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:

- 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.

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.

D.h. wir wollen zu jedem Zeitpunkt effizient

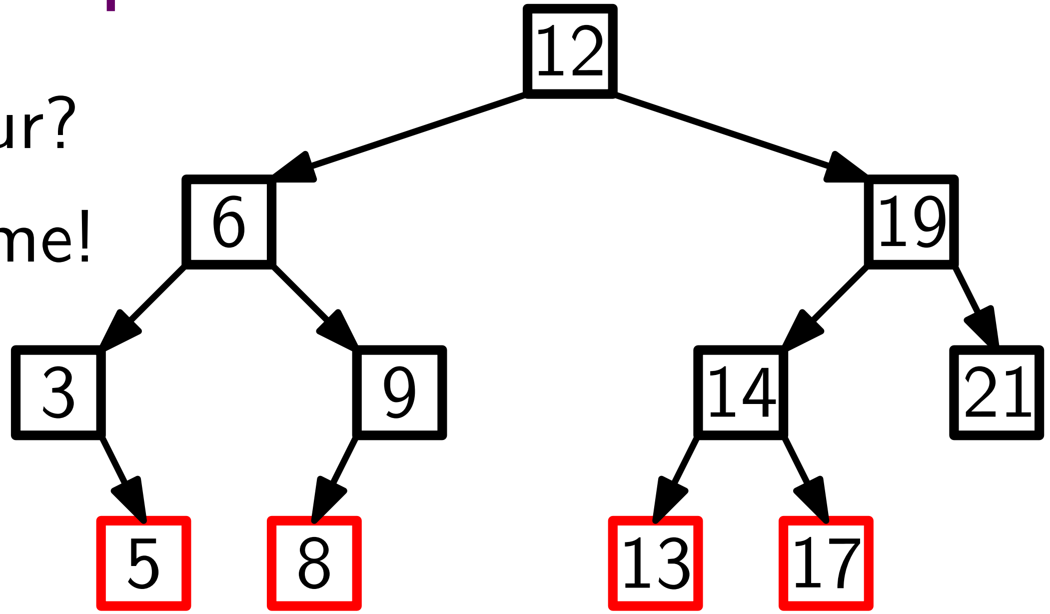
- das i.-kleinste Element (z.B. den Median): Elem Select(int i)
- den *Rang* eines Elements: int Rank(Elem e)

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?
   – 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!

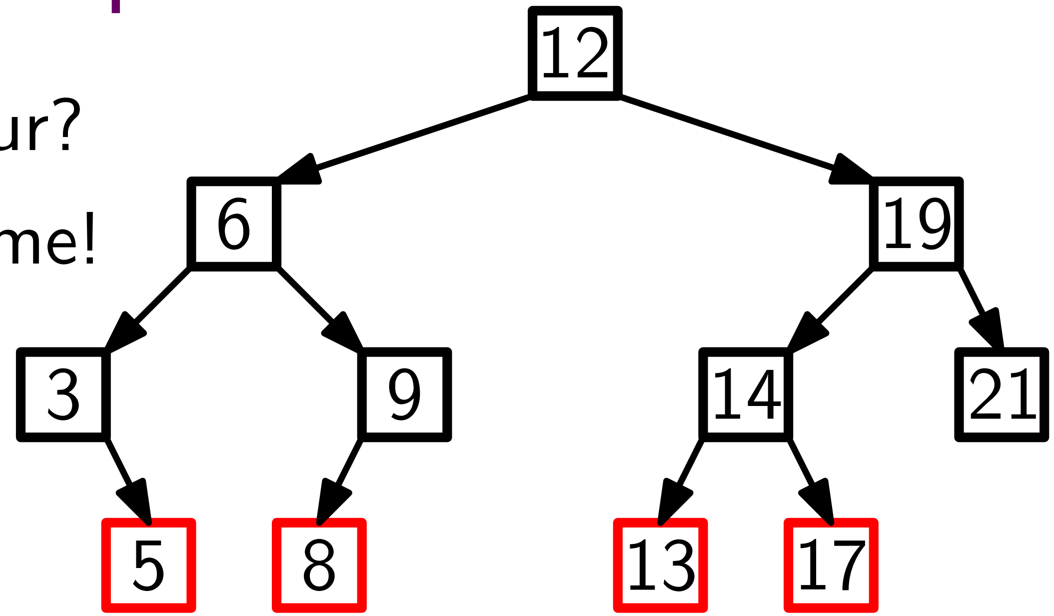
Select(int i):

```
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? Abschätzung bestmöglich?

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

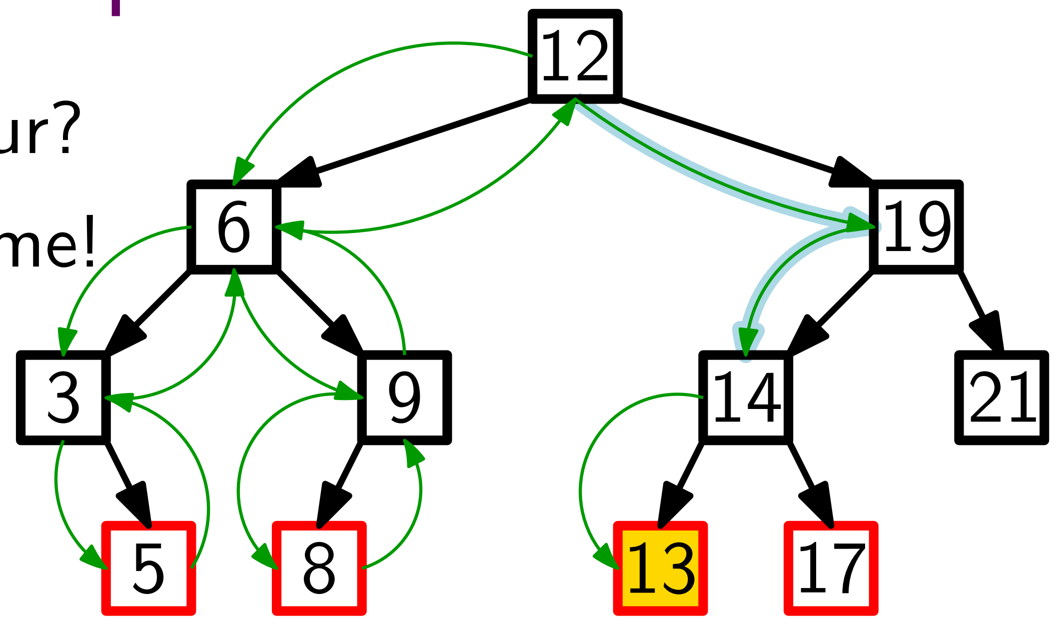
$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

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

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

while $v \neq nil$ **and** $i > 1$ **do**

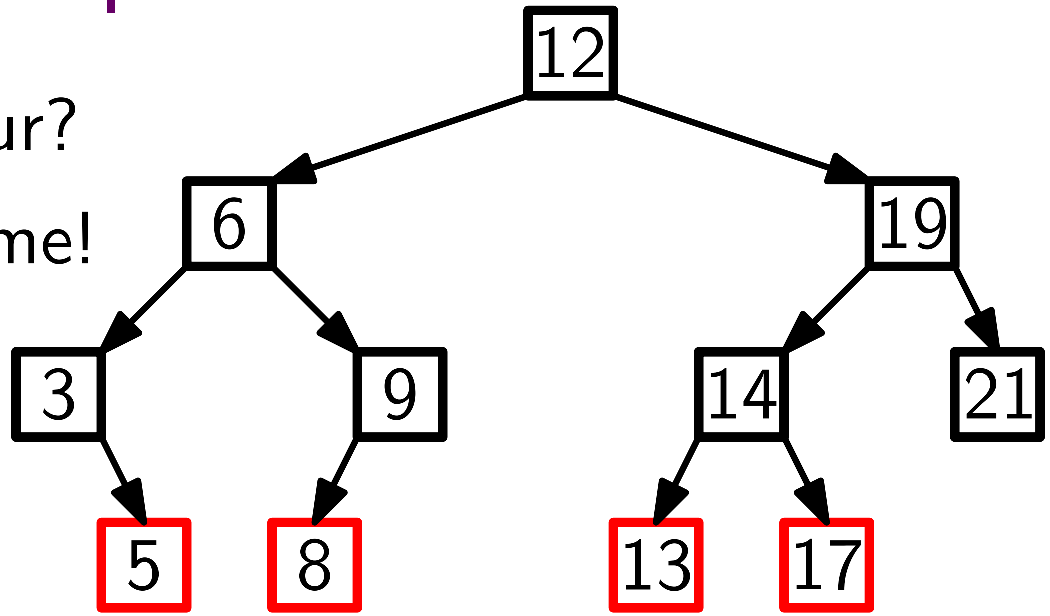
$v = \text{Minimum}()$
while $v \neq nil$ **and** $i > 1$ **do**
  $v = \text{Successor}(v)$
  $i = i - 1$
return v

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

$j = 0$
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 $h \in O(\log n)$

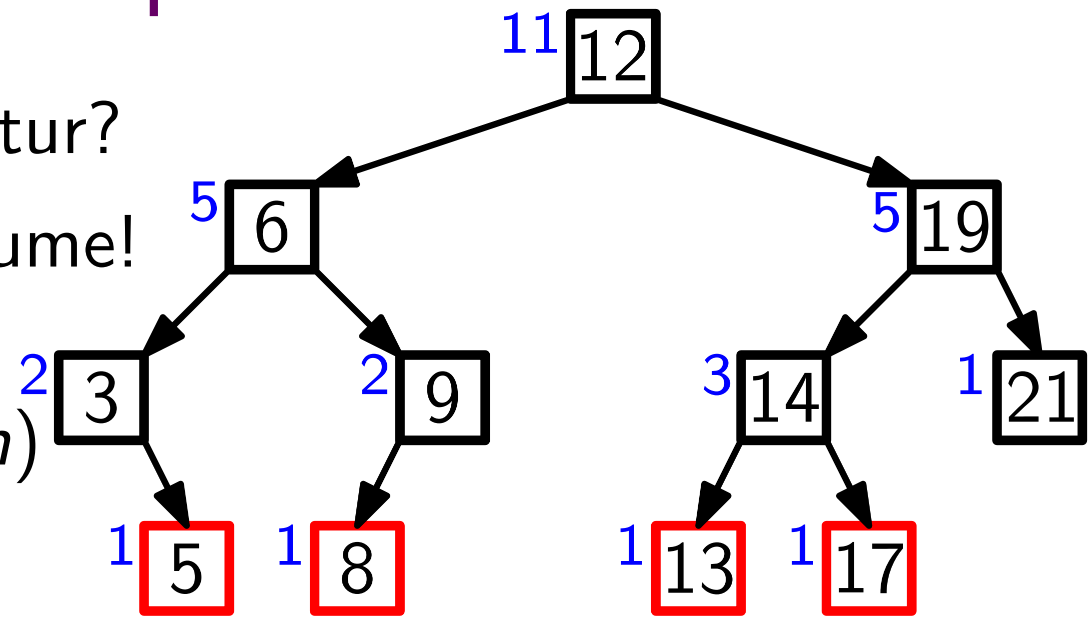


2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?
   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?
   – Größen der Teilbäume: für jeden Knoten v, speichere $v.size$

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

Select:

$r = v.left.size + 1$
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: 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 $T.\mathit{nil}.\mathit{size} = 0$)

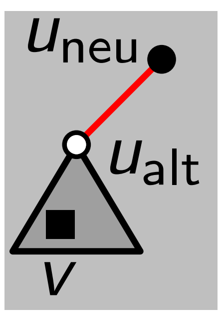
Korrektheit von Rank()

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 ✓

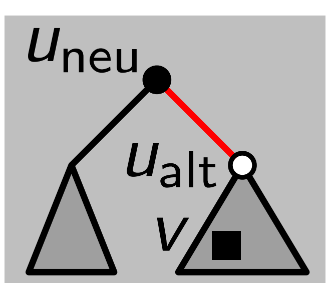
Annahme: Inv. galt zu Beginn der aktuellen Iteration.
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



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 ✓

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 ≠ 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?

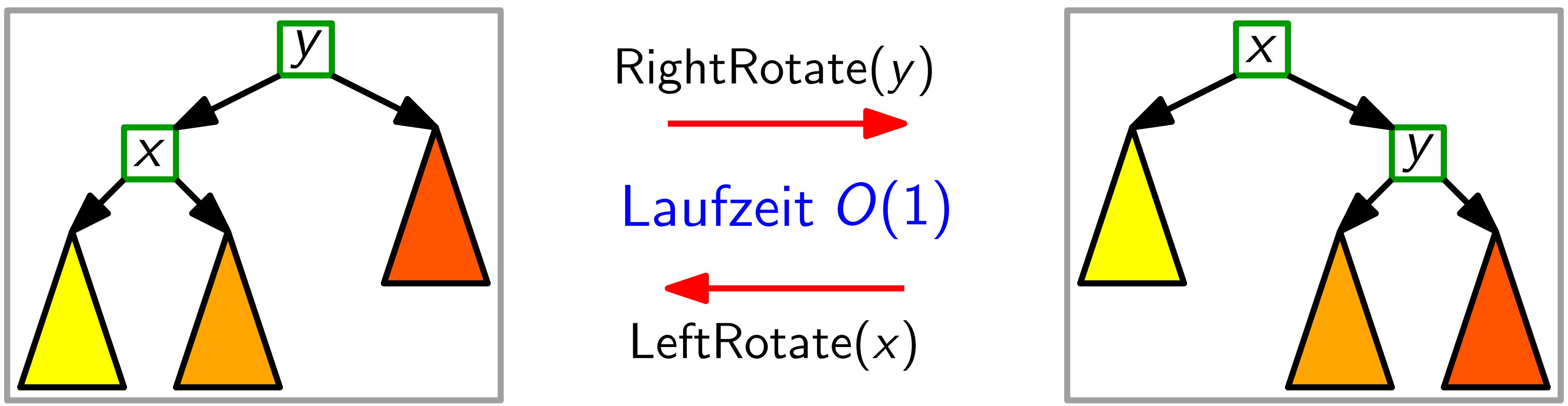
RBInsert() geht in zwei Phasen vor:

zusätzliche Laufzeit fürs Einfügen: $O(h)$

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

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

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?

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

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

[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, $v.\mathit{left}$, $v.\mathit{right}$ (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

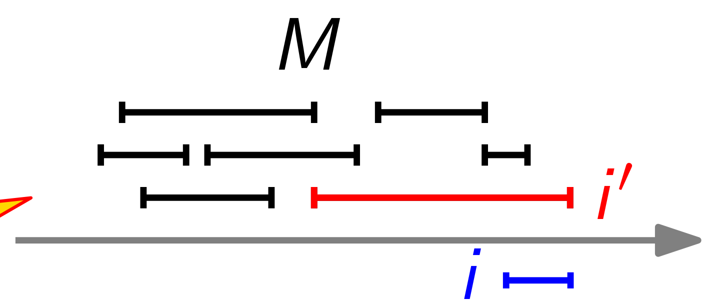
zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

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

  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...