

## Modul: Programmierung B-PRG Grundlagen der Programmierung 1 WS 17/18

### Datenstrukturen in Python

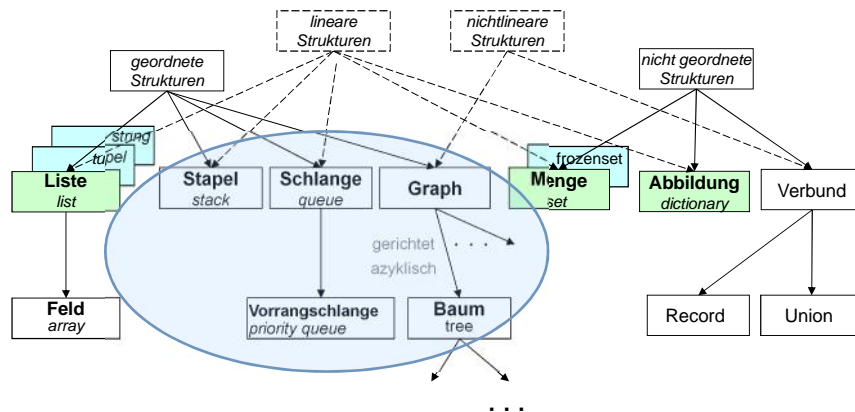
Stapel, Schlangen, Heaps (Haufen, Halden) , Graphen und Bäume

Prof. Dr. Detlef Krömker  
Professur für Graphische Datenverarbeitung  
Institut für Informatik  
Fachbereich Informatik und Mathematik (12)

## Rückblick

- Letzten Freitag:
  - **Daten – Information – Wissen**
- Schon diverse built-in Datenstrukturen besprochen

## Übersicht: Wichtige Datenstrukturen



3

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Unser heutiges Lernziele

- Strukturen und Funktionen der Datentypen
  - Stapel *stack*
  - Schlange *queue*
  - Haufen (Halde) *heap*
  - Graphen *graph*
  - Bäume *tree*
  - kennenlernen ... auch die zugehörige Terminologie.
- Typische Python Implementierung hierzu kennenlernen

4

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Übersicht

- Stapel, (Warte-)Schlangen und Halden
- Graphen
  - Kategorisierung von Graphen
  - Graphen als Datenstruktur
    - Adjazenzmatrix
    - Inzidenzmatrix
    - Adjazenzliste
  - Implementierung von Graphen in Python
- Bäume
  - Implementierung von Bäumen in Python
- Klassen zum Typen in Python 3

## Stapel (stack)

- In einem **Stapelspeicher** (auch Kellerspeicher engl. **stack**) werden Objekte gespeichert, die nur in umgekehrter Reihenfolge wieder gelesen werden können. Die gespeicherten Objekte bleiben geordnet.
- **LIFO-Prinzip** (Last In First Out).
- Zu einem Stapelspeicher gehören **zumindest** die Operationen
  - `push(e)`, um ein Objekt `e` im Stapelspeicher abzulegen und
  - `pop()`, um das zuletzt gespeicherte Objekt wieder zu lesen

oft auch:

- `is_empty()`, prüft, ob der Stapel leer ist,

## Stapel (2)

- Der **verändernde Zugriff** ist nur auf dem obersten Element des Stapels möglich.
- Gelegentlich wird auch ein Befehl zum vertauschen der beiden obersten Elemente angeboten `swap()` oder ein Befehl zum Bestimmen der Länge des Stapels `length()` oder `top()` (`peek()`) liefert das oberste Element.

### Operationen:

push („Anton“)  
push („Berta“)  
length  
push („5“)  
pop  
pop  
pop  
pop



### Ergebnis =

~~Stack~~ **Stack**

5
Berta
Anton

### Kommentar:

Leerer Stapel

5
Berta
Anton

Berta
Anton

Berta
Anton

Anton
-------

leerer Stapel

push („Anton“)

push („Berta“)

length = 2

push („5“)



pop liefert „5“

pop liefert „Berta“

pop liefert „Anton“

pop liefert exception

9
Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume
Prof. Dr. Detlef Krömer

## Implementierung eines Stacks in Python (1)

- Der built-in-Datentyp Liste list() stellt alles benötigte bereit.
- Wenn man dann doch eine DS `stack()` haben will  
→ umbenennen der Methoden
- Wegen ihrer Bedeutung werden stacks häufig schon durch die **Hardware des Prozessors** unterstützt.

10
Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume
Prof. Dr. Detlef Krömer

## Implementierung eines Stacks in Python (2)

(wenn man unbedingt eine Datenstruktur Stack haben will ;-)

```
class Stack:
    def __init__(self):
        self.items = [] # implemented with a list

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def top(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

11

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Nutzung von Stacks

Wichtiges Element zur Realisierung der Unterprogrammtechnik:

- ▶ *Jump To Subroutine* legt auf dem Stack die **Rücksprungsadresse** ab, die später von der Operation *Return From Subroutine* verwendet wird.
- ▶ die **Parameter** von Unterprogrammen werden über den Stack übergeben, der auch die **Rückgabewerte** aufnimmt.
- ▶ **Lokale Variablen** werden auf dem Stack gespeichert. Dies erlaubt unter anderem Rekursion, das Aufrufen einer Funktion aus eben dieser Funktion heraus.
- ▶ Viele Compiler und Interpreter nutzen einen Stack zur Übersetzung des Quellcodes, sehr häufig für den Parser.
- ▶ **Verarbeitung von Klammerstrukturen** in Ausdrücken.

12

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## (Warte-)Schlange (queue)

- ▶ In einer Warteschlange (engl. queue) kann eine beliebige Anzahl von Objekten gespeichert werden, jedoch können die gespeicherten Objekte nur in der gleichen Reihenfolge wieder gelesen werden, wie sie gespeichert wurden.
- ▶ **FIFO-Prinzip** (First In First Out).

Anton

dequeue ( )

- ▶ Zu einer Queue gehören zumindest die Operationen:
  - ▶ *enqueue*, um ein Objekt in der Warteschlange zu speichern und
  - ▶ *dequeue*, um das zuerst gespeicherte Objekt wieder zu lesen und aus der Warteschlange zu entfernen.

13

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Schlange (2) - Ringpuffer

- ▶ Eine Warteschlange wird gewöhnlich als Liste implementiert, kann aber auch ein Vektor sein.
- ▶ Eine spezielle Implementierung ist eine als **Ringpuffer**.
  - ▶ Besonderheit: **feste Größe**
  - ▶ wenn der Puffer voll ist, werden die ältesten Inhalte überschrieben.
- ▶ Queues und Ringpuffer können in Python sehr leicht mithilfe einer Liste implementiert werden.



Bildquelle: wikipedia

14

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Nutzung von Warteschlangen

- Die sogenannte Pipe zur Interprozesskommunikation
- Langsame externe Geräte, z. B. Drucker werden durch Warteschlangen von der Programmabarbeitung entkoppelt.
- zur Datenübergabe zwischen asynchronen Prozessen in verteilten Systemen verwendet, wenn also Daten vor ihrer Weiterverarbeitung gepuffert werden müssen.
- GUIs puffern Maus- und Tastaturereignisse in einer sogenannten „Message Queue“, d. h. in der Reihenfolge ihres Auftretens. Anschließend leiten sie diese dann, abhängig von Position und Eingabefokus, an die korrekten Prozesse weiter.

## Eine Erweiterung: Vorrang(warte-)schlange (*priority queue*) Haufen (Halde, *heap*)

- absolut kein "Schrott" ;-)
- Heaps dienen damit der **Speicherung von Mengen**.
- Den Elementen ist ein **Schlüssel** zugeordnet, der die Priorität der Elemente festlegt.
- Auf den Schlüsseln ist eine **totale Ordnung** definiert.



Quelle: <https://commons.wikimedia.org/w/index.php?curid=1250566>  
Mohylek als Autor angenommen  
(CC BY-SA 3.0)

Die wichtigsten Operationen sind:

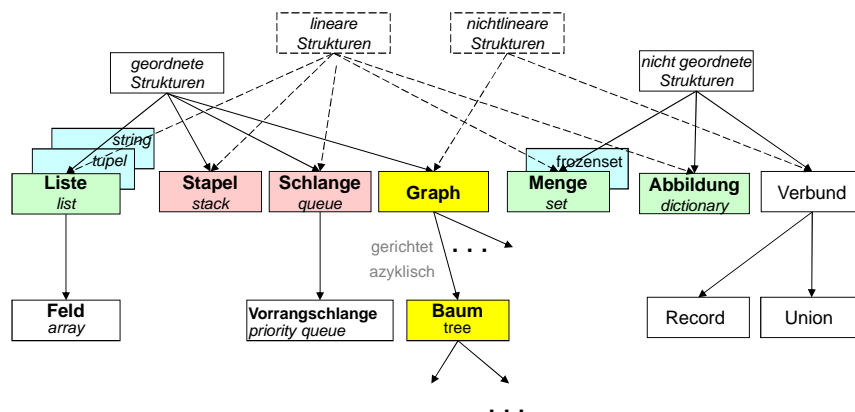
*insert* zum Einfügen eines Elementes,  
*remove* zum Entfernen eines Elementes  
*extractMin* zur Rückgabe und dem Entfernen eines Elementes mit minimalem Schlüssel (=höchster Priorität).



## Nutzung von Heaps

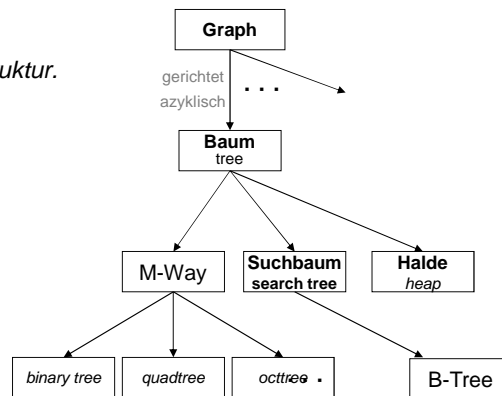
- ▶ In **Vorrangwarteschlangen**, wie sie bei Servern oder Betriebssystemen zur Festlegung der Ausführungsreihenfolge von Aufgaben benötigt wird.
- ▶ Heaps sind "ideale" Datenstrukturen für **Greedy-Algorithmen** dar, die schrittweise lokale optimierte Entscheidungen treffen.
- ▶ ... schon etwas schwieriger. In Python gibt es einen Built-in Modul:
- ▶ **heapq** — Heap queue algorithm  
bitte dort einmal nachlesen:  
<https://docs.python.org/3.6/library/heapq.html#module-heapq>

## Übersicht: Wichtige Datenstrukturen



## Jetzt

Die „Brot und Butter“ Datenstruktur.

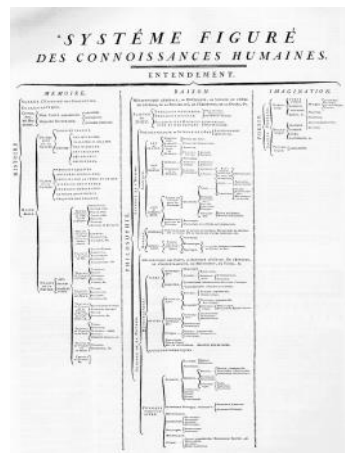


## Nicht wirklich eine „Erfindung“ der Informatik

### „Stammbaum des Wissens“

Band 1 der *Encyclopédie ou Dictionnaire raisonné des sciences, des arts et des métiers* (28 Bände, vollendet im Jahr 1772)

Jean Baptiste le Rond d'Alembert und Denis Diderot

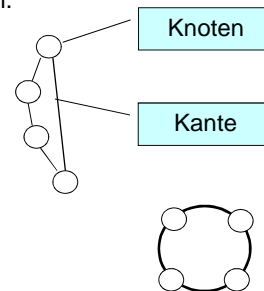


## Graph – Definition (Graphentheorie)

Ein Graph  $G$  ist ein geordnetes Paar zweier Mengen:  $G = (V, E)$

Dabei bezeichnet  $V$  die Menge der im Graph enthaltenen **Knoten** (*Vertex*)  
und  $E$  die Menge der **Kanten** (*Edge*) des Graphen.

Anschaulich ist ein Graph ein Gebilde aus  
Knoten (auch Ecken oder Punkte),  
die durch Kanten verbunden sein können.



**Achtung:** verschiedene Bilder  
können denselben Graphen darstellen.

## Kategorisierung von Graphen

**ungerichtete Graphen ohne**

**Mehrfachkanten:**

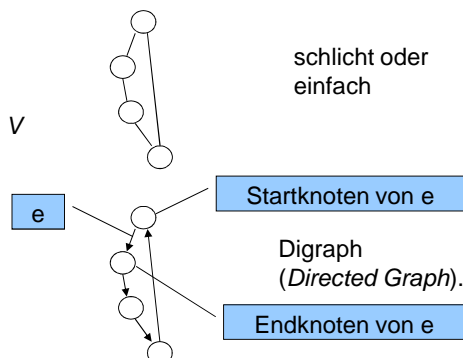
$E$  ist eine Teilmenge aller  
2-elementigen Teilmengen von  $V$

schlicht oder  
einfach

**gerichtete Graphen ohne**

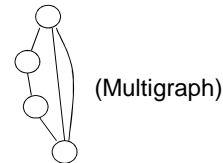
**Mehrfachkanten:**

$E$  ist eine Teilmenge des  
kartesischen Produktes  $V \times V$

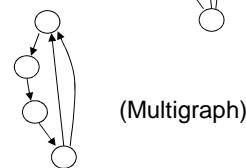


## (Kategorisierung von Graphen (Eigenschaften von E))

**ungerichteten Graphen mit Mehrfachkanten:** E ist eine Multimenge [Elemente können mehrfach vorkommen] über der Menge aller 2-elementigen Teilmengen von V



**gerichteten Graphen mit Mehrfachkanten:** E ist eine Multimenge über dem kartesischen Produkt  $V \times V$



**Hypergraphen** eine Teilmenge der **Potenzmenge** von V.

**Potenzmenge von V:** ist die Menge, die aus allen Teilmengen von V besteht. Die Potenzmenge ist also eine Menge, deren Elemente selbst Mengen sind.

## Graphen als Datenstruktur

Ein Graph als Datentyp sollte mindestens die folgenden Operationen haben

- *Einfügen* (Kante, Knoten)
- *Löschen* (Kante, Knoten)
- *Finden* eines Objekts (Kante, Knoten), **Wichtig auch:**
- *Adjazenztest*: Sind zwei Knoten durch eine Kante verbunden?
- *Bestimmung inzidenter Kanten*: Welche Kanten berühren einen gegebenen Knoten?

Die bekanntesten (klassischen) Repräsentation von Graphen sind:

- die **Adjazenzmatrix** (Nachbarschaftsmatrix)
- die **Adjazenzliste** (Nachbarschaftsliste)
- die Inzidenzmatrix (Knoten-Kanten-Matrix, seltener genutzt)

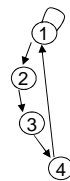
## Adjazenzmatrix

**Adjazenz:** (Aneinandergrenzen oder Berühren)

Ein Graph mit  $n$  Knoten kann durch eine  **$n \times n$ -Matrix** repräsentiert werden.

Dazu nummeriert man die Knoten von 1 bis  $n$  durch und markiert in die Matrix die Beziehungen der Knoten zueinander ein.

	1	2	3	4
1	x	x		
2			x	
3				x
4	x			



Hypergraphen lassen sich **nicht** durch eine Adjazenzmatrix darstellen.

## Adjazenzliste

Die Adjazenzliste wird in ihrer einfachsten Form durch eine Liste aller Knoten des Graphen dargestellt, wobei

- jeder Knoten eine Liste aller seiner Nachbarn (in ungerichteten Graphen) bzw.
- Nachfolger in gerichteten Graphen besitzt.

Vielfachheiten der Kanten, Knotengewichte, und Kantengewichte werden meist in Attributen der einzelnen Elemente gespeichert.

## Adjazenzliste (2)

- **In der Praxis verwendet man meist diese Form, obwohl**
- aufwändiger zu implementieren und zu verwalten, bieten aber eine Reihe von Vorteilen gegenüber Adjazenzmatrizen.
- verbrauchen nur linear viel Speicherplatz, was insbesondere bei dünnen Graphen (also Graphen mit wenig Kanten) von Vorteil ist, während die Adjazenzmatrix quadratischen Platzbedarf bezüglich der Anzahl Knoten besitzt (dafür aber kompakter bei dichten Graphen, also Graphen mit vielen Kanten ist).
- Zum anderen lassen sich viele graphentheoretische Probleme nur mit Adjazenzlisten in linearer Zeit lösen.

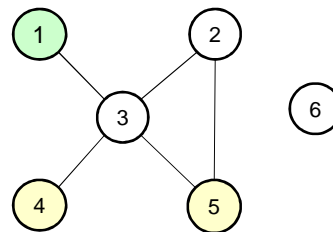
## Inzidenzmatrix (selten genutzt)

- Ein Graph mit  $n$  Knoten und  $m$  Kanten kann auch durch eine  **$n \times m$ -Matrix** repräsentiert werden. Dazu nummeriert man die Knoten von 1 bis  $n$  und die Kanten von 1 bis  $m$  durch und trägt in die Matrix die Beziehungen der Knoten zu den Kanten ein.
- Jede Spalte der Inzidenzmatrix enthält genau zwei von Null verschiedene Einträge.
- In ungerichteten Graphen zweimal die 1 und
- in gerichteten Graphen einmal die 1 (Endknoten) und einmal die -1 (Startknoten).

## Implementierungen in Python

Graphen lassen sich in Python sehr einfach durch **Dictionaries** realisieren.

```
graph = {1 : [3],
        2 : [3, 5],
        3 : [1, 2, 4, 5],
        4 : [3],
        5 : [2, 3],
        6 : []
}
```



Also: Die Schlüssel des Dictionaries entsprechen den Knoten des Graphen.

Die Werte sind Listen mit den Knoten, die mit dem (Schlüssel-)Knoten durch eine Kante verbunden sind.

## Operationen auf dieser Struktur (1)

```
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append((node, neighbour))
    return edges

print(generate_edges(graph))
```

Wenn man die Funktion `generate_edges(graph)` auf dem vorher definierten Dictionary `graph` anwendet erhält man:

```
[(1, 3), (2, 3), (2, 5), (3, 1), (3, 2), (3, 4), (3, 5),
 (4, 3), (5, 2), (5, 3)]
```

## Operationen auf dieser Struktur (2)

```
def find_isolated_nodes(graph):
    """ returns a list of isolated nodes. """
    isolated = []
    for node in graph:
        if not graph[node]:
            isolated.append(node)
    return isolated
```

Wenn man die Funktion `find_isolated_nodes(graph)` auf dem vorher definierten Dictionary `graph` anwendet erhält man:

[6]

## Baum

Als **Wald** bezeichnet man in der Graphentheorie einen ungerichteten Graphen ohne Kreis (Zyklus). Ist dieser **zusammenhängend**, so spricht man von einem (**ungerichteten**) **Baum**.

Die Konstituenten eines Waldes sind Bäume, so dass ein Wald aus einem oder mehreren Bäumen besteht.

Jeder Baum ist also auch ein Wald.



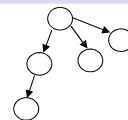
## Gewurzelte Bäume

Neben **ungerichteten Bäumen** betrachtet man auch **gerichtete Bäume**, die häufig auch als **gewurzelte Bäume** bezeichnet werden und sich weiter in **In-Trees** und **Out-Trees** unterscheiden lassen.

Bei gerichteten Bäumen gibt einen ausgezeichneten Knoten, den man **Wurzel** nennt und für den die Eigenschaft gilt,

- dass alle Kanten von diesem wegzeigen (**Out-Tree**) oder
- zu diesem hinzeigen (In-Tree).

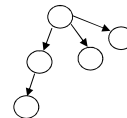
## Eigenschaften (1)



- **Mit Bäumen wird die lineare Struktur der Liste wird aufgebrochen.**
- Der maximale **Ausgangsgrad** wird als **Ordnung** des Baumes bezeichnet.
- Alle Knoten mit Ausgangsgrad 0 bezeichnet man als **Blätter**.
- Alle Knoten, die kein Blatt sind, als **innere Knoten**.
- Als **Tiefe** eines Knotens bezeichnet man die Länge (Anzahl der Kanten) des Pfades von der Wurzel zu ihm und
- als **Höhe** des Baumes die Länge eines längsten Pfades.
- Viele weitere Bezeichnungen sind der Genealogie entlehnt:

## Eigenschaften (2)

Für einen von der Wurzel verschiedenen Knoten  $v$  bezeichnet man den Knoten, durch den er mit einer **eingehenden** Kante verbunden ist als **Vater**, Vaterknoten, Elternknoten oder Vorgänger von  $v$ .



Als **Vorfahren** von  $v$  bezeichnet man alle Knoten, die entweder Vater von  $v$  oder Vater des Vaters, usw. sind.

Umgekehrt bezeichnet man alle Knoten, die von einem beliebigen Knoten  $v$  aus durch eine ausgehende Kante verbunden sind als **Kinder**, Kinderknoten, Sohn oder Nachfolger von  $v$ . Als **Nachfahren** von  $v$  bezeichnet man Kinder von  $v$  oder deren Nachfahren.

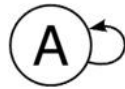
Als **Geschwister** oder **Geschwisterknoten** werden die Knoten bezeichnet, die den gleichen Vater besitzen.

35

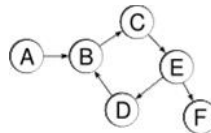
Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

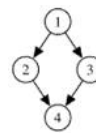
## Baum oder Graph?



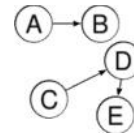
Ein Objekt, das auf sich selbst verweist, ist **kein** Baum.



Dieses Gebilde ist **kein** Baum, da B zwei Eltern hat.



Dieses Gebilde ist **kein** Baum, da 4 zwei Eltern hat.



Zwei unverbundene Bäume ergeben zusammen **keinen** Baum

36

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Spezielle Bäume

- ▶ Bei **Binärbäumen** ist die Anzahl der Kinder höchstens zwei.
- ▶ In **balancierten Bäumen** gilt zusätzlich, dass sich die Höhen des linken und rechten Teilbaums an jedem Knoten höchstens um eins unterscheiden.
- ▶ Bei **Suchbäumen** sind die Elemente in der Baumstruktur geordnet abgelegt, so dass man schnell Elemente im Baum finden kann.

## Partiell geordneter Baum

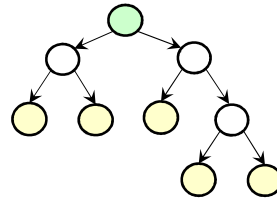
Ein **partiell geordneter Baum**  $T$  ist ein Baum,

- ▶ dessen Knoten markiert sind
- ▶ dessen Markierungen aus einem geordneten Wertebereich stammen
- ▶ in dem für jeden Teilbaum  $T'$  mit der Wurzel  $x$  gilt: Alle Knoten aus  $T'$  sind größer markiert als  $x$  oder gleich  $x$ .

**Intuitiv** bedeutet dies: Die Wurzel jedes Teilbaumes stellt ein Minimum für diesen Teilbaum dar. Die Werte des Teilbaumes nehmen in Richtung der Blätter zu oder bleiben gleich.

## Geordnete, strikte und vollständige Binärbäume

- Ein **Binärbaum** heißt **geordnet**, wenn jeder innere Knoten ein linkes und eventuell zusätzlich ein rechtes Kind besitzt (und nicht etwa nur ein rechtes Kind).
- Man bezeichnet ihn als **voll** oder **strikt**, wenn jeder Knoten entweder Blatt ist (also kein Kind besitzt), oder aber zwei (also sowohl ein linkes wie ein rechtes) Kinder besitzt.
- Man bezeichnet ihn als **vollständig**, wenn alle Blätter die gleiche Tiefe besitzen. Ein vollständiger Binärbaum der Höhe  $n$ ,  $n$  man häufig auch als  $B_n$  bezeichnet, hat genau
  - $2^{n+1}-1$  Knoten,
  - $2n-1$  innere Knoten,
  - $2^i$  Knoten in Tiefe  $i$ , insbesondere also
  - $2n$  Blätter
 mit Höhe  $n$  die Länge des Pfades zu einem tiefsten Knoten bezeichnet wird.



39

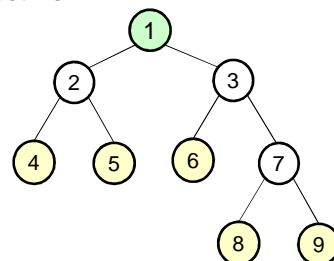
Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

## Eine wichtige Operation auf Bäumen: Traversierung

Es gibt verschiedene Möglichkeiten, die Knoten von Binärbäumen zu durchlaufen. Diesen Prozess bezeichnet man auch als **Linearisierung oder Traversierung**. Man unterscheidet hier in:

- pre-order (W-L-R)**: wobei zuerst die Wurzel (W) betrachtet wird und anschließend zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen wird.
- Beispiel: 1, 2, 4, 5, 3, 6, 7, 8, 9
- Man ersetzt bei Bäumen gern den Richtungspfeil weil in der Regel die Richtung klar ist.



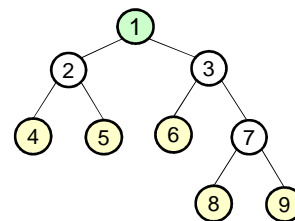
40

Vorlesung PRG 1  
Stapel, Schlangen, Graphen und Bäume

Prof. Dr. Detlef Krömer

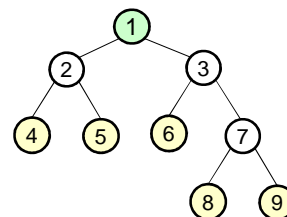
## Traversierung - in-order

- **in-order (L–W–R)**: wobei zuerst der linke (L) Teilbaum durchlaufen wird, dann die Wurzel (W) betrachtet wird und anschließend der rechte (R) Teilbaum durchlaufen wird und
- Beispiel: 4, 2, 5, 1, 6, 3, 8, 7, 9



## Traversierung post-order

- **post-order (L–R–W)**: wobei zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen wird und anschließend die Wurzel (W) betrachtet wird.
- Beispiel: 4, 5, 2, 6, 8, 9, 7, 3, 1



## Rekursive Implementierungen der Baum-Traversierung

```

Funktion Preorder (Baum)
W <- Baum.Wurzel           //W:= Wurzel des übergebenen Baumes
If Baum.Links <> NULL       //Existiert ein linker Unterbaum?
    L <- Preorder(Baum.Links) // dann: L:= Preorder von linkem Unterbaum
If Baum.Rechts <> NULL      //Existiert ein rechter Unterbaum?
    R <- Preorder(Baum.Rechts) // dann: R:= Preorder von rechtem Unterbaum
Return W*L*R               //Rückgabe: Verkettung aus W, L und R

Funktion Inorder (Baum)
W <- Baum.Wurzel           //W:= Wurzel des übergebenen Baumes
If Baum.Links <> NULL       //Existiert ein linker Unterbaum?
    L <- Inorder(Baum.Links) // dann: L:= Inorder von linkem Unterbaum
If Baum.Rechts <> NULL      //Existiert ein rechter Unterbaum?
    R <- Inorder(Baum.Rechts) // dann: R:= Inorder von rechtem Unterbaum
Return L*W*R               //Rückgabe: Verkettung aus L, W und R

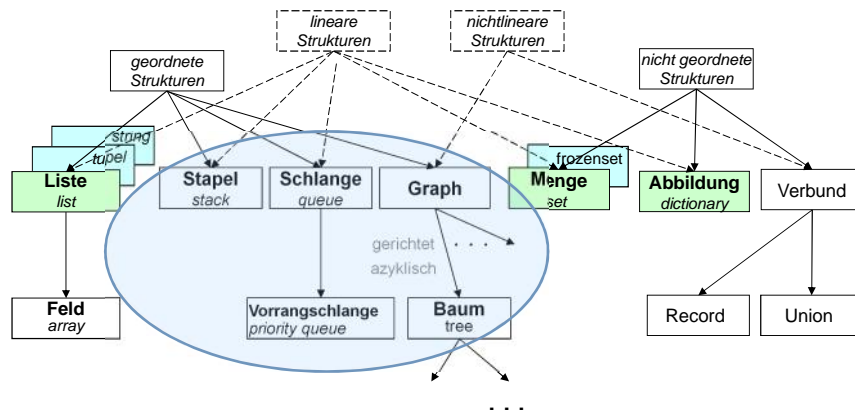
Funktion Postorder (Baum)
W <- Baum.Wurzel           //W:= Wurzel des übergebenen Baumes
If Baum.Links <> NULL       //Existiert ein linker Unterbaum?
    L <- Postorder(Baum.Links) // dann: L:= Postorder von linkem Unterbaum
If Baum.Rechts <> NULL      //Existiert ein rechter Unterbaum?
    R <- Postorder(Baum.Rechts) // dann: R:= Postorder von rechtem Unterbaum
Return L*R*W               //Rückgabe: Verkettung aus L, R und W
    
```

## Man unterscheidet weiter

- in **binäre Suchbäume** mit (**AVL-Bäumen** als balancierte Version) und
- B-Bäumen sowie diversen Varianten, den **B\*-Bäumen** (die Blattknoten in einer Liste miteinander verkettet). (Spezielle Suchbäume in Datenbanksystemen Achtung: B steht **nicht** für binär!)
- In B\*-Bäumen wird neben der effizienten Suche einzelner Datenelemente auch das schnelle sequenzielle Durchlaufen aller Datenelemente unterstützt).

Eine detaillierte Diskussion würde in dieser Veranstaltung zu weit führen, aber Sie werden in Ihrem Studium noch viel davon hören, versprochen ;-)  
;-) schon nächstes Semester in DS!

## Übersicht: Wichtige Datenstrukturen



## Ein Hinweis für die Implementierung eines binären Baums in Python

```

class Node:
    def __init__(self, data=None):
        self.data = data
        self.left = None
        self.right = None

    def __str__(self):
        return "[%s, %i, %i]" % (str(self.data),
                                id(self.left), id(self.right))
    
```

## Zusammenfassung

- Stapel, Schlangen, Graphen und Bäume
- ganz wichtige Datenstrukturen
- Sie wissen, was das ist und wissen grob, wie es zu implementieren ist
- es fehlen aber noch viele weitergehende Aspekte, → Datenstrukturen im 2. Semester, ... Oder Sie haben das schon gemacht!

## Fragen und (hoffentlich) Antworten



## Ausblick ... nächsten Freitag

V 23 Services des Betriebssystems  
+ wichtige Bibliotheken in Python

**... und, danke für Ihre Aufmerksamkeit!**