

# Programovanie v Pythone

*časť 2*

**Andrej Blaho**

**sep 12, 2016**



<b>1 Spájané štruktúry</b>	<b>3</b>
1.1 Referencie . . . . .	3
1.2 Výpis pomocou cyklu . . . . .	6
1.3 Vytvorenie zoznamu pomocou cyklu . . . . .	8
1.4 Zistenie počtu prvkov . . . . .	10
1.5 Hľadanie vrcholu . . . . .	11
1.6 Zmena hodnoty vo vrchole . . . . .	11
1.7 Vloženie vrcholu do zoznamu . . . . .	12
<b>2 Spájané zoznamy, funkcie</b>	<b>15</b>
2.1 Enkapsulácia . . . . .	15
2.2 Urýchlenie pridávania na koniec . . . . .	18
2.3 Parametre funkcií . . . . .	20
2.4 Funkcia ako hodnota . . . . .	23
2.5 Mapovacie funkcie . . . . .	24
2.6 Generátorová notácia . . . . .	25
2.7 Použitie v spájaných zoznamoch . . . . .	25
<b>3 Zásobníky a rady</b>	<b>27</b>
3.1 Stack - zásobník . . . . .	27
3.1.1 Zásobník pomocou spájaného zoznamu . . . . .	27
3.2 Queue - rad, front . . . . .	29
3.2.1 Rad pomocou spájaného zoznamu . . . . .	29
3.2.2 Rad pomocou cyklického poľa . . . . .	30
3.3 Vyplňanie oblasti v dvojrozmernom poli . . . . .	31
<b>4 Binárne stromy</b>	<b>35</b>
4.1 Všeobecný strom . . . . .	35
4.2 Binárny strom . . . . .	36
4.3 Realizácia spájanou štruktúrou . . . . .	36
4.4 Nakreslenie stromu . . . . .	37
4.5 Generovanie nahodného stromu . . . . .	38
4.6 Ďalšie rekurzívne funkcie . . . . .	39
<b>5 Trieda BinarnyStrom</b>	<b>41</b>
5.1 Prechádzanie vrcholov stromu . . . . .	45
5.2 Prechádzanie po úrovniach . . . . .	48
<b>6 Použitie stromov</b>	<b>51</b>

---

6.1	Binárne vyhl'adávacie stromy . . . . .	51
6.1.1	Vkladanie do BVS . . . . .	54
6.1.2	Hľadanie v BVS . . . . .	56
6.1.3	Vypisovanie hodnôt v BVS . . . . .	56
6.1.4	Vyhodenie z BVS . . . . .	59
6.2	Aritmetické stromy . . . . .	59
6.3	Všeobecné stromy . . . . .	63
<b>7</b>	<b>Triedenia</b>	<b>67</b>
7.1	Min sort . . . . .	70
7.2	Bubble sort . . . . .	71
7.3	Vizualizovanie sortov . . . . .	72
7.4	Insert sort . . . . .	75
7.5	Quick sort . . . . .	76
<b>8</b>	<b>Grafy</b>	<b>81</b>
8.1	Termminológia . . . . .	81
8.2	Reprezentácie . . . . .	83
8.2.1	Pole množín susedností . . . . .	83
8.2.2	Pole zoznamov susedností . . . . .	87
8.2.3	Pole asociatívnych polí susedností . . . . .	87
8.2.4	Asociatívne pole množín susedností . . . . .	88
8.2.5	Matica susedností . . . . .	89
8.2.6	Matica susedností s váhami . . . . .	89
<b>9</b>	<b>Prehľadávanie grafu</b>	<b>91</b>
9.1	Algoritmus do hĺbky . . . . .	94
9.2	Všetky komponenty grafu . . . . .	99
9.3	Nerekurzívny algoritmus do hĺbky . . . . .	102
9.4	Algoritmus do šírky . . . . .	104
9.5	Vzdialosť a najkratšia cesta . . . . .	108
<b>10</b>	<b>Backtracking</b>	<b>113</b>
10.1	Generovanie štvoric čísel . . . . .	113
10.2	Rekurzia . . . . .	114
10.3	Zapuzdrime . . . . .	115
10.4	Backtracking . . . . .	117
10.5	Dámy na šachovnici . . . . .	118
10.6	Domček jedným t'ahom . . . . .	122
10.7	Sudoku . . . . .	124
<b>11</b>	<b>Backtracking na grafoch</b>	<b>127</b>
11.1	Základná schéma backtrackingu . . . . .	129
11.2	Hodnota cesty . . . . .	131
11.3	Zapamätanie celej cesty . . . . .	132
11.4	Hľadanie cyklov v grafe . . . . .	134

**Fakulta matematiky, fyziky a informatiky  
Univerzita Komenského v Bratislave**

**Autor** Andrej Blaho

**Názov** Programovanie v Pythone 2 (prednášky k predmetu Programovanie (2) 1-AIN-170/13)

**Vydavateľ** Knižničné a edičné centrum FMFI UK

**Rok vydania** 2016

**Miesto vydania** Bratislava

**Vydanie** prvé

**Počet strán** 222

**Internetová adresa** <http://python.input.sk/>

**ISBN** 978-80-8147-068-4



## Spájané štruktúry

Doteraz sme pracovali s “predpripravenými” dátovými štruktúrami jazyka Python (zjednodušene hovoríme, že štruktúra je typ, ktorý môže obsahovať viac prvkov), napr.

- `list` - pole (postupnosť) hodnôt, ktoré sú očíslované indexmi od 0 do počet prvkov-1
- `dict` - asociatívne pole, kde každému prvku zodpovedá kľúč
- `set` - množina rôznych hodnôt

Okrem toho už vieme konštruovať vlastné dátové typy pomocou definovania tried - inštancie tried obsahujú atribúty, ktoré sú budú stavovými premennými alebo metódami. Takto vieme vytvárať vlastné typy, pričom využívane štruktúry Pythonu.

### 1.1 Referencie

Už vieme, že premenné v Pythone (aj atribúty objektov) sú vlastne pomenované **referencie** na nejaké hodnoty, napr. čísla, reťazce polia, funkcie, atď. Referencie (kreslili sme ich šípkou) sú niečo ako adresou do pamäte, kde sa príslušná hodnota nachádza. Takýmito referenciami sú aj prvky iných štruktúrovaných typov, napr.

- pole čísel `[1, 2, 5, 2]` je v skutočnosti štvorprvkové pole referencií na hodnoty 1, 2, 5 (posledný prvok obsahuje rovnakú referenciu ako druhý prvok pol'a);
- asociatívne pole uchováva dvojice (kľúč, hodnota) a každá z nich je opäť referenciou na príslušné hodnoty;
- množina hodnôt sa tiež uchováva ako množina referencií na hodnoty
- atribúty tried, ktoré sú stavové premenné obsahujú tiež “iba” referencie na hodnoty.

Naučíme sa využívať referencie (adresy rôznych hodnôt, teda adresy objektov) na vytváranie **spájaných štruktúr**.

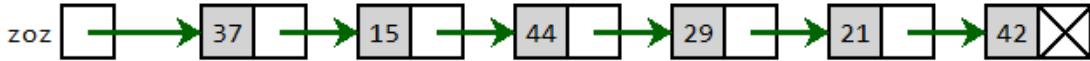
#### spájaný zoznam

Spájaný zoznam je taká štruktúra, kde každý prvok obsahuje referenciu (adresu, niekedy hovoríme aj *link*, *smerník*, *pointer*) na nasledovný prvok štruktúry. Prvkom štruktúry budeme hovoriť **Vrchol** (alebo niekedy po anglicky **Node**). Spájané štruktúry budú vždy prístupné pomocou premennej, ktorá odkazuje (obsahuje referenciu) na prvý prvok (vrchol) zoznamu. Spájaný zoznam reprezentuje postupnosť nejakých hodnôt a v tejto postupnosti je jeden vrchol posledný, za ktorým už nie je nasledovný prvok. Tento jeden vrchol si teda namiesto nasledovníka bude pamätať informáciu, že nasledovník už nie je - najčastejšie na to využijeme hodnotu `None`.

Takúto štruktúru si budeme kresliť takto:

- jedna premenná odkazuje (obsahuje referenciu) na prvý prvok (vrchol) zoznamu

- každý vrchol nakreslíme ako obdlžník s dvomi priečinkami: časť s údajom (pomenujeme to ako `data`) a časť s referenciou na nasledovníka (pomenujeme ako `next`)
- referencie budeme kresliť šípkami, pričom neexistujúcu referenciu (pre nasledovníka posledného vrcholu), t.j. hodnotu `None` môžeme značiť prečiarknutím políčka `next`



### Reprezentácia vrcholu

Vrchol budeme definovať ako objekt s dvoma premennými `data` a `next`:

```
class Vrchol:  
    def __init__(self, data, next=None):  
        self.data = data  
        self.next = next
```

Pozrime, čo definujeme týmto zápisom:

```
>>> v1 = Vrchol(11)
```

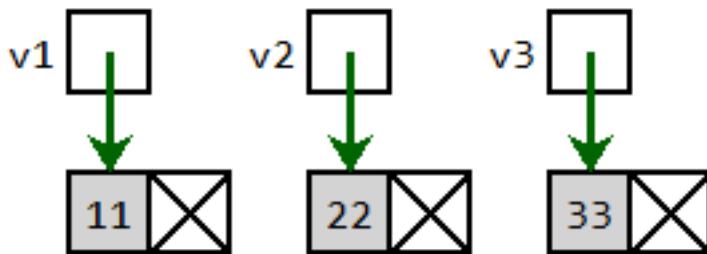
Vytvorili sme jeden vrchol, resp. jednovrcholový spájaný zoznam. Jeho nasledovníkom je `None`. Na tento vrchol odkazuje premenná `v1`. Jedine, čo zatiaľ môžeme s takýmto vrcholom robiť je to, že si vypíšeme jeho atribúty:

```
>>> print(v1.data)  
11  
>>> print(v1.next)  
None
```

Podobne zapíšeme:

```
>>> v2 = Vrchol(22)  
>>> v3 = Vrchol(33)
```

Teraz máme vytvorené 3 izolované vrcholy, ktoré sú prístupné pomocou troch premenných `v1`, `v2` a `v3`. Tieto tri vrcholy sa nachádzajú v rôznych častiach pamäti a nie sú nijako prepojené.



Vytvorme prvé prepojenie: ako nasledovníka `v1` nastavíme vrchol `v2`:

```
>>> v1.next = v2  
>>> print(v1.next)  
<__main__.Vrchol object at 0x01FAF0D0>
```

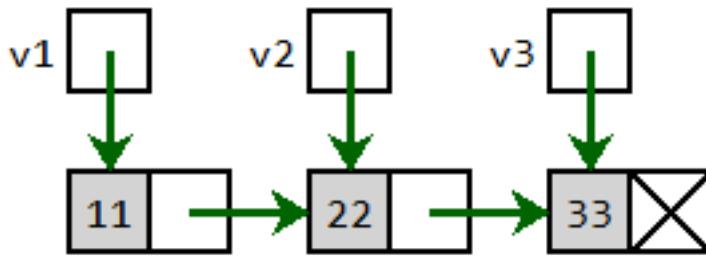
Vidíme, že nasledovníkom `v1` už nie je `None` ale nejaký objekt typu `Vrchol` - zrejme je to vrchol `v2`. Môžeme sa pozrieť na údaj nasledovníka `v1`:

```
>>> print(v1.next.data)
22
>>> print(v1.next.next)
None
```

Jeho nasledovníkom je `None`, teda nasledovníka nemá. Premenná `v1` obsahuje referenciu na vrchol, ktorý má jedného nasledovníka, t.j. `v1` odkazuje na dvojprvkový spájaný zoznam. Pripojme teraz do tejto postupnosti aj vrchol `v3`:

```
>>> v2.next = v3
```

Takže prvým vrcholom v spájanom zozname je `v1` s hodnotou 11, jeho nasledovníkom je `v2` s hodnotou 22 a nasledovníkom `v2` je `v3` s hodnotou 33. Nasledovníkom tretieho vrcholu je stále `None`, teda tento vrchol nemá nasledovníka.



Vytvorili sme trojprvkový spájaný zoznam, ktorého vrcholy majú postupne tieto hodnoty:

```
>>> print(v1.data)
11
>>> print(v1.next.data)
22
>>> print(v1.next.next.data)
33
```

Vidíme, že pomocou referencie na prvý vrchol sa vieme dostať ku každému vrcholu, len treba dostatočný počet krát zapísat' `next`. Premenné `v2` a `v3` teraz už nepotrebujeme a mohli by sme hoci aj zrušiť, na vytvorený zoznam to už nemá žiadnen vplyv:

```
>>> del v2, v3
```

Pozrime ešte na tento zápis:

```
>>> a = Vrchol('a')
>>> b = Vrchol('b')
>>> a.next = b
>>> del b
```

Vytvorí dvojprvkový zoznam, pričom premenná `b` je len pomocná a hned' sa aj zruší. To isté môžeme zapísat' aj bez nej:

```
>>> a = Vrchol('a')
>>> a.next = Vrchol('b')
```

Tu si všimnite, že inicializačná metóda (`Vrchol.__init__()`) má druhý parameter, ktorým môžeme definovať hodnotu `next` už pri vytváraní vrcholu. Preto môžeme tieto dve priradenia zlúčiť do jedného:

```
>>> a = Vrchol('a', Vrchol('b'))
```

Hoci teraz je tu malý rozdiel a to v tom, že vrchol `Vrchol('b')` sa vytvorí skôr ako `Vrchol('a')`, čo ale vo väčšine prípadov nevadí. Podobne by sme vedeli jedným priradením vytvoriť' nielen dvojprvkový, ale aj viacprvkový zoznam, napr.

```
>>> zoznam = Vrchol('P', Vrchol('y', Vrchol('t', Vrchol('h', Vrchol('o', Vrchol('n'))))))
```

Vytvorí šesť prvkový zoznam, pričom každý prvok obsahuje jedno písmeno z reťazca 'Python'.

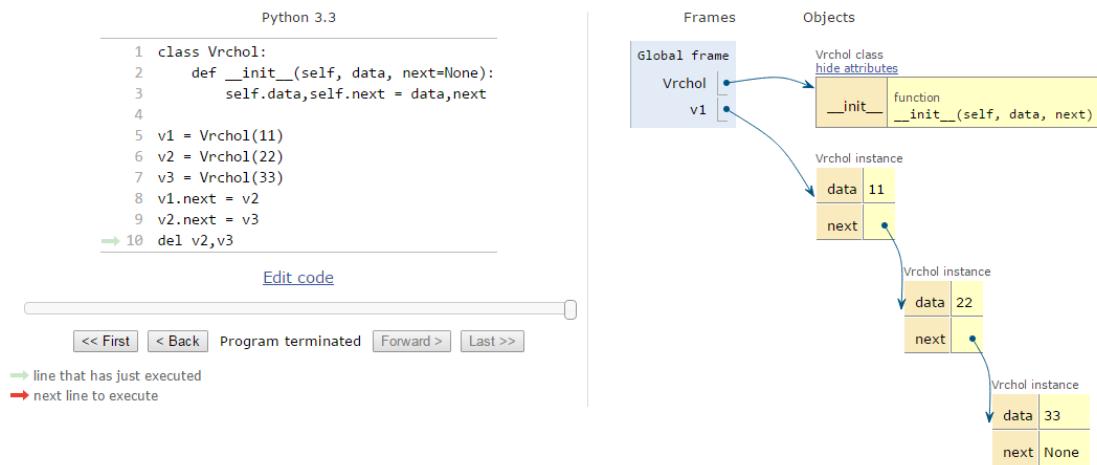
### pythontutor.com

Na webe <http://pythontutor.com/visualize.html> nájdete veľmi užitočný nástroj na vizualizáciu pythonovských programov. Môžete sem preniesť skoro ľubovoľný algoritmus, ktorý sme robili doteraz (okrem grafiky) a odkrokovat' ho. Pritom môžete vidieť, ako sa v pamäti postupne vytvárajú premenné a dátové štruktúry. Môžete sem preniesť napr. tento program

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

v1 = Vrchol(11)
v2 = Vrchol(22)
v3 = Vrchol(33)
v1.next = v2
v2.next = v3
del v2, v3
```

Po spustení vizualizácie dostávate:



Vidíme, že globálna premenná `v1` obsahuje referenciu na inštanciu triedy `Vrchol`, v ktorej atribút `data` má hodnotu 11 a atribút `next` je opäť referenciou na ďalšiu inštanciu triedy `Vrchol`, atď.

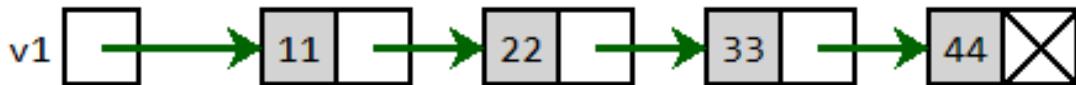
Tiež tu môžete vidieť, že globálna premenná `Vrchol` obsahuje referenciu na definíciu triedy `Vrchol`.

## 1.2 Výpis pomocou cyklu

Predpokladajme, že máme vytvorený nejaký napr. štvorprvkový zoznam:

```
>>> v1 = Vrchol(11, Vrchol(22, Vrchol(33, Vrchol(44))))
```

V pamäti by sme ho mohli vidieť nejako takto:



Teraz chceme vypísat' všetky jeho hodnoty postupne od prvej po poslednú, môžeme to urobiť napr. takto:

```
>>> print(v1.data)
11
>>> print(v1.next.data)
22
>>> print(v1.next.next.data)
33
>>> print(v1.next.next.next.data)
44
```

alebo v jednom riadku:

```
>>> print(v1.data, v1.next.data, v1.next.next.data, v1.next.next.next.data)
11 22 33 44
```

Zrejme pre zoznam ľubovoľnej dĺžky budeme musieť použiť nejaký cyklus, najskôr while-cyklus. Ked' vypíšeme prvú hodnotu, posunieme premennú v1 na nasledovníka prvého vrcholu:

```
>>> print(v1.data)
>>> v1 = v1.next
```

a môžeme to celé opakovať. Zápis v1 = v1.next je veľmi dôležitý a budeme ho v súvislosti so spájanými zoznamami používať veľmi často. Označuje, že do premennej v1 sa namiesto referencie na nejaký vrchol dostáva referencia na jeho nasledovníka. Ak už tento vrchol nasledovníka nemá, do v1 sa dostane hodnota None. Preto kompletnejší výpis hodnôt zoznamu môžeme zapísat' takto:

```
while v1 is not None:
    print(v1.data, end=' -> ')
    v1 = v1.next
print(None)
```

Pre názornosť sme tam medzi každé dve vypisované hodnoty pridali reťazec ' -> ':

```
11 -> 22 -> 33 -> 44 -> None
```

Hoci to vyzerá dobre a dostatočne jednoducho, má to jeden problém: po skončení vypisovania pomocou tohto while-cyklu je v premennej v1 hodnota None:

```
>>> print(v1)
None
```

Teda výpisom sme si zničili jedinú referenciu na prvy vrchol zoznamu a teda Python pochopil, že so zoznamom už pracovať d'alej nechceme a celú štruktúru z pamäti vyhodil. Môžete to skontrolovať aj vo vizualizácii <http://pythontutor.com/visualize.html>. Tento príklad ukazuje to, že niekedy je potrebné si uchovať referenciu na začiatok zoznamu, resp. v takomto cykle nebude pracovať priamo s premenou v1, ale s jej kópiou, napr. takto:

```
pom = v1
while pom is not None:
    print(pom.data, end=' -> ')
    pom = pom.next
print(None)
```

Po skončení tohto výpisu sa premenná `pom` vynuluje na `None`, ale začiatok zoznamu `v1` ostáva neporušený.

Takýto výpis môžeme zapísat' aj do funkcie, pričom tu pomocnú referenciu na začiatok zoznamu zastúpi parameter:

```
def vypis(zoznam):
    while zoznam is not None:
        print(zoznam.data, end=' -> ')
        zoznam = zoznam.next
    print(None)
```

Pri volaní funkcie sa do formálneho parametra `zoznam` priradí hodnota skutočného parametra (napr. obsah premennej `v1`) a teda referencia na začiatok zoznamu sa týmto volaním nepokazí.

Teraz môžeme volať funkciu na výpis nielen so začiatkom zoznamu ale hoci napr. aj od druhého vrcholu:

```
>>> vypis(v1)
11 -> 22 -> 33 -> 44 -> None
>>> vypis(v1.next)
22 -> 33 -> 44 -> None
```

Vidíme, že referencia na prvý vrchol v spájanom zozname má špeciálny význam a preto sa zvykne označovať nejakým dohodnutým menom, napr. `zoznam`, `zoz`, `zac`, `z` (ako začiatok zoznamu) alebo niekedy aj po anglicky `head` (hlavička zoznamu).

### 1.3 Vytvorenie zoznamu pomocou cyklu

Zoznamy sme doteraz vytvárali sériou priradení a to bez cyklov. častejšie ale budem vytvárať možno aj dlhé zoznamy pomocou opakujúcich sa konštrukcií. Začneme vytváraním zoznamu pridávaním nového vrcholu na začiatok doterajšieho zoznamu, keďže toto je výrazne jednoduchšie.

Vytvorime desaťprvkový zoznam s hodnotami 0, 1, 2, ... 9. Začneme s prázdnnym zoznamom:

```
>>> zoz = None
```

Vytvoríme prvý vrchol s hodnotou 0 a dáme ho na začiatok:

```
>>> pom = Vrchol(0)
>>> zoz = pom
```

Keby sme to vypísali, dostali by sme: `0 -> None`

Vytvoríme druhý vrchol a dáme ho opäť na začiatok:

```
>>> pom = Vrchol(1)
>>> pom.next = zoz
>>> zoz = pom
```

Po výpise by sme dostali: `1 -> 0 -> None`

Toto môžeme opakovať viackrát pre rôzne hodnoty - zakaždým sa na začiatok doterajšieho zoznamu pridá nový vrchol:

```
>>> pom = Vrchol(2)
>>> pom.next = zoz
>>> zoz = pom

>>> pom = Vrchol(3)
>>> pom.next = zoz
>>> zoz = pom
```

Takto by sme mohli pokračovať až do 9. Teraz už vidíme, čo sa tu opakuje a čo dáme do cyklu:

```
zoz = None      # zatiaľ ešte prázdny zoznam
for hodnota in range(10):
    pom = Vrchol(hodnota)
    pom.next = zoz
    zoz = pom
```

Týmto postupom sme dostali 10 prvkový zoznam hodnôt v poradí od 9 do 0:

```
>>> vypis(zoz)
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> None
```

Opäť si všimnime zápis tela cyklu:

```
for hodnota in range(10):
    pom = Vrchol(hodnota)
    pom.next = zoz
    zoz = pom
```

Vytvorí sa tu nový vrchol najprv s nasledovníkom `None`, potom sa tento nasledovník zmení na `pom.next = zoz` a na záver sa tento nový vrchol `pom` stáva novým začiatkom zoznamu, t.j. `zoz = pom`. Toto isté sa dá zapísat kompaktnejšie:

```
for hodnota in range(10):
    zoz = Vrchol(hodnota, zoz)
```

Zapamäťte si, že zápis `zoz = Vrchol(hodnota, zoz)` pre `zoz` začiatok zoznamu znamená pridanie nového vrcholu na začiatok.

Takto by sme vedeli vytvoriť ľubovoľné zoznamy, napr.

```
zoz1 = None
for hodnota in range(1000):
    zoz1 = Vrchol(hodnota, zoz1)

zoz2 = None
for hodnota in 'Python':
    zoz2 = Vrchol(hodnota, zoz2)
```

Vytvorí sa dva zoznamy: prvý s 1000 vrcholmi a druhý so šiestimi ale s písmenami reťazca ‘Python’. Len si pri tomto treba uvedomiť, že takto sa vytvárajú zoznamy s hodnotami v opačnom poradí, ako so do neho vkladali, teda:

```
>>> vypis(zoz1)
999 -> 998 -> ... -> 1 -> 0 -> None
>>> vypis(zoz2)
n -> o -> h -> t -> y -> P -> None
```

Častejšie budeme potrebovať vyrábať zoznamy, v ktorých budú prvky v tom poradí ako sme ich vkladali. Teda potrebujeme vedieť pridávať prvky na koniec existujúceho zoznamu. Od predchádzajúcej verzie sa to bude lísiť tým,

že pri pripájaní nového vrcholu do zoznamu, musíme najprv nájsť posledný prvok, za ktorý budeme vrchol vkladat'. Napíšme najprv časť programu, ktorá nájde posledný vrchol a pripojí za neho nový vrchol:

```
pom = zoz
while pom is not None:
    pom = pom.next
pom.next = Vrchol(hodnota)
```

Cyklus while sa tu podobá na cyklus pri vypisovaní. Lenže o tomto cykle vieme, že po jeho skončení sa premenná cyklu nastaví na None a nie na posledný vrchol. Takže cyklus potrebujeme skončiť o jeden prechod skôr ako bude platíť, že pom je None. Budeme teda testovať nie hodnotu pom ale jeho nasledovníka, teda pom.next:

```
pom = zoz
while pom.next is not None:
    pom = pom.next
pom.next = Vrchol(hodnota)
```

Teraz while cyklus skončí na poslednom (nie za posledným) vrchole, a teda môžeme za neho pripojiť nový vrchol. Toto ale bude fungovať len pre neprázdný zoznam, lebo ak zoz má hodnotu None, zoz.next spadne na chybe. Preto budeme prípad pridávania do prázdneho zoznamu riešiť zvlášť. Teraz už môžeme zapísat cyklus, ktorý vytvorí 10-prvkový zoznam pridávaním nových vrcholov na koniec:

```
zoz = None
for hodnota in range(1000):
    if zoz is None:
        zoz = Vrchol(hodnota)
    else:
        pom = zoz
        while pom.next is not None:
            pom = pom.next
        pom.next = Vrchol(hodnota)
```

Tento zápis vytvára nový zoznam pridávaním na koniec:

```
>>> vypis(zoz)
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> None
```

Otestujte vytvorenie napr. 10000-prvkového zoznamu a zamyslite sa, prečo tento cyklus tak dlho trvá.

## 1.4 Zistenie počtu prvkov

Zapíšeme funkciu, ktorá spočíta počet prvkov zoznamu. Bude pracovať na rovnakom princípe ako funkcia vypis () len namiesto samotného vypisovania hodnoty funkcia zvýši počítadlo o 1:

```
def pocet(zoznam):
    vysl = 0
    while zoznam is not None:
        vysl += 1
        zoznam = zoznam.next
    return vysl
```

Otestujeme napr.

```
>>> zoznam = Vrchol('P', Vrchol('y', Vrchol('t', Vrchol('h', Vrchol('o', Vrchol('n'))))))  
>>> pocet(zoznam)  
6
```

## 1.5 Hľadanie vrcholu

Podobný cyklus, ako sme použili pri výpise a pri zistovaní počtu prvkom, môžeme použiť pri hľadaní vrcholu s nejakou konkrétnou hodnotou. Napíšme funkciu, ktorá vráti True, ak nájde konkrétnu hodnotu, inak vráti False:

```
def zisti(zoznam, hodnota):  
    while zoznam is not None:  
        if zoznam.data == hodnota:  
            return True  
        zoznam = zoznam.next  
    return False
```

Otestujeme:

```
>>> zoznam = Vrchol('P', Vrchol('y', Vrchol('t', Vrchol('h', Vrchol('o', Vrchol('n'))))))  
>>> zisti(zoznam, 'h')  
True  
>>> zisti(zoznam, 'g')  
False
```

Táto funkcia skončila prechádzanie prvkov zoznamu pri prvom výskytu hľadanej hodnoty. Ak by sme ale potrebovali zistiť počet výskytovnejakej hodnoty, museli by sme prejsť celý zoznam:

```
def pocet_vyskytov(zoznam, hodnota):  
    pocet = 0  
    while zoznam is not None:  
        if zoznam.data == hodnota:  
            pocet += 1  
        zoznam = zoznam.next  
    return pocet
```

## 1.6 Zmena hodnoty vo vrchole

Hľadanie nejakej konkrétnej hodnoty môžeme využiť aj na zmenu tohto vrcholu. Zapíšme opravu nájdenej hodnoty za nejakú inú:

```
zoz = Vrchol(10, Vrchol(11, Vrchol(12, Vrchol(13, Vrchol(14, Vrchol(15))))))  
hodnota = 13  
nova_hodnota = 'abc'  
pom = zoz  
while pom is not None and pom.data != hodnota:  
    pom = pom.next  
  
if pom is None:  
    print('nenasiel')  
else:  
    pom.data = nova_hodnota
```

Po výpise takého zoznamu:

```
>>> vypis(zoz)
10 -> 11 -> 12 -> abc -> 14 -> 15 -> None
```

Všimnite si, že po skončení while-cyklu sme museli kontrolovať, či premenná `pom` nemá hodnotu `None`. Vtedy to znamená, že sa prešli všetky vrcholy zoznamu a pre žiadensia `pom.data` nerovnalo hľadanej hodnote.

### 1.7 Vloženie vrcholu do zoznamu

Hľadanie vrcholu v zozname môžeme využiť aj pre vloženie nového vrcholu do zoznamu. Zapíšme vloženie nového vrcholu **za nájdený** s nejakou hľadanou hodnotou:

```
zoz = Vrchol(10, Vrchol(11, Vrchol(12, Vrchol(13, Vrchol(14, Vrchol(15))))))
hodnota = 13
nova_hodnota = 'HURA'
pom = zoz
while pom is not None and pom.data != hodnota:
    pom = pom.next

if pom is None:
    print('nenasiel')
else:
    novy = Vrchol(nova_hodnota)      # teda novy = Vrchol(nova_hodnota, pom.
    ↪next)
    novy.next = pom.next
    pom.next = novy
```

Po výpise takého zoznamu:

```
>>> vypis(zoz)
10 -> 11 -> 12 -> 13 -> HURA -> 14 -> 15 -> None
```

Vkladanie **pred nájdený** vrchol bude trochu náročnejšie: keď už nájdeme hľadaný vrchol, potrebovali by sme vedieť (si zapamätať), ktorý vrchol bol tesne pred ním - teda ktorý vrchol bol vo while-cykle v predchádzajúcim prechode cyklu. Použijeme na to pomocnú premennú `pred`, ktorá si bude pamätať predchodcu daného vrcholu:

```
zoz = Vrchol(10, Vrchol(11, Vrchol(12, Vrchol(13, Vrchol(14, Vrchol(15))))))
hodnota = 13
nova_hodnota = 'HURA'
pom = zoz
pred = None      # zatiaľ' je predchodca None, lebo prvý vrchol nemá
    ↪predchodcu
while pom is not None and pom.data != hodnota:
    pred = pom
    pom = pom.next

if pom is None:
    print('nenasiel')
elif pred is None:          # vkladáme pred prvý vrchol
    zoz = Vrchol(nova_hodnota, zoz)
else:
    pred.next = Vrchol(nova_hodnota, pom)
```

Po výpise takého zoznamu:

```
>>> vypis(zoz)
10 -> 11 -> 12 -> HURA -> 13 -> 14 -> 15 -> None
```



---

## Spájané zoznamy, funkcie

---

Budeme pokračovať vo využívaní dátovej štruktúry spájaný zoznam. Jeho prvky sú pospájané pomocou referencií v atribúte `next`:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next
```

Zrejme s takýmto zoznamom sa pracuje dosť nepohodlne, keďže informáciu o prvom prvku zoznamu máme len v nejakej globálnej premennej. Hoci vytváranie a vypisovanie takéhoto zoznamu niekedy nevyzerá až tak zle:

```
def vypis(zoz):
    while zoz is not None:
        print(zoz.data, end='->')
        zoz = zoz.next
    print(None)

def pocet(zoz):
    vysl = 0
    while zoz is not None:
        vysl += 1
        zoz = zoz.next
    return vysl

def pridaj_zac(zoz, data):
    return Vrchol(data, zoz)

z = None          # zatiaľ' ešte prázdny zoznam
for hodnota in range(10):
    z = pridaj_zac(z, hodnota)
vypis(z)
print('pocet prvkov =', pocet(z))
```

### 2.1 Enkapsulácia

Využijeme **zapuzdrenie** (enkapsuláciu) na vytvorenie triedy `Zoznam`, t.j. v triede sa okrem potrebných atribútov (referencia na začiatok zoznamu) budú nachádzať aj všetky metódy. Všimnite si, že namiesto názvov `vypis` a `pocet` sme použili špeciálne mená `__repr__()` a `__len__()`:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

class Zoznam:
    def __init__(self):
        self.zac = None # referencia na začiatok zoznamu

    def __repr__(self): # spájaný zoznam sa bude vypisovať aj so zátvorkami
        zoz = self.zac
        vysl = '('
        while zoz is not None:
            vysl += repr(zoz.data) + '->'
            zoz = zoz.next
        return vysl + ')'

    def __len__(self):
        zoz = self.zac
        vysl = 0
        while zoz is not None:
            vysl += 1
            zoz = zoz.next
        return vysl

    def pridaj_zac(self, data):
        self.zac = Vrchol(data, self.zac)
```

Otestujeme:

```
>>> z = Zoznam()
>>> for i in range(10):
...     z.pridaj_zac(i)
>>> z
(9->8->7->6->5->4->3->2->1->
>>> len(z)
10
```

Ked'že sme nové prvky pridávali na začiadok, vytvorený zoznam ich mal v opačnom poradí. Pridajme metódu, ktorá bude prvky vkladať na koniec:

```
class Zoznam:
    ...
    def pridaj_kon(self, data):
        if self.zac is None:
            self.zac = Vrchol(data)
        else:
            kon = self.zac
            while kon.next is not None:
                kon = kon.next
            kon.next = Vrchol(data)
```

Po otestovaní vidíme, že to asi funguje správne:

```
>>> z = Zoznam()
>>> for i in range(10):
...     z.pridaj_kon(i)
>>> z
(0->1->2->3->4->5->6->7->8->9->)
>>> len(z)
10
```

Pridajme ešte metódu, ktorá zistí či je prvok v zozname (využijeme špeciálne meno metódy `__contains__()`, vďaka čomu budeme môcť použiť operátor `in`) a do inicializácie pridáme vytvorenie počiatočného obsahu zoznamu z nejakého pol'a (prípadne z inej štruktúry, ktorá sa dá prechádzať for-cykлом):

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

class Zoznam:
    def __init__(self, pole=None):
        self.zac = None
        if pole is not None:
            for prvok in pole:
                self.pridaj_kon(prvok)

    def __repr__(self):
        zoz = self.zac
        vysl = ''
        while zoz is not None:
            vysl += repr(zoz.data) + '->'
            zoz = zoz.next
        return '(' + vysl + ')'

    def __len__(self):
        zoz = self.zac
        vysl = 0
        while zoz is not None:
            vysl += 1
            zoz = zoz.next
        return vysl

    def __contains__(self, data):
        zoz = self.zac
        while zoz is not None:
            if zoz.data == data:
                return True
            zoz = zoz.next
        return False

    def pridaj_zac(self, data):
        self.zac = Vrchol(data, self.zac)

    def pridaj_kon(self, data):
        if self.zac is None:
            self.zac = Vrchol(data)
        else:
            kon = self.zac
            while kon.next is not None:
                kon = kon.next
            kon.next = Vrchol(data)
```

Otestujeme napr.

```
>>> z = Zoznam([2, 3, 5, 7, 11, 13])
>>> z
(2->3->5->7->11->13->)
>>> len(z)
6
>>> 7 in z
True
>>> 9 in z
False
```

alebo

```
>>> zoz = Zoznam()
>>> for i in range(11, 100, 11):
    if i%2:
        zoz.pridaj_zac(i)
    else:
        zoz.pridaj_kon(i)

>>> zoz
(99->77->55->33->11->22->44->66->88->)
>>> for i in range(100):
    if i in zoz:
        print(i, end=' ')
11 22 33 44 55 66 77 88 99
```

## 2.2 Urýchlenie pridávania na koniec

Pri experimentovaní s dátovou štruktúrou spájaný zoznam a najmä s pridávaním na koniec:

```
>>> z1 = Zoznam(range(10, 25))
>>> z1
(10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->)
>>> len(z1)
15
>>> z2 = Zoznam('Python')
>>> z2
('P'->'y'->'t'->'h'->'o'->'n'->)
>>> z3 = Zoznam(range(10000))
>>> len(z3)
10000
```

Iste ste si všimli, že vytvorenie 10000-prvkového zoznamu už trvá niekoľko sekúnd (závisí od rýchlosťi vášho procesora). Pritom, ak by sme pri inicializácii zoznamu nepridávali nové prvky na koniec, ale na začiatok zoznamu (namiesto `pridaj_kon()` v `__init__()` dám `pridaj_zac()`), takáto inicializácia sa skráti na stotinky sekundy. V čom je tu problém? Metóda `pridaj_kon()` obsahuje while-cyklus, ktorý prejde všetky prvky zoznamu a snaží sa nájsť posledný prvok. Aj pre niekoľko tisíc prvkový zoznam toto prejde dosť rýchlo. Lenže tento cyklus sa stále opakuje pre stále dlhší a dlhší zoznam a keď budeme niečo krátke (čo trvá napr. tisícinu sekundy) opakovať niekoľko tisíc krát, celkový čas narastie už na sekundy.

Mali by sme urýchliť nájdenie posledného prvku v zozname. Jednoduchým riešením tu bude to, že si okrem atribútú

na začiatok zoznamu (premenná `self.zac`) budeme uchovávať aj referenciu na koniec zoznamu (v premennej `self.kon`). Všimnite si, že sme museli trochu upraviť aj metódu `pridaj_zac()`:

```
class Zoznam:

    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

        def __init__(self, pole=None):
            self.zac = self.kon = None
            if pole is not None:
                for prvok in pole:
                    self.pridaj_kon(prvok)

        def __repr__(self):
            zoz = self.zac
            vysl = ''
            while zoz is not None:
                vysl += repr(zoz.data) + '->'
                zoz = zoz.next
            return '(' + vysl + ')'

        def __len__(self):
            zoz = self.zac
            vysl = 0
            while zoz is not None:
                vysl += 1
                zoz = zoz.next
            return vysl

        def __contains__(self, data):
            zoz = self.zac
            while zoz is not None and zoz.data != data:
                zoz = zoz.next
            return zoz is not None

        def pridaj_zac(self, data):
            self.zac = self.Vrchol(data, self.zac)
            if self.kon is None:
                self.kon = self.zac

        def pridaj_kon(self, data):
            if self.zac is None:
                self.zac = self.kon = self.Vrchol(data, self.zac)
            else:
                self.kon.next = self.Vrchol(data)
                self.kon = self.kon.next
```

*Poznámka:*

V anglickej literatúre, resp. na internete sa môžete stretnúť s terminológiou:

- **node** - vrchol
- **linked list** - spájaný zoznam
- **head** - začiatok zoznamu
- **rear** - koniec zoznamu

Všimnite si, že metódu `__contains__()` sme zapísali trochu inak, aby ste videli aj jej iný zápis. Tiež sme definitívne triedy `Vrchol` prestáhovali do "vnútra" definície triedy `Zoznam`. Tým sa táto pomocná trieda `Vrchol` stala **súkromnou** podtryedou a naznačujeme tým, že by nebolo dobre, keby sme s ňou pracovali mimo triedy. Zároveň s tým sme museli poopraviť vytváranie vrcholov pomocou `self.Vrchol(data, ...)`.

### 2.3 Parametre funkcií

Zapíšme funkciu, ktorá vypočíta súčin niekoľkých čísel. Aby sme ju mohli volať s rôznym počtom parametrov, využijeme náhradné hodnoty:

```
def sucin(a=1, b=1, c=1, d=1, e=1):
    return a * b * c * d * e
```

Túto funkciu môžeme volať aj bez parametrov, ale nefunguje viac ako 5 parametrov:

```
>>> sucin(3, 7)
21
>>> sucin(2, 3, 4)
24
>>> sucin(2, 3, 4, 5, 6)
720
>>> sucin(2, 3, 4, 5, 6, 7)
...
TypeError: sucin() takes from 0 to 5 positional arguments but 6 were given
>>> sucin()
1
>>> sucin(13)
13
```

Ak chceme použiť aj väčší počet parametrov, môžeme využiť pole:

```
def sucin(pole):
    vysl = 1
    for prvok in pole:
        vysl *= prvok
    return vysl
```

Teraz to funguje pre ľubovoľný počet čísel, ale musíme ich uzavrieť do hranatých (alebo okrúhlych) zátvoriek:

```
>>> sucin([3, 7])
21
>>> sucin([2, 3, 4, 5, 6])
720
>>> sucin([2, 3, 4, 5, 6, 7])
5040
>>> sucin(range(2, 8))
5040
>>> sucin(range(2, 41))
815915283247897734345611269596115894272000000000
```

Teraz môžeme namiesto pola poslať ako parameter aj `range(2, 8)`, t.j. ľubovoľnú štruktúru, ktorá sa dá rozobrat' pomocou for-cyklu.

Toto riešenie stále nerieši náš problém: funkciu s ľubovoľným počtom parametrov. Na toto slúži tzv. **zbalený parameter**:

- pred menom parametra v hlavičke funkcie píšeme znak \* (zvyčajne je to posledný parameter)

- pri volaní funkcie sa všetky zvyšné parametre **zbalia** do jednej n-tice (typ `tuple`)

Otestujme:

```
def test(prvy, *zvysne):
    print('prvy =', prvy)
    print('zvysne =', zvysne)
```

po spustení:

```
>>> test('jeden', 'dva', 'tri')
prvy = jeden
zvysne = ('dva', 'tri')
>>> test('jeden')
prvy = jeden
zvysne = ()
```

Funkcia sa môže volať s jedným alebo aj viac parametrami. Prepíšme funkciu `sucin()` s použitím jedného zbaleného parametra:

```
def sucin(*pole):          # zbalený parameter
    vysl = 1
    for prvok in pole:
        vysl *= prvok
    return vysl
```

Uvedomte si, že teraz jeden parameter `pole` zastupuje ľubovoľný počet parametrov a Python nám do tohto parametra automaticky zbalí všetky skutočné parametre ako jednu n-ticu (tuple). Otestujeme:

```
>>> sucin()
1
>>> sucin(3, 7)
21
>>> sucin(2, 3, 4, 5, 6, 7)
5040
>>> sucin(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
479001600
>>> sucin(range(2, 13))
...
TypeError: unsupported operand type(s) for *=: 'int' and 'range'
```

V poslednom príklade vidíte, že `range(...)` tu nefunguje: Python tento jeden parameter zbalí do n-tice a potom sa s týmto `range()` bude chcieť násobiť, čo samozrejme nefunguje. Uvidíme nižšie, že aj toto sa dá riešiť.

Ešte ukážme druhý podobný prípad, ktorý sa môže vyskytnúť pri práci s parametrami funkcií. Napíšme funkciu, ktorá dostáva dva alebo tri parametre a nejako ich vypíše:

```
def pis(meno, priezvisko, rok=2015):
    print('volam sa {} {} a narodil som sa v {}'.format(meno, priezvisko, rok))
```

Napr.

```
>>> pis('Janko', 'Hrasko', 2014)
volam sa Janko Hrasko a narodil som sa v 2014
>>> pis('Juraj', 'Janosik')
volam sa Juraj Janosik a narodil som sa v 2015
```

Malá nepríjemnosť nastáva vtedy, keď máme takéto hodnoty pripravené vnejakej štruktúre:

```
>>> p1 = ['Janko', 'Hrasko', 2014]
>>> p2 = ['Juraj', 'Janosik']
>>> p3 = ['Monty', 'Python', 1968]
>>> pis(p1)
...
TypeError: pis() missing 1 required positional argument: 'priezvisko'
```

Nefunguje volanie tejto funkcie s trojprvkovým poľom, ale musíme prvky tohto poľa **rozbalíť**, aby sa priradili do príslušných parametrov, napr.

```
>>> pis(p1[0], p1[1], p1[2])
volam sa Janko Hrasko a narodil som sa v 2014
>>> pis(p2[0], p2[1])
volam sa Juraj Janosik a narodil som sa v 2015
```

Ked'že sa pri programovaní často stáva situácia, že máme v nejakej štruktúre (napr. v poli) pripravené parametre pre danú funkciu a my potrebujeme túto funkciu zavolať s rozbalenými prvkami štruktúry. Na toto slúži **rozbalovací operátor**, pomocou ktorého môžeme ľubovoľnú štruktúru poslat' ako parametre, pričom sa automaticky rozbalia (a teda prvy sa priradia do formálnych parametrov). Rozbalovací operátor pre parametre je opäť znak \* a používa sa takto:

```
>>> pis(*p1)           # je to isté ako pis(p1[0], p1[1], p1[2])
volam sa Janko Hrasko a narodil som sa v 2014
>>> pis(*p2)           # je to isté ako pis(p2[0], p2[1])
volam sa Juraj Janosik a narodil som sa v 2015
```

Takže, všade tam, kde sa očakáva nie jedna štruktúra ako parameter, ale veľ'a parametrov, ktoré sú prvkami tejto štruktúry, môžeme použiť tento rozbalovací operátor (po anglicky *unpacking argument lists*).

Tento operátor môžeme využiť napr. aj v takýchto situáciách:

```
>>> print(range(10))
range(0, 10)
>>> print(*range(10))
0 1 2 3 4 5 6 7 8 9
>>> print(*range(10), sep='....')
0....1....2....3....4....5....6....7....8....9
>>> param = (3, 20, 4)
>>> print(*range(*param))
3 7 11 15 19
>>> dvenasto = 2**100
>>> print(dvenasto)
1267650600228229401496703205376
>>> print(*str(dvenasto))
1 2 6 7 6 5 0 6 0 0 2 2 8 2 2 9 4 0 1 4 9 6 7 0 3 2 0 5 3 7 6
>>> print(*str(dvenasto), sep='-')
1-2-6-7-6-5-0-6-0-2-2-8-2-2-9-4-0-1-4-9-6-7-0-3-2-0-5-3-7-6
```

Pripomeňme si funkciu `sucin()`, ktorá počítala súčin ľubovoľného počtu čísel - tieto sa spracovali jedným zbaleným parametrom. Teda funkcia očakáva veľ'a parametrov a niečo z nich vypočíta. Ak mám ale jednu štruktúru, ktorá obsahuje tieto čísla, musím použiť rozbalovací operátor:

```
>>> cisla = [7, 11, 13]
>>> sucin(cisla)
[7, 11, 13]
>>> sucin(*cisla)
1001
```

```
>>> sulin(*range(2, 11))
3628800
```

## 2.4 Funkcia ako hodnota

v Pythone sú aj funkcie objektami a môžeme ich priradiť do premennej, napr.

```
>>> def fun1(x): return x*x
>>> fun1(7)
49
>>> cojaviem = fun1
>>> cojaviem(8)
64
```

Funkcie môžu byť prvками polí, napr.

```
>>> def fun2(x): return 2*x+1
>>> pole = [fun1, int, str, fun2]
>>> for f in pole:
...     print(f(3.14))
9.8596
3
3.14
7.28
```

Funkciu môžeme poslať ako parameter do inej funkcie, napr.

```
>>> def urob(fun, x):
...     return fun(x)
>>> urob(fun2, 3.14)
7.28
```

Často sa namiesto jednoriadkovej funkcie, ktorá počíta jednoduchý výraz a tento vráti ako výsledok (`return`) používa špeciálna konštrukcia `lambda`. Tá vygeneruje tzv. anonymnú funkciu, ktorú môžeme priradiť do premennej alebo poslať ako parameter do funkcie, napr.

```
>>> urob(lambda x: 2*x+1, 3.14)
7.28
```

Tvar konštrukcie `lambda` je nasledovný:

```
lambda parametre: výraz
```

Tento zápis nahradza definovanie funkcie:

```
def anonymne_meno(parametre):
    return výraz
```

Môžeme zapísat' napr.

```
lambda x: x%2==0          # funkcia vráti True pre párne čísla
lambda x,y: x**y          # vypočíta príslušné mocniny čísla
lambda x: isinstance(x, int) # vráti True pre celé čísla
```

## 2.5 Mapovacie funkcie

Idee funkcie ako parametra najlepšie ilustruje funkcia `mapuj()`:

```
def mapuj(fun, pole):
    vysl = []
    for prvok in pole:
        vysl.append(fun(prvok))
    return vysl
```

Funkcia aplikuje danú funkciu (prvý parameter) na všetky prvky poľa a z výsledkov poskladá nové pole, napr.

```
>>> mapuj(lambda x: [x]*x, range(1, 6))
[[1], [2, 2], [3, 3, 3], [4, 4, 4, 4], [5, 5, 5, 5]]
```

V Pythone existuje štandardná funkcia `map()`, ktorá robí skoro to isté ako naša funkcia `mapuj()` ale s tým rozdielom, že `map()` nevracia pole, ale niečo ako generátorový objekt, ktorý môžeme použiť ako prechádzanú postupnosť vo `for`-cykle, alebo napr. pomocou `list()` ho previesť na pole, napr.

```
>>> list(map(int, str(2**30)))
[1, 0, 7, 3, 7, 4, 1, 8, 2, 4]
```

Vráti pole cifier čísla  $2^{30}$ .

Podobná funkcia `mapuj()` je aj funkcia `filtruj()`, ktorá z daného poľa vyrobí nový nové pole, ale nechá len tie prvky, ktoré spĺňanú nejakú podmienku. Podmienka je definovaná funkciou, ktorá je prvým parametrom:

```
def filtruj(fun, pole):
    vysl = []
    for prvok in pole:
        if fun(prvok):
            vysl.append(prvok)
    return vysl
```

Napr.

```
>>> def podm(x): return x%2==0      # zistí, či je číslo párne
>>> list(range(1, 20, 3))
[1, 4, 7, 10, 13, 16, 19]
>>> mapuj(podm, range(1, 20, 3))
[False, True, False, True, False, True, False]
>>> filtruj(podm, range(1, 20, 3))
[4, 10, 16]
```

Podobne ako pre `mapuj()` existuje štandardná funkcia `map()`, aj pre `filtruj()` existuje štandardná funkcia `filter()`.

Ukážkovým využitím funkcie `map()` je funkcia, ktorá počíta ciferný súčet nejakého čísla:

```
def cs(cislo):
    return sum(map(int, str(cislo)))
```

```
>>> cs(1234)
10
```

## 2.6 Generátorová notácia

Veľmi podobná funkcií map() je generátorová notácia (po anglicky **list comprehension**):

- je to spôsob, ako môžeme elegantne vygenerovať nejaké pole pomocou for-cyklu a nejakého výrazu
- do [...] nezapíšeme prvky poľa, ale predpis, akým sa majú vytvoriť, základný tvar je tohto zápisu:

```
[vyraz for i in postupnosť]
```

- kde výraz najčastejšie obsahuje premennú cyklu a postupnosť' je ľubovoľná štruktúra, ktorá sa dá prechádzať for-cyklom (napr. list, set, str, range(), riadky otvoreného súboru, ale aj výsledok map() a filter() a pod.

- táto notácia môže používať aj vnorené cykly ale aj podmienku if, vtedy je to v takomto tvare:

```
[vyraz for i in postupnosť if podmienka]
```

- generátorová notácia s podmienkou nechá vo výsledku len tie prvky, ktoré splňajú danú podmienku

Niekoľko príkladov:

```
>>> [i**2 for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> [[i*j for i in range(1, 6)] for j in range(1, 6)]
[[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]]
>>> [i for i in range(100) if cs(i)==5]      # cs() vypočíta ciferný súčet
[5, 14, 23, 32, 41, 50]
```

Pomocou tejto konštrukcie sme vedeli zapísat aj mapovacie funkcie:

```
def mapuj(fun, pole):
    return [fun(prvok) for prvok in pole]

def filtruj(fun, pole):
    return [prvok for prvok in pole if fun(prvok)]
```

Všimnite si, že funkcia filtruj() využíva if, ktorý je vo vnútri generátorovej notácie.

## 2.7 Použitie v spájaných zoznamoch

Ked' už vieme generovať rôzne zadané postupnosti, môžeme to využiť aj pri vytváraní spájaných zoznamov, napr.

```
>>> Zoznam(i**2 for i in range(1, 11))
(1->4->9->16->25->36->49->64->81->100->)
```

V ďalšom príklade do spájaného zoznamu postupne vkladáme nové hodnoty, ktoré sú tiež spájané zoznamy:

```
z = Zoznam()
for i in range(5):
    pom = Zoznam(range(5))
    z.pridaj_kon(pom)
print(z)
```

dostaneme tento výpis:

```
((0->1->2->3->4->)->(0->1->2->3->4->)->(0->1->2->3->4->)->
->(0->1->2->3->4->)->)
```

Čo je 5-prvkový zoznam, ktorého prvkami sú opäť nejaké 5-prvkové zoznamy.

Niečo podobné by sme mohli zapísat aj generátorovou notáciou:

```
>>> z = Zoznam(Zoznam(range(i)) for i in range(1, 6))
>>> z
((0->)->(0->1->)->(0->1->2->)->(0->1->2->3->)->(0->1->2->3->4->)->)
```

Vytvorí 5-prvkový zoznam, ktorého prvkami sú tiež spájané zoznamy, ale rôzne veľké.

Aj pre spájané zoznamy môžeme vytvoriť mapovaciú metódu, ktorá sa bude podobat funkcie mapuj():

```
class Zoznam:

    ...

    def map(self, fun):
        zoz = self.zac
        while zoz is not None:
            zoz.data = fun(zoz.data)
            zoz = zoz.next
```

Pomocou tejto metódy môžeme meniť hodnoty naraz všetkým prvkom zoznamu, napr.

```
>>> z = Zoznam(range(1, 11))
>>> z
(1->2->3->4->5->6->7->8->9->10->
>>> z.map(str)
>>> z
('1'->'2'->'3'->'4'->'5'->'6'->'7'->'8'->'9'->'10'->
>>> z.map(lambda x: x+x)
>>> z
('11'->'22'->'33'->'44'->'55'->'66'->'77'->'88'->'99'->'1010'->
>>> z.map(int)
>>> z
(11->22->33->44->55->66->77->88->99->1010->)
```

---

## Zásobníky a rady

---

### 3.1 Stack - zásobník

V zimnom semestri sme ho realizovali pomocou poľa:

```
class EmptyError(Exception): pass

class Stack:
    def __init__(self):
        ''' inicializácia vnútornej reprezentácie '''
        self.pole = []

    def push(self, data):
        ''' vloží na vrch zásobníka novú hodnotu '''
        self.pole.append(data)

    def pop(self):
        ''' vyberie z vrchu zásobníka jednu hodnotu '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operaci pop()')
        return self.pole.pop()

    def is_empty(self):
        ''' zistí, či je zásobník prázdny '''
        return self.pole == []

    def top(self):
        ''' vráti z vrchu zásobníka jednu hodnotu '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operaci top()')
        return self.pole[-1]
```

#### 3.1.1 Zásobník pomocou spájaného zoznamu

Pre dátovú štruktúru **Stack** môžeme využiť aj spájané zoznamy (linked list), ktoré sme sa naučili na predchádzajúcej prednáške. Opäť si môžeme zvoliť (my ako programátori), na ktorom konci zoznamu bude **dno** a na ktorom **vrch** zásobníka. Asi by sme sa mali rozhodnúť podľa toho, ktoré operácie so spájaným zoznamom sa nám programujú ľahšie, resp. ktoré sú rýchlejšie. Z našich doterajších skúseností už vieme, že:

- pridávanie nového vrcholu na začiatok znamená len niekoľko priradení a teda je veľmi rýchle (nezávisí od momentálnej dĺžky zoznamu)

- aj odoberanie prvého prvku zoznamu sa dá realizovať len niekoľkými priradeniami a zrejme bude veľmi rýchle (tiež nezávisí od momentálnej dĺžky zoznamu)
- práca s prvkami na konci zoznamu môže byť časovo dosť náročná: pre dlhé zoznamy môže vyhľadanie posledného (resp. predposledného) vrcholu trvať dosť dlho, lebo musíme pomocou while-cyklu prejsť celý zoznam
- výrazné zrýchlenie vieme dosiahnuť, ak si pre spájaný zoznam pamäťame nielen referenciu na prvý vrchol ale aj na posledný (v atribúte `self.zac`): pridávanie nového vrcholu na koniec sa realizuje len pár priradeniami a nezávisí od momentálnej dĺžky zoznamu
- najhoršie je to s uberaním posledného prvku: tu nám referencia na posledný prvek nepomôže, my potrebujeme referenciu na predposledný a tú vieme získať len prejdením celého zoznamu od začiatku ... v našich programoch sa pokúsime vyvarovať tejto pomalej operácie, samozrejme, ak sa to bude dať

Zásobník budeme v spájanom zozname organizovať tak, že vrch bude na začiatku zoznamu a dno na konci. Vďaka tomu budú všetky operácie veľmi rýchle

```
class EmptyError(Exception): pass

class Stack:

    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    #-----

    def __init__(self):
        ''' inicializácia vnútornej reprezentácie '''
        self.zac = None

    def push(self, data):
        ''' vloží na vrch zásobníka novú hodnotu '''
        self.zac = self.Vrchol(data, self.zac)

    def pop(self):
        ''' vyberie z vrchu zásobníka jednu hodnotu '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operaci pop()')
        prvy = self.zac
        self.zac = self.zac.next
        return prvy.data

    def is_empty(self):
        ''' zistí, či je zásobník prázdný '''
        return self.zac is None

    def top(self):
        ''' vráti z vrchu zásobníka jednu hodnotu '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operaci top()')
        return self.zac.data
```

Pri definovaní triedy `Stack` sme "šíkovne" ukryli aj pomocnú triedu `Vrchol`, ktorá sa takto stala našou súkromnou (teda súkromnou triedou triedy `Stack`) a očakáva sa, že sa bude používať len vo vnútri tejto triedy. Teraz ale pri vytváraní nového vrcholu musíme zapísať `self.Vrchol(...)`.

## 3.2 Queue - rad, front

Rad pomocou poľa zo zimného semestra:

```
class EmptyError(Exception): pass

class Queue:
    def __init__(self):
        ''' inicializácia vnútornej reprezentácie '''
        self.pole = []

    def is_empty(self):
        ''' zistí, či je rad prázdny '''
        return self.pole == []

    def enqueue(self, data):
        ''' na koniec radu vloží novú hodnotu '''
        self.pole.append(data)

    def dequeue(self):
        ''' vyberie zo začiatku radu prvú hodnotu '''
        if self.is_empty():
            raise EmptyError('rad je prázdny')
        return self.pole.pop(0)

    def front(self):
        ''' vráti prvú hodnotu zo začiatku radu '''
        if self.is_empty():
            raise EmptyError('rad je prázdny')
        return self.pole[0]
```

### 3.2.1 Rad pomocou spájaného zoznamu

Definíciu štruktúry uložíme do súboru (napr. queue1.py):

```
class EmptyError(Exception): pass

class Queue:

    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

        #-----

        def __init__(self):
            ''' inicializácia vnútornej reprezentácie '''
            self.zac = self.kon = None

        def is_empty(self):
            ''' zistí, či je rad prázdny '''
            return self.zac is None

        def enqueue(self, data):
            ''' na koniec radu vloží novú hodnotu '''
            if self.zac is None:
                self.zac = self.kon = Vrchol(data)
            else:
                novy_vrchol = Vrchol(data, self.zac)
                self.zac = novy_vrchol
```

```
        self.zac = self.kon = self.Vrchol(data)
    else:
        self.kon.next = self.Vrchol(data)
        self.kon = self.kon.next

    def dequeue(self):
        '''vyberie zo začiatku radu prvú hodnotu'''
        if self.is_empty():
            raise EmptyError('rad je prázdný')
        prvy = self.zac
        self.zac = prvy.next
        if self.zac is None:
            self.kon = None
        return prvy.data

    def front(self):
        '''vráti prvú hodnotu zo začiatku radu'''
        if self.is_empty():
            raise EmptyError('rad je prázdný')
        return self.zac.data
```

### 3.2.2 Rad pomocou cyklického poľa

Definíciu štruktúry uložíme do súboru (napr. queue2.py):

```
class EmptyError(Exception): pass

class Queue:
    def __init__(self):
        ''' inicializácia vnútornej reprezentácie '''
        self.vel = 0
        self.pole = [None] * 10
        self.zac = 0

    def is_empty(self):
        ''' zistí, či je rad prázdný '''
        return self.vel == 0

    def enqueue(self, data):
        ''' na koniec radu vloží novú hodnotu '''
        if self.vel == len(self.pole):
            self.resize(2*len(self.pole))
        self.pole[(self.zac+self.vel) % len(self.pole)] = data
        self.vel += 1

    def dequeue(self):
        ''' vyberie zo začiatku radu prvú hodnotu '''
        if self.is_empty():
            raise EmptyError('rad je prázdný')
        data = self.pole[self.zac]
        self.zac = (self.zac+1) % len(self.pole)
        self.vel -= 1
        # ak treba, zmensi
        if self.vel < len(self.pole)//4 and len(self.pole)>10:
            self.resize(len(self.pole)//2)
        return data
```

```

def front(self):
    '''vráti prvú hodnotu zo začiatku radu'''
    if self.is_empty():
        raise EmptyError('rad je prázdný')
    return self.pole[self.zac]

def resize(self, nova_vel):
    pom = [None] * nova_vel
    for i in range(self.vel):
        pom[i] = self.pole[(self.zac+i) % len(self.pole)]
    self.pole = pom
    self.zac = 0

```

### 3.3 Vypĺňanie oblasti v dvojrozmernom poli

Budeme riešiť takýto problém: v štvorcovej sieti sú niektoré políčka obsadené - budeme ich kresliť modrou farbou. Úlohou je postupne od niektorého voľného políčka zafarbovať nielen toto ale aj všetkých jeho štyroch susedov a toto zafarbovanie d'alej šíriť, kým sa dá. Zadefinujeme pomocnú funkciu bodka(x, y, farba), ktorá danou farbou zafarbí jedno políčko siete a zároveň si do dvojrozmerného pol'a zaznačí, že je už obsadené. Zrejme obsadené políčka sú pre d'alšie šírenie zafarbovania prekážkou.

Rekurzívne riešenie môže padnúť na pretečení rekurzie:

```

from random import randrange as rr
import tkinter

d, n = 8, 50
c = tkinter.Canvas(bg='white', width=d*(n+1), height=d*(n+1))
c.pack()

def bodka(x, y, farba='blue'):
    c.create_oval(d*x, d*y, d*x+d, d*y+d, fill=farba, outline='')
    pole[x][y] = True

pole = [[0]*n for i in range(n)]

for i in range(n):
    for j in range(n):
        if rr(5) == 0 and (i!=n//2 or j!=n//2):
            bodka(i, j)

def vypln(x, y):
    if not 0<=x<=n or not 0<=y<=n or pole[x][y]:
        return
    bodka(x, y, 'red')
    c.update()
    c.after(10)
    vypln(x-1, y)
    vypln(x, y-1)
    vypln(x+1, y)
    vypln(x, y+1)

vypln(n//2, n//2)

```

Pre veľké pole sa aj rekurzia môže vnárať do veľkej hĺbky - v Pythone je to obmedzené 1000 vnoreniami - teda pre 50x50 políčok je šanca, že toto zafarbovanie sa nedokončí lebo spadne na pretečení rekurzie.

Rekurzívnu funkciu `vypln(x, y)` prerobíme pomocou zásobníka na nerekurzívnu verziu:

- do zásobníka budeme ukladať súradnice políčok štvorcovej siete, ktoré potrebujeme zafarbovať
- na začiatku do zásobníka vložíme súradnice štartového bodu (`x, y`)
- v cykle vyberieme súradnice bodu z vrchu zásobníka, ak sa dá zafarbiť, zafarbíme ho a do zásobníka vložíme súradnice všetkých jeho 4 susedov
- toto celé opakujeme, kým sú ešte nejaké súradnice v zásobníku (kým je zásobník neprázdný)

Riešenie pomocou zásobníka (predpokladáme, že je v súbore `stack1.py`):

```
from stack1 import Stack
from random import randrange as rr
import tkinter

d, n = 4, 200
c = tkinter.Canvas(bg='white', width=d*(n+1), height=d*(n+1))
c.pack()

def bodka(x, y, farba='blue'):
    c.create_oval(d*x, d*y, d*x+d, d*y+d, fill=farba, outline='')
    pole[x][y] = True

pole = [[0]*n for i in range(n)]

for i in range(n):
    for j in range(n):
        if rr(3) == 0 and (i!=n//2 or j!=n//2):
            bodka(i, j)
c.update()

def vypln(x, y):
    s = Stack()
    s.push((x, y))
    while not s.is_empty():
        x, y = s.pop()
        if 0<=x<n and 0<=y<n and not pole[x][y]:
            bodka(x, y, 'red')
            #c.update()
            s.push((x-1, y))
            s.push((x, y-1))
            s.push((x+1, y))
            s.push((x, y+1))

    vypln(n//2, n//2)
```

Pomocou radu (predpokladáme, že je v súbore `queue1.py`):

```
from queue1 import Queue
from random import randrange as rr
import tkinter

d, n = 4, 200
c = tkinter.Canvas(bg='white', width=d*(n+1), height=d*(n+1))
c.pack()
```

```

def bodka(x, y, farba='blue'):
    c.create_oval(d*x, d*y, d*x+d, d*y+d, fill=farba, outline=' ')
    pole[x][y] = True

pole = [[0]*n for i in range(n)]

for i in range(n):
    for j in range(n):
        if rr(3) == 0 and (i!=n//2 or j!=n//2):
            bodka(i, j)
c.update()

def vypln(x, y):
    q = Queue()
    q.enqueue((x, y))
    while not q.is_empty():
        x, y = q.dequeue()
        if 0<=x<n and 0<=y<n and not pole[x][y]:
            bodka(x, y, 'red')
            #c.update()
            q.enqueue((x-1, y))
            q.enqueue((x, y-1))
            q.enqueue((x+1, y))
            q.enqueue((x, y+1))

vypln(n//2, n//2)

```



---

## Binárne stromy

---

### 4.1 Všeobecný strom

Spájaná dátová štruktúra **strom** je zovšeobecnením spájaného zoznamu:

- skladá sa z množiny vrcholov, v každom vrchole sa nachádza nejaká informácia (atribút `data`)
- vrcholy sú navzájom pospájané hranami:
  - jeden vrchol má špeciálne postavenie: je prvý a od neho sa vieme dostat' ku všetkým zvyšným vrcholom
  - každý vrchol okrem prvého má práve jedného **predchodcu**
  - každý vrchol môže mať ľubovoľný počet nasledovníkov (nie iba jeden ako pri spájaných zoznamoch)

Zvykne sa používať takáto terminológia z **botaniky**:

- **koreň** (root) je špeciálny prvý vrchol - jediný nemá svojho predchodcu
- **vetva** (branch) je hrana (budeme ju kresliť šípkou) označujúca nasledovníka
- **list** (leaf) označuje vrchol, ktorý už nemá žiadneho nasledovníka

Okrem botanickej terminológie sa pri stromoch používa aj takáto terminológia **rodinných vzťahov**:

- predchodca každého vrcholu (okrem koreňa) sa nazýva **otec**, resp. **rodič**, **predok** (ancestor, parent)
- nasledovník vrcholu sa nazýva **syn**, resp. **dieťa** alebo **potomok** (child, descendant)
- vrcholy, ktoré majú spoločného predchodcu (otca) sa nazývajú **súrodenci** (siblings)

Ďalej sú dôležité ešte aj tieto pojmy:

- okrem listov sú v strome aj **vnútorné vrcholy** (interior nodes), to sú tie, ktoré majú aspoň jedného potomka
- medzi dvoma vrcholmi môžeme vytvárať **cesty**, t.j. postupnosti hrán, ktoré spájajú vrcholy - samozrejme, že len v smere od predkov k potomkom (v smere šípok) - **dĺžkou cesty** je počet týchto hrán (medzi otcom a synom je cesta dĺžky 1)
- **úroveň**, alebo **hĺbka** vrcholu (level, depth) je dĺžka cesty od koreňa k tomuto vrcholu, teda koreň má hĺbku 0, jeho synovia majú hĺbku 1, ich synovia (vnuci) ležia na úrovni 2, atď.
- **výška** stromu (height) je maximálna úroveň v strome, t.j. dĺžka najdlhšej cesty od koreňa k listom
- **podstrom** (subtree) je časť stromu, ktorá začína v nejakom vrchole a obsahuje všetkých jeho potomkov (nielen synov, ale aj ich potomkov, ...)
- **šírka** stromu (width) je počet vrcholov v úrovni, v ktorej je najviac vrcholov

Niekedy sa všeobecný strom definuje aj takto rekurzívne:

- strom je buď prázdný, alebo
- obsahuje koreň a množinu podstromov, ktoré vychádzajú z tohto koreňa, prečom každý podstrom je opäť všeobecný strom (má svoj koreň a svoje podstomy)

## 4.2 Binárny strom

má dve extra vlastnosti:

- každý vrchol má maximálne dvoch potomkov
- rozlišuje medzi ľavým a pravým synom - budeme to kresliť buď šípkou šikmo vľavo dole alebo šikmo vpravo dole

Aj binárny strom môžeme definovať rekurzívne:

- je buď prázdný,
- alebo sa skladá z koreňa a z dvoch jeho podstromov: z ľavého podstromu a pravého podstromu, tieto sú opäť binárne stromy

Všetko, čo platí pre všeobecné stromy, platí aj pre binárne, t.j. má koreň, má otcov a synov, má listy aj vnútorné vrcholy, má cesty aj úrovne, hovoríme o hĺbke vrcholov aj výške stromu.

Zamyslime sa nad maximálnym počtom vrcholov v jednotlivých úrovniach:

- v 0. úrovni môže byť len koreň, teda len **1 vrchol**
- koreň môže mať dvoch synov, teda v 1. úrovni môžu byť maximálne **2 vrcholy**
- v 2. úrovni môžu byť len synovia predchádzajúcich dvoch vrcholov v 1. úrovni, teda maximálne **4 vrcholy**
- v každej ďalšej úrovni môže byť maximálne dvojnásobok predchádzajúcej, teda v i-tej úrovni môže byť maximálne  **$2^{**}i$  vrcholov**
- strom, ktorý má výšku **h** môže mať maximálne  **$1 + 2 + 4 + 8 + \dots + 2^{**}h$  vrcholov**, teda spolu  **$2^{**}(h+1)-1$  vrcholov**
  - strom s maximálnym počtom vrcholov s hĺbkou **h** sa nazýva **úplný strom**

## 4.3 Realizácia spájanou štruktúrou

Binárny strom budeme realizovať podobne ako spájaný zoznam, t.j. najprv definujeme triedu `Vrchol`, v ktorej definujeme atribúty každého prvku binárneho stromu:

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

Na rozdiel od spájaného zoznamu, tento vrchol binárneho stromu má namiesto jedného nasledovníka `next` dvoch nasledovníkov: ľavého `left` a pravého `right`. Vytvorime najprv 3 izolované vrcholy:

```
>>> a = Vrchol('A')  
>>> b = Vrchol('B')  
>>> c = Vrchol('C')
```

Tieto tri vrcholy môžeme chápať ako 3 jednovrcholové stromy. Pomocou priradení môžeme pripojiť stromy b a c ako ľavý a pravý podstrom a:

```
>>> a.left = b
>>> a.right = c
```

Takto vytvorený binárny strom má 3 vrcholy, z toho 2 sú listy a jeden (koreň) je vnútorný. Výška stromu je 1 a šírka 2. V 0. úrovni je jeden vrchol 'A', v 1. úrovni je 'B' a 'C'.

Takýto istý strom vieme vytvoriť jediným priradením:

```
>>> a = Vrchol('A', Vrchol('B'), Vrchol('C'))
```

Ak chceme teraz vypísat hodnoty vo vrcholoch tohto stromu, môžeme to urobiť, napr. takto:

```
>>> print(a.data, a.left.data, a.right.data)
A B C
```

Ak by bol strom trochu komplikovanejší, výpsi vrcholov by bolo dobre zautomatizovať. Napr. pre takýto strom:

```
>>> a = Vrchol('A', Vrchol('B', Vrchol(1), Vrchol(2)), Vrchol('C'
    ↪ ', None, Vrchol(3)))
```

Tento strom so 6 vrcholmi má už výšku 2, pričom v druhej úrovni sú tri vrcholy: 1, 2 a 3.

## 4.4 Nakreslenie stromu

Strom nakreslíme rekurzívne. Prvá verzia bez kreslenia čiar:

```
import tkinter
canvas = tkinter.Canvas(bg='white', width=400, height=400)
canvas.pack()

def kresli(v, sir, x, y):
    if v is None:
        return
    canvas.create_text(x, y, text=v.data)
    kresli(v.left, sir//2, x-sir//2, y+40)
    kresli(v.right, sir//2, x+sir//2, y+40)

kresli(strom, 200, 200, 40)
```

Prvým parametrom funkcie je koreň stromu (teda referencia na najvrchnejší vrchol stromu). Druhým parametrom je polovičná šírka grafickej plochy. Ďalšie dva parametre sú súradnicami, kde sa nakreslí koreň stromu. Všimnite si, že v rekurzívnom volaní:

- smerom vľavo budeme vykresľovať ľavý podstrom, preto x-ovú súradnicu znížime o polovičnú šírku plochy, y-ovú zvýšime o nejakú konštantu, napr. 40
- smerom vpravo budeme kresliť pravý podstrom a teda x-ovú súradnicu zväčšíme tak, aby bola v polovici šírky oblasti

Ak chceme kesliť aj hrany stromu (spojnice medzi vrcholmi), zapíšeme to napr. takto (táto druhá verzia má ešte chyby):

```
import tkinter
canvas = tkinter.Canvas(bg='white', width=400, height=400)
```

```
canvas.pack()

def kresli(v, sir, x, y):
    if v is None:
        return
    canvas.create_text(x, y, text=v.data)
    if v.left is not None:
        canvas.create_line(x, y, x-sir//2, y+40)
        kresli(v.left, sir//2, x-sir//2, y+40)
    if v.right is not None:
        canvas.create_line(x, y, x+sir//2, y+40)
        kresli(v.right, sir//2, x+sir//2, y+40)
```

Takéto vykresľovanie stromu má ale tú chybu, že nakreslené hrany (čiary) prechádzajú cez vypísané hodnoty vo vrcholoch. Opravíme to tak, že nakreslenie samotného vrcholu (bude to teraz farebný krúžok s textom) prestahuje až na koniec funkcie, teda až potom, keď sa vykreslia čiary (vychádzajúce z tohto vrcholu aj vchádzajúce do tohto vrcholu):

```
import tkinter
canvas = tkinter.Canvas(bg='white', width=400, height=400)
canvas.pack()

def kresli(v, sir, x, y):
    if v is None:
        return
    if v.left is not None:
        canvas.create_line(x, y, x-sir//2, y+40)
        kresli(v.left, sir//2, x-sir//2, y+40)
    if v.right is not None:
        canvas.create_line(x, y, x+sir//2, y+40)
        kresli(v.right, sir//2, x+sir//2, y+40)
    canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray', outline='')
    canvas.create_text(x, y, text=v.data, font='consolas 12 bold')
```

Zrejme neskôr si budete túto verziu prispôsobovať:

- kresliť strom na iný rozmer plochy
- niektoré farebné kolečká môžete prefarbiť podľa obsahu, resp. polohy vrcholu
- hrany sa môžu kresliť ako šípky a nie ako úsečky

## 4.5 Generovanie nahodného stromu

Využijeme náhodný generátor `random.randrange`. Princíp tejto funkcie je nasledovný:

- pridávať budeme len do neprázdnego stromu (strom musí obsahovať aspoň koreň)
- funkcia sa najprv rozhodne, či bude pridávať do ľavého alebo do pravého podstromu (test `rr(2) == 0` testuje, či má náhodné číslo z množiny {0, 1} hodnotu 0, teda je to pravdepodobnosť 50%)
- ďalej zistí, či daný podstrom existuje (napr. `vrch.left is None` označuje, že neexistuje), ak nie, tak na jeho mieste vytvorí nový vrchol s danou hodnotou
- ak podstrom na tomto mieste už existuje, znamená to, že tu nemôžeme "zavesiť" nový vrchol, ale musíme preň hľadať nové miesto, preto sa (rekurzívne) zavolá pridávanie vrcholu do tohto podstromu

```

from random import randrange as rr

def pridaj_vrchol(vrch, hodnota):
    if rr(2) == 0:
        if vrch.left is None:
            vrch.left = Vrchol(hodnota)
        else:
            pridaj_vrchol(vrch.left, hodnota)
    else:
        if vrch.right is None:
            vrch.right = Vrchol(hodnota)
        else:
            pridaj_vrchol(vrch.right, hodnota)

koren = Vrchol(rr(20))
for h in range(20):
    pridaj_vrchol(koren, rr(20))

```

## 4.6 Ďalšie rekurzívne funkcie

Tieto funkcie postupne (rekurzívne) prechádzajú všetky vrcholy v strome: najprv v ľavom podstrome a potom aj v pravom.

Uvádzame dve verzie súčtu hodnôt vo vrcholoch. Prvá verzia, keď zistí, že je strom neprázdny, vypočíta súčet najprv v ľavom podstrome a potom aj v pravom. Výsledok celej funkcie je potom súčet hodnoty v korení stromu + súčet všetkých hodnôt v ľavom podstrome + súčet všetkých hodnôt v pravom podstrome:

```

def sucet(vrch):
    if vrch is None:
        return 0
    vlavo = sucet(vrch.left)
    vpravo = sucet(vrch.right)
    return vrch.data + vlavo + vpravo

print('sucet =', sucet(koren))

```

Skúsenejší programátori to zapisujú aj takto “úspornejšie”:

```

def sucet(vrch):
    if vrch is None:
        return 0
    return vrch.data + sucet(vrch.left) + sucet(vrch.right)

```

Na veľmi podobnom princípe pracuje aj zisťovanie celkového počtu vrcholov v strome:

```

def pocet(vrch):
    if vrch is None:
        return 0
    return 1 + pocet(vrch.left) + pocet(vrch.right)

print('pocet =', pocet(koren))

```

Ďalej budeme zisťovať výšku stromu (opäť bude niekoľko verzií). Najprv verzia, ktorá ešte nie je správna:

```
def vyska(vrch):
    if vrch is None:
        return 0
    vyska_vlavo = vyska(vrch.left)
    vyska_vpravo = vyska(vrch.right)
    return 1 + max(vyska_vlavo, vyska_vpravo)
```

Funkcia najprv rieši situáciu, keď je strom prázdny: vtedy dáva výsledok **0**. Inak sa vypočíta výška ľavého podstromu, potom výška pravého a na záver sa z týchto dvoch výšok vyberie ta väčšia a k tomu sa ešte pripočíta **1**, lebo celkový strom okrem dvoch podstromov obsahuje aj koreň, ktorý je o úroveň vyššie.

Táto funkcia ale nedáva správne výsledky. Keď ju otrestujete, zistíte, že dostávate o **1** väčšiu hodnotu, ako je skutočná výška. Napr.

```
>>> strom = Vrchol(1)
>>> vyska(strom)
1
>>> strom = Vrchol(1, Vrchol(2), Vrchol(3))
>>> vyska(strom)
2
```

Pripomíname, že výška je počet hrán na ceste od koreňa k najnižšiemu vrcholu (k nejakému listu). Strom, ktorý obsahuje len koreň, by mal mať výšku 0 a strom, ktorý má koreň a 2 jeho synov, má dve takéto cesty (od koreňa k listom) a obe majú dĺžku 1 (len 1 hranu). Problémom v tomto riešení je výška prázdneho stromu: tá nemôže byť **0**, lebo **0** je pre strom s 1 vrcholom. Dohodneme sa, že výškou prázdneho stromu bude **-1** a potom to už bude celé fungovať správne:

```
def vyska(vrch):
    if vrch is None:
        return -1
    vyska_vlavo = vyska(vrch.left)
    vyska_vpravo = vyska(vrch.right)
    return 1 + max(vyska_vlavo, vyska_vpravo)
```

alebo druhá verzia, ktorá si výšky jednotlivých podstromov nepočíta do pomocných premenných, ale priamo ich pošle do štandardnej funkcie `max()`:

```
def vyska(vrch):
    if vrch is None:
        return -1           # pre prázdny strom
    return 1 + max(vyska(vrch.left), vyska(vrch.right))

print('vyska =', vyska(koren))
```

Výpis hodnôt vo vrcholoch v nejakom poradí:

```
def vypis(vrch):
    if vrch is None:
        return
    print(vrch.data, end=' ')
    vypis(vrch.left)
    vypis(vrch.right)

vypis(koren)
```

Vo výpise sa objavia všetky hodnoty vo vrchol v nejakom poradí.

---

## Trieda BinarnyStrom

---

Doteraz sme pracovali so stromom tak, že sme zadefinovali triedu `Vrchol` pre jeden vrchol stromu a k tomu sme zadefinovali niekoľko funkcií (väčšinou rekurzívnych), ktoré s pracovali s celým stromom. Napr.

```
def vypis(vrch):
    if vrch is None:
        return
    print(vrch.data, end=' ')
    vypis(vrch.left)
    vypis(vrch.right)

vypis(koren)
```

Vypíše všetky hodnoty vo vrchol stromu v nejakom poradí.

Prirodzenejšie a programátorsky správnejšie by bolo zadefinovanie triedy `Strom`, ktorá okrem definície jedného vrcholu bude obsahovať aj všetky užitočné funkcie ako svoje metódy. Teda zapuzdrime všetky funkcie spolu s dátovou štruktúrou do jedného “obalu”. Pre prístup ku stromu (k jeho vrcholom) si musíme pamätať referenciu na jeho koreň - to bude atribút `self.root`, ktorý bude mať pri inicializácii hodnotu `None`. Zapíšme základ:

```
class BinarnyStrom:

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

        def __repr__(self):
            return repr(self.data)

    #-----

    def __init__(self):
        self.root = None
```

Na minulej prednáške sme zostavili funkciu, ktorá počíta počet prvkov (vrcholov) stromu:

```
def pocet(vrch):
    if vrch is None:
        return 0
    return 1 + pocet(vrch.left) + pocet(vrch.right)
```

Ked'že táto funkcia očakáva ako vstup vrchol, od ktorého sa počíta počet vrcholov (podstromu), musíme aj metódu, ktorá zist'uje počet prvkov, napr. `__len__()`, zapísat' inak. Môžeme túto globálnu funkciu volať z metódy v triede:

```
class BinarnyStrom:  
    ...  
  
    def __len__(self):  
        return pocet(self.root)
```

Toto je veľmi nevhodný spôsob, lebo okrem triedy sa musíme starat' aj o nejakú globálnu funkciu - prípadne ich bude aj viac takýchto. Všetky pomocné funkcie by mali byť tiež zapuzdrené v triede, najlepšie ako vnorené funkcie tam, kde sa budú používať:

```
class BinarnyStrom:  
    ...  
  
    def __len__(self):  
  
        def pocet(vrch):  
            if vrch is None:  
                return 0  
            return 1 + pocet(vrch.left) + pocet(vrch.right)  
  
        return pocet(self.root)
```

Takýmto zápisom sa funkcia `pocet()` stáva lokálou (vnorenou) do metódy `__len__()` a nik okrem nej ju nemôže používať. Takýto spôsob je najbežnejší pri definovaní metód, ktoré sú rekurzívne. Prepíšme niektoré ďalšie funkcie ako metódy tejto triedy:

```
def suchet(vrch):  
    if vrch is None:  
        return 0  
    return vrch.data + suchet(vrch.left) + suchet(vrch.right)  
  
def vyska(vrch):  
    if vrch is None:  
        return -1 # pre prázdný strom  
    return 1 + max(vyska(vrch.left), vyska(vrch.right))  
  
def nachadza_sa(vrch, hodnota):  
    if vrch is None:  
        return False  
    if vrch.data == hodnota:  
        return True  
    if nachadza_sa(vrch.left, hodnota):  
        return True  
    if nachadza_sa(vrch.right, hodnota):  
        return True  
    return False
```

Ako metódy:

```
class BinarnyStrom:  
  
    class Vrchol:  
        def __init__(self, data, left=None, right=None):
```

```

        self.data = data
        self.left = left
        self.right = right

    def __repr__(self):
        return repr(self.data)

    #-----

    def __init__(self):
        self.root = None

    def __len__(self):
        def pocet(vrch):
            if vrch is None:
                return 0
            return 1 + pocet(vrch.left) + pocet(vrch.right)
        return pocet(self.root)

    def sucet(self):
        def sucet_rek(vrch):
            if vrch is None:
                return 0
            return vrch.data + sucet_rek(vrch.left) + sucet_rek(vrch.right)
        return sucet_rek(self.root)

    def vyska(self):
        def vyska_rek(vrch):
            if vrch is None:
                return -1
            return 1 + max(vyska_rek(vrch.left), vyska_rek(vrch.right))
        return vyska_rek(self.root)

    def __contains__(self, hodnota):
        def nachadza_sa(vrch):
            if vrch is None:
                return False
            return vrch.data == hodnota or nachadza_sa(vrch.left) or_
        ↵nachadza_sa(vrch.right)
        return nachadza_sa(self.root)

```

Všimnite si ako sme zjednodušili poslednú z metód, ktorá zistíuje, či sa nejaká hodnota nachádza niekde v strome:

- vnorená pomocná funkcia `nachadza_sa(vrch)` nemá parameter `hodnota` - keďže premenná `hodnota` je lokálnou premennou v nadradenej funkcií, v ktorej je `nachadza_sa()` vnorená, táto vnorená ju vidí (keďže ju nemodifikuje)
- namiesto troch za sebou idúcich `if`-príkazov, sme zapísali jeden logický výraz, v ktorom sú podvýrazy spojené logickou operáciou `or` - toto označuje, že sa takýto výraz bude vyhodnocovať zľava doprava:
  - najprv `vrch.data == hodnota`: ak je to `True`, ďalšie podvýrazy sa nevyhodnocujú a celkový výsledok je `True`, inak sa spustí vyhodnocovanie `nachadza_sa(vrch.left)`
  - `nachadza_sa(vrch.left)` spustí rekurzívne zistovanie, či sa daná hodnota nachádza v ľavom podstrome: ak áno celkový výsledok je `True` inak sa ešte spustí posledný podvýraz `nachadza_sa(vrch.right)`
  - `nachadza_sa(vrch.right)` spustí rekurzívne zistovanie, či sa daná hodnota nachádza v pravom podstrome: ak áno celkový výsledok je `True` inak `False`

Metódu sme nazvali `__contains__()`, vďaka čomu ju môžeme volať nielen takto:

```
>>> strom.__contains__(123)
True
```

ale aj

```
>>> 123 in strom
True
```

Doplňme ešte náhodné generovanie stromu a vykresľovanie do grafickej plochy:

```
from random import randrange as rr
import tkinter

class BinarnyStrom:
    canvas = None

    ...

    def pridaj_nahodne(self, *pole):

        def pridaj_vrchol(vrch):
            if rr(2) == 0:
                if vrch.left is None:
                    vrch.left = self.Vrchol(hodnota)
                else:
                    pridaj_vrchol(vrch.left)
            else:
                if vrch.right is None:
                    vrch.right = self.Vrchol(hodnota)
                else:
                    pridaj_vrchol(vrch.right)

        for hodnota in pole:
            if self.root is None:
                self.root = self.Vrchol(hodnota)
            else:
                pridaj_vrchol(self.root)

    def kresli(self):

        def kresli_rek(v, sir, x, y):
            if v is None:
                return
            if v.left is not None:
                self.canvas.create_line(x, y, x-sir//2, y+40)
                kresli_rek(v.left, sir//2, x-sir//2, y+40)
            if v.right is not None:
                self.canvas.create_line(x, y, x+sir//2, y+40)
                kresli_rek(v.right, sir//2, x+sir//2, y+40)
            self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray')
            self.canvas.create_text(x, y, text=v.data, font='consolas 12 bold')

        if self.canvas is None:
            BinarnyStrom.canvas = tkinter.Canvas(bg='white', width=800, height=400)
```

```

        self.canvas.pack()
        self.canvas.delete('all')
        kresli_rek(self.root, 380, 400, 40)
    
```

Otestujeme, napr.

```

>>> strom = BinarnyStrom()
>>> strom.pridaj_nahodne(*range(20))
>>> strom.kresli()
    
```

pridáme ďalšie dva vrcholy:

```

>>> strom.pridaj_nahodne('A', 'B')
>>> strom.kresli()
    
```

## 5.1 Prechádzanie vrcholov stromu

Globálnu funkciu `vypis()`, ktorá v nejakom poradí vypisuje hodnoty vo všetkých vrcholoch:

```

def vypis(vrch):
    if vrch is None:
        return
    print(vrch.data, end=' ')
    vypis(vrch.left)
    vypis(vrch.right)
    
```

prepíšme ako metódu stromu a otestujme:

```

class BinarnyStrom:
    ...

    def vypis_preorder(self):

        def vypis(vrch):
            if vrch is None:
                return
            print(vrch.data, end=' ')
            vypis(vrch.left)
            vypis(vrch.right)

        vypis(self.root)
        print()
    
```

Vytvorili sme metódu `vypis_preorder()`, ktorá v svojom tele opäť obsahuje len tri príkazy: definíciu vnorenej (lokálnej) funkcie `vypis()`, jej zavolanie s referenciou na koreň stromu a ukončenie vypisovaného riadka. Táto metóda (podobne ako skoro všetky doterajšie) obíde všetky vrcholy v strome tak, že najprv spracuje koreň stromu (vypíše ho príkazom `print()`), potom rekúrživne celý ľavý podstrom a na záver celý pravý podstrom. Tomuto budeme hovoriť poradie spracovania **preorder**. Vo všeobecnosti preorder poradie zapíšeme:

```

class BinarnyStrom:
    ...

    def preorder(self):

        def preorder1(vrch):
            ...
    
```

```
    if vrch is None:
        return
    # spracuj samotný vrchol vrch
    preorder1(vrch.left)
    preorder1(vrch.right)

preorder1(self.root)
```

alebo to isté zapísané trochu inak:

```
class BinarnyStrom:
    ...

    def preorder(self):
        def preorder1(vrch):
            # spracuj samotný vrchol vrch
            if vrch.left is not None:
                preorder1(vrch.left)
            if vrch.right is not None:
                preorder1(vrch.right)

            if self.root is not None:
                preorder1(self.root)
```

Závisí od riešeného problému, ktorú z týchto verzií použijete. Všimnite si, že aj metódy `__len__()`, `vyska()`, `__contains__()`, `sucet()` boli tvaru `preorder`.

Pre nás je zaujímavý výpis z tohto preorder poradia: tento výpis závisí od tvaru binárneho stromu a bude užitočné, keď to budete vedieť nasimulovať aj ručne.

Okrem poradia `preorder` poznáme ešte tieto základné poradia obchádzania vrcholov stromu:

- **inorder** - najprv ľavý podstrom, potom samotný vrchol a na záver pravý podstrom
- **postorder** - najprv ľavý podstrom, potom pravý podstrom a na záver samotný vrchol
- **po úrovniach** - vrcholy sa spracovávajú v poradí ako sú vzdialené od koreňa, t.j. najprv koreň, potom obaja jeho synovia, potom všetci vnuci, atď.

Zapíšme prvé dve metódy:

```
class BinarnyStrom:
    ...

    def vypis_inorder(self):
        def vypis(vrch):
            if vrch is None:
                return
            vypis(vrch.left)
            # spracuj samotný vrchol vrch
            print(vrch.data, end=' ')
            vypis(vrch.right)

        vypis(self.root)
        print()

    def vypis_postorder(self):
```

```

def vypis(vrch):
    if vrch is None:
        return
    vypis(vrch.left)
    vypis(vrch.right)
    # spracuj samotný vrchol vrch
    print(vrch.data, end=' ')
    vypis(self.root)
    print()

```

Otestujte, ako vyzerajú tieto výpisy pre náhodne generovaný strom:

```

s = BinarnyStrom()
s.pridaj_nahodne(*range(10))
print('preorder = ', end=' ')
s.vypis_preorder()
print('inorder = ', end=' ')
s.vypis_inorder()
print('postorder = ', end=' ')
s.vypis_postorder()

```

Tieto metódy by sa dali prepísať ako funkcie, ktoré vrátia znakový reťazec alebo pole, napr.

```

class BinarnyStrom:
    ...

    def preorder_str(self):
        def retazec(vrch):
            if vrch is None:
                return ''
            return str(vrch.data) + ' ' + retazec(vrch.left) + retazec(vrch.
        ↵right)

        return retazec(self.root)

    def preorder_list(self):
        def urob_pole(vrch):
            if vrch is None:
                return []
            return [vrch.data] + urob_pole(vrch.left) + urob_pole(vrch.right)

        return urob_pole(self.root)

```

Tieto algoritmy postupného prechádzania všetkých vrcholov v nejakom poradí sa môžu využiť napr. na postupnú zmenu hodnôt vo vrcholoch stromu. Použijeme počítadlo, ktorého hodnotu budeme pri prechádzaní stromu pridať ovať a za každým ho budeme zvyšovať o 1. Toto počítadlo ale nemôže byť obyčajná lokálna premenná, lebo ju chceme meniť počas behu rekurzie a chceme, aby sa táto zmenená hodnota zapamätať. Preto môžeme počítadlo vypočítať ako atribút triedy, s ktorým už tento problém nebude.

Nasledovná metóda ilustruje spôsob očíslovania vrcholov v strome v poradí preorder, koreň bude mať hodnotu 1:

```

class BinarnyStrom:
    ...

    def ocisluj(self):

```

```
def ocisluj1(vrch):
    if vrch is None:
        return
    vrch.data = self.cislo
    self.cislo += 1
    ocisluj1(vrch.left)
    ocisluj1(vrch.right)

    self.cislo = 1
    ocisluj1(self.root)
```

Veľmi často pri podobných úlohách existuje niekoľko veľmi rozdielnych spôsobov riešení. Toto isté môžeme dosiahnuť aj inak, napr. tak, že počítadlom nebude atribút (stavová premenná inštancie) ale ďalší parameter rekurzívnej vnorenej funkcie. V tomto prípade, ale budeme od tejto pomocnej funkcie vyžadovať, aby nám oznamila, kol'ko čísel z počítadla v danom podstrome už minula. Inými slovami, funkcia bude vraciať hodnotu počítadla, na ktorom skončila pri číslovaní vrcholov v podstrome:

```
class BinarnyStrom:
    ...

    def ocisluj(self):
        def ocisluj1(vrch, cislo):
            if vrch is None:
                return cislo
            vrch.data = cislo
            cislo = ocisluj1(vrch.left, cislo+1)
            return ocisluj1(vrch.right, cislo)

        ocisluj1(self.root, 1)
```

Takže vnorená funkcia najprv očísluje vrchol, v ktorom sa nachádza (lebo je to preorder), potom očísluje vrcholy v ľavom podstrome a dozvie sa (ako výsledok funkcie) s akým číslom sa bude pokračovať v pravom podstrome. Na záver vráti novú hodnotu počítadla ako výsledok funkcie.

## 5.2 Prechádzanie po úrovniach

Tento algoritmus bude používať dátovú štruktúru **queue** (rad, front) takto:

- na začiatku vyrobíme prázdný front a vložíme do neho koreň stromu (čo je referencia na vrchol)
- potom sa v cykle bude robiť toto:
  - z frontu sa vyberie prvý čakajúci vrchol
  - spracuje sa tento vrchol, napr. sa pomocou `print()` vypíše jeho hodnota
  - na koniec frontu sa vložia obaja synovia spracovávaného vrcholu
- po skončení cyklu (práve boli spracované všetky vrcholy) sa môže urobiť nejaký záverečný úkon, napr. ukončiť rozpisáný riadok s výpisom vrcholov

Metóda, ktorá vypisuje hodnoty vo vrcholoch, ale prechádza strom po úrovniach, može vyzeráť takto:

```
class BinarnyStrom:
    ...
```

```

def vypis_levelorder(self) :
    if self.root is None:
        return
    q = [self.root]                      # q = Queue(); q.enqueue(self.root)
    vypis = 0
    while q != []:
        vrch = q.pop(0)                  # while not q.is_empty():
                                         # vrch = q.dequeue()
        #spracuj
        print(vrch.data, end=' ')
        if vrch.left is not None:
            q.append(vrch.left)          # q.enqueue(vrch.left)
        if vrch.right is not None:
            q.append(vrch.right)         # q.enqueue(vrch.right)
    print()

```

V zápisе funkcie môžete vidieť, aké operácie s frontom sme nahradili operáciami s obyčajným poľom. Už vieme, že operácia `pop(0)` je dosť neefektívna, ale v tomto prípade to nemusíme.

Ďalšia verzia pridáva do výpisu informáciu o konkrétnej úrovni - vrcholy, ktoré sú v rovnakej úrovni sa vypisujú do riadku a na nový riadok sa prejde vtedy, keď spracovávame vrchol z vyšej úrovne ako doteraz. Idea v tejto funkcií je taká, že do frontu okrem samotného vrcholu z predchádzajúcej verzie budeme vkladať aj číslo úrovne. Preto, pri vyberaní vrcholu z frontu, získavame aj jeho úroveň. Tiež musíme túto úroveň ukladať do frontu, keď tam vkladáme nové vrcholy (synov momentálneho vrcholu):

```

class BinarnyStrom:
    ...

def vypis_levelorder1(self) :
    if self.root is None:
        return
    q = [(self.root, 0)]
    vypis = 0
    while q != []:
        vrch, uroven = q.pop(0)
        #spracuj
        if vypis != uroven:
            print()
        print(vrch.data, end=' ')
        vypis = uroven
        if vrch.left is not None:
            q.append((vrch.left, uroven+1))
        if vrch.right is not None:
            q.append((vrch.right, uroven+1))
    print()

```

Obe tieto metódy sú nerekurzívne. Toto je veľmi dôležitá vlastnosť tohto algoritmu (hovorí sa mu aj **algoritmus do šírky**). Pomocou idey týchto algoritmov vieme riešiť veľkú skupinu úloh so stromami, napr.

- zist'ovanie, v ktorej úrovni sa nachádza ten ktorý vrchol
- zist'ovanie **šírky** stromu - čo je maximum z počtu vrcholov v jednotlivých úrovniach
- zist'ovanie všetkých vrcholov, ktoré sú v jednej úrovni
- úlohy, ktoré sa dajú riešiť rekurzívne (napr. výška, počet vrcholov, apod.) ale ich nerekurzívne riešenie zabezpečí to, že takáto funkcia nespadne na pretečení rekurzie (čo je do 1000).



---

## Použitie stromov

---

V tejto časti sa zoznámime s niekoľkými zaujímavými aplikáciami stromov.

### 6.1 Binárne vyhľadávacie stromy

Skôr, ako ukážeme špeciálny typ stromu, pozrime sa na jeden problém:

- ak má nejaký binárny strom veľké množstvo vrcholov a my potrebujeme zistíť, či sa medzi nimi nevyskytuje nejaká konkrétna hodnota, neostáva nám nič iné, iba postupne “preliezť” a kontrolovať všetky vrcholy, pritom vieme, že v utriedenom poli vieme nájsť ľubovoľný prvok veľmi rýchlo (pomocou binárneho vyhľadávania)

Budeme vychádzať z implementácie binárneho stromu z predchádzajúcej prednášky (ponechali sme aj metódu kresli() na vykreslenie stromu do grafickej plochy):

```
from random import randrange as rr
import tkinter

class BinarnyStrom:
    canvas = None

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

        def __init__(self):
            self.root = None

    def pridaj_nahodne(self, *pole):

        def pridaj_vrchol(vrch):
            if rr(2) == 0:
                if vrch.left is not None:
                    pridaj_vrchol(vrch.left)
                else:
                    vrch.left = self.Vrchol(hodnota)
            else:
                if vrch.right is None:
                    vrch.right = self.Vrchol(hodnota)
                else:
                    pridaj_vrchol(vrch.right)

        for i in pole:
            pridaj_vrchol(self.root)
```

```
for hodnota in pole:
    if self.root is None:
        self.root = self.Vrchol(hodnota)
    else:
        pridaj_vrchol(self.root)

def kresli(self):

    def kresli_rek(vrch, sir, x, y):
        if vrch.left is not None:
            self.canvas.create_line(x, y, x-sir//2, y+40)
            kresli_rek(vrch.left, sir//2, x-sir//2, y+40)
        if vrch.right is not None:
            self.canvas.create_line(x, y, x+sir//2, y+40)
            kresli_rek(vrch.right, sir//2, x+sir//2, y+40)
            self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray')
            self.canvas.create_text(x, y, text=vrch.data)

        if self.canvas is None:
            BinarnyStrom.canvas = tkinter.Canvas(bg='white', width=800,
            height=400)
            self.canvas.pack()
            self.canvas.delete('all')
        kresli_rek(self.root, 380, 400, 40)

    def __len__(self):

        def pocet_rek(vrch):
            if vrch is None:
                return 0
            return 1 + pocet_rek(vrch.left) + pocet_rek(vrch.right)

        return pocet_rek(self.root)

    def preorder(self):

        def preorder_rek(vrch):
            if vrch is None:
                return ''
            return repr(vrch.data) + ' ' + preorder_rek(vrch.left) +_
            preorder_rek(vrch.right)

        return preorder_rek(self.root)

    def postorder(self):

        def postorder_rek(vrch):
            if vrch is None:
                return ''
            return postorder_rek(vrch.left) + postorder_rek(vrch.right) +_
            repr(vrch.data) + ' '

        return postorder_rek(self.root)

    def inorder(self):
```

```

def inorder_rek(vrch):
    if vrch is None:
        return ''
    return inorder_rek(vrch.left) + repr(vrch.data) + ' ' + inorder_
    rek(vrch.right)

return inorder_rek(self.root)

```

Vygenerujme malý náhodný strom:

```

strom = BinarnyStrom()
strom.pridaj_nahodne(*[rr(100) for i in range(20)])
strom.kresli()

```

Väčší počet vrcholov sa už zobrazuje veľmi neprehľadne.

Dopíšme metódu `je_prvkom()`, ktorá bude zistovať, či sa nejaká hodnota nachádza v strome (použijeme pomocnú vnorenú rekurzívnu funkciu):

```

class BinarnyStrom:
    ...
    def je_prvkom(self, hodnota):

        def je_prvkom_rek(vrch):
            if vrch is None:
                return False
            if hodnota == vrch.data:
                return True
            if je_prvkom_rek(vrch.left):
                return True
            if je_prvkom_rek(vrch.right):
                return True
            return False

        return je_prvkom_rek(self.root)

```

Otestujeme:

```

>>> strom.preorder()
'53 49 64 18 21 30 98 24 50 89 98 32 62 88 54 52 28 6 43 8 '
>>> strom.je_prvkom(43)
True
>>> strom.je_prvkom(44)
False
>>> for i in range(100):
    if strom.je_prvkom(i):
        print(i, end=' ')

```

6 8 18 21 24 28 30 32 43 49 50 52 53 54 62 64 88 89 98

Ak by sme túto metódu premenovali na `__contains__()`, namiesto `strom.__contains__(43)` by sme mohlo písat `43 in strom`.

Je zvykom takýto typ rekurzívnej funkcie zapisovať trochu kompaktnejšie:

```

class BinarnyStrom:
    ...
    def je_prvkom(self, hodnota):

```

```
def je_prvkom_rek(vrch):
    if vrch is None:
        return False
    return hodnota == vrch.data or je_prvkom_rek(vrch.left) or je_
→prvkom_rek(vrch.right)

return je_prvkom_rek(self.root)
```

V niektorých situáciách je vhodné ukladať nové hodnoty do stromu tak, aby sme ich neskôr vedeli čo najrýchlejšie nájsť. Jedna z možností je ukladať ich tak, aby sme sa na základe hodnoty v koreni vedeli jednoznačne rozhodnúť, či pokračujeme v hľadaní do ľavého alebo pravého podstromu. Ak by sme takéto pravidlo zabezpečili pre všetky vrcholy v strome, hľadanie by bolo jednoduché: kým nenájdeme, resp. už sa ďalej pokračovať nedá (prišli sme na `None`), budeme sa vrátať sa do ľavej alebo pravej strany stále hlbšie a hlbšie.

Takto organizovaný binárny strom sa nazýva **binárny vyhľadávací strom**, skrátene **BVS** (binary search tree ([http://en.wikipedia.org/wiki/Binary\\_search\\_tree](http://en.wikipedia.org/wiki/Binary_search_tree))) a má tieto vlastnosti (rekurzívna definícia):

- všetky hodnoty v ľavom podstrome sú menšie ako hodnota v koreni
- všetky hodnoty v pravom podstrome sú väčšie ako hodnota v koreni
- všetky podstromy sú tiež **BVS**, t.j. aj pre ne platí, že vľavo sú iba menšie hodnoty a vpravo iba väčšie, atď.

Vidíme, že v takomto strome sa rovnaká hodnota nemôže nachádzať v rôznych vrcholoch (v strome musia byť všetky hodnoty rôzne) - ak by sme to niekedy potrebovali, treba to zabezpečiť nejakou inak.

### 6.1.1 Vkladanie do BVS

Do existujúceho **BVS** musíme nové vrcholy vkladať podľa presných pravidiel:

- ak sa vkladaná hodnota už nachádza v koreni stromu, môžeme skončiť
- ak je vkladaná hodnota menšia ako hodnota v koreni, zrejme budem vkladať do ľavého podstromu (rekurzívne voláme vkladanie pre ľavý podstrom)
- ak je vkladaná hodnota väčšia ako hodnota v koreni, zrejme budem vkladať do pravého podstromu (rekurzívne voláme vkladanie pre pravý podstrom)
- ak takýto podstrom ešte neexistuje a my sme sa do neho chceli rekurzívne vnoríť, tak na jeho mieste vytvoríme nový vrchol aj s vkladanou hodnotou (podstrom s jedným vrcholom)

Zapišme teraz metódu `pridaj()`, ktorá je vlastne len miernou úpravou `pridaj_nahodne()` (premenujeme pri-tom meno triedy):

```
class BinarnyVyhladavaciStrom:
    ...
    def pridaj(self, hodnota):

        def pridaj_vrchol(vrch):
            if vrch.data == hodnota:
                return
            if vrch.data > hodnota:
                if vrch.left is not None:
                    pridaj_vrchol(vrch.left)
                else:
                    vrch.left = self.Vrchol(hodnota)
            else:
                if vrch.right is None:
```

```

        vrch.right = self.Vrchol(hodnota)
    else:
        pridaj_vrchol(vrch.right)

    if self.root is None:
        self.root = self.Vrchol(hodnota)
    else:
        pridaj_vrchol(self.root)

```

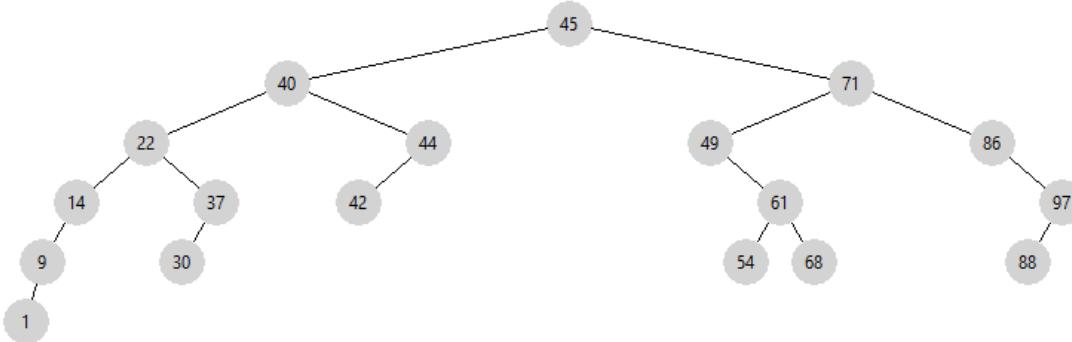
Otestujeme (pridali sme malé pozdržanie, aby sme mohli sledovať efekt postupného vytvárania stromu):

```

strom = BinarnyVyhladavaciStrom()
for i in range(20):
    strom.pridaj(rr(100))
    strom.kresli()
    strom.canvas.update()
    strom.canvas.after(300)

```

a dostávame napr. takýto strom:



Naša prvá verzia metódy `pridaj()` je rekúzívna, pritom rekúzia je tu úplne zbytočná: v tomto algoritme sa len postupne vnáram do stromu hlbšie a hlbšie, až kým nenašrame bud' na vkladanú hodnotu, alebo na prázdne miesto (`None`). Keď pri vnáraní narazíme na `None`, pridáme sem nový vrchol a tým vkladanie vrcholu do stromu skončíme. Nerekúzívna verzia je ešte jednoduchšia ako rekúzívna, môžeme ju zapísat' napr. takto:

```

class BinarnyVyhladavaciStrom:
    ...
    def pridaj(self, hodnota):
        if self.root is None:
            self.root = self.Vrchol(hodnota)
        else:
            vrch = self.root
            while vrch.data != hodnota:
                if vrch.data > hodnota:
                    if vrch.left is not None:
                        vrch = vrch.left
                    else:
                        vrch.left = self.Vrchol(hodnota)
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                    else:
                        vrch = vrch.right

```

## 6.1.2 Hľadanie v BVS

Hľadanie hodnoty v BVS (napr. metódou `je_prvkom()`) bude analogické vkladaniu novej hodnoty. Najprv zapíšme rekurzívnu verziu:

```
class BinarnyVyhladavaciStrom:
    ...
    def je_prvkom(self, hodnota):
        def je_prvkom_rek(vrch):
            if vrch is None:
                return False
            if hodnota == vrch.data:
                return True
            if hodnota < vrch.data:
                return je_prvkom_rek(vrch.left)
            else:
                return je_prvkom_rek(vrch.right)

        return je_prvkom_rek(self.root)
```

Aj táto rekurzívna verzia sa veľmi jednoducho prepíše na while-cyklus:

```
class BinarnyVyhladavaciStrom:
    ...
    def je_prvkom(self, hodnota):
        vrch = self.root
        while vrch is not None:
            if hodnota == vrch.data:
                return True
            if hodnota < vrch.data:
                vrch = vrch.left
            else:
                vrch = vrch.right
        return False
```

Pri tomto algoritme si uvedomte, že počet prechodov tohto while-cyklu nebude väčší ako výška stromu. Takže, ak máme zostavený **BVS** napr. s 1000000 vrcholmi, jeho výška bude väčšinou okolo 50. To znamená, že v tomto strome s milión rôznymi hodnotami vieme nájsť hľadanú hodnotu na maximálne 50 prechodov while-cyklu, t.j. kontrolovaním maximálne 50 vrcholov. Kým to nebol **BVS**, skontrolovať sme niekedy museli až 1000000 vrcholov ... Skúste to nejako otestovať.

## 6.1.3 Vypisovanie hodnôt v BVS

Binárny vyhladávací strom má jednu veľmi dôležitú vlastnosť, keď navštívime vrcholy v poradí **inorder**, resp. vytvoríme pole s hodnotami v poradí **inorder**, dostaneme **usporiadanú** postupnosť všetkých hodnôt. Môžeme to otestovať:

```
strom = BinarnyVyhladavaciStrom()
for i in range(20):
    strom.pridaj(rr(100))
print('inorder = ', strom.inorder())
```

napr.

```
inorder =  4  8 11 21 25 31 43 51 55 56 57 61 71 72 77 81 87 96 98
```

Toto je veľmi dôležitá vlastnosť **BVS** a treba si ju zapamätať.

Výsledný listing definície triedy BinarnyVyhladavaciStrom:

```
from random import randrange as rr
import tkinter

class BinarnyVyhladavaciStrom:
    canvas = None

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

        def __init__(self):
            self.root = None

        def pridaj(self, hodnota):          # rekurzivna verzia

            def pridaj_vrchol(vrch):
                if vrch.data == hodnota:
                    return
                if vrch.data > hodnota:
                    if vrch.left is not None:
                        pridaj_vrchol(vrch.left)
                    else:
                        vrch.left = self.Vrchol(hodnota)
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                    else:
                        pridaj_vrchol(vrch.right)

                if self.root is None:
                    self.root = self.Vrchol(hodnota)
                else:
                    pridaj_vrchol(self.root)

            def pridaj(self, hodnota):
                if self.root is None:
                    self.root = self.Vrchol(hodnota)
                else:
                    vrch = self.root
                    while vrch.data != hodnota:
                        if vrch.data > hodnota:
                            if vrch.left is not None:
                                vrch = vrch.left
                            else:
                                vrch.left = self.Vrchol(hodnota)
                        else:
                            if vrch.right is None:
                                vrch.right = self.Vrchol(hodnota)
                            else:
                                vrch = vrch.right

            def kresli(self):
```

```

def kresli_rek(vrch, sir, x, y):
    if vrch.left is not None:
        self.canvas.create_line(x, y, x-sir//2, y+40)
        kresli_rek(vrch.left, sir//2, x-sir//2, y+40)
    if vrch.right is not None:
        self.canvas.create_line(x, y, x+sir//2, y+40)
        kresli_rek(vrch.right, sir//2, x+sir//2, y+40)
    #self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray')
    #self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
    #self.canvas.create_text(x, y, text=vrch.data)

    if self.canvas is None:
        BinarnyVyhladavaciStrom.canvas = tkinter.Canvas(bg='white', width=800, height=400)
        self.canvas.pack()
        self.canvas.delete('all')
        kresli_rek(self.root, 380, 400, 40)

def __len__(self):

    def pocet_rek(vrch):
        if vrch is None:
            return 0
        return 1 + pocet_rek(vrch.left) + pocet_rek(vrch.right)

    return pocet_rek(self.root)

def preorder(self):

    def preorder_rek(vrch):
        if vrch is None:
            return ''
        return repr(vrch.data) + ' ' + preorder_rek(vrch.left) +_
preorder_rek(vrch.right)

    return preorder_rek(self.root)

def postorder(self):

    def postorder_rek(vrch):
        if vrch is None:
            return ''
        return postorder_rek(vrch.left) + postorder_rek(vrch.right) +_
repr(vrch.data) + ' '

    return postorder_rek(self.root)

def inorder(self):

    def inorder_rek(vrch):
        if vrch is None:
            return ''
        return inorder_rek(vrch.left) + repr(vrch.data) + ' ' + inorder_rek(vrch.right)

    return inorder_rek(self.root)

```

```

def je_prvkom(self, hodnota):          # rekurzivna verzia

    def je_prvkom_rek(vrch):
        if vrch is None:
            return False
        if hodnota == vrch.data:
            return True
        if hodnota < vrch.data:
            return je_prvkom_rek(vrch.left)
        else:
            return je_prvkom_rek(vrch.right)

    return je_prvkom_rek(self.root)

def je_prvkom(self, hodnota):
    vrch = self.root
    while vrch is not None:
        if hodnota == vrch.data:
            return True
        if hodnota < vrch.data:
            vrch = vrch.left
        else:
            vrch = vrch.right
    return False

__contains__ = je_prvkom

```

Samozrejme, že rekurzívne verzie metód `pridaj()` a `je_prvkom()` môžeme odtiaľto vyhodiť. Všimnite si posledný riadok, v ktorom definujeme triedny atribút `__contains__` ako metódu rovnakú ako `je_prvkom()`. Vďaka tomuto môžeme písat' 43 in strom namiesto `strom.je_prvkom(43)`.

#### 6.1.4 Vyhodenie z BVS

Pri uvažovaní o vlastnostiach a algoritmoch binárnych vyhľadávacích stromov sa často spomína aj vyhodenie niekto-  
rého vrcholu tak, aby strom ostal **BVS** a pritom sme ho nemuseli celý prerábať'.

Základnou ideou je zjednodušene toto:

- nájdeme vyhadzovaný vrchol a ak je to list, jednoducho ho vyhodíme (nahradíme `None`)
- ak je to vnútorný vrchol s jediným synom, tak tento vrchol môžeme vyhodiť tiež a namiesto neho otcovi nasta-  
víme podstrom s týmto jedným existujúcim synom
- ak je to vrchol, ktorý má oboch synov, tak v podstrome ľavého syna (tam sú všetci menší, ako vyhadzovaný  
vrchol) nájdeme vrchol s najväčšou hodnotou (stačí íst' v podstrome stále vpravo, kým sa dá): tento vrchol už  
určite nemá pravého syna (inak by neboli najväčší), tak ho vyhodíme a jeho hodnotu dáme namiesto pôvodne  
vyhadzovaného vrcholu

Tento postup budeme programovať' na cvičeniacich.

## 6.2 Aritmetické stromy

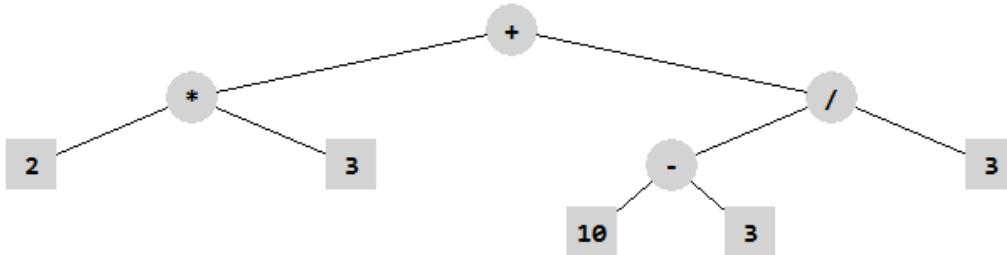
Ďalším pekným príkladom využitia binárnych stromov, sú tzv. **aritmetické stromy**. Môžete ich nájsť aj pod názvom `binary expression tree` ([http://en.wikipedia.org/wiki/Binary\\_expression\\_tree](http://en.wikipedia.org/wiki/Binary_expression_tree)) (aj

inde (<http://penguin.ewu.edu/cscd300/Topic/ExpressionTree/index.html>), ale častočne aj ako parse tree ([http://en.wikipedia.org/wiki/Parse\\_tree](http://en.wikipedia.org/wiki/Parse_tree)).

O čo tu ide:

- budú nás zaujímať také binárne stromy, ktoré majú vo vnútorných vrcholoch znamienka operácií (napr. '+', '-', '\*', '/') a v listoch sú operandy, napr. celé čísla alebo mená premenných
- takýto strom jednoznačne popisuje štandardný aritmetický výraz
- okrem toho, vyhodnocovanie takéhoto aritmetického výrazu (teda binárneho stromu) je veľmi jednoduché - uvidíme rekurzívnu vyhodnocovaciu funkciu

Pozrite napr. takýto aritmetický strom:



Zrejme zodpovedá aritmetickému výrazu  $2 * 3 + (10 - 3) / 3$ .

Pri definovaní novej triedy budeme vychádzat z jemne upravenej verzie BinarnyStrom:

```

import tkinter

class AritmetickyStrom:
    canvas = None

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

        def __init__(self, root=None):
            self.root = root

        def kresli(self):

            def kresli_rek(vrch, sir, x, y):
                if vrch.left is not None:
                    self.canvas.create_line(x, y, x-sir//2, y+40)
                    kresli_rek(vrch.left, sir//2, x-sir//2, y+40)
                if vrch.right is not None:
                    self.canvas.create_line(x, y, x+sir//2, y+40)
                    kresli_rek(vrch.right, sir//2, x+sir//2, y+40)
                if vrch.left is None and vrch.right is None:
                    self.canvas.create_rectangle(x-15, y-15, x+15, y+15, fill='lightgray', outline='')
                else:
                    self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray', outline='')
                self.canvas.create_text(x, y, text=vrch.data, font='consolas 12 bold')

            kresli_rek(self.root, 30, 150, 150)
  
```

```

if self.canvas is None:
    AritmetickyStrom.canvas = tkinter.Canvas(bg='white', width=800,_
height=400)
    self.canvas.pack()
    self.canvas.delete('all')
    kresli_rek(self.root, 380, 400, 40)

def preorder(self):

    def preorder_rek(vrch):
        if vrch is None:
            return ''
        return str(vrch.data) + ' ' + preorder_rek(vrch.left) + preorder_
rek(vrch.right)

    return preorder_rek(self.root)

def postorder(self):

    def postorder_rek(vrch):
        if vrch is None:
            return ''
        return postorder_rek(vrch.left) + postorder_rek(vrch.right) +_
str(vrch.data) + ' '

    return postorder_rek(self.root)

def inorder(self):

    def inorder_rek(vrch):
        if vrch is None:
            return ''
        return inorder_rek(vrch.left) + str(vrch.data) + ' ' + inorder_
rek(vrch.right)

    return inorder_rek(self.root)

```

Teraz sa nebudeme zaoberať algoritmami na vytváranie takýchto stromov (sú podobné algoritmom na prevody infixu a postfixu). Do inicializácie `__init__()` sme pridali parameter `root`, vďaka čomu môžeme jednoducho vytvoriť celý strom "ručne". Pomôžeme si funkciou `v()` na definovanie jedného vrcholu stromu:

```

v = AritmetickyStrom.Vrchol
strom = AritmetickyStrom(v('+', v('*', v(2), v(3)), v('/'), v('-', v(10),_
v(3)), v(3))))
strom.kresli()

```

Metóda `hodnota()`, ktorá vypočíta hodnotu aritmetického výrazu, najprv zistí, či je vrchol list (tedy je to operand, teda číslo), alebo vnútorný vrchol (je to operácia). Pre operáciu rekurzívne spustí vyhodnocovanie pre ľavý aj pravý podstrom a oba tieto medzivýsledky spojí príslušnou operáciou:

```

class AritmetickyStrom:
    ...
    def hodnota(self):

        def hodnota_rek(vrch):
            if vrch is None:
                return 0

```

```
        if vrch.left is None and vrch.right is None:
            return vrch.data
        vlavo = hodnota_rek(vrch.left)
        vpravo = hodnota_rek(vrch.right)
        if vrch.data == '+':
            return vlavo + vpravo
        if vrch.data == '*':
            return vlavo * vpravo
        if vrch.data == '-':
            return vlavo - vpravo
        if vrch.data == '/':
            return vlavo / vpravo

    return hodnota_rek(self.root)
```

otestujeme:

```
>>> strom = AritmetickyStrom(v('+', v('*',
    v(2), v(3)), v('/',
    v(10), v(3))), v(3)))
>>> strom.hodnota()
8.33333333333334
```

Ak by sme tento strom vypísali v rôznych poradiach, dostali by sme známe zápisy:

- preorder() vytvorí **prefixový** zápis
- inorder() vytvorí **infixový** zápis, ale bez zátvoriek
- postorder() vytvorí **postfixový** zápis

otestujme:

```
print('preorder = ', strom.preorder())
print('inorder = ', strom.inorder())
print('postorder = ', strom.postorder())
```

```
preorder = + * 2 3 / - 10 3 3
inorder = 2 * 3 + 10 - 3 / 3
postorder = 2 3 * 10 3 - 3 / +
```

Naozaj vidíme zhodu so zápismi **prefix**, **infix** a **postfix**. Metódu inorder() môžeme upraviť tak, aby sme dostali uzávorkovaný infixový výraz:

```
class AritmetickyStrom:
    ...
    def inorder(self):

        def inorder_rek(vrch):
            if vrch is None:
                return ''
            if vrch.left is None and vrch.right is None:
                return str(vrch.data)
            return '+inorder_rek(vrch.left) + vrch.data + inorder_rek(vrch.
            right) + ' '

        return inorder_rek(self.root)
```

Otestujeme na predchádzajúcom aritmetickom strome:

```
>>> strom.inorder()
'((2*3)+((10-3)/3))'
>>> ((2*3)+((10-3)/3))
8.33333333333334
```

## 6.3 Všeobecné stromy

Ked'že vo všeobecnom strome môže mať každý vrchol ľubovoľný počet potomkov, najlepšie sa na ich uchovanie využije pole referencií (typ list):

```
class VseobecnyStrom:
    class Vrchol:
        def __init__(self, data):
            self.data = data
            self.child = [] # pole synov

        def __init__(self):
            self.root = None
```

Tejto reprezentáciu treba prispôsobiť aj všetky metódy, ktoré pracujú so všeobecným stromom, napr.

```
class VseobecnyStrom:
    class Vrchol:
        def __init__(self, data):
            self.data = data
            self.child = []

        def __init__(self):
            self.root = None

        def __len__(self):

            def pocet_rek(vrch):
                if vrch is None:
                    return 0
                #return sum(map(pocet_rek, vrch.pole)) + 1

                vysl = 1
                for syn in vrch.child:
                    vysl += pocet_rek(syn)
                return vysl

            return pocet_rek(self.root)

        def __repr__(self):

            def repr1(vrch):
                if vrch is None:
                    return '()'
                vysl = [repr(vrch.data)]
                for syn in vrch.child:
                    vysl.append(repr1(syn))
                return '(' + ', '.join(vysl) + ')'

            return repr1(self.root)
```

```

def pridaj_nahodne(self, *pole):

    def pridaj_vrchol(vrch):
        n = len(vrch.child)
        i = rr(n+1)
        if i == n:
            vrch.child.append(self.Vrchol(hodnota))
        else:
            pridaj_vrchol(vrch.child[i])

    for hodnota in pole:
        if self.root is None:
            self.root = self.Vrchol(hodnota)
        else:
            pridaj_vrchol(self.root)

def kresli(self):

    def kresli_rek(vrch, sir, x, y):
        n = len(vrch.child)
        if n != 0:
            sir0 = 2 * sir // n
            for i in range(n):
                x1, y1 = x - sir + i*sir0 + sir0//2, y + 40
                self.canvas.create_line(x, y, x1, y1)
                kresli_rek(vrch.child[i], sir0//2, x1, y1)
            self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray'
→', outline='')
            self.canvas.create_text(x, y, text=vrch.data, font='consolas 14'
→bold')

        if self.canvas is None:
            VseobecnyStrom.canvas = tkinter.Canvas(bg='white', width=800,
→height=400)
            self.canvas.pack()
            self.canvas.delete('all')
            kresli_rek(self.root, 380, 400, 40)

```

Môžeme otestovať:

```

strom = VseobecnyStrom()
strom.pridaj_nahodne(*rr(100) for i in range(20)))
strom.kresli()
print('počet vrcholov v strome =', len(strom))
print('strom =', strom)

```

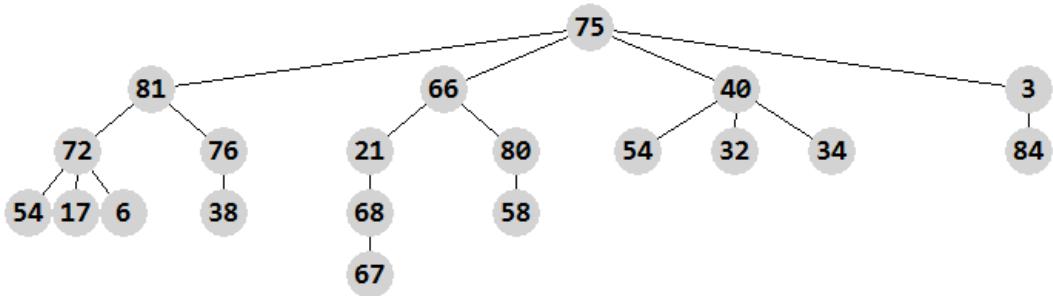
vypíše sa napr.

```

počet vrcholov v strome = 20
strom =
→(75, (81, (72, (54), (17), (6)), (76, (38))), (66, (21, (68, (67)))), (80, (58))), (40, (54)), (32, (34)), (3, (84))

```

A dostávame nejaký takýto obrázok:





---

## Triedenia

---

Štandardná funkcia `sorted()` triedi, napr.

- postupnosť čísel, reťazcov
- znakový reťazec
- postupnosť n-tíc, napr. postupnosť súradníc
- postupnosť dvojíc (ključ, hodnota), ktorá vznikne z asociatívneho poľa pomocou `pole.items()`

Výsledkom je vždy pole (typ `list`). Ak pri volaní uvedieme ďalší parameter `reverse=True`, pole sa utriedí zostupne od najväčších hodnôt po najmenšie.

Pozrime túto ukážku:

```
>>> import random
>>> pole = [random.randrange(10000) for i in range(15)]
>>> pole
[1372, 6518, 7580, 9941, 9518, 3211, 8428, 7624, 35, 9341, 572, 6218, 3819, ↵
 ↵8943, 8527]
>>> sorted(pole)
[35, 572, 1372, 3211, 3819, 6218, 6518, 7580, 7624, 8428, 8527, 8943, 9341, ↵
 ↵9518, 9941]
>>> sorted(pole, reverse=True)
[9941, 9518, 9341, 8943, 8527, 8428, 7624, 7580, 6518, 6218, 3819, 3211, ↵
 ↵1372, 572, 35]
```

Vidíme vzostupne aj zostupne utriedené náhodne vygenerované pole. Ak by sme dostali úlohu toto pole utriediť len podľa posledných dvoch cifier každého čísla, bolo by to ľahšie. Vyrobme si pomocnú funkciu a nové pole, ktoré obsahuje len tieto posledné cifry:

```
>>> def posledne2(x):
    return x % 100

>>> pole1 = [posledne2(p) for p in pole]
>>> pole1
[72, 18, 80, 41, 18, 11, 28, 24, 35, 41, 72, 18, 19, 43, 27]
>>> sorted(pole1)
[11, 18, 18, 18, 19, 24, 27, 28, 35, 41, 41, 43, 72, 72, 80]
```

Ak chceme teraz vrátiť pôvodné čísla k týmto dvom posledným cifrám, musíme si ich pamätať, najlepšie vo dvojici spolu s poslednými 2 ciframi:

```
>>> pole1 = [(posledne2(p),p) for p in pole]
>>> pole1
[(72, 1372), (18, 6518), (80, 7580), (41, 9941), (18, 9518), (11, 3211), (28, 8428), (24, 7624), (35, 35), (41, 9341), (72, 572), (18, 6218), (19, 3819), (43, 8943), (27, 8527)]
>>> pole2 = sorted(pole1)
>>> pole2
[(11, 3211), (18, 6218), (18, 6518), (18, 9518), (19, 3819), (24, 7624), (27, 8527), (28, 8428), (35, 35), (41, 9341), (41, 9941), (43, 8943), (72, 572), (72, 1372), (80, 7580)]
```

Už je to skoro hotové: z týchto utriedených dvojíc zoberieme len pôvodné čísla, t.j. zoberieme druhé prvky dvojíc:

```
>>> [p[1] for p in pole2]
[3211, 6218, 6518, 9518, 3819, 7624, 8527, 8428, 35, 9341, 9941, 8943, 572, 1372, 7580]
```

A máme naozaj to, čo sme na začiatku očakávali: utriedené pôvodné pole, ale triedime len podľa posledných dvoch cifier čísel v poli.

Štandardná funkcia `sorted()` má okrem parametra `reverse` ešte jeden, ktorý slúži práve na toto: zadáme funkciu, ktorá určí, ako sa bude triedenie pri porovnávaní pozerať na každý prvok poľa - podľa tohto sa pôvodné pole bude triediť. Takže, komplikovaný postup s vytváraním dvojíc môžeme zjednodušiť použitím parametra `key`:

```
>>> sorted(pole, key=posledne2)
[3211, 6518, 9518, 6218, 3819, 7624, 8527, 8428, 35, 9941, 9341, 8943, 1372, 572, 7580]
>>> sorted(pole, key=lambda x: x%100)
[3211, 6518, 9518, 6218, 3819, 7624, 8527, 8428, 35, 9941, 9341, 8943, 1372, 572, 7580]
```

Vidíme, že tu môžeme použiť aj konštrukciu `lambda`. Takže, keď zadáme parameter `key`, tak triedenie nebude porovnávať prvky poľa, ale prerobené prvky poľa zadanou funkciou, napr. `posledne2`.

Zamyslite sa, v akom poradí sa teraz utriedi toto isté pole:

```
>>> sorted(pole, key=str)
[1372, 3211, 35, 3819, 572, 6218, 6518, 7580, 7624, 8428, 8527, 8943, 9341, 9518, 9941]
```

Zhrňme použitie štandardnej funkcie `sorted()`:

### štandardná funkcia `sorted()`

Funkcia `sorted()` z prvkov ľubovoľnej postupnosti (napr. pole, n-tica, reťazec, súbor, ...) vytvorí pole týchto hodnôt vo vzostupne usporiadanom poradí. Funkcia môže mať ešte tieto dva nepovinné parametre:

- `reverse=True` spôsobí usporiadanie prvkov vo vzostupnosm poradí (od najväčšieho po najmenšie)
- `key=funkcia`, kde `funkcia` je bud' meno existujúcej funkcie alebo `lambda` konštrukcia - táto funkcia musí byť definovaná pre jednen parameter - potom vzájomné porovnávanie dvoch prvkov pri usporiadavaní bude používať práve túto funkciu

Je dôležité, aby sa všetky triedené prvky dali navzájom porovnávať, napr. nemôžeme usporiadátať pole, ktoré obsahuje čísla aj reťazce. Môžeme ich pomocou parametra `key` previesť na také hodnoty, ktoré porovnávať vieme.

Nasledujúce ukážky ilustrujú najmä použitie parametra key.

- usporiadanie množiny mien (čo sú reťazce v tvare meno priezvisko) najprv podľa priezvisk, a ak sú dve priezviská rovnaké, tak podľa mien:

```
>>> m = {'Janko Hrasko', 'Eugen Suchon', 'Ludovit Stur', 'Andrej Sladkovic',
        ↪'Janko Stur'}
>>> sorted(m)
['Andrej Sladkovic', 'Eugen Suchon', 'Janko Hrasko', 'Janko Stur', 'Ludovit
 ↪Stur']
>>> sorted(m, key=lambda x: x.split()[:-1])
['Janko Hrasko', 'Andrej Sladkovic', 'Janko Stur', 'Ludovit Stur', 'Eugen
 ↪Suchon']
```

- usporiadanie telefónneho zoznamu podľa telefónnych čísel, keďže zoznam je tu definovaný ako asociatívne pole, vytvoríme z neho najprv pole dvojíc (klúč, hodnota):

```
>>> tel = {'Betka':737373, 'Dusan':555444, 'Anka': 363636, 'Egon':210210,
           'Cyril': 911111, 'Gaba':123456, 'Fero':288288}
>>> sorted(tel.items())
[('Anka', 363636), ('Betka', 737373), ('Cyril', 911111), ('Dusan', 555444),
 ('Egon', 210210), ('Fero', 288288), ('Gaba', 123456)]
>>> sorted(tel.items(), key=lambda x: x[1])
[('Gaba', 123456), ('Egon', 210210), ('Fero', 288288), ('Anka', 363636),
 ('Dusan', 555444), ('Betka', 737373), ('Cyril', 911111)]
```

- usporiadanie pola reťazcov bez ohľadu na malé a veľké písmená:

```
>>> pole = ('Prvy', 'DRUHY', 'druha', 'pRva', 'PRVE', 'druhI')
>>> sorted(pole)
['DRUHY', 'PRVE', 'Prvy', 'druhI', 'druha', 'pRva']
>>> sorted(pole, key=str.lower)
['druha', 'druhI', 'DRUHY', 'pRva', 'PRVE', 'Prvy']
>>> sorted(pole, key=lambda s: s.lower())
['druha', 'druhI', 'DRUHY', 'pRva', 'PRVE', 'Prvy']
```

všimnite si dva spôsoby použitia metódy lower()

- usporiadanie pola bodov v rovine (pole dvojíc (x, y)) podľa toho, ako sú vzdialené od bodu (0,0); predpokladáme, že máme danú funkciu vzd(x, y), ktorá vypočíta vzdialosť bodu od počiatku:

```
>>> def vzd(x, y):
    return (x**2 + y**2)**.5
>>> from random import randint as ri
>>> body = [(ri(-5,5),ri(-5,5)) for i in range(10)]
>>> body
[(-1, 5), (2, 3), (-1, 0), (-1, -2), (0, 1), (-4, 4), (-4, 4), (5, -1), (5, -2),
 ↪(-5, 0)]
>>> sorted(body, key=lambda b: vzd(*b))
[(-1, 0), (0, 1), (-1, -2), (2, 3), (-5, 0), (-1, 5), (5, -1), (5, -2), (-4, 4),
 ↪(-4, 4)]
```

všimnite si, že tu sme nemohli priamo zapísat key=vzd, lebo táto funkcia očakáva 2 parametre a my máme triediť dvojice

- v danom slove usporiadaj znaky podľa abecedy:

```
>>> slovo = 'krasokorculovanie'
>>> ''.join(sorted(slovo))
```

```
'aaceikklnooorrsuv'
```

- v poli sa vyskytujú reťazce aj čísla, treba to najeko usporiadat':

```
>>> pole = ['abc', 3.14, 'def', 2.71, 'ghi', 333, 'jkl', 22]
>>> sorted(pole)

TypeError: unorderable types: float() < str()
>>> sorted(pole, key=str)
[2.71, 22, 3.14, 333, 'abc', 'def', 'ghi', 'jkl']
```

V ďalšej časti prednášky uvedieme niekoľko najznámejších triedení:

- tieto funkcie utriedia priamo prvky poľa pričom nebudú používať žiadne ďalšie pomocné pole, preto im hovoríme **in-place**
- uvádzané funkcie nevracajú žiadnu hodnotu (fungujú teda podobne ako metóda `sort` pre typ `list`)

### 7.1 Min sort

Môžeme sa stretnúť aj s názvom `selection sort` ([http://en.wikipedia.org/wiki/Selection\\_sort](http://en.wikipedia.org/wiki/Selection_sort)).

Pracuje na tomto princípe:

- nájde minimálny prvok a zaraď ho na začiatok poľa
- opäť nájde ďalší minimálny prvok (prvý pritom z hľadania vynechá) a zaraď ho ako druhý prvok (za minimálny) - máme dva minimálne prvky na svojom mieste
- opäť nájde ďalší minimálny prvok (prvé dva pritom vynechá) a zaraď ho ako tretí prvok (za oba minimálne) - máme tri minimálne prvky na svojom mieste
- ... takto sa to opakuje, kým sa nespracujú všetky prvky poľa

V tomto algoritme zaraďujeme nájdený minimálny niekde na začiatok. Aby sme pole zbytočne nerozsúvali, urobíme len obyčajnú výmenu dvoch prvkov poľa: vymeníme nájdené minimum s prvkom na pozícii, kam chceme minimum zaraďiť. Uvedieme dve rôzne verzie tohto algoritmu, pričom prvá z nich minimum hľadá tak, že do i-teho prvku (sem má prísť minimum) postupne priraduje všetky, ktoré sú od i-teho menšie:

```
def min_sort1(pole):
    for i in range(len(pole)-1):
        for j in range(i+1, len(pole)):
            if pole[i] > pole[j]:
                pole[i], pole[j] = pole[j], pole[i]
```

Druhá verzia najprv nájde, kde v poli sa nachídza minimum (jeho index) a až potom jednou výmenou vymení i=ty prvak s minimálnym.

```
def min_sort2(pole):
    for i in range(len(pole)-1):
        najmensi = i
        for j in range(i+1, len(pole)):
            if pole[najmensi] > pole[j]:
                najmensi = j
        pole[i], pole[najmensi] = pole[najmensi], pole[i]
```

Ak by sme chceli nejako vidieť, ako tento algoritmus funguje, mohli by sme si na nejaké mesto vložiť kontrolnú tlač, napr.

```

def min_sort2(pole):
    print(*pole)
    for i in range(len(pole)-1):
        najmensi = i
        for j in range(i+1, len(pole)):
            if pole[najmensi] > pole[j]:
                najmensi = j
        pole[i], pole[najmensi] = pole[najmensi], pole[i]
    print(*pole)

import random
p = [random.randrange(10, 100) for i in range(10)]
min_sort2(p)

```

A mohli by sme dostať nejaký takýto výpis:

```

25 35 46 63 84 12 28 80 93 79
12 35 46 63 84 25 28 80 93 79
12 25 46 63 84 35 28 80 93 79
12 25 28 63 84 35 46 80 93 79
12 25 28 35 84 63 46 80 93 79
12 25 28 35 46 63 84 80 93 79
12 25 28 35 46 63 84 80 93 79
12 25 28 35 46 63 79 80 93 84
12 25 28 35 46 63 79 80 93 84
12 25 28 35 46 63 79 80 84 93

```

Každý ďalší riadok ukazuje obsah poľa po jednom prechode vonkajšieho cyklu, t.j. po zaradení i-teho najmenšieho na správne miesto: v prvom riadku je ešte netriedené pole, v druhom je jeho obsah po zaradení prvého minima na svoje miesto (číslo 12), v ďalšom riadku je zaradené už aj druhé číslo 25, atď. V poslednom riadku výpisu je utriedené celé pole.

## 7.2 Bubble sort

Bublinkové triedenie ([http://en.wikipedia.org/wiki/Bubble\\_sort](http://en.wikipedia.org/wiki/Bubble_sort)) pracuje na princípe postupného porovnávania všetkých dvojíc susediacich prvkov: do prvku poľa s menším indexom ide menší z nich. Po jednom prechode celým poľom sa maximálny prvok určite dostane na koniec poľa. Ked' budeme tento postup opakovať toľkokrát, ako je počet prvkov poľa, na záver bude celé pole utriedené.

Všimnite si, že sa tento algoritmus trochu podobá na `min_sort1()`:

```

def bubble_sort(pole):
    for i in range(len(pole)):
        for j in range(len(pole)-1):
            if pole[j] > pole[j+1]:
                pole[j], pole[j+1] = pole[j+1], pole[j]

```

Ked' to otestujeme podobne ako `min_sort`:

```

def bubble_sort(pole):
    print(*pole)
    for i in range(len(pole)):
        for j in range(len(pole)-1):
            if pole[j] > pole[j+1]:
                pole[j], pole[j+1] = pole[j+1], pole[j]

```

```
print(*pole)

import random
p = [random.randrange(10, 100) for i in range(10)]
bubble_sort(p)
```

Vo výpise môžeme vidieť:

```
35 44 40 79 17 87 84 82 67 50
35 40 44 17 79 84 82 67 50 87
35 40 17 44 79 82 67 50 84 87
35 17 40 44 79 67 50 82 84 87
17 35 40 44 67 50 79 82 84 87
17 35 40 44 50 67 79 82 84 87
17 35 40 44 50 67 79 82 84 87
17 35 40 44 50 67 79 82 84 87
17 35 40 44 50 67 79 82 84 87
17 35 40 44 50 67 79 82 84 87
17 35 40 44 50 67 79 82 84 87
17 35 40 44 50 67 79 82 84 87
```

Po každom prechode vonkajšieho cyklu sa ďalšie maximum dostáva na koniec: na konci sa hromadia maximálne hodnoty. Okrem toho menšie prvky sa pomaly presúvajú smerom k začiatku.

### 7.3 Vizualizovanie sortov

Rôznych algoritmov triedenia je dosť veľa a z takýchto výpisov obsahu poľa sa zriedkavo dá zistíť, ako to naozaj funguje. Často sa používajú iné zobrazovania momentálneho obsahu poľa, napr. graficky, kde rôzne veľké hodnoty sú zobrazené rôznou dĺžkou úsečiek.

Aby sme čo najšikovnejšie vytvorili vizualizáciu priebehu algoritmu triedenia, zadefinujeme si vlastnú triedu `Pole`. Ked' teraz spustíme už vytvorenú funkciu triedenia (napr. `bubble_sort()`), namiesto práce s prvkami poľa sa budú volať naše metódy, ktoré budú pracovať s naozajstným poľom, ale mohli by robiť aj nejaké ďalšie veci, napr. kresliť rôzne dlhé úsečky:

- vybrať hodnotu prvku `pole[index]` - budeme definovať metódu `__getitem__()`
- priradiť novú hodnotu prvku `pole[index]` - budeme definovať metódu `__setitem__()`
- zistiť veľkosť poľa `len(pole)` - budeme definovať metódu `__len__()`

Zapíšeme prvú verziu tejto triedy zatiaľ ešte bez vizualizácie:

```
class Pole:
    def __init__(self, pole):
        self.pole = pole

    def __getitem__(self, index):
        return self.pole[index]

    def __len__(self):
        return len(self.pole)

    def __setitem__(self, index, hodnota):
        self.pole[index] = hodnota
```

Zavoláme s touto triedou `bubble_sort()`:

```

import random
pole = [random.randrange(200) for i in range(15)]
print('pred =', *pole)
a = Pole(pole)
bubble_sort(a)
print('po   =', *pole)

```

Takto sa utriedi pôvodné pole, ktoré sa stalo atribútom inštancie a typu `Pole`. Tento test vypíše:

```

pred = 191 139 183 169 3 183 36 124 113 159 196 14 71 168 37
po   = 3 14 36 37 71 113 124 139 159 168 169 183 183 191 196

```

Do triedy tejto `Pole` teraz pridáme kreslenie čiar:

- do inicializácie `__init__()` pridáme vytvorenia grafickej plochy (canvas) a vykreslenie všetkých úsečiek podľa čísel v poli - zároveň sa identifikačné čísla týchto čiar uložia do poľa `self.id`
- metóda `__setitem__(index, hodnota)`, pomocou ktorej sa mení obsah prvku poľa, prekreslí príslušnú čiaru podľa tejto novej hodnoty

```

import tkinter

class Pole:
    def __init__(self, pole):
        self.pole = pole
        self.canvas = tkinter.Canvas(width=800, height=600, bg='white')
        self.canvas.pack()
        self.dx = 780/len(pole)
        self.id = []
        for i in range(len(pole)):
            ii = self.canvas.create_line(i*self.dx, 600, i*self.dx, 600-pole[i], width=3, fill='blue')
            self.id.append(ii)

    def __getitem__(self, index):
        return self.pole[index]

    def __len__(self):
        return len(self.pole)

    def __setitem__(self, index, hodnota):
        self.pole[index] = hodnota
        self.canvas.coords(self.id[index], index*self.dx, 600, index*self.dx, 600-hodnota)
        self.canvas.update()
        # self.canvas.after(100)

```

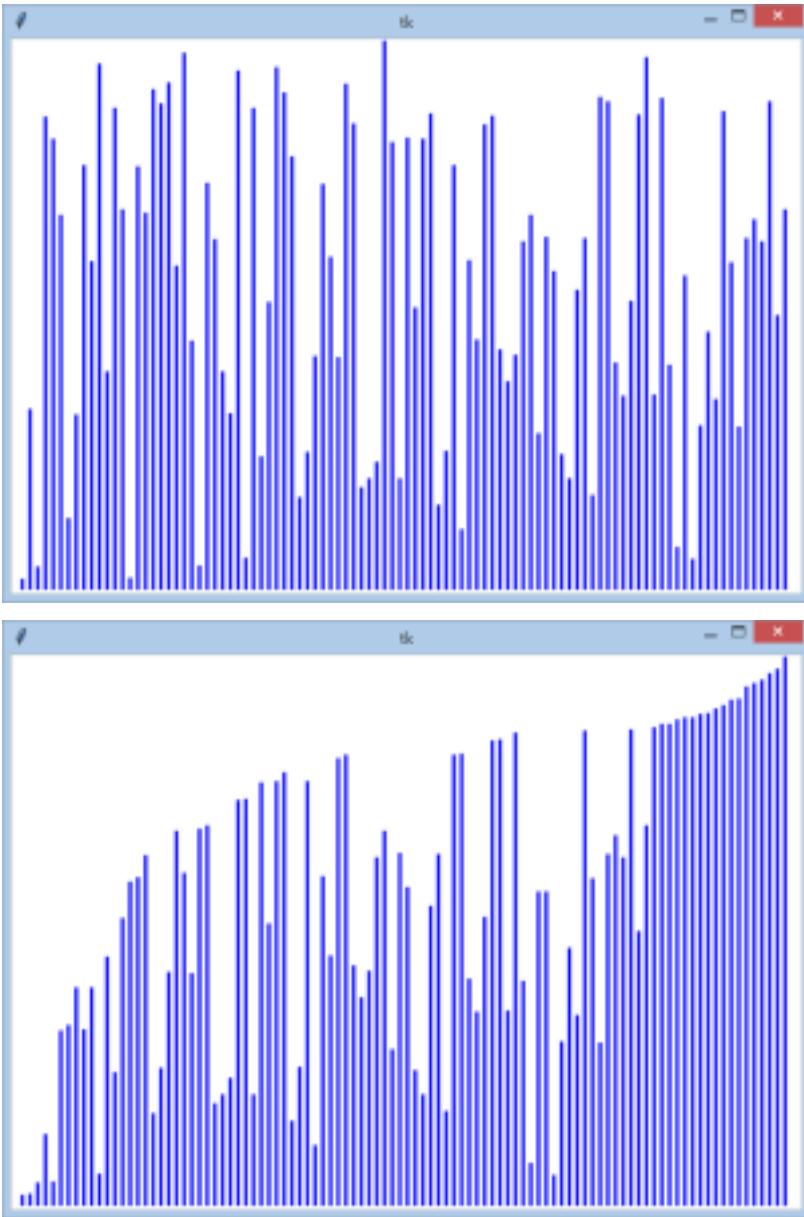
Otestujeme `bubble_sort()` ale so 100-prvkovým poľom, v ktorom budú náhodné hodnoty až do 500:

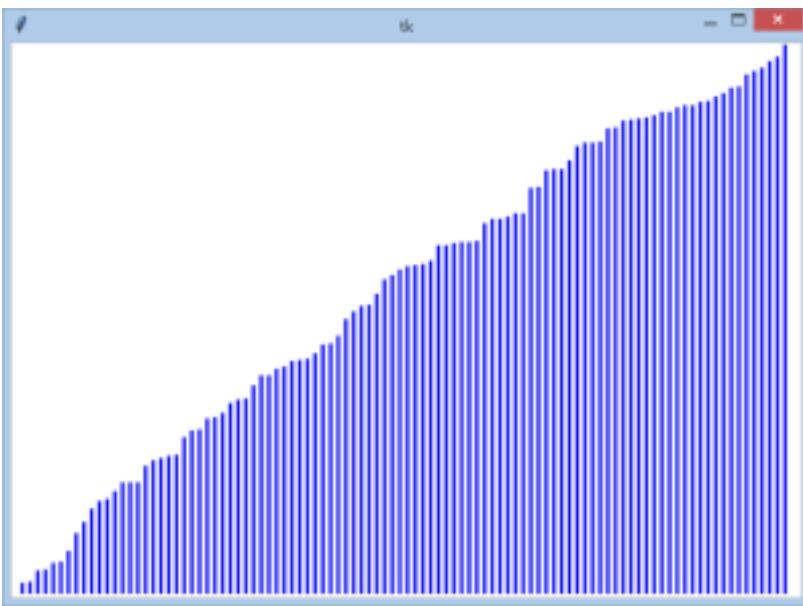
```

import random
pole = [random.randrange(500) for i in range(100)]
print('pred =', *pole)
a = Pole(pole)
bubble_sort(a)
print('po   =', *pole)

```

Môžeme postupne vidieť najprv náhodne vygenerované pole, potom priebežný stav počas triedenia (na konci poľa sa ukladajú maximálne prvky) a na záver utriedené pole:





Otestujte takto aj `min_sort()`.

## 7.4 Insert sort

Triedenie vkladaním ([http://en.wikipedia.org/wiki/insertion\\_sort](http://en.wikipedia.org/wiki/insertion_sort)) pracuje na takomto princípe:

- predpokladajme, že istá časť poľa na začiatku je už utriedená
- počas triedenia sa najbližší ešte neutriedený prvok zaradí do utriedenej časti na svoje miesto a tým sa tento neutriedený úsek predlží o 1

Zapíšme algoritmus:

```
def insert_sort(pole):
    # print(*pole)
    for i in range(1, len(pole)):
        prvok = pole[i]
        j = i-1
        while j >= 0 and pole[j] > prvok:
            pole[j+1] = pole[j]
            j -= 1
        pole[j+1] = prvok
    # print(*pole[:i+1], '|', *pole[i+1:])
```

Ako to funguje:

- v premennej `i` je hranica medzi utriedenými a neutriedenými časťami poľa
  - pri štarte predpokladáme, že prvý prvok je už jednoprvková utriedená časť a preto začíname od indexu 1 (teda druhým prvkom poľa)
- v premennej `prvok` je hodnota nasledovného prvku poľa, ktorú budeme zaradovať do utriedenej časti
- v premennej `j` je index posledného prvku v utriedenej časti - tento prvok budeme porovnávať s `prvok` a kým sú tieto hodnoty väčšie ako `prvok`, budeme utriedenú časť rozsúvať a `j` zmenšovať
- na záver sme v utriedenej časti dosiahli voľné miesto pre vkladaný prvok (v premennej `prvok`)

Výpisy, ktoré sú tu zakomentované, môžu pomôcť pochopíť, ako to funguje. Otestujeme (aj s výpismi):

```
import random
pole = [random.randrange(10, 100) for i in range(10)]
insert_sort(pole)
```

```
59 28 41 26 31 49 71 97 24 63
28 59 | 41 26 31 49 71 97 24 63
28 41 59 | 26 31 49 71 97 24 63
26 28 41 59 | 31 49 71 97 24 63
26 28 31 41 59 | 49 71 97 24 63
26 28 31 41 49 59 | 71 97 24 63
26 28 31 41 49 59 71 | 97 24 63
26 28 31 41 49 59 71 97 | 24 63
24 26 28 31 41 49 59 71 97 | 63
24 26 28 31 41 49 59 63 71 97 |
```

Znak ' | ' slúži na oddelenie utriedenej a ešte neutriedenej časti pol'a.

Vyskúšajte toto triedenie vizualizovať pomocou triedy Pole.

## 7.5 Quick sort

Je to najznámejší algoritmus triedenia (<http://en.wikipedia.org/wiki/Quicksort>), ktorý v roku 1960 vymyslel Tony Hoare ([http://en.wikipedia.org/wiki/Tony\\_Hoare](http://en.wikipedia.org/wiki/Tony_Hoare)).

Popíšeme zjednodušený princíp:

- najprv zvolíme ľubovoľný prvok pol'a ako tzv. **pivot** (my pre jednoduchosť zvolíme prvý prvok pol'a)
- d'alej všetky zvyšné prvky rozdelíme na tri kopy: na prvky, ktoré sú menšie ako **pivot**, na prvky, ktoré sa rovnajú **pivot** a na všetky zvyšné
- teraz rovnakým algoritmom (t.j. rekurzívne volanie) utriedime dve kopy: kopu menších a kopu väčších a takto utriedené časti (spolu s rovnými) nakoniec spojíme do výsledného utriedeného pol'a

Prvá verzia je rekurzívna funkcia, ktorá nemodifikuje svoj parameter, len vráti novo vytvorené pole, ktoré je už teraz utriedené:

```
def quick_sort1(pole):
    if len(pole) < 2:
        return pole
    pivot = pole[0]
    mensie = [prvok for prvok in pole if prvok < pivot]
    rovne = [prvok for prvok in pole if prvok == pivot]
    vacsie = [prvok for prvok in pole if prvok > pivot]
    return quick_sort1(mensie) + rovne + quick_sort1(vacsie)
```

Takéto riešenie nemôžeme vizualizovať v grafickej ploche našou triedou Pole, keďže výsledok sa postupne zlepuje z veľkého množstva malých kúskov pomocných polí (my vieme vizualizovať len zmeny v pôvodnom poli).

Ďalej sa zameriame na verziu, ktorá nebude používať pomocné polia, ale quick\_sort() prebehne v samotnom poli:

- na začiatku sa samotné pole (jeho časť od indexu z do indexu k) rozdelí na dve časti:
- zvolí sa pivot (prvý prvok pol'a) a zvyšné prvky postupne rozdelí do ľavej časti pol'a (pred pivota) a pravej za pivota podľa toho či sú menšie ako pivot alebo nie

- v premennej `index` je pozícia pivota v takto upravenom poli (ešte neutriedenom) - uvedomte si, že pivot je už na svojom mieste (podobne ako boli minimálne v `min_sorte` a maximálne prvky v `bubble_sorte`)
- d'alej nasleduje rekurzívne volanie na ľavú časť poľa pred pivotom a na pravú časť poľa za pivotom

Táto druhá verzia je **in-place**, t.j. nevytvára sa nové pole, ale triedi sa priamo to pole, ktoré je parametrom funkcie:

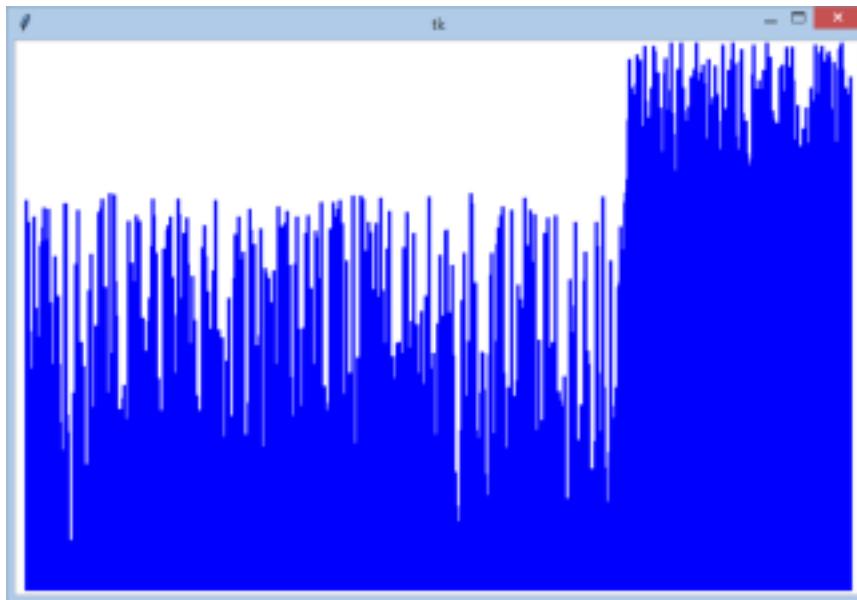
```
def quick_sort2(pole):
    def quick(z, k):
        if z < k:
            # rozdelenie na dve casti
            index = z
            pivot = pole[z]
            for i in range(z+1, k+1):
                if pole[i] < pivot:
                    index += 1
                    pole[index], pole[i] = pole[i], pole[index]
            pole[index], pole[z] = pole[z], pole[index]
            # v index je pozicia pivota
            quick(z, index-1)
            quick(index+1, k)

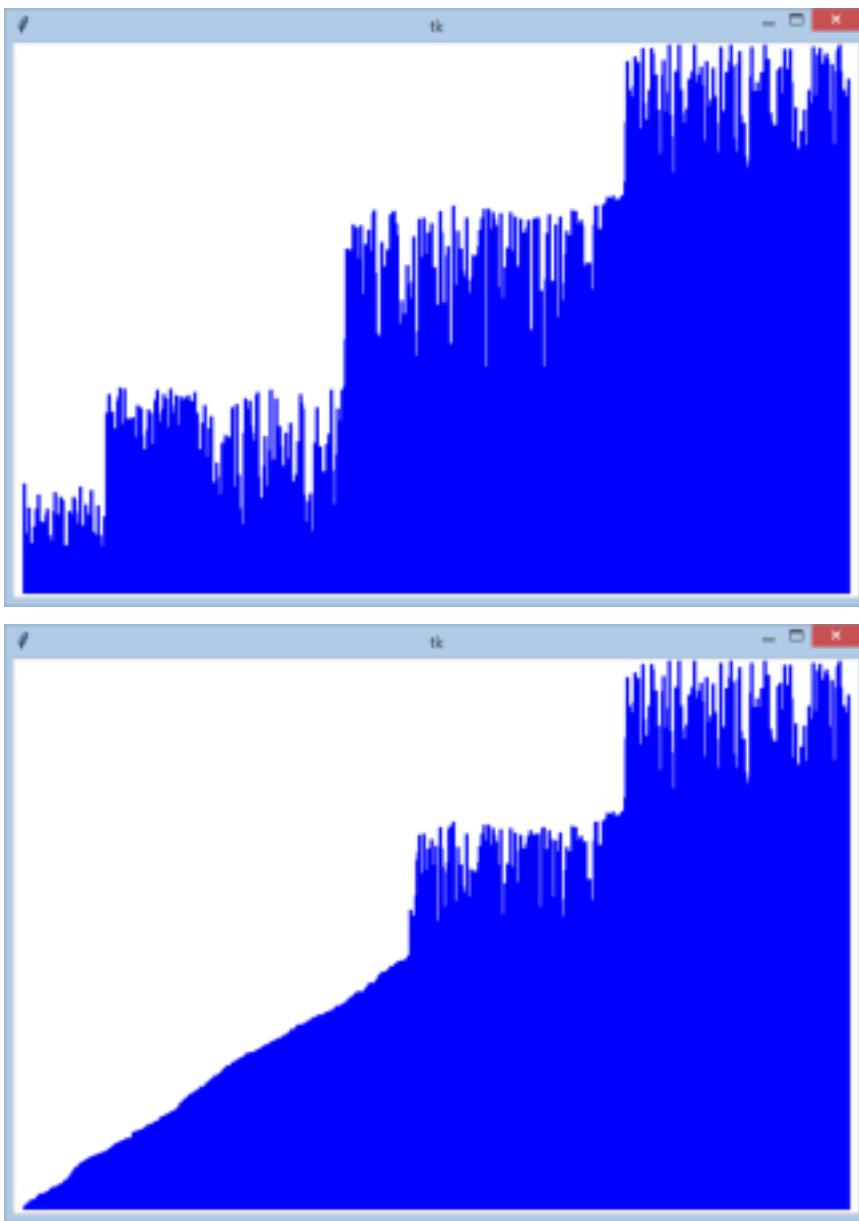
    quick(0, len(pole)-1)
```

Môžete to otestovať pomocou vizualizácie:

```
import random
pole = [random.randrange(500) for i in range(750)]
quick_sort2(Pole(pole))
```

Môžete postupne vidieť, ako sa najprv náhodne vygenerované pole rozdelilo podľa pivota na menšie a väčšie prvky, potom sa takto rekurzívne rozdelil aj prvý úsek a na poslednom zábere je už polovica poľa utriedená a triedi sa zvyšok poľa:





Niekedy sa triedenia zvyknú vizualizovať nie pomocou úsečiek ale len pomocou jedného koncového bodu úsečky - takto na začiatok vidíme náhodne rozhádzané bodky, ktoré sa postupne zoskupujú. Opravíme triedu `Pole`:

```
class Pole:
    def __init__(self, pole):
        self.pole = pole
        self.canvas = tkinter.Canvas(width=800, height=600, bg='white')
        self.canvas.pack()
        self.dx = 780/len(pole)
        self.id = []
        for i in range(len(pole)):
            ii = self.canvas.create_line(i*self.dx, 600-pole[i], i*self.dx, ↵
                                         601-pole[i], fill='blue')
            self.id.append(ii)

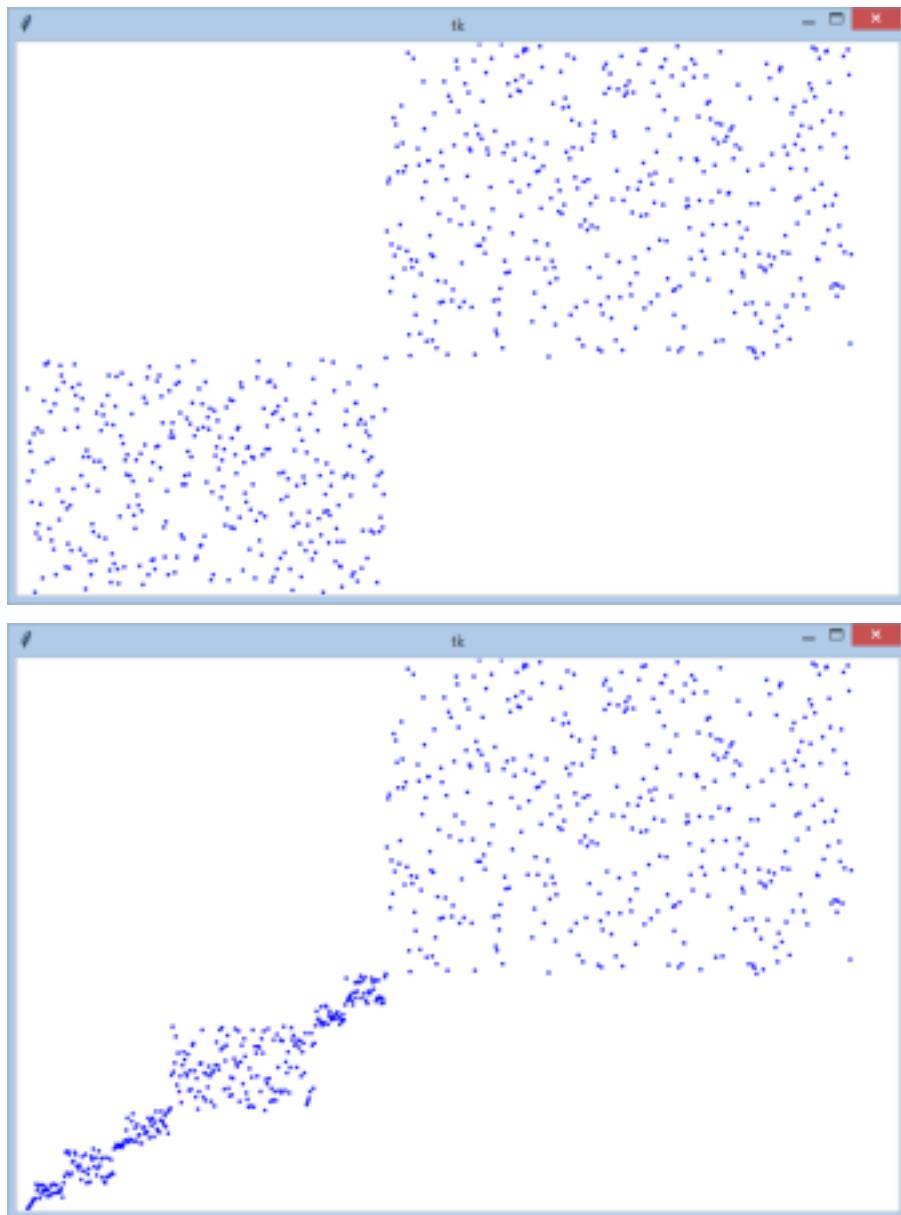
    def __getitem__(self, index):
```

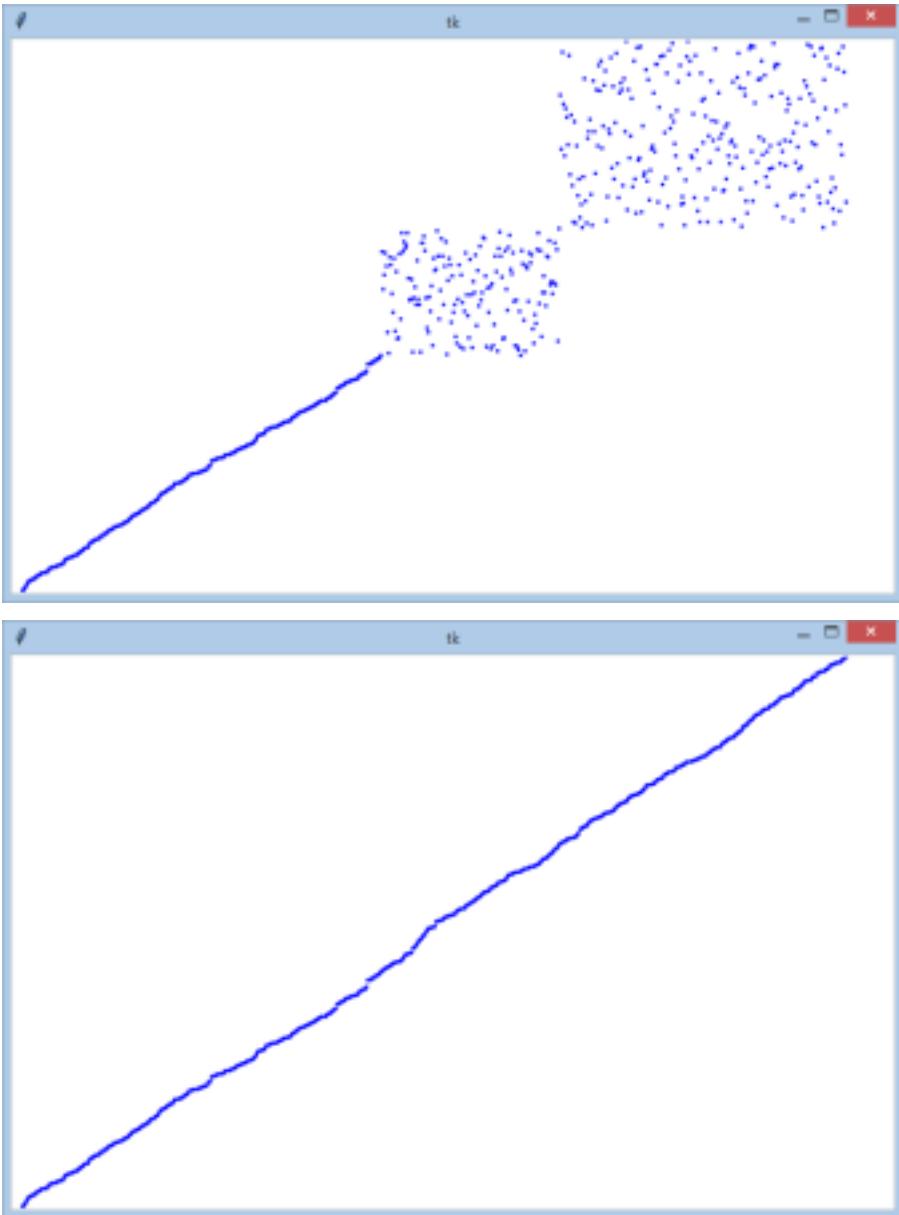
```
    return self.pole[index]

def __len__(self):
    return len(self.pole)

def __setitem__(self, index, hodnota):
    self.pole[index] = hodnota
    self.canvas.coords(self.id[index], index*self.dx, 600-hodnota,
    ↵index*self.dx, 601-hodnota)
    self.canvas.update()
```

Po spustení vizualizácie vidíme:





**Grafy**

## 8.1 Termmínológia

Graf je dátová štruktúra, ktorá sa skladá

- z množiny vrcholov  $V = \{V_1, V_2, \dots\}$
- z množiny hrán  $H$ , pričom každá hrana je dvojica  $(v, w)$ , kde  $v, w \in V$ 
  - ak je to neusporiadaná dvojica, hovoríme tomu **neorientovaný** graf
  - ak je to usporiadaná dvojica, hovoríme tomu **orientovaný** graf

Graf budeme znázorňovať takto:

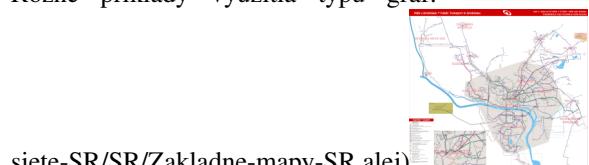
- vrcholy sú kolieska
- hrany sú spojovníky medzi vrcholmi, pričom, ak je graf orientovaný, tak spojovníkmi sú šípky

Graf najčastejšie používame, keď potrebujeme vyjadriť takéto situácie:

- mestá spojené cestami (najčastejšie je to neorientovaný graf: mestá sú vrcholy, hrany označujú cesty medzi mestami)
- križovatky a ulice v meste (križovatky sú vrcholy, hrany sú ulice medzi križovatkami)
- susediace štáty alebo nejaké oblasti (štáty sú vrcholy, hrany označujú, že nejaké štáty susedia)
- mestská verejná doprava (zastávky sú vrcholy, hrany označujú, že existuje linka, ktorá má tieto dve susediace zastávky)
- skupina ľudí, ktorí sa navzájom poznajú (Ľudia sú vrcholy, hrany označujú, že nejaké konkrétnie dve osoby sa poznajú)



Rôzne príklady využitia typu graf:



(<http://www.cdb.sk/sk/Vystupy-CDB/Mapy-cestnej-siete-SR/SR/Zakladne-mapy-SR.alej>)

(<http://imhd.zoznam.sk/ba/media/mn/00000515/Siet-liniek->)



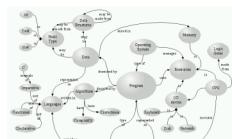
MHD-20120327.png)



(<http://www.mapsofworld.com/images/world-airroute-map.jpg>)

(<http://blog.wcgworld.com/wp->

History of P



content/uploads/2014/04/Networking.jpg)

(<http://cs-alb-pc3.massey.ac.nz/notes/59102/mindmap.gif>)

([http://cdn.oreillystatic.com/news/graphics/prog\\_lang\\_poster.pdf](http://cdn.oreillystatic.com/news/graphics/prog_lang_poster.pdf)) **Dôležité pojmy:**

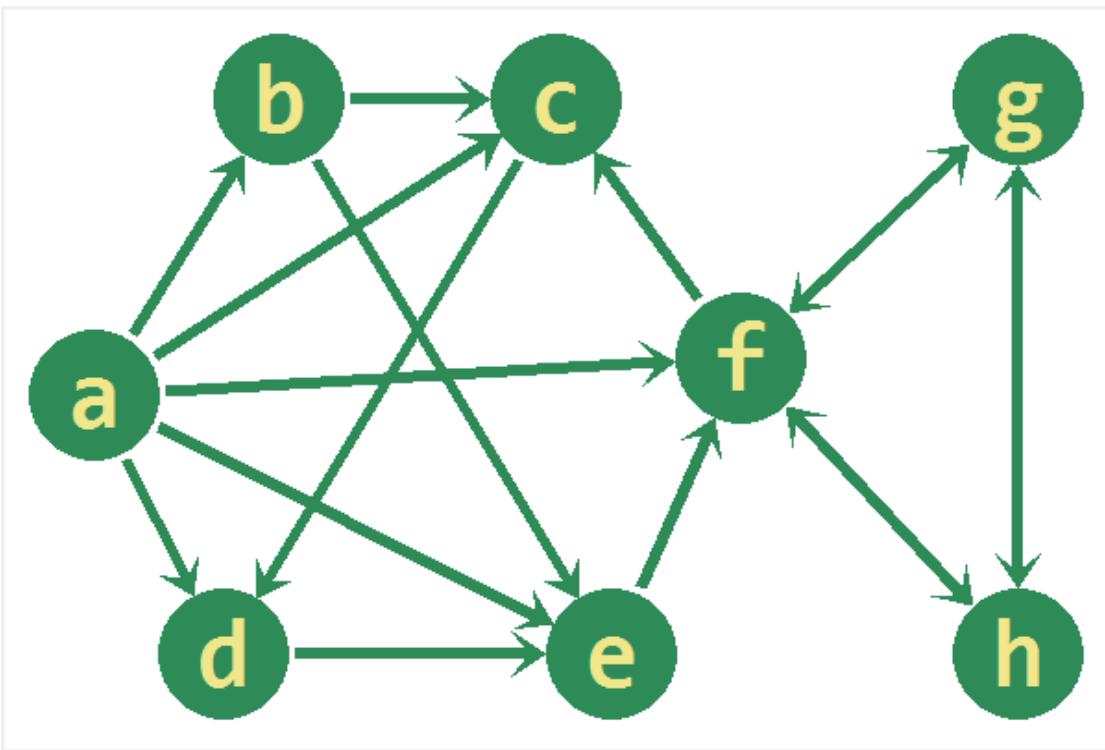
- ak existuje hrana  $(v, w)$  - tak **v** a **w** sú susediace vrcholy
  - v grafe môžeme mať uchované rôzne informácie:
    - v každom vrchole (podobne ako v strome)
    - každá hrana môže mať nejakú hodnotu (tzv. **váha**), vtedy tomu hovoríme **ohodnotený** graf, napr. vzdialenosť dvoch miest, cena cestovného lístka, linky, ktoré spájajú dve zastávky, ...
  - **cesta** je postupnosť vrcholov, ktoré sú spojené hranami
    - dĺžka cesty pre neohodnotený graf počet hrán na ceste, t.j. počet vrcholov cesty - 1
    - dĺžka cesty v ohodnotenom grafe je súčet váh na hranách cesty

- **cyklus** je taká cesta, pre ktorú prvý a posledný vrchol sú rovnaké
  - ak graf neobsahuje ani jeden cyklus, hovoríme že je **acyklický**
- hovoríme, že graf je **súvislý** (spojitý), ak pre každé dva vrcholy  $v, w \in V$ , existuje cesta z  $v$  do  $w$
- niekedy bude pre nás dôležité, keď' nejaký graf bude **súvislý/nesúvislý** bez cyklov, ale aj **súvislý/nesúvislý** s cyklom
- hrane  $(v, v)$  hovoríme **slučka** (toto bude veľmi zriedkavé, ak sa to niekedy objaví, upozorníme na to)
- pojem **komponent** označuje súvislý podgraf, ktorý je disjunktný so zvyškom grafu (neexistuje hrana ku vrcholu vo zvyšku grafu)

Ďalej ukážeme najčastejšie spôsoby reprezentácie grafov. Konkrétna reprezentácia sa potom zvolí väčšinou podľa problémovej oblasti a rôznych obmedzení v zadani.

## 8.2 Reprezentácie

Tento konkrétny graf ukážeme v rôznych reprezentáciách



### 8.2.1 Pole množín susednosti

Aby sme nemuseli pracovať s jednoznakovými reťazcami 'a', 'b', ..., očísľujeme vrcholy číslami 0, 1, 2, ...

Pre čitateľnosť zápisu najprv vytvoríme 8 konštánt  $a=0, b=1, c=2, \dots$  a pomocou nich zadefinujeme celý graf ako pole množín, kde  $i$ -ta množina reprezentuje všetkých susedov  $i$ -teho vrcholu:

```

a,b,c,d,e,f,g,h = range(8)
graf = [ {b,c,d,e,f},   #a
         {c,e},        #b
         {}           #c
         {}           #d
         {}           #e
         {}           #f
         {}           #g
         {}           #h
       ]
  
```

```

{d},          #c
{e},          #d
{f},          #e
{c,g,h},    #f
{f,h},        #g
{f,g},        #h
]

```

Pre túto štruktúru vieme zistiť, počet vrcholov, počet všetkých hrán, stupeň konkrétneho vrcholu a či je hrana medzi dvoma konkrétnymi vrcholmi:

```

>>> print('počet vrcholov:', len(graf))
počet vrcholov: 8
>>> print('počet hrán:', sum(map(len, graf)))
počet hrán: 17
>>> print('stupeň vrcholu f:', len(graf[f]))
stupeň vrcholu f: 3
>>> print('hrana medzi b,e:', e in graf[b])
hrana medzi b,e: True
>>> print('hrana medzi c,a:', a in graf[c])
hrana medzi c,a: False

```

Túto reprezentáciu môžeme “zabalíť” do triedy napr. takto:

```

class Graf:
    def __init__(self, pole=None):
        if pole is None:
            self.pole = []
        else:
            self.pole = pole

    def pridaj_hranu(self, v1, v2):
        while len(self.pole)-1 < max(v1,v2):
            self.pole.append(set())
        self.pole[v1].add(v2)

    def je_hrana(self, v1, v2):
        try:
            return v2 in self.pole[v1]
        except IndexError:
            return False

    def daj_vrcholy(self):
        return list(range(len(self.pole)))

    def daj_hrany(self):
        return [(v1,v2) for v1 in range(len(self.pole)) for v2 in self.
-pole[v1]]

    def stupen(self, v=None):
        if v is not None:
            return len(self.pole[v])
        return max(map(len, self.pole))

    def __str__(self):
        return 'vrcholy: {}\nhrany: {}'.format(self.daj_vrcholy(), self.daj_
-hrany())

```

```
def __repr__(self):
    return 'Graf({})'.format(self.pole)
```

a graf môžeme teraz vytvoriť napr. takto:

```
>>> graf = Graf()
>>> graf.pridaj_hranu(a,b)
>>> graf.pridaj_hranu(b,e)
>>> graf.pridaj_hranu(f,h)
>>> graf.pridaj_hranu(g,f)
>>> graf
Graf([{1}, {4}, set(), set(), set(), {7}, {5}, set()])
>>> print(graf)
vrcholy: [0, 1, 2, 3, 4, 5, 6, 7]
hrany: [(0, 1), (1, 4), (5, 7), (6, 5)]
```

Všimnite si, že sme vyrobili dve rôzne metódy `__repr__()` a `__str__()`. Metóda `__repr__()` sa zavolá, napr. keď v dialógovom režime zapíšeme `>>> graf`. Metóda `__str__()` sa zavolá, napr. keď budeme graf vypisovať príkazom `print()`.

Tak ako sme túto triedu **Graf** definovali, je trochu komplikované vytvoriť izolovaný vrchol, t.j. taký, z ktorého nevychádza žiadna hrana (asi by pomohla metóda `pridaj_vrchol()`).

Ak máme vyrobenú štruktúru grafu `graf`, vieme z nej priamo vytvoriť triedu graf:

```
>>> graf = Graf(G)
>>> print(graf)
vrcholy: [0, 1, 2, 3, 4, 5, 6, 7]
hrany: [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (1, 4), (2, 3), (3, 4),
         (4, 5), (5, 2), (5, 6), (5, 7), (6, 5), (6, 7), (7, 5), (7, 6)]
>>> graf.je_hrana(b,e)
True
>>> graf.je_hrana(c,a)
False
>>> print('stupen vrcholu f:', graf.stupen(f))
stupen vrcholu f: 3
```

V tejto reprezentácii grafu triedou **Graf** o samotnom vrchole nemáme žiadnu špeciálnu informáciu okrem jeho poradového čísla a prislúchajúcej množiny susedov.

Zadefinujeme novú triedu **Graf**, ktorá bude uchovávať pole vrcholov, t.j. pole objektov typu **Vrchol**:

```
class Vrchol:
    def __init__(self, meno):
        self.meno = meno
        self.sus = set()

class Graf:
    def __init__(self):
        self.pole = []

    def pridaj_vrchol(self, meno):
        for v in self.pole:
            if v.meno == meno:
                return v # uz existuje
        self.pole.append(Vrchol(meno))
        return self.pole[-1]

    def hladaj_vrchol(self, meno):
```

```

for v in self.pole:
    if v.meno == meno:
        return v
return None

def pridaj_hranu(self, v1, v2):
    self.pridaj_vrchol(v1).sus.add(v2)

def je_hrana(self, v1, v2):
    try:
        return v2 in self.hladaj_vrchol(v1).sus
    except AttributeError:
        return False

def daj_vrchooly(self):
    return [v.meno for v in self.pole]

def daj_hrany(self):
    return [(v.meno, v2) for v in self.pole for v2 in v.sus]

def stupen(self, v=None):
    if v is not None:
        return len(self.hladaj_vrchol(v).sus)
    return max(len(v.sus) for v in self.pole)

def __str__(self):
    return 'vrchooly: {}\nhrany: {}'.format(self.daj_vrchooly(), self.daj_hrany())

```

S vrcholmi teraz musíme pracovať cez ich identifikátory, t.j. jednoznakové reťazce, napr.

```

>>> graf = Graf()
>>> for v1, v2 in ('ab', 'ac', 'ad', 'ae', 'af', 'bc', 'be', 'cd', 'de',
...                 'ef', 'fc', 'fg', 'fh', 'gf', 'gh', 'hf', 'hg'):
...     graf.pridaj_hranu(v1, v2)
>>> print(graf)
vrchooly: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
hrany: [('a', 'c'), ('a', 'b'), ('a', 'e'), ('a', 'd'), ('a', 'f'), ('b', 'c'),
         ('b', 'e'), ('c', 'd'), ('d', 'e'), ('e', 'f'), ('f', 'g'), ('f', 'c'),
         ('f', 'h'), ('g', 'f'), ('h', 'g'), ('h', 'f')]
>>> print('stupen vrcholu f:', graf.stupen('f'))
stupen vrcholu f: 3
>>> print('stupen grafu:', graf.stupen())
stupen grafu: 5
>>> print('hrana medzi b,e:', graf.je_hrana('b', 'e'))
hrana medzi b,e: True
>>> print('hrana medzi c,a:', graf.je_hrana('c', 'a'))
hrana medzi c,a: False
>>> print('počet hran:', len(graf.daj_hrany()))
počet hran: 17

```

Zrejme efektívnejšie by sa táto definícia realizovala nie poľom vrcholov, ale asociatívnym poľom (typ dict) vrcholov, v ktorom klúčom by bol identifikátor vrcholu (uvidíme nižšie).

## 8.2.2 Pole zoznamov susednosti

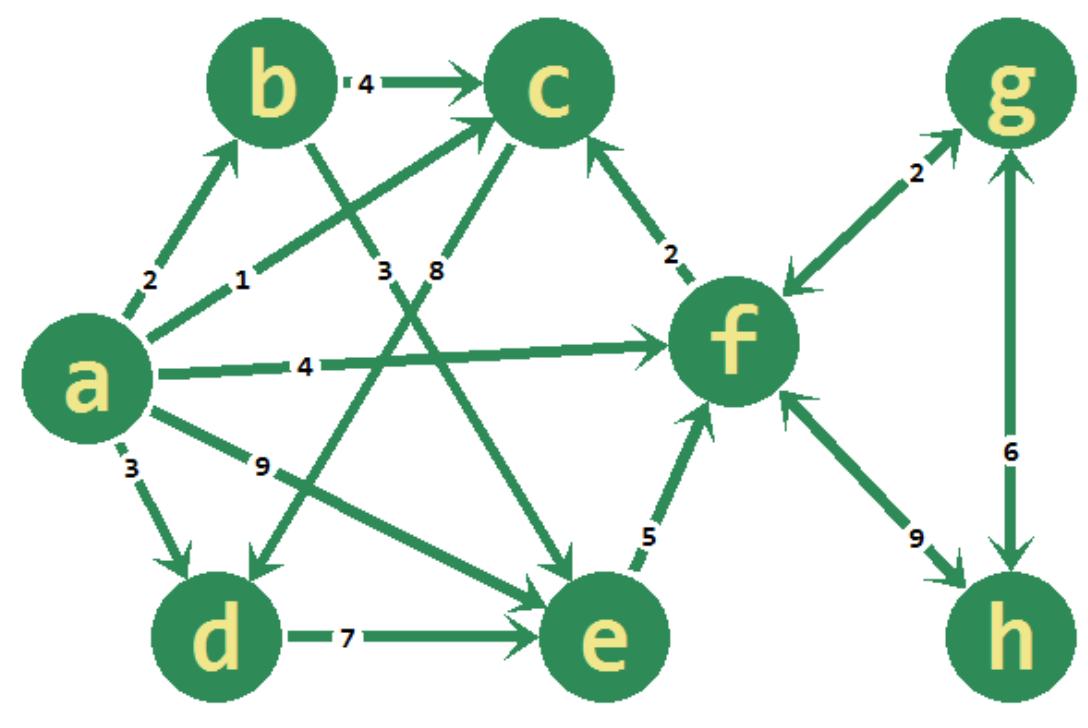
Namiesto množiny susedov využijeme pole (teda pythonovský zoznam):

```
a, b, c, d, e, f, g, h = range(8)
graf = [ [b, c, d, e, f],      #a
         [c, e],            #b
         [d],              #c
         [e],              #d
         [f],              #e
         [c, g, h],        #f
         [f, h],            #g
         [f, g],            #h
     ]
```

Inak je to to isté, ako predchádzajúca reprezentácia, len namiesto množinových operácií (napr. `add()`) musíme pracovať s poľom (napr. použijeme `append()`).

## 8.2.3 Pole asociatívnych polí susednosti

Ak máme ohodnotený graf, napr.



tak namiesto množiny susedností (prvá reprezentácia grafu) použijeme asociatívne pole (typ `dict`), v ktorom si pri každom susedovi budeme pamätať aj číslo na hrane, t.j. váhu:

```
a, b, c, d, e, f, g, h = range(8)
graf = [ {b:2, c:1, d:3, e:9, f:4},      #a
         {c:4, e:3},            #b
         {d:8},              #c
         {e:7},              #d
```

```
{f:5},          #e
{c:2,g:2,h:9}, #f
{f:2,h:6},     #g
{f:9,g:6},     #h
]
```

### 8.2.4 Asociatívne pole množín susednosti

Pri realizácii reprezentácie **množiny susednosti** pomocou tried `Graf` a `Vrchol` sa ukázalo nešikovné, keď sme mali všetky vrcholy uložené v poli. Pri práci s takýmito vrcholmi sme museli veľakrát preliezať celé pole a hľadať vrchol s daným identifikátorom. Ak by sme namiesto toho použili asociatívne pole, výrazne by sa to zjednodušilo, napr.

```
graf = {'a': set('bcdef'),   # {'b', 'c', 'd', 'e', 'f'}
        'b': set('ce'),
        'c': set('d'),
        'd': set('e'),
        'e': set('f'),
        'f': set('cgh'),
        'g': set('fh'),
        'h': set('fg'),
    }
```

Zapíšeme to pomocou tried `Graf` a `Vrchol`:

```
class Vrchol:
    def __init__(self, meno):
        self.meno = meno
        self.sus = set()

    def pridaj_hranu(self, v2):
        self.sus.add(v2)

    def __contains__(self, v2):
        return v2 in self.sus

class Graf:
    def __init__(self):
        self.pole = {}

    def pridaj_vrchol(self, meno):
        if meno not in self.pole:
            self.pole[meno] = Vrchol(meno)

    def pridaj_hranu(self, v1, v2):
        self.pridaj_vrchol(v1)
        self.pole[v1].pridaj_hranu(v2)

    def je_hrana(self, v1, v2):
        try:
            return v2 in self.pole[v1]
        except KeyError:
            return False

    def daj_vrcholy(self):
        return list(self.pole.keys())
```

```

def daj_hrany(self):
    return [(v1,v2) for v1,v in self.pole.items() for v2 in v.sus]

def stupen(self, v=None):
    if v is not None:
        return len(self.pole[v].sus)
    return max(len(v.sus) for v in self.pole.values())

def __str__(self):
    return 'vrcholy: {}\nhranы: {}'.format(self.daj_vrcholy(), self.daj_
→hrany())

```

## 8.2.5 Matica susedností

Graf zapíšeme do dvojrozmerného poľa veľkosti  $n \times n$ , kde  $n$  je počet vrcholov:

```

a,b,c,d,e,f,g,h = range(8)
graf = [[0,1,1,1,1,0,0],
        [0,0,1,0,1,0,0,0],
        [0,0,0,1,0,0,0,0],
        [0,0,0,0,1,0,0,0],
        [0,0,0,0,0,1,0,0],
        [0,0,0,0,0,0,1,1],
        [0,0,1,0,0,0,1,1],
        [0,0,0,0,0,1,0,1],
        [0,0,0,0,0,1,1,0],
        ]

```

Často namiesto **0** a **1** sa píšu `False` a `True`.

Otestujme, ako sa zapisuje práca s takouto reprezentáciou:

```

>>> print('počet vrcholov:', len(graf))
počet vrcholov: 8
>>> print('počet hran:', sum(map(sum,graf)))
počet hran: 17
>>> print('stupen vrcholu f:', sum(grafG[f]))
stupen vrcholu f: 3
>>> print('hrana medzi b,e:', graf[b][e]==1)
hrana medzi b,e: True
>>> print('hrana medzi c,a:', graf[c][a]==1)
hrana medzi c,a: False

```

## 8.2.6 Matica susedností s váhami

Namiesto **0** a **1** zapisujeme priamo váhy, pričom treba nejak označiť hodnotu, ktorá reprezentuje, že tu nie je hrana, napr.

```

a,b,c,d,e,f,g,h = range(8)
_ = -1           # označuje, že tu nie je hrana, aj keď je iná hodnota napr. None
graf = [[_,2,1,3,9,4,_,_],
        [_,_,4,_,3,_,_,_],
        [_,_,_,8,_,_,_,_],
        [_,_,_,_,7,_,_,_],
        ]

```

```
[_,_,_,_,_,5,_,_],  
[_,_,2,_,_,2,9],  
[_,_,_,_,_,2,_,6],  
[_,_,_,_,_,9,6,_,]  
]
```

Otestujme, ako sa zapisuje práca s takouto reprezentáciou:

```
>>> print('počet vrcholov:', len(graf))  
počet vrcholov: 8  
>>> print('počet hran:', sum(1 for r in graf for x in r if x!=_))  
počet hran: 17  
>>> print('stupen vrcholu f:', sum(1 for x in graf[f] if x!=_))  
stupen vrcholu f: 3  
>>> print('hrana medzi b,e:', graf[b][e]!=_)  
hrana medzi b,e: True  
>>> print('hrana medzi c,a:', graf[c][a]!=_)  
hrana medzi c,a: False  
>>> print('vaha na hrane medzi b,e:', graf[b][e])  
vaha na hrane medzi b,e: 3
```

## Prehľadávanie grafu

Budeme skúmať rôzne algoritmy, ktoré prechádzajú vrcholy grafu v nejakom poradí. Keďže sa zameriame na **neohodnotené neorientované grafy**, zvolíme reprezentáciu pole množín susedností. Inštancia triedy Graf bude obsahovať atribút (súkromnú premennú) pole, ktoré bude poľom vrcholov grafu. Samotné vrcholy zadefinujeme vo vnorenej triede Vrchol a preto s ním musíme v triede Graf pracovať pomocou `self.Vrchol`. Graf obsahuje aj metódy, aby sa dal vykresliť do grafickej plochy:

```
import tkinter

class Graf:

    class Vrchol:
        c = None # canvas
        def __init__(self, meno, x, y):
            self.meno = meno
            self.xy = x, y
            self.sus = set()

        def kresli(self):
            x, y = self.xy
            self.c.create_oval(x-10, y-10, x+10, y+10, fill='lightgray')
            self.c.create_text(x, y, text=self.meno)

    #-----

        def __init__(self):
            self.pole = []
            self.c = tkinter.Canvas(bg='white', width=600, height=600)
            self.c.pack()
            self.Vrchol.c = self.c

        def pridaj_vrchol(self, x, y):
            meno = len(self.pole)
            self.pole.append(self.Vrchol(meno, x, y))

        def pridaj_hranu(self, v1, v2):
            self.pole[v1].sus.add(v2)
            self.pole[v2].sus.add(v1)

        def je_hrana(self, v1, v2):
            return v2 in self.pole[v1].sus

        def kresli_hranu(self, v1, v2):
```

```
    self.c.create_line(self.pole[v1].xy, self.pole[v2].xy, width=3, fill=
        ↵'gray')

    def kresli(self):
        for v1 in range(len(self.pole)):
            for v2 in range(v1+1, len(self.pole)):
                if self.je_hrana(v1, v2):
                    self.kresli_hranu(v1, v2)
        for vrch in self.pole:
            vrch.kresli()
```

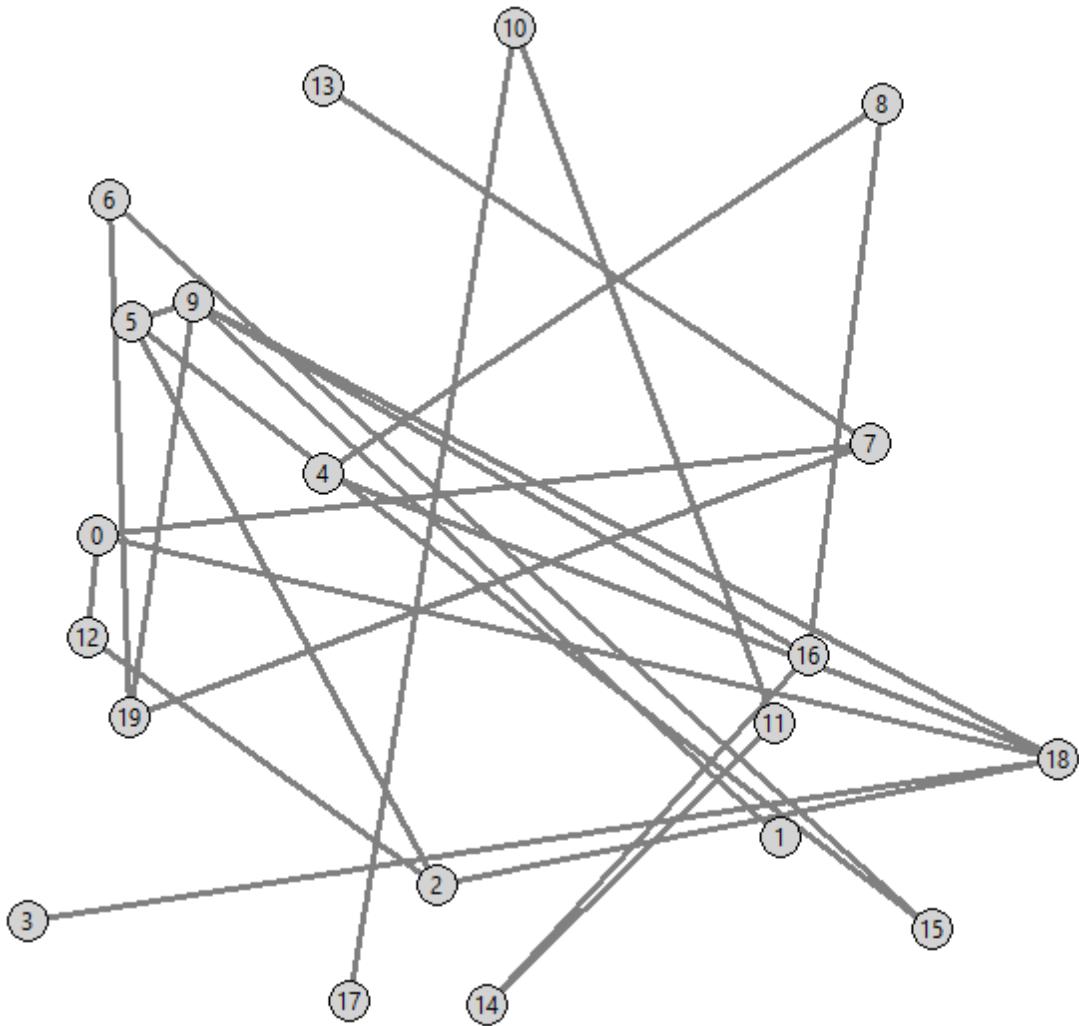
Vygenerujme graf s 20 náhodne rozloženými vrcholmi, pričom sa niektoré vrcholy grafu náhodne pospájajú hranami:

```
#testovanie

import random

g = Graf()
for i in range(20):
    g.pridaj_vrchol(random.randint(20, 580), random.randint(20, 580))
    for j in range(i):
        if random.randrange(8) == 0:
            g.pridaj_hranu(i, j)
g.kresli()
```

Takéto úplne náhodne vygenerované grafy sú veľmi neprehľadné a veľmi ľahko by sa na nich sledovali nejaké prehľadávacie algoritmy.



Vygenerujme preto vrcholy grafu do nejakej mriežky a spájať hranami budeme len niektoré susedné vrcholy v mriežke. Toto generovanie vrcholov zabudujeme priamo do inicializácie triedy `__init__()`, napr. takto:

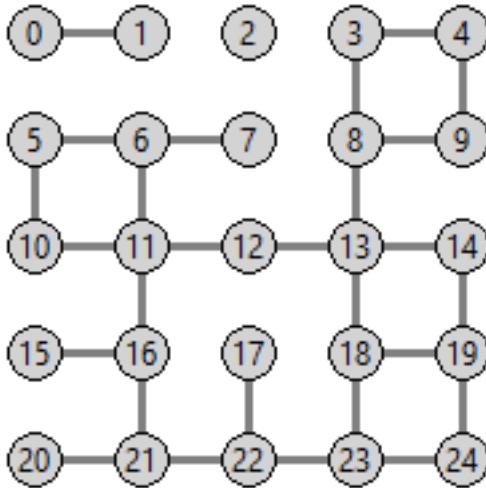
```
import tkinter, random

class Graf:
    ...

    def __init__(self, n):
        self.pole = []
        for i in range(n):
            for j in range(n):
                self.pridaj_vrchol(j*40+20, i*40+20)
                if j > 0 and random.randrange(3):
                    self.pridaj_hranu(i*n+(j-1), i*n+j)
                if i > 0 and random.randrange(3):
                    self.pridaj_hranu((i-1)*n+j, i*n+j)
        self.c = tkinter.Canvas(bg='white', width=600, height=600)
        self.c.pack()
        self.Vrchol.c = self.c
        self.kresli()
```

```
...
#testovanie
g = Graf(5)
```

Všimnite si, že parameter n teraz označuje veľkosť siete vrcholov, ktorá bude mať n x n vrcholov:



Počet náhodne generovaných hrán bude závisieť od konštant pri volaní `random.randrange(3)`: vyskúšajte parameter 3 nahradí 2 a uvidíte "redší" graf, v ktorom je už menej hrán.

## 9.1 Algoritmus do hĺbky

Prehľadávanie grafu označuje taký algoritmus, ktorý postupne v nejakom poradí prechádza vrcholy grafu. S každým vrcholom pritom vykoná nejakú akciu (napr. ho zafarbí) a d'alej pokračuje na niektorom z jeho susedných vrcholov. Prehľadávanie sa začne z nejakého (ľubovoľného) vrcholu. Každý vrchol sa pritom navštíví **maximálne raz**.

Ukážeme dva základné algoritmy:

- **prehľadávanie do hĺbky** - funguje podobne ako **preorder** na stromoch: najprv spracuje vrchol a potom po stupne pokračuje v prehľadávaní všetkých svojich susedov - opäť rekurzívnym algoritmom
- **prehľadávanie do šírky** - prehľadáva vrcholy podobne ako prehľadávanie v stromoch po úrovniach - najprv všetky, ktoré majú vzdialenosť len 1, potom všetky, ktoré majú vzdialenosť presne 2, atď.

Pri prehľadávaní grafu pomocou nejakého algoritmu si potrebujeme evidovať, ktoré vrcholy sa už spracovali. Kedže samotný algoritmus môže byť rekurzívny, túto evidenciu nemôžeme robiť v lokálnej premennej samotnej funkcie - väčšinou budeme používať nový atribút triedy `Graf` množinovú premennú `visited`:

```
self.visited = set()
```

Algoritmus do hĺbky začína prehľadávanie vo vrchole v1 a využíva ďalšiu metódu triedy `Vrchol`, ktorá ho zafarbí:

```
import tkinter, random
class Graf:
```

```

class Vrchol:
    c = None                                # canvas
    def __init__(self, meno, x, y):
        self.meno = meno
        self.xy = x, y
        self.sus = set()

    def kresli(self):
        x, y = self.xy
        self.id = self.c.create_oval(x-10, y-10, x+10, y+10, fill=
→'lightgray')
        self.c.create_text(x, y, text=self.meno)

    def zafarbi(self, farba):
        self.c.itemconfig(self.id, outline=farba, width=2)

    #-----
    ...

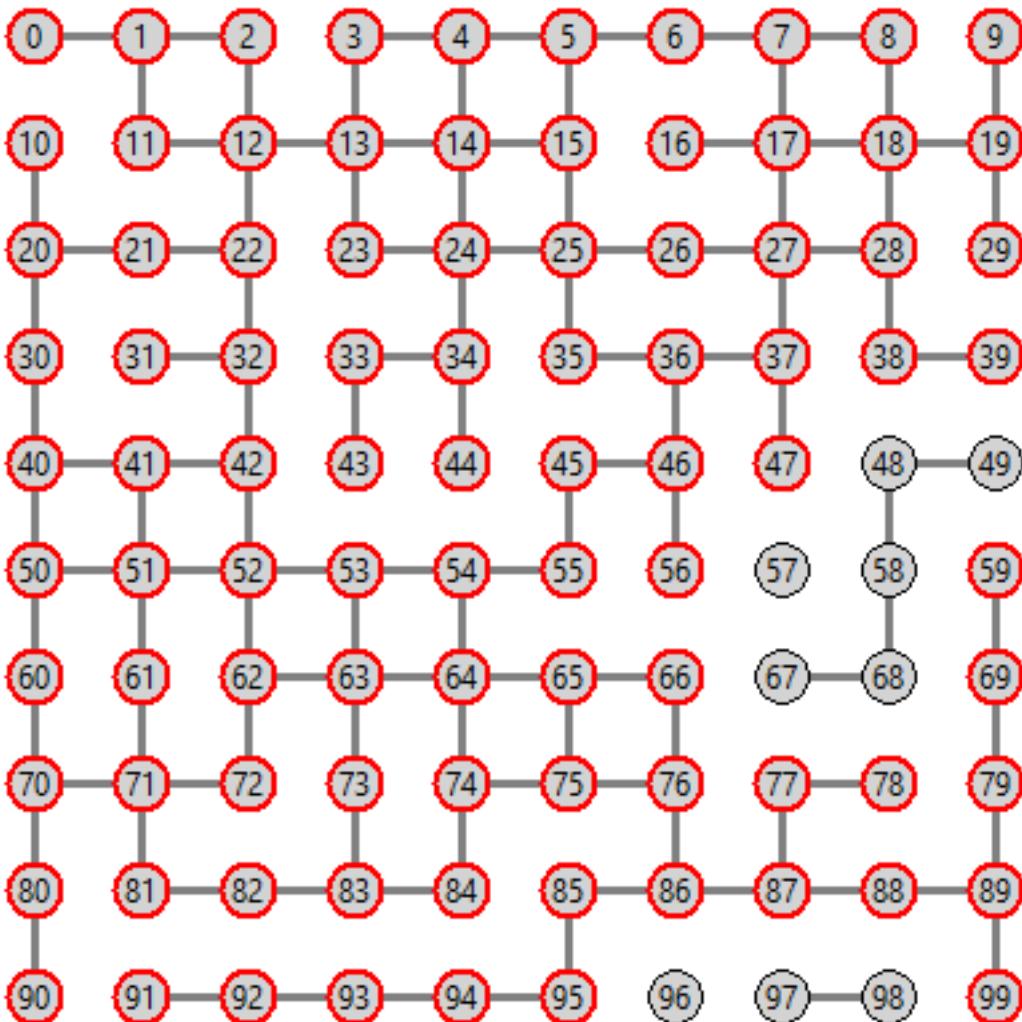
    def dohlbky(self, v1):
        self.visited.add(v1)
        self.pole[v1].zafarbi('red')
        self.c.update()
        self.c.after(100)
        for v2 in self.pole[v1].sus:
            if v2 not in self.visited:
                self.dohlbky(v2)

#testovanie

g = Graf(10)
g.visited = set()
g.dohlbky(0)

```

Algoritmus **dohlbky** sa naštartoval vo vrchole 0 (v ľavom hornom rohu grafu), postupne navštevoval (a pritom zafarboval) jeho susedov a ich susedov atď. Nenavštívené (nezafarbené) ostali tie časti grafu, ktoré nie sú spojené so zvyškom nejakými hranami, napr.



Atribút `visited` sme tu zadefinovali mimo metód triedy `Graf` - čo samozrejme nie je pekné. Často to budeme riešiť ďalšou metódou, napr.

```
class Graf:

    ...

    def dohlbky(self, v1):
        self.visited.add(v1)
        self.pole[v1].zafarbi('red')
        ##      self.c.update()
        ##      self.c.after(100)
        for v2 in self.pole[v1].sus:
            if v2 not in self.visited:
                self.dohlbky(v2)

    def start(self, v1):
        self.visited = set()
        self.dohlbky(v1)

#testovanie
```

```
g = Graf(10)
g.start(55)
```

Ak metódu dohlbky() plánujeme volať len z metódy start(), samotná funkcia dohlbky() tu môže byť vnořená a vtedy aj premenná visited nemusí byť atribútom triedy, ale obyčajnou premennou. Ukážeme to na metóde velkost\_komponentu. Táto metóda dostáva ako parameter jeden vrchol v1 (jeho poradové číslo), od tohto vrcholu spustí algoritmus do hĺbky, nič pritom nezafarbuje "len" sa spolieha na to, že po jeho skončení bude v premennej visited množina všetkých navštívených vrcholov, t.j. tých, ktoré sú spolu s v1 v jednom komponente:

```
class Graf:

    ...

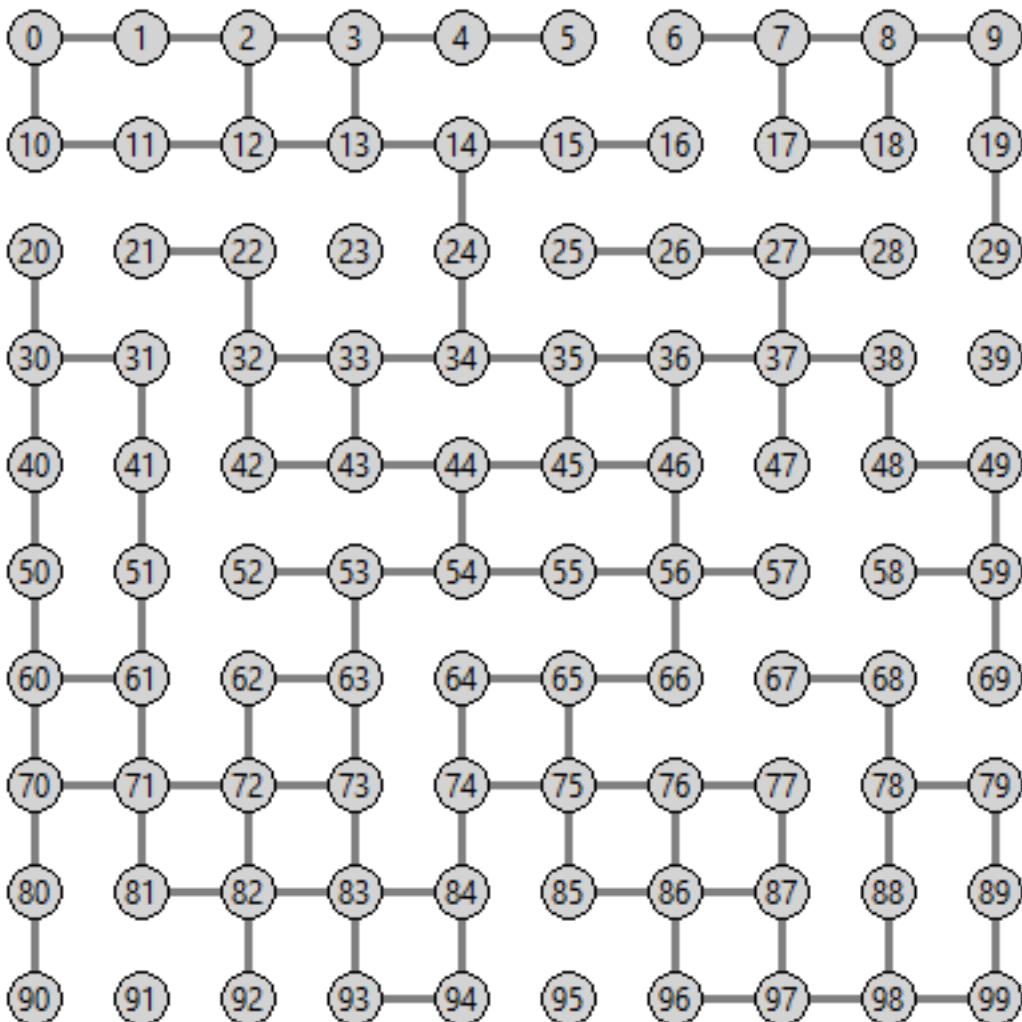
    def velkost_komponentu(self, v1):
        def dohlbky(v1):
            visited.add(v1)
            for v2 in self.pole[v1].sus:
                if v2 not in visited:
                    dohlbky(v2)

            visited = set()
            dohlbky(v1)
        return len(visited)

#testovanie

g = Graf(10)
```

Dostávame napr.



teraz zavoláme:

```
>>> g.velkost_komponentu(55)
88
>>> g.velkost_komponentu(9)
8
```

Takto môžeme zadefinovať niekoľko verzií algoritmu dohlbky pre rôzne využitie: napr. zisťovanie množiny vrcholov v komponente, zisťovanie počtu vrcholov v komponente, zisťovanie počtu komponentov, zafarbovanie vrcholov v rôznych komponentoch rôznymi farbami, rôzne manipulácie s vrcholmi v jednom komponente, ...

Vidíme, že po skončení algoritmu dohlbky () bude premenná visited obsahovať množinu všetkých navštívených vrcholov. Na základe toho, môžeme ľahko zistiť, či je graf súvislý: zistíme veľkosť komponentu, ktorý obsahuje napr. vrchol 0, ak obsahuje všetky vrcholy grafu, zrejme je graf súvislý. Zapíšeme to takto:

```
class Graf:
    ...
    def je_suvisly(self):
        def dohlbky(v1):
            visited.add(v1)
```

```

        for v2 in self.pole[v1].sus:
            if v2 not in visited:
                dohlbky(v2)

    visited = set()
    dohlbky(0)
    return len(visited) == len(self.pole)

#testovanie

g = Graf(10)
print('graf je suvisly:', g.je_suvisly())

```

## 9.2 Všetky komponenty grafu

Algoritmus dohlbky() využijeme nielen pre jeden komponent grafu, ale postupne pre všetky. Ďalšia metóda zafarbuje všetky komponenty grafu - každý komponent inou (náhodnou) farbou:

```

class Graf:

    ...

    def zafarbi_komponenty(self):
        def dohlbky(v1, farba):
            visited.add(v1)
            self.pole[v1].zafarbi(farba)
            for v2 in self.pole[v1].sus:
                #self.zafarbi_hranu(v1,v2,farba)
                if v2 not in visited:
                    dohlbky(v2,farba)

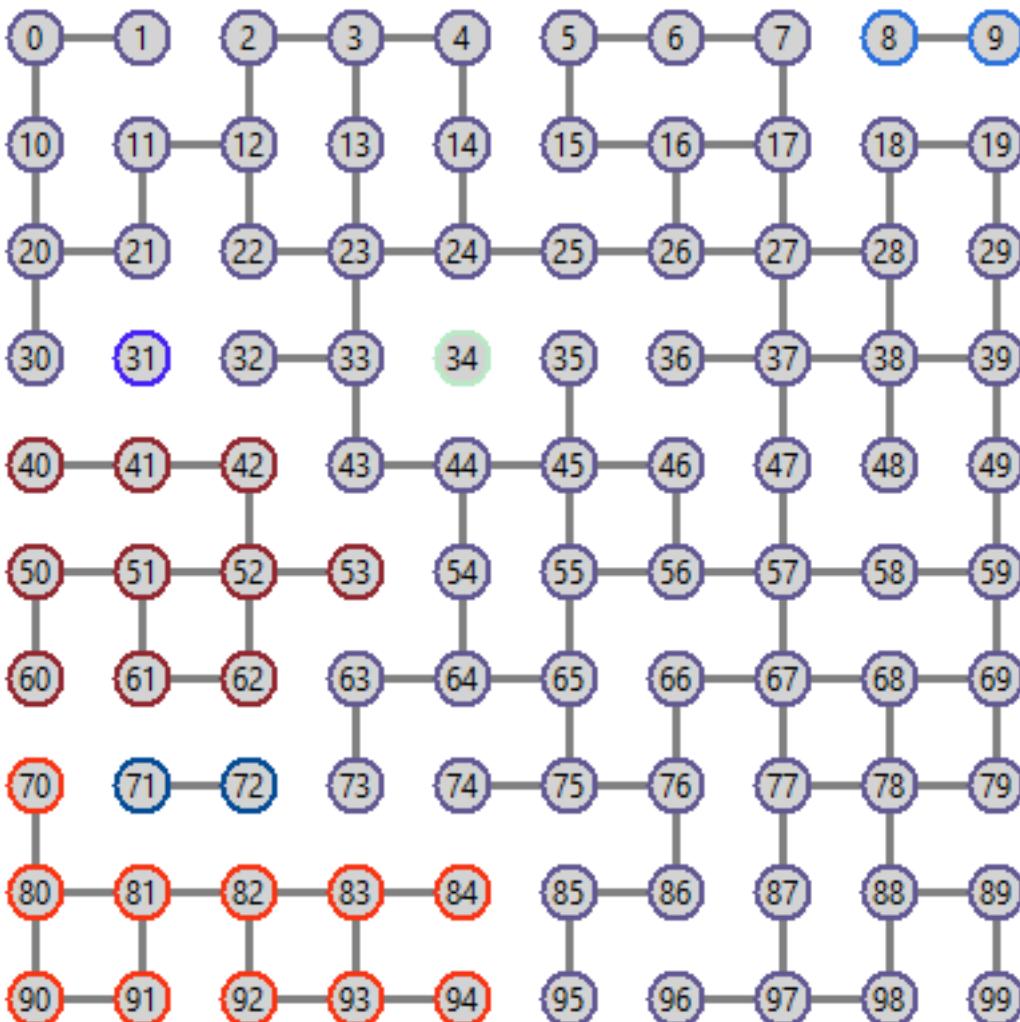
        visited = set()
        for v1 in range(len(self.pole)):
            if v1 not in visited:
                dohlbky(v1, '#{:06x}'.format(random.randrange(256**3)))

#testovanie

g = Graf(10)
g.zafarbi_komponenty()

```

Dostávame napr.



Tu sa oplatí vygenerovať trochu ľahší graf: pri generovaní grafu v inicializácii môžeme nastaviť napr. `random.randrange(2)`.

Všimnite si, že týmto algoritmom môžeme zabezpečiť aj farbenie všetkých hrán vrcholov komponentu. Ak chcete vidieť aj zafarbené hrany komponentu, okrem odkomentovania volanie metódy `zafarbi_hranu()` ju treba aj zadefinovať. Budeme musieť zasahovať aj do iných metód (`kresli()`, `kresli_hranu()`):

```
class Graf:

    ...

    def kresli_hranu(self, v1, v2):
        self.id[v1,v2] = self.c.create_line(self.pole[v1].xy, self.pole[v2].xy, width=3, fill='gray')

    def kresli(self):
        self.id = {} # na zapamatie id kazdej nakreslenej hrany
        for v1 in range(len(self.pole)):
            for v2 in range(v1+1, len(self.pole)):
                if self.je_hrana(v1, v2):
                    self.kresli_hranu(v1, v2)
```

```

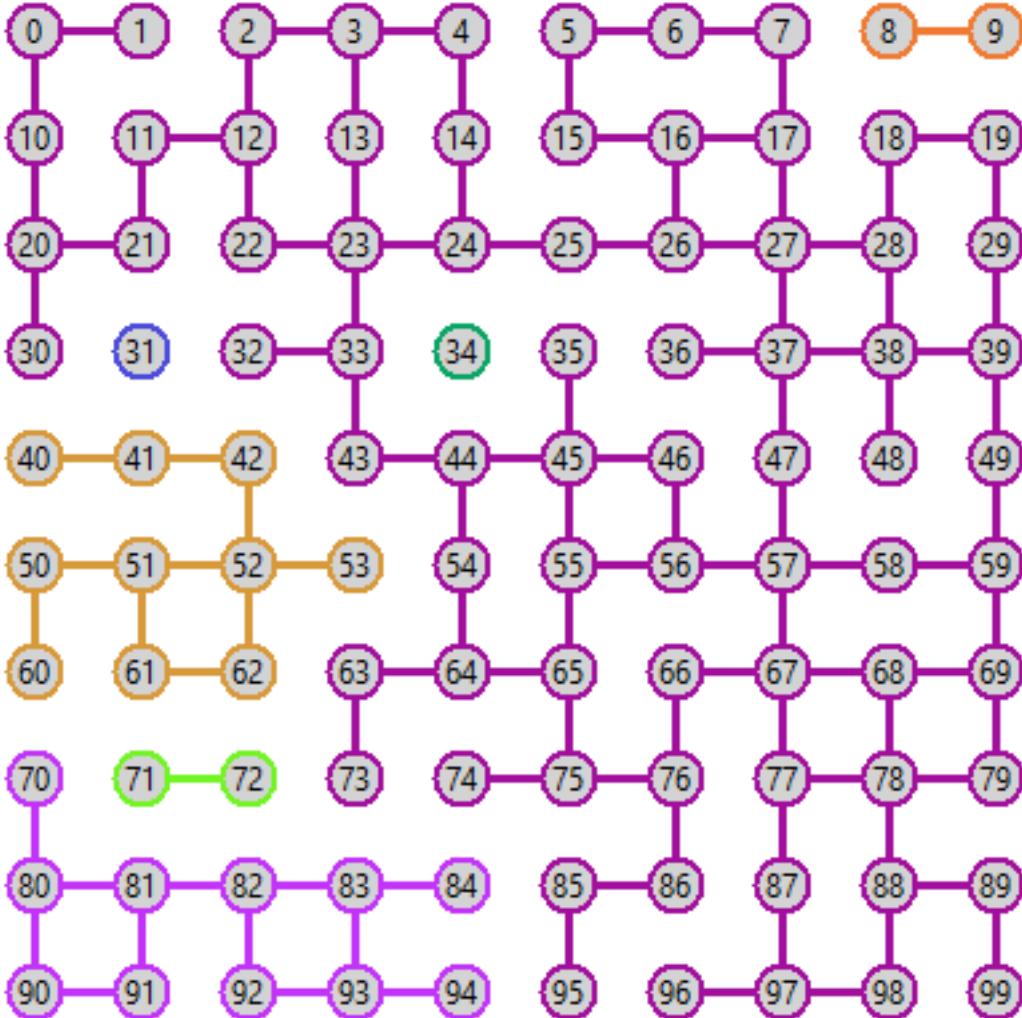
for vrch in self.pole:
    vrch.kresli()

def zafarbi_hranu(self, v1, v2, farba):
    if v1 > v2:
        v1, v2 = v2, v1
    self.c.itemconfig(self.id[v1, v2], fill=farba)

...

```

Teraz (odkomentujeme aj farbenie hrany v `zafarbi_komponenty()`) zafarbený graf vyzera napr. takto:



Ešte ukážeme metódu, ktorá zistí už počet komponentov grafu a je veľmi podobná metóde na farbenie komponentov:

```

class Graf:
    ...

    def pocet_komponentov(self):
        def dohlobky(v1):
            visited.add(v1)
            for v2 in self.pole[v1].sus:

```

```

        if v2 not in visited:
            dohlbky(v2)

    visited = set()
    vysl = 0
    for v1 in range(len(self.pole)):
        if v1 not in visited:
            dohlbky(v1)
            vysl += 1
    return vysl

#testovanie

g = Graf(10)
print('počet komponentov:', g.pocet_komponentov())
g.zafarbi_komponenty()

```

a dozvieme sa, že naposledy zobrazený graf má 7 komponentov.

Vidíme, že algoritmus do hĺbky môžeme použiť napríklad na:

- zistenie počtu komponentov grafu, resp. zist'ovanie, či je graf súvislý
- zistenie, či sú dva vrcholy v tom istom komponente
- zist'ovanie počtov vrcholov v jednotlivých komponentoch grafu (veľkosti komponentov)
- zist'ovanie najväčšieho komponentu grafu
- zafarbovanie vrcholov a hrán jednotlivých komponentov grafu

### 9.3 Nerekurzívny algoritmus do hĺbky

Vieme, že rekurzia v Pythone má obmedzenú hĺbku volaní (okolo 1000), preto pre niektoré väčšie grafy rekurzívny algoritmus dohlbky() môže spadnúť na príliš veľa rekurzívnych volaní (napr. už pre Graf(50) má rekurzia veľkú šancu spadnúť).

Prepíšme preto rekurzívny algoritmus dohlbky() na nerekurzívnu verziu. Ukážeme to na metóde velkost\_komponentu(), ktorá používa rekurzívnu vnorenú funkciu dohlbky(). Na odstránenie rekurzie využijeme zásobník: mohli by sme použiť už predtým definovanú triedu Stack, ale vystačíme si s pomocným poľom s metódami append() (namiesto push) a pop(). Nerekurzívny algoritmus môže vyzeráť napr. takto:

```

class Graf:

    ...

    def velkost_komponentu(self, v1):      # povodna rekurzivna verzia
        def dohlbky(v1):
            visited.add(v1)
            for v2 in self.pole[v1].sus:
                if v2 not in visited:
                    dohlbky(v2)

        visited = set()
        dohlbky(v1)
        return len(visited)

    def velkost_komponentu2(self, v1):      # upravena nerekurzivna verzia

```

```

visited = set()
stack = [v1]           # vlož do zasobnika startovy vrchol
while stack:           # kym je zasobnik neprazdny
    v1 = stack.pop()   # vyber z vrchu zasobnika
    visited.add(v1)
    for v2 in self.pole[v1].sus:
        if v2 not in visited:
            stack.append(v2) # vlož do zasobnika namiesto volania
→dohlbky(v2)
return len(visited)

#testovanie

g = Graf(10)
print(g.velkost_komponentu(0))
print(g.velkost_komponentu2(0))

```

Hoci tento algoritmus vyzerá v poriadku, ak ho otestujeme na nejakom jednoduchom grafe, dá sa zistíť, že niektorý vrchol sa bude spracovávať (navštěvovať) viackrát a to môže byť v niektorých situáciach neprijateľné. Ak by sme si algoritmus odstrasovali ručne, zistili by sme, že niektoré čísla vrcholov sa dostávajú do zásobníka viackrát (napr. vrchol, ktorý ešte neboli navštívený, ale je susedom viacerých už navštívených vrcholov). Jednoducho to môžeme preveriť aj tak, že do tohto algoritmu vložíme počítadlo navštívených vrcholov a po skončení algoritmu porovnáme tento počet so skutočným počtom vrcholov komponentu. Napr. pridáme premennú `pocet`:

```

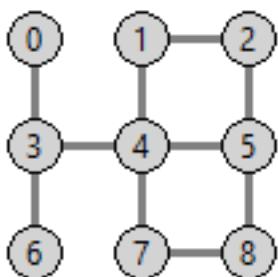
class Graf:

    ...

    def velkost_komponentu2(self, v1):
        visited = set()
        stack = [v1]           # vlož do zasobnika startovy vrchol
        pocet = 0
        while stack:           # kym je zasobnik neprazdny
            v1 = stack.pop()   # vyber z vrchu zasobnika
            visited.add(v1)
            pocet += 1
            for v2 in self.pole[v1].sus:
                if v2 not in visited:
                    stack.append(v2) # vlož do zasobnika namiesto volania
→dohlbky(v2)
        return len(visited), pocet

```

Ak teraz spustíme `velkost_komponentu2()` s rôzne veľkými a s rôzne hustými grafmi, často dostaneme dve rôzne čísla: počet vrcholov v komponente a počet navštívených vrcholov je rôzny. Napr. už aj pre tento graf, ktorý má veľkosť komponentu 9 (je súvislý), navštívil niektoré vrcholy viackrát a vidíme:



```
>>> g = Graf(3)
>>> g.velkost_komponentu2(0)
(9, 11)
```

Správne by mal nerekurzívny algoritmus vyzeráť takto:

```
class Graf:

    ...

    def velkost_komponentu2(self, v1):
        visited = set()
        stack = [v1]
        while stack:
            v1 = stack.pop()
            if v1 not in visited:
                visited.add(v1)
                for v2 in self.pole[v1].sus:
                    if v2 not in visited:
                        stack.append(v2)
        return len(visited)
```

Teda, každý vrchol, ktorý vyberieme zo zásobníka (`stack.pop()`), ešte skontrolujeme, či už náhodou neboli spracovaný skôr.

## 9.4 Algoritmus do šírky

Ak by sme potrebovali prehľadávať napr. binárny strom po úrovniach (algoritmus do šírky), potrebovali by sme na to dátovú štruktúru **rad** (`queue`). Podobne budeme postupovať aj pri algoritme pre graf. Tento algoritmus sa bude od nerekurzívného algoritmu do hĺbky lísiť len tým, že namiesto `stack` použijeme `queue` (a teda zmeníme aj metódu na výber z radu - namiesto `pop()` použijeme `pop(0)`):

```
class Graf:

    ...

    def velkost_komponentu3(self, v1):
        visited = set()
        queue = [v1]           # vlož do radu startovy vrchol
        while queue:          # kým je rad neprazdny
            v1 = queue.pop(0)  # vyber zo zaciatku radu
            if v1 not in visited:
                visited.add(v1)
                for v2 in self.pole[v1].sus:
                    if v2 not in visited:
                        queue.append(v2) # vlož na koniec radu
        return len(visited)
```

Takto zapísaný algoritmus bude robit' presne to isté ako nerekurzívny algoritmus `velkost_komponentu2()`. Rozdiel bude v tom, že sa v inom poradí navštívia všetky vrcholy grafu: najprv sa spracujú všetky najbližšie susediace vrcholy ku štartovému; potom sa spracujú všetky ich ešte nenavštívené susedné vrcholy atď. Aby sme lepšie videli toto poradie navštievovania vrcholov, môžeme pridať výpis poradových čísel:

```
class Graf:
```

```

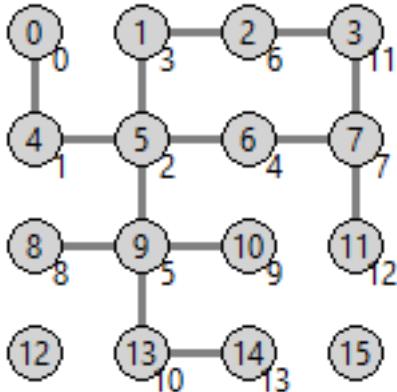
...
def velkost_komponentu3(self, v1):
    visited = set()
    queue = [v1]
    pocet = 0
    while queue:
        v1 = queue.pop(0)
        if v1 not in visited:
            visited.add(v1)
            x, y = self.pole[v1].xy
            self.c.create_text(x+10, y+10, text=pocet)
            pocet += 1
            for v2 in self.pole[v1].sus:
                if v2 not in visited:
                    queue.append(v2)
    return len(visited)

#testovanie

g = Graf(4)
print(g.velkost_komponentu3(0))

```

dostávame takýto obrázok:



Vďaka tejto veľmi užitočnej vlastnosti algoritmu ho môžeme jednoducho vylepšiť: pri každom spracovanom vrchole si budeme pamätať aj jeho momentálnu vzdialenosť - teda niečo ako úroveň "vnorenia" (vzdialosť od štartu). Štartový vrchol prehľadávania bude mať úroveň 0 (vzdialosť 0), všetci jeho bezprostrední susedia budú mať úroveň 1 (vzdialosť 1 od štartu) a každý ďalší vrchol bude mať úroveň o 1 väčšiu ako vrchol, od ktorého sa sem prišlo.

Program prepíšeme tak, že v rade si budeme pri každom vrchole pamätať aj jeho úroveň a pri spracovaní vrcholu túto úroveň vypíšeme (algoritmus sme už premenovali na dosirky()):

```

class Graf:

    ...

    def dosirky(self, v1):
        visited = set()
        queue = [(v1, 0)]
        while queue:
            v1, uroven = queue.pop(0)
            if v1 not in visited:

```

```

    visited.add(v1)
    x, y = self.pole[v1].xy
    self.c.create_text(x+12, y+12, text=uroven)
    self.c.update()          # spomalovanie vypisu
    self.c.after(100)

## 
## 
for v2 in self.pole[v1].sus:
    if v2 not in visited:
        queue.append((v2, uroven+1))

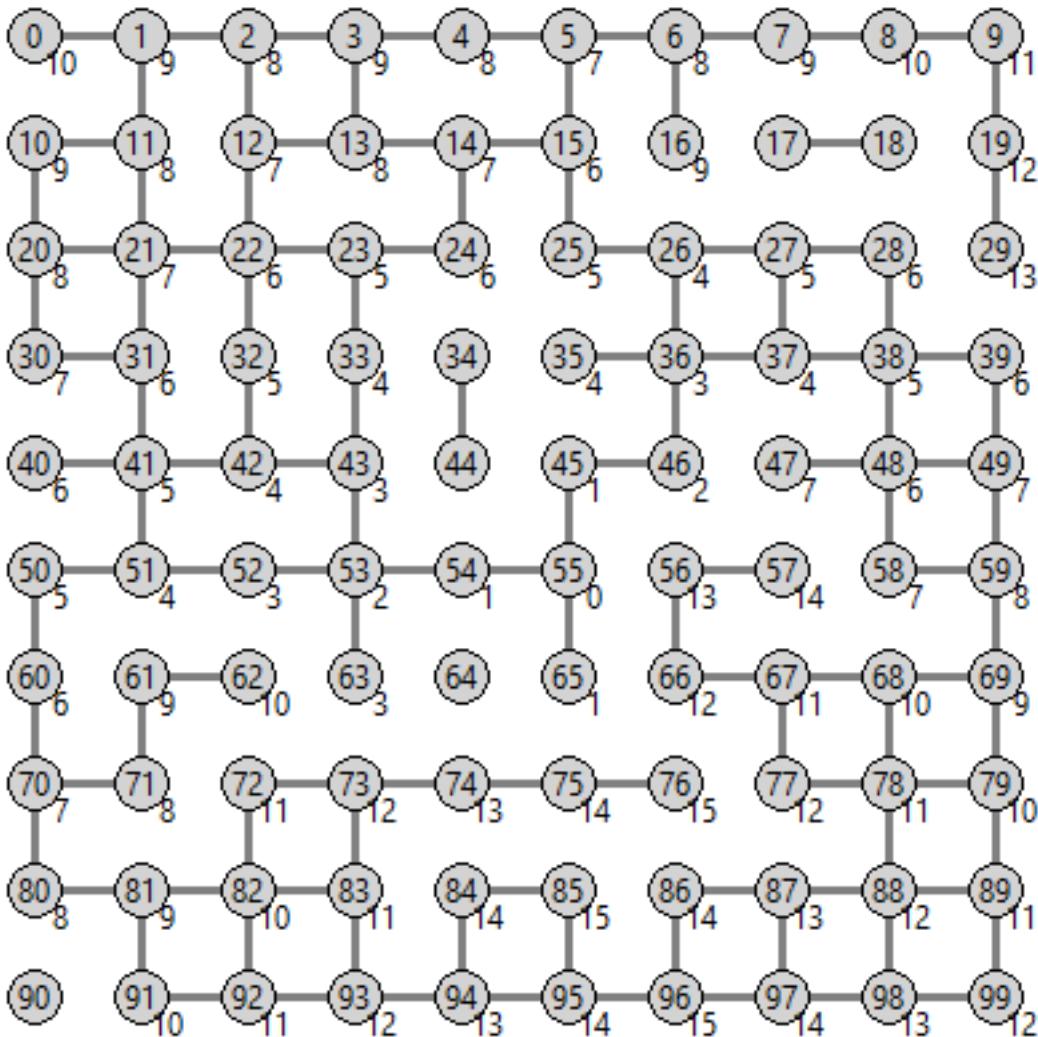
#testovanie

g = Graf(10)
g.dosirky(55)

```

Uvedomte si, že vrchol v ktorom začíname prehľadávanie má číslo úroveň 0, jeho susedia majú úroveň 1, ich susedia úroveň 2, ... Toto číslo vyjadruje vzdialenosť každého vrcholu od štartového.

Dostávame napr. takýto výpis:



V niektorých situáciách je vhodnejšie namiesto výpisu tohto čísla do grafickej plochy ho uložiť priamo do atribútu vo vrchole. (Namiesto `self.c.create_text...` zapíšeme napr. `self.pole[v1].uroven=uroven`)

Tento algoritmus môžeme použiť aj nazafarbovanie vrcholov grafu podľa toho, ako ďaleko sú od nejakého konkrétneho vrcholu:

```
class Graf:

    ...

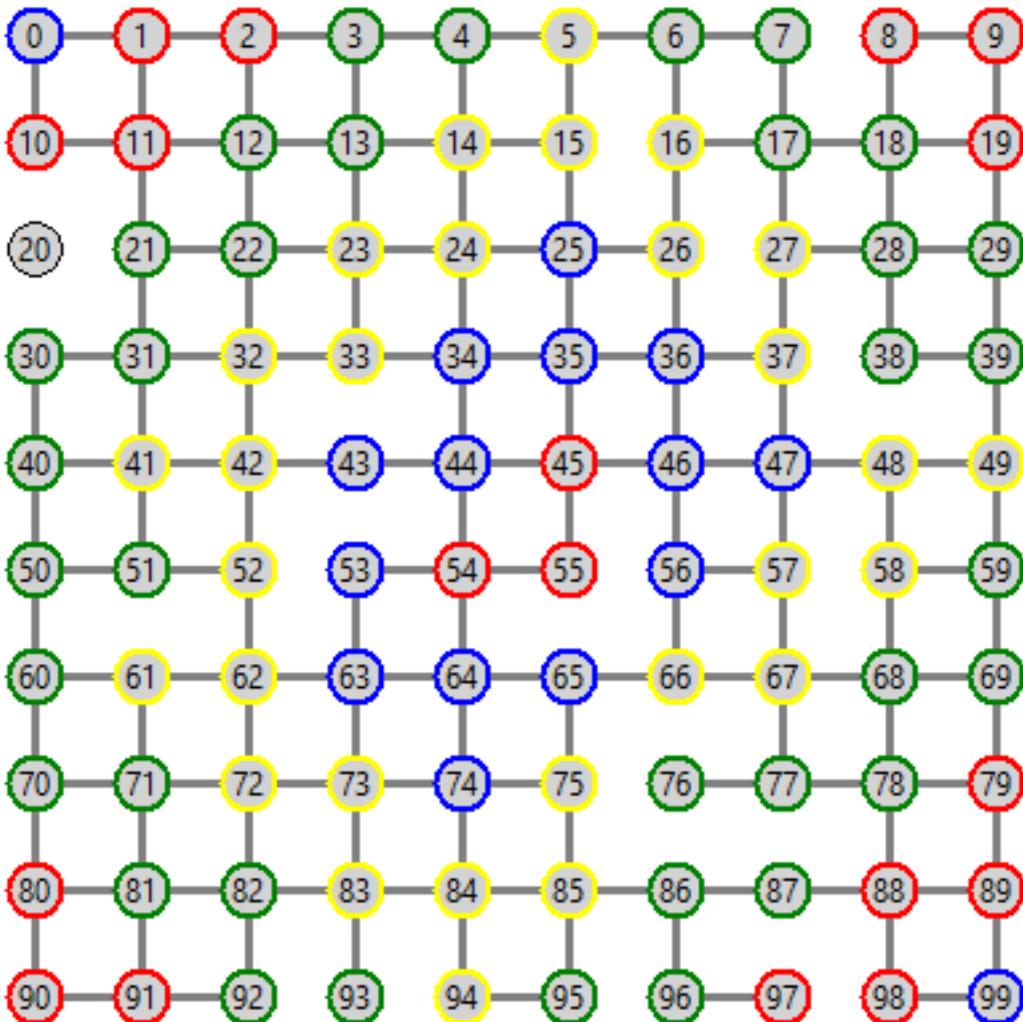
    def zafarbi(self, v1, farby):
        visited = set()
        queue = [(v1, 0)]
        while queue:
            v1, uroven = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                self.pole[v1].zafarbi(farby[uroven%len(farby)])
                for v2 in self.pole[v1].sus:
                    if v2 not in visited:
                        queue.append((v2, uroven+1))
```

V parametri `farby` posielame pole nejakých farieb, pričom samotný štartovací vrchol sa zafarbí farbou `farby[0]`, jeho susedia farbou `farby[1]`, ich susedia ďalšou farbou, atď. Ak je farieb v poli `farby` menej ako rôznych vzdialenosťí v grafe, táto funkcia začne používať farby opäť od začiatku poľa.

Algoritmus sme spustili pre 8 farieb, pričom úmyselne sme prvé dve farby dali rovnaké, aj ďalšie dve, atď. aby sa lepšie zobrazilo farbenie, teda štartový vrchol 55 aj jeho najbližší susedia sú červení, vrcholy vo vzdialosti 2 a 3 od štartu 55 sú modré, atď.

```
g = Graf(10)
g.zafarbi(55, ['red', 'red', 'blue', 'blue', 'yellow', 'yellow', 'green',
               'green'])
```

Dostávame takéto farbenie grafu:



## 9.5 Vzdialenosť a najkratšia cesta

Algoritmus do šírky môžeme využiť aj na zistovanie vzdialosti ľubovoľných dvoch vrcholov v grafe. Pod vzdialenosťou tu rozumieme dĺžku **najkratšej cesty** z jedného vrcholu do druhého. Ak takáto cesta neexistuje (vrcholy sú v rôznych komponentoch grafu), metóda môže vrátiť napr. hodnotu -1:

```
class Graf:

    ...

    def vzdialenosť(self, v1, v2):
        visited = set()
        queue = [(v1, 0)]
        while queue:
            v1, uroven = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                if v1 == v2:
                    return uroven
```

```

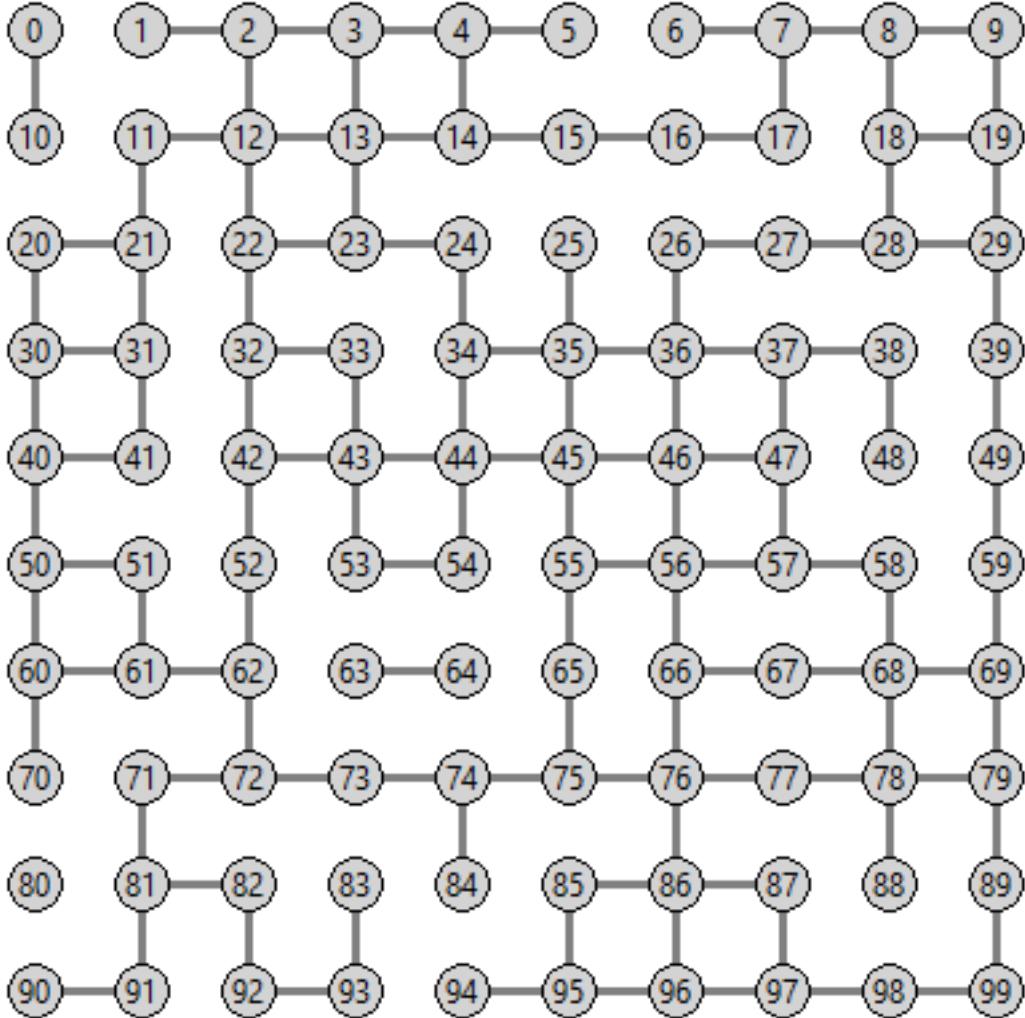
        for v in self.pole[v1].sus:
            if v not in visited:
                queue.append((v, uroven+1))
    return -1

#testovanie

g = Graf(10)
print(g.vzdialenos(11, 55))

```

Metóda `vzdialenos()` funguje na princípe algoritmu **dosirký** len pritom nič nezafarbuje. Napr. pre graf



môžeme otestovať:

```

>>> g.vzdialenos(55, 11)
8
>>> g.vzdialenos(11, 55)
8
>>> g.vzdialenos(1, 99)
17
>>> g.vzdialenos(0, 1)
-1

```

Rozšírime tento algoritmus nielen na zistenie vzdialenosť dvoch vrcholov, ale aj na zapamätanie najkratšej cesty. Použijeme takúto ideu: každý vrchol si namiesto úrovne bude pamätať svojho predchodcu, t.j. číslo vrcholu, z ktorého sme sem prišli. Na záver, keď dorazíme do cieľa, pomocou predchodcov vieme postupne skonštruovať výslednú postupnosť - cestu medzi dvoma vrcholmi. Ak pri zostavovaní cesty dostaneme ako predchodcu **-1** znamená to, že tento vrchol už nemá svojho predchodcu lebo je to štartový vrchol (odtiaľto sme naštartovali hľadanie cesty).

Algoritmus hľadania najkratšej cesty v grafe:

```
class Graf:  
    ...  
  
    def najkratsia_cesta(self, v1, v2):  
        visited = set()  
        queue = [(v1, -1)]  
        while queue:  
            v1, predchodca = queue.pop(0)  
            if v1 not in visited:  
                visited.add(v1)  
                self.pole[v1].pred = predchodca  
                if v1 == v2:  
                    vysl = []  
                    while v1 != -1:  
                        vysl.insert(0, v1)  
                        v1 = self.pole[v1].pred  
                    return vysl  
                for v in self.pole[v1].sus:  
                    if v not in visited:  
                        queue.append((v, v1))  
        return []
```

Pomocou tejto metódy vieme teda zistiť nielen vzdialosť 2 vrcholov ale aj konkrétnu postupnosť vrcholov, cez ktoré treba prejsť. Napr. pre graf z predchádzajúceho obrázka dostávame:

```
>>> g.najkratsia_cesta(55, 11)  
[55, 45, 35, 34, 24, 23, 13, 12, 11]  
>>> g.najkratsia_cesta(11, 55)  
[11, 12, 13, 23, 24, 34, 35, 45, 55]  
>>> g.najkratsia_cesta(1, 99)  
[1, 2, 3, 13, 23, 24, 34, 35, 36, 37, 47, 57, 58, 68, 69, 79, 89, 99]  
>>> g.najkratsia_cesta(1, 0)  
[]
```

Táto metóda vytvorí postupnosť vrcholov, ktoré tvoria najkratšiu cestu medzi dvoma vrcholmi. Túto cestu môžeme tiež vykresliť, napr. takto:

```
class Graf:  
    ...  
  
    def najkratsia_cesta(self, v1, v2):  
        ...  
  
    def kresli_cestu(self, v1, v2, farba):  
        cesta = self.najkratsia_cesta(v1, v2)  
        if cesta:  
            for v1 in range(len(cesta)-1):
```

```

        #self.pole[cesta[v1]].zafarbi(farba)
        self.zafarbi_hranu(cesta[v1], cesta[v1+1], farba)
        #self.pole[cesta[-1]].zafarbi(farba)
    else:
        print('cesta neexistuje')

```

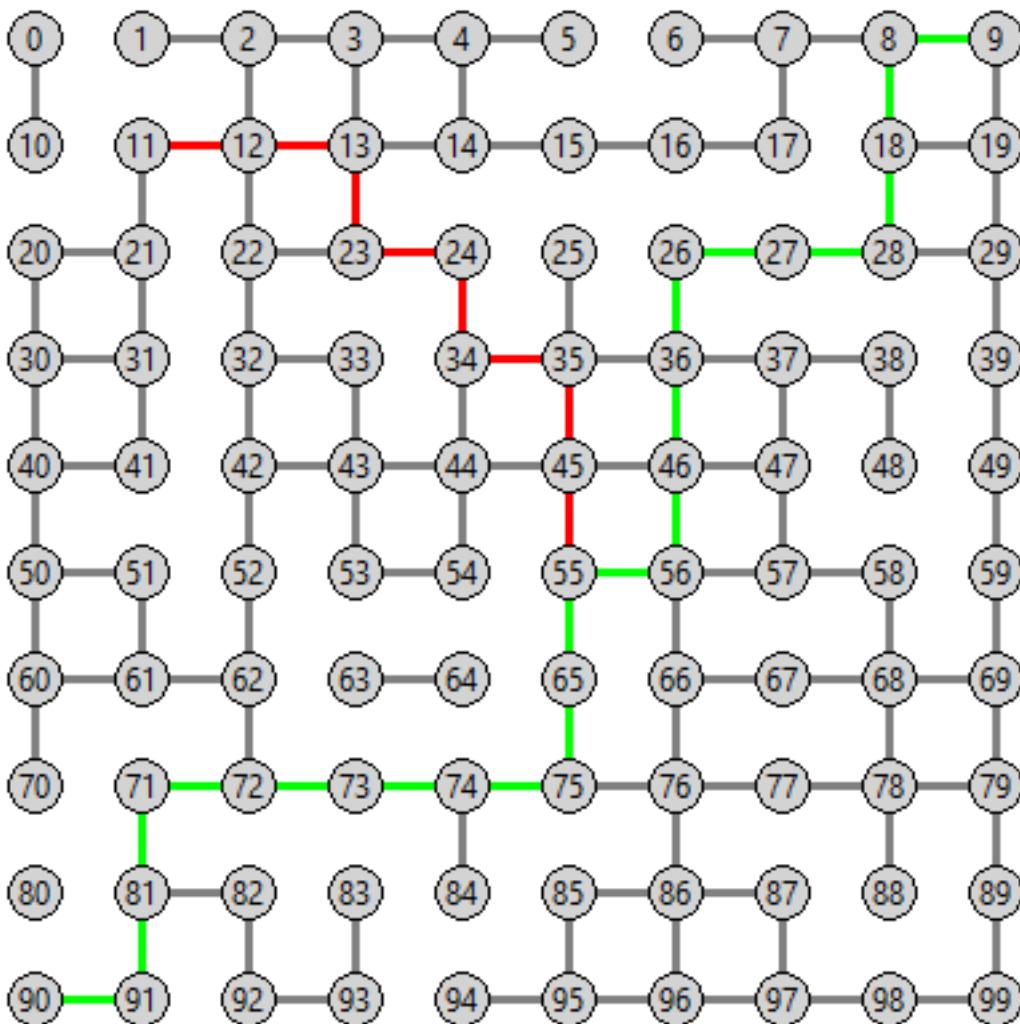
a v predchádzajúcom grafe zobrazíme najkratšie cesty:

```

>>> g.najkratsia_cesta(55, 11)
[55, 45, 35, 34, 24, 23, 13, 12, 11]
>>> g.kresli_cestu(55, 11, 'red')
>>> g.najkratsia_cesta(90, 9)
[90, 91, 81, 71, 72, 73, 74, 75, 65, 55, 56, 46, 36, 26, 27, 28, 18, 8, 9]
>>> g.kresli_cestu(90, 9, 'lime')
>>> g.kresli_cestu(10, 19, 'green')
cesta neexistuje

```

V jednom obrázku vidíme obe cesty:



Zhrňme použitie algoritmu do šírky:

- podobne ako do hľbky, vieme zistit' vrcholy, ktoré patria do jedného komponentu a teda aj zistit' počet kompo-

nentov

- vieme zistit' (zafarbit') vrcholy (alebo hrany), ktoré sú rovnako vzdialené od daného vrcholu
- vieme zistit' vzdialosť, resp. najkratšiu cestu medzi dvoma vrcholmi v grafe

---

## Backtracking

---

### 10.1 Generovanie štvoric čísel

Napíšme program, ktorý vypíše všetky 4-ciferné čísla zložené len z cifier 0 až  $n-1$  a pritom žiadna cifra sa neopakuje viackrát. Takéto štvorce budeme generovať štyrmi vnorenými for-cyklami:

```
def stvorice(n):
    pocet = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                for l in range(n):
                    if i!=j and i!=k and j!=k and i!=l and j!=l and k!=l:
                        print(i, j, k, l)
                        pocet += 1
    print('pocet =', pocet)

stvorice(4)
```

Program okrem všetkých vyhovujúcich štvoric vypíše aj ich počet (zrejme pre  $n=4$  ich bude **24**, t.j. **4!**).

Program môžeme aj trochu vylepšiť: vnútorné cykly nebudú zbytočne skúšať rôzne možnosti napr. vtedy, keď  $i == j$ :

```
def stvorice(n):
    pocet = 0
    for i in range(n):
        for j in range(n):
            if i!=j:
                for k in range(n):
                    if i!=k and j!=k:
                        for l in range(n):
                            if i!=l and j!=l and k!=l:
                                print(i, j, k, l)
                                pocet += 1
    print('pocet =', pocet)

stvorice(4)
```

Jednoduchým vylepšením môžeme generovať nie čísla (z nejakého intervalu), ale slová, ktoré sú zložené len zo zadaných písmen vo vstupnom slove:

```

def stvorice(slovo):
    pocet = 0
    for i in slovo:
        for j in slovo:
            if i!=j:
                for k in slovo:
                    if i!=k and j!=k:
                        for l in slovo:
                            if i!=l and j!=l and k!=l:
                                print(i+j+k+l)
                                pocet += 1
    print('pocet =', pocet)

stvorice('ahoj')

```

Program takto vypíše 24 rôznych slov.

Takýto spôsob riešenia však nie je najvhodnejší. Ak by sme napr. potrebovali nie štvorice, ale všeobecne n-tice čísl, alebo, ak by sme chceli komplikovanejšiu podmienku (cifry sa môžu raz opakovat') a pod., budeme musieť použiť nejaký iný spôsob riešenia.

## 10.2 Rekurzia

Budeme riešiť generovanie n-tíc čísel pomocou rekurzívnej funkcie. Začnime s úlohou, v ktorej generujeme všetky štvorice čísel z intervalu 0 až n-1, pričom čísla sa môžu aj opakovať'. Úlohu budeme riešiť pre ľubovoľné n. n-ticu postupne vytvárame v n-prvkovom poli pole a keď je kompletná, tak ju vypíšeme:

```

n = 4
pole = [0]*n
def generuj(i):          # parametrom je poradové číslo pridávaného prvku
    do pol'a
        for j in range(n):
            pole[i] = j
            if i == n-1:
                print(*pole)
            else:
                generuj(i+1)

generuj(0)

```

Toto riešenie generuje **všetky** n-tice čísel a nielen tie, v ktorých sú všetky cifry rôzne (teda 256 štvoríc). Pridáme teraz test na to, či je vygenerovaná n-tica naozaj správne riešenie: t.j. žiadne dva prvky nesmú byť rovnaké. Využijeme na to množinu, ktorú vytvoríme zo všetkých n prvkov poľa. Ak je aj táto množina n-prvková, žiaden prvak v poli sa neopakoval a teda máme vyhovujúce riešenie:

```

n = 4
pole = [0]*n
def generuj(i):
    for j in range(n):
        pole[i] = j
        if i == n-1:
            if len(set(pole))==n:
                print(*pole)
        else:
            generuj(i+1)

```

```
generuj(0)
```

Program teraz generuje 24 rôznych štvoríc.

## 10.3 Zapuzdrime

Zvykli sme si zapisovať komplexnejšie programy ako metódy triedy, pričom namiesto globálnych premenných pracujeme s atribútmi triedy. Zapíšme rekurzívne generovanie n-tíc ako metódu triedy `Uloha`. Začneme s jednoduchou verziou, v ktorej sa ešte nekontroluje, či sa nejaké cifry opakujú:

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pocet = [0]*n

    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

    def generuj(self, i):
        for j in range(self.n):
            self.pole[i] = j
            if i == self.n-1:
                print(*self.pole)
                self.pocet += 1
            else:
                self.generuj(i+1)

Uloha(4).ries()
```

Program vypíše 256 štvoríc.

Doplňme kontrolu, či vygenerovaná štvorica vyhovuje: do metódy **vyhovuje** zapíšme kontrolu kompletnej štvorce:

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0]*n

    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

    def vyzaduje(self):
        return len(set(self.pole)) == self.n

    def generuj(self, i):
        for j in range(self.n):
            self.pole[i] = j
            if i == self.n-1:
                if self.vyzaduje():
                    print(*self.pole)
                self.pocet += 1
```

```

        else:
            self.generuj(i+1)

Uloha(4).ries()

```

Takéto riešenie už dáva správny výsledok. Treba si ale uvedomiť, že algoritmus zbytočne generuje dosť veľa štvoríc (256), z ktorých cez výstupnú kontrolu (tesne pred výpisom pripravenej n-tice) prejde len niekoľko z nich (24).

Ukážeme riešenie, v ktorom sa vnárame do rekurzie len vtedy, keď doterajšia vygenerovaná časť riešenia vyhovuje podmienkam. Pripravili sme pomocnú funkciu `moze(i, j)`, ktorá otestuje, či momentálna hodnota `j` môže byť priradená na `i`-tu pozíciu výsledného poľa:

```

class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0]*n

    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

    def moze(self, i, j):      # či môžeme na i-tu pozíciu dať j
        return j not in self.pole[:i]

    def generuj(self, i):
        for j in range(self.n):
            if self.moze(i, j):
                self.pole[i] = j
                if i == self.n-1:
                    print(*self.pole)
                    self.pocet += 1
                else:
                    self.generuj(i+1)

Uloha(4).ries()

```

Takže testujeme nie až keď je vytvorená komplettná n-tica, ale vždy pred pridaním ďalšej hodnoty.

Pri riešení niektorých úloh môžeme na kontrolu pridávaného prvku do riešenia využiť ďalšie pomocné štruktúry. Nám by sa tu hodila napr. množina už použitých čísel, prípadne asociatívne pole (`dict`), v ktorom budeme evidovať počet výskytov každého čísla, napr.

```

class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0]*n

    def ries(self):
        self.pocet = 0
        self.vyskyty = {i:0 for i in range(self.n)}      # zatiaľ sú všetky
        ↪výskyty 0
        self.generuj(0)
        print('pocet =', self.pocet)

    def moze(self, j):
        return self.vyskyty[j] == 0

```

```

def generuj(self, i):
    for j in range(self.n):
        if self.moze(j):
            self.pole[i] = j
            self.vyskyty[j] += 1
            if i == self.n-1:
                print(*self.pole)
                self.pocet += 1
            else:
                self.generuj(i+1)
                self.vyskyty[j] -= 1

Uloha(4).ries()

```

Vidíme tu nový veľmi dôležitý detail:

- vždy, keď zaevidujeme nový prvok do pripravovaného poľa (`self.pole[i] = j`), zaznačíme ďalší výskyt tejto hodnoty aj do poľa `vyskyty`
- lenže, takto zaznačený výskyt treba niekedy aj zrušiť, lebo by sme príslušnú hodnotu už nikdy nedostali na žiadnej inej pozícii:
  - samotné zrušenie výskytu (`self.vyskyty[j] -= 1`) robíme na konci tela cyklu, lebo vtedy sa bude skúšať na túto pozíciu priradiť iná hodnota

Počítadlo výskytov by sme mohli využiť aj v úlohe, kde treba generovať  $n$ -tice, ale tak, že každá cifra sa môže raz opakovať. Zmeníme podmienku vo funkcii `moze`:

```

def moze(self, j):
    return self.vyskyty[j] <= 1

```

Pozrite si ešte nasledovný variant rekurzívnej funkcie `generuj`:

```

def generuj(self, i):
    if i == self.n:
        print(*self.pole)
        self.pocet += 1
    else:
        for j in range(self.n):
            if self.moze(j):
                self.pole[i] = j
                self.vyskyty[j] += 1
                self.generuj(i+1)
                self.vyskyty[j] -= 1

```

Takto zapísaná metóda funguje presne rovnako ako predchádzajúca verzia, len sa testovanie, či je kompletne riešenie, presunulo na začiatok (aj s malou zmenou podmienky). V praxi si väčšinou vyberáme ten zápis, ktorý sa nám v konkrétnej situácii viac hodí (nižšie uvidíme aj ďalšie varianty).

## 10.4 Backtracking

Takéto generovanie všetkých možných  $n$ -tíc, ktoré splňajú nejakú podmienku, je základom algoritmu, ktorému hovoríme **prehľadávanie s návratom**, resp. **backtracking**. Tento algoritmus teda:

- v každom kroku vyskúša všetky možnosti (napr. pomocou `for`-cyklu)
- pre každú možnosť preverí, či splňa podmienky (napr. metódou `moze(...)`)

- ak áno, zaeviduje si túto hodnotu (hovoríme tomu, že sa **zaznačí t'ah**) väčšinou v nejakých interných štruktúrach (pole, množina, asociatívne pole, ...)
- skontroluje, či riešenie nie je už kompletné a vtedy ho spracuje (napr. ho vypíše, alebo niekam zaznačí)
- ak riešenie ešte nie je kompletné, treba generovať ďalší prvok, čo zabezpečí rekurzívne volanie
- na konci cyklu, v ktorom sa postupne skúšajú všetky možnosti, treba ešte **vrátiť stav** pred zaevidovaním hodnoty (hovoríme tomu, že sa **odznačí t'ah**)

Schematicky to môžeme zapísat' takto:

```
def backtracking(param):  
    for i in vsetky_moznosti:  
        if moze(i):  
            zaznac_tah(i)  
            if hotovo:  
                vypis_riesenie()  
            else:  
                backtracking(param1)  
                odznac_tah(i)
```

alebo druhá schéma:

```
def backtracking(param):  
    if hotovo:  
        vypis_riesenie()  
    else:  
        for i in vsetky_moznosti:  
            if moze(i):  
                zaznac_tah(i)  
                backtracking(param1)  
                odznac_tah(i)
```

Zapamätajte si, že je veľmi dôležité vrátiť ešte pred koncom cyklu všetko, čo sme zaevidovali na začiatku cyklu. Uvedomte si, že bez tohto vrátenia pôvodného stavu sa tento algoritmus stáva obyčajným prehľadávaním do hĺbky, ktorý sa len rekurzívne vnára hlbšie a hlbšie ...

Pomocou **backtrackingu** môžeme riešiť úlohy typu:

- matematické hlavolamy (8 dám na šachovnici, domček jedným t'ahom, kôň na šachovnici, sudoku, ...)
- rôzne problémy na grafoch (nájst' cestu s najmenším ohodnotením z A do B, vyhodiť max. počet hrán, aby platila nejaká podmienka)

Vo všeobecnosti je backtracking **veľmi neefektívny algoritmus** (tzv. hrubá sila), pomocou ktorého sa dá vyriešiť veľké množstvo úloh (postupne vyskúšam všetky možnosti). V praxi sa mnoho problémov dá vyriešiť oveľa efektívnejšie. Pre backtracking väčšinou platí, že jeho **zložitosť** je exponenciálna, t.j. čím je úloha väčšieho rozsahu, tým rýchlejšie rastie čas na jeho vyriešenie. Pri algoritmoch triedenia sme videli obrovský rozdiel vo výkone bublinkového a rýchleho (quick-sort) triedenia. Pritom bublinkové triedenie má zložitosť rádovo  $n^{**}2$  a pre väčšie pole je už neprijateľne pomalé. Čo potom algoritmy, ktorých zložitosť je rádovo  $2^{**}n$ .

## 10.5 Dámy na šachovnici

Budeme riešiť takýto hlavolam: na šachovnicu s 8x8 treba umiestniť 8 dám tak, aby sa navzájom neohrozovali (vodorovne, zvislo ani uhlopriečne). Zrejme v každom riadku a tiež v každom stĺpci musí byť práve jedna dáma. Dámy očisľujeme číslami od 0 do 7 tak, že  $i$ -ta dáma sa bude nachádzať v  $i$ -tom riadku. Potom každé rozloženie dám na šachovnici môžeme reprezentovať osmicou čísel (v poli `riesenie`):  $i$ -te číslo potom určuje číslo stĺpca  $i$ -tej dámy.

Na riešenie úlohy využime schému backtrackingu:

```
def hladaj(self, i):          # i-ta dáma v i.riadku -> hľadá pre ňu vhodný stĺpec
    for j in range(self.n):
        if self.moze(i, j):
            # zaznač položenie dámy
            self riesenie[i] = j
            ...
            if i == self.n-1:           # ak je už hotovo
                print(*self riesenie)
                self.pocet += 1
            else:
                self.hladaj(i+1)
                # odznač položenie dámy
                self riesenie[i] = None
            ...
        
```

Na rýchle otestovanie, či je nejaká pozícia ohrozená ostatnými dámami použijeme 3 pomocné množiny:

- stlpec - tu si pamätáme, v ktorých stĺpcach je už nejaká dáma
- u1 - pamätáme si, v ktorých uhlopriečkach (sprava doľava) je už nejaká dáma
- u2 - pamätáme si, v ktorých uhlopriečkach (zľava doprava) je už nejaká dáma

Využijeme vlastnosť políčok na šachovnici, vďaka ktorej vieme zistit, či sa dve políčka nachádzajú na tej istej uhlopriečke:

- pre uhlopriečky sprava (hore) dol'ava (dole) platí, že súčet čísla riadku a čísla stĺpca je rovnaký (napr. (3,1),(2,2),(0,4),... ležia na jednej uhlopriečke)
- pre uhlopriečky zl'ava (hore) doprava (dole) platí, že rozdiel čísla riadku a čísla stĺpca je rovnaký (napr. (3,1),(4,2),(6,4),... ležia na jednej uhlopriečke)

Všetky tieto tri pomocné množiny musia byť na začiatku prázne:

```
self.stlpec = set()
self.u1 = set()
self.u2 = set()
```

Zaznačiť ľah potom znamená:

- zapamätať si ho v poli riesenie
- zaznačiť si obsadený príslušný stĺpec v množine stlpec
- zaznačiť si obsadenú príslušnú uhlopriečku v množine u1
- zaznačiť si obsadenú príslušnú 2. uhlopriečku v množine u2

```
# zaznačiť i-tu dámdu v i riadku a j stĺpco
self riesenie[i] = j
self.stlpec.add(j)
self.u1.add(i+j)
self.u2.add(i-j)
```

Kontrola jednej konkrétnej dámy (metóda moze()) potom len skontroluje tieto tri množiny:

```
def moze(self, i, j):          # či môže položiť dámdu na pozícii (i,j)
    return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2
```

Celý program:

```

class Damy:
    def __init__(self, n):
        self.n = n
        self.riesenie = [None]*n

    def ries(self):
        self.stlpec = set()
        self.u1 = set()
        self.u2 = set()
        self.pocet = 0
        self.hladaj(0)
        print('počet riešení:', self.pocet)

    def moze(self, i, j):          # či môže položiť dámou na pozíciu (i,j)
        return j not in self.stlpec and i+j not in self.u1 and i-j not in_
        ↵self.u2

    def hladaj(self, i):          # i-ta dámama v i.riadku -> hľadá pre ňu_
        ↵vhodný stĺpec
        for j in range(self.n):
            if self.moze(i, j):
                # zaznač položenie dámama
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i+j)
                self.u2.add(i-j)
                if i == self.n-1:
                    print(*self.riesenie)
                    self.pocet += 1
                else:
                    self.hladaj(i+1)
                # odznač položenie dámama
                self.riesenie[i] = None
                self.stlpec.remove(j)
                self.u1.remove(i+j)
                self.u2.remove(i-j)

Damy(8).ries()

```

Rekurzívnu metódu `hladaj()` by sme mohli prepísat aj podľa druhej schémy backtrackingu:

```

def hladaj(self, i):          # i-ta dámama v i.riadku -> hľadá pre ňu stĺpec
    if i == self.n:
        print(*self.riesenie)
        self.pocet += 1
    else:
        for j in range(self.n):
            if self.moze(i, j):
                # zaznač položenie dámama
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i+j)
                self.u2.add(i-j)
                self.hladaj(i+1)
                # odznač položenie dámama
                self.riesenie[i] = None
                self.stlpec.remove(j)

```

```
self.ul.remove(i+j)
self.u2.remove(i-j)
```

Výsledný program aj teraz generuje rovnaký výsledok.

Niekedy potrebujeme organizovať backtrackingovú funkciu tak, že priamo v parametroch funkcie dostane konkrétnu hodnotu t'ahu - vtedy si ho najprv zaznačí, potom vygeneruje ďalší t'ah (rekurzívne sa zavolá) a na koniec odznačí t'ah.

Úlohu s dámami môžeme zapísat' aj takto:

```
class Damy:
    def __init__(self, n):
        self.n = n
        self.riesenie = [None]*n

    def ries(self):
        self.stlpec = set()
        self.ul = set()
        self.u2 = set()
        self.pocet = 0
        for j in range(self.n):
            self.dalsi(0,j)
        print('pocet rieseni:', self.pocet)

    def moze(self, i, j):          # či môže položiť dámu na pozícii (i,j)
        return j not in self.stlpec and i+j not in self.ul and i-j not in self.u2

    def dalsi(self, i, j):          # polož dámu na (i,j)
        # zaznač položenie dámy
        self.riesenie[i] = j
        self.stlpec.add(j)
        self.ul.add(i+j)
        self.u2.add(i-j)

        if i == self.n-1:
            print(*self.riesenie)
            self.pocet += 1
        else:
            for k in range(self.n):
                if self.moze(i+1, k):
                    self.dalsi(i+1, k)

        # odznač položenie dámy
        self.riesenie[i] = None
        self.stlpec.remove(j)
        self.ul.remove(i+j)
        self.u2.remove(i-j)

Damy(8).ries()
```

Namiesto metódy `hlada(j)` sme tu zapísali metódu `dalsi()`.

Ďalšia verzia tohto programu ukazuje, ako môžeme vykresliť nájdené riešenie (metóda `vypis()`). Tiež sme na začiatok backtrackingu pridali jeden test, vd'aka ktorému program korektne skončí po nájdení a vypísaní jedného riešenia (ak nám jedno stačí):

```

class Damy:
    def __init__(self, n):
        self.n = n
        self.riesenie = [None]*n

    def ries(self):
        self.stlpec = set()
        self.u1 = set()
        self.u2 = set()
        self.pocet = 0
        self.hladaj(0)
        # print('počet riesení:', self.pocet)
        if self.pocet == 0:
            print('ziadne riesenie')

    def moze(self, i, j):          # či môže položiť dámú na pozíciu (i,j)
        return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2

    def vypis(self):
        for k in range(self.n):
            r = ['.']*self.n
            r[self.riesenie[k]] = 'o'
            print(' '.join(r))
        print('='*2*self.n)

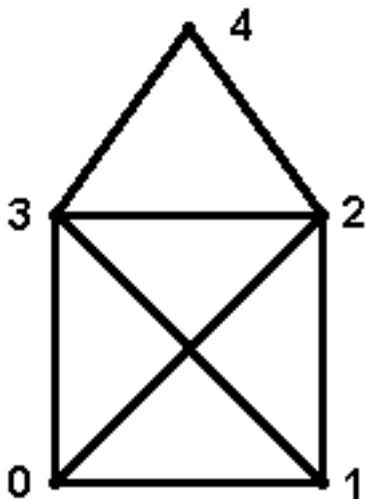
    def hladaj(self, i):
        if self.pocet:                # už máme 1 riešenie, nemusíme d'alej
            pokračovať
        return
        for j in range(self.n):
            if self.moze(i, j):
                # zaznač položenie dámy
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i+j)
                self.u2.add(i-j)
                if i == self.n-1:
                    self.vypis()
                    self.pocet += 1
                else:
                    self.hladaj(i+1)
                    self.riesenie[i] = None
                    self.stlpec.remove(j)
                    self.u1.remove(i+j)
                    self.u2.remove(i-j)

Damy(8).ries()

```

## 10.6 Domček jedným ľahom

Budeme riešiť takúto úlohu: potrebujeme zistiť, kol'kými rôznymi spôsobmi sa dá nakresliť takýto domček jedným ľahom. Pri kreslení môžeme po každej čiare prejsť len raz.



Obrázok domčeka je vlastne neorientovaný graf s 5 vrcholmi. Každé nájdené riešenie vypíšeme v tvare postupnosti vrcholov, cez ktoré sa prechádza pri kreslení domčeka. Keďže hrán je v grafe 8, tak táto postupnosť bude obsahovať presne 9 vrcholov: jeden štartový a 8 nasledovných vrcholov.

Graf budeme reprezentovať čo najjednoduchšie, napr. pomocou pola množín susedností. Pre každý z vrcholov si treba pamätať množinu jeho susedov:

```
graf = [{1,2,3},           # susedia vrcholu 0
        {0,2,3},           # susedia vrcholu 1
        {0,1,3,4},         # susedia vrcholu 2
        {0,1,2,4},         # susedia vrcholu 3
        {2,3},             # susedia vrcholu 4
    ]
```

Kompletný program postupne vytvára pole riesenie s postupnosťou prechádzaných vrcholov grafu:

```
class Domcek:
    def __init__(self):
        self.g = [{1,2,3}, {0,2,3}, {0,1,3,4}, {0,1,2,4}, {2,3}]

    def ries(self):
        self.pocet = 0
        for i in range(len(self.g)):
            self.riesenie = [i]
            self.hladaj()
            print('pocet rieseni:', self.pocet)

    def hladaj(self):
        v1 = self.riesenie[-1]
        for v2 in self.g[v1]:
            # zaznač t'ah
            self.riesenie.append(v2)
            self.g[v1].remove(v2)
            self.g[v2].remove(v1)
            if len(self.riesenie) == 9:
                print(*self.riesenie)
                self.pocet += 1
            else:
                self.hladaj()
            # odznač t'ah
```

```

        self.riesenie.pop()
        self.g[v1].add(v2)
        self.g[v2].add(v1)

Domcek().ries()
    
```

- prvý vrchol riešenia (štartový) sa do poľa priradí už ešte pred zavolením backtrackingu v štartovej metóde `ries`: keďže môžeme začínať z ľubovoľného vrcholu, v tejto metóde štartujeme backtracking v cykle postupne so všetkými možnými začiatocnými vrcholmi
- v backtrackingovej metóde `hlada(j)`:
  - `v1` obsahuje zatiaľ posledný vrchol riešenia, na ktorý bude teraz nadvázovať nasledovný vrchol `v2`
  - **zaznačenie t'ahu** uskutočníme tak, že práve prejdenú hranu dočasne z grafu odstránime (graf je neorientovaný preto treba odstraňovať oba smery tejto hrany)
  - riešenie je kompletné vtedy, keď obsahuje presne 9 vrcholov
  - **odznačenie t'ahu** (teda návrat do pôvodného stavu pre zaznačením) urobíme vrátením dočasne odstránenej hrany

## 10.7 Sudoku

Pomocou prehľadávania do hĺbky vyriešime veľmi populárny hlavolam Sudoku. Hracia plocha sa skladá z  $9 \times 9$  políčok. Na začiatku sú do niektorých políčok zapísané čísla z intervalu  $<1,9>$ . Úlohou je zapísat aj do zvyšných políčok čísla 1 až 9 tak, aby v každom riadku a tiež v každom stĺpci bolo každé z čísel práve raz. Okrem toho je celá plocha rozdelená na 9 menších štvorcov veľkosti  $3 \times 3$  - aj v každom z týchto štvorcov musí byť každé z čísel práve raz.

Napr. súbor s konkrétnym zadaním '`sudoku.txt`' môže vyzerat napr. takto:

```

0 6 0 9 0 0 0 7 0
0 4 0 8 0 0 0 0 0
0 0 0 0 5 0 3 0 0
0 0 0 0 0 0 0 0 0
0 9 0 0 4 0 0 6 0
8 0 5 0 6 0 0 0 0
7 0 8 3 0 0 0 0 4
3 0 0 0 2 1 0 0 8
0 0 0 0 0 0 0 2
    
```

- hodnota 0 označuje, že príslušné políčko je zatiaľ prázdné

Samotná “backtrackingová” rekurzívna procedúra najprv nájde prvé zatiaľ voľné políčko (je tam hodnota 0). Potom sa sem pokúsi zapísat všetky možnosti čísel 1 až 9. Zakaždým otestuje, či zapisované číslo sa v príslušnom riadku, stĺpci a tiež v malom  $3 \times 3$  štvorci ešte nenachádza:

```

def backtracking(self):
    i, j = najdi_volne()          # nájdi volné políčko
    if nenasiel:                  # už nie je žiadne ďalšie volné políčko
        vypis_riesenie()
    else:
        for cislo in range(1, 10):      # pre všetky možnosti čísel od 1
            do 9
                if moze(i, j, cislo):
                    zaznac_tah()
    
```

```
backtracking()
odznac_tah()
```

Aby sa nám čo najjednoduchšie testovalo, či sa nejaké číslo ešte nenachádza v príslušnom riadku, stĺpci a 3x3 štvorci, pre každý riadok, stĺpec aj štvorec 3x3 zavedieme množinovú premennú, t.j. 9-prvkové množinové polia pre riadky a stĺpce a dvojrozmerné 3x3 pole množín pre menšie štvorce:

```
self.vstlpci = [set(), set(), set(), set(), set(), set(), set(), set(), set()]
self.vriadku = [set(), set(), set(), set(), set(), set(), set(), set(), set()]
self.v3x3    = [[set(), set(), set()], [set(), set(), set()], [set(), set(), set()]]
```

Ak budeme teraz potrebovať zistiť, či sa v  $j$ -tom stĺpci nachádza dané cislo zapíšeme:

```
if cislo in self.vstlpci[j]: ...
```

podobne pre  $i$ -ty riadok:

```
if cislo in self.vriadku[j]: ...
```

alebo v príslušnom malom štvorci 3x3 (chceme zistiť štvorec, ktorý obsahuje políčko  $(i, j)$  teda  $i$  aj  $j$  delíme 3):

```
if cislo in self.v3x3[i//3][j//3]: ...
```

Podobným spôsobom budeme do týchto množín vkladať nové čísla (pri zaznačovaní ťahu), resp. ich z množín odberať (pri odznačovaní).

Všimnite si, že tieto množiny sa musia zaplniť počiatočnými hodnotami už pri čítaní vstupného súboru. Kompletný program:

```
class Sudoku:
    def __init__(self, subor):
        with open(subor) as t:
            self.pole = [list(map(int, r.split())) for r in t]
        self.vstlpci = [set() for i in range(9)]
        self.vriadku = [set() for i in range(9)]
        self.v3x3    = [[set(), set(), set()] for i in range(3)]
        for i in range(9):
            for j in range(9):
                cislo = self.pole[i][j]
                if cislo:
                    self.vstlpci[j].add(cislo)
                    self.vriadku[i].add(cislo)
                    self.v3x3[i//3][j//3].add(cislo)

    def vypis(self):
        if self.pocet == 0:                      # vypíše len prvé riešenie
            for riadok in self.pole:
                print(*riadok)
            print('='*17)

    def moze(self, i, j, cislo):
        return (cislo not in self.vstlpci[j] and
                cislo not in self.vriadku[i] and
                cislo not in self.v3x3[i//3][j//3])

    def ries(self):
        self.pocet = 0
        self.backtracking()
```

```
print('počet riešení:', self.počet)

def backtracking(self):
    i, j = 0, 0
    while i<9 and self.pole[i][j]:
        j += 1
        if j == 9:
            i += 1
            j = 0
    if i == 9:                      # už nie je žiadne ďalšie vol'né
→poličko
        self.vypis()
        self.počet += 1
    else:
        for cislo in range(1, 10):
            if self.može(i, j, cislo):
                self.pole[i][j] = cislo
                self.vstlpci[j].add(cislo)
                self.vriadku[i].add(cislo)
                self.v3x3[i//3][j//3].add(cislo)
                self.backtracking()
                self.pole[i][j] = 0
                self.vstlpci[j].remove(cislo)
                self.vriadku[i].remove(cislo)
                self.v3x3[i//3][j//3].remove(cislo)

Sudoku('sudoku.txt').ries()
```

---

## Backtracking na grafoch

---

Využijeme mierne upravenú triedu Graf z 9. prednášky:

```

import tkinter, random

R, HRUBKA = 15, 3
SIRKA, VYSKA = 600, 600

class Graf:

    class Vrchol:
        def __init__(self, x, y):
            self.sus = {}
            self.xy = x, y

        def pridaj_hranu(self, v2, vaha):
            self.sus[v2] = vaha

        def kresli(self, c, farba='gray'):
            self.c = c
            x, y = self.xy
            self.id = self.c.create_oval(x-R, y-R, x+R, y+R, fill=farba, ↴outline=farba)

        def zafarbi(self, farba):
            self.c.itemconfig(self.id, fill=farba, outline=farba)

    #-----

        def __init__(self, n):
            self.pole = []
            self.c = tkinter.Canvas(bg='white', width=SIRKA, height=VYSKA)
            self.c.pack()
            d = (min(VYSKA, SIRKA)-60) // (n-1)
            for i in range(n):
                for j in range(n):
                    v1 = self.pridaj_vrchol(d*j+30+random.randint(-10, 10),
                                            d*i+30+random.randint(-10, 10))
                    if random.randrange(3) and j>0:
                        x1, y1 = self.pole[v1].xy
                        x2, y2 = self.pole[v1-1].xy
                        vaha = round(((x1-x2)**2+(y1-y2)**2)**.5)
                        self.pridaj_hranu(v1, v1-1, vaha)
                    if random.randrange(3) and i>0:

```

```

        x1, y1 = self.pole[v1].xy
        x2, y2 = self.pole[v1-n].xy
        vaha = round(((x1-x2)**2+(y1-y2)**2)**.5)
        self.pridaj_hranu(v1, v1-n, vaha)

    self.kresli()
    self.c.bind('<Button-1>', self.udalost_klik)
    self.klik = []

    def kresli(self):
        self.id = {}
        for i in range(len(self.pole)):
            for j in self.pole[i].sus:
                if i < j:
                    self.kresli_hranu(i, j)
        for vrchol in self.pole:
            vrchol.kresli(self.c)

    def pridaj_vrchol(self, x, y):
        self.pole.append(self.Vrchol(x, y))
        return len(self.pole)-1

    def pridaj_hranu(self, i, j, vaha):
        self.pole[i].pridaj_hranu(j, vaha)
        self.pole[j].pridaj_hranu(i, vaha)

    def je_hrana(self, i, j):
        return j in self.pole[i].sus

    def kresli_hranu(self, i, j, farba='gray'):
        if i > j:
            i, j = j, i
        self.id[i, j] = self.c.create_line(self.pole[i].xy, self.pole[j].xy,
                                           fill=farba, width=HRUBKA)

    def zafarbi_hranu(self, i, j, farba):
        if i > j:
            i, j = j, i
        self.c.itemconfig(self.id[i, j], fill=farba)

    def udalost_klik(self, event):
        for i in range(len(self.pole)):
            x, y = self.pole[i].xy
            if (x-event.x)**2+(y-event.y)**2 < R**2:
                self.pole[i].zafarbi('blue')
                self.klik.append(i)
        return

graf = Graf(7)

```

Upravili sme:

- graf je ohodnotený, t.j. pre každú hranu si uchovávame jej váhu (zvolili sme vzdialenosť vrcholov v rovine)
  - namiesto pol'a množín susedností graf reprezentujeme ako pole vrcholov, v ktorom si každý vrchol pamätá susedov v asociatívnom poli (dict), s hodnotami váha hrany
- generovanie pozícíí vrcholov: náhodne sme ich trochu poposúvali
- pridali sme metódu `udalost_klik()`, ktorá nielen zafarbuje kliknuté vrcholy, ale si ich aj pamätá v poli

```
self.klik
```

## 11.1 Základná schéma backtrackingu

Pripomeňme si, no čo vieme použiť prehľadávanie do hĺbky a do šírky:

- algoritmus **dohľbky** využijeme hlavne na zistenie príslušnosti vrcholov do komponentov
- algoritmus **dosirky** slúži na nájdenie vzdialenosť (najkratšej cesty) dvoch vrcholov
  - takáto dĺžka cesty neberie do úvahy váhu na hranách, ale len počet vrcholov na ceste

Prehľadávanie s návratom

- pracuje na princípe prechádzania (generovania) všetkých možných ciest v grafe, ktoré vychádzajú z nejakého vrcholu
- pri tomto generovaní, môžeme zisťovať rôzne parametre týchto vygenerovaných ciest, napr. súčet váh na hranach cesty, alebo, či cesta prešla cez nejaký konkrétny vrchol a pod.
- uvedomte si, že takéto prehľadávanie (tzv. **hrubá sila, brute force**) bude pre väčšie grafy časovo veľmi náročné, keďže vygenerovaných ciest môže byť obrovské množstvo
- často závisí od detailov v nejakom teste, ako rýchlo takéto prehľadávanie nájde riešenie (hovoríme tomu, že využijeme nejakú **heuristiku**)
- vo vyšších ročníkoch sa zoznámite s algoritmami, ktoré vedia riešiť niektoré z našich úloh aj bez hrubej sily

Základnú schému backtrackingu, môžeme pre graf upraviť napr. takto:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.backtracking(v1, v2)

    def backtracking(self, v1, v2):
        self.visited.add(v1)
        if v1==v2:
            ... # spracuj riešenie
        else:
            for v in self.pole[v1].sus:
                if v not in self.visited:
                    # vizualizuj prechádzanú hranu
                    self.backtracking(v, v2)
                    # zruš vizualizovanie prechádzanej hrany
                self.visited.remove(v1)
```

- podobne ako pri prechádzaní grafu do hĺbky, aj tu musíme mať nejakú štruktúru `visited`, aby sme neprechádzali cez nejaký vrchol viackrát
- parameter `v1` označuje práve prechádzaný vrchol, `v2` cieľový vrchol, do ktorého sa snažíme dostať
- tento algoritmus naozaj nájde všetky možné cesty z vrcholu `v1` do `v2`, pričom pri spracovaní riešenia môžeme tieto nájdené cesty nejako pretriediť a vybrať si len tú najvhodnejšiu

Vytvorme takýto program:

- po kliknutí na dva rôzne vrcholy sa naštartuje backtracking

- samotný algoritmus bude každú prechádzanú hranu prefarbovať a na malú chvíľu pritom spomalí výpočet
- konkrétnie nasledovný program bude počítať počet rôznych ciest a tento počet na záver vypíše

Upravíme aj udalost\_klik():

```
class Graf:
    ...

    def udalost_klik(self, event):
        for i in range(len(self.pole)):
            x, y = self.pole[i].xy
            if (x-event.x)**2+(y-event.y)**2 < R**2:
                self.pole[i].zafarbi('blue')
                self.klik.append(i)
                self.c.update()
            if len(self.klik) == 2:           # <== po druhom kliknutí
                self.start(*self.klik)
                return

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.backtracking(v1, v2)
        # ked' skončí backtracking
        print('pocet riesení =', self.pocet)

    def backtracking(self, v1, v2):
        if v1==v2:
            self.pocet += 1      # spracuj riešenie
            # print('nasiel som cestu')
        else:
            self.visited.add(v1)
            for v in self.pole[v1].sus:
                if v not in self.visited:
                    # vizualizuj prechádzanú hranu
                    self.zafarbi_hranu(v1, v, 'red')
                    self.c.update()
                    self.c.after(100)
                    self.backtracking(v, v2)
                    # zruš vizualizovanie prechádzanej hrany
                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)

graf = Graf(7)
```

- volanie `self.c.after(100)` označuje, že tu program zastane na 100 ms, tento riadok môžeme vyhodiť, aby bolo hľadanie ciest rýchlejšie

V niektorých situáciách sa nám môže hodíť, keď po nájdení prvého riešenia, zastavíme beh backtrackingu ale tak, aby táto cesta ostala nakreslená:

```
class Graf:
    ...

    def backtracking(self, v1, v2):
        if self.pocet > 0: return      # aby sa d'alej nevnáral

        if v1==v2:
            self.pocet += 1      # spracuj riešenie
```

```

        print('nasiel som cestu')
    else:
        self.visited.add(v1)
        for v in self.pole[v1].sus:
            if v not in self.visited:
                # vizualizuj prechádzanú hranu
                self.zafarbi_hranu(v1, v, 'red')
                self.c.update()
                self.c.after(100)
                self.backtracking(v, v2)
                if self.pocet > 0: return          # aby sa nezrušilo
        ↪zafarbenie hrán
                # zruš vizualizovanie prechádzanej hrany
                self.zafarbi_hranu(v1, v, 'gray')
        self.visited.remove(v1)

```

Častejšie budeme ale prechádzať všetky vygenerované cesty a hľadať z nich jednu s nejakou vlastnosťou, napr. cestu s najmenším ohodnením: ak je na hranách napr. cena, tak hľadáme najlacnejšiu cestu. Zrejme je rozdiel medzi najkratšou cestou (s najmenej prechádzanými vrcholmi) a najlacnejšou cestou.

## 11.2 Hodnota cesty

Pri generovaní cesty budeme evidovať aj jej hodnotu (ako súčet ohodnení hrán). Najprv to ukážeme pomocou atribútu hodnota v triede Graf, ktorý sa pre prechode hranou zvyšuje a pri návrate znižuje:

```

class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.hodnota = 0
        self.backtracking(v1, v2)
        print('pocet =', self.pocet)

    def backtracking(self, v1, v2):
        if v1==v2:
            self.pocet += 1
            print('cesta s hodnotou', self.hodnota)
        else:
            self.visited.add(v1)
            for v in self.pole[v1].sus:
                if v not in self.visited:
                    self.hodnota += self.pole[v1].sus[v]
                    self.zafarbi_hranu(v1, v, 'red')
                    self.c.update()
                    #self.c.after(100)

                    self.backtracking(v, v2)

                    self.hodnota -= self.pole[v1].sus[v]
                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)

```

Hodnotu cesty môžeme vytvárať aj v parametri funkcie backtracking(). Na začiatku je hodnota cesty 0, vždy keď algoritmus prejde po nejakej hrane, tak sa táto hodnota zvyšuje. Keď cesta dorazí do cieľa (do vrcholu v2), zistí sa,

či táto nová vygenerovaná cesta je lepšia (napr. menšia ako doterajšie minimum, alebo väčšia ako doterajšie maximu) ako doteraz uchovávané hodnoty v premenných `self.min` a `self.max`. Všimnite si, že takto uchovávanú hodnotu v parametri nemusíme pri návrate z rekurzie znižovať, ako sme to robili v predchádzajúcej verzii:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.min = self.max = None
        self.backtracking(v1, v2, 0)
        print('pocet =', self.pocet)
        print('najkratsia =', self.min)
        print('najdlhsia =', self.max)

    def backtracking(self, v1, v2, hodnota):
        if v1==v2:
            # print('cesta s hodnotou', hodnota)
            if self.min is None or self.min > hodnota:
                self.min = hodnota
            if self.max is None or self.max < hodnota:
                self.max = hodnota
        else:
            self.visited.add(v1)
            for v in self.pole[v1].sus:
                if v not in self.visited:
                    self.zafarbi_hranu(v1, v, 'red')
                    self.c.update()
                    #self.c.after(100)

                    self.backtracking(v, v2, hodnota+self.pole[v1].sus[v])

                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)
```

### 11.3 Zapamätanie celej cesty

Doterajšie verzie algoritmu generovali všetky cesty, ale si ich nikde neuchovávali. Využijeme pomocné pole (premenná `self.cesta`), do ktorého budeme ukladať momentálnu cestu. Tiež si ju uložíme do premennej `self.najcesta`, keď vyhovuje nejakej podmienke. Po skončení algoritmu `backtracking` túto cestu vypíšeme:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.min = self.max = None
        self.cesta, self.najcesta = [], []
        self.backtracking(v1, v2, 0)
        print('pocet =', self.pocet)
        print('min =', self.min)
        print('max =', self.max)
        print('najkratsia cesta =', self.najcesta)
```

```

def backtracking(self, v1, v2, hodnota):
    self.cesta.append(v1)
    if v1==v2:
        self.pocet += 1
        #print('nasiel som cestu s hodnotou', hodnota)
        if self.min is None or self.min > hodnota:
            self.min = hodnota
            self.najcesta = self.cesta[:]
        if self.max is None or self.max < hodnota:
            self.max = hodnota
    else:
        self.visited.add(v1)
        for v in self.pole[v1].sus:
            if v not in self.visited:
                #self.zafarbi_hranu(v1, v, 'red')
                #self.c.update()
                #self.c.after(100)

                self.backtracking(v, v2, hodnota+self.pole[v1].sus[v])

                #self.zafarbi_hranu(v1, v, 'gray')
        self.visited.remove(v1)
    self.cesta.pop()

```

- pri uchovávaní si najlepšej cesty musíme urobiť kópiu z premennej `self.cesta`, lebo nestačí urobiť len obyčajné priradenie, ktoré by priradilo len referenciu na premennú
- aby sme algoritmus urýchlieli, odstránili sme aj vykresľovanie hrán

Tento algoritmus nájde najlepšiu cestu, ale ju nevykreslí, len vypíše čísla vrcholov. Ďalšia verzia programu vykresľuje najlepšiu cestu - teraz hľadáme nie s najmenšou hodnotou, ale s najväčšou:

```

class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.min = self.max = None
        self.cesta, self.najcesta = [], []
        self.backtracking(v1, v2, 0)
        print('min =', self.min)
        print('max =', self.max)
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, hodnota):
        self.cesta.append(v1)
        if v1==v2:
            if self.min is None or self.min > hodnota:
                self.min = hodnota
                self.najcesta = self.cesta[:]
            if self.max is None or self.max < hodnota:
                self.max = hodnota
        else:
            self.visited.add(v1)
            for v in self.pole[v1].sus:
                if v not in self.visited:

```

```

        self.backtracking(v, v2, hodnota+self.pole[v1].sus[v])
        self.visited.remove(v1)
        self.cesta.pop()
    
```

- možno vám napadlo, že keď si pamäťame cestu, nepotrebujeme udržiavať množinu `self.visited`, ved' `self.cesta` obsahuje tie isté vrcholy - je to pravda, len hľadanie vrcholu v množine (`set`) je pre veľa prvkov rýchlejšie ako v poli (`list`) a takýto program sa môže pre niekoho zdať menej čitateľný
- podobne ako sme namiesto atribútu `self.hodnota` pridali nový parameter `hodnota` do metódy `backtracking()`, môžeme pridať aj parameter `cesta`, v ktorom sa vytvára momentálna postupnosť prechádzaných vrcholov

Hľadanie najkratšej cesty môžeme teraz zapísať:

```

class Graf:
    ...

    def start(self, v1, v2):
        self.min = None
        self.najcesta = []
        self.backtracking(v1, v2, 0, [v1])           # vo vytvárannej ceste je už prvý vrchol
        print('min =', self.min)
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, hodnota, cesta):
        if v1==v2:
            if self.min is None or self.min > hodnota:
                self.min = hodnota
                self.najcesta = cesta
        else:
            for v in self.pole[v1].sus:
                if v not in cesta:
                    self.backtracking(v, v2, hodnota+self.pole[v1].sus[v], cesta+[v])

```

## 11.4 Hľadanie cyklov v grafe

Backtracking môžeme využiť aj na generovanie všetkých cyklov, t.j. takých ciest, v ktorých posledný vrchol je ten istý ako prvý. Využijeme poslednú verziu backtrackingu, v ktorej sme pridali parameter `cesta`. Pritom zmien pre hľadanie cyklu (hľadáme cyklus s maximálnou hodnotou) je veľmi málo:

```

class Graf:
    ...

    def udalost_klik(self, event):
        for i in range(len(self.pole)):
            x, y = self.pole[i].xy
            if (x-event.x)**2+(y-event.y)**2 < R**2:
                self.pole[i].zafarbi('blue')
                self.start(i)
                return

    def start(self, v1):                      # metoda ma iba jeden parameter
        self.max = None

```

```

        self.najcesta = []
        self.backtracking(v1, v1, [], [])           # backtracking štartujeme s_
→prázdnou cestou
        print('max =', self.max)
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, hodnota, cesta):
        if cesta != [] and v1 == v2:
            if len(cesta) > 2 and (self.max is None or self.max < hodnota):
                self.max = hodnota
                self.najcesta = [v2]+cesta      # k nájdenému riešeniu pridáme_
→za začiatok štartový vrchol
        else:
            for v in self.pole[v1].sus:
                if v not in cesta:
                    self.backtracking(v, v2, hodnota+self.pole[v1].sus[v],_
→cesta+[v])

```

Zmenili sme:

- zjednodušili sme metódu `udalost_klik()`, lebo nepotrebjeme ukladať vise kliknutých vrcholov a metódu `start()` voláme len s jedným parametrom: začiatkom cyklu, čo je zároveň aj koncom cyklu
- v metóde `start()` sme zmenili naštrovanie backtrackingu: posielame ako začiatok aj koniec cesty ten istý vrchol a tiež prvý vrchol nevkladáme do vytváanej cesty (parameter `cesta`), aby sme ho tam mohli vložiť aj druhýkrát
- úvodný test v metóde `backtracking()` kontroluje nielen to, či sme už v cieli (teda `v1 == v2`), ale aj dĺžku vytvoreného cyklu, ktorá by mala mať aspoň rôzne 2 hrany, t.j. aspoň 3 vrcholy
- v metóde `backtracking()`, keď nájdeme nejaké vyhovujúce riešenie, ktoré je lepšie ako doteraz nájdené najlepšie (v atribúte `self.najcesta`), pridáme k nemu na začiatok štartový vrchol, lebo ten sme úmyselne do cesty nezaradili

Malým zjednodušením tohto riešenie vieme hľadať najdlhší (alebo aj najkratší) cyklus na počet prejdených vrcholov a to bez ohľadu na hodnoty na hranach:

```

class Graf:
    ...

    def start(self, v1):
        self.najcesta = []
        self.backtracking(v1, v1, [])
        if self.najcesta == []:
            print('nenasiel ziadnu cestu')
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, cesta):
        if cesta != [] and v1 == v2:
            if len(cesta) > 2 and len(self.najcesta) < len(cesta):
                self.najcesta = [v2]+cesta
        else:
            for v in self.pole[v1].sus:
                if v not in cesta:
                    self.backtracking(v, v2, cesta+[v])

```

V prípade, že takýto cyklus prejde cez všetky vrcholy, hovoríme mu **Hamiltonovská kružnica** (v skutočnosti je

Hamiltonovská kružnica podgrafom, ktorý obsahuje všetky vrcholy grafu a hrany len z tohto cyklu).