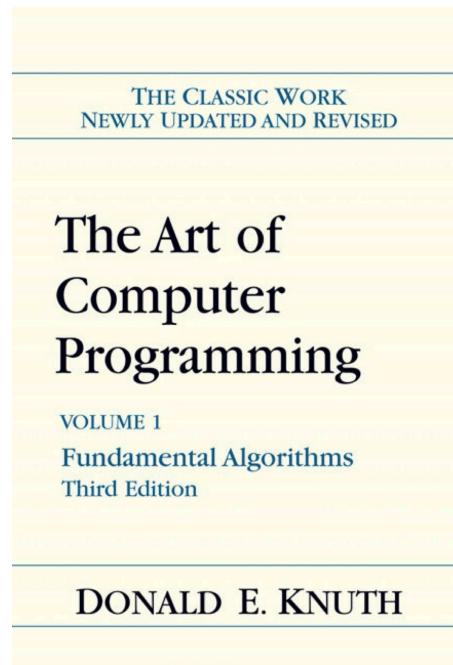


# Algoritmizace

## Základní datové struktury II



# Co bylo minule

- ✎ Abstraktní datové typy zásobník a fronta
  - implementace v poli
- ✎ Spojové seznamy – úvod

# Zásobník / Stack

Množina prvků, k nimž přistupujeme metodou LIFO

- last-in, first-out



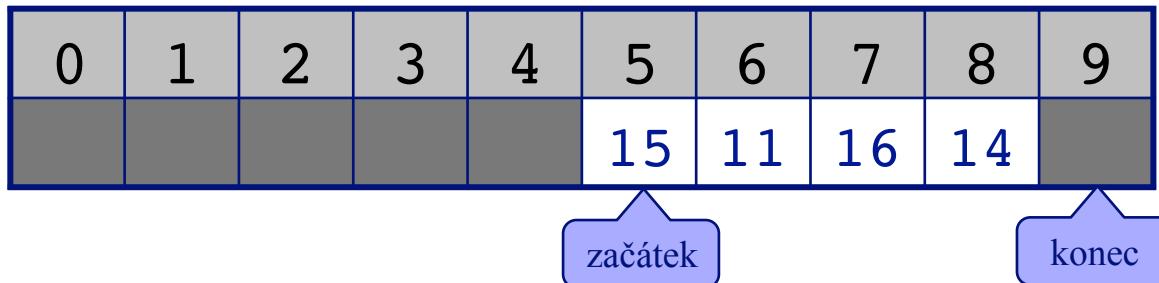
# Fronta / Queue

Množina prvků, k nimž přistupujeme metodou **FIFO**

- first-in, first-out



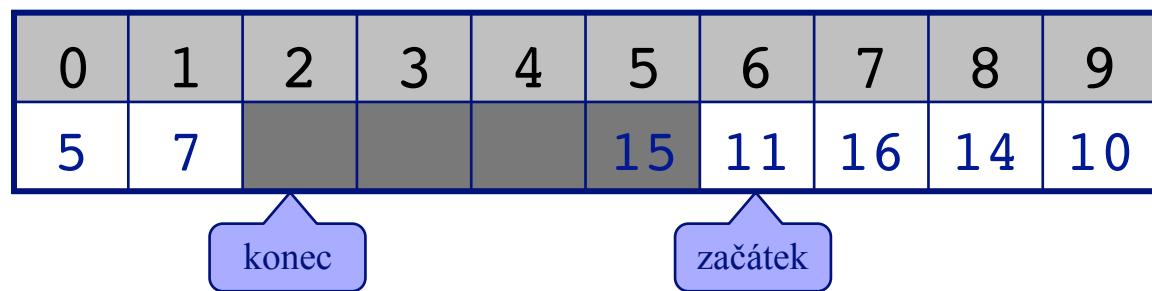
# Cyklická fronta



| Enqueue 10, 5, 7



## Dequeue



# Cyklická fronta v poli

s ošetřením chyb !

```
class FrontaPole:

    def __init__(this,kapacita):
        this.zacatek, this.konec = 0,0
        this.pole = [None] * kapacita
        this.kapacita = kapacita

    def empty(this):
        return this.zacatek == this.konec

    def dequeue(this):
        if this.empty():
            raise IndexError("prázdná fronta")
        x = this.pole[this.zacatek]
        this.zacatek = (this.zacatek + 1) % this.kapacita
        return x
```

# Cyklická fronta v poli

s ošetřením chyb !

```
# pokračování třídy FrontaPole
def enqueue(this,x):
    # nový konec fronty
    novy = (this.konec + 1) % this.kapacita
    # test na volné místo
    if novy == this.zacatek:
        raise IndexError("přeplněná fronta")
    # ted' můžeme bezpečně zařadit nový prvek
    this.pole[this.konec] = x
    this.konec = novy
```

## 👉 Důsledek

- do fronty lze uložit jen **this.kapacita - 1** prvků

# Cyklická fronta v poli – jinak

Jak zajistit, abychom využili plnou kapacitu?

① Jiná inicializace

- atributů **zacatek** a **konec**

② Budeme si pamatovat počet prvků ve frontě

- nový atribut **počet**
- **konec** už nepotřebujeme
- stačí udržovat **zacatek** a **pocet**

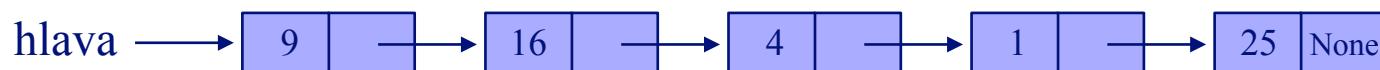
# Spojové seznamy

## Posloupnost prvků

- každý prvek je prezentován **uzlem**
- který kromě dat obsahuje i **odkaz**
- na uzel následující / předchozí (pokud existuje)

## Lineární spojový seznam

- každý uzel vyjma posledního odkazuje na uzel následující



# Uzel spojového seznamu

```
class Uzel:  
    """třída pro reprezentaci uzlu"""  
    def __init__(this,  
                 x = None,  
                 dalsi = None):  
        this.info = x  
        this.dalsi = dalsi
```

# Spojový seznam jako třída

```
class SpojovySeznam:  
    def __init__(this):  
        """vytvoří prázdný seznam"""  
        this.hlava = None # hlava seznamu  
    def najdi(this,klic):  
        """zjistí zda ∃ prvek s klicem"""  
        p = this.hlava  
        while p != None and p.info != klic:  
            p = p.dalsi  
        return p != None
```

# Co bude dnes

## ❖ Spojové seznamy

- další operace
- implementace zásobníku a fronty

## ❖ Prioritní fronta

## ❖ Konstrukce binární haldy v lineárním čase

## ❖ Slovník

# Zásobník ve spojovém seznamu

```
class ZasobnikSpojovySeznam:  
    def __init__(this):  
        this.hlava = None # prázdný zásobník  
    def push(this,x):  
        """vloží x do hlavy spojového seznamu"""  
        this.hlava = Uzel(x,this.hlava)  
    def pop(this):  
        """odebere a vrátí první prvek seznamu"""  
        # zásobník musí být neprázdný  
        x = this.hlava.info  
        this.hlava = this.hlava.dalsi  
        return x
```

# Fronta ve spojovém seznamu

```
class FrontaSpojovySeznam:

    def __init__(this):
        this.konec = None # vytvoř prázdný seznam
        this.delka = 0     # nulové délky

    def dequeue(this):
        # předpokládá neprázdnou frontu
        x = this.konec.dalsi # hlava k odebrání
        if this.delka == 1:
            this.konec = None # fronta se vyprázdnila
        else:
            this.konec.dalsi = x.dalsi # přeskoč původní
        this.delka -=1           # hlavu
        return x.info
```

# Fronta ve spojovém seznamu

```
# pokračování třídy FrontaSpojovySeznam:  
def enqueue(this,x):  
    """vloží x na konec fronty"""  
    novy = Uzel(x) # vytvoř nový uzel  
    if this.delka == 0: # vytvoř cyklickou frontu  
        novy.dalsi = novy # délky 1  
    else:                      # nový odkazuje  
        novy.dalsi = this.konec.dalsi # na hlavu  
        this.konec.dalsi = novy # původní konec na novy  
    this.konec = novy          # novy se stane koncovým  
    this.delka += 1
```

# Spojové seznamy – rekapitulace

Zásobník / fronta ve spojovém seznamu

- Pop, Push, Enqueue, Dequeue v čase  $O(1)$

Další operace

- ulož prvek na začátek / konec / určené místo
- odeber prvek ze začátku / konce / určeného místa
- najdi prvek se zadaným / max / min klíčem
- obrácení seznamu
- zřetězení dvou seznamů

Využití

- seznam v Pythonu → spojový seznam
  - » dlouhá čísla, polynomy, BucketSort

# Problémy: spojové seznamy

- ① Do implementace zásobníku a fronty ve spojových seznamech doplňte ošetření chybových stavů.
- ② Rozmyslete si realizaci dalších operací nad spojovými seznamy (viz předchozí stránka).

# Prioritní fronta

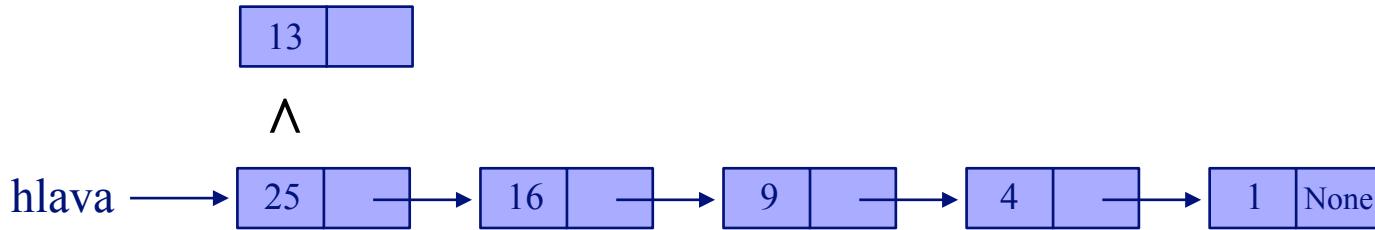
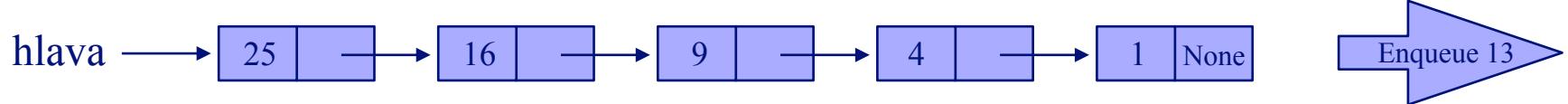
Fronta, v níž má každý prvek definovánu svoji **prioritu**

- odebíráme vždy prvek s maximální prioritou
- je-li takových více, odebereme libovolný z nich

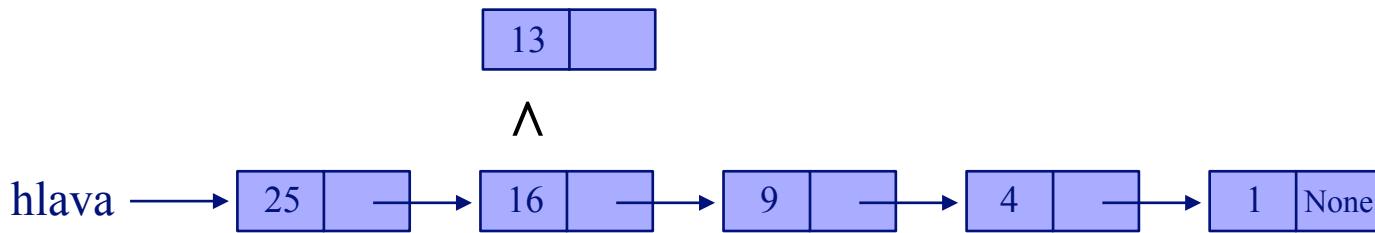
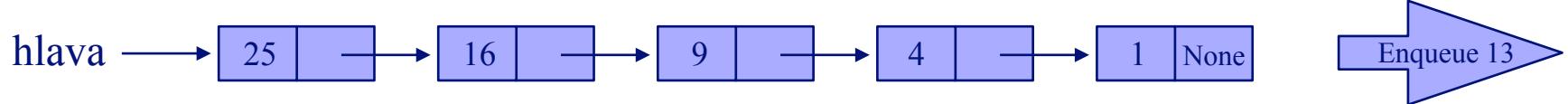
Implementace spojovým seznamem

- stačí si pamatovat odkaz na hlavu (`this.hlava`)
  - » jako v `ZasobnikSpojovySeznam`
- seznam **neudržujeme** setříděný
  - » **Enqueue**: vkládáme do hlavy – čas  $O(1)$
  - » **Dequeue**: prvek s max prioritou musíme najít – čas  $\Theta(n)$
- seznam **udržujeme** setříděný sestupně dle priorit
  - » **Dequeue**: odebíráme z hlavy – čas  $O(1)$
  - » **Enqueue**: vkládaný prvek zatřídíme do seznamu – čas  $\Theta(n)$ 
    - viz třídění vkládáním (`InsertionSort`)

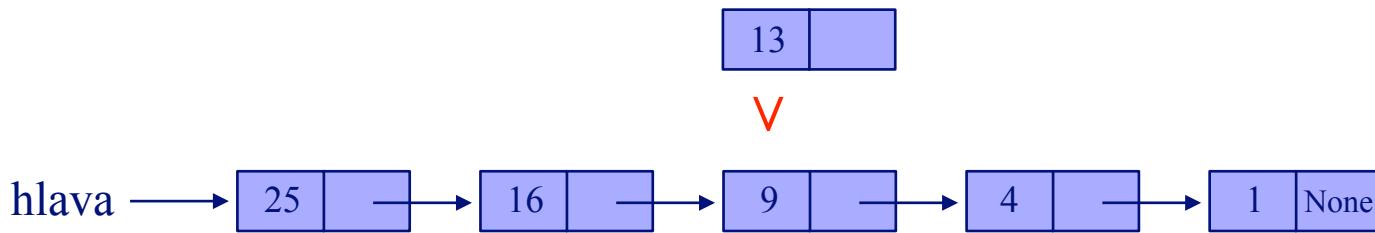
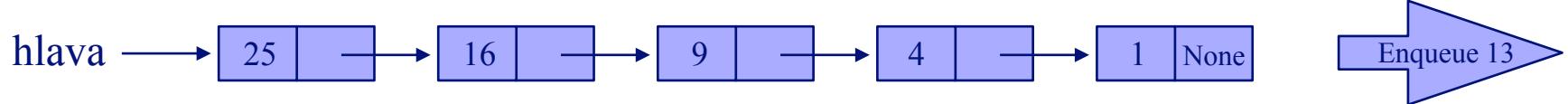
# Prioritní fronta ve spojovém seznamu



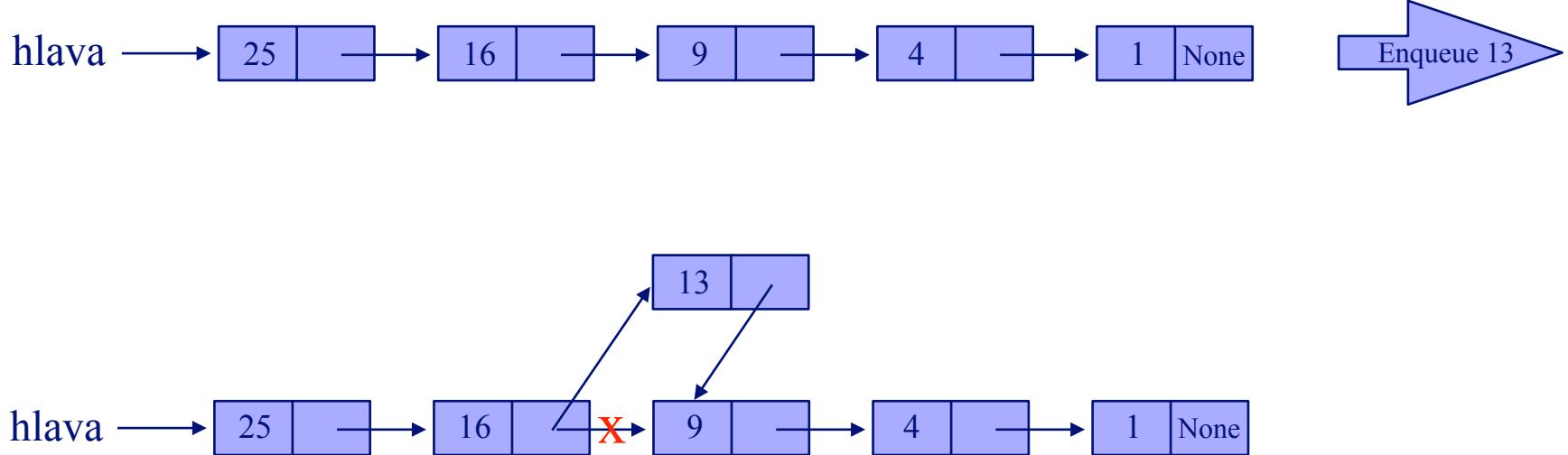
# Prioritní fronta ve spojovém seznamu



# Prioritní fronta ve spojovém seznamu



# Prioritní fronta ve spojovém seznamu



## Pozor na mezní případy

- **Enqueue 100** – vkládáme do hlavy
- **Enqueue 0** – vkládáme na konec seznamu
- prázdná fronta – vytvoříme novou 1prvkovou frontu

# Binární halda jako prioritní fronta

Prioritní frontu lze implementovat jako **binární haldu**

- max-halda uložená v poli
- **Enqueue** – Přidej prvek do haldy
- **Dequeue** – OdeberMax z haldy
- časová složitost obou operací  $O(\log n)$

Konstrukce binární haldy **v lineárním čase**

- prvky v poli  $\leftrightarrow$  prvky v binárním stromě tvaru haldy
- je třeba zajistit haldové uspořádání
- budeme je opravovat od hladiny  $h-1$ ,  $h =$  výška haldy
- v  $i$ -tém kroku opravíme uspořádání pro vrcholy na hladině  $h-i$ ,  $i = 1, 2, \dots, h$
- každý vrchol necháme “probublat dolů”

# Konstrukce haldy v lineárním čase

halda o  $n$  prvcích má vždy  
 $n//2$  vnitřních vrcholů

```
def makeHeap(a):
```

```
    for i in reversed(range(len(a)//2)):
```

porovnej  $a[i]$  s max (max-halda)

či min (min-halda) z jeho dětí

(pokud existují)

v případě potřeby vyměň

pokračuj analogicky dokud

není splněna podmínka uspořádání

nebo dokud nejsme v listu

# Konstrukce haldy – složitost

Halda výšky  $h \geq 2$  o  $n$  prvcích

$i$ -tá hladina,  $i = h-1, h-2, \dots, 0$

- celkem  $2^i$  vrcholů
- pro každý nejvýše  $h-i$  výměn

Celkový # výměn

$$\leq \sum_{i=0}^{h-1} 2^i (h - i) = \sum_{j=1}^h 2^{h-j} j = 2^h \sum_{j=1}^h \frac{j}{2^j} \leq n \sum_{j=1}^h \frac{j}{2^j}$$

 **Problém:** Je  $\sum_{j=1}^h \frac{j}{2^j} \leq \text{konstanta?}$

# Konstrukce haldy – složitost

Součet prvních  $n$  členů geometrické posloupnosti

$$a + aq + aq^2 + \cdots + aq^{n-1} = a \frac{q^n - 1}{q - 1} \text{ pro } q \neq 1$$

$$\sum_{j=1}^h \frac{j}{2^j} = \sum \begin{pmatrix} 1/2 & & & & \\ 1/4 & 1/4 & & & \\ 1/8 & 1/8 & 1/8 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots \\ 1/2^h & 1/2^h & 1/2^h & \dots & 1/2^h \end{pmatrix}$$

1. sloupec  $\frac{1}{2} \cdot \frac{\frac{1}{2^h} - 1}{\frac{1}{2} - 1} = 1 - \frac{1}{2^h} < 1$

2. sloupec < polovina 1. sloupce <  $\frac{1}{2}$

Celkový součet <  $1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{h-1}} = 2 - \frac{1}{2^{h-1}} < 2$  ✓

# Prioritní fronta – operace

Zatím jsme zkoumali

- Enqueue – čas  $O(\log n)$
- Dequeue – čas  $O(\log n)$
- Max – vrátí prvek s max prioritou – čas  $O(1)$

Další operace

- ZvýšeníPriority zadaného prvku
- Odstraň zadaný prvek

✎ **Problém:** Zvládneme obě operace v čase  
 $O(\log n)$  ?

# Slovník

ADT pro reprezentaci dynamické množiny  
s operacemi

**Vyhledej( $k$ )**

- vrátí prvek s klíčem  $k$ , pokud existuje

**Ulož( $x$ )**

- ulož do slovníku prvek  $x$

**Vymaž( $k$ )**

- vymaž ze slovníku prvek s klíčem  $k$

# Implementace slovníku

## Python

- datový typ slovník (dictionary)

Budeme mít později

- vyhledávací stromy
- hašovací tabulky