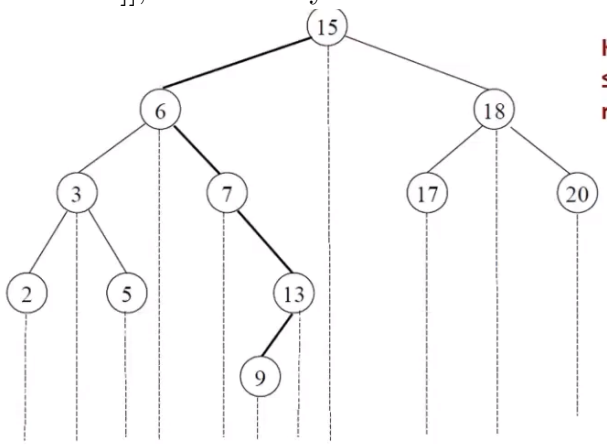
Definition

- [[Binärbäume]], welche in symmetrischer Reihenfolge sortiert sind



Knoten im linken Teilbaum ≤ Wurzel ≤ Knoten im rechten Teilbaum

```
SR(k)
if k≠nil then
    SR(links(k))
    write k
    SR(rechts(k))
```

Zusammenfassung:

- Minimum
- Maximum
- Vorgänger
- Nachfolger
- Einfügen
- Löschen
- Suchen

Alle Operationen in **O(*h*)** Zeit
h ... Baumhöhe

Vorteil: dynamische Lösung des Wörterbuchproblems

Nachteil: Zeiten bis zu Θ(n) (bei entarteten Bäumen).

-

Operationen

Suchen (binäre Suche):

```
SUCHE(b,k)
if k=nil or b=wert(k) then
     write k
else if b<wert(k) then
     SUCHE(b,links(k))
else SUCHE(b,rechts(k))
```

b ... gesuchter Wert
k ... Wurzel des Teilbaums

Aufruf: SUCHE(b, Wurzel(B))

Suchzeit: O(h)

h ... ***Höhe*** des Baumes (= Länge des längsten Astes)

-

Minimum und Maximum:

```
BAUM_MINIMUM(k)
1:  WHILE links(k)≠nil
2:        k ← links(k)
3:  RETURN wert(k)
```

```
BAUM_MAXIMUM(k)
1:  WHILE rechts(k)≠nil
2:        k ← rechts(k)
3:  RETURN wert(k)
```

- **Laufzeit: O(h)**

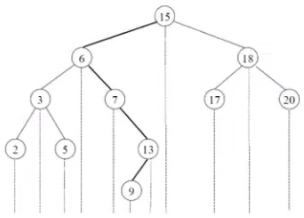
Einfügen:

```
EINFÜGEN(B,k)
1:  y ← nil;  x ← wurzel(B)
2:  WHILE x≠nil
3:     y ← x
4:     IF wert(k) < wert(x) THEN
5:          x ← links(x)
6:     ELSE
7:          x ← rechts(x)
8:  parent(k) ← y
9:  IF y=nil THEN
10:    wurzel(B) ← k
11: ELSE IF wert(k) < wert(y) THEN
12:    links(y) ← k
13: ELSE rechts(y) ← k
```

Fügt den Knoten k in den Binärbaum B ein

Simuliere eine Suche nach k, bis zu einer freien Stelle (x=nil).
y…zukünftiger Parent
Dort fügen wir das Element als Kind von y ein.

Baum B war leer



Vorgänger und **Nachfolger:**

Nachfolger von k: Nächster Knoten in der (nach Werten) sortierten Reihenfolge (= nächstgrößerer Wert im Baum oder gleicher Wert)

```
NACHFOLGER(k)
1:  IF rechts(k) ≠ nil
2:      return BAUM_MINIMUM(rechts(k))
3:  y ← parent(k)
4:  WHILE y ≠ nil  AND k = rechts(y)
5:      k ← y
6:      y ← parent(y)
7:   return(y)
```

Wenn ein **rechtes** Kind existiert, suche das **Minimum** in diesem Teilbaum, ...

... sonst suche den niedrigsten Knoten, bei dem sich k im **linken** Teilbaum befindet

- **Laufzeit: O(h)**

Vorgänger und **Nachfolger:**

Vorgänger von k: Vorgängerknoten in der (nach Werten) sortierten Reihenfolge

```
VORGÄNGER(k)
1:  IF links(k) ≠ nil
2:      return BAUM_MAXIMUM(links(k))
3:  y ← parent(k)
4:  WHILE y ≠ nil  AND k = links(y)
5:      k ← y
6:      y ← parent(y)
7:   return(y)
```

Wenn ein **linkes** Kind existiert, suche das **Maximum** in diesem Teilbaum, ...

... sonst suche den niedrigsten Knoten, bei dem sich k unter dem **rechten** Kind befindet

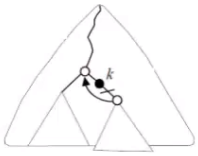
- **Laufzeit: O(h)**

Entfernen:

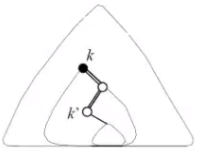
Suchen ⇒ Knoten *k*

a) *k* ist Blatt: abhängen

b) *k* hat nur ein Kind:
Teilbaum von diesem Kind an Parent(*k*) anhängen



c) *k* hat 2 Kinder: Finde *k'*: nächster Knoten in der sortierten Knotenfolge
(***Nachfolger:*** *gehe einmal rechts, dann immer links)*
→ *k'* hat kein linkes Kind!
Setze WERT(*k*) = WERT(*k'*)
Entferne *k'*, (Fall a oder b)



- **Laufzeit: O(h)**

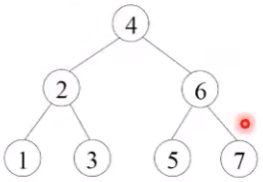
Aufbau eines sortierten Baumes

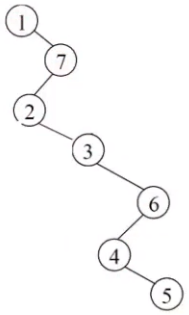
- wiederholtes Einfügen

Binärbaum hängt von der Reihenfolge der Elemente ab

$T(n) \in \Theta(h \cdot n)$

$T(n) \in \Theta(n^2)$, wenn $h \in \Theta(n)$





Einige Reihenfolgen liefern **entartete** Bäume (= Listen)

Fügt man randomisiert ein, ist $E[h] = \Theta(\log n)$