Teil XI

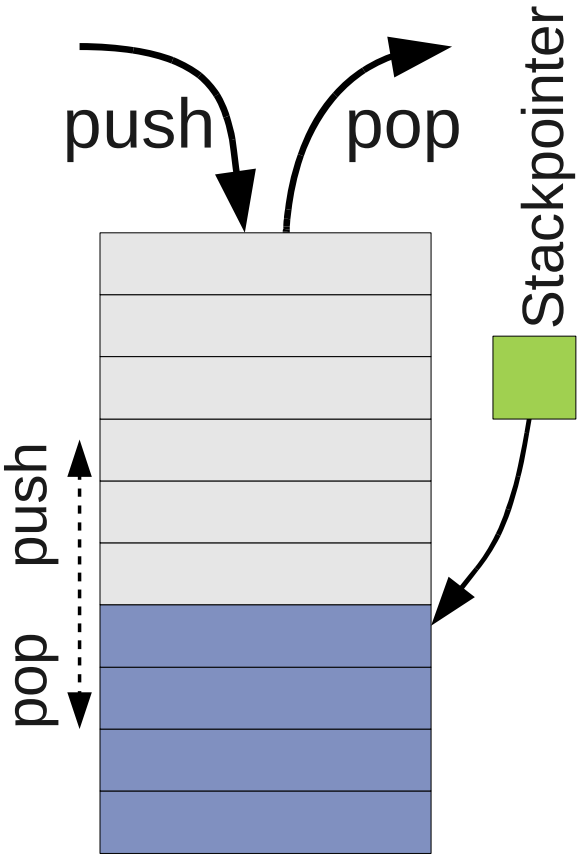
Datenstrukturen: Bäume, Stacks und Queues

Stacks (Kellerspeicher/Stapel)

- Funktioniert wie ein natürlicher Stapel (z.B. Papierstapel auf dem Schreibtisch)
- Elemente werden immer oben auf den Stapel gelegt (PUSH)
- Es kann immer nur das oberste Element entfernt werden (POP)
- Funktionsweise: LIFO (Last In, First Out)

Stacks (Kellerspeicher/Stapel)

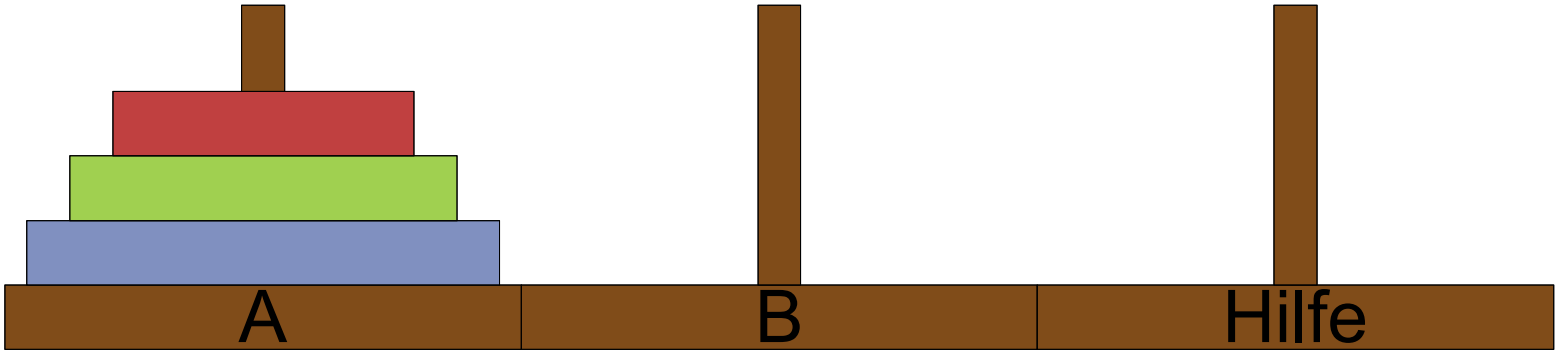
- Funktioniert wie ein natürlicher Stapel (z.B. Papierstapel auf dem Schreibtisch)
- Elemente werden immer oben auf den Stapel gelegt (PUSH)
- Es kann immer nur das oberste Element entfernt werden (POP)
- Funktionsweise: LIFO (Last In, First Out)



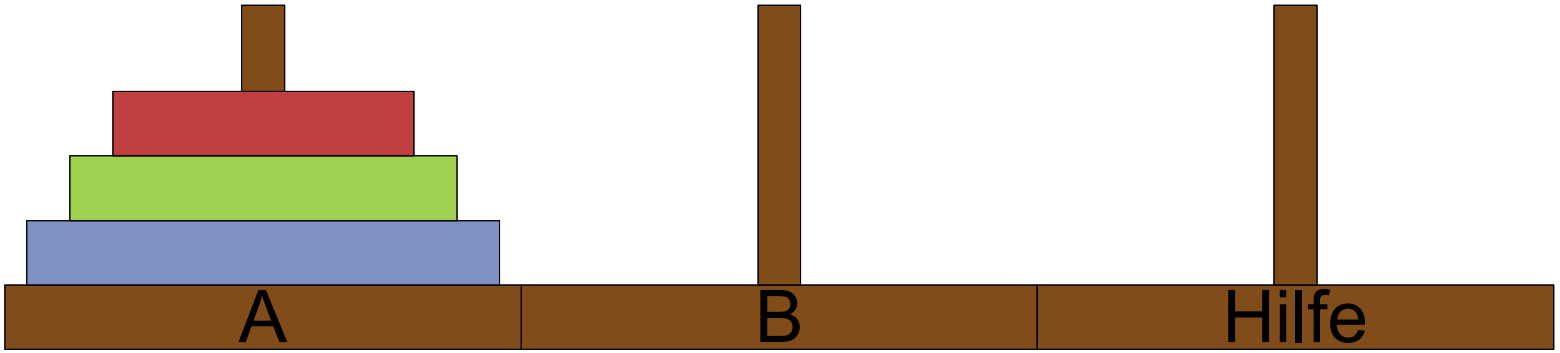
Stacks in Python

- Es gibt keine eigene Stack-Datenstruktur in Python
- Listen können aber als (effiziente) Stacks verwendet werden
- PUSH: `list.append(x)`
- POP: `x = list.pop()`

Stack-Beispiel: Türme von Hanoi



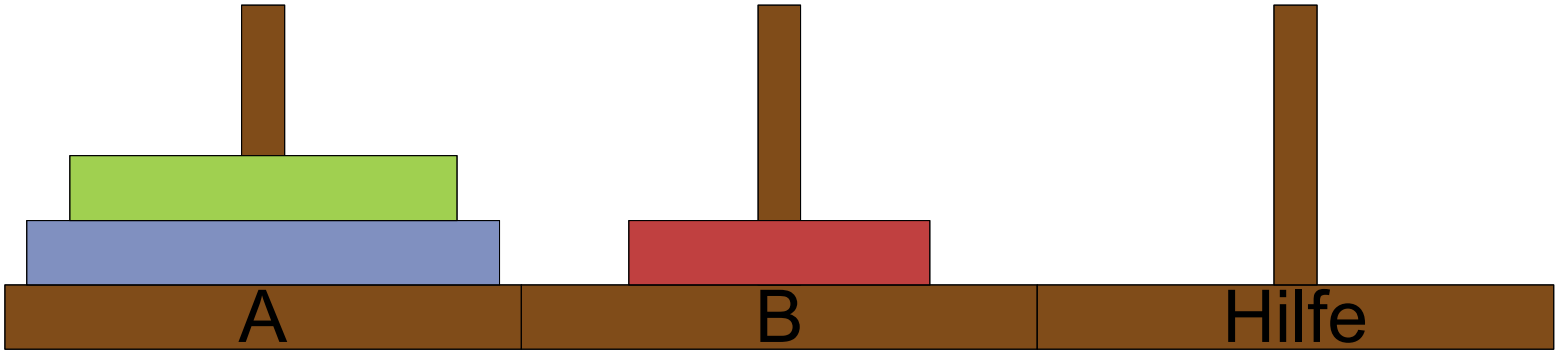
Stack-Beispiel: Türme von Hanoi




```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    printhanoi()
    hanoi(n-1, hilfe, b, a)
```

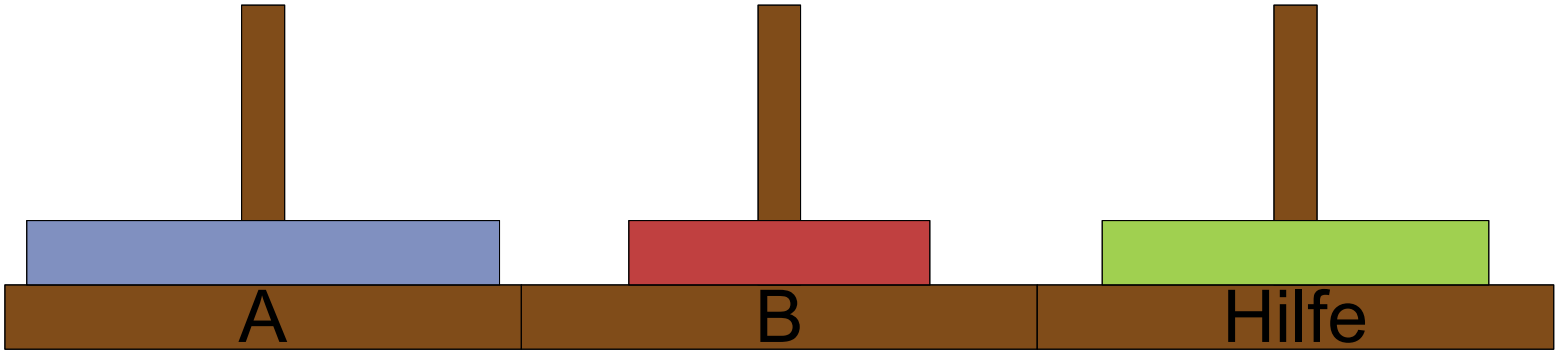
rekursiver Algorithmus:

- n=# zu versch. Scheiben
- Scheiben: a=woher, b=wohin, hilfe=Zwischenziel



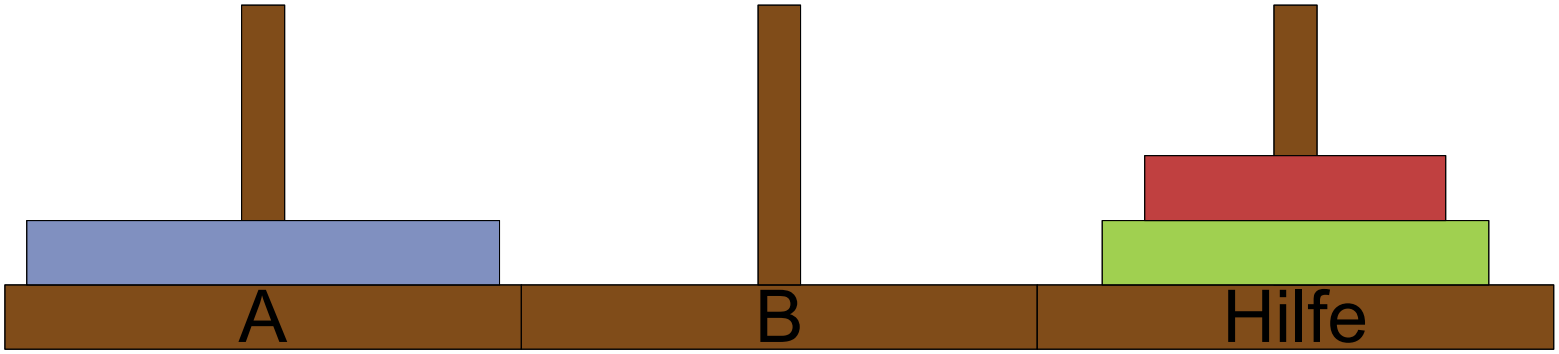

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```



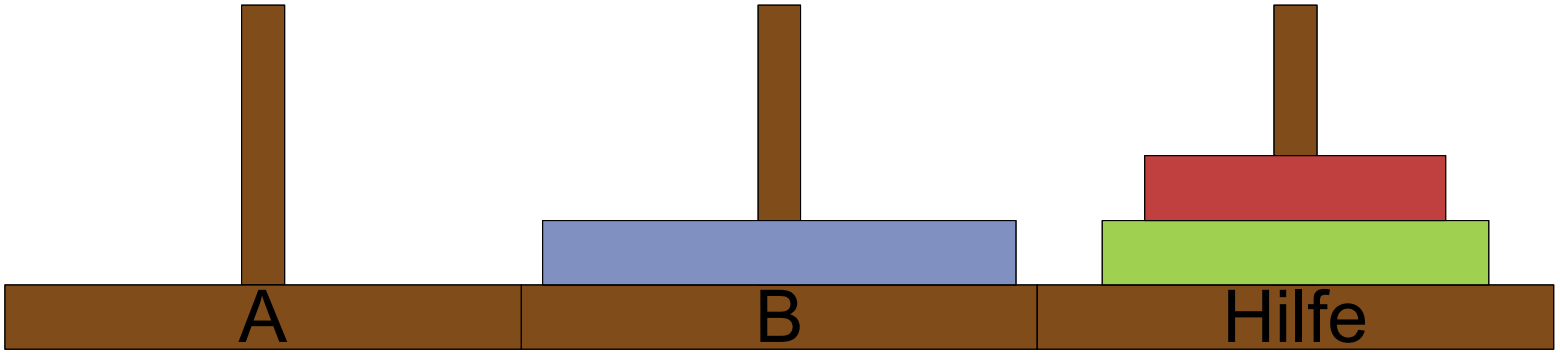

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```



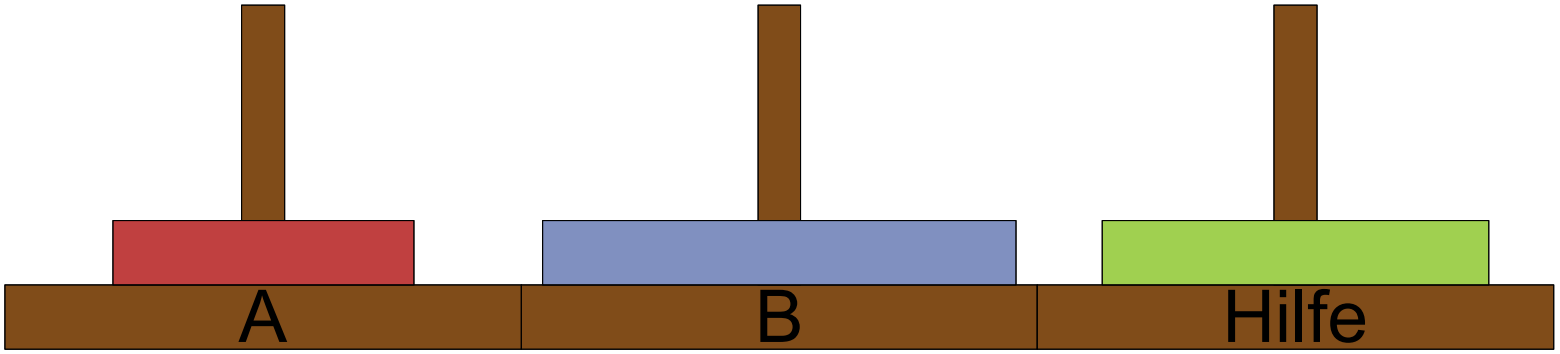

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```



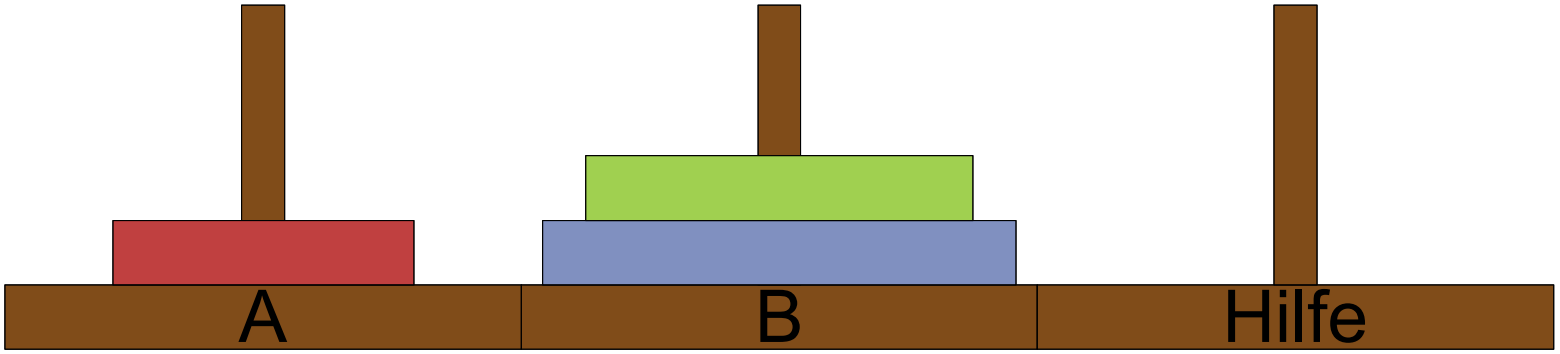

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
  A -> B
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```



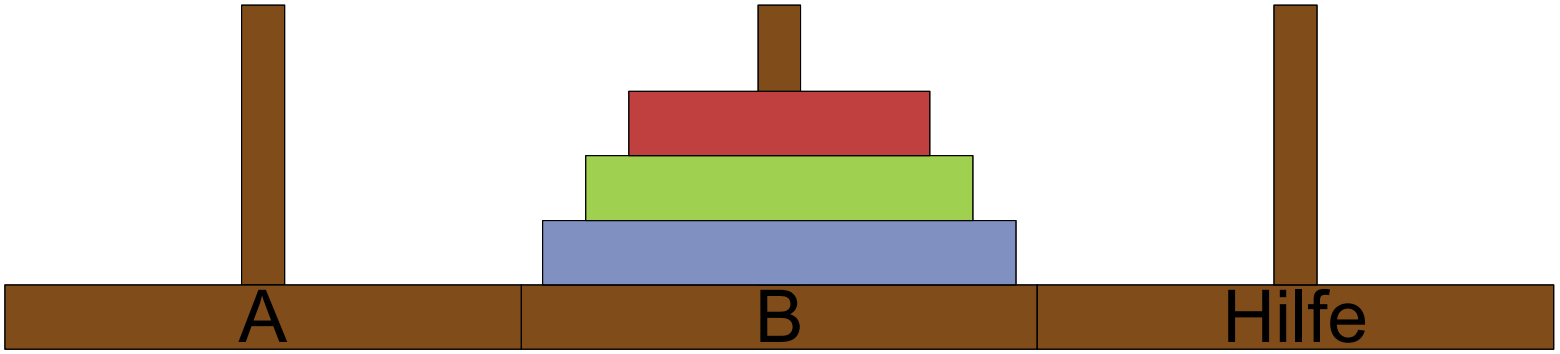

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
  A -> B
  hanoi(2,H,B,A)
    hanoi(1,H,A,B)
    H -> A
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```



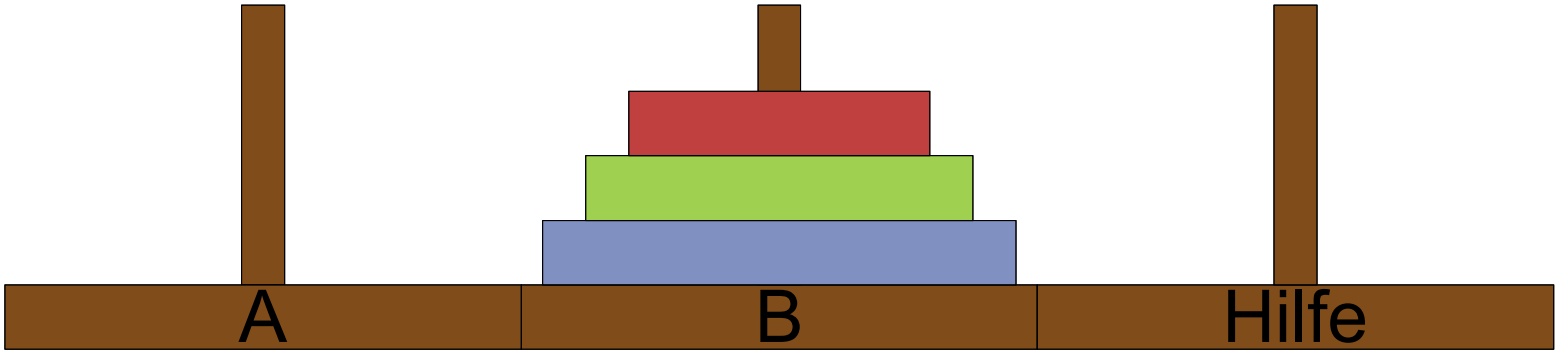

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
  A -> B
  hanoi(2,H,B,A)
    hanoi(1,H,A,B)
    H -> A
  H -> B
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```




```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
  A -> B
  hanoi(2,H,B,A)
    hanoi(1,H,A,B)
      H -> A
  H -> B
  hanoi(1,A,B,H)
    A -> B
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```




```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
  A -> B
  hanoi(2,H,B,A)
    hanoi(1,H,A,B)
      H -> A
  H -> B
  hanoi(1,A,B,H)
    A -> B
```

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```

Optimale Zugfolge: $2^n - 1$ Züge

- $\Rightarrow$ 34 Jahre bei n=30 und 1 Scheibe/s
- $\Rightarrow$ 585 Mrd. Jahre bei n=64

Queues/Warteschlangen

- Funktioniert wie eine natürlicher Warteschlange (z.B. Supermarkt, Postamt, Druckaufträge)
- Elemente werden immer ans Ende der Queue angefügt (PUSH)
- Es wird immer nur vom Anfang der Queue entfernt (POP)
- Funktionsweise: FIFO (First In, First Out)

Queues/Warteschlangen

- Funktioniert wie eine natürlicher Warteschlange (z.B. Supermarkt, Postamt, Druckaufträge)
- Elemente werden immer ans Ende der Queue angefügt (PUSH)
- Es wird immer nur vom Anfang der Queue entfernt (POP)
- Funktionsweise: FIFO (First In, First Out)

Queues in Python

- Listen können nicht als Queues verwendet werden, da der Aufwand für das Entfernen des ersten Elements O(N) ist
- Das Module `Queue` stellt Queues zur Verfügung

```
>>> from Queue import Queue
>>> q = Queue()
>>> q.put(1); q.put(2); q.put(3)
>>> q.get()
1
```

Beispiel: Bürokrat

Hands-On: Programmieren Sie einen Bürostuhl-Akrobaten `Buerokrat`!

- Die Klasse `Buerokrat` besitzt eine Methode `abgelehnt`, welche als Argument lediglich einen Stapel an Formularen (strings) erhält.
- Jedes Formular wird “bearbeitet”:
  - Das Formular wird vom Stapel genommen.
  - Das Formular wird abgelehnt: Geben Sie hierzu am Bildschirm “Formular NAME: Abgelehnt” aus. NAME steht für das entsprechende Formular.
  - Das Formular wird auf einen neuen Stapel gepackt.
- Der Stapel bearbeiteter Formulare soll zurückgegeben werden.
- Testen Sie Ihr Programm mit dem Stapel:

```
["Passierschein␣A-39", "Rundschreiben␣B-65", "Passierschein␣A-38"]
```

Beispiel: Bürokrat

```
class Buerokrat:
  def abgelehnt(self,formulare):
    fertig = []
    while len(formulare)>0:
      f = formulare.pop()
      print "Formular " + f + ": abgelehnt"
      fertig.append(f)
    return fertig

if __name__=="__main__":
  b = Buerokrat()
  formulare = ["Passierschein A-39", "Rundschreiben B-65", \
               "Passierschein A-38"]
  print "Arbeitsanfang:"
  print formulare
  fertig = b.abgelehnt(formulare)
  print "Arbeitsende:"
  print fertig
```

Graphen

- Wir ersparen uns eine formale Definition!
- Ein Graph besteht aus einer Menge Knoten V
- Zwischen den Knoten gibt es Kanten $E \subseteq V \times V$

Graphen

- Wir ersparen uns eine formale Definition!
- Ein Graph besteht aus einer Menge Knoten V
- Zwischen den Knoten gibt es Kanten $E \subseteq V \times V$

Einsatzgebiete

- Verkehrs- und sonstige Netze (kürzeste Wege)
- Ablaufpläne
- Partitionierungsalgorithmen
- Viele algorithmische Probleme lassen sich auf Graphen zurückführen
- ...

Graphen

- Wir ersparen uns eine formale Definition!
- Ein Graph besteht aus einer Menge Knoten V
- Zwischen den Knoten gibt es Kanten $E \subseteq V \times V$

Einsatzgebiete

- Verkehrs- und sonstige Netze (kürzeste Wege)
- Ablaufpläne
- Partitionierungsalgorithmen
- Viele algorithmische Probleme lassen sich auf Graphen zurückführen
- ...

Was prinzipiell möglich ist

- Kanten können gerichtet oder ungerichtet sein
- Kanten und Knoten können Gewichte haben
- zwischen zwei Knoten können beliebig viele Kanten sein
- Ein Graph kann zyklisch, planar oder zusammenhängend (und jeweils auch das Gegenteil sein)
- u.v.m.

Beispiel: Routensuche

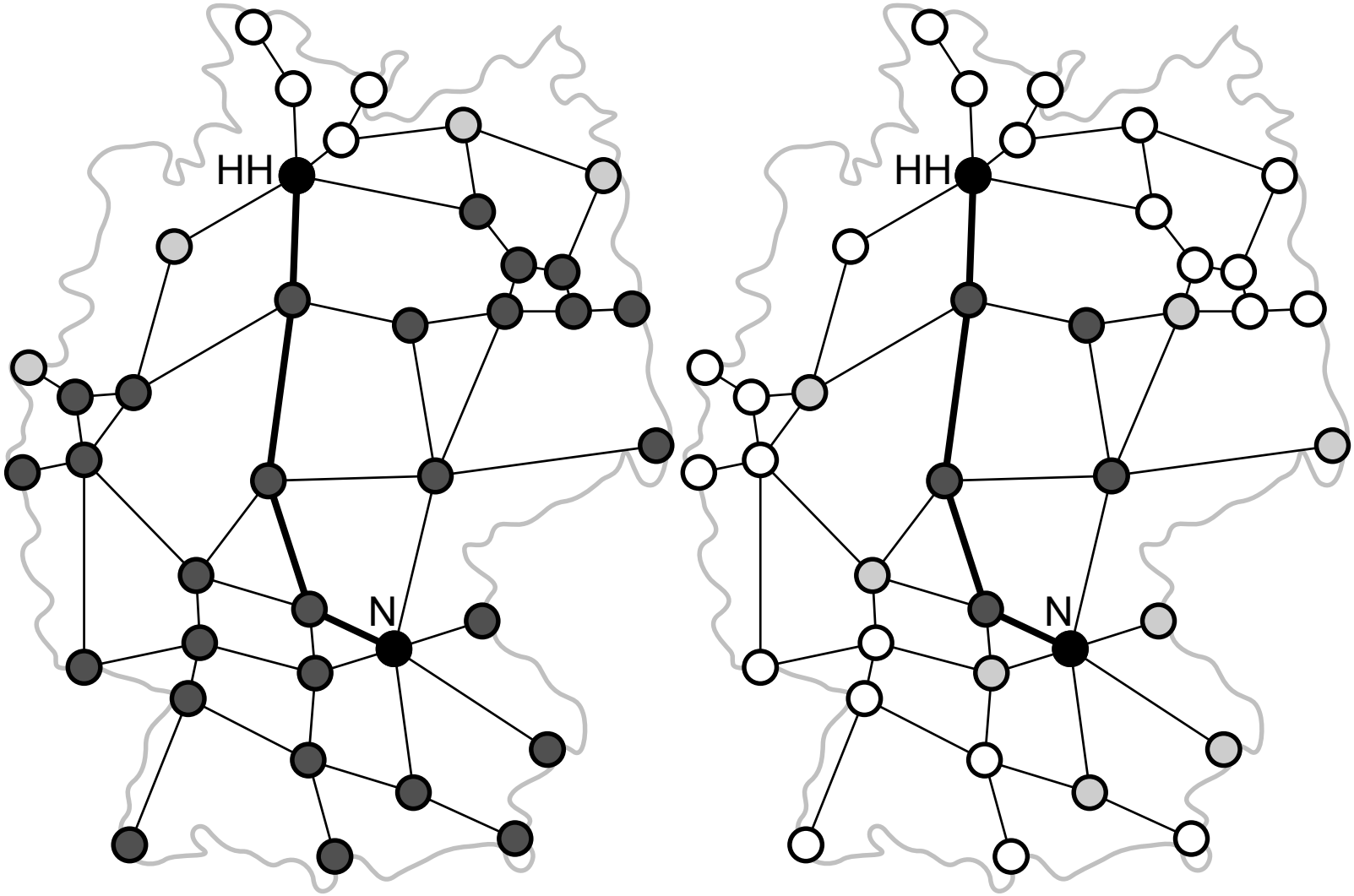


Abbildung: Dijkstra bzw. A*; Quelle: Bungartz, Zimmer, Buchholz, Pflüger: Modellbildung und Simulation - Eine anwendungsorientierte Einführung

Datenstrukturen für Graphen

Adjazenzmatrix

- Matrix A mit Dimension $|V| \times |V|$
- $A[i,j] = 1$ wenn $(i,j) \in E$
- $A[i,j] = 0$ sonst
- Gut geeignet für dichte Graphen

Datenstrukturen für Graphen

Adjazenzmatrix

- Matrix A mit Dimension $|V| \times |V|$
- $A[i,j] = 1$ wenn $(i,j) \in E$
- $A[i,j] = 0$ sonst
- Gut geeignet für dichte Graphen

Adjazenzliste

- Liste A mit $|V|$ Listen
- n-te Liste aus A enthält alle Knoten, die über eine Kante mit Knoten n verbunden sind
- Gut geeignet für dünne Graphen
- In Python alternativ als dictionary realisierbar

Datenstrukturen für Graphen

Adjazenzmatrix

- Matrix A mit Dimension $|V| \times |V|$
- $A[i,j] = 1$ wenn $(i,j) \in E$
- $A[i,j] = 0$ sonst
- Gut geeignet für dichte Graphen

Adjazenzliste

- Liste A mit $|V|$ Listen
- n-te Liste aus A enthält alle Knoten, die über eine Kante mit Knoten n verbunden sind
- Gut geeignet für dünne Graphen
- In Python alternativ als dictionary realisierbar

Objektorientiert

- Knoten sind Objekte
- Jeder Knoten enthält eine Liste mit benachbarten Knoten

(gerichtete) Bäume

- Gerichteter, zusammenhängender, planarer und azyklischer Graph
- Hat einen Wurzelknoten (nur ausgehended Kanten)
- Wenn eine Kante von A zu B gerichtet ist, nennt sich A Vaterknoten von B bzw. B Kindknoten von A
- Knoten ohne Kinder heißen Blätter
- Knoten können so in einer Hierarchie angeordnet werden, dass jeder Knoten (außer dem Wurzelknoten) genau eine Eingangskante hat, die von einem Knoten in der darüberliegenden Hierarchieebene liegt.

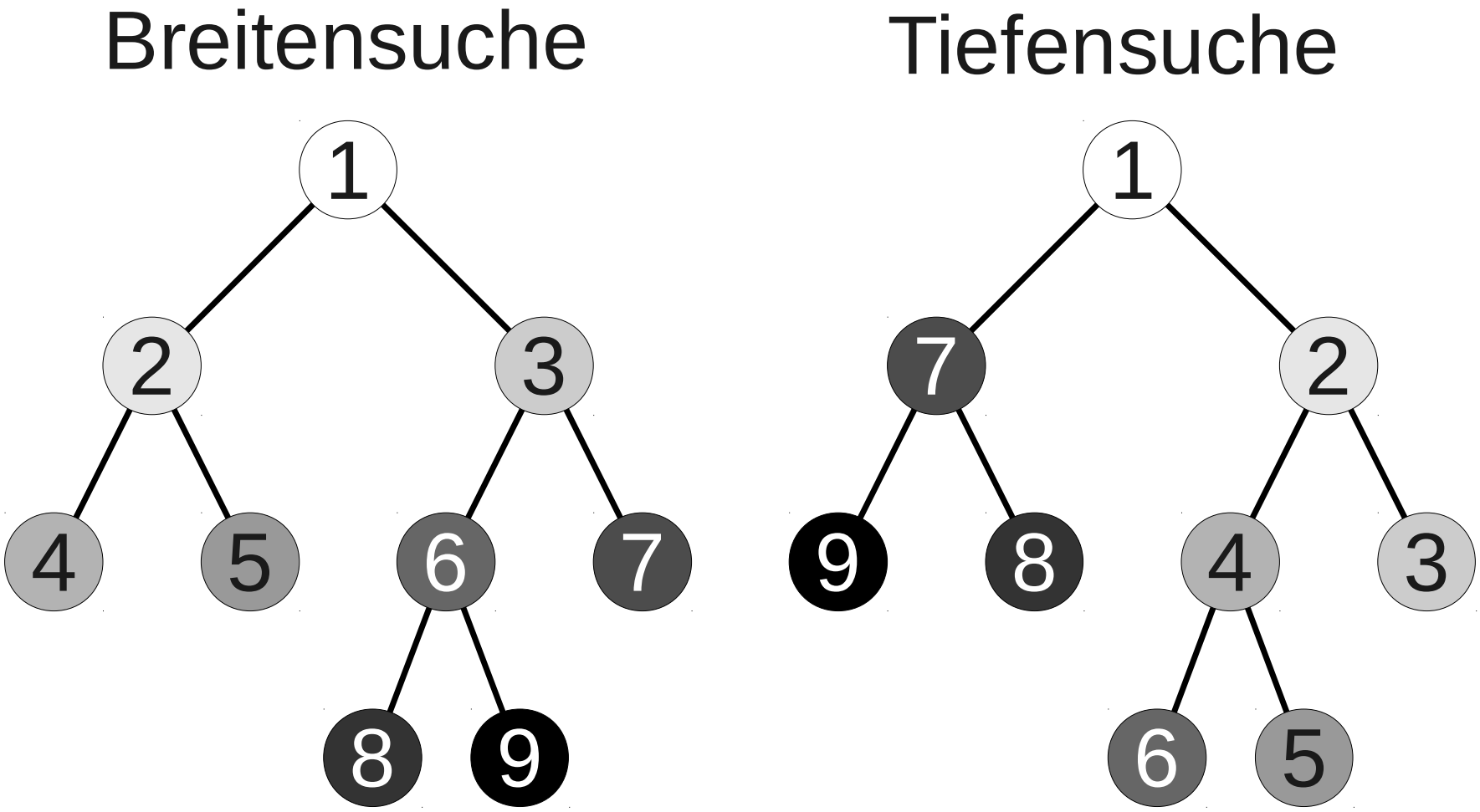
(gerichtete) Bäume

- Gerichteter, zusammenhängender, planarer und azyklischer Graph
- Hat einen Wurzelknoten (nur ausgehended Kanten)
- Wenn eine Kante von A zu B gerichtet ist, nennt sich A Vaterknoten von B bzw. B Kindknoten von A
- Knoten ohne Kinder heißen Blätter
- Knoten können so in einer Hierarchie angeordnet werden, dass jeder Knoten (außer dem Wurzelknoten) genau eine Eingangskante hat, die von einem Knoten in der darüberliegenden Hierarchieebene liegt.

Einsatzgebiet von Bäumen

- Speicherung (sortierter) Daten (logarithmischer Zugriff)
- Entscheidungs- und Suchbäume
- (hierarchische) Gebietszerlegungen
- Viele algorithmische Probleme lassen sich auf Bäume zurückführen

Bäume: Beispiel Breiten- und Tiefensuche



Tiefensuche in (unsortierten) Bäumen/Graphen

- Erster Kind-/Nachbarknoten, dann dessen Kind/Nachbar, ...
- Worst-Case Laufzeit $O(|V| + |E|)$
- Es ist möglich, dass der Knoten nicht gefunden wird (Bei unendlichen Graphen oder wenn Zyklen nicht überprüft werden)

Iterativer Algorithmus mit Stapel

```
def tiefensuche(startknoten, zielknoten):
  startknoten.besucht = True
  stapel = [startknoten]
  while len(stapel)>0:
    knoten = stapel.pop()
    if knoten==zielknoten:
      return knoten
    for nachbar in knoten.nachbarn:
      if nachbar.besucht == False:
        nachbar.besucht = True
        stapel.append(nachbar)
```

Breitensuche in (unsortierten) Bäumen/Graphen

- Erst Knoten mit Entfernung 1 zum Startknoten, dann 2, ...
- Bei Bäumen: ebenenweiser Durchlauf
- Worst-Case Laufzeit $O(|V| + |E|)$
- Wenn der gesuchte Knoten existiert, wird er auch gefunden

Iterativer Algorithmus mit Warteschlange

```
def breitensuche(startknoten, zielknoten):
  startknoten.besucht = True
  warteschlange = deque([startknoten])
  while len(warteschlange)>0:
    knoten = warteschlange.popleft()
    if knoten==zielknoten:
      return knoten
    for nachbar in knoten.nachbarn:
      if nachbar.besucht == False:
        nachbar.besucht = True
        warteschlange.append(nachbar)
```

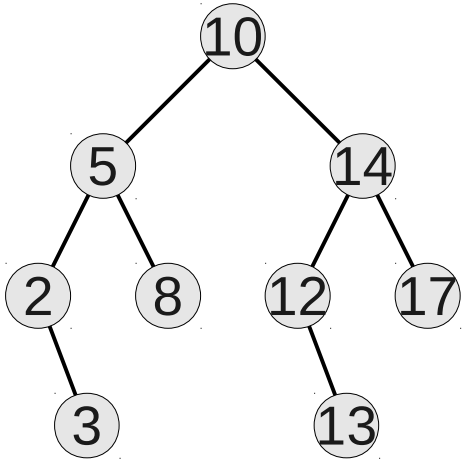
Suchbäume

- Bisherige Verfahren: nicht effizient ($O(|V| + |E|)$)
- Beide Verfahren: „uninformiert“, d.h. alles wird blind abgesucht
- Heuristiken: können Suche beschleunigen (z.B. A*-Algorithmus bei Suche nach kürzesten Wegen)
- Falls Elemente des Baumes Totalordnung unterliegen: **Suchbäume** noch viel effizienter
- Suchbäume unterstützen drei Operationen:
  - Einfügen eines Elements (je nach Baum evtl. $O(1)$)
  - Löschen eines Elements (je nach Baum evtl. $O(1)$)
  - Suchen eines Elements ($O(Baumhoehe)$)
- Unbalancierte Suchbäume: Baumhöhe max. N
- Balancierte Suchbäume (z.B. AVL-Baum): Baumhöhe = log(N)

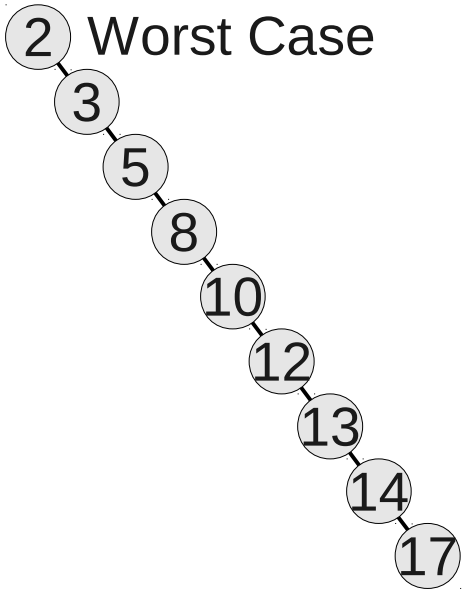
Binärer Suchbaum

- Erstes hinzugefügtes Element: bildet Wurzel
- Neue Elemente:
  - werden als Blätter in Baum aufgenommen
  - werden von Wurzel gemäß Sortierkriterium bis zu entsprechender Stelle durchgereicht
- Eigenschaften:
  - Baumstruktur: wird beim Hinzufügen nicht verändert
  - Suche: analog zum Einfügen (Vergleich des gesuchten Elements mit dem aktuellen Knoten, dann evtl. Abstieg in linken/rechten Teil, ...)
  - Löschen: evtl. etwas komplexer (Teilbäume werden umgehängt)

optimaler Fall



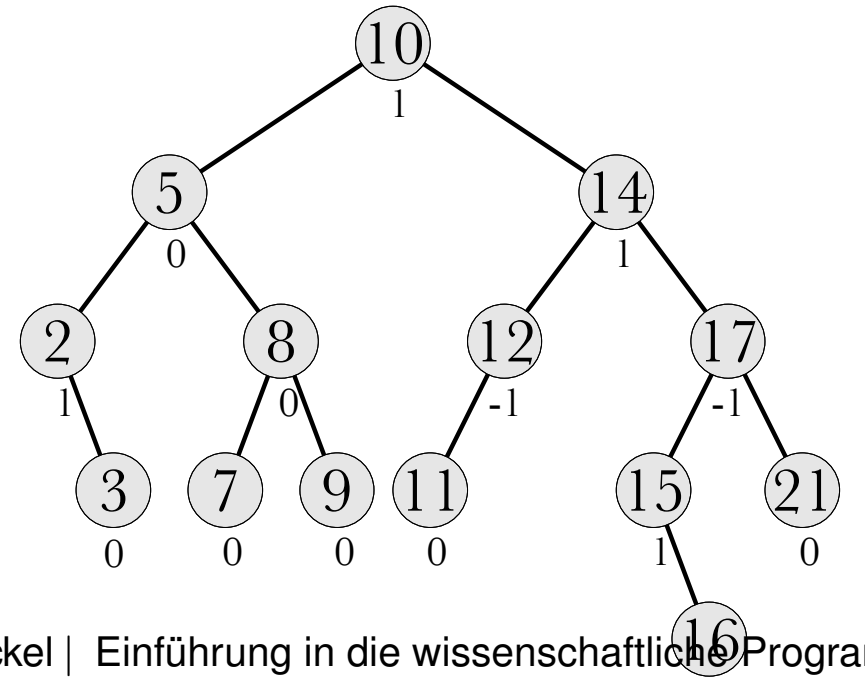
Worst Case



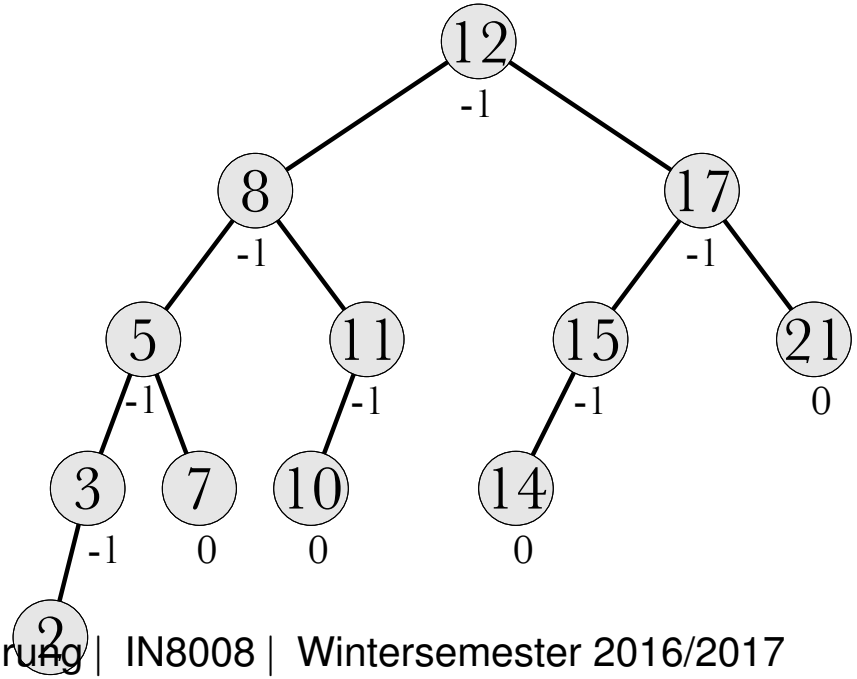
AVL-Baum

- Binärer Suchbaum, bei dem für jeden Knoten k gilt, dass die Höhe von linkem ($H_l(k)$ und rechtem ($H_r(k)$) Teilbaum sich um maximal 1 unterscheiden.
- Höhe im schlimmsten Fall ca. $1.44 \cdot log_2(N+2)$, d.h. Suchen eines Elements immer in $O(log(N))$;
- Für jeden Knoten gibt es ein Balance-Kriterium $bal(k) = H_r(k) - H_l(k)$, das immer $\in \{-1, 0, 1\}$ sein muss.

normaler Fall



Worst Case



Einfügen eines Elements

- Vor dem Einfügen sind alle Balancewerte $\in \{-1, 0, 1\}$.
- Zunächst wir der Einfügepunkt gesucht.

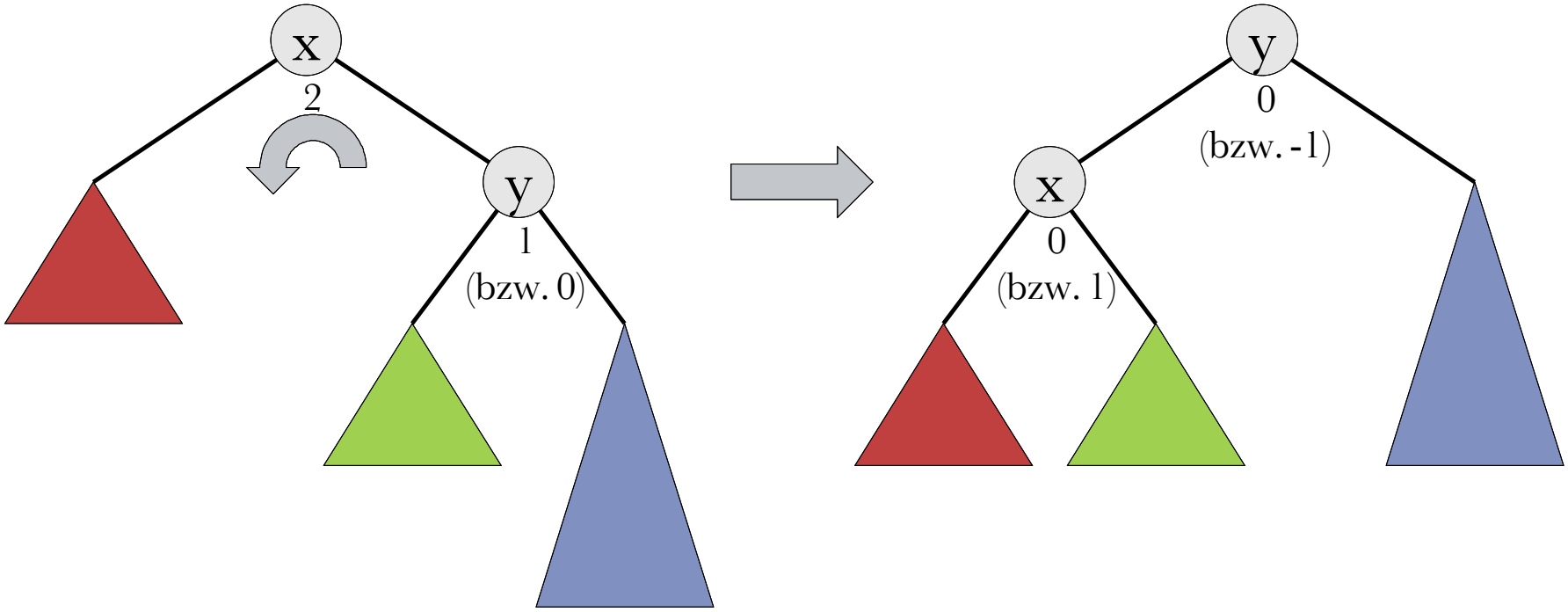
Einfügen bei Knoten mit einem Kind

- Der bisherige Balancewert des Knotens ist $\in \{-1, 1\}$.
- Der neue Balancewert ist 0.
- $\Rightarrow$ keine Rebalancierung notwendig

Einfügen bei Blattknoten

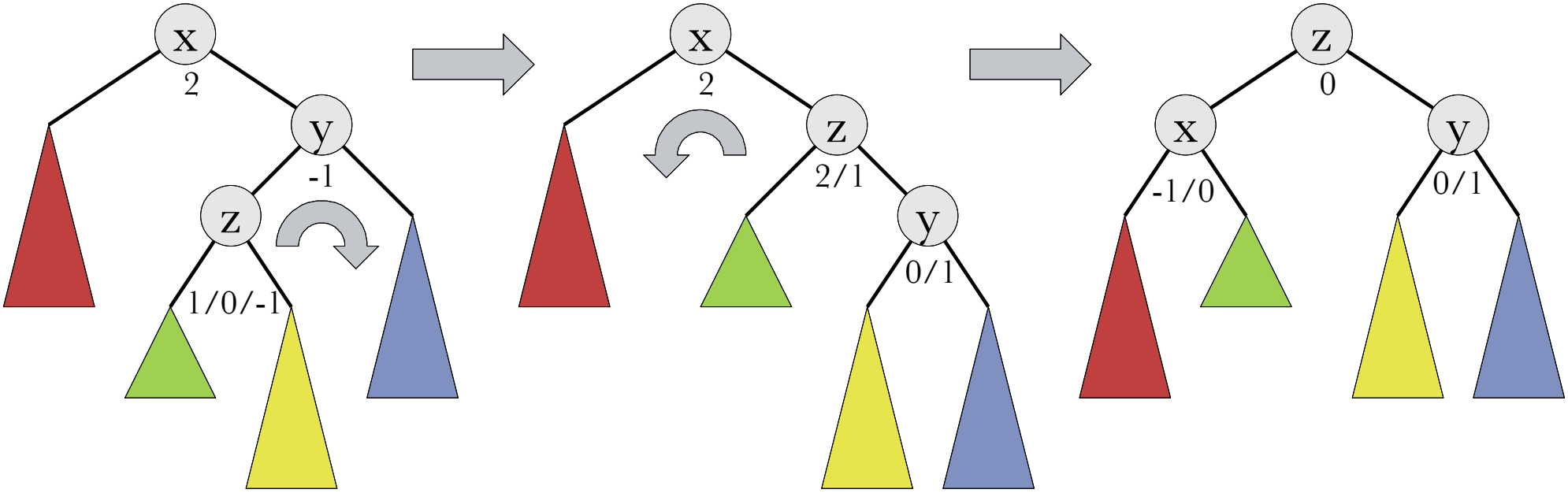
- Der bisherige Balancewert des Knotens ist 0.
- Neuer Balancewert ist $\in \{-1, 1\}$ (aktueller Teilbaum höher).
- Information muss rekursiv nach oben weitergegeben werden.
- Bei jedem Vaterknoten wird der neue Balancewert berechnet.
  - Bei Balancewert $\in \{-1, 1\}$ geht man weiter nach oben.
  - Bei Balancewert 0 kann der Aufstieg abgebrochen werden.
  - Bei Balancewert $\in \{-2, 2\}$ muss neu balanciert werden.
  - Nach Balancierung muss nicht weiter nach oben gegangen werden, die übrigen Balancewerte ändern sich nicht.

Balancieren des Baums: Einfachrotation



- Rechter Sohn von x ist y
- Linker Teilbaum von x ist am niedrigsten
- Linker Teilbaum von y ist nicht höher als rechter Teilbaum von y
- $\Rightarrow$ y wird Wurzel, x linker Sohn, bisheriger linker Teilbaum von y wird rechter Teilbaum von x, Rest bleibt gleich
- Höhe des neuen Baums gleich wie vor dem Einfügen, Balancierung beendet
- Analoges Vorgehen im spiegelverkehrten Fall

Balancieren des Baums: Doppelrotation



- Nun ist der linke Teilbaum von y (mit Wurzel z) höher als der rechte.
- $\Rightarrow$ Einfachrotation würde Balance von x von 2 auf -2 ändern
- Zunächst Rotation, um dafür zu sorgen, dass der rechte Teilbaum der höchste ist;
- Dann eine weitere Einfachrotation wie auf der vorigen Folie.

Löschen eines Elements

- Vor dem Löschen sind alle Balancewerte $\in \{-1, 0, 1\}$.
- Zunächst wir das zu löschende Element k gesucht

Knoten k ist ein Blatt

- Der Knoten wird gelöscht, der Teilbaum um 1 niedriger.
- $\Rightarrow$ Balance des Vaters (rekursiv) anpassen
- $\Rightarrow$ evtl. Rebalancierung

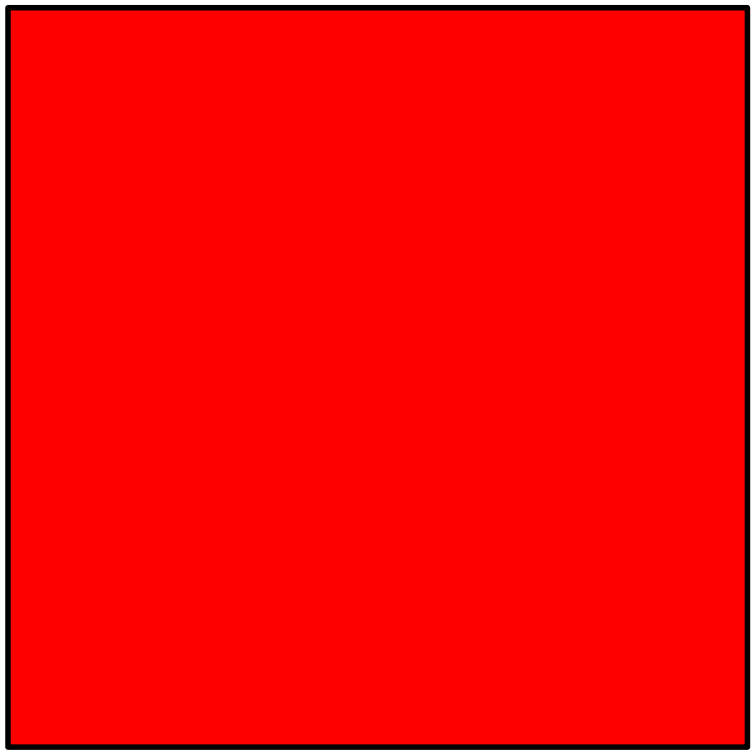
Knoten k hat genau einen Sohn s

- s tritt an Stelle von k, Teilbaum wird um 1 niedriger.
- $\Rightarrow$ rekursiv Balance anpassen

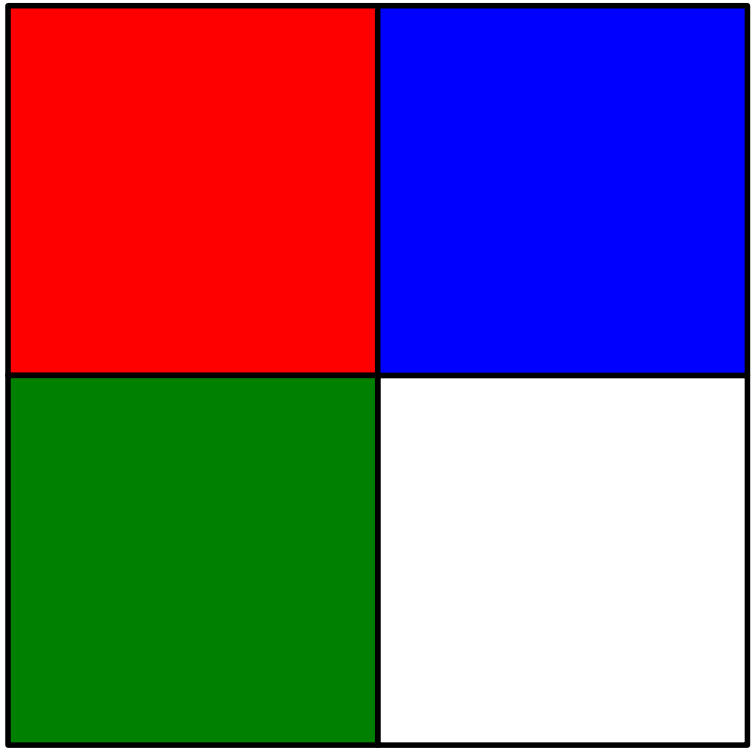
Knoten k hat zwei Söhne

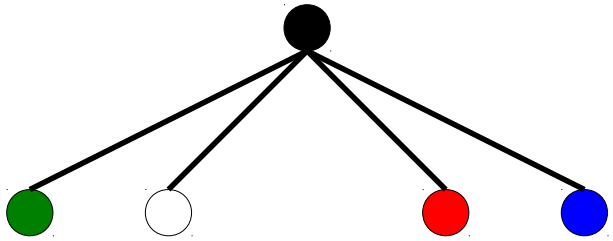
- Bei Balancewert $\in \{-1, 1\}$ geht man weiter nach oben.
- Bei Balancewert 0 kann der Aufstieg abgebrochen werden.
- Bei Balancewert $\in \{-2, 2\}$ muss neu balanciert werden.
- Beim Löschen kann sich im Gegensatz zum Einfügen die Balance des Vaterknotens eines gerade balancierten Teilbaums auch ändern, es muss weiter nach oben gegangen werden.

Beispiel Bäume: Spacetrees (1)

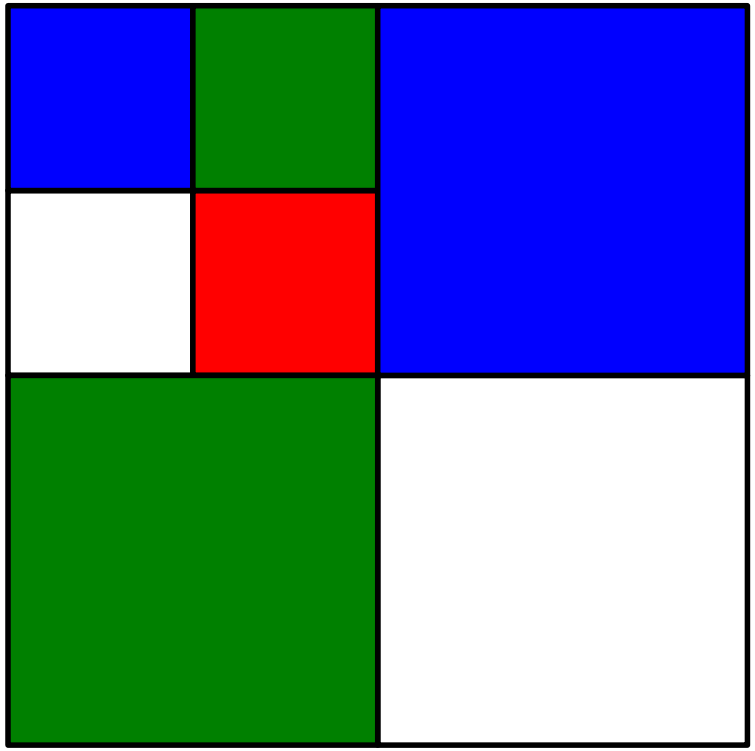


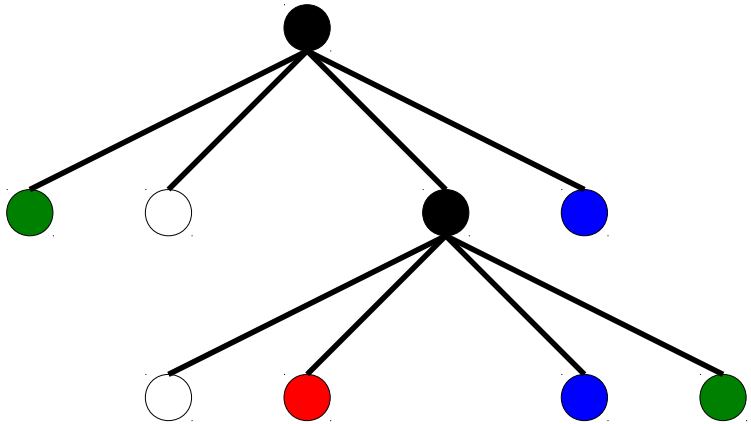
Beispiel Bäume: Spacetrees (1)



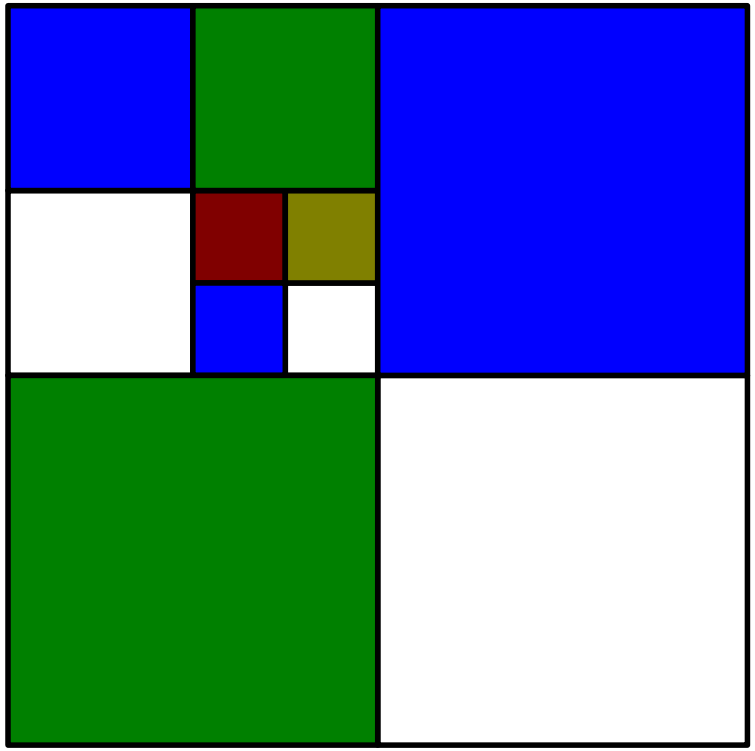


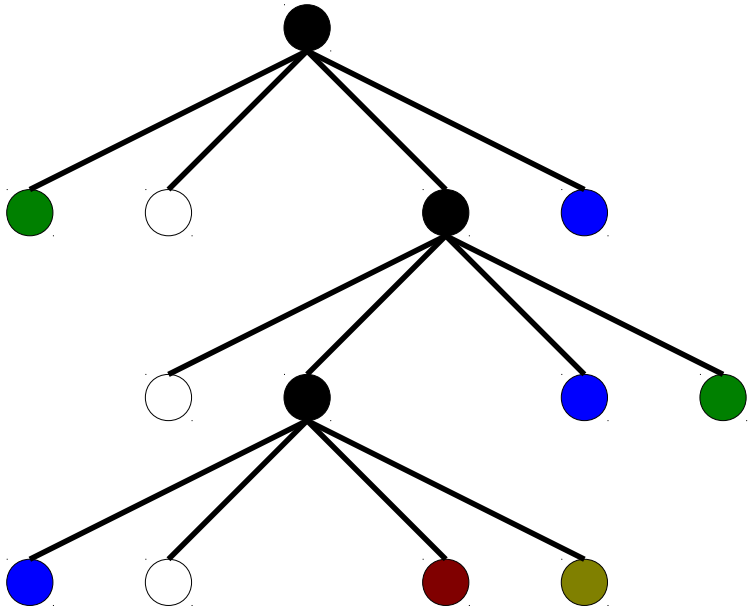
Beispiel Bäume: Spacetrees (1)





Beispiel Bäume: Spacetrees (1)

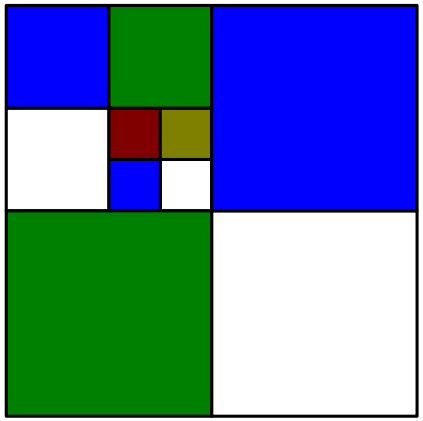


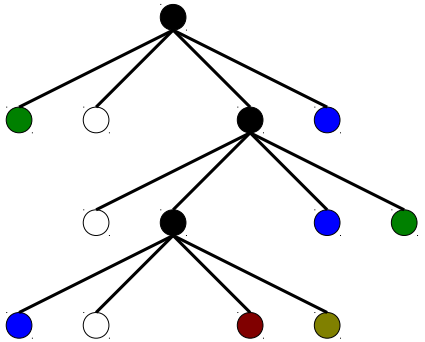


Beispiel Bäume: Spacetrees (2)

Konzept

- Baumbasierte Hierarchische Raumaufteilung
- Erlaubt Gebietsverteilung und Partitionierung
- Raumadaptives Gitter
  - Approximation von Geometrien
  - Simulationsanwendungen





Spacetrees: Kreisapproximation (1)

Beschreibung der Geometrie: Klasse `Geometry`

- Methode `schneidetVolumen(LL,UR)`
  $\rightarrow$ gibt `True` zurück, falls die Geometrie die Box mit unterer linker Ecke `LL` und rechter oberer Ecke `UR` schneidet
- `Sphere`: erbt von `Geometry` und implementiert 2D-Kugel (Kreis)

Exzerpt (`geometry.py`):

```
class Geometry:
  def schneidetVolumen(self,ll,ur):
    pass

class Sphere(Geometry):
  def __init__(self,center,radius):
    ...

  def schneidetVolumen(self,ll,ur):
    ...
```

Spacetrees: Kreisapproximation (2)

Adaptives Gitter: Klasse `Spacetree`

- Konstruktor-Argumente:
  - `Geometry`-Objekt
  - Bounding Box (linke,untere Ecke `LL` und rechte,obere Ecke `UR`)
  - Max. Anzahl von Baumlevels `levels`
- Konstruktor: Baut den Spacetree rekursiv auf:
  - Falls der Baum nur ein Level (`levels`=0) besitzt: lege nur eine Zelle an
  - Andernfalls: Teste, ob die momentane Zelle die Geometrie schneidet
    $\rightarrow$ Zelle schneidet Geometrie: lege 4 Spacetrees der Höhe levels-1 an und bette sie in momentane Gitterzelle ein
    $\rightarrow$ Kein Schnitt: keine weitere Operation

Spacetrees: spacetree.py (1)

```
class Spacetree:
  def __init__(self,LL,UR,geometry,levels):
    self.LL = LL          #lower left corner of grid cell
    self.UR = UR          #upper right corner of grid cell
    self.levels=levels    #number of grid levels
    self.kinder=[]        #subtrees

    if (levels==0):
      return

    ...
```

Spacetrees: spacetree.py (2)

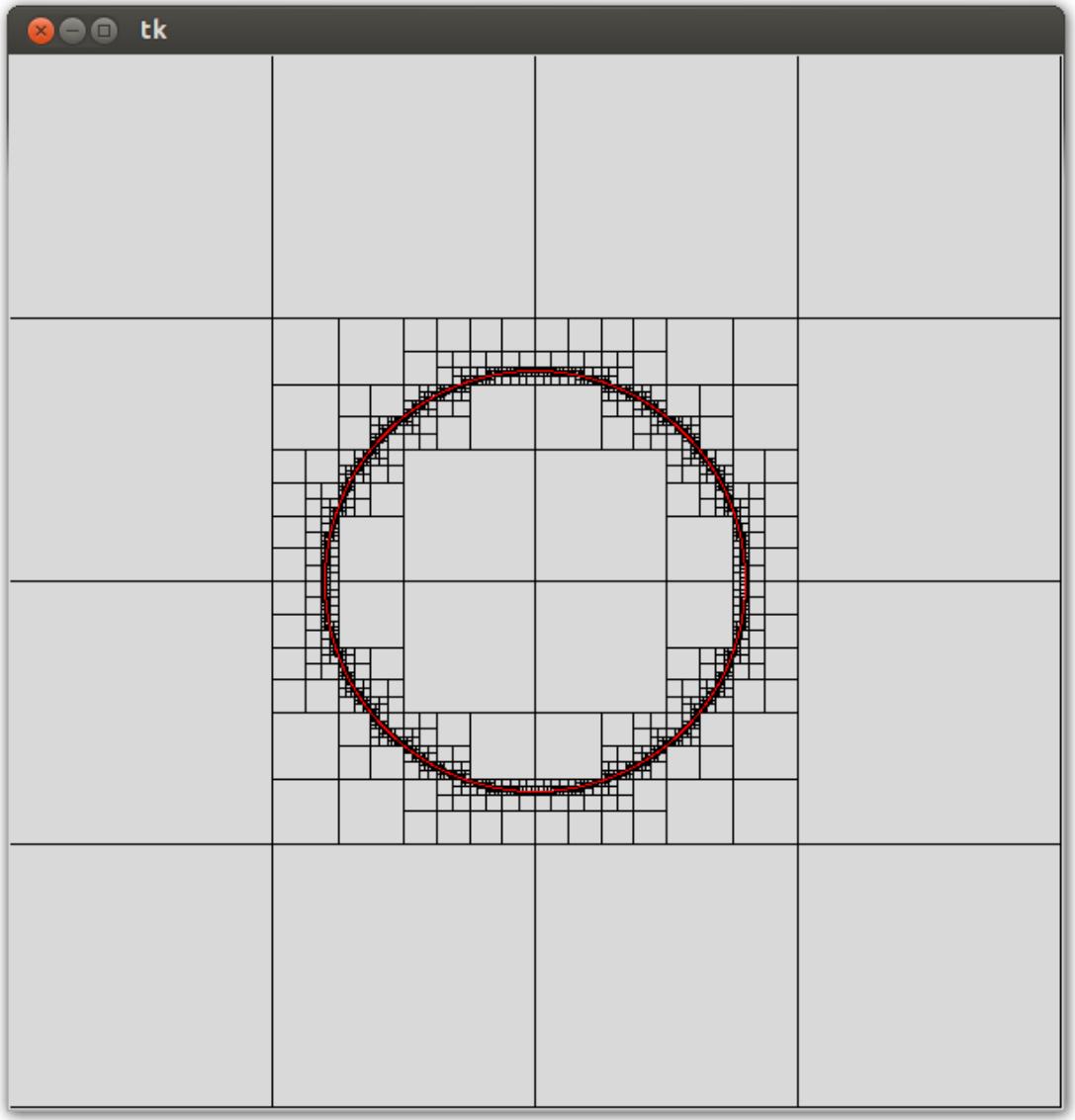
```
...
# berechne Punkte in LL,UR Flaeche
# x6----x7----UR
# :     :     :
# x3----x4----x5
# :     :     :
# LL----x1----x2
x1 = ( 0.5*(LL[0]+UR[0]), LL[1] )
x3 = ( LL[0]            , 0.5*(LL[1]+UR[1]) )
x4 = ( 0.5*(LL[0]+UR[0]), 0.5*(LL[1]+UR[1]) )
x5 = ( UR[0]            , 0.5*(LL[1]+UR[1]) )
x7 = ( 0.5*(LL[0]+UR[0]), UR[1] )
# Falls nahe am Rande der Geometrie -> Verfeinern
if geometry.schneidetVolumen(LL,UR):
  self.kinder = [Spacetree(LL,x4,geometry,levels-1), \
                 Spacetree(x1,x5,geometry,levels-1), \
                 Spacetree(x3,x7,geometry,levels-1), \
                 Spacetree(x4,UR,geometry,levels-1) ]
```

Spacetrees: spacetree.py (3)

- Methode `visualisiere(myvis)`
  - Durchlaufe Baum “depth-first” (vgl. Tiefensuche)
  - Visualisiere jedes Blatt mit Hilfe des Visualisierungsobjekts `myvis` (vgl. Vis. der Partikelsysteme)

```
def visualisiere(self,myvis):
  # Falls Kinder vorhanden: rekursives Plotten
  if len(self.kinder)>0:
    for k in self.kinder:
      k.visualisiere(myvis)
  # Falls Blatt: visualisiere Gitterzelle
  else:
    myvis.visualisiereZelle(self.LL,self.UR)
```

Spacetrees: Visualisierung

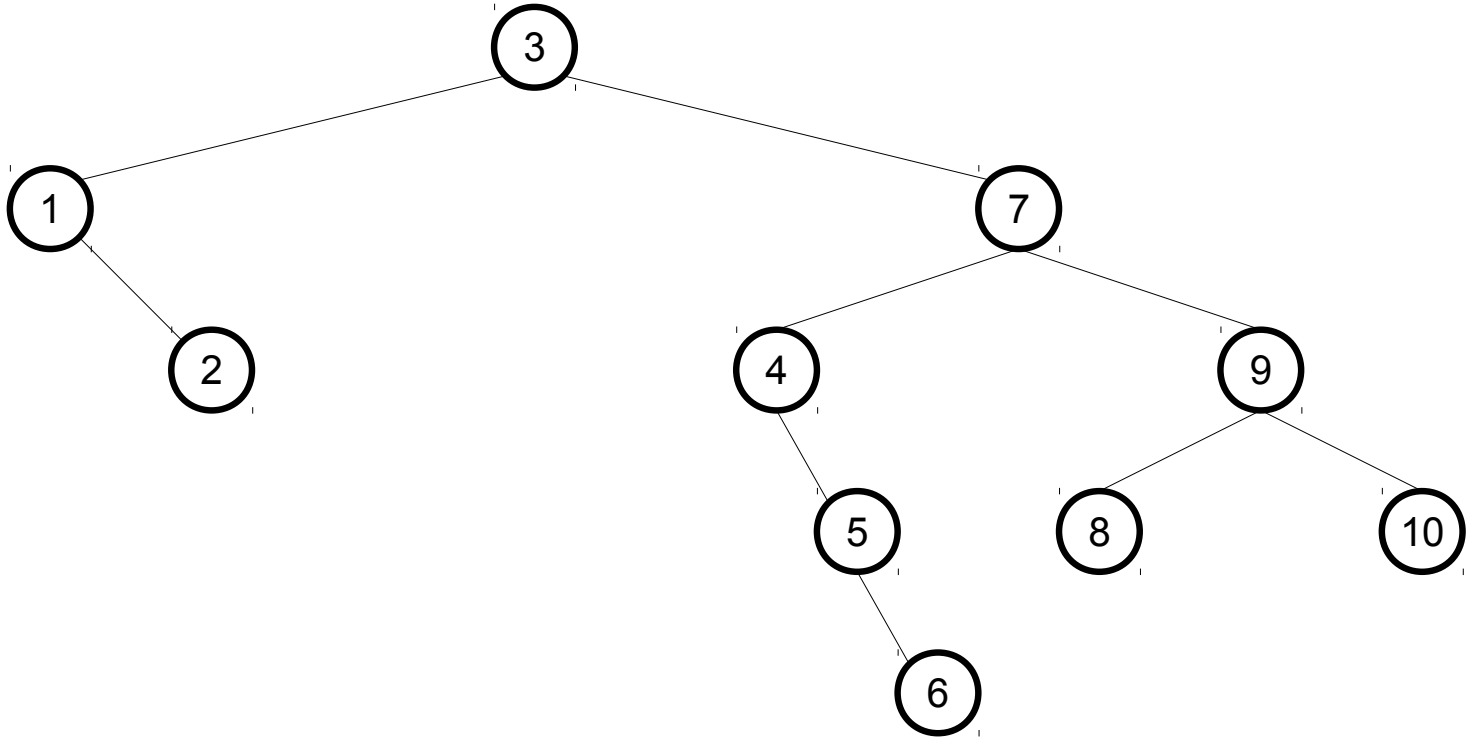


Wiederholung

Hands-On: Sortieren von Zahlen

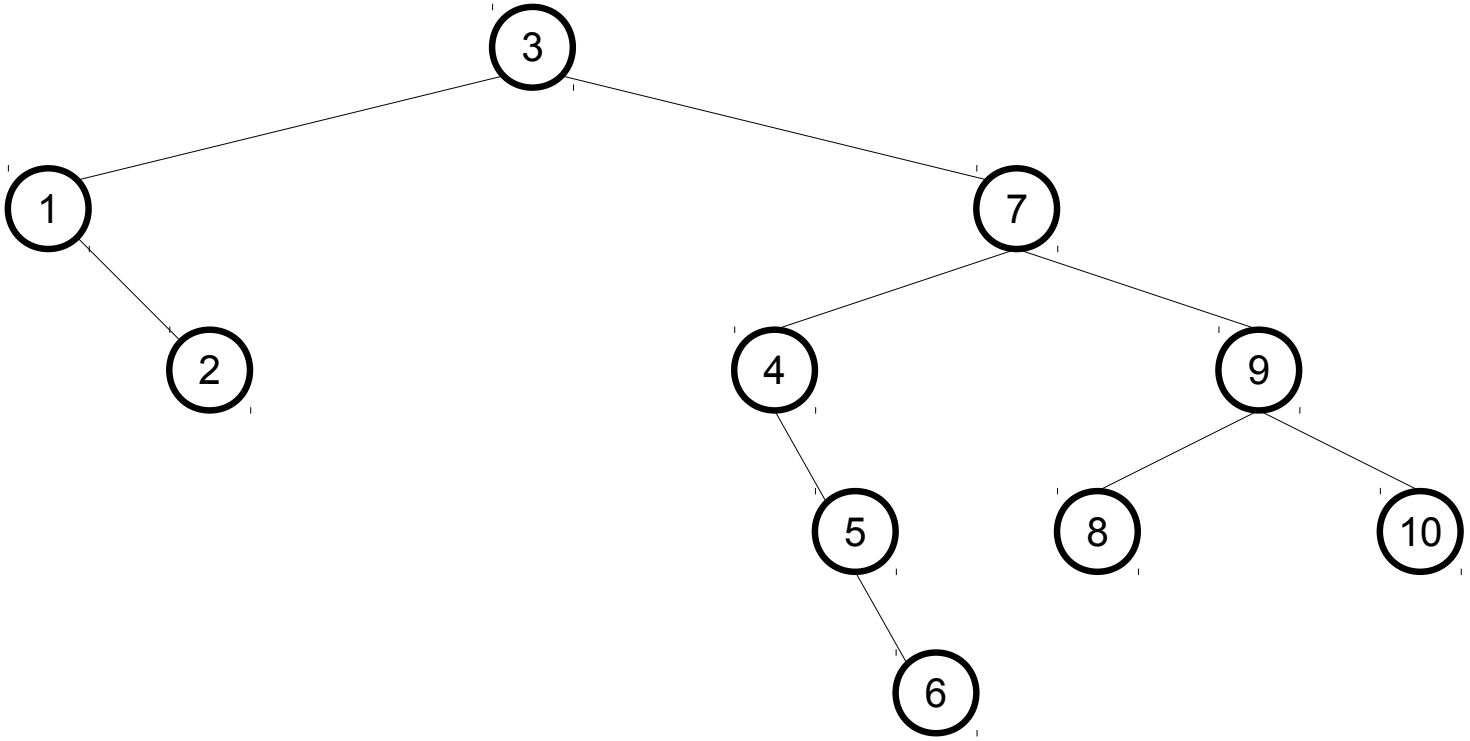
1. Sortieren Sie die folgenden Zahlen mit Hilfe eines binären Suchbaums:
   3 1 7 9 4 5 10 2 8 6
   Die Zahlen sollen in einer Liste vorliegen und von links nach rechts eingelesen und einsortiert werden.
   Stellen Sie den entstehenden Suchbaum graphisch dar.
2. Ist der entstandene Baum ein AVL-Baum? Geben Sie eine kurze Begründung.
3. Wie kann der entstandene Baum mit möglichst wenigen Operationen in einen AVL-Baum überführt werden?

1. Sortieren Sie die folgenden Zahlen mit Hilfe eines binären Suchbaums:
   3 1 7 9 4 5 10 2 8 6
   Die Zahlen sollen in einer Liste vorliegen und von links nach rechts eingelesen und einsortiert werden.
   Stellen Sie den entstehenden Suchbaum graphisch dar.



2. Ist der entstandene Baum ein AVL-Baum? Geben Sie eine kurze Begründung.

1. Sortieren Sie die folgenden Zahlen mit Hilfe eines binären Suchbaums:
   3 1 7 9 4 5 10 2 8 6
   Die Zahlen sollen in einer Liste vorliegen und von links nach rechts eingelesen und einsortiert werden.
   Stellen Sie den entstehenden Suchbaum graphisch dar.



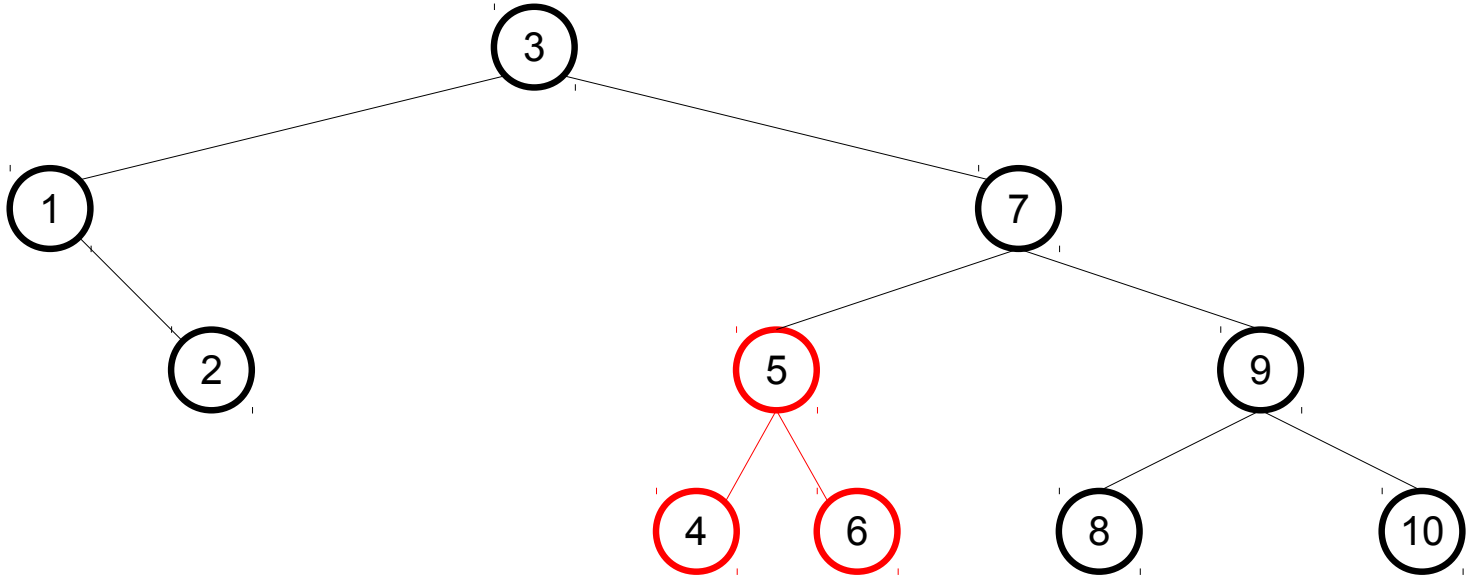
2. Ist der entstandene Baum ein AVL-Baum? Geben Sie eine kurze Begründung.
   Antwort: Nein. Höhe rechter Subbaum: 4, Höhe linker Subbaum: 2.
   $\rightarrow$ Max. Differenz (in AVL-Baum: 1) ist somit überschritten.

Wiederholung

3. Wie kann der entstandene Baum mit möglichst wenigen Operationen in einen AVL-Baum übergeführt werden?

Wiederholung

3. Wie kann der entstandene Baum mit möglichst wenigen Operationen in einen AVL-Baum übergeführt werden?



$\rightarrow$ Einfachrotation im Zweig 4-5-6