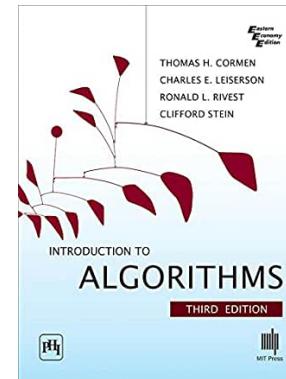
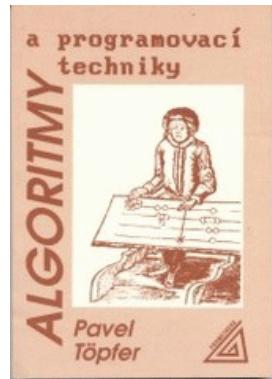


# Algoritmizace

Tomáš Dvořák

ZS 2019/20



# O přednášce

- ⌚ ZS 2/1 (Zk, Z)
- ⌚ Po 12:20 - 13:50 S5
- 💻 <http://ksvi.mff.cuni.cz/~dvorak/vyuka>



MOODLE pro výuku 1  
Univerzita Karlova

<https://dl1.cuni.cz/course/view.php?id=8186>

# O přednášce

 Úvodní kurz algoritmů a datových struktur pro posluchače 1.r. Bc. studia informatiky a učitelství informatiky

 Prerekvizity

 Lineární algebra 1, Diskrétní matematika

- základní pojmy: množina, posloupnost, funkce ...

 Programování 1

- programovací jazyk Python

# O zkoušce

## Zkouška

písemná část

ústní část

pro přihlášení ke zkoušce **bude třeba zápočet!**

# O cvičení

## ⌚ 10 sekcí



## ✉ Požadavky k zápočtu

- řešení domácích úloh
  - ✓ zadávány na cvičení
  - ✓ odevzdání prostřednictvím systému
- další požadavky → cvičící
  - ✓ účast



# O přednášejícím

- ☺ Tomáš Dvořák
- ✉ Tomas.Dvorak *at* mff.cuni.cz
- 👉 MS, 4. patro, č. 405



# Sylabus přednášky

- algoritmy a jejich efektivita
- vyhledávání a třídění
- základní datové struktury
- rekurze
- stromové datové struktury
- prohledávání do hloubky a do šířky
- algoritmy teorie her
- základní grafové algoritmy
- obecné metody návrhu

# Literatura

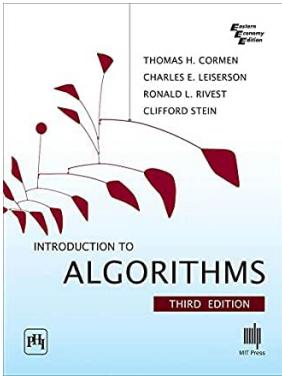


Pavel Töpfer  
*Algoritmy a programovací techniky*  
2. vydání  
Prometheus, Praha 2007

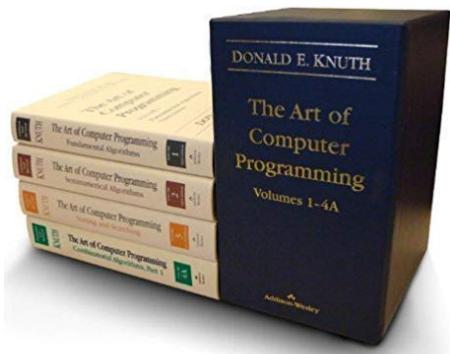


Martin Mareš, Tomáš Valla  
*Průvodce labyrintem algoritmů*  
CZ.NIC, Praha 2017  
<http://pruvodce.ucw.cz>

# Literatura



Thomas H. Cormen, Charles E. Leiserson,  
Ronald L. Rivest, Clifford Stein  
*Introduction to Algorithms*  
3<sup>rd</sup> edition, MIT Press  
Cambridge, MA 2009



Donald E. Knuth  
*The art of computer programming*  
Volumes 1-4A  
Addison Wesley, 2011  
[[html](#)]

# Algoritmizace

## Algoritmy a jejich efektivita



# Osnova

- ❖ Co je to algoritmus?
- ❖ Jak budeme algoritmy popisovat?
- ❖ Jak budeme ověřovat jejich správnost?
- ❖ Jak změřit efektivitu algoritmu?
- ❖ Asymptotická složitost

# Algoritmus

أبو عبد الله محمد بن موسى الخوارزمي أبو جعفر

Abú Abd Alláh Muhammad Ibn Músá al-Chórezmí  
Perský učenec, cca 780 - 850



# Algoritmus

أبو عبد الله محمد بن موسى الخوارزمي أبو جعفر

Abú Abd Alláh Muhammad Ibn Músá al-Chórezmí

Perský matematik & astronom, cca 780 - 850

- systém arabských číslic
- základy algebra
- řešení lineárních  
& kvadratických rovnic



# Co je to algoritmus?

Intuitivní pojem

Popis takového řešení problému,  
které lze realizovat na počítači.



# Co je to algoritmus?

*Konečná posloupnost  
elementárních příkazů,  
jejichž provádění umožňuje  
pro každá přípustná vstupní data  
mechanickým způsobem  
získat po konečném počtu kroků  
příslušná výstupní data.*



*J. Drózd, R. Kryl, Začínáme s programováním, Grada, Praha 1992.*

# Vlastnosti algoritmu

Konečnost

Hromadnost (obecnost, univerzálnost)

Resultativnost (výstup)

Jednoznačnost

Determinismus

# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

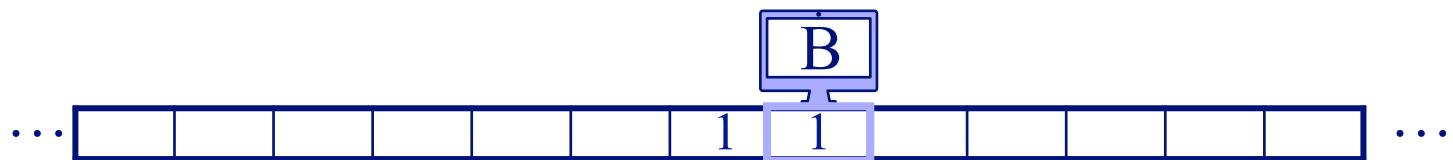
symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

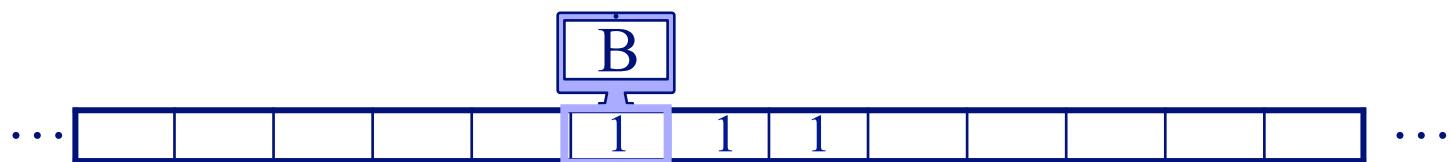
symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

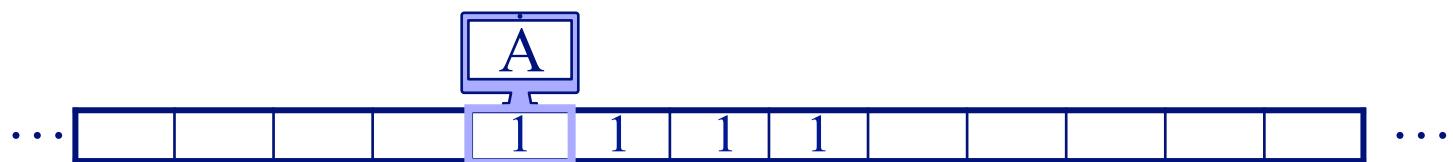
symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

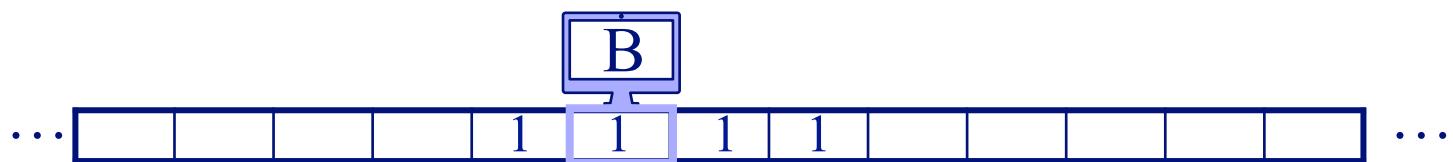
symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

Busy Beaver  
(T. Radó, 1962)

symbol na pásce	stav A	stav B
□	1, B, $\rightarrow$	1, A, $\leftarrow$
1	1, B, $\leftarrow$	1, HALT, $\rightarrow$

n	$\Sigma$
1	1
2	4
3	6
4	13
5	?



# Formální modely algoritmu

Turingův stroj (Alan Turing, 1936)

- Churchova teze

RAM počítač

Rekurzivní funkce (Kurt Gödel, 1934)

Lambda kalkul (Alonzo Church, 1941)

# Jak budeme algoritmy popisovat?

## Zápis v pseudokódu

- použití přirozeného jazyka
- řídící struktury vypůjčené z jazyka Python

Nebudeme se zabývat problémy softwarového inženýrství jako

- modularita
- objektový přístup
- ošetření chyb apod.

# Problém



Jsou dány rovnoramenné váhy a  $n$  kuliček.  
Navrhнete algoritmus, který najde

- ① nejtěžší kuličku na co nejmenší počet vážení
- ② nejtěžší i nejlehčí kuličku s použitím nejvýše  
 $3\lceil n/2 \rceil$  vážení
- ③ druhou nejtěžší kuličku s použitím nejvýše  
 $n-2+\lceil \log_2 n \rceil$  vážení.

# Ověření správnosti algoritmu

= ověření konečnosti + částečné správnosti

## Konečnost

- pro každá přípustná vstupní data obdržíme v konečném čase nějaký výstup

## Částečná správnost (parciální korektnost)

- když výpočet nad přípustnými vstupními daty skončí
- pak na výstupu obdržíme správný výsledek

Algoritmus je **správný** = částečně správný  
+ konečný

# Porovnávání efektivity algoritmů

## Dvě míry

- čas
- prostor (paměť)

Jak změřit časovou / prostorovou náročnost výpočtu?

- délka
- prostorová náročnost

výpočtu počet kroků  
výpočtu rozsah použité pracovní paměti

# Co je to krok výpočtu ?

## Krok výpočtu

- elementární operace
- kterou lze provést v **konstantním** čase

## Příklady

- provedení logického testu
- aritmetické operace
- přiřazení

# Co je to složitost algoritmu?

Délka (prostorové nároky) výpočtu závisí na

- velikosti vstupních dat
- konkrétní hodnotě vstupních dat

Přirozené zjednodušení

- složitost algoritmu **bude funkcí velikosti vstupu**

## 👉 Problém

- pro dané  $n$  může existovat více přípustných vstupů o této velikosti !

# Přístupy k analýze složitosti

Nejhorší případ

maximální délka výpočtu nad vstupem délky  $n$

Nejlepší případ

minimální délka výpočtu nad vstupem délky  $n$

Průměrný případ

součet délek výpočtů nad všemi vstupy délky  $n$   
/ počet vstupů délky  $n$

Pravděpodobnostní analýza algoritmů

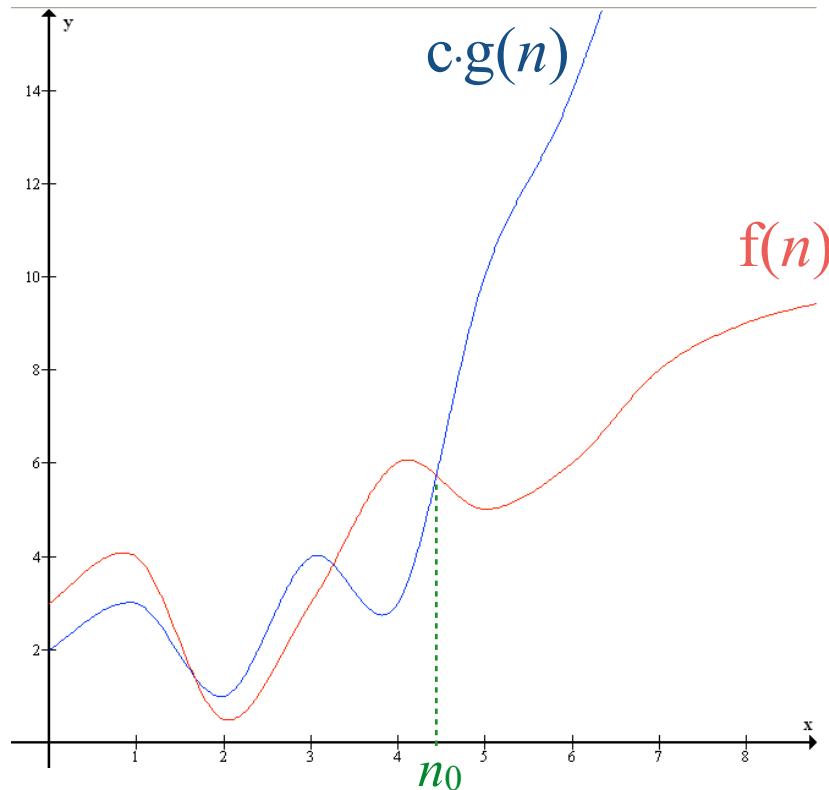
 Příklad

Navrhňte algoritmus, který setřídí  $n$  zadaných kuliček  $a_1, \dots, a_n$  od nejlehčí po nejtěžší.

```
for j in range(n-1):
    for i in range(1, n-j):
        if a[i] těžší než a[i+1]:
            vyměň a[i] ↔ a[i+1]
```

# Asymptotická notace

Funkce  $f(n) = \mathcal{O}(g(n))$ , pokud  $\exists c > 0$  a  $n_0 \in \mathbb{N}$  tak,  
že  $0 \leq f(n) \leq c \cdot g(n)$  pro každé  $n \geq n_0$ .



# Asymptotická notace

Funkce  $f(n) = \Omega(g(n))$ , pokud  $\exists c > 0$  a  $n_0 \in \mathbf{N}$  tak,  
že  $0 \leq c \cdot g(n) \leq f(n)$  pro každé  $n \geq n_0$ .

Funkce  $f(n) = \Theta(g(n))$ , pokud  
 $f(n) = O(g(n))$  a  $f(n) = \Omega(g(n))$ .

# Spektrum časové složitosti

$\Theta(1)$  (např. je číslo liché / sudé?)

$\Theta(\log n)$  (binární vyhledávání)

$\Theta(n)$  (nalezení minima / maxima)

$\Theta(n \log n)$  (HeapSort, MergeSort)

$\Theta(n^2)$  (BubbleSort, InsertSort)

$\Theta(n^3)$  (násobení matic dle definice)

...

---

$\Theta(2^n)$

$\Theta(n!)$

...

---

pracují v  
polynomiálně  
omezeném čase

pracují v  
exponenciálním  
čase

algoritmicky nerozhodnutelné

# Problém



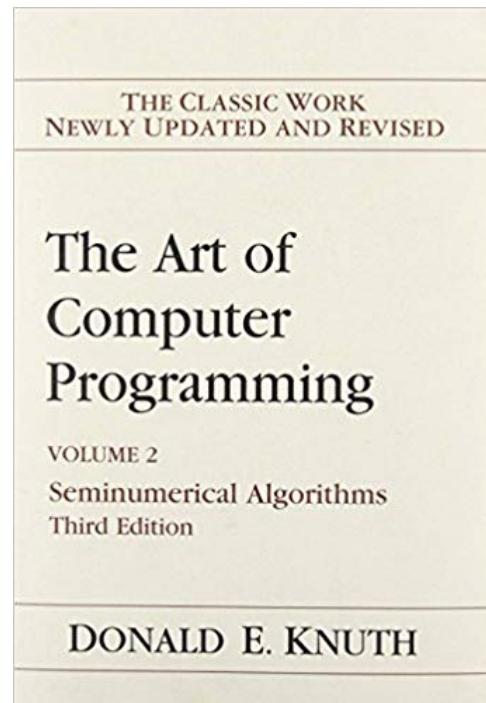
Dokažte nebo vyvrát'te:

Pro každou dvojici funkcí  $f, g: \mathbf{N} \rightarrow \mathbf{R}$  platí

- ① Pokud  $f(n) = O(g(n))$ , pak  $g(n) = O(f(n))$
- ② Pokud  $f(n) = O(g(n))$ , pak  $2^{f(n)} = O(2^{g(n)})$
- ③ pokud  $f(n) = O(g(n))$ , pak  $g(n) = \Omega(f(n))$
- ④  $f(n) = O(f(n)^2)$

# Algoritmizace

## Algoritmy teorie čísel



# Test prvočíselnosti

Vstup: přirozené číslo  $N > 1$

Výstup: True pokud  $N$  je prvočíslo

False je-li  $N$  číslo složené

```
def prvocislo(n):  
    for d in range(2,n):  
        if n % d == 0:  
            return False  
    return True
```

# Test prvočíselnosti

👉 **Diskuze:** zrychlení “hrubé síly”

- stačí prověřit dělitele  $\leq \sqrt{N}$
- stačí se omezit na lichá čísla

👉 Protože délka vstupu  $n = \lfloor \log_2 N \rfloor + 1$ ,  
algoritmus má ve skutečnosti

exponenciální časovou složitost !

# Test prvočíselnosti – složitost

Složitost problému určování prvočíselnosti čísla  $N$

Agrawal, Kayal, Saxena (2002)

- $\tilde{O}(\log^6 N)$

Pomerance, Lenstra (2005)

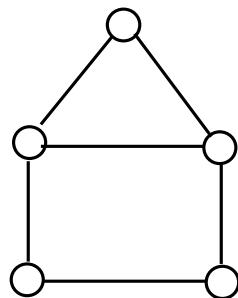
- $\tilde{O}(\log^{12} N)$

# Jak měřit délku vstupu?

$a_1, a_2, \dots, a_n$

$n =$  počet prvků posloupnosti

graf



$n =$  počet vrcholů

$m =$  počet hran

matice

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

$n =$  řád matice

přirozené číslo  $N$        $n = \lfloor \log_2 N \rfloor + 1$

# Generování prvočísel

Vstup: přirozené číslo  $n > 1$

Výstup: všechna prvočísla z  $\{2, 3, \dots, n\}$

Eratosthenovo síto

Eratosthenés z Kyrény

- řecký matematik, astronom, geograf
- 276 – 195/194 př.n.l.

 **Idea.** Pro každé vygenerované prvočíslo lze vyloučit všechny jeho násobky  $\leq n$ .

# Erastóthenovo síto

```
def sito0(n):  
  
    prvocisla = []  
    je_prv = [False, False] + [True] * (n-1)  
  
    for p in range(2, n+1):  
        if je_prv[p]:  
            prvocisla.append(p)  
            for i in range(2*p, n+1, p):  
                je_prv[i] = False  
  
    return prvocisla
```

# Erastóthenovo síto – zrychlení

## ☀️ Vylepšení

- ① Stačí “prosívat” od  $p^2$  místo  $2 \cdot p$ 
  - násobky  $k \cdot p$  pro  $k < p$  již byly vyškrtnuty dříve

```
def sito(n):  
    prvocisla = []  
    je_prv = [False, False] + [True] * (n - 1)  
    for p in range(2, n + 1):  
        if je_prv[p]:  
            prvocisla.append(p)  
            for i in range(p**2, n + 1, p):  
                je_prv[i] = False  
    return prvocisla
```

# Erastóthenovo síto – vylepšení

## ☀ Vylepšení

- ② `je_prv[]` nemusí evidovat **sudá** čísla !
  - úspora paměti i času

# Největší společný dělitel

## ✎ Problém

- jsou dána přirozená čísla  $x$  a  $y$
- určete jejich největší společný dělitel  $\text{NSD}(x,y)$

## Algoritmy

### ① Hrubá síla

- $\text{NSD}(x,y) = \max\{d \in \{1, 2, \dots, \min\{x, y\}\} \mid d \mid x \text{ a } d \mid y\}$
- postupně prověřit kandidáty od největšího

# Největší společný dělitel

## ✎ Problém

- jsou dána přirozená čísla  $x$  a  $y$
- určete jejich největší společný dělitel  $\text{NSD}(x,y)$

## Algoritmy

### ② Prvočíselný rozklad

👉 **Věta.** Každé přirozené číslo  $>1$  lze jednoznačně rozložit na součin prvočísel.

### 💡 **Příklad:** $\text{NSD}(30, 24) = ?$

- $30 = 2 \cdot 3 \cdot 5$
- $24 = 2 \cdot 2 \cdot 2 \cdot 3$
- $\text{NSD}(30, 24) = 2 \cdot 3 = 6$

# Největší společný dělitel

## ❖ Problém

- jsou dána (kladná) přirozená čísla  $x$  a  $y$
- určete jejich největší společný dělitel  $\text{NSD}(x,y)$

## Algoritmy

### ③ Euklidův algoritmus

Eukleidés / Euklides / Euklid / Εὐκλείδης

- řecký matematik, 325 - 260 př. n. l
- Alexandria (Egypt)
- základy geometrie, teorie čísel
- Základy / Στοιχεῖα
  - » “nejúspěšnější matematické dílo”, 13 knih



# Euklidův algoritmus

 **Pozorování.** Pro přirozená čísla  $x > y$  platí:

$$d \mid x \text{ a } d \mid y \iff d \mid x - y \text{ a } d \mid y$$

 **Proč?**

 **Důsledek.**  $\text{NSD}(x, y) = \text{NSD}(x - y, y)$  pro  $x > y$ .

 **Příklad**

$$\text{NSD}(30, 24) = ?$$

$$= \text{NSD}(6, 24) = \text{NSD}(24, 6)$$

$$= \text{NSD}(18, 6)$$

$$= \text{NSD}(12, 6)$$

$$= \text{NSD}(6, 6) = 6$$

# Euklidův algoritmus

```
def euklid0(x,y):  
    while x != y:  
        if x > y:  
            x -= y  
        else:  
            y -= x  
    return x
```

## Správnost Euklidova algoritmu

- konečnost
  - » invariant cyklu:  $x,y > 0$
  - » tedy i  $x+y > 0$
  - » po provedení těla **while**-cyklu se  $x+y$  sníží alespoň o 1
  - » po nejvýše  $x+y$  iteracích **while**-cyklu výpočet skončí

# Euklidův algoritmus

```
def euklid0(x,y):  
    while x != y:  
        if x > y:  
            x -= y  
        else:  
            y -= x  
    return x
```

## Správnost Euklidova algoritmu

- částečná správnost
  - » invariant cyklu: viz **Důsledek**
  - »  $\text{NSD}(x,x)=x$

# Euklidův algoritmus – zrychlení

## ☀ Příklad

$$\begin{aligned}\text{NSD}(27,21) &= \text{NSD}(21,6) \\ &= \text{NSD}(15,6) \\ &= \text{NSD}(9,6) \\ &= \text{NSD}(6,3) \\ &= \text{NSD}(3,3) = 3\end{aligned}$$

}

zbytek po  
celočíselném dělení

$$21 \bmod 6 = 3$$

☀ **Idea.** Opakované odečítání lze nahradit zbytkem po celočíselném dělení!

👉 **Důsledek.**  $\text{NSD}(x, y) = \text{NSD}(y, x \bmod y)$   
pro (kladná) přirozená čísla  $x, y$ .

👉 **Pozorování.**  $x \bmod y = 0 \Rightarrow y \mid x$   
 $\Rightarrow \text{NSD}(x, y) = y$

# Euklidův algoritmus - finální verze

```
def euklid(x,y):  
    while y > 0:  
        x,y = y,x % y  
    return x
```

## ☀️ Příklad

$$\begin{aligned}\text{NSD}(27,21) &= \text{NSD}(21,6) \\ &= \text{NSD}(6,3) \\ &= \text{NSD}(3,0) = 3\end{aligned}$$

# Euklidův algoritmus – složitost

```
def euklid(x, y):  
    while y > 0:  
        x, y = y, x % y  
    return x
```

- 👉 Počet iterací těla **while**-cyklu je nejvýše  $\log_2 x + \log_2 y + 1$ .

## 👉 Důkaz

- $x = y$  : jen jedna iterace
- $x < y$  : hodnoty se vymění
- $x > y$  :  $x \cdot y$  se zmenší alespoň o polovinu

# Euklidův algoritmus – složitost

## 👉 Důkaz

Případ  $x > y$  podrobněji:

- $x \bmod y \leq \min\{y - 1, x - y\} < \frac{x}{2}$
- $y \cdot x \bmod y < \frac{x \cdot y}{2}$

Bud'te  $x^{(i)}, y^{(i)}$  hodnoty proměnných  $x, y$   
po provedení  $i$ -té iterace těla while-cyklu, pak

- $x^{(i)} \cdot y^{(i)} < \frac{x \cdot y}{2^i}$

Není-li  $i$ -tá iterace poslední, pak  $x^{(i)} > y^{(i)} > 0$ , čili

- $2 \leq x^{(i)} \cdot y^{(i)} < \frac{x \cdot y}{2^i}$
- $i + 1 < \log_2(x \cdot y) = \log_2 x + \log_2 y$

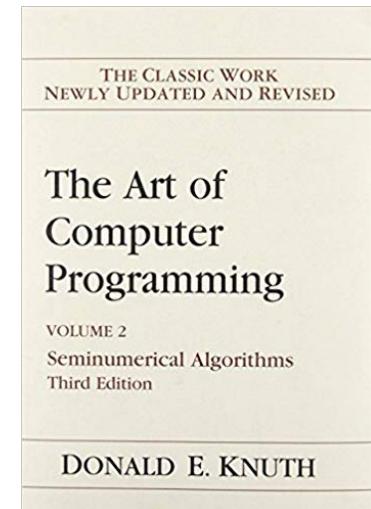
# Euklidův algoritmus – složitost

```
def euklid(x,y):  
    while y > 0:  
        x,y = y,x % y  
    return x
```

☞ Euklidův algoritmus výpočtu NSD( $x,y$ ) přirozených čísel  $x, y \in \{1, 2, \dots, n\}$  vykoná v průměrném případě nejvýše

$$\frac{12 \ln 2}{\pi^2} \ln n \approx 0.5842 \log_2 n$$

dělení.



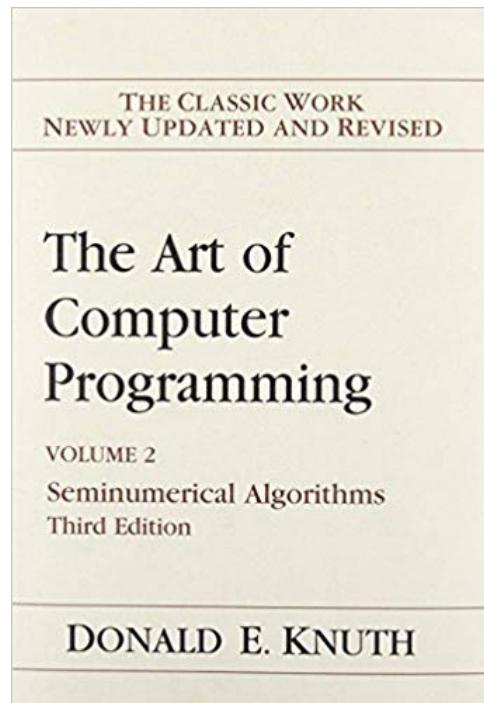
# Problémy



- ① Srovnáte složitost Euklidova algoritmu se složitostí algoritmu výpočtu NSD pomocí rozkladu na prvočinitele.
- ② Navrhněte efektivní algoritmus výpočtu nejmenšího společného násobku dvou zadaných přirozených čísel.

# Algoritmizace

## Algoritmy teorie čísel II



# Osnova

- ❖ Výpočet hodnoty polynomu
- ❖ Převody mezi číselnými soustavami
- ❖ Rychlé umocňování
- ❖ Výpočty s libovolnou přesností

# Vyhodnocení polynomu

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- polynom stupně  $n$
- s koeficienty  $a_n, a_{n-1}, \dots, a_1, a_0$
- $p(x) = 5x^3 + 10x + 1$
- $p(2) = 61$

## Přímý výpočet

- $\Theta(n^2)$  operací

# Vyhodnocení polynomu

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- **polynom** stupně  $n$
- s koeficienty  $a_n, a_{n-1}, \dots, a_1, a_0$
- $p(x) = 5x^3 + 10x + 1$
- $p(2) = 61$

## Hornerovo schéma

- William George Horner (1819)

$$p(x) = (\dots ((a_n x + a_{n-1}) x + a_{n-2}) x + \cdots + a_1) x + a_0$$

- $\Theta(n)$  operací

# Hornerovo schéma

koeficienty polynomu  
jako hodnota typu list

```
def horner(a, x):  
  
    h = 0  
  
    for i in range(len(a)):  
        h = h*x + a[i]  
  
    return h
```

# Převody mezi číselnými soustavami

## Desítková soustava

- $4321 = 4 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10 + 1$

## Číselná soustava o základu $b$

- řetězec  $a_n a_{n-1} \dots a_1 a_0$ , kde  $0 \leq a_i < b$
- $a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \dots + a_1 \cdot b + a_0$

☀ **Příklad:** převod z binární do desítkové soustavy

- použijeme Hornerovo schéma

$$\begin{aligned}10111_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2 + 1 \\&= (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1 \\&= 23_{10}\end{aligned}$$

# Převod z binární do desítkové

číslo v binární soustavě  
zadané jako hodnota typu str

```
def bin2dec(bin):  
  
    dec = 0  
  
    for i in range(len(bin)):  
        dec = dec * 2 + int(bin[i])  
  
    return dec
```

# Převod z desítkové do binární

☀️ **Příklad:** převod dekadického čísla 23  
do binární soustavy

$$\begin{aligned}23_{10} &= (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1 \\&= 10111_2\end{aligned}$$

Cifru nejnižšího rádu obdržíme  
jako zbytek po dělení 2

# Převod z desítkové do binární

☀️ **Příklad:** převod dekadického čísla 23 do binární soustavy

$$\begin{aligned} 23_{10} &= (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1 \\ &= 10111_2 \end{aligned}$$

Celočíselně vydělíme 2

# Převod z desítkové do binární

☀️ **Příklad:** převod dekadického čísla 23 do binární soustavy

$$\begin{aligned} 23_{10} &= (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1 \\ &= 10111_2 \end{aligned}$$

Další cifru obdržíme  
opět jako zbytek po dělení 2

# Převod z desítkové do binární

☀️ **Příklad:** převod dekadického čísla 23  
do binární soustavy

$$\begin{aligned}23_{10} &= (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1 \\&= 10111_2\end{aligned}$$

$$23 \bmod 2 = 1$$

$$23 \text{ div } 2 = 11$$

$$11 \bmod 2 = 1$$

$$11 \text{ div } 2 = 5$$

$$5 \bmod 2 = 1$$

$$5 \text{ div } 2 = 2$$

$$2 \bmod 2 = 0$$

$$2 \text{ div } 2 = 1$$

$$1 \bmod 2 = 1$$

$$1 \text{ div } 2 = 0$$

# Převod z desítkové do binární

přirozené číslo  
hodnota typu int

```
def dec2bin(dec):  
    bin = ""  
  
    while dec > 0:  
        bin = str(dec % 2) + bin  
        dec //= 2  
  
    return bin
```

# Problém



① Zobecněte funkce **bin2dec** a **dec2bin** tak, aby prováděly konverzi z / do libovolné číselné soustavy o základu  $b$ ,  $2 \leq b \leq 16$ .

Je-li  $b > 10$ , chybějící cifry reprezentujte velkými písmeny ze začátku abecedy, tj.

A, B, C, D, E, F.

# Rychlé umocňování

## ✎ Problém

- je dáno (velké) přirozené číslo  $N$  a hodnota  $X$
- určete  $X^N$

## Přímočaře z definice

- $X^N = X \cdot X \cdot \dots \cdot X$
- $N$ - 1 násobení
- **exponenciální čas !**

# Rychlé umocňování

## ❖ Problém

- je dáno (velké) přirozené číslo  $N$  a hodnota  $X$
- určete  $X^N$

Imitace převodu do binární soustavy

- $X^{16} = (((X^2)^2)^2)^2$
- jen 4 násobení namísto 15 !
- je-li  $N$  mocninou 2, lze použít opakované umocňování
- co když  $N$  není mocninou 2?

# Rychlé umocňování

## 💡 Jak spočítat $X^{13}$ ?

- převod exponentu do binární soustavy
- $13_{10} = (1101)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$   
 $= 2^3 + 2^2 + 2^0 = 8 + 4 + 1$
- $X^{13} = X^8 \cdot X^4 \cdot X$

# Rychlé umocňování

```
def mocnina(x, n):  
    mocnina = 1  
  
    while n > 0:  
        if n & 1 == 1: # n % 2 == 1  
            mocnina *= x  
        x, n = x*x, n >> 1 # n // 2  
  
    return mocnina
```

👉 **Pozorování.** Algoritmus rychlého umocňování vypočte  $X^N$  pomocí nejvýše

$$2 \log_2 N + 2 \text{ násobení.}$$

# Rychlé umocňování – aplikace

## Modulární umocňování

- $X^N \bmod m$
- $(X \cdot X) \bmod m = (X \bmod m \cdot X \bmod m) \bmod m$

## Aplikace

- kryptografický systém RSA

# Výpočty s libovolnou přesností

Vstup: dvě přirozená čísla  
počet cifer omezen jen velikostí paměti

Výstup: výsledek aritmetické operace  
(součet, rozdíl, součin, ...)

## Programovací jazyky

- omezení délkou strojového slova (64b) : C, C++
- podpora libovolné přesnosti: Python (bignum)

## Reprezentace

- pole (v Pythonu seznam) cifer
- pořadí od nejvyššího / nejnižšího řádu

# Příklad: součin dlouhých čísel

seznam cifer  
od nejvyššího řádu

```
def soucin(a,b):  
  
    soucin = [0]*(len(a)+len(b))  
  
    for i in reversed(range(len(a))):  
        for j in reversed(range(len(b))):  
            soucin[i+j+1] += a[i]*b[j]  
            soucin[i+j] += soucin[i+j+1] // 10  
            soucin[i+j+1] %= 10  
  
    if soucin[0] == 0:  
        return soucin[1:]  
    else:  
        return soucin
```

# Dlouhá čísla

## Celá čísla

- evidence znaménka

## Desetinná čísla

- poloha desetinné čárky
- celá část / desetinná část (2 pole)

## Prostorově úspornější reprezentace

- číselná soustava o základu  $b$
- kde  $b$  je mocnina  $2^8$  (např.  $b = 2^{32}$ )

# Problémy



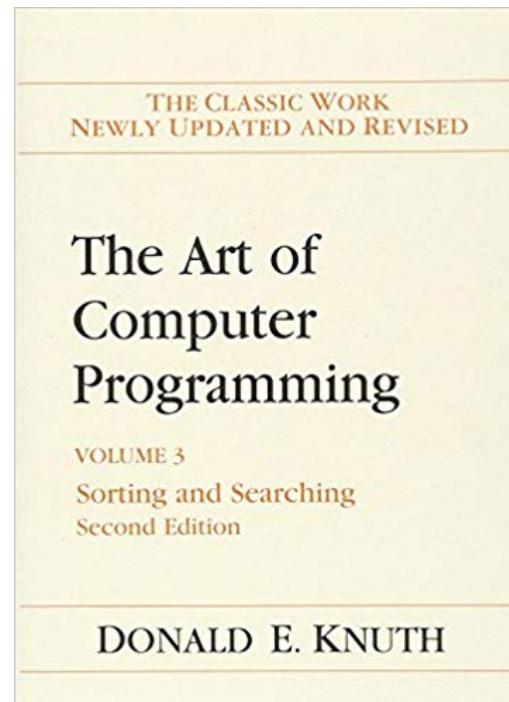
① V jazyce Python navrhněte funkci

soucet(a,b) ,

která vrátí součet dvou čísel, zadaných  
seznamem svých cifer. Zvažte obě varianty  
pořadí (od nejvýznamějšího / od nejméně  
významného řádu).

# Algoritmizace

## Vyhledávání



# Osnova

## ❖ Vyhledávání v poli

- sekvenční průchod
- binární vyhledávání

# Datová struktura pole (array)

Posloupnost položek **stejného typu**, které jsou uloženy za sebou v **souvislému bloku paměti**

Přístup k položkám v čase  $O(1)$  pomocí indexu

- identifikuje pořadí prvku
- nejčastěji přirozené číslo

prvocisla

0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19

Vestavěný datový typ

- ve většině programovacích jazyků
- C, C++, C#, Java, ...

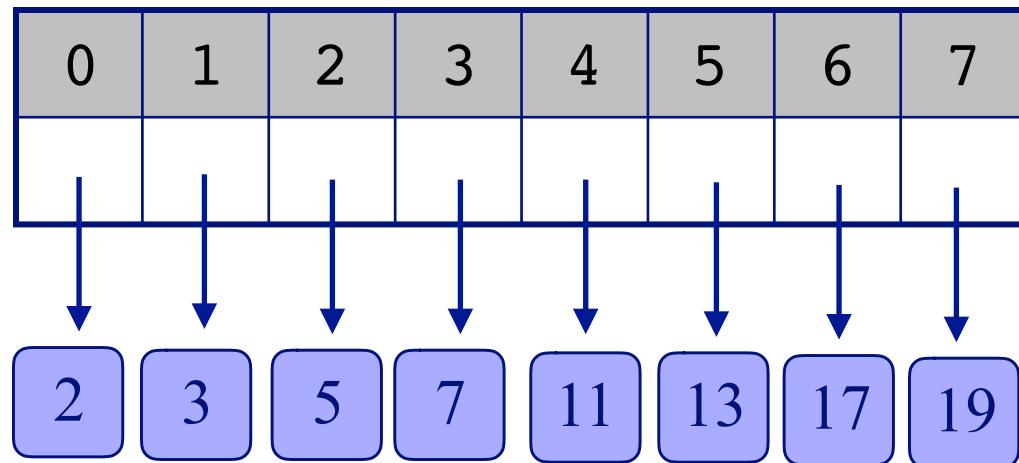
pevná délka (např. 4B)

# Datová struktura seznam (list)

## Seznam (list) v jazyce Python

- posloupnost prvků **libovolného typu**
- “heterogenní pole”
- indexovaná 0, 1, 2, ...
- přístup k prvku v čase  $O(1)$

prvocisla



# Vyhledávání v poli

Vstup: pole **a** indexované od 0, prvek **x**

Výstup: pokud se **x** v **a** vyskytuje, index 1. výskytu  
False jinak

👉 **Poznámka:** V jazyce Python k dispozici

- operátor **in**
  - metody **index()** a **count()**
- } čas  $\Theta(n)$   
} v nejhorším případě

```
>>> a = [10, 20, 30, 20]
>>> 20 in a
True
>>> a.index(20)
1
>>> a.count(20)
2
```

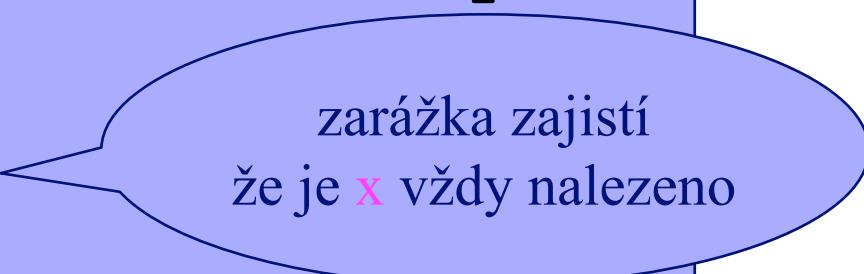
# Vyhledávání v poli – sekvenční průchod

```
def hledej(a, x):  
    for i in range(len(a)):  
        if a[i] == x:  
            return i  
    return False
```

Čas  $\Theta(n)$  v nejhorším případě

# Vyhledávání v poli – se zarážkou

```
def hledej1(a,x):  
    a.append(x) # vložení zarážky  
    # na konec seznamu  
    n = len(a)-1 # index zarážky  
  
    i = 0  
    while a[i] != x:  
        i += 1  
    del a[n] # odstranění zarážky  
  
    if i == n:  
        return False  
    else:  
        return i
```



zarážka zajistí  
že je **x** vždy nalezeno

# Vyhledávání v uspořádaném poli

Položky pole **a** jsou **uspořádané**

- `a[i] ≤ a[i+1]` **for** `i in range(n)`
- kde `n = len(a) - 1`

## Binární vyhledávání (půlení intervalu)

- je-li `a[n // 2] == x`, jsme hotovi
- je-li `a[n // 2] > x`, prohledáme
$$a[0], a[1], \dots, a[n // 2 - 1]$$
- je-li `a[n // 2] < x`, prohledáme
$$a[n // 2 + 1], a[n // 2 + 2], \dots, a[n]$$

# Binární vyhledávání

seznam

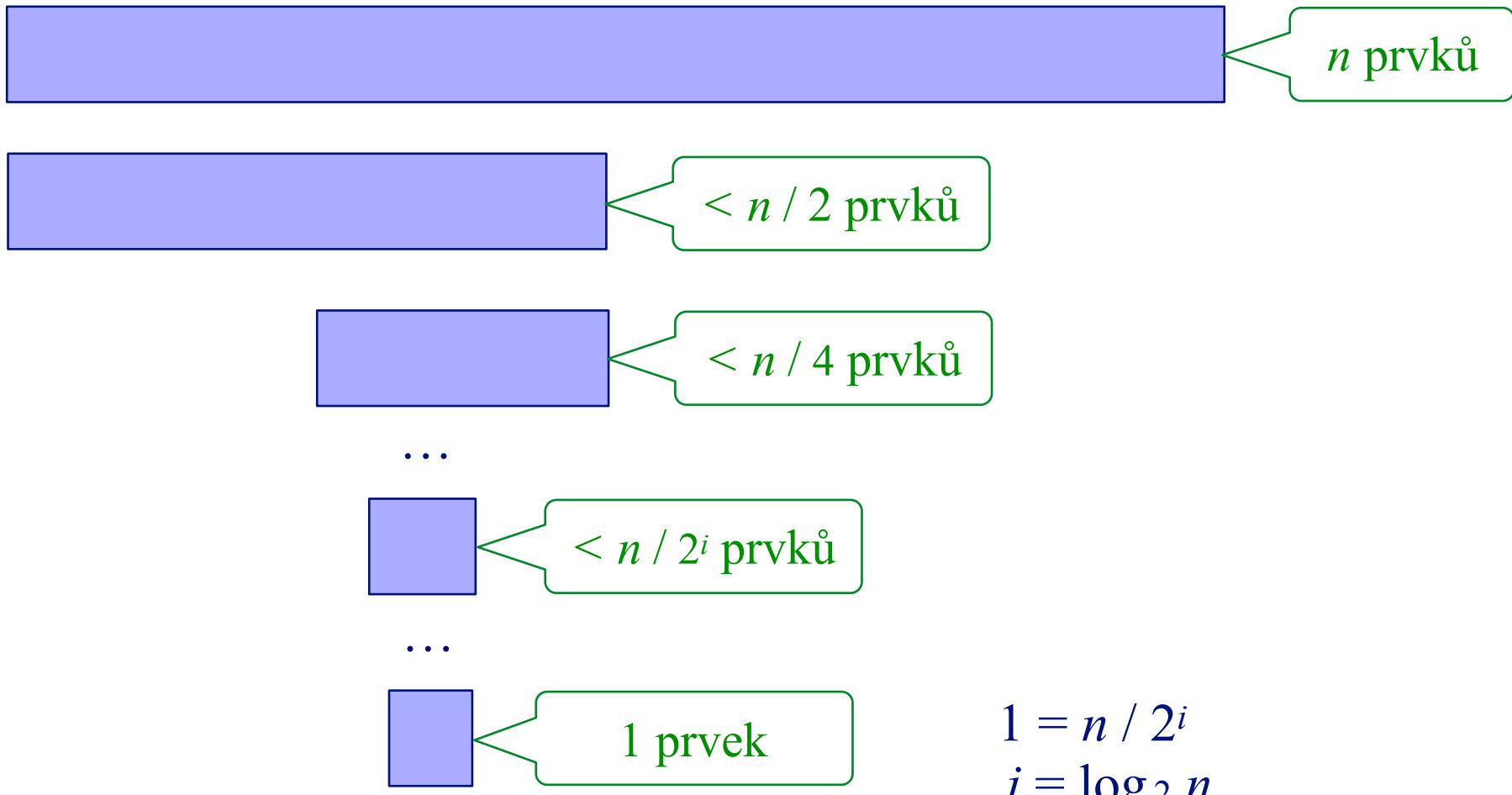
```
def binSearch(a, x):
    dolni, horni = -1, len(a)
    stred = (dolni + horni) // 2

    while horni - dolni > 1:
        if a[stred] == x:
            return stred
        elif a[stred] < x:
            dolni = stred
        else:
            horni = stred
        stred = (dolni + horni) // 2

    return False
```

Invariant cyklu:  $a[i] \neq x$  pro  $i \leq \text{dolni}$  a  $i \geq \text{horni}$

# Binární vyhledávání – složitost



Časová složitost  $\Theta(\log n)$

# Výpočet $\sqrt{n}$ půlením intervalu

$n > 1$

```
def odmocnina(n):  
  
    eps = 0.01  
    dolni, horni = 1.0, n  
    stred = (dolni + horni)/2.0  
    mocnina = stred**2  
  
    while abs(mocnina - n) >= eps:  
        if mocnina < n:  
            dolni = stred  
        else:  
            horni = stred  
        stred = (dolni + horni)/2.0  
        mocnina = stred**2  
  
    return stred
```

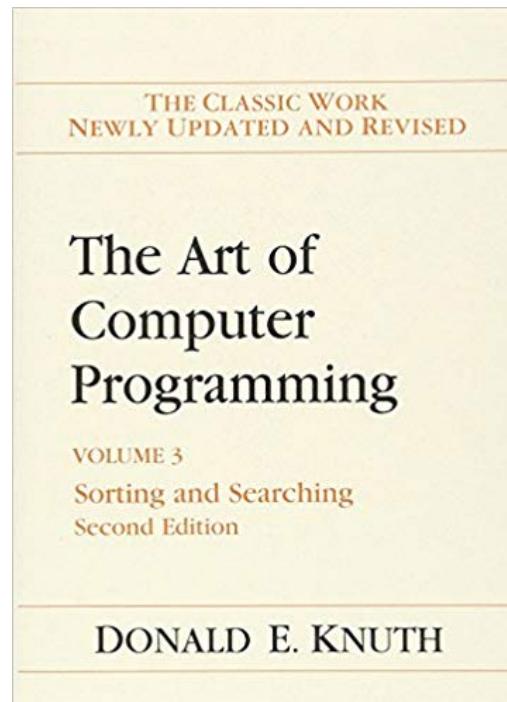
# Problémy



- ① Zobecněte funkci odmocnina( $n$ ) tak, aby fungovala pro libovolné kladné číslo  $n$ .

# Algoritmizace

## Třídění



# Osnova

## ❖ Třídění v poli

- přímé metody (BubbleSort, SelectionSort, InsertionSort)
- haldové třídění (HeapSort)

## ❖ Složitost problému vnitřního třídění

## ❖ Třídění v lineárním čase

# Vnitřní třídění

Vstup: pole **a** s prvky, které lze porovnávat

Výstup: pole **a** s prvky uspořádanými vzestupně

## Bublinkové třídění BubbleSort

- projdi pole a porovnej dvojice sousedních prvků
- v případě potřeby dvojici vyměň
- po dosažení konce seznamu začni znova od začátku
- pokračuj až na pozici poslední výměny v předchozím kroku
- není-li již žádná dvojice pro výměnu, výpočet končí

# Bublinkové třídění BubbleSort

```
def bubbleSort(a):  
  
    n = len(a)  
  
    while n > 1:  
        vymena = 0  
        for i in range(n-1):  
            if a[i] > a[i+1]:  
                a[i+1],a[i] = a[i],a[i+1]  
                vymena = i+1  
        n = vymena  
  
    return a
```

✖ Časová složitost  $\Theta(n^2)$

# Třídění výběrem SelectionSort

## První krok

- najdi minimální prvek
- a vyměň s prvkem na pozici 0

## Další krok

- mezi zbývajícími prvky najdi minimální
- a vyměň s prvkem na pozici 1

## Invariant cyklu

- po provedení  $i$ -tého kroku
- tvoří  $a[0], a[1], \dots, a[i-1]$  setříděný úsek
- který obsahuje  $i$  minimálních prvků pole  $a$

# Třídění výběrem SelectionSort

```
def selectionSort(a):  
    for i in range(len(a) - 1):  
        # a[i] vyměň s minimem z a[j], j≥i  
        minIndex = i  
        for j in range(i+1, len(a)):  
            if a[minIndex] > a[j]:  
                minIndex = j  
        a[i], a[minIndex] = a[minIndex], a[i]  
    return a
```

# Třídění výběrem – analýza

✖ **Proti:** Časová složitost  $\Theta(n^2)$

Data malého rozsahu (desítky prvků)

- lepší nežli BubbleSort

✓ **Pro:**

- Jen  $n - 1$  výměn
- jen  $O(n)$  zápisů do pole **a**

# Třídění vkládáním InsertionSort

Jako třídíme karty

- vezměte novou kartu z balíčku
- a postupným porovnáváním zprava doleva
- s již setříděnými kartami, které držíte v ruce
- vložte na správné místo



# Třídění vkládáním InsertionSort

```
def insertionSort(a):  
  
    for i in range(1, len(a)):  
        # vlož a[i] do setříděného  
        # a[0..i-1]  
        x, j = a[i], i  
        while j > 0 and a[j-1] > x:  
            a[j] = a[j-1]  
            j -= 1  
        a[j] = x  
  
    return a
```

Invariant: po  $i$ -tém kroku je  $a[0..i]$  setříděno

# Třídění vkládáním – analýza

- ✗ Časová složitost  $\Theta(n^2)$
- ✓ Vhodné pro data malého rozsahu (desítky prvků)
  - lepší nežli BubbleSort

## Srovnání s SelectionSort

- SelectionSort musí vždy projít zbývající prvky pro nalezení maxima
- InsertionSort může stačit jen jediné porovnání
- ✓ výhodné pro částečně setříděné vstupy
- ✓ v průměrném případě provede cca polovinu porovnání nežli SelectionSort

# Haldové třídění HeapSort

Datová struktura **binární halda** (**binary heap**)

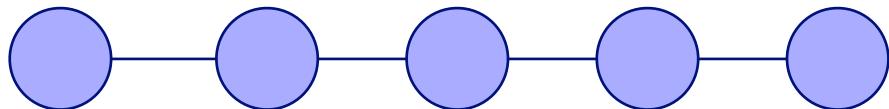
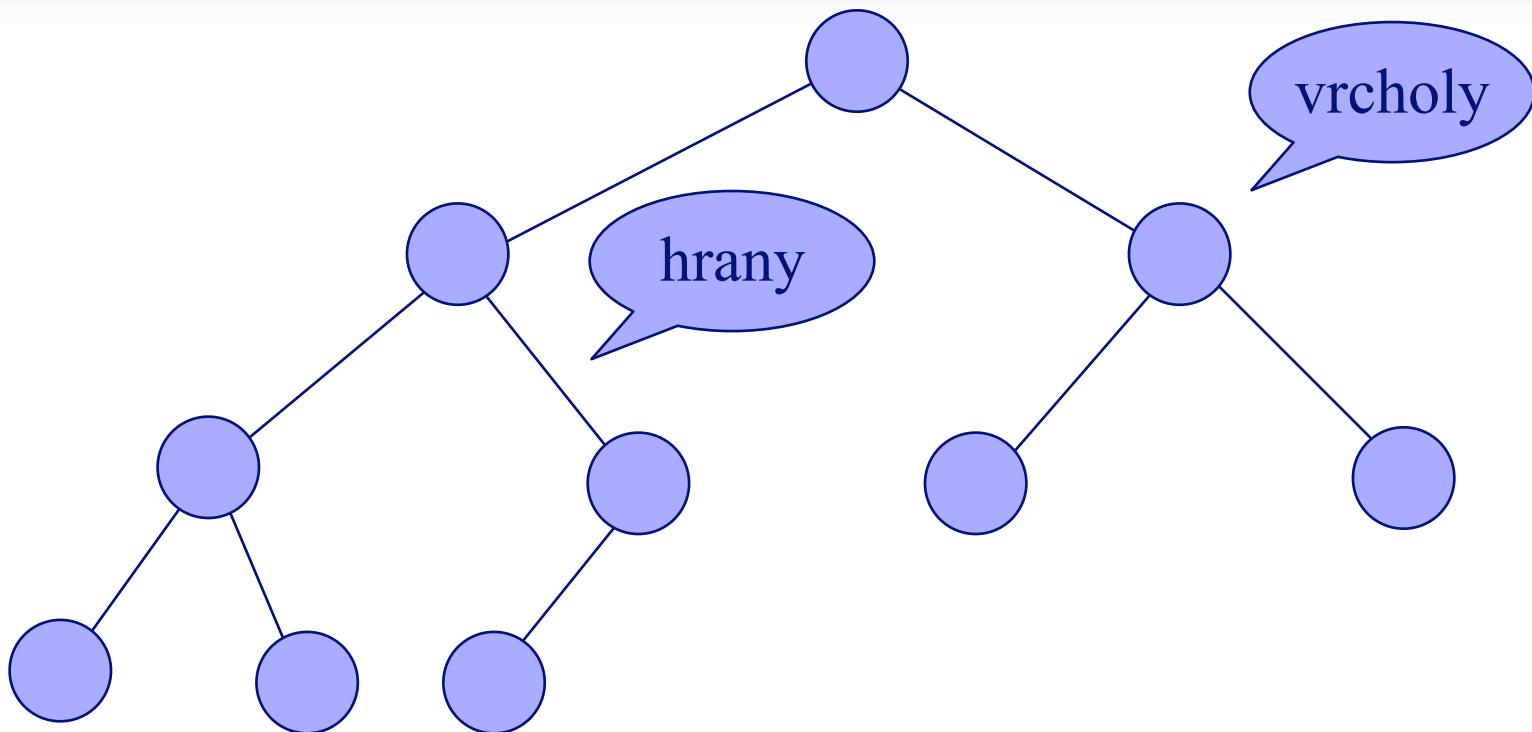
## Operace

- Přidej - vložení nového prvku
- OdeberMin - odebrání minimálního prvku
- lze provést v čase  $O(\log n)$
- $n$  = počet prvků uložených v haldě

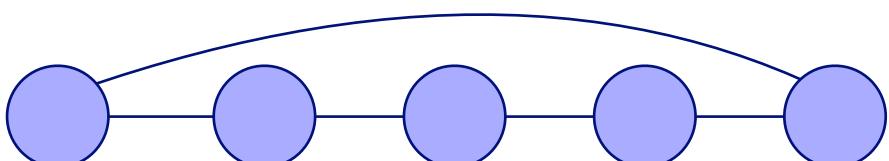
## HeapSort

- z  $n$  zadaných prvků postav haldu : čas  $O(n \log n)$
- $n$ -krát odeber minimum : čas  $O(n \log n)$
- třídění v čase  **$O(n \log n)$**

# Graf

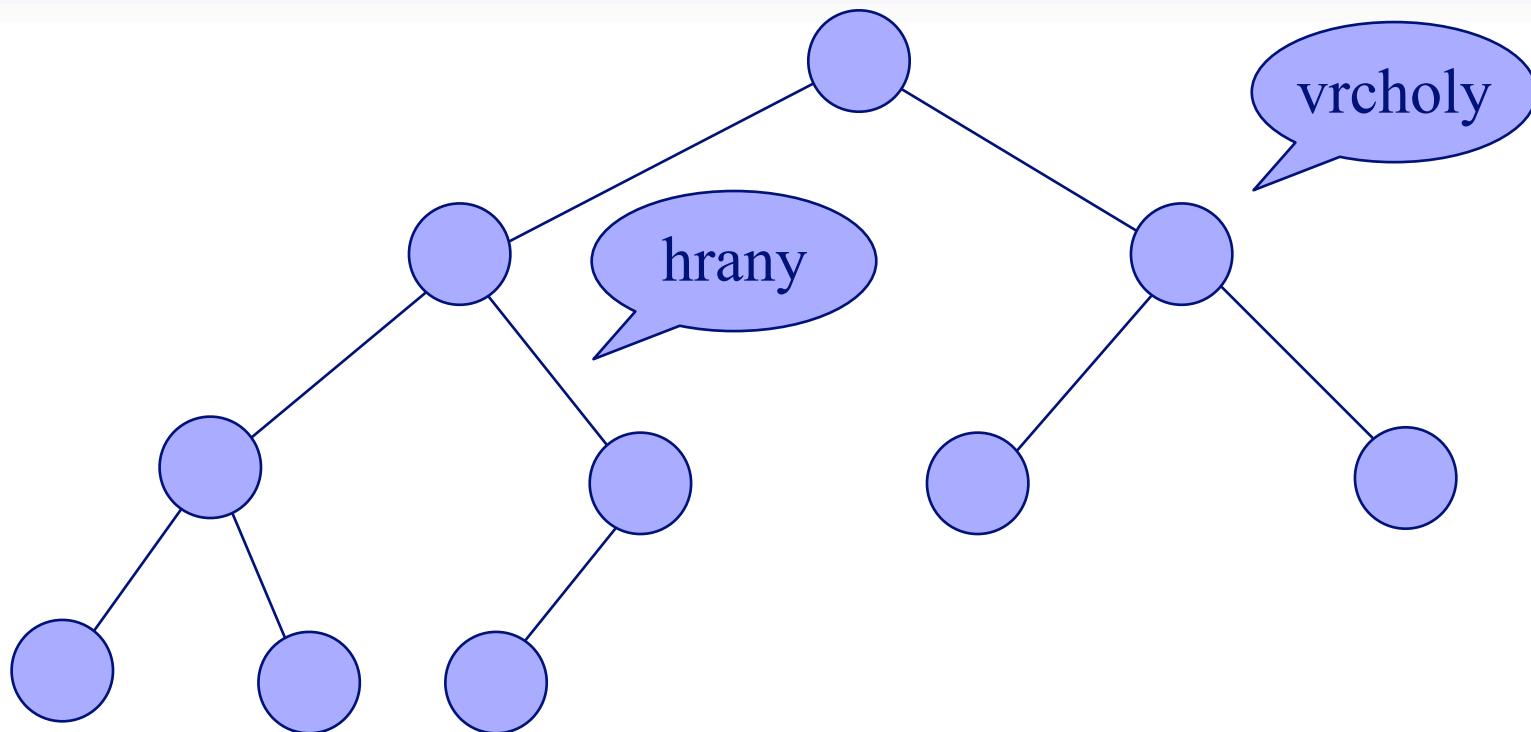


cesta (délky 4)



cyklus (délky 5)

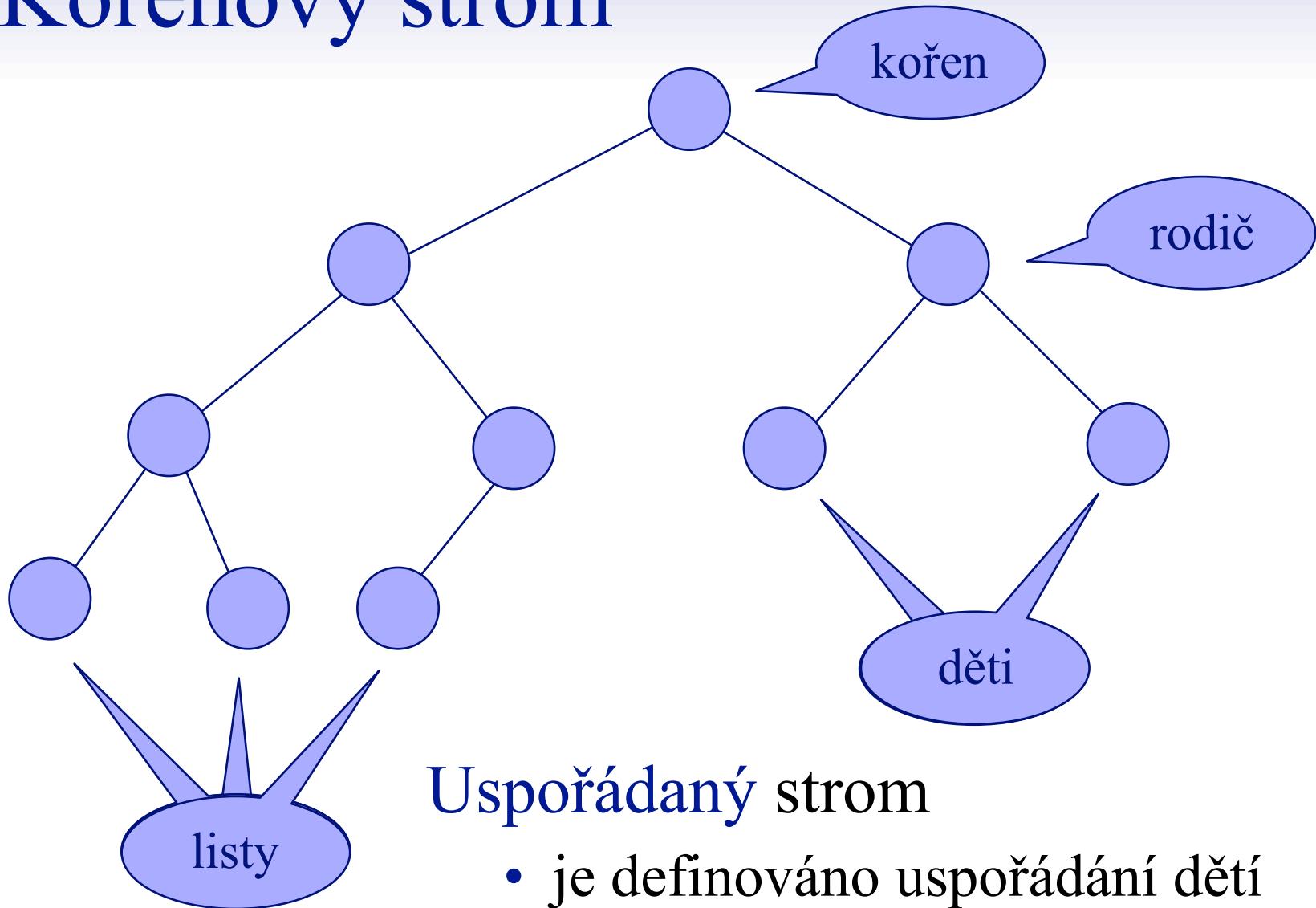
# Graf



Graf je

- souvislý - mezi každou dvojicí vrcholů existuje cesta
- strom - souvislý a acyklický

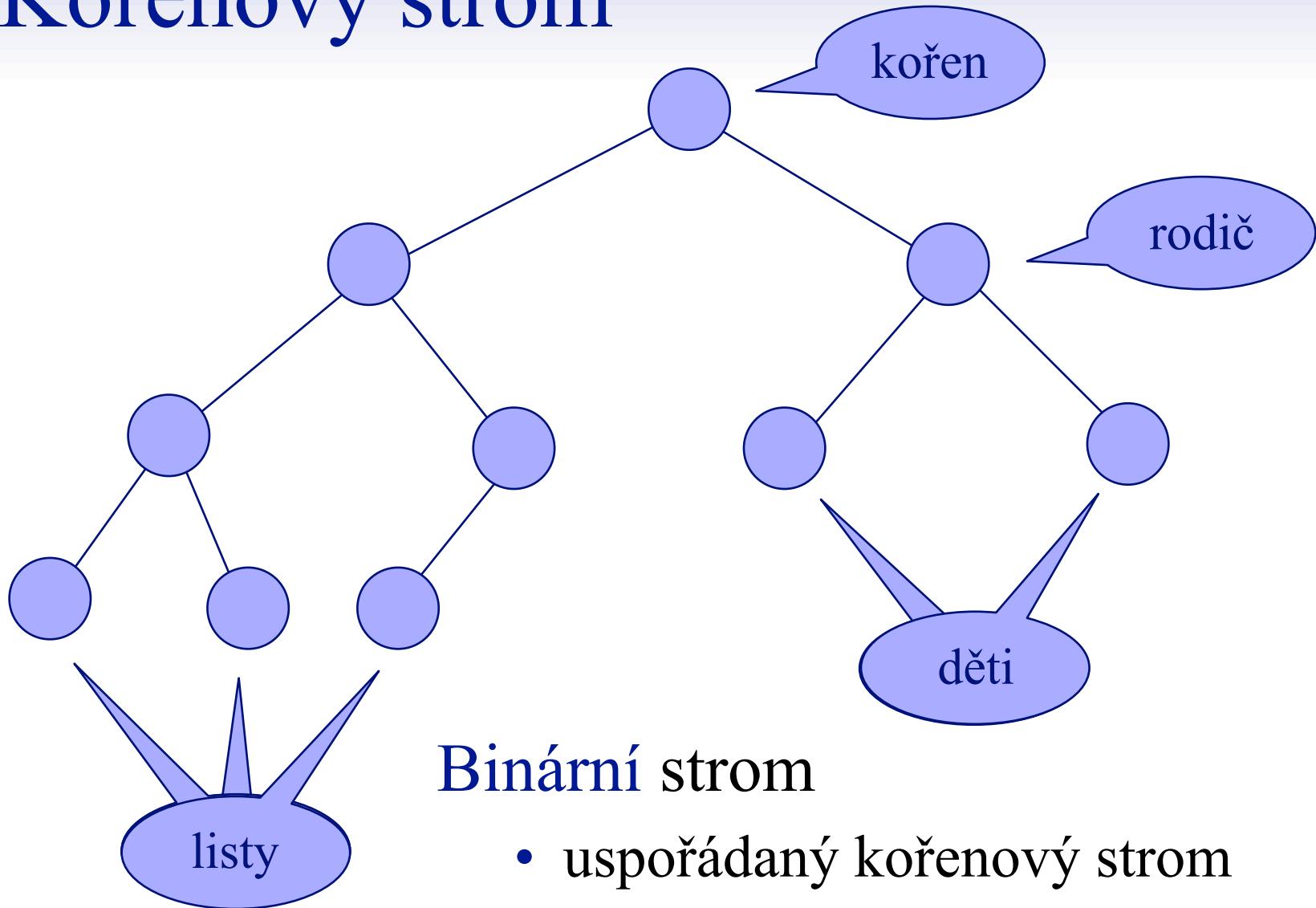
# Kořenový strom



## Uspořádaný strom

- je definováno uspořádání dětí
- pro každého rodiče

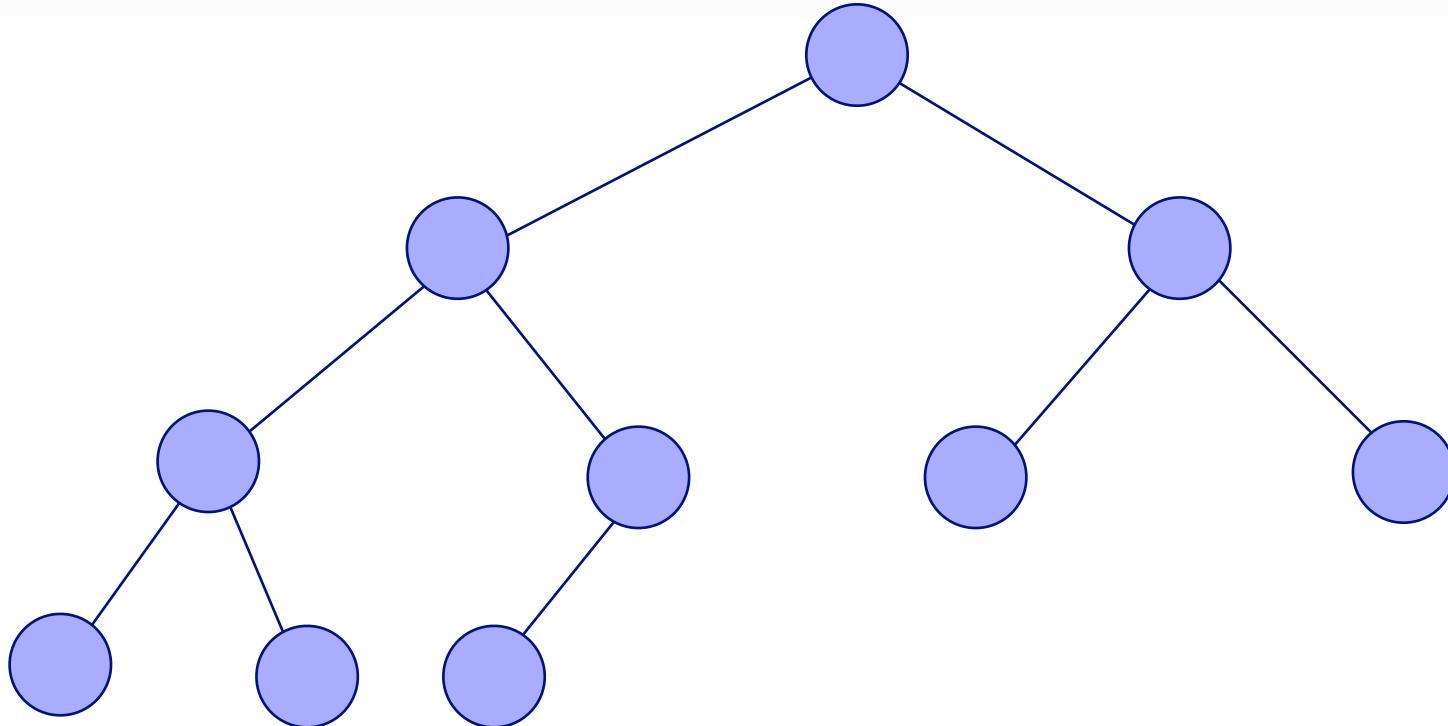
# Kořenový strom



## Binární strom

- uspořádaný kořenový strom
- každý rodič má nejvýše dvě děti

# Kořenový strom



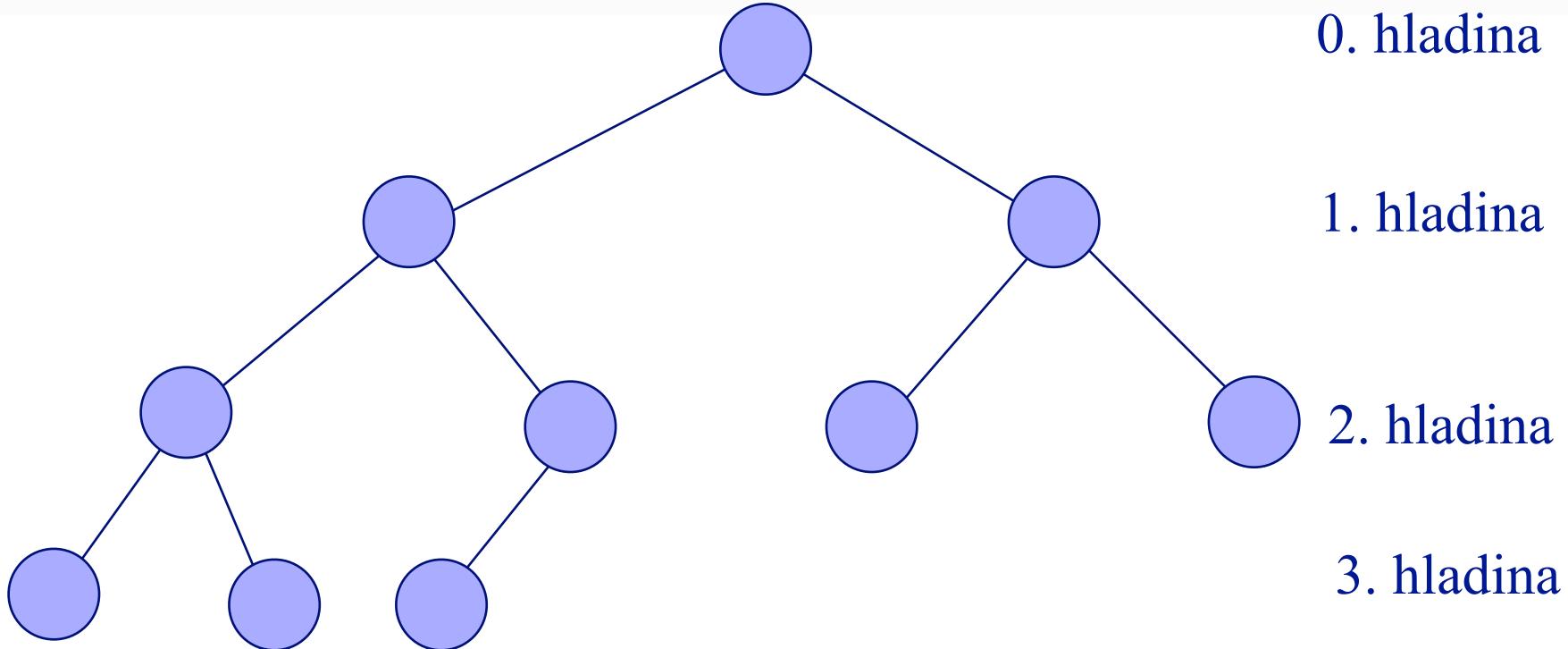
Vzdálenost vrcholů  $u$  a  $v$

- délka nejkratší cesty mezi  $u$  a  $v$

Výška stromu

- délka nejdelší cesty z kořene do listu

# Kořenový strom

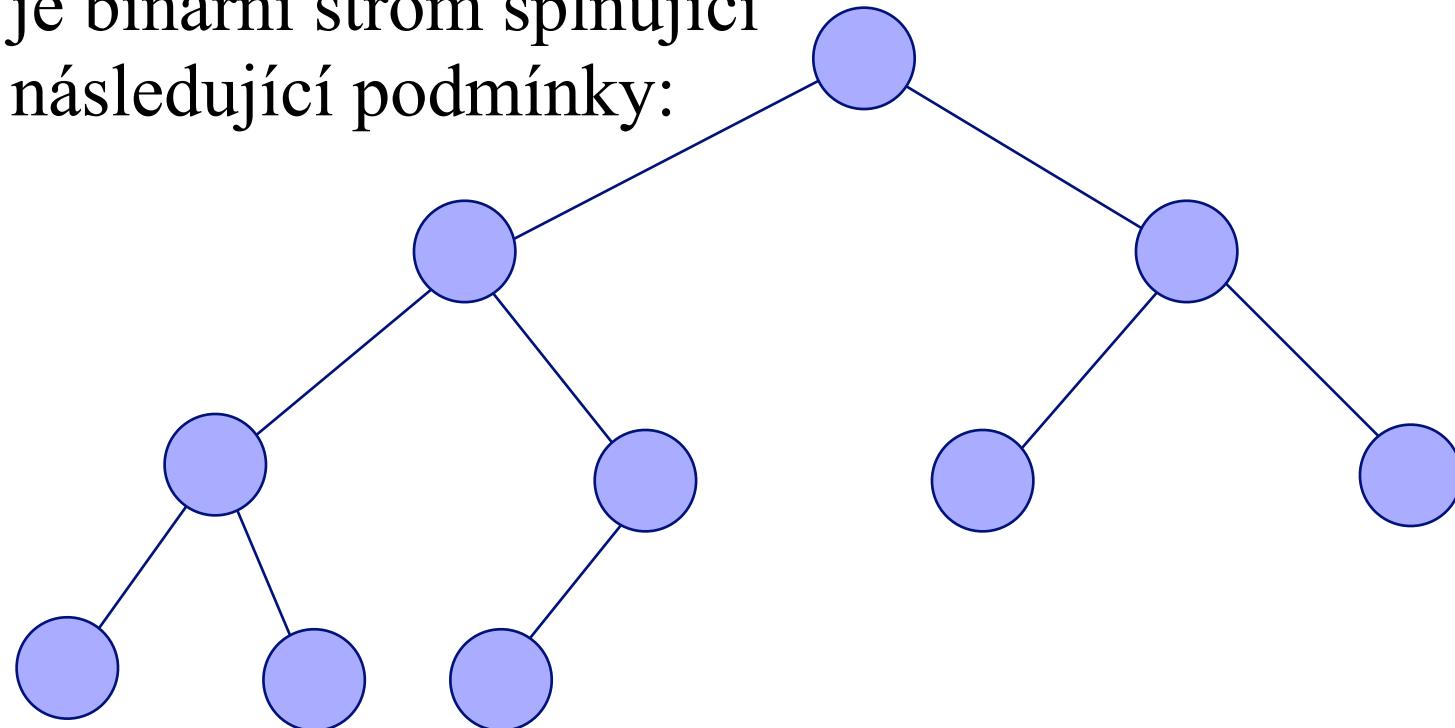


$i$ -tá hladina

- je tvořena vrcholy ve vzdálenosti  $i$  od kořene

# Binární halda

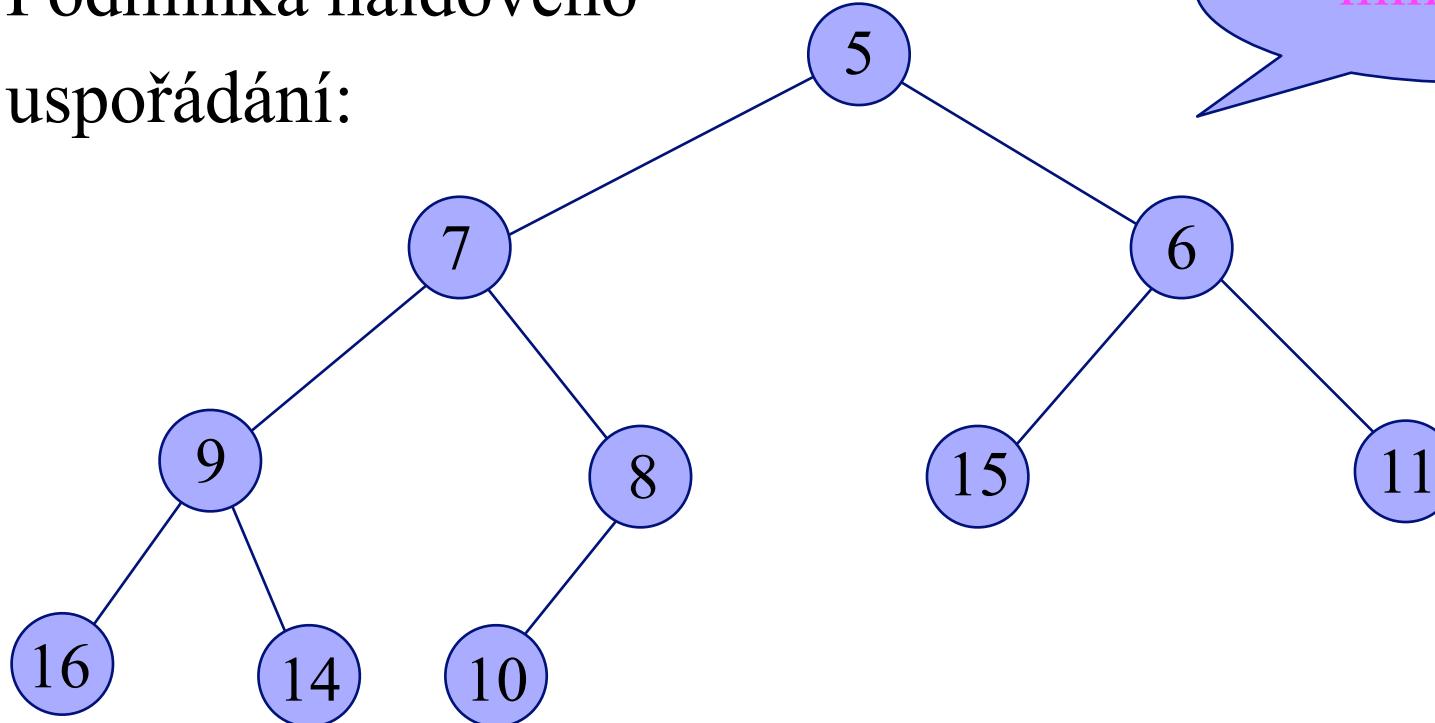
je binární strom splňující následující podmínky:



- v každé hladině od první do předposlední je max # vrcholů
- poslední hladina se zaplňuje zleva
- hodnoty uložené ve vrcholech splňují podmínu haldového uspořádání

# Binární halda

Podmínka haldového uspořádání:

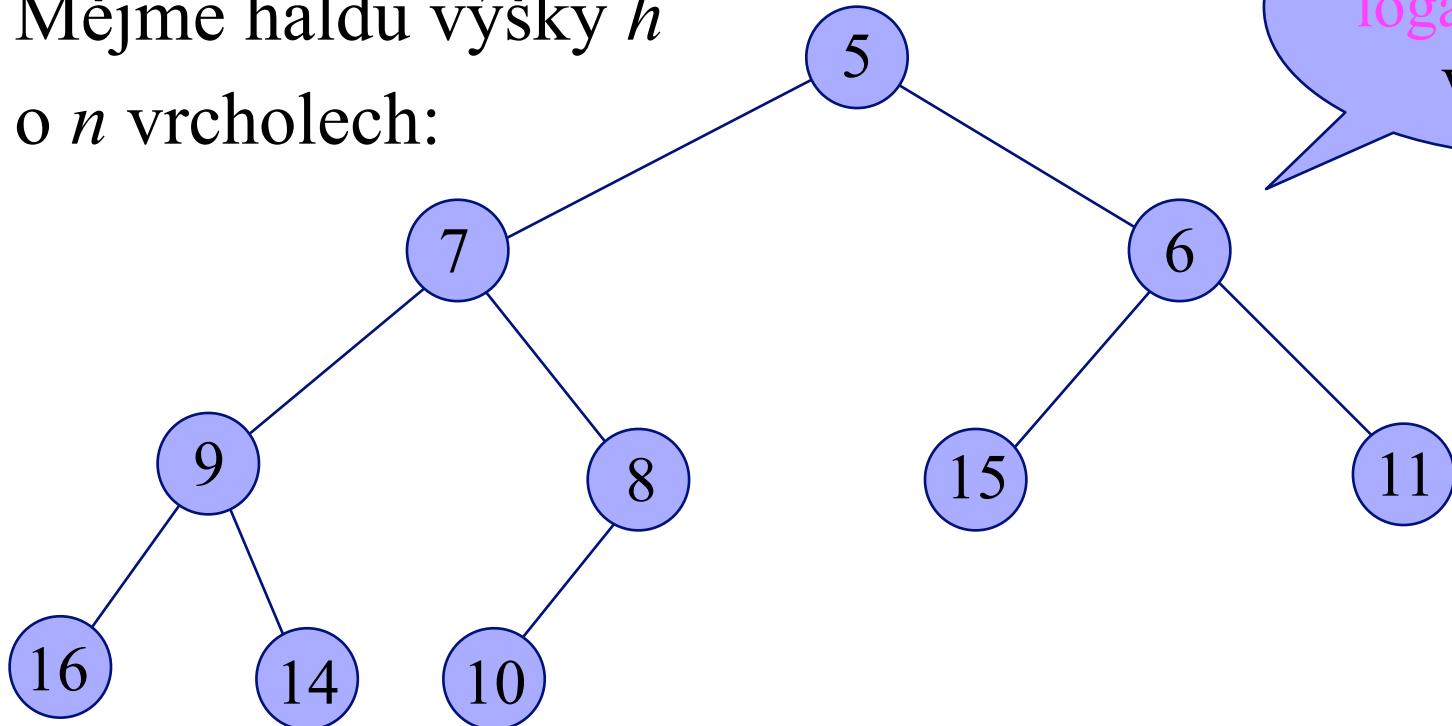


Pro každý vrchol platí, že hodnota v něm uložená je

- menší nebo rovna než hodnota v libovolném z jeho dětí (**min-halda**)
- Větší nebo rovna než hodnota v libovolném z jeho dětí (**max-halda**)

# Binární halda – vlastnosti

Mějme haldu výšky  $h$   
o  $n$  vrcholech:



halda má  
logaritmickou  
výšku !

Pak platí

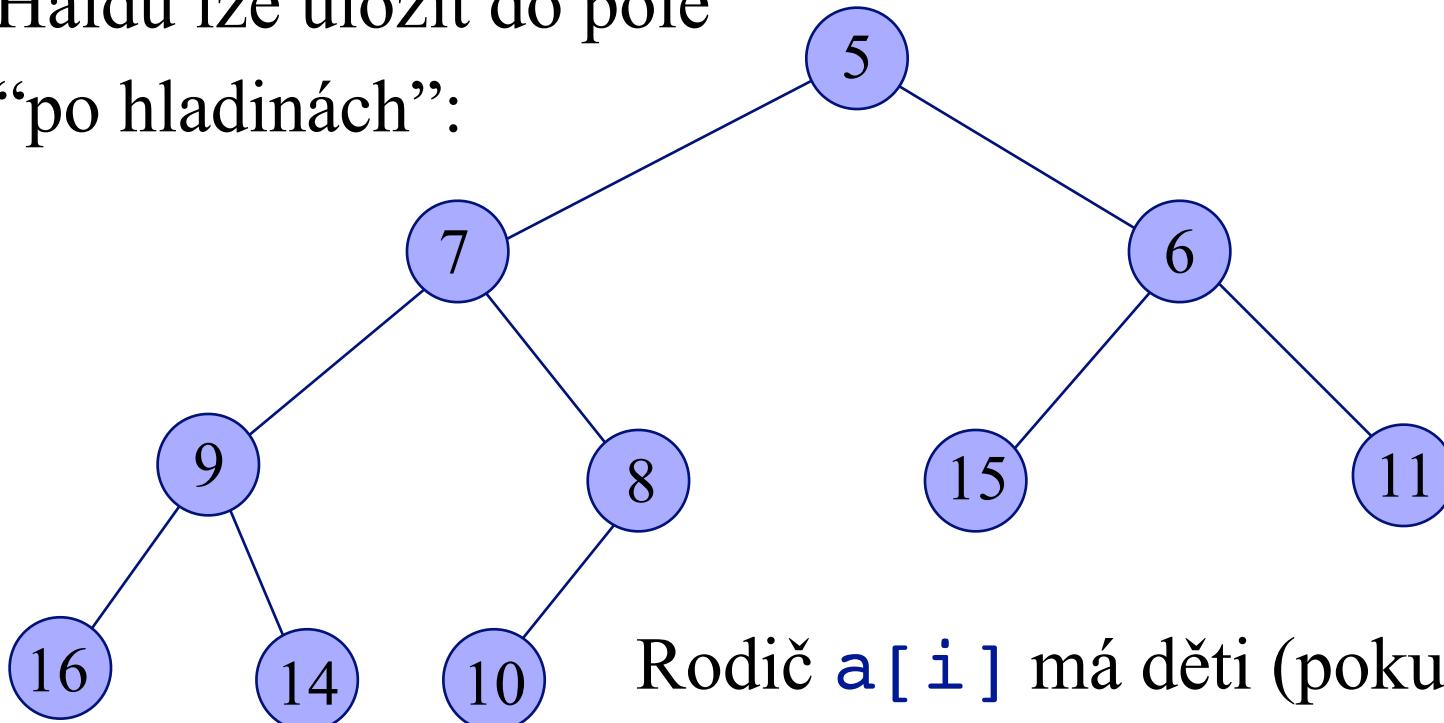
- na  $i$ -té hladině je  $2^i$  vrcholů ( $0 \leq i \leq h - 1$ )
- na poslední hladině je alespoň 1 vrchol

Tedy  $n \geq \sum_{i=0}^{h-1} 2^i + 1 = 2^h \Rightarrow h \leq \log_2 n$

# Binární halda – vlastnosti

Haldu lze uložit do pole

“po hladinách”:



Rodič  $a[i]$  má děti (pokud existují)

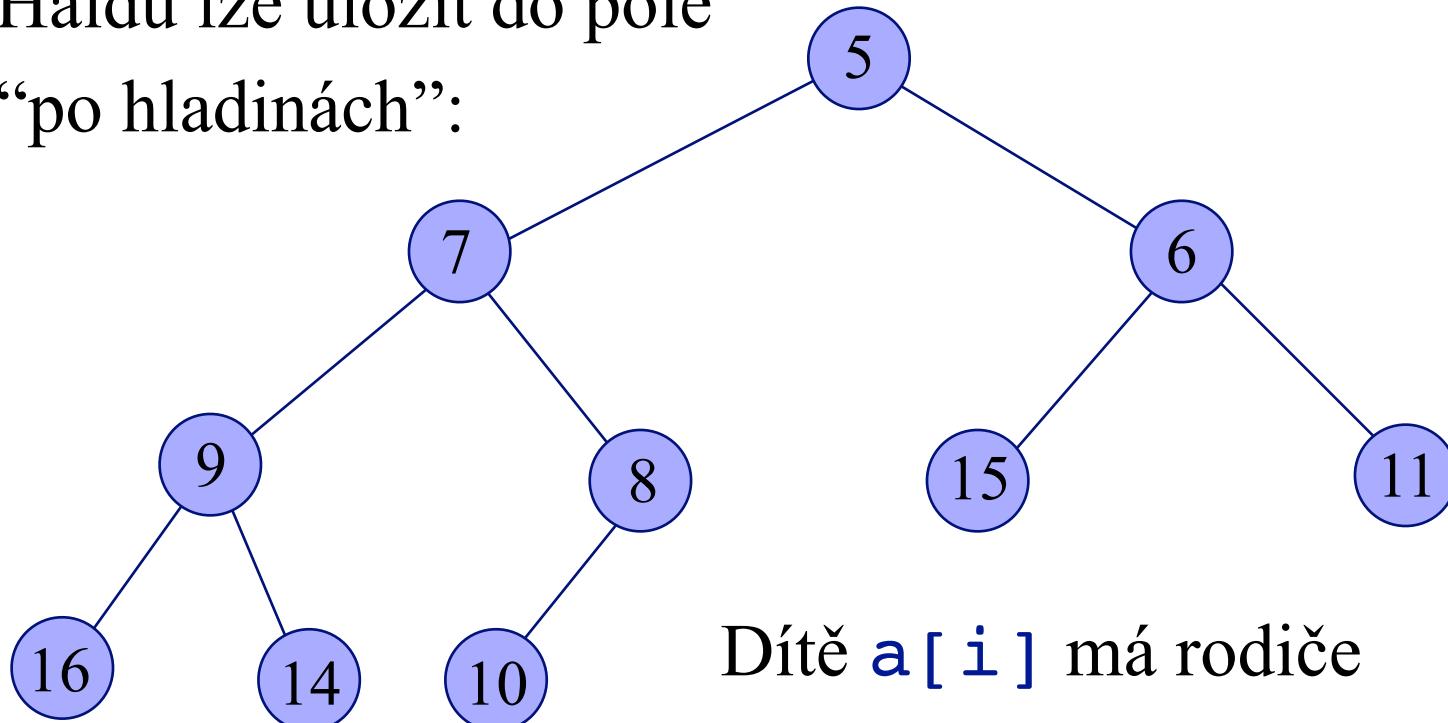
$a[2*i+1]$  a  $a[2*i+2]$

0	1	2	3	4	5	6	7	8	9
5	7	6	9	8	15	11	16	14	10

# Binární halda – vlastnosti

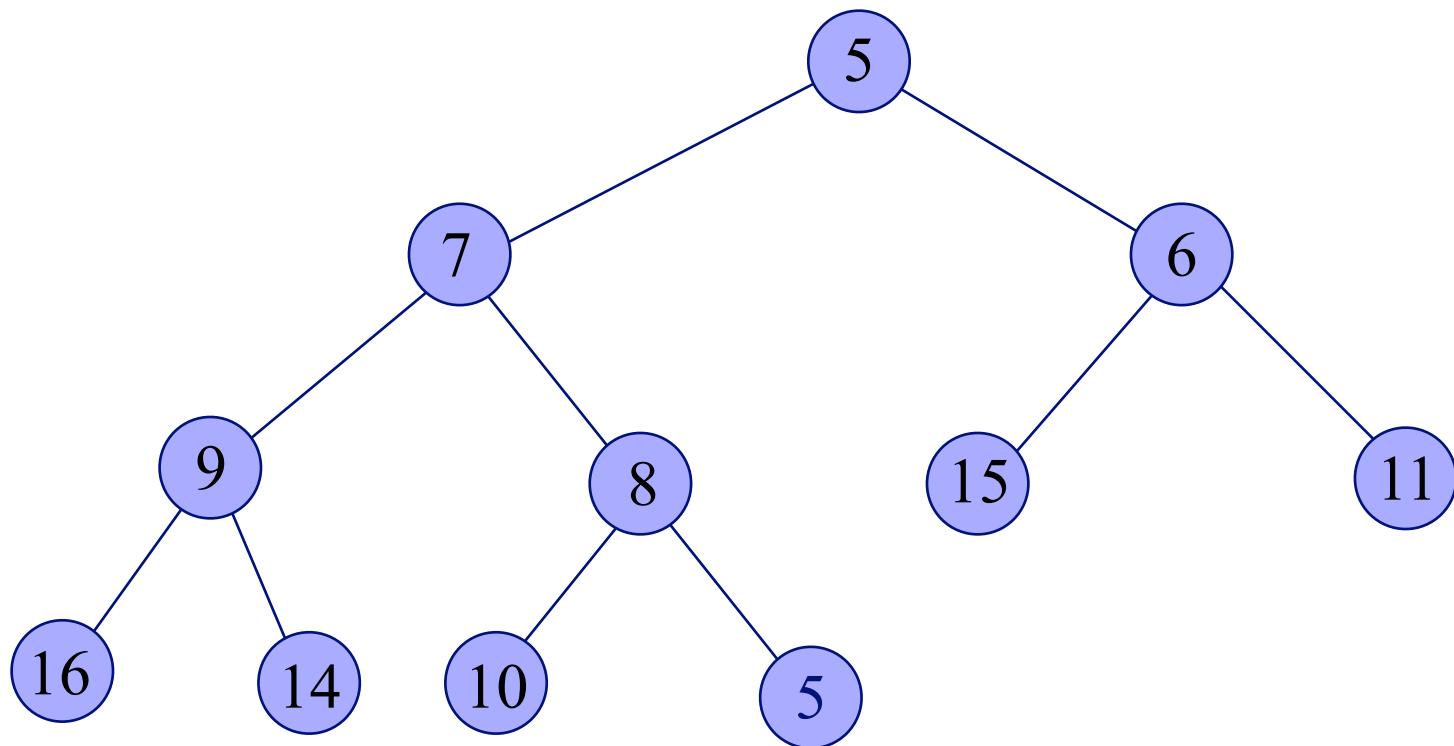
Haldu lze uložit do pole

“po hladinách”:



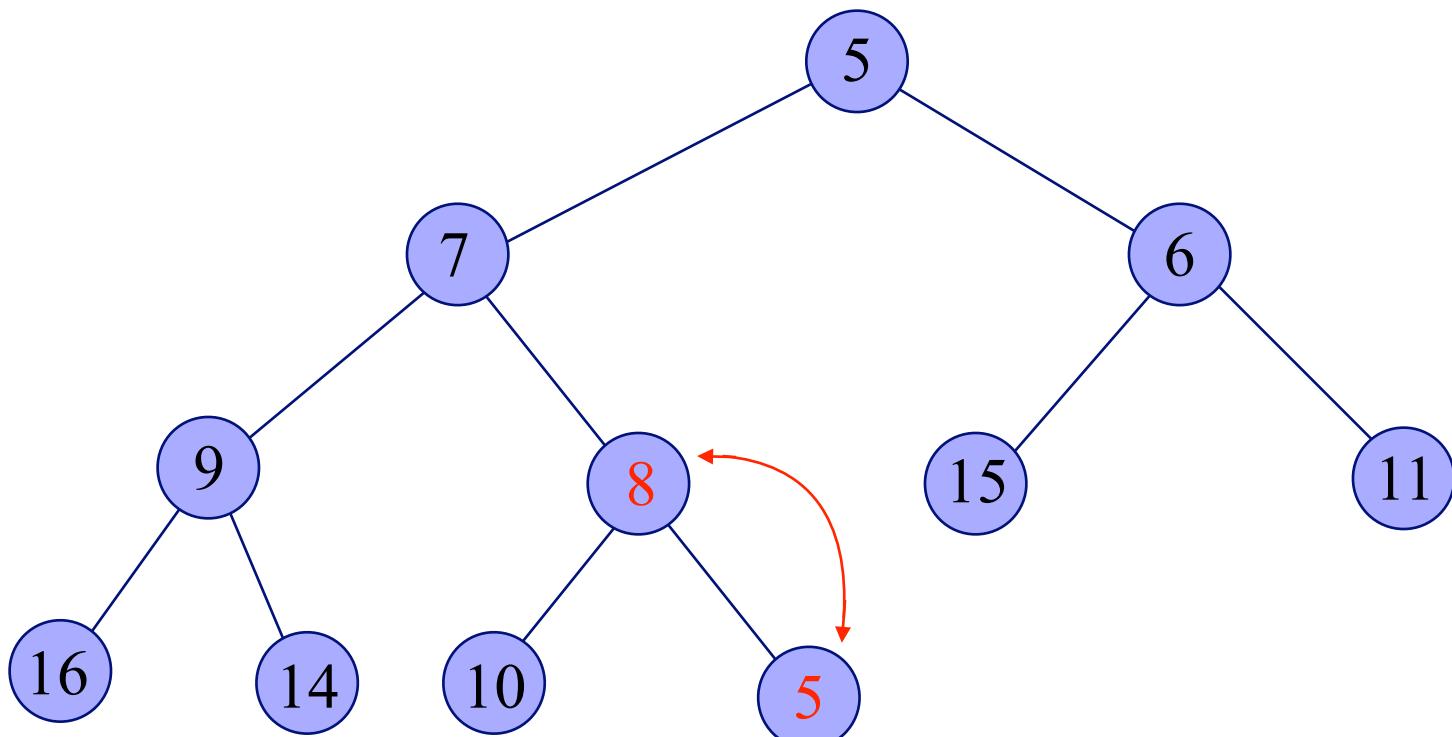
0	1	2	3	4	5	6	7	8	9
5	7	6	9	8	15	11	16	14	10

# Binární halda – Přidej



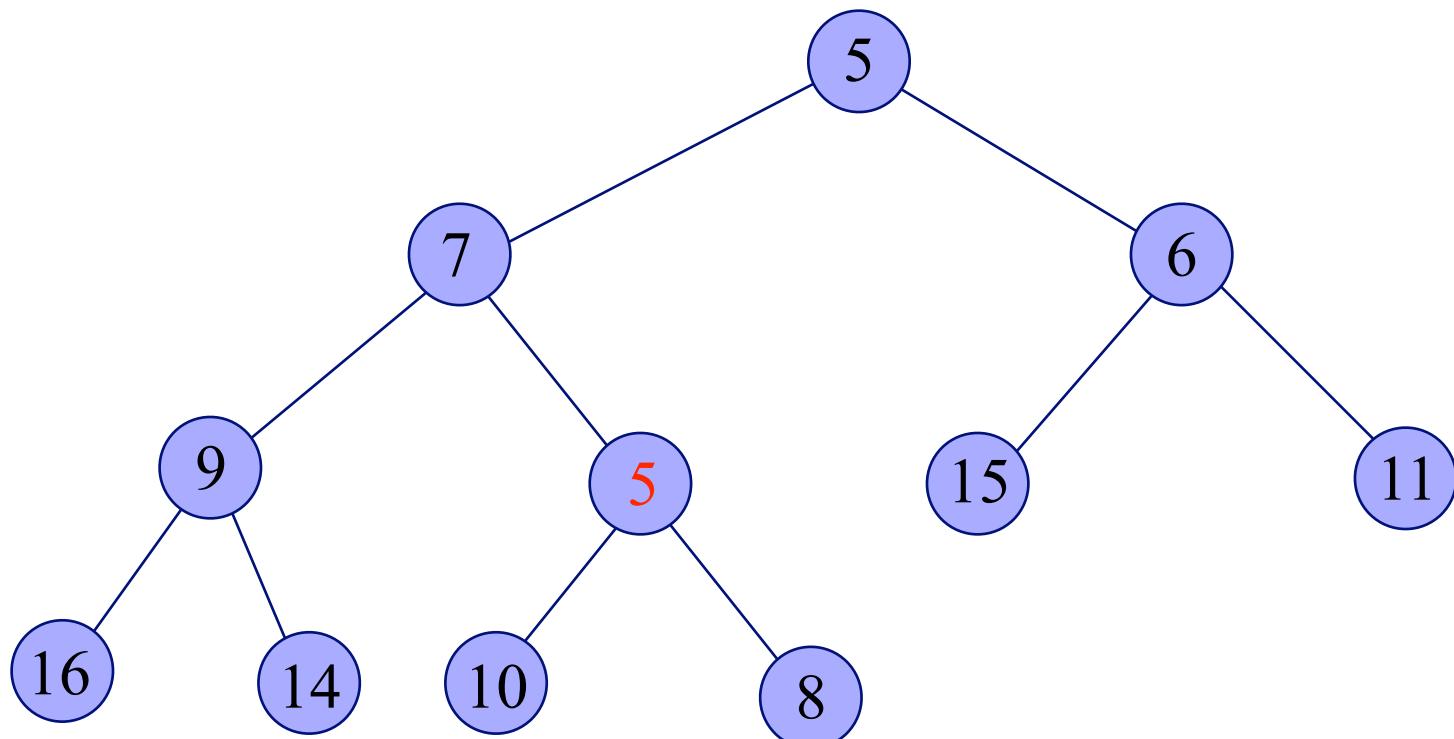
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	5

# Binární halda – Přidej



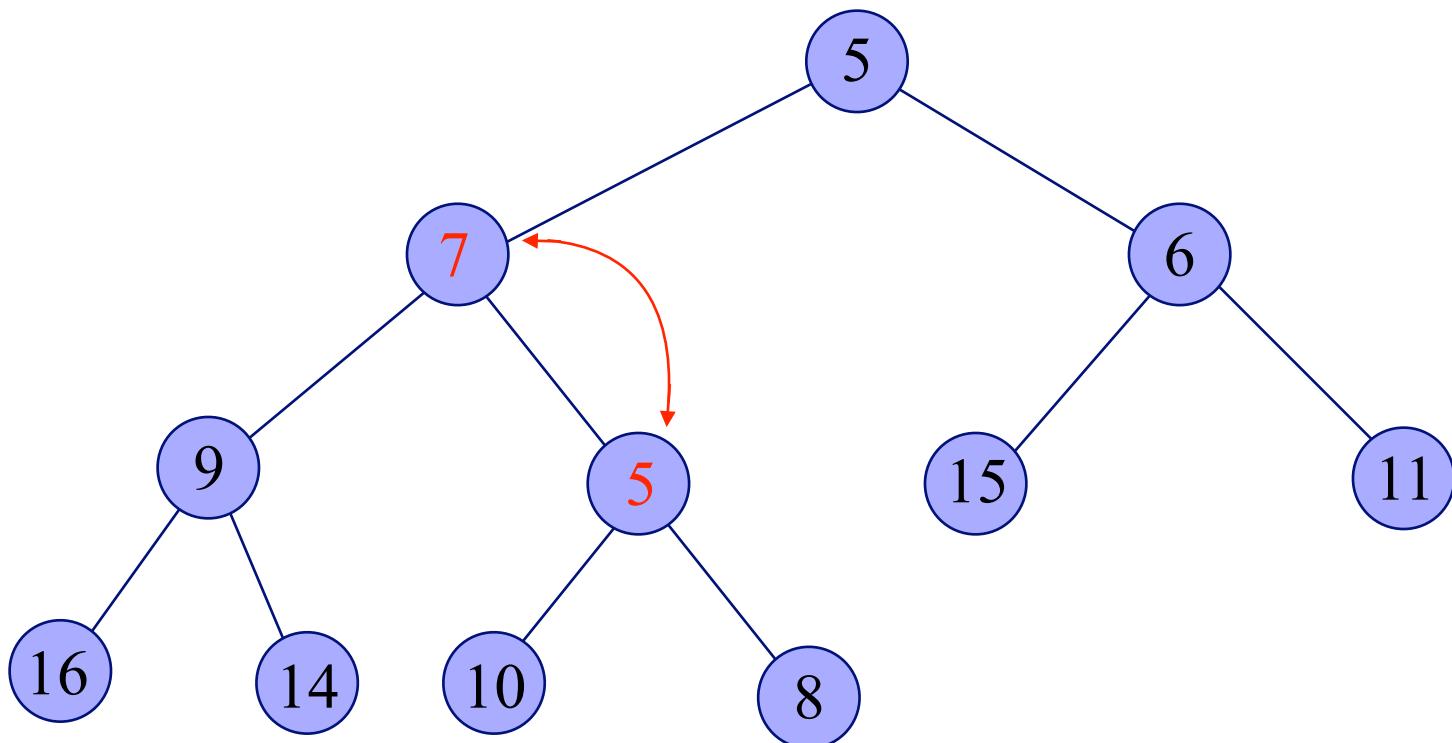
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	5

# Binární halda – Přidej



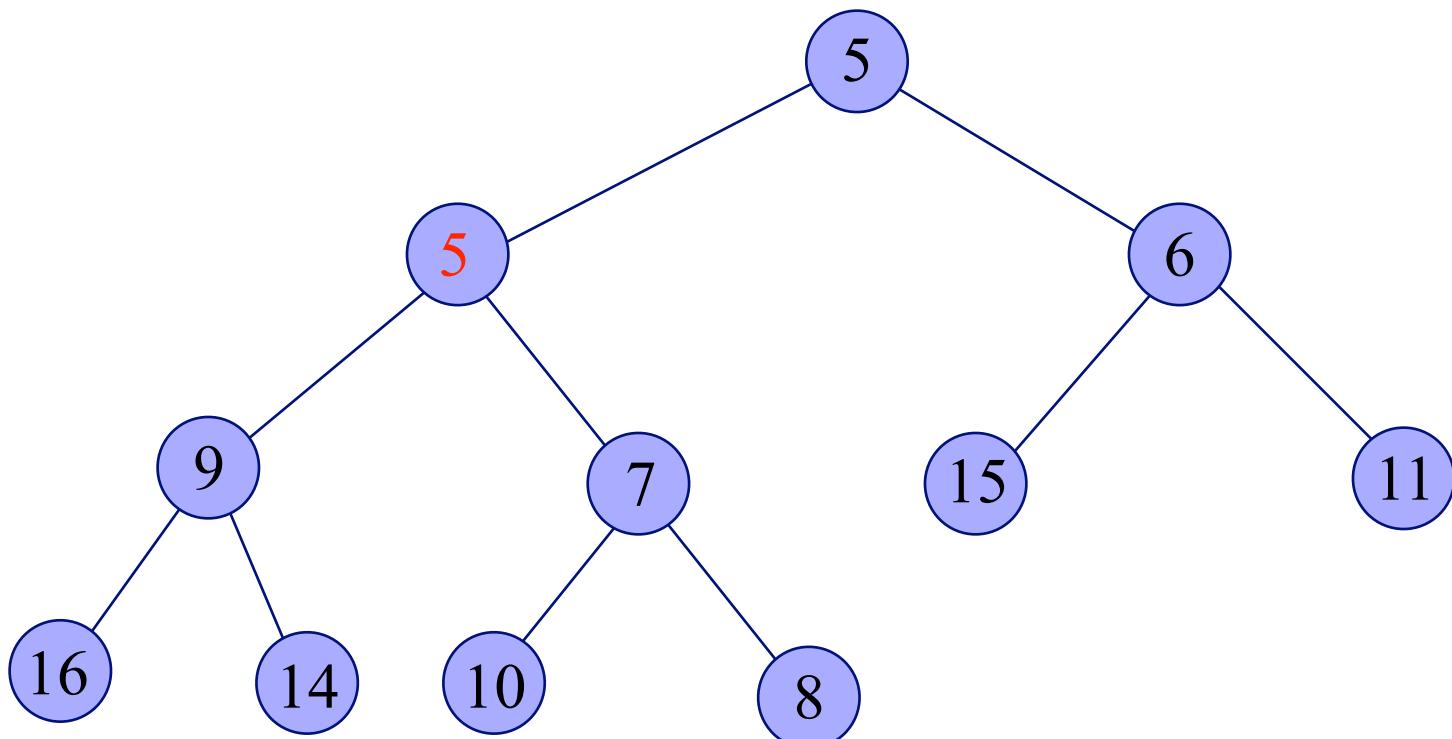
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	5	15	11	16	14	10	8

# Binární halda – Přidej



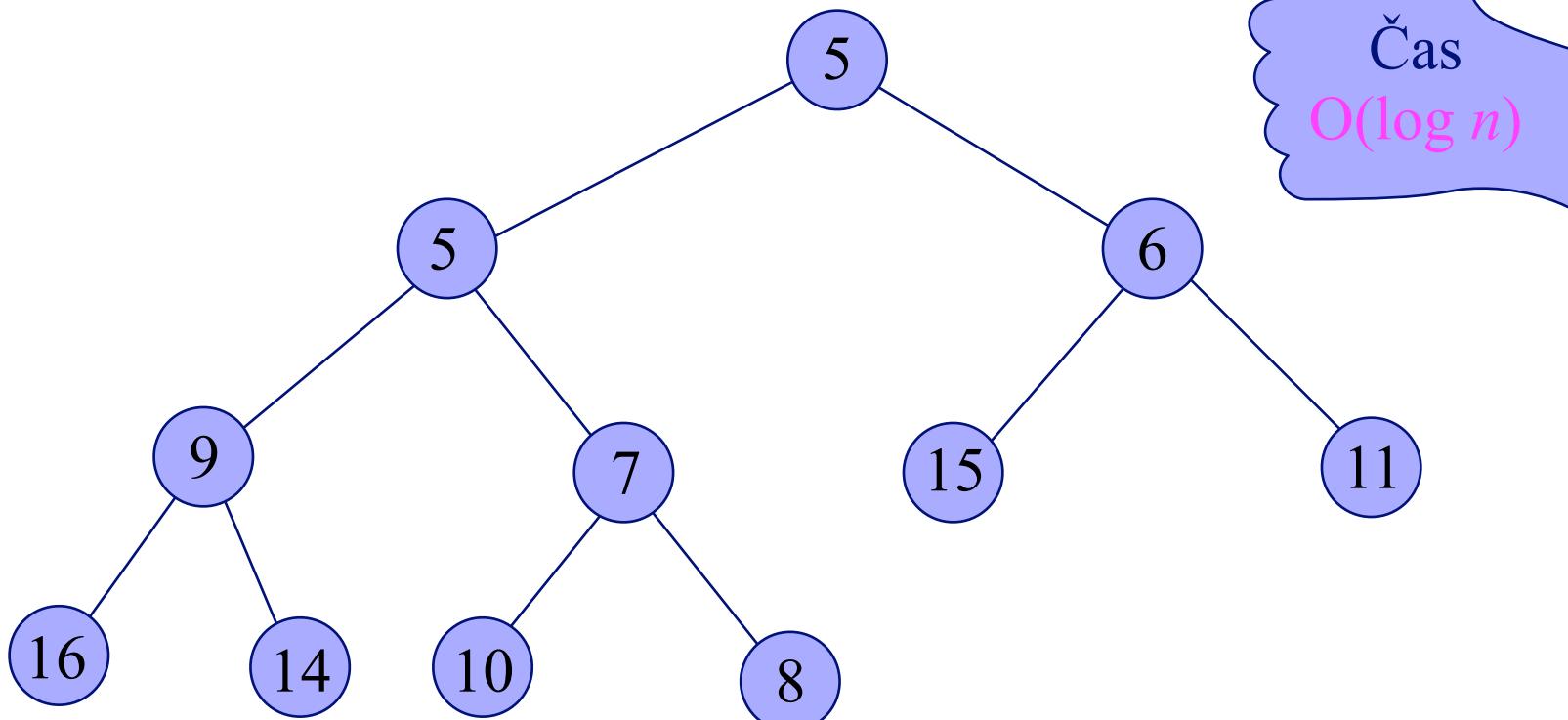
0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	5	15	11	16	14	10	8

# Binární halda – Přidej



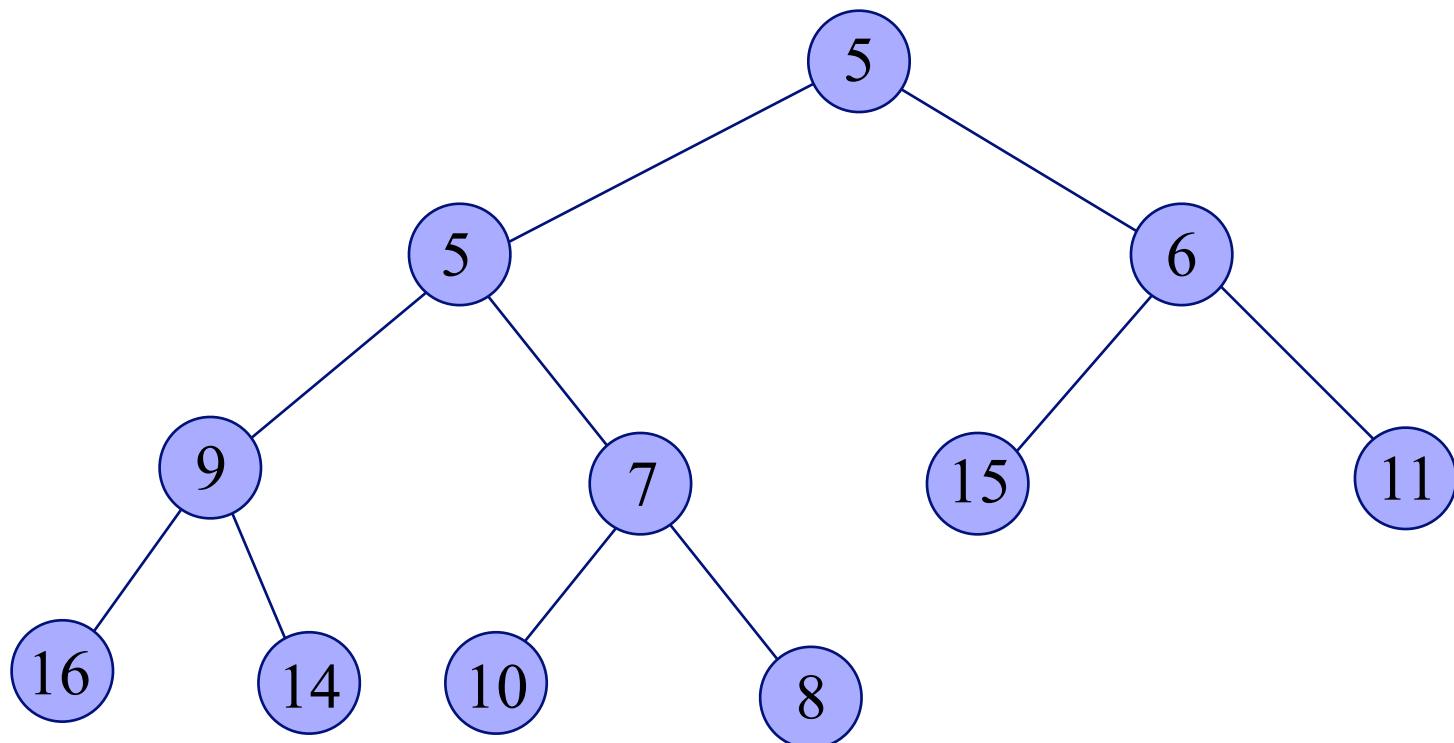
0	1	2	3	4	5	6	7	8	9	10
5	5	6	9	7	15	11	16	14	10	8

# Binární halda – Přidej



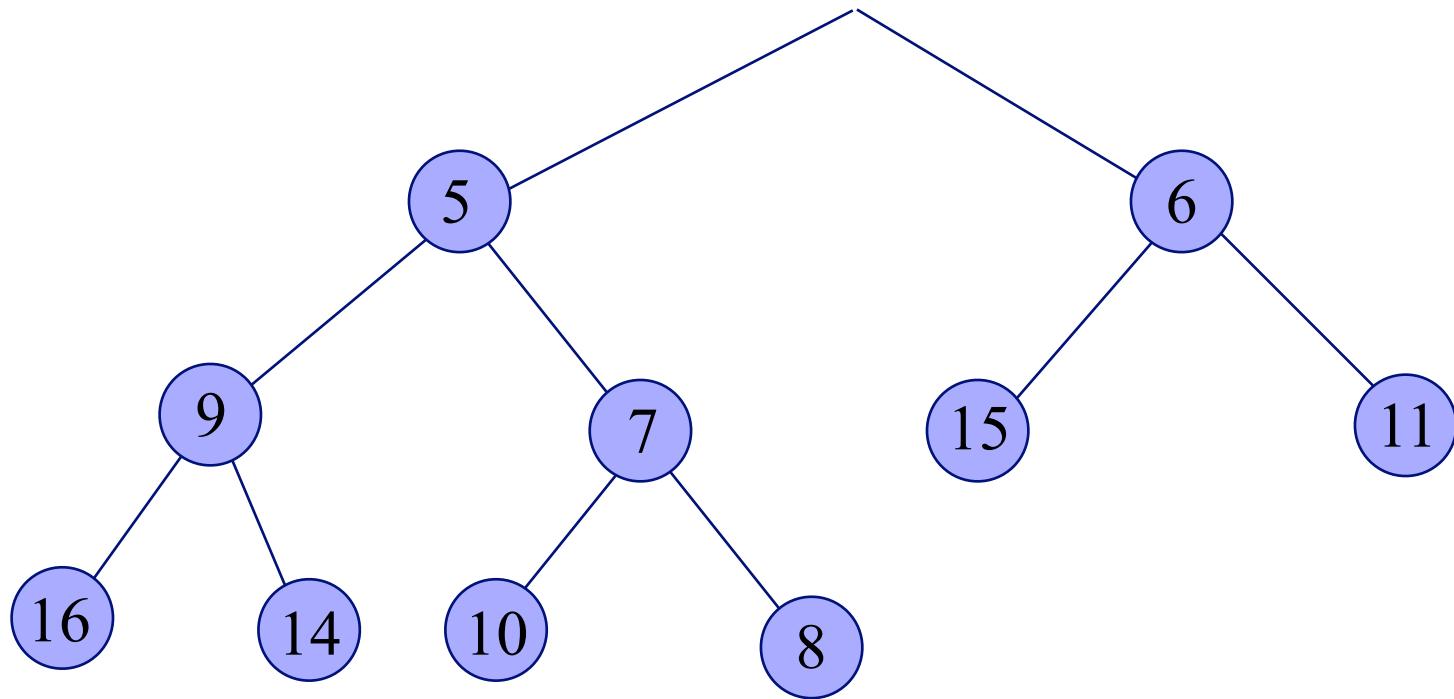
0	1	2	3	4	5	6	7	8	9	10
5	5	6	9	7	15	11	16	14	10	8

# Binární halda – OdeberMin



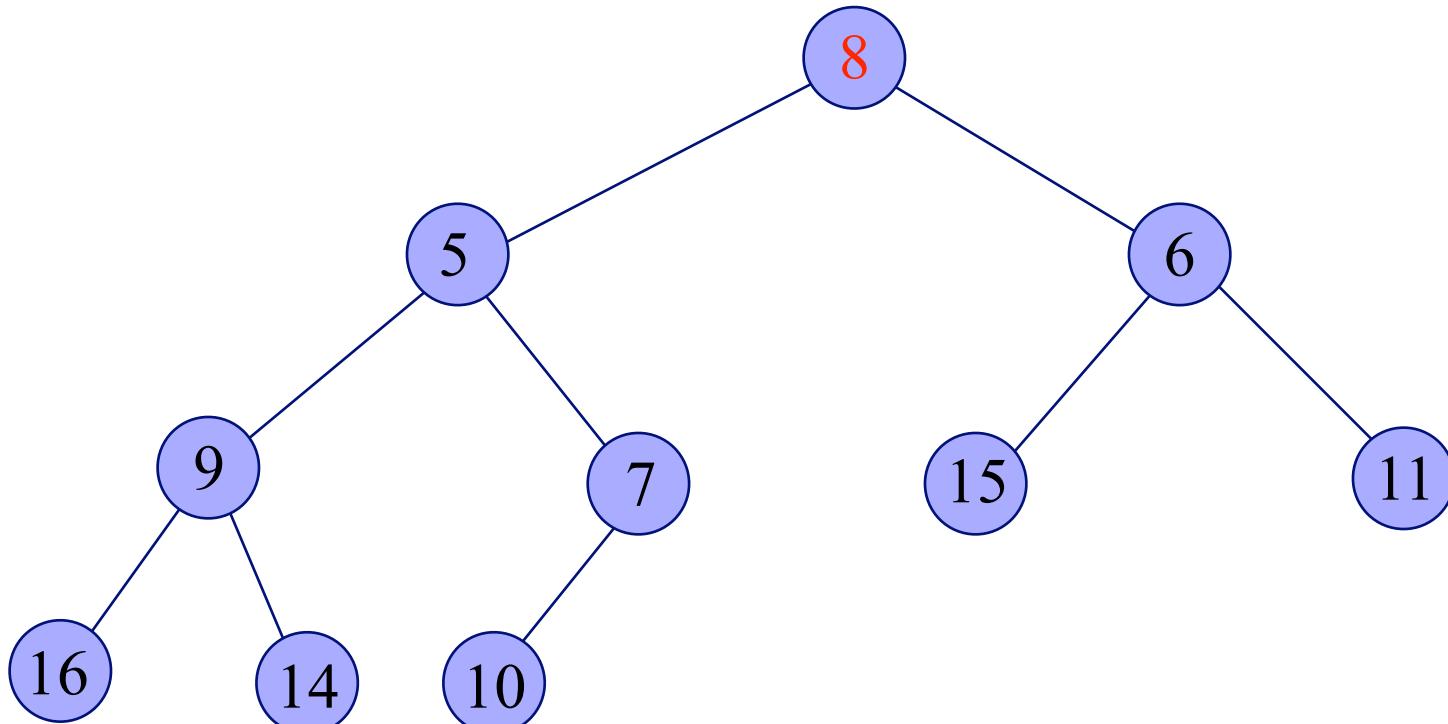
0	1	2	3	4	5	6	7	8	9	10
5	5	6	9	7	15	11	16	14	10	8

# Binární halda – OdeberMin



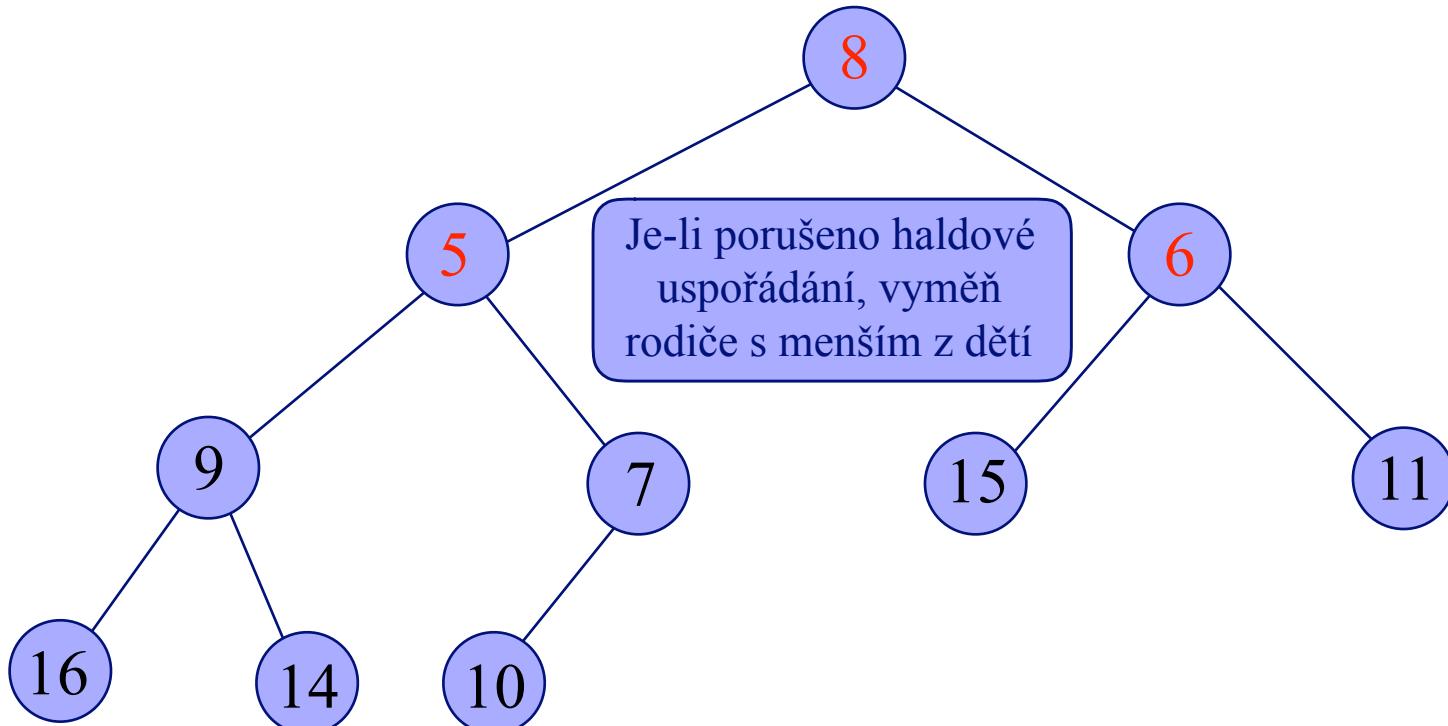
0	1	2	3	4	5	6	7	8	9	10
	5	6	9	7	15	11	16	14	10	8

# Binární halda – OdeberMin



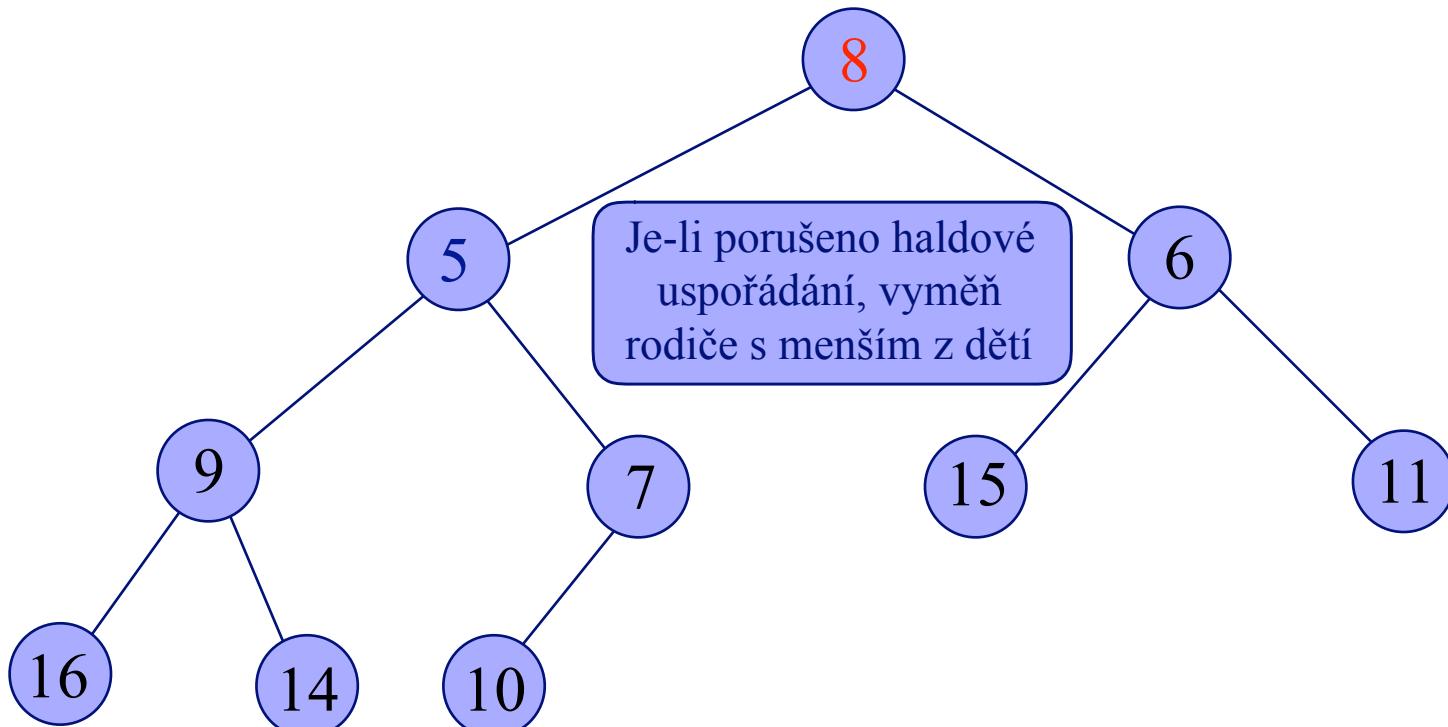
0	1	2	3	4	5	6	7	8	9	10
8	5	6	9	7	15	11	16	14	10	

# Binární halda – OdeberMin



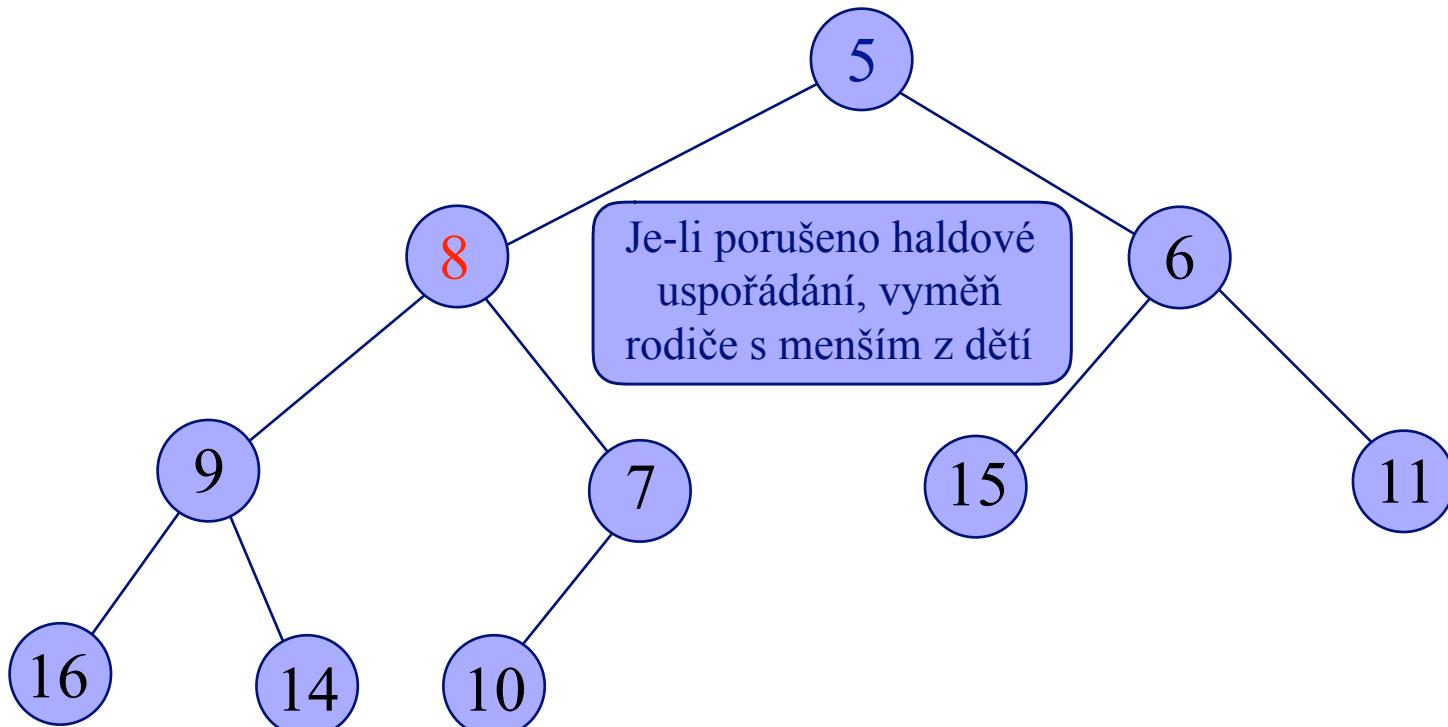
0	1	2	3	4	5	6	7	8	9	10
8	5	6	9	7	15	11	16	14	10	

# Binární halda – OdeberMin



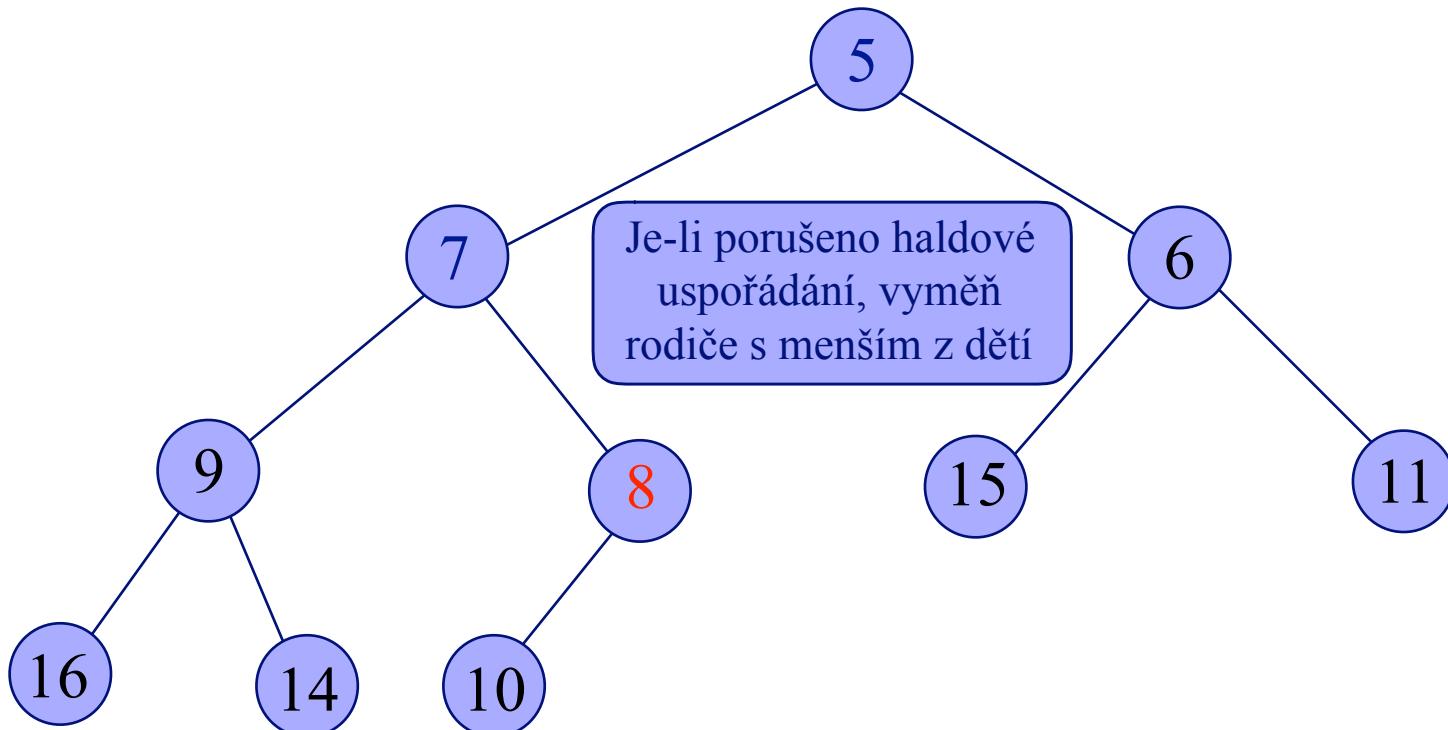
0	1	2	3	4	5	6	7	8	9	10
8	5	6	9	7	15	11	16	14	10	

# Binární halda – OdeberMin



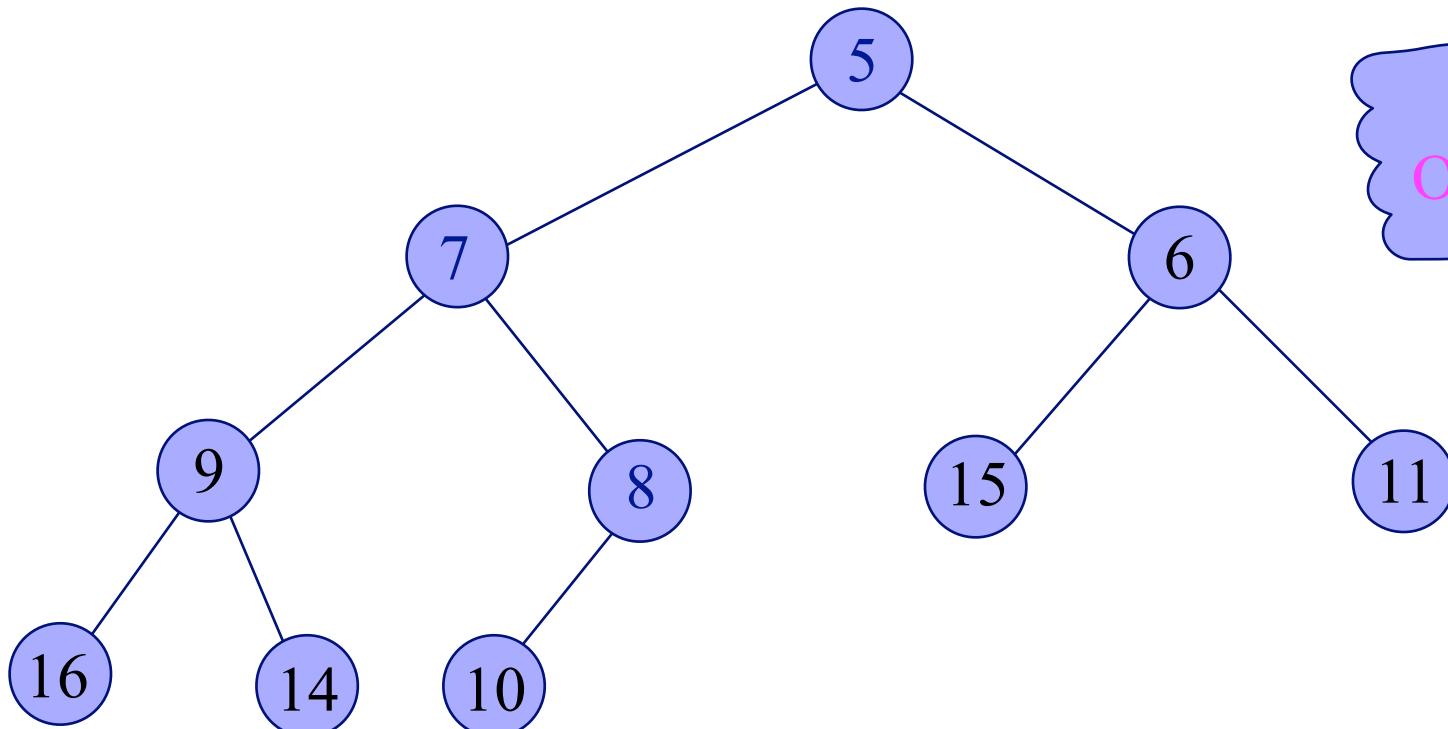
0	1	2	3	4	5	6	7	8	9	10
5	8	6	9	7	15	11	16	14	10	

# Binární halda – OdeberMin



0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	

# Binární halda – OdeberMin



0	1	2	3	4	5	6	7	8	9	10
5	7	6	9	8	15	11	16	14	10	

# Haldové třídění

Vstup: pole **a**

① Z prvků pole **a** vybuduj haldu

- začni s triviální haldou obsahující jen **a[ 0 ]**
- postupně vkládej **a[ 1 ]**, **a[ 2 ]**, ... pomocí **Přidej**
- v poli **a** je nyní uložena halda

② Z haldy postupně odebírej minima

- pomocí **OdeberMin**

## Problém

- jak zajistíme **třídění na místě (in situ)** ?



# Problémy



① V jazyce Python sestavte funkci

`heapSort(a)`

která setřídí prvky zadaného pole `a` vzestupně  
haldovým tříděním. Váš algoritmus byl měl třídit  
na místě, tj. může využívat jen konstantní  
pracovní paměť.

# Dolní odhad časové složitosti třídění

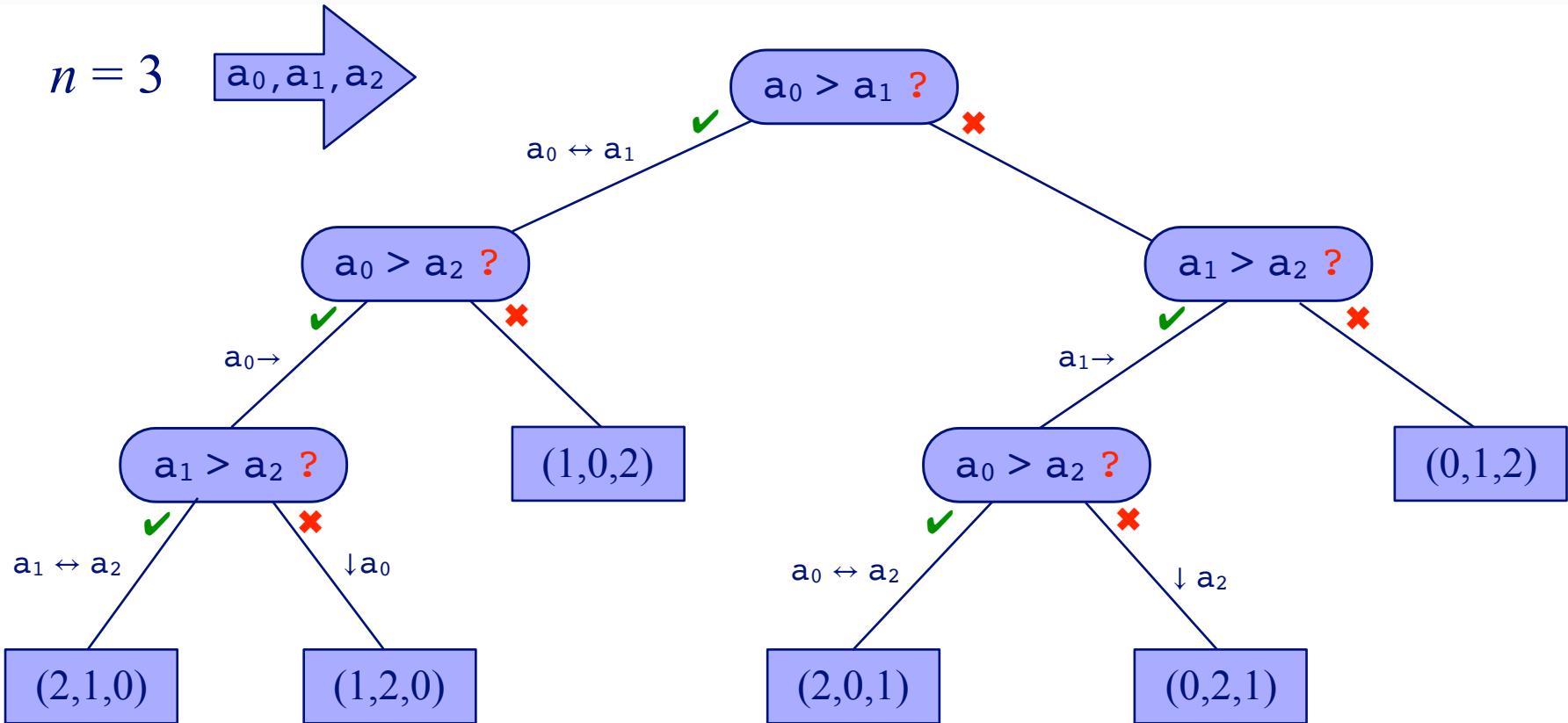
## Porovnávací třídící algoritmus

- třídí na základě porovávání dvojic prvků
  - »  $a_i < a_j, a_i > a_j, a_i \leq a_j, a_i \geq a_j, a_i = a_j$
- hodnoty tříděných prvků nevyužívá

## Rozhodovací strom

- reprezentuje průběh porovnání prováděných
- (konkrétním) porovnávacím algoritmem
- nad vstupem délky  $n$
- vnitřní vrcholy – porovnání
- listy – koncové stavy výpočtu

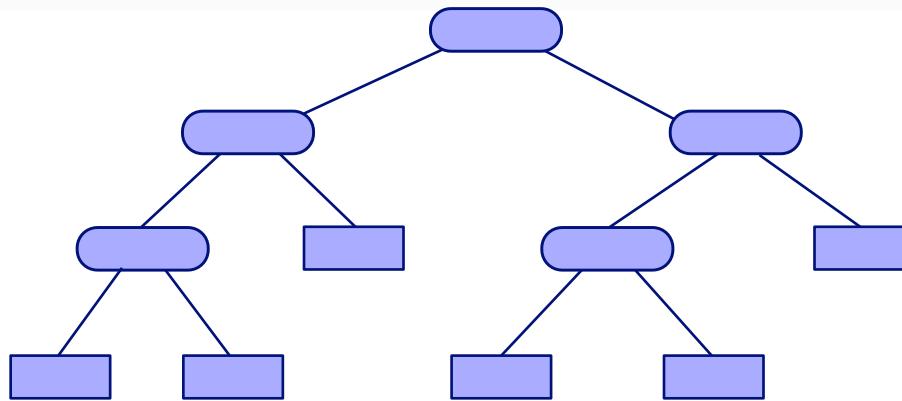
# Rozhodovací strom pro InsertionSort



# listů = # permutací množiny  $\{0,1,2\} = 3!$

👉 **Zobecnění:** Každý rozhodovací strom porovnávacího algoritmu nad vstupem délky  $n$  má alespoň  $n!$  listů.

# Rozhodovací stromy – vlastnosti



Cesta z kořene do listu = výpočet

- délka cesty = # porovnání

Maximální # porovnání nad vstupem délky  $n$

- = výška rozhodovacího stromu

# listů binárního stromu výšky  $h$  je  $\leq 2^h$

- # listů rozhodovacího stromu  $\geq n!$

$$\text{👉 } 2^h \geq n! \quad \text{👉 } h \geq \log_2(n!)$$

# Hrátky s funkcí $n!$

$$\begin{aligned} n! &= \sqrt{n!} \cdot \sqrt{n!} = \sqrt{n \cdot (n-1) \cdot \dots \cdot 1} \cdot \sqrt{1 \cdot 2 \cdot \dots \cdot n} \\ &= \sqrt{n \cdot 1} \cdot \sqrt{(n-1) \cdot 2} \cdot \dots \cdot \sqrt{1 \cdot n} \end{aligned}$$

Pro  $0 \leq k \leq n-1$  platí:

$$\begin{aligned} (n-k)(k+1) &= nk + n - k^2 - k \\ &= n + k(n-1-k) \geq n \end{aligned}$$

$$n! \geq (\sqrt{n})^n = n^{n/2}$$

$$\log_2(n!) \geq \log_2(n^{n/2}) = \frac{1}{2} \cdot n \cdot \log_2 n$$

👉 Maximální počet porovnání nad vstupem délky  $\textcolor{violet}{n}$  je  
 $\geq \text{konstanta} \cdot \textcolor{violet}{n} \cdot \log_2 \textcolor{violet}{n}$

# Složitost problému třídění

- 👉 **Závěr:** Každý porovnávací třídící algoritmus provede v nejhorším případě  $\Omega(n \log n)$  porovnání.
- 👉 **Důsledek:** Algoritmus haldového třídění má asymptoticky optimální časovou složitost.
- 👉 **Důsledek:** Složitost problému vnitřního třídění porovnávacím algoritmem je  $\Theta(n \log n)$ .

# Problémy



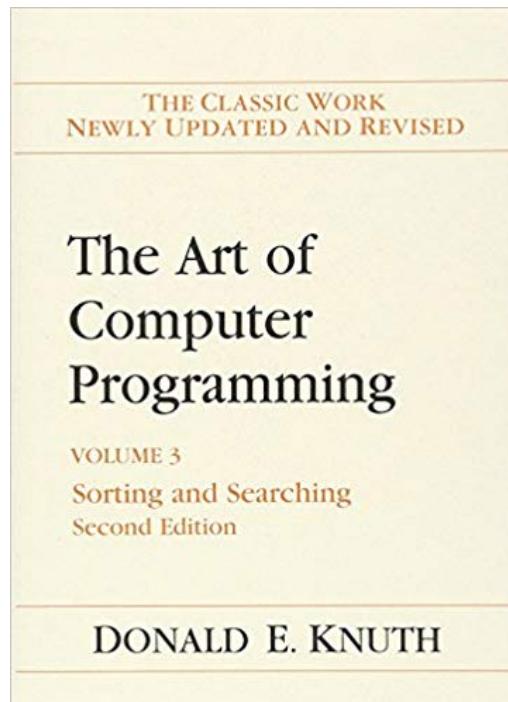
② Uvažte třídící algoritmus, který pro libovolné dva prvky  $a_i, a_j$  na vstupu může na dotaz  $a_i ? a_j$  obdržet jednu ze tří možných odpovědí:

- $a_i < a_j$
- $a_i > a_j$
- $a_i = a_j$

Bude náš dolní odhad platit i tomto případě?

# Algoritmizace

## Třídění v lineárním čase



# Osnova

- ❖ Třídění počítáním (CountingSort)
- ❖ Přihrádkové třídění (BucketSort)
- ❖ Radixové třídění (RadixSort)

# Třídění v lineárním čase

Předpoklad: tříděné hodnoty (klíče) jsou přirozená čísla z množiny  $\{0, 1, \dots, k\}$ .

## Třídění počítáním CountingSort

- pro každý prvek  $x$  spočítej # prvků  $x$
- zapiš  $x$  na správnou pozici v setříděné posloupnosti na výstup

Vstup: pole **a** splňující předpoklad výše

Výstup: pole **b** obsahující prvky pole **a** uspořádané vzestupně

Pomocná datová struktura: pole **c** [ 0 .. k ]

- pro uložení čítačů výskytů tříděných hodnot

# Třídění počítáním CountingSort

```
def countingSort0(a,k):  
    c = [0] * (k+1)  
    for x in a:  
        c[x] += 1 # četnosti  
    i = 0  
    for x in range(k+1):  
        for cetnost in range(c[x]):  
            a[i] = x  
            i += 1  
  
    return a
```

- ✖ Nefunguje, jsou-li tříděné prvky strukturované
  - klíč + satelitní data

# Třídění počítáním CountingSort

```
def countingSort(a,k,klic):  
    c = [0]*(k+1)  
    for x in a:  
        c[klic(x)] += 1      # četnosti  
    sum = 0  
    for i in range(k+1): # kumulované č.  
        c[i],sum = sum,c[i]+sum  
    b = [0]*len(a)          # výstup  
    for x in a:  
        b[c[klic(x)]] = x  
        c[klic(x)] += 1  
    return b
```

Čas i prostor  
 $\Theta(n+k)$

stabilní třídění  
pořadí prvků  
se stejným klíčem  
se nemění

# Přihrádkové třídění BucketSort

Třídění rozdělováním do přihrádek

## CountingSort

- $c[0..k]$  pole počítadel
- $c[i]$  četnost prvku s klíčem  $i$

## BucketSort

- $c[0..k]$  pole přihrádek
- $c[i]$  seznam prvků s klíčem  $i$

# Příhrádkové třídění BucketSort

```
def bucketSort(a,k,klic):  
    # prázdné příhrádky  
    c = [ [] for i in range(k+1) ]  
    for x in a:  
        # vlož x do příhrádky  
        c[klic(x)].append(x)  
    b = [ ]  
    for prihradka in c:  
        b += prihradka  
    return b
```

Čas i prostor  $\Theta(n+k)$

# Přihrádkové třídění – variace

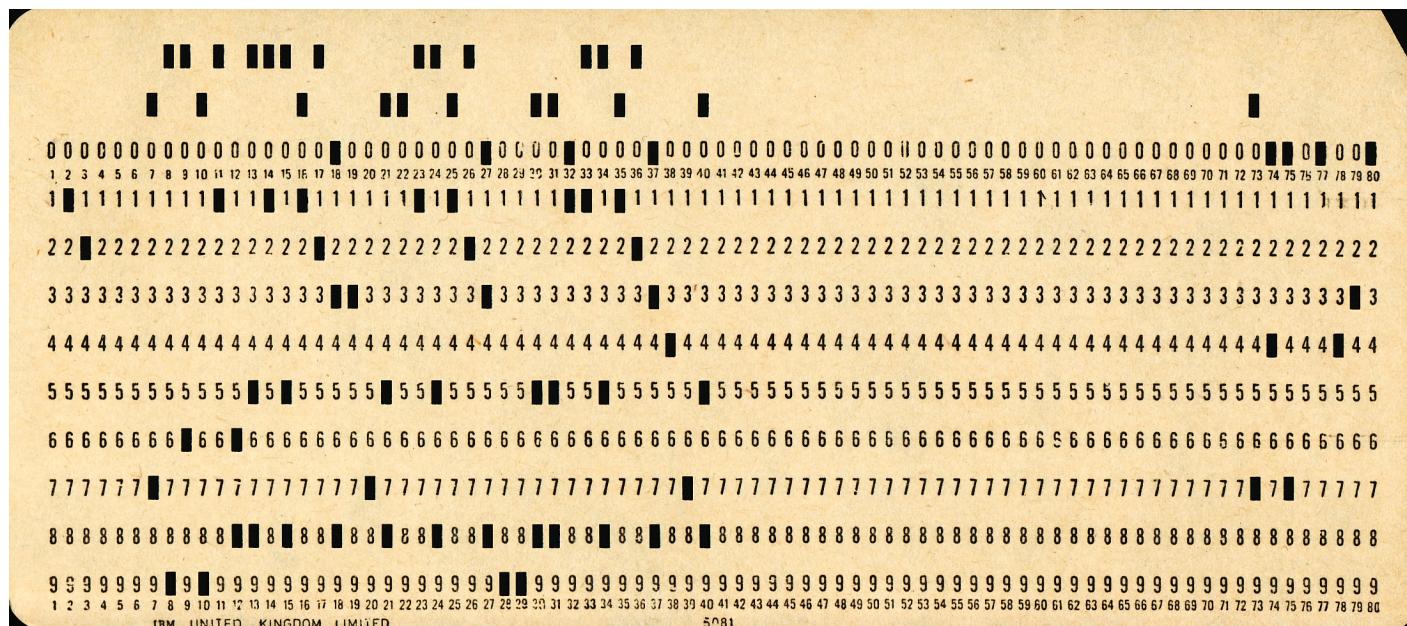
- 👉 Přihrádky  $c[i]$  lze implementovat jako spojové seznamy
- 💡 Co když jsou klíči desetinná čísla?
  - $klic(x) \in \langle 0, 1 \rangle$
  - pole  $c[0..n-1]$  kde  $n$  je délka vstupního pole  $a$
  - vlož  $x$  do přihrádky (= na konec seznamu)  
 $c[\lfloor n \cdot klic(x) \rfloor]$
  - každou přihrádku setříd' algoritmem **InsertionSort**
  - klíče rozmístěny rovnoměrně po  $\langle 0, 1 \rangle \Rightarrow$  čas  $O(n)$

# Radixové třídění RadixSort

Vicecestné příhrádkové třídění

Číslicové třídění

Herman Hollerith (1887)





# Radixové třídění RadixSort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

```
def radixSort(a,k,d,klic):  
    # klic(a[i]) je d-tice ∈ {0,1,...,k}d  
    for i in reversed(range(d)):  
        stabilním algoritmem setříd'  
        pole a dle i-té položky klíče
```

# Radixové třídění – analýza

👉 **Invariant cyklu:** Po  $i$ -tém průchodu jsou prvky uspořádány dle posledních  $i$  souřadnic klíče.

👉 korektnost algoritmu

👉 **Složitost:** RadixSort pracuje v čase  $O(d(n+k))$ , pokud využívá stabilní třídění pracující v čase  $O(n+k)$ .

- $d = O(1)$  a  $k = O(n) \Rightarrow$  třídění v čase  $O(n)$

# Radixové třídění – analýza

💡 Jak převést klíče na  $d$ -tice ?

- klíč uložen v  $b$  bitech
- pro lib.  $r \leq b$  lze klíč rozdělit na  $\lceil b/r \rceil$   $r$ -bitových “cifer”
- každá “cifra”  $\in \{0, 1, \dots, 2^r - 1\} \Rightarrow$  CountingSort pro  $k = 2^r - 1$
- jedna iterace v čase  $O(n+k) = O(n+2^r)$ , celkem  $d$  iterací
- čas  $O(d(n+2^r)) = O((b/r)(n+2^r))$

💡 Je-li dáno  $n$  a  $b$ , jak zvolit  $r$ ,  $r \leq b$  ?

$$b < \lfloor \log_2 n \rfloor$$

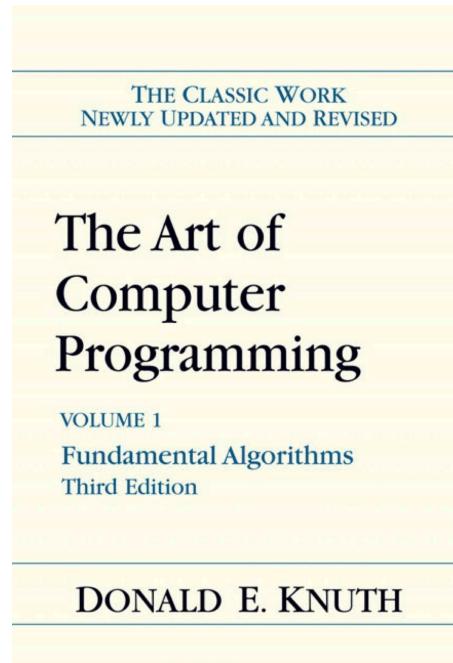
- $n+2^r = O(n)$
- zvolme  $r = b \Rightarrow$  časová složitost  $O(n)$

$$b \geq \lfloor \log_2 n \rfloor$$

- zvolme  $r = \lfloor \log_2 n \rfloor \Rightarrow$  časová složitost  $O(bn / \log n)$

# Algoritmizace

## Základní datové struktury



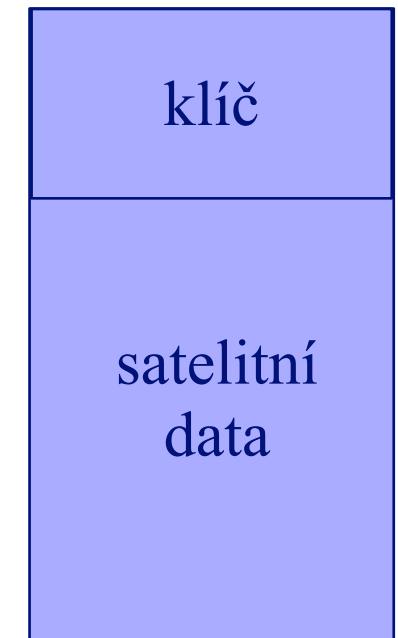
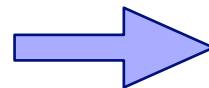
# Dynamické množiny

## Množina

- matematika / informatika
- statická / dynamická

## Abstraktní datový typ

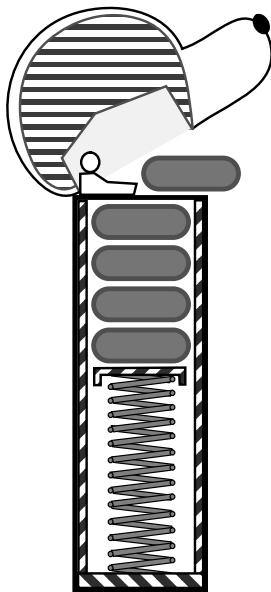
- množina prvků
- operace nad nimi
- různé implementace prostřednictvím
- konkrétních datových struktur



# Zásobník / Stack

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

- last-in, first-out



# Zásobník v reálném světě



# Zásobník – operace

## Operace

- Vlož / Push – vloží zadaný prvek do zásobníku
- Odeber / Pop – odebere a vrátí prvek, který byl vložen jako poslední

## Implementace v poli

- $z[0]$  dno zásobníku
- $z[-1]$  vrchol zásobníku

## Python: seznam (list), metody

- `pop()` odebere a vrátí poslední prvek v čase  $O(1)$
- `append(x)` vloží  $x$  na konec seznamu v čase
  - »  $\Theta(n)$  – nejhorší případ
  - »  $O(1)$  – amortizovaná složitost

# Zásobník v poli

```
class ZasobnikPole:

    def __init__(this):
        this.pole = [ ] # prázdný zásobník

    def push(this,x):
        this.pole.append(x)

    def pop(this):
        # předpokládá neprázdný zásobník
        return this.pole.pop()
```

při nesplnění je vhodné vyvolat  
výjimku !

# Fronta / Queue

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

- first-in, first-out



# Fronta – operace

## Operace

- Vlož / **Enqueue** – vloží zadaný prvek do fronty
- Odeber / **Dequeue** – odebere a vrátí prvek, který byl vložen jako první

## Implementace v poli

- $z[0]$  začátek fronty, zde se odebírá
- $z[-1]$  konec fronty, za něj se vkládá

## Python: seznam (list), metody

- ✓ `append(x)` vloží  $x$  na konec seznamu, jako u zásobníku
- ✗ `pop(0)` odebere a vrátí první prvek seznamu v čase  $\Theta(n)$ 
  - po odebrání je třeba posunout všechny zbývající prvky “doleva”



# Problém: složitost pop(0)

① Zavedeme proměnnou **zacatek** s indexem začátku fronty

- **Dequeue** vrátí pole[ **zacatek** ]
- **zacatek** +=1

✓ **Dequeue** v čase  $O(1)$

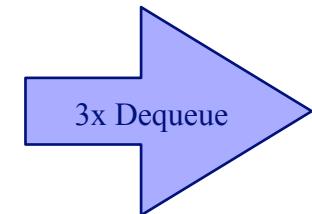
✗ Prostor  $\Theta(m)$

- $m = \#$  operací **Enqueue**

# Dequeue bez pop(0)

0	1	2	3	4	5	6	7	8	9
5	7	6	9	8	15	11	16	14	10

začátek



0	1	2	3	4	5	6	7	8	9
5	7	6	9	8	15	11	16	14	10

začátek



# Problém: složitost pop(0)

## ② Posouvat budeme jen občas

- Dequeue má lepší amortizovanou složitost

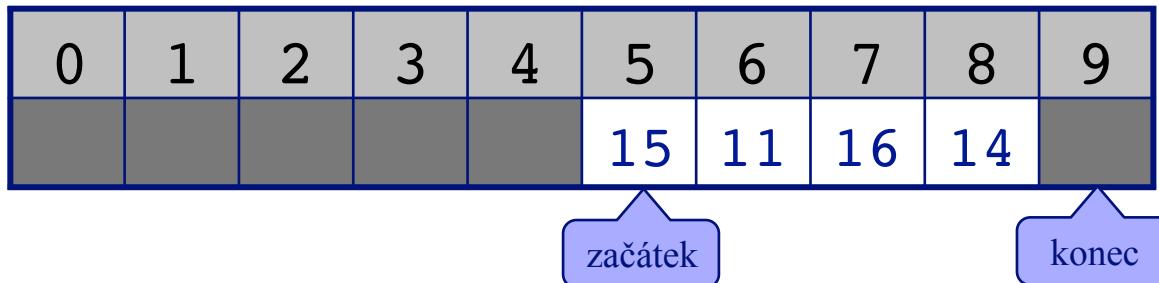
## ③ Stanovíme maximální kapacitu fronty

- posouváme jen když není místo na právě vkládaný prvek
- Dequeue v čase  $O(1)$
- Enqueue v čase  $O(n)$  – nejhorší,  $O(1)$  – nejlepší

## ④ Cyklická fronta

- v “zacykleném” poli stanovené kapacity
- bez posouvání
- po dosažení `pole[-1]` pokračujeme na `pole[0]`
- potřebujeme si pamatovat **zacatek** a **konec** fronty

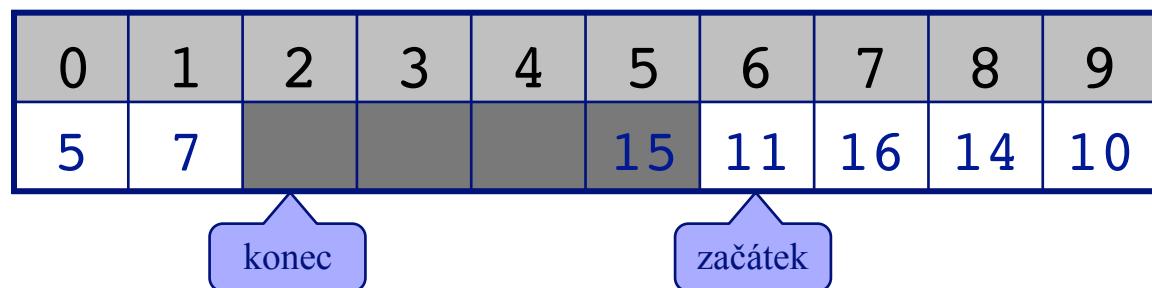
# Cyklická fronta



| Enqueue 10, 5, 7



## Dequeue



# Cyklická fronta v poli

obě operace v čase  $O(1)$  !

```
class FrontaPole:
```

```
    def __init__(this,kapacita):
```

```
        this.zacatek, this.konec = 0, 0
```

```
        this.pole = [None] * kapacita
```

```
        this.kapacita = kapacita
```

```
    def enqueue(this,x):
```

```
        # předpokládá volnou kapacitu
```

```
        this.pole[this.konec] = x
```

```
        this.konec = (this.konec + 1) % this.kapacita
```

nedostatečná kapacita ?  
alokace většího pole  
+ kopírování

```
    def dequeue(this):
```

```
        # předpokládá neprázdnou frontu
```

```
        x = this.pole[this.zacatek]
```

```
        this.zacatek = (this.zacatek + 1) % this.kapacita
```

```
        return x
```

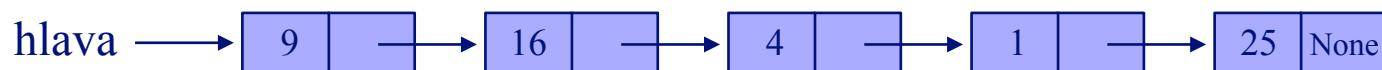
# Spojové seznamy

## 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

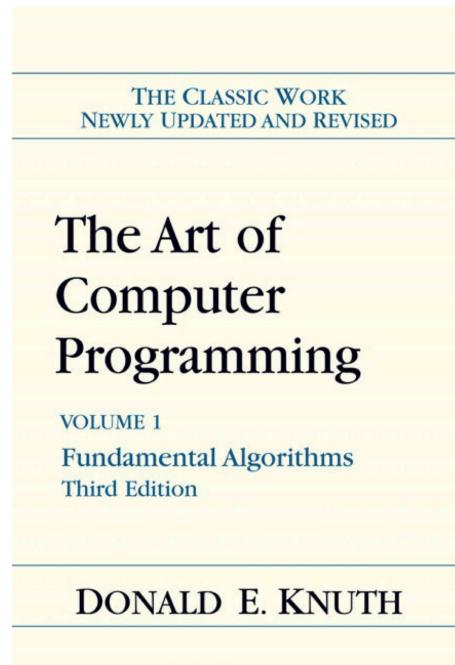
```
class Uzel:  
    """třída pro reprezentaci uzlu"""\n    def __init__(this,  
                 x = None,  
                 dalsi = None):  
        this.info = x  
        this.dalsi = dalsi
```

# 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
```

# 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



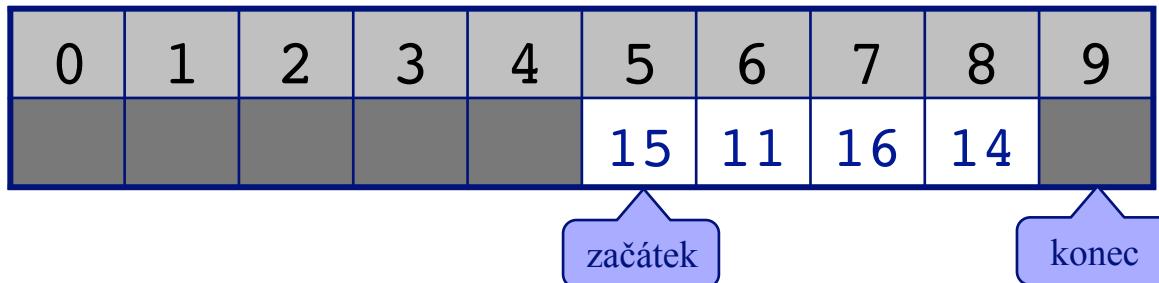
# Fronta / Queue

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

- first-in, first-out



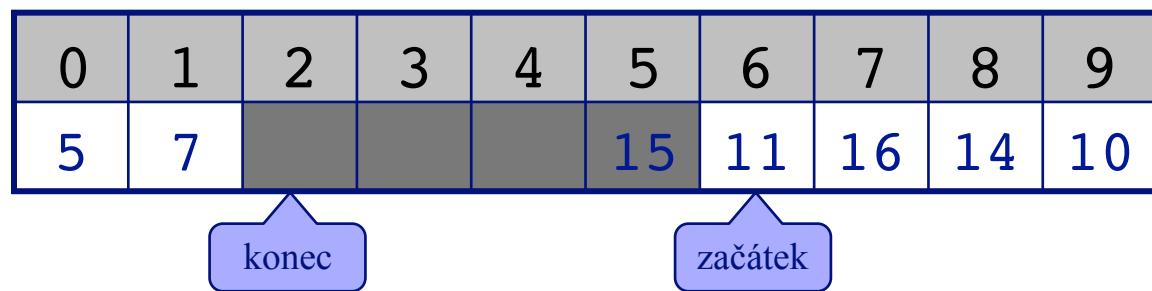
# Cyklická fronta



| 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**

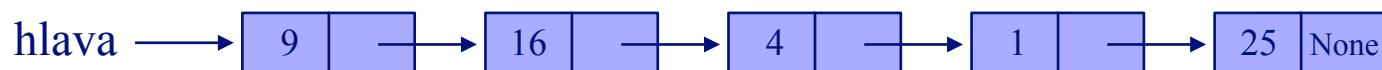
# Spojové seznamy

## 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  $\rightarrow$  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

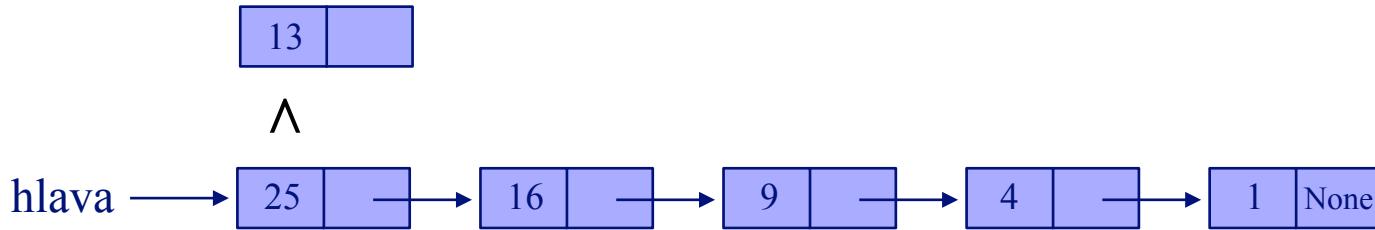
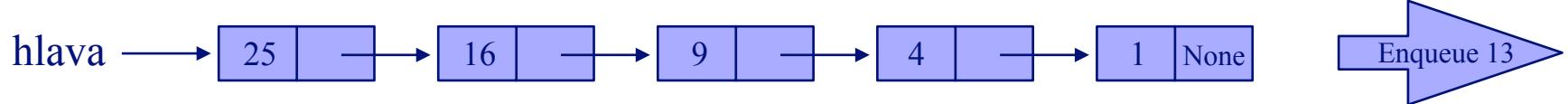
Fronta, v níž má každý prvek definovánu svoji **prioritu**

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

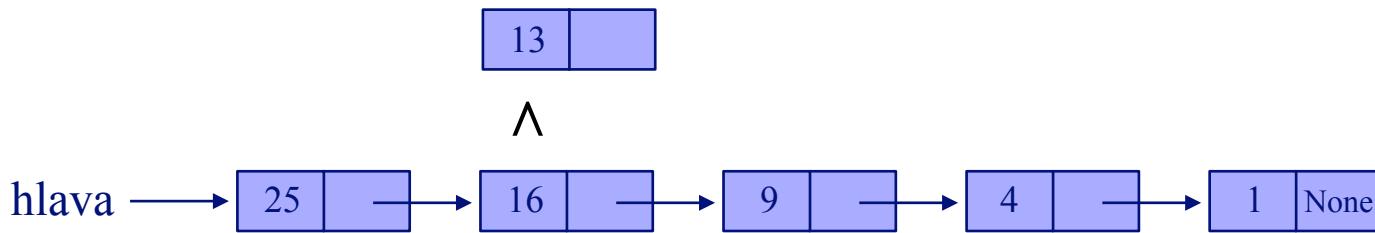
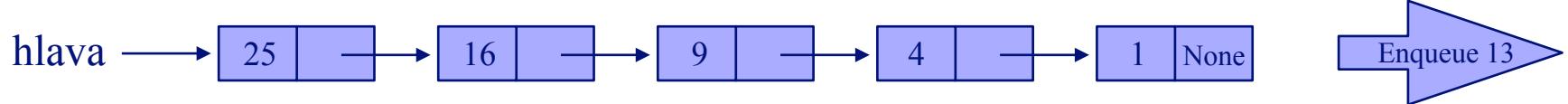
Implementace spojovým seznamem

- 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`)

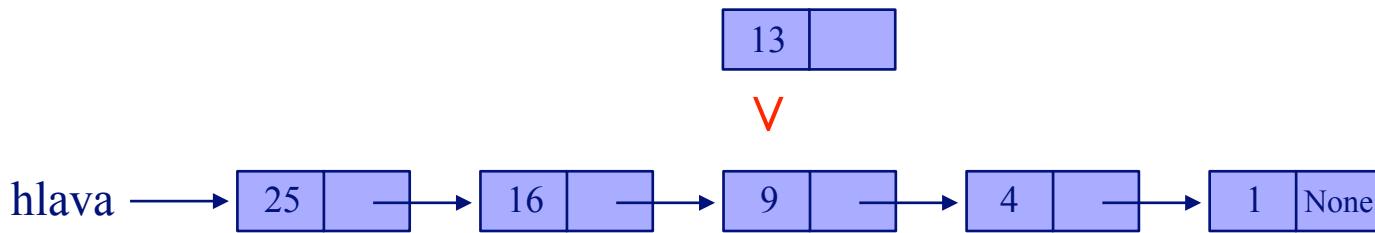
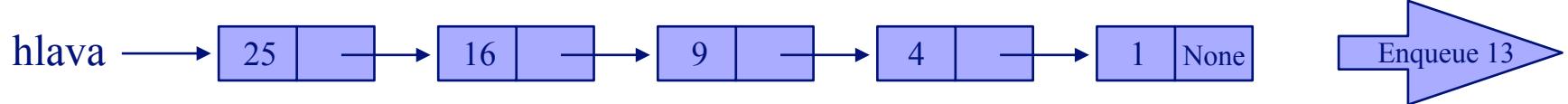
# Prioritní fronta ve spojovém seznamu



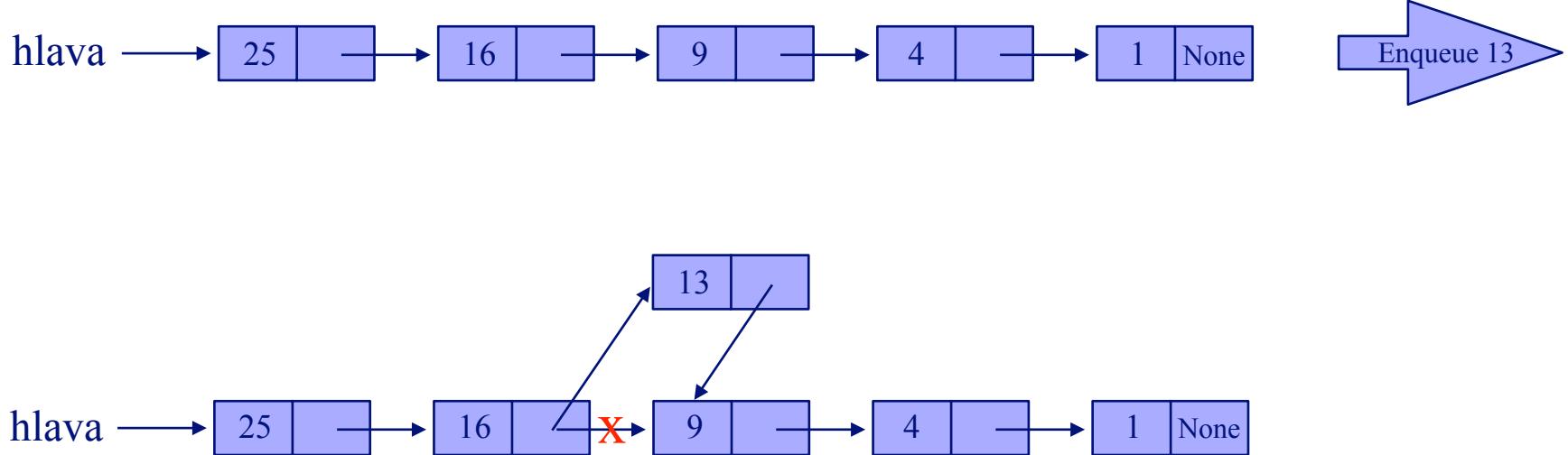
# Prioritní fronta ve spojovém seznamu



# 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

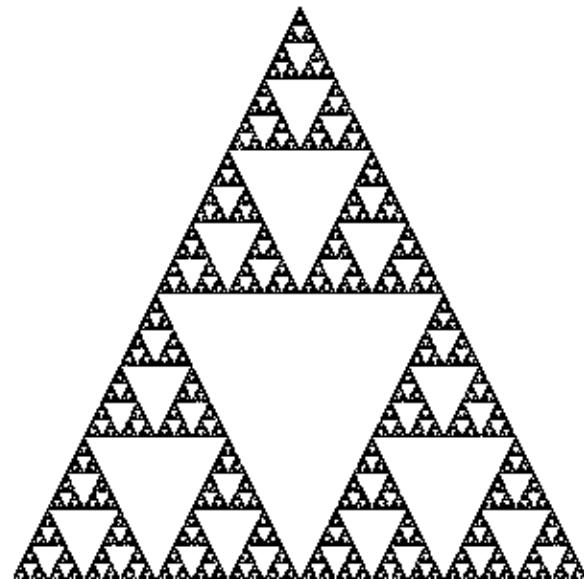
- datový typ **slovník** (dictionary)

Budeme mít později

- vyhledávací stromy
- hašovací tabulky

# Algoritmizace

## Rekurze



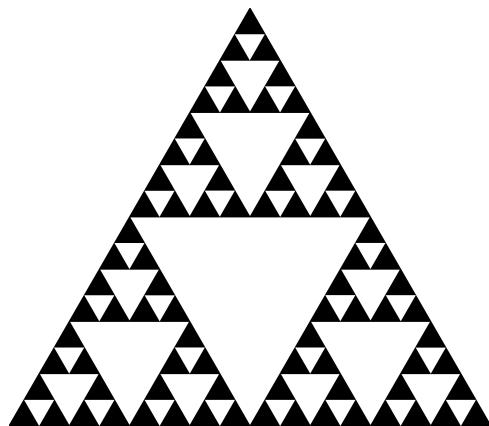
# Rekurze

Objekt je definován pomocí sebe sama

- z latinského *recurso*

## ☀ Příklady

- ze světa: rekurze u holice
- z geometrie
- z matematiky



$$0! = 1$$

$$n! = n \cdot (n - 1)!, n \geq 1$$

$$f_0 = f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}, n \geq 2$$

Fibonacciho posloupnost

# Rekurze v informatice

Algoritmus řešení problému

- pomocí řešení menších instancí téhož problému

Rekurzivní funkce (podprogram)

- volá sama sebe (přímo / nepřímo)
- na vstup menšího rozsahu
- podmínka ukončení (báze)

V teorii

- rekurzivní funkce : formální model algoritmu

V praxi

- logické / funkcionální programování : pouze rekurze
- imperativní programování: rekurzivní volání



# Příklad: palindromy

Palindrom – čte se stejně zleva doprava  
i zprava doleva

```
def palindrom(s):  
    # rekurzivní test na palindrom  
    if len(s) <=1 :  
        return True  
    else:  
        return s[0] == s[-1] and  
               palindrom(s[1:-1])
```

# Palindromy: rekurze vs. iterace

```
def palindrom_iter(s):  
    # iterativní test na palindrom  
    n = len(s)  
    for i in range(n//2):  
        if s[i] != s[n-i-1]:  
            return False  
    return True
```

## Výhody a nevýhody rekurze

- ✓ jednoduchost, intuitivnost
- ✓ často efektivní řešení složitého problému
- ✗ obsluha rekurze není zadarmo
- ✗ může existovat efektivnější iterativní řešení



# Problém: rekurze vs. iterace

① V jazyce Python sestavte funkce

- pro výpočet funkce  $n!$
- pro výpočet  $n$ -tého prvku Fibonacciho posloupnosti

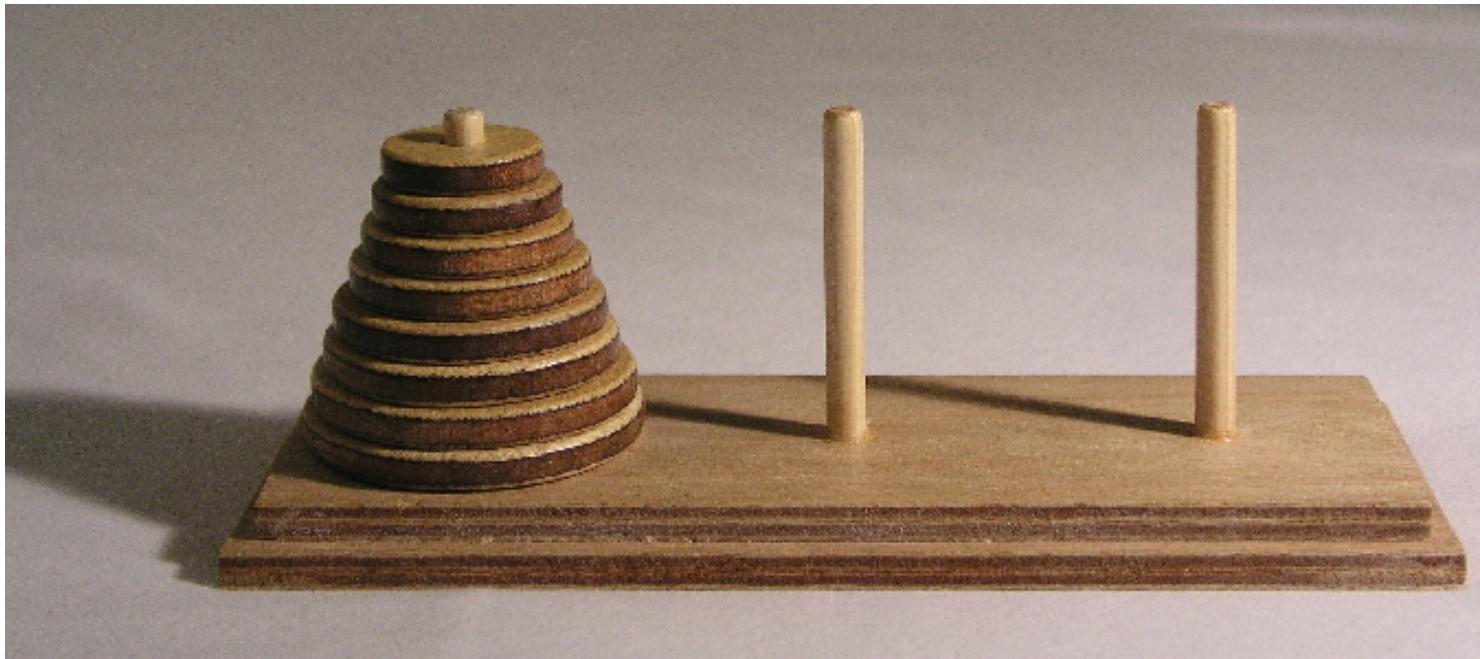
Pro každý problém se pokuste navrhnout dvě řešení

- rekurzivní
- iterativní (bez rekurze)

a porovnat jejich efektivitu.

# Příklad: Hanojská věž

Édouard Lucas (1883)



Hlavolam: Jak přemístit všech  $n$  kotoučů na prostřední “věž”?

- v jednom tahu se přesouvá vždy jeden kotouč
- větší kotouč nesmí nikdy ležet na menším

# Hanojská věž: rekurzivní řešení

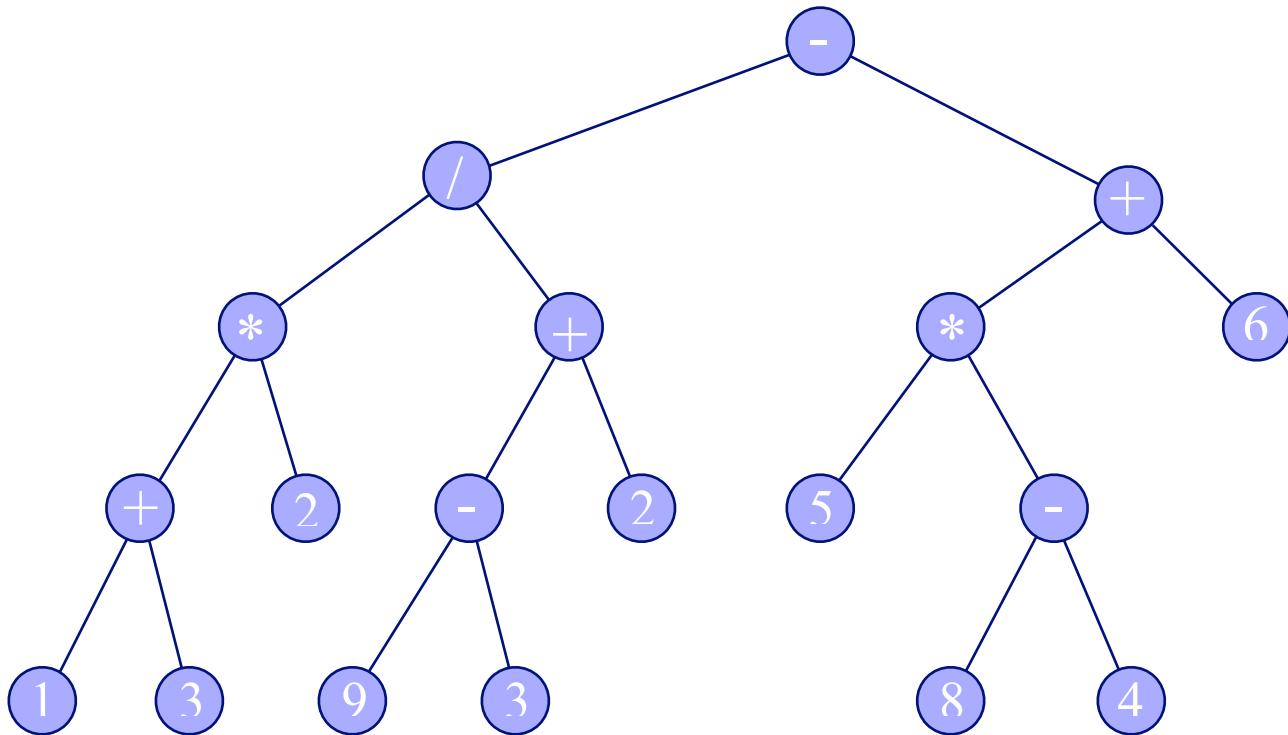
```
def hanoj(n, odkud, kam, rezervni):  
    if n == 1:  
        print("presuň kotouč z věže",  
              odkud, "na věž", kam)  
    else:  
        hanoj(n-1, odkud, rezervni, kam)  
        hanoj(1, odkud, kam, rezervni)  
        hanoj(n-1, rezervni, kam, odkud)
```

# Problémy: Hanojská věž

- ② Kolik tahů potřebuje náš (rekurzivní) algoritmus na přemístění  $n$  kotoučů?
- ③ Dle legendy existuje v Asii chrám, v němž každý den v poledne mniši slavnostně přemístí jeden z 64 zlatých kotoučů. Jakmile bude přemístěna celá “věž”, nastane konec světa. Spočítejte, za jak dlouho k tomu může dojít.

# Algoritmizace

## Rekurzivní datové struktury

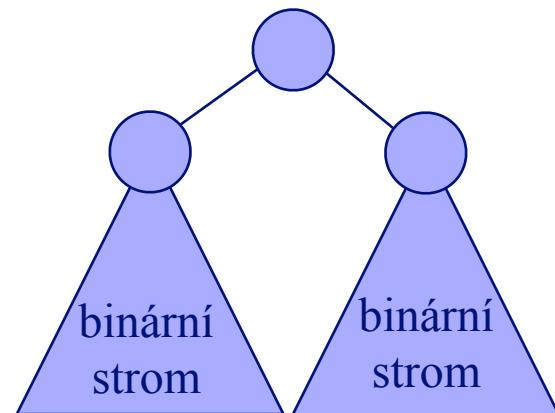
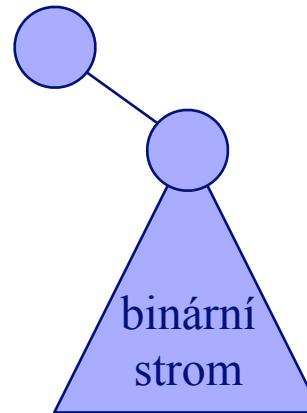
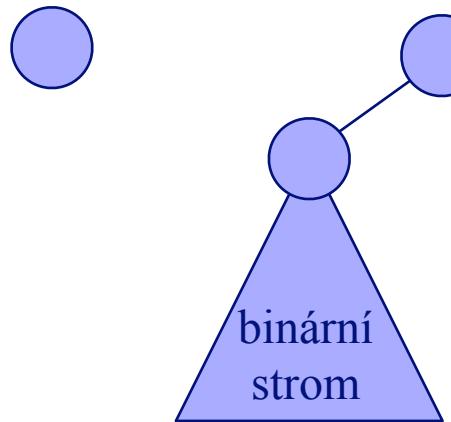


# Binární stromy

## Binární strom

- uspořádaný kořenový strom
- každý vrchol má nejvýše dvě děti

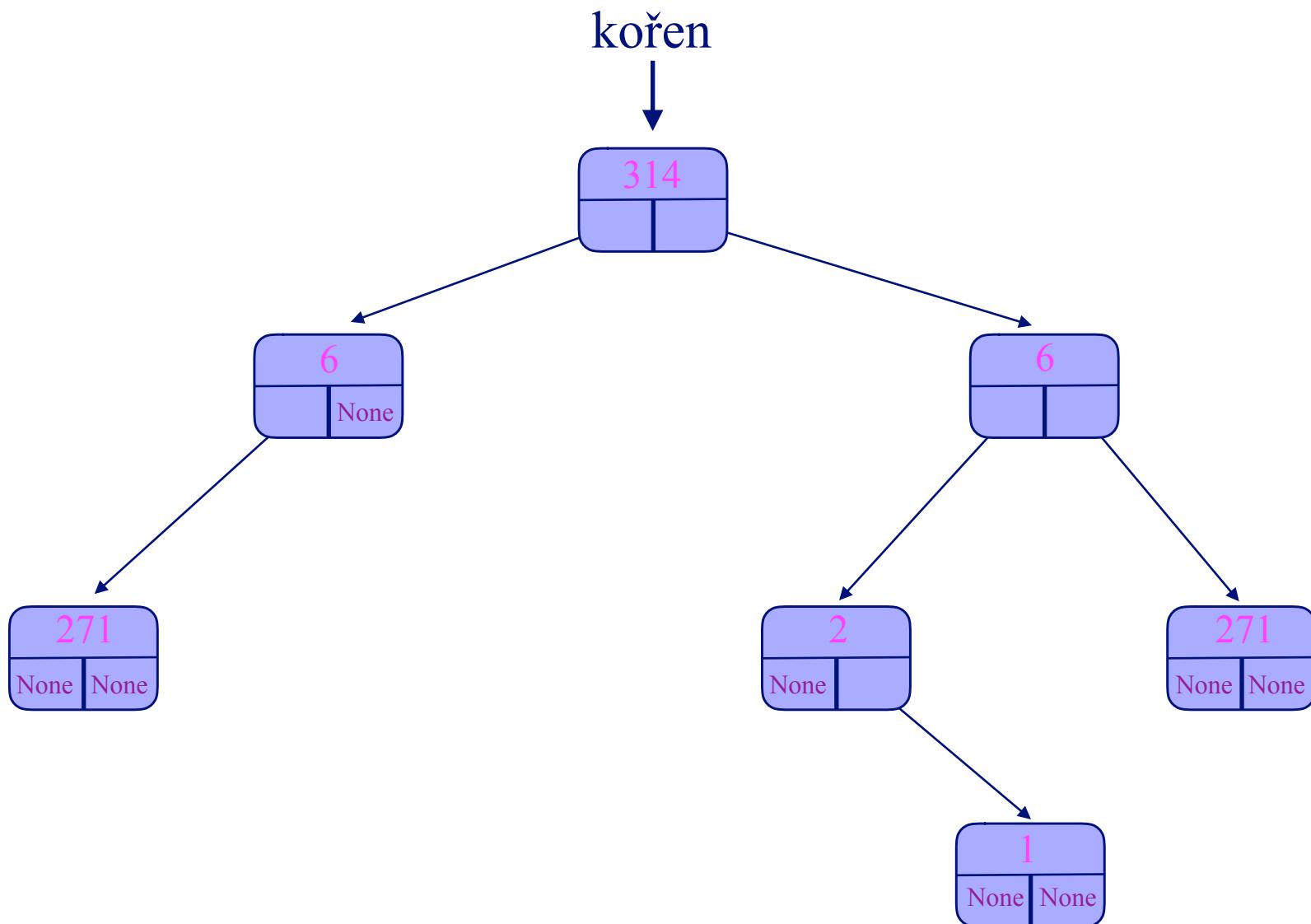
## Rekurzivní definice



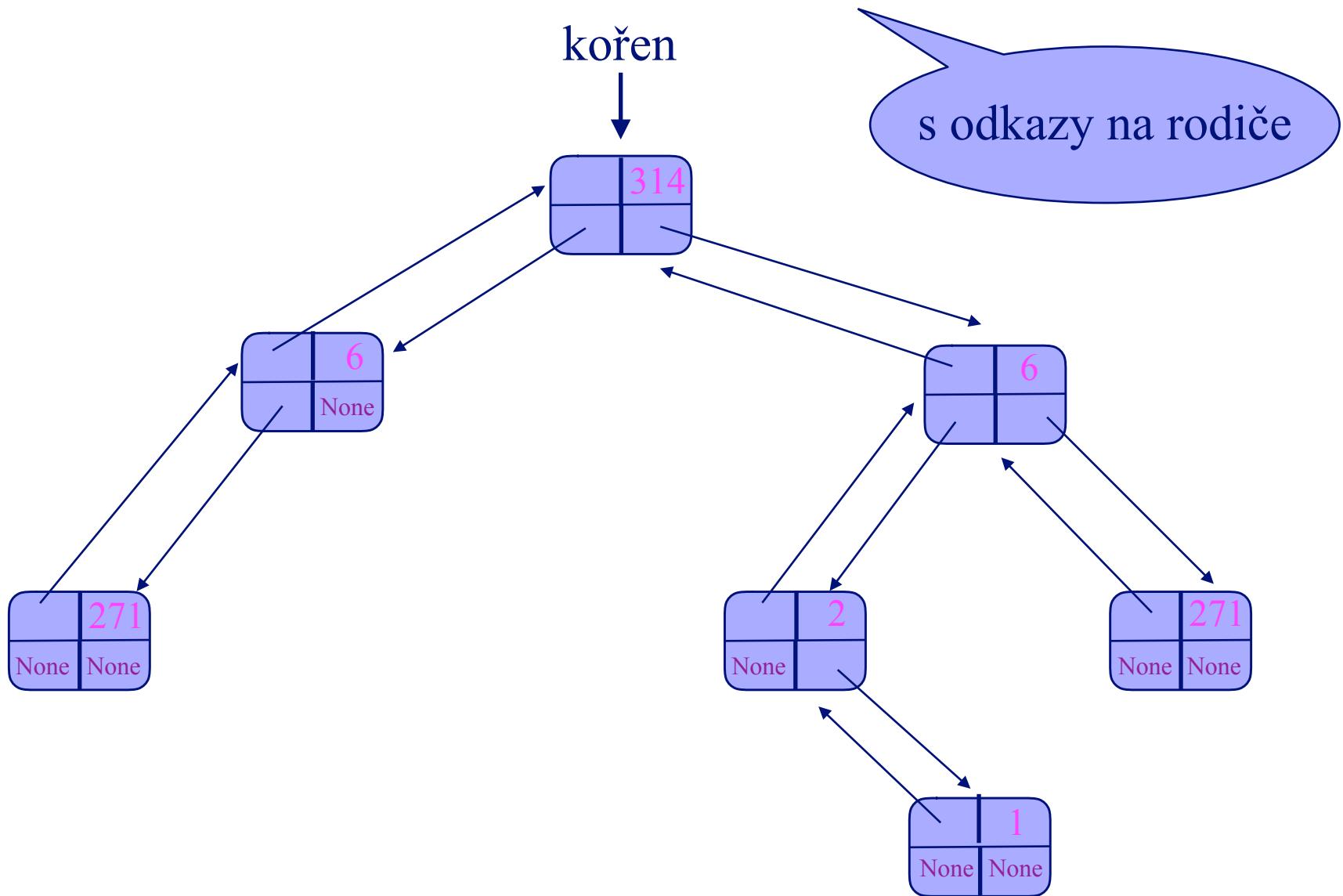
# Reprezentace binárního stromu

```
class VrcholBinStromu:  
    """třída pro reprezentaci vrcholu  
    binárního stromu"""  
  
    def __init__(this,  
                 x = None,  
                 levy = None,  
                 pravy = None):  
  
        this.info    = x      # data  
        this.levy   = levy   # levé dítě  
        this.pravy = pravy # pravé dítě
```

# Reprezentace binárního stromu



# Alternativní reprezentace



# Průchod binárním stromem

Navštívíme postupně každý vrchol stromu

- můžeme v něm provést zadanou akci

## Preorder

- zpracuj kořen stromu
- rekurzivně projdi levý podstrom
- rekurzivně projdi pravý podstrom

## Inorder

- rekurzivně projdi levý podstrom
- zpracuj kořen stromu
- rekurzivně projdi pravý podstrom

## Postorder

- rekurzivně projdi levý podstrom
- rekurzivně projdi pravý podstrom
- zpracuj kořen stromu

# Preorder – implementace

```
def preorder(koren):
    """vypíše vrcholy stromu se zadáným
    kořenem v pořadí preorder"""
    if kořen != None:
        print(koren.info) # zpracuj kořen
        preorder(koren.levy)
        preorder(koren.pravy)
```

## Časová složitost

- každou hranou projedene v každém směru 1x
- čas  $O(n)$ ,  $n = \text{počet vrcholů} = \text{počet hran} + 1$

# Inorder – implementace

```
def inorder(koren):  
    """vypíše vrcholy stromu se zadáným  
    kořenem v pořadí inorder"""  
  
    if kořen != None:  
        inorder(koren.levy)  
        print(koren.info) # zpracuj kořen  
        inorder(koren.pravy)
```

Čas  $O(n)$

# Postorder – implementace

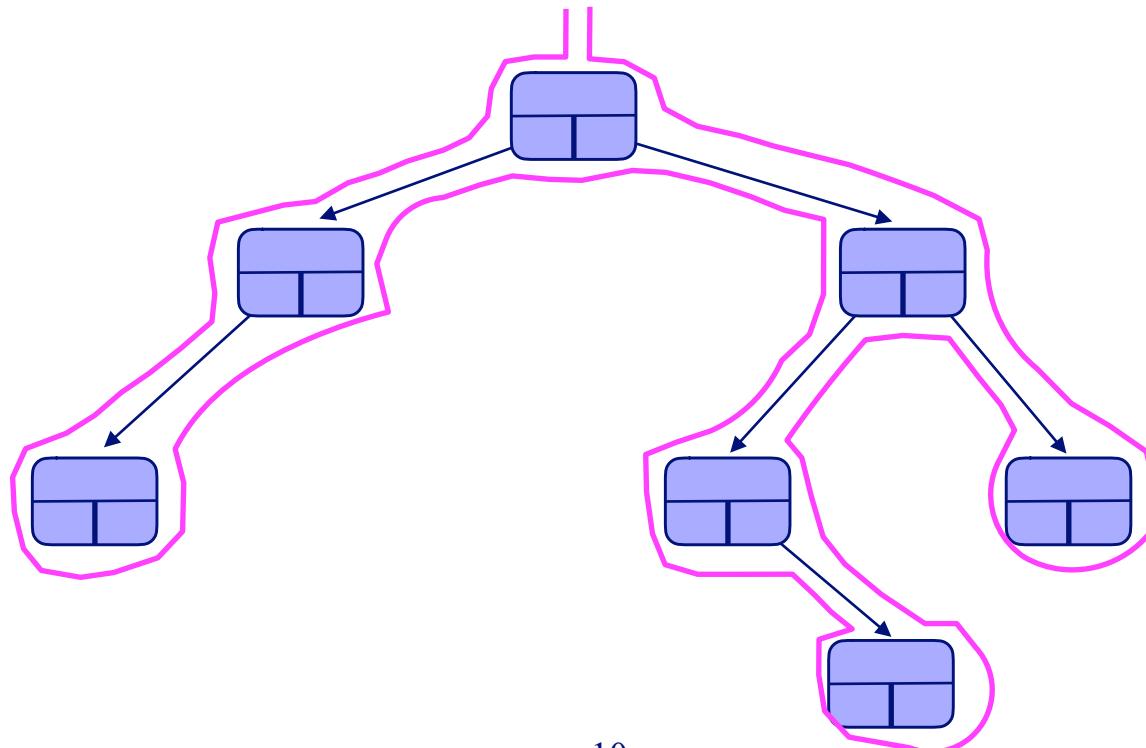
```
def postorder(koren):
    """vypíše vrcholy stromu se zadáným
    kořenem v pořadí postorder"""
    if kořen != None:
        postorder(koren.levy)
        postorder(koren.pravy)
        print(koren.info) # zpracuj kořen
```

Čas  $O(n)$

# Průchod do hloubky

## Průchod do hloubky (depth-first search, DFS)

- navštiv kořen
- rekurzivně projdi podstromy
- preorder, inorder a postorder – speciální případy



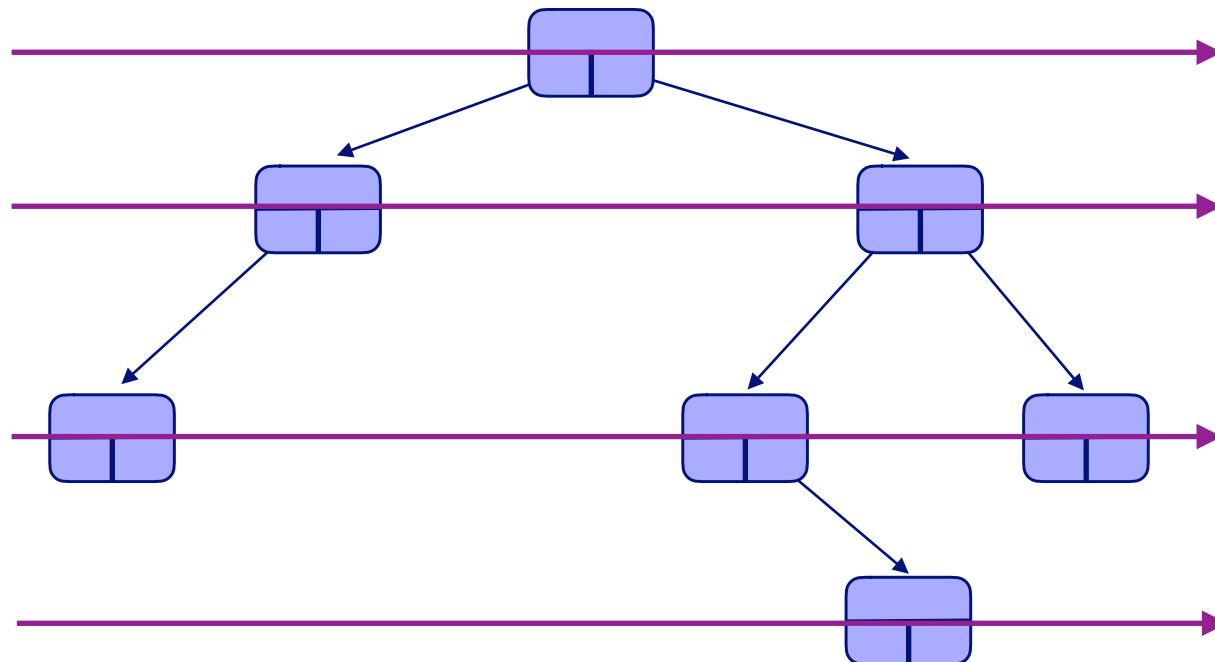
# Průchod do hloubky bez rekurze

```
def dfs(koren):
    z = Zasobnik()
    z.push(koren)
    while not z.empty():
        vrchol = z.pop()
        if vrchol != None:
            print(vrchol.info)
            z.push(vrchol.pravy)
            z.push(vrchol.levy)
```

# Průchod do šířky

## Průchod do šířky (breadth-first search, BFS)

- místo zásobníku použijeme frontu
- vrcholy navštěvuje po hladinách
- “algoritmus vlny”



# Průchod do šířky – implementace

```
def bfs(koren):
    f = Fronta()
    f.enqueue(koren)
    while not f.empty():
        vrchol = f.dequeue()
        if vrchol != None:
            print(vrchol.info)
            f.enqueue(vrchol.levy)
            f.enqueue(vrchol.pravy)
```

# Problémy: Binární stromy

- ① V jazyce Python navrhněte funkci, která obdrží binární strom a spočítá jeho
  - výšku
  - průměrnou výšku

Přitom výška (průměrná výška) stromu je definována jako maximální (průměrná) délka cesty z kořene do listu.

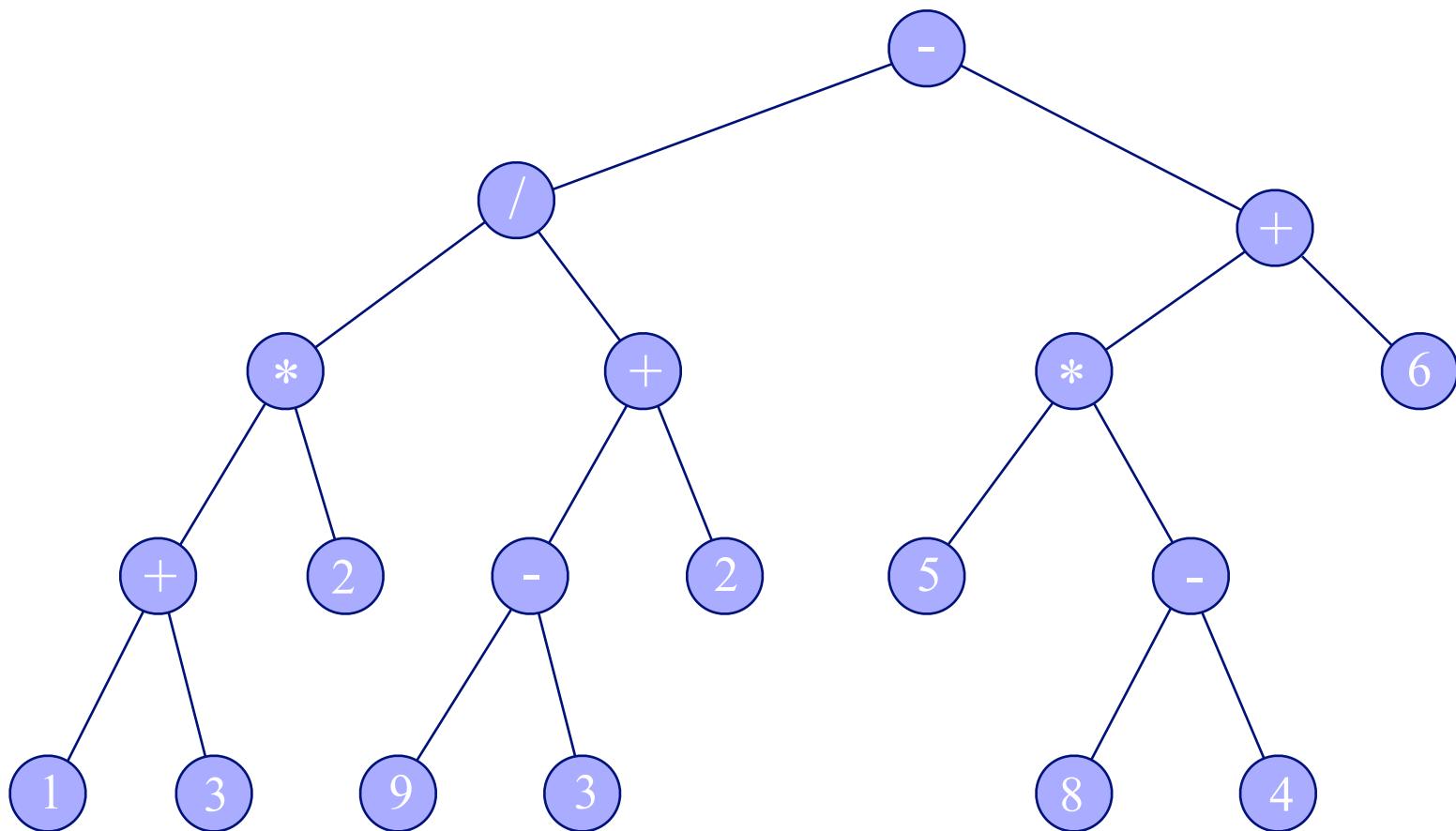
- ② Navrhněte efektivní algoritmus, který zjistí, zdali je zadaný binární strom symetrický dle svislé osy, procházející jeho kořenem.

# Aritmetické výrazy

Každý aritmetický výraz lze reprezentovat  
binárním stromem

- vnitřní vrcholy – operátory
- listy – operandy
- závorky explicitně ve stromě nejsou
- implicitně jsou reprezentovány podstromy

# Příklad: strom arit. výrazu



$$(((1+3)*2)/((9-3)+2))-((5*(8-4))+6)$$

# Strom arit. výrazu – konstrukce

Vstup: aritmetický výraz v infixové notaci,  
plně uzávorkovaný

① Rekurzivně, shora dolů

**if** výraz je tvořen operandem  $o$  :

**return** VrcholBinStromu( $o$ )

**else**:

vyhledej operátor  $op$ , který by byl v kořeni stromu

- plně uzávorkovaný výraz: operátor mimo závorky
- rekurzivně sestroj stromy  $l, p$  pro podvýrazy vlevo a vpravo
- **return** VrcholBinStromu( $op, l, p$ )

Čas  $\Theta(n^2)$



- $n$  – délka výrazu (počet operátorů, operandů a závorek)
- na vyhledání operátoru v kořeni je třeba čas  $\Theta(n)$

# Strom rekurze

## Vrcholy

- podúlohy (rekurzivně) volané během (rekurzivního) algoritmu

## Kořen

- původní úloha

## Děti každého rodiče

- představují podúlohy, (rekurzivně) volané v rodičovské úloze

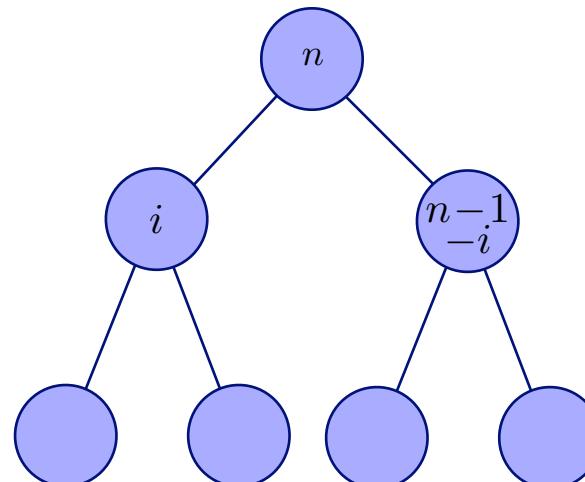
# Strom rekurze

hladina

čas

0

$O(n)$



1

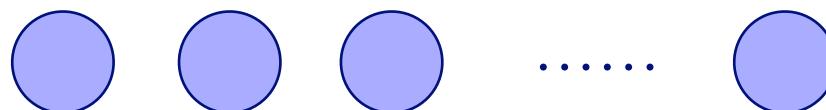
$O(n)$

2

$O(n)$

.....

$\leq n$



$O(n)$

## Časová složitost

$$\bullet \quad T(n) = n \cdot O(n) = O(n^2)$$

✍ **Problém:**

Platí  $T(n) = \Omega(n^2)$  ?

# Strom arit. výrazu – konstrukce

② Zdola nahoru, využijeme zásobník  $z$

Další prvek  $p$  na vstupu je

- operand :  $v = \text{VrcholBinStromu}(p)$ ,  $z.push(v)$
- operátor :  $z.push(p)$
- ) :  $\text{pravý}, \text{op}, \text{levý} = z.pop(), z.pop(), z.pop()$
- $v = \text{VrcholBinStromu}(\text{op}, \text{levý}, \text{pravý})$
- $z.push(v)$

Na konci obsahuje  $z$  kořen stromu výrazu

Čas  $\Theta(n)$

- $n$  – délka výrazu (počet operátorů, operandů a závorek)

✎ **Problém:** Co když výraz není plně uzávorkovaný?

# Aritmetické výrazy – notace

## Notace

- infixová: operátor mezi operandy,  $(1+2)*3$
- postfixová: operátor za operandy,  $1\ 2\ +\ 3\ *$
- prefixová: operátor před operandy,  $*\ +\ 1\ 2\ 3$

Souvisí s průchodem stromem aritmetického výrazu

- průchod binárním stromem
- metodami inorder, postorder a preorder
- zpracování vrcholu = výpis hodnoty v něm uložené



# Aritmetické výrazy – notace

Průchod stromem aritmetického výrazu

- preorder  $\Rightarrow$  výraz v prefixové notaci
- postorder  $\Rightarrow$  výraz v postfixové notaci
- inorder  $\Rightarrow$  výraz v infixové notaci, ale **bez závorek !**
  - » lze snadno napravit

Výpis výrazu v infixové notaci

- průchod stromem výrazu metodou inorder
- před rekurzivním voláním na levý / pravý podstrom
- stačí vytisknout ' ( ' , po návratu ' ) '

# Aritmetické výrazy – vyhodnocení

```
def vyhodnot(koren):  
    """vyhodnocení aritmetického výrazu  
    reprezentovaného stromem s korenem"""  
  
    if koren.levy = None:  
        return koren.info # operand v listu  
    else:  
        l = vyhodnot(koren.levy)  
        p = vyhodnot(koren.pravy)  
        if koren.info = '+':  
            return l + p  
        elif koren.info = '-':  
            return l - p  
        elif koren.info = '*':  
            return l * p  
        elif koren.info = '/':  
            return l / p
```



# Vyhodnocení výrazu – postfix

Vyhodnocení aritmetického výrazu v postfixové notaci

- jeden průchod zleva doprava
- zásobník **z**

Na vstupu je

- operand **o**: **z.push(o)**
- operátor **op**:
  - » **pravý, levý = z.pop(), z.pop()**
  - » aplikuj **op** na **levý** a **pravý** operand
  - » výsledek ulož do zásobníku

konstrukce stromu  
z postfix. výrazu:  
analogicky !

Po zpracování celého výrazu

- **z** obsahuje výslednou hodnotu výrazu

Čas  $O(n)$

- **n** – délka výrazu

# Vyhodnocení výrazu – prefix

Vyhodnocení aritmetického výrazu v prefixové notaci

- popíšeme 3 metody

## ① Zprava doleva, zásobník

- podobně jako postfix

Rozdíl

- na vrcholu zásobníku je vždy **levý** operand
- pod ním je **pravý**
- tj. **levý, pravý =  $z.pop()$ ,  $z.pop()$**

Čas  $O(n)$

- $n$  – délka výrazu

# Vyhodnocení výrazu – prefix

## ② Zleva doprava, zásobník $z$

Na vstupu je operand nebo operátor  $\circ$

- $z.push(\circ)$

konstrukce stromu  
z prefix. výrazu:  
analogicky !

**while** na vrcholu zásobníku jsou dvě čísla :

- **pravý, levý, op** =  $z.pop(), z.pop(), z.pop()$
- aplikací **op** na **levý** a **pravý** operand získáme **vysledek**
- $z.push(vysledek)$

Po zpracování celého výrazu (a celého zásobníku)

- $z$  obsahuje výslednou hodnotu výrazu

Čas  $O(n)$

- $n$  – délka výrazu

# Vyhodnocení výrazu – prefix

## ③ Zleva doprava, rekurzivně

Na vstupu je číslo **c**

- **return** **c**

Na vstupu je operátor **op**

- rekurzivním voláním na dosud nezpracovanou část vstupu obdržíme **levý** operand
- dalším rekurzivním voláním na dosud nezpracovanou část vstupu obdržíme **pravý** operand
- aplikací **op** na **levý** a **pravý** operand získáme **vysledek**
- **return** **vysledek**

Čas  $O(n)$

- **n** – délka výrazu

konstrukce stromu  
z prefix. výrazu:  
analogicky !

# Vyhodnocení výrazu – infix

## ① Rekurzivně

Imitace konstrukce / vyhodnocení stromu arit. výrazu

**if** výraz je tvořen operandem **o** : **return o**

**else**: vyhledej operátor **op**, který by byl v kořeni stromu

- plně uzávorkovaný výraz: operátor mimo závorky
- neúplně u.v.: operátor nejnižší priority mimo závorky co nejvíce vpravo
- rekurzivně vyhodnot' podvýrazy vlevo a vpravo
- na takto získané hodnoty operandů aplikuj **op**
- **return výsledek**

Čas  $\Theta(n^2)$

- na vyhledání operátoru v kořeni je třeba čas  $\Theta(n)$

# Vyhodnocení výrazu – infix

## ② Převodem do postfixové notace

Převod výrazu z infixové do postfixové notace

- v lineárním čase, viz dále

Vyhodnocení výrazu v postfixové notaci

Lze realizovat souběžně

- vytvářený postfixový zápis se průběžně vyhodnocuje

Čas  $O(n)$

# Převod z infixové do postfixové notace

Zásobník **z** operátorů a závorek

- pořadí operandů je v obou notacích stejné
- operátory je třeba při převodu “zdržet” v zásobníku

Další prvek **p** na vstupu je

- operand : `print(p)`
- `(` : `z.push(p)`
- `)` : `z.pop(op), while op != '(': print(op)`
- operátor : nesmí “předběhnout” operátory  $\geq$  priority
  - » `z.pop(op), print(op)` dokud nenarazíme na dno, na `(`, nebo na operátor  $<$  priority než **p**
  - » `z.push(p)`

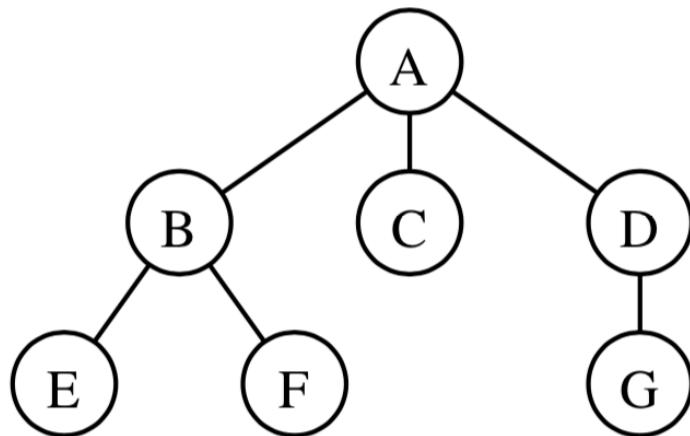
Na konci odešleme operátory ze zásobníku na výstup

# Obecné ( $k$ -ární) stromy

## $k$ -ární strom

- uspořádaný kořenový strom
- každý vrchol má nejvýše  $k$  dětí

💡 **Příklad:** ternární strom



# Reprezentace $k$ -árního stromu

## ① Pole odkazů na děti

- každý vrchol obsahuje pole délky  $k$
- vhodné, pokud  $k$  známe a je malé

## ② Spojový seznam dětí

- každý vrchol obsahuje odkaz
  - » na nejlevější dítě
  - » na sourozence (bezprostředně vpravo)
    - » pokud neexistují, odkaz nabývá hodnoty `None`
- fakticky jde tedy o reprezentaci binárním stromem

# Reprezentace obecného stromu

```
class VrcholStromu:  
    """třída pro reprezentaci vrcholu  
    obecného stromu"""  
  
    def __init__(this,  
                 x = None,  
                 dite = None,  
                 sou = None):  
  
        this.info = x      # data  
        this.dite = dite  # nejlev. dítě  
        this.sou = sou    # pravý  
                           # sourozenc
```



# Příklad: obecný strom

