Kapitel 4: Suchen

4.1 Suchbäume

- Datenstrukturen für **Wörterbücher**:
  - dynamische Datenstruktur: Objekte können eingefügt (INSERT-Operation) und gelöscht (DELETE-Operation)
  - Objekte besitzen einen Schlüssel key[], für den die totale Ordnungsrelation $\leq$ definiert ist.
  - Die Datenstruktur erlaubt den geordneten Zugriff auf Objekte:
    - SEARCH(T,x): Objekt mit dem Schlüssel x
    - MINIMUM(T) / MAXIMUM(T): Objekt mit kleinstem/größtem Schlüssel
    - PREDECESSOR(T,x) / SUCCESSOR(T,x): Objekt mit nächst kleinerem/größerem Schlüssel

- **Ziel:** Entwicklung einer Datenstruktur mit asymptotischer Worst-Case Laufzeit O(log N) für alle Operationen bei Speicherung von N Objekten.

Suchbäume

- **Binärer Suchbaum**:

  Geordneter Baum T mit ausgezeichneter Wurzel T.root, die Knoten x haben die Attribute:

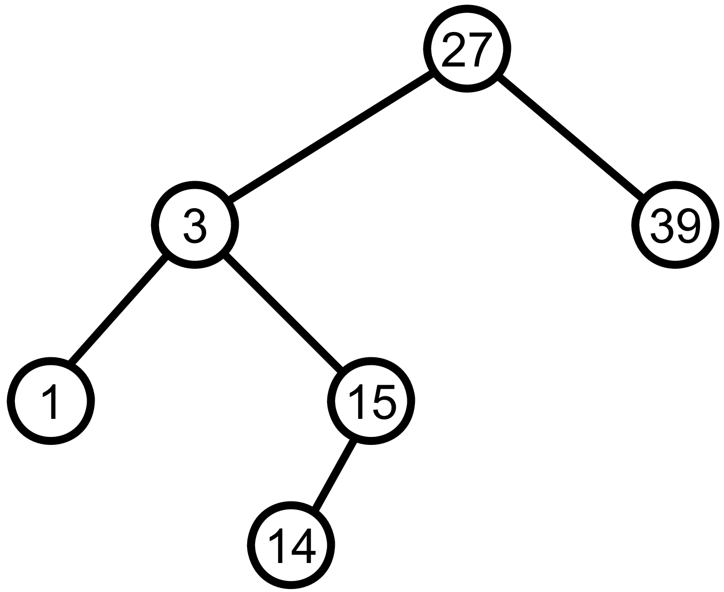
  - x.key : Schlüssel des gespeicherten Objekts
  - x.left : linker Kind-Knoten von x
  - x.right : rechter Kind-Knoten von x
  - x.p : Elter-Knoten von x

- **Binäre Suchbaumeigenschaft:**

  Für jeden Knoten x gilt:

  Sei y ein Knoten im linken Teilbaum von x, dann gilt y.key < x.key.

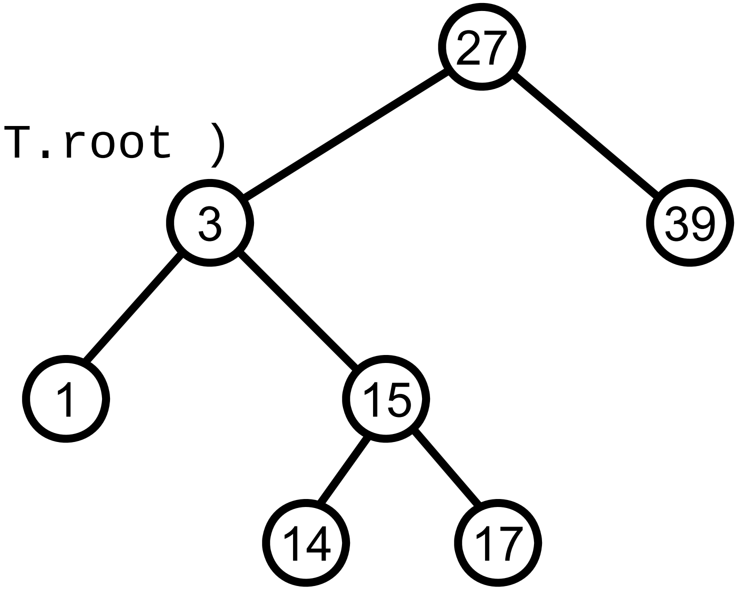
  Sei y ein Knoten im rechten Teilbaum von x, dann gilt y.key ≥ x.key.



Traversieren von Bäumen

- **Traversierung:** Systematisches Durchlaufen aller Knoten eines Baumes
- Mögliche Reihenfolgen:
  - **Preorder:** Wurzel, linker Teilbaum, rechter Teilbaum
  - **Postorder:** linker Teilbaum, rechter Teilbaum, Wurzel
  - **Inorder:** linker Teilbaum, Wurzel, rechter Teilbaum
- INORDER-TREE-WALK(x)

```
// Aufruf mit INORDER-TREE-WALK( T.root )
if  x ≠ NIL
    INORDER-TREE-WALK( x.left )
    print x.key
    INORDER-TREE-WALK( x.right )
```



Inorder: 1, 3, 14, 15, 17, 27, 39

Traversierung von Bäumen

- **Theorem 1:** Erfüllt ein Baum T die binäre Suchbaumeigenschaft, dann gibt die Funktion INORDER-TREE-WALK() die Objekte nach aufsteigenden Schlüsselwerten aus.
- Beweis: Für jedes Objektpaar x, y mit x.key $\leq$ y.key gilt: Sei p der kleinste gemeinsame Vorfahr von x und y.
  - Fall 1: x=p   y befindet sich im rechten Teilbaum.
  - Fall 2: y=p   x befindet sich im linken Teilbaum.
  - Fall 3: x$\neq$p$\neq$y   x befindet sich im linken, y im rechten Teilbaum.
- **Theorem 2:** INORDER-TREE-WALK() benötigt auf einem Baum mit N Knoten die Laufzeit O(N)
- Beweis: Sei k die Größe des linken Teilbaums. Es gilt die Rekurrenzgleichung:

$$T(N) = \begin{cases} c & : \quad N \leq 1 \\ T(k) + T(N-k-1) + d & : \quad \text{sonst} \end{cases}$$

Es gilt $T(N) = (c+d)N + c = O(N)$. (Beweis durch Induktion)

Suchen in Suchbäume

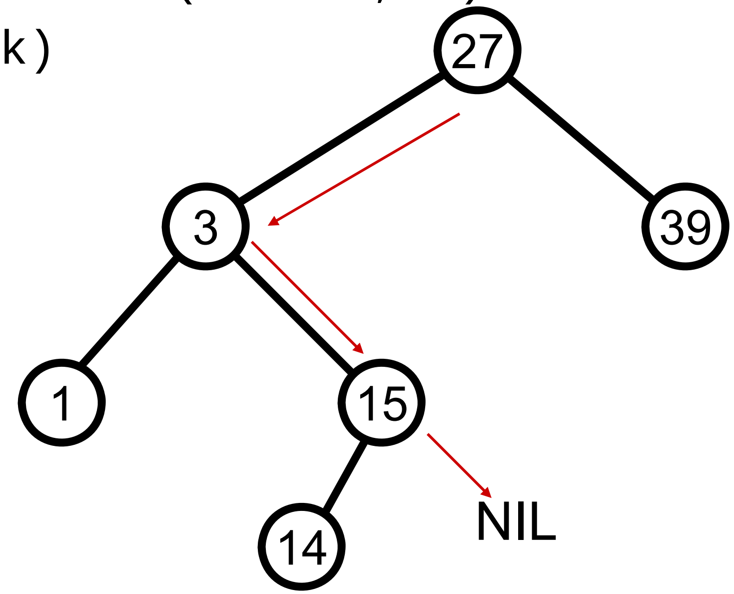
- `TREE-SEARCH(x,k)`

```
 if( x == NIL or k == x.key ) return x
 elseif( k < key[x] ) return TREE-SEARCH(x.left, k)
 else return TREE-SEARCH(x.right, k)
```

- `ITERATIVE-TREE-SEARCH(x,k)`

```
while( x ≠ NIL and k ≠ x.key )
    if( k < key[x] ) x = x.left
    else x = x.right
return x
```

- Bsp: TREE-SEARCH(root[T],17):

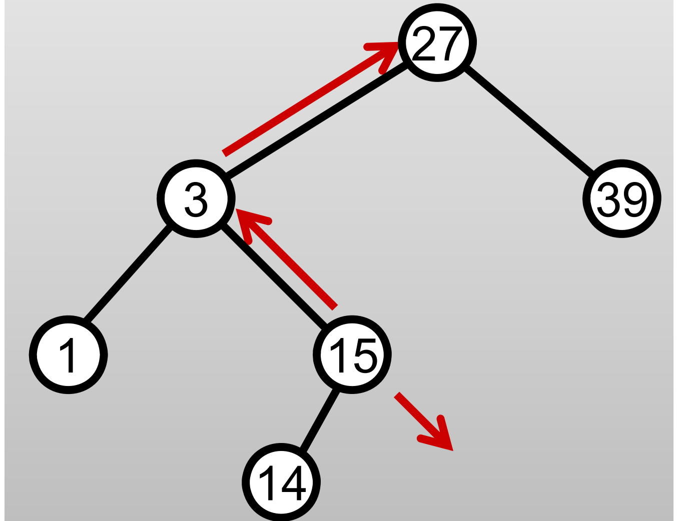


- **Theorem 3:** Sei h die Höhe des Suchbaum T, dann hat TREE-SEARCH(root[T],k) Laufzeit O(h).
- Beweis: TREE-SEARCH durchläuft einen Pfad von der Wurzel bis zu einem Blatt mit konstanter Laufzeit pro besuchten Knoten.

Minimum- und Successor-Funktion

- `TREE-MINIMUM(x)`
```
// Aufruf mit TREE-MINIMUM(root[T])
while( x.left ≠ NIL ) x = x.left
return x
```
- `TREE-SUCCESSOR(x)`
```
// finde Knoten y mit key[y] > key[x], key[y] minimal
if( x.right ≠ NIL )
  return TREE-MINIMUM( x.right )
else
  y = x.p
  while( y ≠ NIL and x == y.right )
    x = y; y = p[y]
return y
```

Beispiel:
TREE-SUCCESSOR(15)



- **Theorem 4:** Sei h die Höhe des Suchbaum T, dann hat TREE-MINIMUM(root[T]) Laufzeit O(h); TREE-SUCCESSOR() für jeden Knoten x des Suchbaums Worst-Case Laufzeit O(h).
  - Beweis: Für jeden Knoten x durchläuft TREE-SUCCESSOR entweder den Pfad von x zu einem Blatt oder zur Wurzel.

Korrektheit von TREE-SUCCESSOR()

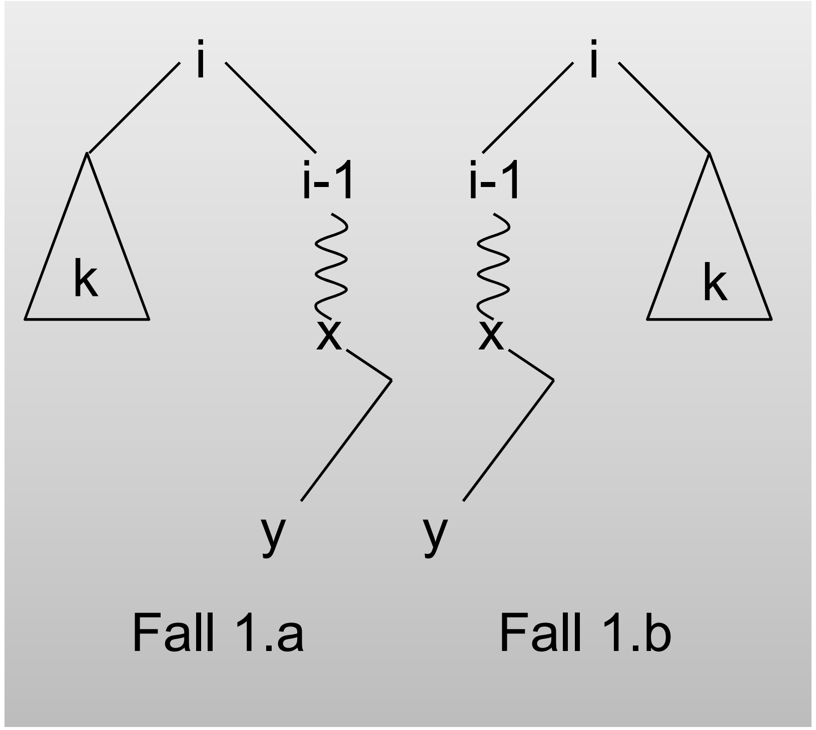
- **Theorem 5:** TREE-SUCCESSOR(x) bestimmt Knoten y mit den Eigenschaften $y.key \geq x.key$ und y.key minimal.
- Beweis: Sei k.T der Teilbaum unter Knoten k. Sei $x = p_0, p_1, \ldots, p_l =$ T.root der Pfad von x zur Wurzel.
  - **Fall 1**: x hat einen rechten Nachfolger: Sei $y \in x.right.T$ mit y.key minimal. $\forall\, i \in \{1,\ldots,l\}$ gilt:

    Fall 1.a: $p_{i-1} = p_i.right$:

    $\forall\, k \in p_i.left.T: k.key \leq p_i.key \leq x.key$

    Fall 1.b: $p_{i-1} = p_i.left$:

    $\forall\, k \in p_i.right.T: k.key \geq p_i.key \geq y.key$



Korrektheit von TREE-SUCCESSOR()

- **Fall 2**: x hat keinen rechten Nachfolger.
  Sei y = p_j mit $p_{j-1} = p_j$.left, j minimal.

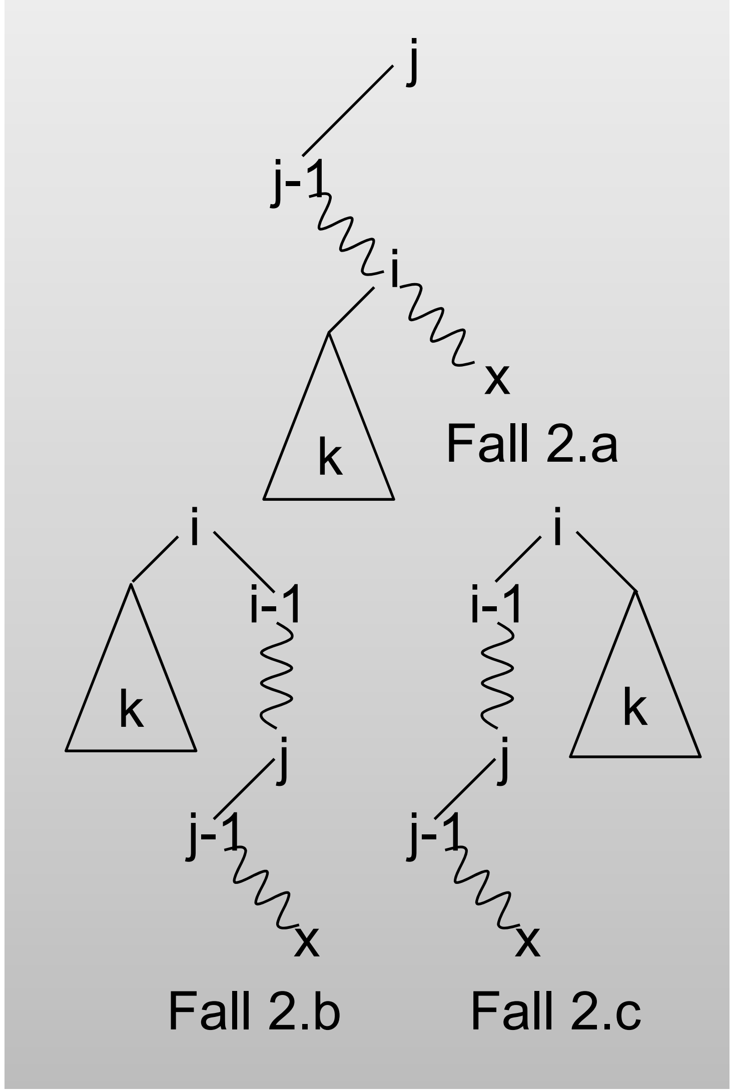
  Fall 2.a: $\forall$ i $\in$ {1,…,j-1}:
  $\forall$ k $\in p_i$.left.T: k.key $\leq p_i$.key $\leq$ x.key

  Fall 2.b: $\forall$ i $\in$ {j+1,…,l} mit $p_{i-1} = p_i$.right:
  $\forall$ k $\in$ p_i.left.T: k.key $\leq p_i$.key $\leq$ x.key

  Fall 2.c: $\forall$ i $\in$ {j+1,…,l} mit $p_{i-1} = p_i$.left:
  $\forall$ k $\in$ p_i.right.T: k.key $\geq p_i$.key $\geq$ y.key

  Desweiteren gilt
  $\forall$ k $\in$ p_j.right.T: k.key $\geq p_j$.key $\geq$ y.key

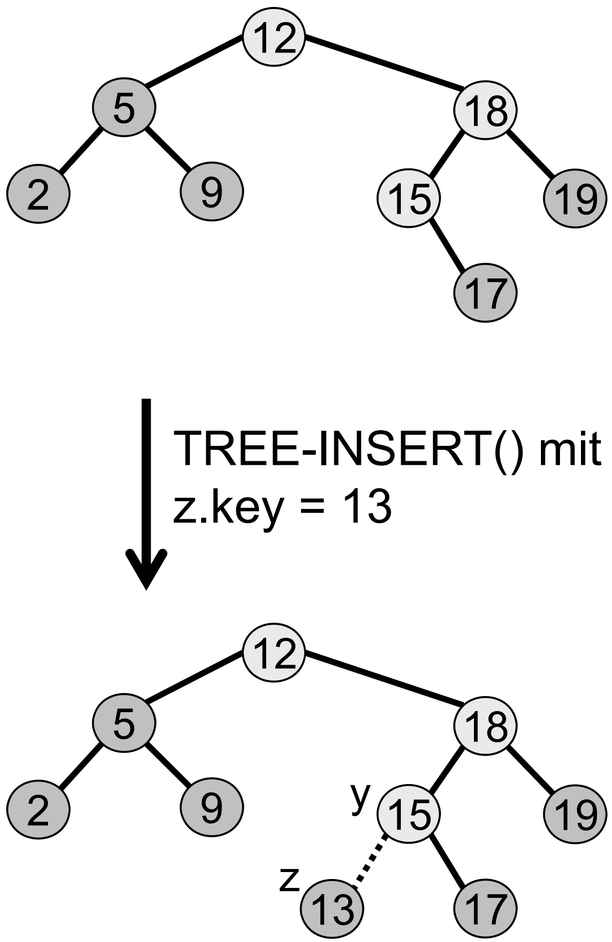
Da TREE-SUCCESSOR() y gemäß Fall 1 oder 2 berechnet, folgt das Theorem.



Einfügen in Suchbäume

- Idee: laufe unter Berücksichtigung der Suchbaumeigenschaft von der Wurzel zu einem Blatt, hänge das neue Objekt an.
- `TREE-INSERT(T, z)`

```
x = T.root
y = NIL    // speichert x.p
while( x ≠ NIL )
    y = x
    if( z.key < x.key ) x = x.left
    else x = x.right
p[z] = y
if( y == NIL ) root[T] = z
elseif( z.key < y.key ) y.left = z
else y.right = z
```



Einfügen in Suchbäume

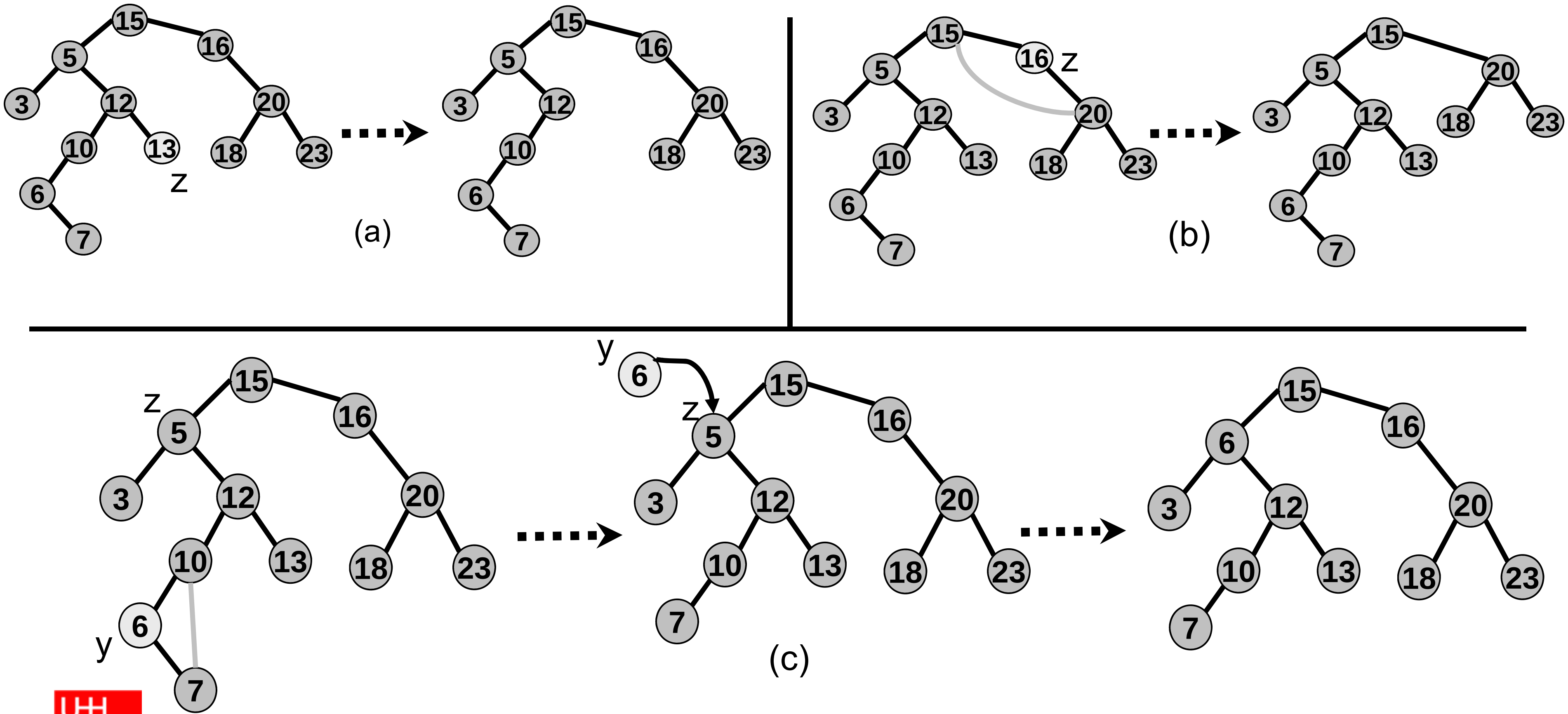
- `REC-TREE-INSERT(x, z)`

  ```
  // Aufruf mit REC-TREE-INSERT(root[T], z) für nicht-leere Bäume
  if( z.key < x.key )
    if( x.left ≠ NIL ) REC-TREE-INSERT(x.left, z)
    else z.p = x; x.left = z
  else
    if( x.right ≠ NIL ) REC-TREE-INSERT(x.right, z)
    else z.p = x; x.right = z
  ```

- **Theorem 6:** Sei h die Höhe des Suchbaum T, dann hat TREE-INSERT(T, z) und REC-TREE-INSERT(T.root, z) die asymptotische Worst-Case Laufzeit O(h).

Löschen aus Suchbäumen

- Löschen eines Knotens z unter Erhalt der Suchbaum-Eigenschaft:
  - Fall 1: z ist ein Blatt: Lösche z
  - Fall 2: z hat nur einen Nachfolger x: x wird Nachfolger von z.p
  - Fall 3: z hat zwei Nachfolger: Ersetze z durch TREE-SUCCESSOR(z)



Löschen aus Suchbäumen

- `TREE-DELETE-NODE(T, z)` // Cormen: TREE-DELETE()

  // y: der (nach Tausch) zu löschende Knoten

```
if( z.left == NIL or z.right == NIL ) y = z
else y = TREE-MINIMUM( z.right )   // Cormen: TREE-SUCCESSOR(z)
// x: Kind von y, das nicht NIL ist
if( y.left ≠ NIL ) x = y.left
  else x = y.right
// entferne y aus der Baumstruktur
if( x ≠ NIL ) x.p = y.p
if( y.p == NIL ) T.root = x
else
    if( y == y.p.left ) y.p.left = x
    else y.p.right = x
// kopiere Daten von y nach z
if( y ≠ z ) z.key = y.key
delete y
```

- `TREE-DELETE(T, k )`

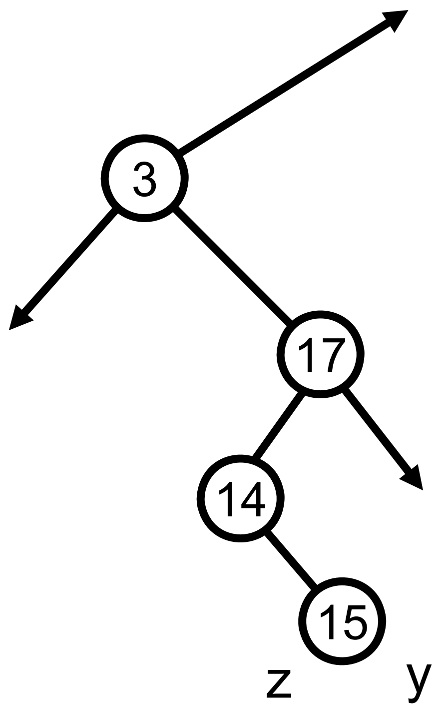
```
TREE-DELETE-NODE( T, TREE-SEARCH(T.root, k) )
```

Löschen aus Suchbäumen

- TREE-DELETE-NODE(T, z)

```
// y: der (nach Tausch) zu löschende Knoten
1 if( z.left == NIL or z.right == NIL )
    y = z
2 else y = TREE-MINIMUM( z.right )
// x: Kind von y, das nicht NIL ist
3 if( y.left ≠ NIL ) x = y.left
  else x = y.right
// entferne y aus der Baumstruktur
4 if( x ≠ NIL ) x.p = y.p
5 if( y.p == NIL ) T.root = x
6 else
    if( y == y.p.left ) y.p.left = x
    else y.p.right = x
// kopiere Daten von y nach z
8 if( y ≠ z ) z.key = y.key
9 delete y
```

Fall 1:



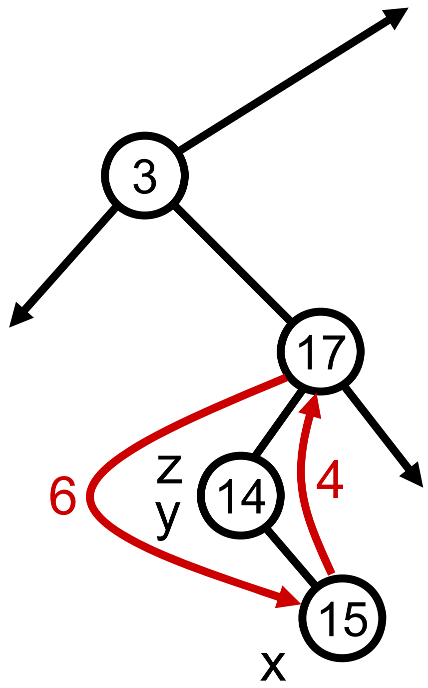
x = NIL

Löschen aus Suchbäumen

- TREE-DELETE-NODE(T, z)

```
// y: der (nach Tausch) zu löschende Knoten
1 if( z.left == NIL or z.right == NIL )
    y = z
2 else y = TREE-MINIMUM( z.right )
// x: Kind von y, das nicht NIL ist
3 if( y.left ≠ NIL ) x = y.left
  else x = y.right
// entferne y aus der Baumstruktur
4 if( x ≠ NIL ) x.p = y.p
5 if( y.p == NIL ) T.root = x
6 else
    if( y == y.p.left ) y.p.left = x
    else y.p.right = x
// kopiere Daten von y nach z
8 if( y ≠ z ) z.key = y.key
9 delete y
```

Fall 2:

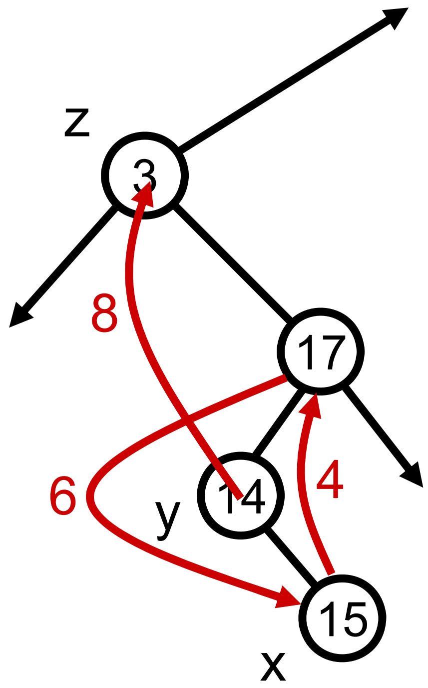


Löschen aus Suchbäumen

- TREE-DELETE-NODE(T, z)

```
// y: der (nach Tausch) zu löschende Knoten
1 if( z.left == NIL or z.right == NIL )
     y = z
2 else y = TREE-MINIMUM( z.right )
// x: Kind von y, das nicht NIL ist
3 if( y.left ≠ NIL ) x = y.left
  else x = y.right
// entferne y aus der Baumstruktur
4 if( x ≠ NIL ) x.p = y.p
5 if( y.p == NIL ) T.root = x
6 else
     if( y == y.p.left ) y.p.left = x
     else y.p.right = x
// kopiere Daten von y nach z
8 if( y ≠ z ) z.key = y.key
9 delete y
```

Fall 3:



Suchbäume

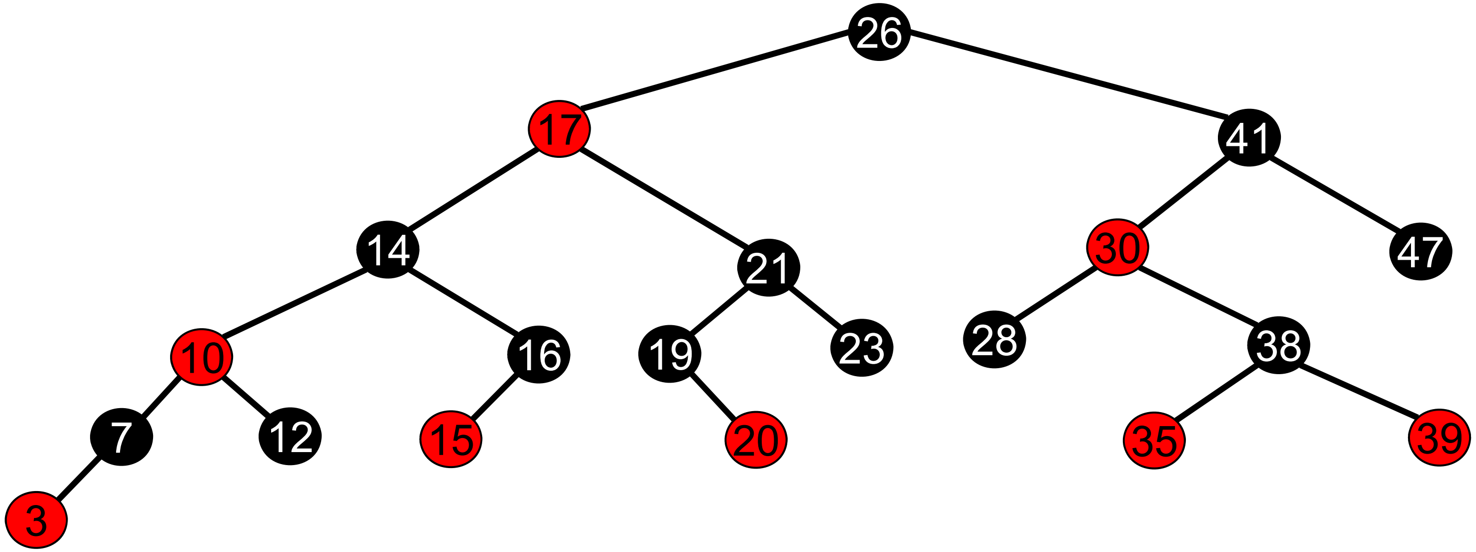
- **Theorem 7:** Sei h die Höhe des Suchbaum T, dann hat TREE-DELETE(T, k) die asymptotische Worst-Case Laufzeit O(h).
  - TREE-SEARCH durchläuft einen Pfad von der Wurzel zum zu löschenden Knoten z.
  - TREE-DELETE-NODE durchläuft einen Pfad vom zu löschenden Knoten z zu einem Blatt.

- Alle Wörterbuch-Operationen können im Suchbaum in Laufzeit O(h) realisiert werden.
- Definition: Ein Suchbaum T mit Höhe h(T) heißt **balanciert**, g.d.w. $h(T) = O(\log |T|)$ gilt.

- **Welche Höhe hat ein Suchbaum?**

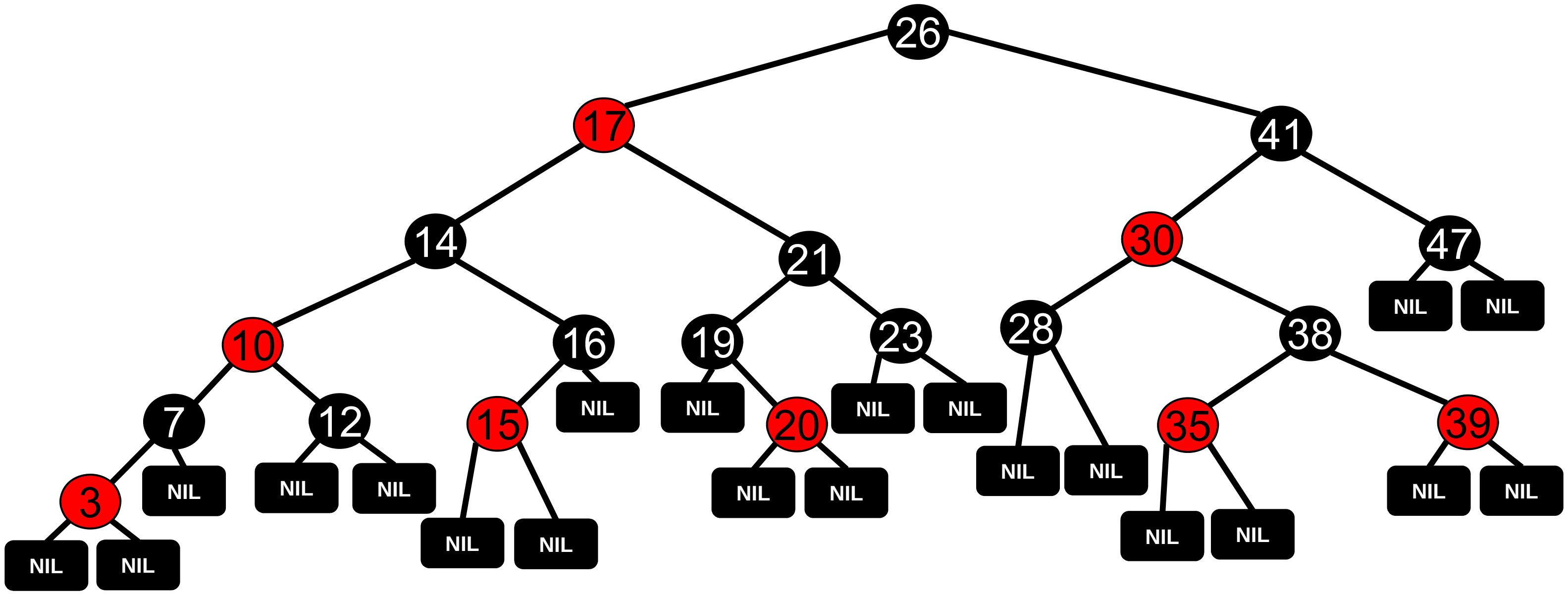
Best Case	Zufällig erzeugter Binärbaum	Worst Case
$O(\log N)$	$O(\log N)$	$\Omega(N)$

4.2 Rot-Schwarz-Bäume



- **Rot-Schwarz-Baume**: Suchbäume mit den Knoten-Attributen:
  - x.p : Vorgänger
  - x.left, x.right : linker und rechter Nachfolger
  - x.key : Schlüsselelement
  - x.color : Farbe (rot oder schwarz)
  - color erfüllt die *Rot-Schwarz-Eigenschaften*
- Blätter werden durch den Wächter NIL= T.nil repräsentiert
- Definition: Sei T(x) der Teilbaum unter x, h(T) die Höhe eines Baumes

Rot-Schwarz-Bäume



- **Rot-Schwarz-Eigenschaften:**
  1. Jeder Knoten ist entweder rot oder schwarz.
  2. Die Wurzel ist schwarz.
  3. Jedes Blatt (der Wächter) ist schwarz.
  4. Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.
  5. Für jeden Knoten x gilt: Alle Pfade von x zu einem Blatt enthalten die gleiche Anzahl bh(x) schwarzer Knoten.
- Definition: bh(x): Schwarz-Höhe des Knotens x

Höhe von Rot-Schwarz-Bäumen

- **Lemma**: Ein Rot-Schwarz-Baum T mit N inneren Knoten hat höchstens die Höhe $2\log_2(N+1)$.
- Beweis:
  - **Teil 1:** Zeige $\forall$ x : $|T(x)| \geq 2^{bh(x)} - 1$ durch Induktion über h(T(x))

    Für x mit h(T(x))=0 gilt: bh(x)=0, $1 = |T(x)| \geq 2^0 - 1 = 0$

    Sei x ein Knoten mit h(T(x)) > 0:
    - **Fall 1:** x.color = rot

      => x.left.color = x.right.color = schwarz und somit

      bh(x.left) = bh(x.right) = bh(x)

      $|T(x)| = 1 + |T(x.left)| + |T(x.right)| \geq 1 + 2^{bh(x)}-1 + 2^{bh(x)}-1 \geq 2^{bh(x)}-1$
    - **Fall 2:** x.color = schwarz

      bh(x.left)= bh(x.right) = bh(x) -1

      $|T(x)| = 1 + |T(x.left)| + |T(x.right)| \geq 1 + 2^{bh(x)-1}-1 + 2^{bh(x)-1}-1 = 2^{bh(x)}-1$

  - **Teil 2:**

    Es gilt $h(T) \leq 2\, bh(root[T])$ (nach Rot-Schwarz-Eigenschaft 4)

    => $N = |T| \geq 2^{bh(root[T])} - 1 \geq 2^{h(T)/2} - 1 \iff h(T) \leq 2 \log_2(N + 1)$

Such-Komplexität in Rot-Schwarz-Bäumen

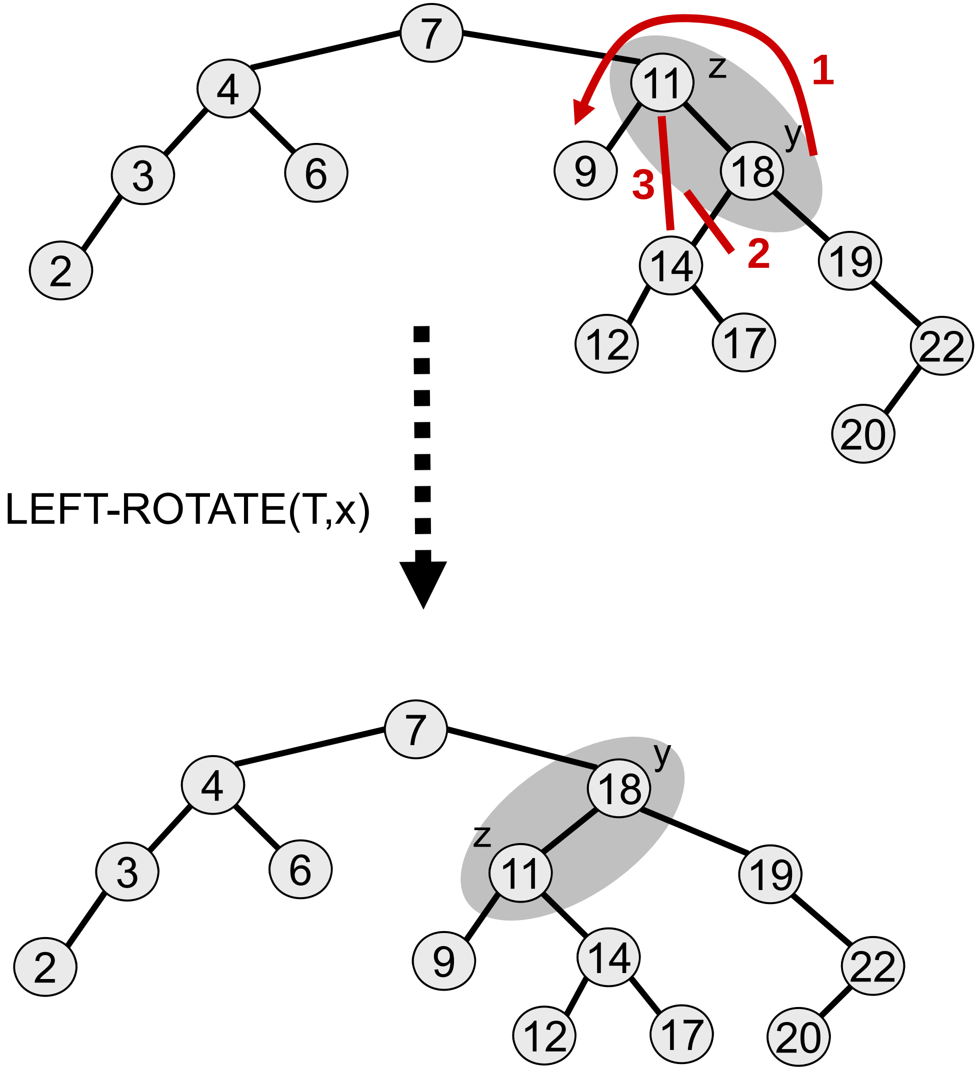
- Ein Rot-Schwarz-Baum mit n Knoten hat eine maximale Höhe von O(log N) und ist somit balanciert.
- Sei T ein Suchbaum, der die Rot-Schwarz-Eigenschaft erfüllt.
  - Die Operationen SEARCH(), MINIMUM(), MAXIMUM(), SUCCESSOR() und PREDECESSOR() laufen in Zeit

    O(h) = O(log N)
  - Die Operationen TREE-INSERT() und TREE-DELETE() laufen in Zeit O(log N)

- TREE-INSERT() und TREE-DELETE() können zu Suchbäumen führen, die die Rot-Schwarz-Eigenschaft verletzen!

  ➔ Korrektur-Mechanismus zur Wiederherstellung der Rot-Schwarz-Eigenschaft (und somit der Balance) wird benötigt.

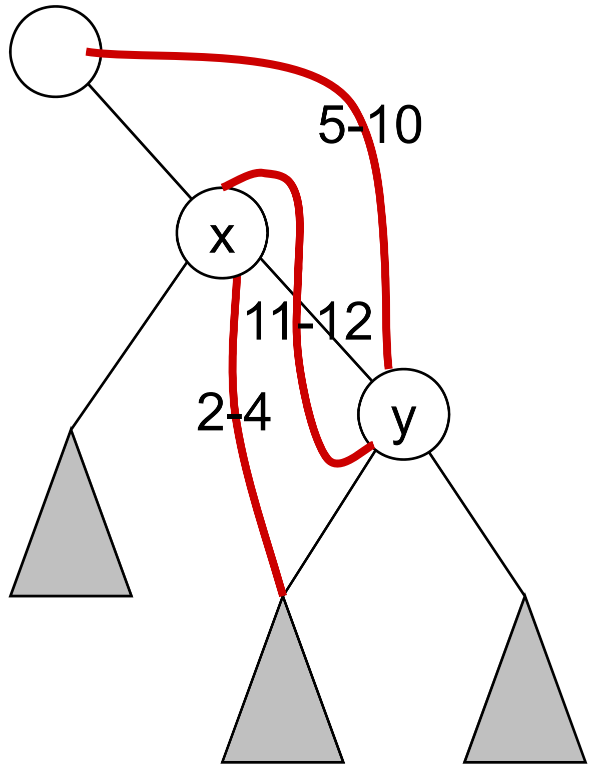
Rotationen in Suchbäumen



Rotationen in Suchbäumen

- Definition: Eine **Rotation** ist eine lokale Operation (Umsetzen einer konstanten Anzahl von Zeigern) in Suchbäumen, die die Suchbaum-Eigenschaft erhält.
- LEFT-ROTATE(T, x)

```
1  y = x.right
// y's linker Teilbaum → x's rechter Teilbaum
2  x.right = y.left
3  if( y.left ≠ T.nil )
     y.left.p = x
// verbinde Vater von x mit y
5  y.p = x.p
6  if( x.p == T.nil )
     T.root = y
8  else if( x = x.p.left )
         x.p.left = y
10     else x.p.right = y
// verschiebe x auf die linke Seite von y
11 y.left = x
12 x.p = y
```



Einfügen in Rot-Schwarz-Bäumen

- Durchlaufe den Suchbaum von der Wurzel bis zu einem Blatt, füge den Schlüssel ein, färbe den Knoten rot.
- RB-INSERT(T, z)

```
x = T.root
y = T.nil  // speichert p[x]
while( x ≠ T.nil )
  y = x
  if( z.key < x.key ) x = x.left
  else x = x.right
z.p = y
if( y == T.nil ) T.root = z
else if( z.key < y.key ) y.left = z
     else y.right = z
z.left = z.right = T.nil
z.color = ROT
// korrigiere die Rot-Schwarz-Eigenschaften
RB-INSERT-FIXUP(T, z)
```

Korrektur der Rot-Schwarz-Eigenschaft

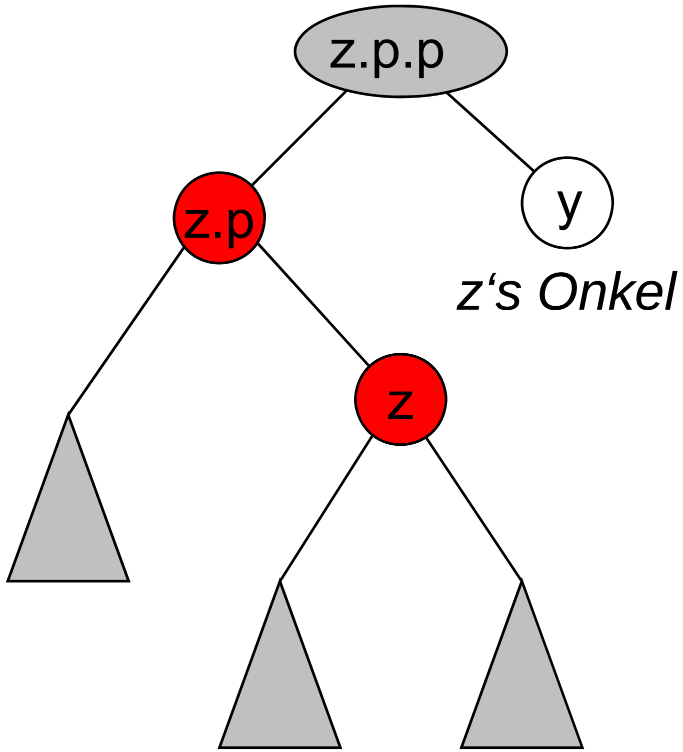
- Welche Eigenschaften können verletzt sein?
  1. Jeder Knoten ist entweder rot oder schwarz. √
  2. Die Wurzel ist schwarz.
  3. Jedes Blatt (der Wächter) ist schwarz. √
  4. Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.
  5. Für jeden Knoten x gilt: Alle Pfade von x zu einem Blatt enthalten die gleiche Anzahl bh(x) schwarzer Knoten. √
- zu 2.: Ist verletzt, falls z die Wurzel ist
- zu 4.: Ist verletzt, falls z.p.color = ROT ist
- Die Rot-Schwarz-Eigenschaften sind höchstens an einem Knoten (Eigenschaft 2: z oder Eigenschaft 4: z.p) verletzt.
- Idee von RB-INSERT-FIXUP:

  Korrigiere die Eigenschaftsverletzung 4 lokal oder verschiebe die Eigenschaftsverletzung 4 sukzessiv zum Vorgänger. Wenn die Wurzel erreicht wird, korrigiere Eigenschaft 2.

Die Funktion RB-INSERT-FIXUP()

- RB-INSERT-FIXUP(T, z)

```
while( z.p.color == ROT )
  if( z.p == z.p.p.left )
    y = z.p.p.right
    if( y.color == ROT )
      z.p.color = SCHWARZ
      y.color = SCHWARZ
      z.p.p.color = ROT
      z = z.p.p
    else
      if( z == z.p.right )
        z = z.p
        LEFT-ROTATE(T, z)
      z.p.color = SCHWARZ
      z.p.p.color = ROT
      RIGHT-ROTATE(T, z.p.p )
  else // z.p == z.p.p.right
     :   // wie then-Teil, vertausche right ⇔left
T.root.color = SCHWARZ
```



Die Funktion RB-INSERT-FIXUP() – Fall 1

- RB-INSERT-FIXUP(T, z)

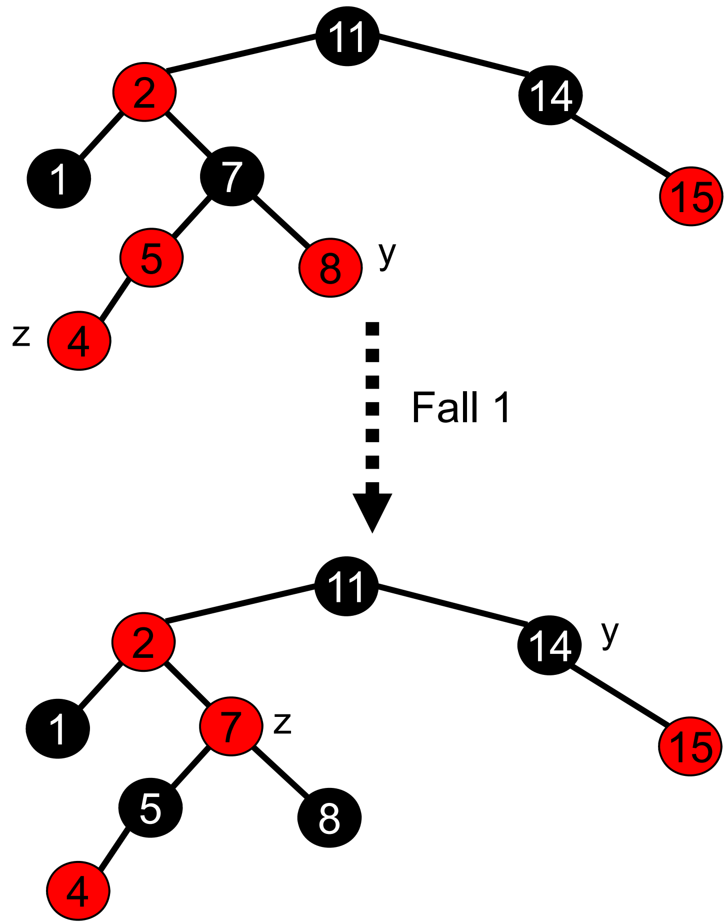
```
while( z.p.color == ROT )
  if( z.p == z.p.p.left )
    y = z.p.p.right
    if( y.color == ROT )
      z.p.color = SCHWARZ
      y.color = SCHWARZ
      z.p.p.color = ROT
      z = z.p.p
    else
      if( z == z.p.right )
        z = z.p
        LEFT-ROTATE(T, z)
      z.p.color = SCHWARZ
      z.p.p.color = ROT
      RIGHT-ROTATE(T, z.p.p )
  else // z.p == z.p.p.right
   :   // wie then-Teil, vertausche right ⇔left
T.root.color = SCHWARZ
```

Fall 1

Fall 2

Fall 3

Fall 4-6 (symmetrisch zu 1-3)



Die Funktion RB-INSERT-FIXUP() – Fall 2

- RB-INSERT-FIXUP(T, z)

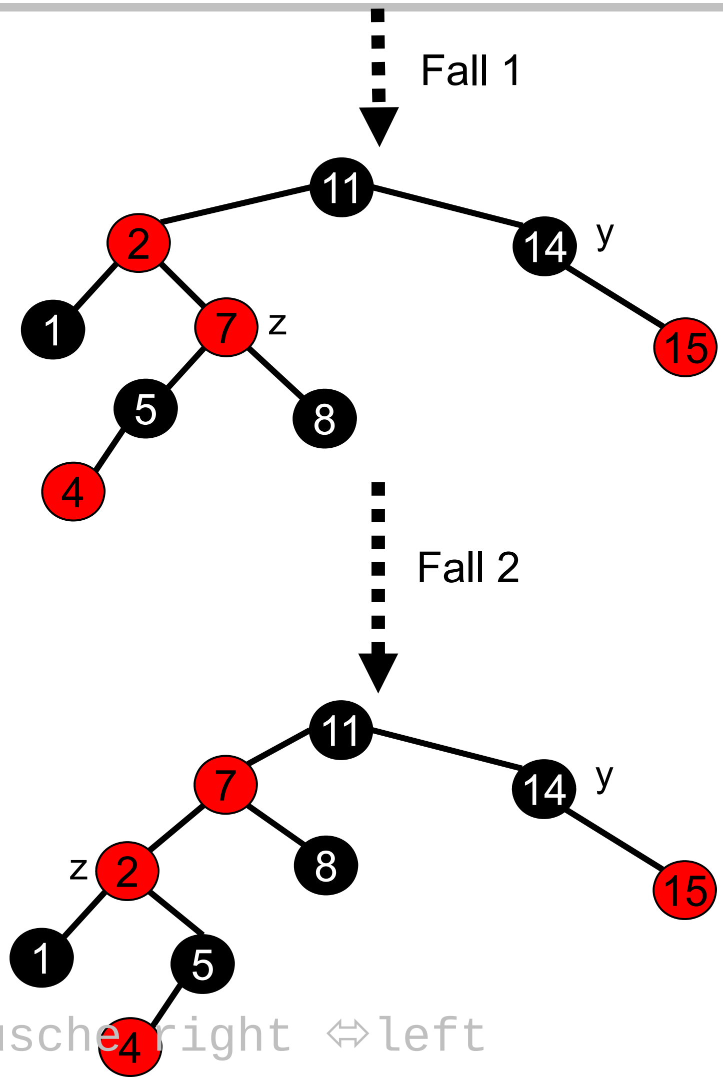
Fall 4-6 (symmetrisch zu 1-3)

Fall 1

Fall 2

Fall 3

```
while( z.p.color == ROT )
  if( z.p == z.p.p.left )
    y = z.p.p.right
    if( y.color == ROT )
      z.p.color = SCHWARZ
      y.color = SCHWARZ
      z.p.p.color = ROT
      z = z.p.p
    else
      if( z == z.p.right )
        z = z.p
        LEFT-ROTATE(T, z)
      z.p.color = SCHWARZ
      z.p.p.color = ROT
      RIGHT-ROTATE(T, z.p.p )
  else // z.p == z.p.p.right
   :    // wie then-Teil, vertausche right ⇔ left
T.root.color = SCHWARZ
```



Die Funktion RB-INSERT-FIXUP() – Fall 3

- RB-INSERT-FIXUP(T, z)

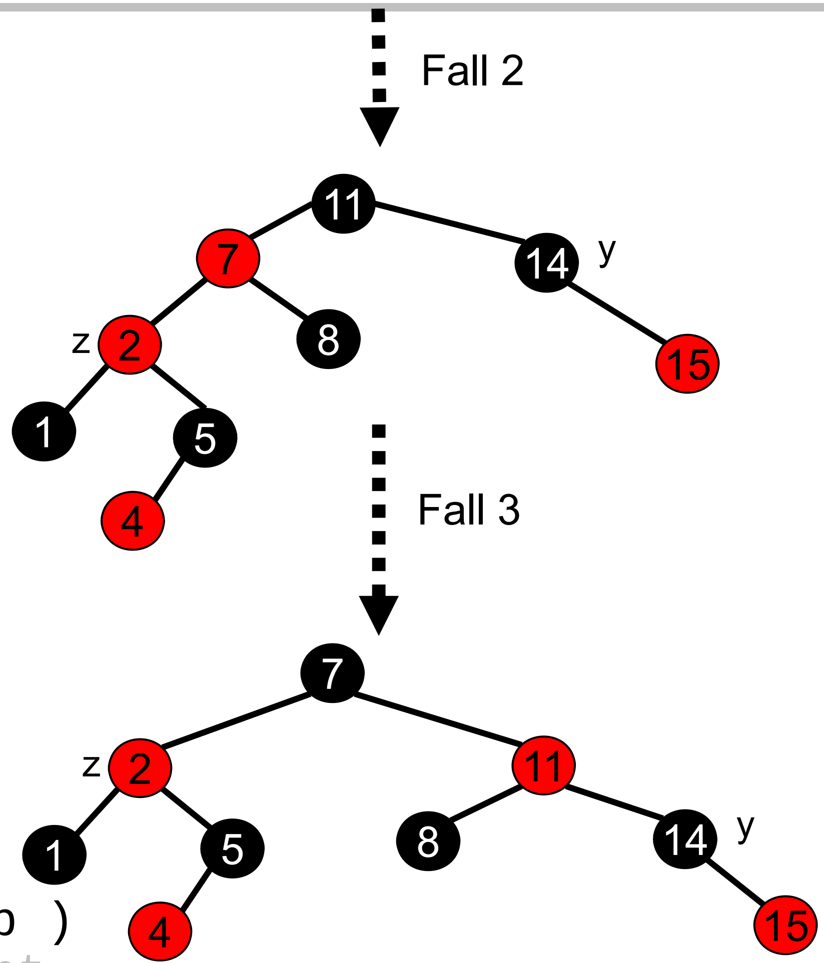
```
while( z.p.color == ROT )
  if( z.p == z.p.p.left )
    y = z.p.p.right
    if( y.color == ROT )
      z.p.color = SCHWARZ
      y.color = SCHWARZ
      z.p.p.color = ROT
      z = z.p.p
    else
      if( z == z.p.right )
        z = z.p
        LEFT-ROTATE(T, z)
      z.p.color = SCHWARZ
      z.p.p.color = ROT
      RIGHT-ROTATE(T, z.p.p )
  else // z.p == z.p.p.right
   :    // wie then-Teil, vertausche right ⇔left
T.root.color = SCHWARZ
```

Fall 1

Fall 2

Fall 3

Fall 4-6 (symmetrisch zu 1-3)



Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Invariante der while-Schleife 1-15:**
  1. z.color = ROT
  2. Falls z.p = T.root, gilt: z.p.color = SCHWARZ
  3. Es gibt maximal eine Verletzung der Rot-Schwarz-Eigenschaft:
     1. Eigenschaft 2 => z = T.root und z.color = ROT
     2. Eigenschaft 4 => z.color = ROT und z.p.color = ROT

- Gültigkeit der Invariante zum Zeitpunkt der **Initialisierung**:
  - zu 1.: Knoten z mit z.color = ROT wurde in einen Rot-Schwarz-Baum ohne Verletzung eingefügt.
  - zu 2.: Falls z.p = T.root: z.p.color wurde nicht verändert, da T ein Rot-Schwarz-Baum war, gilt z.p.color = SCHWARZ.
  - zu 3.: Offensichtlich gelten Rot-Schwarz-Eigenschaften 1, 3 und 5 (siehe Folie 25).

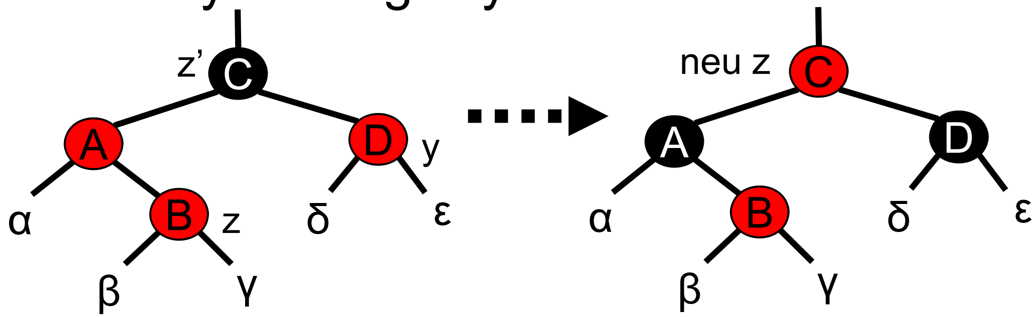
Die Funktion RB-INSERT-FIXUP() - Korrektheit

- Korrektheit von RB-INSERT-FIXUP() zum Zeitpunkt der **Terminierung**:
  - Schleifenbedingung nicht mehr erfüllt: z.p.color = SCHWARZ
  - [Falls z die Wurzel ist, ist z.p = T.nil (Wächter) mit T.nil.color = SCHWARZ]

    Rot-Schwarz-Eigenschaften (zur Erinnerung)
    1. *Jeder Knoten ist entweder rot oder schwarz.*
    2. *Die Wurzel ist schwarz.*
    3. *Jedes Blatt (der Wächter) ist schwarz.*
    4. *Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.*
    5. *Für jeden Knoten x gilt: Alle Pfade von x zu einem Blatt enthalten die gleiche Anzahl bh(x) schwarzer Knoten.*
  - Rot-Schwarz-Eigenschaften 1,3 und 5 sind erfüllt.
  - z.color = ROT und z.p.color = SCHWARZ => Eigenschaft 4. ist nicht verletzt.
  - Eigenschaft 2 kann nach Terminierung der Schleife verletzt sein. Zeile 16 von RB-INSERT-FIXUP() stellt Eigenschaft 2 her.

Die Funktion RB-INSERT-FIXUP() - Korrektheit

- Die Invariante bleibt bei der **Fortsetzung** der Schleife erhalten:
  - Ist z.p.p definiert? z.p.color = ROT, mit Teil 2. der Invariante gilt z.p $\neq$ T.root, damit existiert z.p.p.
- **Fall 1:** Für Onkel y von z gilt: y.color = ROT



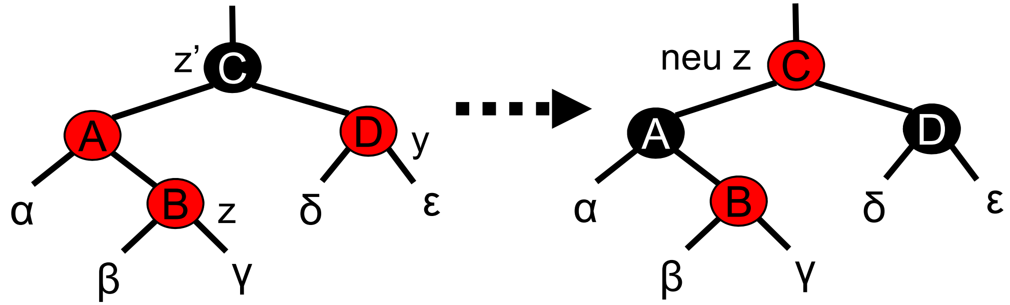
  - z.p.p.color = SCHWARZ, da y.color = ROT und y.p = z.p.p gilt
  - Alle Nachfolger α-ε von z, z.p (außer z) und y sind schwarz und haben die gleiche Schwarz-Höhe bh().
  - Sei z' der Wert der Variablen z nach Ausführung der Iteration:
    1. *Zeige: z.color = ROT*

       z' = z.p.p und z.p.p.color = ROT nach Zeilen 7 und 8.
    2. *Zeige: Falls z.p = T.root, gilt: z.p.color = SCHWARZ*

       z'.p = z.p.p.p ändert seine Farbe nicht.

Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Fall 1:** Für Onkel y von z gilt: y.color = ROT



  - Fortsetzung:

3. *Zeige: Es gibt maximal eine Verletzung der Rot-Schwarz-Eigenschaft:*
   1. *Eigenschaft 2 => T.z = root und z.color = ROT*
   2. *Eigenschaft 4 => z.color = ROT und z.p.color = ROT*

Eigenschaft 1., 3. und 5. (siehe Zeichnung) sind nicht verletzt.

Annahme: z‘ = T.root

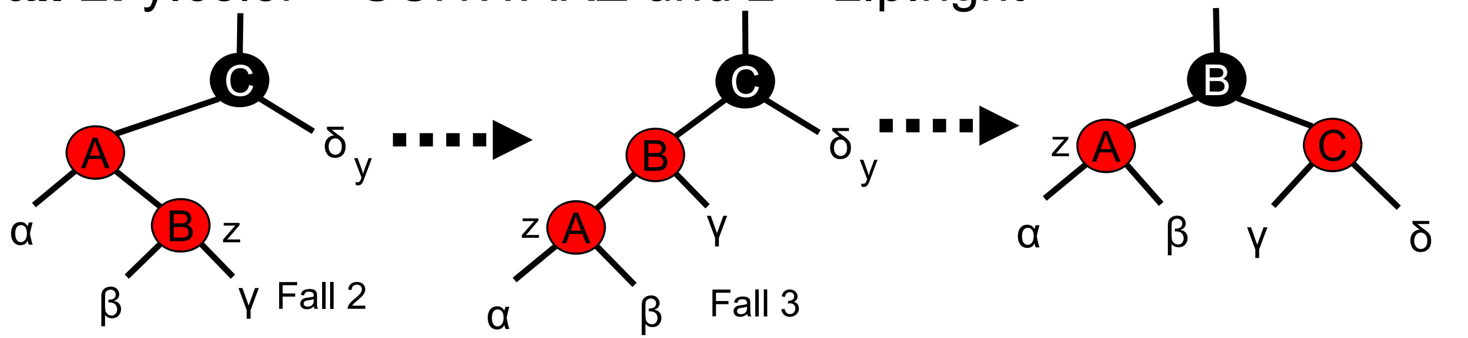
- z‘.color = ROT => Eigenschaft 2 ist verletzt
- z‘.p.color = SCHWARZ => Eigenschaft 4 ist nicht verletzt

Annahme: z‘ $\neq$ T.root

- T.root.color hat sich nicht geändert => Eigenschaft 2 ist nicht verletzt.
- Eigenschaft 4 an Knotenpaar (z, z.p) erfüllt; falls z‘.p.color = ROT, liegt eine Verletzung von Eigenschaft 4 an Knotenpaar (z‘, z‘.p) vor.

Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Fall 2:** y.color = SCHWARZ und z = z.p.right



  - Alle Nachfolger α-γ von z, z.p (außer z) und δ sind schwarz und haben die gleiche Schwarz-Höhe bh().
  - Überführung in Fall 3 durch Links-Rotation (Zeilen 10-11), Eigenschaft 5 bleibt erhalten.

- **Fall 3:** y.color = SCHWARZ und z = z.p.left

  1. *Zeige: z.color = ROT*

     nach Fall 2 gilt z' = z.p und z.p.color = ROT

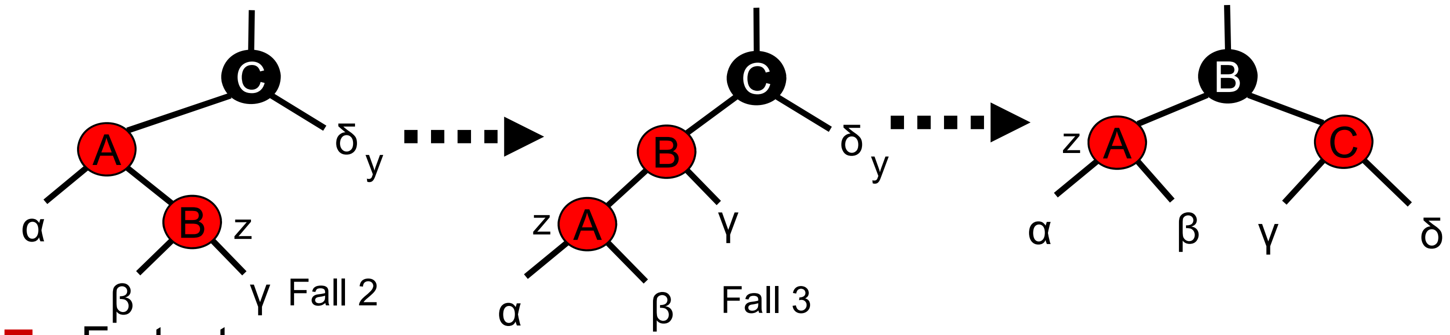
     in Fall 3 werden weder z noch z.color verändert.

  2. *Zeige: Falls z.p = T.root, gilt: z.p.color = SCHWARZ*

     nach Fall 3 (Zeile 12) gilt z.p.color = SCHWARZ.

Die Funktion RB-INSERT-FIXUP() - Korrektheit

- **Fall 3:** y.color = SCHWARZ und z = z.p.left



  - Fortsetzung:
    3. *Zeige: Es gibt maximal eine Verletzung der Rot-Schwarz-Eigenschaft:*
       1. *Eigenschaft 2 => z = T.root und z.color = ROT*
       2. *Eigenschaft 4 => z.color = ROT und z.p.color = ROT*

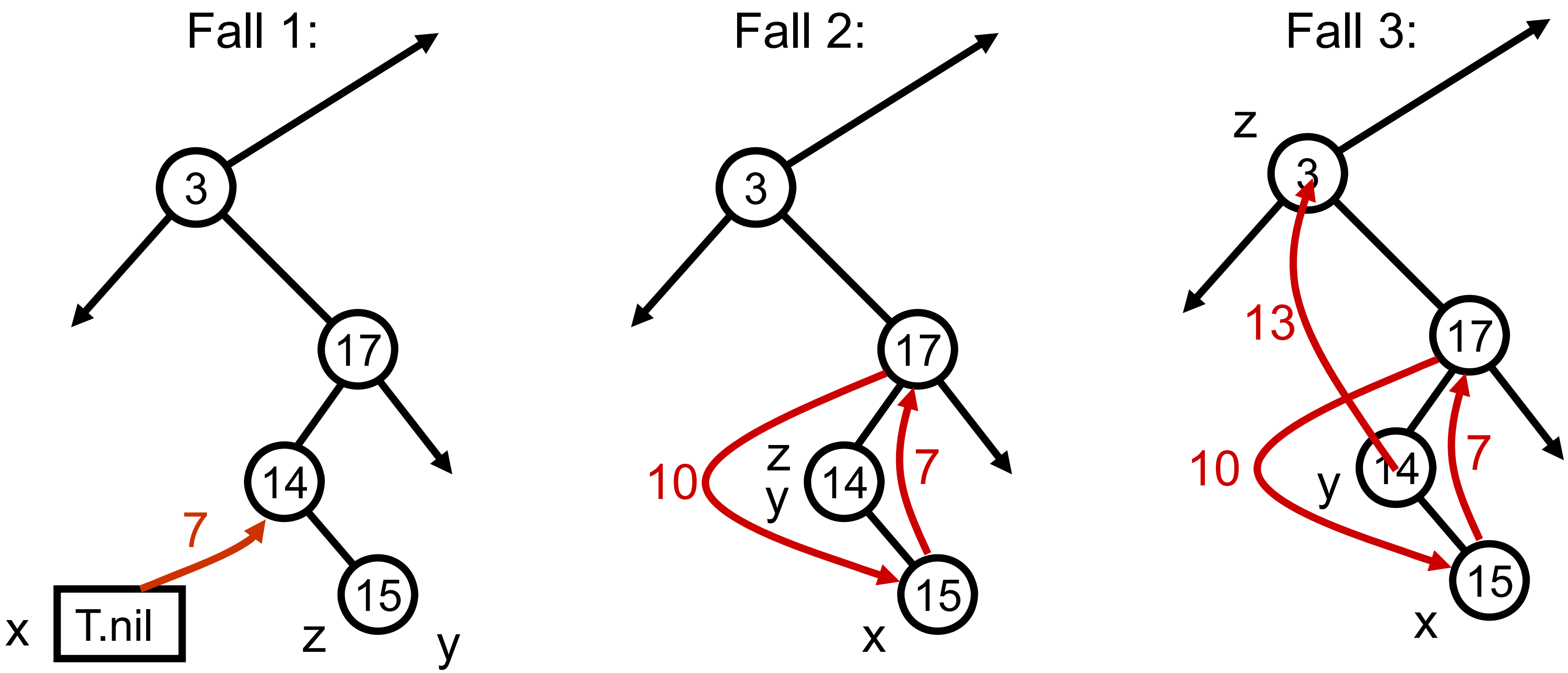
    Eigenschaft 1., 3. und 5. (siehe Zeichnung) sind nicht verletzt.
    - Eigenschaft 2 ist nicht verletzt, da z‘.p.color = SCHWARZ
    - Eigenschaft 4 wird für das Knotenpaar (z, z.p) korrigiert. Da z.p.color = SCHWARZ, gibt es keine weitere Verletzung von Eigenschaft 4.
- **Theorem 8**: RB-INSERT-NODE() fügt einen Knoten unter Erhalt der Rot-Schwarz-Eigenschaften in einen Rot-Schwarz-Baum in Zeit O(log N) ein.

Löschen aus Rot-Schwarz-Bäumen

- `TREE-DELETE-NODE(T, z)` // Cormen: RB-DELETE()

```
// y: der (nach Tausch) zu löschende Knoten
if( z.left == T.nil or z.right == T.nil )
    y = z
else y = TREE-MINIMUM( z.right )// Cormen: TREE-SUCCESSOR(z)
// x: Kind von y, das nicht NIL ist
if( y.left ≠ T.nil ) x = y.left
else x = y.right
 // entferne y aus der Baumstruktur
if( x ≠ NIL ) x.p = y.p          // if-Abfrage wg. Wächter unnötig
if( y.p == NIL ) T.root = x
else
    if( y == y.p.left ) y.p.left = x
    else y.p.right = x
 // kopiere Daten von y nach z
if( y ≠ z ) z.key = y.key
if( y.color == SCHWARZ ) RB-DELETE-FIXUP(T,x)
delete y
```

Löschen aus Rot-Schwarz-Bäumen



- **Korrektheit** von RB-DELETE-NODE() für den Fall y.color = ROT:
  - Die Schwarz-Höhen ändern sich nicht.
  - Es entstehen keine benachbarten roten Knoten.
  - y ≠ T.root, da T.root.color = SCHWARZ

Die Funktion RB-DELETE-FIXUP()

- Welche Rot-Schwarz-Eigenschaften können verletzt sein:
  1. *Jeder Knoten ist entweder rot oder schwarz.* √
  2. *Die Wurzel ist schwarz.* -
  3. *Jedes Blatt (der Wächter) ist schwarz.* √
  4. *Für jeden roten Knoten gilt: beide Nachfolger sind schwarz.* -
  5. *Für jeden Knoten x gilt: Alle Pfade von x zu einem Blatt enthalten die gleiche Anzahl bh(x) schwarzer Knoten.* -

- zu 2.: verletzt, falls y = T.root und x.color = ROT
- zu 4.: verletzt, falls y.p.color = ROT und x.color = ROT
- zu 5.: verletzt für alle Knoten auf den Pfad von x.p bis zur Wurzel

  Alternative Sicht: x.color = ‚doppelt-schwarz' oder ‚rot-schwarz'

  Dann ist 5. erfüllt und dafür 1. (nur an Knoten x) verletzt.

Die Funktion RB-DELETE-FIXUP()

- RB-DELETE-FIXUP(T, x)

Fall 1

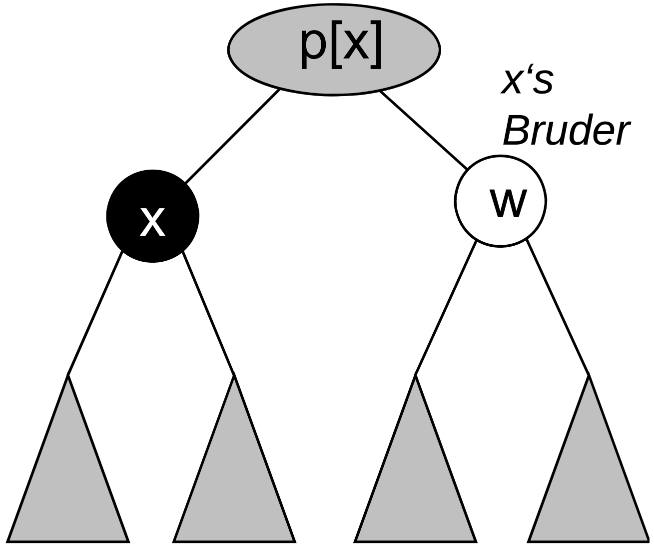
F. 2

Fall 3

Fall 4

Fall 5-8 (symmetrisch zu 1-4)

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x.p.color = ROT
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if( w.left.color == SCHWARZ and
        w.right.color == SCHWARZ )
      w.color = ROT
      x = x.p
    else if( w.right.color == SCHWARZ )
           w.left.color = SCHWARZ
           w.color = ROT
           RIGHT-ROTATE(T, w)
           w = x.p.right
      w.color = x.p.color
      x.p.color = SCHWARZ
      w.right.color = SCHWARZ
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else   // x == x.p.right    // wie then-Teil, vertausche right ⇔left
x.color = SCHWARZ
```



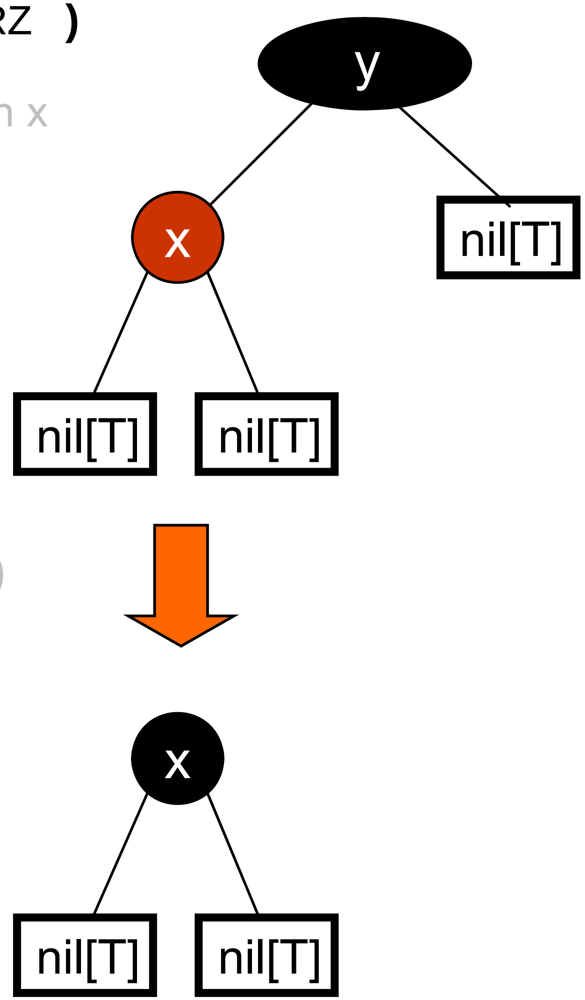
RB-DELETE-FIXUP() – Korrektur von Eigenschaft 2

- RB-DELETE-FIXUP(T, x)

Fall 5-8 (symmetrisch zu 1-4)

Fall 1 | F. 2 | Fall 3 | Fall 4

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x.p.color = ROT
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if( w.left.color == SCHWARZ and
        w.right.color == SCHWARZ )
      w.color = ROT
      x = x.p
    else if( w.right.color == SCHWARZ )
          w.left.color = SCHWARZ
          w.color = ROT
          RIGHT-ROTATE(T, w)
          w = x.p.right
      w.color = x.p.color
      x.p.color = SCHWARZ
      w.right.color = SCHWARZ
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else   // x == x.p.right   // wie then-Teil, vertausche right ⇔left
x.color = SCHWARZ
```



RB-DELETE-FIXUP() – Korrektur von Eigenschaft 4

- RB-DELETE-FIXUP(T, x)

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x.p.color = ROT
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if( w.left.color == SCHWARZ and
        w.right.color == SCHWARZ )
      w.color = ROT
      x = x.p
    else if( w.right.color == SCHWARZ )
          w.left.color = SCHWARZ
          w.color = ROT
          RIGHT-ROTATE(T, w)
          w = x.p.right
      w.color = x.p.color
      x.p.color = SCHWARZ
      w.right.color = SCHWARZ
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else  // x == x.p.right   // wie then-Teil, vertausche right ⇔left
x.color = SCHWARZ
```

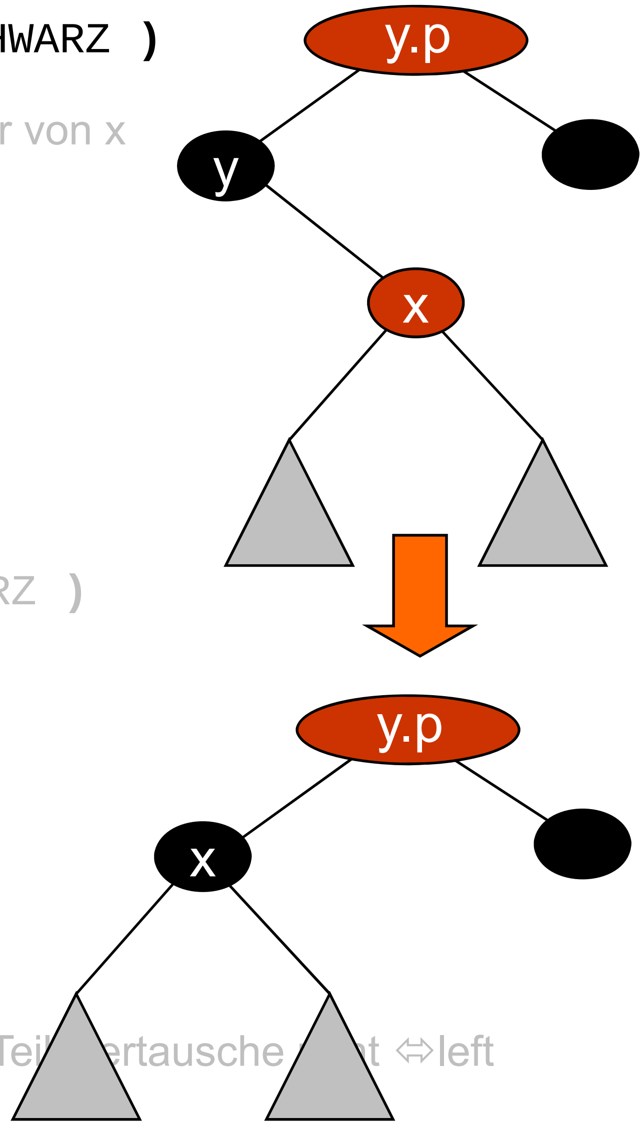
Fall 1

F. 2

Fall 3

Fall 4

Fall 5-8 (symmetrisch zu 1-4)



RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 1

- RB-DELETE-FIXUP(T, x)

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x.p.color = ROT
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if( w.left.color == SCHWARZ and
        w.right.color == SCHWARZ )
          RIGHT-ROTATE(T, w)
          w = x.p.right
      w.color = x.p.color
      x.p.color = SCHWARZ
      w.right.color = SCHWARZ
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else  // x == x.p.right   // wie then-Teil, vertausche right ⇔ left
x.color = SCHWARZ
```

Fall 5-8 (symmetrisch zu 1-4)

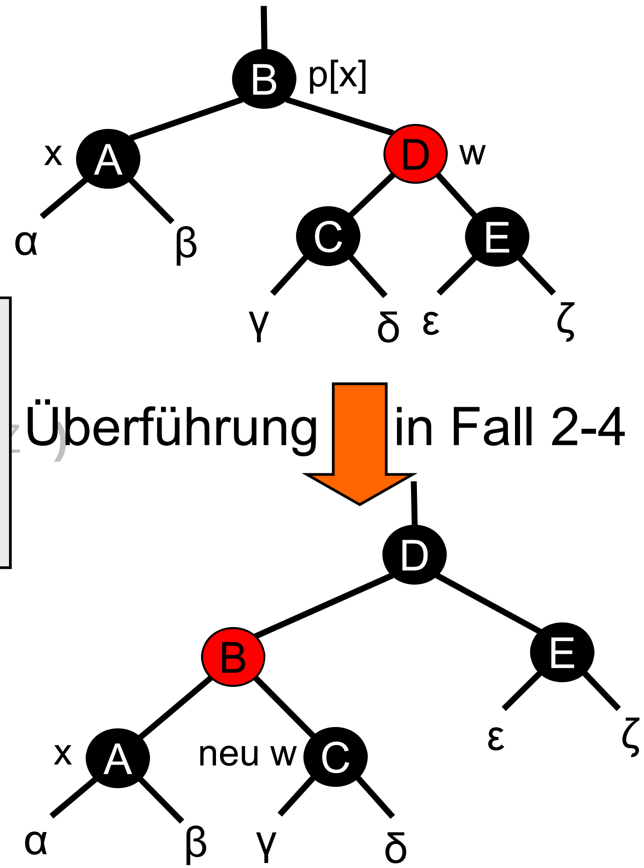
Fall 1

F. 2

Fall 3

Fall 4

Fall 1:
Bedingung: w ist rot
Aktion: vertausche Farben, L-Rotation
Nachbedingung: w ist schwarz (da w schwarze Kinder haben muss)



RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 2

Fall 5-8 (symmetrisch zu 1-4)

Fall 1

F. 2

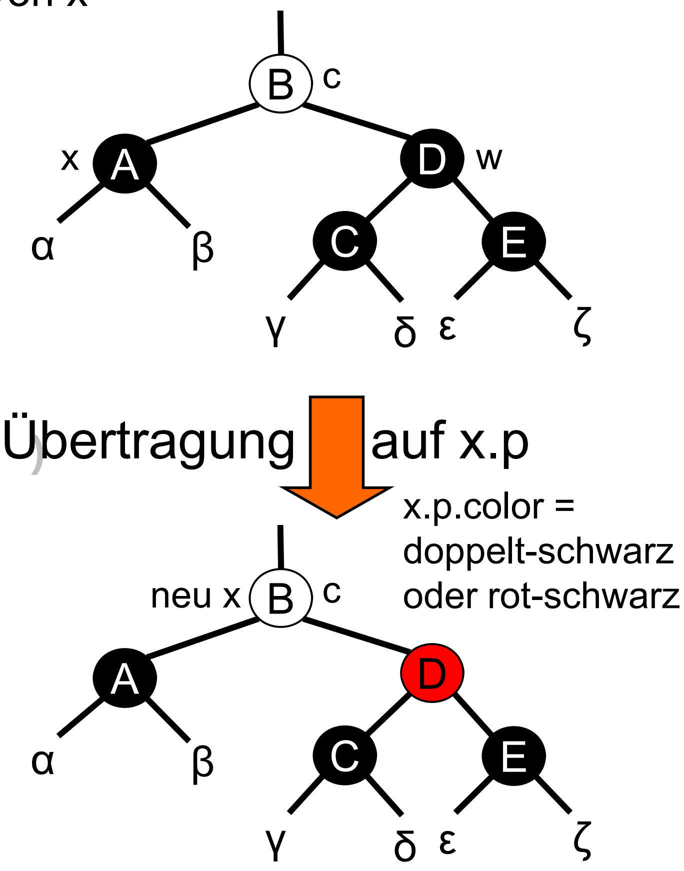
Fall 3

Fall 4

- RB-DELETE-FIXUP(T, x)

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x.p.color = ROT
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if( w.left.color == SCHWARZ and
        w.right.color == SCHWARZ )
      w.color = ROT
      x = x.p
    else if( w.right.color == SCHWARZ )
          w.left.color = SCHWARZ
      ...
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else  // x == x.p.right   // wie then-Teil, vertausche right ⇔left
x.color = SCHWARZ
```

Fall 2:
Bedingung: w hat zwei schwarze Kinder
Aktion: w wird rot, verschiebe doppel-schwarz zu p[x]
Nachbedingung: x ist (einfach-)schwarz



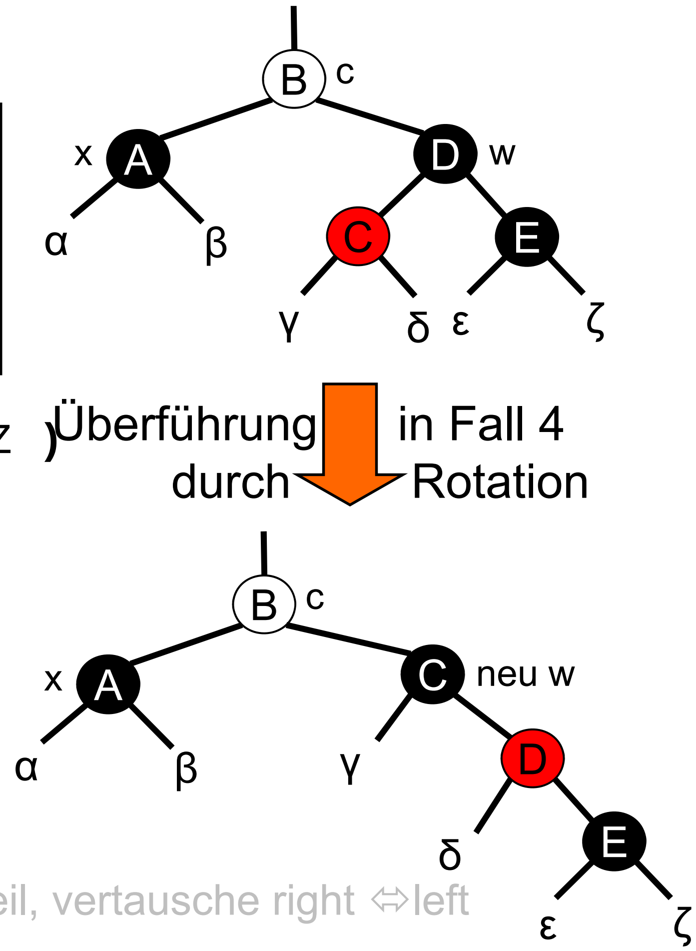
RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 3

- RB-DELETE-FIXUP(T, x)

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x = x.p
    else if( w.right.color == SCHWARZ )
          w.left.color = SCHWARZ
          w.color = ROT
          RIGHT-ROTATE(T, w)
          w = x.p.right
      w.color = x.p.color
      x.p.color = SCHWARZ
      w.right.color = SCHWARZ
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else   // x == x.p.right   // wie then-Teil, vertausche right ⇔left
x.color = SCHWARZ
```

Fall 3:
Bedingung: w ist schwarz, w‘s rechtes Kind ist schwarz, w‘s linkes Kind ist rot
Aktion: Farbentausch und R-Rotation
Nachbedingung: w‘s rechtes Kind ist rot

Fall 1
F. 2
Fall 3
Fall 4
Fall 5-8 (symmetrisch zu 1-4)



RB-DELETE-FIXUP() – Korrektur von Eigenschaft 1: Fall 4

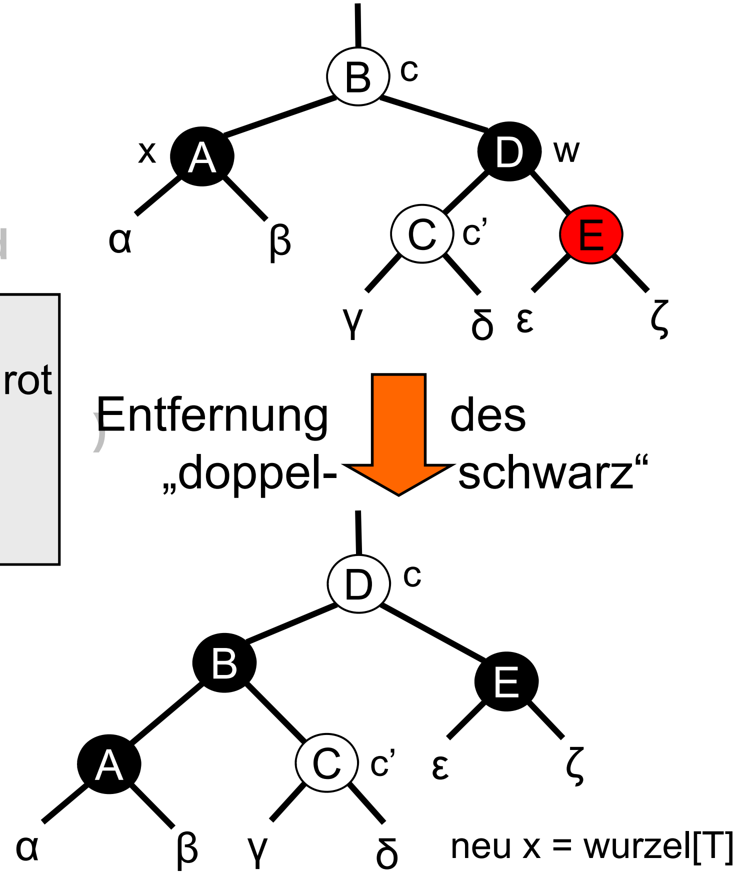
Fall 5-8 (symmetrisch zu 1-4)

Fall 1 | F. 2 | Fall 3 | Fall 4

- RB-DELETE-FIXUP(T, x)

```
while( x ≠ T.root and x.color == SCHWARZ )
  if( x == x.p.left )
    w = x.p.right  // w zeigt auf den Bruder von x
    if( w.color == ROT )
      w.color = SCHWARZ
      x.p.color = ROT
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if( w.left.color == SCHWARZ and
        w.right.color == SCHWARZ )
        RIGHT-ROTATE(T, w)
        w = x.p.right
      w.color = x.p.color
      x.p.color = SCHWARZ
      w.right.color = SCHWARZ
      LEFT-ROTATE(T, p[x] )
      x = T.root
  else  // x == x.p.right   // wie then-Teil, vertausche right ⇔left
x.color = SCHWARZ
```

Fall 4:
Bedingung: w ist schwarz, w‘s rechtes Kind ist rot
Aktion: Farbentausch (nach rechts) und
L-Rotation, x wird (einfach-)schwarz
Nachbedingung: Eigenschaft 1 erfüllt



RB-DELETE-NODE() - Komplexität

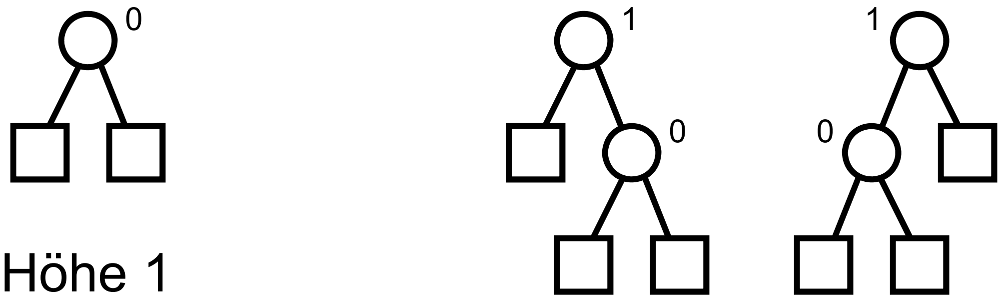
- **Theorem 8**: RB-DELETE-NODE() löscht einen Knoten unter Erhalt der Rot-Schwarz-Eigenschaften in einen Rot-Schwarz-Baum in Zeit O(log N).
  - Die Korrektheit von RB-DELETE-NODE() folgt aus der Diskussion der Fälle.
  - Die Bearbeitung aller Fälle erfolgt in konstanter Zeit.
  - Fall 1 überführt in Fälle 2-4
  - Nach Fall 3 und 4 terminiert die Schleife
  - In Fall 2 wird die Schleife auf dem Elter-Knoten ausgeführt.

- Insgesamt folgt: Ein Rot-Schwarz-Baum realisiert ein Wörterbuch mit asymptotischer Worst-Case Laufzeit O(log N) für die Operationen INSERT, DELETE und SEARCH.

4.3 AVL-Bäume

[aus Ottmann/Widmeyer, Spektrum Akad. Verlag, 2002]

- AVL-Baum: (Adel'son, Velskiĭ und Landis, 1962)
- Für jeden Knoten p gilt:
  - Sei p.bal = Höhe(rechten Teilbaum) – Höhe(linker Teilbaum)
  - Es gilt p.bal $\in \{-1, 0, 1\}$
- Ist ein AVL-Baum balanciert?
  - ein AVL-Baum der Höhe h hat mindestens F_{h+2} Blätter
    (F_h: h-te Fibonacci-Zahl; $F_0 = 0$, $F_1 = 1$, $F_{i+2} = F_i + F_{i+1}$)

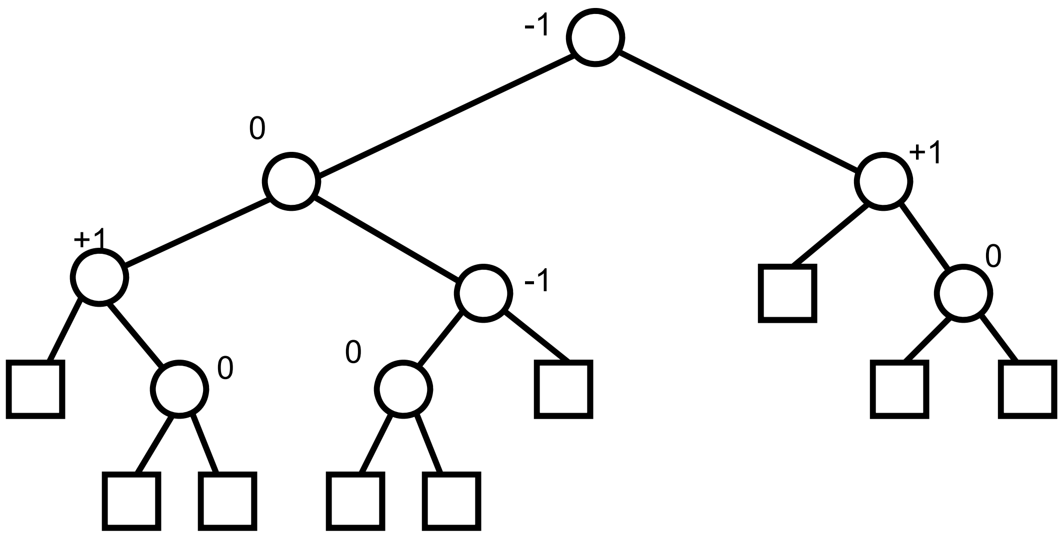


Höhe 1

Höhe 2, min. Anzahl Blätter

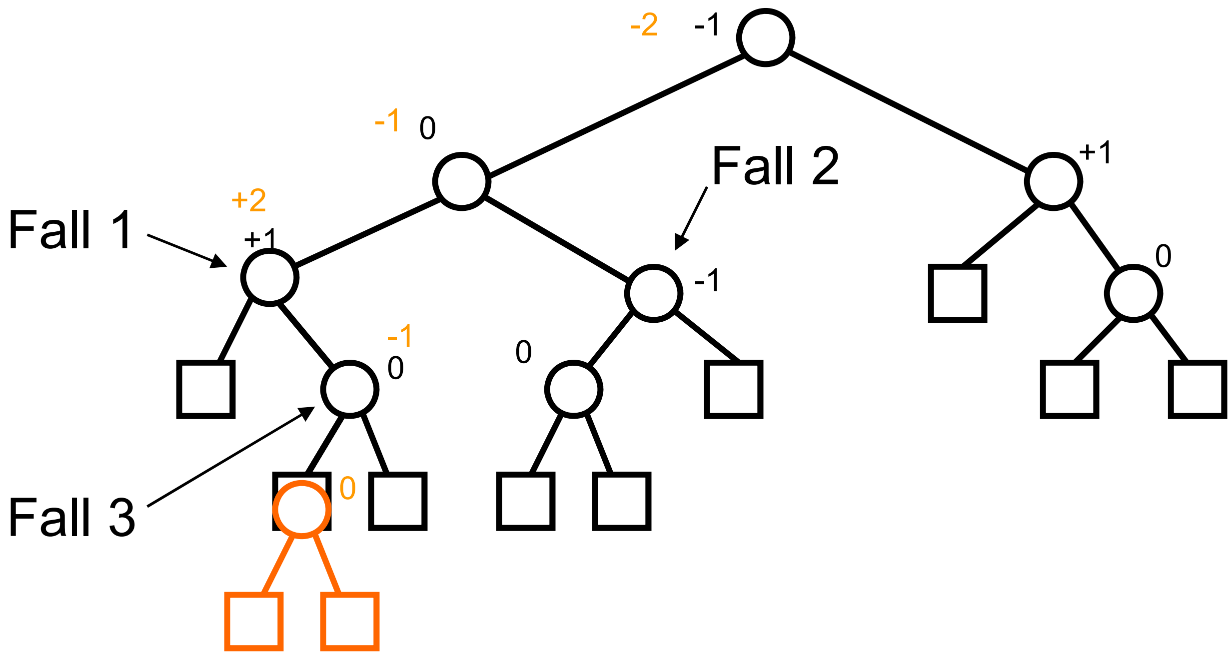
- Es gilt: $F_h \approx 0.72 * 1.62^h$
- N = 2 * Anzahl der Blätter –1 => $N_h \geq 2 * F_{h+2} - 1$
- h = O(log N)

Einfügen in AVL-Bäume 1



- Einfügen in AVL-Bäume:
  - Teil 1: Einfügen wie in natürlichen Bäumen (Einfügestelle p)
    Balancewerte auf dem Pfad von p zur Wurzel ändern sich
  - Teil 2: Wandere von p zur Wurzel und rebalanciere durch Rotationen

Einfügen in AVL-Bäume 2



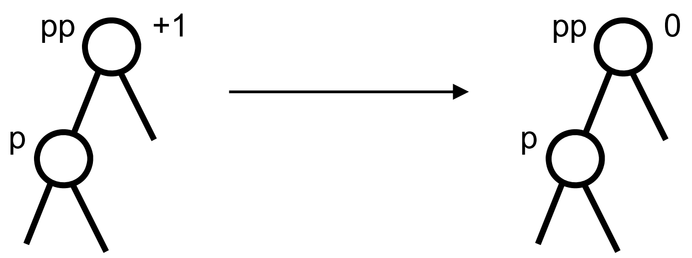
Einfügen unter Knoten p:

- Fall 1: p.bal = +1: füge x als linken Sohn ein, p.bal = 0
- Fall 2: p.bal = -1: füge x als rechten Sohn ein, p.bal = 0
- Fall 3.1: p.bal = 0 und x > p.key: füge x als rechten Sohn ein, p.bal = +1
- Fall 3.2: p.bal = 0 und x < p.key: füge x als linken Sohn ein, p.bal = -1

  Im Fall 3 verändern sich die Balancewerte von p zur Wurzel!

Balancieren nach dem Einfügen: upin-Prozedur

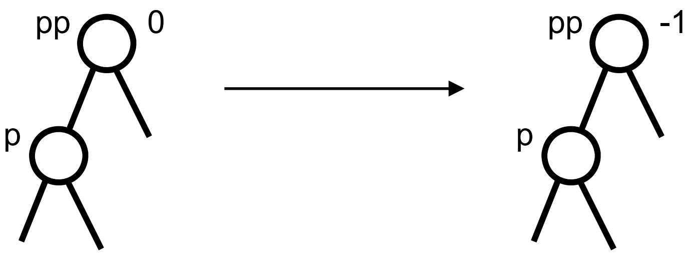
- upin(p : Zeiger auf AVL_node)
  - Vorbedingung: p.bal $\in$ {-1, 1},
  - Höhe des Teilbaums unter p ist um 1 gewachsen
  - upin(p) balanciert die Teilbäume unter p.parent
    und geht dann rekursiv zum Vater
  - Es gibt 6 verschiedene Fälle (sei pp = p.parent) :
    - pp.left = p (p ist linker Sohn) und pp.bal = +1, 0 oder –1 Fall 1.1-1.3
    - pp.right = p (p ist rechter Sohn) und pp.bal = +1, 0, -1 Fall 2.1-2.3
  - Fall 1.1: pp.left = p und pp.bal = +1



- pp.bal ← 0
- Höhe von pp bleibt gleich

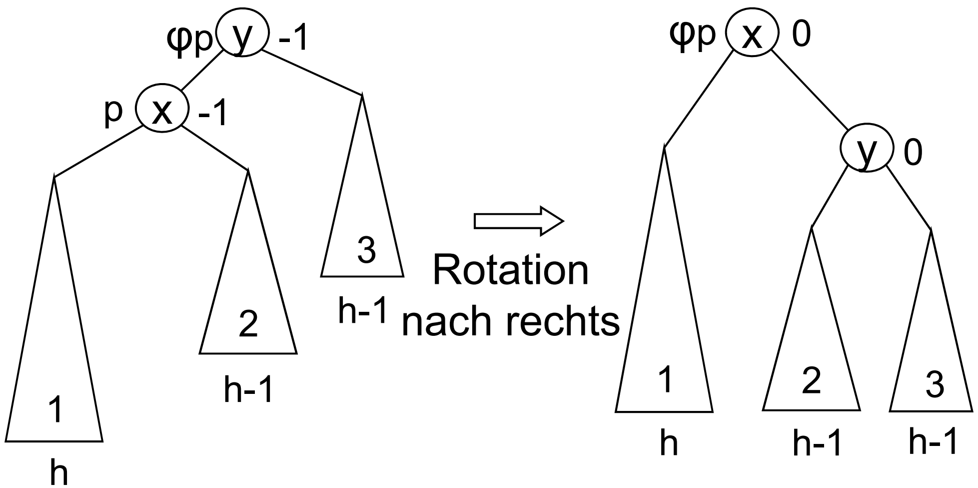
Balancieren nach dem Einfügen: upin-Prozedur

- Fall 1.2: pp.left = p und bal(pp) = 0



- pp.bal ← -1
- Höhe von pp wächst um 1
- ➔ führe upin(pp) aus

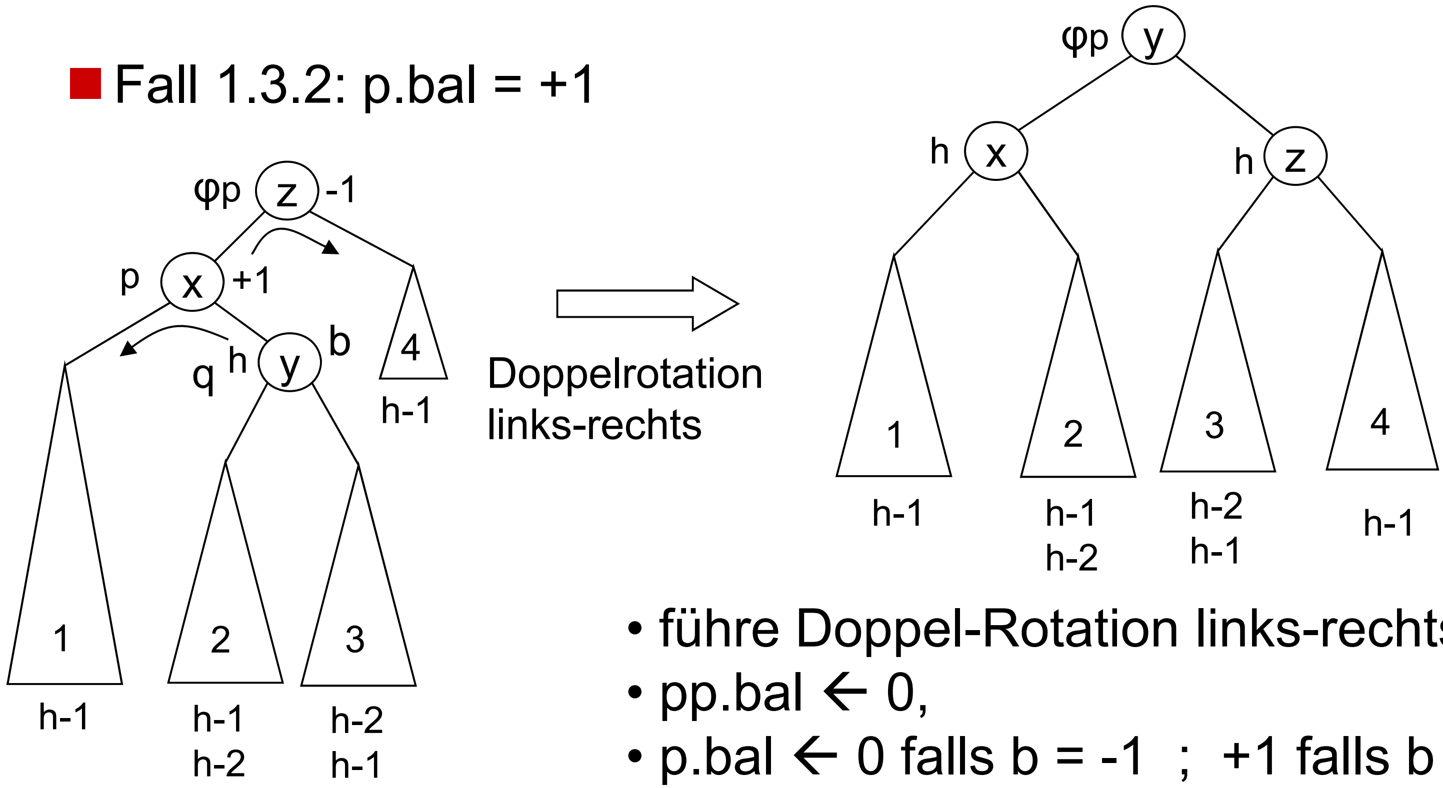
- Fall 1.3: pp.left = p und pp.bal = -1
  - pp.bal = -2, Anpassung der Baumstruktur notwendig
  - Fall 1.3.1: p.bal = -1



- führe Rechts-Rotation durch
- pp.bal ← 0, p.bal ← 0
- Höhe von pp bleibt gleich

Balancieren nach dem Einfügen: upin-Prozedur

- Fall 1.3.2: p.bal = +1



- führe Doppel-Rotation links-rechts aus
- pp.bal ← 0,
- p.bal ← 0 falls b = -1 ; +1 falls b = +1
- q.bal ← -1 falls b = -1 ; 0 falls b = +1
- Höhe von pp bleibt gleich

- Fälle 2.1 – 2.3 sind spiegelsymmetrisch analog
- Löschen in AVL-Bäumen: ähnliche Überlegung, Funktion upout zur Höhenbalancierung

Abschließende Kommentare zu Suchbäumen

- Suchbäume eignen sich als **dynamische Datenstruktur für Wörterbücher** mit Operationen INSERT, DELETE und SEARCH.
- Die Laufzeit aller Operationen sind abhängig von der Höhe des Suchbaums. Im Average Case verhält sich die Höhe logarithmisch zur Anzahl gespeicherter Objekte.
- Ein Suchbaum hat im Worst Case die Höhe $\Theta(N)$, im Best Case und Average Case die Höhe $O(\log N)$. Ein Suchbaum mit Höhe $O(\log N)$ heißt **balanciert**.
- **Rot-Schwarz-Bäume** stellen einen Mechanismus zum Balancieren der Suchbäume nach INSERT- und DELETE-Operationen zur Verfügung. Beide Operationen haben im Worst Case Laufzeit $O(\log N)$.
- **AVL-Bäume** verwenden ein alternatives Balance-Kriterium. INSERT- und DELETE-Operationen erzeugen ebenfalls in $O(\log N)$ Zeit balancierte Suchbäume.

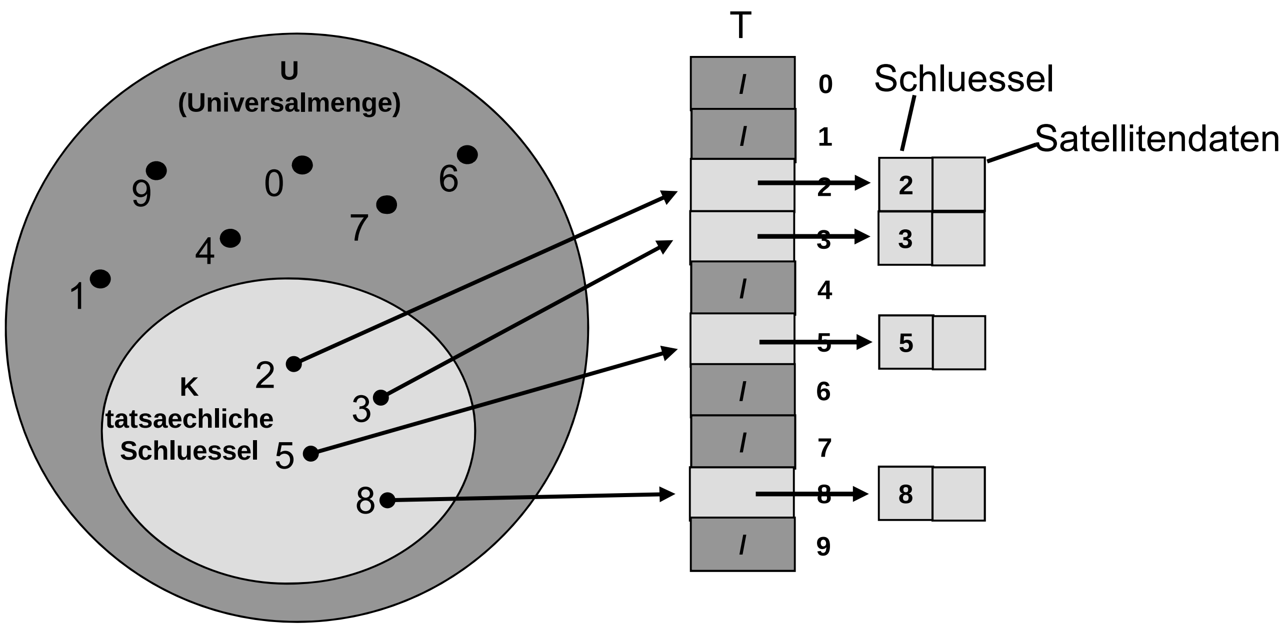
4.4 Hashing mit Verkettung

- Bisherige Voraussetzung:
  - Schlüssel mit Ordnungsrelation $\leq$: Suchbäume mit O(log N)-Laufzeiten für Einfügen, Löschen und Suchen
- Neue Annahme:
  - Schlüssel lassen sich auf natürliche Zahlen abbilden.
- Idee (Hashing):
  - Berechne aus dem Schlüssel einen Index h(key), speichere die Daten in einem Array an Position h(key). Im RAM-Modell erfolgt der Array-Zugriff in konstanter Zeit.

- Probleme:
  - Mehrere Schlüssel werden auf den gleichen Index abgebildet
    ➔ ***Kollision***
  - Der Index-Raum ist viel größer als die zu speichernden Daten

Direkte Adressierung

- Annahme:
  - Schlüssel sind aus der Menge U = {0, …, m-1} (*Universalmenge*).
  - Alle Schlüssel sind voneinander verschieden.
  - m = O(|K|), wobei K die Menge der verwendeten Schlüssel ist.
- Idee: Adresstabelle mit direktem Zugriff:



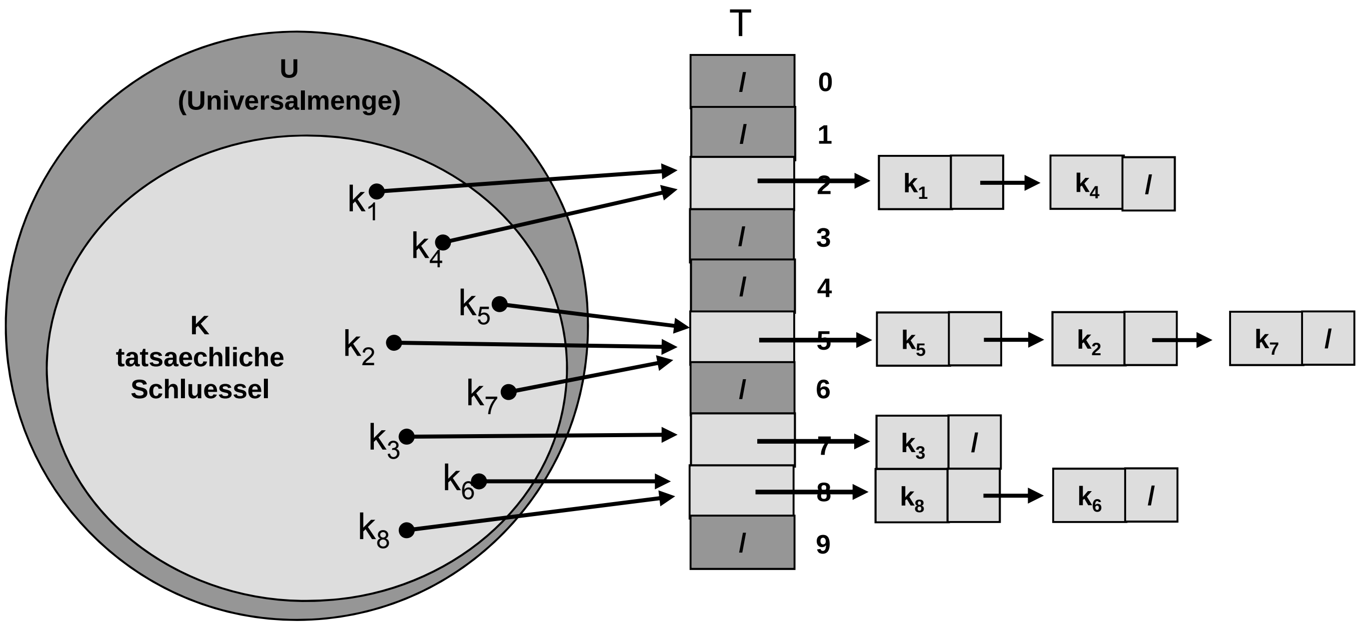
- Operationen INSERT, DELETE, SEARCH sind in O(1) realisierbar.

Hashing

- In der Praxis:
  - |U| >> |K|, d.h. direkte Adressierung ist speicher-ineffizient.
- Ziel: Speicherung mit Platzbedarf $\Theta(|K|)$ unter Erhalt der Laufzeit
- **Hashfunktion**:
  - Funktion zur Abbildung der Universalmenge auf die Menge verfügbarer Speicheradressen: $h: U \rightarrow \{0,\ldots,m-1\}$. $h(k)$ ist der **Hashwert** des Schlüssels k. Die Daten werden in einer **Hashtabelle** $T[0,\ldots,m-1]$ gespeichert.
- **Kollisionen**:
  - Gilt für zwei Schlüssel $k \neq k'$: $h(k) = h(k')$, spricht man von einer **Kollision**.
  - Da $|U| > m$ gilt, sind Kollisionen nicht vermeidbar.
- Entwicklungsziele:
  - Handhabung von Kollisionen
  - Minimierung der auftretenden Kollisionen durch Wahl einer geeigneten Hashfunktion

Kollisionsauflösung durch Verkettung

- Idee:
  - Speicherung aller Elemente mit gleichem Hashwert in einer linearen Liste.
- Operationen:
  - INSERT(T,x): füge x an den Kopf der Liste T[h(x.key)] ein O(1)
    (ohne Test auf Vorhandensein von x)
  - DELETE(T,x): entferne x aus der Liste T[h(x.key)] O(1)
    (dies setzt eine doppelte Verkettung voraus)
  - SEARCH(T,k): suche in der Liste T[h(k)] O(|T[h(x.key)]|)



Analyse von Hashing mit Verkettung

- **Belegungsfaktor**: $\alpha = n / m$
  - $n = |K|$: Anzahl der in der Hashtabelle gespeicherter Objekte
  - $m = |T|$: Die Größe der Hashtabelle
  - beschreibt die mittlere Anzahl Objekte pro Eintrag in der Hashtabelle
- Annahme: **einfaches, gleichmäßiges Hashing**

  Sei X_{ij} das Ereignis, dass Objekt i den Hashwert j erhält. Dann sei $Pr\{X_{ij}=1\} = 1/m$ und X_{ij} und X_{ik} sind stochastisch unabhängig.
- **Lemma**: Sei $n_j = T[j].length$, dann gilt $E[n_j] = \alpha = n / m$

- **Theorem 9:** In einer Hashtabelle mit Verkettung unter Annahme des einfachen, gleichmäßigen Hashings benötigt die *erfolglose* Suche $O(1+\alpha)$ erwartete Laufzeit.
  - Ein nicht enthaltener Schlüssel k wird im Wahrscheinlichkeit 1/m in Liste h(k) abgelegt. Es gilt $E[n_{h(k)}] = \alpha$.
  - Bei erfolgloser Suche wird die Liste T[h(k)] in erwarteter Zeit $O(\alpha)$ vollständig durchlaufen, die Berechnung von h(k) erfolgt in O(1).

Analyse von Hashing mit Verkettung

- **Theorem 10:** In einer Hashtabelle mit Verkettung unter Annahme des einfachen, gleichmäßigen Hashings benötigt die *erfolgreiche* Suche $O(1+\alpha)$ erwartete Laufzeit.
  - Sei $X = x_1,\ldots,x_n$ die Folge, in der die Objekte mit Schlüsseln $k_1,\ldots,k_n$ in die Hashtabelle eingefügt wurden. Sei X_{ij} das Kollisions-Ereignis $h(k_i) = h(k_j)$.
  - SEARCH(T, x_i) durchläuft alle Objekte x_j, die nach x_i eingefügt wurden und in der gleichen Liste stehen, also mit X_{ij}=1. Dann folgt:

$$E[T_{search}(n)] = E\left[\frac{c}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

Mittelung über alle mgl. Suchanfragen x_i

$$= \frac{c}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

wg. Linearität des Erwartungswertes

$$= \frac{c}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

wg. einfachem gleichmäßigen Hashing

$$= c + \frac{c}{nm}\sum_{i=1}^{n}\left(\sum_{j=i+1}^{n}1\right)$$

wg. Gauß'scher Summenformel

$$= c + \frac{c}{nm}\sum_{i=1}^{n}(n-i) = c + \frac{c}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) = c\left(1+\frac{n-1}{2m}\right) = c\left(1+\frac{\alpha}{2}-\frac{\alpha}{2n}\right) = O(1+\alpha)$$

Hashfunktionen

- Anforderungen an Hashfunktionen:
  - leicht und schnell berechenbar
  - gleichmäßige Verteilung der Schlüsselmenge auf die Plätze 0,…,m-1
  - Achtung: Häufung hängt von der Anwendung ab, d.h.
    - Vermeidung anwendungsverursachter Häufungen
- Überprüfung der Qualität von Hashfunktionen:
  - Es gibt viele mögliche Hashfunktionen, zur sinnvollen Auswahl muss man etwas über die Verteilung von K wissen.
- Heuristische Wahl:
  - Hashwerte sollten keine Häufungen für erwartete Muster zeigen.

- Annahme: Im folgenden gelte:

  Schlüsselwerte k sind nicht-negative, ganze Zahlen, also $k \in IN_0$
  - die meisten Schlüssel lassen sich sinnvoll in Zahlen aus IN_o konvertieren
    - Universalität des binären Codes

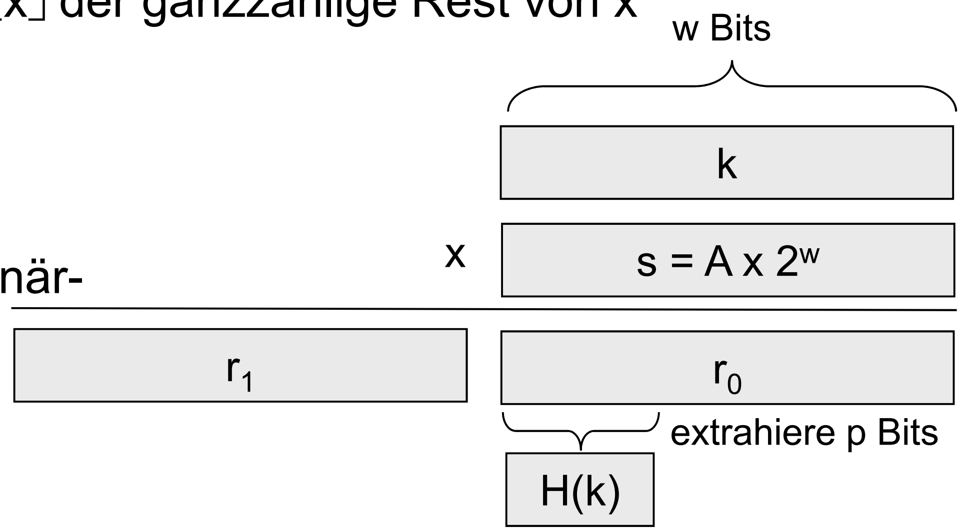
    Achtung: Auch bei dieser Konvertierung kann bereits Häufung eine Rolle spielen.

Hashfunktionen: Divisions- und Multiplikationsmethode

- **Divisionsmethode** h: U → {0,...,m-1} : h(k) = k mod m
  - m gerade: k gerade => h(k) ist gerade
  - m = 2^p: nur die p-letzten Bits werden zur Generierung von h(k) herangezogen
  - Im allgemeinen sollte m möglichst wenig Teiler haben (Primzahl!)

  ABER: Wenn die Schlüsselmenge K in U gleichverteilt ist, dann ergibt auch m = 2^p keine Häufung. Operation ‚mod 2^p' ist wesentlich effizienter als ‚mod m' mit einer Primzahl m!

- **Multiplikatonsmethode** h: U → {0,…,m-1} : h(k) = $\lfloor m(k\ A \bmod 1)\rfloor$
  - Definition: Es sei x mod 1 = x - $\lfloor x \rfloor$ der ganzzahlige Rest von x
    - m kann unabhängig von A gewählt werden, z.B. m= 2^p
    - mit A= s/2^w (w = Wortlänge des Computers) kann die Binär-arithmetik des Prozessors optimal ausgenutzt werden
    - $A \approx \left(\sqrt{5}-1\right)/2$ (D. Knuth)



Universelles Hashing

- Für jede Hashfunktion gibt es eine Menge von n Schlüsseln, die auf den gleichen Platz abgebildet werden (vorausgesetzt U > nm).
- Wie können wir im Mittel eine gute Performanz erhalten?
- **Universelles Hashing**: Zufällige Auswahl einer Hashfunktion unabhängig von der Schlüsselmenge
- Definition: Eine Menge $H = \{ h: U \rightarrow \{0,\ldots,m-1\}\}$ von Hashfunktionen heißt **universell**, g.d.w. $\forall k,l \in U : |\{ h \in H \mid h(k) = h(l)\}| \leq |H|/m$
- Bei zufälliger Wahl einer Hashfunktion und zweier Zahlen $k,l \in U$ ist die Wahrscheinlichkeit der Kollision $[h(k) = h(l)] \leq 1/m$.

Universelles Hashing

- **Theorem 11**: Sei h∈H aus einer Menge universeller Hashfunktionen mit h: U → {0,…,m-1}. Nach Speicherung von n Schlüsseln in Tabelle T (Hashing mit Verkettung) gilt für einen Schlüssel k die erwartete Listenlänge: $$E\left[n_{h(k)}\right] \le \begin{cases} \alpha & : \ \mathrm{k} \notin \mathrm{T} \\ 1+\alpha & : \ \text{sonst} \end{cases}$$

- Beweis:
  - Für ein Paar k,l∈U sei X_{kl} das Kollisionsereignis h(k) = h(l).
  - Da H universell ist, folgt Pr{h(k) = h(l)} ≤ 1/m und somit

$$E[X_{kl}] = \sum_{z=0}^{1} z \Pr\{X_{kl} = z\} \le 1/m$$

  - Fall 1: k ∉ T

$$E\left[n_{h(k)}\right] = E\left[\sum_{l \in T} X_{kl}\right] = \sum_{l \in T} E[X_{kl}] \le \sum_{l \in T} \frac{1}{m} = \frac{n}{m} = \alpha$$

  - Fall 2: k ∈ T

$$E\left[n_{h(k)}\right] = E\left[\sum_{l \in T} X_{kl}\right] = 1 + \sum_{l \in T \setminus \{k\}} E[X_{kl}] \le 1 + \sum_{l \in T \setminus \{k\}} \frac{1}{m} = 1 + \frac{n-1}{m} < 1 + \alpha$$

Universelles Hashing

- **Korollar**: Bei universellem Hashing mit Verkettung mit m Plätzen ist die erwartete Laufzeit von p INSERT-, SEARCH- und DELETE-Operationen mit O(m) INSERT-Operationen O(p).
- Beweis:
  - Nach O(m) INSERT-Operationen gilt $\alpha \leq \frac{cm}{m} = O(1)$
  - $T_{INSERT} = O(1)$, $T_{DELETE} = O(1)$
  - SEARCH-Operation mit Schlüssel k:
    - Für die erwartete Listenlänge gilt $E[n_{h(k)}] \leq \alpha$, bzw. $E[n_{h(k)}] \leq 1+\alpha$
    - Somit folgt $E[T_{SEARCH}] = O(1+\alpha) = O(1)$
  - Für p Operationen ergibt sich somit eine erwartete Laufzeit von O(p)
- Klasse von universellen Hashfunktionen (ohne Beweis):

  Sei p eine Primzahl mit p > m und p >k $\forall$ k$\in$U, sei $Z_x = \{0,\ldots,x-1\}$. Dann ist $H_{p,m}$ mit

$$h_{a,b} : U \to Z_m : h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$$

$$H_{p,m} = \left\{ h_{a,b} : a \in Z_p \setminus \{0\} \text{ und } b \in Z_p \right\}$$

  eine Klasse universeller Hashfunktionen mit $|H_{p,m}| = p(p-1)$

4.5 Offene Adressierung

- Nachteile von Hashing mit Verkettung:
  - zusätzlicher Speicherbedarf für Zeiger
  - komplexe Speicherverwaltung (Belegen und Befreien von Listen-elementen)
  - ineffiziente Speichernutzung bei Häufung

- Offene Adressierung
  - einmaliges Anlegen von Speicher für die Hashtabelle
  - für Überläufer wird ein anderer Platz **in** der Hashtabelle gesucht (eine *offene* Stelle)
  - dieser andere Platz wird berechnet, so dass Zeiger zur Verkettung nicht notwendig sind.
  - **Sondieren**: Sukzessives Überprüfen der Hashtabelle nach freien Plätzen
  - **Sondierungssequenz**: Folge von Hashadressen, die für einen Schlüssel k nach offenen Stellen durchsucht werden
    - Hashfunktion $h: U \times \{0,...,m-1\} \rightarrow \{0,...,m-1\}$
    - **Sondierungszahl** (2. Parameter): gibt den Sondierungsversuch an

Offene Adressierung: Einfügen

- Eigenschaften von Sondierungsfolgen
  - $\forall k \in U$: h(k,j) ist eine bijektive Funktion
    Im Laufe der Sondierung werden alle Speicherplätze getestet.
  - Sondierungssequenz hängt vom Schlüssel k ab.
- Eigenschaften der Hashtabelle
  - T[0,…,m-1], T[i] $\in$ {NIL, DELETED} $\cup$ U
  - keine Satellitendaten (diese können jedoch einfach integriert werden)
  - Initialisierung: T[i] = NIL $\forall$ i $\in$ {0,…,m-1}
- HASH-INSERT-PRE(T, k)

```
i = 0
repeat
  j = h(k,i)
  if( T[j] == NIL ) T[j] = k; return j
  else i = i+1
until( i == m )
error „Hashtable overflow!"
```

Offene Adressierung: Suchen

- Suchalgorithmus:
  - Folge der Sondierungssequenz für k und vergleiche den Schlüssel
- HASH-SEARCH(T,k)

```
i = 0
repeat
   j = h(k,i)
   if( T[j] == k ) return j
   else i = i+1
until( T[j] == NIL or i == m )
return NIL
```

- Was passiert, wenn ein Schlüssel k gelöscht wird?
  - Schlüssel, die nach k gespeichert werden, können nicht wiedergefunden werden, da die Repeat-Schleife bei T[j]=NIL terminiert!

Offene Adressierung: Einfügen und Löschen

- Löschverfahren:
  - Suche den Platz T[i] mit T[i] = k und setze T[i] = DELETED
  - Suchen: behandle DELETED wie einen belegten Platz
  - Einfügen: behandle DELETED wie einen freien Platz
- HASH-INSERT(T, k)

```
i = 0
repeat
  j = h(k,i)
  if( T[j] ∈ {NIL,DELETED})
    T[j] = k; return j
  else i = i+1
until( i == m )
error „Hashtable overflow!“
```

- HASH-DELETE(T,k)

```
i = 0
repeat
  j = h(k,i)
  if( T[j] == k )
    T[j] = DELETED; return j
  else i = i+1
until( T[j] == NIL or i == m )
return NIL
```

- ACHTUNG: Laufzeit von HASH-SEARCH() hängt nicht mehr vom Belegungsfaktor α ab, sondern von $|\{ i \in \{0,\ldots,m-1\} \mid T[i] \neq \text{NIL}\}|$

  => In Anwendungen mit vielen DELETE-Operationen sollte Hashing mit Verkettung zur Kollisionsauflösung verwendet werden.

Offene Adressierung: Ein Beispiel

- Beispiel:
  - Markierungen: N: NIL
    D: DELETED
    U: USED, d.h. ein Schlüssel ist dort gespeichert
  - Hashfunktion: $h(k,i) = (k + i) \bmod 7$

leere Hashtabelle

0	1	2	3	4	5	6
N	N	N	N	N	N	N

insert(15):

0	1	2	3	4	5	6
	15					
N	U	N	N	N	N	N

insert(36):

0	1	2	3	4	5	6
	15	36				
N	U	U	N	N	N	N

search(71): → -1

0	1	2	3	4	5	6
	15	36				
N	U	U	N	N	N	N

Offene Adressierung: Ein Beispiel

insert(17):

0	1	2	3	4	5	6
	15	36	17			
N	U	U	U	N	N	N

insert(29):

0	1	2	3	4	5	6
	15	36	17	29		
N	U	U	U	U	N	N

search(36): → 2

0	1	2	3	4	5	6
	15	36	17	29		
N	U	U	U	U	N	N

delete(36):

0	1	2	3	4	5	6
	15	36	17	29		
N	U	D	U	U	N	N

search(29): → 4

0	1	2	3	4	5	6
	15	36	17	29		
N	U	D	U	U	N	N

insert(71):

0	1	2	3	4	5	6
	15	71	17	29		
N	U	U	U	U	N	N

Offene Adressierung: Sondierungsfunktionen

- Hilfshashfunktion h‘: U → {0,…,m-1}
- Sondierungsfunktionen:
  - **Lineares Sondieren** (linear probing): h(k,i) = (h‘(k) + i) mod m
    - Von h‘(k) aus wird jeweils der nächst größere Platz getestet.
    - Problem: Es entstehen lange Ketten besetzter Plätze:
      Pr{Kette der Länge i -> Kette der Länge i+1} = (i+1)/m
    - Bei insert ist die Wahrscheinlichkeit groß, lange Ketten zu verlängern
      → „**primäres Clustern**“

  - **Quadratisches Sondieren** (quadratic probing):
    - $h(k,i) = (h'(k) + c_1\, i + c_2\, i^2) \bmod m$ mit $c_2 \neq 0$
    - m, c_1 und c_2 müssen so gewählt werden, dass h(k,i) bzgl. i bijektiv ist.
    - Problem: Bei Kollisionen h‘(k) = h‘(k‘) haben beide Schlüssel die gleiche Sondierungsfolge → „**sekundäres Clustern**“

Offene Adressierung: Doppeltes Hashing

- Hilfshashfunktionen h_1: U → {0,…m-1}, h_2: U → {1,…,m‘} mit m‘ < m
  - Verwende h_1, um den Startpunkt der Sondierungsfolge zu bestimmen
  - Verwende h_2, um den Offset der Sondierungsfolge zu bestimmen
- **Doppeltes Hashing** (double hashing):
  - $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$
    - h(k,i) ist bzgl. i nur bijektiv, falls $h_2(k) \neq 0$ und $h_2(k)$ teilerfremd zu m ist
    - Ideal: h_1 und h_2 sind bzgl. der Kollisionswahrscheinlichkeit stochastisch **unabhängig**:

      $P(\ h_1(k)=h_1(k') \text{ und } h_2(k)=h_2(k')\ ) = P(\ h_1(k)=h_1(k')\ ) * P(\ h_2(k)=h_2(k')\ )$
    - Es werden $\Theta(m^2)$ verschiedene Sondierungssequenzen verwendet.
- Beispiele:
  1. m ist eine Primzahl, $h_1(k)= k \bmod m$; $h_2(k)= 1 + k \bmod (m-2)$
  2. $m = 2^l$; $h_1(k)= k \bmod m$; $h_2(k)= 1 + 2\, k \bmod (m/2\ )$

Offene Adressierung: Analyse

- Annahmen:
  - Komplexität in Abhängigkeit vom Belegungsfaktor α, $\alpha < 1$
  - Gleichmäßiges Hashing: Alle Sondierungssequenzen h(k,i), $0 \leq i < m$ sind gleich wahrscheinlich (Achtung: durch keine Sondierungsfunktion erfüllt!)
- Wie viele Sondierungsschritte sind im Falle einer erfolglosen / erfolgreichen Suche zu erwarten?

- **Theorem 12**: Für eine Hashtabelle mit offener Adressierung und Belegungsfaktor $\alpha < 1$ ist die erwartete Anzahl der Sondierungen bei *erfolgloser Suche* unter der Annahme des gleichmäßigen Hashings höchstens $1 / (1 - \alpha)$.
- Beweis:
  - Jede Sondierung außer die letzte greift auf einen belegten Platz zu.
  - X: Anzahl der durchgeführten Sondierungen
  - A_i: Ereignis, dass es eine i-te Sondierung gibt, die auf einen belegten Platz zugreift.

Offene Adressierung: Analyse

- Beweis Theorem 12 (Fortsetzung)

1. Hilfsgleichung:

$$E[X] = \sum_{i=1}^{m} i \Pr\{X = i\} = \sum_{i=1}^{m} i\left(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}\right)$$

$$= \sum_{i=1}^{m} i \Pr\{X \geq i\} - \sum_{i=1}^{m} i \Pr\{X \geq i+1\}$$

$$= \sum_{i=1}^{m} i \Pr\{X \geq i\} - \sum_{i=2}^{m+1} (i-1) \Pr\{X \geq i\}$$

$$= \sum_{i=1}^{m} i \Pr\{X \geq i\} - \sum_{i=1}^{m} (i-1) \Pr\{X \geq i\}$$

$$= \sum_{i=1}^{m} \Pr\{X \geq i\}$$

siehe Cormen C.24

2. Hilfsgleichung:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad \text{für } |x| < 1$$

unendlich fallende geometrische Reihe, siehe Cormen A.6

Offene Adressierung: Analyse

- Beweis Theorem 12 (Fortsetzung)

$$\Pr\{X \geq i\} = \Pr\left\{\bigcap_{k=1}^{i-1} A_k\right\}$$

$$= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\left\{A_{i-1} \mid \bigcap_{k=1}^{i-2} A_k\right\}$$

$$= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

$$E[X] = \sum_{i=1}^{m} i \Pr\{X = i\} = \sum_{i=1}^{m} \Pr\{X \geq i\}$$

$$\leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Offene Adressierung: Analyse

- Interpretation:
  - $E[X] \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$
  - Eine Sondierung findet immer statt.
  - Die Wahrscheinlichkeit für eine zweite Sondierung ist α
  - Die Wahrscheinlichkeit für eine dritte Sondierung ist α^2 …

- Beispiele (erfolglose Suche):
  - $\alpha = 0{,}5$, d.h. die Hashtable ist halb voll: $E[X] \leq 1/(1-0{,}5) = 2$
  - $\alpha = 0{,}9$, d.h. die Hashtable ist 90% voll: $E[X] \leq 1/(1-0{,}9) = 10$
  - $\alpha = 0{,}99$, d.h. die Hashtable ist 99% voll: $E[X] \leq 1/(1-0{,}99) = 100$

- **Korollar**: Unter der Annahme des gleichmäßigen Hashings benötigt HASH-INSERT() im Mittel höchstens $1/(1-\alpha)$ Sondierungen.
- HASH-INSERT() benötigt konstante Zeit pro Sondierung + konstante Zeit für das Einfügen des Elements, also asymptotisch im Mittel $O(1/(1-\alpha))$.

Offene Adressierung: Analyse

- **Theorem 13**: Für eine Hashtabelle mit offener Adressierung und Belegungsfaktor $\alpha < 1$ ist die Anzahl der Sondierungen bei *erfolgreicher Suche* unter den Annahmen (1) gleichmäßiges Hashing und (2) gleiche Wahrscheinlichkeit für den Suchschlüssel höchstens $(1/\alpha) \ln(1 / (1 - \alpha))$.
- Beweis:
  - Angenommen, der gesuchte Schlüssel k war der (i+1)-te Schlüssel bzgl. der Einfüge-Reihenfolge. Dann war zum Zeitpunkt der Einfügung α=i/m und nach Theorem 12 gilt: $E[X(\text{HASH-SEARCH}(T,k))] \leq \frac{1}{1-i/m}$
  - Mittelung über alle möglichen Suchschlüssel ergibt:

$$E[X] = \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\left(\sum_{i=0}^{n-1}\frac{1}{m-i}\right)$$

$$= \frac{1}{\alpha}\left(\frac{1}{m}+\frac{1}{m-1}+\cdots+\frac{1}{m-n+1}\right) = \frac{1}{\alpha}\left(\sum_{i=m-n+1}^{m}\frac{1}{i}\right)$$

$$= \frac{1}{\alpha}\left(\sum_{i=1}^{m}\frac{1}{i}-\sum_{i=1}^{m-n}\frac{1}{i}\right) = \frac{1}{\alpha}\left(H_m - H_{m-n}\right)$$

H_m: m-te harmonische Zahl siehe Cormen A.7

Offene Adressierung: Analyse

- Beweis Theorem 13 (Fortsetzung)

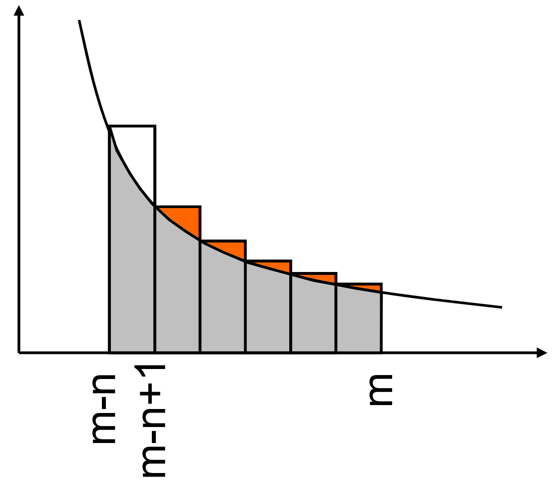
$$E[X] = \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\left(\sum_{i=0}^{n-1}\frac{1}{m-i}\right)$$

$$= \frac{1}{\alpha}\left(\frac{1}{m}+\frac{1}{m-1}+\cdots+\frac{1}{m-n+1}\right) = \frac{1}{\alpha}\left(\sum_{i=m-n+1}^{m}\frac{1}{i}\right)$$

$$\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\,dx$$

$$= \frac{1}{\alpha}\left(\ln(m)-\ln(m-n)\right)$$

$$= \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) = \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$



Approximation von Summe durch Integral, siehe Cormen A.2

- Beispiele:
  - α = 0,5, d.h. die Hashtable ist halbvoll: E[X] $\leq$ 2 ln(2) $\approx$ 1,39
  - α = 0,9, d.h. die Hashtable ist 90% voll: E[X] $\leq$ 10/9 ln(10) $\approx$ 2,56
  - α = 0,99, d.h. die Hashtable ist 99% voll: E[X] $\leq$ 100/99 ln(100) $\approx$ 4,65

Einfügen mit Vertauschen: Brents INSERT-Operation

[siehe Ottmann, Widmayer, Spektrum Akad. Verlag 2002, Kap. 4.3]

- Doppeltes Hashing mit Brents INSERT-Algorithmus:
  - Betrachte zwei alternative Plätze für k (j_next) und T[k] (j_alt)
- BRENT-INSERT(T, k)

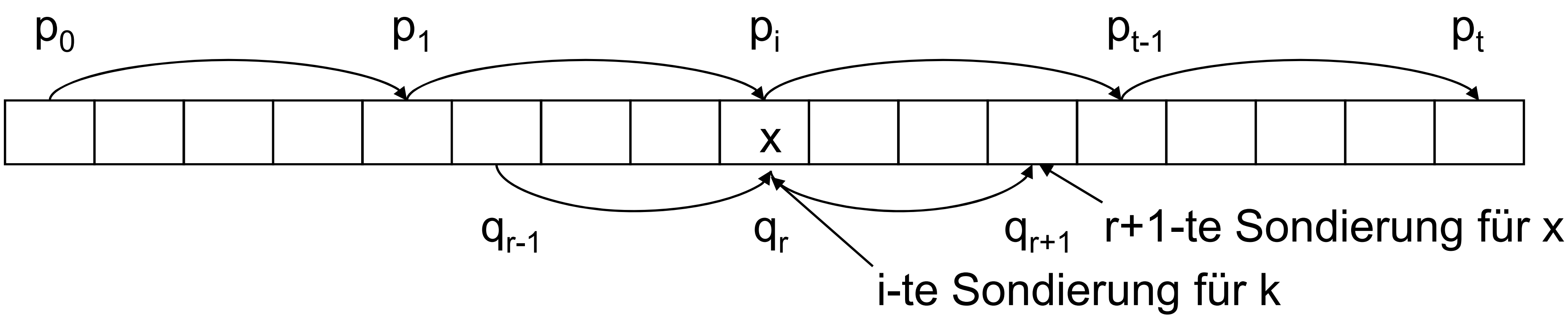
```
t = 0; j = h1(k)
while( T[j] ∉ {NIL, DELETED} and t < m )
  t      = t+1
  j_next = (j + h2(k)) mod m
  j_alt  = (j + h2(T[j])) mod m
  if( T[j_next] ∈ {NIL,DELETED}
      or T[j_alt] ∉ {NIL, DELETED} ) j = j_next
  else SWAP( k, T[j] ); j = j_alt
if( t == m ) error „Hashtable overflow!“
else T[j] = k
```

Der Algorithmus ist korrekt, allerdings ist es nicht Brents INSERT-Operation.

Einfügen mit Vertauschen: Brents INSERT-Operation

[siehe Knuth, Art of Computer Programming, Vol 3, Kap. 6.4]

- Doppeltes Hashing mit Brents INSERT-Algorithmus:
  - Betrachte für eine Sondierungsfolge $p_0, p_1, \ldots, p_t$ nach jedem erfolglosem Sondierungsschritt i die Möglichkeit, die Schlüssel $T[p_0], \ldots T[p_{i-1}]$ durch (t-i-1)-faches Sondieren zu platzieren.
  - Worst-Case Laufzeit für BRENT-INSERT() ist quadratisch in der Anzahl der Sondierungen
  - durchschn. Suchzeit < 2.5 (ohne Löschen) ist unabhängig von α!



Falls $T[q_{r+(t-i-1)}] \in$ {NIL,DELETED}:

$T[q_{r+(t-i-1)}] \leftarrow x$		Sondierungsfolge q für x verlängert sich um $t - i - 1$
$T[p_i]$	$\leftarrow k$	Sondierungsfolge p für k verkürzt sich um $t - i$
Bilanz:		Sondierungsfolgen sind um 1 kürzer als bei doppeltem Hashing

Einfügen mit Vertauschen: Brents INSERT-Operation

- BRENT-INSERT(T, k)

```
t = 0          // zählt die Anzahl der Sondierungsschritte
j = h1(k)      // aktuell sondierte Position für Schlüssel k
while( T[j] ∉ {NIL, DELETED} and t < m )
   t = t+1
   j = (j + h2(k)) mod m
   if( t ≥ 2 and T[j] ∉ {NIL, DELETED} )       // Brents Modifikation
      p_i = h1(k)
      for i = 0 to t-2 do
         // teste, ob T[p_i] mit einem weiteren Sondierungsschritt eingefügt
         // werden kann
         x = T[p_i]
         q_next = ( p_i + (t-i-1)h2(x) ) mod m
         if( T[q_next] ∈ {NIL,DELETED} )
            T[q_next] = x
            T[p_i] = DELETED; j = p_i
            break;
         p_i ← ( p_i + h2(k) ) mod m
if( t == m ) error „Hashtable overflow!“
else T[j] = k; return j
```

Abschließende Kommentare zu Hashing

- Durch Umrechnung von Schlüssel in Indizes (**Hashfunktion**) lässt sich ein Wörterbuch-Datenstruktur mit nahezu konstanten durchschnittlichen Laufzeiten für INSERT, DELETE und SEARCH realisieren.
- Laufzeiten im Average Case hängen vom **Belegungsfaktor** ab.
- **Kollisionen sind nicht vermeidbar**, im Worst-Case haben die Wörterbuch-Operationen Laufzeit $\Theta(N)$.
- Werden häufig Daten gelöscht, sollten **Kollisionen durch Verkettung** aufgelöst werden.
- **Offene Adressierung** ist speichereffizienter als Verkettung, allerdings wirkt die Operation DELETE laufzeitschädlich auf SEARCH. Offene Adressierung sollte nur angewandt werden, wenn wenige DELETE-Operationen notwendig sind.
- Bei der offenen Adressierung ist **Doppeltes Hashing** die zu bevorzugende Sondierungsmethode.
- **Brents INSERT-Operation** mit Doppeltem Hashing ermöglicht das Suchen mit durchschnittlich weniger als 2.5 Sondierungsschritten.
- Für statische Datenmengen gibt es **perfekte Hashverfahren** mit Worst-Case Laufzeit O(1) für INSERT und SEARCH (nicht Teil dieser Vorlesung).