
Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Stefan Werner (Übungen)

sowie viele Tutoren



Dynamische Menge

- Datenstruktur, die Objekte verwaltet, für die Schlüssel definiert sind: (1)
 - einfache Menge
 - Multimenge
 - geordnete (Multi-)Menge
- Mengen können verändert werden (anders als in Mathematik)
- Obligatorische Operationen:
 - **test**(s, k): testet, ob ein Element mit Schlüssel k in s enthalten ist (liefert true/false)
 - **search**(s, k):
 - (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 $key \geq k$ gilt
 - (2) liefert eine Menge von Elementen aus s, deren Schlüssel k ist
 - **insert**(s, x): fügt das Element x in s ein (s wird ggf. modifiziert)
 - **delete**(s, x): 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)`



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 verketteter oder sogar zyklischer)

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

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(A, k)
  i := 1; n := length(A)
  while i ≤ n do
    if k = key(A[i]) then
      return true
    i := i + 1
  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 zugeworfenen 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: $7 \times 7 / 7$



Beispiel selbstanordnende Listen, MF-Regel

- Beispiel (für „beinahe“ Best Case)

Zugriff	(resultierende) Liste	Aufwand in zugriffene 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*



Suchstruktur für die Darstellung von Mengen

s: Menge von Elementen

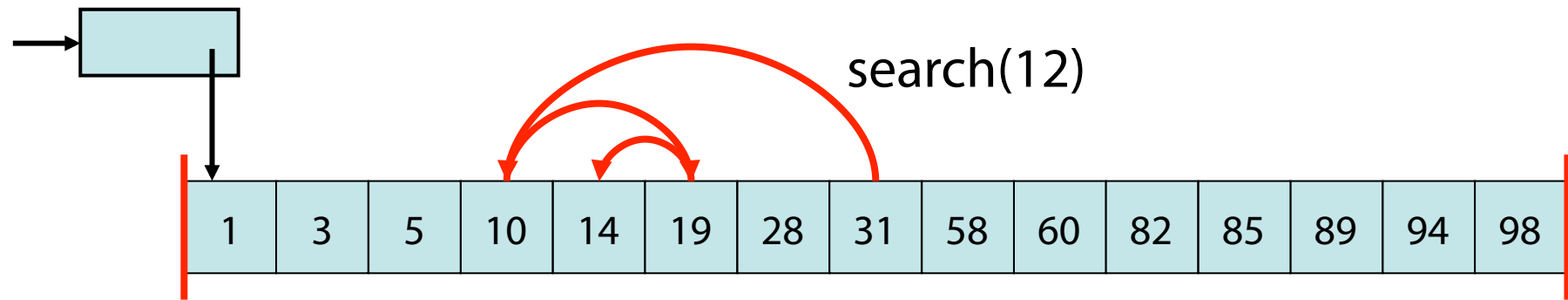
Jedes Element **x** identifiziert über **key(x)**.

Operationen:

- **insert(s, x)**: modifiziere **s** auf $s \cup \{x\}$
- **delete(s, k)**: modifiziere **s** auf $s \setminus \{x\}$, wobei **x** das Element ist mit $\text{key}(x)=k$
- **search(s, k)**: gib $x \in s$ aus mit minimalem **key(x)** so dass $\text{key}(x) \geq k$

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 $A[1], \dots, A[n]$

function search(A, k)

$n := \text{length}(A); l := 1; r := n$

while $l < r$ **do**

$m := (r+l) \text{ div } 2$

if $\text{key}(A[m]) = k$ **then return** $A[m]$

if $\text{key}(A[m]) < k$ **then** $l := m+1$

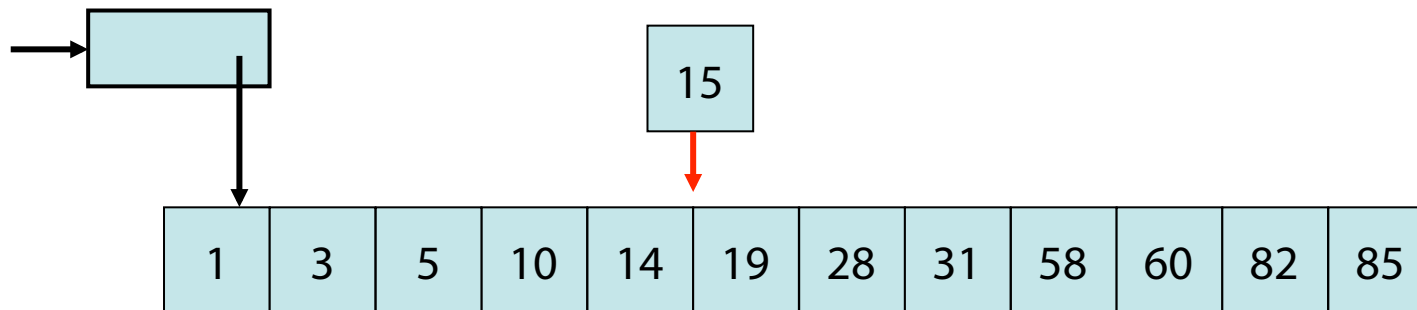
else $r := m$

return $A[l]$

Dynamische Suchstruktur

insert und delete Operationen:

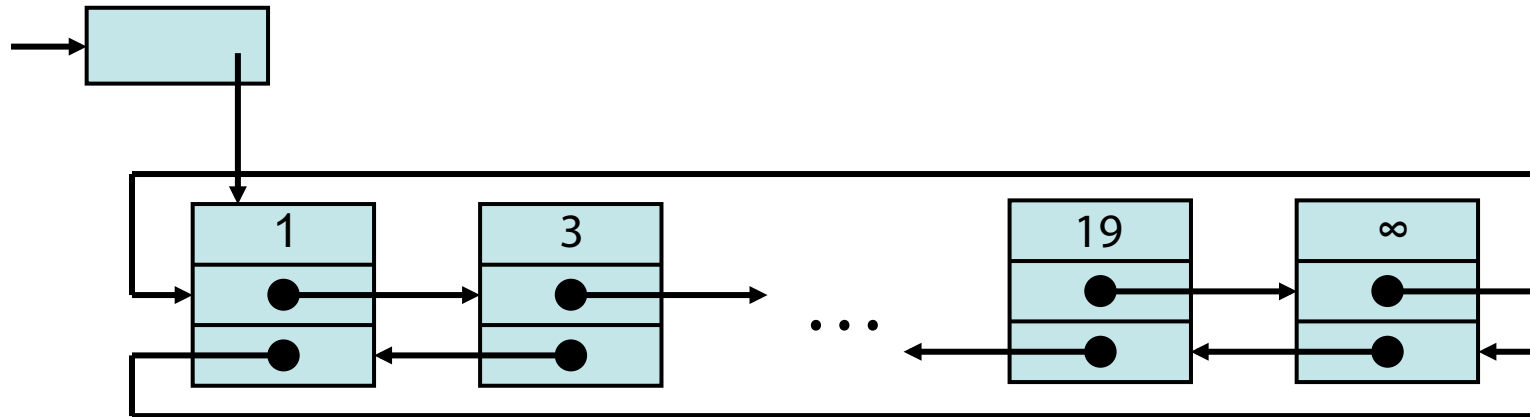
Sortiertes Feld schwierig zu aktualisieren!



Worst case: $\Theta(n)$ Zeit

Suchstruktur

2. Idee: Sortierte Liste

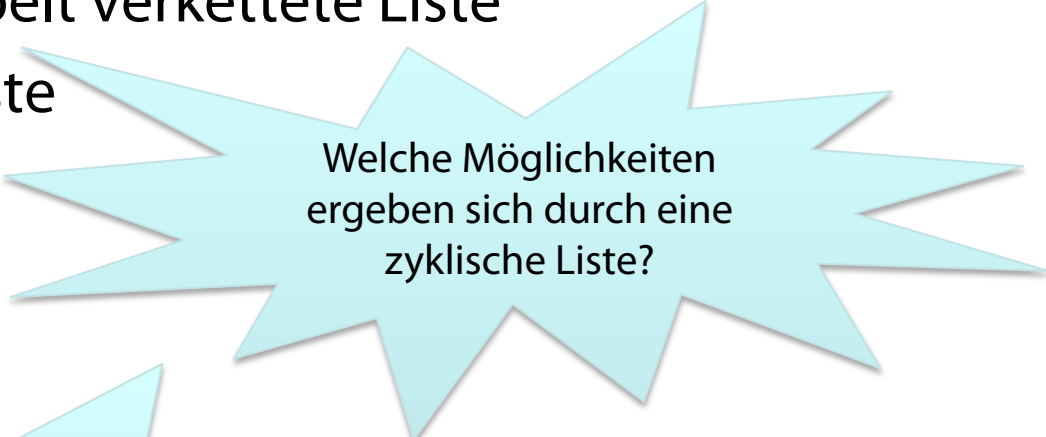


Problem: insert, delete und search kosten im worst case $\Theta(n)$ Zeit

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

Suchstruktur

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



Welche Möglichkeiten ergeben sich durch eine zyklische Liste?



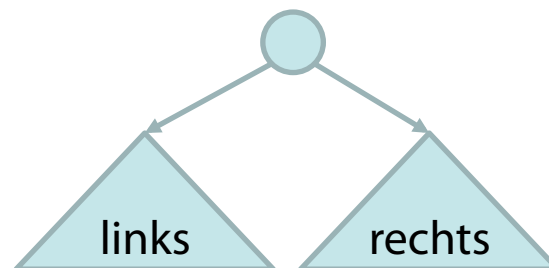
Was wäre die Folge bei einer einfach verketteten Liste?

Motivation: Suchen in Bäumen

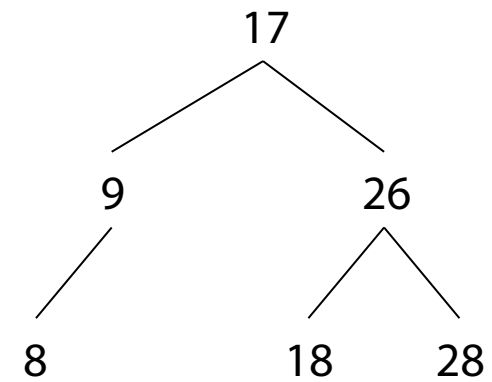
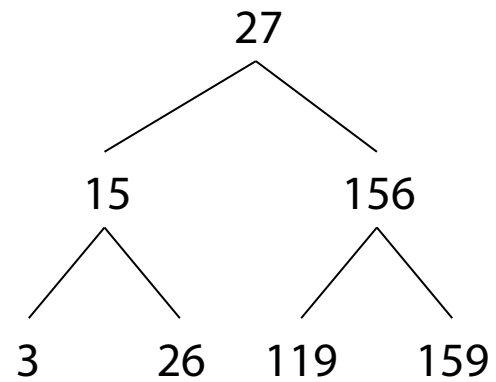
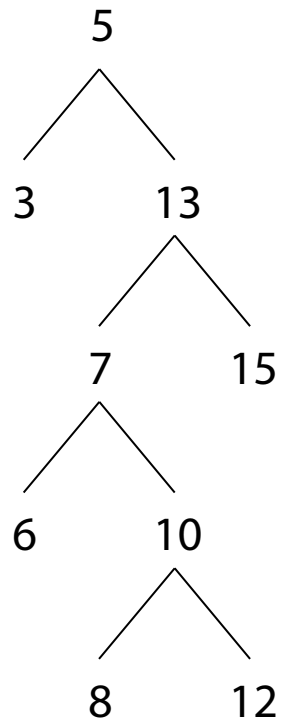
- Halbierungssuche durchsucht Felder
 - Einfügen neuer Elemente erfordert ggf. ein größeres Feld und Umkopieren der Elemente
- Einfügen in Listen in konstanter Zeit
 - Zugriffe auf Elemente jedoch sequentiell
- (Verzeigertes) Baum
 - Einfügen in konstanter Zeit möglich (falls Einfügeposition gegeben)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

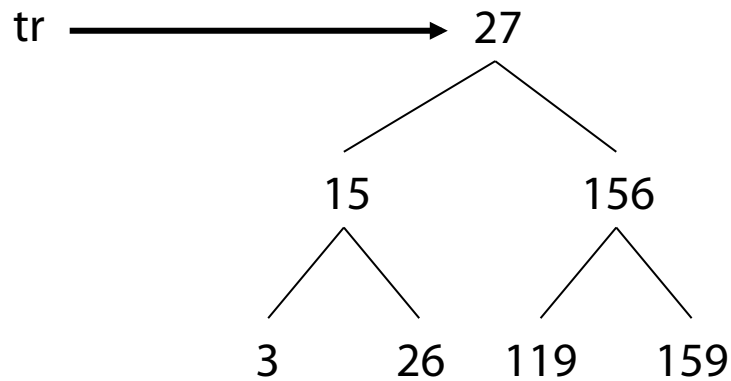


Beispiele für binäre Suchbäume



Aufgabe: Wende Funktion auf Elemente an

• Algorithmus?



fold(+, tr, 0)
→ 505

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

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

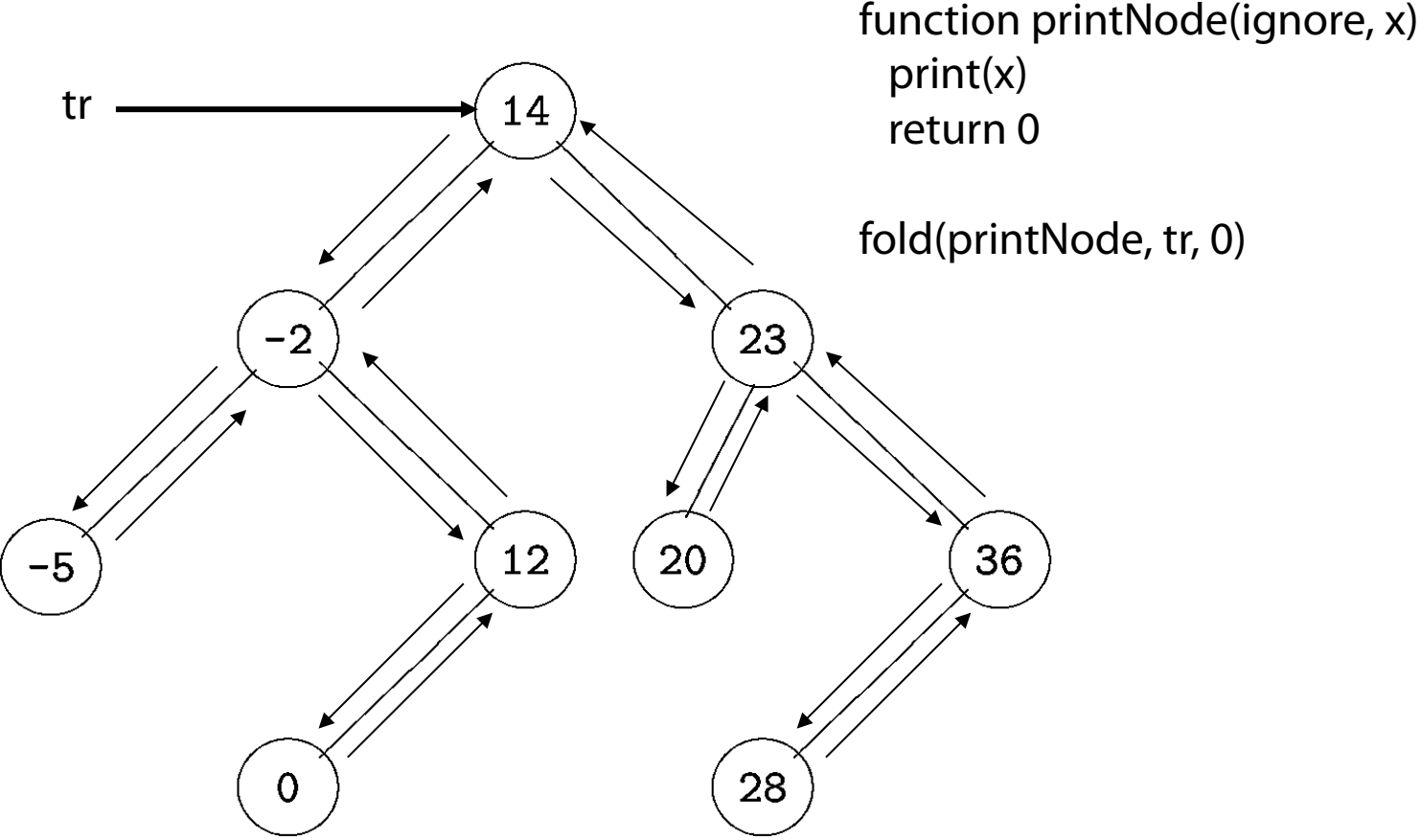
fold(max, tr, 0)
→ 159

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



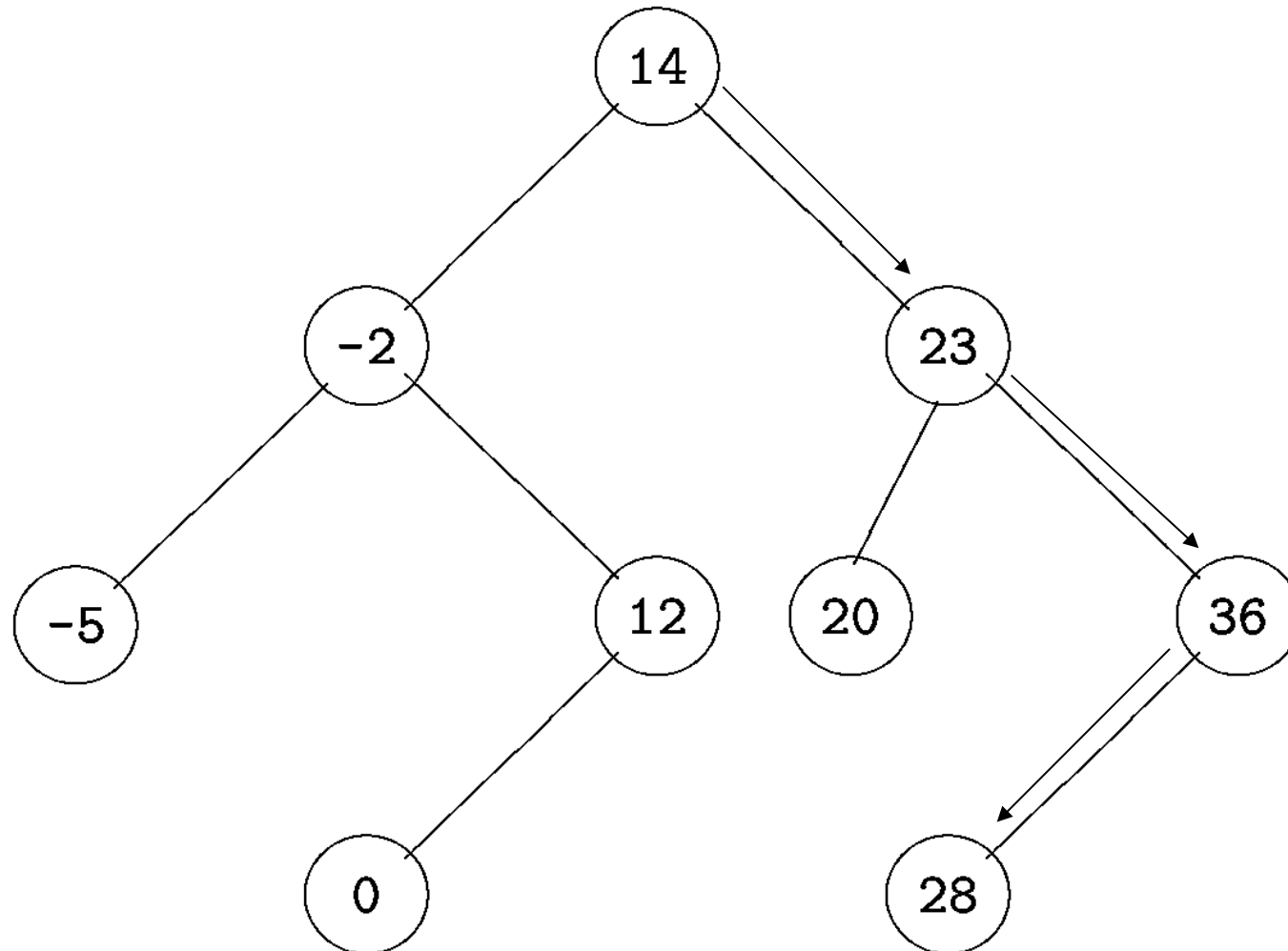
-5 -2 0 12 14 20 23 28 36



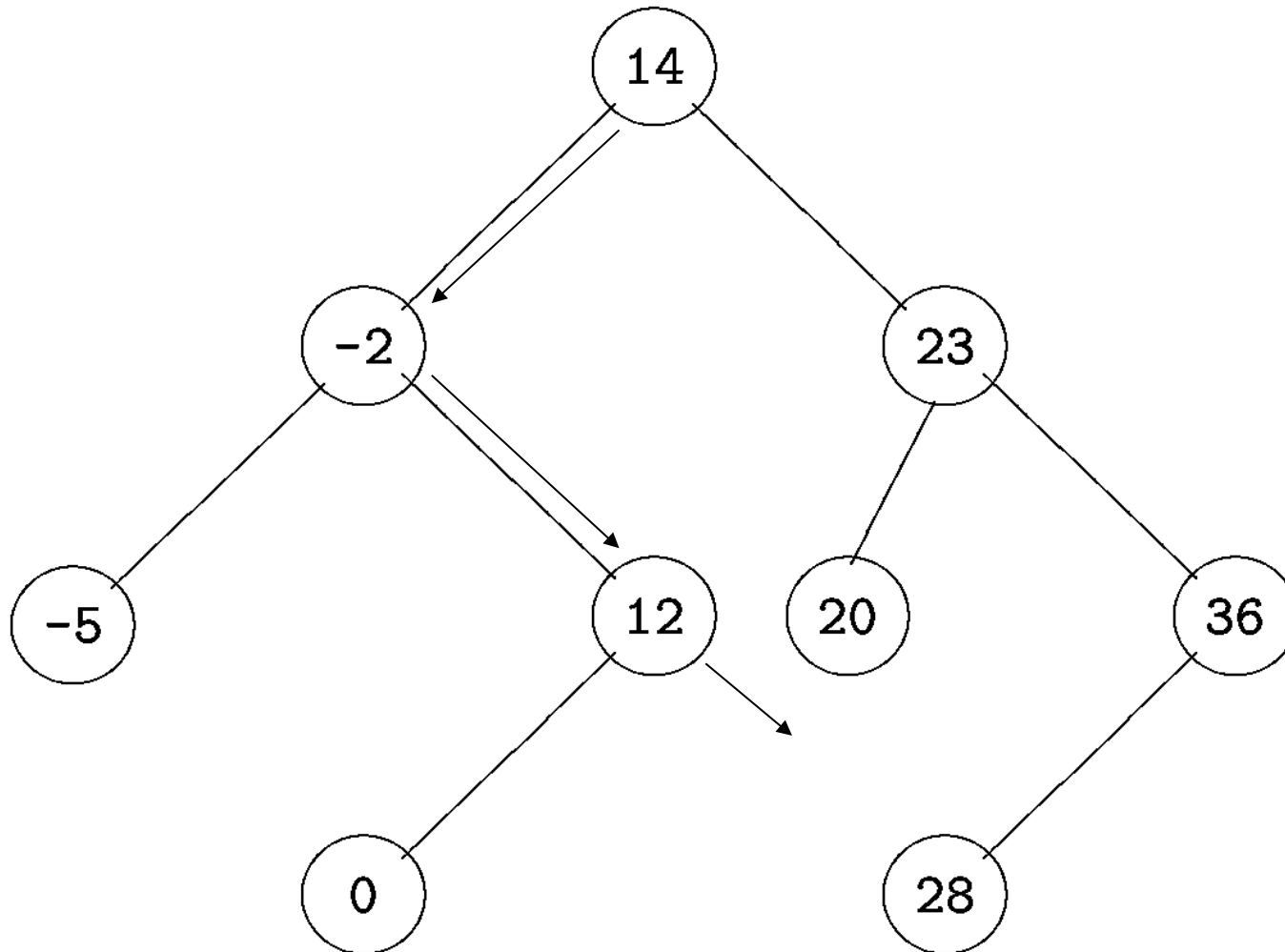
Suche im binären Suchbaum

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

Suche nach 28



Suche nach 13

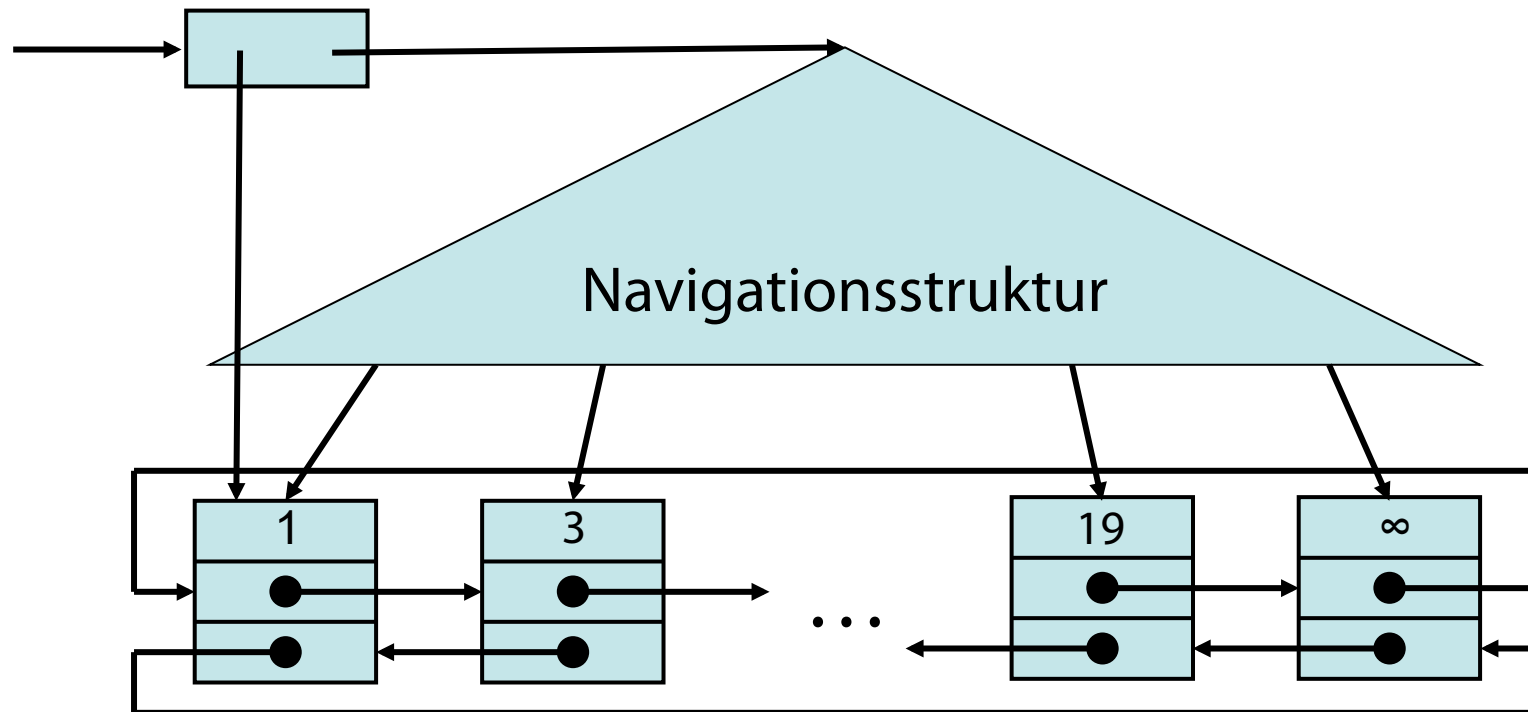


Binärer Suchbaum

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

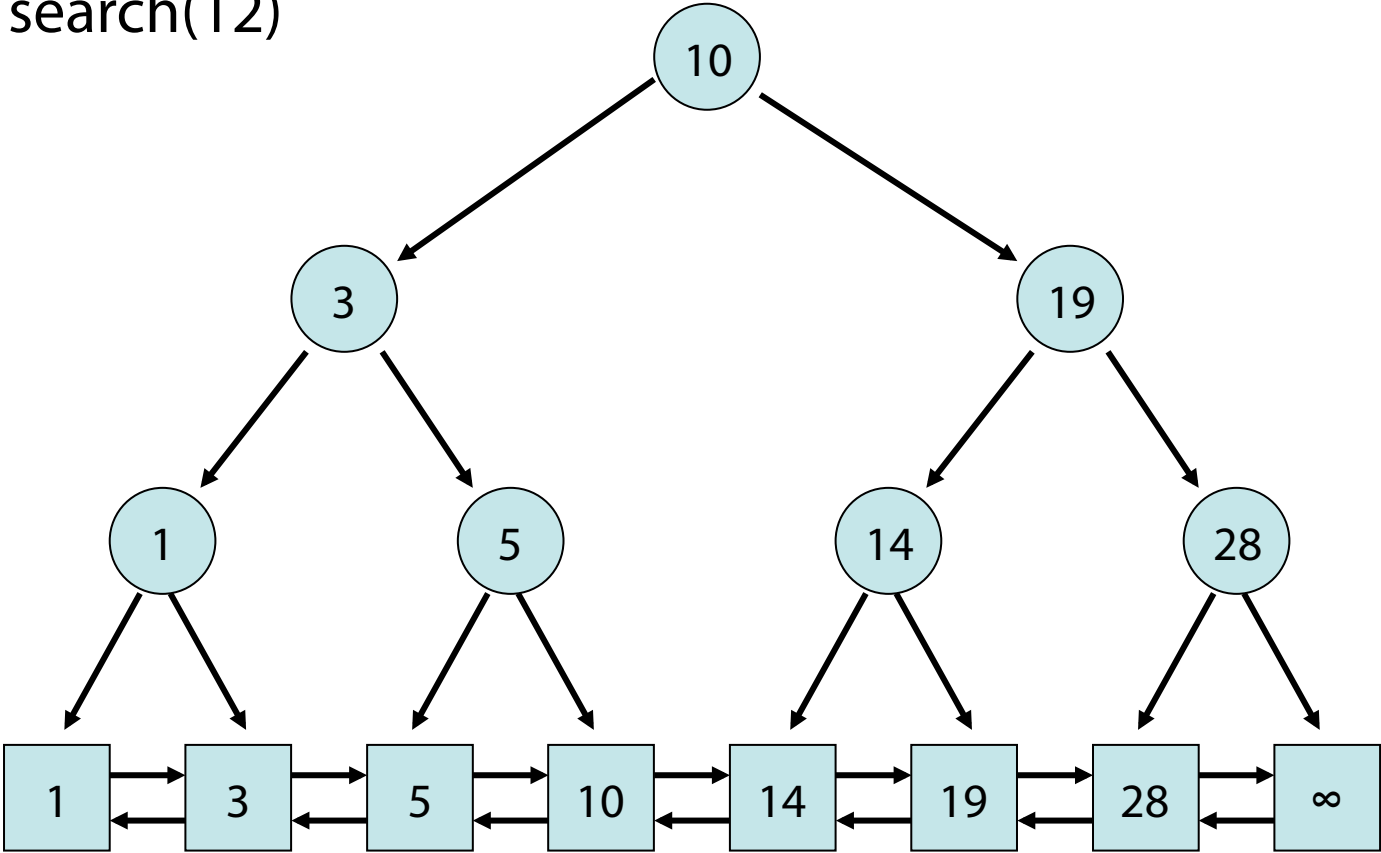
Suchstruktur

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



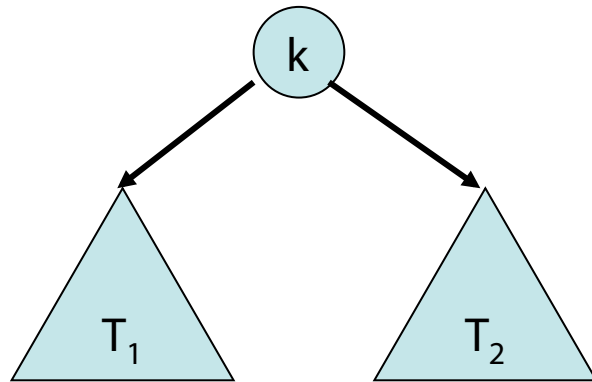
Binärer Suchbaum (ideal)

search(12)



Binärer Suchbaum

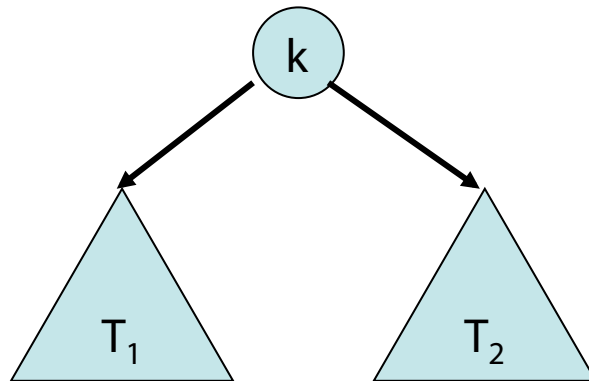
Suchbaum-Regel:



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:

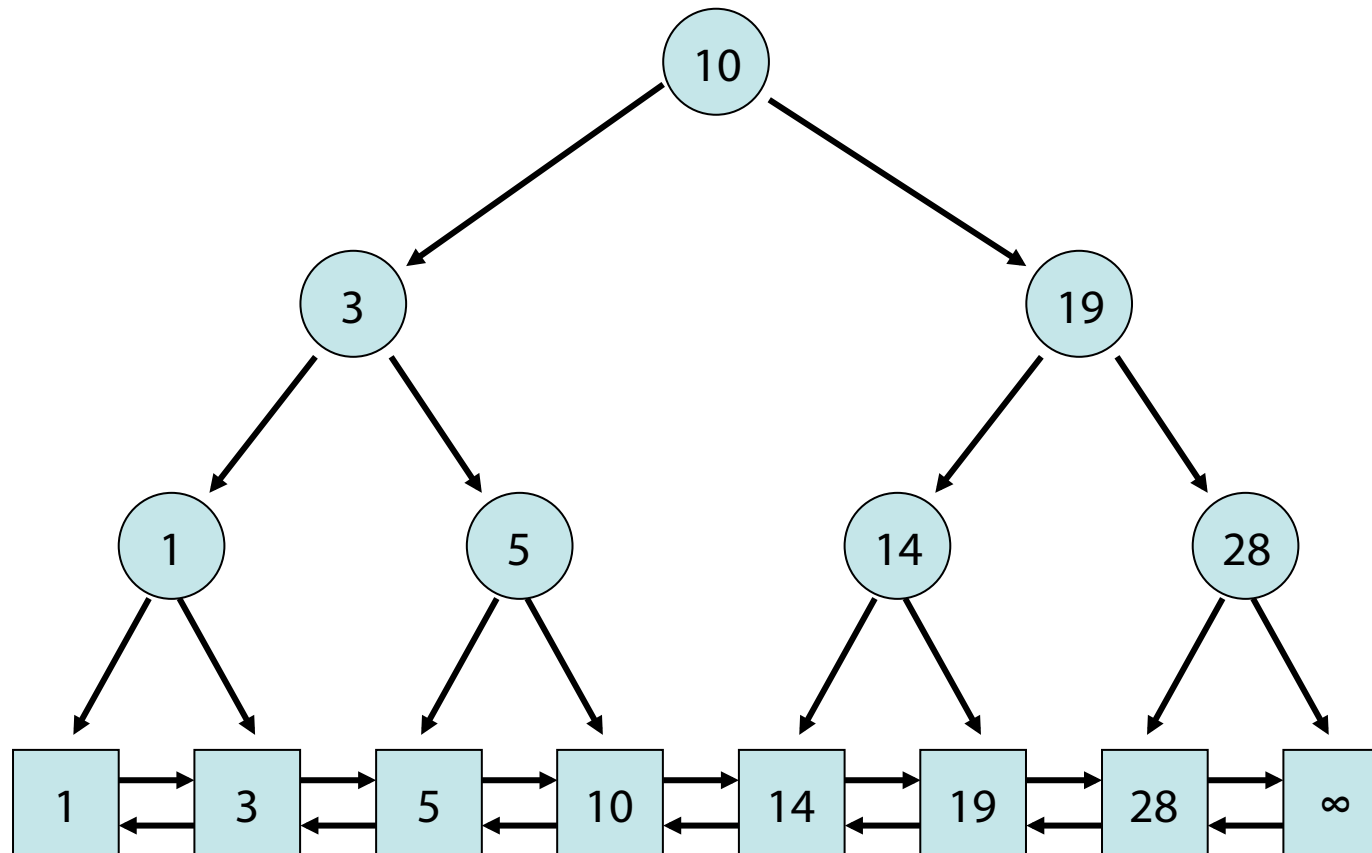
- Starte in Wurzel des Suchbaums
- Für jeden erreichten Knoten v :
 - Falls $\text{key}(v) \leq k$, gehe zum linken Kind von v , sonst gehe zum rechten Kind

Binärer Suchbaum

Formell: 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)

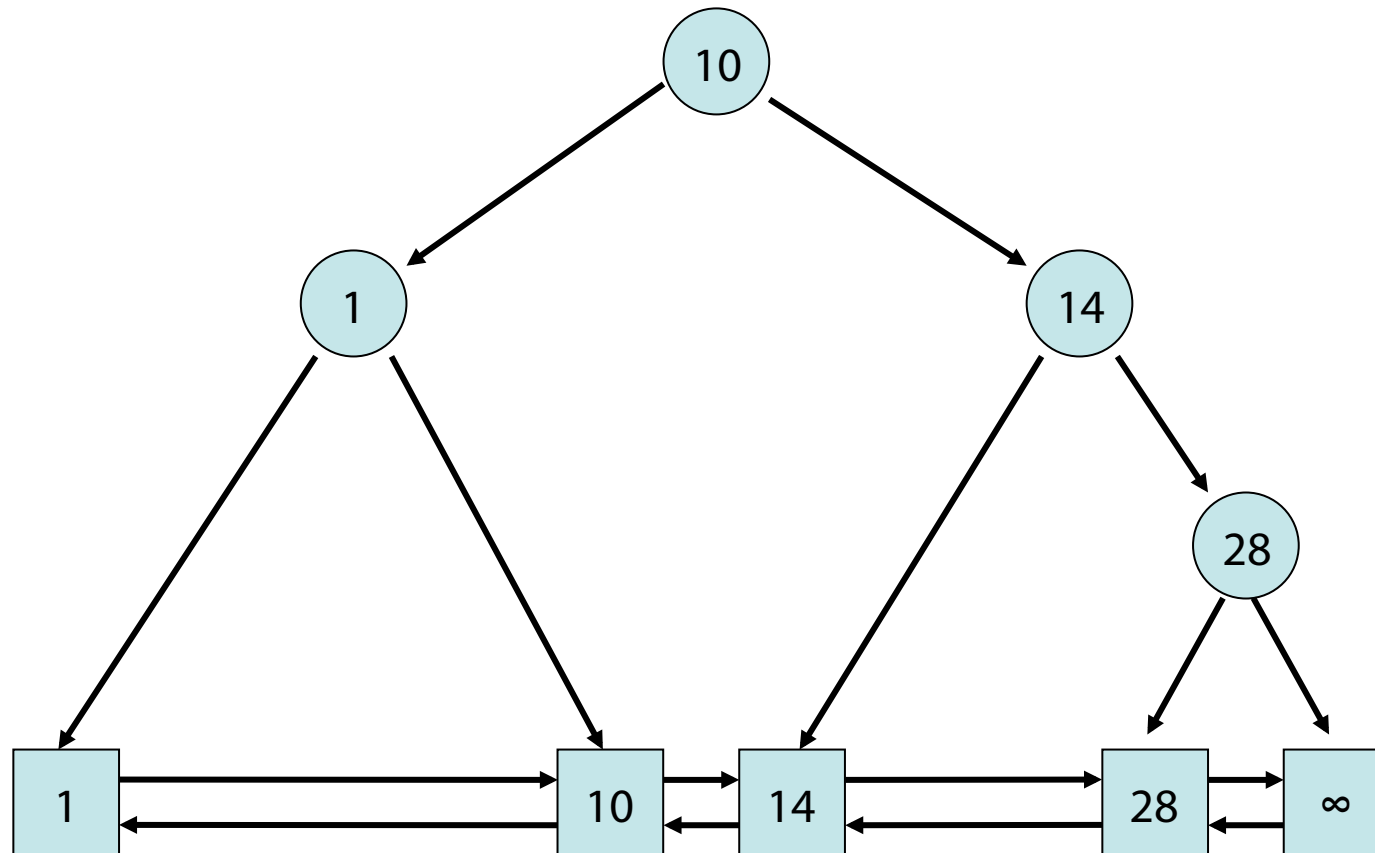


Insert und Delete Operationen

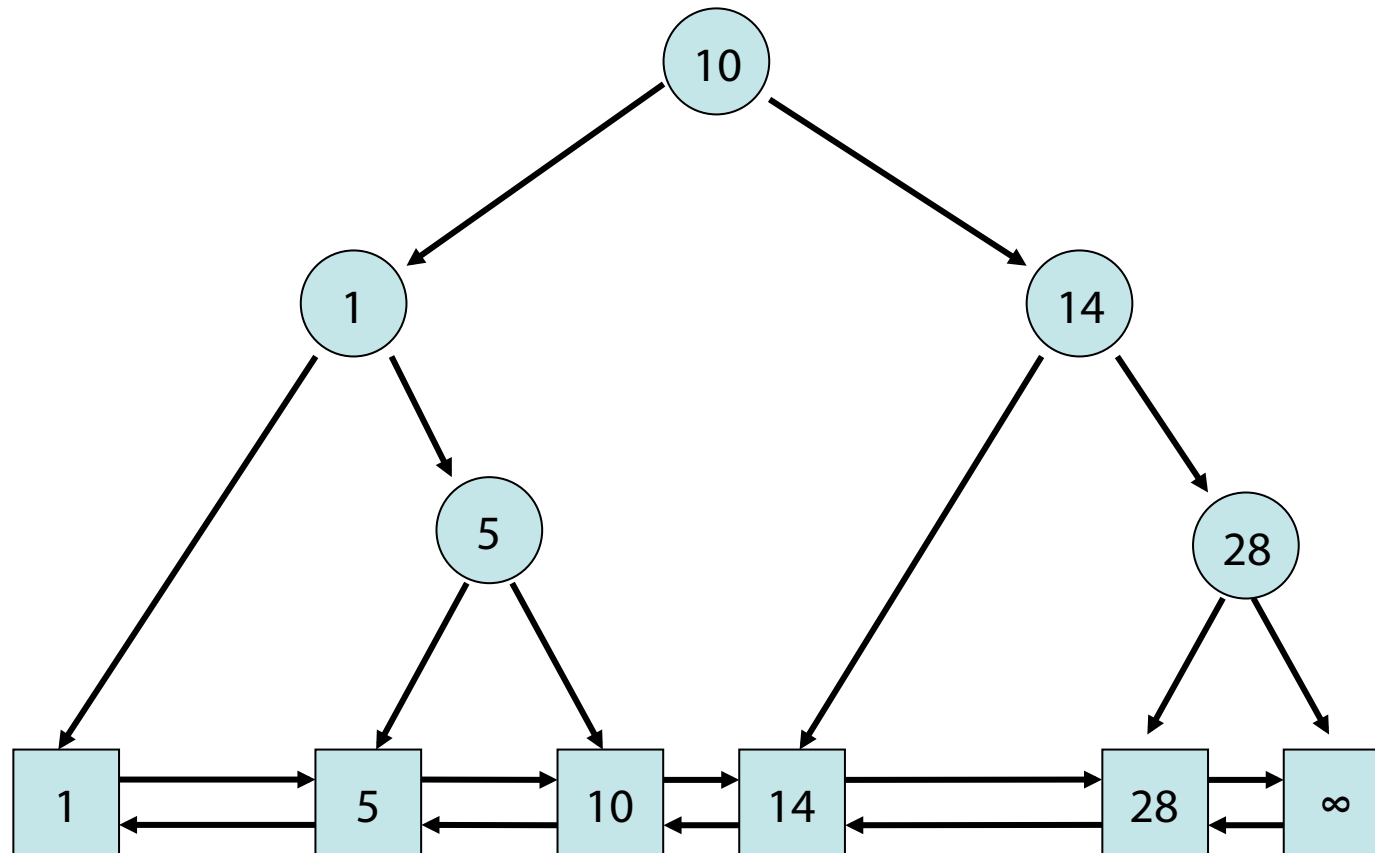
Strategie:

- **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)**

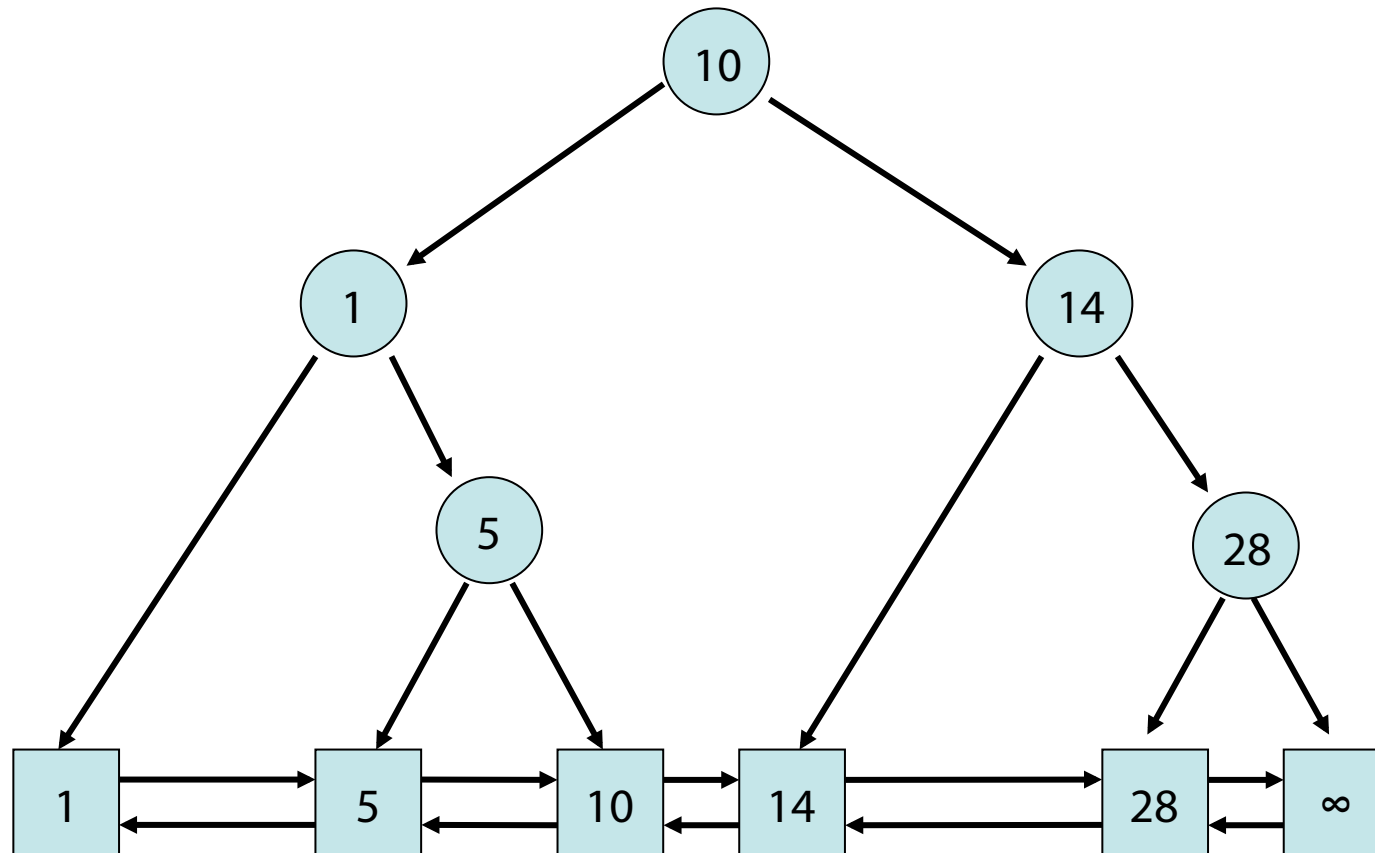
Insert(5)



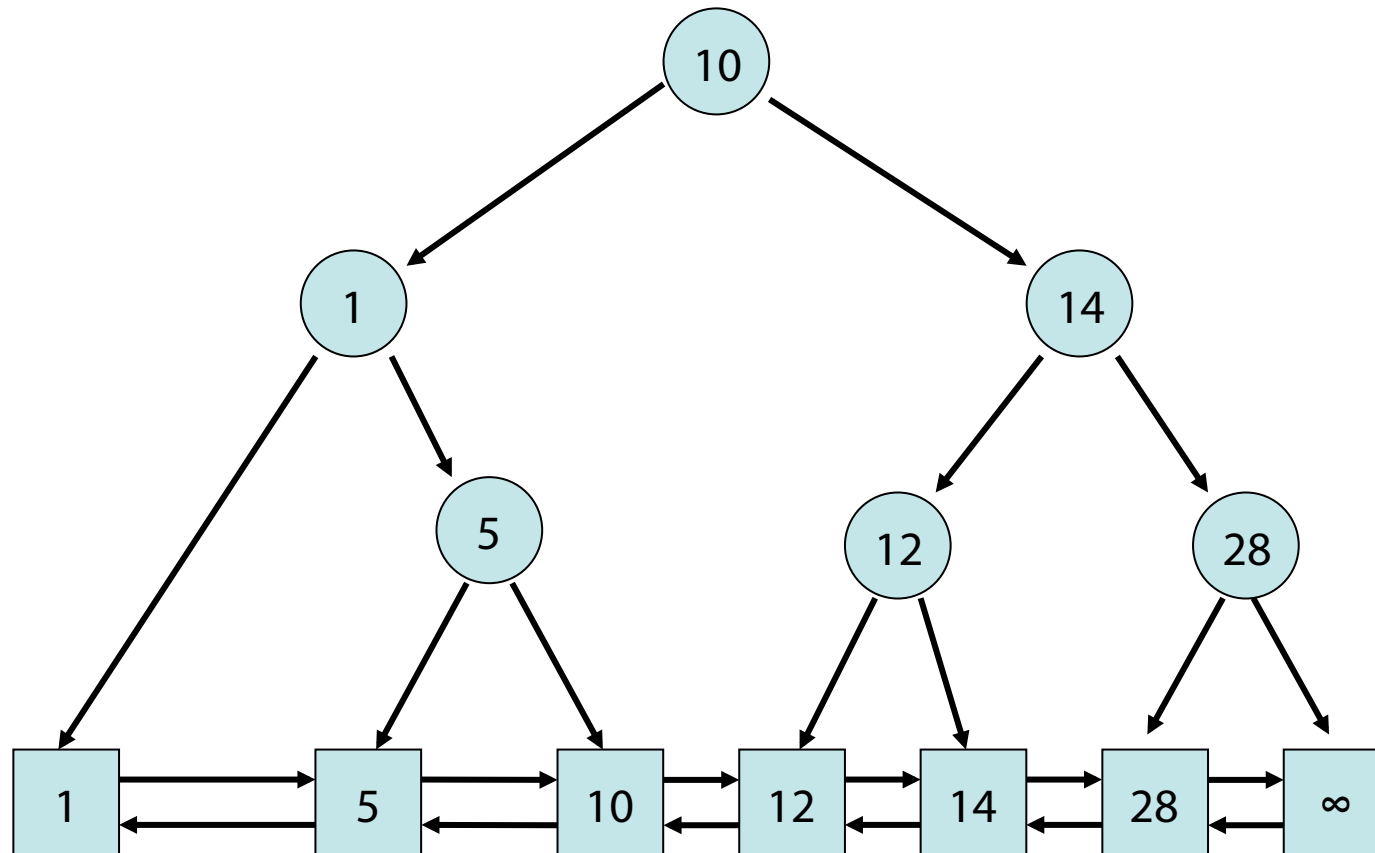
Insert(5)



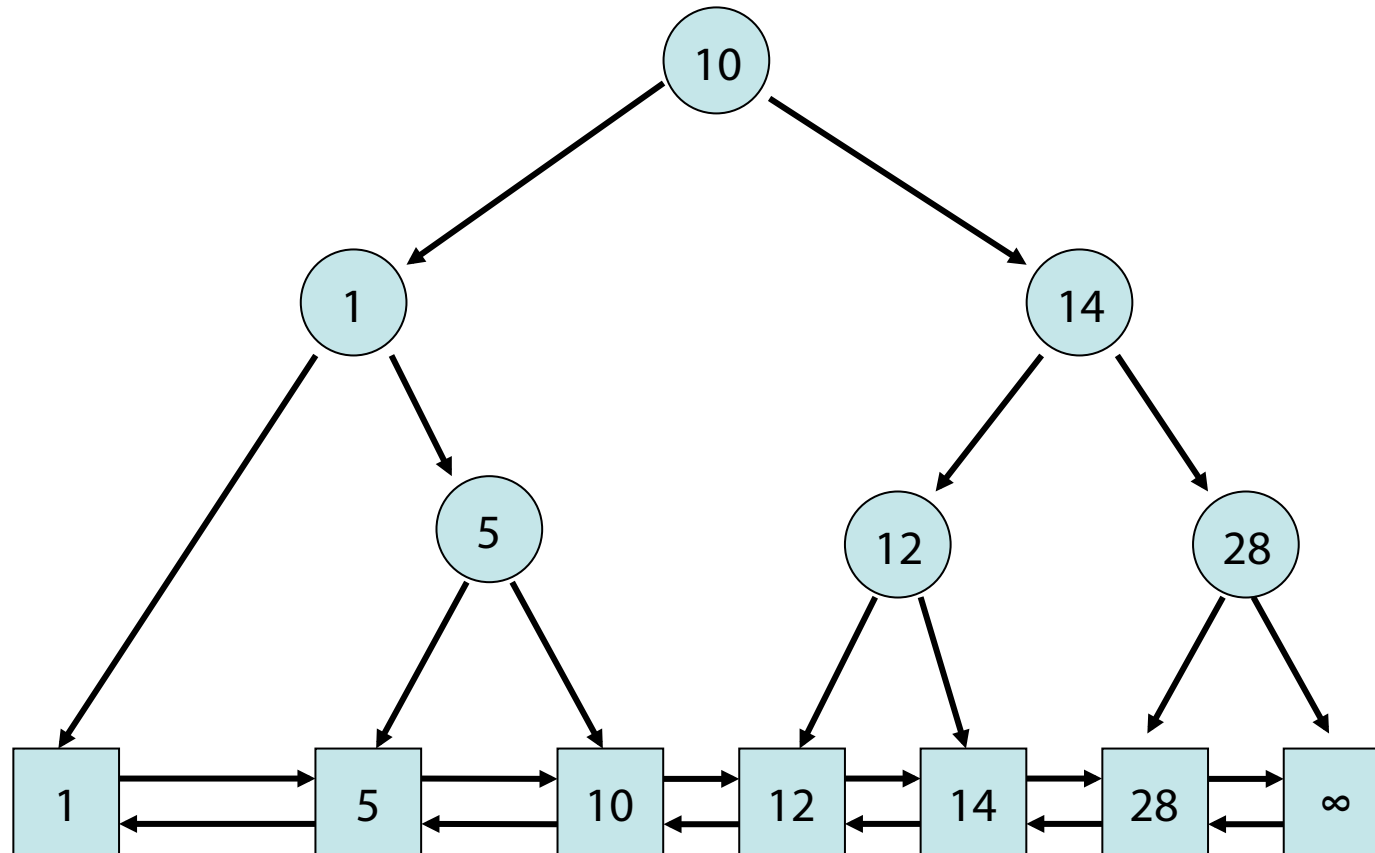
Insert(12)



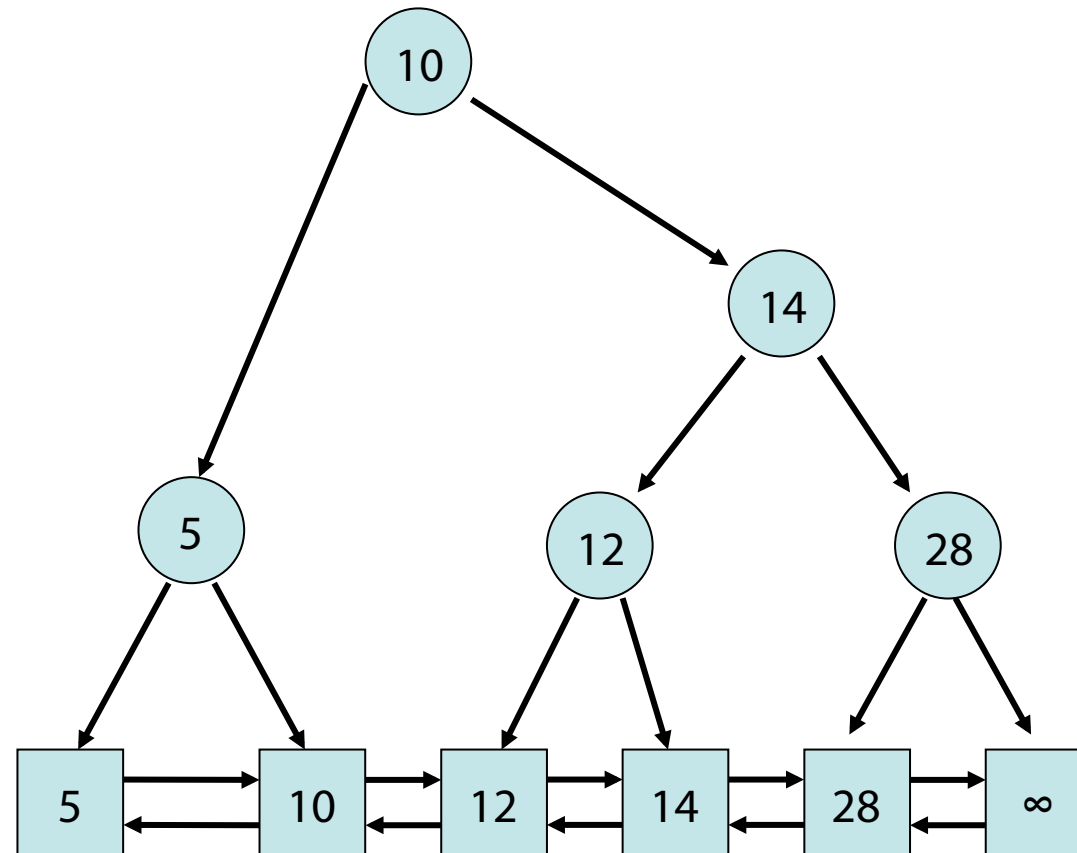
Insert(12)



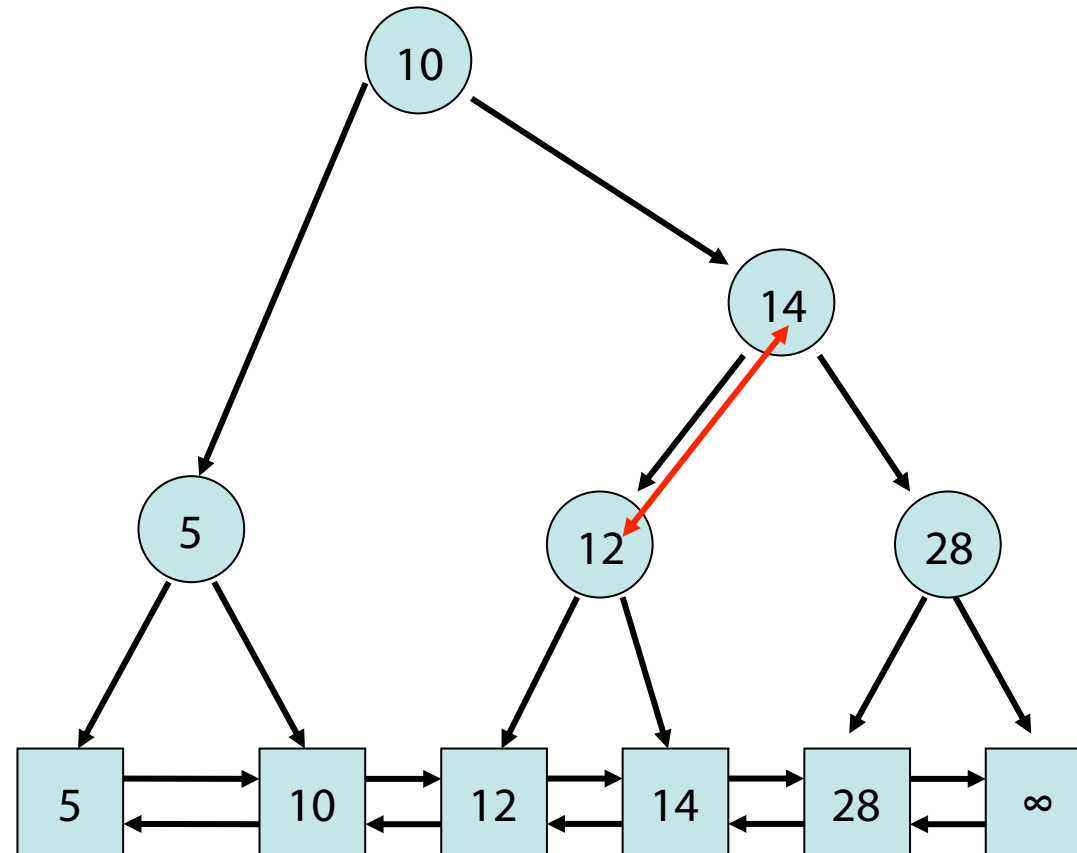
Delete(1)



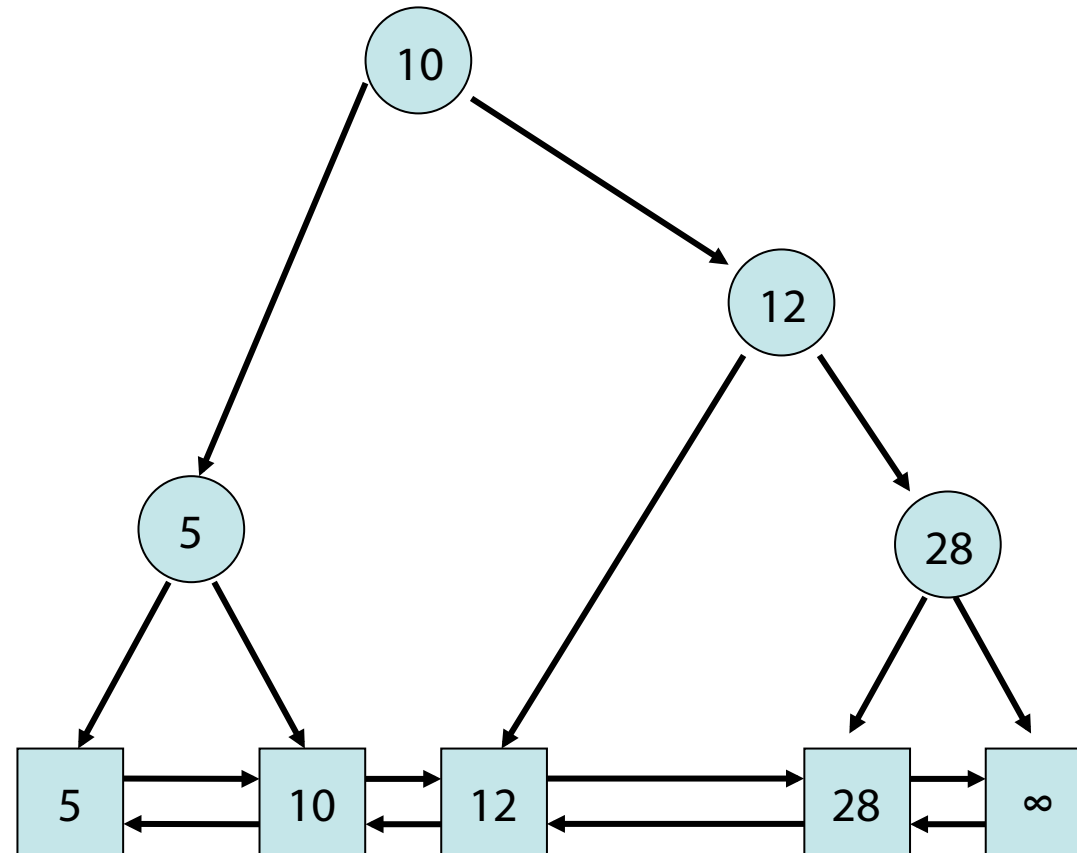
Delete(1)



Delete(14)



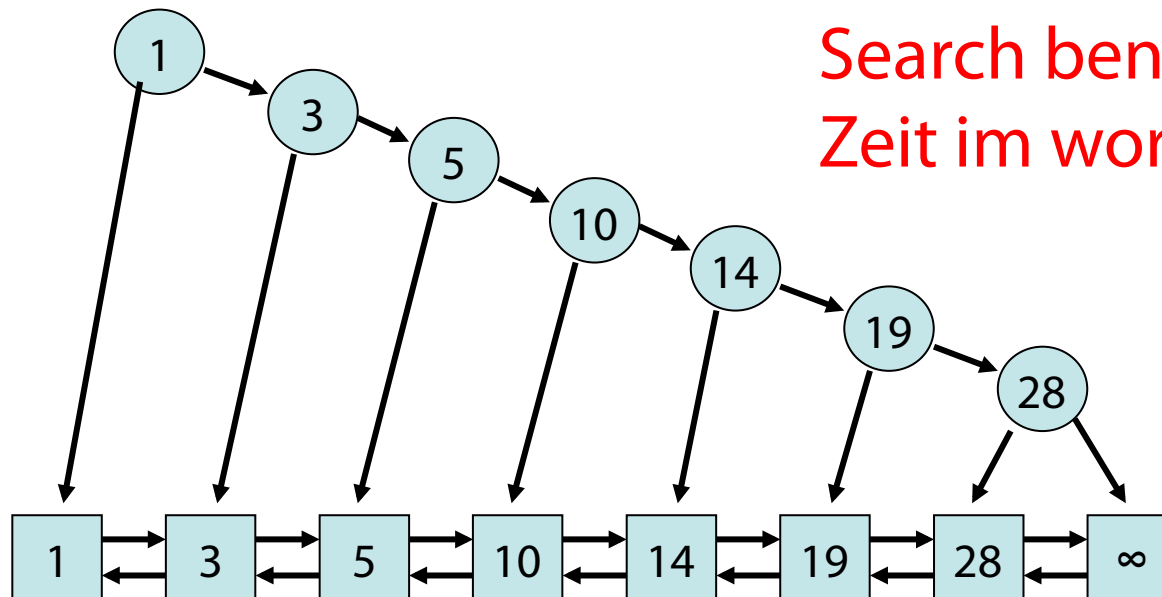
Delete(14)



Binärbaum

Problem: Binärbaum kann entarten!

Beispiel: Zahlen werden in sortierter Folge eingefügt



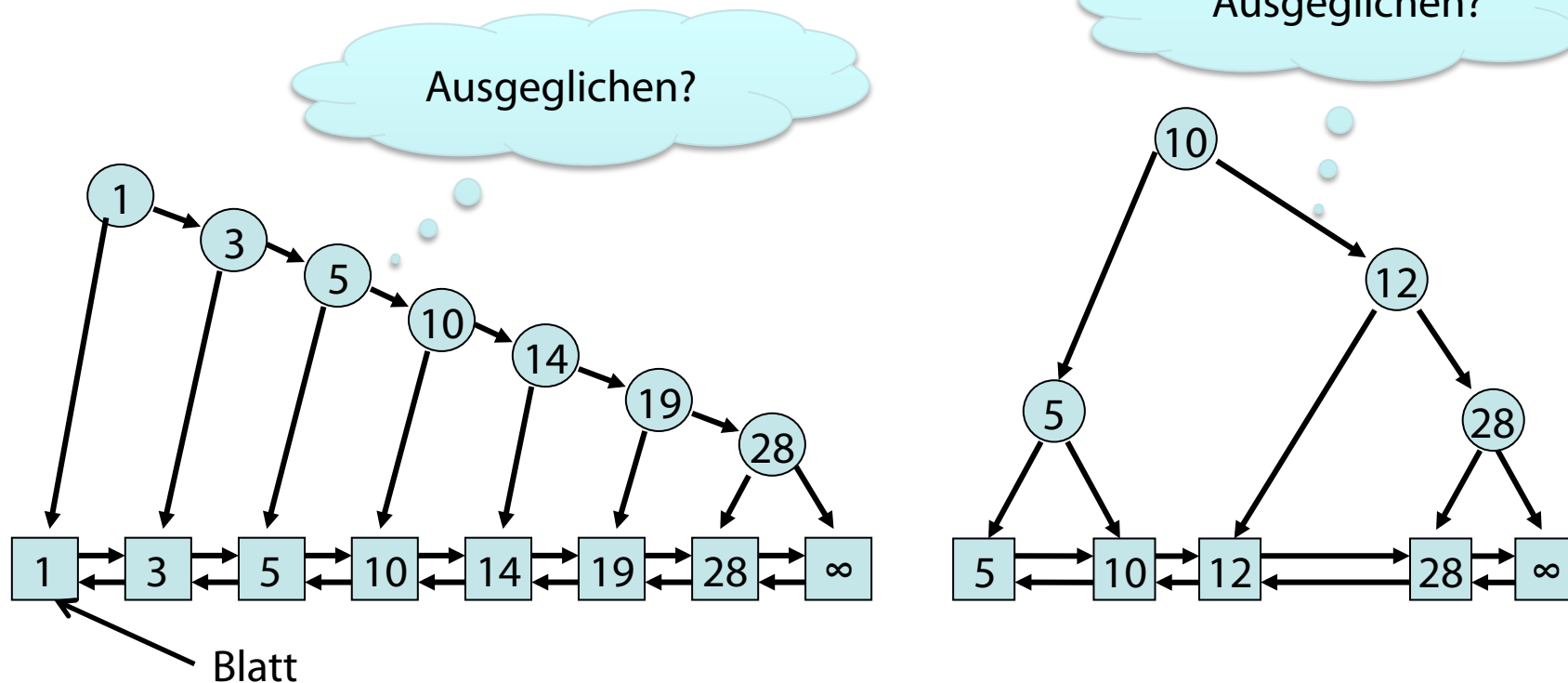
Search benötigt $\Theta(n)$
Zeit im worst case

Selbstanordnung?

- Anpassung beim Anfragen (Operation search/test)
- Balancierung beim Einfügen neuer Elemente

Definition: Ausgeglichener Suchbaum

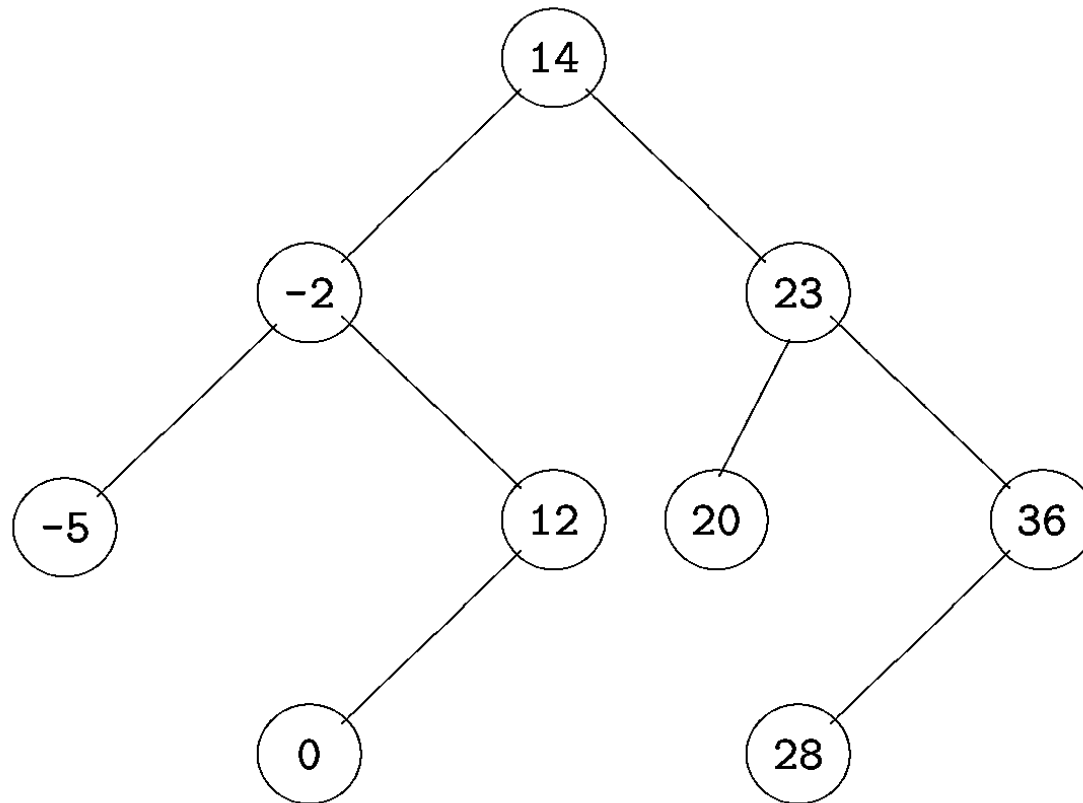
- Die Längen der Pfade von den Blättern zur Wurzel unterscheiden sich maximal um 1



- „Alle Ebenen bis auf Blattebene voll gefüllt“

Suchbäume mit Information in Knoten

- Gleiche Definition von Ausgeglichenheit bei „internen“ Suchbäumen

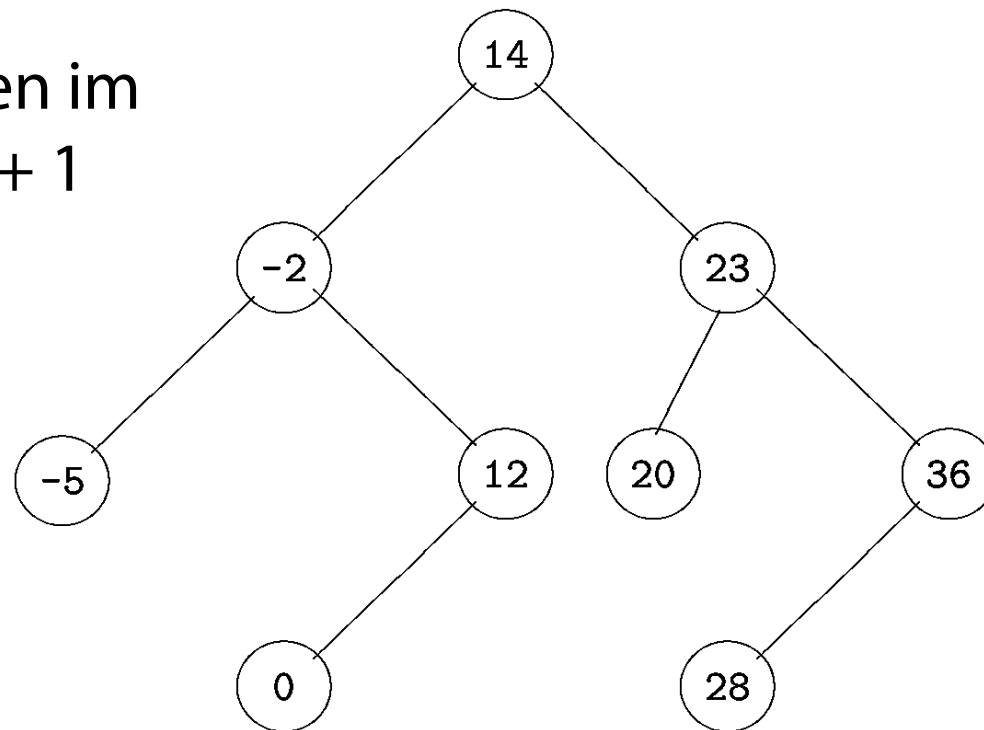


Gewichtete Binärbäume

- 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

Rang eines Knotens x im Binärbaum

- Position von x innerhalb der nach Schlüsseln sortierten Folge von Knoten
- Entspricht k -tem Knoten, der bei In-Order-Traversierung bearbeitet wird
- Anzahl der Knoten im linken Teilbaum + 1



Hier: Andere Definition für Rang

- Rang eines Knotens:
 - minimale Länge eines Pfades zu einem Blatt:

$$\text{Rang}(x) = \begin{cases} 0, & \text{falls } x \text{ Blatt ist} \\ 1 + \min(\text{Rang}(\text{left}(x)), \text{Rang}(\text{right}(x))) & \text{sonst} \end{cases}$$

Auch Level genannt

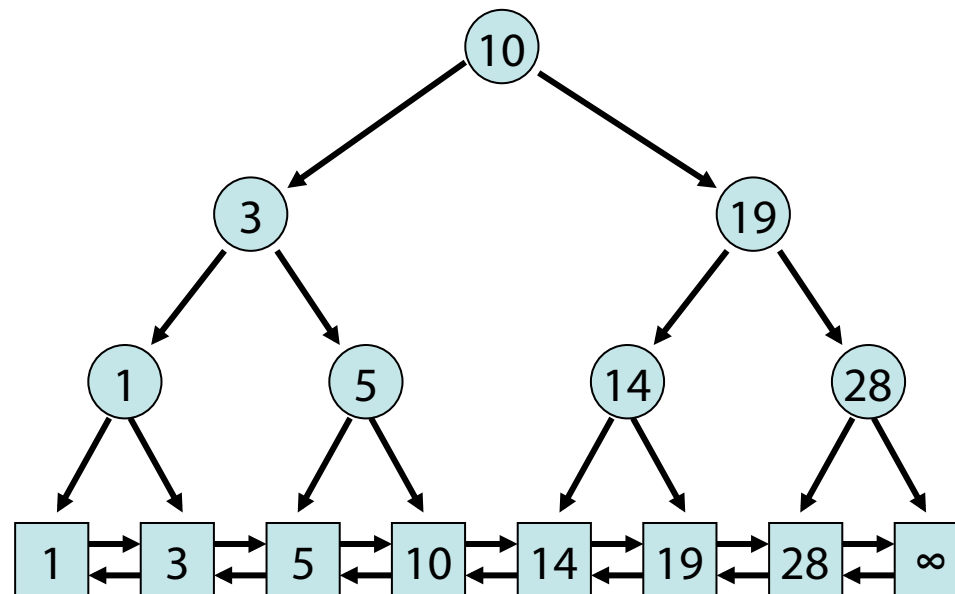
Bei einem ausgeglichenen Binärbaum:

$\text{Rang} = \lfloor \log(n) \rfloor$, wobei n die Anzahl der Knoten im Baum ist

Manchmal $\log(n)$ auch für nicht ausgeglichene Bäume als Rang der Wurzel verwendet (Abrundung weggelassen)

Ausgeglichener Baum mit n Knoten

- Annahme:
Relative Zugriffshäufigkeit für alle Schlüssel gleich
- Suchaufwand optimal: $\Theta(\log n)$, da Höhe $\lfloor \log n \rfloor$



- Suchaufwand durch Rang der Wurzel bestimmt

Selbstorganisierende Bäume

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

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>



Splay-Baum

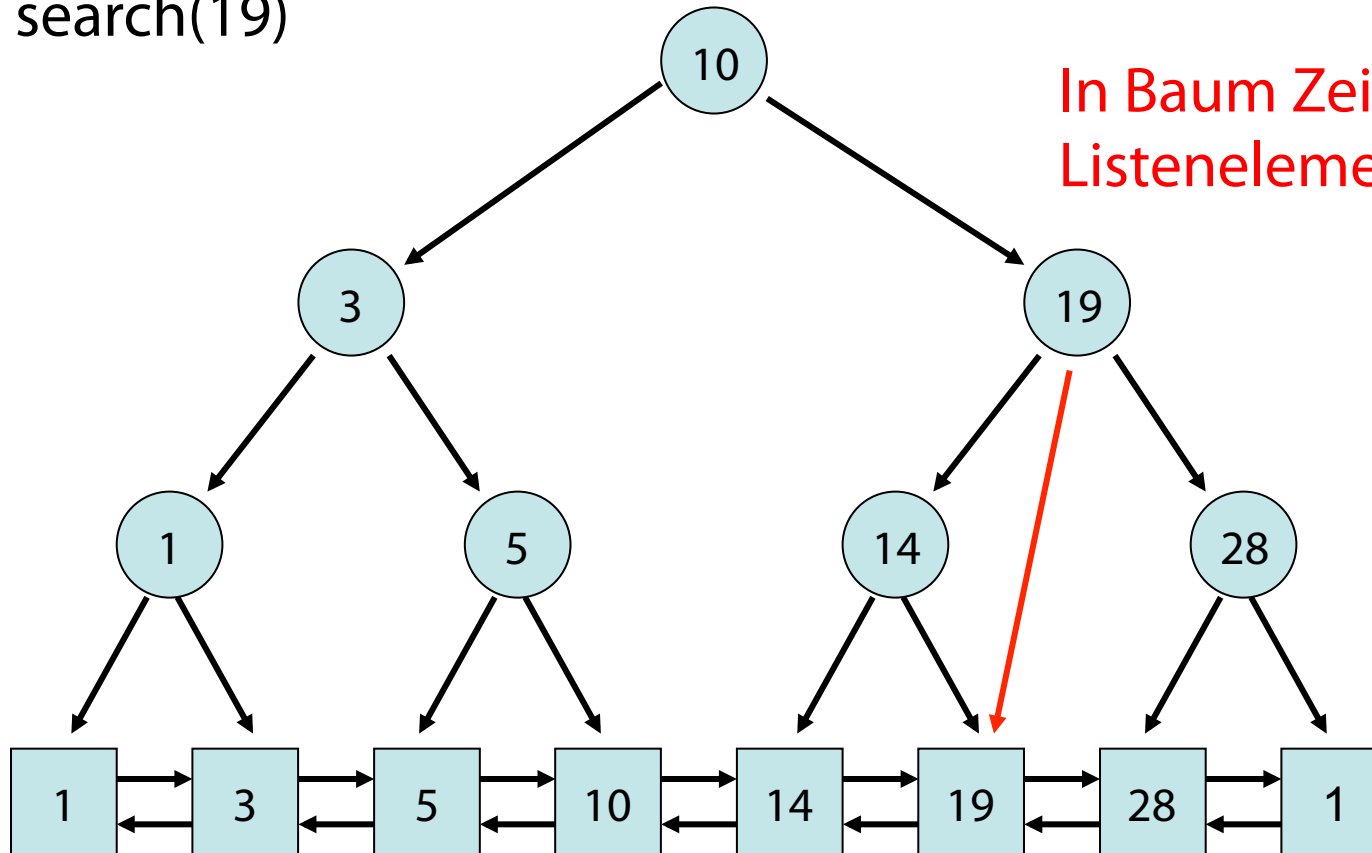
Üblicherweise: Implementierung als interner Suchbaum
(d.h. Elemente direkt integriert in Baum und nicht in
extra Liste)

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

Modifikation nicht nur bei Modifikationen, sondern auch
bei Anfragen

Splay-Baum

search(19)



Splay-Baum

Ideen:

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

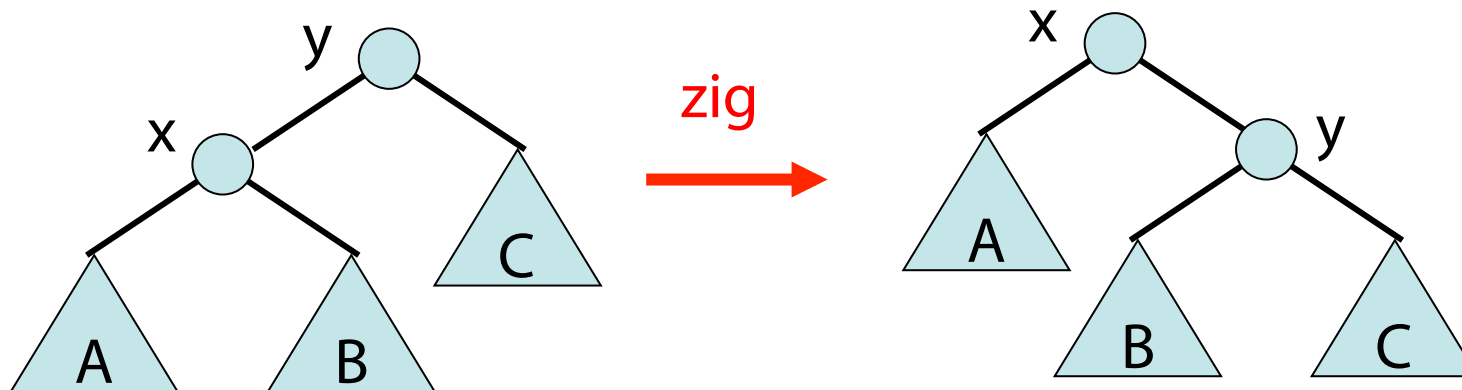
2. Idee: über **Splay-Operation**

Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

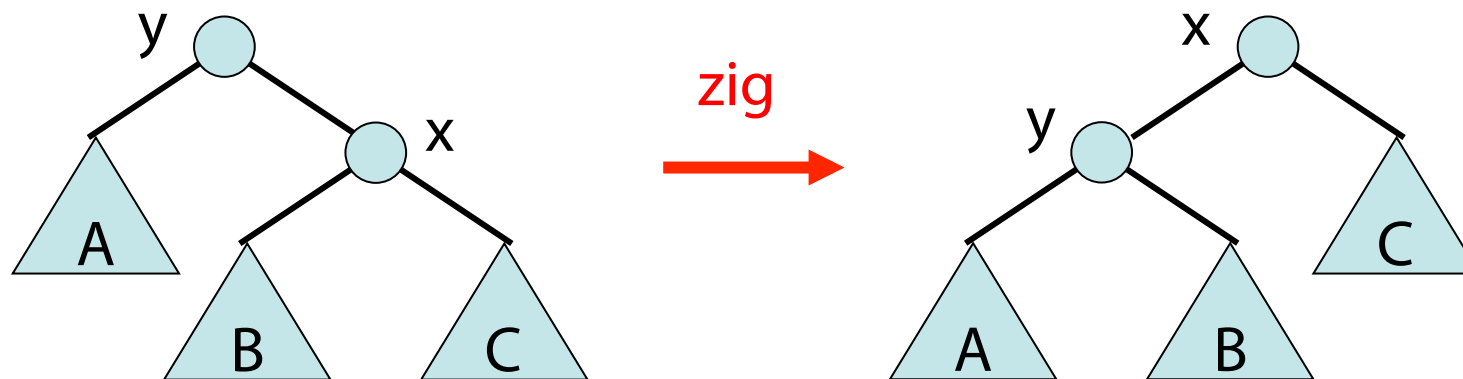


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

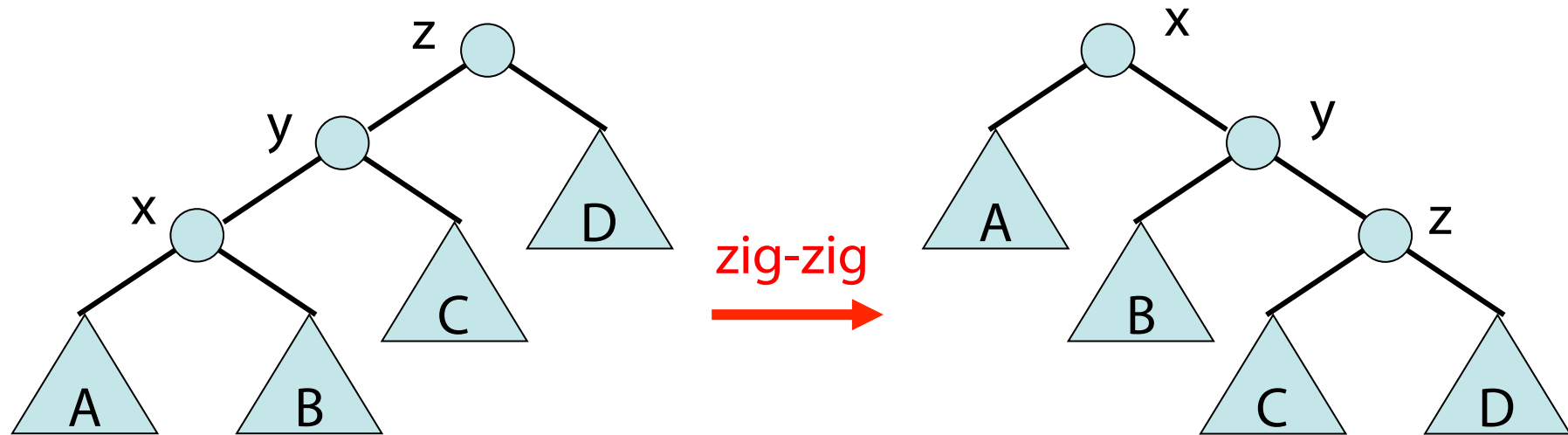
1b. x ist Kind der Wurzel:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

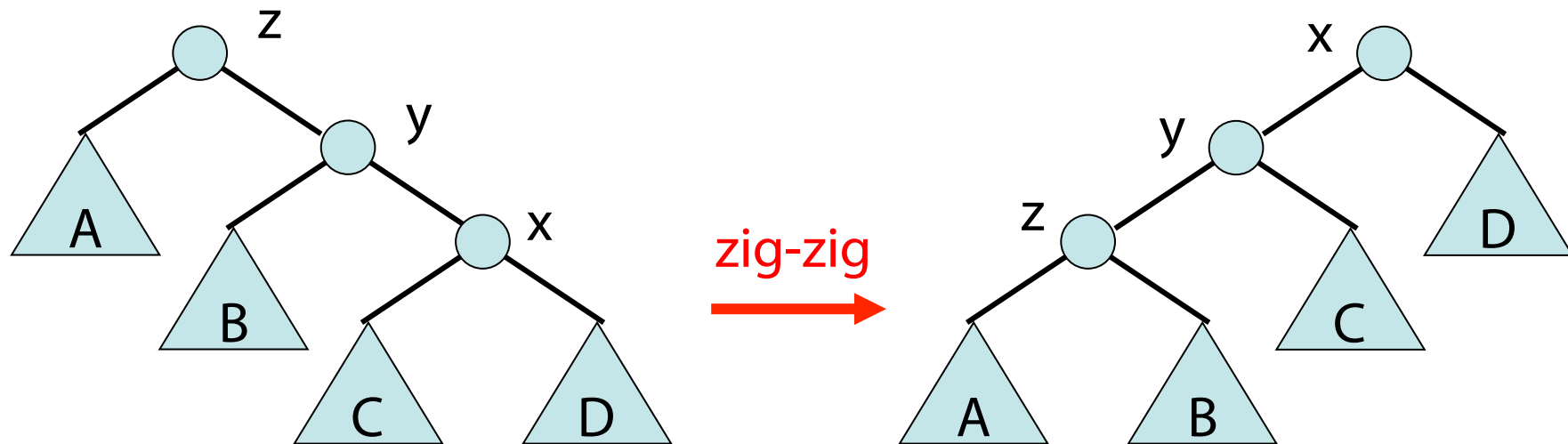
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

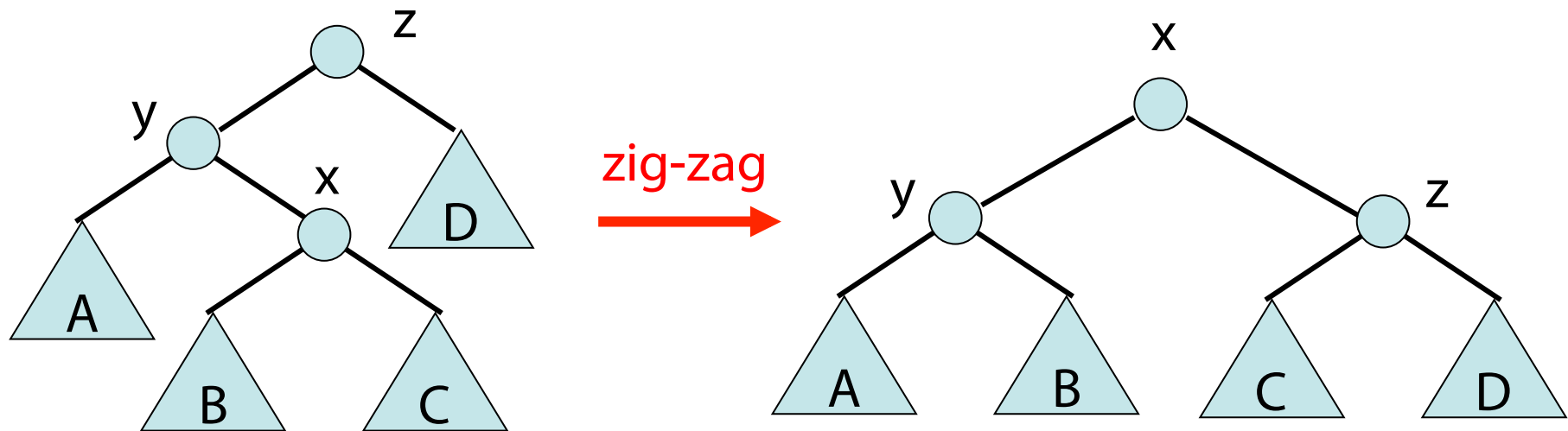
2b. x hat Vater und Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

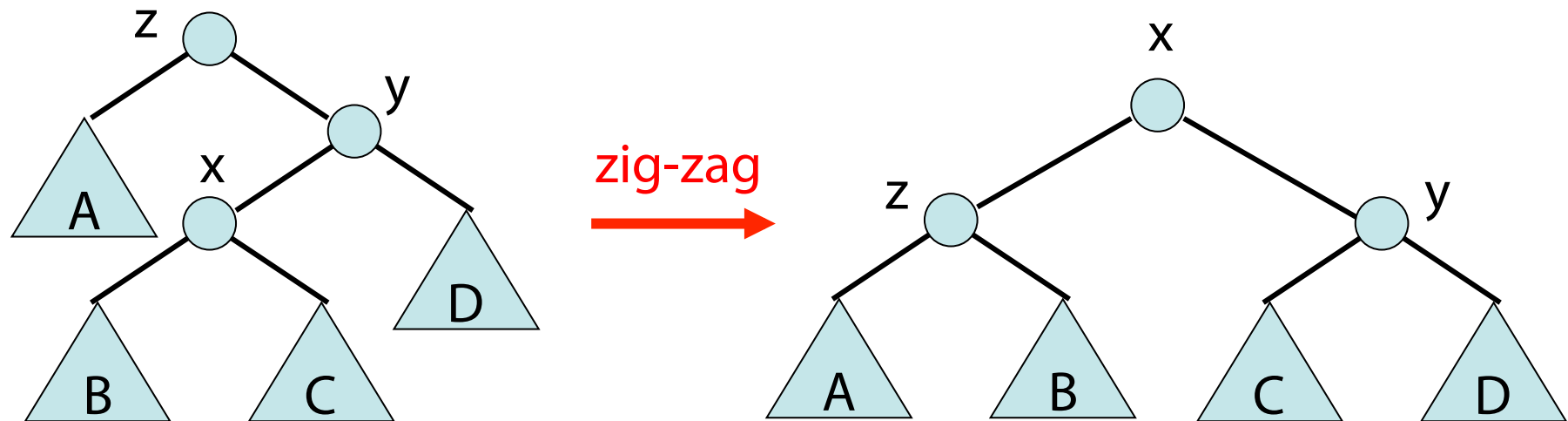
3a. x hat Vater links, Großvater rechts:



Splay-Operation

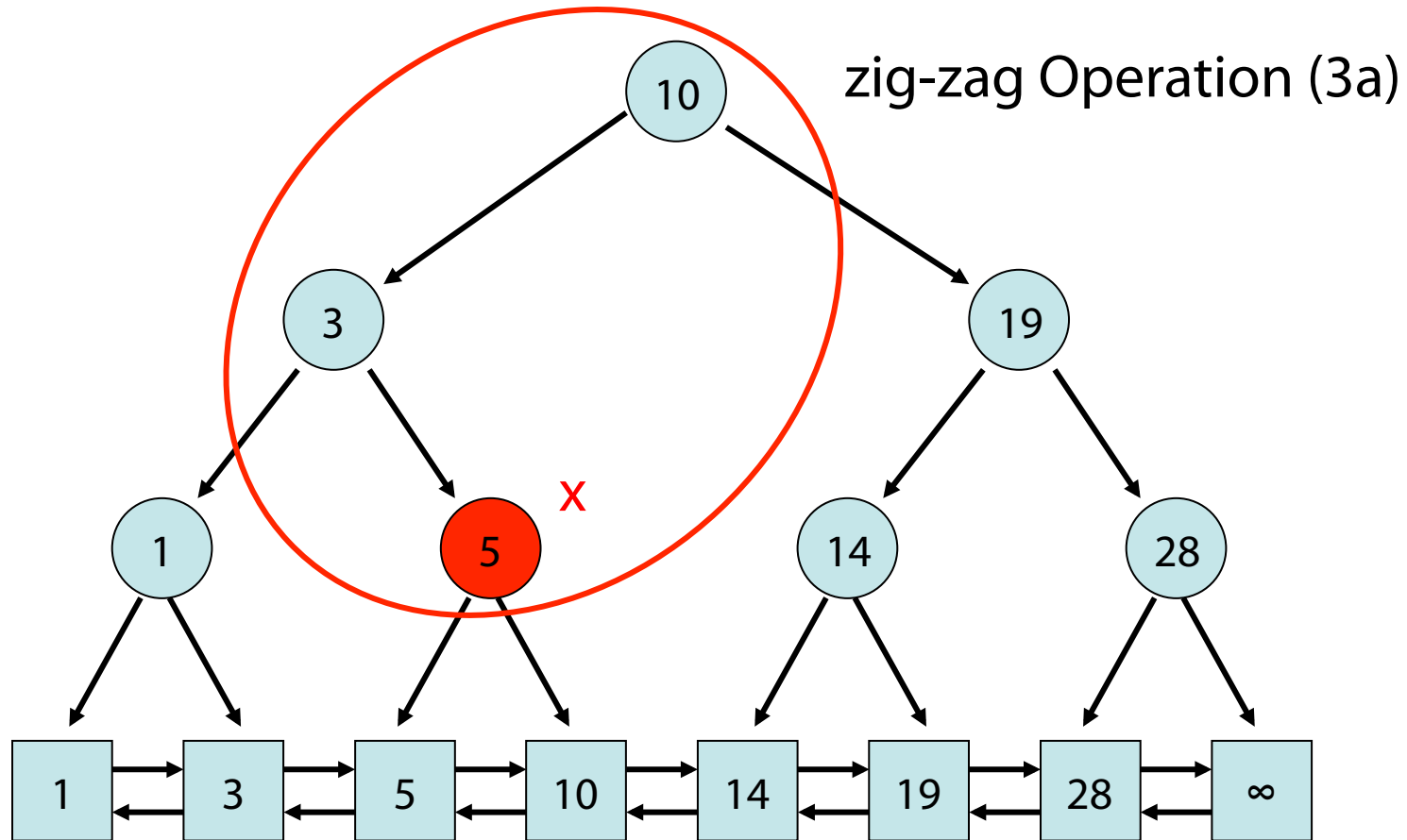
Wir unterscheiden zwischen 3 Fällen.

3b. x hat Vater rechts, Großvater links:

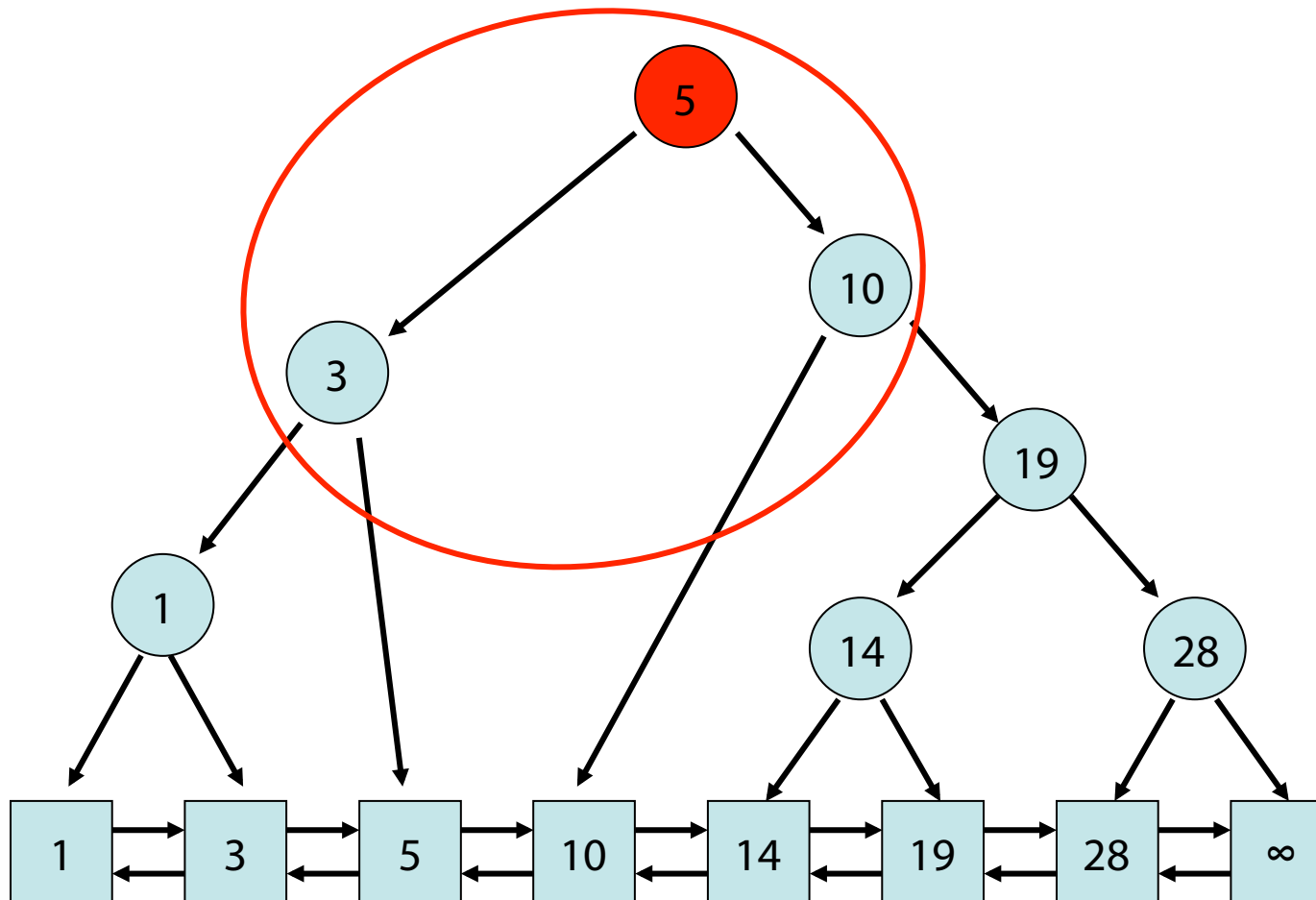


Splay-Operation

Beispiel:

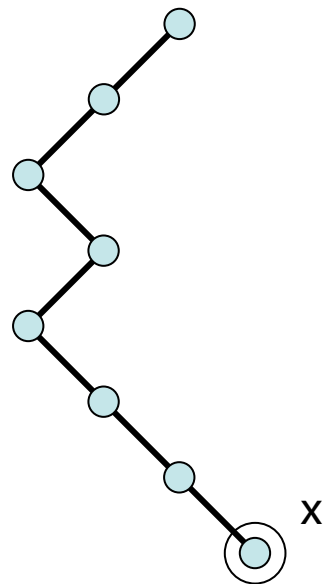


Splay-Operation

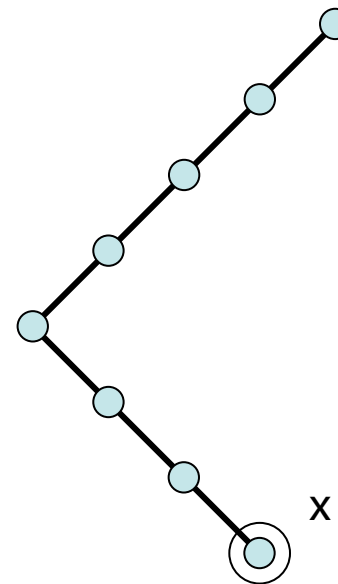


Splay-Operation

Beispiele:



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



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

Splay

procedure splay(x, T)

while parentExists(x, T) **do**

// x wandert hoch

Wende geeignete Splay-Operation an

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 ($m > n$)

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

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

Splay-Operation

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

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

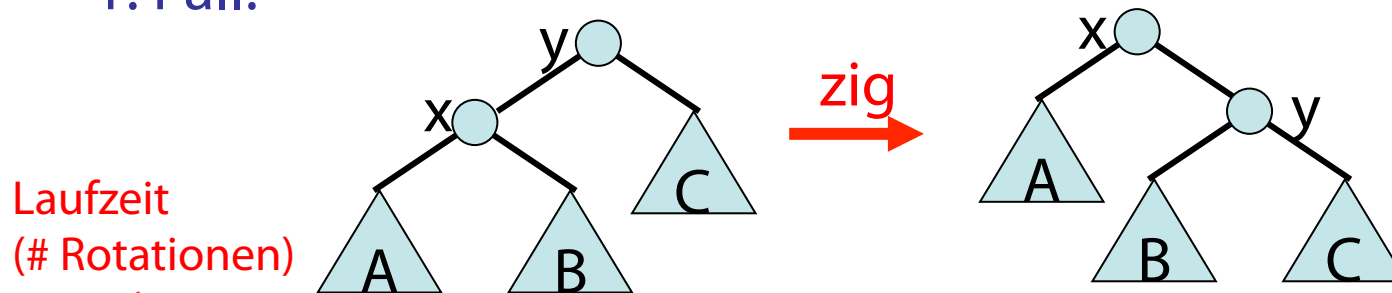
Splay-Operation

Beweis:

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:

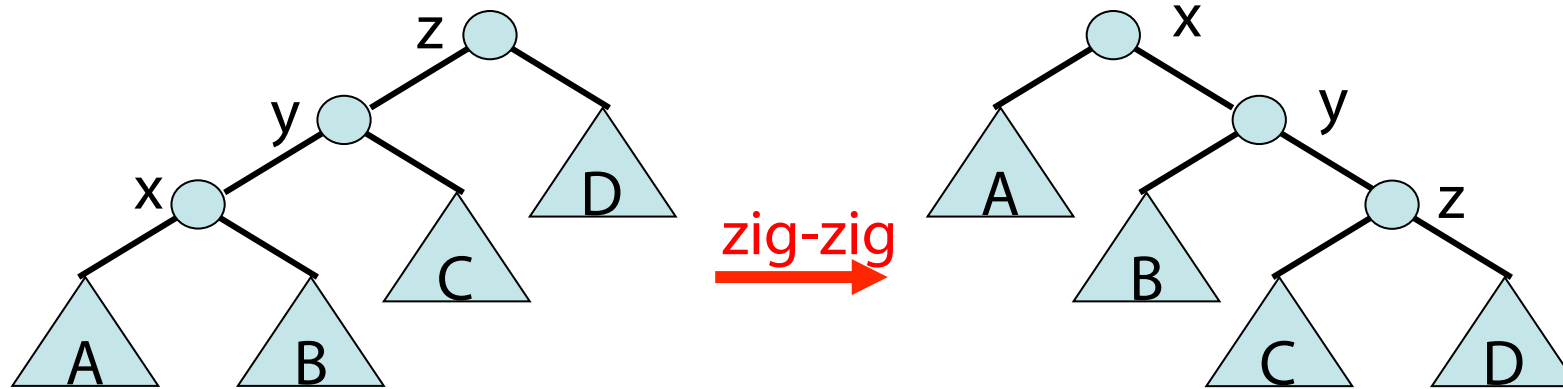
$$\leq 1 + r'(x) + r'(y) - r(x) - r(y) \leq 1 + r'(x) - r(x) \quad \text{da } r'(y) \leq r(y)$$

$$\leq 1 + 3(r'(x) - r(x)) \quad \text{da } r'(x) \geq r(x)$$

Änderung von ϕ

Splay-Operation

2. Fall:



Amortisierte Kosten:

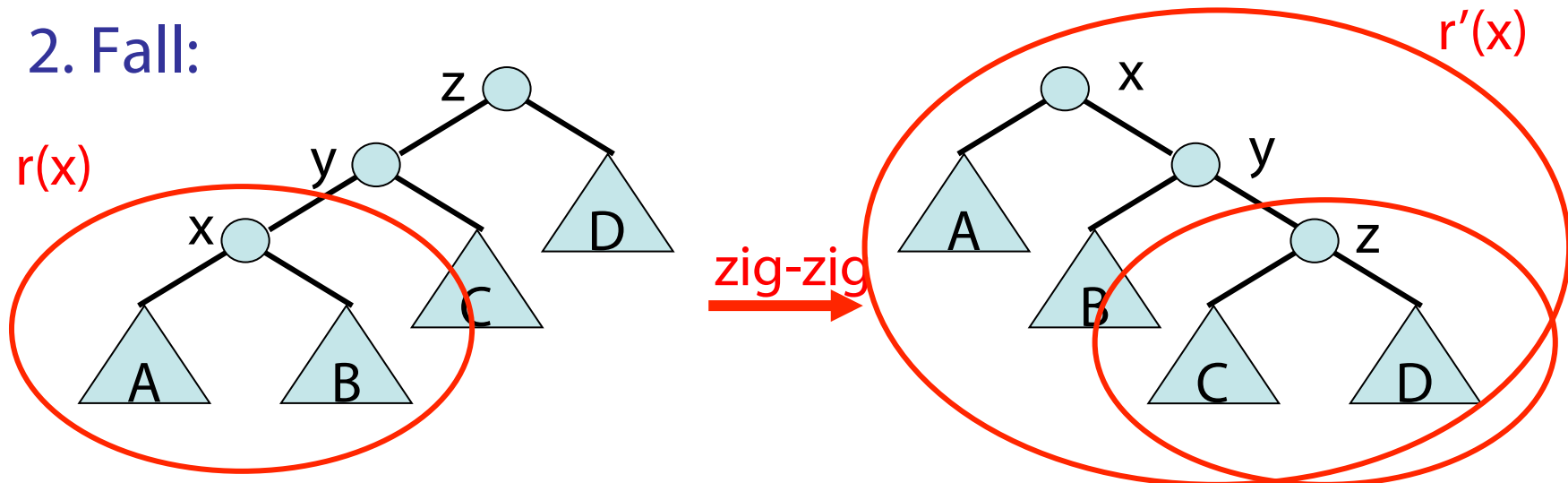
$$\leq 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$

$$= 2 + r'(y) + r'(z) - r(x) - r(y) \quad \text{da } r'(x) = r(z)$$

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

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

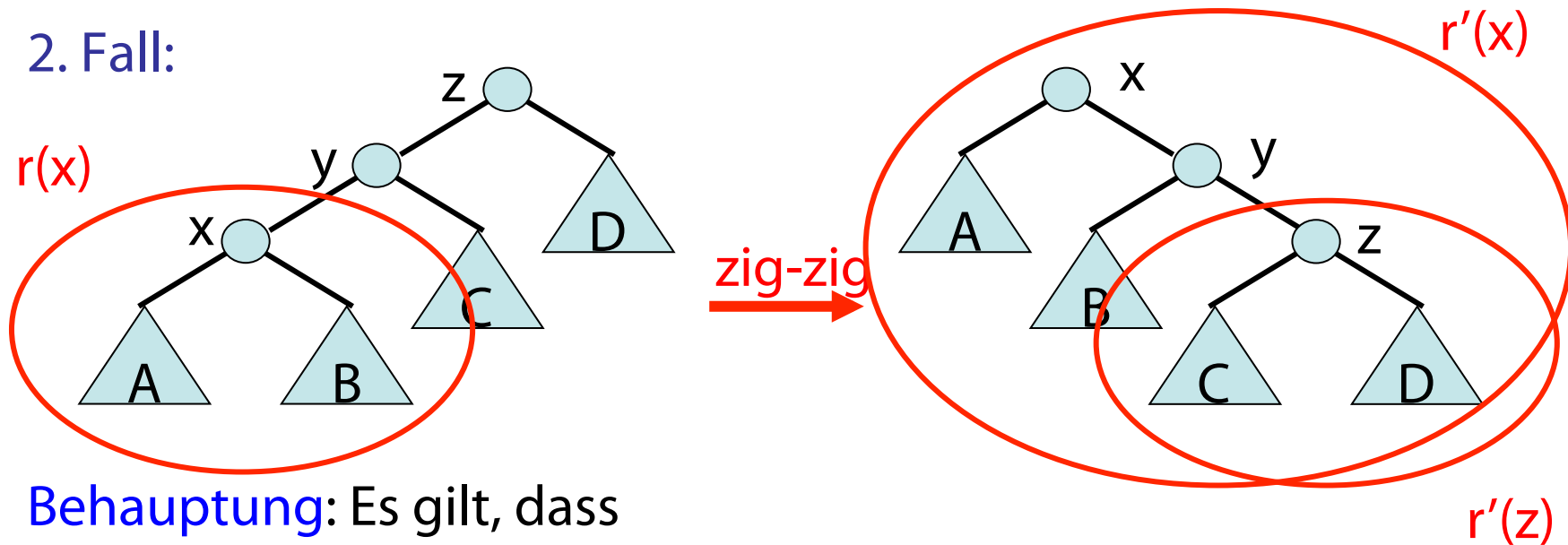
$$2+r'(x)+r'(z)-2r(x) \leq 3(r'(x)-r(x))$$

d.h.

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

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

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

Ersetzungen: $r(x) \rightarrow \log x$, $r'(z) \rightarrow \log y$, $r'(x) \rightarrow \log 1$.

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

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

Splay-Operation

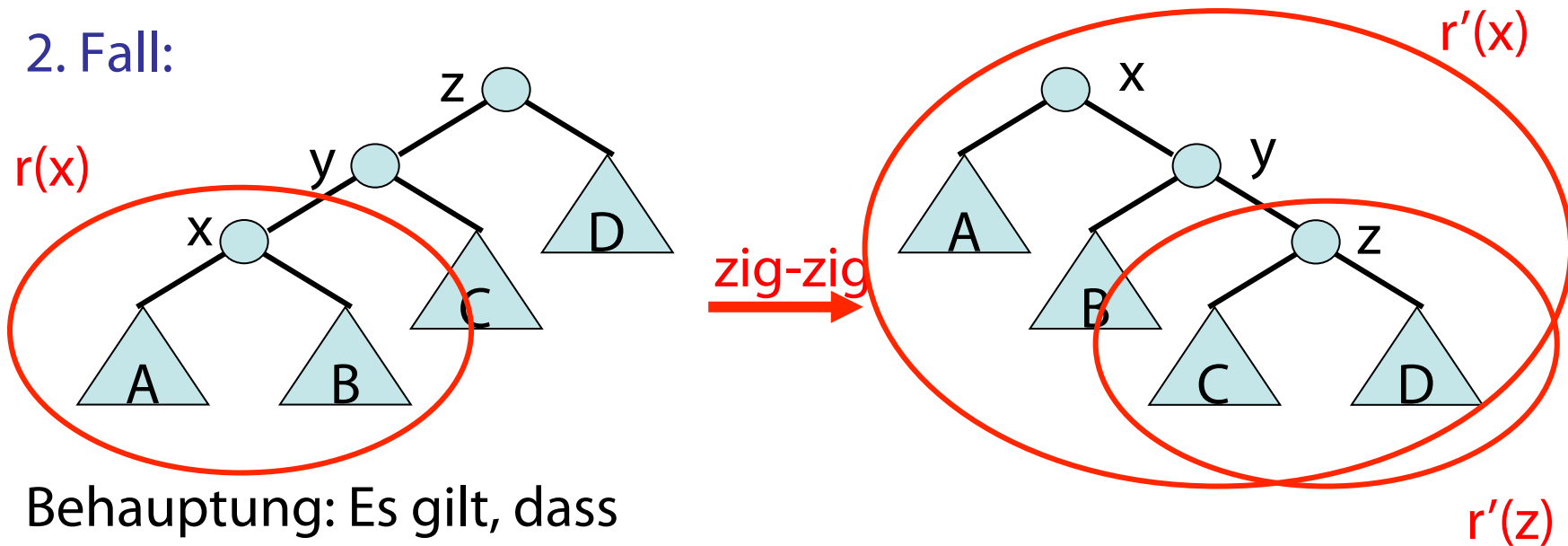
Behauptung: Die Funktion $f(x,y)=\log x + \log y$ hat in dem Bereich $x,y>0$ mit $x+y\leq 1$ im Punkt $(\frac{1}{2},\frac{1}{2})$ ihr Maximum.

Beweis:

- 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 $(\frac{1}{2},\frac{1}{2})$ ihr Maximum.

Splay-Operation

2. Fall:



Behauptung: Es gilt, dass

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

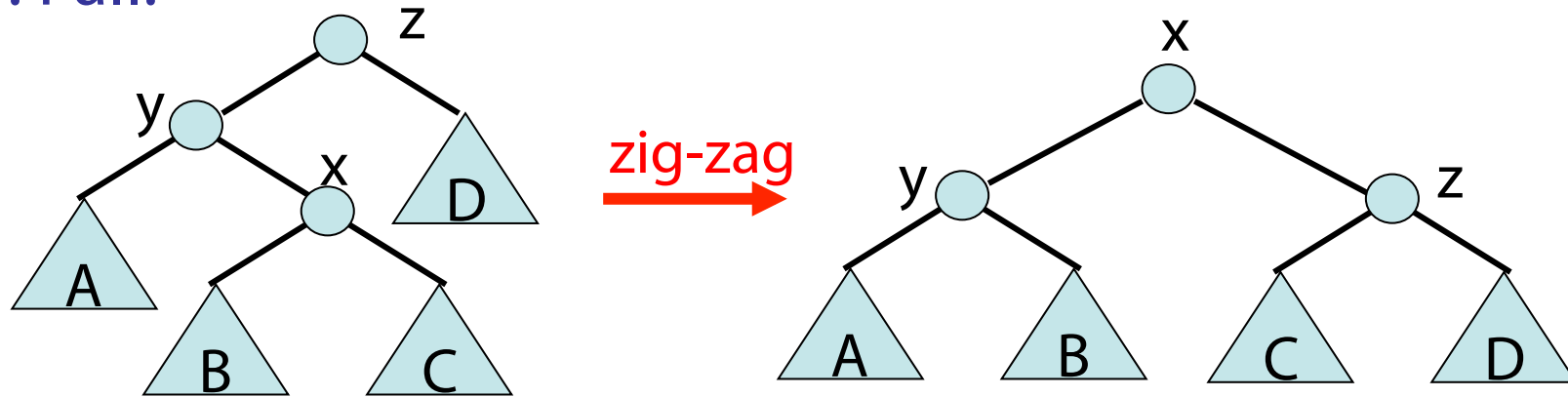
Ersetzungen: $r(x) \rightarrow \log x$, $r'(z) \rightarrow \log y$, $r'(x) \rightarrow \log 1$.

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

Die Behauptung ist also korrekt.

Splay-Operation

3. Fall:



Amortisierte Kosten:

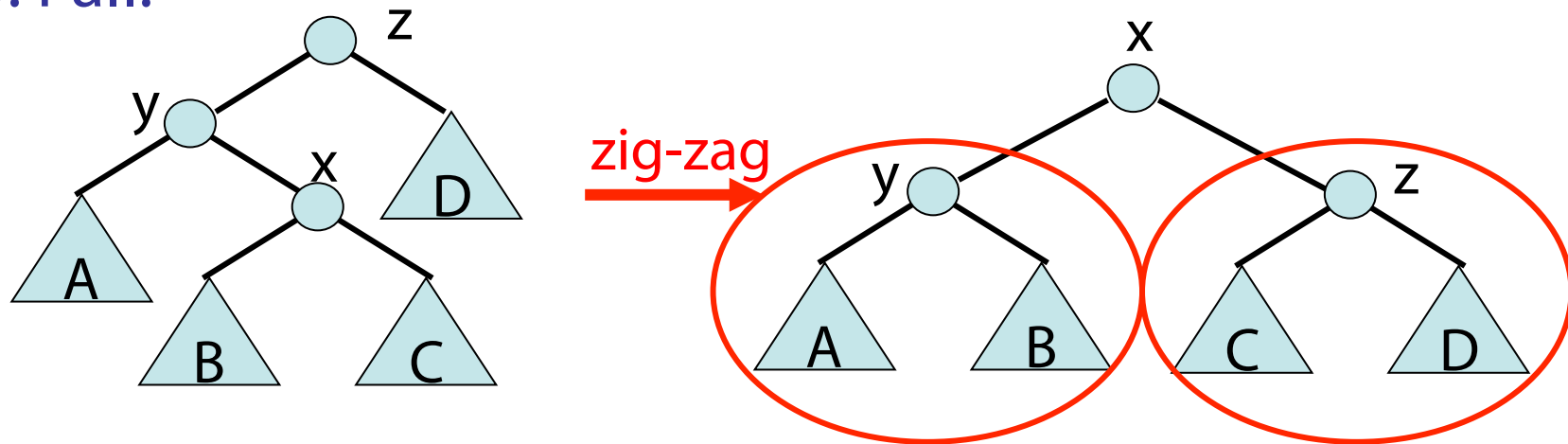
$$\leq 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$

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

$$\leq 2(r'(x) - r(x)) \quad \text{denn...}$$

Splay-Operation

3. Fall:



...es gilt:

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

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

$$\Leftrightarrow r'(y)+r'(z) \leq 2(r'(x)-1) \text{ analog zu Fall 2}$$

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)
- Aufsummierung der Kosten ergibt max.
 $1 + \sum_{\text{Rot.}} 3(r'(x)-r(x)) = 1+3(r(u)-r(x))$

Splay-Operation

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

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

Splay-Baum: Balance-Theorem

“Balance Theorem”: Die Laufzeit für m search Operationen in einem n -elementigen Splay-Baum T ist höchstens

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

Beweis:

- Sei $w(x) = 1$ für alle Schlüssel x in T .
- Dann ist $W=n$ und $r(x) \leq \log W = \log n$ 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
- $\phi(s_0) = \sum_{x \in T} r_0(x) \leq n \log n$
- Aus dem Korollar von oben ergibt sich das Theorem

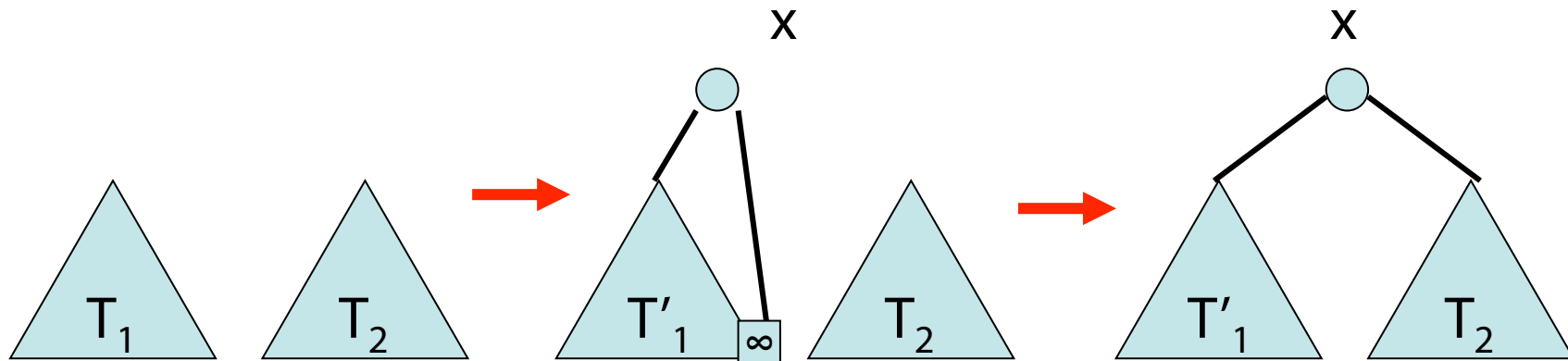
Deutung: Splay-Bäume arbeiten so effektiv wie ausgeglichene Bäume bei Search-Sequenzen länger als n



Splay-Baum Operationen

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

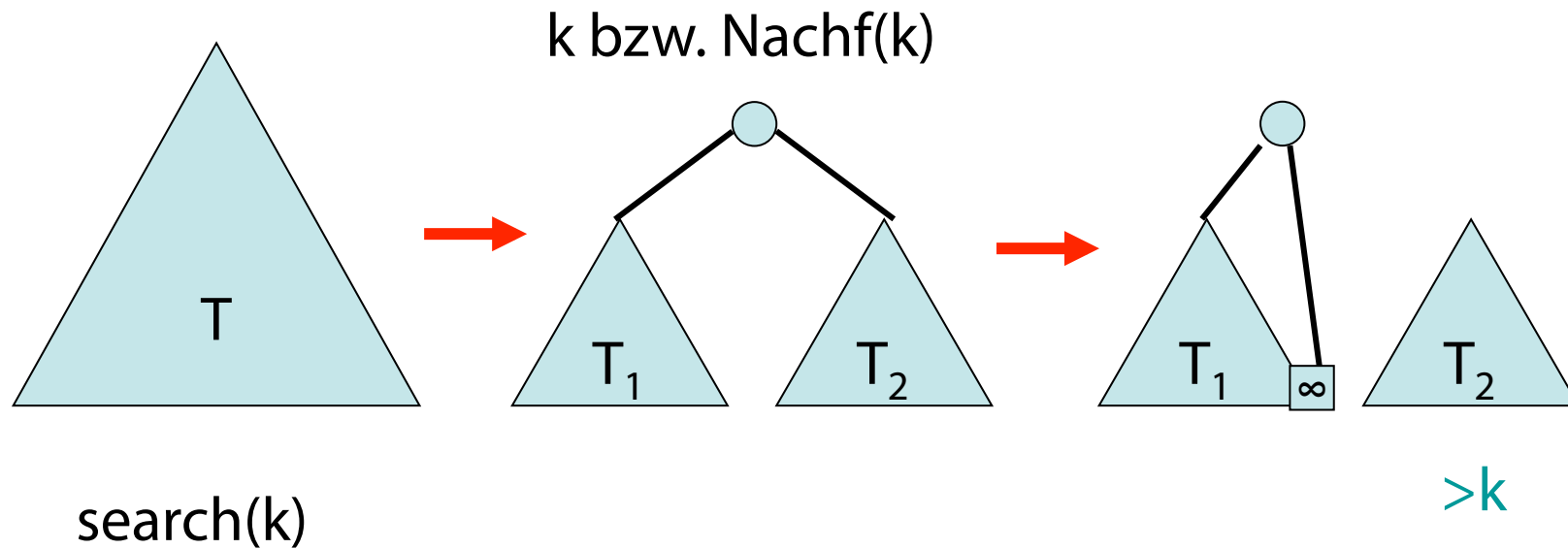
merge(T_1, T_2):



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

Splay-Bäume: Zusammenfassung

- k_- : größter Schlüssel in T kleiner als k
- k_+ : kleinster Schlüssel in T größer gleich k

Amortisierte Kosten der Operationen im Splay-Baum:

- $\text{search}(k)$: $O(1 + \log(W/w(k_+)))$
- $\text{split}(k)$: $O(1 + \log(W/w(k_+)))$
- $\text{merge}(T_1, T_2)$: $O(1 + \log(W/w(k_{\max}(T_1))))$
- $\text{insert}(e)$: $O(1 + \log(W/w(\text{key}(e))))$
- $\text{delete}(k)$: $O(1 + \log(W/w(k_+)) + \log(W/w(k_-)))$

Binärbaum

Problem: Binärbaum kann entarten!

Lösungen:

- **Splay-Baum**
(sehr effektive Heuristik)
- **Rot-Schwarz-Baum**
(konstanter Reorganisationsaufwand)

Rot-Schwarz-Baum

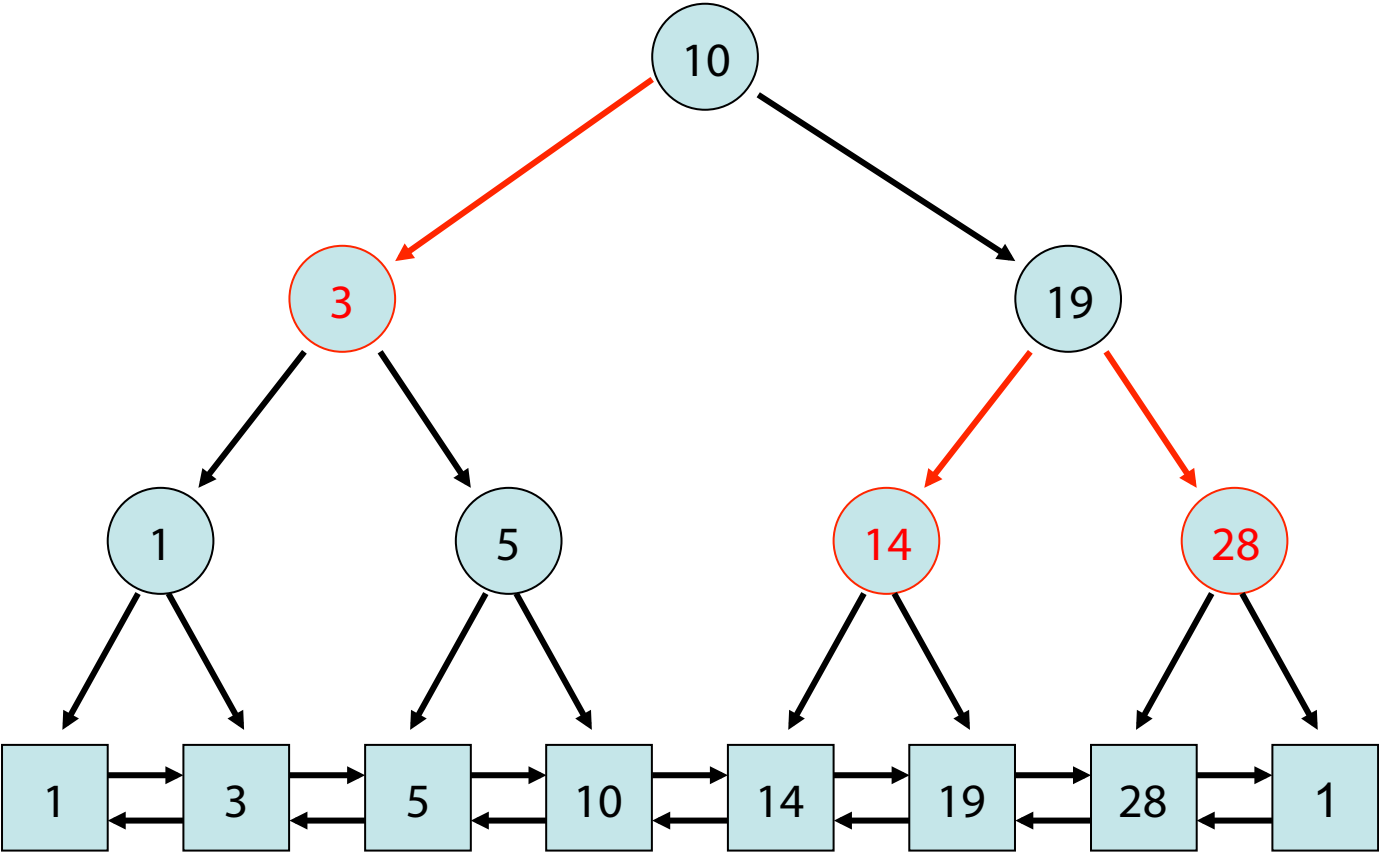
Rot-Schwarz-Bäume sind binäre Suchbäume mit roten und schwarzen Knoten, so dass gilt:

- **Wurzelregel:** Die Wurzel ist schwarz.
- **Externe Regel:** Jeder Listenknoten ist schwarz.
- **Interne Regel:** Die Kinder eines roten Knotens sind schwarz.
- **Tiefenregel:** Alle Listenknoten haben dieselbe "Schwarztiefe"

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

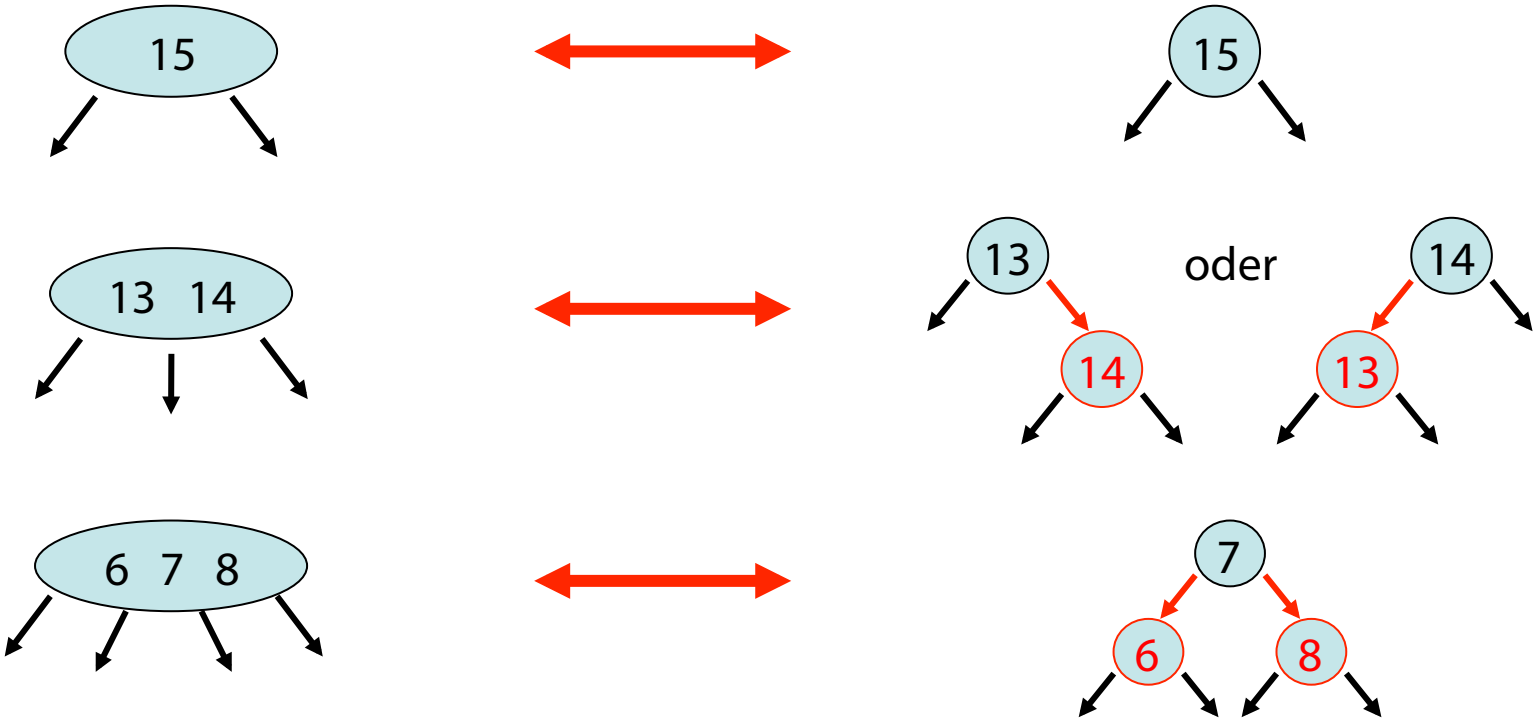
Rot-Schwarz-Baum

Beispiel:



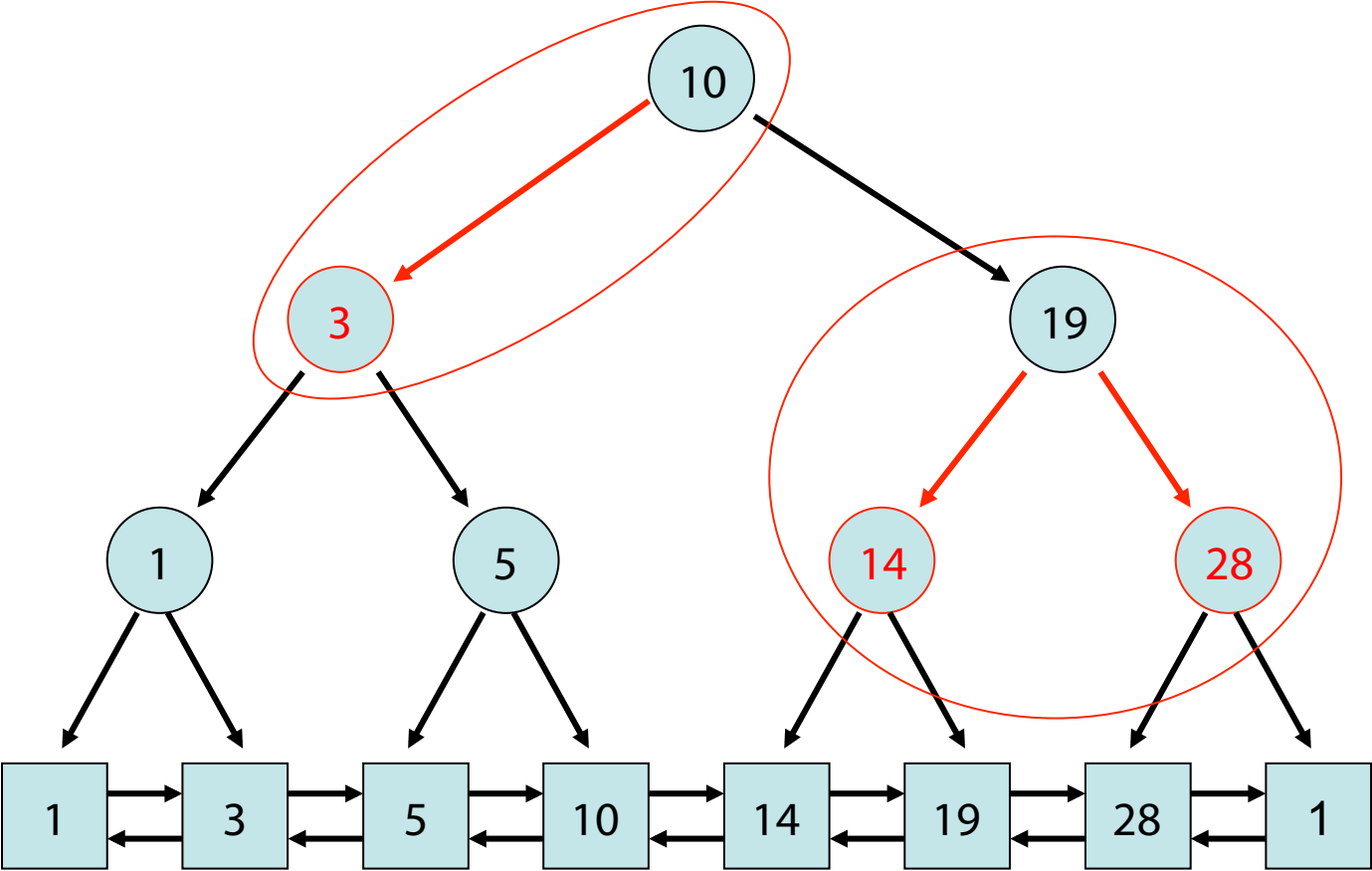
Rot-Schwarz-Baum

Andere Deutung von Rot-schwarz-Mustern:
 B-Baum (Baum mit "Separatoren" und min 2 max 4 Nachfolger)



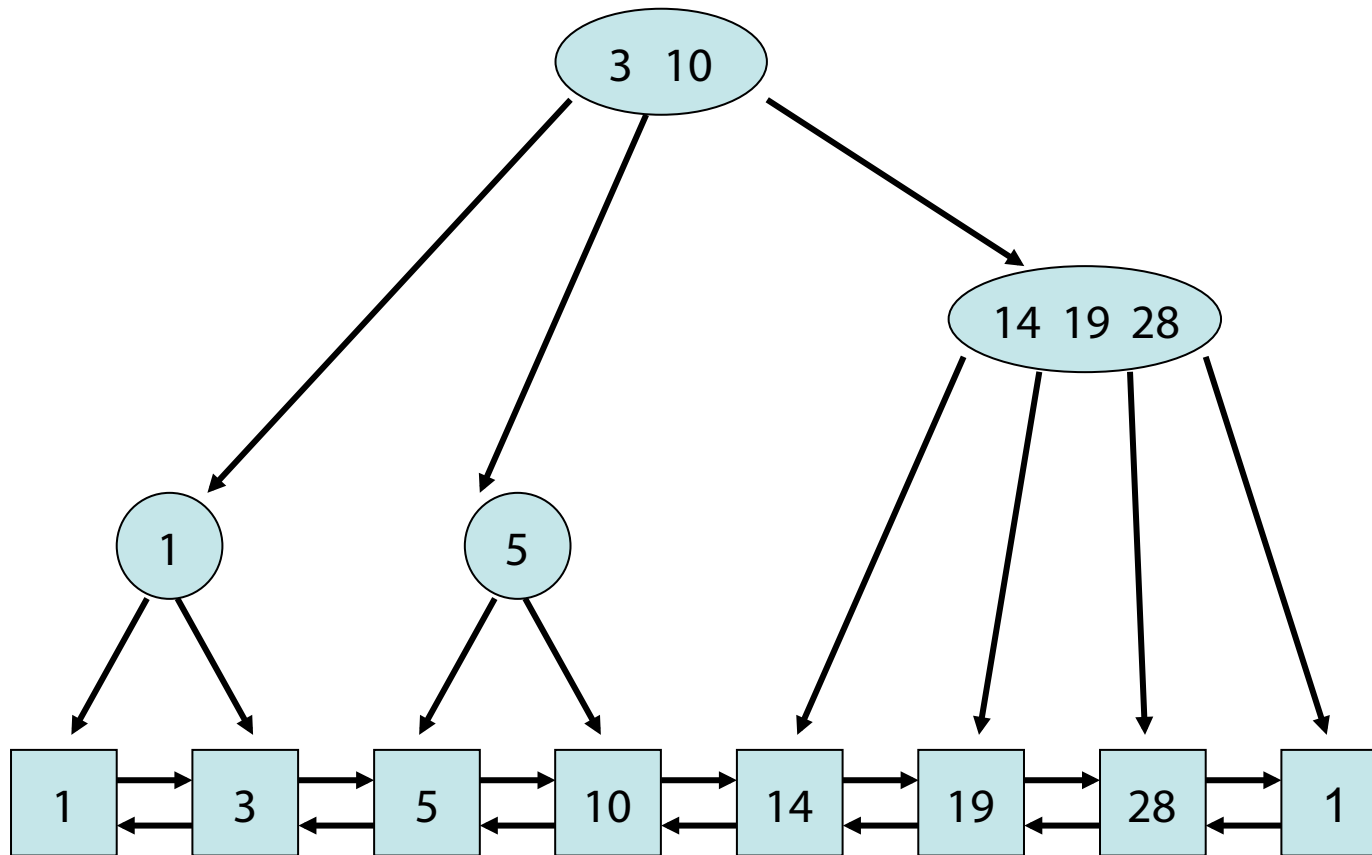
Rot-Schwarz-Baum

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 gilt $t \leq 2d$
- Außerdem ist $t \geq \log(n+1)$, da Rot-Schwarz-Baum ein Binärbaum ist.

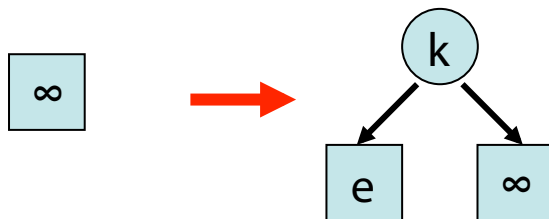
Rot-Schwarz-Baum

search(k): wie im binären Suchbaum

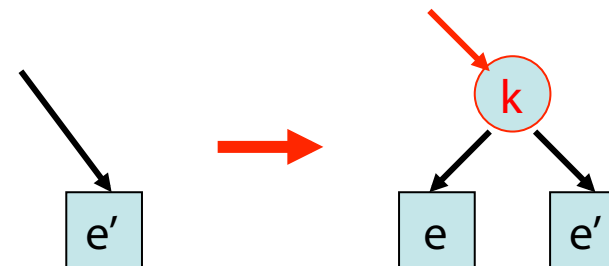
insert(e):

- 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



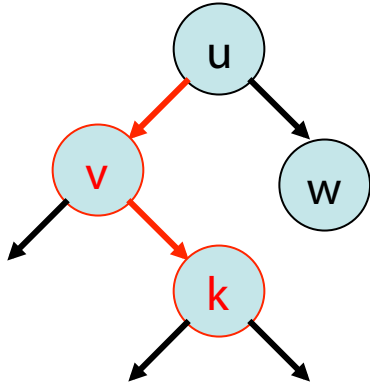
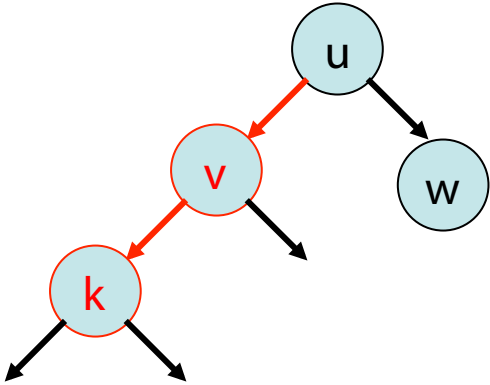
Rot-Schwarz-Baum

insert(e):

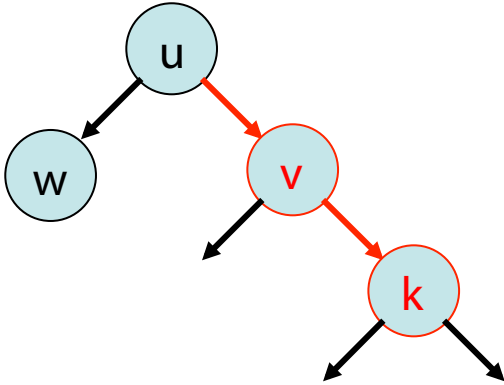
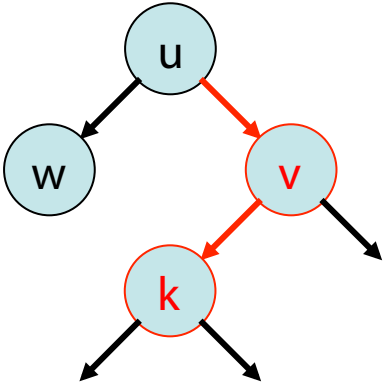
- Führe `search(k)` mit `k=key(e)` aus
- Füge `e` vor Nachfolger `e'` in Liste ein
(bewahrt alles bis auf evtl. interne Regel)
- Interne Regel verletzt (Fall 2 vorher): 2 Fälle
 - Fall 1: Vater von `k` in `T` hat schwarzen Bruder
(Restrukturierung, aber beendet Reparatur)
 - Fall 2: Vater von `k` in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



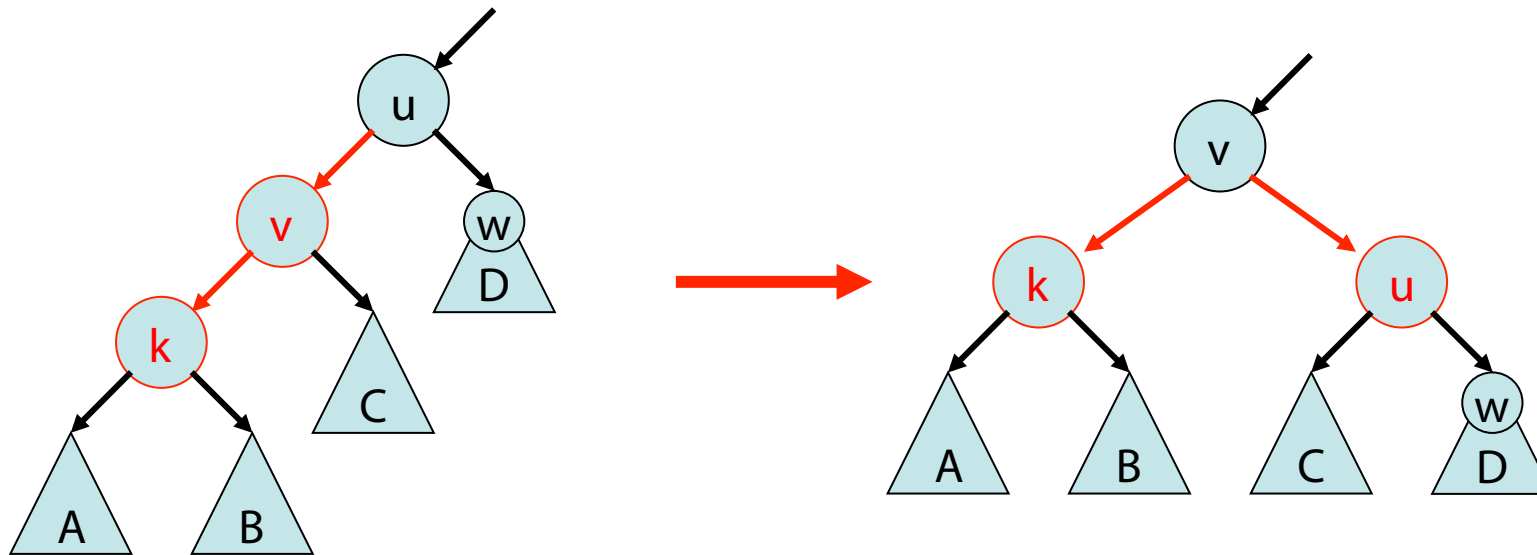
Alternativen
für Fall 1



Rot-Schwarz-Baum

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

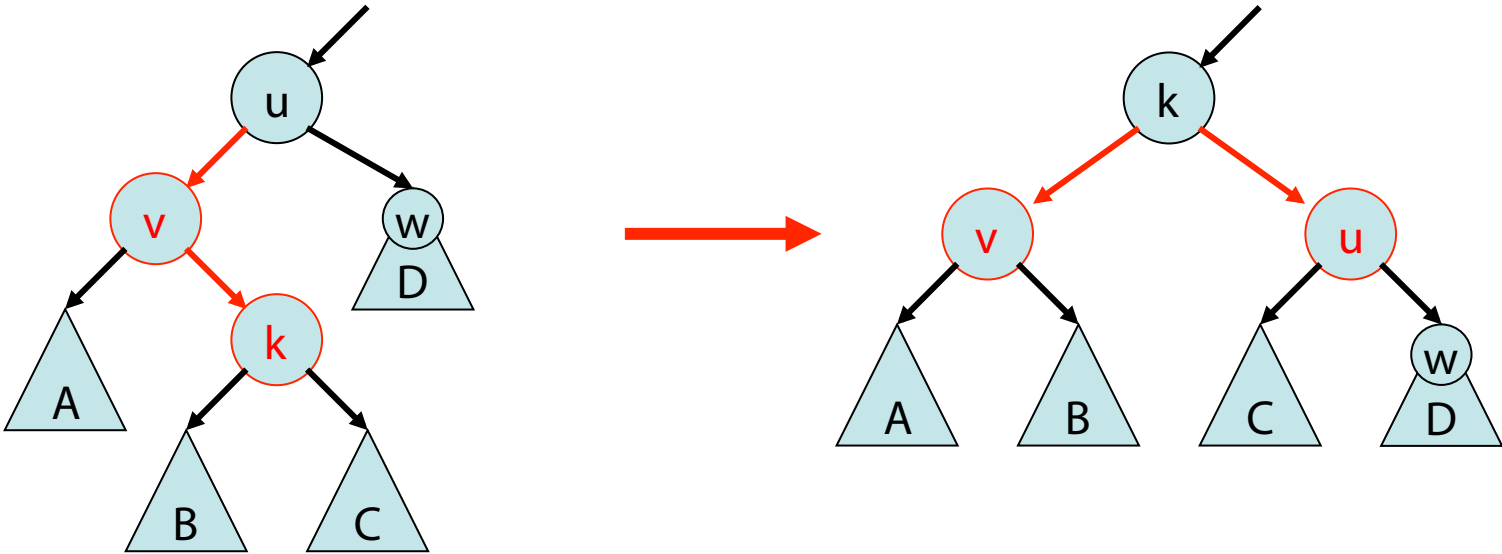
Lösung:



Rot-Schwarz-Baum

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

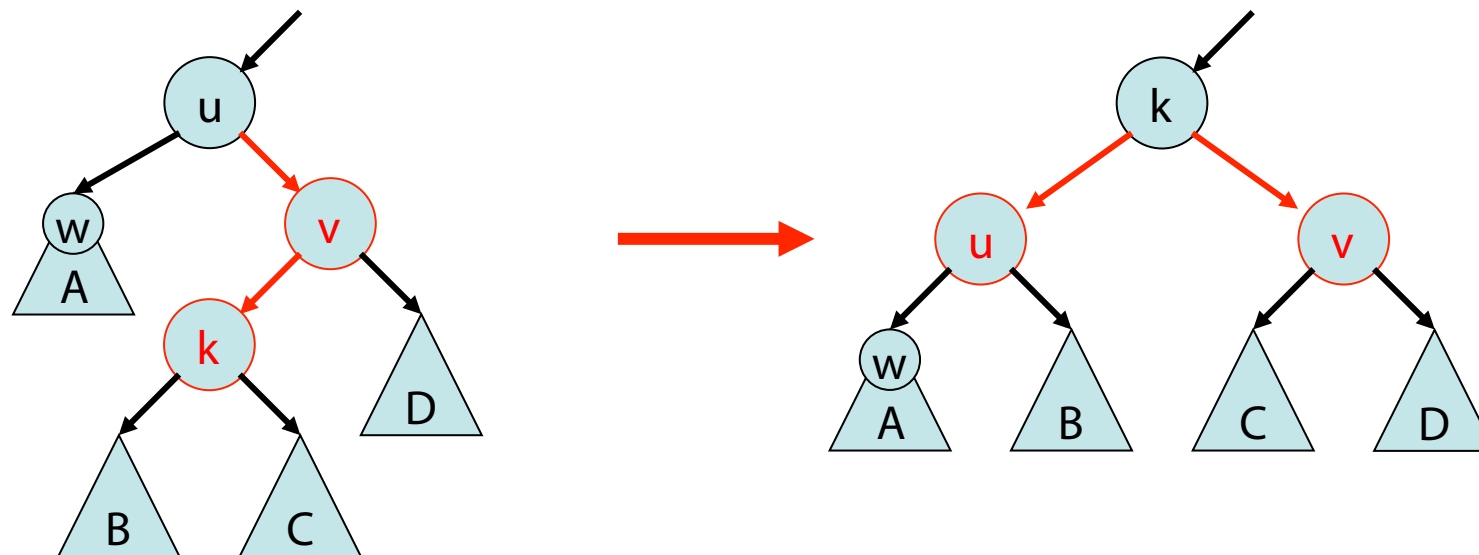
Lösung:



Rot-Schwarz-Baum

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

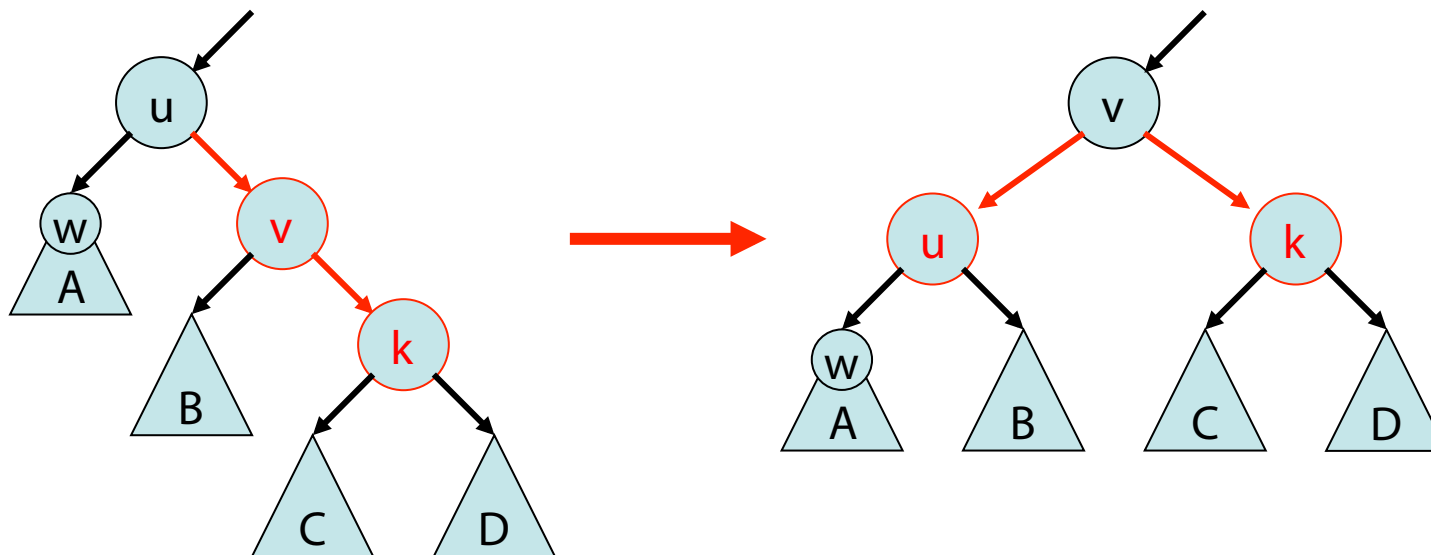
Lösung:



Rot-Schwarz-Baum

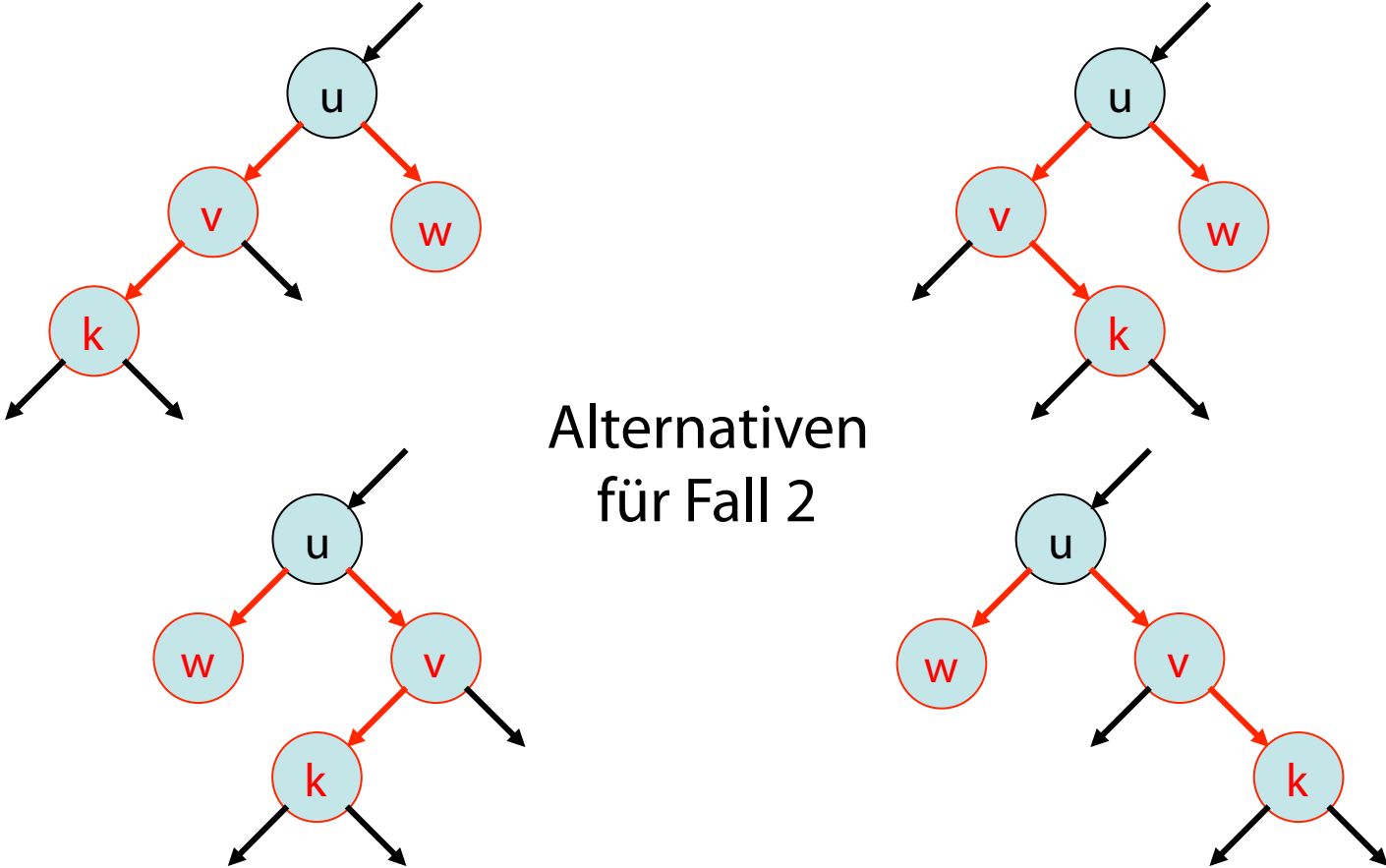
Fall 1: Vater v von k in T hat schwarzen Bruder w

Lösung:



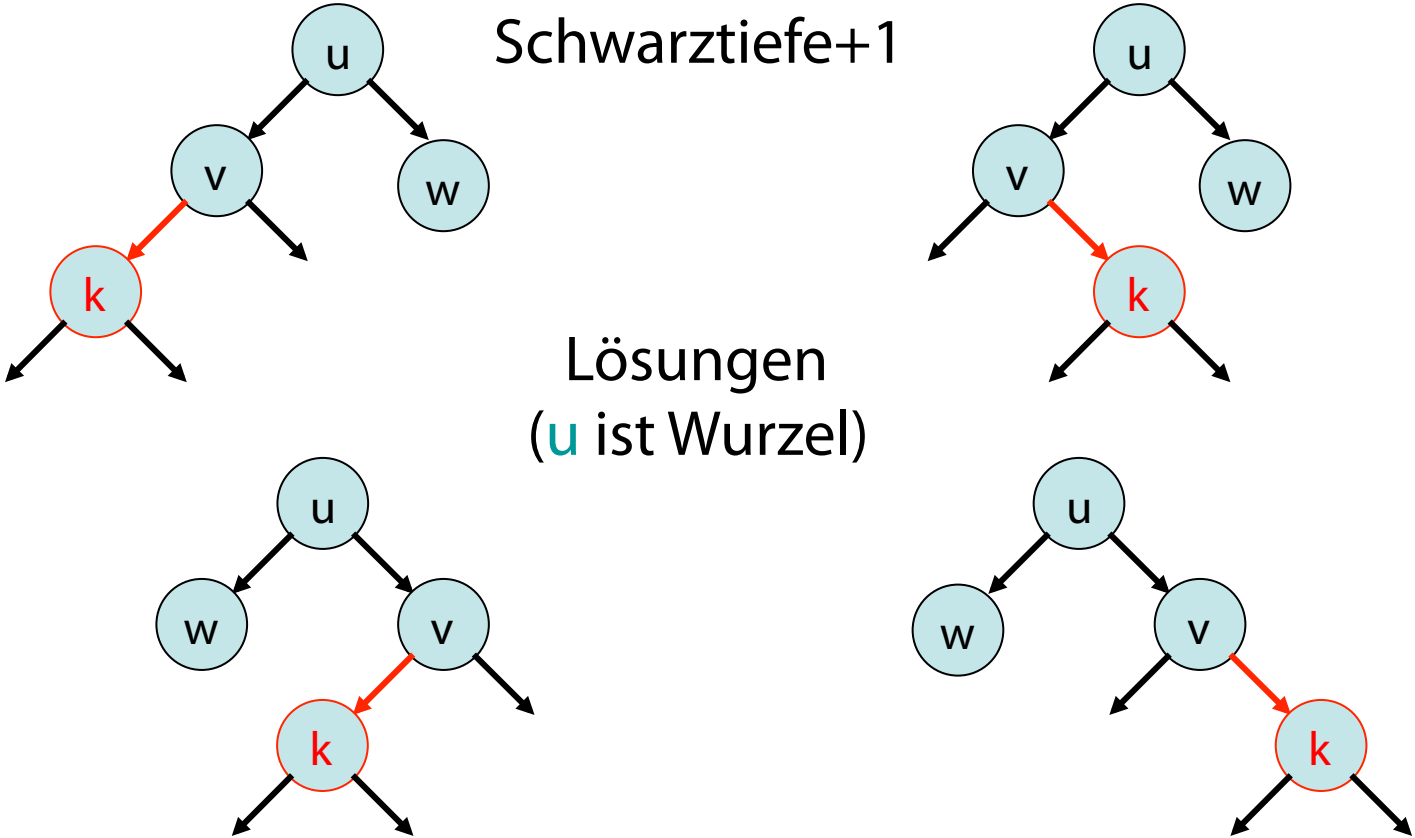
Rot-Schwarz-Baum

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



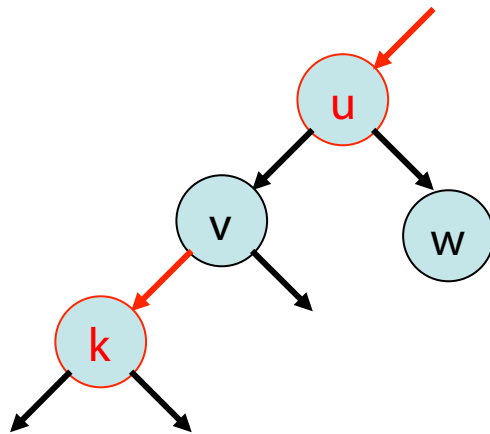
Rot-Schwarz-Baum

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

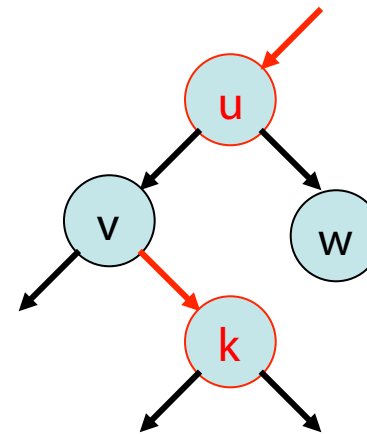


Rot-Schwarz-Baum

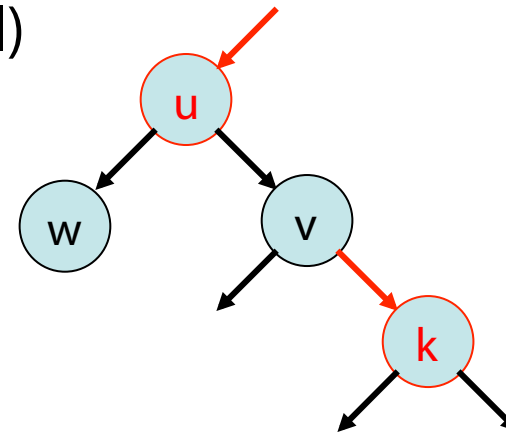
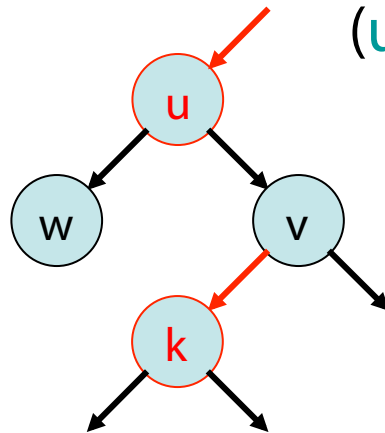
Fall 2: Vater v von k in T hat roten Bruder w



bewahrt
Schwarztiefe!

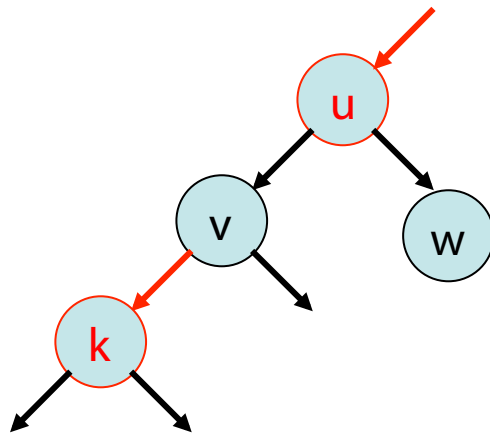


Lösungen
(u keine Wurzel)

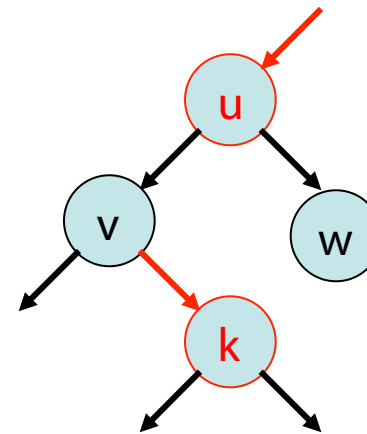


Rot-Schwarz-Baum

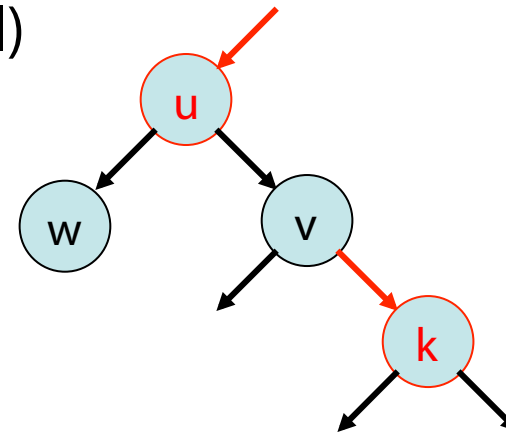
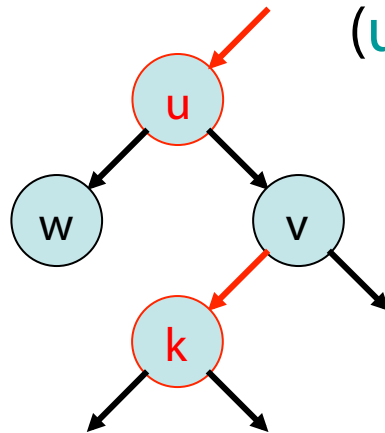
Fall 2: Vater v von k in T hat roten Bruder w



weiter mit u
wie mit k



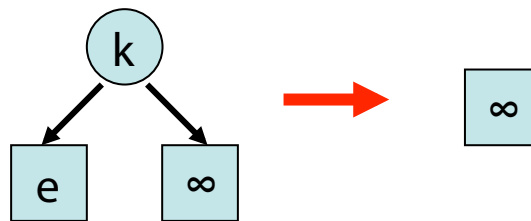
Lösungen
(u keine Wurzel)



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):

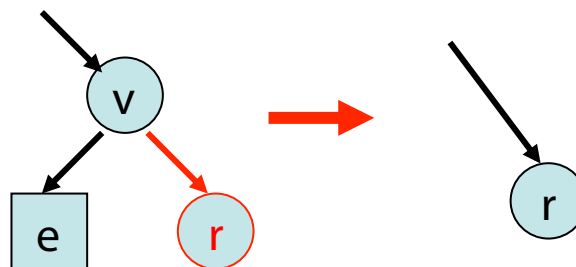
- Führe `search(k)` auf Baum aus
- Lösche Element `e` mit `key(e)=k` wie im binären Suchbaum
- Fall 2: Vater `v` von `e` ist rot (d.h. Bruder schwarz)



Rot-Schwarz-Baum

delete(k):

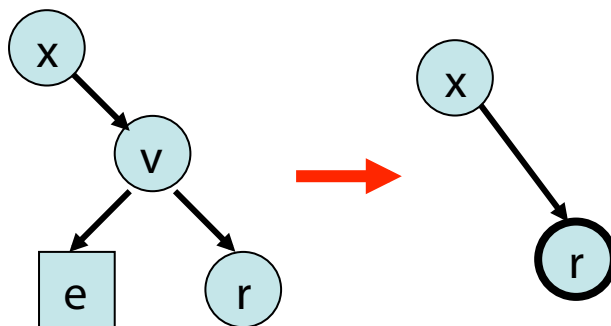
- Führe `search(k)` auf Baum aus
- Lösche Element `e` mit `key(e)=k` wie im binären Suchbaum
- Fall 3: Vater `v` von `e` ist schwarz und Bruder rot



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):

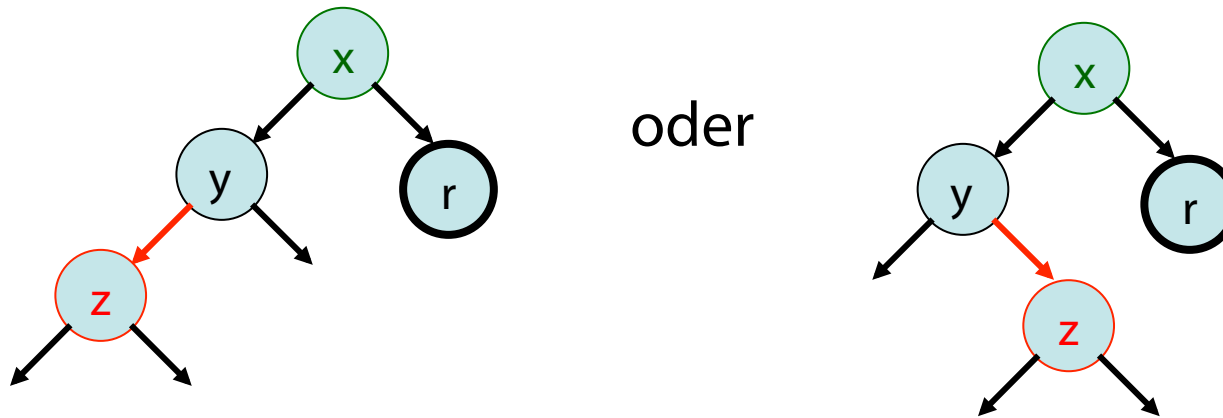
- Führe `search(k)` auf Baum aus
- Lösche Element `e` mit `key(e)=k` wie im binären Suchbaum
- Falls Vater `v` von `e` und Bruder `r` sind schwarz, dann 3 weitere Fälle:
 - Fall 1: Bruder `y` von `r` ist schwarz und hat rotes Kind
 - Fall 2: Bruder `y` von `r` ist schwarz und beide Kinder von `y` sind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

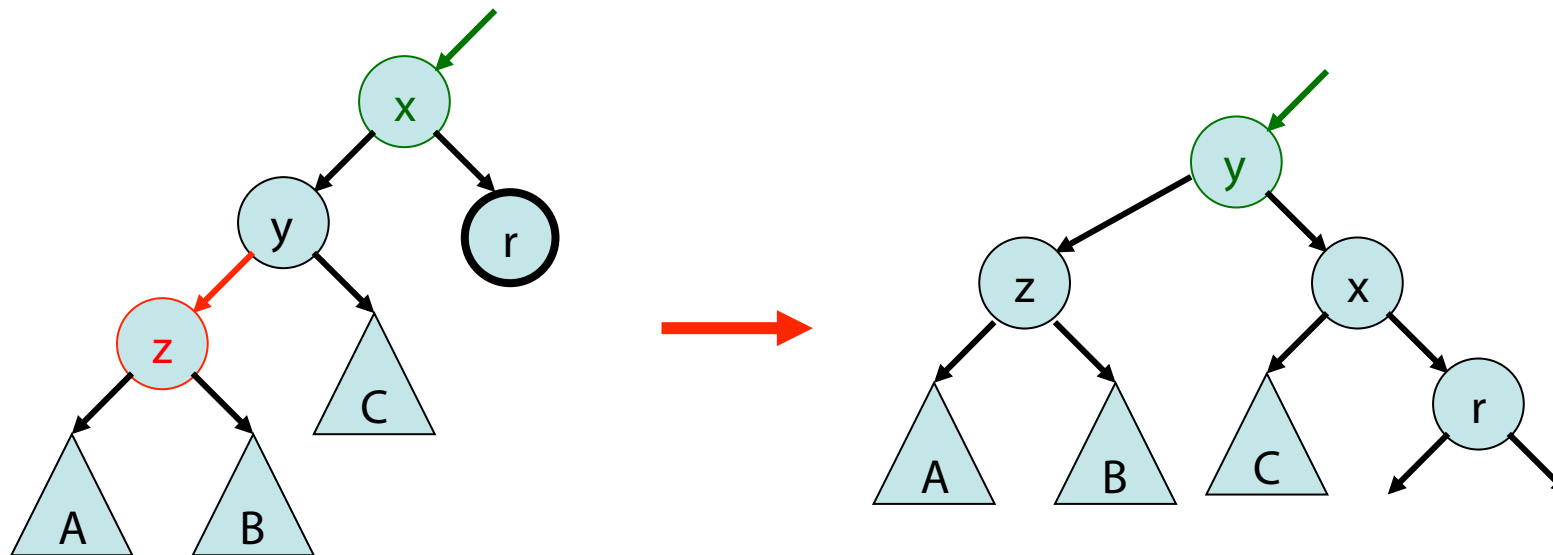
Alternativen für Fall 1: (x : beliebig gefärbt)



Rot-Schwarz-Baum

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

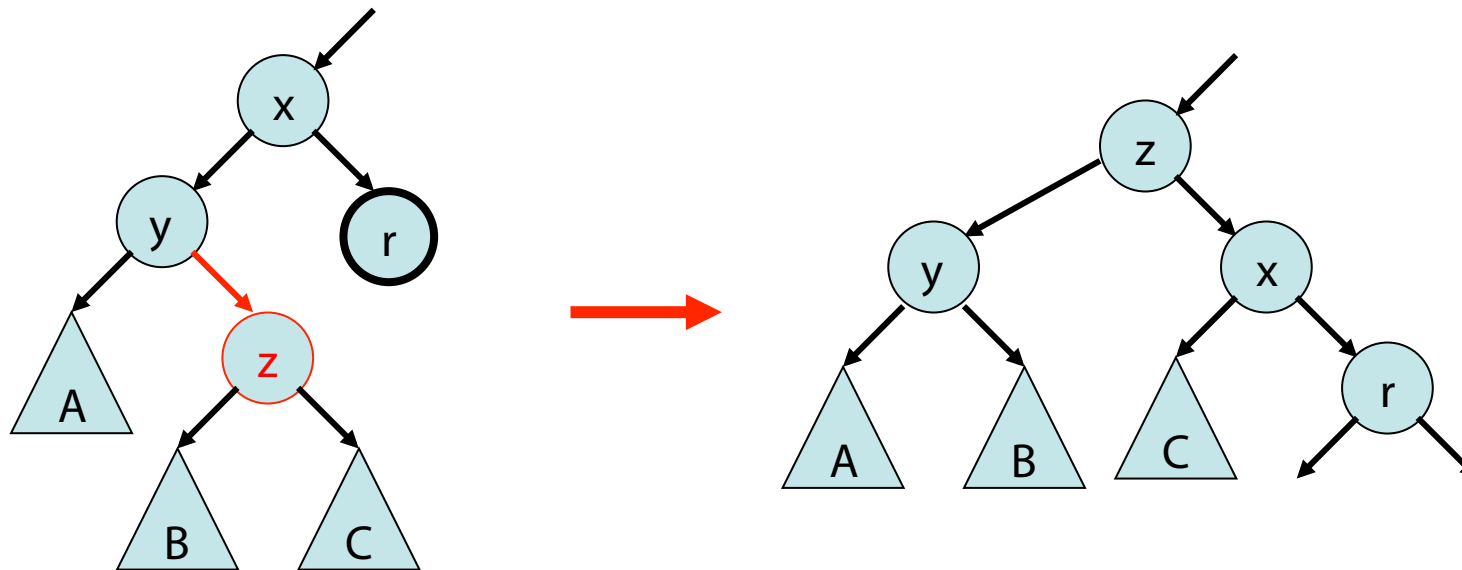
O.B.d.A. sei r rechtes Kind von x (links: analog)



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)

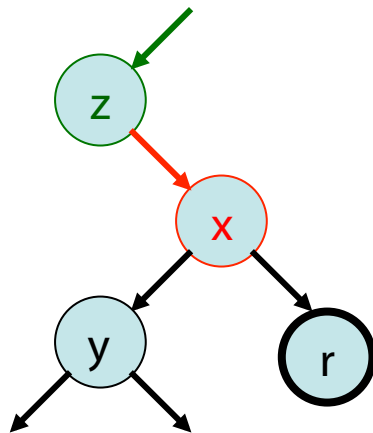


Rot-Schwarz-Baum

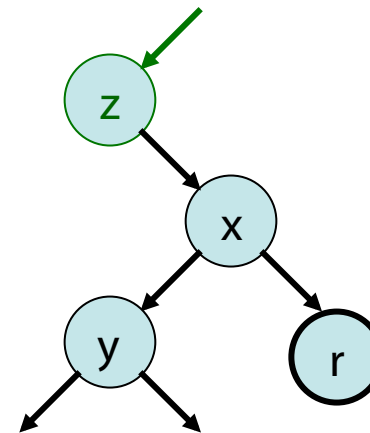
Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz

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

Alternativen für Fall 2: (z beliebig gefärbt)



oder

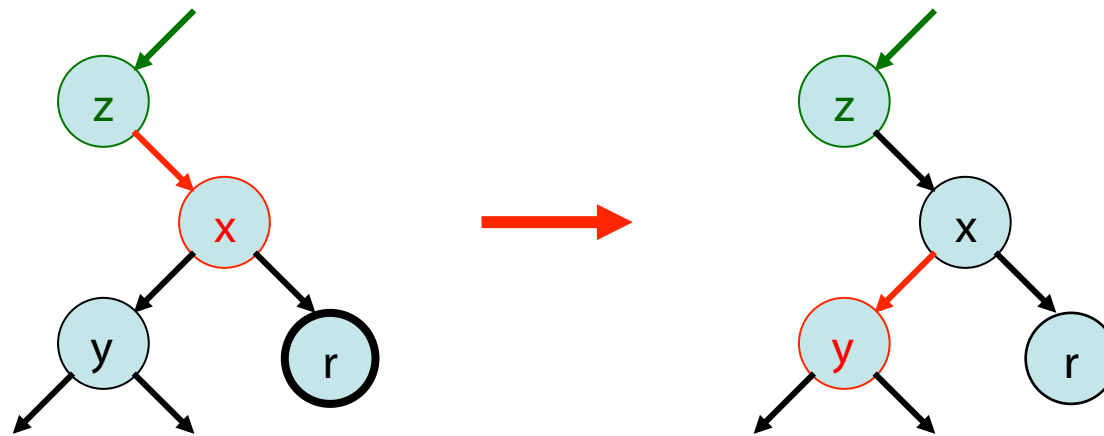


Rot-Schwarz-Baum

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)

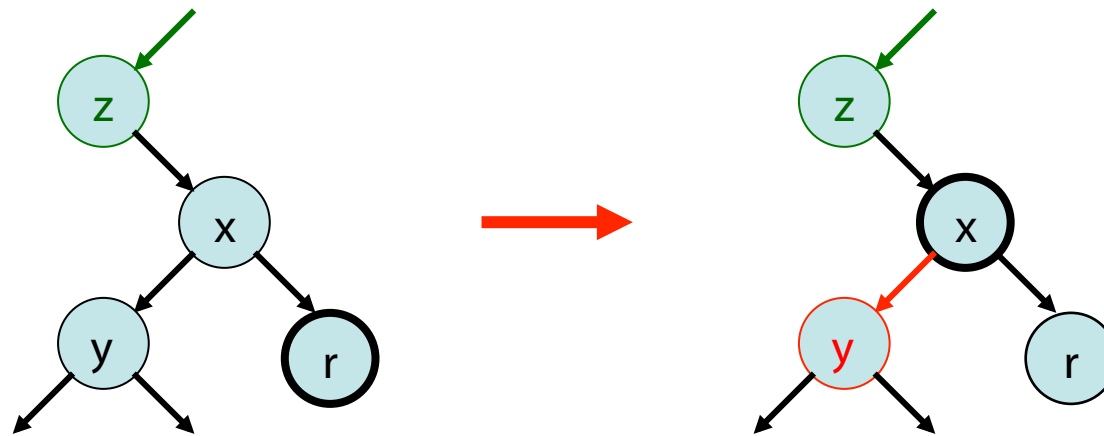


Rot-Schwarz-Baum

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)



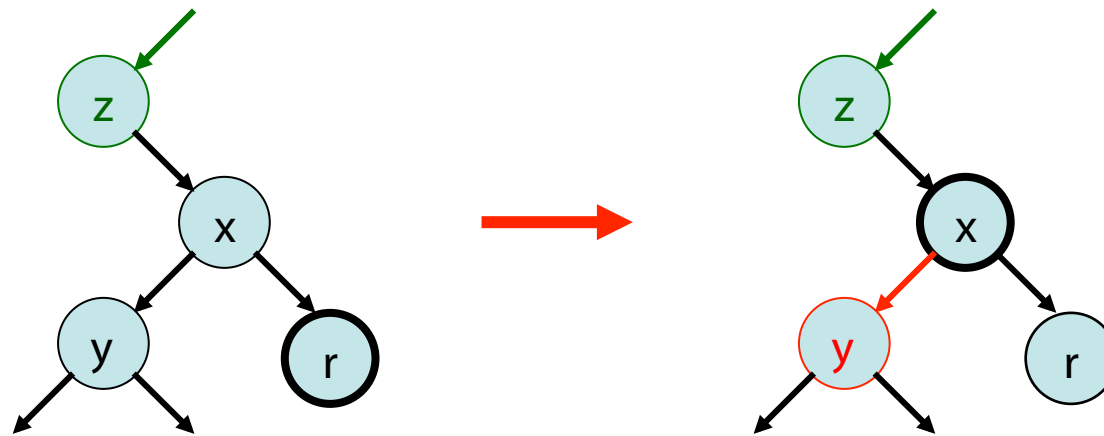
x ist Wurzel: fertig (Schwarztiefe-1)

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)

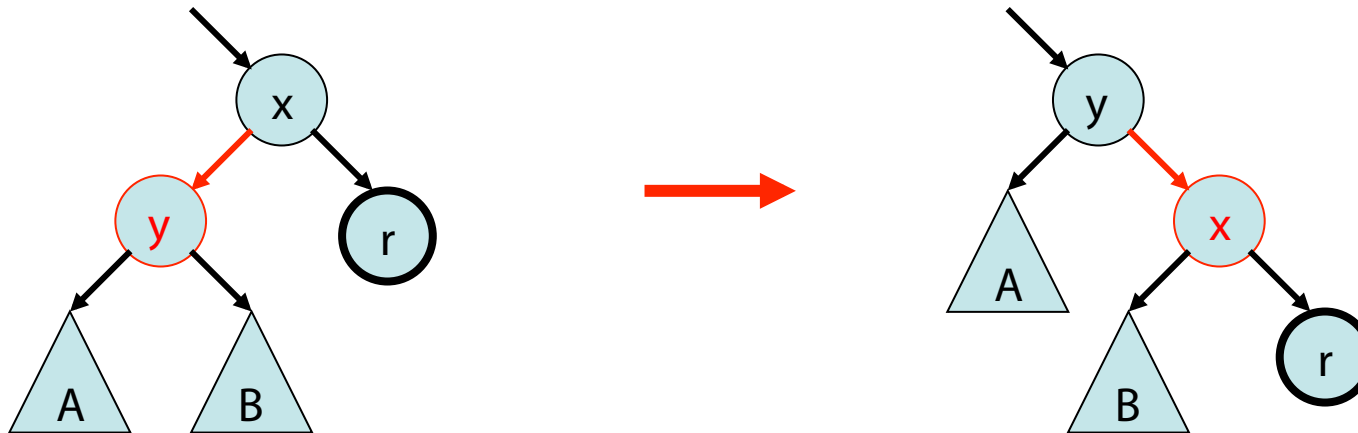


x keine Wurzel: weiter wie mit r

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: $O(\log n)$ amort.
- delete: $O(\log n)$ amort.

AVL-Bäume

- Ein binärer Suchbaum heißt **AVL-Baum**, falls für die beiden Teilbäume $T1$ und $T2$ der Wurzel gilt:
 - $|h(T1) - h(T2)| \leq 1$
 - $T1$ 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**.
- Die **minimale Höhe** eines AVL-Baumes mit n Schlüsseln ist $\log_2(n+1)$.
- Die **maximale Höhe** eines AVL-Baumes mit n Schlüsseln ist $O(\log n)$.
- Im **Durchschnitt** ist ein AVL-Baum **ca. 44% höher** als ein vollständig ausgeglichener Baum.

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/infoll/>

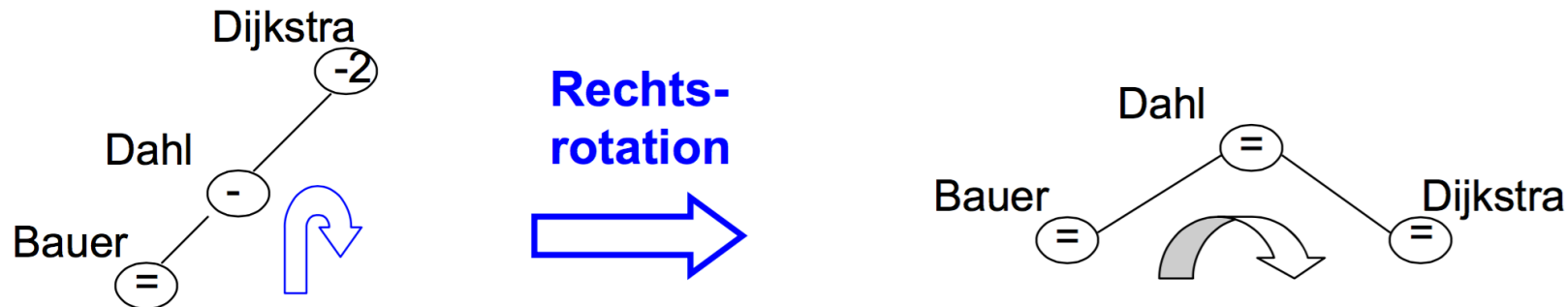


Einfügen in AVL-Baum



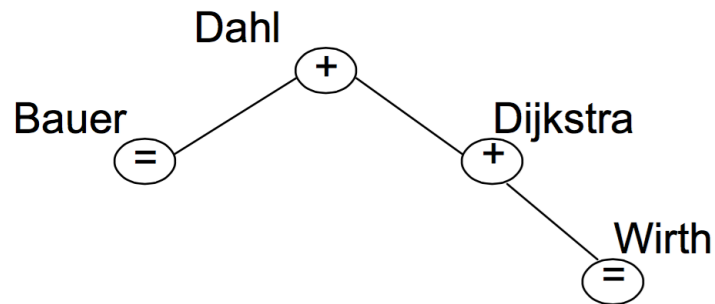
Einfügen von Dahl:
Neuberechnung des
Balancierungsfaktors,
AVL Kriterium erfüllt

Einfügen von Bauer: Verletzung des AVL Kriteriums



Nach Rechtsrotation: AVL Kriterium wieder erfüllt

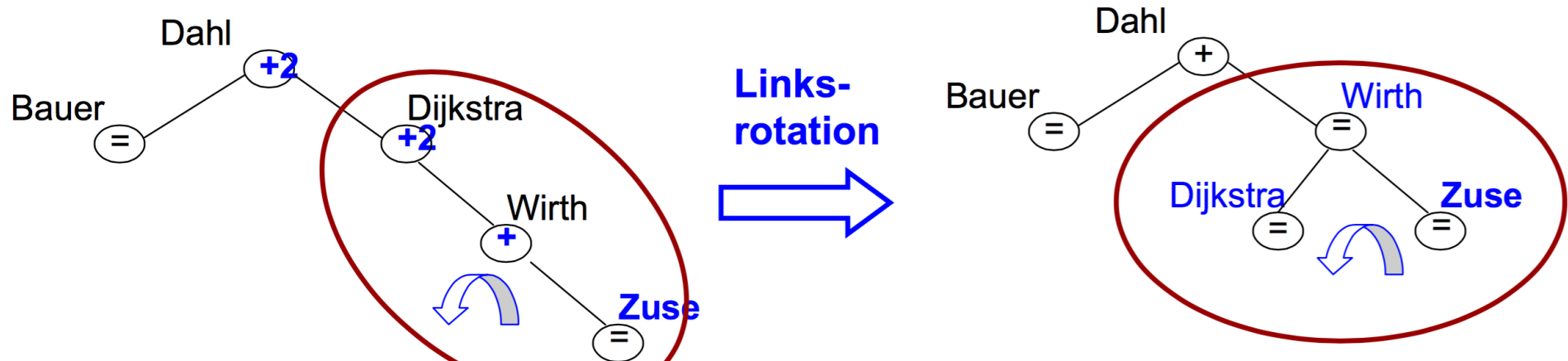
Einfügen in AVL-Baum



Einfügen von Wirth:

Nach Neuberechnung des Balancierungsfaktors ist AVL-Kriterium weiter erfüllt

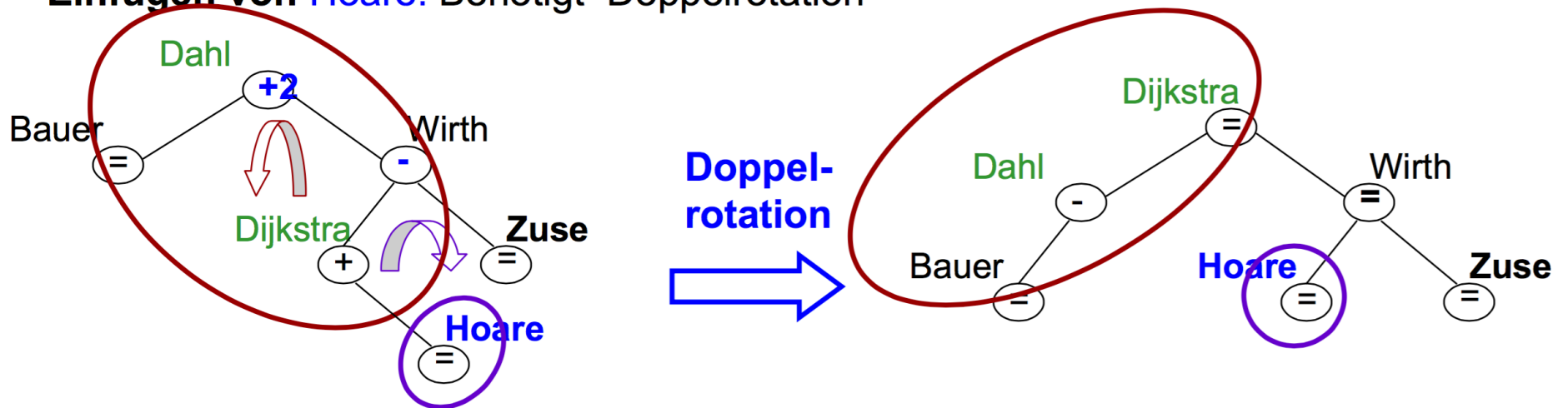
Einfügen von Zuse:



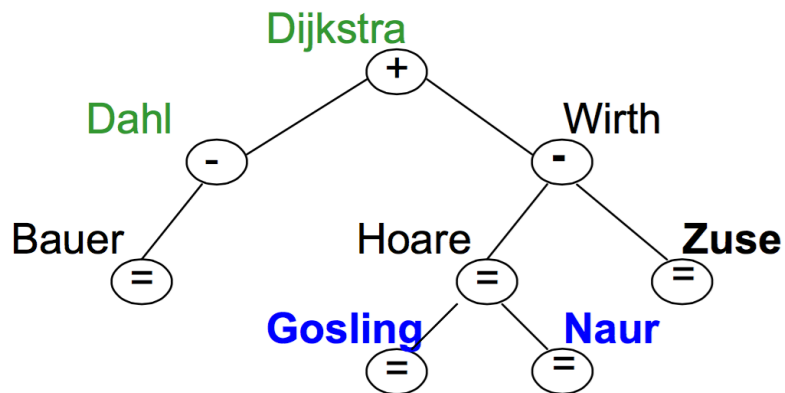
Nach Linksrotation: AVL Kriterium wieder erfüllt

Einfügen in AVL-Baum

Einfügen von **Hoare**: Benötigt Doppelrotation

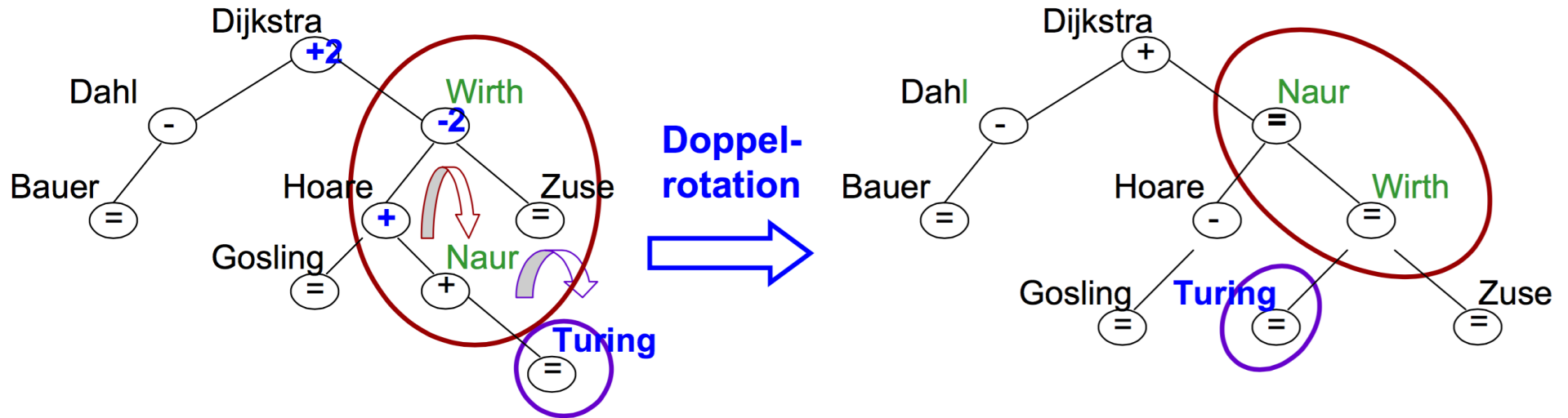


Einfügen von **Gosling** und **Naur**: ohne Probleme



Einfügen in AVL-Baum

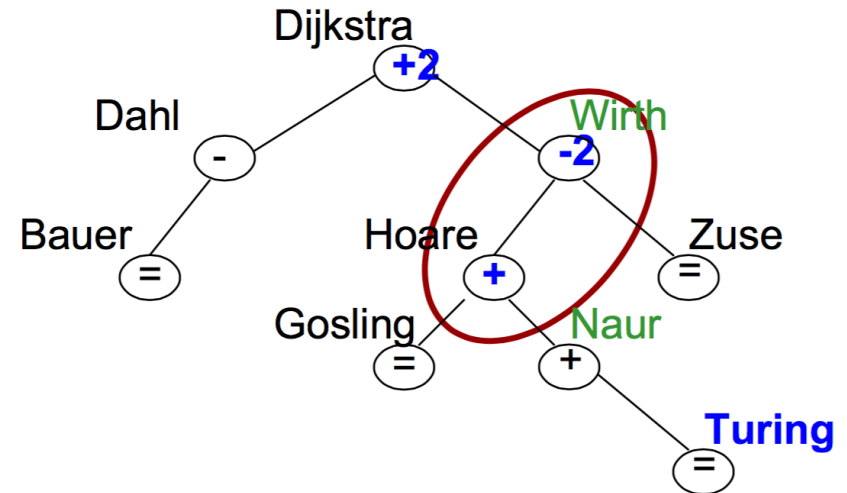
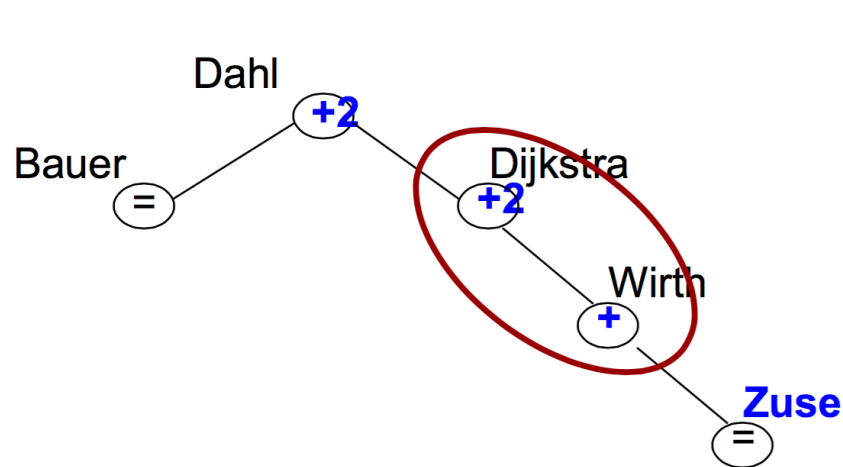
Einfügen von Turing: Noch einmal Doppelrotation



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

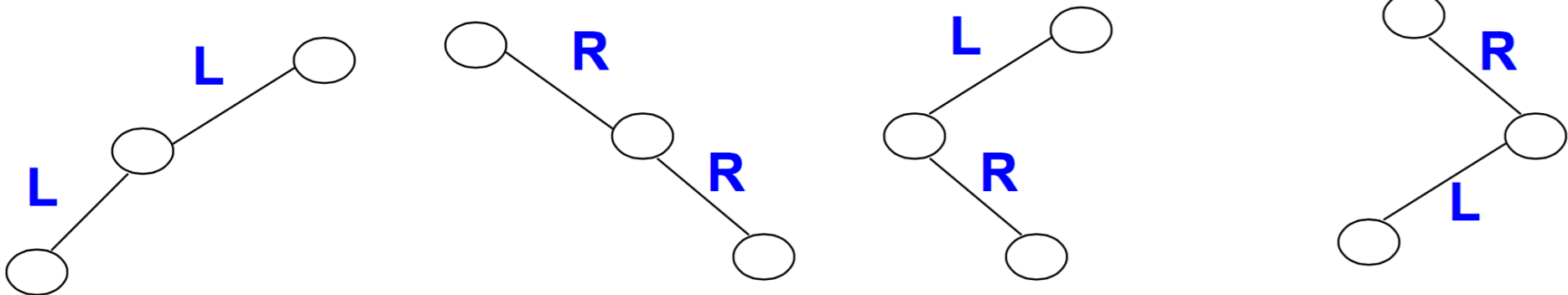
Anwendungsstelle der Rotation

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



Rotationstypen

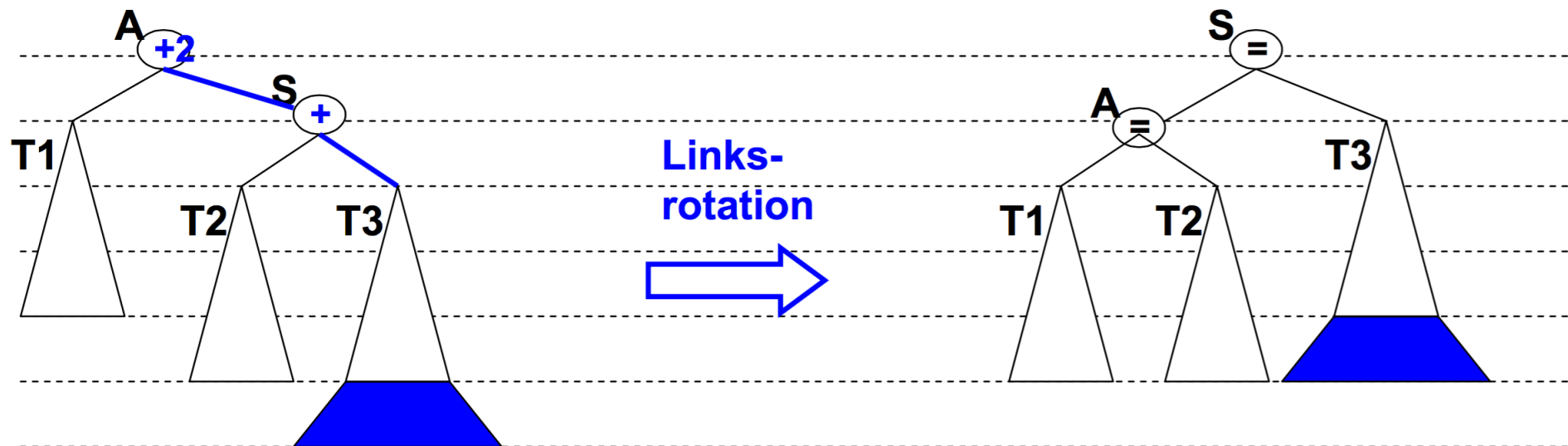
- Betrachte ausgehend vom tiefsten Knoten mit $BF = 2$ den Pfad zur Einfügeposition:
 - **RR: Rechts-Rechts** Linksrotation
 - **LL: Links-Links** Rechtsrotation
 - **RL: Rechts-Links** Doppelrotation „rechts“
 - **LR: Links-Rechts** Doppelrotation „links“



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

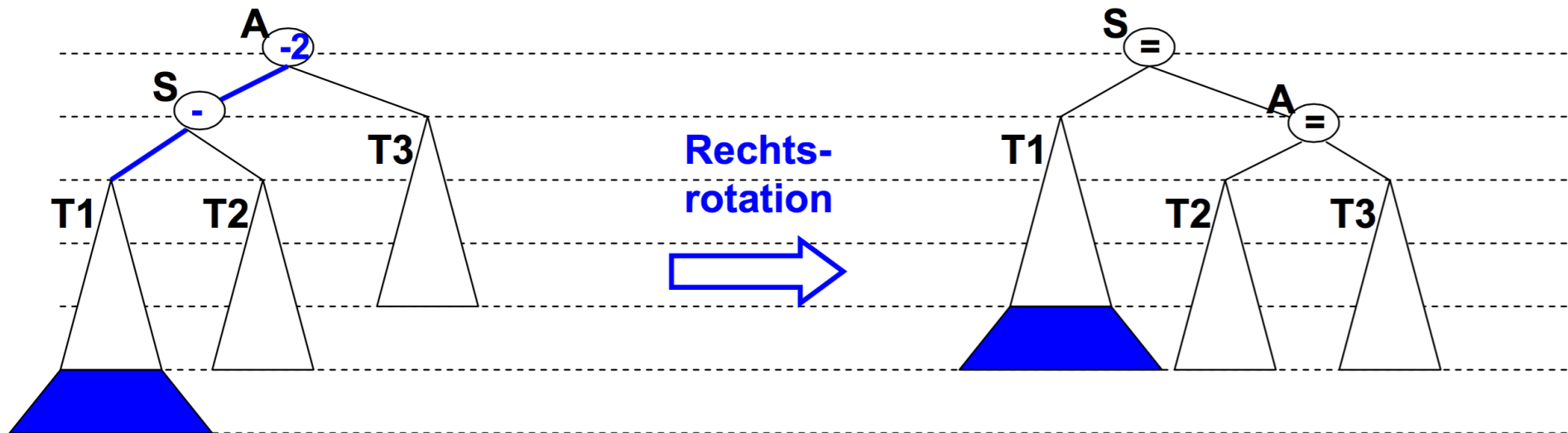
Typ RR: Linksrotation

- 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“



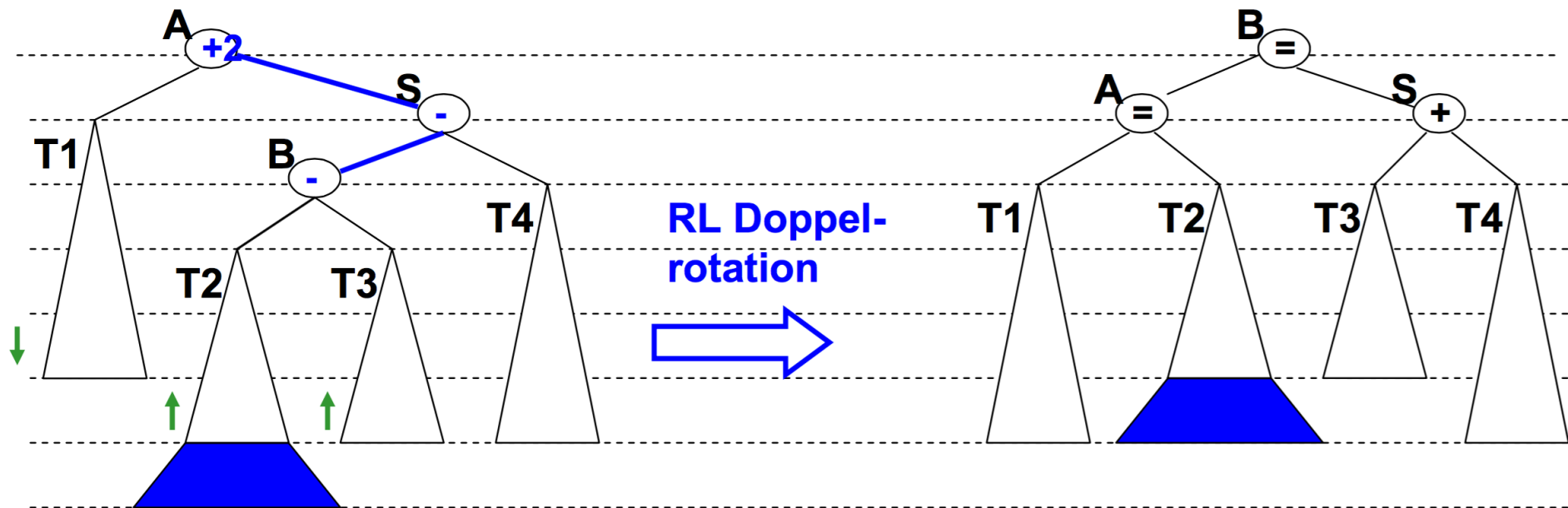
Typ LL: Rechtsrotation

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



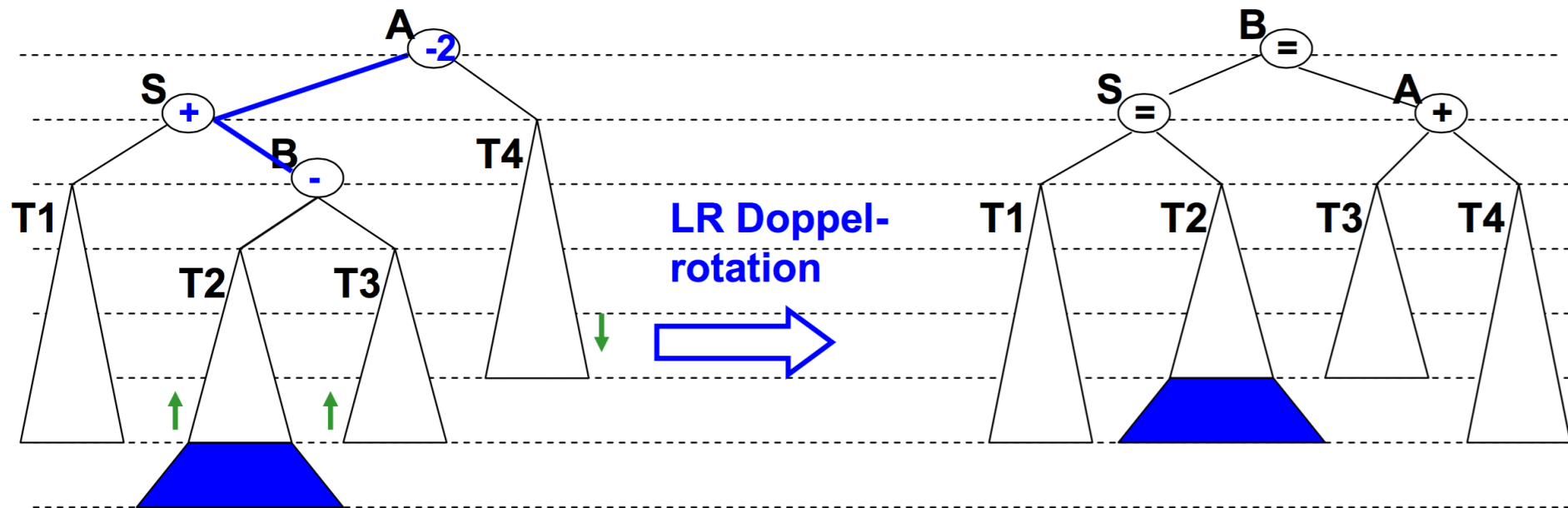
Typ RL: Doppelrotation

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



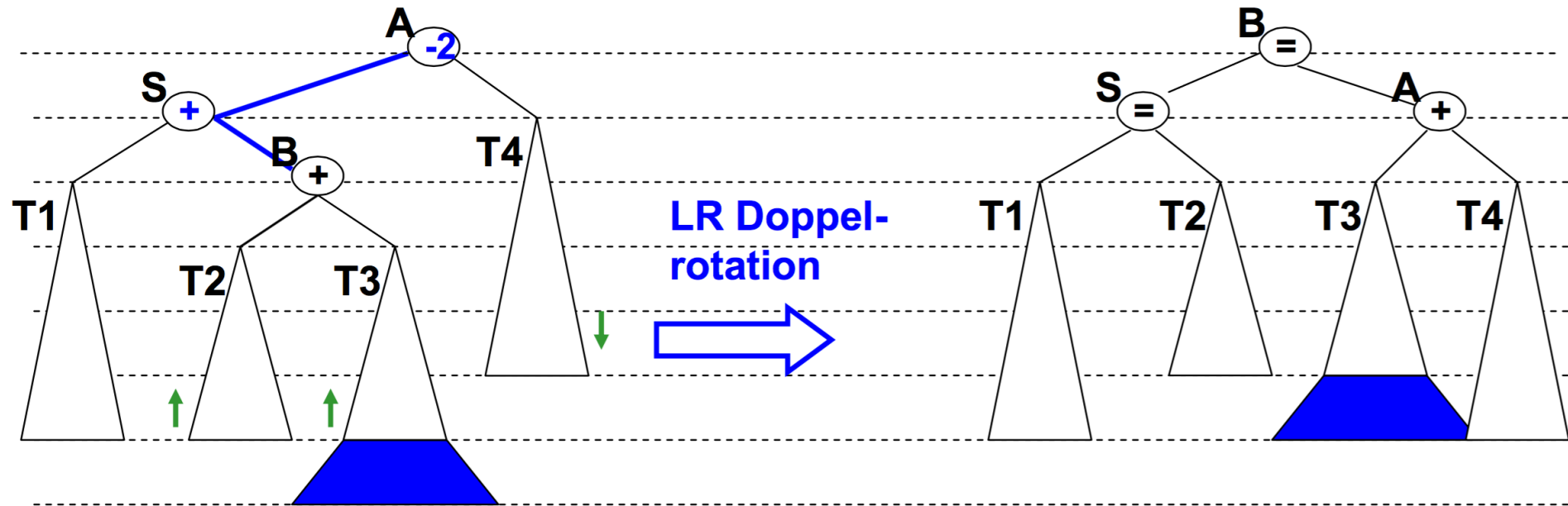
Typ LR: Doppelrotation

- 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 das Einfügen) zu Strukturverletzungen führen, die durch Rotationen ausgeglichen werden
 - Beim Löschen genügt nicht immer eine einzige Rotation oder Doppelrotation
 - Im schlechtesten Fall muss auf dem Suchpfad bottom-up vom zu entfernenden Schlüssel bis zur Wurzel auf jedem Level eine Rotation bzw. Doppelrotation durchgeführt werden

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. $\log n$

Splay-Baum:

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

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