Reprezentace dat v paměti

- proměnná, deklarace

- statické a dynamické typování

- hodnotové a referenční typy

- mutable (list, dict, set, uživatelské objekty) a
  immutable (int, float, bool, string, tuple) objekty v Pythonu

- celé číslo `int`

obvykle omezení velikosti uložených hodnot 2^{31} resp. 2^{63}
znaménko + binární zápis hodnoty

v Pythonu hodnoty libovolné velikosti
seznam cifer („dlouhé“ číslo)
dva způsoby uložení hodnot typu `int`
→ má vliv na efektivitu operací

- číslo s pohyblivou řádovou čárkou `float`
  znaménko, mantisa, exponent

  mantisa → nepřesné uložení, zaokrouhlovací chyby

```
>>> 0.1+0.2
0.30000000000000004
>>> 1/49*49
0.9999999999999999
>>> 1/6+1/6+1/6+1/6+1/6+1/6
0.9999999999999999
```

  exponent → omezení rozsahu

```
>>> 1e308
1e+308
>>> 1e309
inf
```

- znak
- znakový řetězec

- seznam
  - prvky: libovolné objekty (dokonce různého typu)
  - časová složitost operací, realokace paměti
- pole
  - velikost určena při vytvoření pole
  - homogenní – všechny prvky stejného typu
  - `array.array, NumPy.array`

- objekt (uživatelský)
  - v některých jazycích vlastní správa paměti (uvolňování)
  - v některých garbage collector (např. v Pythonu)

- ukazatel
  - struktury spojované ukazateli, dynamické datové struktury

Operace se seznamem

insert(index, hodnota), remove(hodnota), pop(index), del(prvek)
– časová složitost O(*n*)

na konci seznamu: append(hodnota), pop()
– časová složitost amortizovaně O(1)

realokace seznamu při operacích append():

teoretický příklad – dvojnásobek:
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, …
pro $n = 1000$ se provede realokace devětkrát (počet přesunů $\log_2 n$),
celkem se vykoná na přesuny práce 1 + 2 + 4 + 8 + … + 512 = 1023,
tzn. celková práce O(*n*), což je 1 přesun na 1 append(), tedy O(1)

implementace v Pythonu (podle dokumentace):
0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, 269,
309, 354, 405, 462, 526, 598, 679, 771, 874, 990, 1120, …

Abstraktní datové typy

- datové struktury, které jsou definovány svým chováním
  * bez ohledu na způsob implementace
    (v poli, v lineárním spojovém seznamu apod.)
  * bez ohledu na typ uložených dat
    (čísla, řetězce, objekty...)

- příklad: zásobník, fronta, halda, …

- v objektových programovacích jazycích je realizujeme často ve formě třídy s metodami odpovídajícími požadovanému chování

- implementace těchto tříd se pak může podle potřeby změnit, aniž by to ovlivnilo chování struktury navenek

Zásobník (stack, LIFO – last in / first out)

- pamatuje si pořadí prvků
- pevné dno, přidává se na vrchol, odebírá se z vrcholu
- tedy odebírá se vždy ***nejmladší prvek***
- jiný prvek než vrchol není přístupný

- při implementaci v poli je dno na indexu 0,
  kapacita zásobníku je omezena velikostí pole
- při implementaci spojovým seznamem je vrchol
  zásobníku na začátku seznamu (aby byl snadno přístupný)
- konstantní časová složitost všech operací

Příklady použití:

mechanismus volání funkcí
prohledávání do hloubky
vyhodnocení aritmetického výrazu

V Pythonu:

- zásobník reprezentován seznamem ***z***
- dno zásobníku: z[0]
- vrchol zásobníku (kde se přidávají a odebírají prvky): z[-1]
- prázdný zásobník (také inicializace): z = [ ]

Vložení hodnoty ***x*** do zásobníku: `z.append(x)`

Odebrání hodnoty ze zásobníku a vložení do proměnné ***x***
(předpokládáme, že tam nějaká hodnota je, tzn. len(z) > 0,
jinak by bylo třeba přidat příslušný test):

```
x = z.pop()
```

Obě operace mají konstantní časovou složitost (amortizovaně).

```
class Zasobnik:
    """zásobník uložený v seznamu"""

    def __init__(self):
        self.s = []                 # prázdný zásobník

    def pridej(self, x):
        self.s.append(x)

    def odeber(self):
        return self.s.pop()
```

```
z = Zasobnik()
#je jedno, jakou implementaci třídy Zasobnik použijeme

z.pridej(20)
z.pridej(30)
print(z.odeber())
print(z.odeber())
```

Poznámka: v metodě odeber() jsme pro jednoduchost nekontrolovali, zda zásobník není prázdný.

Fronta (queue, FIFO – first in / first out)

- pamatuje si pořadí prvků
- přidává se na konec, odebírá se ze začátku
- tedy odebírá se vždy ***nejstarší prvek***
- jiný prvek fronty není přístupný

- při implementaci v poli si udržujeme aktuální indexy začátku a konce fronty,
  kapacita fronty je omezena velikostí pole
- při implementaci spojovým seznamem je začátek fronty na začátku seznamu (lze snadno odebírat)
  a konec fronty na konci seznamu (lze snadno přidávat, pokud máme pomocný ukazatel na poslední prvek)
- konstantní časová složitost operací (podle implementace)

Příklady použití:

čekající procesy (např. tisková fronta)
prohledávání do šířky
počítačová simulace

V Pythonu:

- fronta reprezentována seznamem ***f***
- konec fronty (místo příchodu do fronty): f[-1]
- začátek fronty (místo odchodu z fronty): f[0]
- prázdná fronta (také inicializace): f = [ ]

Vložení hodnoty ***x*** do fronty (stejné jako u zásobníku):

```
f.append(x)
```

Odebrání hodnoty z fronty a vložení do proměnné ***x***
(předpokládáme, že tam nějaká hodnota je, tzn. len(f) > 0,
jinak by bylo třeba přidat příslušný test):

```
x = f.pop(0)
```

Přidání prvku do fronty má konstantní časovou složitost (amortizovaně), ale odebrání prvku má složitost lineární!

Problém:
Při každém odebrání prvku z fronty se zbývající prvky fronty posunou v paměti o 1 místo „doleva“
$\rightarrow$ časová složitost operace odebrání je $\Theta(N)$, kde N je délka fronty.

Možnosti řešení:

1. Posunutí zbytku fronty v paměti provádět jen občas, třeba vždy po 10 odebráních prvku z fronty
→ posunuje se méně často a vždy na větší vzdálenost (což nevadí)

- musíme si pamatovat aktuální pozici začátku fronty
- časová složitost odebrání prvku z fronty zůstane lineární v nejhorším případě, ale v průměru se několikanásobně sníží

```
x = f[zacatek]
zacatek += 1
if zacatek == 10:
    del f[0:10]
    zacatek = 0
```

2. Stanovit si „kapacitu fronty“ a posunutí provádět jen tehdy, když je to nutné, tzn. není-li místo na právě vkládaný prvek
→ posunuje se ještě méně často a na větší vzdálenost

- musíme si pamatovat aktuální pozici začátku fronty
- časová složitost odebrání prvku z fronty bude vždy konstantní

```
x = f[zacatek]
zacatek += 1
```

- časová složitost vkládání bude v nejhorším případě lineární
  (ale velmi často se provede vložení prvku v konstantním čase)

```
if len(f) == kapacita:
    del f[0..zacatek]
    zacatek = 0
f.append(x)
```

3. Paměť zvolené kapacity přidělenou pro frontu ***f*** využíváme **cyklicky**, tzn. po posledním prvku následuje opět první prvek. Data se v paměti nikdy neposouvají → časová složitost obou operací zůstává ***konstantní***.

- musíme si pamatovat aktuální pozici začátku a konce fronty

```
f = [0] * kapacita        # prázdná fronta
zacatek = 0
konec = 0

f[konec] = x              # přidání prvku
konec = (konec + 1) % kapacita

x = f[zacatek]            # odebrání prvku
zacatek = (zacatek + 1) % kapacita
```

4. Některé programovací jazyky mají efektivní implementaci fronty připravenu v knihovnách. V Pythonu:

```
from collections import deque
f = deque()                 # prázdná fronta
f.append(x)                 # přidání prvku
x = f.popleft()             # odebrání prvku
```

Poznámka:

- třída `deque` je obecnější než fronta
- umožňuje přidávat a odebírat zprava i zleva (double ended queue)
  metody `append(), pop(), appendleft(), popleft()`
- lze tedy využít pro implementaci zásobníku i fronty
- je implementována jako obousměrný spojový seznam
- všechny uvedené operace mají konstantní časovou složitost
- má řadu dalších metod (podobné jako u třídy `list`)

Halda (heap)

- nepamatuje si pořadí příchodu prvků
- prvky musí být porovnatelné (definováno vzájemné uspořádání)
- odebírá se vždy ***nejmenší prvek***
- typické haldové operace: *přidat prvek*
  určit hodnotu minimálního prvku
  odebrat minimální prvek

- to by šlo implementovat různě (seznam, uspořádaný sezam),
  ale požadujeme časovou složitost všech operací O(log *N*)
- haldu si představujeme jako binární strom,
  který ovšem typicky implementujeme v poli

Příklady použití:

prioritní fronta
třídicí algoritmus HeapSort

Reprezentace haldy

- binární strom

- zcela zaplněné hladiny až do předposlední,
  poslední hladina zaplněná souvisle zleva
  důsledek: výška haldy = horní celá část z $\log_2 N$

- uspořádání hodnot (klíčů) ve všech uzlech: **otec ≤ syn**
  důsledek: v kořeni je uložena minimální hodnota

Efektivita operací

- určení minima: konstantní časová složitost
- přidání a odebrání prvku: logaritmická časová složitost **O(log *N*)**
  počet kroků výpočtu = nejvýše výška stromu

Operace na haldě

Přidání prvku:
- nový uzel přidat do haldy na poslední hladinu co nejvíce vlevo
- do nového uzlu vložit přidávanou hodnotu
- novou hodnotu opakovaně zaměňovat vždy s hodnotou uloženou v jejím otci, dokud je třeba (tzn. dokud má otec větší hodnotu)

Odebrání minima:
- odebrat minimum z kořenu haldy
- do kořenu vložit hodnotu z posledního uzlu haldy (uzel v poslední hladině co nejvíce vpravo), tento uzel zrušit
- přesunutou hodnotu opakovaně zaměňovat vždy s hodnotou uloženou v jejím synovi, dokud je třeba (tzn. dokud je syn menší; mají-li menší hodnotu oba synové, tak zaměnit s menším z obou synů)

Implementace haldy v poli nebo seznamu

- jednorozměrné pole/seznam ukládaných záznamů
- obsah haldy je v poli uložen po vrstvách vždy zleva doprava

2 možnosti:

a) prvek seznamu s indexem 0 nevyužijeme, kořen haldy má index 1
uzel s indexem i má syny uložené v poli na indexech $2i$, $2i+1$
tedy uzel s indexem i má otce uloženého v poli na indexu $i \,//\, 2$

b) kořen haldy má index 0
uzel s indexem i má syny uložené v poli na indexech $2i+1$, $2i+2$
tedy uzel s indexem i má otce uloženého v poli na indexu $(i-1) \,//\, 2$

```
halda = [None]      # prvek halda[0] nepoužijeme

def pridej(h, x):
    """do haldy 'h' přidá prvek 'x' """
    h.append(x)
    j = len(h)-1
    while j > 1 and h[j] < h[j//2]:
        h[j], h[j//2] = h[j//2], h[j]
        j //= 2
```

```
def zrusmin(h):
    """z haldy 'h' odebere minimální prvek"""
    if len(h) == 1:
        return None                 # prázdná halda
    zrus = h[1]
    h[1] = h[-1]
    del h[-1]
    j = 1
    while 2*j < len(h):
        n = 2*j
        if n < len(h)-1:
            if h[n+1] < h[n]:
                n += 1
        if h[j] > h[n]:
            h[j], h[n] = h[n], h[j]
            j = n
        else:
            break
    return zrus
```

Konstrukce haldy v lineárním čase

- výchozí rozložení dat představuje úplný binární strom hloubky *d* (bez uspořádání hodnot do haldy)
- nejprve postavíme „haldy“ z podstromů, jejichž kořeny mají hloubku *d*-1, potom pro *d*-2, … atd., až do kořene celé haldy
- stavění hald se provádí záměnami hodnot od kořene k listům (výměna s menším z obou synů)

Časová složitost

hloubka	počet uzlů	max. počet výměn pro každý z nich
0	2^0	d
1	2^1	d-1
…	…	…
j	2^j	d-j
…	…	…
d-1	2^{d-1}	1

Pozorování: Při tomto postupu konstrukce haldy má hodně uzlů malý maximální počet výměn a jen málo z nich může absolvovat výměn hodně $\rightarrow$ celková časová složitost **O(*N*)**.

Celkem se provede výměn (viz tabulka):

$$\sum_{j=0}^{d-1} 2^j \left(d-j\right) = d\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j.2^j =$$

$$= d.\left(2^d - 1\right) - \left(\left(d-2\right).2^d + 2\right) = O\left(2^d\right) = O\left(N\right)$$

… viz dva důkazy matematickou indukcí dále

Důkaz matematickou indukcí č.1: $\sum_{j=0}^{d-1} 2^j = 2^d - 1$

1. pro $d = 1$ zjevně platí

2. nechť platí pro všechna $d < D$, dokazujeme platnost pro $D > 1$:

$$\sum_{j=0}^{D-1} 2^j = \sum_{j=0}^{D-2} 2^j + 2^{D-1} = \left(2^{D-1} - 1\right) + 2^{D-1} = 2.2^{D-1} - 1 = 2^D - 1$$

↑ qed

podle indukčního předpokladu

Důkaz matematickou indukcí č.2: $\sum_{j=0}^{d-1} j.2^j = (d-2).2^d + 2$

1. pro d=1 zjevně platí

2. nechť platí pro všechna $d < D$, dokazujeme platnost pro $D > 1$:

$$\sum_{j=0}^{D-1} j.2^j = \sum_{j=0}^{D-2} j.2^j + (D-1).2^{D-1} = \left((D-3).2^{D-1} + 2\right) + (D-1).2^{D-1} =$$

↑

podle indukčního předpokladu

$$= D.2^{D-1} - 3.2^{D-1} + D.2^{D-1} - 2^{D-1} + 2 = 2.D.2^{D-1} - 4.2^{D-1} + 2 =$$

$$= D.2^D - 2.2^D + 2 = (D-2).2^D + 2$$ qed

Třídění haldou (haldové třídění, HeapSort)

- z prvků postavit haldu (*N* x přidání prvku do haldy)
  → časová složitost O(*N*.log *N*)
  nebo zdola v lineárním čase O(*N*)
- haldu postupně rozebrat (*N* x odebrat minimum z haldy)
  → časová složitost O(*N*.log *N*)

→ tedy celková časová složitost **O(*N*.log *N*)** i v nejhorším případě

- třídí „na místě“ (tzn. nepotřebuje další datovou strukturu velikosti *N*):
  * prvky uložené v poli postupně řadí do haldy,
    přičemž haldu staví v levé části téhož pole
  * pak rozebírá postupně haldu a vyřazované prvky ukládá
    postupně odzadu do pravé části téhož pole, kde se uvolňuje
    místo po zkracující se haldě

Prioritní fronta

- podobné jako fronta, prvky se „předbíhají“ podle svých priorit
- zachování vzájemného pořadí mezi prvky téže priority požadovat můžeme, ale nemusíme (záleží na konkrétní aplikaci)

Možnosti implementace:

- seznam (pole, LSS), do něhož zařazujeme podle priority
- seznam (pole, LSS), z něhož vybíráme podle priority
- samostatné seznamy pro každou hodnotu priority (pokud tyto hodnoty známe a není jich mnoho)
- halda řazená podle priorit – pokud nepožadujeme zachovat pořadí
- halda řazená podle dvojic (priorita, čas příchodu) – pokud požadujeme zachovat vzájemné pořadí mezi prvky téže priority

Slovník

- uchovává dvojice *klíč – hodnota* (klíč je jednoznačný)
- abstraktní datový typ s operacemi
  vyhledej(klíč), vlož(klíč, hodnota) a vymaž(klíč)
- asociativní pole
- uložené údaje se vyhledávají podle klíče („indexuje se“ klíčem)
- klíč může mít jakoukoliv neměnitelnou hodnotu,
  dokonce třeba každý záznam může mít klíč jiného typu
- některé programovací jazyky přímo podporují
  v Pythonu: `dictionary`
- efektivní implementace je složitější (pomocí hešování)
- časová složitost přístupu k prvku je v průměru konstantní,
  ale v nejhorším případě lineární vzhledem k počtu prvků