

Algorithmen und Datenstrukturen

Wintersemester 2023/24

15. Vorlesung

Augmentieren von Datenstrukturen

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel und Schlange
- 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 zusätzliche Informationen 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!

```
double getMean() // gibt Fließkommazahl zurück  
    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!

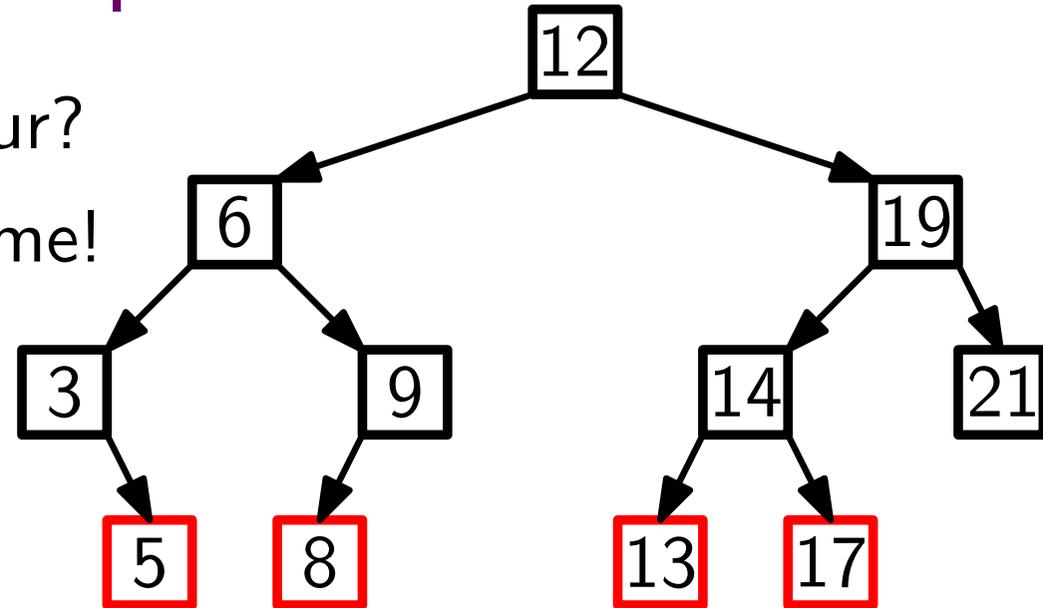
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe $h \in O(\log n)$



2. Welche Extraintformation 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
  
```

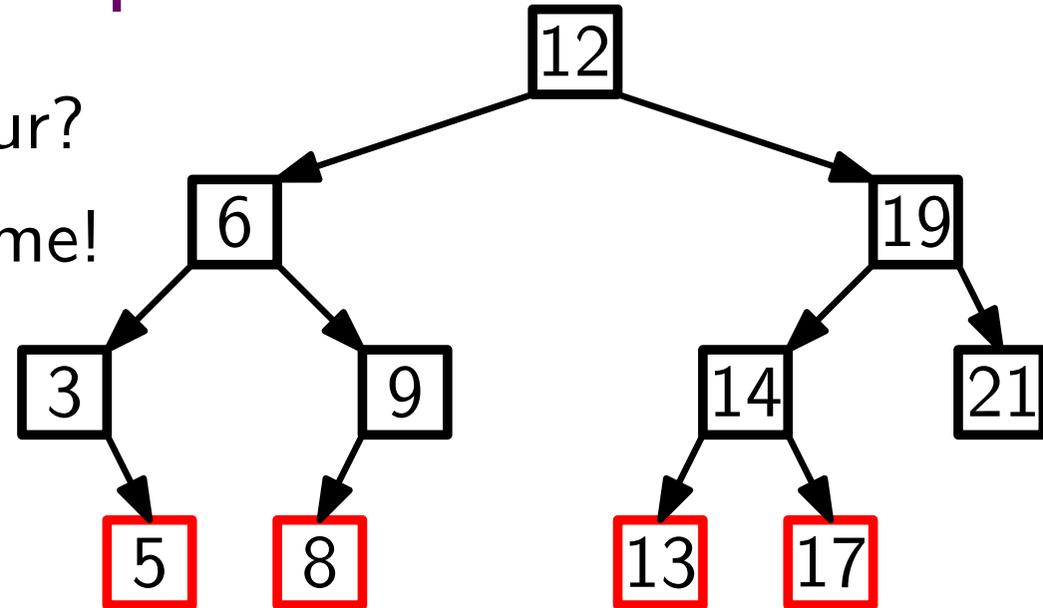
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe $h \in O(\log n)$



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

4. Implementiere! Laufzeit? Abschätzung bestmöglich?

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

```

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

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

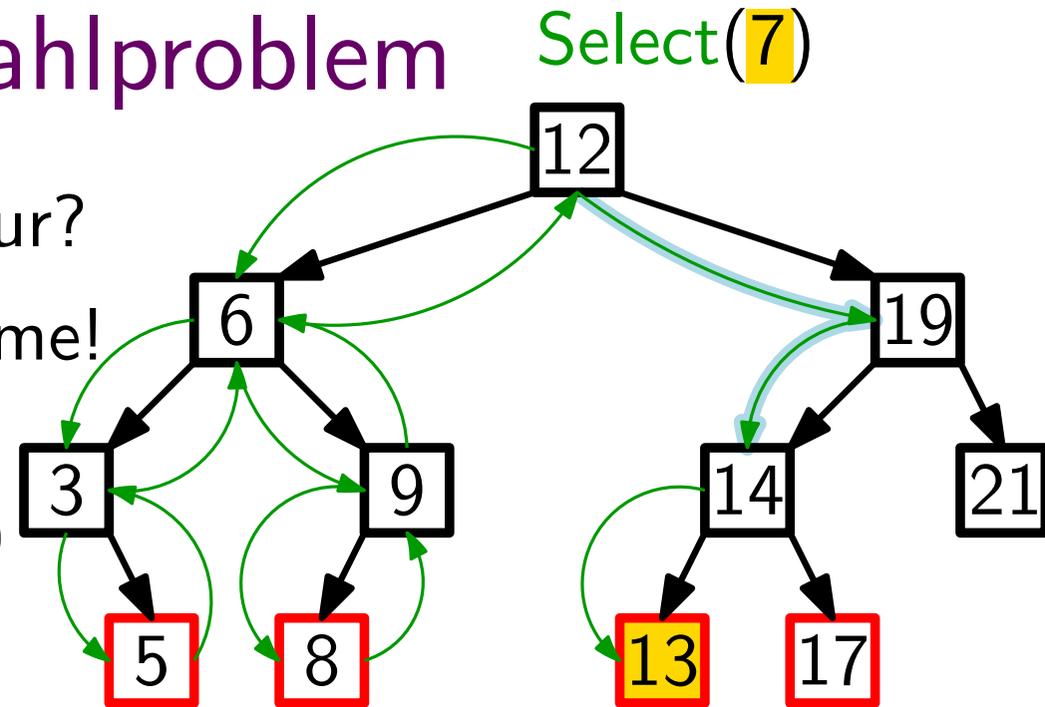
```

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

Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

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



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

4. Implementiere! Laufzeit?

Select(int i): $O(i + h)$

```

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

```

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

```

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

```

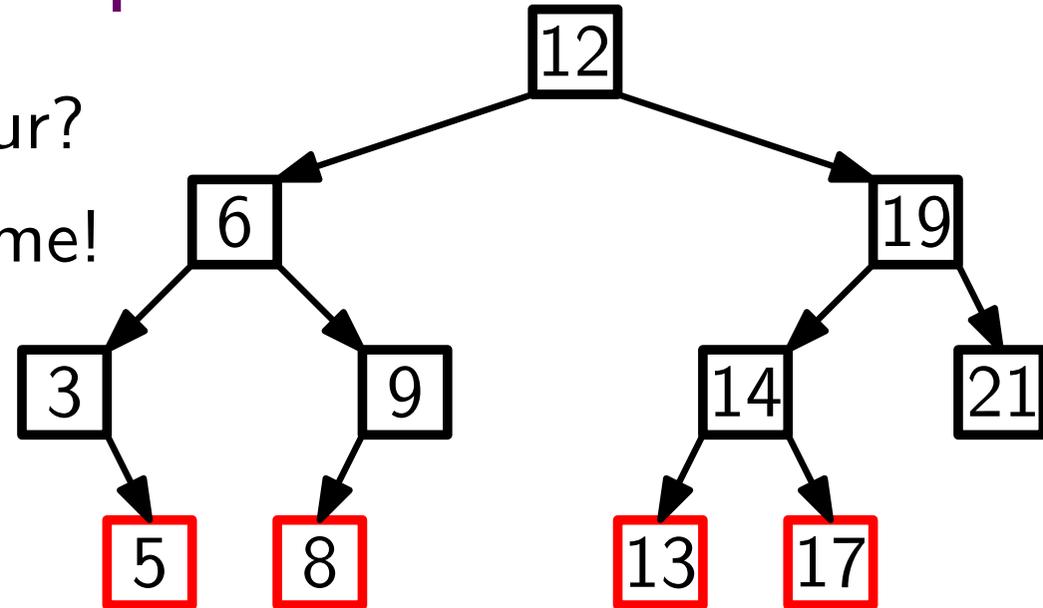
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe $h \in O(\log n)$



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

4. Implementiere! Laufzeit?

Select(int i): $O(i + h)$ **Rank**(Node v): $O(rank + h)$

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



Vorlesungsumfrage

Sehr geehrter Herr Prof. Dr. Wolff,
die Rücklaufquote der Umfrage "Algorithmen und
Datenstrukturen" liegt aktuell bei 8%
(Teilnehmerzahl insgesamt: 296).

Dieser Wert liegt unter einer definierten Schwelle
von 50%. Wir möchten Sie daher bitten, Ihre
Teilnehmer nochmals zu ermuntern, sich an der
Umfrage zu beteiligen.

Bitte beachten Sie, dass die Umfrage am **15.12.2023**
23:59:00 geschlossen wird.

Ihr EvaSys Administrator

Vorlesungsumfrage – jetzt!

Bitte suchen Sie die E-Mail, die Sie von EvaSys bekommen haben, und klicken Sie dort auf den Link zur Umfrage.

4. Additional remarks

4.1 What did you especially like in this course?

4.2 From your perspective, what could be improved? What do you criticize?

Zum Beispiel:

- Folien
- Übungsaufgaben
- Roter Faden?
- Beweise?
- Reaktion auf Fragen?
- Buch zur Vorlesung?
- Zwischentests
- Aufgaben in der VL
- ...

All questions are optional. Should a question not be applicable to a course, you can leave the answer open.

1 Course as a whole

1.1 Please rate the course as a whole.

very good insufficient

2 Lecture

2.1 The lecturer is well prepared and presents the material in a way that is easy to understand.

completely agree do not agree at all

2.2 The supplied lecture materials (writings on the blackboard, presentation slides, videos, additional literature) are well prepared and improve my understanding of the course contents.

completely agree do not agree at all

2.3 Portions of the lecture were held online (live stream, recorded lecture) and helped improve the lecture.

completely agree do not agree at all no online parts

3 Exercises

3.1 The tutor is well prepared and presents the material in a way that is easy to understand.

completely agree do not agree at all no exercises

3.2 The exercises are comprehensible and improve my understanding of the course contents.

completely agree do not agree at all no exercises

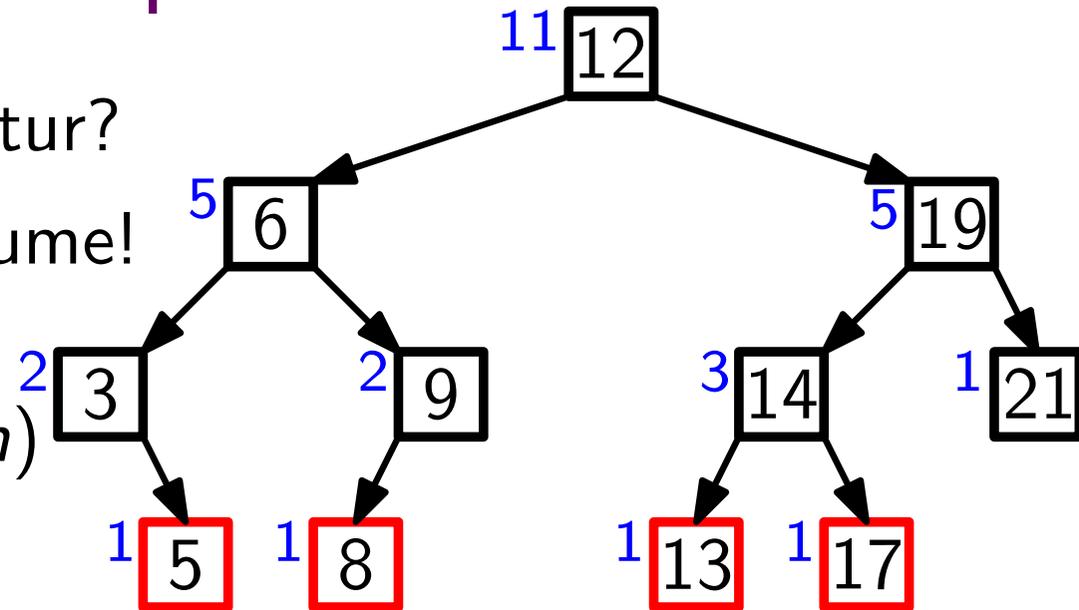
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe $h \in O(\log n)$



2. Welche Extraintformation aufrechterhalten?

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

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

```

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

```

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

```

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

```

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

Korrektheit von Rank()

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

u-Rang von v

1.) Initialisierung

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

Rank(Node v):

```
 $r = v.\text{left.size} + 1$ 
```

```
 $u = v$ 
```

```
while  $u \neq \text{root}$  do
```

```
  if  $u == u.p.\text{right}$  then
```

```
     $r = r + u.p.\text{left.size} + 1$ 
```

```
     $u = u.p$ 
```

```
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 v im Teilbaum mit Wurzel u .

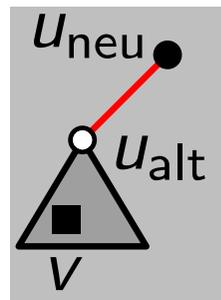
1.) *Initialisierung* ✓

u -Rang von v

2.) *Aufrechterhaltung* ✓

Annahme: Inv. galt zu Beginn der aktuellen Iteration.

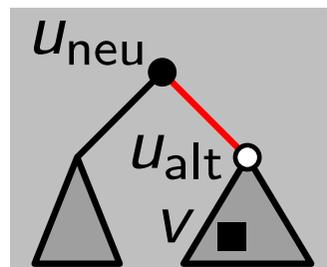
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.
 \Rightarrow u -Rang von v bleibt gleich.

2. Fall: u war rechtes Kind.

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



Rank(Node v):

$r = v.left.size + 1$

$u = v$

while $u \neq root$ **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 Rang von v im Teilbaum mit Wurzel u ,

u-Rang von v

1.) *Initialisierung* ✓

2.) *Aufrechterhaltung* ✓

3.) *Terminierung* ✓

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

Zusammenfassung:

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

Rank(Node v):

$r = v.left.size + 1$

$u = v$

while $u \neq root$ **do**

if $u == u.p.right$ **then**

$r = r + u.p.left.size + 1$

$u = u.p$

return r

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

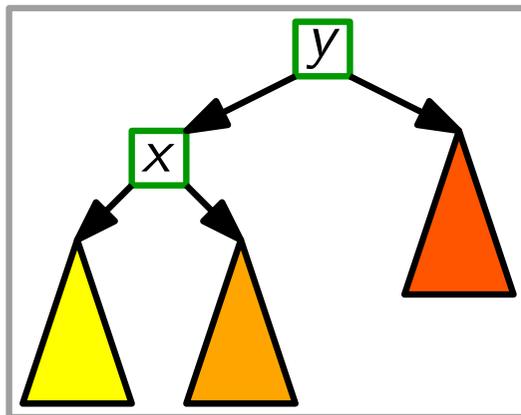
3. Aufwand zur Aufrechterhaltung der Extraintormation?

RBInsert() geht in zwei Phasen vor:

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

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

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



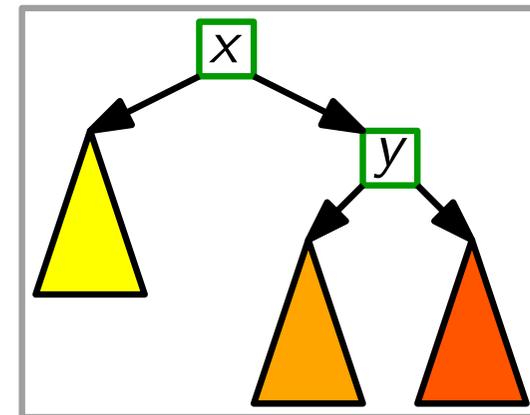
RightRotate(y)



Laufzeit $O(1)$



LeftRotate(x)



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: $O(h)$

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 $O(\log n)$ Zeit laufen.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.
Falls für jeden Knoten v gilt:
 $f(v)$ lässt sich in $O(1)$ Zeit aus Information in v ,
 $v.left$, $v.right$ (inklusive $f(v.left/right)$) berechnen.



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 $f \equiv size$.

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]

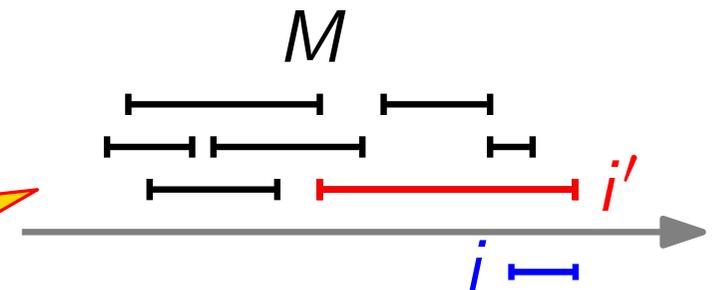
Noch ein Beispiel

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)



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...