Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

Übersicht

Binäre Suchbäume

- Einführung und Begriffe
- Binäre Suchbäume

Binäre Suchbäume

- Datenstruktur für dynamische Mengen
  - Unterstützen viele Operationen effizient
- Wörterbuchoperationen
  - Suchen, einfügen, löschen
  - Aufwand proportional zur Höhe des Baumes
  - Höhe zwischen $\Theta(n)$ und $\Theta(\lg n)$
- Zusätzlich
  - Prioritätswarteschlangen (Minimum, Maximum effizient abfragen)
  - Sortierte Auflistung in $\Theta(n)$
  - Vorgänger, Nachfolger effizient abfragen

Vergleich mit Hashing

- Einfügen, löschen, suchen effizienter mit Hashing
  - Hashing (Verkettung): einfügen $\Theta(1)$, löschen $\Theta(1)$, suchen (average case) $\Theta(1+\alpha)$
  - Binäre Suchbäume $\Theta(h)$, h Höhe des Baumes (kann garantieren, dass $h = O(n)$, nächstes Mal)
- Ausgabe der Schlüssel in sortierter Reihenfolge nur mit binären Suchbäumen
- Java
  - Hashing: HashMap https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
  - Binäre Suchbäume: TreeMap https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

Binäre Suchbäume

- Heute: grundlegende Operationen auf binären Suchbäumen
  - Traversierung (Aufzählung der Elemente)
  - Abfragen (suchen)
  - Einfügen
  - Löschen
- Viele weitere Varianten von Suchbäumen
  - Binäre Suchbäume, rot-schwarz Bäume, AVL Bäume, B-Bäume, etc.

    http://de.wikipedia.org/wiki/Rot-Schwarz-Baum
    http://de.wikipedia.org/wiki/AVL-Baum
    http://de.wikipedia.org/wiki/B-Baum

Begriffe

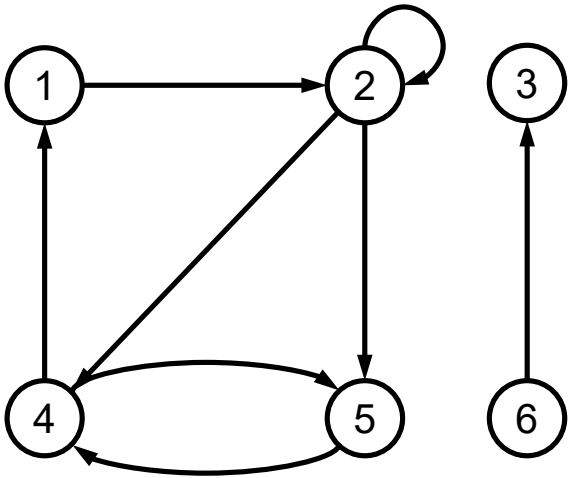
- Bäume sind spezielle Typen von Graphen
  - Siehe auch Anhänge B.4 und B.5 im Buch
  - Mehr über Graphen später
- Intuitiv
  - Graphen bestehen aus *Knoten* und *Kanten* zwischen Paaren von Knoten
  - Graphen modellieren Beziehungen zwischen Daten
  - Implementierung: Knoten als Objekte,
    Kanten als Zeiger auf Objekte
- Beispiele
  - Knoten: Schlüsselwerte
    Kanten: Kante von Knoten a nach b existiert, wenn $a.\mathrm{key} < b.\mathrm{key}$
  - Knoten: geografische Orte
    Kanten: Distanz, falls Zugstrecke existiert

Graph

- Definition (Graph)
  - Paar (V, E) von Knoten- und Kantenmenge
  - Kantenmenge E enthält Paare von Knoten
- Gerichteter Graph
  - Knotenpaare geordnet
  - Knotenpaare (a, b) und (b, a) werden unterschieden
- Ungerichteter Graph
  - Knotenpaare ungeordnet

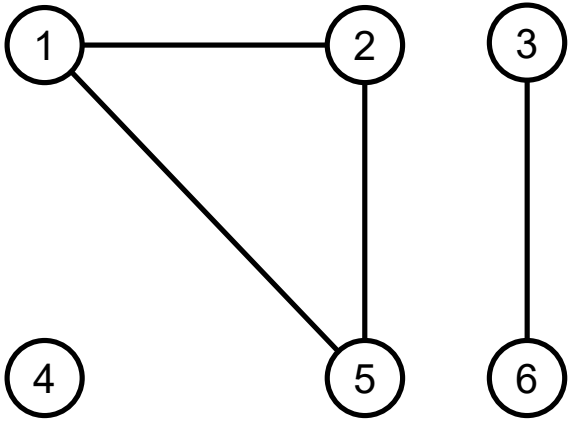
Beispiel: gerichteter Graph

- Graph $G = (V, E)$
- Knotenmenge $V = \{1,2,3,4,5,6\}$
- Kantenmenge $E =$
  $\{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$



Beispiel: ungerichteter Graph

- Graph $G = (V, E)$
- Knotenmenge $V = \{1,2,3,4,5,6\}$
- Kantenmenge $E = \{(1,2), (2,3), (1,5), (3,6)\}$

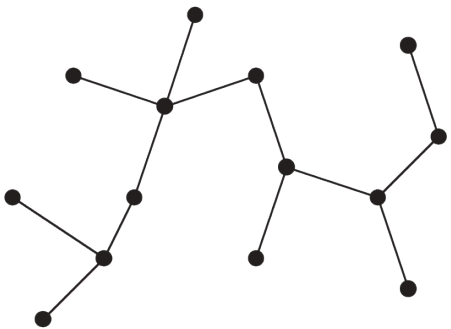


Begriffe

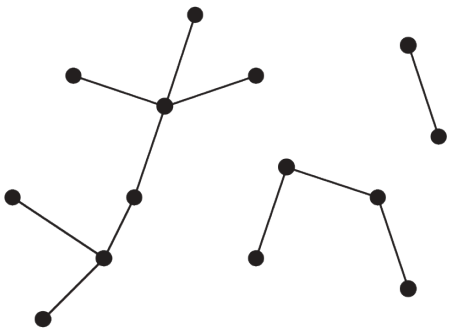
- Pfad: Sequenz von Knoten die über Kanten entsprechender Richtung verbunden sind
- Erreichbar: Knoten ist von einem anderen erreichbar, wenn es einen Pfad gibt, der beide Knoten verbindet
- Zusammenhängend (ungerichteter Graph): jedes Knotenpaar ist durch einen Pfad verbunden
- Zyklus: Pfad, der als ersten und letzten Knoten denselben Knoten enthält

Begriffe

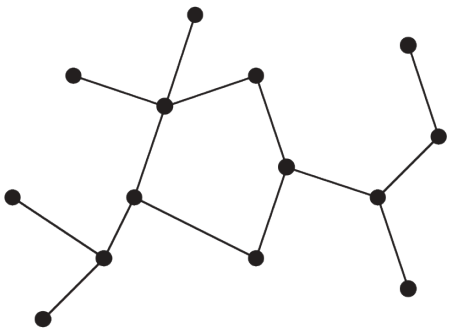
- Freier Baum: zusammenhängender, azyklischer, ungerichteter Graph
- Wald: azyklischer, ungerichteter Graph, nicht unbeding zusammenhängend



Freier Baum



Wald

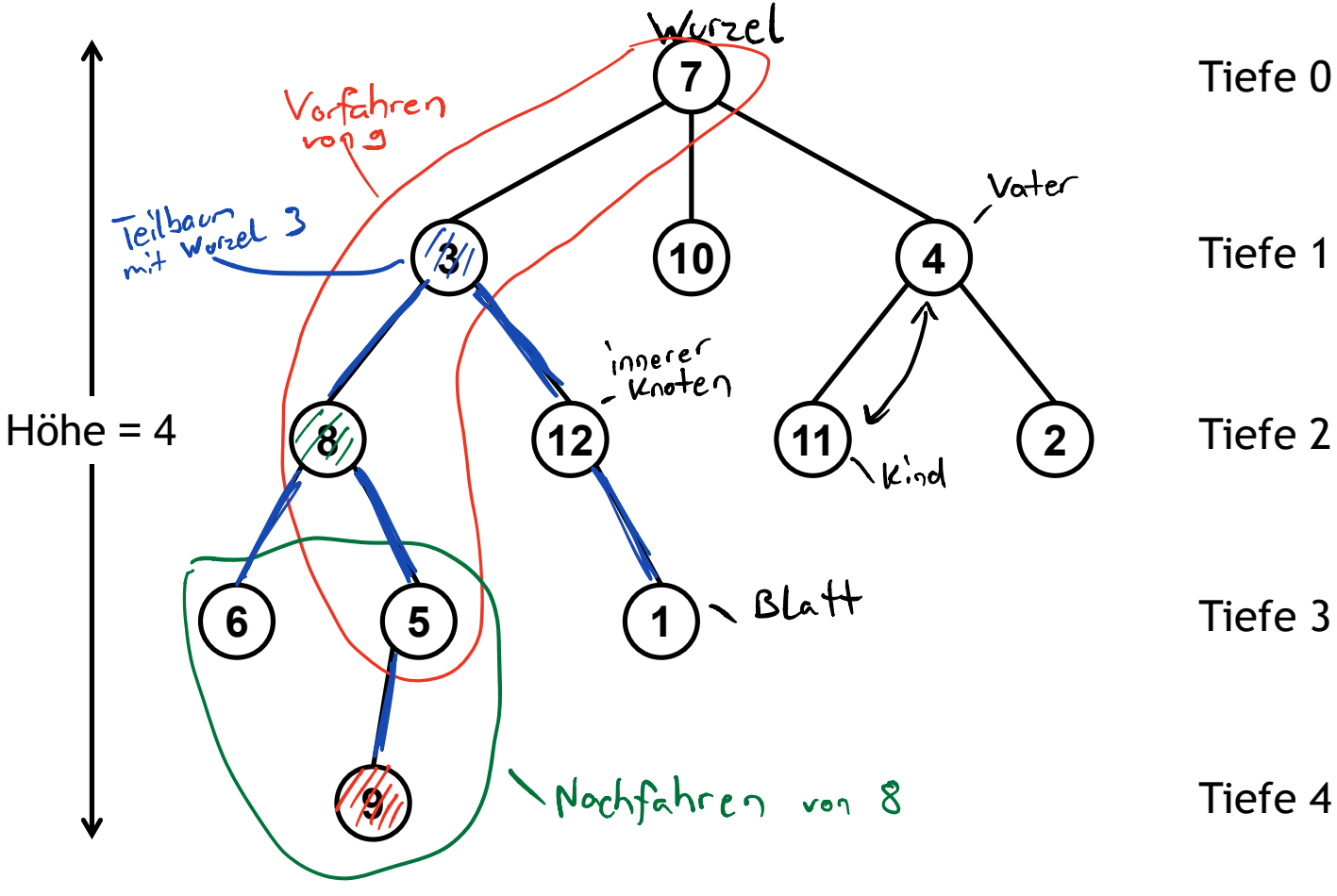


Zyklischer Graph
(weder Baum noch Wald)

Begriffe

- (Gerichteter) Baum: freier Baum mit ausgezeichnetem Knoten, Wurzel
- Innerer Knoten, Blatt
- Vater, Kind
- Vorfahre, Nachfahre
- Teilbaum
- Tiefe eines Knotens: Abstand zur Wurzel
- Höhe eines Knotens: # Kanten des längsten, einfachen Pfades zu einem Blatt
- Höhe des Baumes: # Kanten des längsten, einfachen Pfades von Wurzel zu einem Blatt

Beispiel


Wurzel
7
Vorfahren von 9
Teilbaum mit Wurzel 3
3
10
4
Vater
innerer Knoten
8
12
11
2
Kind
6
5
1
Blatt
9
Nachfahren von 8
Höhe = 4
Tiefe 0
Tiefe 1
Tiefe 2
Tiefe 3
Tiefe 4

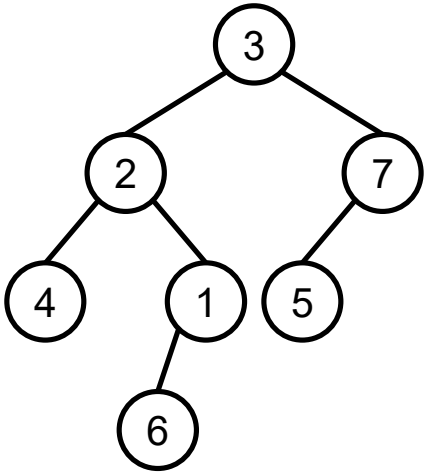
Begriffe

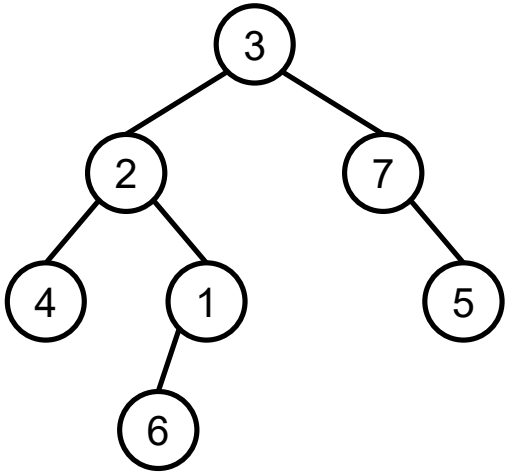
Binärer Baum: rekursive Definition

- Leerer Baum, oder
- Baum bestehend aus
  - Wurzel
  - Linker Teilbaum, der binärer Baum ist
  - Rechter Teilbaum, der binärer Baum ist

Beispiel

- Zwei binäre Bäume hier nicht identisch
- Linker und rechter Teilbaum werden unterschieden



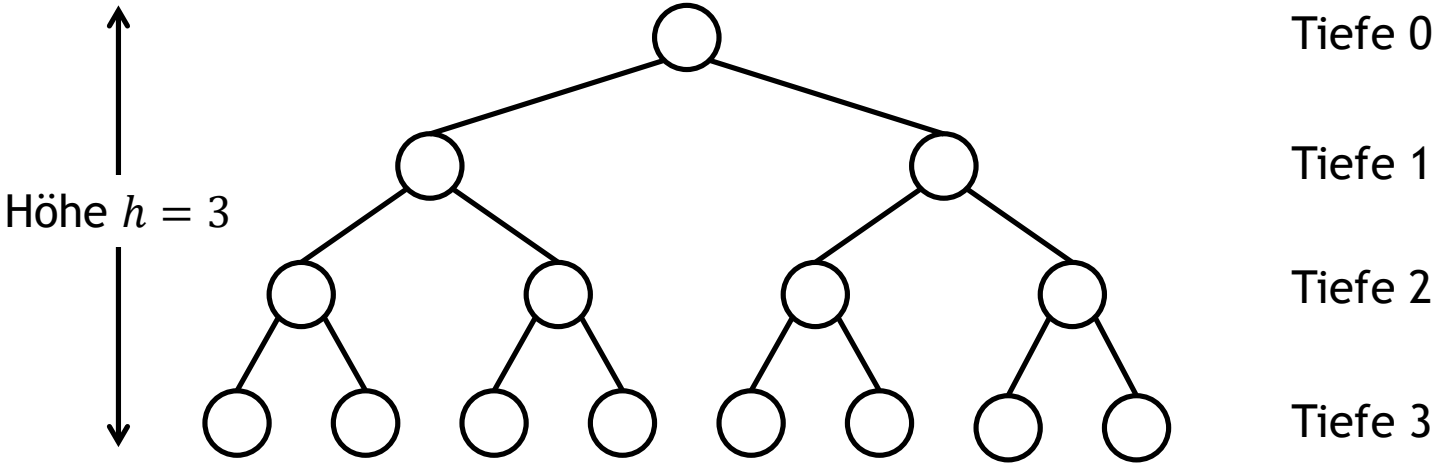


Begriffe

- Vollständiger binärer Baum
  - Alle Blätter haben gleiche Tiefe
  - Alle internen Knoten haben zwei nicht-leere Teilbäume
- Vollständiger binärer Baum mit Höhe h hat
  - $n = 2^h$ Blätter
  - Höhe ist $h = \log_2 n$
  - $2^h - 1$ interne Knoten

Beispiel

- Vollständiger binärer Baum der Höhe 3, 8 Blätter und 7 interne Knoten hat



Binäre Bäume: Implementation

- Verkettete Struktur
- Blätter haben `left = right = nil`

```
class BinaryNode {
  BinaryNode left, right, p;
  int key;
}
```

Übersicht

Binäre Suchbäume

- Einführung und Begriffe
- Binäre Suchbäume

Binäre Suchbaum-Eigenschaft

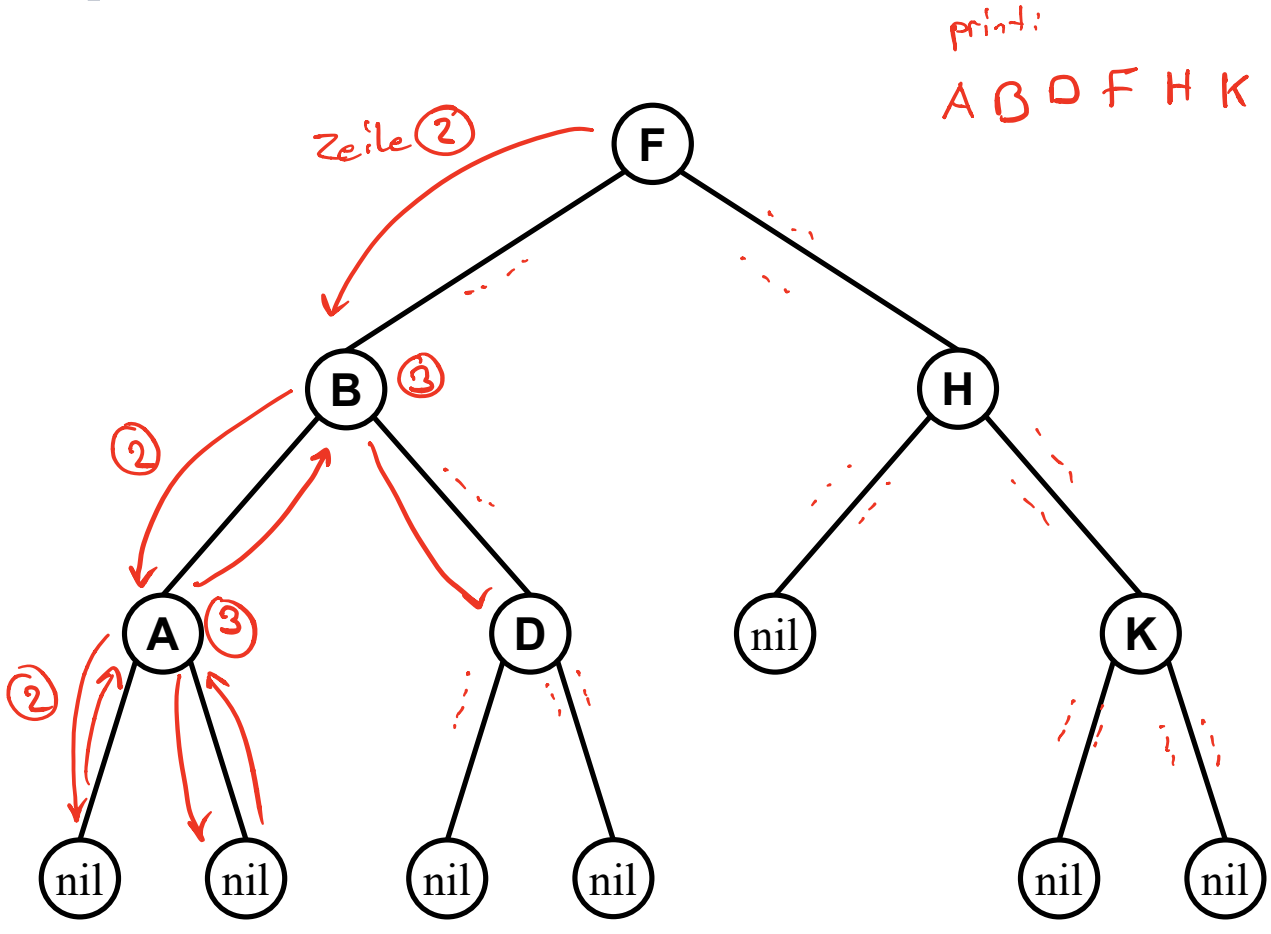
- Falls Knoten y im linken Teilbaum von Knoten x, dann $\boldsymbol{y.\mathrm{key} \leq x.\mathrm{key}}$
- Falls Knoten y im rechten Teilbaum von Knoten x, dann $\boldsymbol{y.\mathrm{key} \geq x.\mathrm{key}}$

Inorder Traversierung

- Besucht Knoten in aufsteigender Reihenfolge

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2        INORDER-TREE-WALK(x.left)
3        print x.key
4        INORDER-TREE-WALK(x.right)
```

Beispiel



Inorder Traversierung

- Korrektheit: folgt mittels Rekursion aus Suchbaum-Eigenschaft
- Zeitaufwand: Linear, jeder Knoten wird genau einmal besucht und ausgegeben

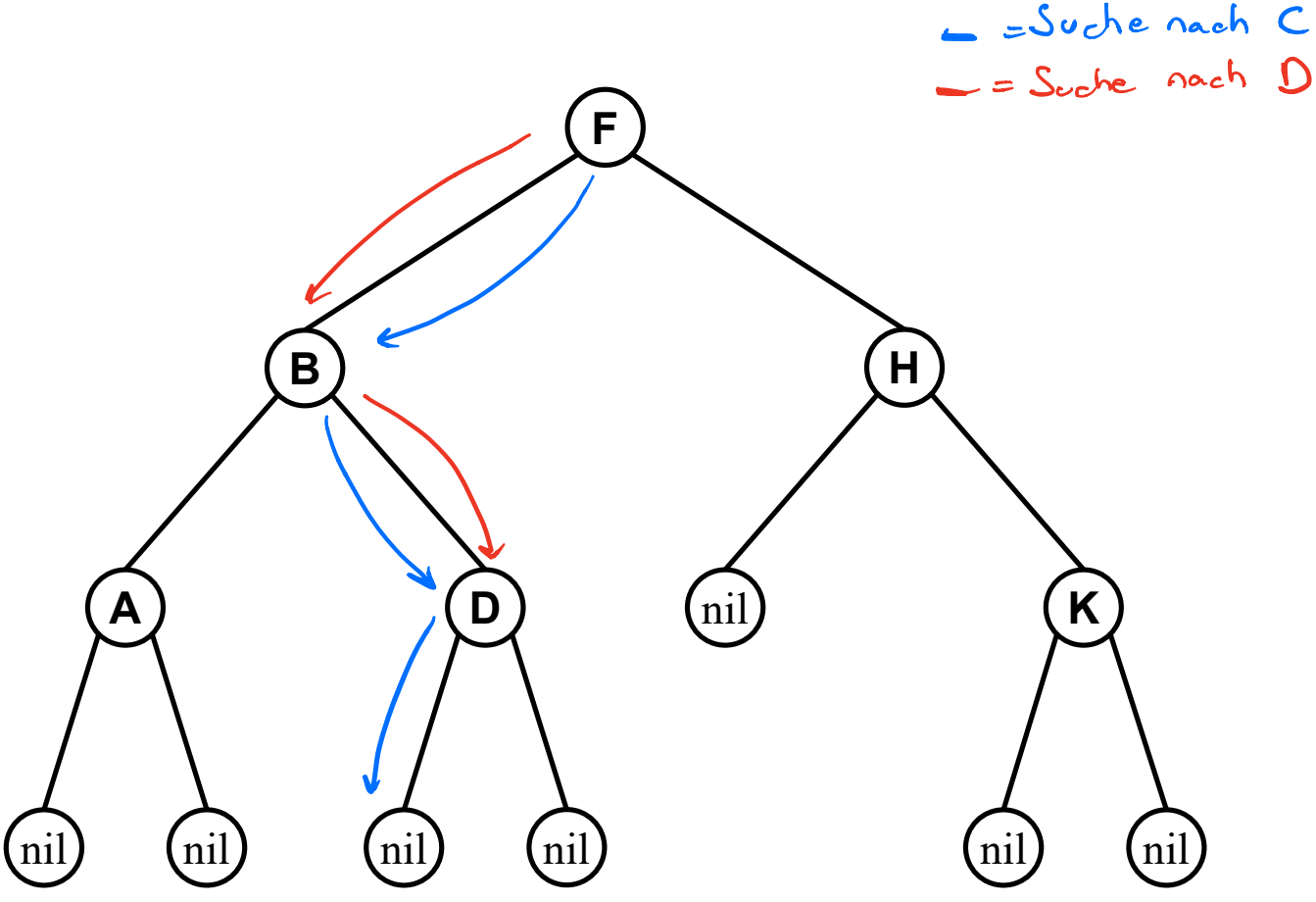
Alternative Reihenfolgen

- Inorder-Traversierung: traversiere linken Teilbaum, gib Knoten aus, traversiere rechten Teilbaum
- Preorder-Traversierung: gibt Knoten aus, traversiere linken Teilbaum, traversiere rechten Teilbaum
- Postorder-Traversierung: traversiere linken Teilbaum, traversiere rechten Teilbaum, gib Knoten aus
- Nur Inorder-Traversierung gibt Schlüssel sortiert aus

Suchen

```
TREE-SEARCH(x, k)
1   if x == NIL or k == x.key
2         return x
3   if k < x.key
4         return TREE-SEARCH(x.left, k)
5   else
6         return TREE-SEARCH(x.right, k)
```

Beispiel



Suchen

- Überprüfte Knoten bilden Pfad von Wurzel nach unten
- Laufzeit in $O(h)$, h Höhe des Baumes
- Iterative Variante, ähnlich wie Traversierung einer Liste

ITERATIVE-TREE-SEARCH(x, k)

```
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else
5              x = x.right
6  return x
```

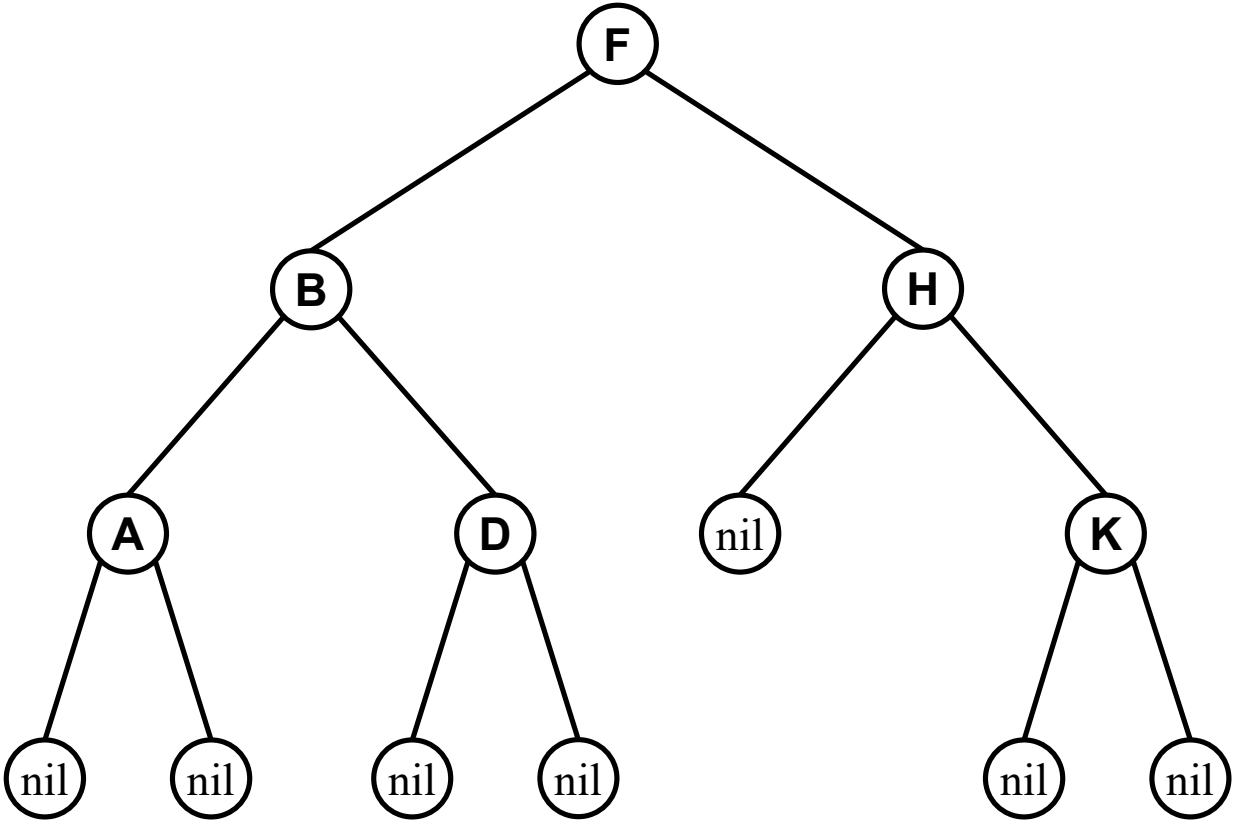
Minimum

- Aus Suchbaum-Eigenschaft folgt
  - Kleinster Knoten ist Knoten am weitesten links
  - Grösster Knoten ist Knoten am weitesten rechts

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Beispiel



Vorgänger und Nachfolger

- (Nicht dasselbe wie Vorfahre, Nachfahre)
- Annahme: Schlüssel paarweise verschieden
- Vorgänger von Element x: Element mit grösstem Schlüssel, der kleiner ist als x
- Nachfolger von Element x: Element mit kleinstem Schlüssel, der grösser ist als x
- Kann gefunden werden ohne Schlüssel-vergleiche!

Nachfolger

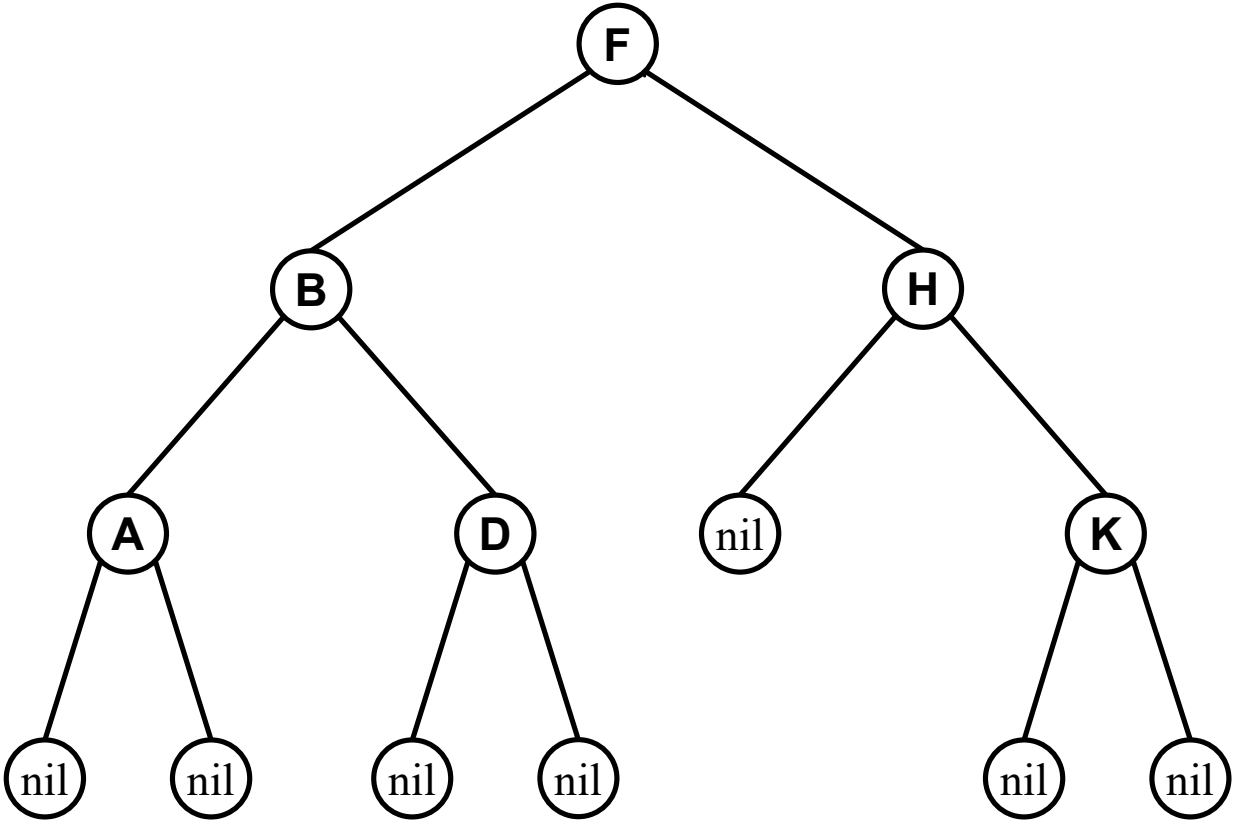
Nachfolger bestimmen: Knoten suchen, der bei Inorder Traversierung als nächster ausgegeben würde.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x ≠ y.left
5        x = y
6        y = y.p
7  return y
```

Nachfolger ist kleinster Schlüssel im rechten Teilbaum

Kein rechter Teilbaum, traversiere Baum nach oben bis ein Kind (x) das linke Kind des Vaters (y) ist. Der Vater (y) ist der Nachfolger.
→ Inorder Traversierung

Beispiel



Zusammenfassung

- Operationen Suchen, Minimum, Maximum, Vorgänger und Nachfolger können auf Binärbaum der Höhe h in $O(h)$ ausgeführt werden
- Grund: Alle Operationen folgen entweder einem strikt aufwärts oder abwärts laufenden Pfad im Baum

Einfügen und Löschen

- Einfügen und Löschen ändern die Struktur des Suchbaumes
- Müssen sicherstellen, das Suchbaum Eigenschaft wieder hergestellt wird
- Einfügen einfacher als Löschen

Einfügen

Eig eine Erfolglose Suche

TREE-INSERT(T,z)

```
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4         y = x
5         if z.key < x.key
6               x = x.left
7         else
8               x = x.right
9   z.p = y
10  if y == NIL
11        T.root = z
12  elseif z.key < y.key
13        y.left = z
14  else
15        y.right = z
```

Starte bei Wurzel x, traversiere Baum nach unten. y ist immer Vater von x.

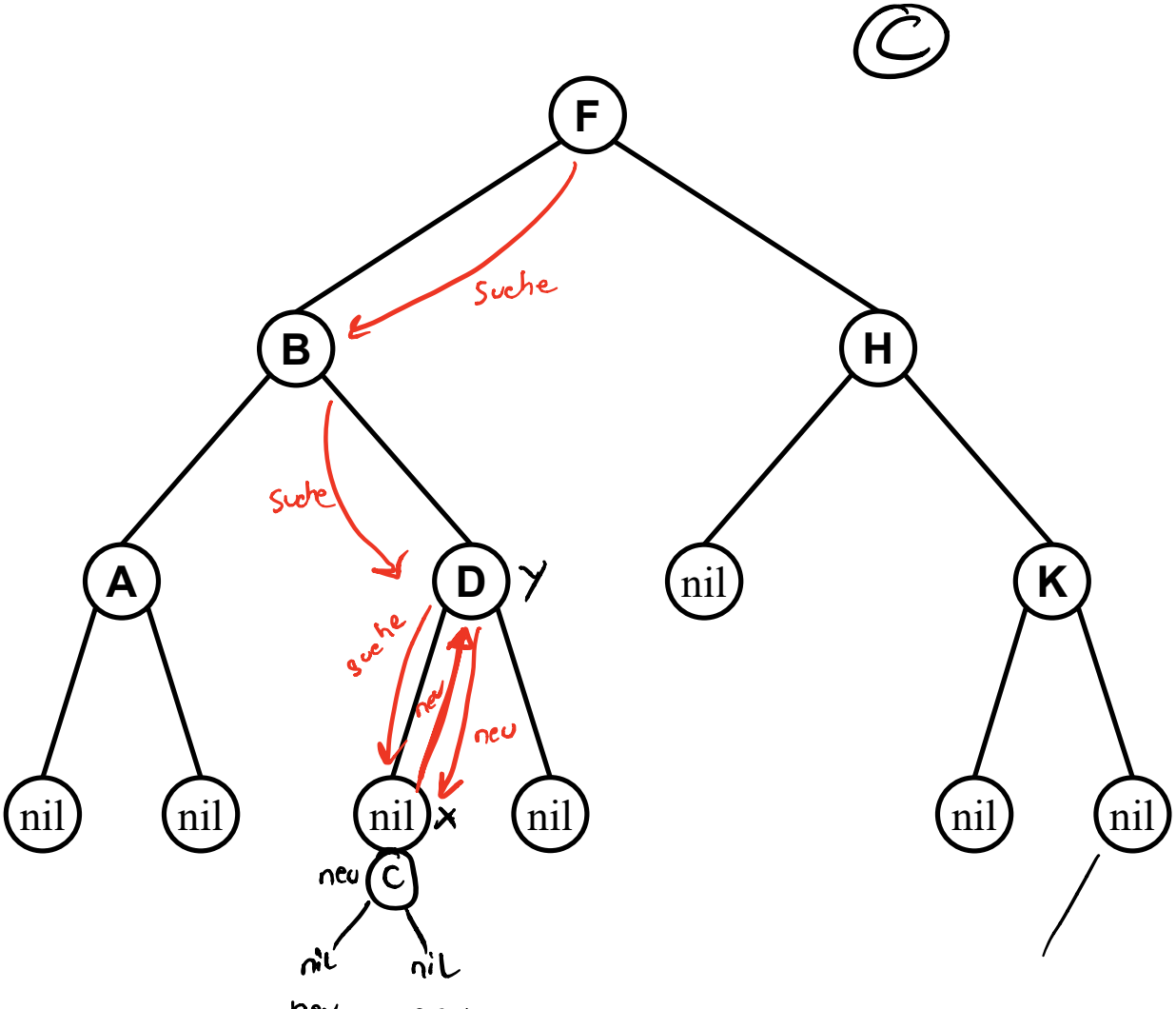
x traversiert Baum nach unten, wie beim Suchen nach $z.key$. Am Schluss der Schleife immer $y = x.p$

y wird Vater des eingefügten Elements.

Baum war leer, z wird Wurzel.

z wird je nach Schlüssel linkes oder rechtes Kind von y.

Beispiel


F
B
H
A
D
nil
K
nil
nil
nil
nil
nil
nil
C
nil
nil
Suche
Suche
suche
neu
neu
neu
y
x

Einfügen

- Laufzeit $O(h)$, wie Suchen
- Einfügen und Inorder-Traversierung können als Sortieralgorithmus verwendet werden

Löschen

3 Fälle: Lösche Knoten z

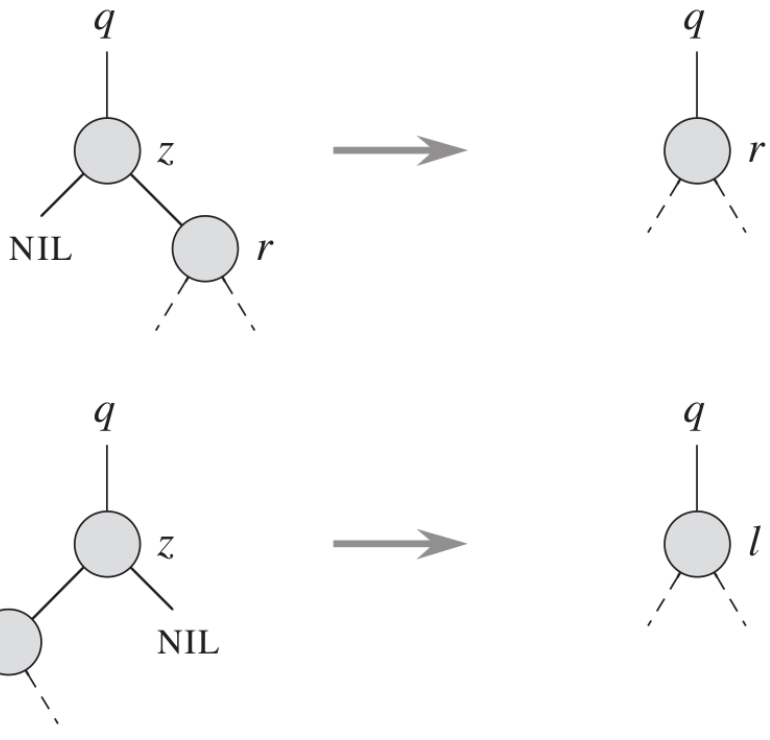
1. Fall: Knoten z hat keine Kinder

- Lösche Knoten z indem bei Vater von z Zeiger auf z durch Wert NIL ersetzt wird

Löschen

2. Fall: Knoten z hat ein Kind

Vater von Knoten z zeigt auf Kind von z anstatt auf z



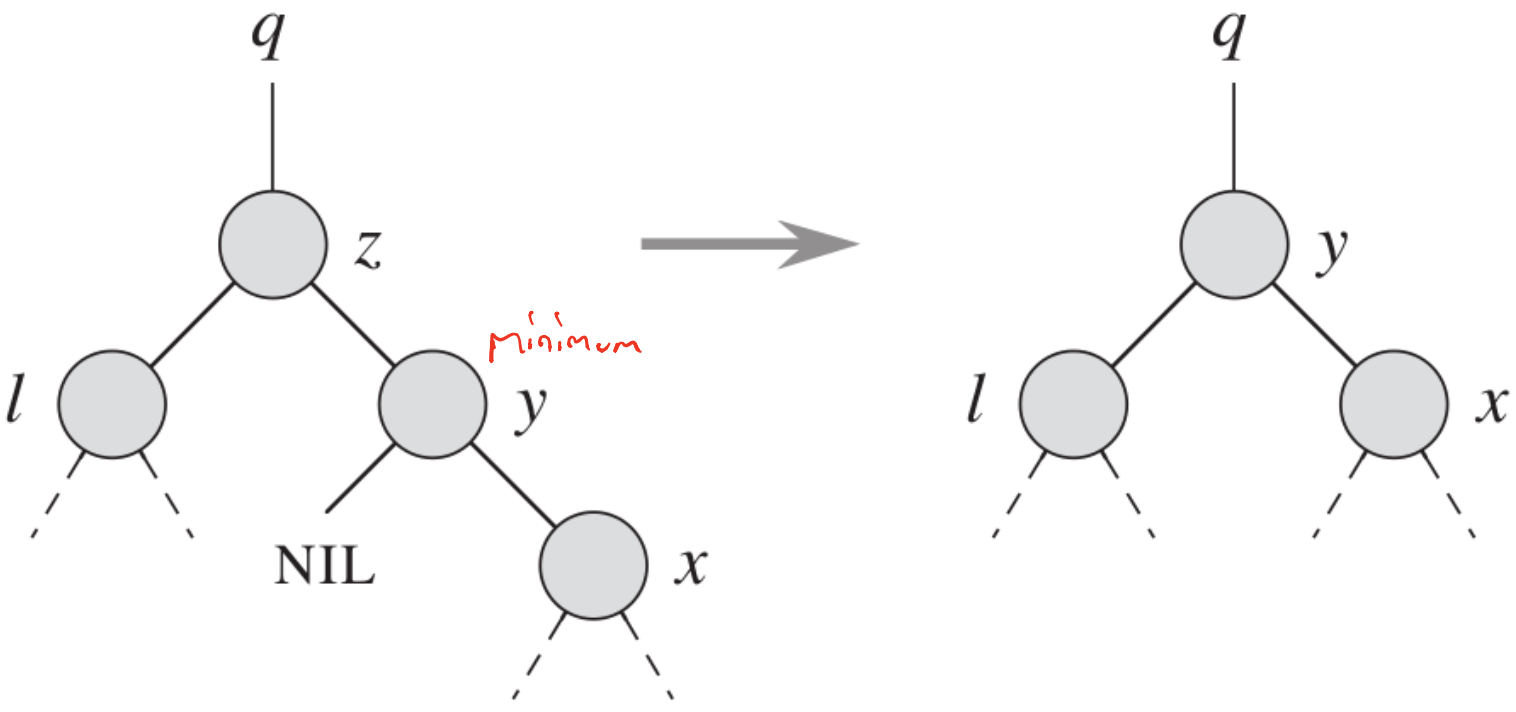
Löschen

3. Fall: Knoten z hat zwei Kinder

- Finde Nachfolger y von z. Es gilt: y hat entweder kein oder nur ein rechtes Kind (Nachfolger von z bedeutet der "linkste" Knoten im Rechten Teilbaum von z) (Minimum)
- Ersetze z mit Nachfolger y → zwei Fälle

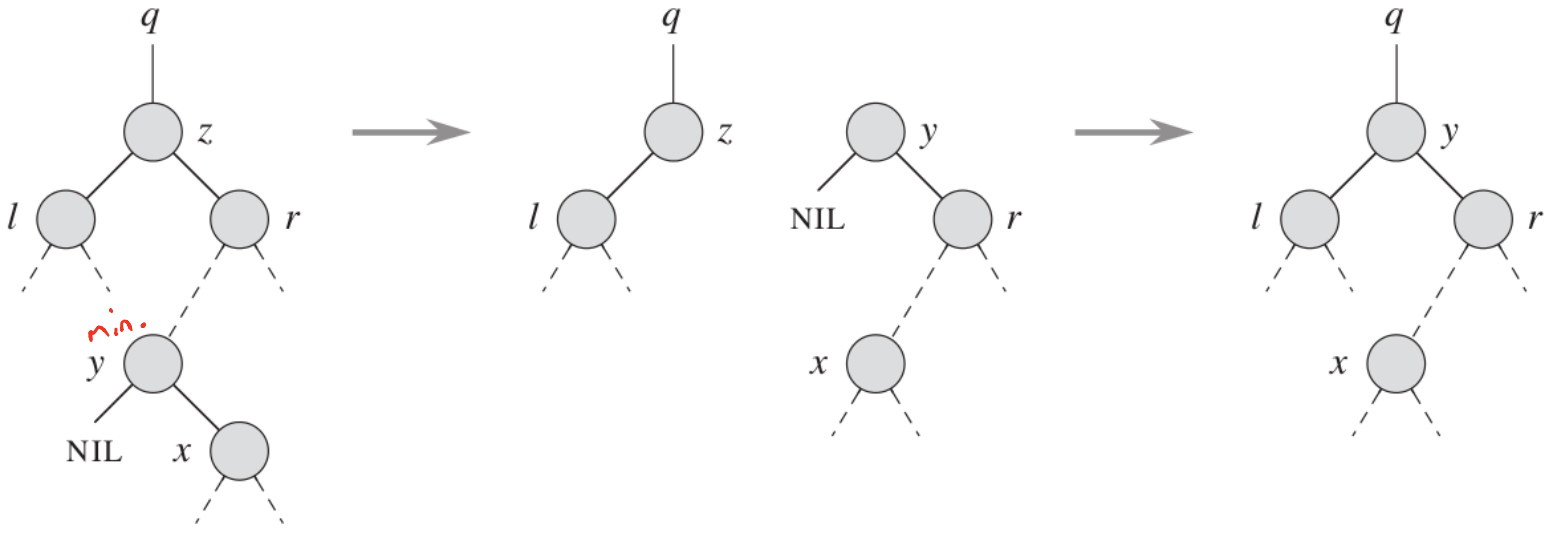
Löschen

Fall 3a, Nachfolger ist rechtes Kind von z



Löschen

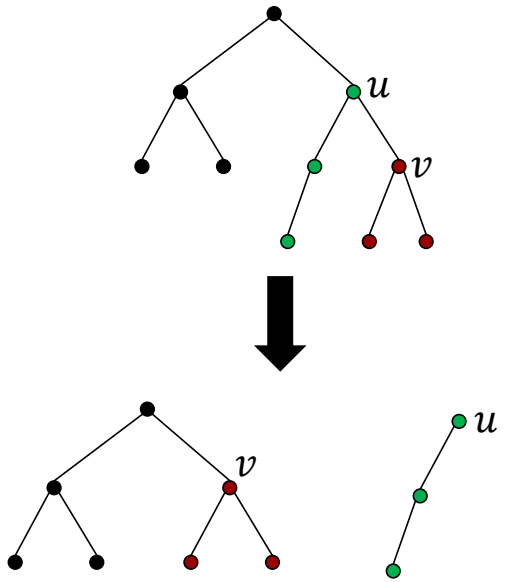
Fall 3b, Nachfolger ist nicht rechtes Kind von z



Löschen

- Hilfsfunktion TRANSPLANT
  Ersetzt Unterbaum mit Wurzel u
  durch Unterbaum mit Wurzel v

```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else
6       u.p.right = v
7   if v ≠ NIL
8       v.p = u.p
```



Löschen

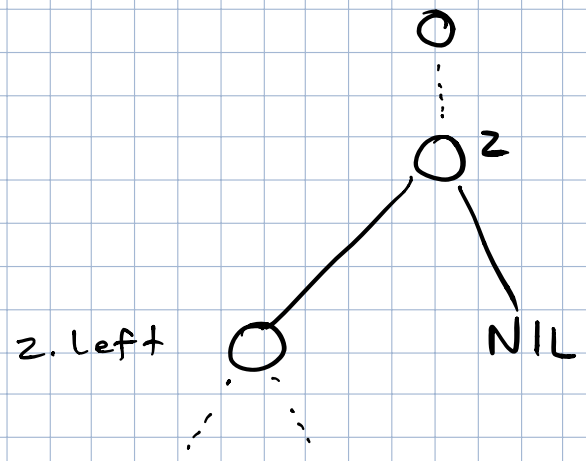
TREE-DELETE(T, z)

```
 1  if z.left == NIL
 2        TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSPLANT(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSPLANT(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

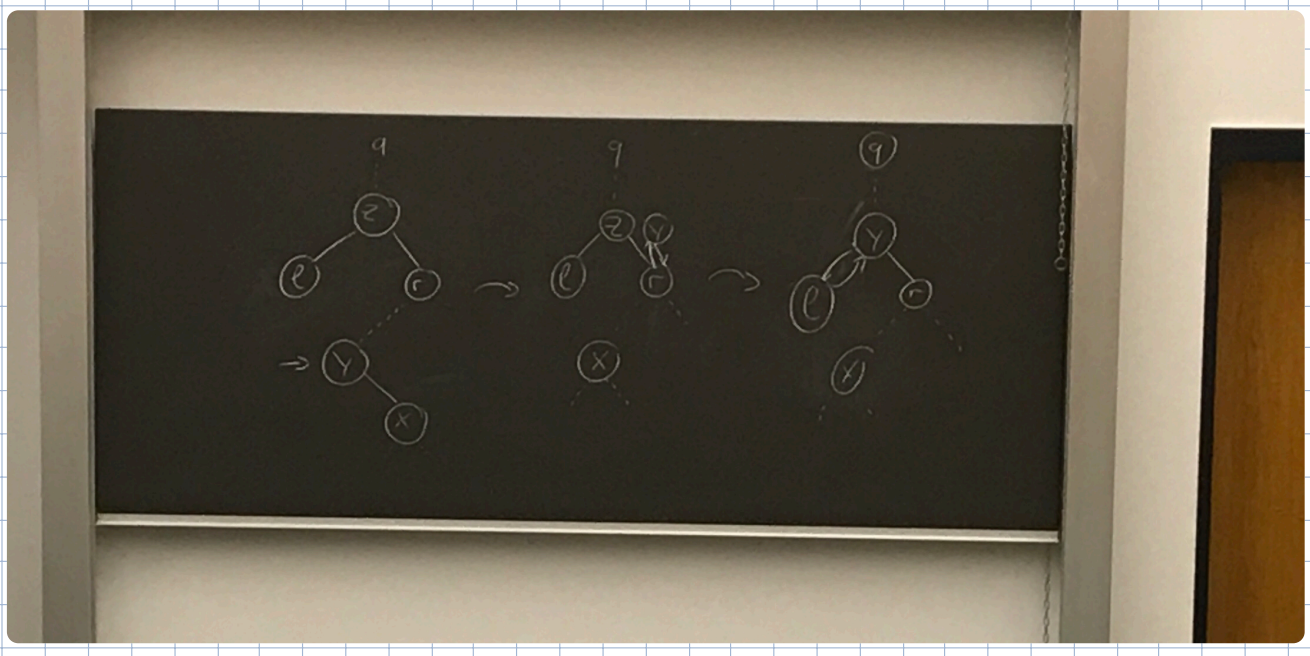
Fall 1 & 2

Fall 3

Fall 3b



Fall 2



Zusammenfassung

- Alle Operationen laufen mit Aufwand $\Theta(h)$
- Problem
  - Im Worst Case degeneriert Baum zu einer Liste
  - Z.B. unglückliche Reihenfolge beim Einfügen
  - Höhe ist von Ordnung $h = O(n)$
- Lösung
  - Garantiere kleine Höhe
  - Balancierte Bäume, so dass immer $h = \Theta(\lg n)$
- Ansatz
  - Restrukturiere Baum falls nötig nach Einfügen und Löschen
  - Verschiedene Strategien,
    nächste Vorlesung: rot-schwarz Bäume

Nächstes Mal

- Kapitel 13: Rot-schwarz Bäume