Algorithmen und Datenstrukturen

Vorlesung 15:
Augmentieren von Datenstrukturen

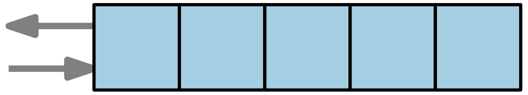
Alexander Wolff

Wintersemester 2025

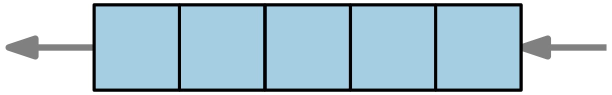
Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

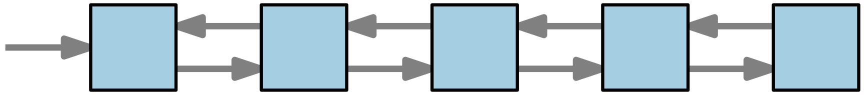
- Stapel



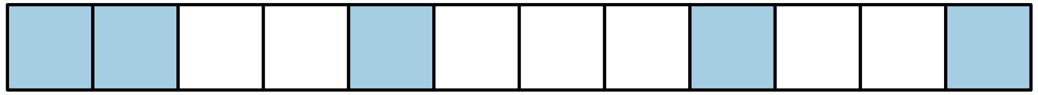
- Schlange



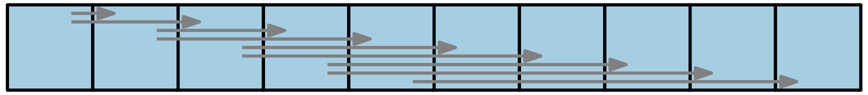
- doppelt verkettete Liste



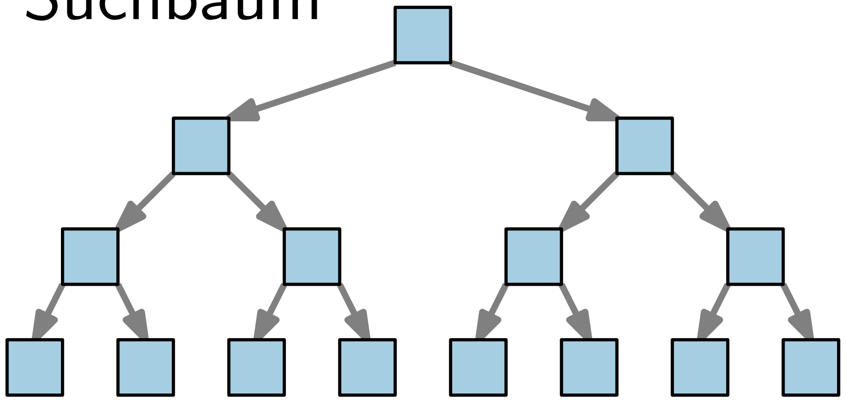
- Hashtabelle



- 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 Information hinzufügen und 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 ($size$)
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.

D.h. wir wollen zu jedem Zeitpunkt effizient

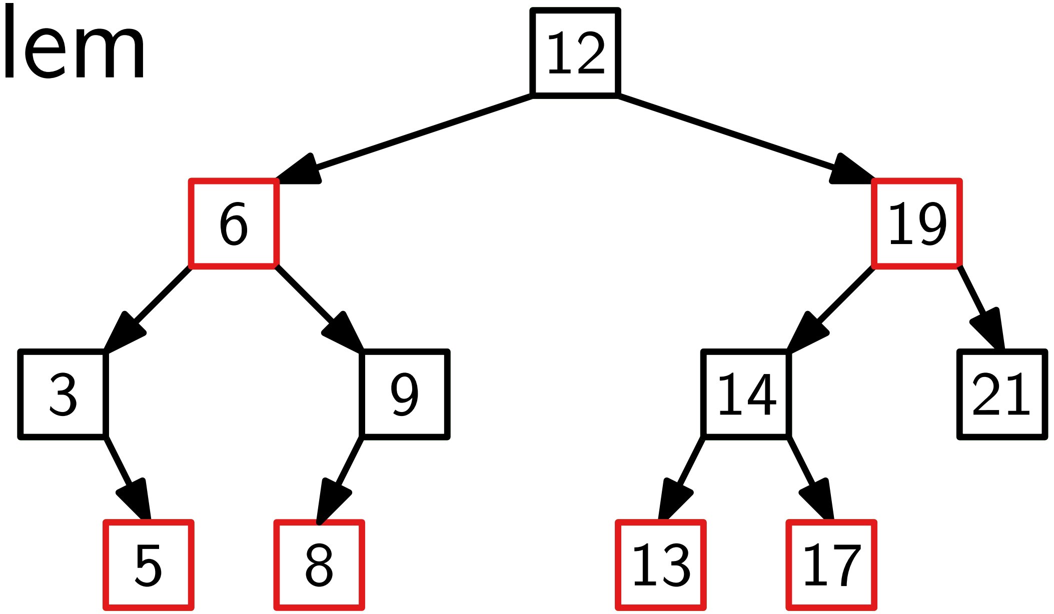
- das i.-kleinste Element (z.B. den Median): ptr $\textsc{Select}$(int i)
- den **Rang** eines Elements: int $\textsc{Rank}$(ptr x)

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?

- balancierte binäre Suchbäume
  z.B. Rot-Schwarz-Bäume
  $\Rightarrow$ 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?

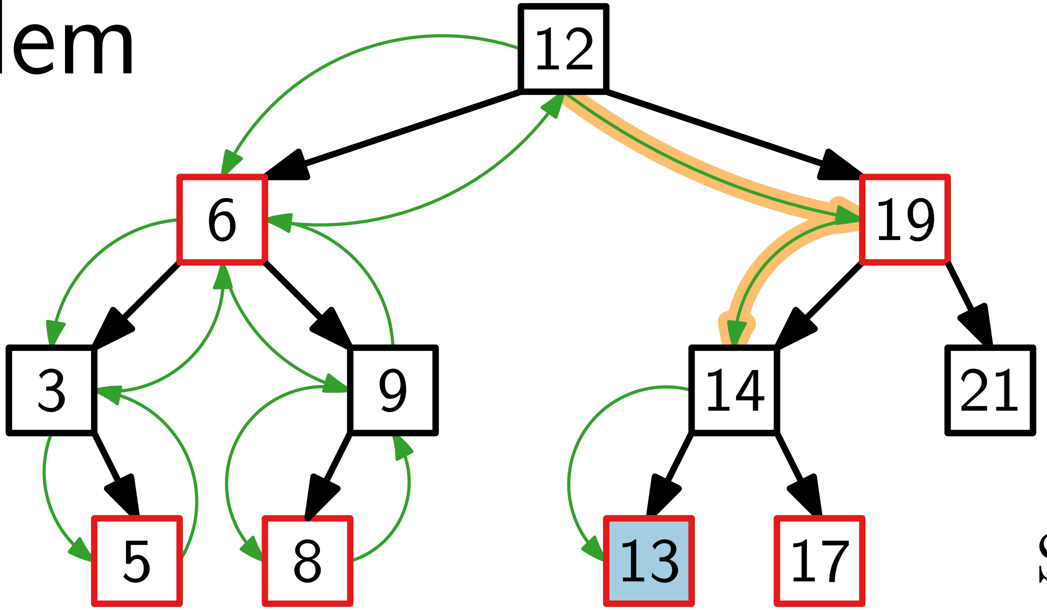
Node $\textsc{Select}$(int i): $\mathcal{O}(i \cdot h)$

$x = \textsc{Minimum}()$
while $x \neq nil$ **and** $i > 1$ **do**
  $x = \textsc{Successor}(x)$ $\mathcal{O}(h)$
  $i = i - 1$
return x

int $\textsc{Rank}$(Node x): $\mathcal{O}(rank \cdot h)$

$i = 0$
while $x \neq nil$ **do**
  $x = \textsc{Predecessor}(x)$ $\mathcal{O}(h)$
  $i = i + 1$
return i

Das dynamische Auswahlproblem



$\text{SELECT}(7)$

1. Welche Ausgangsdatenstruktur?
   - balancierte binäre Suchbäume
     z.B. Rot-Schwarz-Bäume
     $\Rightarrow$ Baumhöhe $h \in \mathcal{O}(\log n)$
2. Welche Extrainformation aufrechterhalten?
3. Aufwand zur Aufrechterhaltung?

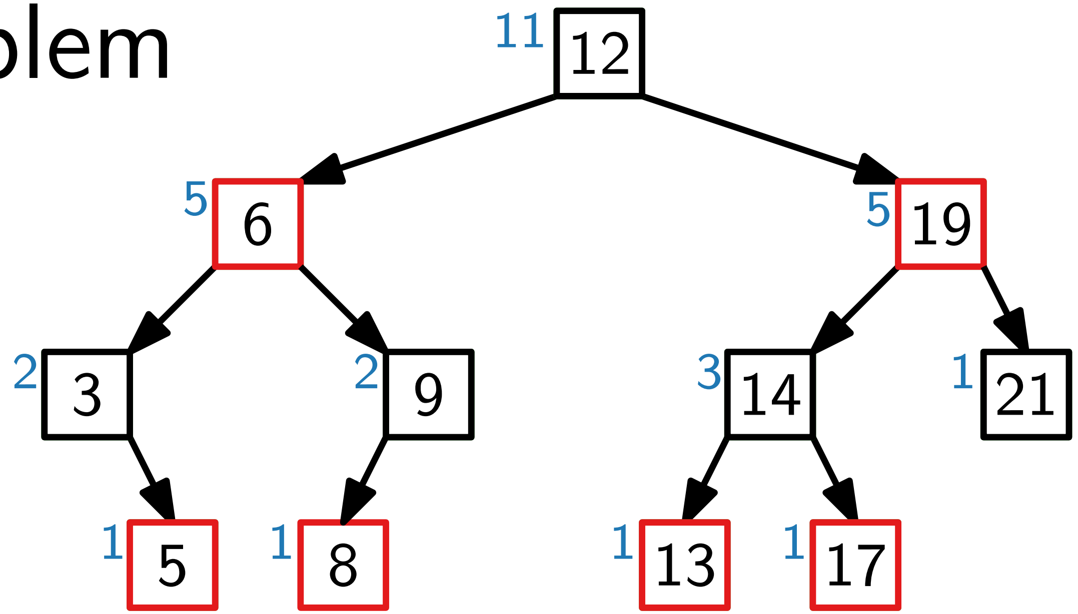
- gar keine?
- gar keiner

4. Implementiere neue Operationen! Laufzeit?

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

int RANK(Node x): $\mathcal{O}(rank+h)$
- $i = 0$
- **while** $x \neq nil$ **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
  $\Rightarrow$ 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? später...

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 ≠ 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 $\textsc{Rank}()$

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

```
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 $T.\mathit{nil}.\mathit{size} = 0$)

Korrektheit von Rank()

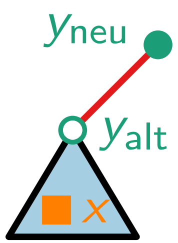
Invariante: Zu Beginn jeder Iteration der while-Schleife ist r der Rang von x im Teilbaum mit Wurzel y.

y-Rang(x)

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

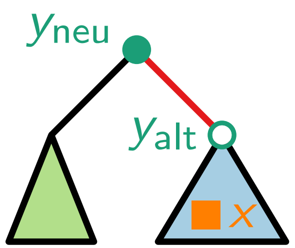
Annahme: Invariante galt zu Beginn der aktuellen Iteration.
Zu zeigen: Invariante gilt dann auch am Ende der aktuellen Iteration.



1. Fall: y war linkes Kind.

$\Rightarrow$ y-Rang von x bleibt gleich.

2. Fall: y war rechtes Kind.



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

```
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 $\textsc{Rank}()$

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

y-Rang(x)

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

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

Zusammenfassung:

Die Methode $\textsc{Rank}()$ liefert wie gewünscht den Rang des übergebenen Knoten.

```
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 $T.nil.size = 0$)

3. Aufwand zur Aufrechterhaltung der Extrainformation?

RBInsert() geht in zwei Phasen vor:

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

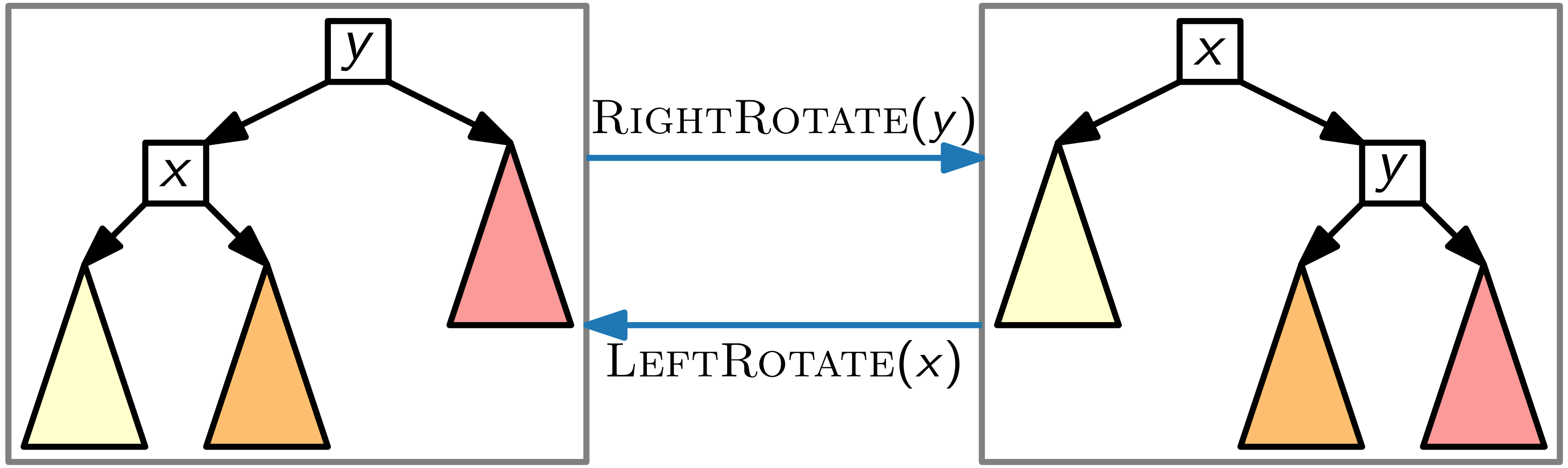
Phase I: Suche die Stelle, wo der neue Knoten z eingefügt wird.

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“.

Ergebnis

Satz. Das dynamische Auswahlproblem kann man so lösen, dass $\textsc{Select}()$ und $\textsc{Rank}()$ 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.

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 $f(v.\mathit{right})$) in konstanter Zeit berechnen.

$$\Downarrow$$

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 $f \equiv \mathit{size}$.

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]

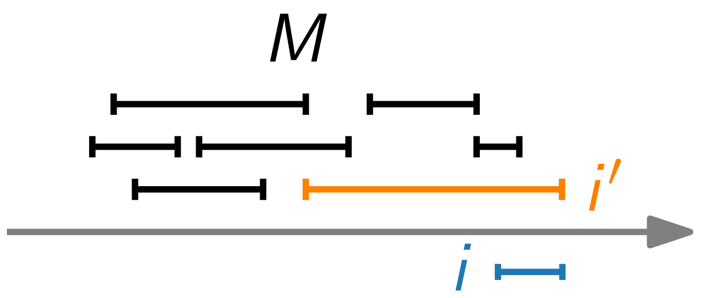
Noch ein Beispiel

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

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- ptr $\textsc{Insert}$ (Interval i)
- $\textsc{Delete}$(ptr x)
- ptr $\textsc{Search}$(Interval i) neu



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