

Datenstrukturen

Maturavorbereitung

Aufgabe 1

Erkläre, was ein **abstrakter Datentyp** ist und gib ein Beispiel dafür an.

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

Beschreibe die Datenstruktur *Array* und nenne Anwendungen dafür.

Aufgabe 2

Eine Array ist eine Datenstruktur, 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

Gegeben ist ein Array. Nenne neben dem Lese- und Schreibzugriff auf seine einzelnen Elemente noch weitere wünschenswerte Operationen auf Arrays.

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) Wie wird das Funktionsprinzip von Stacks oft abgekürzt?
- (b) Zähle Standardoperationen für Stacks auf.
- (c) Nenne zwei Anwendungen für Stacks.

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.

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 Anwendungen für Stacks.
- ▶ Undo-Funktion in Anwendungsprogrammen
 - ▶ Prüfen, ob ein Klammerterm korrekt ist
 - ▶ History-Funktion in Browsern
 - ▶ ...

Aufgabe 5

Vervollständige die Methoden der Python-Klasse Stack.

```
class Stack:

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

    def push(self, item):
        ...

    def pop(self):
        ...

    def is_empty(self):
        ...

    def size(self):
        ...
```

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

Stelle den Stack S beim Ausführen folgender Operationen als Liste dar, die von links nach rechts wächst. Zu Beginn ist der Stack leer.

```
S = Stack()  
S.push(3)  
S.push(5)  
x = S.pop()  
S.push(1)  
S.push(x)  
S.push(4)
```

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

Erkläre, wie die Funktion `check(string)` mit Hilfe des Stacks `S` eine Zeichenketten auf korrekte Klammerung prüft.

```
1 from stack import Stack
2 def check(string):
3     S = Stack()
4     for c in string:
5         if c == '(':
6             S.push(c)
7         elif c == ')':
8             if S.is_empty():
9                 return False
10            else:
11                S.pop()
12        else:
13            pass
14    if S.is_empty():
15        return True
16    return False
```

Aufgabe 7

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

Aufgabe 8

- (a) Wie wird das Funktionsprinzip von Warteschlangen abgekürzt?
- (b) Zähle Standardoperationen für Warteschlangen auf.
- (c) Nenne zwei Anwendungen für Warteschlangen.

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

Vervollständige die Methoden der Python-Klasse Queue.

```
class Queue:

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

    def enqueue(self, item):
        ...

    def dequeue(self):
        ...

    def is_empty(self):
        ...

    def size(self):
        ...
```

Warum ist eine Implementierung mit Listen nicht effizient?

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.

enqueue $O(n)$ →  → dequeue $O(1)$

Die Umkehrung der Einfügerichtung 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.

dequeue $O(n)$ ←  ← enqueue $O(1)$

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

Aufgabe 10

Stelle die Warteschlange Q beim Ausführen folgender Operationen als Liste dar, wenn die Elemente links eingefügt und rechts entfernt werden. Zu Beginn ist Q leer.

```
Q = Queue()
Q.enqueue(5)
Q.enqueue(1)
Q.enqueue(4)
x = Q.dequeue()
Q.enqueue(3)
Q.enqueue(x)
y = Q.dequeue()
```

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
```

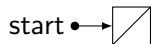
Aufgabe 11

Erläutere den Aufbau einfach verketteter Listen (EVL) anhand einer Skizze und beschreibe, wie man ...

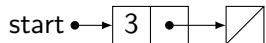
- ▶ ein Element am Anfang einer EVL hinzugefügt,
- ▶ ein Element vom Anfang einer EVL entfernt,
- ▶ eine EVL durchläuft (z. B. nach einem Wert sucht oder zählt).

Aufgabe 11

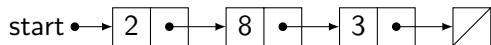
Neue EVL: Referenz (●) zeigt auf einen leeren Knoten (◻)



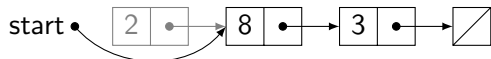
Füge 3 zur EVL hinzu:



Füge zuerst 8 und dann 2 zur EVL hinzu:

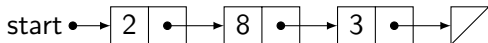


Entferne erstes Element aus EVL



- (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

Nenne drei verschiedene Anwendungen von verketteten Listen.

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

Vervollständige die Methoden der Python-Klasse `LinkedList`, indem du den leeren Knoten als leere Liste `[]` und Listen der Form `[value, reference]` als Container für den zu speichernden Wert die Referenz auf den nächsten Knoten implementierst.

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

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

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

    def is_empty(self):
        '''Ist EVL leer, gib True zurück; sonst False'''
        ...

    def remove(self):
        '''Entferne 1. Knoten und gib den Wert zurück'''
        '''(Ist die EVL leer, gib None zurück)'''
        ...
```

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

Stelle die einfach Verkettete Liste EVL beim Ausführen folgender Operationen als Verkettung von Listen der Länge 2 und der Länge 0 dar, wobei neue Knoten links eingefügt und entfernt werden.

```
EVL = LinkedList()  
EVL.insert(7)  
EVL.insert(3)  
x = EVL.remove()  
EVL.insert(2)  
print(x)
```

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

Stelle die im Speicherabbild enthaltene Daten einer einfach verkettete Liste mit der Startadresse 0x14 sequentiell dar.

- ▶ Der Speicher ist linear aufgebaut (auf die Adresse 0x0F folgt 0x10). Aus Platzgründen erfolgt die Darstellung mehrzeilig. Die Adressierung der Daten erfolgt durch eine zweistellige Hexadezimalzahl, wobei die linke Ziffer die Zeilennummer und die rechte Ziffer Spaltennummer darstellt.
- ▶ Ein *Knoten* besteht aus zwei benachbarten Speicherzellen. Die erste Speicherzelle enthält einen hexadezimalen Datenwert, die zweite die eine Adresse.
- ▶ 0x00 steht für den Nullzeiger oder eine leere Speicherzelle.
- ▶ Der Speicher kann auch noch andere oder verwaiste Daten enthalten.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	X	00	0B	41	00	00	00	00	00	00	00	00	00	00	00	22
1	37	00	00	00	3A	0F	00	32	41	00	00	00	00	00	00	00
2	00	00	00	00	00	00	00	00	00	0C	14	17	00	00	46	20
3	00	00	0A	22	00	00	00	32	00	00	00	00	00	24	46	00
4	00	00	00	00	00	00	00	02	04	00	00	00	00	00	00	00

Aufgabe 15

Start

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	⊗	00	0B	41	00	00	00	00	00	00	00	00	00	00	00	22
1	37	00	00	00	3A	0F	00	32	41	00	00	00	00	00	00	00
2	00	00	00	00	00	00	00	00	00	0C	14	17	00	00	46	20
3	00	00	0A	22	00	00	00	32	00	00	00	00	00	24	46	00
4	00	00	00	00	00	00	00	02	04	00	00	00	00	00	00	00

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

Was ist ein ungerichteter Graph?

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

Was ist ein gerichteter Graph?

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

Stelle den Graphen G mit

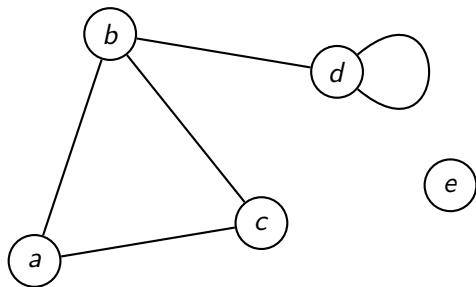
$$V = \{a, b, c, d, e\}$$

und

$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{d, d\}\}.$$

graphisch dar.

Aufgabe 18



Aufgabe 19

Stelle den Graphen $G = (V, E)$ mit

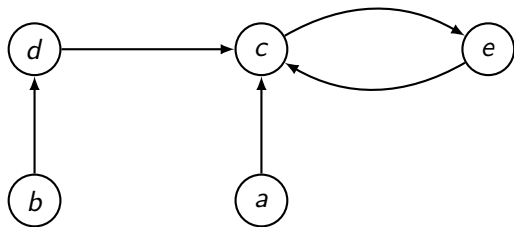
$$V = \{a, b, c, d, e\}$$

und

$$E = \{(a, c), (b, d), (c, e), (d, c), (e, c)\}$$

graphisch dar.

Aufgabe 19



Aufgabe 20

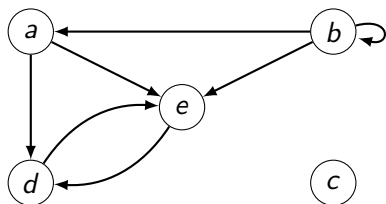
Nenne drei verschiedene Anwendungen von Graphen.

Aufgabe 20

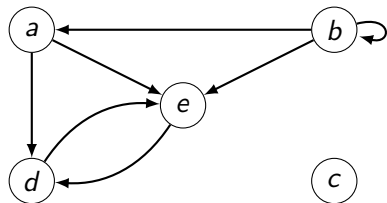
- ▶ *Modellierung von Verkehrsnetzen*: Orte und Kreuzungen bilden die Knoten und Verkehrswege die verbindenden Kanten. Haben die Kanten Gewichte, so können mit geeigneten Algorithmen kürzeste Wege berechnet werden.
- ▶ *Soziale Netzwerke*: Haben zwei Benutzer (Knoten) eine Beziehung, so wird dies durch 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 Websites (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

Stelle den gerichteten Graphen (a) als Adjazenzmatrix (b) als Adjazenzliste dar.



Aufgabe 21



(a) Adjazenzmatrix

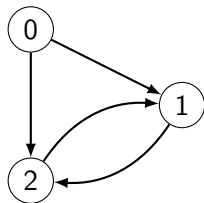
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	0	0	0	1	1
<i>b</i>	1	1	0	0	1
<i>c</i>	0	0	0	0	0
<i>d</i>	0	0	0	0	1
<i>e</i>	0	0	0	1	0

(b) Adjazenzliste

von	nach
<i>a</i>	<i>d, e</i>
<i>b</i>	<i>a, b, e</i>
<i>c</i>	–
<i>d</i>	<i>e</i>
<i>e</i>	<i>d</i>

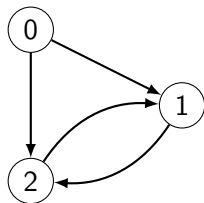
Aufgabe 22

Stelle den gerichteten Graphen



in Python irgendwie durch Listen dar.

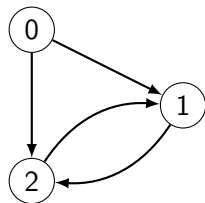
Aufgabe 22



(a) Lösung mit Adjazenzmatrix

$G = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, # Kanten 0 \rightarrow 1 und 0 \rightarrow 2
Kante 1 \rightarrow 2
Kante 2 \rightarrow 1

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

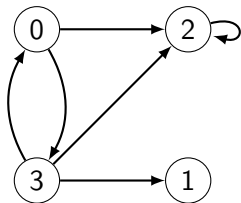
Das folgende Python-Codefragment stellt einen gerichteten Graphen dar.

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

Skizziere diesen Graphen.

Aufgabe 23

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



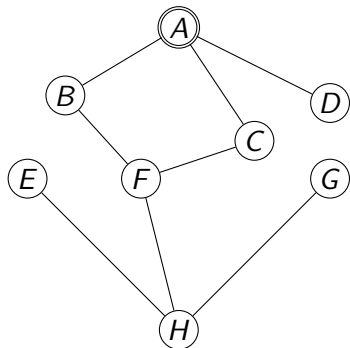
Adjazenzlistendarstellung:

- $G[0] = [2, 3]$
- $G[1] = []$
- $G[2] = [2]$
- $G[3] = [0, 1, 2]$

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

Aufgabe 24

Führe eine Tiefensuche gemäss der rechts angegebenen Adjazenzliste durch.



von	nach
A	C D B
B	F A
C	A F
D	A
E	H
F	C H B
G	H
H	F E G

Aufgabe 24

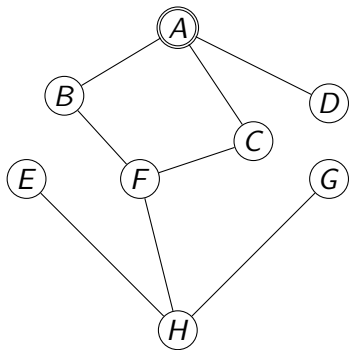
Die Tiefensuche *Depth-First-Search* (DFS) ist eine Methode, um einen Graphen zu **traversieren**, d. h. von einem Startknoten aus jeden erreichbaren Knoten zu besuchen.

```
Funktion dfs(graph, v_start):
```

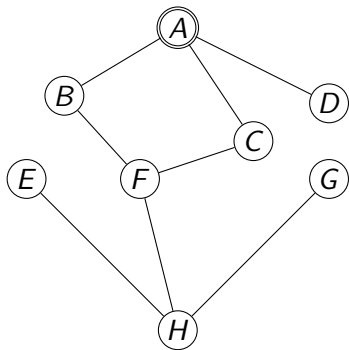
```
    Markiere alle Knoten von graph als unbesucht
    go_deeper(graph, v_start)
```

```
Hilfsfunktion go_deeper(graph, v):
```

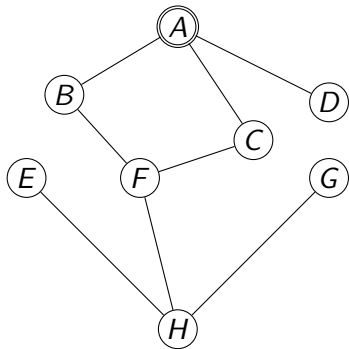
```
    markiere v als besucht
    verarbeite Knoten v /* z.B. print(v) */
    für jeden Nachbarn w von v:
        wenn w noch nicht besucht wurde:
            go_deeper(w)
```



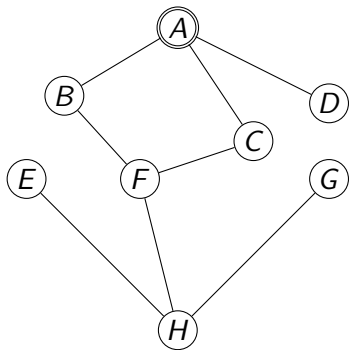
von	nach
A	C D B
B	F A
C	A F
D	A
E	H
F	C H B
G	H
H	F E G



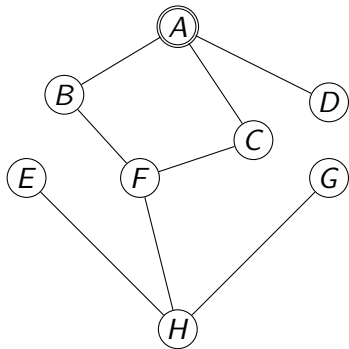
	von	nach		
1	A	C	D	B
	B	F	A	
	C	A	F	
	D	A		
	E	H		
	F	C	H	B
	G	H		
	H	F	E	G



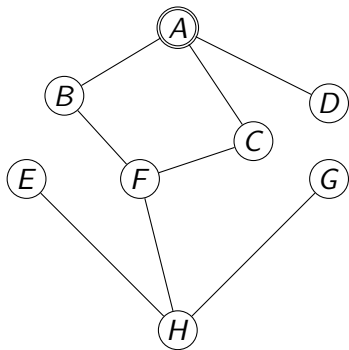
	von	nach
1	A	∅ D B
	B	F A
2	C	A F
	D	A
	E	H
	F	∅ H B
	G	H
	H	F E G



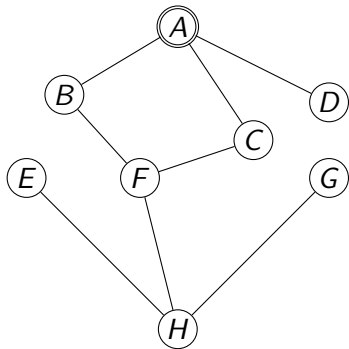
	von	nach
1	A	A D B
	B	F A
2	C	A F
	D	A
	E	H
	F	A H B
3	G	H
	H	F E G



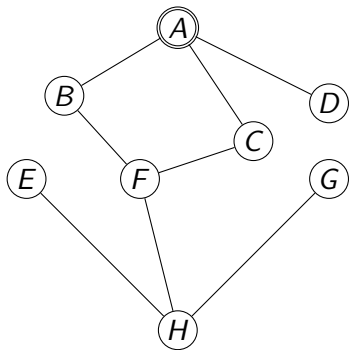
	von	nach
1	A	A D B
	B	F A
2	C	A F
	D	A
	E	H
3	F	A H B
	G	H
4	H	F E G



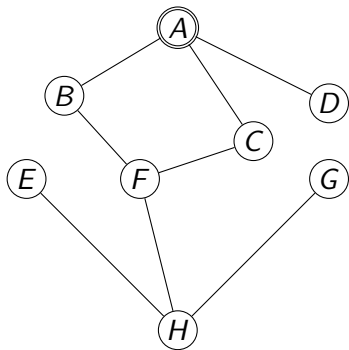
	von	nach
1	A	A D B
	B	F A
2	C	A F
	D	A
5	E	H
3	F	A H B
	G	H
4	H	F G



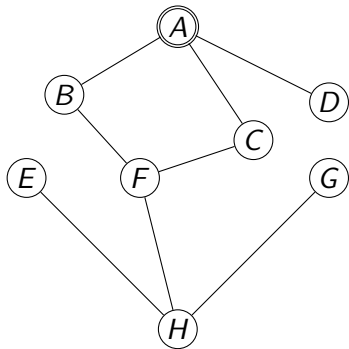
	von	nach
1	A	A D B
	B	F A
2	C	A F
	D	A
5	E	H
3	F	A H B
6	G	H
4	H	F E G



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

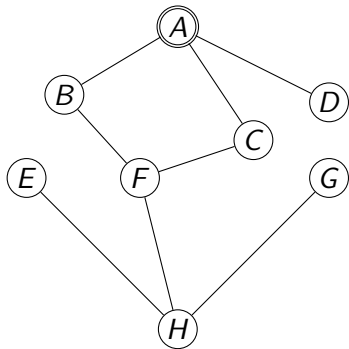


	von	nach
1	A	C D B
7	B	F A
2	C	A F
8	D	A
5	E	H
3	F	C H B
6	G	H
4	H	F E G



	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



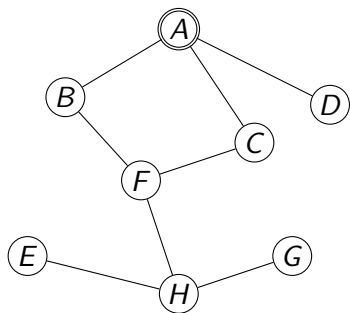
	von	nach
1	A	C D B
7	B	F A
2	C	A F
8	D	A
5	E	H
3	F	C H B
6	G	H
4	H	F E G

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 Knoten 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

Führe eine Breitensuche auf dem gegebenen Graphen durch, wobei Nachbarknoten in alphabetischer Reihenfolge besucht werden sollen.

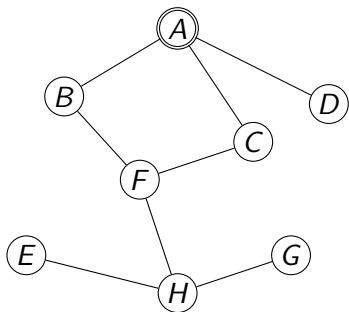


Aufgabe 25

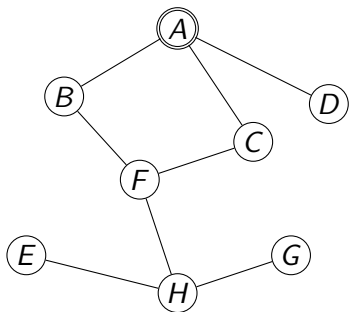
Die Breitensuche *Breadth-First-Search* (BFS) ist eine weitere Methode, um in einem Graphen von einem Startknoten aus jeden erreichbaren Knoten zu besuchen. Im Gegensatz zur Tiefensuche liefert der Algorithmus zusätzlich die kürzteste Wege zwischen dem Startknoten und den erreichbaren Knoten.

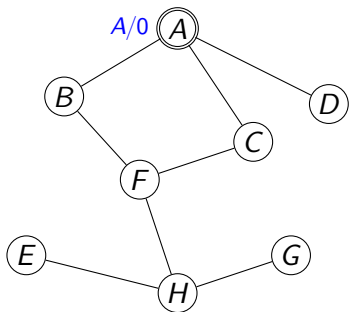
1. Wähle den Startknoten, markiere ihn als gesehen und füge ihn in eine Warteschlange ein.
2. Entnimm einen Knoten vom Beginn der Warteschlange.
 - (a) Falls das gesuchte Element gefunden wurde, brich die Suche ab und gib *gefunden* zurück.
 - (b) Anderenfalls hänge alle bisher unmarkierten Nachbarn 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

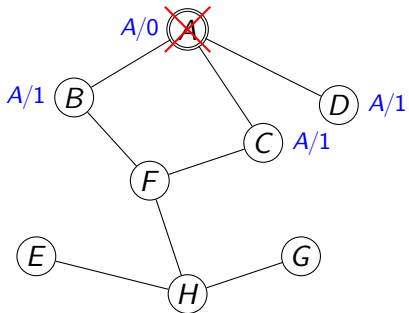


Warteschlange





Warteschlange
[A]

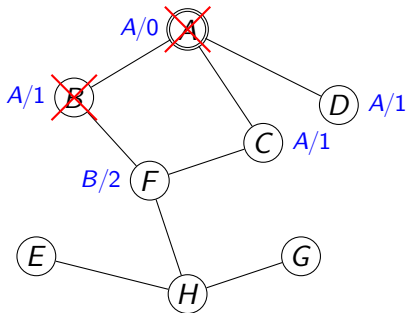


Warteschlange

[A]

A

[B, C, D]

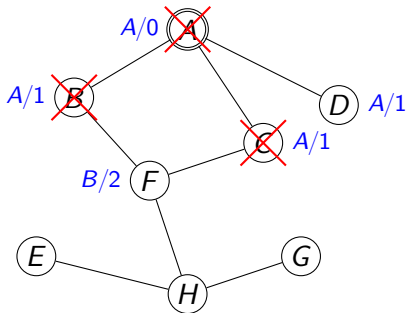


Warteschlange

[A]

A [B, C, D]

B [C, D, F]



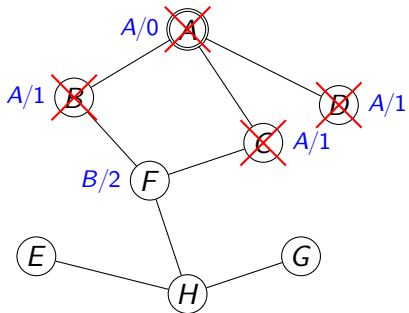
Warteschlange

[A]

A [B, C, D]

B [C, D, F]

C [D, F]



Warteschlange

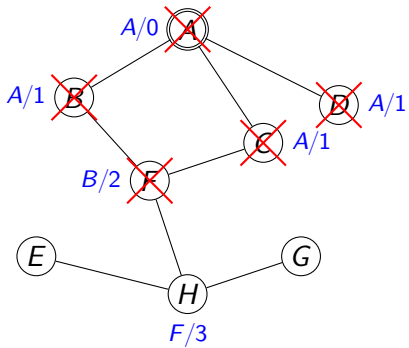
[A]

A [B, C, D]

B [C, D, F]

C [D, F]

D [F]



Warteschlange

[A]

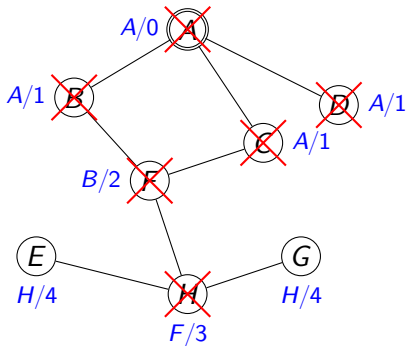
A [B, C, D]

B [C, D, F]

C [D, F]

D [F]

F [H]



Warteschlange

[A]

A [B, C, D]

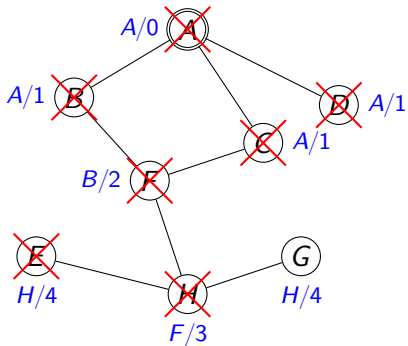
B [C, D, F]

C [D, F]

D [F]

F [H]

H [E, G]



Warteschlange

[A]

A [B, C, D]

B [C, D, F]

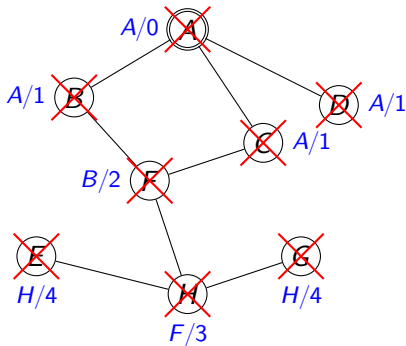
C [D, F]

D [F]

F [H]

H [E, G]

E [G]



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

Beschreibe die Datenstruktur *Baum* so genau wie möglich.

Aufgabe 26

Ein Baum T ist ein 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

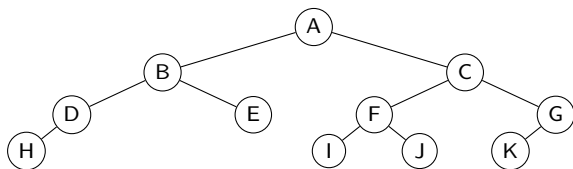
Nenne zwei Anwendungen für Bäume als Datenstruktur.

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

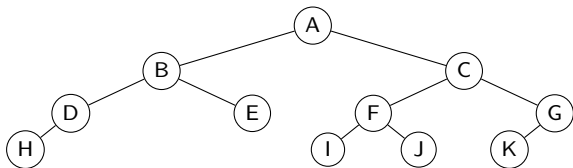
Aufgabe 28

Beantworte die Fragen zum folgenden Baum:



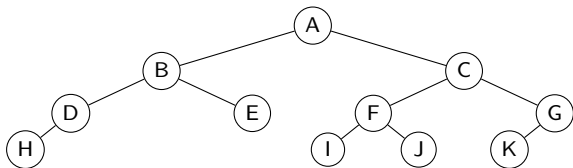
- (a) Welchen Schlüssel hat die Wurzel?
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C?
- (c) Welche Geschwister hat der Knoten mit dem Schlüssel H?
- (d) Welchen Elternknoten hat der Knoten mit dem Schlüssel E?
- (e) Welches sind die Blätter des Baums?
- (f) Welches sind die inneren Knoten des Baums?
- (g) Welche Tiefe hat der Knoten mit dem Schlüssel D?
- (h) Welche Höhe hat der Baum?

Aufgabe 28



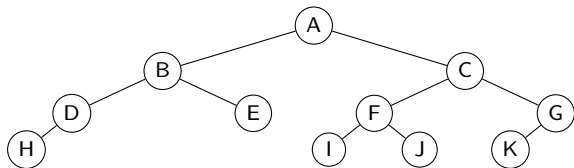
(a) Welchen Schlüssel hat die Wurzel?

Aufgabe 28



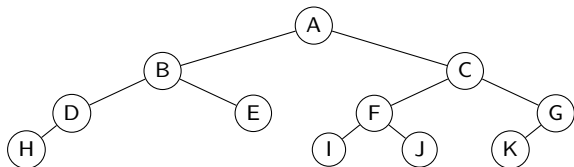
(a) Welchen Schlüssel hat die Wurzel? **A**

Aufgabe 28



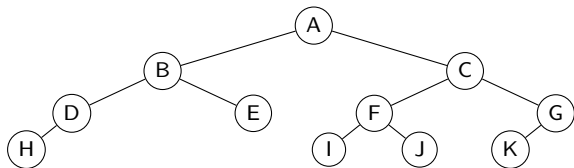
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C?

Aufgabe 28



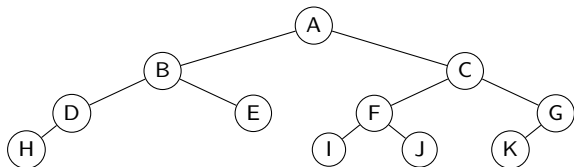
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**

Aufgabe 28



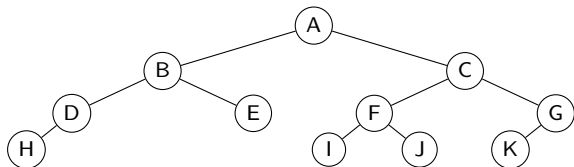
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H?

Aufgabe 28



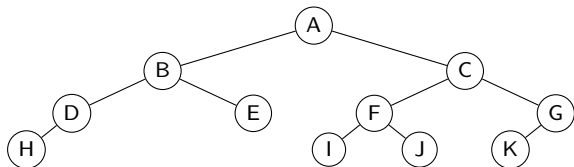
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**

Aufgabe 28



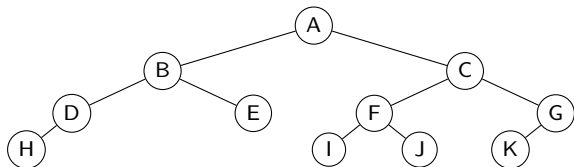
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E?

Aufgabe 28



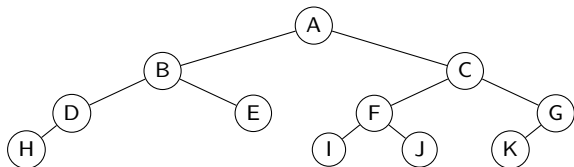
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**

Aufgabe 28



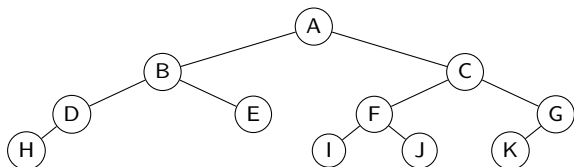
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums?

Aufgabe 28



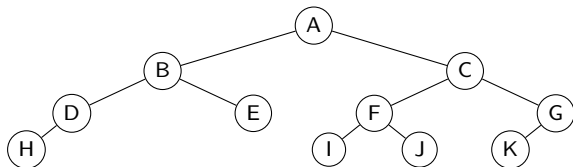
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums? **H, E, I, J, K**

Aufgabe 28



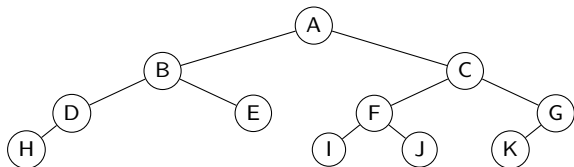
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums? **H, E, I, J, K**
- (f) Welches sind die inneren Knoten des Baums?

Aufgabe 28



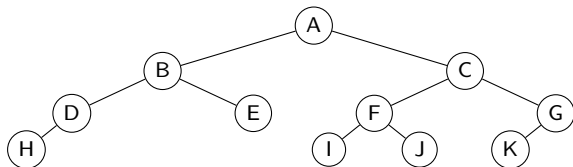
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums? **H, E, I, J, K**
- (f) Welches sind die inneren Knoten des Baums? **A, B, C, D, F, G**

Aufgabe 28



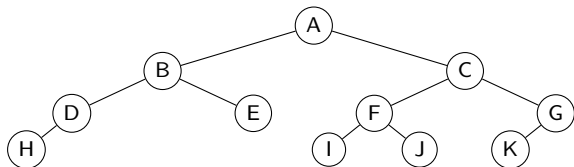
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums? **H, E, I, J, K**
- (f) Welches sind die inneren Knoten des Baums? **A, B, C, D, F, G**
- (g) Welche Tiefe hat der Knoten mit dem Schlüssel D?

Aufgabe 28



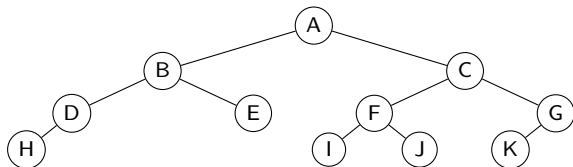
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums? **H, E, I, J, K**
- (f) Welches sind die inneren Knoten des Baums? **A, B, C, D, F, G**
- (g) Welche Tiefe hat der Knoten mit dem Schlüssel D? **2**

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?

Aufgabe 28



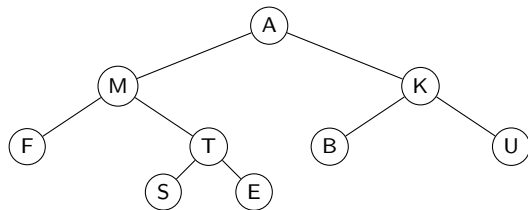
- (a) Welchen Schlüssel hat die Wurzel? **A**
- (b) Welche Kinder hat der Knoten mit dem Schlüssel C? **F, G**
- (c) Welche Geschwister hat der Knoten H? **keine**
- (d) Welche Eltern hat der Knoten mit dem Schlüssel E? **B**
- (e) Welches sind die Blätter des Baums? **H, E, I, J, K**
- (f) Welches sind 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

Skizziere den in Zeile 12–16 vom Python-Code erzeugten Baum.

```
1 class Tree:
2
3     def __init__(self, key):
4         self.root = key
5         self.edges = dict()
6
7     def add_children(self, parent, children):
8         self.edges[parent] = children
9         for c in children:
10             self.edges[c] = []
11
12 t = Tree('A')
13 t.add_children('A', ['M', 'K'])
14 t.add_children('M', ['F', 'T'])
15 t.add_children('T', ['S', 'E'])
16 t.add_children('K', ['B', 'U'])
```

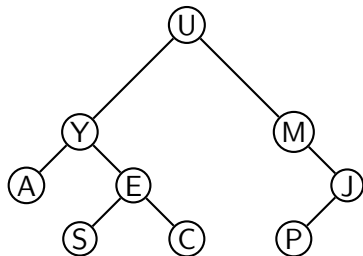
Aufgabe 29



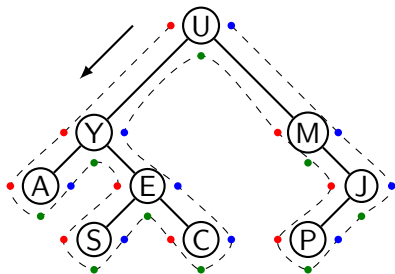
Hinweis: Der Python-Code in der Frage muss nicht selbständig entwickelt werden können.

Aufgabe 30

Gib die Folge der Knoten an, wenn der folgende Baum in
(a) Preorder-Reihenfolge (b) Postorder-Reihenfolge durchlaufen wird.



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).