Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Tanya Braun (Übungen)

sowie viele Tutoren

Dynamische Menge

- Datenstruktur, die Objekte verwaltet, für die Schlüssel definiert sind:
  - einfache Menge (zunächst betrachtet)
  - Multimenge
  - geordnete (Multi-)Menge
- Mengen können verändert werden (anders als in der Mathematik)
- Obligatorische Operationen:
  - test(k, s): testet, ob ein Element mit Schlüssel k in s enthalten ist (liefert true/false)
  - search(k, s):

    (1a) liefert das Elements aus s, dessen Schlüssel k ist, oder nil

    (1b) liefert das Elements aus s, dessen Schlüssel key minimal in s ist und für den key≥k gilt

    (2) liefert eine Menge von Elementen aus s, deren Schlüssel k ist
  - insert(x, s): fügt das Element x in s ein (s wird ggf. modifiziert)
  - delete(x, s): löscht Element x aus s (s wird ggf. modifiziert)

Dynamische Mengen

- Zusätzliche Operationen
  - union(s1, s2) (merge)
  - intersect(s1, s2)
  - map(f, s): wendet f auf jedes Element aus s an und gibt Ergebnisse als neue Menge zurück
  - fold(f, s, init): wendet die Funktion f kaskadierend auf Objekte in s an, liefert einen Wert
- Iteratoren für einfache Mengen und Multimengen
  - getIterator(s),
  - testNextElement(i), testPreviousElement(i)
  - nextElement(i), previousElement(i)
- Iteratoren für geordnete Mengen
  - getIterator(s, fromKey), getIterator(s, fromKey, toKey)

Iteration über "lineare Strukturen" (Mengen)

- Wir wollen über die genaue Datenstruktur abstrahieren

```
function test(key, S)
  for x ∈ S do
    if key = key(x) then
      return true
  return false
```

- Das ist eine Kurzschreibweise für die Iteratoranwendung

```
function test(key, S)
  iter := getIterator(S)            // Erzeuge Iterator mit Zustand
  while testNextElement(iter)       // Noch ein Element vorhanden?
    x := nextElement(iter)          // Funktion mit Iteratorzustandsänderung
    if key = key(x) then
      return true
  return false
```

Prioritätswarteschlangen vs. Mengen

- Spezialfall einer dynamischen Menge: Prioritätswarteschlange (Heap)
  - geordnete Schlüssel
  - besondere Suchfunktion min, Löschfunktion deleteMin sowie Funktion decreaseKey
  - Keine Operationen/Iteration auf Elementen definiert
- Hier nun:
  - nicht notwendigerweise geordnete Schlüssel
  - Beliebiger Zugriff: Elementtest bzw. Suchen
  - Ausführung von Operation(en) auf jedem Element (map bzw. fold) sowie Iteration über Elemente (Zugriff auf **alle** Elemente)

Einfache Realisierung von Mengen

- Felder
- Listen (ggf. doppelt verkettetet oder sogar zyklisch)

Felder: Lineares/Sequentielles Suchen

- Datenbestand ist unsortiert
- Es wird sequentiell von Anfang bis zum Ende gesucht bis der Schlüssel gefunden ist oder das Ende erreicht wird

```
function test(k, A)
  i := 1; n := length(A)
 while i ≤ n do
   if k = key(A[i]) then
     return true
   i := i + 1
 return false
```

```
function test(key, S)
  for x ∈ S do
    if key = key(x) then
      return true
 return false
```

- Listen würden in vergleichbarer Weise behandelt

Komplexität Naives/Lineares/Sequentielles Suchen

- **Günstigster Fall:** Element wird an 1. Stelle gefunden: $T_{min}(n) \in \Theta(1)$
- **Ungünstigster Fall:** Element wird an letzter Stelle gefunden (komplette Folge wurde durchlaufen): $T_{max}(n) \in \Theta(n)$
- **Durchschnittlicher Fall (Element ist vorhanden):**
  Annahme: kein Element wird bevorzugt gesucht:

$$T_{avg}(n) = \frac{1}{n} \times \sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n \times (n+1)}{2} = \frac{n+1}{2} \in \Theta(n)$$

- **Fall Misserfolg der Suche (Element nicht gefunden):**
  Es muss die gesamte Folge durchlaufen werden: $T_{fail}(n) \in \Theta(n)$

Selbstanordnende Listen

- **Idee:**
  - Ordne die Elemente bei der sequentiellen Suche so an, dass die Elemente, die am häufigsten gesucht werden, möglichst weit vorne stehen
    - Meistens ist die Häufigkeit nicht bekannt, man kann aber versuchen, *aus der Vergangenheit auf die Zukunft* zu *schließen*

- **Vorgehensweise:**
  - Immer wenn nach einem Element gesucht wurde, wird dieses Element weiter vorne in der Liste platziert

Strategien von selbstanordnenden Listen

- ***MF - Regel*, Move-to-front:**
  Mache ein Element zum ersten Element der Liste, wenn nach diesem Element erfolgreich gesucht wurde. Alle anderen Elemente bleiben unverändert.
- ***T - Regel*, Transpose:**
  Vertausche ein Element mit dem unmittelbar vorangehenden nachdem auf das Element zugegriffen wurde
- ***FC - Regel*, Frequency Count:**
  Ordne jedem Element einen Häufigkeitszähler zu, der zu Beginn mit 0 initialisiert wird und der bei jedem Zugriff auf das Element um 1 erhöht wird. Nach jedem Zugriff wird die Liste neu angeordnet, so dass die Häufigkeitszähler in absteigender Reihenfolge sind.

Beispiel selbstanordnende Listen, MF-Regel

- Beispiel (für Worst Case)

Zugriff	(resultierende) Liste	Aufwand in zugegriffenen Elementen
	7-6-5-4-3-2-1	
1	1-7-6-5-4-3-2	7
2	2-1-7-6-5-4-3	7
3	3-2-1-7-6-5-4	7
4	4-3-2-1-7-6-5	7
5	5-4-3-2-1-7-6	7
6	6-5-4-3-2-1-7	7
7	7-6-5-4-3-2-1	7

- Durchschnittliche Kosten: 7x7/7

Beispiel selbstanordnende Listen, MF-Regel

- Beispiel (für „beinahe“ Best Case)

Zugriff	(resultierende) Liste	Aufwand in zugegriffene Elemente
	7-6-5-4-3-2-1	
1	1-7-6-5-4-3-2	7
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1

- Durchschnittliche Kosten: $7 + 6 \times 1/7 \approx 1.86$

Beispiel selbstanordnende Listen, MF-Regel

- **Feste Anordnung und naives Suchen** hat bei einer 7-elementigen Liste durchschnittlich den Aufwand:

$$\frac{1}{7} \times \sum_{i=1}^{7} i = \frac{1}{7} \times \frac{7 \times 8}{2} = 4$$

- Die *MF -Regel* kann also Vorteile haben gegenüber einer festen Anordnung
  - Dies ist insbesondere der Fall, wenn die Suchschlüssel stark gebündelt auftreten
  - Näheres zu selbstanordnenden Listen findet man im Buch von *Ottmann und Widmayer*

Danksagung

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus:

- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 2: Suchstrukturen) gehalten von Christian Scheideler an der TUM http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

- „Algorithmen und Datenstrukturen“ gehalten von Sven Groppe an der UzL

Suchstruktur für die Darstellung von Mengen

s: Menge von Elementen

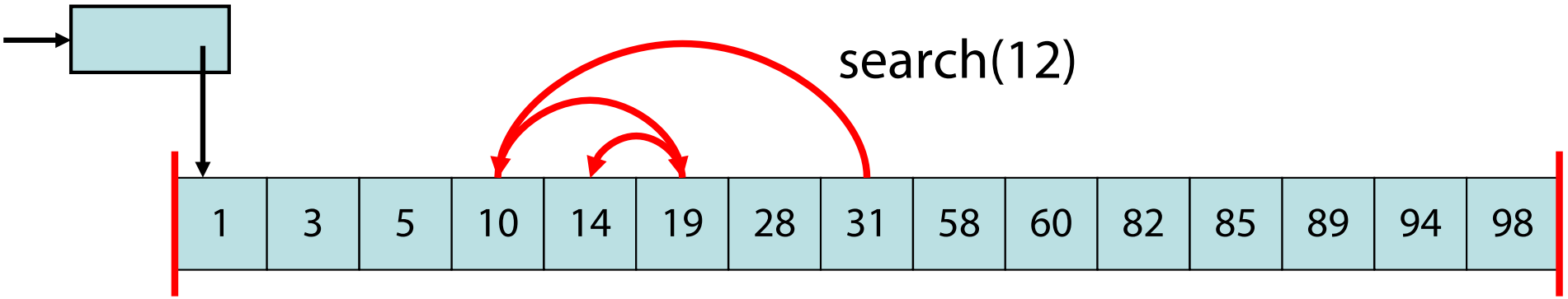
Jedes Element x identifiziert über key(x).

Operationen:

- insert(x, s): modifiziere s auf $s \cup \{x\}$
- delete(k, s): modifiziere s auf $s \setminus \{x\}$, sofern ein Element x enthalten ist mit key(x)=k
- search(k, s): gib $x \in s$ aus mit minimalem key(x) so dass $key(x) \geq k$ (gib nil zurück, wenn ein solches x nicht vorhanden)

Statische Suchstruktur

1. Idee: Speichere Elemente in sortiertem Feld.



search: über binäre Suche (O(log n) Zeit)

Binäre Suche

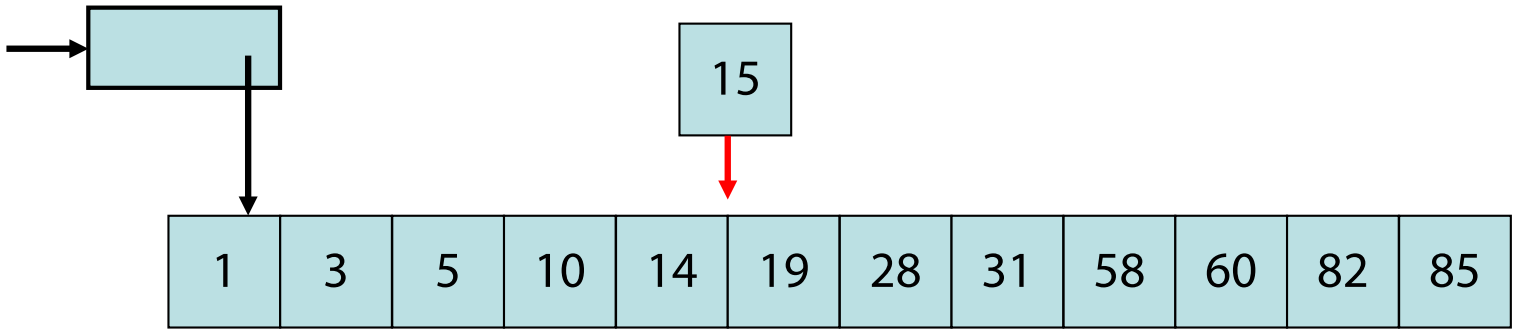
Eingabe: Zahl x und ein sortiertes Feld A[1],…,A[n]

```
function search(k, A)
n := length(A); l := 1; r := n
while l < r do
   m:=(r+l) div 2
   if key(A[m]) = k then return A[m]
   if key(A[m]) < k then l:=m+1
   else r:=m
return A[l]
```

Dynamische Suchstruktur

insert und delete Operationen:

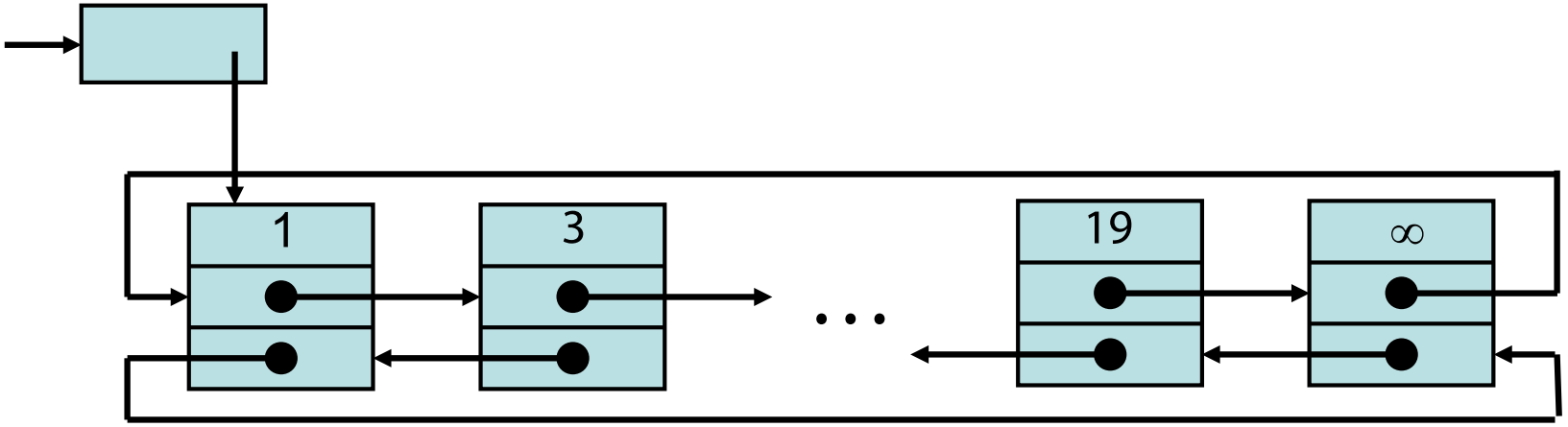
Sortiertes Feld schwierig zu aktualisieren!



Worst case: $\theta(n)$ Zeit

Suchstruktur

2. Idee: Sortierte Liste

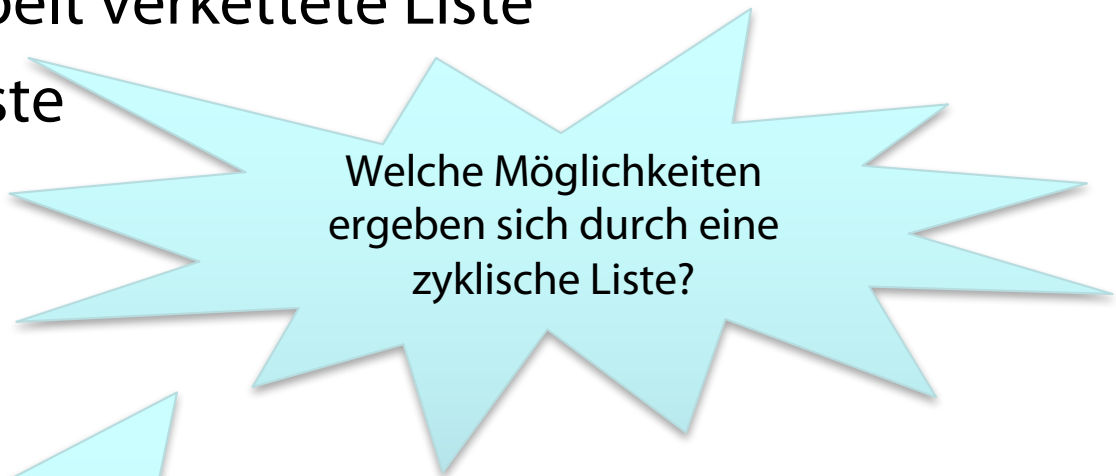


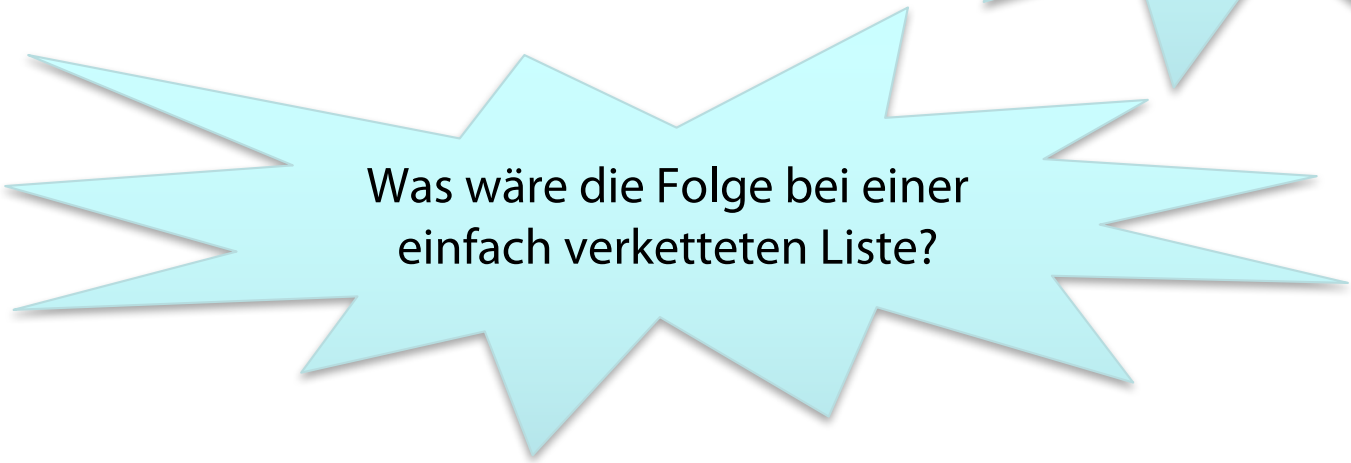
Problem: insert, delete und search kosten im worst case $\theta(n)$ Zeit

Einsicht: Wenn search effizient zu implementieren wäre, dann auch alle anderen Operationen

Suchstruktur

- Alternativen
  - Nicht-zyklische, doppelt verkettete Liste
  - Einfach verkettete Liste
  - Baum



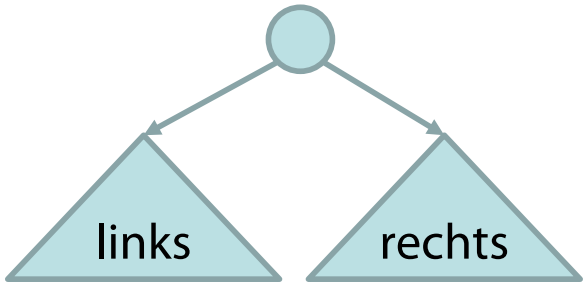


Motivation für Suche nach neuer Trägerstruktur

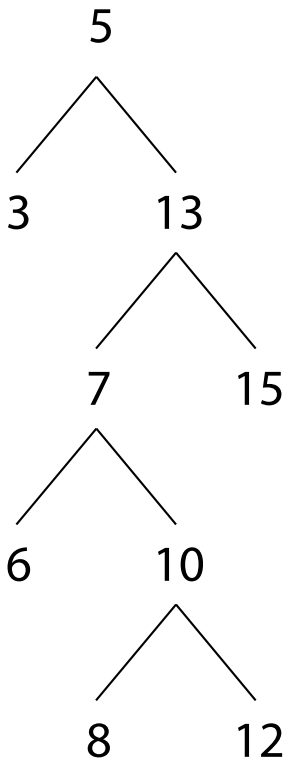
- Halbierungssuche durchsucht Felder
  - Einfügen neuer Elemente erfordert ggf. ein größeres Feld und Umkopieren der Elemente
- Einfügen in Listen in konstanter Zeit
  - Zugriffe auf Elemente jedoch sequentiell
- (Verzeigerter) Baum
  - Einfügen in konstanter Zeit möglich (falls Einfügeposition gegeben)
  - zum Suchen verwendbar

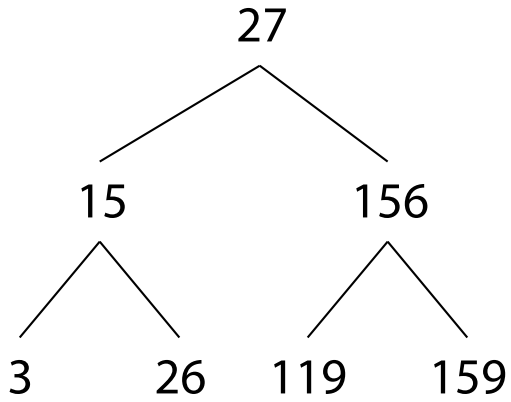
Binäre Suchbäume - Definition

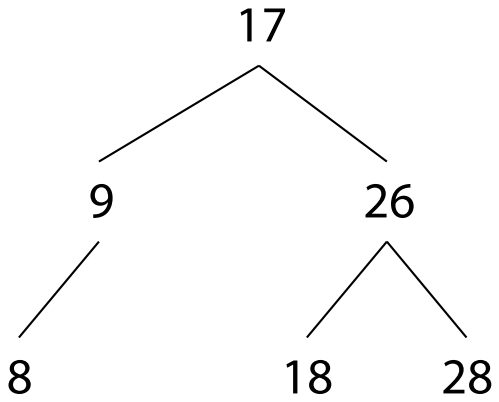
- Ein binärer Suchbaum
  1. ist ein Binärbaum, und
  2. zusätzlich muss für jeden seiner Knoten gelten, dass das im Knoten *gespeicherte Element*
     a) ***größer*** ist als alle *Elemente* im *linken Unterbaum*
     b) ***kleiner*** ist als alle *Elemente* im *rechten Unterbaum*



Beispiele für binäre Suchbäume

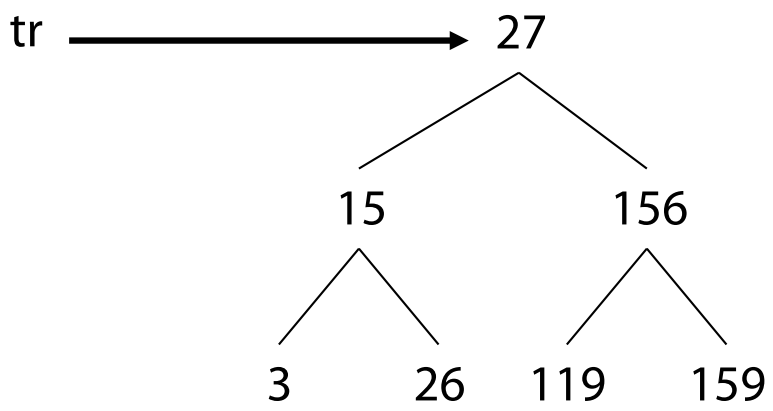






Aufgabe: Wende Funktion auf Elemente an

- Algorithmus?



fold(+, tr, 0)
→ 505

function max (x, y)
if x > y **then**
return x
else return y

fold(max, tr, 0)
→ 159

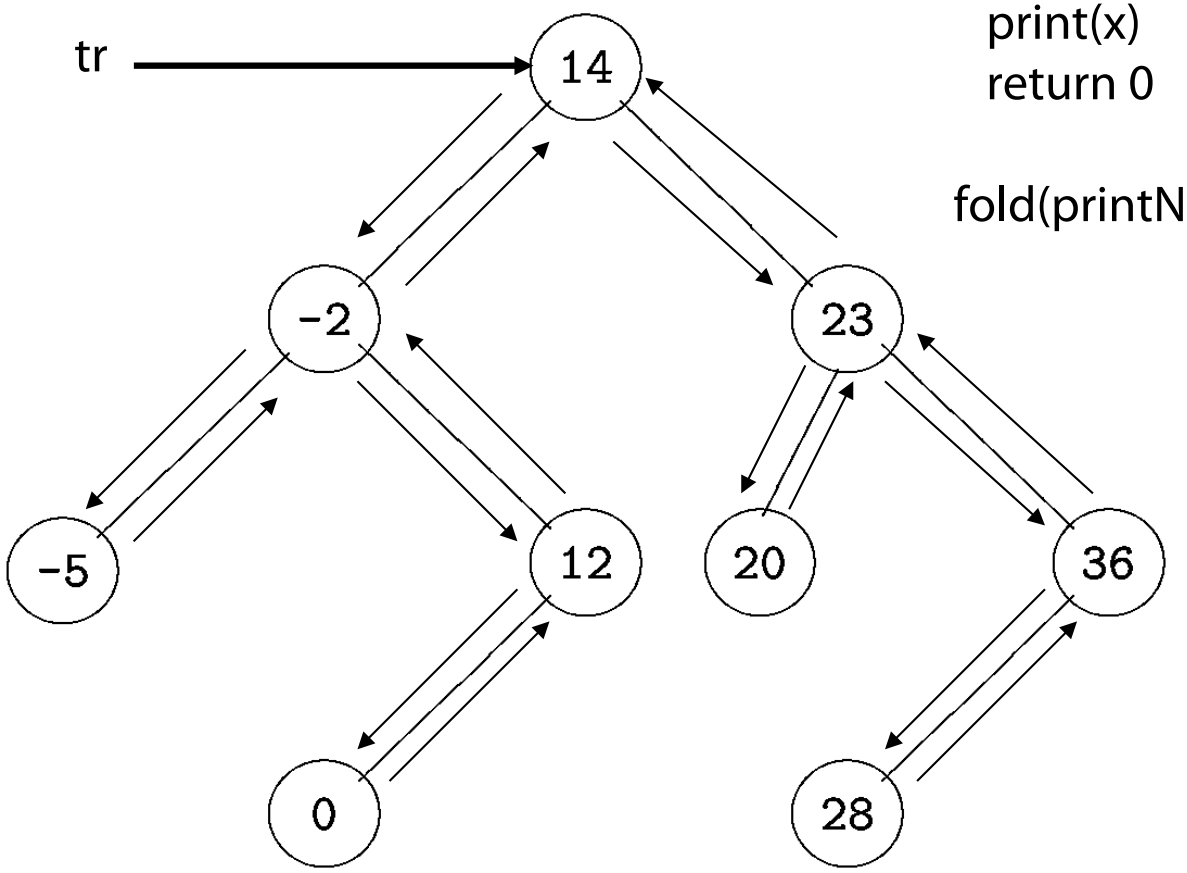
function min (x, y)
if x > y **then**
return y
else return x

fold(min, tr, ∞)
→ 3

Aufgabe: Wende Funktion auf Elemente an

```
function fold(f, tr, init)
  if emptyTree(tr) then
    return init
  if leaf(tr) then
    return f(init, key(tr))
  if leftExists(tr) then
    x := f(  fold(f, left(tr), init),  key(tr) )
    if rightExists(tr) then
      return fold(f, right(tr), x)
    else return x
  if rightExists(tr) then // linker Nachfolger existiert nicht
    return f( fold(f, right(tr), init),  key(tr) )
```

Inorder-Ausgabe ergibt Sortierreihenfolge



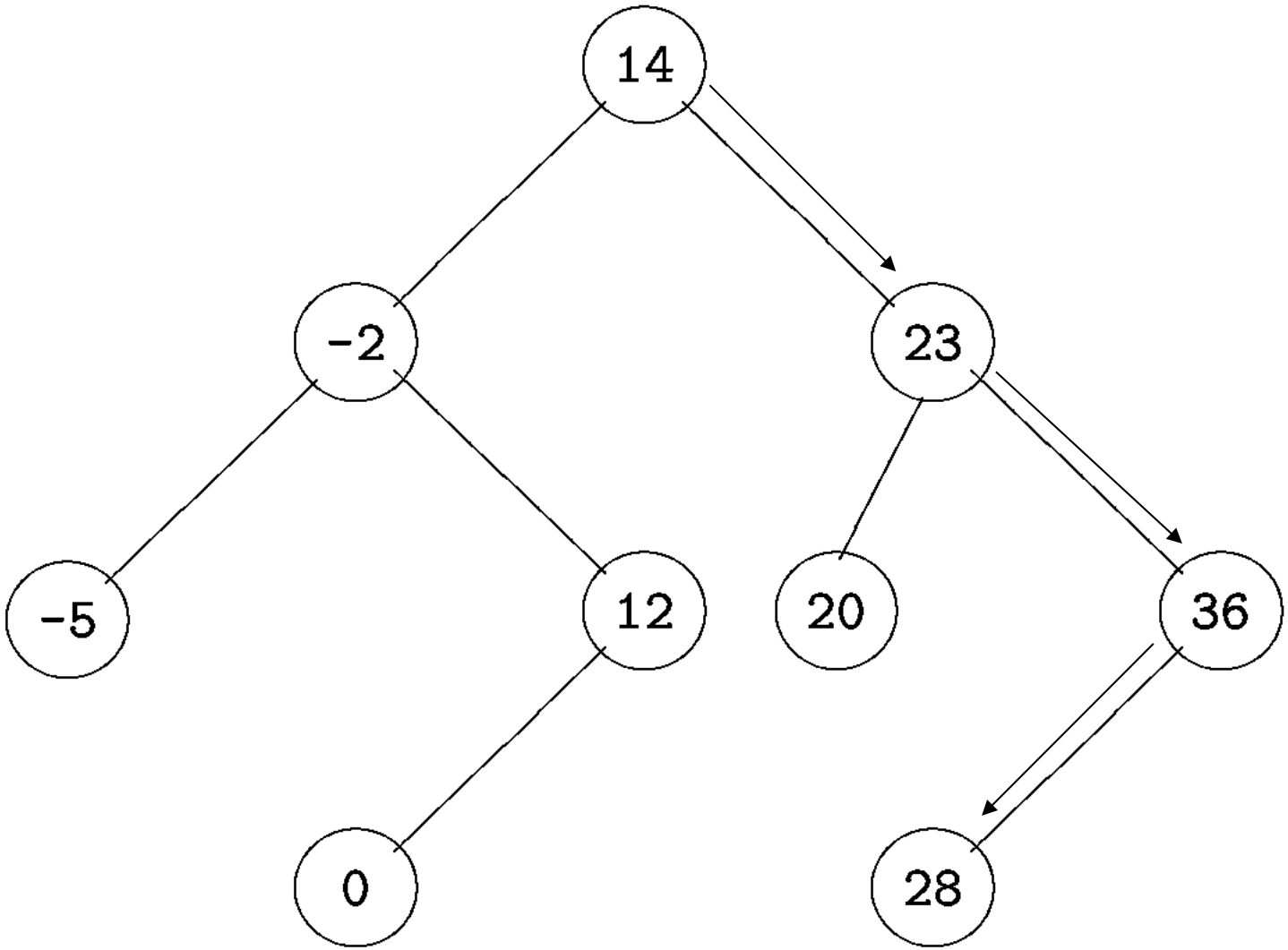

```
function printNode(ignore, x)
  print(x)
  return 0

fold(printNode, tr, 0)
```

-5 -2 0 12 14 20 23 28 36

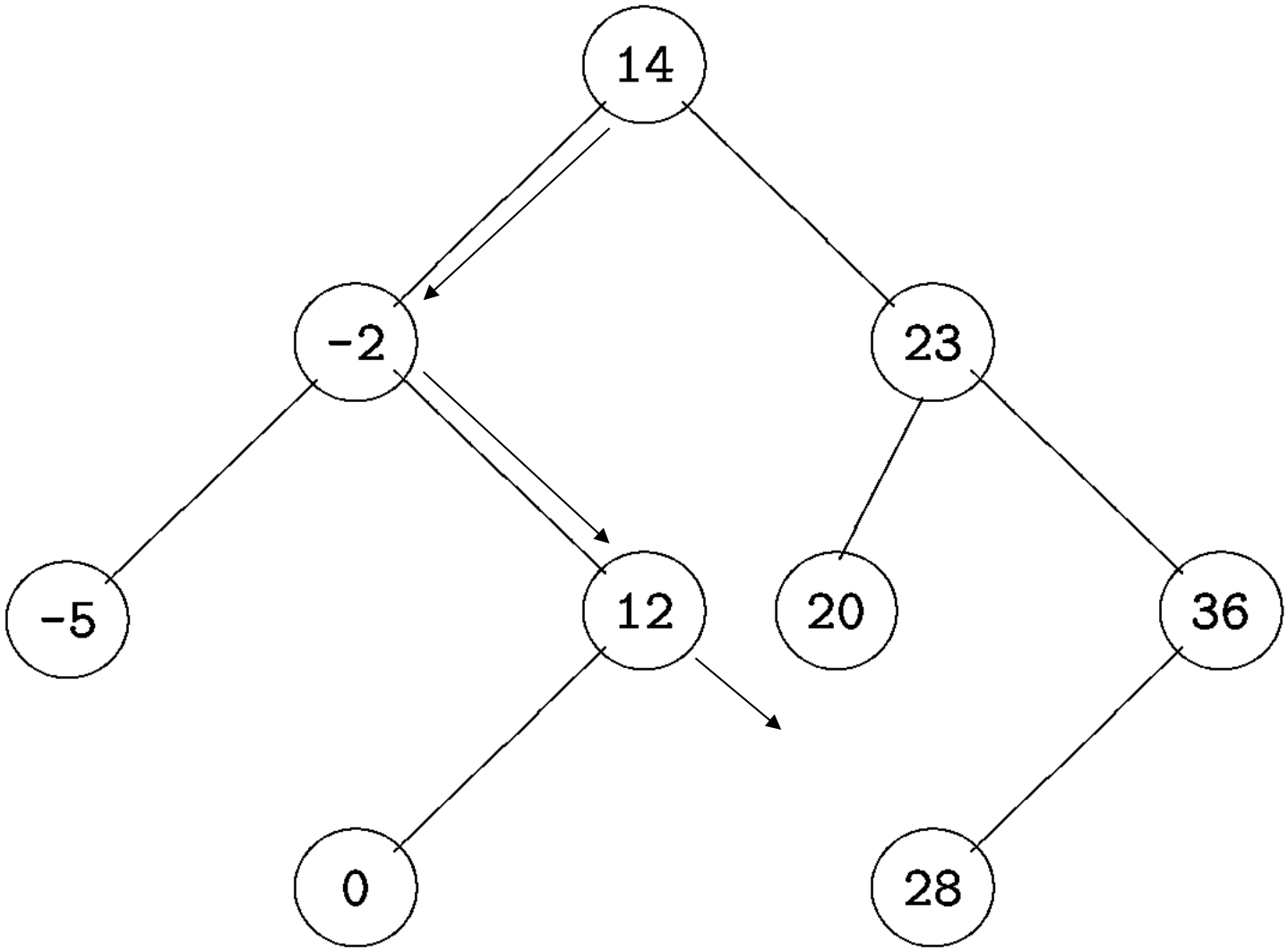
Suche im binären Suchbaum

- Suche nach einem Element im binären Suchbaum:
  - Baum ist leer:
    - Element nicht gefunden
  - Baum ist nicht leer:
    - Wurzelelement ist gleich dem gesuchten Element:
      - Element gefunden
    - Gesuchtes Element ist kleiner als das Wurzelelement:
      - Suche im linken Unterbaum rekursiv
    - Gesuchtes Element ist größer als das Wurzelelement:
      - Suche im rechten Unterbaum rekursiv

Suche nach 28



Suche nach 13

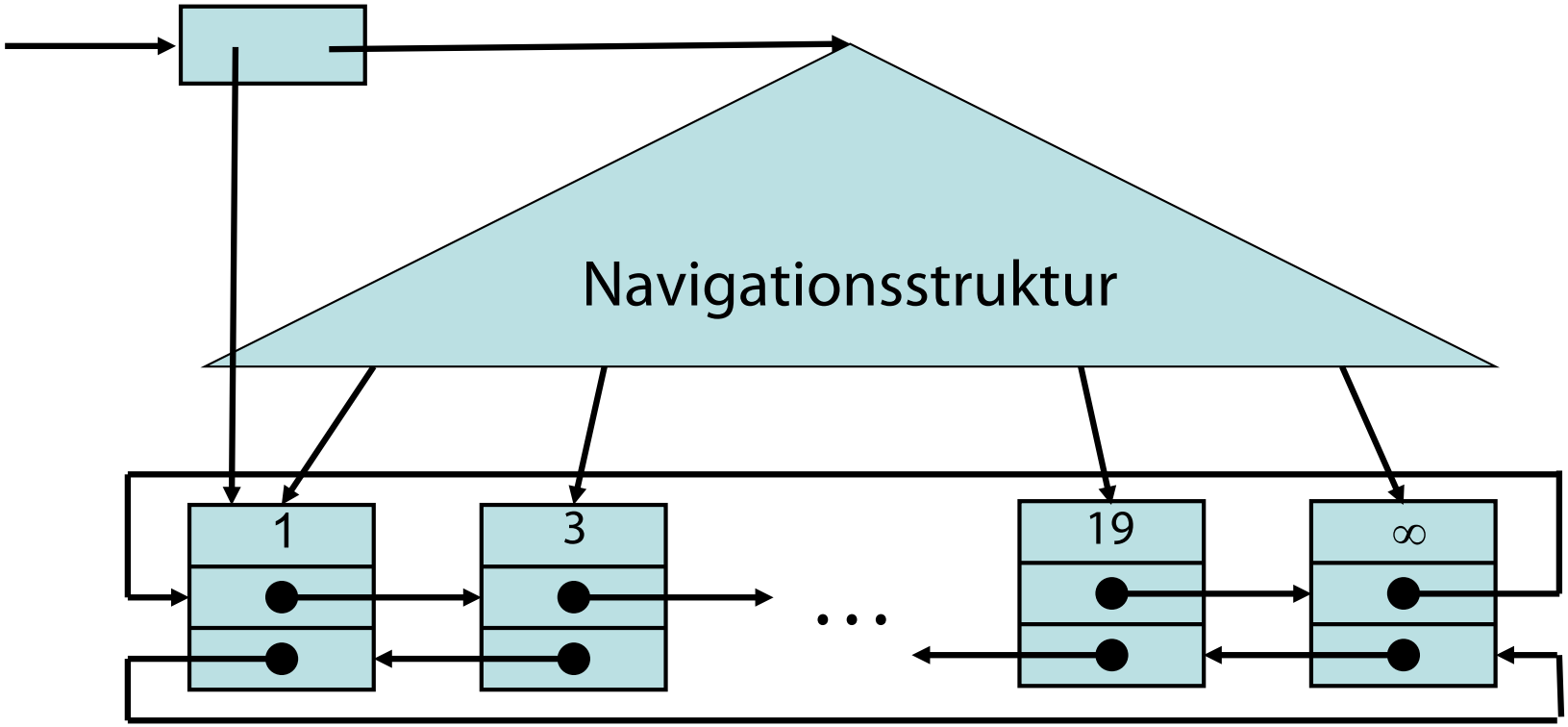


Binärer Suchbaum

- Wie kann man Iteratoren realisieren?
- getIterator(s) bzw. getIterator(s, firstKey)

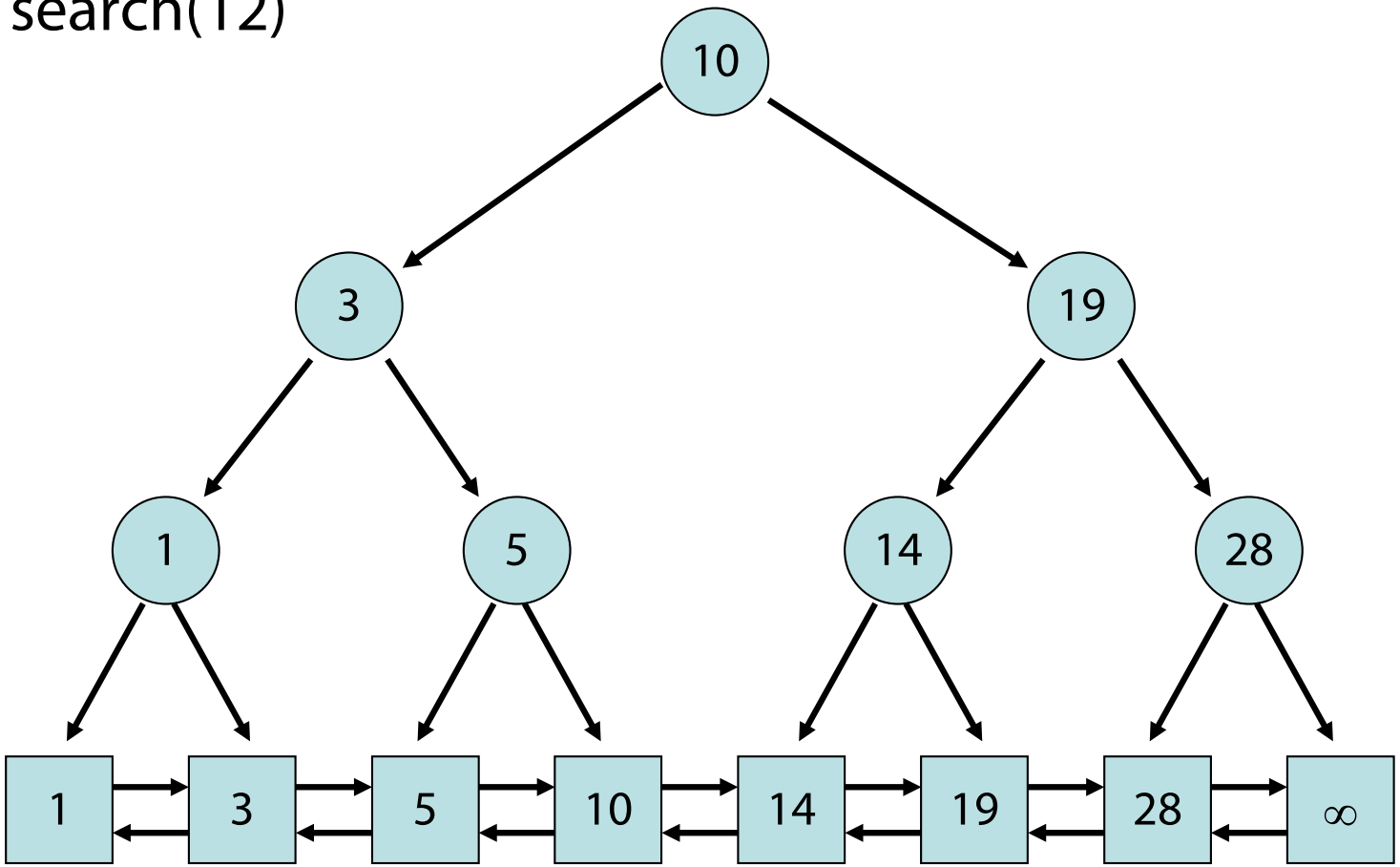
Suchstruktur

- Idee: füge Navigationsstruktur hinzu, die search effizient macht, Navigationsstruktur enthält nur Schlüssel



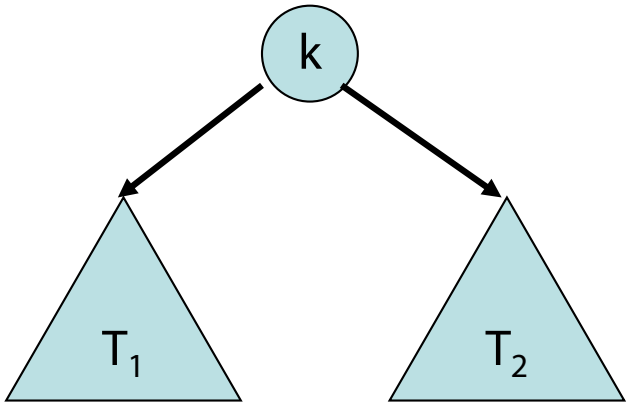
Binärer Suchbaum (ideal)

search(12)



Binärer Suchbaum

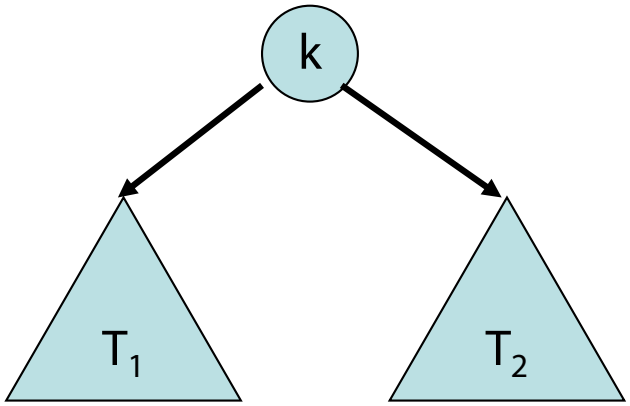
Suchbaum-Regel:



Für alle Schlüssel k' in T_1 und k'' in T_2: $k' \leq k < k''$

- Damit lässt sich die search Operation einfach implementieren.

search(k) Operation



Für alle Schlüssel k' in T_1 und k'' in T_2: $k' \leq k < k''$

Suchstrategie:

- Starte in Wurzel des Suchbaums
- Für jeden erreichten Knoten v:
  - Falls key(v) ≤ k, gehe zum linken Kind von v, sonst gehe zum rechten Kind

Binärer Suchbaum

Für einen Baumknoten v sei

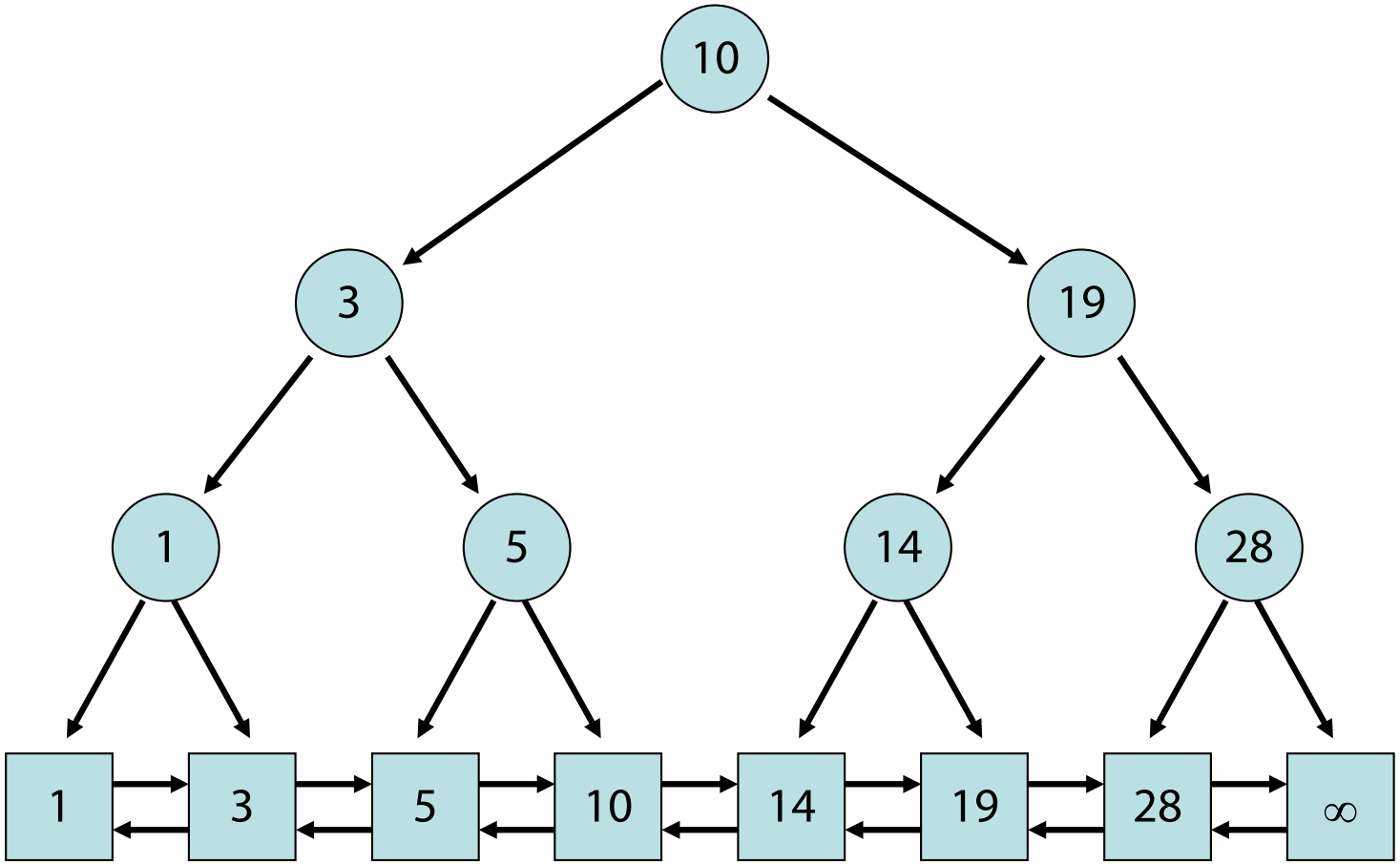
- key(v) der Schlüssel in v
- d(v) die Anzahl Kinder von v

- Suchbaum-Regel: (s.o.)

- Grad-Regel: (Grad = Anzahl der Kinder)
  Alle Baumknoten haben zwei Kinder
  (sofern #Elemente >1)

- Schlüssel-Regel:
  Für jedes Element e in der Liste gibt es genau einen Baumknoten v mit key(v)=key(e).

Search(9)

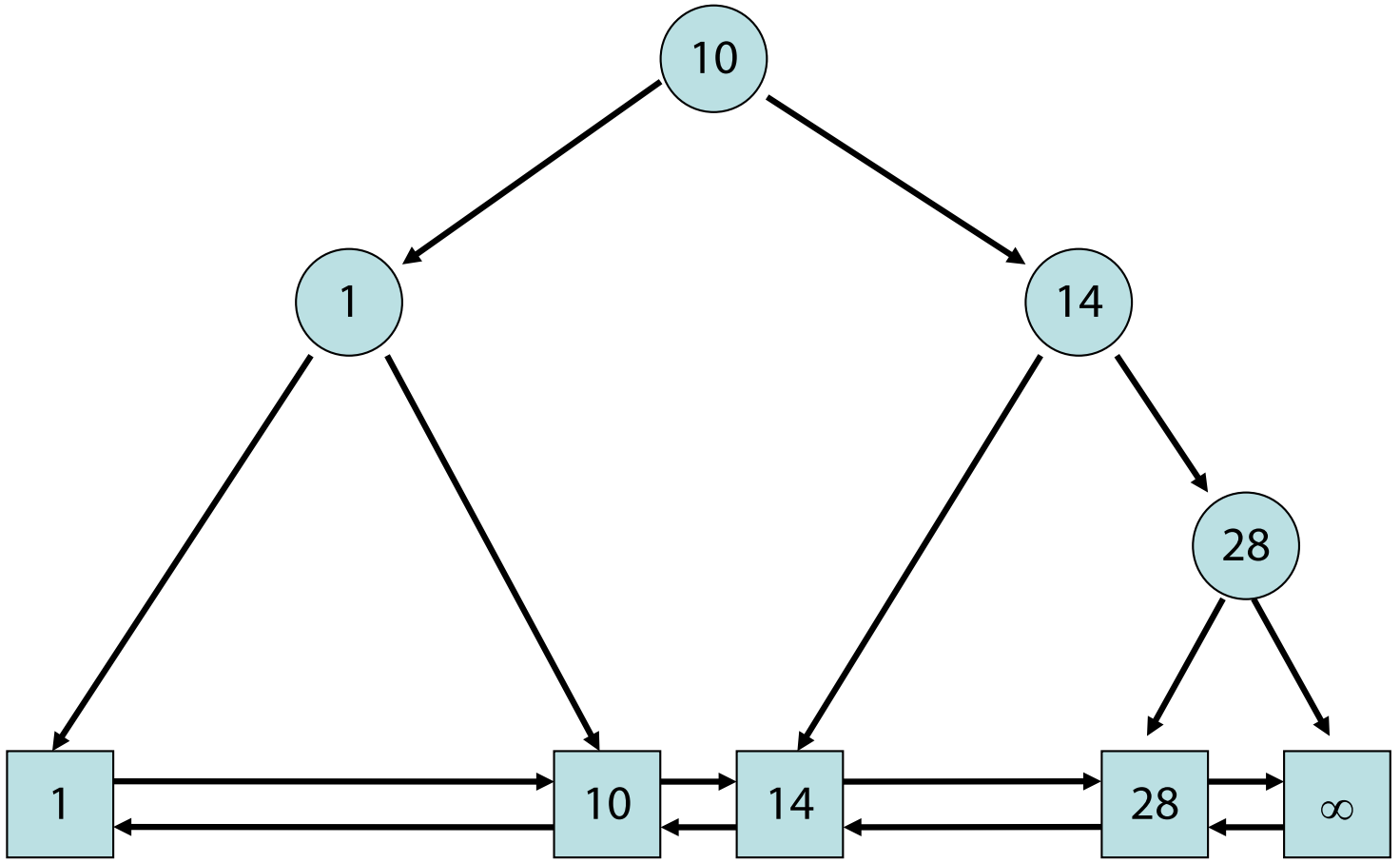


Insert und Delete Operationen

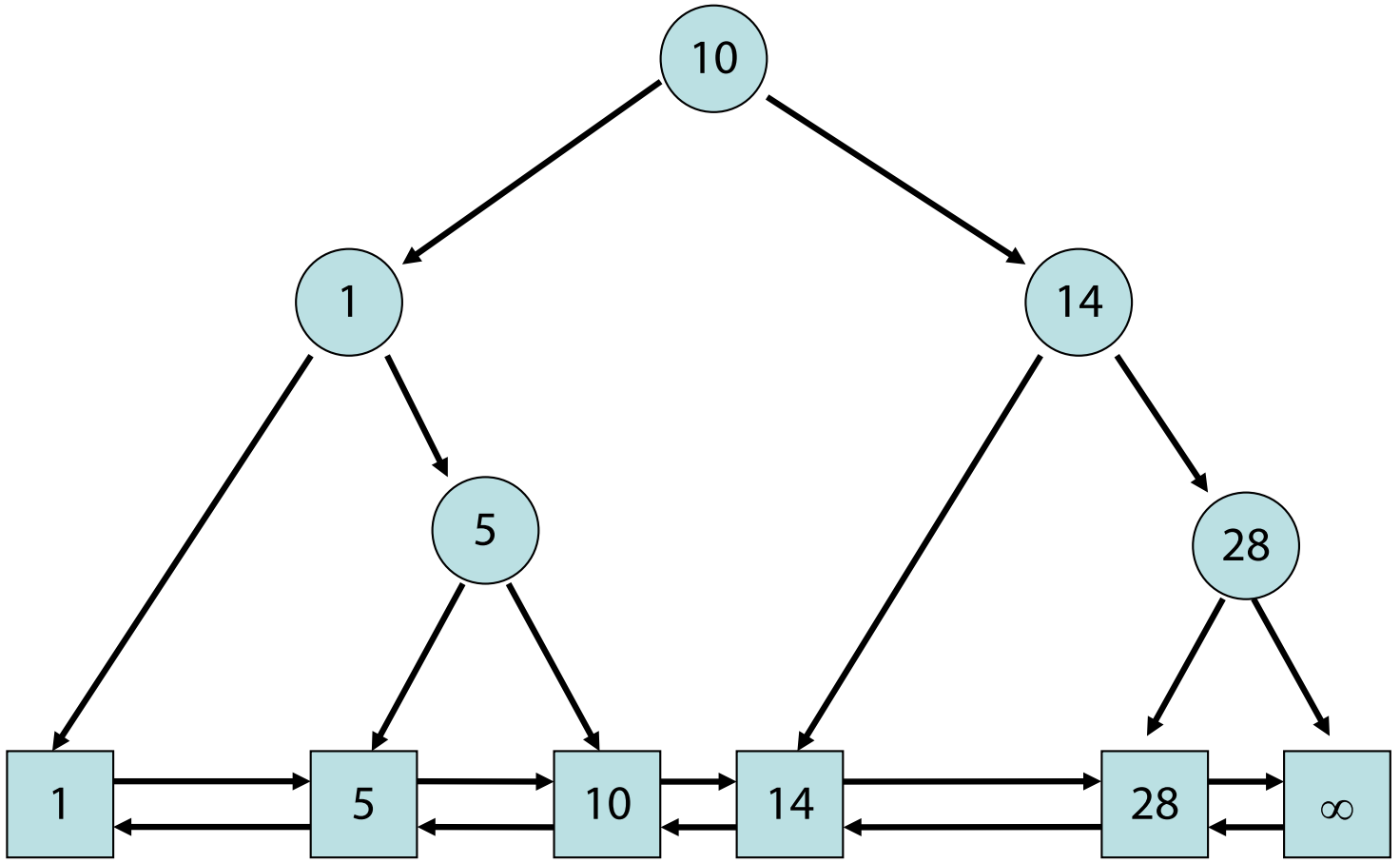
Strategie:

- insert(e):
  Erst search(key(e)) bis Element e' in Liste erreicht. Falls key(e')>key(e), füge e vor e' ein und ein neues Suchbaumblatt für e und e' mit key(e), so dass Suchbaum-Regel erfüllt.
- delete(k):
  Erst search(k) bis ein Element e in Liste erreicht. Falls key(e)=k, lösche e aus Liste und Vater v von e aus Suchbaum, und setze in dem Baumknoten w mit key(w)=k: key(w):=key(v)

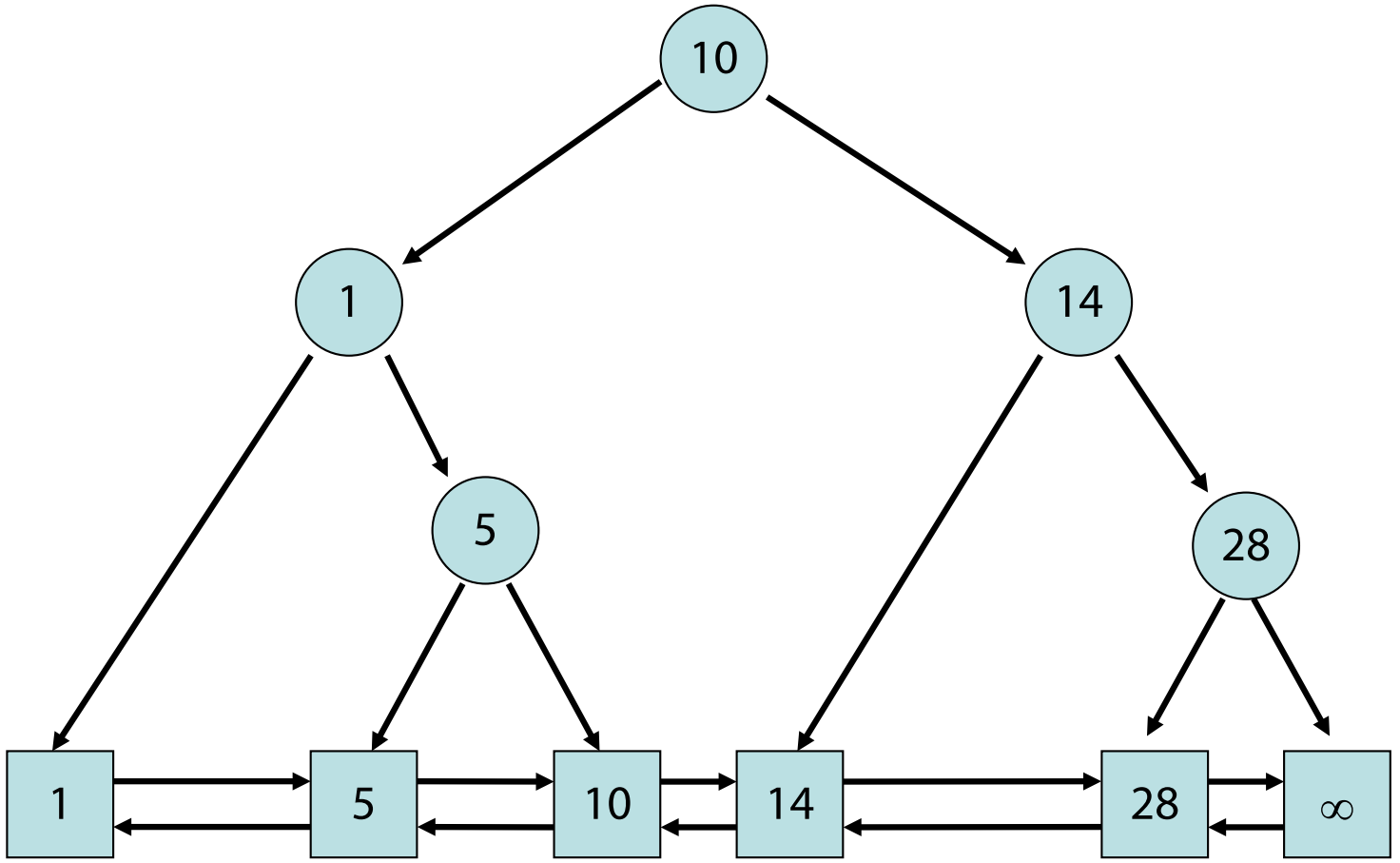
Insert(5)



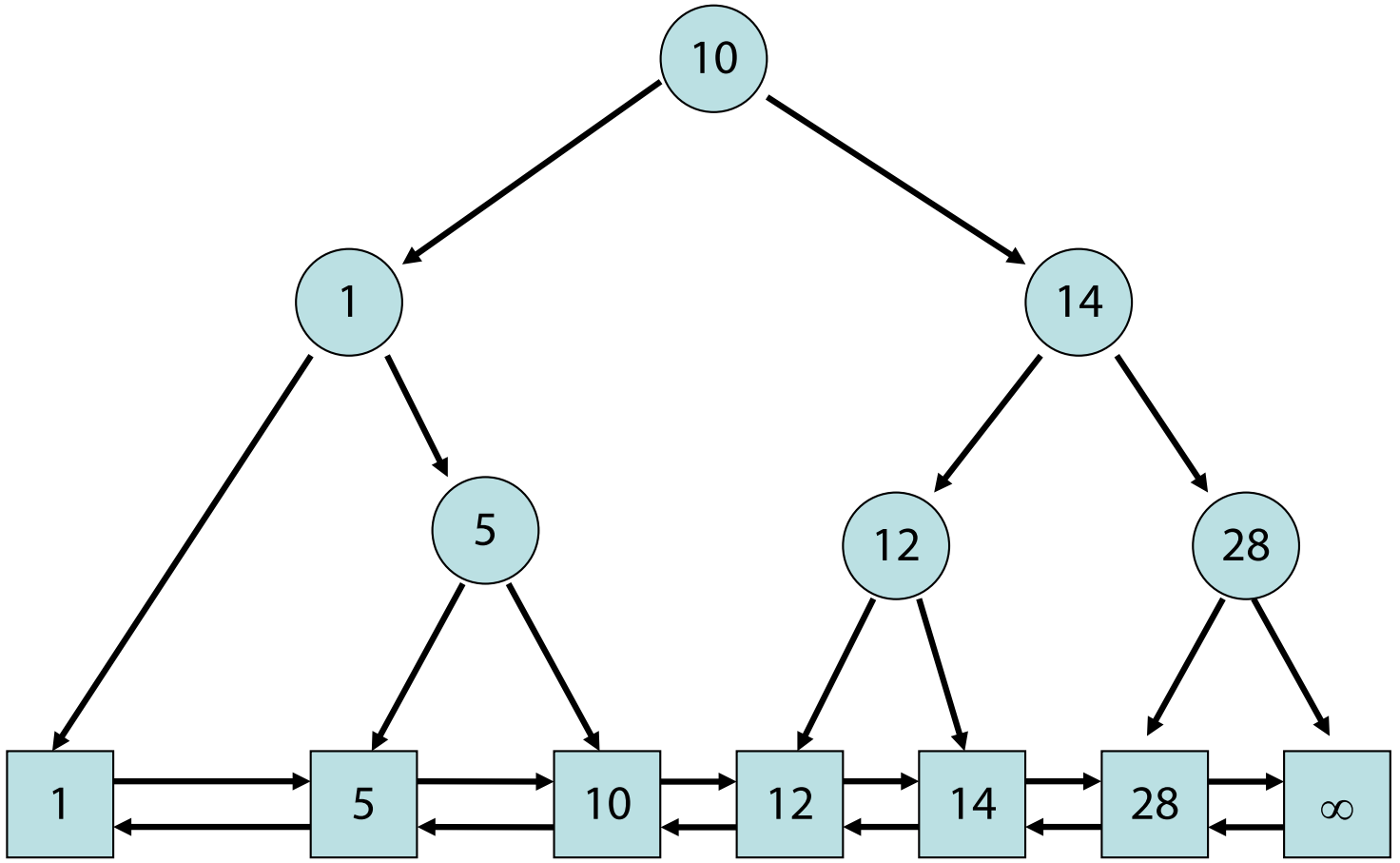
Insert(5)



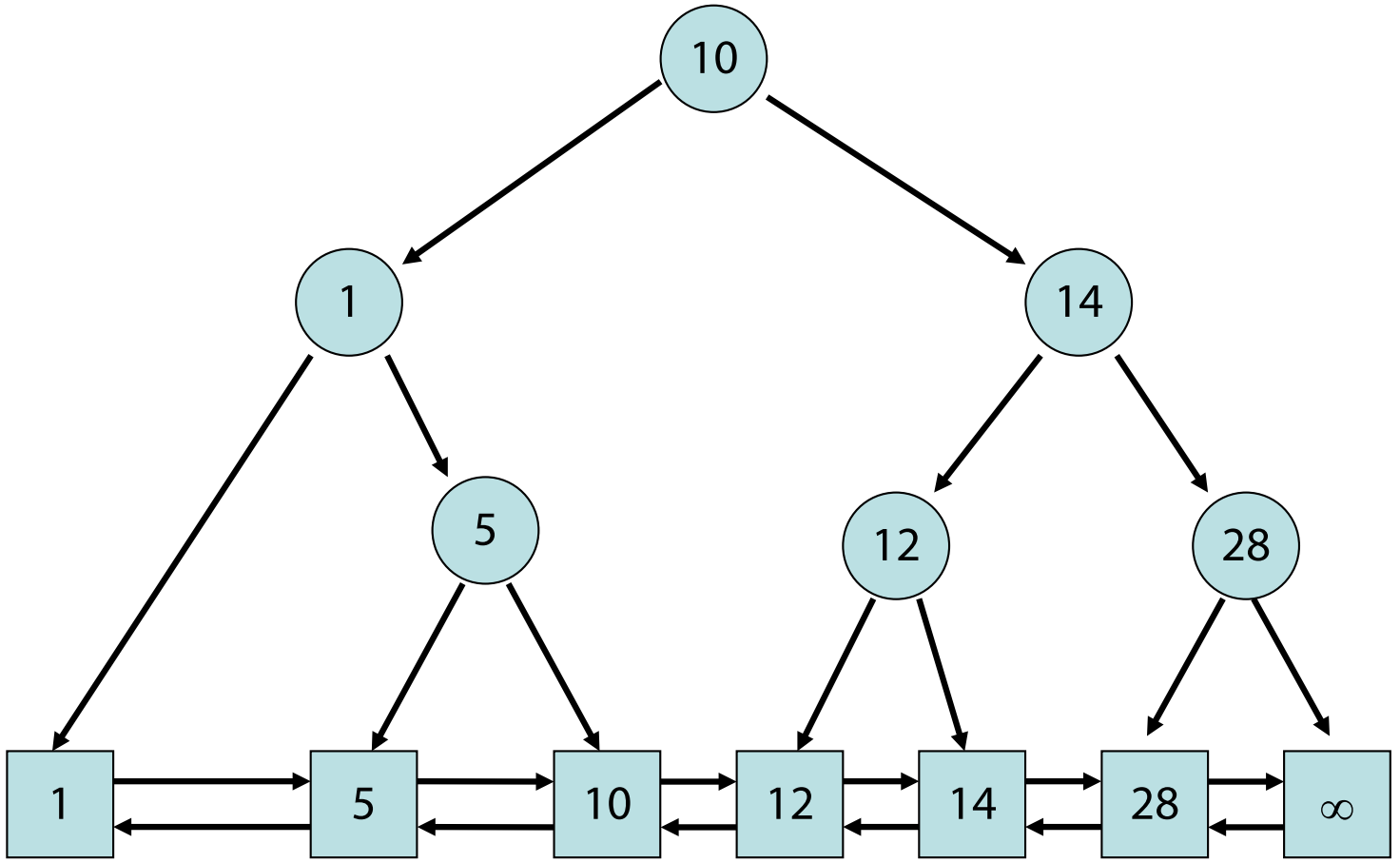
Insert(12)



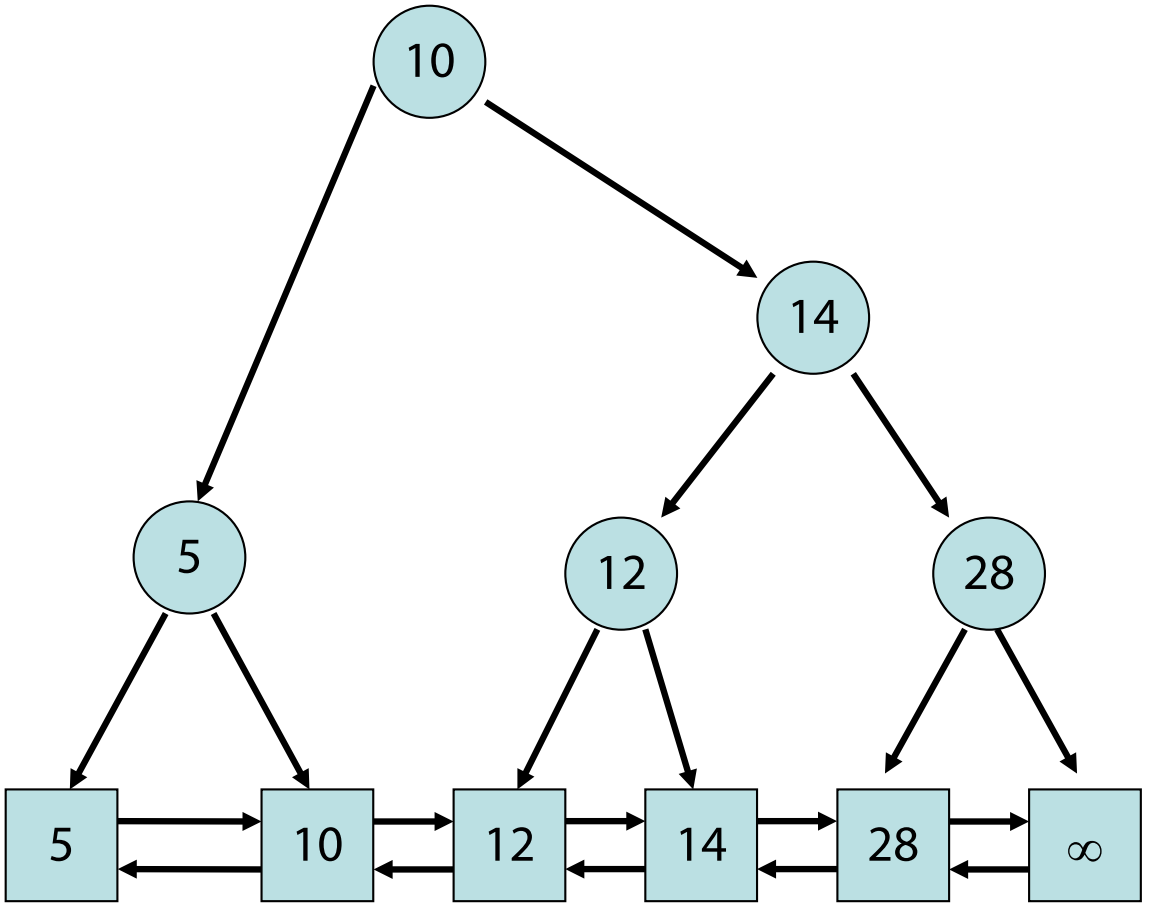
Insert(12)



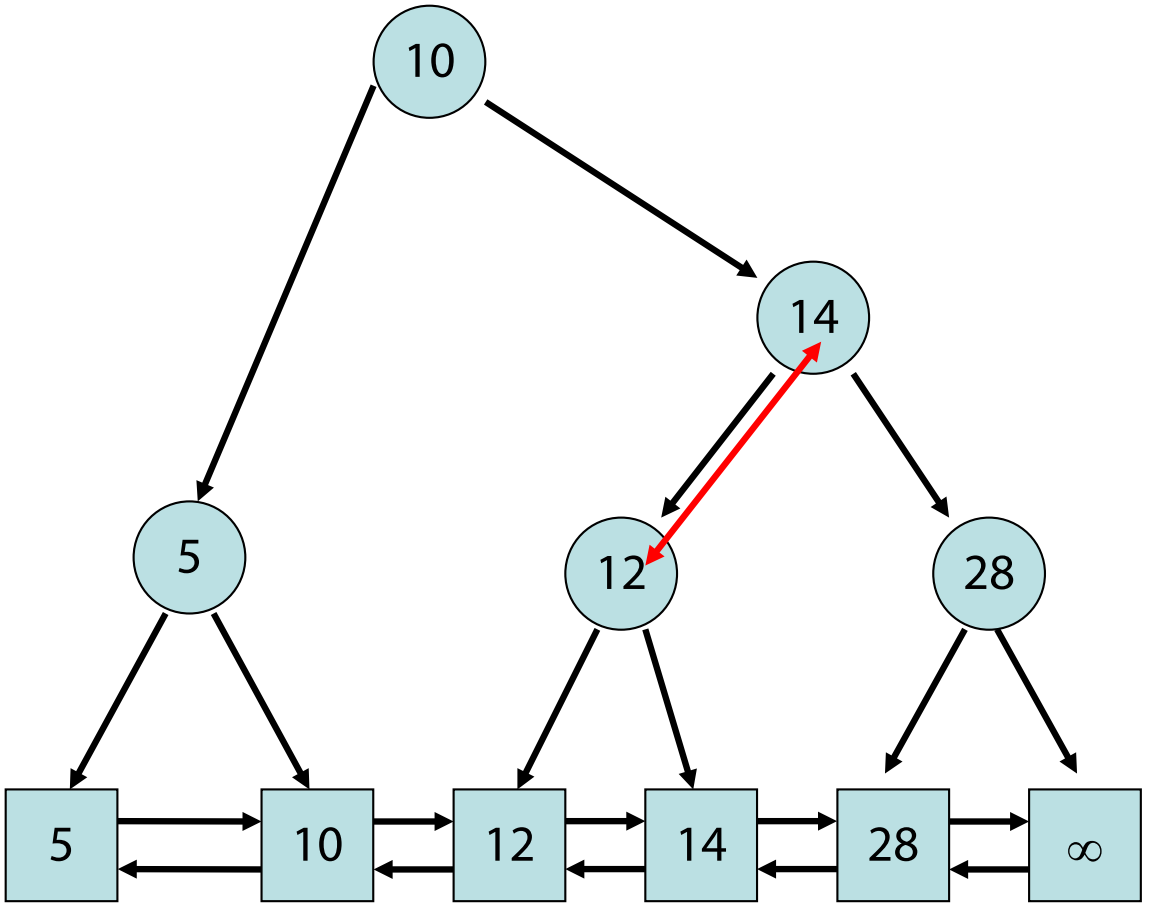
Delete(1)



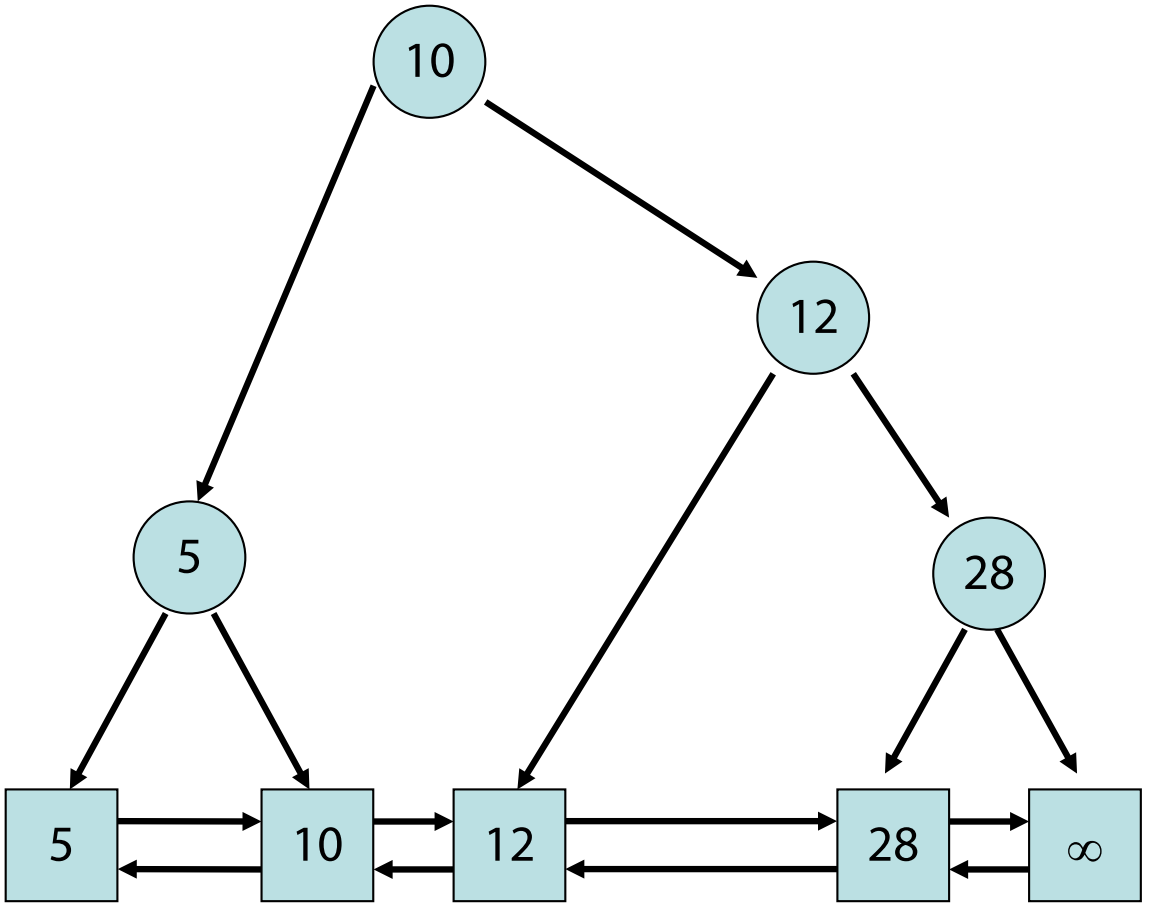
Delete(1)



Delete(14)



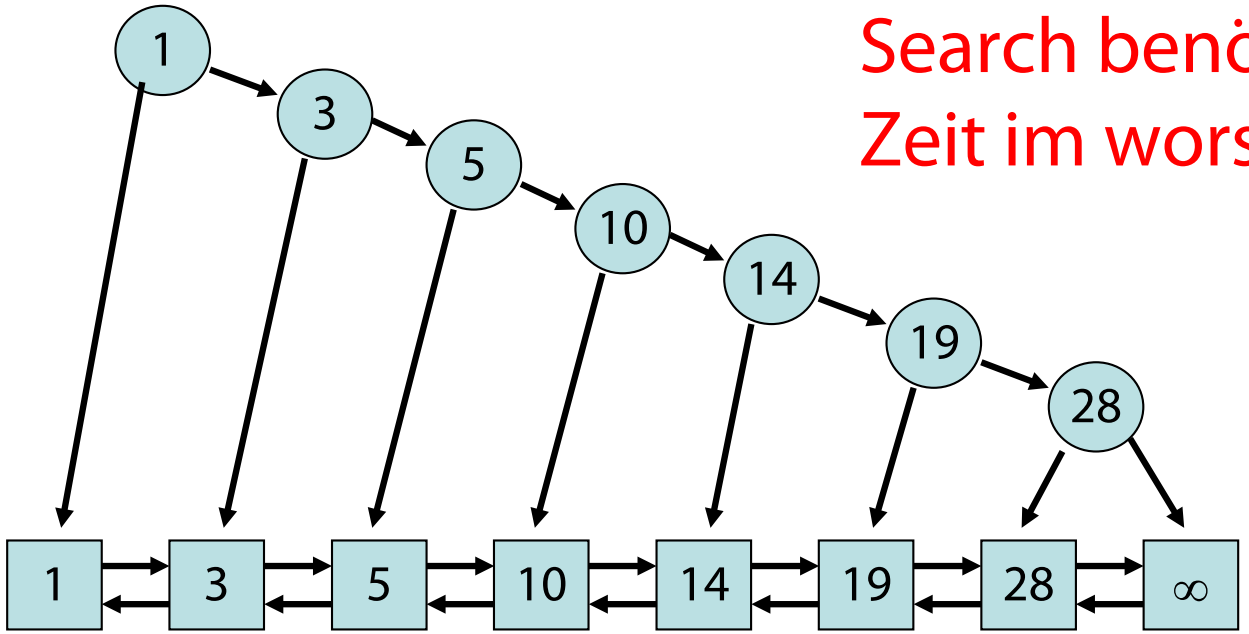
Delete(14)



Binärbaum

Problem: Binärbaum kann entarten!

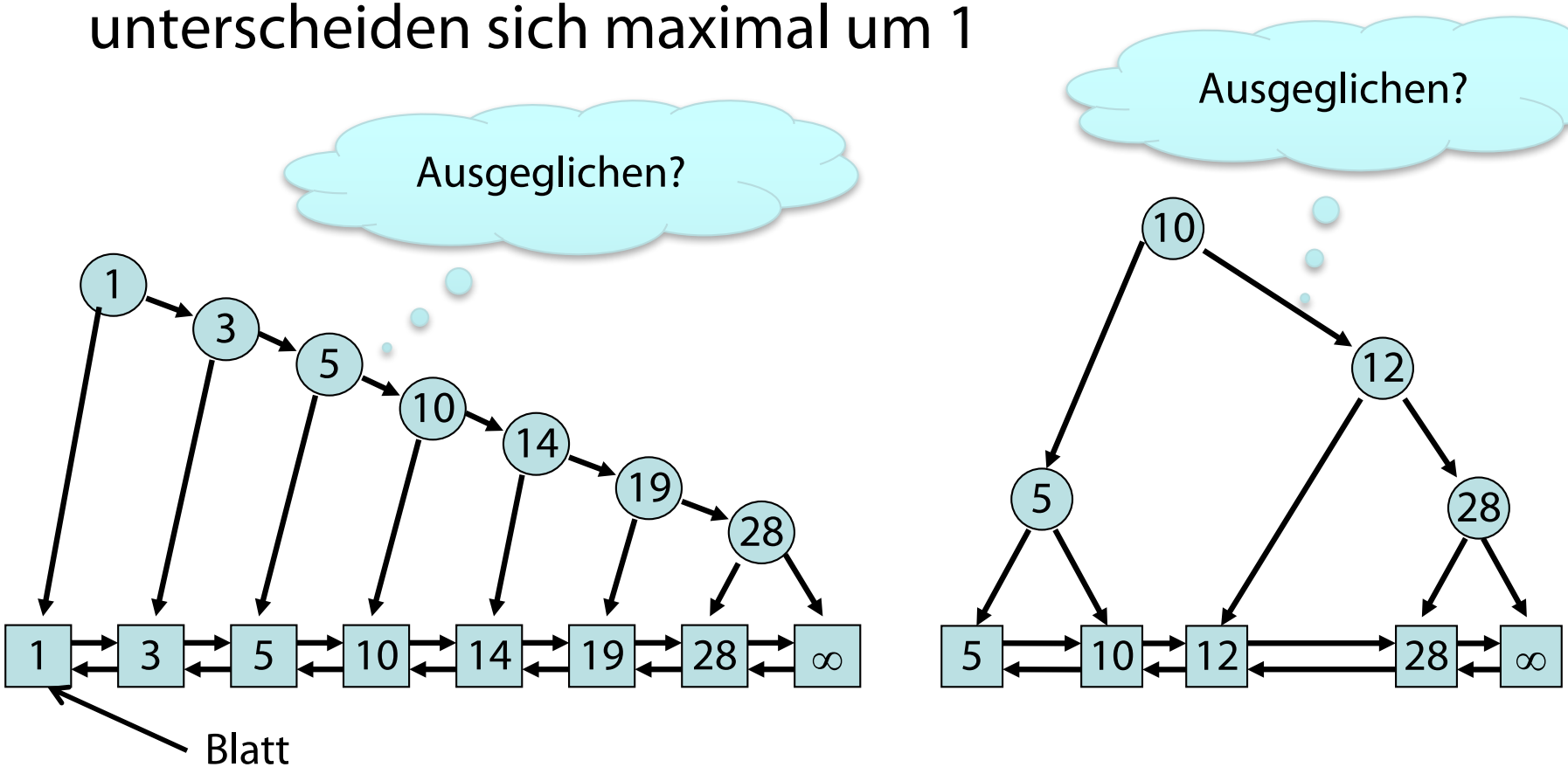
Beispiel: Zahlen werden in sortierter Folge eingefügt



Search benötigt $\theta(n)$
Zeit im worst case

Definition: Ausgeglichener Suchbaum

- Die Längen der Pfade von den Blättern zur Wurzel unterscheiden sich maximal um 1



- „Alle Ebenen bis auf Blattebene voll gefüllt“

Gewichtete Binärbäume

- Ausgeglichenheit nur optimal, wenn relative Häufigkeit des Zugriffs bei allen Schlüsseln gleich
- Ist dies nicht der Fall, sollte relative Zugriffshäufigkeit bei der Baumkonstruktion berücksichtigt werden
- Idee: Ordne den Schlüsseln Gewichte zu
  - Häufiger zugegriffene Schlüssel: hohes Gewicht
  - Weniger oft zugegriffene Schlüssel: kleines Gewicht
- Knoten mit Schlüsseln, denen ein höheres Gewicht gegeben wird, sollen weiter oben stehen (interne Bäume)
- Wir besprechen später, wie solche Bäume erstellt werden können

Selbstorganisierende Bäume

- Man beachte: Suchaufwand $\Theta(\log n)$
  - Elementtests mehrfach mit dem gleichen Element:
    → dann $O(\log n)$ „zu teuer“
- Weiterhin: Mit bisheriger Technik des Einfügens kann Ausgeglichenheit nicht garantiert werden: Zugriff für bestimmte Elemente $> \log n$
  - Elementtest für diese Elemente häufig:
    → Performanz sinkt
- Idee: Häufig zugegriffene Elemente sollten trotz Unausgeglichenheit schneller gefunden werden
- Umsetzung: Splay-Baum (selbstorganisierend)

Daniel D. Sleator, Robert Tarjan: Self-Adjusting Binary Search Trees, In: Journal of the ACM (Association for Computing Machinery). 32, Nr. 3, S. 652–686, **1985**

Splay-Baum

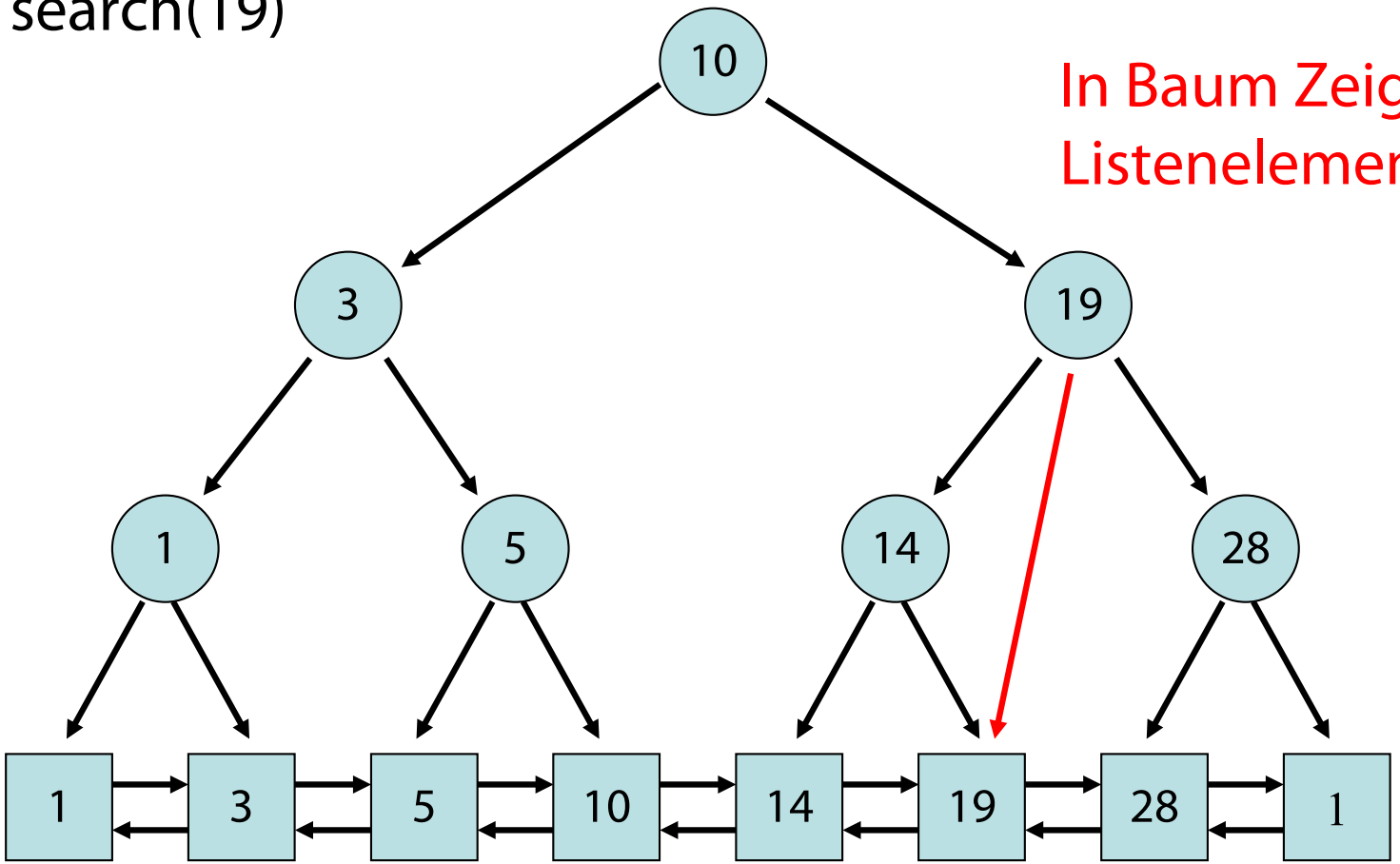
Üblicherweise: Implementierung als interner Suchbaum (d.h. Elemente direkt integriert in Baum und nicht in extra Liste)

Hier: Implementierung als externer Suchbaum (wie beim Binärbaum oben)

Modifikation nicht nur bei insert oder delete, sondern auch bei Anfragen

Splay-Baum

search(19)



In Baum Zeiger auf Listenelement

Splay-Baum

Ideen:

1. Im Baum Zeiger auf Listenelemente
2. Bewege Schlüssel von zugegriffenem Element immer zur Wurzel

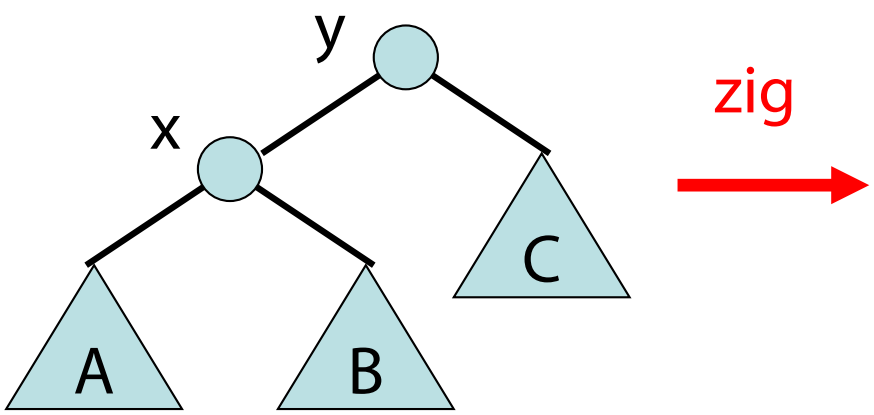
2. Idee: über Splay-Operation

Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

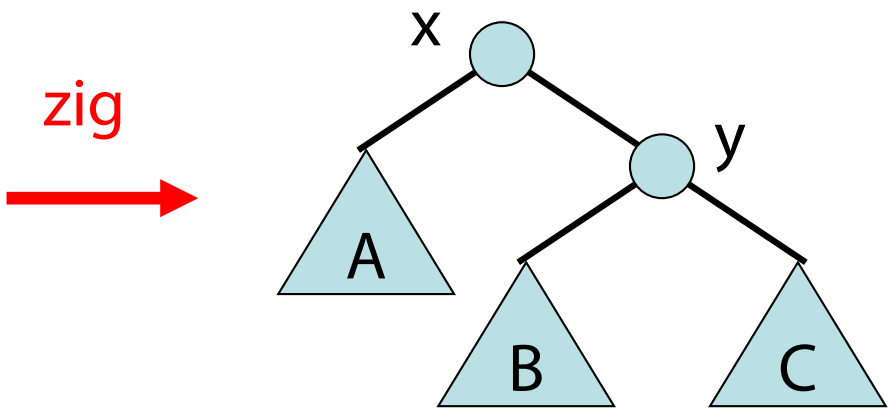


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

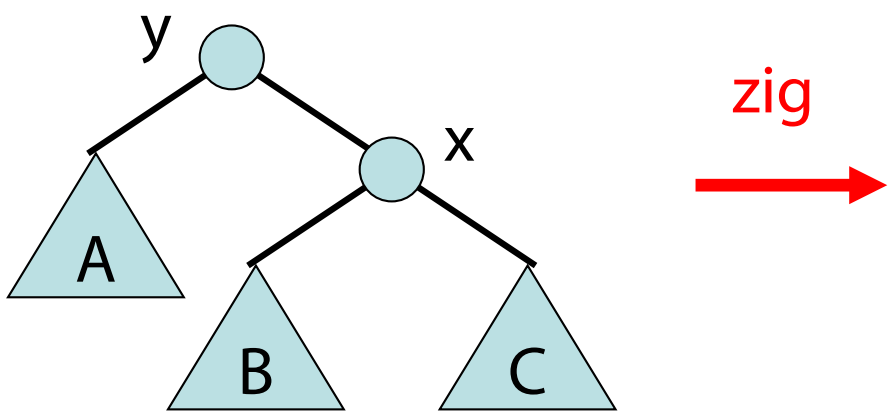


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1b. x ist Kind der Wurzel:

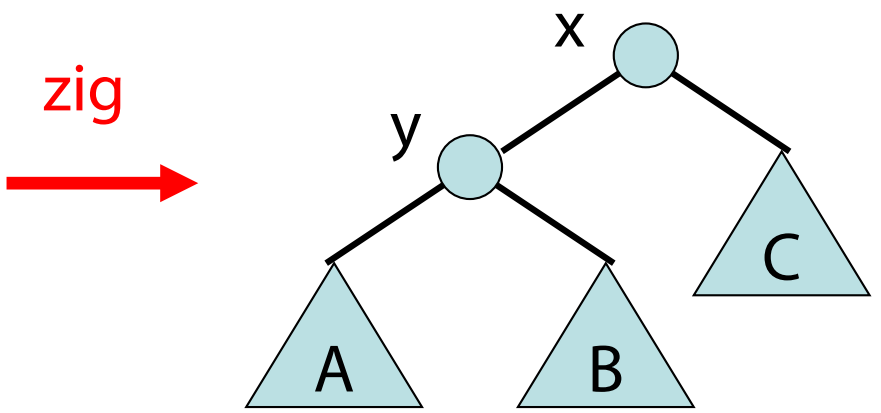


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

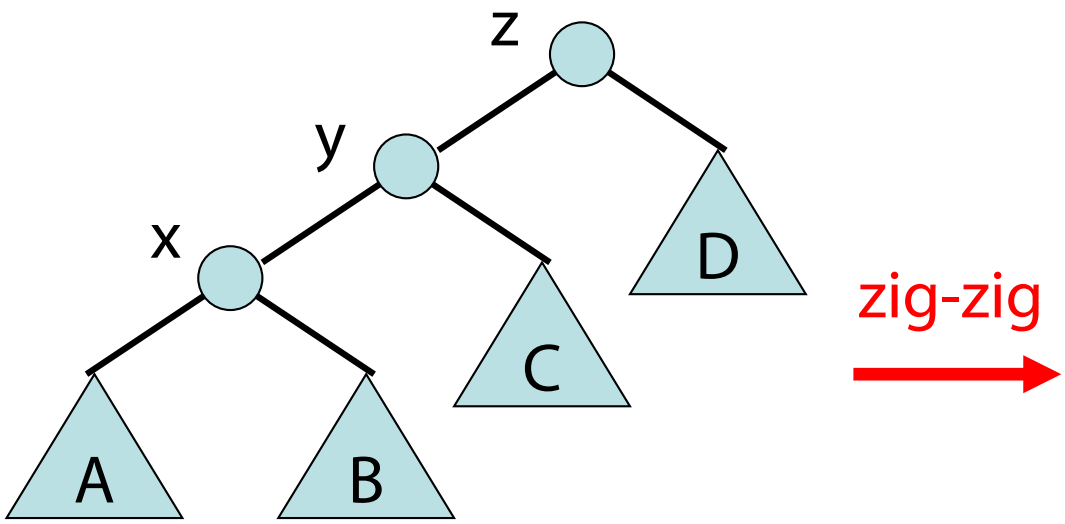
1b. x ist Kind der Wurzel:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

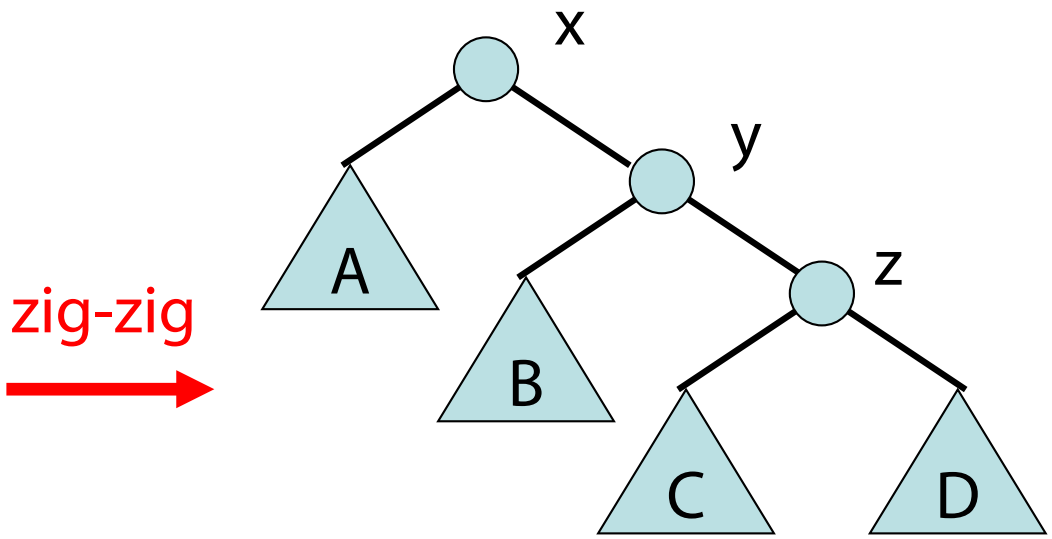
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

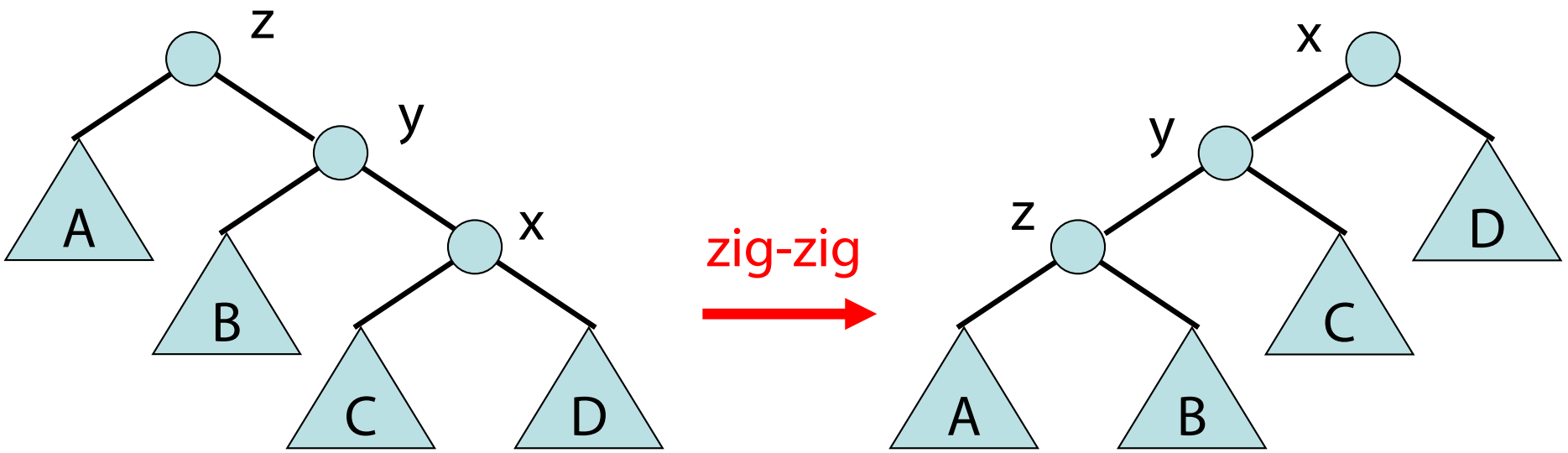
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

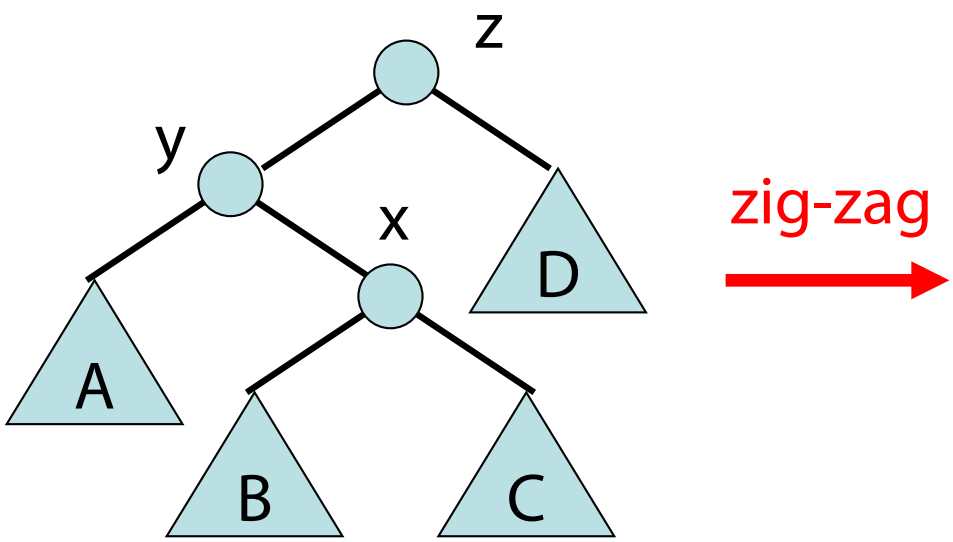
2b. x hat Vater und Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

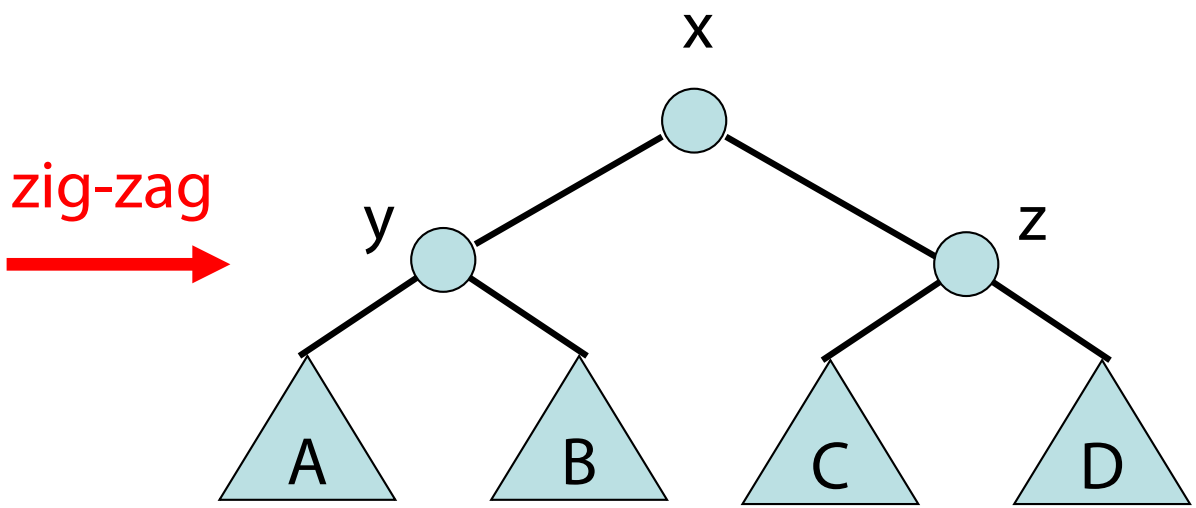
3a. x hat Vater links, Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

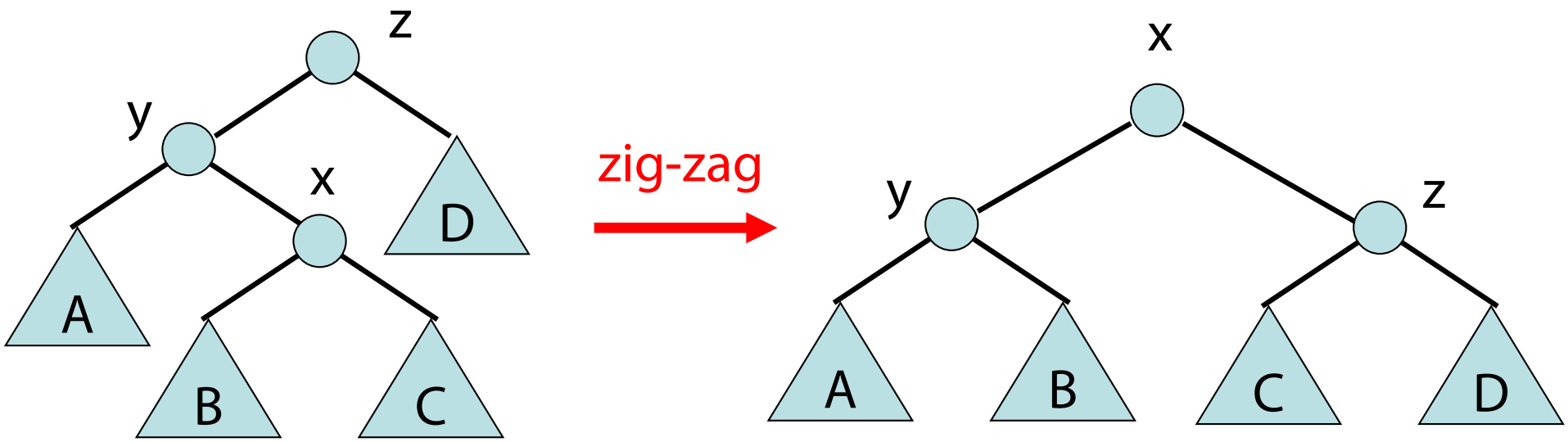
3a. x hat Vater links, Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

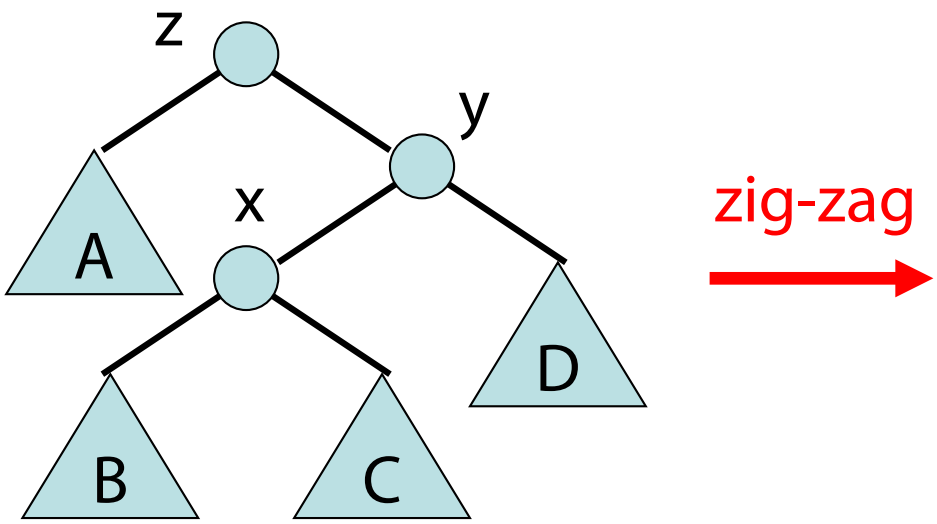
3a. x hat Vater links, Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

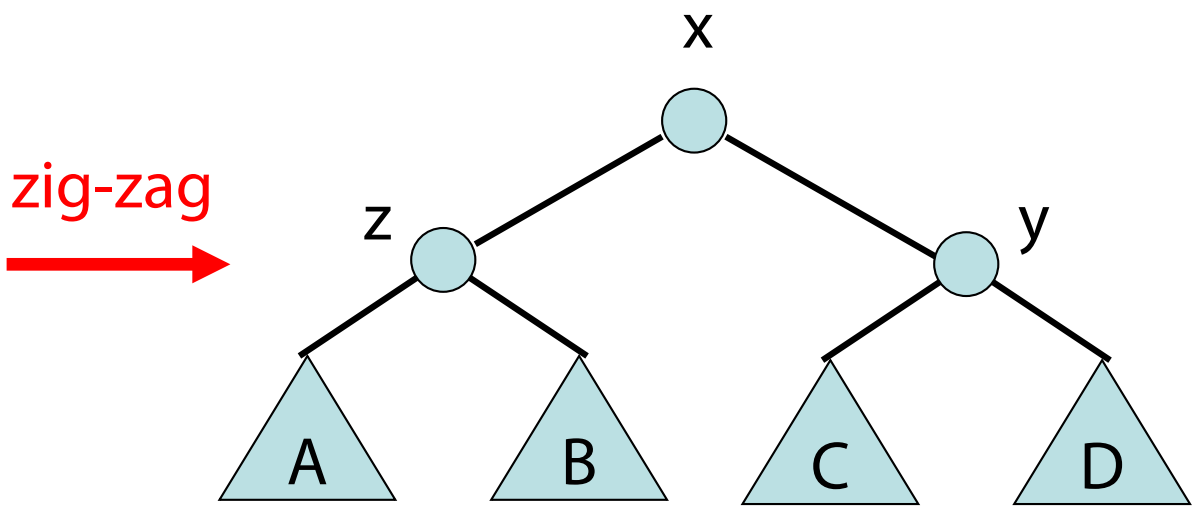
3b. x hat Vater rechts, Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

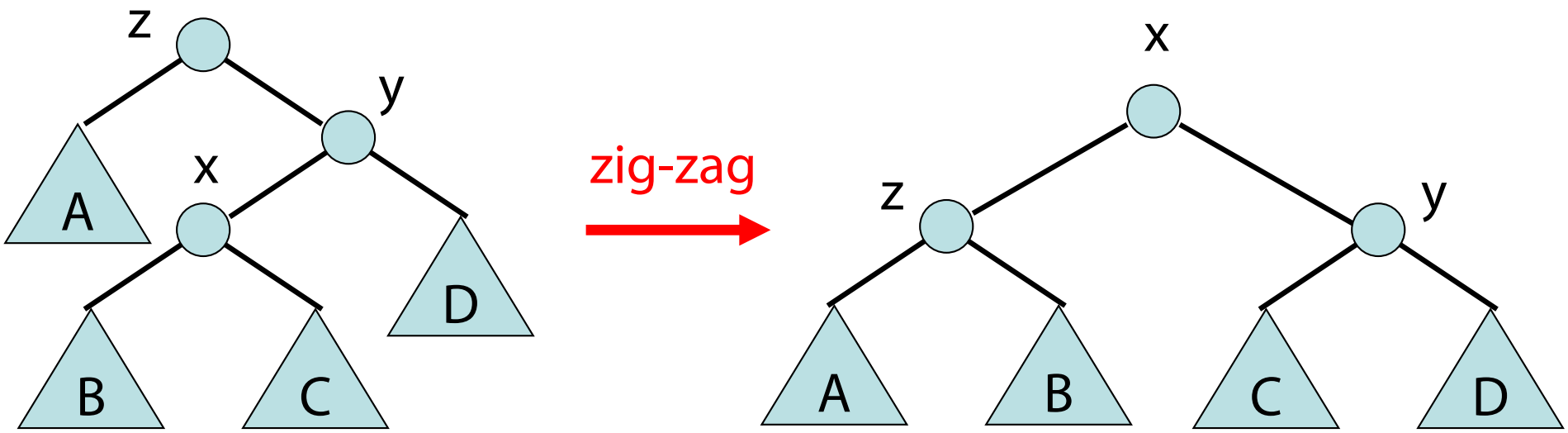
3b. x hat Vater rechts, Großvater links:



Splay-Operation

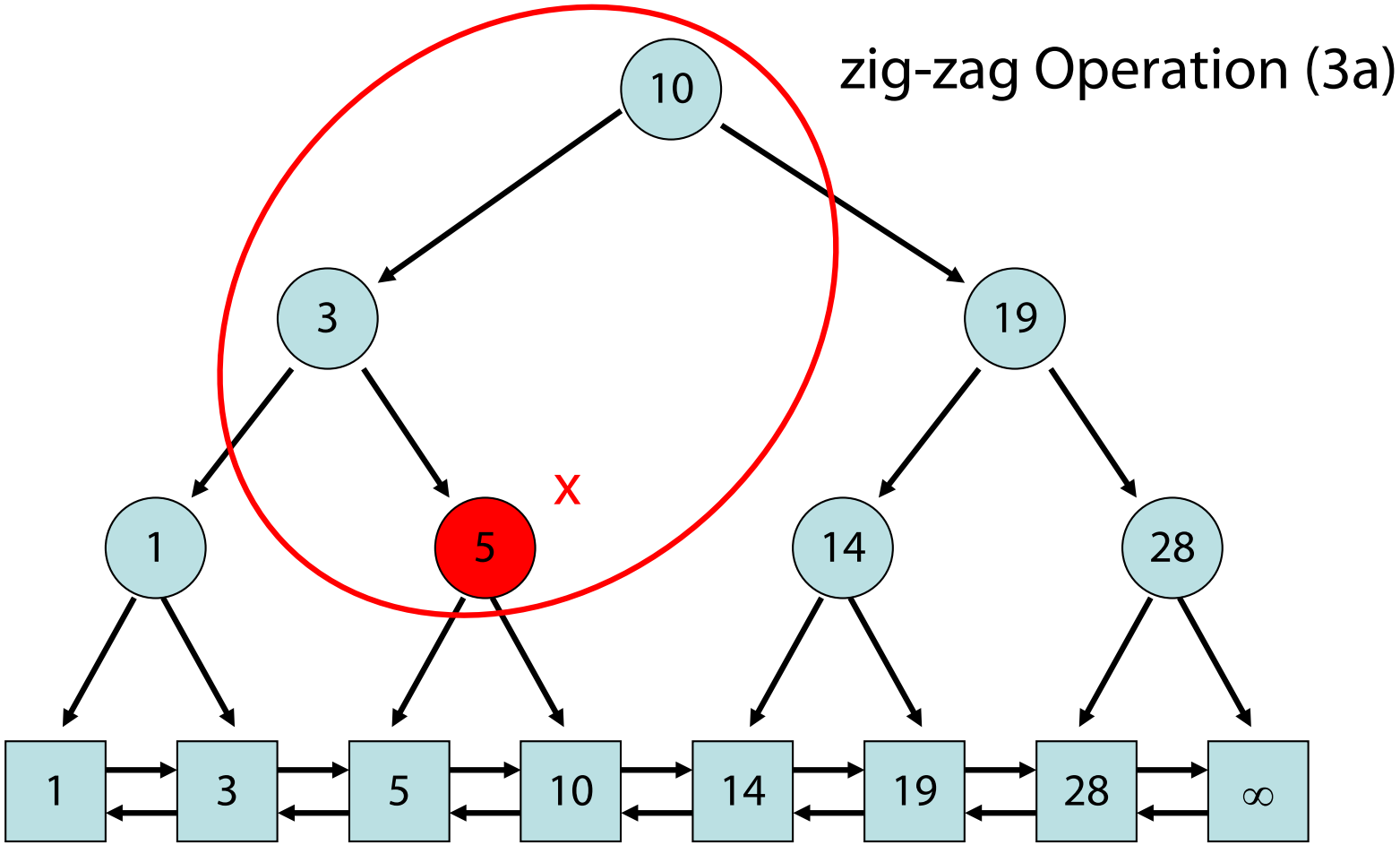
Wir unterscheiden zwischen 3 Fällen.

3b. x hat Vater rechts, Großvater links:

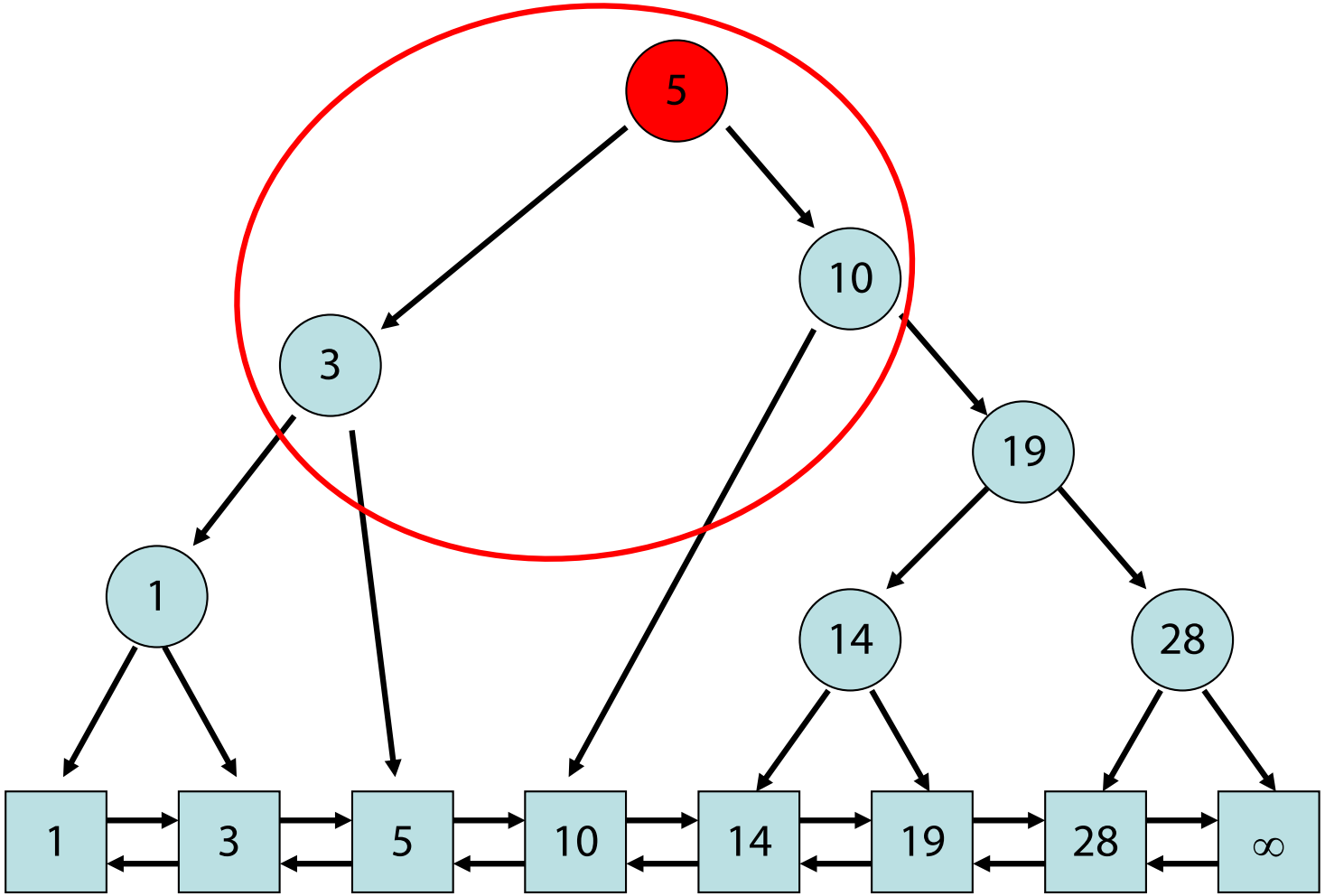


Splay-Operation

Beispiel:



Splay-Operation

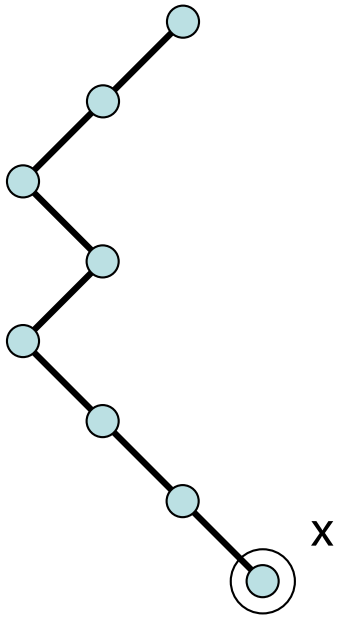


Splay

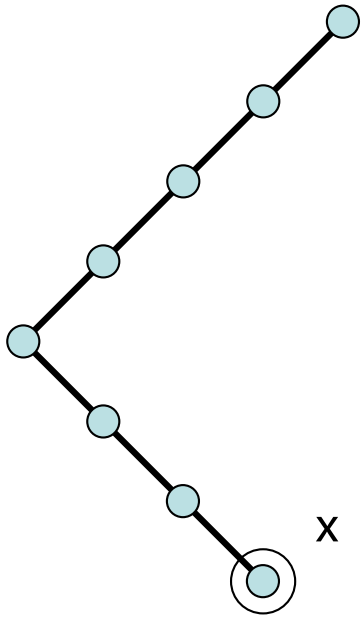
```
procedure splay(x, T)
  while parentExists(x, T) do
    // x wandert hoch
    Wende geeignete Splay-Operation an
```

Splay-Operation: Maximal ein zig

Beispiele:



zig-zig, zig-zag, zig-zag, zig



zig-zig, zig-zag, zig-zig, zig

Splay-Operation

search(k)-Operation: (exakte Suche)

- Laufe von Wurzel startend nach unten,
  bis k im Baumknoten gefunden (Abkürzung zur Liste) oder
  bei Liste angekommen (in diesem Fall Schlüssel nicht vorhanden)
- k in Baum: rufe splay(k) auf

Amortisierte Analyse:

Sei F eine Folge von m Splay-Operationen auf beliebigem Anfangsbaum mit n Elementen (m>n)

$T(F) \leq c + \Sigma_{i=1}^{m} A_{Op_i}(s_{i-1})$

$A_X(s) := \phi(s') - \phi(s) + T_X(s) := \Delta\, \phi(s) + T_X(s)$ für $s \xrightarrow{X} s'$

Splay-Operation

- Gewicht von Knoten x: w(x)
  relative Zugriffshäufigkeit
- Baumgewicht von Baum T mit Wurzel x:
  $tw(x) = \Sigma_{y \in T\_x} w(y)$
- Rang von Knoten x: $r(x) = \log(tw(x))$
- Potential von Baum T: $\phi(T) = \Sigma_{x \in T} r(x)$

Behauptung "amortisierte Splaykosten": Sei T ein Splay-Baum mit Wurzel u und x ein Knoten in T. Die amortisierten Kosten für splay(x, T) sind max. $1+3(r(u)-r(x))$.

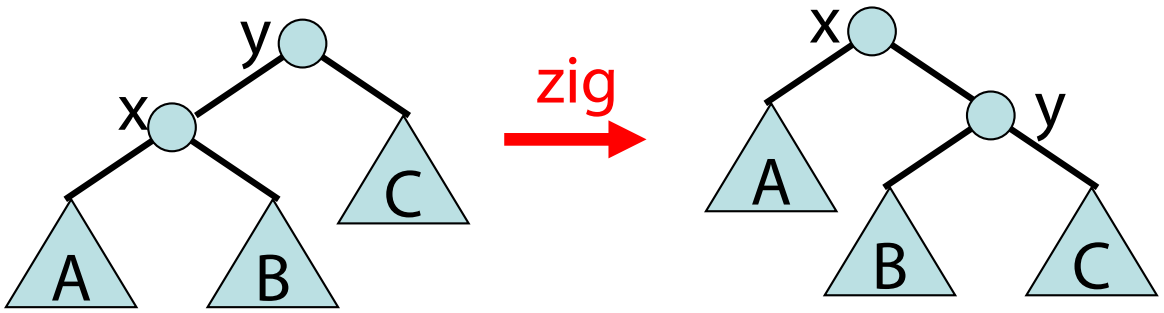
Splay-Operation

Begründung:

Induktion über die Folge der Rotationen.

- r und tw : Rang und Gewicht vor Rotation
- r′ und tw′: Rang und Gewicht nach Rotation

1. Fall:



Laufzeit
(# Rotationen)

Amortisierte Kosten:

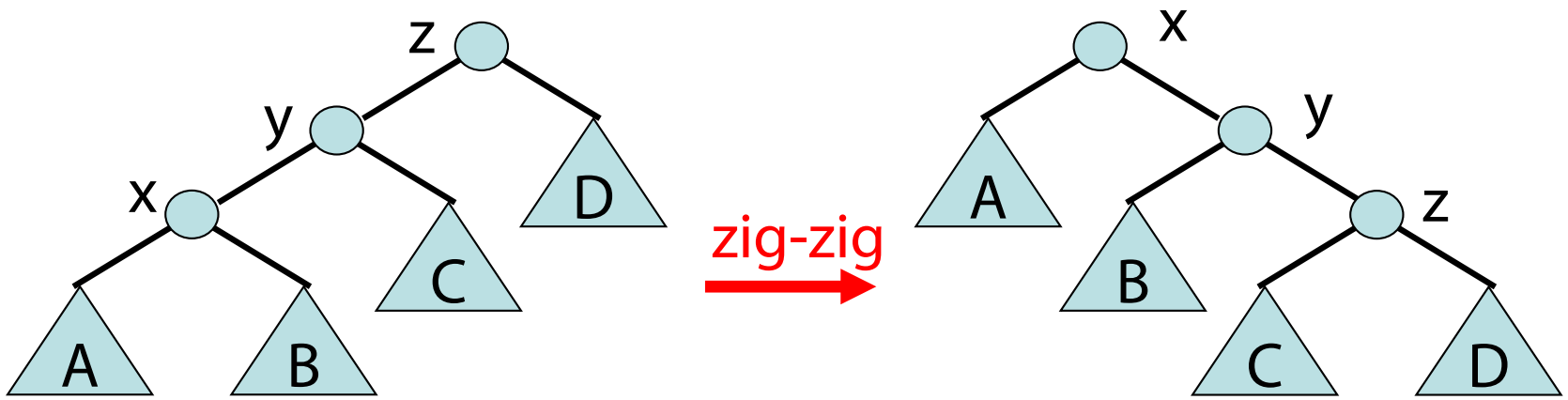
$\leq 1+r'(x)+r'(y)-r(x)-r(y) \leq 1+r'(x)-r(x)$ da $r'(y) \leq r(y)$

$\leq 1+3(r'(x)-r(x))$ da $r'(x) \geq r(x)$

Änderung von ϕ

Splay-Operation

2. Fall:



Amortisierte Kosten:

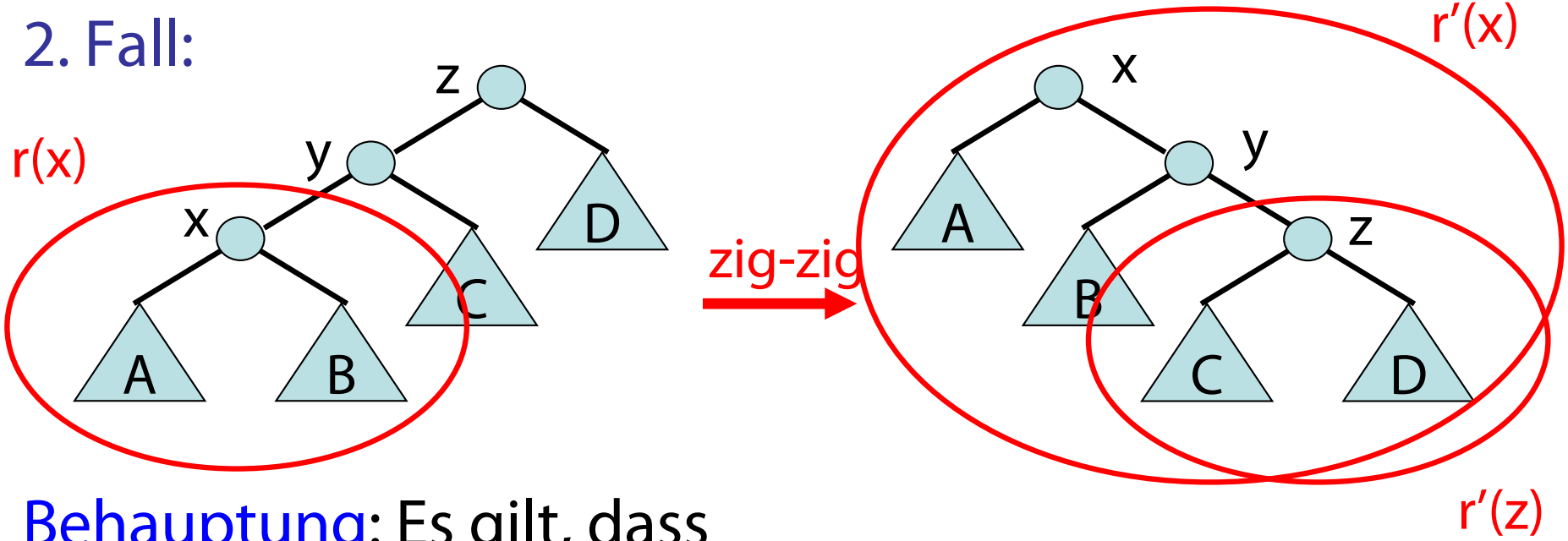
$\leq 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)$

$= 2+r'(y)+r'(z)-r(x)-r(y)$ da $r'(x)=r(z)$

$\leq 2+r'(x)+r'(z)-2r(x)$ da $r'(x) \geq r'(y)$ und $r(y) \geq r(x)$

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

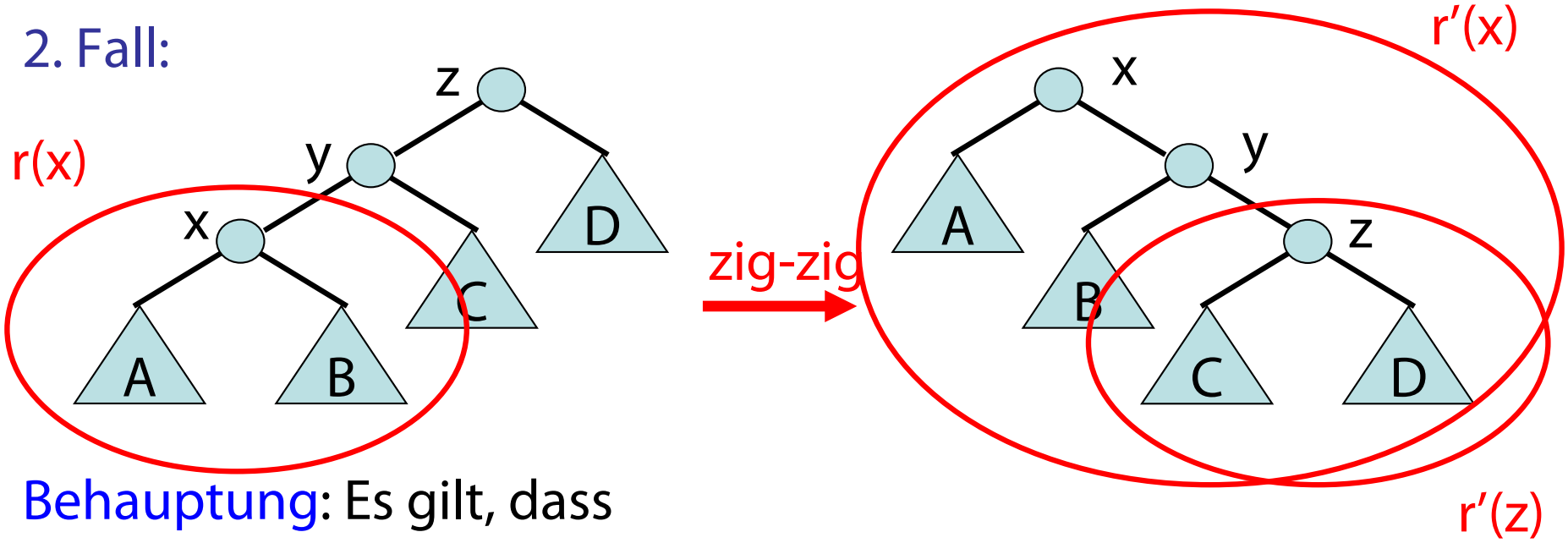
$$2+r'(x)+r'(z)-2r(x) \leq 3(r'(x)-r(x))$$

d.h.

$$r(x)+r'(z) \leq 2(r'(x)-1)$$

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

$$r(x)+r'(z) \leq 2(r'(x)-1)$$

Abschätzung: $r(x) = \log tw(x)$, $r'(z) = \log tw(z)$, $r'(x) \leq \log 1$.

Betrachte die Funktion $f(x,y)=\log x + \log y$.

Wir wollen zeigen: $f(x,y) \leq -2$ für alle $x,y>0$ mit $x+y<1$.

Splay-Operation

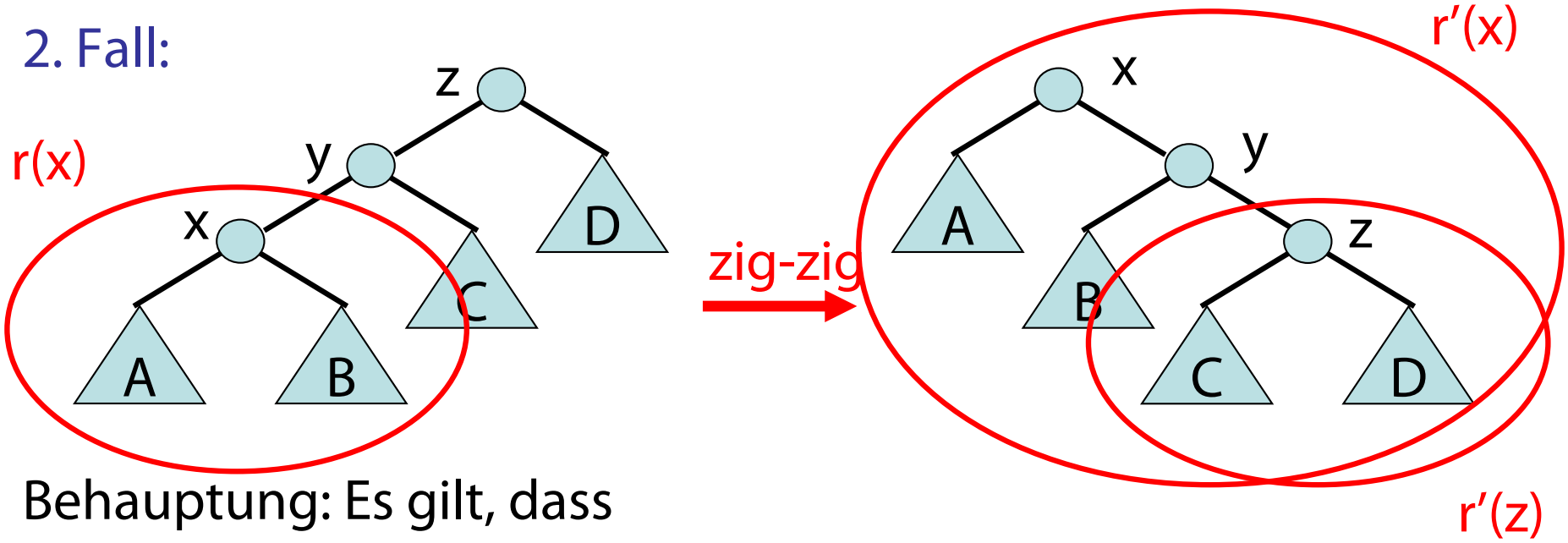
Behauptung: Die Funktion $f(x,y)=\log x + \log y$ hat in dem Bereich $x,y>0$ mit $x+y<=1$ im Punkt $(½,½)$ ihr Maximum.

Begründung:

- Da die Funktion log streng monoton wachsend ist, kann sich das Maximum nur auf dem Geradensegment $x+y=1$, $x,y>0$, befinden.
- Neues Maximierungsproblem: betrachte $g(x) = \log x + \log (1-x)$
- Einzige Nullstelle von $g'(x) = 1/x - 1/(1-x)$ ist $x=1/2$.
- Für $g''(x)= -(1/x^2 + 1/(1-x)^2))$ gilt $g''(1/2) < 0$.
- Also hat Funktion f im Punkt $(½,½)$ ihr Maximum.

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

$$r(x)+r'(z) \leq 2(r'(x)-1)$$

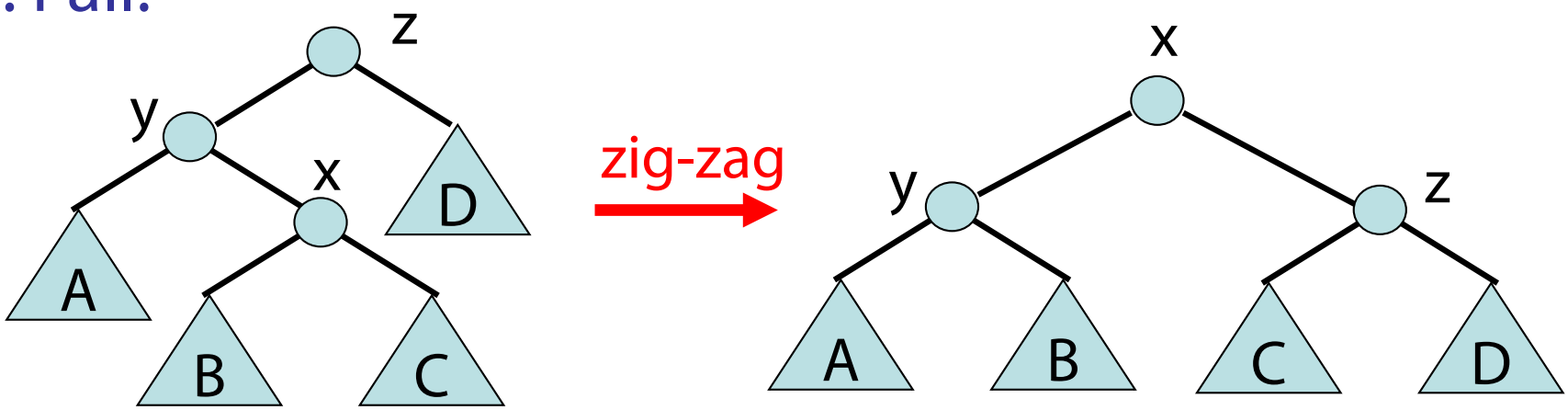
Abschätzung: $r(x) = \log tw(x)$, $r'(z) = \log tw(z)$, $r'(x) \leq \log 1$.

Da $f(x,y)=\log x + \log y < -2$ für alle $x,y>0$ mit $x+y<1$.

Ist die Behauptung also korrekt.

Splay-Operation

3. Fall:



Amortisierte Kosten:

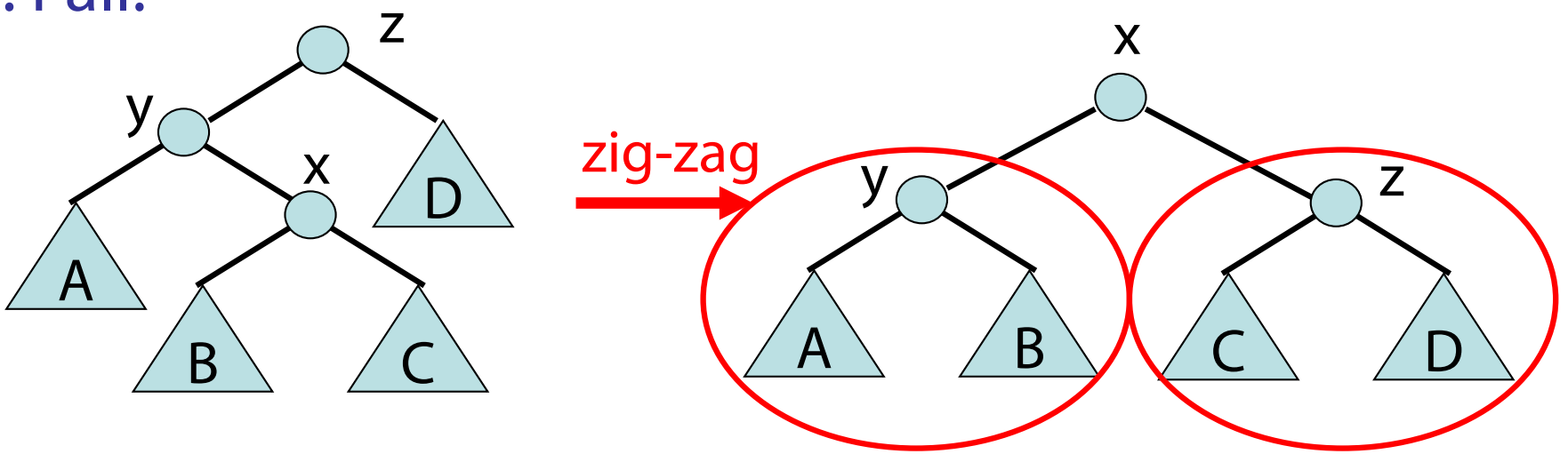
$\leq 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)$

$\leq 2+r'(y)+r'(z)-2r(x)$ da $r'(x)=r(z)$ und $r(x)\leq r(y)$

$\leq 2(r'(x)-r(x))$ und…

Splay-Operation

3. Fall:



…es gilt:

$$2+r'(y)+r'(z)-2r(x) \leq 2(r'(x)-r(x))$$

gdw $2r'(x)-r'(y)-r'(z) \geq 2$

gdw $r'(y)+r'(z) \leq 2(r'(x)-1)$ analog zu Fall 2

Splay-Operation

Beweis von Behauptung “amortisierte Splaykosten”: (Fortsetzung)

Induktion über die Folge der Rotationen.

- r und tw : Rang und Gewicht vor Rotation
- r’ und tw’: Rang und Gewicht nach Rotation
- Für jede Rotation ergeben sich amortisierte Kosten von max. $1+3(r'(x)-r(x))$ (Fall 1) bzw. $3(r'(x)-r(x))$ (Fälle 2 und 3) (NB: max 1 zig)
- Aufsummierung der Kosten pro Zugriff ergibt max. $1 + \Sigma_{Rot.} 3(r'(x)-r(x)) = 1+3(r(u)-r(x))$

Splay-Operation

- Baumgewicht von Baum T mit Wurzel x:
  $tw(x)= \Sigma_{y \in T(x)} w(y)$
- Rang von Knoten x: $r(x) = \log(tw(x))$
- Potential von Baum T: $\phi(T) = \Sigma_{x \in T} r(x)$

"amortisierte Splaykosten": Sei T ein Splay-Baum mit Wurzel u und x ein Knoten in T. Die amortisierten Kosten für splay(x,T) sind max. $1+3(r(u)-r(x)) = 1+3\cdot\log (tw(u)/tw(x))$.

Korollar: Sei $W=\Sigma_x w(x)$ und w_i das Gewicht von k_i in i-tem search. Für m search-Operationen der Folge F sind die amortisierten Kosten $A(F) = m+3\Sigma_{i=1}^{m} \log (W/w_i)$.

Splay-Baum: Balance-Theorem

Balance Theorem: " Splay-Bäume arbeiten wie ausgeglichene Bäume"
Die Laufzeit für m search Operationen in einem n-elementigen Splay-Baum T ist in

$$O(m \cdot \log n) \quad (\text{NB: } m > n)$$

Begründung:

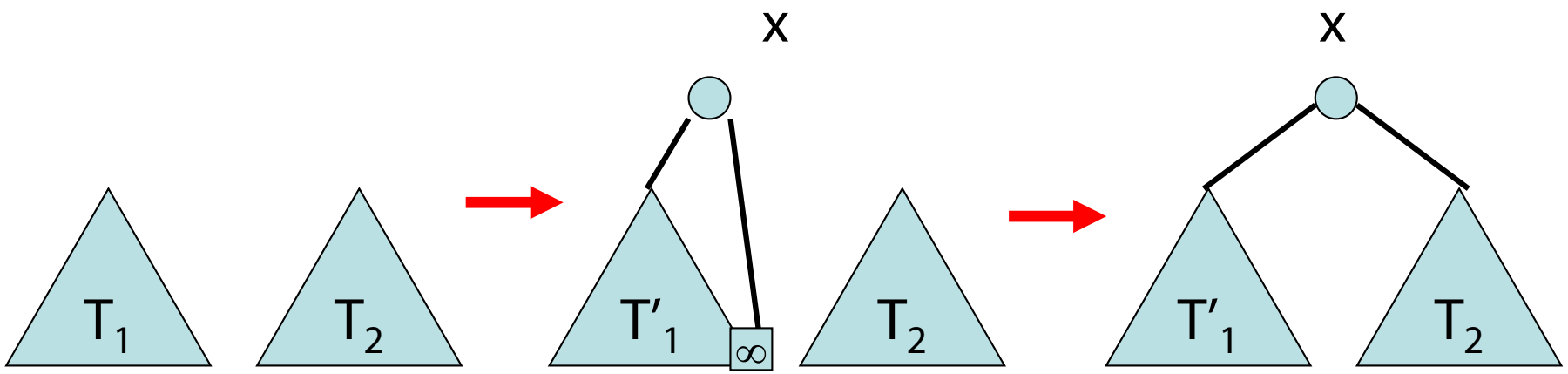
- Sei $w(x) = 1/n$ für alle Schlüssel x in T (gleiche relative Zugriffshäufigkeit).
- Dann ist $W=1$ und $r(x) \leq \log W = \log 1$ für alle x in T.
- Erinnerung: für eine Operationsfolge F ist die Laufzeit $T(F) \leq A(F) + \phi(s_0)$ für amortisierte Kosten A und Anfangszustand s_0
- $\phi(s_0) = \Sigma_{x \in T} r_0(x) \leq n \log 1 = 0$
- Aus dem Korollar von oben ergibt sich das Theorem:
- $T(F) \leq m+3\Sigma_{i=1}^{m} \log (W/w_i) = m + 3 \Sigma_{i=1}^{m} \log(1/ (1/n))$
  $= m + 3 \Sigma_{i=1}^{m} \log n = m + 3m \log n \in O(m \log n)$

Deutung: Splay-Bäume arbeiten so effektiv wie ausgeglichene Bäume bei Search-Sequenzen länger als n

Splay-Baum Operationen

Annahme: zwei Splay-Bäume T_1 und T_2 mit key(x)<key(y)
für alle $x \in T_1$ und $y \in T_2$.

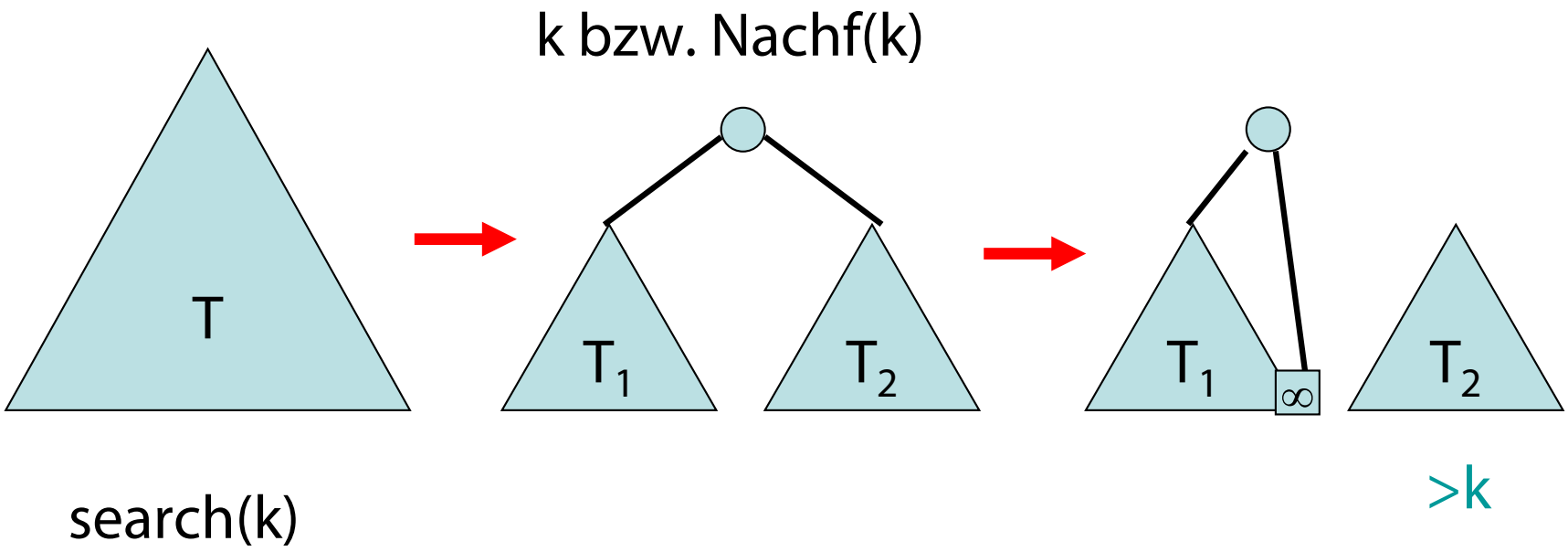
merge(T_1,T_2):



search(x), $x<\infty$ max. in T_1

Splay-Baum Operationen

split(k,T):



Splay-Baum Operationen

insert(e):

- insert wie im Binärbaum
- splay-Operation, um key(e) in Wurzel zu verschieben

delete(k):

- führe search(k) aus (bringt k in die Wurzel)
- entferne Wurzel und führe merge(T_1,T_2) der beiden Teilbäume durch

Offene Fragen [Wikipedia 17.5.2018]

Dynamic optimality conjecture [edit]

Main article: Optimal binary search tree

List of unsolved problems in computer science

Do splay trees perform as well as any other binary search tree algorithm?

In addition to the proven performance guarantees for splay trees there is an unproven conjecture of great interest from the original Sleator and Tarjan paper. This conjecture is known as the *dynamic optimality conjecture* and it basically claims that splay trees perform as well as any other binary search tree algorithm up to a constant factor.

Dynamic Optimality Conjecture:[1] Let A be any binary search tree algorithm that accesses an element x by traversing the path from the root to x at a cost of $d(x) + 1$, and that between accesses can make any rotations in the tree at a cost of 1 per rotation. Let $A(S)$ be the cost for A to perform the sequence S of accesses. Then the cost for a splay tree to perform the same accesses is $O[n + A(S)]$.

There are several corollaries of the dynamic optimality conjecture that remain unproven:

Traversal Conjecture:[1] Let T_1 and T_2 be two splay trees containing the same elements. Let S be the sequence obtained by visiting the elements in T_2 in preorder (i.e., depth first search order). The total cost of performing the sequence S of accesses on T_1 is $O(n)$.

Deque Conjecture:[8][10][11] Let S be a sequence of m double-ended queue operations (push, pop, inject, eject). Then the cost of performing S on a splay tree is $O(m + n)$.

Split Conjecture:[5] Let S be any permutation of the elements of the splay tree. Then the cost of deleting the elements in the order S is $O(n)$.

Rot-Schwarz-Baum

Rot-Schwarz-Bäume sind binäre Suchbäume mit roten und schwarzen Knoten, so dass gilt:

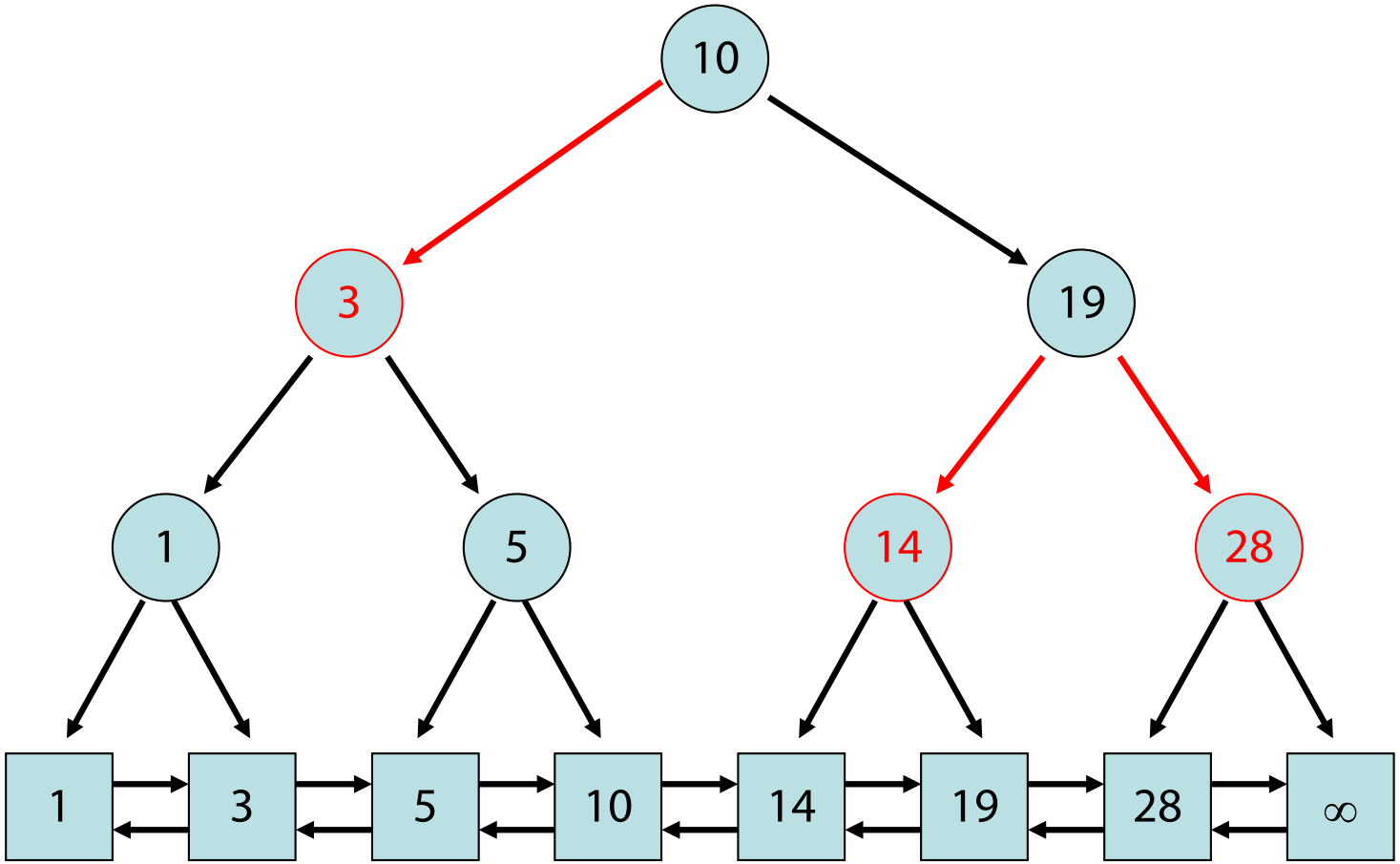
- Wurzeleigenschaft: Die Wurzel ist schwarz.
- Externe Eigenschaft: Jeder Listenknoten ist schwarz.
- Interne Eigenschaft: Die Kinder eines roten Knotens sind schwarz.
- Tiefeneigenschaft: Alle Listenknoten haben dieselbe "Schwarztiefe"

"Schwarztiefe" eines Knotens: Anzahl der schwarzen Baumknoten (außer der Wurzel) auf dem Pfad von der Wurzel zu diesem Knoten.

Leonidas J. Guibas and Robert Sedgewick, A Dichromatic Framework for Balanced Trees, In: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, S. 8–21, **1978**

Rot-Schwarz-Baum

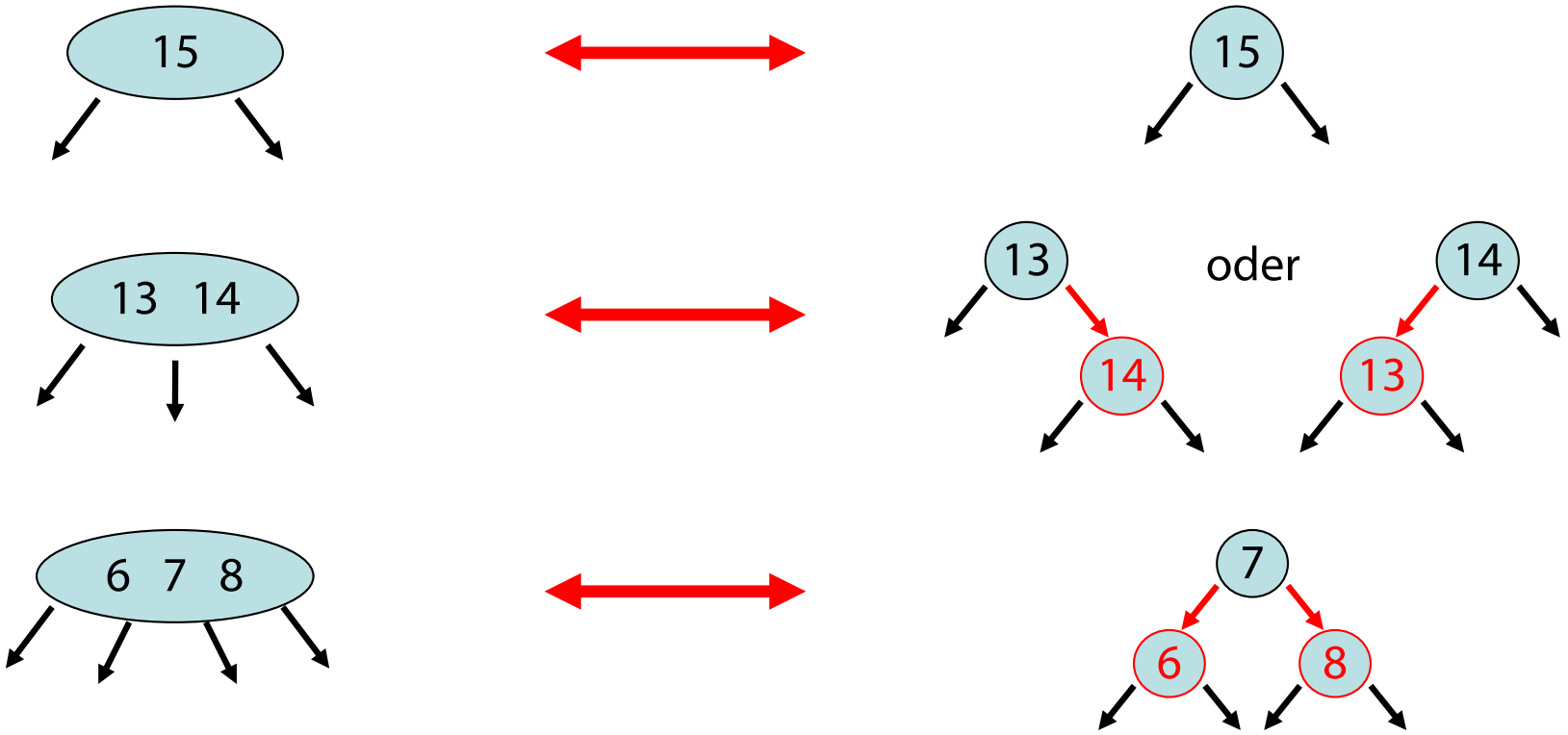
Beispiel:



Rot-Schwarz-Baum

Andere Deutung von Rot-schwarz-Mustern:

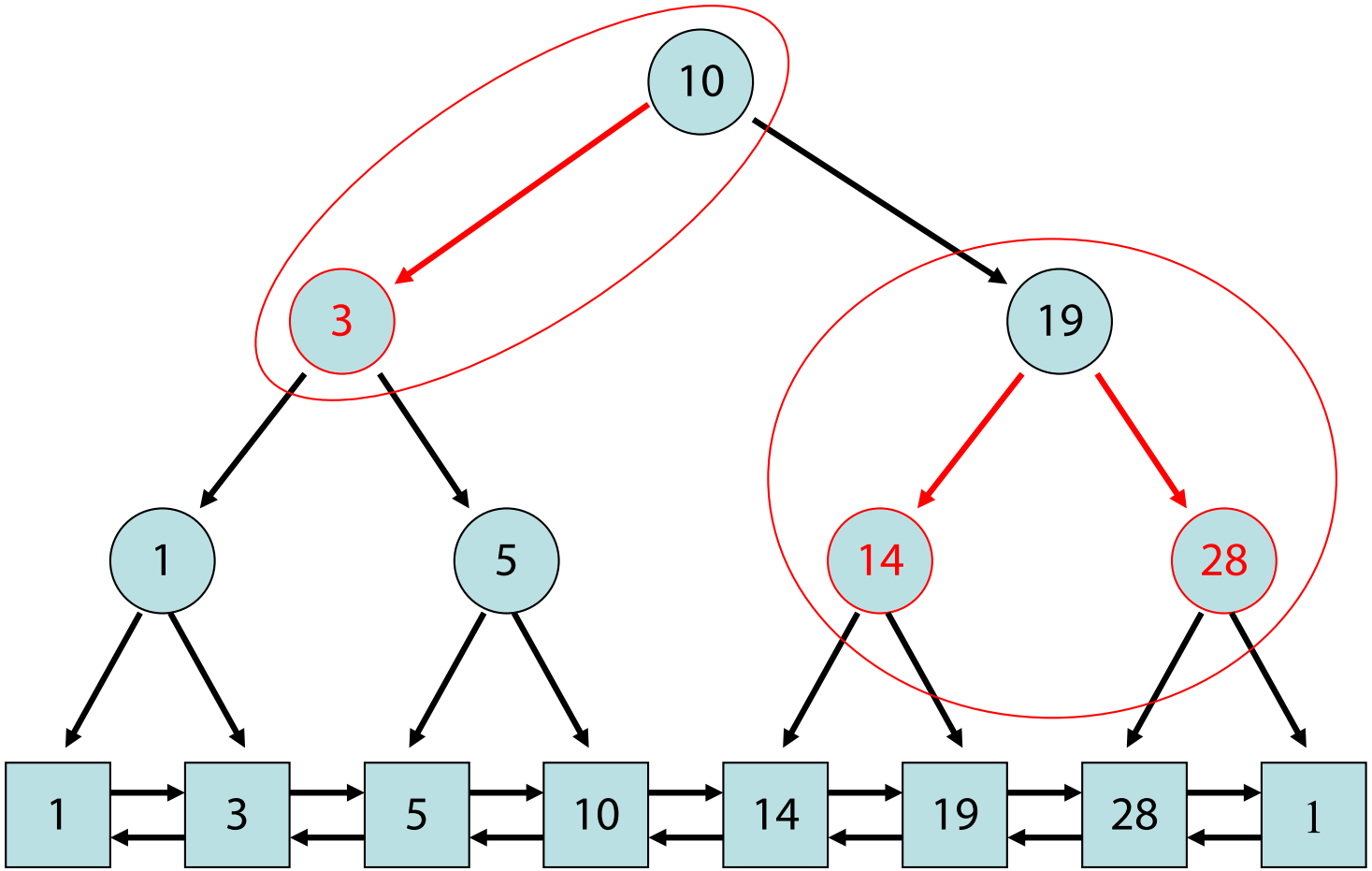
B-Baum (Baum mit “Separatoren” und min 2 max 4 Nachfolger)



Rudolf **B**ayer, Symmetric binary **B**-Trees: Data structure and maintenance algorithms. Acta Informatica 1 (4), S. 290–306, **1972**

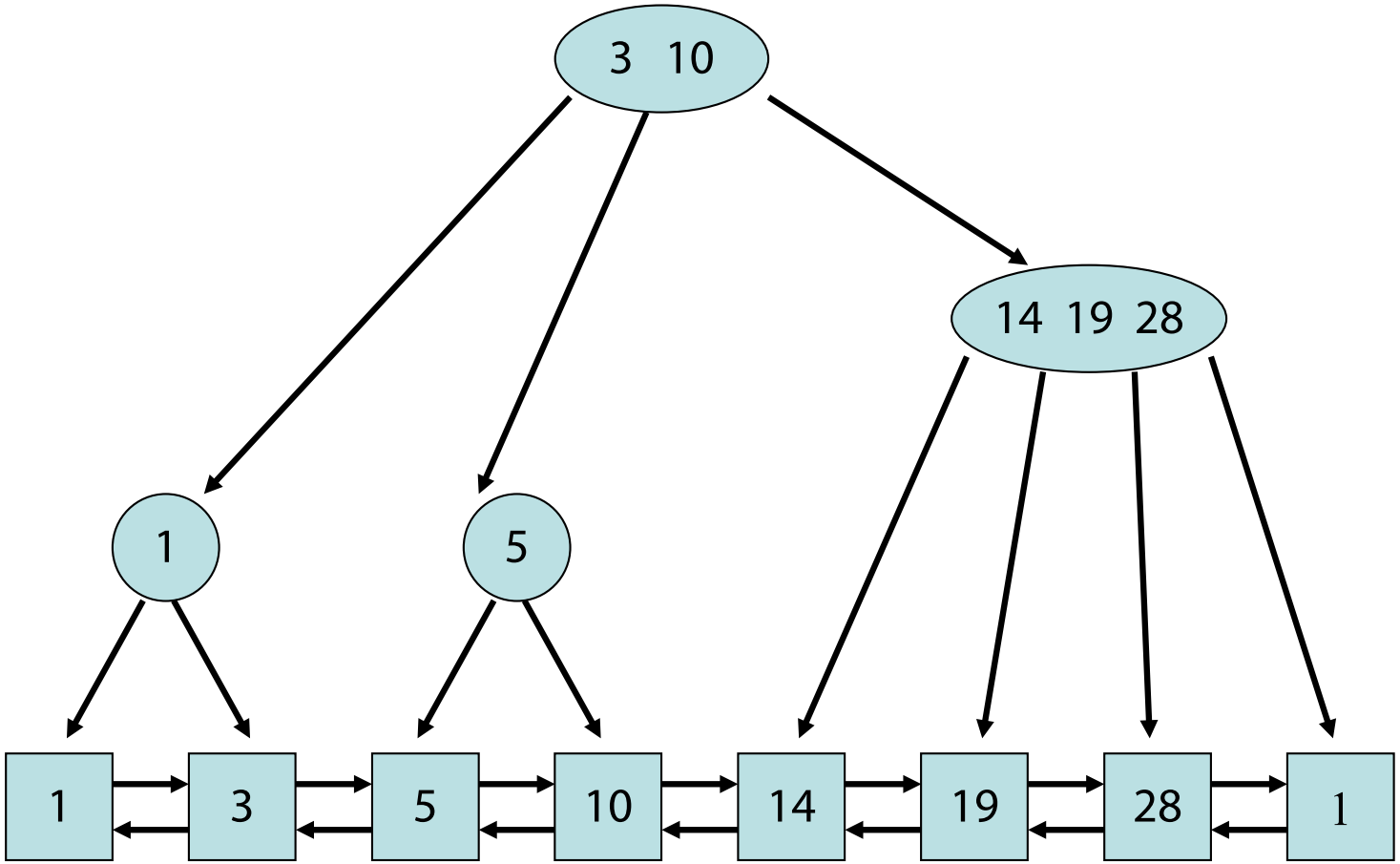
Rot-Schwarz-Baum

R-S-Baum:



Rot-Schwarz-Baum

B-Baum:



Rot-Schwarz-Baum

Behauptung: Die Tiefe t eines Rot-Schwarz-Baums T mit n Elementen ist $O(\log n)$.

Beweis:

Wir zeigen: $\log(n+1) \leq t \leq 2\log(n+1)$ für die Tiefe t des Rot-Schwarz-Baums.

- d: Schwarztiefe der Listenknoten
- T': B-Baum zu T
- T' hat Tiefe exakt d überall und $d \leq \log(n+1)$
- Aufgrund der internen Eigenschaft (Kinder eines roten Knotens sind schwarz) gilt: $t \leq 2d$
- Außerdem ist $t \geq \log(n+1)$, da Rot-Schwarz-Baum ein Binärbaum ist.

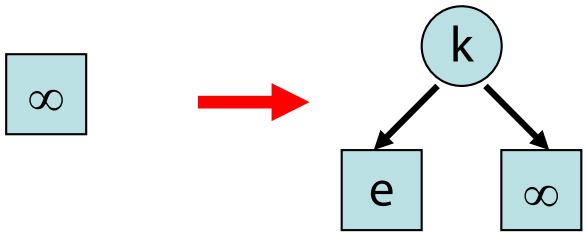
Rot-Schwarz-Baum

search(k): wie im binären Suchbaum

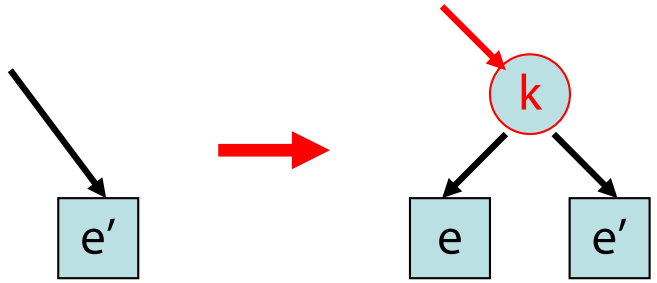
insert(e):

- Führe search(k) mit k=key(e) aus
- Füge e vor Nachfolger e' in Liste ein

Fall 1: Baum leer
(nur Markerelement ∞ enthalten)



Fall 2: Baum nicht leer



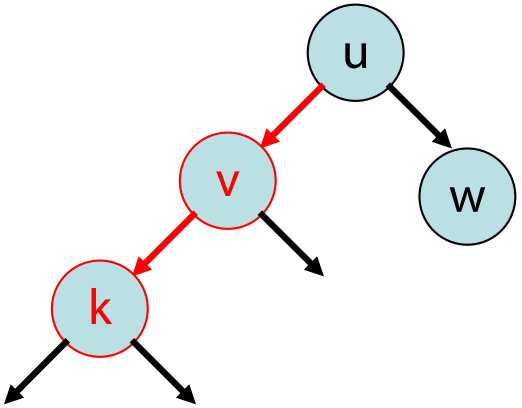
Rot-Schwarz-Baum

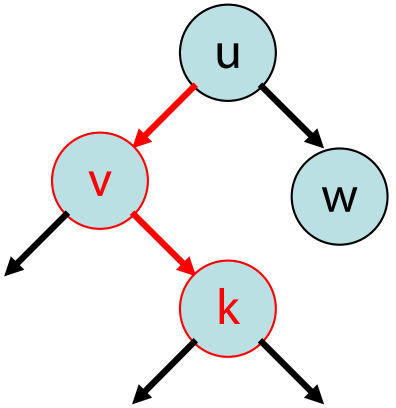
insert(e):

- Führe search(k) mit k=key(e) aus
- Füge e vor Nachfolger e' in Liste ein
  (bewahrt alles bis auf evtl. interne Eigenschaft)
- Interne Eigenschaft verletzt (Fall 2 vorher): 2 Fälle
  - Fall 1: Vater von k in T hat **schwarzen Bruder**
    (Restrukturierung, aber beendet Reparatur)
  - Fall 2: Vater von k in T hat **roten Bruder**
    (setzt Reparatur nach oben fort, aber keine Restrukturierung)

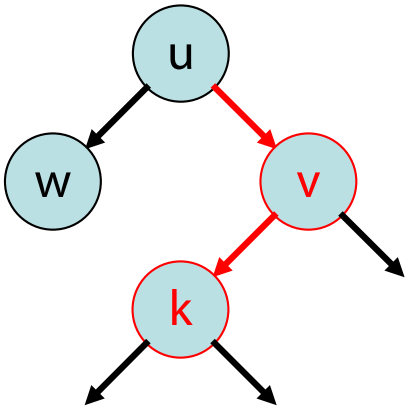
Rot-Schwarz-Baum

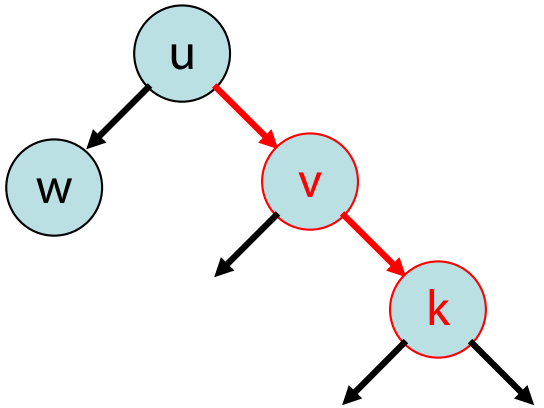
Fall 1: Vater v von k in T hat schwarzen Bruder w





Alternativen
für Fall 1

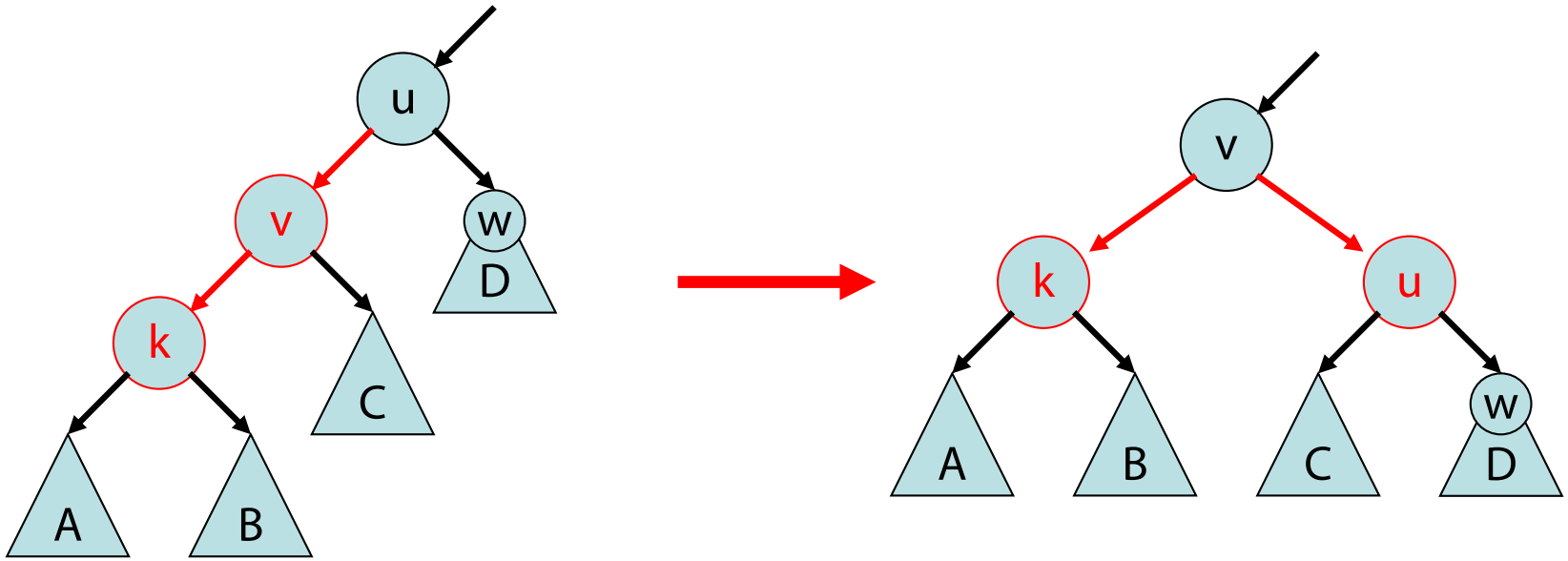




Rot-Schwarz-Baum

Fall 1: Vater v von k in T hat schwarzen Bruder w

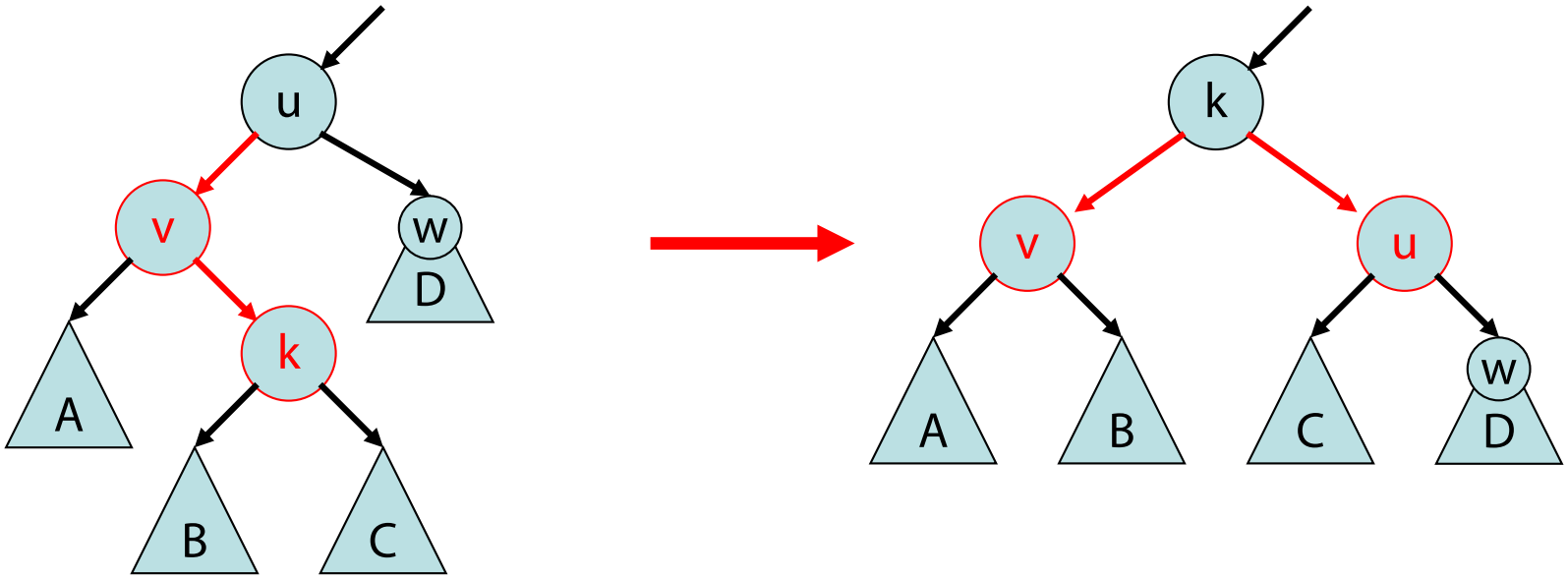
Lösung:



Rot-Schwarz-Baum

Fall 1: Vater v von k in T hat schwarzen Bruder w

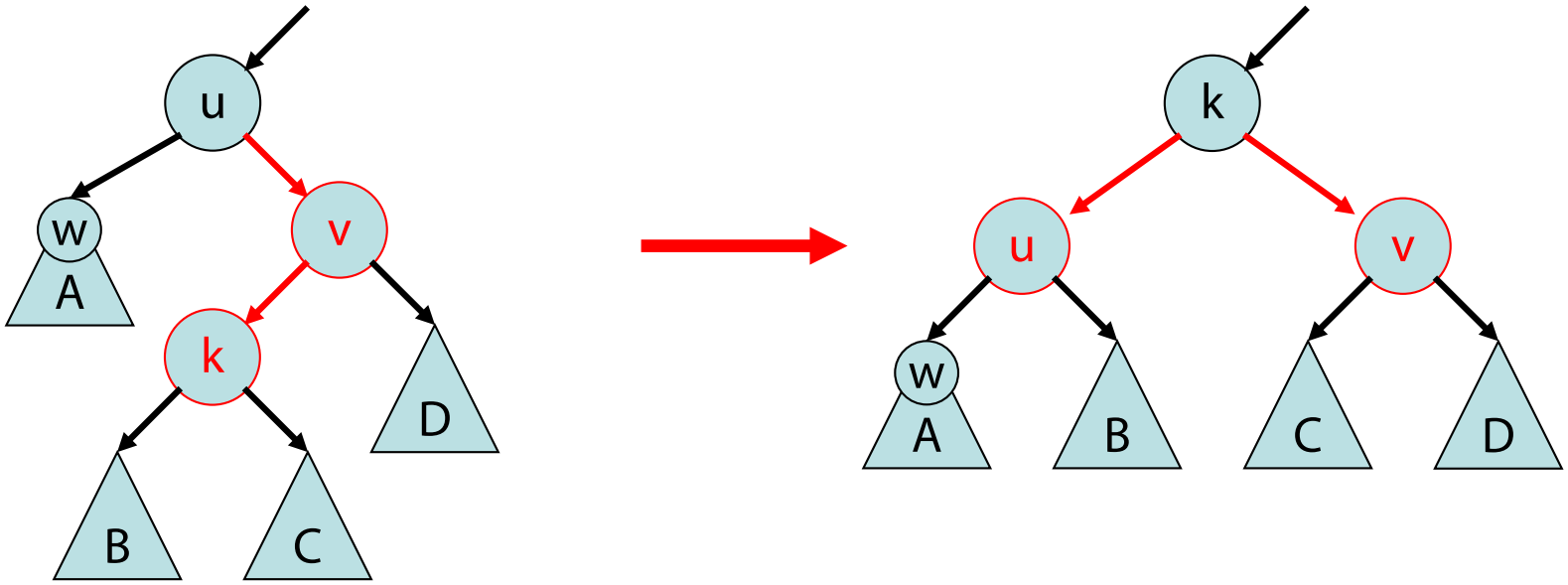
Lösung:



Rot-Schwarz-Baum

Fall 1: Vater v von k in T hat schwarzen Bruder w

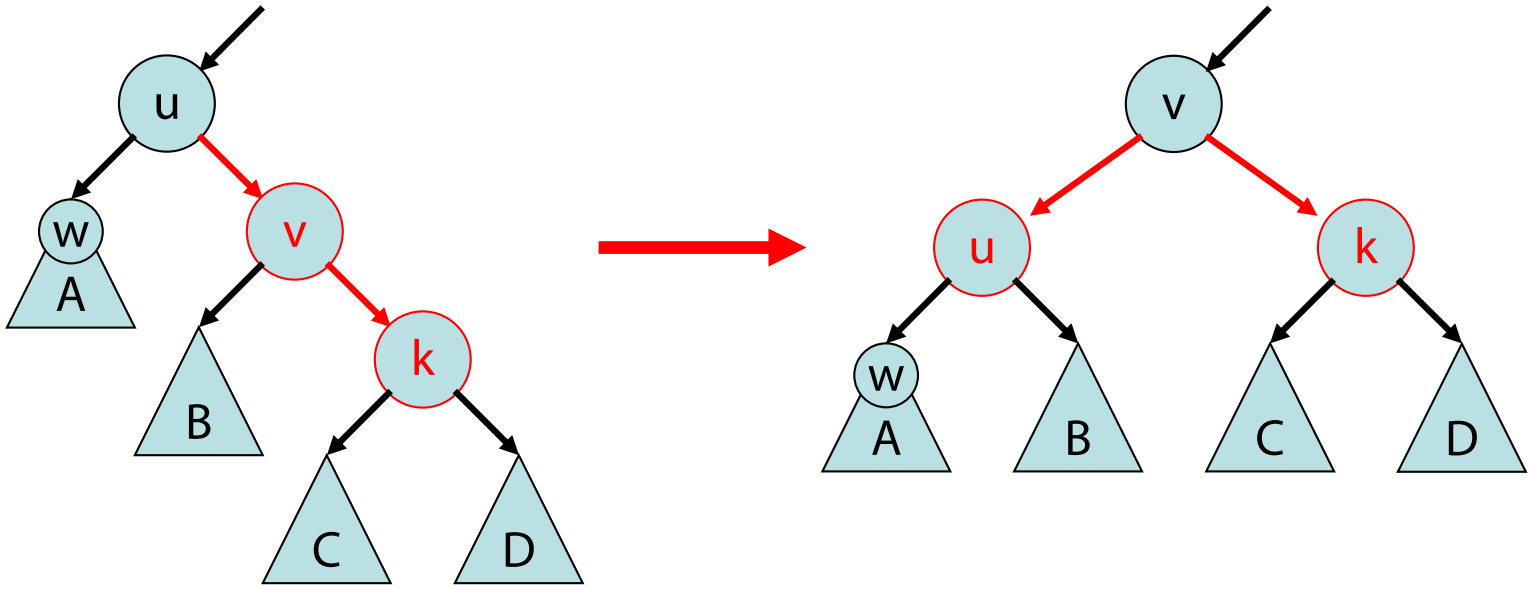
Lösung:



Rot-Schwarz-Baum

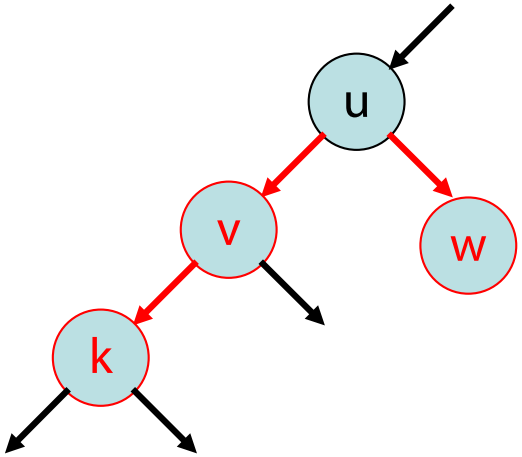
Fall 1: Vater v von k in T hat schwarzen Bruder w

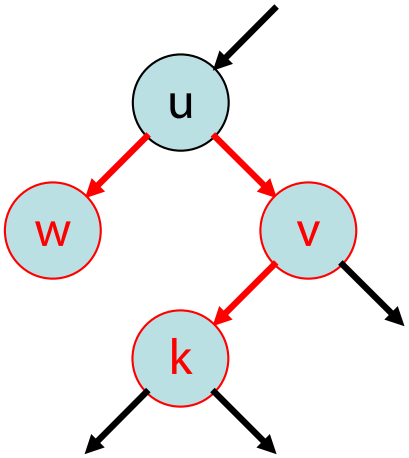
Lösung:



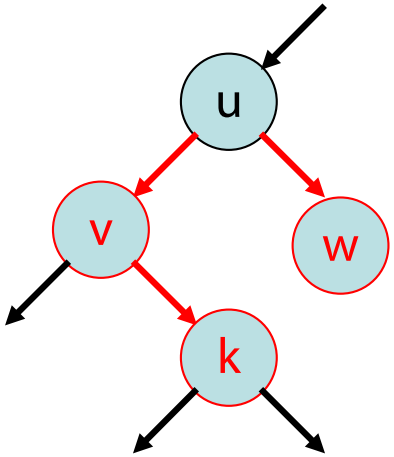
Rot-Schwarz-Baum

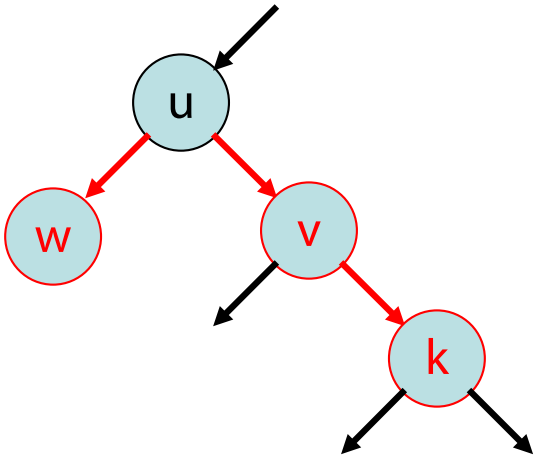
Fall 2: Vater v von k in T hat roten Bruder w





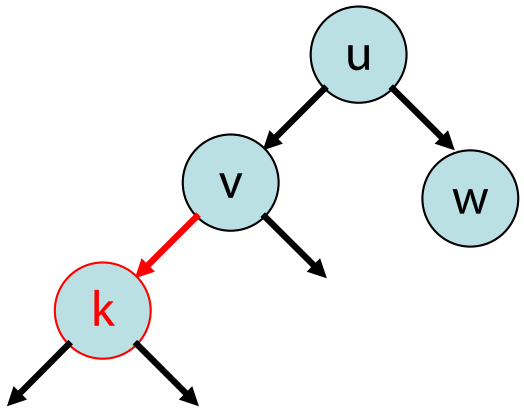
Alternativen
für Fall 2



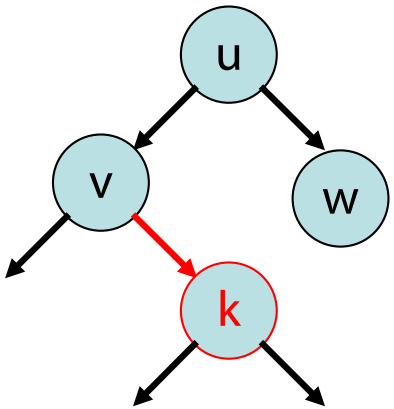


Rot-Schwarz-Baum

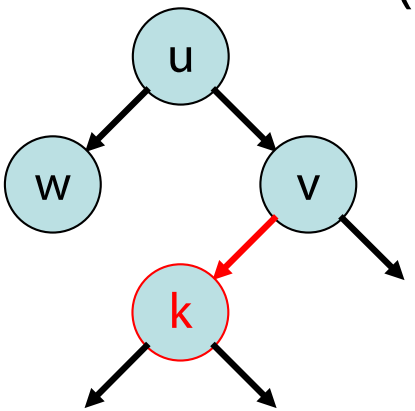
Fall 2: Vater v von k in T hat roten Bruder w

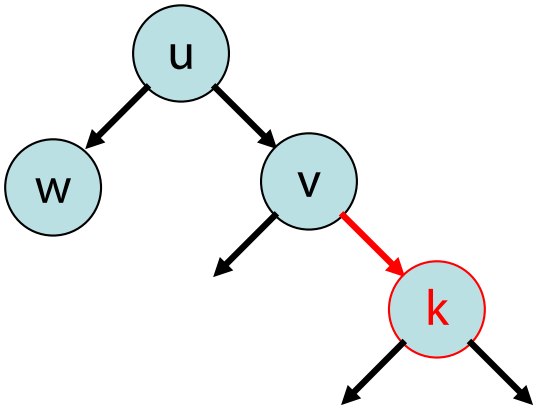


Schwarztiefe+1



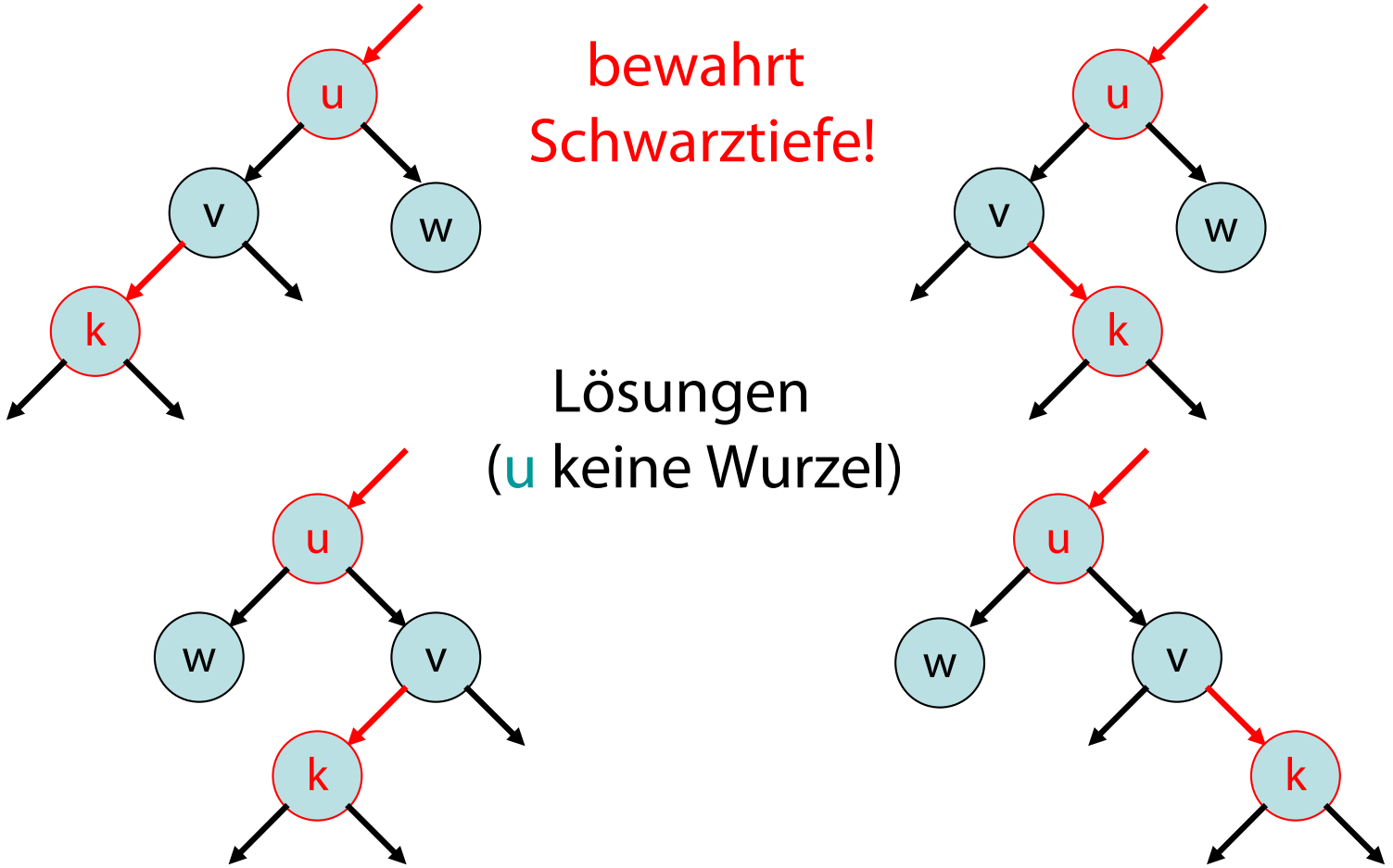
Lösungen
(u ist Wurzel)





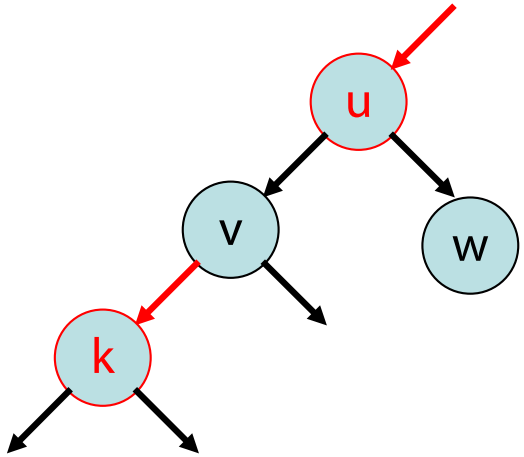
Rot-Schwarz-Baum

Fall 2: Vater v von k in T hat roten Bruder w

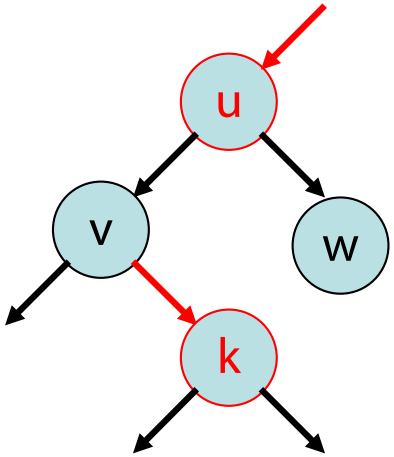


Rot-Schwarz-Baum

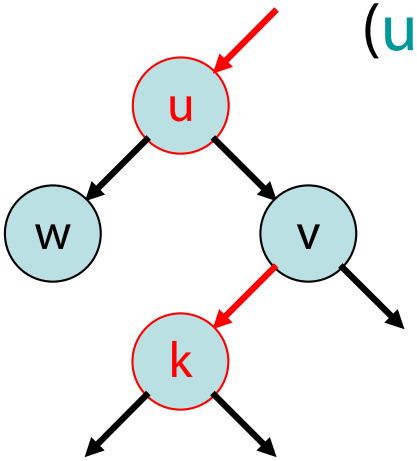
Fall 2: Vater v von k in T hat roten Bruder w

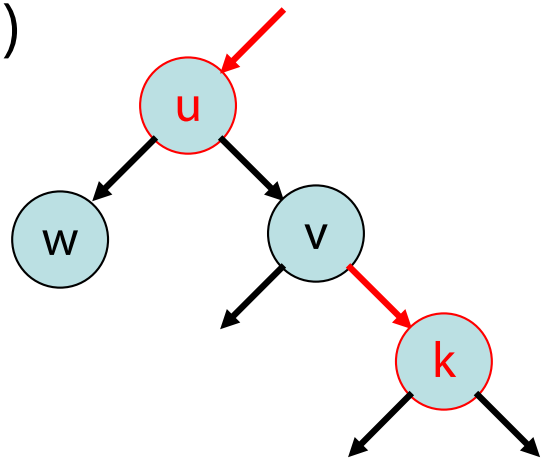


weiter mit u
wie mit k



Lösungen
(u keine Wurzel)

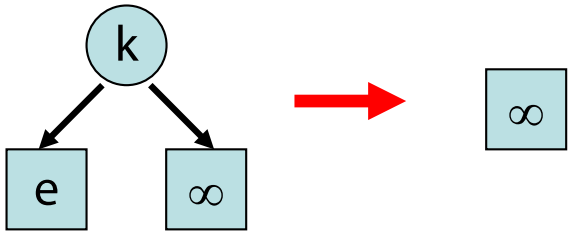




Rot-Schwarz-Baum

delete(k):

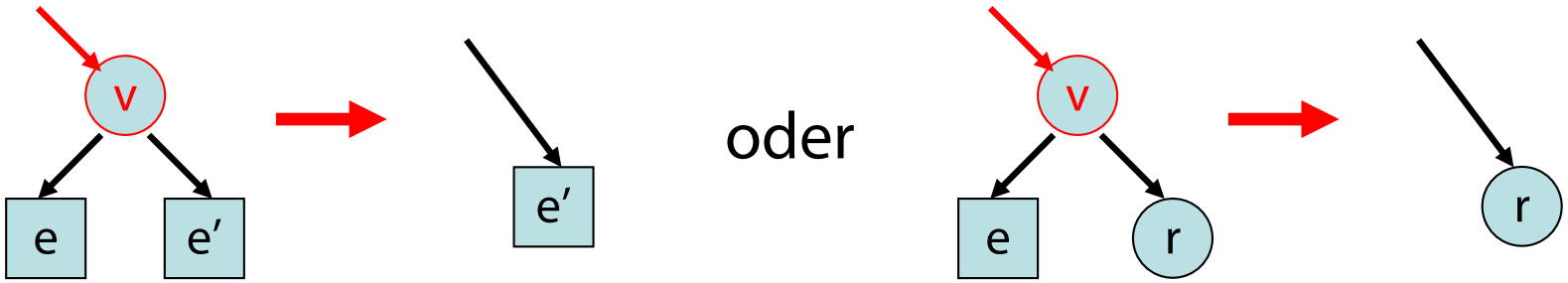
- Führe search(k) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 1: Baum ist dann leer



Rot-Schwarz-Baum

delete(k):

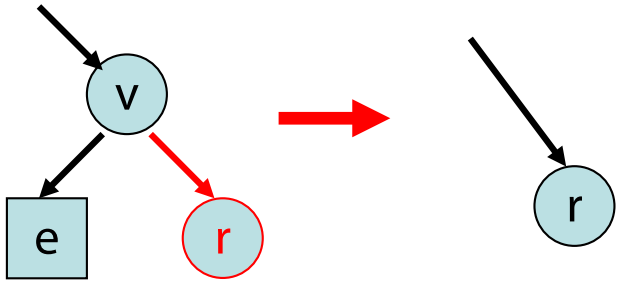
- Führe search(k) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 2: Vater v von e ist rot (d.h. Bruder schwarz)



Rot-Schwarz-Baum

delete(k):

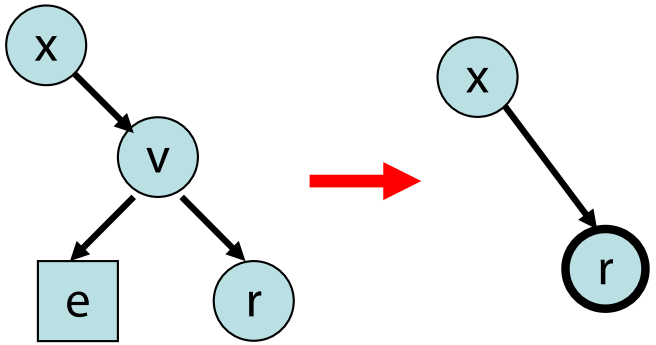
- Führe search(k) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 3: Vater v von e ist schwarz und Bruder rot



Rot-Schwarz-Baum

delete(k):

- Führe search(k) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 4: Vater v von e und Bruder r sind schwarz



Tiefenregel verletzt!
r heißt dann doppelt schwarz

Rot-Schwarz-Baum

delete(k):

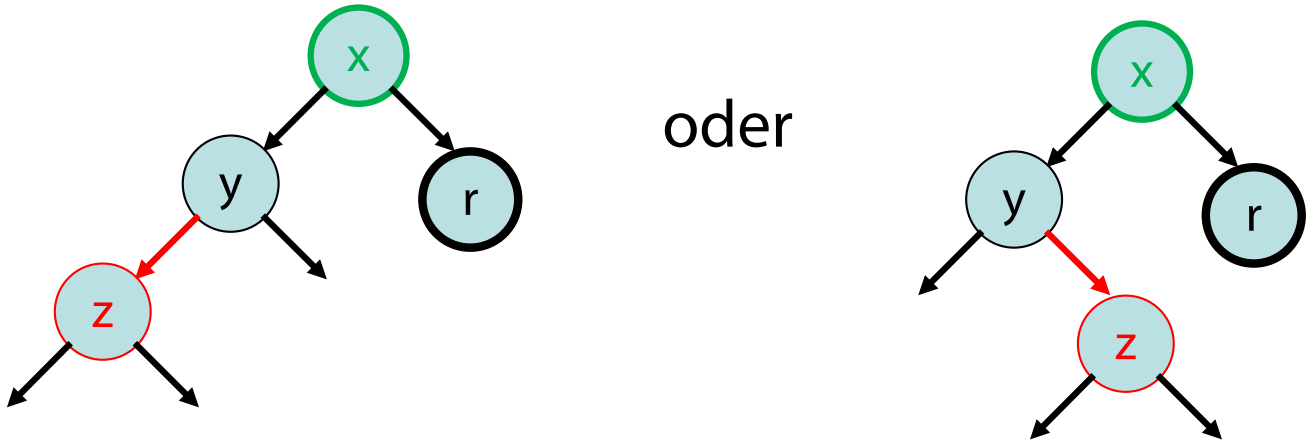
- Führe search(k) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Falls Vater v von e und Bruder r sind schwarz, dann 3 weitere Fälle:
  - Fall 1: Bruder y von r ist schwarz und hat rotes Kind
  - Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz (evtl. weiter, aber keine Restrukt.)
  - Fall 3: Bruder y von r ist rot

Rot-Schwarz-Baum

Fall 1: Bruder y von r ist schwarz, hat rotes Kind z

O.B.d.A. sei r rechtes Kind von x (links: analog)

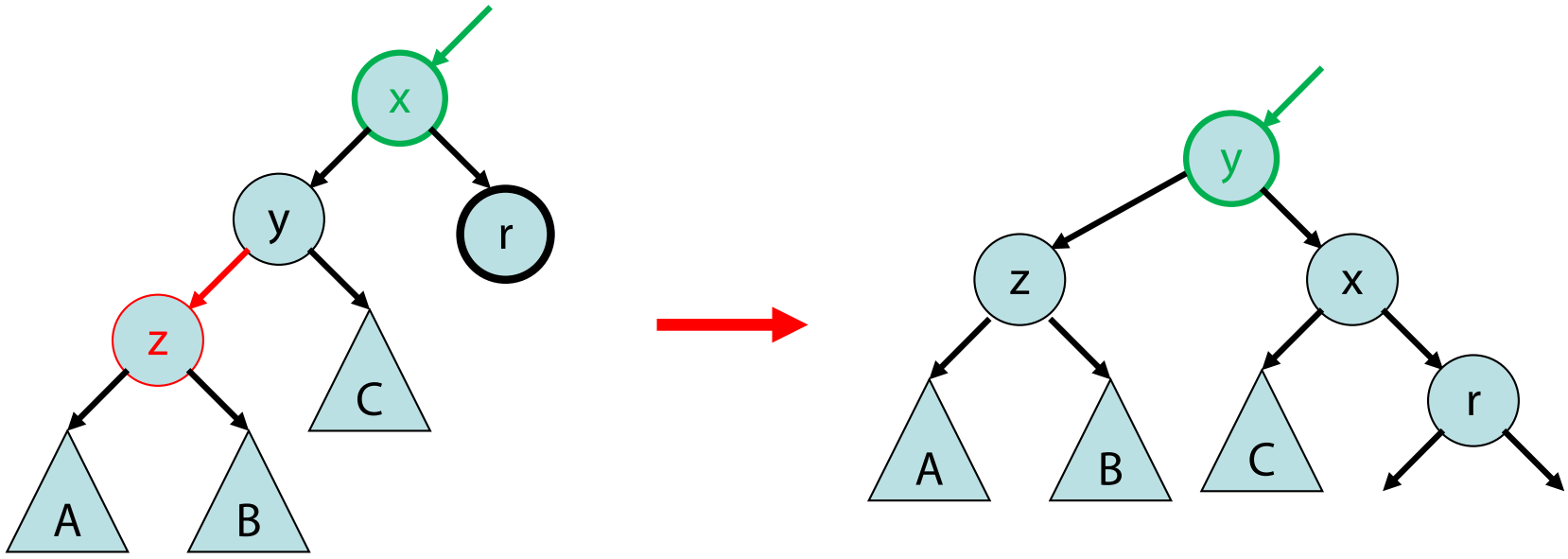
Alternativen für Fall 1: (x: beliebig gefärbt)



Rot-Schwarz-Baum

Fall 1: Bruder y von r ist schwarz, hat rotes Kind z

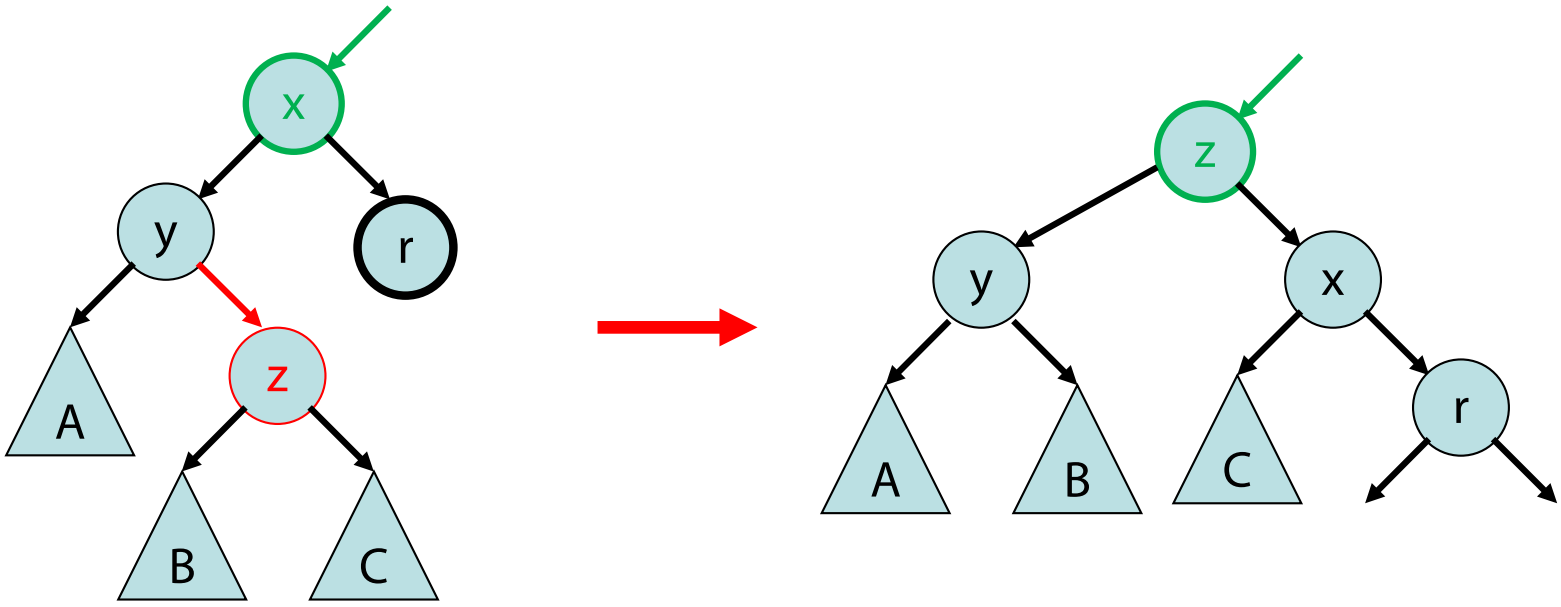
O.B.d.A. sei r rechtes Kind von x (links: analog)



Rot-Schwarz-Baum

Fall 1: Bruder y von r ist schwarz, hat rotes Kind z

O.B.d.A. sei r rechtes Kind von x (links: analog)

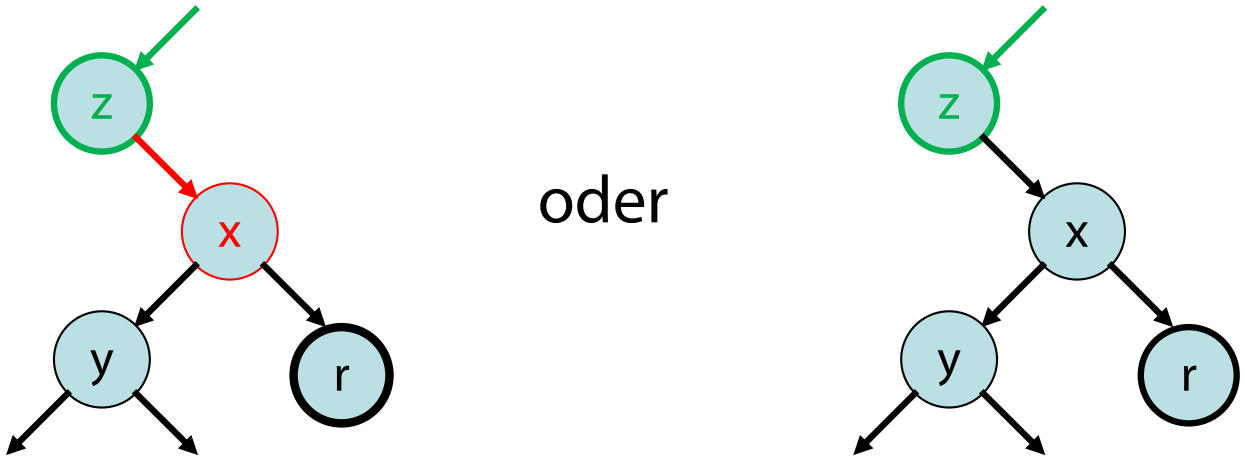


Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

Alternativen für Fall 2: (z beliebig gefärbt)

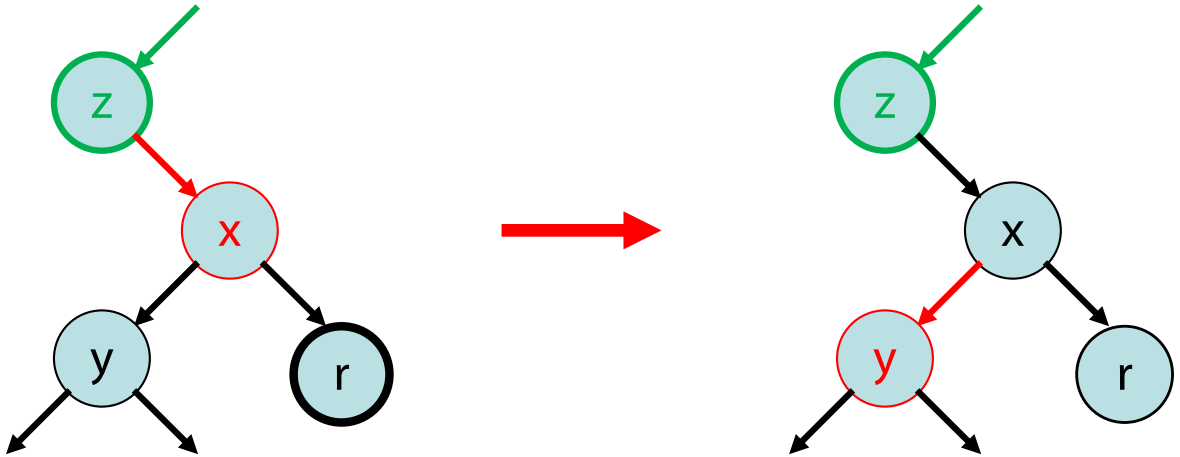


Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

2a)

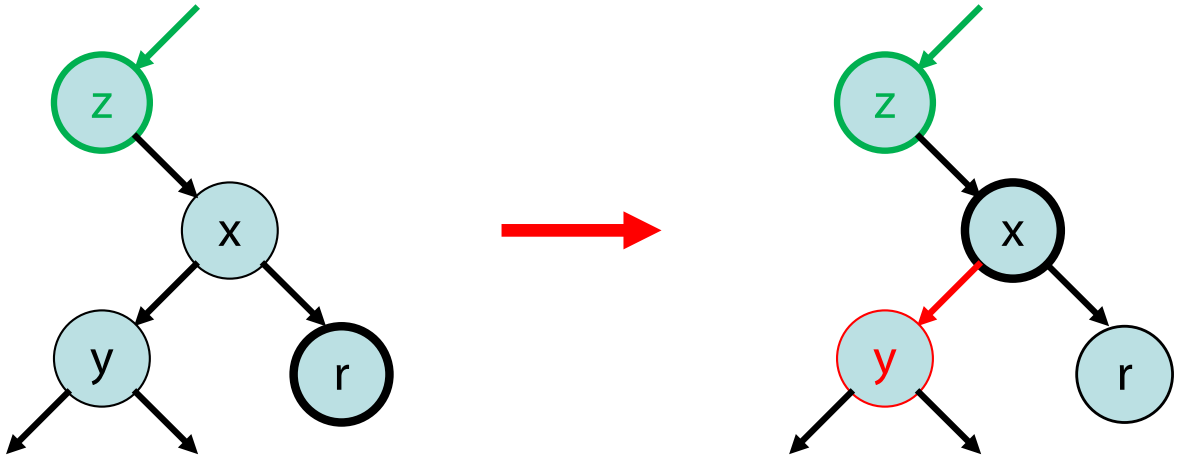


Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

2b)



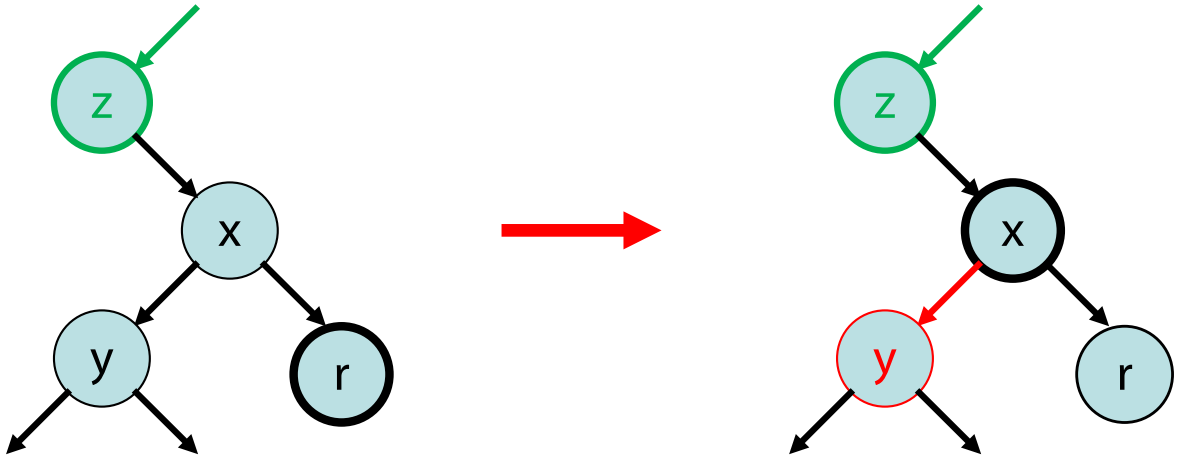
x ist Wurzel: fertig (Schwarztiefe-1)

Rot-Schwarz-Baum

Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

O.B.d.A. sei r rechtes Kind von x (links: analog)

2b)

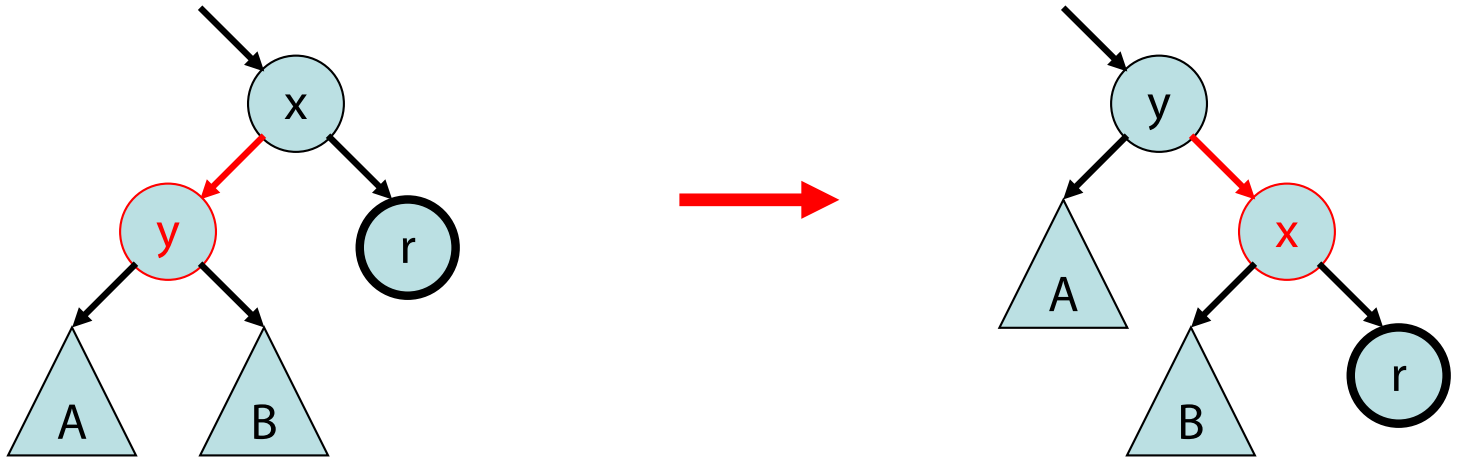


x keine Wurzel: weiter wie mit r

Rot-Schwarz-Baum

Fall 3: Bruder y von r ist rot

O.B.d.A. sei r rechtes Kind von x (links: analog)



Fall 1 oder 2a
→ terminiert dann

Rot-Schwarz-Baum

Laufzeiten der Operationen:

- search(k): O(log n)
- insert(e): O(log n)
- delete(k): O(log n)

Restrukturierungen:

- insert(e): max. 1
- delete(k): max. 2

Zum Vergleich:

Splay-Bäume

- search: O(log n) amort.
- insert: O(log n)
- delete: O(log n)

AVL-Bäume

- Ein binärer Suchbaum heißt **AVL-Baum**, falls für die beiden Teilbäume *T1* und *T2* der Wurzel gilt:
  - $|h(T1) - h(T2)| <= 1$
  - *T1* und *T2* sind ihrerseits AVL-Bäume.
- Der Wert $|h(T1) - h(T2)|$ wird als **Balancefaktor** (BF) eines Knotens bezeichnet. Er kann in einem AVL-Baum nur die Werte -1, 0 oder 1 (dargestellt durch -, = und +) annehmen.
- Jeder AVL-Baum ist ein binärer Suchbaum.
- Strukturverletzungen durch Einfügen oder Entfernen von Schlüsseln erfordern **Rebalancierungsoperationen**.

G. M. Adelson-Velskii, E. M. Landis; In: Doklady Akademii Nauk SSSR 146, S. 263-266, **1962**; Englische Übersetzung in Soviet Math 3, S. 1259-1263

Danksagung

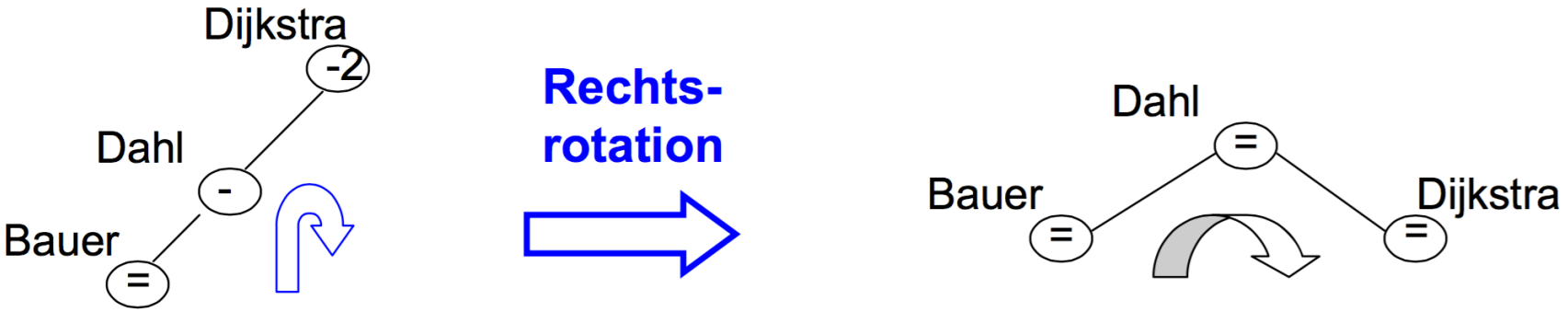
Die AVL-Präsentationen wurden übernommen aus:

- „Informatik II“ (Kapitel: Balancierte Bäume) gehalten von Martin Wirsing an der LMU http://www.pst.ifi.lmu.de/lehre/SS06/infoII/

Einfügen in AVL-Baum

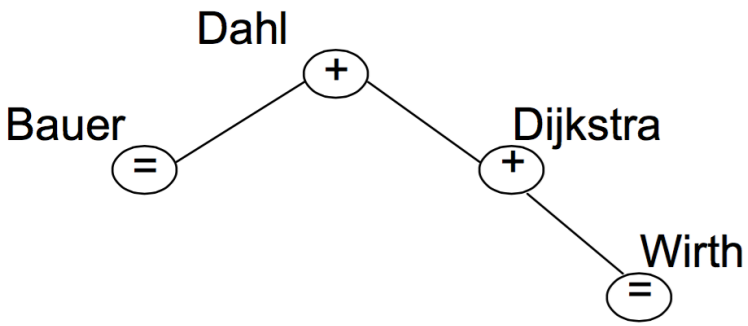
Einfügen von Dahl:
Neuberechnung des Balancierungsfaktors, AVL Kriterium erfüllt

Einfügen von Bauer: Verletzung des AVL Kriteriums



Nach Rechtsrotation: AVL Kriterium wieder erfüllt

Einfügen in AVL-Baum

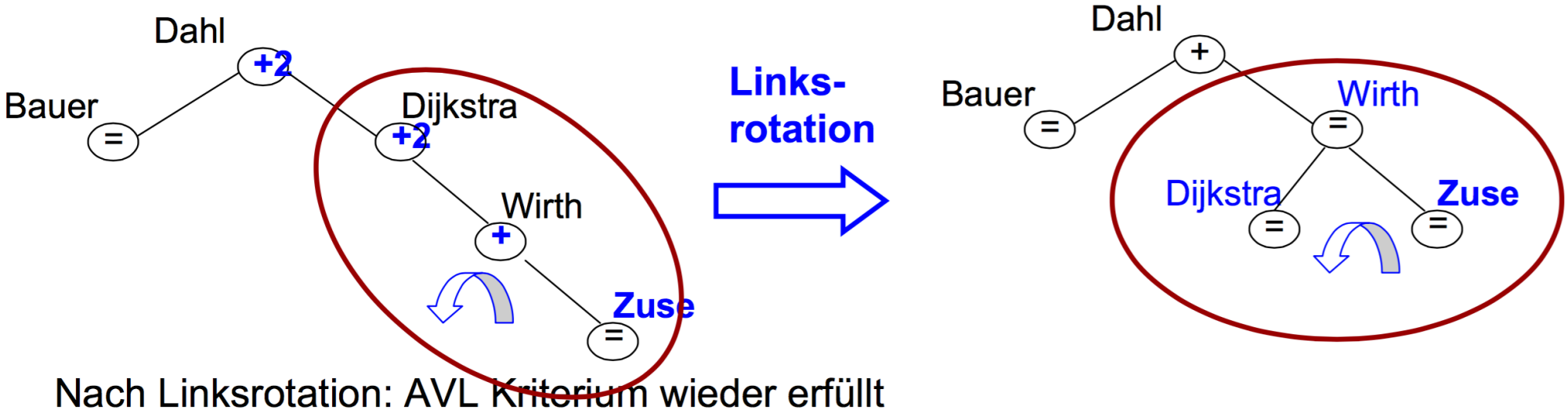


Einfügen von Wirth:

Nach Neuberechnung des Balancierungsfaktors ist

AVL-Kriterium weiter erfüllt

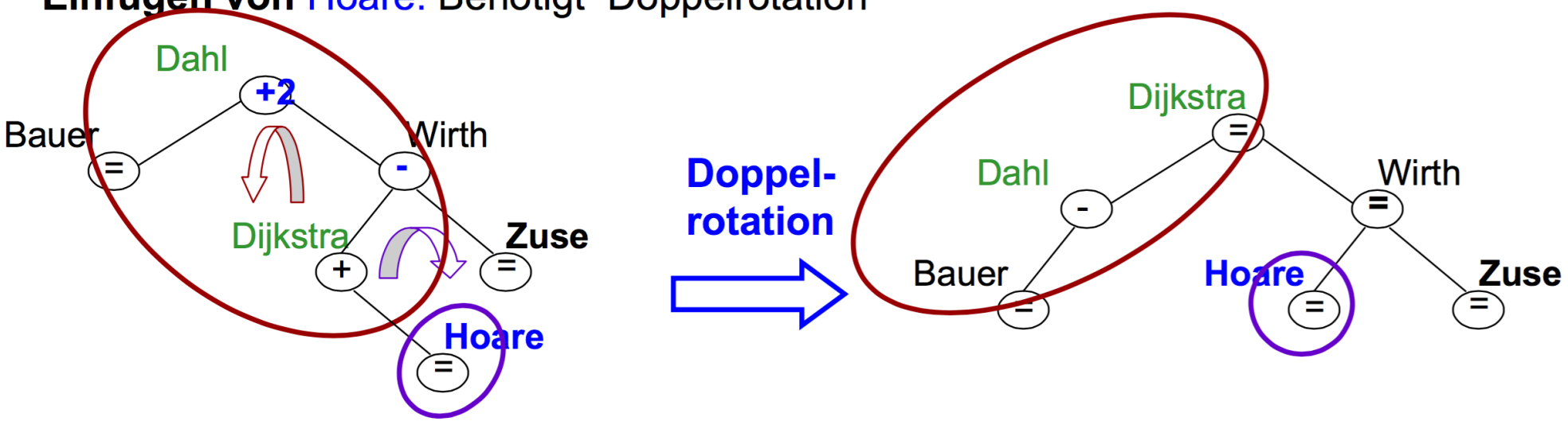
Einfügen von Zuse:



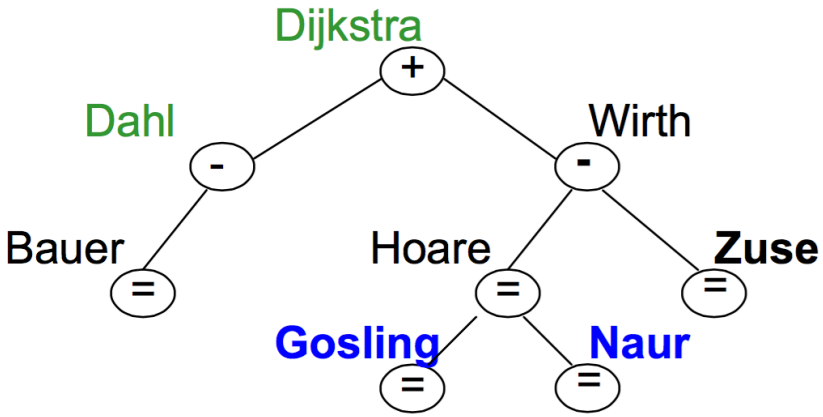
Nach Linksrotation: AVL Kriterium wieder erfüllt

Einfügen in AVL-Baum

Einfügen von Hoare: Benötigt Doppelrotation

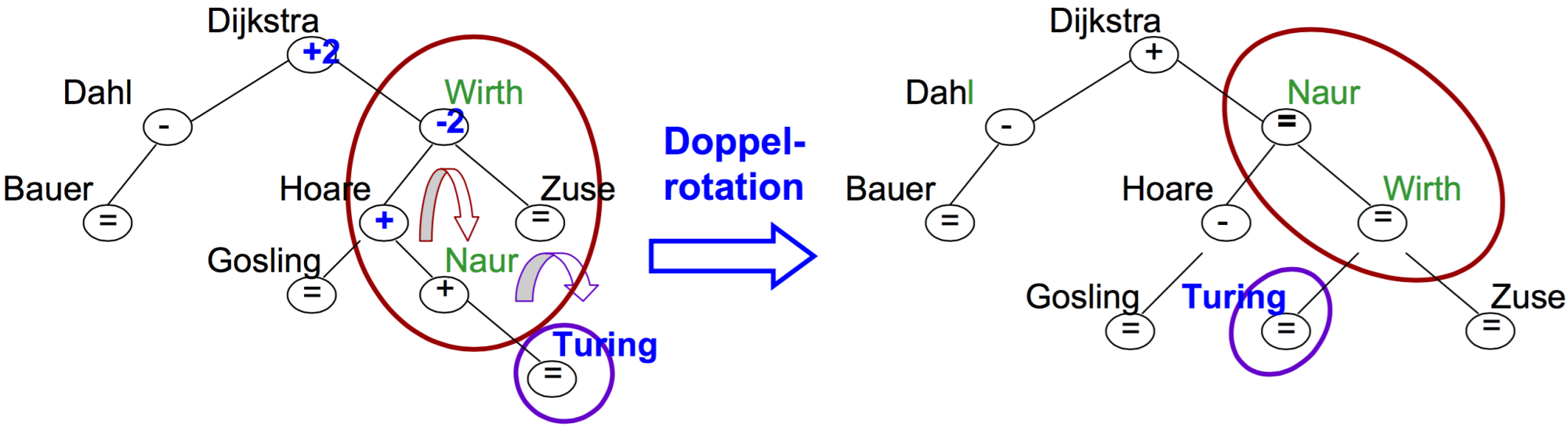


Einfügen von Gosling und Naur: ohne Probleme



Einfügen in AVL-Baum

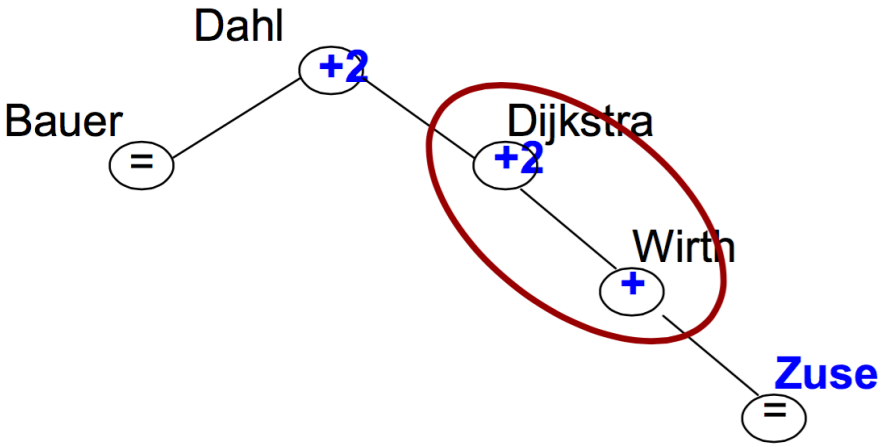
Einfügen von Turing: Noch einmal Doppelrotation

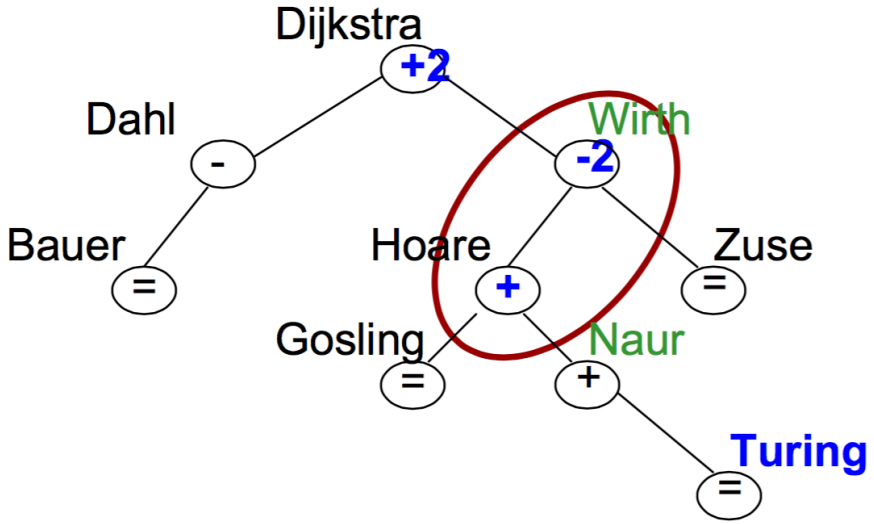


- Doppelrotation stellt AVL Kriterium wieder her
- Sind die vorgestellten Rotationen ausreichend?

Anwendungsstelle der Rotation

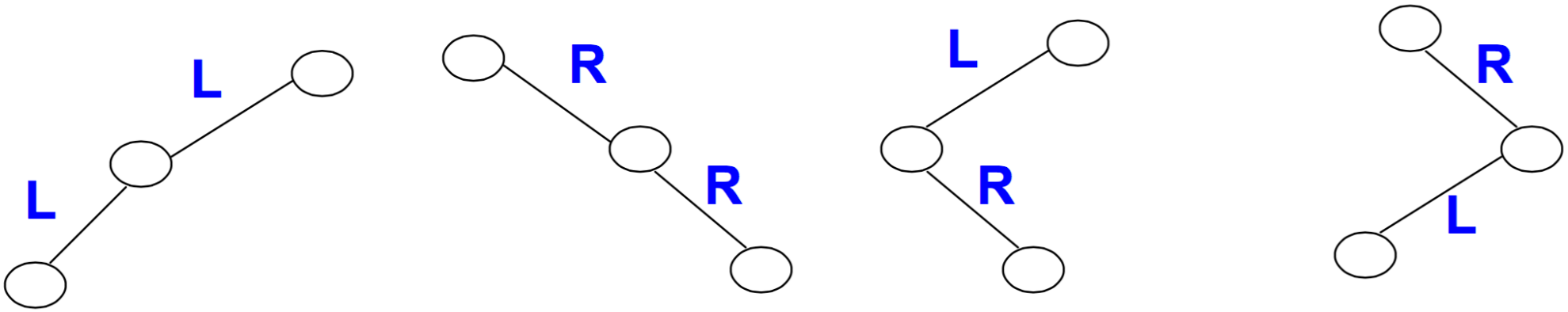
- Veränderungen der Balancierungsfaktoren geschehen ausschließlich auf dem **Pfad von der Wurzel zur Einfügeposition**
- Ausgangspunkt der Rotation ist immer der „**tiefste“ Elternknoten mit BF = ±2** (dieser Knoten hatte vorher BF = ±1)
- Der (auf dem Pfad) **darunter liegende Knoten** hat **BF = ±1**





Rotationstypen

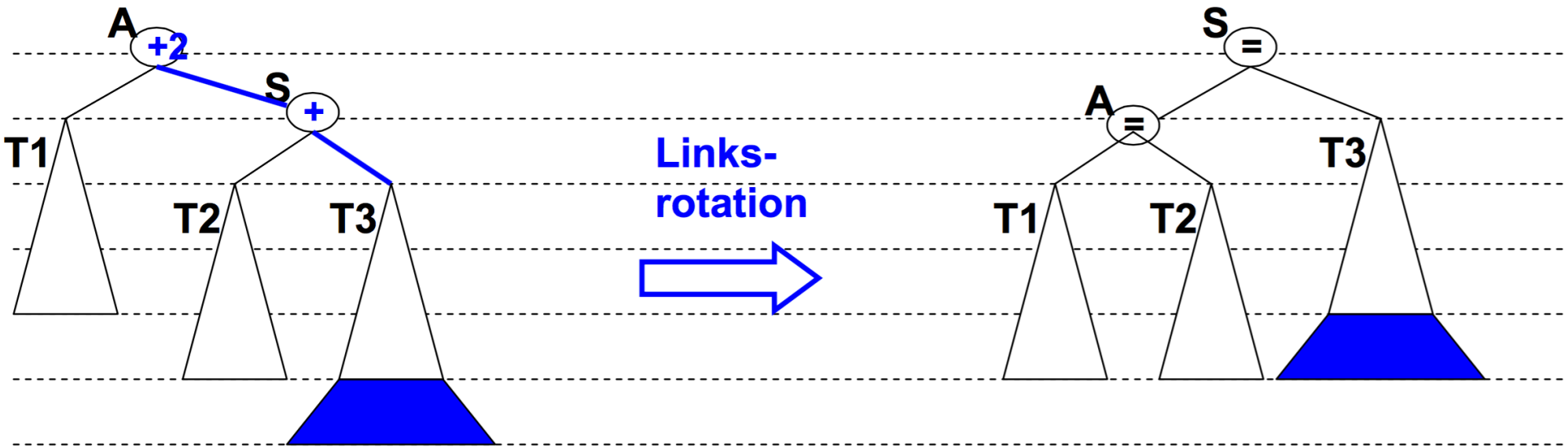
- Betrachte ausgehend vom tiefsten Knoten mit BF = 2 den Pfad zur Einfügeposition:
  - **RR: Rechts-Rechts** Linksrotation
  - **LL: Links-Links** Rechtsrotation
  - **RL: Rechts-Links** Doppelrotation „rechts“
  - **LR: Links-Rechts** Doppelrotation „links“



- Rotation ist immer eindeutig bestimmt
- Jetzt genauere Betrachtungen der einzelnen Typen

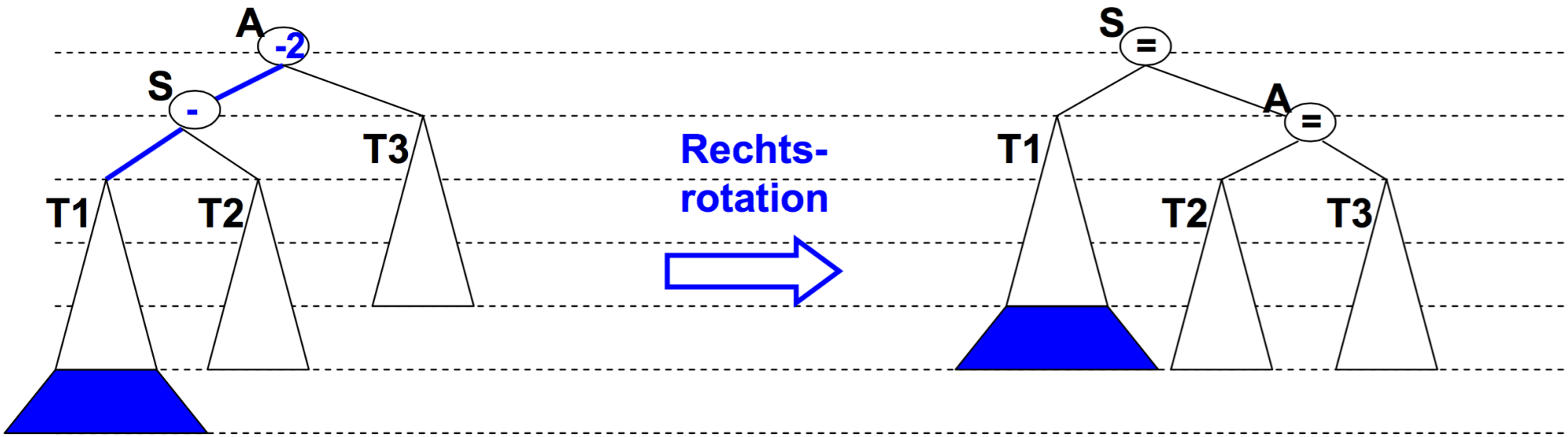
Typ RR: Linksrotation

- Wir bezeichnen den „tiefsten“ Knoten mit Strukturverletzung mit A, dessen Kind mit S und den Enkelknoten mit B
- Bei der Linksrotation hat S den BF „+“ und A den BF „+2“



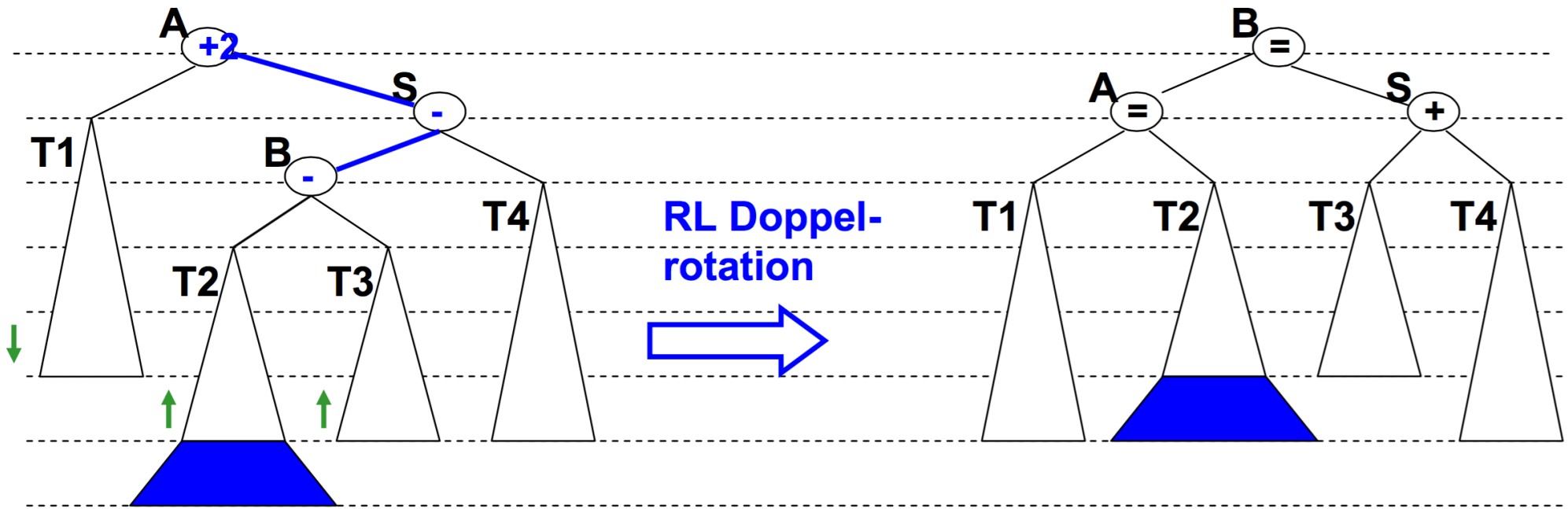
Typ LL: Rechtsrotation

- Bei der Rechtsrotation hat S den BF „-“ und A den BF „-2“



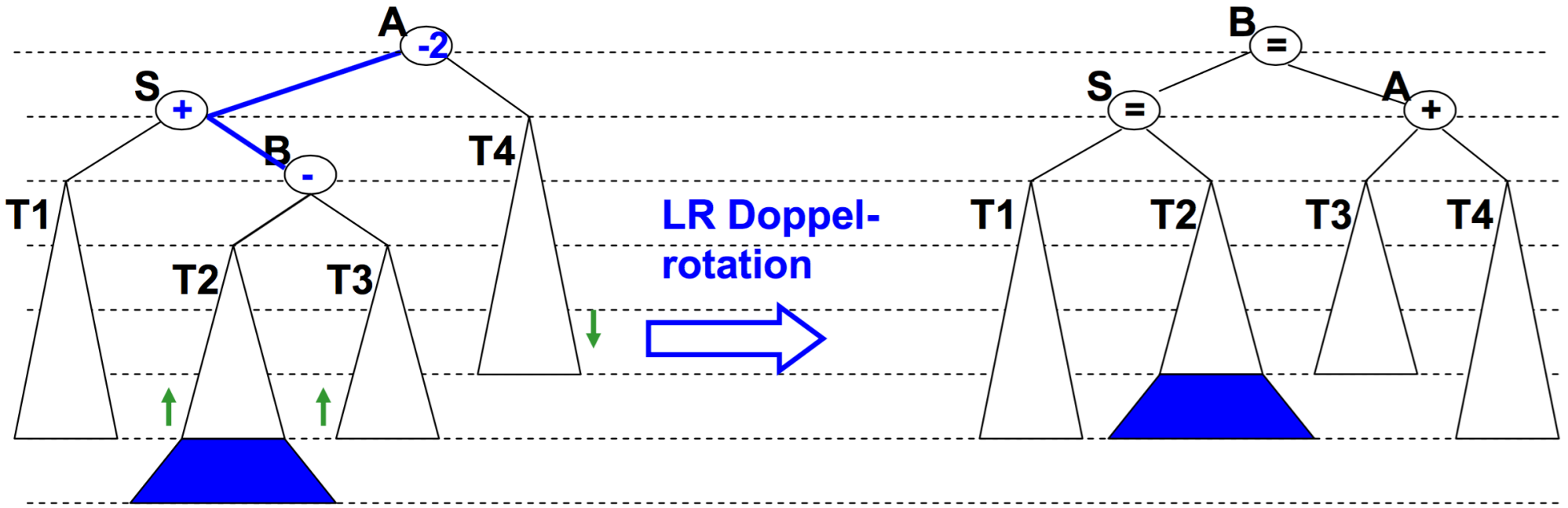
Typ RL: Doppelrotation

- Bei der RL-Doppelrotation hat A den BF „+2“, S den BF „- “, B den BF „+“ oder „-“
- Wir wählen „-“ für den BF von B.



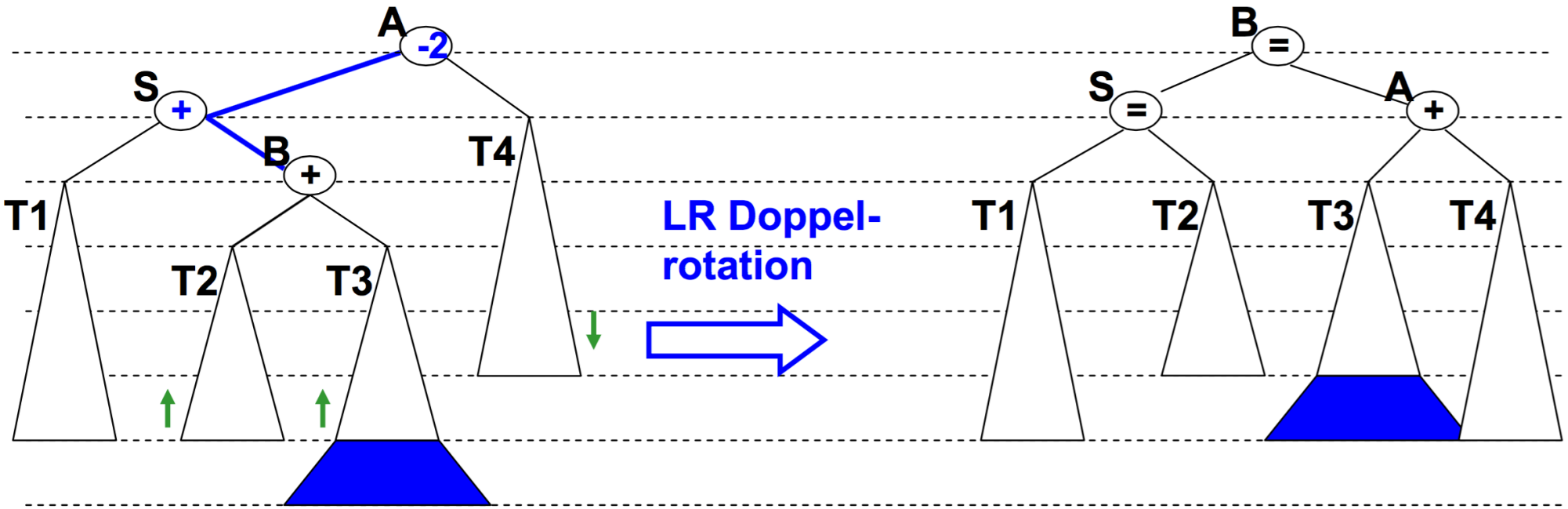
Typ LR: Doppelrotation

- Bei der LR-Doppelrotation hat A den BF „-2“, S den BF „+“, B den BF „+“ oder „-“
- Wir wählen „-“ für den BF von B.



Typ LR: Doppelrotation

- Variante der LR-Doppelrotation mit Balancefactor „+“ für B



Löschen von Knoten in AVL-Bäumen

- Löschen erfolgt wie bei Suchbäumen und ...
- kann (wie Einfügen) zu Strukturverletzungen führen, ...
- die durch Rotationen ausgeglichen werden
  - Es genügt nicht immer eine einzige Rotation oder Doppelrotation
  - Im schlechtesten Fall:
    - auf dem Suchpfad bottom-up vom zu entfernenden Schlüssel bis zur Wurzel
    - auf jedem Level Rotation bzw. Doppelrotation

Vergleich

AVL-Baum:

- search(k): O(log n)
- insert(e): O(log n)
- delete(k): O(log n)

Restrukturierungen:

- insert(e): max. 1
- delete(k): max. log n

Rot-Schwarz-Baum:

- search(k): O(log n)
- insert(e): O(log n)
- delete(k): O(log n)

Restrukturierungen:

- insert(e): max. 1
- delete(k): max. 2

Splay-Baum:

- search: ≈O(log n) amort.
- insert: O(log n)
- delete: O(log n)