# 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 2*i*, 2*i*+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 2*i*+1, pravý syn 2*i*+2)

```python
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
```

```python
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 → celková časová složitost **O(*N*)**.

*Celkem se provede výměn (viz tabulka):*

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

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

… 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 :

$$\sum_{j=0}^{D-1} j\,.\,2^j = \sum_{j=0}^{D-2} j\,.\,2^j + (D-1).2^{D-1} = \Big(\underset{\uparrow}{(D-3).2^{D-1} + 2}\Big) + (D-1)\,.\,2^{D-1} =$$

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)
  $\rightarrow$ časová složitost O(*N*.log *N*)
  nebo zdola v lineárním čase O(*N*)
- haldu postupně rozebrat (*N* x odebrat minimum z haldy)
  $\rightarrow$ časová složitost O(*N*.log *N*)

$\rightarrow$ 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 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

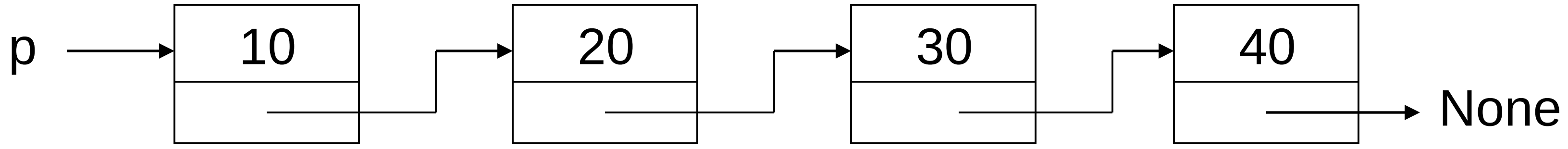
# Slovník (dictionary)

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

# Lineární spojový seznam

```
class Uzel:
    """uzel spojového seznamu"""

    def __init__(self, x = None, dal = None):
        self.info = x                  # uložená hodnota
        self.dalsi = dal               # následník
```



```
# vytvoření spojového seznamu p s hodnotami 10 20 30
p = Uzel(10)
q = Uzel(20)
r = Uzel(30)
p.dalsi = q
q.dalsi = r

p = Uzel(10, Uzel(20, Uzel(30)))

# průchod a výpis
s = p
while s != None:
    print(s.info, end = " ")
    s = s.dalsi
print()
```

```python
# poslední uzel
if p == None:
    print("prazdny seznam")
else:
    s = p
    while s.dalsi != None:
        s = s.dalsi
    print(s.info)
```

```
# vyhledání zadané hodnoty
hodnota = 20
print("hledame", hodnota)
s = p
while s != None and s.info != hodnota:
    s = s.dalsi
if s == None:
    print("nenalezen")
else:
    print("nalezen", s.info)
```

```
# přidání na začátek seznamu
t = Uzel(40)
t.dalsi = p
p = t

# přidání na konec seznamu
if p == None:
    p = Uzel(50)
else:
    s = p
    while s.dalsi != None:
        s = s.dalsi
    s.dalsi = Uzel(50)
```

# Operace se spojovým seznamem

- určit počet prvků
- vypsat všechny hodnoty
- nalezení posledního prvku
- vyhledání prvku s danou hodnotou

- přidání prvku na začátek, na konec seznamu
- vytvoření seznamu z dat na vstupu
- vytvoření kopie seznamu

- přidání prvku na dané místo
- přidání prvku do uspořádaného seznamu (na správné místo)

- odebrání prvku ze začátku, z konce seznamu
- odebrání daného prvku

## Operace se spojovým seznamem (pokr.)

- zrušení všech prvků v seznamu s danou hodnotou
- obrácení pořadí prvků v seznamu
- uspořádání prvků v seznamu podle hodnoty

- spojení dvou seznamů do jednoho (ze sebe)
- slití dvou uspořádaných seznamů do jednoho (merge)

- rozdělení seznamu do dvou (podle pozice lichý/sudý)
- rozdělení seznamu do dvou (podle hodnoty lichý/sudý)

## Příklady použití lineárních spojových seznamů

*Zásobník*

- realizován jednoduchým LSS, ukazatel na začátek seznamu ukazuje na vrchol zásobníku (dno zásobníku = **`None`**)
- prázdný zásobník = prázdný LSS
- přidávání i odebírání prvků se provádí **na začátku** LSS – snadné

```python
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*

- 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 odebírá prvek,
  na konci LSS se dobře přidává, ale špatně odebírá
  → **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)

*Dlouhá čísla*

- uložena po cifrách nebo po skupinách cifer podobně jako v poli
- nejsme omezeni maximálním možným počtem cifer v čísle
- směr řazení LSS závisí na prováděných operacích (např. pro sčítání nebo násobení odzadu od cifer nejnižšího řádu, pro výpis ale potřebujeme cifry odpředu – bude nutno otáčet seznam)

*Polynomy*

- opět podobná reprezentace jako u polí – v každém prvku LSS uložen koeficient a exponent jednoho členu polynomu, členy s nulovým koeficientem se do LSS neukládají
- pro provádění operací je výhodné mít LSS uspořádaný (vzestupně nebo sestupně) podle hodnot exponentu