Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG
Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität
Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2020/21
16. Vorlesung

Augmentieren von Datenstrukturen

Prof. Dr. Alexander Wolff *Lehrstuhl für Informatik I*

Zwischentest II: Do, 21. Jan., 8:30 – 10:00

- Zufallsexperimente, (Indikator-) Zufallsvariable, Erwartungswert
- (Randomisiertes) QuickSort
- Untere Schranke für WC-Laufzeit von vergleichsbasierten Sortierverfahren
- Linearzeit-Sortierverfahren
- Auswahlproblem (Median): randomisiert & deterministisch
- Elementare Datenstrukturen
- Hashing
- Binäre Suchbäume (Rot-Schwarz-Bäume noch nicht)

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel
- Hashtabelle
- Heap
- binärer Suchbaum

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel
- Hashtabelle
- Heap
- binärer Suchbaum

Allerdings gibt es viele Situationen, wo keine davon genau passt.

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel
- Hashtabelle
- Heap
- binärer Suchbaum

Allerdings gibt es viele Situationen, wo keine davon genau passt.

Herangehensweise: *Augmentieren* von Datenstrukturen

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel
- Hashtabelle
- Heap
- binärer Suchbaum

Allerdings gibt es viele Situationen, wo keine davon genau passt.

Herangehensweise: *Augmentieren* von Datenstrukturen,

d.h. wir verändern Datenstrukturen,
indem wir extra Information
hinzufügen und aufrechterhalten.

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?
   – Summe der Elemente (sum)
   – Anzahl der Elemente ($size$)

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?
   – Summe der Elemente (sum)
   – Anzahl der Elemente ($size$)
3. Aufwand zur Aufrechterhaltung der Extrainformation?

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?
   – Summe der Elemente (sum)
   – Anzahl der Elemente ($size$)
3. Aufwand zur Aufrechterhaltung der Extrainformation?
   – konstanter Aufwand beim Einfügen und Löschen

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?
   – Summe der Elemente (sum)
   – Anzahl der Elemente ($size$)
3. Aufwand zur Aufrechterhaltung der Extrainformation?
   – konstanter Aufwand beim Einfügen und Löschen
4. Implementiere neue Operationen!

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?
   – Summe der Elemente (sum)
   – Anzahl der Elemente ($size$)
3. Aufwand zur Aufrechterhaltung der Extrainformation?
   – konstanter Aufwand beim Einfügen und Löschen
4. Implementiere neue Operationen!
   getMean()
      return $sum/size$

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
   – Liste
2. Welche Extrainformation aufrechterhalten?
   – Summe der Elemente (*sum*)
   – Anzahl der Elemente (*size*)
3. Aufwand zur Aufrechterhaltung der Extrainformation?
   – konstanter Aufwand beim Einfügen und Löschen
4. Implementiere neue Operationen!

   getMean()

   return $sum/size$

   Ähnlich für Standardabweichung $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(a_i - \bar{a})^2}$.

Probieren Sie's!

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

D.h. wir wollen zu jedem Zeitpunkt effizient

- das i.-kleinste Element (z.B. den Median)
- den *Rang* eines Elements

in einer dynamischen Menge bestimmen können.

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

D.h. wir wollen zu jedem Zeitpunkt effizient

- das i.-kleinste Element (z.B. den Median): Elem Select(int i)
- den *Rang* eines Elements: int Rank(Elem e)

in einer dynamischen Menge bestimmen können.

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

D.h. wir wollen zu jedem Zeitpunkt effizient

- das i.-kleinste Element (z.B. den Median): Elem Select(int i)
- den *Rang* eines Elements: int Rank(Elem e)

in einer dynamischen Menge bestimmen können.

Fahrplan:

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

D.h. wir wollen zu jedem Zeitpunkt effizient

- das i.-kleinste Element (z.B. den Median): Elem Select(int i)
- den *Rang* eines Elements: int Rank(Elem e)

in einer dynamischen Menge bestimmen können.

Fahrplan:

1. Welche Ausgangsdatenstruktur?
2. Welche Extrainformation aufrechterhalten?
3. Aufwand zur Aufrechterhaltung?
4. Implementiere neue Operationen!

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

Das dynamische Auswahlproblem

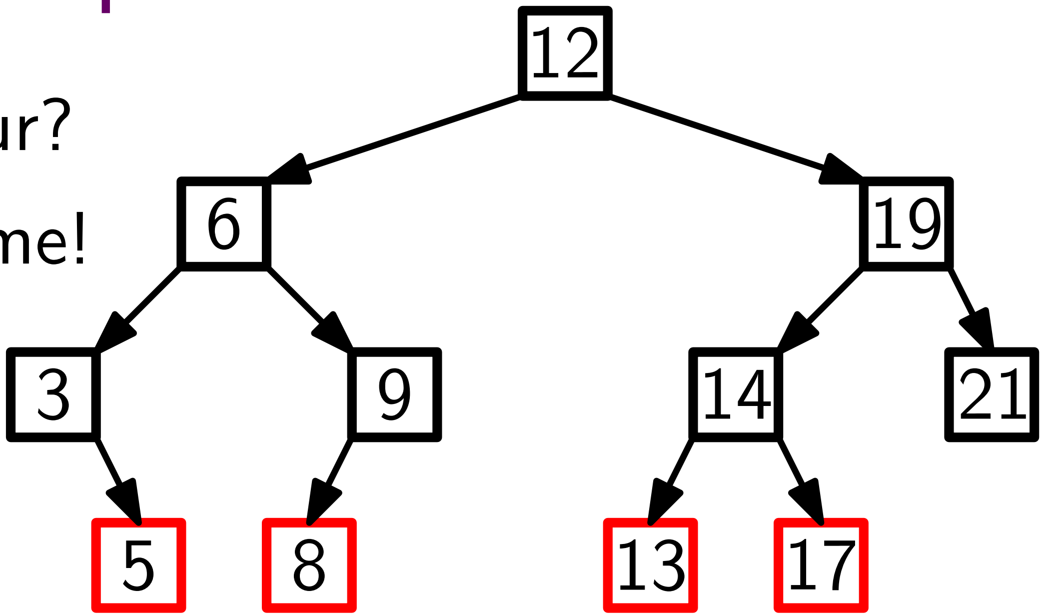
1. Welche Ausgangsdatenstruktur?
   - balancierte binäre Suchbäume!
     z.B. Rot-Schwarz-Bäume

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
     z.B. Rot-Schwarz-Bäume
     $\Rightarrow$ Baumhöhe $h \in O(\log n)$

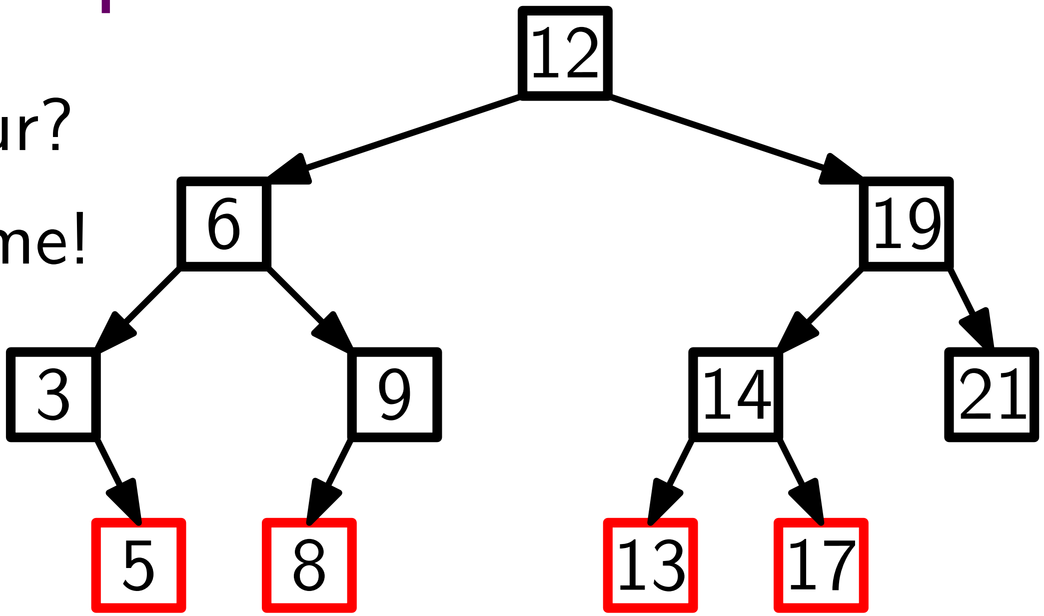
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
     z.B. Rot-Schwarz-Bäume
     $\Rightarrow$ Baumhöhe $h \in O(\log n)$



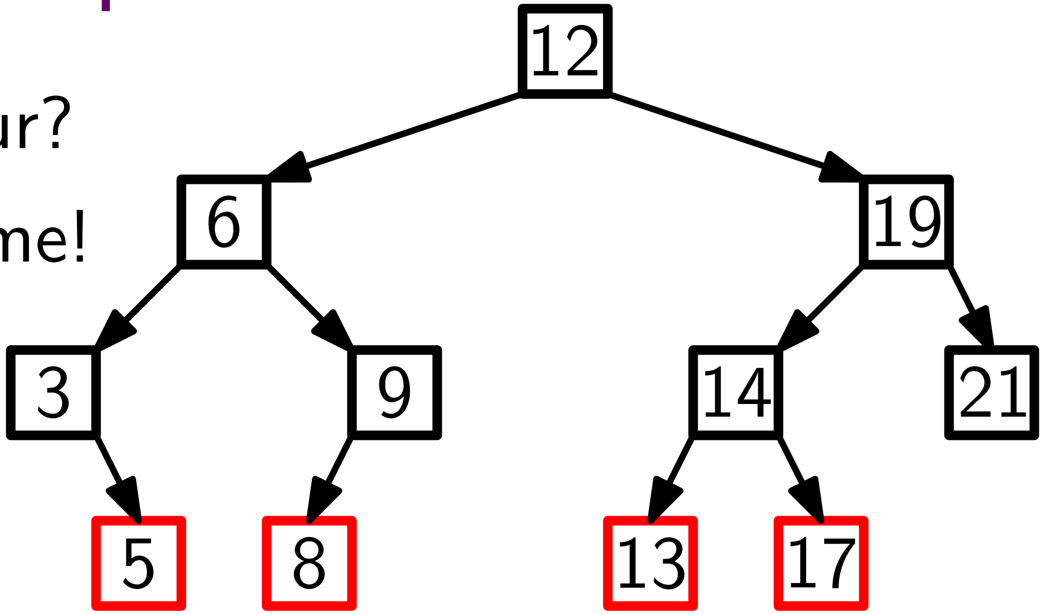
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
     z.B. Rot-Schwarz-Bäume
     $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?

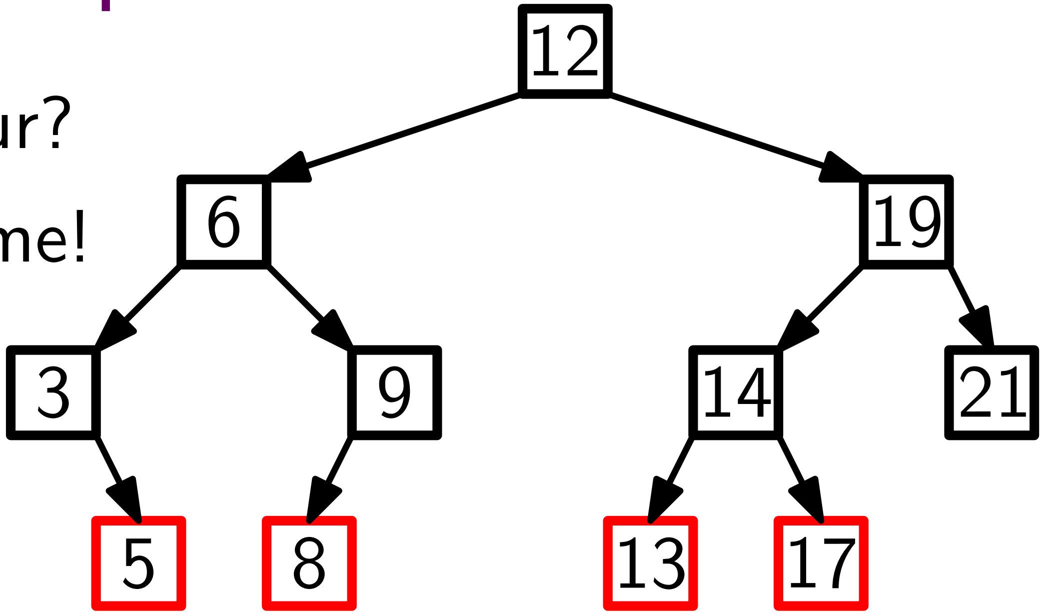
Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

Das dynamische Auswahlproblem

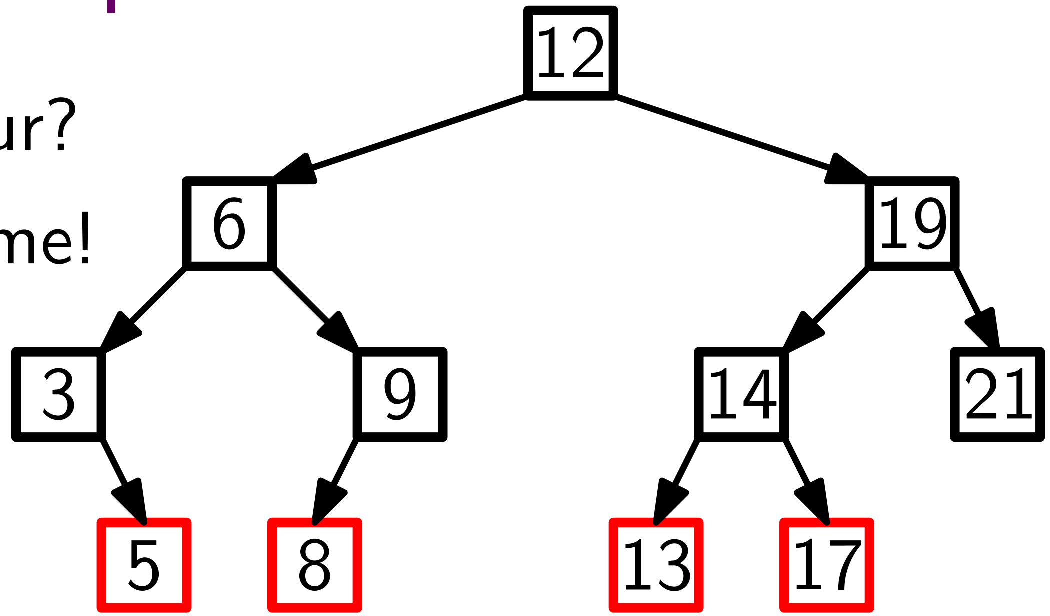


1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere neue Operationen!

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere neue Operationen!

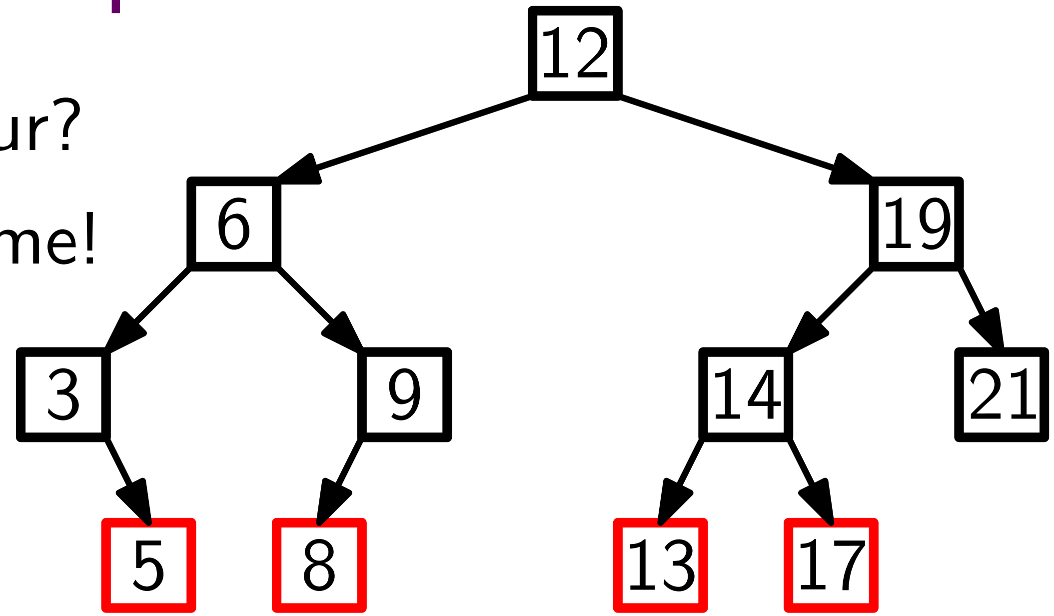
Select(int i):

liefert i.-kleinstes Element

Rank(Node v):

Wievieltes Element ist v?

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere neue Operationen!

Select(int i):

liefert i.-kleinstes Element

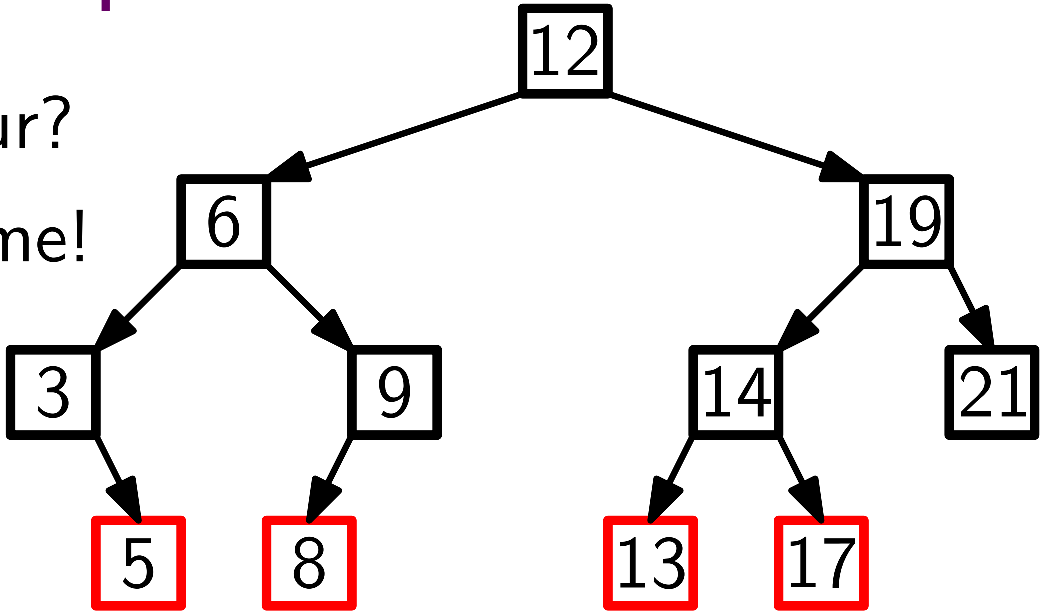
Rank(Node v):

Wievieltes Element ist v?

Aufgabe

Schreiben Sie Pseudocode für Select() und Rank() unter Benutzung von Successor() u. Predecessor()!

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

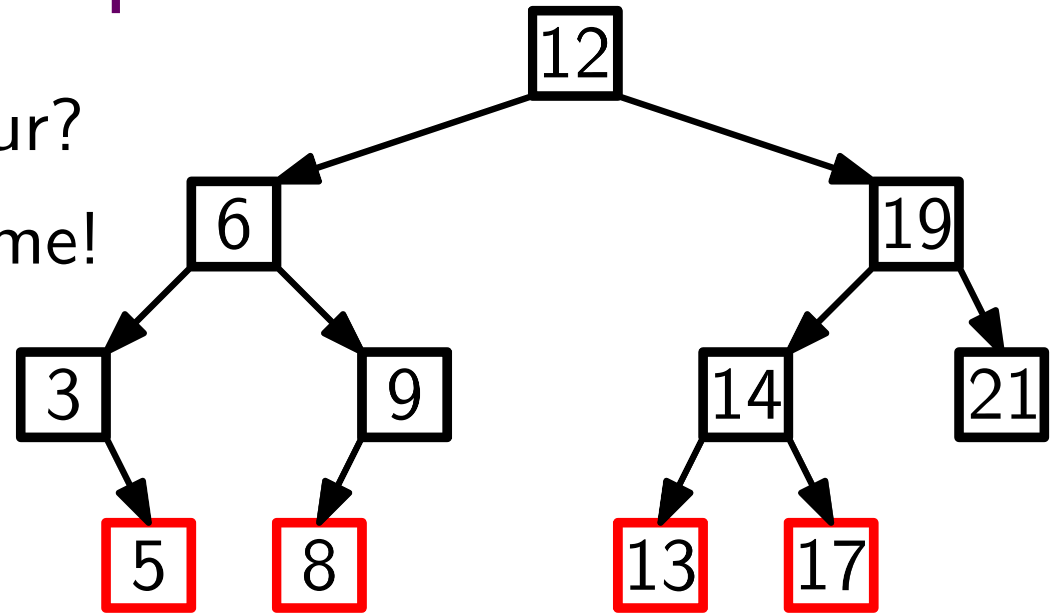
4. Implementiere neue Operationen!

Select(int i):

$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

Rank(Node v):

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere neue Operationen!

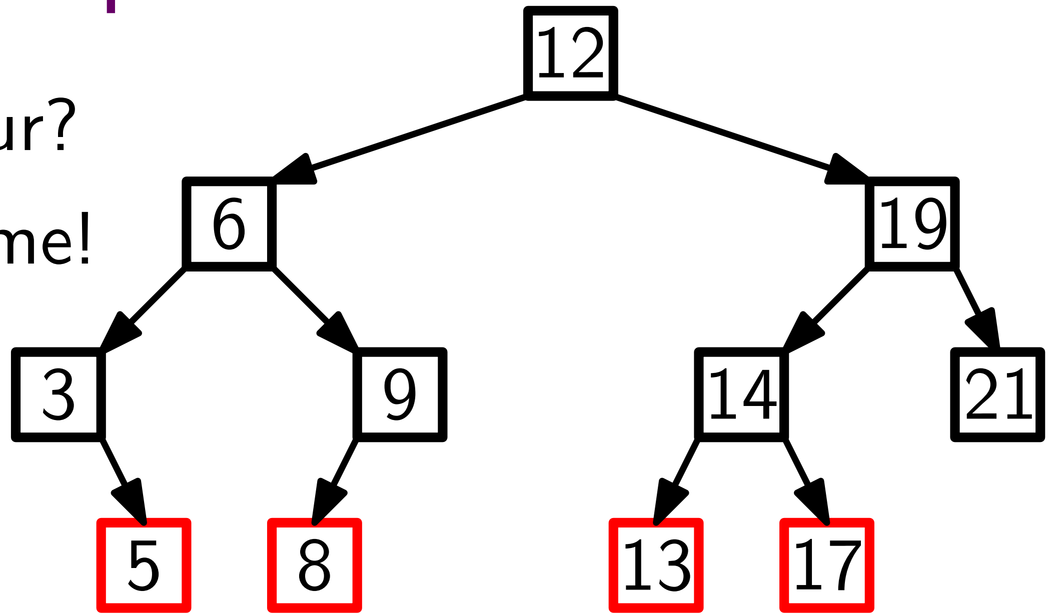
Select(int i):

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v):

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

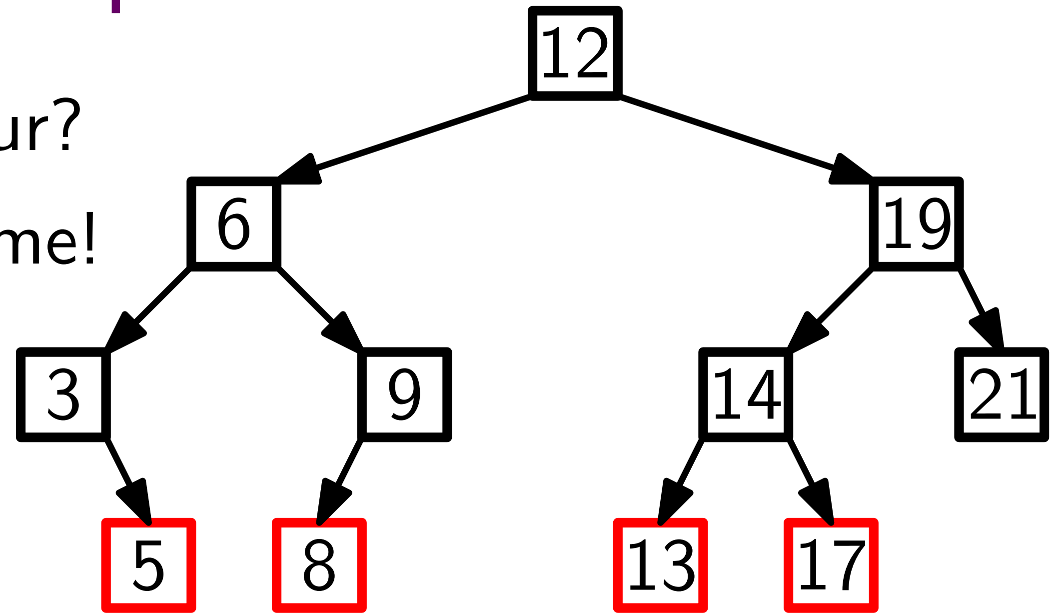
Select(int i):

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v):

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

Select(int i): $O(i \cdot h)$

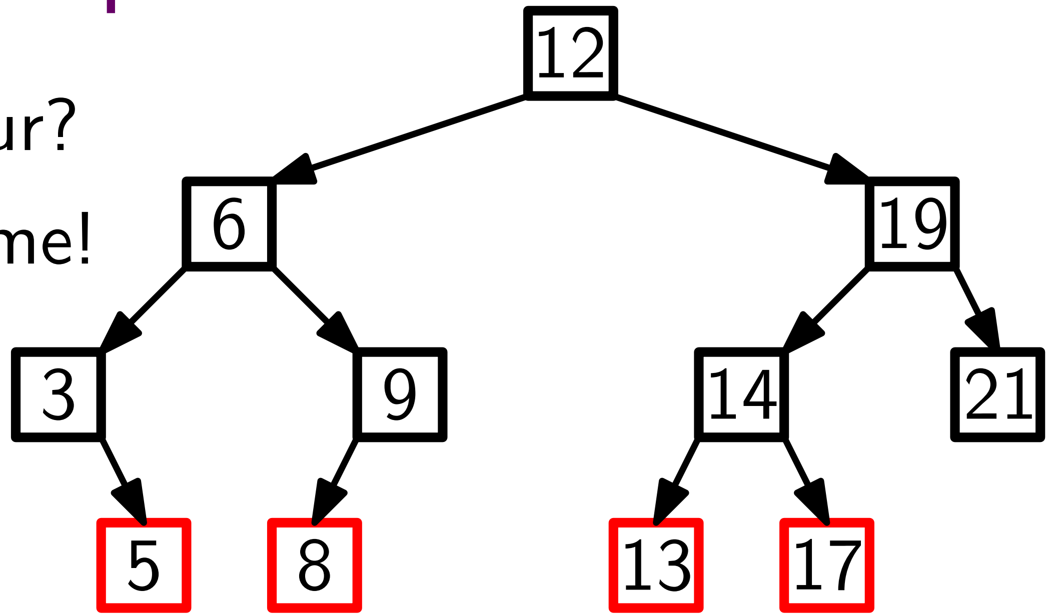
```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v):

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

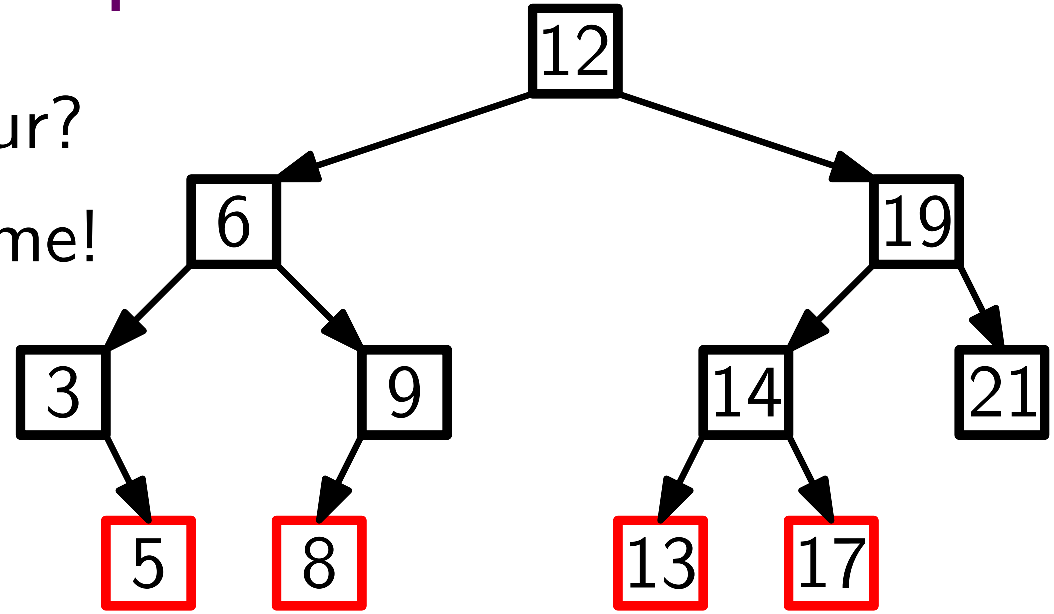
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich?

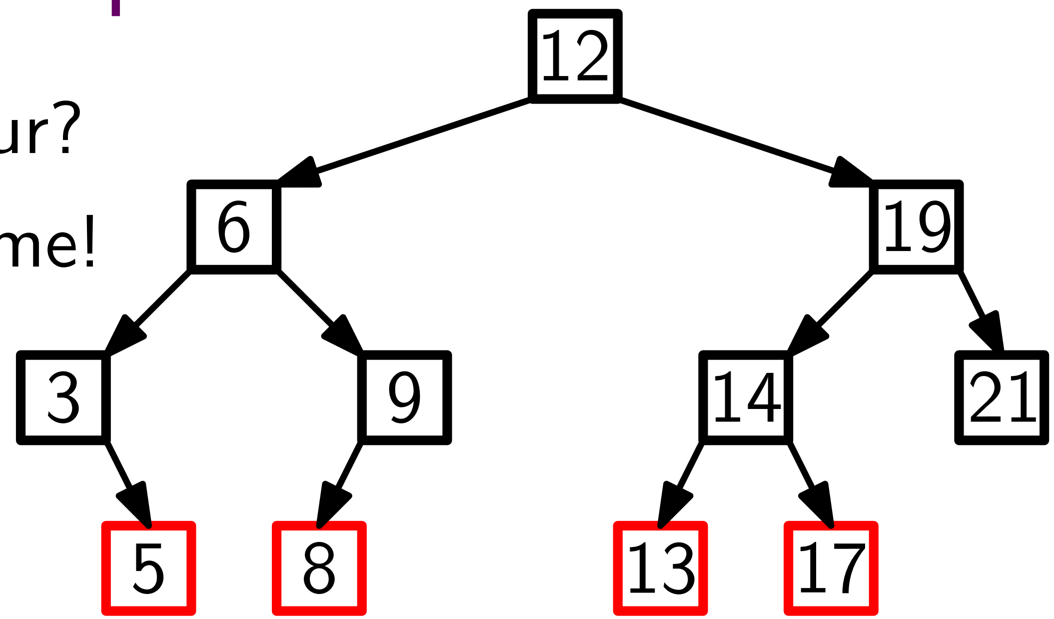
Select(int i): $O(i \cdot h)$

$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

Rank(Node v): $O(rank \cdot h)$

$j = 0$
while $v \neq nil$ **do**
  $v = \text{Predecessor}(v)$
  $j = j + 1$
return j

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

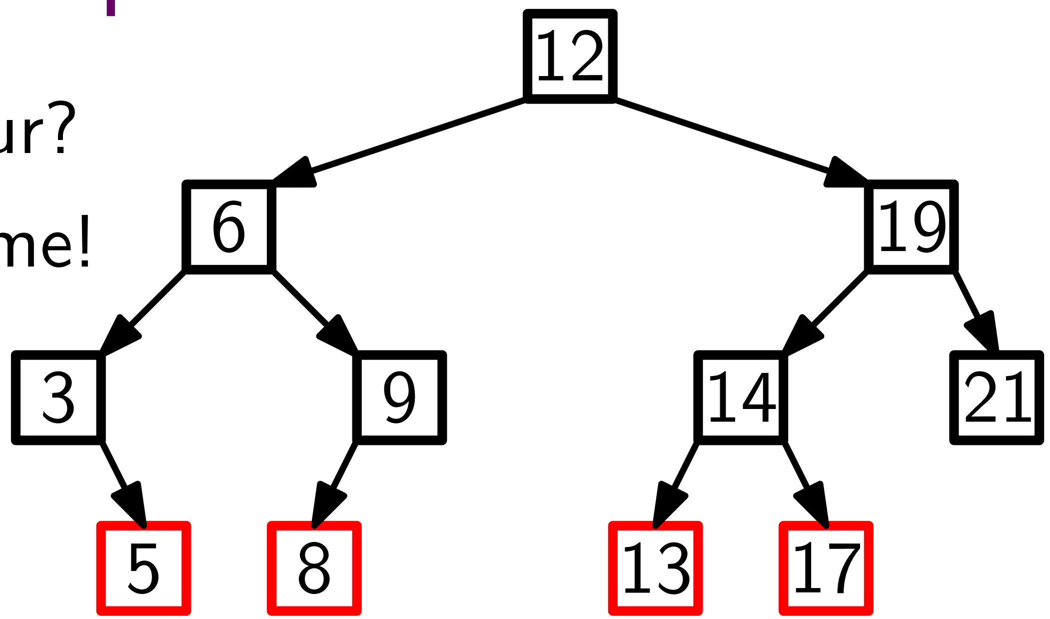
Select(int i): $O(i \cdot h)$

$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

Rank(Node v): $O(rank \cdot h)$

$j = 0$
while $v \neq nil$ **do**
  $v = \text{Predecessor}(v)$
  $j = j + 1$
return j

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

Select(int i): $O(i \cdot h)$

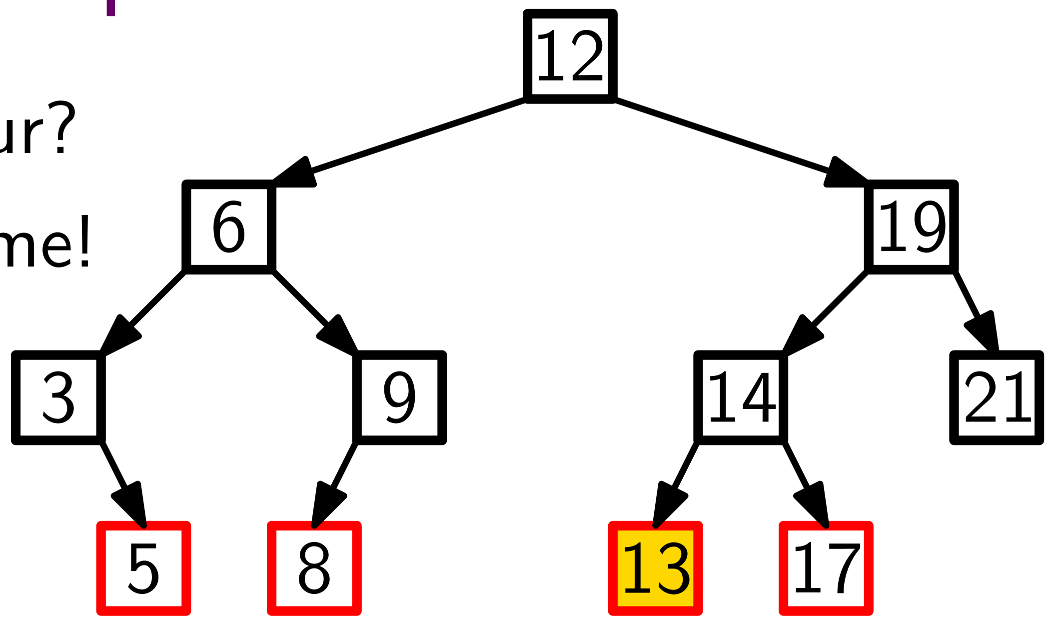
```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem

Select(7)



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

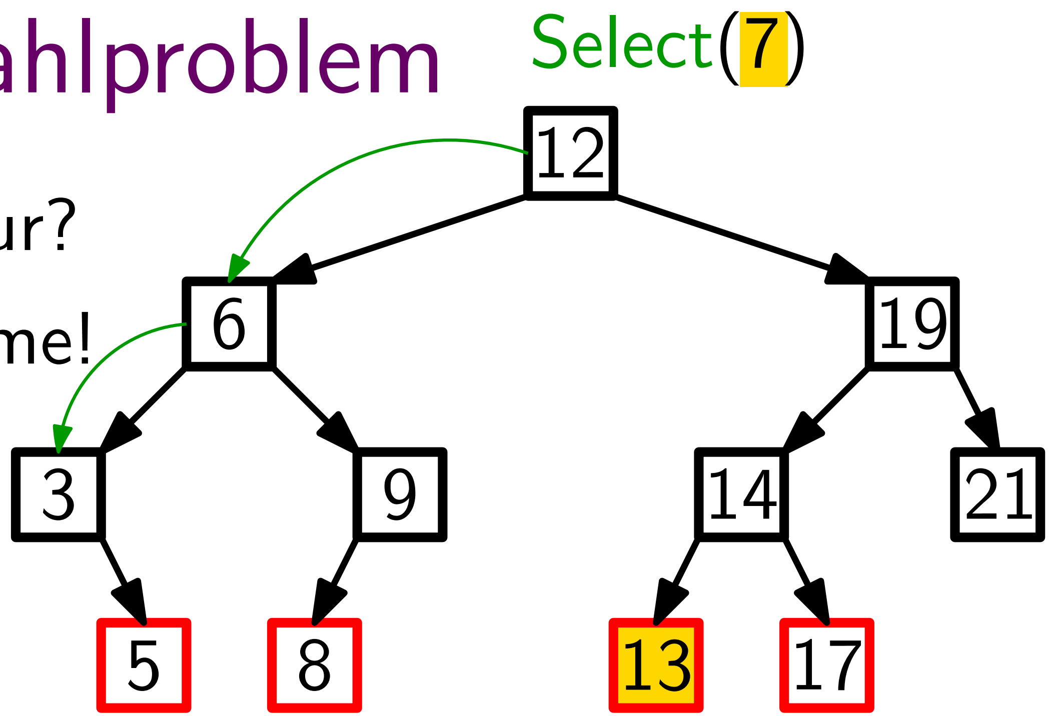
Select(int i): $O(i \cdot h)$

$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

Rank(Node v): $O(rank \cdot h)$

$j = 0$
while $v \neq nil$ **do**
  $v = \text{Predecessor}(v)$
  $j = j + 1$
return j

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

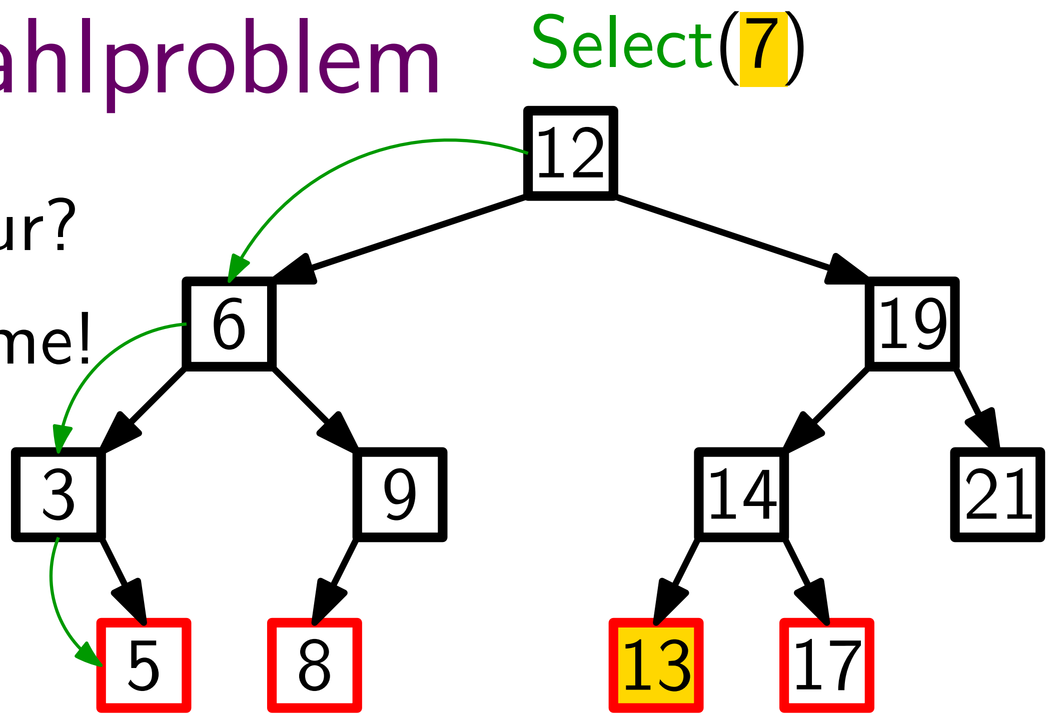
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

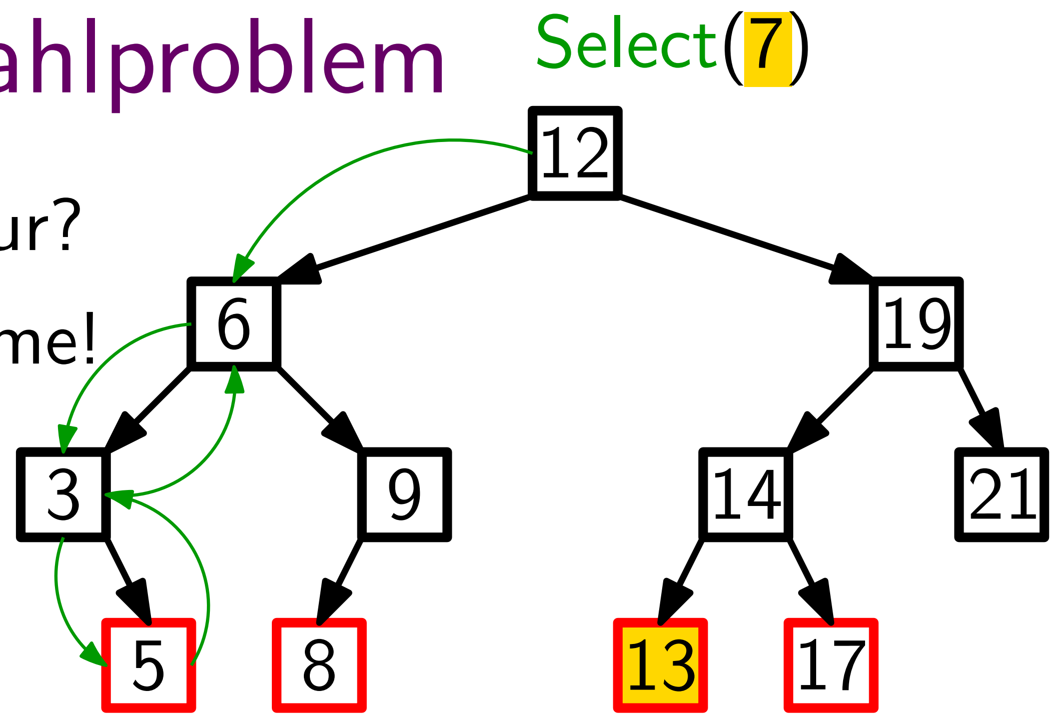
Select(int i): $O(i \cdot h)$

$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

Rank(Node v): $O(rank \cdot h)$

$j = 0$
while $v \neq nil$ **do**
  $v = \text{Predecessor}(v)$
  $j = j + 1$
return j

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

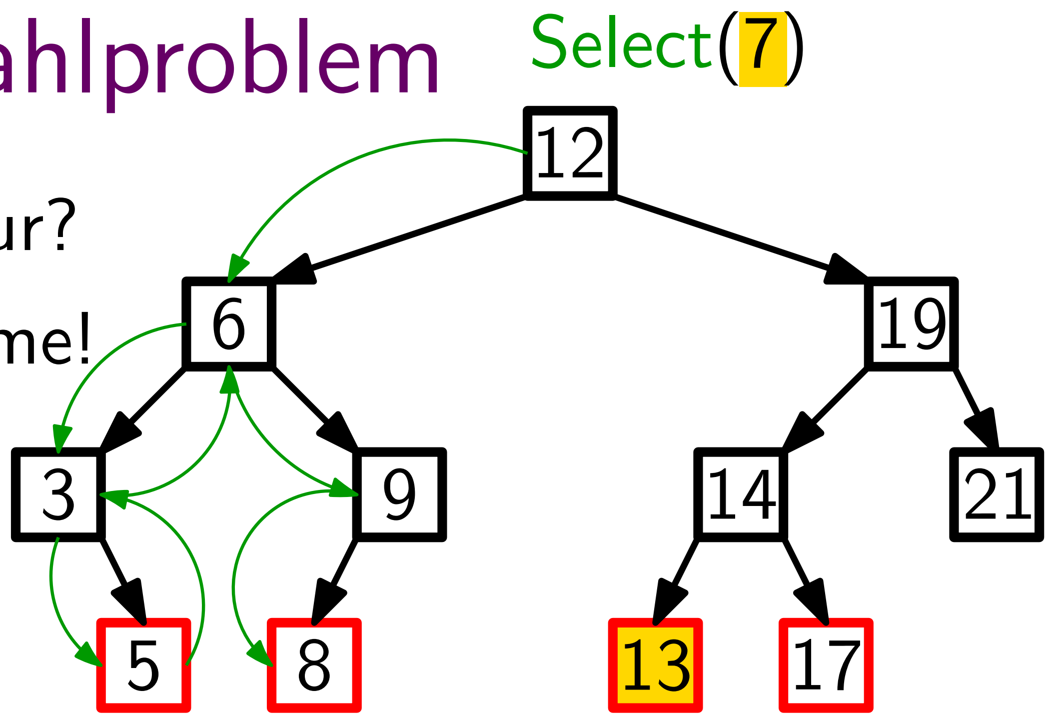
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

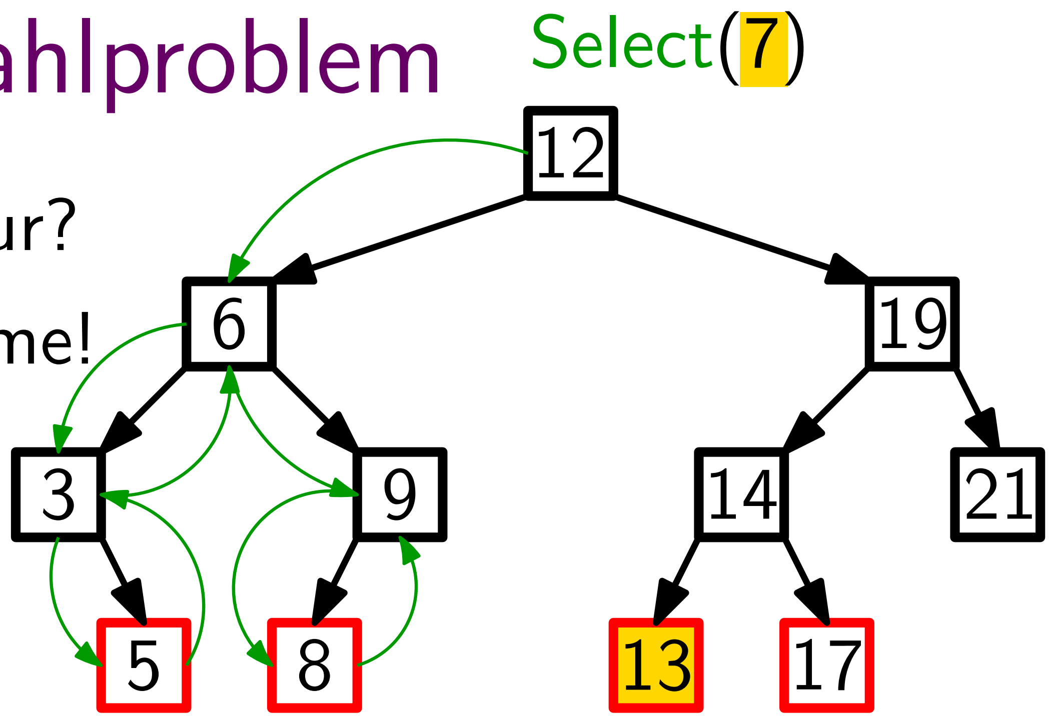
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

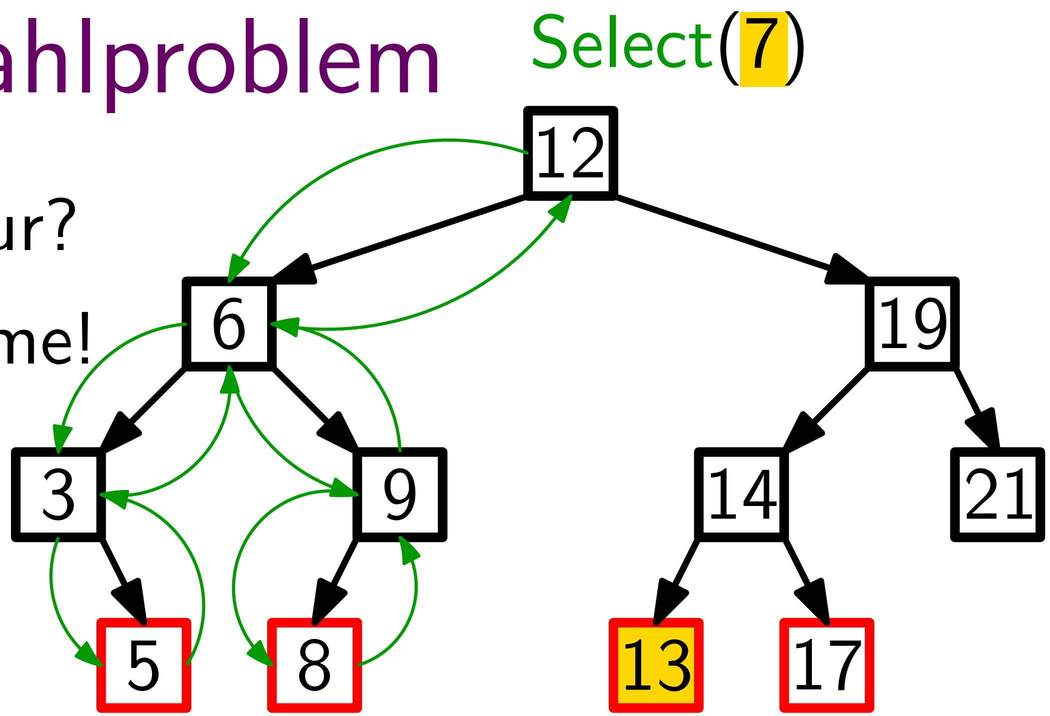
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

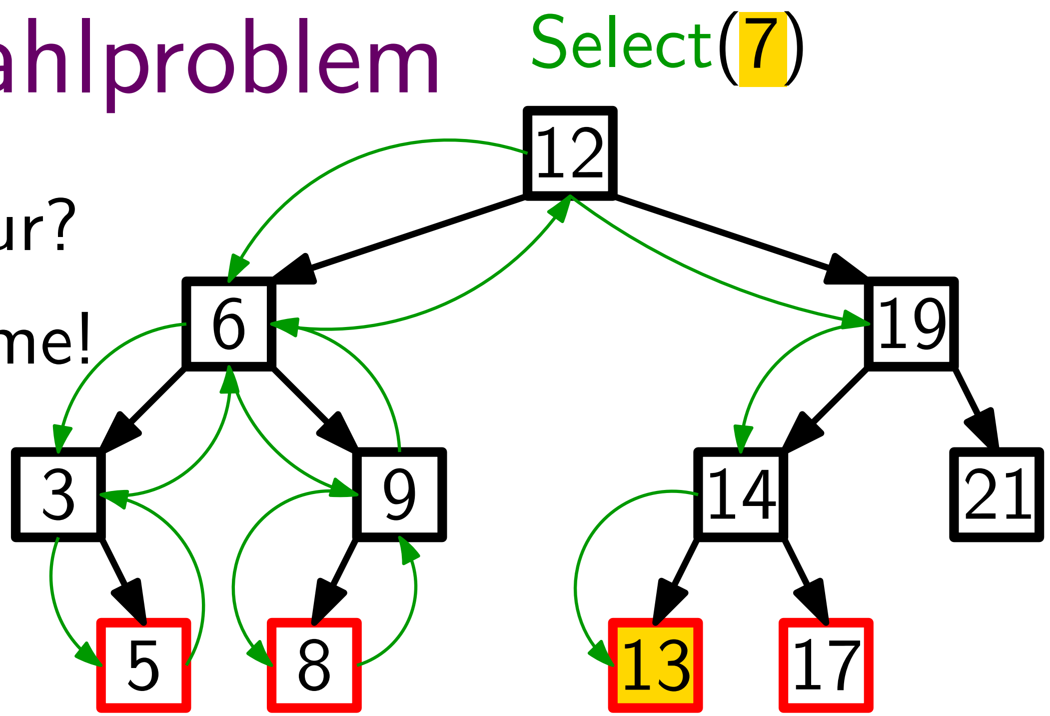
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

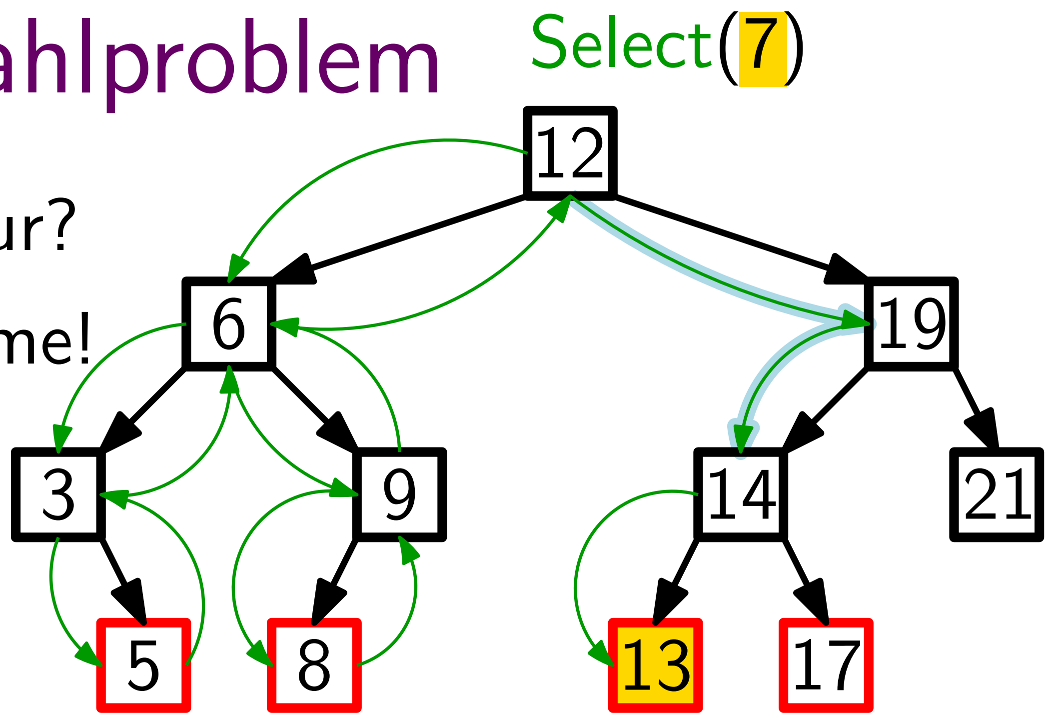
Select(int i): $O(i \cdot h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank \cdot h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

Select(int i): $O(i \cdot h)$

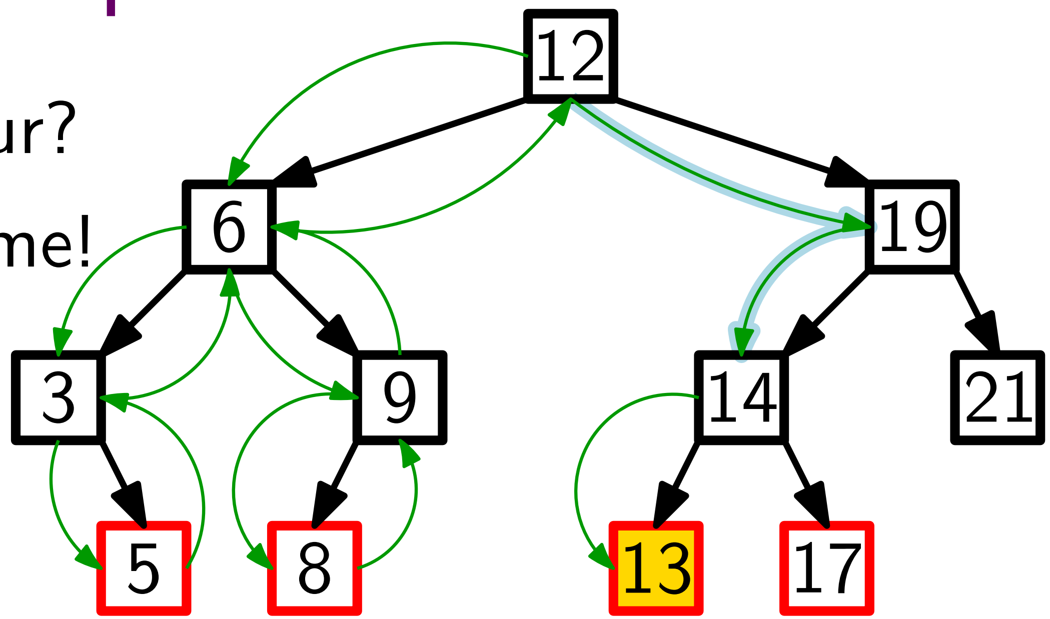
v = Minimum()
while $v \neq nil$ **and** $i > 1$ **do**
  v = Successor(v)
  $i = i - 1$
return v

Rank(Node v): $O(rank \cdot h)$

$j = 0$
while $v \neq nil$ **do**
  v = Predecessor(v)
  $j = j + 1$
return j

Das dynamische Auswahlproblem

Select(7)



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

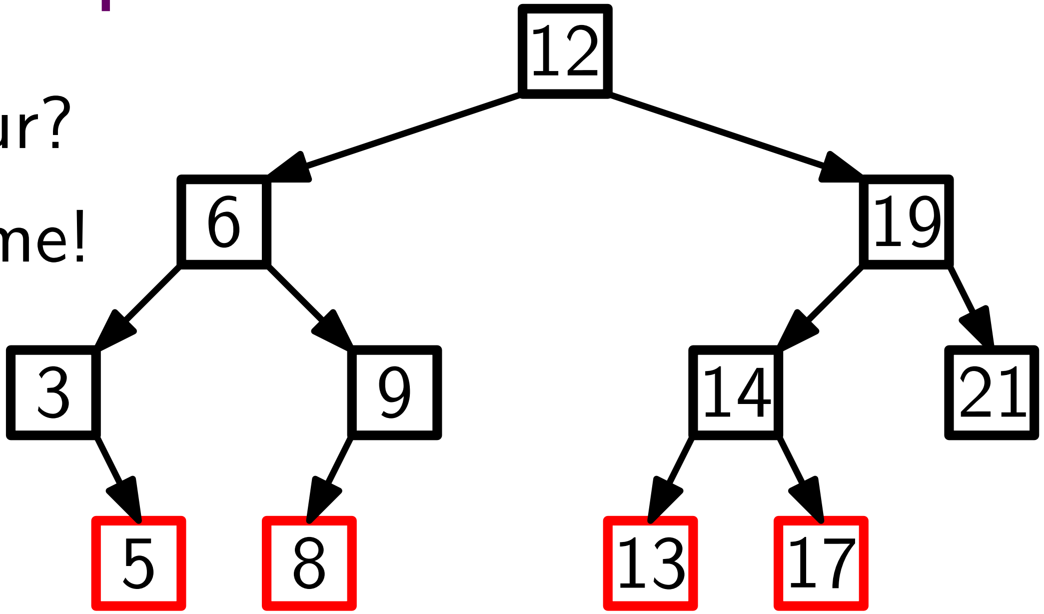
Select(int i): $O(\boxed{i} + h)$

```
v = Minimum()
while v ≠ nil and i > 1 do
    v = Successor(v)
    i = i − 1
return v
```

Rank(Node v): $O(rank + h)$

```
j = 0
while v ≠ nil do
    v = Predecessor(v)
    j = j + 1
return j
```

Das dynamische Auswahlproblem



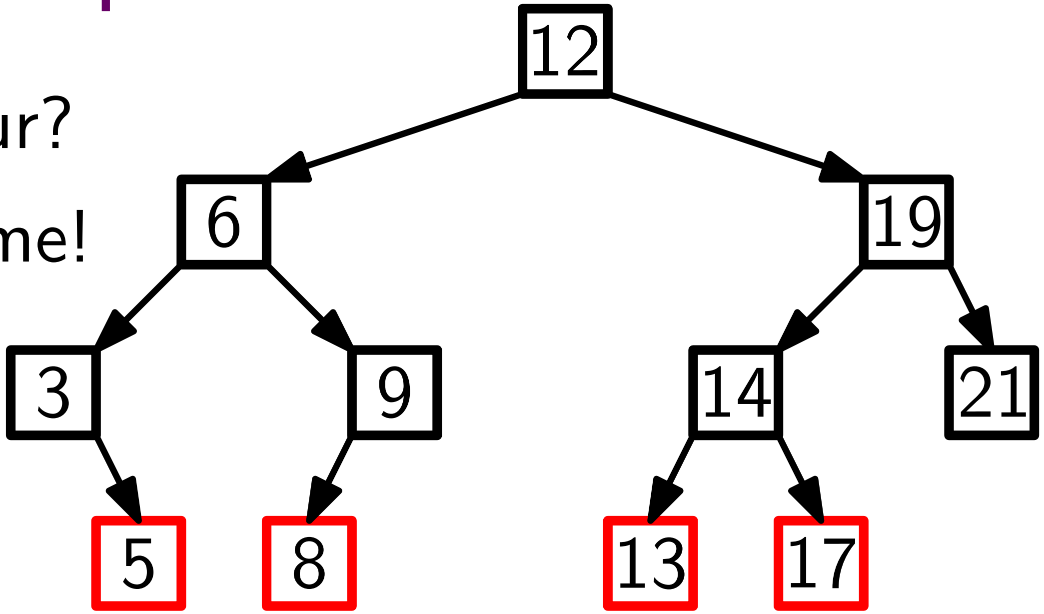
1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?
   Select(int i): $O(i + h)$ Rank(Node v): $O(\mathit{rank} + h)$

Problem: Wenn $i \in \Theta(n)$ – z.B. beim Median –,
dann ist die Laufzeit linear (wie im statischen Fall!).

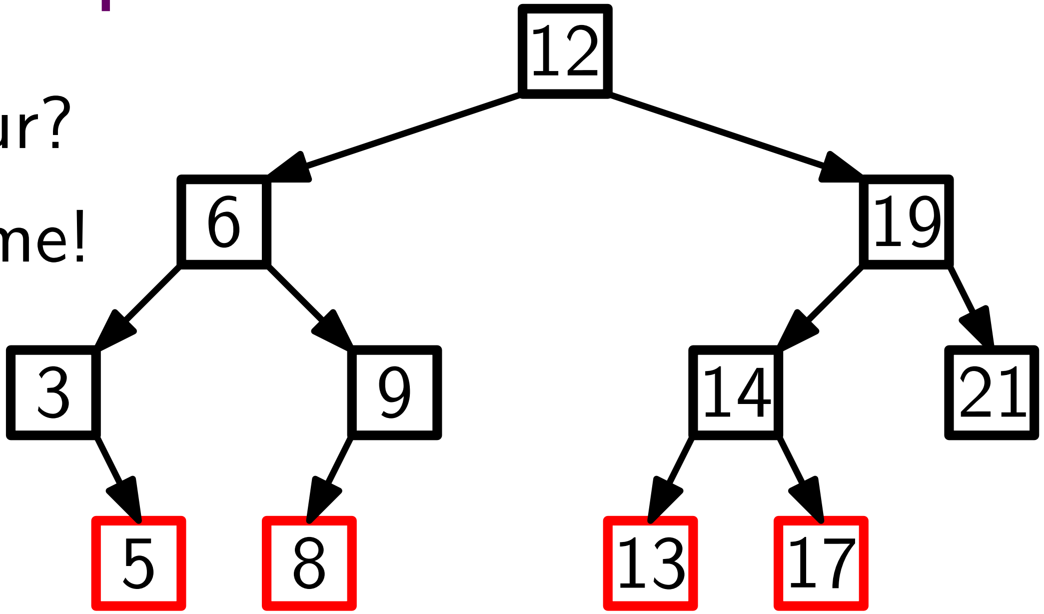
Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
     z.B. Rot-Schwarz-Bäume
     $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?

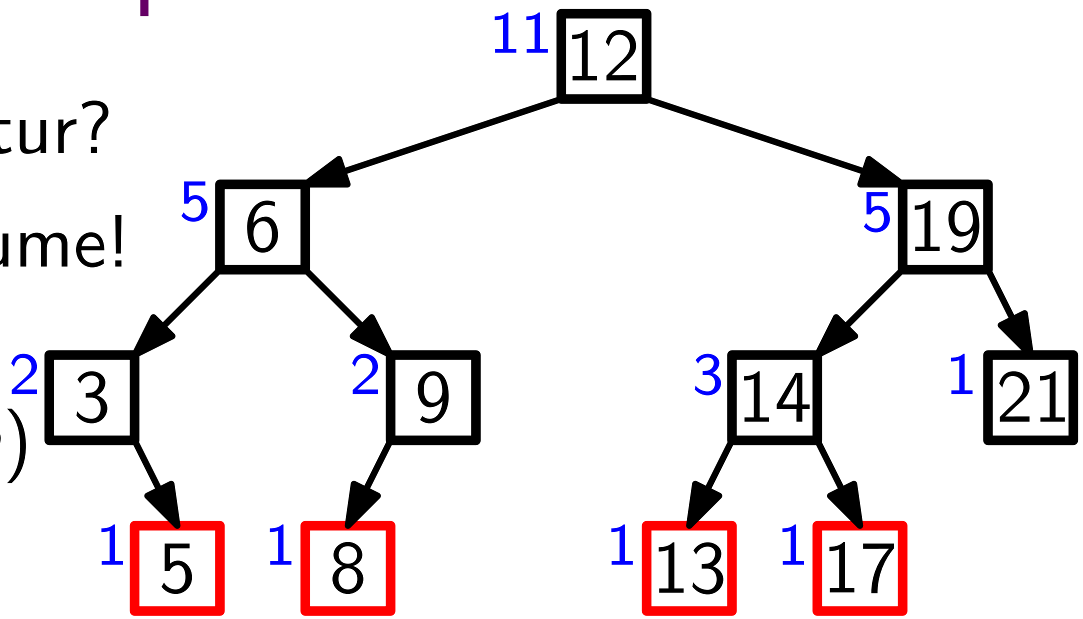
Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume

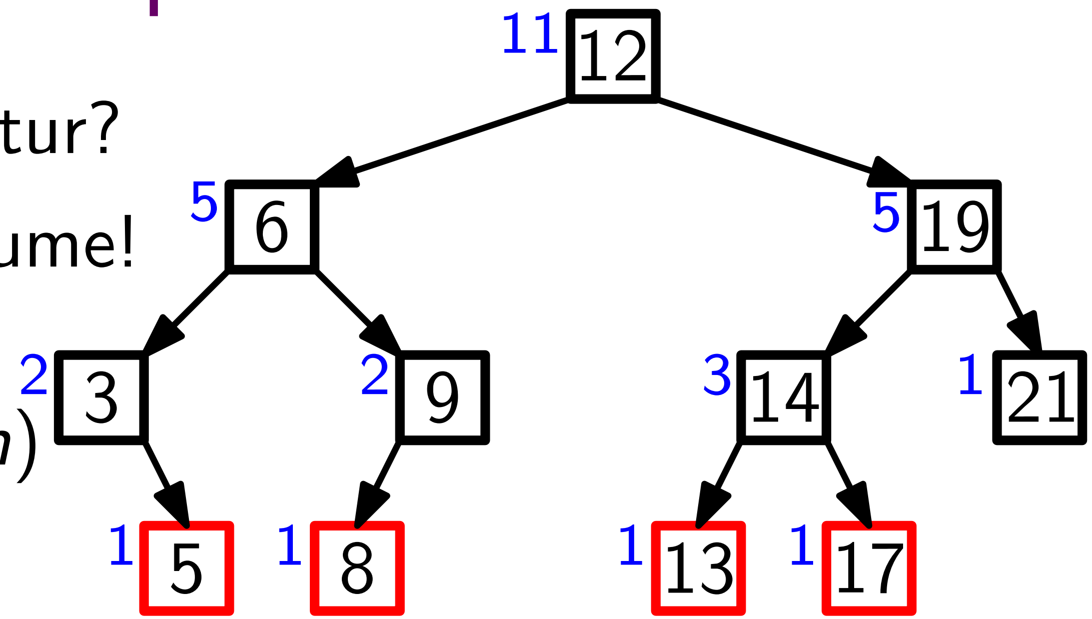
Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

Das dynamische Auswahlproblem



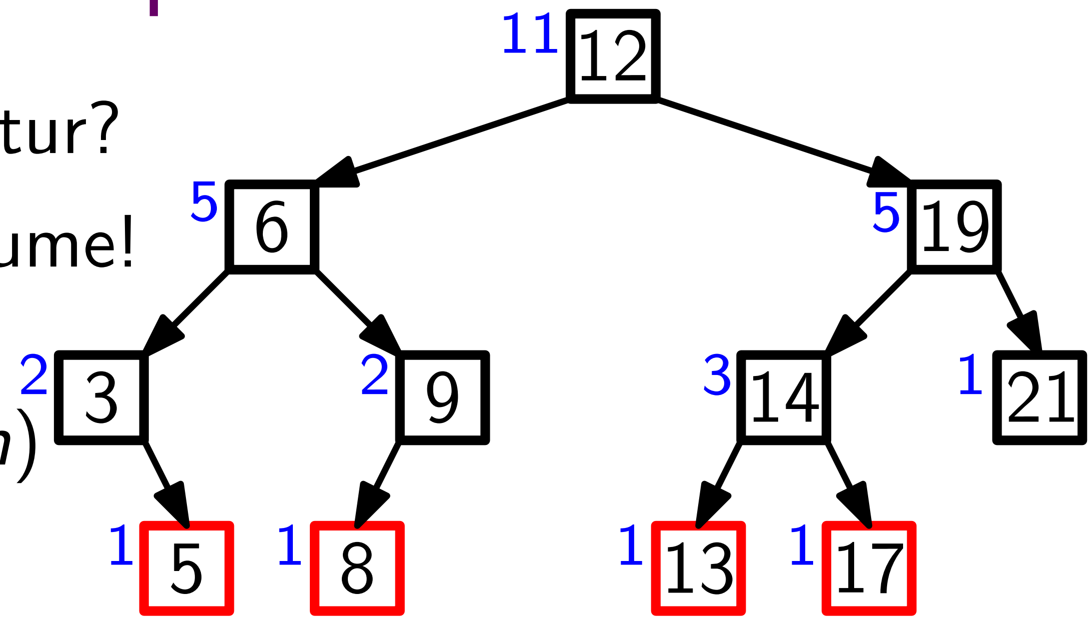
1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

   Rank(Node v):

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

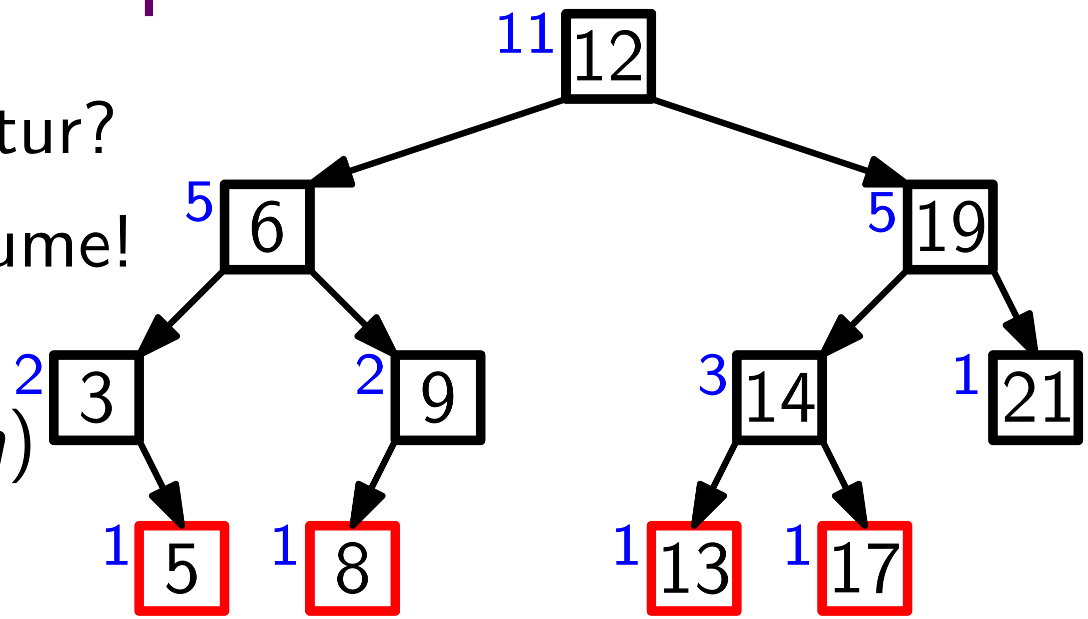
4. Select(Node $v = root$, int i):

$r = v.left.size + 1$
if $i == r$ **then return**
else
  if $i < r$ **then**
    return
  else
    return

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

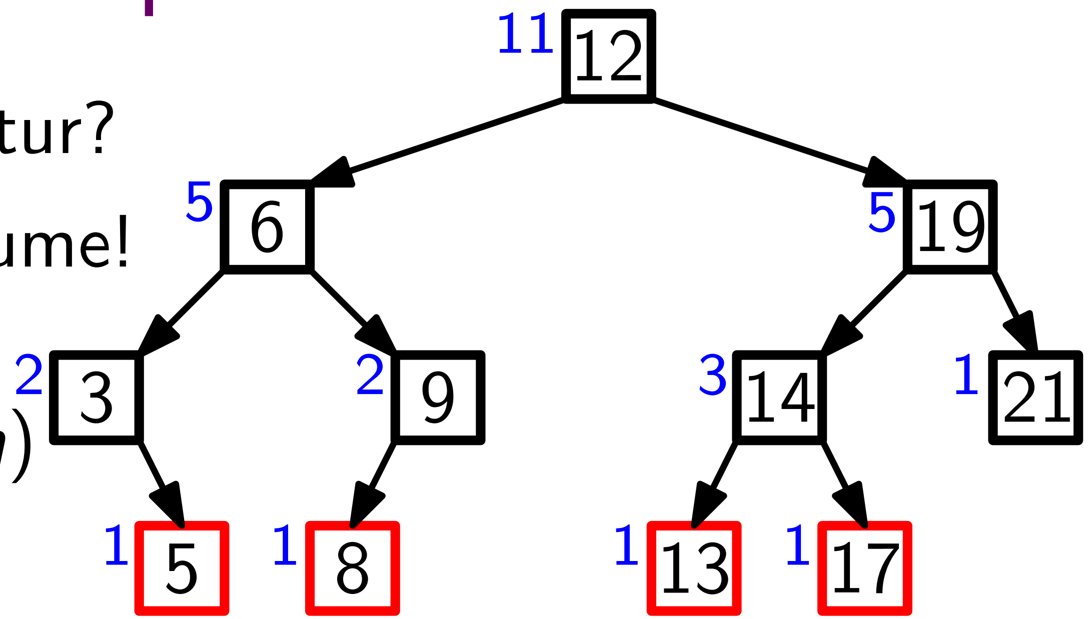
4. Select(Node $v = root$, int i):

$r = v.left.size + 1$
if $i == r$ **then return** v
else
    if $i < r$ **then**
        return
    else
        return

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

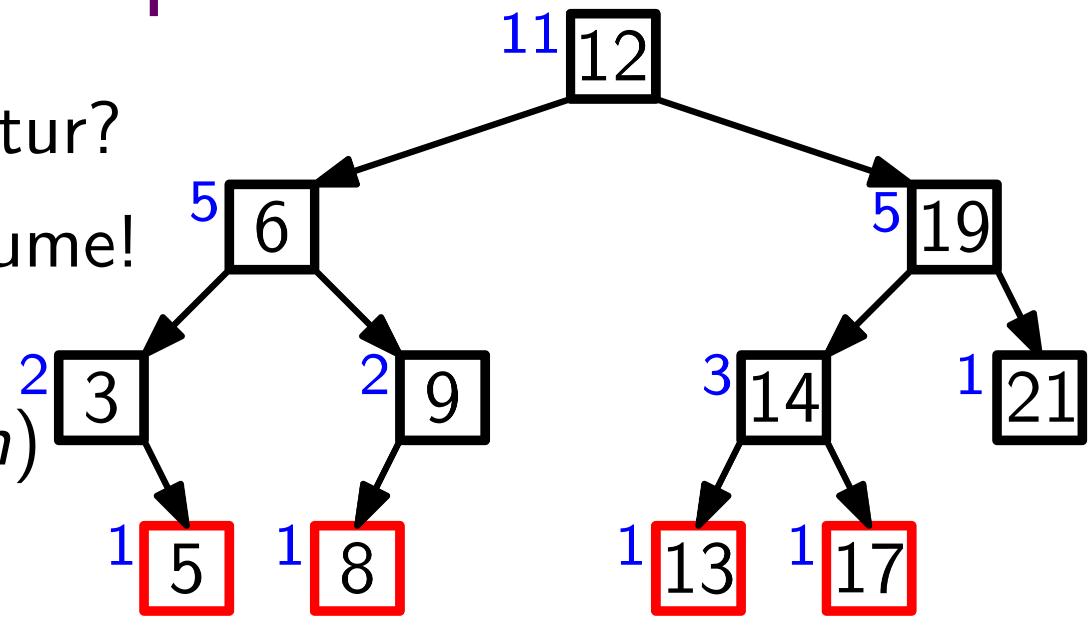
$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left$, i)
  else
    return

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

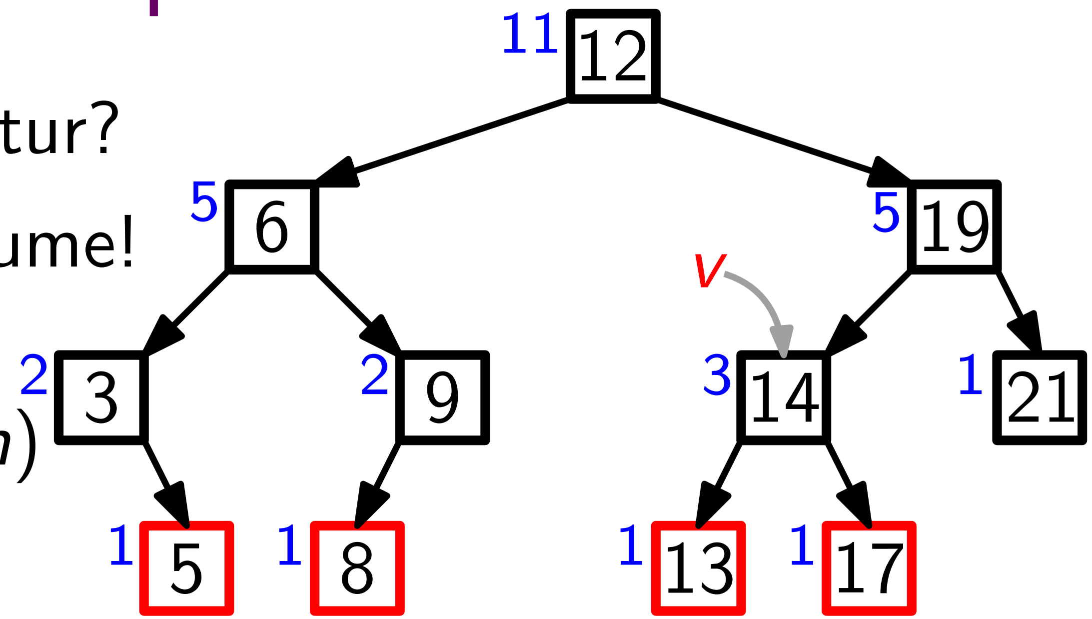
4. Select(Node $v = root$, int i):

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left, i$)
  else
    return Select($v.right, i - r$)

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

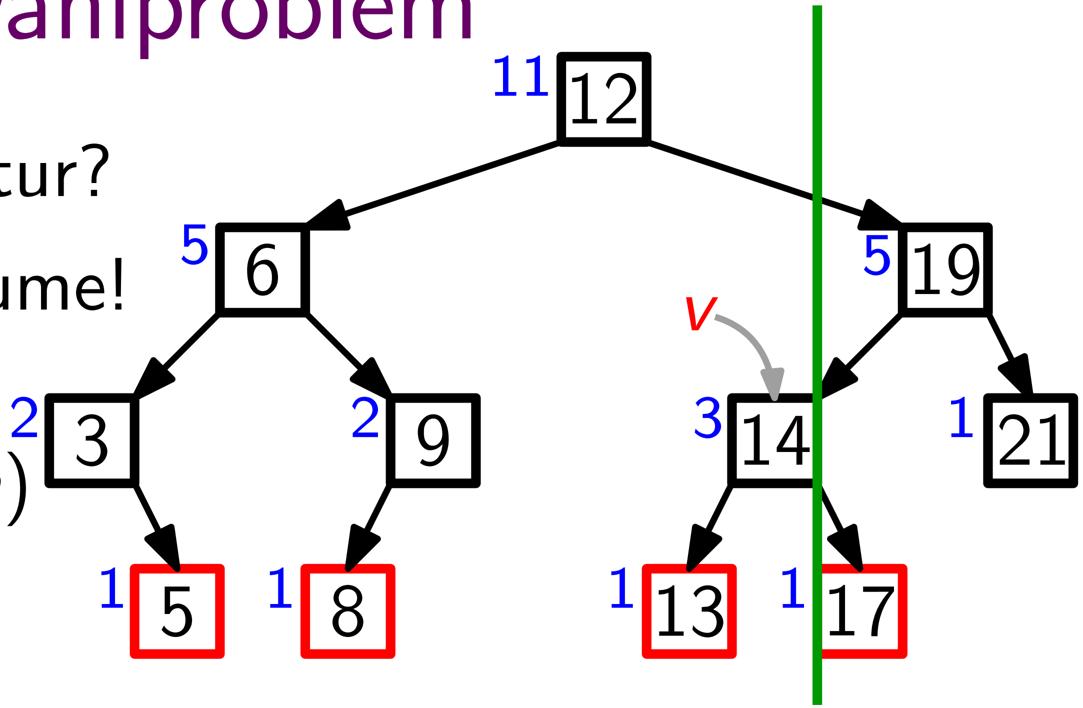
4. Select(Node $v = root$, int i):

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left$, i)
  else
    return Select($v.right$, $i - r$)

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

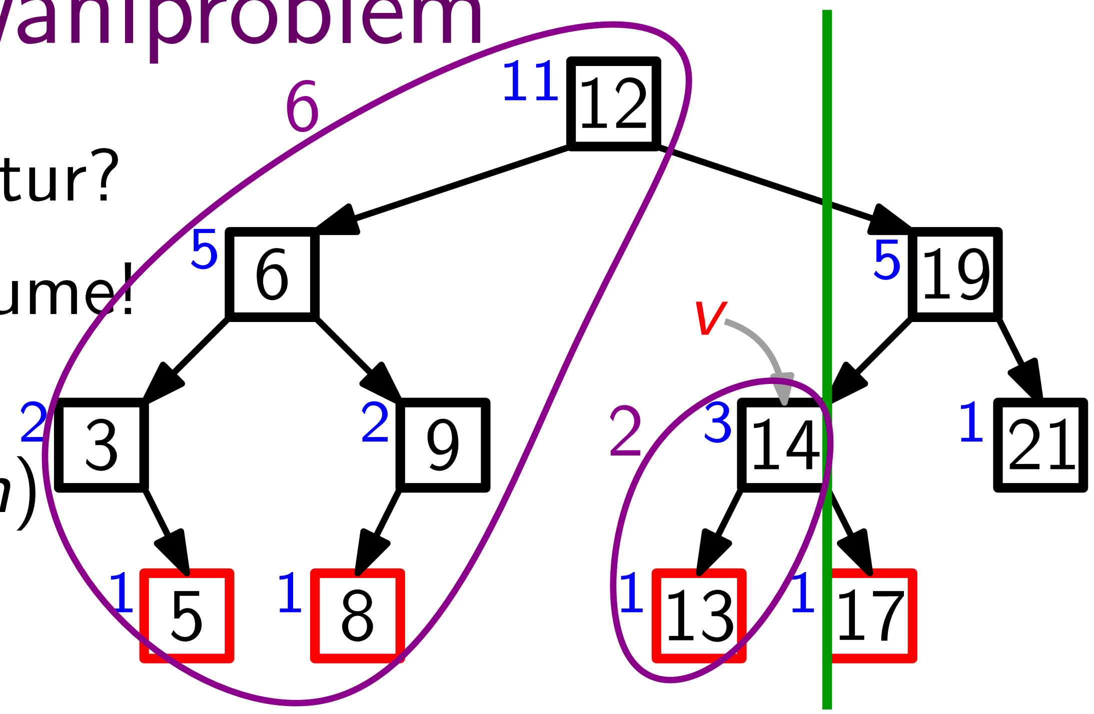
```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

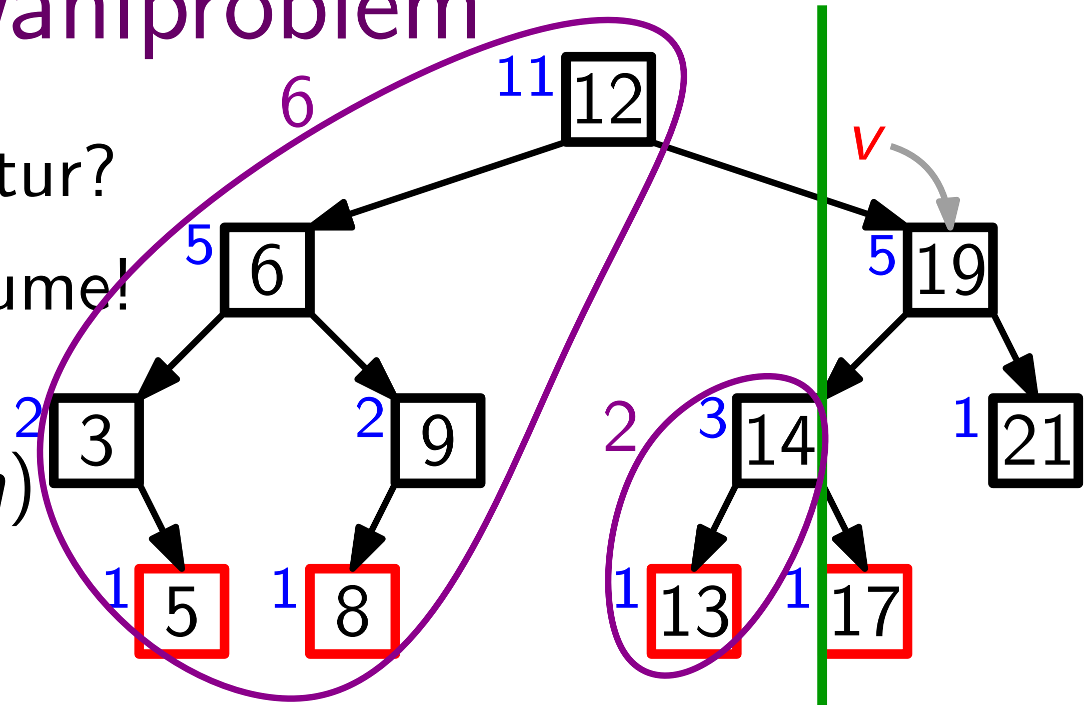
$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left$, i)
  else
    return Select($v.right$, $i - r$)

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

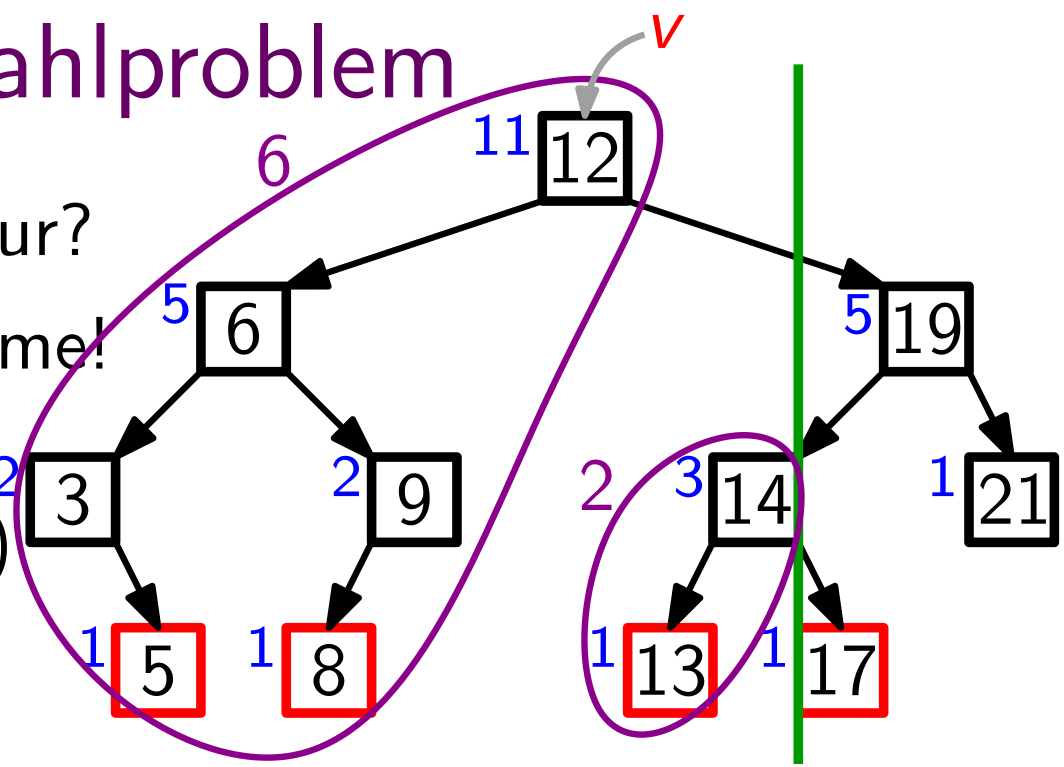
4. Select(Node $v = root$, int i):

```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
     z.B. Rot-Schwarz-Bäume
     $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

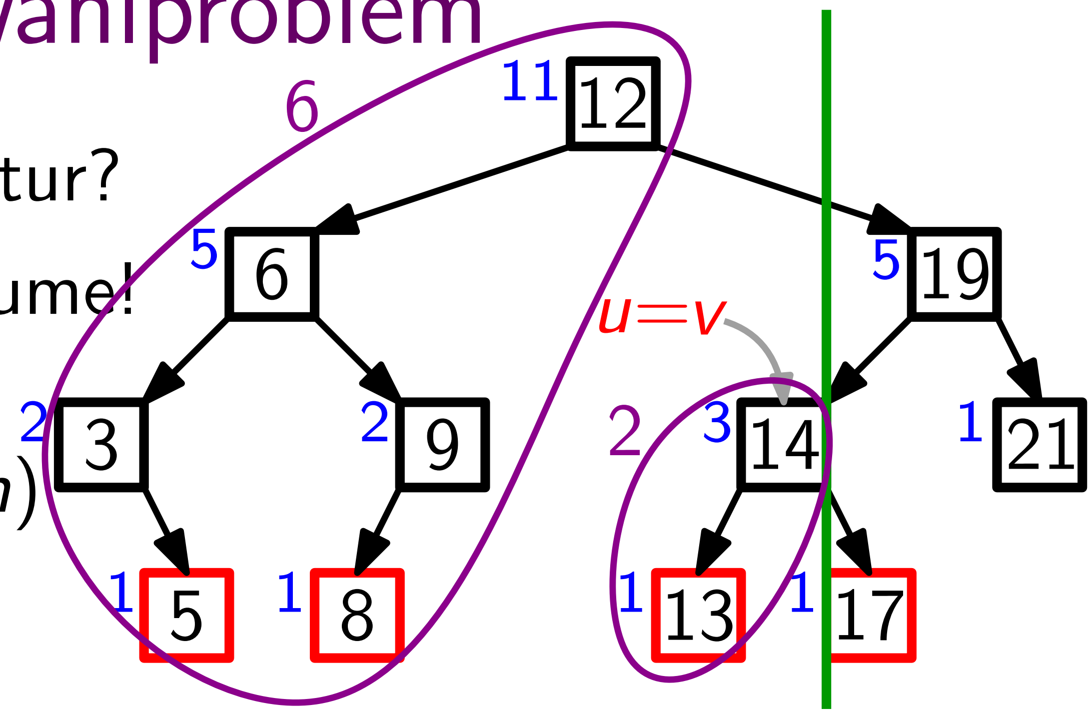
4. Select(Node $v = root$, int i):

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left, i$)
  else
    return Select($v.right, i - r$)

Rank(Node v):

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

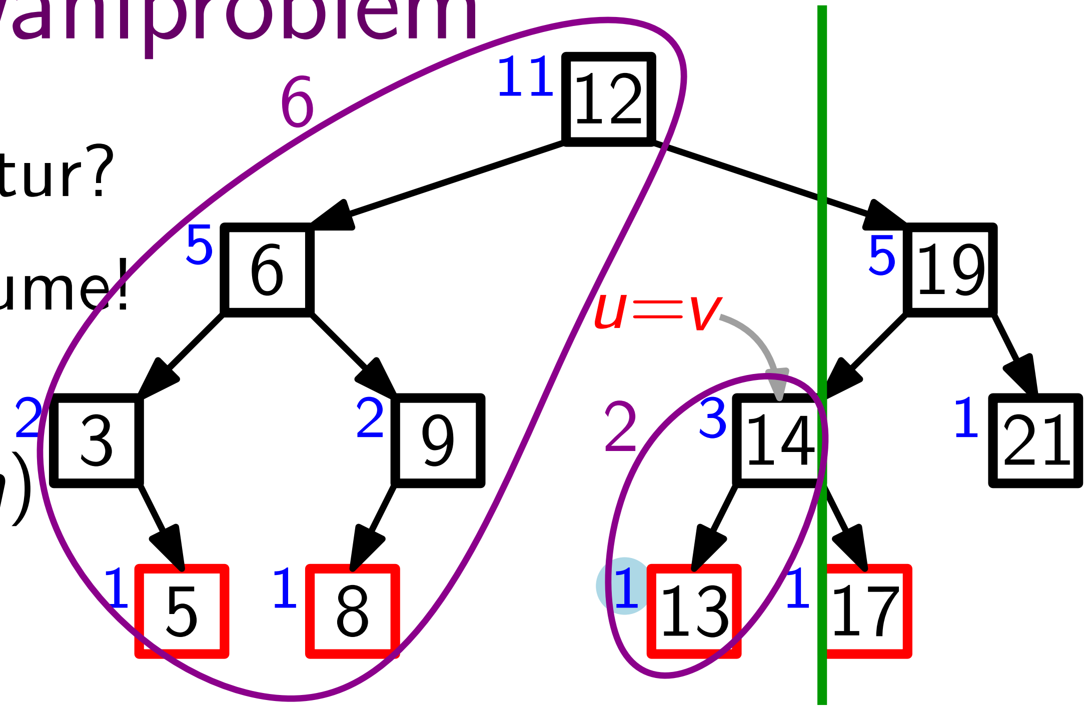
```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

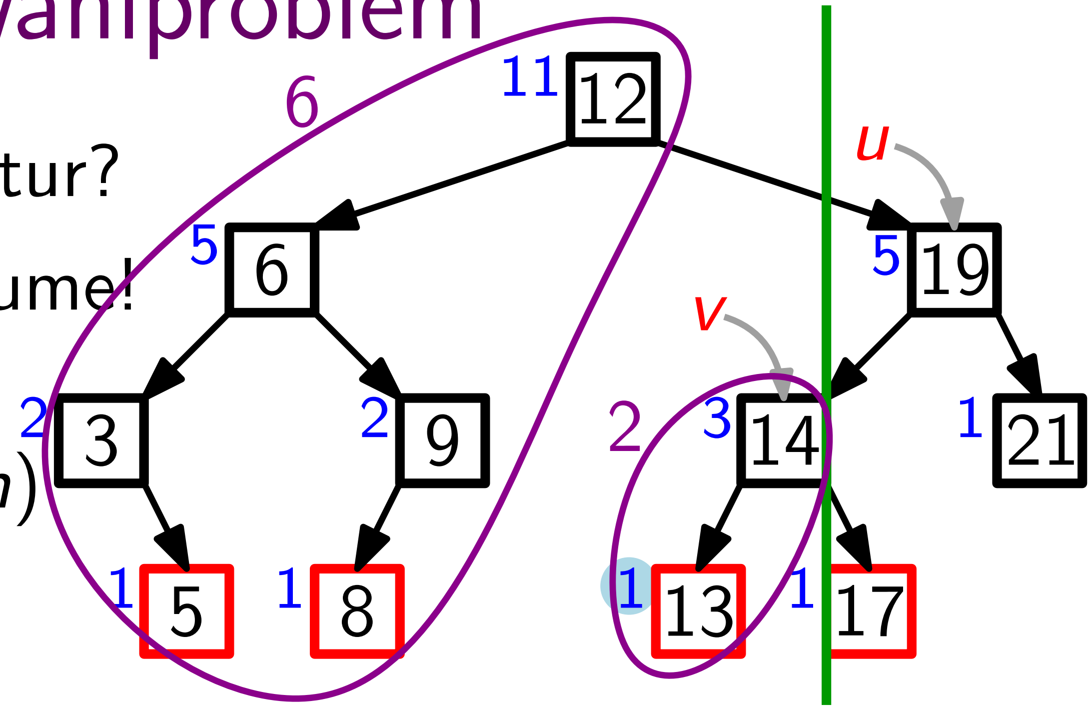
$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left, i$)
  else
    return Select($v.right, i - r$)

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  $u = u.p$
return r

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

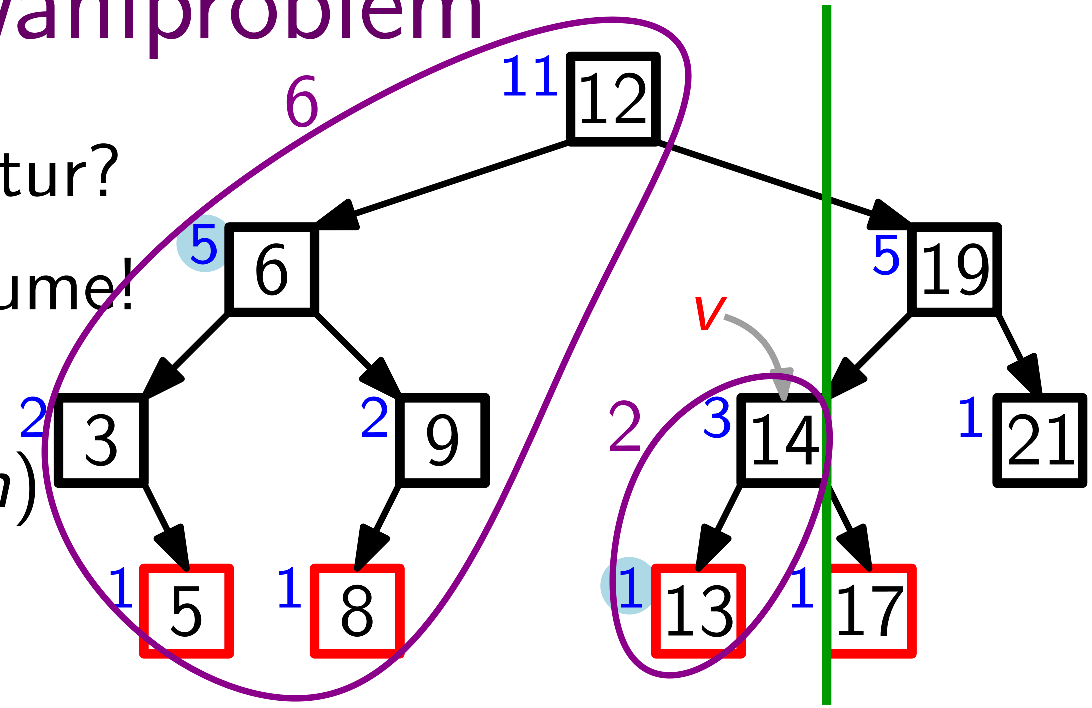
```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

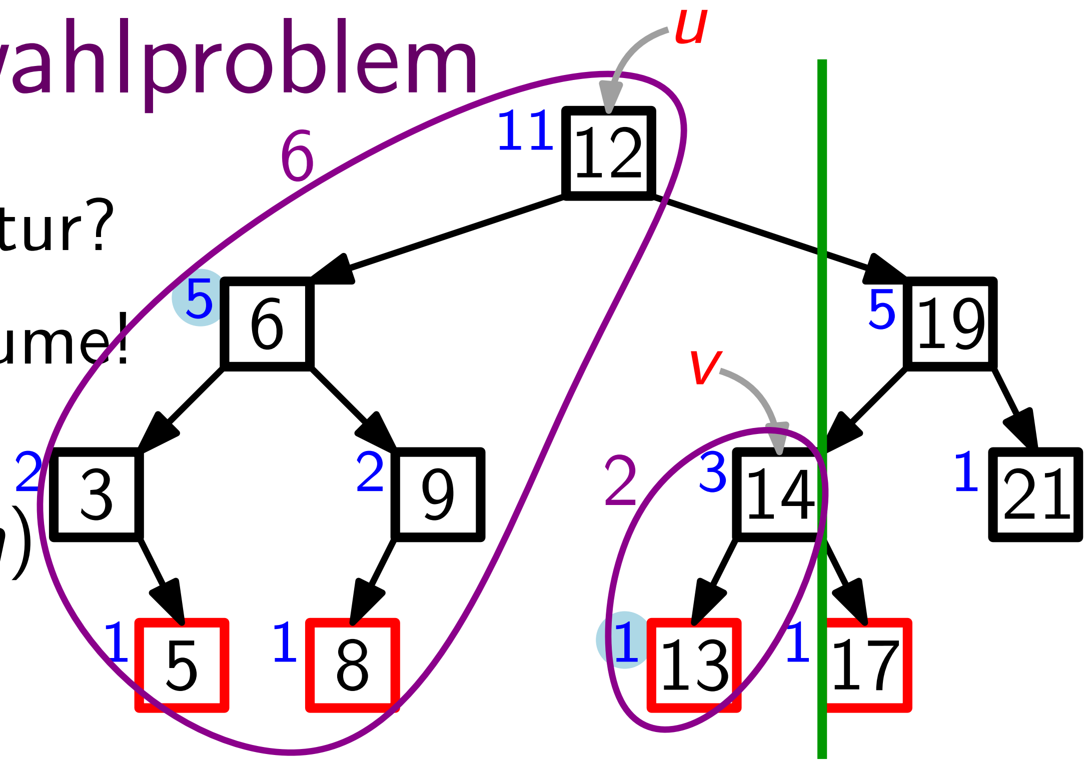
4. Select(Node $v = root$, int i):

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left, i$)
  else
    return Select($v.right, i - r$)

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

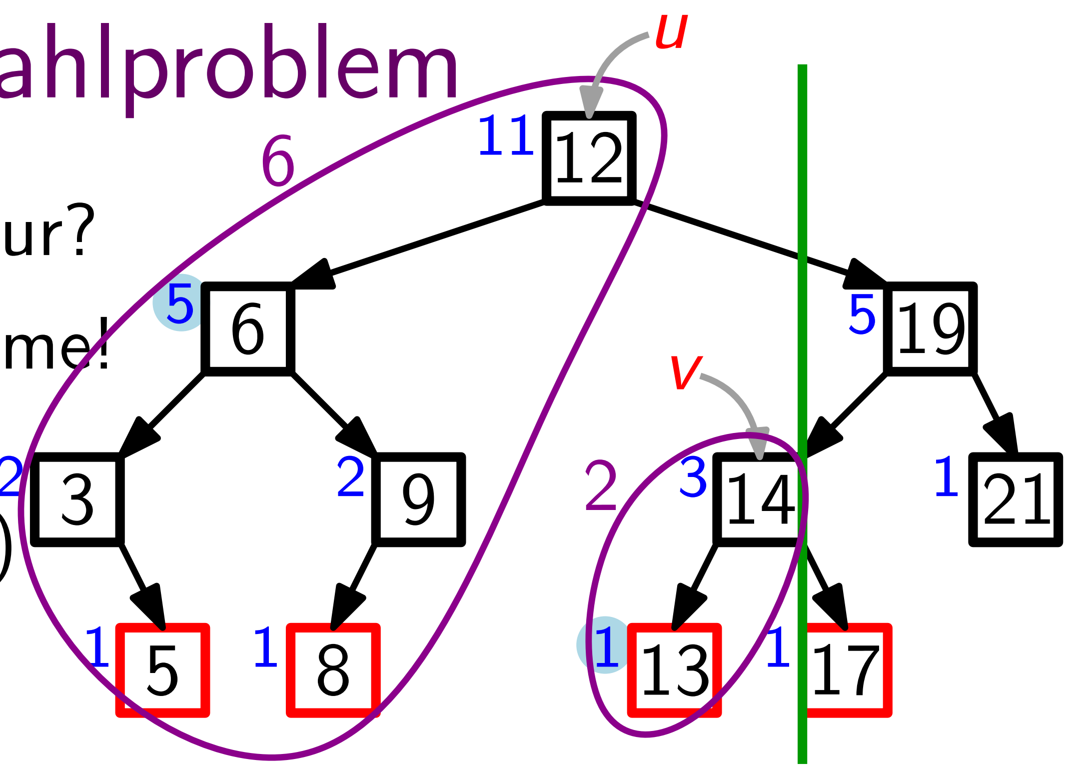
Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do

    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

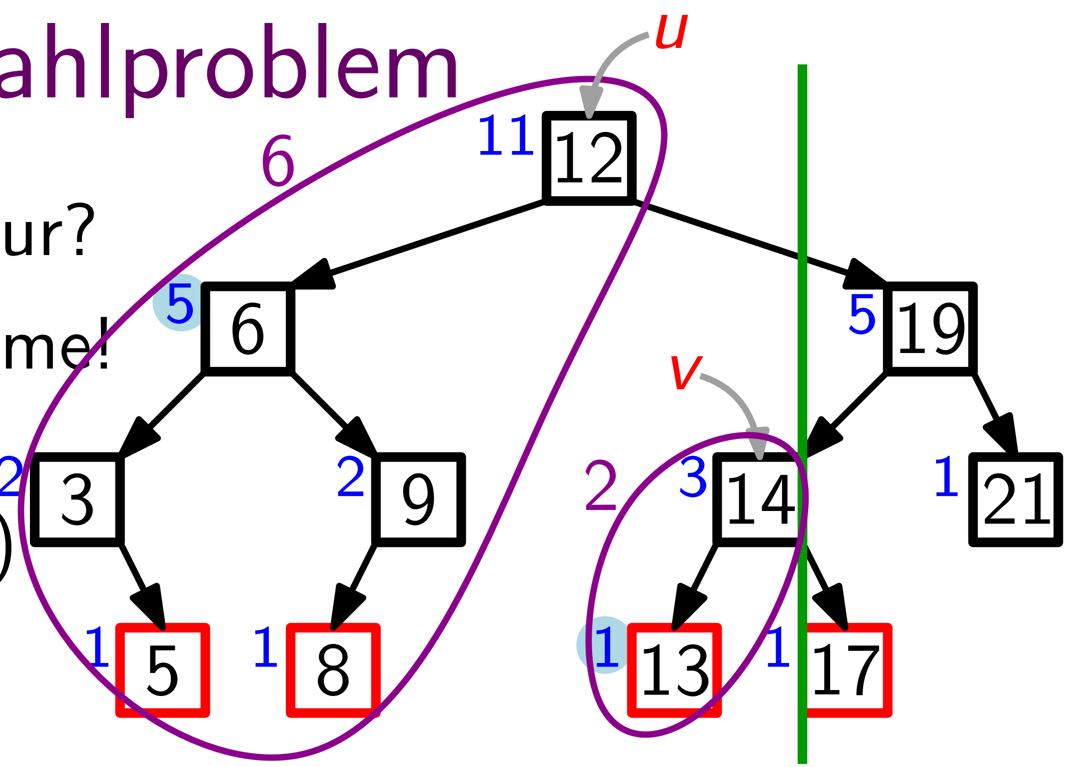
$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left, i$)
  else
    return Select($v.right, i - r$)

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
  $u = u.p$
return r

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i):

```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

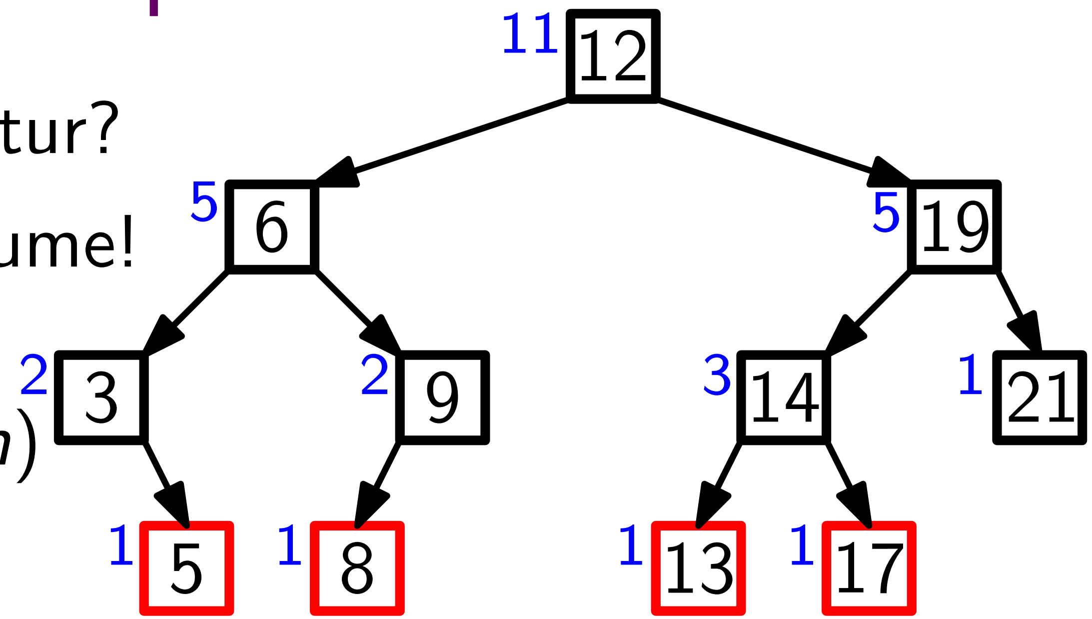
Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i): $O(\ \ )$

```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

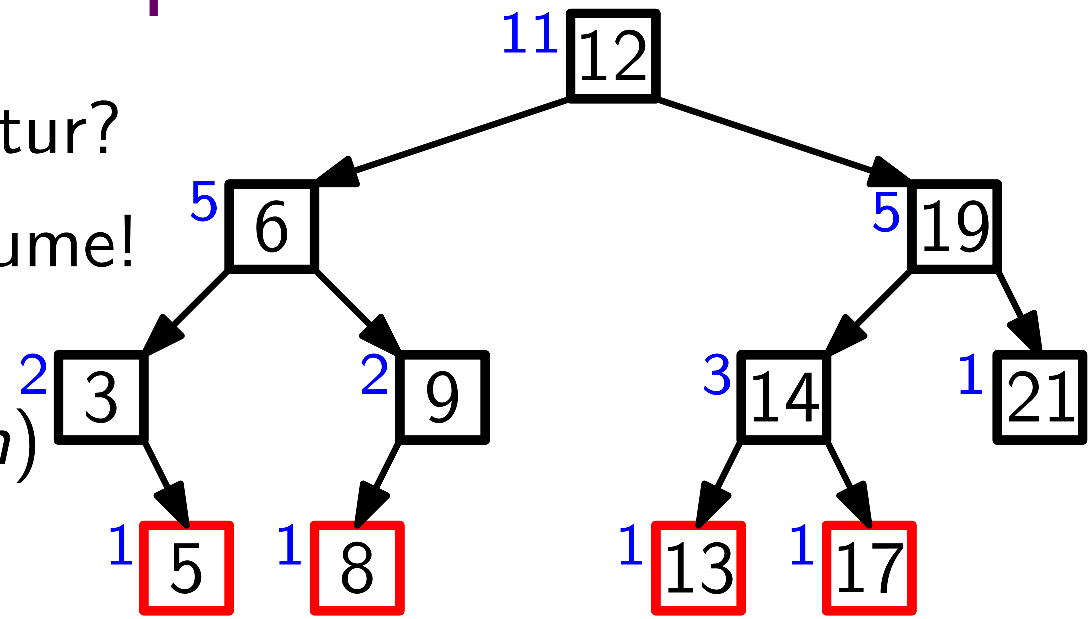
Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i): $O(\ \ )$

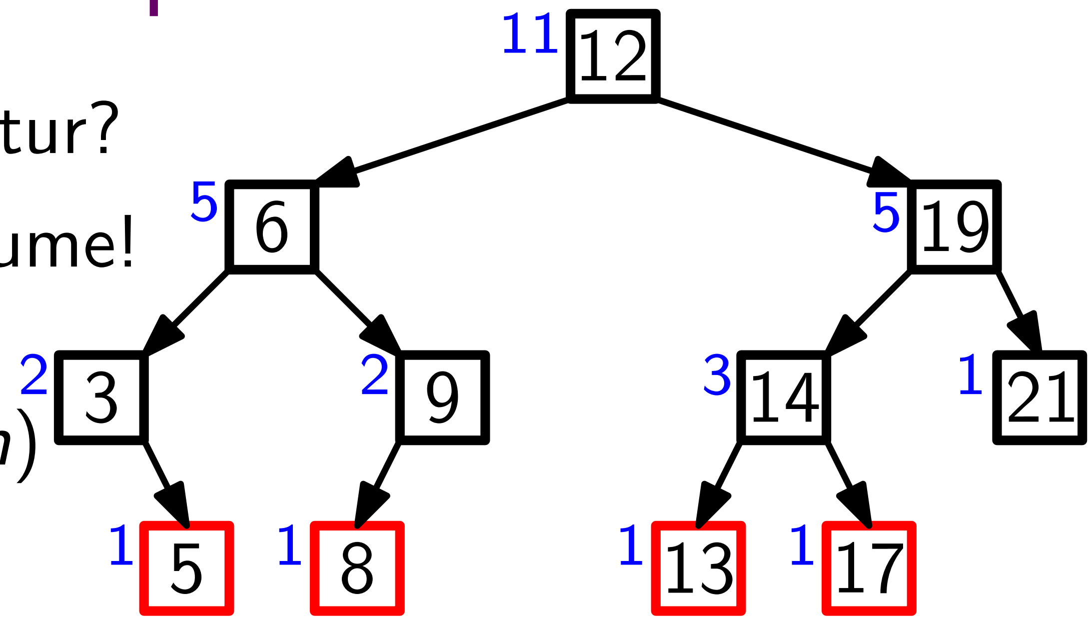
```
r = v.left.size + 1
if i == r then return v
else
    if i < r then
        return Select(v.left, i)
    else
        return Select(v.right, i − r)
```

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

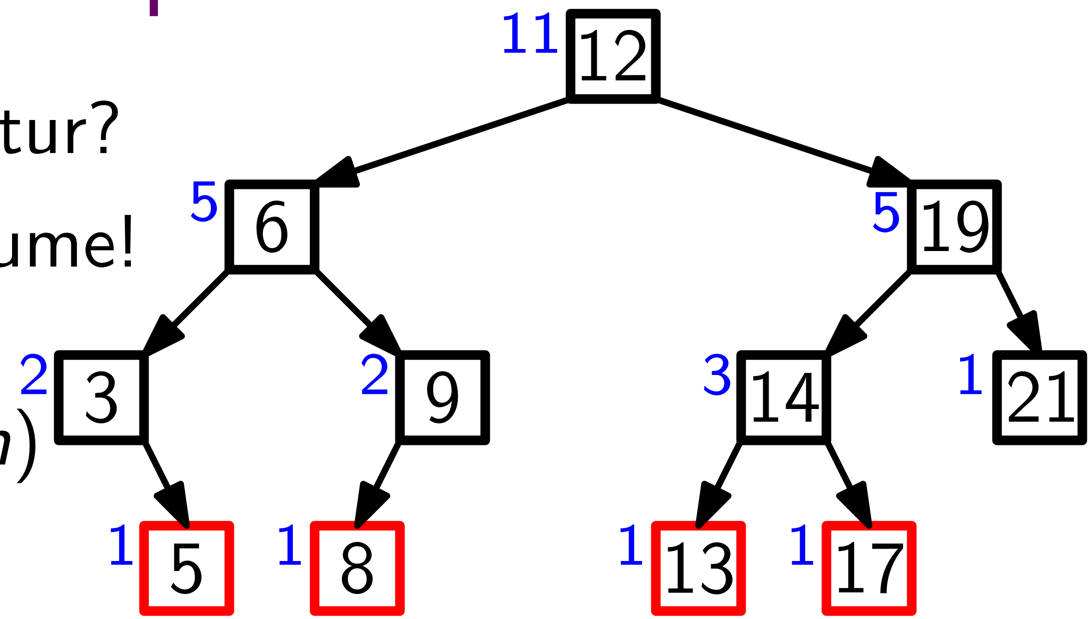
4. Select(Node $v = root$, int i): $O(h)$

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left$, i)
  else
    return Select($v.right$, $i - r$)

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

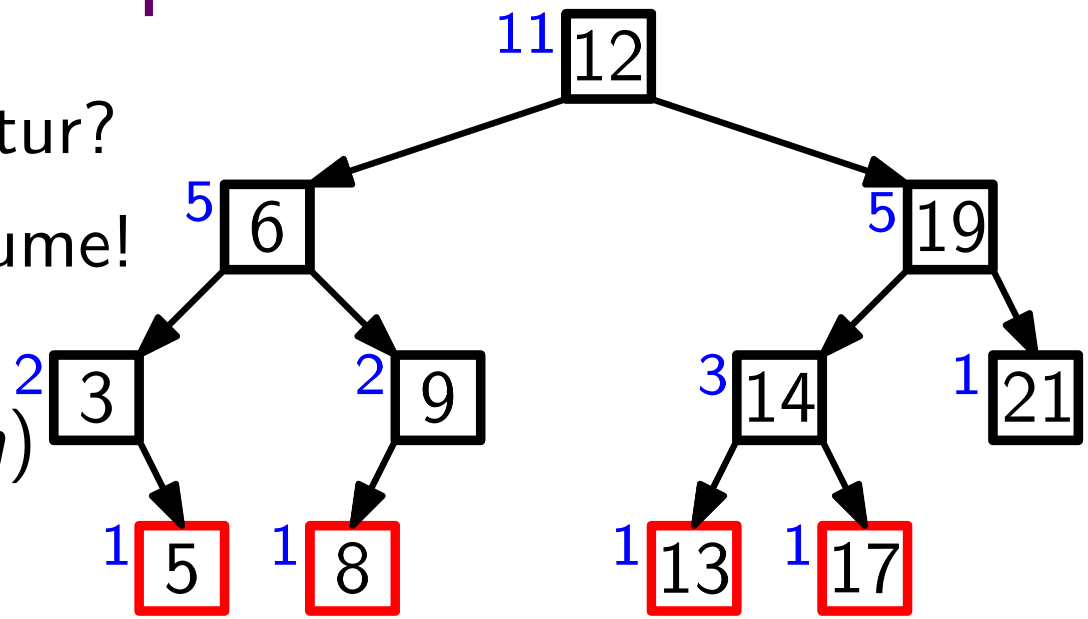
4. Select(Node $v = root$, int i): $O(h)$

$r = v.left.size + 1$
if $i == r$ **then return** v
else
    if $i < r$ **then**
        return Select($v.left$, i)
    else
        return Select($v.right$, $i - r$)

Rank(Node v): $O(\quad)$

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
    if $u == u.p.right$ **then**
        $r = r + u.p.left.size + 1$
    $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

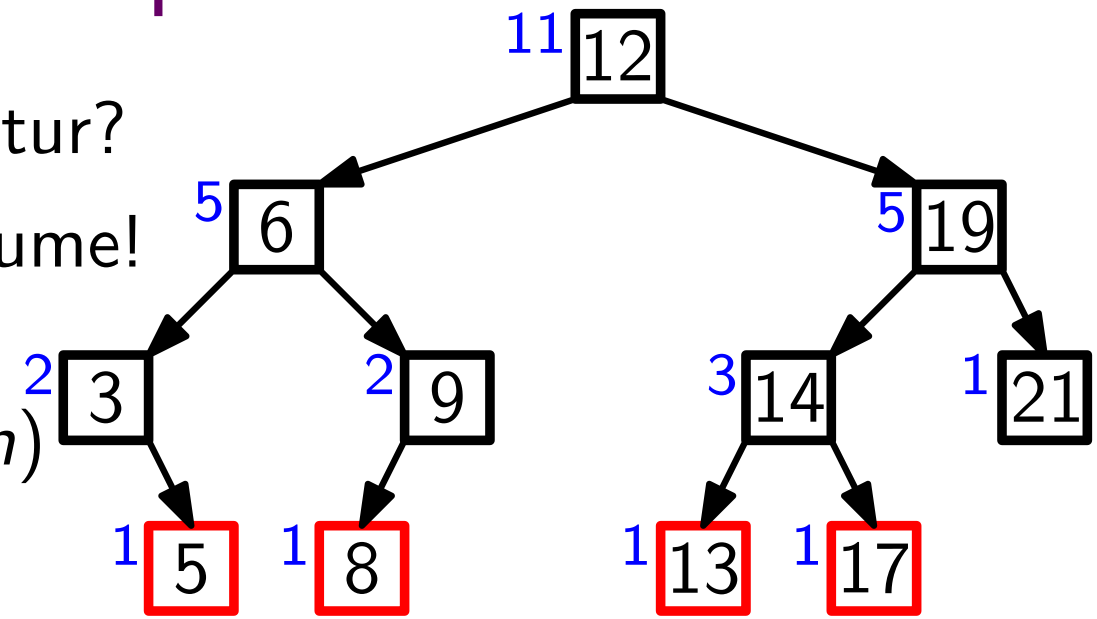
4. Select(Node $v = root$, int i): $O(h)$

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left$, i)
  else
    return Select($v.right$, $i - r$)

Rank(Node v): $O(\ )$

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Das dynamische Auswahlproblem



1. Welche Ausgangsdatenstruktur?
   – balancierte binäre Suchbäume!
   z.B. Rot-Schwarz-Bäume
   $\Rightarrow$ Baumhöhe $h \in O(\log n)$

2. Welche Extrainformation aufrechterhalten?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

4. Select(Node $v = root$, int i): $O(h)$

$r = v.left.size + 1$
if $i == r$ **then return** v
else
  if $i < r$ **then**
    return Select($v.left$, i)
  else
    return Select($v.right$, $i - r$)

Rank(Node v): $O(h)$

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante:

Rank(Node v):

$r = v.\mathit{left}.\mathit{size} + 1$
$u = v$
while $u \neq \mathit{root}$ **do**
  if $u == u.p.\mathit{right}$ **then**
    $r = r + u.p.\mathit{left}.\mathit{size} + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der Rang von v im Teilbaum mit Wurzel u.

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung

Vor 1. Iteration gilt $u = v \Rightarrow u\text{-Rang}(v) = v.\mathit{left}.\mathit{size} + 1$.

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung

Vor 1. Iteration gilt $u = v \Rightarrow u\text{-Rang}(v) = v.\mathit{left}.\mathit{size} + 1$. ✓

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung

Rank(Node v):

$r = v.\mathit{left}.\mathit{size} + 1$
$u = v$
while $u \neq \mathit{root}$ **do**
  if $u == u.p.\mathit{right}$ **then**
    $r = r + u.p.\mathit{left}.\mathit{size} + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung

Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

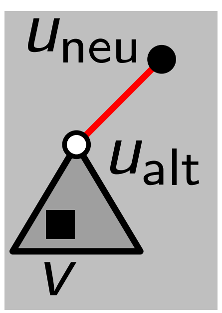
Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung

Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.

Rank(Node v):

$r = v.\mathit{left}.\mathit{size} + 1$
$u = v$
while $u \neq \mathit{root}$ **do**
  if $u == u.p.\mathit{right}$ **then**
    $r = r + u.p.\mathit{left}.\mathit{size} + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

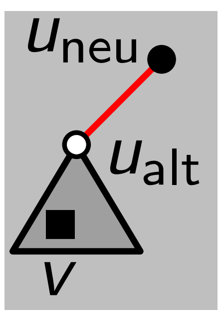
Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung

Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.

$\Rightarrow$ u-Rang von v bleibt gleich.

Rank(Node v):

$r = v.\mathit{left}.\mathit{size} + 1$
$u = v$
while $u \neq \mathit{root}$ **do**
  if $u == u.p.\mathit{right}$ **then**
    $r = r + u.p.\mathit{left}.\mathit{size} + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

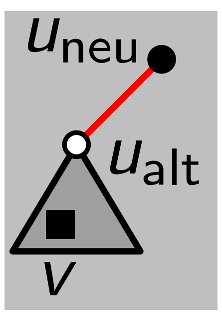
Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung

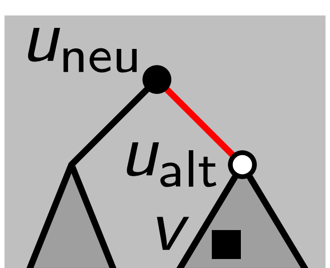
Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.

$\Rightarrow$ u-Rang von v bleibt gleich.

2. Fall: u war rechtes Kind.



Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

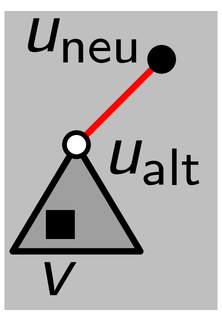
Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung

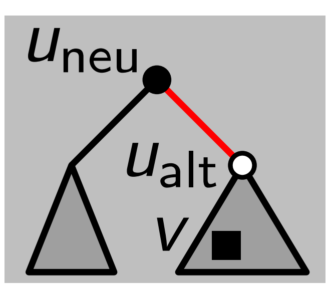
Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.

$\Rightarrow$ u-Rang von v bleibt gleich.

2. Fall: u war rechtes Kind.



$\Rightarrow$ u-Rang von v erhöht sich um Größe des li. Teilbaums von u plus 1 (für u selbst).

Rank(Node v):

$r = v.\mathit{left}.\mathit{size} + 1$
$u = v$
while $u \neq \mathit{root}$ **do**
    if $u == u.p.\mathit{right}$ **then**
        $r = r + u.p.\mathit{left}.\mathit{size} + 1$
    $u = u.p$
return r (vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

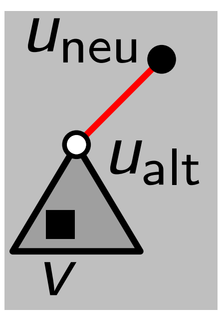
Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

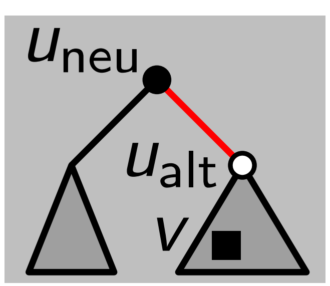
2.) Aufrechterhaltung ✓

Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.

$\Rightarrow$ u-Rang von v bleibt gleich.



2. Fall: u war rechtes Kind.

$\Rightarrow$ u-Rang von v erhöht sich um Größe des li. Teilbaums von u plus 1 (für u selbst).

Rank(Node v):

```
r = v.left.size + 1
u = v
while u ≠ root do
    if u == u.p.right then
        r = r + u.p.left.size + 1
    u = u.p
return r
```

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung

Rank(Node v):

$r = v.\mathit{left}.\mathit{size} + 1$
$u = v$
while $u \neq \mathit{root}$ **do**
  if $u == u.p.\mathit{right}$ **then**
    $r = r + u.p.\mathit{left}.\mathit{size} + 1$
  $u = u.p$
return r

(vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung

Bei Schleifenabbruch: $u = root$.
$\Rightarrow r = u\text{-Rang}(v) = \text{Rang}(v)$.

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

Bei Schleifenabbruch: $u = root$.
$\Rightarrow r = u\text{-Rang}(v) = \text{Rang}(v)$.

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der $\underbrace{\text{Rang von } v \text{ im Teilbaum mit Wurzel } u}_{u\text{-Rang von } v}$.

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

Bei Schleifenabbruch: $u = root$.
$\Rightarrow r = u\text{-Rang}(v) = \text{Rang}(v)$.

Zusammenfassung:

Die Methode Rank() liefert wie gewünscht den Rang des übergebenen Knotens.

Rank(Node v):

$r = v.left.size + 1$
$u = v$
while $u \neq root$ **do**
  if $u == u.p.right$ **then**
    $r = r + u.p.left.size + 1$
  $u = u.p$
return r (vorausgesetzt, dass $T.nil.size = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Für alle Knoten v auf dem Weg von der Wurzel zu z:

Erhöhe $v.size$ um 1.

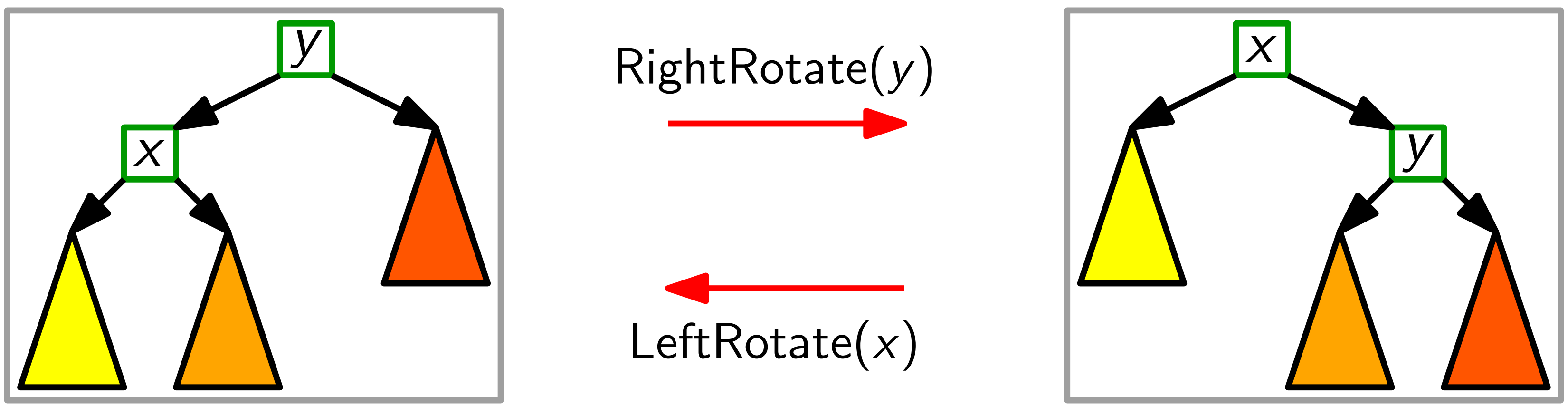
Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.
Für alle Knoten v auf dem Weg von der Wurzel zu z:
Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



3. Aufwand zur Aufrechterhaltung der Extrainformation?

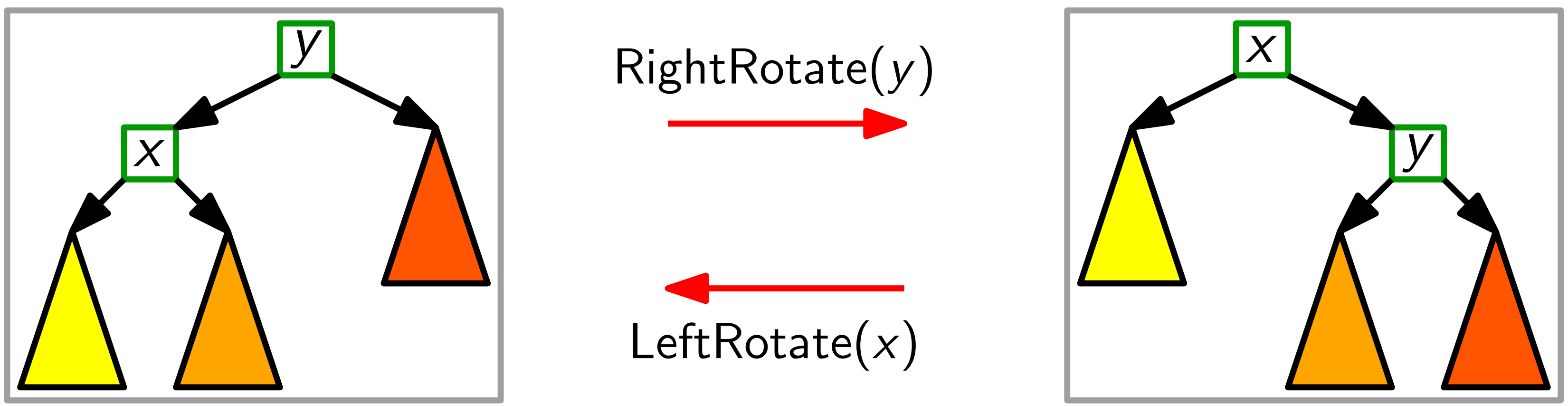
RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Für alle Knoten v auf dem Weg von der Wurzel zu z:

Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle $size$-Einträge wieder stimmen?

$x.size =$

$y.size =$

3. Aufwand zur Aufrechterhaltung der Extrainformation?

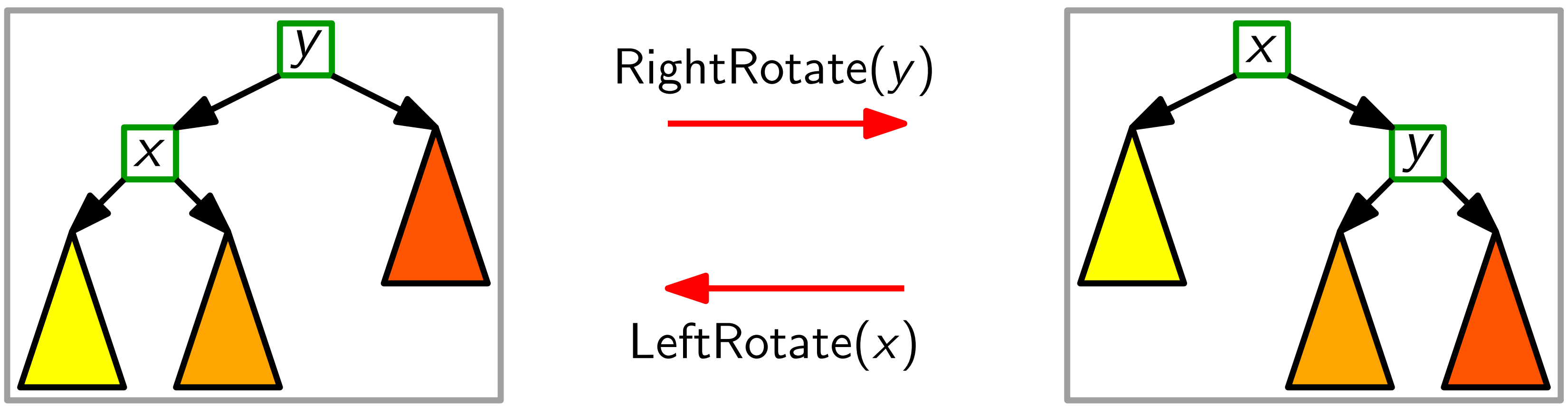
RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Für alle Knoten v auf dem Weg von der Wurzel zu z:

Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle $size$-Einträge wieder stimmen?

$x.size = y.size$

$y.size =$

3. Aufwand zur Aufrechterhaltung der Extrainformation?

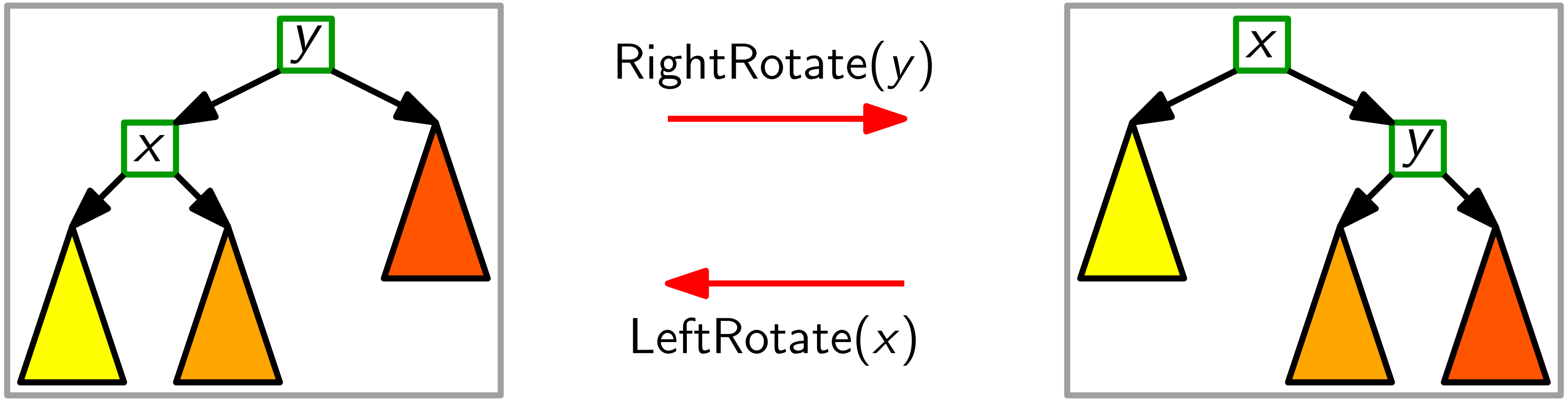
RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Für alle Knoten v auf dem Weg von der Wurzel zu z:

Erhöhe $v.\mathit{size}$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle size-Einträge wieder stimmen?

$$\boxed{\begin{aligned} x.\mathit{size} &= y.\mathit{size} \\ y.\mathit{size} &= y.\mathit{left}.\mathit{size} + y.\mathit{right}.\mathit{size} + 1 \end{aligned}}$$

3. Aufwand zur Aufrechterhaltung der Extrainformation?

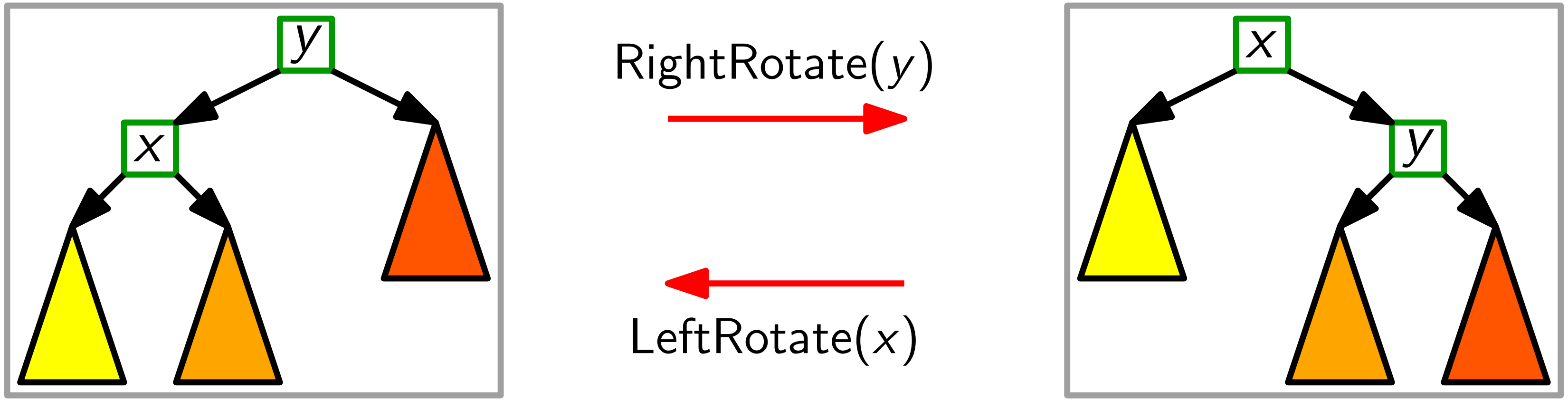
RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Für alle Knoten v auf dem Weg von der Wurzel zu z:

Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle $size$-Einträge wieder stimmen?

$$x.size = y.size$$

$$y.size = y.left.size + y.right.size + 1$$

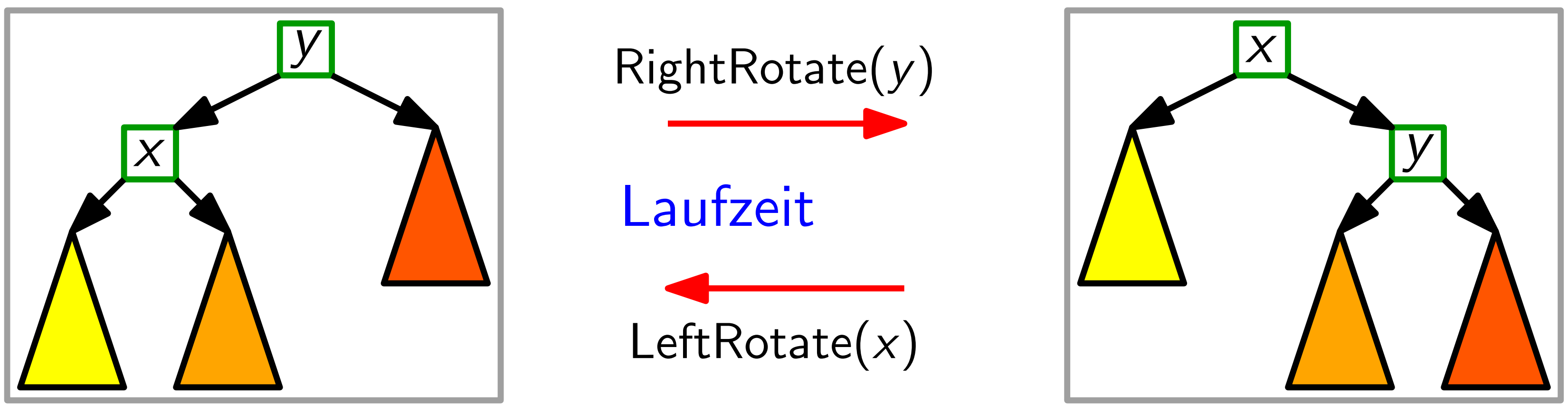
(vorausgesetzt, dass $T.nil.size = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.
Für alle Knoten v auf dem Weg von der Wurzel zu z:
Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle $size$-Einträge wieder stimmen?

$$x.size = y.size$$
$$y.size = y.left.size + y.right.size + 1$$

(vorausgesetzt, dass $T.nil.size = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

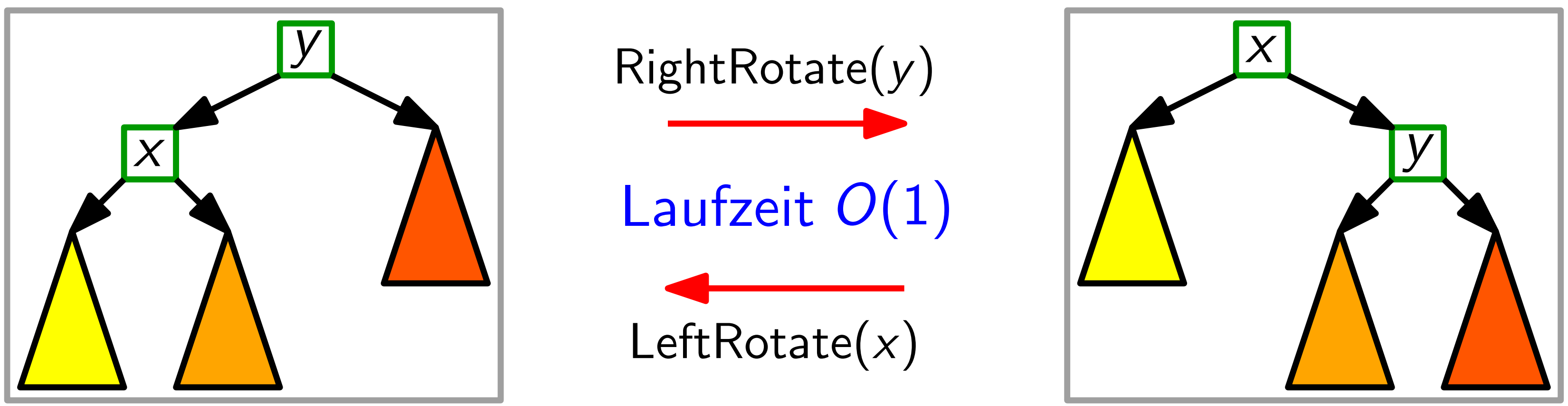
RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Für alle Knoten v auf dem Weg von der Wurzel zu z:

Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle $size$-Einträge wieder stimmen?

$$x.size = y.size$$
$$y.size = y.left.size + y.right.size + 1$$

(vorausgesetzt, dass $T.nil.size = 0$)

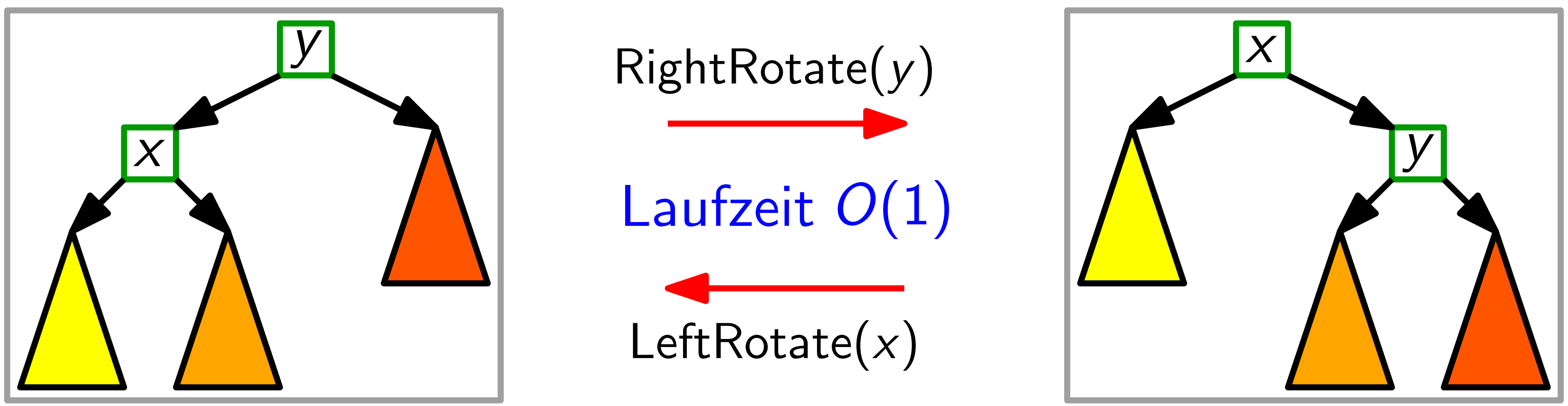
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $\left\{\begin{array}{l} \textit{Für alle Knoten } v \textit{ auf dem Weg von der Wurzel zu } z\textit{:} \\ \quad \text{Erhöhe } v.\mathit{size} \text{ um } 1. \end{array}\right.$

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle size-Einträge wieder stimmen?

$$x.\mathit{size} = y.\mathit{size}$$
$$y.\mathit{size} = y.\mathit{left}.\mathit{size} + y.\mathit{right}.\mathit{size} + 1$$

(vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

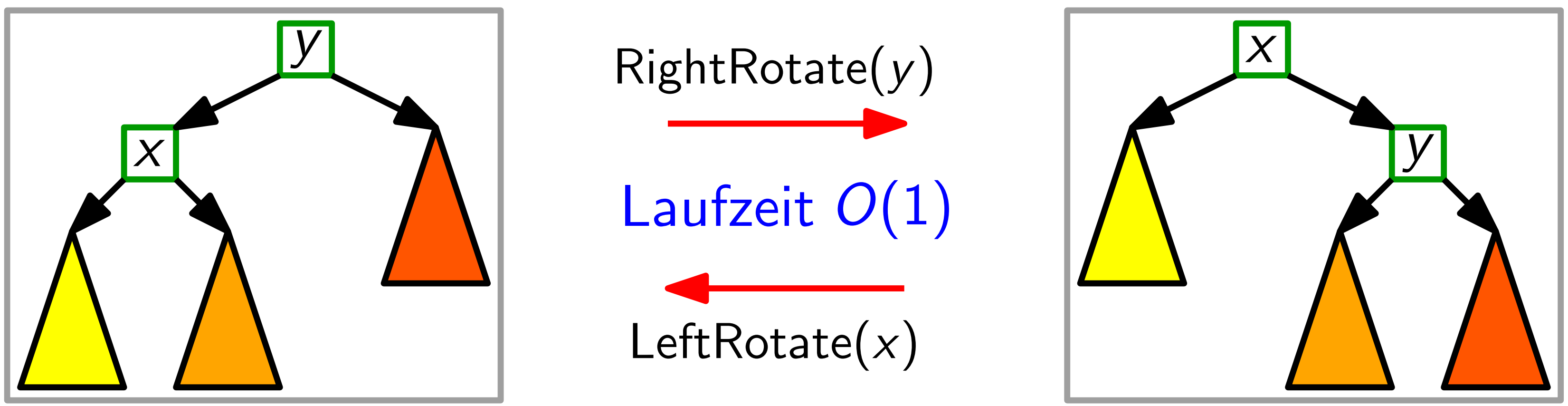
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $O(h)$ $\begin{cases} \textit{Für alle Knoten } v \textit{ auf dem Weg von der Wurzel zu } z\textit{:} \\ \quad \text{Erhöhe } v.\mathit{size} \text{ um } 1. \end{cases}$

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle size-Einträge wieder stimmen?

$$x.\mathit{size} = y.\mathit{size}$$
$$y.\mathit{size} = y.\mathit{left}.\mathit{size} + y.\mathit{right}.\mathit{size} + 1$$

(vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

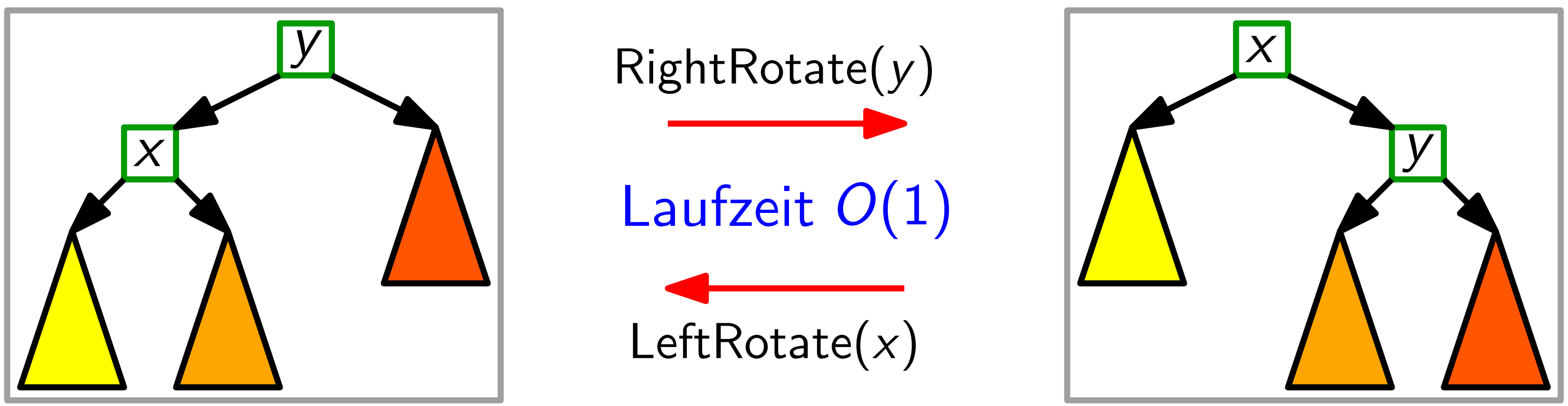
zusätzliche Laufzeit fürs Einfügen:

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $O(h)$

Für alle Knoten v auf dem Weg von der Wurzel zu z:
Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle $size$-Einträge wieder stimmen?

$$x.size = y.size$$
$$y.size = y.left.size + y.right.size + 1$$

(vorausgesetzt, dass $T.nil.size = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

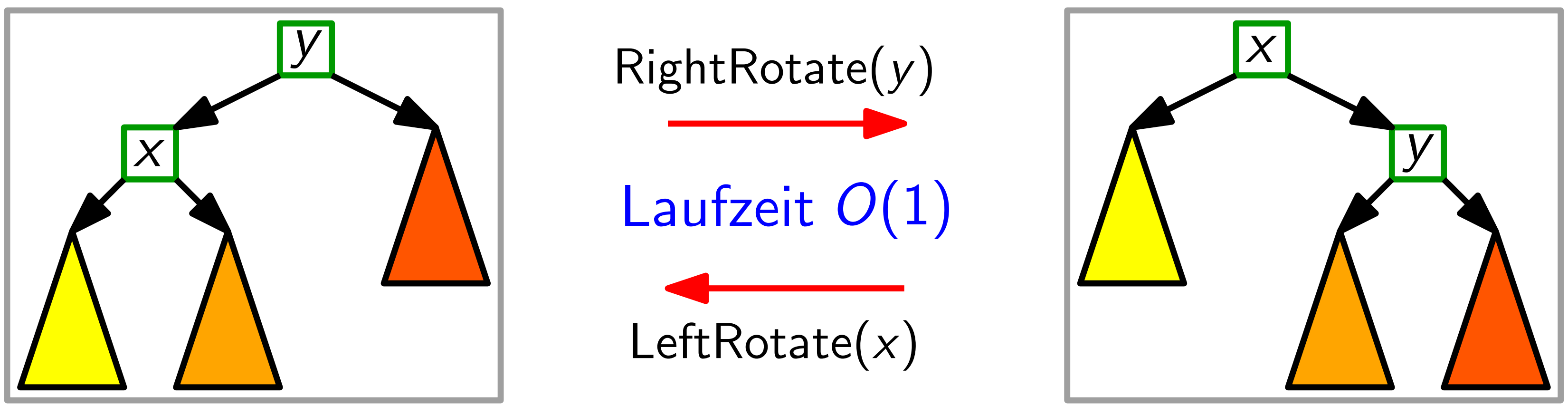
RBInsert() geht in zwei Phasen vor:

zusätzliche Laufzeit fürs Einfügen: $O(h)$

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $O(h)$ $\begin{cases} \textit{Für alle Knoten } v \textit{ auf dem Weg von der Wurzel zu } z\textit{:} \\ \quad \text{Erhöhe } v.\mathit{size} \text{ um } 1. \end{cases}$

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle size-Einträge wieder stimmen?

$$x.\mathit{size} = y.\mathit{size}$$
$$y.\mathit{size} = y.\mathit{left}.\mathit{size} + y.\mathit{right}.\mathit{size} + 1$$

(vorausgesetzt, dass $T.\mathit{nil}.\mathit{size} = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

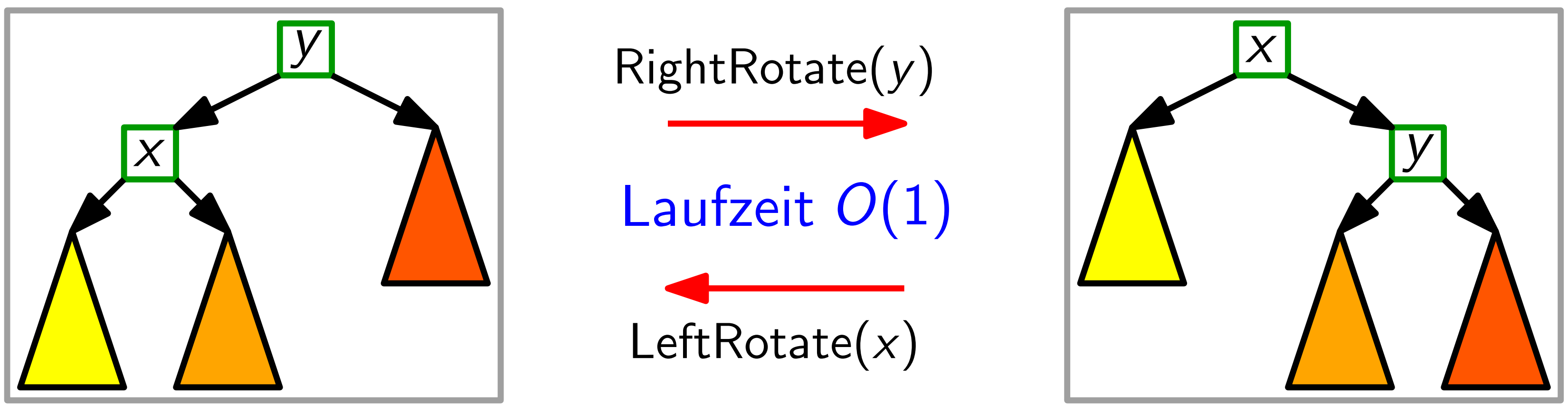
RBInsert() geht in zwei Phasen vor:

zusätzliche Laufzeit fürs Einfügen: $O(h)$

Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $O(h)$ $\left\{ \begin{array}{l} \textit{Für alle Knoten } v \textit{ auf dem Weg von der Wurzel zu } z\textit{:} \\ \quad \text{Erhöhe } v.size \text{ um } 1. \end{array} \right.$

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 *Rotationen*:



Welche Befehle müssen wir an RightRotate(Node y) anhängen, damit nach der Rotation alle *size*-Einträge wieder stimmen?

$$x.size = y.size$$
$$y.size = y.left.size + y.right.size + 1$$

(vorausgesetzt, dass $T.nil.size = 0$)

[RBDelete() kann man analog „upgraden“.]

Ergebnis

Satz. Das dynamische Auswahlproblem kann man so lösen, dass Select() und Rank() sowie alle gewöhnlichen Operationen für dynamische Mengen in einer Menge von n Elementen in $O(\log n)$ Zeit laufen.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v, $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) berechnen.

$$\Downarrow$$

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v, $v.\mathit{left}$, $v.\mathit{right}$ (inklusive $f(v.\mathit{left})$ und $f(v.\mathit{right})$) berechnen.

$$\Downarrow$$

Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v, $v.\mathit{left}$, $v.\mathit{right}$ (inklusive $f(v.\mathit{left})$ und $f(v.\mathit{right})$) berechnen.

$$\Downarrow$$

Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Beweisidee.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v, $v.\mathit{left}$, $v.\mathit{right}$ (inklusive $f(v.\mathit{left})$ und $f(v.\mathit{right})$) berechnen.

$$\Downarrow$$

Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Beweisidee. Im Prinzip wie im Spezialfall $f \equiv \mathit{size}$.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v, $v.\mathit{left}$, $v.\mathit{right}$ (inklusive $f(v.\mathit{left})$ und $f(v.\mathit{right})$) berechnen.

$$\Downarrow$$

Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Beweisidee. Im Prinzip wie im Spezialfall $f \equiv \mathit{size}$.

Allerdings ist es im Prinzip möglich, dass sich die Veränderungen von einem gewissen veränderten Knoten bis in die Wurzel hochpropagieren.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v, $v.\mathit{left}$, $v.\mathit{right}$ (inklusive $f(v.\mathit{left})$ und $f(v.\mathit{right})$) berechnen.

$$\Downarrow$$

Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Beweisidee. Im Prinzip wie im Spezialfall $f \equiv \mathit{size}$.

Allerdings ist es im Prinzip möglich, dass sich die Veränderungen von einem gewissen veränderten Knoten bis in die Wurzel hochpropagieren. [Details Kapitel 14.2, CLRS]

Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

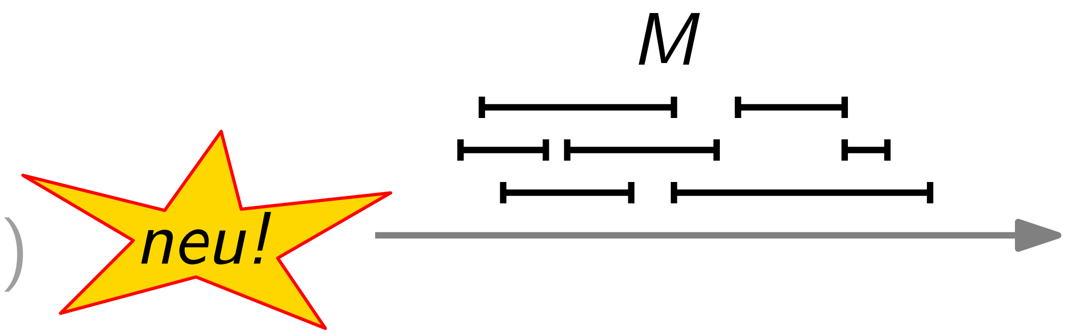
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element **Insert**(Interval i)
- **Delete**(Element e)
- Element **Search**(Interval i)



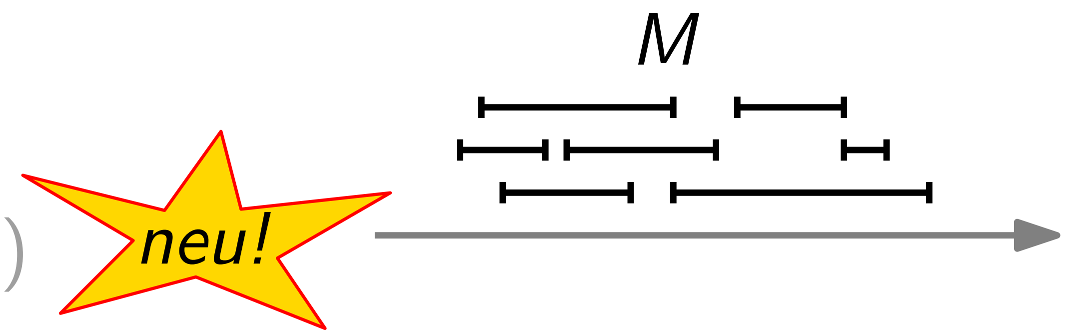
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element Insert(Interval i)
- Delete(Element e)
- Element Search(Interval i)



liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst nil.

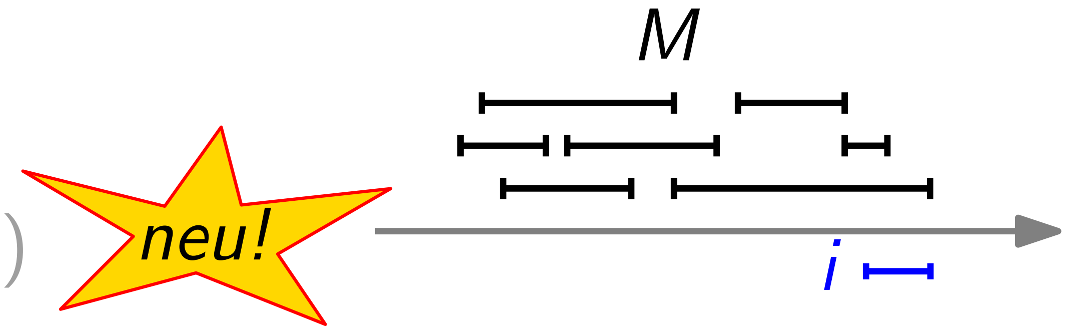
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element Insert(Interval i)
- Delete(Element e)
- Element Search(Interval i)



liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst nil.

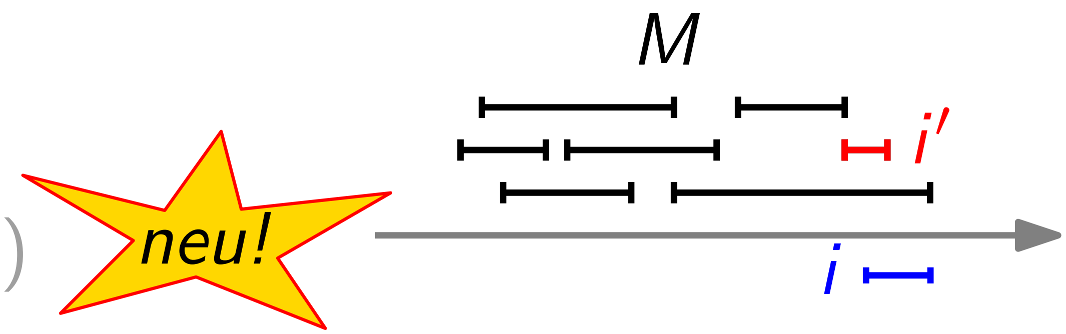
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element Insert(Interval i)
- Delete(Element e)
- Element Search(Interval i)



  liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst nil.

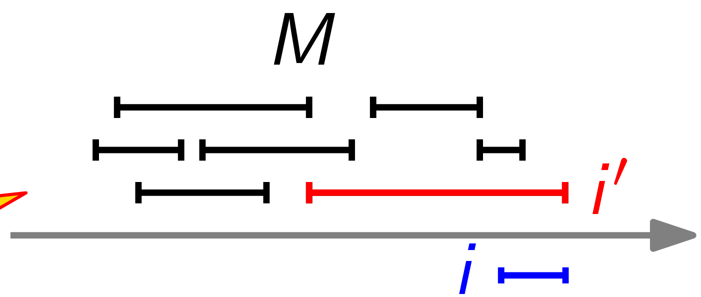
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element **Insert**(Interval i)
- **Delete**(Element e)
- Element **Search**(Interval i)



  liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst nil.

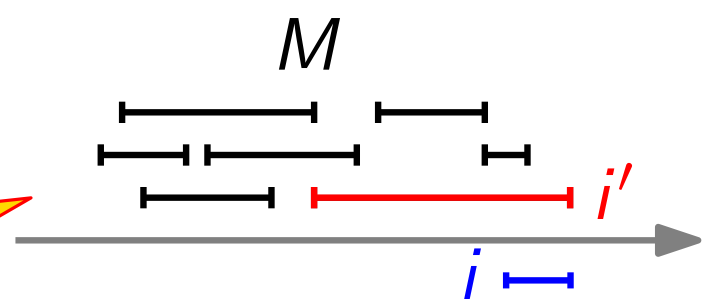
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element **Insert**(Interval i)
- **Delete**(Element e)
- Element **Search**(Interval i)



  liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst nil.

Bitte lesen Sie's und stellen Sie Fragen...