Algoritmy a dátové štruktúry

rok 2015

Andrej Blaho

sep 12, 2016

Obsah

Fakulta matematiky, fyziky a informatiky

Univerzita Komenského v Bratislave

Autor Andrej Blaho

Názov Algoritmy a dátové štruktúry (prednášky k predmetu Algoritmy a dátové štruktúry 1-AIN-210/15)

Vydavateľ Knižničné a edičné centrum FMFI UK

Rok vydania 2016

Miesto vydania Bratislava

Vydanie prvé

Počet strán 222

Internetová adresa http://struct.input.sk/

ISBN 978-80-8147-069-1

KAPITOLA 1

Úvod do analýzy algoritmov

Pripomeňme si, ako sme doteraz na programovaní v prvom ročníku zvykli merať čas behu programu a podľa toho sme usudzovali o efektívnosti toho ktorého algoritmu. Napr.

```
import time

start = time.time()

...     # meraný algoritmus

cas = time.time() - start
print('algoritmus bežal', round(cas, 3), 'sekúnd')
```

Hoci je jasné, že je to veľmi nepresná metóda, veľa krát to môže naozaj pomôcť.

V praxi sa ale často stretávame aj s problémami, ktoré nevieme takto merať, resp. sa nedajú merať kvôli komplikovaným vstupným podmienkam.

1.1 Odhad počtu inštrukcii

Predpokladajme, že vykonávanie elementárnych inštrukcií trvá procesoru približne rovnako dlho. Niekedy sa zvykne hovoriť, koľko približne inštrukcií dokáže procesor vykonať za jednu sekundu (alebo tisícinu sekundy). Elementárnymi inštrukciami by mohli byť napr. tieto:

- priradenie hodnoty do premennej
- výber hodnoty z premennej
- porovnanie dvoch jednoduchých hodnôt
- aritmetická operácia (npr. sčítanie, násobenie, ...)
- práca s pvkom poľa (výber hodnoty, alebo priradenie) - t.j. prístup pomocou indexu
- volanie funkcie (bez vykonania operácií vo funkcii)
- návrat z funkcie

Pre jednoduchosť budeme predpokladať, že všetky tieto elementárne operácie sa vykonávajú rovnako dlho, napr. 1 časovú jednotku (čo môže byť 1 miliónтina sekundy).

Pozrime chronicky známy algoritmus bublinkového triedenia:

```
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]
```

Ak predpokladáme, že veľkosť poľa je `n`, tak vonkajší cyklus prejde `n`-krát a vnútorný `n-1`-krát. `for`-cyklus môžeme predpokladať, že okrem inicializácie premennej cyklu (1 inštrukcia) sa pri každom prechode vykoná jedno testovanie a jedno zýšenie premennej cyklu o 1, t.j. pri každom prechode 3 elementárne inštrukcie (test + zvýšenie o 1). Vo vnútri cyklu je paralné priradenie do dvoch prvkov poľa, ktoré musíme rozpísať tak, aby sme vedeli spočítať počet elementárnych inštrukcií:

```
# pole[j],pole[j+1] = pole[j+1],pole[j]
pom1 = pole[j+1]        # 5 inštrukcií: priradenie + výber hodnoty pole[] +␣
↪indexovanie + výber hodnoty j + pričítanie
pom2 = pole[j]          # 4 inštrukcie: priradenie + výber hodnoty +␣
↪indexovanie + výber hodnoty
pole[j] = pom1          # 4 inštrukcie
pole[j+1] = pom2        # 5 inštrukcií
```

testovanie v `if` je 8 inštrukcií

Vidíme, že v najhoršom prípade sa vo vnútornom cykle bude po každom testovaní (8 inštr.) aj priraďovať (18 insštr.) Ak k tomu pripočítame inštrukcie na obsluhu `for`-cyklu, dostaneme:

```
1 + (n-1) * (5 + 8 + 18)       # pre len(pole)-1 jedna inštrukcia
```

Vonkajší cyklus to vykoná `n`-krát aj s obsluhou cyklu:

```
1 + n * (5 + 1 + (n-1) * (5 + 8 + 18))
```

Teda v najhoršom prípade to je:

```
31 * n * n - 25 * n + 1
```

Pre naozaj veľké `n` malé členy polynomu nehrajú dôležitú úlohu: rádovo dostávame vzťah, že rýchlosť algoritmu závisí od druhej mocniny `n` teda veľkosti triedeného poľa.

V tejto prednáške nás bude zaujímať väčšinou odhad len pre **najhorší prípad**. Najlepší ani priemerný nebude obsahom tejto prednášky (zoznámite sa stým vo vyšších ročníkoch). Namiesto takého presného počítania s nepresnými odhadmi rýchlosti vykonávania elementárnych inštrukcií, nás bude skôr zaujímať **časová závisloť** behu algoritmu od veľkosti dát (najčastejšie veľkosti poľa). Budeme to značiť funkciou:

```
f(n) = odhad času trvania pre údaje vel'kosti n
```

Takúto funkciu budeme nazývať **časová zložitosť** algoritmu.

Ďalej nás bude zaujímať týchto 7 základných skupín typov funkcií:

1.2 Základné typy funkcií časovej zložitosti

1. **konštantné funkcie**, t.j. funkcie, ktoré nezávisia od veľkosti vstupu, napr. zistenie maximálneho prvku v utriedenom poli:

```
def maxim(utriedene_pole):
    return utriedene_pole[-1]
```

alebo súčet celých čísel od 1 do `n` pomocou vzorca:

```
def sucet(n):
    return n * (n+1) // 2
```

2. **logaritmické funkcie**, napr. zisťovanie, či existuje hodnota v utriedenom poli pomocou binárneho vyhľadávania, napr.

```
def je_kluc(pole, kluc):
    zac = 0
    kon = len(pole)-1
    while zac <= kon:
        stred = (zac + kon) // 2
        if pole[stred][0] > kluc:
            kon = stred - 1
        elif pole[stred][0] < kluc:
            zac = stred + 1
        else:
            return True
    return False
```

   každý prechod `while`-cyklu zmenší hľadaný úsek poľa <`zac`, `kon`> na polovicu, čo sa dá urobiť maximálne `log(n)`-krát t.j. dvojkový logaritmus dĺžky poľa

3. **lineárne funkcie**, napr. prechádzanie každého prvku poľa pomocou `for`-cyklu, napr.

```
def sucet(pole):
    vysl = 0
    for prvok in pole:
        vysl += prvok
    return vysl
```

4. **n log n** funkcie, napr. vo `for`-cykle s každým prvkom poľa vykonáme nejaký algoritmus s **logaritmickou zložitosťou**, alebo niektoré triedenia ako **merge-sort** a **quick-sort**, ale aj štandardná pythonovská funkcia `sorted()`

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

5. **kvadratické funkcie**, napr. skoro všetky funkcie, v ktorých spracovávame prvky poľa dvomi vnorenými `for`-cyklami, napr. **bubble-sort**, **insert-sort**, **min-sort**, ...

6. **kubické funkcie**, v ktorých potrebujeme 3 vnorené `for`-cykly, ale často sú to funkcie len s dvoma vnorenými cyklami, pričom vo vnutormom cykle sa robí operácia, ktrorá má **lineárnu** zložitosť, napr. zreťazovanie celého poľa

7. **exponenciálne funkcie** sú najčastejšie časovou zložitosťou **prehľadávania s návratom**, t.j. algoritmov **backtracking**, asi už máte skúsenosť s tým, že prehľadávanie grafu pomocou backtrackingu dlho trvá aj pre veľmi malé grafy.

1.3 Veľké O

Na formálne označovanie skupiny algoritmov, ktoré patria do jednej skupiny, je zavedené označenie **veľké O**. Ďalej budeme pracovať s funkciami časovej zložitosti, t.j. funkciami, ktoré pre veľkosť vstupu `n` vrátia odhad času (alebo

počtu inštrukcií) najhoršieho prípadu. Ak máme nejakú funkciu `g(n)`, tak `O(g(n))` označuje množinu všetkých takých funkcií `f(n)`, ktoré nemajú horšiu zložitosť ako `g(n)`. Pri takomto porovnávaní časovej zložitosti algoritmov nás ale nebudú zaujímať konštantné koeficienty funkcií, t.j. `O(g(n))` budeme definovať ako množinu funkcií, ktorých zložitosť nie je horšia ako `c * g(n)` pre nejakú konštantu `c`. Keďže chceme zanedbať pomalšie rastúce členy funkcií, budeme sa zaoberať len prípadmi, keď `n` (veľkosť vstupu) je dostatočne veľké, preto formálna definícia veľkého **O()** obsahuje:

- `O(g(n))` je množina funkcií, pre ktoré existuje reálna konštanta `c` a celočíselné `n0`, že `c*g(n) <= f(n)` pre všetky `n > n0`
- t.j. funkcia `f()` rastie rádovo nanajvýš tak rýshlo ako funkcia `g()`
- napr. `O(n**2)` označuje všetky funkcie, ktoré nerastú rýchlešie ako kvadratická funkcia
- namiesto funkcia `f` patrí do `O(g)` budeme často hovoriť aj funkcia `f` je `O(g)`, napr.
  - `31 * n * n -25 * n + 1` je **O(n**2)**
  - `n * (n-1) / 2` je **O(n**2)**
  - aj `10000*n` je je **O(n**2)**, ale je aj je **O(n)**

Môžeme tvrdiť, že čo sa týka časovej zložitosti, platí:

- **O(1) < O(log n) < O(n) < O(n log n) < O(n**2) < O(n**3) < O(2**n)**

t.j. najefektívnejšie sú algoritmy s časovou zložitosťou **O(1)**, trochu horšie sú na tom algoritmy s **O(log n)**, atď.

1.4 Porovnanie zložitosti

Predpokladajme, že náš počítač zvládne za jednu milisekundu (tisícinu sekundy) vykonať 5000 elementárnych operácií, inými slovami zvládne spracovať vstup veľkosti 5000 pri lineárnej zložitosti. Skúsme odhadnuť, aký veľký vstup by náš počítač zvládol pre rôzne typy zložitosti a pre rôzny čas: teda za jednu milisekundu, sekundu, minútu, hodinu, deň, mesiac, rok a 1000 rokov. Vidíme, že pre pre niektoré algoritmy s veľkou zložitosťou nám nepomôže pre väčšie vstupy ani niekoľko rokov:

	n	nlogn	n**2	n**3	2**n
mili	5000	550	71	17	12
sek	5000000	280000	2200	170	22
min	300000000	13000000	17000	670	28
hod	vel'a	620000000	130000	2600	34
den	vel'a	vel'a	660000	7600	39
mesiac	vel'a	vel'a	2300000	17000	42
rok	vel'a	vel'a	13000000	54000	47
1000	vel'a	vel'a	400000000	540000	57

1.5 Cvičenie

1. zadefinujte triedu `Pole` s týmito metódami:
   - `__getitem__()`, `__setitem__()`, `__len__()`, `__contains__()`, `__add__()`
   - `append()`, `pop()`, `insert()`, `remove()`, `index()`

   Tieto metódy by sa mali správať rovnako ako sú definované v triede `list`. Vnútorne pre túto triedu využite atribút `pole` typu `list`, pričom využijete len tieto jeho operácie: `pole[index] = hodnota`, `pole[index]`, `len(pole)`, `pole = [None]*číslo`

- pre každú metódy odhadnite časovú zložitosť **O()**
- otestujte funkčnosť vašich metód pomocou nejakých jednoduchých testov

2. v prvom ročníku sme aj takto definovali rad `Queue` pomocou poľa:

```
class EmptyError(Exception): pass

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

    def is_empty(self):
        return len(self.pole) == 0

    def enqueue(self, data):
        self.pole.append(data)

    def dequeue(self):
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole.pop(0)

    def front(self):
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole[0]
```

   Predefinujte tieto metódy tak, aby pracoval na princípe cyklického poľa:

   - na začiatku má atribút `pole` nejakú dĺžku (napr. 10) a pamätá sa, že v ňom zatiaľ nie sú uložené žiadne údaje
   - metóda `enqueue()` pridá novú hodnotu za posledne pridávanú do poľa (cyklicky modulo dĺžka poľa), pričom si zapamätá nový počet vložených údajov; ak sa hodnota do poľa už nezmestí, tak pole nafúkne (dvojnásobne zväčší jeho dĺžku) a novú hodnotu potom vloží
   - metóda `dequeue()` zoberie prvý vložený údaj, posunie si index vložených údajov (cyklicky modulo dĺžka poľa) a zníži ich počet

   Funkčnosť otestujte a porovnajte rýchlosť s pôvodnou verziou triedy (spustite pre veľký počet údajov).

3. Naprogramujte funkciu `sort0(pole)` podľa algoritmu **bubble-sort**, pričom parameter je typu `tuple`. Funkcia vráti utriedené pole ako `tuple`, pričom pracuje len s n-ticami (nepoužíva žiadny iný štruktúrovaný typ, napr. `list`). Uvedomte si, že pre `t` typu `tuple` do `i`-teho prvku priradíme novú hodnotu napr. pomocou `t = t[:i]+(hodnota,)+t[i+1:]`.

   - odhadnite časovú zložitosť algoritmu (operácia + pre n-tice má zložitosť **O(n)**)

```
def sort0(pole):
    ...
    return nove_pole
```

4. v 1. ročníku sme takto definovali triedenie **insert sort**:

```
def sort1(pole):
    for i in range(1,len(pole)):
        t = pole[i]
        j = i-1
        while j >= 0 and pole[j] > t:
            pole[j+1] = pole[j]
```

```
            j -= 1
        pole[j+1] = t
```

Zistite, pre aké najväčšie n-prvkové pole triedenie na vašom počítači beží 1 sekundu, 2 sekundy, 3 sekundy ... Polia generujte s náhodnými hodnotami. Hľadanie takýchto n skúste nejako zautomatizovať.

Zistite, koľko elementárnych inštrukcií sa vykoná v najhoršom prípade. Elementárnymi inštrukciami sú: výber hodnoty premennej, priradenie do premennej, indexovanie do poľa, aritmetická operácia, porovnanie dvoch hodnôt, zistenie dĺžky poľa.

5. napíšte druhú verziu tohto triedenia, v ktorom sa zaraďovanie prvku na správne miesto utriedenej časti poľa použije metóda `list.insert()` a vyhodenie prvku pomocou `del`

```
def sort2(pole):
    ...
```

Zistite, aké najväčšie n-prvkové pole sa dá teraz utriediť za 1 sekundu, za 2 sekundy, ...

6. ďalšia verzia tohto triedenia bude postupne vytvárať nové utriedené pole, do ktorého bude vkladať (zreťazovaním polí) ďalšie prvky na správne miesto a na záver obsah tohto utriedeného pomocného poľa prekopíruje do formálneho parametra funkcie

```
def sort3(pole):
    pom = [pole[0]]
    for prvok in pole[1:]:
        # nájdi index ix v pom, kam zaradit' prvok
        pom = pom[:ix] + [prvok] + pom[ix:]
    # kopíruj pom do pole
```

Zistite, aké najväčšie n-prvkové pole sa dá teraz utriediť za 1 sekundu, za 2 sekundy, ...

7. navrhnite funkciu `test()`, pomocou ktorej budete merať rýchlosť vykonávania rôznych algoritmov triedenia:

- prvým parametrom bude veľkosť poľa - váš algoritmus vygeneruje náhodné pole a toto sa použije pre rôzne verzie algoritmu
- program potom postupne spustí triedenie pre všetky zadané verzie algoritmu s tým istým poľom
- odmeria čas a skontroluje správnosť utriedenia poľa (môžete použiť funkciu `sorted()`)
- na záver vypíše tabuľku: číslo algoritmu, čas trvania, správnosť utriedenia (True/False)

```
def test(n, zoznam_algoritmov):
    ...

def sort1(pole):
    ...

def sort2(pole):
    ...

def sort3(pole):
    ...

>>> test(10000, [sort1, sort2, sort3])
```

KAPITOLA 2

Polia, iterátory

V Pythone sú 3 sekvenčné typy, ktoré sú vnútrone reprezentované ako dynamické polia:

- `str` - pole znakov - nemeniteľný typ (immutable)
- `tuple` - pole referencií na objekty - nemeniteľný typ (immutable)
- `list` - pole referencií na objekty - meniteľný typ (mutable)

Idea polí v počítači:

- RAM (random access memory) - rovnaký čas na prístup k ľubovoľnému pamäťovému miestu = O(1)
- prvky poľa sú postupne uložené v pamäťových miestach tak, aby sa dala čo najjednoduchšie vypočítať adresa ľubovoľného prvku
  - adresa `A[i]` = adresa `A` + `i` * počet_bajtov_prvku
- typ `str` je v Pythone realizovaný ako kompaktné pole znakov - každý znak je v Unicode a pravdepodobne zaberá 4 bajty
- typy `tuple` a `list` sú poliami referencií na objekty (smerníky)

2.1 Kompaktné pole v Pythone

Modul `array` umožňuje definovať kompaktné číselné pole (na rozdiel od poľa referencií na objekty čísla). Takto definované pole poskytuje komfort operácií a metód triedy `list`. Môže sa využiť napr. pri práci s binárnymi súbormi. Napr.

```
>>> import array
>>> a = array.array('b', [2,3,5,7,11,13,17,19])
>>> a
array('b', [2, 3, 5, 7, 11, 13, 17, 19])
>>> a = a + a
>>> a
array('b', [2, 3, 5, 7, 11, 13, 17, 19, 2, 3, 5, 7, 11, 13, 17, 19])
>>> b = array.array('B', [1]*10)
>>> b
array('B', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> for i in range(1,len(b)): b[i] = 2*b[i-1]
...
OverflowError: unsigned byte integer is greater than maximum
>>> b
```

```
array('B', [1, 2, 4, 8, 16, 32, 64, 128, 1, 1])
>>>
```

Definované typy pre kompaktné pole:

kód	C-typ	Python-typ	bajty
'b'	signed char	int	1
'B'	unsigned char	int	1
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

2.2 Dynamické pole a amortizácia

Otestujme metódu `append` štandardného typu `list`. Sledovať budeme to, ako sa mení vnútorná pamäť Pythonu pre danú štruktúru (funkcia `getsizeof` z modulu `sys` vráti počet bajtov):

```
import sys

def zisti(n):
    pole = []
    for i in range(n):
        size = sys.getsizeof(pole)
        print('len:{:4} sizeof:{:6}'.format(len(pole), size))
        pole.append(None)

zisti(12)
```

zdá sa, že 64 bajtov používa Python pre základné info o objekte typy `list` (pod iným OS a inou verziou Pythonu môžete mať iné výsledky):

```
len:   0 sizeof:    64
len:   1 sizeof:    96
len:   2 sizeof:    96
len:   3 sizeof:    96
len:   4 sizeof:    96
len:   5 sizeof:   128
len:   6 sizeof:   128
len:   7 sizeof:   128
len:   8 sizeof:   128
len:   9 sizeof:   192
len:  10 sizeof:   192
len:  11 sizeof:   192
```

Python si pre `list` vyhradí vždy aj nejakú rezervu prvkov, aby niekoľko ďalších `append` nemusel objekt už zväčšovať

Sledujme, pri akých hodnotách sa nafukuje štandarné pythonovské dynamické pole (t.j. `list`, opäť predpokladáme, že jedna referencia zaberá 8 bajtov, preto vyhradenú pamäť delíme 8):

```
import sys

def zisti(n):
    pole = []
    size0 = 0
    for i in range(n):
        size = sys.getsizeof(pole)
        if size != size0:
            size0 = size
            print('len:{:4} sizeof:{:6}  {:3}'.format(len(pole), size, (size-
↪64)//8))
        pole.append(None)

zisti(300)
```

Interval medzi nafukovaním poľa sa stále zväčšuje:

```
len:   0 sizeof:    64    0
len:   1 sizeof:    96    4
len:   5 sizeof:   128    8
len:   9 sizeof:   192   16
len:  17 sizeof:   264   25
len:  26 sizeof:   344   35
len:  36 sizeof:   432   46
len:  47 sizeof:   528   58
len:  59 sizeof:   640   72
len:  73 sizeof:   768   88
len:  89 sizeof:   912  106
len: 107 sizeof:  1072  126
len: 127 sizeof:  1248  148
len: 149 sizeof:  1448  173
len: 174 sizeof:  1672  201
len: 202 sizeof:  1928  233
len: 234 sizeof:  2216  269
len: 270 sizeof:  2536  309
```

Aby sme lepšie pochopili, ako to v Pythone funguje, definujme vlastné dynamické pole referencií na pythonovské objekty. Využijeme nízkoúrovňové (na úrovni jazyka C) definovanie poľa pomocou modulu `ctype`. Pri metóde `append` budeme pole nafukovať vždy o dvojnásobok dĺžky:

```
import ctypes

class DynamickePole:
    def __init__(self):
        self.n = 0
        self.vyhr = 1
        self.pole = self.vyrob_pole(self.vyhr)

    def __len__(self):
        return self.n

    def __repr__(self):
        res = ''
        for i in range(self.n):
            res += ', ' + repr(self.pole[i])
```

```
        return 'dyn[' + res[2:] +']'

    def append(self, prvok):
        if self.n == self.vyhr:
            self.resize(2 * self.vyhr)
        self.pole[self.n] = prvok
        self.n += 1

    def resize(self, velkost):
        pole2 = self.vyrob_pole(velkost)
        for i in range(self.n):
            pole2[i] = self.pole[i]
        self.pole = pole2
        self.vyhr = velkost

    def vyrob_pole(self, d):
        return (d * ctypes.py_object)()

a = DynamickePole()
for i in range(20):
    a.append(i)
print(a)
```

po spustení:

```
dyn[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

otestujeme metódu `append()`:

```
def zisti(n):
    pole = DynamickePole()
    size0 = 0
    for i in range(n):
        size = pole.vyhr
        if size != size0:
            size0 = size
            print('len:{:4} sizeof:{:6}'.format(len(pole), size))
        pole.append(None)

zisti(200)
```

naozaj sa zdvojnásobuje dĺžka poľa:

```
len:   0 sizeof:     1
len:   2 sizeof:     2
len:   3 sizeof:     4
len:   5 sizeof:     8
len:   9 sizeof:    16
len:  17 sizeof:    32
len:  33 sizeof:    64
len:  65 sizeof:   128
len: 129 sizeof:   256
```

Vďaka tomuto mechanizmu má "priemerný" čas operácie `append()` zložitosť **O(1)**, tzv. **amortizovaná zložitosť**:

- ak by sme zisťovali zložitosť metódy `append` postupne pre prázdny zoznam, jednoprvkový, dvojprvkový, ... až po `n`, a celkovú zložitosť (týchto `n` volaní `append`) vydelíme `n`, dostaneme "priemernú" zložitosť
- v našej realizácii sa pole nafukuje len pri mocninách 2 a medzi tým je veľakrát zložitosť **O(1)** - jednu časovo

náročnú operáciu, vieme rozložiť k rýchlym (pri rýchlych operáciách si vieme ušetriť - amortizovať - dosť, aby sme si mohli raz začas dovoliť jednu časovo náročnú operáciu)

- ak by sme nezdvojnásobovali veľkosť poľa ale ju len zväčšovali o konštantu, napr. 2, amortizovaná zložitosť bude **O(n)**
- amortizovaná zložitosť **O(1)** bude aj vtesy, keď interval zväčšovania veľkosti poľa nebude konštantný ale geometrická postupnosť

Otestujme, ako je to s časom pre pythonovskú metódu `append()`:

```
import time

def priemer(n):
    pole = []
    start = time.time()
    for i in range(n):
        pole.append(None)
    cas = time.time()-start
    return cas*1000000/n

for n in 100, 1000,10000,100000, 1000000, 10000000, 100000000:
    print('{:10} {:10.3f}'.format(n, priemer(n)))
```

Zdá sa, že `append` v Pythone má (amortizovanú) zložitosť **O(1)** - dostávame skoro rovnaké výsledky:

```
       100      0.000
      1000      1.001
     10000      0.301
    100000      0.250
   1000000      0.330
  10000000      0.305
 100000000      0.308
```

2.3 Zložitoť pythonovských operácií na sekvenčných typoch

operácie, ktoré nemenia obsah `list`, resp. `tuple` (immutable):

operácia	zložitosť
`len(pole)`	O(1)
`pole[j]`	O(1)
`pole.count(obj)`	O(n)
`pole.index(obj)`	O(k+1)
`obj in pole`	O(k+1)
`pole1 == pole2`	O(k+1)
(tiež !=, <, <=, >, >=)	
`pole[j:k]`	O(k-j+1)
`pole1 + pole2`	O(n1+n2)
`c * pole`	O(cn)

operácie, ktoré menia obsah `list` (mutable):

operácia	zložitosť
`pole[j] = obj`	O(1)
`pole.append(obj)`	O(1)*
`pole.insert(k,obj)`	O(n-k+1)*
`pole.pop()`	O(1)*
`pole.pop(k)`	O(n-k)*
`del pole[k]`	O(n-k)*
`pole.remove(obj)`	O(n)*
`pole1.extend(pole2)`	O(n2)*
`pole1 += pole2`	O(n2)*
`pole.reverse()`	O(n)
`pole.sort()`	O(nlog n)

(* amortizovaná)

2.4 Znakové reťazce

keďže sú **immutable**, vždy sa konštruuje nový reťazec:

```
s1 = '... dlhý ret'azec ...'
s2 = ''
for znak in s1:
    if 'A' <= znak <= 'z':
        s2 += znak
```

zložitosť je **O(n**2)** lebo `s2` sa stále zväčšuje o 1 znak a pritom sa stále kopíruje pôvodný reťazec do nového reťazca; jeho dĺžka (ktorá sa kopíruje) je postupne 1, 2, 3, ..., n

ak využijeme pole a operáciu `append`:

```
s1 = '... dlhý ret'azec ...'
pom = []
for znak in s1:
    if 'A' <= znak <= 'z':
        pom.append(znak)
s2 = ''.join(pom)
```

čo má zložitosť **O(n)**

môžeme urýchliť:

```
s2 = ''.join([znak for znak in s1 if 'A' <= znak <= 'z'])
```

alebo

```
s2 = ''.join(znak for znak in s1 if 'A' <= znak <= 'z')
```

2.5 Iterovateľný typ

V Pythone sú niektoré základné typy iterovateľné - môžeme prechádzať ich prvky napr. pomocou for cyklu:

- `list`, `tuple`, `str`, `dict`, `set`
- postupnosť celých čísel `range(...)`, otvorený súbor na čítanie `open(...)`

- výsledky funkcií `map()` a `filter()` aj generátorová notácia `[... for ...]`

Keď máme vlastný definovaný typ, aj pre tento môžeme zabezpečiť **iterovateľnosť**. Možností je niekoľko, dnes ukážeme dve z nich:

- v triede zadefinujeme magickú metódu `__getitem__()`: v prípade prechádzania pomocou for-cyklu, Python zabezpečí postupné generovanie indexov od 0 vyššie a pri prvom neexistujúcom prvku, skončí
- v triede zadefinujeme dvojicu magických metód `__iter__()` a `__next__()`, ktoré zabezpečia **iterovateľnosť**

for-cyklus by sme si mohli v Pythone zapísať pomocou **while**-cyklu a iterátora. Napr. takýto for-cyklus:

```
pole = [2, 3, 5, 7, 11, 13, 17]
for i in pole:
    print(i, i*i)
```

môžeme pomocou iterátora prepísať:

```
iterator = iter(pole)
while True:
    try:
        i = next(iterator)
        print(i, i*i)
    except StopIteration:
        break
```

Funguje to takto:

- Python si najprv z daného typu vyrobí špeciálny objekt (tzv. iterátor), pomocou ktorého bude neskôr postupne prechádzať všetky prvky
- iterátor sa vytvára štandardnou funkciou `iter()`, ktorá by pre neiterovateľný typ spadla s chybovou hláškou
- ďalšia štandardná funkcia `next()` z iterátora vráti nasledovnú hodnotu, alebo vyhlási chybu `StopIteration`, ak už ďalšia neexistuje

Môžete to vidieť aj na tomto príklade s iterovaním znakového reťazca:

```
>>> it = iter('ahoj')
>>> next(it)
'a'
>>> next(it)
'h'
>>> next(it)
'o'
>>> next(it)
'j'
>>> next(it)
...
StopIteration
```

Ak v našej triede zadefinujeme metódy `__iter__()` a `__next__()`, tieto metódy sa automaticky zavolajú zo štandardných funkcií `iter()` a `next()`. Metóda `__iter__()` najčastejšie obsahuje vrátenie seba ako svojej hodnoty `return self`, lebo predpokladáme, že samotná inštancia je potom iterátor a teda tento iterátor musí obsahovať aj definíciu metódy `__next__()`. Táto druhá metóda sa automaticky zavolá pri volaní štandardnej funkcie `next()`. Preto musí metóda `__next__()` skontrolovať, či ešte má nasledovnú hodnotu (vtedy ju vráti) alebo vyvolá výnimku `StopIteration`.

Vyskúšajte:

```
class Test:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        self.ix = 0
        return self

    def __next__(self):
        if self.ix >= self.n:
            raise StopIteration
        self.ix += 1
        return self.ix-1

for i in Test(5):
      print(i)
```

2.6 Cvičenie

1. upravte testovanie vyhradzovanej pamäti (funkcia `zisti()` z prednášky) pre pythonovský `list.append()` (kde sa sleduje, pri akých hodnotách sa nafukuje vnútorné pole)
   - funkciu upravte pre váš počítač

     ```
     def zisti():
         ...
     ```

   - údaje vložte do excelovskej tabuľky a graficky znázornite priebeh funkcie, na základe ktorej Python určuje veľkosť pridanej pamäti pri `append` aj pre väčšie polia
2. v prednáške je aj príklad so spracovaním dlhého znakového reťazca
   - so zložitosťou **O(n**2)**

     ```
     s1 = '... dlhý ret'azec ...'
     s2 = ''
     for znak in s1:
         if 'A' <= znak <= 'z':
             s2 += znak
     ```

   - pomocou pomocného poľa sa úloha rieši so zložitosťou **O(n)**

     ```
     s1 = '... dlhý ret'azec ...'
     pom = []
     for znak in s1:
         if 'A' <= znak <= 'z':
             pom.append(znak)
     s2 = ''.join(pom)

     #s2 = ''.join([znak for znak in s1 if 'A' <= znak <= 'z'])
     #s2 = ''.join(znak for znak in s1 if 'A' <= znak <= 'z')
     ```

   - otestujte reálnu rýchlosť pre reťazce rôznych dĺžok (100000, 1000000, ..., 10000000)
     - môžete využiť napr. stránky s textami na internete: biblia v angličtine 4.24 MB (http://www.gutenberg.org/cache/epub/10/pg10.txt), Vojna a mier od Tolsteho 3.14 MB

(http://www.gutenberg.org/cache/epub/2600/pg2600.txt) alebo manuál k programovaniu PC hier 1.95 MB (http://textfiles.com/programming/pcgpe10.txt)

- porovnajte 3 rôzne rýchle verzie

3. v prednáške sme skúmali použitú pamäť (pomocou `sys.getsizeof()`) a rýchlosť metódy `append()` (amortizovaná zložitosť)
   - zistite, ako sa mení použitá pamäť a priemerný čas pre operáciu `list.pop()` (`pop()` bez parametra), skúmajte pre veľké polia

```
import time

def priemer(n):
    pole = [... veľké pole ...]
    start = time.time()
    ...
```

4. do triedy `DynamickePole` dodefinujte metódu `pop()` - navrhnite vlastnú stratégiu, kedy a o koľko sa bude pole zmenšovať (napr. keď je vyhradená pamäť `n` využitá len na `n/4`, tak sa zmeší na `n/2`)
   - otestujte ju rovnakými testami ako v úlohe (3)

```
class DynamickePole:
    ...
    def pop(self):
        ...
```

5. otestujte či je trieda `DynamickePole` iterovateľná (či sa dajú vypísať jeho prvky pomocou for-cyklu)
   - pridajte metódu `__getitem__()` a otestujte:

```
class DynamickePole:
    ...
    def __getitem__(self, index):
        ...
```

   - pridajte metódy `__iter__()` a `__next__()` a otestujte

```
class DynamickePole:
    ...
    def __iter__(self):
        ...
    def __next__(self):
        ...
```

6. použite triedu spájaný zoznam z prvého ročníka:
   - zistite, či funguje napr. `for data in Zoznam(range(10)):`

```
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 data in pole:
                self.pridaj_kon(data)
```

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

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

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

- definujte `__getitem__()` a otestujte iterovateľnosť - odmerajte čas pre väčší spájaný zoznam, napr.

```
zoz = Zoznam(range(10000))
print('pocitam...')
sucet = 0
for p in zoz:
    sucet += p
print(p)
```

- namiesto `__getitem__()` definujte metódy `__iter__()` a `__next__()` tak, aby sa prvky zoznamu dali prechádzať for-cyklom, otestujte funkčnosť a porovnajte rýchlosť s predchádzajúcim testom
- zdôvodnite výrazný rozdiel v čase behu oboch testov

7. Naprogramujte tri rekurzívne verzie funkcie, ktorá počíta n-ty člen **fibonacciho postupnosti** (0,1,1,2,3,5,8,13,...):
   1. funkcia `fib1(n)`, ktorá rekurzívne volá samu seba pre (n-1) aj pre (n-2) člen
   2. funkcia `fib2(n)`, ktorá rekurzívne volá samu seba, ale keďže vracia dvojicu hodnôt (n-tý člen, n-1-člen), môže fungovať výrazne efektívnejšie ako `fib1()`
   3. funkcia `fib3(n)`, ktorá pracuje skoro rovnako ako funkcia `fib1(n)`, ale pamätá si v tabuľke doterajšie vypočítané hodnoty a ak by nejakú mala znovu počítať, tak ju len vyberie z tabuľky
   - odhadnite zložitosť všetkých verzií, odmerajte beh aj pre väčšie `n` (asi `fib1()` už nepobeží na niektorých vstupoch)

   Tretia (c) verzia sa nazýva **memoizácia**: funkcia si pamätá všetky doterajšie výsledky a keby ich mala

počítať znovu, tak iba siahne do tejto pamäte. Môžete využiť ideu nepovinného druhého parametra funkcie, ktorý bude typu `dict` (keď budete túto funkciu volať len s prvým parametrom, druhý parameter sa vnútorne pamätá ako jediná premenná, ktorá si uchováva obsah aj vo všetkých ďalších volaniach):

```
def fib3(n, memo={}):
    if n in memo:
        return memo[n]
    ...    # podobné ako fib1(), ale zapamätá si aj výsledok do
↪tabul'ky
```

8. zapíšte funkciu, ktorá zisťuje, či sú všetky prvky poľa **navzájom rôzne** (funkcia vráti `True` alebo `False`):

- rôzne verzie funkcie:

```
def zisti1(pole):
    '''pre každý prvok pol'a prejde zvyšok pol'a a hl'adá, či sa tam nenachádza
↪rovnaký'''

def zisti2(pole):
    '''rekurzívne:
           (1) zistí, či sú všetky rôzne, ak sa vynechá prvý prvok
           (2) potom zistí, či sú všetky rôzne, ak sa vynechá len posledný prvok
           ak platí (1) aj (2), ešte porovná prvý a posledný prvok'''

def zisti3(pole):
    '''z pol'a skonštruuje množinu a zistí či má táto rovnaký počet prvkov ako
↪samotné pole
       - môžete predpokladat', že zložitost' vytvorenia množiny z n-prvkov je
↪O(n)
    '''
```

- odhadnite zložitosť a porovnajte rýchlosť všetkých troch algoritmov

KAPITOLA 3

Stromy a generátory

Pripomeňme si, čo vieme o stromoch z programovania v prvom ročníku:

- strom je množina vrcholov (**node**), v ktorej okrem jedného vrcholu (tzv. koreň stromu teda **root**) má každý vrchol práve jedného predka (tiež hovoríme otec alebo **parent**)
- každý vrchol má množinu potomkov (tzv. synov alebo **children**) = sú to tie vrcholy, pre ktoré je tento vrchol otcom
- vrcholu, ktorý nemá žiadnych potomkov, hovoríme list (tiež vonkajší alebo niekedy ako external alebo **leaf**)
- vrcholu, ktorý má aspoň jedného potomka, hovoríme vnútorný (**internal**)
- vrcholom, ktoré majú spoločného predka, hovoríme bratia, súrodenci, resp. **siblings**
- hrana stromu (**edge**) je dvojica vrcholov `(u,v)`, v ktorej buď `v` je otcom `u`, alebo naopak
- cesta (**path**) je postupnosť vrcholov, v ktorej každé dva súsedné vrcholy sú hranou

Neprázdny strom môžeme definovať napr. takto:

- ako jeden špeciálny vrchol - koreň
- pre všetky zvyšné vrcholy (rôzne od koreňa) platí, že každý z nich má jediný iný vrchol v strome, ktorý je jeho otcom

Niekedy sa strom definuje aj rekurzívne:

- strom je buď prázdny
- alebo sa skladá z jedného vrchola `r` (koreň stromu) a množiny **podstromov** (možno prázdnej), ktorých korene sú synmi vrcholu `r` (zrejme podstromy sú tiež stromy a platí pre nich tiež táto definícia)

3.1 Abstraktný dátový typ

obsahuje metódy, ktoré by mali fungovať bez ohľadu na konkrétnu implementáciu pomocou nejakej dátovej štruktúry:

- `t.root()` - vráti **koreň** stromu, resp. `None`, ak je strom prázdny
- `t.parent(n)` - pre daný vrchol `n` stromu `t` vráti jeho **predka**
- `t.children(n)` - pre daný vrchol stromu vráti postupnosť jeho **potomkov**
- `t.num_children(n)` - pre daný vrchol stromu zistí počet jeho **potomkov**
- `len(t)` - zistí počet všetkých vrcholov stromu (táto funkcia je definovaná ako metóda `__len__`)
- `t.data(n)` - pre daný vrchol stromu vráti hodnotu vo vrchole

- `t.is_root(n)` - pre daný vrchol stromu zistí, či je to **koreň**, teda pre koreň vráti `True`
- `t.is_leaf(n)` - pre daný vrchol stromu zistí, či je to **list**, teda vráti `True`, ak vrchol nemá žiadnych potomkov
- `t.is_empty()` - zistí, či je strom prázdny
- `t.__iter__()` - vráti iterovateľný objekt všetkých vrcholov stromu
  - najčastejšie to bude generátorový objekt
  - vďaka tomu budeme môcť zapísať konštrukciu `for vrchol in strom: ...`, pomocou ktorej môžeme postupne (v nejakom poradí) navštíviť a spracovať každý vrchol stromu

Zapíšme to ako základnú (bázovú) abstraktnú triedu, z ktorej budeme odvádzať ďalšie triedy:

```
class Tree:
    def root(self):
        raise NotImplementedError()

    def parent(self, node):
        raise NotImplementedError()

    def children(self, node):
        raise NotImplementedError()

    def num_children(self, node):
        raise NotImplementedError()

    def __len__(self):
        raise NotImplementedError()

    def data(self, node):
        raise NotImplementedError()

    def is_root(self, node):
        return self.root() == node

    def is_leaf(self, node):
        return self.num_children(node) == 0

    def is_empty(self):
        return len(self) == 0

    def __iter__(self):
        raise NotImplementedError()
```

Zrejme, kým nepredefinujeme všetky abstraktné metódy (ktoré vyvolajú výnimku `NotImplementedError`), nemá zmysel vytvárať inštancie tejto triedy. Všimnite si, že nie všetky metódy sú abstraktné: niektoré sme vedeli zapísať pomocou ostatných, napr. metóda `is_empty()` je definovaná pomocou metódy `__len__()`, teda keď neskôr zadefinujeme `__len__()`, bude fungovať aj `is_empty()`.

Takto by to fungovalo v poriadku, hoci inštancia triedy `Tree` by bola úplne nepoužiteľná. Až inštancie z odovodených tried, v ktorých všetky **abstraktné metódy** (obsahujú `raise NotImplementedError()`) sú prekryté konkrétnou realizáciou, budú mať nejaký zmysel. V praxi, hlavne pri väčších projektoch, sa zaužívala prax, v ktorej **abstraktný dátový typ** doplníme o špeciálnu kontrolu:

- kým sa nerealizujú všetky abstraktné metódy, tak z takejto triedy nedovolí vytvoriť inštanciu
- pri definovaní tejto našej abstraktnej triedy zapíšeme špeciálneho predka triedy: `metaclass=ABCMeta`, čím sa zabezpečí samotná kontrola

- okrem toho každú abstraktnú metódu označíme popisom (tzv. dekorátorom) `@abstractmethod`, aby kontrola vedela zistiť, ktoré metódy má strážiť, aby neostali abstraktné
- obe označenia `ABCMeta` a `abstractmethod` sú definované v štandardnom module `abc` (čo znamená modul **abstract base classes**)

Prepíšme abstraktný dátový typ `Tree` s využitím modulu `abc`:

```
from abc import ABCMeta, abstractmethod

class Tree(metaclass=ABCMeta):
    @abstractmethod
    def root(self):
        pass

    @abstractmethod
    def parent(self, node):
        pass

    @abstractmethod
    def children(self, node):
        pass

    @abstractmethod
    def num_children(self, node):
        pass

    @abstractmethod
    def __len__(self):
        pass

    @abstractmethod
    def data(self, node):
        pass

    def is_root(self, node):
        return self.root() == node

    def is_leaf(self, node):
        return self.num_children(node) == 0

    def is_empty(self):
        return len(self) == 0

    @abstractmethod
    def __iter__(self):
        pass
```

Keď sa teraz pokúsime vytvoriť inštanciu tejto triedy, dostávame chybovú správu:

```
>>> t = Tree()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    t = Tree()
TypeError: Can't instantiate abstract class Tree with abstract methods __
 ↪iter__,
__len__, children, data, num_children, parent, root
```

Vidíme, že Python teraz stráži, vytváranie inštancie a vypisuje, ktoré abstraktné metódy treba ešte implementovať.

3.2 Hĺbka a výška

Do triedy `Tree` pridáme ešte dve ďalšie metódy, ktoré sú pre dátovú štruktúru veľmi dôležité:

- **hĺbka** vrcholu (metóda `depth`):
  - vzdialenosť konkrétneho vrcholu od koreňa stromu
  - teda pre daný vrchol zistíme jeho predka, pre predka zistíme jeho predka a toto budeme opakovať, kým prídeme na vrchol, ktorý už predka nemá (otca) a počet týchto prechodov na predkov označuje vzdialenosť
  - zrejme koreň stromu má hĺbku 0
- **výška** vrcholu, resp. celého stromu (metóda `height`):
  - vzdialenosť konkrétneho vrcholu od najvzdialenejšieho listu v tomto podstrome
  - teda budeme generovať všetky cesty od daného vrcholu ku všetkým listom a maximálna dĺžka cesty je výška vrcholu
  - výškou celého stromu rozumieme výšku koreňa stromu
  - zrejme všetky listy majú výšku 0

Obe tieto metódy pridáme do základnej triedy `Tree`:

```
class Tree(metaclass=ABCMeta):

    ...

    def depth(self, node):
        if self.is_root(node):
            return 0
        return 1 + self.depth(self.parent(node))

    def height(self, node=None):
        if node is None:
            node = self.root()
        if node is None or self.is_leaf(node):
            return 0
        return 1 + max(self.height(n) for n in self.children(node))
```

Vďaka tomu, že obe tieto metódy využívajú len ďalšie metódy základnej triedy, môžeme ich definovať už na tejto úrovni a teda každá ďalšia implementácia stromu, ktorá vychádza z bázovej triedy `Tree`, má už zadefinované obe tieto nie až tak jednoduché metódy. Zložitosť oboch algoritmov pre hĺbku aj výšku stromu je **O(n)**.

Výšku celého stromu by sme mohli počítať aj ako maximum hĺbok všetkých listov stromu:

```
class Tree(metaclass=ABCMeta):

    ...

    def height1(self):
        return max(self.depth(v) for v in self if self.is_leaf(v))
```

Tento algoritmus je veľmi neefektívny: **O(n**2)**

- zápis `for v in self` zavolá metódu `strom.__iter__()`, t.j. postupne vygeneruje všetky vrcholy stromu (predpokladáme, že jeho zložitosť je **O(n)**

3.3 Binárne stromy

vlastnosti:

- každý vrchol má maximálne dvoch potomkov (synov), pričom sú pomenované ako ľavý a pravý syn
- v zozname potomkov (metóda `children()`) je uvedený najprv ľavý a potom pravý syn

rekurzívna definícia: binárny strom je buď prázdny alebo sa skladá z

- koreňa, v ktorom je uložená nejaká informácia
- binárneho stromu (možno prázdneho), ktorý sa volá ľavý podstrom
- binárneho stromu (možno prázdneho), ktorý sa volá pravý podstrom

ďalšie metódy do ADT pre binárny strom:

- `t.left(node)` - ľavý syn alebo `None`
- `t.right(node)` - pravý syn alebo `None`
- `t.sibling(node)` - súrodenec vrcholu alebo `None`

Teraz už vieme definovať aj metódu `children()` - metóda vráti zoznam všetkých synov vrcholu - zoznam môže byť prázdny, jednoprvkový alebo dvojprvkový.

Aj trieda `BinaryTree` ja abstraktný dátový typ, lebo tiež obsahuje abstraktné metódy (predpokladáme, že trieda `Tree` je definovaná v súbore `tree.py`):

```
from abc import abstractmethod
from tree import Tree

class BinaryTree(Tree):
    @abstractmethod
    def left(self, node):
        pass

    @abstractmethod
    def right(self, node):
        pass

    def sibling(self, node):
        parent = self.parent(node)
        if parent is None:
            return None
        if self.left(parent) == node:
            return self.right(parent)
        return self.left(parent)

    def children(self, node):
        res = []
        if self.left(node) is not None:
            res.append(self.left(node))
        if self.right(node) is not None:
            res.append(self.right(node))
        return res

    def num_children(self, node):
        count = 0
        if self.left(node) is not None:
            count += 1
```

```
        if self.right(node) is not None:
            count += 1
        return count
```

Metódu `children()` budeme neskôr ešte vylepšovať. To, že aj táto trieda je abstraktná vidíme po otestovaní:

```
>>> t = BinaryTree()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    t = BinaryTree()
TypeError: Can't instantiate abstract class BinaryTree with abstract methods␣
↪__iter__,
__len__, data, left, parent, right, root
```

Vlastnosti binárnych stromov:

- označme vrcholy stromu s rovnakou hĺbkou `d` ako **úroveň** `d`
- v úrovni 0 je len koreň stromu
- v úrovni `u` je maximálne `2**u` vrcholov
- ak `n` je počet všetkých vrcholov, `l` je počet listov, `h` je výška stromu, tak
  - `h+1 <= n <= 2**(h+1)-1`
  - `1 <= l <= 2**h`
  - `log(n+1)-1 <= h <= n-1`

3.4 Implementovanie binárnych stromov

Najčastejšie sú to tieto dva spôsoby:

- pomocou poľa:
- koreň `pole[0]`, i-ty vrchol má synov `pole[2*i+1]` a `pole[2*i+2]`, ak `pole[i]` je `None` alebo `i >= len(pole)`, taký vrchol v strome neexistuje
- pomocou spájanej štruktúry:
- každý vrchol stromu (trieda `Node`) má okrem atribútu `data` (samotný údaj vo vrchole) aj referencie na ďalšie vrcholy: `parent`, `left` a `right`

Ďalej budeme binárny strom implementovať pomocou spájanej štruktúry, trieda `LinkedBinaryTree` využíva definíciu `BinaryTree`:

```
from bintree import BinaryTree

class LinkedBinaryTree(BinaryTree):

    class Node:
        def __init__(self, data, parent=None, left=None, right=None):
            self._data = data
            self._parent = parent
            self._left = left
            self._right = right

    #---------------------------------
```

```
    def __init__(self):
        self._root = None
        self._size = 0

    def root(self):
        return self._root

    def parent(self, node):
        return node._parent

    def left(self, node):
        return node._left

    def right(self, node):
        return node._right

    def data(self, node):
        return node._data

    def __len__(self):
        return self._size
```

Všimnite si, že všetky atribúty tried `Node` aj `LinkedBinaryTree`, ktoré sú premenné, začínajú podčiarkovníkom. Týmto sa v Pythone zvyknú označovať atribúty, ktoré by sa nemali používať ako verejné (public). Hoci je na programátorovi, ako ich bude používať. Táto trieda je stále ešte abstraktná lebo chýba implementácia metódy `__iter__()`. Túto naprogramujeme neskôr, zatiaľ ju môžeme zapísať ako `return self`.

Ďalej zadefinujeme niekoľko pomocných metód, ktoré budú slúžiť na pridávanie vrcholov do existujúceho stromu na presné miesto:

- `add_root()` vytvorí koreň prázdneho stromu, ak koreň už existoval, metóda vyvolá chybu
- `add_left()` konkrétnemu vrcholu pridá ľavého syna, ak ľavý syn už existoval, metóda vyvolá chybu
- `add_right()` konkrétnemu vrcholu pridá pravého syna, ak pravý syn už existoval, metóda vyvolá chybu
- `add_random()` konkrétnemu vrcholu pridá ľavého alebo pravého syna, metóda sa rozhoduje náhodne
  - ak v náhodne vybranom smere, už príslušný syn existuje, metóda sa presunie na tohto syna a na ňom spustí `add_random()`

Pridáme metódy:

```
class LinkedBinaryTree(BinaryTree):

    ...

    def __iter__(self):
        return self

    def add_root(self, data):
        if self.root() is not None:
            raise ValueError()
        self._root = self.Node(data)
        self._size = 1
        return self._root

    def add_left(self, node, data):
        if self.left(node) is not None:
            raise ValueError()
```

```
        node._left = self.Node(data, node)    # node je pre tento vrchol otcom
        self._size += 1
        return node._left

    def add_right(self, node, data):
        if self.right(node) is not None:
            raise ValueError()
        node._right = self.Node(data, node)
        self._size += 1
        return node._right

    def add_random(self, node, data):
        if random.randrange(2):
            if self.left(node) is None:
                self.add_left(node, data)
            else:
                self.add_random(self.left(node), data)
        else:
            if self.right(node) is None:
                self.add_right(node, data)
            else:
                self.add_random(self.right(node), data)
```

Zložitosť všetkých základných operácií okrem `depth()` a `height()` je **O(1)**. Už vieme, že zložitosť `height()` je **O(n)**, zložitosť `depth()` je **O(h)**, kde `h` je výška stromu.

Vytvorenie binárneho stromu, napr. takto:

```
t = LinkedBinaryTree()
k = t.add_root('koren')
p1 = t.add_left(k, 11)
p2 = t.add_right(k, 22)
t.add_left(p1, 33)
t.add_right(p1, 44)
t.add_left(p2, 55)
t.add_right(p2, 66)
```

alebo

```
t = LinkedBinaryTree()
k = t.add_root('koren')
for i in range(1000):
    t.add_random(k, i)
print('vyska =', t.height())
```

3.5 Prechádzanie vrcholov stromu

Už z prvého ročníka poznáme tieto základné algoritmy na prechádzanie binárneho stromu * **preorder** - najprv koreň, potom postupne všetky podstromy * **postorder** - najprv postupne všetky podstromy, potom koreň * **breadth-first** - do šírky, t.j. po úrovniach: najprv koreň, potom 1. úoveň, potom 2. úroveň, ... * **inorder** - najprv ľavý podstrom, potom koreň a na záver postupne všetky zvyšné podstromy

- väčšinou sa definuje iba pre binárne stromy

3.5.1 Preorder

Potrebujeme pomocnú rekurzívnu funkciu:

```
class LinkedBinaryTree(BinaryTree):

    ...

    def preorder(self):
        if not self.is_empty():
            self.subtree_preorder(self.root())
        print()

    def subtree_preorder(self, node):
        print(self.data(node), end=' ')
        for n in self.children(node):
            self.subtree_preorder(n)
```

To isté, ale pomocná funkcia je vnorená priamo do metódy `preorder`:

```
class LinkedBinaryTree(BinaryTree):

    ...

    def preorder(self):

        def subtree_preorder(node):
            print(self.data(node), end=' ')
            for n in self.children(node):
                subtree_preorder(n)

        if not self.is_empty():
            subtree_preorder(self.root())
        print()
```

Vidíme, že metóda `preorder()` nevyužíva nič z toho, že je určená iba pre binárny strom. Môžeme ju teda presunúť do abstraktnej triedy `Tree` a pritom ešte namiesto výpisu hodnôt vo vrcholoch budeme vytvárať pole vrcholov (referencií na vrcholy). Teraz bude použiteľná vo všetkých ďalších implementáciách.

```
class Tree(metaclass=ABCMeta):

    ...

    def preorder(self):

        def subtree_preorder(node):
            res.append(node)
            for n in self.children(node):
                subtree_preorder(n)

        res = []
        if not self.is_empty():
            subtree_preorder(self.root())
        return res
```

3.5.2 Postorder

Postorder zapíšme podobne ako preorder, ktorý vytvára pole vrcholov:

```
class Tree(metaclass=ABCMeta):

    ...

    def postorder(self):

        def subtree_postorder(node):
            for n in self.children(node):
                subtree_postorder(n)
            res.append(node)

        res = []
        if not self.is_empty():
            subtree_postorder(self.root())
        return res
```

Otestujeme: vytvorme testovací strom:

```
t = LinkedBinaryTree()
k = t.add_root('koren')
p1 = t.add_left(k, 11)
p2 = t.add_right(k, 22)
t.add_left(p1, 33)
t.add_right(p1, 44)
t.add_left(p2, 55)
t.add_right(p2, 66)
```

a výpis oboch postupností preorder a postorder:

```
print('preorder = ', end='')
for n in t.preorder():
    print(t.data(n), end=' ')
print()

print('postorder = ', end='')
for n in t.postorder():
    print(t.data(n), end=' ')
print()
```

3.6 Generátory a iterátory

V Pythone existuje zaujímavý spôsob ako generovať postupnosti. Najčastejšie sme to doteraz robili pomocou poľa, napr. generovanie všetkých deliteľov nejakého čísla:

```
def delitele(n):
    res = []
    for i in range(1,n+1):
        if n % i == 0:
            res.append(i)
    return res
```

```
>>> delitele(100)
[1, 2, 4, 5, 10, 20, 25, 50, 100]
```

Pre postupnosti je základnou vlastnosťou to, aby boli **iterovateľné**, t.j. aby sa jeho prvky dali postupne prechádzať pomocou for-cyklu. Napr.

```
>>> for i in delitele(100):
      print(i, end=' ')

1 2 4 5 10 20 25 50 100
```

T.j. nie je dôležité mať všetky prvky k dispozícii naraz v nejakej dátovej štruktúre, ale dôležité je ich postupne získavať vždy keď potrebujeme získať ďalší (teda vlastne niečo ako `__iter__()` a `__next__()`). Na toto využijeme nový mechanizmus **generátorov**. Tieto sa podobajú na bežné funkcie ale namiesto `return` používajú príkaz `yield`. Generátory fungujú na takomto princípe:

- keď takúto **generátorovú funkciu** zavoláme, nevytvorí sa ešte žiadna hodnota, ale vytvorí sa **generátorový objekt**
- keď si od **generátorového objektu** teraz vypýtame jednu hodnotu, dozvieme sa prvú z nich (slúži na to štandardná funkcia `next()`)
- každé ďalšie vypýtanie hodnoty (funkcia `next()`) nám odovzdá ďalšiu hodnotu postupnosti
- keď už **generátorový objekt** nemá ďalšiu hodnotu, tak volanie funkcie `next()` vyvolá chybovú správu `StopIteration`

Samotná **generátorová funkcia** pri výskyte príkazu `yield` nekončí "len" odovzdá jednu z hodnôt postupnosti a pokračuje ďalej. Funkcia končí až na príkaze `return` (alebo na konci funkcie) a vtedy automaticky vygeneruje chybu (exception) **StopIteration**. Samotné **odovzdanie hodnoty** (príkazom `yield`) preruší vykonávanie generátorovej funkcie s tým, že sa presne zapamätá miesto, kde sa bude pokračovať aj s momentálnym menným priestorom. Volanie `next()` pokračuje na tomto mieste, aby odovzdal ďalšiu hodnotu.

Zadefinujme funkciu `delitele()` ako generátor:

```
def delitele(n):
    for i in range(1,n+1):
        if n % i == 0:
            yield i
```

vytvoríme generátorový objekt:

```
>>> d = delitele(15)
```

premenná `d` je naozaj generátorový objekt, ktorý zatiaľ nevygeneroval žiadny prvok postupnosti:

```
>>> d
<generator object delitele at 0x0000000003042828>
```

keď chceme prvý prvok, zavoláme metódu `next()` rovnako ako pri iterátoroch:

```
>>> next(d)
1
```

každé ďalšie volanie `next()` vygeneruje ďalšie prvky:

```
>>> next(d)
3
>>> next(d)
```

```
5
>>> next(d)
15
>>> next(d)
...
StopIteration
```

Po poslednom prvku funkcia `next()` vyvolala výnimku `StopIteration`. Mohli by sme to zapísať aj pomocou for-cyklu:

```
>>> for i in delitele(15):
        print(i, end=' ')

1 3 5 15
```

3.6.1 Ukážky generátorových funkcií

- postupnosť piatich hodnôt:

```
def prvo():
    yield 2
    yield 3
    yield 5
    yield 7
    yield 11


>>> list(prvo())
[2, 3, 5, 7, 11]
```

- to isté pomocou for-cyklu:

```
def prvo():
    for i in [2, 3, 5, 7, 11]:
        yield i

>>> list(prvo())
[2, 3, 5, 7, 11]
```

For-cyklus v generátorových funkciách môžeme skrátene zapísať aj pomocou verzie **yield from**:

- to isté ako predchádzajúca verzia:

```
def prvo():
    yield from [2, 3, 5, 7, 11]

>>> list(prvo())
[2, 3, 5, 7, 11]
```

Parametrom `yield from` môže byť ľubovoľný iterovateľný objekt nielen pole, napr. aj `range()` alebo aj iný generátorový objekt (napr. v rekurzívnych funkciách).

- využitie range():

```
def test(n):
     yield from range(n+1)
     yield from range(n-1,-1,-1)
```

```
>>> list(test(3))
[0, 1, 2, 3, 2, 1, 0]
>>> list(test(5))
[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]
```

- skoro to isté ale rekurzívne:

```
def urob(n):
    if n < 1:
        yield 0
    else:
        yield n
        yield from urob(n-1)
        yield n

>>> list(urob(3))
[3, 2, 1, 0, 1, 2, 3]
>>> list(urob(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]
```

- fibonacciho postupnosť:

```
def fib(n):
    a,b = -1,1
    while n > 0:
        a,b = b,a+b
        yield b
        n -= 1

>>> list(fib(10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> for i in fib(10):
        print(i, end=' ')

0 1 1 2 3 5 8 13 21 34
```

ak by sme chceli z fibonacciho postupnosti vypísať len po prvý člen, ktorý je aspoň 10000 a my nevieme odhadnúť, koľko ich budeme potrebovať, zapíšeme napr.

```
>>> for i in fib(10000):
        print(i, end=' ')
        if i > 10000:
            break

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
```

vďaka tomu, že `fib()` je generátor a nie funkcia, ktorá vytvára pole hodnôt, nebolo pre tento for-cyklus potrebné vyrobiť 10000 prvkov, ale len toľko, koľko ich bolo treba v cykle.

Generátorovým funkciám se niekedy hovorí **lenivé vyhodnocovanie** (**lazy evaluation**), lebo funkcia počíta ďalšiu hodnotu až keď je o ňu požiadaná (pomocou `next()`) - teda nič nepočíta zbytočne dopredu.

3.6.2 Generované zoznamy

už sme sa dávnejšie stretli so zápismi:

```
>>> pole = [i for i in range(20) if i%7 in [2,3,5]]
>>> pole
[2, 3, 5, 9, 10, 12, 16, 17, 19]
>>> mn = {i for i in range(20) if i%7 in [2,3,5]}
>>> mn
{2, 3, 5, 9, 10, 12, 16, 17, 19}
>>> ntica = tuple(i for i in range(20) if i%7 in [2,3,5])
>>> ntica
(2, 3, 5, 9, 10, 12, 16, 17, 19)
```

Podobne vieme vygenerovať nielen pole (`list`), množinu (`set`) a nticu (`tuple`), ale aj slovník (`dict`). Hovoríme tomu **list comprehension** (resp. iný typ) - po slovensky **generované zoznamy**. Všimnite si, že nticu musíme generovať pomocou funkcie `tuple()`, lebo inak:

```
>>> urob = (i for i in range(20) if i%7 in [2,3,5])
>>> urob
<generator object <genexpr> at 0x022A6760>
>>> list(urob)
[2, 3, 5, 9, 10, 12, 16, 17, 19]
```

dostávame **generátorový objekt** úplne rovnaký ako napr.

```
def gg():
    for i in range(20):
        if i%7 in [2,3,5]:
            yield i

>>> urob = gg()
>>> urob
<generator object gg at 0x022A6828>
>>> list(urob)
[2, 3, 5, 9, 10, 12, 16, 17, 19]
```

Takže jednoduché generátorové objekty môžeme vytvárať aj takto zjednodušene:

```
def gg(*pole):
    return (i for i in range(20) if i%7 in pole)

>>> urob = gg([2,3,5])
>>> urob
<generator object <genexpr> at 0x0229E8A0>
>>> list(urob)
[2, 3, 5, 9, 10, 12, 16, 17, 19]
```

3.7 Generátory pri stromoch

Najprv zmeňme metódu `children()` na gererátorovú funkciu:

```
class BinaryTree(Tree):

    ...

    def children(self, node):
        if self.left(node) is not None:
            yield self.left(node)
```

```
        if self.right(node) is not None:
            yield self.right(node)
```

Teraz metódy, ktoré prechádzajú všetky vrcholy v nejakom poradí, teda `preorder()` a `postorder()`:

```
class Tree(metaclass=ABCMeta):

    ...

    def preorder(self):

        def subtree_preorder(node):
            yield node
            for n in self.children(node):
                yield from subtree_preorder(n)

        if not self.is_empty():
            yield from subtree_preorder(self.root())

    def postorder(self):

        def subtree_postorder(node):
            for n in self.children(node):
                yield from subtree_postorder(n)
            yield node

        if not self.is_empty():
            yield from subtree_postorder(self.root())
```

3.8 Iterovanie stromu

V niektorých algoritmoch potrebujeme prechádzať všetky vrcholy stromu, ale je nám jedno v akom poradí (napr. ako sme to použili v metóde `height1()`). Vtedy využijeme takýto zápis:

```
for vrchol in strom:
    'spracuj vrchol'
```

takéto správanie funguje vďaka metóde `__iter__()`. Predchádzajúci zápis je vlastne:

```
for vrchol in strom.__iter__():
    'spracuj vrchol'
```

Od metódy sa očakáva, že bude generátorom. Keďže my už nejaké generátory hotové máme, môžeme jeden z nich využiť:

```
class Tree(metaclass=ABCMeta):

    ...

    def __iter__(self):
        yield from self.preorder()
```

v tomto konkrétnom prípade bude fungovať aj:

```
class Tree(metaclass=ABCMeta):

    ...

    def __iter__(self):
        return self.preorder()
```

3.9 Cvičenie

1*. Na 3. prednáške sa postupne definovali triedy `Tree`, `BinaryTree` a `LinkedBinaryTree`. Zapíšte ich postupne do a otestujte funkčnosť vygenerovaním náhodného stromu s 20 vrcholmi a vygenerovaním `preorder()` a `postorder()` (tieto dve netódy sú generátory definované v abstraktnej triede `Tree`).

- v súbore `tree.py` (nezabudnite na metódy `depth()`, `height()` a generátorové verzie `prefix()` a `postfix()`):

```
from abc import ...

class Tree(...):
    ...
```

- v súbore `bintree.py` (nezabudnite na generátorovú verziu metódy `children()`):

```
from abc import ...
from tree import ...

class BinaryTree(...):
    ...
```

- v súbore `linkbintree.py` (nezabudnite na metódy `add_root()`, metódy `add_left()`, ...):

```
from bintree import ...

class LinkedBinaryTree(...):
    ...
```

2*. Zadefinujte ďalšie metódy ako generátory:

- do triedy `BinaryTree` metódu `inorder()`:

```
class BinaryTree(Tree):

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

- do triedy `Tree` metódu `breadth_first()` (algoritmus do šírky):

```
class Tree(...):

    def breadth_first(self):
        ...
```

  algoritmus "do šírky" generuje vrcholy po úrovniach: najpr. koreň, potom jeho synovia, potom jeho vnuci, ... (zrejme využijete nejaký svoj **queue**), napr.

```
algoritmus breadthfirst(T):
    inicializuj queue Q s hodnotou T.root()
    while Q not empty:
        p = Q.dequeue()
        spracuj vrchol p (napr. yield)
        for vrchol in T.children(p):
            Q.enqueue(vrchol)
```

- obe metódy otestujte a porovnajte s výsledkami z `preorder()` a `postorder()`

3. V prvom ročníku sme vedeli nakresliť binárny strom. Doplňte metódu `draw(canvas)` do `BinaryTree`, ktorá vykreslí celý strom.

   - definujte

   ```
   class BinaryTree(...):
       ...

       def draw(self, canvas):
           ...
   ```

   - môžete sa inšpirovať napr. kódom

   ```
   def kresli(strom):
       def kresli1(strom, sirka, x, y):
           if strom is None:
               return
           if strom.left is not None:
               c.create_line(x,y,x-sirka//2,y+50)
               kresli1(strom.left,sirka//2,x-sirka//2,y+50)
           if strom.right is not None:
               c.create_line(x,y,x+sirka//2,y+50)
               kresli1(strom.right,sirka//2,x+sirka//2,y+50)
           c.create_oval(x-10,y-10,x+10,y+10,fill='white')
           c.create_text(x,y,text=strom.data,font='arial 10')

   kresli1(strom,300,300,20)
   ```

   - šírku grafickej plochy vieme zistiť napr. `int(c['width'])`

4. Napíšte metódu `gener()` pre binárny strom, ktorá prejde (vygeneruje) všetky prvky (v poradí `preorder`) ale bez rekurzie a zásobníka

   - môžete použiť "pravidlo pravej ruky", keďže každý vrchol si eviduje aj svojho predka

   ```
   class BinaryTree(...):
       ...

       def gener(self):
           ...
   ```

5. Implementujte triedu `ArrayBinaryTree`, v ktorej na reprezentáciu binárneho stromu využijete pole hodnôt:

   - koreň je v nultom prvku `pole[0]`
   - i-ty prvok (ak existuje) má svojich synov: ľavý v `pole[2*i+1]`, pravý v `pole[2*i+2]`
   - vrchol neexistuje, buď je jeho index >= `len(pole)`, alebo jeho hodnota v poli je `None` (predpokladáme, že hodnota vo vrchole je rôzna od `None`) - v týchto prípadoch metódy napr. `left()`, `parent()`, ... vrátia `None`

- parameter `node` v týchto metódach označuje index do tohto poľa (teda celé číslo)
- naprogramujte všetky metódy:

```
class ArrayBinaryTree(BinaryTree):
    def __init__(self):
        self._pole = []
        self._size = 0

    def root(self):
        ...
```

- vašu implementáciu otestujte (mali by fungovať aj metódy `preorder`, `postorder`, `inorder`, `height`, `depth`)

6. Vytvorte štyri verzie funkcie `mocniny(n)`, ktorá
   - vráti postupnosť (pole) druhých mocnín [1,4,9,16,...,n**2] vytvorenú pomocou for-cyklu a metódy `append()`
   - vráti postupnosť (pole) ale vytvorené pomocou **generátorovej notácie** (napr. `[... for i in ...]`)
   - túto postupnosť vráti ako generátorovú funkciu (použitím `yield`)
   - túto postupnosť vráti ako generátorovú funkciu (použitím generátorového zápisu `(... for i in ...)` bez `yield`)
7. Zapíšte dve verzie funkcie `kopia(pole)`
   - vráti kópiu poľa (výsledok je typu `list`)
   - prvky vráti ako postupnosť vygenerovanú generátorovou funkciou (použitím `yield` alebo `yield from`)
   - otestujte, ako sa bude správať `kopia()`, ak jej parametrom je generátor namiesto poľa

8*. Zapíšte dve verzie funkcie `map(funkcia,pole)`, ktorá vráti prvky poľa prerobené funkciou `funkcia` (podobne ako to robí štandardná funkcia `map`, ale bez tejto funkcie)

- výsledok vytvorte najprv ako pole
- výsledok ako generátor
- otestujte, ako sa bude správať `map()`, ak druhým parametrom je generátor
- porovnajte so štandardnou funkciou `map()`

9. Zapíšte dve verzie funkcie `filter(funkcia,pole)`, ktorá vráti len tie prvky poľa, pre ktoré je splnená logická `funkcia`
   - výsledok najprv ako pole
   - výsledok ako generátor
   - otestujte, ako sa bude správať `filter()`, ak druhým parametrom je generátor
   - porovnajte so štandardnou funkciou `filter()`
10. Zapíšte funkciu `zdvoj(gen)`, ktorá vygeneruje každý prvok 2-krát za sebou - funkcia vráti generátor

- vyskúšajte nielen s parametrom typu generátor, ale napr. aj so poľom alebo stringom

  napr.

```
>>> g = zdvoj((i**2 for i in range(1, 5)))
>>> g
<generator object zdvoj at 0x022A6828>
>>> list(g)
[1, 1, 4, 4, 9, 9, 16, 16]
```

11*. Zapíšte dve verzie funkcie `spoj(gen1,gen2)`, ktorá vygeneruje (vráti ako generátor) najprv všetky prvky `gen1` potom všetky prvky `gen2`

- vyskúšajte nielen s parametrami typu generátor, ale napr. aj s poliami a stringami
- zapíšte verziu funkcie `spoj(*gen)`, v ktorej sa spája ľubovoľne veľa generátorov

```
>>> g = spoj(iter(range(5)), iter(range(10, 0, -2)))
>>> g
<generator object spoj at 0x00A823C0>
>>> print(*g)
0 1 2 3 4 10 8 6 4 2
>>> g = spoj(iter(range(5)), iter('ahoj'), iter(range(10, 0, -2)))
>>> print(*g)
0 1 2 3 4 a h o j 10 8 6 4 2
```

12. Zapíšte tri verzie funkcie `mix(gen1,gen2)`, ktorá generuje prvky na striedačku - ak v jednom skončí skôr, tak už berie len zvyšné druhého

- najprv s pomocným poľom (prvý generátor najprv presype prvky do poľa, a potom počas prechodu druhým generátorom dáva aj prvky z poľa)
- bez pomocného poľa len pomocou štandardnej funkcie `next()`
- porozmýšľajte nad verziou `mix(*gen)`, v ktorej sa mixuje ľubovoľne veľa generátorov

```
>>> print(*mix(iter('PYTHON'), iter(range(4)), iter('ahoj')))
P 0 a Y 1 h T 2 o H 3 j O N
```

Prioritné fronty

operácie nad dátovým typom:

- `p.is_empty()` - či je front prázdny
- `len(p)` - počet prvkov vo fronte
- `p.add(key,value)` - pridaj nový prvok s kľúčom `key` a ľubovoľnou hodnotou `value`
  - všetky kľúče by sa mali dať navzájom porovnávať operáciou "menší" (zabezpečí metóda `__lt__()`)
- `p.min()` - vráti `(key,value)` (teda `tuple`) prvku s najmenším kľúčom
  - bude hlásiť chybu, ak je front prázdny
  - ak je vo fronte viac prvkov s minimálnym kľúčom, vráti jeden z nich
- `p.remove_min()` - vráti `(key,value)` (teda `tuple`) prvku s najmenším kľúčom a zároveň ho z frontu odstráni

Prioritný front ako abstraktná trieda:

```
from abc import ABCMeta, abstractmethod

class Empty(Exception): pass

class PriorityQueue(metaclass=ABCMeta):

    class Item:
        def __init__(self, key, value):
            self.key, self.value = key, value

        def __lt__(self, other):
            return self.key < other.key

        def __repr__(self):
            return str((self.key, self.value))

    @abstractmethod
    def __len__(self):
        '''vrati pocet'''
        pass

    @abstractmethod
    def add(self, key, value):
        '''prida dvojicu (key, value)'''
        pass
```

```
    @abstractmethod
    def min(self):
        '''vrati tuple (key, value)'''
        pass

    @abstractmethod
    def remove_min(self):
        '''vrati a odstrani tuple (key, value)'''
        pass

    def is_empty(self):
        return len(self) == 0
```

4.1 Implementácia pomocou neutriedenej postupnosti

Použijeme pythonovské pole (`list`) - vhodnejší by bol dvojsmerný spájaný zoznam:

```
class UnsortedPriorityQueue(PriorityQueue):

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

    def __len__(self):
        '''vrati pocet'''
        return len(self._data)

    def add(self, key, value):
        '''prida dvojicu (key, value)'''
        self._data.append(self.Item(key, value))

    def _find_min(self):
        '''vrati tuple (key, value)'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        index = 0
        for i in range(1, len(self._data)):
            if self._data[i] < self._data[index]:
                index = i
        return index

    def min(self):
        '''vrati tuple (key, value)'''
        index = self._find_min()
        item = self._data[index]
        return item.key, item.value

    def remove_min(self):
        '''vrati a odstrani tuple (key, value)'''
        index = self._find_min()
        item = self._data.pop(index)
        return item.key, item.value
```

- keďže `min()` hľadá minimálny prvok v neutriedenom poli, musí toto pole prejsť celé, táto operácia teda stojí **O(n)**

- podobne aj `remove_min()` najprv hľadá minimálny prvok (teda **O(n)**) a potom tento prvok vyhodí z poľa metódou `pop`, t.j. ďalších **O(n)**
  - ak by sme toto realizovali spájaným zoznamom, vyhodenie nájdeného prvky by stálo iba **O(1)**, ale aj tak má táto operácia s hľadaním minima zložitosť **O(n)**

operácia	zložitosť
`len()`	O(1)
`is_empty()`	O(1)
`add()`	O(1)*
`min()`	O(n)
`remove_min()`	O(n)

* označuje amortizovanú zložitosť, lebo sa využíva metóda `list.append()`

Môžeme otestovať:

```
p = UnsortedPriorityQueue()
p.add(5,'A')
p.add(9,'B')
p.add(3,'C')
p.add(7,'D')
p.add(4,'E')
p.add(6,'F')
print('pole=', p._data)
print('min=', p.min(), 'pole=', p._data)
print('remove_min=', p.remove_min(), 'pole=', p._data)
print('remove_min=', p.remove_min(), 'pole=', p._data)
print('len=', len(p))
while not p.is_empty():
    print('remove_min=', p.remove_min(), 'pole=', p._data)
print('is_empty=', p.is_empty())
print('remove_min=', p.remove_min(), 'pole=', p._data)
```

dostávame:

```
pole= [(5,'A'), (9,'B'), (3,'C'), (7,'D'), (4,'E'), (6,'F')]
min= (3, 'C') pole= [(5,'A'), (9,'B'), (3,'C'), (7,'D'), (4,'E'), (6,'F')]
remove_min= (3, 'C') pole= [(5,'A'), (9,'B'), (7,'D'), (4,'E'), (6,'F')]
remove_min= (4, 'E') pole= [(5,'A'), (9,'B'), (7,'D'), (6,'F')]
len= 4
remove_min= (5, 'A') pole= [(9,'B'), (7,'D'), (6,'F')]
remove_min= (6, 'F') pole= [(9,'B'), (7,'D')]
remove_min= (7, 'D') pole= [(9,'B')]
remove_min= (9, 'B') pole= []
is_empty= True

Empty: prioritný front je prázdny
```

4.2 Implementácia pomocou utriedenej postupnosti

Opäť použijeme pythonovské pole (`list`) - vhodnejší by bol dvojsmerný spájaný zoznam. Prvky budeme vkladať do poľa tak, aby boli stále utriedené:

```
class SortedPriorityQueue(PriorityQueue):

    def __init__(self):
```

```
        self._data = []

    def __len__(self):
        '''vrati pocet'''
        return len(self._data)

    def add(self, key, value):
        '''prida dvojicu (key, value)'''
        new = self.Item(key, value)
        index = len(self._data)
        while index > 0 and new < self._data[index-1]:
            index -= 1
        self._data.insert(index, new)

    def min(self):
        '''vrati tuple (key, value)'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = self._data[0]
        return item.key, item.value

    def remove_min(self):
        '''vrati a odstrani tuple (key, value)'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = self._data.pop(0)
        return item.key, item.value
```

Naša verzia pre `remove_min()` požíva metódu `pop(0)`, ktorej zložitosť je **O(n)**

- ak by sme namiesto poľa využili spájaný zoznam, táto operácia by mala zložitosť **O(1)**:

operácia	unsorted	sorted	sorted spáj.zoznam
`len()`	O(1)	O(1)	O(1)
`is_empty()`	O(1)	O(1)	O(1)
`add()`	O(1)*	O(n)	O(n)
`min()`	O(n)	O(1)	O(1)
`remove_min()`	O(n)	O(n)	O(1)

Funkčnosť môžete otestovať rovnakým spôsobom, ako pre prvú verziu:

```
pole= [(3,'C'), (4,'E'), (5,'A'), (6,'F'), (7,'D'), (9,'B')]
min= (3, 'C') pole= [(3,'C'), (4,'E'), (5,'A'), (6,'F'), (7,'D'), (9,'B')]
remove_min= (3, 'C') pole= [(4,'E'), (5,'A'), (6,'F'), (7,'D'), (9,'B')]
remove_min= (4, 'E') pole= [(5,'A'), (6,'F'), (7,'D'), (9,'B')]
len= 4
remove_min= (5, 'A') pole= [(6,'F'), (7,'D'), (9,'B')]
remove_min= (6, 'F') pole= [(7,'D'), (9,'B')]
remove_min= (7, 'D') pole= [(9,'B')]
remove_min= (9, 'B') pole= []
is_empty= True

Empty: prioritný front je prázdny
```

4.3 Implementácia pomocou haldy

Využijeme dátovú štruktúru, ktorá má pre nás veľmi užitočné vlastnosti

halda = heap

je binárny strom, ktorý musí spĺňať tieto dve podmienky:

- vzťah medzi vrcholmi: každý vrchol (okrem koreňa) má hodnotu kľúča >= ako hodnota v jeho predkovi (zrejme v koreni je min. hodnota v strome)
- tvar stromu: chceme aby mal minimálnu hĺbku => vlastnosť (skoro) úplného binárneho stromu:
  - okrem najnižšej úrovni sú všetky úrovne úplné
  - v najnižšej úrovni sú všetky vrcholy umiestňované len zľava
- výška takéhoto binárneho stromu je **log n**

Haldu budeme implementovať v poli takto:

- do poľa budeme prvky stromu ukladať po úrovniach:
  - najprv koreň (do 0. prvku)
  - potom dva vrcholy z ďalšej úrovne
  - za tým 4 vrcholy z 2. úrovne, atď.
  - na koniec všetky zvyšné vrcholy z poslednej (možno neúplnej) úrovne
- hĺbka tohto stromu je **log n**
- pre potomkov a predka vrcholu s indexom `i` platí:
  - predok má index `(i-1)//2`
  - ľavý syn má index `2*i+1`
  - pravý syn má index `2*i+2`

Metóda `add()` - vloženie nového prvku:

- aby sme zachovali vlastnosť (skoro) úplného binárneho stromu, pridáme vrchol na koniec (za posledný prvok v poli), teda `pole.append()`
- asi sme tým pokazili pravidlo medzi vrcholom a jeho predkom: haldu treba upratať - budeme prebublávať v halde od tohto prvku nahor smerom ku koreňu:
- ak je práve pridaný prvok menší ako jeho predok (s indexom `(i-1)//2`), vymení ho s predkom a znovu kontroluje už túto jeho novú pozíciu s novým predkom - toto opakuje, kým nenatrafí na menšieho alebo sa dostane do koreňa stromu

Metóda `remove_min()` - vyhodenie minimálneho prvku, t.j. koreň haldy

- vyhodený koreň nahradíme posledným prvkom v halde (najpravejší v najnižšej úrovni) a haldu pritom o 1 skrátime
- asi sme týmopäť haldu pokazili - budeme tento prvok kontrolovať s jeho synmi a vymieňať, t.j. budeme prebublávať smerom nadol:
- práve presťahovaný 0. prvok poľa porovnám s oboma jeho synmi (prvky s indexami `2*i+1` a `2*i+2`) a ak je jeden z nich menší ako koreň, tak ho s koreňom vymením a znovu túto novú pozíciu porovnávam s jeho synmi - toto opakuje, kým neprídeme do listu, alebo obaja synovia nemajú memšiu hodnotu

Definícia triedy:

```
class HeapPriorityQueue(PriorityQueue):
    #------------ pomocne funkcie -------------------------------
    def _left(self, index):
        return 2 * index + 1

    def _right(self, index):
        return 2 * index + 2

    def _parent(self, index):
        return (index - 1) // 2

    def _has_left(self, index):
        return self._left(index) < len(self)

    def _has_right(self, index):
        return self._right(index) < len(self)

    def _swap(self, index1, index2):
        self._data[index1], self._data[index2] = self._data[index2], self._
↪data[index1]

    def _heap_up(self, index):
        parent = self._parent(index)
        if index > 0 and self._data[index] < self._data[parent]:
            self._swap(index, parent)
            self._heap_up(parent)              # rekurzia - teraz už s otcom

    def _heap_down(self, index):
        if self._has_left(index):
            left = self._left(index)
            smaller = left
            if self._has_right(index):
                right = self._right(index)
                if self._data[right] < self._data[left]:
                    smaller = right
            if self._data[smaller] < self._data[index]:
                self._swap(index, smaller)
                self._heap_down(smaller)          # rekurzia - teraz už s␣
↪menším synom

    #-------------------------------------------

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

    def __len__(self):
        '''vrati pocet'''
        return len(self._data)

    def add(self, key, value):
        '''prida dvojicu (key, value)'''
        self._data.append(self.Item(key, value))
        self._heap_up(len(self._data) - 1)

    def min(self):
        '''vrati tuple (key, value)'''
        if self.is_empty():
```

```
            raise Empty('priority queue is empty')
        item = self._data[0]
        return item.key, item.value

    def remove_min(self):
        '''vrati a odstrani tuple (key, value)'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        self._swap(0, len(self._data)-1)
        item = self._data.pop()
        self._heap_down(0)
        return item.key, item.value
```

Zložitosť operácií:

- `min()` - je bez cyklu, len vráti prvú hodnotu v poli => **O(1)**
- `add()` - algoritmus pri prebublávaní nahor urobí maximálne toľko porovnávaní, aká je hĺbka stromu, t.j. **O(log n)**
- `remove_min()` - algoritmus pri prebublávaní nadol urobí maximálne toľko porovnávaní, aká je hĺbka stromu, t.j. **O(log n)**
  - ak by sme namiesto `_swap()` a `pole.pop()` robili najprv `pole.pop(0)` a potom vložili posledný prvok na začiatok pomocou `pole.insert(0)`, tak zložitosť `remove_min()` by stúpla na **O(n)**

Zhrňme to do tabuľky:

operácia	unsorted	sorted	sorted spáj.zoznam	heap
`len()`	O(1)	O(1)	O(1)	O(1)
`is_empty()`	O(1)	O(1)	O(1)	O(1)
`add()`	O(1)*	O(n)	O(n)	O(log n)*
`min()`	O(n)	O(1)	O(1)	O(1)
`remove_min()`	O(n)	O(n)	O(1)	O(log n)*

* označuje amortizovanú zložitosť, lebo sa využívajú metódy `list.append()` a `list.pop()`

Po otestovaní:

```
pole= [(3,'C'), (4,'E'), (5,'A'), (9,'B'), (7,'D'), (6,'F')]
min= (3, 'C') pole= [(3,'C'), (4,'E'), (5,'A'), (9,'B'), (7,'D'), (6,'F')]
remove_min= (3, 'C') pole= [(4,'E'), (6,'F'), (5,'A'), (9,'B'), (7,'D')]
remove_min= (4, 'E') pole= [(5,'A'), (6,'F'), (7,'D'), (9,'B')]
len= 4
remove_min= (5, 'A') pole= [(6,'F'), (9,'B'), (7,'D')]
remove_min= (6, 'F') pole= [(7,'D'), (9,'B')]
remove_min= (7, 'D') pole= [(9,'B')]
remove_min= (9, 'B') pole= []
is_empty= True

Empty: prioritný front je prázdny
```

všimnite si, ako sa v poli uchováva halda - po každom vyhodení prvého prvku sa pole preuprace

4.4 Využitie modulu heapq

Štandardný pythonovský modul `heapq` pracuje s haldou - ponúka funkcie:

- `heappush(pole,prvok)` - vloží nový prvok
- `heappop(pole)` - robí to isté ako naše `remove_min()`
- `heapify(pole)` - preusporiada pole tak, aby spĺňalo podmienky haldy - zložitosť tejto funkcie je **O(n)**

cvične vyskúšame, ako sa s tým pracuje:

```
import heapq

class HeapPriorityQueue(PriorityQueue):

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

    def __len__(self):
        '''vrati pocet'''
        return len(self._data)

    def add(self, key, value):
        '''prida dvojicu (key, value)'''
        heapq.heappush(self._data, self.Item(key, value))

    def min(self):
        '''vrati tuple (key, value)'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = self._data[0]
        return item.key, item.value

    def remove_min(self):
        '''vrati a odstrani tuple (key, value)'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = heapq.heappop(self._data)
        return item.key, item.value
```

a po otestovaní:

```
pole= [(3,'C'), (4,'E'), (5,'A'), (9,'B'), (7,'D'), (6,'F')]
min= (3, 'C') pole= [(3,'C'), (4,'E'), (5,'A'), (9,'B'), (7,'D'), (6,'F')]
remove_min= (3, 'C') pole= [(4,'E'), (6,'F'), (5,'A'), (9,'B'), (7,'D')]
remove_min= (4, 'E') pole= [(5,'A'), (6,'F'), (7,'D'), (9,'B')]
len= 4
remove_min= (5, 'A') pole= [(6,'F'), (9,'B'), (7,'D')]
remove_min= (6, 'F') pole= [(7,'D'), (9,'B')]
remove_min= (7, 'D') pole= [(9,'B')]
remove_min= (9, 'B') pole= []
is_empty= True

Empty: prioritný front je prázdny
```

4.5 Triedenie pomocou prioritného frontu

- pomocou neutriedenej postupnosti ==> **min-sort** (selection sort)
- pomocou utriedenej postupnosti ==> **insert-sort**

Funkcia `sort()` nevracia žiadnu hodnotu ale triedi priamo prvky poľa:

```
def sort(pole):
    p = UnsortedPriorityQueue()              # alebo SortedPriorityQueue()
    for i in pole:
        p.add(i, i)                          # kľúč bude rovnaký ako hodnota
    for i in range(len(pole)):
        pole[i] = p.remove_min()[1]          # z tuple (key,value) zoberieme␣
↪value
```

vidíme, že oba sorty majú zložitosť **O(n**2)**

4.6 Triedenie pomocou prioritného frontu s haldou = heap sort

Funkcia triedi priamo prvky poľa:

```
def heap_sort(pole):
    p = HeapPriorityQueue()
    for i in pole:                           # vytváranie haldy, zložitosť O(n␣
↪log n)
        p.add(i, i)
    for i in range(len(pole)):               # vyberanie z haldy, zložitosť O(n␣
↪log n)
        pole[i] = p.remove_min()[1]
```

Otestujeme:

```
import time, random

for n in 1000,10000,100000:
    pole = [random.randrange(10000) for i in range(n)]
    pole0 = pole[:]

    start = time.time()
    heap_sort(pole)
    end = time.time()
    print('{:<8} {:<10.6f}'.format(n, end-start), pole==sorted(pole0))
```

výsledky:

```
1000     0.067004   True
10000    0.836048   True
100000   10.709612  True
1000000  135.977778 True
```

všimnite si, ako sa mení čas, keď program spúšťame s 10-krát väčším poľom - malo by byť vidieť, že zložitosť algoritmu je **O(n log n)**

4.7 Cvičenie

1*. Zoberte z prednášky `SortedPriorityQueue` a otestujte toto triedenie:

- triedenie prioritným frontom:

```
def sort(pole):
    p = SortedPriorityQueue()
    for i in pole:
        p.add(i, i)
    for i in range(len(pole)):
        pole[i] = p.remove_min()[1]
```

- postupne vygenerujte náhodné `n`-prvkové pole pre `n`: 1000, 2000, 4000, 8000, 16000, 32000, spustite toto triedenie a vypíšte čas jeho trvania, zároveň skontrolujte, či je výsledné pole správne utriedené.

2. Implementujte prioritný front, v ktorom sa prvky uchovávajú utriedené, pomocou dvojsmerného (alebo jednosmerného) spájaného zoznamu

   - použite deklarácie

```
class SortedPriorityQueue(PriorityQueue):
    class Item:
        ``pomocna trieda pre vrchol spajaneho zoznamu``
        def __init__(self, key, value, next=None, prev=None):
            self.key = key
            self.value = value
            self.next = next
            self.prev = prev

        def __lt__(self, other):
            return self.key < other.key

        def __repr__(self):
            return str((self.key, self.value))

    def __init__(self):
        self.data = None       # zaciatok spajaneho zoznamu
    ...
```

   - zapíšte to tak, aby sa zachovala zložitosť operácií podľa tejto tabuľky:

operácia	zložitosť
`len`	O(1)
`is_empty`	O(1)
`add`	O(n)
`min`	O(1)
`remove_min`	O(1)

3*. Predpokladajte, že na vstupe máme tieto hodnoty:

- vstupné pole

```
8, 13, 7, 10, 5, 15, 12, 17, 9, 14, 4, 11, 18, 16, 6
```

- ručne vytvorte z tohto poľa **haldu** (postupne pridávate najprv do prázdneho frontu akokeby metódou `add()`)
- potom skontrolujte s haldou vytvorenou pomocou:

```
p = HeapPriorityQueue()
for i in pole:
    p.add(i, i)
print(p._data)
```

- priebeh ukladania do haldy si môžete vizualizovať napr. na stránke Heap Visualization (https://www.cs.usfca.edu/~galles/visualization/Heap.html)

4*. V prednáške sme ukázali `heapsort` zapísaný pomocou prioritného frontu s haldou:

- triedenie haldou:

```
def heap_sort(pole):
    p = HeapPriorityQueue()
    for i in pole:
        p.add(i, i)
    for i in range(len(pole)):
        pole[i] = p.remove_min()[1]
```

- prvá časť algoritmu, ktorá ukladá prvky vstupného poľa do haldy (pomocou metódy `add()`) má zložitosť **O(n log n)**, pričom sa táto fáza dá dosiahnuť ešte lepšou zložitosťou **O(n)**; použite tento postup:
  - prvky sa do haldy nepridávajú po jednom (pomocou `p.add(...)`, pričom by sa halda zakaždým upratala pomocou `_heap_up()`), ale sa pridajú naraz všetky prvku poľa a halda sa upratuje až na záver volaním metódy `_heap_down()` pre všetky prvky poľa
  - do triedy `HeapPriorityQueue()` pridajte metódu `heapify()` (podobná existuje aj v module `heapq`), ktorá dostane ako parameter celé vstupné pole hodnôt, prekopíruje si ho do svojho poľa s haldou `p._data` (kde kľúč aj hodnota sú rovnaké) a potom spustí algoritmus upratovania haldy:
  - postupne prechádza pole **od konca** a pre každý `i`-ty prvok zabezpečí, aby podstrom s koreňom `i` sa stal haldou (použijeme `_heap_down(i)`): zrejme na začiatku to budú malé binárne stromy a čím ideme vyššie (blížime sa ku koreňu), budú aj tie binárne stromy - haldy väčšie a väčšie; na záver sa uhalduje celý strom
  - uvedomte si, že nemusíme začínať od vrcholov stromu, ktoré sú listami, stačí začínať s vrcholmi, ktoré majú aspoň jedného syna: teda začnete od vrcholu `n//2`
- zamyslite sa, či by ste vedeli dokázať, že tento algoritmus má zložitosť **O(n)**
- na stránke Heap Visualization (https://www.cs.usfca.edu/~galles/visualization/Heap.html) môžete vidieť aj algoritmus heapify - **BuildHeap**

5. Inšpirujte sa algoritmami `_heap_up()` a `_heap_down()` a zapíšte triedenie **heap_sort(pole)** tak, aby sa nepoužívali žiadne ďalšie polia ani iné štruktúry. Pole sa má utriediť na mieste (**in place**). Nebudete si pritom pamätať dvojicu `(key,value)`, ale prvky poľa budú kľúčami, ktoré priamo triedite. Haldu treba konštruovať tak, že v koreni bude **maximálny** prvok a všetci synovia nebudú väčší ako ich otcovia.
   - zadefinujte:

```
def heap_sort(pole):
    def heap_down(i):
        # porovná pole[i] s oboma synmi a ak treba, vymení ho s väčším z nich
        # toto opakuje, kým treba - rekurziu nahrad'te while-cyklom
        ...

    # vytvorí sa halda priamo v samotnom poli:
    # for i in range(len(pole)//2, -1, -1):
    #     heap_down(i)

    # postupne vyberá prvok pole[0] a prest'ahuje ho na koniec pol'a (vymení
↪ich navzájom) a uprace haldu:
    # for n in range(len(pole)-1, 0, -1):
    #     swap(0, n)
    #     heap_down(0)
    ...
```

KAPITOLA 5

Asociatívne polia I.

Už by sme mali mať v Pythone dosť dobré skúsenosti so štandardným typom `dict`, t.j. asociatívnym poľom, resp. slovníkom. Typickou úlohou, ktorá sa zvykne riešiť je frekvenčná tabuľka: máme dané náhodne vygenerované celočíselné pole a úlohou je zistiť najčastejšie sa vyskytujúce hodnoty. Zapíšeme:

```
import random
pole = [random.randrange(1000) for i in range(10000)]
d = {}     # prazdne asociativne pole
for prvok in pole:
    d[prvok] = d.get(prvok, 0) + 1

pole1 = [(v, k) for k,v in d.items()]
print(sorted(pole1, reverse=True)[:10])
```

Zaujímavé je aj to, že to úplne rovnako funguje aj pre text: vieme vypísať najčastejšie vyskytujúce sa slová:

```
import random
pole = 'mama ma emu a ema ma mamu a mama emy ma mamu mamy'.split()
d = {}     # prazdne asociativne pole
for prvok in pole:
    d[prvok] = d.get(prvok, 0) + 1

pole1 = [(v, k) for k,v in d.items()]
print(sorted(pole1, reverse=True)[:5])
```

Pre asociatívne pole v tomto prípade potrebujeme, aby zvládalo tieto operácie:

- vybrať hodnotu z poľa (buď `__getitem__()` alebo `get()`)
- priradiť do prvku poľa (`__setitem__()`)
- iterovať cez všetky kľúče alebo všetky dvojice (kľúč, hodnota) pomocou iterátora kľúčov `__iter__()` alebo generátora `items()`

5.1 Abstraktný dátový typ

Tento typ (hovorí sa mu aj **dictionary** alebo **map**) je rozšírený aj v mnohých iných programovacích jazykoch, preto navrhneme minimálnu množinu operácií, ktoré tento typ charakterizujú a pomocou ktorých sa dajú definovať aj ďalšie operácie nezávisle od konkrétnej realizácie. Asociatívne pole je vlastne nejaká neusporiadaná kolekcia dvojích (kľúč, hodnota), v ktorej **kľúč** slúži na identifikáciu príslušnej hodnoty - pomocou neho máme prístup k hodnote: vieme ju zistiť, zmeniť alebo zrušiť.

Operácie abstraktného dátového typu sú (kde `d` označuje asociatívne pole):

- **get** získanie príslušnej (asociovanej, namapovanej) hodnoty k danému kľúču,
  - v Pythone tomu zodpovedá magická metóda `d.__getitem__(key)`, čo môžeme zapísať aj ako `d[key]`
  - operácia spôsobí vyvolanie chyby `KeyError` v prípade, že sa daný kľúč v poli nenachádza
- **set** zmena alebo priradenie príslušnej (asociovanej, namapovanej) hodnoty k danému kľúču,
  - v Pythone tomu zodpovedá magická metóda `d.__setitem__(key,value)`, čo môžeme zapísať aj ako `d[key] = value`
- **del** zrušenie (asociovanej, namapovanej) hodnoty k danému kľúču,
  - v Pythone tomu zodpovedá magická metóda `d.__delitem__(key)`, čo môžeme zapísať aj ako `del d[key]`
  - operácia spôsobí vyvolanie chyby `KeyError` v prípade, že sa daný kľúč v poli nenachádza
- **len** získanie počtu dvojíc (kľúč, hodnota) v asociovanom poli,
  - v Pythone tomu zodpovedá magická metóda `d.__len__()`, čo môžeme zapísať aj ako `len(d)`
- **iter** postupné získanie (iterovanie) všetkých kľúčov v asociovanom poli,
  - v Pythone tomu zodpovedá magická metóda `d.__iter__()`, čo môžeme zapísať aj ako `iter(d)` alebo použiť ako `for key in d:`

Navrhneme abstraknú dátovú triedu, pričom dvojice (kľúč, hodnota) budeme reprezentovať pomocou atribútov `_key` a `_value` pomocnej triedy `_Item`:

```
from abc import ABCMeta, abstractmethod

class MapBase(metaclass=ABCMeta):

    class _Item:
        def __init__(self, key, value):
            self._key, self._value = key, value

        def __repr__(self):
            return repr(self._key) + ':' + repr(self._value)

    #------------------

    @abstractmethod
    def __getitem__(self, key):
        pass

    @abstractmethod
    def __setitem__(self, key, value):
        pass

    @abstractmethod
    def __delitem__(self, key):
        pass

    @abstractmethod
    def __len__(self):
        pass

    @abstractmethod
```

```
    def __iter__(self):
        pass
```

Zrejme táto trieda sa zatiaľ testovať nedá, musíme najprv implementovať nejakú konkrétnu realizáciu.

5.2 Realizácia neutriedeným poľom

Táto najjednoduchšia realizácia slúži len na demonštráciu funkčnosti asociatívneho poľa. Základom je pythonovské pole (`list`), do ktorého postupne ukladáme prichádzajúce dvojice (kľúč, hodnota). Operácia **get** zrejme musí prejsť všetky dvojice, pole dvojic, teda prvky poľa sú dvojice `_Item`

```
class UnsortedMap(MapBase):
    def __init__(self):
        self._table = []

    def __getitem__(self, key):
        for item in self._table:
            if key == item._key:
                return item._value
        raise KeyError

    def __setitem__(self, key, value):
        for item in self._table:
            if key == item._key:
                item._value = value
                return
        self._table.append(self._Item(key, value))

    def __delitem__(self, key):
        for ix in range(len(self._table)):
            if key == self._table[ix]._key:
                self._table.pop(ix)
                return
        raise KeyError

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

    def __iter__(self):
        for item in self._table:
            yield item._key
```

V atribúte `_table` (typu `list`) uchovávame všetky dvojice (kľúč, hodnota) v tom poradí, ako prichádzali. Všetky tri operácie **get**, **set** aj **del** musia najprv postupne prejsť celé doterajšie pole, aby zistili kde sa nachádza dvojica so zadaným kľúčom. Zrejme preto ich zložitosť je **O(n)**, kde `n` je počet uložených dvojíc - teda veľkosť vnútorného poľa `_table`:

operácia	zložitosť
get	**O(n)**
set	**O(n)**
del	**O(n)**
len	**O(1)**

Funkčnosť môžeme otestovať tak, že vygenerujeme nejaké celočíslené náhodné pole, vyrobíme z neho frekvenčnú tabuľku pomocou `UnsortedMap` aj pomocou štandardného `dict` a obe tieto tabuľky potom porovnáme, či obsahujú rovnaké dvojice:

```
dic1 = UnsortedMap()
pole = [random.randrange(100) for i in range(100)]
dic2 = {}
for i in pole:
    try:
        dic1[i] = dic1[i]+1
    except KeyError:
        dic1[i] = 1
    dic2[i] = dic2.get(i, 0) + 1
set1 = {(k, dic1[k]) for k in dic1}
set2 = {(k, dic2[k]) for k in dic2}
print(set1==set2)
```

Konštrukciu `try - except` sme museli použiť preto, lebo náš abstraktný typ neobsahuje metódu `get()`. Ak by sme do `BaseMap` dodefinovali:

```
class MapBase(metaclass=ABCMeta):
    ...

    def get(self, key, default=None):
        try:
            return self[key]
        except KeyError:
            return default
```

tak môžeme odteraz používať metódu `get()` aj vo všetkých ďalších odvodených triedach od `MapBase`, napr.

```
dic1 = UnsortedMap()
pole = [random.randrange(100) for i in range(100)]
dic2 = {}
for i in pole:
    dic1[i] = dic1.get(i, 0) + 1
    dic2[i] = dic2.get(i, 0) + 1
set1 = {(k, dic1[k]) for k in dic1}
set2 = {(k, dic2[k]) for k in dic2}
print(set1==set2)
```

Úplne rovnako by to fungovalo aj pre pole slov (kľúčmi sú znakové reťazce), napr. pre

```
pole = 'mama ma emu a ema ma mamu a mama emy ma mamu mamy'.split()
...
```

Ešte je tu jedna dôležitá vlastnosť tejto implementácie: typy kľúčov môžeme ľubovoľne miešať, napr.

```
>>> d = UnsortedMap()
>>> d['ab'] = 1
>>> d[3.14] = 2
>>> d[1234] = 3
>>> d._table
['ab':1, 3.14:2, 1234:3]
```

5.3 Realizácia utriedeným poľom

Pozrime, ako sa urýchli táto realizácia oproti predchádzajúcej:

```
class SortedMap(MapBase):

    def _find_index(self, key, low, high):     # vrati bud index, alebo kam␣
↪vlozit
        if high < low:
            return high + 1
        else:
            mid = (low + high) // 2
            if key == self._table[mid]._key:
                return mid
            elif key < self._table[mid]._key:
                return self._find_index(key, low, mid - 1)
            else:
                return self._find_index(key, mid + 1, high)

    #-----------------------------

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

    def __getitem__(self, key):
        ix = self._find_index(key, 0, len(self._table) - 1)
        if ix == len(self._table) or self._table[ix]._key != key:
            raise KeyError
        return self._table[ix]._value

    def __setitem__(self, key, value):
        ix = self._find_index(key, 0, len(self._table) - 1)
        if ix < len(self._table) and self._table[ix]._key == key:
            self._table[ix]._value = value
        else:
            self._table.insert(ix, self._Item(key, value))

    def __delitem__(self, key):
        ix = self._find_index(key, 0, len(self._table) - 1)
        if ix == len(self._table) or self._table[ix]._key != key:
            raise KeyError
        self._table.pop(ix)

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

    def __iter__(self):
        for item in self._table:
            yield item._key
```

Pridali sme pomocnú metódu `_find_index()`, ktorá pre každú zo základných metód pomôže nájsť pozíciu prvku s daným kľúčom. Keďže pole, v ktorom sa hľadá je utriedené, môžeme použiť binárne vyhľadávanie: najprv sa pozrie do strdeu poľa a podľa toho či je hľadaný kľúč menší alebo väčší ako prvok v strede, ďalšie hľadanie pokračuje len v príslušnej polovici poľa. Takto maximálne po **log n** dotazov v stále menších a menších poloviciach nájde hľadaný prvok, resp. rozhodne, že tam taký nie je. Takže napr. metóda `__getitem__()` vráti hľadanú hodnotu so zložitosťou **O(log n)**.

Takéto rýchle hľadanie funguje aj pre `__setitem__()` a `__delitem__()`. Lenže `__setitem__()` pre novú hodnotu, ktorá ešte v poli nebola použije metódu `list.insert(index,hodnota)`, ktorej zložitosť je **O(n)**. Podobne aj `__delitem__()` pri vyhadzovaní prvku použije `list.pop(index)`, ktorej zložitosť je opäť **O(n)**. Zhrňme to do tabuľky:

operácia	unsorted	sorted
get	**O(n)**	**O(log n)**
set	**O(n)**	**O(n) O(log n)**
del	**O(n)**	**O(n)**
len	**O(1)**	**O(1)**

Operácia `__setitem__()` má väčšinou zložitosť **O(n)**, ale v prípade, ža daný kľúč sa už v poli nachádza, jej zložitosť je **O(log n)**.

Keď to otestujete, môžete zistiť, že je to o trochu rýchlejšie ako realizácia pomocou neutriedeného poľa. Funguje aj frekvenčná tabuľka čísel aj slov.

Lenže, teraz sa kľúče v asociatívnom poli navzájom porovnávajú, aby sa v poli dalo vyhľadávať aj zaraďovať na správne miesto. Python nepodporuje porovnávanie rôznych typov, preto neprejde ani:

```
>>> d = SortedMap()
>>> d['ab'] = 1
>>> d[1234] = 2
...
TypeError: unorderable types: int() < str()
```

5.4 Realizácia, pri ktorej sú kľúče indexmi do poľa

Ak obmedzíme všeobecnosť kľúča na nejaký interval celých čísel, naše riešenie sa výrazne zjednoduší a urýchli. Otestujme to takouto realizáciou (nie je to plnohodnotné asociatívne pole):

```
class TestMap(MapBase):
    def __init__(self, capacity=10000):
        self._table = [None] * capacity
        self._num = 0

    def __getitem__(self, key):
        if key < 0 or key >= len(self._table) or self._table[key] is None:
            raise KeyError
        return self._table[key]._value

    def __setitem__(self, key, value):
        if key < 0 or key >= len(self._table):
            raise KeyError
        if self._table[key] is None:
            self._num += 1
        self._table[key] = self._Item(key, value)

    def __delitem__(self, key):
        if key < 0 or key >= len(self._table) or self._table[key] is None:
            raise KeyError
        self._table[key] = None
        self._num -= 1

    def __len__(self):
        return self._num

    def __iter__(self):
        for item in self._table:
            if item is not None:
                yield item._key
```

Ako to funguje:

- na začiatku máme vyhradené pole nejakej danej veľkosti (môžeme určiť parametrom `capacity`) a zatiaľ sú všetky jeho prvky `None`
- keď budeme pridávať novú dvojicu, samotný kľúč je indexom: stačí zistiť, či tam už nejaká dvojica je a ak ešte nie je, pridať ju na správne miesto
- výber hodnoty metódou `__getitem__()` najprv skontroluje povolený interval kľúča a tiež, či príslušná hodnota nie je `None`
- všimnite si, že sme museli trochu meniť aj `__iter__()` nakoľko pole obsahuje aj "prázdne" prvky

Zložitosť metód:

operácia	unsorted	sorted	test
get	**O(n)**	**O(log n)**	**O(1)**
set	**O(n)**	**O(n) O(log n)**	**O(1)**
del	**O(n)**	**O(n)**	**O(1)**
len	**O(1)**	**O(1)**	**O(1)**

Otestujme:

```
import random

dic1 = TestMap(1000000)
pole = [random.randrange(1000000) for i in range(50000)]
dic2 = {}
for i in pole:
    dic1[i] = dic1.get(i, 0) + 1
    dic2[i] = dic2.get(i, 0) + 1
set1 = {(k, dic1[k]) for k in dic1}
set2 = {(k, dic2[k]) for k in dic2}
print(set1==set2)
```

Samozrejme, že teraz nefunguje iný typ kľúča ako celé číslo z intervalu <0, n-1>, čo je ale dosť veľké obmedzenie.

Ďalej rozšírime túto ideu o to, že indexom nebude priamo kľúč, ale zvyšok po delení kľúča veľkosťou poľa. Vďaka tomuto sa ľubovoľne veľký (celočíselný) kľúč zmestí do náško intervalu <0, n-1> (pre veľkosť poľa `n`).

Dostávame ale ďalší ešte väčší problém: čo v prípade, keď dva rôzne kľúče dostanú rovnaký index do poľa (majú rovnaký zvyšok po delení veľkosťou poľa), teda vznikne **kolízia**? Toto sa dá riešiť viacerými spôsobmi, tu si ukážeme najjednoduchší z nich (neskôr uvidíme aj lepšie spôsoby):

- každý prvok poľa nebude obsahovať iba jedinú dvojicu (kľúč, hodnota), ale skupinu všetkých dvojíc, pre ktorú dostávame rovnaký zvyšok po delení (niekedy tomu hovoríme "vedierkové pole", "bucket array", lebo do jedného vedierka spadnú všetky tieto dvojice)
- skupinu dvojíc môžeme zabezpečiť rôznymi spôsobmi (napr. spájaný zoznam), my použijeme pythonovské pole `list`, do ktorého novú dvojicu budeme pridávať pomocou `list.append()`, hľadať dvojicu s daným kľúčom budeme pomocou for-cyklu a vyhadzovať budeme pomocou `list.pop(index)` - tieto dvojice budeme reťaziť za seba
- ideálne by bolo, keby boli tieto skupiny dvojíc čo najmenšie, teda keby bolo čo najmenej kolízií

Zapíšme túto vylepšenú verziu:

```
class TestMap1(MapBase):
    def __init__(self, capacity=10000):
        self._table = [[] for i in range(capacity)]
        self._num = 0
```

```
    def __getitem__(self, key):
        tab = self._table[key % len(self._table)]
        for item in tab:
            if item._key == key:
                return item._value
        raise KeyError

    def __setitem__(self, key, value):
        tab = self._table[key % len(self._table)]
        for item in tab:
            if item._key == key:
                item._value = value
                return
        # mozno kolizia
        self._num += 1
        tab.append(self._Item(key, value))

    def __delitem__(self, key):
        tab = self._table[key % len(self._table)]
        for ix in range(len(tab)):
            if tab[ix]._key == key:
                tab.pop(ix)
                self._num -= 1
                return
        raise KeyError

    def __len__(self):
        return self._num

    def __iter__(self):
        for tab in self._table:
            for item in tab:
                yield item._key
```

Všimnite si, že tieto skupiny dvojíc sú vlastne malé asociatívne polia realizované neutriedeným poľom (rovnaké ako `UnsortedMap`), takže zložitosť manipulácie s týmito skupinkami je **O(k)**, kde `k` je veľkosť skupiny.

Môžeme odhadnúť zložitosť operácií:

operácia	unsorted	sorted	test	test1
get	**O(n)**	**O(log n)**	**O(1)**	**O(k)**
set	**O(n)**	**O(n) O(log n)**	**O(1)**	**O(k)**
del	**O(n)**	**O(n)**	**O(1)**	**O(k)**
len	**O(1)**	**O(1)**	**O(1)**	**O(1)**

kde `k` pre dobre zvolenú veľkosť celého poľa (kapacitu) môže byť dosť malá konštanta a teda zložitosť sa v tomto prípade blíži k **O(1)**.

Otestujeme (len pre celočíselné kľúče):

```
import random

dic1 = TestMap1(1000)
pole = [random.randrange(100000) for i in range(500)]
dic2 = {}
for i in pole:
    dic1[i] = dic1.get(i, 0) + 1
    dic2[i] = dic2.get(i, 0) + 1
set1 = {(k, dic1[k]) for k in dic1}
```

```
set2 = {(k, dic2[k]) for k in dic2}
print(set1==set2)
```

Pozrime, ako je to s počtom kolízií. Do triedy `TestMap1` pridáme atribút `self._col`:

```
class TestMap1(MapBase):
    def __init__(self, capacity=10000):
        self._table = [[] for i in range(capacity)]
        self._num = 0
        self._col = 0

    ...

    def __setitem__(self, key, value):
        tab = self._table[key % len(self._table)]
        for item in tab:
            if item._key == key:
                item._value = value
                return
        # mozno kolizia
        if tab != []:       # skupinka je uz neprazdna
            self._col += 1
        self._num += 1
        tab.append(self._Item(key, value))

    ...
```

Pozrime, koľko prvkov poľa je neprázdnych (po spustení testu s 500 náhodnými číslami a 1000 prvkovým vedierkovým poľom):

```
>>> dic1._col
104
>>> len([len(tab) for tab in dic1._table if len(tab)>0])
394
>>> sum([len(tab)-1 for tab in dic1._table if len(tab)>1])
104
```

Vyšlo nám 104 kolízií, čo sa dá vypočítať, ak spočítame veľkosť všetkých vedierok zmenšených o 1.

Prípadne, aké veľké skupiny dvojíc (vedierka) sa nám vytvorili:

```
>>> print(*[len(tab) for tab in dic1._table if len(tab)>0])
1 2 2 2 1 1 1 2 1 1 1 1 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 2 1 2 1 1 1 1 1 1 1 2 1 1 2 1 1 1 3 1 2 2 1 1 1 1 1 1 1
1 1 1 1 2 1 1 2 1 1 1 2 1 2 1 2 1 1 1 1 2 1 1 1 1 2 1 1 1 1
2 2 1 1 2 1 2 1 1 1 2 1 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1 1 1
1 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 2 2 2 1 1 2 1
2 4 1 2 1 1 3 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 2 1 1 4 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1
1 1 2 1 2 1 2 1 1 2 1 1 1 1 1 1 2 2 1 1 1 3 1 1 2 1 2 1 1 1
1 1 3 1 1 2 1 1 1 1 2 1 1 1 2 1 2 1 1 3 1 2 2 2 1 1 1 1 2 2
1 1 1 2 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1
1 1 1 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 2 1 1 2 1 2 1 1 1 1 1 1
2 2 3 1 1 1 1 2 1 2 2 1 1 3 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1
1 1 1 1 1 2 1 1 3 1 1 2 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 4 2 1
1 3 1 1
```

Väčšina z nich je s jedinou dvojicou, zopár ich je s dvomi a len výnimočne obsahujú viac dvojíc.

5.5 Realizácia hašovacou tabuľkou

Naše posledné riešenie, v ktorom sme celočíselný kľúč delili veľkosťou poľa by mohlo byť univerzálnym riešením, keby sme dokázali ľubovoľný typ kľúča prepočítať na celé číslo. Použijeme takúto ideu:

- ak by bol kľúč znakový reťazec, vieme ho rozobrať na samostatné znaky, tieto previesť na ich ascii-kódy (funkciou `ord()`) a tieto už celočíselné kódy nejako spočítať
- ak by sme naozaj iba spočítavali ascii-kódy, tak napr. všetky tieto trojznakové reťazce `'abc'`, `'cab'`, `'bbb'`, ... by vytvorili rovnaký súčet a tým pádom by sa všetky dostali do rovnakej skupiny:

```
def kod(kluc):
    vysl = 0
    for znak in kluc:
        vysl = vysl + ord(znak)
    return vysl
```

- z tohto dôvodu sa vymýšľajú dômyselnejšie vzorce, v ktorých bude podstatné aj poradie znakov, napr. pre trojznakový reťazec sa ascii-kód prvého znaku vynásobí 10000, druhého 100 a tretí sa nenásobí, preto kód `'abc'` bude 979899, ale kód `'cab'` bude 999798, ... hoci sú to veľké čísla, tieto sa budú aj tak ešte deliť veľkosťou poľa, kódovacia funkcia môže vyserať napr. takto:

```
def kod(kluc):
    vysl = 0
    for znak in kluc:
        vysl = vysl * 100 + ord(znak)
    return vysl
```

- namiesto konštanty 100 sa zvyknú požívať iné konštanty, ktorými môže počítač násobiť rýchlejšie (napr. 32), prípadne sa môže výsledok ešte orezávať na taký počet bitov, aby bola celočíselná aritmetika čo najrýchlejšia (napr. na 64 bitov)
- ako by sme využili túto ideu celočíselného kódu znakového reťazca aj na iné typy (napr. float, tuple, ...)? Riešenie je veľmi jednoduché: každý typ najprv prevedieme na znakový reťazec pomocou `str()` a ďalej ho spracujeme rovnakým spôsobom

V praxi sa taký spôsob ukladania do poľa dvojíc, v ktorom kľúč prerobíme nejakou dômyselnou funkciou na celé číslo, nazýva **hašovacia tabuľka**, lebo samotná táto funkcia sa nazýva **hašovacia funkcia** (z anglického **hash**). Hovoríme, že kľúč transformujeme hašovacou funkciou na celočíselný index.

Zrejme od kvality tejto hašovacej funkcie bude závisieť kvalita realizácie celého asociatívneho poľa, lebo vďaka nej sa môže výrazne znížiť počet kolízií (indexy do poľa budú čo najlepšie rozptýlené po celom poli).

Zapíšme túto novú verziu realizácie asociatívneho poľa:

```
class HashMap(MapBase):
    def __init__(self, capacity=10000):
        self._table = [[] for i in range(capacity)]
        self._num = 0

    def _hash(self, key):
        res = 0
        for ch in str(key):
            res = res * 32 + ord(ch)
        return res

    def __getitem__(self, key):
        ix = self._hash(key) % len(self._table)
        bucket = self._table[ix]
```

```
        for item in bucket:
            if item._key == key:
                return item._value
        raise KeyError

    def __setitem__(self, key, value):
        ix = self._hash(key) % len(self._table)
        bucket = self._table[ix]
        for item in bucket:
            if item._key == key:
                item._value = value
                return
        self._num += 1
        bucket.append(self._Item(key, value))

    def __delitem__(self, key):
        ix = self._hash(key) % len(self._table)
        bucket = self._table[ix]
        for i in range(len(bucket)):
            if bucket[i]._key == key:
                bucket.pop(i)
                self._num -= 1
                return
        raise KeyError

    def __len__(self):
        return self._num

    def __iter__(self):
        for bucket in self._table:
            for item in bucket:
                yield item._key
```

Všimnite si:

- použili sme pomocnú metódu `_hash()`, ktorá ľubovoľnú hodnotu prevedie na celé číslo (najprv ju prevedie na reťazec a z neho potom postupne poskladá celé číslo)
- aby sem z tohto čísla dostali index do poľa, zistíme zvyšok po delení veľkosťou poľa - takto sa dostávame k príslušnému vedierku (bucket)
- zrejme musíme na začiatku nejako rozumne odhadnúť veľkosť poľa, skúsenosti ukazujú, že by malo byť aspoň dvojnásobne veľké ako sa odhaduje počet kľúčov, inak bude veľmi narastať počet kolízií a budú neúmerne veľké skupiny dvojíc s rovnakým kľúčom

Túto triedu môžete otestovať rovnakými testami ako pre `TestMap1`. Len si uvedomte, že teraz môže byť kľúčom skoro ľubovoľný typ, napr.

```
>>> d = HashMap()
>>> d['abc'] = 111
>>> d[3.141] = 222
>>> d[-9999] = 333
>>> for key in d:
        print(key, d[key])

abc 111
-9999 333
3.141 222
```

5.6 Cvičenie

1*. Otestujte `UnsortedMap` z prednášky:

- najprv skúste čo najväčšie pole náhodných celých čísel, pre ktoré sa frekvenčná tabuľka počíta max 2 sekundy
- potom prečítajte textový súbor, rozoberte ho na slová a vytvorte z nich frekvenčnú tabuľku
- vypíšte 10 najčastejších slov aj s ich počtami výskytov
- na testovanie môžete využiť niektoré z týchto textových súborov (alebo ich častí):
  - slovník anglických slov: `text1.txt`
  - Dobšinského rozprávka: `text2.txt`
  - Sherlock Holmes: `text3.txt`
  - Huckleberry Finn: `text4.txt`

2. Do triedy `BaseMap` doprogramujte metódy, ktoré ale už nebudú abstraktné (budú fungovať pre všetky neskôr odvodené triedy, napr. `UnsortedMap`, `SortedMap`, `HashMap`, bez toho aby sa museli preprogramovať):

- `__repr__()` vráti reťazec v tvare `{key1:value1,key2:value2,key3:value3,...}`
- `__contains__(key)` zistí, či sa daný kľúč nachádza v poli
- `setdefault(key,default=None)` vráti príslušnú hodnotu pre daný kľúč, ale v prípade, že sa v poli nenachádza, priradí `default` a potom aj vráti
- `pop(key,default=None)` vyhodí dvojicu (kľúč, hodnota), pritom ako výsledok funkcie vráti príslušnú hodnotu; ak sa kľúč v poli nenachádza, vyvolá výnimku `KeyError`, alebo ak `default` parameter nie je `None`, vráti túto hodnotu
- `values()` - iterátor, ktorý postupne vygeneruje všetky hodnoty v asociatívnom poli
- `items()` - iterátor, ktorý postupne vygeneruje všetky dvojice (kľúč, hodnota) (ako `tuple`)

3*. Otestujte `HashMap` z prednášky:

- na rôzne veľkých náhodne generovaných celočíselných poliach - merajte časy (nastavte rôzne kapacity poľa aj rôzne veľké množiny kľúčov)
- pre veľký textový súbor (rozobrať na slová)

4*. Experimentujte s hašovacou funkciou `_hash()`, napr. namiesto násobenia číslom **32**, skúste násobiť napr. **2** alebo **4**, prípadne veľkým číslom **256**

- zistite, či sa pri týchto zmenách hašovacej funkcie mení počet kolízií (nepr. pre slová z textového súboru), prípadne či sú vedierka (bucket) dobre rozložené (najlepšie by bolo, keby bola v každom vedierku len jedna dvojica)

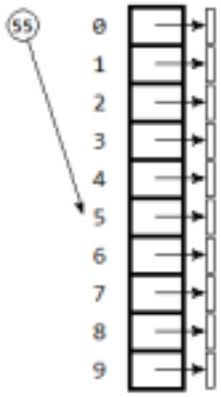
5. Otvorte modul `_collections_abc` (v IDLE menu File -> Open Module...), nájdite v ňom triedy `Mapping` a `MutableMapping` a porovnajte niektoré tam zapísané metódy s vašim riešením zadania (2)

Asociatívne polia II.

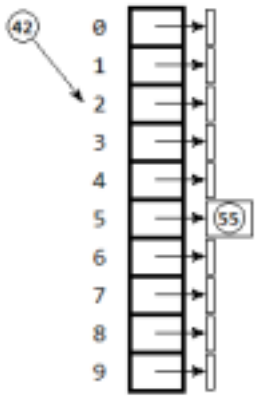
Minulú prednášku sme skončili hašovacou tabuľkou, v ktorej sme kolízie riešili tak, že všetky dvojice (kľúč, hodnota), ktorým vyšiel rovnaký vypočítaný index do hašovacej tabuľky, sme sústredili do poľa - tzv. vedierka (bucket). V tejto verzii teda hašovacia tabuľka je pole "vedierok" (na začiatku sú všetky prázdne), do ktorých pridávame dvojice (kľúč, hodnota), resp. ich tu hľadáme, alebo odtiaľto vyhadzujeme.

Ukážme to na takomto príklade:

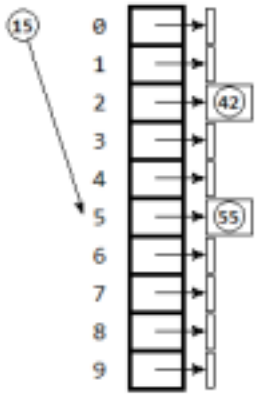
- pre jednoduchosť predpokladajme, že máme 10-prvkovú hašovaciu tabuľku (s indexmi od 0 do 9)
- do tabuľky budeme postupne vkladať kľúče (hodnoty si teraz nevšímame): 55, 42, 15, 60, 78, 35, 22



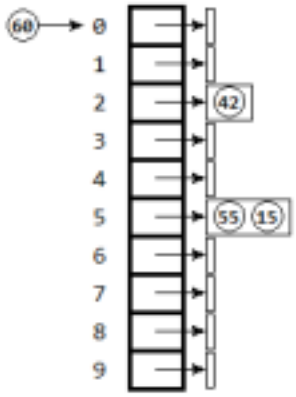
Požiadavka pridať 55 do prázdnej tabuľky



Teraz pridať 42 ako 2. prvok

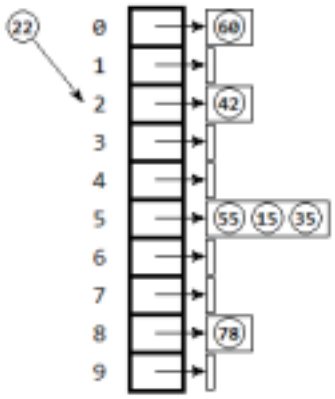


Ďalej pridať 15



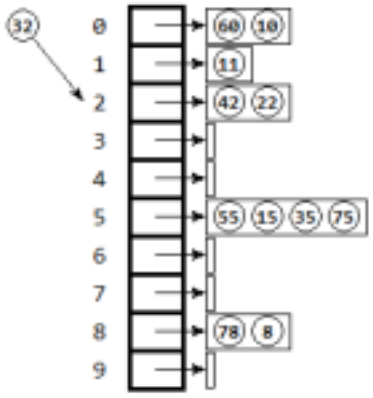
Ešte pridať 60

- po pridaní ešte 78 a 35 ešte ostáva:



Požiadavka pridať 22

- po vložení ďalších: 10, 11, 8, 75 a čaká vloženie 32:



čaká vloženie 32

Môžete si všimnúť, že do vedierok sa postupne nareťazujú ďalšie a ďalšie kľúče: niektoré vedierka sú stále prázdne, ale vedierko na indexe 5 obsahuje už 4 kľúče.

Takémuto hašovaniu sa hovorí **uzavreté** alebo **zreťazené hašovanie** (kolízie sú uzavreté v jednej reťazi dvojíc - niekedy sa vedierko realizuje nie poľom dvojíc ale spájaným zoznamom). Táto realizácia má tú výhodu, že sa ľahko implementuje, ale má aj niekoľko nedostatkov, napr.:

- má vyššie pamäťové nároky (okrem samotných dvojíc, poľa ako hašovacej tabuľky potrebujeme ešte minimálne toľko ďalších štruktúr ako je počet neprázdnych vedierok) - čím viac prvkov je vo vedierkach tým viac je nevyužitého priestoru v samotnej tabuľke (sú prázdne vedierka)
- ak sa vedierka zaplnia nad nejakú kritickú hranicu (napr. príde priveľa dvojíc s rovnakým indexom do tabuľky), výrazne sa spomalí celá realizácia asociatívneho poľa (operácie môžu mať zložitosť **O(n)**) - čím viac je prvkov v tabuľke, tým sú všetky operácie s ňou pomalšie
- ak máme viac kľúčov s veľmi blízkymi hašovacími hodnotami, tak tieto sa zvyknú sústrediť blízko seba (ak robíme len modulo veľkosť tabuľky)

Toto uzavreté hašovanie môžeme trochu vylepšiť, napr. takto:

- na začiatku rezervujeme malé pole vedierok, ktoré budeme v niektorých prípadoch nafukovať (podobne ako pri dynamických poliach a operácii `append`)

- prázdne vedierko nebudeme mať ako prázdne pole `[]` ale ako `None`

```
class HashMap(MapBase):
    def __init__(self):
        capacity = 11
        self._table = [None] * capacity
        self._num = 0

    ...

    def _resize(self, new_size):
        ...
```

- vylepšíme vzorec na výpočet indexu do tabuľky:
  - nebudeme si sami počítať hašovaciu funkciu, ale využijeme štandardnú funkcu `hash()`, ktorá z ľubovoľného (hašovateľného) typu vie vytvoriť nejaké celé číslo - my už z tohto čísla vypočítame zvyšok po delení veľkosťou tabuľky:

```
class HashMap(MapBase):
    ...

    def _hash(self, key):
        return hash(key) % len(self._table)
```

- keďže prázdne vedierka (bucket) sú teraz v poli zaznačené ako `None`, nemôžeme ich prechádzať pomocou for-cyklu, ale najprv musíme otestovať, či nie sú prázdne, napr.

```
class HashMap(MapBase):
    ...

    def __getitem__(self, key):
        ix = self._hash(key)
        bucket = self._table[ix]
        if bucket is not None:
            for item in bucket:
                ...
```

Nafukovať pole budeme vtedy, keď počet prvkov v ňom presiahne nejakú konkrétnu hranicu. Tu bude dôležitý práve pomer zaplnenia tabuľky (tzv. **load factor**), t.j. pomer počtu zaplnených prvkov v poli k veľkosti poľa. Nemalo by to presiahnuť hodnotu **1**. V praxi sa ukazuje, že tabuľka má dobrú veľkosť, keď je zaplnená na maximálne 90%. Všimnite si, že vo vyššie uvedenom príklade s hašovacou tabuľkou veľkosti 10, po pridaní aj posledného kľúča 22, bude load factor **1.2**, čo je viac ako 90%. Tabuľku zrejme bude treba nafukovať vtedy, keď sa do nej pridávať nový prvok (pomocou `__setitem__()`) a pritom sa presiahne pomer zaplnenia, napr.

```
class HashMap(MapBase):
    ...

    def __setitem__(self, key, value):
        ...
        if self._num > len(self._table) * 0.9:
            self._resize(len(self._table) * 2)
```

Samotný **resize** tabuľky bude podobný tomu, ako sme to robili pri nafukovaní dynamického poľa v operácii `append()`:

- do pomocného poľa si odložíme momentálny obsah tabuľky (všetky dvojice (kľúč, hodnota))
- vytvoríme novú prázdnu tabuľku požadovanej veľkosti

- postupne sem pridáme všetky zapamätané dvojice (kľúč, hodnota)

Kompletný listing pre **zreťazené hašovanie**:

```
class ChainHashMap(MapBase):
    def __init__(self):
        capacity = 11
        self._table = [None] * capacity
        self._num = 0

    def _hash(self, key):
        return hash(key) % len(self._table)

    def __getitem__(self, key):
        ix = self._hash(key)
        bucket = self._table[ix]              # zisti spravne vedierko
        if bucket is not None:                # ak je neprazdne
            for item in bucket:               # skontroluj v nom vsetky kluce
                if item._key == key:
                    return item._value
        raise KeyError

    def __setitem__(self, key, value):
        ix = self._hash(key)
        bucket = self._table[ix]
        if bucket is not None:
            for item in bucket:
                if item._key == key:
                    item._value = value
                    return
        self._num += 1
        if bucket is None:
            bucket = self._table[ix] = []
        bucket.append(self._Item(key, value))
        if self._num > len(self._table) * 0.9:
            self._resize(len(self._table) * 2)

    def __delitem__(self, key):
        ix = self._hash(key)
        bucket = self._table[ix]
        if bucket is not None:
            for i in range(len(bucket)):
                if bucket[i]._key == key:
                    bucket.pop(i)
                    self._num -= 1
                    return
        raise KeyError

    def __len__(self):
        return self._num

    def __iter__(self):
        for bucket in self._table:
            if bucket is not None:
                for item in bucket:
                    yield item._key

    def items(self):
        for bucket in self._table:
```

```
            if bucket is not None:
                for item in bucket:
                    yield item._key, item._value

    def _resize(self, new_size):
        old_table = list(self.items())
        self._table = [None] * new_size
        self._num = 0
        for key, value in old_table:
            self[key] = value
```

Môžeme to otestovať podobným testovaním ako minule:

```
import random

dic1 = ChainHashMap()
pole = [random.randrange(100000) for i in range(50000)]
dic2 = {}
for i in pole:
    dic1[i] = dic1.get(i, 0) + 1
    dic2[i] = dic2.get(i, 0) + 1
set1 = {(k, dic1[k]) for k in dic1}
set2 = {(k, dic2[k]) for k in dic2}
print(set1==set2)
for i in range(1, 100000, 2):
    try:
        del dic1[i]
    except KeyError:
        pass
    try:
        del dic2[i]
    except KeyError:
        pass
set1 = {(k, dic1[k]) for k in dic1}
set2 = {(k, dic2[k]) for k in dic2}
print(set1==set2)
```

V tomto teste nielen vytvárame asociatívne pole (ako frekvenčnú tabuľku výskytu čísel od 0 do 99999 v náhodnom poli), ale potom z tejto tabuľky vyhodíme všetky nepárne kľúče.

6.1 Otvorené hašovanie

Zmeňme stratégiu pri riešení kolízií:

- do tabuľky budeme na vypočítané pozície ukladať priamo dvojicu (kľúč, hodnota) - namiesto reťaze dvojíc
- kým nepríde ku kolízii (máme umiestniť novú dvojicu (kľúč, hodnota) na obsadenú pozíciu), je všetko v poriadku
- ak je teda vypočítaná pozícia už obsadená, treba vybrať nejakú inú, ale tak, aby sme ju pri neskoršom hľadaní našli
- možností je viac, ale najjednoduchšou je použiť nasledovné políčko v tabuľke, resp. postupne hľadať najbližšie voľné
- tomuto hovoríme **otvorené hašovanie**

Tomuto hovoríme **linear probing**, t.j. lineárne pokusy. Výpočet indexu by sme mohli zapísať:

```
index = (hash(key) + i) % N
```

Kde `N` je veľkosť tabuľky a `i` je i-ty pokus o nájdenie voľného miesta, teda na začiatku 0, potom 1, ...

Hľadanie skutočnej pozície kľúča bude teraz trochu komplikovanejšie (zapíšeme to do pomocnej metódy `_find(key)`):

- podľa kľúča zistíme, na akom indexe by sa mal nachádzať (pomocná metóda `_hash(key)`)
- ak je to prázdne políčko tabuľky, znamená to, že sme nenašli a vrátime hodnotu `False`
- ak je to dvojica (kľúč, hodnota), porovnáme kľúč s hľadaným `key`, ak to sedí, vrátime `True` a nájdený index
- inak zvýšime index o 1 (prípadne urobíme modulo veľkosť tabuľky) a pokračujeme v hľadaní

Ak nie je tabuľka úplne zaplnená, určite skončíme buď na `None` alebo na políčku s hľadaným kľúčom. Podobne ako pri zreťazenom hašovaní, aj pri tomto otvorenom hašovaní musíme zabezpečiť, aby boli v tabuľke voľné miesta. V tomto prípade sú voľné miesta ešte dôležitejšie, lebo nám robia zarážky pri hľadaní. Preto **load factor** by mal byť výrazne menší ako 1, odporúča sa medzi 0.5 a 0.8. Napr. rôzne implementácie hašovacích tabuliek v rôznych programovacích jazykoch používajú rozne hodnoty, napr. v Jave je to 75%, v Pythone 66%.

6.1.1 Vyhodenie prvku z tabuľky

Keď nájdeme prvok, ktorý chceme vyhodiť (v metóde `__delitem__()`), nemôžeme ho jednoducho nahradiť `None`, lebo takýto `None` pre nás znamená zarážku v hľadaní, teda v lineárnych pokusoch (linear probing). Vyhadzovaný prvok nahradíme špeciálnou hodnotou `_avail` (môže byť ľubovoľného typu, len aby sme to rozpoznali od `None` a od dvojice `Item`). Pri hľadaní políčka s kľúčom budeme túto hodnotu `_avail` preskakovať, ale pri pridávaní nového kľúča (v metóde `__setitem__()`), tento `_avail` je kandidátom na pridanie novej dvojice.

Takže musíme opraviť pomocnú metódu `_find(key)` tak, aby preskakovala políčka s `_avail`, ale pritom si prvý takýto výskyt zapamätala, keby bolo treba do tabuľky pridávať. Funkcia bude vždy vracať dvojicu:

- `True, index` - keď nájde políčko s hľadaným kľúčom, v `index` je jeho pozícia
- `False, index` - keď nenájde takéto políčko, v `index` je pozícia prvého voľného políčka na zápis (buď je to políčko s `_avail` alebo `None`)

```
class HashMap(MapBase):
    _avail = 'avail'

    ...

    def _find(self, key):
        ix = self._hash(key)
        av = None                # prvy volny
        while True:
            item = self._table[ix]
            if item is None:
                if av is None:
                    av = ix
                return False, av
            elif item == self._avail:
                if av is None:
                    av = ix
            elif item._key == key:
                return True, ix
            ix = (ix + 1) % len(self._table)
```

Všimnite si, že `_avail` je tu nejaký znakový reťazec, čo sa ľahko rozlíši od `None` alebo `Item`.

Kompletný listing pre **otvorené hašovanie**:

```
class OpenHashMap(MapBase):
    _avail = 'avail'

    def __init__(self):
        capacity = 11
        self._table = [None] * capacity
        self._num = 0

    def _hash(self, key):
        return hash(key) % len(self._table)

    def _find(self, key):
        ix = self._hash(key)
        av = None
        while True:
            item = self._table[ix]
            if item is None:
                if av is None:
                    av = ix
                return False, av
            elif item == self._avail:
                if av is None:
                    av = ix
            elif item._key == key:
                return True, ix
            ix = (ix + 1) % len(self._table)

    def __getitem__(self, key):
        found, ix = self._find(key)
        if found:
            return self._table[ix]._value
        raise KeyError

    def __setitem__(self, key, value):
        found, ix = self._find(key)
        if found:
            self._table[ix]._value = value
            return
        self._num += 1
        self._table[ix] = self._Item(key, value)
        if self._num > len(self._table) * 0.5:
            self._resize(len(self._table) * 2)

    def __delitem__(self, key):
        found, ix = self._find(key)
        if found:
            self._table[ix] = self._avail
            self._num -= 1
        else:
            raise KeyError

    def __len__(self):
        return self._num

    def __iter__(self):
```

```
        for item in self._table:
            if item is not None and item != self._avail:
                yield item._key

    def items(self):
        for item in self._table:
            if item is not None and item != self._avail:
                yield item._key, item._value

    def _resize(self, new_size):
        old_table = list(self.items())
        self._table = [None] * new_size
        self._num = 0
        for key, value in old_table:
            self[key] = value
```

Resize tabuľky robíme vždy vtedy, keď počet prvkov tabuľke presiahne 50% veľkosti celej tabuľky.

Otestovanie môžete urobiť rovnaké, ako v predchadzajúcej realizácii `ChainHashMap`.

6.1.2 Iné metódy riešenia kolízií

Informatici hľadajú aj vhodnejšie spôsoby, ako vyriešiť kolízie: linear probing nie je najlepší, lebo kľúče s blízkymi hašovacími hodnotami sa zhlukujú (clustering) vedľa seba a tým sa spomaľuje samotné hľadanie - niekedy treba prezerať dlhú postupnosť prvkov v poli, ktoré sú tesne vedľa seba. Aj pri malom load factore (v tabuľke je veľa voľných políčok) bude treba často prekontrolovať skoro všetky prvky tabuľky. Bolo by vhodnejšie, keby mohli byť v poli viac rozptýlené. Vymenujme niekoľko ďalších možností:

- lineárne pokusy nejakým krokom `c` > 1:

  ```
  index = (hash(key) + i * c) % N
  ```

  kde `c` by mala byť konštanta, ktorá je nesúdeliteľná s `N` - veľkosťou tabuľky

  - ani toto nerieši problémom so zhlukovaním (clustering)

- kvadratické pokusy:

  ```
  index = (hash(key) + i**2) % N
  ```

  krok sa stále zväčšuje a teda je väčšia šanca, že sa prvky nebudú až tak zhlukovávať

- dvojité hašovanie (double hashing):

  ```
  index = (hash(key) + i * hash2(key)) % N
  ```

  druhá hašovacia funkcia `hash2()` by pre žiaden kľúč nemala vrátiť 0

  vychádza sa z toho, že ak majú dva rôzne kľúče rovnakú hodnotu `hash()`, tak práve v `hash2()` by sa mohli líšiť a teda sa znižuje šanca zhlukovania prvkov v poli

- využitie pseudo-random generátora (využíva to aj štandardný typ `dict` v Pythone):

  - namiesto `hash(key)` najprv nastavíme náhodný generátor pomocou `random.seed(key)`
  - a potom postupne voláme `random.randrange(N)`, ktorý nám vracia indexy do tabuľky
  - dôležité je, že pre každý kľúč bude táto postupnosť vygenerovaných indexov vždy rovnaká (pre rôzne kľúče bude samozrejme rôzna)

Python využíva asociatívne polia aj na vnútornú reprezentáciu menných priestorov (name space) - t.j. každý objekt, každé volanie funkcie (metódy) vytvára nové a nové menné priestory - sem si ukladá mená atribútov, lokálnych premenných a pod. Preto musí byť realizácia asociatívnych polí v Pythone veľmi rýchla a vysokospoľahlivá. Spomeňte si napr.

6.2 Realizácia množiny

V Pythone je štandardný typ množina `set` realizovaný pomocou hašovacej tabuľky:

- do tabuľky neukladáme dvojice (kľúč, hodnota), ale len samotné kľúče
- použijeme otvorené hašovanie, napr. linear probing, pričom vyhodené prvky tiež označíme ako `_avail` - v tomto prípade ale `_avail` nemôže byť ani znakový reťazec ani žiaden iný typ, ktorý by mohol byť prvkom množiny

Abstraktný dátový typ prispôsobíme základným operáciám na množine (namiesto `__getitem__()`, `__setitem__()` a `__delitem__()`):

```
from abc import ABCMeta, abstractmethod

class SetBase(metaclass=ABCMeta):

    @abstractmethod
    def __contains__(self, key):
        pass

    @abstractmethod
    def add(self, key):
        pass

    @abstractmethod
    def remove(self, key):
        pass

    @abstractmethod
    def __len__(self):
        pass

    @abstractmethod
    def __iter__(self):
        pass

    def discard(self, key):
        try:
            self.remove(key)
        except KeyError:
            pass
```

Samotná trieda pre množiny bude len veľmi zjednodušenou verziou `OpenHashMap`. Nebudeme ju tu uvádzať, keďže je to pekná úloha pre cvičenie. Len začiatok definície triedy:

```
class HashSet(SetBase):
    _avail = object()

    def __init__(self):
        capacity = 11
        self._table = [None] * capacity
```

```
        self._num = 0

    def _hash(self, key):
        return hash(key) % len(self._table)

    def _find(self, key):
        ...

    ...
```

Všimnite si, ako sme tu zadefinovali hodnotu `_avail`.

Pomocou hašovacích tabuliek sa zvyknú definovať aj ďalšie užitočné štruktúry, napr. **MultiSet** a **MultiMap**.

6.3 MultiSet

Je dátová štruktúra množina, v ktorej sa môžu prvky vyskytovať aj viackrát. Môžeme to riešiť napr. tak, že v hašovacej tabuľke ukladáme dvojice (kľúč, hodnota), kde kľúč je opäť prvok množiny (ako pri obyčajných množinách) ale v položke hodnota si pamätáme počet opakovaní tohto kľúča v množine. Takže pre množinu {'a', 'a', 'b', 'a'} vytvoríme v hašovacej tabuľke dve dvojice: ('a', 3), ('b', 1).

6.4 MultiMap

Je také asociatívne pole, v ktorom každý kľúč môže mať niekoľko hodnôt. Realizuje sa to pomocou hašovacej tabuľky napr. tak, že pre každý kľúč sa pamätá nie jedna hodnota, ale pole zodpovedajúcich hodnôt.

6.5 Cvičenie

1*. Ručne odsimulujte pridávanie kľúčov do otvorenej hašovacej tabuľky:

- začíname s prázdnou tabuľkou veľkosti 10 prvkov (s indexami od 0 do 9), pridávame kľúče, ktoré sú celé čísla a index vypočítate ako kľúč % veľkosť tabuľky
- na riešenie kolízií použite algoritmus "linear probing"
- zapĺňanie tabuľky môžete znázorňovať takto (postupne pridávame kľúče 55, 42, 15):

```
  0     1     2     3     4     5     6     7     8     9
None  None  None  None  None  None  None  None  None  None
None  None  None  None  None   55   None  None  None  None
None  None   42   None  None   55   None  None  None  None
None  None   42   None  None   55    15   None  None  None
```

  číslo 15 sme vložili na index 6, lebo 5. políčko už bolo obsadené

- pokračujte v pridávaní týchto kľúčov: 60, 28, 65, 10, 25
- keďže teraz je už tabuľka zaplnená na 80%, zrealizuje **resize**, t.j. vytvorte dvojnásobne veľkú tabuľku a postupne do nej (zľava doprava) prekopírujte všetky kľúče (už na nové pozície - budete robiť kľúč % 20):

```
  0     1     2     3     4     5     6     7     8     9    10
↪11    12    13    14    15    16    17    18    19
None  None  None  None  None  None  None  None  None  None  None
↪None  None  None  None  None  None  None  None  None
 60   None  None  None  None  None  None  None  None  None  None
↪None  None  None  None  None  None  None  None  None
 60   None  None  None  None  None  None  None  None  None   10
↪None  None  None  None  None  None  None  None  None
...
```

- potom pridajte do tejto tabuľky ešte aj tieto kľúče: 96, 86, 77, 34, 35, 36
- odstraňovanie kľúčov z tabuľky sa robí tak, že sa dané hodnoty nahradia hodnotou `avail`, odstráňte z tabuľky všetky párne čísla a potom postupne pridajte: 81, 61, 41, 21, 1

2*. Otestujte `OpenHashMap` z prednášky:

- Vytvorte frekvenčnú tabuľku slov zo súboru `text4.txt` z minulého cvičenia.
- Vysledné asociatívne pole porovnajte s riešením pomocou štandardného typu `dict`

3*. Zabudujte do `OpenHashMap` evidovanie počtu kolízií:

- použite nový atribút `_col`, ktorý sa bude zvyšovať o 1 pri kolíziách v pomocnej metóde `_find()`, t.j. vždy, keď sa premenná `ix` zvýši na nasledovný index
- zrejme pri **resize** sa počítadlo `_col` vynuluje a ďalšie volania `__setitem__()` vo for-cykle ho opäť zvýšia
- zistite počet kolízií v teste úlohy (2)

4*. Do triedy `OpenHashMap` pridajte možnosť nastaviť **load factor**, t.j. hranicu, kedy sa uskutoční **resize**. Použijete ho v metóde `__setitem__()`, keď sa zisťuje, či momentálny počet prvkov je väčší ako veľkosť tabuľky krát **load factor**. Tento **load factor** sa bude nastavovať pri vytváraní objektu, napr. `OpenHashMap(0.5)` bude označovať, že otvorené hašovanie bude fungovať s **load factorom** 0.5 (teda 50% zaplnenie tabuľky).

- Spustite test z úlohy (2) pre rôzne hodnoty **load factor**: 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
- Pre každý takýto test vypíšte počet kolízií a veľkosť hašovacej tabuľky (`_table`)
- Spustite tieto testy viackrát - zistite, prečo zakaždým dostávate rôzne výsledky

5. Realizácia asociatívneho poľa pomocou `OpenHashMap` rieši kolízie lineárnymi pokusmi (linear probing) s krokom 1. Dorobte do triedy ďalší parameter, vďaka ktorému môžeme nastaviť krok lineárnych pokusov, napr. `OpenHashMap(0.5,3)` bude označovať **load factor** 0.5 a krok, o ktorý sa bude zvyšovať index pri kolízii 3.

- Otestujte, či má veľkosť tohto kroku výrazný význam v počte kolízií (pre rôzne hodnoty **load factor** skúste rôzne kroky, najlepšie nepárne čísla, napr. 1, 3, 5, 7).
- Pre niektoré hodnoty kroku môže nastať zacyklenie hľadania - zabezpečte, aby sa v tomto prípade hľadanie `_find()` prerušilo nejakou výnimkou.

6*. Zrealizujte všetky metódy triedy `HashSet` z prednášky.

- Otestujte podobne ako v úlohe (2), ale namiesto frekvenčnej tabuľky vytvorte množinu (typu `HashSet`) slov.
- Vysledok porovnajte s riešením pomocou štandradného typu `set`.

7. Otestujte, ako sa bude správať `OpenHashMap` ak v metóde `_find()` využijeme pseudo-random generátor:

- namiesto `ix = self._hash(key)` použijeme `random.seed(key)` a `ix = random.randrange(len(self._table))`
- namiesto zvyšovania `ix` pri kolízii opäť použijeme `ix = random.randrange(len(self._table))`
- vyskúšajte test z úlohy (2) a zistite počet kolízií

KAPITOLA 7

Vyhľadávacie stromy

V predchádzajúcich prednáškach sme videli, ako realizovať asociatívne pole pomocou hašovacích tabuliek. Vďaka tomu, tri základné operácie (get, set, del) majú časovú zložitosť **O(1)**. Ak ale požadujeme, aby boli pritom všetky kľúče zoradené vzostupne, tak tu hašovaciu tabuľku nevyužijeme (jedine, ak vy sme zakaždým znovu a znovu všetky kľúče pretriedili).

Videli sme aj realizáciu asociovaného poľa pomocou utriedenej postupnosti kľúčov. Tu ale mali všetky operácie výrazne horšiu zložitosť:

- **get** - **O(log n)**
- **set** - **O(n)**
- **del** - **O(n)**

Vyhľadávať v utriedenom poli vieme binárnym vyhľadávaním (so zložitosťou **O(log n)**), ale zmena obsahu poľa, tak aby ostalo utriedené si bude vyžadovať drahé operácie **O(n)**.

V prvom ročníku sme sa zoznámili s binárnymi vyhľadávacími stromami, ktorých základné operácie majú zložitosť závislú len od výšky stromu, t.j. **O(h)**. Ak by sme predpokladali, že výška stromu je blízka **log n**, tak by sme vedeli skonštruovať asociatívne pole s utriedenými kľúčmi operáciami so zložitosťou **O(log n)**.

7.1 Binárne vyhľadávacie stromy

Zhrňme, čo o nich vieme z prvého ročníka:

- všetky vrcholy v ľavom podstrome sú menšie ako hodnota v koreni
- všetky vrcholy v pravom podstrome sú väčšie ako hodnota v koreni
- všetky podstromy sú tiež BVS

Ako dôsledok tohto sa dá dokázať, že prejdenie všetkých vrcholov v poradí **inorder** ich navštívi (vráti) v utriedenom poradí.

Pripomeňme si základné operácie. Vyhľadanie hodnoty (operácia **get**):

- porovná hľadanú hodnotu s koreňom stromu a ak sa rovná, našli sme hľadaný prvok
- inak, ak je hodnota v koreni väčšia ako hľadaný prvok, rekurzívne pokračujeme v ľavom podstrome
- inak, ak je hodnota v koreni menšia ako hľadaný prvok, rekurzívne pokračujeme v pravom podstrome
- ak je niektorý z podstromov prázdny, hľadanie končí neúspechom

Keďže sa tento algoritmus stále hlbšie a hlbšie vnára do podstromov celého stromu, ale vždy len jedným smerom, takýchto vnorení nemôže byť viac ako výška stromu (teda **O(h)**).

Operácia **set** najprv nájde vrchol s danou hodnotou (týmto istým postupom), alebo zistí, že sa v strome nenachádza - na danom mieste sa algoritmus zastavil na prázdnom podstrome. V tomto prípade na toto miesto napojí nový vrchol so zadanou hodnotou.

Operácia **del** tiež využije tento postup na hľadanie vyhadzovaného prvku, len bude potom musieť nejako meniť aj iné prvky stromu:

- ak má vyhadzovaný vrchol maximálne jedného syna, tak rodičovi namiesto vhadzovaného vrcholu napojíme tohto jediného syna (alebo `None`, ak to bol list stromu)
- ak má vyhadzovaný vrchol oboch synov, tak v ľavom podstrome nájdeme vrchol s maximálnou hodnotou (kým má daný vrchol pravého syna, presúvame sa vpravo), túto prekopírujeme ako novú hodnotu do vrcholu, ktorý sme mali vyhodiť a namiesto tohto vrcholu, vyhodíme bývalý maximálny vrchol (tento určite nemá pravého syna a teda sa ľahko zo stromu vyhadzuje)

Binárny vyhľadávací strom využijeme pri realizácii asociatívneho poľa takto:

- metódy pre hľadanie, vkladanie a vyhodzovania nazveme `__getitem__()`, `__setitem__()` a `__delitem__()`
- do každého vrcholu okrem samotnej hodnoty (teda kľúča) vložíme aj príslušnú hodnotu, ktorá je s týmto kľúčom asociovaná (vo vrcholoch si pamätáme dvojicu (kľúč, hodnota) ako dva atribúty `key` a `value`)
- iterátor `__iter__()` zapíšeme ako **inorder** prechádzanie stromu

Najprv definícia obyčajného binárneho stromu (uložená napr. v súbore `bin_tree.py`):

```
import tkinter
import random

class BinTree:
    class _Node:
        def __init__(self, key, value=None, parent=None):
            self._key = key
            self._value = value
            self._parent = parent
            self._left = None
            self._right = None

        def __repr__(self):
            if self._value is not None:
                return '({!r},{!r})'.format(self._key, self._value)
            else:
                return repr(self._key)

    #--------------------------------------------------------------------

    def __init__(self):
        self._root = None
        self._size = 0

    def add_root(self, key, value=None):
        if self._root:
            raise ValueError('koren existuje')
        self._size = 1
        self._root = self._Node(key, value)

    def add_left(self, node, key, value=None):
```

```
        if node._left:
            raise ValueError('lavy syn existuje')
        self._size += 1
        node._left = self._Node(key, value, node)

    def add_right(self, node, key, value=None):
        if node._right:
            raise ValueError('pravy syn existuje')
        self._size += 1
        node._right = self._Node(key, value, node)

    def delete(self, node):
        '''vyhodi vrchol node a nahradi ho potomkom (ak ma prave jedneho)

        ValueError ak ma vrchol dvoch potomkov
        '''
        if node._left and node._right:
            raise ValueError('vrchol ma dvoch synov')
        child = node._left if node._left else node._right    # moze byt None
        if child:
            child._parent = node._parent      # prepise sa potomkom
        if node is self._root:
            self._root = child               # moze sa stat rootom
        else:
            parent = node._parent
            if node is parent._left:
                parent._left = child
            else:
                parent._right = child
        self._size -= 1

    #--------------------------------------------------------------------

    def __len__(self):
        return self._size

    def is_empty(self):
        return len(self) == 0     # return self._root is None

    def __iter__(self):
        yield from self.inorder()

    def inorder(self, node=None):
        if node is None:
            if self.is_empty():
                return
            node = self._root
        if node._left:
            yield from self.inorder(node._left)
        yield node
        if node._right:
            yield from self.inorder(node._right)

    def children(self, node):
        '''generuje iterovanie synov vrcholu'''
        if node._left:
            yield node._left
        if node._right:
```

```
            yield node._right

    def depth(self, node):
        if node == self._root:
            return 0
        else:
            return 1 + self.depth(node._parent)

    def _height(self, node):
        if node._left is None and node._right is None:
            return 0
        else:
            return 1 + max(self._height(child) for child in self.
↪children(node))

    def height(self, node=None):
        if node is None:
            node = self._root
        return self._height(node)

    def preorder(self, node=None):
        if node is None:
            if self.is_empty():
                return
            node = self._root
        yield node
        if node._left:
            yield from self.preorder(node._left)
        if node._right:
            yield from self.preorder(node._right)

    def postorder(self, node=None):
        if node is None:
            if self.is_empty():
                return
            node = self._root
        if node._left:
            yield from self.postorder(node._left)
        if node._right:
            yield from self.postorder(node._right)
        yield node

    def breadthfirst(self):
        '''generuje iterovanie vrcholov stromu algoritmom do sirky'''
        if not self.is_empty():
            queue = [self._root]            # simuluje queue pomocou pola
            while queue:
                node = queue.pop(0)        # draha operacia - zisiel by sa␣
↪spajany zoznam
                yield node
                for child in self.children(node):
                    queue.append(child)

    #--------------------- testovacie metody ----------------------------

    def add_random(self, key, value=None):
        if self.is_empty():
            self.add_root(key, value)
```

```
        else:
            node = self._root
            while True:
                if random.randrange(2):
                    if node._left:
                        node = node._left
                    else:
                        self.add_left(node, key, value)
                        return
                else:
                    if node._right:
                        node = node._right
                    else:
                        self.add_right(node, key, value)
                        return

    canvas_width = 800
    canvas = None

    def draw(self, node=None, width=None, x=None, y=None):
        if self.canvas is None:
            self.canvas = tkinter.Canvas(bg='white', width=self.canvas_
↪width, height=600)
            self.canvas.pack()
        elif node is None:
            self.canvas.delete('all')
        if node is None:
            self.canvas.delete('all')
            if self.is_empty():
                return
            node = self._root
            if width is None: width = int(self.canvas['width'])//2
            if x is None: x = width
            if y is None: y = 30
        if node._left:
            self.canvas.create_line(x, y, x - width//2, y + 50)
            self.draw(node._left, width//2, x - width//2, y + 50)
        if node._right:
            self.canvas.create_line(x, y, x + width//2, y + 50)
            self.draw(node._right, width//2, x + width//2, y + 50)
        self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
        self.canvas.create_text(x, y, text=node)
        if node is self._root:
            self.canvas.update()
            self.canvas.after(100)
```

Všimnite si, že sme sem doplnili metódu `draw()` na vykreslenie stromu do grafickej plochy. Opätovné jej zavolania, vymaže predchádzajúci obrázok a nakreslí strom znovu. Tiež si všimnite, že v každom vrchole stromu okrem referencií na iné vrcholy máme aj dva atribúty `kye` a `value`.

Do tohto súboru môžeme vložiť aj nejaké testovanie, napr.

```
if __name__ == '__main__':
    strom = BinTree()
    for i in range(25):
        strom.add_random(random.randrange(20))

    print('preorder     =', *strom.preorder())
```

```
    print('inorder      =', *strom.inorder())
    print('postorder    =', *strom.postorder())
    print('breadthfirst =', *strom.breadthfirst())
    strom.draw()
```

Aby sme mohli realizovať asociatívne pole, zadefinujeme abstraktný dátový typ `MapBase` aj so všetkými metódami. Zapíšme to do súboru, napr. `map_base.py`:

```
from abc import ABCMeta, abstractmethod

class MapBase(metaclass=ABCMeta):

    class _Item:
        def __init__(self, key, value):
            self._key, self._value = key, value

        def __repr__(self):
            return repr(self._key) + ':' + repr(self._value)

    #------------------

    @abstractmethod
    def __getitem__(self, key):
        pass

    @abstractmethod
    def __setitem__(self, key, value):
        pass

    @abstractmethod
    def __delitem__(self, key):
        pass

    @abstractmethod
    def __len__(self):
        pass

    @abstractmethod
    def __iter__(self):
        pass

    #------------------------------------------------------------
    def __contains__(self, key):
        try:
            self[key]
        except KeyError:
            return False
        else:
            return True

    def get(self, key, default=None):
        'D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.'
        try:
            return self[key]
        except KeyError:
            return default

    def keys(self):
```

```
        "D.keys() -> a set-like object providing a view on D's keys"
        for key in self:
            yield key

    def items(self):
        "D.items() -> a set-like object providing a view on D's items"
        for key in self:
            yield (key, self[key])

    def values(self):
        "D.values() -> an object providing a view on D's values"
        for key in self:
            yield self[key]

    def __eq__(self, other):
        if not isinstance(other, MapBase):
            return NotImplemented
        return dict(self.items()) == dict(other.items())

    def __ne__(self, other):
        return not (self == other)

    def pop(self, key, default=object()):
        '''D.pop(k[,d]) -> v, remove specified key and return the␣
↪corresponding value.
          If key is not found, d is returned if given, otherwise KeyError is␣
↪raised.
        '''
        try:
            value = self[key]
        except KeyError:
            if default is object():
                raise
            return default
        else:
            del self[key]
            return value

    def popitem(self):
        '''D.popitem() -> (k, v), remove and return some (key, value) pair
           as a 2-tuple; but raise KeyError if D is empty.
        '''
        try:
            key = next(iter(self))
        except StopIteration:
            raise KeyError
        value = self[key]
        del self[key]
        return key, value

    def clear(self):
        'D.clear() -> None.  Remove all items from D.'
        try:
            while True:
                self.popitem()
        except KeyError:
            pass
```

```
    def update(*args, **kwds):
        ''' D.update([E, ]**F) -> None.  Update D from mapping/iterable E␣
↪and F.
            If E present and has a .keys() method, does:     for k in E:␣
↪D[k] = E[k]
            If E present and lacks .keys() method, does:     for (k, v) in E:
↪ D[k] = v
            In either case, this is followed by: for k, v in F.items(): D[k]␣
↪= v
        '''
        if len(args) > 2:
            raise TypeError('update() takes at most 2 positional '
                            'arguments ({} given)'.format(len(args)))
        elif not args:
            raise TypeError('update() takes at least 1 argument (0 given)')
        self = args[0]
        other = args[1] if len(args) >= 2 else ()

        if isinstance(other, MapBase):
            for key in other:
                self[key] = other[key]
        elif hasattr(other, 'keys'):
            for key in other.keys():
                self[key] = other[key]
        else:
            for key, value in other:
                self[key] = value
        for key, value in kwds.items():
            self[key] = value

    def setdefault(self, key, default=None):
        'D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D'
        try:
            return self[key]
        except KeyError:
            self[key] = default
        return default

    def __repr__(self):
        res = ''
        for key in self:
            if res != '': res += ','
            res += '{!r}:{!r}'.format(key, self[key])
        return '{' + res + '}'
```

V tomto súbore je definovaná aj podtrieda `_Item` pre dvojice v asociatívnom poli. Hoci v binárnom vyhľadávacom strome to nevyužijeme, ponechali sme to tu kvôli iným realizáciám.

Asociatívne pole `TreeMap` pomocou binárneho vyhľadávacieho stromu (zapíšeme do súboru `tree_map.py`):

```
import bin_tree, map_base
import random

class TreeMap(bin_tree.BinTree, map_base.MapBase):

    def _find(self, node, key):
        if key == node._key:
            return node                             # našiel
```

```
        elif key < node._key and node._left:
            return self._find(node._left, key)      # vnorenie do l'avého␣
↪podstromu
        elif key > node._key and node._right:
            return self._find(node._right, key)     # vnorenie do pravého␣
↪podstromu
        return node

    def __getitem__(self, key):
        if self.is_empty():
            raise KeyError
        node = self._find(self._root, key)
        if key == node._key:
            return node._value if node._value is not None else node._key
        else:
            raise KeyError

    def __setitem__(self, key, value):
        if self.is_empty():
            self.add_root(key, value)
        else:
            node = self._find(self._root, key)
            if key == node._key:
                node._value = value                 # nastav novú asociovanú␣
↪hodnotu
            elif key < node._key:
                self.add_left(node, key, value)     # zdedené z BinTree
            else:
                self.add_right(node, key, value)    # zdedené z BinTree

    def __iter__(self):
        if not self.is_empty():
            for node in self.inorder():
                yield node._key

    def __delitem__(self, key):
        if self.is_empty():
            raise KeyError
        node = self._find(self._root, key)
        if key != node._key:
            raise KeyError
        if node._left and node._right:              # vrchol má oboch synov
            r = node._left                          # hlada vrchol s␣
↪maximálnou hodnotou
            while r._right:
                r = r._right
            node._key = r._key
            node._value = r._value
            node = r                                # nastavíme, že␣
↪vyhadzovat' budeme r
        # teraz má vrchol max. 1 syna
        self.delete(node)                           # zdedené z BinTree
```

Všimnite si, že táto nová trieda je odvodená od dvoch iných tried: `BinTree` a `MapBase`. Tomuto hovoríme **viacnásobná dedičnosť** tried.

Do tohto súboru môžeme pridať aj testovanie (podobné tomu, ako sme testovali napr. hašovacie tabuľky pomocou frekvenčnej tabuľky výskytov nejakých celočíselných hodnôt):

```
if __name__ == '__main__':
    dic1 = TreeMap()
    pole = [random.randrange(100000) for i in range(50000)]
    dic2 = {}
    for i in pole:
        dic1[i] = dic1.get(i, 0) + 1
        dic2[i] = dic2.get(i, 0) + 1
    print(dic1.height(), dict(dic1.items())==dic2)
    for i in range(1, 100000, 2):
        try:
            del dic1[i]
        except KeyError:
            pass
        try:
            del dic2[i]
        except KeyError:
            pass
    print(dic1.height(), dict(dic1.items())==dic2)
```

Ak by táto tabuľka nemala 50000 pridaných hodnôt, ale výrazne menej, mohli by sme takýto BVS aj priebežne vykreslovať, napr.

```
if __name__ == '__main__':
    dic1 = TreeMap()
    pole = [random.randrange(100) for i in range(50)]
    for i in pole:
        dic1[i] = dic1.get(i, 0) + 1
        dic1.draw()
    for i in range(1, 100, 2):
        try:
            del dic1[i]
        except KeyError:
            pass
        dic1.draw()
```

7.2 Vyvažovanie vyhľadávacích stromov

Veľkým problémom vyhľadávacích stromov je to, že ich výška je veľmi závislá od poradia hodnôt, ktoré sa do neho vkladajú. Ak by sem napríklad vložili usporiadanú postupnosť hodnôt:

```
if __name__ == '__main__':
    dic1 = TreeMap()
    pole = sorted([random.randrange(100) for i in range(20)])
    ...
```

Vytvorili by sem degenerovaný strom: všetky vrcholy majú iba pravého syna, čo je obyčajný spájaný zoznam. Výškou takéhoto stromu je počet vrcholov, teda všetky tri základné operácie sa degradujú na **O(n)**. Toto je najhorší prípad a teda naša očakávaná zložitosť **O(log n)** je v skutočnosti **O(n)**.

Tu by veľmi pomohol mechanizmus, ktorým by sem vedeli zabezpečiť, aby bol tvar stromu pri ľubovoľných operáciách **vyvážený**, t.j. výška ľavého aj pravého podstromu (pre každý vrchol) by bola približne rovnaká. Takýmto vyhľadávacím stromom budeme hovoriť **vyvážené vyhľadávacie stromy**, resp. **balanced search tree**.

Rôznych algoritmov, ktorými sa to dá zabezpečiť je viac, my si ukážeme len jeden z nich (učebnicovo) najbežnejší **AVL** (pomenovaný podľa mien dvoch ruských informatikov - autorov tejto idey: Adelson-Velsky a Landis). Vyvažovacie techniky vychádzajú s idey **rotácie vrcholov**, t.j. ak v BVS dva vrcholy `a` a `b` majú pod sebou zavesené

podstromy (všetko sú to BVS) `t0`, `t1` a `t3` každý nejakých výšok, tak tieto dva tvary stromov sú oba BVS s rovnakými hodnotami vrcholov len sú trochu inak poukladané. Takýmto prerobením stromu z jedného tvaru na druhý vieme "jemne" zmeniť výšku stromu, resp. podstromov. Ak by sa napr. ukázalo, že pridaním alebo odobraním nejakého vrcholu sa zmenila vyváženosť stromu (napr. ľavý podstrom má výrazne väčšiu výšku ako pravý), môžeme zarotovať nejaké vrcholy, prípadne zarotovať aj viackrát a tým strom opäť vyvážiť.

Budeme hovoriť, že BVS je vyvážený vtedy, keď výšky jeho podstromov sa líšia maximálne o 1. Pre každý vrchol vieme určiť jeho výšku (ako hlboko je v jeho celom podstrome: listy majú výšku 1, vrchol, z ktorého vychádza iba 1 alebo 2 listy, má výšku 2, ...).

Tu existuje niekoľko prístupov v počítaní balansu pre každý vrchol (balans = výška pravého podstromu - výška ľavého podstromu). My si budeme v každom vrchole pamätať jeho výšku (atribút `_height`). Niektoré iné prístupy si v každom vrchole pamätajú priamo tento balans.

Vyvažovanie stromu potom znamená po každom vložení nového vrcholu (v metóde `__setitem__()`), resp. vyhodenie (v metóde `__delitem__()`) vyvážiť celý strom. Toto väčšinou spôsobí 1 alebo 2 rotácie v strome. Keďže rotácia vrcholov je vlastne len niekoľko priradení referencií, dalo by sa ukázať, že časová zložitosť tejto operácie je **O(1)**.

Aby sme mohli vyvažovať binárny vyhľadávací strom, trochu upravíme jeho metódy:

```
class TreeMap(bin_tree.BinTree, map_base.MapBase):

    ...

    def __setitem__(self, key, value):
        if self.is_empty():
            self.add_root(key, value)
        else:
            node = self._find(self._root, key)
            if key == node._key:
                node._value = value                 # nastav novú asociovanú␣
↪hodnotu
            elif key < node._key:
                self.add_left(node, key, value)     # zdedené z BinTree
                self.rebalance(node)
            else:
                self.add_right(node, key, value)    # zdedené z BinTree
                self.rebalance(node)

    def __delitem__(self, key):
        if self.is_empty():
            raise KeyError
        node = self._find(self._root, key)
        if key != node._key:
            raise KeyError
        if node._left and node._right:              # vrchol má oboch synov
            r = node._left                          # hlada vrchol s␣
↪maximálnou hodnotou
            while r._right:
                r = r._right
            node._key = r._key
            node._value = r._value
```

```
            node = r                                    # nastavíme, že␣
↪vyhadzovat' budeme r
        # teraz má vrchol max. 1 syna
        parent = node._parent
        self.delete(node)                               # zdedené z BinTree
        self.rebalance(parent)

    def rebalance(self, node):
        pass
```

Nová metóda `rebalance()` zatiaľ nerobí nič, ale jej volania sú už pripravené na správnych miestach metód `__setitem__()` a `__delitem__()`.

Teraz pripravíme triedu `AVLTreeMap` odvodenú od `TreeMap`, v ktorej zadefinujeme samotné vyvažovanie pomocou `rebalance()`. Inštancie tejto triedy už budú asociatívne polia realizované pomocou binárnych vyhľadávacích stromov, pričom po každej operácii, ktorá mení tvar stromu sa automaticky spustí vyvažovanie.

```
import tree_map

class AVLTreeMap(tree_map.TreeMap):

    #-------------------------- vnorena trieda Node --------------------------
↪-
    class _Node(tree_map.TreeMap._Node):

        def __init__(self, key, value=None, parent=None):
            super().__init__(key, value, parent)
            self._height = 1            # este sa to prepocita

        def left_height(self):
            return self._left._height if self._left is not None else 0

        def right_height(self):
            return self._right._height if self._right is not None else 0

    #------------------------- pomocne metody -------------------------------
    def recompute_height(self, node):
        node._height = 1 + max(node.left_height(), node.right_height())

    def isbalanced(self, node):
        return abs(node.left_height() - node.right_height()) <= 1

    def tall_child(self, node, favorleft=False):
        if node.left_height() + (1 if favorleft else 0) > node.right_
↪height():
            return node._left
        else:
            return node._right

    def tall_grandchild(self, node):
        child = self.tall_child(node)
        alignment = (child == node._left)
        return self.tall_child(child, alignment)

    def rebalance(self, node):
        while node is not None:
            old_height = node._height
            if not self.isbalanced(node):
```

```
                node = self.restructure(self.tall_grandchild(node))
                self.recompute_height(node._left)
                self.recompute_height(node._right)
            self.recompute_height(node)
            if node._height == old_height:
                node = None
            else:
                node = node._parent

  #---------------------- pomocne metody pre tree balancing ------------------
↪---

    def relink(self, parent, child, make_left_child):
        if make_left_child:
            parent._left = child
        else:
            parent._right = child
        if child is not None:
            child._parent = parent

    def rotate(self, node):
        """Rotate node p above its parent.

        Switches between these configurations, depending on whether p==a or␣
↪p==b.

              b                 a
             / \               /  \
            a  t2            t0   b
           / \                    / \
          t0  t1                 t1  t2

        Caller should ensure that p is not the root.
        """
        """Rotate Position p above its parent."""
        x = node
        y = x._parent
        z = y._parent
        if z is None:
            self._root = x
            x._parent = None
        else:
            self.relink(z, x, y == z._left)

        if x == y._left:
            self.relink(y, x._right, True)
            self.relink(x, y, False)
        else:
            self.relink(y, x._left, False)
            self.relink(x, y, True)

    def restructure(self, x):
        """Perform a trinode restructure among Position x, its parent, and␣
↪its grandparent.

        Return the Position that becomes root of the restructured subtree.

        Assumes the nodes are in one of the following configurations:
```

```
            z=a                 z=c             z=a                 z=c
           /   \               /   \           /   \               /   \
         t0   y=b            y=b   t3        t0    y=c           y=a   t3
             /   \           /   \                 /   \         /   \
           t1    x=c       x=a   t2              x=b   t3      t0    x=b
                /   \      /   \                 /   \               /   \
              t2    t3    t0    t1              t1    t2            t1    t2

        The subtree will be restructured so that the node with key b becomes␣
↪its root.

                  b
                /   \
              a       c
             / \     / \
            t0  t1  t2  t3

        Caller should ensure that x has a grandparent.
        """
        """Perform trinode restructure of Position x with parent/grandparent.
↪"""
        y = x._parent
        z = y._parent
        if (x == y._right) == (y == z._right):
            self.rotate(y)
            return y
        else:
            self.rotate(x)
            self.rotate(x)
            return x
```

7.3 Zložitosť operácií

Zhrňme zložitosť základných operácií pre všetky realizácie asociatívnych polí:

operácia	unsorted	sorted	hash	BVS	AVL
get	**O(n)**	**O(log n)**	**O(1)**	**O(h)**	**O(log n)**
set	**O(n)**	**O(n)**	**O(1)**	**O(h)**	**O(log n)**
del	**O(n)**	**O(n)**	**O(1)**	**O(h)**	**O(log n)**
sort	**O(n log n)**	**O(n)**	**O(n log n)**	**O(n)**	**O(n)**

Posledný riadok **sort** označuje zložitosť operácie, ak by sme potrebovali získať utriedenú postupnosť kľúčov (napr. ako iterátor). **h** pre BVS označuje výšku stromu, čo je často **O(log n)**, ale najhorší prípad je **O(n)**.

7.4 Cvičenie

1. Ručne vyrobte BVS (bez vyvažovania) z nejakých 20 náhodných čísel
   - ku každému vrcholu pripíšte jeho výšku aj balans
2. Spojazdnite `TreeMap` z prednášky (asociatívne pole realizované pomocou BVS)
   - otestujte frekvenčnú tabuľku (z prednášky)

- otestujte na iba 30 náhodných údajoch a pritom priebežne vykresľujte BVS
- na takto nakreslenom strome skúste pre každý vrchol zistiť balans (rozdiel výšok pravého a ľavého podstromu)

3. Pomocná metóda `_find()` v triede `TreeMap` je rekurzívna:

- v niektorých situáciách to spadne na preplnení rekurzie - vygenerujte také dáta, aby táto rekurzia spadla
- prepíšte túto metódu bez rekurzie a otestujte na dátach, na ktorých to predtým spadlo

4. Do `BinTree` dopíšte dve metódy `next(p)` a `prev(p)`, pre ktoré je parametrom referencia na nejaký vrchol (typu `_Node`):
   - metóda `next(None)` vráti najľavejší vrchol stromu (bol by prvý pri výpise inorder)
   - metóda `next(p)` pre `p` rôzne od `None` vráti nasledovný vrchol, ktorý by nasledoval vo výpise inorder za daným `p`
   - za posledným vrcholom (najpravejším v strome, t.j. posledným v inorder) metóda vráti `None`

```
p = strom.next(None)
while p is not None:
    print(p._key, end=' ')
    p = strom.next(p)
print()
```

   takto by sa vypísala postupnosť inorder

   - metóda `prev()` funguje opačne k `next()`, t.j. pre `None` vráti posledný vrchol v inorder, inak vráti predchádzajúci vrchol, resp. `None` pre prvý
   - obe metódy otestujte na `TreeMap` - mali by vygenerovať usporiadané postupnosti hodnôt (kľúčov)

5. Spojazdnite `AVLTreeMap` z prednášky:

- sledujte (vykresľujte), ako sa priebežne vyvažuje celý strom, keď sa do neho pridávajú, resp. vyhadzujú vrcholy
- otestujte na veľkých údajoch (frekvenčná tabuľka z prednášky)

6. Upravte vykresľovanie AVL-stromu do grafickej plochy tak, aby sa vo vrcholoch namiesto hodnôt zobrazoval balans

- otestujte na malom počte náhodných údajov

KAPITOLA 8

Triedenia

Pripomeňme si, čo už vieme o niektorých triedeniach z programovania v prvom ročníku, ale aj z niektorých prednášok v tomto predmete:

- niektoré triedenia sú veľmi pomalé a majú zložitosť **O(n**2)**, napr.
- **bubble_sort** - **n**- krát prejde celé pole a zakaždým porovnáva všetky susedné prvky a prípadne ich navzájom vymení:

```
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]
```

  po každom prechode sa na koniec poľa presťahuje ďalšie maximum, teda po každom prechode je ďalší a ďalší prvok na svojom mieste, môžeme tento algoritmus trochu vylepšiť (vnorený cyklus bude zakaždým o 1 kratší), ale napriek tomu to bude stále **O(n)**:

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

- **selection_sort** (volali sme ho aj **min_sort**) najprv nájde v poli príslušné minimum a toto presťahuje niekde na začiatok:

```
def selection_sort(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]
```

  počet porovnaní je tu opäť **O(n**2)**

- **insert_sort** postupne vkladá na správne miesto do utriedenej časti všetky prvky poľa:

```
def insert_sort(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
```

Najprv je utriedenou časťou len 1. prvok, potom sa sem pridá na správne miesto 2. prvok, potom medzi ne 3. prvok, ...

8.1 Rozdeľuj a panuj

Táto programátorská schéma (divide-and-conquer pattern) sa používa na riešenie mnohých algoritmických problémov. My si ju ukážeme na dvoch algoritmoch triedenia. Skladá sa z troch krokov:

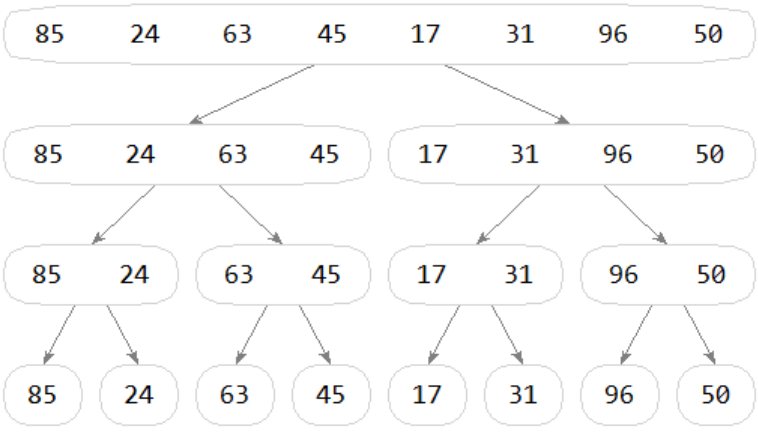
1. ak má úloha nejakú aspoň minimálnu veľkosť, rozdeľ ju na niekoľko disjunktných častí, inak vyrieš triviálny prípad (bez rekurzie)
2. rekurzívne vyrieš úlohu pre každú z častí
3. spoj tieto riešenia do výsledného celku

8.2 Merge sort

poznáme z 1. ročníka:

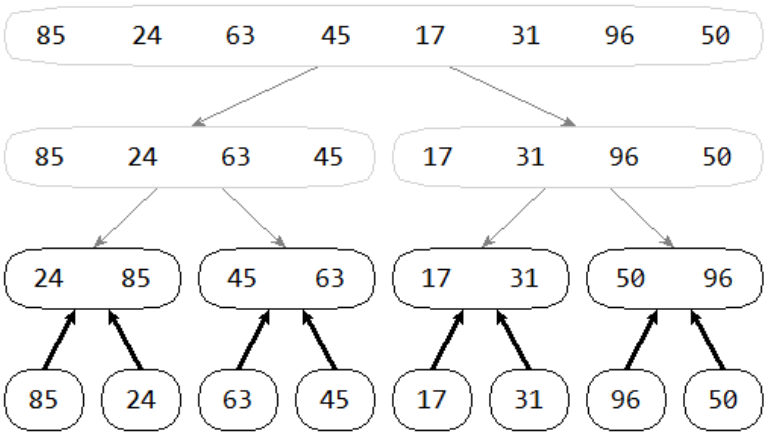
1. ak má pole aspoň 2 prvky, rozdeľ ho na dve rovnaké časti (napr. veľkosti `n//2` a `n-n//2`)
2. rekurzívne zavolaj triedenie zvlášť pre každú z častí - dostaneme 2 utriedené časti celého poľa
3. zlúč (merge) obe utriedené časti do výsledného poľa

Ideu rozdeľovania poľa na polovice vidíme (tzv. merge_sort strom):

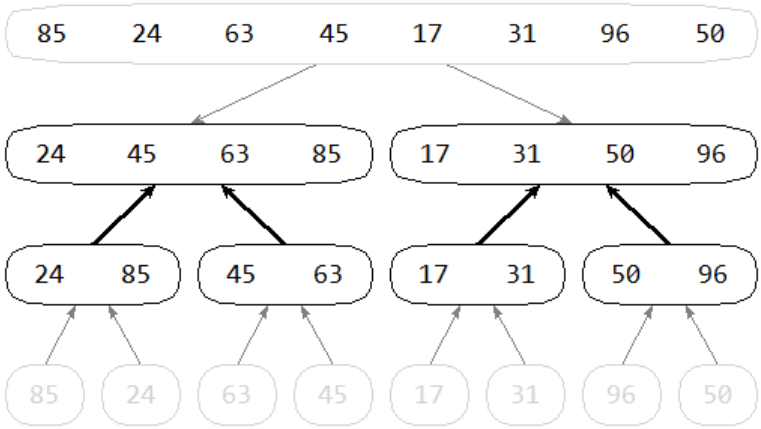


Toto bola 1. fáza algoritmu, keď sa pole rozdelilo na dve polovice a pre každú polovicu sa spustilo rekurzívne volanie, t.j. opäť rozdelenie na polovice. Toto pokračovalo dovtedy, kým bolo čo deliť, teda 1-prvkové pole sa už ďalej nedelilo.

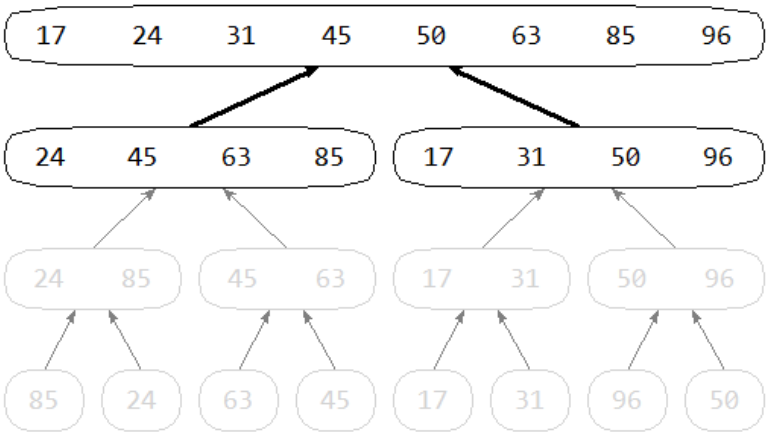
Teraz prichádza fáza návratu z rekurzie, keď sa majú dve polovice poľa zlúčiť (tzv. merge) do utriedeného celku. Lenže vďaka rekurzii sa najprv budú zlučovať dve najvnorenejšie polovice, t.j. 1-prvkové polia: z nich sa vytvoria utriedené dvojprvkové:



Na tomto obrázku vidíme princíp triedenia a nie skutočný algoritmus, lebo tu sme zrealizovali vynáranie z rekurzie naraz vo všetkých prípadoch, keď boli polia jednoprvkové. Pri ďalšom vynáraní sa z rekurzie opäť prebehne fáza zlučovania, keď sa z utriedených dvojprvkových polí vytvoria štvorprvkové:



Takto to celé pokračuje, až kým sa postupne nezlúčia všetky polovice a teda nedostaneme výsledné utriedené pole:



Zapíšeme tento rekurzívny algoritmus ako **in-place** triedenie, t.j. taká funkcia, ktorá modifikuje samotné pole a nevracia žiadnu hodnotu:

```
def merge_sort(pole):
    if len(pole) < 2:
        return
    stred = len(pole)//2
    pole1 = pole[:stred]
    pole2 = pole[stred:]
    merge_sort(pole1)
    merge_sort(pole2)
    # zlucovanie oboch casti do vysledneho pola:
    i = j = 0
    while i + j < len(pole):
        if j == len(pole2) or i < len(pole1) and pole1[i] < pole2[j]:
            pole[i+j] = pole1[i]
            i += 1
        else:
            pole[i+j] = pole2[j]
            j += 1
```

Zložitosť tohto triedenia je **O(n log n)**, lebo rekurzia sa vnára **log n** krát a samotné zlučovanie má zložitosť **O(n)**. Pekne je to vidieť aj na "merge_sort strome": výška tohto stromu je naozaj dvojkový **log n**. Tiež vidíte, že v každej úrovni tohto strom sa nachádza presne **n** pôvodných prvkov, ktoré sú zoskupené do nejakých malých polí. Fáza zlučovania (merge) bude všetky tieto prvky (z malých polí) presúvať na správne miesta do dvojnásobne väčších polí. Keďže samotný cyklus "merge" je while-cyklus, ktorý prejde toľko-krát koľko je spolu prvkov v dvoch malých poliach, tak spolu všetky-merge-cykly prejdú presne **n** krát.

Všimnite si, že sa tu veľmi intenzívne využívajú pomocné polia - odhadnite, aká je celková veľkosť všetkých pomocných polí, ktoré sa použijú počas vykonávania tohto algoritmu (opäť si môžete pomôcť "merge_sort stromom").

Ak by sme netriedili pole, ale spájaný zoznam (napr. s metódami pre `Zoznam`: `__len__()`, `pridaj_kon()` a `daj_prvy()`, `prvy()`, ...), mohli by sme výrazne ušetriť pomocnú pamäť: totiž namiesto `pole1` a `pole2` by sme mohli použiť pomocné zoznamy, ktoré by sa vytvorili prekopírovaním prvkov z pôvodného zoznamu najprv do prvého zoznamu a potom do druhého. Podobne by aj záverečné zlučovanie z týchto dvoch zoznamov vytvorilo jediný. Napr.

```
def merge_sort(zoznam):
    n = len(zoznam)
    if n < 2:
        return
    zoz1 = Zoznam()
    zoz2 = Zoznam()
    while len(zoz1) < n//2:
        zoz1.pridaj_kon(zoznam.daj_prvy())        # cita zo zaciatku zoznamu␣
↪a pridava na koniec pomocneho
    while not zoznam.je_prazdny():
        zoz2.pridaj_kon(zoznam.daj_prvy())
    merge_sort(zoz1)
    merge_sort(zoz2)
    #zlucovanie oboch zoznamov do vysledneho
    while not zoz1.je_prazdny() and not zoz2.je_prazdny():      # kym su oba␣
↪zoznamy neprazdne
        if zoz1.prvy() < zoz2.prvy():
            zoznam.pridaj_kon(zoz1.daj_prvy())
        else:
            zoznam.pridaj_kon(zoz2.daj_prvy())
    while not zoz1.je_prazdny():
        zoznam.pridaj_kon(zoz1.daj_prvy())
    while not zoz2.je_prazdny():
        zoznam.pridaj_kon(zoz2.daj_prvy())
```

Hoci sme tu ušetrili pomocnú pamäť, pravdepodobne toto riešenie stráca na rýchlosti kvôli manipuláciu s triedou `Zoznam` a jej metódami.

8.2.1 Nerekurzívny algoritmus zdola nahor

Užitočnou verziou merge sortu je **nerekurzívny algoritmus zdola nahor**. Opäť pozrieme na obrázok "merge_sort stromu" a aj postup, ako sa postupne zlučovali najprv 1-prvkové polia na 2-prvkové, potom 2-prvkové na 4-prvkové, atď. Nakoniec sa takto dosiahlo utriedenie celého poľa. Presne tento postup využíva nerekurzívny algoritmus zdola nahor:

1. porovnávaj dvojice susedných prvkov poľa: pole[0] a pole[1], pole[2] a pole[3], pole[4] a pole[5], ..., keď v niektorej z dvojíc nie je dobré poradie (prvý nesmie byť väčší ako druhý), tak ich navzájom vymeň - takto dostávame utriedené malé dvojprvkové polia pole[0:2], pole[2:4], pole[4:6], ...
2. zober susedné dvojprvkové polia (sú už utriedené) a zlúč ich do 4-prvkových utriedených, t.j. zlúč pole[0:2], pole[2:4], potom pole[4:6], pole[6:8], ... takto dostávame utriedené 4-prvkové úseky pole[0:4], pole[4:8], pole[8:12], ...
3. rob toto isté ale s väčším krokom `k=4`: zober susedné `k` -prvkové polia a zlúč ich do `2*k` veľkých častí poľa - tento krok opakuj pre `k = 2*k`, kým platí, že `k` je menšie ako počet prvkov celého poľa

Keďže `k` sa tu stále zdvojnásobuje a cyklus končí pre `k >= n` (počet prvkov poľa), zrejme cyklus skončí po **log n** opakovaniach. Ak by sme mali pomocnú funkciu `merge(a,b,c)`, ktorá zlúči úsek poľa v rozsahu `range(a,b)` s úsekom v `range(b,c)`, mohli by sme zapísať:

```
def merge_sort(pole):
    def merge(a, b, c):
        # prvy usek range(a, b), druhy usek range(b, c)
        ...

    n = len(pole)
    k = 1
    while k < n:
        i = 0
        while i + k < n:
            merge(i, i+k, min(i+2*k, n))
            i += 2*k
        k += k
```

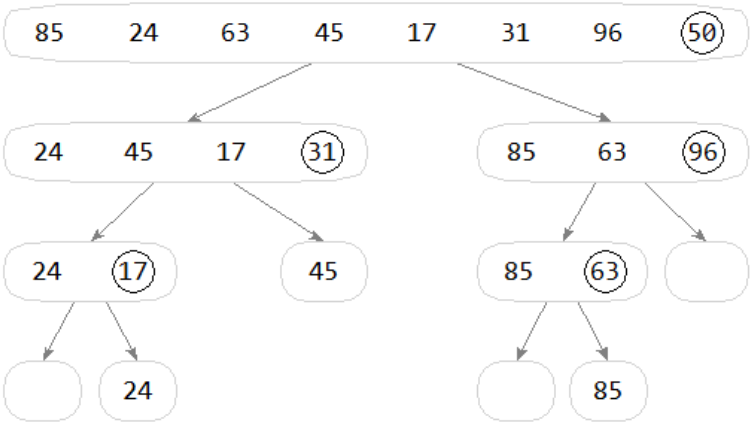
Podobnú ideu by sme vedeli zapísať aj pomocou spájaných zoznamov.

8.3 Quick sort

Toto triedenie poznáme z 1. ročníka (opäť je to princíp "rozdeľuj a panuj"):

1. ak má pole aspoň 2 prvky, zvoľ hodnotu **pivot** a rozdeľ pole na časť menších ako pivot a väčších ako pivot
2. rekurzívne zavolaj triedenie zvlášť pre každú z častí - dostaneme 2 utriedené časti celého poľa
3. spoj obe utriedené časti do výsledného poľa

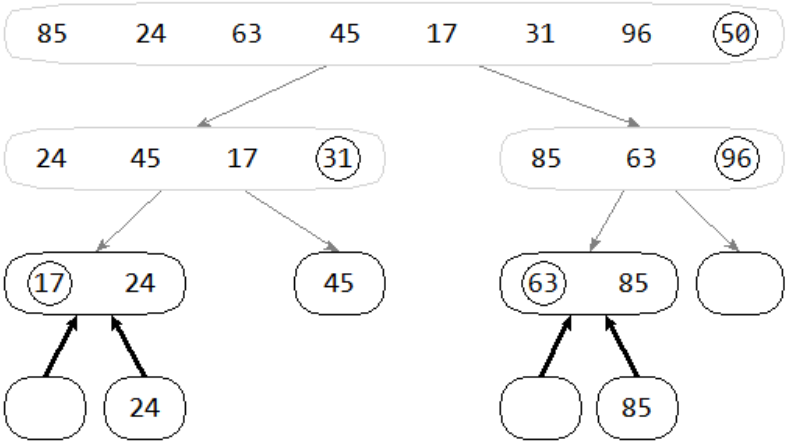
Ideu rozdeľovania poľa na dve časti podľa pivota vidíme:



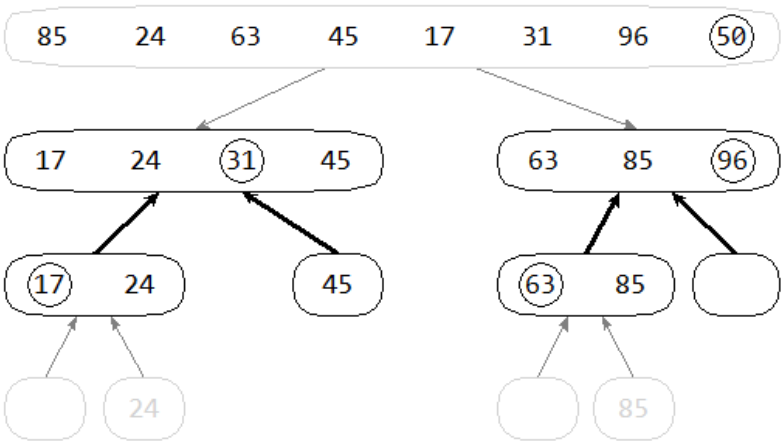
Pivota sme tu vybrali ako posledný prvok (je zakrúžkovaný). Rozdelené dve časti sú už bez tohto pivota (ten sa neskôr objaví vo výsledku v strede medzi nimi).

Toto bola 1. fáza algoritmu, keď sa pole rozdelilo na dve časti (menšie prvky ako pivot a väčšie prvky ako pivot) a pre každú časť sa spustilo rekurzívne volanie, t.j. opäť rozdelenie na dve časti. Toto pokračovalo dovtedy, kým bolo čo deliť, teda 1-prvkové (resp. prázdne) pole sa už ďalej nedelilo.

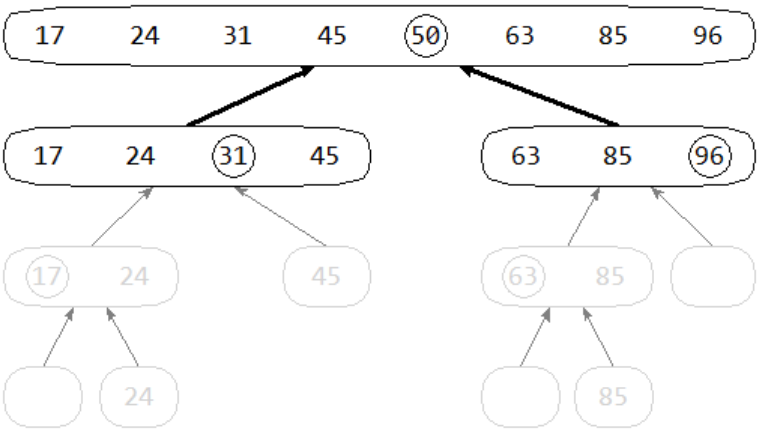
Teraz prichádza fáza návratu z rekurzie, keď sa majú obe časti spojiť do utriedeného celku (medzi ne sa zaradí pivot, ktorý sa v rekurzii netriedil). Vďaka rekurzii sa najprv budú spájať dvojice najvnorenejších častí:



Opäť tento obrázok ukazuje princíp triedenia a nie skutočný algoritmus, lebo tu sme zrealizovali vynáranie z rekurzie naraz vo všetkých prípadoch, keď boli polia jednoprvkové. Pri ďalšom vynáraní sa z rekurzie opäť prebehne fáza spájania, keď sa z utriedených malých častí vytvoria väčšie (opäť sa medzi ne vkladá pivot):



Takto to celé pokračuje, až kým sa postupne nespoja aj časti na najvyššej úrovni. Teda teraz dostávame výsledné utriedené pole:



Zapíšme najprv rekurzívny algoritmus, ktorý ale nie je **in-place**, lebo nemodifikuje vstupné pole, ale vracia nové utriedené pole:

```
def quick_sort(pole):
    if len(pole) < 2:
        return pole
    pivot = pole[0]           # resp. pole[-1], alebo ina hodnota
    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_sort(mensie) + rovne + quick_sort(vacsie)
```

Táto verzia quick sortu je veľmi prehľadná a môže slúžiť ako osnova rôznym iným verziám tohoto triedenia.

Známa (napr. z 1. ročníka) **in-place** verzia tohto triedenia:

```
def quick_sort(pole):
    def quick(z, k):
        if z < k:
            # rozdelenie na dve casti
            index = z
            pivot = pole[z]
```

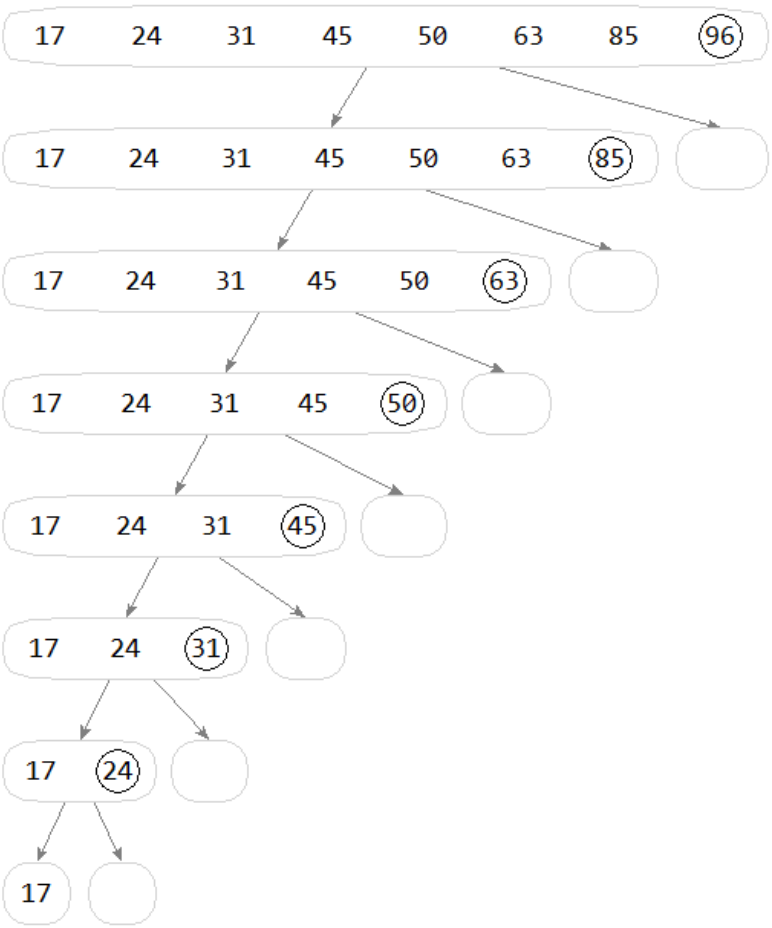
```
            for i in range(z+1, k+1):
                if pole[i] < pivot:
                    index += 1
                    pole[index], pole[i] = pole[i], pole[index]
            pole[index], pole[z] = pole[z], pole[index]
            # v index je pozicia pivota
            quick(z, index-1)
 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

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

Na internete nájdete veľa iných podobných verzií, ktoré realizujú **in-place** triedenie.

Podobne, ako sme vedeli upraviť merge sort pre triedenie spájaného zoznamu, môžeme to zapísať aj pre quick_sort:

```
def quick_sort(zoznam):
    if len(zoznam) < 2:
        return
    pivot = zoznam.prvy()
    mensie = Zoznam()
    rovne = Zoznam()
    vacsie = Zoznam()
    while not zoznam.je_prazdny():
        p = zoznam.daj_prvy()
        if p < pivot:
            mensie.pridaj_kon(p)
        elif p == pivot:
            rovne.pridaj_kon(p)
        else:
            vacsie.pridaj_kon(p)
    quick_sort(mensie)
    quick_sort(vacsie)
    while not mensie.je_prazdny():
        zoznam.pridaj_kon(mensie.daj_prvy())
    while not rovne.je_prazdny():
        zoznam.pridaj_kon(rovne.daj_prvy())
    while not vacsie.je_prazdny():
        zoznam.pridaj_kon(vacsie.daj_prvy())
```

Najväčším problémom quick sortu je jeho veľké riziko: najhorší prípad má kvadratickú zložitosť **O(n**2)**. Toto nastane vtedy, keď pivot nerozdelí pole na dve časti, ale jedna z častí bude prázdna, napr. keď bolo už na začiatku pole utriedené a my vyberáme za pivota, napr. prvý prvok (alebo posledný). Zrejme v tomto prípade sú všetky zvyšné prvky väčšie ako pivot a teda prvá časť rozdelenia poľa na dve časti je prázdna a druhá obsahuje všetky prvky okrem pivota:



Preto je snaha vyberať **dobrého pivota**, ktorý rozdelí pole na dve podobne veľké časti (nemusia byť rovnako veľké, len by nemala byť žiadna z nich prázdna - samozrejme toto je zaujímavé len pre väčšie polia, keď má pole napr. len 4 prvky, často bude jedna z častí prázdna). Stratégií výberu pivota je niekoľko a väčšina z nich je tak dobrá, že ich nemusíme ďalej riešiť, napr.

- náhodný výber pivota (napr. pomocou `random.choice()`)
- priemer prvého a posledného prvku
- priemer prvého, stredného a posledného prvku

Quick sort má výborný výkon (v porovnaní s inými triedeniami) a to hlavne pre veľké polia. Pre malé polia je tu už priveľká strata najmä s obhospodarením rekurzie a manipulácie s pivotmi. V tomto prípade sa využíva hybridný prístup: kým je pole veľké, postupuje sa štandardným quick sortom, ale keď sa príde už na malý úsek (napr. 50 prvkov), použije sa iné, možno "neefektívne" triedenie, ktoré ale pre malé polia môže fungovať veľmi rýchlo - často sa v týchto prípadoch využíva napr. **insert sort**.

8.4 Bucket sort

Všetky triedenia, s ktorými sme sa zatiaľ stretli mali zložitosť buď **O(n**2)** alebo **O(n log n)**. Dá sa ukázať, že triedenie, ktoré pracuje na princípe porovnávania hodnôt prvkov poľa, nemôže byť rýchlejšie ako **O(n log n)**:

- zjednodušená úvaha: máme nejaké triedenie, ktoré triedi `n` prvkové pole, uvažujme, že prvky sú navzájom rôzne

- z triediaceho algoritmu si všímajme len operácie porovnávania dvoch prvkov `pole[i]` a `pole[j]` (napr. `pole[i]<pole[j]`), algoritmus sa pri týchto testoch rozhoduje, ako bude ďalej pokračovať, teda podľa odpovede "ano" alebo "nie"
- zakreslime tieto porovnávania aj s ďalším rozvetvovaním sa do binárneho stromu: prvé takéto porovnanie v algoritme bude v koreni stromu, tu sa to ďalej vetví na nejaké ďalšie dve porovnania (zrejme nejakých iných prvkov), atď.
- keby sme zostavili kompletný strom takýchto rozvetvovaní pri porovnávaní dvojice prvkov poľa, zrejme v listoch takéhoto stromu (tu algoritmus skončil a teda pole už utriedil) budú všetky možné utriedenia `n` prvkového poľa a tých je `n!` (faktoriál)
- takže máme binárny strom, ktorý má `n!` listov, otázkou je, aká minimálna môže byť výška takéhoto stromu (t.j. na minimálne koľko operácií porovnania vieme usporiadať prvky poľa), teda zaujíma nás odhad `log n!`
- matematickými úpravami sa dá ukázať, že

```
log n! > n log n
```

- z tohto môžeme usudzovať, že žiadny algoritmus, ktorý triedi `n` prvkové pole porovnávaním dvojíc hodnôt, nemôže byť efektívnejší ako **O(n log n)**

Za istých podmienok, keď poznáme vlastnosti triedených hodnôt, vieme zostrojiť triedenie, ktoré má zložitosť napr. **O(n)**. Predstavme si situáciu, že o triedených hodnotách (kľúčoch) vieme, že sú to celé čísla z intervalu `<0,K-1>`, teda `K` rôznych hodnôt. Na utriedenie takého poľa použijeme tzv. priehradkové triedenie (bucket sort):

1. priprav `K` prázdnych priehradiek (napr. polia, alebo zoznamy)
2. postupne prejdi všetky prvky vstupného poľa a na základe hodnoty (kľúča) ich zaraď na koniec príslušnej priehradky
3. ak treba, ešte uprav (možno nejako usporiadaj) všetky priehradky
4. spoj priehradky do výsledného poľa

Ak by sa v 3. kroku tieto priehradky ďalej rekurzívne triedili, mali by sme klasické **rozdeľuj a panuj**.

Dôležitou vlastnosťou tu je **stabilita triedenia**: ak majú dva prvky `pole[i]` a `pole[j]` rovnakú hodnotu (kľúč) a pritom `i<j`, tak v utriedenom výslednom poli sa bude pôvodná hodnota `pole[i]` nachádzať pred `pole[j]`. Niektoré verzie vyššie uvedených triedení boli stabilné (napr. insert sort, ale aj prvý quick sort, ktorý ešte nebol **in-place**), iné nie sú stabilné (napr. bubble sort ani selection sort).

Pre bucket sort je stabilita dôležitá napr. vtedy, keď sa táto idea používa pre tzv. **radix sort**. Toto triedenie pracuje na princípe priehradiek, ale vychádza sa z toho, že triedená hodnota (kľúč) sa skladá z viacerých zložiek, pričom každá z nich má relatívne malý interval hodnôt. Napr. kľúčom je 6-ciferné číslo (číslo z intervalu 0 a 999999, zložkami sú cifry čísla), kľúčom je znakový reťazec ohraničenej veľkosti (napr. max. 10 znakov, zložkami sú znaky znakového reťazca), kľúčom je dvojica súradníc, z ktorých každá je z intervalu -100 a 100, atď.

1. predpokladáme, že kľúč sa skladá z `k` zložiek (prípadne je doplnený nulovými hodnotami zľava alebo sprava podľa významu)
2. priprav toľko priehradiek, koľko rôznych hodnôt má `k` -ta zložka kľúča (posledná cifra, posledný znak, ...)
3. postupne presuň všetky prvky poľa do príslušných priečinkov (vždy na koniec momentálneho obsahu) - rozhoduj sa len na základe `k` -tej zložky
4. postupne spoj všetky neprázdne priečinky do jedného celku, prejdi na predposlednú zložku (`k=k-1`) a opakuj od 2. kroku
5. keď prešiel tento postup pre všetky zložky, po poslednom spojení všetkých priečinkom máme utriedené celé pole

Jeden prechod tohto algoritmu má zložitosť **O(n)**, lebo každý prvok sa najprv presunie na koniec nejakého priečinka (čo je **O(1)**), to je spolu **O(n)** a potom sa všetky priečinky spoja do jedného celku, čo nebude zložitejšie ako **O(n)**. Keďže sa toto opakuje pre `k` krát pre všetky zložky, celková zložitosť radix sortu je **O(k * n)**, pre malé `k` je to **O(n)**

8.5 Hľadanie prvku v poli

V praxi sa stretáme s úlohami, v ktorých máme nejaké `n` prvkové neutriedené pole a my potrebujeme čo najrýchlejšie nájsť v poradí `k` -ty najmenší (alebo najväčší) prvok poľa. Pre `k=0` to znamená nájsť minimum, čo vieme urobiť so zložitosťou **O(n)**. Podobne pre `k=1` vieme nájsť druhé minimum na **O(n)**, ale už je to trochu zložitejšie ako prvé minimum. Tiež pre `k=n-1` je nájdenie maxima jednoduchá úloha so zložitosťou **O(n)**. Ale ako je to vo všeobecnosti s nájdením `k` -teho najmenšieho prvku? Ak by sme použili nejaké rýchle triedenie, môžeme zapísať:

```
kty_prvok = sorted(pole)[k]
```

Zrejne takýto algoritmus má už zložitosť **O(n log n)**.

Dá sa to ale aj rýchlejšie: jedna z možností je využiť ideu rekurzívneho quick sortu:

```
def select(pole, k):
    if len(pole) == 1:
        return pole[0]
    pivot = random.choice(pole)
    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]
    # ak sa k nachadza v mensie (teda k < len(mensie)), rekurzivna volaj␣
↪select(mensie, k)
    # inak ak sa k nachadza v rovne (teda k < len(mensie)+len(rovne)),␣
↪return pivot
    # inak nachadza sa vo vacsie teda rekurzivne volaj select(vacsie, k-
↪len(mensie)-len(rovne))
```

Tento algoritmus sa podobne ako quick sort rekurzívne vnára len do jednej z častí (`mensie` alebo `vacsie`), len do tej do ktorej patrí dané `k`. Zložitosť tohto algoritmu môžeme počítať podobne ako pri quick sorte, kde sme uvažovali s približne polovičným rozdelením poľa na dve časti. V prvom prechode treba prejsť všetky prvky poľa, aby sme ich rozdelili do príslušných pomocných polí, teda zložitosť je `n`. V druhom rekurzívnom volaní má pole približne polovicu, teda `n/2`. Každom ďalšom je to približne polovica z predchádzajúcej dĺžky. Toto skončí buď nájdením prvku (v časti `rovne`) alebo keď má celé pole dĺžku 1. Vieme zapísať celkový počet:

```
spolu = n + n/2 + n/4 + n/8 + ... + 1
```

Z matematiky vieme, že sa toto blíži k `2*n`, teda celková zložitosť algoritmu `select()` je **O(n)**.

8.6 Cvičenie

1*. Zapíšte rekurzívny in-place merge sort (s pomocným poľom pri zlučovaní) a otestujte jeho rýchlosť (pre rôzne veľké náhodné polia) a tiež, zistite, či je to stabilný sort

2*. Zapíšte nerekurzívny merge sort zdola nahor a otestujte jeho rýchlosť v porovnaní s rekurzívnym algoritmom v (1)

3. Zapíšte a otestujte rekurzívny merge sort, ktorý triedi spájaný zoznam (`Zoznam = class: ...`).

4*. Otestujte oba rekurzívne quick sorty (z prednášky) či sú stabilné triedenia

5*. Otestovať, či quick sort s pivotom na začiatku (resp. na konci) úseku je naozaj problémový pre utriedené pole: toto je prípad, keď má tento algoritmus zložitosť **O(n**2)**, vaše testovanie by to malo potvrdiť

6*. Porovnajte rýchlosť triedenia, ak sa pivot vyberá náhodne, teda či výber pivota pomocou `random.choice(pole)` príliš nezdržuje oproti voľbe pivota ako `pole[0]`

7*. Zrealizujte quick sort bez rekurzie pomocou vlastného zásobníka:

- do zásobníku treba ukladať dvojicu (začiatok, koniec úseku)
- do zásobníka najprv vložíme väčší z dvoch úsekov na triedenie a potom až menší z nich (zamyslite sa prečo)

8*. Otestujte, ako sa zmení rýchlosť quick sortu, ak sa bude úsek menší ako nejaké `X` triediť pomocou `insert_sort(pole)` - tento insert sort bude trochu upravený tak, aby sa dal použiť na triedenie len zadaného úseku

- otestujte pre rôzne zvolenú veľkosť `X`, napr. 10, 50, 100, ... a to pre to isté veľké náhodné pole

9*. Do quick sortu pridajte ďalší parameter `key`, ktorý má rovnaký význam ako v štandardnej funkcii `sorted()`: týmto parametrom je to funkcia, ktorá sa pri triedení aplikuje na každý prvok, aby sa zistilo, aká časť vstupu sa triedi. Napr.

- keď chceme triediť dvojice hodnôt len podľa druhého prvku (napr. výška), zapíšeme

```
>>> pole = [('Dusan', 180), ('Elena', 166), ('Boris', 199), ('Zuza',␣
↪181)]
>>> quick_sort(pole, key=lambda x: x[1])
>>> pole
[('Elena', 166), ('Dusan', 180), ('Zuza', 181), ('Boris', 199)]
```

10. Zrealizujte quick sort triedením spájaného zoznamu

KAPITOLA 9

Spracovanie textov

9.1 Hľadanie podreťazca

štandardná metóda `str.find()`:

```
>>> 'python'.find('th')
2
>>> 'python'.find('ty')
-1
```

je to rýchle:

```
>>> ('a'*1000000).find('a'*10000+'b')
-1
```

ale toto už trvá citeľne dlhšie:

```
>>> ('a'*1000000).find('a'*10000+'b'+'a'*10000)
-1
>>> ('a'*1000000).find('a'*100000+'b'+'a'*100000)
-1
```

Prečo? Skúsme naprogramovať túto metódu - použijeme tzv. hrubú silu (brute force):

```
def hladaj(ret, pod):                # str.find(sub)
    n, m = len(ret), len(pod)
    for i in range(n-m+1):
        j = 0
        while j < m and ret[i+j] == pod[j]:
            j += 1
        if j == m:
            return i
    return -1
```

Funkcia hľadá najľavejší výskyt reťazca `pod` v nejakom reťazci `ret`. To čo sme skúšali s `str.find()` už pomocou našej funkcie `hladaj()` trvá nepoužiteľne pomaly. Už len toto:

```
>>> hladaj('a'*10000, 'a'*500+'b'+'a'*500)
-1
```

trvá niekoľko sekúnd.

Zložitosť nášho algoritmu `hladaj()` je **O(n*m)**, kde `n` je dĺžka reťazca `ret` a `m` je dĺžka hľadaného podreťazca `pod`. Pre veľké `m` blížiace sa k `n` je to vlastne kvadratické a teda dosť pomalé.

9.2 Knuth-Morris-Pratt

Algoritmus **KMP** publikovali 1976:

```
def hladaj_kmp(ret, pod):
    n, m = len(ret), len(pod)
    if m == 0:
        return 0
    skok = pocitaj_kmp(pod)
    #print(*skok)
    j = k = 0
    while j < n:
        if ret[j] == pod[k]:
            if k == m-1:
                return j-m+1
            j += 1
            k += 1
        elif k > 0:
            k = skok[k-1]
        else:
            j += 1
    return -1
```

Aby toto fungovalo, konštruuje sa špeciálna tabuľka rýchlych skokov:

```
def pocitaj_kmp(p):
    m = len(p)
    skok = [0] * m
    j, k = 1, 0
    while j < m:
        if p[j] == p[k]:
            skok[j] = k+1
            j += 1
            k += 1
        elif k > 0:
            k = skok[k-1]
        else:
            j += 1
    return skok
```

Vďaka tomuto sú niektoré vyhľadávania veľmi rýchle, napr.

```
>>> hladaj_kmp('a'*1000000, 'a'*10000+'b'+'a'*10000)
-1
```

je rýchlejšie ako pythonovská štandardná metóda:

```
>>> ('a'*1000000).find('a'*10000+'b'+'a'*10000)
-1
```

Zložitosť algoritmu KMP je **O(m+n)**. Keď porovnáme s algoritmom hrubej sily, v ktorom sme niektoré znaky reťazca `ret` porovnávali s reťazcom `pod` veľakrát, v algoritme KMP sa index `j` do poľa `ret` iba zvyšuje o 1 a nikdy sa nevracia späť (ako pri hrubej sile), t.j. každý znak reťazca `ret` sa spracuje len raz.

9.3 Kompresia

Rôzne postupnosti znakov treba niekedy čo najviac zhustiť do čo najmenšej postupnosti bitov. Napr.

```
'aaabacaabada'
```

môžeme pomocou `ord()` zakódovať do 12 * 8 bitov (každý ASCII znak má 8 bitov), teda **96**. Keďže sú to len 4 rôzne znaky, vieme každý z nich zakódovať do 2 bitov, napr. takto

```
'a'  00
'b'  01
'c'  10
`d`  11

'aaabacaabada' => 000000010010000010001100
```

Teraz by náš skomprimovaný reťazec mal iba 12 * 2 bity, teda spolu **24** bitov. Lenže `'a'` sa v reťazci vyskytuje výrazne častejšie ako zvyšné znaky. Možno by sme ho mohli zakódovať len 1 bitom a zvyšné môžu mať prípadne bitov viac, napr. takto

```
'a'  0
'b'  10
'c'  110
`d`  111

'aaabacaabada' => 000100110001001110
```

V tomto prípade sme to skomprimovali na **18** bitov. Teraz treba dať pozor, aby sme zvolili také kódovanie, ktoré budeme vedieť spätne jednoznačne rozkódovať:

- nemalo by sa stať, že kód jedného znaku je spoločný nejakému začiatku (prefixu) iného kódu

Zrejme časté znaky by mohli byť kódované menším počtom bitov a zriedkavé môžu byť aj dlhšie. Na toto slúži tzv. **Huffmanovo kódovanie**. Algoritmus pre toto kódovanie pracuje na tomto princípe:

- zoberie sa nejaký text, ktorý reprezentuje reťazce, ktoré sa budú komprimovať týmto kódovaním
- vytvorí sa frekvenčná tabuľka `ft[c]` výskytov jednotlivých znakov
- pripraví sa prázdny prioritný front `q`
- pre každý znak z množiny vstupných znakov sa vytvorí binárny strom s jediným vrcholom, ktorý je koreňom a zároveň listom stromu, a obsahuje tento znak, potom sa dvojica `ft[c]` a tento jednovrcholový strom vloží do prioritného frontu `q`, pričom frekvenčná hodnota je kľúčom a strom je príslušnou hodnotou
- prioritný front teraz obsahuje dvojice (číslo, strom) usporiadané od najmenšieho čísla (najmenšieho počtu výskytov)
- v ďalšej fáze algoritmu sa z `q` vyberajú vždy prvé dva prvky: z týchto dvoch stromov sa poskladá nový tak, že do koreňa ite súčet ich frekvencií, ľavý podstrom je prvý vybratý strom a pravý podstrom je druhý vybratý strom - do `q` sa teraz vloží nová dvojica: kľúčom je súčet frekvencií a hodnotou je tento nový strom
- toto sa opakuje, kým vo fronte neostane jediný prvok a tým je kompletný binárny strom

Tento strom budeme používať na kódovanie jednotlivých znakov zo vstupu:

- vyhľadáme pozíciu kódovaného znaku od koreňa stromu a cesta k nemu bude kódom: posun vľavo je bit **0** a posun vpravo **1**
- čím je nejaký znak v strome hlbšie, aj jeho cesta je dlhšia a teda aj jeho bitová reprezentácia je dlhšia

- čím je znak v strome vyššie, má kratšiu cestu a teda aj kratšiu bitovú reprezentáciu

Strom sme predsa skonštruovali tak, že najprv sme spolu skladali znaky s malou frekvenciou a nad ne sme stále pridávali znaky s vyššou a vyššou frekvenciou. Samotnú funkciu by sme mohli zapísať napr. takto:

```
def huffman(ret):
    ft = {}
    for c in ret:
        ft[c] = ft.get(c, 0) + 1
    q = PriorityQueue()
    for c in set(ret):
        t = BinTree(c)
        q.add(ft[c], t)
    while len(q) > 1:
        f1, t1 = q.remove_min()
        f2, t2 = q.remove_min()
        t = BinTree(f1+f2, t1.root, t2.root)
        q.add(f1+f2, t)
    f, t = q.remove_min()
    return t
```

Trieda `BinTree` má konštruktor, ktorý vytvorí koreň s danou hodnotou a prípadne nastaví ľavý a pravý podstrom.

9.4 Trie

je stromová dátová štruktúra, v ktorej uchovávame nejakú zadanú množinu reťazcov. Predpokladáme, že v tejto množine budeme potrebovať veľmi rýchlo hľadať buď celé slová alebo ich prefixy (začiatky slov). Táto stromová štruktúra využíva to, že sa každé takto uložené slovo rozloží na písmená a táto postupnosť písmen potom tvorí cesty ku niektorým vrcholom stromu (podobne ako v strome Huffmanovho kódovania). Tejto štruktúre sa hovorí **Trie** ale niekedy aj **prefixový**, **písmenkový** alebo **znakový** strom. Dá sa nájsť aj s názvom **lexikografický** strom.

Každý vrchol tohto stromu má toľko podstromov, koľko rôznych písmen z tohto vrcholu pokračuje. Každá takáto hrana k podstromu je označená príslušným písmenom. Najlepšie to vysvetlíme na príklade: zostrojíme **trie**, ktorý bude uchovávať túto množinu slov: {bear, bell, bid, bull, buy, sell, stock, stop}.

- z koreňa stromu budú vychádzať dve hrany (dva podstromy), lebo všetky slová začínajú písmenom `'b'` alebo `'s'`:
- jeden podstrom bude obsahovať všetky slová na `'b'`: {bear, bell, bid, bull, buy} a druhý slová na `'s'`: {sell, stock, stop}
- podstrom pre `'b'` bude mať toľko synov (podstromov), koľko je rôznych druhých písmen v slovách: sú to `'e'`, `'i'`, `'u'`
  - takže prvý jeho podstrom bude uchovávať slová s `'e'`: {bear, bell}
  - druhý slová s `'i'`: {bid}
  - tretí s `'u'`: {bull, buy}
- podobne podstrom pre prvé písmeno `'s'` sa bude rozvetvovať podľa druhých písmen slov {sell, stock, stop}, teda na dva podstromy `'e'`, `'t'`:
  - jeho prvý podstrom bude uchovávať slová, ktoré začínajú `'s'` a druhé písmeno je `'e'`: {sell}
  - druhý slová s druhým písmenom `'t'`: {stock, stop}
- takto by sme pokračovali, kým by sme nerozobrali na písmená všetky slová

Zrejme výška stromu je daná dĺžkou najdlhšieho slova, ktoré je uložené v strome, v našom prípade je to slovo `'stock'`, teda výška je 5. V tomto prípade všetky slová končia v listoch stromu, lebo žiadne z nich nie je prefixom nejakého iného. Toto je v niektorých aplikáciách **trie** podmienkou: všetky slová musia končiť v listoch stromu a teda žiadne nie je prefixom iného. Inokedy nám to nevadí a vtedy treba nejakú informáciu o tom, že nejaký vnútorný vrchol stromu je koncovým písmenkom nejakého slova. Ak by sme do tohto stromu chceli vložiť aj anglické slovo 'be', museli by sme príslušný vrchol (z ktorého pokračujú zvyšné písmená pre `'bear'` a `'bell'`) nejako označiť (atribút s príznakom konca) alebo by sme na koniec každého slova prilepili nejaký špeciálny koncový znak napr. `'#'`.

Najčastejšie vrchol prefixového stromu definujeme takto:

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

teda každý vrchol môže obsahovať nejakú informáciu (napr. že je koncový nejakého slova) a asociatívne pole všetkých svojich podstromov.

Vložiť nové slovo do stromu znamená:

```
def vloz(slovo):
    vrch = ... # koreň stromu
    for znak in slovo:
        if znak not in vrch.pole:    # este nemam podstrom
            vrch.pole[znak] = Vrchol()
        vrch = vrch.pole[znak]
    vrch.data = ... # je to koncovy vrchol slova
```

Pomocou rekurzívnej funkcie vieme vygenerovať všetky uložené slová v takomto strome:

```
def vsetky(vrch, slovo):
    if vrch is not None:
        if vrch.data == ...:    # koncovy vrchol slova
            yield slovo
        for znak in vrch.pole:
            vsetky(vrch.pole[znak], slovo+znak)

for slovo in vsetky(koren, ''):
    print(slovo)
```

9.5 Cvičenie

1. Merajte čas trvania štandardenj metódy `str.find()` pri práci s dlhými reťazcami, ktoré obsahujú len písmená 'a' a 'b' (podobne ako v prednáške) - reťazec, v ktorom hľadáme nech má aspoň 1000000 znakov, hľadaný reťazec nech má dĺžky od 1000 do 100000.
   - napr. čas trvania príkazov:

     ```
     >>> ('a'*1000000).find('a'*10000+'b')
     >>> ('a'*1000000).find('a'*10000+'b'+'a'*10000)
     ```

   - porovnajte s algoritmom, ktorý hľadá hrubou silou (asi bude treba testovať kratšie reťazce)
2. Skúmajte, ako funguje algoritmus `hladaj_kmp()` napr. pre reťazce 'abababadababacabab' a 'ababaca':

- vypíšte zostavené pomocné pole `skok`

- pri každom prechode vnútorným while-cyklom vypíšte reťazec `ret` pod ním reťazec `pod` ale odsunutý tak, aby zodpovedal momentálnym porovnávaným znakom (posun o `j-k` vpravo) a pod týmto riadkom riadok, v ktorom bude jediný znak '^' na pozícii porovnávaných znakov (t.j. `j`)

3. Rovnaké testy ako v (1) spustite aj pre algoritmus hľadania **KMP** (bez trasovania z (2))
4. Zostavte (ručne na papieri) strom Huffmanovho kódovania pre slová nejakej konkrétnej vety , napr. 'mama ma emu a ema ma mamu'

- vytvorte kódovaciu tabuľku pre každé písmeno z tejto vety (t.j. aká postupnosť 0 a 1 zodpovedá, ktorému písmenu)
- zakódujte (ako postupnosť bitov) celú vstupnú vetu

5. Spojazdnite huffmanov algoritmus z prednášky ak

```
import tkinter

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

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

    #--------------------------------------------------------------------

    def __init__(self, root_data=None, root_left=None, root_right=None):
        self.root = None
        if root_data is not None:
            self.root = self.Node(root_data, root_left, root_right)


    canvas_width = 1200
    canvas = None

    def draw(self, node=None, width=None, x=None, y=None):
        def vstrede(x, y, z):
            self.canvas.create_oval(x-5,y-5,x+5,y+5,outline='',fill='white')
            self.canvas.create_text(x, y, text=z)
        if self.canvas is None:
            self.canvas = tkinter.Canvas(bg='white', width=self.canvas_
↪width, height=600)
            self.canvas.pack()
        if node is None:
            self.canvas.delete('all')
            if self.root is None:
                return
            node = self.root
            if width is None: width = int(self.canvas['width'])//2
            if x is None: x = width
            if y is None: y = 30
        if node.left is not None:
            self.canvas.create_line(x, y, x - width//2, y + 50)
            vstrede(x-width//4, y+25, 0)
            self.draw(node.left, width//2, x - width//2, y + 50)
        if node.right is not None:
```

```
            self.canvas.create_line(x, y, x + width//2, y + 50)
            vstrede(x+width//4, y+25, 1)
            self.draw(node.right, width//2, x + width//2, y + 50)
        self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
        f = 'consolas 12 bold' if node.left is None else 'consolas 10'
        self.canvas.create_text(x, y, text=node, font=f)
        if node is self.root:
            self.canvas.update()
            self.canvas.after(100)
```

```
import heapq

class PriorityQueue:

    class Item:
        def __init__(self, key, value):
            self.key, self.value = key, value

        def __lt__(self, other):
            return self.key < other.key


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

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

    def add(self, key, value):
        heapq.heappush(self.pole, self.Item(key, value))

    def remove_min(self):
        item = heapq.heappop(self.pole)
        return item.key, item.value
```

- otestujte vykreslením stromu z (4)

6. Napíšte funkcie `urob_tab(strom)`, `koduj(tab,slovo)` a `rozkoduj(tab,kod)`, ktoré

- z huffmanovho stromu vyrobia kódovaciu tabuľku (asociatívne pole: pre každé písmeno zistia postupnosť 0 a 1)
- zakódujú slovo (postupnosť písmen) podľa danej tabuľky
- rozkódujú slovo

7. Zostavte (ručne na papieri) prefixový strom pre množinu slov

```
{auto, aula, ano, body, byk, boja, balet, cop, cap, cip, cakan}
```

8. Dopropgramujte triedu `Trie`:

```
class Trie:
    class Vrchol:
        ...

    def __init__(self):
        self.koren = self.Vrchol()
```

```
    def vloz(self, slovo):
        ...

    def __iter__(self):
        ...
```

- vytvorte strom pre slová z (7) - prípadne dopíšte metódu, ktorá ho vykreslí

9. Zadefinujte triedu asociatívne pole realizovanú pomocou **Trie**: predpokladáme, že kľúčami budú iba reťazce, hodnotami k daným kľúčom budú ľubovoľné hodnoty okrem `None` - vložíte ich do príslušného vrcholu ako `vrchol.data`

- otestujte to tak, že prečítate nejaký veľký text (napr. z 5. cvičenia) a vytvoríte frekvenčnú tabuľku výskytu slov z tohto text = výsledok porovnáte riešením pomocou pythonovského asociatívneho poľa (`dict`)

KAPITOLA 10

Dynamické programovanie

Minulá prednáška sa zaoberala algoritmami so znakovými reťazcami. Riešenie jedného dôležitého problému sme ale vynechali:

- **LCS** (longest common subsequence) - hľadanie najdlhšej (vybranej) spoločnej podpostupnosti dvoch reťazcov (resp. dvoch polí)
- predpokladajme, že máme dané 2 reťazce `x` a `y` s dĺžkami `n` a `m` znakov, úlohou je vybrať čo najdlhšiu podpostupnosť znakov z prvého reťazca (znaky sa nemuseli nachádzať tesne vedľa seba, len ich z postupne vyberáme zľava doprava) takú, že sa celá nachádza v druhom reťazci (tiež to nemuseli byť znaky tesne vedľa seba, len sa v reťazci nachádzali postupne na pozíciách zľava doprava)
- napr. reťazce `'programovanie'` a `'krasokorculovanie'` majú najdlhší spoločný podreťazec `'rorovanie'`

Doteraz by sme túto úlohu riešili asi hrubou silou:

1. postupne by sme z prvého reťazca generovali všetky podpostupnosti znakov (asi od najdlhších po najkratšie)
2. pre každý jeden vygenerovaný reťazec by sme skontrolovali, či sa celý presne v tomto poradí znakov nachádza v druhom reťazci
3. ak áno, našli by sme najdlhší a môžeme skončiť

Tento algoritmus má zložitosť **O(2**n)**, lebo všetkých vygenerovaných podreťazcov reťazca dĺžky **n** je **2**n**.

Na riešenie niektorých úloh, v ktorých hľadáme nejaké optimálne riešenie, môžeme niekedy použiť metódu **dynamické programovanie**. Idea sa trochu podobá metóde **rozdeľuj a panuj**, pripomeňme si:

1. ak sa ešte dá rozdeľ veľký problém na (disjunktné) podproblémy
2. rekurzívne vyrieš každý z podproblémov
3. spoj dokopy všetky tieto čiastkové riešenia do riešenia celého problému

Videli sme napr. ako sa táto metóda využila pre **merge sort** aj **quick sort**. Uvedomte si, že pri tom sa problém rieši **zhora nadol**.

Metóda **dynamické programovanie** postupuje opačne:

1. vyriešime najjednoduchšie prípady zložitého problému - riešenia si zapamätáme najlepšie v nejakej tabuľke (podporblémy nemusia byť navzájom disjunktné - môžu sa navzájom prekrývať)
2. z týchto jednoduchých riešení postupne skladáme stále zložitejšie a zložitejšie riešenia
3. takto pokračujeme, kým sa nedostaneme k riešeniu požadovaného problému

Táto metóda funguje na princípe **zdola nahor** a väčšinou využíva pomocnú pmaäť na pamätanie si čiastkových riešení - často sú to jednorozmerné alebo dvojrozmerné tabuľky (veľkosti **O(n)** alebo **O(n**2)**). Vďaka tejto metóde sa niekedy podarí z problému exponenciálnej zložitosti **O(2**n)** vyrobiť riešenie zložitosti **O(n)** alebo **O(n**2)**.

Na riešenie úloh pomocou dynamického programovania treba mať veľkú skúsenosť s analýzou problému, rozložením na podproblémy, vymyslením, ako skladať z riešení malých podproblémov väčšie, ako využívať tabuľky. Ukážeme to na sérii jednoduchých úloh: mnohé z nich sme už veľakrát riešili, ale netušili sme, že za nimi sú skryté princípy dynamického programovania.

10.1 Fibonacciho postupnosť

Poznáme rekurzívnu verziu funkcie:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Už vieme, že táto funkcia má exponenciálnu zložitosť.

Ak by sme najprv postupne (bez rekurzie) vypočítali do tabuľky prvých `n` členov postupnosti a potom `n` -tý vrátili ako výsledok funkcie, zložitosť tohto zápisu by bola len **O(n)**. Hoci v tomto prípade sme tiež minuli **O(n)** pomocnej pamäte:

```
def fib(n):
    tab = [0, 1]
    for i in range(2, n+1):
        tab.append(tab[i-1] + tab[i-2])
    return tab[n]
```

Toto je najjednoduchší prípad použitia **dynamického programovania**: veľký problém (zložitosti **O(2**n)**) sme pomocou postupného zostavovania tabuľky (v nej sme riešili jednoduchšie podproblémy a z nich sme zostavovali stále zložitejšie a zložitejšie riešenia) previedli na úlohu zložitosti **O(n)**. Vidíme tu, že úloha sa rieši metódou **zdola nahor**, t.j. najprv najjednoduchšie podúlohy a potom stále zložitejšie, ktoré sa blížia k požadovanému problému.

Fibonacciho postupnosť tiež vieme vyriešiť aj bez rekurzie a pomocnej tabuľky, napr.

```
def fib(n):
    f1, f2 = 1, 0
    for i in range(n):
        f1, f2 = f2, f1+f2
    return f2
```

V tomto prípade tiež riešime rekurzívnu úlohu postupným riešením najprv najjednoduchších podproblémov a z nich zostavujeme zložitejšie riešenia - tu sme tabuľku veľkosti **n** nahradili len dvomi pomocnými premennými.

Veľakrát sa aj pri riešení iných úloh pomocou dynamického programovania ukáže, že nemusíme zostavovať kompletnú dvojrozmernú tabuľku, ale bude nám z nej stačiť len posledný riadok (prípadne posledné dva).

V 2. cvičení sme presne túto istú úlohu riešili pomocou **memoizácie**:

- je to spôsob, ktorým si funkcia pamätá už raz vypočítané výsledky a keď ju voláme druhýkrát, tak už nič nepočíta, len zapamätanú hodnotu vráti ako svoj výsledok
- vďaka tomu nemusíme funkciu prerábať na nerekurzívnu verziu s vytváraním tabuľky, ale tabuľka sa vytvára automaticky

Potom `fib()` vyzerá takto:

```
def fib(n, mem={}):
    if n in mem:
        return mem[n]
    if n < 2:
        mem[i] = n
        return n
    res = fib(n-1) + fib(n-2)
    mem[n] = res
    return res
```

Vidíme, že toto riešenie je **zhora nadol**, ale sa v ňom tiež vytvára tabuľka s riešeniami všetkých podúloh. Hovoríme, že **memoizácia** je špeciálnym prípadom **dynamického programovania**. Využije sa hlavne pri jednoduchších úlohách, ktoré vieme zapísať rekurzívne.

10.2 Kombinačné čísla

Kombinačné čísla vieme vypočítať aj pomocou rekurzívneho vzťahu:

```
komb(n, k) = komb(n-1, k) + komb(n-1, k-1)
```

Tento vzťah sa dá jednoducho zapísať pomocou rekurzie (pre triviálne prípady `k=0` a `n=k`), pričom zložitosť takejto funkcie je **O(2**n)**.

Zo strednej školy poznáme **Pascalov trojuholník**, ktorý je poskladaný z hodnôt kombinačných čísel:

```
             1
          1     1
       1     2     1
    1     3     3     1
 1     4     6     4     1
1   5    10    10    5    1
1   6    15    20    15    6    1
```

čo sú:

```
                        komb(0,0)
                    komb(1,0) komb(1,1)
                komb(2,0) komb(2,1) komb(2,2)
            komb(3,0) komb(3,1) komb(3,2) komb(3,3)
        komb(4,0) komb(4,1) komb(4,2) komb(4,3) komb(4,4)
...
```

Každé číslo (okrem krajných 1) sa dá vypočítať ako súčet dvoch čísel nad ním (vľavo a vpravo). Takže, ak by sme namiesto rekurzie zostavovali tieto hodnoty do dvojrozmernej tabuľky, použili by sme **dynamické programovanie**: postupným zapĺňaním tabuľky riešime jednoduchšie podúlohy, pričom využívame riešenia ešte jednoduchších podúloh.

Samozrejme, že aj táto úloha sa dá riešiť memoizáciou: aj v tomto prípade je to dynamické programovania, ale tabuľku za nás zapĺňa Python rekurzívnych volaniach.

10.3 Najdlhšia (vybraná) spoločná podpostupnosť

Algoritmus **LCS** (longest common subsequence) je typickým predstaviteľom **dynamického programovania**. Aby sme mohli túto úlohu riešiť touto metódou, musíme vymyslieť, ako na to využijeme postupné budovanie (možno

dvojrozmernej) tabuľky: čo si budeme v tejto tabuľke pamätať (aké podúlohy) a ako sa z nich budú dať vytvárať riešenia zložitejších podúloh. Na toto treba získať istú skúsenosť riešením veľkého množstva podobných úloh - v tejto prednáške začínate získavať prvé skúsenosti...

Na riešenie využijeme dvojrozmernú tabuľku, v ktorej si budeme uchovávať informácie o dĺžkach LCS, ale len pre nejaké podreťazce (teda nie pre celé reťazce `x` a `y`). Predpokladajme, že reťazec `x` má dĺžku `n` (s indexmi od 0 do `n-1`) a reťazec `y` má dĺžku `m` (s indexmi od 0 do `m-1`). Potom v tabuľke `tab[i][j]` bude dĺžka LCS (najdlhšej spoločnej podpostupnosti) ale vypočítaná len pre časť reťazca `x[:i]` a časť reťazca `y[:j]`. Preto:

- nultý riadok tabuľky označuje, že z reťazca `x` berieme len rez `x[:0]`, t.j. prázdny reťazec a preto tento riadok obsahuje samé 0 (LCS prázdneho reťazca a reťazca `y` je vždy 0)
- podobne nultý stĺpec tiež obsahuje samé 0
- prvý riadok `tab[1][j]` označuje LCS, ak je `x` iba jednoznakové, teda môže byť buď **0**, kým sú rôzne znaky v `x[0]` a v `y` alebo **1**, ak sme našli prvú zhodu, t.j. pozíciu `j`, v ktorej `x[0]==y[j+1]` - uvedomte si, že od tejto pozície bude mať v tabuľke `tab[1]` všetky hodnoty **1**, lebo LCS `x[0]` a `y[:j]` je 1 (existuje v ňom podreťazec dĺžky 1)
- každý ďalší, teda `i` -ty riadok, budeme postupne počítať takto:
  - ak `x[i]==y[j]`, nám umožní predĺžiť doterajšie LCS o 1, ktoré boli pre podreťazce `x[:i]` a `y[:j]` (teda o 1 kratšie) - ale pre tieto kratšie `x` a `y` už máme vypočítanú dĺžku LCS v našej tabuľke na pozícii `tab[i][j]`, preto nová hodnota v tabuľke `tab[i+1][j+1]` bude `tab[i][j]+1`
  - ak sa `x[i]!=y[j]`, nič sa predlžovať nebude, budeme len kopírovať niektorú z hodnôt v doteraz vytvorenej tabuľke `tab`: zoberieme buď hodnotu `tab[i][j+1]` (o jedna kratšie `x`) alebo `tab[i+1][j]` (o jedna kratšie `y`), zrejme to bude väčšia z týchto hodnôt

Týmto postupom vytvoríme dvojrozmernú tabuľku maximálnych dĺžok LCS pre všetky podreťazce `x` a `y` (ktoré začínajú na index 0) a zrejme hľadaná dĺžka LCS bude v tabuľke na `tab[n][m]`.

Ukážme to na reťazcoch:

```
x = 'abbccbddbd'
y = 'cadbddbbada'
```

Náš postup z nich vytvorí takúto tabuľku:

```
    c a d b d d b b a d a
  0 0 0 0 0 0 0 0 0 0 0 0
a 0 0 1 1 1 1 1 1 1 1 1 1
b 0 0 1 1 2 2 2 2 2 2 2 2
b 0 0 1 1 2 2 2 3 3 3 3 3
c 0 1 1 1 2 2 2 3 3 3 3 3
c 0 1 1 1 2 2 2 3 3 3 3 3
b 0 1 1 1 2 2 2 3 4 4 4 4
d 0 1 1 2 2 3 3 3 4 4 5 5
d 0 1 1 2 2 3 4 4 4 4 5 5
b 0 1 1 2 3 3 4 5 5 5 5 5
d 0 1 1 2 3 4 4 5 5 5 6 6
```

Tu sme do prvého riadka nad príslušné stĺpce tabuľky `tab` zapísali zodpovedajúce znaky reťazca `y`. Podobne sme na začiatok každého riadka zapísali znaky reťazca `x`. Všimnite si, že aj v tomto výpise `i` -temu znaku `x` zodpovedá `i+1` riadok tabuľky a `j` -temu znaku `y` zodpovedá `j+1` stĺpec tabuľky. Tabuľka má zrejme o 1 riadok viac ako je dĺžka reťazca `x` a o 1 stĺpec viac ako je dĺžka reťazca `y`.

Zapíšme funkciu, ktorá vytvára túto tabuľku:

```
def lcs_tab(x, y):
    n, m = len(x), len(y)
```

```
    tab = [[0]*(m+1) for i in range(n+1)]
    for i in range(n):
        for j in range(m):
            if x[i] == y[j]:
                tab[i+1][j+1] = tab[i][j] + 1
            else:
                tab[i+1][j+1] = max(tab[i][j+1], tab[i+1][j])
    return tab
```

Ak by nám stačila informácia o dĺžke LCS, mohli by sme skončiť s hodnotou `tab[n][m]`. Ak ale potrebujeme získať aj konkrétny podreťazec, môžeme z tejto tabuľky toto **zrekonštruovať**:

- začneme v pravom dolnom rohu tabuľky - tu vieme, aká je dĺžka hľadaného podreťazca

```
i, j = len(x), len(y)
```

- hľadáme smerom "hore" (`i-=1`) a "vľavo" (`j-=1`) znak, ktorý majú `x` aj `y` rovnaký - keďže v tabuľke `tab[i][j]` označuje znaky `x[i-1]` a `y[j-1]`, testujeme

```
if x[i-1] == y[j-1]:
```

- v prípade, že nájdeme hľadaný spoločný znak, zaradíme ho do výsledku (na začiatok, lebo výsledok skladáme od konca):

```
ries = x[i-1] + ries
```

- keď sa hýbeme "hore" alebo "vľavo" vyberáme si smer, kde je menšia hodnota, teda porovnávame `tab[i-1][j]` a `tab[i][j-1]`

Funkcia, ktorá z tabuľky vytvorí hľadanú podpostupnosť:

```
def lcs_ries(x, y, tab):
    ries = ''
    i, j = len(x), len(y)
    while tab[i][j] > 0:
        if x[i-1] == y[j-1]:
            ries = x[i-1] + ries
            i -= 1
            j -= 1
        elif tab[i-1][j] >= tab[i][j-1]:
            i -= 1
        else:
            j -= 1
    return ries
```

V našom príklade funkcia vrátireťazec:

```
abddbd
```

10.4 Cvičenie

10.4.1 kombinačné čísla

1. zapíšte a otestujte rekurzívne riešenie komb(n, k) = komb(n-1, k) + komb(n-1, k-1) ... pozor na triviálny prípad

- zistite počet rekurzívnych volaní pre n in (20,21,22,...30) a k=15

- pre tieto volania merajte čas
- odhadnite zložitosť

2. vytvorte program, ktorý najprv skonštruuje dvojrozmernú tabuľku pre n riadkov a k stĺpcov
- merajte čas ako v (1), odhadnite zložitosť
3. riešte ako v (2) ale z tabuľky pracujte len s jedným riadkom (z i-teho vyrobte (i+1)-ty riadok)
- otestujte správnosť
4. riešte memoizáciou, t.j. do rekurzívneho riešenia pridajte pamätanie si vypočítaných hodnôt a ich neskoršie použitie

10.4.2 funkcia P

funkcia `P(i,j)` pre nezáporné `i` a `j` je zadaná takto:

- ak `j=0`, výsledkom je 0
- ak `i=0`, výsledkom je 1
- inak výsledkom je `(P(i-1,j)+P(i,j-1))/2`

5. ručne vytvorte tabuľku veľkosti 4x4 pre hodnoty z `range(4)`
6. zapíšte rekurzívnu funkciu
- skontrolujte správnosť vašej ručnej tabuľky z (5)
- zistite počet rekurzívnych volaní pre i in (10,11,12,...20) a k=10
- pre tieto volania merajte čas
- odhadnite zložitosť
7. vyriešte memoizáciou
8. vyriešte vytvorením dvojrozmernej tabuľky veľkosti `i` x `j`, ktorú vyplníte po riadkoch bez rekurzie
- skontrolujte správnosť vašej ručnej tabuľky z (5)
- odhadnite zložitosť funkcie
- spustite testy z (6)

10.4.3 LCS

9*. spojazdnite algoritmy z prednášky

- otestujte na slovách `'programovanie'` a `'krasokorculovanie'`: vypíšte tabuľku a skontrolujte výsledok
- otestujte: náhodne vygenerujte dve DNA postupnosti (z písmen {A,C,G,T}) dĺžky 100 a zistite ich LCS
- otestujte: zo súborov `'text3.txt'` a `'text4.txt'` (z 5. cvičenia) prečítajte po 1000 znakov ale až od 100. znaku, zistite LCS

10*. ručne vytvorte tabuľku pre reťazce `'aabaacdb'` a `'abcabcabc'`

10.4.4 najdlhšia vybraná rastúca podpostupnosť

Úloha sa rieši podobne ako LCS:

- z postupnosti čísel treba nájsť najdlhšiu vybranú podpostupnosť, ktorá je rastúca
- napr. pre `[3,7,2,9,4,1,5,6,10,8,0]` je takou `[2,4,5,6,8]` ale aj `[3,4,5,6,10]`

Rieši sa pomocou dynamického programovania napr. takto:

- postupne sa zostavuje tabuľka `tab`, v ktorej `i` -ty prvok obsahuje dĺžku najdlhšej vybranej rastúcej podpostupnosti, ktorá **končí** `i`-tym prvkom, pričom ju počítame len z prvých `i` prvkov, t.j.
  - `tab[0]` je zrejme 1, lebo ak berieme prvky poľa len po 0, tak tento je 1-prvkovou postupnosťou
  - `tab[1]` je dĺžkou najdlhšej rastúcej podpostupnosti, ak berieme do úvahy len 0. a 1. prvok poľa a pritom vybraná podpostupnosť končí 1. prvkom: buď je to 1 alebo 2
  - `tab[2]` je opäť dĺžkou najdlhšej, ak berieme len prvé 3 prvky, pričom určite končí 3. prvkom
  - ...
- dynamicky toto pole `tab` budete vytvárať takto:
  - pre `tab[i]` nájdeme najväčšiu hodnotu medzi `tab[0:i-1]` (na indexe `j`), pre ktorú je `pole[j]<pole[i]` - do `tab[i]` zapíšeme túto najväčšiu hodnotu zvýšenú o 1
  - ak medzi `tab[0:i-1]` nie je žiadna, pre ktorú by platilo `pole[j]<pole[i]`, do `tab[i]` zapíšeme 1 (zrejme `pole[i]` nie je väčšie ako žiadne z `pole[0:i-1]`)
- z takto zostavenej tabuľky `tab` vieme zistiť hľadanú dĺžku podpostupnosti (je to maximálna hodnota v tabuľke) a tiež vieme zrekonštruovať hľadanú podpostupnosť:
  - v `tab` pôjdeme od konca a postupne hľadáme prvý výskyt pozícií s jednotlivými dĺžkami (najprv max, potom max-1, max-2, ..., 2, 1)

11*. ručne vytvorte tabuľku pre pole `[3,7,2,9,4,1,5,6,10,8,0]`

- potom zrekonštruujte z tejto tabuľky hľadanú najdlhšiu rastúcu podpostupnosť

12*. zapíšte a odlaďte funkciu, ktorá hľadá najdlhšiu vybranú rastúcu podpostupnosť

- funkcia vráti ako výsledok túto podpostupnosť
- odhadnite zložitosť algoritmu
- otestujte aj na vzostupne a tiež zostupne utriedených poliach
- otestujte na náhodne vygenerovaných postupnostiach čísel
- otestujte na 100-prvkovom poli slov, ktoré získate zo súboru `'text3.txt'` z (9) úlohy

10.4.5 mincovka

Známa úloha o rozmieňaní nejakej sumy na čo najmenší počet mincí. Zrejme musíme poznať, aké druhy mincí máme k dispozícii, napr. ak máme mince `[1,2,5,10]`, tak sumu `6` vieme rozmeniť napr. ako `2+2+2` alebo `1+5` ale aj `1+1+1+1+1+1`. V tomto prípade je `1+5` hľadaným riešením, lebo na rozmenenie sme použili najmenší počet mincí (zrejme menej ako dvomi mincami sa to nedá).

13. Vyriešte túto úlohu pomocou tzv. **greedy** metódy:
    - na rozmieňanie sa snažíme použiť čo najväčšiu mincu, ktorou sa to ešte dá
    - túto mincu si zapamätáme, odčítame ju od rozmieňanej sumy a celé to opakujeme, kým je rozmieňaná suma väčšia ako 0

```
def mincovka1(mince, suma):
    ...
    return # zoznam mincí

for suma in range(1, 21):
    print(suma, mincovka1([1, 2, 5, 10], suma))
```

14. Algoritmus z úlohy (13) funguje dobre len pre niektoré sady mincí, napr. pre mince `[1,3,7,10]` by sumu `14` rozmenil ako `1+3+10` pričom správnym riešením by malo byť `7+7` (teda iba 2 mince). Vyriešte túto úlohu hrubou silou, teda pomocou backtrackingu:
    - funkcia vráti len správny počet mincí, nie zoznam s použitými mincami
    - funkcia bude rekurzívna a má triviálny prípad: keď sa rozmieňaná suma rovná niektorej minci, vtedy funkcia vráti 1
    - inak funkcia postupne vyskúša odobrať po jednej z každej mincí (ktoré nemajú hodnotu vyššiu ako rozmieňaná suma), rekurzívne zistí, na koľko mincí sa dá rozmeniť táto zmenšená suma a najlepší z týchto výsledkov (plus 1) vráti ako výsledok rekurzívnej funkcie

```
def mincovka2(mince, suma):
    if suma in mince:
        return 1
    ...
    return # minimálny počet mincí
```

```
>>> mincovka2([1, 3, 7, 10], 14)
2
```

    - porovnajte výsledky z volaní *mincovka1()‘‘* a `mincovka2()` pre mince `[1,3,4]` a rozmieňané sumy od 1 do 20

15. Rekurzívny algoritmus z úlohy (14) je pre väčšie hodnoty veľmi pomalý (podobne ako rekurzívna fibonacciho funkcia). Môžete ho výrazne urýchliť pomocou **memoizácie**:
    - zadefinujte **globálnu** premennú `mem` ako prázdne asociatívne pole a rekurzívna funkcia `mincovka3()` najprv v tomto poli zistí, či už túto hodnotu nemá vypočítanú a ak áno vráti ju ako výsledok

```
def mincovka3(mince, suma):
    if suma in mem:
        return mem[suma]          # už sme to počítali predtým
    if suma in mince:
        mem[suma] = 1
        return 1
    ...
    mem[suma] = pocet
    return pocet
```

    - odhadnite časovú zložitosť funkcií `mincovka2()` a `mincovka3()`
    - vyskúšajte zavolať obe tieto funkcie aj pre väčšie hodnoty, napr.

```
>>> mincovka2([1,3,7,10,20], 45)
5
>>> mem = {}
>>> mincovka3([1,3,7,10,20], 45)
5
```

- preskúmajte obsah poľa `mem`: čo je v tomto poli v prvku `mem[i]`?

16. Obsah pomocného poľa `mem`, ktoré sa využilo pre memoizáciu môže byť inšpiráciou pre riešenie pomocou ozajstného **dynamického programovania**:
    - funkcia `mincovka4()` najprv zostaví tabuľku podobnú `mem`, ale urobí to bez rekurzie metódou zdola nahor: postupne ju bude zapĺňať zľava doprava, t.j. z hodnôt na menších indexoch vypočíta hodnoty na vyšších
    - prvkami tabuľky budú minimálne počty rozmieňania na mince pre dané sumy, t.j. `tab[suma]` bude minimálny počet mincí, na ktoré sa dá daná suma rozmeniť
    - bude sa pri tom používať podobný cyklus ako v `mincovka2()`, ktorý hľadal minimálny počet mincí rozmieňaní, ale namiesto rekurzie tu bude vytiahnutie hodnoty z tabuľky `tab`

```
def mincovka4(mince, suma):
    tab = [0] * (suma+1)
    for s in range(1, suma+1):
        ...
    return tab[suma]
```

    - odhadnite časovú zložitosť algoritmu

17. Dopíšte do funkcie `mincovka4()` záverečnú časť, ktorá na základe tabuľky `tab` zrekonštruuje zoznam použitých mincí a funkcia potom namiesto počtu mincí vráti zoznam (pole) použitých mincí
    - ak by sme túto tabuľku vypísali napr. pre `mincovka4([1,3,7,10,20],19)`, dostali by sme

```
[0, 1, 2, 1, 2, 3, 2, 1, 2, 3, 1, 2, 3, 2, 2, 3, 3, 2, 3, 4]
```

    - v tejto tabuľke posledná `4` označuje, že suma `19` sa dá rozmeniť 4 mincami, preto musí v tejto tabuľke existovať nižšia suma, ktorá sa dá rozmeniť 3 mincami a od 19 sa líši presne o niektorú z mincí (t.j. treba skontrolovať 19-1, 19-3, 19-7, ... a ktorá z týchto súm sa dá rozmeniť 3 mincami, tak tú vyberieme), nech je to napr. minca `7`
    - zaradíme mincu `7` do zoznamu pre výsledok a pokračujeme v hľadaní ďalšej mince: v tabuľke `tab[19-7]` má hodnotu `3`, preto hľadáme takú mincu, že `tab[12-minca]==2`, ak je tabuľka skonštruovaná korektne, taká minca bude aspoň jedna
    - takto pokračujeme, kým neposkladáme kompletný zoznam mincí

KAPITOLA 11

Grafy a union-find

11.1 Reprezentácie grafov

Už z programovania v 1. ročníku poznáme väčšie množstvo rôznych reprezentácií. Tu sa pozrieme na konkrétne štyri:

1. **zoznam hrán** (edge list)
   - všetky hrany sú sútredené do jedného zoznamu (pole alebo spájaný zoznam) pričom nie sú usporiadané v žiadnom poradí
   - v tomto zozname sú buď priamo dvojice (usporiadané alebo neusporiadané) vrcholov alebo hrán, ktoré sú objektami s atribútmi začiatok a koniec hrany
   - ak je graf ohodnotený, tento zoznam pri každej hrane obsahuje aj jej váhu
   - reprezentácia si ešte pamätá aj zoznam všetkých vrcholov
2. **zoznam susedov** (adjacency list)
   - pre každý vrchol sa udržuje zoznam (pole alebo spájaný zoznam) všetkých susediacich vrcholov
   - v neorientovanom grafe sa každá hrana musí nachádzať v oboch zoznamoch pre oba vrcholy
   - aby boli všetky operácie čo najefektívnejšie, zvykne sa zoznam realizovať dvojsmerným spájaným zoznamom: vyhodenie konkrétnej hrany (`remove_edge()`) má potom časovú zložitosť **O(1)**, inak bude mať táto operácia zložitosť **O(d)**
3. **asociatívne pole susedov** (adjacency map)
   - veľmi podobná realizácia **zoznamu susedov**: namiesto zoznamu použijeme asociatívne pole
   - vďaka tomu má operácia vyhľadania hrany (`get_edge()`) zložitosť len **O(1)**
   - zrejme kľúčom v týchto asociatívnych poliach susedov bude druhý vrchol, tento by ale mal byť **hashovateľný**
4. **matica susedností** (adjacency matrix)
   - každý riadok matice (i-ty) obsahuje informácie o susedoch i-teho vrcholu, teda j-ty stĺpec označuje hranu medzi i-tym a j-tym vrcholom, napr. hodnoty `False`/`True` alebo `0`/`1` označujú: neexistuje hrana/hrana existuje, resp. pre ohodnotené grafy priamo táto matica označuje váhu konkrétnej hrany (resp. nejaká hodnota označuje neexistenciu hrany, napr. `None`)
   - pre **riedke** grafy, v ktorých má každý vrchol malý počet hrán (susedov) je toto veľmi neefektívna reprezentácia, lebo väčšina matice (možno 90%) obsahuje informáciu o neexistencii príslušnej hrany
   - zrejme matica pre neorientovaný graf je symetrická

Ak by sme definovali abstraktnú triedu **graf** pravdepodobne by obsahovala tieto metódy:

	#1	#2	#3	#4	
`vertex_count()`	**O(1)**	**O(1)**	**O(1)**	**O(1)**	*počet vrcholov*
`edge_count()`	**O(1)**	**O(1)**	**O(1)**	**O(1)**	*počet hrán*
`vertices()`	**O(n)**	**O(n)**	**O(n)**	**O(n)**	*všetky vrcholy*
`edges()`	**O(m)**	**O(m)**	**O(m)**	**O(n**2)**	*všetky hrany*
`get_edge(u,v)`	**O(m)**	**O(d)**	**O(1)**	**O(1)**	*či je konkrétna hrana*
`degree(v)`	**O(m)**	**O(1)**	**O(1)**	**O(n)**	*stupeň vrcholu*
`incident_edges(v)`	**O(m)**	**O(d)**	**O(d)**	**O(n)**	*všetky susediace vrcholy*
`insert_vertex(x)`	**O(1)**	**O(1)**	**O(1)**	**O(n**2)**	*vlož vrchol*
`remove_vertex(v)`	**O(m)**	**O(d)**	**O(d)**	**O(n**2)**	*odstráň vrchol*
`insert_edge(u,v,x)`	**O(1)**	**O(1)**	**O(1)**	**O(1)**	*vlož hranu*
`remove_edge(e)`	**O(1)**	**O(1)**	**O(1)**	**O(1)**	*odstráň hranu*
pamäť	**O(n+m)**	**O(n+m)**	**O(n+m)**	**O(n**2)**	

V tejto tabuľke je okrem zložitostí operácií pre tieto 4 reprezentácie aj ich pamäťová zložitosť. Jednotlivé premenné tu majú tento význam:

- **n** počet vrcholov
- **m** počet hrán
- **d** stupeň vrcholu (pre riedke grafy sa blíži k **O(1)**, pre husté k **O(n)**)

11.2 Union-Find problém

motivácia:

- tajná služba sa snaží odhaliť identitu agentov
- každý agent má niekoľko svojich mien (zrejme pre každé má aj svoj pas)
- tajná služba by chcela mať systém, pomocou ktorého by vedela rýchlo zistiť, či dve rôzne mená patria jednému agentovi (napr. 'James Bond' a '007')
- zrejme takýto systém sa dá reprezentovať ako graf, v ktorom vrcholmi sú rôzne mená agentov a hranami vyjadrujeme, že dve mená patria jednému agentovi

V takomto systéme práve komponenty grafu vyjadrujú vzťah medzi agentom a jeho rôznymi menami. Takže tajná služba chce rýchlo vedieť zistiť, či sa dve mená nachádzajú v tom istom komponente.

V 1. ročníku sme konštruovali jednoduchý algoritmus, ktorý pomocou prehľadávania **do_hĺbky** alebo **do_šírky** vedel zostrojiť množiny vrcholov pre jednotlivé komponenty. Najväčší problém ale nastáva, keď sa tajná služba dozvie nový vzťah medzi dvoma menami (novú hranu grafu) a bude treba znovu prepočítať všetky komponenty, pritom by možno stačilo s existujúcimi komponentmi niečo urobiť. (Hoci, ak to potrebujeme zistiť len raz na začiatku bez ďalších pridávaní hrán, algoritmus do_hĺbky je dostatočne efektívny.)

Tento problém sa nazýva **union-find**, resp. sústava disjunktných množín (disjoint set (https://en.wikipedia.org/wiki/Disjoint-set_data_structure)):

- máme niekoľko disjunktných množín
- chceme vedieť rýchlo nájsť, v ktorej množine sa nachádza nejaký prvok (v ktorom komponente sa nachádza nejaký vrchol): operácia **find**
- a tiež chceme raz začas dve množiny spojiť do jednej (pridali sme novú hranu, musíme spojiť dva komponenty do jedného): operácia **union**

Ako reprezentovať takúto dátovú štruktúru (zrejme ju bude využívať napr. náš graf mien agentov), aby tieto operácie boli čo najefektívnejšie. Tieto disjunktné množiny by ste nemali pliesť so štandardným pythonovským typom `set`, preto sa niekedy používa aj terminológia disjunktné skupiny (group).

Aby sme vedeli manipulovať s takýmito množinami, každá bude mať svojho reprezentanta (jeden konkrétny prvok zo skupiny) - niekedy sa mu hovorí aj líder (leader).

Základné operácie tejto dátovej štruktúry:

- **make_set(x)** - vytvorí novú jednoprvkovú množinu
  - zrejme samotný prvok je aj reprezentantom tejto množiny
- **union(p, q)** - spojí dve množiny, ktoré obsahujú dané dva prvky, do jednej
  - väčšinou sa reprezentant jednej z týchto množín stáva reprezentantom aj ich zjednotenia
- **find(p)** - vráti reprezentanta množiny, v ktorej sa daný prvok nachádza

11.2.1 Triviálne riešenie

Predpokladajme, že každý prvok má informáciu o reprezentantovi množiny, do ktorej patrí. Napr. ak sme mali graf definovaný ako pole vrcholov:

```
class Vertex:
    def __init__(self, data):
        self.data = data
        self.rep = self                # reprezentant

class Graph:
    def __init__(self):
        self.vertex = []
    ...
```

Pri vytvorení vrcholu sme mu automaticky pridelili reprezentanta (samého seba). Operácia **FIND(p)** potom vráti túto hodnotu. Zložitosť tejto operácie je **O(1)**. Operácia **UNION(p, q)** najprv zistí reprezentantov oboch prvkov (pomocou **FIND**()) a ak sú rôzni, bude treba všetky výskyty druhého reprezentanta v poli všetkých prvkov (`self.vertex`) zmeniť na hodnoty prvého reprezentanta, teda schematicky:

```
def UNION(p, q):
    r1 = FIND(p)
    r2 = FIND(q)
    if r1 != r2:
        for v in vsetky_prvky:      # napr. self.vertex
            if v.rep == r2:
                v.rep = r1
```

Zrejme zložitosť tejto operácie je **O(n)** pre **n** počet prvkov (napr. vrcholov grafu).

Vo všeobecnosti toto riešenie znamená pomocnú tabuľku veľkosti **n**, v ktorej i-ty prvok tabuľky zodpovedá reprezentantovi i-teho prvku. Operácia **FIND()** iba siahne do tabuľky na príslušné miesto (teda **O(1)**) a operácia **UNION()** bude možno musieť prejsť celú tabuľku a niektorým prvkom zmeniť hodnotu (teda **O(n)**).

Jednorozmerné pole `r`, ktoré pre každý prvok obsahuje jeho reprezentanta skupiny (napr. číslo komponentu). Operácia **find(p)** má zložitosť **O(1)**, lebo len vyberie hodnotu z tabuľky. Operácia **union(p, q)** zistí reprezentantov pre oba prvky (r[p] a r[q]) a ak sú rôzne, prejde celú tabuľku a reprezentantov r[p] nahradí hodnotou r[q]. Zložitosť tejto operácie je **O(n)**.

Ak by sme zapísali túto reprezentáciu do tabuľky (do poľa), v ktorej i-ty prvok označuje reprezentanta, tak pre izolované vrcholy tabuľka vyzerá takto:

0	1	2	3	4	5	6	7	8	9
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Každý prvok je sám pre seba reprezentantom. Ak budeme teraz zjednocovať niektoré množiny, začnú sa prerábať reprezentanti pre jednotlivé prvky, napr. po operácii **UNION(1,3)** všetky prvky s reprezentantom **3** sa prerobia na **1**:

0	1	2	3	4	5	6	7	8	9
0	**1**	**2**	**1**	**4**	**5**	**6**	**7**	**8**	**9**

Momentálne máme už len 9 disjunktných množín, pričom jedna z nich má 2 prvky.

Po ďalších niekoľkých volaniach: **UNION(2,4)**, **UNION(7,6)**, **UNION(8,9)**:

0	1	2	3	4	5	6	7	8	9
0	**1**	**2**	**1**	**2**	**5**	**7**	**7**	**8**	**8**

Teraz máme už len 6 množín: dve sú 1 prvkové a štyri dvojprvkové.

Ďalšie dve volania **UNION(3,4)**, **UNION(7,8)** zlúčia ďalšie množiny:

0	1	2	3	4	5	6	7	8	9
0	**1**	**1**	**1**	**1**	**5**	**7**	**7**	**7**	**7**

Prvé volanie **UNION(3,4)** najprv zistilo reprezentantov prvkov **3** a **4**, to sú hodnoty **1** a **2** a preto prvky s hodnotami **2** sa nahradili **1**.

Ďalšie volanie **UNION(6,4)** zlúči množiny s reprezentantmi **7** (pre prvok **6**) a **1** (pre prvok **4**), teda nahradia **1** hodnotami **7**, vznikne tým jedna 8 prvková množina. Ak by sme ešte zavolali **UNION(1,9)**, neudeje sa nič, lebo oba prvky majú teraz rovnakého reprezentanta:

0	1	2	3	4	5	6	7	8	9
0	**7**	**7**	**7**	**7**	**5**	**7**	**7**	**7**	**7**

Vidíme, že tu teraz máme 3 disjunktné množiny a vieme veľmi rýchlo zistiť, či sa ľubovoľné dva prvky `x` a `y` nachádzajú v tej istej množine: stačí otestovať

```
tab[x] == tab[y]
```

Zrejme operácia **UNION()** tu má časovú zložitosť **O(n)**, keďže kvôli prečíslovaniu reprezentantov, musíme prejsť celú tabuľku. Ak by sme volali **UNION()** len pre prvky, ktoré sú v rôznych množinách, tak po **n-1** volaniach by sme dostali jedinú množinu, ktorá by obsahovala všetky prvky.

11.2.2 Riešenie so zoznamami

Aby sme v predchádzajúcom tabuľkovom riešení nemuseli pri **UNION()** zakaždým prechádzať celé pole, pre každý komponent si vytvoríme "spájaný zoznam" tých prvkov, ktoré do neho patria. Okrem toho si uložíme aj veľkosť tohto komponentu. Operácia **UNION()** bude teraz prechádzať a modifikovať len prvky menšieho komponentu. Potom ešte tieto dva spájané zoznamy zreťazí a nastaví si novú veľkosť komponentu.

V najhoršom prípade je operácia **UNION()** opäť **O(n)**, ale pre každý prvok platí, že jeho hodnota reprezentanta sa zmení iba vtedy, keď sa nachádza v menšom (alebo rovnako veľkom) komponente. Lenže toto sa môže stať maximálne **log n** krát, preto **n** operácií **UNION()** bude trvať **O(n log n)** a preto amortizovaná zložitosť jednej operácie **UNION()** je **O(log n)**.

V našom príklade s komponentmi grafu, by sme to mohli zapísať:

```
class Vertex:
    def __init__(self, data):
        self.data = data
        self.rep = self                 # reprezentant
        self.next = None                # nasledovník v komponente
```

```
        self.size = 1                   # počet prvkov v komponente

class Graph:
    def __init__(self):
        self.vertex = []
    ...
```

Asi bude najvhodnejšie bude, keď reprezentantom bude prvý prvok komponentu, teda spájaného zoznamu. Potom schematicky:

```
def UNION(p, q):
    r1 = FIND(p)
    r2 = FIND(q)
    if r1 != r2:
        if r1.size < r2.size:
            # všetkým prvkom v zozname r1 zmen reprezentanta na r2
            p = last = r1
            while p.next is not None:
                p.rep = r2
                last, p = p, p.next
            # zret'az oba zoznamy
            last.next = r2.next
            r2.next = r1
            # zvýš počet
            r2.size += r1.size
        else:
            # všetkým prvkom v zozname r2 zmen reprezentanta na r1
            ...
            # zret'az oba zoznamy
            ...
            # zvýš počet
            ...
```

11.2.3 Riešenie stromami

Predchádzajúce riešenie pomocou spájaných zoznamov ešte trochu vylepšíme. Aby sme nemuseli pri operácii **UNION()** prechádzať veľkú časť prvkov (všetky v nejakom komponente), nebudeme z nich vytvárať spájané zoznamy, ale orientované stromy: každý prvok si bude namiesto nasledovníka (`next`) pamätať svojho rodiča v strome (teda `parent`), pričom koreň bude obsahovať referenciu na samého seba. Každý prvok sa teda nachádza v nejakom strome (komponent, teda disjunktná množina), pričom do koreňa (to bude reprezentantom množiny) sa vieme dostať veľmi jednoducho sledovaním smerníka `parent`.

Aby sa nám čo najlepšie robila operácia **UNION()** (budeme zlučovať dva stromy do jedného), pre každý komponent si budeme uchovávať (atribút `rank`) aj jeho výšku (najdlhšia cesta smerom k listom). Pri inicializácii nastavíme `parent` na seba samého (`self`) a `rank` na 0:

```
class Vertex:
    def __init__(self, data):
        self.data = data
        self.parent = self              # predchodca v strome
        self.rank = 0                   # výška stromu

class Graph:
    def __init__(self):
        self.vertex = []
    ...
```

Operácia **FIND()** schematicky:

```
def FIND(p):
    while p != p.parent:
        p = p.parent
    return p
```

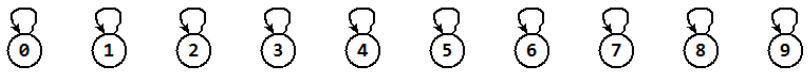
Takto sa dostane do kereňa stromu, čo je reprezentatntom celej množiny (komponentu). Operácia **UNION()** zistí, ktorý z komponentov má menší **rank** (výšku stromu) a ten potom pripojí ako ďalšieho syna ku koreňu väčšieho stromu. Keďže výška nového stromu sa pritom nezmení, netreba meniť ani hodnotu `rank`. Ak mali oba stromy rovnaký `rank`, tak pripojením jedného stromu ako syna k druhému sa o 1 zväčší jeho `rank`. Schematicky zapíšeme:

```
def UNION(p, q):
    r1 = FIND(p)
    r2 = FIND(q)
    if r1 != r2:
        if r1.rank > r2.rank:
            r2.parent = r1
        else:
            r1.parent = r2
            if r1.rank == r2.rank:
                r2.rank += 1
```

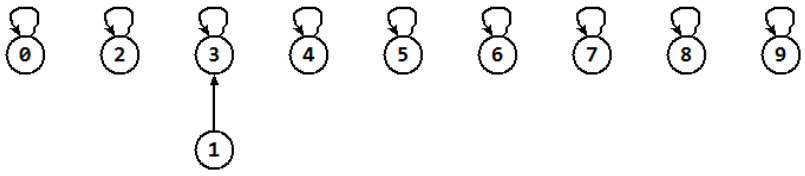
Časová zložitosť operácie **FIND(p)** závisí od výšky stromu, ktorý sa takto prechádza, čo je `rank` reprezentanta, teda koreňa stromu. Operácia **UNION()** urobí najprv 2 volania **FIND()** a potom len príkazy **O(1)**. Teda obe operácia majú rovnakú zložitosť. Keďže strom výšky **k** mohol vzniknúť len spojením (**UNION()**) dvoch stromov výšky **k-1**, každý strom s koreňom `rank` rovný **k** má aspoň **2**k** prvkov a preto takýto strom s **n** vrcholmi má výšku (`rank`) maximálne **log n**. Teda zložitosť oboch operácií je **O(log n)**.

Existujú ešte ďalšie verzie a vylepšenia tejto dátovej štruktúry, my sa nimi v tomto predmete zaoberať nebudeme.

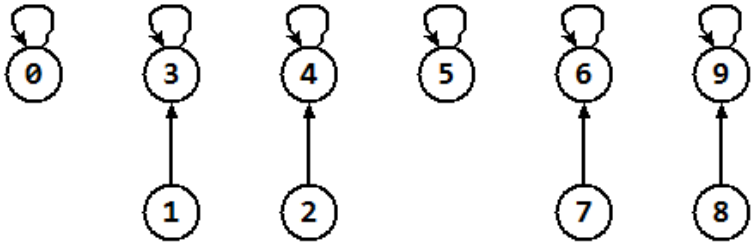
Príklad, ktorý sme robili pri tabuľkovej reprezentácii disjunktných množín, môžeme zopakovať aj pre stromčeky. Začnime s 10 jednoprvkovými množinami:



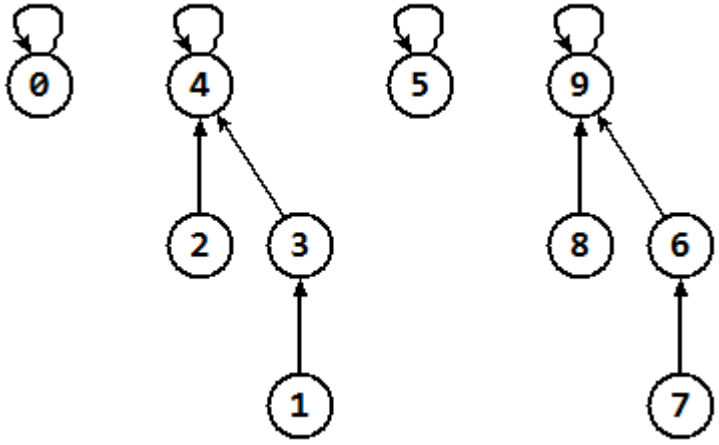
Najprv spojíme **UNION(1,3)**:



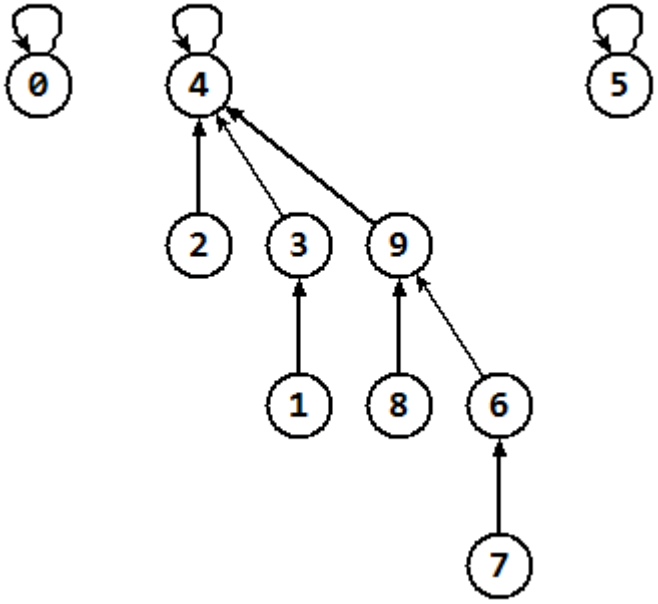
Teraz postupne **UNION(2,4)**, **UNION(7,6)**, **UNION(8,9)**:



Potom **UNION(3,4)**, **UNION(7,8)**:



Na záver **UNION(6,4)**, **UNION(1,9)**:



11.3 Iné využitie union-find

Dátová štruktúra **disjoint set** sa používa nielen na udržiavanie množín vrcholov jednotlivých komponentov grafu, ale má využitie, napr. pre

- zisťovanie, či je v grafe cyklus: postupne konštruujeme najprv z jednoprvkových množín (pre každý vrchol jedna) skladáme väčšie (prechádzame všetky hrany a pre každú spojíme dve zodpovedajúce množiny), ak zistíme, že oba vrcholy danej hrany sa už nachádzajú v jednom komponente, znamená to, že v grafe je cyklus
- vytváranie minimálnej kostry grafu

11.4 Cvičenie

11.4.1 reprezentácie grafov

1. Naprogramujte metódy triedy `Graph` (neorientovaný neohodnotený graf) pre reprezentáciu pomocou **zoznamu hrán**
   - využite triedy:

```
class Vertex:
    def __init__(self, data):
        self.data = data

class Edge:
    def __init__(self, start, end):      # start aj end sú typu Vertex
        self.start = start
        self.end = end

    def __eq__(self, other):
        return isinstance(other, Edge) and
               (self.start == other.start and self.end == other.end␣
↪or
                self.start == other.end and self.end == other.start)
```

   - naprogramujte tieto metódy (abstraktný dátový typ) podľa možnosti čo najefektívnejšie:

```
class Graph:
    def __init__(self):
        self.vertex = []
        self.edge = []
    def vertices(self):             # generátor, vracia objekty typu␣
↪Vertex
        ...
    def vertex_count(self):
        ...
    def edges(self):                # generátor, vracia objekty typu␣
↪Edge
        ...
    def edge_count(self):
        ...
    def get_edge(self, v1, v2):     # vráti objekt typu Edge alebo␣
↪None
        ...
    def incident_edges(self, v):    # generátor, vracia objekty typu␣
↪Edge
```

```
        ...
    def insert_vertex(self, x):    # x je dátová časť vrcholu,␣
↪funkcia vráti objekt typu Vertex
        ...
    def remove_vertex(self, v):    # v je objekt typu Vertex
        ...
    def insert_edge(self, v1, v2): # vráti objekt typu Edge
        ...
    def remove_edge(self, e):      # e je objekt typu Edge
        ...
```

2. Pomocou algoritmu do hĺbky zistite, či sa dva vrcholy nachádzajú v tom istom komponente:
   - rekurzívnu funkciu **zapíšte** tak, aby používala len metódy z úlohy (1), teda aby vaša funkcia fungovala pre ľubovoľnú reprezentáciu grafu

```
class Graph:
    ...
    def depth_first(self, v1, v2):
        ...
```

   - **odhadnite** zložitosť tohto algoritmu
   - **otestujte** na náhodne vygenerovanom grafe, ktorý obsahuje niekoľko rôzne veľkých komponentov
3. Pomocou algoritmu do hĺbky zistite, či je hrana medzi dvoma susednými vrcholmi **most**, t.j. po jej odstránení z grafu by sa zvýšil počet komponentov
   - rekurzívnu funkciu zapíšte tak, aby používala len metódy z úlohy (1), teda aby vaša funkcia fungovala pre ľubovoľnú reprezentáciu grafu

```
class Graph:
    ...
    def is_bridge(self, v1, v2):
        ...
```

   - odhadnite zložitosť tohto algoritmu
4. Pomocou algoritmu do hĺbky zistite počet komponentov grafu.
   -

```
class Graph:
    ...
    def component_count(self):
        ...
```

   - odhadnite zložitosť tohto algoritmu

11.4.2 union-find

5*. Postupne ručne odtrasujte všetky 3 reprezentácie operácií **UNION()** a **FIND()** (podobne, ako je to ukázané v prednáške) na 12 prvkovom poli celočíselných hodnôt `range(12)` a volaním **UNION()** pre dvojice

- (0, 11), (1, 10), (2, 9), (3, 8), (4, 7)
- (0, 7), (8, 1), (5, 9), (2, 6)
- (11, 3), (2, 4)

V ďalších úlohách operácie **UNION()** a **FIND()** realizujte v rôznych reprezentáciách a najprv otestujte na údajoch z úlohy (5). Pre každú z reprezentácií si vhodne zvoľte dátovú štruktúru, napr. ako pole objektov typu `Vertex`.

6*. reprezentujte tabuľkou

- ```
  def FIND(p):
      ...
  def UNION(p, q):
      ...
  ```

7*. reprezentujte zoznamami

- ```
  def FIND(p):
      ...
  def UNION(p, q):
      ...
  ```

8*. reprezentujte stromčekmi s informáciou `rank` (výška stromu)

- ```
  def FIND(p):
      ...
  def UNION(p, q):
      ...
  ```

9. reprezentujte stromčekmi ale bez ďalšej informácie (t.j. bez `rank`: rozhoduje sa náhodne s pravdepodobnosťou 1/2)

- ```
  def FIND(p):
      ...
  def UNION(p, q):
      ...
  ```

10. pre jednotlivé reprezentácie zapíšte funkciu, ktorá zistí počet disjunktných množín (komponentov grafu) len na základe informácií v tejto reprezentácií

- pravdepodobne budete prechádzať pole prvkov a nejako si pri tom budete evidovať rôzne množiny