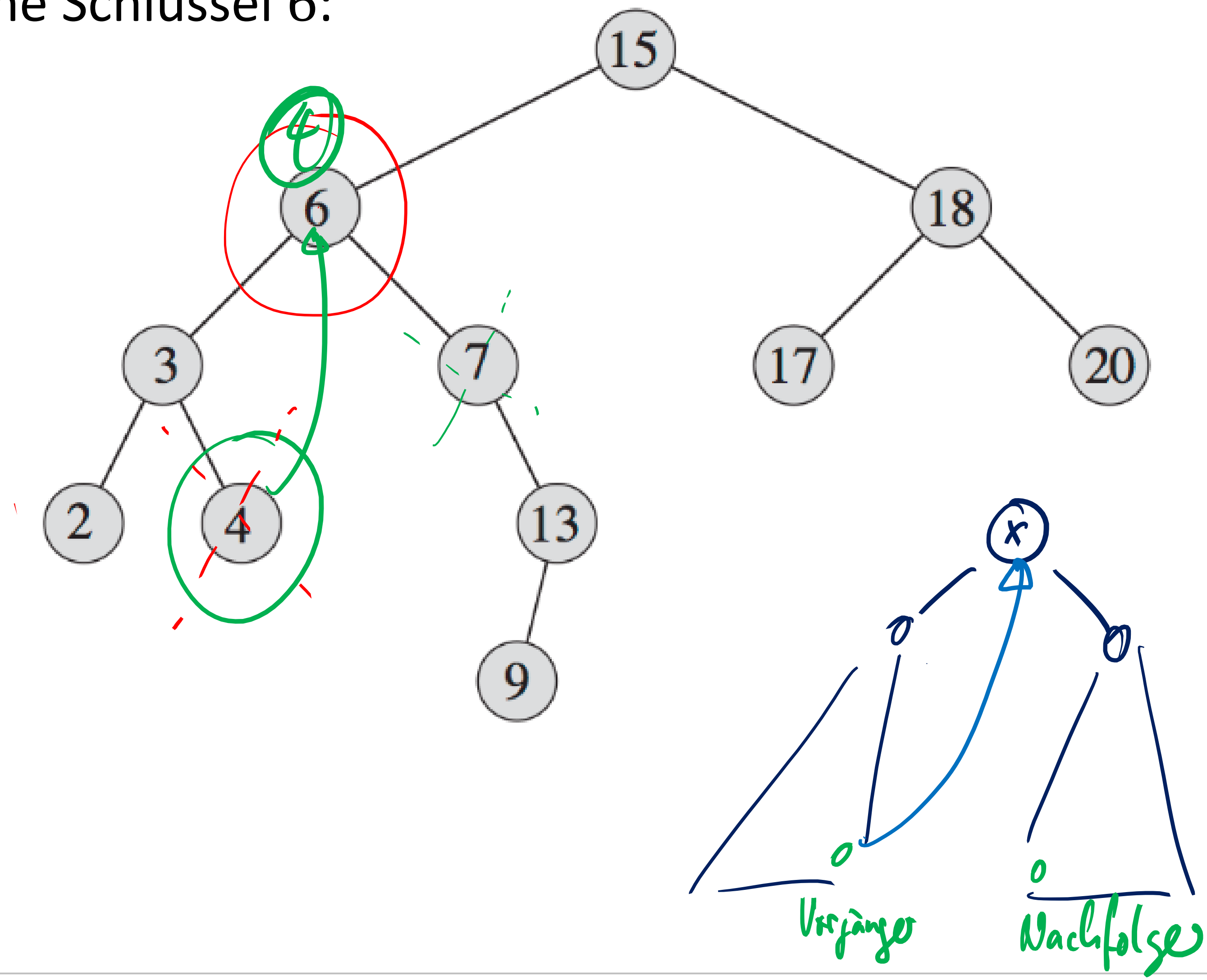
w.left = v
(delete u)



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

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 $v.parent.left =$ None
   – Falls u ein Kind w hat, wird $v.parent.left = w$
3. Falls u zwei Kinder hat, dann bestimme Vorgängerknoten v
   – Funktioniert auch mit Nachfolgerknoten
4. Setze $\underline{u}.key = \underline{v}.key$ und $\underline{u}.data = \underline{v}.data$
5. Lösche Knoten v (gleich, wie oben u gelöscht wird)
   – Knoten v hat höchstens 1 Kind!

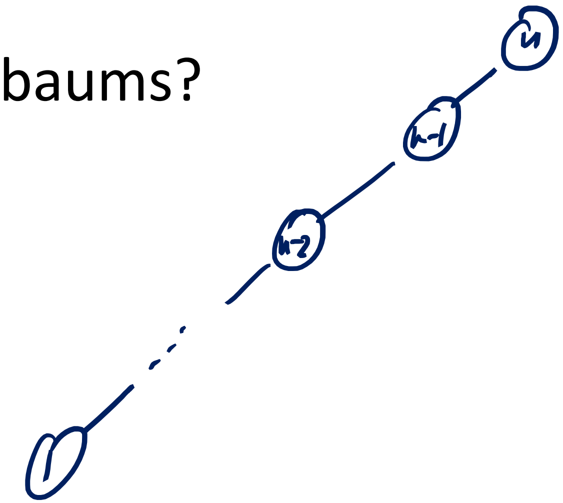
Laufzeiten Binärer Suchbaum

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: $\Theta(\log n)$

typical case / avg. case: $\Theta(\log n)$

z.B. in zufälliger Reihenfolge einfügen

Sortieren mit binärem Suchbaum

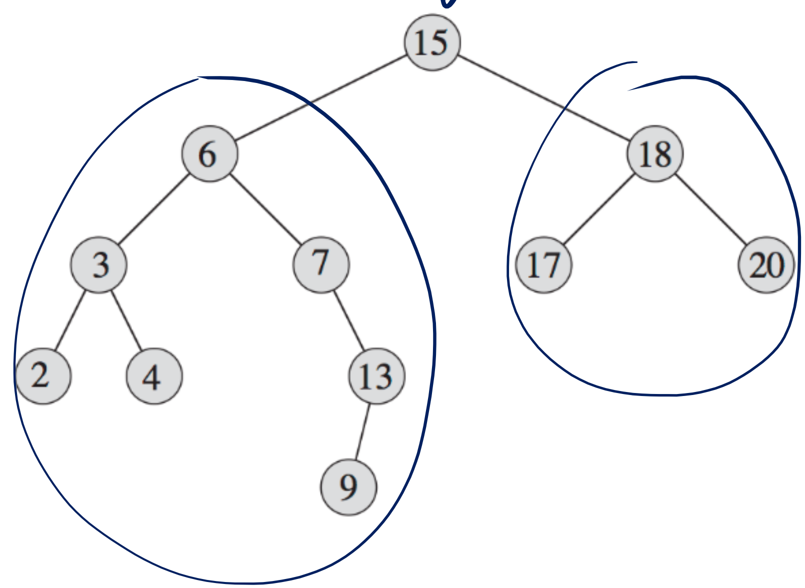
1. Füge alle Elemente in einen binären Suchbaum ein
2. Lese die Elemente in sortierter Reihenfolge aus
   - Einfachste Lösung: suche und entferne das Minimum
   - Oder: suche Minimum und dann $n-1$ Mal *getSuccessor*

falls Tiefe des Baums $O(\log n)$

best case: $O(n \log n)$

Bessere Lösung: Auslesen aller Elemente:

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

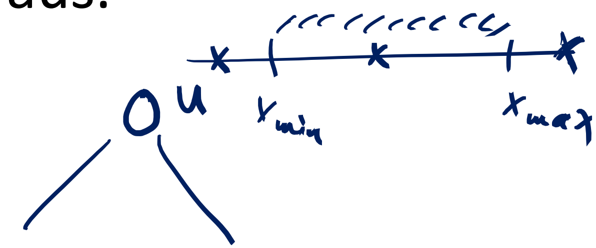



```
sort(u):
    sort(u.left)
    output(u.key)
    sort(u.right)
```

Auslesen eines Teils der Elemente

Gegeben: Schlüssel $x_{\min}$ und $x_{\max}$ $(\underline{x_{\min}} \leq \underline{x_{\max}})$ $[x_{min}, x_{max}]$

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



visit(u):
  if u.key > x_{min}
    visit(u.left)
  if $x_{min} \leq$ u.key $\leq x_{max}$
    output(u.key)
  if u.key < x_{max}
    visit(u.right)

Laufzeit?
- $O(n)$, $\boxed{O(k + \text{Tiefe})}$?
- $\Omega(k + \text{Tiefe})$ (im worst case)

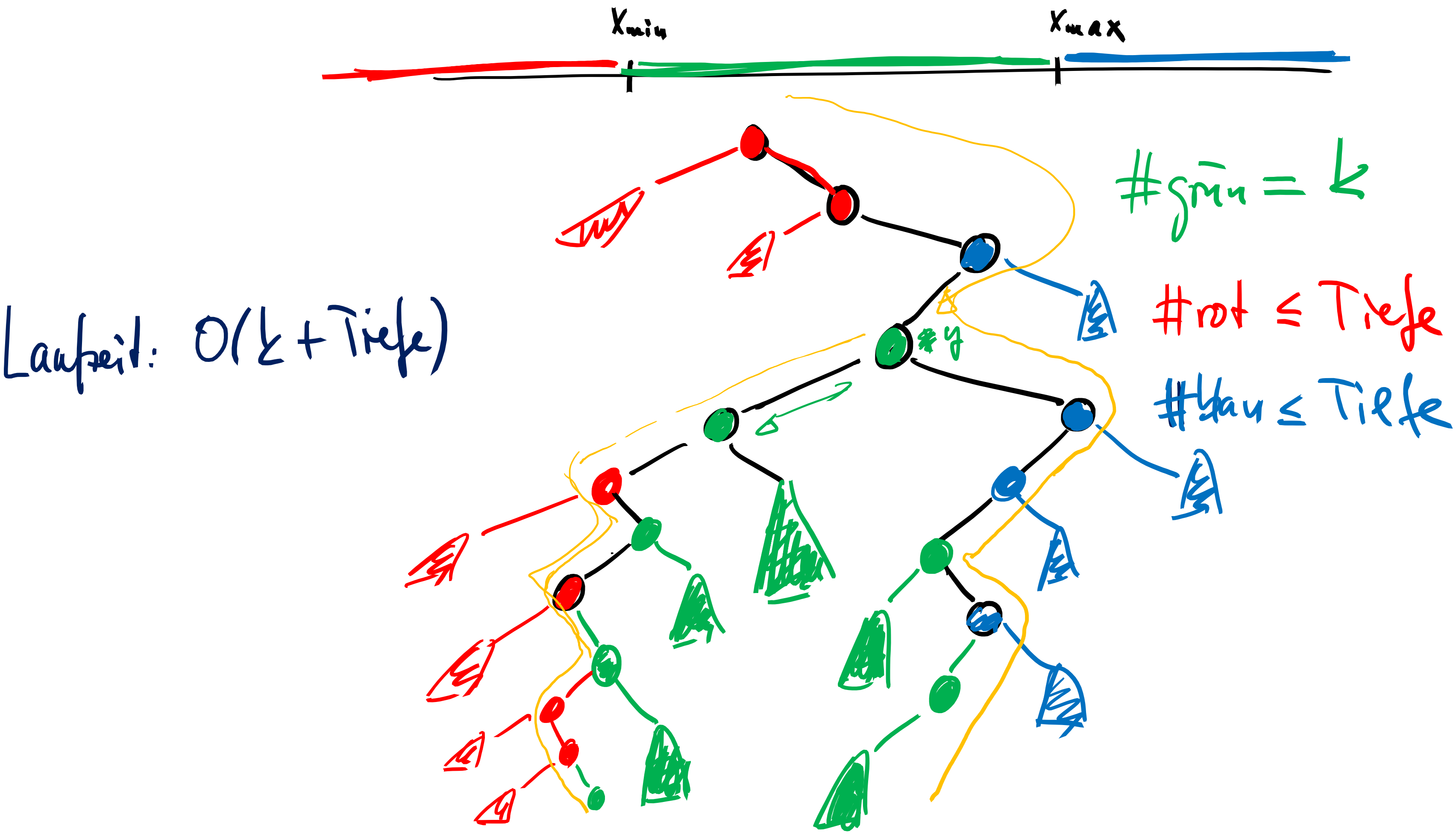
gibt alle Schlüssel im Bereich $[x_{min}, x_{max}]$ aus

Schlüssel in $[x_{min}, x_{max}]$ ist k

Auslesen eines Teils der Elemente

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

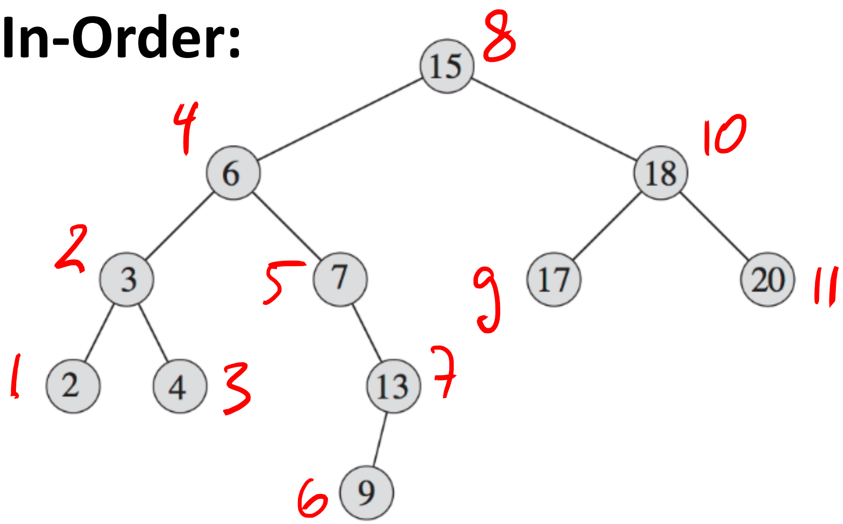
Ziel: Gebe **alle Schlüssel x, $x_{\min} \leq x \leq x_{\max}$** aus.



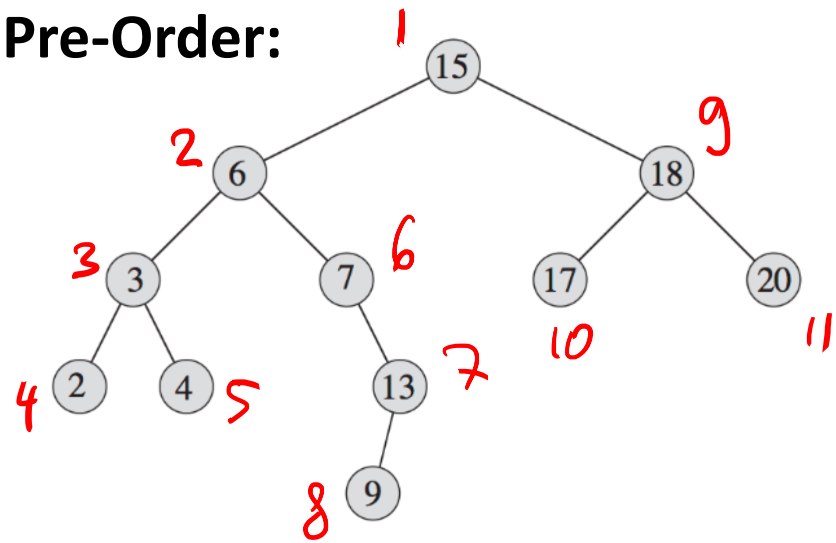
Traversieren eines binären Suchbaums

Ziel: Besuche alle Knoten eines binären Suchbaums einmal

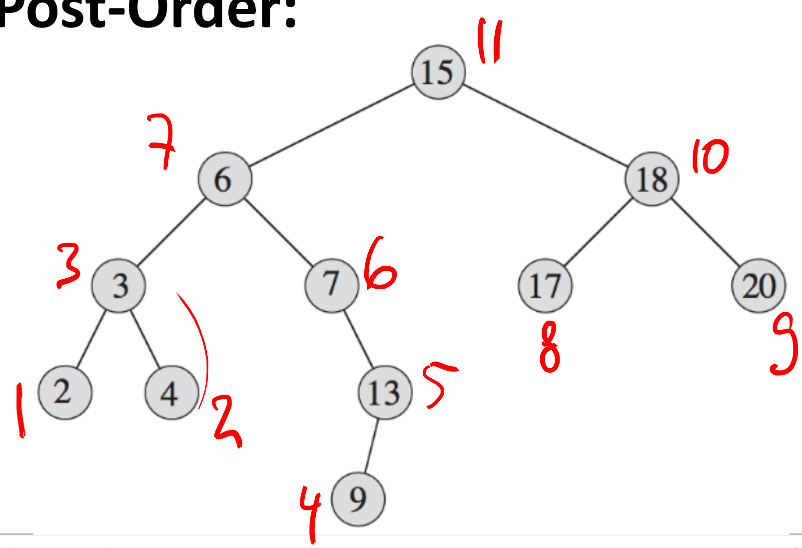
In-Order:



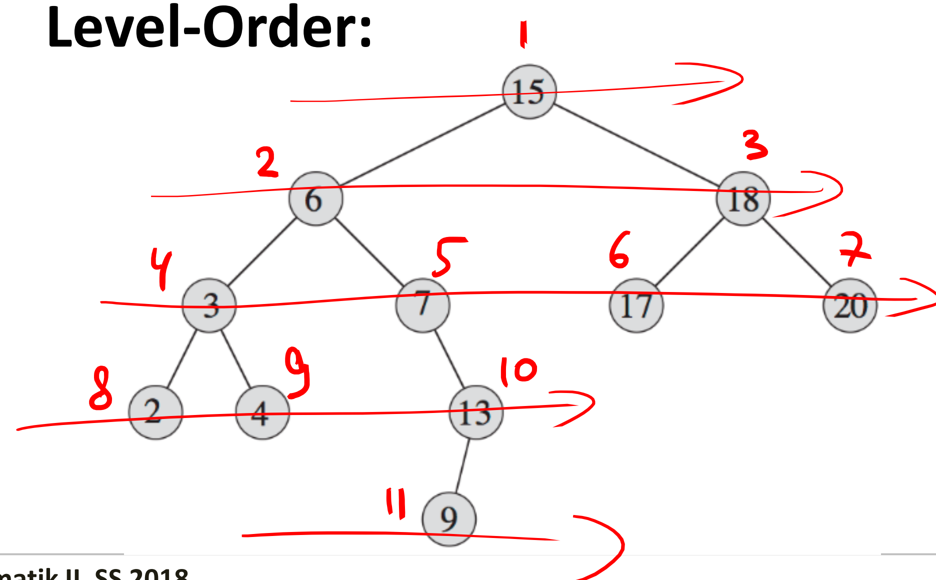
Pre-Order:



Post-Order:



Level-Order:



Traversieren eines binären Suchbaums

Tiefensuche (Depth First Search / DFS Traversal)

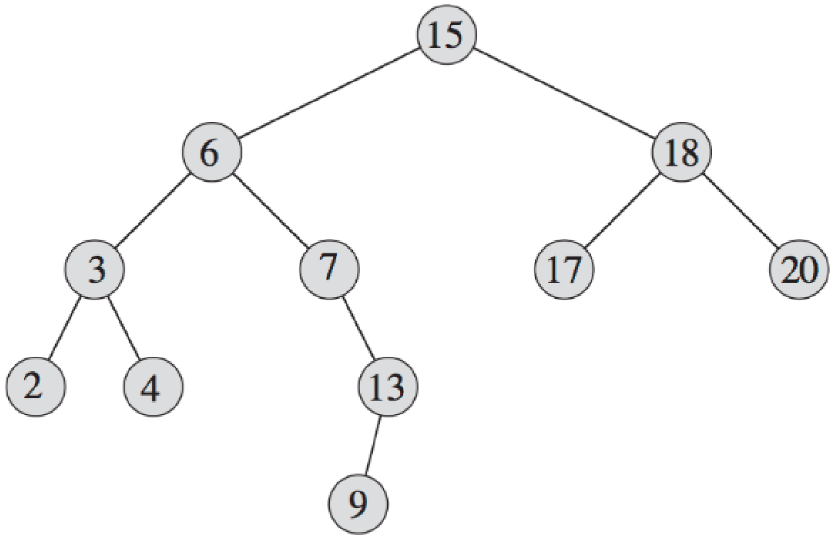
Pre-Order: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20

In-Order: sortiert

Post-Order:

Breitensuche (Breadth First Search / BFS Traversal)

Level-Order:



Tiefensuche / DFS Traversierung

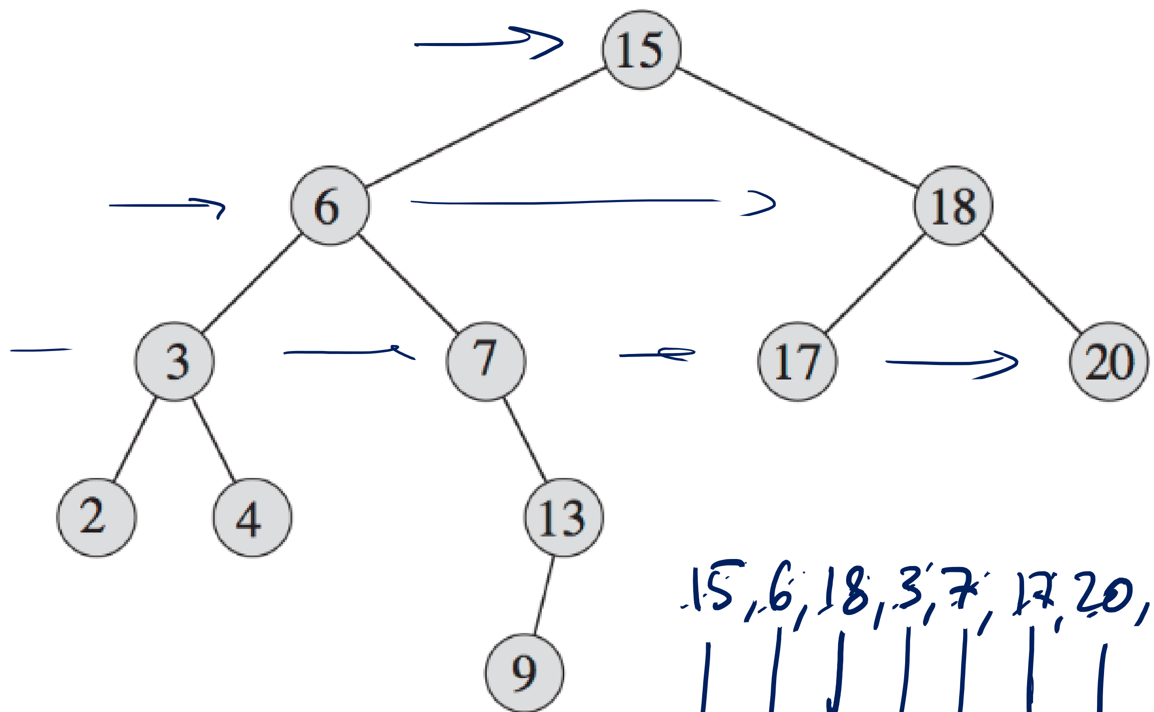
```
preorder(node):
    if node != None
        visit(node)
        preorder(node.left)
        preorder(node.right)

inorder(node):
    if node != None
        inorder(node.left)
        visit(node)
        inorder(node.right)

postorder(node):
    if node != None
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche



15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9

15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9

Warteschlange (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() do
        node = Q.dequeue()
        visit(node)
        if node.left != None
            Q.enqueue(node.left)
        if node.right != None
            Q.enqueue(node.right)
```

Analyse Baum-Traversierung

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 $O(1)$ bei Binärbaum
  - Aber: Jeder Knoten wird genau einmal in die FIFO-Queue eingefügt
- Kosten pro Knoten (~~amortisiert~~): $O(1)$
- **Gesamtzeit** für BFS Traversierung: $\underline{\boldsymbol{O(n)}}$

Anwendungen Tiefensuche I

In-Order Traversierung:

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

(range queries)

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

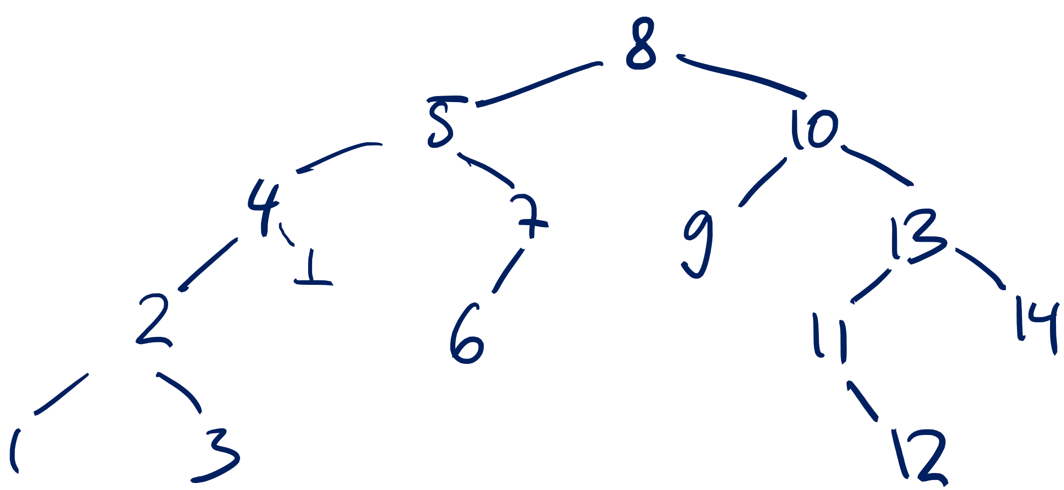
Anwendungen Tiefensuche II

Pre-Order Traversierung:

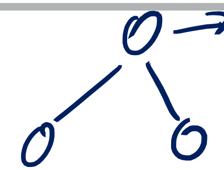
- Aus der Pre-Order-Reihenfolge lässt sich der Baum in eindeutiger (und effizienter) Weise rekonstruieren
- Geeignet, um den Baum z.B. in einer Datei zu speichern

Beispiel: Pre-Order $\boxed{8}, 5, 4, 2, 1, 3, 7, 6, 10, 9, 13, 11, 12, 14$

root l. Teilbaum r. Teilbaum



Anwendungen Tiefensuche III



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
```

Tiefe eines binären Suchbaums

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

$$\boldsymbol{O}(\textbf{Tiefe des Baums})$$

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

Was ist die Tiefe in einem typischen Fall?

- Was ist ein typischer Fall?

Ist es möglich, in einem binären Suchbaum immer Tiefe $O(\log n)$ zu garantieren?

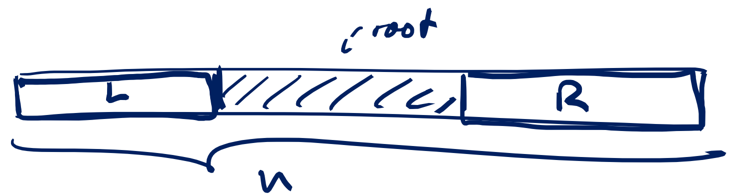
“Typischer” Fall

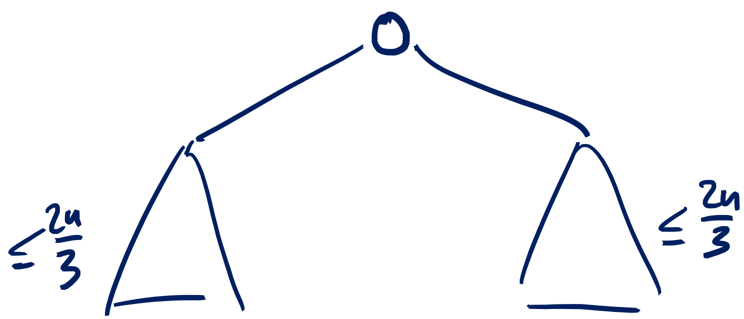
Zufälliger binärer Suchbaum:

- n **Schlüssel werden in zufälliger Reihenfolge eingefügt**

Beobachtung:

- Mit Wahrscheinlichkeit $^1/_3$ haben beide Teilbäume der Wurzel mindestens $^n/_3$ Knoten.





$3 \cdot \log_{3/2} n$

"Typischer" Fall

Zufälliger binärer Suchbaum:

- **n Schlüssel werden in zufälliger Reihenfolge eingefügt**

Beobachtung:

- Mit Wahrscheinlichkeit $1/3$ haben beide Teilbäume der Wurzel mindestens $n/3$ Knoten.
- Analoges gilt auch für alle Teilbäume
- Im Durchschnitt wird deshalb auf jedem 3. Schritt von der Wurzel Richtung eines Blattes, der Teilbaum um einen Faktor $2/3$ kleiner!
- Verkleinern um einen Faktor $2/3$ geht nur $O(\log n)$ oft.
- Tiefe eines zufälligen binären Suchbaums ist deshalb $O(\log n)$
- Genaue Rechnung ergibt:

  Erwartete Tiefe eines zufälligen bin. Suchbaums: $\boxed{\mathbf{4.311 \cdot \ln n}}$

"Typischen" Fall erzwingen?

"Typischer" Fall:

- Falls die Schlüssel in zufälliger Reihenfolge eingefügt werden, hat der Baum Tiefe $O(\log n)$
- Operationen haben Laufzeit $O(\log n)$

Problem:

- Zufällige Reihenfolge ist nicht unbedingt der typische Fall!
- Vorsortierte Werte kann genau so typisch sein
  - Das ergibt einen sehr schlechten binären Suchbaum

Idee:

- Können wir zufällige Reihenfolge erzwingen?
- Schlüssel werden in beliebiger Reihenfolge eingefügt, aber Struktur soll immer wie bei zufälliger Reihenfolge sein!