Algorithmen und Datenstrukturen

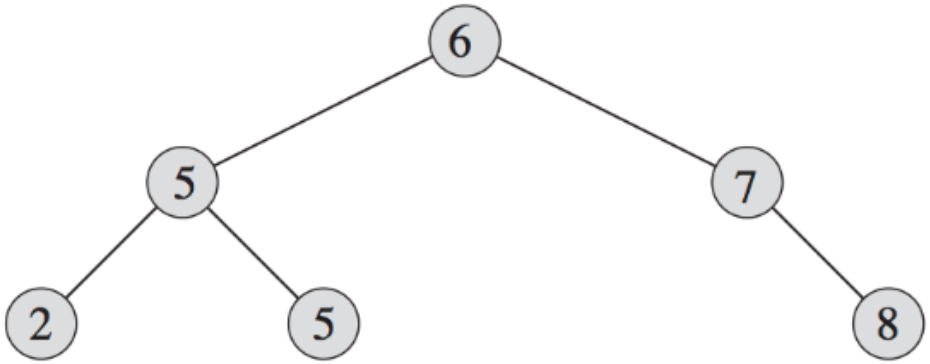
Vorlesung 7

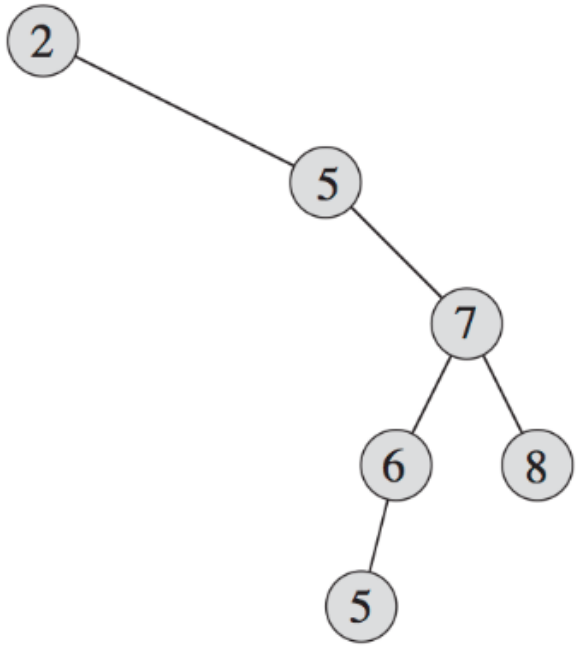
Binäre Suchbäume II

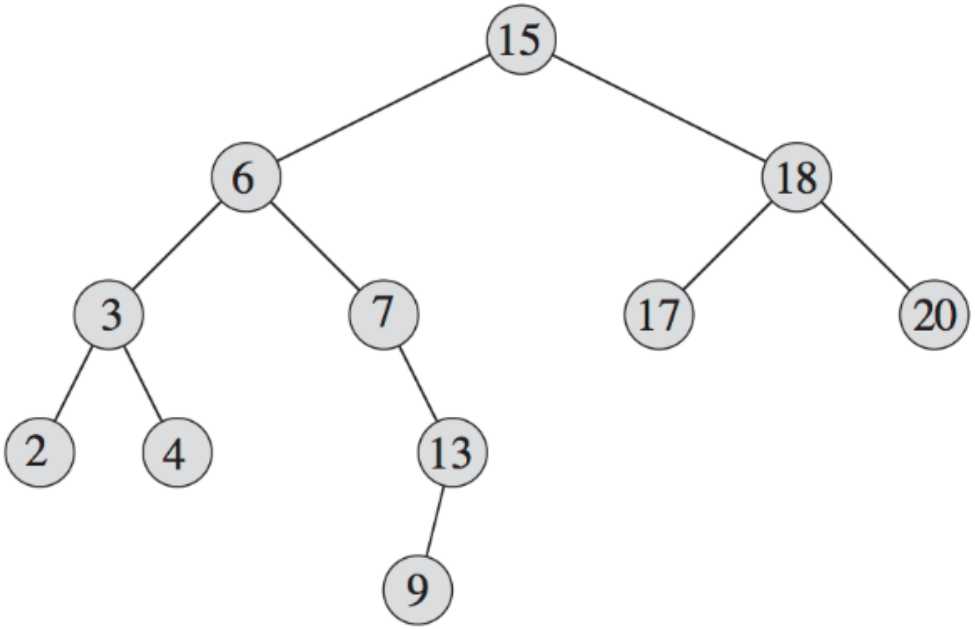
UNI FREIBURG

Fabian Kuhn
Algorithmen und Komplexität

Binäre Suchbäume







Quelle: [CLRS]

Tiefe eines binären Suchbaums

UNI FREIBURG

Worst-Case Laufzeit der Operationen
find, min, max, predecessor, successor, insert, delete:

$$O(\text{Tiefe des Baums})$$

- Im **besten Fall** ist die Tiefe $\log_2 n$
  - Definition Tiefe: Länge des längsten Pfades von der Wurzel zu einem Blatt
- Im **schlechtesten Fall** ist die Tiefe $n - 1$

- Im **durchschnittlichen Fall** ist die Tiefe $O(\log n)$
  - Durchschnittlich heisst hier bei zufälliger Einfügereihenfolge

Ist es möglich, in einem binären Suchbaum immer Tiefe $O(\log n)$ zu garantieren?

“Typischer” Fall

UNI FREIBURG

Zufälliger binärer Suchbaum:

- **n Schlüssel werden in zufälliger Reihenfolge eingefügt**

Beobachtung:

- Mit Wahrscheinlichkeit $1/3$ haben beide Teilbäume der Wurzel mindestens $n/3$ Knoten.

1. Drittel der Schlüssel	2. Drittel der Schlüssel	3. Drittel der Schlüssel

- Die Wurzel ist der erste eingefügte Schlüssel
- Mit Wahrscheinlichkeit $1/3$ kommt die Wurzel aus dem mittleren Drittel

“Typischer” Fall

UNI FREIBURG

Zufälliger binärer Suchbaum:

- **n Schlüssel werden in zufälliger Reihenfolge eingefügt**

Beobachtung:

- Mit Wahrscheinlichkeit $1/3$ haben beide Teilbäume der Wurzel mindestens $n/3$ Knoten.
- Analoges gilt auch für alle Teilbäume
- Im Durchschnitt wird deshalb auf jedem 3. Schritt von der Wurzel Richtung eines Blattes, der Teilbaum um einen Faktor $2/3$ kleiner!
- Verkleinern um einen Faktor $2/3$ geht nur $O(\log n)$ oft.
- Tiefe eines zufälligen binären Suchbaums ist deshalb $O(\log n)$
- Genaue Rechnung ergibt:

  Erwartete Tiefe eines zufälligen bin. Suchbaums: $\mathbf{4.311 \cdot \ln n}$

“Typischen” Fall erzwingen?

UNI FREIBURG

“Typischer” Fall:

- Falls die Schlüssel in zufälliger Reihenfolge eingefügt werden, hat der Baum Tiefe $O(\log n)$
- Operationen haben Laufzeit $O(\log n)$

Problem:

- Zufällige Reihenfolge ist nicht unbedingt der typische Fall!
- Vorsortierte Werte kann genau so typisch sein
  - Das ergibt einen sehr schlechten binären Suchbaum

Idee:

- Können wir zufällige Reihenfolge erzwingen?
- Schlüssel werden in beliebiger Reihenfolge eingefügt, aber Struktur soll immer wie bei zufälliger Reihenfolge sein!

Treap: «Erzwinge» zufällige Reihenfolge

UNI FREIBURG

- Schlüssel werden in beliebiger Reihenfolge eingefügt, aber die Struktur ist immer so, wie wenn die Schlüssel in zufälliger Reihenfolge eingefügt wären.

- Dazu wird zu jedem Schlüssel beim Einfügen ein Zufallswert bestimmt und die Struktur des Baums wird dann immer so umgeformt, dass sie so ist, wie wenn die Schlüssel in der sortierten Reihenfolge nach diesen Zufallswerten eingefügt worden wären.

- Die notwendigen Umformungen der Baumstruktur bei insert und delete kann man jeweils in Zeit $O(\text{Tiefe})$ machen

- Mit hoher Wahrscheinlichkeit sind dann alle Operationen $O(\log n)$.

- Details z.B. in der Vorlesung von 2018.

Rot-Schwarz-Bäume

UNI FREIBURG

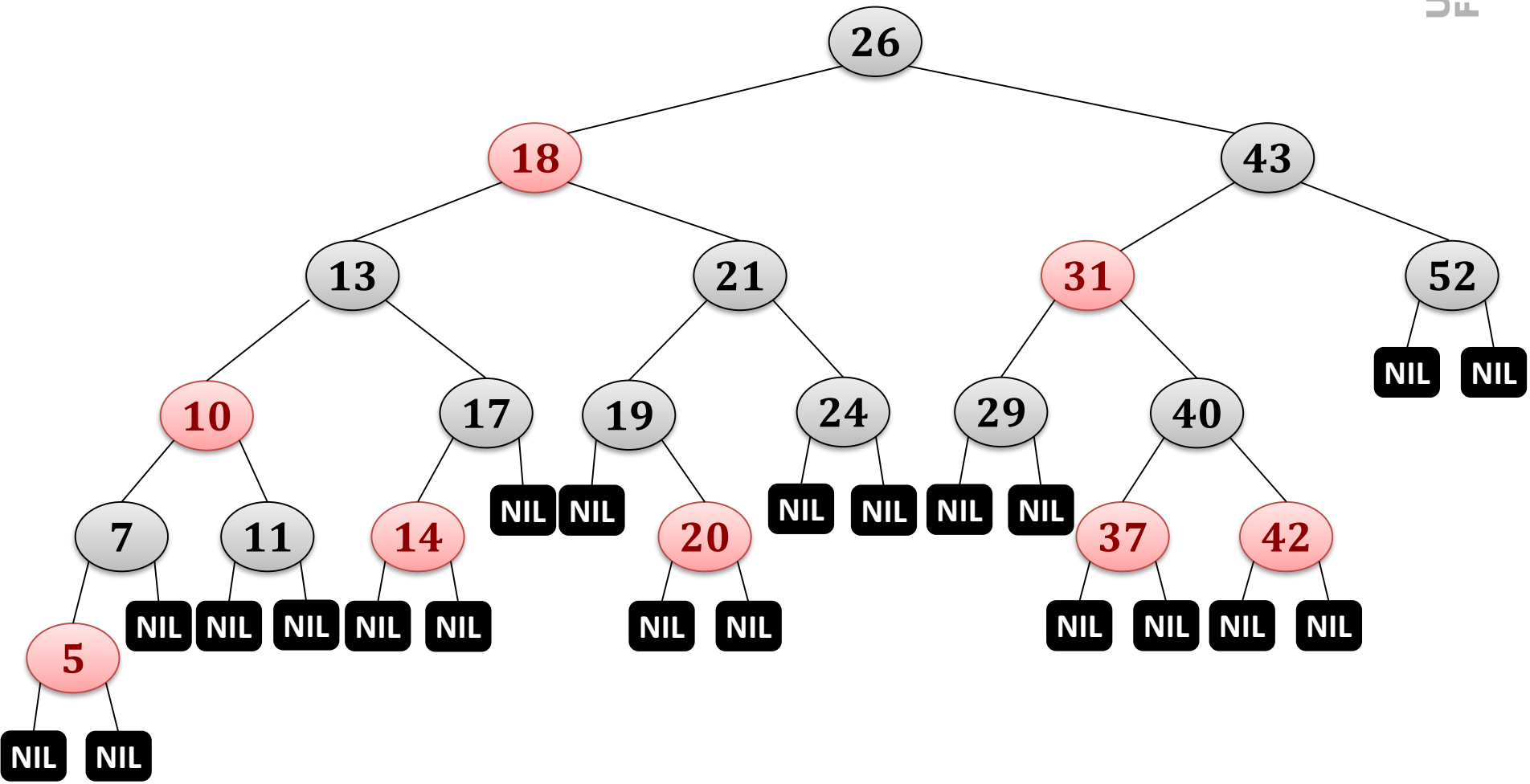
Ziel: Binäre Suchbäume, welche **immer balanciert** sind

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

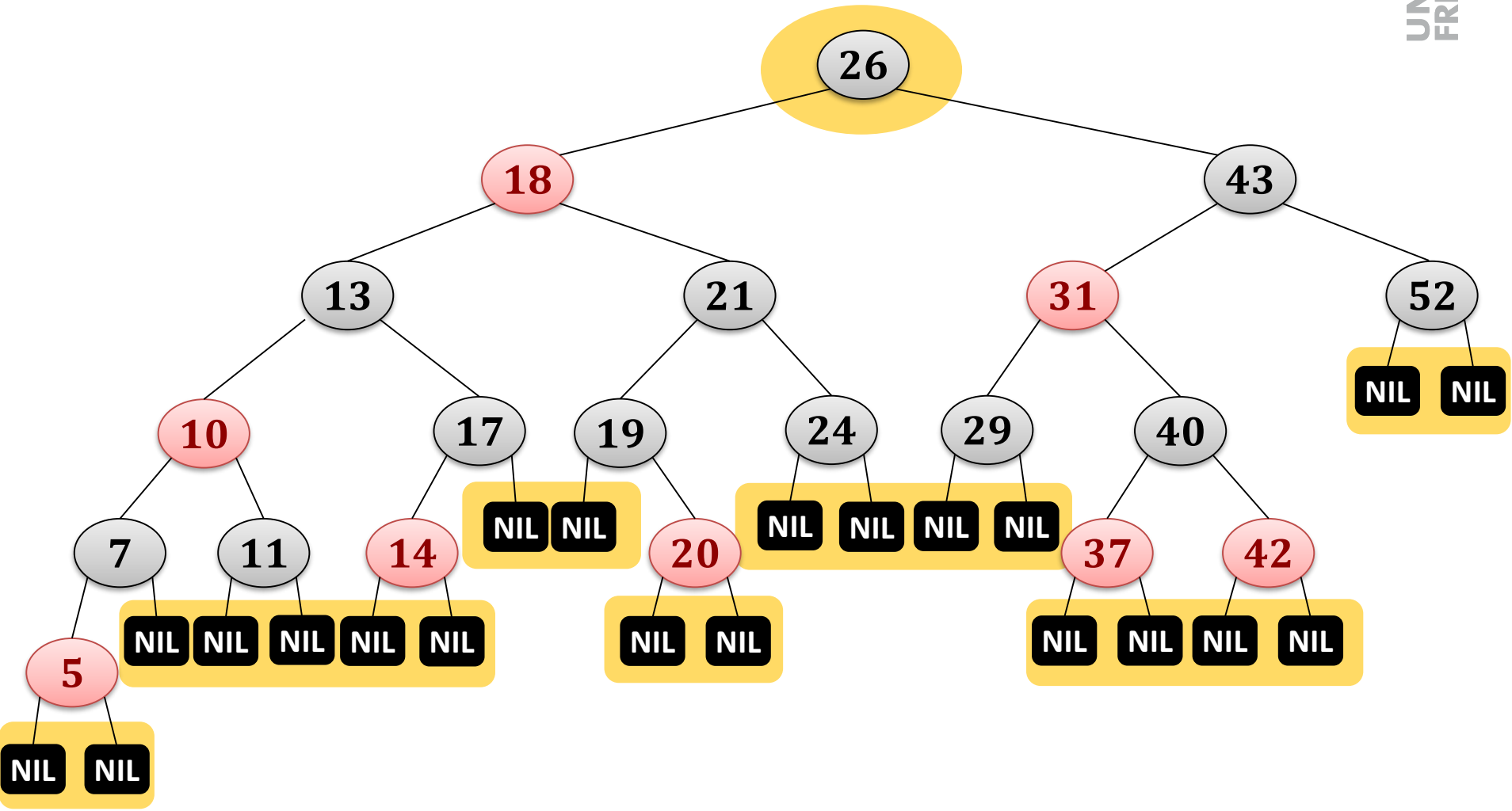
Rot-Schwarz-Bäume sind binäre Suchbäume, für die gilt:

1) Alle Knoten sind **rot** oder **schwarz**
2) Wurzel ist schwarz
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

Rot-Schwarz-Bäume: Beispiel

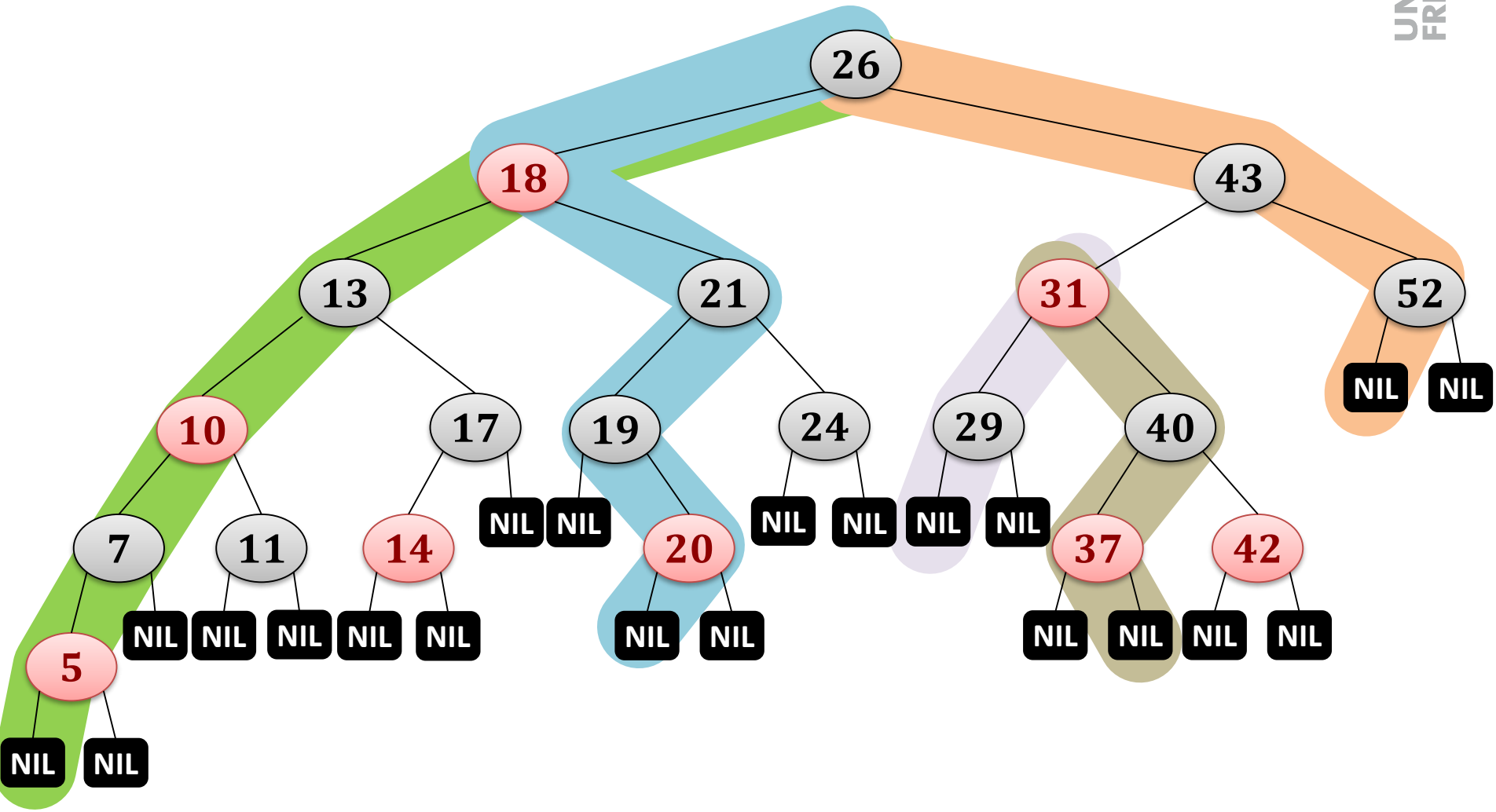


Rot-Schwarz-Bäume: Eigenschaften



- Wurzel und Blätter (NIL-Knoten) sind schwarz
- Rote Knoten haben zwei schwarze Kinder

Rot-Schwarz-Bäume: Eigenschaften



- In jedem Teilbaum haben alle direkten Pfade von den Wurzel zu den Blättern des Teilbaums die gleiche Anzahl schwarze Knoten.

Sentinel

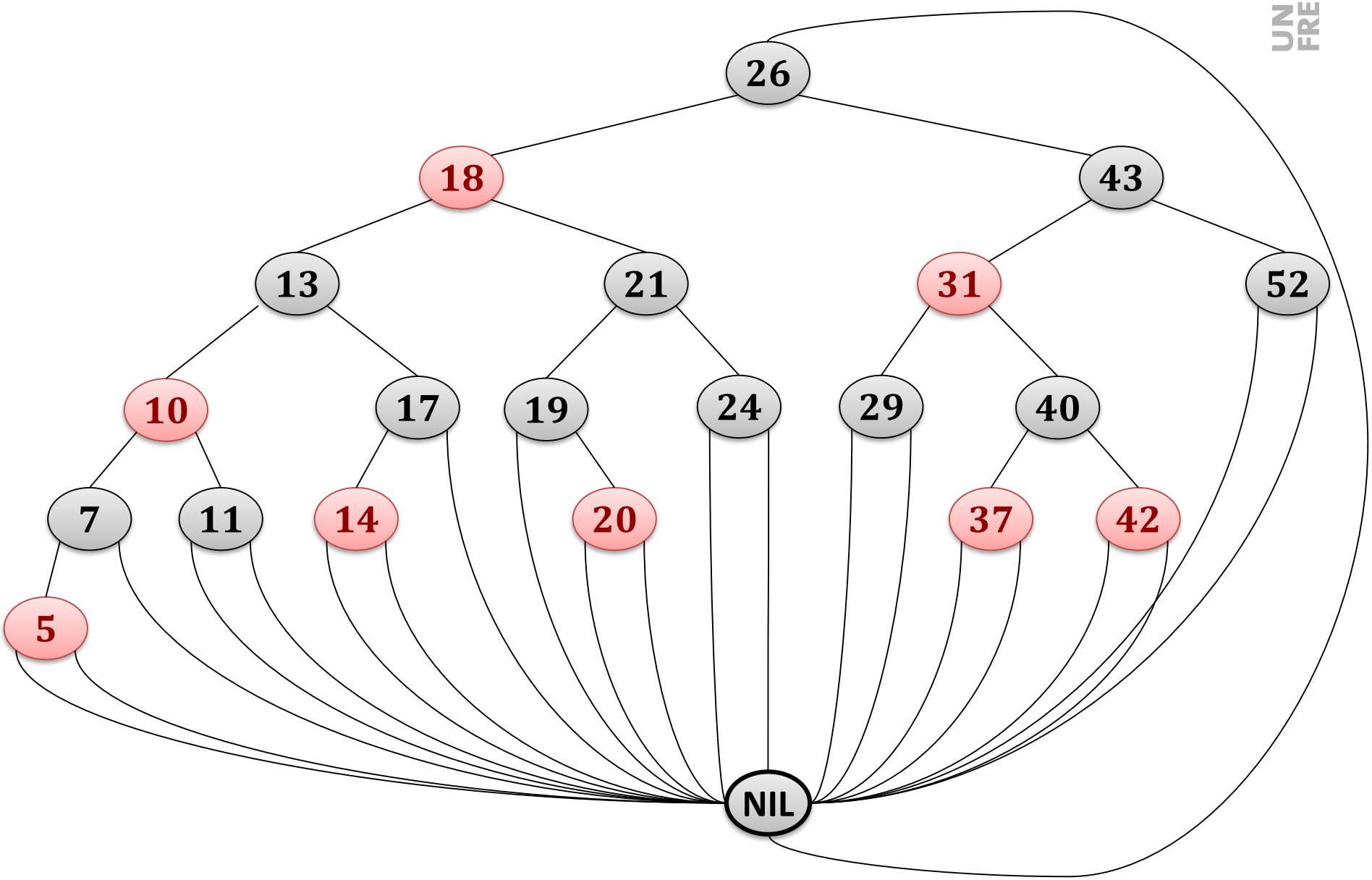
UNI FREIBURG

Um den Code zu vereinfachen…

Sentinel-Knoten: *NIL*

- Ersetzt alle None/null-Pointer
- *NIL.key* ist nicht definiert
- *NIL.color = black*
  - Als Blätter des Baumes verstehen wir die NIL-Knoten (sind alle schwarz)
  - repräsentiert alle Blätter des Baumes
- *NIL.left*, *NIL.right*, *NIL.parent* können beliebig gesetzt sein
  - Wir müssen darauf achten, dass sie nie ausgelesen werden
  - Wenn es den Code vereinfacht, kann man *NIL.parent*, ... neu setzen

Rot-Schwarz-Bäume: Sentinel

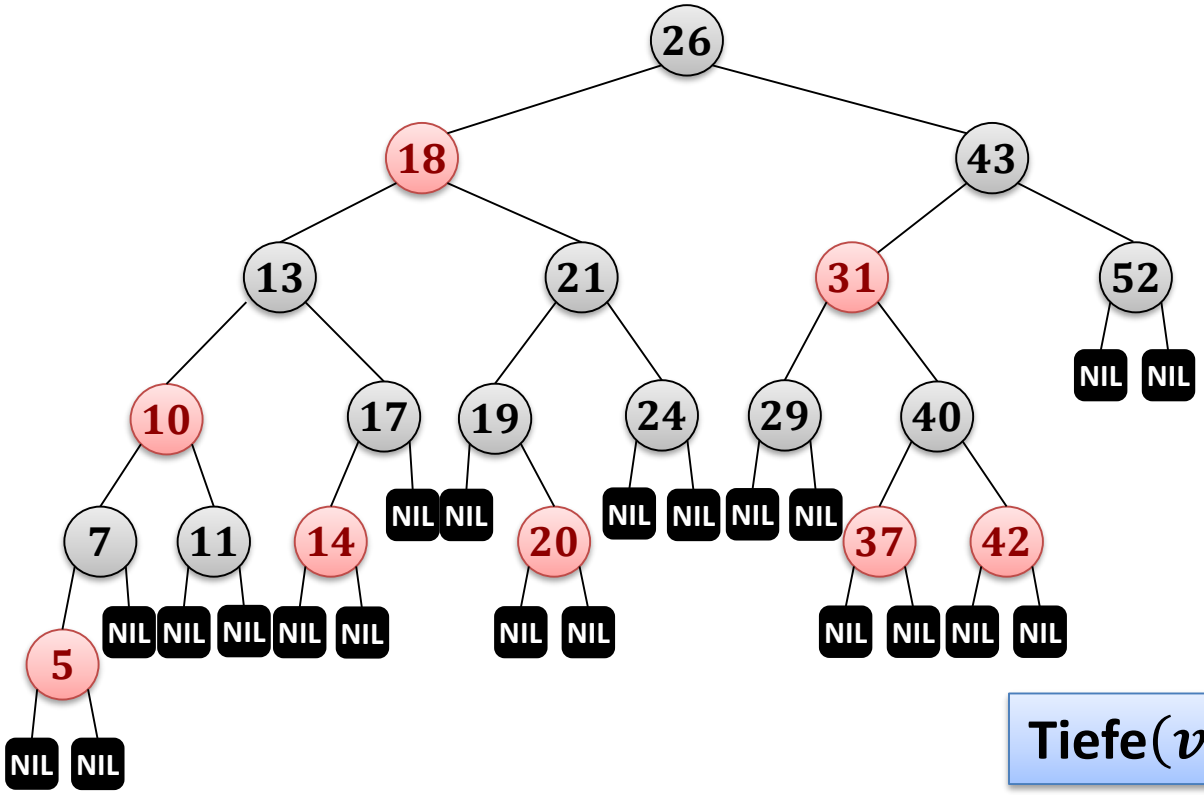


Tiefe / Schwarz-Tiefe

Definition: Die **Tiefe (T)** eines Knoten v ist die maximale Länge eines direkten Pfades im Teilbaum 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!



$$\text{Tiefe}(v) \leq 2 \cdot \text{Schwarz-Tiefe}(v)$$

Schwarz-Tiefe ↔ Anzahl Knoten

UNI FREIBURG

Lemma: Die **Anzahl innerer Knoten** im Teilbaum eines Knoten v ($v \neq$ NIL) mit **Schwarz-Tiefe $ST(v)$** ist

$$\geq 2^{ST(v)} - 1$$

Beweis per Induktion über die Schwarz-Tiefe $ST(v)$ von v:

- **Verankerung:** $ST(v) = 1$: Anz. innere Knoten $\geq 2^1 - 1 = 1$
  - v selbst ist ein innerer Knoten!

- **Induktionsschritt:**
  - Betrachte einen Knoten v mit Schwarz-Tiefe $ST(v) > 1$
  - **Induktionsvoraussetzung:**
    - Teilbäume mit Schwarz-Tiefe $s < ST(v)$ haben $\geq 2^s - 1$ innere Knoten
    - Teilbäume mit Schwarz-Tiefe $s = ST(v)$ haben $\geq 2^{s-1} - 1$ innere Knoten
      - Solche Teilbäume enthalten Teilbäume mit Schwarz-Tiefe $s - 1$

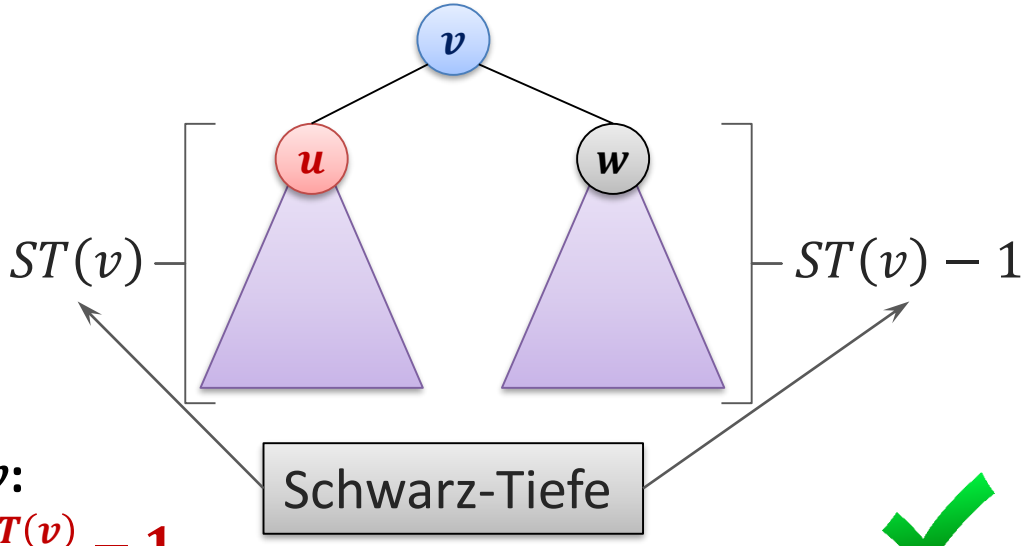
Schwarz-Tiefe ↔ Anzahl Knoten

Lemma: Die **Anzahl innerer Knoten** im Teilbaum eines Knoten v ($v \neq$ NIL) mit **Schwarz-Tiefe $ST(v)$** ist

$$\geq 2^{ST(v)} - 1$$

Beweis per Induktion über die Schwarz-Tiefe $ST(v)$ von v:

- **Induktionsschritt:**
  - Betrachte einen Knoten v mit Schwarz-Tiefe $ST(v)$
  - Beide Teilbäume haben Schwarz-Tiefe $\geq ST(v) - 1$
  - Induktionsvoraussetzung: Beide Teilbäume haben $\geq 2^{ST(v)-1} - 1$ Knoten.
  - **Anz. Knoten im Teilbaum von v:** $\geq 2 \cdot \left(2^{ST(v)-1} - 1\right) + 1 = 2^{ST(v)} - 1$



Tiefe eins Rot-Schwarz-Baumes

UNI FREIBURG

Theorem:

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

Beweis:

- Anzahl innerer Knoten : n (alle ausser den NIL-Knoten)
- Aus dem Lemma von vorher folgt, dass

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

- Wenn man nach $ST(root)$ auflöst, erhält man

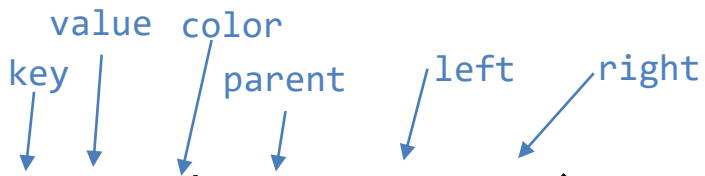
$$ST(root) \leq \log_2(n+1)$$

- Das Theorem folgt nun, weil

$$\text{Tiefe} \leq 2 \cdot \text{Schwarz}-\text{Tiefe}$$

Rot-Schwarz-Bäume: Einfügen

insert(x): Einfügen zuerst wie üblich, neuer Knoten ist **rot**




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

Rot-Schwarz-Bäume: Einfügen

UNI FREIBURG

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

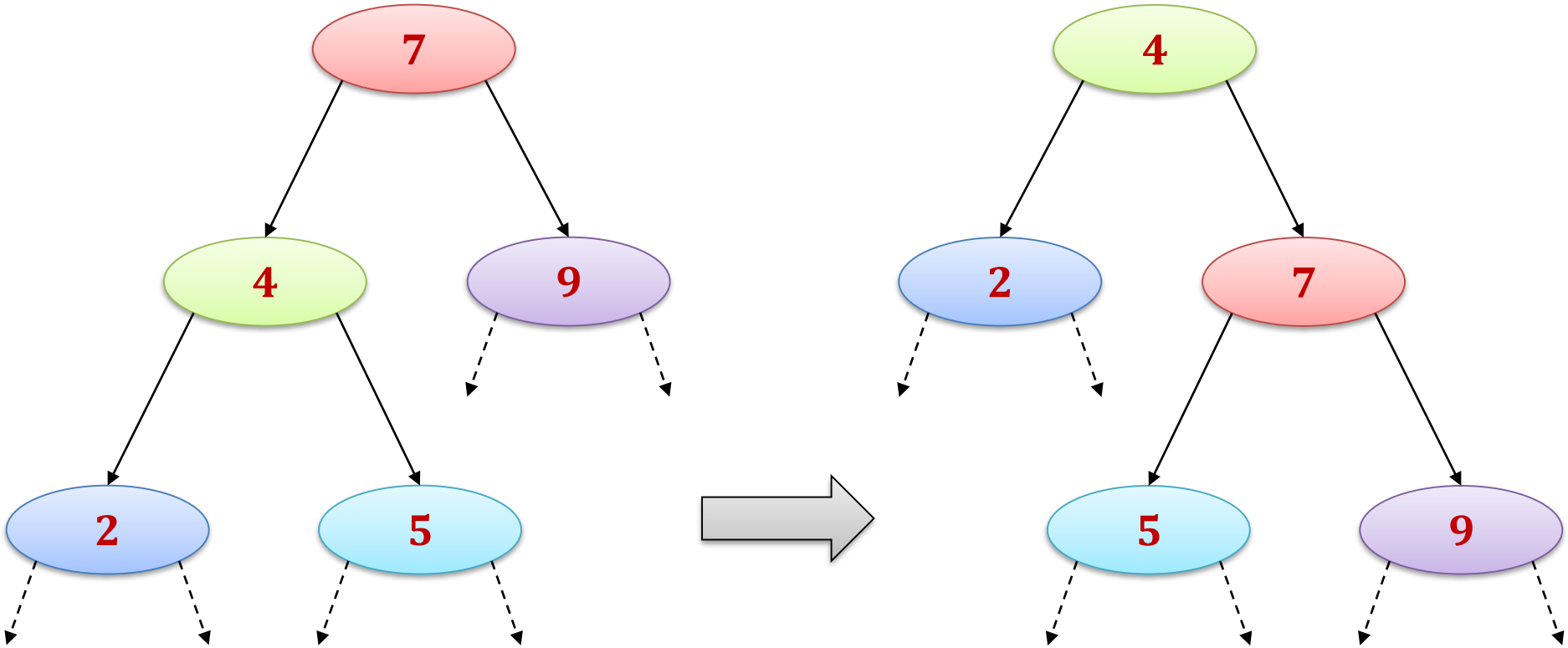
- Bedingungen sind alle erfüllt, ausser
  - Eingefügter Knoten v ist die Wurzel (Bedingung 2 nicht erfüllt)
  - Der Knoten $v.\mathrm{parent}$ ist rot (Bedingung 4 nicht erfüllt)
- Falls v die Wurzel ist, kann man v einfach schwarz einfügen
- Falls $v.\mathrm{parent}$ rot ist, müssen wir den Baum anpassen
  - so, dass 1, 3 und 5 erfüllt bleiben und 2 und 4 am Schluss erfüllt sind

Rotationen

UNI FREIBURG

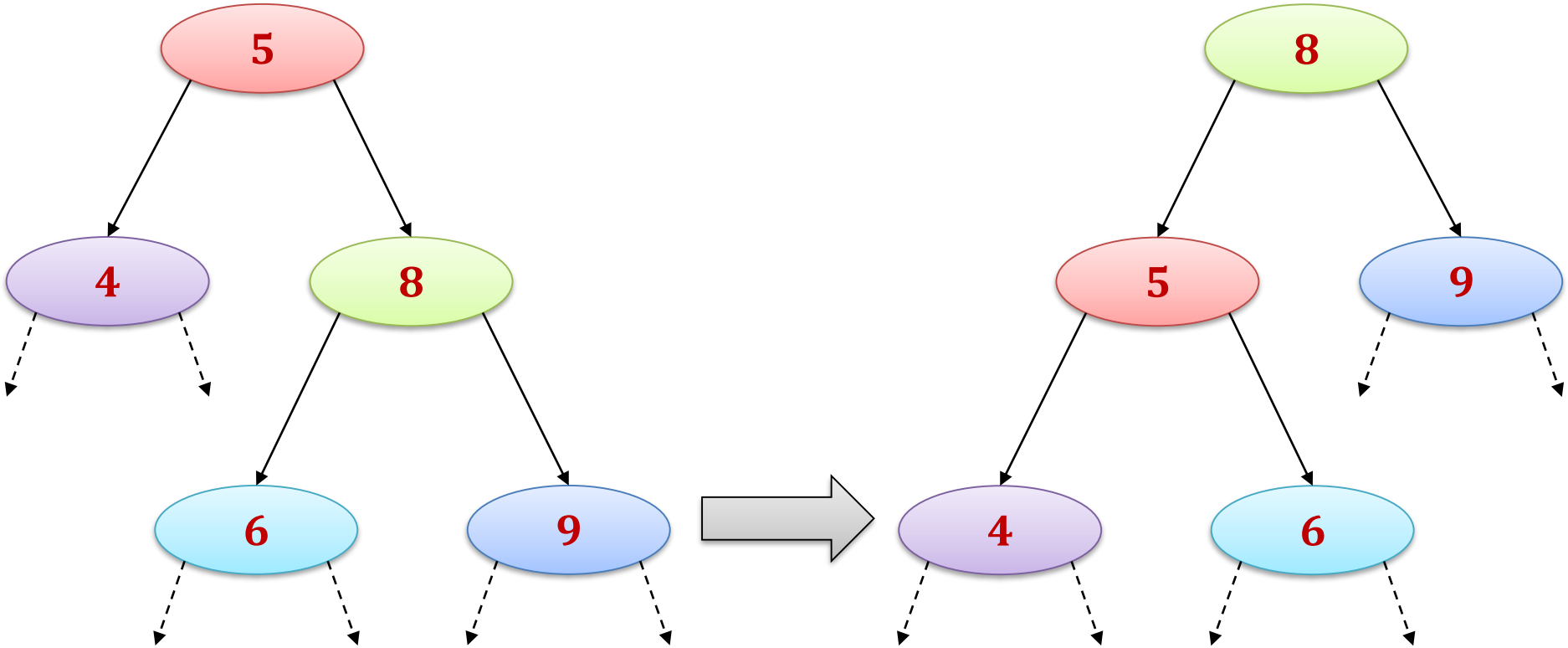
Rechtsrotation

- Operation, um den binären Suchbaum lokal umzuordnen
- Ändert Topologie, erhält Binäre Suchbaum Eigenschaft

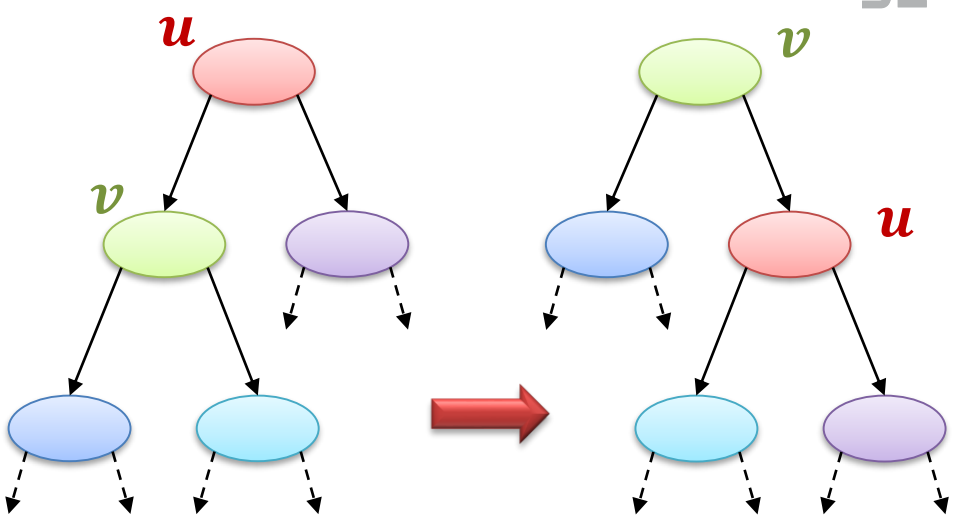


Rotationen

Linksrotation



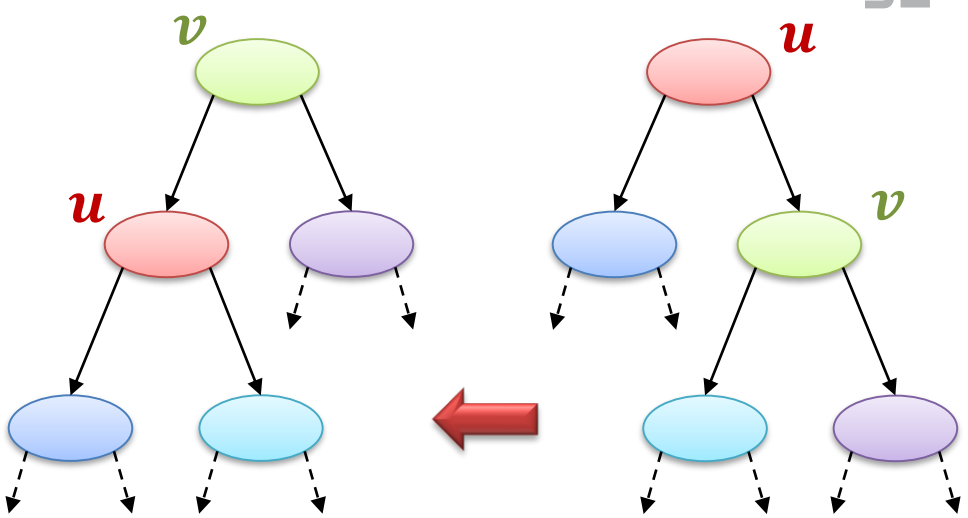
Rechtsrotation




```
right-rotate(u,v):
  u.left = v.right
  u.left.parent = u
  if u == root then
     root = v
  else
     if u == u.parent.left then
        u.parent.left = v
     else
        u.parent.right = v
  v.parent = u.parent
  v.right = u
  u.parent = v
```

Laufzeit: $O(1)$

Linksrotation




```
left-rotate(u,v):
  u.right = v.left
  u.right.parent = u
  if u == root then
     root = v
  else
     if u == u.parent.left then
        u.parent.left = v
     else
        u.parent.right = v
  v.parent = u.parent
  v.left = u
  u.parent = v
```

Laufzeit: $O(1)$

Korrektheit: Rotationen

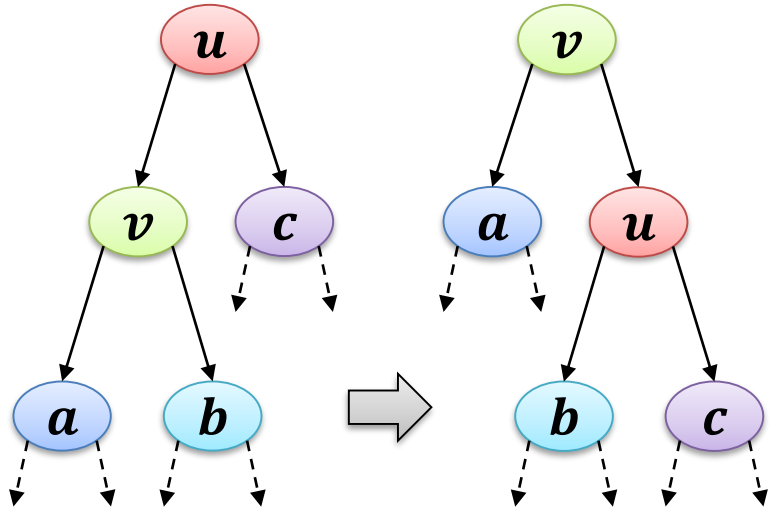
Lemma: Rotationen erhalten die “Bin. Search Tree”-Eigenschaft

Sortierte Reihenfolge vor Rotation:

- $(a < v < b) < u < c$

Sortierte Reihenfolge nach Rotation:

- $a < v < (b < u < c)$



Rot-Schwarz-Bäume: Einfügen

Rot-Schwarz-Baum Bedingungen nach dem Einfügen

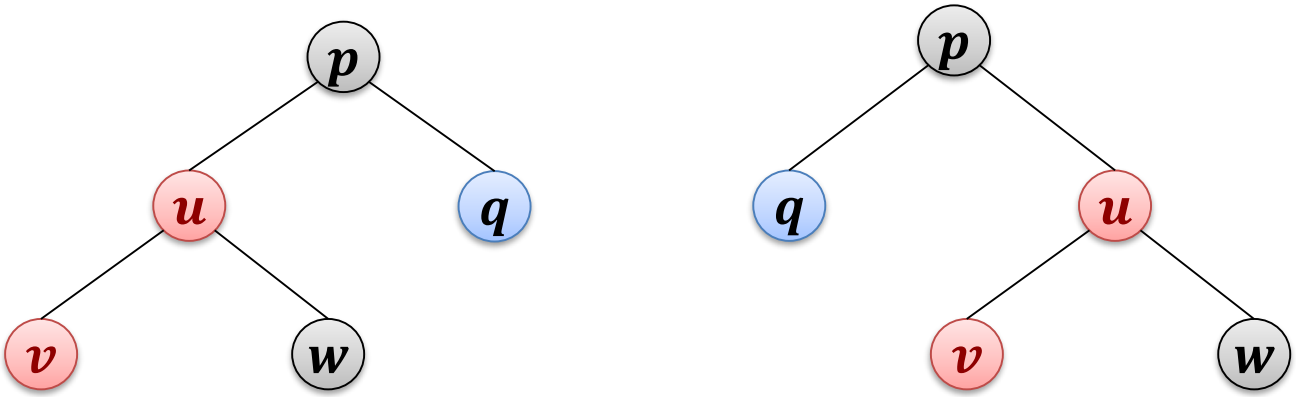
1. Alle Knoten sind rot oder schwarz
2. Die Wurzel ist schwarz
3. Die Blätter (NIL) sind schwarz
4. Rote Knoten haben zwei schwarze Kinder
5. Von jedem Knoten v haben alle direkten Pfade zu Blättern gleich viele schwarze Knoten

- Bedingungen sind alle erfüllt, ausser
  - Eingefügter Knoten v ist die Wurzel (Bedingung 2 nicht erfüllt)
  - Der Knoten $v.\mathrm{parent}$ ist rot (Bedingung 4 nicht erfüllt)
- Falls v die Wurzel ist, kann man v einfach schwarz einfügen
- Falls $v.\mathrm{parent}$ rot ist, müssen wir den Baum anpassen
  - so, dass 1, 3 und 5 erfüllt bleiben und 2 und 4 am Schluss erfüllt sind

Rot-Schwarz-Bäume: Einfügen

Baum anpassen nach dem Einfügen:

- Annahmen:
  - v ist rot, v hat zwei schwarze Kinder
  - $u := v.\text{parent}$ 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



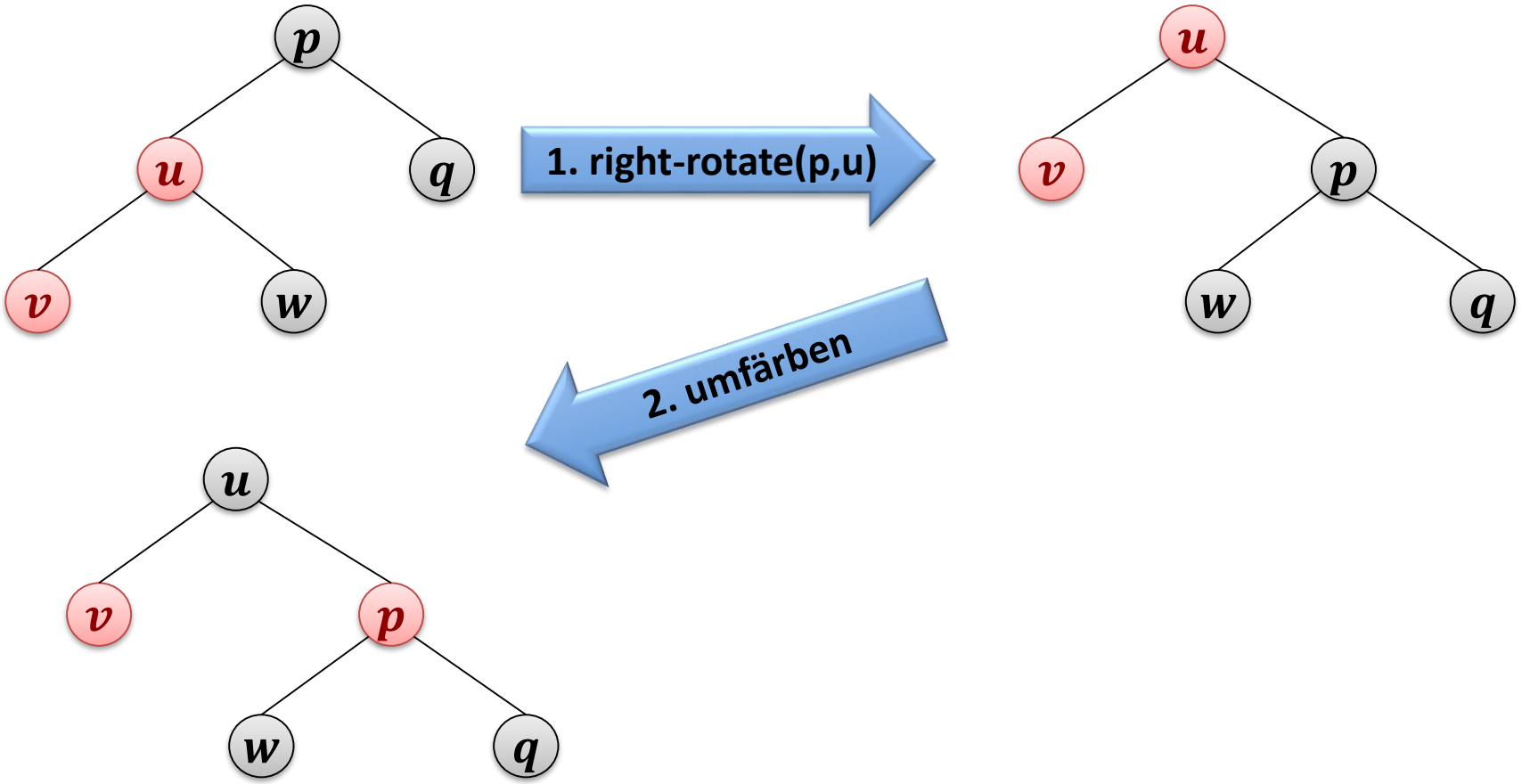
- Fallunterscheidung anhand von **Farbe von q (Geschwister von u)** und anhand von $\boldsymbol{u = p.\text{left}}$ **oder** $\boldsymbol{u = p.\text{right}}$

Rot-Schwarz-Bäume: Einfügen

UNI FREIBURG

Fall 1: Geschwisterknoten q von u ist schwarz

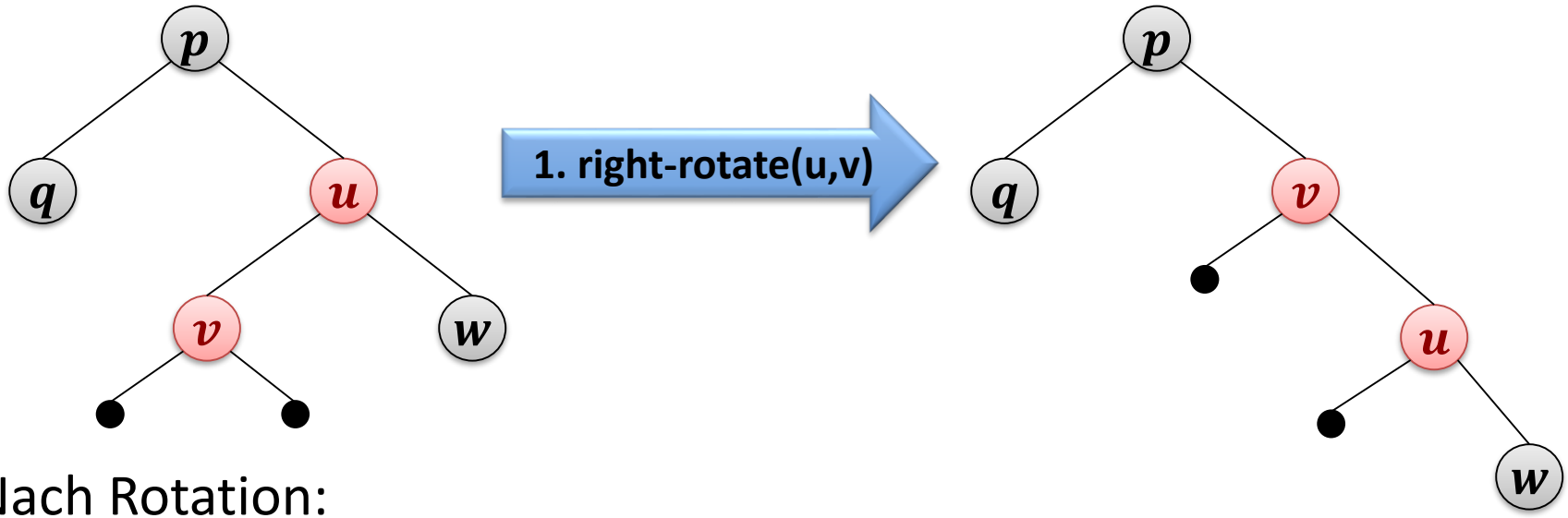
- **Fall 1a: $u = p.\text{left}$**



Rot-Schwarz-Bäume: Einfügen

Fall 1: Geschwisterknoten q von u ist schwarz

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



- 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

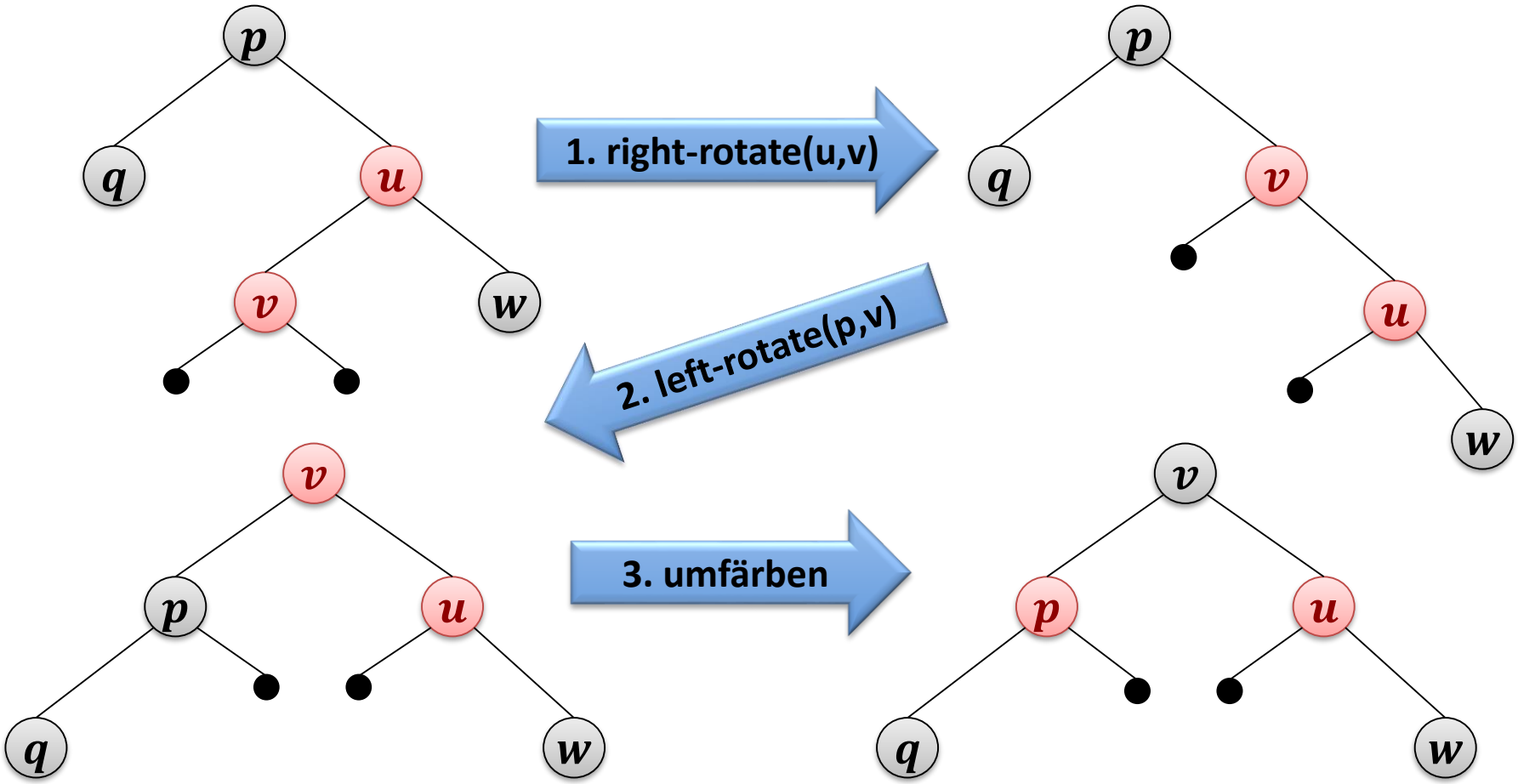
    left-rotate(p,v) und umfärben

Rot-Schwarz-Bäume: Einfügen

UNI FREIBURG

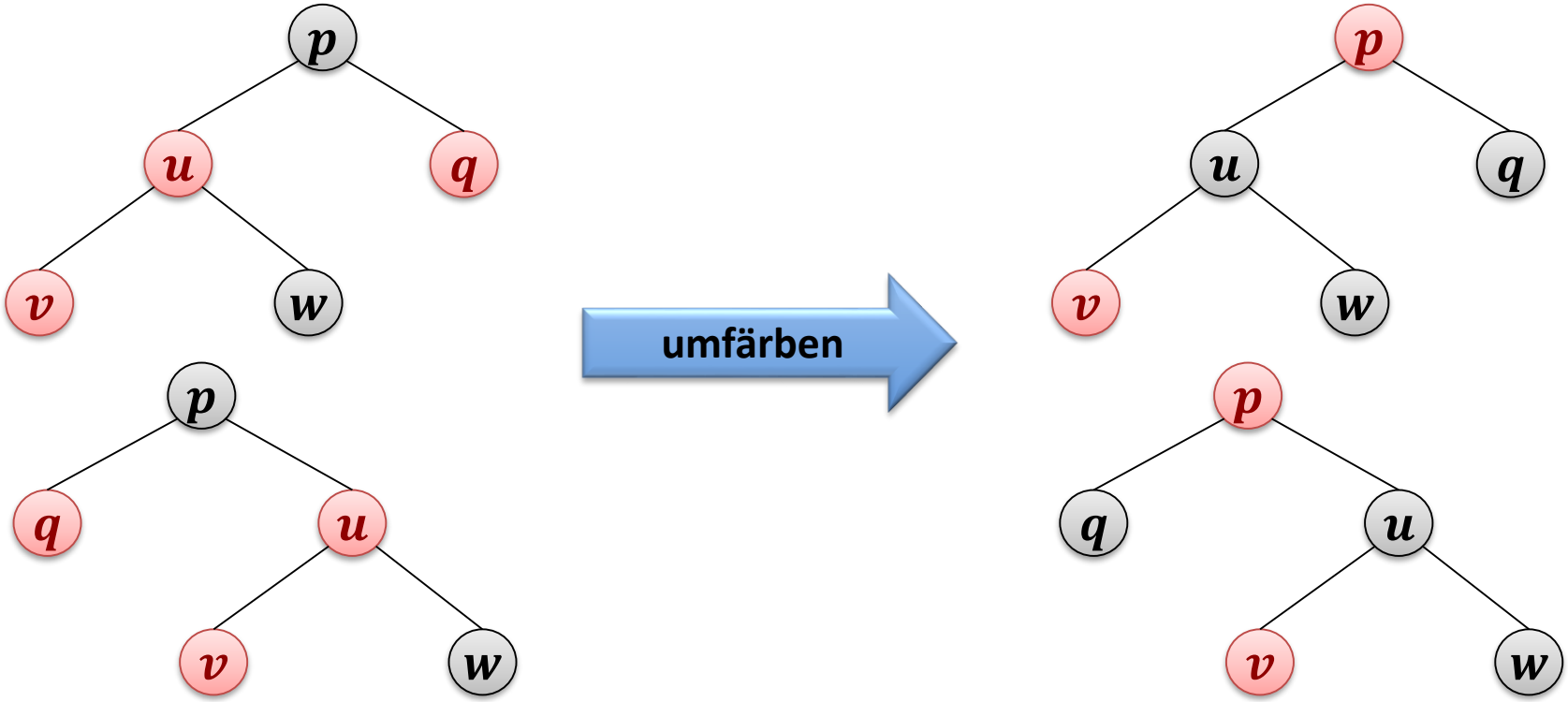
Fall 1: Geschwisterknoten q von u ist schwarz

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



Rot-Schwarz-Bäume: Einfügen

Fall 2: Geschwisterknoten q von u ist rot



- Falls $p.\text{parent}$ schwarz ist, sind wir fertig
  - das ist auch der Fall, falls $p == \text{root}$ (dann noch $\text{root.color} := \text{black}$)
- 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

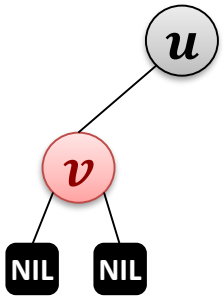
1. Finde wie üblich einen Knoten v, welcher gelöscht wird
   - Knoten v hat höchstens ein Nicht-NIL-Kind!

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

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

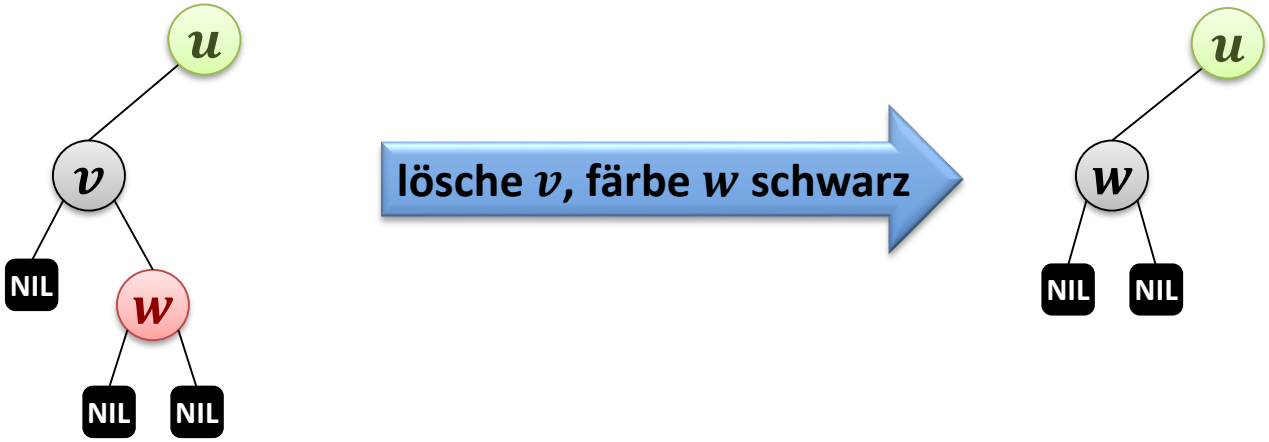


- v kann einfach gelöscht werden
  - Der Baum bleibt ein Rot-Schwarz-Baum

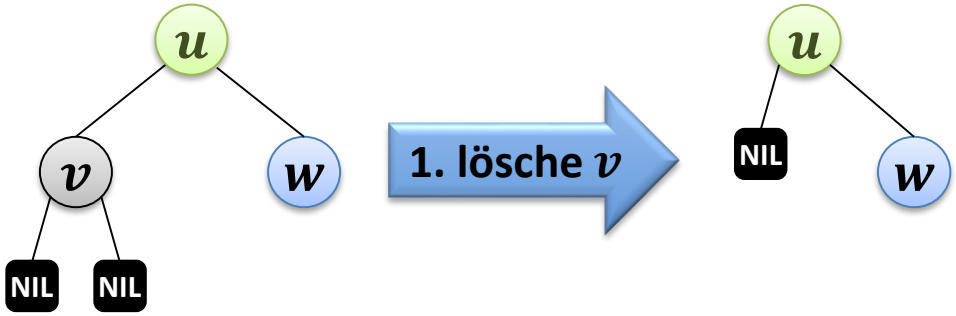
Rot-Schwarz-Bäume: Löschen

Fall 2: Knoten v ist schwarz

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



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



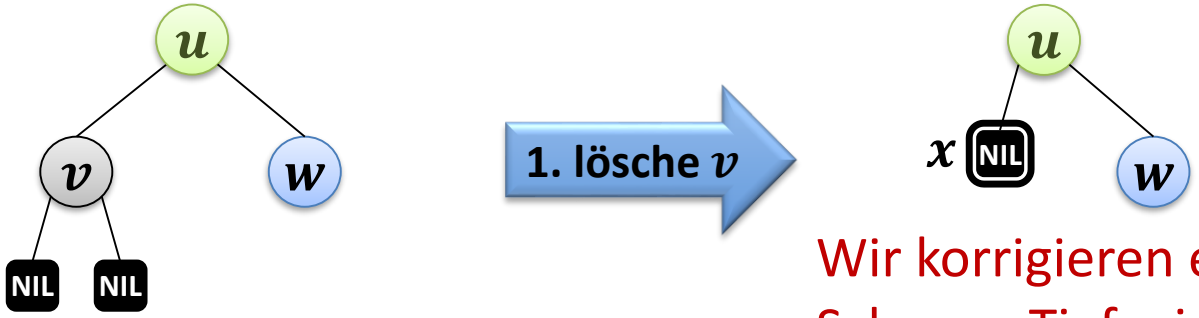
Knoten u hat jetzt nach links nur noch Schwarz-Tiefe 1 (statt 2)

→ **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)!

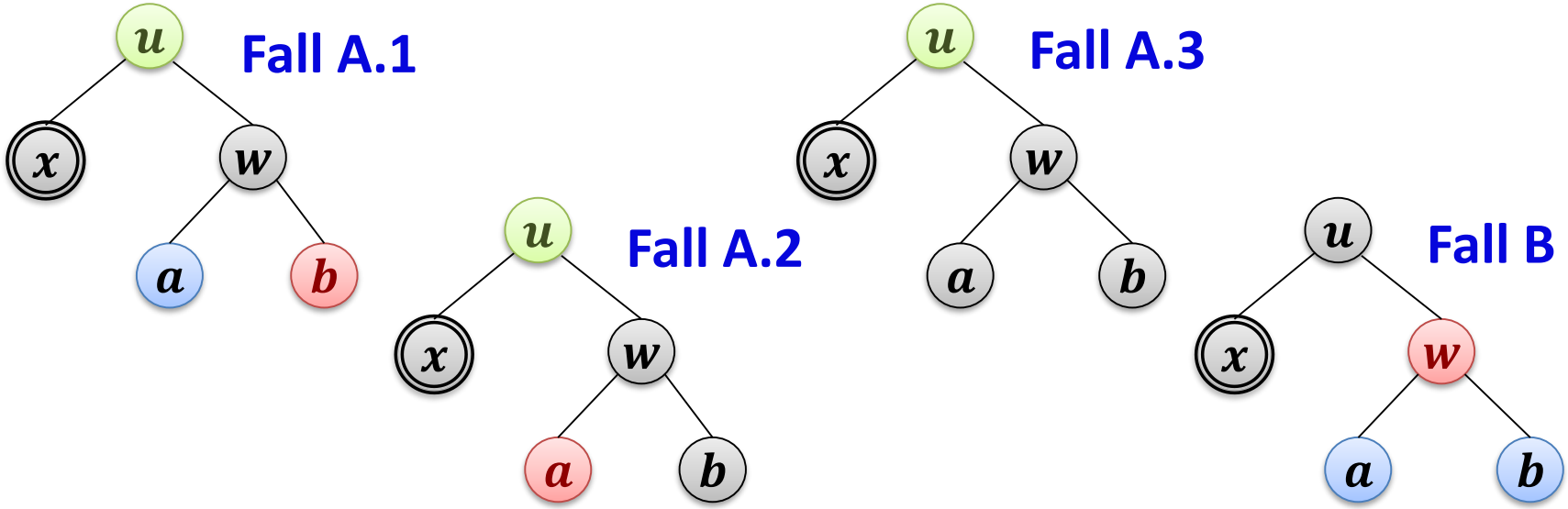
Rot-Schwarz-Bäume: Löschen

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

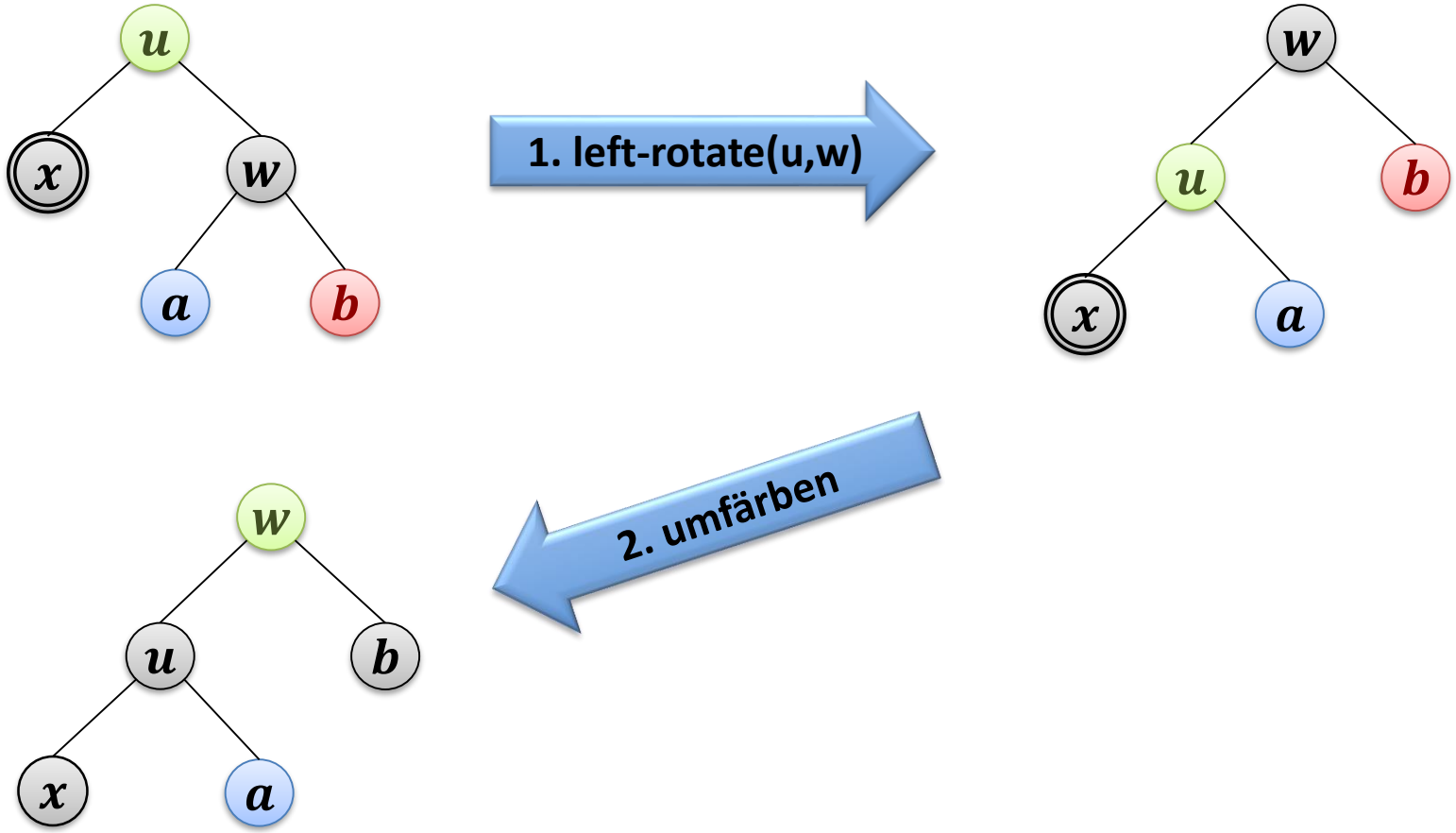
Fallunterscheidung:

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



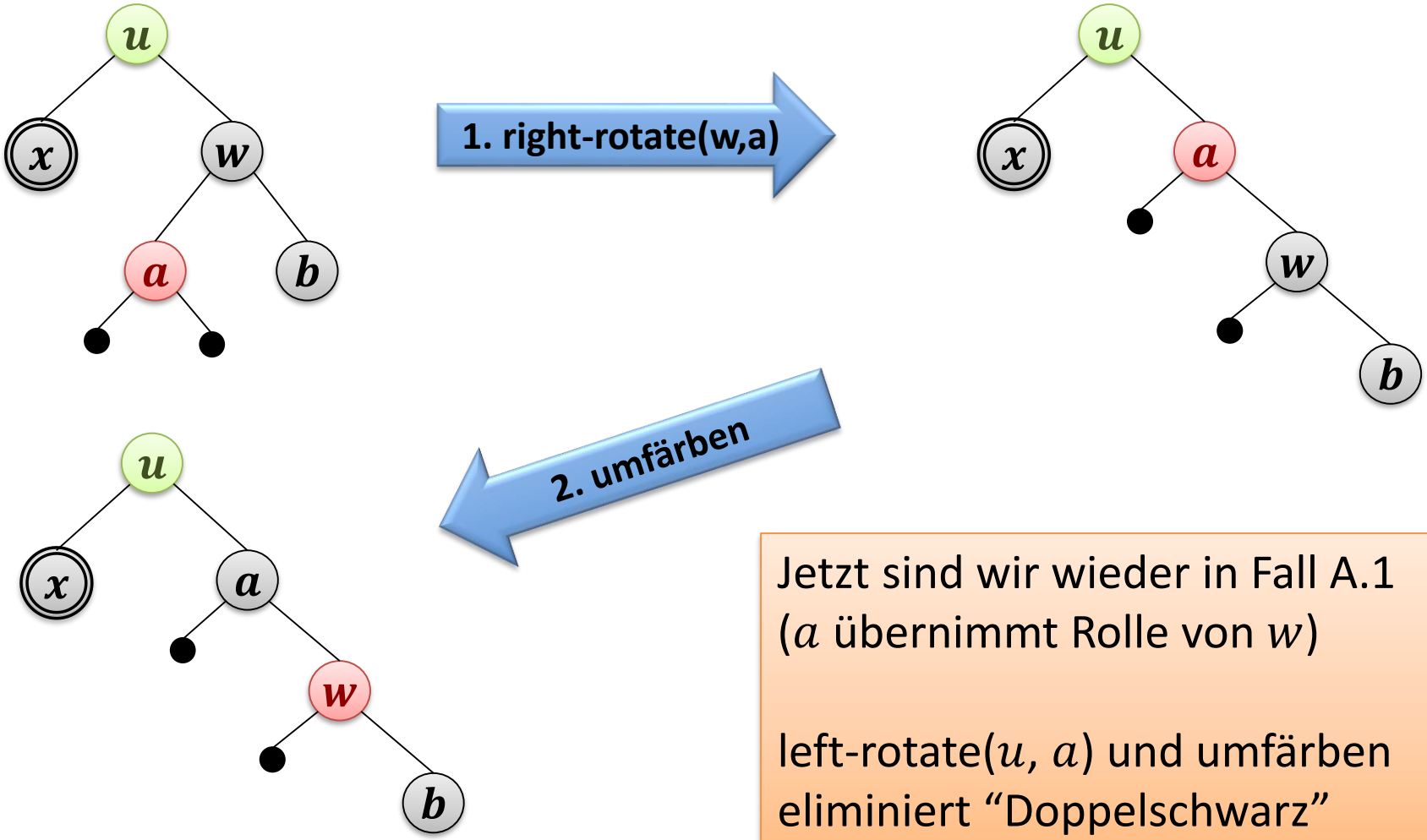
Rot-Schwarz-Bäume: Löschen

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



Rot-Schwarz-Bäume: Löschen

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



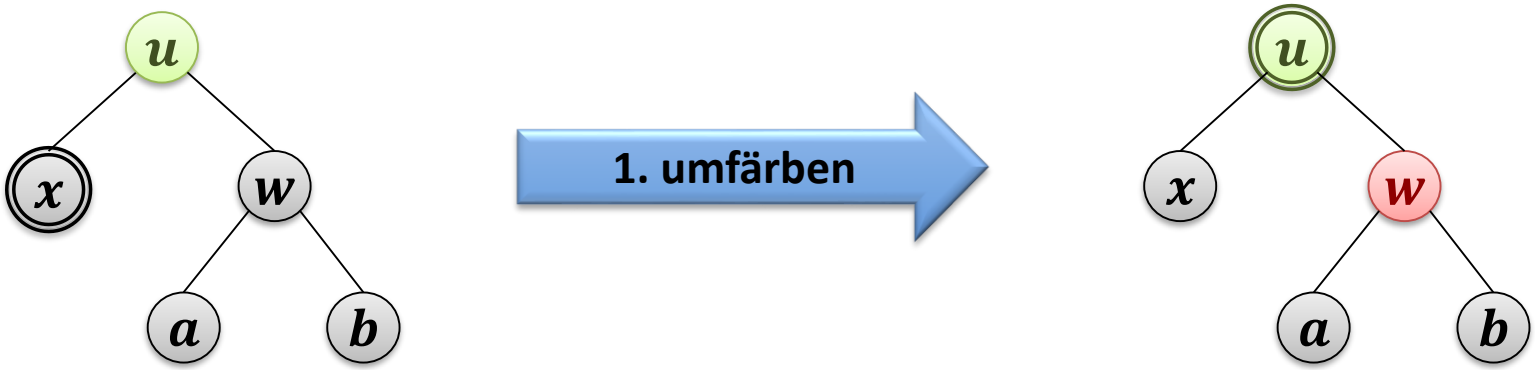
Jetzt sind wir wieder in Fall A.1 (a übernimmt Rolle von w)

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

Rot-Schwarz-Bäume: Löschen

UNI FREIBURG

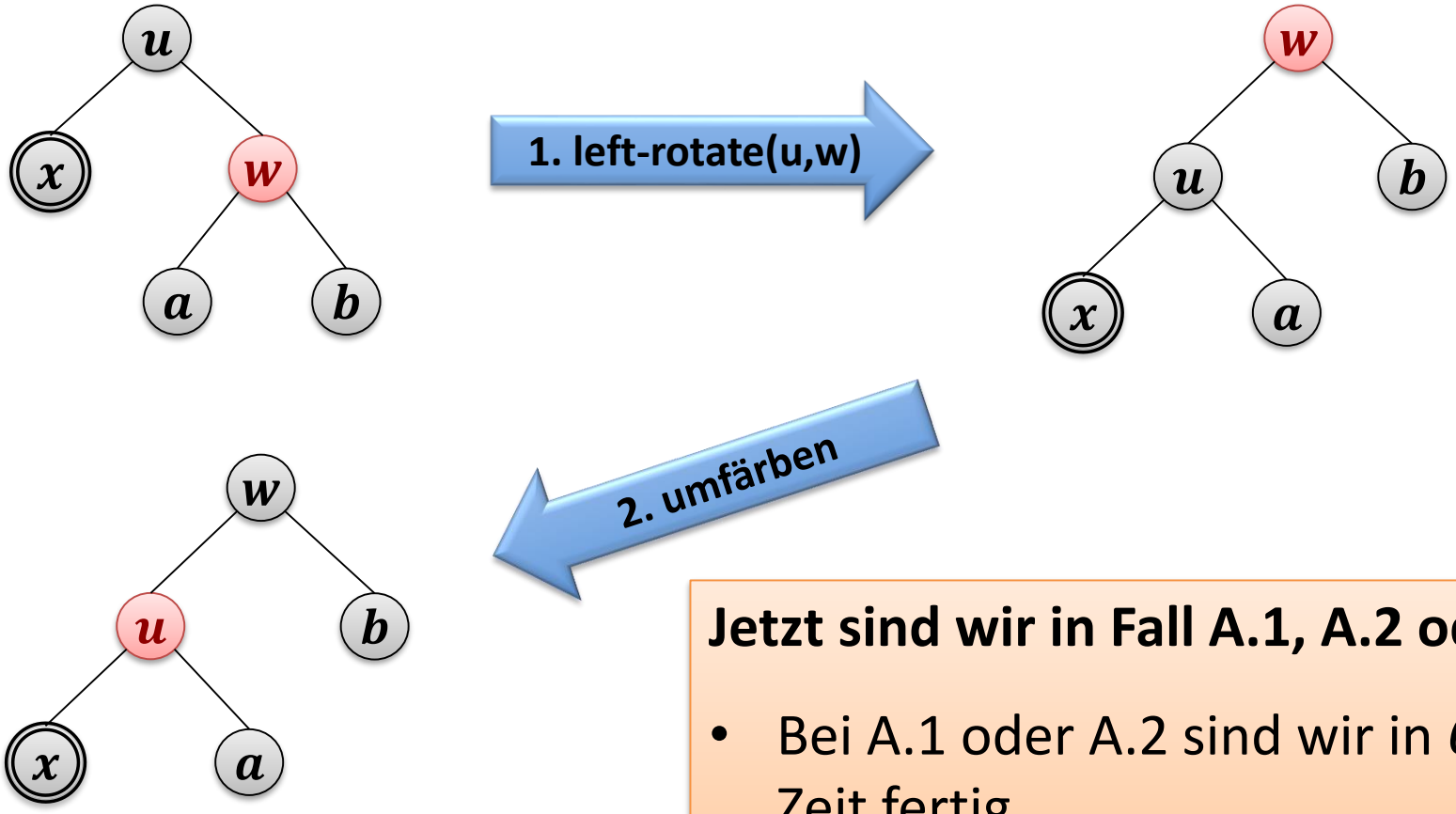
Fall A.3: w ist schwarz, beide Kinder von w sind schwarz



- 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

UNI FREIBURG

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 v schwarz mit zwei NIL-Kindern ist, muss man korrigieren
   - Man hat dann 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:** $O(\text{Baumtiefe}) = O(\log n)$

Zusammenfassung Rot-Schwarz-Bäume

UNI FREIBURG

Rot-Schwarz-Bäume

- Binäre Suchbäume, die immer Tiefe $O(\log n)$ haben.
- Man kann in $O(\log n)$ Zeit ein neues (Schlüssel, Wert)-Paar einfügen oder das Paar zu einem gegebenen Schlüssel löschen.

Dictionary Implementierung

- **Worst-Case Laufzeit $O(\log n)$** für die Operationen

  find, insert, delete, minimum, maximum, predecessor, successor

- Range Query, bei der k (Schlüssel, Wert)-Paare zurückgegeben werden hat Laufzeit $O(k + \log n)$.

Vergleich zu Hashtabellen

- Amortisiert $O(1)$ Laufzeit $\Leftrightarrow$ Worst-Case $O(\log n)$ Laufzeit
- Binäre Suchbäume unterstützen diverse weitere Operationen.

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

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

- 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: 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:

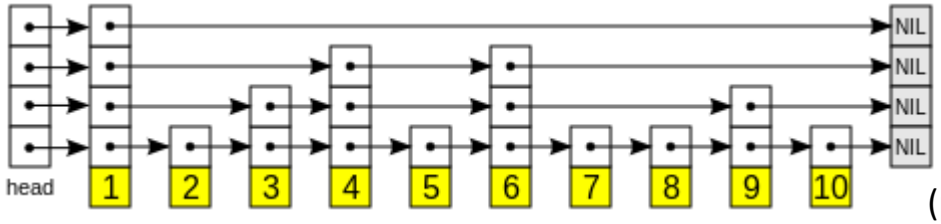
- ähnlich wie Rot-Schwarz-Bäume (nur rechte Kinder können rot sein)

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)