

Algorithmen und Datenstrukturen

Wintersemester 2017/18

15. Vorlesung

Augmentieren von Datenstrukturen

Special Guest: Fabian Feitsch

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!

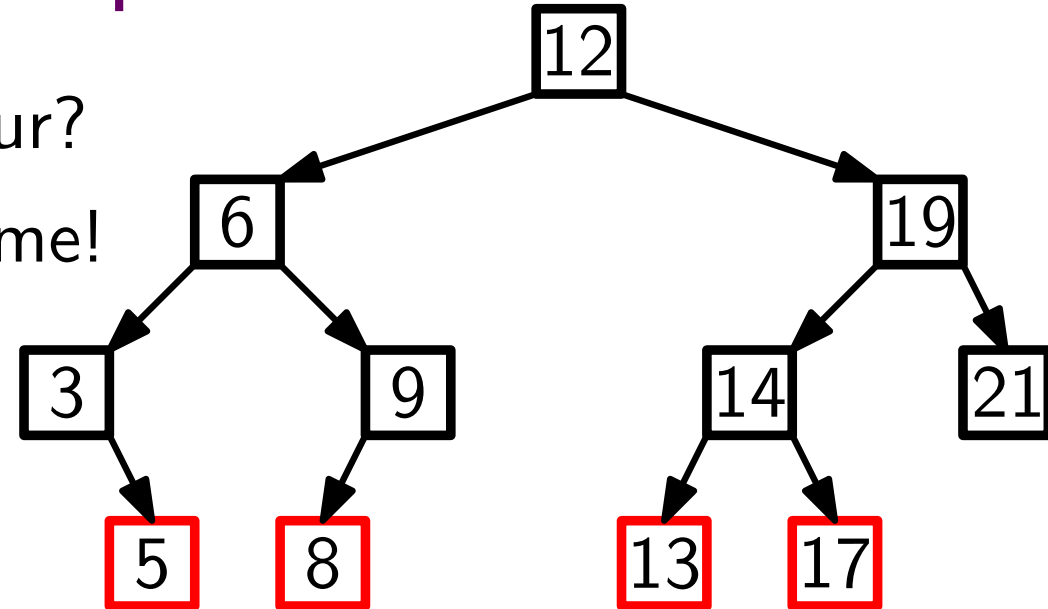
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe = $O(\log n)$



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

4. Implementiere neue Operationen!

Select(int i):

```

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

Rank(Node v):

```

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

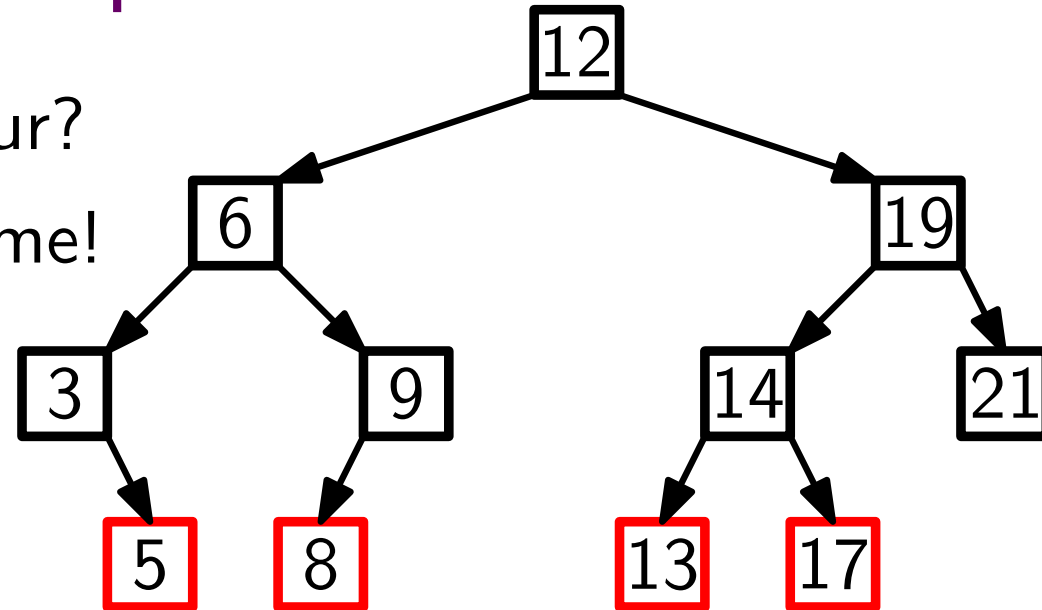
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe = $O(\log n)$



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

4. Implementiere! Laufzeit? Abschätzung bestmöglich?

Select(int i): $O(i \cdot \log n)$

```

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

Rank(Node v): $O(\text{rank} \cdot \log n)$

```

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

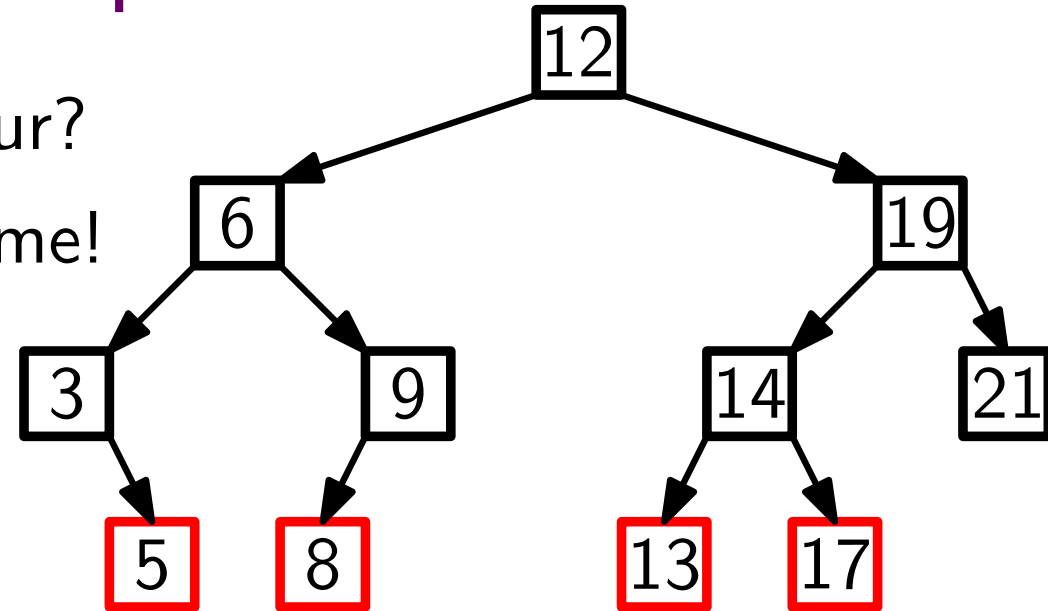
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe = $O(\log n)$



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

4. Implementiere! Laufzeit? Abschätzung bestmöglich? Nein!

Select(int i): $O(i + \log n)$ **Rank**(Node v): $O(\text{rank} + \log n)$

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



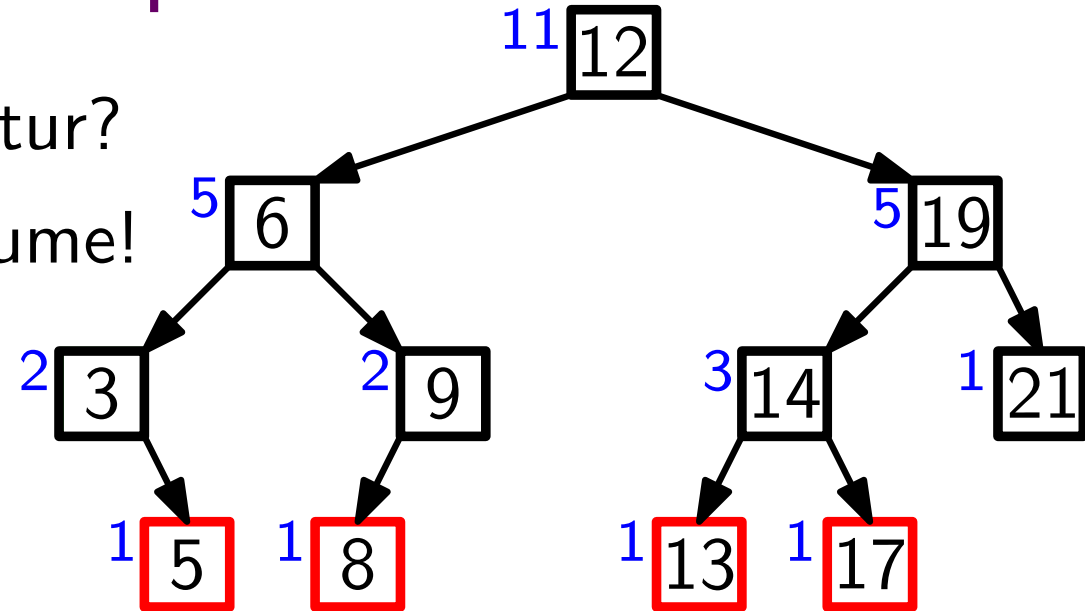
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

– balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

⇒ Baumhöhe = $O(\log n)$



2. Welche Extraintformation aufrechterhalten?

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

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

```

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)

```

```

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 ,

1.) *Initialisierung* *u-Rang von v*

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

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.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 .

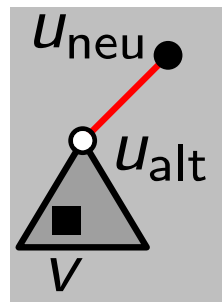
1.) *Initialisierung* ✓

u-Rang von v

2.) *Aufrechterhaltung* ✓

Annahme: Inv. galt zu Beginn der aktuellen Iteration.

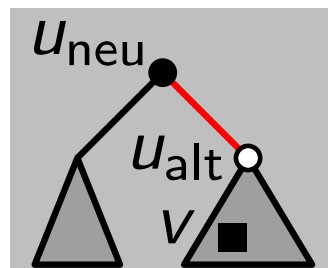
Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



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 ≠ 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)

```

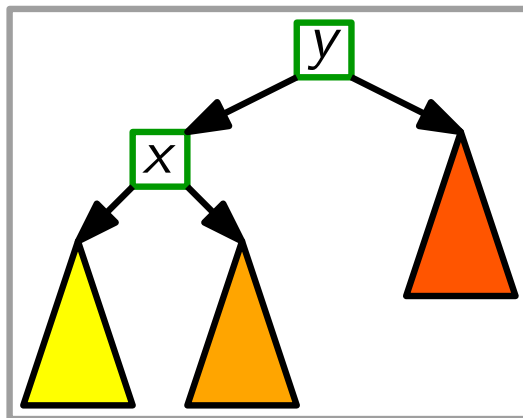
3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

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

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

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



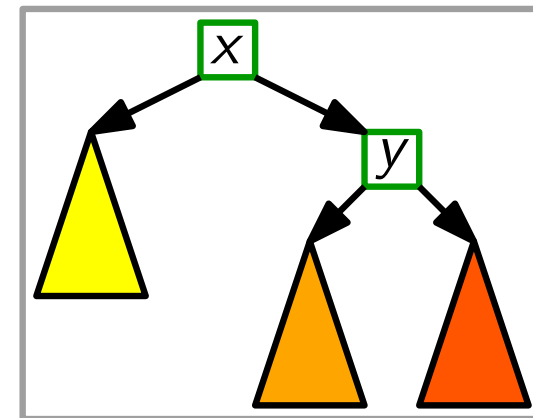
RightRotate(y)



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: $O(\log n)$

Ergebnis

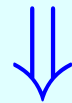
Satz. Das dynamische Auswahlproblem kann man so lösen, dass `Select()` und `Rank()` sowie alle gewöhnlichen Operationen für dynamische Mengen in einer Menge von n Elementen in $O(\log n)$ Zeit laufen.

Verallgemeinerung

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

Falls für jeden Knoten v gilt:

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



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]

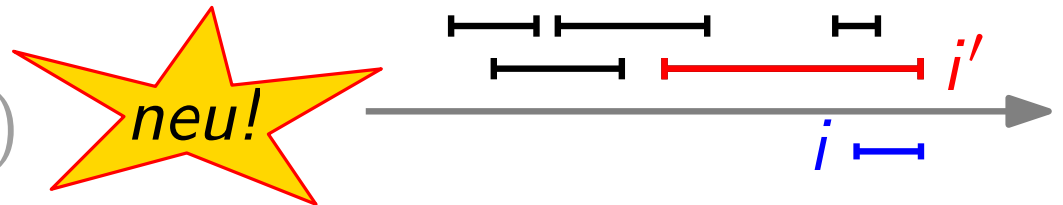
Noch ein Beispiel

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...