Algoritmy a dátové štruktúry, 2. vydanie

Andrej Blaho

04. sep 2018

Obsah

Fakulta matematiky, fyziky a informatiky

Univerzita Komenského v Bratislave

Autor Andrej Blaho

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

Vydavateľ Knižničné a edičné centrum FMFI UK

Rok vydania 2018

Miesto vydania Bratislava

Vydanie druhé doplnené vydanie

Počet strán 218

Internetová adresa http://input.sk/struct2017

Aktuálna adresa kurzu http://struct.input.sk/

ISBN 978-80-8147-085-1

ISBN webovej verzie 978-80-8147-086-8

KAPITOLA 1

1. Úvod

1.1 Priebeh semestra

Vyučovanie počas semestra sa skladá z

- týždenne jedna dvojhodinová prednáška
- týždenne jedno dvojhodinové cvičenie
- skoro každý týždeň jedno domáce zadanie
- jeden semestrálny projekt
- jeden písomný test
- praktická skúška pri počítači

cvičenia

prebiehajú v počítačových halách H3 a H6

- bude sa precvičovať látka hlavne z prednášky v danom týždni
- môžete pracovať na vlastnom notebooku
- spolupráca na cvičeniach nie je zakázaná - je odporúčaná
  - môžete pracovať aj po dvojiciach (dvaja riešia úlohy pri jednom počítači)
- na cvičeniach je povinná **aktívna účasť**
  - budú sa kontrolovať vyriešené úlohy
  - riešenie budete ukladať na úlohový server LIST
- povolené sú maximálne 2 absencie

domáce zadania

počas semestra dostanete niekoľko povinných samostatných domácich zadaní

- na ich vyriešenie dostanete väčšinou 2 až 3 týždne
- vaše riešenia budú bodované (väčšinou 10 bodov)
  - maximálny počet bodov získate len za úplne správne riešenie
  - riešenie budete ukladať na úlohový server LIST

priebežné hodnotenie

aktívna účasť na cvičeniach a body získané za domáce zadania sú priebežným hodnotením semestra

- zo všetkých cvičení počas semestra môžete mať maximálne **2 neúčasti**
- zo všetkých domácich zadaní musíte získať spolu aspoň **50% bodov**
- ak nesplníte podmienky priebežného hodnotenia, získavate známku **Fx** (bez možnosti robiť skúšku)

semestrálny projekt

v priebehu semestra dostávate možnosť riešiť jeden semestrálny projekt:

- tému si zvolíte sami, mala by obsahovať niečo, čo je obshom nášho predmetu ADŠ
- za načas odovzdaný projekt môžete získať maximálne 10 bodov - body nad 5 sa pripočítavajú k bodom ku skúške (ak máte aspoň 5 bodov, započítavajú sa ku skúške)
- pri splnení podmienky ročníkového projektu, sa tento projekt môže hodnotiť aj ako ročníkový projekt

skúška

sa skladá z dvoch častí:

1. písomný test (posledný týždeň semestra **18.12.**) - max. 40 bodov
2. praktická skúška pri počítači (v skúškovom období) - max. 60 bodov

- máte nárok na 2 opravné termíny

---

hodnotenie skúšky

spočítajú sa body z písomného testu, praktickej skúšky, príp. bodov za semestrálny projekt:

- **:-)** známka **A** 88 bodov
- **:-)** známka **B** 81 bodov
- **:-)** známka **C** 74 bodov
- **:-)** známka **D** 67 bodov
- **:-)** známka **E** 60 bodov
- **:-(** známka **Fx** menej ako 60 bodov

užitočné linky

- Problem Solving with Algorithms and Data Structures using Python
- Python Tutor

1.2 Analýza algoritmov

1.2.1 Časová zložitosť

Budeme skúmať rôzne algoritmy z pohľadu ich rýchlosti behu pre rôzne veľké vstupné údaje. Už z programovania v prvom ročníku máme nejaké skúsenosti, že niektoré programy sú pre veľké vstupy veľmi pomalé, niekedy sa dokonca ani nedočkáme výsledkov. Zatiaľ boli naše úvahy skôr intuitívne. Teraz sa zoznámime so základmi posudzovania časovej zložitosti rôznych skupín algoritmov.

Ukážeme to na riešení takejto úlohy: v danom číselnom poli potrebujeme zistiť počet všetkých dvojíc rovnakých hodnôt. Pozrime si takéto jednoduché riešenie:

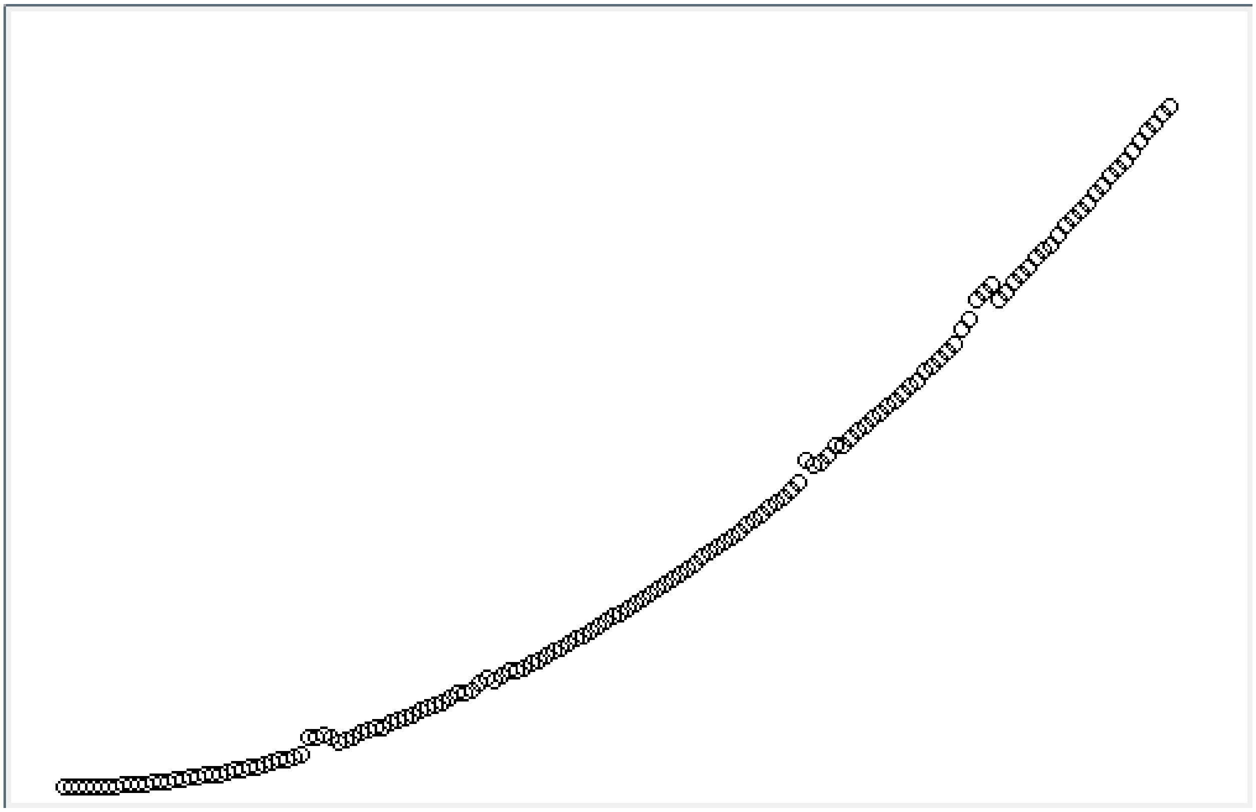
```
pole = [2, 6, 7, 5, 4, 7, 8, 3, 1, 5, 9]
n = len(pole)
pocet = 0
for i in range(n):
    for j in range(i + 1, n):
        if pole[i] == pole[j]:
            pocet += 1
print(pocet)
```

Uvedomte si, aký bude výsledok pre pole rovnakých hodnôt, napr. pre pole = [1, 1, 1, 1, 1].

Poďme teraz odmerať, ako dlho bežal tento program. Urobíme to tak, ako sme to robili v prvom ročníku. Jednou z možností je merať čas pre rôzne veľké náhodné polia:

```
for n in range(1000, 10001, 1000):
    pole = [random.randrange(n) for i in range(n)]
    tim = time.time()
    pocet = 0
    for i in range(n):
        for j in range(i + 1, n):
            if pole[i] == pole[j]:
                pocet += 1
    tim = time.time() - tim
    print(n, round(tim, 3))
```

Z nameraných hodnôt môžeme nakresliť napr. takýto graf:



Zrejme si uvedomujete, že takéto merania nie sú presné, ale za to aspoň vidíme, ako tento odmeraný čas rastie.

Skúsme to urobiť inak. Program nebudeme spúšťať, len budeme počítať. Budeme počítať počet **elementárnych inštrukcií**, ktoré sa vykonajú pre pole veľkosti `n`. Predpokladajme, že tieto operácie majú približne rovnaký jednotkový čas:

- priradenie do premennej `prem = hodnota`
- zistenie hodnoty premennej `prem`
- vykonanie aritmetickej operácie `hodnota1 op hodnota2`
- porovnanie dvoch hodnôt `hodnota1 rel hodnota2`
- zaindexovanie prvku poľa `prem[hodnota]`
- volanie funkcie
- návrat z funkcie

Tiež predpokladajme, že for-cyklus:

```
for i in range(n):
    telo_cyklu
```

sa pre zjednodušenie bude prepisovať takto:

```
i = 0                          # 1 inštrukcia
while i < n:                   # 2 inštrukcie
    #telo_cyklu
    i += 1                     # 1 inštrukcia
```

Poďme teraz počítať telo vnútorného cyklu:

```
if pole[i] == pole[j]:        # 5: i, pole[i], j, pole[j], ==
    pocet += 1                # 3: pocet, pocet+1, pocet = hodnota
```

Telo vnútorného cyklu bude trvať 5 alebo 8 časových jednotiek, podľa výsledku porovnania. Keďže sa toto vykonáva vo for-cykle, pripočítajme k tomuto času 2 jednotky na začiatok a jednu na koniec. Takže telo cyklu je 8 alebo 11 jednotiek za každý prechod.

Prepíšme oba for-cykly:

```
pocet = 0                             # 1
i = 0                                 # 1
while i < n:                          # 2
    j = i + 1                         # 3
    while j < n:                      # 2
        if pole[i] == pole[j]:        # 5
            pocet += 1                # 3
        j += 1                        # 1
    i += 1                            # 1
```

Vnútorný for-cyklus sa bude vykonávať najprv. pre `i=0`, potom pre `i=1`, potom pre `i=2`, atď. až naposledy pre `i=n-1`, teda prvýkrát pôjde `n-2` krát, potom `n-3` krát, . . . až 1-krát. Spolu je to súčet: 1 + 2 + 3 + . . . + `n-2`, teda `(n-1)*(n-2)/2`. Budeme predpokladať najhorší prípad (časovo najnáročnejší), teda že telo vnútorného cyklu beží 11 časových jednotiek. Vonkajší for-cyklus (teda while) beží `n` krát, teda `n` krát sa vykoná test `i<n` aj `i+=1`.

Celkový čas potom:

```
f(n) = (n-1)*(n-2)/2 * 11 + n * 6 + 2 = 5.5 * n ** 2 - 16.5 * n + 6 * n + 3
```

Takúto funkciu, ktorá vyjadruje časový odhad trvania nejakého konkrétneho algoritmu, budeme nazývať **časová zložitosť**. Pravdepodobne, keby sme nakreslili priebeh tejto funkcie, dostali by sme veľmi podobné výsledky ako graf v našich prvých meraniach.

V skutočnosti nás nebude zaujímať až taký presný vzorec (s nepresnými predpokladmi o trvaní jednotlivých inštrukcií), ale dôležitý bude charakter takejto funkcie, teda či rastie rýchlo, či rastie rovnako ako funkcia iného algoritmu, alebo či rastie pomalšie ako iná funkcia.

V prvom rade budeme zanedbávať všetky „nepodstatné“ konštanty. Na priebeh funkcie to nemá skoro žiaden vplyv. Tiež, ak sa nejaká funkcia skladá z viacerých členov, pričom jeden rastie rýchlejšie a iný pomalšie, tak ten pomalší tiež „zanedbáme“. Teda pre náš algoritmus je dôležité, že jeho časová zložitosť je kvadratická funkcia (rastie tak rýchlo, ako **n**2**).

1.2.2 Triedy funkcií veľké O()

Funkcie časovej zložitosti budeme zaraďovať do tried. Každá z tried bude charakteristická nejakou funkciou, ktorá ju **ohraničuje zhora**.

Formálnejšie musí platiť:

- funkcia `f(n)` patrí do tej istej triedy ako `g(n)`, ak existuje číselná konštanta `c`, že pre dosť veľké `n` platí `c * g(n) >= f(n)`
- takúto triedu funkcií potom označujeme **O(g(n))** (hovoríme trieda veľké O)
- budeme hovoriť, že `f(n)` patrí do **O(g(n))**, alebo aj `f(n)` je **O(g(n))**

Funkcii `g(n)` sa hovorí **horný asymptotický odhad** funkcie `f(n)`.

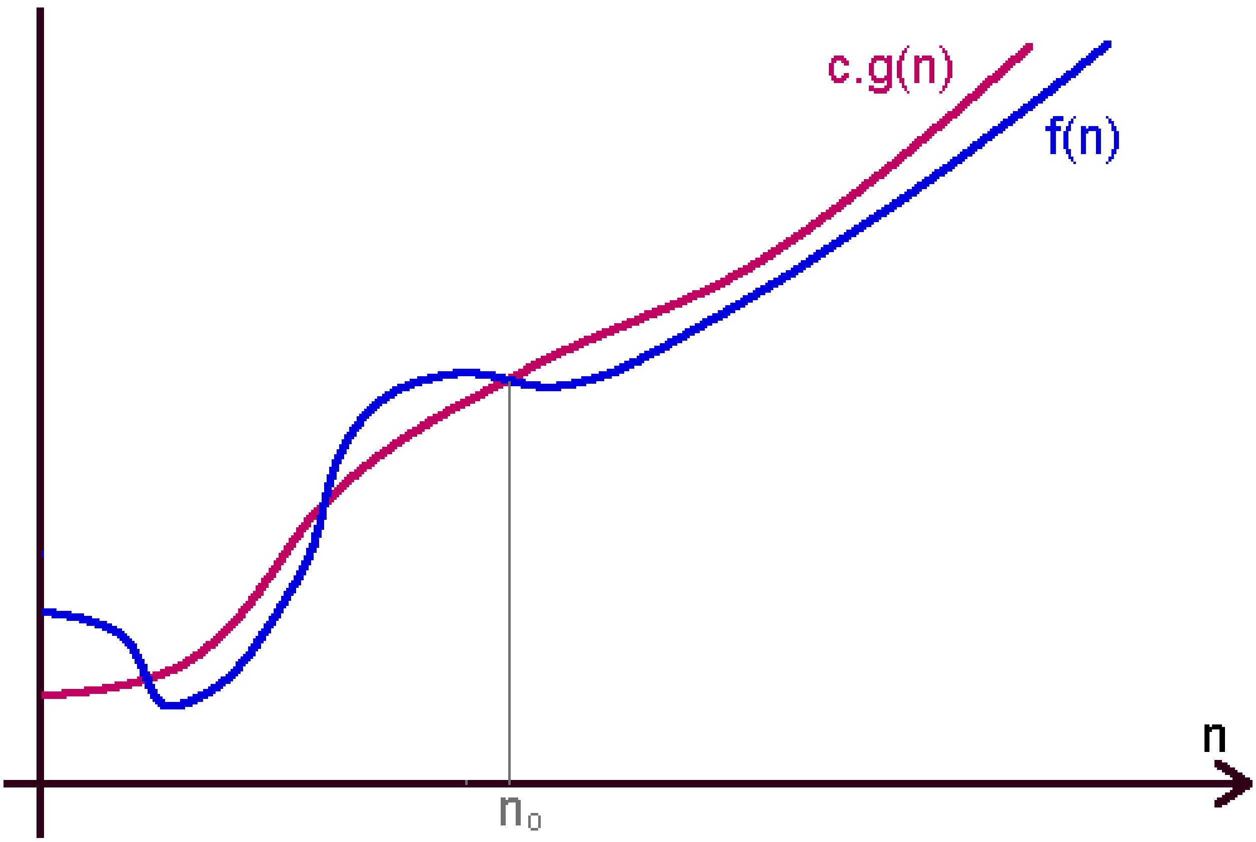
Napr. časová zložitosť nášho skúmaného programu je **O(n**2)**.

Formálne

Pre dané dve funkcie `f(n)` (odhad počtu elementárnych inštrukcií algoritmu) a `g(n)` hovoríme, že `f(n)` je triedy **O(g(n))**, ak existujú také dve kladné konštanty `c` a `n0`, že pre každé `n > n0` platí:

c * g(n) >= f(n)

Môžete si predstaviť napr. takto:



1.2.3 Základné skupiny funkcií časovej zložitosti

V tomto kurze sa budeme stretávať len s týmito skupinami funkcií:

O(1) konštantná zložitosť

Algoritmus, ktorý nezávisí od veľkosti vstupu. Napr. ak chceme zistiť súčet radu čísel od 1 do `n` a poznáme na to vzorec:

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

O(log n) logaritmická zložitosť

Časová zložitosť je logaritmická funkcia, napr. hľadanie prvku v utriedenom poli algoritmom binárne vyhľadávanie:

```
def hladaj(hodnota, pole):
    od, do = 0, len(pole)
    while od <= do:
        stred = (od + do) // 2
        if pole[stred] == hodnota:
            return True
        if pole[stred] < hodnota:
            od = stred + 1
        else:
            do = stred - 1
    return False
```

Uvedomte si, že pri logaritmickej funkcii nemusíme špecifikovať základ logaritmov, nakoľko platí

```
logA n = logB n / logB A
```

pre ľubovoľné základy `A` a `B`.

O(n) lineárna zložitosť

Algoritmus, ktorého zložitosť rastie lineárne s veľkosťou vstupu. Napr. ak chceme zistiť súčet radu čísel od 1 do `n` a počítame to pomocou cyklu:

```
def sucet(n):
    res = 0
    for i in range(1, n+1):
        res += i
    return res
```

Ale tiež hľadanie hodnoty v neutriedenom poli:

```
def hladaj(hodnota, pole):
    for prvok in pole:
        if prvok == hodnota:
            return True
    return False
```

O(n log n) zložitosť n * log n

Neskôr budeme vidieť, že najrýchlejšie algoritmy triedenia poľa s prvkami ľubovoľných typov má práve túto zložitosť. Dokonca ukážeme, že ani nemôže neexistovať algoritmus s lepšou zložitosťou.

O(n**2) kvadratická zložitosť

Časová zložitosť je kvadratická funkcia, napr. bublinkové triedenie:

```
def tried(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]
```

O(n3) kubická zložitosť**

Algoritmy s touto zložitosťou sú zriedkavejšie ako kvadratické, ale uvidíme ich napr. pri riešení grafových úloh.

O(2n) exponenciálna zložitosť**

Veľmi často sú to algoritmy prehľadávania s návratom (backtracking), v ktorých prechádzame všetky vygenerované možnosti a vyberáme niektoré podľa nejakých kritérií. Najčastejšie sú to rekurzívne algoritmy.

Podobne ako pri logaritmických zložitostiach aj do tejto triedy patria funkcie s ľubovoľným základom.

O(n!) faktoriálová zložitosť

Napr. rekurzívne vygenerovanie všetkých permutácií:

```
def generuj(i):
    for j in range(n):
        if not bolo[j]:
            pole[i] = j
            bolo[j] = True
            if i == n - 1:
                print(pole)
            else:
                generuj(i + 1)
            bolo[j] = False

n = 3
pole = [0] * n
bolo = [False] * n
generuj(0)
```

1.2.4 Porovnanie

Zrejme medzi týmito triedami zložitosti platí takýto vzťah:

O(1) < O(log n) < O(n) < O(n log n) < O(n2) < O(n**3) < O(2**n) < O(n!)**

1.2.5 Ako si zjednodušiť zisťovanie časovej zložitosti

1. postupnosť príkazov, v ktorej je každý **O(1)**, má zložitosť **O(1)**
2. for-cyklus s n-prechodmi, ktorý vykonáva len **O(1)**, má zložitosť **O(n)**
3. všeobecnejšie for-cyklus s n-prechodmi, ktorý vykonáva telo cyklu **O(f)**, má zložitosť **O(n * f)**
4. while cyklus, ktorý znižuje (zvyšuje) nejakú testovanú premennú o konštantu, má zložitosť **O(n * f)**, kde **O(f)** je telo cyklu, **n** je testovaná hodnota
5. while cyklus, ktorý delí nejakú testovanú premennú o konštantu (napr. na polovicu), má zložitosť **O(log n * f)**, kde **O(f)** je telo cyklu, **n** je testovaná hodnota
6. ak máme spolu dva algoritmy, pri ktorých sa druhý vykoná po skončení prvého, potom majú celkovú zložitosť **O(f1 + f2)**

7. rekurzívne funkcie majú často zložitosť rádovo porovnateľnú s počtom vygenerovaných výsledkov

Ukážme to na tomto príklade: pripomeňme si **insert-sort** z prvého ročníka:

```
def insert_sort(pole):
    for i in range(1, len(pole)):                 # O(n**2)
        prvok = pole[i]                    # O(1)
        j = i-1                            # O(1)
        while j >= 0 and pole[j] > prvok:         # O(n)
            pole[j+1] = pole[j]            # O(1)
            j -= 1                         # O(1)
        pole[j+1] = prvok                  # O(1)
```

Všetky príkazy okrem cyklov sú zložitosti **O(1)**. Keďže premenná while-cyklu `j` sa znižuje o 1 a jej rozsah je priamo úmerný **n**, zložitosť tohto while cyklu je **O(n)**, čo je vlastne telo for-cyklu. For-cyklus je potom **n** krát telo **O(n)**, t.j. tento algoritmus je **O(n**2)**.

1.2.6 Tabuľka, v ktorej sa porovnáva rýchlosť

Teraz, keď už pre naše algoritmy vieme zistiť ich časovú zložitosť, lepšie si budeme vedieť predstaviť časové obmedzenia, ktoré prichádzajú s rôzne zložitými algoritmami. Predpokladajme, že máme k dispozícii počítač, ktorý zvláda napr. 5000000 elementárnych operácií za sekundu (súčasné počítače sú výrazne rýchlejšie). Zaujíma nás, aký veľký problém by sme vedeli vyriešiť za nejaký konkrétny čas. Napr. z tabuľky nižšie je vidieť, že za jednu sekundu algoritmus zložitosti **O(n log n)** zvládne rozsah 280000, ale pre zložitosť **O(n**3)** už len rozsah 170. Podobne z tejto tabuľky môžeme vidieť, že ak nejaké pomalé triedenie (napr. bubble-sort) 17000-prvkové pole triedilo minútu, 660000-prvkové pole sa bude triediť celý deň.

```
                   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                ∞  620000000     130000       2600         34
den                ∞          ∞     660000       7600         39
mesiac             ∞          ∞    2300000      17000         42
rok                ∞          ∞   13000000      54000         47
1000               ∞          ∞  400000000     540000         57
```

Môže nám to potvrdiť aj takýto spôsob výpočtu:

Predpokladajme, že nejaký náš program bežal pre vstup veľkosti napr. n=17000 6 sekúnd. Ak vieme, že jeho zložitosť je **O(n**2)**, mohli by sme dosť presne odhadnúť, ako dlho bude bežať dvojnásobne väčší vstup:

- teda `f(17000) = 6` sekúnd
- potrebujeme zistiť `f(2*17000)`, pričom vieme, že `f()` je kvadratická funkcia
- preto `f(2*17000) = 4 * f(17000) = 24` sekúnd

Porozmýšľajte, ako by sa zmenil výsledok, keby bol náš program inej zložitosti, napr. **O(1)**, **O(n)**, **O(n log n)**.

1.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Naprogramujte a zistite, akú časovú zložitosť má algoritmus na zistenie ciferného súčtu (v desiatkovej sústave). Nepoužívajte prácu s reťazcami.

   - funkcia ciferný súčet:

   ```
   def cs(n):
       ...
   ```

2. Zistite, aká je zložitosť týchto algoritmov:

   - hľadanie minimálneho prvku vo vzostupne utriedenom poli

   ```
   def minimal(pole):
       return pole[0]
   ```

   - hľadanie minimálneho prvku v poli (dosť neefektívne)

   ```
   def minimal(pole):
       naj = pole[0]
       i = 1
       while i < len(pole):
           if naj <= pole[i]:
               i += 1
           else:
               naj = pole[i]
               i = 1
       return naj
   ```

   - zisťuje akýsi súčet

   ```
   i, j = 1, 1
   sum = 0
   while i < n:
       sum += i * j
       i = 10 * i
       j = (j + 1) % 10
   print(sum)
   ```

   - rekurzívny výpočet

   ```
   def f(n):
       if n <= 1:
           return 1
       return f(n // 2) + f(n // 2)
   ```

   - rekurzívny výpočet - to isté ako predchádzajúci príklad, ale máličko inak

   ```
   def f(n):
       if n <= 1:
           return 1
       return 2 * f(n // 2)
   ```

   - ešte jeden cyklus

```
f = 1
while 2 * f <= n:
    f *= 2
print(f)
```

- rozklad čísla na prvočinitele

```
def rozklad(n):
    i = 2
    while n > 1:
        if n % i == 0:
            print(i)
            n = n // i
        else:
            i += 1
```

3. Potrebujeme nájsť počet výskytov maximálneho prvku v poli.

- naprogramovali sme to takto:

```
def max(pole):
    res = None
    for prvok in pole:
        if res is None or prvok > res:
            res = prvok
    return res

def pmax(pole):
    pocet = 0
    for prvok in pole:
        if prvok == max(pole):
            pocet += 1
    return pocet
```

- zistite zložitosť tohto algoritmu
- preprogramujte ho tak, aby sa znížila jeho zložitosť

4. Príklad na zistenie počtu všetkých dvojíc rovnakých čísel v poli z prednášky prerobte takto:

(a) utrieďte najprv celé pole nejakým rýchlym **O(n log n)** triedením (ale nie štandardný pythonovský sort)

(b) v utriedenom poli ľahko nájdeme všetky úseky rovnakých čísel - stačí nám na to lineárny cyklus, t.j. **O(n)**

- porovnajte výsledky oboch algoritmov na nejakom veľkom náhodnom poli
- aká je zložitosť tohto algoritmu?
- porovnajte čas trvania behu oboch algoritmov

5. Potrebujeme nájsť počet takých prvkov v poli, ktoré sa tam nachádzajú len raz.

- naprogramovali sme to takto:

```
def pocet_len_raz(pole):
    pocet = 0
    for prvok in pole:
        pocet1 = 0
        for prvok1 in pole:
            if prvok == prvok1:
                pocet1 += 1
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        if pocet1 == 1:
            pocet += 1
    return pocet
```

- zistite zložitosť tohto algoritmu
- preprogramujte ho tak, aby sa znížila jeho zložitosť

možnosti

1. O(log n)
2. O(n)
3. O(n log n)
4. O(n**2)
5. O(2**n)
6. O(n!)

6. Vyriešte 1. príklad z testu z pred dvoch rokov

- Pre všetky nasledujúce funkcie odhadnite časovú zložitosť: ku každej pripíšte jedno z písmen A až F.

```
def fun1(n):
    x = 0
    for i in range(n):
        x += 1
    return x

def fun2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x

def fun3(n):
    if n == 0: return 1
    x = 0
    for i in range(n):
        x += fun3(n-1)
    return x

def fun4(n):
    if n == 0: return 0
    return fun4(n//2) + fun1(n) + fun4(n//2)

def fun5(n):
    x, i = 0, n
    while i > 0:
        x += fun1(i)
        i //= 2
    return x

def fun6(n):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    if n == 0: return 1
    return fun6(n-1) + fun6(n-1)

def fun7(n):
    if n == 1: return 0
    return 1 + fun7(n//2)
```

KAPITOLA 2

2. Polia, iterátory

2.1 Polia v počítači

V Pythone sú 3 sekvenčné typy, ktoré sú vnútorne 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, pamäť s náhodným prístupom) - rovnaký čas na prístup k ľubovoľnému pamäťovému miestu na základe indexu = **O(1)** (na rozdiel od sekvenčného prístupu, keď O(1) je len pre nasledovníka, napr. spájané zoznamy, súbory, . . .)
- prvky poľa sú postupne uložené v pamäťových miestach tak, aby sa dala čo najjednoduchšie vypočítať adresa ľubovoľného prvku
  - adresa `A[i]` = adresa začiatku `A` + `i` * počet_bajtov_prvku
- typ `str` je v Pythone realizovaný ako kompaktné pole znakov - každý znak je v Unicode a pre celé pole je vyhradené buď 1 bajt na každý prvok, 2 bajty na prvok alebo 4 bajty (zrejme podľa prvku, ktorý zaberá najviac bajtov)
- typy `tuple` a `list` sú poľami referencií na objekty (smerníky)

Môžeme si zjednodušiť pohľad na pamäť: všetky hodnoty, ktoré sú uložené v premenných (t.j. sú prístupné pomocou referencií v premenných), sa nachádzajú v jednom pamäťovom priestore. Tento priestor je vlastne jedno veľké pole pamäťových jednotiek rovnakej veľkosti (napr. bajtov = 8 bitov). Do takéhoto poľa sa potom ukladajú všetky hodnoty: aj celé čísla, ktoré môžu byť aj niekoľko 1000-bitové, aj desatinné, ktoré sú 64-bitové, aj znakové reťazce, ktoré môžu zaberať len niekoľko bajtov, ale aj milióny bajtov. **Správca pamäti** sa potom stará o to, aby táto pamäť bolo čo najefektívnejšie využitá a pritom, aby sa s týmito hodnotami v pamäti pracovalo prijateľne rýchlo.

Problémom dynamických polí je to, že môžu veľmi často meniť svoju veľkosť a tým môžu veľmi zaťažovať správcu pamäte: ten musí nájsť dostatočne veľký súvislý úsek, do ktorého sa zmestí nová veľkosť takéhoto poľa. Zrejme bude potom treba aj zabezpečiť, aby sa pôvodný obsah poľa presunul na novú pozíciu.

2.1.1 Kompaktné pole v Pythone

Okrem typu `str`, ktorý je kompaktným poľom znakov (pole obsahuje priamo znaky a nie referencie na iné pamäťové miesta so znakmi), Python poskytuje možnosť vytvárať kompaktné pole aj iných typov. Tieto polia musia mať všetky prvky rovnakého (číselného) typu, aby sa mohol zabezpečiť **RAM** (náhodný) prístup k prvkom. Zrejme všetky prvky jedného poľa musia zaberať rovnaký počet bajtov.

Pythonovský modul `array` umožňuje definovať takéto kompaktné číselné pole (ktoré sa líši 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. Na tejto ukážke môžete vidieť jednoduché použitie kompaktných polí dvoch rôznych bajtových typov:

```
>>> import array
>>> a = array.array('b', [2,3,5,7,11,13,17,19])      # pole bajtov so znamienkom, t.j.
↪ hodnôt od -128 do 127
>>> 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)                     # pole bezznamienkových bajtov,␣
↪t.j. hodnôt od 0 do 255
>>> 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])
>>>
```

Pri vytváraní kompaktného poľa (pomocou `array.array()`) musíme určiť konkrétny typ prvkov, na základe čoho sa bude dať vyhradiť súvislá pamäť 1-bajtových, 2-bajtových, alebo 4-bajtových, ... čísel. Definované typy pre kompaktné pole sú tieto:

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**

2.1.2 Dynamické pole a amortizácia

Zameriame sa na efektívnosť metódy `append()`, ktorá sa v Pythonovských algoritmoch využíva veľmi frekventovane, napr.

```
a = []
for i in range(1000000):
    a.append(i)
```

V tejto ukážke vidíme, že dané pole sa v cykle 1000000-krát zväčšuje stále o 1 prvok a teda aj správa pamäti musí nejako zabezpečiť, aby sa takéto pole mohlo stále nafukovať až dosiahne 1 milión prvkov. Zrejme dopredu netuší, aké veľké pole budeme potrebovať. Asi vám napadne, že stačí v pamäti len zväčšovať existujúci vyhradený priestor pre toto jedno pole, veď v programe sa so žiadnymi ďalšími údajmi nerobí, takže celá pamäť je k dispozícii na toto jediné pole. Lenže Python to dobre zvláda aj vtedy, keď bude pracovať napr. s dvoma poľami:

```
a, b = [], []
for i in range(1000000):
    a.append(i)
    b.append(i)
```

Asi za tým bude nejaký dobrý nápad, aby sa nemuselo sťahovať celé pole na novú pozíciu v pamäti pri každom volaní metódy `append()`. Pravdepodobne si Python vyhradí niekoľko prvkov do rezervy tak, aby nie po každom volaní `append()` musel robiť drahé upratovanie. Poďme teraz skúmať, koľko pamäti naozaj zaberá nejaká konkrétna dátová štruktúra. Využijeme funkciu `getsizeof()` z modulu `sys`, ktorá vráti počet bajtov, ktoré sú vyhradené pre nejakú hodnotu. Najprv poexperimentujme (je možné, že na vašom počítači dostanete trochu iné výsledky - závisí od toho, či máte 32 alebo 64 bitový systém, či máte Windows, alebo Linux, . . .):

```
>>> sys.getsizeof(1)
14
>>> sys.getsizeof(2**1000)
146
>>> sys.getsizeof(1.)
16
>>> sys.getsizeof(1e300)
16
```

Podľa veľkosti celého čísla sa vyhradzuje rôzne veľká pamäť. Desatinné čísla zaberajú stále rovnaký počet bajtov. Uvedomte si, že okrem samotnej hodnoty si Python musí pamätať aj nejaké ďalšie informácie, napr. konkrétny typ, momentálny rozsah (napr. aké veľké číslo sa sem zmestí), . . .

Podobne aj pre zložené typy:

```
>>> sys.getsizeof([])
36
>>> sys.getsizeof([1])
40
>>> sys.getsizeof([1, 2])
44
>>> sys.getsizeof([1, 2, 3])
48
>>> sys.getsizeof([1] * 100)
436
>>> sys.getsizeof('')
25
>>> sys.getsizeof('a')
26
>>> sys.getsizeof('aa')
27
>>> sys.getsizeof('a' * 100)
125
```

Môžeme tu odsledovať, že na mojom počítači je pre pole (typ `list`) vyhradená pamäť tak, že 36 bajtov je pre všeobecné informácie a 4 bajty (zrejme mám 32 bitový systém) na každý prvok poľa. Podobne je to so znakovými reťazcami, kde sa k číslu 25 pripočítava toľko bajtov, aký je počet prvkov. Zrejme sú zatiaľ všetky znaky v reťazci jednobajtové (ak by sme skladali reťazec, ktorý obsahuje napr. `'č'`, vyhradili by sa po 2 bajty na každý znak v reťazci).

Otestujme teraz, ako je to s metódou `append()`:

```
import sys

def zisti(n):
    pole = []
    for i in range(n):
        size = sys.getsizeof(pole)
        print(f'len: {len(pole):3} sizeof: {size:5}')
        pole.append(None)

zisti(12)
```

Už vieme, že Python si pre `list` vyhradí vždy aj nejakú priestor pre základnú informáciu (36 bajtov) plus k tomu aj pamäť pre samotné prvky. Vidíme ale, že `append()` nezväčšuje vyhradenú pamäť o 1 prvok, ale vyhradí aj nejakú rezervu navyše:

```
len:   0 sizeof:    36
len:   1 sizeof:    52
len:   2 sizeof:    52
len:   3 sizeof:    52
len:   4 sizeof:    52
len:   5 sizeof:    68
len:   6 sizeof:    68
len:   7 sizeof:    68
len:   8 sizeof:    68
len:   9 sizeof:   100
len:  10 sizeof:   100
len:  11 sizeof:   100
```

Keďže každá referencia na hodnotu v poli zaberá u nás 4 bajty, upravíme testovací program tak, aby sme priamo videli, koľko prvkové pole sa momentálne vyhradilo. Tiež to upravíme tak, aby sme nevypisovali tie volania `append()`, ktoré nemenia vyhradenú pamäť:

```
import sys

def zisti(n):
    pole = []
    size0 = 0
    for i in range(n):
        size = sys.getsizeof(pole)
        if size != size0:
            size0 = size
            print(f'len: {len(pole):3} sizeof: {size:5} {(size-36)//4:3}')
        pole.append(None)

zisti(300)
```

Interval medzi nafukovaním poľa sa stále zväčšuje:

```
len:   0 sizeof:    36    0
len:   1 sizeof:    52    4
len:   5 sizeof:    68    8
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
len:   9 sizeof:   100   16
len:  17 sizeof:   136   25
len:  26 sizeof:   176   35
len:  36 sizeof:   220   46
len:  47 sizeof:   268   58
len:  59 sizeof:   324   72
len:  73 sizeof:   388   88
len:  89 sizeof:   460  106
len: 107 sizeof:   540  126
len: 127 sizeof:   628  148
len: 149 sizeof:   728  173
len: 174 sizeof:   840  201
len: 202 sizeof:   968  233
len: 234 sizeof:  1112  269
len: 270 sizeof:  1272  309
```

Python nielenže, nenafukuje pole pri každom volaní `append()`, ale nejako mení veľkosť nafukovanej časti.

Aby sme lepšie pochopili, ako to v Pythone funguje, definujme vlastné dynamické pole referencií na pythonovské objekty. Nebudeme pritom využívať štandardný typ `list` (ten chceme predsa naprogramovať) a tiež nevyužijeme žiadne kompaktné pole (`array.array()`) čísel - my potrebujeme pole referencií, t.j. pole smerníkov na ľubovoľné pythonovské hodnoty. Využijeme nízkoúrovňové (na úrovni jazyka C) definovanie poľa pomocou modulu `ctype`. Naprogramujeme aj metódu `append()`, v ktorej (ak bude treba) budeme naše pole nafukovať vždy o dvojnásobok jej momentálnej dĺžky:

```python
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):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        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(f'len: {len(pole):3} sizeof: {size:5}')
        pole.append(None)

zisti(300)
```

naozaj sa v prípade nutnosti veľkosť poľa zdvojnásobuje:

```
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
len: 257 sizeof:   512
```

Vďaka tomuto mechanizmu má „priemerný" čas operácie `append()` zložitosť **O(1)**, hovoríme tomu **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 vtedy, keď interval zväčšovania veľkosti poľa nebude konštantný ale nejaká geometrická postupnosť
- ak by sme nakreslili graf, ako závisí čas každej jednej operácie `append()` v závislosti od veľkosti poľa, vyzeralo by to približne takto:

```
|                   *
|                   *
|           *       *
|           *       *
|           *       *
|           *       *
|      *    *       *
|      *    *       *
|   *  *    *       *
| * *  *    *       *
|***********************************
|___________________________________
```

– teda zložitosť je skoro stále **O(1)** len veľmi zriedkakedy vyskočí dosť vysoko a závisí to od momentálneho `n`, lebo taký veľký úsek treba v pamäti vyhradiť a presťahovať sem starý obsah poľa

– ak by sme ale všetky vyčnievajúce stĺpiky v tomto grafe sklopili do nasledujúcich stĺpcov (kde je momentálne po jednej hviezdičke), tak v každom stĺpci budú práve dve hviezdičky a teraz už nikde nič netrčí - vyzerá to, akoby v každom stĺpci (pre každé `n`) bolo priemerne **O(2)**

– teraz je jasné, že ak by sme pole nezväčšovali so stále rastúcim prírastkom (dvojnásobok, alebo nejaký geometrický rad), nemohli by sme to takto ľahko spriemerovať a takýto `append()` by mal v konečnom dôsledku zložitosť **O(n)**

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 * 1e6 / n

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

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

```
       100      0.000
      1000      0.000
     10000      0.150
    100000      0.140
   1000000      0.175
  10000000      0.177
 100000000      0.186
```

V testovacom programe sme nameraný čas ešte vynásobili konštantou 1000000. Inak by boli namerané čísla príliš malé a Python by to zaokrúhlil na 0. Takže vidíme, že jedno volanie `append()` trvá (na mojom počítači) priemerne 0.17 milióntiny sekundy.

2.1.3 Zložitosť pythonovských operácií na sekvenčných typoch

Operácie, ktoré nemenia obsah `list`, resp. `tuple` (sú **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` (tiež !=, <, <=, >, >=)	O(k+1)
`pole[j:k]`	O(k-j+1)
`pole1 + pole2`	O(n1+n2)
`c * pole`	O(cn)

Operácie, ktoré menia obsah `list` (sú **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, n2 označuje veľkosť pole2)

Z týchto dvoch tabuliek je vidieť, že pri práci s poľom sa oplatí rozmýšľať, ktoré operácie použijeme. Napr. ak by sme posledný testovací program volania metódy `pole.append(None)` zmenili na testovanie `pole = pole + [None]`, dostali by sme približne takéto výsledky:

```
    100      0.000
   1000      3.000
  10000     23.103
 100000    229.964
1000000   4284.291
```

čo naznačuje, že už to nie je **O(1)** ale naskôr **O(n)**

Tiež si všimnite rozdiel medzi `pole1 + pole2` a `pole1 += pole2`. Preto:

```
pole = pole + [x]      # zložitosť O(n)
pole += [x]            # zložitosť O(1)* amortizovaná
```

2.1.4 Znakové reťazce

Keďže znakové reťazce (typ `str`) sú v Pythone **immutable**, vždy sa konštruuje nový reťazec:

```
s1 = '... dlhý reťazec ...'
s2 = ''
for znak in s1:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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 by sme využili pole a operáciu append():

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

Dostávame zložitosť **O(n)**. Samozrejme, že tvorcovia Pythonu o tomto probléme s rýchlosťou reťazcových operácií vedia, preto sú naprogramované v jazyku C čo najefektívnejšie. Takže, ak by sme merali reálnu rýchlosť oboch algoritmov, zistili by sme, že skutočná rýchlosť nemá s použitím polí až také zrýchlenie.

Aj prácu s reťazcami môžeme ešte urýchliť aj týmito zápismi:

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

alebo ešte lepšie:

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

2.2 Iterovateľný typ

Už vieme, že 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 ...]

Aj pre vlastný definovaný typ môžeme zabezpečiť **iterovateľnosť**. Možností je niekoľko, 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**

Pozrime sa najprv na 2. spôsob vytvorenia iterovateľnosti a to pomocou štandardných funkcií iter() a next() (pomocou metód __iter__() a __next__() vieme zabezpečiť funkčnosť aj pre našu novú triedu). Aby sme lepšie pochopili ich princíp fungovania, vysvetlime, ako Python „vidí“ obyčajný **for**-cyklus. Python ho vnútorne realizuje 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)
```

v skutočnosti Python realizuje pomocou iterátora približne takto:

```
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
```

Zadefinujme vlastnú triedu s metódou `__getitem__()`:

```
class Moj:
    def __init__(self):
        self.p = []

    def append(self, x):
        self.p.append(x)

    def __getitem__(self, i):
        return self.p[i]
```

Otestujme:

```
>>> a = Moj()
>>> for i in 'Python':
        a.append(i)
>>> a
<__main__.Moj object at 0x02AD0F70>
>>> for i in a:
        print(i, end=' ')
P y t h o n
>>> len(a)
...
TypeError: object of type 'Moj' has no len()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> list(a)
['P', 'y', 't', 'h', 'o', 'n']
>>> len(list(a))
6
```

Ak v nejakej 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 má ešte nasledovnú hodnotu (vtedy ju vráti) alebo vyvolá výnimku `StopIteration`.

Vyskúšajte:

```
class Moj2:
    def __init__(self):
        self.p = []

    def append(self, x):
        self.p.append(x)

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

    def __next__(self):
        if self.ix >= len(self.p):
            raise StopIteration
        self.ix += 1
        return self.p[self.ix-1]
```

Otestujeme:

```
>>> a = Moj2()
>>> for i in 2, 3, 5, 7, 11:
            a.append(i)
>>> for i in a:
            print(i, end=', ')
2, 3, 5, 7, 11,
>>> print(*a)
2 3 5 7 11
```

Poznámka: táto verzia iterátora je veľmi zjednodušená a niekedy nepracuje úplne korektne. Napr. pre obyčajné polia funguje:

```
a = [2, 3, 5, 7, 11]
for i in a:
    for j in a:
        print(i, j)
```

sa vypíše 25 dvojíc čísel. Pre náš iterovateľný objekt:

```
a = Moj2()
for i in 2, 3, 5, 7, 11:
    a.append(i)
for i in a:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    for j in a:
        print(i, j)
```

sa vypíše len 5 dvojíc. Zamyslite sa nad tým, ako by sa to dalo opraviť.

2.3 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

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 (možno budete musieť zmeniť konštanty `36` a `4`)

   ```
   def zisti(n):
       pole = []
       size0 = 0
       for i in range(n):
           size = sys.getsizeof(pole)
           if size != size0:
               size0 = size
               print(f'len: {len(pole):3} sizeof: {size:5} {(size-36)//4:3}')
           pole.append(None)
   ```

   - údaje vložte do excelovskej tabuľky a graficky znázornite priebeh funkcie, na základe ktorej Python určuje veľkosť pridávanej pamäti pri metóde `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) - nezahrňte do tohto času načítanie reťazca

    - môžete využiť napr. stránky s textami na internete: biblia v angličtine 4.24 MB, Vojna a mier od Tolsteho 3.14 MB alebo manuál k programovaniu PC hier 1.95 MB

  - porovnajte všetky 3 rôzne rýchle verzie (ďalšie dve sú s generátorovou notáciou)

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 = [... vel'ké pole ...]
    start = time.time()
    ...
```

4. Do triedy `DynamickePole` dodefinujte metódu `pop()` - navrhnite vlastnú stratégiu, kedy a o koľko sa bude pole zmenšovať (napr. vtedy, keď je vyhradená pamäť `n` využitá len na `n/4`, tak sa zmenší 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 prvky takéhoto pomocou for-cyklu)

  - napr.

```
pole = DynamickePole()
pole.append(13)
for i in pole:
    print(i)
```

  - pridajte metódu `__getitem__()` a otestujte iterovateľnosť:

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

  - namiesto metódy `__getitem__()` pridajte metódy `__iter__()` a `__next__()` a otestujte iterovateľnosť:

```
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,...):

   (a) funkcia `fib1(n)`, ktorá rekurzívne volá samu seba pre (n-1) aj pre (n-2) člen

   (b) funkcia `fib2(n)`, ktorá rekurzívne volá samu seba, ale vždy vracia dvojicu hodnôt (n-tý člen, n-1-člen), môže fungovať výrazne efektívnejšie ako `fib1()`

(c) funkcia `fib3(n)`, ktorá pracuje skoro rovnako ako funkcia `fib1(n)`, ale pamätá si v nejakej globálnej tabuľke všetky doterajšie vypočítané hodnoty a ak by nejakú mala znovu počítať, tak ju len vyberie z tabuľky; zrejme na začiatku je táto tabuľka prázdna a vždy keď nejakú ďalšiu hodnotu vypočíta, skôr ako ju vráti ako výsledok funkcie, zapamätá si ju aj v tabuľke

- odhadnite zložitosť všetkých verzií, odmerajte beh aj pre väčšie `n` (asi `fib1()` na niektorých vstupoch už nepobeží)

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 (rekurzívne), či sú všetky rôzne, ak sa vynechá prvý␣
↪prvok
           (2) potom zistí (rekurzívne), 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

3. Stromy a generátory

Keď sme ale definovali triedu **rad**, vedeli sme to urobiť tak, aby práca s radom nezávisela od konkrétnej realizácie.

```
class Queue:
    def __init__(self):
        '''nicializácia dátovejh štruktúry'''

    def is_empty(self):
        '''zistí, či je rad prázdny'''

    def enqueue(self, data):
        '''vloží na koniec radu'''

    def dequeue(self):
        '''vyberie zo začiatku radu'''

    def front(self):
        '''vráti prvý prvok radu'''
```

Práca s radom bude vždy rovnaká bez ohľadu na to, či je realizovaná poľom, spájaným zoznamom alebo cyklickým poľom. Preto sa budeme zaoberať štruktúrou strom tak, aby metódy nezáviseli od konkrétnej realizácie.

V prvom ročníku sme dátovú štruktúru **strom** riešili vždy len ako spájanú štruktúru, t. j. potomkovia vrcholu boli uložení ako referencie na objekty:

```
class BinarnyStrom:

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

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __len__(self):

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

        return pocet_rek(self.root)

    def preorder(self):

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

        return preorder_rek(self.root)

    def postorder(self):

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

        return postorder_rek(self.root)

    def inorder(self):

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

        return inorder_rek(self.root)
```

podobne pre všeobecný strom:

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

    def __init__(self):
        self.root = None

    def __len__(self):

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

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

        return pocet_rek(self.root)

    def __repr__(self):

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

        return repr1(self.root)
```

Pripomeňme si, ako sme mohli skonštruovať nejaký strom a potom v nejakom poradí vypísať všetky vrcholy:

```
>>> t = BinarnyStrom()
>>> t.root = t.Vrchol(11)
>>> t.root.left = t.Vrchol(12)
>>> t.root.right = t.Vrchol(13)
>>> t.root.right.left = t.Vrchol(14)
>>> t.root.right.right = t.Vrchol(15)
>>> t.root.left.right = t.Vrchol(16)
>>> t.preorder()
'11 12 16 13 14 15 '
>>> t.inorder()
'12 16 11 14 13 15 '
```

Ak by sme potrebovali realizovať strom nejakým iným spôsobom, museli by sme úplne zmeniť spôsob manipulácie s vrcholmi stromu - nemohli by sme takto jednoducho pracovať s ľavými a pravými potomkami vrcholov.

Stromy

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 susedné 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 vrcholu `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

ADT je taký popis dátového typu, ktorý je nezávislý od konkrétnej implementácie. Cieľom je zjednodušiť a hlavne sprehľadniť programy, ktoré pracujú s daným dátovým typom. ADT teda obsahuje metódy, ktoré by mali fungovať úplne rovnako bez ohľadu na konkrétnu implementáciu pomocou nejakej dátovej štruktúry. V spojitosti s ADT sa často spomína duck typing. Tento označuje taký programátorský štýl, pri ktorom sa nezisťuje typ objektu ale len metódy objektu, resp. sa používajú atribúty objektu („ak nejaký vták chodí ako kačica, pláva ako kačica a kváka ako kačica, tak potom je to kačica").

Vymenujme základné metódy, ktoré charakterizujú štruktúru **strom**, teda ADT pre strom sa skladá z metód:

- `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 ďalej odvádzať ďalšie triedy:

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

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

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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 odvodených 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: `ABC`, 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 `ABC` 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 ABC, abstractmethod

class Tree(ABC):

    @abstractmethod
    def root(self):
        pass

    @abstractmethod
    def parent(self, node):
        pass

    @abstractmethod
    def children(self, node):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        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.

3.1.1 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(ABC):

    ...

    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(ABC):

    ...

    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)**)

3.2 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`

3.2.1 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())
```

3.2.2 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. úroveň, potom 2. úroveň, ...
- **inorder** - najprv ľavý podstrom, potom koreň a na záver postupne všetky zvyšné podstromy
  - väčšinou sa definuje iba pre binárne stromy

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):

    ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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(ABC):

    ...

    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
```

Postorder

Postorder zapíšme podobne ako preorder, ktorý vytvára pole vrcholov:

```
class Tree(ABC):

    ...

    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')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
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()
```

Zamyslite sa ným, ako by ste zrealizovali metódu `postorder()` ako iterátor, čo znamená, že každé volanie `next()` vráti nasledovný prvok postupnosti postorder. Bez generátorov v nasledovnej časti by to bol programátorsky netriviálny problém.

3.3 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
```

Teda 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** - bude to ďalší spôsob vytvárania iterátora. 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** (pri iterátoroch sme tomu hovorili iterátorový 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
```

3.3.1 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, ktorý generuje `yield`, 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)        # alebo reversed(range(n))

>>> 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]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> 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.

3.3.2 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 n-ticu (`tuple`), ale aj slovník (`dict`). Hovoríme tomu **list comprehension** (resp. iný typ) - po slovensky **generované zoznamy** (niekedy aj generátorový zápis alebo notácia). Všimnite si, že n-ticu musíme generovať pomocou funkcie `tuple()`, lebo inak:

```
>>> urob = (i for i in range(20) if i%7 in [2,3,5])
>>> urob
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
<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]
```

3.3.3 Generátory pri stromoch

Najprv zmeňme metódu `children()` na generátorovú 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(ABC):

    ...

    def preorder(self):

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        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())
```

3.3.4 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(ABC):

    ...

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

v tomto konkrétnom prípade bude fungovať aj:

```
class Tree(ABC):

    ...

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

3.4 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Vytvorte a otestujte tieto štyri verzie funkcie `mocniny(n)`, ktorá
   - vráti postupnosť (`list`) druhých mocnín `[1, 4, 9, 16, ..., n**2]` vytvorenú pomocou for-cyklu a metódy `append()`
   - vráti postupnosť (`list`) 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`)
2. Zapíšte generátor `frange(zac, kon=None, krok=1)`, ktorý funguje podobne ako štandardná funkcia `range(...)`, ale nielen s celými ale aj s desatinnými číslami (`float`).
   - napr.

```
>>> g = frange(3)
>>> g
<generator object frange at 0x7f87e8d78e60>
>>> list(g)
[0, 1, 2]
>>> list(frange(1, 5, 0.5))
[1, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
>>> list(frange(10, 0, -1.25))
[10, 8.75, 7.5, 6.25, 5.0, 3.75, 2.5, 1.25]
```

3. Zapíšte dve verzie funkcie `map(funkcia, pole)`, ktorá vráti prvky poľa prerobené funkciou `funkcia` (podobne ako to robí štandardná funkcia `map()`, ale riešte to bez tejto funkcie)
   - výsledok vytvorte najprv ako pole
   - výsledok ako generátor
   - otestujte, ako sa bude správať `map()`, ak druhým parametrom je nejaký generátor
   - porovnajte so štandardnou funkciou `map()`
4. 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 nejaký generátor
   - porovnajte so štandardnou funkciou `filter()`
5. Zapíšte funkciu `zdvoj(gen)`, ktorá vygeneruje každý prvok 2-krát za sebou - funkcia vráti generátor
   - vyskúšajte nielen s parametrom typu generátor, ale napr. aj s poľom alebo s reťazcom

     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]
>>> zdvoj('Python')
...
>>> zdvoj([2, 3, 5])
```

6. 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
```

7. 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 pomocného 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
```

8. Na prednáške sa postupne definovali triedy `Tree`, `BinaryTree` a `LinkedBinaryTree`. Zapíšte ich postupne do súborov a otestujte funkčnosť vygenerovaním náhodného stromu s 20 vrcholmi a vygenerovaním `preorder()` a `postorder()` (tieto dve metódy 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(...):
    ...
```

9. 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: najprv koreň, potom jeho synov, potom jeho vnukov, ... (zrejme využijete nejaký svoj vlastný **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()`

10. 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`)

11. 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):
        ...
```

KAPITOLA 4

4. Prioritné fronty

Dátová štruktúra front (rad) umožňuje uchovávať ľubovoľné dáta a poskytuje ich presne v tom poradí, ako sem boli vkladané. Pripomeňme si túto štruktúru z 1. ročníka, ktorá bola realizovaná typom `list`:

```
class EmptyError(Exception): pass

class Queue:

    def __init__(self):
        '''inicializuje pole'''
        self._pole = []

    def enqueue(self, data):
        '''na koniec radu vlozi novu hodnotu'''
        self._pole.append(data)

    def dequeue(self):
        '''zo zaciatku radu vyberie prvu hodnotu, alebo vyvola EmptyError'''
        if self.empty():
            raise EmptyError('prazdny rad')
        return self._pole.pop(0)

    def front(self):
        '''zo zaciatku radu vrati prvu hodnotu, alebo vyvola EmptyError'''
        if self.empty():
            raise EmptyError('prazdny rad')
        return self._pole[0]

    def empty(self):
        '''zisti, ci je rad prazdny'''
        return self._pole == []
```

Niekedy môžeme potrebovať trochu zmenené správanie frontu:

- už pri vkladaní dát do frontu zadávame aj ešte jednu špeciálnu hodnotu, tzv. **kľúč**, ktorý označuje **prioritu** tohto prvku: čím menšia hodnota kľúča, tým vyššia priorita, teda dôležitosť vloženého prvku

- pri vyberaní z frontu dostávame nie ten prvok, ktorý bol vložený ako prvý, ale ten, ktorý má najmenší kľúč
- okrem vkladania a vyberania z takéhoto frontu sa vieme pozrieť aj na prvok, ktorý má najnižší kľúč (najlepšiu prioritu), teda ten, ktorý sa bude najbližšie z frontu vyberať

Abstraktný dátový typ

Budeme sa zaoberať tým, ako čo najefektívnejšie realizovať takýto front. Budú nás zaujímať práve operácie s takouto štruktúrou. Najprv sa teda dohodneme, aké operácie budeme očakávať pre všetky možné realizácie:

- `p.is_empty()` - zistí, či je front prázdny
- `len(p)` - vráti počet prvkov vo fronte
- `p.add(key, value)` - pridá 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 dvojicu `(key, value)` (typ `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 dvojicu `(key, value)` (typ `tuple`) prvku s najmenším kľúčom a zároveň ho z frontu odstráni

Aby sme zabezpečili, že niektorá konkrétna realizácia prioritného frontu naozaj zrealizuje všetky požadované operácie, zadefinujeme príslušnú abstraktnú triedu. Zrejme všetky ďalšie triedy budú od tejto abstraktnej odvodené.

```
from abc import ABC, abstractmethod

class Empty(Exception): pass

class PriorityQueue(ABC):

    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):
        '''vráti počet prvkov'''
        pass

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

    @abstractmethod
    def min(self):
        '''vráti dvojicu (key, value) s najmenším kľúčom'''
        pass
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    @abstractmethod
    def remove_min(self):
        '''vráti a odstráni dvojicu (key, value) s najmenším kľúčom'''
        pass

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

Všimnite si, že na uchovávanie dvojíc použijeme pomocnú triedu `_Item` a nie pythonovskú dvojicu (typ `tuple`). Dôvod je taký, aby sme lepšie vyjadrili, že pri vzájomnom porovnávaní dvoch dvojíc `(key1, value1)` a `(key2, value2)` musíme porovnávať len hodnoty kľúčov `key1` a `key2` a nemali by sme nikdy navzájom porovnávať hodnoty `value1` a `value2` (o týchto dvoch nemáme žiadne predpoklady, že sú to porovnávateľné dáta). Ak by sme totiž urobili `if (key1, value1) < (key2, value2)` a kľúče by sa rovnali, Python by porovnával ďalšie prvky dvojice a tu by to mohlo spadnúť.

Pomocou neutriedenej postupnosti

Začneme realizáciou pomocou neutriedenej postupnosti, t. j. dvojice `(key, value)` budeme ukladať do pythonovského poľa (typ `list`). Novú dvojicu budeme pridávať na momentálny koniec poľa (operácia `list.append()` je rýchla) a na vyhľadanie minimálneho kľúča budeme musieť prejsť a porovnať všetky prvky poľa. Ak by sme na realizáciu namiesto poľa použili dvojsmerný spájaný zoznam, vkladanie aj vyhľadanie by bolo rovnako rýchle (pomalé), ale vyberanie prvku z frontu je rýchlejšie pri zozname ako z poľa.

Zapíšme túto implementáciu:

```
class UnsortedPriorityQueue(PriorityQueue):

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

    def __len__(self):
        '''vráti počet prvkov'''
        return len(self._data)

    def add(self, key, value):
        '''pridá dvojicu (key, value)'''
        self._data.append(self._Item(key, value))

    def _find_min(self):
        '''pomocná funkcia:
             vráti index dvojice (key, value) s najmenším kľúčom
        '''
        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):
        '''vráti dvojicu (key, value) s najmenším kľúčom'''
        index = self._find_min()
        item = self._data[index]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        return item.key, item.value

    def remove_min(self):
        '''vráti a odstráni dvojicu (key, value) s najmenším kľúčom'''
        index = self._find_min()
        item = self._data.pop(index)
        return item.key, item.value
```

- keďže `min()` hľadá minimálny prvok v neutriedenom poli, musí vždy 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 prvku by stálo iba **O(1)**, ale aj tak má táto operácia s hľadaním minima zložitosť **O(n)**

Tabuľka 1: Zložitosť operácií

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')]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
remove_min= (7, 'D') pole= [(9,'B')]
remove_min= (9, 'B') pole= []
is_empty= True

Empty: priority queue is empty
```

Pomocou utriedenej postupnosti

Pokúsme sa zefektívniť operácie prioritného frontu tak, že pythonovské pole (typ `list`) budeme udržiavať utriedené. Zrejme budeme musieť prvky vkladať do poľa na správne miesto, teda pred všetky s väčšou hodnotou kľúča. Hľadanie a vyberanie minimálnej hodnoty bude veľmi jednoduché Opäť použijeme pythonovské pole - vhodnejší by bol dvojsmerný spájaný zoznam. tak, aby boli stále utriedené:

```
class SortedPriorityQueue(PriorityQueue):

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

    def __len__(self):
        '''vráti počet prvkov'''
        return len(self._data)

    def add(self, key, value):
        '''pridá 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):
        '''vráti dvojicu (key, value) s najmenším kľúčom'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = self._data[0]
        return item.key, item.value

    def remove_min(self):
        '''vráti a odstráni dvojicu (key, value) s najmenším kľúčom'''
        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)**

Tabuľka 2: Zložitosť operácií

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

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: priority queue is empty
```

Uvažujte:

- Ako by sa zmenila implementácia tejto triedy, keby sme pole udržiavali utriedené zostupne, teda minimálny prvok by bol posledný? Zmenila by sa zložitosť operácií?
- Ako by sa zmenila implementácia tejto triedy, keby sme namiesto poľa využili spájaný zoznam? Zmenila by sa zložitosť operácií?

4.1 Pomocou haldy

Tretia verzia implementácie prioritného frontu využíva dátovú štruktúru **halda**, ktorá má veľmi užitočné vlastnosti:

halda = heap

je binárny strom, ktorý musí spĺňať tieto dve podmienky:

- **vzťah medzi vrcholmi**: každý vrchol (okrem koreňa) má hodnotu kľúča väčšiu alebo rovnú ako hodnota kľúča v jeho predkovi
  - vďaka tejto vlastnosti je v koreni minimálna hodnota kľúča z celého stromu
- **tvar stromu**: chceme, aby strom mal minimálnu hĺbku, preto budeme požadovať takúto vlastnosť (skoro) úplného binárneho stromu:
  - okrem najnižšej úrovne sú všetky úrovne úplné
  - v najnižšej úrovni sú všetky vrcholy umiestňované len zľava
- výška takéhoto binárneho stromu je **log n**

4.1.1 Halda implementovaná v poli

Keďže je halda „skoro“ úplný binárny strom, nemusíme ju realizovať pomocou spájanej štruktúry, ale použijeme pythonovské pole (typ `list`):

- do poľa budeme prvky stromu ukladať po úrovniach:
  - najprv koreň (do 0. prvku)
  - potom dva vrcholy z ďalšej úrovne (obaja potomkovia koreňa)

  - za tým 4 vrcholy z 2. úrovne (najprv obaja potomkovia ľavého syna koreňa, potom obaja potomkovia pravého syna koreňa),
  - … 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 úplný 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 sa opakuje, kým nenatrafí na menšieho predka (teda je to už OK) alebo sa dostane až do koreňa stromu
  - takto sa opraví celá halda

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

- nesmieme naozaj vyhodiť 0. prvok poľa (koreň stromu), lebo by sa halda pokazila tak, že jej oprava by bola príliš náročná, namiesto toho:
- vyhodený koreň nahradíme posledným prvkom v halde (najpravejší v najnižšej úrovni) a haldu (pole) pritom o 1 skrátime
- asi sme tým opäť 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`)
  - ak je jeden z nich menší ako koreň, tak ho s koreňom vymením a znovu túto novú pozíciu porovnáva s jeho synmi
  - toto opakuje, kým neprídeme do listu stromu (vrchol už nemá ani jedného syna), alebo obaja synovia už nemajú menšiu hodnotu
  - takto sa opraví celá halda

Definícia triedy

Pre lepšiu čitateľnosť kódu, najprv pripravíme niekoľko pomocných funkcií a až potom zadefinujeme samotné operácie prioritného frontu:

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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):
        '''vráti počet prvkov'''
        return len(self._data)

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

    def min(self):
        '''vráti dvojicu (key, value) s najmenším kľúčom'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = self._data[0]
        return item.key, item.value

    def remove_min(self):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        '''vráti a odstráni dvojicu (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)**
  - uvedomte si, že 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:

Tabuľka 3: Zložitosť operácií

operácia	unsorted	sorted	heap	
`len()`	O(1)	O(1)	O(1)	
`is_empty()`	O(1)	O(1)	O(1)	
`add()`	O(1)*	O(n)	O(log n)*	
`min()`	O(n)	O(1)	O(1)	
`remove_min()`	O(n)	O(n)	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: priority queue is empty
```

Všimnite si, ako sa v poli uchováva halda: po každom vyhodení prvého prvku sa pole preuprace

4.2 Využitie modulu heapq

Idea uchovávania prvkov v poli tak, aby mali vlastnosť haldy, je pre programátorov tak užitočná, že v Pythone jeden zo štandardných modulov `heapq` pracuje s haldou, teda pracuje s obyčajným poľom (typ `list`), ale prvky udržiava v správnom poradí. Modul `heapq` ponúka tieto funkcie:

- `heappush(pole, prvok)` - vloží nový prvok do poľa, tak aby to stále bola halda (predpokladáme, že v poli už predtým bolo správne haldové usporiadanie) - zložitosť tejto funkcie je **O(log n)**
- `heappop(pole)` - robí to isté ako naše `remove_min()`, teda vráti z poľa 0. prvok, pritom ho z poľa vyhodí a pole preusporiada, aby to bola opäť halda - zložitosť tejto funkcie je **O(log n)**
- `heapify(pole)` - preusporiada pole tak, aby spĺňalo podmienky haldy - zložitosť tejto funkcie je **O(n)**

Zrejme prvky vkladané do poľa musia byť navzájom porovnávateľné, preto nemôžeme do haldy vkladať dvojice `(key, value)` (typ `tuple`), ak sú druhé zložky neporovnávateľné, napr. `(15, 3.14)` a `(15, 'a')` majú rovnaký kľúč a neporovnávateľné hodnoty.

Zapíšme definíciu triedy pomocou modulu `heapq`:

```
import heapq

class HeapPriorityQueue(PriorityQueue):

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

    def __len__(self):
        '''vráti počet prvkov'''
        return len(self._data)

    def add(self, key, value):
        '''pridá dvojicu (key, value)'''
        heapq.heappush(self._data, self._Item(key, value))

    def min(self):
        '''vráti dvojicu (key, value) s najmenším kľúčom'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = self._data[0]
        return item.key, item.value

    def remove_min(self):
        '''vráti a odstráni dvojicu (key, value) s najmenším kľúčom'''
        if self.is_empty():
            raise Empty('priority queue is empty')
        item = heapq.heappop(self._data)
        return item.key, item.value
```

Po otestovaní vidíme, že to funguje rovnako, ako naša vlastná realizácia haldy:

```
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')]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
remove_min= (7, 'D') pole= [(9,'B')]
remove_min= (9, 'B') pole= []
is_empty= True

Empty: priority queue is empty
```

4.2.1 Triedenie pomocou prioritného frontu

Dátovú štruktúru prioritný front môžeme použiť aj na utriedenie ľubovoľného poľa. Použijeme takýto algoritmus:

- najprv sa vytvorí zo všetkých prvkov poľa prioritný front (n-krát sa zavolá operácia `add()`)
  - ako kľúč použijeme samotnú hodnotu z poľa, preto
- potom opäť v cykle vyberáme stále najmenší prvok a vkladáme ho späť do poľa na správne miesto

Tento algoritmus nevracia žiadnu hodnotu, ale priamo triedi vstupné pole:

```
def sort(trieda, pole):
    p = trieda()                        # jedna z našich tried, ktorá realizuje␣
↪front
    for prvok in pole:
        p.add(prvok, prvok)             # kľúč bude rovnaký ako hodnota
    for i in range(len(pole)):
        pole[i] = p.remove_min()[1]     # z dvojice (key, value) zoberieme value
```

Ak zavoláme takéto triedenie pomocou triedy `UnsortedPriorityQueue` (teda volanie `sort(UnsortedPriorityQueue, pole)`), dostávame nám známy algoritmus triedenia **min-sort** (nazýva sa aj *selection-sort*). V tomto prípade sa stále vyberá minimálny prvok a vkladá sa za už utriedenú časť na začiatku poľa.

Ak zavoláme takéto triedenie pomocou triedy `SortedPriorityQueue` (teda volanie `sort(SortedPriorityQueue, pole)`), dostávame opäť nám známy algoritmus triedenia **insert-sort**. V tomto prípade už vytvorenie prioritného frontu (n-krát vloženie pomocou operácie `add()`) vytvorí usporiadané pole. Druhá časť algoritmu ho len prekopíruje do pôvodného poľa.

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

4.2.2 Triedenie pomocou prioritného frontu s haldou = heap sort

Použitie algoritmu `sort()` s triedou `HeapPriorityQueue` (teda volanie `sort(HeapPriorityQueue, pole)`) bude mať ale inú zložitosť ako v dvoch predchádzajúcich prípadoch:

- prvá časť algoritmu, ktorá pomocou operácie `add()` vytvára front (teda poskladá haldu zo všetkých prvkov poľa), má zložitosť **O(n log n)** (n-krát vloženie do haldy, ktorého zložitosť je **O(log n)**)
- druhá časť algoritmu, ktorá pomocou operácie `remove_min()` postupne vyberá minimálne prvky a vkladá ich na príslušné miesta v poli, má tiež zložitosť **O(n log n)** (n-krát vyberanie z haldy, ktorého zložitosť je **O(log n)**)
- celková zložitosť je teda súčet zložitostí týchto dvoch častí, teda opäť je to **O(n log n)**

Otestujeme rýchlosť tohto triedenia pre rôzne veľké vstupné polia:

```
import time
import random

for n in 1000, 10000, 100000, 1000000:
    pole = [random.randrange(10000) for i in range(n)]
    pole0 = pole[:]       # kontrolná kópia pôvodného pol'a

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

Pri výpise vidíme, že porovnanie výsledného poľa s utriedením pomocou štandardného pythonovského triedenia (`sorted()`) prebehlo v poriadku, teda vrátilo `True`:

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

Všimnite si, ako sa postupne mení čas, keď program spúšťame s 10-krát väčším poľom: zrejme je to o trochu väčší prírastok, ako keby to bola zložitosť **O(n)**, teda mohli by sme usudzovať, že zložitosť tohto algoritmu je naozaj **O(n log n)**.

Triedenie pomocou haldy (tzv. **heap-sort**) je teda rýchle triedenie a je porovnateľné napr. s **quick-sortom**. Táto naša realizácia ale využíva na triedenie pomocné pole rádovo veľkosti `n` (pole s haldou sa vytvorilo mimo samotného triedeného poľa). Ak by sme chceli zefektívniť takéto triedenie, asi by boli vhodné takéto vylepšenia:

- nepotrebujeme uchovávať dvojice `(key, value)`, stačí, keď budeme pracovať priamo s hodnotami, ktoré sú porovnávanými kľúčmi
- pracovať budeme priamo so vstupným poľom bez pomocného poľa, teda haldu vytvoríme preusporiadaním prvkov poľa
- aby sme v druhej fáze algoritmu nepotrebovali pomocné pole na postupné ukladanie minimálnych prvkov (`remove_min()`) do výsledného poľa, pravidlá pre haldu zmeníme tak, že v koreni bude maximálny prvok a všetky preusporiadania haldy (`_heap_up()` a `_heap_down`) budú robiť opačné testy; potom stačí maximálny prvok vymieňať s posledným prvkom haldy a namiesto skracovania haldy, len zmenšíme premennú, ktorá bude označovať počet prvkov haldy (namiesto `len(pole)` budeme vo všetkých metódach pracovať s touto premennou)

Aj vytváranie haldy (prvá fáza algoritmu heap-sort), ktorá má zložitosť **O(n log n)** sa dá ešte urýchliť tak, aby mala zložitosť **O(n)**. Tomuto sa hovorí algoritmus **heapify** (vie to aj modul `heapq`) a pracuje takto:

- postupne prechádza pole od konca
- pre každý `i`-ty prvok zabezpečí, aby sa podstrom s koreňom `i` 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 takto uhalduje celý strom
- uvedomte si, že v tomto cykle nemusíme začínať od vrcholov stromu, ktoré sú listami, stačí začínať s vrcholmi, ktoré majú aspoň jedného syna: teda až začnete od vrcholu s indexom `n // 2`

Zamyslite sa, či by ste vedeli dokázať, že tento algoritmus má naozaj zložitosť **O(n)**.

Z pohľadu zložitosti algoritmov takéto vylepšenia (bez pomocného poľa, otočená halda od maximálnej hodnoty, heapify, …) nemení celkovú zložitosť, stále je to **O(n log n)**, ale v reálnych situáciách sa robia aj takéto vylepšenia, lebo výrazne pomôžu skutočnému času behu.

4.3 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. 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 stromu rovnako ako metóda `add()`)
   - ručne z haldy niekoľkokrát odstráňte minimálny prvok (operácia `remove_min()`)
   - priebeh ukladania do haldy (aj vyberania najmenšieho) si môžete vizualizovať napr. na stránke Heap Visualization

2. Naprogramujte vytvorenie haldy (`HeapPriorityQueue`) a preverte, či je vaša halda z úlohy (1) rovnaká ako v tomto programe. Tiež skontrolujte, či aj vaše ručné odstraňovanie minimálnych prvkov haldu upratalo úplne rovnako ako v programe.
   - haldou vytvoríte pomocou:

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

   - postupné odstraňovanie najmenšieho prvku:

   ```
   for i in range(4):
       print(p.remove_min(), p._data)
   ```

3. Napíšte funkciu `kontrola_na_haldu(pole)`, ktorá skontroluje, či dané pole spĺňa podmienky haldy.
   - napr.

   ```
   >>> pole = [8, 13, 7, 10, 5, 15, 12, 17, 9, 14, 4, 11, 18, 16, 6]
   >>> kontrola_na_haldu(pole)
   False
   ```

   ```
   >>> p = HeapPriorityQueue()
   >>> ... pridá prvky pomocou p.add(...)
   >>> kontrola_na_haldu(p._data)
   True
   ```

   - zamyslite sa, či ľubovoľné vzostupne usporiadané pole je automaticky haldou? napr.

   ```
   >>> pole = list(range(30))
   >>> kontrola_na_haldu(pole)
   ???
   ```

4. Štvrtá úloha záverečného testu z minulého roku sa venovala haldám:

   Z rôznych čísel 1 až 10 vytvorte haldu (v koreni s indexom 0. je minimum) v 10-prvkovom poli tak, aby:

(a) v prvku poľa s indexom 4 bola maximálne možná hodnota

(b) v prvku poľa s indexom 4 bola minimálne možná hodnota

Pre obe podúlohy zrealizujte aj operáciu `remove_min()`.

5. Algoritmus, ktorý z poľa vytvára haldu (prvá fáza nášho triedenia `heap_sort()`) má zložitosť **O(n log n)** (n-krát sa urobí `_heap_up()` a ten má zložitosť **O(log n)**). Ak by sme ale haldu nevytvárali zhora postupným pridávaním prvku na koniec, ale naopak zdola, mohli by sme zabezpečiť zložitosť tejto fázy algoritmu **O(n)**. Tento algoritmus sa nazýva **heapify** a pracuje na takomto princípe:

   - pre jednoduchosť predpokladajme, že strom je úplny (najnižšia úroveň je kompletná)
   - každý vrchol najnižšej úrovne stromu je malá halda
   - postupne prechádzame predposlednú úroveň vrcholov: vytvoríme malé haldy z tohto vrcholu a jeho dvoch synov (algoritmus `_heap_down()`), ktoré sú už teraz malé haldy
   - potom prejdeme o úroveň vyššie a vytvoríme haldy s koreňmi v týchto vrcholoch, keďže ich synovia sú už malé haldy (opäť `_heap_down()`)
   - takto postupne prejdeme všetky vrcholy stromu až po koreň - keď už sme v koreni, celý strom je haldou

   Dá sa ukázať, že tento algoritmus má naozaj zložitosť **O(n)**. Na stránke Heap Visualization môžete vidieť aj algoritmus heapify, tu sa volá **BuildHeap**.

   - ručne vytvorte haldu z poľa z úlohy (1):

   ```
   >>> pole = [8, 13, 7, 10, 5, 15, 12, 17, 9, 14, 4, 11, 18, 16, 6]
   ```

6. Do triedy `HeapPriorityQueue` doprogramujte metódu `heapify(pole)`, ktorá dané pole najprv zapíše do `self._data` (ako dvojice `_Item`) a potom ho podľa algoritmu z úlohy (5) uprace na haldu.

   - odkontrolujte, či je takto vytvorené pole haldou, napr.

   ```
   >>> pole = [8, 13, 7, 10, 5, 15, 12, 17, 9, 14, 4, 11, 18, 16, 6]
   >>> p = HeapPriorityQueue()
   >>> p.heapify(pole)
   >>> kontrola_na_haldu(p._data)
   True
   ```

7. V niektorých situáciách je vhodné, aby prioritný front nepracoval s minimálnymi kľúčmi, ale s maximálnymi. Vytvorte verziu triedu `MaxHeapPriorityQueue`, ktorá implementuje metódy tejto triedy.

   - zrejme v koreni stromu je maximálny prvok a mnohé testy bude treba nejako otočiť:

   ```
   class MaxHeapPriorityQueue:
     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))
     #......................
     def __len__(self):
         '''vráti počet prvkov'''
         pass
   ```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def add(self, key, value):
        '''pridá dvojicu (key, value)'''
        pass

    def max(self):
        '''vráti dvojicu (key, value) s najväčším kľúčom'''
        pass

    def remove_max(self):
        '''vráti a odstráni dvojicu (key, value) s najväčším kľúčom'''
        pass

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

- túto triedu otestujte, napr. triedením pomocou tohto prioritného frontu

8. Triedenie `heap_sort()` sme doteraz implementovali pomocou prioritného frontu `HeapPriorityQueue` a vyzeralo to nejako takto:

   - heap-sort pomocou prioritného frontu:

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

   Namiesto použitia prioritného frontu môžeme haldu realizovať priamo v samotnom triedenom poli:

   - najprv sa pole preusporiada tak, aby spĺňalo kritérium haldy, t. j. `pole[i]<=pole[2*i+1]` a zároveň `pole[i]<=pole[2*i+2]`
     - upratovať môžete napr. tak, že postupne pre každý prvok od 1 do `len(pole)-1` spustíte podobný algoritmus ako bol `_heap_up()`
     - teraz už nepotrebuje pracovať s dvojicami `(key, value)`, ale pracujeme priamo s hodnotami - tieto hodnoty priamo porovnávame
   - keď je halda hotová, postupne sa odoberá 0. prvok (minimum) a odkladá sa na koniec poľa (samotné pole sa naozaj neskracuje pomocou `pop()` len sa zaeviduje, že už je o 1 kratšie) - zakaždým sa obnoví halda (niečo ako operácia `_heap_down()`)
   - obe tieto pomocné upratovacie funkcie `_heap_up()` aj `_heap_down()` prepíšte na ich nerekurzívnu verziu bez pomocných funkcií `_left()`, `_has_left()`, `_parent()`, ...
   - takto sa na koniec poľa postupne dostávajú minimálne prvky a tým dostávame utriedené pole ale vzostupne
   - keby sme operácie `_heap_up()` aj `_heap_down()` modifikovali ako pre `MaxHeapPriorityQueue`, dostali by sme vzostupne utriedené pole

9. Naimplementujte triedy `Stack` aj `Queue` pomocou `PriorityQueue`

   - to znamená, že na operácie `push()`, `pop()`, `enqueue()`, `dequeue()` použijete len `add` a `remove_min()`
   - aká je zložitosť týchto operácii

KAPITOLA 5

5. Asociatívne polia

S dátovou štruktúrou asociatívne pole (pythonovský typ `dict`, niekedy sa mu hovorí aj **map**, keďže sa mapujú kľúče na nejaké hodnoty) už máme dlhšie nemalé programátorské skúsenosti:

- je podobné obyčajnému poľu s tým rozdielom, že indexom nemusia byť len celé čísla od 0 po nejakú konkrétnu hodnotu, ale indexom môže byť skoro hocijaká hodnota, napr. aj desatinné čísla, reťazce aj n-tice hodnôt, napr.

```
>>> d = {}
>>> d[3.14] = 'pi'
>>> d['sto'] = 100
>>> d[100, 200] = turtle.Turtle()
```

- indexy asociatívnych polí nazývame kľúče a máme možnosť získať postupnosť všetkých kľúčov, napr.

```
>>> list(d.keys())    # metóda keys() je iterátor
[3.14, (100, 200), 'sto']
```

- vieme získať aj všetky hodnoty, ktoré sú asociované (namapované) na kľúče, napr.

```
>>> list(d.values())    # metóda values() je iterátor
['pi', <turtle.Turtle object>, 100]
```

- okrem tohoto vieme získať aj všetky dvojice (kľúč, hodnota), z ktorých je asociatívne pole zložené, napr.

```
>>> list(d.items())    # metóda values() je iterátor
[(3.14, 'pi'), ((100, 200), <turtle.Turtle object>), ('sto', 100)]
```

V prvom ročníku sme použili takéto asociatívne pole na zisťovanie počtu výskytov nejakých hodnôt (frekvenčná tabuľka), napr. v poli:

```
import random

pole = [random.randrange(100) for i in range(1000)]
d = {}
for prvok in pole:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    d[prvok] = d.get(prvok, 0) + 1

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

Tento program vypíše 5 čísel s najčastejším výskytom v danom poli. Kľúčmi asociatívneho poľa `d` sú prvky poľa `pole` a príslušnými hodnotami sú počty výskytov daného prvku. Kľúčmi nemusia byť len čísla, ale aj reťazce. Tento program bude fungovať napr. aj pre takéto pole slov:

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

Cieľom tejto témy je ukázať, akým spôsobom musia byť naprogramované tieto operácie (metódy), prípadne aká je ich zložitosť.

Abstraktný dátový typ

Budeme sa zaoberať rôznymi implementáciami a budeme skúmať efektívnosť zodpovedajúcich operácií. Preto najprv zvolíme minimálnu množinu operácií, ktoré charakterizujú tento typ, teda, keď ich zrealizujeme, môžeme tvrdiť, že sme implementovali asociatívne pole. V každom prípade je asociatívne pole dátovou štruktúrou, ktorá nejako uchováva dvojice (kľúč, hodnota) pričom kľúč je jedinečný, t. j. môže sa vyskytovať len v jednej dvojici.

Operácie budeme porovnávať s pythonovským typom `dict`, teda s premennou `d` a s dvojicou (`key`, `value`), teda pre asociatívne pole `a`:

- `a.valueof(key)` vráti príslušnú hodnotu k danému kľúču
  - ak pre daný kľúč neexistuje príslušná hodnota, metóda vyvolá `KeyError`
  - v Pythone tomu zodpovedá `d[key]`
- `a.add(key, value)` zaradí do asociatívneho poľa novú dvojicu (`key`, `value`)
  - ak už predtým v poli existovala dvojica s týmto kľúčom, tak ju nahradí novou dvojicou
  - v Pythone tomu zodpovedá `d[key] = value`
- `a.delete(key)` vyhodí z asociatívneho poľa dvojicu s daným kľúčom
  - ak daný kľúč neexistuje, metóda vyvolá `KeyError`
  - v Pythone tomu zodpovedá `del d[key]`
- `len(a)` (teda metóda `a.__len__()`) vráti počet prvkov (dvojíc) v asociatívnom poli
  - v Pythone tomu tiež zodpovedá `len(d)`
- `iter(a)` (teda metóda `a.__iter__()`) vráti iterátor, vďaka ktorému môžeme postupne prechádzať všetky kľúče
  - v Pythone tomu tiež zodpovedá `iter(d)`
  - najčastejšie iterovanie využijeme vo for-cykle: `for kluc in a: ...`

Zadefinujme abstraktnú triedu `MapBase`, ktorá bude obsahovať všetky tieto metódy, ale aj podtriedu `_Item` na uchovávanie dvojíc (`key`, `value`):

```
from abc import ABC, abstractmethod

class MapBase(ABC):

```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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 valueof(self, key):
        pass

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

    @abstractmethod
    def delete(self, key):
        pass

    @abstractmethod
    def __len__(self):
        pass

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

Zrejme sa táto trieda zatiaľ testovať nedá, musíme najprv implementovať nejakú konkrétnu realizáciu. V skutočnosti nebude rôzne realizácie implementovať ako triedy, ale pre čitateľnosť algoritmov ich zapíšeme len ako tri globálne funkcie `valueof(key)`, `add(key, value)` a `delete(key)`, ktoré pracujú s jedným globálnym poľom `table`. Z tohto dôvodu presťahujeme aj pomocnú lokálnu triedu `_Item` ako globálnu definíciu `Item`:

```
class Item:
    def __init__(self, key, value):
        self.key, self.value = key, value
    def __repr__(self):
        return repr(self.key) + ':' + repr(self.value)

table = [...]

def valueof(key):
    ...
    raise KeyError

def add(key, value):
    ...

def delete(self, key):
    ...
    raise KeyError
```

Podobne upravíme aj samotné otestovanie realizácie:

```
import random
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
pole = [random.randrange(100) for i in range(10000)]
d = {}
for i in pole:
    try:
        add(i, valueof(i) + 1)
    except KeyError:
        add(i, 1)
    d[i] = d.get(i, 0) + 1

set1 = {(it.key, it.value) for it in table}      # pripadne tu pozmeníme generovanie␣
↪dvojíc
set2 = set(d.items())
print(set1 == set2)
```

Asociatívne pole pomocou neutriedenej postupnosti

Najjednoduchším typom implementácie asociatívneho poľa bude použitie obyčajného pythonovského poľa (typ `list`), do ktorého budeme ukladať dvojice v tom poradí, ako prichádzajú (s novým kľúčom) pomocou operácie `add`. Základné operácie budeme teda realizovať takto:

- samotný obsah asociatívneho poľa budeme ukladať do tabuľky `table` typu `list`, prvkami tohto poľa budú neskôr dvojice typu `Item`
- funkcia `valueof(key)` vyhľadá `key` v poli `table`: keďže pole nie je nijako usporiadané, musí pole prehľadávať postupne cez všetky jeho prvky, keď nájde prvok s hľadaným kľúčom, vráti príslušnú hodnotu, inak vyvolá výnimku `KeyError`
- funkcia `add(key, value)` opäť vyhľadá prvok s daným kľúčom a ak ho nájde vymení mu príslušnú hodnotu, inak pridá nový prvok s kľúčom a hodnotou `(key, value)` na koniec poľa `table`
- funkcia `delete(key)` vyhľadá prvok s daným kľúčom a ak ho nájde, z poľa ho vyhodí, inak ak nenašiel prvok s hľadaným kľúčom, vyvolá výnimku `KeyError`

Zapíšme túto implementáciu:

```
# implementácia UnsortedMap

table = []

def valueof(key):
    for item in table:
        if key == item.key:
            return item.value
    raise KeyError

def add(key, value):
    for item in table:
        if key == item.key:
            item.value = value
            return
    table.append(Item(key, value))

def delete(self, key):
    for ix in range(len(table)):
        if key == table[ix].key:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            del table[ix]
            return
    raise KeyError
```

Môžete prekontrolovať, že tento program bude fungovať nielen pre pole čísel, ale aj pole reťazcov. Ak ale pole reťazcov bude trochu väčšie (vytvoríme ho napr. prečítaním zo súboru), výpočet môže trvať aj desiatky sekúnd.

Dôvodom, prečo je realizácia neutriedeným poľom tak pomalá, je ten, že operácie `add()` aj `valueof()` sú výrazne pomalšie ako originálne pythonovské operácie s asociatívnym poľom. Zapíšme do tabuľky zložitosť operácií našej triedy:

Tabuľka 1: Zložitosť operácií

operácie	unsorted	
`valueof()`	**O(n)**	
`add()`	**O(n)**	
`delete()`	**O(n)**	

Teraz by malo byť jasné, prečo náš testovací program s frekvenčnou tabuľkou môže trvať tak dlho: for-cyklus, ktorý zvyšuje o 1 ďalší výskyt hodnoty, má zložitosť **O(n**2)**. Toto pre veľké **n** môže naozaj trvať dosť dlho.

Asociatívne pole pomocou utriedenej postupnosti

Predchádzajúcu implementáciu neutriedeným poľom vylepšíme tak, že pole `table` budeme udržiavať utriedené podľa kľúča. Hoci v tejto verzii predpokladáme jedno dôležité obmedzenie, aj tak to otestujeme. Tým obmedzením je podmienka, že všetky kľúče sa musia dať navzájom porovnávať. To ale znamená, že v našej novej implementácii by už nemalo šancu prejsť ani pythonovské:

```
>>> d = {}
>>> d[3.14] = 'pi'
>>> d['sto'] = 100
```

Kľúče `3.14` a `'sto'` sa nedajú navzájom porovnať na to, aby sme zistili, ktorý z nich je menší. Keď ale zanedbáme toto obmedzenie, dostávame realizáciu, ktorá má šancu byť rýchlejšia ako `UnsortedMap`.

V implementácii sme použili pomocnú funkciu `find_index()`, pomocou ktorej veľmi rýchlo vyhľadáme index požadovaného kľúča.

```
# implementácia SortedMap

table = []

def find_index(key, low, high):
    '''vrati bud index kluca, alebo poziciu, kam vlozit'''
    if high < low:
        return high + 1
    else:
        mid = (low + high) // 2
        if key == table[mid].key:
            return mid
        elif key < table[mid].key:
            return find_index(key, low, mid - 1)
        else:
            return find_index(key, mid + 1, high)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
def valueof(key):
    ix = find_index(key, 0, len(table) - 1)
    if ix == len(table) or table[ix].key != key:
        raise KeyError
    return table[ix].value

def add(key, value):
    ix = find_index(key, 0, len(table) - 1)
    if ix < len(table) and table[ix].key == key:
        table[ix].value = value
    else:
        table.insert(ix, Item(key, value))

def delete(key):
    ix = find_index(key, 0, len(table) - 1)
    if ix == len(table) or table[ix].key != key:
        raise KeyError
    del table[ix]
```

Ak teraz spustíme rovnaké testy, ako pri predchádzajúcej implementácii, môžeme vidieť, že aj táto verzia je funkčná, dokonca je o trochu rýchlejšia, ale stále pre veľké pole trvá desiatky sekúnd. Doplňme tabuľku zložitosti operácií aj pre túto triedu:

Tabuľka 2: Zložitosť operácií

operácie	unsorted	sorted	
`valueof()`	**O(n)**	**O(log n)**	
`add()`	**O(n)**	**O(log n), O(n)**	
`delete()`	**O(n)**	**O(n)**	

Operácia `add()` má zložitosť **O(log n)** len v prípade, že daný kľúč sa v poli nachádzal už predtým. Inak musí vložiť nový prvok `(key, value)` na správne miesto do poľa `table`. Keďže toto vkladanie sa robí metódou `insert()`, zložitosť operácie `add()` bude vtedy **O(n)**.

5.1 Hašovacia tabuľka

V ďalšej časti budeme v niekoľkých krokoch vylepšovať jednu novú ideu, ktorú postupne dotiahneme až do veľmi zaujímavej zložitosti operácií.

5.1.1 Index ako kľúč

Začneme tým, že budeme predpokladať:

- kľúče budú len celé čísla
- kľúče budú mať hodnoty len z intervalu `<0, max-1>` (pre dopredu známe `max`, tzv. kapacita tabuľky)

Vďaka tomuto obmedzeniu môžeme výrazne zjednodušiť realizáciu všetkých operácií:

- už na začiatku sa v poli `table` vyhradí maximálna kapacita a do každého prvku sa priradí `None`
- funkcia `valueof(key)`: keďže `key` je indexom do poľa, stačí pozrieť, či tam nie je `None` (vtedy vyvolá výnimku `KeyError`) a vráti hodnotu v tomto prvku; zrejme skontroluje aj to, či je kľúč zo správneno intervalu

- funkcia `add(key, value)`: ak je `key` index zo správneho intervalu, priamo na túto pozíciu priradí novú hodnotu
- funkcia `delete(key)`: ak je `key` index zo správneho intervalu, do poľa na príslušné miesto priradí `None`

Implementácia:

```
# implementácia TestMap

table = [None] * 1000     # nejaka predpokladana kapacita tabulky

def valueof(key):
    if key < 0 or key >= len(table) or table[key] is None:
        raise KeyError
    return table[key].value

def add(key, value):
    if key < 0 or key >= len(table):
        raise KeyError
    table[key] = Item(key, value)

def delete(key):
    if key < 0 or key >= len(table) or table[key] is None:
        raise KeyError
    table[key] = None
```

Zložitosť všetkých operácií je teraz **O(1)**:

Tabuľka 3: Zložitosť operácií

operácie	unsorted	sorted	test	
`valueof()`	**O(n)**	**O(log n)**	**O(1)**	
`add()`	**O(n)**	**O(n)**	**O(1)**	
`delete()`	**O(n)**	**O(n)**	**O(1)**	

Vyzerá to veľmi optimisticky. Naozaj aj test s väčším počtom dvojíc, kde je kľúč aj z väčšieho intervalu, prejde veľmi rýchlo:

```
pole = [random.randrange(1000000) for i in range(50000)]
...
```

Zrejme veľkosť poľa `table` treba nastaviť na maximálnu hodnotu kľúča, teda na 1000000.

5.1.2 Riešenie kolízií

V ďalšom kroku vylepšovania tejto idey zrušíme predpoklad, vďaka ktorému sme všetky prvky asociatívneho poľa mohli ukladať priamo do jedného veľkého poľa, lebo kľúče zodpovedali indexom do poľa (od 0 do nejakého max-1). Teraz vyhradíme menšie pole, ako je rozsah očakávaných kľúčov a pozíciu v poli zistíme tak, že kľúč vydelíme veľkosťou tohto poľa a zvyšok po delení bude hľadaný index. Kvôli tomuto sa ale môže stať to, že pre dva rôzne kľúče dostávame rovnaký index do poľa. Tomuto hovoríme **kolízia** a musíme to nejako rozumne vyriešiť.

Naša prvá verzia riešenia kolízií, bude pomocou tzv. **vedierok** (**bucket**): prvkami samotného poľa budú namiesto dvojíc (kľúč, hodnota) vedierka, t. j. postupnosti všetkých takých dvojíc (kľúč, hodnota), pre ktoré sa kľúč prepočítal na ten istý index prvku poľa (majú rovnaký zvyšok po delení kľúča veľkosťou poľa). **Vedierka** môžeme realizovať rôznymi spôsobmi (napr. spájaným zoznamom), my na to použijeme pythonovské pole (typ `list`) a pridávať nový kľúč budeme vždy na koniec tohto malého poľa (vedierka).

Nová verzia realizácie asociatívneho poľa pomocou vedierok:

```
# implementácia TestMap1

table = [[] for i in range(50)]     # nejaka predpokladana kapacita tabulky

def valueof(key):
    bucket = table[key % len(table)]
    for item in bucket:
        if item.key == key:
            return item.value
    raise KeyError

def add(key, value):
    bucket = table[key % len(table)]
    for item in bucket:
        if item.key == key:
            item.value = value
            return
    bucket.append(Item(key, value))

def delete(key):
    bucket = table[key % len(table)]
    for i in range(len(bucket)):
        if bucket[i].key == key:
            del bucket[i]
            return
    raise KeyError
```

Táto definícia sa veľmi podobá predchádzajúcej realizácii `TestMap` s tým rozdielom, že pole `table` obsahuje vedierka namiesto samotných prvkov `Item`. Každé vedierko je na začiatku prázdne pole, t. j. `[]`. Každá operácia najprv zistí, s ktorým vedierkom sa bude pracovať a potom túto operáciu urobí práve len s týmto jedným vedierkom. Každé vedierko je v teda malé asociatívne pole realizované ako `UnsortedMap`. Z toho potom vyplýva táto zložitosť operácií:

Tabuľka 4: Zložitosť operácií

operácie	unsorted	sorted	test	test1	
`valueof()`	**O(n)**	**O(log n)**	**O(1)**	**O(k)**	
`add()`	**O(n)**	**O(n)**	**O(1)**	**O(k)**	
`delete()`	**O(n)**	**O(n)**	**O(1)**	**O(k)**	

kde **k** v **O(k)** označuje zložitosť vedierka, keďže zložitosť operácií závisí od veľkosti poľa s vedierkom. Kým sú vedierka malé, **k** je malá konštanta, potom sa aj všetky operácie blížia k **O(1)**. Keď sa budú vedierka veľkosťou blížiť k **n**, zložitosť operácií bude **O(n)**. Takže by mohlo stačiť vhodne zvoliť veľkosť poľa tak, aby sa zložitosť blížila k **O(1)**.

5.1.3 Hašovacia funkcia

Verzia `TestMap1` má ešte ten podstatný nedostatok (obmedzenie), že kľúčom musí byť celé číslo. Ak by sme potrebovali napr. ukladať kľúče typu znakový reťazec, museli by sme na to zvoliť nejakú rozumnú stratégiu. Zvoľme takýto spôsob prekódovania reťazca na celé číslo:

```
def koduj(retazec):
    vysl = 0
    for znak in retazec:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        vysl = vysl + ord(znak)
    return vysl
```

Takáto funkcia naozaj prevedie ľubovoľný reťazec na celé číslo, ale má takúto nie najlepšiu vlastnosť: všetky tieto reťazce `'abc'`, `'acb'`, `'bac'`, `'bca'`, ... sa zakódujú rovnakým celým číslom 294. Zrejme tomuto algoritmu nezáleží na poradí znakov v reťazci. Vhodnejšie by bolo, keby funkcia počítala kód napr. takto:

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

Konštanta 100 sa častejšie nahradí nejakou mocninou 2, napr. 32. Dokonca malým vylepšením môžeme zabezpečiť, aby funkcia fungovala nielen pre reťazce ale aj pre ľubovoľný iný typ:

```
def koduj(kluc):
    vysl = 0
    for znak in str(kluc):
        vysl = 32 * vysl + ord(znak)
    return vysl
```

Vďaka takémuto kódovaniu, by sme mohli zabezpečiť, aby asociatívne pole fungovalo naozaj pre ľubovoľný typ kľúča. V našom prípade sa takejto funkcii hovorí **hašovacia** funkcia (hash function). Teda hašovacou funkciou sa nazýva taká funkcia, ktorá z hodnoty (skoro) ľubovoľného typu vyrobí kód, ktorý sa dá použiť na mapovanie kľúčov na indexy tabuľky. Štruktúra, ktorej kľúče mapujeme na indexy tabuľky pomocou hašovacej funkcie, sa nazýva **hašovacia tabuľka** (hash table). Upravme `TestMap1` na hašovaciu tabuľku:

```
# implementácia HashMap

table = [[] for i in range(50)]     # nejaka predpokladana kapacita tabulky

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

def valueof(key):
    ix = hash(key) % len(table)
    bucket = table[ix]
    for item in bucket:
        if item.key == key:
            return item.value
    raise KeyError

def add(key, value):
    ix = hash(key) % len(table)
    bucket = table[ix]
    for item in bucket:
        if item.key == key:
            item.value = value
            return
    bucket.append(Item(key, value))

def delete(key):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    ix = hash(key) % len(table)
    bucket = table[ix]
    for i in range(len(bucket)):
        if bucket[i].key == key:
            del bucket[i]
            return
    raise KeyError
```

Zhrňme:

- použili sme tu pomocnú funkciu `hash()`, ktorá ľubovoľný kľúč prevedie na celé číslo (najprv ju prevedie na reťazec a z neho potom postupne poskladá celé číslo)
- aby sme z tohto čísla dostali index do poľa, zistíme zvyšok po delení veľkosťou poľa - takto sa dostávame k príslušnému vedierku (bucket)
- zrejme musíme už 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

Teraz zvoľme takýto zjednodušený test:

```
pole = [(55,'a'),(42,'b'),(15,'c'),(60,'d'),(78,'e'),
        (35,'f'),(22,'g'),(10,'h'),(11,'i'),(8,'j'),
        (75,'k'),(32,'l')]
# kapacita nech je 10
for key, value in pole:
    add(key, value)
```

Do asociatívneho poľa sme postupne pridali týchto 12 dvojíc. Keďže kľúčom sú len celé čísla, zjednodušíme funkciu `hash()`, tak aby hašovacia funkcia priamo vrátila kľúč:

```
def hash(key):
    return key
```

Vďaka tomuto vieme aj ručne odkrokovať, ako sa vedierka postupne zapĺňajú (budeme počítať zvyšok po delení 10, teda nás zaujíma len posledná cifra kľúča). Po zaradení všetkých dvanástich dvojíc (kľúč, hodnota) dostávame takýto obsah poľa `table` aj s vedierkami:

```
for index, bucket in enumerate(table):
    print(index, bucket)
```

takýto výpis:

```
0 [60:d, 10:h]
1 [11:i]
2 [42:b, 22:g, 32:l]
3 []
4 []
5 [55:a, 15:c, 35:f, 75:k]
6 []
7 []
8 [78:e, 8:j]
9 []
```

Vidíme, že niektoré vedierka sú prázdne, kým v niektorých je viac prvkov, napr. piate vedierko obsahuje až 4 prvky.

Takémuto hašovaniu sa hovorí **uzavreté adresovanie** alebo **zreťazené hašovanie** (kolízie sú uzavreté v jednej reťazi dvojíc - niekedy sa vedierko realizuje nie poľom dvojíc ale spájaným zoznamom). Aby nás to viac poplietlo, **uzavreté adresovanie** označuje **otvorené hašovanie**. 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 veľkosťou tabuľky)

Toto uzavreté adresovanie môžeme trochu vylepšiť, napr. takto:

- na začiatku rezervujeme len **malé pole** vedierok a toto pole budeme v niektorých prípadoch nafukovať (podobne ako sme to robili pri dynamických poliach a operácii `append`)
- prázdne vedierko nebudeme uchovávať ako prázdne pole `[]` ale ako hodnotu `None` (urýchli to zmenu veľkosti `resize`)
- takže tabuľka `table` bude mať počiatočnú vyhradenú veľkosť napr. 11:

```
table = [None] * 11
```

- upravíme aj hašovaciu funkciu tak, že bude priamo počítať index do tabuľky, teda bude počítať zvyšok po delení veľkosťou tabuľky:

```
def hash(key):
    res = 0
    for ch in str(key):
        res = res * 32 + ord(ch)
    return res % len(table)
```

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

```
def valueof(key):
    bucket = table[hash(key)]
    if bucket is not None:
        for item in bucket:
            if item.key == key:
                return item.value
    raise KeyError
```

Nafukovať pole budeme vo funkcii `add()` vždy vtedy, keď počet prvkov, ktoré sú v ňom uložené, 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 `add()`) a pritom sa presiahne pomer zaplnenia

- napr.

```
def add(key, value):
    ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    if num > len(table) * 0.9:
        resize(len(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**:

```
# implementácia ChainHashMap

table = [None] * 11
num = 0                 # skutocny pocet prvkov

def hash(key):
    res = 0
    for ch in str(key):
        res = res * 32 + ord(ch)
    return res % len(table)

def valueof(key):
    bucket = table[hash(key)]
    if bucket is not None:
        for item in bucket:
            if item.key == key:
                return item.value
    raise KeyError

def add(key, value):
    ix = hash(key)
    bucket = table[ix]
    if bucket is not None:
        for item in bucket:
            if item.key == key:
                item.value = value
                return
    else:
        bucket = table[ix] = []
    bucket.append(Item(key, value))
    global num; num += 1
    if num > len(table) * 0.9:
        resize(len(table) * 2)

def delete(key):
    ix = hash(key)
    bucket = table[ix]
    if bucket is not None:
        for i in range(len(bucket)):
            if bucket[i].key == key:
                del bucket[i]
                global num; num -= 1
                if len(bucket) == 0:
                    table[ix] = None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                return
    raise KeyError

def resize(new_size):
    global table, num
    old_table = table
    table = [None] * new_size
    num = 0
    for bucket in old_table:
        if bucket is not None:
            for item in bucket:
                add(item.key, item.value)
```

Aj túto implementáciu môžeme otestovať podobne ako sme to testovali predtým.

5.2 Otvorené adresovanie

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 adresovanie**, tieto dva pojmy sú synonymá, lebo:
  - **otvorené adresovanie** (open addressing) označuje, že vypočítaná adresa (pomocou `hash()`) ešte nie je konečná, ale od tejto adresy sa bude hľadať konečná pozícia, bude to fungovať len v uzavretom hašovaní
  - **uzavreté hašovanie** (closed hashing) označuje, že vkladať sa bude iba niekam do tejto jednej tabuľky a nikam inam, dá sa to docieliť len otvoreným adresovaním
- na rozdiel od **uzavreté adresovanie**, kde aj tieto dva pojmy sú synonymá:
  - **uzavreté adresovanie** (closed addressing) označuje, že vypočítaná adresa jednoznačne určuje presnú pozíciu tabuľky (teda kde sa nachádza vedierko), ale toto bude fungovať len v otvorenom hašovaní, kde na „jednej pozícii“ (v jednom vedierku) sa nejako môže nachádzať viac prvkov
  - **otvorené hašovanie** (open hashing) označuje, že do tabuľky sa naozaj žiaden objekt nevkladá, vkladá sa do vedierok, ktoré sú niekde mimo samotnej tabuľky

Keďže pri kolízii hľadáme najbližšie voľné políčko v tabuľke, budeme tomu hovoriť **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 funkcie `find(key)`):

- podľa kľúča zistíme, na akom indexe by sa mal nachádzať (pomocná funkcia `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 adresovaní (uzavretom 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 rôzne hodnoty, napr. v Jave je to 75%, v Pythone 66%.

5.2.1 Vyhodenie prvku z tabuľky

Keď nájdeme prvok, ktorý chceme vyhodiť (vo funkcii `delete()`), 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 (vo funkcii `add()`) je tento `avail` kandidátom na pridanie novej dvojice.

Takže musíme opraviť pomocnú funkciu `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`)

```
avail = 'avail'

def find(key):
    ix = hash(key)
    av = None                # prvy volny
    while True:
        item = table[ix]
        if item is None:
            if av is None:
                av = ix
            return False, av
        if item == avail:
            if av is None:
                av = ix
        elif item.key == key:
            return True, ix
        ix = (ix + 1) % len(table)
```

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

Kompletný listing pre **otvorené adresovanie**:

```
# implementácia ProbeHashMap

table = [None] * 11     # nejaka predpokladana kapacita tabulky
avail = 'avail'
num = 0

def hash(key):
    res = 0
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    for ch in str(key):
        res = res * 32 + ord(ch)
    return res % len(table)

def find(key):
    '''vrati dvojicu
        (True, index) - ak najde kluc
        (False, prvy volny) - ak nenajde kluc
    '''
    ix = hash(key)
    av = None                   # prvy volny
    while True:
        item = table[ix]
        if item is None:
            if av is None:
                av = ix
            return False, av
        if item == avail:
            if av is None:
                av = ix
        elif item.key == key:
            return True, ix
        ix = (ix + 1) % len(table)

def valueof(key):
    found, ix = find(key)
    if found:
        return table[ix].value
    raise KeyError

def add(key, value):
    found, ix = find(key)
    if found:
        table[ix].value = value
        return
    global num; num += 1
    table[ix] = Item(key, value)
    if num > len(table) * 0.66:
        resize(len(table) * 2)

def delete(key):
    found, ix = find(key)
    if found:
        table[ix] = avail
        global num; num -= 1
    else:
        raise KeyError

def resize(new_size):
    global table, num
    old_table = table
    table = [None] * new_size
    num = 0
    for item in old_table:
        if item is not None and item is not avail:
            add(item.key, item.value)
```

Resize tabuľky robíme vždy vtedy, keď počet prvkov tabuľke presiahne 66% veľkosti celej tabuľky.

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

5.2.2 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 vysoko spoľahlivá.

Aby sa naša implementácia čo najviac priblížila k pythonovskému typu `dict`, mali by sme tomu prispôsobiť aj metódy, teda premenovať

- `valueof()` na `__getitem__()`
- `add()` na `__setitem__()`
- `delete()` na `__delitem__()`

Okrem toho môžeme využiť štandardnú pythonovskú funkciu `hash()`, ktorá za nás vypočíta hašovaciu hodnotu.

Ešte by sme do abstraktnej triedy mali dodefinovať všetky metódy tak ako to má `dict`, teda

```
clear()
copy()
fromkeys()
get()
pop()
popitem()
setdefault()
update()
```

aj magické metódy:

```
__contains__()
__repr__()
```

ale aj iterátory

```
items()
keys()
values()
```

Python okrem modulu `abc` (abstract base classes) ponúka aj modul `collections`, z ktorého môžeme využiť:

```
from collections import MutableMapping as MapBase
```

Už len dodefinujeme podtriedu `_Item` a máme komplet všetky metódy z `dict`.

5.3 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 adresovanie, napr. linear probing, pričom vyhodené prvky tiež označíme ako `_avail`

Abstraktný dátový typ prispôsobíme základným operáciám na množine (namiesto `valueof()`, `add()` a `delete()`):

```
from abc import ABC, abstractmethod

class SetBase(ABC):

    @abstractmethod
    def __contains__(self, key):
        '''zisti, ci je v mnozine'''
        pass

    @abstractmethod
    def add(self, key):
        '''ak nie je v mnozine, prida'''
        pass

    @abstractmethod
    def discard(self, key):
        '''ak je v mnozine, vyhodi'''
        pass
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    @abstractmethod
    def __len__(self):
        '''pocet prvkov v mnozine'''
        pass

    @abstractmethod
    def __iter__(self):
        '''prechadza vsetky prvky v mnozine'''
        pass
```

Samotná trieda pre množiny je veľmi zjednodušenou verziou `ProbeHashMap`:

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

    def __init__(self):
        self._table = [None] * 11         # capacity = 11
        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
            if item == self._avail:
                if av is None:
                    av = ix
            elif item == key:
                return True, ix
            ix = (ix + 1) % len(self._table)

    def __contains__(self, key):
        found, ix = self._find(key)
        return found

    def add(self, key):
        found, ix = self._find(key)
        if found:
            return
        self._num += 1
        self._table[ix] = key
        if self._num > len(self._table) * 0.66:
            self._resize(len(self._table) * 2)

    def discard(self, key):
        found, ix = self._find(key)
        if found:
            self._table[ix] = self._avail
            self._num -= 1
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

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

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

    def _resize(self, new_size):
        old_table = self._table
        self._table = [None] * new_size
        for item in old_table:
            if item is not None and item is not self._avail:
                found, ix = self._find(item)
                self._table[ix] = item
```

Hodnota `_avail` nemôže byť ani znakový reťazec ani žiaden iný typ, ktorý by mohol byť prvkom množiny, preto sme zvolili inštanciu `object()`. Táto sa vytvorí pri definovaní triedy ako „privátny" triedny atribút a je to určite unikátna hodnota, ktorá sa nikdy neobjaví ako pridávaný kľúč. Všimnite si, že naša definícia triedy nebude správne fungovať pre prvok `None`. Mohli by sme použiť takú istú fintu ako sme použili pre `_avail`.

Podobne ako môžeme využiť pythonovský modul `collections` pre abstraktnú triedu `MapBase` môžeme využiť aj

```
from collections import MutableSet as SetBase
```

Vďaka čomu sú zadefinované nielen všetky metódy, ale aj operácie (napr. zjednotenie, prienik, rozdiel).

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

5.3.1 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).

5.3.2 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.

5.4 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Hašovaciu tabuľku vytvárame v 7-prvkovom poli, pričom kolízie riešime pomocou vedierok, ktoré realizujeme jednorozmernými poľami (prvky pridávame na ich koniec). Kľúčmi sú celé čísla (asociované hodnoty si teraz nevšímame), pričom hašovacia funkcia počíta zvyšok po delení kľúča veľkosťou tabuľky.

- Postupne vložte do tabuľky tieto čísla

```
17, 36, 76, 76, 9, 52, 40, 24, 29, 26, 68, 7, 89, 76, 80, 59, 59, 2
```

zapíšte tieto kľúče do príslušných vedierok (na začiatku sú všetky prázdne):

```
0 ->
1 ->
2 ->
3 ->
4 ->
5 ->
6 ->
```

V tejto úlohe neriešte problém zaplnenia tabuľky pomocou `resize()`

2. Úlohu (1) riešte pre **otvorené adresovanie** (čo je vlastne **uzavreté hašovanie**), v ktorom hodnoty vkladáme do jednorozmerného poľa a kolízie riešime metódou **linear probing**. Na začiatku je vyhradené 11-prvkové pole a po vložení 8. prvku bude **load factor** väčší ako 0.66 preto sa zmení veľkosť poľa na dvojnásobok. Pritom bude treba všetky doterajšie prvky presypať do nového paľa, pričom zrejme dostanú nové pozície (ich `hash()` sa bude deliť 22). Keď už vložíme do tohto nového poľa 15. prvok, opäť sa prešvihne **load factor** a treba presypať pole na nové pozície.
   - realizujte postupné vkladanie týchto 18 prvkov aj s prípadnými **resize** tabuľky
   - urobte aspoň niekoľko krokov ručne, na zvyšok dát môžete použiť program
   - skontrolujte, akú konkrétnu hodnotu bude mať **load factor** po vložení 8. a 15. prvku
   - zistite, po koľkých ďalších vloženiach do tabuľky bude opäť **load factor** väčší ako 0.66 a mal by sa robiť **resize**
3. Otestujte algoritmus **ProbeHashMap** z prednášky na jednom z týchto súborov. Do tabuľky vkladajte len samotné slová - t.j. kľúčmi sú slová, hodnota môže byť `None`. Na záver vypíšte počet vkladaných slov, počet rôznych slov v tabuľke, pri akom zaplnení tabuľky sa robil **resize** a aká najdlhšia bola séria **kolízií** pri vkladaní jednej hodnoty.
   - slovník anglických slov: text1.txt
   - Dobšinského rozprávka: text2.txt
   - Sherlock Holmes: text3.txt
   - Huckleberry Finn: text4.txt
4. Vyrobte a otestujte triedu `ProbeHashMap`:
   - použite tento kód:

```
from map_base import MapBase

class ProbeHashMap(MapBase):
    _avail = 'avail'

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

    def _hash(self, key):
        res = 0
        for ch in str(key):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            res = res * 32 + ord(ch)
        return res % len(self._table)

    def _find(self, key):
        ...

    def valueof(self, key):
        ...

    def add(self, key, value):
        ...

    def delete(self, key):
        ...

    def __len__(self):
        ...

    def __iter__(self):
        ...

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

otestujte ho podobne ako v tretej úlohe

5. V definícii tried `MapBase` aj `ProbeHashMap` premenujte metódy:
   - `valueof()` nahradiť `__getitem__()`
   - `add()` nahradiť `__setitem__()`
   - `delete()` nahradiť `__delitem__()`

   Otestujte volania týchto nových metód (napr. namiesto `pole.add(kluc, hodnota)` by malo fungovat `pole[kluc] = hodnota`) na podobnom programe, ktorým ste testovali 3. a 4. úlohy.

6. Do triedy `MapBase` doprogramujte metódy, ktoré ale už nebudú abstraktné (používať budú iba metódy z `MapBase` a budú fungovať pre všetky neskôr odvodené triedy, napr. `UnsortedMap`, `SortedMap`, ..., `ProbeHashMap`, 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`)

7. Otvorte modul `_collectins_abc` (je súčasťou modulu `collections`) a nájdite, ako sú v ňom realizované metódy z úlohy (6)
   - hľadajte triedu `MutableMapping`

8. V definícii triedy `ProbeHashMap` nahraďte základnú triedu `MapBase` z prednášky triedou z modulu `collections`

   - vložte na začiatok

     ```
     from collections import MutableMapping as MapBase
     ```

   - z pôvodnej triedy `MapBase` bude treba niekam preniesť `_Item`

KAPITOLA 6

6. Vyhľadávacie stromy

Na minulej prednáške sme sa venovali asociatívnym poliam z pohľadu efektívnosti rôznych implementácií:

Tabuľka 1: Zložitosť operácií

operácie	unsorted	sorted	hash	
`valueof()`	**O(n)**	**O(log n)**	**O(1)**	
`add()`	**O(n)**	**O(n)**	**O(1)**	
`delete()`	**O(n)**	**O(n)**	**O(1)**	

Vidíme, že hašovacie tabuľky sú bezkonkurenčne najefektívnejšou realizáciou asociatívnych polí.

Lenže, čo by sme mali použiť, ak je požiadavkou mať stále k dispozícii aj utriedenú postupnosť kľúčov? Zatiaľ to vyzerá tak, že sa zmierime s realizáciou **SortedMap**: jedine tu utriedenú postupnosť kľúčov získame so zložitosťou **O(n)**. Pre obe zvyšné realizácie bude utriedaná postupnosť kľúčov stáť prinajlepšom **O(n log n)**.

6.1 Binárny vyhľadávací strom

V prvom ročníku sme sa zoznámili s binárnymi vyhľadávacími stromami (označujeme **BVS**), 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)**.

Skôr ako si pripomenieme vlastnosti **BVS**, pripravme si základnú triedu **BinTree** pre binárne stromy. Táto bude vychádzať z definícii triedy v 3. prednáške:

6.1.1 Implementácia BinTree

Súbor `bin_tree.py`:

```
import tkinter
import random
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
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:
```

(pokračuje na d'alšej strane)

(pokračovanie z predošlej strany)

```
            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):
    return 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 height(self, node=None):
    if node is None:
        node = self._root
    if node is None or node._left is None and node._right is None:
        return 0
    h1 = self.height(node._left) if node._left else 0
    h2 = self.height(node._right) if node._right else 0
    return 1 + max(h1, h2)

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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(300)
```

Môžeme sem pridať aj otestovanie:

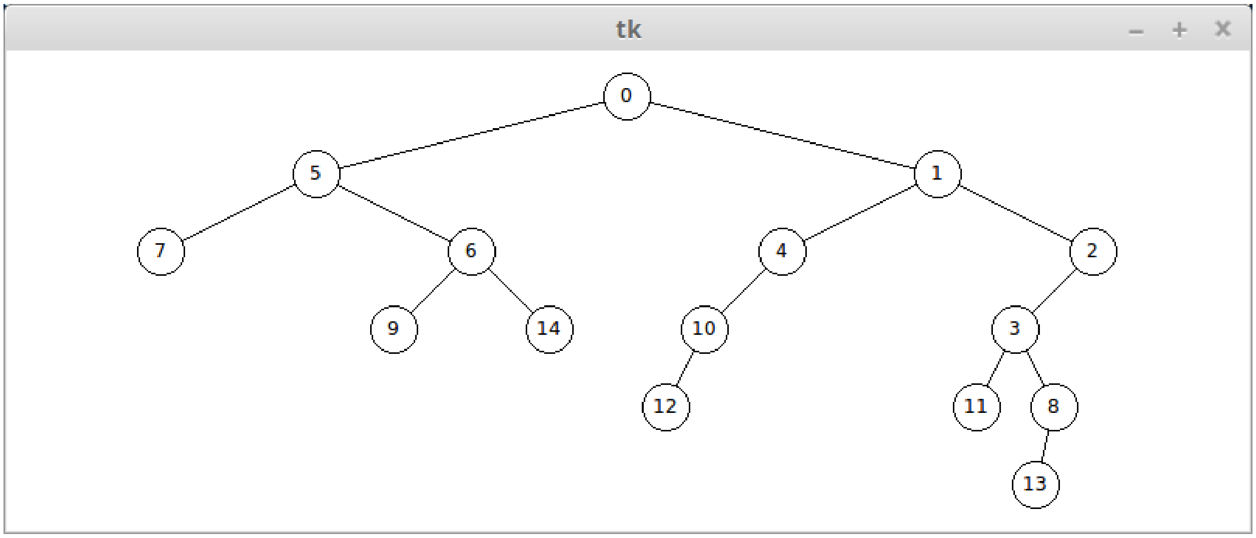
```
if __name__ == '__main__':
    strom = BinTree()
    for prvok in range(15):
        strom.add_random(prvok)
        strom.draw()
```

6.1.2 Pomocné metódy

Trieda `BinTree` obsahuje aj niekoľko pomocných metód, pričom niektoré z nich budú slúžiť len na ladenie:

- `add_root()`, `add_left()`, `add_right()` - pridajú nový vrchol na konkrétne miesto stromu
- `add_random()` - pridá nový vrchol náhodne na niektoré voľné miesto v strome
- `draw()` - nakreslí strom, ak sa zavolá znovu, pôvodný obrázok zmaže a nakreslí znovu
- `delete()` - vymaže zo stromu konkrétny zadaný vrchol
  - lenže vrchol vyhadzuje len v prípade, že je to **list** (nemá synov) alebo má **iba jedného syna**
  - ak je vrchol listom, vyhodenie je jednoduché: jeho otcovi nastavíme namiesto tohto syna `None`
  - ak má vyhadzovaný vrchol len jedného syna, tak jeho otcovi nastavíme namiesto neho jeho jediného syna (vnuk sa stáva synom)

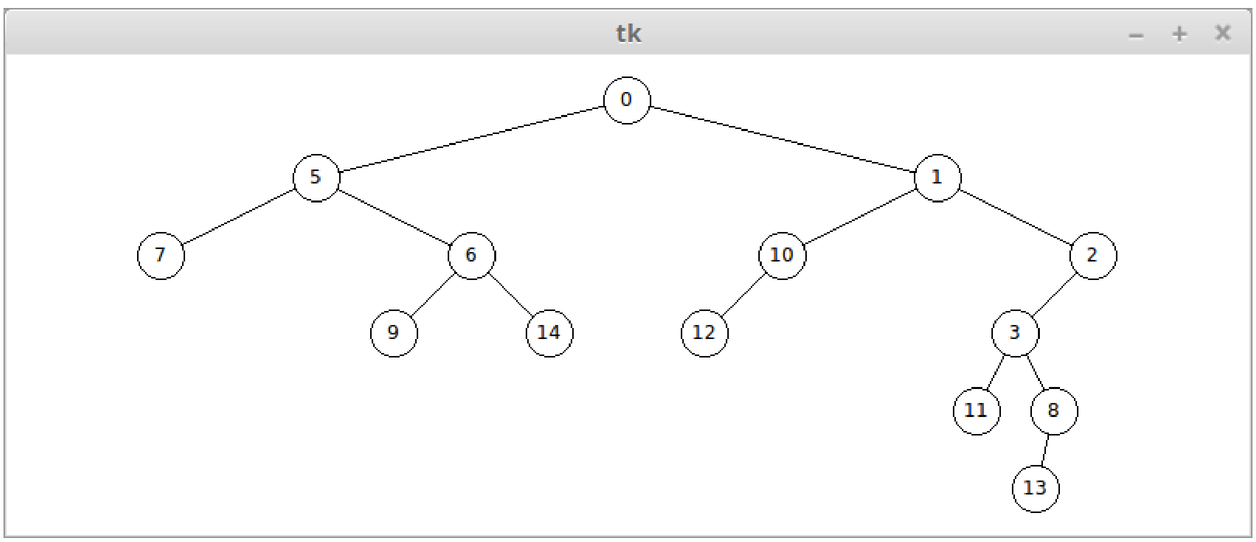
Po spustení testu, ktorý sme pridali do modulu `bin_tree.py`, dostaneme napr. takýto strom:



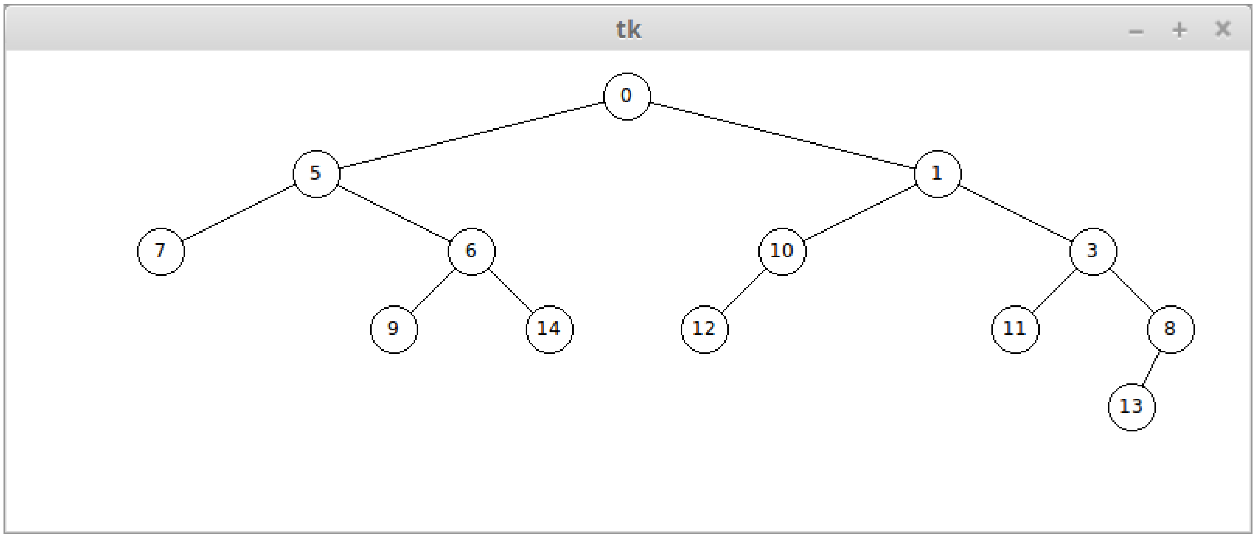
Ešte skontrolujeme metódy `height()` a `inorder()`:

```
>>> strom.height()
5
>>> list(strom.inorder())
[7, 5, 9, 6, 14, 0, 12, 10, 4, 1, 11, 3, 13, 8, 2]
```

Metóda `delete()` bude v tomto strome fungovať len pre listy alebo vrcholy, ktoré majú iba jedného syna. Napr. môžeme vyhodiť vrchol `4` (`strom.delete(strom._root._right._left)`): jeho otec `1` dostáva nového **ľavého** syna a to vrchol `10` (znovu zavoláme `strom.draw()`):



Podobne môžeme vyhodiť aj vrchol `2` (`strom.delete(strom._root._right._right)`), jeho otec `1` dostáva nového **pravého** syna, vrchol `3`:



V tomto konkrétnom strome nám metóda `delete()` nedovolí vyhodiť tieto vrcholy: `0`, `5`, `6`, `1`, `3`.

6.1.3 Vlastnosti BVS

Pripomeňme si, čo už vieme o binárnych vyhľadávacích stromoch z 1. ročníka:

- sú to **binárne stromy**
- pre koreň platí, že v jeho ľavom podstrome sú iba menšie hodnoty ako v koreni a v pravom iba väčšie
- táto vlastnosť platí pre všetky vrcholy stromu, t.j. každý má v ľavom podstrome iba menšie hodnoty a v pravom iba väčšie

Z prvého ročníka si pripomeňme hľadanie nejakej hodnoty v strome, resp. vkladanie novej hodnoty do stromu:

```
class BVS:                                  # kopia z 1. rocnika
    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

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

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

    def vloz(self, hodnota):
        if self.root is None:
            self.root = self.Vrchol(hodnota)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        else:
            vrch = self.root
            while vrch.data != hodnota:
                if vrch.data > hodnota:
                    if vrch.left is None:
                        vrch.left = self.Vrchol(hodnota)
                        break
                    vrch = vrch.left
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                        break
                    vrch = vrch.right
```

Algoritmus `hladaj()` (približne zodpovedá metóde `valueof()`):

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

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)**).

Algoritmus `vloz()` 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.

V prvom ročníku sme neriešili problém vyhadzovania vrcholu z BVS, budeme to robiť teraz.

Vyhodenie vrcholu z BVS

Už vieme, že metóda `delete()` triedy `BinTree` dokáže korektne vyhodiť vrchol z ľubovoľného binárneho stromu, pričom, ak bol tento strom BVS, táto vlastnosť sa tým zachová. Potrebujeme doriešiť situáciu, keď vyhadzovaný vrchol má oboch synov. Zišlo by sa nám také riešenie, ktoré nebude mať veľkú zložitosť a podľa možnosti len prehodí v strome zopár referencií. Použijeme takúto ideu algoritmu:

- vyhadzovaný vrchol (nazveme ho `node`) naozaj nevyhodíme, ale nahradíme ho iným obsahom tak, aby nepokazil zvyšok stromu
- z ľavého podstromu tohoto vrcholu (všetky majú hodnotu menšiu ako v `node`) vyberieme najväčšiu a tá sa stane novým obsahom `node` - všetky zvyšné vrcholy v ľavom podstrome majú menšiu hodnotu a v pravom podstrome `node` sú aj tak všetky hodnoty väčšie - preto je táto hodnota dobrým kandidátom, aby nahradila hodnotu v `node`
- maximálna hodnota v BVS sa hľadá veľmi jednoducho: stačí prechádzať po pravých synoch a keď narazíme na vrchol, ktorý už pravého syna nemá, toto je maximálny vrchol:

  ```
  max = node.left
  while max.right:
      max = max.right
  ```

- prekopírujeme obsah tohto vrcholu do `node` a samotný tento vrchol teraz môžeme jednoducho vyhodiť pomocou `delete()`, keďže tento určite nemá pravého syna (buď je to list alebo má iba jedného syna)

Teraz prepíšeme metódy tejto verzie BVS tak, aby spolupracovali s triedou `BinTree`. Pomocná metóda `_find()` nájde hľadaný vrchol, prípadne nájde vrchol, kam by sa hľadaný vrchol mal pripojiť. Tiež pridáme metódu na vyhadzovanie vrcholu:

6.1.4 Implemetácia BVS - 0. verzia

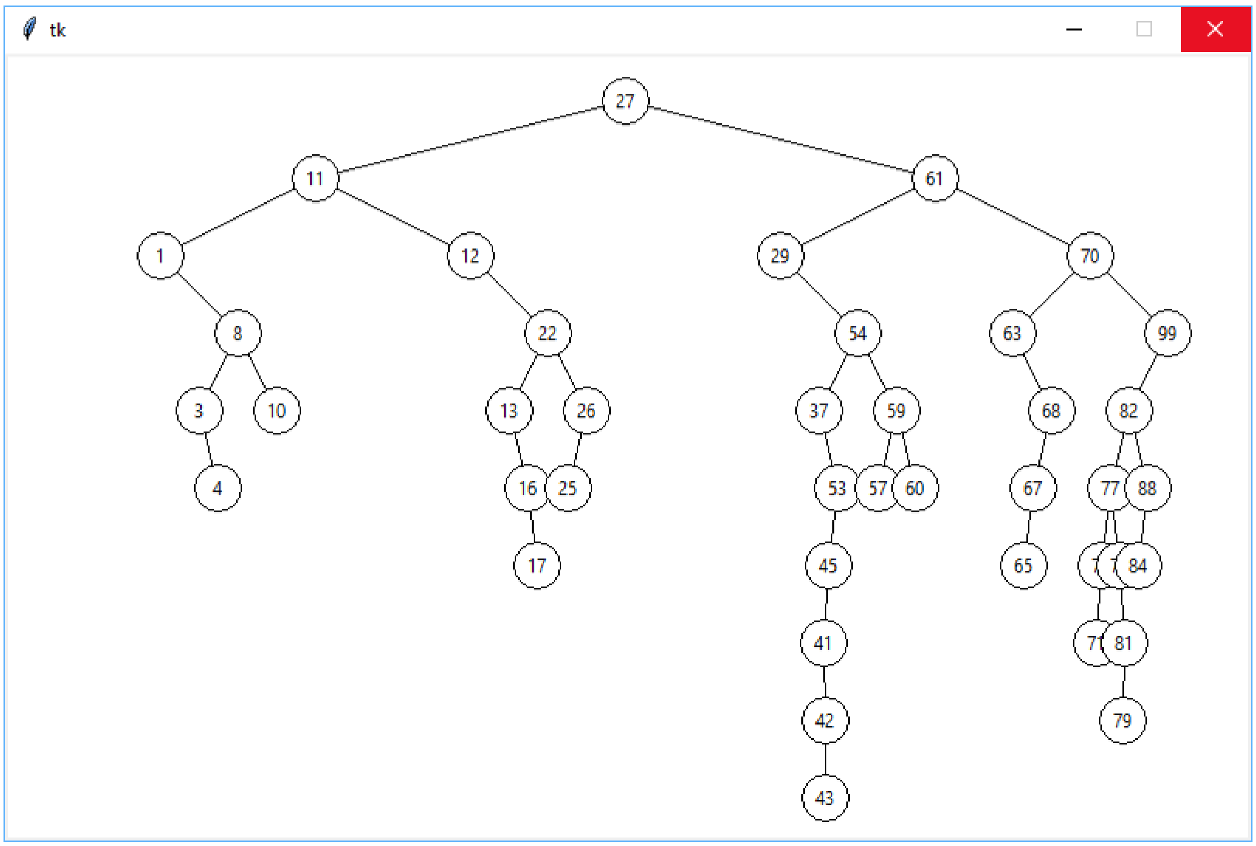
Táto verzia triedy `BVS` okrem `hladaj()` a `vloz()` realizuje aj vyhadzovanie vrcholy `vyhod()`.

```
from bin_tree import BinTree

class BVS(BinTree):                              # BVS - 0. verzia

    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 hladaj(self, key):
        if self.is_empty():
            return False
        node = self._find(self._root, key)
        return key == node._key

    def vloz(self, key, value=None):
        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 vyhod(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                       # hl'adá 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
```

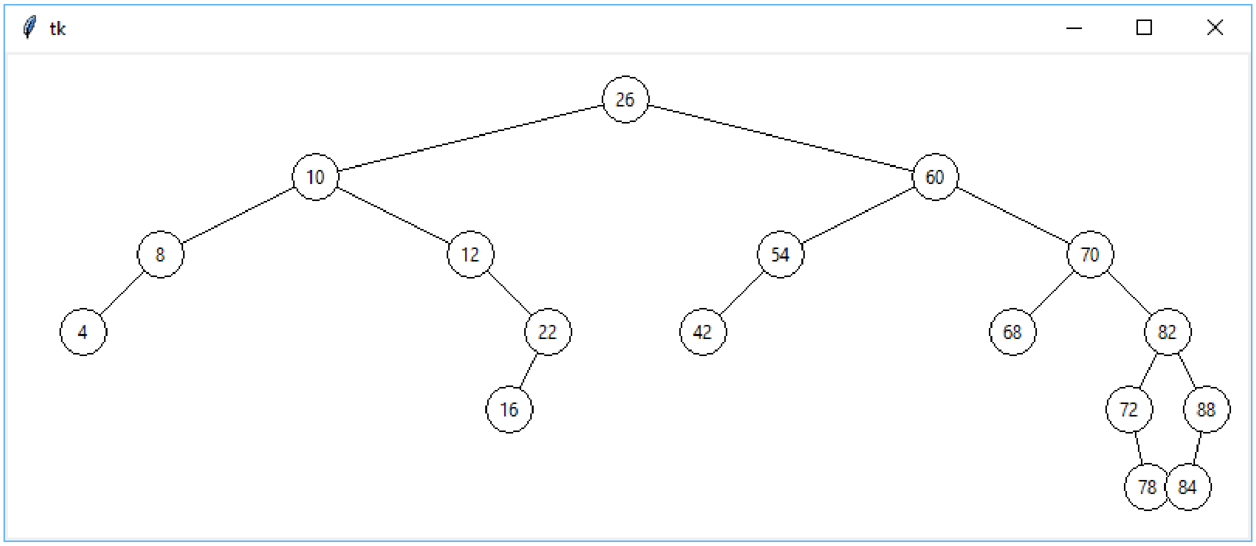
Môžeme sem pridať aj otestovanie:

```
if __name__ == '__main__':
    import random
    strom = BVS()
    pole = [random.randrange(100) for i in range(50)]
    for i in pole:
        strom.vloz(i)
        strom.draw()
    print('pocet =', len(strom), '  vyska =', strom.height())
    for i in range(1, 100, 2):
        if strom.hladaj(i):
            strom.vyhod(i)
        strom.draw()
    print('pocet =', len(strom), '  vyska =', strom.height())
    print('inorder:', *strom.inorder())
```

Tento test najprv vytvorí BVS z 50 náhodných hodnôt a potom postupne všetky nepárne vyhodí. Napr.



a teraz vyhodenie nepárnych:



Výpis môžeme dostať napr. takýto:

```
pocet = 41   vyska = 9
pocet = 17   vyska = 5
inorder: 4 8 10 12 16 22 26 42 54 60 68 70 72 78 82 84 88
```

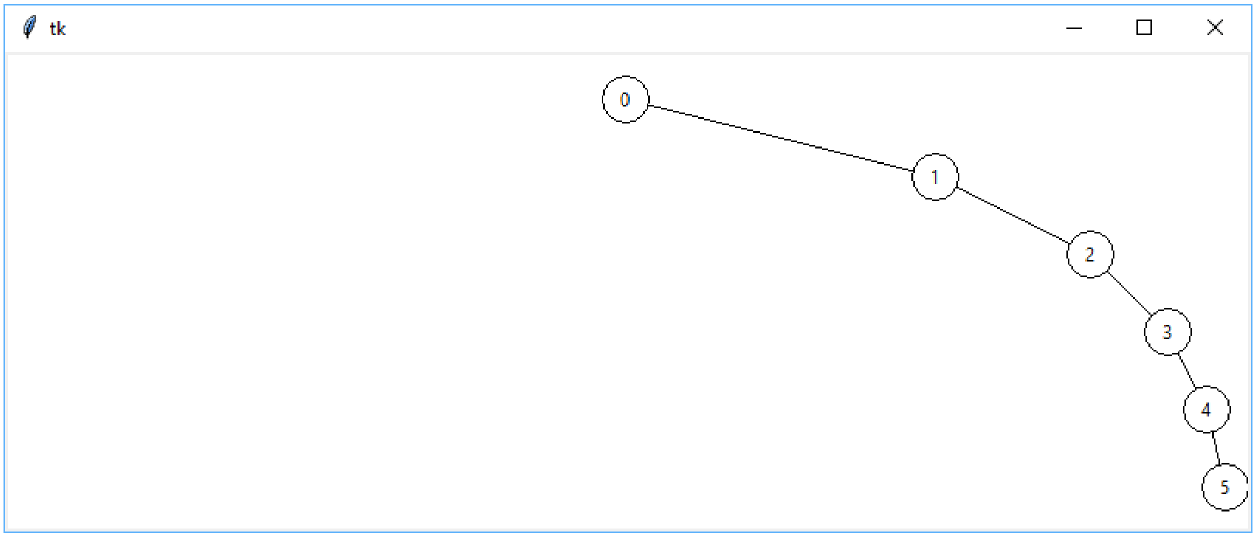
6.1.5 Zložitosť operácií

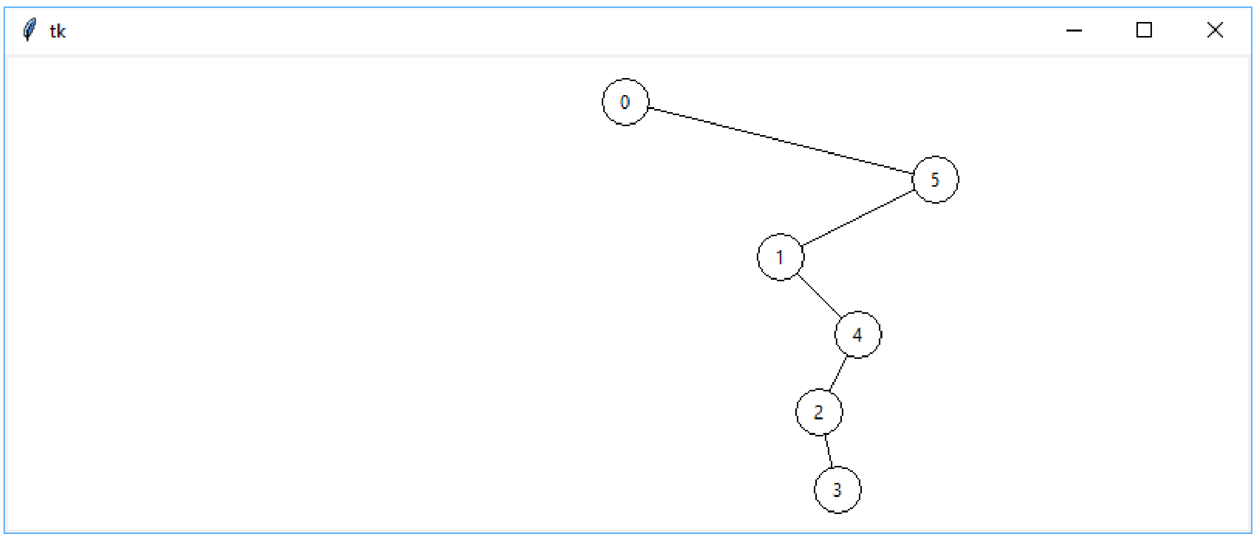
Máme teraz tri nové operácie `hladaj()`, `zmen()`, `vyhod()`, z ktorých by sa po malých úpravách dali vyrobiť tri operácie `valueof()`, `add()` a `delete()` (resp. `__getitem__()`, `__setitem__()`, `__delitem__()`). Zložitosť všetkých týchto troch operácií závisí od výšky stromu. Všetky tri totiž obsahujú cyklus, ktorý sa vnára len smerom k listom. Ak by sme predpokladali, že binárny strom má **dobrý tvar** a jeho výška je približne **log n**, máme výborné riešenie asociatívneho poľa s utriedenými kľúčmi (niečo ako `SortedMap`).

Tabuľka 2: Zložitosť operácií

operácie	unsorted	sorted	hash	dobré BVS	
`valueof()`	**O(n)**	**O(log n)**	**O(1)**	**O(log n)**	
`add()`	**O(n)**	**O(n)**	**O(1)**	**O(log n)**	
`delete()`	**O(n)**	**O(n)**	**O(1)**	**O(log n)**	

Lenže nás zaujíma nie dobrý prípad, ale najhorší. Najhoršími prípadmi sú napr.





Teda so to také stromy, ktoré majú jediný list. Sú to vlastne obyčajné jednosmerné spájané zoznamy, ktoré bude treba preliezať vždy celé pre hľadanie, nahrádzanie alebo vyhadzovanie ľubovoľnej hodnoty. Zložitosť všetkých troch operácií v takýchto prípadoch je **O(n)** a nie očakávané **O(log n)**. Výška týchto stromov nie je **log n** ale **n-1**

6.2 Vyvažovanie vyhľadávacích stromov

Tu by veľmi pomohol mechanizmus, ktorým by sme 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, môžete si pozrieť wikipediu). Vyvažovacie techniky vychádzajú z 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 `t2` každý nejakých výšok, tak tieto dva tvary stromov sú oba BVS s rovnakými hodnotami vrcholov len sú trochu inak poukladané:

```
     b                    a
    / \                  / \
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

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

Takýmto prerobením stromu z jedného tvaru na druhý niekedy 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 (spĺňa podmienku **AVL**) vtedy, keď sa výšky jeho podstromov líšia maximálne o 1. Pre každý vrchol vieme určiť jeho výšku (ako hlboko je v jeho celom podstrome: budeme predpokladať, že listy majú výšku 1, vrchol, z ktorého vychádza iba 1 alebo 2 listy, má výšku 2, . . .).

Potom *rovnováha* (balance) pre každý vrchol vypočítame napr. takto *balance* = *výška ľavého podstromu* - *výška pravého podstromu*. V každom vrchole stromu si budeme pamätať jeho výšku (atribút `_height`), aby sme mohli rýchlo vypočítať jeho rovnováhu. Niektoré iné prístupy si v každom vrchole pamätajú priamo tento *balance*.

6.2.1 Implemetácia BVS - s prípravou na vyvažovanie

Všimnite si, že do súboru `bvs.py`:

```python
from bin_tree import BinTree

class BVS(BinTree):                               # BVS - s prípravou na vyvažovanie

    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 ľavého podstromu
        elif key > node._key and node._right:
            return self._find(node._right, key)   # vnorenie do pravého podstromu
        return node

    def hladaj(self, key):
        if self.is_empty():
            return False
        node = self._find(self._root, key)
        return key == node._key

    def vloz(self, key, value=None):
        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 vyhod(self, key):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        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                      # hl'adá 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
```

sme na niektoré miesta vložili volanie metódy `self.rebalance(...)`, pričom samotná metóda je zatiaľ prázdna:

```
class BVS(BinTree):
    ...
    def rebalance(self, node):
        pass
```

Sú to všetky tie miesta, ktoré môžu ovplybniť tvar stromu a teda tu predpokladáme, že bude treba robiť vyvažovanie.

6.2.2 Implementácia AVL

Zapíšme súbor `avl.py`:

```
from bvs import BVS

class AVL(BVS):

    #-------------------------- vnorena trieda Node --------------------------
    class _Node(BVS._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())
```

(pokračuje na d'alšej strane)

(pokračovanie z predošlej strany)

```
    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
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            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
```

Táto nová trieda je odvodená od triedy `BVS`, pričom prekýva definíciu `_Node` (pridáva sem nový atribút `_height`) a prekýba aj metódu `rebalance()`, vďaka čomu vkladanie a vyhadzovanie vrcholov do vyhľadávacieho stromu do bude automaticky vyvažovať.

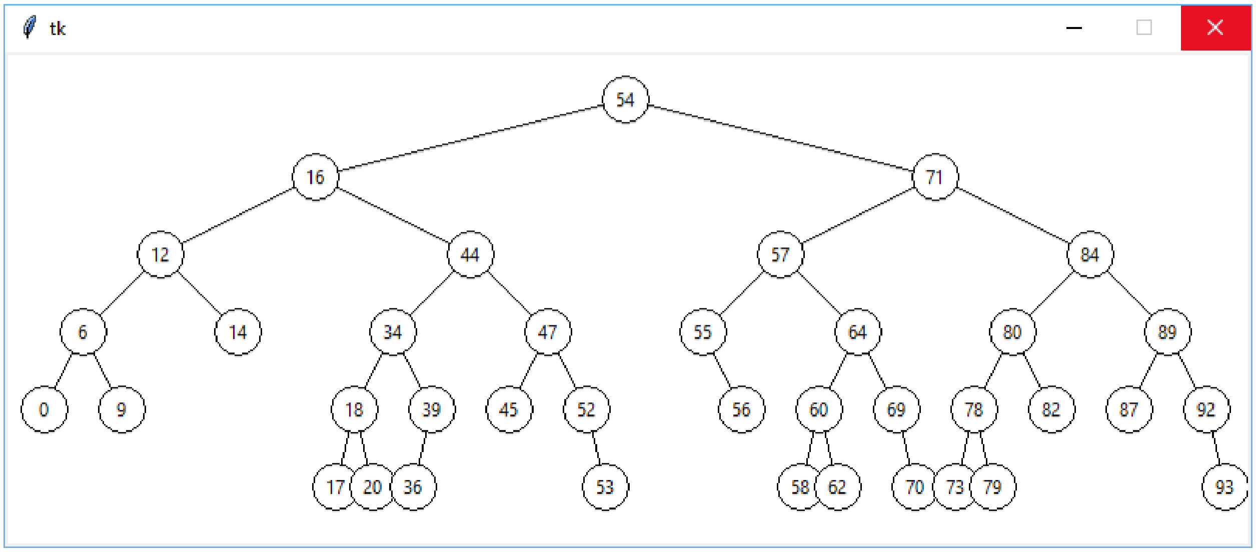
Odsledujte napr. takéto otestovanie:

```
if __name__ == '__main__':
    import random
    strom = AVL()
```

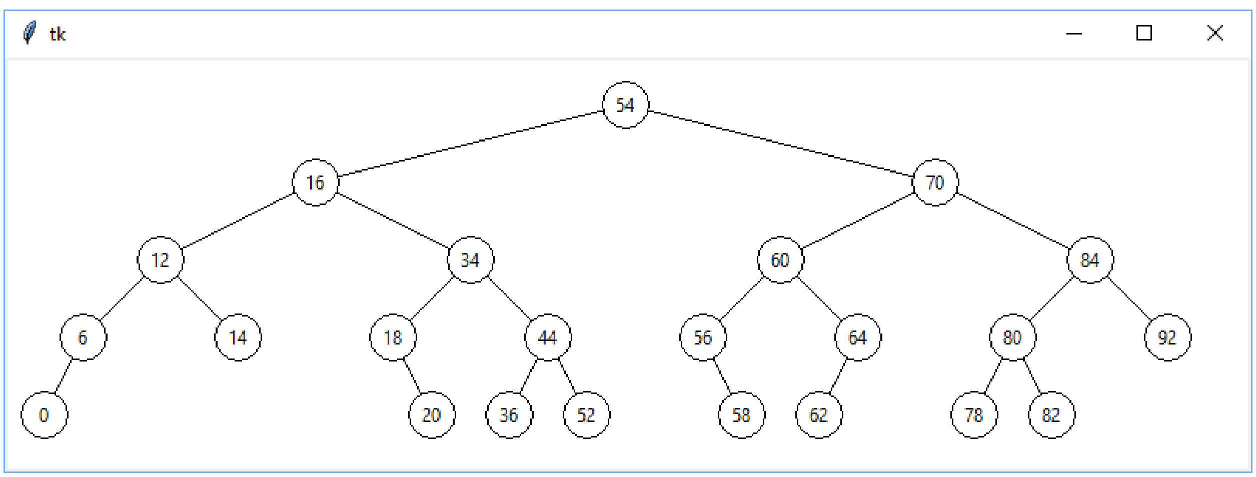
(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    pole = [random.randrange(100) for i in range(50)]
    for i in pole:
        strom.vloz(i)
        strom.draw()
    print('pocet =', len(strom), '  vyska =', strom.height())
    for i in range(1, 100, 2):
        if strom.hladaj(i):
            strom.vyhod(i)
        strom.draw()
    print('pocet =', len(strom), '  vyska =', strom.height())
```



a teraz vyhodenie nepárnych:



Mohli ste spozorovať, že v každom okamihu bol vyhľadávací strom pekne vyvážený (rozdiel výšok podstromov pre každý vrchol bol maximálne 1). Tento test vypísal:

```
pocet = 38   vyska = 5
pocet = 23   vyska = 4
```

6.2.3 Asociatívne pole

Zatiaľ sme vytvárali dátové štruktúry **BVS**, resp. **AVL**, ktoré realizujú príslušné algoritmy, ale zatiaľ to nie je asociatívne pole. Potrebovali by sme premenovať a trochu aj prerobiť niektoré metódy, prípadne dodefinovať všetky abstraktné aj neabstraktné metódy, ktoré potrebujeme, aby to bolo naozaj asociatívne pole (aby sa to podobalo na pythonovský `dict`).

Teraz využijeme už zadefinovaný abstraktný dátový typ v štandardnom module `collections`. V tomto module má táto abstraktná trieda meno `MutableMapping` preto si tento názov pri importe upravíme:

```
from collections import MutableMapping as MapBase
```

Namiesto triedy `BVS` vytvoríme triedu `TreeMap`, ktorá bude odvodená od `MapBase` (aby to bolo asociatívne pole). Ale zároveň musí byť odvodená aj od `BinTree`, keďže tu sa nachádzajú všetky dôležité metódy, ktoré potrebujeme pre fungovanie binárneho stromu. Tomuto hovoríme **viacnásobná dedičnosť** tried.

6.2.4 Implementácia TreeMap a AVLTreeMap

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

```
from bin_tree import BinTree
from collections import MutableMapping as MapBase

class TreeMap(BinTree, 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
        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
                self.rebalance(node)
            else:
                self.add_right(node, key, value)  # zdedené z BinTree
                self.rebalance(node)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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                          # hl'adá 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
```

A trieda, ktorá bude realizovať aj AVL vyvažovanie:

```
from tree_map import TreeMap

class AVLTreeMap(TreeMap):

    #-------------------------- vnorena trieda Node --------------------------
    class _Node(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():
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            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)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            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
```

Príslušný test potom môže vyzerať napr. takto:

```
if __name__ == '__main__':
    import random
    dic1 = AVLTreeMap()
    pole = [random.randrange(10000) for i in range(5000)]
    dic2 = {}
    for i in pole:
        dic1[i] = dic1.get(i, 0) + 1
        dic2[i] = dic2.get(i, 0) + 1
    print(len(dic1), dict(dic1.items())==dic2, list(dic1)==sorted(dic2), dic1.
↪height())
    for i in range(1, 10000, 2):
        try:
            del dic1[i]
```

(pokračuje na d'alšej strane)

(pokračovanie z predošlej strany)

```
        except KeyError:
            pass
        try:
            del dic2[i]
        except KeyError:
            pass
    print(len(dic1), dict(dic1.items())==dic2, list(dic1)==sorted(dic2), dic1.
↪height())
```

Všimnite si, že teraz pracujeme s binárnym vyhľadávacím stromom ako s asociatívnym poľom.

6.2.5 Zložitosť operácií

Zapíšme zložitosť operácií aj pre dnešné nové dve dátové štruktúry:

Tabuľka 3: Zložitosť operácií

operácie	unsorted	sorted	hash	BVS	AVL	
`valueof()`	**O(n)**	**O(log n)**	**O(1)**	**O(h)**	**O(log n)**	
`add()`	**O(n)**	**O(n)**	**O(1)**	**O(h)**	**O(log n)**	
`delete()`	**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)**.

6.3 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Ručne vyrobte **BVS** (bez vyvažovania) z čísel (kľúčov) v tomto poradí: 30, 40, 24, 58, 48, 26, 11, 13
   - ku každému vrcholu pripíšte jeho výšku (`self._height`) aj balance (výška ľavého podstromu - výška pravého podstromu)
   - navrhnite, ktoré dva vrcholy v tomto strome by sa mohli „zrotovať", aby sa strom stal **AVL** - ručne tento strom prekreslite a prepočítajte balance
2. **Úloha z minuloročného testu**: Nakreslite všetky binárne vyhľadávacie stromy, v ktorých sú uložené kľúče `{1,2,3,4}` a vyznačte tie z nich, ktoré spĺňajú podmienku **AVL** stromov.
   - táto úloha sa potom upravená objavila aj ako skúškový príklad
3. Spojazdnite triedu `TreeMap` z prednášky (asociatívne pole realizované pomocou **BVS**)
   - otestujte frekvenčnú tabuľku (z prednášky) ale zatiaľ bez AVL
   - otestujte na 30 náhodných údajoch a pritom priebežne vykresľujte celý strom
   - trieda `TreeMap` je odvodená z dvoch tried: `BinTree` a `MapBase`; mali by ste rozumieť, na čo tu slúžia obe tieto základné triedy a čo by sa napr. stalo, keby sme `TreeMap` odvodili len z `BinTree`, teda by sme zapísali:

```
class TreeMap(BinTree):
    ...
```

- prestane teraz niečo fungovať?

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

   - v niektorých situáciách to spadne na preplnení rekurzie - vygenerujte také testovacie dáta, aby táto funkcia naozaj spadla na rekurzii
   - prepíšte túto metódu bez rekurzie a otestujte jej funkčnosť na dátach, na ktorých to v predchádzajúcom teste spadlo

5. Do triedy `TreeMap` dopíšte pomocné metódy:

   - `balance(node)` - pre daný vrchol vráti jeho rovnováhu, teda rozdiel výšok l'avého a pravého podstromu metód
   - `is_bvs()` - zistí, či daný strom spĺňa podmienky pre binárny vyhl'adávací strom (či je korektný)
   - `is_avl()` - zistí, či daný strom spĺňa AVL podmienky
   - upravte metódu `draw()` tak, aby namiesto hodnoty vo vrchole vypisovala jeho balance

6. Spojazdnite `AVLTreeMap` z prednášky:

   - sledujte (vykresl'ujte), ako sa priebežne vyvažuje celý strom, keď sa do neho pridávajú, resp. vyhadzujú vrcholy
   - otestujte na vel'kých údajoch (napr. frekvenčná tabul'ka z prednášky)
   - skontrolujte, či je AVL-strom skonštruovaný korektne - využite metódu `is_avl()`

7. Do `BinTree` dopíšte dve metódy `next(p)` a `prev(p)`, pre ktoré je parametrom referencia na nejaký vrchol (typu `_Node`):

   - metóda `next(None)` vráti najl'avejší 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 kompletná 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 (kl'účov)

KAPITOLA 7

7. Triedenia

S triedeniami sa stretáme už od prvého ročníka. Pri triedeniach si všímame rôzne kritériá, na základe ktorých môžeme usudzovať o ich kvalitách:

- **počet porovnaní** - všetky doterajšie naše triedenia boli založené na porovnávaní prvkov
  - zložitosť týchto triedení je minimálne **O(n log n)** a maximálne **O(n**2)**
  - exitujú triedenia, ktoré nefungujú na princípe vzájomného porovnávania prvkov a ich zložitosť je **O(n)**
- **počet výmen** - mnohé triedenia menia poradia prvkov vzájomným vymieňaním prvkov, v niektorých situáciách môže byť toto dosť dôležité kritérium
- **použitie pomocnej pamäte** - koľko ďalšej pamäti je potrebný k danému algoritmu triedenia
  - niektoré algoritmy triedia priamo samotné pole, tzv. **in-place**, často potrebujú **O(1)** alebo **O(log n)** pomocnej pamäte
  - triednia, ktoré nie sú in-place, vytvárajú utriedené nové pole a ich zložitosť je teda min. **O(n)**
- **rekurzia** - už poznáme nerekurzívne triedenia (napr. **bubble-sort**), rekurzívne (napr. **quick-sort**) a zoznámime sa s takým, ktroé je rekurzívne sj nerekurzívne (**merge-sort**)
- **stabilita** - vlastnosť stabilita označuje, že pre dva prvky poľa, ktoré majú rovnaký kľúč (`pole[i].kluc==pole[j].kluc`), ak bolo `i<j` tak aj po utriedení bude prevý prvok pred druhým (relatívna poloha rovnakých prvkov ostane po triedení zachovaná)
- vnútorné/vonkajšie triedenia - či sa samotné triedenie aj pomocná pamúť nachádza v operačnej pamäti, alebo používame externú pamäť (napr. disk) - s týmto sa v tomto semestri nebudeme zaoberať

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

bubble_sort

Veľmi pomalé neefektívne **in-place** triedenie - použiteľné len pre malé polia.

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**2)**:

```
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]
```

Ak vnútorný cyklus (jeden prechod triedenia) neurobí ani jednu výmenu, pole je už utriedené a môžeme okamžite ukončiť:

```
def bubble_sort(pole):
    for i in range(1, len(pole)):
        bola_vymena = False
        for j in range(len(pole)-i):
            if pole[j] > pole[j+1]:
                pole[j], pole[j+1] = pole[j+1], pole[j]
                bola_vymena = True
        if not bola_vymena:
            return
```

Vďaka tejto malej zmene najlepší prípad (pole už bolo utriedené), bude zložitosť len **O(n)** - toto je jedna z mála výhod **bubble-sort** - vie rozpoznať, že je pole utriedené. Počet výmen je tiež **O(n**2)**.

Pamäťová zložitosť je **O(1)** - využíva len konštantný počet premenných, ich počet (veľkosť) nezávisí od veľkosti triedeného poľa.

selection_sort

in-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)**, počet výmen je len **O(n)**, vhodný v prípadoch, keď sa pracuje s dátami, kde je výmena dvoch prvkov drahá operácia. Považuje sa za efektívnejší ako **bubble-sort**.

Stretli sme sa s ním aj pri prioritných frontoch, keď sme pomocou nich triedili. Realizácia `UnsortedPriorityQueue` triedila rovnako ako **selection-sort**.

insert_sort

in-place triedenie 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, . . . Počet porovnaní je **O(n**2)**. považuje sa za efektívnejší ako **selection-sort**. Jeho výhodou je to, že pre skoro utriedené polia má zložitodť skoro **o(n)**.

Stretli sme sa s ním aj pri prioritných frontoch, keď sme pomocou nich triedili. Realizácia `SortedPriorityQueue` triedila rovnako ako **insert-sort**.

heap_sort

Stretli sme sa s ním pri `HeapPriorityQueue`:

1. krok - `heapify` - pole sa prerobí na haldu - táto operácia má zložitosť **O(n)**
2. krok - **n**-krát odoberieme minimálny prvok `remove_min`, ktorý zrejme uprace haldu operáciou `hesp_down` - keďže `heap_down` má zložitosť `O(log n)`, `n`-krát to znamená zložitosť **O(n log n)**

Celková zložitosť algoritmu je **O(n log n)**, je to **in-place** sort, ktorý **nie je stabilný**. Pamäťová zložitosť je **O(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:

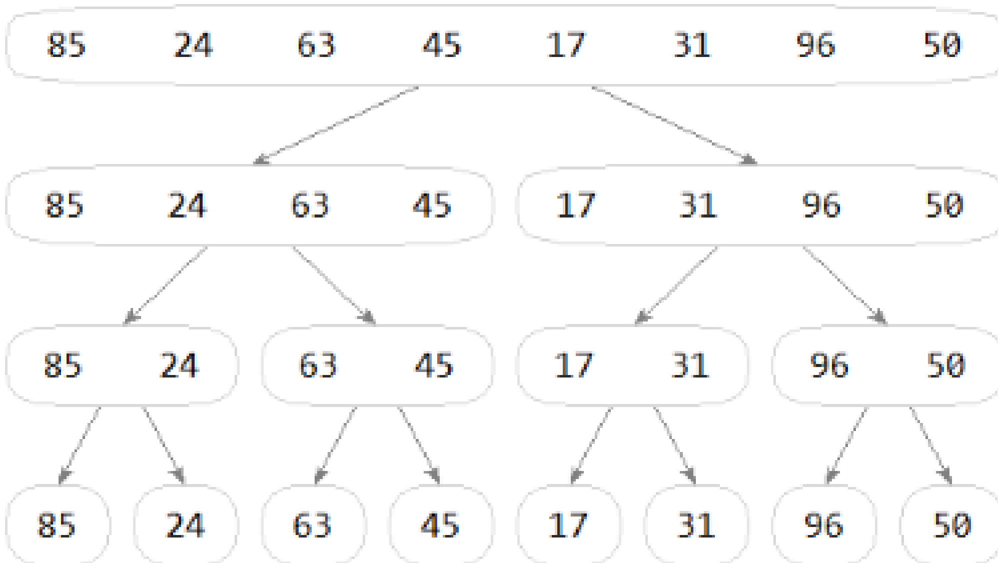
1. ak má úloha nejakú aspoň minimálnu veľkosť, rozdeľ ju na niekoľko disjunktných častí, inak vyrieš triviálny prípad (bez rekurzie)
2. rekurzívne vyrieš úlohu pre každú z častí
3. spoj tieto riešenia do výsledného celku

7.1 Merge sort

Rozdeľuj a panuj:

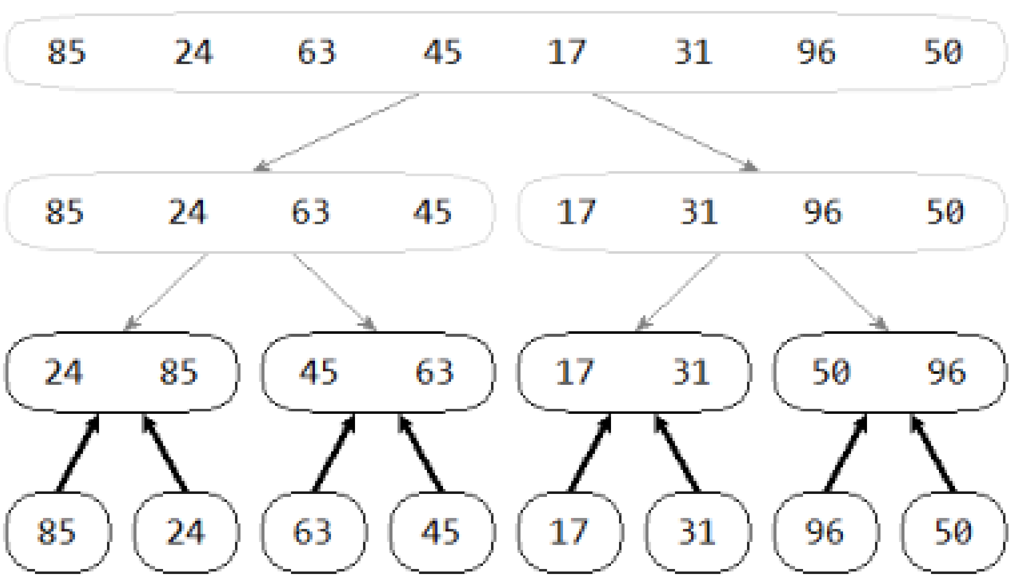
1. ak má pole aspoň 2 prvky, rozdeľ ho na dve rovnaké časti (napr. veľkosti `n//2` a `n-n//2`)
2. rekurzívne zavolaj triedenie zvlášť pre každú z častí - dostaneme 2 utriedené časti celého poľa
3. zlúč (merge) obe utriedené časti do výsledného poľa

Ideu rozdeľovania poľa na polovice vidíme (tzv. merge_sort strom):

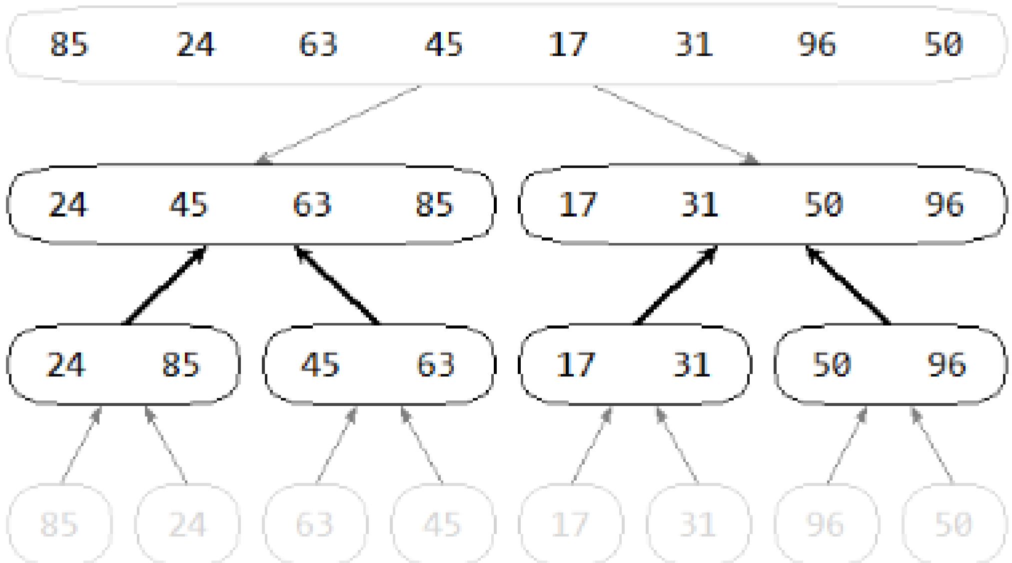


Toto bola 1. fáza algoritmu, keď sa pole rozdelilo na dve polovice a pre každú polovicu sa spustilo rekurzívne volanie, t.j. opäť rozdelenie na polovice. Toto pokračovalo dovtedy, kým bolo čo deliť, teda 1-prvkové pole sa už ďalej nedelilo.

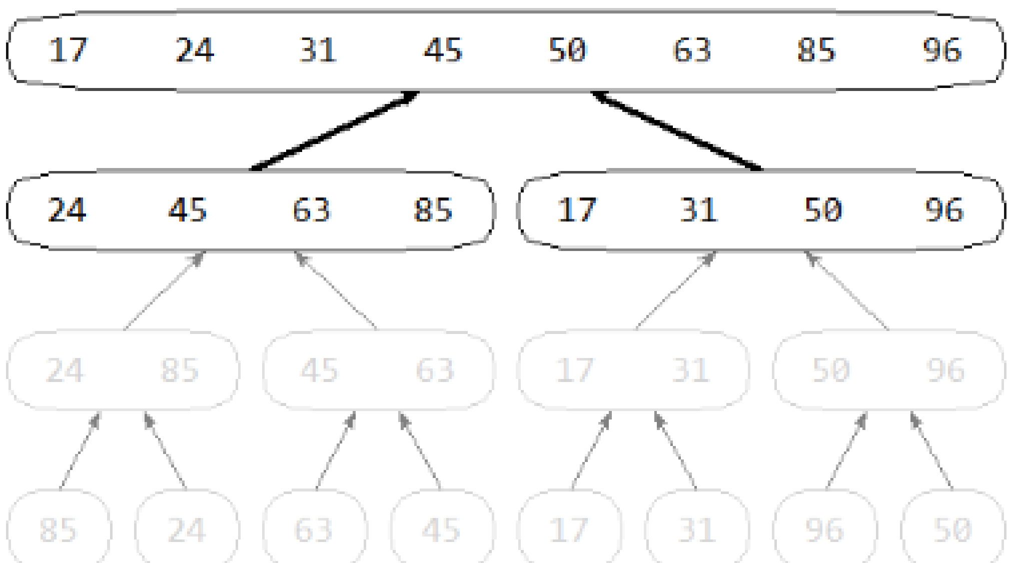
Teraz prichádza fáza návratu z rekurzie, keď sa majú dve polovice poľa zlúčiť (tzv. merge) do utriedeného celku. Lenže vďaka rekurzii sa najprv budú zlučovať dve najvnorenejšie polovice, t.j. 1-prvkové polia: z nich sa vytvoria utriedené dvojprvkové:



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:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    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.

7.1.1 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
```

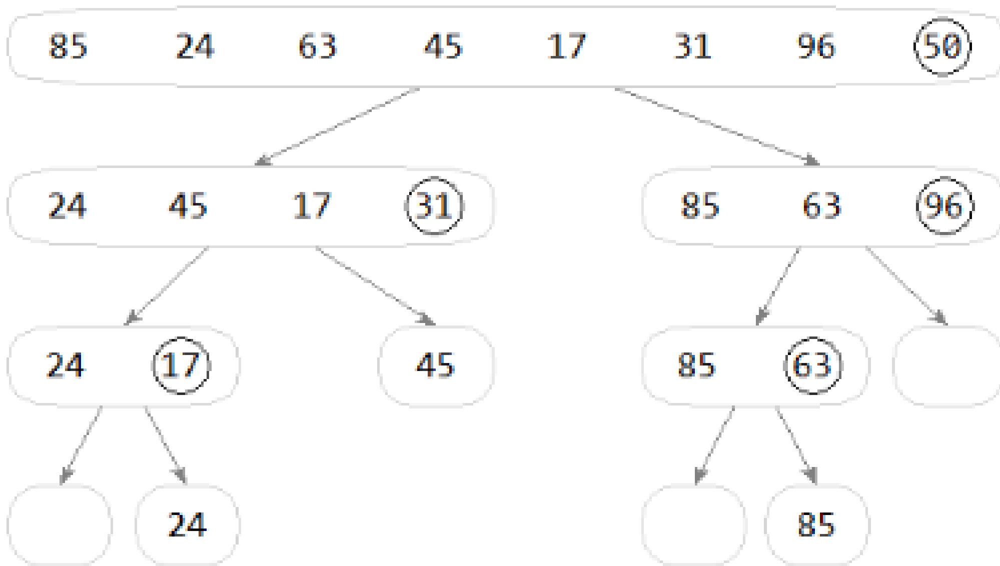
Podobnú ideu by sme vedeli zapísať aj pomocou spájaných zoznamov.

7.2 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

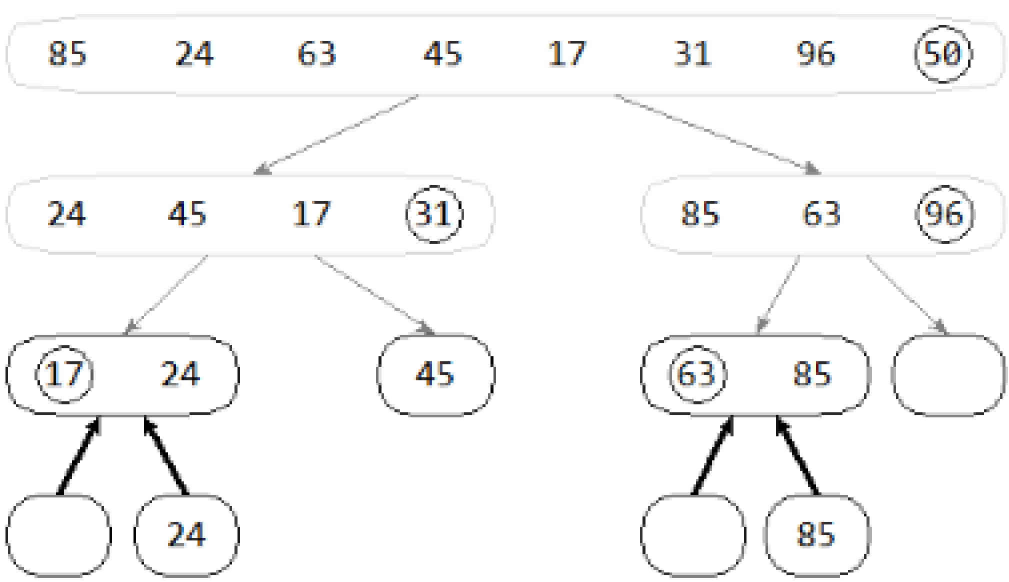
Ideu rozdeľovania poľa na dve časti podľa pivota vidíme:



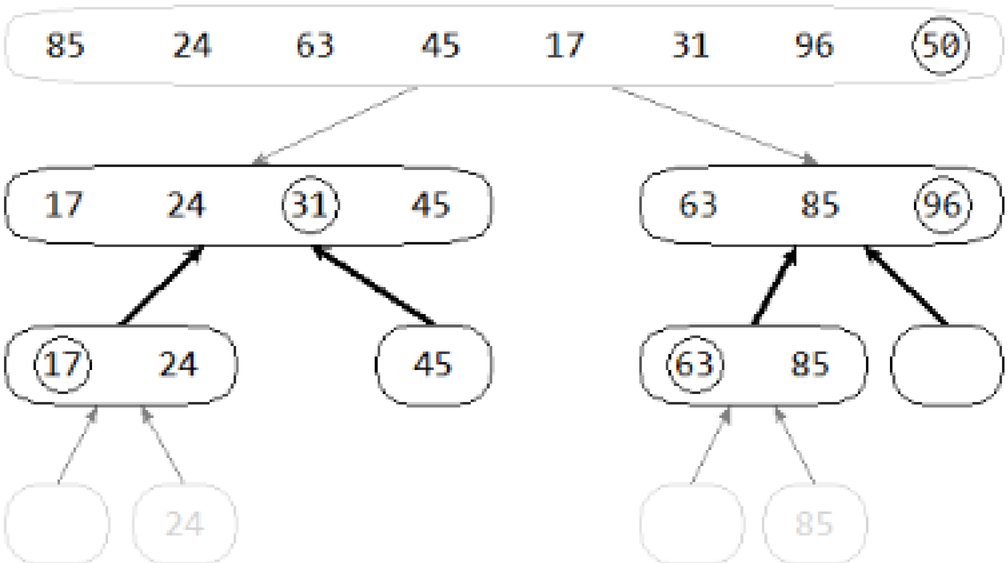
Pivota sme tu vybrali ako posledný prvok (je zakrúžkovaný). Rozdelené dve časti sú už bez tohto pivota (ten sa neskôr objaví vo výsledku v strede medzi nimi).

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.

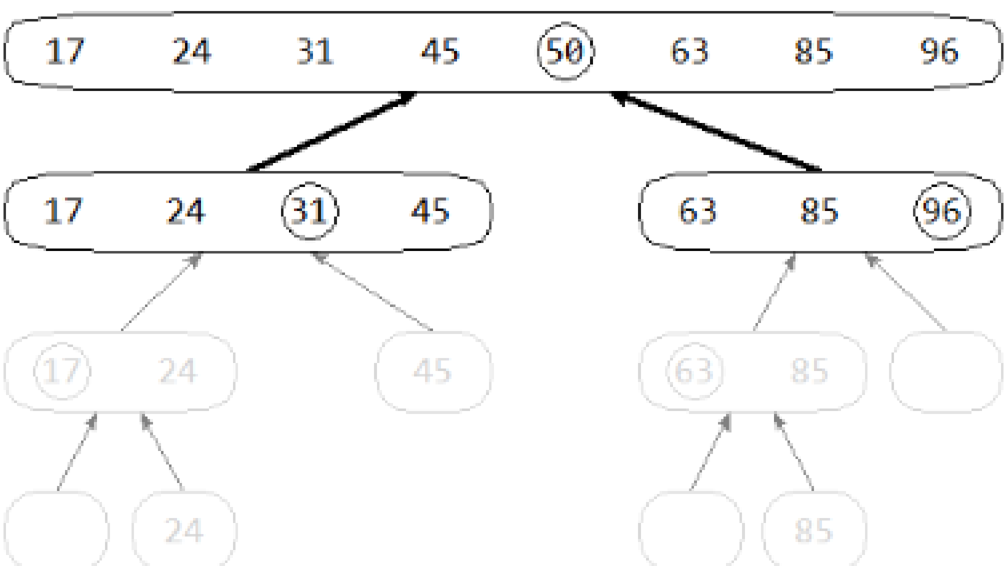
Teraz prichádza fáza návratu z rekurzie, keď sa majú obe časti spojiť do utriedeného celku (medzi ne sa zaradí pivot, ktorý sa v rekurzii netriedil). Vďaka rekurzii sa najprv budú spájať dvojice najvnorenejších častí:



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:

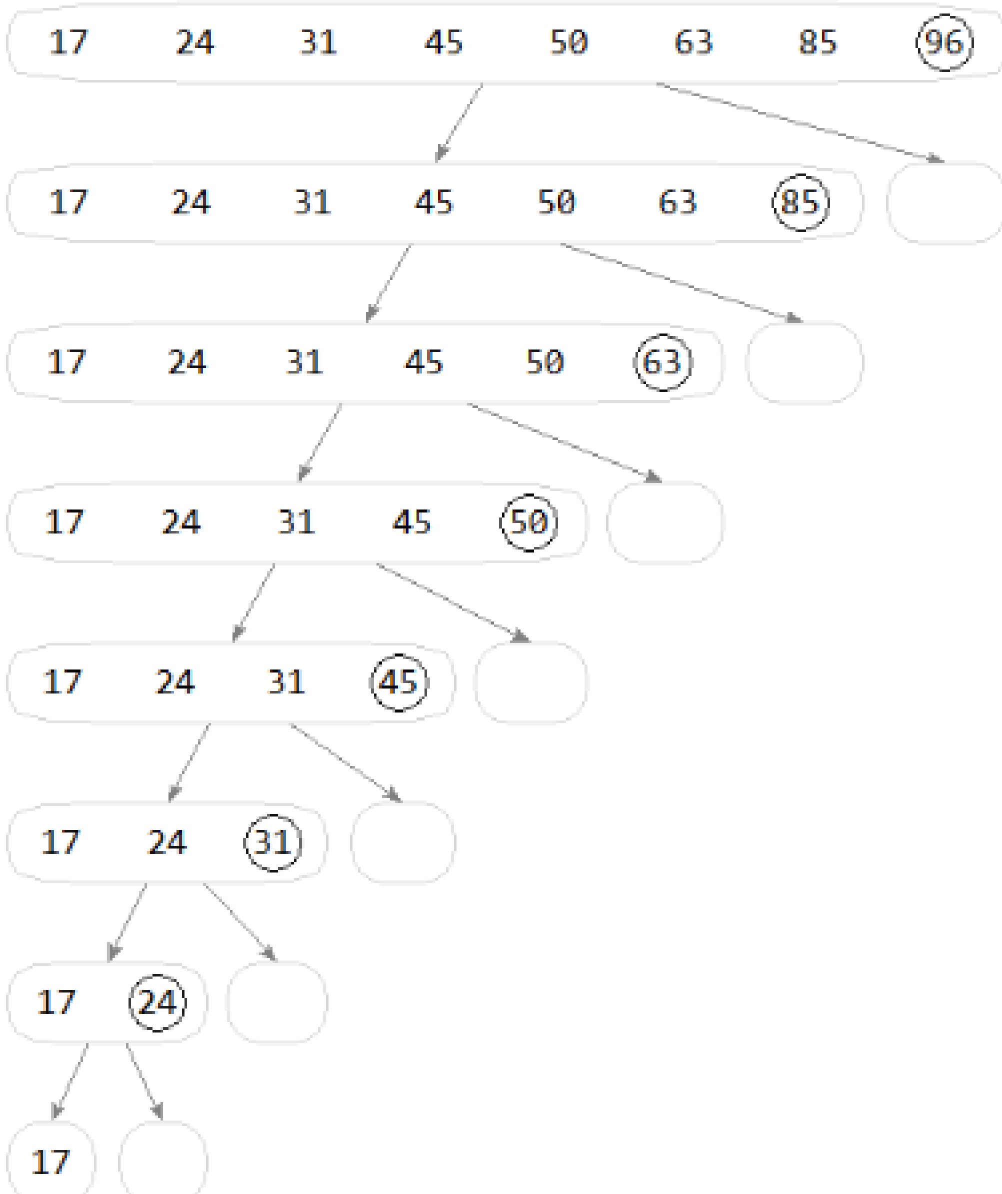
```
def quick_sort(pole):
    def quick(z, k):
        if z < k:
            # rozdelenie na dve casti
            index = z
            pivot = pole[z]
            for i in range(z+1, k+1):
                if pole[i] < pivot:
                    index += 1
                    pole[index], pole[i] = 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**.

7.3 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. bubble sort, insert sort, ale aj prvý quick sort, ktorý ešte nebol **in-place**), iné nie sú stabilné (napr. 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)**

7.4 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 toto sa blíži k `2*n`, teda celková zložitosť algoritmu `select()` je **O(n)**.

7.5 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

selection-sort

1. Navrhnite také vstupné pole pre **selection-sort**, na ktorom ukážete, že tento algoritmus nie je **stabilné** triedenie:

```
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]
```

merge-sort

2. Zapíšte rekurzívny **in-place** algoritmus **merge-sort** (s pomocným poľom pri zlučovaní, dajte pozor, aby bol stabilny), ktorý bude triediť dvojice `(key, value)` len podľa kľúča. Otestujte jeho rýchlosť a skontrolujte správnosť utriedenia (pre rôzne veľké náhodné kľúče a hodnoty) v porovnaní so štandardným `sorted()`.
3. Zapíšte nerekurzívny **merge-sort** zdola nahor a otestujte jeho rýchlosť v porovnaní s rekurzívnym algoritmom v úlohe (2). Dbajte pri tom, aby bolo aj toto triedenie stabilné.

quick-sort

4. Upravte oba rekurzívne algoritmy **quick-sort** z prednášky tak, aby triedili dvojice `(key, value)` len podľa kľúča. Otestujte ich rýchlosť a korektnosť (pre rôzne veľké náhodné kľúče a hodnoty) v porovnaní so štandardným `sorted()`. Týmto otestujete aj to, či sú to stabilné triedenia. V oboch algoritmoch zvoľte výber pivota rovnako.
5. Pre prvý algoritmus **quick-sort** z prednášky, ktorý nie je **in-place**, nastavte výber pivota ako stredný prvok poľa (`pivot=pole[len(pole)//2]`). Vygenerujte také testovacie 1000-prvkové pole, na ktorom tento algoritmus spadne (na pretečení rekurzie).
6. Otestujte, ako sa zmení rýchlosť algoritmu **quick-sort**, ak sa bude úsek menší ako nejaké `X` triediť pomocou **insert-sort** - toto tredenie **insert-sort** upravte tak, aby triedilo len zadaný úsek poľa (a nie celé pole).
   - otestujte pre rôzne zvolenú veľkosť `X`, napr. 10, 50, 100, ... a to pre to isté veľké náhodné pole
7. Do algoritmu **quick-sort** pridajte ďalší parameter `key`, ktorý má rovnaký význam ako v štandardnej funkcii `sorted()`: týmto parametrom je 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)]
```

8. Zrealizujte algoritmus **quick-sort** bez rekurzie pomocou vlastného zásobníka:
   - do zásobníku treba ukladať dvojice (začiatok, koniec úseku)

- algoritmus teda najprv rozhádže prvky poľa do dvoch úsekov (menšie ako pivot a väčšie ako pivot), potom hranice oboch týchto úsekov vloží do zásobníka a v cykle vyberie z vrchu zásobníka aktuálne spracovávaný úsek, spracuje ho a prípadne do zásobníka opäť vloží hranice úsekov, …
- do zásobníka najprv vkladajte väčší z dvoch úsekov na triedenie a potom až menší z nich (zamyslite sa prečo)

bucket-sort

9. Vygenerujte n-prvkové náhodné pole dvojíc `(key, value)`, kde `key` bude z intervalu `<0, k)` a `value` je ľubovoľná náhodná hodnota. Otestujte triedenie tohto poľa pomocou **bucket-sortu** pre rôzne hodnoty `n` a `k`: porovnajte čas triedenia a správnosť výsledku so štandardným triedením `sorted()`. Zabezpečte, aby bolo triedenie stabilné.
10. Upravte predchádzajúce triedenie na **radix-sort**: v ktorom v každom prechode sa utriedi len jedna cifra z kľúča. T.j. v prvom prechode sa vytvorí 10 priehradiek a vstupné pole sa roztriedi podľa poslednej cifry; v druhom prechode sa pole roztriedi podľa predposlednej cifry, …

hľadanie k-teho najmenšieho prvku

11. Naprogramujte rýchle hľadanie k-teho prvku v poli (typu `list`) úpravou algoritmu **quick-sort**

- algoritmus:

```
def select(pole, k):
    ...
```

- ak je `k` mimo rozsah indexov poľa, funkcia vygeneruje chybu `IndexError`
- funkciu otestujte na rýchlosť aj správnosť pre rôzne veľké polia

KAPITOLA 8

8. Dynamické programovanie

Na riešenie niektorých úloh, v ktorých hľadáme nejaké optimálne riešenie, môžeme niekedy použiť metódu **dynamické programovanie**. Idea sa trochu podobá metóde **rozdeľuj a panuj**, preto si ju pripomeňme:

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 (podproblémy nemusia byť navzájom disjunktné - bolo by dokonca dobré, keby sa navzájom prekrývali)
2. z týchto jednoduchých riešení postupne skladáme riešenia stále zložitejších a zložitejších prípadov problému a všetko toto zapisujeme do tabuľky
3. takto pokračujeme, kým sa nedostaneme k riešeniu požadovaného problému

Táto metóda funguje na princípe **zdola nahor** a väčšinou využíva pomocnú pamäť 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, rozdelením 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.

8.1 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 bez pomocnej tabuľky, napr.

```
def fib(n):
    f1, f2 = 1, 0
    for i in range(n):
        f1, f2 = f2, f1+f2
    return f2
```

V tomto riešení sme prepísali rekurzívnu úlohu postupným riešením najprv najjednoduchších podproblémov (postupne pre `n=0` a `n=1`) a z nich zostavujeme zložitejšie riešenia - tu sme tabuľku veľkosti **n** nahradili len dvomi pomocnými premennými.

Inokedy sa pri riešení iných úloh pomocou dynamického programovania ukáže, že namiesto kompletnej dvojrozmernej tabuľky stačí zostavovať len jej posledný riadok (prípadne posledné dva). Podobne ako nám z jednorozmernej tabuľky pre fibonacciho čísla nakoniec stačili len dve posledné hodnoty.

Túto istú úlohu môžeme riešiť pomocou **memoizácie** (riešili sme to na 2. cvičení v (7) úlohe):

- 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:

```
mem = {}                # globálna premenná

def fib(n):
    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
```

alebo s využitím inicializovaného *mutable* parametra:

```
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 tiež sa v ňom vytvára tabuľka s riešeniami všetkých podúloh. Hovoríme, že **memoizácia** je špeciálnym prípadom **dynamického programovania**. Využije sa hlavne pri jednoduchších úlohách, ktoré vieme zapísať rekurzívne. V Pythone existuje machanizmus, pomocou ktorého sa dá memoizácia zapísať veľmi elegantne (tzv. dekorátory), napr.

```
import functools

@functools.lru_cache(maxsize=None)          # dekorátor
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

```
>>> print(*(fib(n) for n in range(16)))
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

8.2 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)**:

```
def komb(n, k):
    if k == 0 or n == k:
        return 1
    return komb(n - 1, k) + komb(n - 1, k - 1)
```

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 programovanie, ale tabuľku za nás postupne zapĺňa Python pri rekurzívnych volaniach.

8.3 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).

Postupne ju vyriešime niekoľkými spôsobmi

8.3.1 Pomocou Greedy metódy

Greedy t.j. **pažravý** algoritmus označuje:

- 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):
    zoznam = []
    mince = sorted(mince)
    while suma > 0:
        i = len(mince)-1
        while i >= 0 and mince[i] > suma:
            i -= 1
        if i < 0:
            return None
        suma -= mince[i]
        zoznam.append(mince[i])
    return zoznam[::-1]
```

otestujeme:

```
for suma in range(1, 21):
    print(suma, sorted(mincovka1([1, 2, 5, 10], suma)))
```

Predchádzajúci algoritmus 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). Zrejme je tento algoritmus veľmi rýchly, ale nie vždy funguje optimálne: niekedy nájde nie najlepšie riešenie.

8.3.2 Hrubá sila

Vyriešme túto úlohu hrubou silou, teda pomocou backtrackingu:

```
def mincovka2(mince, suma):
    if suma in mince:
        return [suma]
    naj = None
    for minca in mince:
        if minca < suma:
            ries = mincovka2(mince, suma-minca) + [minca]
            if naj is None or len(ries) < len(naj):
                naj = ries
    return naj
```

```
>>> mincovka2([1, 3, 7, 10], 14)
[7, 7]
```

Zložitosť tohto algoritmu je **O(2**n)**

8.3.3 Memoizácia

Predchádzajúci rekurzívny algoritmus je pre väčšie hodnoty veľmi pomalý (podobne ako rekurzívna fibonacciho funkcia). Môžeme ho výrazne urýchliť pomocou **memoizácie**:

- zadefinujeme **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, mem={}):
    if suma in mem:
        return mem[suma]              # už sme to počítali predtým
    if suma in mince:
        mem[suma] = [suma]
        return [suma]
    naj = None
    for minca in mince:
        if minca < suma:
            ries = mincovka3(mince, suma-minca) + [minca]
            if naj is None or len(ries) < len(naj):
                naj = ries
    mem[suma] = naj
    return naj
```

Zložitosť tohlo algoritmu je **O(mn)**, keď **m** je počet mincí a **n** je samotná rozmieňaná suma.

Vyskúšajte zavolať obe tieto funkcie aj pre väčšie hodnoty, napr.

```
>>> mincovka2([1, 3, 7, 10, 20], 45)
?
>>> mincovka3([1, 3, 7, 10, 20], 45)
?
```

Preskúmajme obsah poľa `mem` po jednom zavolaní `mincovka3`:

```
>>> mincovka3([1, 3, 7, 10], 14)
[7, 7]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> mincovka3.__defaults__[0]
{10: [10], 7: [7], 3: [3], 1: [1], 4: [3, 1], 2: [1, 1], 5: [3, 1, 1],
 8: [7, 1], 11: [10, 1], 6: [3, 3], 9: [7, 1, 1], 12: [10, 1, 1],
 13: [10, 3], 14: [7, 7]}
>>> sorted(mincovka3.__defaults__[0])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

8.3.4 Dynamické programovanie

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]
```

Odhadnime časovú zložitosť algoritmu:

Teraz dopíšeme 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. Vypíšme túto tabuľku napr. pre `mincovka4([1,3,7,10,20], 19)`:

```
[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

Výsledný program:

```
def mincovka4(mince, suma):
    tab = [0] + [None] * suma
    for s in range(1, suma+1):
        for m in mince:
            if s >= m and tab[s-m] is not None:
                if tab[s] is None or tab[s] > tab[s-m]+1:
                    tab[s] = tab[s-m]+1
    #print(tab)
    return tab[suma]
```

napr.

```
>>> mincovka4([3, 5], 7)
[0, None, None, 1, None, 1, 2, None]
```

8.4 Najdlhšie podpostupnosti

Ďalšou skupinou úloh sú problémy, v ktorých sa v jednorozmernom poli čísel (alebo pre dve jednorozmerné polia, nemusia byť číselné) hľadá:

1. najdlhšiu *súvislú* rastúcu podpostupnosť (sekvencia)
   - napr. pre `3, 7, 2, 9, 4, 1, 5, 6, 10, 8, 0` dostaneme `1, 5, 6, 10`
2. najdlhšiu *vybranú* rastúcu podpostupnosť
   - napr. pre `3, 7, 2, 9, 4, 1, 5, 6, 10, 8, 0` dostaneme, napr. `2, 4, 5, 6, 8` alebo `3, 4, 5, 6, 10`
3. najdlhšiu *súvislú* spoločnú podpostupnosť dvoch postupností
4. najdlhšiu *vybranú* spoločnú podpostupnosť dvoch postupností

Algoritmy všetkých štyroch úloh sú navzájom veľmi podobné a samozrejme, že sa riešia pomocou dynamického programovania. My si ukážeme riešenie 2. úlohy.

8.4.1 Najdlhšia vybraná rastúca podpostupnosť

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, zrejme túto hodnotu budeme počítať len z prvých `i-1` prvkov, t.j.
  - `tab[0]` je zrejme 1, lebo ak berieme prvky poľa len po 0, tak tento je 1-prvkovou postupnosťou
  - `tab[1]` má byť dĺžkou najdlhšej rastúcej podpostupnosti, ak berieme do úvahy len 0. a 1. prvok poľa a pritom vybraná podpostupnosť končí 1. prvkom: bude tu 1 alebo 2
  - `tab[2]` je opäť dĺžkou najdlhšej, ak berieme len prvé 3 prvky, pričom nás zaujímajú len podpostupnosti, ktoré končia 3. prvkom: tretí prvok môže nadväzovať buď na postupnosť končiacu v prvom alebo končiacu v druhom prvku (závisí to od toho, či je tretí prvok väčší ako prvý alebo druhý a ktorá z týchto podpostupností je dlhšia)
  - takto postupne vybudujeme celú tabuľku `tab`
- dynamicky toto pole `tab` budete vytvárať budete pre vstupnú postupnosť `post` takto:
  - pre `tab[i]` nájdeme najväčšiu hodnotu medzi `tab[0:i-1]` (na indexe `j`), pre ktorú je `post[j] < post[i]` - do `tab[i]` zapíšeme túto najväčšiu hodnotu zvýšenú o 1
  - ak medzi `tab[0:i-1]` nie je žiadna, pre ktorú by platilo `post[j] < post[i]`, do `tab[i]` zapíšeme 1 (zrejme `post[i]` nie je väčšie ako žiadne z `post[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)

Riešenie:

```
def hladaj(post):
    tab = [0] * len(post)
    for i in range(len(post)):
        max = 0
        for j in range(i):
            if post[j] < post[i]:
                if tab[j] > max:
                    max = tab[j]
        tab[i] = max + 1
    print('tabulka:', *tab)
    max_i = 0
    for i in range(len(tab)):
        if tab[i] > tab[max_i]:
            max_i = i
    dlzka = tab[max_i]
    print('naj dlzka =', dlzka)
    vysl = [post[max_i]]
    i = max_i
    for d in reversed(range(1, dlzka)):
        for j in range(i):
            if tab[j] == d and post[j] < post[i]:
                break
        vysl.insert(0, post[j])
        i = j
    print('hladana podpostupnost:', *vysl)
    return vysl
```

```
>>> pole = [3, 7, 2, 9, 4, 1, 5, 6, 10, 8, 0]
>>> hladaj(pole)
tabulka: 1 2 1 3 2 1 3 4 5 5 1
naj dlzka = 5
hladana podpostupnost: 3 4 5 6 10
[3, 4, 5, 6, 10]
```

8.5 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

kombinačné čísla

1. Zapíšte a otestujte rekurzívne riešenie `komb(n, k) = komb(n-1, k) + komb(n-1, k-1)`
   - zistite počet rekurzívnych volaní pre n in (20,21,22,...30): `komb(n, k=15)`
2. Zapíšte nerekurzívnu verziu `komb1(n, k)`, ktorá najprv skonštruuje dvojrozmernú tabuľku pre `n` riadkov a `k` stĺpcov a na základe nej vráti výsledok
   - funkcia:

```
def komb1(n, k):
    # ... vytvorte dvojrozmernú tabuľku veľkosti n x k, kde tab[i][j] =␣
↪hodnota komb(i, j)
    # ... tabuľku vytvorte metódou zdola nahor, t.j. najprv pre malé i a j
    return tab[n][k]
```

3. Riešte ako v (2) ale namiesto dvojrozmernej tabuľky pracujte len s jedným riadkom (z i-teho vyrobte (i+1)-ty riadok)

   - funkcia:

```
def komb2(n, k):
    ...
```

   - otestujte správnosť, napr.

```
for n in range(10):
    for k in range(0, n+1):
        if not komb(n, k) == komb1(n, k) == komb2(n, k):
            print(n, k, komb(n, k), komb1(n, k), komb2(n, k))
```

4. Riešte memoizáciou, t.j. do rekurzívneho riešenia pridajte pamätanie si vypočítaných hodnôt a ich neskoršie použitie

   - funkcia:

```
def komb3(n, k, mem={}):
    ...
```

   - otestujte

funkcia P

5. 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`

   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 j=10
   - 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 tejto verzie

najdlhšia vybraná podpostupnosť

9. Na základe algoritmu z prednášky pre „najdlhšia vybraná rastúca podpostupnosť" najprv
   - ručne bez počítača vytvorte tabuľku dĺžok pre danú postupnosť:

   ```
   post = [5, 4, 2, 4, 1, 1, 3, 4, 3, 4, 4, 1, 1, 2, 3, 3, 5, 4, 2, 2]
   ```

   - z tejto tabuľky dĺžok zrekonštruujte hľadanú vybranú podpostupnosť (úloha môže mať viac riešení)
   - tabuľku aj výsledok skontrolujte spustením programu z prednášky
10. Upravte funkciu `hladaj()` z prednášky tak, aby hľadal **najdlhšiu vybranú neklesajúcu podpostupnosť** (z predchádzajucej postupnosti `post` vie takto vybrať podpostupnosť dĺžky 8)
    - teraz pre postupnosť:

    ```
    >>> post = [5, 4, 2, 4, 1, 1, 3, 4, 3, 4, 4, 1, 1, 2, 3, 3, 5, 4, 2, 2]
    >>> len(hladaj_neklesajucu(post))
    8
    ```

11. Algoritmus z prednášky najprv vytvára tabuľku maximálnych dĺžok podpostupností, potom v tejto tabuľke nájde maximálny prvok a na koniec z nej zrekonštruuje hľadanú podpostupnosť. Úloha by sa dala vyriešiť aj trochu inak:
    - tabuľka sa vytvára rovnakým postupom, ale zapisujú sa do nej nie dĺžky ale nájdené maximálne podpostupnosti
    - z tejto tabuľky potom stačí vyhľadať ľubovoľnú podpostupnosť maximálnej dĺžky
    - naprogramujte túto ideu a skontrolujte s výsledkami úloh v (9) a (10) - mali by ste dostať rovnaké výsledky
    - odhadnite pamäťovú zložitosť oboch algoritmov (zaujíma nás použitá pamäť v najhoršom prípade)
      – z porovnania pamäťovej zložitosti týchto dvoch algoritmov by mohlo byť jasné, prečo sa použil variant z prednášky

najdlhšia súvislá rastúca podpostupnosť

12. Riešenie tejto úlohy sa veľmi podobá predchádzajúcemu algoritmu z prednášky pre vybranú podpostupnosť:
    - najprv sa vytvorí tabuľka maximálnych dĺžok súvislých podpostupností, ktoré končia na danom indexe:
      – ak je `post[i-1] < post[i]`, potom `tab[i] = tab[i-1] + 1`, inak `tab[i] = 1`
    - potom sa v tejto tabuľke nájde maximálny prvok - to je dĺžka maximálnej podpostupnosti a index vyjadruje index posledného prvku z tejto podpostupnosti
    - spätne sa vytvorí hľadaná súvislá podpostupnosť
    - ručne vytvorte tabuľku pre

    ```
    pole = [3, 7, 2, 9, 4, 1, 5, 6, 10, 8, 0]
    ```

    - zapíšte algoritmus a skontrolujte vytvorenú tabuľku
13. Úloha sa dá riešiť aj bez použitia tabuľky: stačí si pamätať doteraz najlepšiu dĺžku a index, kde končí táto podpostupnosť
    - zapíšte algoritmus bez použitia tabuľky (pamäťová zložitosť bude teraz **O(1)**)
    - skontrolujte výsledky porovnaním s algoritmom, ktorý pracuje s tabuľkou

KAPITOLA 9

9. Spracovanie textov

Otestujme štandardnú metódu `str.find()`, ktorá hľadá prvý výskyt nejakého podreťazca:

```
>>> 'python'.find('th')
2
>>> 'python'.find('ty')
-1
```

Je to rýchle aj pre dlhšie reťazce:

```
>>> ('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):          # podobné ako str.find(ret, pod)
    n, m = len(ret), len(pod)
    for j in range(n-m+1):
        k = 0
        while k < m and ret[j+k] == pod[k]:
            k += 1
        if k == m:
            return j
    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.

9.1 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)
    k = 0
    for j in range(n):
        while k > 0 and ret[j] != pod[k]:
            k = skok[k-1]
        if ret[j] == pod[k]:
            k += 1
        if k == m:
            return j-m+1
    return -1
```

Aby toto fungovalo, konštruuje sa špeciálna tabuľka rýchlych skokov:

```
def pocitaj_kmp(p):
    m = len(p)
    skok = [0] * m
    k = 0
    for j in range(1, m):
        while k > 0 and p[j] != p[k]:
            k = skok[k-1]
        if p[j] == p[k]:
            k += 1
        skok[j] = k
    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.

Algoritmus **KMP** má veľmi názorné vizualizácie na stránkach:

- Knuth-Morris-Pratt String Search
- Visualizing String Matching Algorithms

9.2 Longest Common Subsequence

Algoritmus **LCS** označuje 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**.

Ukážeme riešenie pomocou metódy dynamické programovanie.

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 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 väčšia 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 = lcs_tab(x, y)
    ries = ''
    i, j = len(x), len(y)
    while tab[i][j] > 0:
        if x[i-1] == y[j-1]:
            #print(i-1, j-1, x[i-1])
            tab[i][j] = -tab[i][j]
            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
```

Otestujme:

```
x, y = 'abbccbddbd', 'cadbddbbada'
print(f'x = {x!r}\ny = {y!r}')
print('lcs =', lcs_ries(x, y))
```

```
x = 'abbccbddbd'
y = 'cadbddbbada'
lcs = abddbd
```

Aj algoritmus **LCS** má veľmi názornú vizualizáciu na stránke:

- Dynamic Programming (Longest Common Subsequence)

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
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
'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:

```python
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.

Algoritmus **Huffman** má tiež veľmi názornú vizualizáciu na stránke:

- Huffman Coding

9.4 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

9.4.1 KMP

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:

   ```
   >>> str.find('a'*1000000, 'a'*10000+'b')
   >>> str.find('a'*1000000, '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 `'ababadababacabab'` a `'ababaca'`:

   - vypíšte zostavené pomocné pole `skok`
   - zamyslite sa nad tým, ako by ste toto pole vytvárali ručne na papieri
   - 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`)
   - zamyslite sa nad tým, ako by ste tento algoritmus raelizovali ručne na papieri

3. Rovnaké testy ako v (1) spustite aj pre algoritmus hľadania **KMP** (bez trasovania z (2))

9.4.2 LCS

4. Spojazdnite algoritmy **LCS** 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'` (zo 6. cvičenia) prečítajte po 1000 znakov ale až od 100. znaku, zistite **LCS**

5. Ručne vytvorte LCS-tabuľku pre reťazce `'aabaacdb'` a `'abcabcabc'`

   - zamyslite sa nad tým, ako by sa zmenila táto tabuľka, keby sme ju vytvárali pre reťazce `'abcabcabc'` a `'aabaacdb'`
   - potom z tejto tabuľky ručne zistite nájdený najdlhší spoločný vybraný podreťazec týchto dvoch reťazcov

9.4.3 Huffmanovo kódovanie

6. 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
   - zakódujte reťazec `'uma eum uaa eme'` - dostávate asi 36 bitový reťazec `0` a `1` - teraz tento reťazec bitov ručne rozkódujte
7. Spojazdnite **Huffmanov algoritmus** z prednášky pre tieto implementácie stromu a frontu

implementácia BinTree a PriorityQueue

```
import tkinter

class BinTree:
    class Node:
        def __init__(self, data, left=None, right=None):
            self.data, self.left, self.right= data, left, 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)
```
(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        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

- otestujte vykreslením stromu z úlohy (6)

8. 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

KAPITOLA 10

10. Prefixové stromy

10.1 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. `'#'`.

Vrchol prefixového stromu definujeme najčastejšie takto:

```
class Node:
    def __init__(self):
        self.data = None
        self.child = {}
```

teda každý vrchol môže obsahovať nejakú informáciu (napr. že je to koncový vrchol nejakého slova) a asociatívne pole všetkých svojich podstromov.

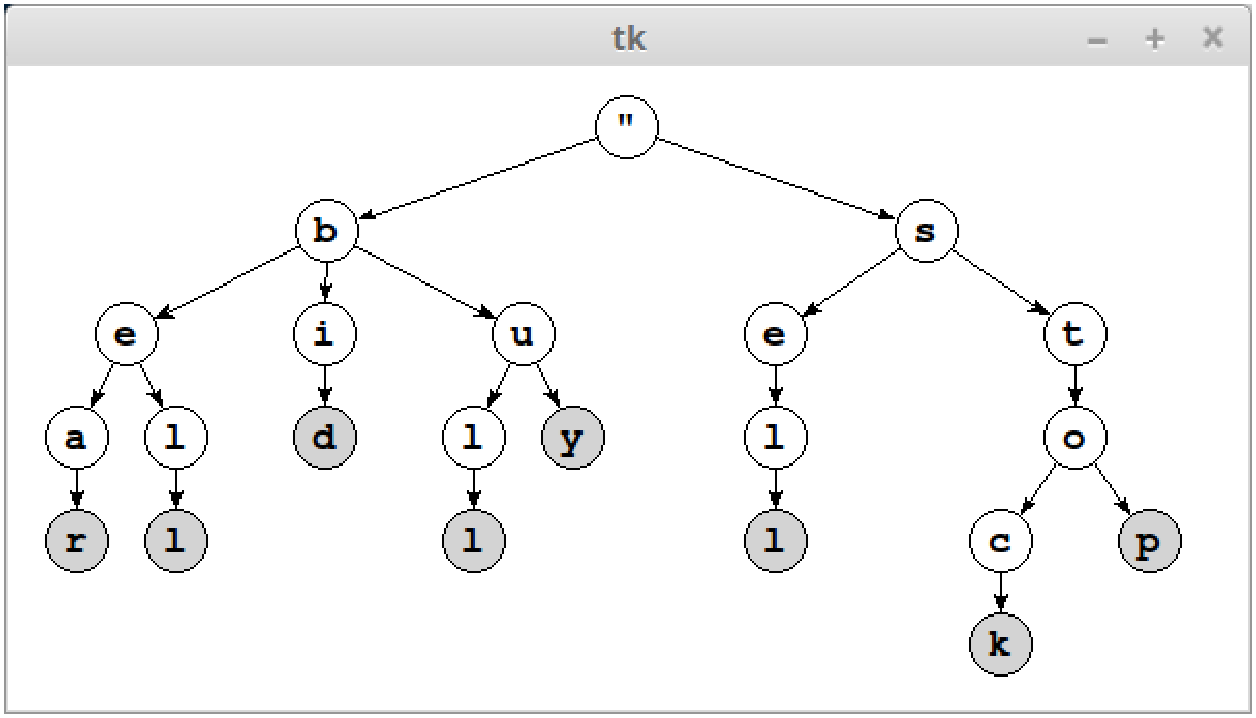
Vložiť nové slovo do stromu znamená:

```
class Trie:

    class Node:
        def __init__(self):
            self.data = None
            self.child = {}

    def __init__(self):
        self.root = self.Node()

    def insert(self, word, data=1):
        node = self.root
        for char in word:
            if char not in node.child:
                node.child[char] = self.Node()
            node = node.child[char]
        node.data = data
```

Slová do stromu pridáme napr. takto:

```
tr = Trie()
for slovo in 'bear bell bid bull buy sell stock stop'.split():
    tr.insert(slovo)
```

Dostávame takýto prefixový strom:



Uvedomte si, že vo vrcholoch tohto stromu nie sú tie znaky, ktoré sú tu zobrazené.

Pomocou rekurzívnej funkcie vieme vygenerovať všetky uložené slová v takomto strome:

```
class Trie:
    ...

    def all(self, node, word=''):
        if node.data is not None:
            yield word
        for char in node.child:
            yield from self.all(node.child[char], word + char)
```

```
for slovo in tr.all(tr.root):
    print(slovo)
```

Vieme z toho urobiť iterátor:

```
class Trie:
    ...

    def __iter__(self):
        yield from self.all(self.root)
```

teraz môžeme získať všetky slová z tohto stromu:

```
print(*tr)
```

Vďaka prefixovému stromu vieme veľmi jednoducho získať všetky slová, ktoré majú zadaný **prefix**:

```
class Trie:
    ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def prefix(self, word):
        node = self.root
        for char in word:
            if char not in node.child:
                return None
            node = node.child[char]
        return self.all(node, word)
```

napr. pre súbor s anglickými slovami text5.txt:

```
with open('text5.txt') as f:
    for w in f.read().split():
        t.insert(w)
print(len(list(t)))
```

Vidíme, že sa vytvoril trie s 67527 slovami. Môžeme experimentovať s hľadaním všetkých slov, ktoré majú rovnaký nejaký **prefix**, napr.

```
>>> list(t.prefix('tree'))
['tree', 'treehopper', 'trees', 'treenail']
>>> list(t.prefix('trie'))
['trier', 'tried', 'triennial', 'triennium']
>>> list(t.prefix('python'))
['python', 'pythoness', 'pythonidae', 'pythoninae']
```

10.1.1 Realizácia asociatívneho poľa

Prefixový strom vieme využiť ako asociatívne pole, ale musíme na to zadefinovať niekoľko metód:

- `__getitem__()`
- `__setitem__()`
- `__delitem__()`
- `__len__()`

Počet prvkov v trie by sme vedeli zistiť nejako takto:

```
class Trie:
    ...

    def count(self, node=None):
        if node is None:
            node = self.root
        p = node.data is not None
        for char in node.child:
            p += self.count(node.child[char])
        return p
```

Lenže táto metóda má zbytočne veľkú zložitosť zložitosť. Vhodnejšie by bolo použitie jedného počídla, ktoré sa zvyšuje alebo znižuje v metódach `__setitem__()` a `__delitem__()`.

Uvedomte si, že takéto asociatívne pole bude fungovať len pre kľúče znakové reťazce. Momentálne realizácia trie zisťuje, či nejaký vrchol reprezentuje koncový znak reťazca tým, že je tam hodnota rôzna od `None`. Lenže to znamená, že asociovaná hodnota by nemohla byť `None`. Aby to bola korektná realizácia asociatívneho poľa, bude treba vyriešiť aj tento problém.

10.1.2 Frekvenčná tabuľka

Prefixové stromy sa dajú veľmi vhodne využiť na evidovanie počtu výskytov jednotlivých slov. Vtedy samotná hodnota vo vrcholoch stromu bude označovať počet výskytov príslušného slova (ktoré skončilo v tomto vrchole).

Niekedy sa do každého vrcholu stromu vkladá ešte jedno počítadlo. Toto bude označovať počet rôznych slov, ktoré končia v celom podstrome tohto vrchola. Zrejme, v koreni vtedy bude počet rôznych slov v celom prefixovom strome (teda `len()`).

10.1.3 Binárny trie

Zaujímavé môžu byť aj špeciálne typy trie, ktoré reprezentujú len slová zložené z dvoch typov znakov.

- napr. `{'a', 'b'}`, `{0, 1}`
- dostávame obyčajný binárny strom - môžeme použiť implementáciu obyčajného binárneho stromu (t.j. ľavý podstrom je pre prvé písmeno abecedy a pravý pre druhé)
- v takomto trie môžeme ukladať aj postupnosti bitov celého čísla, t.j. bežné `int`

10.1.4 Realizácia pomocou brat/syn

Naša realizácia `Trie` využíva pythonovský `dict` - to môže byť niekedy veľký problém (napr. realizácia v inom programovacom jazyku). Z tohto dôvodu sa často používa realizácia všeobecného stromu pomocou dvoch smerníkov (referencií) v každom vrchole (v programovaní v 1. ročníku sme tomuto hovorilo „syn/brat“):

- jedna referencia (`child`) je na prvého syna
- druhá (`next`) na brata
- v každom vrchole je okrem dátového atribútu (`data`) aj samotný symbol (resp. podreťazec)

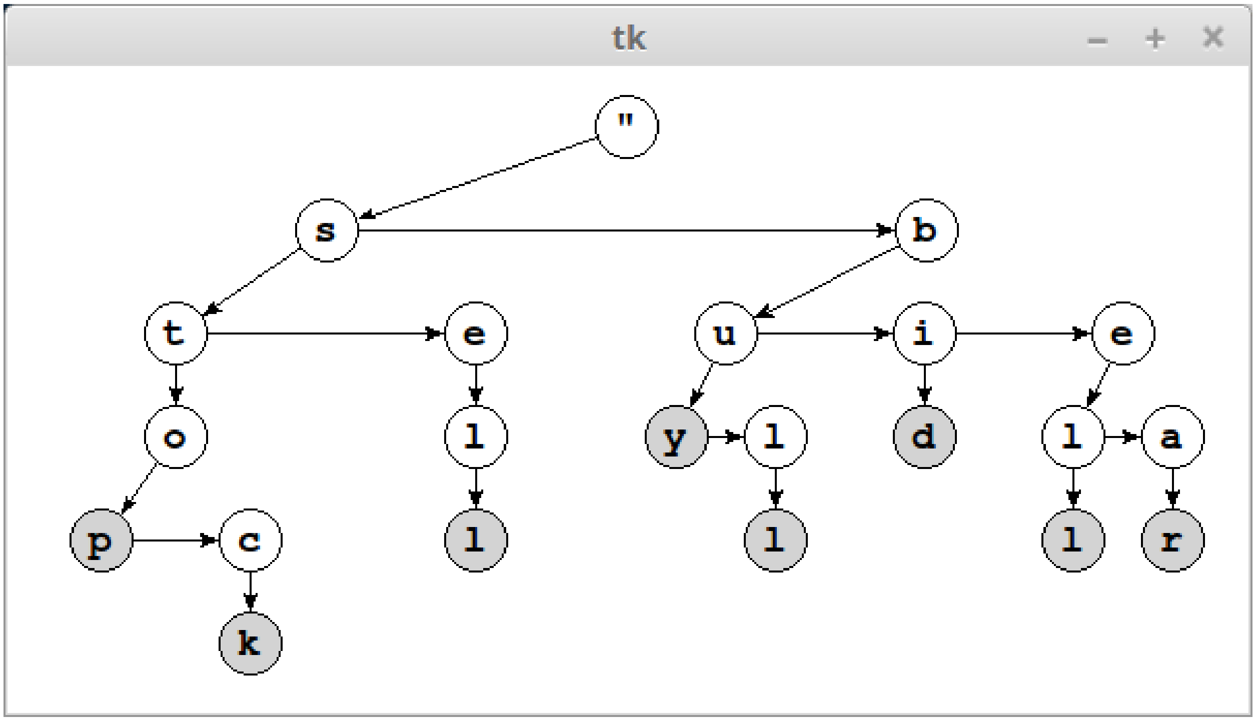
Vrchol takéhoto stromu zapíšeme:

```
class Node:
    def __init__(self, char='', next=None):
        self.child = None
        self.next = next
        self.char = char
        self.data = None
```

Vkladanie nového slova:

```
def insert(self, word, data=1):
    node = self.root
    for char in word:
        node1 = node.child
        while node1 is not None and node1.char != char:
            node1 = node1.next
        if node1 is None:
            node1 = node.child = self.Node(char, node.child)
        node = node1
    node.data = data
```

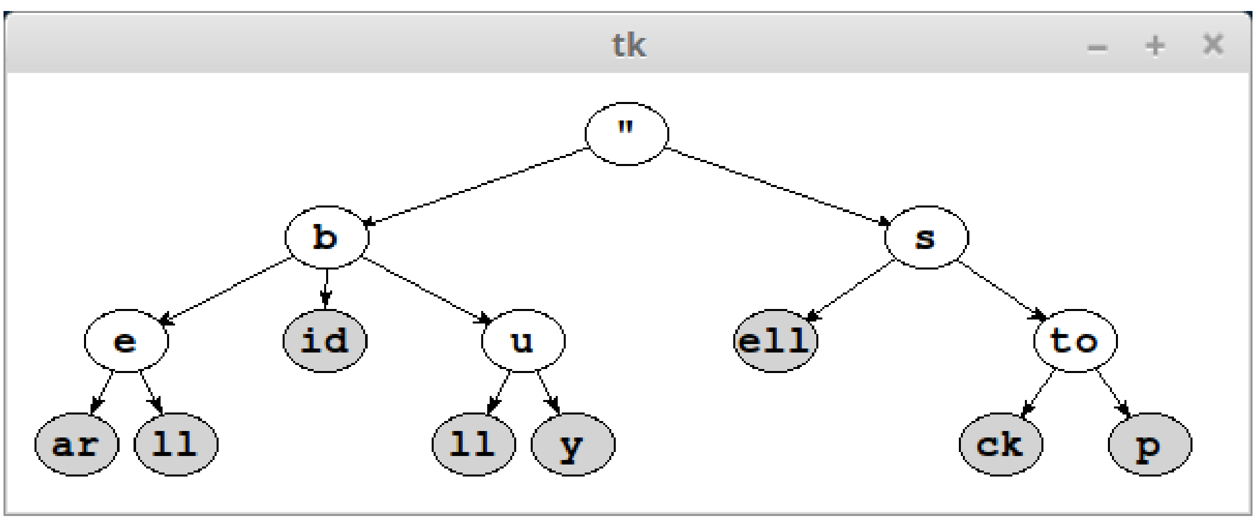
Ten istý trie ako je zobrazený vyššie, teraz vyzerá takto:



Uvedomte si, že vodorovné šípky označujú referenciu `next`. t.j. „brat“ a smerom nadol označujú `child`, t.j. „prvý syn“. Všetci synovia nejakého vrchola sú vlastne všetci bratia prvého syna plus tento prvý syn.

10.1.5 Compressed trie

Zhustený prefixový strom - všetky vrcholy, ktoré by mali len jedného syna, sa spoja s týmto jediným synom a teda hrana nemusí označovať len jeden symbol, ale niekoľko za sebou idúcich symbolov. Predchádzajúce trie by teraz vyzeralo takto:



10.1.6 Sufixový trie

Je to prefixový strom, ktorý je zložený zo všetkých „sufixov“, t.j. prípon slova, resp. slov. Napr. slovo `'abrakadabra'` má tieto sufixy: `'a'`, `'ra'`, `'bra'`, `'abra'`, `'dabra'`, `'adabra'`, `'kadabra'`, 'akadabra','rakadabra','brakadabra','abrakadabra' a všetky tieto sa vložia do sufixového stromu. Sufixové trie sa využívajú hlavne na rýchle hľadanie reťazcov a podreťazcov v nejakých textoch.

10.1.7 Využitie

Pomocou trie by sem vedeli riešiť napr. takýto problém:

- spracujeme nejakú množinu webovú tak, že všetky „slová" z týchto stránok vložíme do prefixového stromu, hodnotami budú množiny stránok, kde sa tieto slová vyskytujú
- takto dostávame dosť veľký strom, v ktorom pre každé slovo uchovávame adresy príslušných webových stránok
- keďže tieto množiny stránok môžu byť dosť rozsiahle, uchovávať ich nemusíme v operačnej pamäti, ale napr. niekde na disku
- keď teraz dostaneme nejakú množinu slov, vyhľadáme ich v tomto trie, urobíme prienik príslušných množín, prípadne ich ešte usporiadame podľa nejakých preferencií

Takto dostávame veľmi rýchly algoritmus, ktorý (podobne ako google) nájde všetky stránky, ktoré obsahujú všetky zadané slová.

10.2 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

KAPITOLA 11

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 sústredené 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 objektmi 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:

Tabuľka 1: Tabuľka zložitostí metód

metóda	graf1	graf2	graf3	graf4	
`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):

- 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)**).

Nech `r` je jednorozmerné pole, 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, 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 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
```

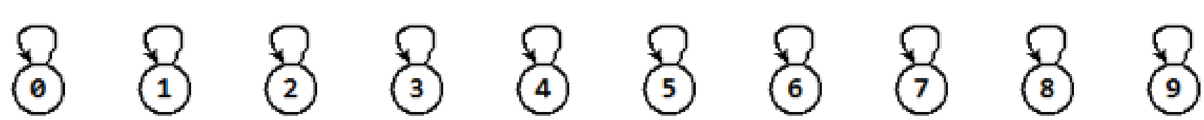
Takto sa dostane do koreňa stromu, čo je reprezentantom celej množiny (komponentu). Operácia **UNION()** zistí, ktorý z komponentov má menší **rank** (výšku stromu) a ten potom pripojí ako ďalšieho syna ku koreňu väčšieho stromu. Keďže výška nového stromu sa pritom nezmení, netreba meniť ani hodnotu `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
```

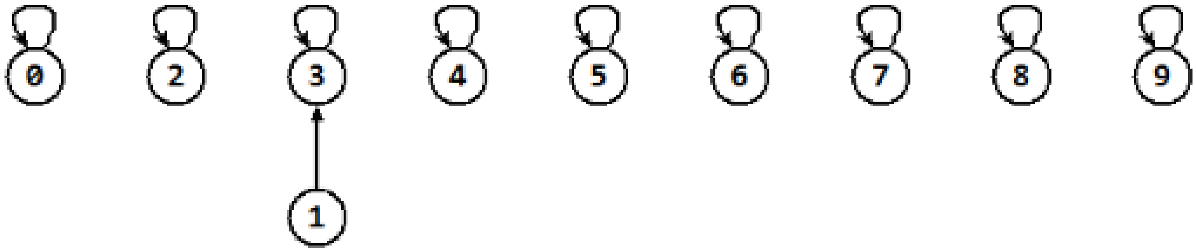
Časová zložitosť operácie **FIND(p)** závisí od výšky stromu, ktorý sa takto prechádza, čo je `rank` reprezentanta, teda koreňa stromu. Operácia **UNION()** urobí najprv 2 volania **FIND()** a potom len príkazy **O(1)**. Teda obe operácia **UNION()** aj **FIND()** majú rovnakú zložitosť. Keďže strom 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.

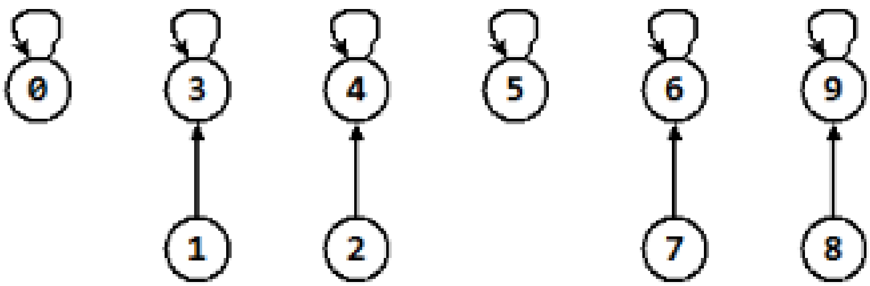
Príklad, ktorý sme robili pri tabuľkovej reprezentácii disjunktných množín, môžeme zopakovať aj pre stromčeky. Začnime s 10 jednoprvkovými množinami:



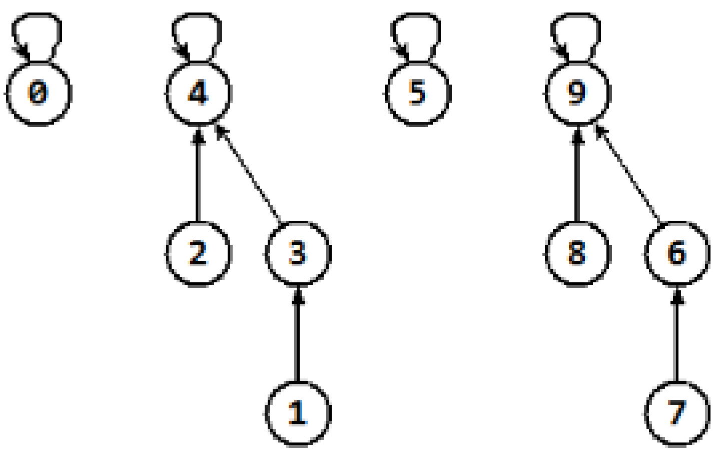
Najprv spojíme **UNION(1,3)**:



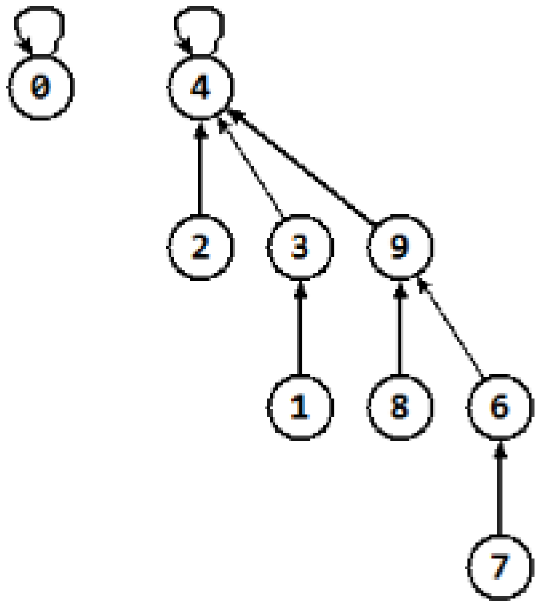
Teraz postupne **UNION(2,4)**, **UNION(7,6)**, **UNION(8,9)**:



Potom **UNION(3,4)**, **UNION(7,8)**:



Na záver **UNION(6,4)**, **UNION(1,9)**:



11.2.4 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 pomocou **Kruskalovho** algoritmu (budeme vytvárať takú množinu hrán, ktorých súčet ohodnotení je minimálny):
  1. každý vrchol grafu vytvoríme ako samostatnú množinu (**make_set(v)**)
  2. vytvoríme prioritný front všetkých hrán, pričom kľúčmi budú ohodnotenia hrán
  3. kým vo výslednej množine nie je **n-1** hrán, opakuje:
     - vyberie z prioritného frontu hranu s najmenšou váhou (**remove_min()**) -> (v1, v2)
     - ak tieto vrcholy ešte nie sú v tej istej disjunktnej množine (teda **find(v1)!=find(v2)**), tak ich zjednoť pomocou **union(v1, v2)** a pridaj do výslednej množiny hrán
  4. výsledná množina hrán je hľadaná minimálna kostra grafu

11.3 Cvičenie

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

11.3.1 Union-find

1. 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 (1). Pre každú z reprezentácií si vhodne zvoľte dátovú štruktúru, napr. ako pole objektov typu `Vertex`.

2. reprezentujte tabuľkou

   - definujte:

   ```
   def FIND(p):
       ...
   def UNION(p, q):
       ...
   ```

3. reprezentujte zoznamami

   - definujte:

   ```
   def FIND(p):
       ...
   def UNION(p, q):
       ...
   ```

4. reprezentujte stromčekmi s informáciou `rank` (výška stromu)

   - definujte:

   ```
   def FIND(p):
       ...
   def UNION(p, q):
       ...
   ```

5. reprezentujte stromčekmi ale bez ďalšej informácie (t.j. bez `rank`: rozhoduje sa náhodne s pravdepodobnosťou 1/2)

   - definujte:

   ```
   def FIND(p):
       ...
   def UNION(p, q):
       ...
   ```

6. 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

11.3.2 Reprezentácie grafov

7. 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 čast' 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
        ...
```

8. Pomocou algoritmu do hĺbky zistite, či sa dva vrcholy nachádzajú v tom istom komponente:

   - rekurzívnu funkciu **zapíšte** tak, aby používala len metódy z úlohy (2), 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

9. 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

10. Pomocou algoritmu do hĺbky zistite počet komponentov grafu.

- zapíšte:

```
class Graph:
    ...
    def component_count(self):
        ...
```

- odhadnite zložitosť tohto algoritmu

KAPITOLA 12

Prílohy

12.1 Test z ADŠ 2014/2015

1. Metóda `height` pre triedu `Strom` počíta výšku stromu pomocou metódy `depth` (hĺbky konkrétneho vrcholu, t.j. vzdialenosť vrcholu od koreňa):

```
def depth(self, p):
    if self.is_root(p):
        return 0
    else:
        return 1 + self.depth(p.parent)

def height(self):
    return max(self.depth(p) for p in self. preorder() if self.is_leaf(p))
```

   Odhadnite zložitosť (najhoršieho prípadu) metódy `height` pre strom s `n` vrcholmi.

2. Funkcia `podmnoziny` vráti zoznam všetkých podmnožín nejakej množiny:

```
def podmnoziny(mnozina):
    zoznam = [set()]
    for prvok in mnozina:
        for mn in zoznam[:]:
            zoznam.append(mn | {prvok})
    return zoznam
```

   Zistite, koľkokrát sa v tejto funkcii vykoná operácia zjednotenia:

```
mn | {prvok}
```

3. Operácie `enqueue` a `dequeue` pre dátovú štruktúru front (`Queue`) sme realizovali poľom: pridávaním na koniec a odoberaním prvého prvku poľa:

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

    def enqueue(self, prvok):
        self.__pole.append(prvok)

    def dequeue(self):
        if self.is_empty():
            raise Empty('prázdny front pre dequeue')
        return self.__pole.pop(0)
```

   Vidíme, že `dequeue` má zložitosť **O(n)**. Prepíšte tieto metódy: namiesto obyčajného poľa (pythonovský `list`) použijete asociatívne pole (pythonovský `dict`), o ktorom vieme, že jeho niektoré operácie majú zložitosť **O(1)**. Operácie takéhoto frontu by mali mať zložitosť **O(1)**.

4. Trieda `BS` popisuje zjednodušený binárny strom:

```
class BS:
    def __init__(self, info=0, l=None, p=None):
        self.info = info
        self.l = l
        self.p = p

    def daj(self):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        if self.l:
            self.l.daj()
        if self.p:
            self.p.daj()
        print(self.info, end=' ')
```

Zistite, čo vypíše:

```
BS('x',BS('a',BS('c'),BS('d')),BS('b',BS('e'),BS('f'))).daj()
```

5. Prepíšte metódu `daj()` z predchádzajúceho príkladu (4) tak, aby nevypisovala žiadne hodnoty, ale vrátila iterátor, t.j. vygenerovala postupnosť hodnôt.

```
def daj(self):






    ...
```

teda, aby sme pomocou tohto zápisu dostali rovnaký výsledok ako v príklade (4):

```
print(*list(BS('x',BS('a',BS('c'),BS('d')),BS('b',BS('e'),BS('f'))).daj()))
```

6. Odhadnite, ako dlho bude trvať (zložitosť) odstránenie **(log n)** najmenších prvkov z haldy, ktorá má n-prvkov. Na odstraňovanie sa použije metóda `remove_min()`:

```
def remove_min(self):
    if self.is_empty():
        raise Empty('Priority queue is empty.')
    self._swap(0, len(self._data) - 1)     # daj minimálny prvok na koniec
    item = self._data.pop()                # a vyhoď ho zo zoznamu
    self._downheap(0)                      # a oprav koreň
    return (item._key, item._value)
```

teda, ak máme v halde 1024 prvkov, zaujíma nás, koľko bude trvať odstránenie 10 najmenších prvkov.

7. Odhadnite, aká je zložitosť najhoršieho prípadu vkladania `n` dvojíc `(key, value)` do prázdneho asociatívneho poľa, ktoré je realizované pomocou triedy `UnsortedTableMap`.

```
def __setitem__(self, k, v):
    for item in self._table:
        if k == item._key:
            item._value = v
            return
    self._table.append(self._Item(k,v))
```

8. Do koľkých rôznych binárnych vyhľadávacích stromov vieme uložiť kľúče `{1, 2, 3, 4}`? Koľko z nich nespĺňa podmienku **AVL** stromov?

```
.
.
.
```

9. Uvádzali sme tento algoritmus `merge-sort`:

```
def merge_sort(pole):

    def merge(p1, p2):
        result,i1,i2 = [],0,0
        while i1 < len(p1) or i2 < len(p2):
            if i1 < len(p1) and (i2 == len(p2) or p1[i1] < p2[i2]):
                result.append(p1[i1])
                i1 += 1
            else:
                result.append(p2[i2])
                i2 += 1
        return result

    if len(pole) <= 1:
        return pole
    stred = len(pole)//2
    return merge(merge_sort(pole[:stred]),merge_sort(pole[stred:]))
```

Zistite, či je tento algoritmus stabilné triedenie, t.j. ak sme triedili dvojice (kľúč, hodnota) a v pôvodnom poli mali dve dvojice rovnaký kľúč, teda (kľúč,hodnota1) a (kľúč,hodnota2), pričom prvá dvojica bola pred druhou dvojicou, tak aj v utriedenom poli bude pre obe dvojice platiť rovnaký vzťah, teda, že prvá dvojica sa nachádza pred druhou. Ak toto triedenie nie je stabilné, navrhnite 8-prvkové pole, na ktorom sa to potvrdí.

10. Huffmanov kódovací algoritmus pracuje na tomto princípe:

```
Algoritmus Huffman(X):
  Input: ret'azec X dĺžky n
  Output: kódovací strom pre X
Vypočítaj frekvenčnú tabul'ku ft(c) pre každý znak c z X
Initializuj prioritný front Q
for each znak c in X do
    Vytvor binárny strom T s jediným vrcholom a s info c
    Vlož T do Q s kl'účom ft(c)
while len(Q) > 1 do
    (f1,T1) = Q.remove min()      # remove min() vráti dvojicu (kl'úč,hodnota)
    (f2,T2) = Q.remove min()
    Vytvor nový binárny strom T, ktorého l'avý podstrom je T1 a pravý T2
    Vlož T do Q s kl'účom f1+f2
(f,T) = Q.remove min()
return strom T
```

Nakreslite Huffmanov kódovací strom pre reťazec:

```
meno+" programuje v programovacom jazyku python"
```

kde `meno` je vaše meno a priezvisko (malými písmenami bez diakritiky, s jednou medzerou).

Potom pomocou tohto kódovacieho stromu zakódujte reťazec:

```
"peter je prvy"
```

Kódom bude postupnosť 0 a 1: každé písmeno zodpovedá ceste v strome k danému písmenu, pričom ľavý smer v ceste je 0 a pravý 1. Rôzne písmená budú mať rôzne dlhé kódy.

12.2 Test z ADŠ 2015/2016

zložitosť

1. `O(log n)`
2. `O(n)`
3. `O(n log n)`
4. `O(n2)`
5. `O(2n)`
6. `O(n!)`

1. Pre všetky nasledujúce funkcie odhadnite časovú zložitosť: ku každej pripíšte jedno z písmen **A** až **F**.

```python
def fun1(n):
    x = 0
    for i in range(n):
        x += 1
    return x

def fun2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x

def fun3(n):
    if n == 0: return 1
    x = 0
    for i in range(n):
        x += fun3(n-1)
    return x

def fun4(n):
    if n == 0: return 0
    return fun4(n//2) + fun1(n) + fun4(n//2)

def fun5(n):
    x, i = 0, n
    while i > 0:
        x += fun1(i)
        i //= 2
    return x

def fun6(n):
    if n == 0: return 1
    return fun6(n-1) + fun6(n-1)

def fun7(n):
    if n == 1: return 0
    return 1 + fun7(n//2)
```

2. Binárny strom sme reprezentovali nie pomocou smerníkov na ľavý a pravý podstrom ale v jednorozmernom poli:

koreň (jeho dátová časť) sa nachádza v nultom prvku poľa, pre každý vrchol na indexe `i` sú jeho ľavý a pravý syn na indexoch `2*i+1` a `2*i+2` (neexistujúci vrchol má index mimo poľa). Napíšte metódu `postorder`, ktorá vráti postupnosť navštívených vrcholov ako generátor.

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

    def postorder(self):



        ...
```

3. Napíšte metódu `hlbky`, ktorá rekurzívne prejde všetky vrcholy binárneho stromu v poradí preorder a pre každý vrchol vygeneruje (`yield`) dvojicu (data, hĺbka). Metóda hlbky môže mať definovanú svoju vnorenú pomocnú funkciu.

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

    def __init__(self):
        self.root = None

    def hlbky(self):
       ...
       yield node.data, hlbka
```

4. Prioritný front sme realizovali pomocou utriedeného spájaného zoznamu. Dopíšte mu metódy `add` a `remove_min`:

```
class SortedPriorityQueue:
    class Item:
        def __init__(self, key, value, next=None):
            self.key = key
            self.value = value
            self.next = next

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

    def add(self, key, value):
        ...

    def remove_min(self):
        if ...:
            raise Empty('priority queue is empty')
        ...
```

5. Definujte metódy triedu `Set` (množina) pomocou štandardnej triedy `dict`:

```
class Set:
    def __init__(self):
        self.slovnik = dict()

    def __contains__(self, key):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        ...

    def add(self, key):
        ...

    def remove(self, key):
        ...

    def __len__(self):
        ...

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

Telo každej z týchto metód zapíšte len jediným riadkom!

6. Nakreslite všetky binárne vyhľadávacie stromy, v ktorých sú uložené kľúče `{1, 2, 3, 4}` a spĺňajú podmienku **AVL** stromov.

```
.


.
```

7. Pri riešení úlohy mincovka pomocou dynamického programovania sa vytvára tabuľka, ktorá obsahuje minimálne počty mincí, ktoré je treba pre danú sumu (suma je index do tabuľky). Doplňte tabuľku, ak ju vytvárame pre mince `[1, 3, 5, 9]`:

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|  0 |  1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

8. Toto je algoritmus rýchleho triedenia z prednášky – druhý parameter sa zatiaľ nepoužíva:

```
def quick_sort(pole, key=None):
    if len(pole) < 2:
        return pole
    pivot = pole[0]
    mensie = [prvok for prvok in pole if prvok < pivot]
    rovne = [prvok for prvok in pole if prvok == pivot]
    vacsie = [prvok for prvok in pole if prvok > pivot]
    return quick_sort(mensie) + rovne + quick_sort(vacsie)
```

Upravte program tak, aby mal 2. parameter rovnakú úlohu ako v štandardnej funkcii `sorted`. Napr. volanie

```
quick_sort(pole, key=lambda x:abs(x-50))
```

neporovnáva prvky poľa, ale hodnoty, ktoré sa z prvkov vypočítajú pomocou funkcie `key`. Ak má tento parameter hodnotu `None`, triedi sa rovnako ako doteraz.

9. Pri riešenie problému **UNION-FIND** môžeme algoritmus **UNION(p, q)** implementovať pomocou stromov:

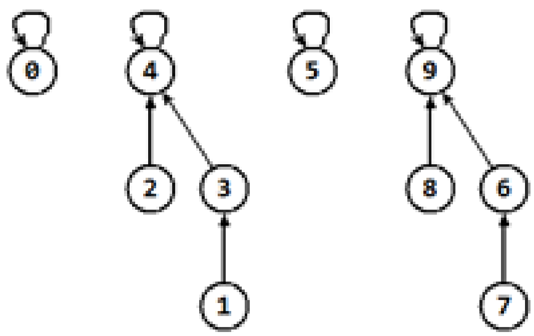
```
def UNION(p, q):
    r1 = FIND(p)
    r2 = FIND(q)
```

(pokračuje na ďalšej strane)

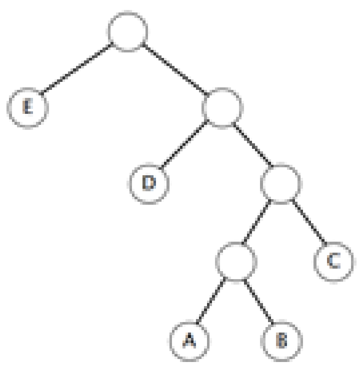
(pokračovanie z predošlej strany)

```
    if r1 != r2:
        if r1.rank > r2.rank:
            r2.parent = r1
        else:
            r1.parent = r2
            if r1.rank == r2.rank:
                r2.rank += 1
```

Nakreslite výsledok, ak pre stromy na obrázku zavoláme `UNION(0, 2)` a `UNION(1, 9)`.



10. Predpokladajme, že tento **huffmanov strom** vznikol zo správy, ktorá sa skladala z písmen `A`, `B`, `C`, `D`, `E`:



Ku každému z nasledovných tvrdení dopíšte, či je „pravdivé", „nepravdivé", alebo „nedá sa rozhodnúť" (napríklad, závisí od konkrétnej správy):

- Frekvencia písmena `A` musí byť menšia ako frekvencia `B`. ....
- Frekvencia písmena `C` musí byť väčšia alebo rovná ako frekvencia `A`. ....
- Frekvencia písmena `D` musí byť väčšia ako frekvencia `A` ....
- Frekvencia písmena `D` musí byť väčšia alebo rovná ako súčet frekvencií `A`, `B` a `C`. ....
- Frekvencia písmena `E` musí byť menšia ako súčet frekvencií `A`, `B` a `C`. ....

12.3 Test z ADŠ 2016/2017

1. Pre všetky nasledujúce algoritmy odhadnite časovú zložitosť. Veľkosť poľa pole označíme ako `n`.

```python
def fun1(n, x=None):
    if x is None:
        n, x = 1, n
    if n < x:
        return fun1(2 * n, x // 2) + 1
    return 0

def fun2(m, n):
    if m == 0:
        return n
    if m > n:
        return fun2(n, m)
    return fun2(m, n-m)

def fun3(pole):
    vysl = i = j = 0
    while i < len(pole):
        if i < len(pole)-1:
            vysl += pole[i] - pole[i+1]
            i += 1
        else:
            i = j = j+1
    return vysl

def fun4(pole):
    vysl = 0
    for i in range(len(pole)):
        j = 1
        while j < len(pole):
            if pole[i] < pole[j]:
                vysl += 1
            j += j
    return vysl

def fun5(pole):
    return sum(i**2 for i in pole)

def fun6(pole):
    if len(pole) == 1:
        return pole[0]
    i = len(pole) // 2
    return fun6(pole[:i]) + fun6(pole[i:])
```

2. Binárny strom sme reprezentovali pomocou smerníkov na ľavý a pravý podstrom. Napíšte metódu `breadth_first`, ktorá vráti postupnosť navštívených vrcholov (algoritmus do šírky) ako generátor.

```python
class BinTree:
    root = None

    class Node:
        def __init__(self, data, left, right):
            self.data, self.left, self.right = data, left, right

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

3. Realizujte zásobník tak, aby obe operácie `push()` aj `pop()` mali konštantnú (t.j. nie len amortizovane konštantnú) časovú zložitosť.

```
class Stack:


    def __init__(self):
        ...

    def push(self, hodnota):
        ...

    def pop(self):      # vyvolá TypeError pri prázdnom zásobníku
        ...

    def is_empty(self):
        ...
```

4. Z rôznych čísel 1 až 10 vytvorte haldu (v koreni s indexom 0. je minimum) v 10-prvkovom poli tak, aby:

   (a) v prvku poľa s indexom 4 bola maximálne možná hodnota

   (b) v prvku poľa s indexom 4 bola minimálne možná hodnota

   Pre obe podúlohy vypíšte dva riadky tabuľky, pričom v prvom bude vytvorená halda a v druhom zrealizujte operáciu `remove_min()`.

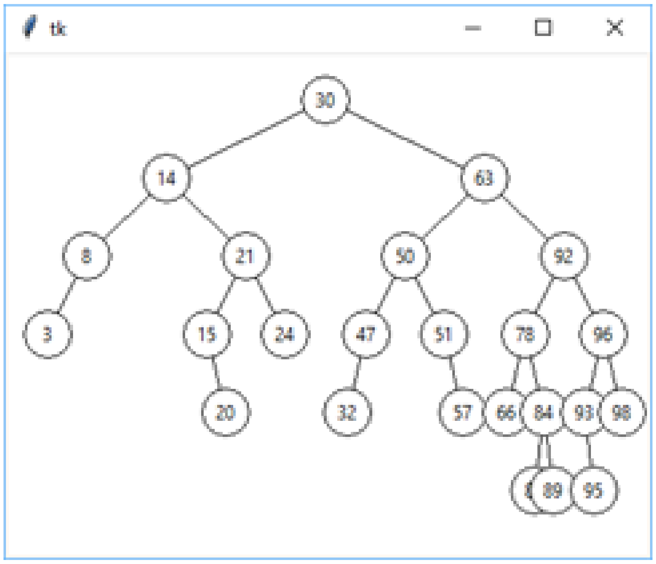
5. Hašovaciu tabuľku vytvárame v 7-prvkovom poli, pričom kolízie riešime pomocou vedierok, ktoré realizujeme jednorozmerným poľom (prvky pridávame na koniec). Kľúčmi sú celé čísla (asociované hodnoty si teraz nevšímame), pričom hašovacia funkcia počíta zvyšok po delení veľkosťou tabuľky. Postupne vložte do tabuľky tieto čísla:

```
17, 36, 76, 76, 9, 52, 40, 24, 29, 26, 68, 7, 89, 76, 80, 59, 59, 2
```

   zapíšte tieto kľúče do príslušných vedierok (na začiatku sú všetky prázdne):

```
   +----+
0  |    |
   +----+
1  |    |
   +----+
2  |    |
   +----+
3  |    |
   +----+
4  |    |
   +----+
5  |    |
   +----+
6  |    |
   +----+
```

6. Máme daný vyhľadávací strom, o ktorom nevieme, či spĺňa podmienku **AVL** stromov. Dopíšte ku každému vrcholu jeho balans (rozdiel výšok podstromov) a vyznačte, ktoré vrcholy nespĺňajú podmienku **AVL**.



7. Ručne zrealizujte nerekurzívny algoritmus `merge_sort` **zdola nahor**. V 16-prvkovom poli máme 16 celočíselných hodnôt (prvý riadok tejto tabuľky). Do druhého riadku zapíšte, ako sa zmení obsah poľa po prvom prechode triedenia, t.j. pri porovnávaní susedných prvkov v dvojiciach. Do tretieho riadku zapíšte, ako sa zmení obsah po druhom prechode triedenia, t.j. po zlúčení prvkov v dvojiciach a vytvorením utriedených štvoríc. Ďalšie dva riadky by mali obsahovať zmeny po 3. a 4. prechode triedenia.

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| 86 | 68 |  3 | 71 |  8 | 90 | 31 | 85 | 45 |  2 | 22 | 73 | 36 | 66 | 25 | 97 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

8. Naprogramujte algoritmus rýchleho hľadania k-teho najmenšieho prvku v neutriedenom poli. Využite rekurzívnu ideu algoritmu `quick_sort`:

```
def select(pole, k):











    ...
```

Zrejme pre `k=0` funkcia vráti minimálny prvok, t.j. `pole[0]`.

9. Ručne vytvorte **LCS**-tabuľku pre reťazce `'babkabraba'` a `'abrakadabra'`:

```
  a b r a k a d a b r a
 0 0 0 0 0 0 0 0 0 0 0 0
b 0
a 0
b 0
k 0
a 0
b 0
r 0
a 0
b 0
a 0
```

Aký najdlhší spoločný podreťazec dostávame z tejto tabuľky?.

10. Z daného poľa hodnôt vytvorte **huffmanov strom** a z neho pre každú z rôznych hodnôt vytvorte kódovaciu tabuľku (pre číselnú hodnotu zodpovedajúcu postupnosť 0 a 1)

```
57, 62, 57, 57, 11, 57, 64, 96, 62, 96, 79, 62, 96, 64, 11, 64, 57, 57, 64, 64
```

12.4 Test z ADŠ 2017/2018

1. Zistiť **zložitosť** funkcií:

```
def f1(pole1, pole2):
    if len(pole2) == 0:
        return 0
    if len(pole2) == 1:
        return int(pole2[0] in pole1)
    stred = len(pole2) // 2
    return f1(pole1, pole2[:stred]) + f1(pole1, pole2[stred:])

def f2(n):
    if n == 0:
        return 1
    return n * f2(n - 1)

def f3(pole):
    i = len(pole)
    while i > 0:
        for j in range(i):
            for k in range(j, 10**5):
                pole[j] += 1
        i -= 2

def f4(n, mem={}):
    if n in mem:
        return mem[n]
    if n < 2:
        res = n
    else:
        res = f4(n-1) + f4(n-2)
    mem[n] = res
    return res

def f5(pole):
    return len(pole) == len(set(pole))
```

2. Vyrobte iterátor z algoritmu prechádzania **stromu do šírky** (po úrovniach). Dopíšte dve vyznačené metódy tak, aby iterátor postupne vracal vrcholy v poradí po úrovniach. Nepoužite pritom `yield`.

```
def breadthfirst(t):
    q = [t.root]
    while q:
        p = q.pop(0)
        print(p.data)
        for node in t.children(p):
            q.append(node)

class Tree:
    root = None
    def __iter__(self):
        ...
    def __next__(self):
        ...
```

3. Pre danú postupnosť hodnôt:

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

vytvorte **haldu** v 15-prvkovom poli. Prvky vkladajte (`add()`) presne v tomto poradí:

```
...
```

Teraz z nej postupne trikrát odstráňte minimálny prvok (`remove_min()`) a zakaždým vypíšte obsah haldy:

```
...
...
...
```

4. Zapíšte metódy triedy **MultiSet** (množina s viacnásobnými výskytmi prvkov), ktorú budete realizovať pomocou pythonovského asociatívneho poľa (`dict`):

```
class MultiSet:
    def __init__(self):
        self.data = {}

    def __contains__(self, key):
        return ...

    def add(self, key):         # pridá do množiny, zapamätá si počet výskytov␣
↪tejto hodnoty
        ...

    def discard(self, key):     # ak je v množine, vyhodí jedeb výskyt
        ...

    def __len__(self):          # počet aj viacnásobných výskytov
        return ...

    def __iter__(self):         # aj viacnásobné výskyty
        yield from ...
```

5. Zapísali sme funkciu `test()`, ktorá pracuje s binárnymi stromami (s vrcholmi `Node`):

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

def test(node):
    if node is None:
        return 0, True
    height1, test1 = test(node.left)
    height2, test2 = test(node.right)
    if not test1 or not test2:
        return 0, False
    return max(height1, height2) + 1, abs(height1 - height2) < 2
```

Nakreslite tento strom a zistite, čo na ňom vráti volanie funkcie `test()`:

```
t = Node(0, Node(1, Node(3)), Node(2, Node(4, None, Node(6)), Node(5, Node(7),␣
↪Node(8))))
>>> test(t)
```

6. Dopíšte:

```
def bucket_sort(pole):
    p = [[] for i in range(100000)]        # inicializácia prázdnych vedierok
    for key, value in pole:
        ________________________________
    res = []
    for key in range(len(p)):
        ________________________________
    return res
```

7. Zapíšte nerekurzívnu verziu `komb1(n, k)`, ktorá najprv skonštruuje dvojrozmernú tabuľku pre `n` riadkov a `k` stĺpcov a na základe nej vráti výsledok:

```
def komb1(n, k):
    # ... vytvorte dvojrozmernú tabuľku veľkosti n x k, kde tab[i][j] = hodnota␣
↪komb(i, j)
    # ... tabuľku vytvorte metódou zdola nahor, t.j. najprv pre malé i a j
    return tab[n][k]
```

8. V tejto štruktúre máme už skonštruovaný **Huffmanov strom**:

```
class Node:
    def __init__(self, data, lavy=None, pravy=None):
        self.data = data
        self.child = [lavy, pravy]
```

Dopíšte funkciu, ktorá rozkóduje zadanú postupnosť 0 a 1:

```
def rozkoduj(root, sequence):
    res = ''
    seq = list(sequence)
    while seq:
        node = root
        while ____________________:

            node = ______________________

        res = _____________________
    return res
```

Napr. pre

```
>>> strom = Node('', Node('a'), Node('', Node('b'), Node('c')))
>>> rozkoduj(strom, (0, 1, 0, 0, 1, 1, 0))
'abaca'
```

Predpokladáme, že strom aj postupnosť sú zadané korektne.

9. Zostavte prefixový strom pre množinu slov:

```
{auto, aula, ano, body, byk, boja, balet, cop, cap, cip, cakan}
```

10. UNION-FIND realizujeme poľom reprezentantov:

```
def FIND(p):
    return tab[p]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
def UNION(p, q):
    r1 = FIND(p)
    r2 = FIND(q)
    if r1 != r2:
        for v in vsetky_prvky_pola:
            if tab[v] == r2:
                tab[v] = r1
```

Zistite, aký bude obsah deväťprvkového poľa po zjednocovaní týchto dvojíc:

```
(6, 2), (4, 7), (6, 3), (5, 8), (3, 2), (1, 5), (5, 6), (0, 4), (8, 3)
```

12.5 Výsledky testu ku skúške

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Bačkovská Simona	0.6	0	0	0	3	-	-	1.1	-	1.1	5.8	
Bagyanszký Marián	1.7	0	0	4.7	5	-	2.3	0	4	2.2	19.9	
Bernát Martin	1.7	-	4	-	4.5	-	-	4.4	-	4.4	19	
Demjen Dávid	1.1	-	2	4.7	5	2.9	-	4.4	4	3.3	27.4	
Freivolt Pavol	1.1	0	1	0	-	-	-	0	-	-	2.1	
Gablíková Júlia	1.7	1	4	6.3	4	2.3	1.1	2.2	4	5.5	32.1	
Gajdošech Lukáš	2.2	-	2.5	6.3	4	2.9	5.2	2.2	4	5.5	34.8	
Groholʼ Daniel	0.6	2	4	-	-	-	-	-	4	4.4	15	
Halgašová Zuzana	0.6	-	4	5.2	5	1.1	0	1.7	4	0	21.6	
Horváth Balázs	0.6	-	4	2.1	3	1.1	-	0.6	3	2.8	17.2	
Hrebeňár Martin	1.7	0	4	1.1	5	2.9	0.6	3.3	2	-	20.6	
Hrušovský Ondrej	1.1	0	4	4.7	5	-	-	4.4	4	2.8	26.0	
Chamula Michal	0.6	2	0	1.1	4	-	2.9	1.1	0	-	11.7	
Janočko Miroslav	0.6	0	0	0.5	4	0	0	1.1	0	0.6	6.8	
Kalla Adam	1.1	-	4	1.6	4	-	-	-	-	-	10.7	
Kán Hugo	0.6	-	1	1.1	4	2.3	-	1.1	-	2.8	12.9	
Knor Michal	1.1	0	3.5	1.1	4	-	4	1.7	4	1.7	21.1	
Kováč Michal	1.7	2	1	5.8	4	-	-	0	4	-	18.5	
Kracina Jakub	1.7	6	4	6.3	5	3.4	0	5.5	4	4.4	40	
Krafčík Marek												
Kubík Jozef	1.7	0	4	1.6	4	2.3	1.1	2.8	4	3.3	24.8	
Kyselica Daniel	1.7	0	4	5.2	5	4.6	3.4	5.5	4	5.5	38.9	
Maďarová Mária	0	1	2.5	2.6	3	2.3	0.6	1.1	4	3.3	20.4	
Michalík Tomáš	0.6	0	0	4.2	-	1.1	-	-	3	2.8	11.7	
Müller Konrád	1.1	-	4	1.1	5	1.1	0.3	2.2	4	3.3	22.1	
Pavlove Filip	1.7	5	4	3.2	4	-	2.9	0	4	2.2	27.0	
Pázmány Michal	1.7	-	4	0.5	5	-	0	1.1	4	0	16.3	
Risnyovszký András	0.6	-	3	3.2	4	2.9	5.2	1.7	4	4.4	29	
Rychtárik Matej	2.2	2	4	4.2	5	2.9	5.8	5.5	4	5.5	40	
Savkova Tamara	1.7	1	4	3.2	3	2.9	0	1.7	4	0	21.5	
Séleš Michal	1.1	-	3	1.1	5	0	0	1.1	4	5.5	20.8	
Senkovičová Soňa	2.8	1	4	4.7	5	2.9	4.6	1.1	4	5.5	35.6	
Silný Boris	0	0	4	4.7	5	1.1	0	2.8	4	5.5	27.1	
Slaninka Lukáš	1.1	-	4	1.1	4	2.9	1.1	2.8	4	3.3	24.3	
Soviš Jakub	1.1	1	0	3.7	5	2.9	0.6	2.2	4	2.8	23.3	
Šabík Oliver	1.1	0	1.5	0.5	1	1.1	-	-	4	0.6	9.8	
Šipula Branislav	0.6	0	4	0.3	3	2.9	0	5	2	2.8	20.6	
Šuba Dávid	1.7	0	1	4.7	3	2.9	4.6	3.9	4	4.4	30.2	
Takács Tomáš	2.2	0	4	5.8	5	2.9	5.6	0.6	4	0	30.1	
Timoranský Ján	0.6	-	1	4.2	5	0	0	1.1	0	0	11.9	
Trizna Adam	1.1	0	4	5.2	3.5	2.9	1.1	1.1	1	1.7	21.6	
Turčeková Klaudia	1.7	0	2	5.8	5	2.9	0.6	1.1	4	3.3	26.4	
Valent Jakub	0.6	0	0	3.7	3	1.1	2.9	-	-	-	11.3	
Velich Tomáš	1.1	0	1	3.7	4	2.9	2.9	4.4	5	4.4	29.4	
Vlčková Monika	0.6	-	3	3.2	3	-	-	-	0	5.5	15.3	
Zdarilek Ján	0.6	1	4	0	-	3.4	0.6	0	4	1.1	14.7	
Zemko Andrej	0	-	0.5	-	3	0	-	0.6	-	-	4.1	

Pokračovanie na ďalšej strane

Tabuľka 1 – pokračovanie z predošlej strany

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Zvarík Emil	1.7	0	0	5.2	5	5.8	4	4.4	4	3.3	33.4	

12.6 Skúška 15.1.2018 - TrieMap

Implementujte **asociatívne pole**, v ktorom sa kľúče ukladajú do **prefixového stromu** (dátová štruktúra `TrieMap`). Kľúčmi asociatívneho poľa budú len znakové reťazce. Asociovaná hodnota sa priradí do príslušného vrcholu stromu (atribút `value`).

Na riešenie úlohy použite deklarácie v súbore `skuska.py` z úlohového servera L.I.S.T. kde

- `__setitem__(key, value)` pre daný kľúč priradí príslušnú hodnotu.
- `__getitem__(key)` pre daný kľúč vráti príslušnú hodnotu, resp. vyvolá chybu `KeyError`, ak taký kľúč neexistuje.
- `__delitem__(key)` vymaže daný kľúč (resp. vyvolá chybu `KeyError`), pričom zo stromu vyhodí tie vrcholy, ktoré už nemajú potomkov (asociatívne pole `child` je prázdne) a ani nenesú informáciu `value`.
- `node_count()` vráti počet vrcholov (typu `Node`) v celom prefixovom strome: prázdny strom má tento počet `0`, strom s jediným kľúčom `''` má tento počet `1`, strom s jediným kľúčom napr. `'a'` má tento počet `2` (aj strom s dvoma kľúčmi `''` a `'a'` má tento počet `2`), atď.
- `__iter__()` vráti **iterátor** všetkých kľúčov v strome, zrejme tieto kľúče môže vrátiť v ľubovoľnom poradí.
- Vnorenú triedu `TrieMap.Node` nemeňte (testovač aj tak použije jeho pôvodnú verziu).
- Uvedomte si, že niektoré vrcholy v strome nereprezentujú kľúč a preto treba v atribúte `Node.value` uložiť nejakú informáciu tak, aby ste vedeli rozpoznať, že vo vrchole nekončí žiaden kľúč; nepoužite na to ale hodnotu `None`, keďže aj toto môže byť hodnotou v asociovanom poli.

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    m = TrieMap()
    for w in 'mama ma emu a ema ma mamu'.split():
        try:
            m[w] = m[w] + 1
        except KeyError:
            m[w] = 1

    print(list(m), m.node_count(), len(m))
    for w in list(m):
        del m[w]
        print(w, m.node_count(), len(m))
```

Výpis potom bude takýto:

```
['ma', 'mama', 'mamu', 'emu', 'ema', 'a'] 11 6
ma 11 5
mama 10 4
mamu 6 3
emu 5 2
ema 2 1
a 0 0
```

Aby ste mohli spúšťať skúškové testy, program musí byť uložený v súbore `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

12.7 Skúška 26.1.2018 - HeapPriorityQueue

Implementujte **prioritný front** pomocou **haldy**, ktorá ale nebude uložená v jednorozmernom poli, ale v dvojrozmernom poli `data` takto:

- prvý riadok tohto dvojrozmerného poľa `data[0]` obsahuje jednoprvkové pole s koreňom haldy
- druhý riadok `data[1]` obsahuje dvojprvkové pole s ľavým a pravým synom koreňa
- tretí riadok `data[2]` obsahuje štvorprvkové pole so všetkými synmi predchádzajúceho riadku
- každý ďalší riadok dvojrozmerného poľa `data` má dvojnásobnú dĺžku oproti predchádzajúcemu riadku
- posledný riadok poľa `data` môže byť kratší ako táto dvojnásobná dĺžka (ale nie je prázdny)

Keby sme zlepili za seba všetky riadky dvojrozmerného poľa `data`, dostali by sme pôvodnú reprezentáciu haldy v jednorozmernom poli.

Uvedomte si, že prvkami dvojrozmerného poľa `data` sú jednorozmerné polia (`data` je typu `list`, ktorého každý prvok je typu `list`), ktorých prvky sú typu `HeapPriorityQueue.Item`. V dátovej štruktúre `HeapPriorityQueue` nepoužívajte žiadne ďalšie atribúty (stavové premenné) okrem `data` a `reverse`. Môžete si zadefinovať ľubovoľné pomocné metódy, napr. `swap`, `has_left` a pod. Je vhodné si pritom každý prvok haldy indexovať nie jedným indexom (ako to bolo v obyčajnej halde, kde pre index `i` bol jeho otcom `i//2` a synovia mali indexy `2*i+1` a `2*i+2`), ale dvomi pre index riadku dvojrozmerného poľa a index v príslušnom riadku. Nemeňte podtriedu `Item` ani inicializáciu `__init__` v triede `HeapPriorityQueue`.

Na riešenie úlohy použite deklarácie v súbore `skuska.py` z úlohového servera L.I.S.T. kde

- parameter `reverse` v inicializácii `__init__` nastaví vytváranie haldy tak, že v koreni bude maximálny prvok a preto metódy `min` a `remove_min` budú namiesto minimálnej hodnoty vracať maximálnu
- metóda `heapify` pôvodný obsah dvojrozmerného poľa `data` nahradí prvkami vstupnej postupnosti `seq`; predpokladajte, že prvkami `seq` sú dvojice (typu `tuple`) hodnôt `(key, value)` a metóda z nich vyrobí prvky `Item`; naprogramujte ju tak, aby využila metódu `heap_down`, t.j. aby mala zložitosť **O(n)**
- dávajte pozor, aby ste kľúče navzájom porovnávali len pomocou relácie menší <, iné realačné operácie nemusia s kľúčmi fungovať

```
class EmptyError(Exception): pass

class HeapPriorityQueue:
    class Item:
        def __init__(self, key, value=None):
            self.key, self.value = key, value

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

        def __repr__(self):
            if self.value is None:
                return repr(self.key)
            return str((self.key, self.value))

    def __init__(self, reverse=False):
        self.data = []
        self.reverse = reverse

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

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def is_empty(self):
        return ...

    def add(self, key, value=None):
        ...

    def min(self):
        return ...     # vrati dvojicu alebo EmptyError

    def remove_min(self):
        return ...     # vrati dvojicu alebo EmptyError

    def heapify(self, seq):
        ...

    def heap_up(self, ...):
        ...

    def heap_down(self, ...):
        ...
```

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    h = HeapPriorityQueue()
    pp = (8, 13, 7, 10, 5, 15, 12, 17, 9, 14, 4, 11, 18, 16, 6)
    for i in pp:
        h.add(i)
    print(*h.data, sep='\n')
    print('==================================')
    p = []
    while not h.is_empty():
        p.append(h.remove_min())
    print(p == sorted(p, key=lambda x: x[0]))
    h = HeapPriorityQueue()
    h.heapify(zip(pp, 'programujeme v pythone'))
    print(*h.data, sep='\n')
```

Výpis potom bude takýto:

```
[4]
[5, 6]
[9, 7, 11, 8]
[17, 13, 14, 10, 15, 18, 16, 12]
==================================
True
[(4, 'm')]
[(5, 'r'), (6, ' ')]
[(9, 'j'), (8, 'p'), (11, 'e'), (7, 'o')]
[(17, 'u'), (10, 'g'), (14, 'e'), (13, 'r'), (15, 'a'), (18, ' '), (16, 'v'), (12, 'm
↪')]
```

Aby ste mohli spúšťať skúškové testy, program musí byť uložený v súbore `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

12.8 Skúška 5.2.2018 - ProbeHashMap

Implementujte metódy **asociatívneho poľa** pomocou **uzavretého hašovania**, v ktorom môžeme nastaviť veľkosť kroku lineárnych pokusov (linear probing). Kľúčmi v asociatívnom poli budú len celé čísla, preto hašovacia funkcia bude z tohto celého čísla počítať len zvyšok po delení veľkosťou tabuľky. Tabuľka sa bude resizovať len v tom prípade, keď počet kľúčov presiahne (bude väčší) polovicu veľkosti tabuľky.

Na riešenie úlohy použite deklarácie v súbore `skuska.py` z úlohového servera L.I.S.T. kde

- inicializácia `__init__` nastaví počiatočnú veľkosť hašovacej tabuľky _table; prvkami tejto tabuľky budú buď `None` (voľný prvok), `_avail` (uvoľnený prvok pomocou `delete`) alebo dvojica typu `_Item`; počítajte s tým, že `step` (krok lineárnych pokusov) bude vždy nepárny a nesúdeliteľný s `capacity` (počiatočná veľkosť tabuľky) , môže byť aj záporný
- metóda `add`, ak daný kľúč existuje, zmení mu v tabuľke hodnotu, ak neexistuje, do tabuľky vloží dvojicu `key`, `value`; ak je v tabuľke viac ako polovica obsadená, tabuľka sa resize na dvojnásobok
- nemeňte `_Item` a `_hash`, môžete dodefinovať ďalšie pomocné metódy
- metóda `__repr__` slúži len na pomoc pri ladení, pri testovaní sa nepoužije

```
class ProbeHashMap:
    _avail = 'avail'

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

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

    def __init__(self, capacity=11, step=1):
        self._table = ...

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

    def valueof(self, key):
        # pre dany key vrati value alebo vyvola KeyError
        raise KeyError

    def add(self, key, value):
        ...

    def delete(self, key):
        # pre dany kluc vyhodi dvojicu key, value alebo vyvola KeyError
        raise KeyError

    def __len__(self):
        # vrati pocet klucov
        return 0

    def __iter__(self):
        # vrati generator vsetkych klucov
        ...

    def __repr__(self):
        res = []
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        for i, item in enumerate(a._table):
            res.append(repr((i, item)))
        return '; '.join(res)
```

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    a = ProbeHashMap()
    for k in 38, 10, 17, 24, 53:
        a.add(k, str(k))
    print(a)
    print('***** pridal som 9 a vyhodil 53')
    a.add(9, 999)
    a.delete(53)
    print(a)
    print('***** pridal som 31')
    a.add(31, 333)
    print(a)
```

Výpis potom bude takýto:

```
(0, None); (1, None); (2, 24:'24'); (3, None); (4, None); (5, 38:'38'); (6, 17:'17');
(7, None); (8, None); (9, 53:'53'); (10, 10:'10')
***** pridal som 9 a vyhodil 53
(0, None); (1, None); (2, 24:'24'); (3, None); (4, None); (5, None); (6, None);
(7, None); (8, None); (9, 9:999); (10, 'avail'); (11, 10:'10'); (12, None);
(13, None); (14, None); (15, None); (16, 38:'38'); (17, 17:'17'); (18, None);
(19, None); (20, None); (21, None)
***** pridal som 31
(0, None); (1, None); (2, 24:'24'); (3, None); (4, None); (5, None); (6, None);
(7, None); (8, None); (9, 9:999); (10, 31:333); (11, 10:'10'); (12, None);
(13, None); (14, None); (15, None); (16, 38:'38'); (17, 17:'17'); (18, None);
(19, None); (20, None); (21, None)
```

Aby ste mohli spúšťať skúškové testy, program musí byť uložený v súbore `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

12.9 Skúška 12.2.2018 - ChainHashMap

Implementujte metódy **asociatívneho poľa** pomocou **otvoreného hašovania**, v ktorom sa kolízie riešia zreťazovaním - ukladaním do vedierok (bucket). Kľúčmi v asociatívnom poli budú len celé čísla, preto hašovacia funkcia bude z tohto celého čísla počítať len zvyšok po delení veľkosťou tabuľky. Tabuľka sa bude resizovať len v tom prípade, keď počet kľúčov presiahne (bude väčší) ako 90% veľkosti tabuľky. Samotné vedierka realizujte jednosmerným spájaným zoznamom (linked list) tak, že v každom prvku `_Item` sa okrem kľúča a hodnoty (`_key` a `_value`) nachádza aj referencia na nasledovný prvok vo vedierku (`_next`).

Na riešenie úlohy použite deklarácie v súbore `skuska.py` z úlohového servera L.I.S.T. kde

- inicializácia `__init__` nastaví počiatočnú veľkosť hašovacej tabuľky _table; prvkami tejto tabuľky budú buď `None` (voľný prvok) alebo objekt typu `_Item`
- metóda `add`, ak daný kľúč existuje, zmení mu v tabuľke hodnotu, ak neexistuje, najprv do tabuľky (do príslušného vedierka) vloží dvojicu `key, value`; ak je teraz tabuľka **viac ako** na 90% obsadená, tabuľka sa resize na dvojnásobok plus 1
- nemeňte `_Item` a `_hash`, môžete dodefinovať ďalšie pomocné metódy
- metóda `print` slúži len na pomoc pri ladení, pri testovaní sa nepoužije

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    a = ChainHashMap(10)
    pole = [(55,'a'),(42,'b'),(15,'c'),(60,'d'),(78,'e'),
            (35,'f'),(22,'g'),(10,'h'),(11,'i'),(15,'j')]
    for key, value in pole:
        a.add(key, value)
    a.print()

    a = ChainHashMap(5)
    d = {}
    for i in 3, 16, 5, 11, 23, 25, 15, 18, 15, 14, 25:
        try:
            a.add(i, a.valueof(i) + 1)
        except KeyError:
            a.add(i, 1)
        d[i] = d.get(i, 0) + 1
    set1 = {(it, a.valueof(it)) for it in a}
    set2 = set(d.items())
    print('=== druhy test', set1 == set2)
    a.print()
```

Výpis potom bude takýto:

```
0 10:'h' -> 60:'d' -> None
1 11:'i' -> None
2 22:'g' -> 42:'b' -> None
3 None
4 None
5 35:'f' -> 15:'j' -> 55:'a' -> None
6 None
7 None
8 78:'e' -> None
9 None
=== druhy test True
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
0 11:1 -> None
1 23:1 -> None
2 None
3 14:1 -> 25:2 -> 3:1 -> None
4 15:2 -> None
5 16:1 -> 5:1 -> None
6 None
7 18:1 -> None
8 None
9 None
10 None
```

Aby ste mohli spúšťať skúškové testy, program musí byť uložený v súbore `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

12.10 1. tréningové zadanie - skúška z 23.1.2017 - tree_sort

Naprogramujte `tree_sort`. Ten funguje tak, že sa najprv vyrobí binárny vyhľadávací strom (**BVS**) a z neho sa pomocou **inorder** získa utriedená postupnosť. Binárny strom BVS sa ale vybuduje trochu inak:

- namiesto reprezentácie vrcholov stromu, v ktorej všetky vrcholy obsahujú atribúty `data`, `left`, `right`, použijeme pre celý strom len dve `n`-prvkové polia (kde `n` je veľkosť poľa)
- v koreni stromu je 0-ty prvok poľa
- `i`-ty vrchol stromu popisuje `i`-ty prvok poľa, pričom ľavý syn (jeho index) je v `left[i]` a pravý je v `right[i]`, koreň má ľavého syna v `left[0]` a pravého v `right[0]`
- ak niektorý syn pre nejaký vrchol neexistuje, príslušný prvok poľa `left`, resp. `right` má hodnotu `None`
- na vytvorenie **BVS** využite metódu `insert(index)`, ktorá na správne miesto pridá ďalší vrchol - asi bude vhodné ju nedefinovať rekurzívne
- keďže v poli sa niektorá hodnota môže nachádzať aj viackrát, v strome sa rovnaké prvky vkladajú do pravého podstromu, teda v ľavom podstrome sú všetky prvky menšie, v pravom sú väčšie alebo rovné

Po skonštruovaní stromu metódou `insert()`, teda po skonštruovaní dvoch polí `left` a `right`, algoritmus triedenia zavolá metódu `inorder()`, ktorá postupne vygeneruje utriedenú postupnosť prvkov pôvodného poľa. Metóda `inorder()` bude prvky poľa postupne vracať pomocou `yield`. Tiež bude vhodné ju nedefinovať rekurzívne, inak by mohla pre väčšie pole spadnúť na pretečení rekurzie.

Je zrejmé, že pôvodné triedené pole sa pri volaní `tree_sort` nesmie zmeniť. Okrem dvoch pomocných polí `left` a `right` nepoužívajte ďalšie pomocné polia. Zrejme si v inicializácii triedy `BVS` vytvoríte referenciu na pôvodné pole.

Použite tieto deklarácie:

```
class BVS:
    def __init__(self, pole):
        self.pole = pole
        self.left = ...
        self.right = ...
        ...

    def insert(self, index):
        ...

    def inorder(self, reverse=False):
        ...

def tree_sort(pole, reverse=False):
    tree = BVS(pole)
    for i in range(1, len(pole)):
        tree.insert(i)
    return tree.inorder(reverse)
```

kde

- metóda `insert(index)`: vloží do stromu prvok s daným indexom, teda v skutočnosti bude modifikovať len polia `left` a `right`;
- metóda `inorder(reverse)`: vráti ako generátor inorder-postupnosť prvkov poľa; parameter `reverse` (rovnako ako pre štandardnú funkciu `sorted()`) označuje, že metóda vygeneruje postupnosť v opačnom poradí prvkov
- funkciu `tree_sort()` nemá zmysel modifikovať, lebo spúšťať sa bude táto pôvodná verzia

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    p = (27, 25, 34, 23, 25, 31, 28, 21)
    print(*tree_sort(p))
    print(*tree_sort(p, reverse=True))
```

Výpis potom bude:

```
21 23 25 25 27 28 31 34
34 31 28 27 25 25 23 21
```

V tomto prípade pre pole:

```
p = (27, 25, 34, 23, 25, 31, 28, 21)
```

vytvorená trieda BVS bude obsahovať tieto 2 pomocné polia:

```
left = [1, 3, 5, 7, None, 6, None, None]
right = [2, 4, None, None, None, None, None, None]
```

Toto označuje:

- koreň (index 0) s hodnotou 27 má ľavého syna vrchol 1 (s hodnotou 25) a pravého syna vrchol 2 (s hodnotou 34)
- vrchol na indexe 1 s hodnotou 25 má ľavého syna vrchol 3 (s hodnotou 23) a pravého syna vrchol 4 (s hodnotou 25)
- všetky zvyšné vrcholy majú len ľavého syna
- vrchol s indexom 2 (hodnota 34) má ľavého syna vrchol 5 (hodnota 31)
- vrchol s indexom 3 (hodnota 23) má ľavého syna vrchol 7 (hodnota 21)
- ...

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

12.11 2. tréningové zadanie - skúška z 30.1.2017 - TrieMap

Implementujte asociatívne pole, v ktorom sa kľúče ukladajú do lexikografického stromu (dátová štruktúra `Trie`). Kľúčmi asociatívneho poľa budú len nezáporné celé čísla a preto sa do štruktúry `Trie` tieto kľúče budú rozoberať na cifry. Napr. kľúč `213` sa do `Trie` vloží takto: najprv sa zoberie posledná cifra `3`; táto označuje, že z koreňa stromu sa pokračuje tretím podstromom (podstrom s indexom `3`, t.j. `node.next[3]`) ale s už hodnotou kľúča `21`; v tomto podstrome sa pokračuje podstromom `1` (posledná cifra `21`) s novou hodnotou kľúča `2`; znovu sa pokračuje podstromom `2` a teraz už s hodnotou kľúča `0` - tu sa vnáranie do podstromov končí. Koreň tohto podstromu bude obsahovať samotnú asociovanú hodnotu. Preto metóda `t.add(213,'abc')` pre lexikografický strom `t` v skutočnosti zrealizuje: `t.root.next[3].next[1].next[2].value='abc'`. Uvedomte si, že takýto lexikografický strom je v skutočnosti postfixový strom a nie prefixový.

Na riešenie úlohy použite tieto deklarácie (kompletné definície v súbore `skuska.py` si stiahnite z úlohového servera L.I.S.T.):

```
class Trie:

    class Node:
        def __init__(self, value):
            self.value = value
            self.next = [None] * 10

    def __init__(self):
        self.root = None

    def add(self, key, value):
        ...

    def node_count(self):
        ...
    ...

class TrieMap:
    def __init__(self):
        self.trie = Trie()

    def __setitem__(self, key, value):
        ...
    ...
```

kde

- `Trie.valueof(key)` pre daný kľúč vráti príslušnú hodnotu, resp. vyvolá chybu `KeyError`, ak taký kľúč neexistuje.
- `Trie.delete(key)` vymaže daný kľúč (resp. vyvolá chybu `KeyError`), pričom zo stromu vyhodí tie vrcholy, ktoré už nemajú potomkov (pole `next` obsahuje len samé `None`) a ani nenesú informáciu `value`.
- `Trie.node_count()` vráti počet vrcholov v celom lexikografickom strome: prázdny strom má tento počet `0`, strom s jediným kľúčom `0` má tento počet `1`, strom s jediným kľúčom napr. `1` má tento počet `2` (aj strom s dvoma kľúčmi `0` a `1` má tento počet `2`), atď.
- `Trie.__iter__()` vráti **iterátor** všetkých kľúčov v strome, zrejme tieto kľúče môže vrátiť v ľubovoľnom poradí.
- Vnorenú triedu `Trie.Node` nemeňte (testovač aj tak použije jeho pôvodnú verziu).
- Uvedomte si, že niektoré vrcholy v strome nereprezentujú kľúč a preto treba v atribúte `Node.value` uložiť nejakú informáciu tak, aby ste vedeli rozpoznať, že vo vrchole nekončí žiaden kľúč; nepoužite na to ale hodnotu

`None`, keďže aj toto môže byť hodnotou v asociovanom poli.

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    d = TrieMap()
    d[123] = 'prvy'
    d[132] = 'druhy'
    d[213] = 'treti'
    d[231] = 'stvrty'
    d[312] = 'piaty'
    d[321] = 'siesty'
    print('pocet vrcholov =', d.trie.node_count())
    print('pocet klucov =', len(d))
    for key in d:
        print(key, d[key])
    for key in list(d):
        del d[key]
        print('po vyhodeni', key, 'je pocet vrcholov =', d.trie.node_count())
```

Výpis potom bude takýto:

```
pocet vrcholov = 16
pocet klucov = 6
321 siesty
231 stvrty
312 piaty
132 druhy
213 treti
123 prvy
po vyhodeni 321 je pocet vrcholov = 14
po vyhodeni 231 je pocet vrcholov = 11
po vyhodeni 312 je pocet vrcholov = 9
po vyhodeni 132 je pocet vrcholov = 6
po vyhodeni 213 je pocet vrcholov = 4
po vyhodeni 123 je pocet vrcholov = 0
```

Aby ste mohli spúšťať skúškové testy, program musí byť uložený v súbore `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

12.12 3. tréningové zadanie - skúška z 6.2.2017 - triedenie pomocou prioritného frontu

Implementujte prioritný front pomocou binárneho vyhľadávacieho stromu: vkladanie nového prvku (metóda `add`) je štandardným vložením do **BVS**, vyhodenie minimálnej hodnoty (metóda `remove_min`) je štandardným hľadaním minimálneho prvku a jeho vyhodením z **BVS**. Tento prioritný front môžeme využiť na triedenie bežným spôsobom: najprv sa všetky prvky triedeného poľa postupne presunú do prioritného frontu a potom pomocou metódy `remove_min` sa postupne všetky vrátia späť do poľa ale už v utriedenom poradí.

Binárny strom **BVS** sa pre účely prioritného frontu musí trochu prispôsobiť:

- keďže sa do frontu môže vložiť viac dvojíc (kľúč, hodnota) s rovnakým kľúčom (možno aj s rovnakou hodnotou), musíme zabezpečiť správne fungovanie **BVS**;
- vo vrcholoch **BVS** (typu `Node`) budú všetky kľúče rôzne, ale prislúchajúce hodnoty sa v každom vrchole uchovávajú jednosmernými spájanými zoznamami: na začiatku každého zoznamu sa nachádza hodnota, ktorá bola s daným kľúčom vložená do stromu ako prvá, za tým nasledujú všetky ďalšie hodnoty s daným kľúčom v poradí, ako sa vkladali do stromu;
- prvky jednosmerných spájaných zoznamov realizujte dátovou štruktúrou `Item`, v ktorej v atribúte `next` budete ukladať nasledovný prvok zoznamu;
- keď sa bude zo stromu odoberať nejaká hodnota, vždy sa zoberie hodnota zo začiatku zoznamu; keď sa takto príslušný zoznam vyprázdni, bude treba odstrániť samotný vrchol **BVS**;
- uvedomte si, že z tohto **BVS** sa budú odstraňovať len vrcholy s minimálnym kľúčom, teda sú to len také vrcholy, ktoré nemajú ľavého syna.

Použite tieto deklarácie (z úlohového servera L.I.S.T. si stiahnite kompletnú verziu `skuska.py`):

```
class EmptyError(Exception): pass

class Item:
    def __init__(self, value):
        self.value = value
        self.next = None

class Node:
    def __init__(self, key, value):
        self.key = key
        self.value_list = Item(value)
        self.left = self.right = None

class PriorityQueue:
    def __init__(self):
        self.root = None

    def add(self, key, value):
        ...

    def remove_min(self):
        return None, None

    ...

def pq_sort(pole):
    pq = PriorityQueue()
    for key, value in pole:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        pq.add(key, value)
    for i in range(len(pole)):
        pole[i] = pq.remove_min()
```

kde

- definície tried `EmptyError`, `Item` a `Node` vo svojom riešení nemeňte – testovač aj tak použije presne túto definíciu týchto tried;
- atribút `root` v triede `PriorityQueue` musí obsahovať koreň **BVS**, pre prázdny strom musí obsahovať `None` (testovač to kontroluje);
- metódy `add(key,value)`, `min()`, `remove_min()` a `__len__()` by sa mali správať ako bežné metódy prioritného frontu, t.j. mali by modifikovať **BVS** (s koreňom v `root`), prípadne poskytovať o ňom požadované informácie; metódy `min()` a `remove_min()` vrátia buď dvojicu (typu `tuple`) pre kľúč a hodnotu alebo vyvolajú výnimku `EmptyError`;
- metóda `tree_height()` vráti momentálnu výšku celého **BVS** (výška prázdneho stromu je -1);
- ani funkciu `pq_sort()` nemá zmysel modifikovať, lebo spúšťať sa bude táto pôvodná verzia.

Váš program môžete testovať napr. takto:

```
if __name__ == '__main__':
    pole = [(2,'prvy'),(1,'druhy'),(3,3.14),(2,444),(5,'piaty')]
##    pq_sort(pole)
    pq = PriorityQueue()
    for key, value in pole:
        pq.add(key, value)
    print('pocet:', len(pq))
    print('vyska:', pq.tree_height())
    print('min:', pq.min())
    for i in range(len(pole)):
        pole[i] = pq.remove_min()
        print(pole[i], len(pq), pq.tree_height())
    print(pq.remove_min())
```

Výpis potom bude:

```
pocet: 5
vyska: 2
min: (1, 'druhy')
(1, 'druhy') 4 2
(2, 'prvy') 3 2
(2, 444) 2 1
(3, 3.14) 1 0
(5, 'piaty') 0 -1
...
EmptyError
```

V tomto príklade sa zostrojí **BVS**, ktorý má len 4 vrcholy, pričom vo vrchole s kľúčom `2` sú uložené 2 hodnoty v poradí najprv `'prvy'` a potom `444`. Vo všetkých ostatných vrcholoch je len po jednej hodnote. Vo vašom riešení nepoužívajte funkcie `sort` ani `sorted`.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`.

Riešenie odovzdajte na úlohový server L.I.S.T..

12.13 4. tréningové zadanie - skúška z 30.5.2016 - množina AVL stromov

Definujte všetky metódy tried `BST` a `AVLSet`, pomocou ktorých sa bude dať vyriešiť takáto úloha:

- Zistite všetky binárne vyhľadávacie stromy, v ktorých sú uložené kľúče {1,2,3,4} a spĺňajú podmienku **AVL** stromov.

Trieda `BST `` (binary search tree) okrem základnej metódy ``insert` (vloží do stromu novú hodnotu na správne miesto) má aj ďalšie pomocné metódy:

- `preorder()` vráti n-ticu (`tuple`) všetkých kľúčov v strome v poradí **preorder**
  - uvedomte si, že toto poradie jednoznačne určuje tvar stromu
- `avl()` zistí, či daný binárny vyhľadávací strom spĺňa podmienku AVL stromov: t.j. pre každý vrchol platí, že výšky ľavého a pravého podstromu sa líšia maximálne o 1; vráti `True` alebo `False`
- `__eq__(other)` zistí, či daný binárny vyhľadávací strom je presne rovnaký ako nejaký iný (má rovnaký tvar aj hodnoty vo vrcholoch); vráti `True` alebo `False`
- `__repr__()` vráti znakový reťazec reprezentácie stromu, ktorý bude reťazcovou reprezentáciou preorderu
- `__hash__()` vráti celé číslo, ktoré reprezentuje celý strom: pre rôzne stromy vráti rôzne čísla, pre rovnaké vráti rovnaké číslo (využite preorder)

Vďaka dvom magickým metódam (`__eq__` a `__hash__`) Python dovolí z objektov `BST` vytvárať normálnu pythonovskú množinu.

Trieda `AVLSet` slúži na vytváranie množiny rôznych binárnych vyhľadávacích stromov, ktoré spĺňajú podmienku **AVL**. Naprogramujte tieto metódy:

- `add(seq)` - zo zadanej postupnosti `seq` vytvorí binárny vyhľadávací strom (objekt `BST`) a ak to spĺňa podmienku `avl()`, zaradí ho do množiny (atribút `self.set`); metóda testuje, či je to **AVL** po každom pridaní ďalšej hodnoty z postupnosti `seq` do stromu (hoci výsledný strom by mohol byť AVL, ale už pri jeho vytváraní môže vzniknúť strom, ktorý by nebol AVL, preto sa takýto strom do množiny zaraďovať nebude)
- `all(seq)` - z danej postupnosti hodnôt vygeneruje všetky permutácie prvkov (zrejme ich bude faktoriál počtu prvkov) a z každej takejto permutácie sa bude snažiť vytvoriť **AVL** strom - použije metódu `self.add()`

Ak by sme triedy otestovali týmito volaniami metód:

```
if __name__ == '__main__':
    t1 = BST()
    for i in (1, 2, 3, 4):
        t1.insert(i)
    print(t1)
    print('avl =', t1.avl())
    t2 = BST()
    for i in (2, 4, 3, 1):
        t2.insert(i)
    print(t2)
    print('avl =', t2.avl())
    t3 = BST()
    for i in (2, 1, 4, 3):
        t3.insert(i)
    print('eq =', t2 == t3)
    mn = AVLSet()
    mn.add((1, 2, 3, 4))
    mn.add((2, 4, 3, 1))
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    mn.add((2, 1, 4, 3))
    mn.add((2, 4, 1, 3))
    print('set:')
    for t in mn:
        print(t)
    print('all:')
    mn = AVLSet()
    mn.all((1, 2, 3, 4))
    print(*mn, sep='\n')
    mn = AVLSet()
    mn.all(range(5))
    print('all range(5) =', len(mn))
```

dostaneme tento výstup:

```
(1, 2, 3, 4)
avl = False
(2, 1, 4, 3)
avl = True
eq = True
set:
(2, 1, 4, 3)
all:
(3, 2, 1, 4)
(2, 1, 3, 4)
(3, 1, 2, 4)
(2, 1, 4, 3)
all range(5) = 6
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Mali by ste dodržať tieto vlastnosti programu:

- Nemeňte me ná už zadaných atribútov (metód a premenných).
- Do existujúcich tried môžete pridávať vlastné atribúty a metódy.
- Pri testovaní vášho riešenia sa budú kontrolovať aj vami vytvorené binárne vyhľadávacie stromy (s koreňom v atribúte `BST.root`).

Aby ste mohli spúšťať skúškové testy, riešenie (bez ďalších súborov) odovzdajte na úlohový server L.I.S.T.. Testy postupne preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie preruší a ďalšie časti sa netestujú:

- 20% bodov za vytvorenie **BST** stromu a metódu `repr()`
- 20% bodov za testovanie, či má **BST** strom **AVL** vlastnosť
- 20% bodov za funkčnosť metód `__eq__()` a `__hash__()`
- 20% bodov za metódu `add()` v triede `AVLSet`
- 20% bodov za metódu `all()` v triede `AVLSet`

12.14 5. tréningové zadanie - skúška z 1.6.2016 - kostra grafu

Definujte všetky metódy triedy `Graph`, pomocou ktorých sa bude dať nájsť **kostra grafu** (Minimum Spanning Tree):

- Predpokladáme, že graf je neorientovaný a súvislý. Potom kostra grafu je taká najmenšia podmnožina všetkých hrán, pre ktorú je graf stále súvislý a pritom súčet ohodnotení hrán je najmenší možný.
- Ak má graf `n` vrcholov, potom kostra grafu má `n-1` hrán.

Kostru grafu by sme mohli motivovať takouto úlohou:

- Vedecký park sa skladá z `n` budov pričom niektoré z nich sú priamo spojené sieťovým káblom. Spojenia sú navrhnuté tak, že medzi každými dvoma budovami existuje buď priame alebo nepriame spojenie cez niekoľko ďalších budov (graf je súvislý).
- Zistilo sa, že niektoré spojenia by sme mohli ušetriť a stále by ostal graf súvislý, teda potom existuje aspoň nepriame spojenie medzi každými dvoma budovami.
- Keďže poznáme cenu sieťových káblov medzi spojenými budovami, mohli by sme navrhnúť takú sieť (podsieť pôvodnej), ktorá bude najlacnejšia, a preto takto získame najväčšiu úsporu.
- Teda hľadáme takú podmnožinu existujúcich hrán grafu, ktorá je najmenšia možná aby bol graf ešte súvislý a pritom má najmenší súčet ohodnotení hrán.

Kostru grafu budete hľadať pomocou **Kruskalovho algorimu**, ktorý využíva ideu **UNION-FIND**:

1. algoritmus si buduje sústavu disjunktných množín vrcholov, do ktorej postupne pridáva niektoré hrany (teda spája niektoré disjunktné množiny), až kým nevznikne len jediná množina, teda graf aj s touto menšou množinou hrán bude súvislý - uvedomte si, že disjunktné množiny sú v tomto prípade komponenty grafu
2. na začiatku bude každý vrchol tvoriť samostatnú množinu (operácia `make_set(v)`)
3. potom usporiada všetky hrany do rastúceho usporiadania podľa váh (môžete využiť prioritný front (s kľúčom váha hrany) alebo len usporiadať pole hrán podľa ich váh)
4. ďalej opakuje, kým už nemá vybraných `n-1` hrán:
   - vyberie hranu s najmenšou váhou (napr. `remove_min` z prioritného frontu), nech je to `(v1,v2)`
   - ak tieto dva vrcholy ešte nie sú spojené (teda `find(v1)!=find(v2)`), zaradí túto hranu do výslednej množiny hrán a označí, že teraz sú už oba vrcholy vtom istom komponente (`union(v1,v2)`)
5. takto má hotovú minimálnu množinu hrán, ktorá vytvorila hľadanú kostru grafu

Trieda `Graph` je dátová štruktúra pre **neorientovaný ohodnotený graf**. Vrcholmi sú objekty typu `Vertex` - túto triedu už nemeňte, je už hotová. Trieda `Graph` bude obsahovať tieto metódy:

- `make_set(v)`, `find(v)`, `union(v1,v2)` - sú tri metódy, ktoré zabezpečia riešenie **union-find** pre vrcholy grafu
  - použite riešenie, v ktorom do každého vrcholu pridáte dva atribúty: `parent` (referenciu na predchodcu v strome jednej skupiny) a `rank` na momentálnu výšku stromu
  - `make_set(v)` vytvorí strom len s koreňom, `parent` odkazuje na samého seba, `rank` je 1
  - `find(v)` vráti koreň stromu (postupuje po smerníkoch `parent`)
  - `union(v1,v2)` - spojí dva stromy tak, že menšiemu z nich (s menším `rank`) koreňovému vrcholu zmení `parent` na koreň väčšieho stromu, ak majú rovnaký `rank`, tak ich spojí tiež, ale vtedy aj zvýši `rank` o 1
- `insert_vertex(x)` - pridá do grafu nový vrchol a pomocou **union-find** vytvorí novú množinu (strom len so samotným koreňom, je to komponent sjediným vrcholom)

- `insert_edge(v1, v2, weight)` - vytvorí novú hranu: vloží ju do `self.edges` a tiež spojí pomocou **union-find** obe množiny (oba komponenty)
- `is_connected()` - zistí, či je graf súvislý (vráti `True` alebo `False`), urobte totak, že prejdete všetky vrcholy grafu a zistíte, či sú všetky v tom istom komponente (majú rovnakú hodnotu `find()`)
- `reaches(v1,v2)` - zistí, či sú dva vrcholy v tej istej množine (komponente), t.j. majú rovnaký `find()`

Vďaka magickej metóde `__hash__` v triede `Vertex` Python dovolí aj objekty tohto typu vkladať donormálnych pythonovských množín a korektne s nimi pracovať.

Ak by sme triedu otestovali týmito volaniami metód:

```
if __name__ == '__main__':
    g = Graph()
    v1 = g.insert_vertex('A')
    v2 = g.insert_vertex('B')
    v3 = g.insert_vertex('C')
    v4 = g.insert_vertex('D')
    v5 = g.insert_vertex('E')
    print('vrcholy =', g.vertices)
    print('da sa z v3 do v5 =', g.reaches(v3, v5))
    print('da sa z v2 do v4 =', g.reaches(v2, v4))
    g.insert_edge(v1, v2, 2)
    g.insert_edge(v3, v1, 3)
    g.insert_edge(v2, v5, 3)
    g.insert_edge(v1, v5, 2)
    print('hrany =', g.edges)
    print('je suvisly =', g.is_connected())
    print('da sa z v3 do v5 =', g.reaches(v3, v5))
    print('da sa z v2 do v4 =', g.reaches(v2, v4))
    g.insert_edge(v4, v3, 3)
    print('je suvisly =', g.is_connected())
    print('da sa z v2 do v4 =', g.reaches(v2, v4))
    print('kostra grafu =', g.get_MST())
```

dostaneme tento výstup:

```
vrcholy = {'D', 'E', 'B', 'A', 'C'}
da sa z v3 do v5 = False
da sa z v2 do v4 = False
hrany = {('C', 'A'): 3, ('B', 'E'): 3, ('A', 'B'): 2, ('A', 'E'): 2}
je suvisly = False
da sa z v3 do v5 = True
da sa z v2 do v4 = False
je suvisly = True
da sa z v2 do v4 = True
kostra grafu = {('C', 'A'): 3, ('D', 'C'): 3, ('A', 'B'): 2, ('A', 'E'): 2}
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Mali by ste dodržať tieto vlastnosti programu:

- Nemeňte mená už zadaných atribútov (metód a premenných).
- Do existujúcich tried môžete pridávať vlastné atribúty a metódy (môžete si vytvoriť aj vlastnú triedu).
- Pri testovaní vášho riešenia sa bude kontrolovať aj štruktúra vami vytvoreného grafu a tiež vytvorená informácia pre **union-find**.
- Ak budete chcieť vytvárať vlastný prioritný front, môžete využiť modul `heapq`.

Aby ste mohli spúšťať skúškové testy, riešenie (bez ďalších súborov) odovzdajte na úlohový server L.I.S.T.. Testy postupne preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie preruší a ďalšie časti sa netestujú:

- 10% bodov za vytvorenie grafu
- 20% bodov za množiny **union-find**
- 20% bodov za metódy `is_connected()` a `reaches()`
- 50% bodov za algoritmus hľadania kostry grafu pre rôzne veľké grafy (pozrite si testovacie dáta v súboroch `'subor01.txt'`, `'subor02.txt'`, `'subor03.txt'`, ..., ktoré používa testovač)

12.15 6. tréningové zadanie - skúška z 13.6.2016 - minimálna sieť

V Kocúrkove dostali európsky grant, aby si zaviedli internet do všetkých domov v obci. Financie ale dostanú len vtedy, keď to zabezpečia najmenšou možnou dĺžkou použitých optických káblov. Ďalšou podmienkou komisie bolo to, aby káble ťahali len priamo medzi domami, teda tak, že rozvetvovať sa budú len v domoch a nie mimo nich. Pre fungovanie internetu bude stačiť, keď bude existovať aspoň nepriame spojenie medzi každými dvoma domami (jeden z nich je obecný úrad, ktorý má prepojenie na internetového poskytovateľa).

Úlohu budete riešiť takto:

- uložíte si polohy `(x, y)` všetkých domov v obci (dve celé čísla) - prečítate ich zo zadaného súboru
- všetky dvojice domov potom uložíte do prioritného frontu, pričom kľúčom bude ich vzdialenosť
  - uvedomte si, že ak je v obci `n` domov, tak tento front musí obsahovať presne `n*(n-1)/2` takýchto dvojíc (graf je neorientovaný, každá hrana bude vo fronte len raz)
- postupne budete z tohto frontu vyberať dvojicu s najmenšou vzdialenosťou a ak pre tieto dva domy ešte neexistuje ani nepriame prepojenie, tak ich zaradíte do výsledného zoznamu internetovej siete
  - na zisťovanie, či sú dva domy už zosieťované, použijete ideu **UNION-FIND** (teda zistíte, či sú oba domy ešte v rôznych komponentoch)
- ak bolo v obci `n` domov, tak zrejme stačí `n-1` prepojení dvojíc domov, aby bola celá obec zosieťovaná (uvedomte si, že toto je *Kruskalov algoritmus* pre nájdenie **kostry grafu**).

Na prioritný front môžete (nemusíte) využiť modul `heapq`.

Trieda `Kocurkovo` bude obsahovať tri podtriedy a tieto metódy:

- `Dom.__init__(...)` inicializuje **union-find** vrchol, t.j. vytvorí strom len s koreňom, `parent` odkazuje na samého seba, `rank` je 1
- `Dom.find()` vráti koreň **union-find** stromu (postupuje po smerníkoch `parent`)
- `Kocurkovo.union(...)` spojí dva **union-find** stromy tak, že menšiemu z nich (s menším `rank`) koreňovému vrcholu zmení `parent` na koreň väčšieho stromu, ak majú rovnaký `rank`, tak ich spojí tiež, ale vtedy aj zvýši `rank` o 1
- `Spoj.__init__(...)` vytvorí hranu grafu, t.j. dvojicu dvoch vrcholov (domov), pričom v atribúte `key` bude dĺžka hrany, teda vzdialenosť dvoch domov
- `Kocurkovo.__init__(...)` prečíta súbor a vytvorí z neho atribút `pole` (typu `list`), ktorý bude obsahovať všetky domy (objekty typu `Dom`)
- `Kocurkovo.prioritny_front()` vytvorí prioritný front (ako haldu) so všetkých hrán grafu - prvkami budú objekty typu `Spoj`
  - uvedomte si, že v prioritnom fronte sa porovnávajú len kľúče
- `min_siet()` vytvorí kostru grafu - vráti dvojicu hodnôt: jej dĺžku (súčet dĺžok hrán kostry) a zoznam hrán ako pole (`list`) dvojíc súradníc domov
  - parametrom je prioritný front, ktorý sa vytvoril metódou `prioritny_front()`
- do triedy `Spoj` môžete dodefinovať metódu `Spoj.__lt__(self, iny)`, ktorá zabezpečí, že prioritný front pre porovnávanie prvkov použije len kľúče, teda táto metóda vráti `True`, keď `self.key` je menší ako `iny.key`

Ak by sme triedu otestovali týmito volaniami metód:

```
if __name__ == '__main__':
    k = Kocurkovo('subor1.txt')
    print('pocet domov =', len(k.pole))
    pq = k.prioritny_front()
    print('dlzka frontu =', len(pq.pole))
    d, r = k.min_siet(pq)
    print('dlzka kostry =', d)
    for d1, d2 in r:
        print(d1, '-', d2)
```

dostaneme tento výstup:

```
pocet domov = 8
dlzka frontu = 28
dlzka kostry = 654.1493569838469
(127, 66) - (158, 23)
(318, 178) - (278, 232)
(156, 135) - (127, 66)
(266, 309) - (278, 232)
(151, 317) - (43, 338)
(266, 309) - (151, 317)
(156, 135) - (278, 232)
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Mali by ste dodržať tieto vlastnosti programu:

- Nemeňte mená už zadaných atribútov (metód a premenných).
- Do existujúcich tried môžete pridávať vlastné atribúty a metódy (môžete si vytvoriť aj vlastnú triedu).

Aby ste mohli spúšťať skúškové testy, riešenie (bez ďalších súborov) odovzdajte na úlohový server L.I.S.T.. Testy postupne preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie preruší a ďalšie časti sa netestujú:

- 10% bodov za prečítanie súboru
- 30% bodov za vytvorenie prioritného frontu
- 60% bodov za algoritmus hľadania kostry grafu pre rôzne veľké grafy (pozrite si testovacie dáta v súboroch `'subor01.txt'`, `'subor02.txt'`, `'subor03.txt'`, ..., ktoré používa testovač)

12.16 7. tréningové zadanie - skúška z 19.1.2015 - huffmanovo kódovanie

Definujte všetky metódy triedy `Huffman`, pomocou ktorej sa budú dať zakódovať a rozkódovať postupnosti údajov (reťazce a polia) **Huffmanovým** algoritmom. Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Mali by ste dodržať tieto vlastnosti programu:

- Nemeňte mená už zadaných atribútov (metódy a premenné).
- Môžete definovať ďalšie pomocné triedy a aj pridávať vlastné atribúty do existujúcich tried.
- Pri testovaní vášho riešenia sa bude kontrolovať aj vami vytvorený **Huffmanov** kódovací strom (v atribúte `Huffman.strom`).

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez ďalších súborov) odovzdajte na úlohový server L.I.S.T..

12.17 Copyright

Vytvoril Andrej Blaho.

Materiály k predmetu **Algoritmy a dátové štruktúry** 1-AIN-210/15 na FMFI UK.

ISBN PDF verzie 978-80-8147-085-1

ISBN webovej verzie 978-80-8147-086-8

Môžete si stiahnuť PDF verziu kompletných materiálov Algoritmy a dátové štruktúry.

12.18 Priebeh semestra

semestrálny projekt

v priebehu semestra dostávate možnosť riešiť jeden semestrálny projekt:

- tému si zvolíte sami, nemusí byť naprogramovaná v Pythone, ale mala by obsahovať nejaký algoritmus alebo dátovú štruktúru z tohto semestra, resp. je to náročnejšia verzia nášho semestra
- tematicky sú najvhodnejšie tieto okruhy:
  - generátory na stromoch
  - haldy
  - hashovacie tabuľky
  - vyvažované vyhľadávacie stromy
  - efektívne algoritmy triedenia
  - dynamické programovanie
  - algoritmy vyhľadávania v reťazcoch typu KMP a LCS
  - Huffmanovo kódovanie
  - trie - lexikografické stromy
  - union-find
  - netriviálne algoritmy na stromoch
- za načas odovzdaný projekt môžete získať maximálne 10 bodov

---

Test ku skúške

V pondelok 18. decembra 2017 bol test ku skúške *Test z ADŠ 2017/2018*. Pozri *Výsledky testu ku skúške*.

Skúšky v zimnom semestri

Riadne termíny skúšok boli: *15.1.2018* a *26.1.2017*

- praktická časť skúšky bude prebiehať v halách H3 a H6 od **8:30** do **11:00**
- po obede bude vyhodnotenie (m-162) a zapísanie známky do indexu
- na skúšku sa prihlasujete cez AIS2
- prvý opravný termín bol *5.2.2018* a druhý opravný *12.2.2018*

Testovacie zadania skúšky z minulých školských rokov

Zadania:

- *1. tréningové zadanie - skúška z 23.1.2017 - tree_sort*

- *2. tréningové zadanie - skúška z 30.1.2017 - TrieMap*
- *3. tréningové zadanie - skúška z 6.2.2017 - triedenie pomocou prioritného frontu*
- *4. tréningové zadanie - skúška z 30.5.2016 - množina AVL stromov*
- *5. tréningové zadanie - skúška z 1.6.2016 - kostra grafu*
- *6. tréningové zadanie - skúška z 13.6.2016 - minimálna sieť*
- *7. tréningové zadanie - skúška z 19.1.2015 - huffmanovo kódovanie*

Riešenia týchto zadaní môžete otestovať v L.I.S.T.