Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 12 (3.6.2016)

Binäre Suchbäume IV

Fabian Kuhn

Algorithmen und Komplexität

UNI FREIBURG

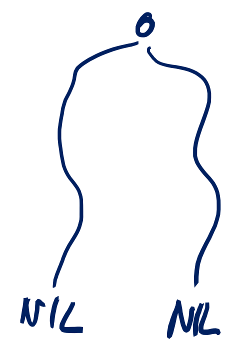
Rot-Schwarz-Bäume

Ziel: Binäre Suchbäume, welche **immer balanciert** sind

- balanciert, intuitiv: in jedem Teilbaum, links & rechts $\approx$ gleich gross
- balanciert, formal: Teilbaum mit k Knoten hat Tiefe $O(\log k)$

Rot-Schwarz-Bäume sind binäre Suchbäume, für die gilt:

1) Alle Knoten sind **rot** oder **schwarz**
2) Wurzel ist schwarz   Sentinel (NIL)
3) Blätter (= NIL-Knoten) sind schwarz
4) Rote Knoten haben zwei schwarze Kinder
5) Von jedem Knoten v aus, haben alle (direkten) Pfade zu Blättern (NIL) im Teilbaum von v die gleiche Anzahl schwarze Knoten

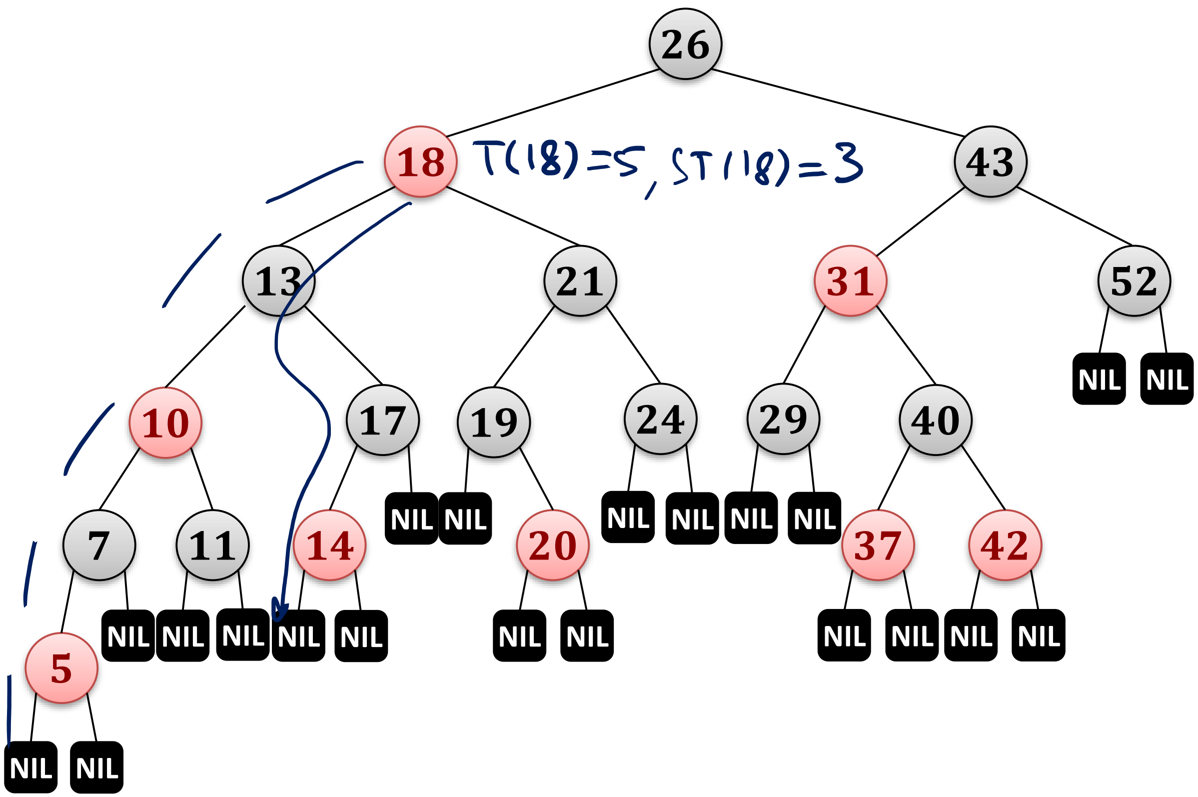


Tiefe / Schwarz-Tiefe

Definition: Die **Tiefe (T)** eines Knoten v ist die maximale Länge eines direkten Pfades von v zu einem Blatt (NIL).

Definition: Die **Schwarz-Tiefe (ST)** eines Knoten v ist die Anzahl schwarzer Knoten auf jedem direkten Pfad von v zu einem Blatt (NIL)

- Der Knoten v wird dabei nicht gezählt, das Blatt (NIL, falls $\neq v$) jedoch schon!



Tiefe eins Rot-Schwarz-Baumes

Theorem:

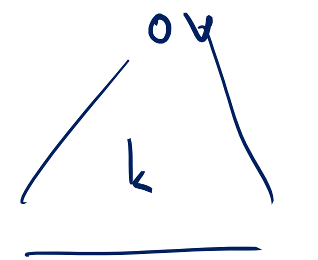
Die **Tiefe** eines Rot-Schwarz-Baumes ist $\boxed{\leq \mathbf{2\log_2(n+1)}.}$

Beweis:

- Anzahl innerer Knoten : n (alle ausser den NIL-Knoten)
- Lemma:

$$n \geq 2^{ST(root)} - 1 \qquad \boxed{2^{ST(v)} - 1}$$

$ST(v) \geq T(v)/2$

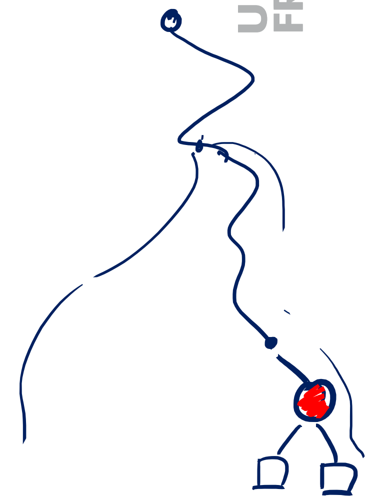


$2\log_2(k+1)$

Rot-Schwarz-Bäume: Einfügen

insert(x):

1. Einfügen wie üblich, neu eingefügter Knoten ist **rot**



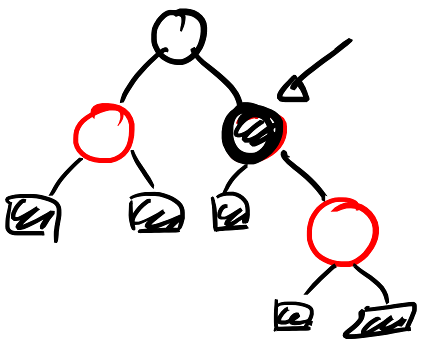
key color parent left right

```
if root == NIL then
    root = new Node(x,red,NIL,NIL,NIL)
else
    v = root;
    while v != NIL and v.key1 != x do
        if v.key1 > v.x then
            if v.left == NIL then
                w = new Node(x,red,v,NIL,NIL); v.left = w
            v = v.left
        else
            if v.right == NIL then
                w = new Node(x,red,v,NIL,NIL); v.right = w
            v = v.right
```

Rot-Schwarz-Bäume: Einfügen

Rot-Schwarz-Baum Bedingungen nach dem Einfügen

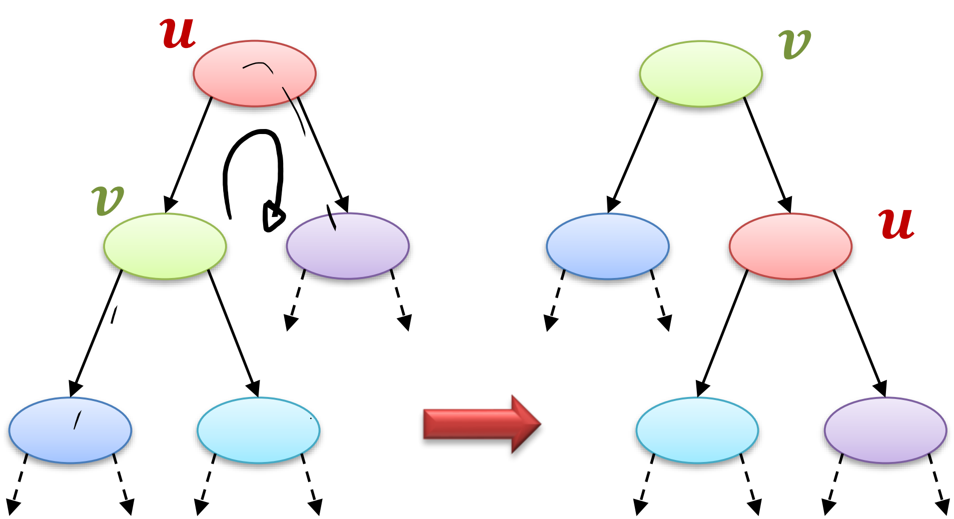
1. Alle Knoten sind rot oder schwarz
2. Die Wurzel ist schwarz
3. Die Blätter (NIL) sind schwarz
4. Rote Knoten haben zwei schwarze Kinder
5. Von jedem Knoten v haben alle direkten Pfade zu Blättern gleich viele schwarze Knoten



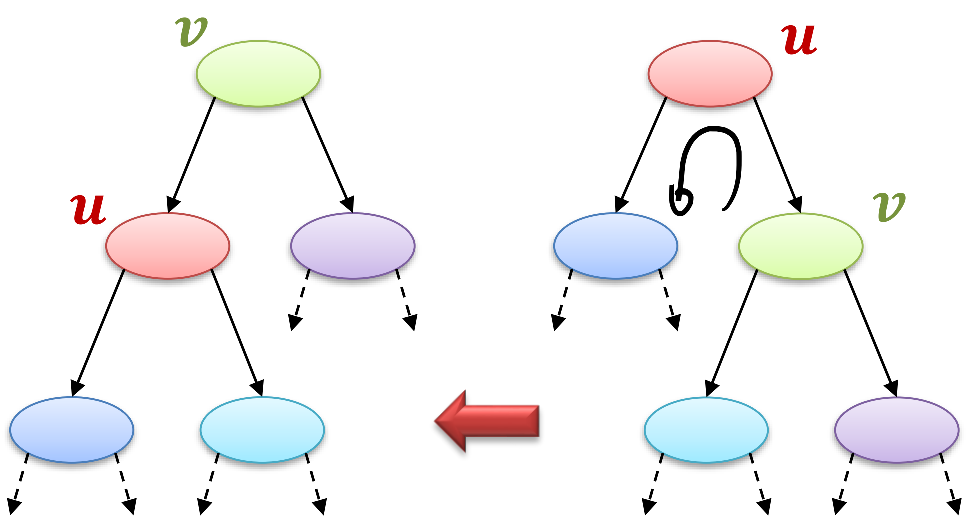
- Falls v (eingefügter Knoten) nicht die Wurzel ist oder $v.\text{parent}$ schwarz ist, sind alle Bedingungen erfüllt
- Falls v die Wurzel ist, kann man v einfach schwarz einfügen
- Falls $v.\text{parent}$ rot ist, müssen wir den Baum anpassen
  - so, dass 1, 3 und 5 immer erfüllt sind
  - dabei kann die Wurzel auch rot werden…

Erinnerung: Rotationen

Rechtsrotation:



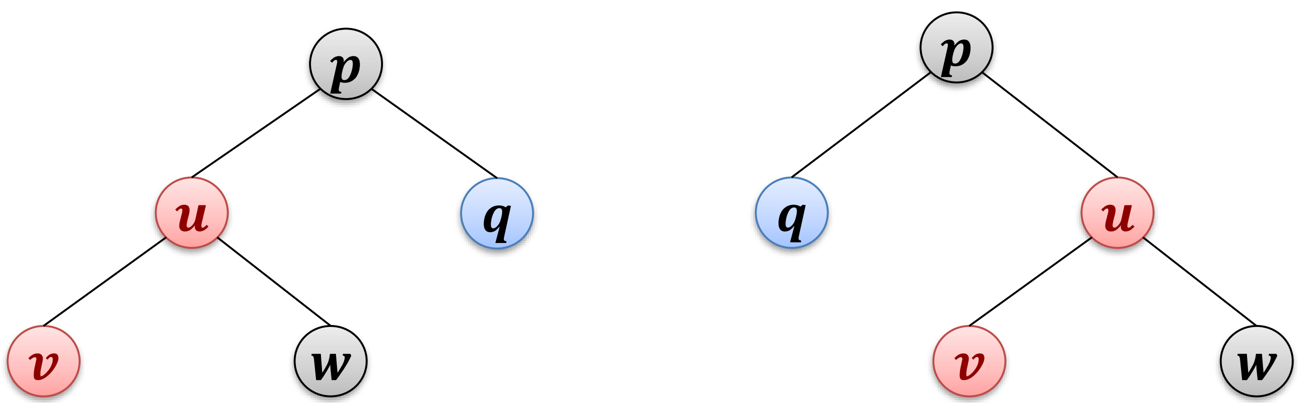
Linksrotation:



Rot-Schwarz-Bäume: Einfügen

Baum anpassen nach dem Einfügen:

- Annahmen:
  - v ist rot,
  - $u \coloneqq v.\text{parent}$ ist rot (sonst sind wir fertig)
  - v ist linkes Kind von u (anderer Fall symmetrisch)
  - v's Geschwisterknoten w (rechtes Kind von u) ist schwarz
  - Alle roten Knoten ausser u haben 2 schwarze Kinder

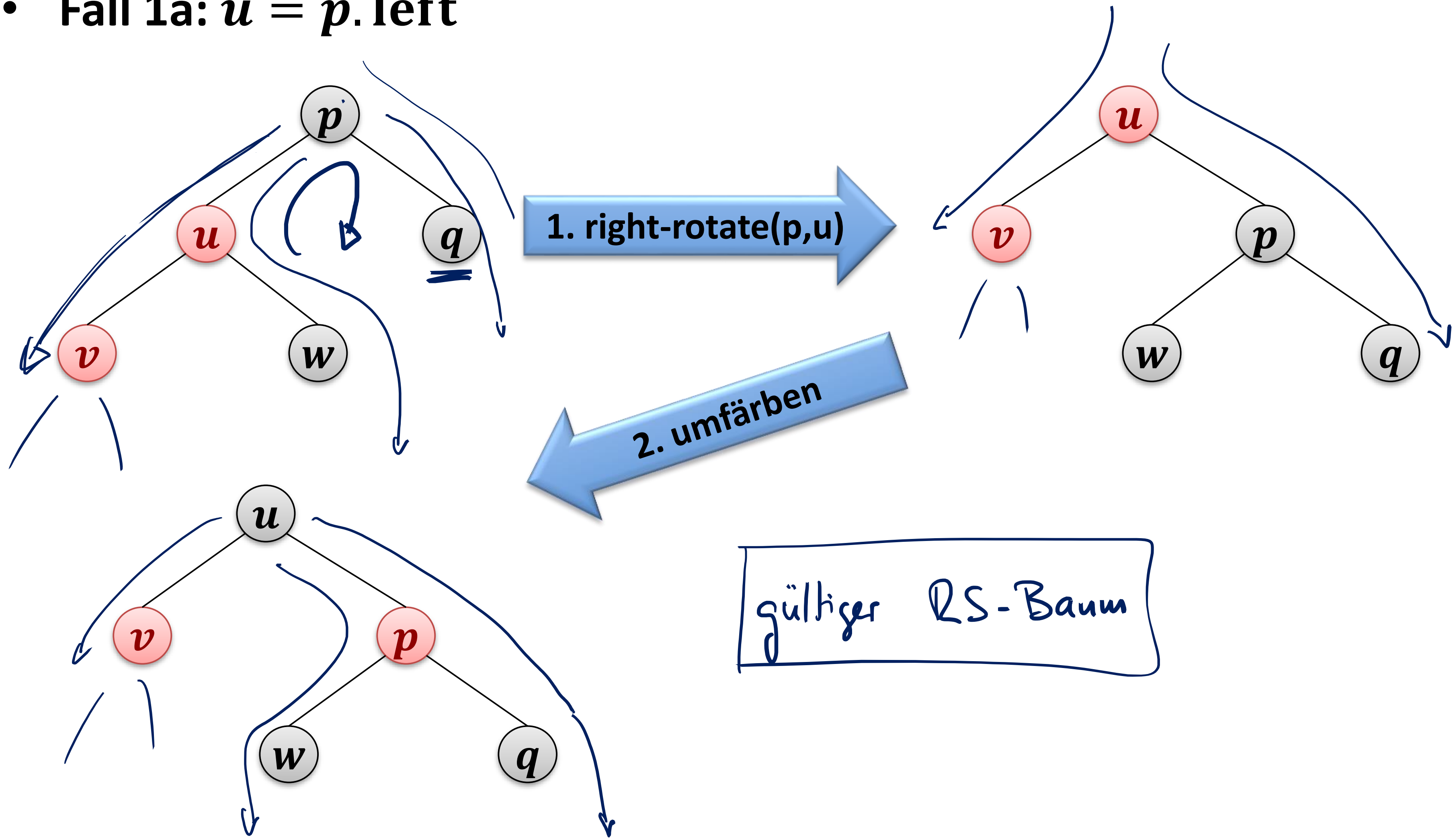


- Fallunterscheidung anhand von **Farbe von $\boldsymbol{q}$ (Geschwister von $\boldsymbol{u}$)** und anhand von $\boldsymbol{u = p.\textbf{left}}$ **oder** $\boldsymbol{u = p.\textbf{right}}$

Rot-Schwarz-Bäume: Einfügen

Fall 1: Geschwisterknoten q von u ist schwarz

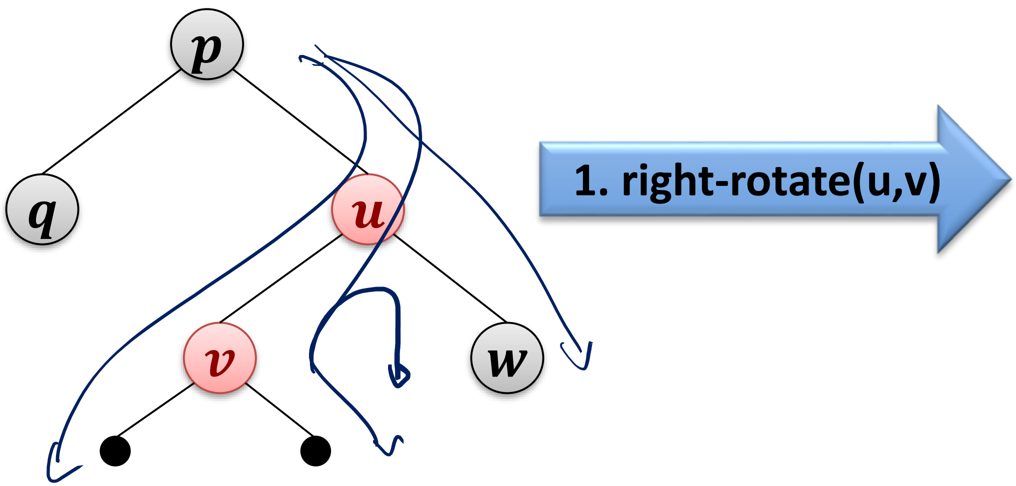
- **Fall 1a: $u = p.\text{left}$**

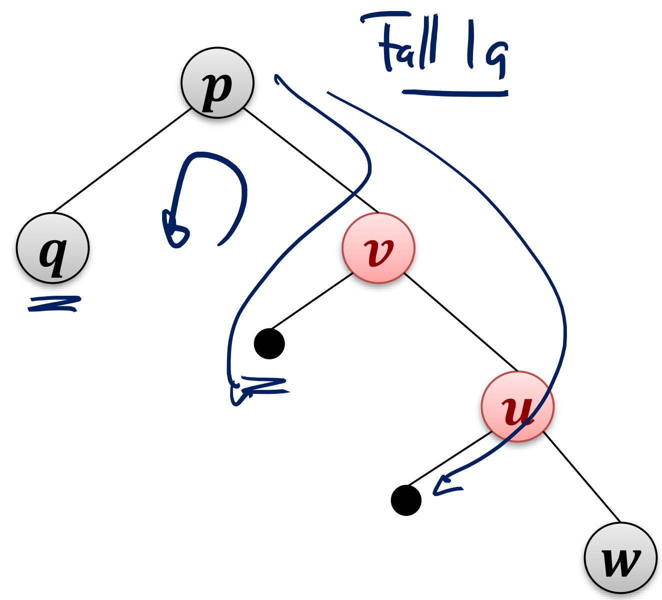


Rot-Schwarz-Bäume: Einfügen

Fall 1: Geschwisterknoten q von u ist schwarz

- **Fall 1b: $u = p.\mathrm{right}$**





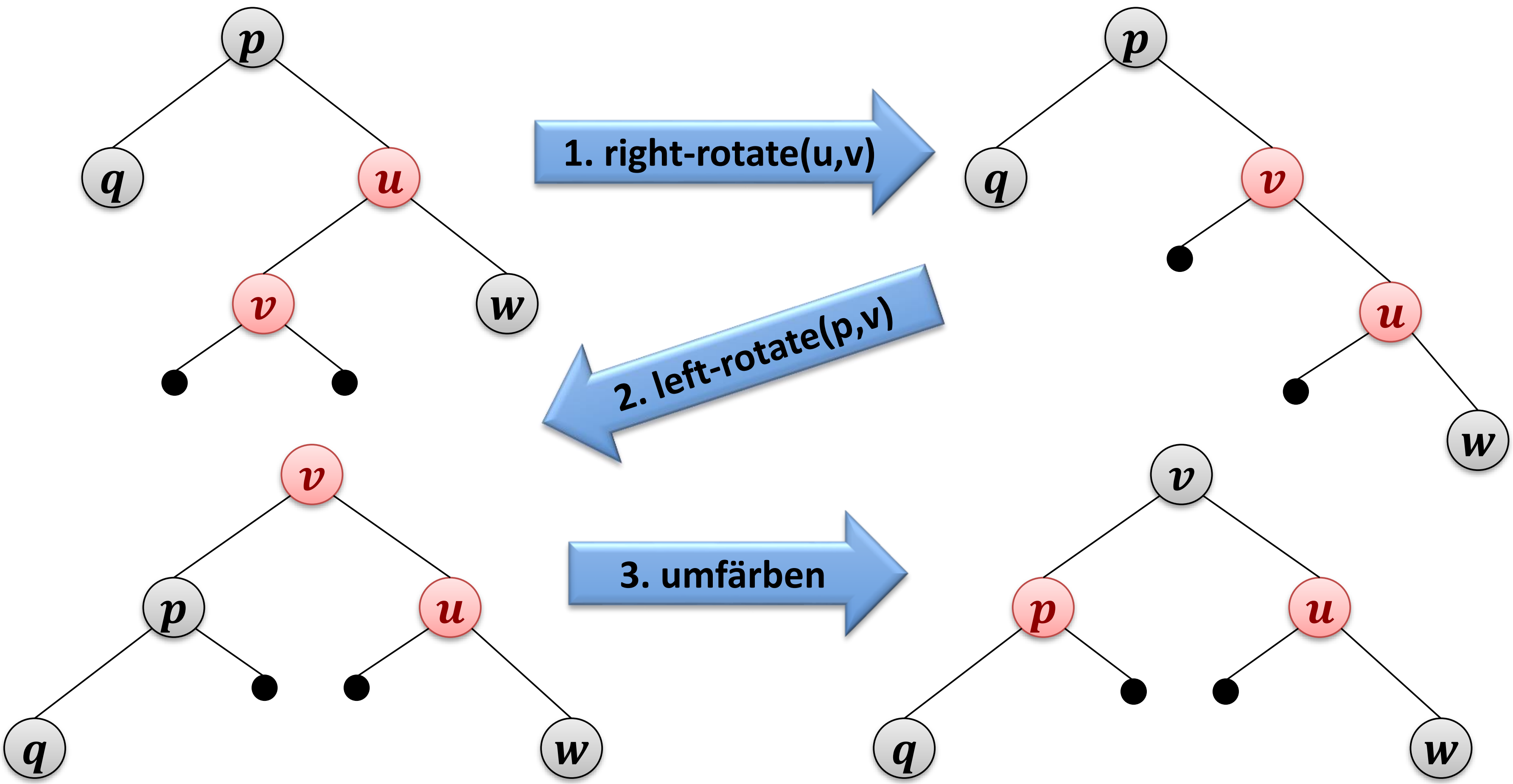
- Nach Rotation:
  - symmetrisch zu Fall 1a
  - u, v sind rot, Geschwister q is schwarz
  - u ist rechtes Kind von v, v ist rechtes Kind von p
  - Aulösen durch

    left-rotate(p,v) und umfärben

Rot-Schwarz-Bäume: Einfügen

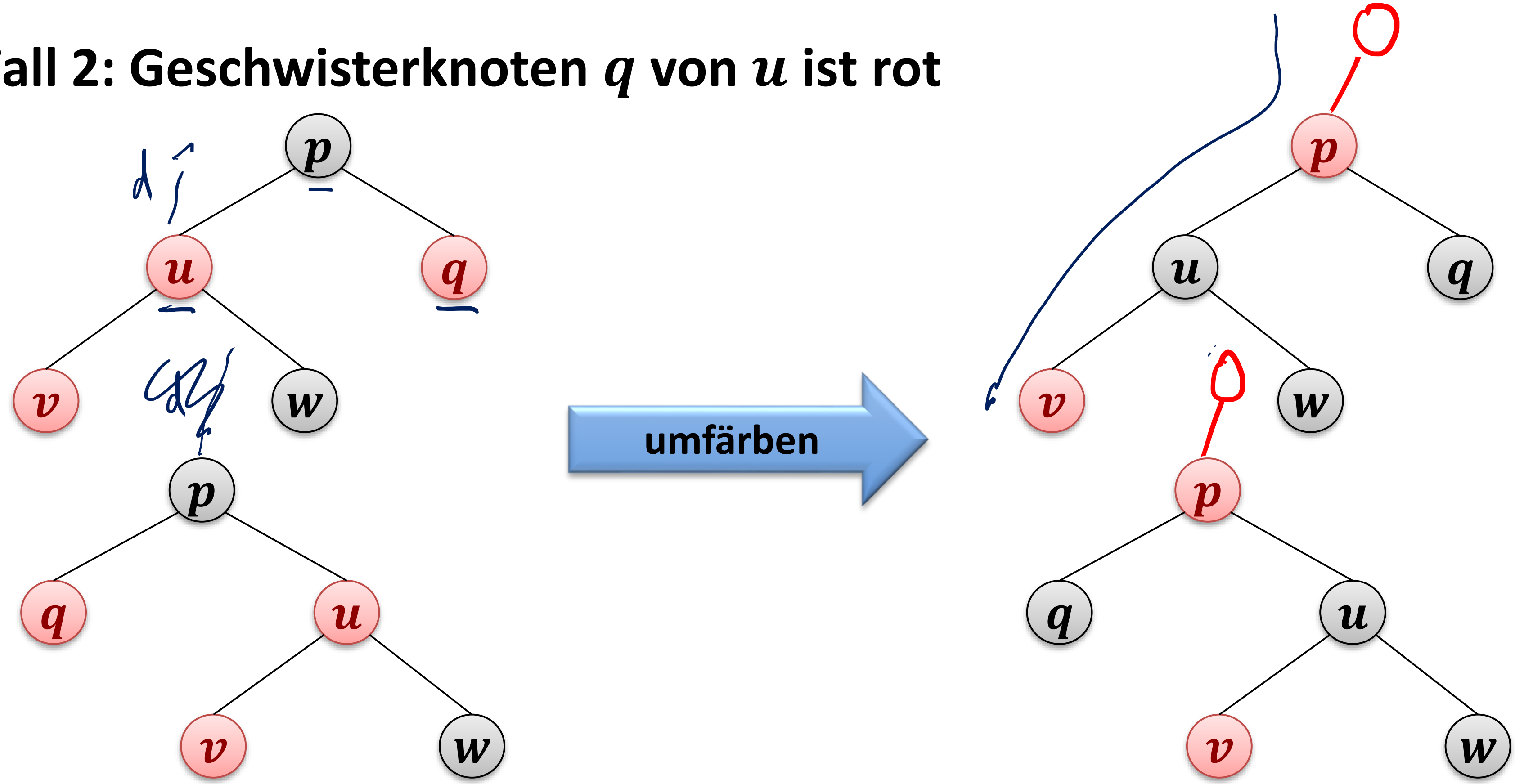
Fall 1: Geschwisterknoten q von u ist schwarz

- **Fall 1b: $u = p.\text{right}$**



Rot-Schwarz-Bäume: Einfügen

Fall 2: Geschwisterknoten q von u ist rot



- Falls p. parent schwarz ist, sind wir fertig
  - das ist auch der Fall, falls $p ==$ root (dann noch root. color $:=$ black)
- Sonst sind wir im gleichen Fall, wie am Anfang
  - aber näher an der Wurzel!

Rot-Schwarz-Bäume: Einfügen

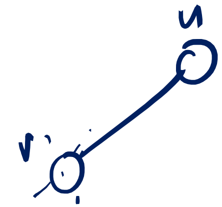
1. Füge neuen Schlüssel normal ein
   - Knotenfarbe des neuen Knoten ist rot
2. Solange man in Fall 2 ist, färbe um
   - Fall 2: roter Knoten v mit rotem Parent-Knoten u
   - Geschwisterknoten von u ist auch rot
3. Sobald nicht mehr in Fall 2
   - Falls es ein Rot-Schwarz-Baum ist, sind wir fertig
   - Falls die Wurzel rot ist, muss die Wurzel schwarz gefärbt werden
   - Ansonsten ist man in Fall 1a oder 1b (oder symmetrisch) und kann mit Hilfe von höchstens 2 Rotationen und Umfärben von 2 Knoten einen Rot-Schwarz-Baum erhalten

- **Laufzeit:** $O(\text{Baumtiefe}) = O(\log n)$

Rot-Schwarz-Bäume: Löschen

1. Finde wie üblich einen Knoten v, welcher gelöscht wird
   - Knoten v hat höchstens ein Nicht-NIL-Kind!

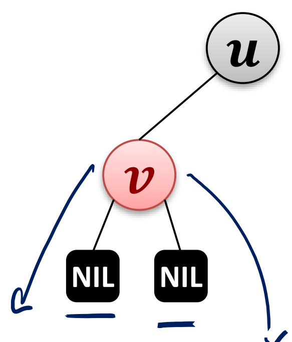
Fallunterscheidung (Farbe von v und $\overbrace{v.\mathbf{parent}}^{u}$)



Annahme: v ist linkes Kind von u (sonst symmetrisch)

Fall 1: Knoten v ist rot

- Da v mind. 1 NIL-Kind haben muss und es ein Rot-Schwarz-Baum ist, muss v 2 NIL-Kinder haben

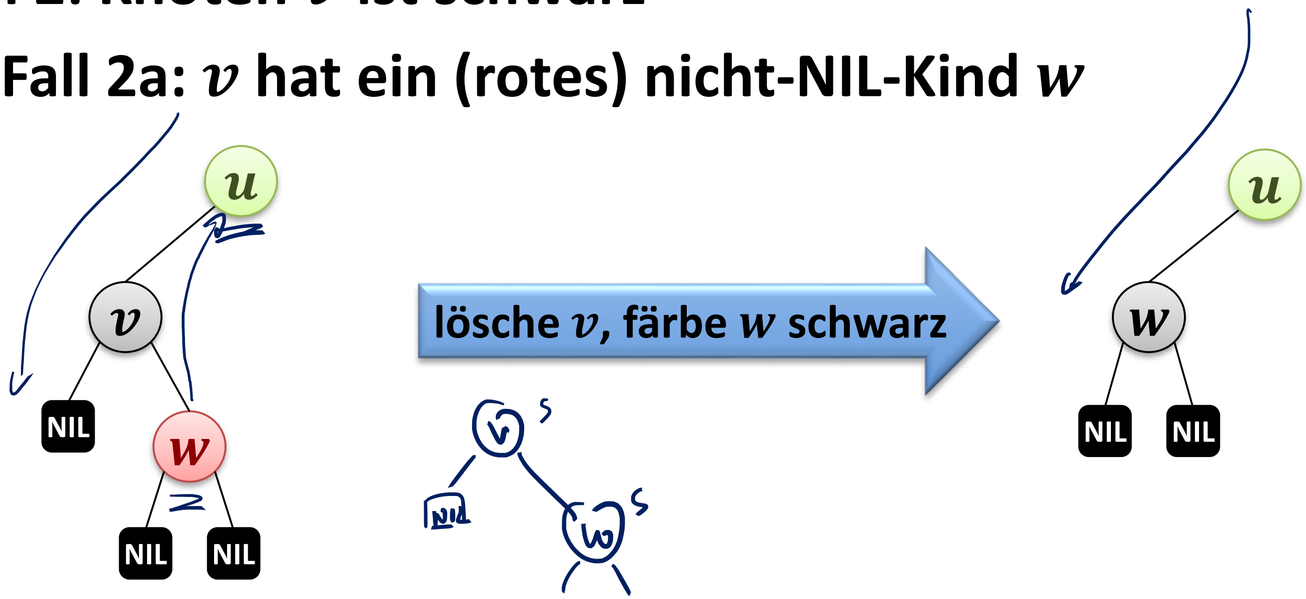


- v kann einfach gelöscht werden
  - Der Baum bleibt ein Rot-Schwarz-Baum

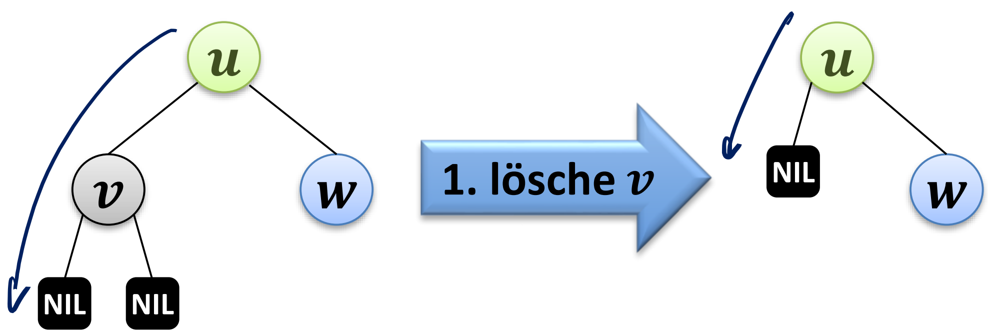
Rot-Schwarz-Bäume: Löschen

Fall 2: Knoten v ist schwarz

- **Fall 2a: v hat ein (rotes) nicht-NIL-Kind w**



- **Fall 2b: v hat nur NIL-Kinder**



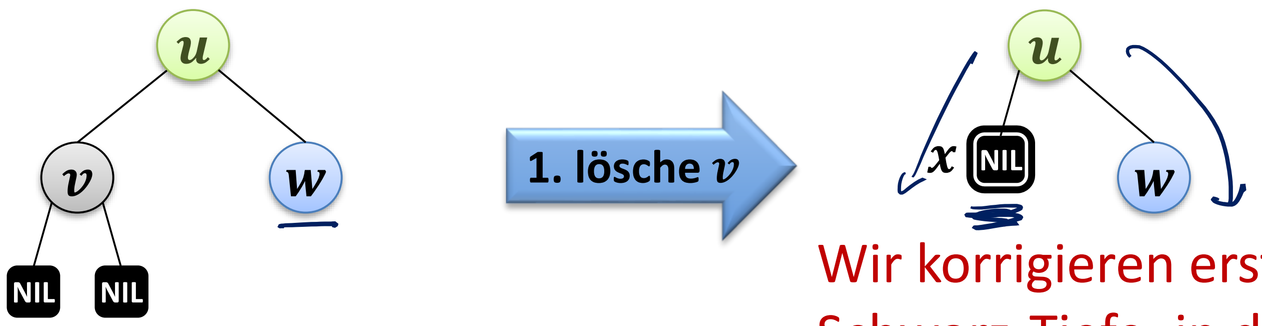
Knoten u hat jetzt nach links nur noch Schwarz-Tiefe 1 (statt 2)

→ Wir müssen den Baum anpassen

Rot-Schwarz-Bäume: Löschen

Problemfall:

- **Knoten v hat nur NIL-Kinder**



Wir korrigieren erstmal die Schwarz-Tiefe, in dem wir doppelt schwarz färben

- **Ziel**: Wir möchten das zusätzliche "Schwarz" den Baum hochbringen bis wir es entweder bei einem roten Knoten abladen können oder bis wir die Wurzel erreichen (und damit kein Problem mehr haben).
- **Fallunterscheidung**: Farbe von w und der Kinder von w
  - Beobachtung: w kann nicht NIL sein (wegen Schwarz-Tiefe)!

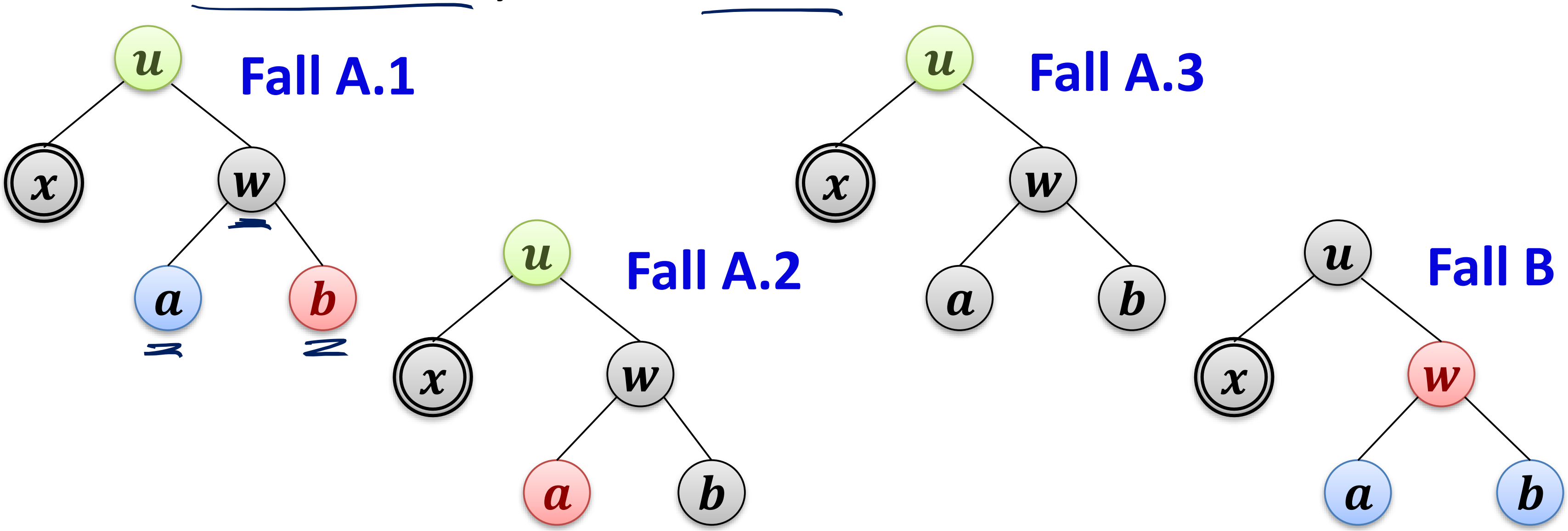
Rot-Schwarz-Bäume: Löschen

Annahme:

- Doppelt schwarzer Knoten x
- Parent u hat beliebige Farbe (markiert als grün)
- x ist linkes Kind von u (rechtes Kind: symmetrisch)
- Geschwisterknoten von x (rechtes Kind von u) ist w

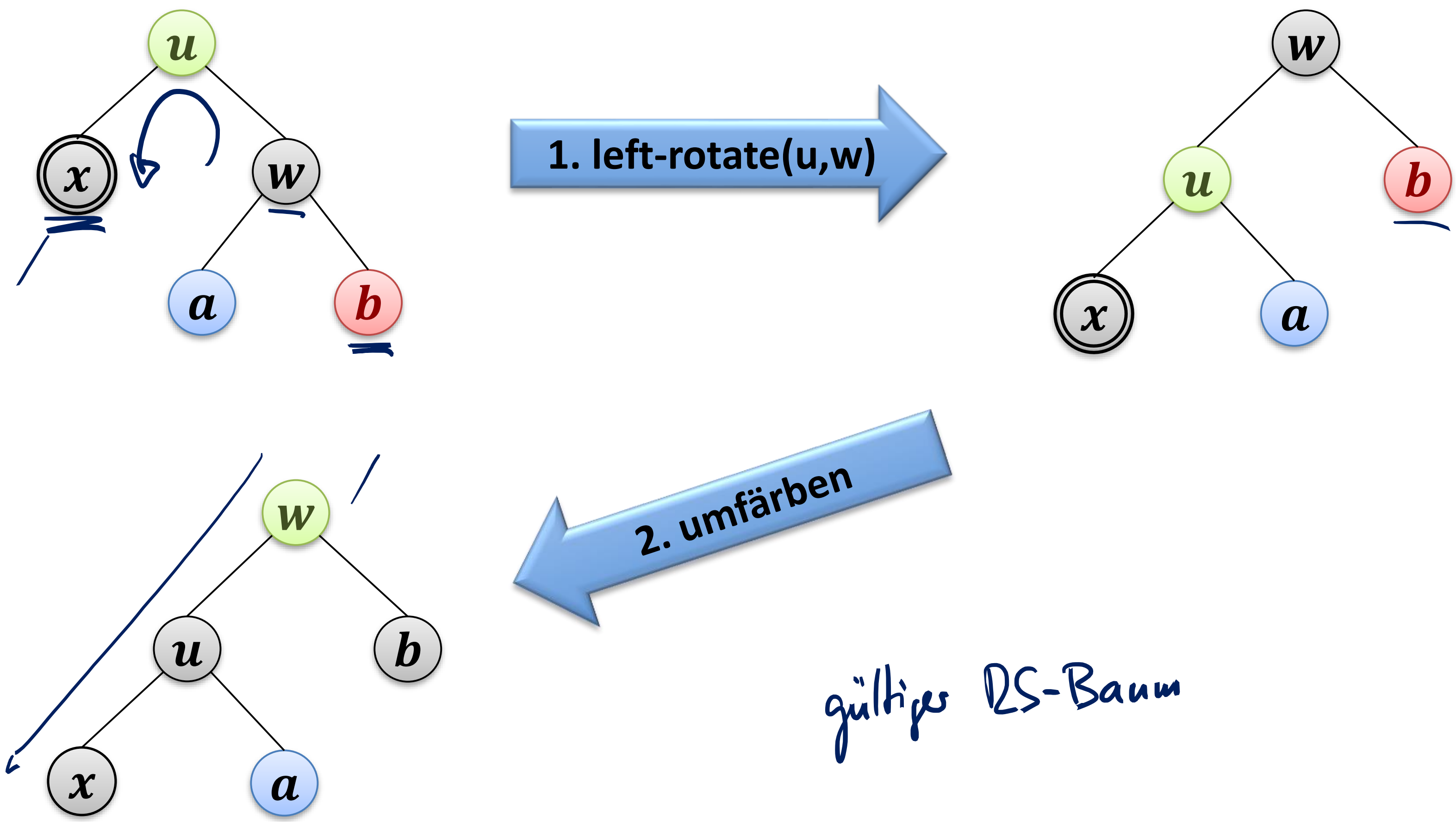
Fallunterscheidung:

- **Fall A: w ist schwarz, Fall B: w ist rot**



Rot-Schwarz-Bäume: Löschen

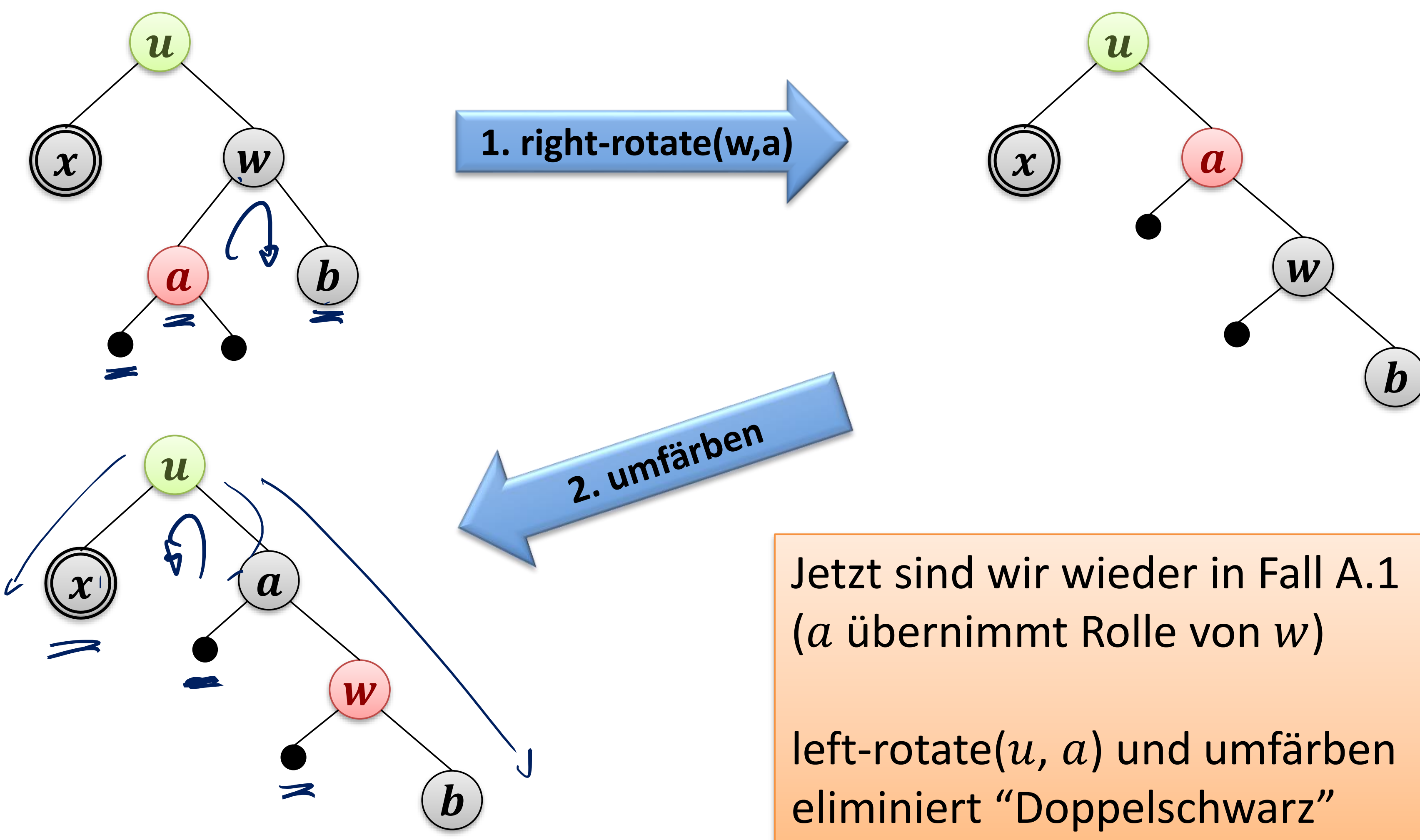
Fall A.1: w ist schwarz, rechtes Kind von w ist rot



gültiger RS-Baum

Rot-Schwarz-Bäume: Löschen

Fall A.2: w ist schwarz, l. Kind von w ist rot, r. Kind ist schwarz



Jetzt sind wir wieder in Fall A.1 (a übernimmt Rolle von w)

left-rotate(u, a) und umfärben eliminiert "Doppelschwarz"

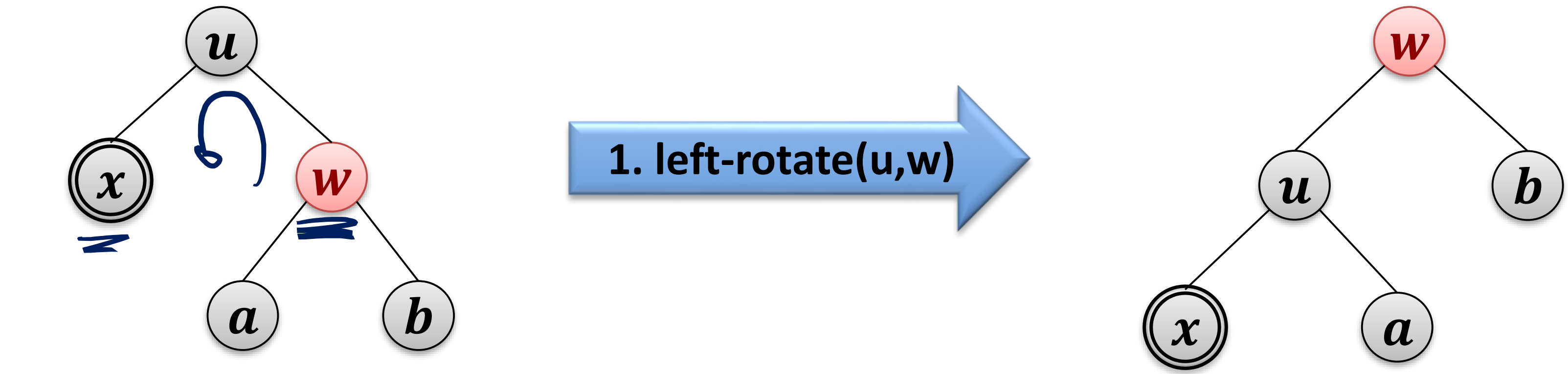
Rot-Schwarz-Bäume: Löschen

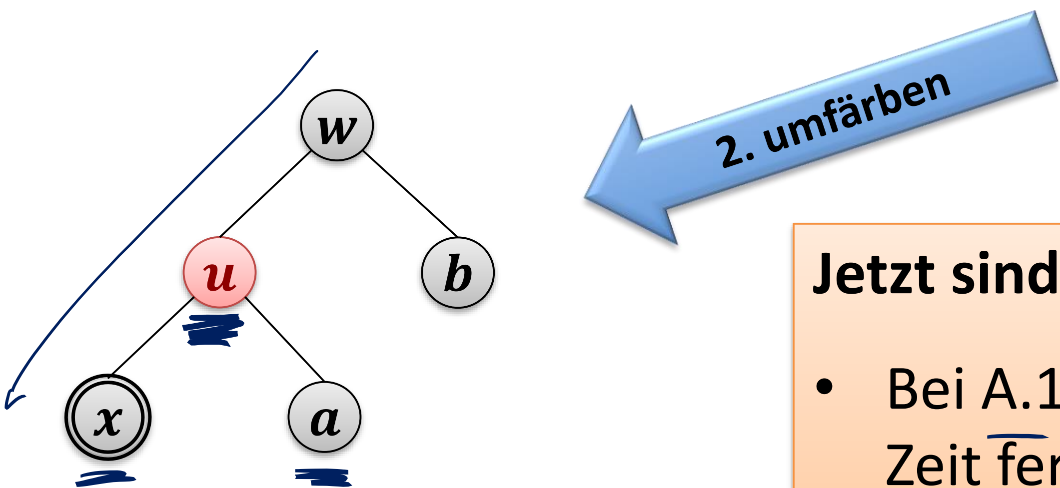
Fall A.3: w ist schwarz, beide Kinder von w sind schwarz

- Das zusätzliche “Schwarz” wandert eins nach oben
- Falls u rot ist, kann man u jetzt einfach schwarz färben
- Knoten u übernimmt sonst die Rolle von x und kann wieder in einem der Fälle A.1, A.2, A.3 oder B (siehe nächste Folie) sein.
- Fall A.3 kann höchstens $O(\log n)$ oft vorkommen
  - Falls $u == \text{root}$, können wir das zusätzliche “Schwarz” einfach entfernen
- Bei Fall A.1 und A.2 sind wir direkt fertig

Rot-Schwarz-Bäume: Löschen

Fall B: w ist rot





Jetzt sind wir in Fall A.1, A.2 oder A.3

- Bei A.1 oder A.2 sind wir in $O(1)$ Zeit fertig
- Bei A.3 sind wir auch in $O(1)$ Zeit fertig, weil u jetzt rot ist!

Rot-Schwarz-Bäume: Löschen

1. Wie üblich
   - Finde Knoten v mit mind. 1 NIL-Kind, welcher gelöscht werden kann
   - v ist evtl. Vorgänger/Nachfolger von Knoten mit zu löschendem Schlüssel
2. Falls der gelöschte Knoten v schwarz ist, muss man korrigieren
   - Es hat einen schwarzen Knoten x mit zusätzlichem “Schwarz”
3. Mögliche Fälle: A.1, A.2, A.3, B
   - Fall A.1: Mit 1 Rotation und Umfärben von $O(1)$ Knoten fertig
   - Fall A.2: Mit 1 Rotation und Umfärben von $O(1)$ Knoten in Fall A.1
   - Fall A.3: Falls $x.\mathrm{parent}$ rot ist, mit Umfärben von $O(1)$ Knoten fertig, falls $x.\mathrm{parent}$ schwarz ist, wandert zusätzliches “Schwarz” Richtung Wurzel und man ist ist wieder in A.1, A.2, A.3 oder B
   - Fall B : 1 Rotation und Umfärben von $O(1)$ Knoten gibt A.1, A.2 oder A.3, Falls A.3, dann ist $x.\mathrm{parent}$ rot

- **Laufzeit:** $O(\text{Baumtiefe}) = O(\log n)$

AVL Bäume

- Siehe z.B. Buch von Ottmann/Widmayer
- Direkte Alternative zu Rot-Schwarz-Bäumen

- AVL Bäume sind binäre Suchbäume, bei welchem für jeden Knoten v gilt, dass

$$\left|T(v.\mathbf{left}) - T(v.\mathbf{right})\right| \leq 1$$

- Anstatt einer Farbe (rot/schwarz) merkt man sich die Tiefe jedes Teilbaums
- AVL Bäume haben auch immer Tiefe $O(\log n)$
  - Sogar mit etwas besserer Konstante als Rot-Schwarz-Bäume
- AVL-Bedingung kann bei insert/delete jeweils mit $O(\log n)$ Rotationen wieder hergestellt werden
- Vergleich zu Rot-Schwarzbäumen
  - Suche ist in AVL Bäumen etwas schneller
  - Einfügen / Löschen ist in AVL Bäumen etwas langsamer

(a, b)-Bäume

- Siehe z.B. Vorlesung vom letzten Jahr
- Parameter $a \geq 2$ und $b \geq 2a - 1$
- Elemente/Schlüssel sind nur in den Blättern gespeichert
- Alle Blätter sind in der gleichen Tiefe
- Falls die Wurzel kein Blatt ist, hat sie zwischen 2 und b Kinder
- Alle anderen inneren Knoten haben zwischen a und b Kinder
  - Ein (a, b)-Baum ist also kein Binärbaum!

Ähnlich: $\boldsymbol{B}$-Bäume

- Da werden in den inneren Knoten Schlüssel gespeichert
- Für grosse a, b braucht man etwas mehr Speicher als bei BST
  - Da man meistens gleich für b Elemente Platz macht
- Dafür sind die Bäume viel flacher
- Speziell gut z.B. für Dateisysteme (Zugriff sehr teuer)

Weitere Alternativen

AA-Trees:

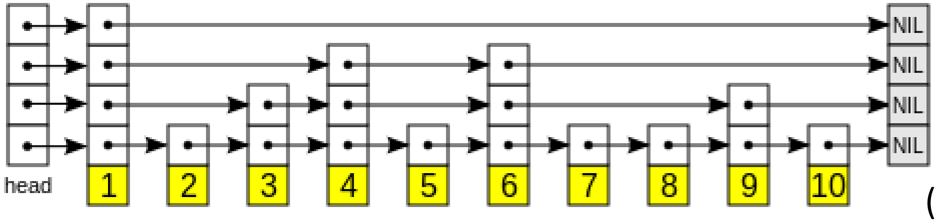
- ähnlich wie Rot-Schwarz-Bäume (nur rechte Kinder können rot sein)

Splay Trees:

- Binäre Suchbaum mit zusätzlichen guten Eigenschaften
  - Elemente, auf welche kürzlich zugegriffen wurde, sind weiter oben
  - Gut, falls mehrere Knoten den gleichen Schlüssel haben können
  - Allerdings nicht streng balanciert

Skip Lists:

- Verkettete Listen mit zusätzlichen Abkürzungen
  - kein balancierter Suchbaum, hat aber ähnliche Eigenschaften



(picture from wikipedia)

Praktische Übungsaufgabe

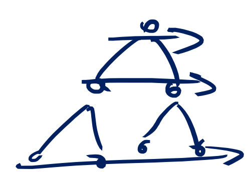
Programmieren eines Treaps:

- Wir stellen eine Binary Search Tree Implementierung zur Verfügung, welche Sie dann anpassen sollen
  - Sie können entweder die Klasse erweitern oder einfach den gegebenen Programmcode ergänzen, um einen Treap zu erhalten
- Operationen: insert, delete, find (+ triviale Op.)

- Zusätzlich:
  - avgDepth (gibt durchschnittliche Knotentiefe zurück)
  - BFS range query (gibt alle Schlüssel in einem Bereich [a,b] in BFS-Reihenfolge aus)

$\boxed{a \ldots b}$

O(Tiefe + #Elm in Range [a,b])

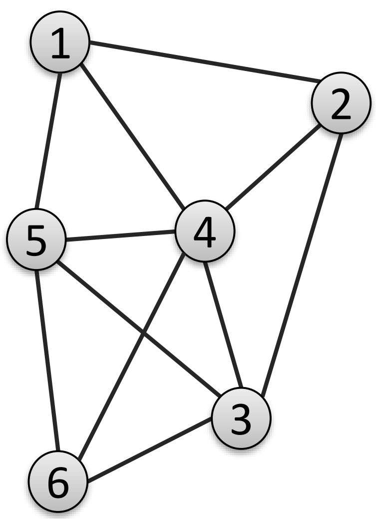


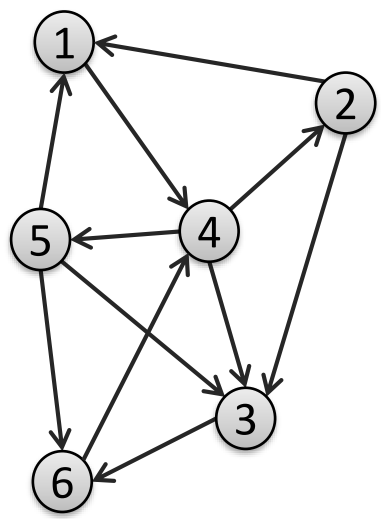
Graph

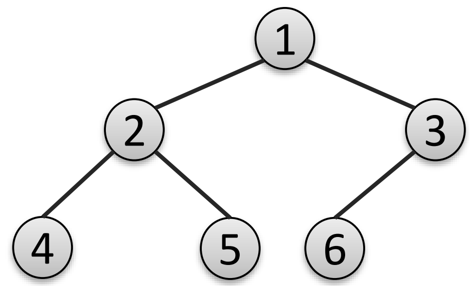
Graph $G = (V, E)$:

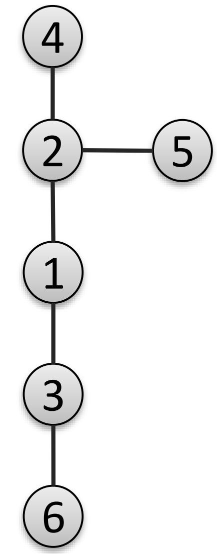
- Knotenmenge V und Kantenmenge E

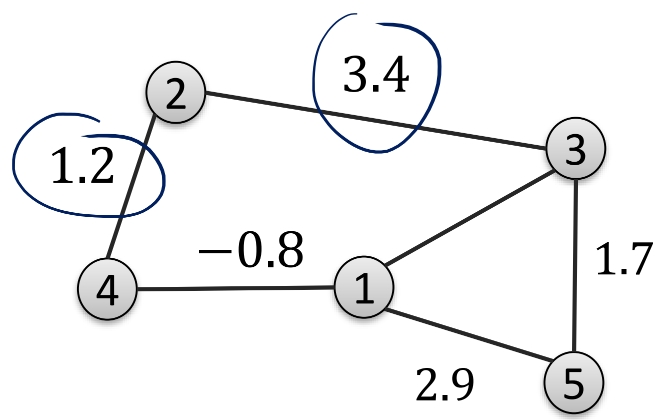
Beispiele:











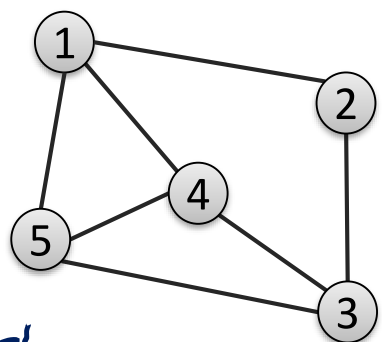
Graph: Notation

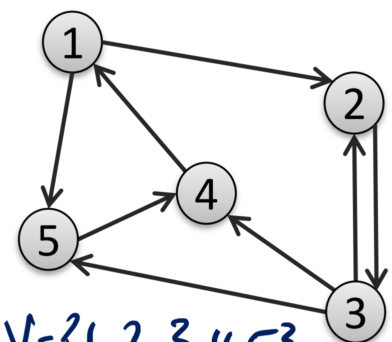
Knotenmenge V, typischerweise $n := |V|$

Kantenmenge E, typischerweise $m := |E|$ $\{u,v\}$

- ungerichteter Graph: $E \subseteq \{\{u,v\} : u,v \in V\}$
- gerichteter Graph: $E \subseteq V \times V$ (u,v) $u \to v$

Beispiele:





$V = \{1,2,3,4,5\}$

$V = \{1,2,3,4,5\}$

$E = \{\{1,2\},\{1,4\},\{1,5\},\{2,3\},\{3,4\},\{3,5\},\{4,5\}\}$

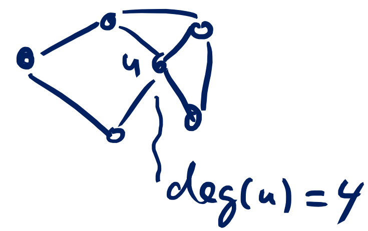
$E = \{(1,2),(1,5),(2,3),(3,2),\ldots\}$

Knotengrade

Graph $G = (V, E)$ ungerichtet:

- Grad eines Knoten $u \in V$: Anzahl Kanten (Nachbarn) von u

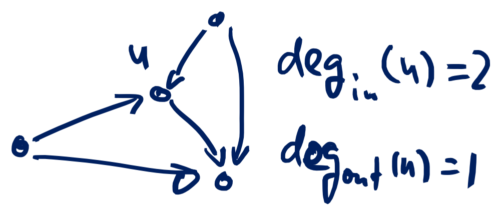
$$\deg(u) := |\{u, v\} : \{u, v\} \in E|$$



Graph $G = (V, E)$ gerichtet:

- Eingangsgrad eines Knoten $u \in V$: Anzahl eingehende Kanten

$$\deg_{in}(u) := |(v, u) : (v, u) \in E|$$

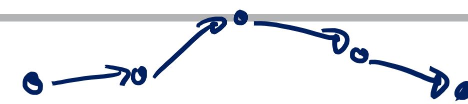


- Ausgangsgrad eines Knoten $u \in V$: Anzahl ausgehende Kanten

$$\deg_{out}(u) := |(u, v) : (u, v) \in E|$$

Pfade

einfache Pfade: Knoten wiederholen sich nicht



Pfade in einem Graph $G = (V, E)$

- Ein Pfad in G ist eine Folge $u_1, u_2, \dots, u_k \in V$ mit
  - gerichteter Graph: $(u_i, u_{i+1}) \in E$ für alle $i \in \{1, \dots, k-1\}$
  - ungerichteter Graph: $\{u_i, u_{i+1}\} \in E$ für alle $i \in \{1, \dots, k-1\}$

Länge eines Pfades (z. T. auch Kosten eines Pfades)

- ohne Kantengewichte: Anzahl der Kanten
- mit Kantengewichten: Summe der Kantengewichte

Kürzester Pfad (shortest path) zwischen Knoten u und v

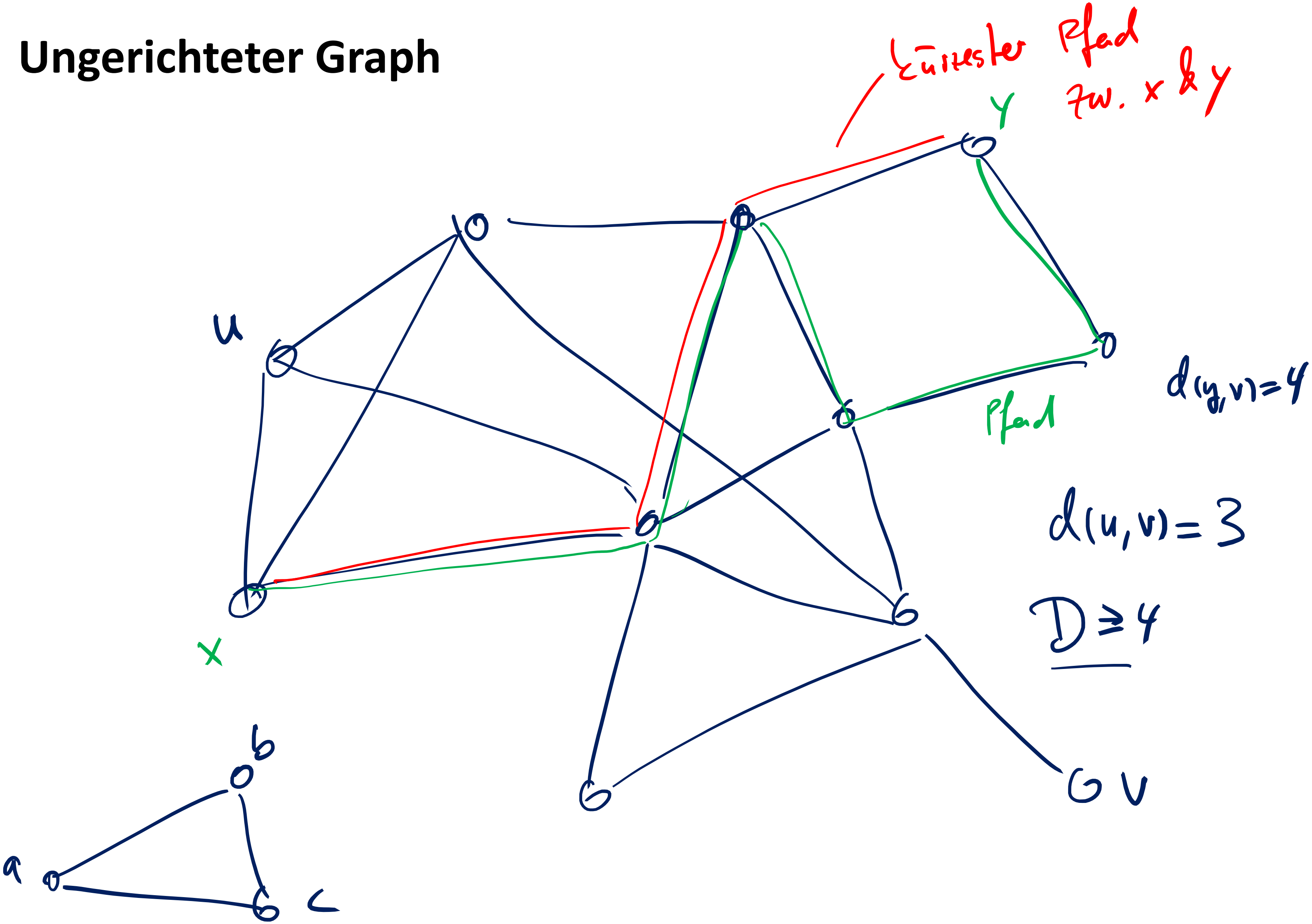
- Pfad $u, \dots, v$ mit kleinster Länge
- **Distanz** $d(u, v)$: Länge eines kürzesten Pfades zwischen u und v

Durchmesser $\boldsymbol{D \coloneqq \max_{u,v \in V} d(u, v)}$

- Länge des längsten kürzesten Pfades

Beispiele: Pfade, Distanzen, Durchmesser

Ungerichteter Graph



Beispiele: Pfade, Distanzen, Durchmesser

Gerichteter Graph

stark zusammenhängend:

es gibt einen ger. Pfad von jedem zu jedem anderen Knoten