Aufgabe 1

Ein abstrakter Datentyp (ADT) ist eine Menge von Objekten und eine Beschreibung der darauf zulässigen Operationen. *Beispiele:* (Pseudocode)

- Objekt: Array `A`

  Operationen: `A[i]`, `A.length()`, `A.sort()`, ...
- Objekt: Stack `S`

  Operationen: `S.push(item)`, `S.pop()`, `S.is_empty()`, ...
- ...

Aufgabe 2

Eine Array ist eine Datensruktur, die eine Sammlung von Elementen des gleichen Typs in einer festen Reihenfolge speichert.

Der Zugriff auf bestimmte Inhalte des Arrays erfolgt mit Hilfe von Indizes (`A[0]`, `A[1]`, ...) in $O(1)$.

Anwendungen:

- Verarbeiten von Zahlen, Zeichen oder Pixeln eines Bildes
- als Grundlage für andere Datenstrukturen

Aufgabe 3

Hier eine Auswahl wünschenswerter Operationen auf Arrays

- Ein Element im Array finden
- Die Häufigkeit eines Elements im Array bestimmen
- Den Index eines Elements in einem Array ermitteln
- Ein Array sortieren
- Die Reinehfolge der Elemente eines Array umkehren
- Zwei Arrays zu einem neuen Array verbinden („addieren“)
- ...

Aufgabe 4

(a) Funktionsprinzip von Stacks: *LIFO – Last in First out*

(b) Standardoperationen: (`S` steht hier für einen Stack)

- `S.push(item)`: Element `item` auf Stack ablegen.
- `S.pop()`: Entfernt oberstes Element von `S` und gibt es zurück.
- `S.is_empty()`: Gibt `True` zurück, wenn der Stack leer ist.
- `S.size()`: Gibt Anzahl der Elemente von Stack `S` zurück.
- `S.peek()`: Gibt oberstes Element von `S` als Wert zurück.

(c) Nenne zwei konkrete Anwendunge für Stacks.

- Undo-Funktion in Anwendungsprogrammen
- Prüfen, ob ein Klammerterm korrekt ist
- History-Funktion in Browsern
- ...

Aufgabe 5

```
class Stack:

    def __init__(self):
        self.items = []

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

    def pop(self):
        return self.items.pop() # entfernt letztes Element

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

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

Aufgabe 6

```
S = Stack()  # => []
S.push(3)    # => [3]
S.push(5)    # => [3, 5]
x = S.pop()  # => [3] => x=5
S.push(1)    # => [3, 1]
S.push(x)    # => [3, 1, 5]
S.push(4)    # => [3, 1, 5, 4]
```

Aufgabe 7

```
from stack import Stack
def check(string):
    S = Stack()      # Erzeuge leeren Stack
    for c in string: # Zeichen c durchläuft string
        if c == '(':   # ist c öffnende Klammer?
            S.push(c)  # ja: auf den Stack damit
        elif c == ')': # nein: ...
            if S.is_empty(): # falls S leer ist ...
                return False  # .. gibt es kein '(' zu ')'
            else:
                S.pop() # sonst entferne '(' vom Stack
        else: # bei allen anderen Zeichen c
            pass  # überspringe es
    if S.is_empty(): # ist der Stack am Ende leer,
        return True  # dann stimmt die Klammerung
    return False # wenn nicht, gibt es zu viele '('
```

Aufgabe 8

(a) Funktionsprinzip von Warteschlangen: *FIFO – First in First out*

(b) Standardoperationen: (`Q` steht hier für eine Queue)

- `Q.enqueue(item)`: Setze `item` ans Ende der Queue.
- `Q.dequeue()`: Entfernt ältestes Element und gibt es zurück.
- `Q.is_empty()`: Gibt `True` zurück, wenn die Queue leer ist.
- `S.size()`: Gibt Anzahl der Elemente in der Queue zurück.

(c) Anwendungen für Queues

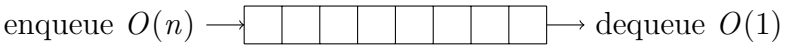
- Betriebssysteme verwenden Warteschlangen für Prozesse, die darauf warten, von der CPU ausgeführt zu werden.
- Netzwerke reihen empfangene Segmente in eine Warteschlange ein und verarbeiten sie weiter, wenn sie an der Reihe sind.
- Graphen- und Baumalgorithmen (Breitensuche, Dijkstra, ...) verwenden Warteschlangen, um Knoten in der Reihenfolge zu verarbeiten, in der sie entdeckt werden.
- Simulationsmodelle setzen Warteschlangen ein, um das Eintreffen von Kunden oder Fahrzeugen zu modellieren.
- ...

Aufgabe 9

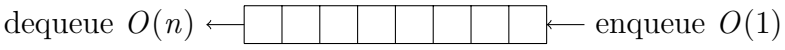
```
class Queue:

    def __init__(self):
        self.items = []

    def enqueue(self, item):
        '''Fügt ein Element am "Anfang" der Queue ein.'''
        self.items.insert(0, item)

    def dequeue(self):
        '''Entferne Element am "Ende" der Queue ...'''
        return self.items.pop() # und gib es zurück

    def is_empty(self):
        '''Return True, wenn Queue leer; sonst False.'''
        return self.items == []

    def size(self):
        '''Gib Anzahl der Element der Queue zurück.'''
        return len(self.items)
```

Die schlechte Nachricht: Fügt man ein Element am Anfang einer Liste ein, müssen die übrigen Elemente alle um eine Position verschoben werden, was Kosten von $O(n)$ bedeutet, wenn die Liste n Elemente hat.

Die gute Nachricht: Das Entfernen von Elementen am Ende einer Python-Liste der Länge n ist „ungefähr“ in $O(1)$ möglich.



Die Umkehrung der Einfügerichtigung löst das Problem nicht. Dann ist das Einfügen von Elementen effizient; nicht aber die Entnahme, da auch das Entfernen eines Elements am Anfang der Liste intern eine Verschiebung der übrigen Elemente bewirkt.



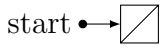
Daher ist zur Implementierung einer Queue eine *verkettete Liste* (*linked list*) besser geeignet als eine Liste oder ein Array.

Aufgabe 10

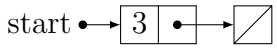
```
Q = Queue()      # -> [] ->
Q.enqueue(5)     # -> [5] ->
Q.enqueue(1)     # -> [1, 5] ->
Q.enqueue(4)     # -> [4, 1, 5] ->
x = Q.dequeue() # -> [4, 1] -> x=5
Q.enqueue(3)     # -> [3, 4, 1]
Q.enqueue(x)     # -> [5, 3, 4, 1]
y = Q.dequeue() # -> [5, 3, 4] -> y=1
```

Aufgabe 11

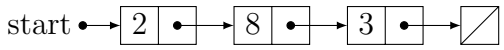
Neue EVL: Referenz (•) zeigt auf einen leeren Knoten (⊘)



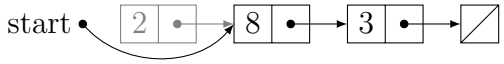
Füge `3` zur EVL hinzu:



Füge zuerst `8` und dann `2` zur EVL hinzu:

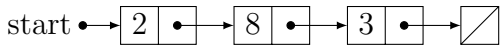


Entferne erstes Element aus EVL



(1) Erstes Element in temporärer Variable speichern (2) Ersten Knoten kurzschliessen (3) Wert der temporären Variablen zurückgeben

Durchlaufe die EVL:



Hier stellen wir uns den leeren Knoten als leere Liste vor. Einen Knoten mit Daten ist eine Liste mit zwei Elementen, wobei das erste Element den Wert und das zweite Element die Referenz auf den nächsten Knoten enthält. So erhalten wir auch eine einfache Implementierung von EVL in Python.

Initialisierung: `cursor` ← `start` (`cursor` zeigt auf ersten Knoten)
So lange `cursor` nicht auf den leeren Knoten `[]` zeigt ...
    Hole den Wert `cursor[0]` aus dem Knoten und verarbeite ihn
    `cursor` ← `cursor[1]` `cursor` zeigt auf den nächsten Knoten

Aufgabe 12

Anwendungen von verketteten Listen:

- *Dynamische Speicherverwaltung:* Einfach verkettete Listen erlauben eine dynamische Speicherwaltung in Situationen, in denen die Grösse der Datenstruktur zur Laufzeit unbekannt ist.
- *Implementierung von Stacks:* Mit einer einfach verketteten Liste kann ein Stack implementiert werden.
- *Implementierung von Warteschlangen (Queues):* Fügt man zu einer einfach verkettete Liste noch eine Referenz auf das letzte Element hinzu, lassen sich damit effizient Elemente sowohl am Anfang hinzufügen als auch vom Ende entfernen.
- *Textverarbeitung:* In Texteditoren können verkettete Listen verwendet werden, um Zeichen oder Wörter zu speichern, die dynamisch hinzugefügt oder entfernt werden.

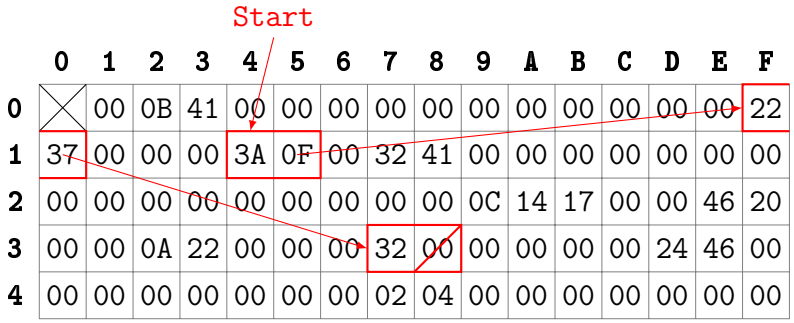
Aufgabe 13

```
class Linkedlist:
    '''Klasse für einfach verkettete Listen (EVL)'''

    def __init__(self):
        self.start = []

    def insert(self, item):
        '''Füge item am Anfang der EVL ein.'''
        self.start = [item, self.start]

    def is_empty(self):
        '''Gibt True zurück wenn EVL leer, False sonst'''
        return self.start == []

    def remove(self):
        '''Entfernt 1. Knoten und gib den Wert zurück'''
        '''(Ist die EVL leer, gib None zurück)'''
        if self.start == []: # Leere EVL handeln ...
            return None
        tmp = self.start[0] # Wert sichern
        self.start = self.start[1] # Start -> nächster Knoten
        return tmp # Wert zurückgeben
```

Aufgabe 14

```
# o in [value, o-]-> symbolisiert die Referenz
# auf den nächsten Knoten

EVL = Linkedlist()  # start -> []
EVL.insert(7)       # start -> [7, o-]->[]
EVL.insert(3)       # start -> [3, o-]->[7, o-]->[]
x = EVL.remove()    # start -> [7, o-]->[]
EVL.insert(2)       # start -> [2, o-]->[7, o-]->[]
print(x)            # => 3
```

Aufgabe 15



Der Node bei `0x14` enthält den Wert `0x3A` und die Adresse `0x0F`.

Der Node bei `0x0F` enthält den Wert `0x22` und die Adresse `0x37`.

Der Node bei `0x37` enthält den Wert `0x32` und die Adresse `0x00`.

Die Adresse `0x00` kennzeichnet das Listenende.

Somit sind in der Liste die Werte `0x3A`, `0x22` und `0x32` gespeichert.

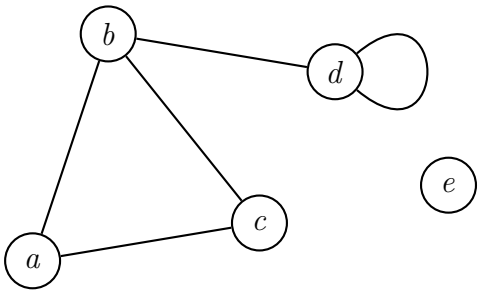
Aufgabe 16

Ein ungerichteter Graph ist ein Paar (V, E) bestehend aus einer Menge V von Knoten und einer Menge E von ungerichteten Kanten $\{v, w\}$ mit $v, w \in V$.

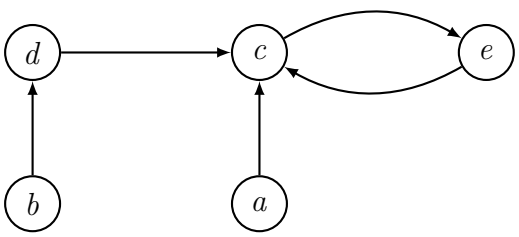
Aufgabe 17

Ein gerichteter Graph ist ein Paar (V, E) bestehend aus einer Menge V von Knoten und einer Menge E von gerichteten Kanten (v, w) mit $v, w \in V$.

Aufgabe 18



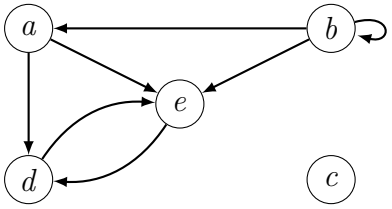
Aufgabe 19



Aufgabe 20

- *Modellierung von Verkehrsnetzen:* Orte und Kreuzungen bilden die Knoten und Verkehrswege die verbindenden Kanten. Haben die Kanten Gewichte, so können mit geeigneten Algorithmen kürzeste Wege berechnet werden.
- *Soziale Netzwerke:* Haben zwei Benutzer (Knoten) eine Beziehung, so wird dies durch eine Kante dargestellt.
- *Computernetzwerke:* Die am Netzwerk angeschlossenen Geräte stellen die Knoten dar während die Verbindungen zwischen diesen Geraden die Kanten sind.
- *World Wide Web:* Die durch Hyperlinks (Kanten) verbundenen Websiten (Knoten) bilden einen riesigen gerichteten Graphen.
- *Produktionsplanung:* Das Projektmanagement verwendet gerichtete Graphen, um Aufgaben (Knoten) und deren Abhängigkeiten (Kanten) darzustellen. Die gerichtete Kante zeigt an, dass die Aufgabe im Ausgangsknoten erledigt sein muss, bevor mit der Aufgabe im Zielknoten begonnen wird.

Aufgabe 21



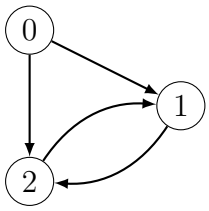
(a) Adjazenzmatrix

	a	b	c	d	e
a	0	0	0	1	1
b	1	1	0	0	1
c	0	0	0	0	0
d	0	0	0	0	1
e	0	0	0	1	0

(b) Adjazenzliste

von	nach
a	d, e
b	a, b, e
c	–
d	e
e	d

Aufgabe 22



(a) Lösung mit Adjazenzmatrix

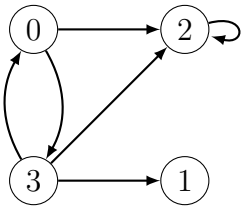
```
G = [[0, 1, 1], # Kanten 0 -> 1 und 0 -> 2
     [0, 0, 1], # Kante 1 -> 2
     [0, 1, 0]] # Kante 2 -> 1
```

(b) Lösung mit Adjazenzlisten (Liste von Nachbarschaftslisten)

```
G = [[1, 2], [2], [1]]

# G[0] -> [1, 2] (Nachbarknoten von 0)
# G[1] -> [2]    (Nachbarknoten von 1)
# G[2] -> [1]    (Nachbarknoten von 2)
```

Aufgabe 23

```
G = [[2, 3], [], [2], [0, 1, 2]]
```



Adjazenlistendarstellung:

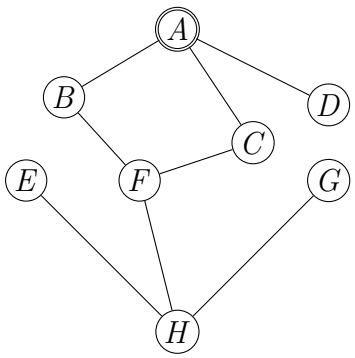
```
G[0] = [2, 3]
G[1] = []
G[2] = [2]
G[3] = [0, 1, 2]
```

Wie müsste die Adjazenmatrix dieses Graphen als Python-Liste aussehen?

Aufgabe 24

Die Tiefensuche *Depth-First-Search* (DFS) ist eine Methode, um einen Graphen zu *traversieren*, d. h. von einem Startknoten aus jeden erreichbaren Knoten zu besuchen.

```
Funktion dfs(graph, v_start):
    Markiere alle Knoten von graph als unbesucht
    go_deeper(graph, v_start)

Hilfsfunktion go_deeper(graph, v):
    markiere v als besucht
    verarbeite Knoten v /* z.B. print(v) */
    für jeden Nachbarn w von v:
        wenn w noch nicht besucht wurde:
            go_deeper(w)
```



	von	nach
1	A	~~C~~ ~~D~~ ~~B~~
7	B	~~F~~ ~~A~~
2	C	~~A~~ ~~F~~
8	D	~~A~~
5	E	~~H~~
3	F	~~C~~ ~~H~~ ~~B~~
6	G	~~H~~
4	H	~~F~~ ~~E~~ ~~G~~

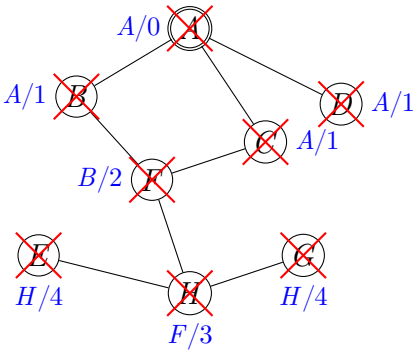
A, C, F, H, E, G, B, D

Die Tiefensuche besucht nicht der Reihe nach alle Nachbarn des aktuellen Knotens [das nämlich macht die Breitesuche] sondern geht zum ersten Nachbarn, der noch nicht besucht wurde und von dort auch wieder zum ersten Nachbarn der noch nicht besucht wurde usw., bis es nicht mehr weitergeht. Erst dann geht der Algorithmus eine „Ebene“ zurück und knöpft sich dort den nächsten unbesuchten Knoten vor, sofern es einen gibt. Von dort geht er gleich wieder in die Tiefe. Erst wenn es keine unbesuchten oder nur noch unerreichbare Knorten gibt, taucht dieser Prozess aus der Tiefensuche wieder auf und stoppt, wenn er wieder beim Startknoten ankommt und dort keine unbesuchten Knoten mehr vorfindet.

Aufgabe 25

Die Breitensuche *Breadth-First-Search* (BFS) ist eine weitere Methode, um in einem Graphen von einem Startknoten aus jeden erreichbaren Knoten zu besuchen. Im Gegensatz zur Tiefensuche liefert der Algorithmus zusätzlich die kürzteste Wege zwischen dem Startknoten und den erreichbaren Knoten.

1. Wähle den Startknoten, markiere ihn als gesehen und füge ihn in eine Warteschlange ein.
2. Entnimm einen Knoten vom Beginn der Warteschlange.
   (a) Falls das gesuchte Element gefunden wurde, brich die Suche ab und gib *gefunden* zurück.
   (b) Anderenfalls hänge alle bisher unmarkierten Nachbarn dieses Knotens ans Ende der Warteschlange an und markiere sie als gesehen.
3. Ist die Warteschlange leer, dann wurde jeder Knoten untersucht. Beende die Suche und gib *nicht gefunden* zurück.
4. Wiederhole Schritt 2.



	Warteschlange
	$[A]$
A	$[B, C, D]$
B	$[C, D, F]$
C	$[D, F]$
D	$[F]$
F	$[H]$
H	$[E, G]$
E	$[G]$
G	$[\,]$ Ende!

Interessieren uns die kürzesten Wege zum Startknoten A, weisen wir dem Startknoten A die Distanz 0 zu. Sobald wir vom Knoten v den Knoten w besuchen, ordnen wir w den Namen und den um 1 vergrösserten Distanzwert von v zu. Am Schluss weisen uns die angehefteten Knotennamen den kürzesten Weg bis zum Startknoten.

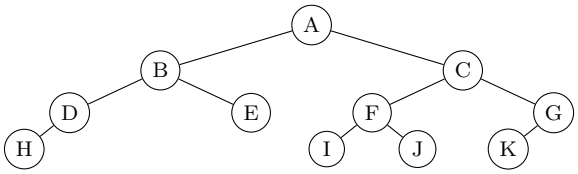
Aufgabe 26

Ein Baum T ist eine zusammenhängender Graph $G(V, E)$, in dem es keine Zyklen (geschlossene Wege) gibt. Ein Baum mit n Knoten hat immer $n - 1$ Kanten.

Aufgabe 27

- Datenkompression (Huffman-Bäume)
- Organisation hierarchischer Dateisysteme (in Ordnern und Unterordnern)
- Spieleentwicklung (Minimax-Algorithmus zum Auffinden optimaler Spielzüge)
- Minimale Spannbäume in Graphen
- Phylogenetische Bäume: Darstellung evolutionärer Beziehungen zwischen verschiedenen Organismen
- Die Struktur von HTML-Seiten

Aufgabe 28



(a) Welchen Schlüssel hat die Wurzel? *A*

(b) Welche Kinder hat der Knoten mit dem Schlüssel C? *F, G*

(c) Welche Geschwister hat der Knoten H? *keine*

(d) Welche Eltern hat der Knoten mit dem Schlüssel E? *B*

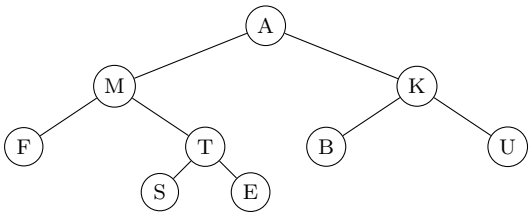
(e) Welches sind die Blätter des Baums? *H, E, I, J, K*

(f) Welches sind die inneren Knoten des Baums? *A, B, C, D, F, G*

(g) Welche Tiefe hat der Knoten mit dem Schlüssel D? *2*

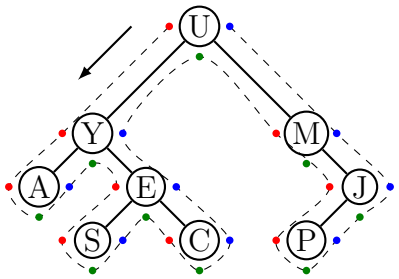
(h) Welche Höhe hat der Baum? *3* (Maximale Tiefe eines Blatts)

Aufgabe 29



Hinweis: Der Python-Code in der Frage muss nicht selbständig entwickelt werden können.

Aufgabe 30



Preorder: U Y A E S C M J P

Postorder: A S C E Y P J M U

In jedem Fall wird der Baum rekursiv durchlaufen („Besuche immer zuerst den linken Ast und erst wenn das nicht möglich ist, besuche den rechten Ast.“). Bei der Preorder-Traversierung wird der Knoten verarbeitet (z. B. ausgegeben) *bevor* seine Kinder verarbeitet wurden; bei der Postorder-Traversierung wird er verarbeitet *nachdem* seine Kinder verarbeitet wurden. Bei Binärbäumen (Bäume, deren Knoten maximal zwei Kinder haben) gibt es auch noch eine Inorder-Traversierung, wo die Verarbeitung *zwischen* der Verarbeitung der beiden Äste erfolgt (grüne Punkte).