Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren

DataStructures.jl

This package implements a variety of data structures, including

- Deque (implemented with an unrolled linked list)
- CircularBuffer
- CircularDeque (based on a circular buffer)
- Stack
- Queue
- Priority Queue
- Fenwick Tree
- Accumulators and Counters (i.e. Multisets / Bags)
- Disjoint Sets
- Binary Heap
- Mutable Binary Heap
- Ordered Dicts and Sets
- RobinDict and OrderedRobinDict (implemented with Robin Hood Hashing)
- SwissDict (inspired from SwissTables)
- Dictionaries with Defaults
- Trie
- Linked List and Mutable Linked List
- Sorted Dict, Sorted Multi-Dict and Sorted Set
- DataStructures.IntSet
- SparseIntSet
- DiBitVector
- Red Black Tree
- AVL Tree
- Splay Tree

```
julia> using DataStructures
julia> pq = PriorityQueue();
```

Wiederholung

Klare Trennung zwischen Beispielimplementierung und Julia-Internem/ Julia-Paketen

Beispielimplementierung aus der Vorlesung

```
l = make_list()
insert(1, l)
insert(2, l)
print(l) # [2, 1]
```

- Keine ! in den Namen der Funktionen
- Parameter in der Reihenfolge
  1. Element
  2. Datenstuktur

In Julia integrierte Arrays

```
l = []
insert!(l, 1, 1)
insert!(l, 1, 2)
print(l) Any[2, 1]
```

- ! im Namen von Funktion → verändert Datenstruktur
- Parameter in der Reihenfolge
  1. Datenstruktur
  2. Element

Arrays in Julia

```
l = []
push!(l, 1)
push!(l, 2)
print(l) # [1, 2]
```

push! ∈ O(1)amort

```
l = []
insert!(l, 1, 1)
insert!(l, 1, 2)
print(l) # [2, 1]
```

insert! ∈ O(1)amort??

insert!(.,1,.) ∈ O(1)amort??

Mengen in Julia

- Datenstruktur zum Sammeln von Objekten …
- … ohne Duplikate

```
julia> s = Set([42, 23, 17, 42])
Set{Int64} with 3 elements:
  42
  17
  23

julia>
```

- Typen werden vom Julia-System ermittelt…
- … und bei den Daten vermerkt

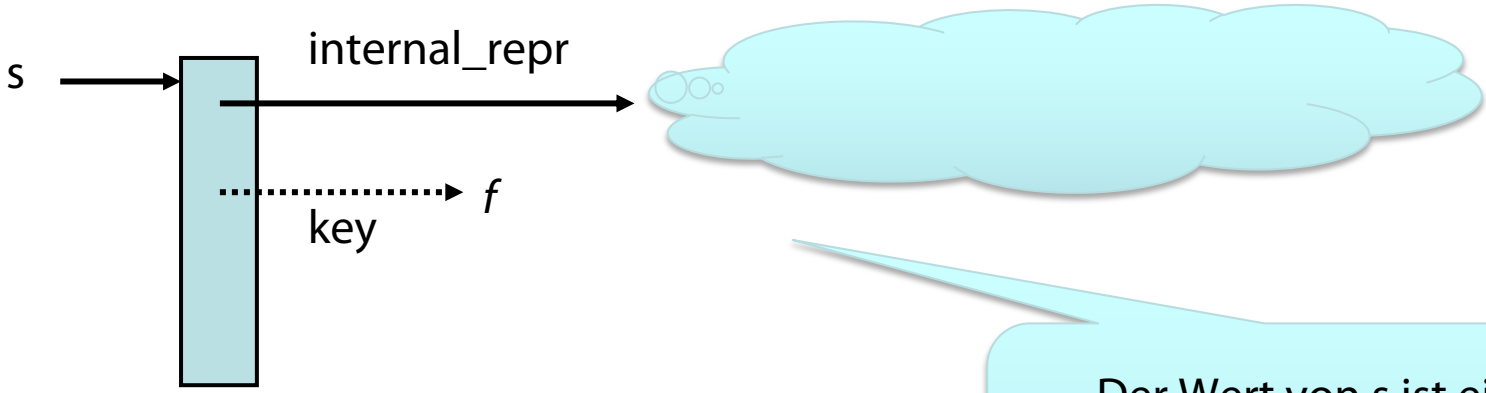
Implementierung von Mengen

```
function identity(x)
    return x
end
```

- Wir werden hinter die Kulissen blicken
- Eigene Implementierung von Set: KeySet
  - s = KeySet([1, 2, 3]) // key ist identity
  - s = KeySet([1, 2, 3], key) // key wird vorgegeben

Beispielimplementierung für KeySet in Julia vorhanden.

KeySet



Der Wert von s ist eine
Instanz
des Datentyps Menge

```
function key(set::KeySet)
    return set.key
end
```

key(s) liefert Funktion *f*

Sei f eine einstellige Funktion

key(s)(x) wendet Funktion f auf x an

Abstrakter Datentyp: Iteratoren

- Wie kann ich über eine Struktur iterieren, deren interne Repräsentation verborgen ist?

- Iteratoren für einfache Mengen und Multimengen
  - `Base.iterate(set)`
  - `Base.iterate(set, state)`

Iteratoren auch definierbar für Keller, Schlangen, PQs,…

- `Base` ist ein Julia Modul und liefert grundlegende Funktionen
- `Base` ist immer verfügbar und beinhaltet z.B.
  - `Base.length()`, die bereits bekannte Funktion `length()`

Iteration über die Elemente von ADTs

- Wir wollen von der genauen Datenstruktur abstrahieren

```
element_state = iterate(set)
while !isnothing(element_state)
    (element, state) = element_state

    # Etwas mit dem Element tun
    print(element)

    element_state = iterate(set, state)
end
```

- Funktion `iterate(set)`
  - Iterator erzeugen
  - Tupel mit erstem Element und Zustand oder nothing falls leer
- Funktion `iterate(set, state)`
  - Tupel mit nächstem Element und Zustand oder nothing falls zu Ende

Kurzschreibweise für die Iteratoranwendung

- Kurzschreibweise

```
for element in set
    print(element)
end
```

```
for element ∈ set
    print(element)
end
```

- … wird zu

```
element_state = iterate(set)
while !isnothing(element_state)
    (element, state) = element_state

    print(element)

    element_state = iterate(set, state)
end
```

Typdefinition und Methoden

- Typdefinition

```
struct MyType
  values :: Array{Any}
end
```

```
struct MyOtherType
  values :: Array{Any}
end
```

```
julia> MyOtherType([6, 6])
MyOtherType(Any[6, 6])
```

- Objekterstellung

```
MyType([12])
MyType([6, 6])
```

```
MyOtherType([12])
MyOtherType([6, 6])
```

- Methoden

```
function Base.show(io::IO, o::MyType)
    x = sum(o.values)
    print(io, ”MyType<$x>")
end
```

```
julia> MyType([6, 6])
MyType<12>
```

Wiederholung: Structs als Typen

- Objekte sind Zeiger auf ihre Attributwerttupel

```
mutable struct MyType
      values :: Array{Any}
end

a = MyType([12])
```

- Gilt `a == a`?
  - Ja
- Gilt `MyType([12]) == MyType([12])` ?
  - Nein
- Problem?
  - `in( MyType([12]), Set([MyType([12]), MyType([13])]) )`
  - Liefert nie true
- Wenn man == für MyType-Objekte definiert, sieht es anders aus

Vergleichsoperatoren für Typen

```
struct MyType
  values :: Array{Any}
end
```

```
struct MyOtherType
  values :: Array{Any}
end
```

- Vergleichsoperatoren für `MyType` definieren
  - ```
    function Base.:(==)(o::MyType, o_::MyType)
      return o.values == o_.values
    end
    ```
  - ```
    function Base.:(>=)(o::MyType, o_::MyType)
      return sum(o.values) >= sum(o_.values)
    end
    ```
- Was ergibt?
  - `MyType([12]) == MyType([12])` → `true`
  - `MyOtherType([12]) == MyOtherType([12])` → `false`
  - `MyType([6, 6]) >= MyType([12])` → `true`
  - `MyOtherType([6, 6]) >= MyOtherType([12])` → Fehler

Noch einmal: Vergleiche

- Für Julia-eigene Typen ist == häufig sinnvoll definiert
- Es funktioniert z.B. bei
  - `"ab" == "ab"`
  - `[1,2,3] == [1,2,3]`
  - `Set([1,2]) == Set([2,1])`
- Auch selbst definierte non-mutable Structs werden richtig verglichen

```
struct MyInt
  value :: Int
end
```

```
MyInt(12) == MyInt(12)
```
true

```
MyInt(12) <= MyInt(13)
```
Fehler

- Aber: Bei mutable Structs muss man den Vergleich selbst definieren.

Mengen: API (1)

- Obligatorische Operationen
  - test(k, s) testet, ob ein Element mit Schlüssel k in s enthalten ist
    liefert true oder false
  - search(k, s)

    (1a) liefert das Element aus s, dessen Schlüssel k ist, oder nothing

    (1b) liefert das Element 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, wenn Element
    mit key(s) (x) vorher in s enthalten, wird es aus s entfernt)
  - delete(k, s) löscht Element x mit key(s)(x) = k aus s
    (s wird modifiziert, wenn x enthalten war)

Mengen: API (2)

- Iterationsoperatoren
  - map(f, s)
    - Wendet f auf jedes Element aus s an
      und gibt Ergebnisse als neue Menge zurück
    - Beispiel
      - `function plus_1(x) return x + 1 end`
      - `map(plus_1, [42, 23, 17])` liefert `[43, 24, 18]`
      - Oder kurz `map((x) -> x+1, [42, 23, 17])`
  - fold(f, s, init) *oft auch reduce genannt*
    - Wendet die Funktion f kaskadierend
      auf dem jeweils vorigen Wert, beginnend mit init, und jeweils alle Elemente in s an und liefert letzten Wert (bzw. init, wenn s leer)
    - Beispiel
      - `function max(a, b) return if a > b a else b end end`
      - `fold(max, KeySet([4, 12, 24, 2]), 0)` liefert `24`

Mengen: API (3)

- Zusätzliche Operationen
  - union(s1, s2) (siehe auch merge bei PQs)
  - intersect(s1, s2)

Repräsentation einer Menge als verkettete Liste?

- **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:

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

- **Falls Misserfolg bei 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.

Beispielimplementierung in Julia vorhanden.

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

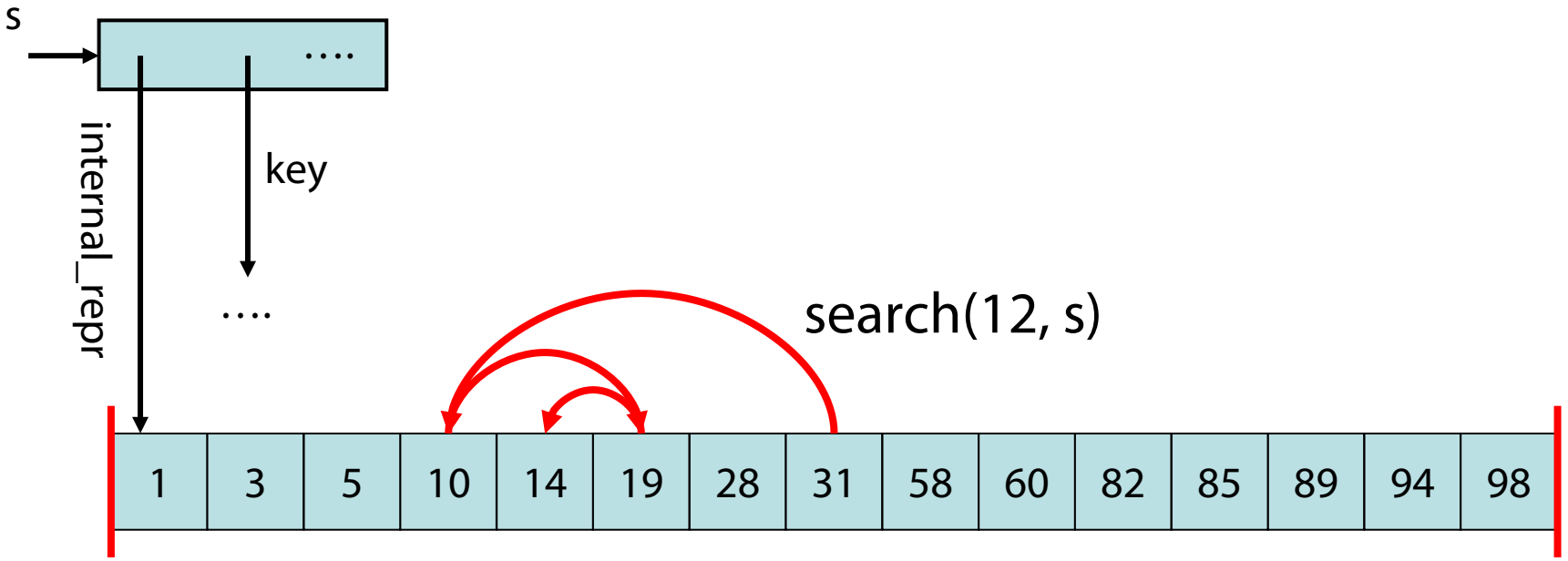
- **Feste Anordnung und naives Suchen** hat bei einer 7-elementigen Liste durchschnittlich den Aufwand:

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

- 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*

Repräsentation einer Menge als Array?

Speichere Elemente in sortiertem Feld



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

Binäre Suche

Eingabe: Zahl k und eine Menge repräsentiert als sortiertes Feld

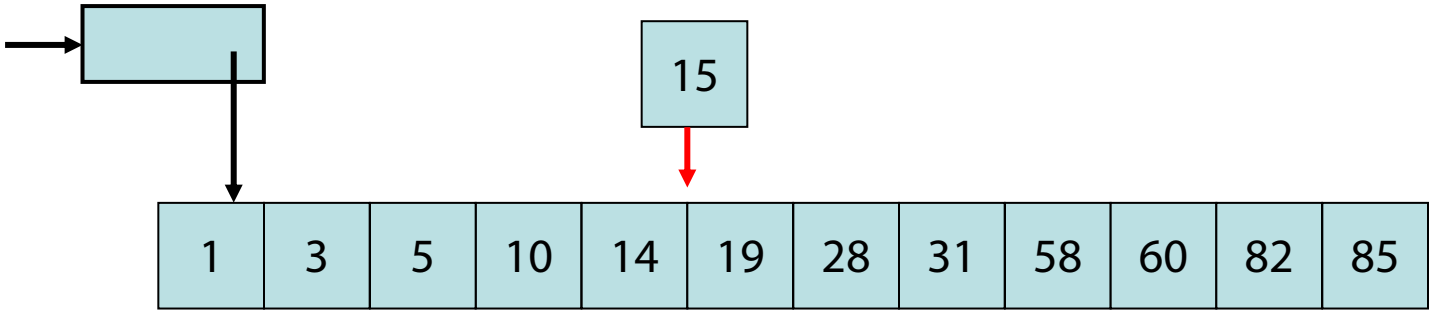
```
function binary_search(k, A)
    l = 1
    r = length(A)
    while l < r
        m = (r + l) ÷ 2
        if A[m] == k
            return A[m]
        elseif A[m] < k
            l = m + 1
        else
            r = m
        end
    end
    return A[l]
end
```

Gibt Element, sofern vorhanden oder nächstgrößeres zurück

Array als interne Repräsentation von Mengen

insert und delete Operationen:

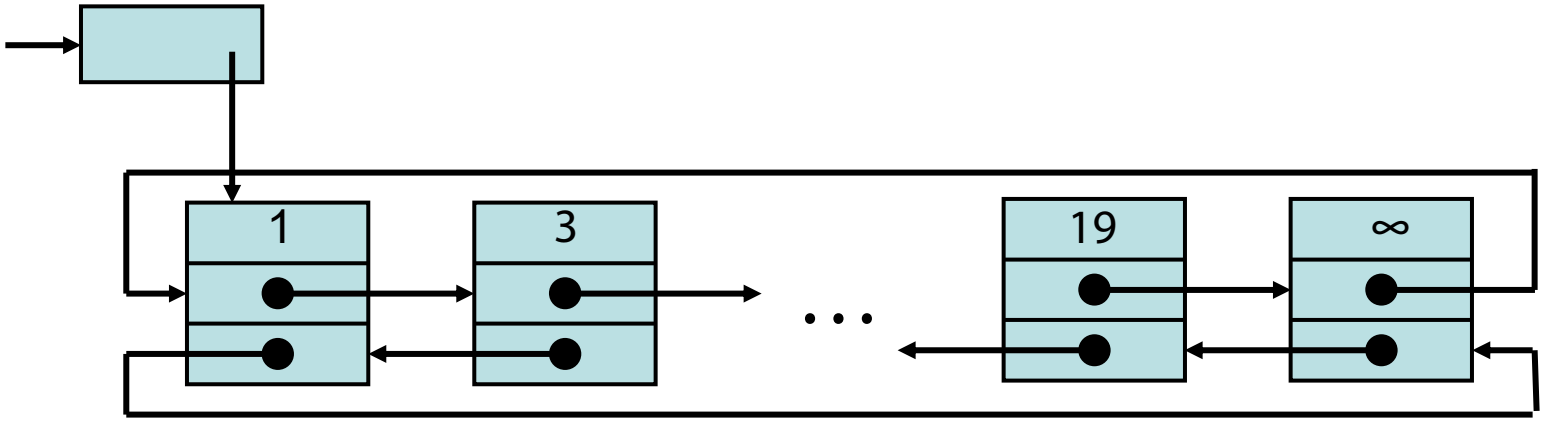
Sortiertes Feld schwierig zu aktualisieren!



Schlimmster Fall: $\theta(n)$ Zeit

Repräsentation als sortierte Liste?

Sortierte Liste (hier zyklisch und doppelt verkettet)



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

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

Search gibt Element oder nächstgrößeres zurück

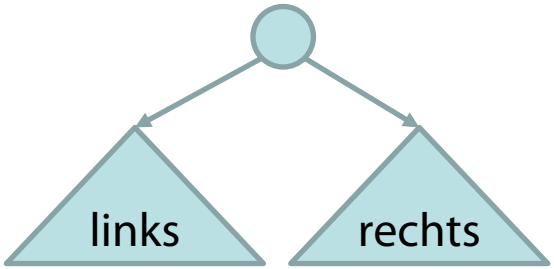
Motivation für Suche nach neuer Trägerstruktur

- Binäre Suche durchsucht Felder in O(log(n))
  - Einfügen neuer Elemente erfordert sequentielle Verschiebung (und ggf. ein größeres Feld und Umkopieren der Elemente)
- Einfügen in Listen in konstanter Zeit
  - Zugriffe auf Elemente jedoch sequentiell
- (Verzeigerter) Baum
  - Zum Suchen verwendbar
  - Einfügen: erst Suche, dann an richtiger Position einfügen
  - Iteration über Mengenelemente?

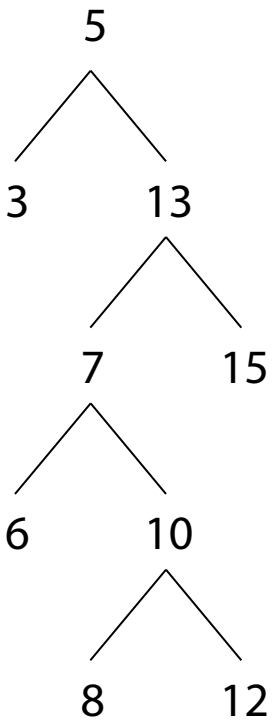
Binäre Suchbäume - Definition

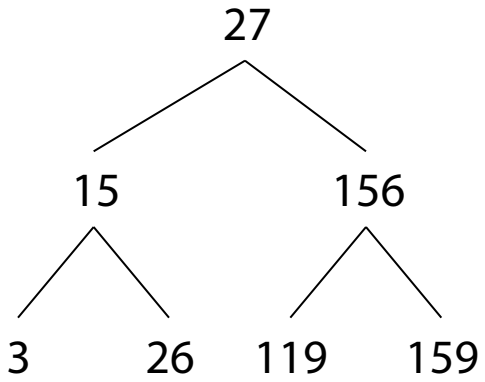
- Ein binärer Suchbaum
  1. ist ein Binärbaum, und
  2. zusätzlich muss für jeden seiner Knoten gelten, dass das im Knoten *gespeicherte Element*
     a) **≥** ist als alle *Elemente* im *linken Unterbaum*
     b) **<** ist als alle *Elemente* im *rechten Unterbaum*

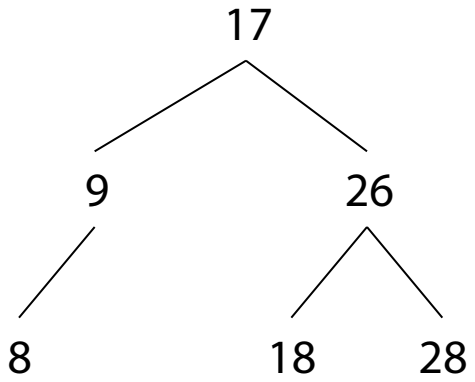
Unterschied zum Heap?



Beispiele für binäre Suchbäume



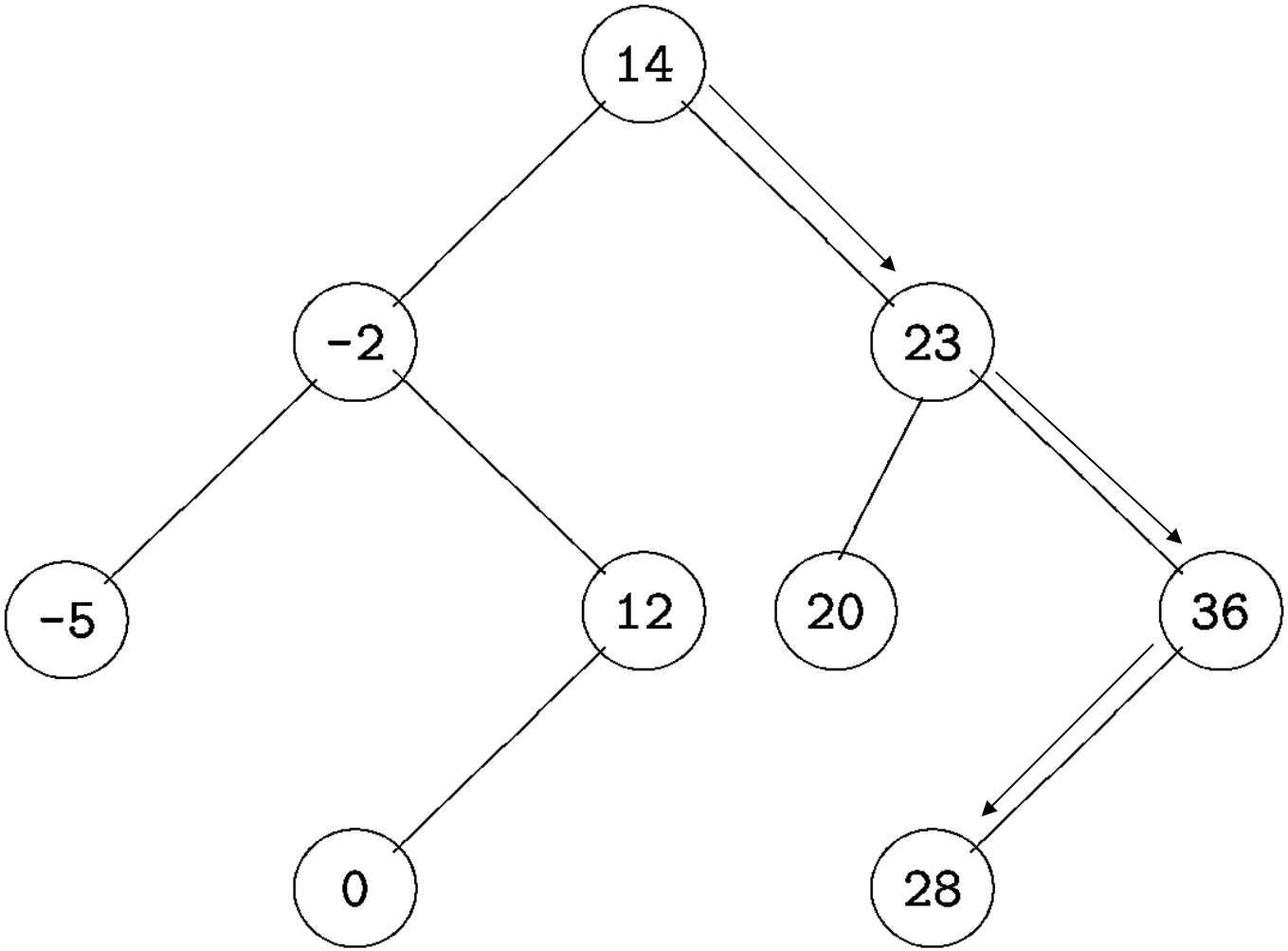




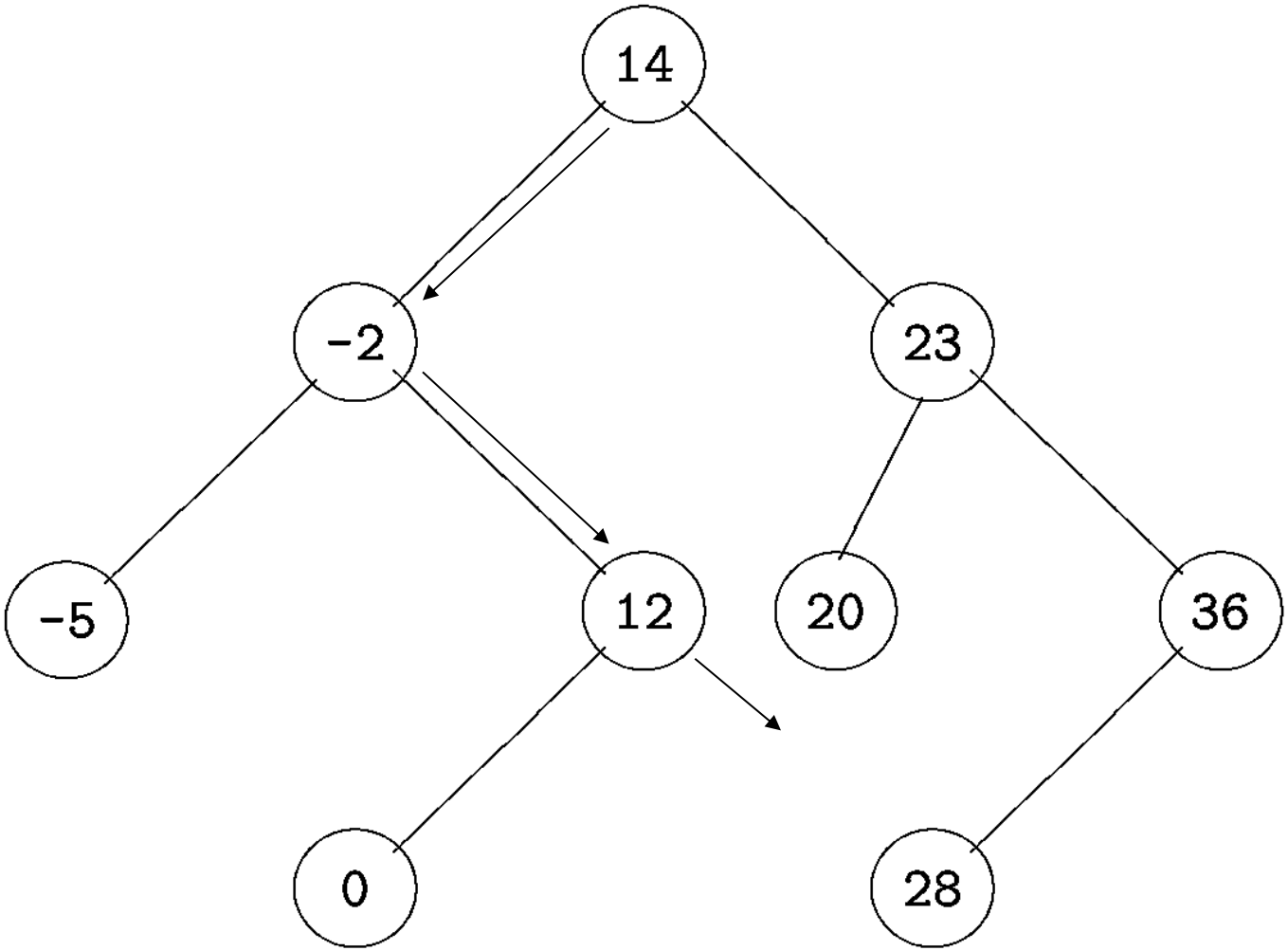
Suche im binären Suchbaum

- 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

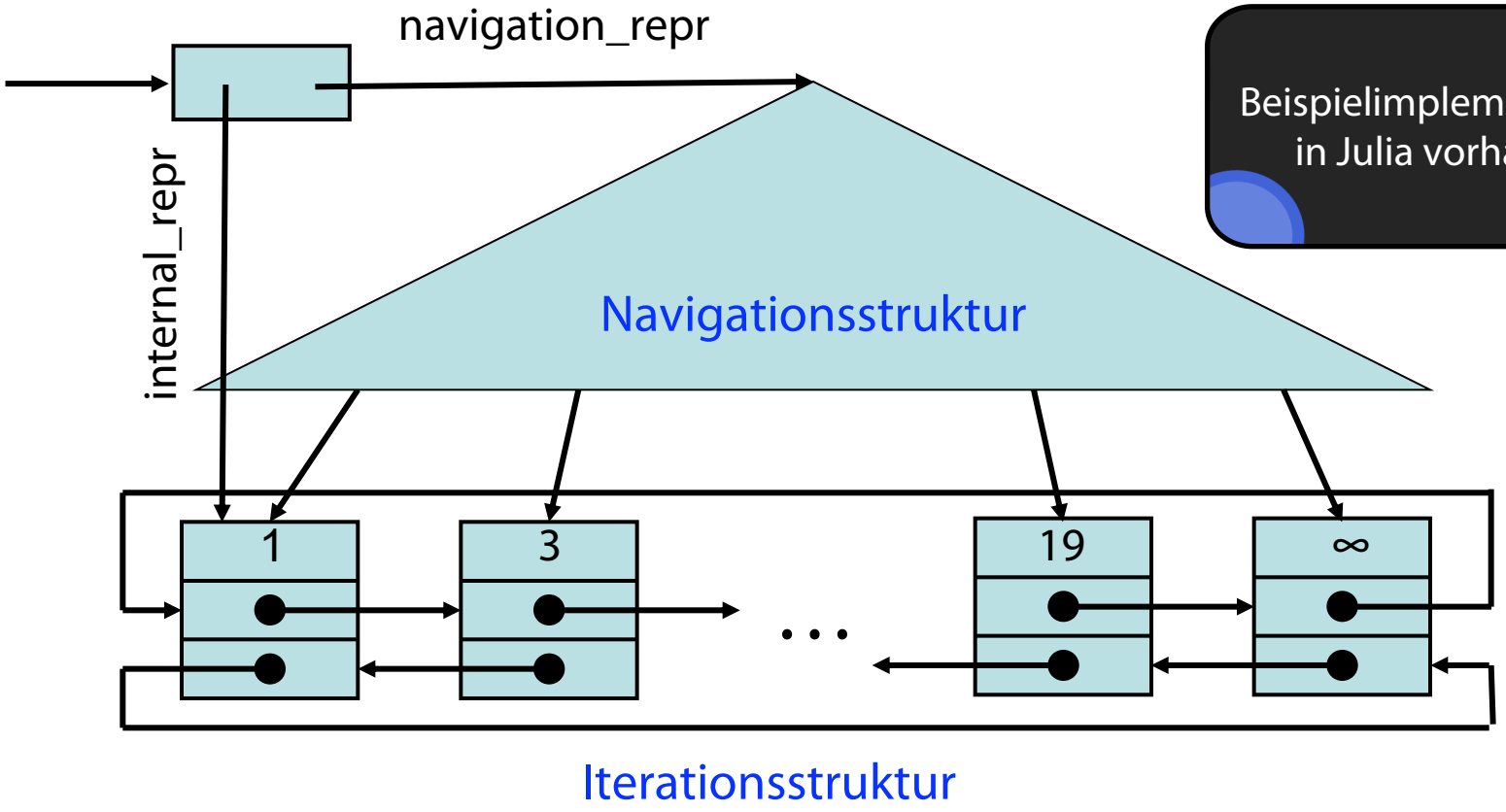


Binärer Suchbaum

- Wie kann man Iteratoren realisieren?
  - `Base.iterate(set)`
  - `Base.iterate(set, state)`

Suchstruktur

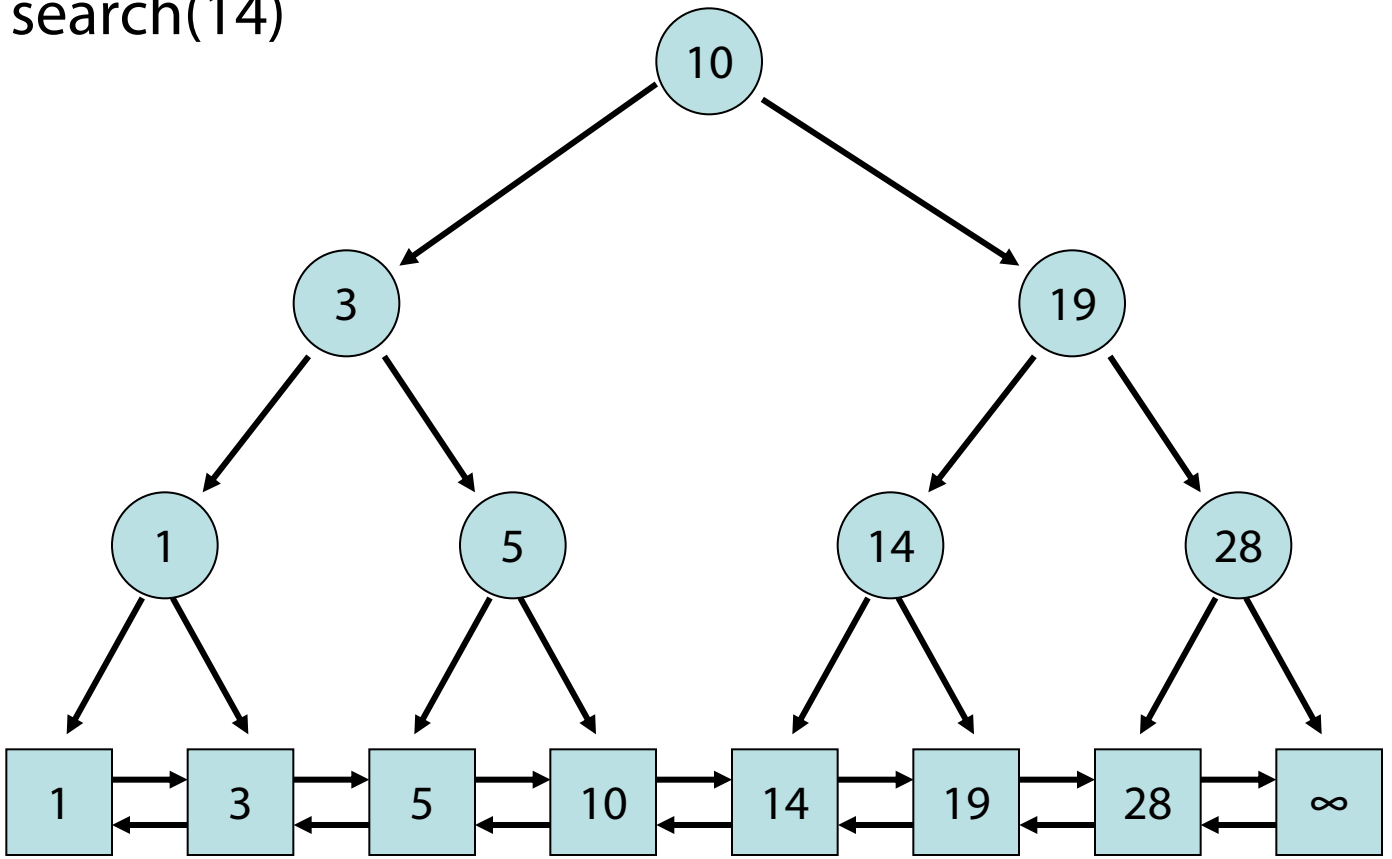
Idee: Baum als Navigationsstruktur (nur Schlüsselwerte), die search effizient macht, plus doppelt verkettete zyklische Liste als Iterationsstruktur und zum einfachen Einfügen



Beispielimplementierung in Julia vorhanden.

Binärer Suchbaum (ideal)

search(14)

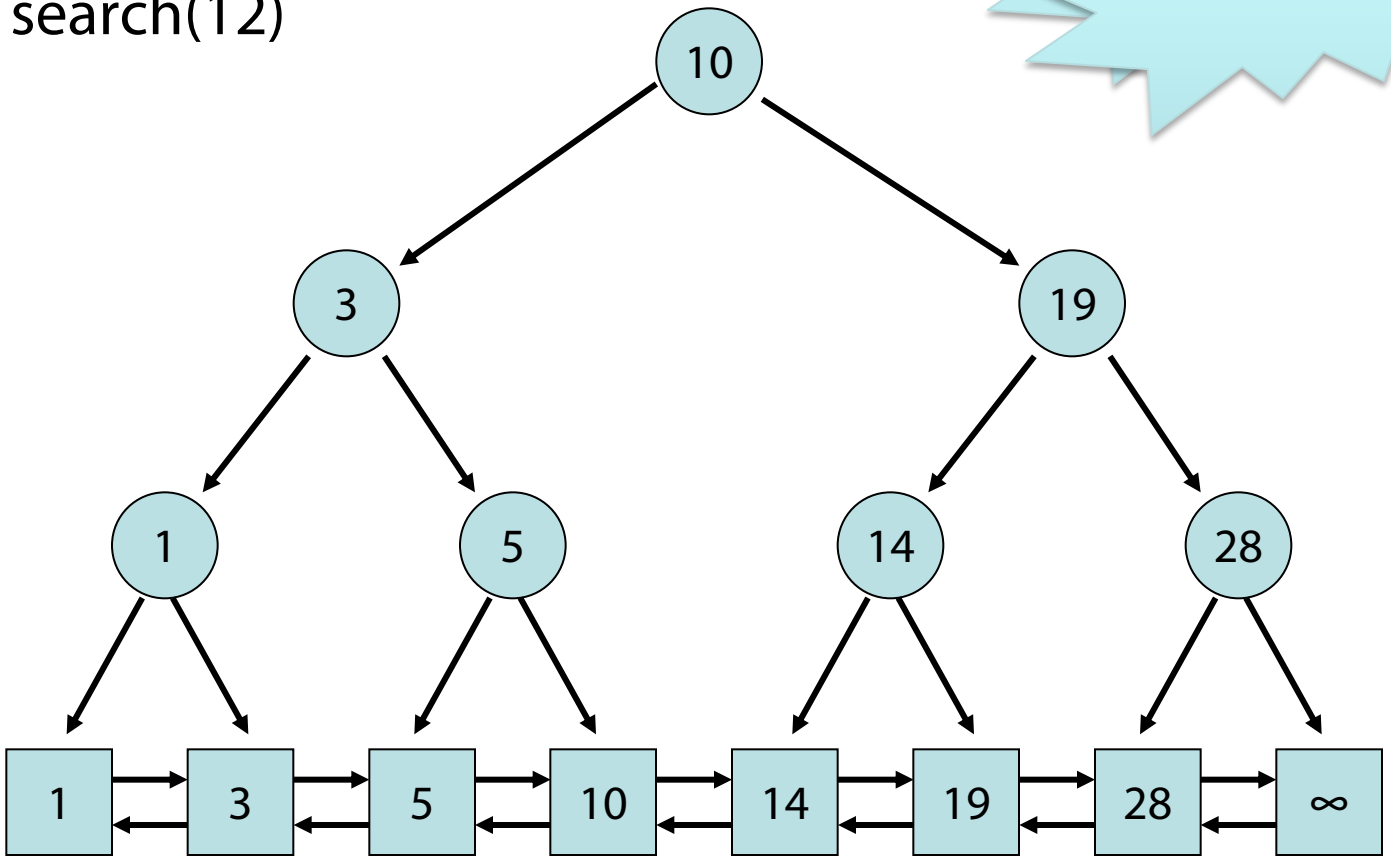


Zyklus nicht gezeigt

Binärer Suchbaum (ideal)

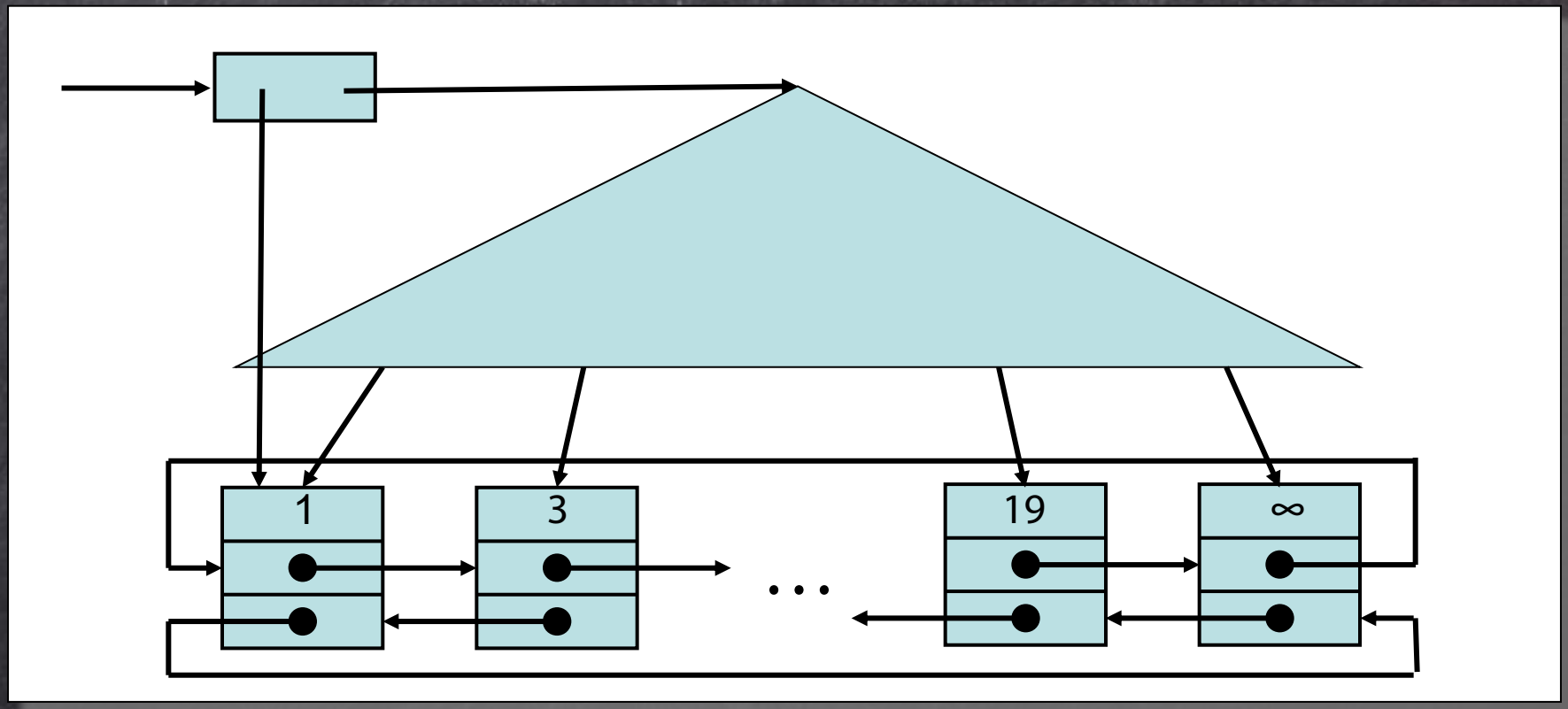
Geordnete Menge?

search(12)



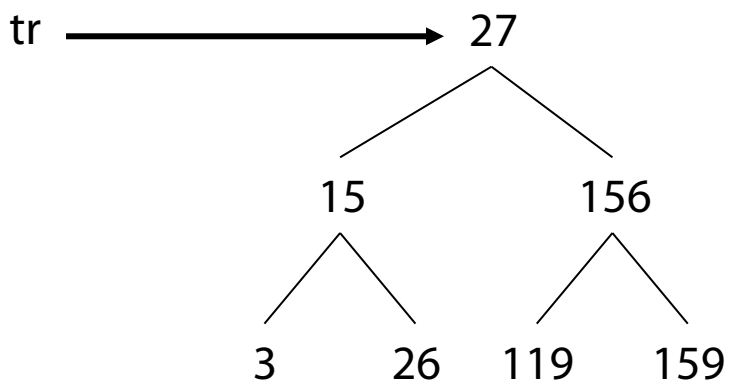
Aufgabe: Iteration über alle Elemente?

- Wie Elemente addieren?
- Wie Maximum/Minimum bestimmen?
- K-Kleinstes Element?



Aufgabe: Iteration auch über Baumknoten?

- Wie Elemente addieren?
- Wie Maximum/Minimum bestimmen?




```
fold(+, tr, 0)
```
→ 505

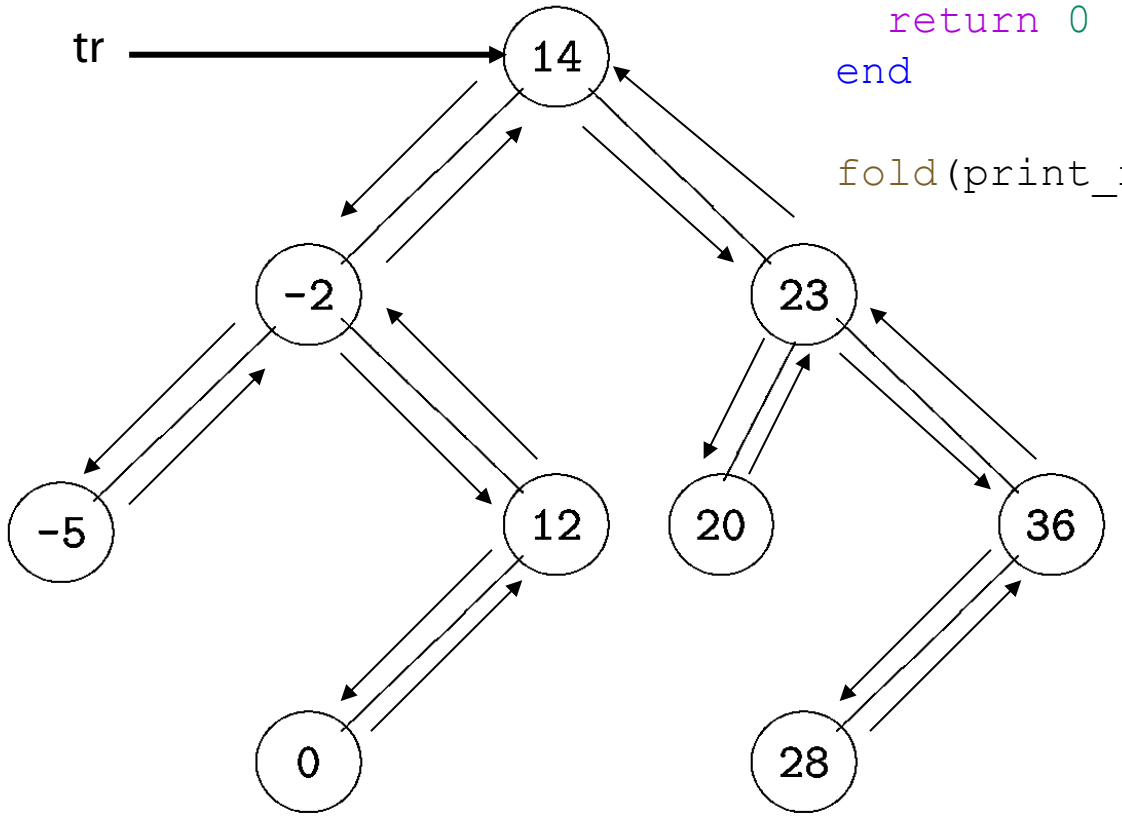
```
function max(x, y)
  if x > y return x
  else return y
  end
end
```

```
function min(x, y)
  if y > x return x
  else return y
  end
end
```

```
fold(max,tr,0)
```
→ 159

```
fold(min,tr,∞)
```
→ 3

Inorder-Ausgabe ergibt Sortierreihenfolge




```
function print_node(_, x)
  print(x)
  return 0
end

fold(print_node, tr, 0)
```

Können Sie fold für Bäume so realisieren, dass das klappt?

-5 -2 0 12 14 20 23 28 36

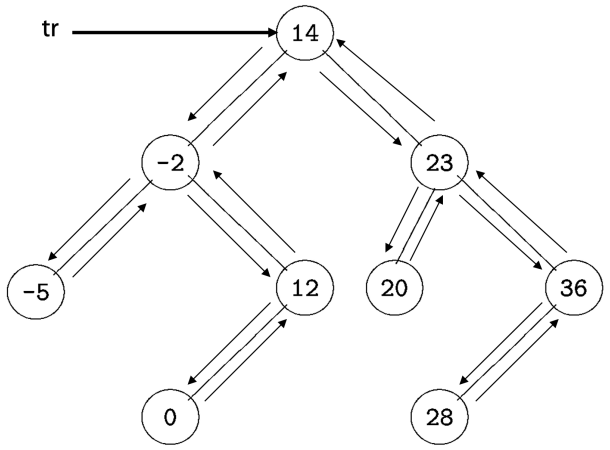
Aufgabe: Fold für Bäume

```
function fold(f, v, init)
  if empty_tree(v)
    return init
  end
  if is_leaf(v)
    return f(init, v.key)
  end
  if left_exists(v)
    x = f(fold(f, v.left, init), v.key)
    if right_exists(v)
      return fold(f, v.right, x)
    else
      return x
    end
  else # linker Nachfolger existiert nicht,
       rechter ja, sonst leaf
    return fold(f, v.right, f(init, v.key))
  end
end
```

`v.key` liefert Knotenbeschriftung als Komponente von einem TreeNode

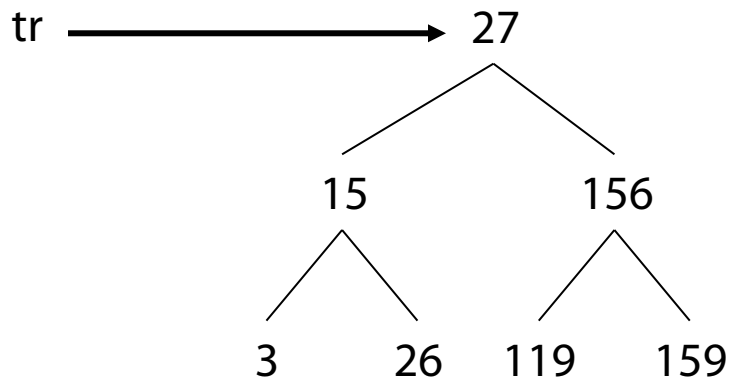
```
function print_node(_, x)
  print(x)
  return 0
end

fold(print_node, tr, 0)
```



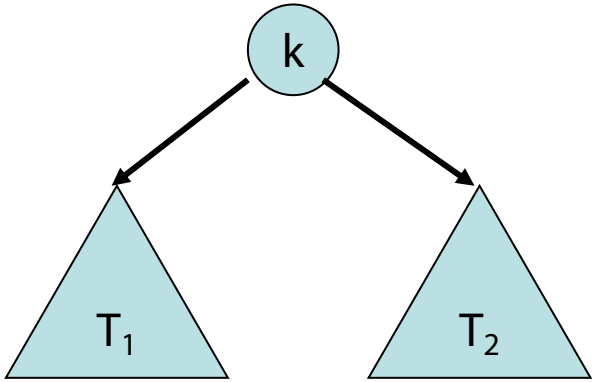
Aufgabe: Iteration auch über Baumknoten?

- K-Kleinstes Element bestimmen?
- Mit fold?



Binärer Suchbaum

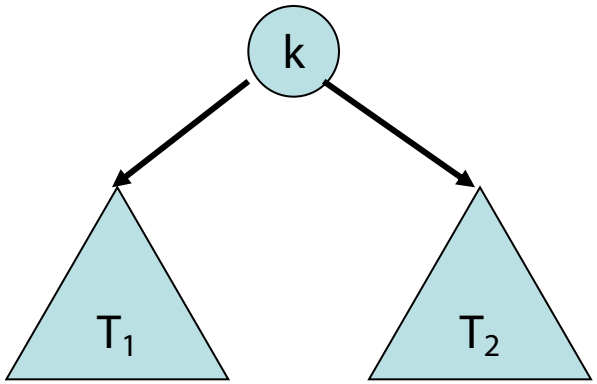
Suchbaum-Regel:



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

- Damit lässt sich die search Operation für Mengen 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 als Navigationsstruktur

Für einen Baumknoten v sei

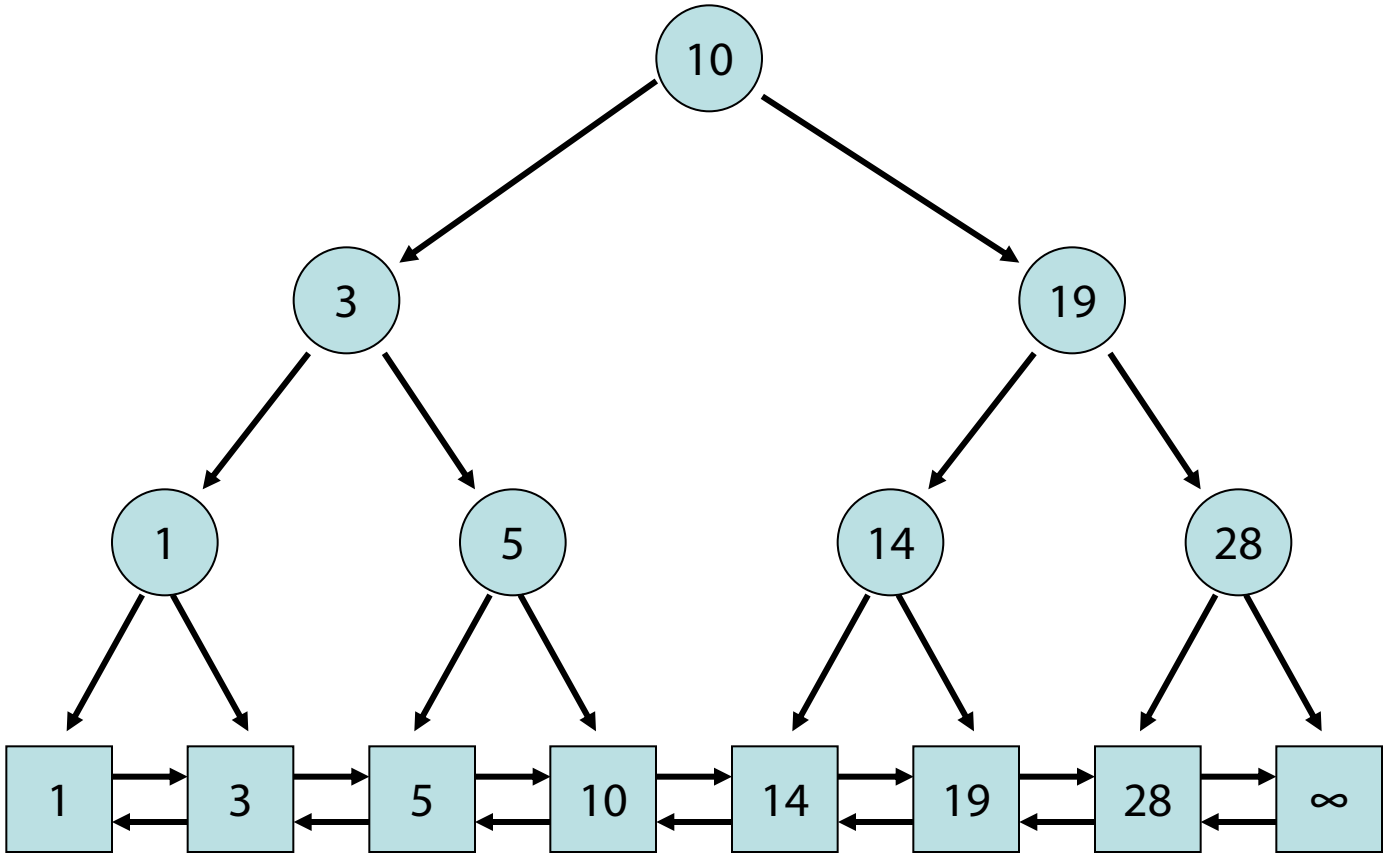
- key(v) der Schlüssel in v
- d(v) die Anzahl Kinder von v

- Suchbaum-Regel: (s.o.)

- Grad-Regel: (Grad = Anzahl der Kinder)
  Alle Baumknoten v haben zwei Kinder: d(v) = 2
  (wenn #Elemente =0 keine Baumknoten vorhanden)
  Unten sind die Knoten der verketteten Liste.

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

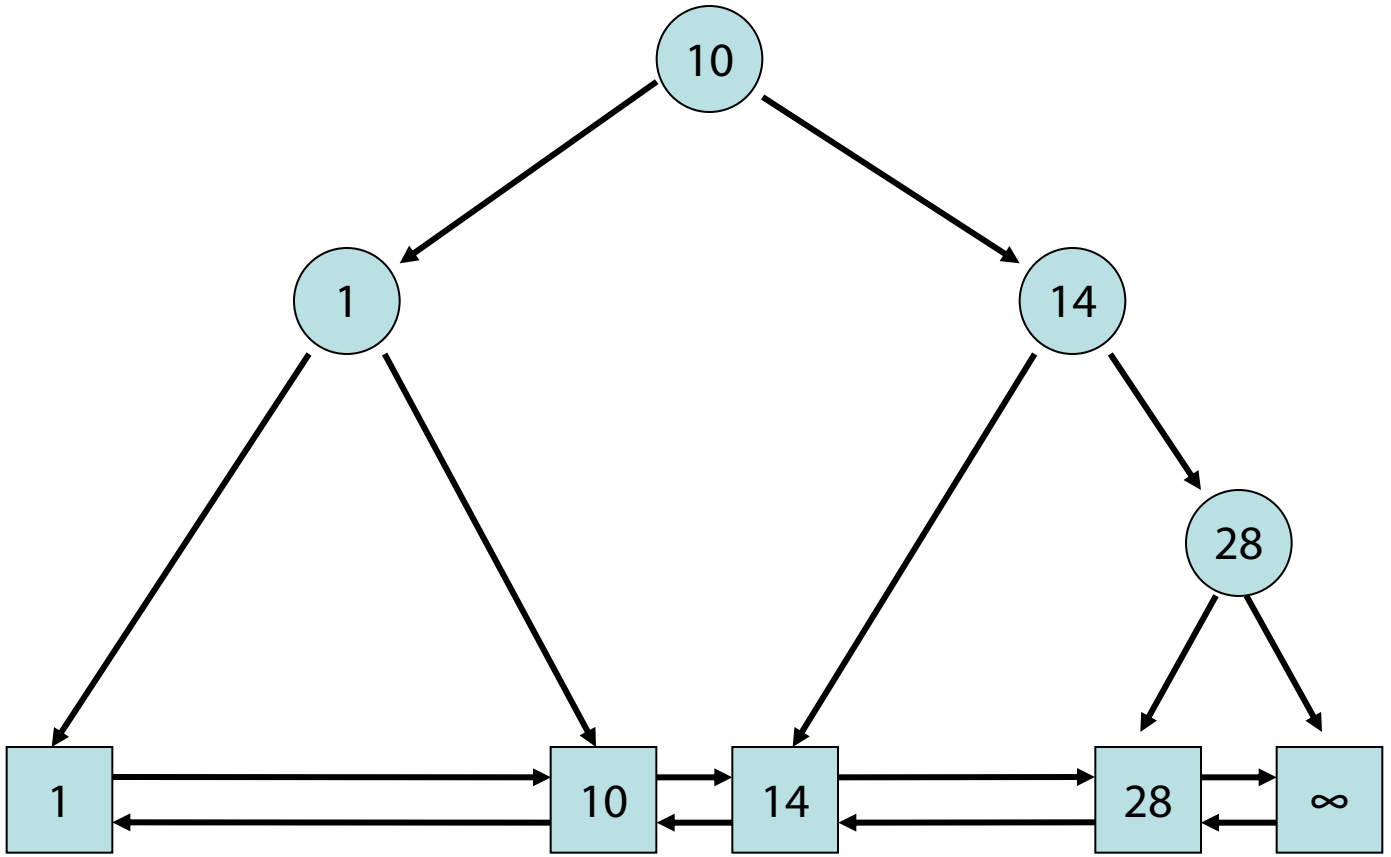
Search(9)



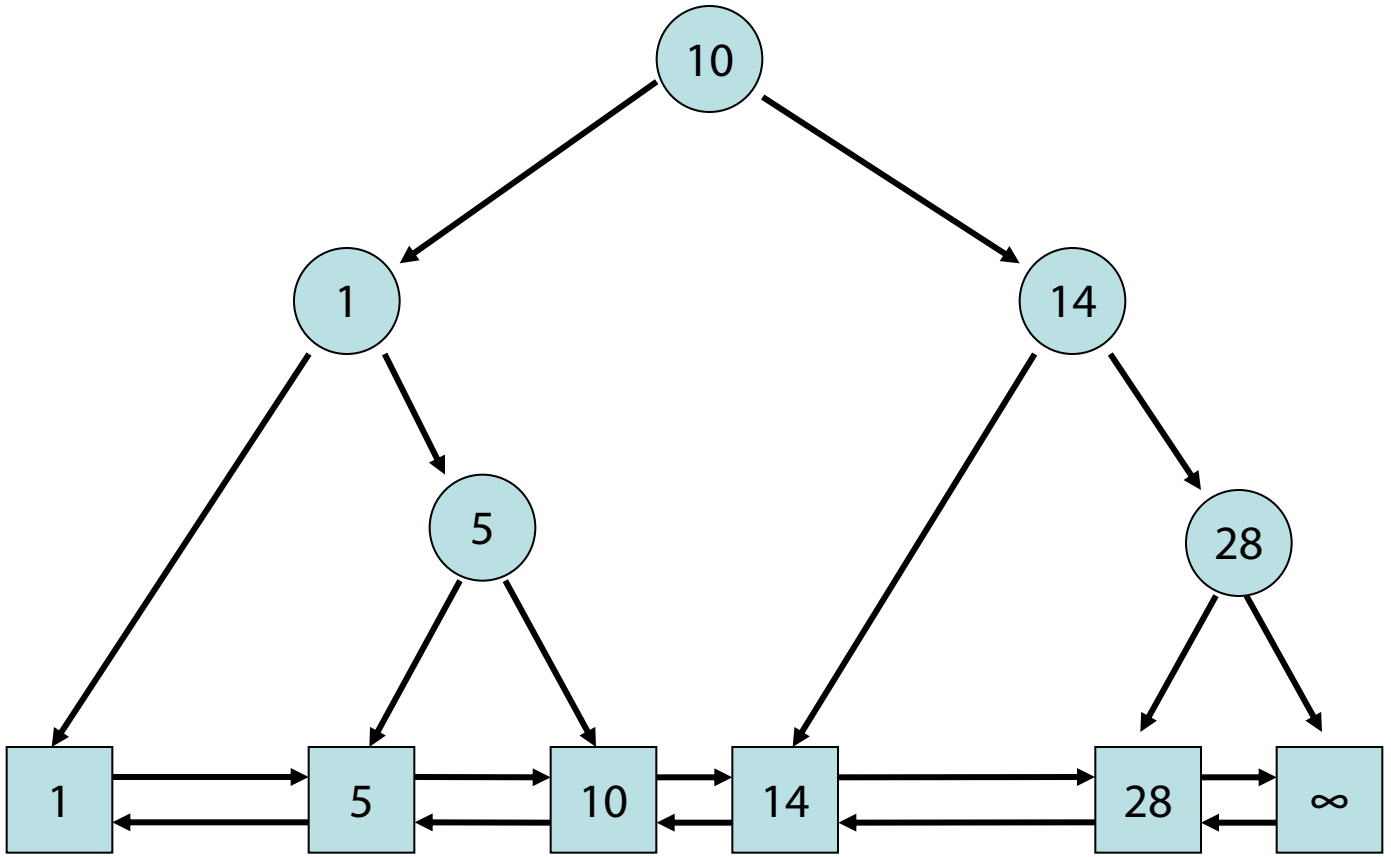
Insert Operation

- insert(e, s)
  Erst search(key(e), s) bis Element e' in Liste erreicht.
  Falls key (e‘) > key (e) dann:
    Füge e vor e' ein und ein neues Suchbaumblatt
    für e und e' mit key(e), so dass Suchbaum-Regel erfüllt.
  Falls key (e‘) = key (e) dann:
    Ersetze e‘ durch e (keine Duplikate)

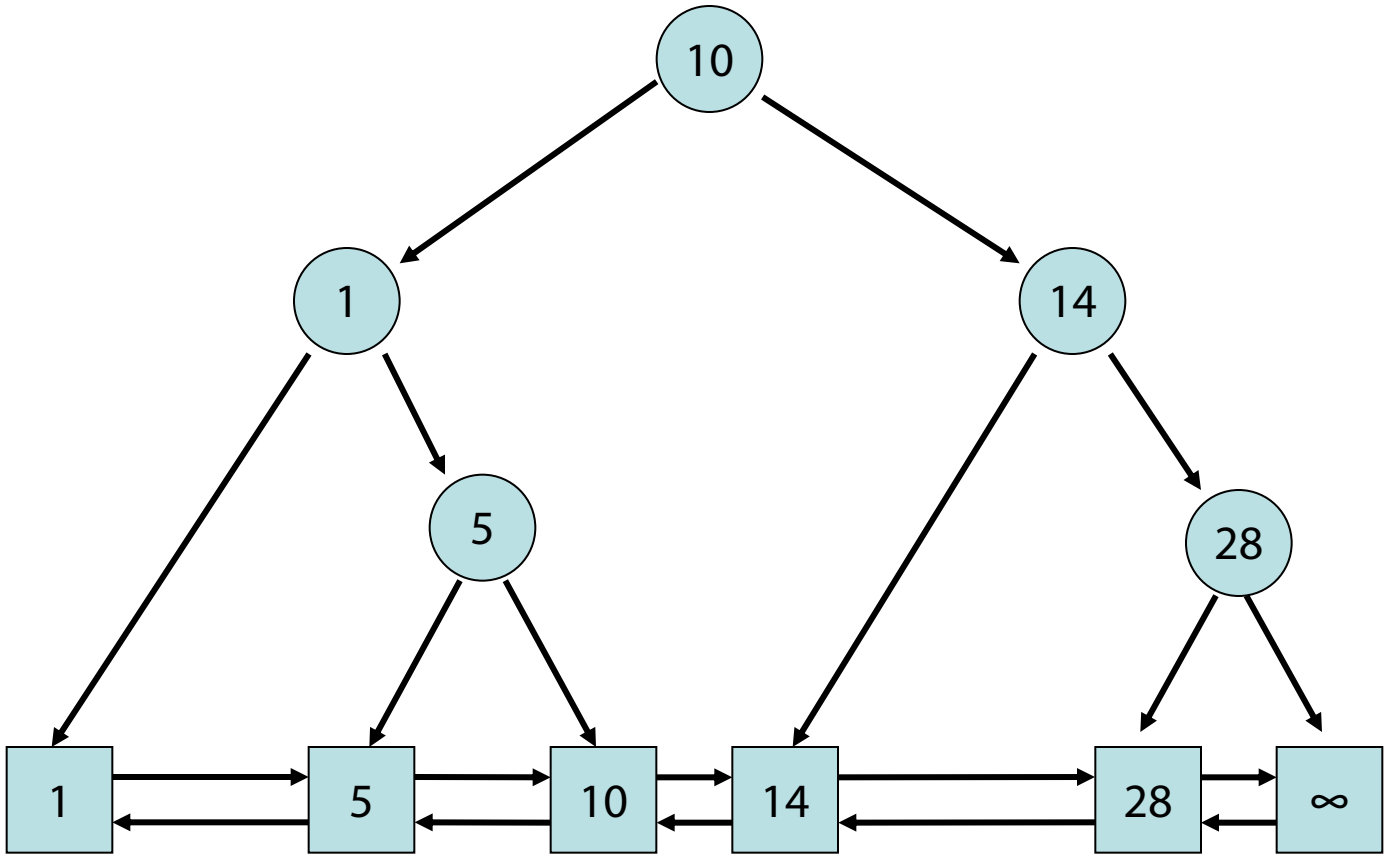
Insert(5)



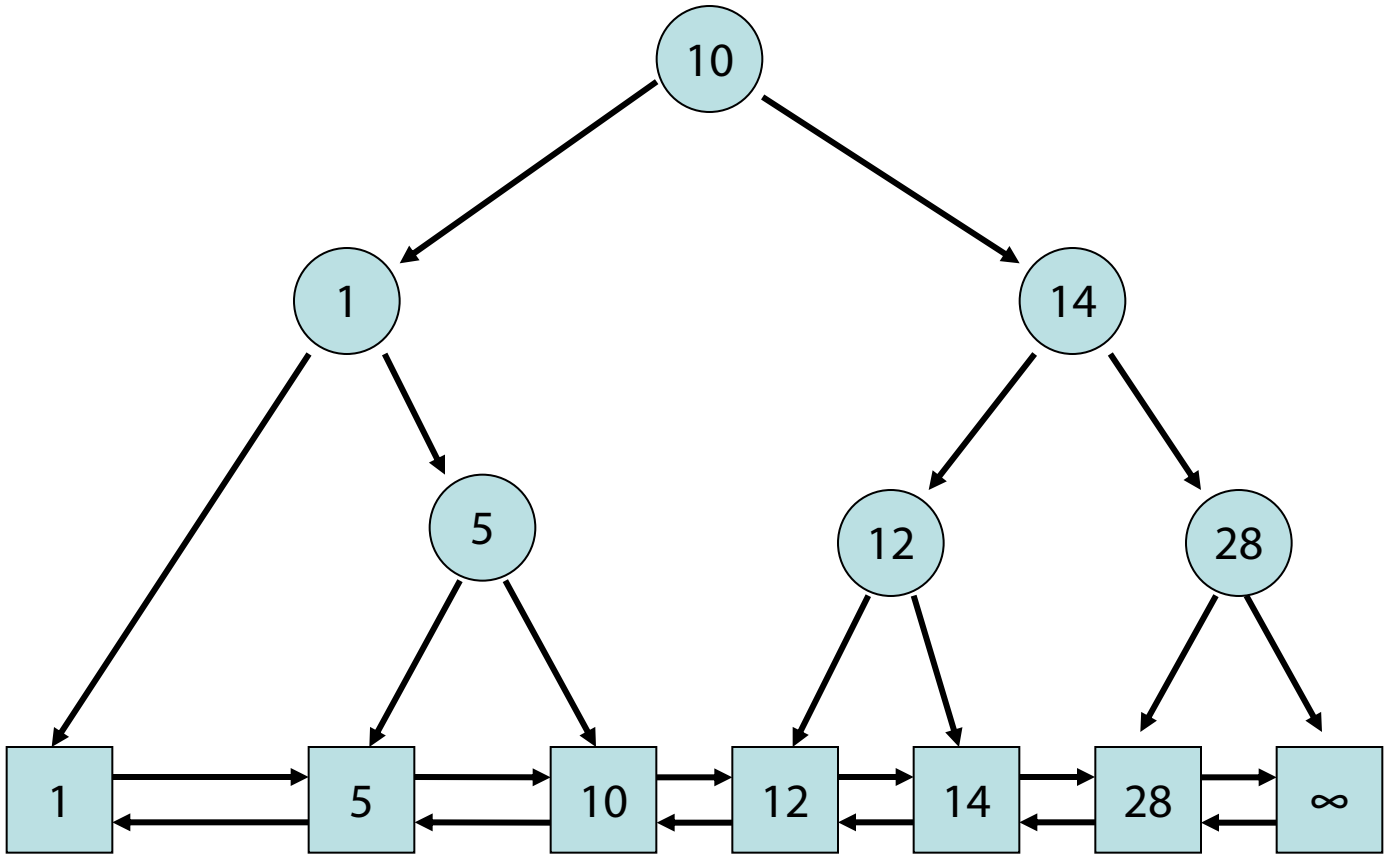
Insert(5)



Insert(12)



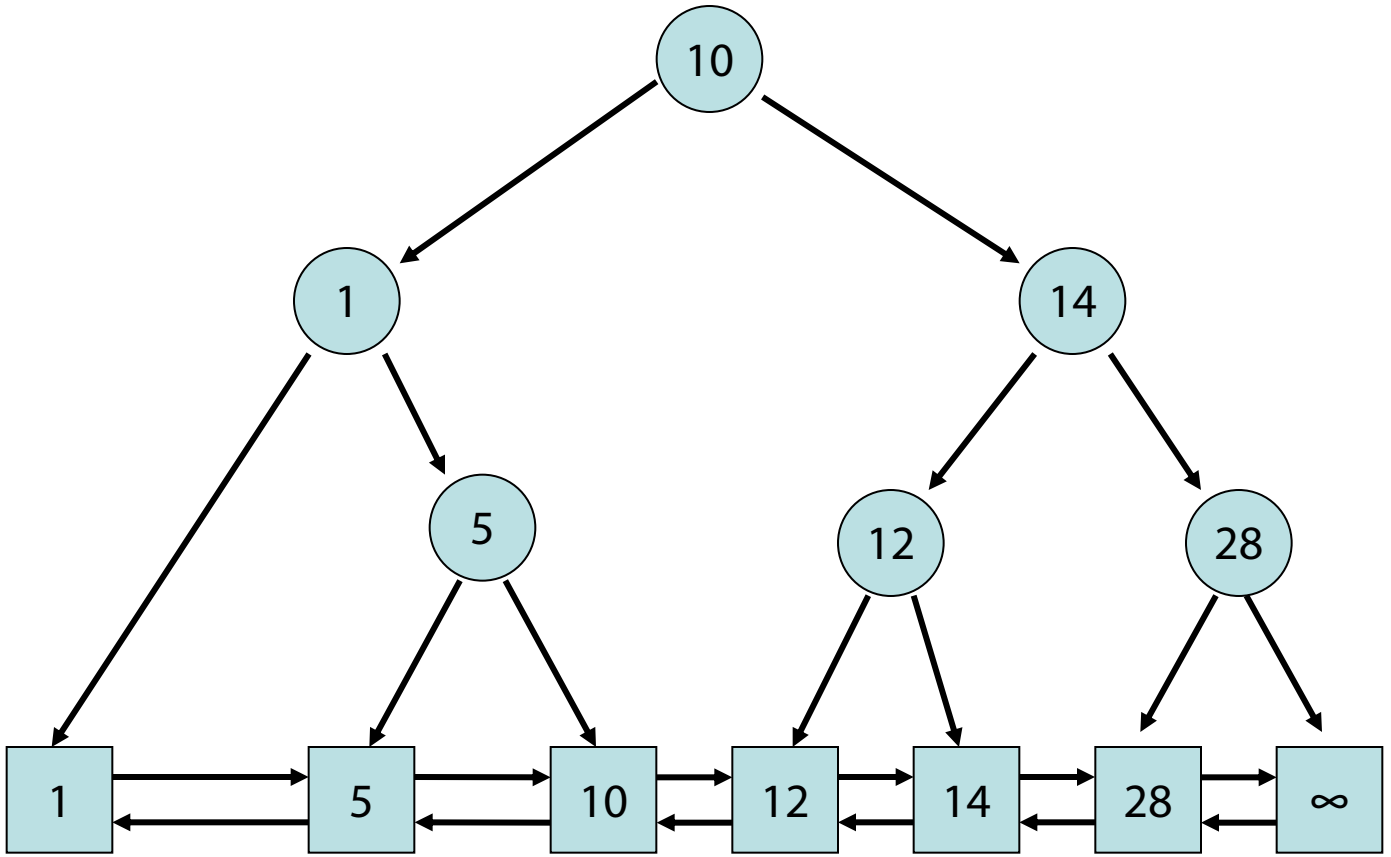
Insert(12)



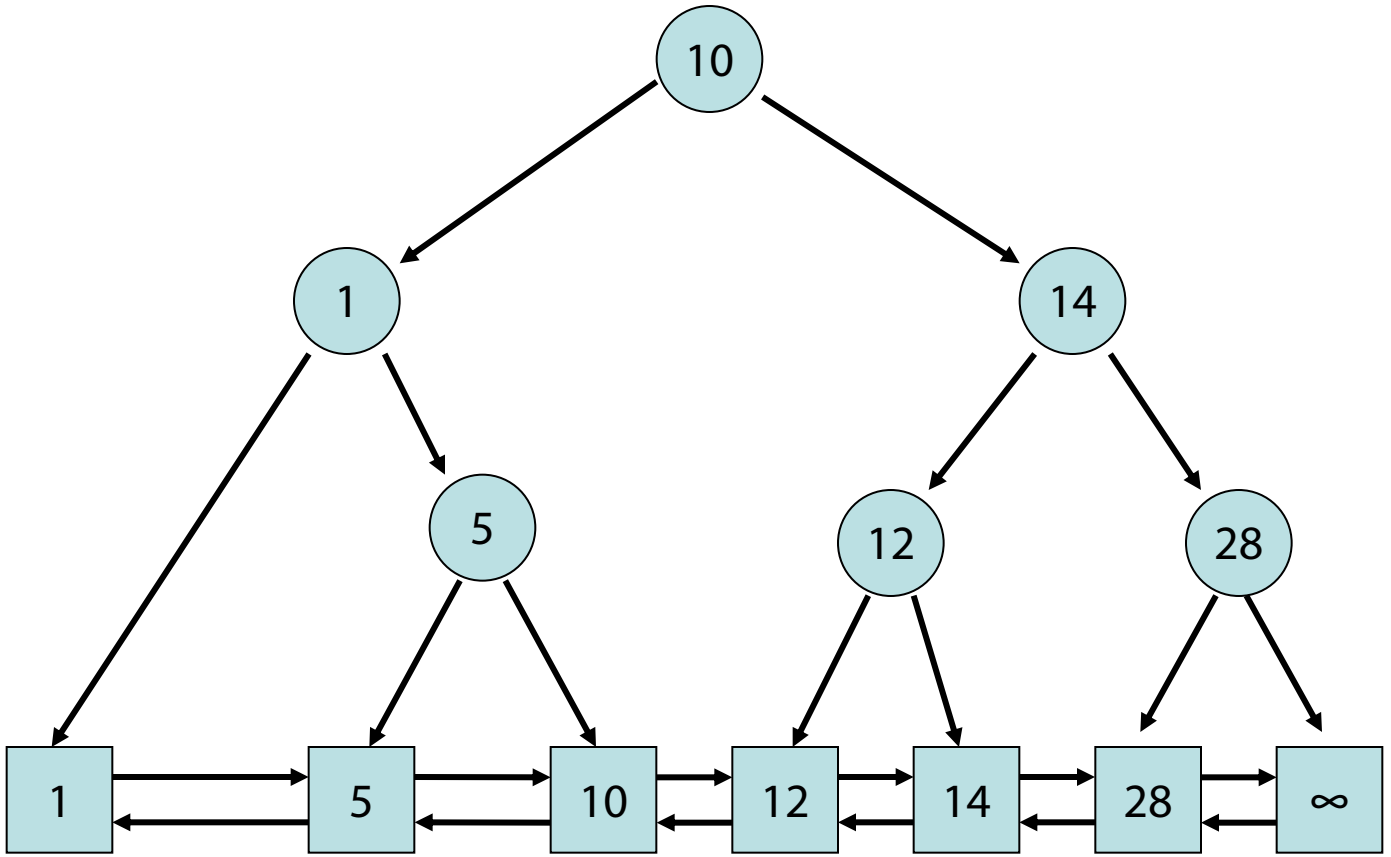
Delete Operation

- delete(k, s)
  Erst search(k, s) bis ein Element e in Liste erreicht.
  Falls key (e) == k, lösche e aus Liste und Vater v von e aus Suchbaum, und setze den Baumknoten w mit key (w) == k: w.key = key(v)

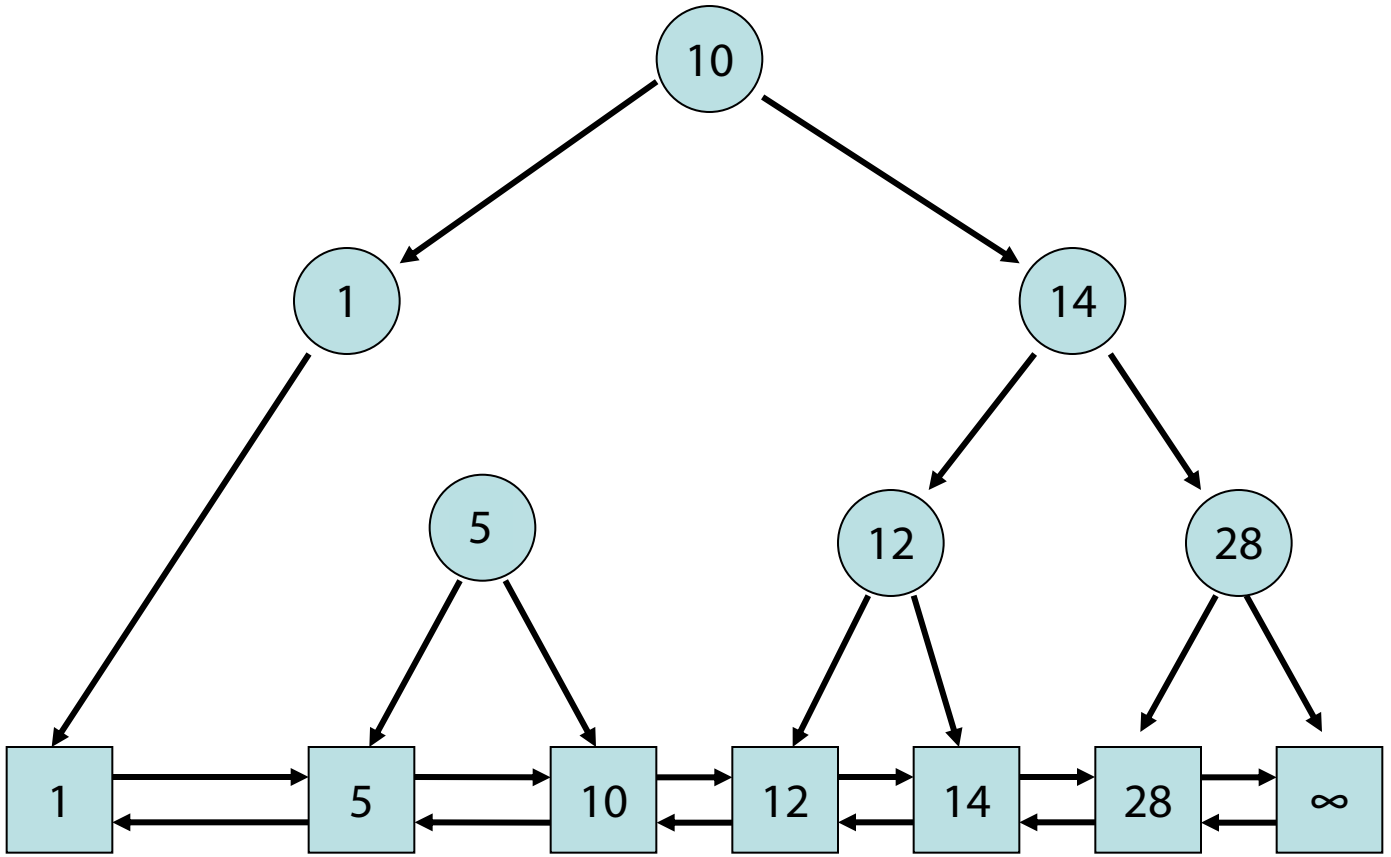
Delete(1)



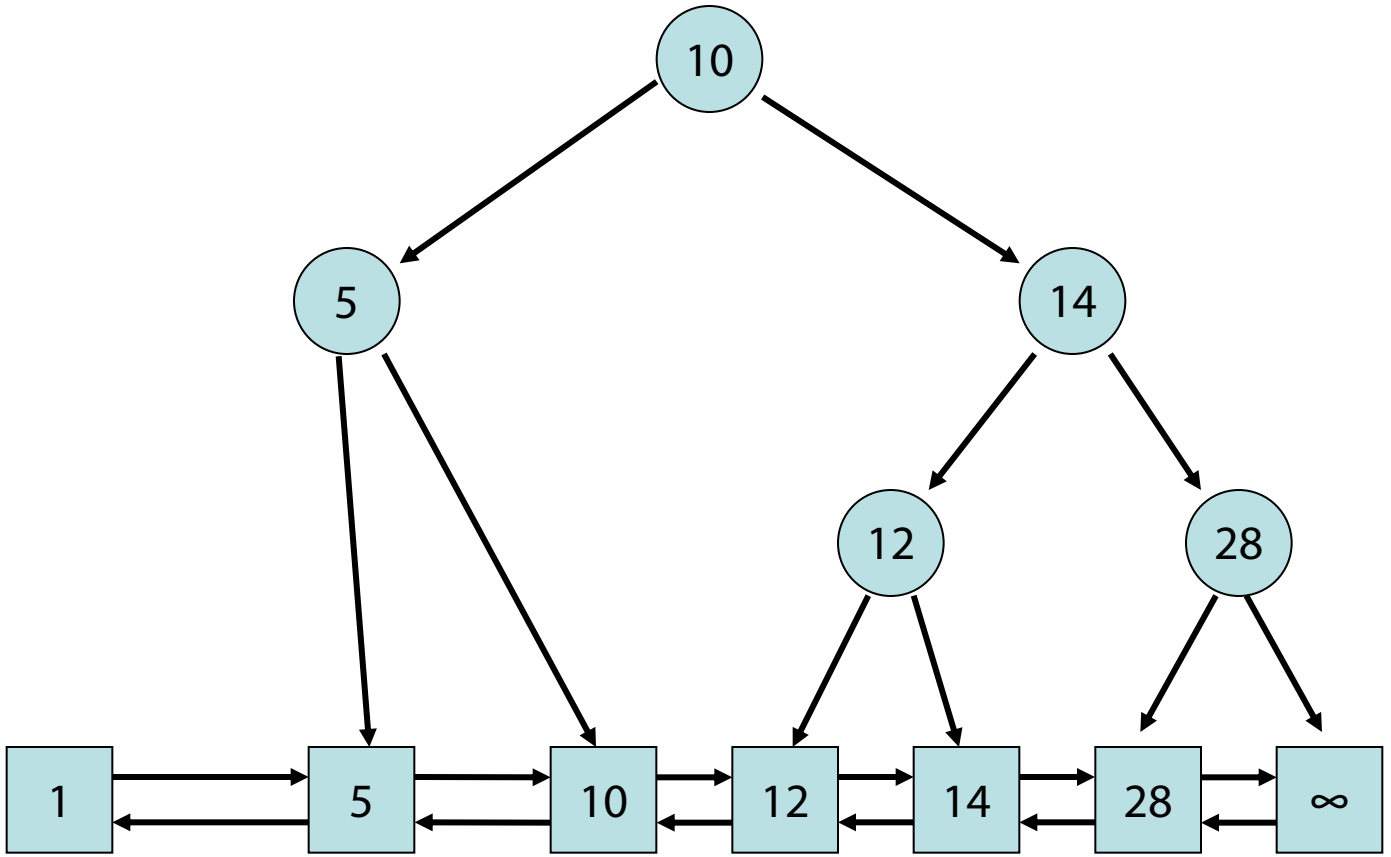
Delete(1)



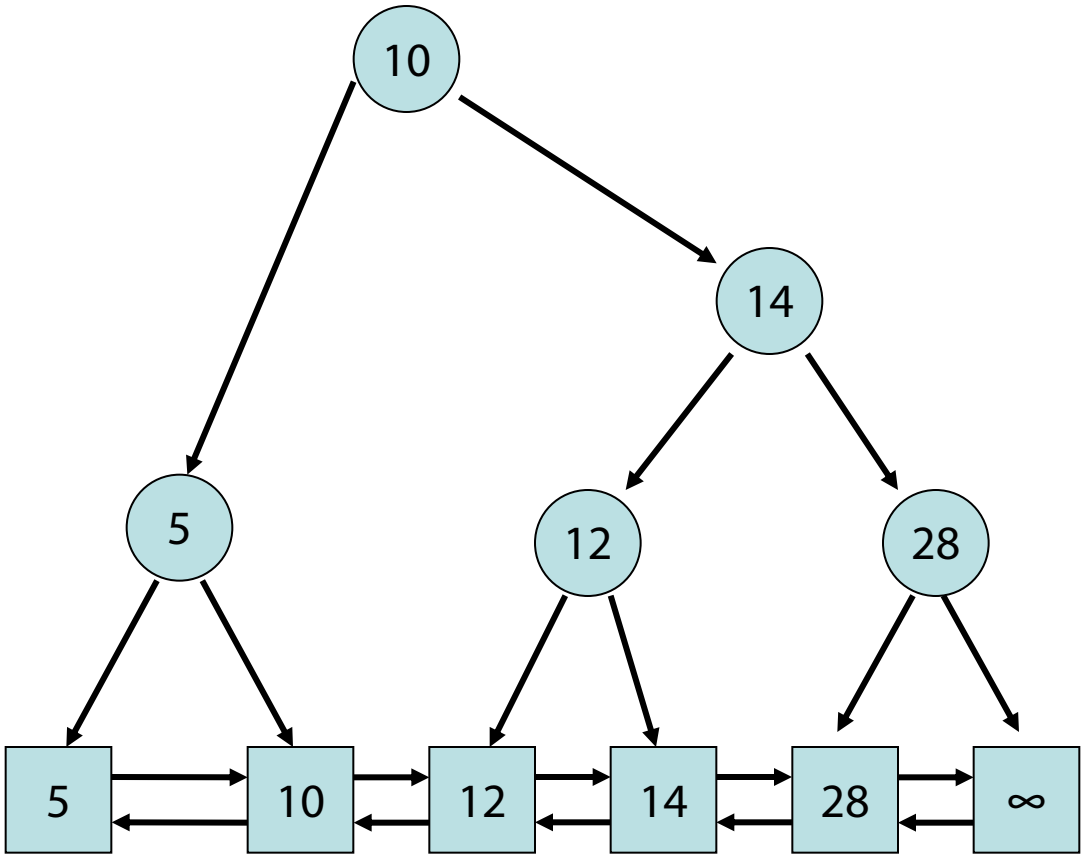
Delete(1)



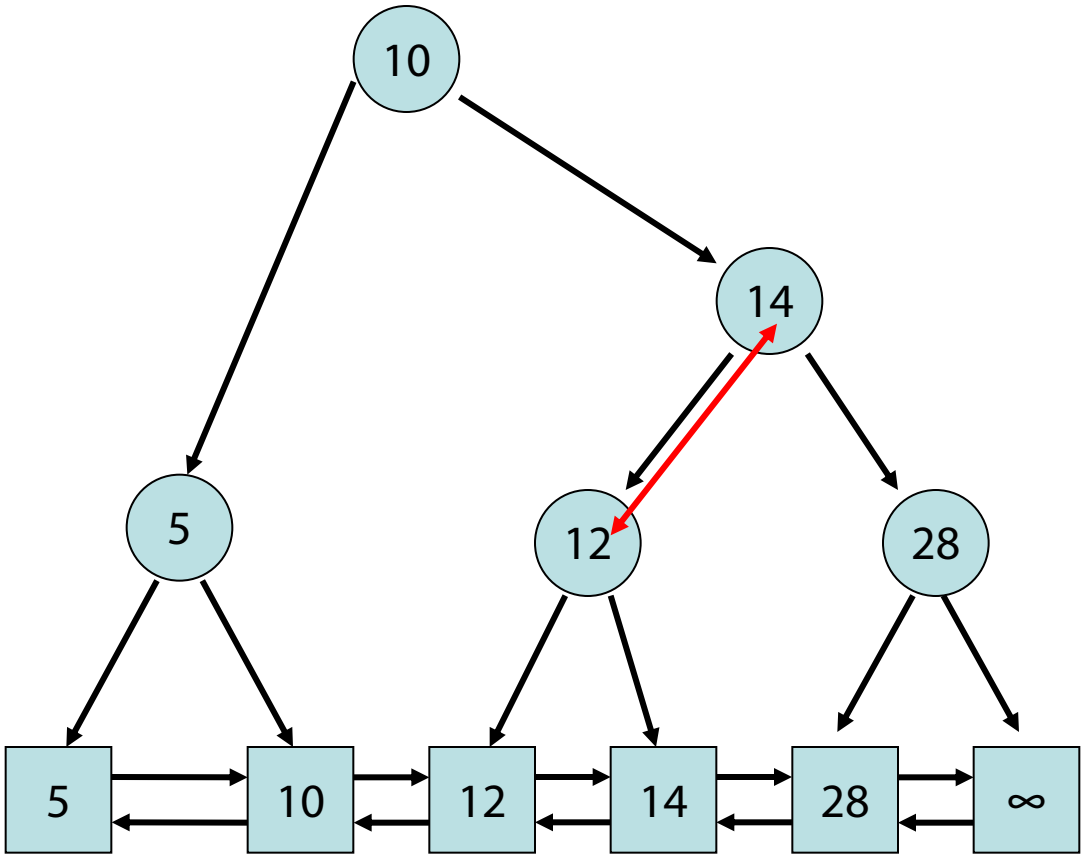
Delete(1)



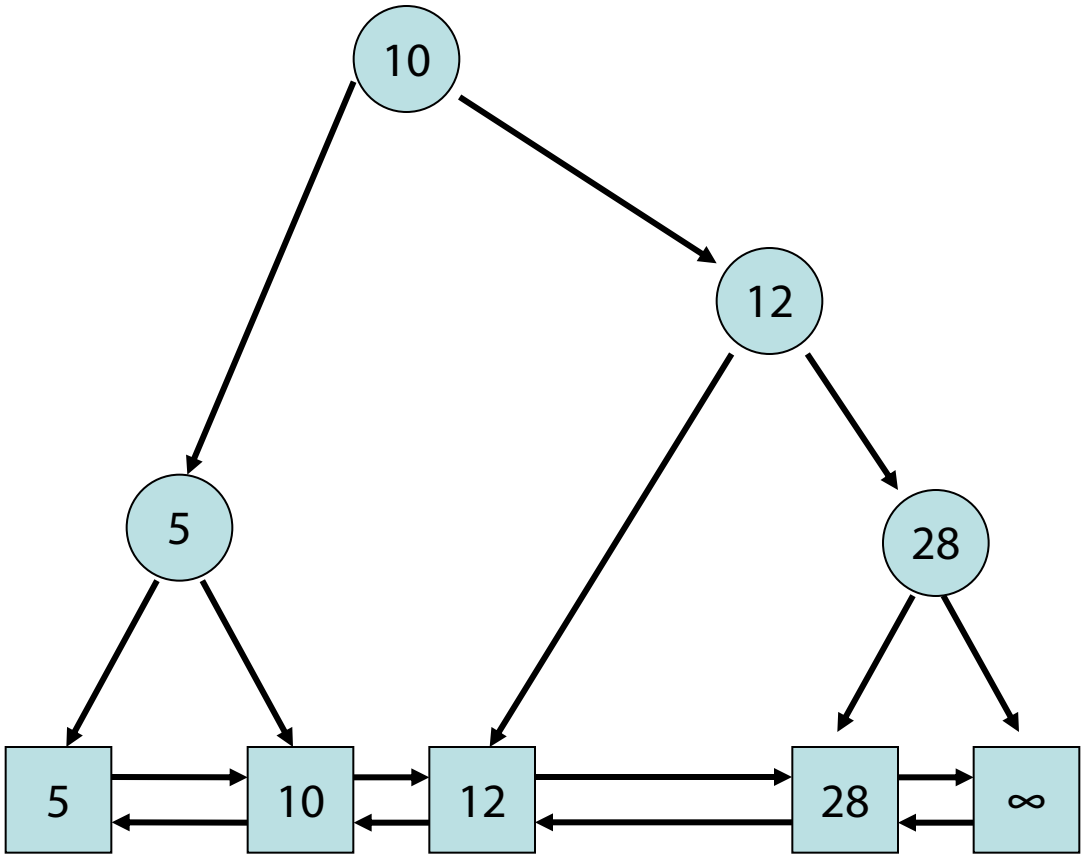
Delete(1)



Delete(14)



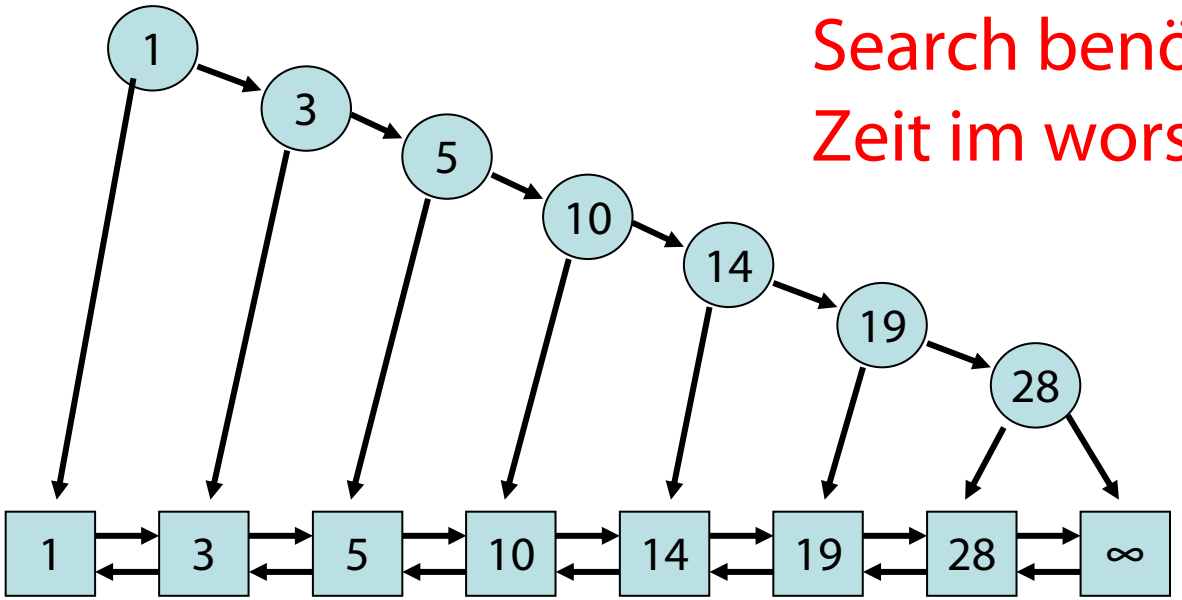
Delete(14)



Entarteter Binärbaum

Problem: Binärbaum kann bei bestimmter Einfügereihenfolge in entarteter Form aufgebaut werden!

Beispiel: Zahlen werden in sortierter Folge eingefügt



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

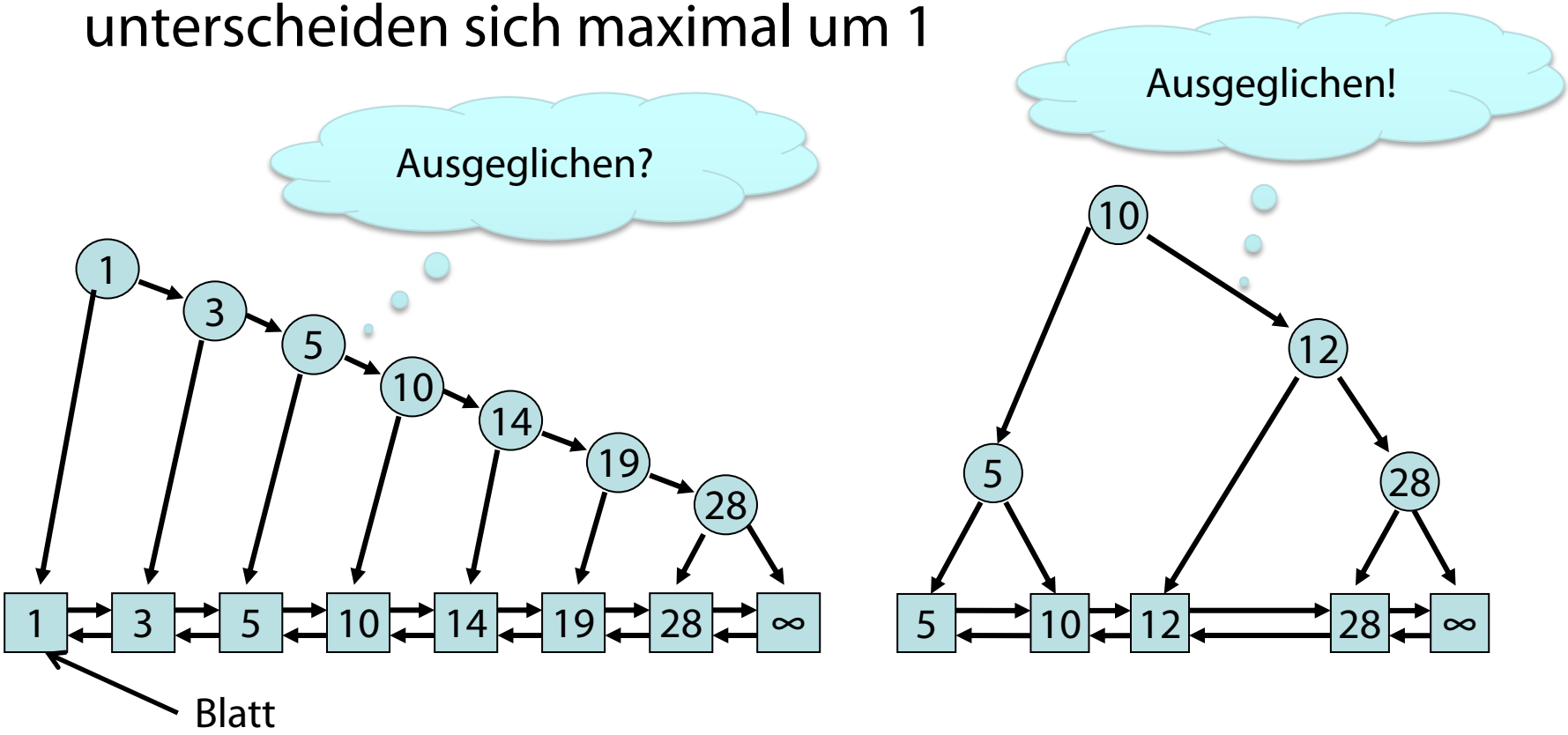
Automatische Umstrukturierung

Optionen

- Anpassung bei Anfragen (Operationen search oder test)
- Anpassung beim Einfügen (insert) oder Löschen (delete) neuer Elemente

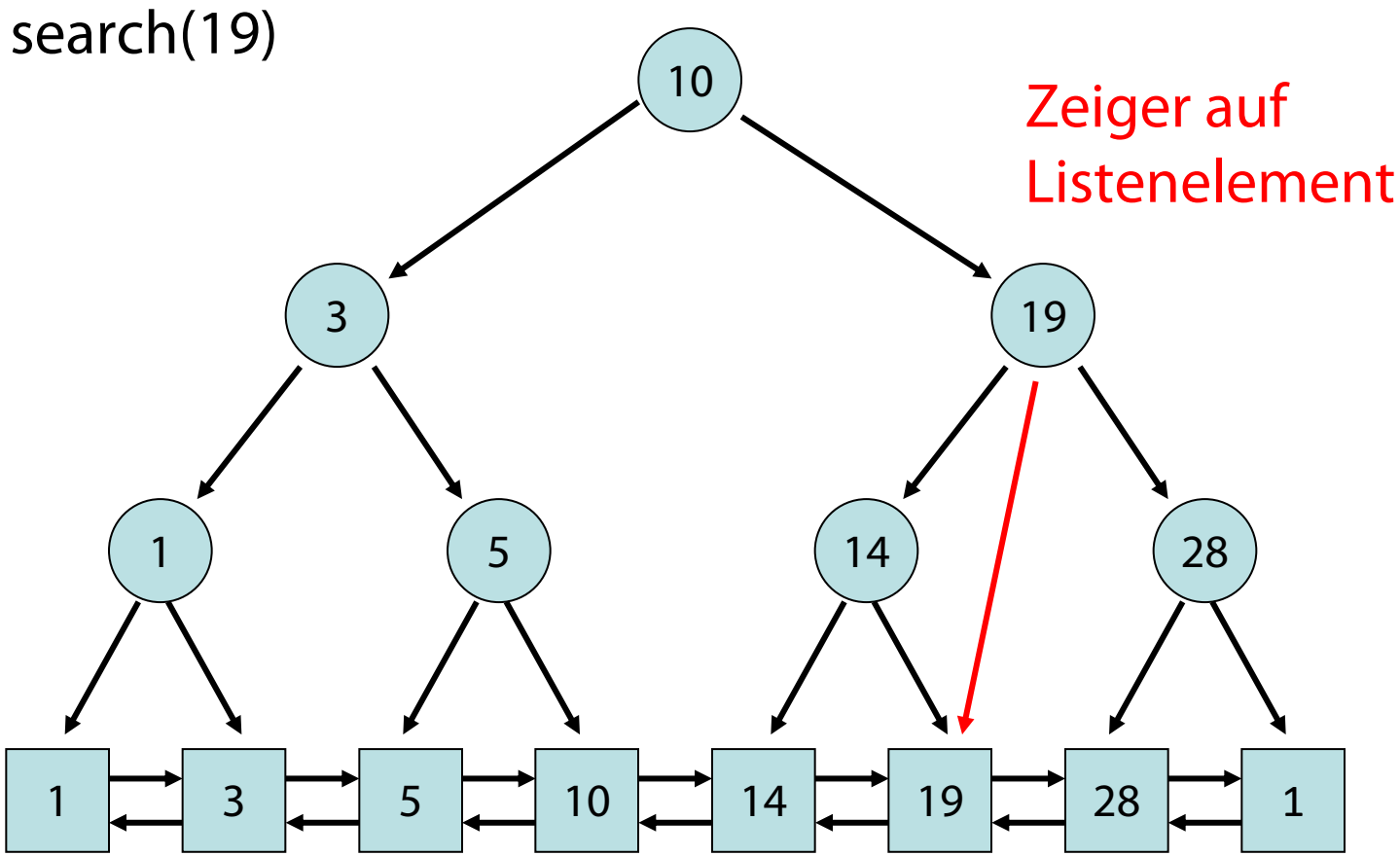
Definition: Ausgeglichener Suchbaum

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



- „Alle Ebenen bis auf Blattebene voll gefüllt“

Ein weiterer Zeiger pro Knoten…



Navigationsbäume mit Zeigern auf Listenelemente

- 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

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 (amortisiert betrachtet dann insgesamt $O(\log n)$)
- 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**

Danksagung

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus:

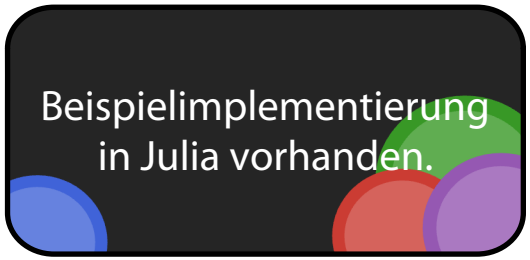
- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 2: Suchstrukturen) gehalten von Christian Scheideler an der TUM
  http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

Splay-Baum

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

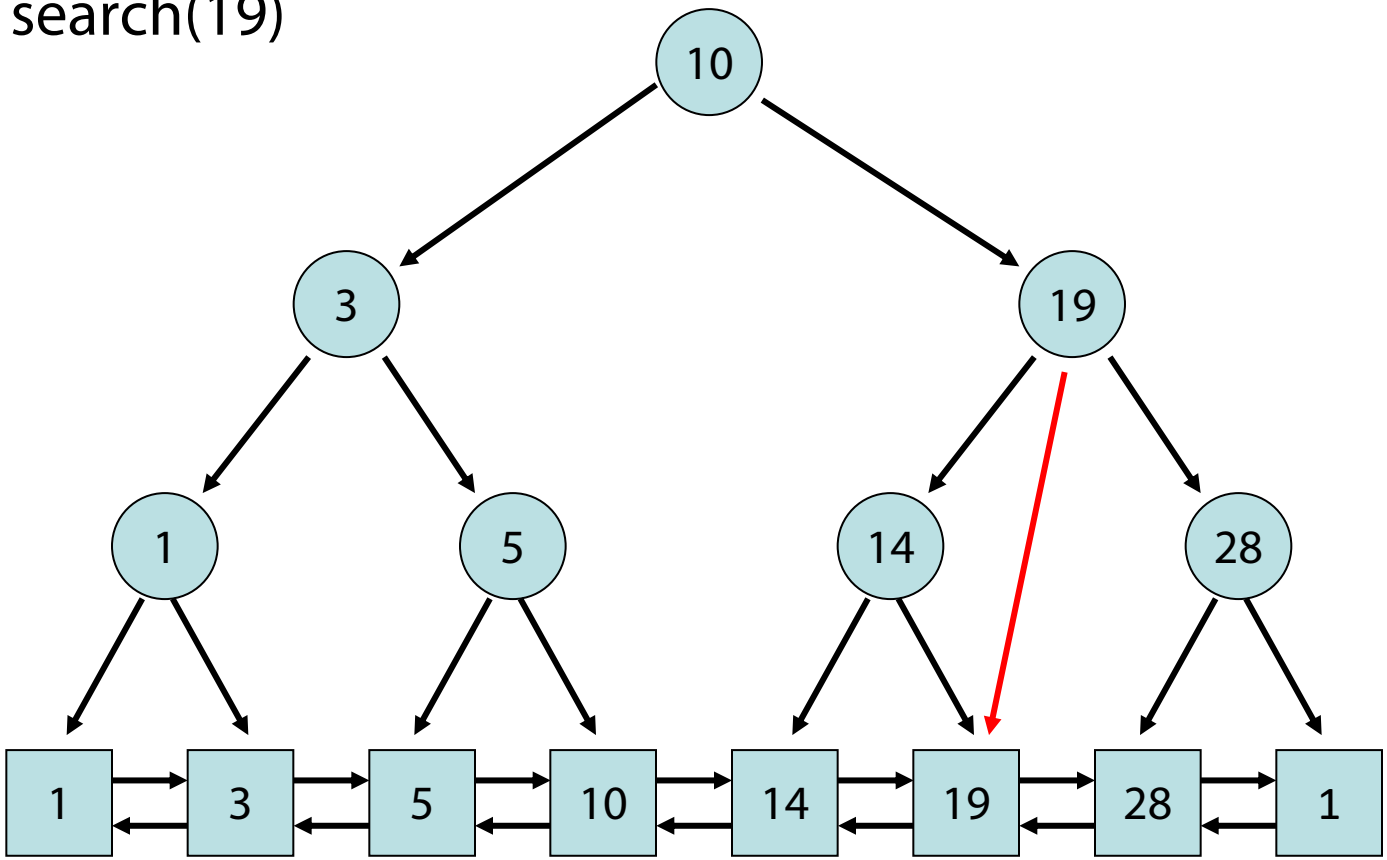
Hier: Implementierung als externer Suchbaum (wie beim binären Navigationsbaum oben)

Modifikation bei Anfragen



Splay-Baum

search(19)



Splay-Baum

Idee:

Bewege Schlüssel vom zugegriffenem Element zur Wurzel

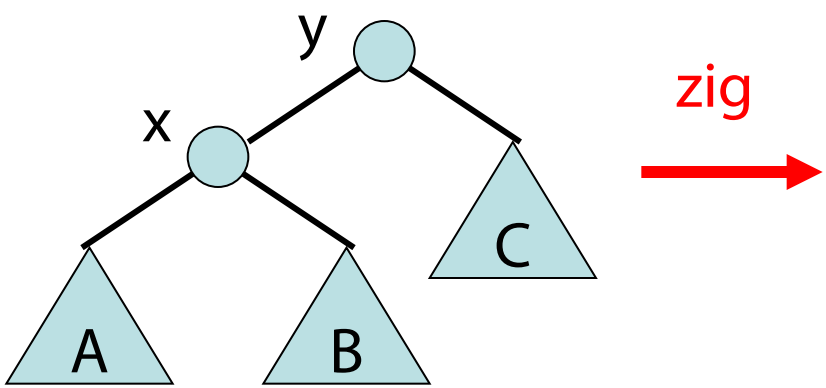
- Können wir einfach nach oben sieben wie beim Heap?
  - Nein
- Wie dann Bewegung zur Wurzel realisieren?
  - Idee: über sog. Splay-Operation

Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

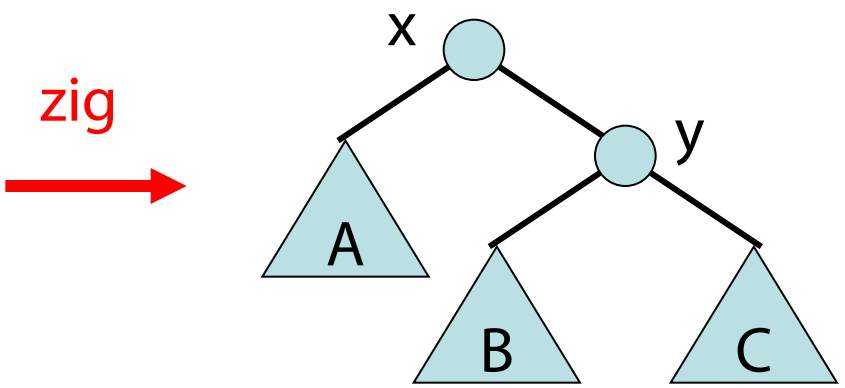


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

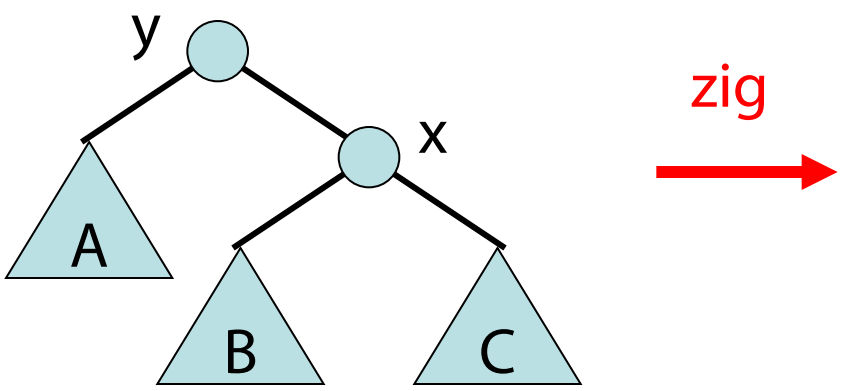


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1b. x ist Kind der Wurzel:

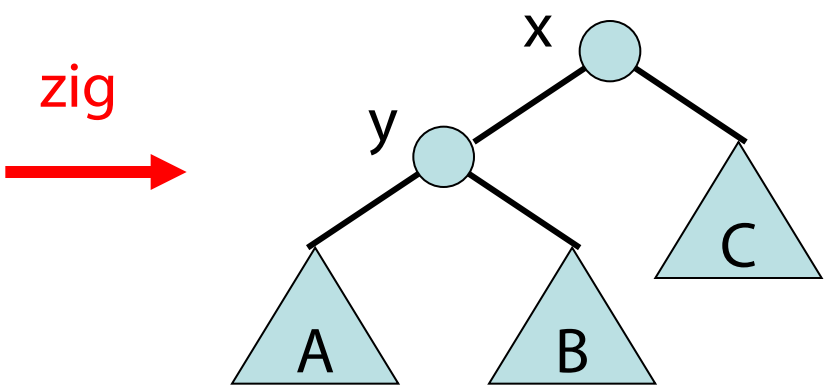


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

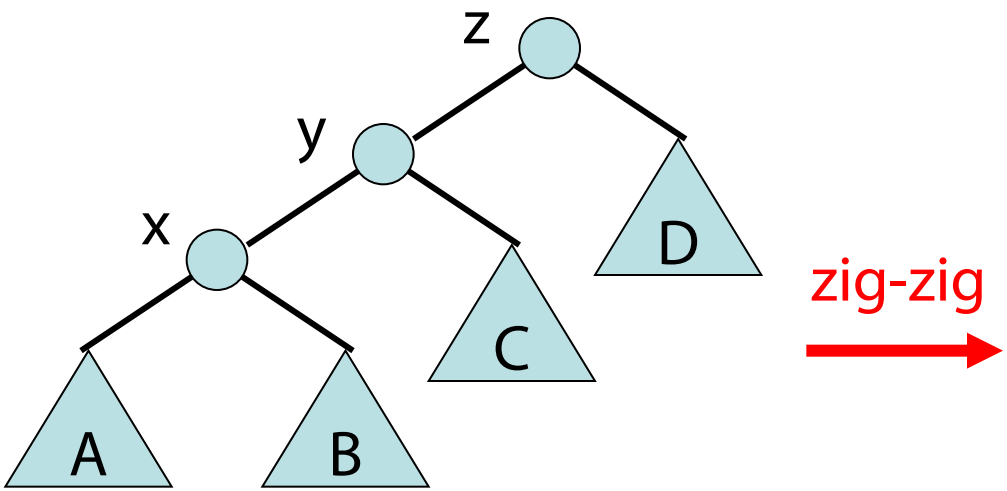
1b. x ist Kind der Wurzel:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

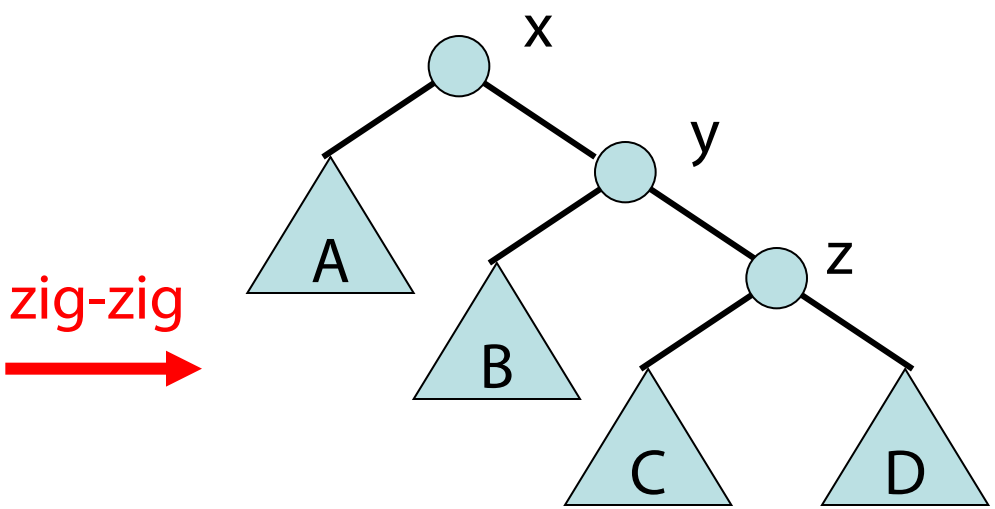
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

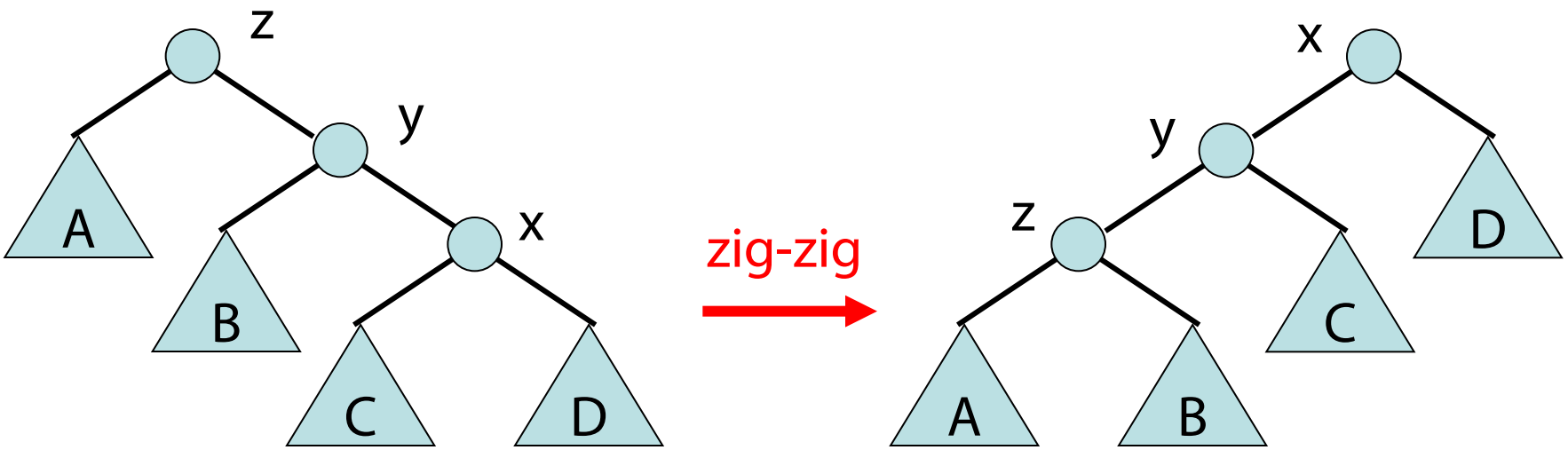
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

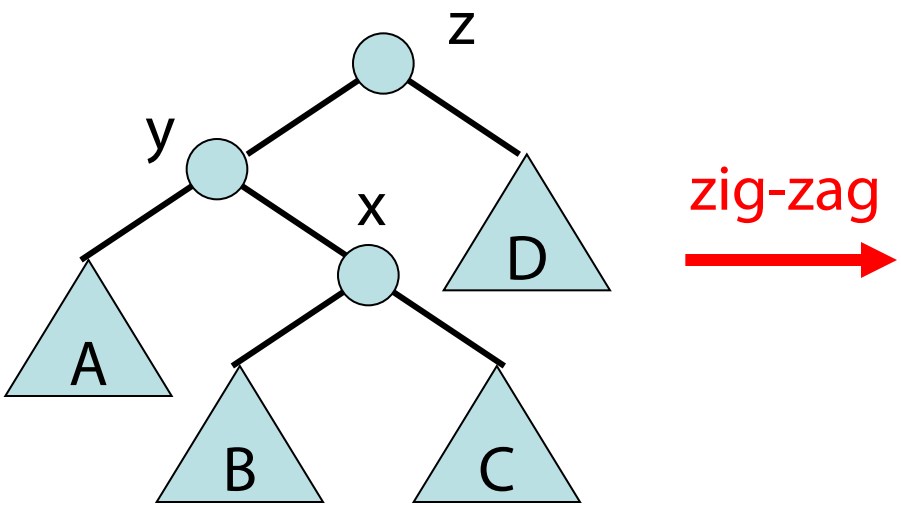
2b. x hat Vater und Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

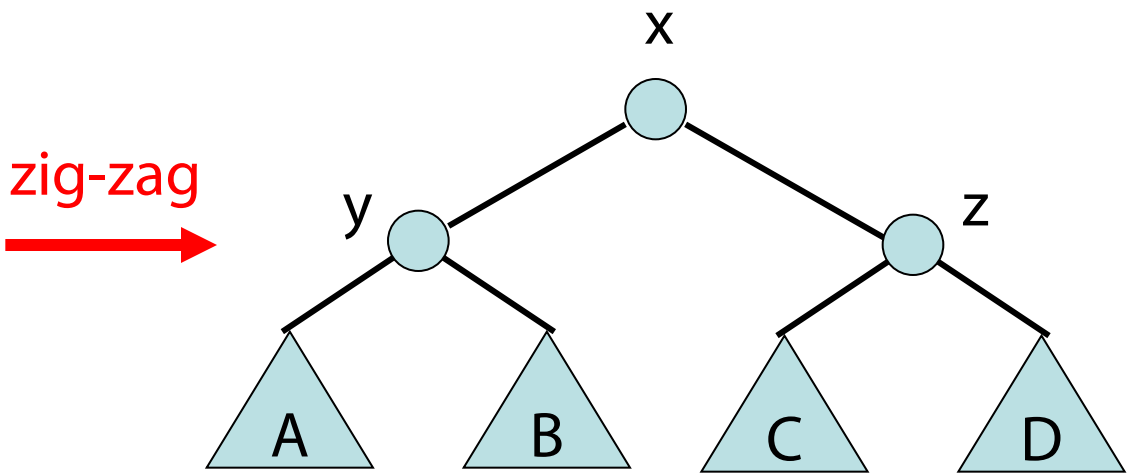
3a. x hat Vater links, Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

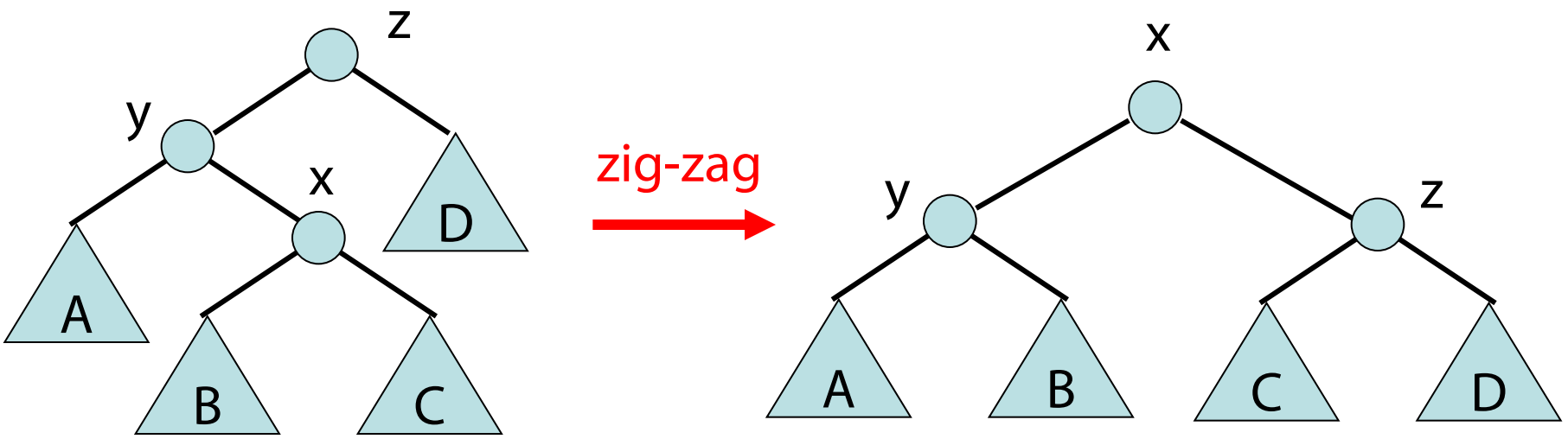
3a. x hat Vater links, Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

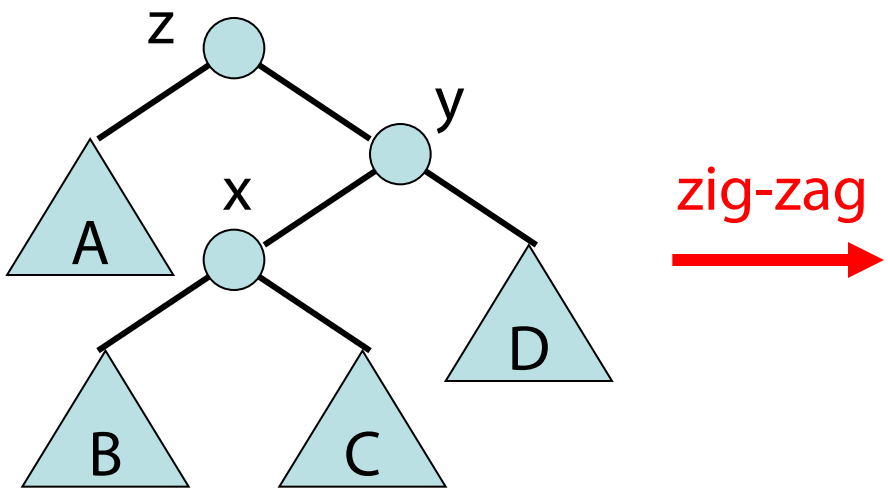
3a. x hat Vater links, Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

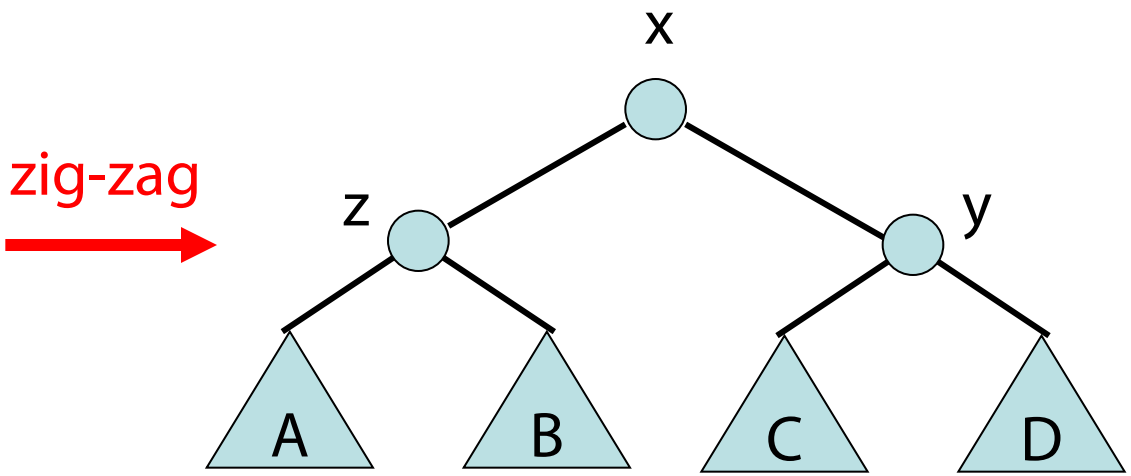
3b. x hat Vater rechts, Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

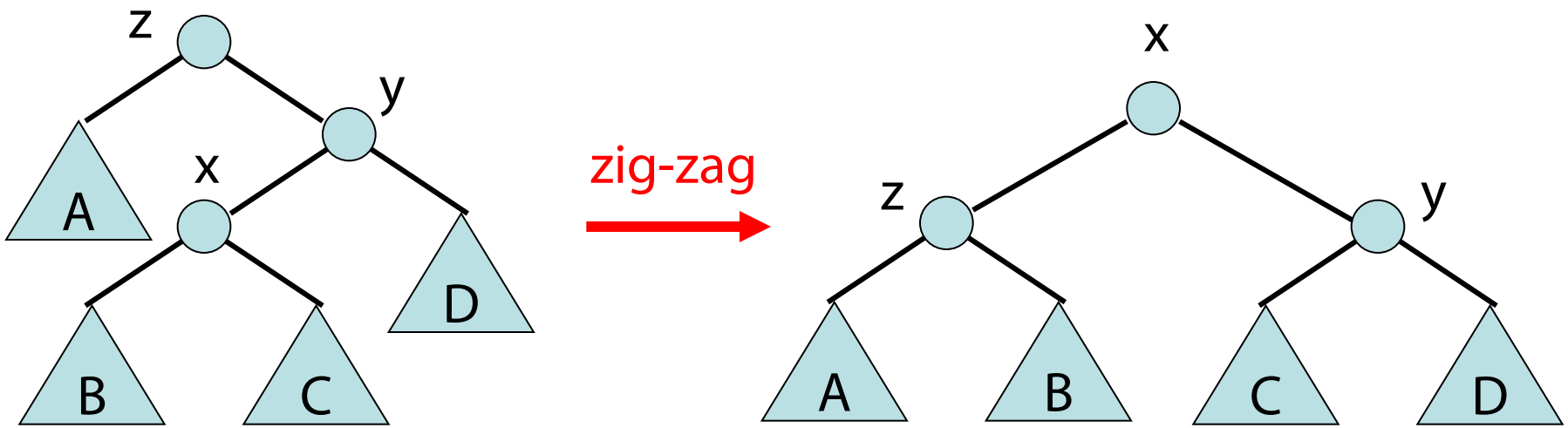
3b. x hat Vater rechts, Großvater links:



Splay-Operation

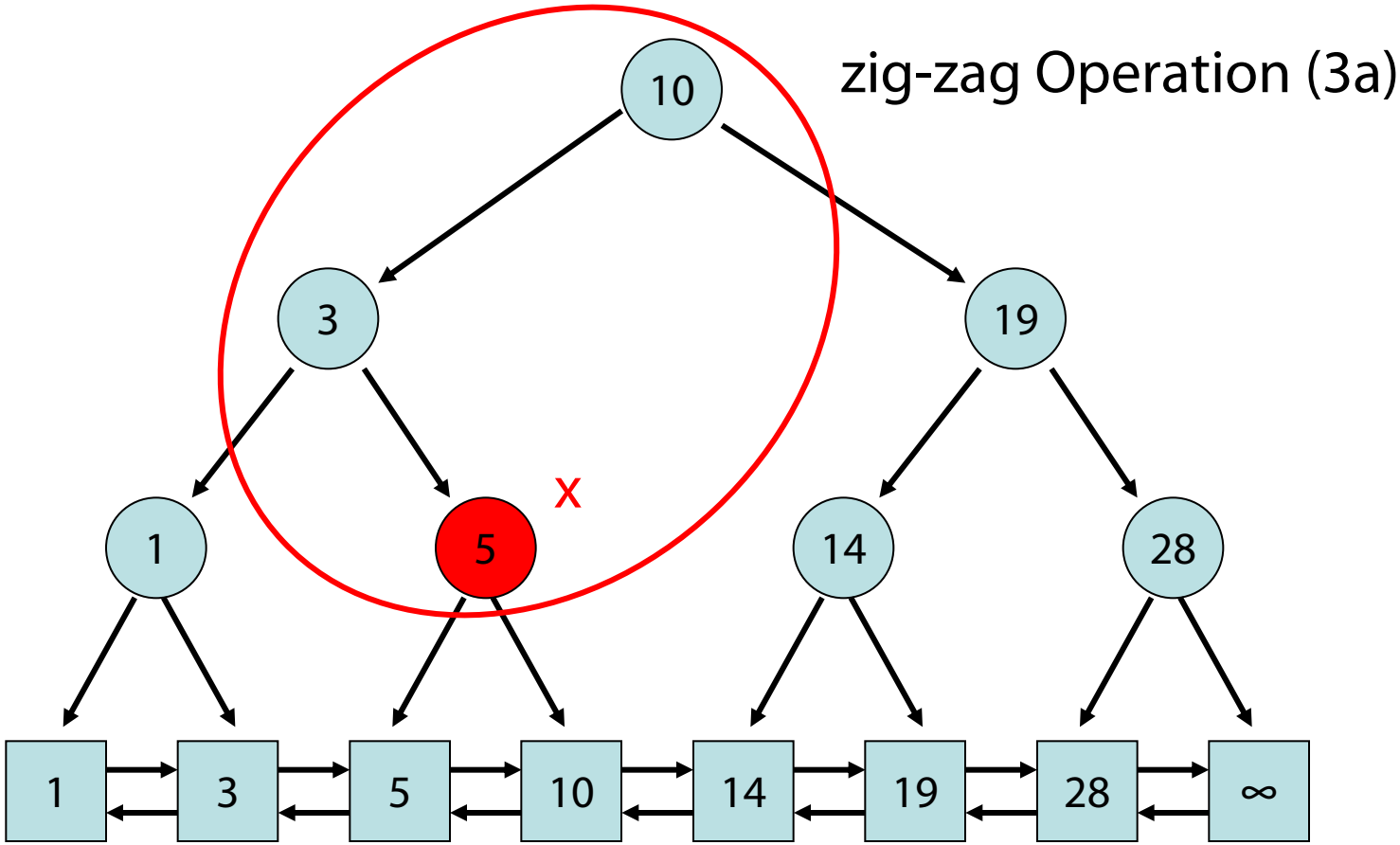
Wir unterscheiden zwischen 3 Fällen.

3b. x hat Vater rechts, Großvater links:

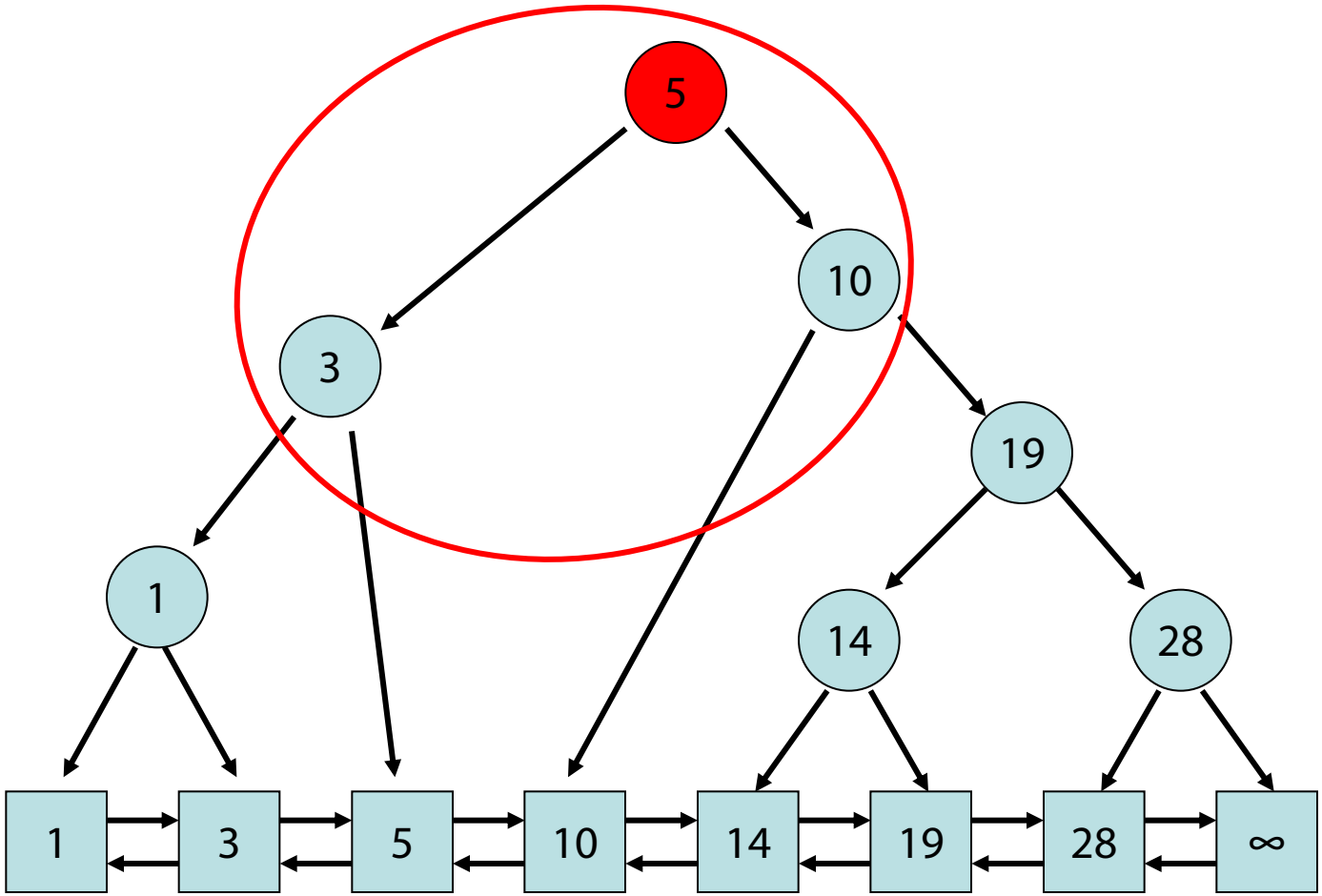


Splay-Operation

Beispiel:

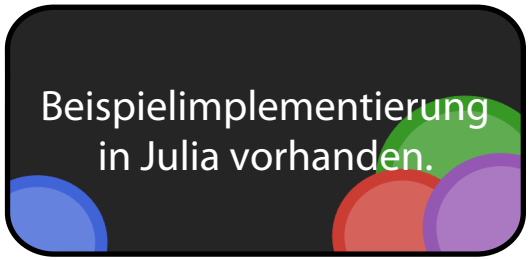


Splay-Operation



Splay

```
function splay(x, tree)
   while !isnothing(x.parent)
      # x wandert hoch
      Wende geeignete Splay-Operation an
   end
   tree.root = x
end
```



DataStructures.jl

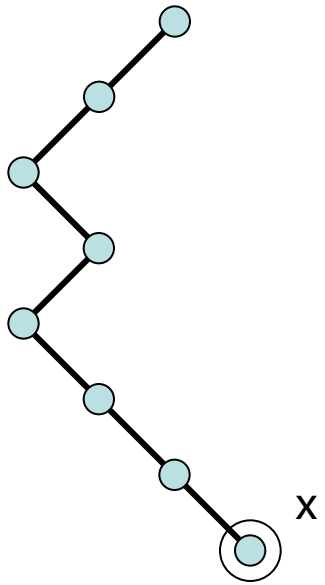
This package implements a variety of data structures, including

- Deque (implemented with an unrolled linked list)
- CircularBuffer
- CircularDeque (based on a circular buffer)
- Stack
- Queue
- Priority Queue
- Fenwick Tree
- Accumulators and Counters (i.e. Multisets / Bags)
- Disjoint Sets
- Binary Heap
- Mutable Binary Heap
- Ordered Dicts and Sets
- RobinDict and OrderedRobinDict (implemented with Robin Hood Hashing)
- SwissDict (inspired from SwissTables)
- Dictionaries with Defaults
- Trie
- Linked List and Mutable Linked List
- Sorted Dict, Sorted Multi-Dict and Sorted Set
- DataStructures.IntSet
- SparseIntSet
- DiBitVector
- Red Black Tree
- AVL Tree
- Splay Tree

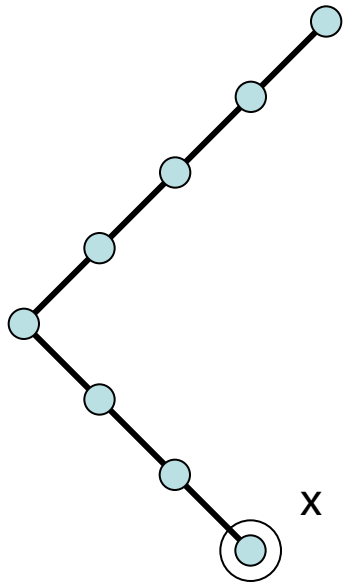
```
julia> using DataStructures
julia> pq = PriorityQueue();
```

Splay-Operation: Maximal ein zig

Beispiele:

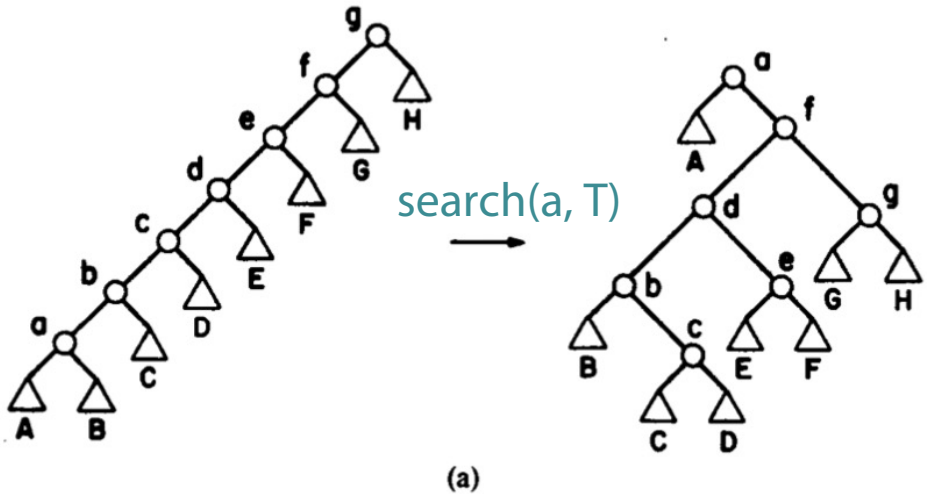


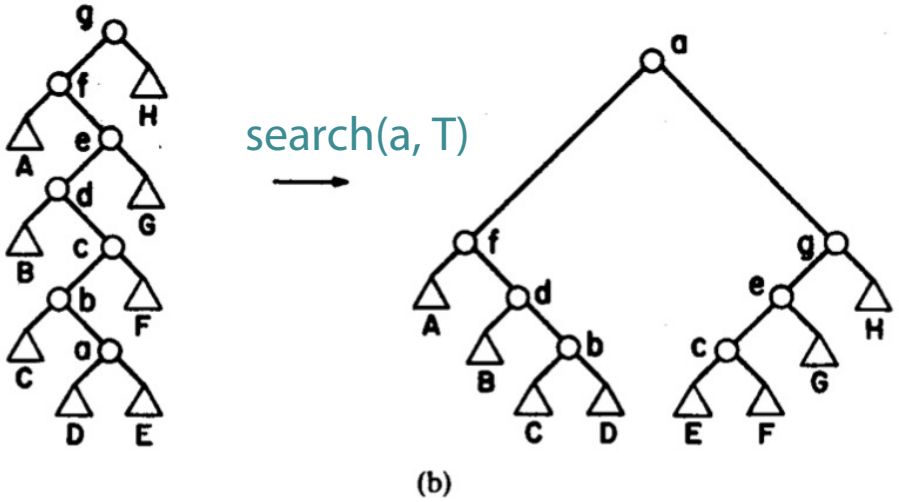
zig-zig, zig-zag, zig-zag, zig



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

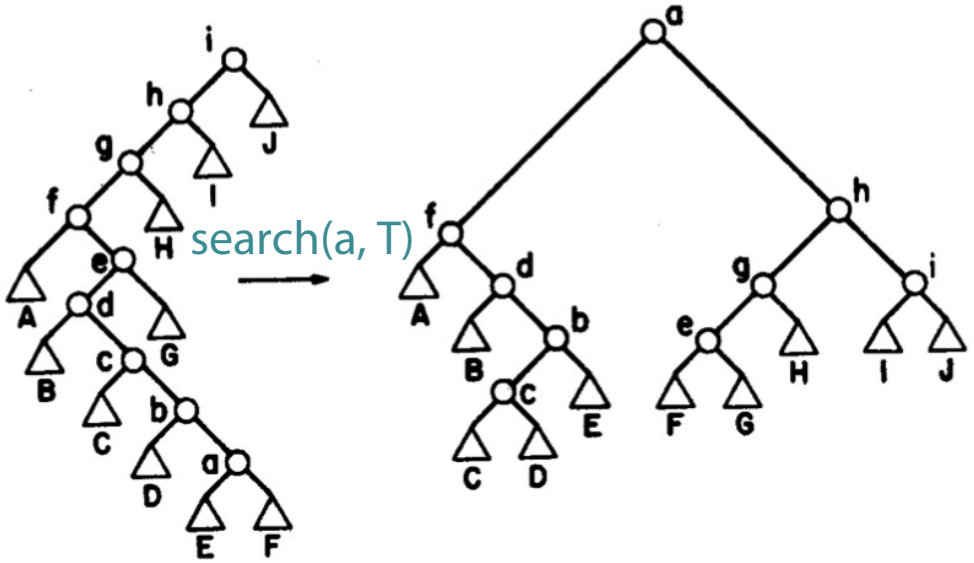
Wirkung der Splay-Operationen





Analyse von Splay-Bäumen

- Über eine Folge von Zugriffen mit search entsteht ein ausgeglichener Baum
- Argumentationstechnik:
  - $T_{search}(n)$ amortisiert in $O(\log n)$

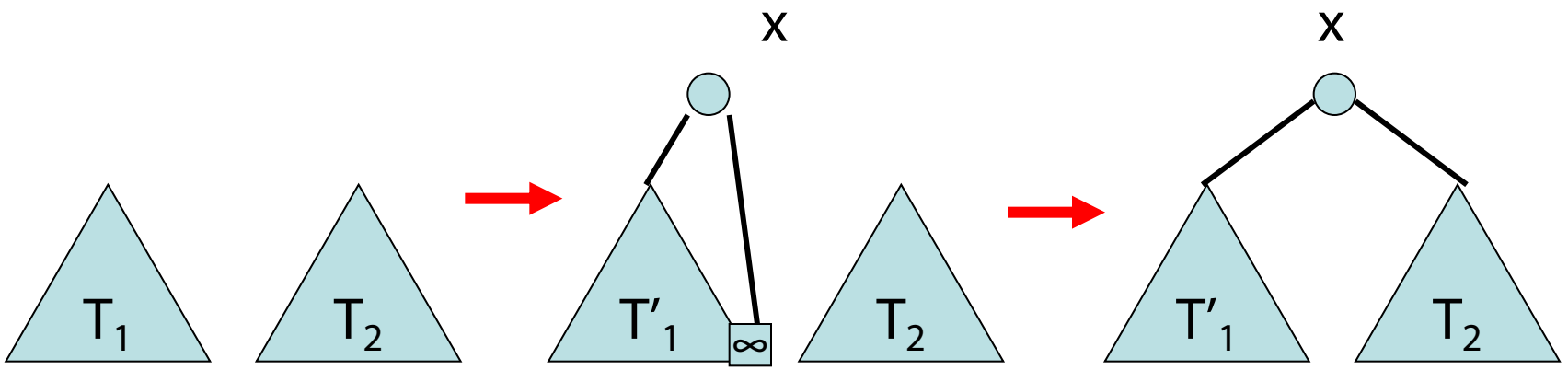


Daniel D. Sleator, Robert Tarjan: Self-Adjusting Binary Search Trees, In: Journal of the ACM (Association for Computing Machinery). 32, Nr. 3, S. 652–686, **1985**

Splay-Baum Operationen

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

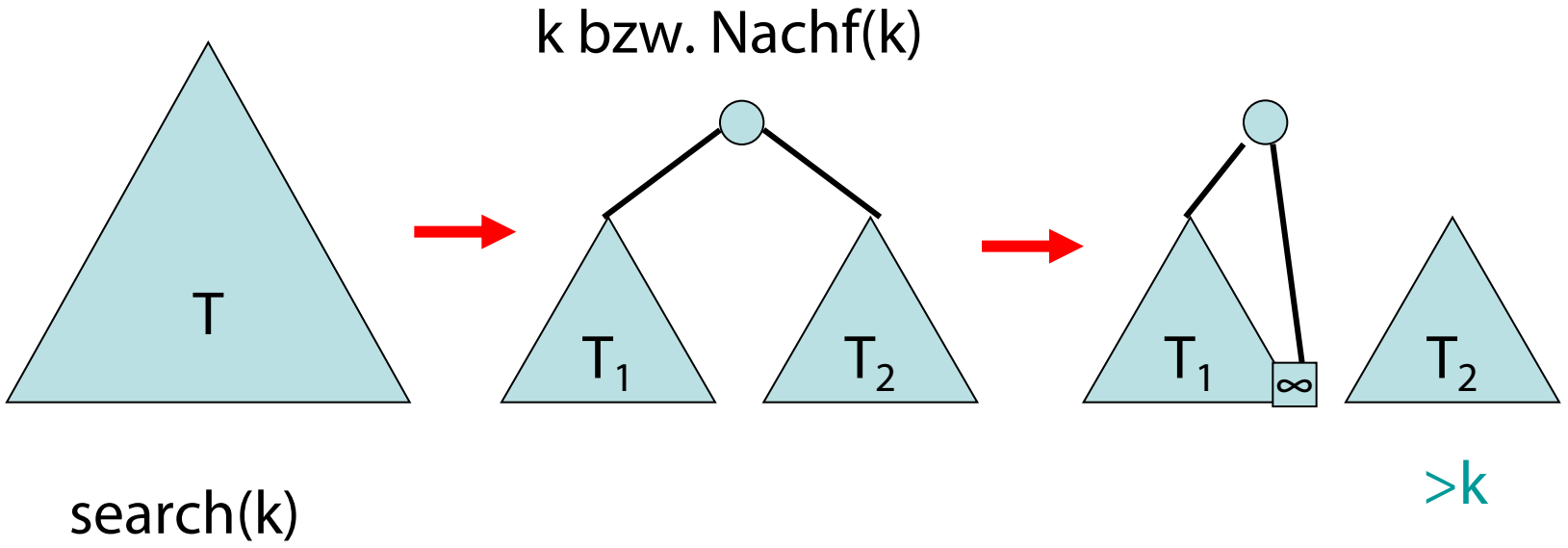
merge(T_1,T_2):



search(x, T_1), x<∞ max. in T_1

Splay-Baum Operationen

split(k,T):



Splay-Baum Operationen

insert(e, T):

- Führe split(key(e), T) aus
  (key(e) in Wurzel des linken Baums)

delete(k, T):

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

Zusammenfassung Splay-Baum

- Umstrukturierung bei der Suche
  - Search, insert, delete sind effizient (O(log n))
- Statische Optimalität
  - Für geg. Anfragesequenz ist Splay-Baum-Zugriffszeit asymptotisch gleich zu der Zugriffszeit mit optimalem Baum bzgl. a priori gegebener Gewichte der Knoten
- Dynamische Optimalität?
  - Gilt das auch bzgl. der besten dynamischen Baumstruktur?
  - Offenes Problem

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

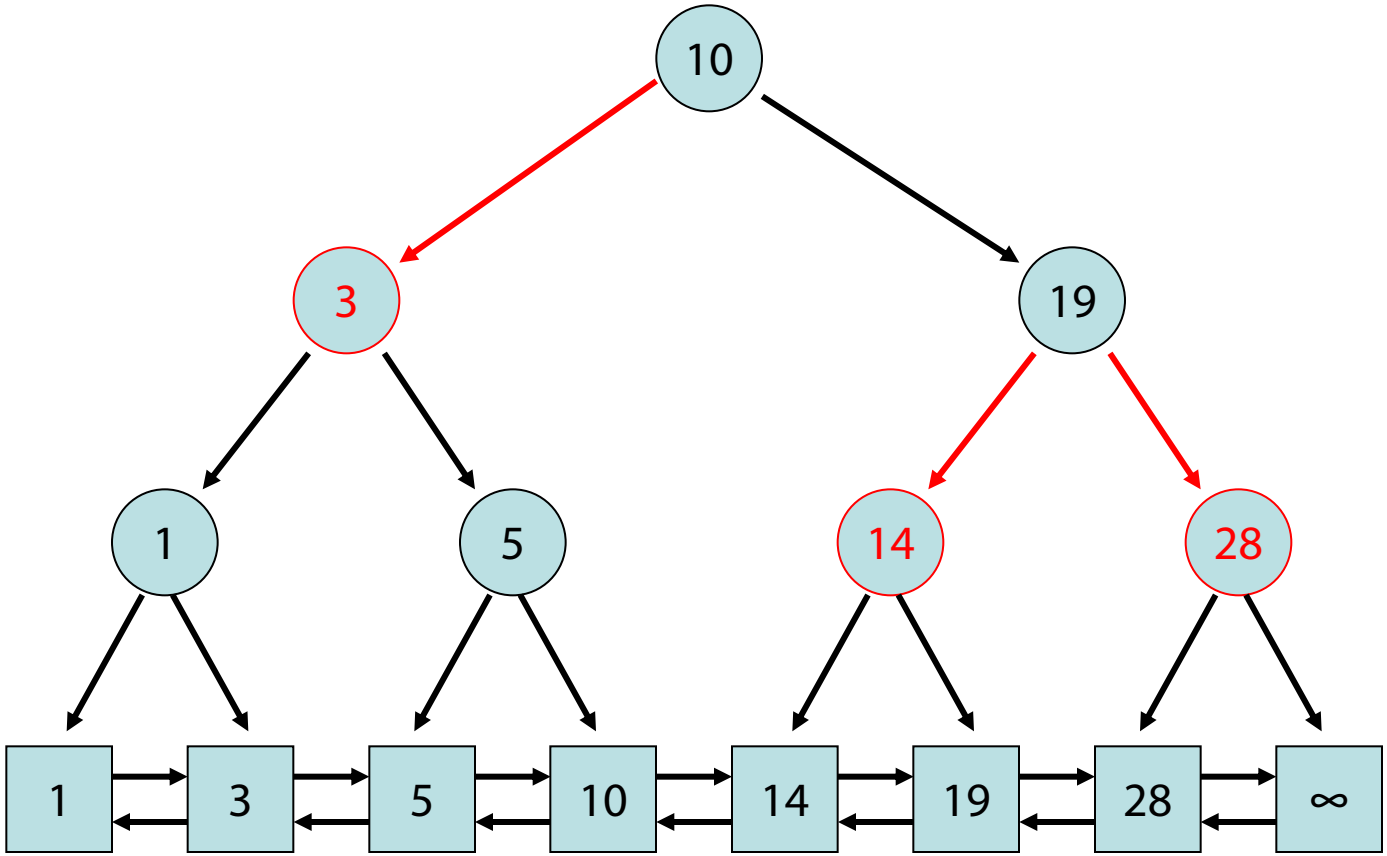
Es gibt keine direkten Kanten zu Listenelementen

Beispielimplementierung in Julia vorhanden.

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

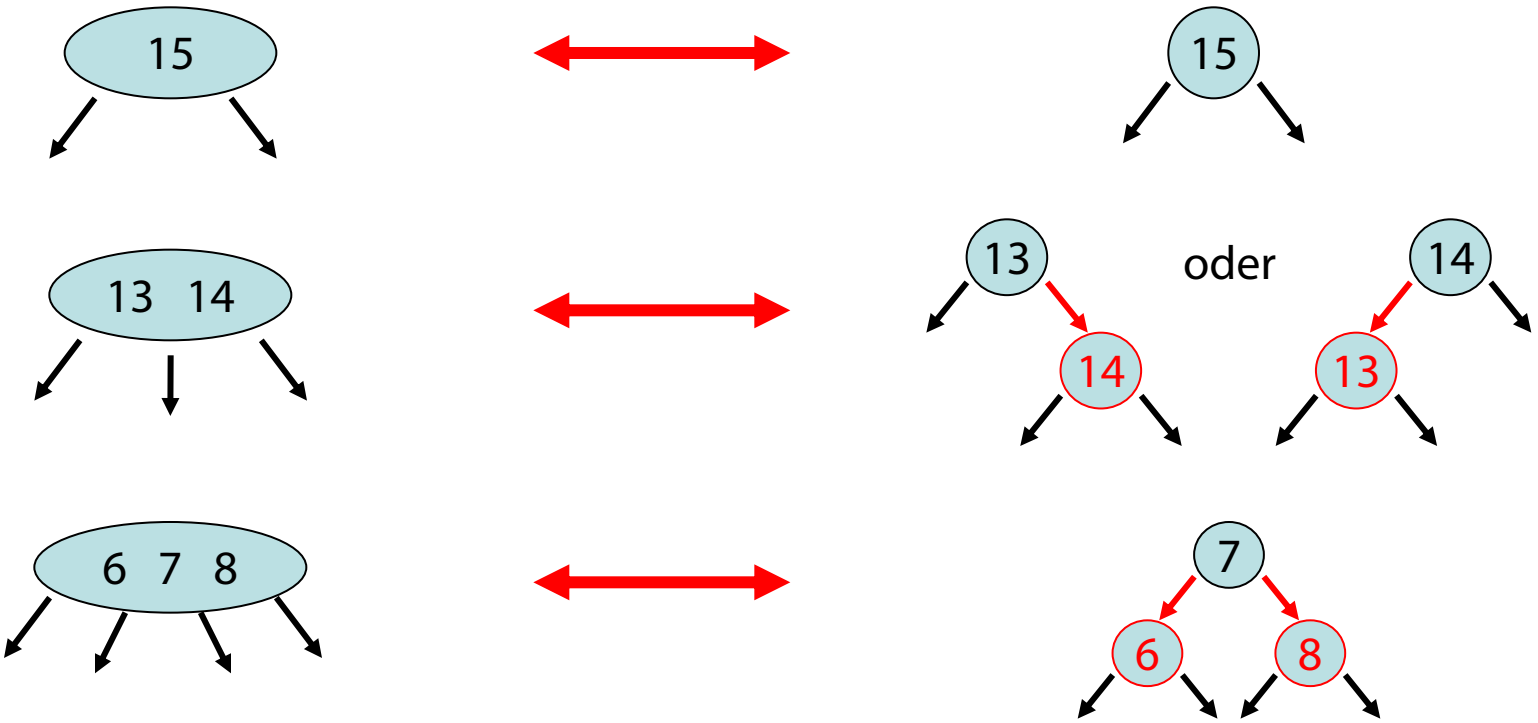
Beispiel:



Rot-Schwarz-Baum

Andere Deutung von Rot-schwarz-Mustern:

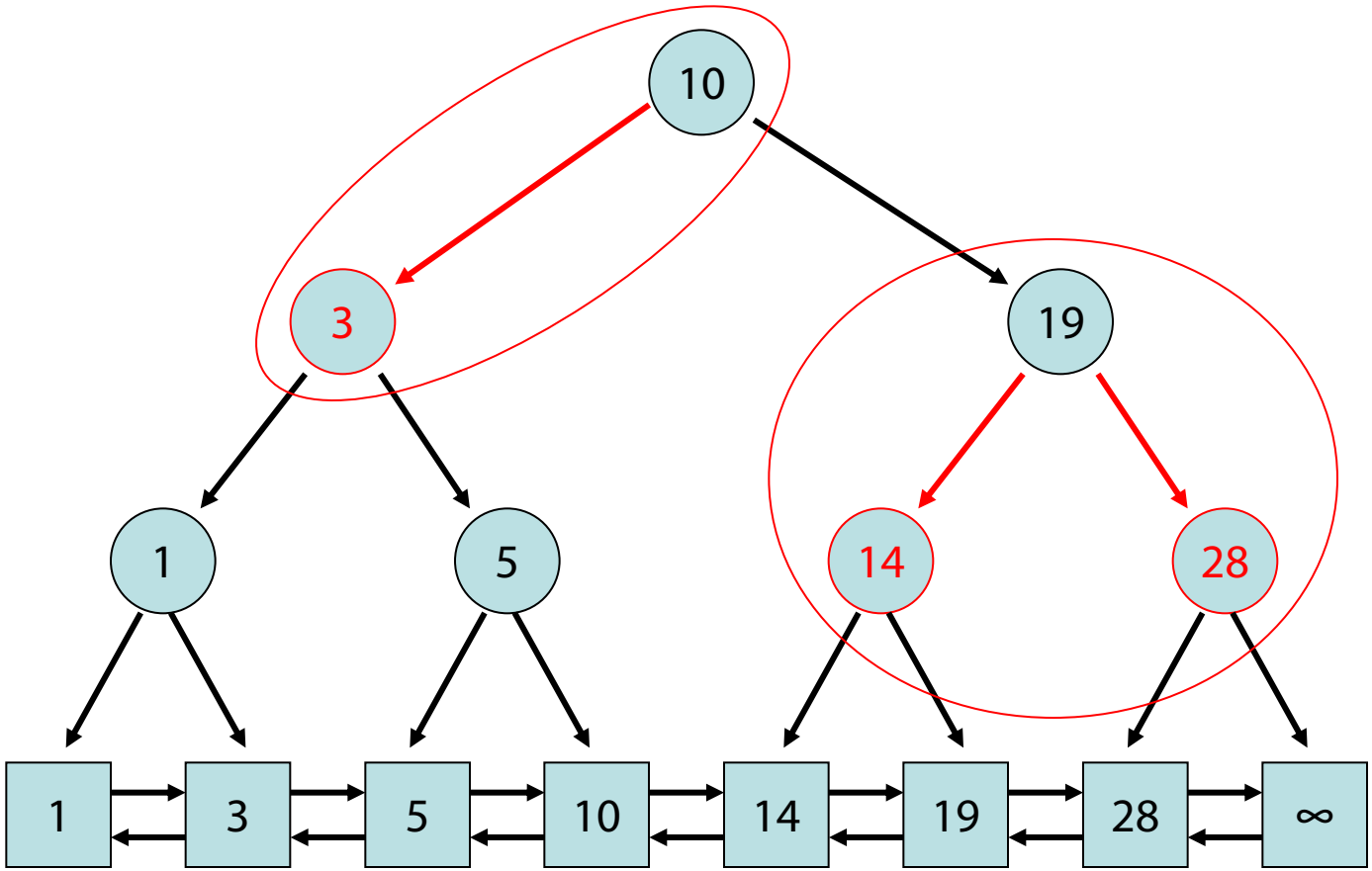
B-Baum (Baum mit “Separatoren” und min 2 max 4 Nachfolger)



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

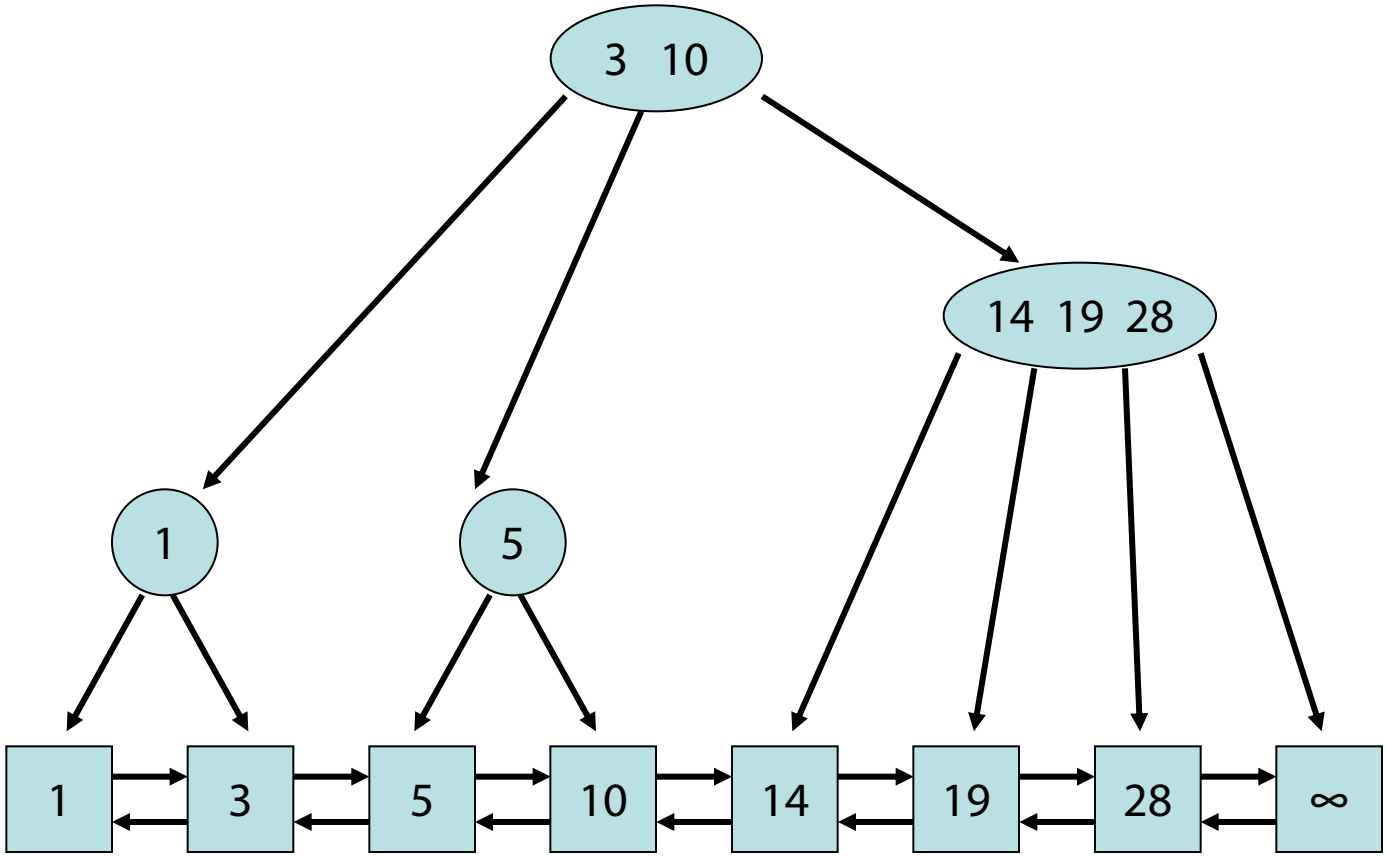
Rot-Schwarz-Baum

R-S-Baum:



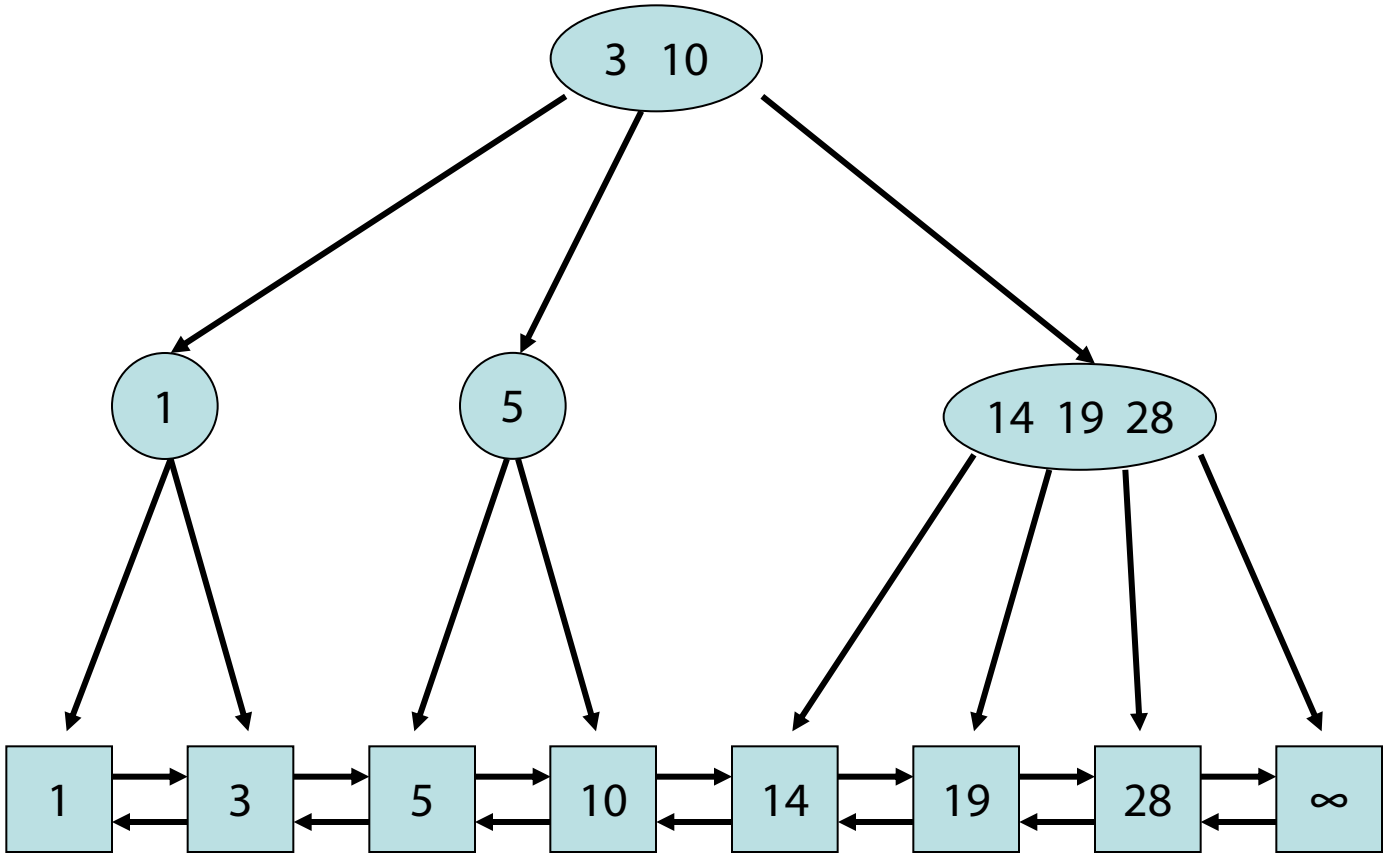
Rot-Schwarz-Baum

B-Baum:



Rot-Schwarz-Baum

B-Baum:



Rot-Schwarz-Baum

Behauptung: Die Tiefe t eines Rot-Schwarz-Baums T mit n Elementen ist in $\Theta(\log n)$. Also: Der Rot-Schwarz-Baum T ist ausgeglichen.

Beweis:

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

- 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 ein Rot-Schwarz-Baum ein Binärbaum ist und rote "dazwischen" sind.

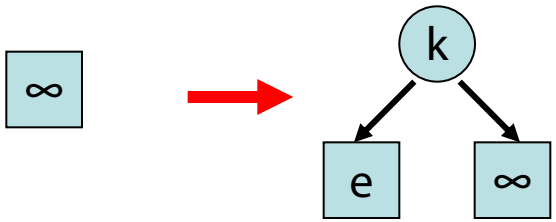
Rot-Schwarz-Baum

search(k, T): wie im binären Suchbaum

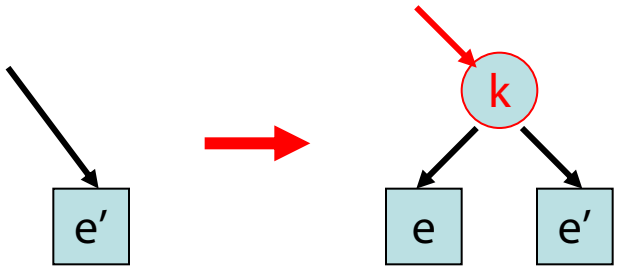
insert(e, T):

- Führe search(k, T) mit k=key(e) aus
- Füge e vor Nachfolger e' in Liste ein

Fall 1: Baum leer
(nur Markerelement ∞ enthalten)

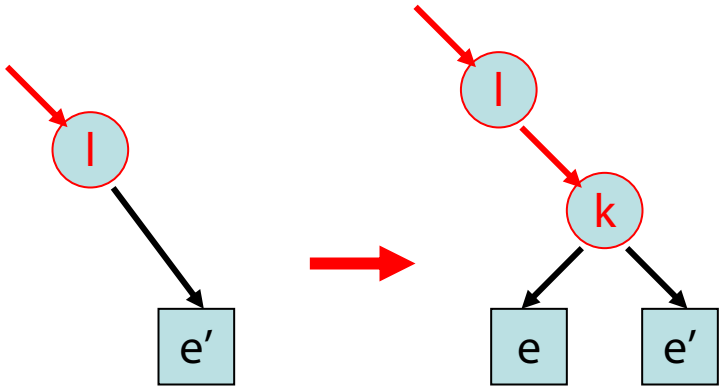


Fall 2: Baum nicht leer



Problem

- Knoten e' kann schon roten Vorgänger haben



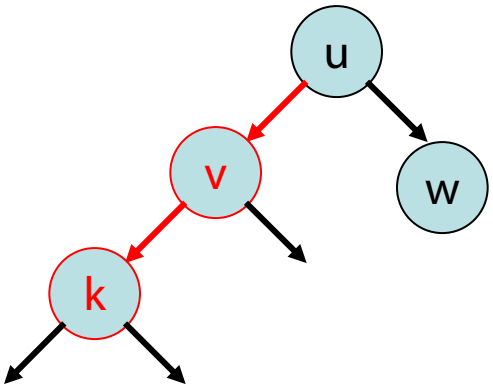
Rot-Schwarz-Baum

insert(e, T):

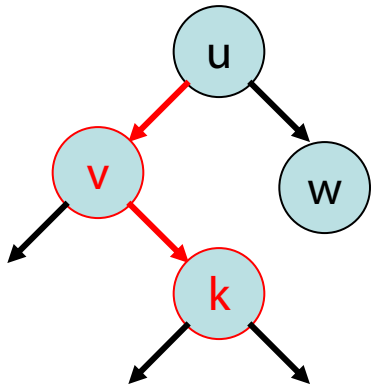
- Führe search(k, T) mit k=key(e) aus
- Füge e vor Nachfolger e' in Liste ein
  (bewahrt alles bis auf evtl. interne Eigenschaft)
- Interne Eigenschaft verletzt (Fall 2 vorher): 2 Fälle
  - Fall 1: Vater von k in T hat **schwarzen Bruder**
    (Restrukturierung, aber beendet Reparatur)
  - Fall 2: Vater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

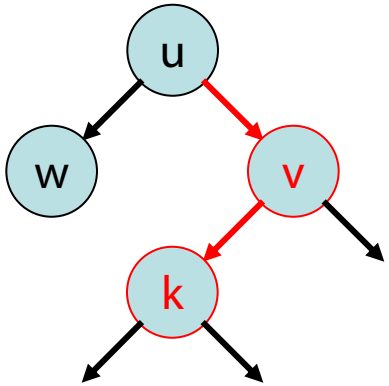
Rot-Schwarz-Baum

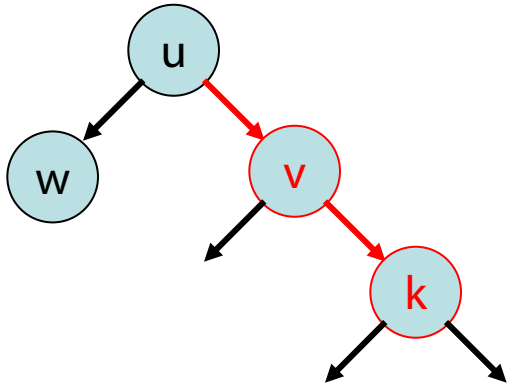
Fall 1: Vater v von k in T hat schwarzen Bruder w



Alternativen für Fall 1



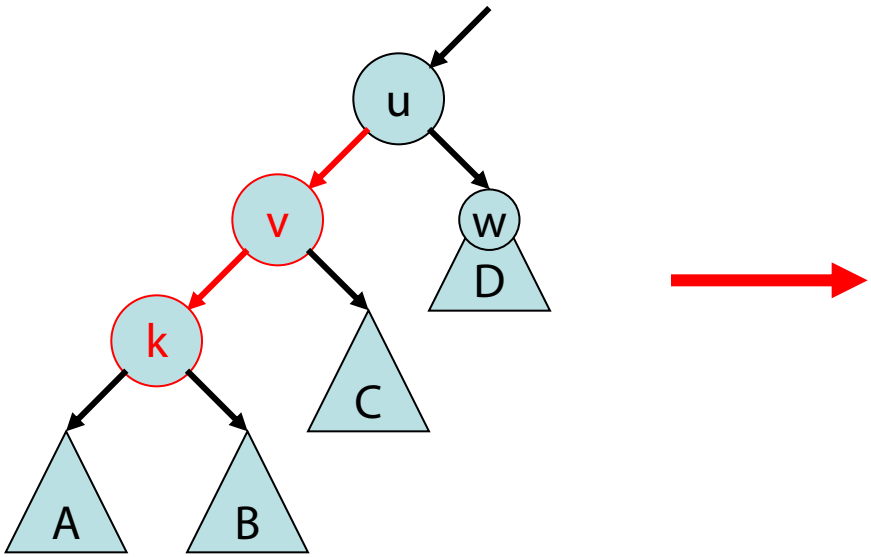




Rot-Schwarz-Baum

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

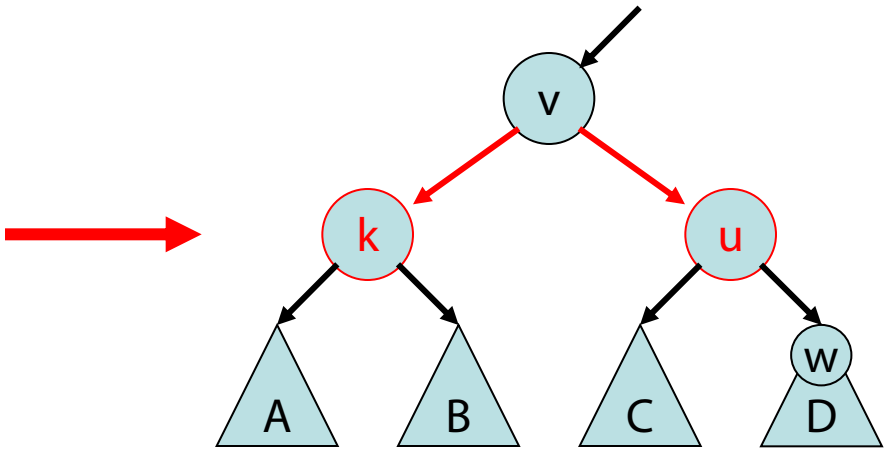
Lösung:



Rot-Schwarz-Baum

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

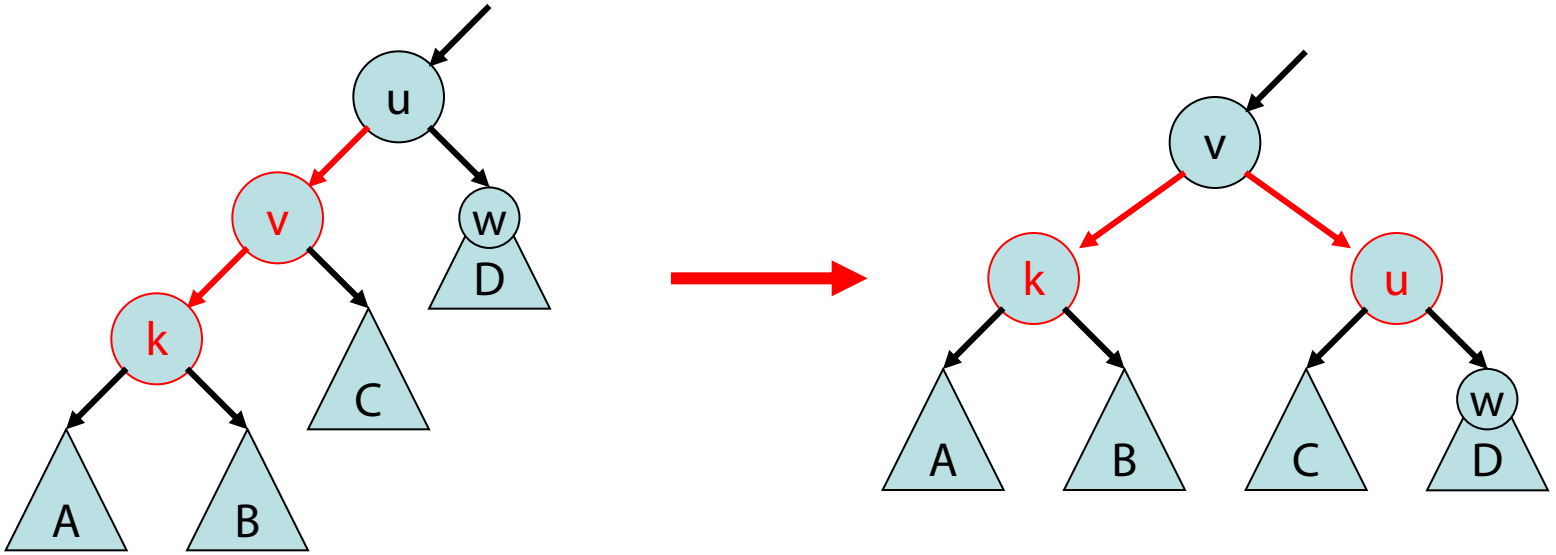
Lösung:



Rot-Schwarz-Baum

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

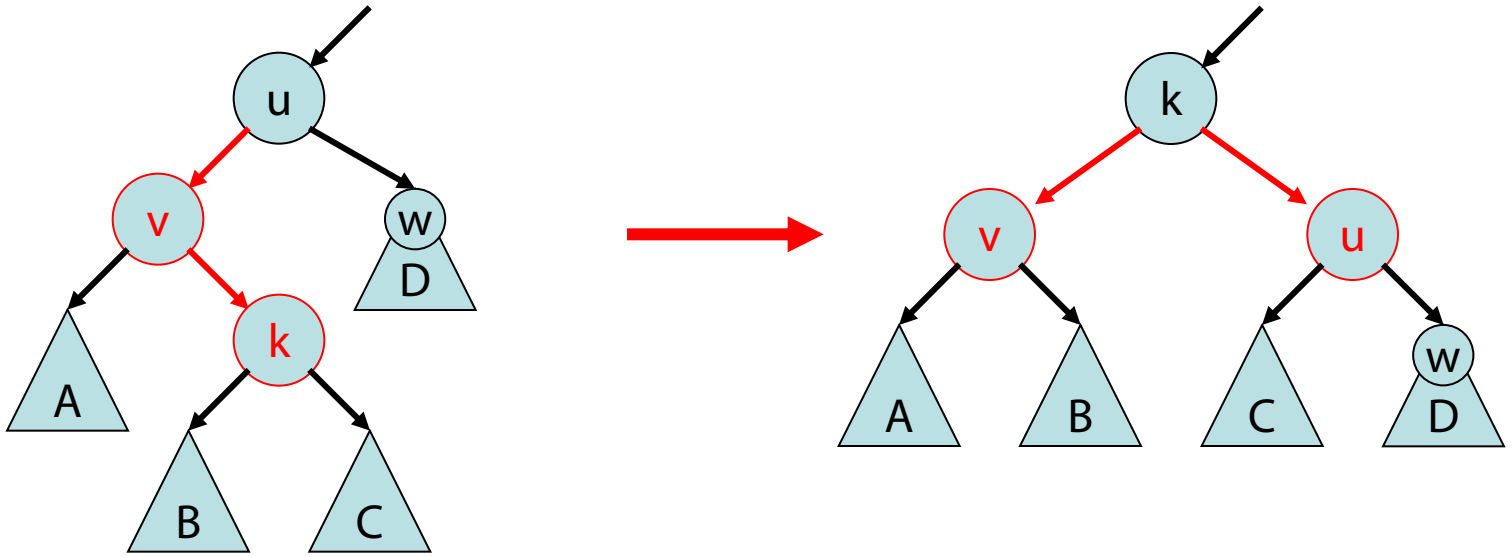
Lösung:



Rot-Schwarz-Baum

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

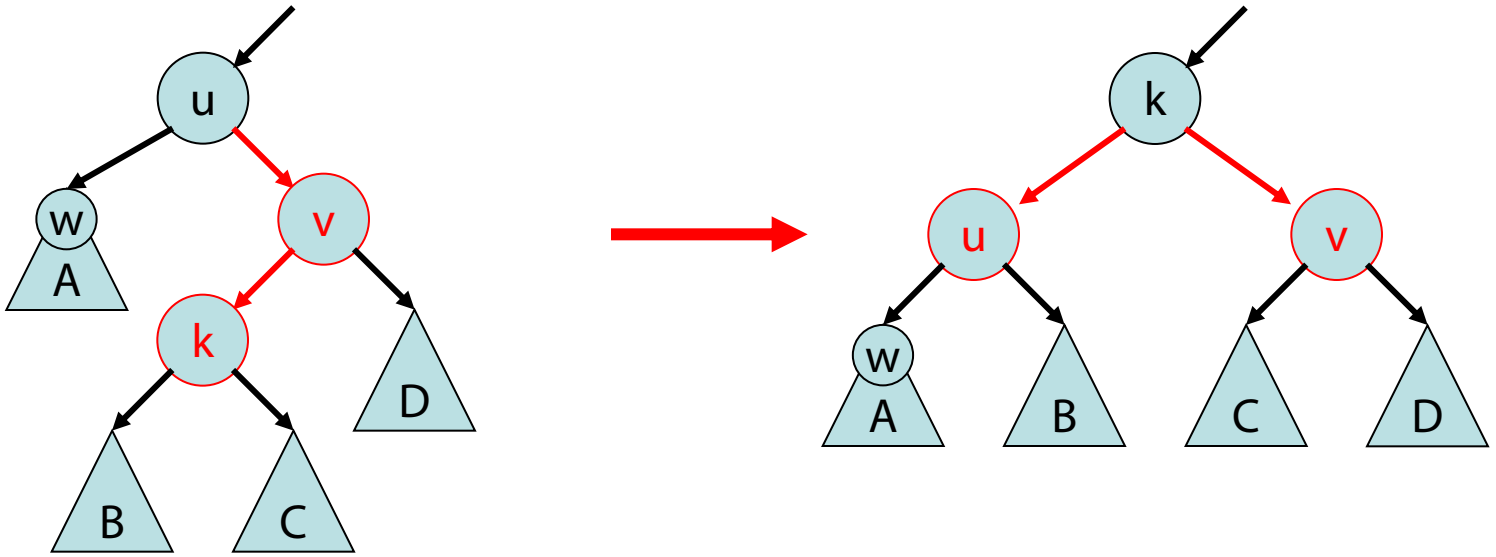
Lösung:



Rot-Schwarz-Baum

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

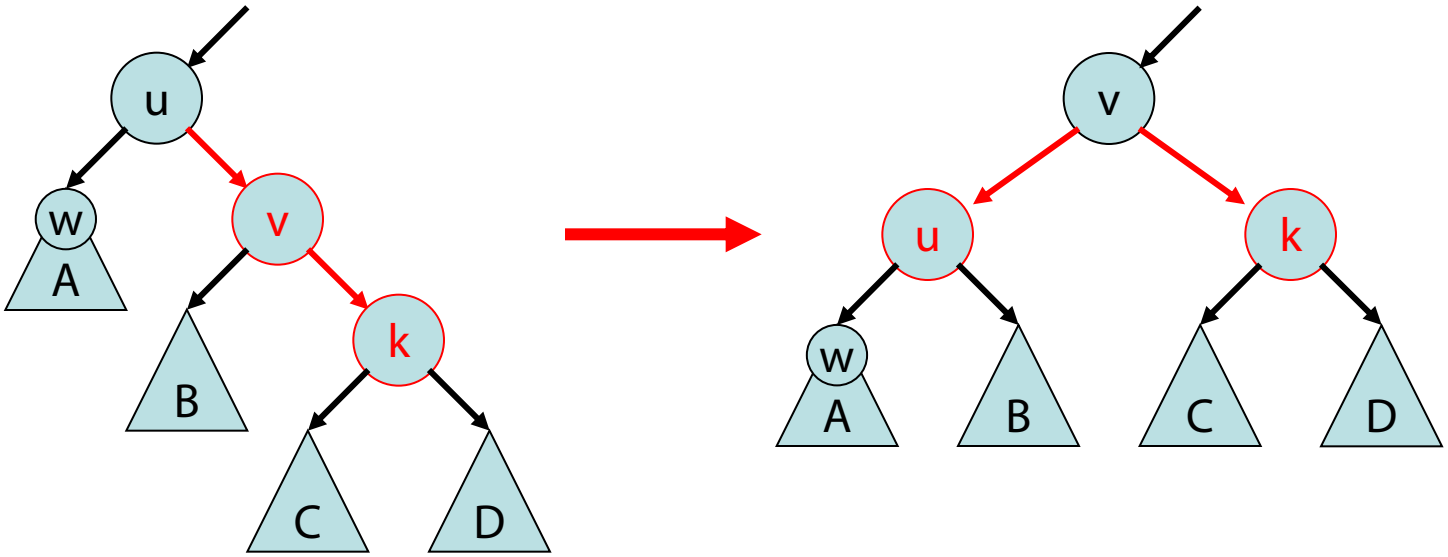
Lösung:



Rot-Schwarz-Baum

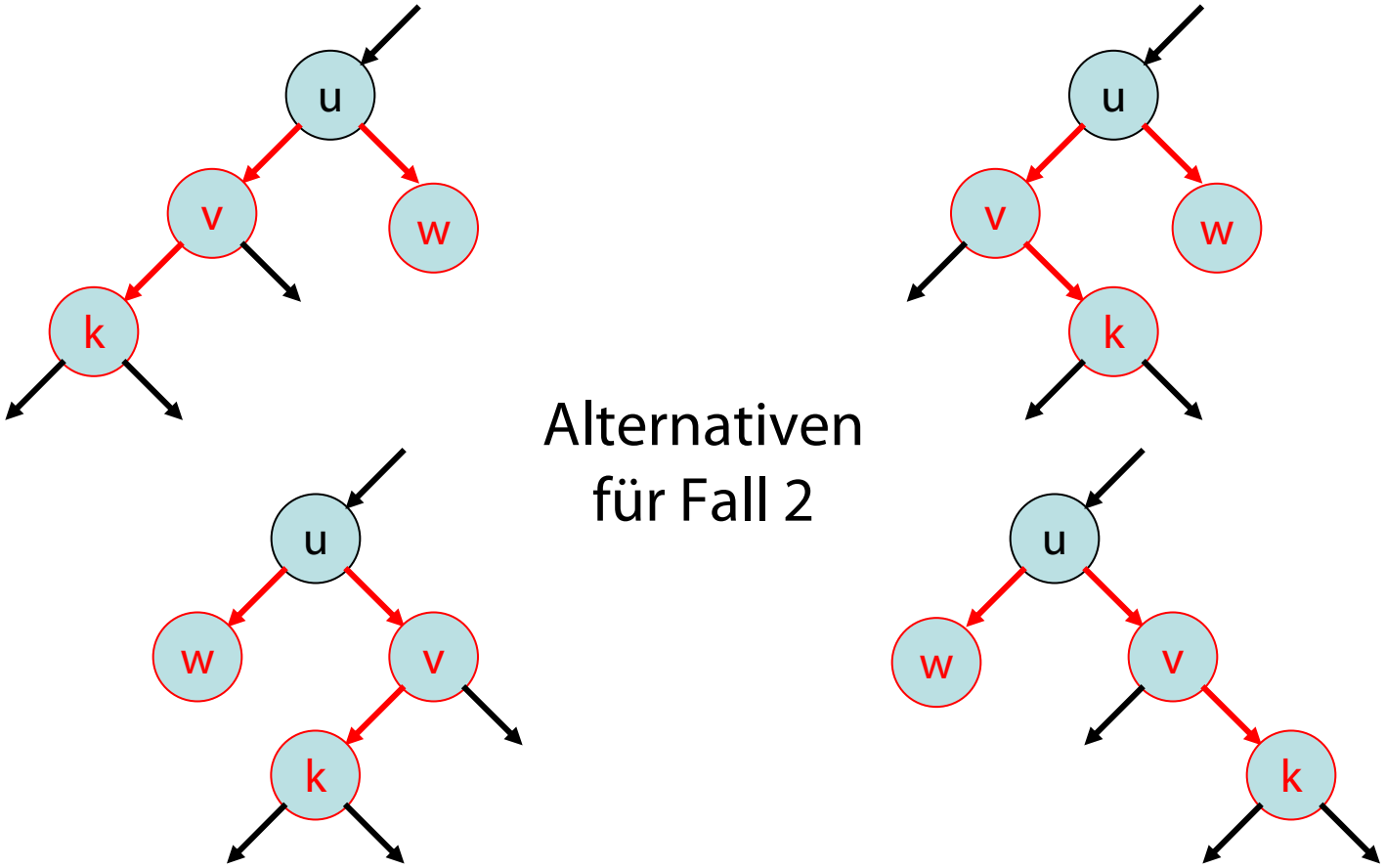
Fall 1: Vater v von k in T hat schwarzen Bruder w

Lösung:



Rot-Schwarz-Baum

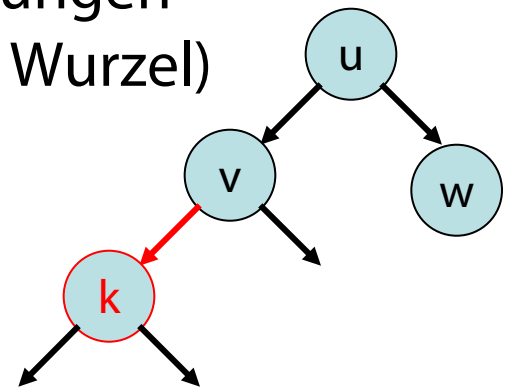
Fall 2: Vater v von k in T hat roten Bruder w



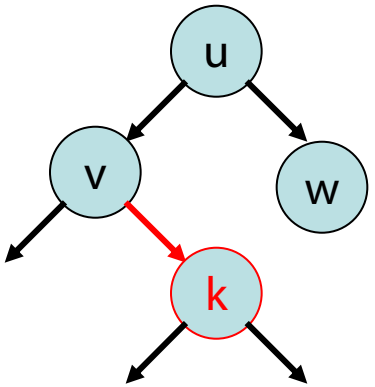
Rot-Schwarz-Baum

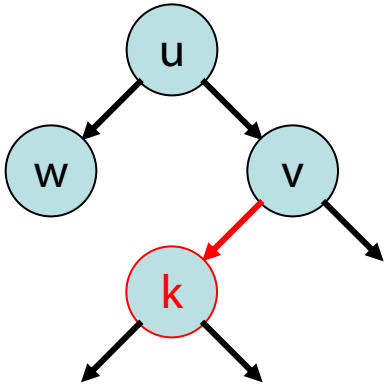
Fall 2: Vater v von k in T hat roten Bruder w

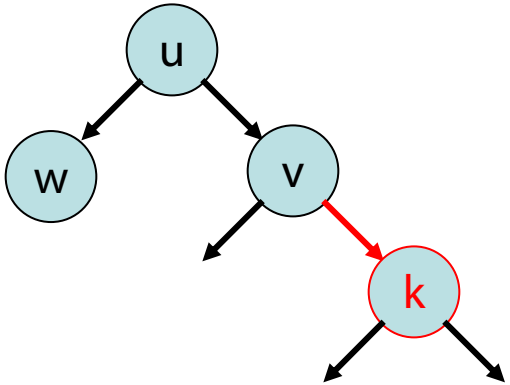
Lösungen
(u ist Wurzel)



Schwarztiefe+1



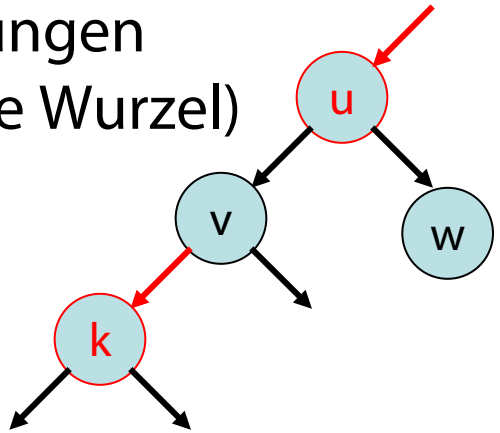




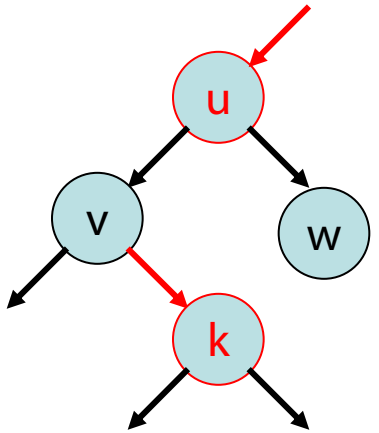
Rot-Schwarz-Baum

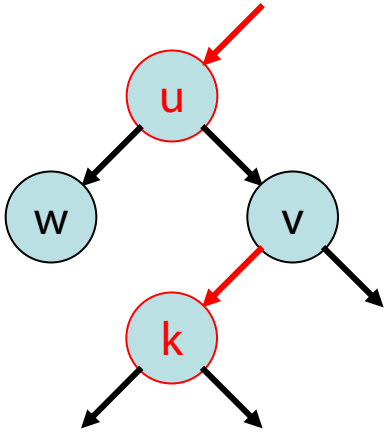
Fall 2: Vater v von k in T hat roten Bruder w

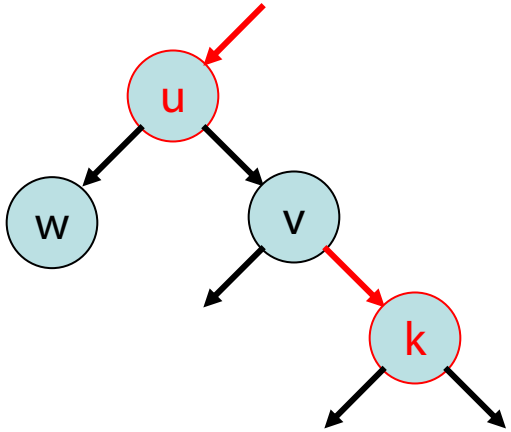
Lösungen
(u keine Wurzel)



bewahrt
Schwarztiefe!

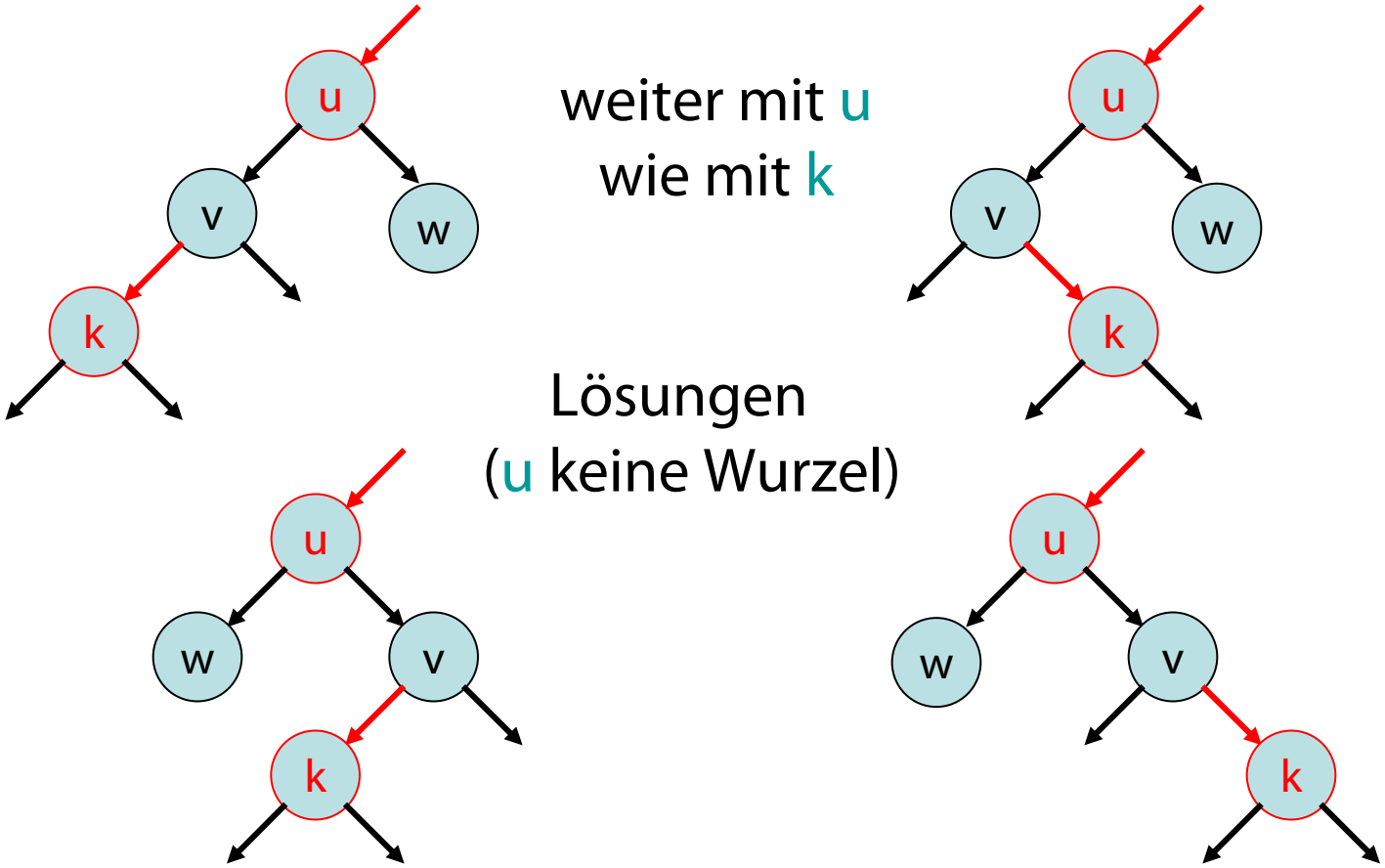






Rot-Schwarz-Baum

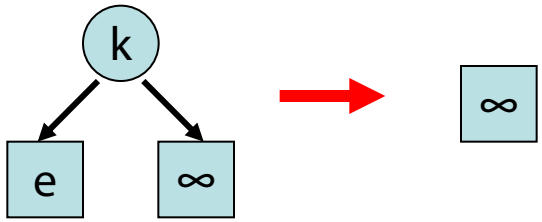
Fall 2: Vater v von k in T hat roten Bruder w



Rot-Schwarz-Baum

delete(k, T):

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



Rot-Schwarz-Baum

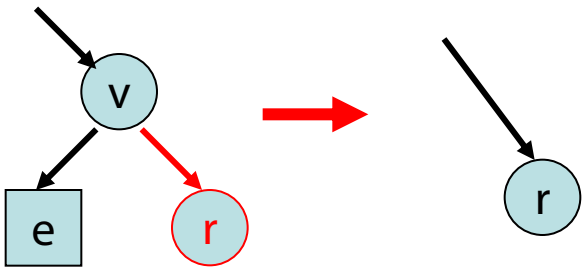
delete(k, T):

- Führe search(k, T) 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, T):

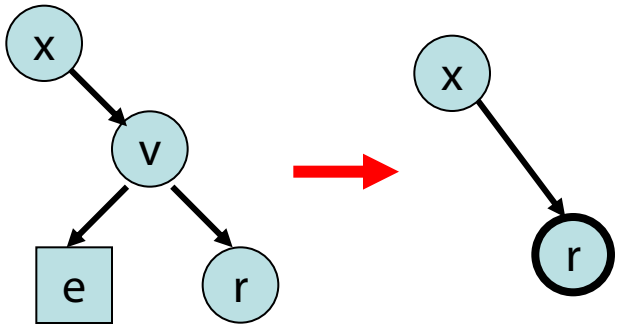
- Führe search(k, T) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 3: Vater v von e ist schwarz und Bruder rot



Rot-Schwarz-Baum

delete(k, T):

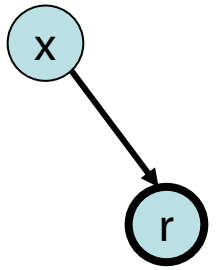
- Führe search(k, T) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Fall 4: Vater v von e und Bruder r sind schwarz



Tiefenregel verletzt!
r heißt dann doppelt schwarz

Rot-Schwarz-Baum

delete(k, T):



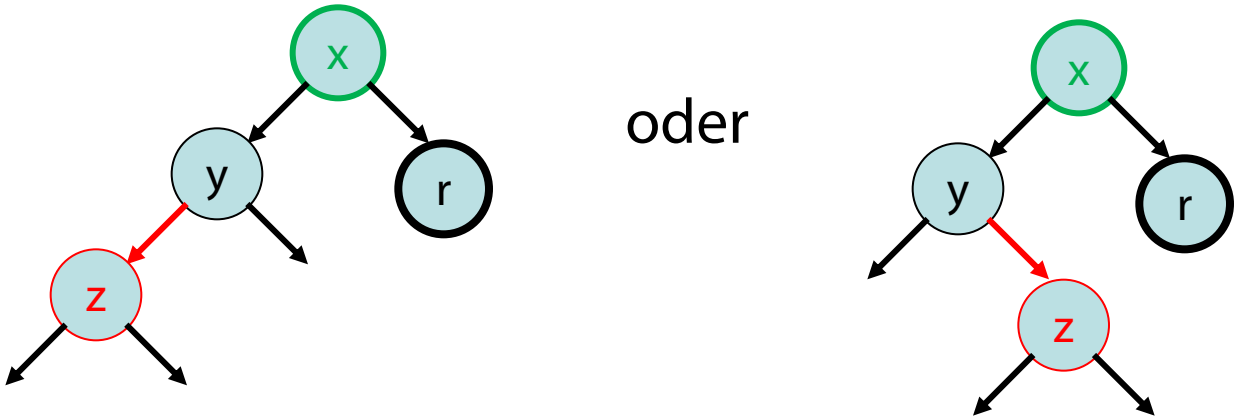
- Führe search(k, T) auf Baum aus
- Lösche Element e mit key(e)=k wie im binären Suchbaum
- Falls Vater v von e und Bruder r schwarz (s.o.), dann 3 weitere Fälle:
  - Fall 1: Bruder y von r ist schwarz und hat rotes Kind
  - Fall 2: Bruder y von r ist schwarz und beide Kinder von y sind schwarz (evtl.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

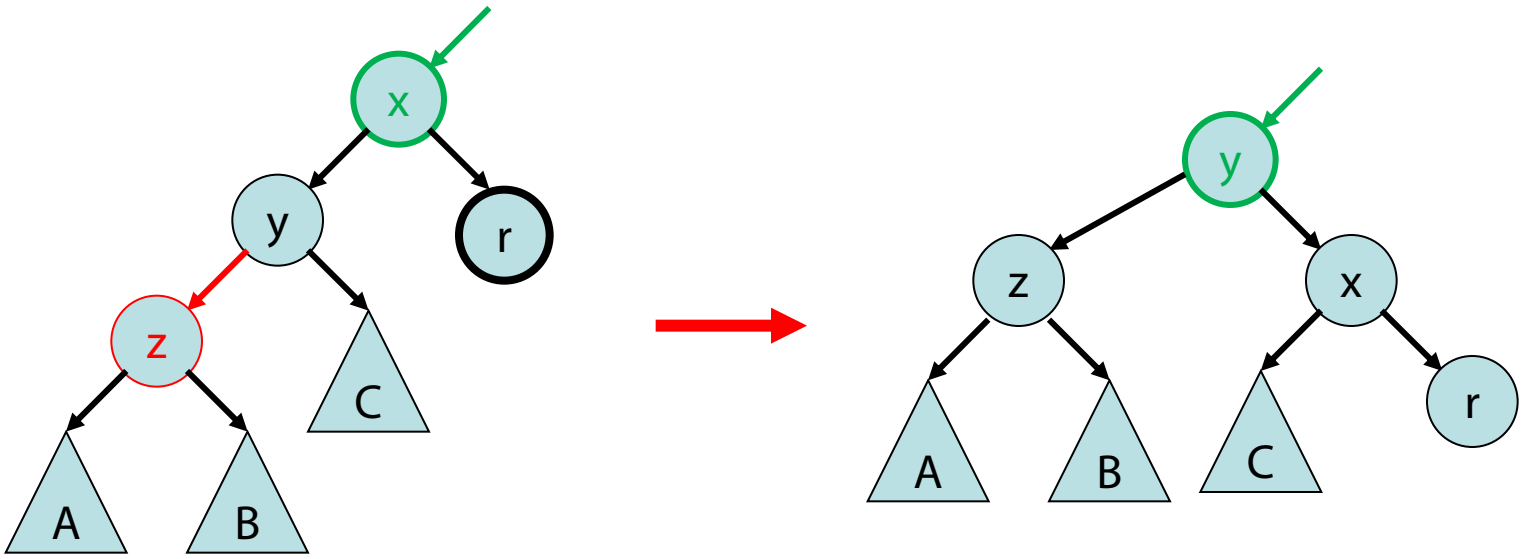
Alternativen für Fall 1: (x: beliebig gefärbt)



Rot-Schwarz-Baum

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

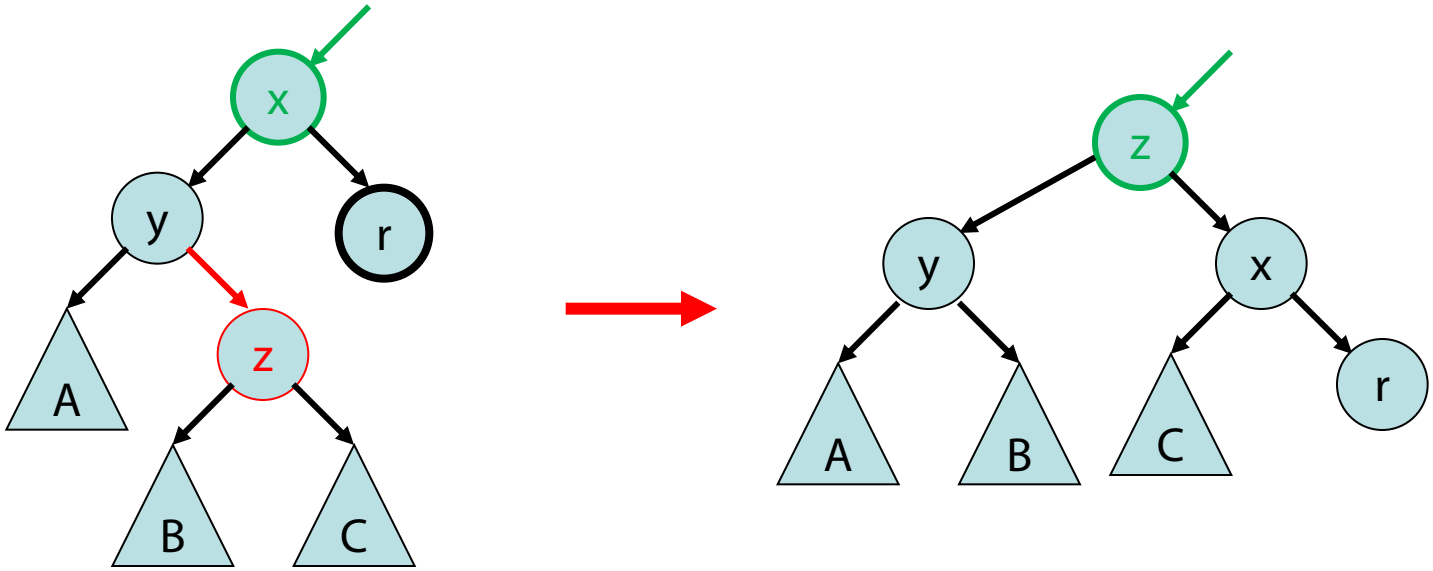
O.B.d.A. sei r rechtes Kind von x (links: analog)



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)

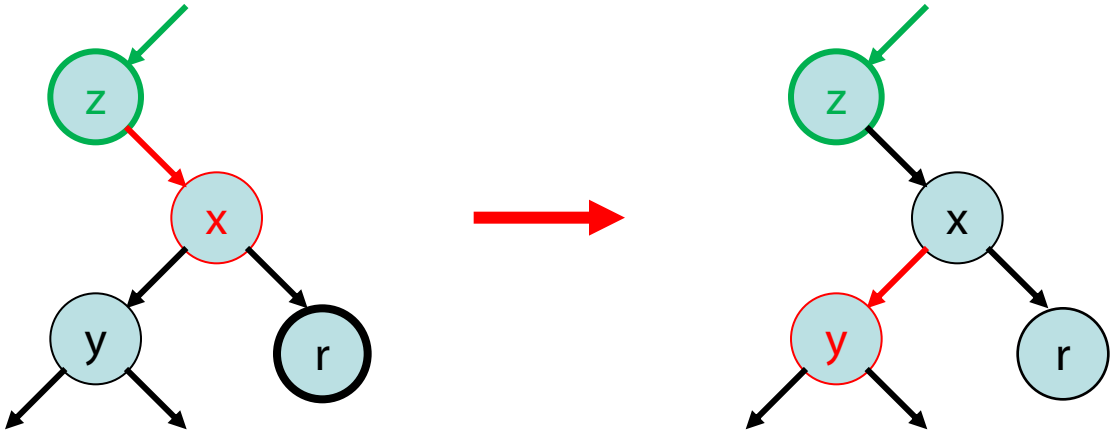
Alternativen für Fall 2: (z beliebig gefärbt)

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)

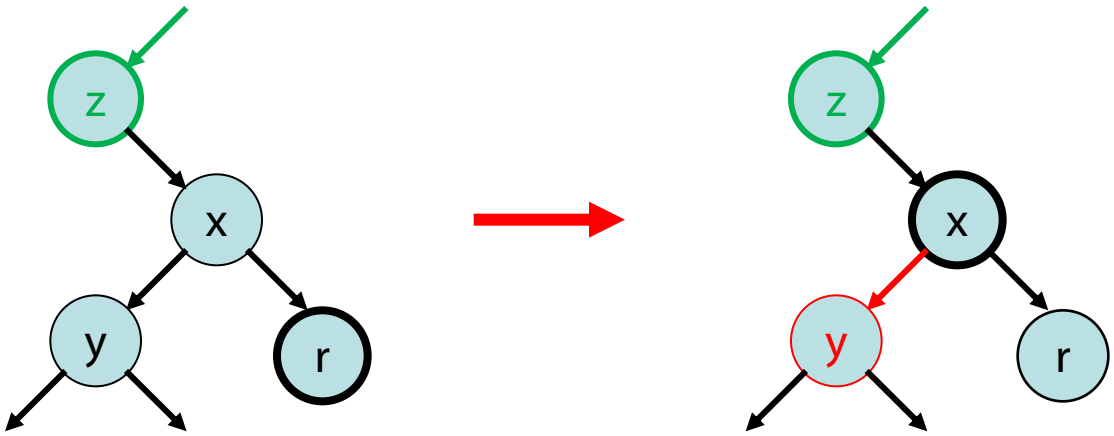


Rot-Schwarz-Baum

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)



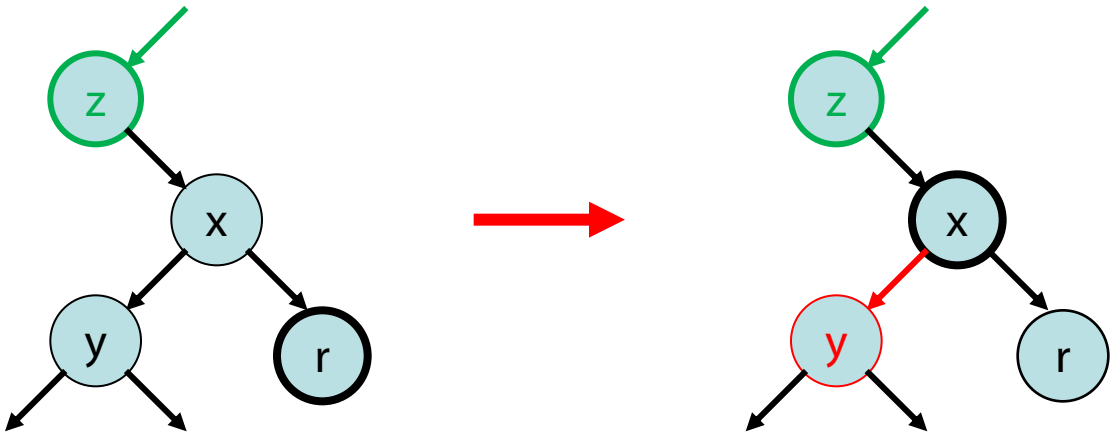
x ist Wurzel: fertig (Schwarztiefe-1)

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)

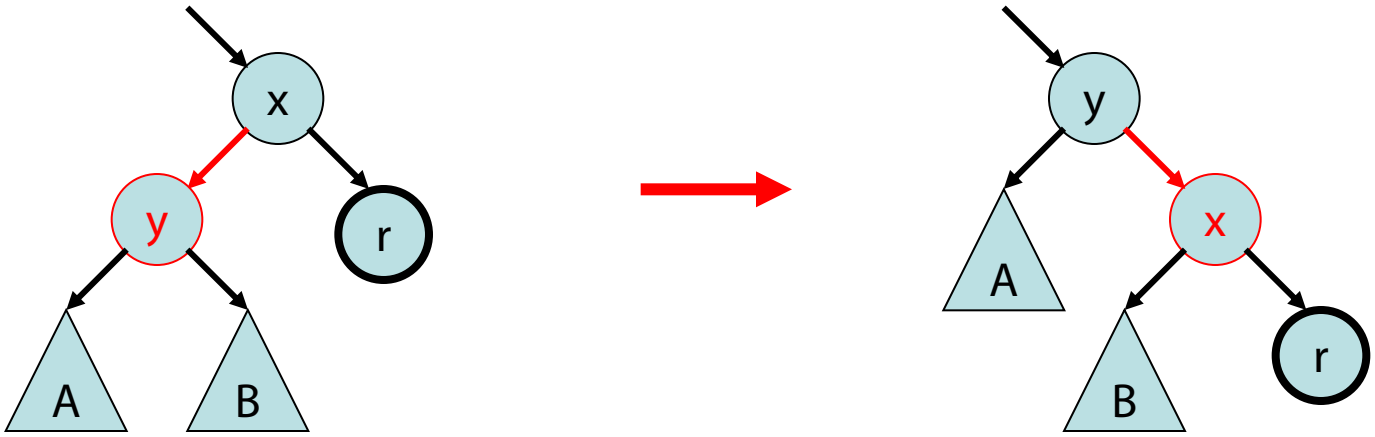


x keine Wurzel: weiter wie mit r

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)

Zum Vergleich:

Splay-Bäume

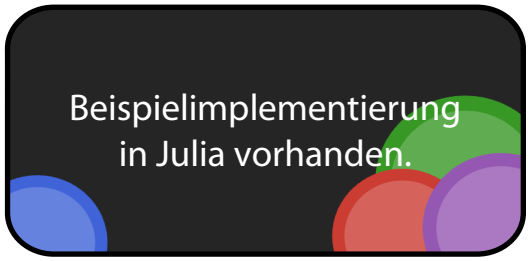
- search: O(log n) amort.
- insert: O(log n)
- delete: O(log n)

Restrukturierungen (Drehoperationen)

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

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

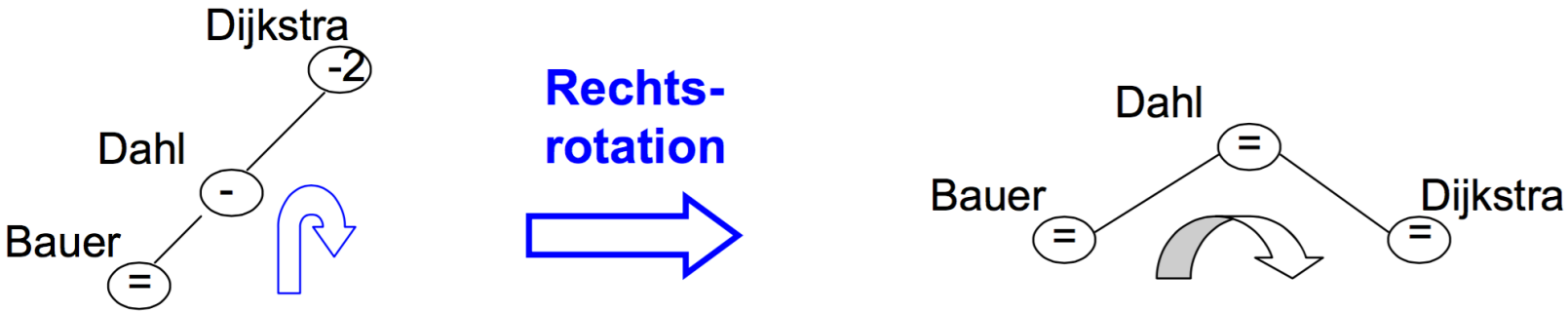
Die AVL-Präsentationen wurden übernommen aus:

- „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

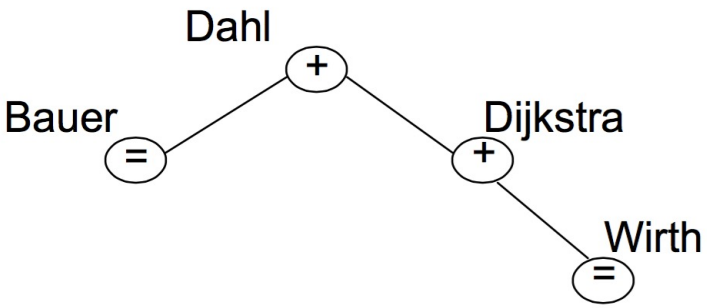
Einfügen von Dahl: Neuberechnung des Balancierungsfaktors, AVL Kriterium erfüllt

Einfügen von Bauer: Verletzung des AVL Kriteriums



Nach Rechtsrotation: AVL Kriterium wieder erfüllt

Einfügen in AVL-Baum

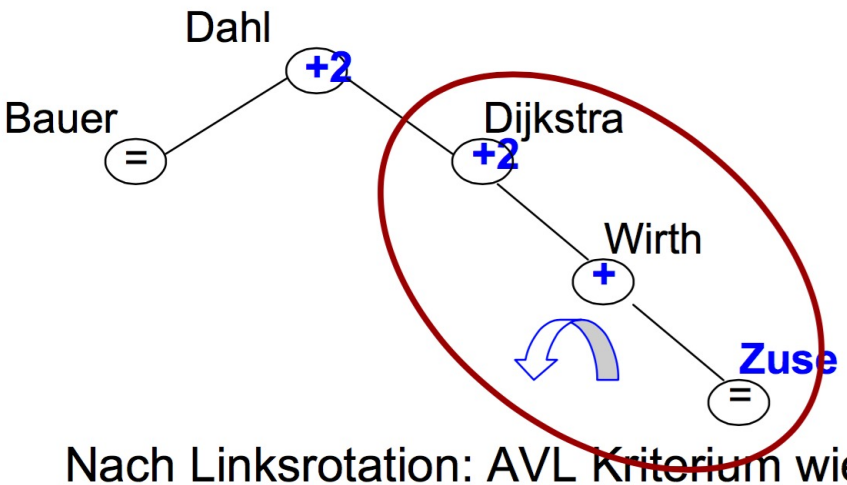


Einfügen von Wirth:

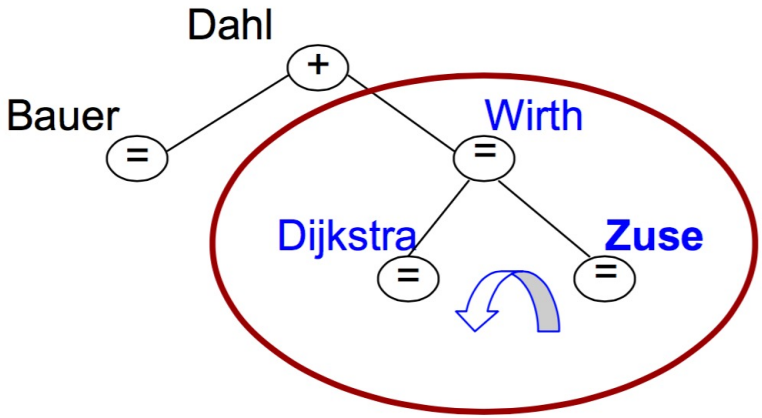
Nach Neuberechnung des Balancierungsfaktors ist

AVL-Kriterium weiter erfüllt

Einfügen von Zuse:

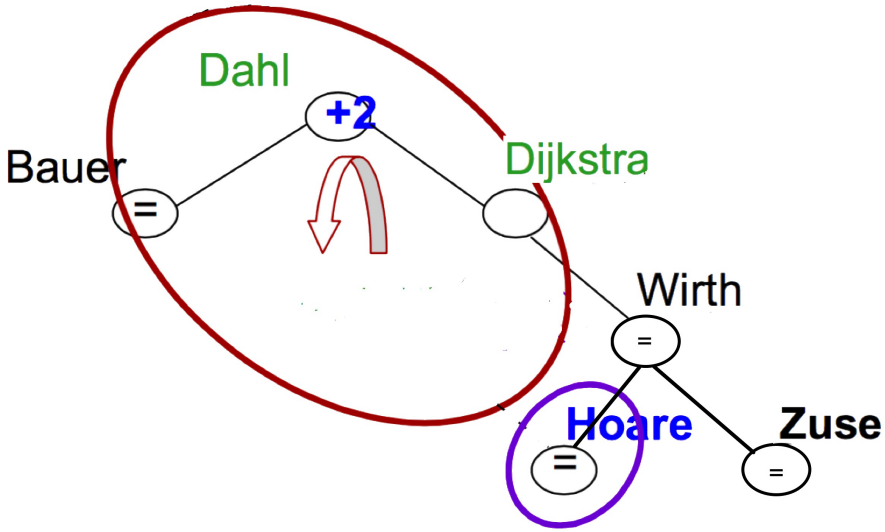


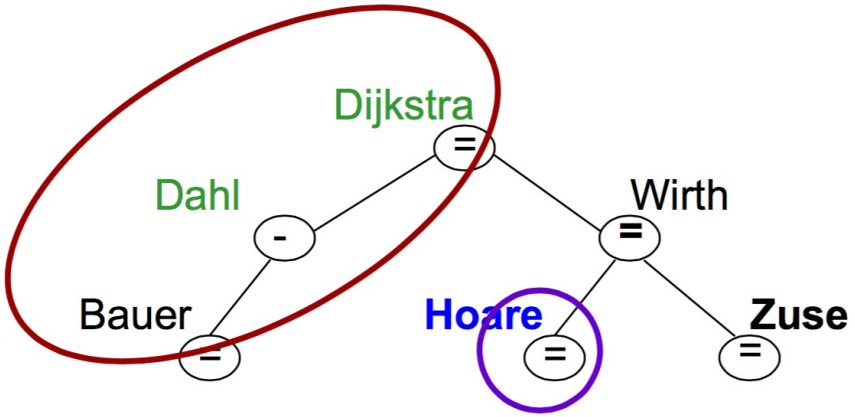
Links-rotation



Nach Linksrotation: AVL Kriterium wieder erfüllt

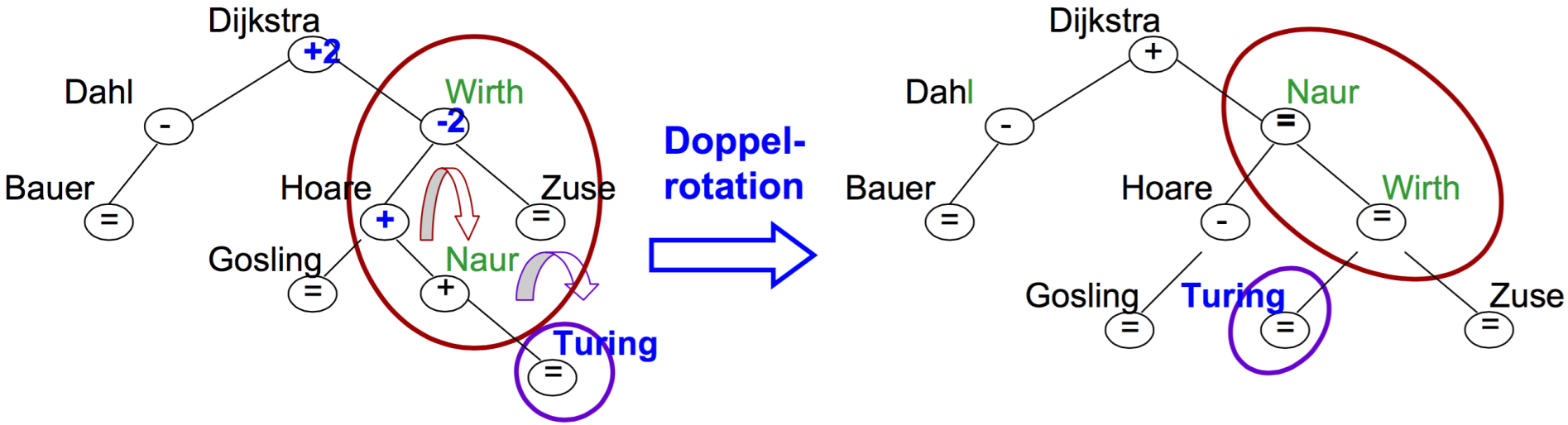
Einfügen in AVL-Baum





Einfügen in AVL-Baum

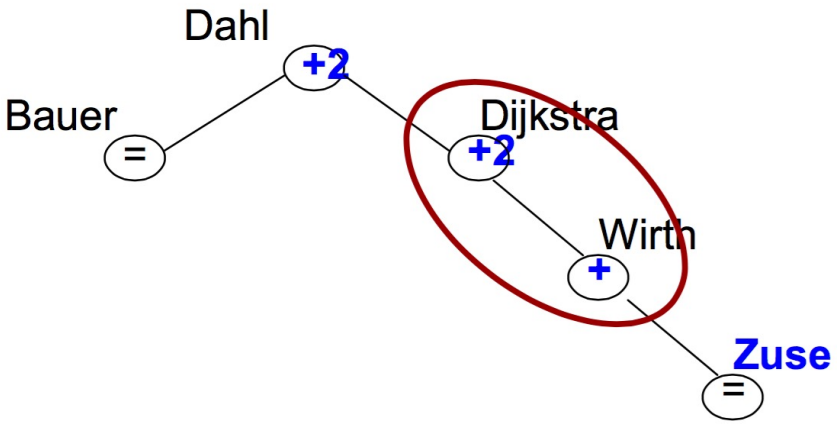
Einfügen von Turing: Noch einmal Doppelrotation

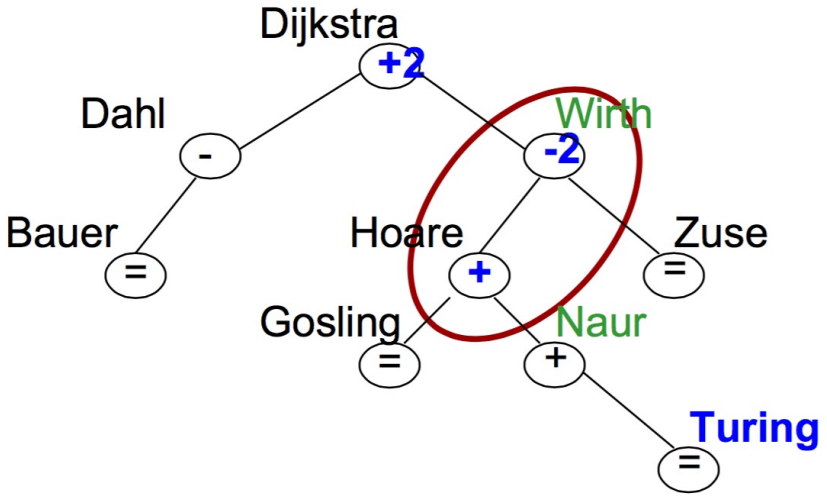


- Doppelrotation stellt AVL Kriterium wieder her
- Sind die vorgestellten Rotationen ausreichend?

Anwendungsstelle der Rotation

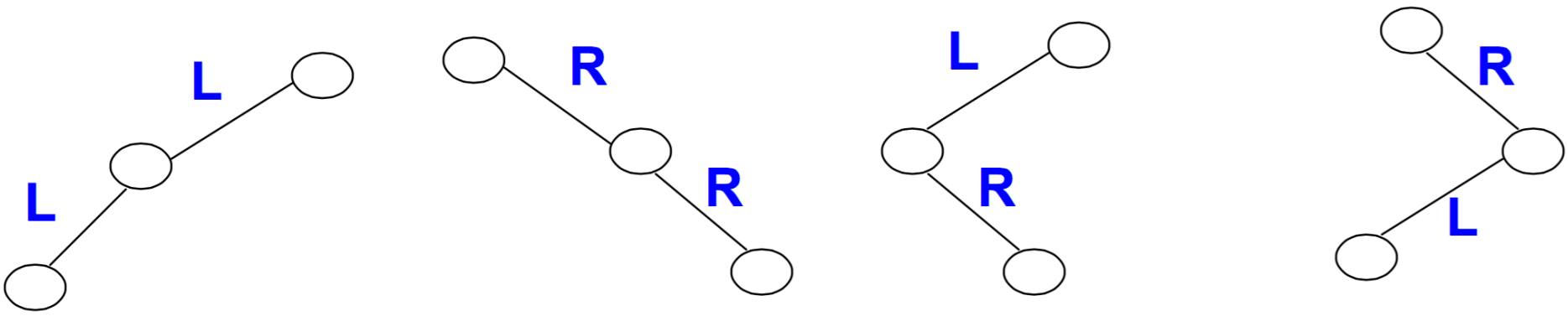
- Veränderungen der Balancierungsfaktoren geschehen ausschließlich auf dem **Pfad von der Wurzel zur Einfügeposition**
- Ausgangspunkt der Rotation ist immer der „**tiefste“ Elternknoten mit BF = ±2** (dieser Knoten hatte vorher BF = ±1)
- Der (auf dem Pfad) **darunter liegende Knoten** hat **BF = ±1**





Rotationstypen

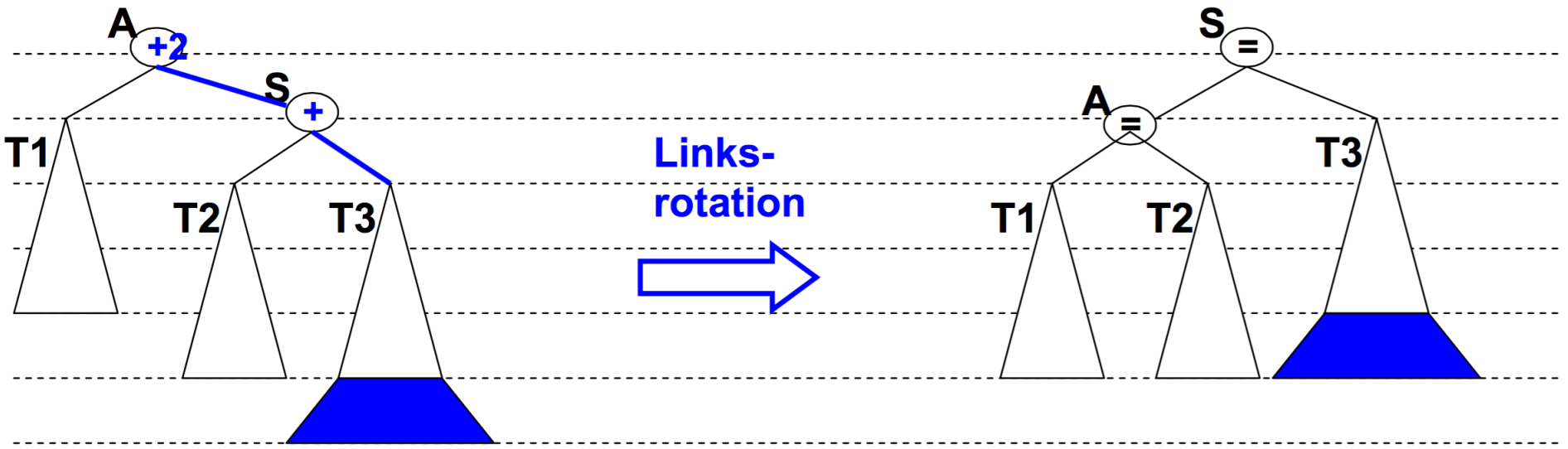
- Betrachte ausgehend vom tiefsten Knoten mit BF = 2 den Pfad zur Einfügeposition:
  - **RR: Rechts-Rechts** Linksrotation
  - **LL: Links-Links** Rechtsrotation
  - **RL: Rechts-Links** Doppelrotation „rechts“
  - **LR: Links-Rechts** Doppelrotation „links“



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

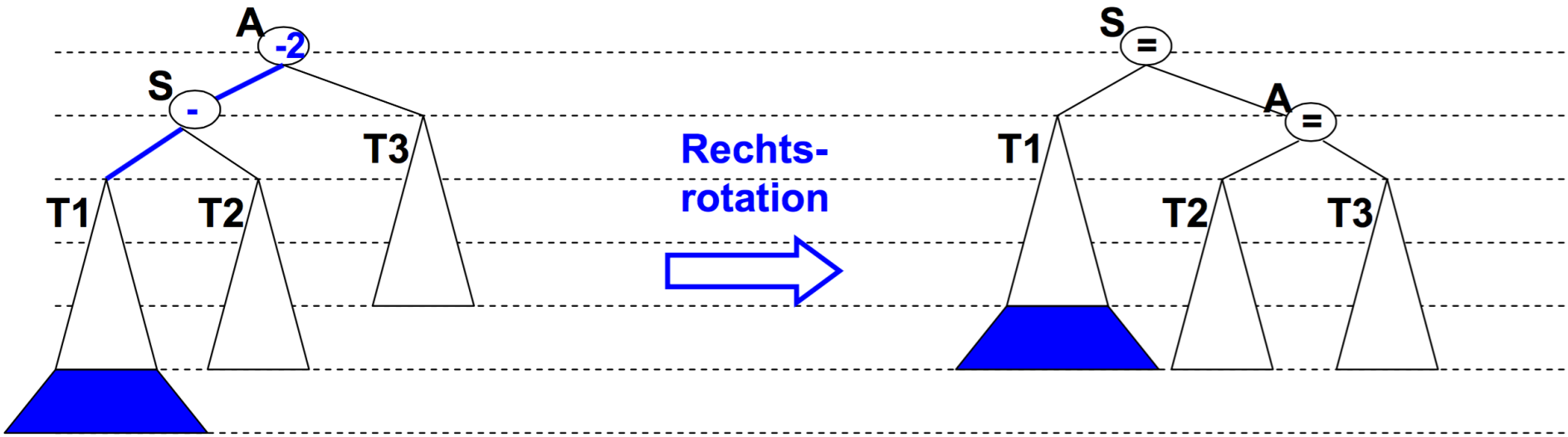
Typ RR: Linksrotation

- 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“



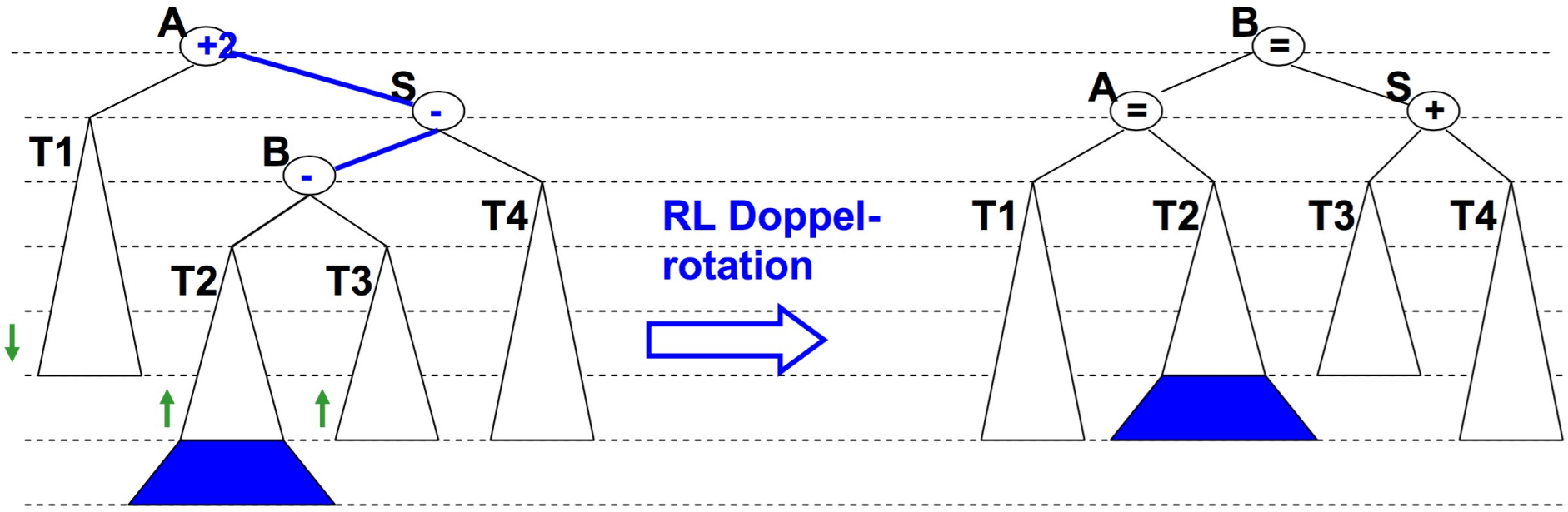
Typ LL: Rechtsrotation

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



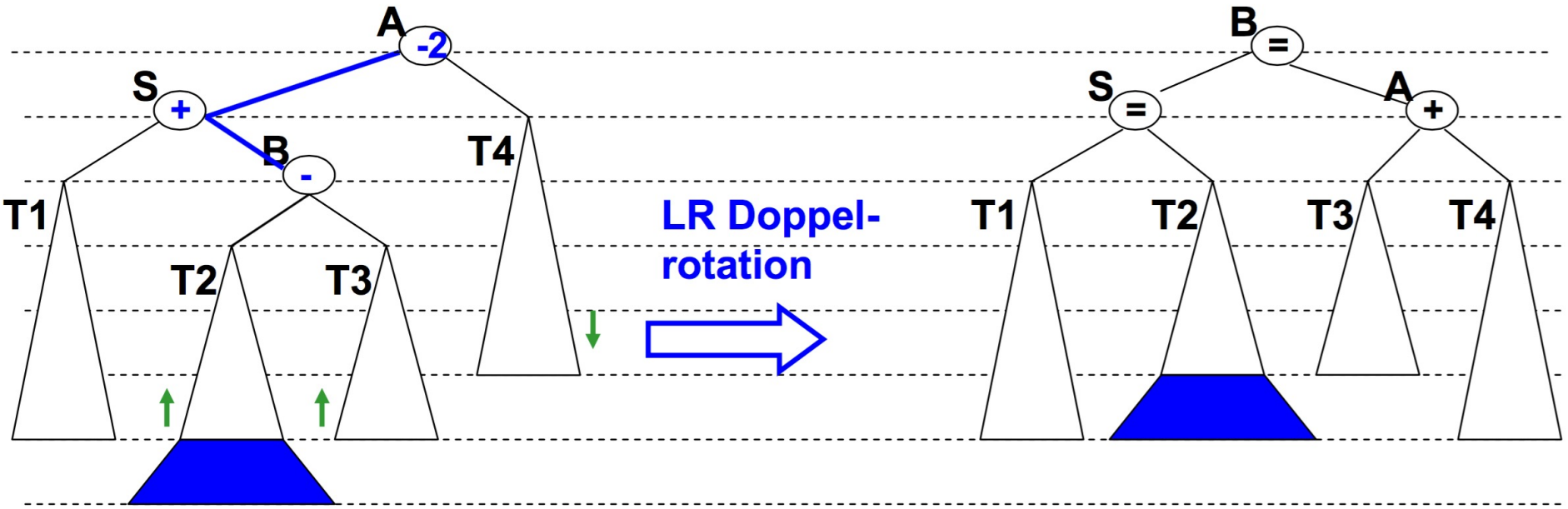
Typ RL: Doppelrotation

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



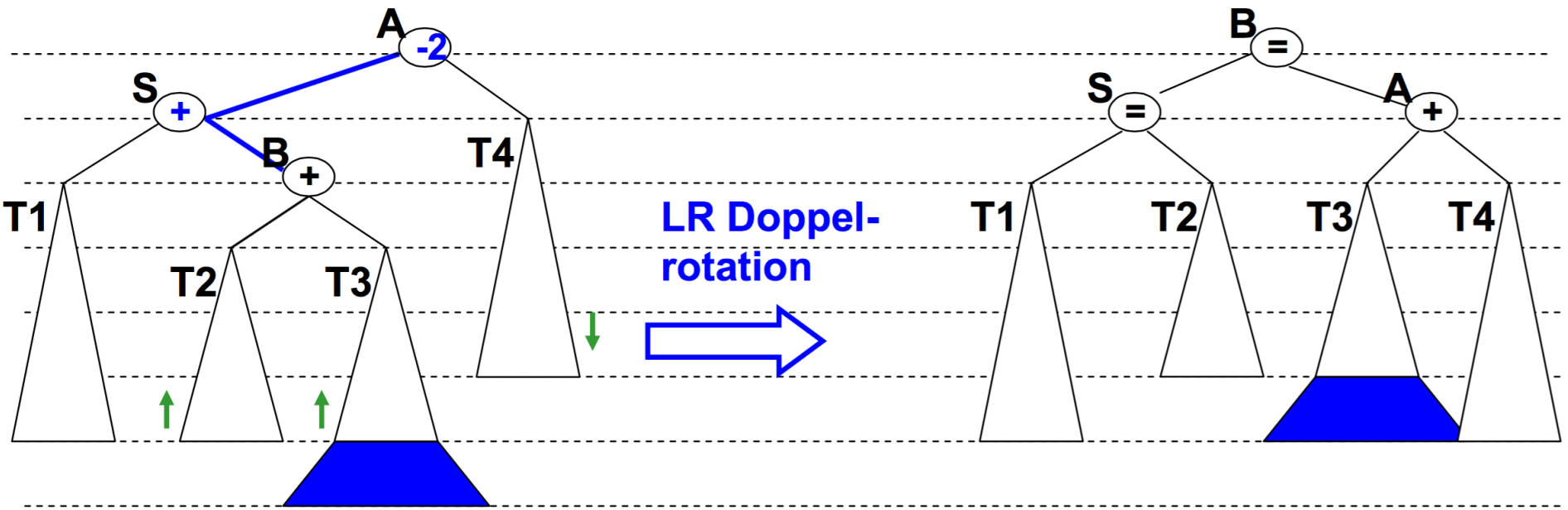
Typ LR: Doppelrotation

- 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 zu Strukturverletzungen führen (wie beim Einfügen), ...
- … die durch Rotationen ausgeglichen werden
  - Es genügt nicht immer eine einzige Rotation oder Doppelrotation (Restrukturierung)
  - Im schlechtesten Fall:
    - auf dem Suchpfad bottom-up vom zu entfernenden Schlüssel bis zur Wurzel
    - auf jedem Level Rotation bzw. Doppelrotation
    - Aufwand O(log n)

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)