

Algorithmen und Datenstrukturen

Vorlesung 15: Augmentieren von Datenstrukturen

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- Stapel



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- Stapel



- Schlange



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

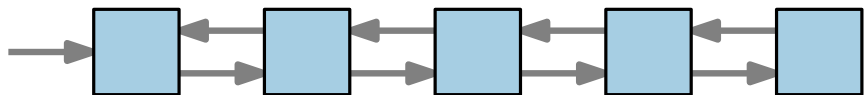
- Stapel



- Schlange



- doppelt verkettete Liste



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

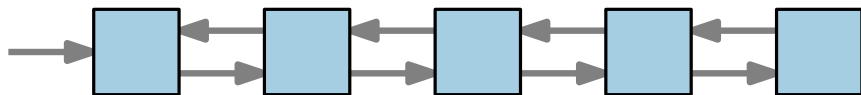
- Stapel



- Schlange



- doppelt verkettete Liste



- Hashtabelle



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

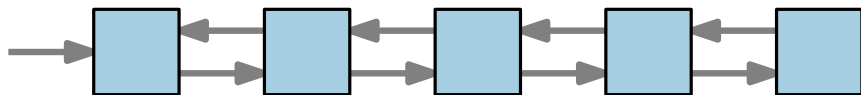
- Stapel



- Schlange



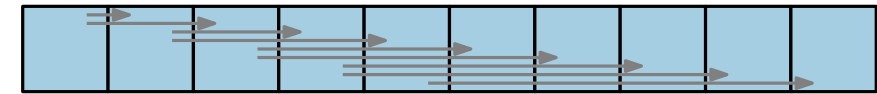
- doppelt verkettete Liste



- Hashtabelle



- Heap



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

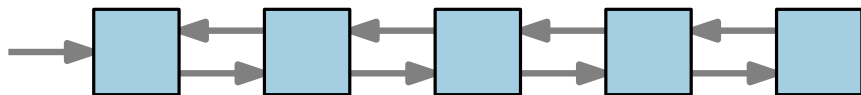
- Stapel



- Schlange



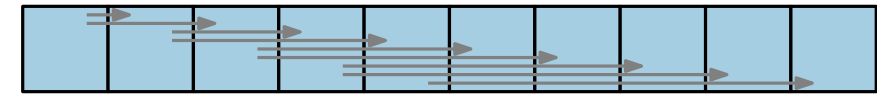
- doppelt verkettete Liste



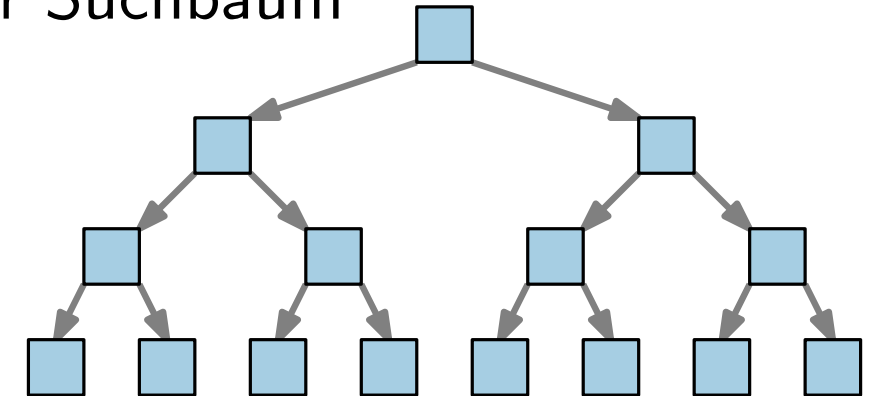
- Hashtabelle



- Heap



- binärer Suchbaum



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

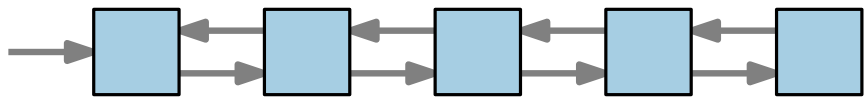
■ Stapel



■ Schlange



■ doppelt verkettete Liste

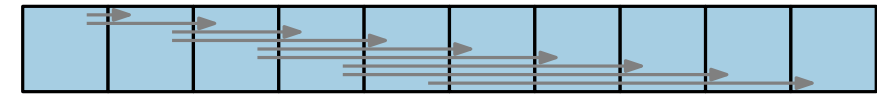


Allerdings gibt es viele Situationen, wo keine davon **genau** passt.

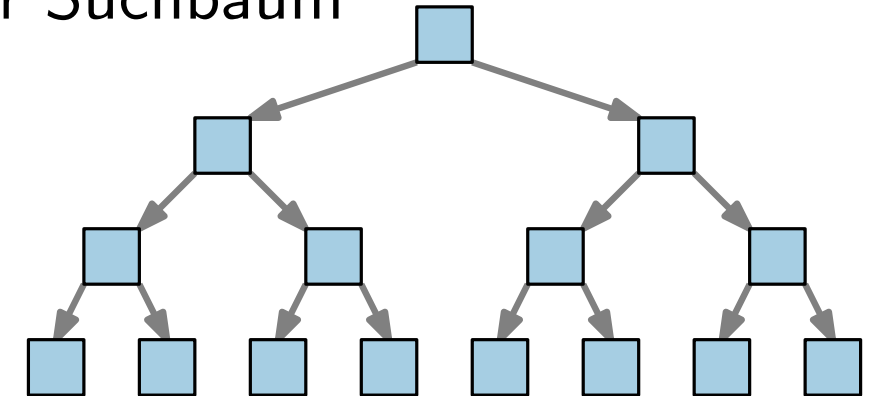
■ Hashtabelle



■ Heap



■ binärer Suchbaum



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

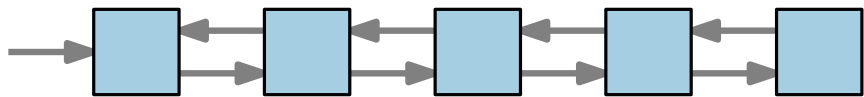
■ Stapel



■ Schlange



■ doppelt verkettete Liste



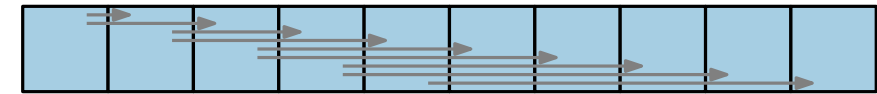
Allerdings gibt es viele Situationen, wo keine davon **genau** passt.

Herangehensweise: **Augmentieren** von Datenstrukturen

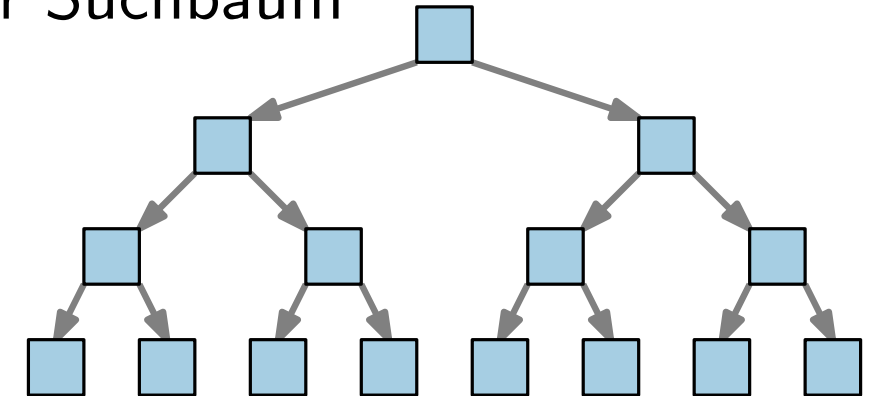
■ Hashtabelle



■ Heap



■ binärer Suchbaum



Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

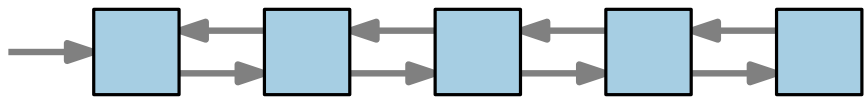
■ Stapel



■ Schlange



■ doppelt verkettete Liste



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

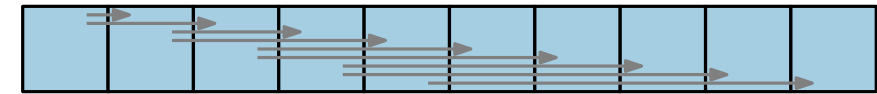
Herangehensweise: **Augmentieren** von Datenstrukturen

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

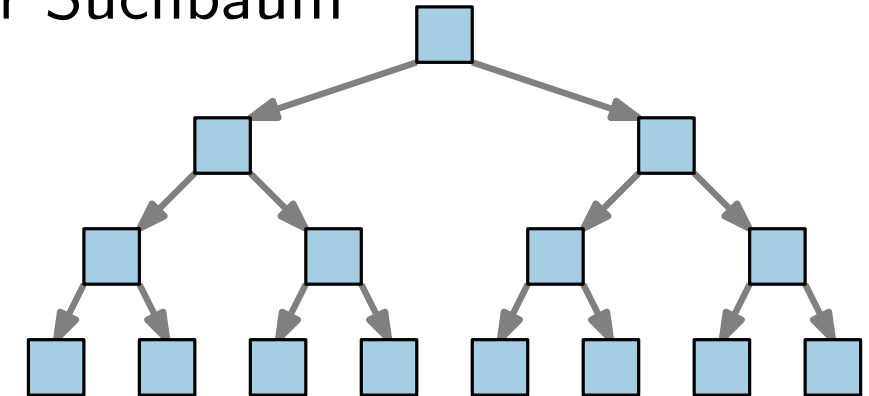
■ Hashtabelle



■ Heap



■ binärer Suchbaum



Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation aufrechterhalten?

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation aufrechterhalten?

- Summe der Elemente (*sum*)
- Anzahl der Elemente (*size*)

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation aufrechterhalten?

- Summe der Elemente (*sum*)
- Anzahl der Elemente (*size*)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation 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 von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation 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 von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation 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!

```
double MEAN()  
    return sum/size
```

Ein Beispiel

Bestimme für eine dynamische Menge von Zahlen den **Mittelwert**.

1. Welche Ausgangsdatenstruktur?

- Beliebig, z.B. Liste

2. Welche Zusatzinformation 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!

```
double MEAN()  
    return sum/size
```

Übung.

Tun Sie das gleiche für die Standardabweichung

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

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

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):

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): `ptr SELECT(int i)`

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): `ptr SELECT(int i)`
- 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): `ptr SELECT(int i)`
- den **Rang** eines Elements: `int RANK(ptr x)`

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): `ptr SELECT(int i)`
- den **Rang** eines Elements: `int RANK(ptr x)`

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): `ptr SELECT(int i)`
- den **Rang** eines Elements: `int RANK(ptr x)`

in einer dynamischen Menge bestimmen können.

Fahrplan: **1.** Welche Ausgangsdatenstruktur?

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): `ptr SELECT(int i)`
- den **Rang** eines Elements: `int RANK(ptr x)`

in einer dynamischen Menge bestimmen können.

- Fahrplan:**
1. Welche Ausgangsdatenstruktur?
 2. Welche Extrainformation aufrechterhalten?

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): `ptr SELECT(int i)`
- den **Rang** eines Elements: `int RANK(ptr x)`

in einer dynamischen Menge bestimmen können.

- Fahrplan:**
1. Welche Ausgangsdatenstruktur?
 2. Welche Extrainformation aufrechterhalten?
 3. Aufwand zur Aufrechterhaltung?

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): `ptr SELECT(int i)`
- den **Rang** eines Elements: `int RANK(ptr x)`

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

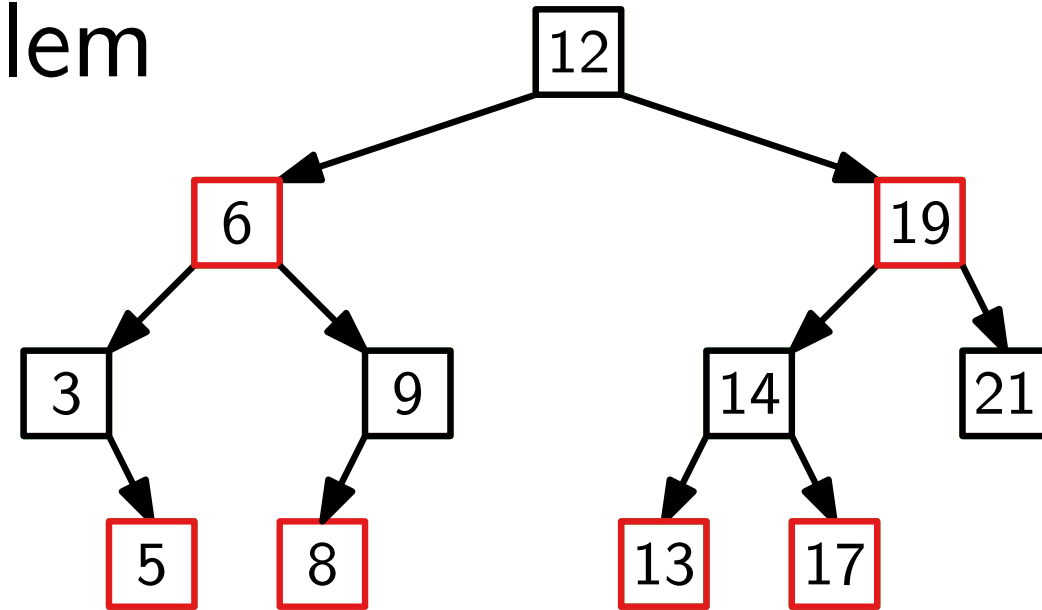
1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
 \Rightarrow Baumhöhe $h \in \mathcal{O}(\log n)$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

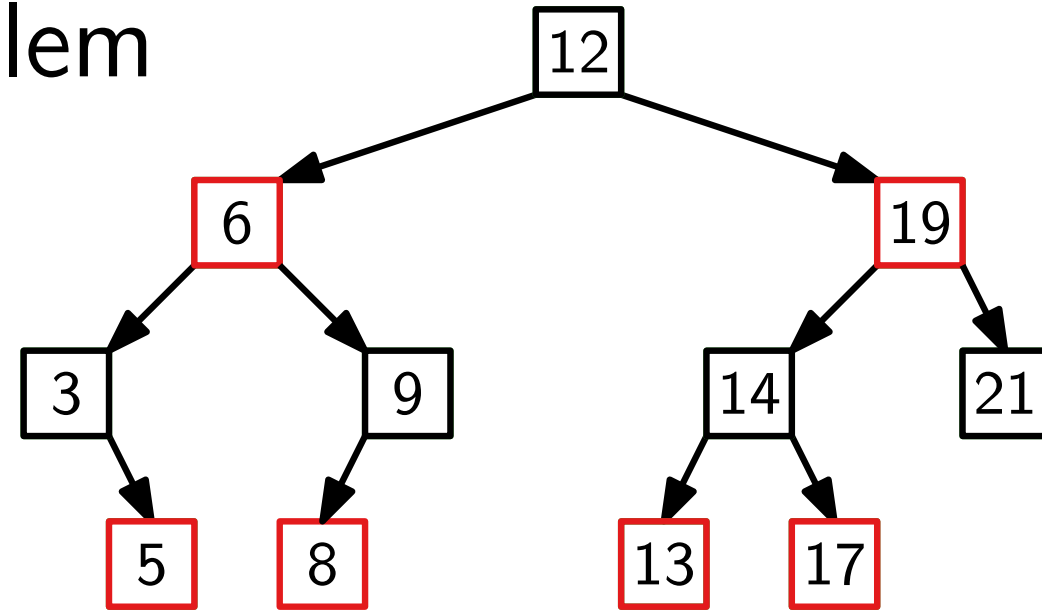
- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
 \Rightarrow Baumhöhe $h \in \mathcal{O}(\log n)$

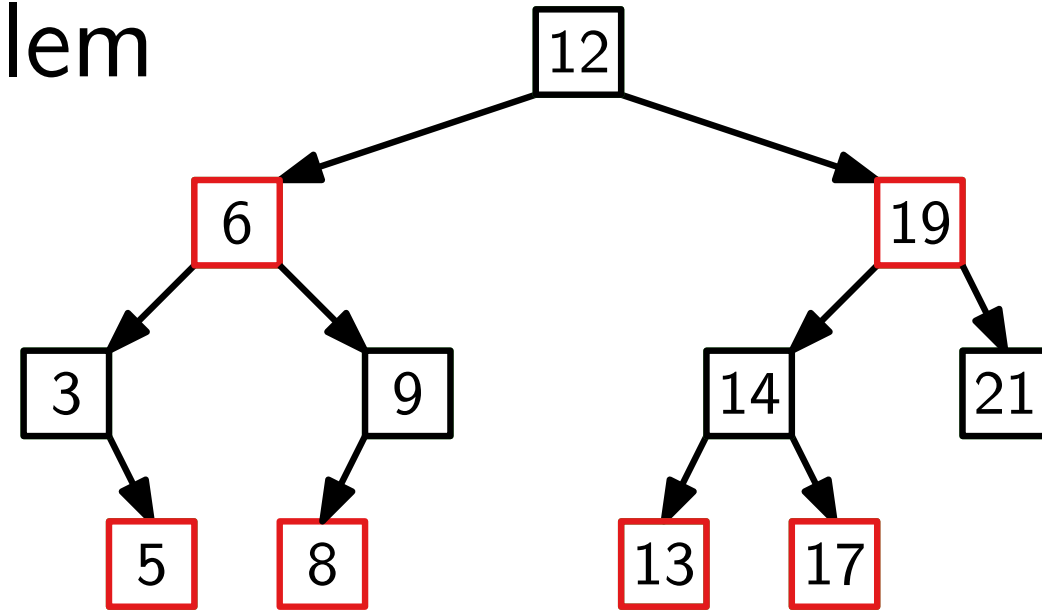


2. Welche Extrainformation aufrechterhalten?

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
 \Rightarrow Baumhöhe $h \in \mathcal{O}(\log n)$



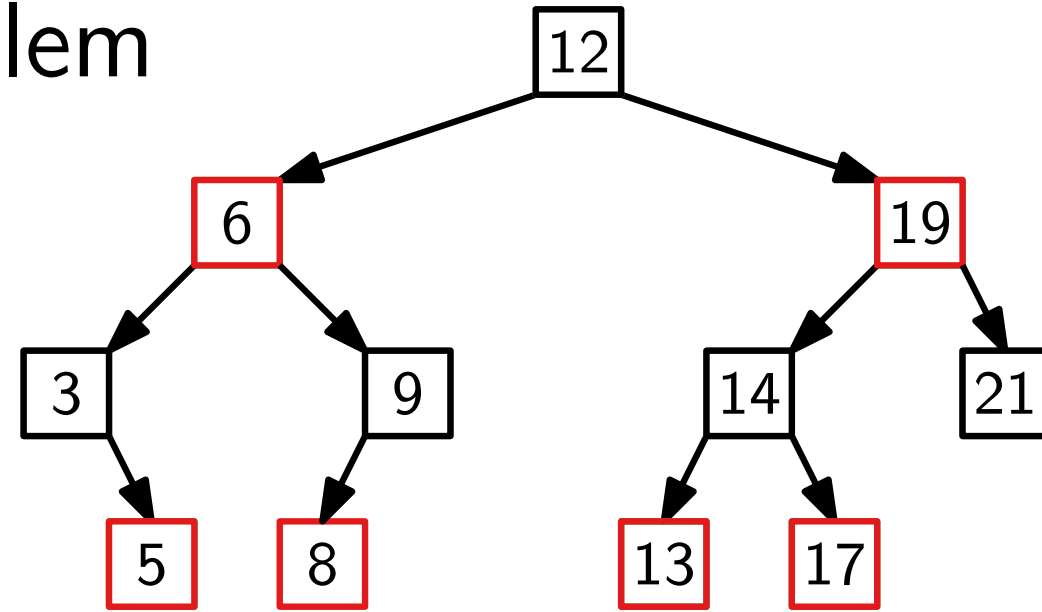
2. Welche Extrainformation aufrechterhalten?

- gar keine?

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

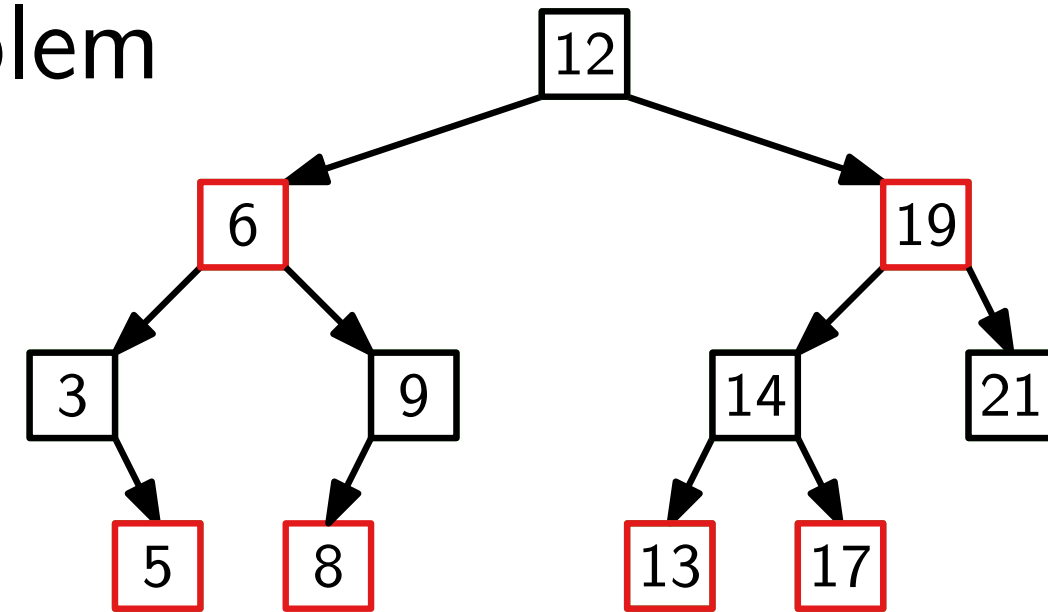
- gar keine?

3. Aufwand zur Aufrechterhaltung?

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

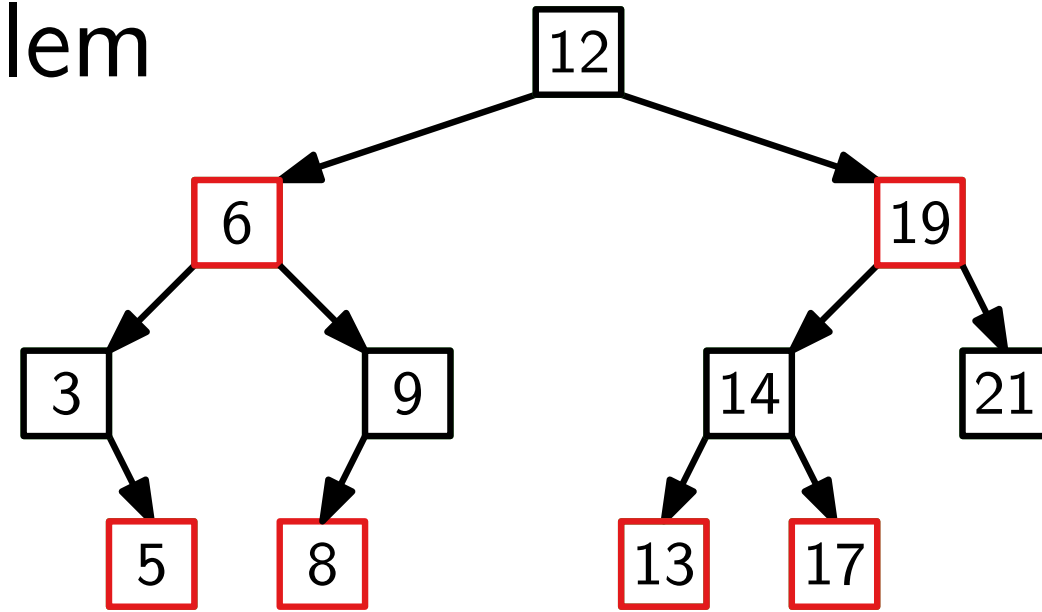
3. Aufwand zur Aufrechterhaltung?

- gar keiner

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

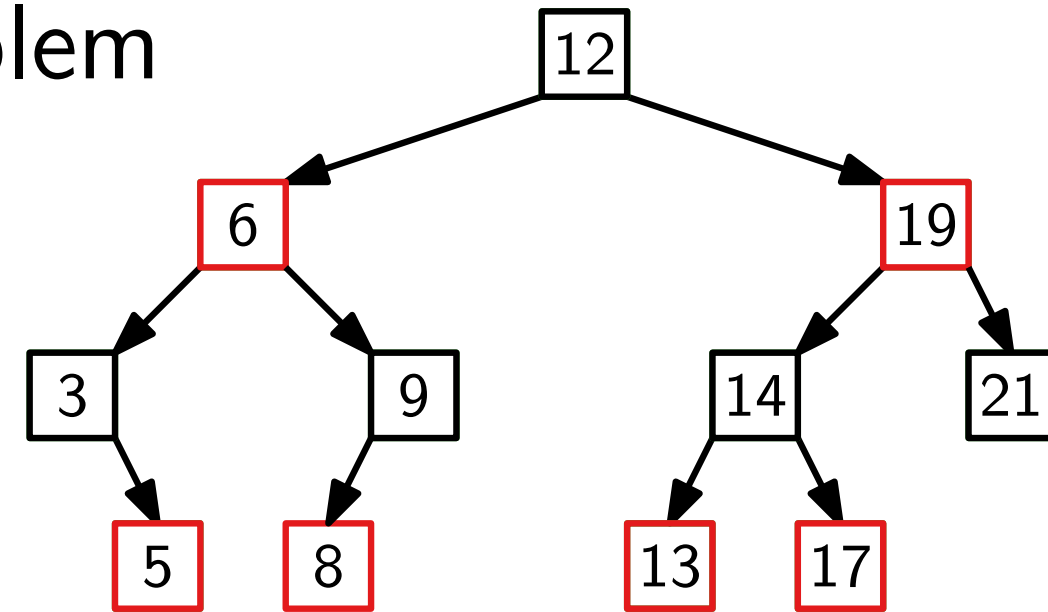
- gar keiner

4. Implementiere neue Operationen!

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Node SELECT(int i):

liefert i .-kleinstes Element

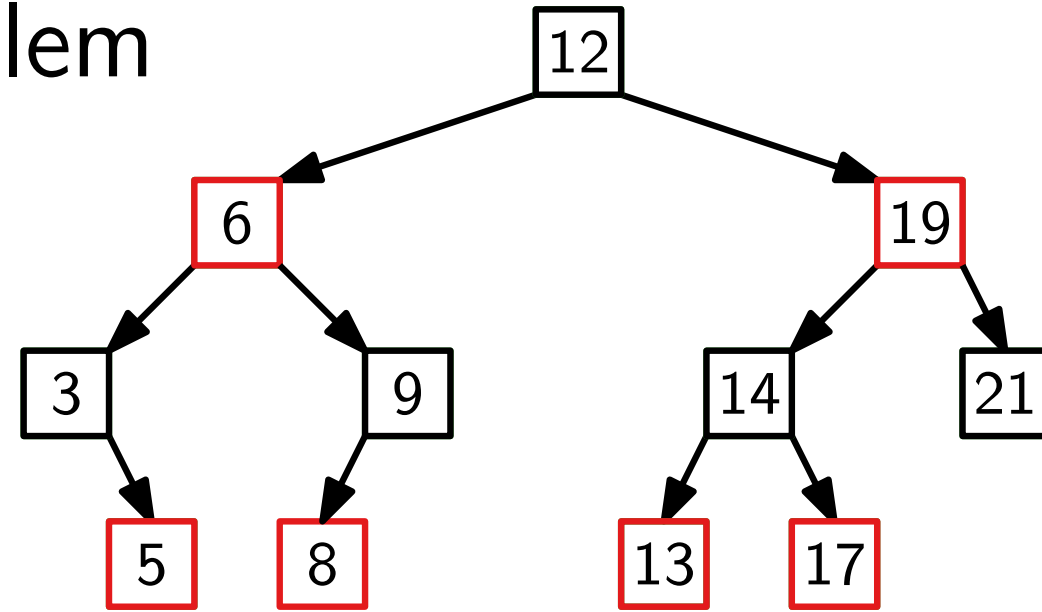
int RANK(Node x):

Wieviertes Element ist v ?

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

`Node SELECT(int i):`

liefert i .-kleinstes Element

`int RANK(Node x):`

Wieviertes Element ist v ?

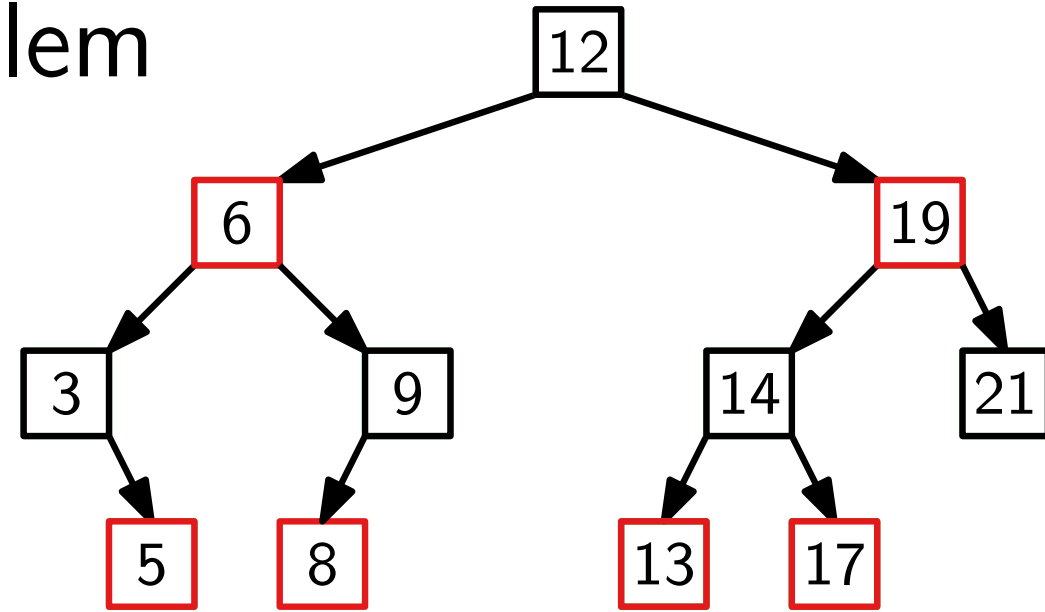
Aufgabe.

Schreiben Sie Pseudocode für `SELECT()` und `RANK()`
unter Benutzung von `SUCCESSOR()` und `PREDECESSOR()`

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

```
Node SELECT(int i):
```

```
    x = MINIMUM()
```

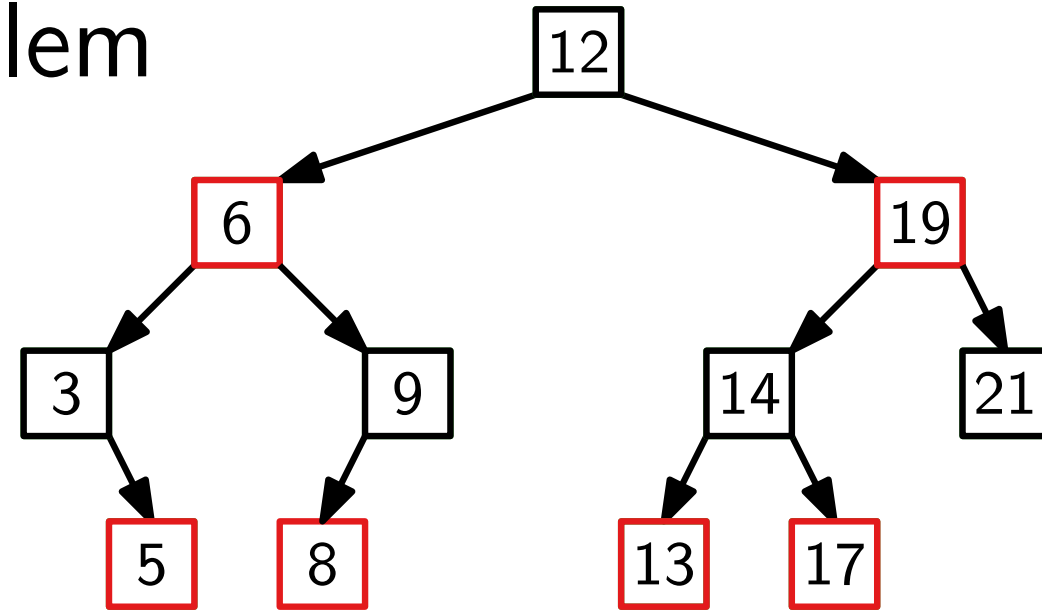
```
    return x
```

```
int RANK(Node x):
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    |
```

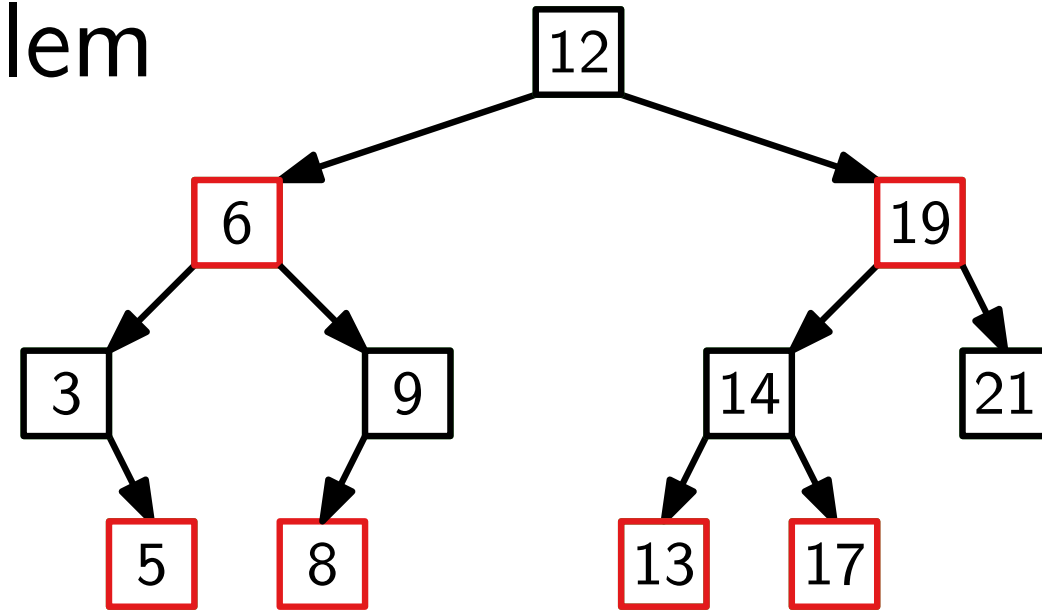
```
  return x
```

```
int RANK(Node x):
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    | x = SUCCESSOR(x)
```

```
    | i = i - 1
```

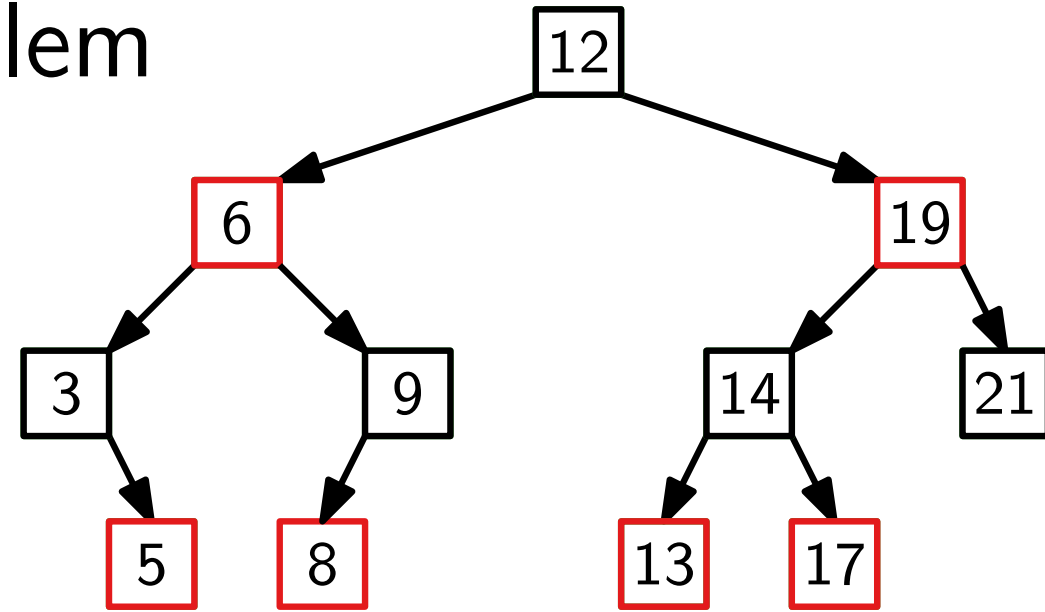
```
  return x
```

```
int RANK(Node x):
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):
```

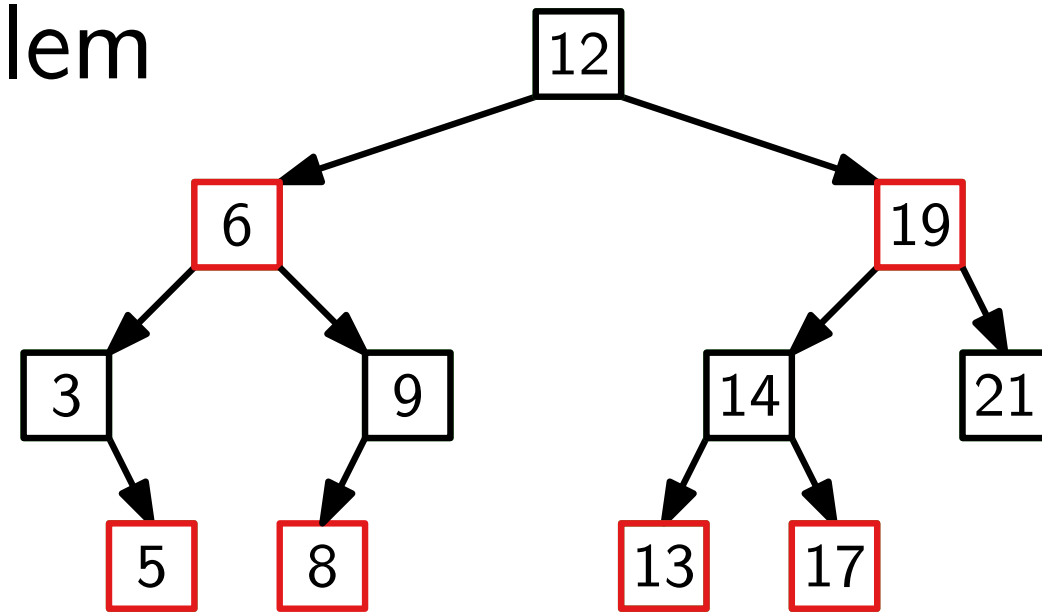
```
  i = 0
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    | x = SUCCESSOR(x)
```

```
    | i = i - 1
```

```
  return x
```

```
int RANK(Node x):
```

```
  i = 0
```

```
  while x ≠ nil do
```

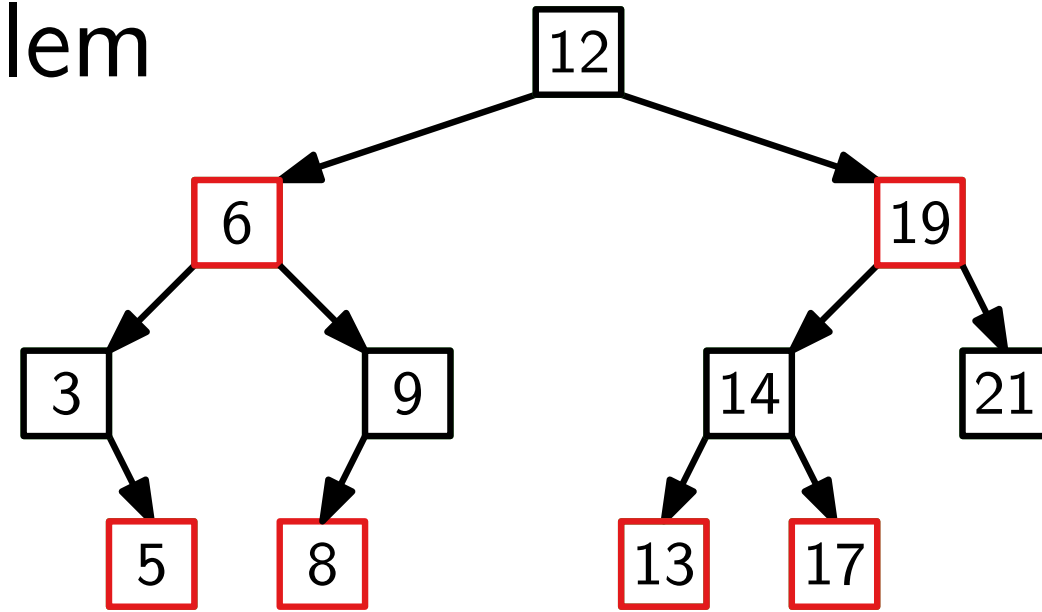
```
    |
```

```
  return i
```


Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)
```

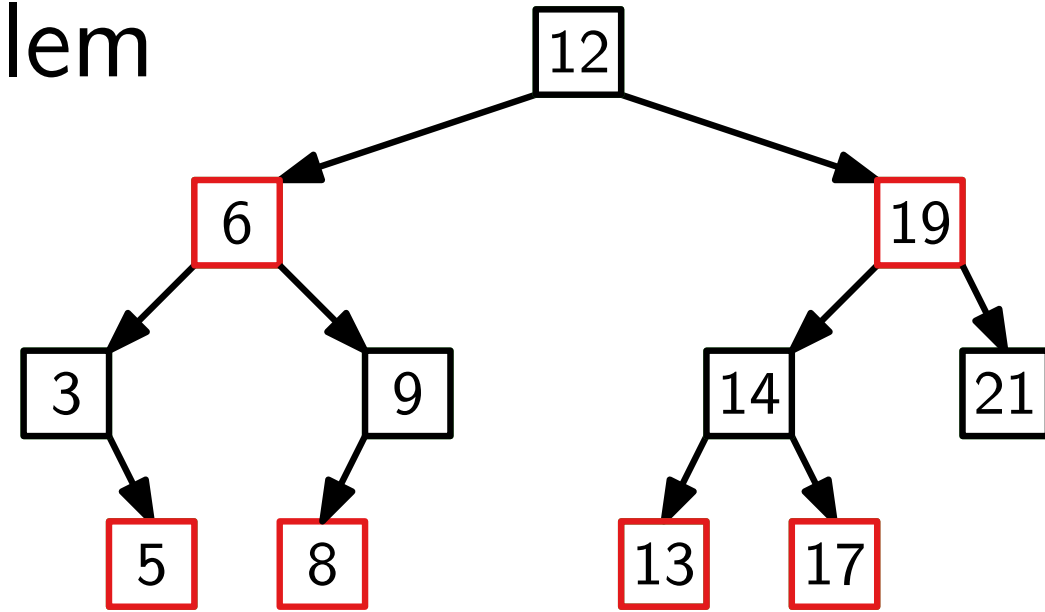
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)
```

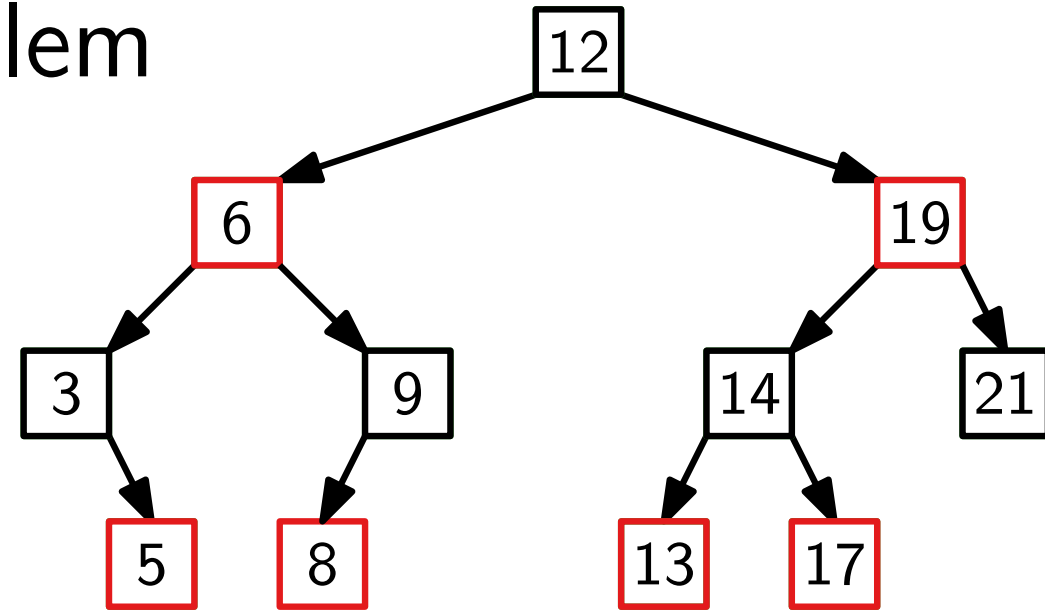
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)
```

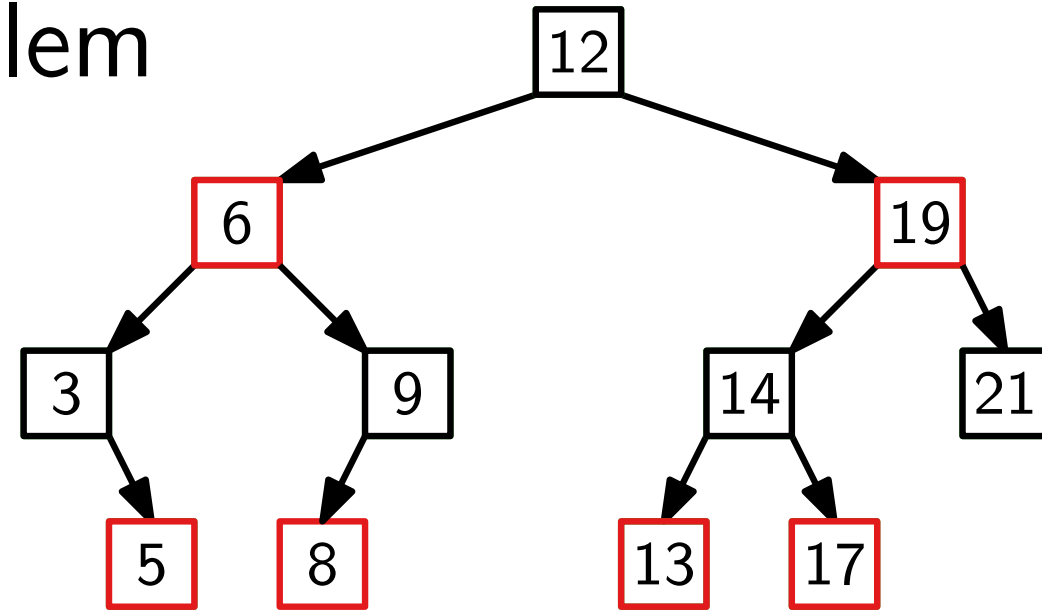
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

```
Node SELECT(int i):
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)
```

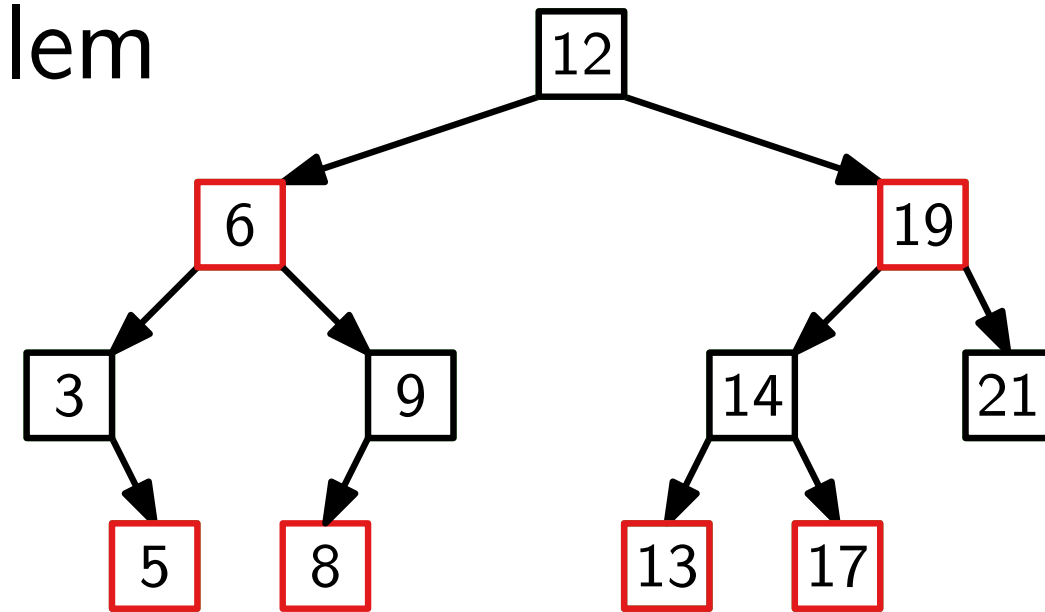
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

```

Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
  x = MINIMUM()
  while x ≠ nil and i > 1 do
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
    i = i - 1
  return x
  
```

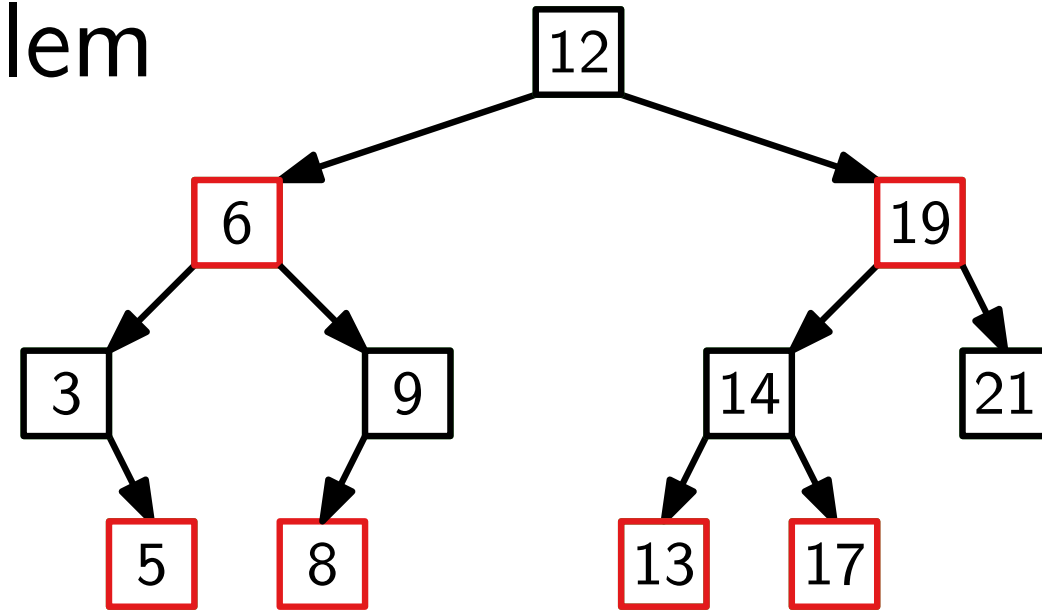
```

int RANK(Node x):
  i = 0
  while x ≠ nil do
    x = PREDECESSOR(x)
    i = i + 1
  return i
  
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

```

Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
  x = MINIMUM()
  while x ≠ nil and i > 1 do
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
    i = i - 1
  return x
  
```

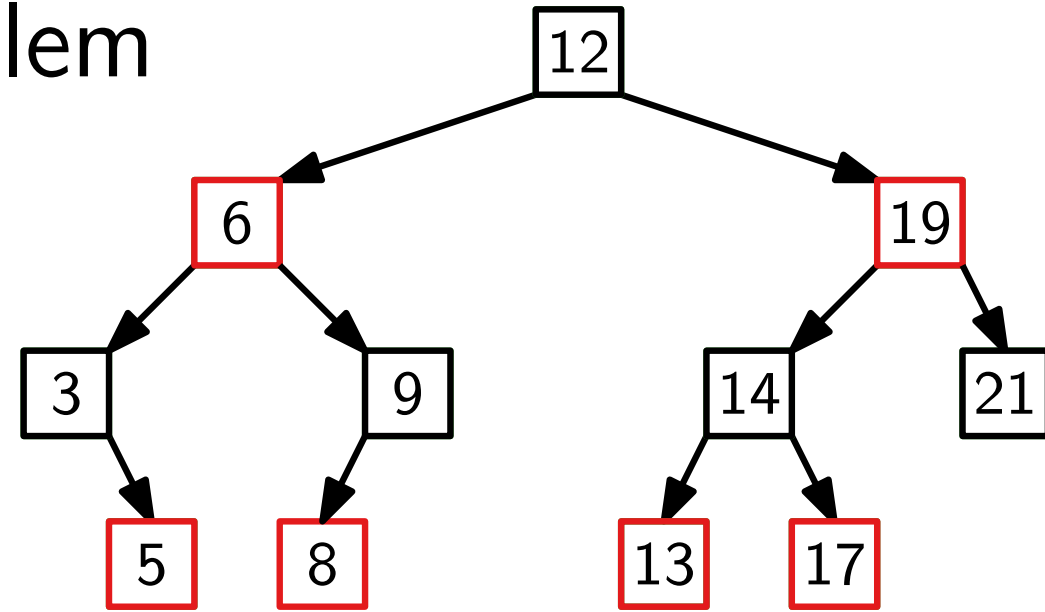
```

int RANK(Node x):
  i = 0
  while x ≠ nil do
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
    i = i + 1
  return i
  
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

```

Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
  x = MINIMUM()
  while x ≠ nil and i > 1 do
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
    i = i - 1
  return x
  
```

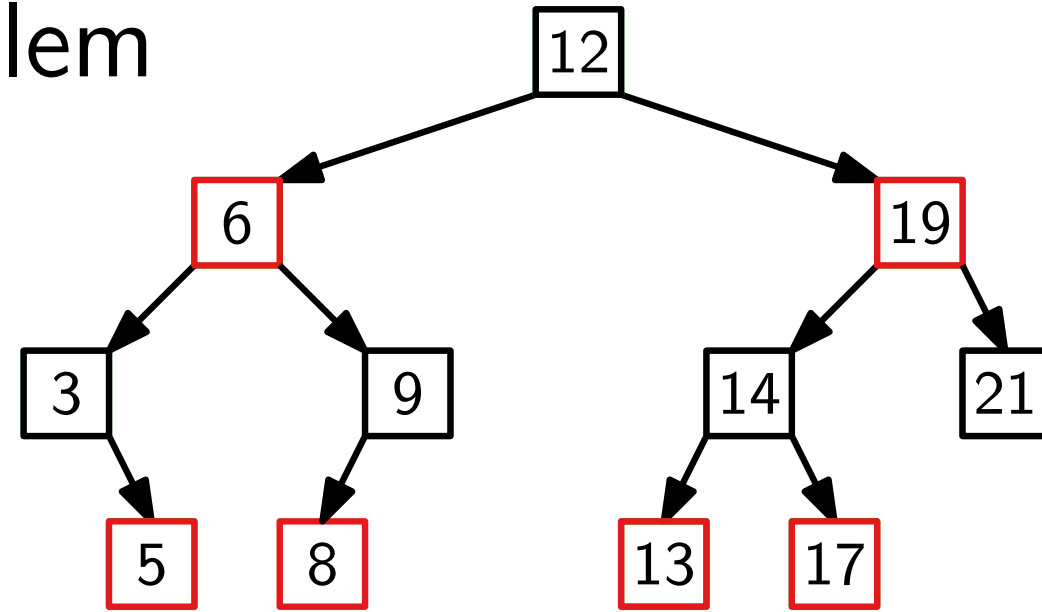
```

int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
  i = 0
  while x ≠ nil do
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
    i = i + 1
  return i
  
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

```

Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
  x = MINIMUM()
  while x ≠ nil and i > 1 do
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
    i = i - 1
  return x
  
```

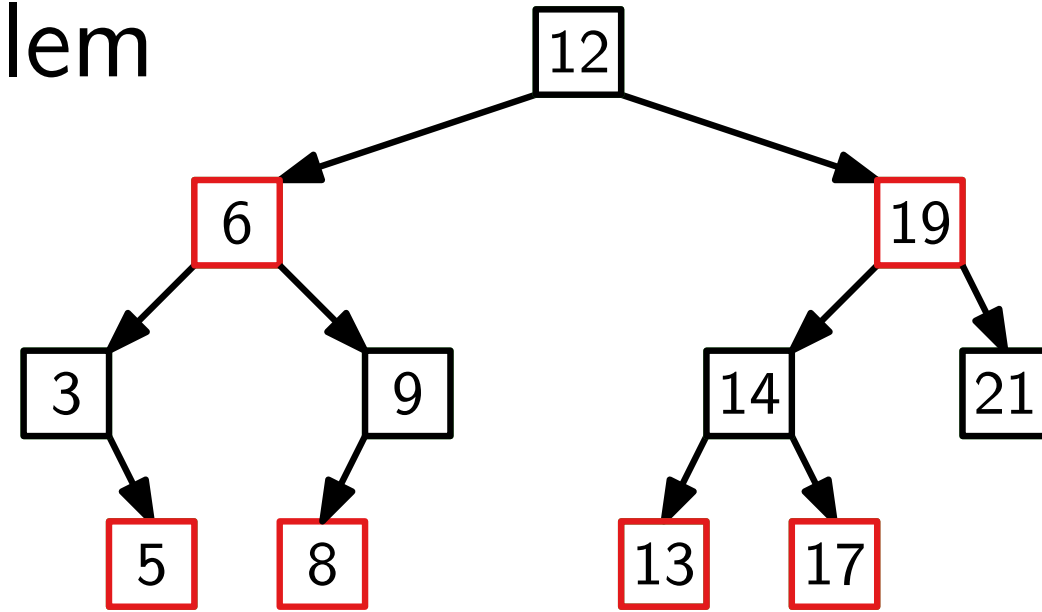
```

int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
  i = 0
  while x ≠ nil do
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
    i = i + 1
  return i
  
```


Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
x = MINIMUM()
```

```
while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
i = 0
```

```
while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

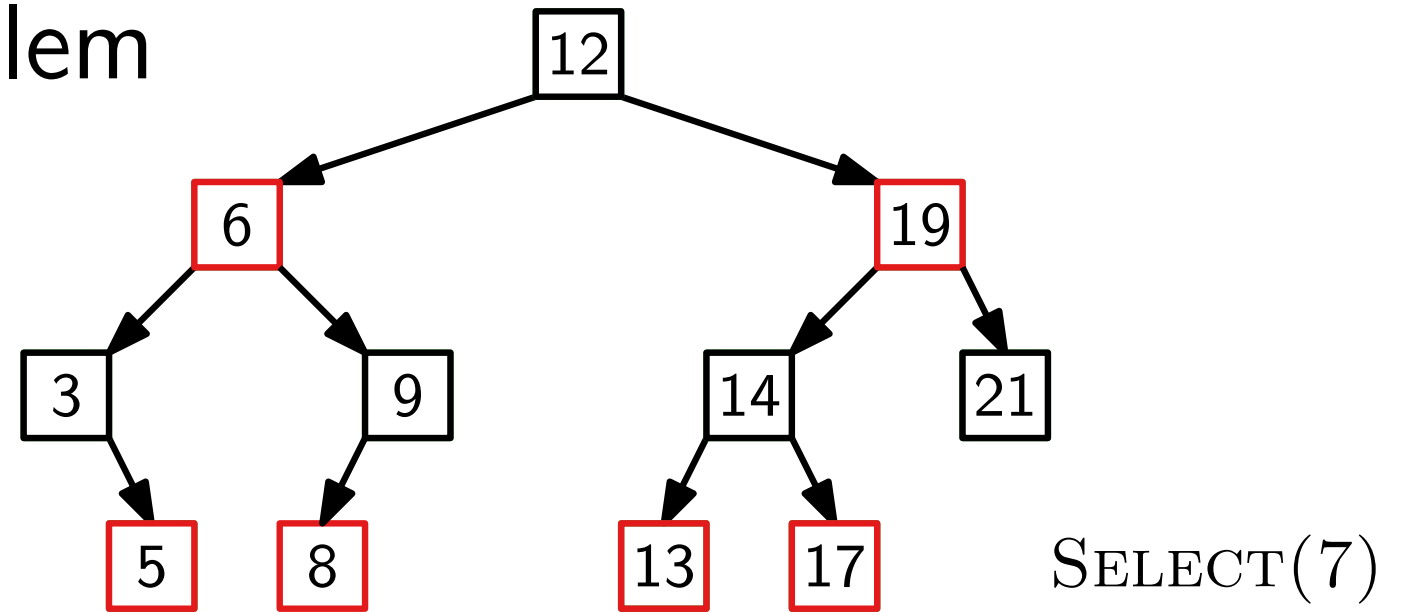
```
    i = i + 1
```

```
return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

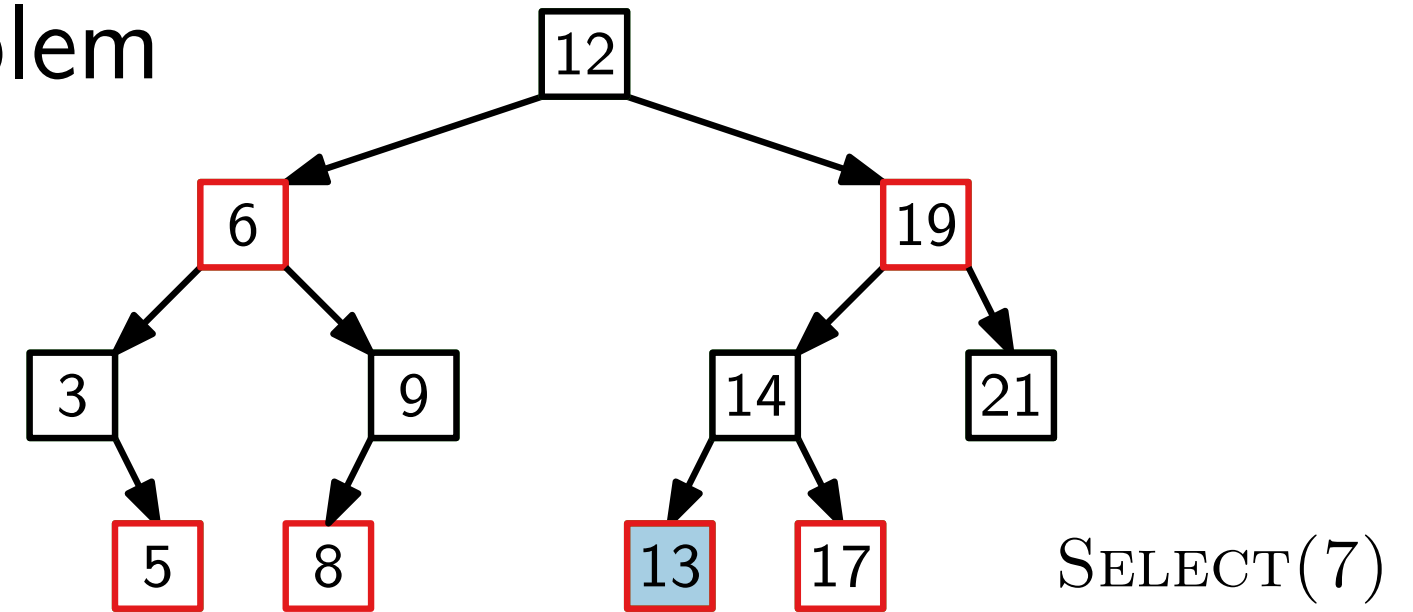
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

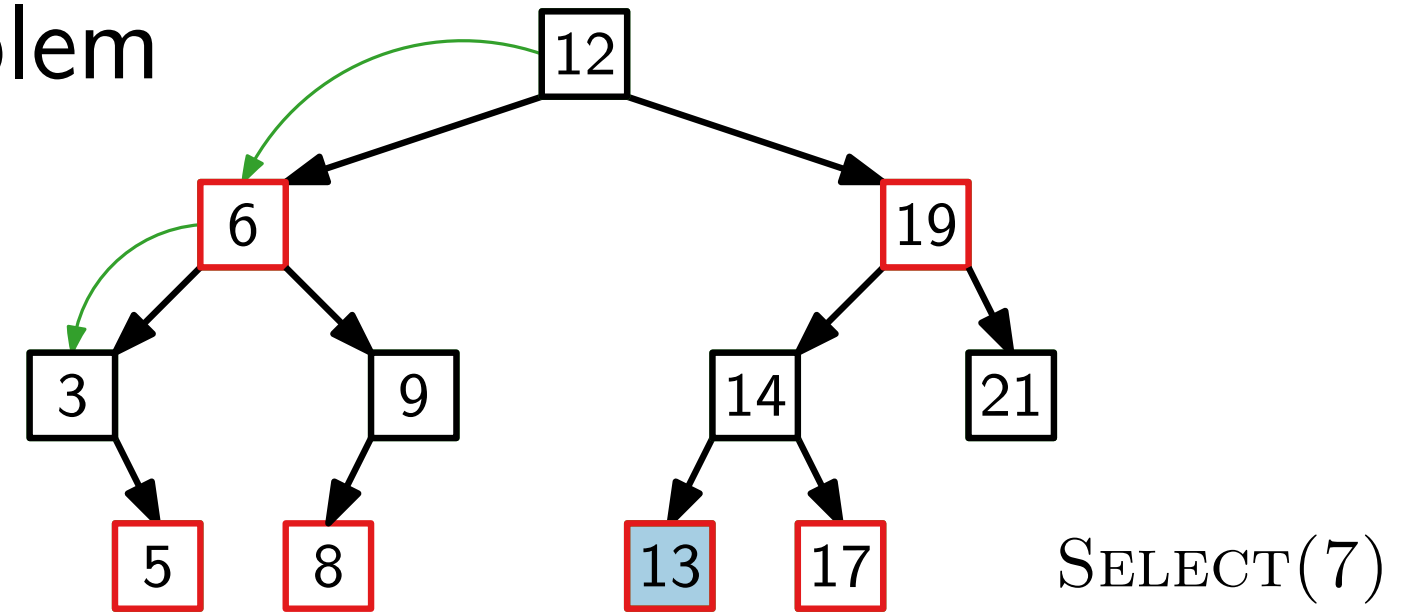
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

Node SELECT(int i): $\mathcal{O}(i \cdot h)$

$x = \text{MINIMUM}()$

while $x \neq \text{nil}$ **and** $i > 1$ **do**

$x = \text{SUCCESSOR}(x)$ $\mathcal{O}(h)$

$i = i - 1$

return x

int RANK(Node x): $\mathcal{O}(\text{rank} \cdot h)$

$i = 0$

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

$x = \text{PREDECESSOR}(x)$ $\mathcal{O}(h)$

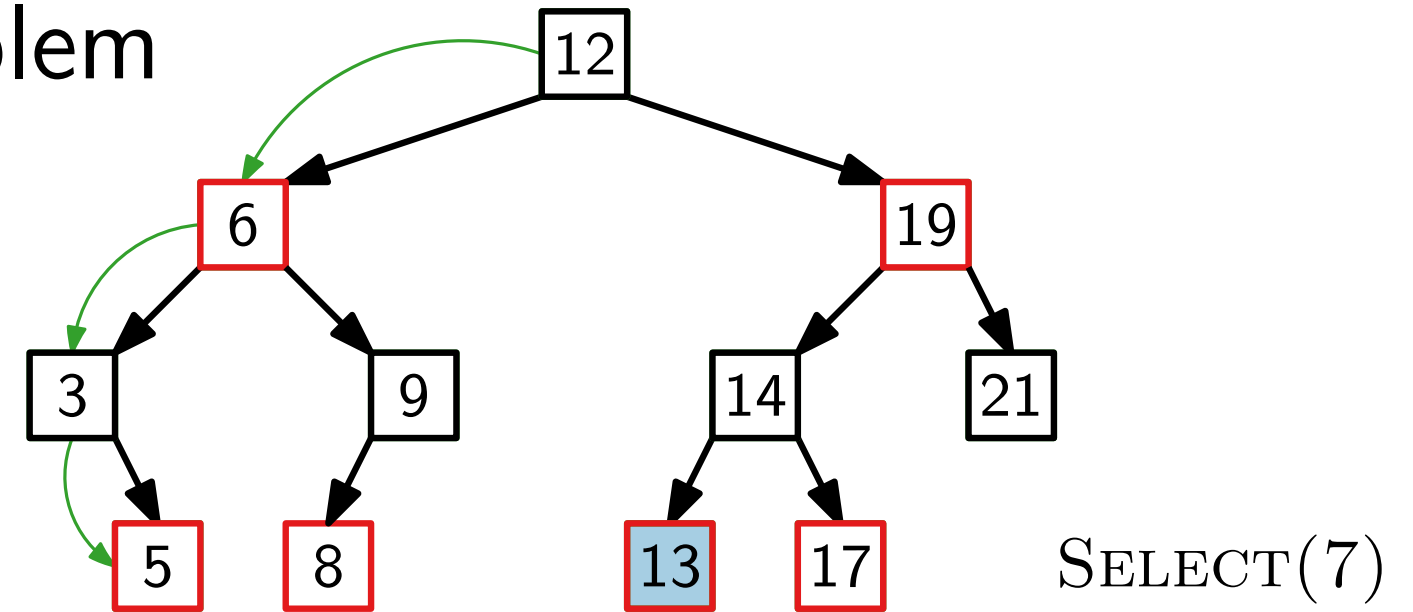
$i = i + 1$

return i

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

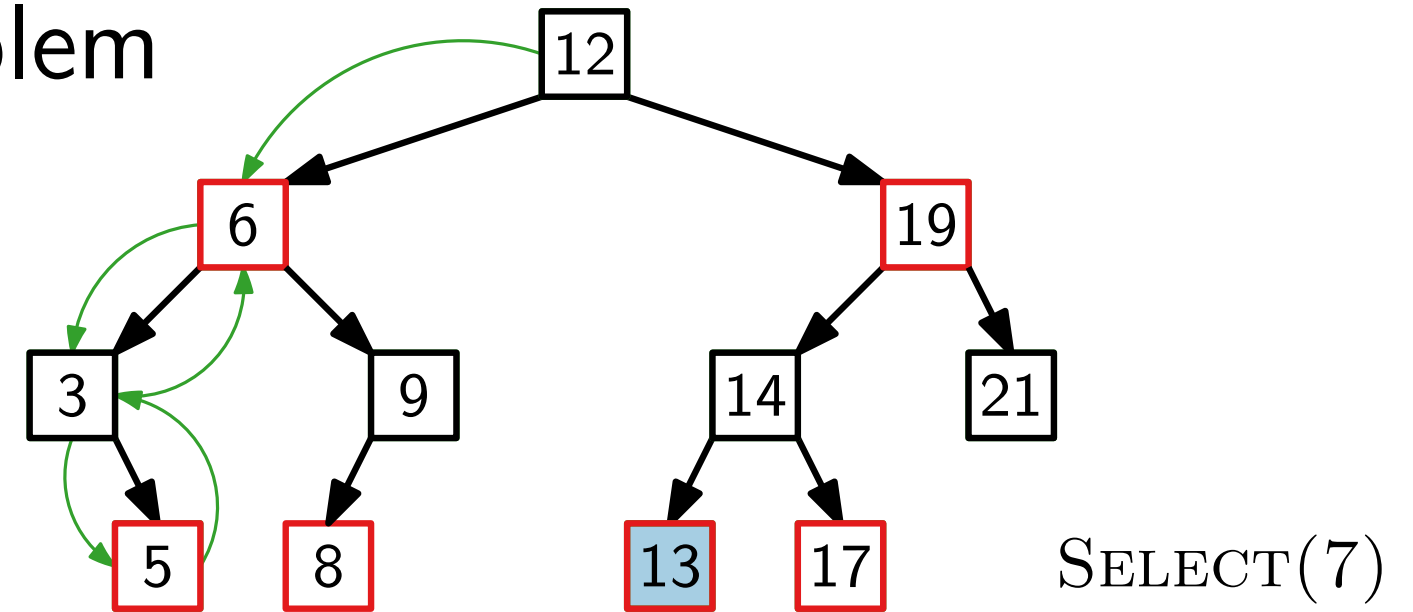
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

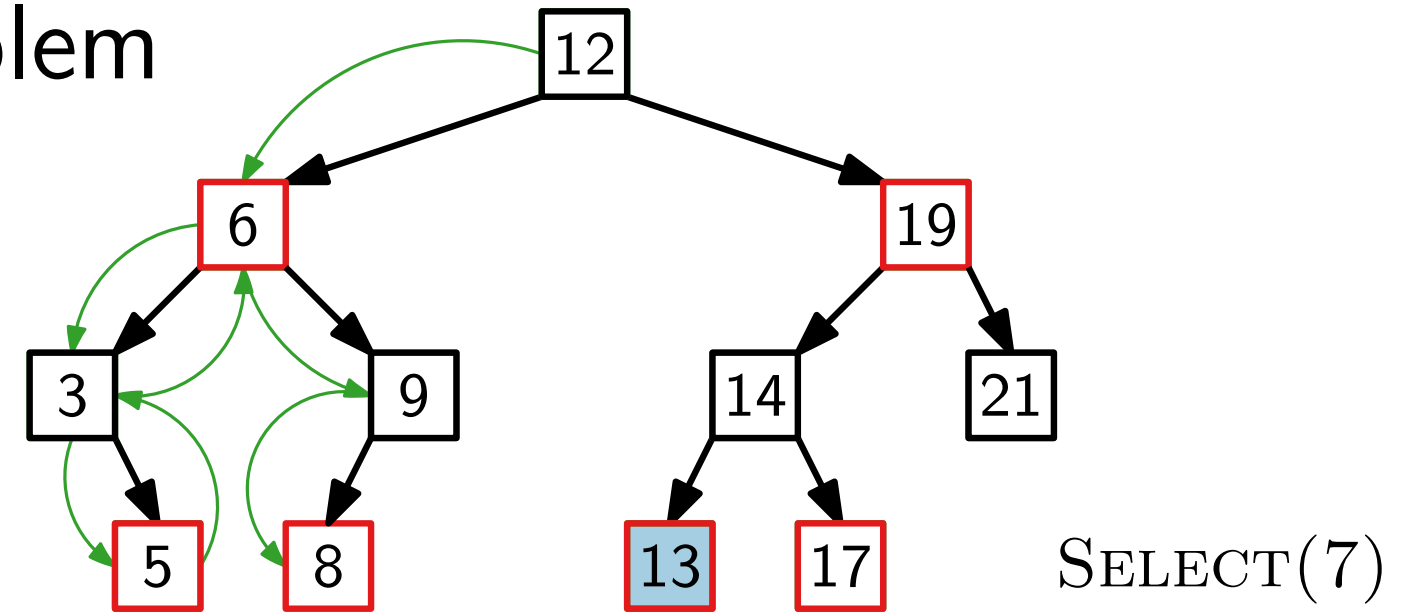
```
    i = i + 1
```

```
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
x = MINIMUM()
```

```
while x  $\neq$  nil and i > 1 do
```

```
└ x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
└ i = i - 1
```

```
return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
i = 0
```

```
while x  $\neq$  nil do
```

```
└ x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

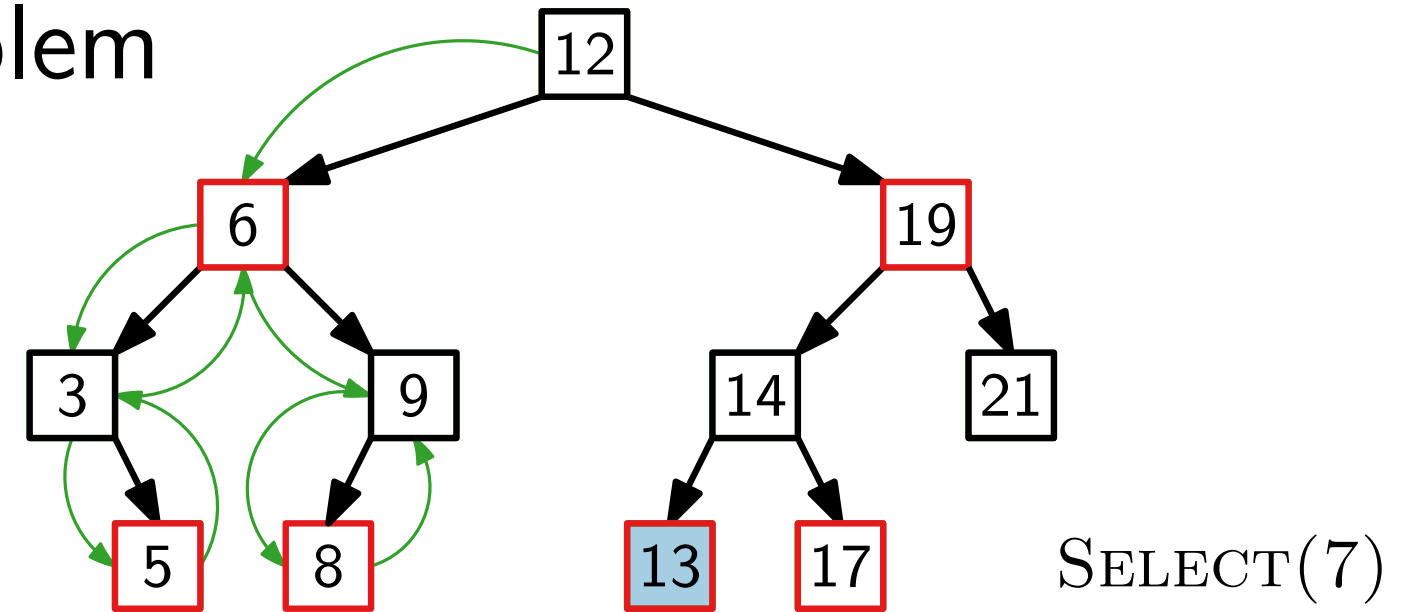
```
└ i = i + 1
```

```
return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
  x = MINIMUM()
```

```
  while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
  return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
  i = 0
```

```
  while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

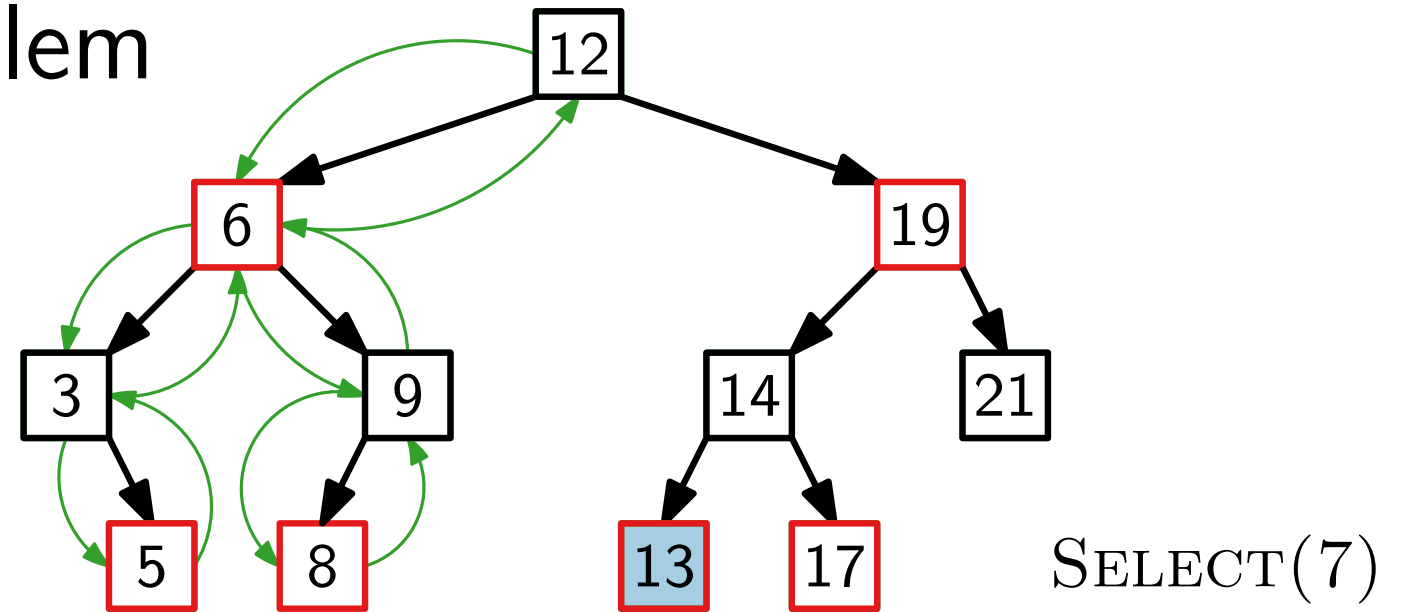
```
    i = i + 1
```

```
  return i
```


Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
x = MINIMUM()
```

```
while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
i = 0
```

```
while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

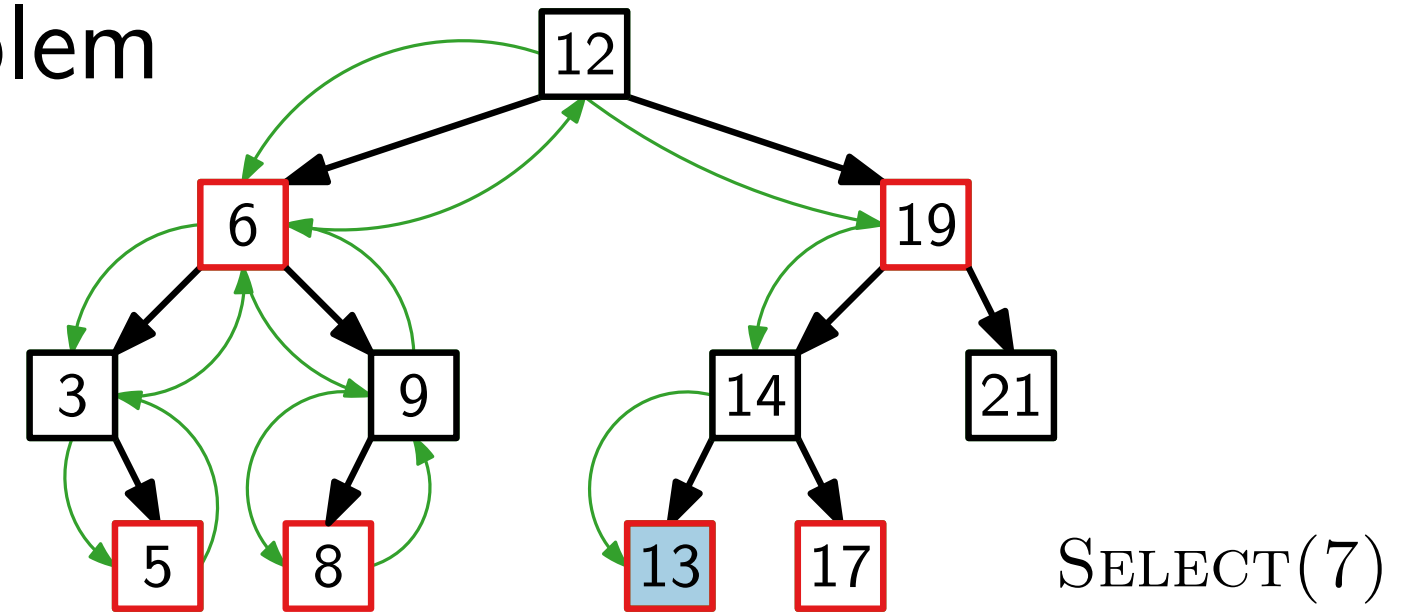
```
    i = i + 1
```

```
return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

```
Node SELECT(int i):  $\mathcal{O}(i \cdot h)$ 
```

```
x = MINIMUM()
```

```
while x ≠ nil and i > 1 do
```

```
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
```

```
    i = i - 1
```

```
return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank} \cdot h)$ 
```

```
i = 0
```

```
while x ≠ nil do
```

```
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
```

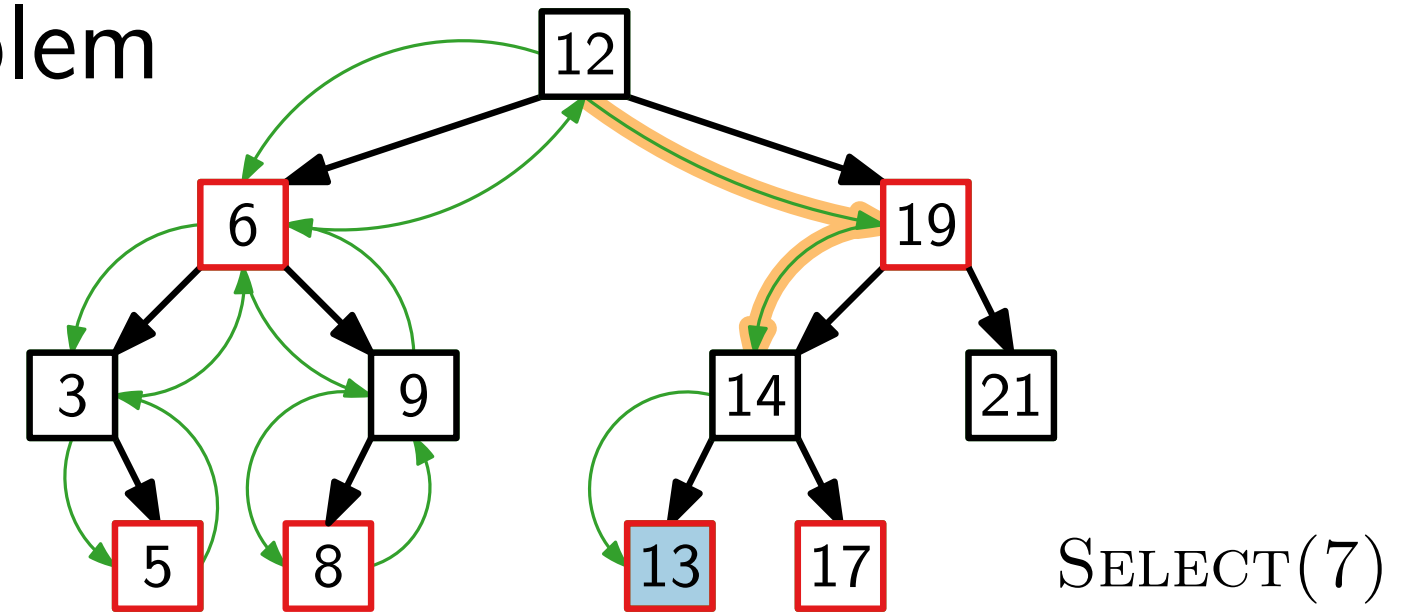
```
    i = i + 1
```

```
return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- gar keine?

3. Aufwand zur Aufrechterhaltung?

- gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Abschätzung bestmöglich?

Nein!

Node SELECT(int i): $\mathcal{O}(i \cdot h)$

$x = \text{MINIMUM}()$

while $x \neq \text{nil}$ **and** $i > 1$ **do**

$x = \text{SUCCESSOR}(x)$ $\mathcal{O}(h)$

$i = i - 1$

return x

int RANK(Node x): $\mathcal{O}(\text{rank} \cdot h)$

$i = 0$

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

$x = \text{PREDECESSOR}(x)$ $\mathcal{O}(h)$

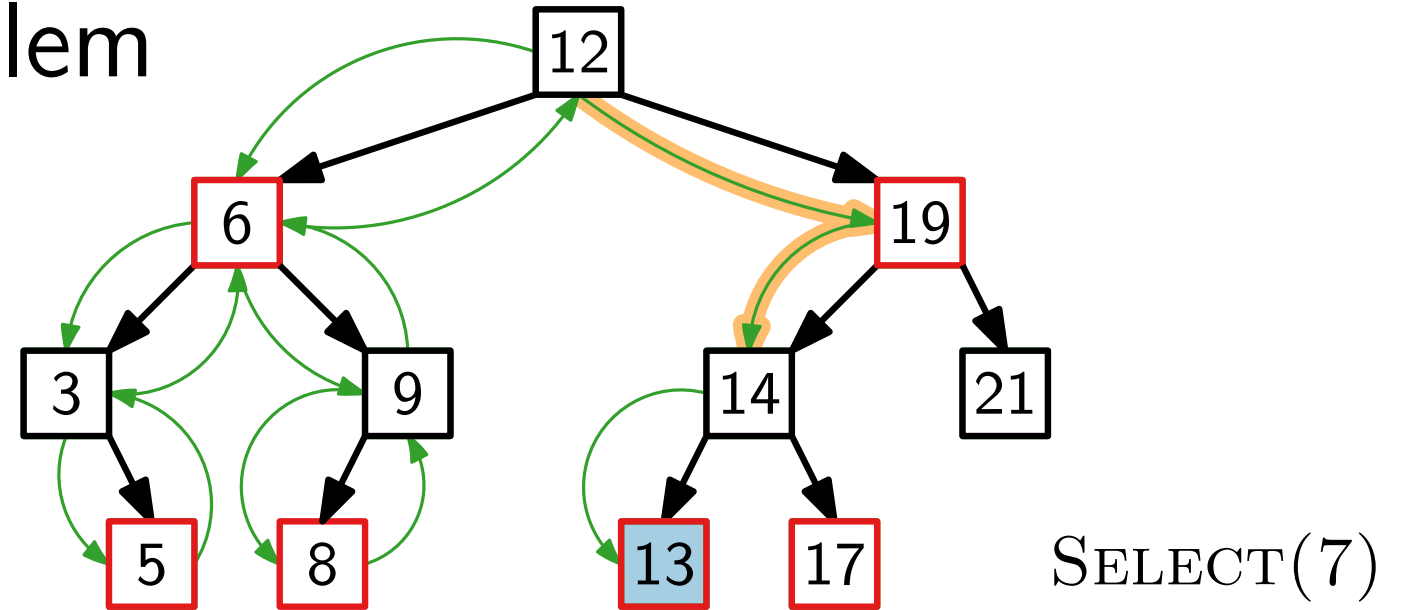
$i = i + 1$

return i

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

■ gar keine?

3. Aufwand zur Aufrechterhaltung?

■ gar keiner

4. Implementiere neue Operationen!

Laufzeit?

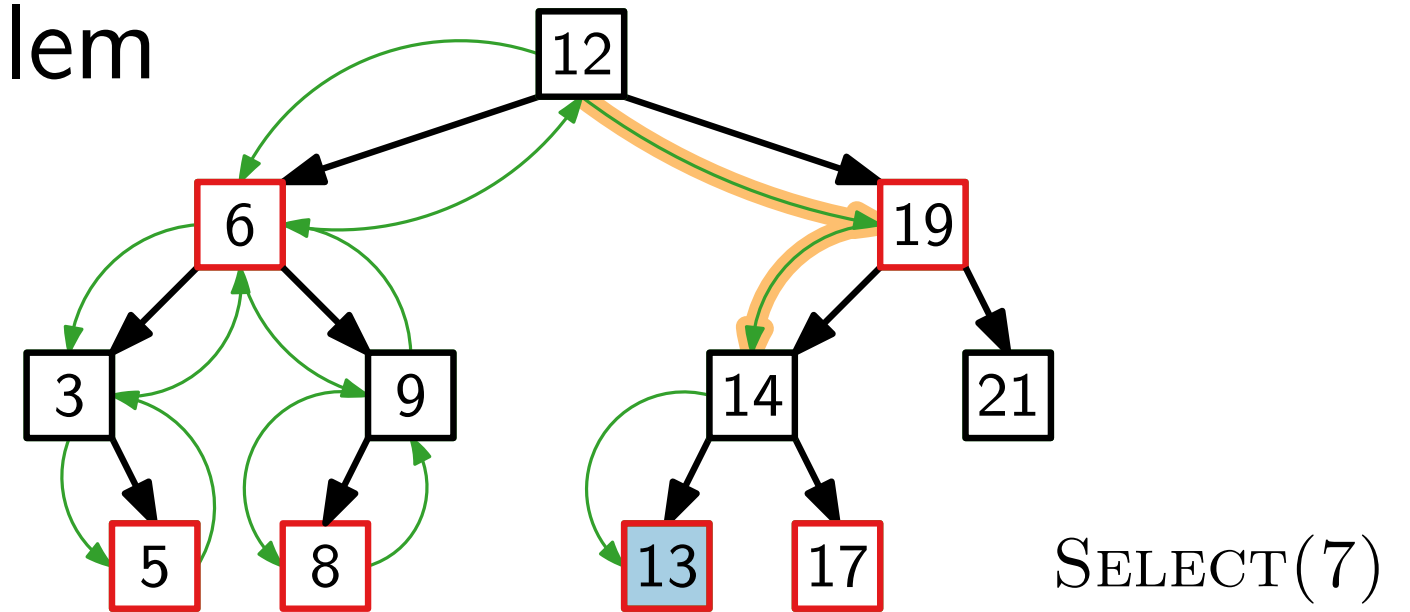
```
Node SELECT(int i):  $\mathcal{O}(i+h)$ 
  x = MINIMUM()
  while x ≠ nil and i > 1 do
    x = SUCCESSOR(x)  $\mathcal{O}(h)$ 
    i = i - 1
  return x
```

```
int RANK(Node x):  $\mathcal{O}(\text{rank}+h)$ 
  i = 0
  while x ≠ nil do
    x = PREDECESSOR(x)  $\mathcal{O}(h)$ 
    i = i + 1
  return i
```

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

■ gar keine?

3. Aufwand zur Aufrechterhaltung?

■ gar keiner

4. Implementiere neue Operationen!

Laufzeit?

Node SELECT(int i): $\mathcal{O}(i+h)$

$x = \text{MINIMUM}()$

while $x \neq \text{nil}$ **and** $i > 1$ **do**

$x = \text{SUCCESSOR}(x)$ $\mathcal{O}(h)$

$i = i - 1$

return x

int RANK(Node x): $\mathcal{O}(\text{rank}+h)$

$i = 0$

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

$x = \text{PREDECESSOR}(x)$ $\mathcal{O}(h)$

$i = i + 1$

return i

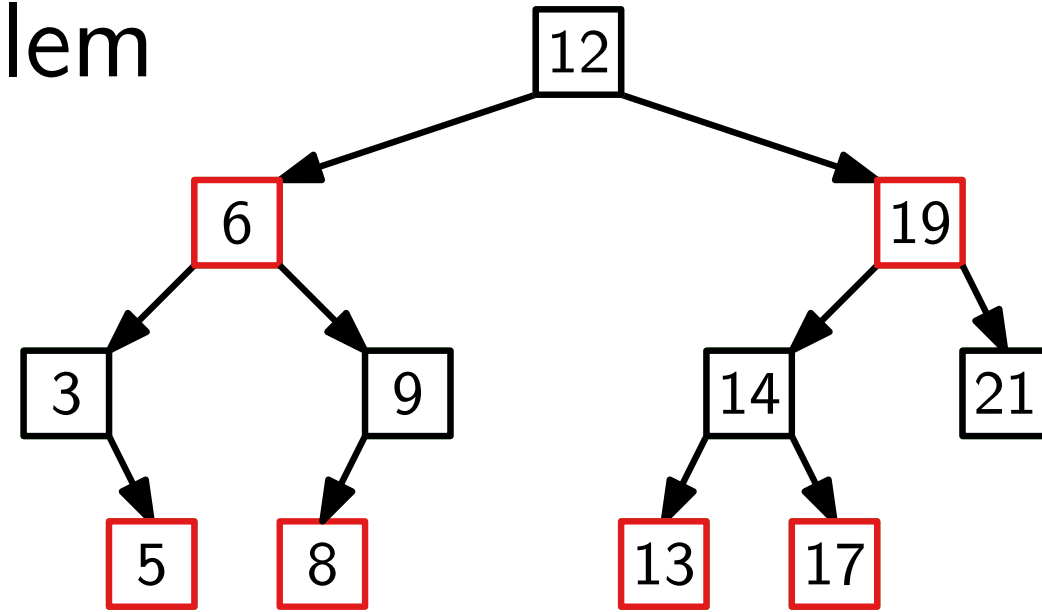
Problem:

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 $h \in \mathcal{O}(\log n)$

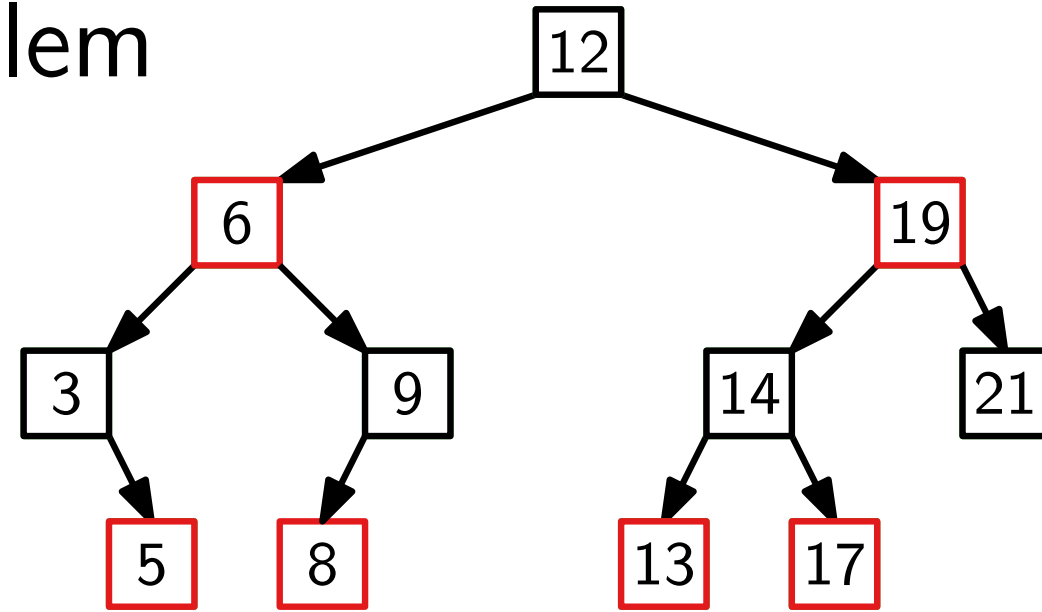


2. Welche Extrainformation aufrechterhalten?

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



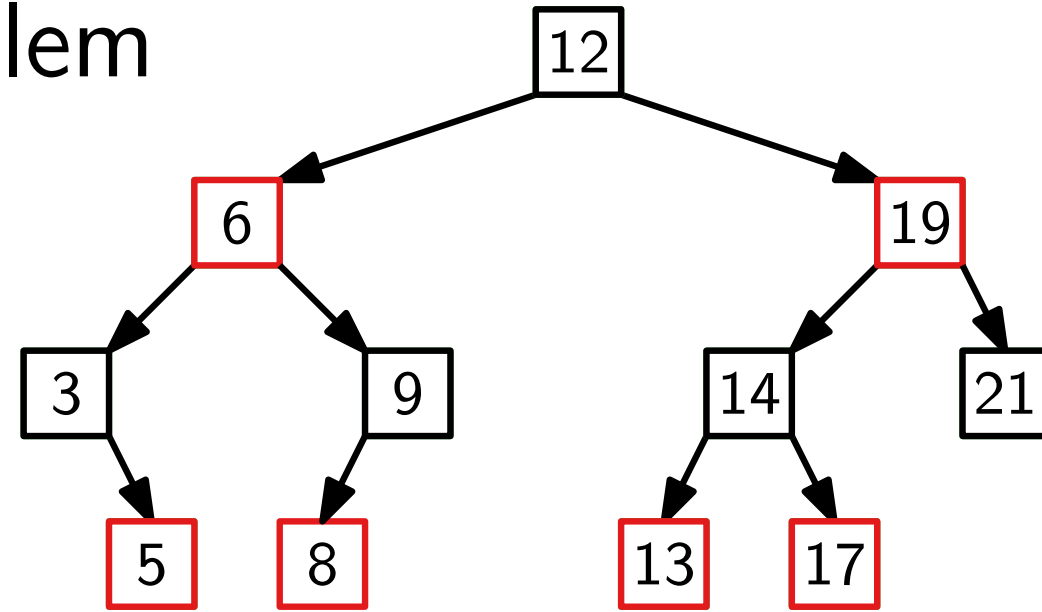
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



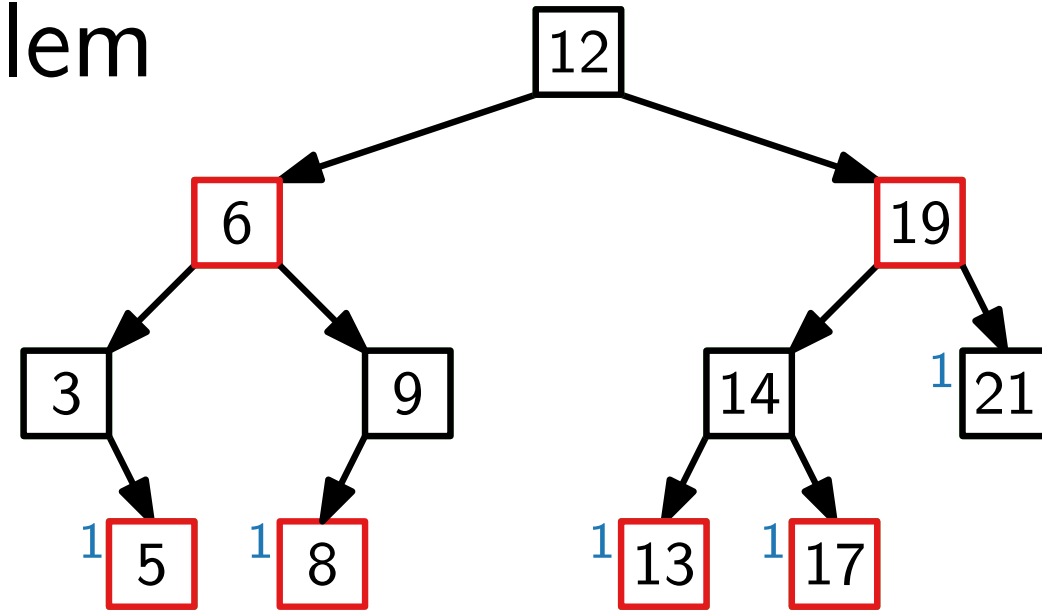
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



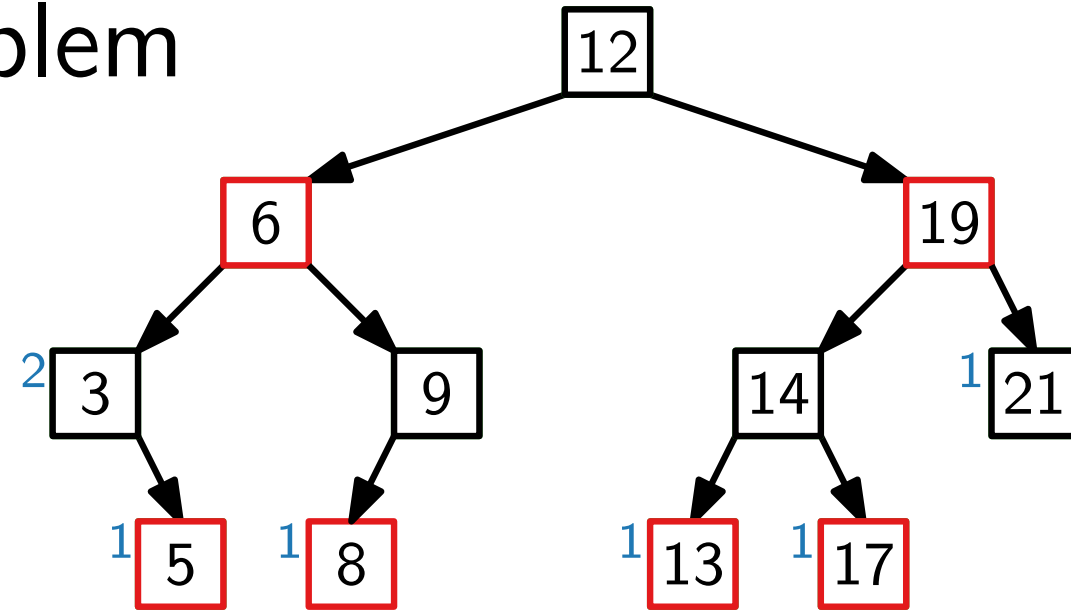
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



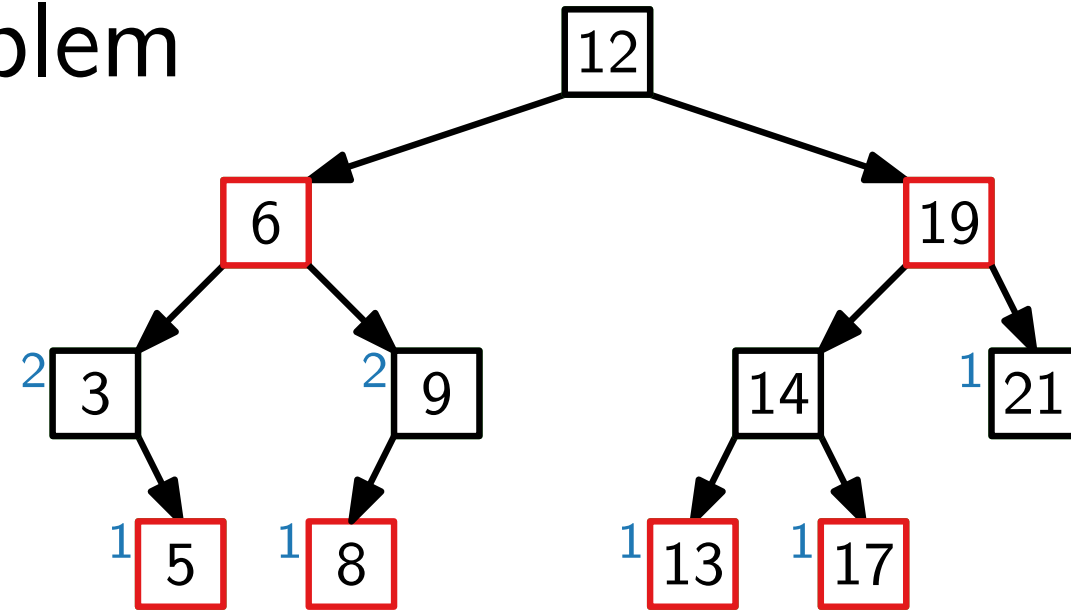
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



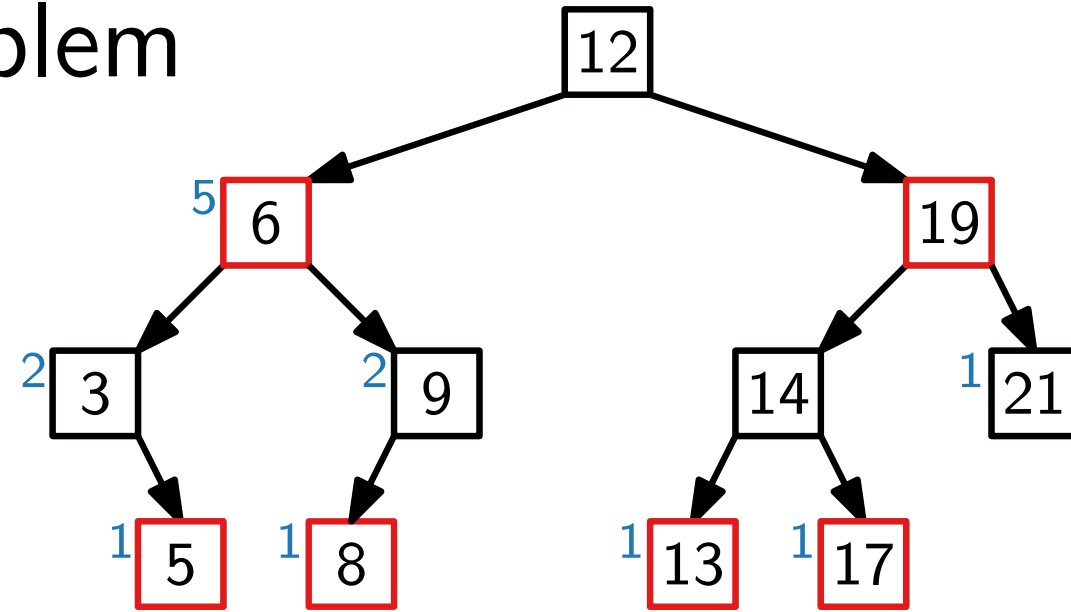
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



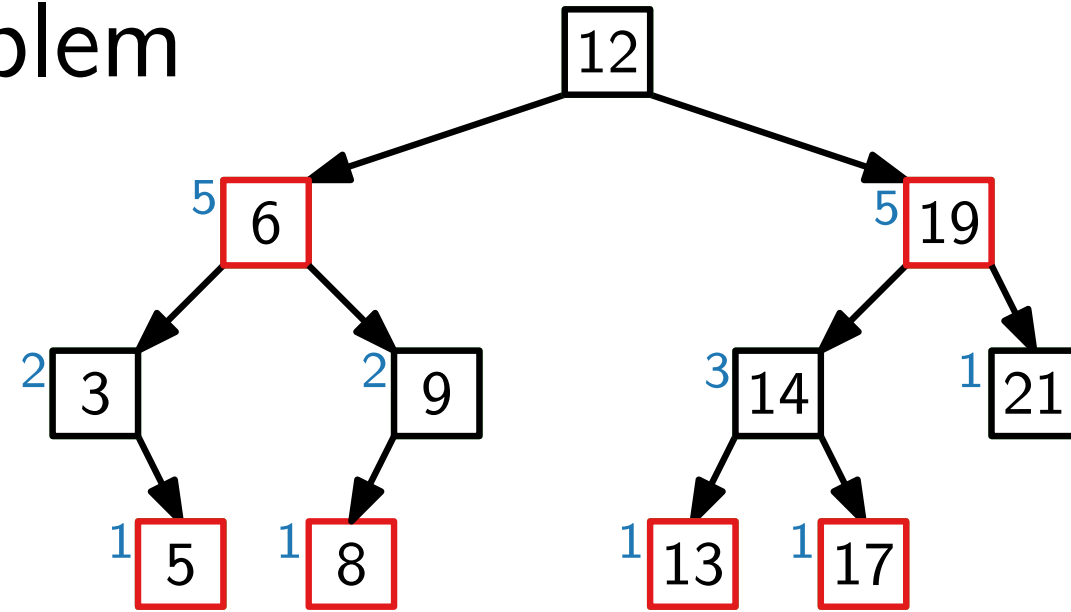
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



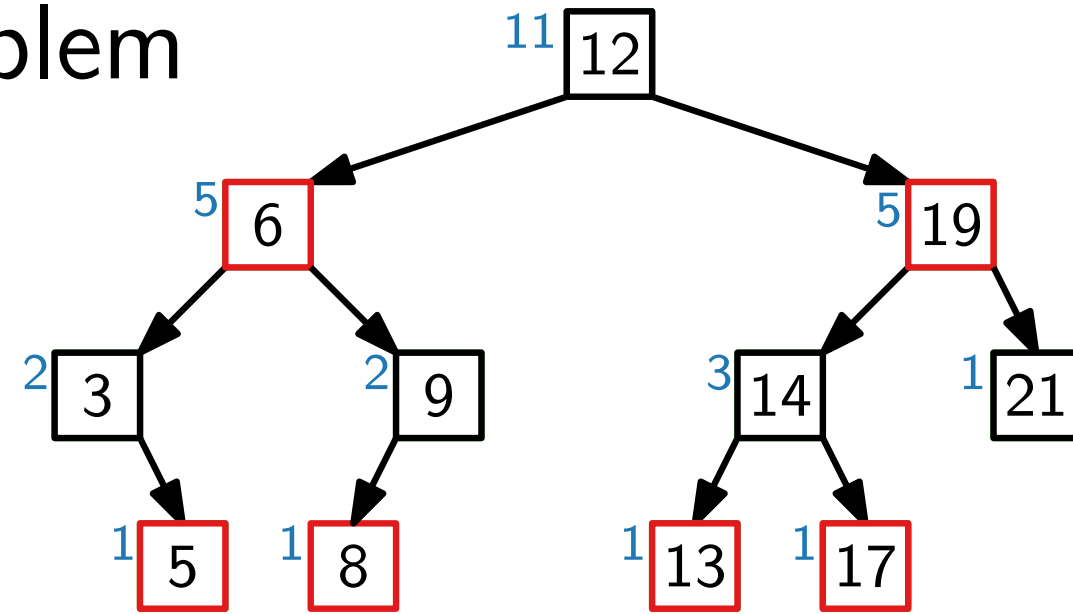
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



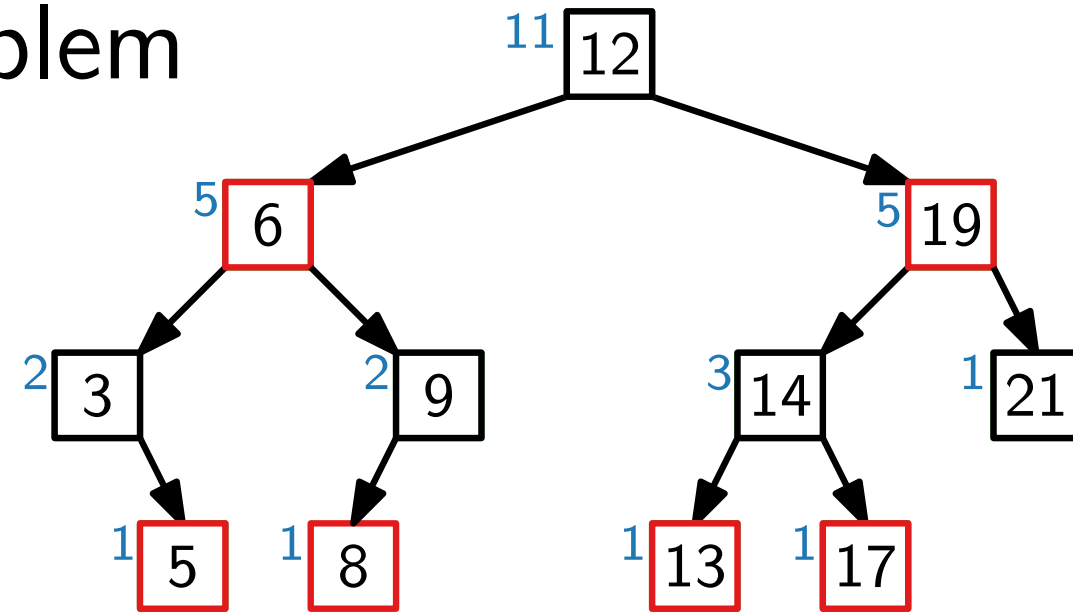
2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

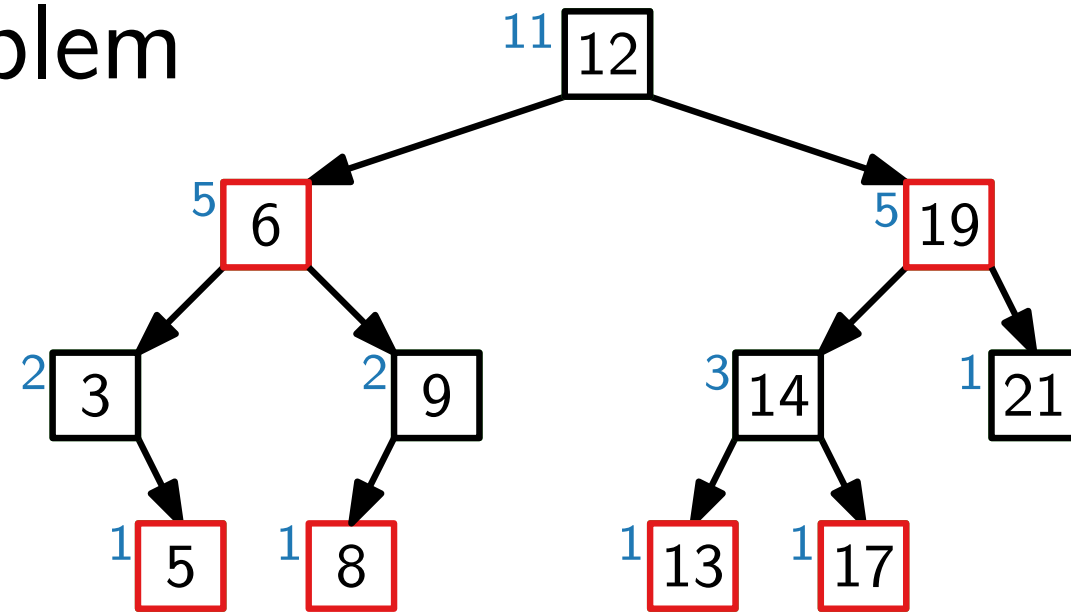
- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

3. Aufwand zur Aufrechterhaltung?

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

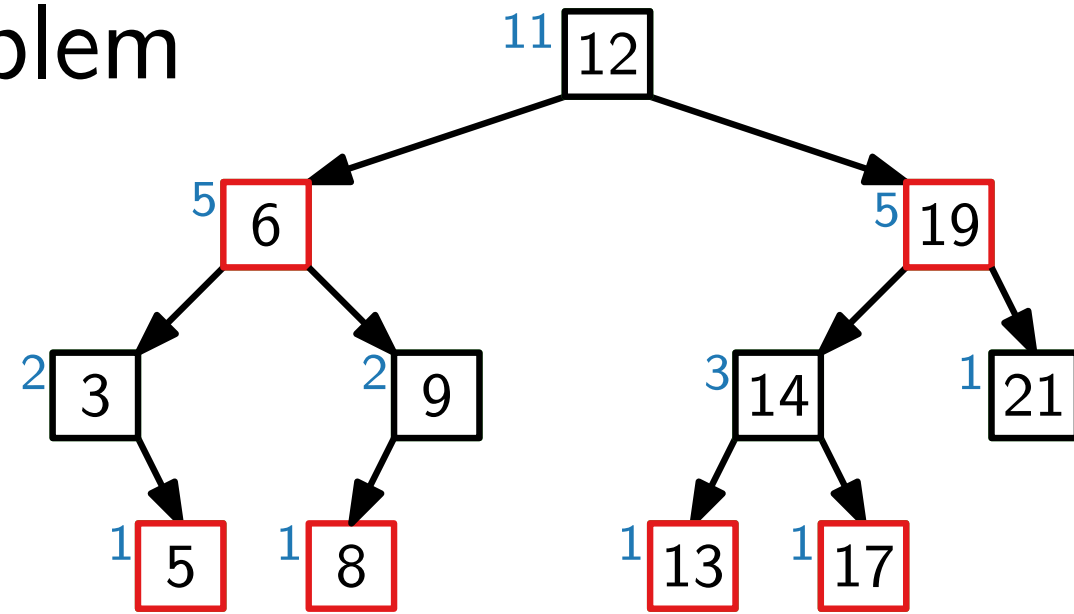
3. Aufwand zur Aufrechterhaltung? später...

für jeden Knoten v , speichere $v.size$

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

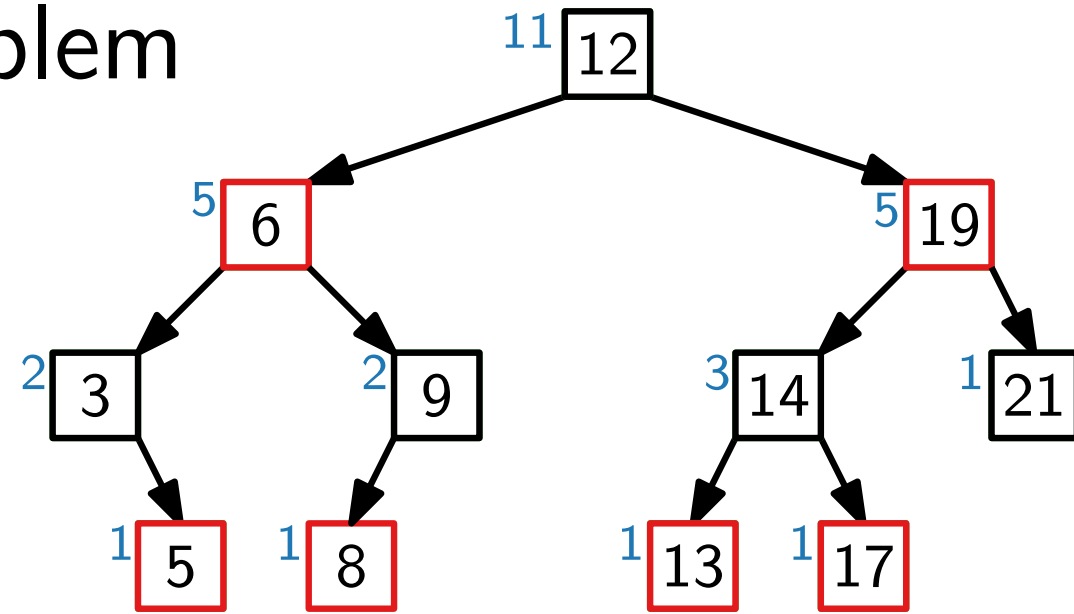
4. SELECT(Node $x = root$, int i):

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung? später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

r = x.left.size + 1
if i == r then
else
    if i < r then
    else

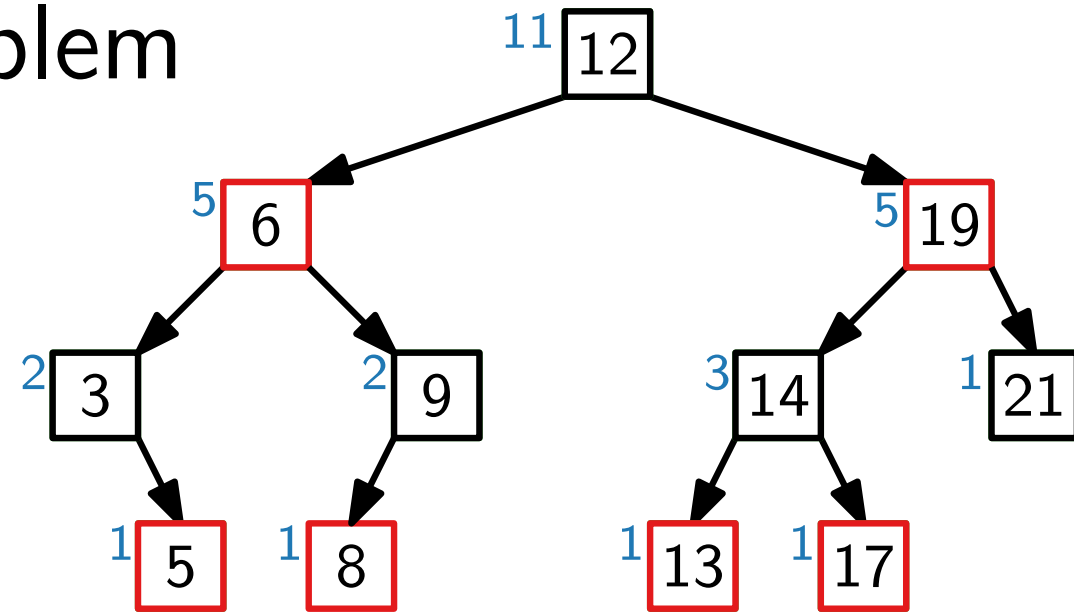
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

r = x.left.size + 1
if i == r then return x
else
    if i < r then
    else

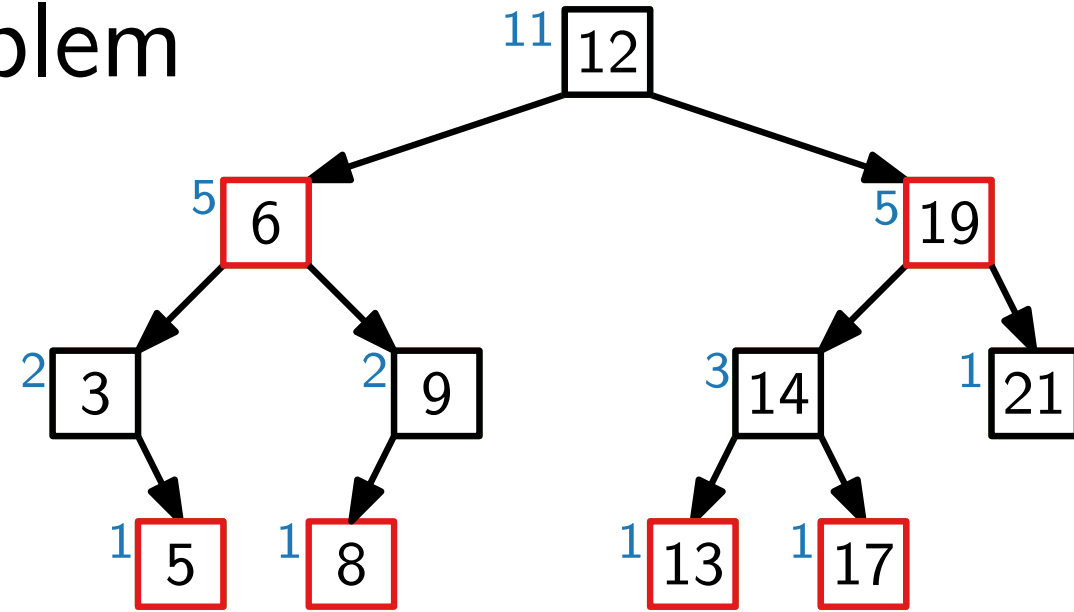
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

r = x.left.size + 1
if i == r then return x
else
    if i < r then
        return SELECT(x.left, i)
    else
        return SELECT(x.right, i - r)

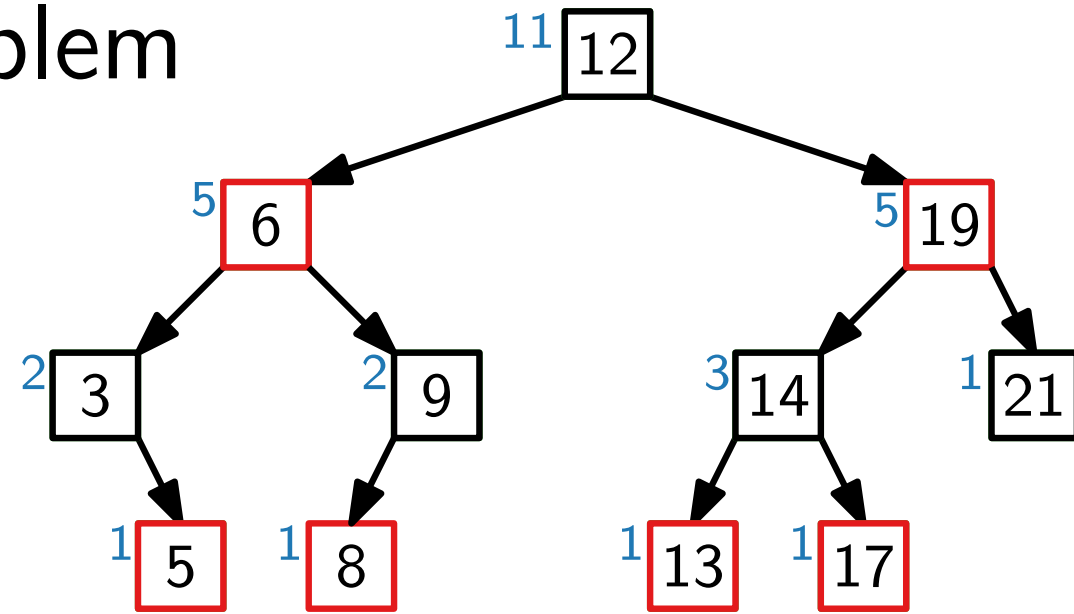
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

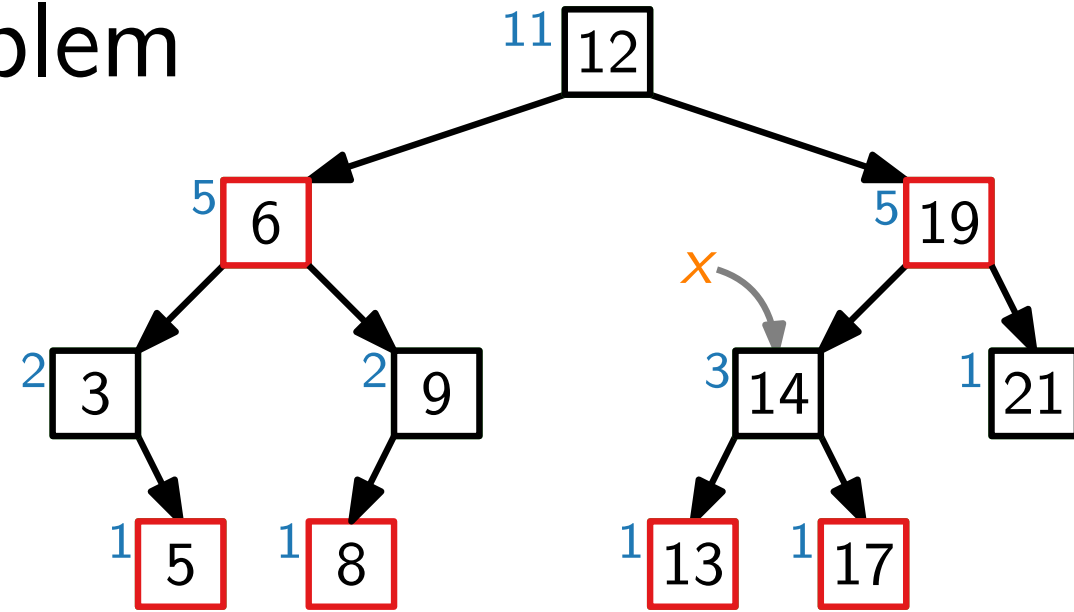
 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left, i$ )
    else
        return SELECT( $x.right, i - r$ )
  
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

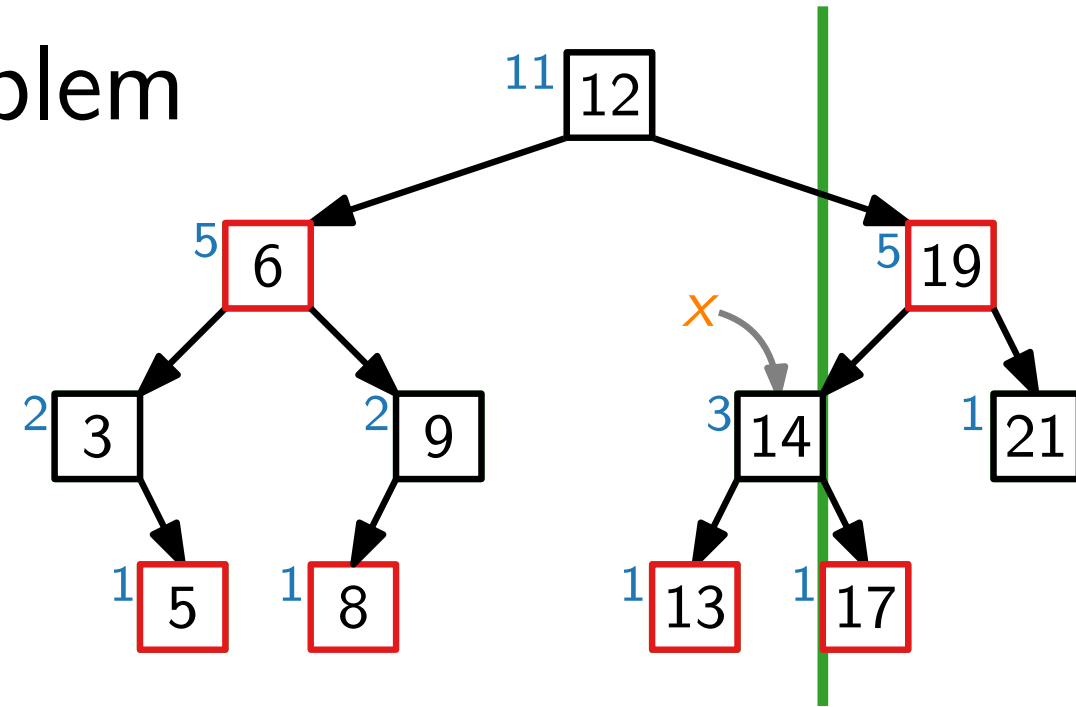
 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung? später...

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )

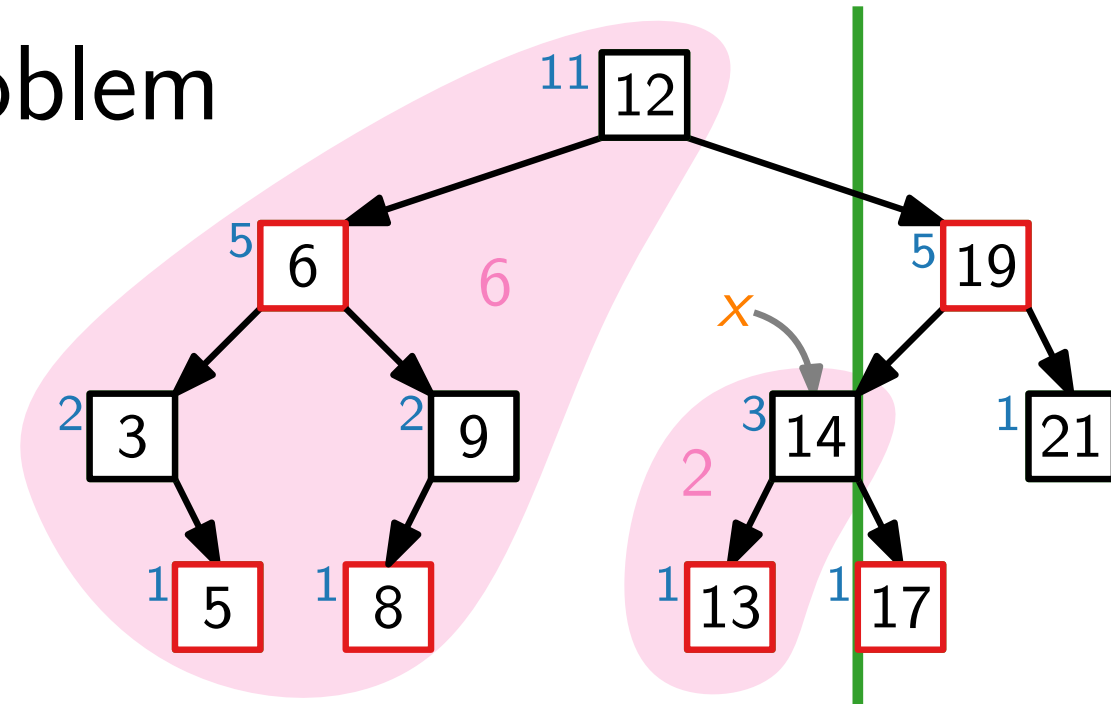
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung? später...

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        | return SELECT( $x.left$ ,  $i$ )
    else
        | return SELECT( $x.right$ ,  $i - r$ )

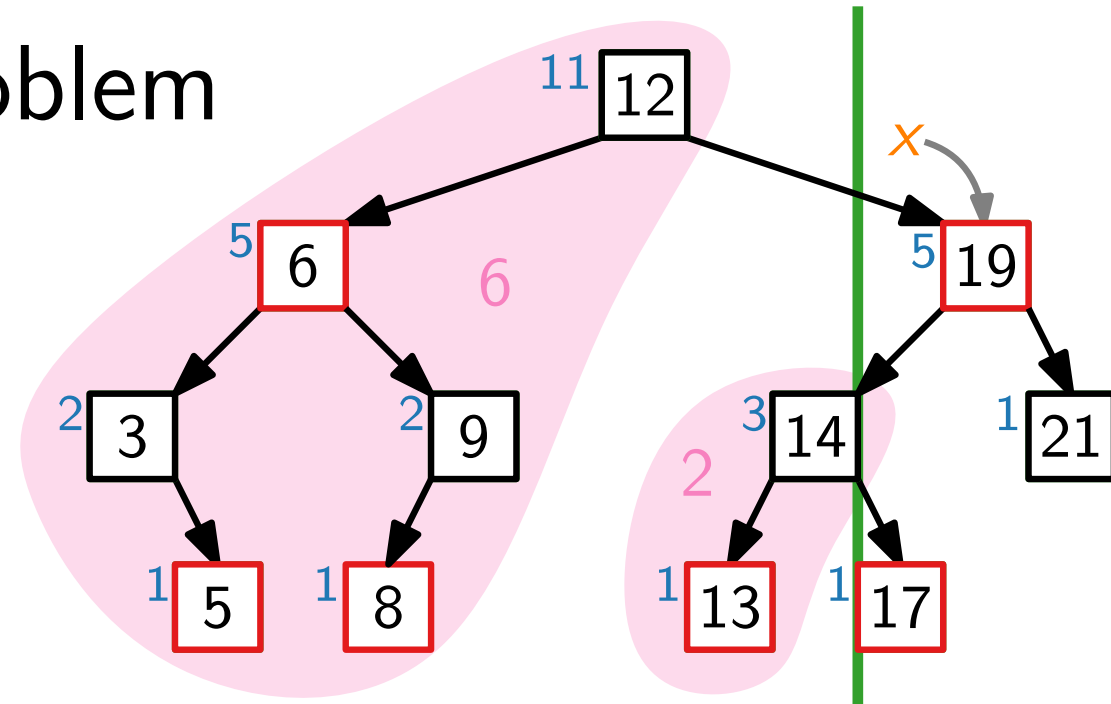
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung? später...

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )

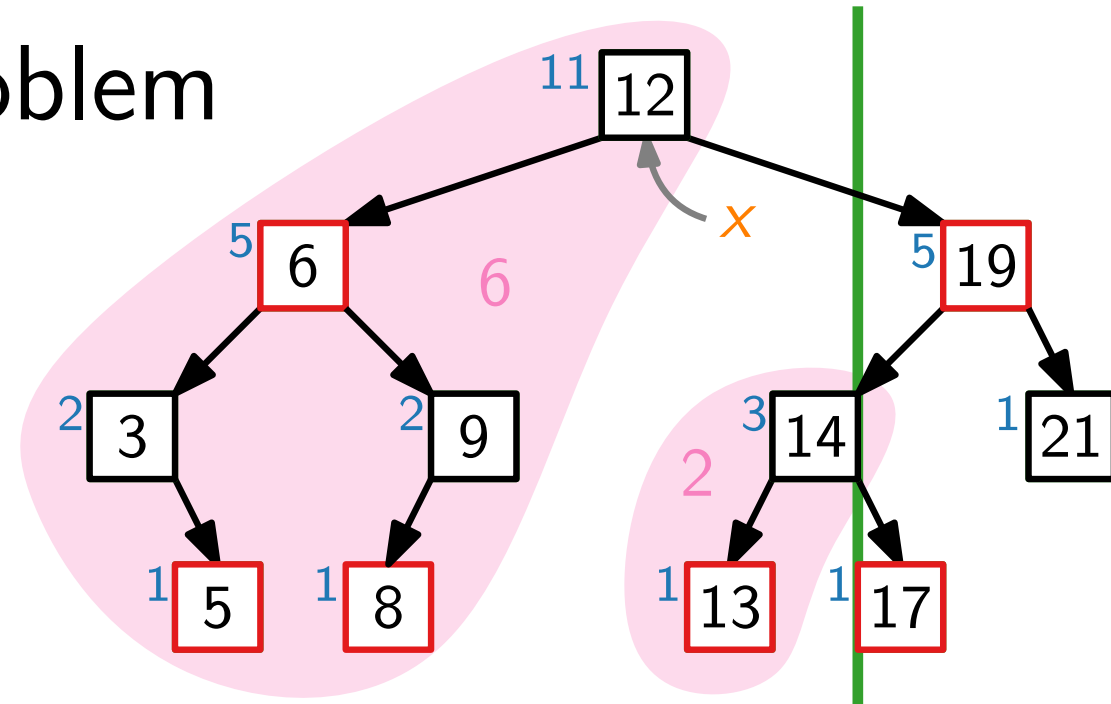
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung? später...

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )

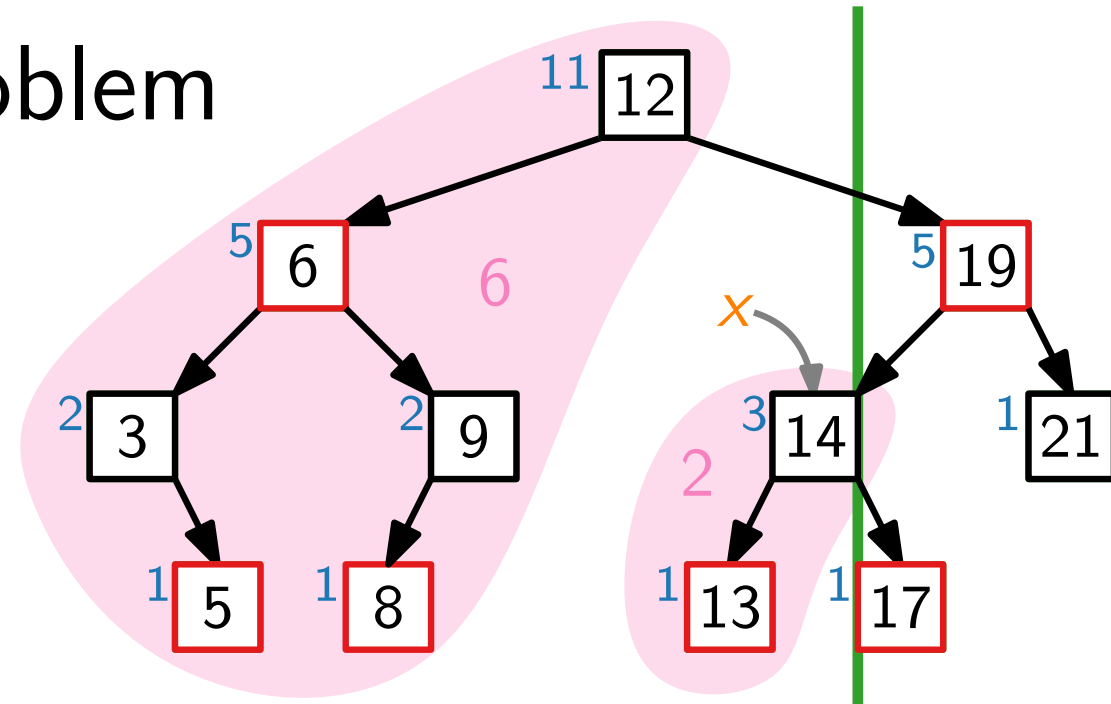
```

RANK(Node x):

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung?

später...

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

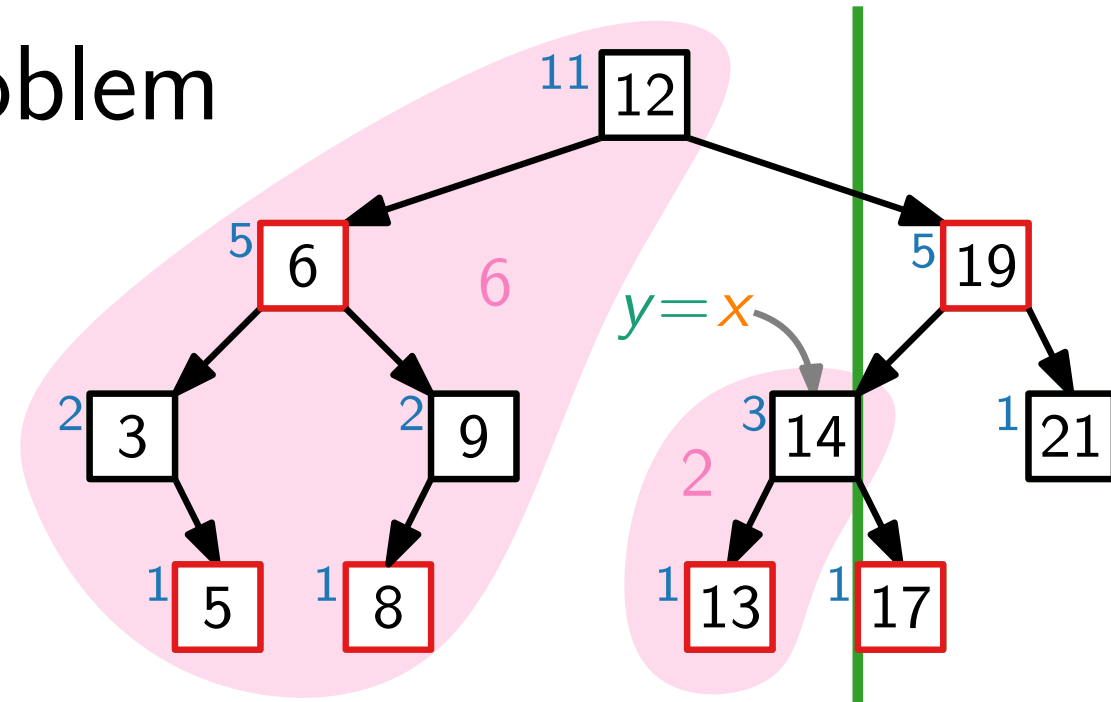
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung? später...

- Größen der Teilbäume:
für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

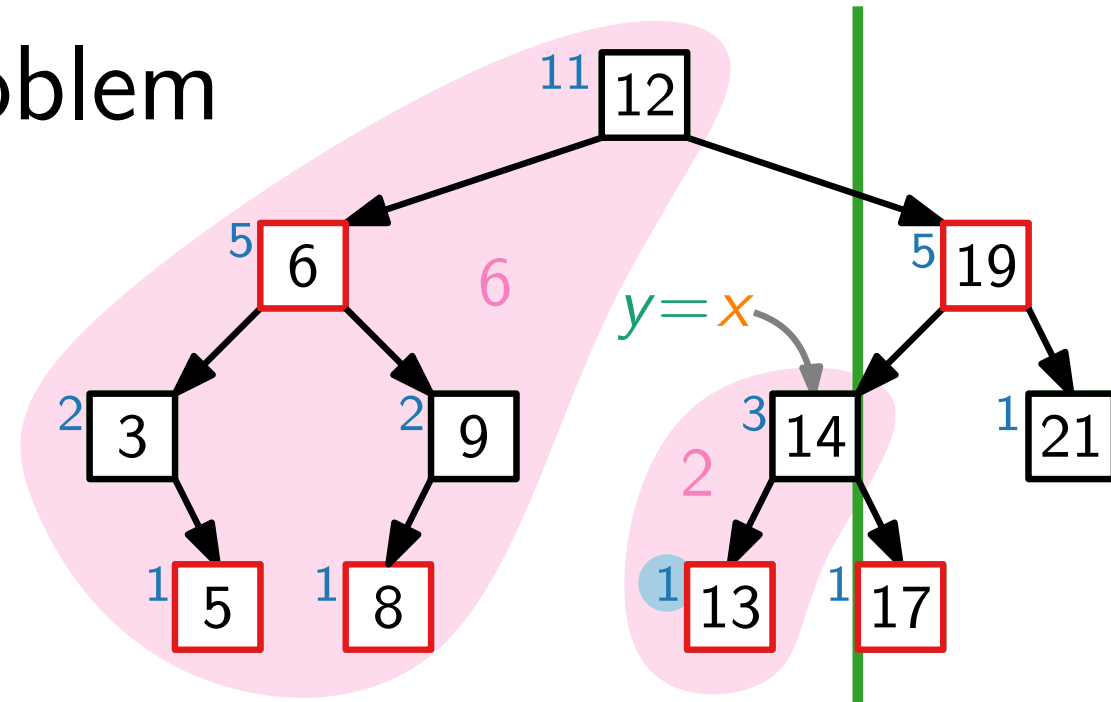
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung?

später...

- Größen der Teilbäume:

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
  if  $i < r$  then
    return SELECT( $x.left$ ,  $i$ )
  else
    return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

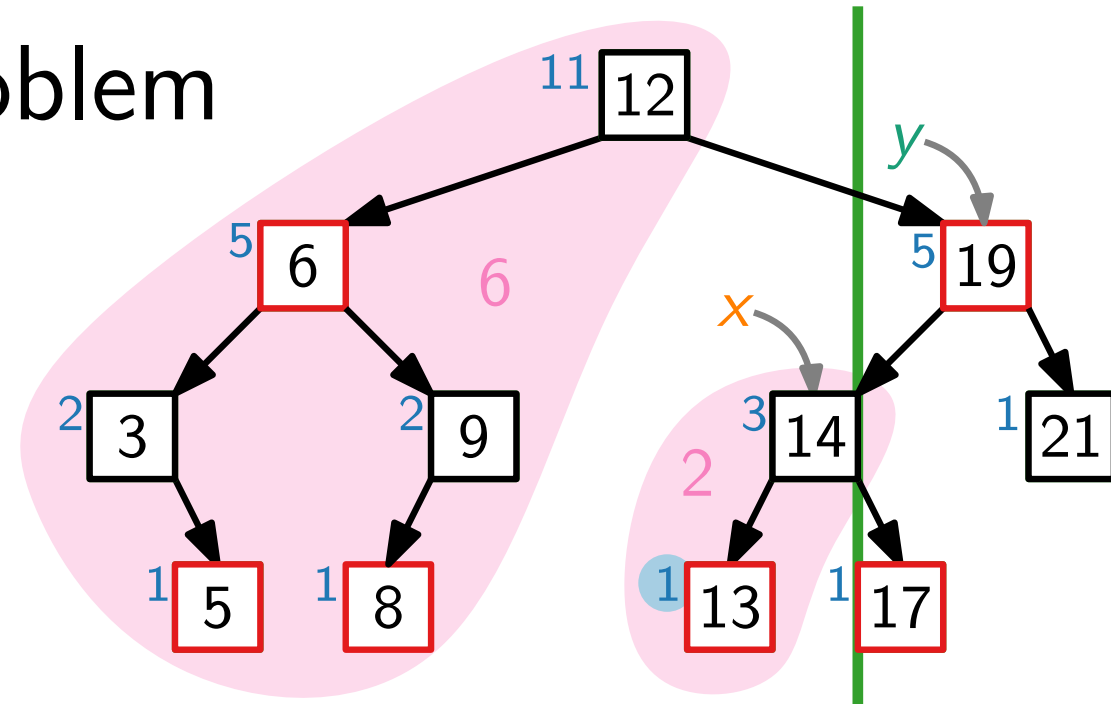
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
   $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung?

später...

- Größen der Teilbäume:

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
  if  $i < r$  then
    return SELECT( $x.left$ ,  $i$ )
  else
    return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

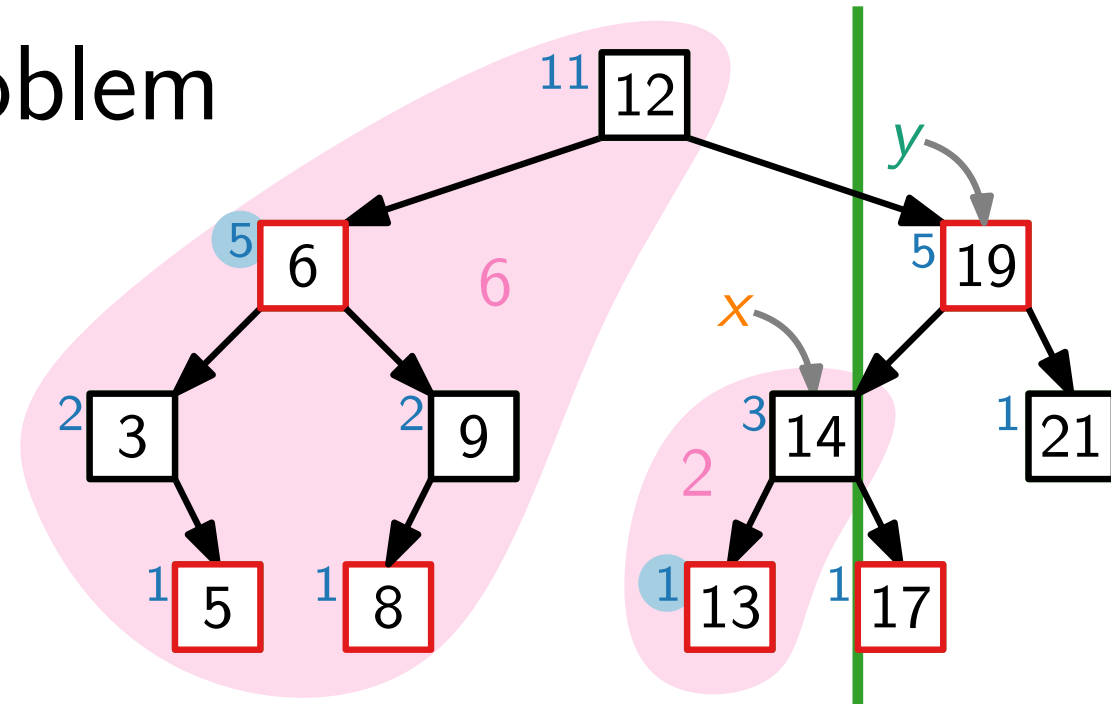
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
   $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
  if  $i < r$  then
    return SELECT( $x.left$ ,  $i$ )
  else
    return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

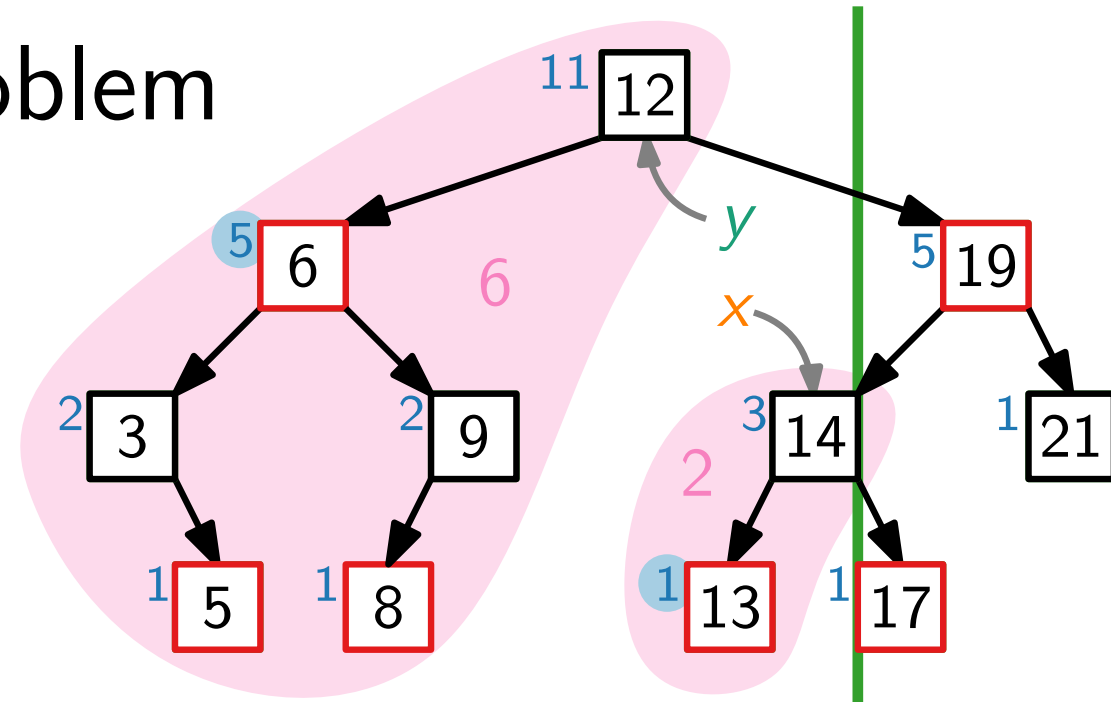
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
   $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

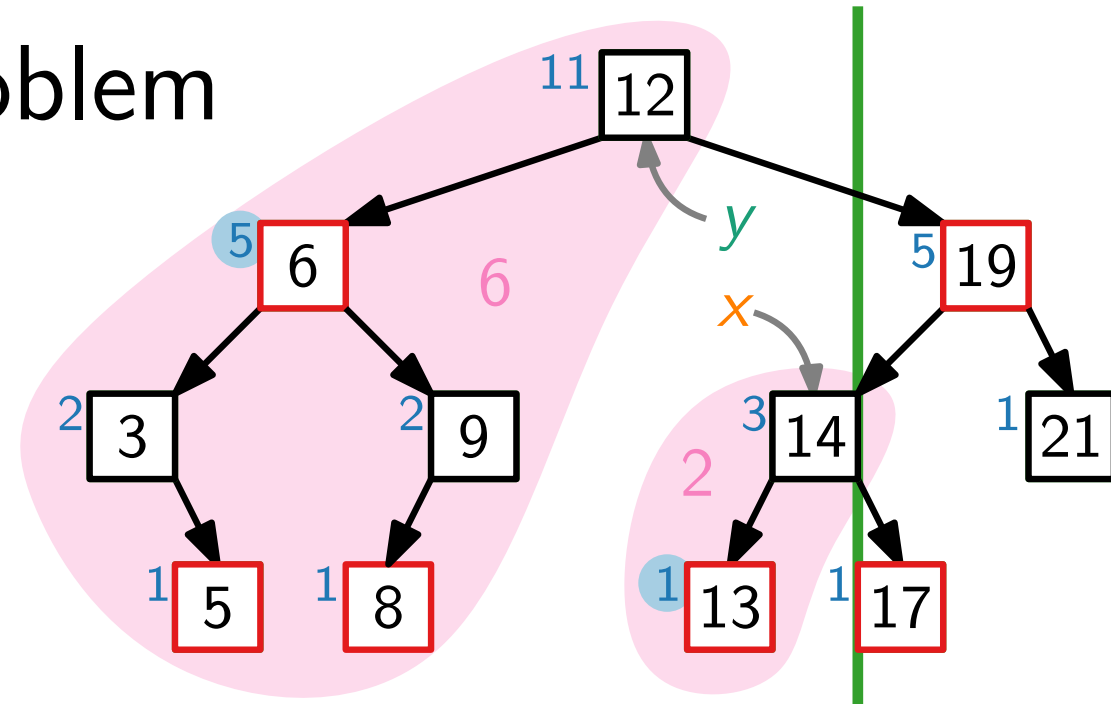
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

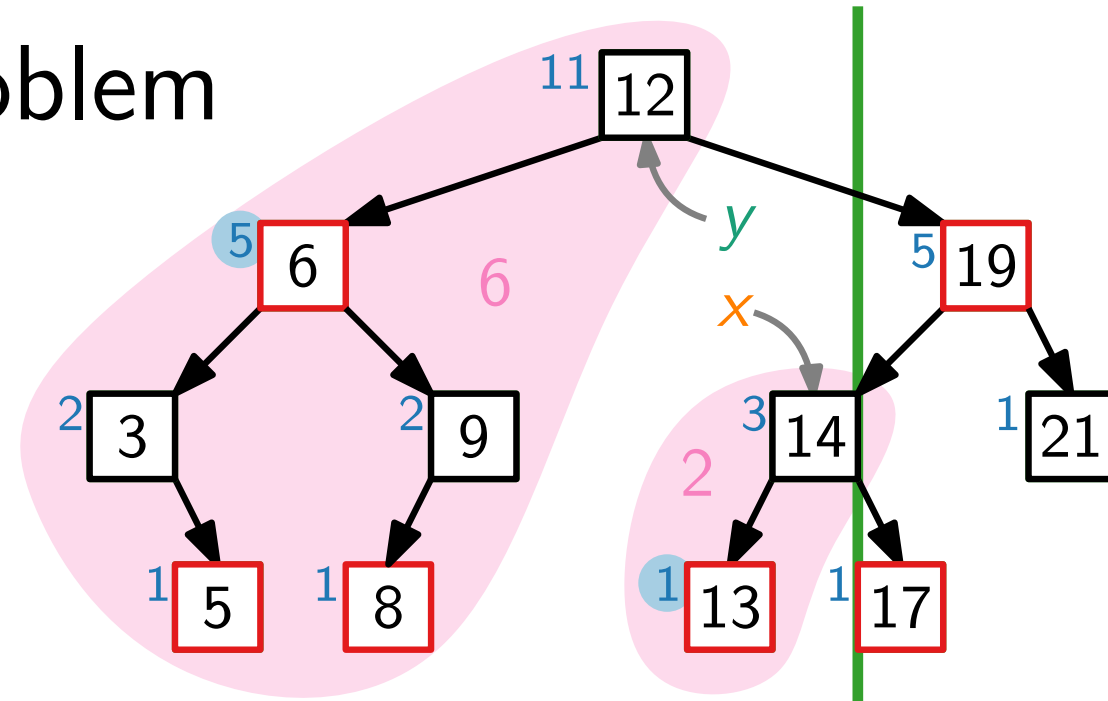
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i):

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

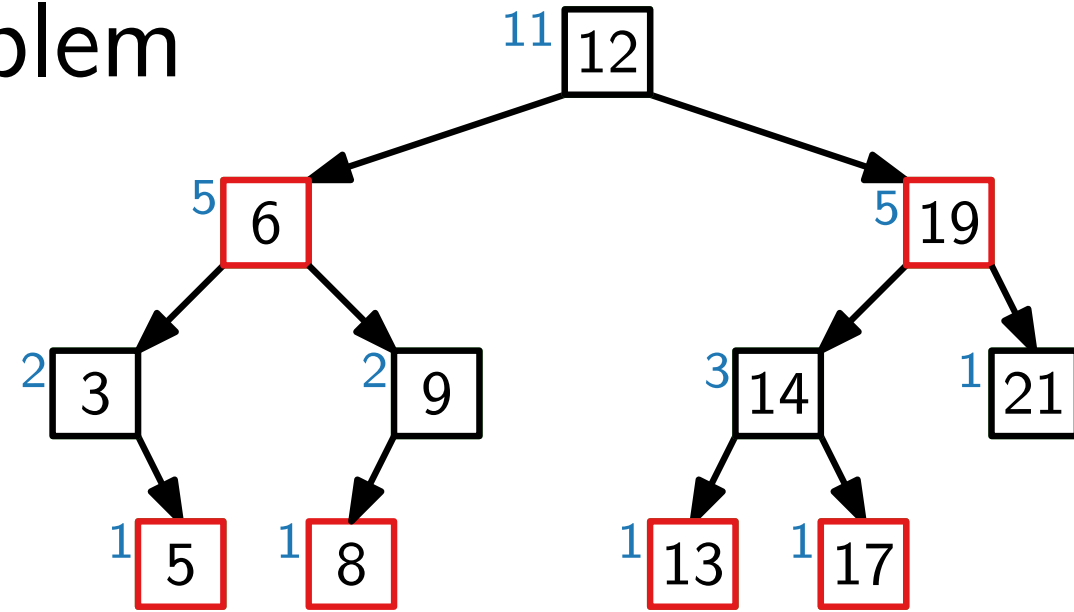
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $r = r + y.p.left.size + 1$ 
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i): $\mathcal{O}(\quad)$

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

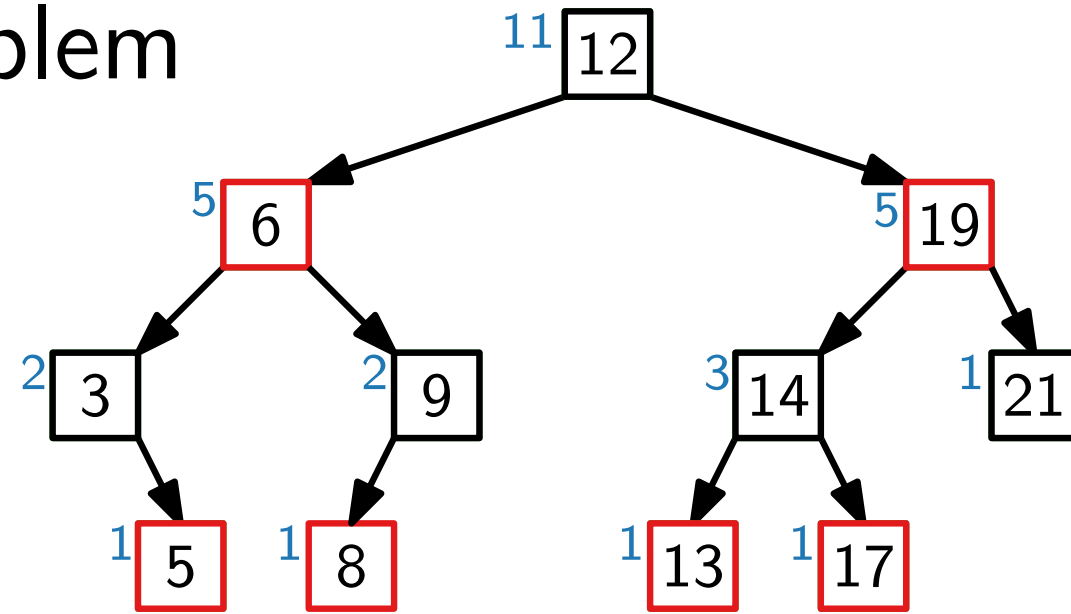
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $r = r + y.p.left.size + 1$ 
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i): $\mathcal{O}(\quad)$

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

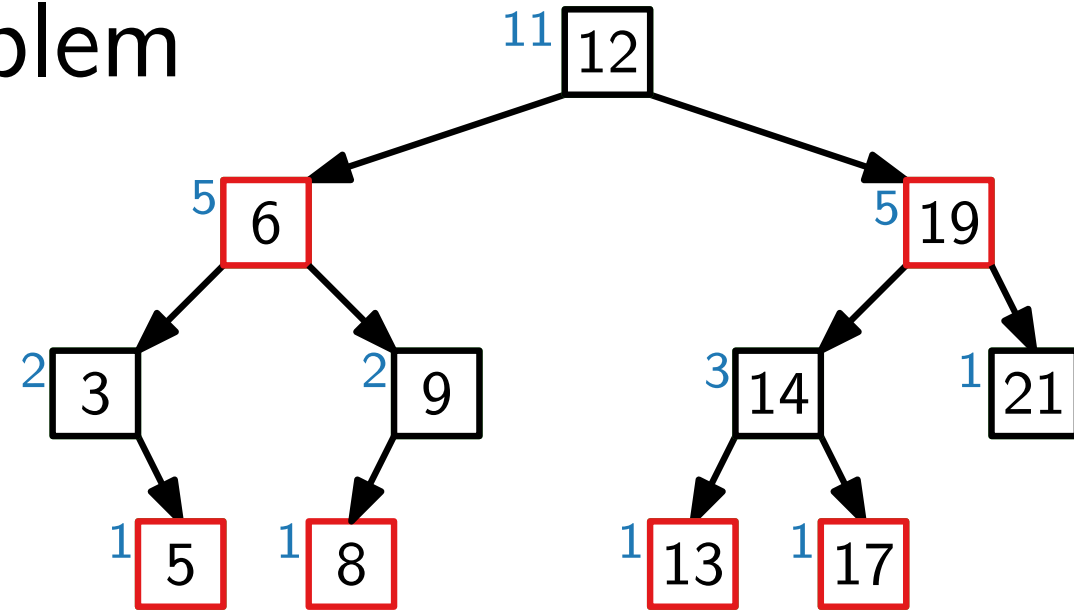
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $r = r + y.p.left.size + 1$ 
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i): $\mathcal{O}(h)$

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x):

```

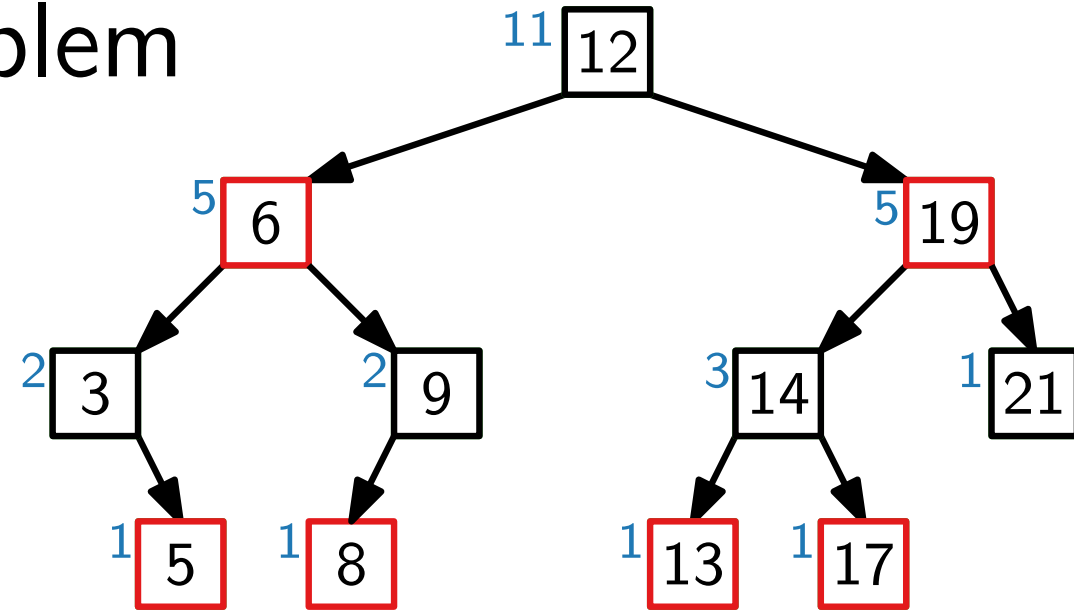
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $r = r + y.p.left.size + 1$ 
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i): $\mathcal{O}(h)$

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
  if  $i < r$  then
    return SELECT( $x.left$ ,  $i$ )
  else
    return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x): $\mathcal{O}()$

```

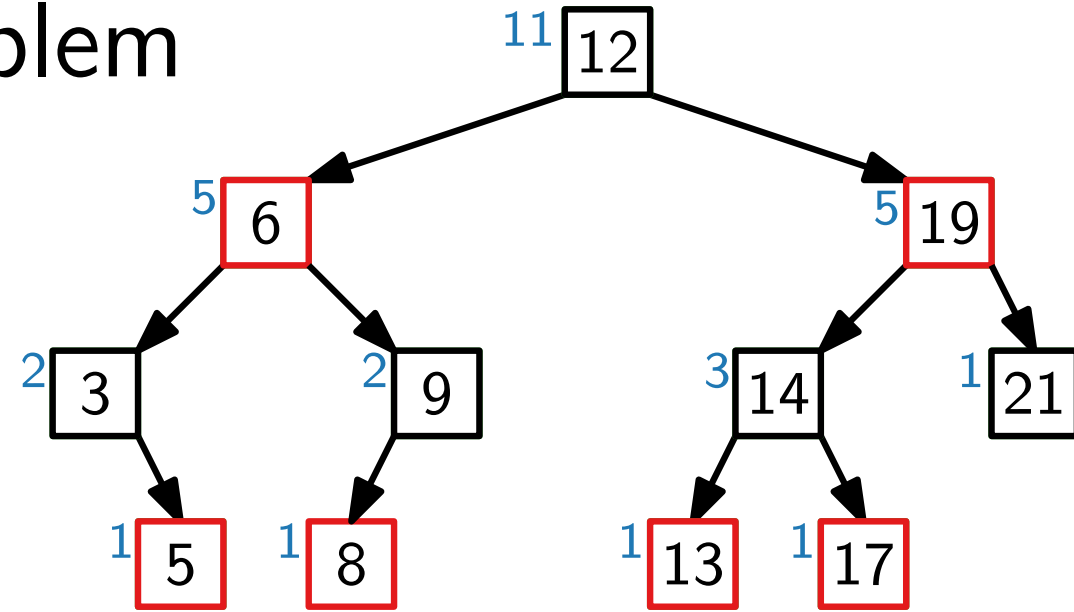
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
  if  $y == y.p.right$  then
     $r = r + y.p.left.size + 1$ 
   $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i): $\mathcal{O}(h)$

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x): $\mathcal{O}()$

```

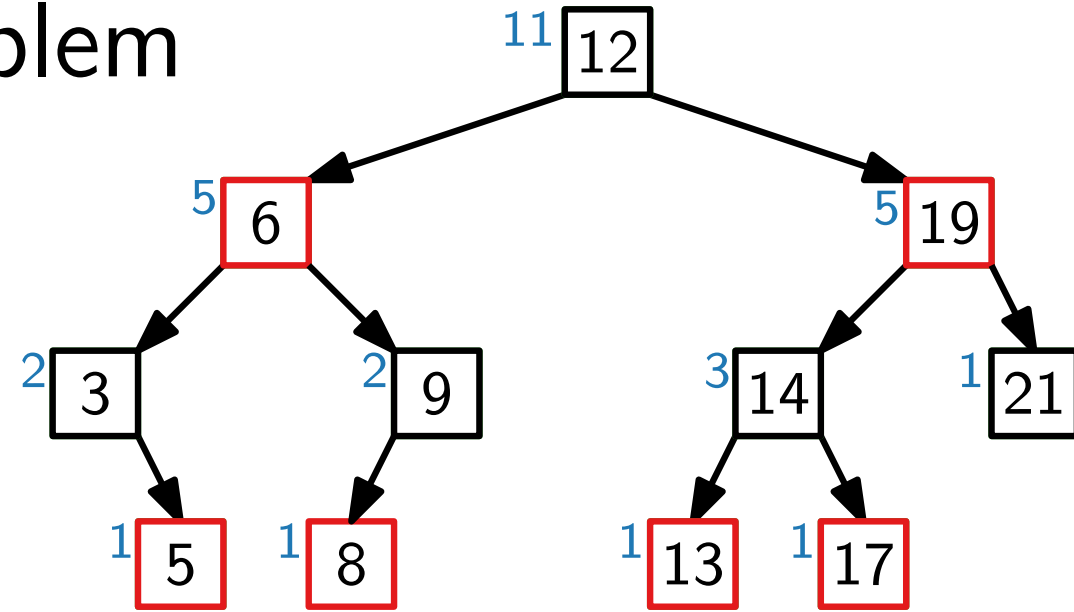
 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $r = r + y.p.left.size + 1$ 
     $y = y.p$ 
return  $r$ 
  
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume
z.B. Rot-Schwarz-Bäume
⇒ Baumhöhe $h \in \mathcal{O}(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume:

3. Aufwand zur Aufrechterhaltung?

später...

für jeden Knoten v , speichere $v.size$

4. SELECT(Node $x = root$, int i): $\mathcal{O}(h)$

```

 $r = x.left.size + 1$ 
if  $i == r$  then return  $x$ 
else
    if  $i < r$  then
        return SELECT( $x.left$ ,  $i$ )
    else
        return SELECT( $x.right$ ,  $i - r$ )
  
```

RANK(Node x): $\mathcal{O}(h)$

```

 $r = x.left.size + 1$ 
 $y = x$ 
while  $y \neq root$  do
    if  $y == y.p.right$  then
         $r = r + y.p.left.size + 1$ 
     $y = y.p$ 
return  $r$ 
  
```

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

Korrektheit von $\text{RANK}()$

Invariante:

```
RANK(Node  $x$ ):  
   $r = x.\text{left.size} + 1$   
   $y = x$   
  while  $y \neq \text{root}$  do  
    if  $y == y.p.\text{right}$  then  
       $r = r + y.p.\text{left.size} + 1$   
       $y = y.p$   
  return  $r$   
  
(vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
```

Korrektheit von `RANK()`


Invariante: Zu Beginn jeder Iteration der while-Schleife ist

```
RANK(Node x):  
    r = x.left.size + 1  
    y = x  
    while y ≠ root do  
        if y == y.p.right then  
            r = r + y.p.left.size + 1  
            y = y.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 x im Teilbaum mit Wurzel y .

```
RANK(Node  $x$ ):  
     $r = x.left.size + 1$   
     $y = x$   
    while  $y \neq root$  do  
        if  $y == y.p.right$  then  
             $r = r + y.p.left.size + 1$   
             $y = y.p$   
    return  $r$   
  
(vorausgesetzt, dass  $T.nil.size = 0$ )
```




Korrektheit von RANK()

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

y -Rang(x)

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```



Korrektheit von RANK()


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

y -Rang(x)

1.) Initialisierung

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```



Korrektheit von `RANK()`

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

y -Rang(x)

1.) Initialisierung

Vor 1. Iteration gilt $y = x \Rightarrow y\text{-Rang}(x) = x.\text{left.size} + 1$.

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
         $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von RANK()

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

y -Rang(x)

1.) Initialisierung

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

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
         $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von `RANK()`


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

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```



Korrektheit von `RANK()`

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

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 

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

Korrektheit von `RANK()`

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

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```

Korrektheit von `RANK()`

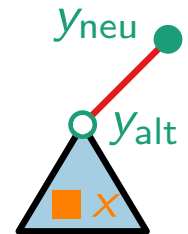
Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der Rang von x im Teilbaum mit Wurzel y .

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.



1. Fall: y war linkes Kind.

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```

Korrektheit von `RANK()`

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

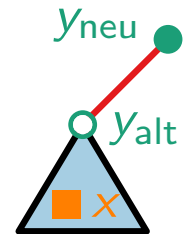
1.) Initialisierung ✓

$y\text{-Rang}(x)$

2.) Aufrechterhaltung

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.



1. Fall: y war linkes Kind.

$\Rightarrow y\text{-Rang}$ von x bleibt gleich.

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```

Korrektheit von `RANK()`

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

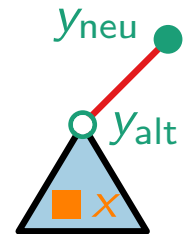
1.) Initialisierung ✓

$y\text{-Rang}(x)$

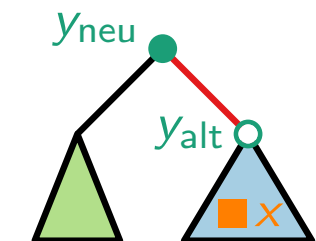
2.) Aufrechterhaltung

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.



1. Fall: y war linkes Kind.
 $\Rightarrow y\text{-Rang von } x \text{ bleibt gleich.}$



2. Fall: y war rechtes Kind.

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von `RANK()`

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

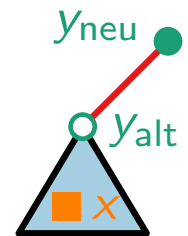
1.) Initialisierung ✓

$y\text{-Rang}(x)$

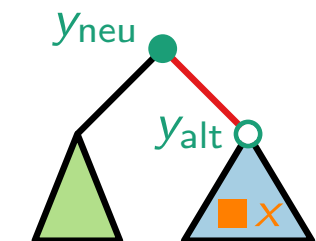
2.) Aufrechterhaltung

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.



1. Fall: y war linkes Kind.
 $\Rightarrow y\text{-Rang}$ von x bleibt gleich.



2. Fall: y war rechtes Kind.
 $\Rightarrow y\text{-Rang}$ von x erhöht sich um Größe des li. Teilbaums von y plus 1 (für y selbst).

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von `RANK()`

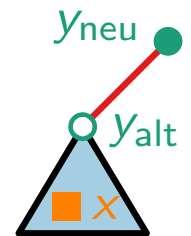
Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der Rang von x im Teilbaum mit Wurzel y .

1.) Initialisierung ✓ $y\text{-Rang}(x)$

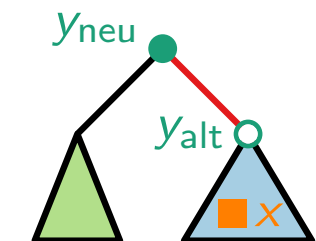
2.) Aufrechterhaltung ✓

Annahme: Invariante galt zu Beginn der aktuellen Iteration.

Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.



1. Fall: y war linkes Kind.
 $\Rightarrow y\text{-Rang}$ von x bleibt gleich.



2. Fall: y war rechtes Kind.
 $\Rightarrow y\text{-Rang}$ von x erhöht sich um Größe des li. Teilbaums von y plus 1 (für y selbst).

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von RANK()

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

1.) Initialisierung ✓ y -Rang(x)

2.) Aufrechterhaltung ✓

3.) Terminierung

```

RANK(Node  $x$ ):
     $r = x.left.size + 1$ 
     $y = x$ 
    while  $y \neq root$  do
        if  $y == y.p.right$  then
             $r = r + y.p.left.size + 1$ 
             $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.nil.size = 0$ )
  
```


Korrektheit von `RANK()`

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

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung ✓

3.) Terminierung

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

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
         $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von `RANK()`

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

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

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

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
         $y = y.p$ 
    return  $r$ 
    (vorausgesetzt, dass  $T.\text{nil.size} = 0$ )
  
```

Korrektheit von `RANK()`

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

1.) Initialisierung ✓ $y\text{-Rang}(x)$

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

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

Zusammenfassung:

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

```

RANK(Node  $x$ ):
     $r = x.\text{left.size} + 1$ 
     $y = x$ 
    while  $y \neq \text{root}$  do
        if  $y == y.p.\text{right}$  then
             $r = r + y.p.\text{left.size} + 1$ 
             $y = y.p$ 
    return  $r$ 

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

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 die Stelle, wo der neue Knoten z eingefügt wird.

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

- Phase I:** Suche die Stelle, wo der neue Knoten z eingefügt wird.
Für alle Knoten y auf dem Weg von der Wurzel zu z :

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

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

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

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

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

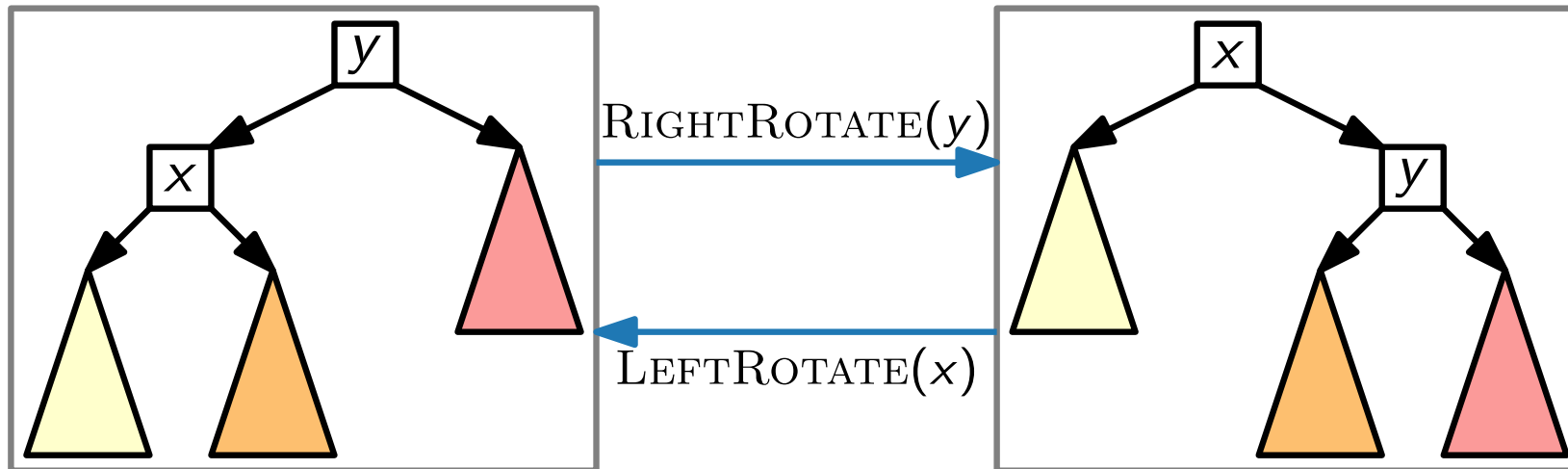
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

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

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:



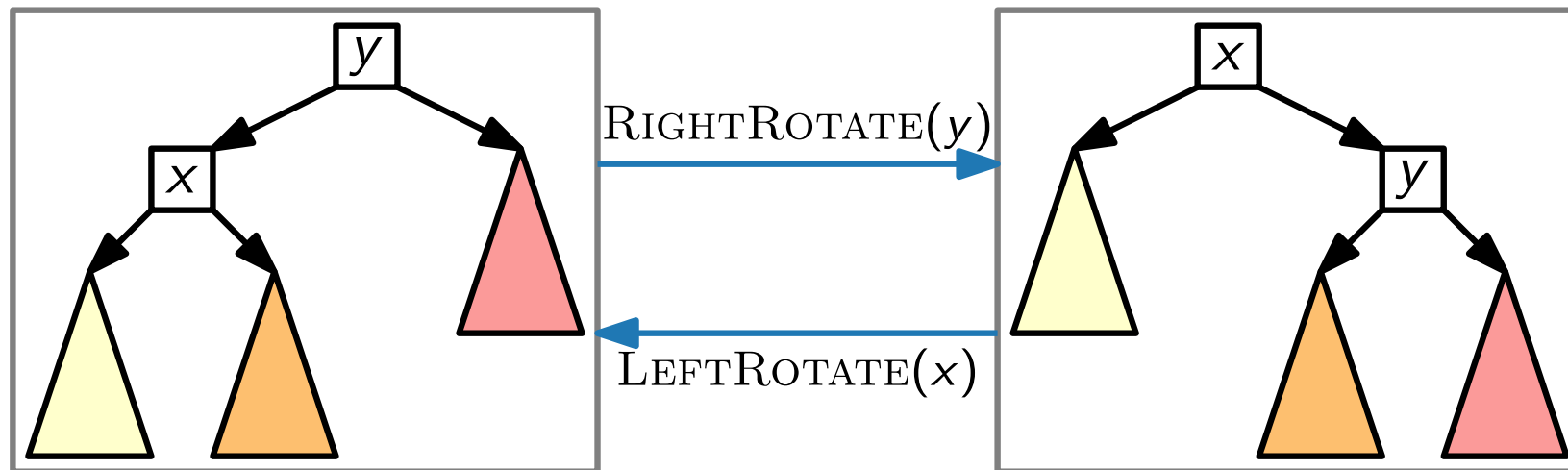
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

Für alle Knoten y auf dem Weg von der Wurzel zu z : Erhöhe $y.size$ 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?

$x.size =$

$y.size =$

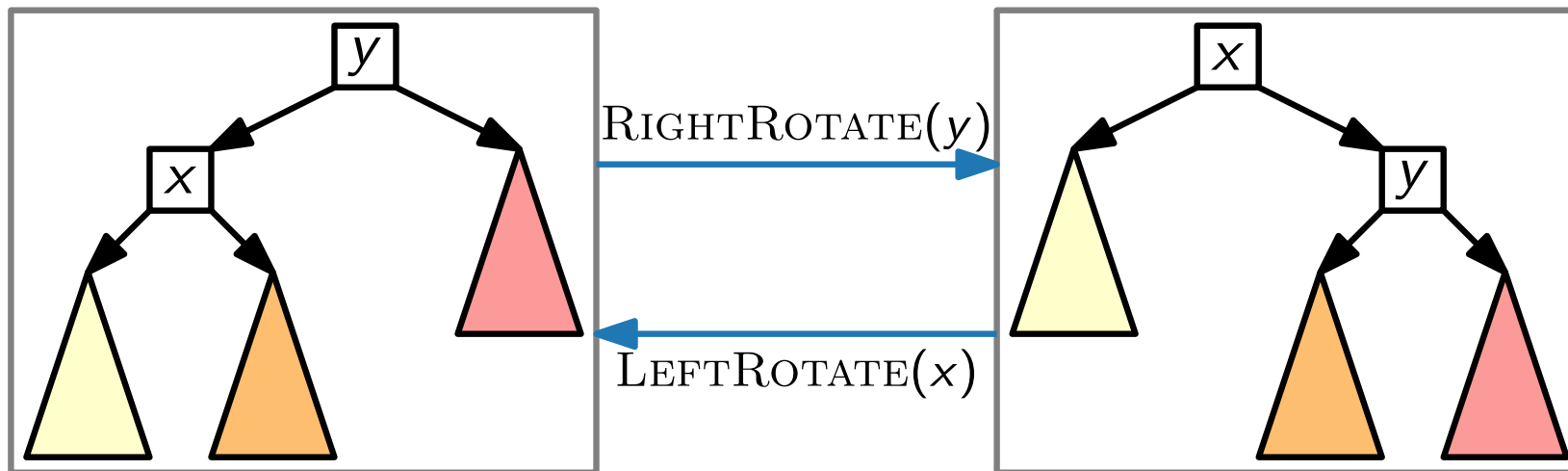
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

Für alle Knoten y auf dem Weg von der Wurzel zu z : Erhöhe $y.size$ 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?

$x.size = y.size$

$y.size =$

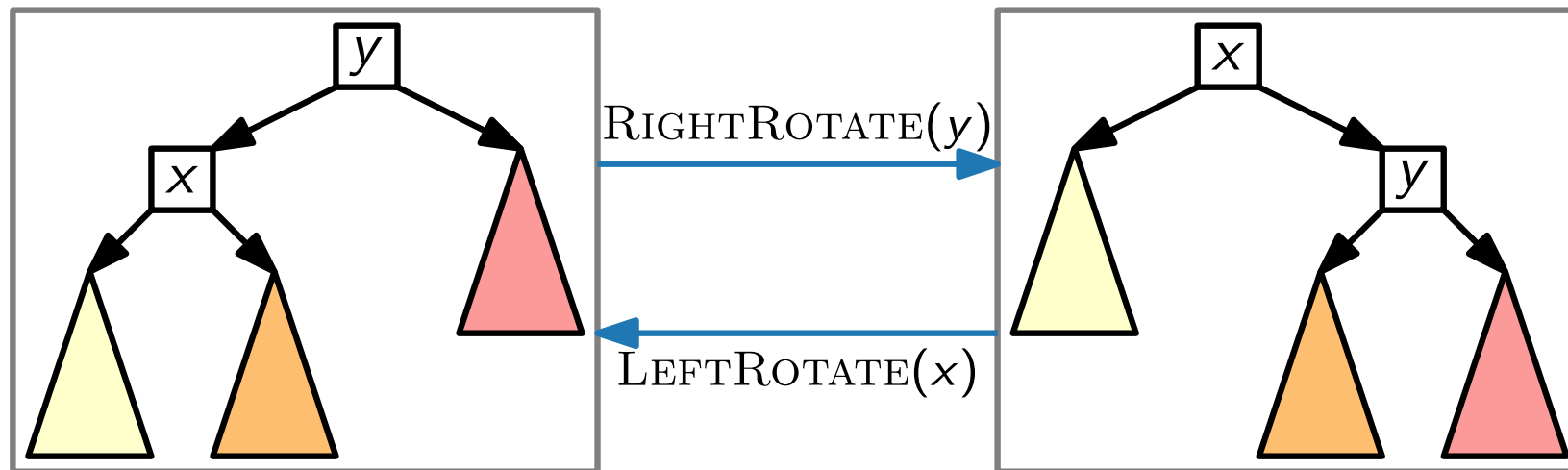
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

Für alle Knoten y auf dem Weg von der Wurzel zu z : Erhöhe $y.size$ 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?

$$x.size = y.size$$

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

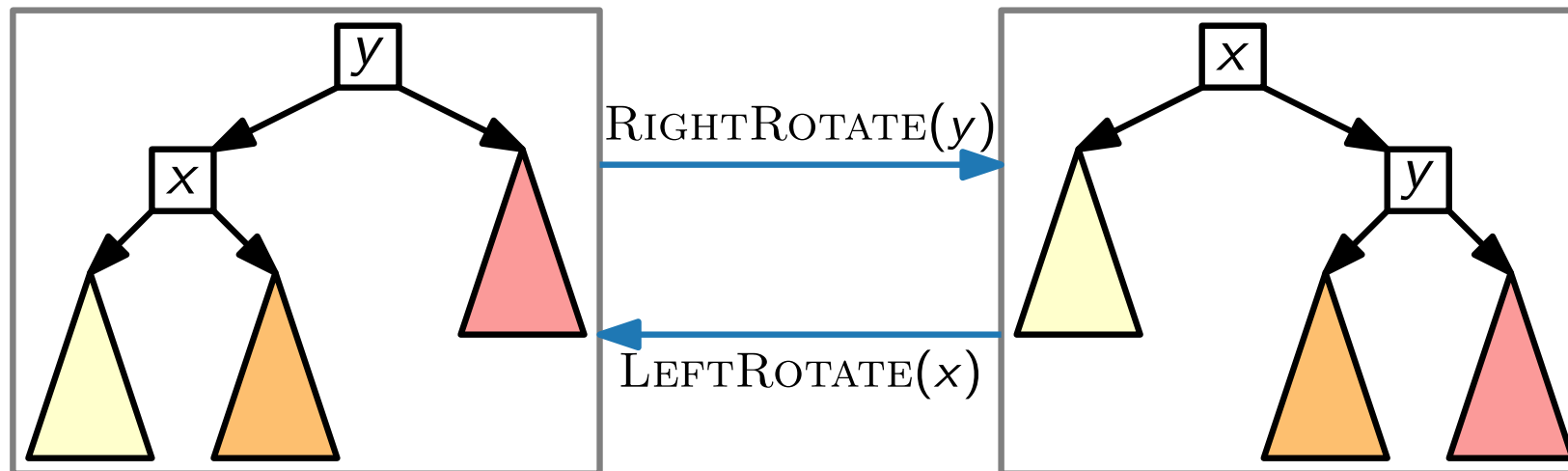
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

Für alle Knoten y auf dem Weg von der Wurzel zu z : Erhöhe $y.size$ 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 $T.nil.size = 0$)

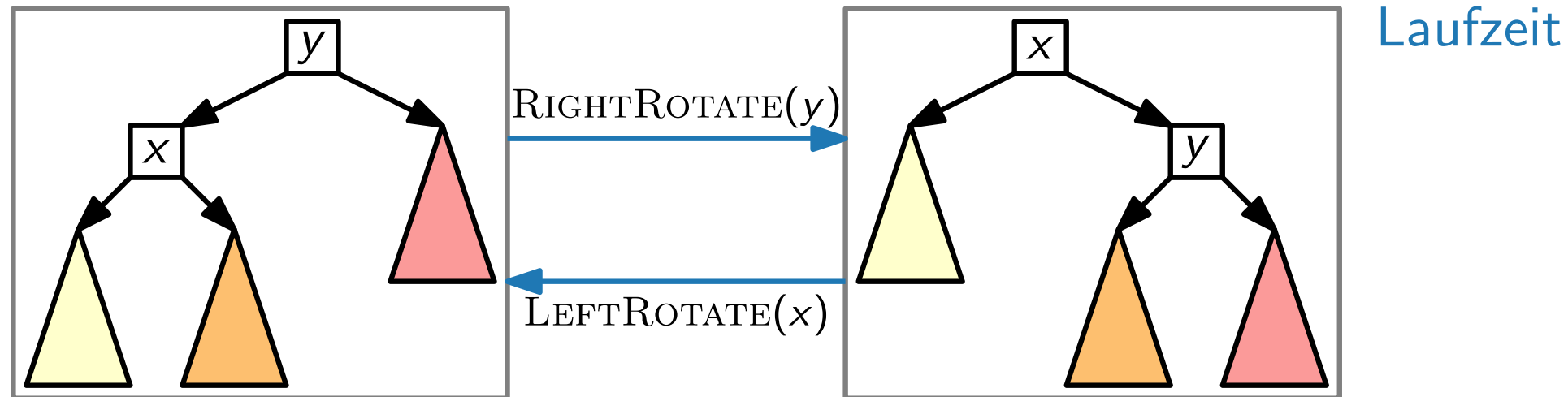
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

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

Für alle Knoten y auf dem Weg von der Wurzel zu z : Erhöhe $y.size$ 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?

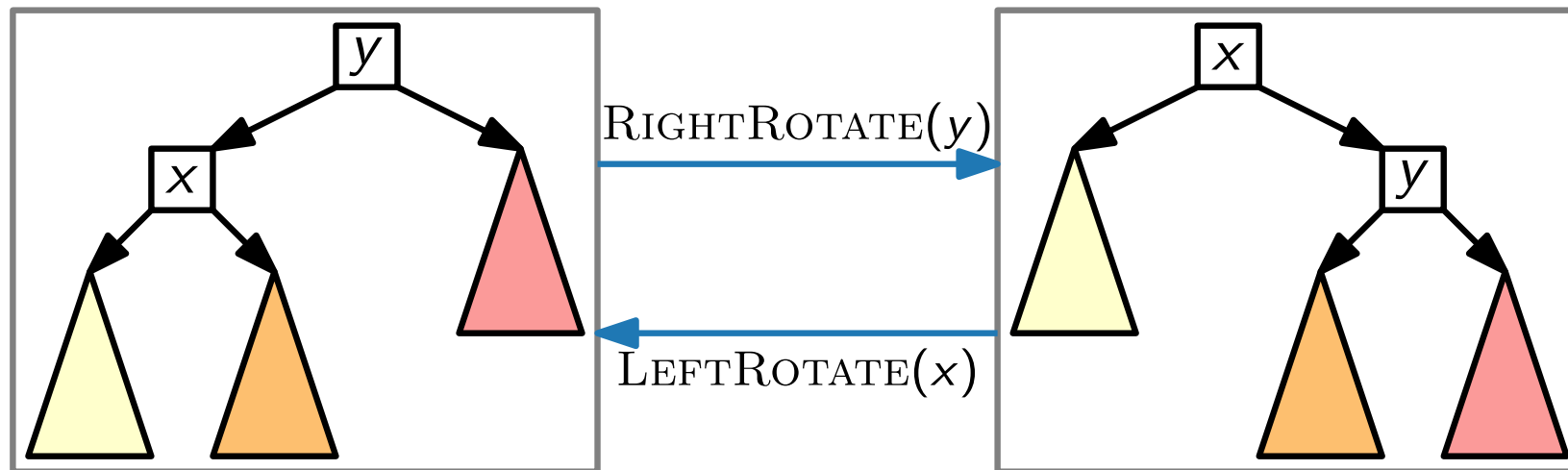
RBINSERT() geht in zwei Phasen vor:

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

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

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

Laufzeit $\mathcal{O}(1)$



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:

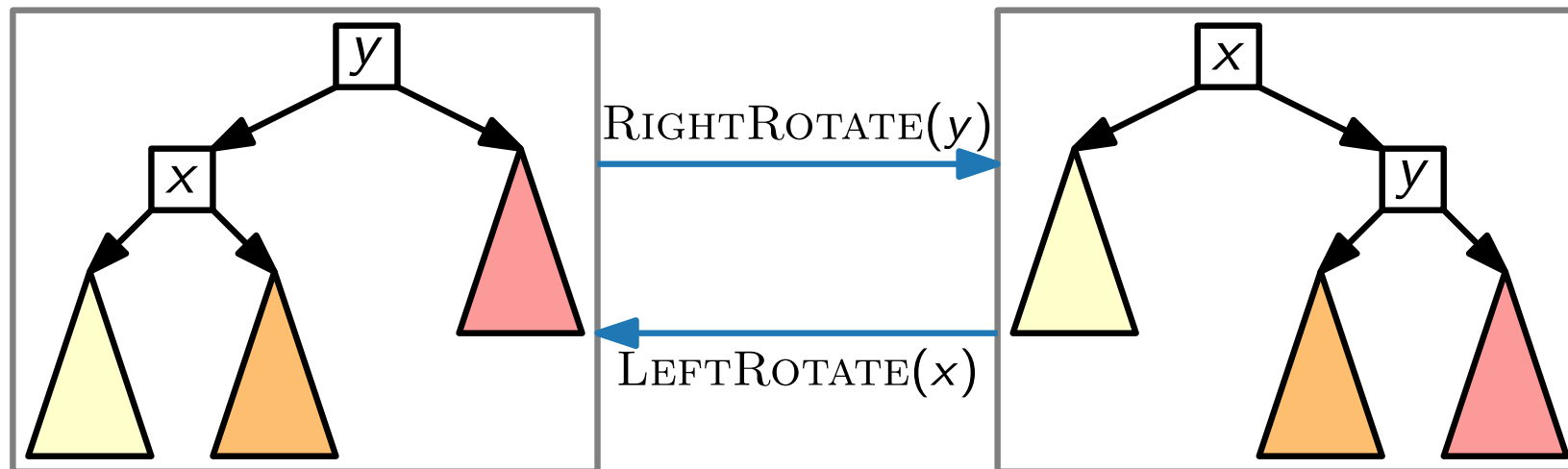
Phase I: Suche die Stelle, wo der neue Knoten z eingefügt wird.

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

Laufzeit

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

Laufzeit $\mathcal{O}(1)$



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:

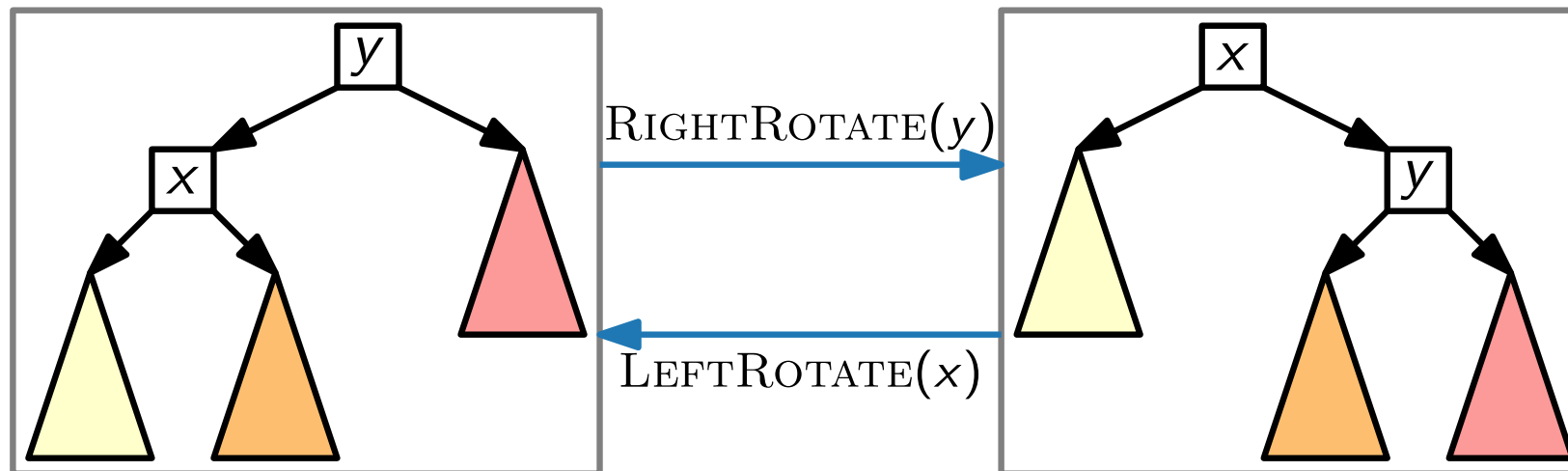
Phase I: Suche die Stelle, wo der neue Knoten z eingefügt wird.

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

Laufzeit $\mathcal{O}(h)$

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

Laufzeit $\mathcal{O}(1)$



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:

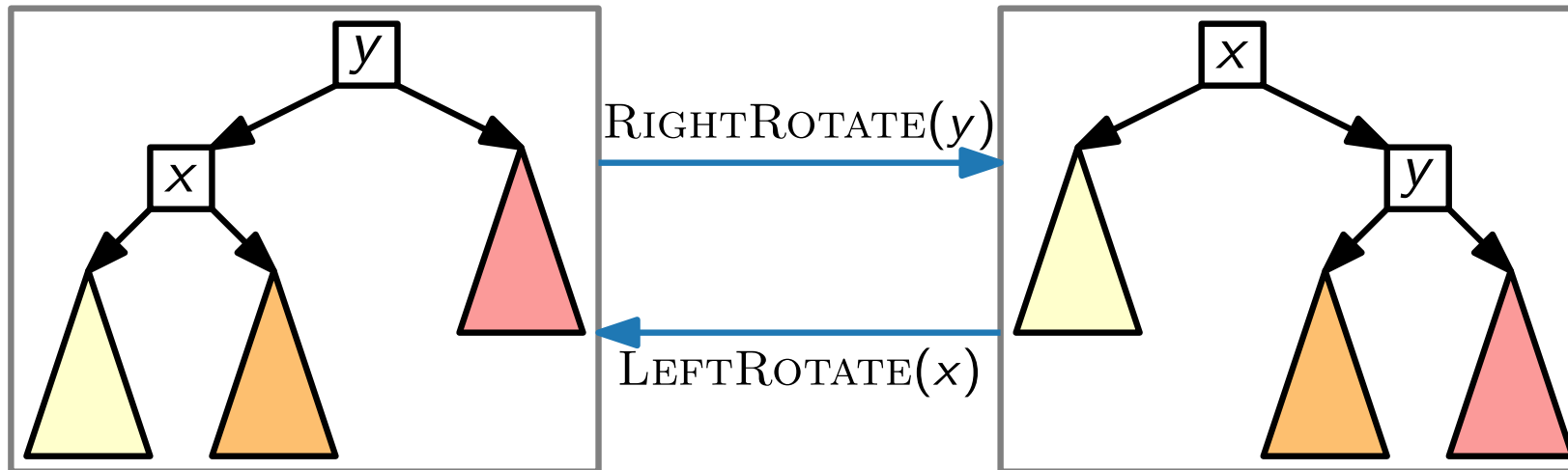
Phase I: Suche die Stelle, wo der neue Knoten z eingefügt wird.

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

Laufzeit $\mathcal{O}(h)$

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

Laufzeit $\mathcal{O}(1)$



Welche Befehle müssen wir an $\text{RIGHTROTATE}(\text{Node } y)$ 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 $T.nil.size = 0$)

zusätzliche Laufzeit fürs Einfügen:

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

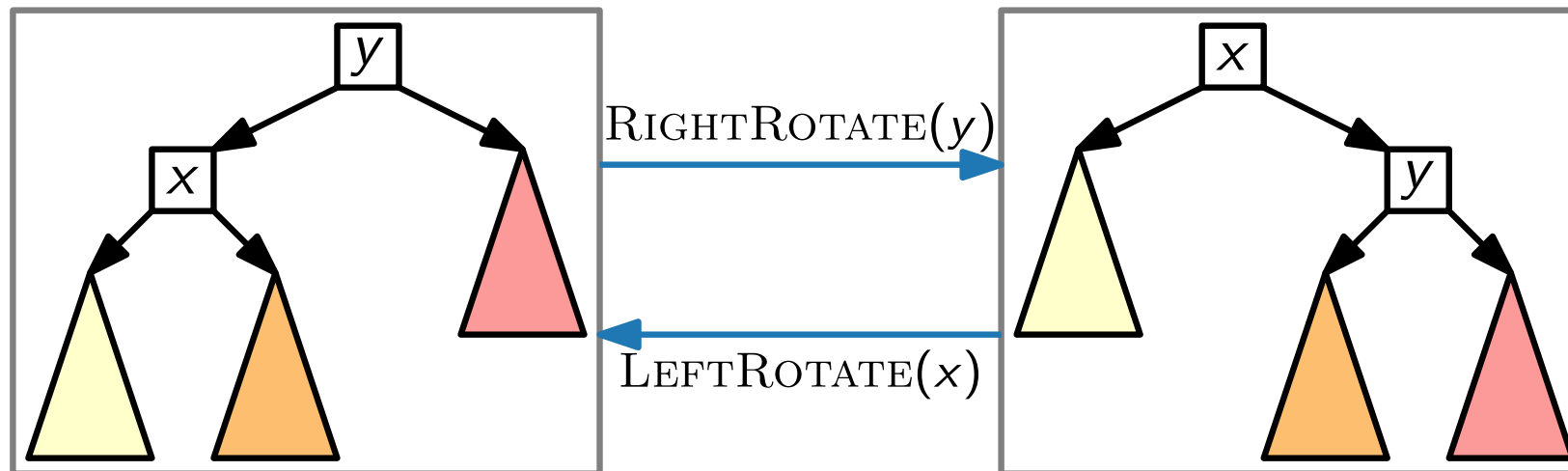
Phase I: Suche die Stelle, wo der neue Knoten z eingefügt wird.

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

Laufzeit $\mathcal{O}(h)$

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

Laufzeit $\mathcal{O}(1)$



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 $T.nil.size = 0$)

zusätzliche Laufzeit fürs Einfügen: $\mathcal{O}(h)$

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBINSERT() geht in zwei Phasen vor:

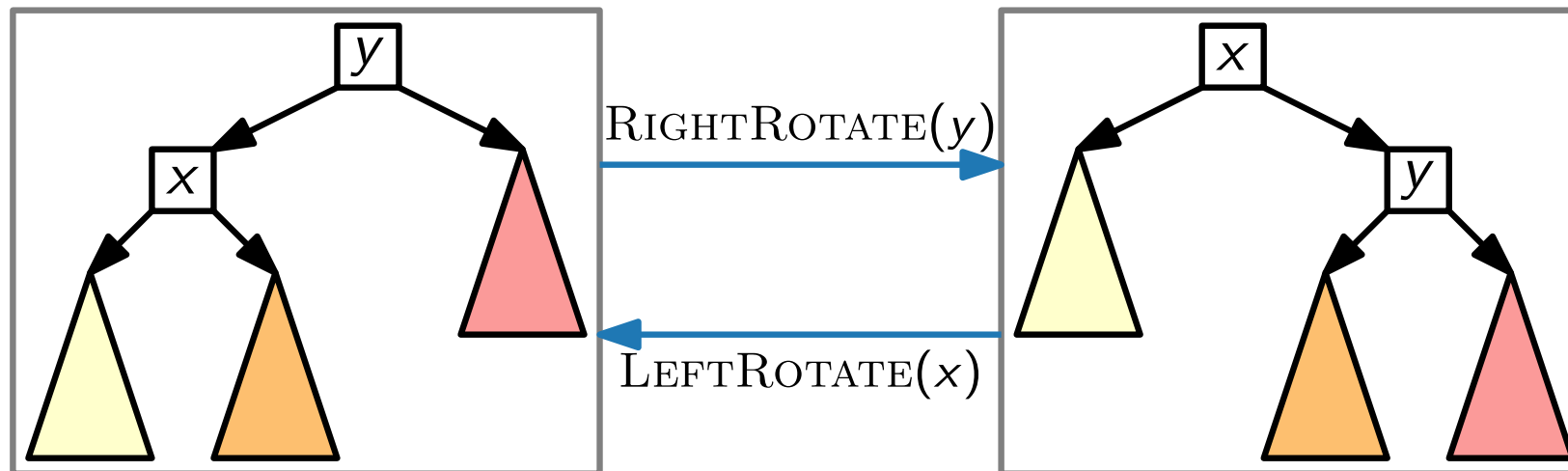
Phase I: Suche die Stelle, wo der neue Knoten z eingefügt wird.

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

Laufzeit $\mathcal{O}(h)$

Phase II (RBINSERTFIXUP): Strukturänderung nur in ≤ 2 Rotationen:

Laufzeit $\mathcal{O}(1)$



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“.

zusätzliche Laufzeit fürs Einfügen: $\mathcal{O}(h)$

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 $\mathcal{O}(\log n)$ Zeit laufen.

Verallgemeinerung

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

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.
Falls für jeden Knoten v gilt:

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 $f(v.right)$) in konstanter Zeit berechnen.



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 $f(v.right)$) in konstanter Zeit berechnen.



Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten,

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 $f(v.right)$) in konstanter Zeit berechnen.



Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $\mathcal{O}(\log n)$ 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 , $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) in konstanter Zeit berechnen.



Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $\mathcal{O}(\log n)$ 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 , $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) in konstanter Zeit berechnen.



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

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

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 , $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) in konstanter Zeit berechnen.



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

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

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 , $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) in konstanter Zeit berechnen.



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

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

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 [CLRS, Kapitel 14.3]):

Intervall-Baum

Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

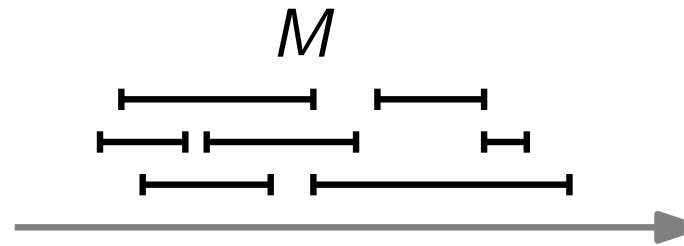
verwaltet eine Menge M von Intervallen

Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen

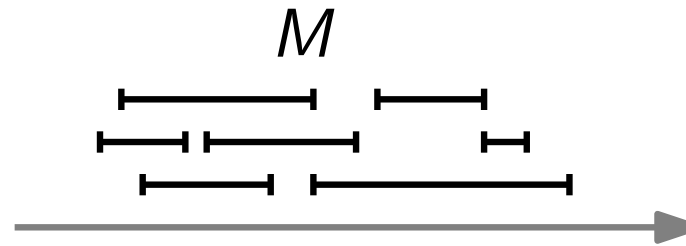


Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:



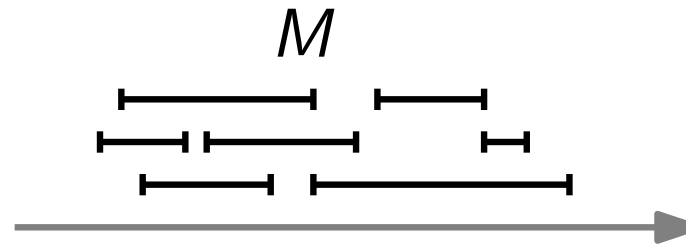
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- ptr INSERT (Interval i)



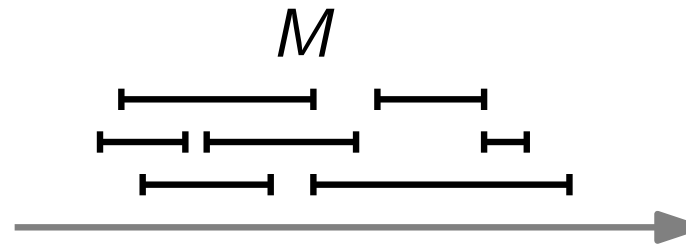
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- `ptr INSERT (Interval i)`
- `DELETE(ptr x)`



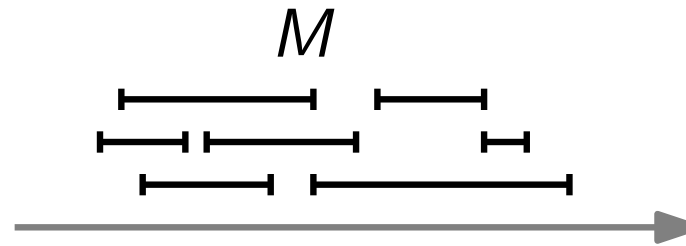
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- `ptr INSERT (Interval i)`
- `DELETE(ptr x)`
- `ptr SEARCH(Interval i)`



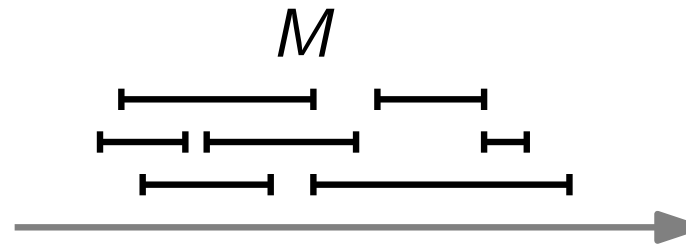
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- `ptr INSERT (Interval i)`
- `DELETE(ptr x)`
- `ptr SEARCH(Interval i)` ^{neu}



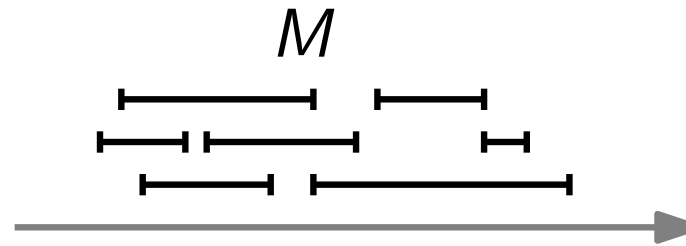
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- `ptr INSERT (Interval i)`
- `DELETE(ptr x)`
- `ptr SEARCH(Interval i)` neu



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

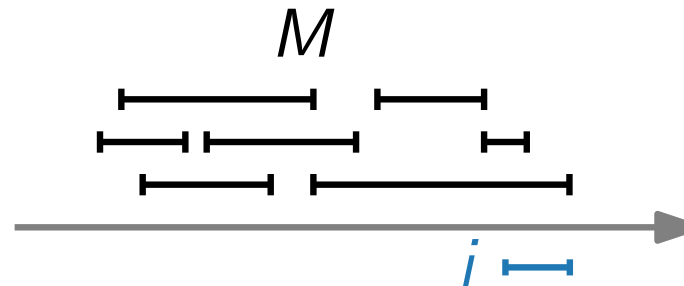
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- ptr INSERT (Interval i)
- DELETE(ptr x)
- ptr SEARCH(Interval i) neu



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

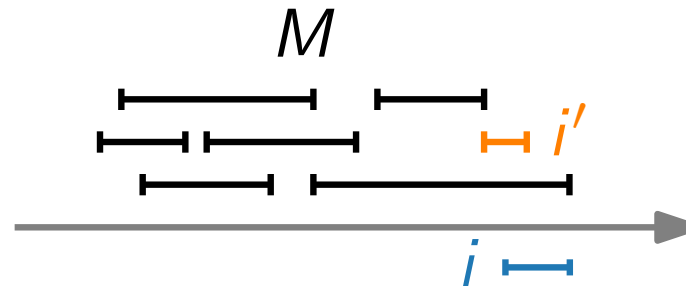
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- `ptr INSERT (Interval i)`
- `DELETE(ptr x)`
- `ptr SEARCH(Interval i)` neu



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

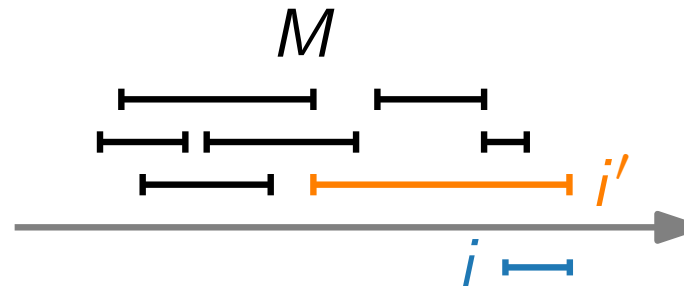
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen [CLRS, Kapitel 14.3]):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- `ptr INSERT (Interval i)`
- `DELETE(ptr x)`
- `ptr SEARCH(Interval i)` neu



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