

# BucketSort (příhrádkové třídění)

- třídíme celé záznamy podle zvoleného celočíselného klíče

*Realizace:*

**a** – původní seznam záznamů délky  $N$

**b** – setříděný seznam záznamů délky  $N$

**c** – pomocný seznam celých čísel délky  $R$

představuje pole celých čísel s indexy  $D:H$

\* nejprve velikosti příhrádek pro prvky s daným klíčem  
(= čítače výskytů hodnot klíče jako u CountingSortu)

\* potom index v seznamu **b**, kde příslušná příhrádka začíná  
(získá se prefixovými součty)

\* dále index prvního volného místa v příhrádce  
(zvyšování hodnoty při zaplňování příhrádky)

*Příklad:*

máme seznam dvojic údajů o dětech (jméno, věk)  
chceme ho uspořádat podle věku

```
a = [ ("Jan", 8), ("Petr", 6), ("Karel", 5),  
      ("Tomáš", 6), ("Václav", 4), ("Josef", 6),  
      ("Martin", 7), ("Jakub", 2), ("Pavel", 5) ]
```

```

def trid_prihradky(a, d, h):
    c = [0] * (h-d)          # prázdné přihrádky

    for x in a:
        c[x[1]-d] += 1        # velikosti přihrádek
                                # x[1] je zde klíč prvku x

    z = c[0]
    c[0] = 0
    for i in range(1, h-d):
        c[i], z = z, z + c[i] # začátky přihrádek

    b = [None] * (len(a))     # prázdný výsledný seznam
    for x in a:
        klic = x[1]-d
        b[c[klic]] = x
        c[klic] += 1          # volné místo v přihrádce

    return b

```

*Příklad:*

máme seznam dvojic údajů o dětech (jméno, věk)  
chceme ho uspořádat podle věku

```
a = [("Jan", 8), ("Petr", 6), ("Karel", 5), ("Tomáš", 6), ("Václav", 4),  
      ("Josef", 6), ("Martin", 7), ("Jakub", 2), ("Pavel", 5)]
```

Zavoláme `a = trid_prihradky(a, 1, 10)`

Hodnoty seznamu **c** po jednotlivých krocích výpočtu:

<code>[0, 0, 0, 0, 0, 0, 0, 0, 0]</code>	prázdné přihrádky
<code>[0, 1, 0, 1, 2, 3, 1, 1, 0]</code>	velikosti přihrádek
<code>[0, 0, 1, 1, 2, 4, 7, 8, 9]</code>	začátky přihrádek

Výsledek:

```
[('Jakub', 2), ('Václav', 4), ('Karel', 5), ('Pavel', 5), ('Petr', 6),  
 ('Tomáš', 6), ('Josef', 6), ('Martin', 7), ('Jan', 8)]
```

*Paměťová složitost*

seznam délky  $N$  a pomocný seznam délky  $R$   $\rightarrow O(N + R)$

*Časová složitost*

dva cykly délky  $N$  a dva cykly délky  $R$   $\rightarrow O(N + R)$

tzn. je **lineární**, ale nejen vzhledem k počtu záznamů  $N$ ,  
nýbrž i k přípustnému rozsahu hodnot  $R$

Přihrádkové třídění je (při vhodné implementaci) **stabilní**,  
tzn. zachovává vzájemné pořadí prvků se stejnou hodnotou klíče.

## Jiný způsob implementace

**c** není pole čítačů, kolikrát se objevila která hodnota klíče, ale je to pole seznamů obsahujících záznamy s příslušným klíčem - tedy  $c[k]$  je přímo přihrádka pro záznamy s klíčem  $k$

→ prvky ze seznamu **a** rozdělujeme do správných přihrádek a nakonec jenom obsah všech přihrádek spojíme za sebe

## Varianty, modifikace

- čísla nejsou celá, ale desetinná  
→ při předem známém omezeném počtu desetinných míst lze snadno převést na celé číslo

*Příklad:* řadíme studenty podle průměrného prospěchu evidovaného na 2 desetinná místa, např. 1,36. Po vynásobení stem máme celočíselné klíče v rozsahu od 100 do 500.

- rozsah klíčů  $R$  je příliš velký (náročné na paměť i čas)  
→ víceprůchodové přihrádkové třídění

# RadixSort (víceprůchodové přihrádkové třídění)

- je-li rozsah hodnot  $R$  příliš velký  
(např. záznamů je sice jen několik stovek, ale klíčem je libovolné šesticiferné číslo → potřebovali bychom seznam  $c$  délky 1.000.000)

*Řešení:*

- klíč rozdělíme např. na dvě trojciferné (nebo na tři dvojciferné) části
- provedeme nejprve přihrádkové třídění celého pole podle **dolní** (méně významné) trojice cifer
- v další fázi setřídíme celé pole přihrádkovým tříděním podle **horní** (významnější) trojice cifer

Díky **stabilitě** přihrádkového třídění se při závěrečné fázi zachová uspořádání záznamů získané v předchozích fázích výpočtu.



- stačí podstatně menší pole **c**, které se využije opakovaně  
→ úspora paměti i času  
(např. místo jednoho průchodu seznamem **c** délky 1.000.000  
máme nyní dva průchody délky 1.000)
- v krajním případě rozdělíme klíč po jednotlivých cifrách  
→ pak nám stačí pole velikosti 10 a potřebný počet průchodů  
je dán počtem cifer v klíči  
→ časová složitost  $O(N \cdot \log R)$

# 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
- číslo s pohyblivou řádovou čárkou
- znak
- znakový řetězec
- pole
- seznam (realokace paměti)
- objekt
- ukazatel, struktury spojované ukazateli
- garbage collector

## Realokace seznamu

append(), pop() – časová složitost amortizovaně  $O(1)$

insert(), remove() – časové složitost amortizovaně  $O(n)$

realokace seznamu při operacích append():

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, ...

pro  $n = 1000$  se provede realokace 27-krát

# Abstraktní datové typy

- datové struktury, které jsou definovány svým chováním
  - \* bez ohledu na způsob implementace (v poli, v LSS 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 metod se pak může podle potřeby změnit, aniž by to ovlivnilo chování struktury navenek

# Zásobník (stack, LIFO)

- 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 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).

*Pozn.:* každé jedno volání metody `append()` má v nejhorším případě časovou složitost  $\Theta(n)$  – může vyvolat realokaci paměti.

```
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()
```



```

class Zasobnik:
    """zásobník jako spojový seznam"""

    def __init__(self):
        self.p = None                # prázdný zásobník

    def pridej(self, x):                # na začátek
        q = Uzel(x)
        q.dalsi = self.p
        self.p = q

    def odeber(self):                # ze začátku
        q = self.p
        self.p = self.p.dalsi
        return q.info

```

```
z = Zasobnik()  
#je jedno, kterou 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)

- 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 (stejně jako u zásobníku):

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

Odebrání hodnoty z fronty a vložení do proměnné  $x$   
(předpokládáme, že tam nějaká hodnota je, tzn.  $\text{len}(f) > 0$ ,  
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“

→ časová složitost operace odebrání je  $\Theta(N)$ .

## *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 zlepší

```
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 připravenou 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, připomíná spíše seznam
- umožňuje přidávat i odebírat zprava i zleva (double ended queue)
- má mnoho dalších metod (podobné jako u třídy list)

## ***Implementace spojovým seznamem***

- fronta realizována jednoduchým LSS a dvěma ukazateli:  
na začátek (tj. na první prvek LSS)  
a na konec (tj. na poslední prvek LSS)
- na začátku LSS se dobře přidává i odebírání prvek,  
na konci LSS se dobře přidává, ale špatně odebírání  
→ **na začátku LSS bude odchod, na konci LSS bude příchod**  
*(tzn. každý ukazuje na toho, kdo přišel po něm,  
což je obráceně, než ve frontách v běžném životě)*
- prázdná fronta = prázdný LSS

Jiná možná realizace: LSS s hlavou  
(snáze se pak programují operace s dočasně prázdnou frontou)

# 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, ale požadujeme časovou složitost všech haldových operací  $O(\log N)$
- haldu si představujeme jako binární strom, který ovšem typicky implementujeme v poli

*Příklady použití:*

prioritní fronta  
třídící algoritmus HeapSort

## **Výška $H$ binárního stromu o $N$ uzlech**

= délka nejdelší cesty z kořenu do listu

minimální výška: vyvážený strom

$$N = 2^0 + 2^1 + \dots + 2^H = 2^{H+1} - 1 \quad \rightarrow H \approx \log_2 N$$

maximální výška: degenerovaný strom  $\rightarrow H \approx N$

v průměrném případě výška  $O(\log N)$

## 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 nejmenší hodnota*

## Efektivita operací

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

## Operace

### *Přidání prvku:*

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

### *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 postupně 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

- jednorozměrné pole ukládaných záznamů, indexované od 1
- obsah haldy je v poli uložen po vrstvách vždy zleva doprava (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  $i // 2$ )

V Pythonu je pole realizováno seznamem.

Prvek seznamu s indexem 0 nevyužijeme (nebo do něj umístíme kořen haldy a patřičně upravíme indexování synů → levý syn  $2i+1$ , pravý syn  $2i+2$ )

```
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

```