Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
    - Einfügen
    - Vorgänger und Nachfolger
    - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

Holger Dell

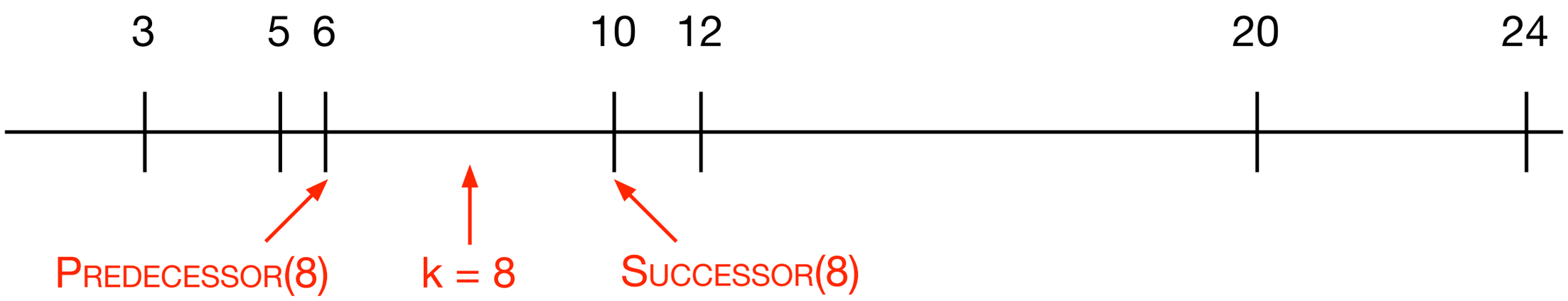
Folien adaptiert von Philip Bille

Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
  - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

Nächster Nachbar

- Nächster Nachbar. Verwalte dynamische Menge S, die folgende Operationen unterstützt. Jedes Element hat einen Schlüssel x.key und Daten x.data.
  - PREDECESSOR(k): liefere größten Schlüssel ≤ k.
  - SUCCESSOR(k): liefere kleinsten Schlüssel ≥ k.
  - INSERT(x): füge x zu S hinzu (wir nehmen an, dass x noch nicht in S ist)
  - DELETE(x): entferne x von S.



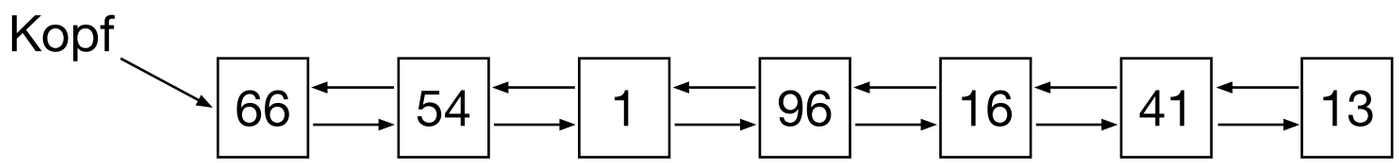
Nächster Nachbar

- Anwendungen.
  - Suchen nach ähnlichen Daten (typischerweise multidimensional)
  - Routing im Internet.

- Frage. Wie würden wir die abstrakte Datenstruktur implementieren mit dem was wir schon kennen?

Nächster Nachbar

- Lösung 1: Verkettete Liste. Verwalte S in einer doppelt verkettete Liste.



- PREDECESSOR(k): lineare Suche nach größtem Schlüssel ≤ k.
- SUCCESSOR(k): lineare Suche nach kleinstem Schlüssel ≥ k.
- INSERT(x): füge x vorne in die Liste ein.
- DELETE(x): entferne x aus der Liste.

- Zeit.
  - PREDECESSOR und SUCCESSOR in O(n) Zeit (n = |S|).
  - INSERT und DELETE in O(1) Zeit.
- Platz.
  - O(n).

Nächster Nachbar

- Lösung 2: Sortiertes Feld. Verwalte S in einem sortierten Feld.

1	2	3	4	5	6	7
1	13	16	41	54	66	96

- PREDECESSOR(k): binäre Suche nach größtem Schlüssel ≤ k.
- SUCCESSOR(k): binäre Suche nach kleinstem Schlüssel ≥ k.
- INSERT(x): konstruiere ein neues Feld der Größe +1 wo x eingefügt ist.
- DELETE(x): konstruiere ein neues Feld der Größe –1 wo x entfernt ist.

- Zeit.
  - PREDECESSOR und SUCCESSOR in O(log n) Zeit.
  - INSERT und DELETE in O(n) Zeit.
- Platz.
  - O(n).

Nächster Nachbar

Datenstruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Platz
verkettete Liste	O(n)	O(n)	O(1)	O(1)	O(n)
Sortiertes Feld	O(log n)	O(log n)	O(n)	O(n)	O(n)

- Frage. Geht das signifikant besser?

Binäre Suchbäume

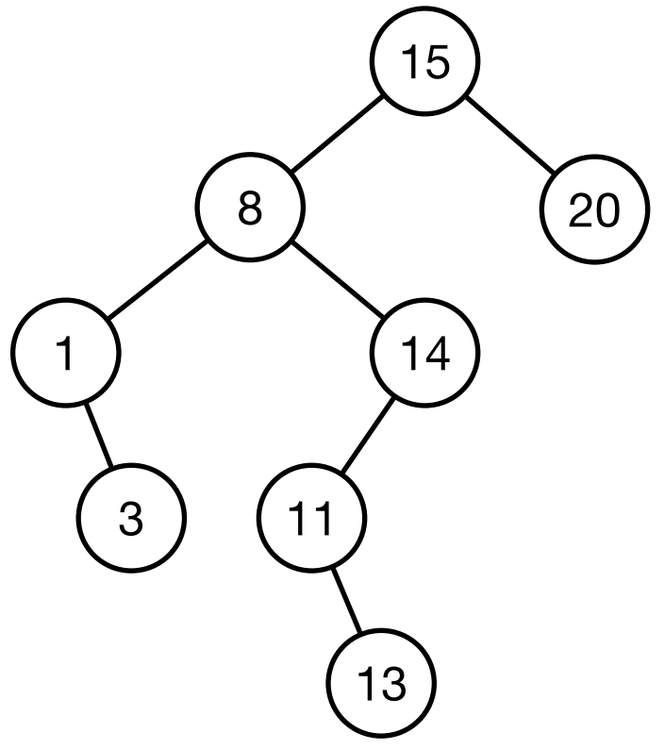
- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
    - Vorgänger und Nachfolger
    - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

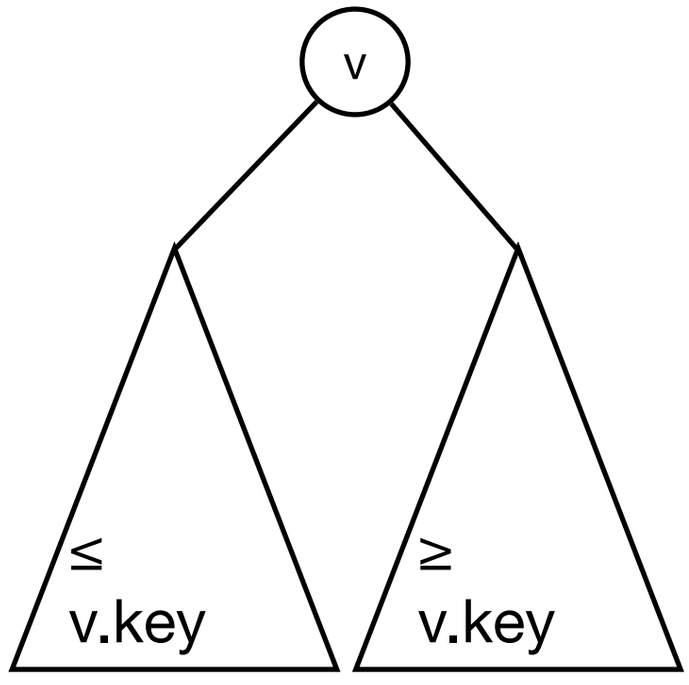
Binäre Suchbäume

- Binärbaum. Gewurzelter Baum, wo jeder interne Knoten ein linkes Kind und/oder ein rechtes Kind hat.

- Binärer Suchbaum. Binärbaum, der die Suchbaumeigenschaft erfüllt.

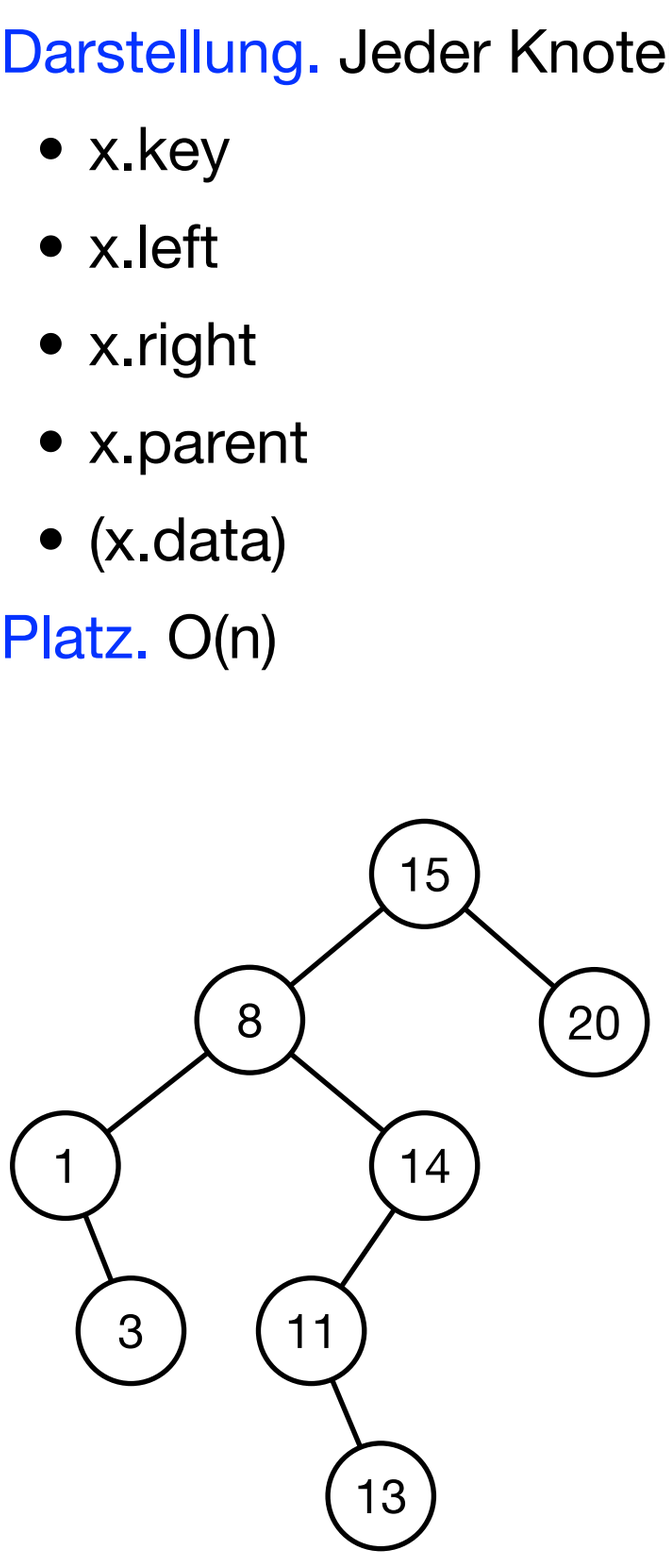
- Suchbaumeigenschaft.
  - Jeder Knoten speichert ein Element.
  - Für jeden Knoten v gilt:
    - alle Knoten im linken Unterbaum haben Schlüssel ≤ v.key.
    - alle Knoten im rechten Unterbaum haben Schlüssel ≥ v.key.

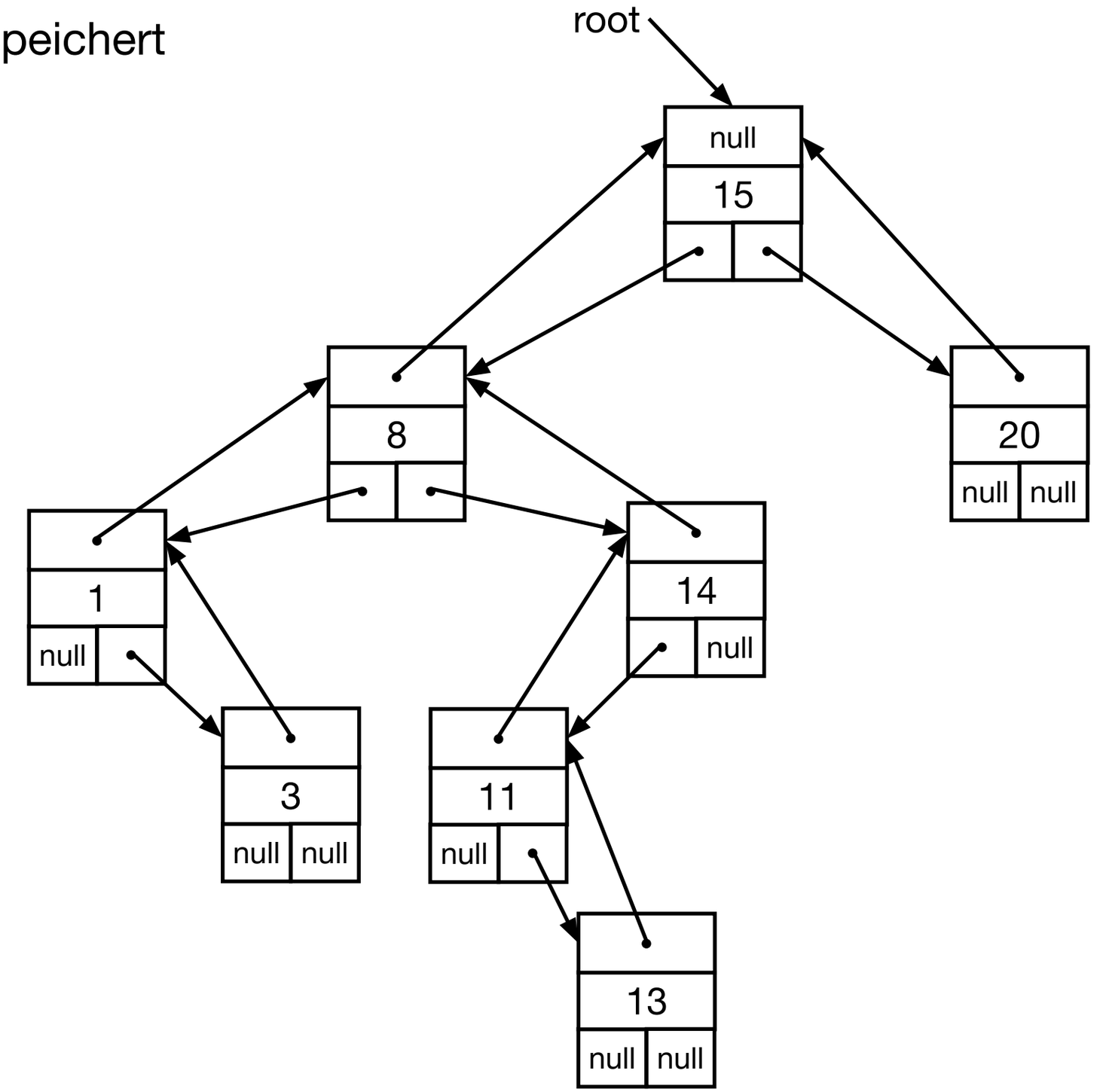




Binäre Suchbäume

- Darstellung. Jeder Knoten x speichert
  - x.key
  - x.left
  - x.right
  - x.parent
  - (x.data)
- Platz. O(n)



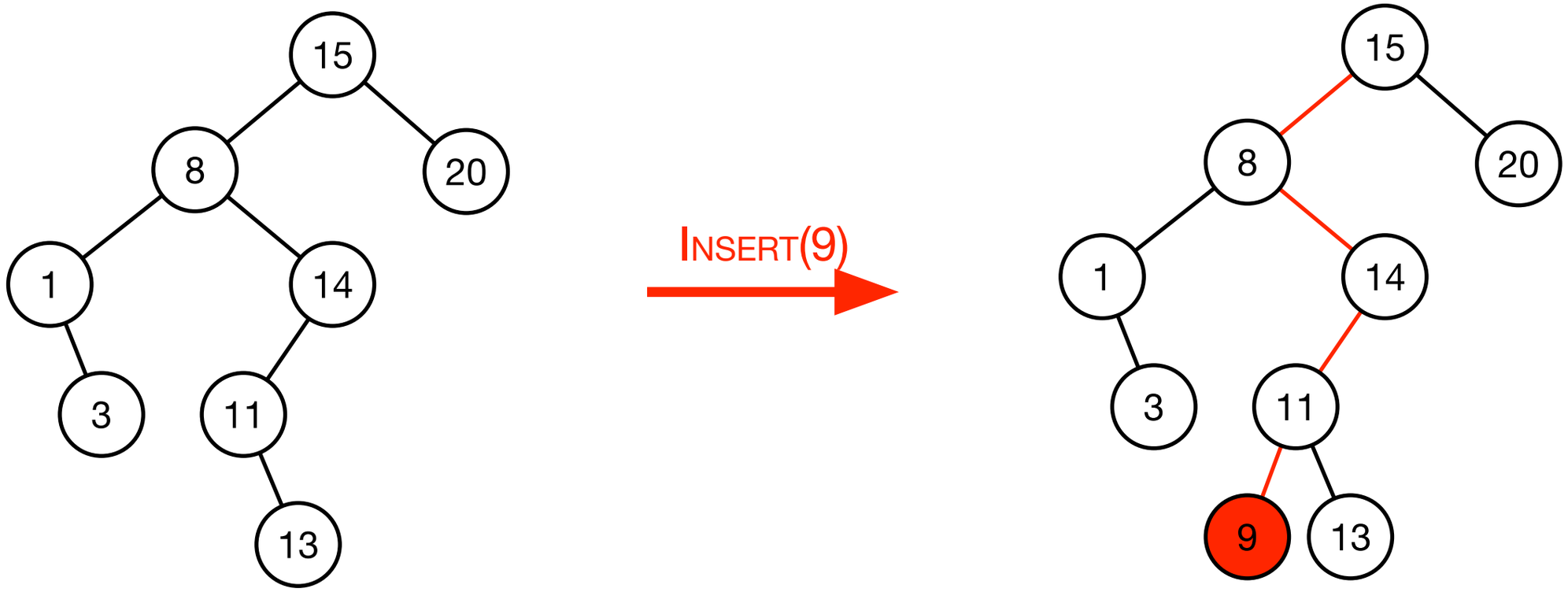


Binäre Suchbäume

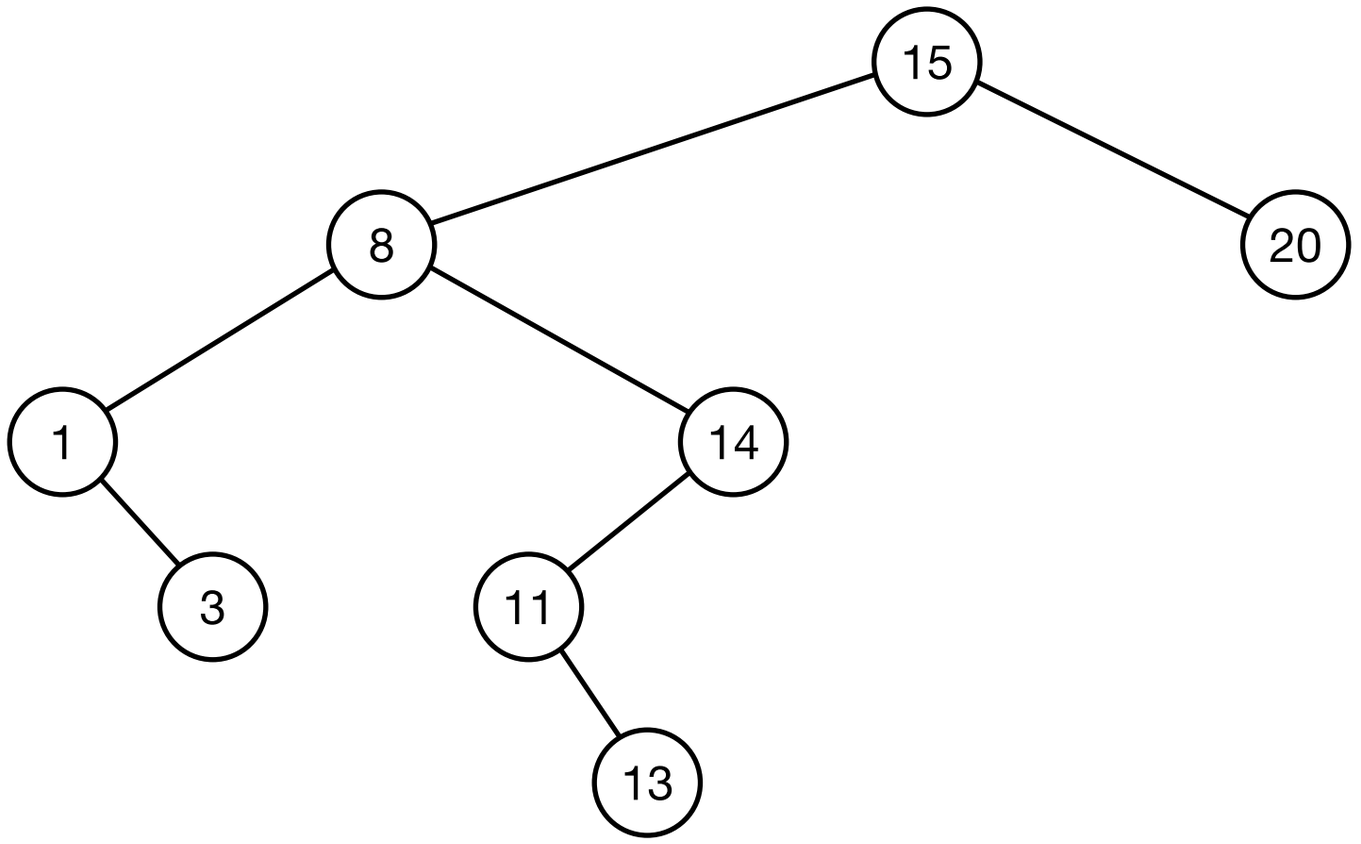
- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
  - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

Einfügen

- INSERT(x): starte in Wurzel. Am Knoten v:
  - wenn x.key ≤ v.key geh nach links.
  - wenn x.key > v.key geh nach rechts.
  - wenn null, füge x hier ein



INSERT 15 8 20 14 1 3 11 13



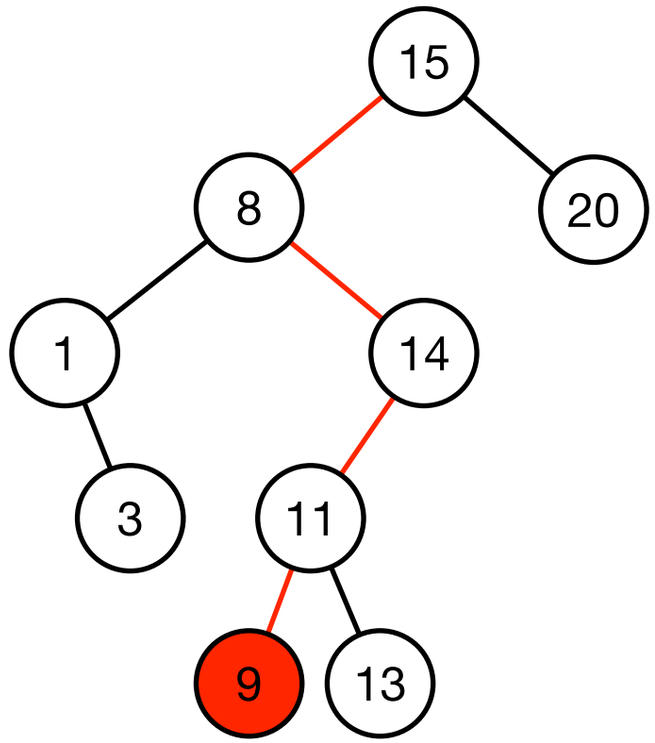
Einfügen

- INSERT(x): starte in Wurzel. Am Knoten v:
  - wenn x.key ≤ v.key geh nach links.
  - wenn x.key > v.key geh nach rechts.
  - wenn null, füge x hier ein

- Übung. Füge die folgende Sequenz in einen binären Suchbaum ein: 6, 14, 3, 8, 12, 9, 34, 1, 7

Einfügen

```
INSERT(x,v)
   if (v == null) return x
   if (x.key ≤ v.key)
      v.left = INSERT(x, v.left)
   if (x.key > v.key)
      v.right = INSERT(x, v.right)
   return v
```



- Zeit. O(h)

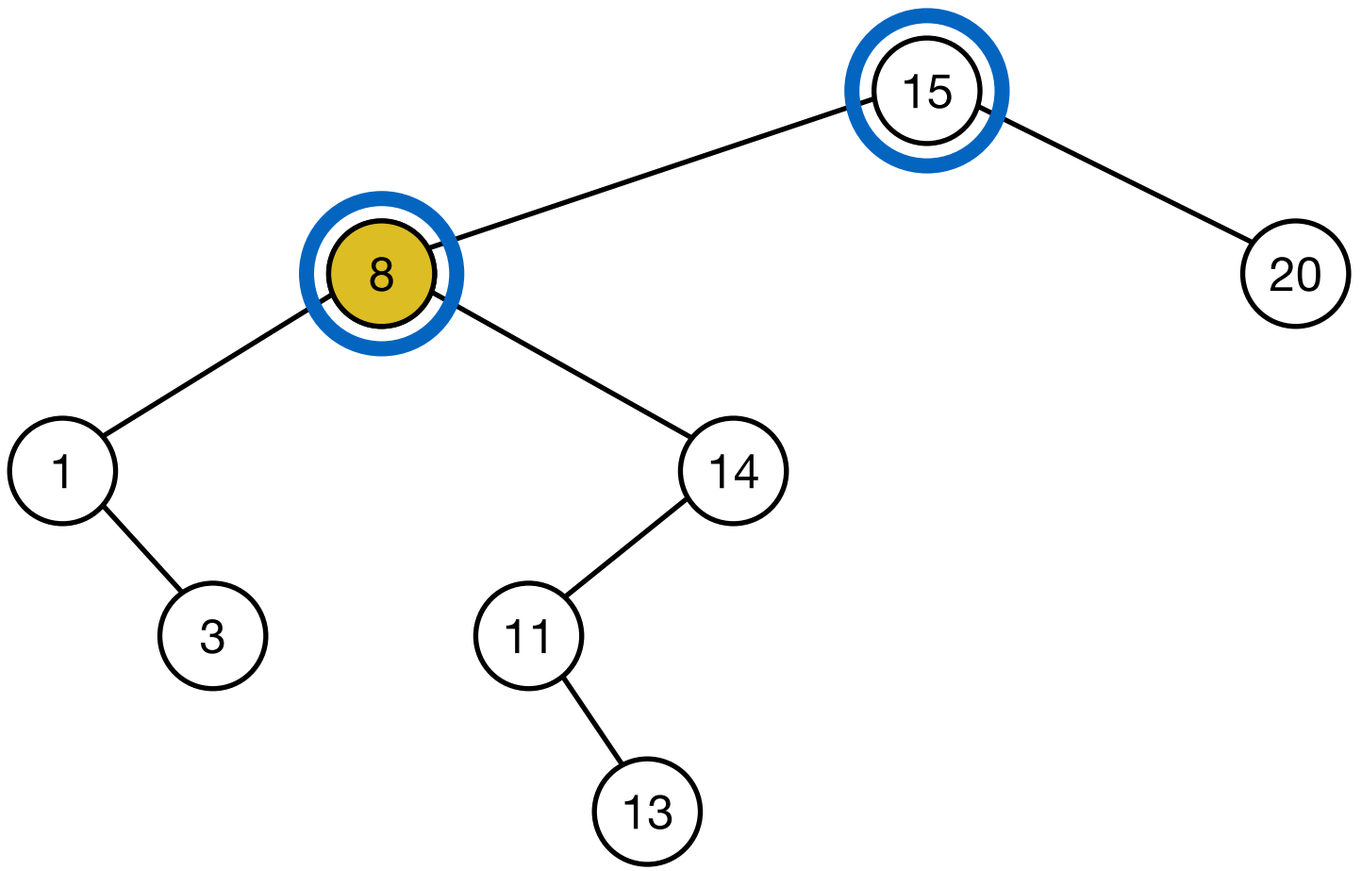
Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
  - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

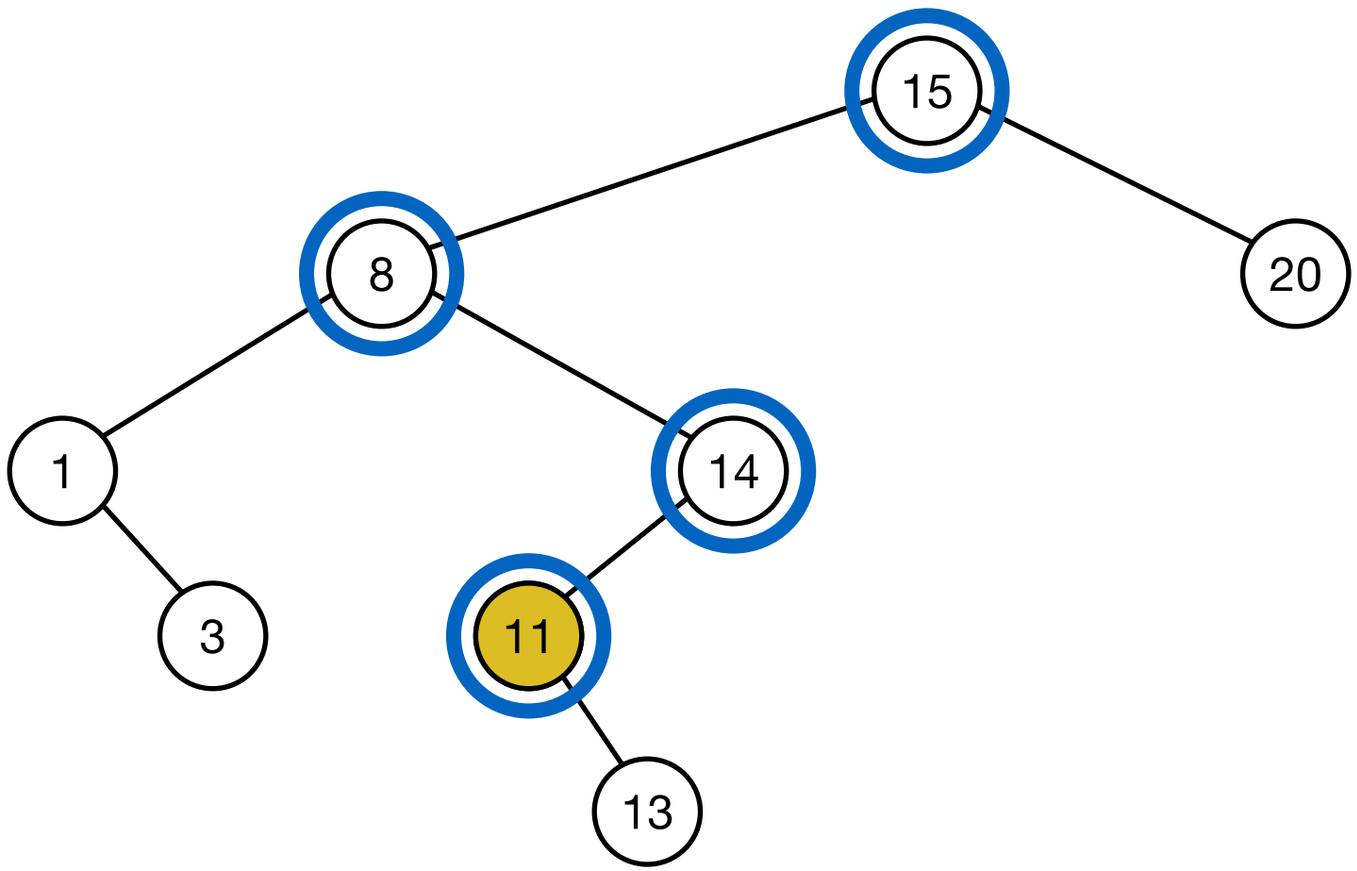
Vorgänger

- PREDECESSOR(k): starte in Wurzel. Am Knoten v:
  - wenn v == null, dann liefere null.
  - wenn k == v.key, dann liefere v.
  - wenn k < v.key, dann suche im linken Unterbaum weiter.
  - wenn k > v.key, dann suche im rechten Unterbaum weiter.
    - Wenn es ein Element x mit key ≤ k im rechten Unterbaum gibt, liefere x.
    - Ansonsten liefere v

PREDECESSOR 8

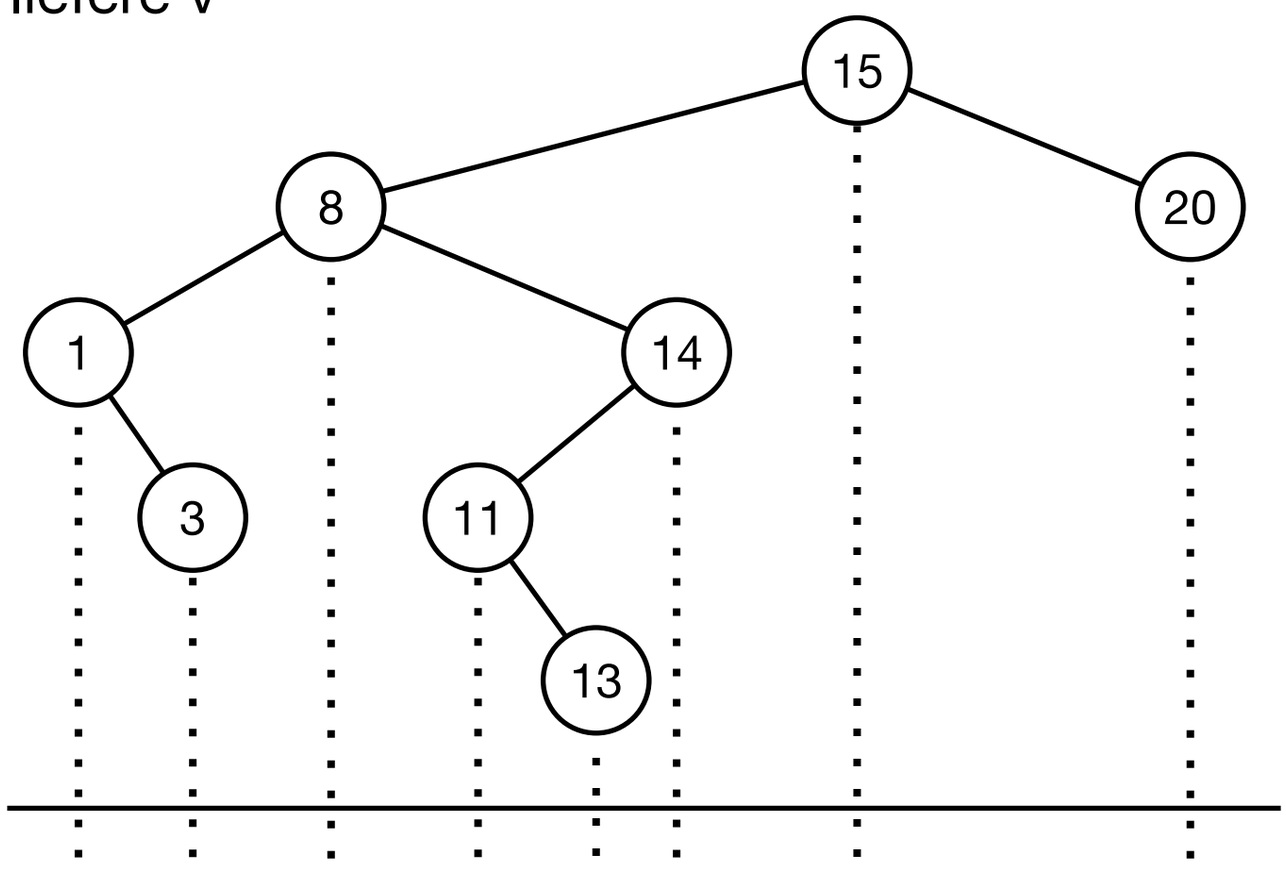


PREDECESSOR 12



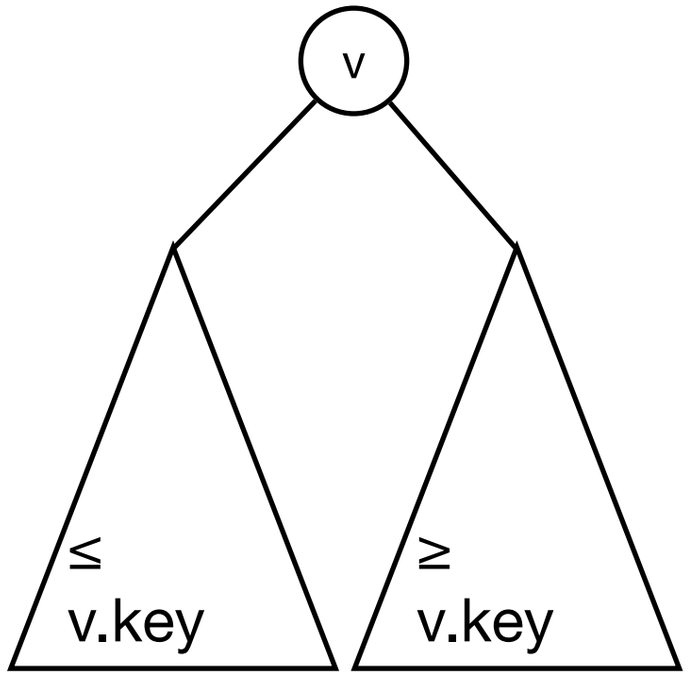
Vorgänger

- PREDECESSOR(k): starte in Wurzel. Am Knoten v:
  - wenn v == null, dann liefere null.
  - wenn k == v.key, dann liefere v.
  - wenn k < v.key, dann suche im linken Unterbaum weiter.
  - wenn k > v.key, dann suche im rechten Unterbaum weiter.
    - Wenn es ein Element x mit key ≤ k im rechten Unterbaum gibt, liefere x.
    - Ansonsten liefere v



Vorgänger

```
PREDECESSOR(v, k)
   if (v == null) return null
   if (v.key == k) return v
   if (k < v.key)
      return PREDECESSOR(v.left, k)
   t = PREDECESSOR(v.right, k)
   if (t ≠ null) return t
   else return v
```



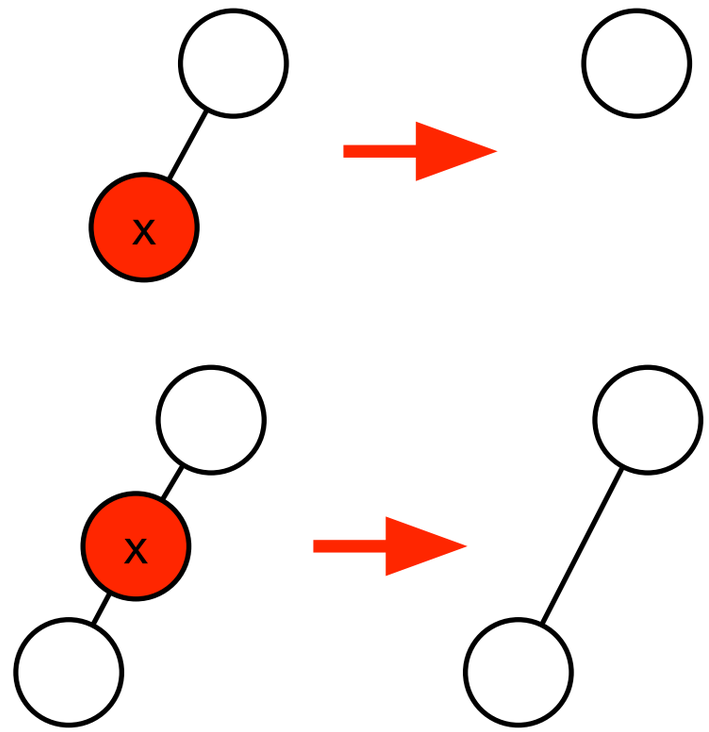
- Zeit. O(h)
- SUCCESSOR mit ähnlichem Algorithmus in Zeit O(h).

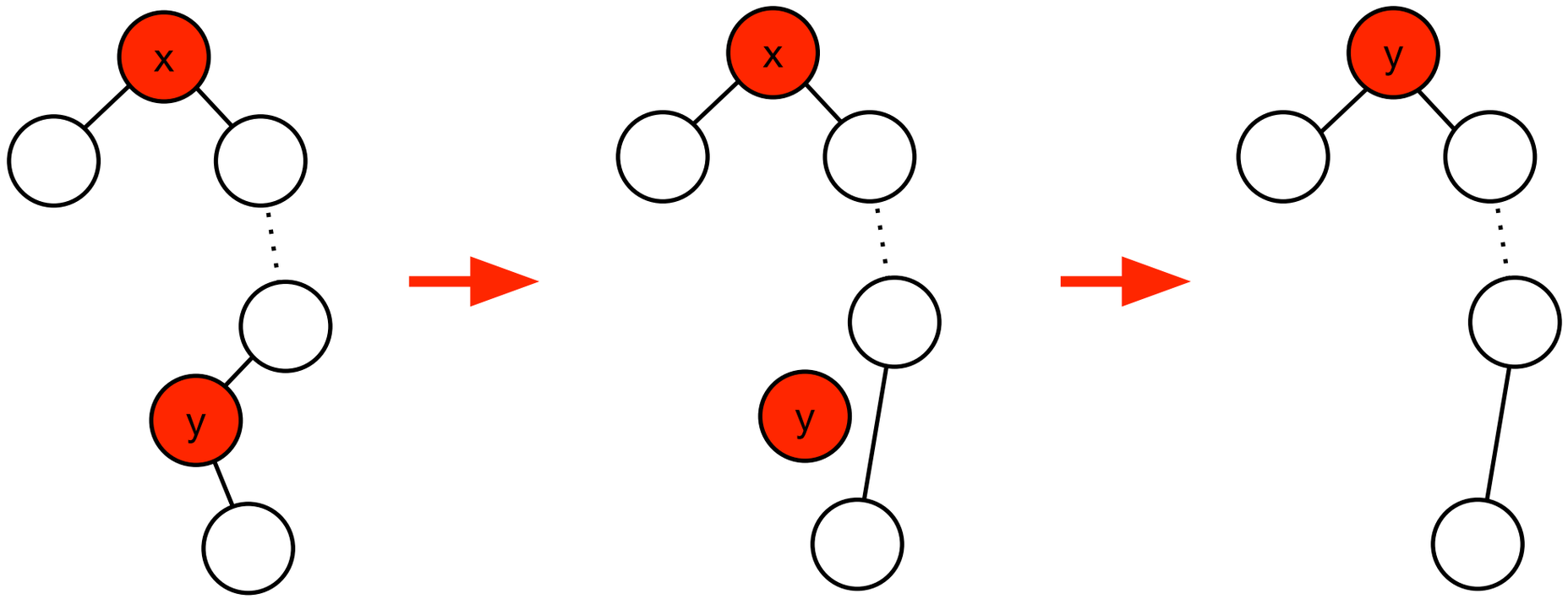
Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
  - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

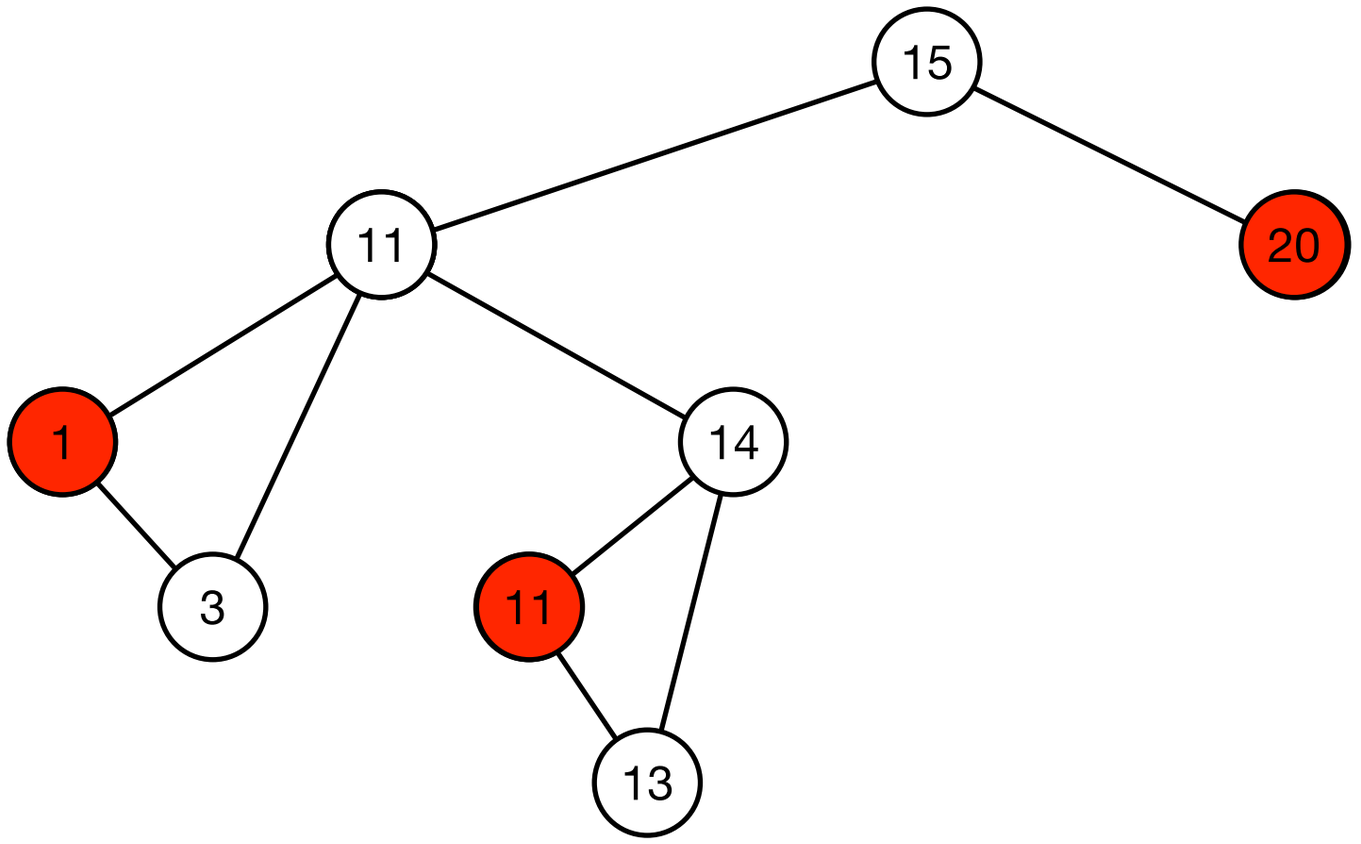
Löschen

- DELETE(x):
  - 0 Kinder: lösche x.
  - 1 Kind: kontrahiere x.
  - 2 Kinder:
    Finde y = Knoten mit kleinstem Schlüssel > x.key.
    Kontrahiere y und ersetze x durch y.





DELETE 20 1 8



Löschen

- DELETE(x):
  - 0 Kinder: lösche x.
  - 1 Kind: kontrahiere x.
  - 2 Kinder:
    Finde y = Knoten mit kleinstem Schlüssel > x.key.
    Kontrahiere y und ersetze x durch y.

- **Zeit.** **O(h)**

Nächster Nachbar

Datenstruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Platz
verkettete Liste	O(n)	O(n)	O(1)	O(1)	O(n)
Sortiertes Feld	O(log n)	O(log n)	O(n)	O(n)	O(n)
Binärer Suchbaum	O(h)	O(h)	O(h)	O(h)	O(n)

- Höhe h. Hängt von der Sequenz von Operationen ab.
  - h = Ω(n) im schlimmsten Fall und h = Θ(log n) im mittleren Fall (*average case*).

- Frage. Wie können wir die Höhe klein halten?

Binäre Suchbäume

- Nächster Nachbar
  - PREDECESSOR(k): liefere größten Schlüssel ≤ k.
  - SUCCESSOR(k): liefere kleinsten Schlüssel ≥ k.
  - INSERT(x): füge x zu S hinzu (wir nehmen an, dass x noch nicht in S ist)
  - DELETE(x): entferne x von S.

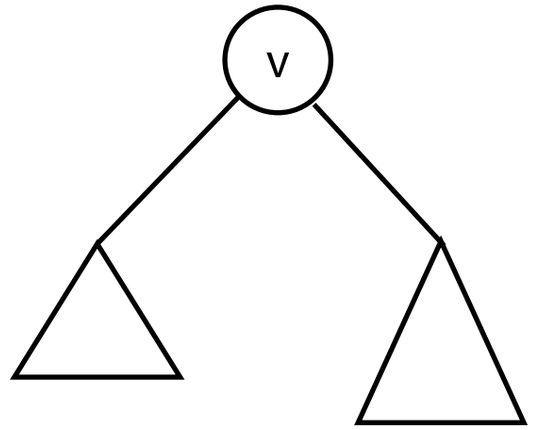
- Andere Operationen auf binären Suchbäumen.
  - SEARCH(k): stelle fest, ob Element mit Schlüssel k in S ist und liefere es zurück.
  - TREE-SEARCH(x, k): stelle fest, ob das Element mit Schlüssel k im Unterbaum von Knoten x vorliegt, und liefere es zurück.
  - TREE-MIN(x): liefere das kleinste Element im Unterbaum von x.
  - TREE-MAX(x): liefere das größte Element im Unterbaum von x.
  - TREE-PREDECESSOR(x): liefere das Element mit größtem Schlüssel ≤ x.key.
  - TREE-SUCCESSOR(x): liefere das Element mit kleinstem Schlüssel ≥ x.key.

Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
  - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

AVL-Bäume

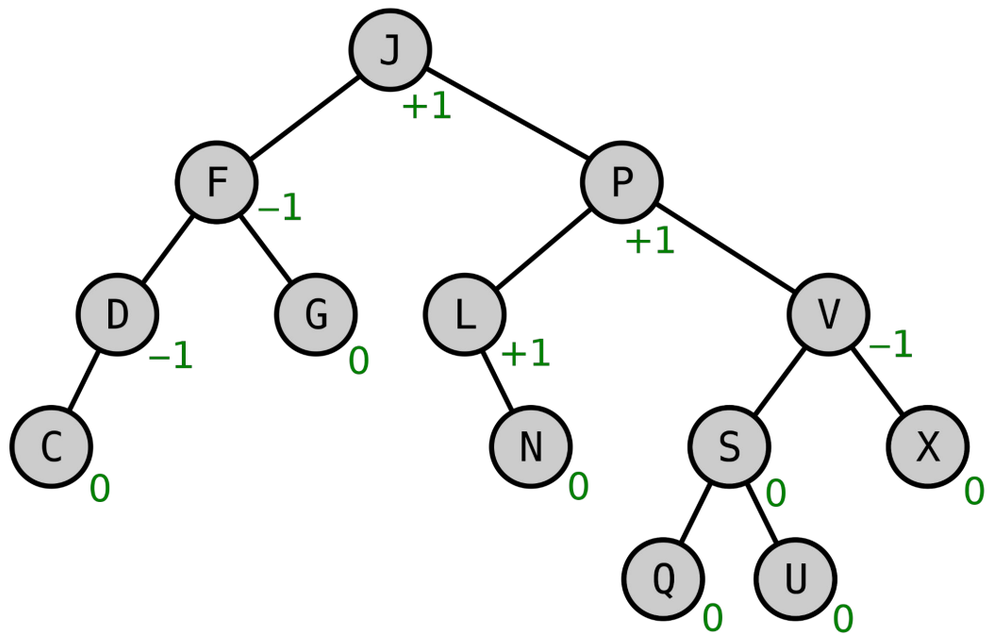
- AVL-Baum.
  - Binärer Suchbaum.
  - Speichert an jedem Knoten v zusätzlich dessen Höhe `v.height`.
  - Erhält AVL-Eigenschaft beim Einfügen/Löschen.



- AVL-Eigenschaft an Knoten v.
  - Die Höhen von v.left und v.right unterscheiden sich um ≤ 1.

- Balancefaktor.
  - BF(v) = v.right.height – v.left.height

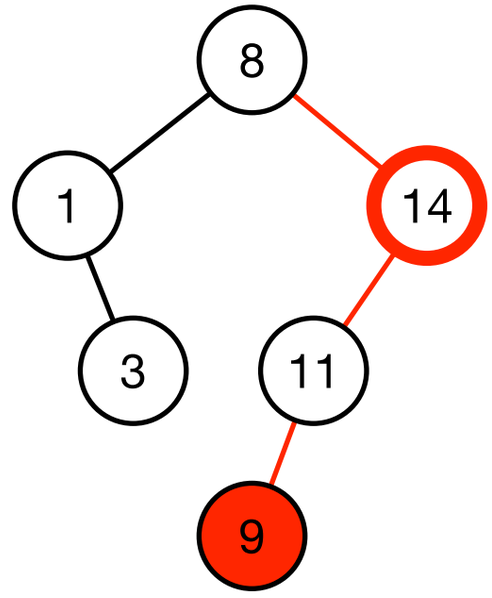
- AVL-Eigenschaft. BF(v) ∈ {–1, 0, +1} für alle v.



Einfügen in AVL-Baum

- Einfügen. Füge Element x in AVL-Baum mit Wurzel v ein:

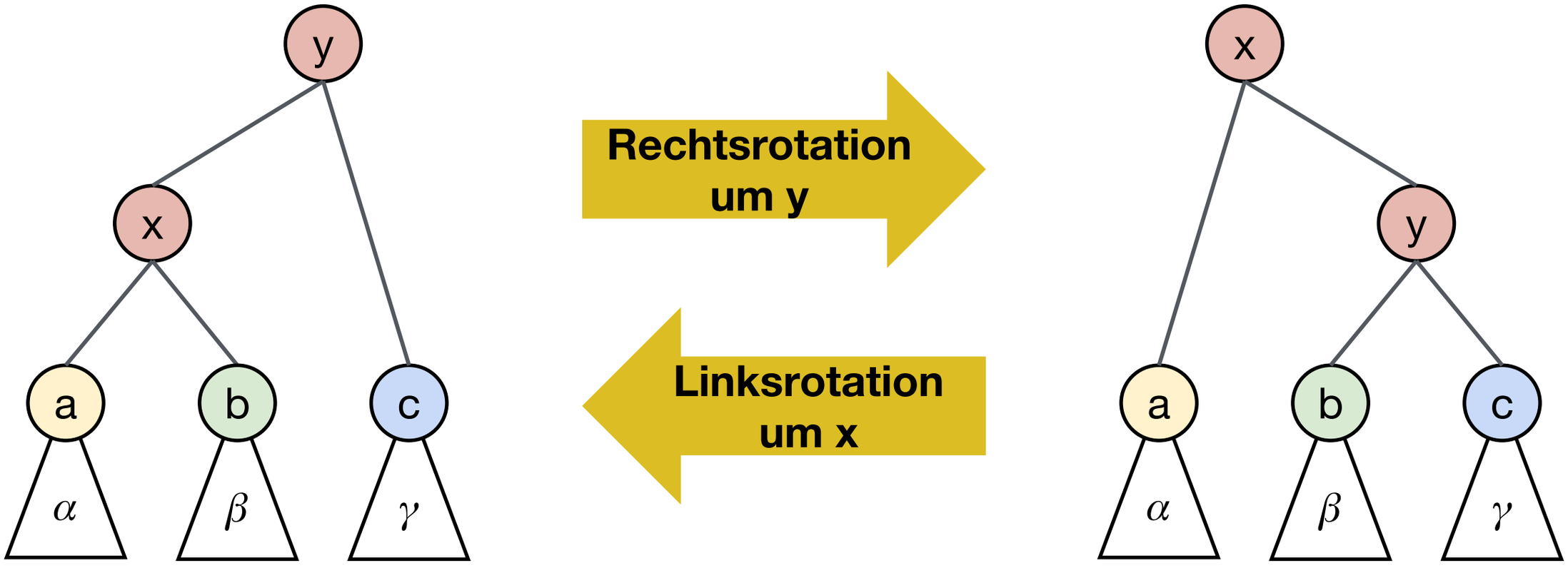
  1. Füge x rekursiv im linken oder rechten Unterbaum von v ein.
  2. Wenn BF(v) = **±2**, rebalanciere v.
  3. Aktualisiere die Höhen.



- Eigenschaften. **Nach** Schritt 1, aber **vor** Schritt 2:
  - Neuer linker und rechter Unterbaum erfüllen induktiv die AVL-Eigenschaft.
  - Die Höhe des Unterbaums, wo x eingefügt wurde, ist maximal 1 größer.
  - ⇒ BF(v) um maximal 1 geändert.
  - ⇒ Mögliche Werte von BF(v) sind {0, ±1, **±2**}.

Rotationen

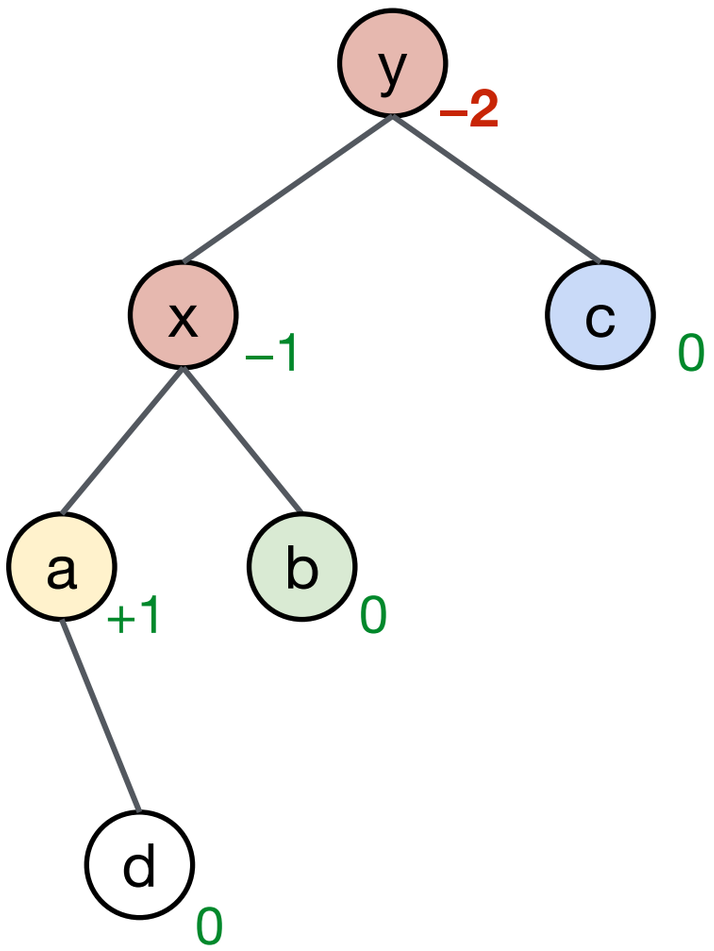
- Einzelner Defekt. **BF(y) = ±2**, alle anderen Knoten haben BF(v) = 0, ±1.
- Rotationen.
  - Erhalten die Suchbaumeigenschaft.
  - Beeinflussen die Höhen von x und y.
- Frage. Können Rotationen einzelne Defekte beheben?



Rotationen lösen Defekttyp 1

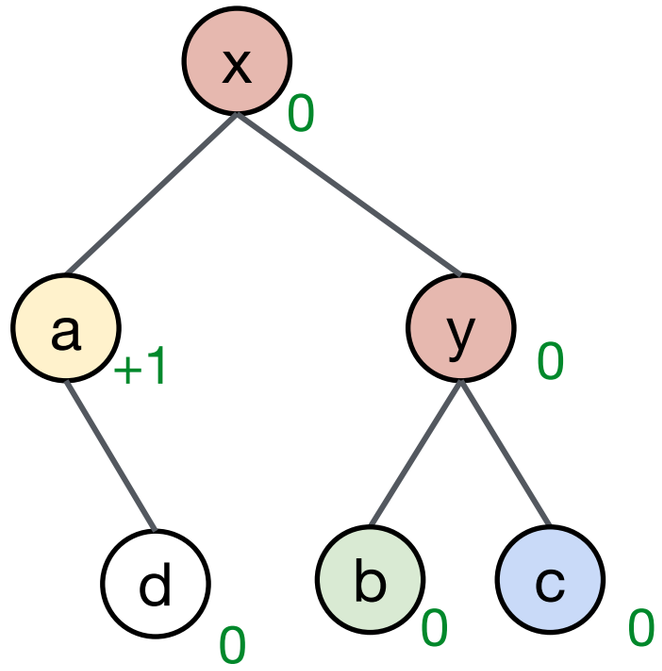
Defekttyp 1.
BF(y) = **–2** und linkes Kind x von y hat BF(x) ≤ 0.

Defekttyp 1 in y.



Rechtsrotation um y

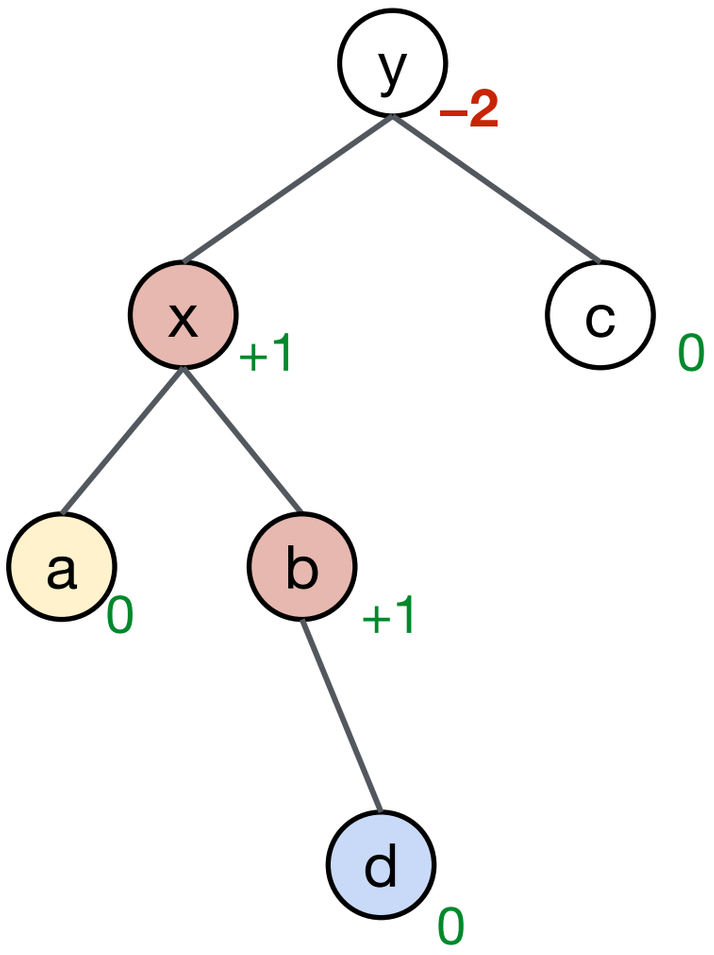
kein Defekt



Rotationen reduzieren Defekttyp 2 auf Defekttyp 1

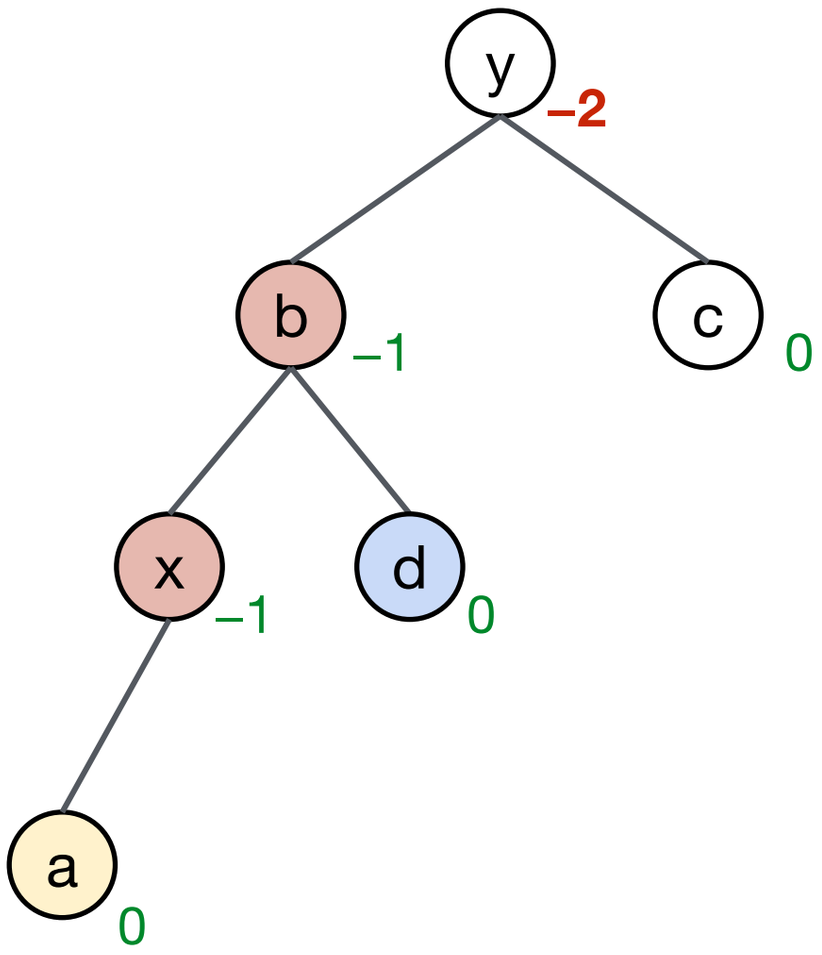
Defekttyp 2.
BF(y) = **–2** und linkes Kind x von y hat BF(x) = +1.

Defekttyp 2 in y.



Linksrotation um x

Defekttyp 1 in y.

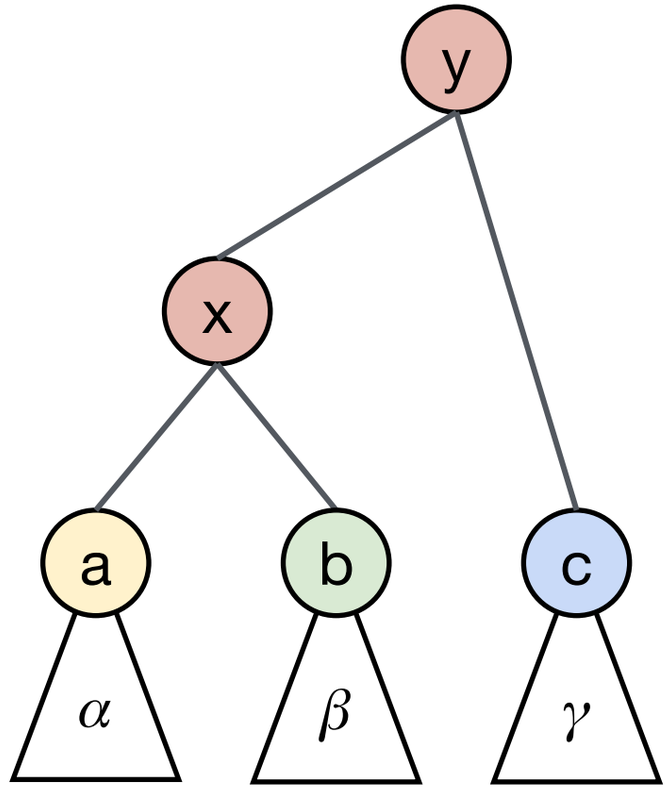


Für vollständige Lösung nutzen wir **zwei** Rotationen: Linksrotation um x, dann Rechtsrotation um y.

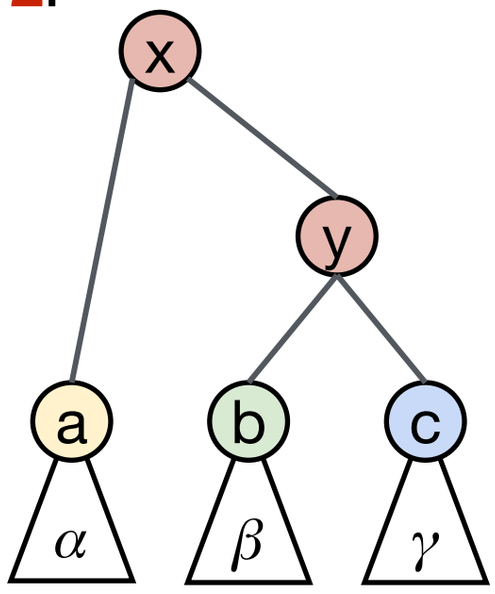
Rebalancieren

- Rebalancieren. Gegeben ein Baum, dessen Wurzel y der einzig mögliche AVL-Defekt ist mit Balancefaktor **±2**, wir wollen den Baum so rotieren, dass wieder ein AVL-Baum entsteht.

- Pseudocode. `Rebalance(y)`:
  - Wenn BF(y) = **–2**:
    - Sei x linkes Kind von y.
    - Wenn BF(x) = 0 oder BF(x) = –1:
      - Rechtsrotation um y.
    - Sonst (BF(x) = +1):
      - Linksrotation um x.
      - Rechtsrotation um y.



  - Wenn BF(y) = **+2**:
    - (Übung.)



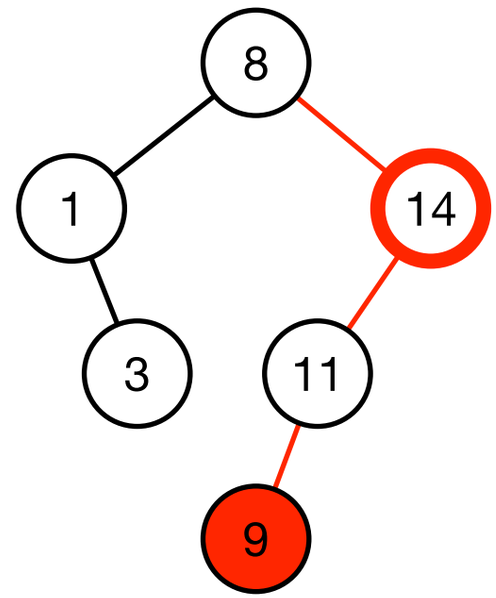
Einfügen in AVL-Baum

```
AVL-INSERT(x,v)
   if (v == null)
      x.height = 0
      return x
   if (x.key ≤ v.key)
      v.left = AVL-INSERT(x, v.left)
   if (x.key > v.key)
      v.right = AVL-INSERT(x, v.right)

   v.height =
      1 + max(v.left.height, v.right.height)

   Rebalance(v)

   return v
```



- Zeit. O(h)

Nächster Nachbar

- AVL-Bäume.
  - Verwalten binären Suchbaum
  - AVL-Eigenschaft ⇒ Höhe ≤ O(log n). (Übung.)

Datenstruktur	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Platz
Verkettete Liste	O(n)	O(n)	O(1)	O(1)	O(n)
Sortiertes Feld	O(log n)	O(log n)	O(n)	O(n)	O(n)
Binärer Suchbaum	O(h)	O(h)	O(h)	O(h)	O(n)
AVL-Bäume	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

- Andere balancierte binäre Suchbäume.
  - 2-3-Bäume, rot-schwarz Bäume, etc. verwalten binären Suchbaum von Höhe O(log n).
- Noch bessere Schranken für Nächster Nachbar sind mit fortgeschrittenen Datenstrukturen möglich.

Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
    - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung

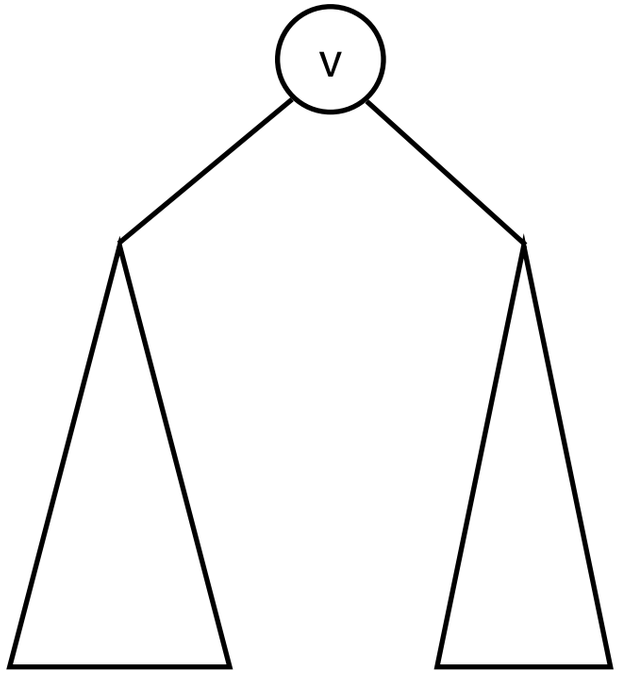
Algorithmen auf Bäumen

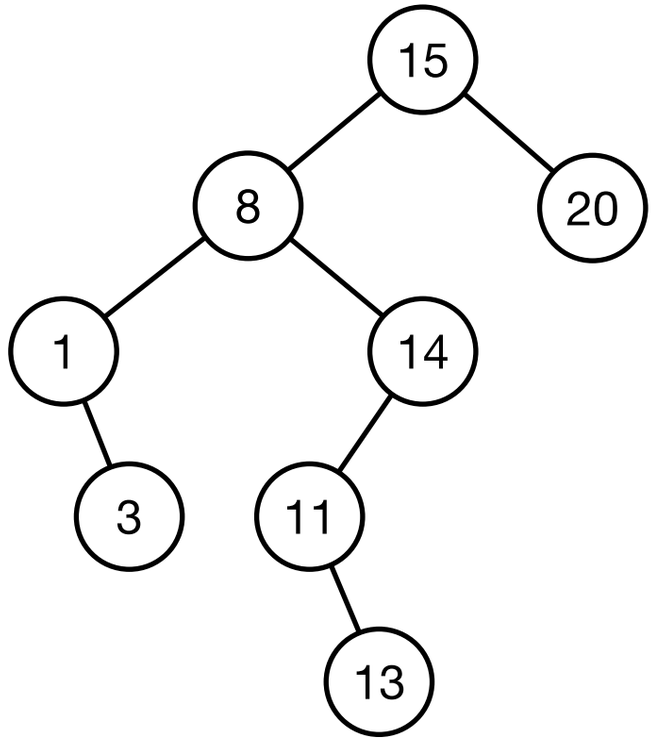
- Bisherige Algorithmen.
  - Heaps (MAX, EXTRACT-MAX, INCREASE-KEY, INSERT, …)
  - Union find (INIT, UNION, FIND, …)
  - Binäre Suchbäume (PREDECESSOR, SUCCESSOR, INSERT, DELETE, …)

- Frage. Wie entwerfen wir Algorithmen auf Binärbäumen?

Algorithmen auf Bäumen

- Divide-and-Conquer Rekursion auf Binärbäumen.
  - Löse Problem auf T(v):
    - Löse Problem rekursiv auf T(v.left) und T(v.right).
    - Kombiniere Teillösungen zu einer Lösung für T(v).



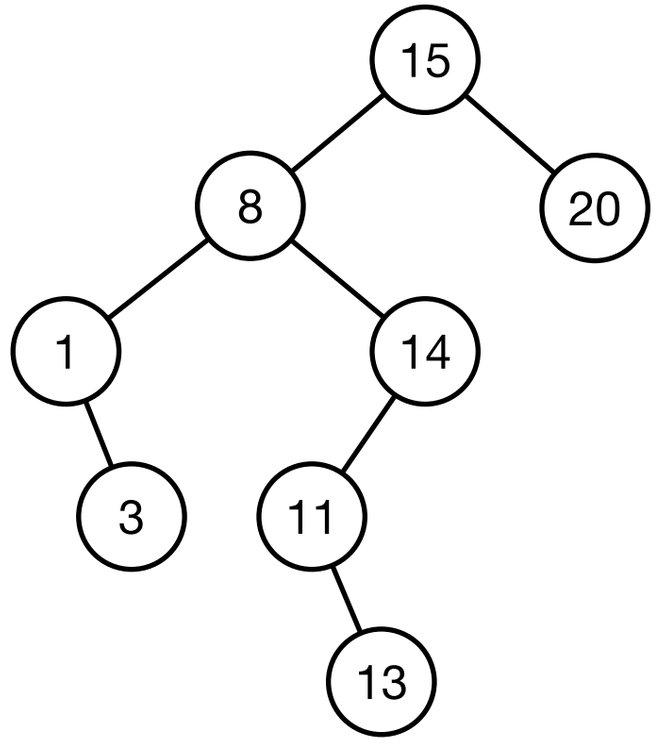


Algorithmen auf Bäumen

- Beispiel. Berechne size(v) (= Anzahl der Knoten im Baum T(v)).
  - Wenn T(v) der leere Baum ist, dann ist size(v) = 0
  - Sonst gilt size(v) = size(v.left) + size(v.right) + 1.

```
Size(v)
   if (v == null) return 0
   else return Size(v.left) + Size(v.right) + 1
```

- Zeit. O(size(v))

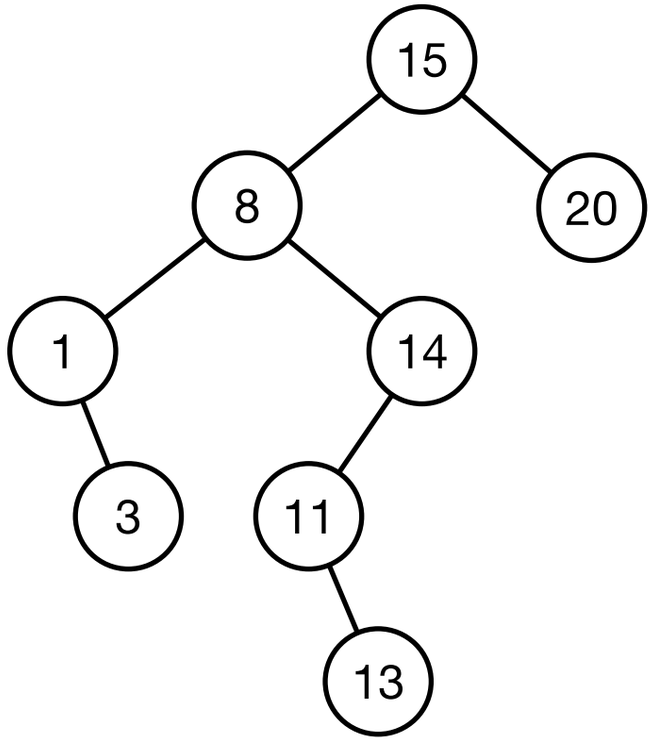


Traversierung von Bäumen

- Inorder-Traversierung.
  - Besuche linken Unterbaum rekursiv.
  - Besuche Knoten.
  - Besuche rechten Unterbaum rekursiv.
- In binären Suchbäumen druckt das alle Knoten in sortierter Reihenfolge.

- Preorder-Traversierung.
  - Besuche vertex.
  - Besuche linken Unterbaum rekursiv.
  - Besuche rechten Unterbaum rekursiv.

- Postorder-Traversierung.
  - Besuche linken Unterbaum rekursiv.
  - Besuche rechten Unterbaum rekursiv.
  - Besuche vertex.



Inorder: 1, 3, 8, 11, 13, 14, 15, 20

Preorder: 15, 8, 1, 3, 14, 11, 13, 20

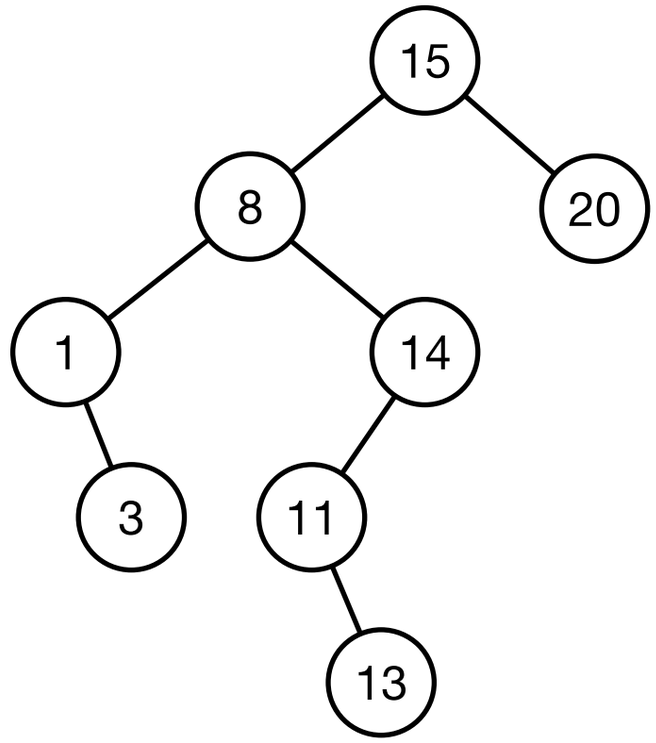
Postorder: 3, 1, 13, 11, 14, 8, 20, 15

Tree Traversals

```
Inorder(v)
    if (v == null) return
    Inorder(v.left)
    print v.key
    Inorder(v.right)
```

```
Preorder(v)
    if (v == null) return
    print v.key
    Preorder(v.left)
    Preorder(v.right)
```

```
Postorder(v)
    if (v == null) return
    Postorder(v.left)
    Postorder(v.right)
    print v.key
```



Inorder: 1, 3, 8, 11, 13, 14, 15, 20

Preorder: 15, 8, 1, 3, 14, 11, 13, 20

Postorder: 3, 1, 13, 11, 14, 8, 20, 15

- Zeit. O(n)

Binäre Suchbäume

- Nächster Nachbar
  - Binäre Suchbäume
  - Einfügen
    - Vorgänger und Nachfolger
  - Löschen
- AVL-Bäume
- Algorithmen auf Bäumen, Traversierung