Teil XI

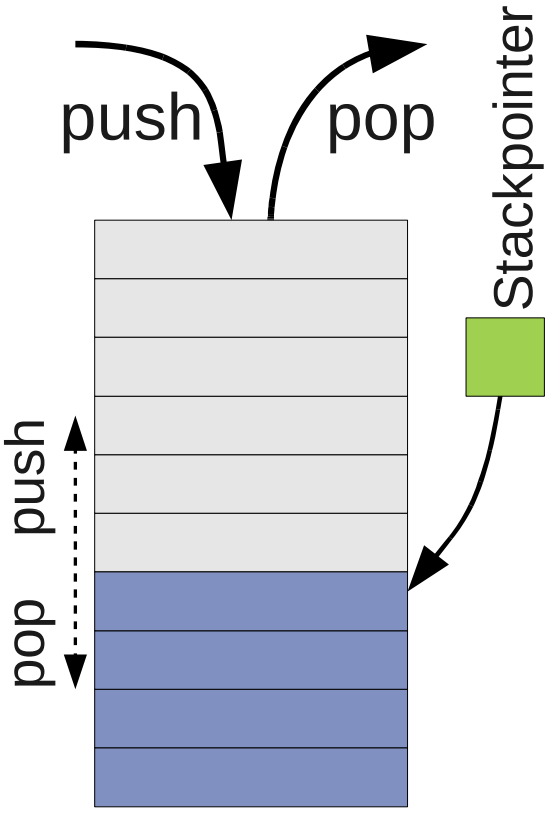
Datenstrukturen: Bäume, Stacks und Queues

Stacks (Kellerspeicher/Stapel)

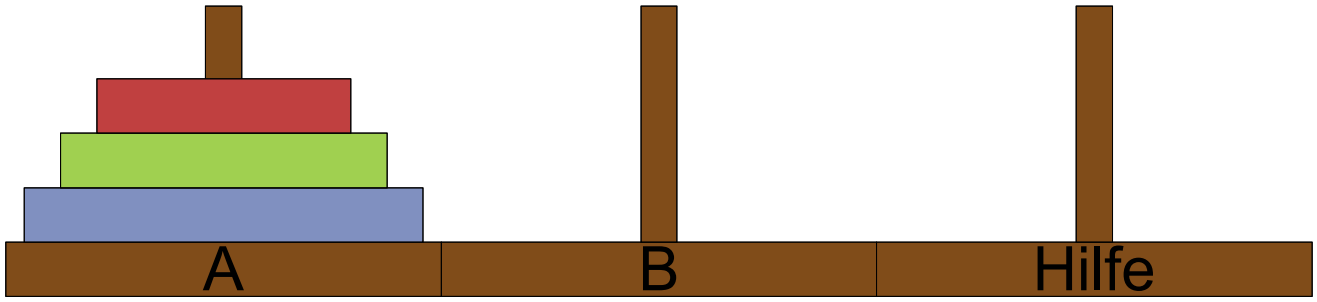
- Funktioniert wie ein natürlicher Stapel (z.B. Papierstapel auf dem Schreibtisch)
- Elemente werden immer oben auf den Stapel gelegt (PUSH)
- Es kann immer nur das oberste Element entfernt werden (POP)
- Funktionsweise: LIFO (Last In, First Out)

Stacks 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()`



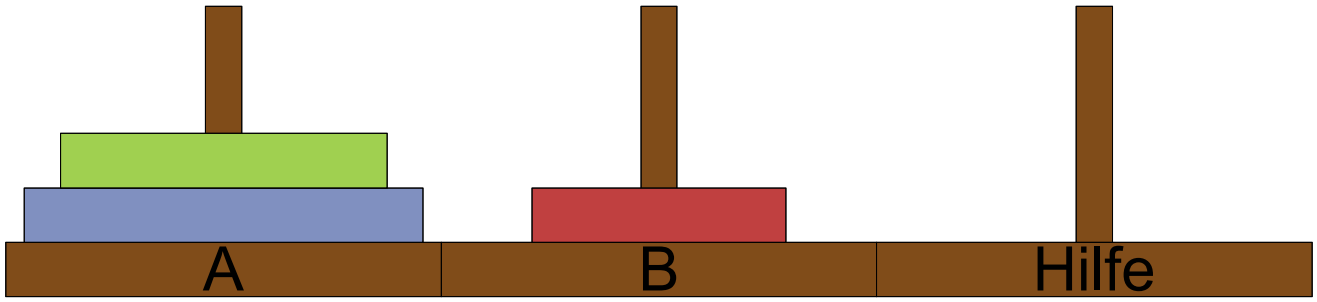
Stack-Beispiel: Türme von Hanoi




```
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)
```

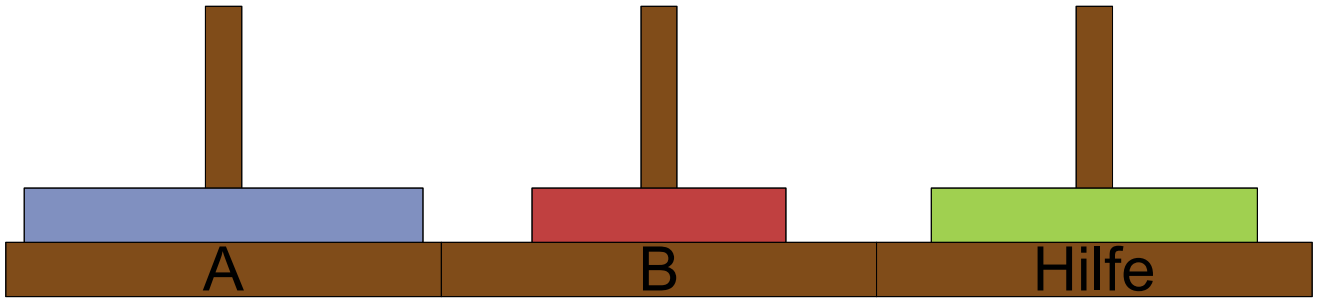
Randoffscher Algorithmus:

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



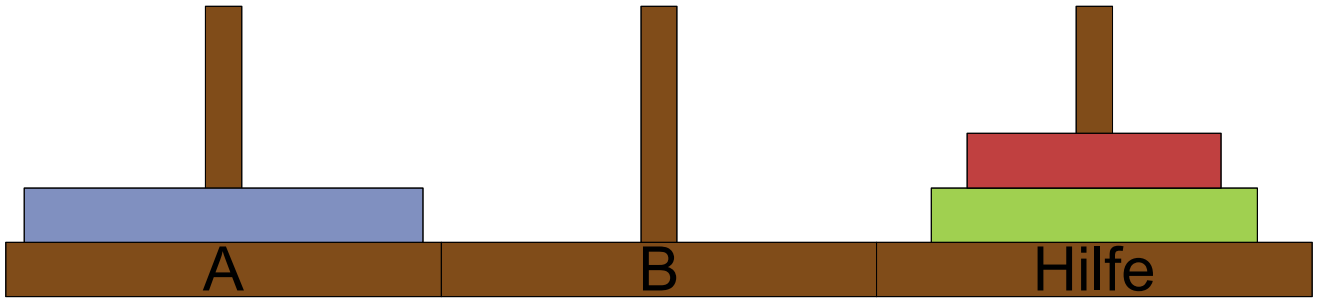

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
```

```
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)
```



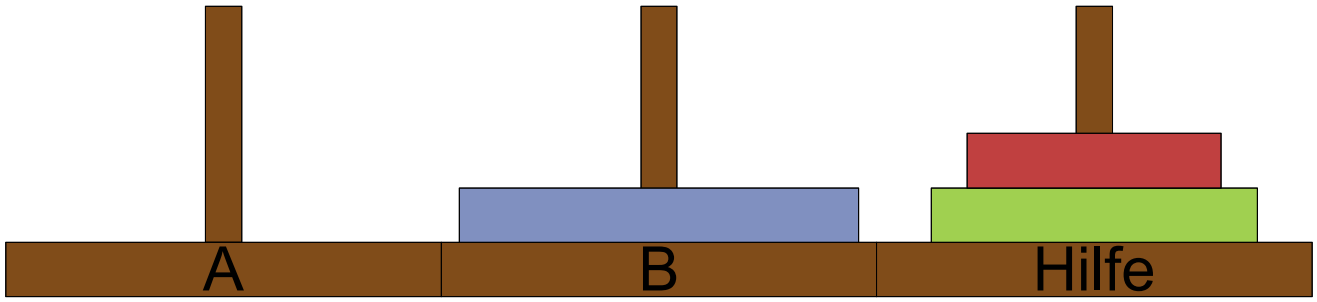

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
```

```
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)
```



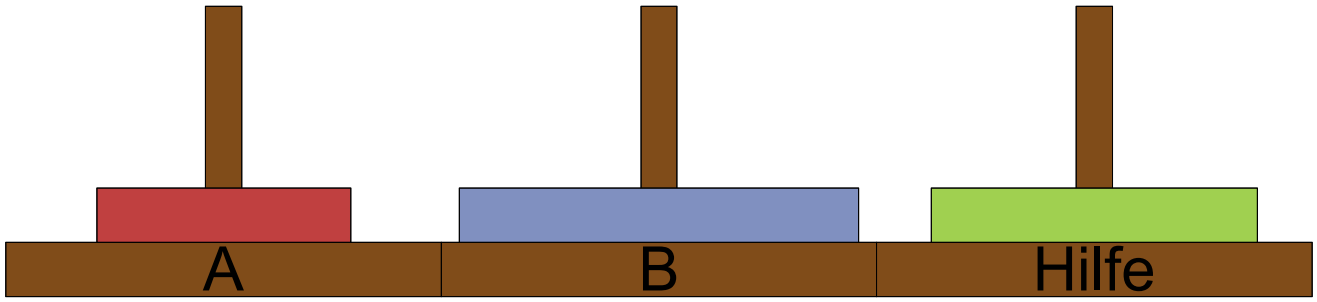

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
```

```
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)
```



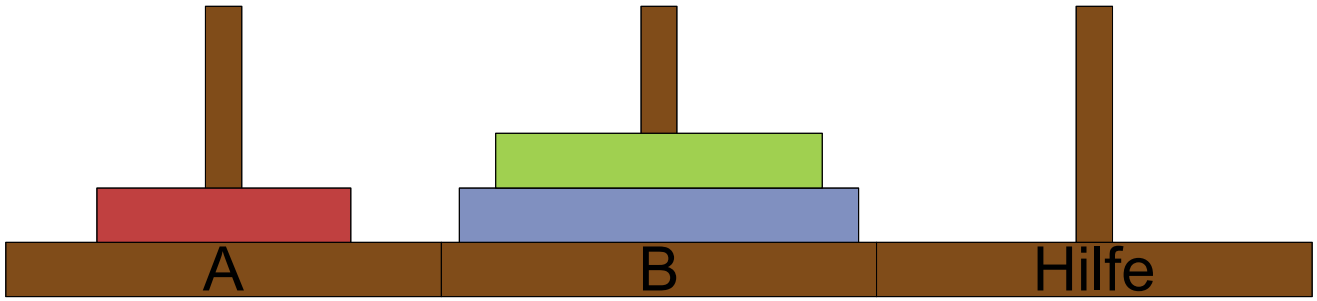

```
hanoi(3,A,B,H)
  hanoi(2,A,H,B)
    hanoi(1,A,B,H)
      A -> B
    A -> H
    hanoi(1,B,H,A)
      B -> H
  A -> B
```

```
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)
```



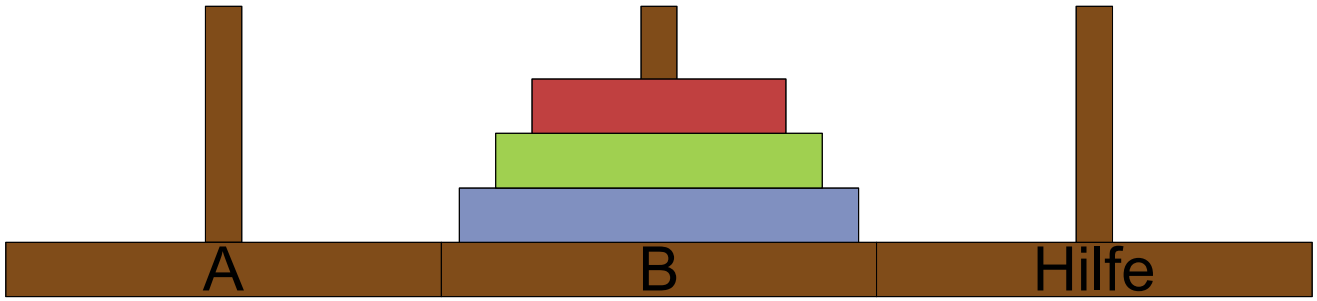

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```

```
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
```



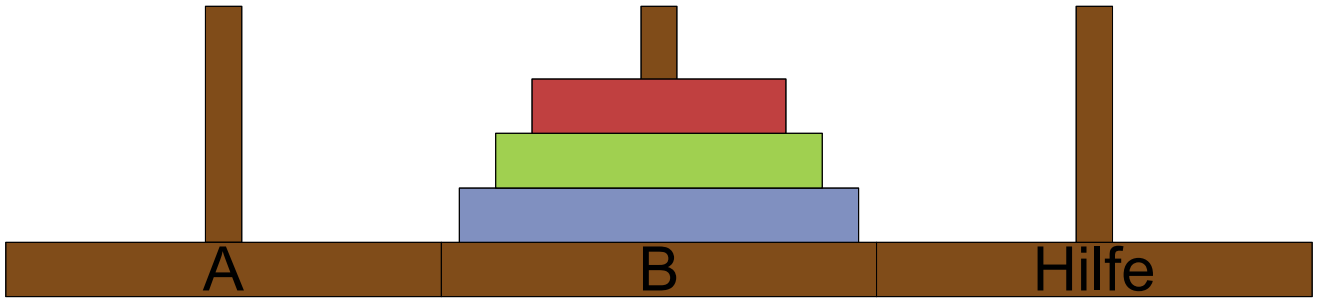

```
def hanoi(n, a, b, hilfe):
  if(n > 0):
    hanoi(n-1, a, hilfe, b)
    b.append(a.pop())
    hanoi(n-1, hilfe, b, a)
```

```
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)
```

Optimale Zugfolge: $2^n - 1$ Züge

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

```
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
```

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
```

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

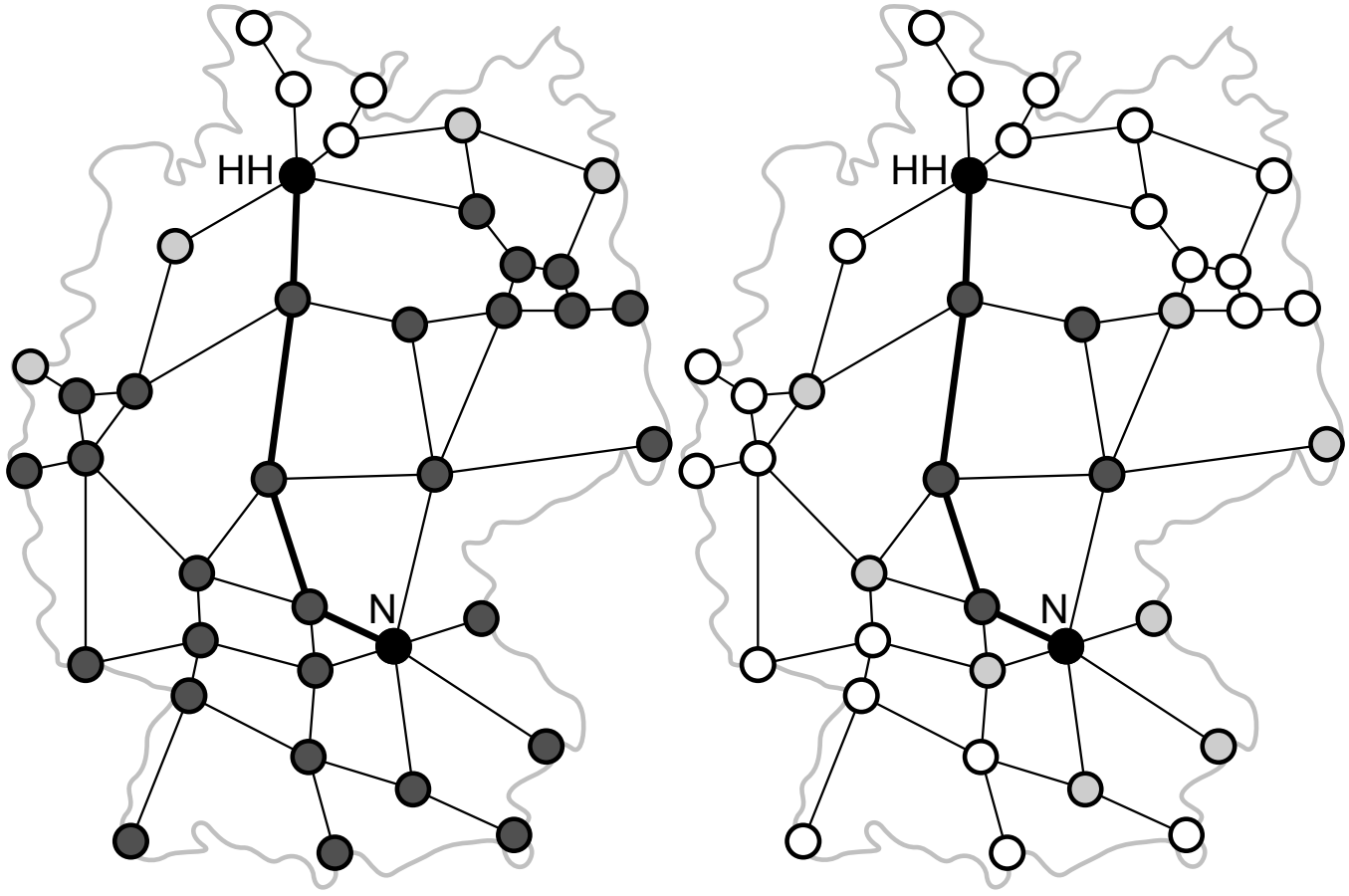


Abbildung: Dijkstra bzw. A*; Quelle: Bungartz, Zimmer, Buchholz, Pflüger: Modellbildung und Simulation - Eine anwendungsorientierte Einführung

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

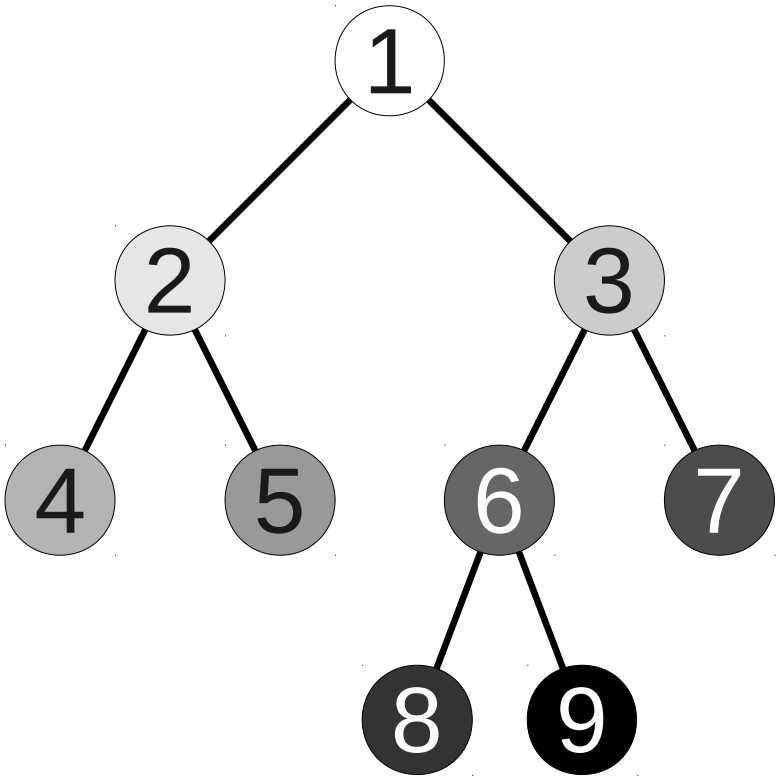
- Gerichteter, zusammenhängender, planarer und azyklischer Graph
- Hat einen Wurzelknoten (nur ausgehended Kanten)
- Wenn eine Kante von A zu B gerichtet ist, nennt sich A Vaterknoten von B bzw. B Kindknoten von A
- Knoten ohne Kinder heißen Blätter
- Knoten können so in einer Hierarchie angeordnet werden,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

Einsatzgebiet von Bäumen

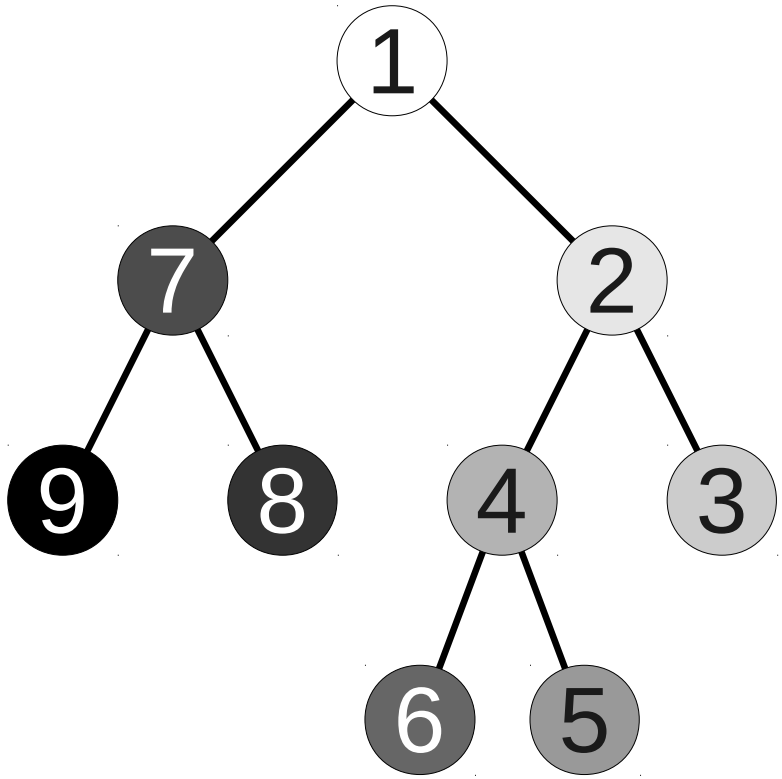
- Speicherung (sortierter) Daten (logarithmischer Zugriff)
- Entscheidungs- und Suchbäume
- (hierarchische) Gebietszerlegungen
- Viele algorithmische Probleme lassen sich auf Bäume zurückführen

Beispiel Breiten- und Tiefensuche

Breitensuche



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

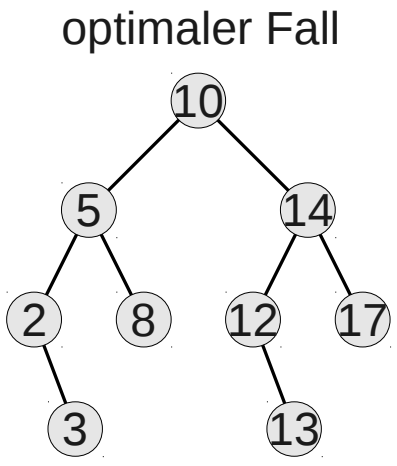
```
def breitensuche(startknoten, zielknoten):
  startknoten.besucht = True
  warteschlange = deque([startknoten])
  while len(warteschlange)>0:
    knoten = warteschlange.popleft()
    if knoten==zielknoten:
      return knoten
    for nachbar in knoten.nachbarn:
      if nachbar.besucht == False:
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

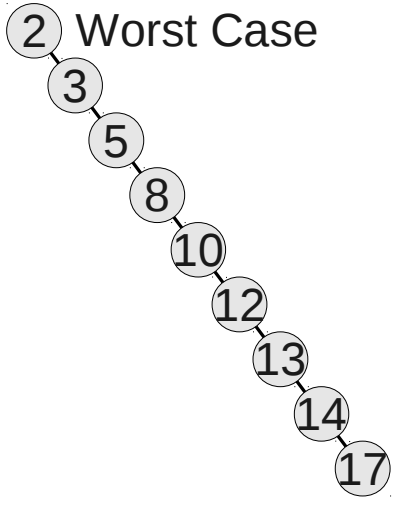
Suchbäume

- Die bisherigen Verfahren sind 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)
- Unterliegen die Elemente des Baumes einer Totalordnung, geht es mit Suchbäumen noch viel schneller
- 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 den Baum aufgenommen.
- Sie werden von der Wurzel gemäß dem Sortierkriterium bis zur entsprechenden Stelle durchgereicht.
- Die Baumstruktur wird beim Hinzufügen von Elementen nicht verändert.
- Suche läuft 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).

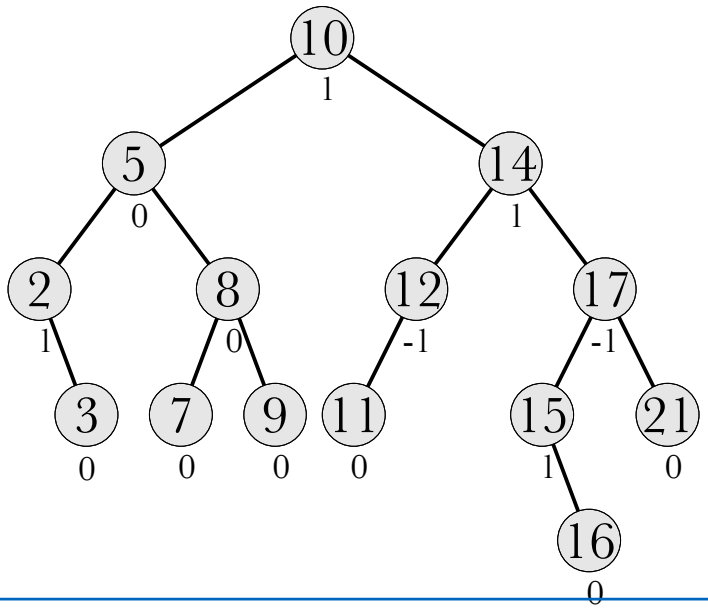




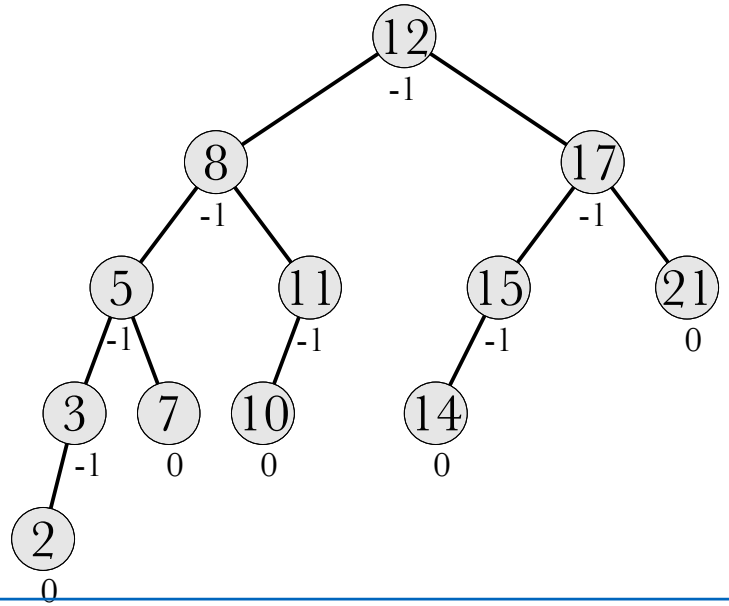
AVL-Baum

- 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.

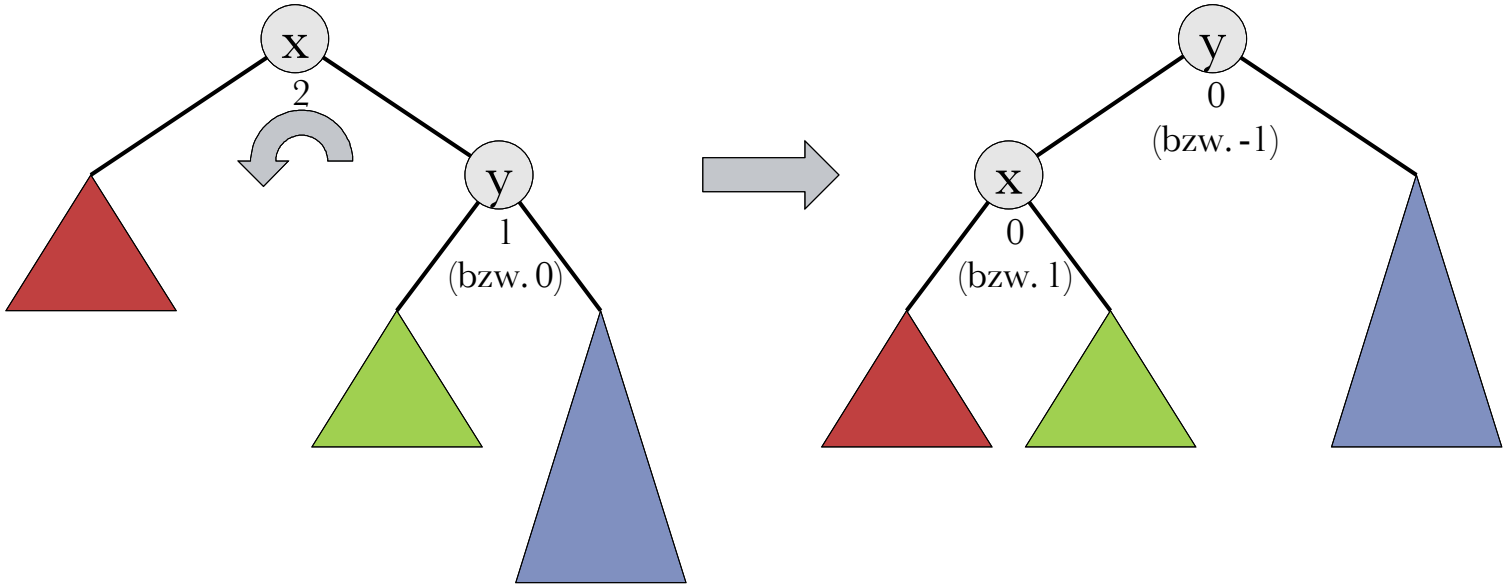
Einfügen bei Knoten mit einem Kind

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

Einfügen bei Blattknoten

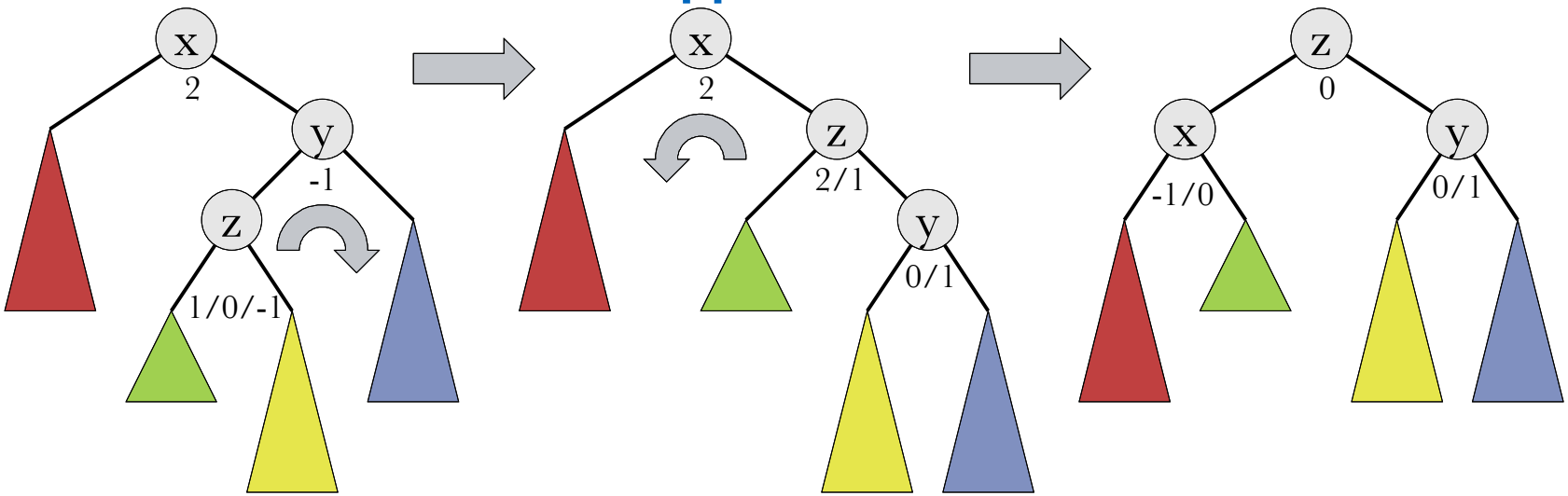
- Der bisherige Balancewert des Knotens ist 0.
- Neuer Balancewert ist $\in \{-1, 1\}$ (aktueller Teilbaum höher).
- Information muss rekursiv nach oben weitergegeben werden.
- Bei jedem Vaterknoten wird der neue Balancewert berechnet.
  - Bei Balancewert $\in \{-1, 1\}$ geht man weiter nach oben.
  - 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.

Baumbasierte KI für TicTacToe

Schnittstelle

- Bei jeden Zug wird eine Prozedur aufgerufen, bei der der KI-Spieler ein `'o'` setzen muss
- KI-Spieler erhält Liste, die das 3x3-Gitter (row-order) enthält
- Drei Zeichen möglich:
  - `'␣'`: nicht belegt
  - `'x'`: vom Mensch belegt
  - `'o'`: vom PC belegt
- KI muss ein leeres Element (`'␣'`) mit `'o'` belegen, z.B.:
  - Vorher: `['o', 'x', '␣', '␣', 'x', '␣', '␣', '␣', '␣']`
  - Nachher: `['o', 'x', '␣', '␣', 'x', '␣', '␣', 'o', '␣']`
- Keine Überprüfungen (Bin ich überhaupt dran, ...)

```
def pcSchritt(zellen):
  ...
  zellen[...] = 'o'
```

Vorgehensweise

- Aufbau eines kompletten Entscheidungsbaums (rekursiv)
- Jeder Knoten entspricht einer möglichen Spielsituation (z.B. `['o', 'x', '␣', '␣', 'x', '␣', '␣', 'o', '␣']`)
- Die Kinder des Knotens sind die möglichen Züge des PC.
- Für jeden Knoten bestimmen, welcher Spieler im kompletten Teilbaum wie oft gewinnt;
- Teilbaum mit besten Gewinnchancen auswählen und das entsprechende Feld belegen;
- Achtung: Vorgehensweise ist weder effizient noch optimal!

Aufbau des Baumes: Parameter für Konstruktor

- `self`
- `zellen`: Liste mit der Spielsituation
- `winproc`: Prozedur, die für die Spielsituation den Gewinner $\in$ {`'␣'`, `'x'`, `'o'`} ermittelt
- `player='o'`: Aktueller Spieler (für Wurzelknoten immer `'o'`)
- `zug=None`: Zug der zur aktuellen Spielsituation geführt hat (für Wurzelknoten None)

Aufbau des Baumes: member-Variablen

- `zellen`: Liste mit der Spielsituation
- `kinder`: Liste der Kinder des Baumes
- `zug`: Welcher Zug (0-8) hat zu der Spielsituation geführt
- `gewinner` Evtl. gibt es für die Situation schon einen Gewinner (`'x'` oder `'o'`), sonst `'␣'`

Konstruktor der Klasse

```
class TicTacToeBaum(object):
  def __init__(self, zellen, winproc, player='o',
               zug=0):
    self.zellen = zellen
    self.kinder = []
    self.zug = zug
    nextplayer = {'x': 'o', 'o': 'x'}
    self.gewinner = winproc(zellen)
    if self.gewinner == '␣':
      for i in range(9):
        if (zellen[i] == '␣'):
          copy = list(zellen)
          copy[i] = player
          self.kinder.append(TicTacToeBaum(copy,
                             winproc, nextplayer[player], i))
```

Gewinner für Teilbäume bestimmen

- Erneut rekursive Berechnung
- dictionary `anzGewinne = {'␣': 0, 'x': 0, 'o': 0}`
- Wenn ein Knoten keine Kinder hat, ist `anzGewinne[gewinner] = 1`
- Sonst wird die Summe der Gewinne der Kinder ermittelt

```
def zaehleGewinner(self):
    self.anzGewinne = {'␣': 0, 'x': 0, 'o': 0}
    if len(self.kinder)==0:
        self.anzGewinne[self.gewinner] += 1
    for kind in self.kinder:
        (nowin, xwin, owin) = kind.zaehleGewinner()
        self.anzGewinne['␣'] += nowin
        self.anzGewinne['x'] += xwin
        self.anzGewinne['o'] += owin
    return (self.anzGewinne['␣'], self.anzGewinne['x'],
            self.anzGewinne['o'])
```

Wahl des „optimalen“ Feldes

- Einschränkung:
  - Perfekte Strategie relativ aufwändig
  - Hier deswegen nur eine mittelmäßige (damit es auf eine Folie geht)
- Für alle Spielzüge (alle Kinder) wird die ”Gewinnchance“ bestimmt.
- Wenn in einer Spielsituation nur noch der PC-Spieler gewinnen kann (auch kein Unentschieden), ist das optimal, die Gewinnchance riesig.
- Wenn zusätzlich noch Unentschieden möglich sind, ist die Chance immer noch groß.
- Wenn beide gewinnen können, wird die Chance als Quotient der beiden Häufigkeiten gewählt.
- Wenn nur noch der Mensch gewinnen kann, entspricht die Chance der negativen Häufigkeit der menschlichen Gewinnmöglichkeiten.
- Das Kind (der Zug $\in \{0-8\}$) mit der höchsten Chance (nach obiger Heuristik) wird ausgewählt.

```
def optWahl(self):
  chance = -1e100
  wahl = self.kinder[0].zug
  for kind in self.kinder:
    nowin = float(kind.anzGewinne['␣'])
    xwin = float(kind.anzGewinne['x'])
    owin = float(kind.anzGewinne['o'])
    if owin > 0 and xwin == 0 and nowin== 0:
      neueChance = 1e100
    elif owin > 0 and xwin == 0:
      neueChance = 1e10*owin/nowin
    elif owin > 0 and xwin > 0:
      neueChance = owin/xwin
    else:
      neueChance = -xwin
    if neueChance > chance:
      chance = neueChance
      wahl = kind.zug
  return wahl
```