

# Algoritmizace

Jiří Vyskočil, Marko Genyg-Berezovskyj

2010 - 2020

- stránky předmětu:

<https://cw.fel.cvut.cz/wiki/courses/b4b33alg/start>

- cíle předmětu

Cílem je schopnost samostatné implementace různých variant základních úloh informatiky. Hlavní téma jsou algoritmy řazení a vyhledávání a jim odpovídající datové struktury. Důraz je kladen na algoritmický aspekt úloh a efektivitu praktického řešení.

- předpoklady

Kurs předpokládá **schopnost programování** v alespoň jednom z jazyků C/C++/Java. Součástí cvičení jsou programovací úlohy na řešení problematiky ALG. Adept musí ovládat základní datové struktury jako pole, seznam, soubor a musí být schopen manipulovat s daty v těchto strukturách.

# Problémy a algoritmy

## ■ Výpočetní problém P

- Úkol zpracovat vstupní data IN na výstupní data OUT se zadánými vlastnostmi.

## ■ Algoritmus A

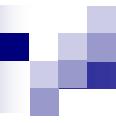
- Výpočetní postup řešení problému P.
- Tedy přesný popis posloupnosti kroků, která vezme vstupní data IN a vyprodukuje výstupní data OUT dle zadaných vlastností problémem P.

## ■ Instance problému

- Problém s konkrétními vstupními daty potřebnými pro jeho řešení.

## ■ Korektnost algoritmu A pro problém P

- Algoritmus A je korektní, pokud pro každou instanci problému P vydá v **konečném** čase **správný** výstup (tedy takový, který řeší problém P).



# Jak měřit algoritmy?

- Podle algoritmu vytvoříme program v programovacím jazyku a několik vybraných instancí problému.
- Algoritmy pak porovnáme podle rychlosti a paměťové náročnosti na konkrétním počítači.
- Ale co když bychom změnili počítač, nebo jen OS, nebo co kdybychom vybrali jiné instance problému, nebo kdybychom změnili programovací jazyk?
- Budou algoritmy výše popsaným způsobem stále stejně porovnatelné? .... zřejmě nikoliv ...
- → Budeme potřebovat nějakou nezávislou metodu (na programovacím jazyku, počítači, atd ...) na porovnávání algoritmů.

# Růst funkcí

- Čas potřebný ke zpracování dat velikosti  $n$ , jestliže počet operací při provádění algoritmu je dán funkcí  $T(n)$  a provedení jedné operace trvá jednu mikrosekundu. (Připomeňme, že počet atomu ve vesmíru se odhaduje na  $10^{80}$  a stáří na  $14 \times 10^9$  let)

$T(n)/n$	20	40	60	80	100
$\log(n)$	4.3 μs	5.3 μs	5.9 μs	6.3 μs	6.6 μs
$n$	20 μs	40 μs	60 μs	80 μs	0.1 ms
$n \log(n)$	86 μs	0.2 ms	0.35 ms	0.5 ms	0.7 ms
$n^2$	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms
$n^3$	8 ms	64 ms	0.22 s	0.5 s	1 s
$n^4$	0.16 s	2.56 s	13 s	41 s	100 s
$2^n$	1 s	12.7 dní	36600 let	$10^{11}$ let	$10^{16}$ let
$n!$	77100 let	$10^{34}$ let	$10^{68}$ let	$10^{105}$ let	$10^{144}$ let

# Asymptotické odhady

- horní asymptotický odhad (velké omikron odhad):

$$f(n) \in O(g(n))$$

- význam:

$f$  je shora asymptoticky ohraničená funkcí  $g$  (až na multiplikativní konstantu)

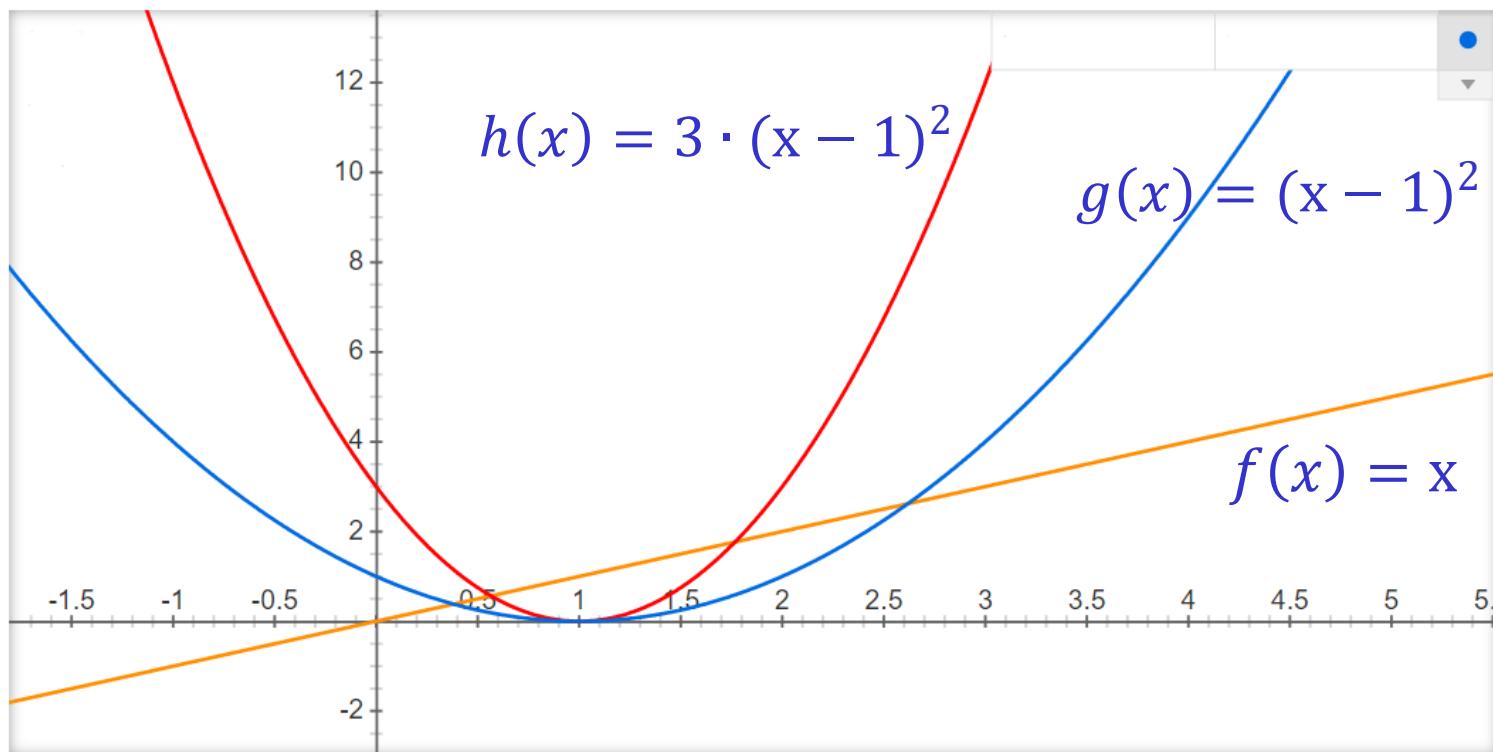
- definice:

$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : f(n) \leq c \cdot g(n)$$

kde  $c \in \mathbb{R}^{>0}$   $n_0, n \in \mathbb{N}$   $f, g \in \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$

# Asymptotické odhady

- příklad  $f(x) \in O(g(n))$ ,  $h(x) \in O(g(n))$



$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : f(n) \leq c \cdot g(n)$$

# Asymptotické odhady

- horní asymptotický odhad pro více proměnných:

$$f(n_1, \dots, n_k) \in O(g(n_1, \dots, n_k))$$

- definice:

$$(\exists c > 0)(\exists n_0)(\forall n_1 > n_0) \cdots (\forall n_k > n_0) :$$

$$f(n_1, \dots, n_k) \leq c \cdot g(n_1, \dots, n_k)$$

kde  $c \in \mathbb{R}^{>0}$   $n_0, n_1, \dots, n_k \in \mathbb{N}$   $f, g \in \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$

# Asymptotické odhady

## ■ poznámka

v literatuře se často místo

$$f(n) \in O(g(n))$$

používá zápis

$$f(n) = O(g(n))$$

není to ale zcela přesné z matematického hlediska

# Asymptotické odhady

- dolní asymptotický odhad (velké omega odhad):

$$f(n) \in \Omega(g(n))$$

- význam:

$f$  je zdola asymptoticky ohraničená funkcí  $g$  (až na konstantu)

- definice:

$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : c \cdot g(n) \leq f(n)$$

kde  $c \in \mathbb{R}^{>0}$   $n_0, n \in \mathbb{N}$   $f, g \in \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$

# Asymptotické odhady

- optimální asymptotický odhad (velké théta odhad):

$$f(n) \in \Theta(g(n))$$

- význam:

$f$  je asymptoticky ohraničená funkcí  $g$  z obou stran (až na konstantu)

- definice:  $\Theta(g(n)) \stackrel{\text{def}}{=} O(g(n)) \cap \Omega(g(n))$

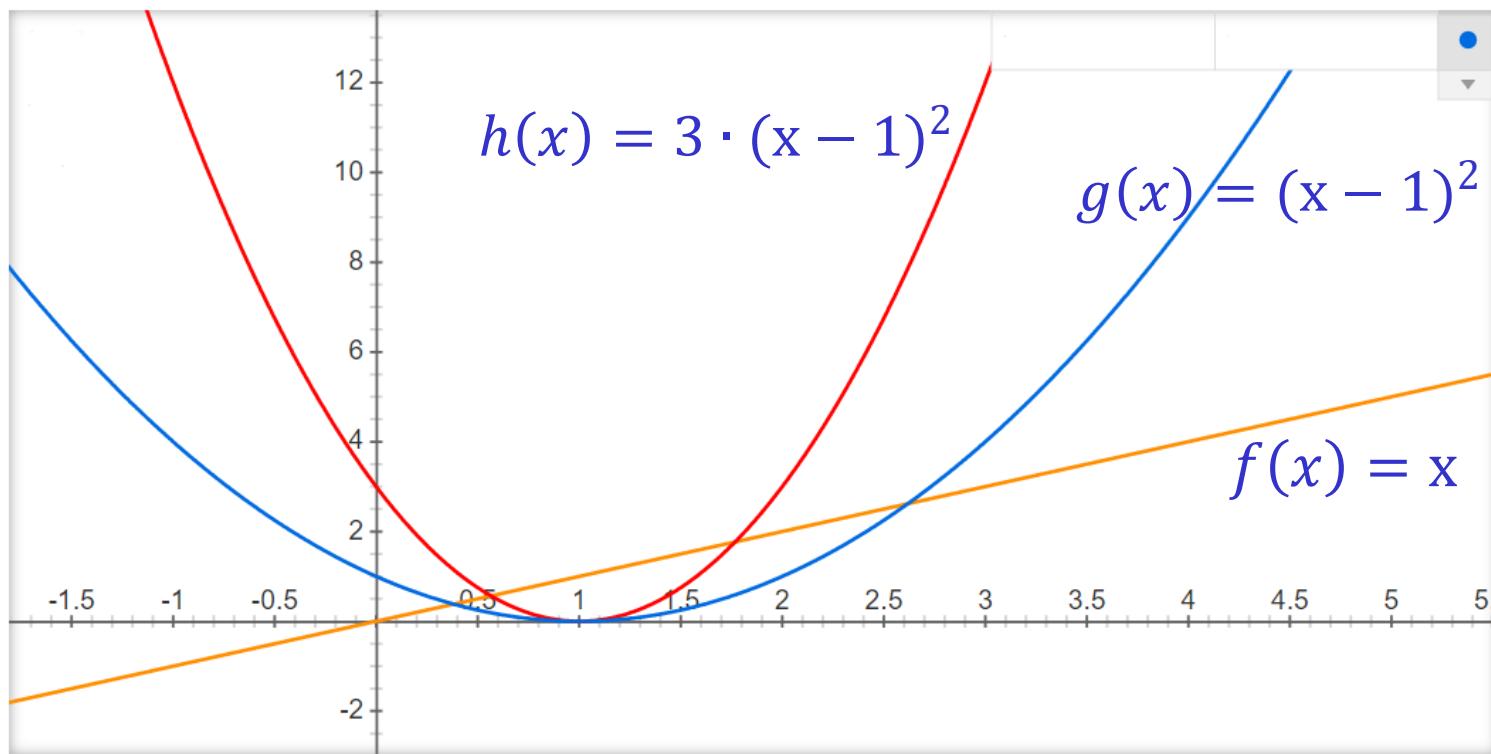
- nebo alternativně:

$$(\exists c_1, c_2 > 0)(\exists n_0)(\forall n > n_0): c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

kde  $c_1, c_2 \in \mathbb{R}^{>0}$   $n_0, n \in \mathbb{N}$   $f, g \in \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$

# Asymptotické odhady

- příklad  $g(n) \in \Theta(h(n))$ ,  $g(n) \notin \Theta(f(n))$



# Asymptotické odhady

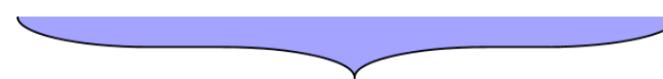
- příklad: Mějme dvojrozměrné pole MxN celých čísel. Jaká je asymptotická složitost problému nalezení největšího čísla v tomto poli?

- horní:

- $O((M+N)^2)$  ✓
- $O(\max(M,N)^2)$  ✓
- $O(N^2)$  ✗
- $O(M*N)$  ✓

- dolní:

- $\Omega(1)$  ✓
- $\Omega(M)$  ✓
- $\Omega(M*N)$  ✓



- optimální:

- $\Theta(M*N)$

# Asymptotické odhady

- O algoritmu se složitostí  $f(n)$  říkáme, že je
  - logaritmický**, pokud  $f(n) \in \Theta(\log(n))$
  - lineární**, pokud  $f(n) \in \Theta(n)$
  - kvadratický**, pokud  $f(n) \in \Theta(n^2)$
  - kubický**, pokud  $f(n) \in \Theta(n^3)$
  - polynomiální**, pokud  $f(n) \in \Theta(n^k)$  pro  $k \in \mathbb{N}$
  - exponenciální**, pokud  $f(n) \in \Theta(k^n)$  pro  $k \in \mathbb{N}$
- Poznámka: U asymptotických odhadů nemá smysl u logaritmických složitostí uvádět základ logaritmu, protože platí  $\log_a(n) \in \Theta(\log_b(n))$  pro libovolná nenulová kladná  $a, b$ .

# Asymptotické odhady

- Jak dokážeme  $\log_a(n) \in \Theta(\log_b(n))$  ?

$$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \cdot \log_b n$$

vzoreček

konstanta

# Vlastnosti asymptotických odhadů

$$n^m \in O(n^{m'}) \text{ pokud } m \leq m'$$

$$f(n) \in O(f(n))$$

$$c \cdot O(f(n)) = O(c \cdot f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$$

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

$$O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$$

- Třídu složitosti polynomu určuje člen s nejvyšší mocninou:

$$\sum_{i=0}^k a_i \cdot n^{k-i} \in \sum_{i=0}^k O(n^k) = k \cdot O(n^k) = O(k \cdot n^k) = O(n^k)$$

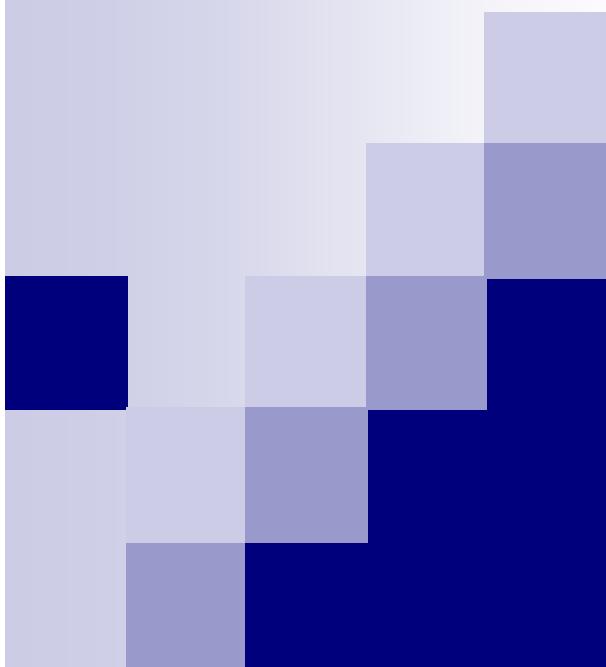
# Vlastnosti asymptotických odhadů

- Věta: Jsou-li funkce  $f(n)$ ,  $g(n)$  vždy kladné, pak pro limitu  $n \rightarrow \infty$  platí
  - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , pak  $f(n) \in O(g(n))$ , ale **neplatí**  $f(n) \in \Theta(g(n))$
  - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = a$ , kde  $0 < a < \infty$ , pak  $f(n) \in \Theta(g(n))$
  - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ , pak  $g(n) \in O(f(n))$ , ale **neplatí**  $g(n) \in \Theta(f(n))$
- Důsledek: Mějme pevně zvolené číslo  $k \in \mathbb{N}$ , pak platí
$$(\log(n))^k \in O(n)$$
- Důkaz lze provést pomocí L'Hopitalova pravidla.

# Vlastnosti asymptotických odhadů

- Dokážeme  $(\ln(n))^2 \in O(n)$

$$\begin{aligned}\lim_{x \rightarrow \infty} \frac{(\ln x)^2}{x} &= \lim_{x \rightarrow \infty} \frac{((\ln x)^2)'}{x'} = \lim_{x \rightarrow \infty} \frac{2 \cdot \frac{1}{x} \cdot \ln x}{1} \\ &= \lim_{x \rightarrow \infty} \frac{2 \cdot \ln x}{x} = \lim_{x \rightarrow \infty} \frac{(2 \cdot \ln x)'}{x'} = \lim_{x \rightarrow \infty} \frac{2}{x} = 0\end{aligned}$$



# Algoritmizace

složitost rekurzivních algoritmů

Jiří Vyskočil, Marko Genyg-Berezovskyj

2010

# Vyjádření složitosti rekurzivního algoritmu rekurentním tvarem

- Příklad vyjádření složitosti rekurzivního algoritmu rekurencí:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1. \end{cases}$$

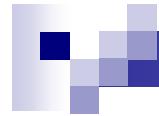
Kde  $T(n)$  je celková složitost algoritmu.

Na pravé straně jsou jednotlivé případy složitostí pro různá  $n$ .

- Okrajové případy (pro  $n <$  konstanta) můžeme opomenout, protože mají konstantní asymptotickou složitost. Zaokrouhlení rovněž většinou neovlivní celkový výsledek (Pozor existují i výjimky!).

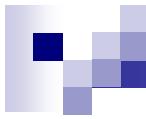
Z toho dostáváme rekurentní vztah:

$$T(n) = 2T(n/2) + \Theta(n),$$



# Převod rekurence na přímé vyjádření

- Přímým vyjádřením složitosti myslíme vyjádření složitosti bez rekurence.
  - Např:  $T(n) = \Theta(\log(n))$
- Jaké jsou možnosti řešení?
  - **Substituční metoda**
    - „Uhádneme“ řešení a potom dokážeme, že je správné indukcí.
  - **Metoda rekurzívного stromу**
    - Spočítáme složitost celého rekurzivního stromu.
  - **Použití „kuchařky“** (Master theorem – mistrovská věta)
    - Pro některé speciální tvary rekurentních vztahů známe předem vypočítané řešení dle mistrovské věty.



# Substituční metoda

- Řešíme ve dvou krocích
  1. **Odhadneme přesný tvar řešení.**
    - Odhad lze stanovit například pomocí zjišťováním složitosti pro různá vstupní  $n$ .
  2. **Matematicky dokážeme, že je náš odhad správný.**
    - Obvykle se dokazuje pomocí matematické indukce.
- Metoda bývá zpravidla velmi účinná.
- Její nevýhodou je určování přesného tvaru řešení v kroku 1 pro které neexistuje obecný postup.

# Substituční metoda - příklad

- Příklad:

$$T(n) = 2T(n/2) + n$$

- Předpokládejme, že jsme odhadli přímé vyjádření vztahem:

$$T(n) = O(n \log(n))$$

- Z definice horního odhadu  $O$ , chceme tedy dokázat, že

$$T(n) \leq cn \log(n)$$

pro nějaké vhodné  $c > 0$ .

- Nyní stanovíme vhodný indukční předpoklad (tj. nechť odhad platí pro  $n/2$ ):

$$T(n/2) \leq c(n/2) \log(n/2)$$

# Substituční metoda - příklad

- Nyní dosadíme indukční předpoklad do rekurentního vztahu a pokusíme se dokázat jeho platnost vyjádřením přímého (nerekurentního) původně odhadnutého vztahu pro  $n$ .

$$\begin{aligned} T(n) &\leq 2(c(n/2) \log(n/2)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log(n) - cn \log(2) + n \\ &= cn \log(n) - cn + n \\ &\leq cn \log(n) \end{aligned}$$

kde poslední krok platí pro  $c \geq 1$ .

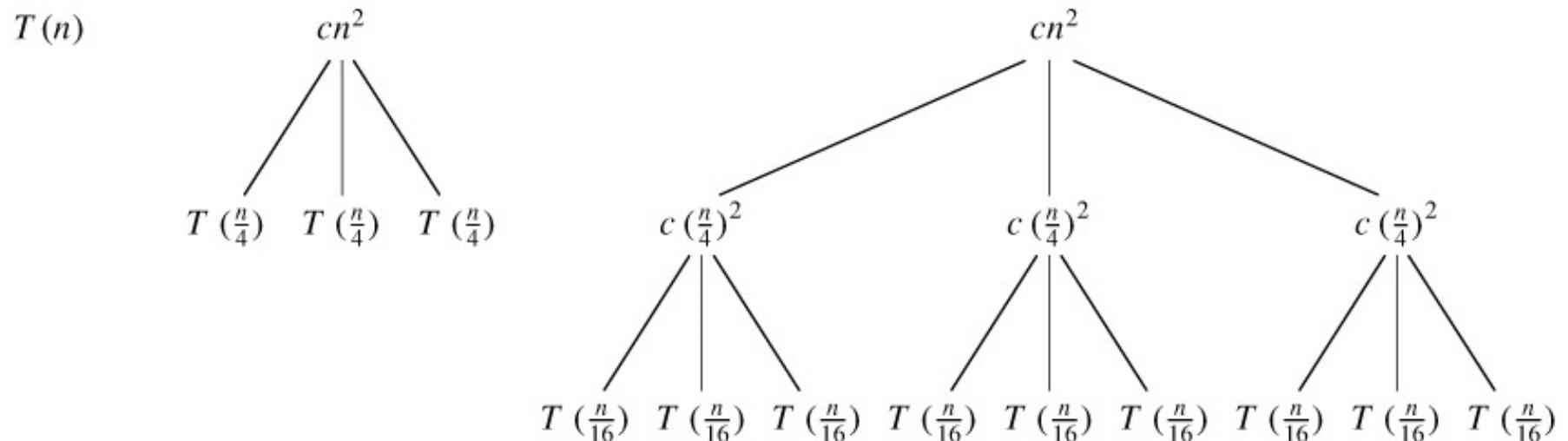
- Počáteční krok indukce platí triviálně. Díky asymptotické notaci stačí ukázat, že odhad platí pro nějaké  $n_0$  a  $c > 0$ . V našem příkladě tedy platí pro  $n_0=3$  a  $c \geq 2$ ).
- Tím je důkaz hotov.

# Metoda rekurzívního stromu - příklad

- Příklad:

$$T(n) = 3T(n/4) + cn^2$$

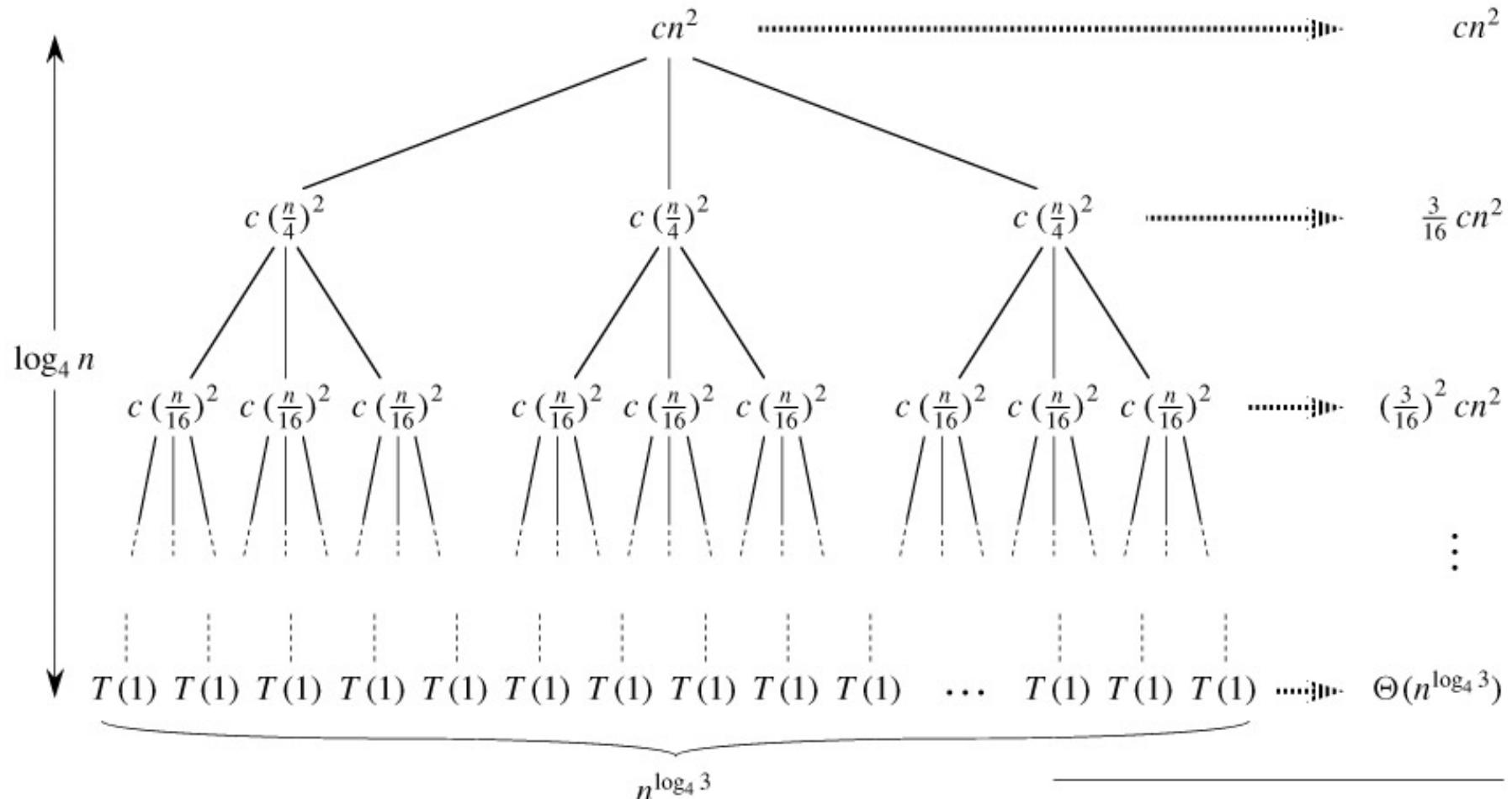
- Iterativně rozkládáme do rekurzivních stromů:



- Pro každý strom platí, že součet všech uzlů dá složitost  $T(n)$  podle původního rekurentního vztahu.
- Rekurzivní stromy jsou pouze grafická vizualizace rozvoje rekurentního vztahu.

# Metoda rekurzívního stromu - příklad

- Výsledný strom má následující tvar:



- Vyjádříme součty jednotlivých pater stromu.
- Všechna patra sečteme a dostaneme výslednou složitost:

$O(n^2)$

# Metoda rekurzívního stromu - příklad

- Součet pater lze spočítat následovně:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \quad \text{podle vzorce } \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \quad \text{pro } |x| < 1 \\ &= O(n^2). \end{aligned}$$

# Použití „kuchařky“

- Použití „kuchařky“ nebo tzv. mistrovské věty (master theorem) řeší rekurentní složitost, která má následující tvar:

$$T(n) = a T(n/b) + f(n)$$

Kde  $a \geq 1$  a  $b > 1$  jsou konstanty

a  $f(n)$  je asymptoticky kladná funkce.

- Zaokrouhlení u členu  $T(n/b)$  na  $T(\lfloor n/b \rfloor)$  nebo  $T(\lceil n/b \rceil)$  neovlivní v tomto případě výslednou složitost.

# Použití „kuchařky“

- Master theorem (mistrovská nebo také kuchařková věta)
  - Nechť jsou  $a \geq 1$  a  $b > 1$  konstanty, nechť je  $f(n)$  funkce a nechť  $T(n)$  je definováno pro nezáporná celá čísla rekurencí

$$T(n) = a T(n/b) + f(n)$$

kde  $n/b$  má význam bud'  $\lceil n/b \rceil$  nebo  $\lfloor n/b \rfloor$ . Potom lze asymptoticky vyjádřit následovně:

1. Pokud  $f(n) \in O(n^{\log_b(a)-\varepsilon})$  pro nějakou konstantu  $\varepsilon > 0$ , potom

$$T(n) \in \Theta(n^{\log_b(a)}).$$

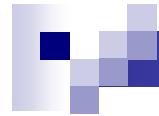
2. Pokud  $f(n) \in \Theta(n^{\log_b(a)})$ , potom

$$T(n) \in \Theta(n^{\log_b(a)} \log(n)).$$

3. Pokud  $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$  pro nějakou konstantu  $\varepsilon > 0$  a pokud

$a f(n/b) \leq c f(n)$  pro nějakou konstantu  $c < 1$  a všechna dostatečně velké  $n$ , potom

$$T(n) \in \Theta(f(n)).$$



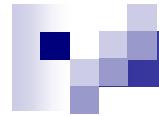
# Použití „kuchařky“ – příklad 1

- Příklad 1:

$$T(n) = 9T(n/3) + n$$

- Z toho dostáváme, že  $a = 9$ ,  $b = 3$ ,  $f(n) = n \in O(n^{\log_3(9)-1})$ .  
Jedná se tedy o případ číslo 1.
- Dostáváme tedy složitost:

$$T(n) \in \Theta(n^{\log_3(9)}) = \Theta(n^2)$$



# Použití „kuchařky“ – příklad 2

- Příklad 2:

$$T(n) = T(2n/3) + 1$$

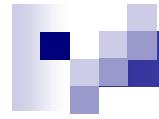
- Z toho dostáváme, že  $a = 1$ ,  $b = 3/2$ ,

$$f(n) = 1 = n^{\log_{3/2}(1)} \in \Theta(n^{\log_{3/2}(1)}).$$

Jedná se tedy o případ číslo 2.

- Dostáváme tedy složitost:

$$T(n) \in \Theta(n^{\log_{3/2}(1)} \log(n)) = \Theta(\log(n))$$



# Použití „kuchařky“ – příklad 3

- Příklad 3:

$$T(n) = 3T(n/4) + n \log(n)$$

- Z toho dostáváme, že  $a = 3$ ,  $b = 4$ ,

$$f(n) = n \log(n) \text{ a víme, že } n^{\log_4(3)} = O(n^{0.793}) .$$

Platí tedy, že  $f(n) \in \Omega(n^{\log_4(3)+0.2})$ .

Pokud by se mělo jednat o případ 3 musí ještě platit pro  $c < 1$  a všechna dostatečně velká  $n$ , že  $af(n/b) \leq cf(n)$  tedy  $af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \log(n) = cf(n)$  pro  $c = 3/4$ .

- Dostáváme tedy složitost:

$$T(n) \in \Theta(n \log(n))$$

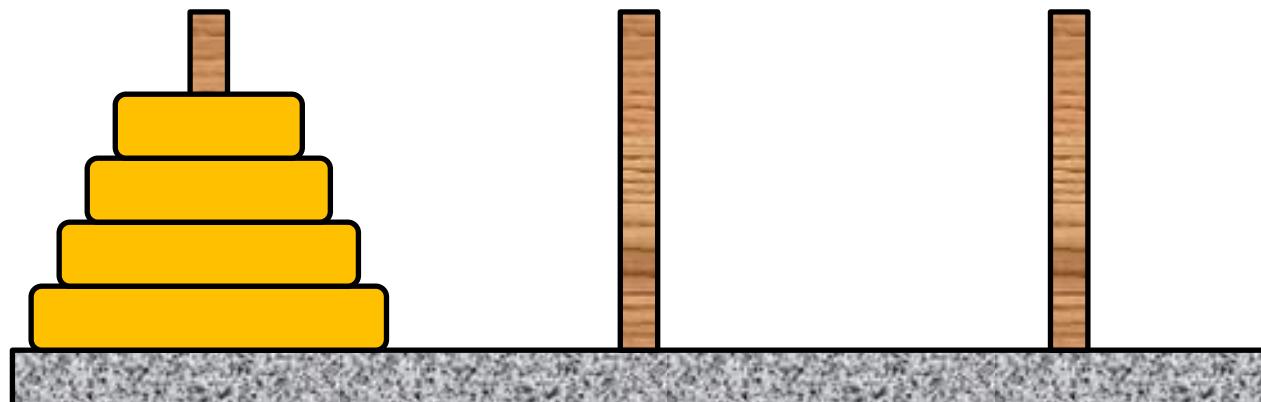
# Hanojská věž

---

Úkol: přemístit disky z tyče vlevo na tyč vpravo

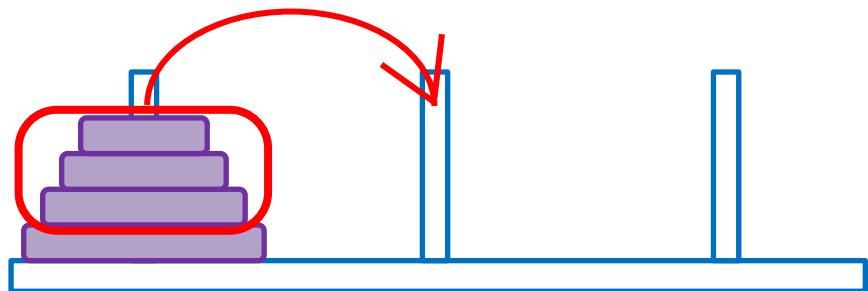
Omezení:

- disky přesouváme pouze jednotlivě, z tyče na tyč
- větší disk nesmí nikdy ležet na menším disku

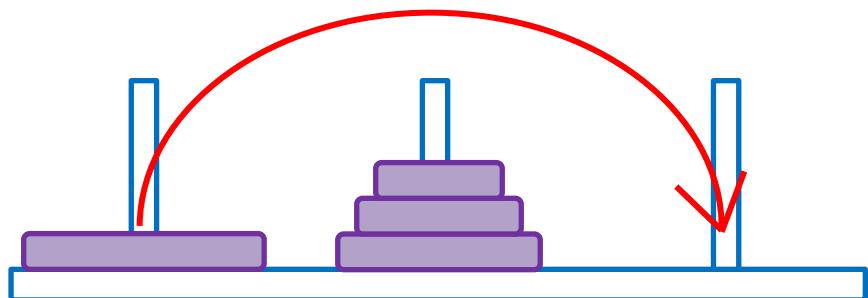


# Hanojská věž – rekurzivní řešení

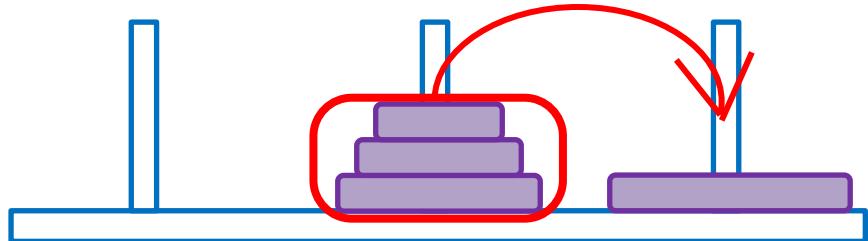
1)



2)



3)



```
Hanoj(n, t1, t2, t3):  
    if n>0 then  
        Hanoj(n-1, t1, t3, t2);  
        přesuň disk z t1 na t3;  
        Hanoj(n-1, t2, t1, t3);  
    endif
```

# Hanojská věž – výpočet složitosti

---

Časová složitost je úměrná počtu tahů (přesunů), které provedeme.

Rekurentní vztah:

$$T(1) = 1$$

$$T(n) = 2T(n-1) + 1$$

odpovídá počtu vrcholů v úplném binárním stromě hloubky  $n$

$$T(n) = 2^{n-1} + 2^{n-2} + \dots + 2^1 + 1$$

$$T(n) = 2^n - 1 = O(2^n)$$

$$T(64) = 18\,446\,744\,073\,709\,551\,615$$

Trvá-li jeden tah sekundu, pak je potřeba 600 miliard let.

## ALGORITMIZACE 2010/03

**STROMY, BINÁRNÍ STROMY  
VZTAH STROMŮ A REKURZE  
ZÁSOBNÍK IMPLEMENTUJE REKURZI  
PROHLEDÁVÁNÍ S NÁVRATEM (BACKTRACK)**

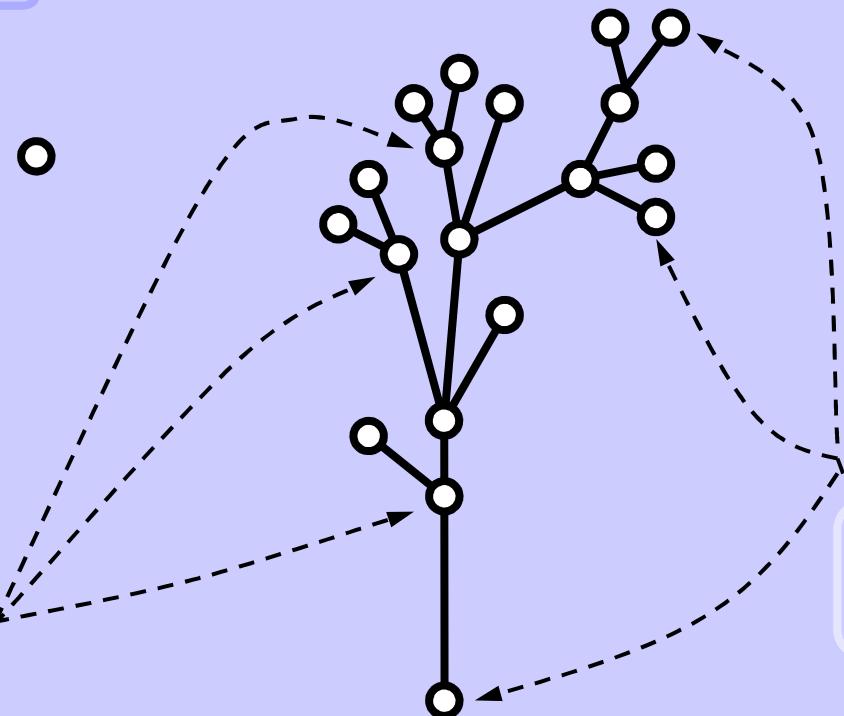
### Strom / tree

uzel, vrchol /  
node, vertex

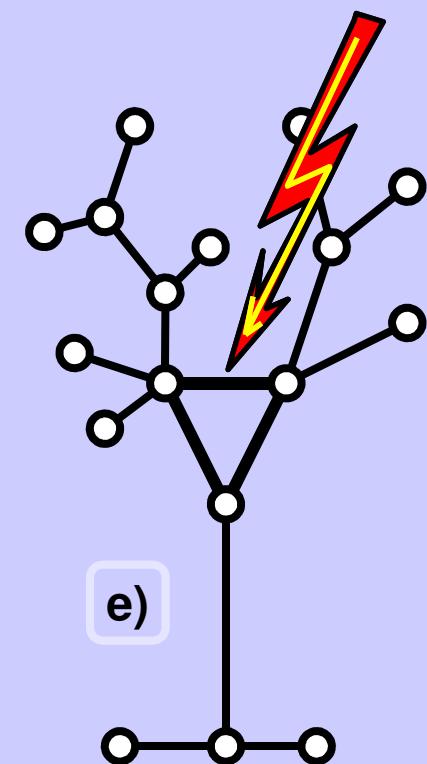
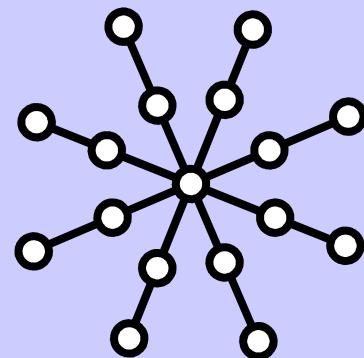
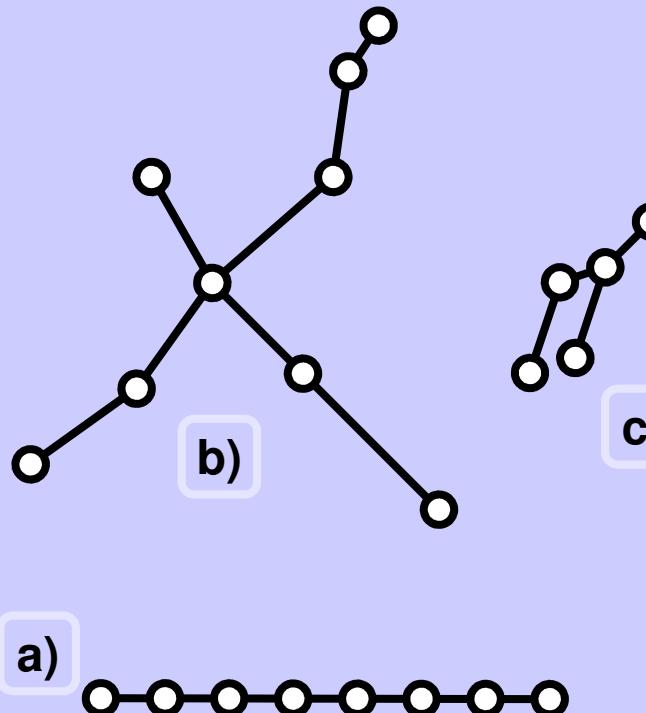
hrana / edge

vnitřní uzel /  
internal node

list /  
leaf (pl. leaves)



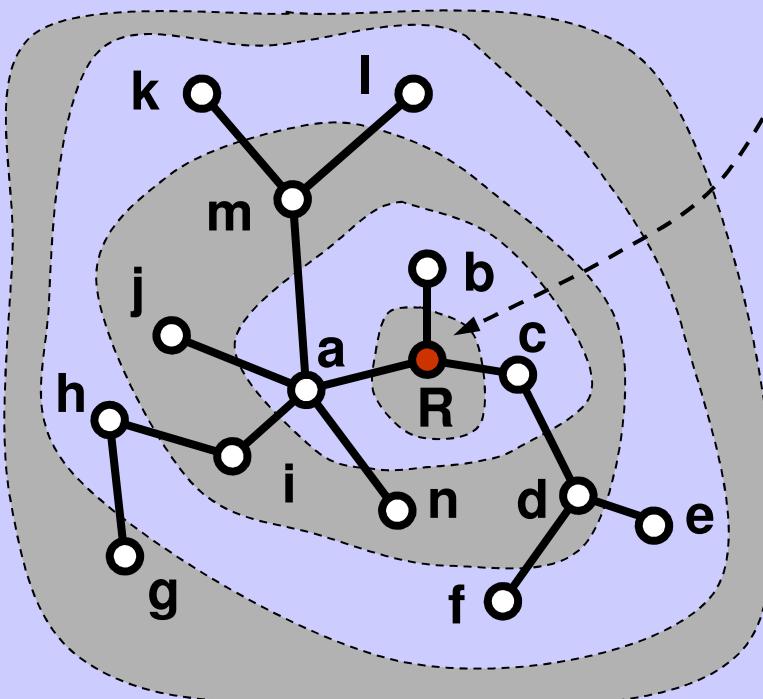
## Příklady stromů



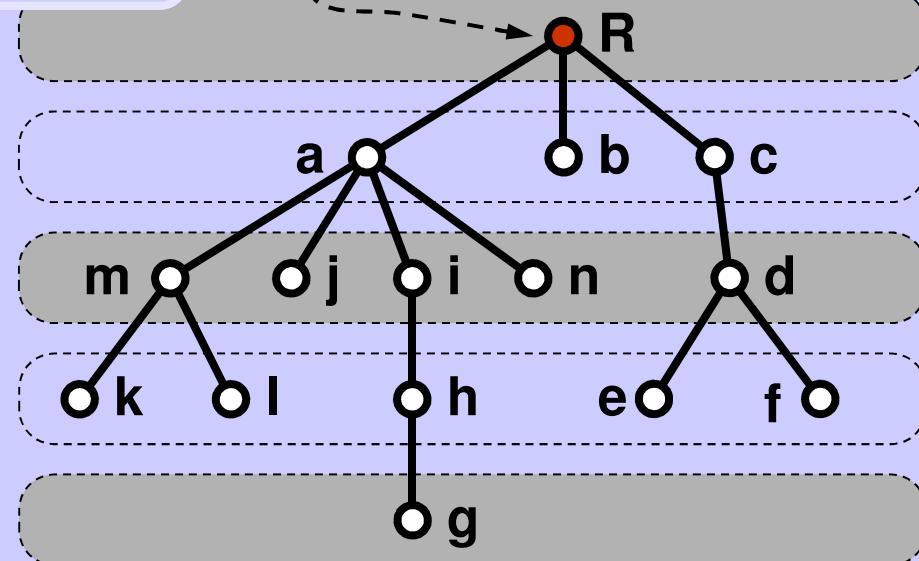
## Vlastnosti stromů

1. Strom je souvislý, tj. mezi každými dvěma jeho uzly vede cesta.
2. Mezi každými dvěma uzly ve stromu vede jen jediná cesta.
3. Po odstranění libovolné hrany se strom rozpadá na dvě části.
4. Počet hran ve stromu je vždy o 1 menší než počet uzelů.

## Kořenový strom / rooted tree



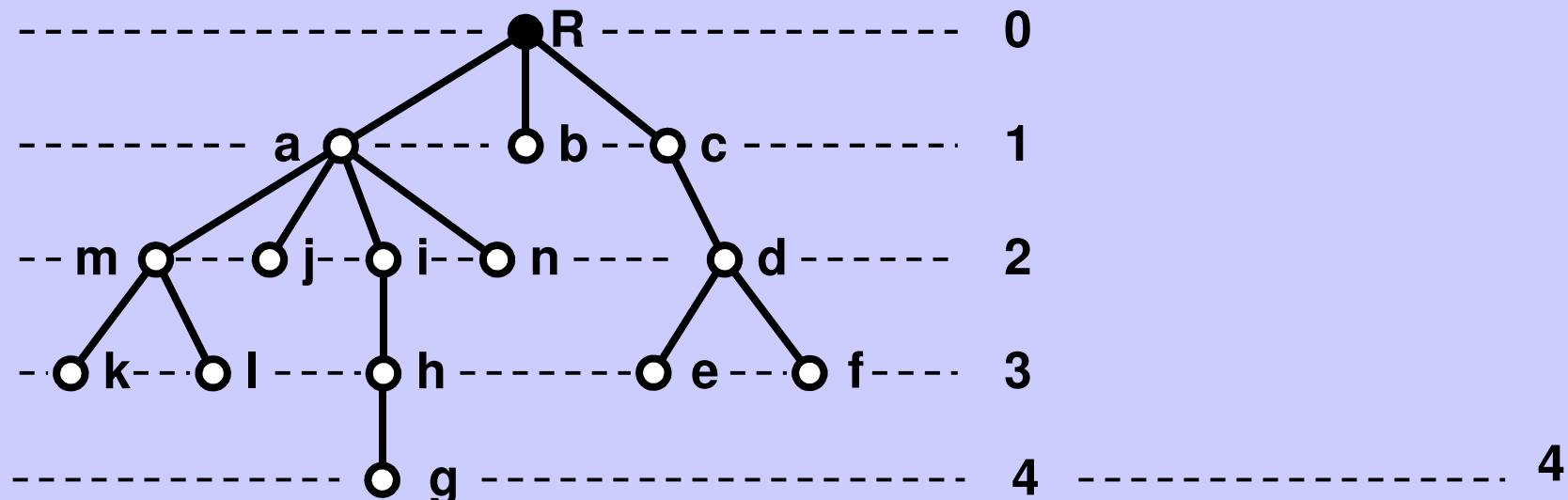
Kořen /  
root



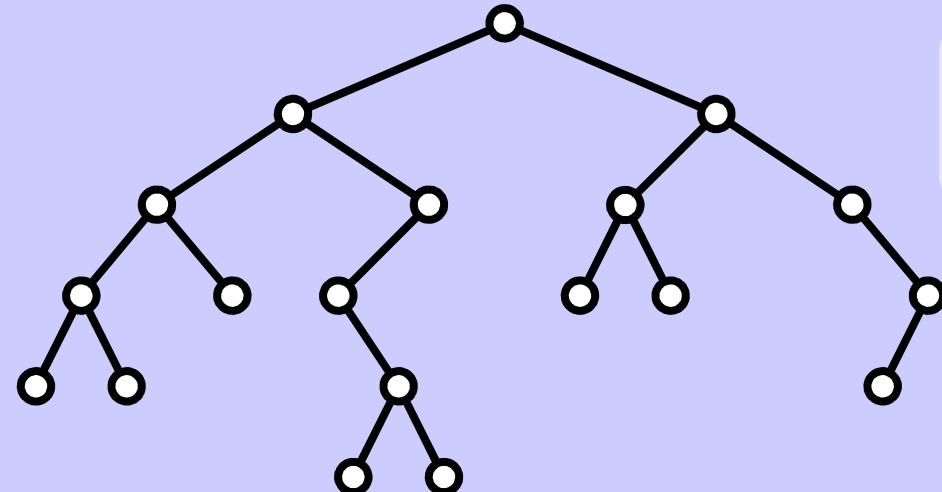
## Názvosloví



Hloubka / depth

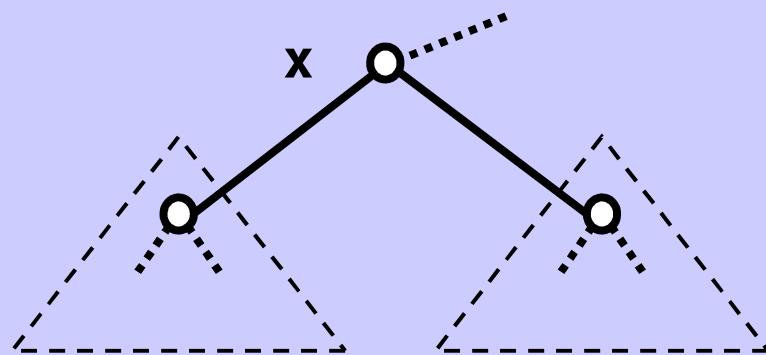


## Binární (kořenový!!) strom / binary (rooted!!) tree



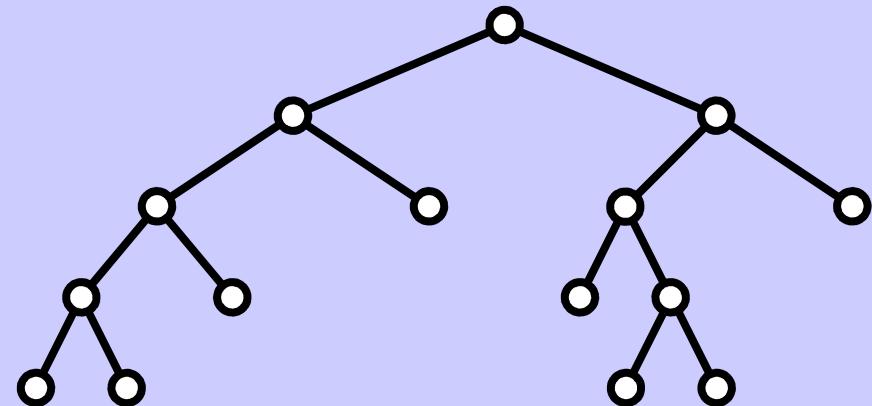
Počet potomků každého uzlu je 0,1, nebo 2.

## Levý a pravý podstrom / left and right subtree



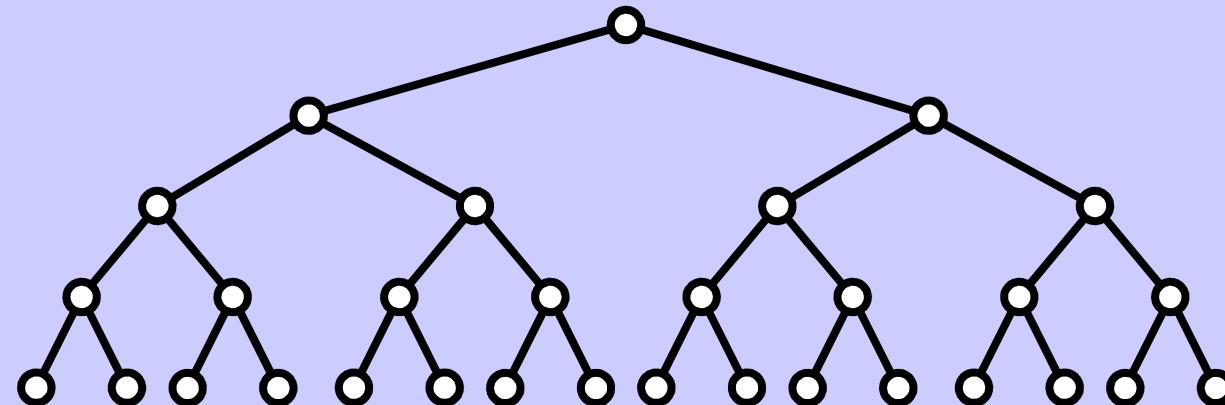
Podstrom uzlu x ..... levý ..... právý

### Pravidelný binární strom / regular binary tree



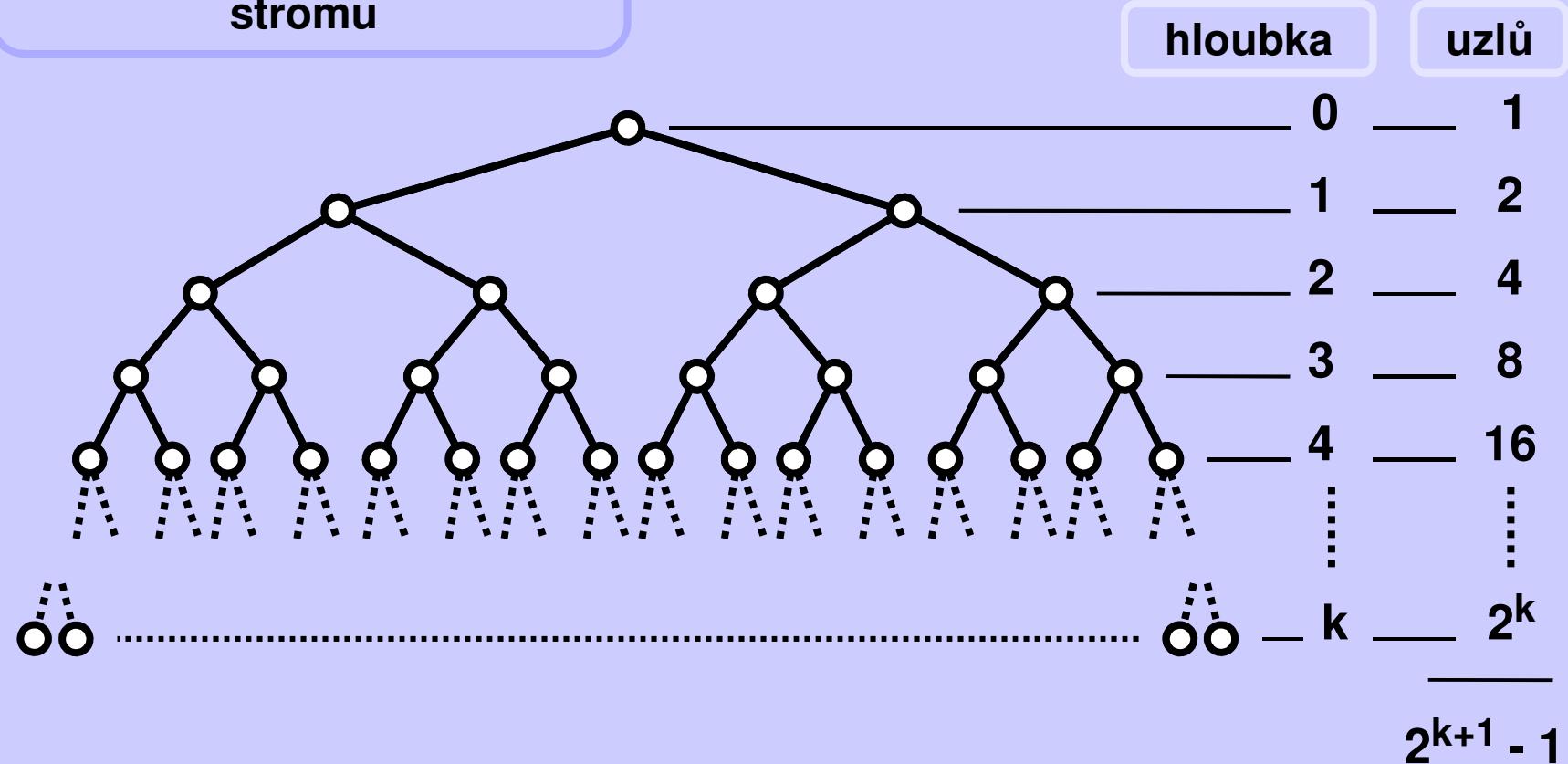
Počet potomků každého uzlu  
je jen 0 nebo 2.

### Vyvážený strom / balanced tree



Hloubky všech listů jsou (víceméně) stejné.

**Hloubka vyváženého  
(binárního a pravidelného !)  
stromu**



$$(2^{(\text{hloubka vyváženého stromu})+1} - 1) \sim \text{uzlů}$$

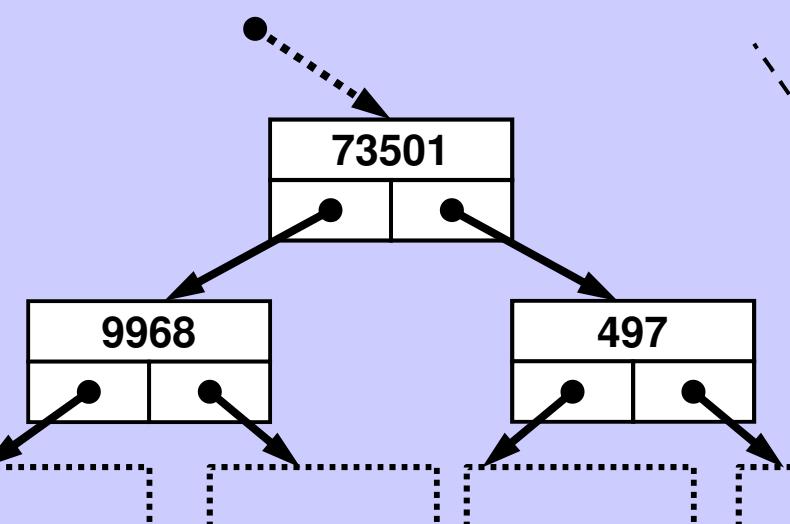
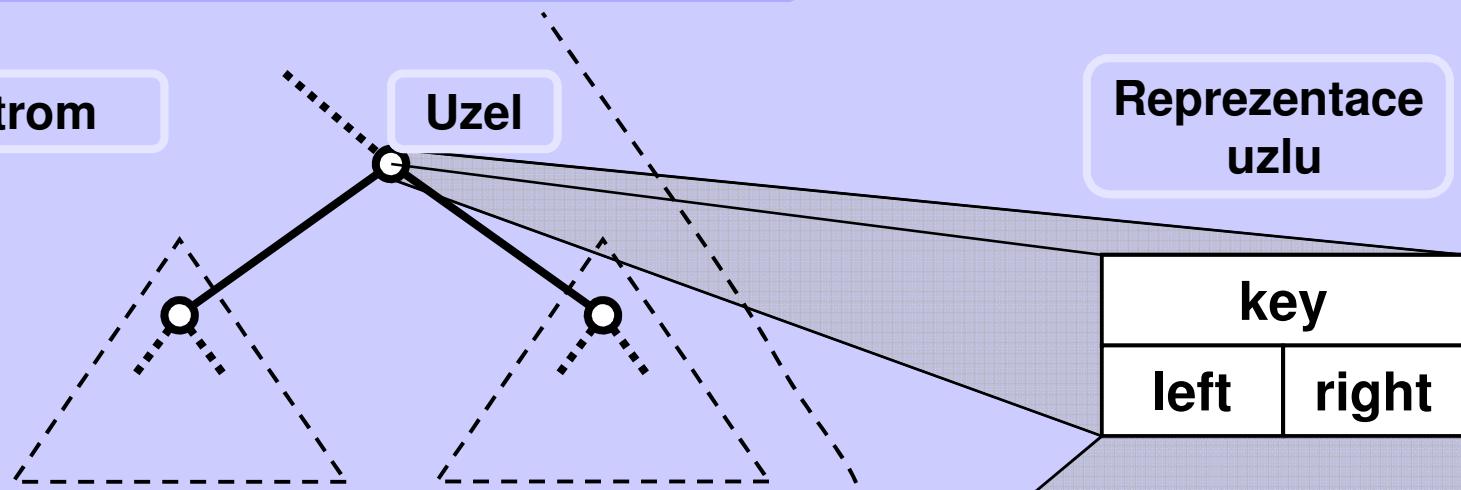
$$\text{hloubka vyváženého stromu} \sim \log_2(\text{uzlů}+1) - 1 \sim \log_2(\text{uzlů})$$

## Implementace binárního stromu -- C

Strom

Uzel

Reprezentace  
uzlu

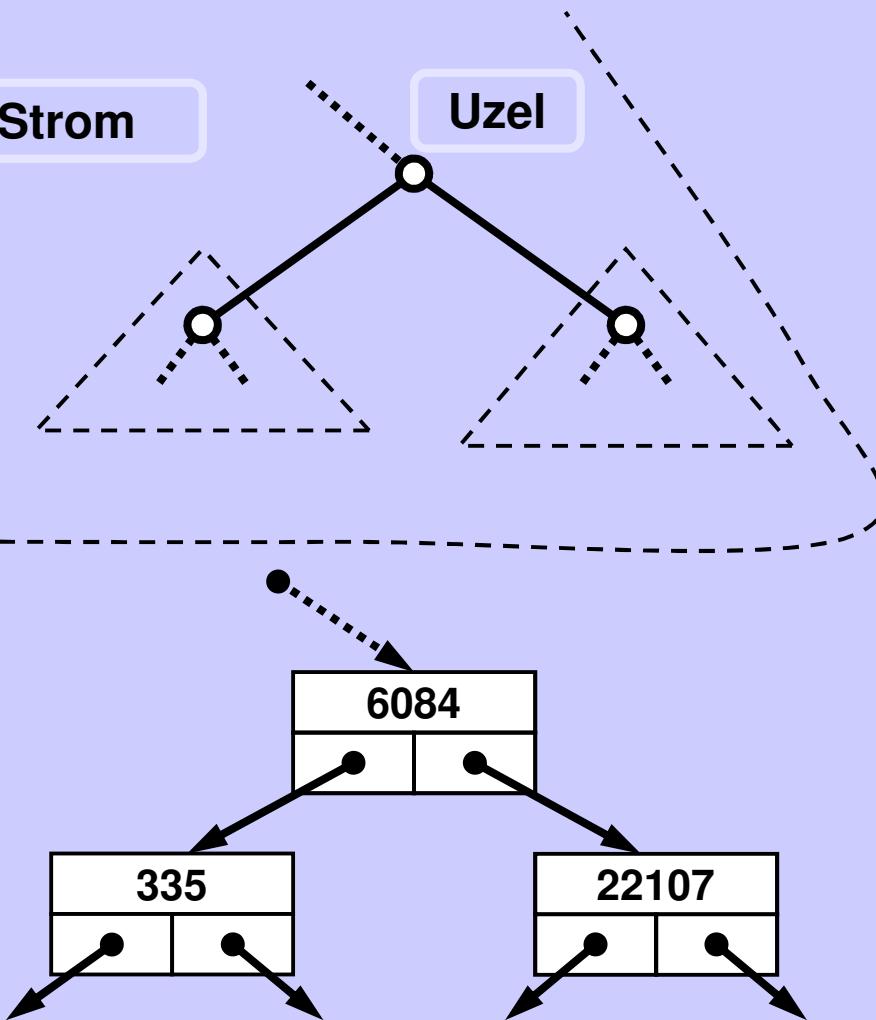


```
typedef struct node {  
    int key;  
    struct node *left;  
    struct node *right;  
} NODE;
```

## Implementace binárního stromu -- Java

Strom

Uzel



```

public class Node {
    public Node left;
    public Node right;
    public int key;
    public Node(int k) {
        key = k;
        left = null;
        right = null;
    }
}

public class Tree {
    public Node root;
    public Tree() {
        root = null;
    }
}

```

## Vybudování náhodného binárního stromu -- C

```

NODE *randTree(int depth) {
    NODE *pnode;
    if ((depth <= 0) || (random(10) > 7))
        return (NULL);                                //stop recursion
    pnode = (NODE *) malloc(sizeof(NODE)); // create node
    if (pnode == NULL) {
        printf("%s", "No memory.");
        return NULL;
    }
    pnode->left = randTree(depth-1);           // make left subtree
    pnode->key = random(100);                  // some value
    pnode->right = randTree(depth-1);          // make right subtree
    return pnode;                            // all done
}

```

Příklad  
volání funkce

```

NODE *root;
root = randTree(4);

```

Poznámka. Volání random(n) vrací náhodné celé číslo od 0 do n-1.  
Zde neimplementováno.

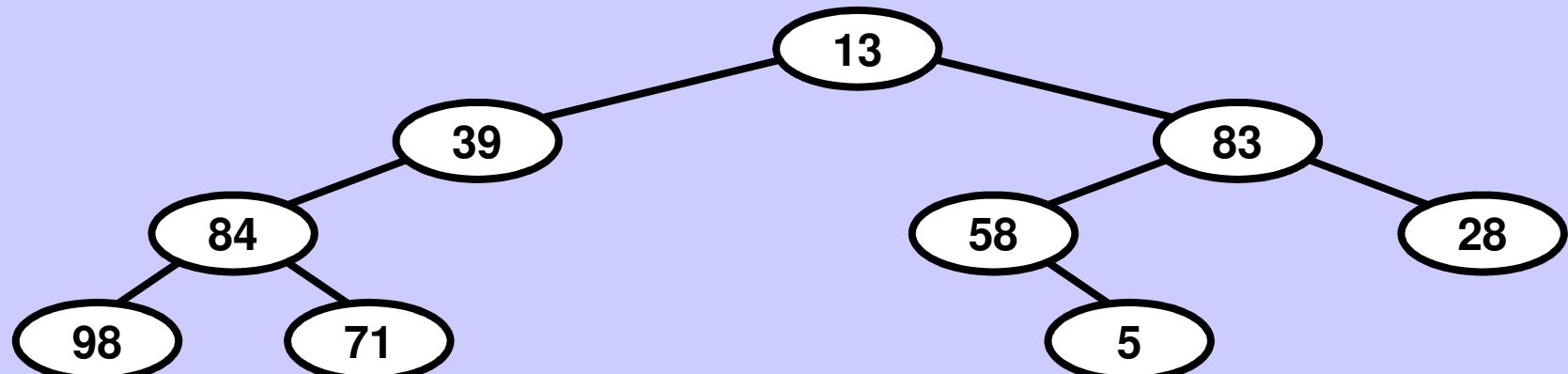
## Vybudování náhodného binárního stromu -- Java

```
public Node randTree(int depth) {  
    Node node;  
    if ((depth <= 0) || ((int) Math.random()*10 > 7))  
        return null; // create node with a key value  
    node = new Node((int) (Math.random()*100));  
  
    node.left = randTree(depth-1); // create left subtree  
    node.right = randTree(depth-1); // create right subtree  
    return node; // all done  
}
```

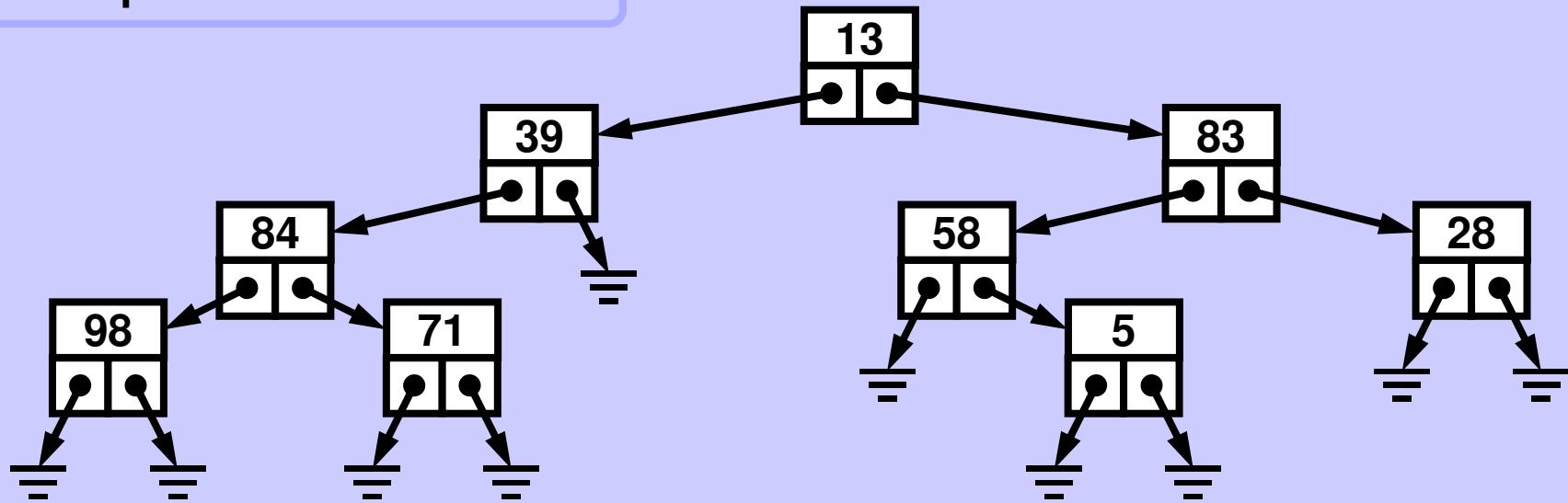
Příklad  
volání funkce

```
Node root;  
root = randTree(4);
```

## Náhodný binární strom

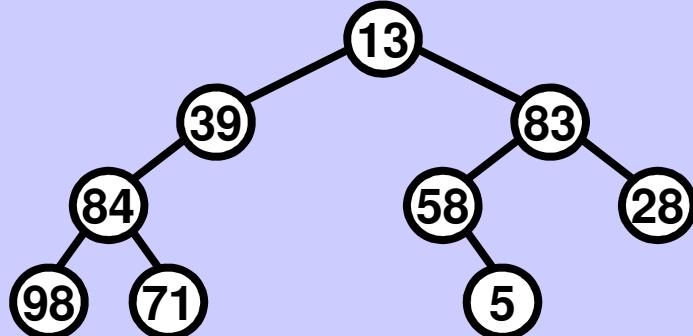


## Reprezentace stromu



## Průchod (binárním) stromem v pořadí Inorder

Strom



Průchod  
stromem  
v pořadí  
**INORDER**

```
void listInorder( NODE *ptr) {  
    if (ptr == NULL) return;  
    listInorder(ptr->left);  
    printf("%d ", ptr->key);  
    listInorder(ptr->right);  
}
```

Výstup

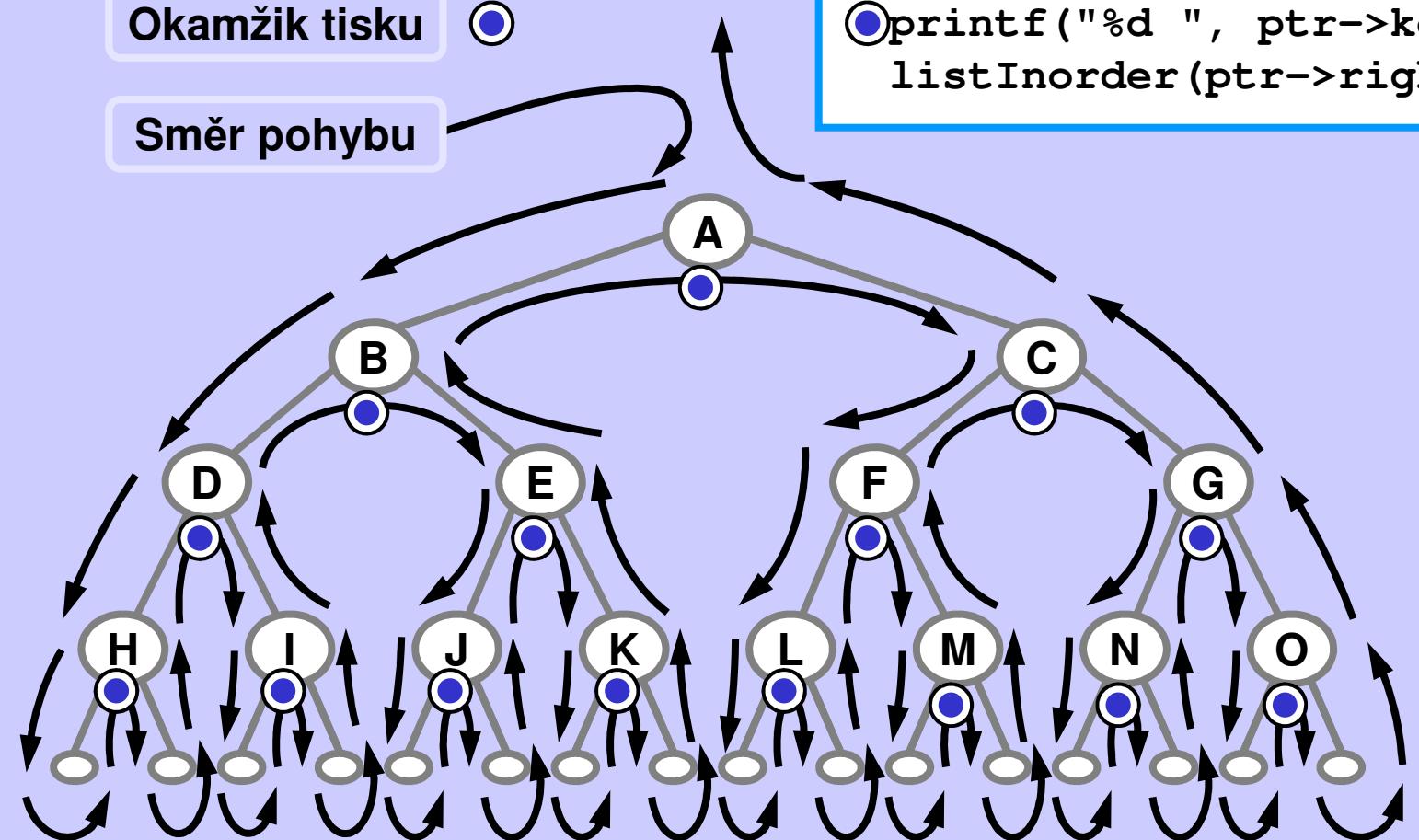
98 84 71 39 13 58 5 83 28

## Pohyb ve stromu v průchodu Inorder

Okamžik tisku

Směr pohybu

```
listInorder(ptr->left);
printf("%d ", ptr->key);
listInorder(ptr->right);
```

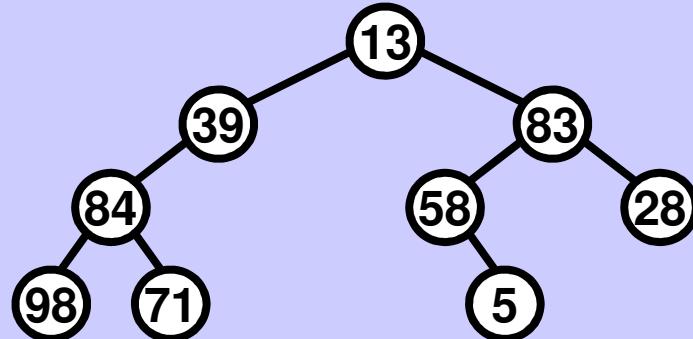


Výstup

H D I B J E K A L F M C N G O

## Průchod (binárním) stromem v pořadí Preorder

Strom



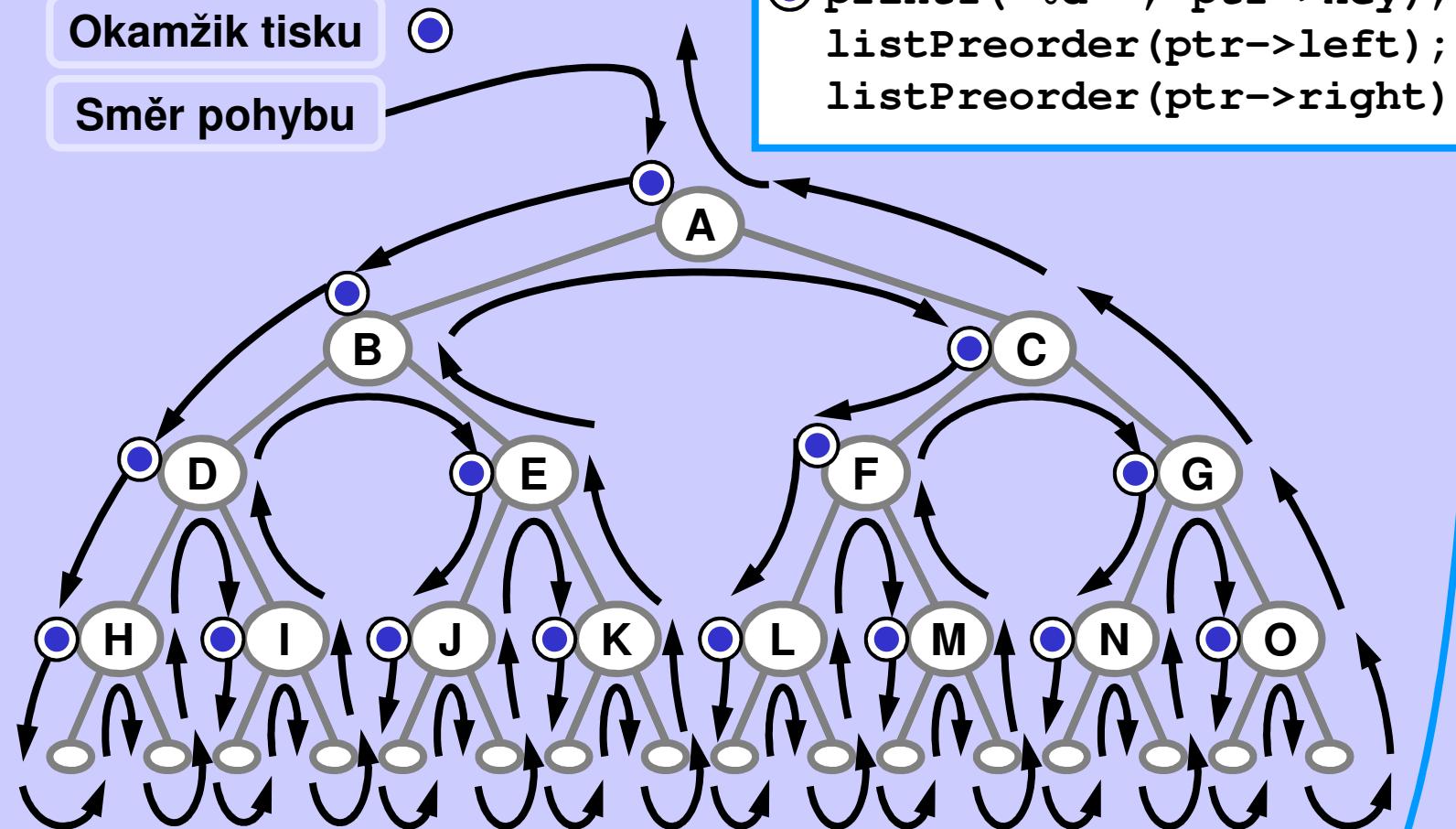
Průchod  
stromem  
v pořadí  
**PREORDER**

```
void listPreorder( NODE *ptr) {  
    if (ptr == NULL) return;  
    printf("%d ", ptr->key);  
    listPreorder(ptr->left);  
    listPreorder(ptr->right);  
}
```

Výstup

13 39 84 98 71 83 58 5 28

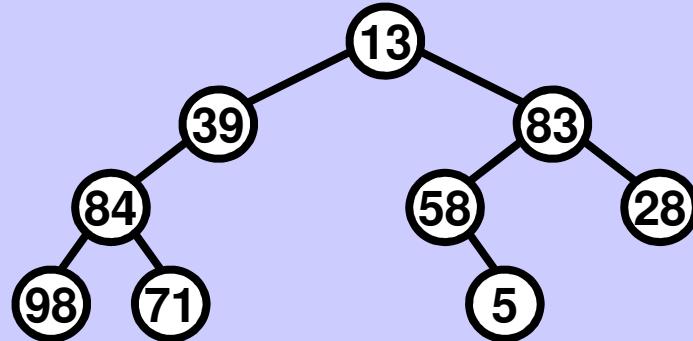
## Pohyb ve stromu v průchodu Preorder



Výstup A B D H I E J K C F L M G N O

## Průchod (binárním) stromem v pořadí Postorder

Strom



Průchod  
stromem  
v pořadí

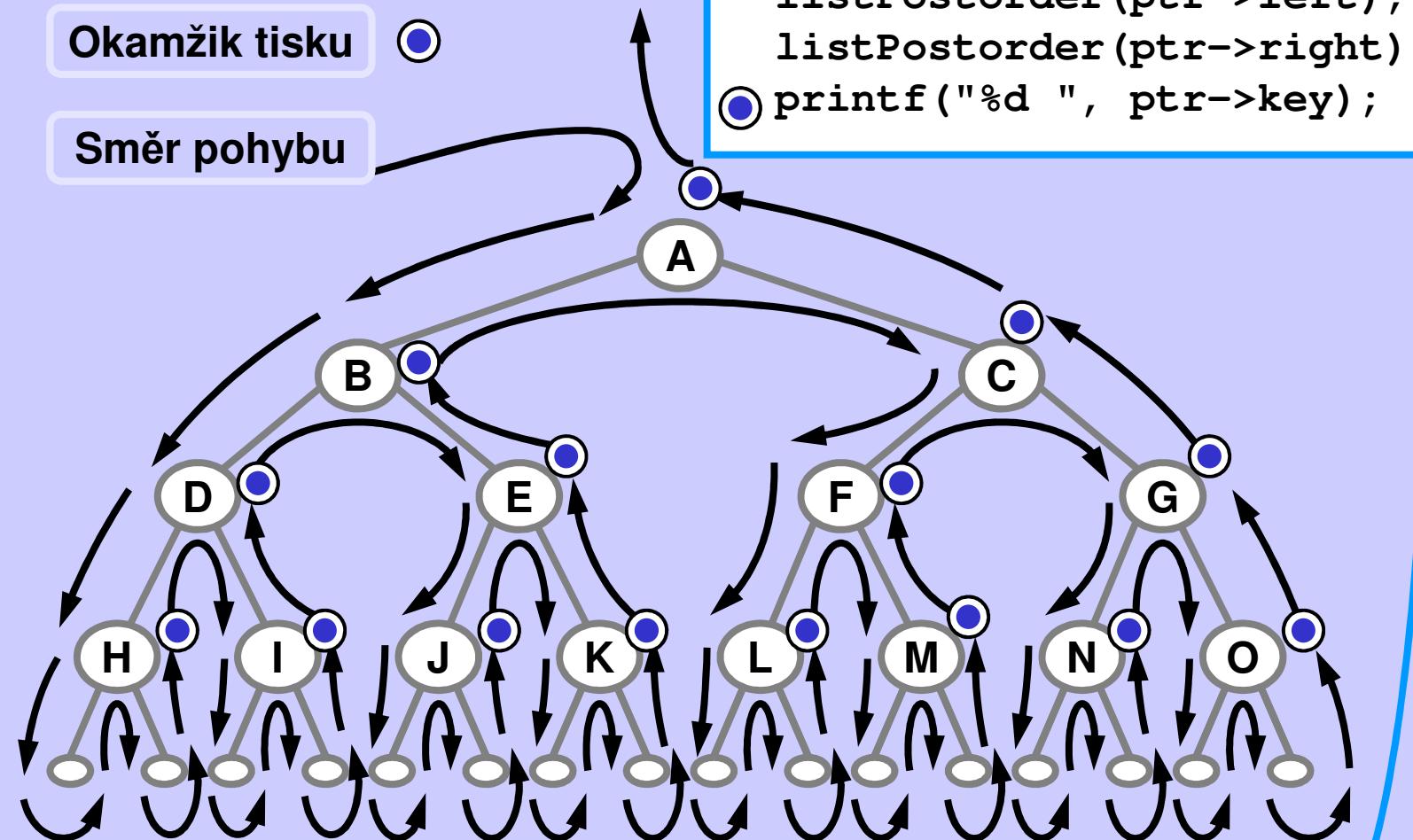
POSTORDER

Výstup

```
void listPostorder( NODE *ptr) {  
    if (ptr == NULL) return;  
    listPostorder(ptr->left);  
    listPostorder(ptr->right);  
    printf("%d ", ptr->key);  
}
```

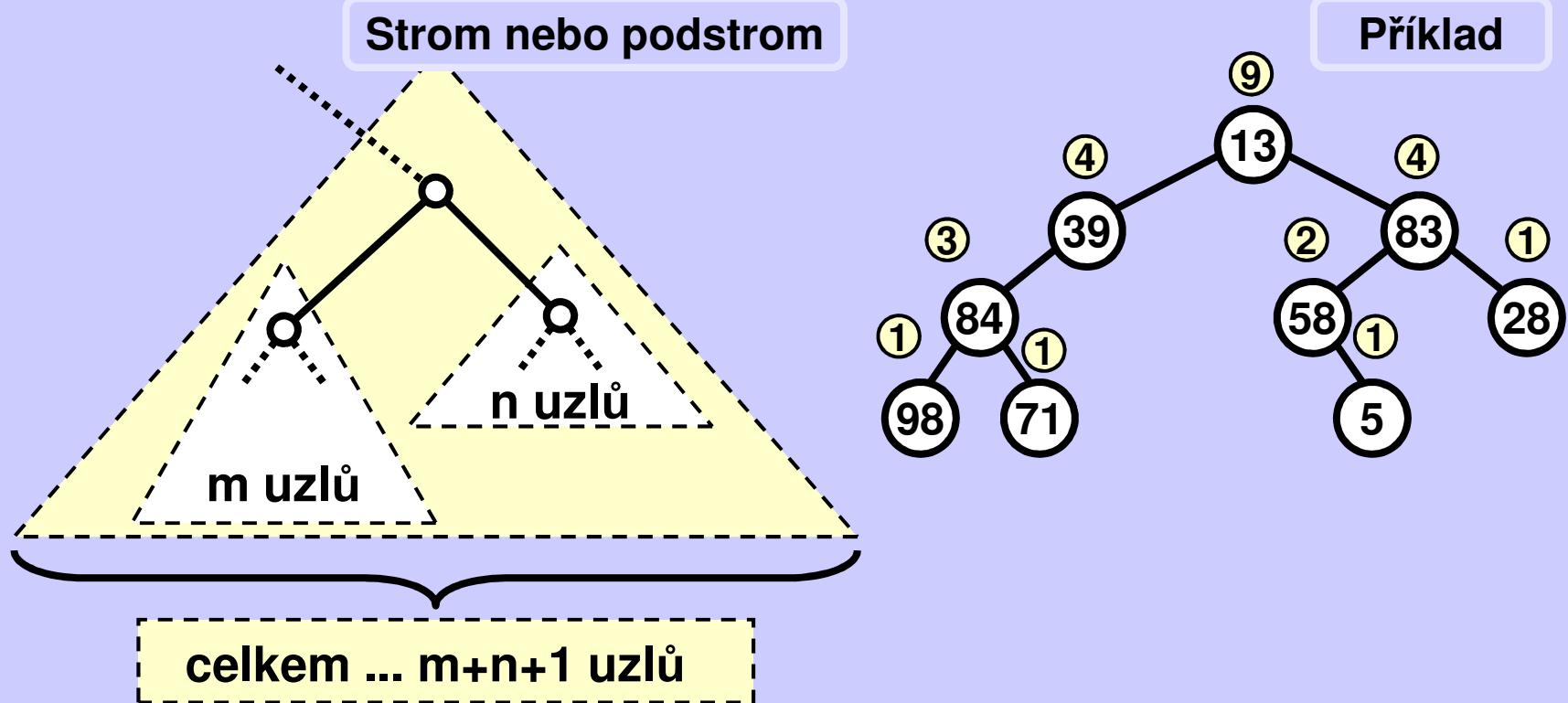
98 71 84 39 5 58 28 83 13

## Pohyb ve stromu v průchodu Postorder



Výstup H I D J K E B L M F N O G C A

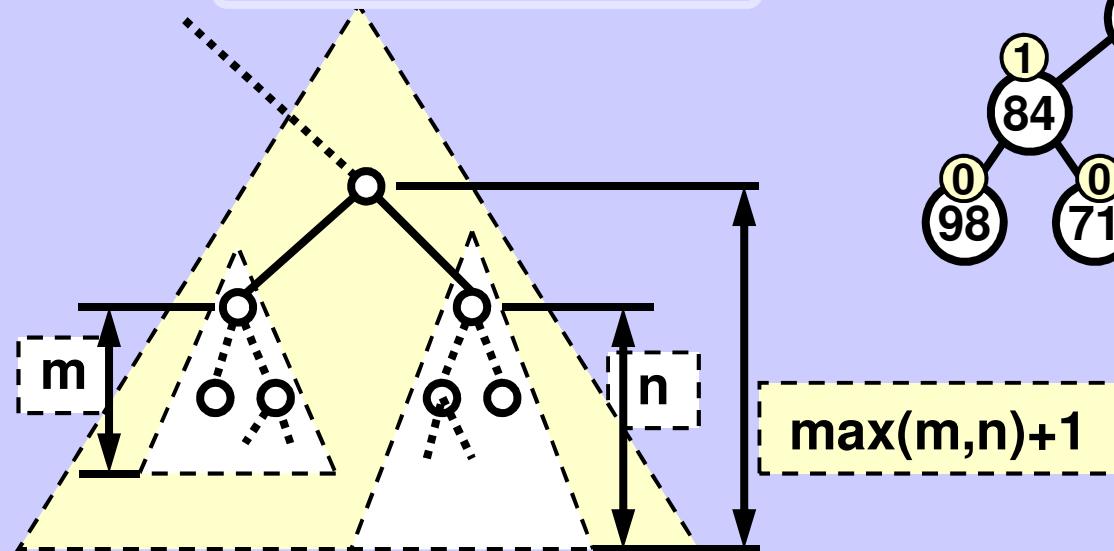
## Velikost stromu (= počet uzlů) rekuzivně



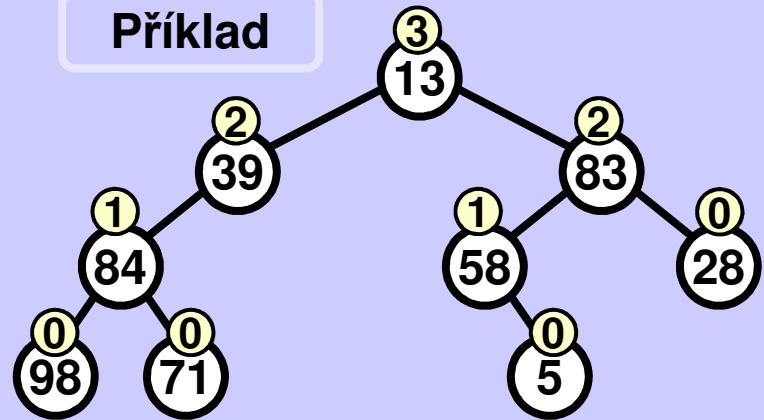
```
int count(NODE *ptr) {
    if (ptr == NULL) return (0);
    return (count(ptr->left) + count(ptr->right)+1);
}
```

## Hloubka stromu (= max hloubka uzlu) rekurzivně

### Strom nebo podstrom



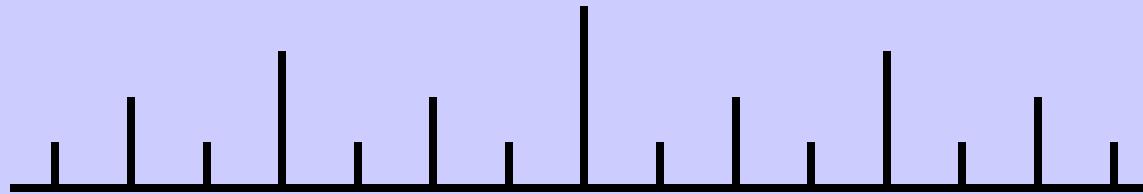
### Příklad



```
int depth(NODE *ptr) {
    if (ptr == NULL) return (-1);
    return ( max(depth(ptr->left), depth(ptr->right)) +1 );
}
```

## Jednoduchý příklad rekurze

**Binární pravítka**



**Rysky pravítka**

**Délky rysek**

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

**Kód vypíše délky rysek pravítka**

```
void ruler(int val) {
    if (val < 1) return;

    ruler(val-1);
    print(val);
    ruler(val-1);
}
```

---

Call: ruler(4);

Domácí úkol: Ternárně!:



## Jednoduchý příklad rekurze

### Binární pravítko vs. průchod inorder

#### Pravítko

```
void ruler(int val) {
    if (val < 1) return;

    ruler(val-1);
    print(val);
    ruler(val-1);
}
```

#### Inorder

```
void listInorder( NODE *ptr) {
    if (ptr == NULL) return;

    listInorder(ptr->left);
    printf("%d ", ptr->key);
    listInorder(ptr->right);
}
```

Strukturní podobnost, shoda!

Výstup pravítka

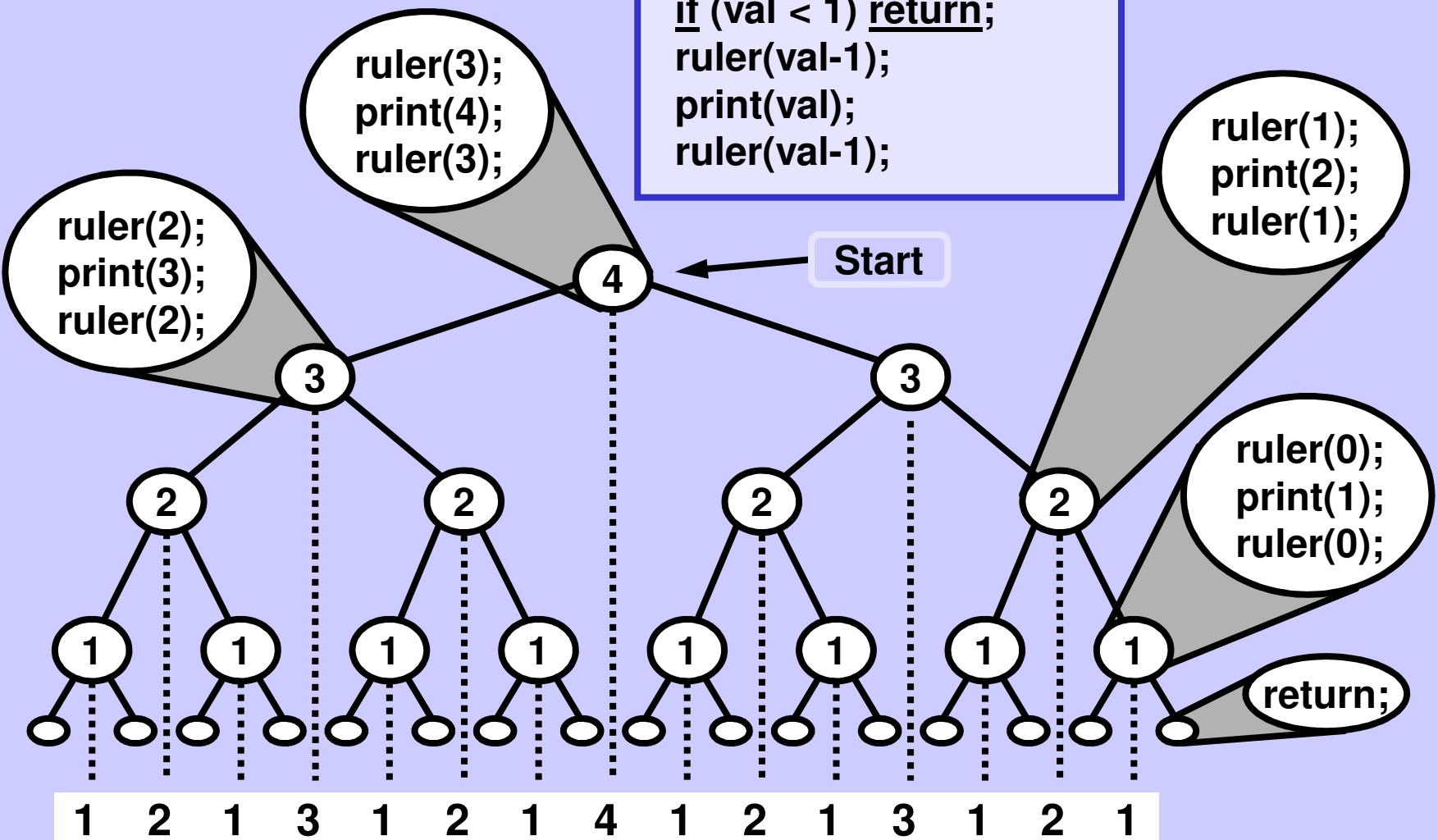
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

## Jednoduchý příklad rekurze

Činnost binárního pravítka

Kód

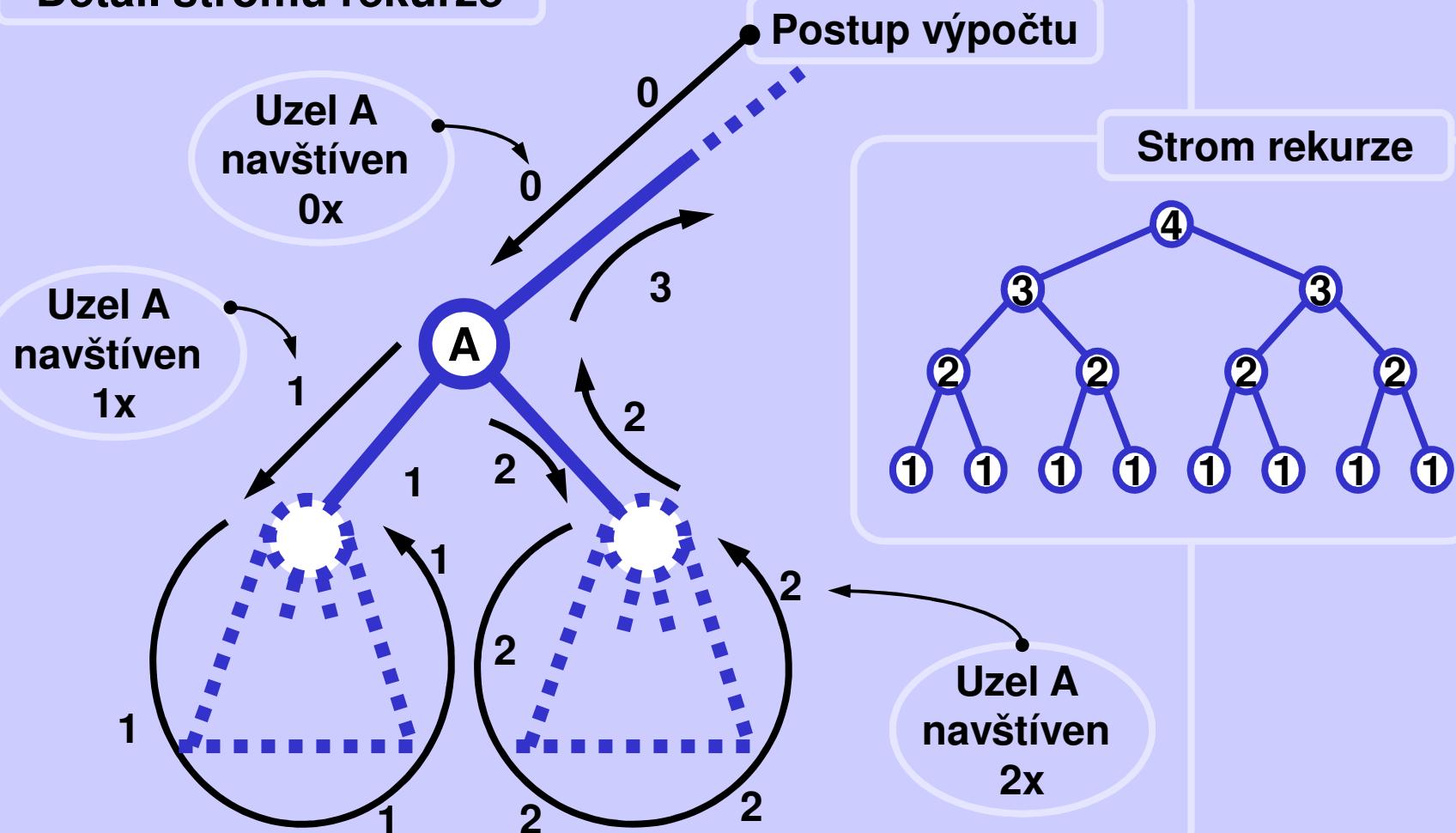
```
if (val < 1) return;
ruler(val-1);
print(val);
ruler(val-1);
```



# Zásobník implementuje rekurzi

Binární pravítko bez rekurze

Detail stromu rekurze



## Zásobník implementuje rekurzi

### Standardní strategie

Při používání zásobníku:

**Je-li to možné, zpracovávej jen data ze zásobníku.**

### Standardní postup

**Ulož první uzel (první zpracovávaný prvek) do zásobníku.**

**Každý další uzel (zpracovávaný prvek) ulož také na zásobník.**

**Zpracovávej vždy pouze uzel na vrcholu zásobníku.**

**Když jsi s uzlem (prvkem) hotov, ze zásobníku ho odstraň.**

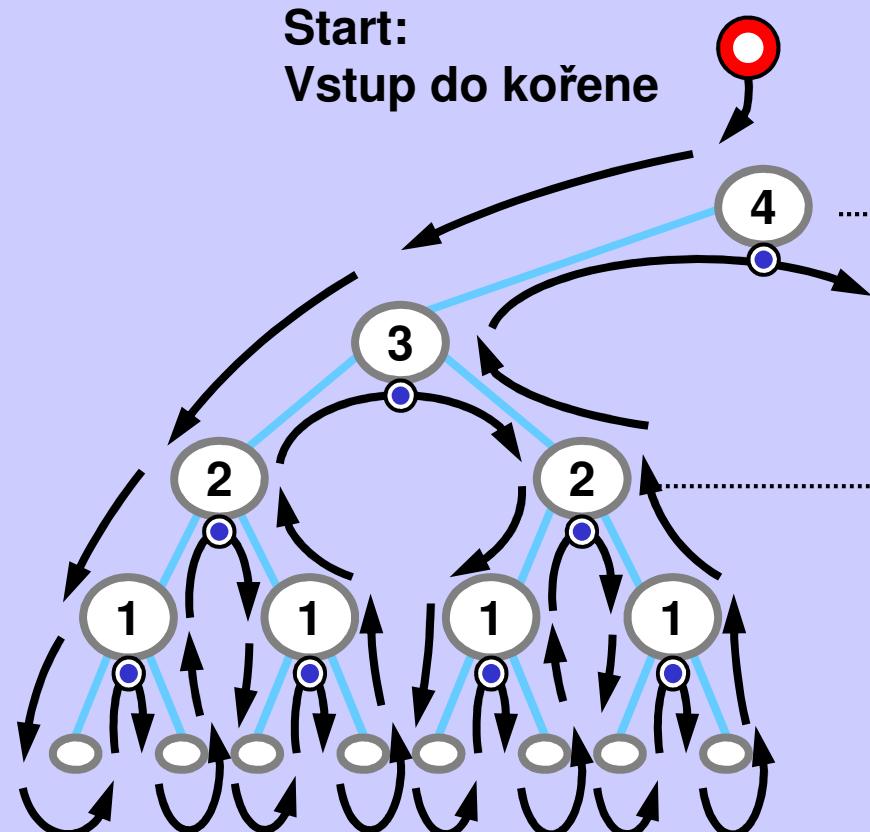
**Skonči, když je zásobník prázdný.**

## Zásobník implementuje rekurzi

Každý záběr v následující sekvenci předvádí situaci PŘED zpracováním uzlu.



Aktuální pozice

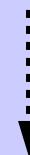


Zásobník

hodnota      návštěv

4	0
---	---

push(4,0)

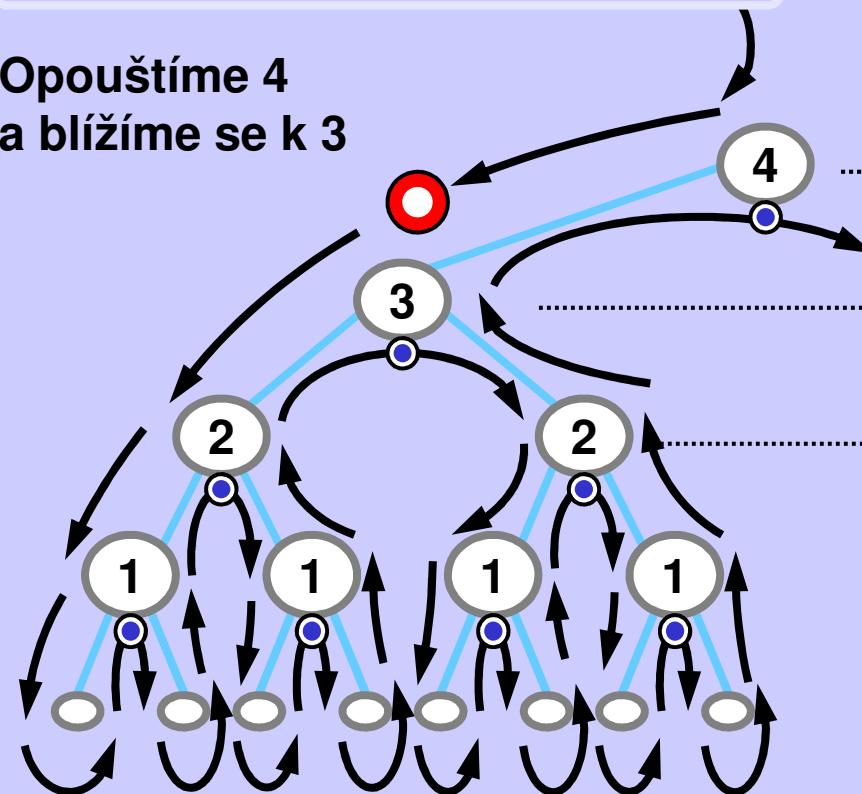


Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 4  
a blížíme se k 3



### Zásobník

hodnota	návštěv
4	1
3	0

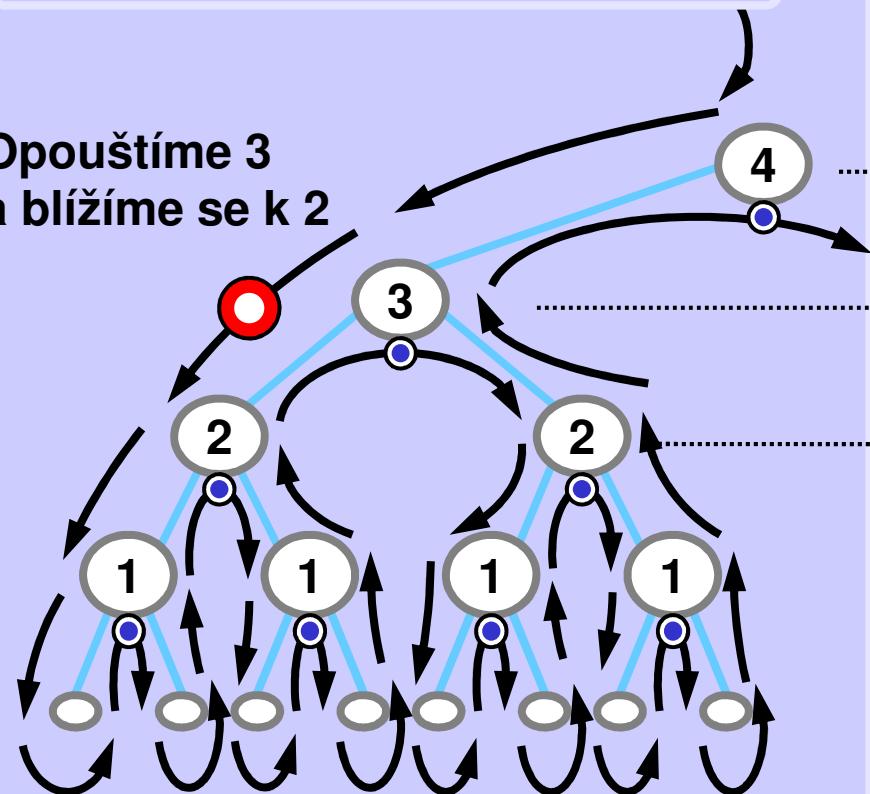
push(3,0)

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 3  
a blížíme se k 2



### Zásobník

hodnota	návštěv
4	1
3	1
2	0

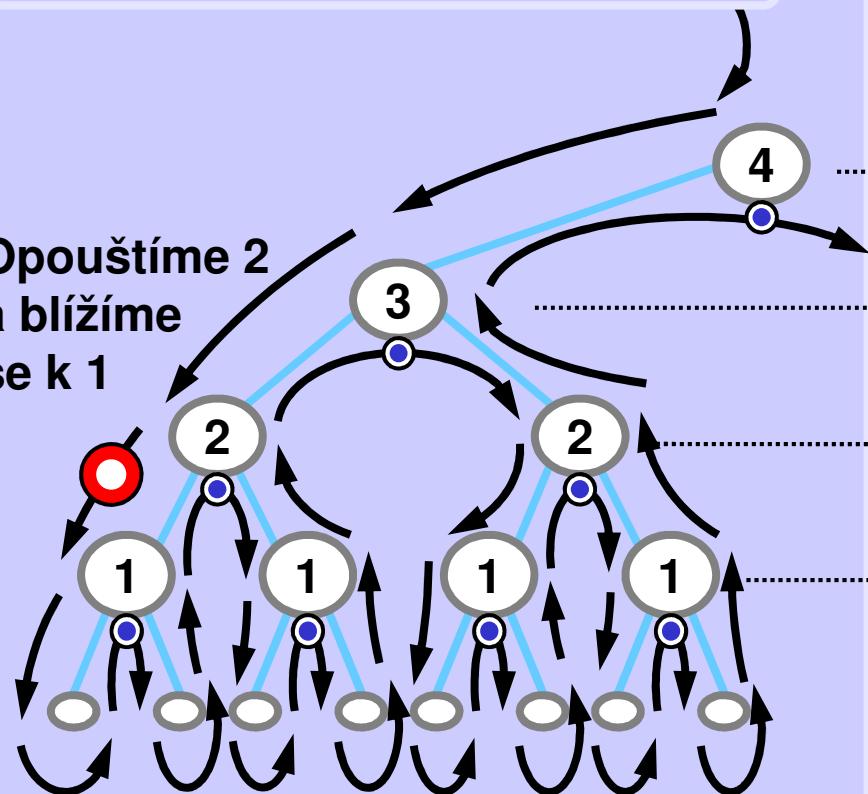
push(2,0)

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 2  
a blížíme  
se k 1



### Zásobník

hodnota	návštěv
4	1
3	1
2	1
1	0

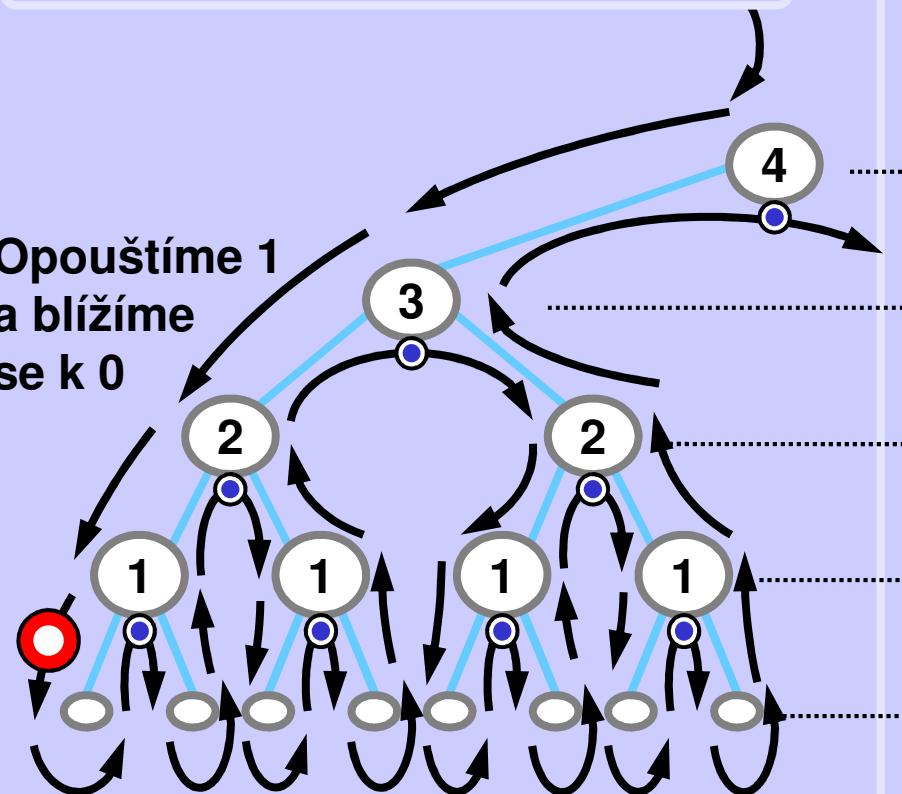
push(1,0)

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 1  
a blížíme  
se k 0



### Zásobník

hodnota	návštěv
4	1
3	1
2	1
1	1
0	0

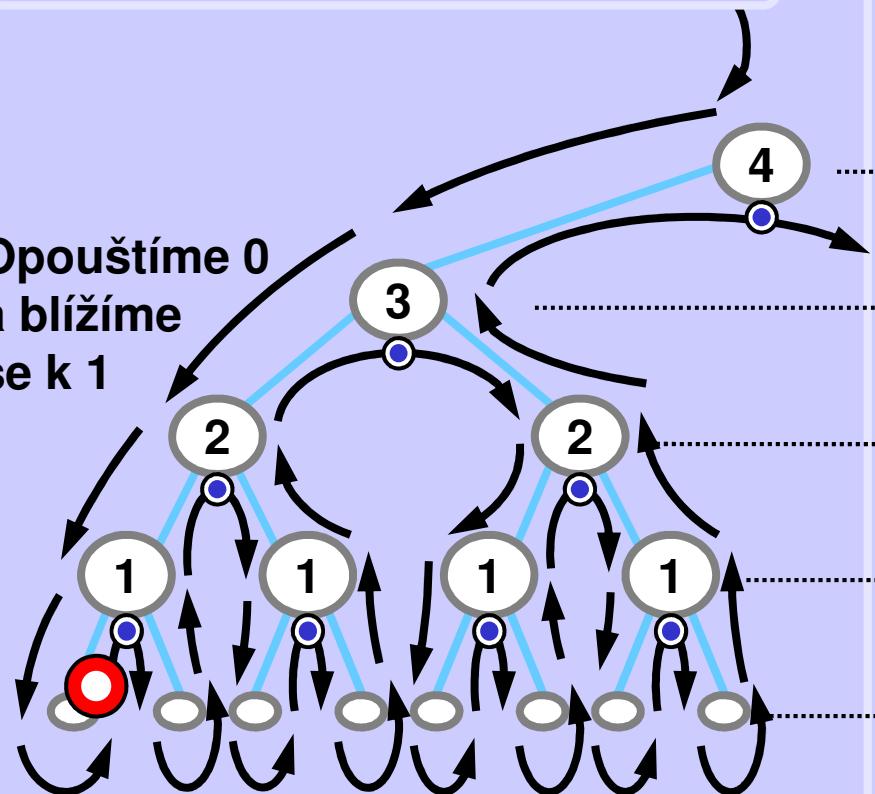
push(0,0)

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 0  
a blížíme  
se k 1



### Zásobník

hodnota	návštěv
4	1
3	1
2	1
1	1
0	0

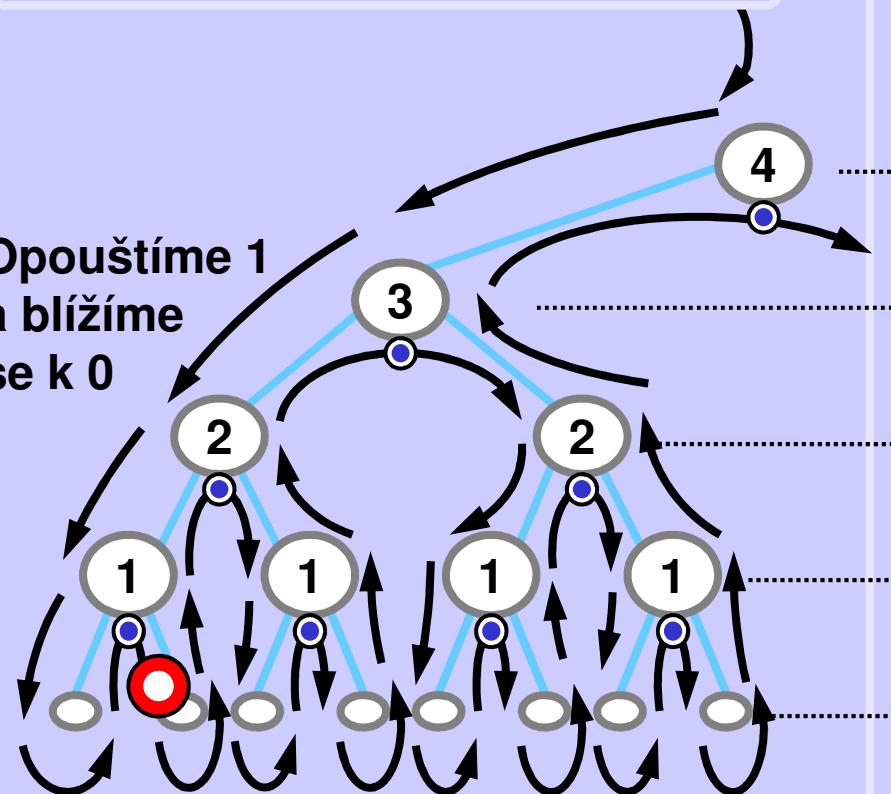
pop()

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 1  
a blížíme  
se k 0



1

### Zásobník

hodnota	návštěv
4	1
3	1
2	1
1	2
0	0

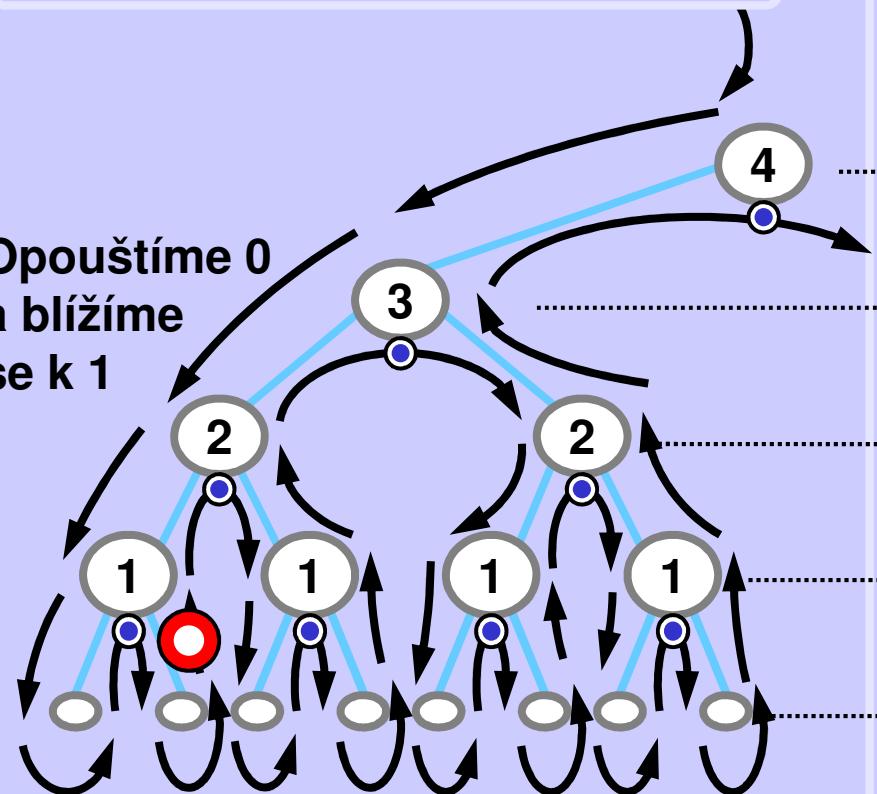
push(0,0)

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 0  
a blížíme  
se k 1



1

Výstup

### Zásobník

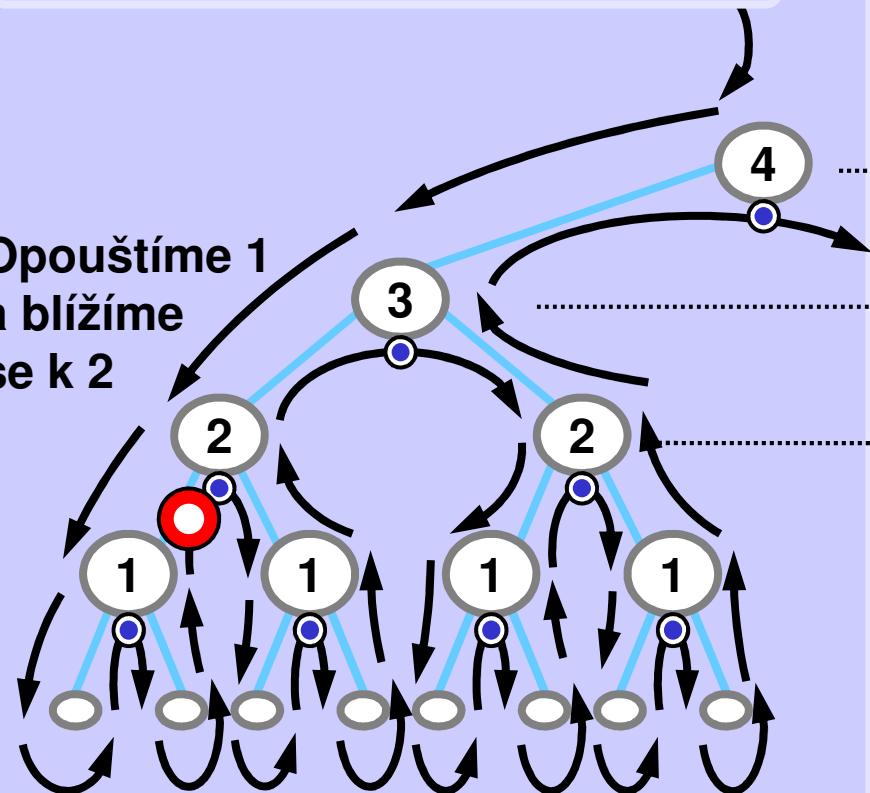
hodnota	návštěv
4	1
3	1
2	1
1	2
0	0

pop()

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 1  
a blížíme  
se k 2



1

Výstup

### Zásobník

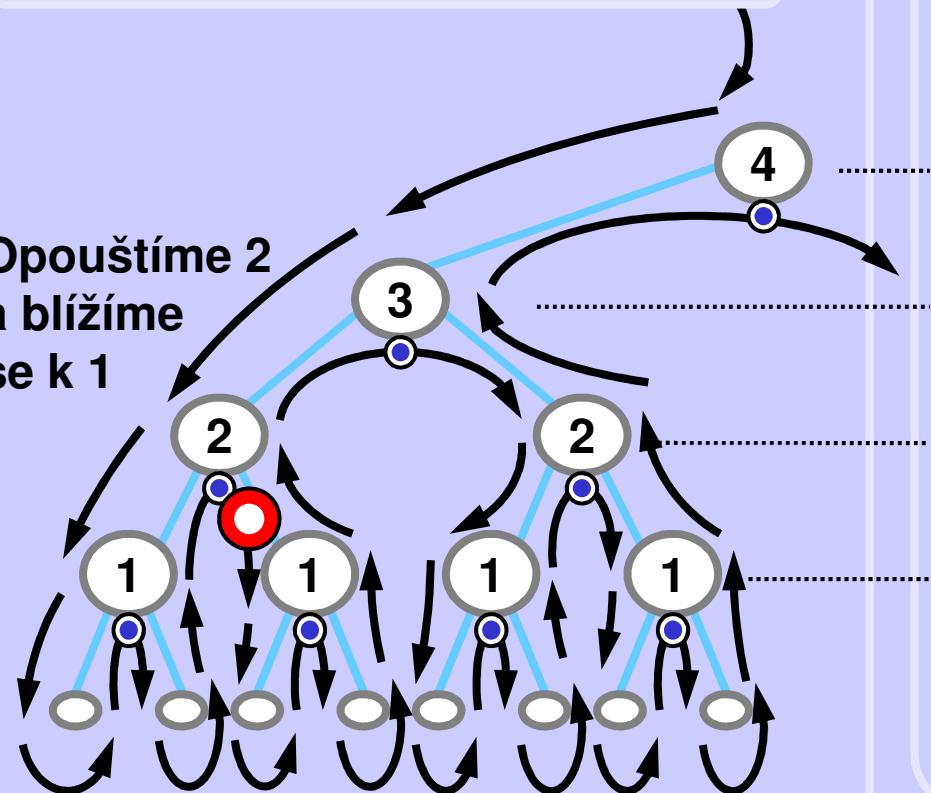
hodnota	návštěv
4	1
3	1
2	1
1	2

pop()

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 2  
a blížíme  
se k 1



### Zásobník

hodnota	návštěv
4	1
3	1
2	2
1	0

push(1,0)

1 2

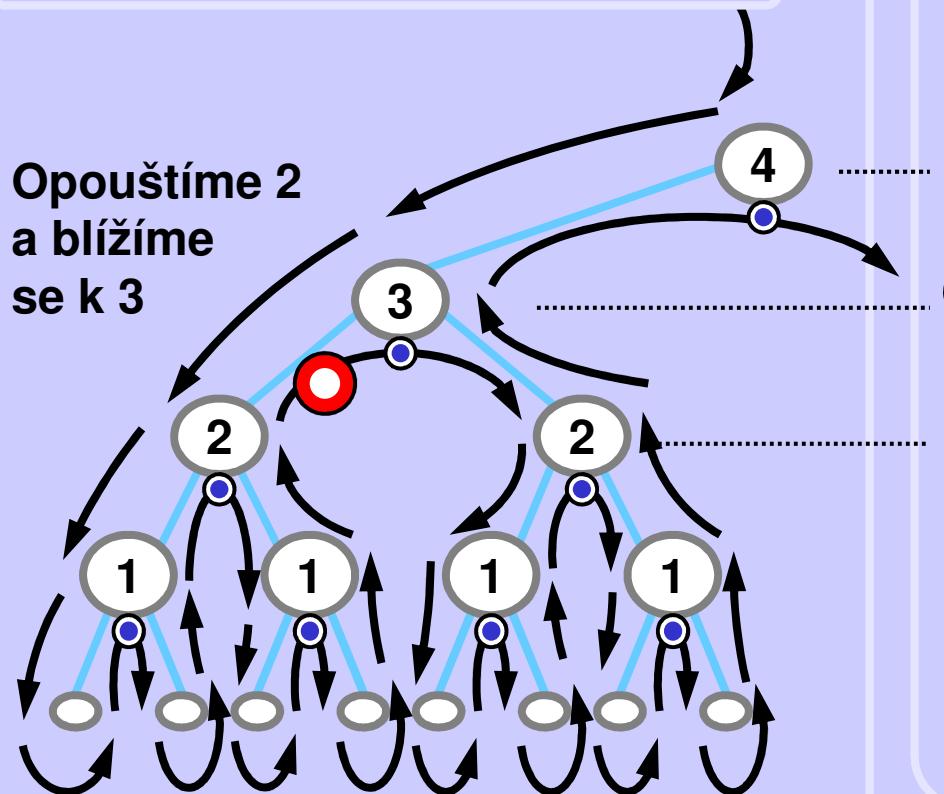
Výstup  
atd...

# Zásobník implementuje rekurzi

... po chvíli ...

## Průchod stromem rekurze

Opouštíme 2  
a blížíme  
se k 3



1 2 1

## Zásobník

hodnota	návštěv
4	1
3	1
2	2

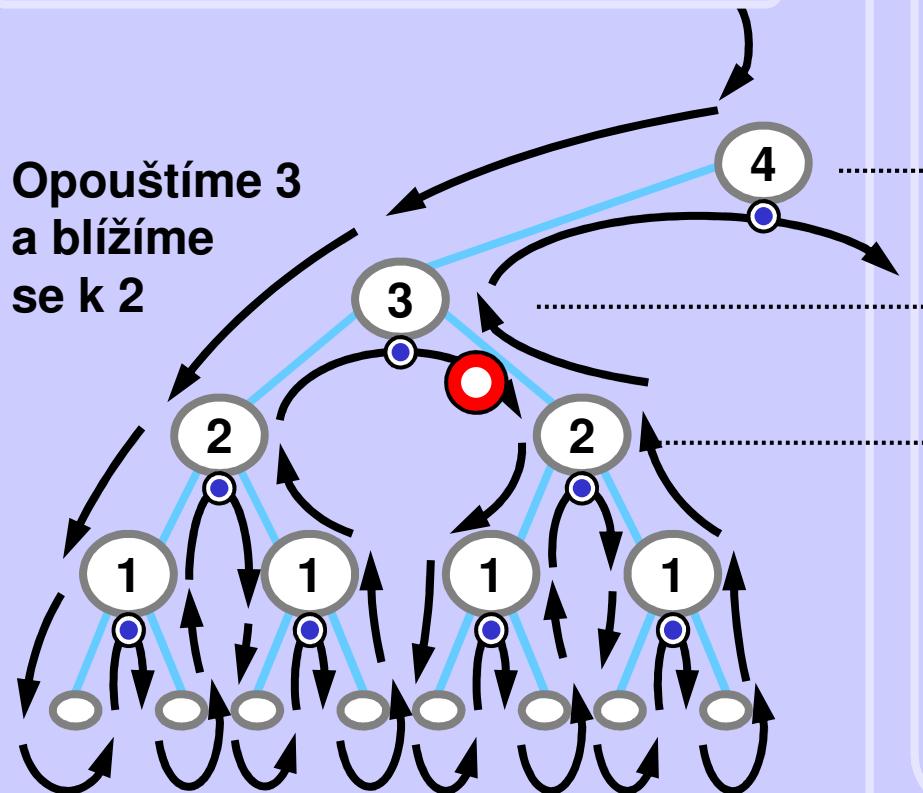
pop()

Výstup

## Zásobník implementuje rekurzi

### Průchod stromem rekurze

Opouštíme 3  
a blížíme  
se k 2



1 2 1 3

### Zásobník

hodnota	návštěv
4	1
3	2
2	0

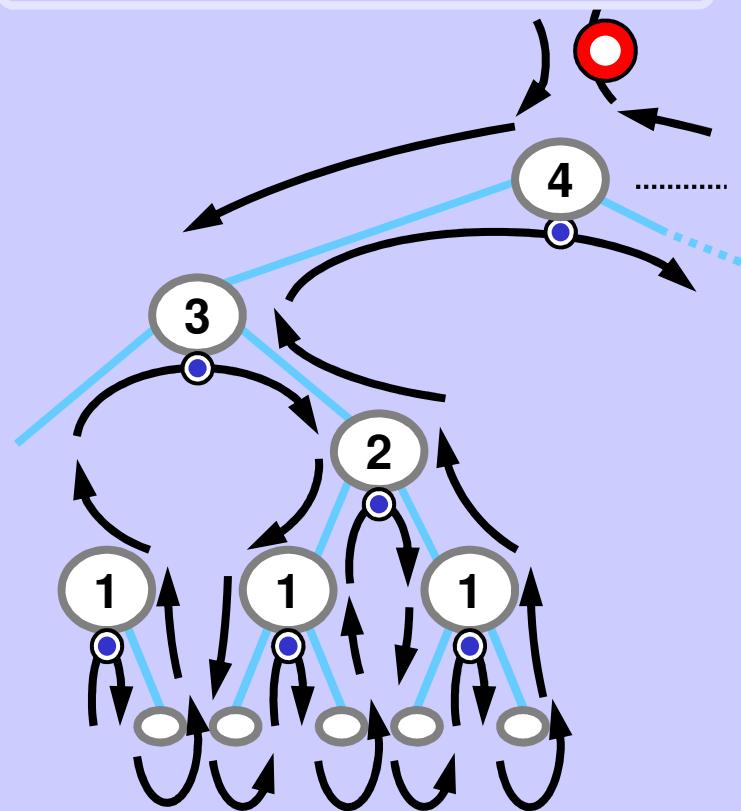
push(2,0)

Výstup  
atd...

# Zásobník implementuje rekurzi

... a je hotovo

Průchod stromem rekurze



Zásobník



(empty == true)  
pop()

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Výstup

## Zásobník implementuje rekurzi

Rekurzivní pravítko bez rekurzivního volání  
Pseudokód (skoro kód :-)) pro objektový přístup

```
stack.init();
stack.top.value = N; stack.top.visits = 0;
while (!stack.empty()) {
    if (stack.top.value == 0) stack.pop();
    if (stack.top.visits == 0) {
        stack.top.visits++;
        stack.push(stack.top.value-1, 0);
    }
    if (stack.top.visits == 1) {
        print(stack.top.value);
        stack.top.visits++;
        stack.push(stack.top.value-1, 0);
    }
    if (stack.top.visits == 2) stack.pop();
}
```

Rekurzivní pravítko  
bez rekurzivního volání,  
jednoduchá implementace polem

Zásobník implementuje rekurzi

```

int stackVal[10];  int stackVis[10];
void ruler2(int N) {
    int SP = 0;                                // stack pointer
    stackVal[SP] = N;  stackVis[SP] = 0;        // init
    while (SP >= 0) {                          // while unempty
        if (stackVal[SP] == 0) SP--;            // pop: in leaf
        if (stackVis[SP] == 0) {                // first visit
            stackVis[SP]++;
            SP++;
            stackVal[SP] = stackVal[SP-1]-1;   // go left
            stackVis[SP] = 0;
        }
        if (stackVis[SP] == 1) {                // second visit
            printf("%d ", stackVal[SP]);       // process the node
            stackVis[SP]++;
            SP++;
            stackVal[SP] = stackVal[SP-1]-1;   // go right
            stackVis[SP] = 0;
        }
        if (stackVis[SP] == 2) SP--;           // pop: node done
    }
}

```

Rekurzivní pravítko  
bez rekurzivního volání,  
jednoduchá implementace polem

Zásobník implementuje rekurzi

Poněkud kompaktnější kód

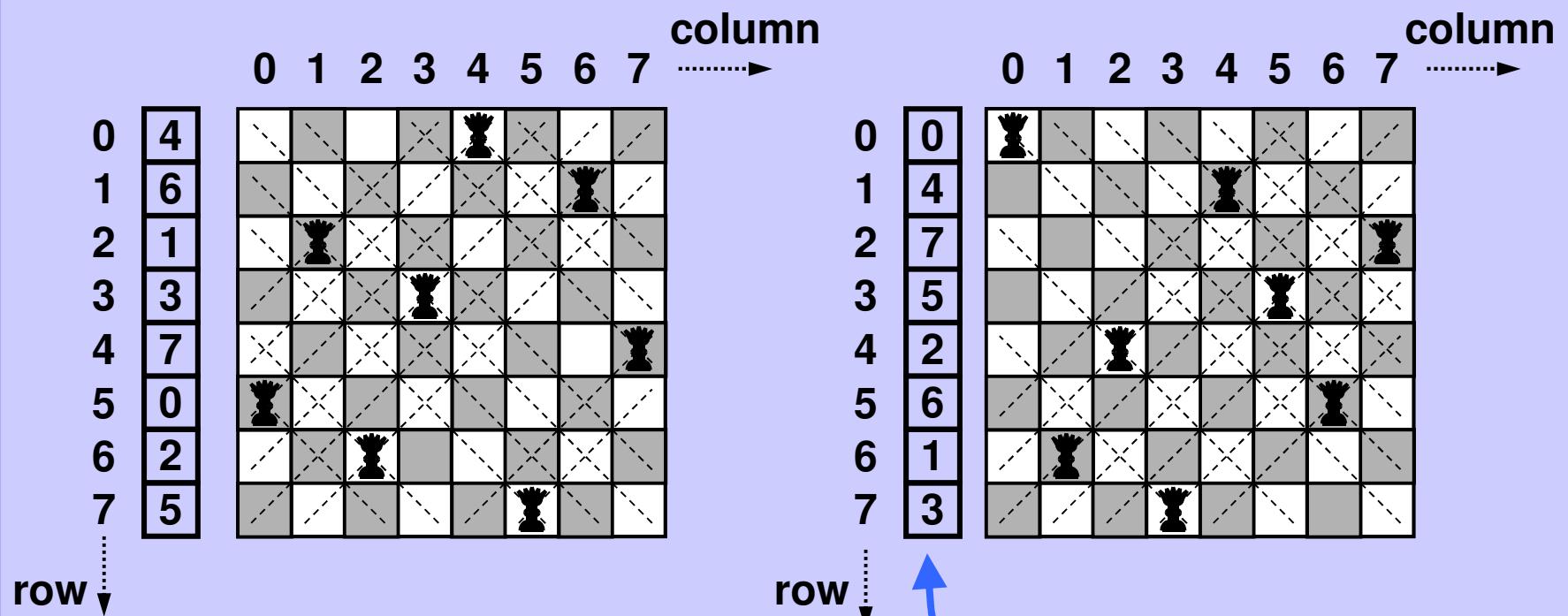
```
int stackVal[10];  int stackVis[10];

void ruler2(int N) {
    int SP = 0;                                // stack pointer
    stackVal[SP] = N;  stackVis[SP] = 0;        // init
    while (SP >= 0) {                          // while unempty
        if (stackVal[SP] == 0) SP--;            // pop: in leaf
        if (stackVis[SP] == 2) SP--;            // pop: node done
        if (stackVis[SP] == 1)
            printf("%d ", stackVal[SP]);       // process the node
        stackVis[SP]++;  SP++;                 // otherwise
        stackVal[SP] = stackVal[SP-1]-1;       // go deeper
        stackVis[SP] = 0;
    }
}
```

## Snadné prohledávání s návratem (Backtrack)

Problém osmi dam na šachovnici

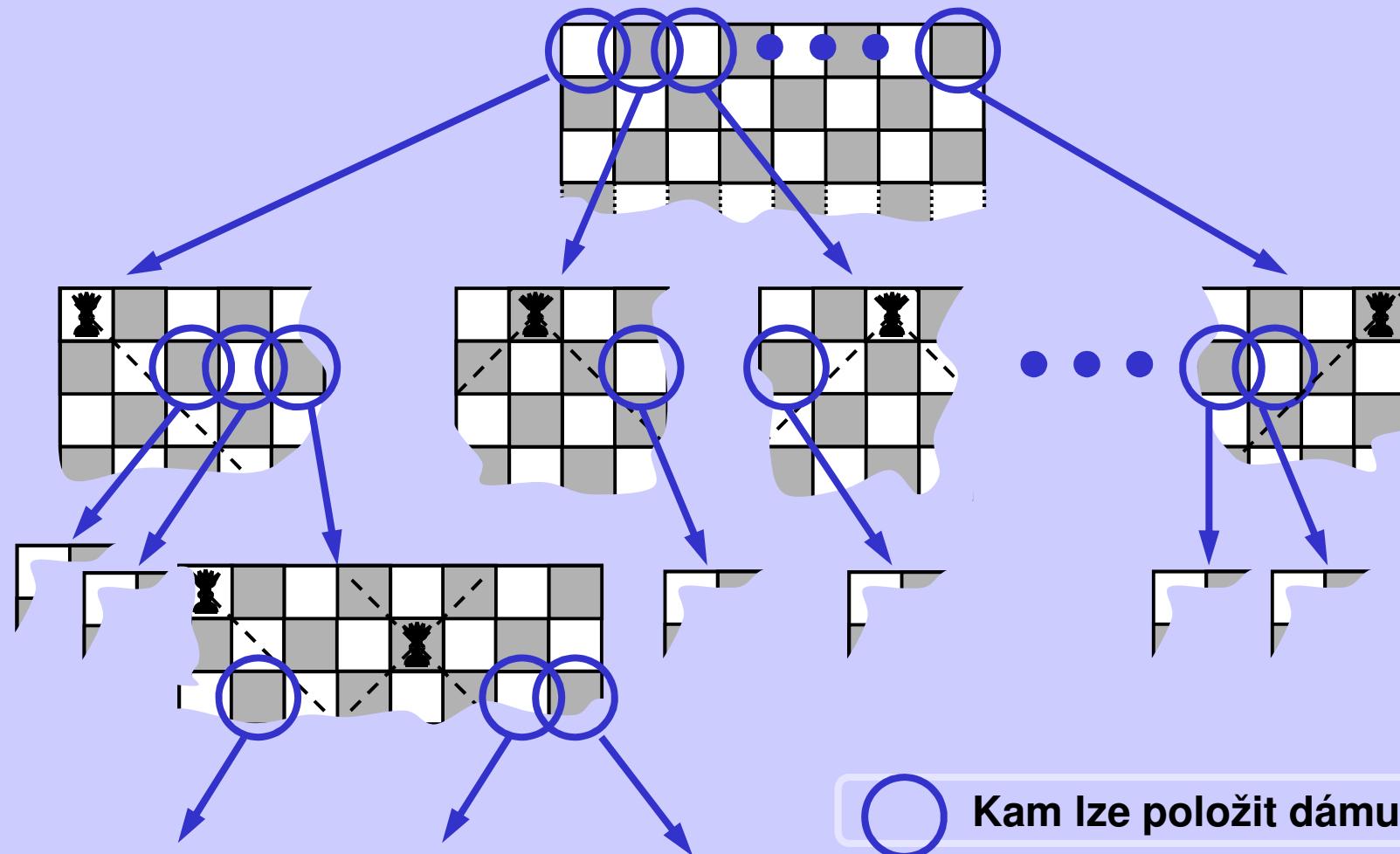
Některá řešení



Jediná datová struktura: pole `queenCol[ ]` (viz kód)

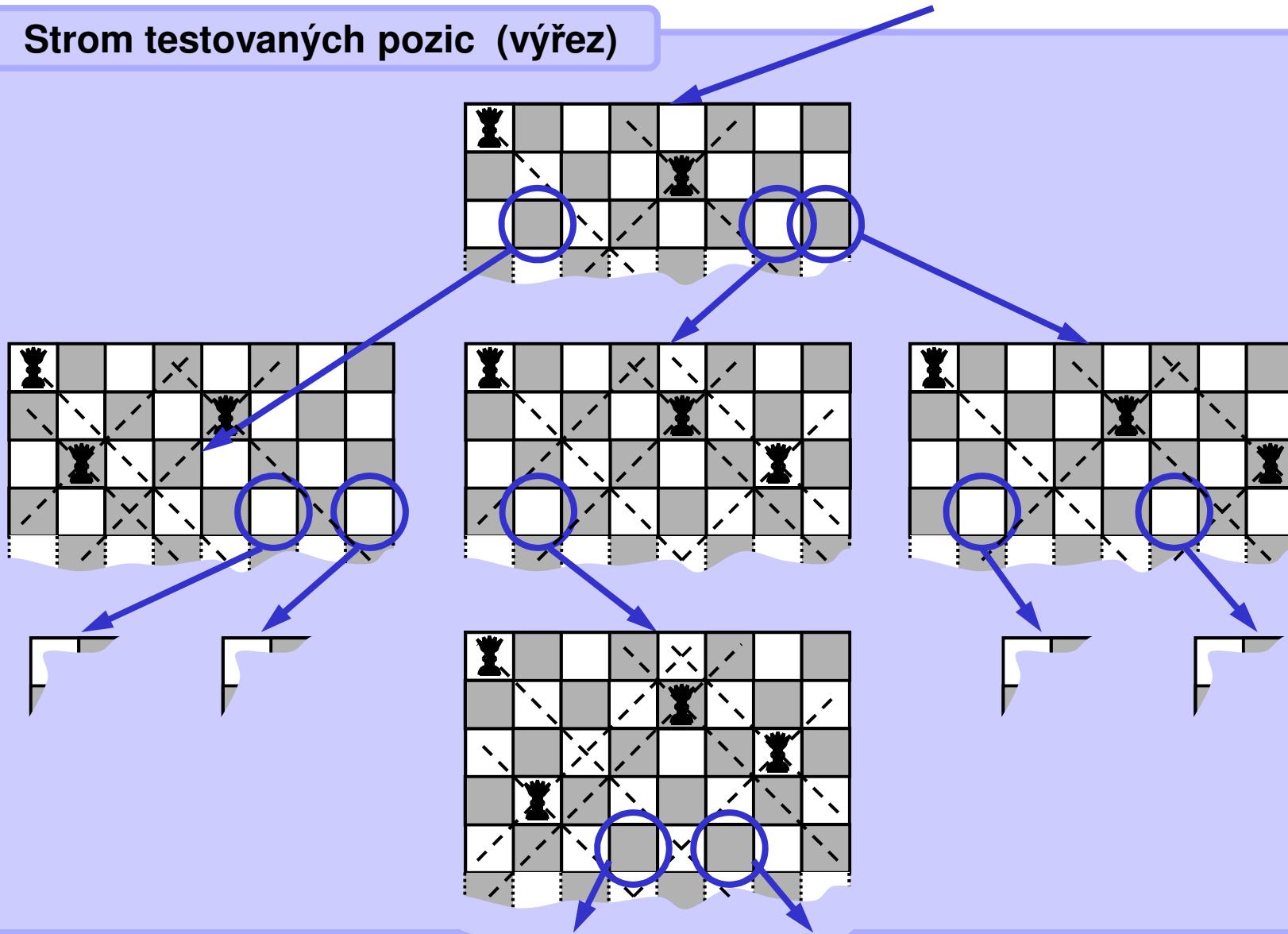
## Snadné prohledávání s návratem (Backtrack)

Strom testovaných pozic (kořen a několik potomků)



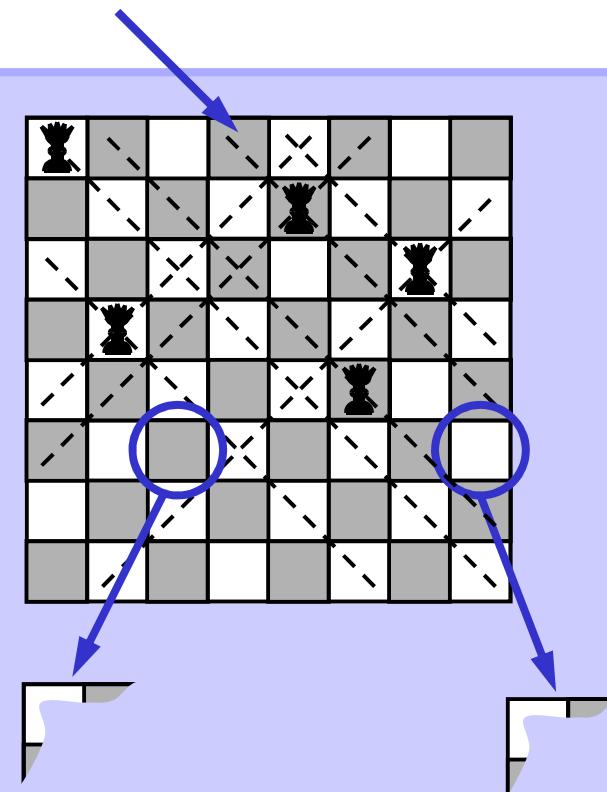
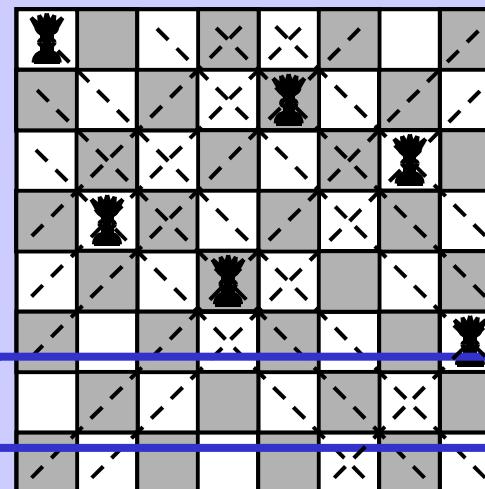
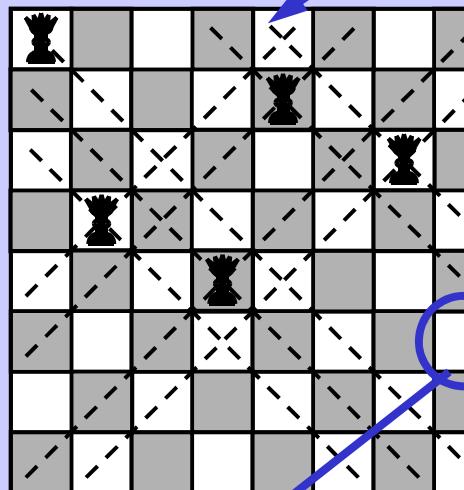
## Snadné prohledávání s návratem (Backtrack)

Strom testovaných pozic (výřez)



## Snadné prohledávání s návratem (Backtrack)

Strom testovaných pozic (výřez)



Stop!

## Snadné prohledávání s návratem (Backtrack)

### N dam na šachovnici N x N

N poč. dam	Počet řešení	Počet testovaných pozic dámy		Zrychlení
		Hrubá síla ( $N^N$ )	Backtrack	
4	2	256	240	1.07
5	10	3 125	1 100	2.84
6	4	46 656	5 364	8.70
7	40	823 543	25 088	32.83
8	92	16 777 216	125 760	133.41
9	352	387 420 489	651 402	594.75
10	724	10 000 000 000	3 481 500	2 872.33
11	2 680	285 311 670 611	19 873 766	14 356.20
12	14 200	8 916 100 448 256	121 246 416	73 537.00

Tab 3.1 Rychlosti řešení problému osmi dam

## Snadné prohledávání s návratem, 8 dam (Backtrack)

```

boolean positionOK(int r, int c) {      // r: row, c: column
    for (int i = 0; i < r; i++)
        if ((queenCol[i] == c) ||           // same column or
            (abs(r-i) == abs(queenCol[i]-c))) // same diagonal
            return false;
    return true;
}

void putQueen(int row, int col) {
    queenCol[row] = col;                  // put a queen there
    if (++row == N)                    // if solved
        print(queenCol);                // output solution
    else
        for(col = 0; col < N; col++)   // test all columns
            if (positionOK(row, col))   // if free
                putQueen(row, col);       // next row recursion
}
Call: for(int col = 0; col < 8; col++)
        putQueen(0, col);

```

# ALG 04

**Zásobník**

**Fronta**

**Operace Enqueue, Dequeue, Front, Empty....**

**Cyklická implementace fronty**

**Průchod stromem do šířky**

**Grafy**

**průchod grafem do šířky**

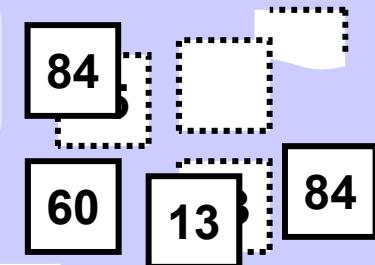
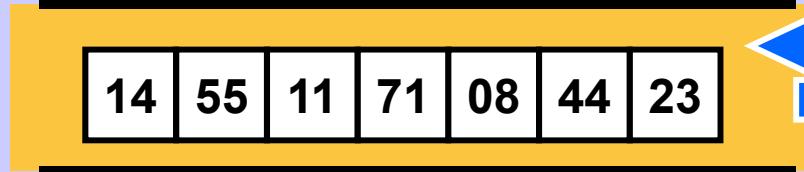
**průchod grafem do hloubky**

**Ořezávání a heuristiky**

## Zásobník / stack

Prvky se před zpracováním vkládají na vrchol zásobníku.

Vrchol / top



Prvky se odebírají z vrcholu zásobníku a pak se zpracovávají.

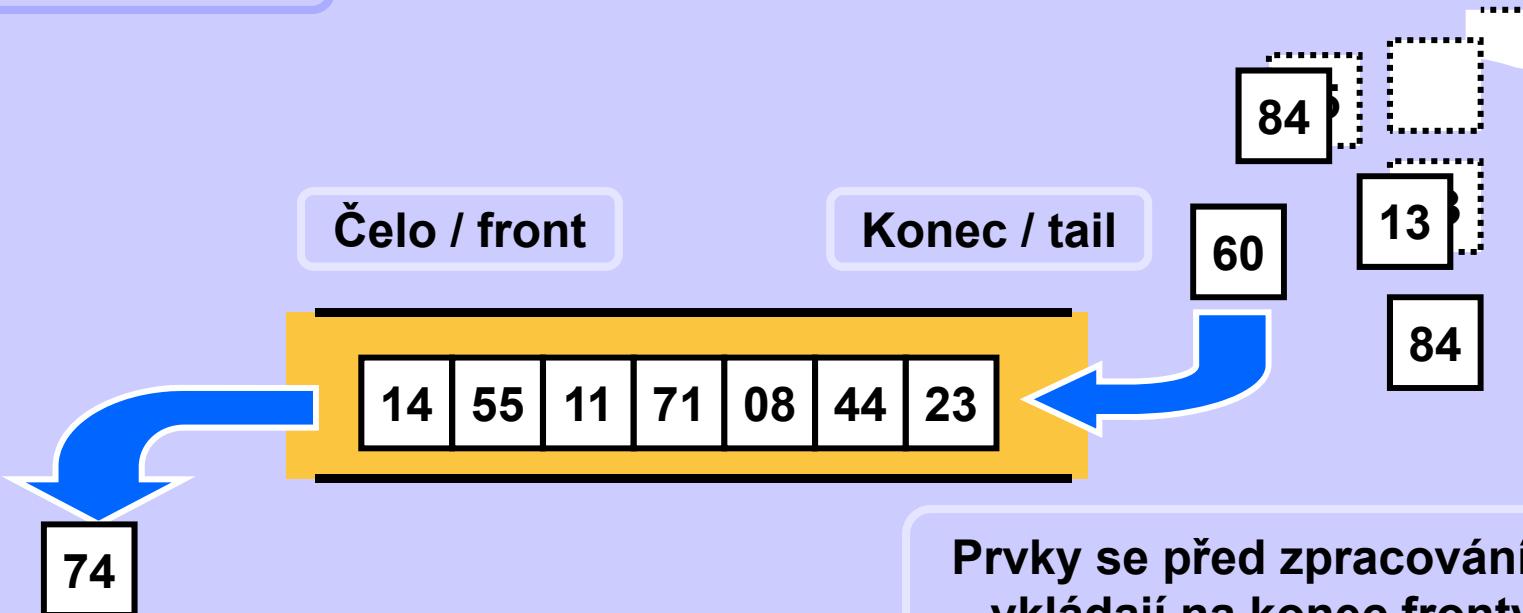
## Operace

Vlož na vrchol	Push
Odeber z vrcholu	Pop
Čti začátek	Top
Je prázdný?	Empty

## Fronta / queue

Čelo / front

Konec / tail



Prvky se před zpracováním  
vkládají na konec fronty.

Prvky se odebírají z čela  
fronty a pak se zpracovávají.

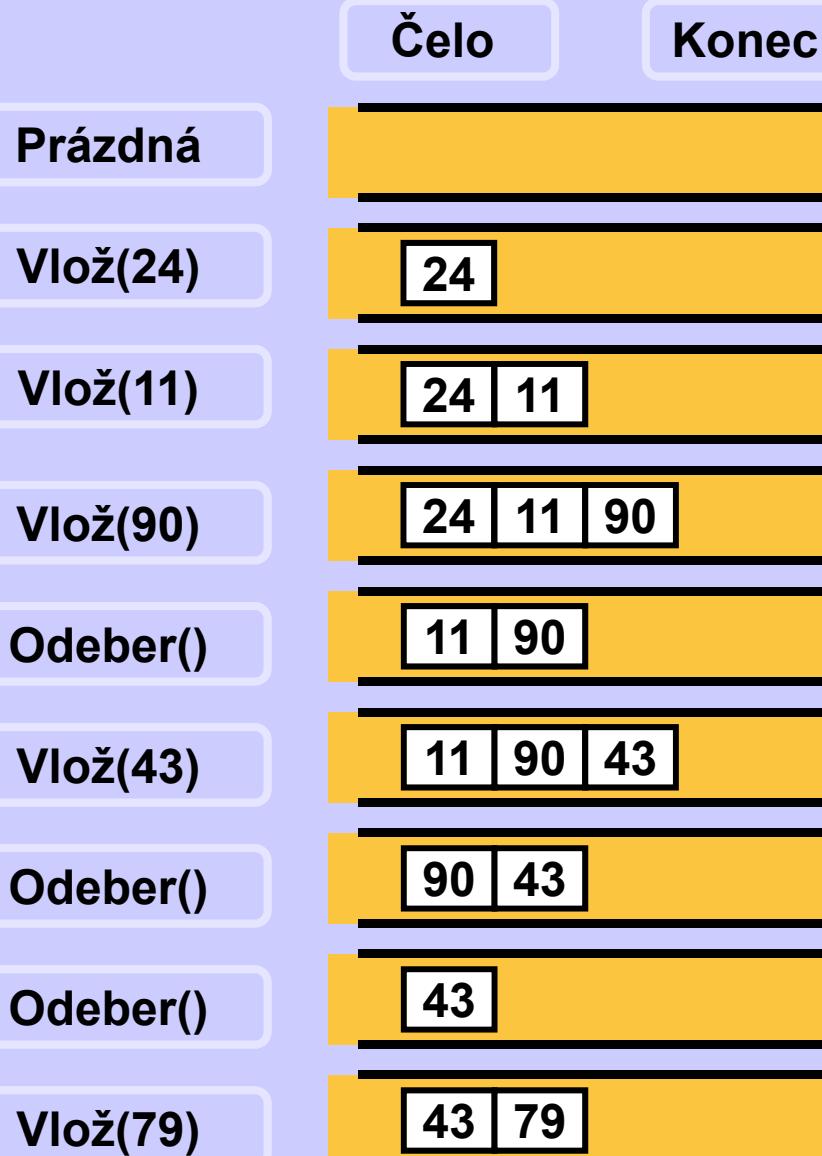
### Operace

Vlož na konec  
Odeber ze začátku  
Čti začátek  
Je prázdná?

Enqueue / InsertLast / Push ...  
Dequeue / delFront / Pop ...  
Front / Peek ...  
Empty

# Fronta

Jednoduchý  
příklad života  
fronty



## Cyklická implementace fronty polem

Prázdná fronta  
v poli pevné délky

Vlož 24, 11, 90, 43, 70.

Odeber, odeber, odeber.

Vlož 10, 20.

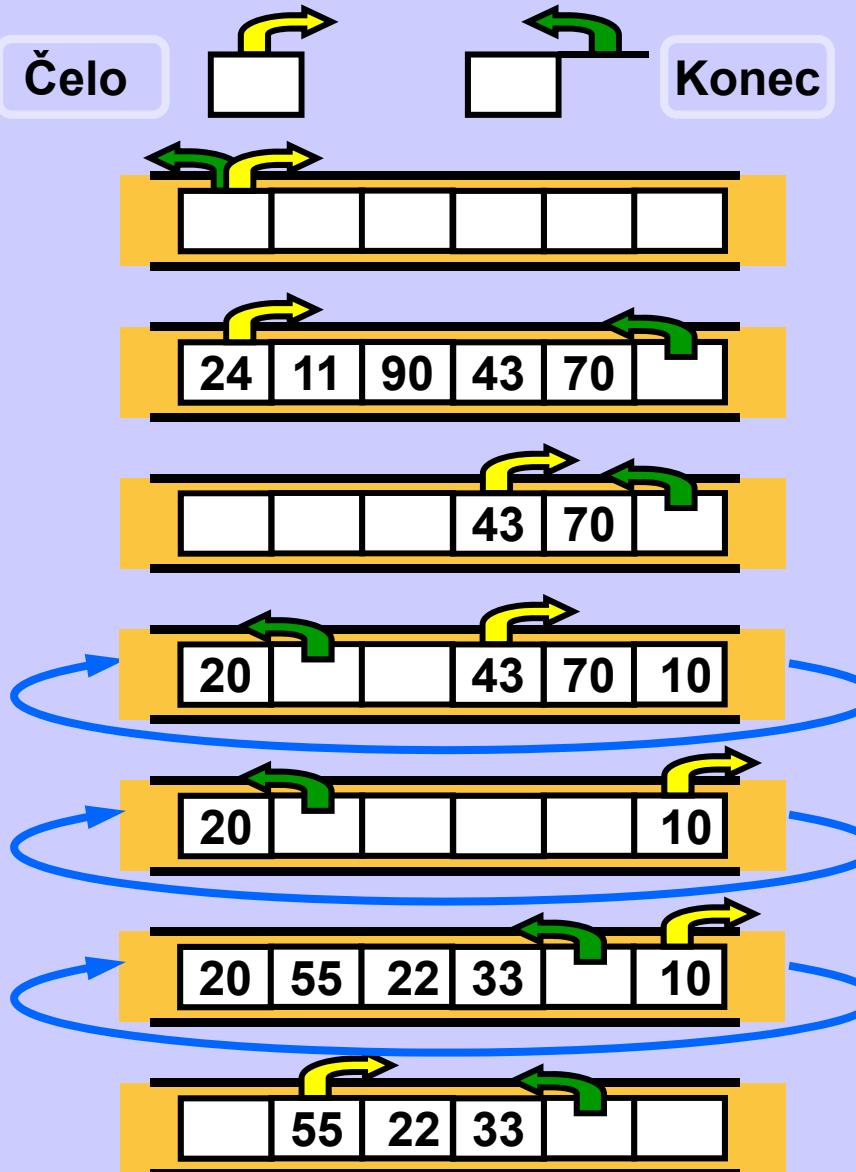
Odeber, odeber.

Vlož 55, 22, 33.

Odeber, odeber.

Čelo

Konec



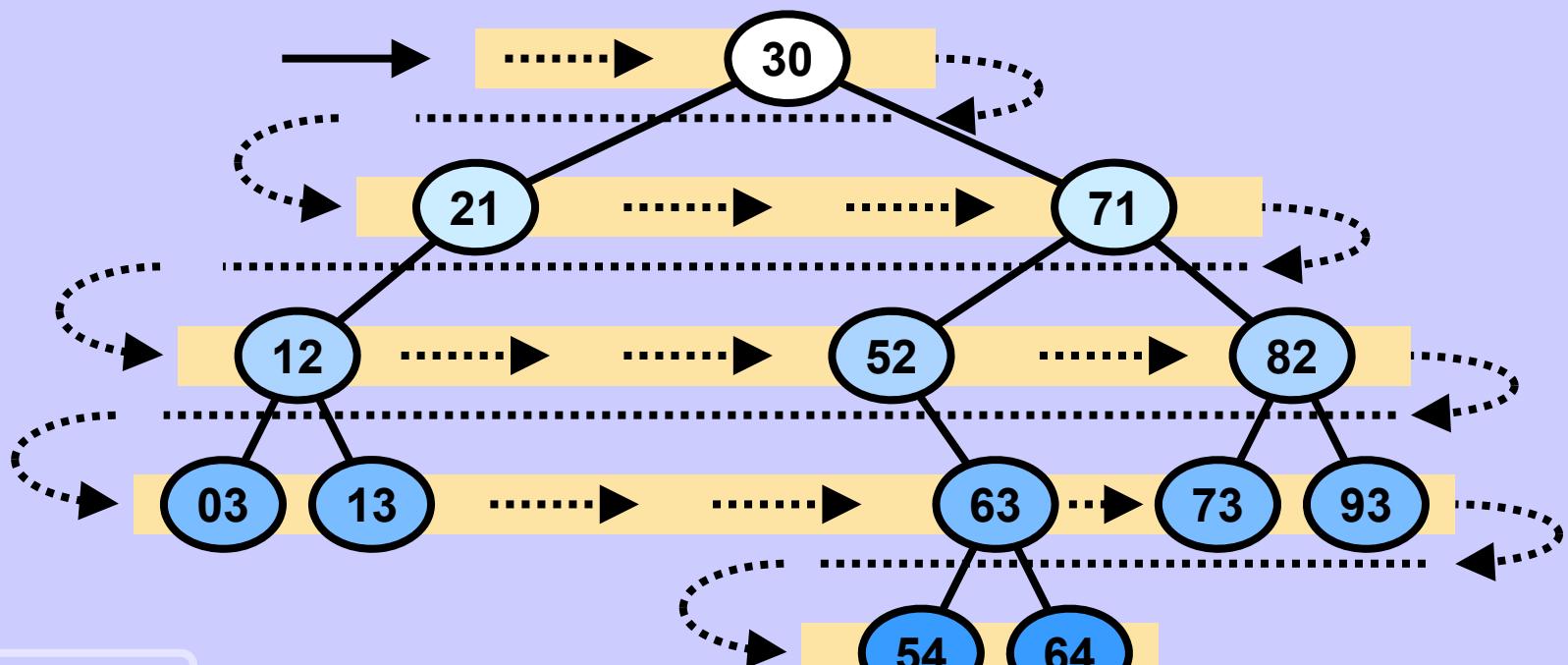
## Cyklická implementace fronty polem

Index/ukazatel konce fronty ukazuje na první volnou pozici za posledním prvkem fronty. Index/ukazatel čela fronty ukazuje na první obsazenou pozici. Pokud oba ukazují tamtéž, fronta je prázdná.

```
class Queue {  
    Node q [];  
    int size;  
    int front;  
    int tail;  
  
    Queue( int qsize ){  
        size = qsize;  
        q = new Node[size];  
        front = 0;  
        tail = 0;  
    }  
  
    boolean Empty() {  
        return ( tail==front );  
    }  
  
    void Enqueue( Node node ){  
        if( (tail+1 == front) ||  
            (tail-front == size-1) )  
            ... // queue full, fix it  
  
        q[t++] = node;  
        if( tail==size ) tail = 0;  
    }  
  
    Node Dequeue() {  
        Node n = q[front++];  
        if( front == size ) front = 0;  
        return n;  
    }  
} // end of Queue
```

## Průchod stromem do šířky

Strom s naznačeným směrem průchodu



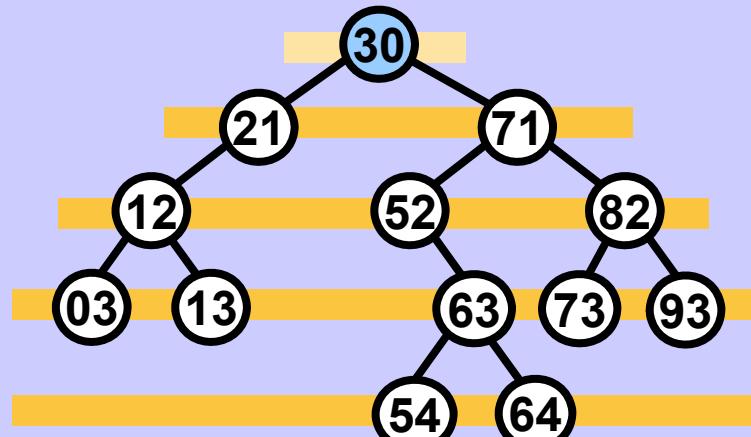
Pořadí uzelů

30 21 71 12 52 82 03 13 63 73 93 54 64

Struktura stromu ani rekurzivní přístup tento průchod nepodporují.

# Průchod stromem do šířky

## Inicializace



Výstup

2.

Vytvoř prázdnou frontu

Do fronty vlož kořen stromu

30

2.

Čelo

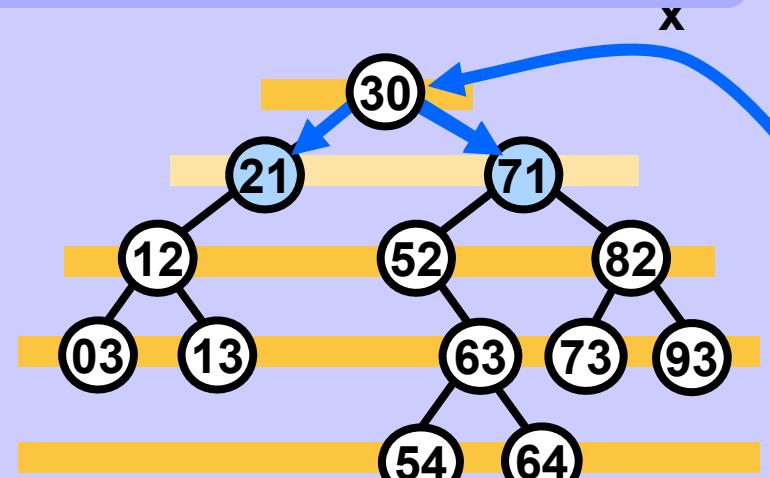
Konec

## Hlavní cyklus

Dokud není fronta prázdná, opakuj:

1. Odeber první uzel z fronty a zpracuj ho.
2. Do fronty vlož jeho potomky, pokud existují.

## Průchod stromem do šířky



Výstup

30

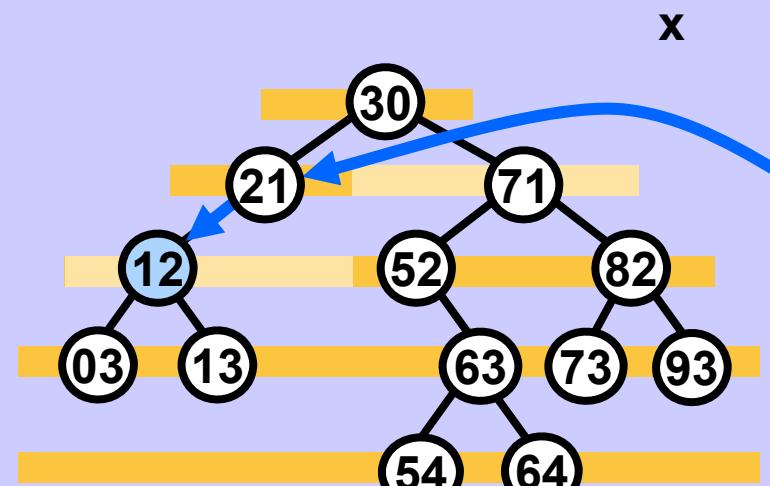
1.  $x = \text{Odeber}(), \text{tisk}(x.\text{key})$ .



2.  $\text{Vlož}(x.\text{left}), \text{vlož}(x.\text{right})$ . \*



\*) pokud existuje



Výstup

30 21

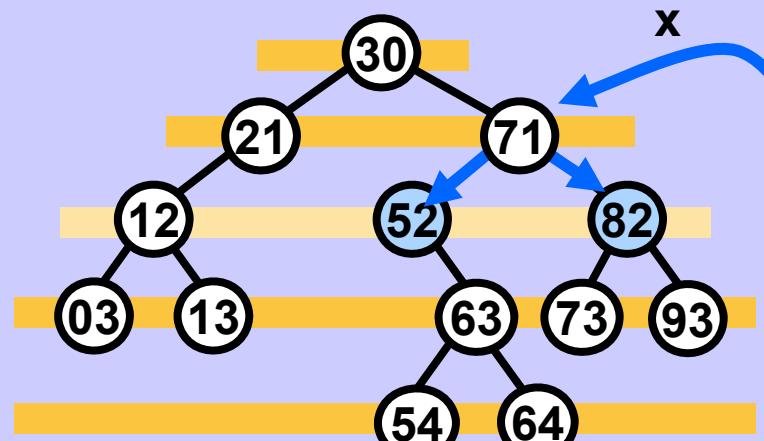
1.  $x = \text{Odeber}(), \text{tisk}(x.\text{key})$ .



2.  $\text{Vlož}(x.\text{left}), \text{vlož}(x.\text{right})$ . \*



## Průchod stromem do šířky



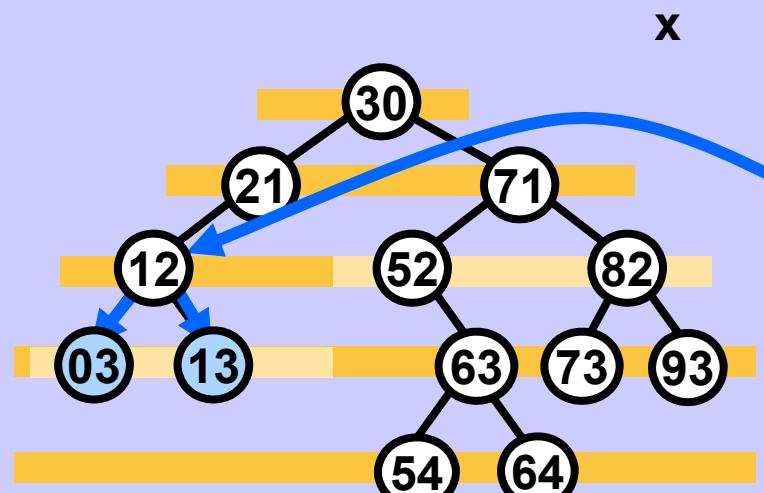
Výstup

30 21 71

1.  $x = \text{Odeber}()$ , tisk ( $x.\text{key}$ ).

2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*

\*) pokud existuje



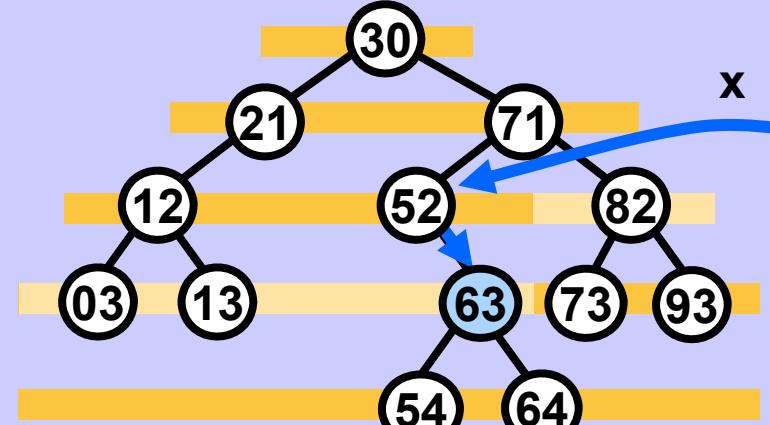
Výstup

30 21 71 12

1.  $x = \text{Odeber}()$ , tisk( $x.\text{key}$ ).

2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*

## Průchod stromem do šířky



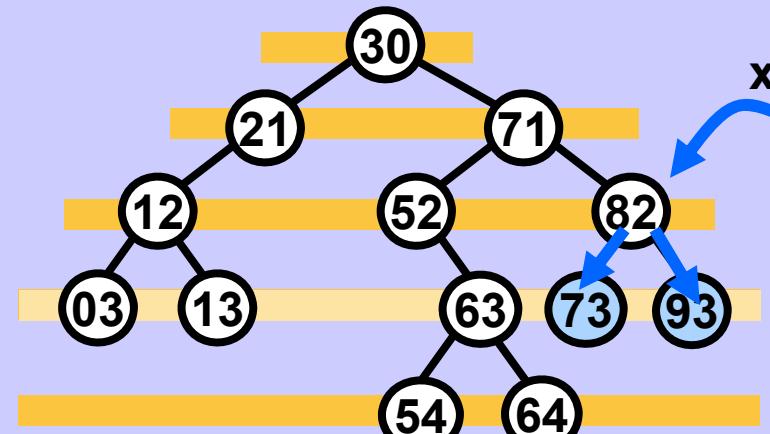
Výstup

30 21 71 12 52

1.  $x = \text{Odeber}()$ , tisk ( $x.key$ ).



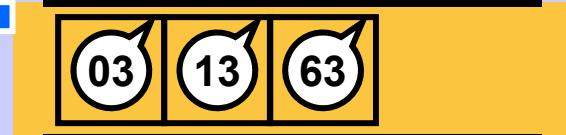
2. Vlož( $x.left$ ), vlož( $x.right$ ). \*



Výstup

30 21 71 12 52 82

1.  $x = \text{Odeber}()$ , tisk( $x.key$ ).

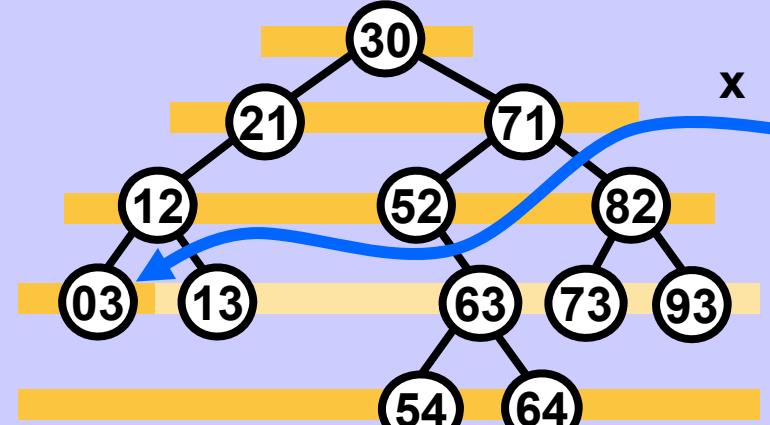


2. Vlož( $x.left$ ), vlož( $x.right$ ). \*



\*) pokud existuje

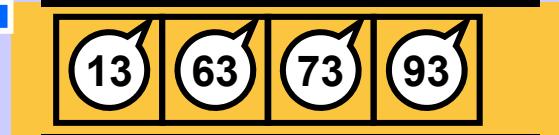
## Průchod stromem do šířky



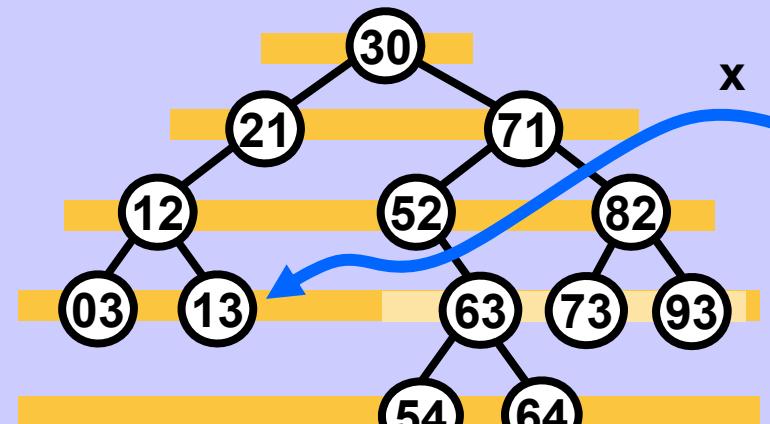
Výstup

30 21 71 12 52 82 03

1.  $x = \text{Odeber}()$ , tisk ( $x.\text{key}$ ).



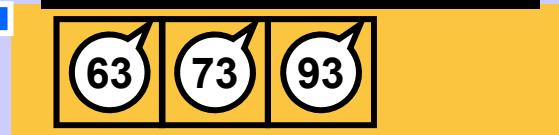
2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*



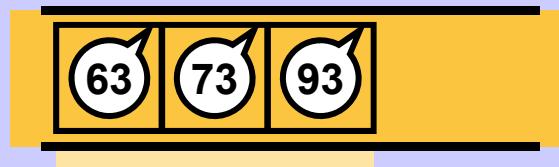
Výstup

30 21 71 12 52 82 03 13

1.  $x = \text{Odeber}()$ , tisk( $x.\text{key}$ ).

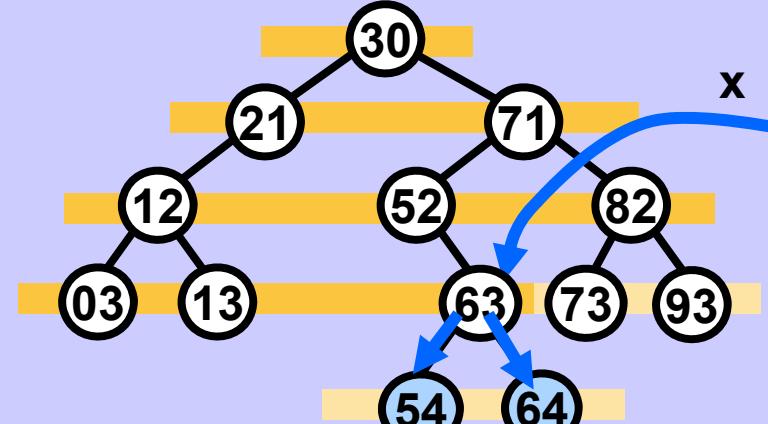


2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*



\*) pokud existuje

## Průchod stromem do šířky



Výstup

30 21 71 12 52 82 03 13 63

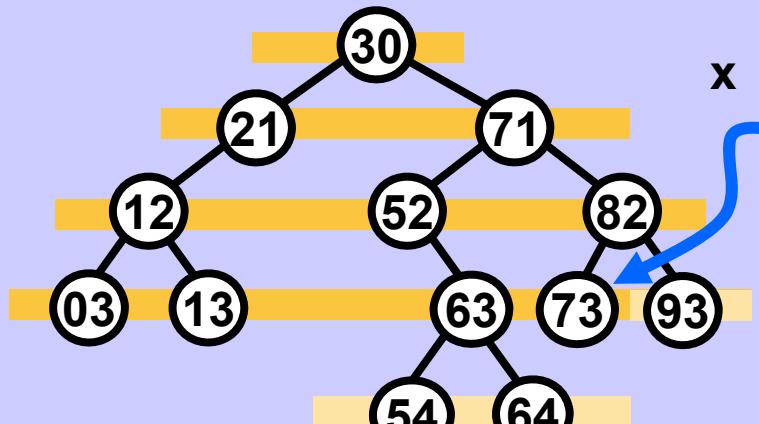
1.  $x = \text{Odeber}()$ , tisk ( $x.key$ ).



2.  $\text{Vlož}(x.left)$ ,  $\text{vlož}(x.right)$ . \*



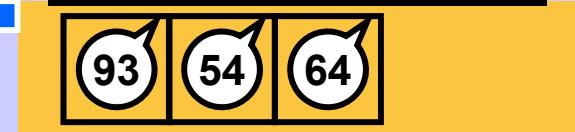
\*) pokud existuje



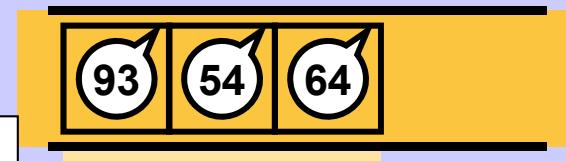
Výstup

30 21 71 12 52 82 03 13 63 73

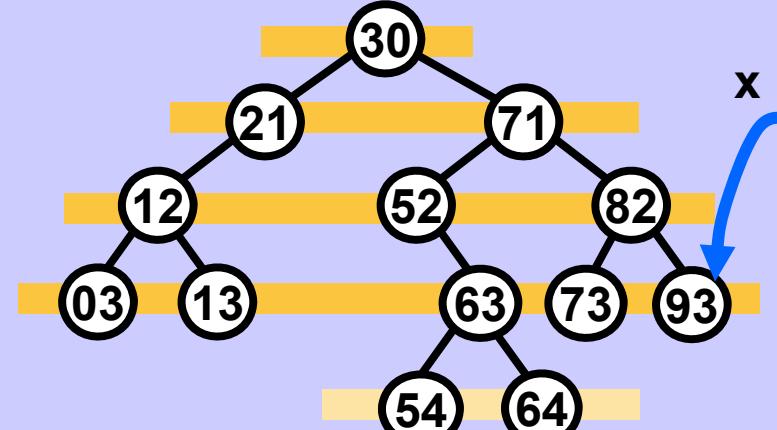
1.  $x = \text{Odeber}()$ , tisk( $x.key$ ).



2.  $\text{Vlož}(x.left)$ ,  $\text{vlož}(x.right)$ . \*



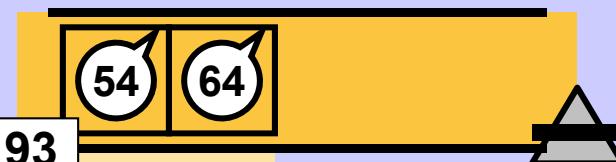
## Průchod stromem do šířky



1.  $x = \text{Odeber}()$ , tisk ( $x.\text{key}$ ).



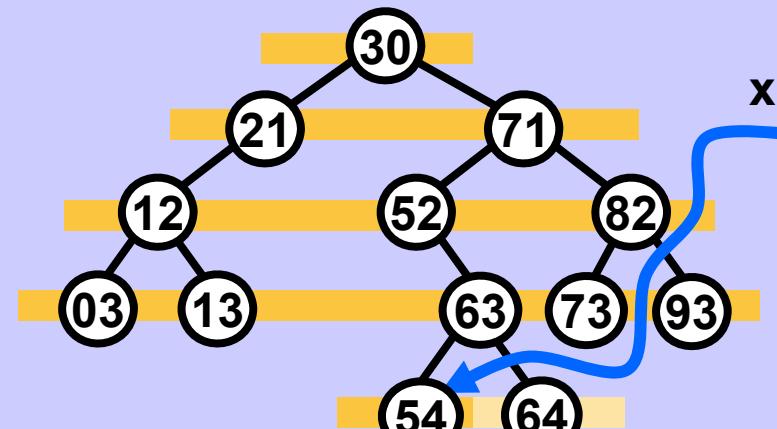
2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*



Výstup

30 21 71 12 52 82 03 13 63 73 93

\*) pokud existuje



1.  $x = \text{Odeber}()$ , tisk( $x.\text{key}$ ).



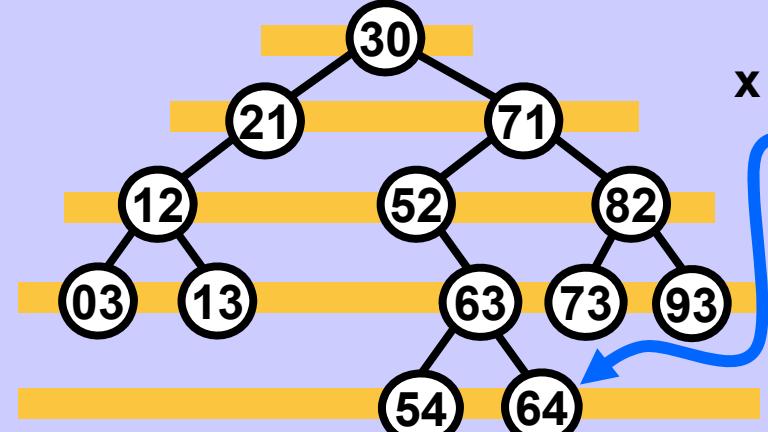
2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*



Výstup

30 21 71 12 52 82 03 13 63 73 93 54

## Průchod stromem do šířky



Výstup

30 21 71 12 52 82 03 13 63 73 93 54 64

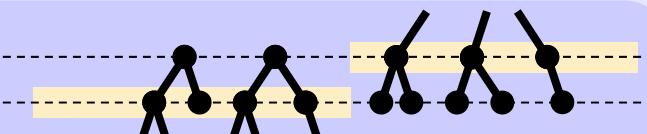
1.  $x = \text{Odeber}()$ , tisk ( $x.\text{key}$ ).

2.  $\text{Vlož}(x.\text{left})$ ,  $\text{vlož}(x.\text{right})$ . \*

\*) pokud existuje

Fronta je prázdná,  
průchod stromem končí.

V neprázdné frontě jsou vždy právě  
-- některé (třeba všechny) uzly jednoho patra  
-- a všichni potomci těch uzel tohoto patra, které už nejsou ve frontě.



Někdy jsou ve frontě přesně všechny uzly jednoho patra. Viz výše.

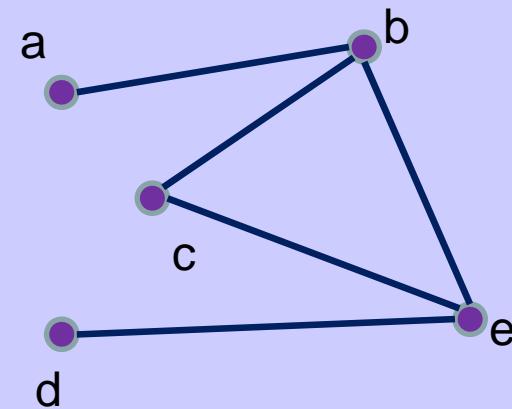


## Průchod stromem do šířky

```
void traverseTreeBreadthFirst( Node node ) {
    if( node == null ) return;
    Queue q = new Queue();           // init
    q.Enqueue( node );              // root into queue
    while( !q.Empty() ) {
        node = q.Dequeue();
        print( node.key );          // process node
        if( node.left != null ) q.Enqueue( node.left );
        if( node.right != null ) q.Enqueue( node.right );
    }
}
```

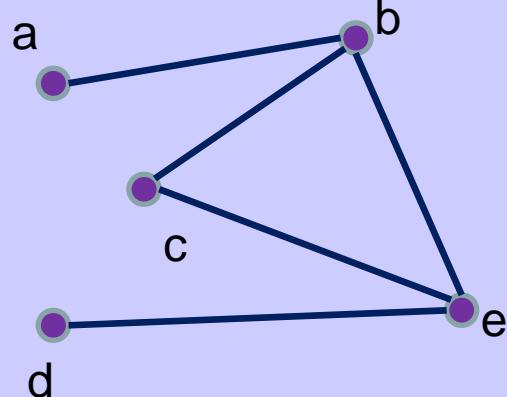
## Grafy

- graf je uspořádaná dvojice
  - množiny vrcholů  $\mathcal{V}$  a
  - množiny hran  $\mathcal{E}$
- $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- příklad:
  - $\mathcal{V} = \{a, b, c, d, e\}$
  - $\mathcal{E} = \{\{a,b\}, \{b,e\}, \{b,c\}, \{c,e\}, \{e,d\}\}$

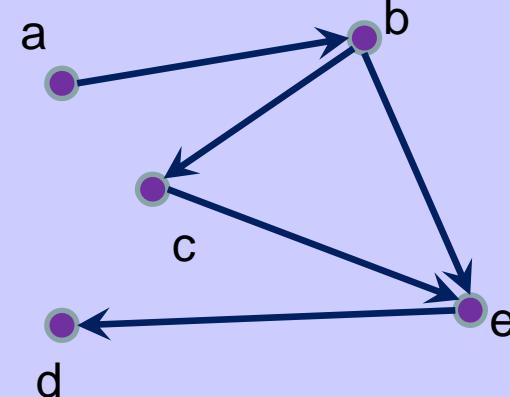


## Grafy - orientovanost

- **neorientovaný graf**
  - hrana je neuspořádaná dvojice vrcholů
- $E = \{\{a,b\}, \{b,e\}, \{b,c\}, \{c,e\}, \{e,d\}\}$



- **orientovaný graf**
  - hrana je uspořádaná dvojice vrcholů
- $E = \{\{a,b\}, \{b,e\}, \{b,c\}, \{c,e\}, \{e,d\}\}$



## Grafy – matice sousednosti

- Nechť  $G = (\mathcal{V}, \mathcal{E})$  je graf s  $n$  vrcholy
- Označme vrcholy  $v_1, \dots, v_n$  (v nějakém libovolném pořadí)
- Matice sousednosti grafu G je čtvercová matice

$$A_G = (a_{i,j})_{i,j=1}^n$$

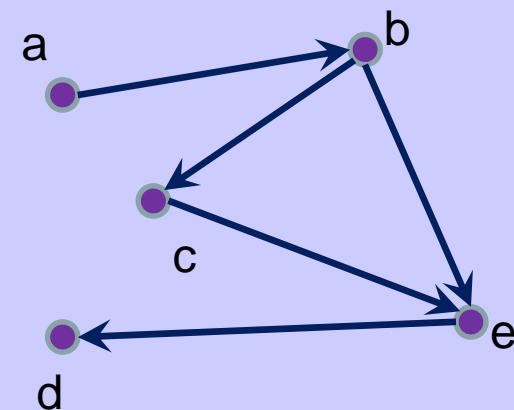
definovaná předpisem

$$a_{i,j} = \begin{cases} 1 & \text{pro } \{v_i, v_j\} \in E \\ 0 & \text{jinak} \end{cases}$$

## Grafy – matice sousednosti

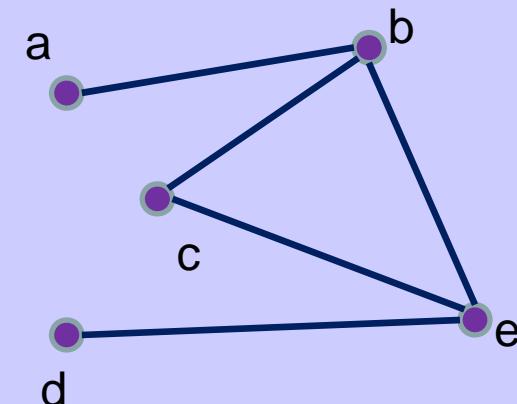
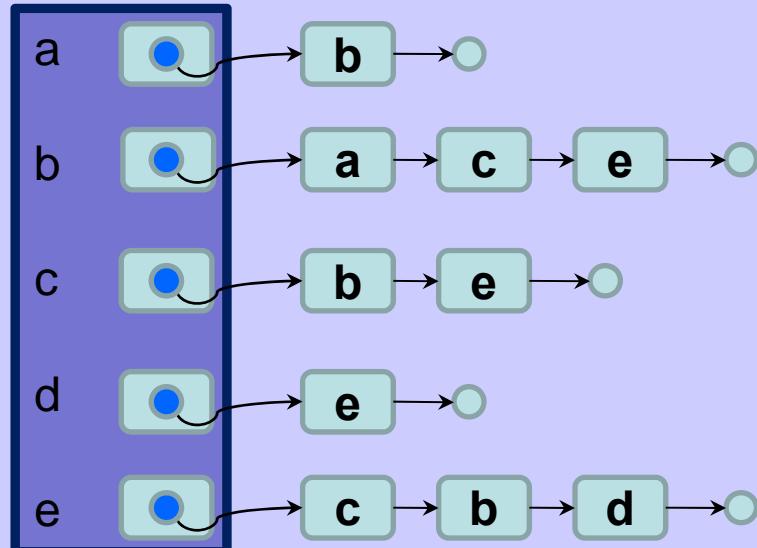
- pro orientovaný graf

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	0	1
c	0	0	0	0	1
d	0	0	0	0	0
e	0	0	0	1	0

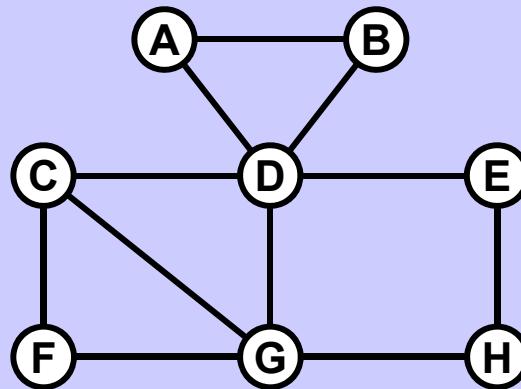


## Grafy – seznam sousedů

- Nechť  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  je (ne)orientovaný graf s  $n$  vrcholy
- Označme vrcholy  $v_1, \dots, v_n$  (v nějakém libovolném pořadí)
- Seznam sousedů grafu  $\mathcal{G}$  je pole  $\mathcal{P}$  ukazatelů velikosti  $n$ 
  - kde  $\mathcal{P}[i]$  ukazuje na spojový seznam vrcholů, se kterými je vrchol  $v_i$  spojen hranou

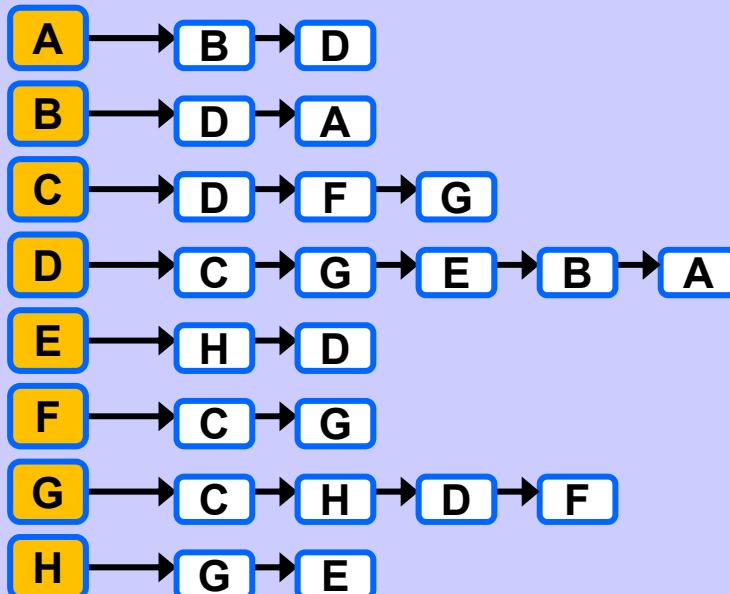


## Průchod grafem do hloubky



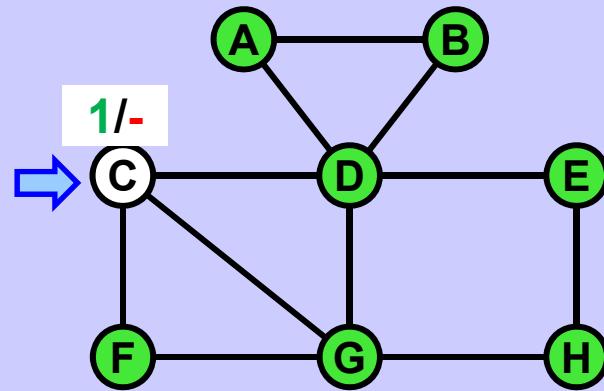
Matice sousednosti

## Spojová reprezentace



	A	B	C	D	E	F	G	H
A	0	1	0	1	0	0	0	0
B	1	0	0	1	0	0	0	0
C	0	0	0	1	0	1	1	0
D	1	1	1	0	1	0	1	0
E	0	0	0	1	0	0	0	1
F	0	0	1	0	0	0	1	0
G	0	0	1	1	0	1	0	1
H	0	0	0	0	1	0	1	0

## Průchod grafem do hloubky

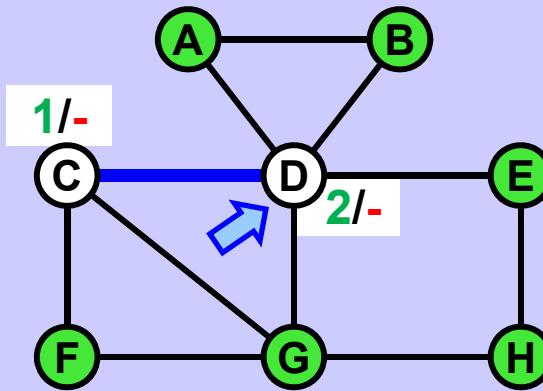


Zásobník

C

Výstup

C

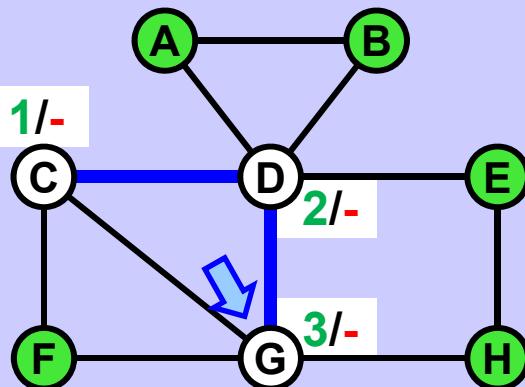


Zásobník

C D

Výstup

C D

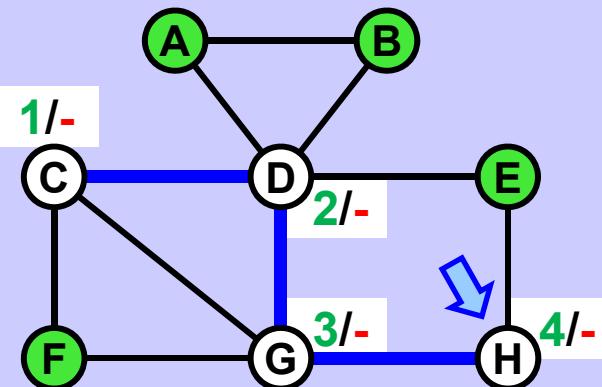


Zásobník

C D G

Výstup

C D G



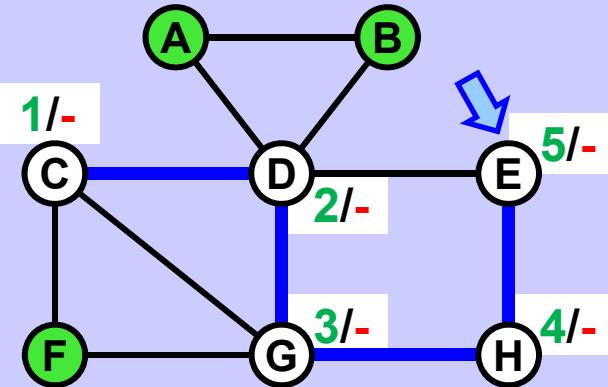
Zásobník

C D G H

Výstup

C D G H

## Průchod grafem do hloubky

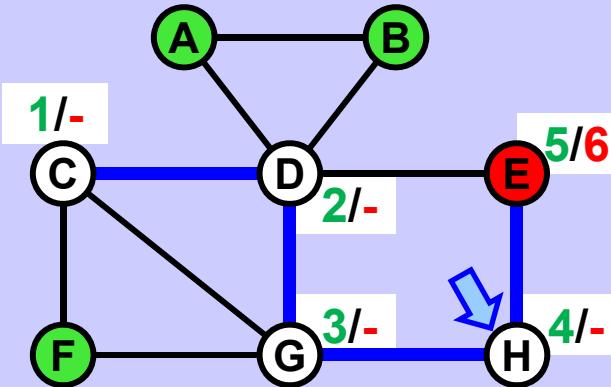


Zásobník

C D G H E

Výstup

C D G H E

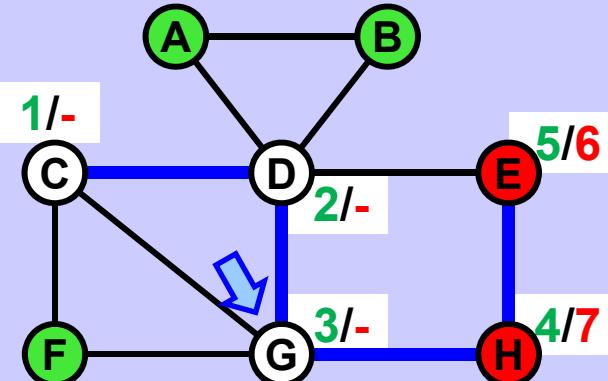


Zásobník

C D G H

Výstup

C D G H E

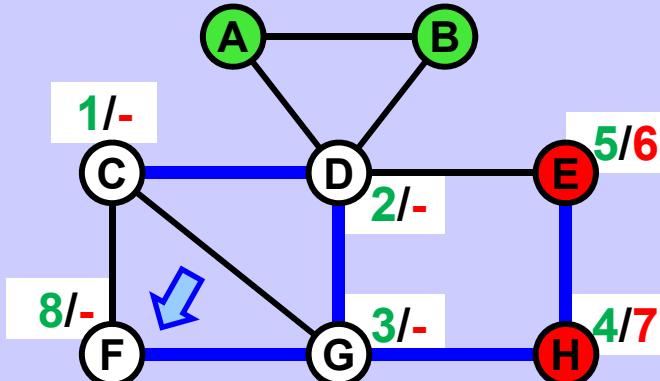


Zásobník

C D G

Výstup

C D G H E



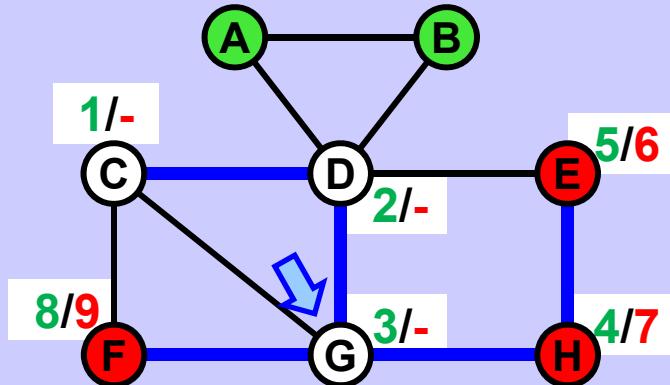
Zásobník

C D G F

Výstup

C D G H E F

## Průchod grafem do hloubky

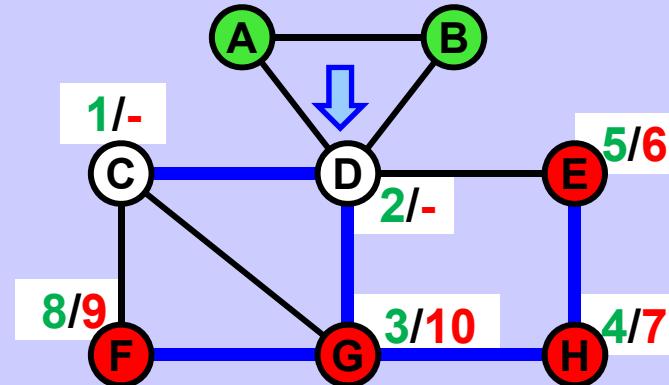


Zásobník

C D G

Výstup

C D G H E F

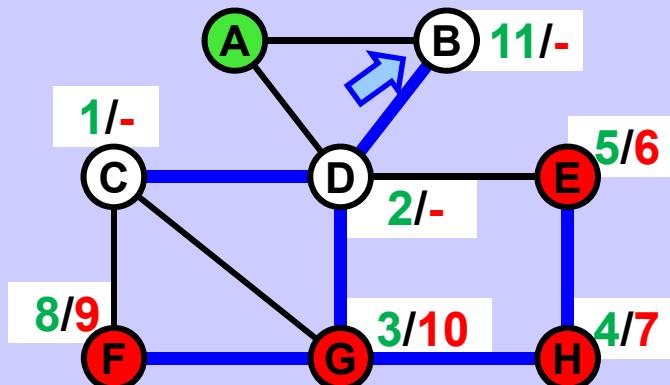


Zásobník

C D

Výstup

C D G H E F

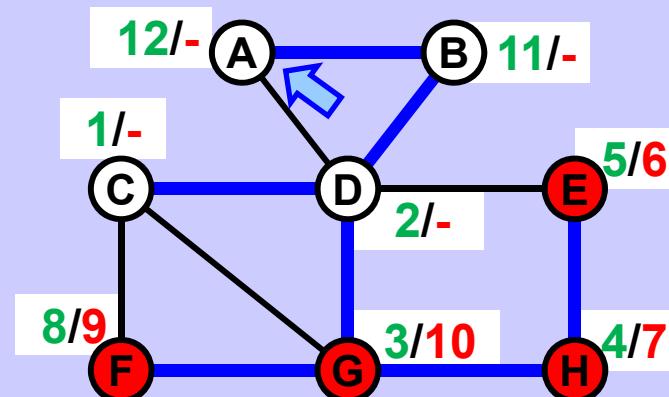


Zásobník

C D B

Výstup

C D G H E F B



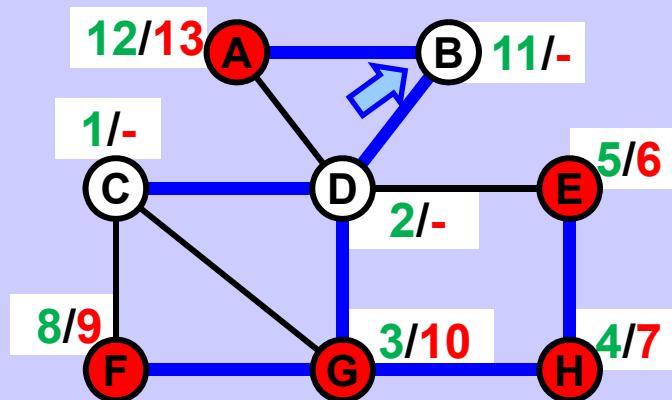
Zásobník

C D B A

Výstup

C D G H E F B A

## Průchod grafem do hloubky

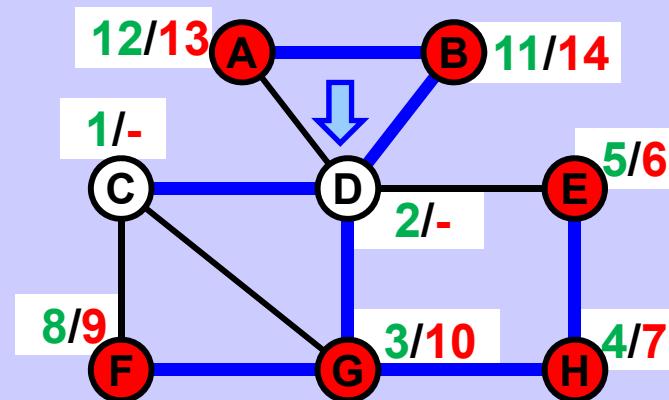


Zásobník

C D B

Výstup

C D G H E F B A

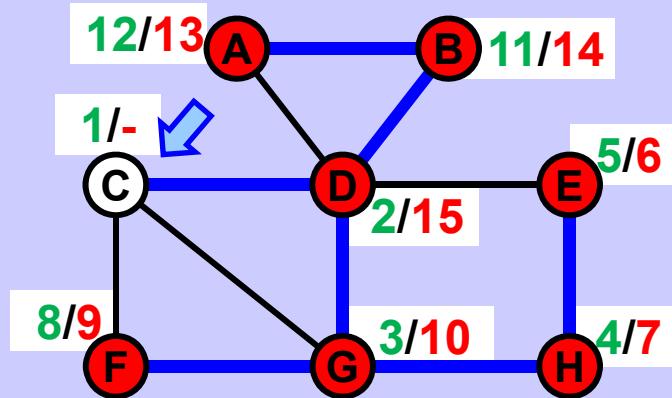


Zásobník

C D

Výstup

C D G H E F B A

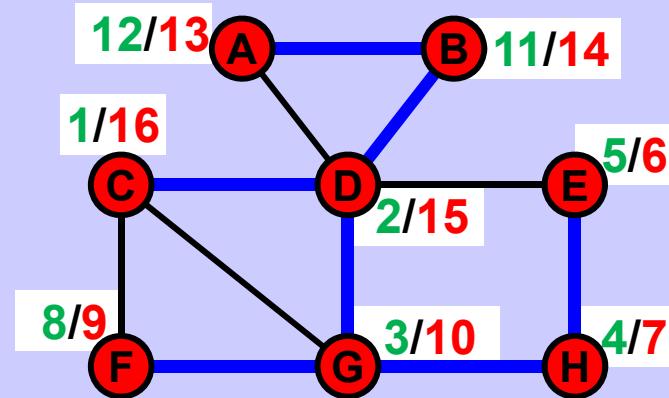


Zásobník

C

Výstup

C D G H E F B A



Zásobník

C D G H E F B A

Výstup

## Průchod grafem do hloubky

### Životní cyklus uzlu při prohledávání grafu Fresh - open - closed ( čerstvý - otevřený - uzavřený )

#### Fresh

Čerstvé uzly jsou všechny dosud ani jednou navštívené uzly.  
Před začátkem prohledávání jsou všechny uzly čerstvé.  
Při první návštěvě uzlu se uzel stává otevřeným.  
Množina čerstvých uzelů se během prohledávání nikdy nezvětšuje vzhledem k inkluzi.

#### Open

Otevřené uzly jsou alespoň jednou navštívené uzly, které dosud nebyly uzavřeny.  
Množina otevřených uzelů se může během prohledávání zvětšovat i zmenšovat.

#### Closed

Uzavřené uzly jsou uzly, které už během prohledávání nebudou navštíveny.  
Pokud jsou všechny sousedy aktuálního uzlu otevřené nebo uzavřené, aktuální uzel se stává uzavřeným.  
Množina uzavřených uzelů se během prohledávání nikdy nezmenšuje vzhledem k inkluzi.  
Na konci prohledávání jsou všechny uzly uzavřené.

## Průchod grafem do hloubky

### Implementační poznámka

**Fresh:** Čerstvý uzel nemá přiřazen otevírací (ani zavírací) čas.

**Open:** Otevřený uzel nemá přiřazen zavírací čas.

**Closed:** Uzavřený uzel má přiřazen zavírací čas.

Otevírací a zavírací časy uzelů v některých případech prohledávání není nutno udržovat.

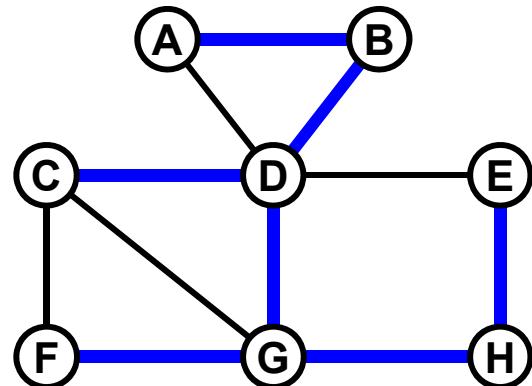
Při iterativním prohledávání s vlastním zásobníkem je ale nutno průběžně udržovat informaci u každého uzlu, zda je čerstvý, otevřený nebo zavřený.

V rekurzivním zpracování není nutno dělat explicitně ani to. Každé volání rekurzivní funkce odpovídá zpracování jednoho uzlu a všem jeho návštěvám. Při zavolání funkce se otevírá uzel, který je aktuálním parametrem volání, a na konci volání se tento uzel uzavírá. V těle funkce probíráme postupně sousedy aktuálního uzlu a voláme rekurzivně prohledávání pouze na ty z nich, které jsou ještě čerstvé (fresh). Stačí pak v každém uzlu udržovat jen informaci jednobitovou -- fresh nebo not fresh.

## Průchod grafem do hloubky

Postupný obsah zásobníku

C
C D
C D G
C D G H
C D G H E
C D G H
C D G
C D G F
C D G
C D
C D B
C D B A
C D B
C D
C
--



Výpis (zpracování) uzlu při otevřání uzlu vede na posloupnost

C D G H E F B A

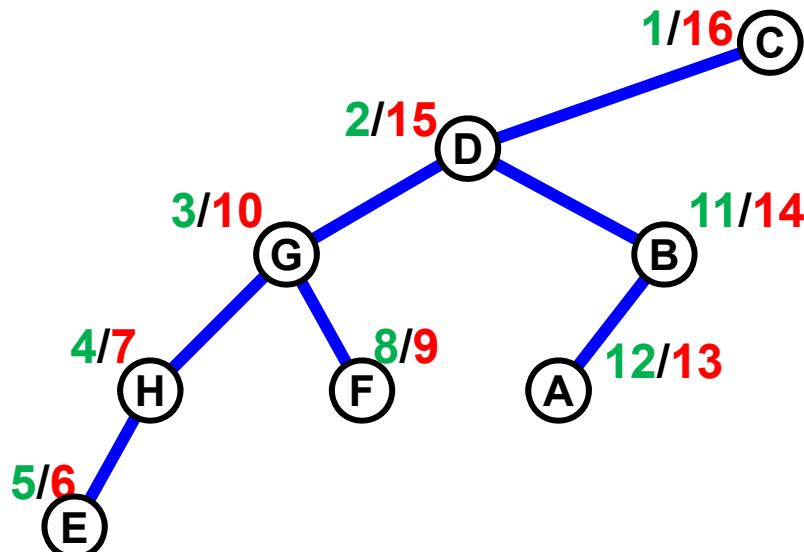
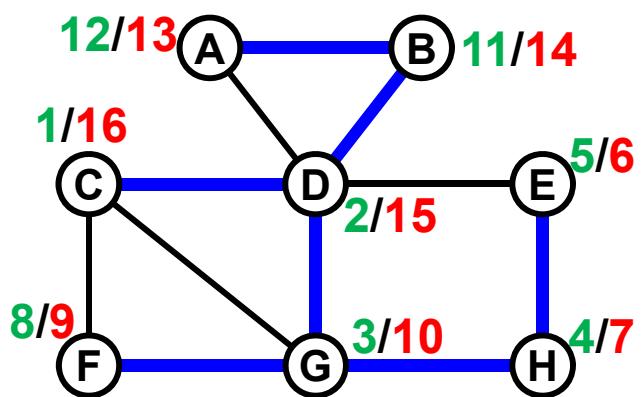
Výpis (zpracování) uzlu při zavírání uzlu vede na posloupnost

E H F G A B D C

Zpracování uzlu při jeho závírání se uplatní např. při hledání mostů nebo artikulací v neorientovaném grafu a při detekci silně souvislých komponent v orientovaném grafu.

## Průchod grafem do hloubky

Strom průchodu do hloubky  
s otevíracími a zavíracími  
časy jednotlivých uzelů



Všimněme si, že v podstromu s kořenem X pro každý uzel Y různý od X platí  
 $\text{Open\_time}(X) < \text{Open\_time}(Y) < \text{Close\_time}(Y) < \text{Close\_time}(X)$ .  
Naopak, pokud Y neleží v podstromu s kořenem X, pak platí  
 $\text{Close\_time}(X) < \text{Open\_time}(Y)$  nebo  $\text{Close\_time}(Y) < \text{Open\_time}(X)$

Počet uzelů v podstromu s kořenem X je pak  
 $(\text{Close\_time}(X) + 1 - \text{Open\_time}(X)) / 2$ .

## Průchod grafem do hloubky

```
void DFS_rec_full( int start ) {
    vector<int> openT( N, 0 );
    vector<int> closeT( N, 0 );
    vector<int> pred( N, -1 ); // -1 == no predecessor
    int time = 0;
    DFS_rec_full( start, time, openT, closeT, pred );

    for( int n = 0; n < N; n++ )
        cout << "node " << n << " open/close "
            << openT[n] << "/"<< closeT[n]
            << " pred " << pred[n] << endl;
}
```

## Průchod grafem do hloubky rekurzivně

```
void DFS_rec_full( int currNode, int & time,
                    vector<int> & openT,
                    vector<int> & closeT,
                    vector<int> & pred ){

    int neigh;
    openT[currNode] = ++time;
    for( int j = 0; j < edge[currNode].size(); j++ ){
        neigh = edge[currNode][j];
        if( openT[neigh] == 0 ) {
            pred[neigh] = currNode;
            DFS_rec_full( neigh, time, openT, closeT, pred );
            cout << currNode << " --> " << neigh << endl;
        }
    }
    closeT[currNode] = ++time;
}
```

## Průchod grafem do hloubky rekurzivně, základní varianta

```
void DFS_rec_plain( int currNode, vector<bool> & fresh ) {
    int neigh;
    fresh[currNode] = false;
    for( int j = 0; j < edge[currNode].size(); j++ ){
        neigh = edge[currNode][j];
        if( fresh[neigh] ) {
            DFS_rec_plain( neigh, fresh );
            cout << currNode << " --> " << neigh << endl;
        }
    }
}

void DFS_rec_plain( int start) {
    vector<bool> fresh( N, true );
    DFS_rec_plain(start, fresh);
}
```

## Průchod grafem do šířky

**Životní cyklus uzlu při prohledávání grafu je koncepčně identický jako při prohledávání do hloubky**

### Fresh

Čerstvé uzly jsou všechny dosud ani jednou navštívené uzly.  
Před začátkem prohledávání jsou všechny uzly čerstvé.  
Při první návštěvě uzlu se uzel stává otevřeným.  
Množina čerstvých uzelů se během prohledávání nikdy nezvětšuje vzhledem k inkluzi.

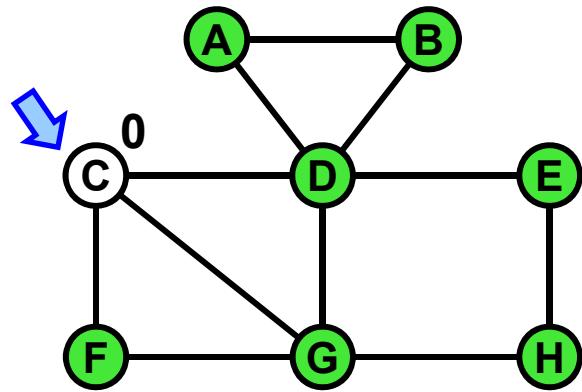
### Open

Otevřené uzly jsou alespoň jednou navštívené uzly, které dosud nebyly uzavřeny.  
Množina otevřených uzelů se může během prohledávání zvětšovat i zmenšovat.

### Closed

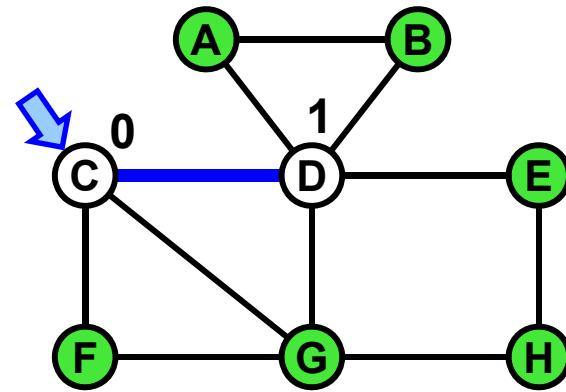
Uzavřené uzly jsou uzly, které už během prohledávání nebudou navštíveny.  
Pokud jsou všechny sousedy aktuálního uzlu otevřené nebo uzavřené, aktuální uzel se stává uzavřeným.  
Množina uzavřených uzelů se během prohledávání nikdy nezmenšuje vzhledem k inkluzi.  
Na konci prohledávání jsou všechny uzly uzavřené.

## Průchod grafem do šírky



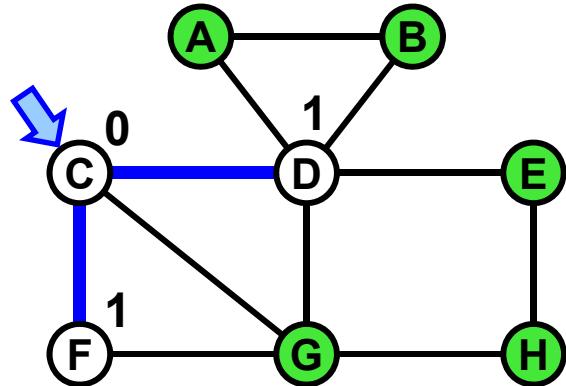
<b>Fronta</b>	C
<b>Výstup</b>	C

---



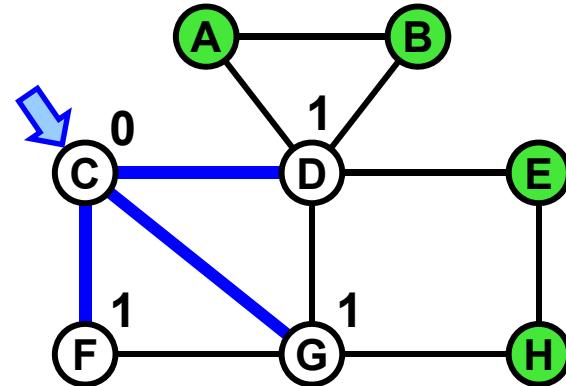
<b>Fronta</b>	D
<b>Výstup</b>	C

---



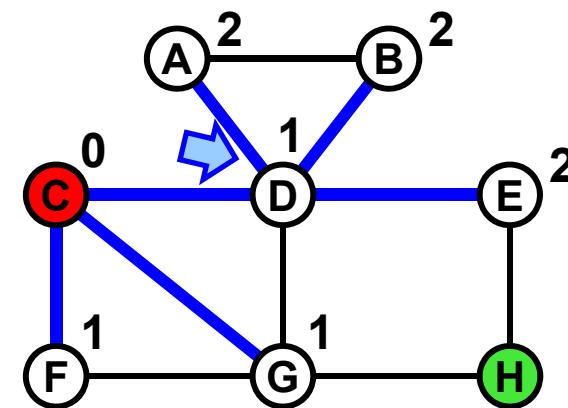
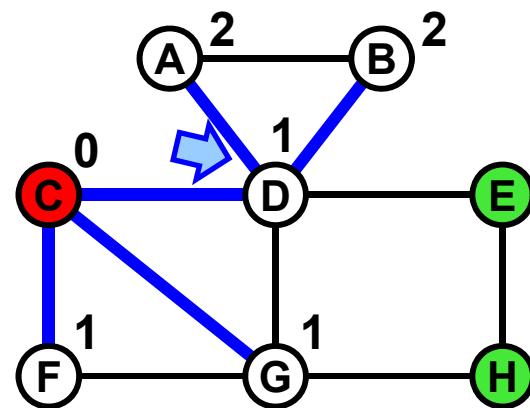
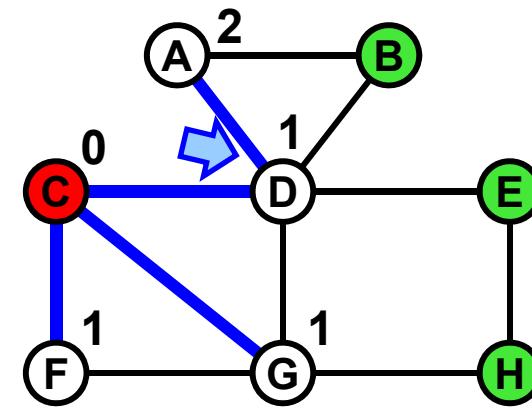
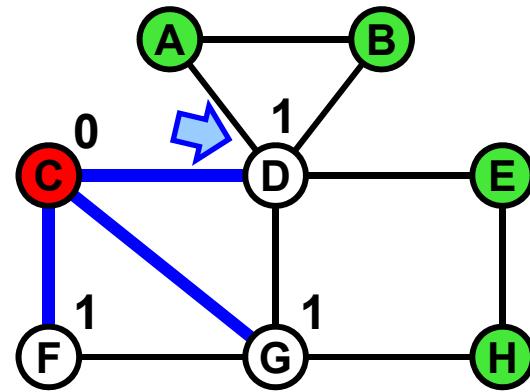
<b>Fronta</b>	D F
<b>Výstup</b>	C

---

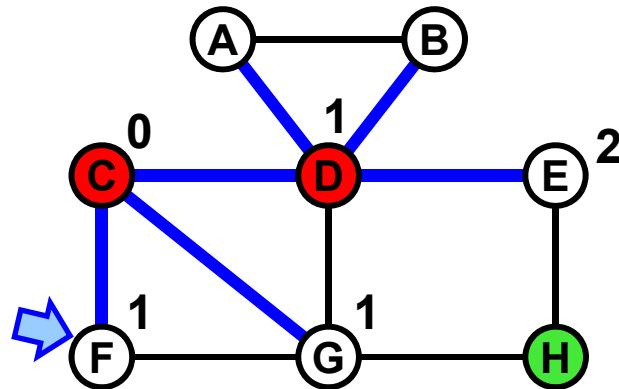


<b>Fronta</b>	D F G
<b>Výstup</b>	C

## Průchod grafem do šírky



## Průchod grafem do šírky

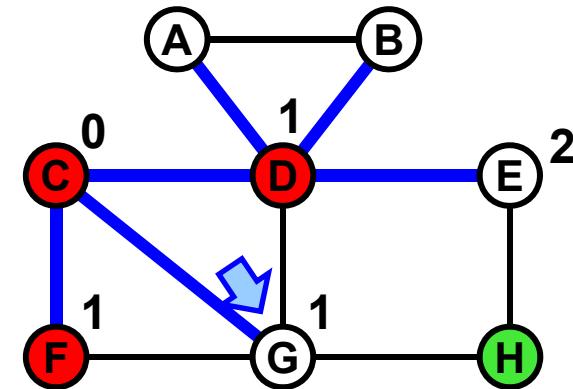


Fronta

Výstup

F G A B E

C D F

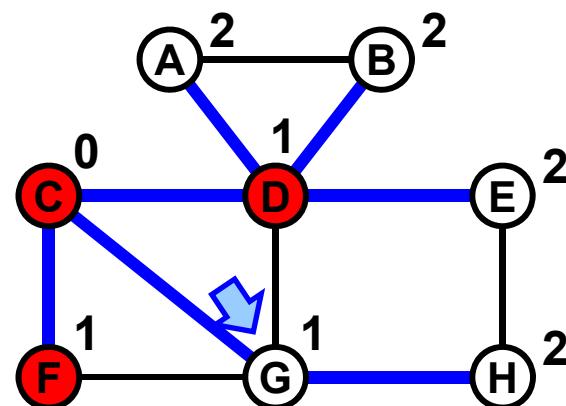


Fronta

Výstup

G A B E

C D F G

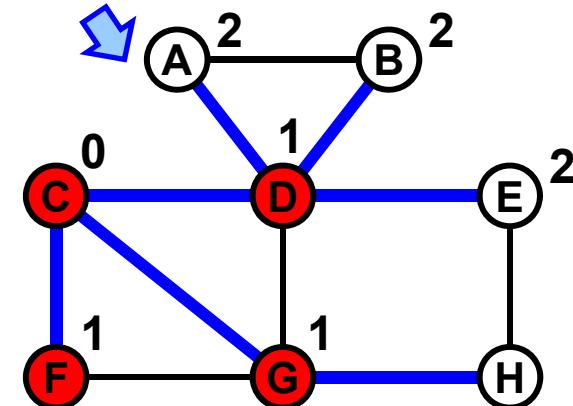


Fronta

Výstup

A B E H

C D F G



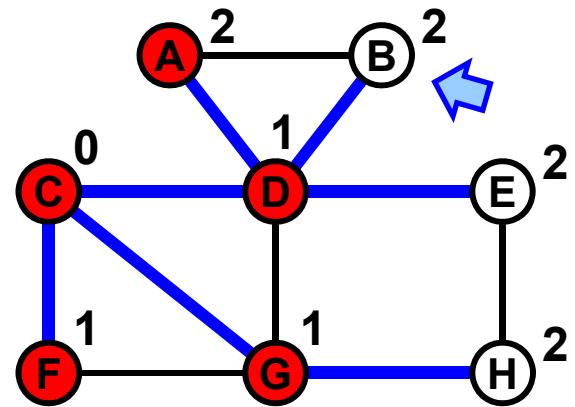
Fronta

Výstup

A B E H

C D F G A

## Průchod grafem do šírky

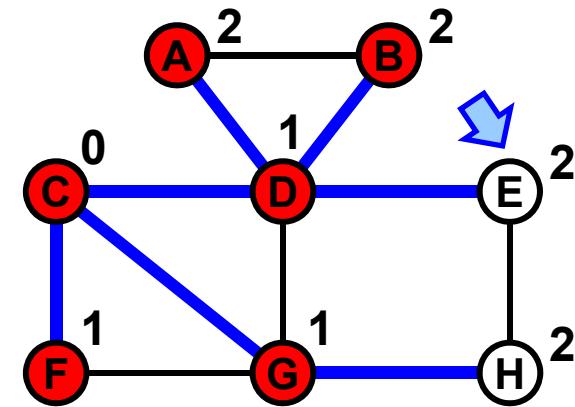


Fronta

B E H

Výstup

C D F G A B

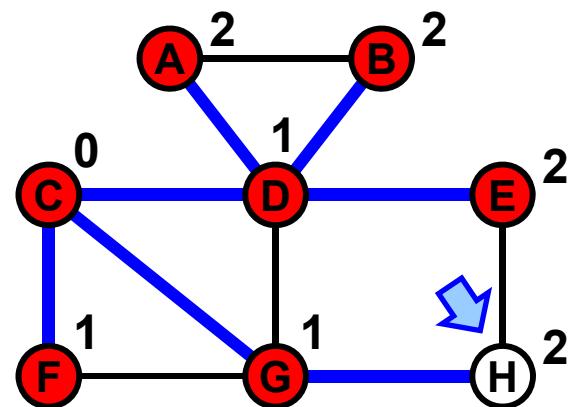


Fronta

E H

Výstup

C D F G A B E

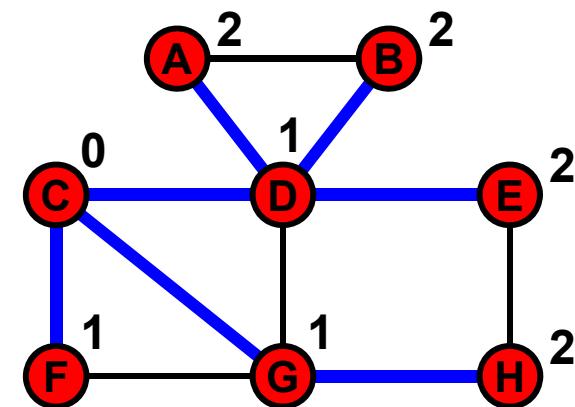


Fronta

H

Výstup

C D F G A B E H



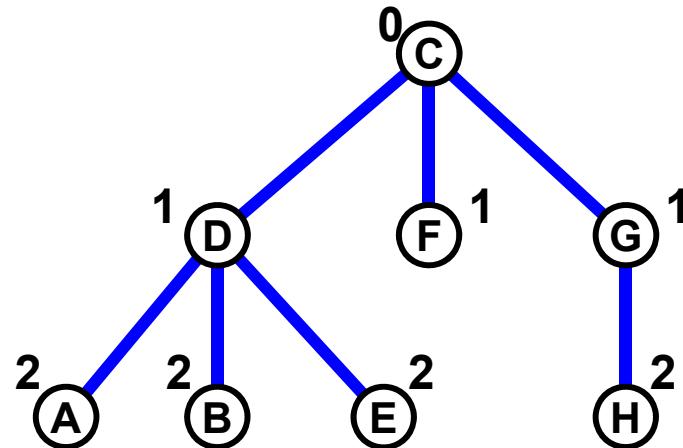
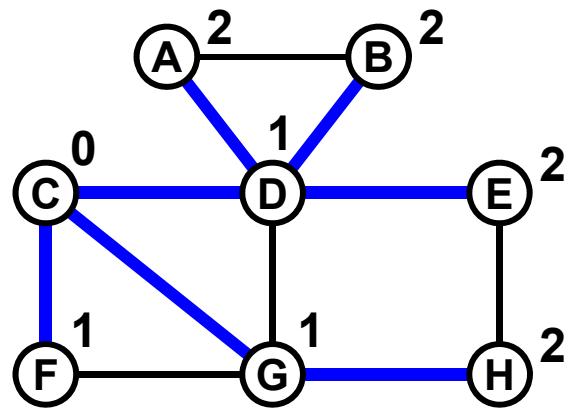
Fronta

C D F G A B E H

Výstup

## Průchod grafem do šířky

Strom průchodu do šířky  
s vzdálenostmi od kořene jednotlivých uzlů



Hloubka uzlu ve stromu průchodu do šířky  
je rovna jeho vzdálenosti od startovního uzlu.

Na rozdíl od průchodu do hloubky nejsou otevírací a zavírací časy důležité.

Průchod do šířky se uplatní např. při testování souvislosti grafu, testování existence cyklů ve grafu, testování bipartitnosti a zejména při výpočtu vzdálenosti mezi startovním a cílovým uzlem.

## Průchod grafem do šířky

### Implementační poznámka

**Fresh:** Čerstvý uzel nemá přiřazenu vzdálenost od startovního uzlu.

**Open:** Otevřený uzel má přiřazenu vzdálenost od startovního uzlu a je ve frontě.

**Closed:** Uzavřený uzel má přiřazenu vzdálenost od startovního uzlu a není ve frontě.

Stavy fresh/open/closed není nutno explicitně udržovat, obsah fronty společně s vzdáleností jednoznačně tyto stavy určují každému uzlu.

Průchod do šířky je iterativní postup, rekurzivní varianta se nepoužívá (byla pouze nepřehlednější).

## Průchod grafem do šířky

### Pseudokód

```
void graphBreadthFirstSearch( Node startNode ) {
    Set visited = new Set();           // visited == not fresh
    Queue q = new Queue();
    q.Enqueue( startNode );
    visited.add( startNode );
    while( !q.Empty() ) {
        node = q.Dequeue();
        print( node.key );           // process node
        forall x in node.Neighbors()
            if( x not in visited ){
                visited.add( x );
                q.Enqueue( x );
            }
    }
}
```

## Průchod grafem do šířky v C++

```
void BFS( int start ){
    queue<int> q;
    vector<int> dist( N, -1 ); // -1 ... fresh node
    vector<int> pred( N, -1 ); // -1 ... no predecessor (yet)

    q.push( start );
    dist[start] = 0;

    int currn, neigh;
    while( !q.empty() ){
        currn = q.front(); q.pop();
        cout << "node " << currn << endl; // process the node
        for( int j = 0; j < edge[currn].size(); j++ ){
            neigh = edge[currn][j];
            if( dist[neigh] == -1 ){           // neigh is fresh
                q.push( neigh );
                dist[neigh] = dist[currn] + 1;
                pred[neigh] = currn;
                cout << "BFS edge: " << currn << "->" << neigh << endl;
            }
        }
    }
}
```

## Průchod grafem do hloubky i do šířky

### Asymptotická složitost

Každá jednotlivá operace se zásobníkem/frontou a s použitými datovými strukturami má konstantní složitost pro jeden uzel (včetně inicializace).

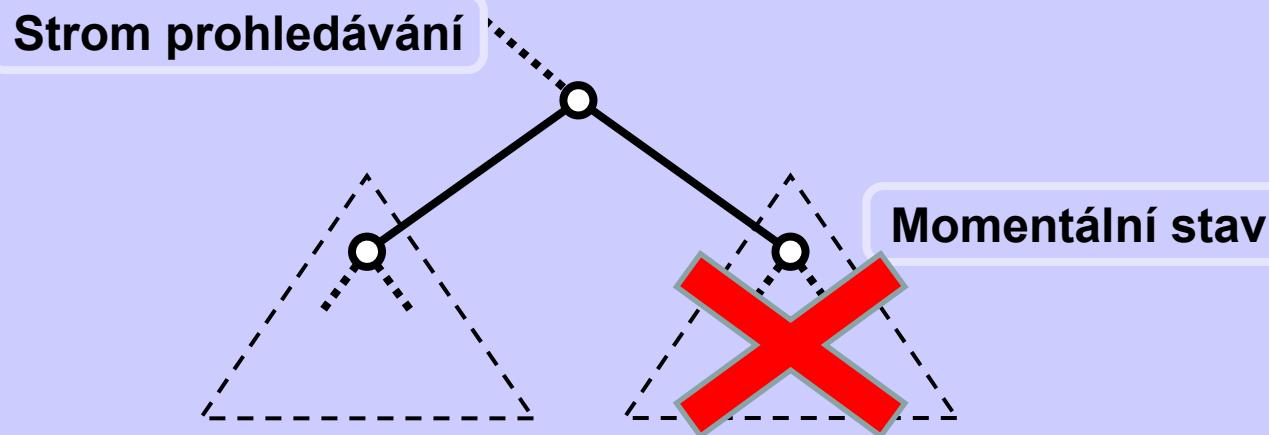
Každý uzel jen jednou vstoupí do zásobníku/fronty a jednou z ní vystoupí.  
Stav uzlu (fresh/open/closed) se testuje tolíkrát, kolik je stupeň tohoto uzlu.  
Součet stupnů všech uzlů je roven dvojnásobku počtu hran.

Celkem

$$\Theta(|V| + |E|).$$

## Ořezávání

- Urychlení prohledávání
  - Ořezávání neperspektivních větví
  - Pokud jsme schopni na základě vyhodnocení momentálního stavu zjistit,
    - že je to stav neperspektivní a
    - že rozhodně nepovede k řešení úlohy
- „odřízneme“ ze stromu celý podstrom momentálního stavu



## Příklad ořezávání – magický čtverec

- Magický čtverec řádu  $\mathcal{N}$ 
  - čtvercové schéma čísel velikost  $\mathcal{N} \times \mathcal{N}$
  - obsahuje právě jednou každé celé číslo od 1 do  $\mathcal{N}^2$
  - součet čísel ve všech řádcích a ve všech sloupcích stejný
- Příklad
  - |   |   |   |
|---|---|---|
| 2 | 9 | 4 |
| 7 | 5 | 3 |
| 6 | 1 | 8 |
- Triviální řešení: generování všech možných rozmístění čísel od 1 do  $\mathcal{N}^2$
- Ořezávání: kdykoliv je součet na řádku neperspektivní
  - součet všech čísel čtverce je  $\frac{1}{2} \mathcal{N}^2 (\mathcal{N}^2 + 1)$
  - součet čísel na řádku je  $\frac{1}{2} \mathcal{N} (\mathcal{N}^2 + 1)$

## Heuristiky

- Heuristika je návod, který nám říká, jaký postup řešení úlohy vede obvykle k rychlému dosažení výsledku.
- Nezaručuje vždy zrychlení výpočtu.
- Heuristika se používá pro stanovení pořadí,
  - v jakém se zkoumají možné průchody stromem/grafem

- Příklad: úloha projít šachovým koněm celou šachovnici  $\mathcal{N} \times \mathcal{N}$
- účinná heuristika: nejprve se navštíví ta dosud nenavštívená pole, z nichž bude nejméně možností dalšího bezprostředního pokračování cesty koně.
- urychlení na šachovnici  $8 \times 8$  až **stotisíckrát**.

# ALG 04

## **VYHLEDÁVÁNÍ (jednorozměrné vyhledávání)**

### **Vyhledávání v poli**

naivní, binární, interpolační

### **Binární vyhledávací strom (BVS)**

operace Find, Insert, Delete

## Hledání v seřazeném poli — lineární, POMALÉ

Dané pole

Seřazené pole:

Velikost = N

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

najdi 993 !

Testů: N



363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

najdi 363 !

Testů: 1



363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



# Hledání v seřazeném poli — binární, RYCHLEJŠÍ

najdi 863 !

2 testy

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993
363	369	388	603	638	693	803	833		839	860	863	938	939	966	968	983	993

2 testy

2 testy

839	860	863	938	939	966	968	983	993
839	860	863	938		966	968	983	993

2 testy

839	860	863	938
839		863	938

1 test

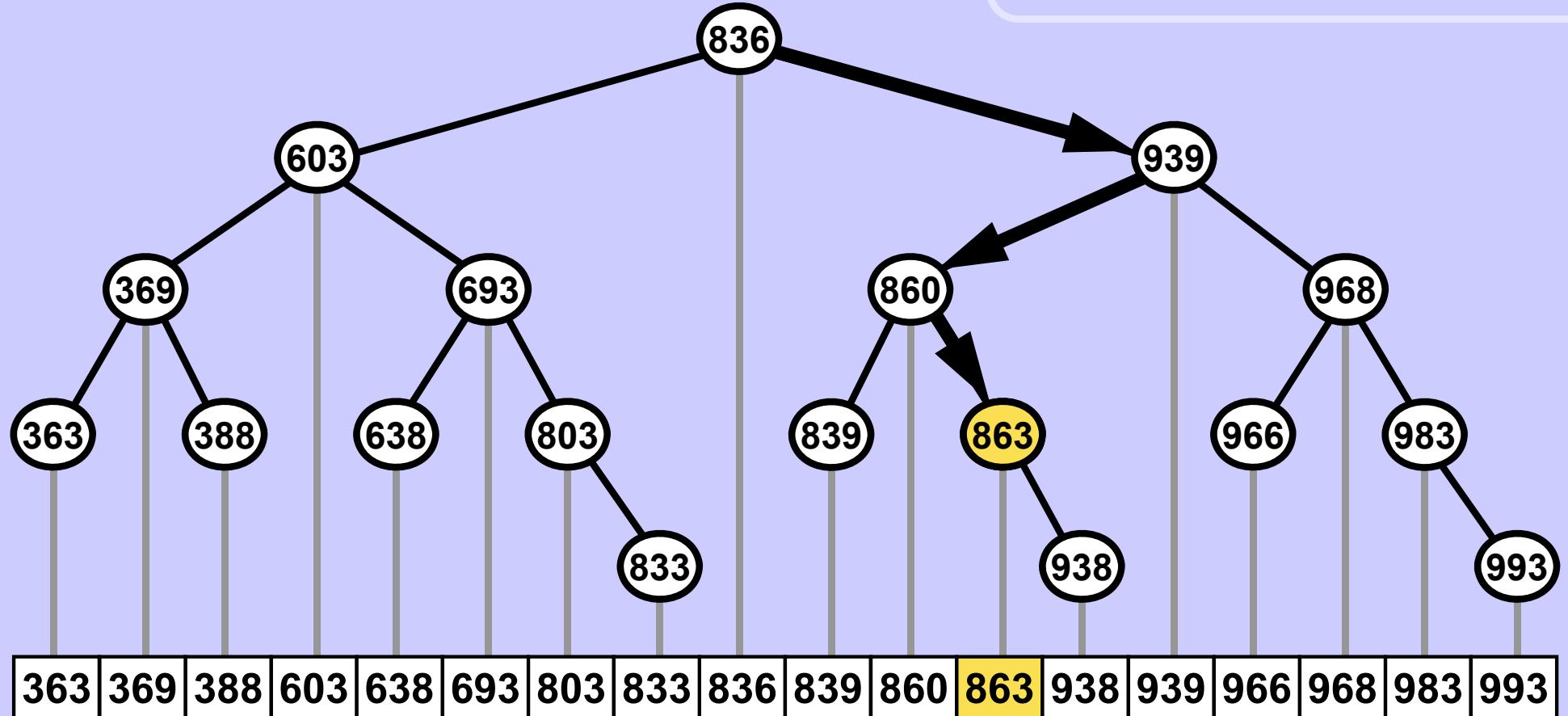




## Hledání v seřazeném poli — binární, RYCHLEJŠÍ

najdi 863 !

Hledání kopíruje strukturu binárního stromu





# Hledání v seřazeném poli — binární, JEŠTĚ RYCHLEJŠÍ

najdi 863 !

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
363	369	388	600	630	693	803	833	836	839	860	863	938	939	966	968	983	993

Typicky ( v průměrném případě) je hledaná hodnota blízko listu ve stromu vyhledávání.

Je zbytečné během sestupu stromem testovat, zda byla již hledaná hodnota nalezena.

Nejprve se najde místo, kde přesně má být a teprve pak se kontroluje, zda tam opravdu je.

Prohledávaný úsek se vždy pouze půlí na levou část s hodnotami menšími nebo rovnými q a na pravou část s hodnotami většími než q.

839	860	863	938	939	966	968	983	993
-----	-----	-----	-----	-----	-----	-----	-----	-----

839	860	863	938	939
-----	-----	-----	-----	-----

839	860	863
-----	-----	-----



Celkem 5 testů

Hledaný prvek sice algoritmus "našel" už při 3. testu, ale jeho výskyt v poli potvrdil až později.

## Binární hledání -- rychlá varianta

Nejprve je s ohledem na uspořádání hodnot nalezen přesně index, na kterém se má hledaná hodnota vyskytovat. Teprve nakonec se kontroluje, zda na určeném indexu tato hodota opravdu je. V každé úrovni stromu se tak ušetří jeden test hledané hodnoty.

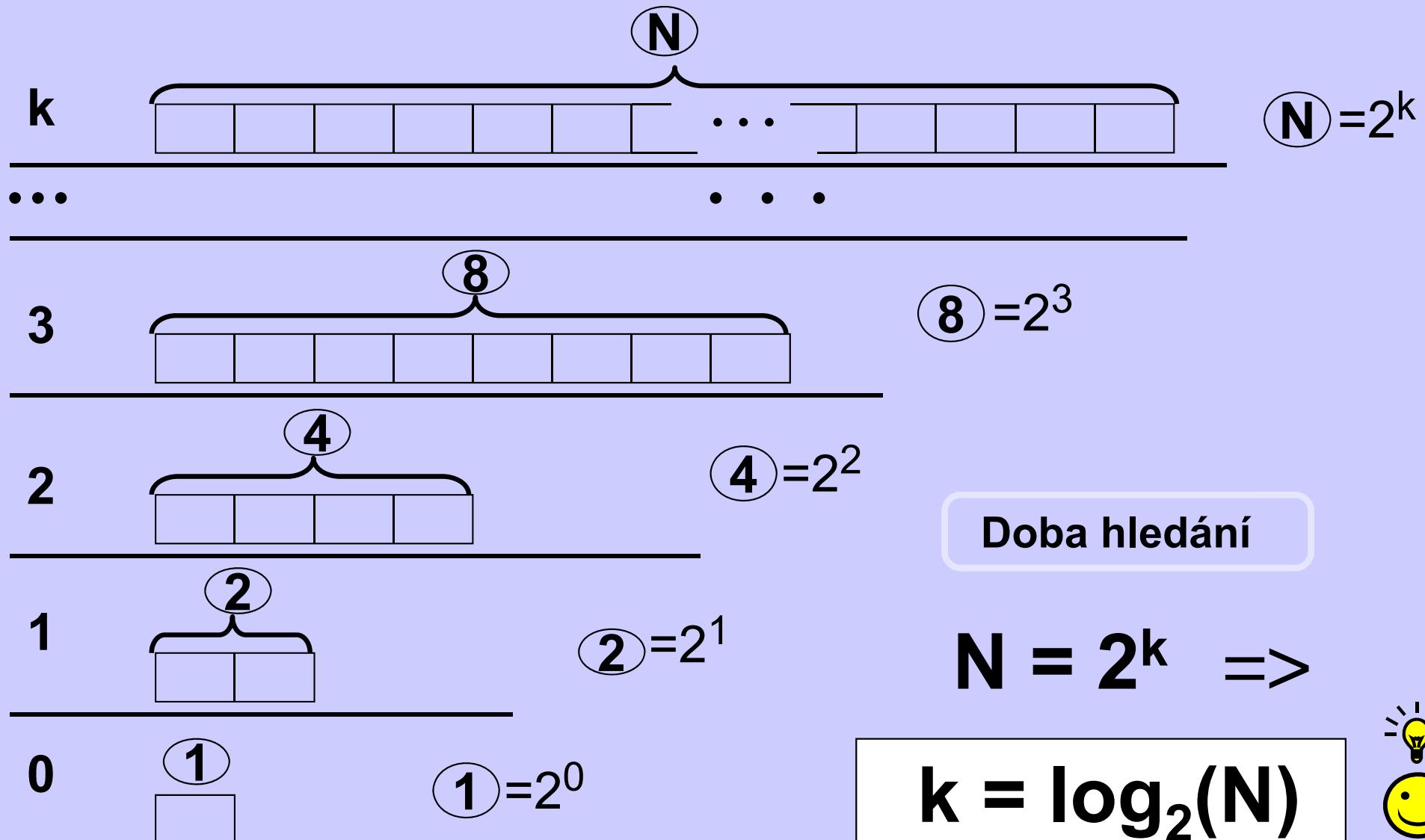
```
int binSearch( int [] arr, int q ) {
    int low = 0, high = length(arr)-1, mid;

    while( low < high ) {
        mid = (low+high)/2 ;           // bug ?
                                         // fix: mid = low + (high-low)/2;
        if( q > arr[mid] ) low = mid+1;
        else                  high = mid;
    }
    if( arr[low] == q ) return low;
    else return -1;
}
```

Bug? : Pro  $low + high > \text{max int}$  nastává chyba přetečení

<https://research.googleblog.com/2006/06/extr-extra-read-all-about-it-nearly.html>

## Hledání v seřazeném poli — binární, RYCHLEJŠÍ



## Interpolační hledání

Pole a[ ]

Najdi  $q = 939$

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993
0	1	2											13		15		17
first													position			last	

Mají-li hodnoty v poli víceméně rovnoměrně náhodné rozložení, je možno použít lineární interpolaci.

Poloha prvku v poli by měla přibližně odpovídat jeho velikosti.

$$\text{position} = \text{first} + \frac{(\text{q} - \text{a[first]})}{\text{a[last]} - \text{a[first]}} * (\text{last} - \text{first})$$

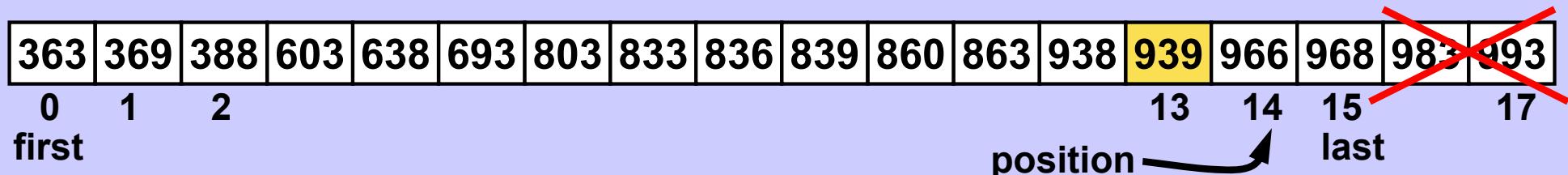
$$\text{position} = 0 + \frac{939 - 363}{993 - 363} * (17 - 0) = 15.54$$

Celá část!

## Interpolační hledání

Pole a[ ]

Najdi  $q = 939$



Když se na vypočtené pozici prvek nenalézá, je bud' vlevo nebo vpravo od ní a pak lze (rekurzivně) vzít za výchozí interval příslušnou levou nebo pravou část pole a výpočet opakovat.

$$\text{position} = \text{first} + \frac{(\text{q} - \text{a[first]})}{\text{a[last]} - \text{a[first]}} * (\text{last} - \text{first})$$

$$\text{position} = 0 + \frac{939 - 363}{968 - 363} * (15 - 0) = 14.12$$

Celá část!

## Interpolaci hledání

Pole a[ ]

Najdi q = 939

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	988	983	993
0	1	2											13	14	15		17
first													position	last			

Když se na vypočtené pozici prvek nenalézá, je bud' vlevo nebo vpravo od ní a pak lze (rekurzivně) vzít za výchozí interval příslušnou levou nebo pravou část pole a výpočet opakovat.

$$\text{position} = \text{first} + \frac{(\text{q} - \text{a[first]})}{\text{a[last]} - \text{a[first]}} * (\text{last} - \text{first})$$

$$\text{position} = 0 + \frac{939 - 363}{966 - 363} * (14 - 0) = 13.37$$

Celá část!

Hotovo

## Interpolaci hledání

```
int interpol( int [ ] arr, int q ) {  
    int first = 0;  
    int last = length(arr)-1;  
    do{  
        pos = first + round( (q-arr[first])/(arr[last]-arr[first])  
                            *(last-first) );  
        if( arr[pos] < q )      first = pos+1; // check left side  
        else if( arr[pos] > q ) last = pos-1; //check right side  
    }while( (arr[pos] != q) && (first < last) );  
  
    if( arr[pos] == q ) return pos;  
    else return -1; // not found  
}
```

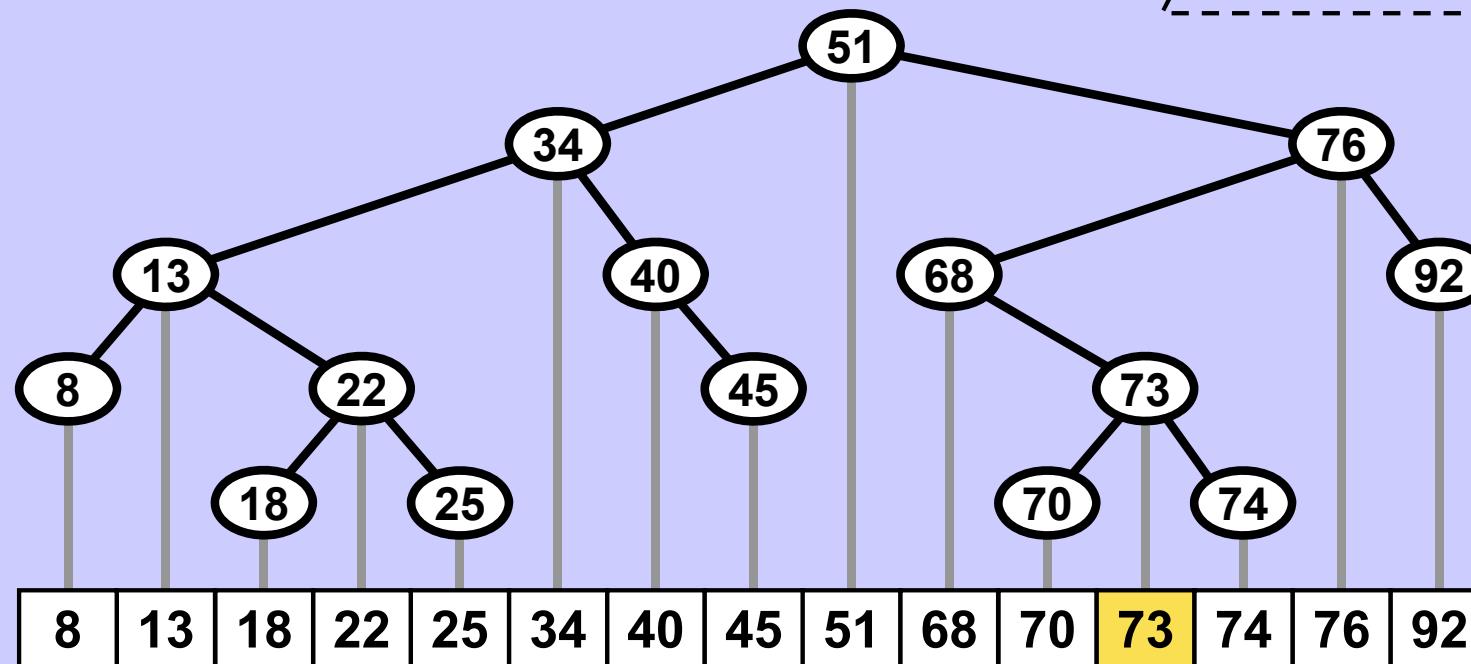
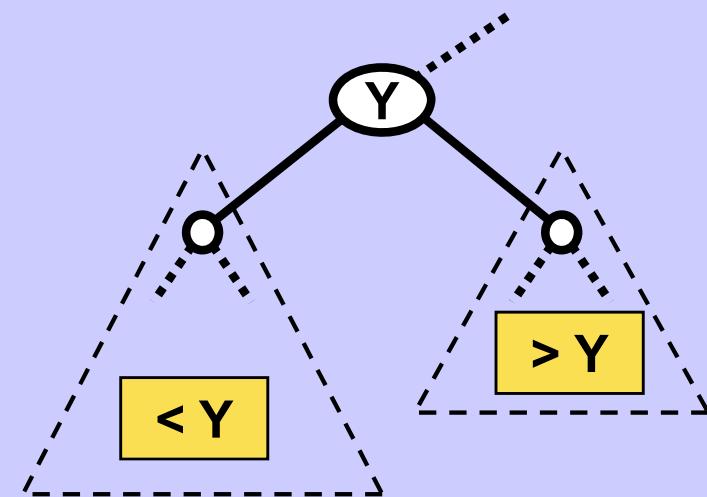
## Hledání v seřazeném poli — srovnání rychlostí

Velikost pole N	Lineární hledání průměrný případ	Interpolační hledání průměrný případ	Binární hledání pomalé nejhorší případ	rychlé pokaždé
10	5.5	1.60	9	4
30	15.5	2.12	11	5
100	50.5	2.56	15	7
300	150.5	2.89	19	9
1 000	500.5	3.18	21	10
3 000	1 500.5	3.41	25	12
10 000	5 000.5	3.63	29	14
30 000	15 000.5	3.80	31	15
100 000	50 000.5	3.96	35	17
300 000	150 000.5	4.11	39	19
1 000 000	500 000.5	4.24	41	20
	Zřejmě $\Theta(n)$	Na náhodném rovnoměrném rozdělení, teoreticky $\Theta(\log(\log(N)))$	Podle struktury binárního stromu $\Theta(\log(n))$	

## Binární vyhledávací strom

**V levém podstromu každého uzlu jsou všechny klíče menší.**

**V pravém podstromu každého uzlu jsou všechny klíče větší.**



## Binární vyhledávací strom

BVS nemusí být  
a nebývá vyvážený.

BVS nemusí být  
a nebývá pravidelný.

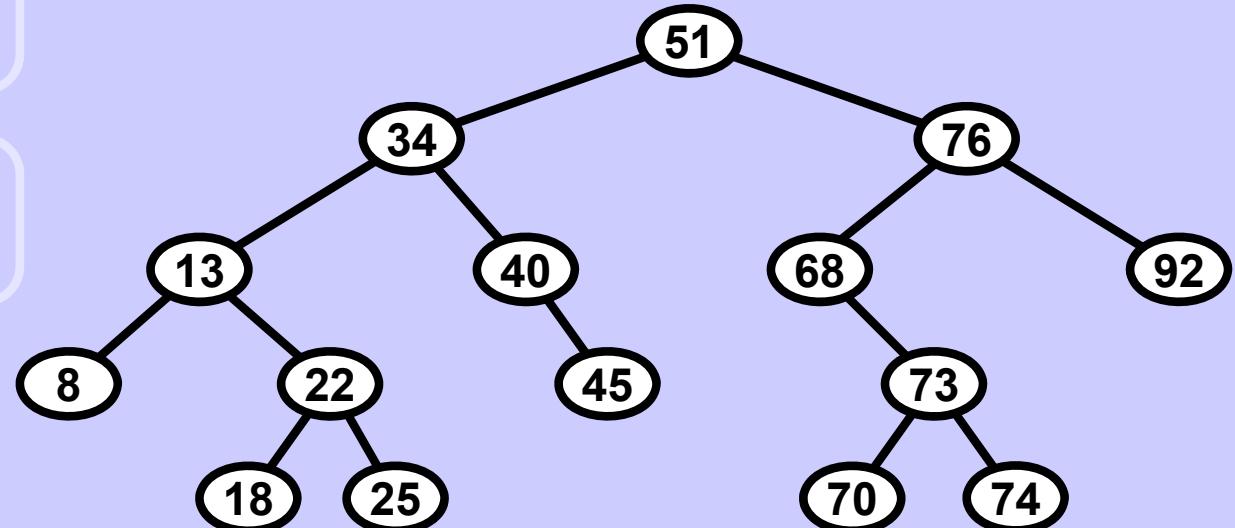
Výpisem prvků BVS  
v pořadí INORDER  
získáme uspořádané  
hodnoty klíčů.

BVS je flexibilní díky operacím:

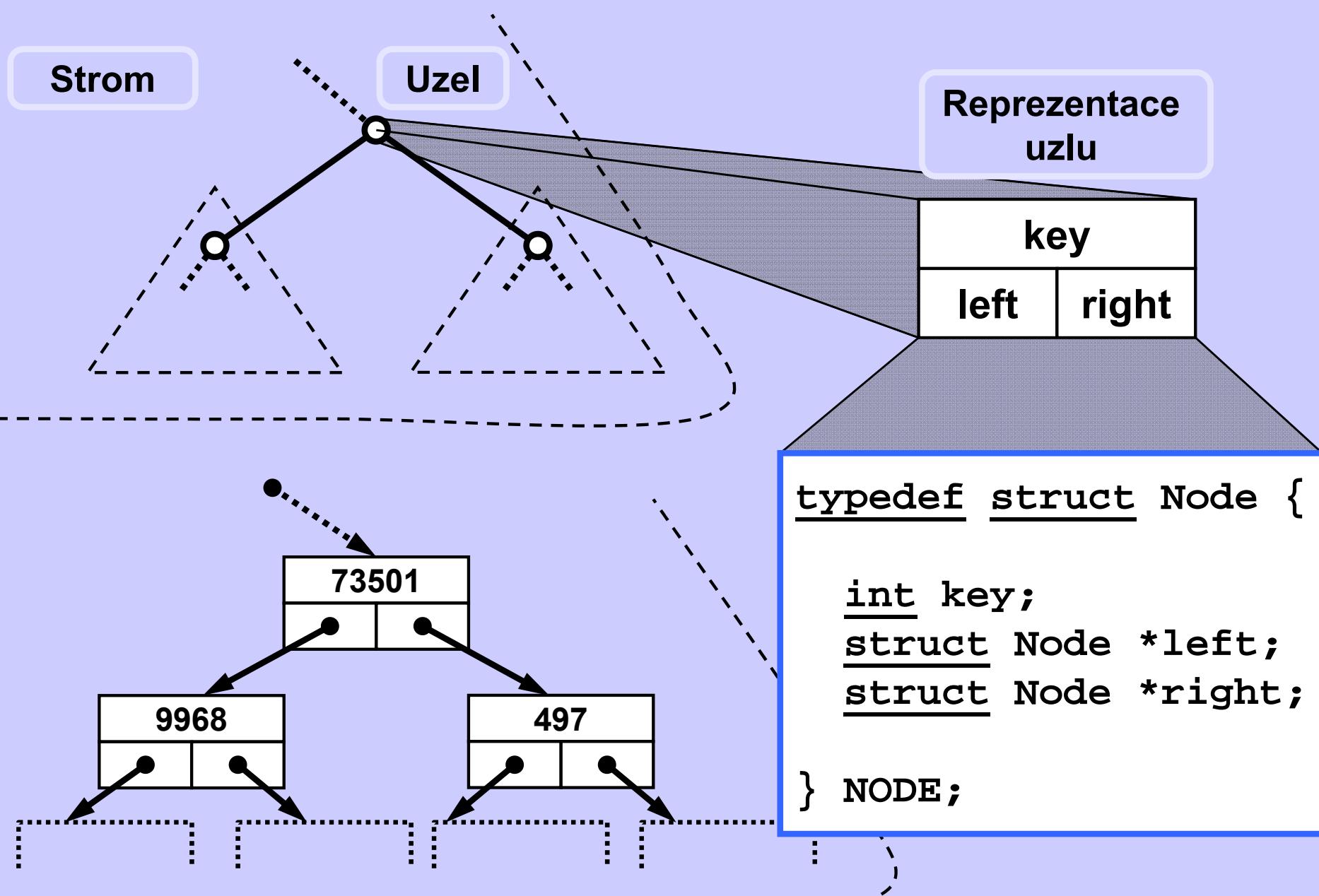
Find – najdi prvek s daným klíčem

Insert – vlož prvek s daným klíčem

Delete – (najdi a) odstraň prvek s daným klíčem

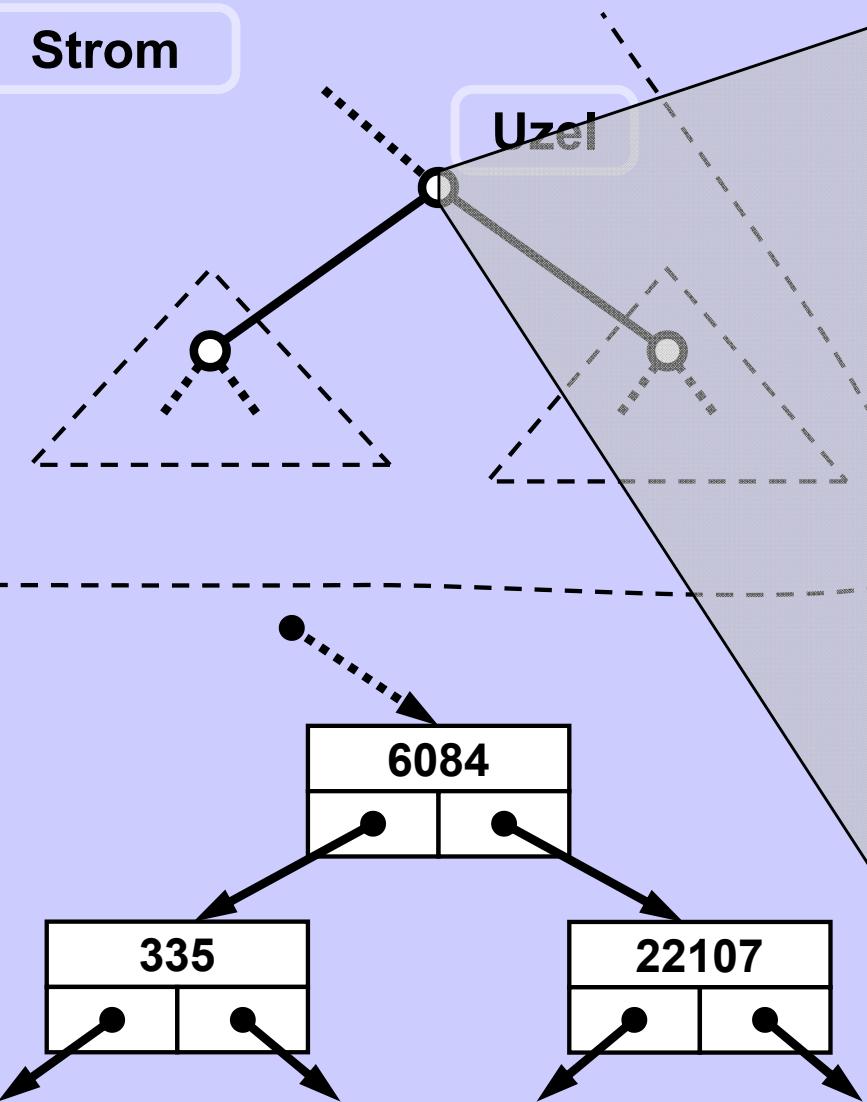


## Implementace binárního stromu -- C



## Implementace binárního stromu -- java

Strom



```

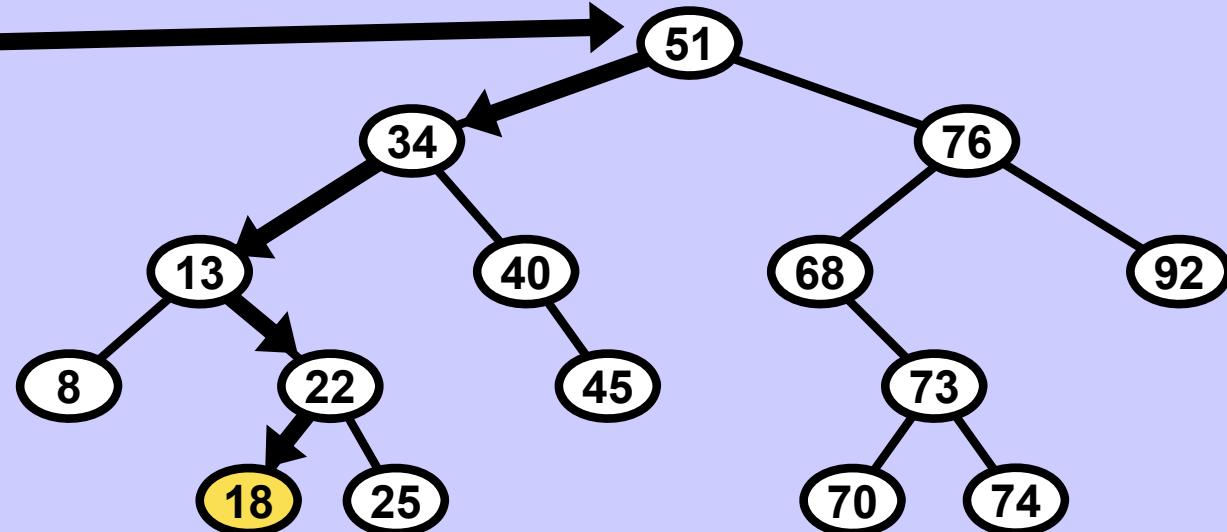
public class Node {
    public Node left;
    public Node right;
    public int key;
    public Node(int k) {
        key = k;
        left = null;
        right = null;
    }
}

public class Tree {
    public Node root;
    public Tree() {
        root = null;
    }
}
  
```

## Operace Find v BVS

Najdi 18

Každá operace se stromem začíná v kořeni.



Iterativně

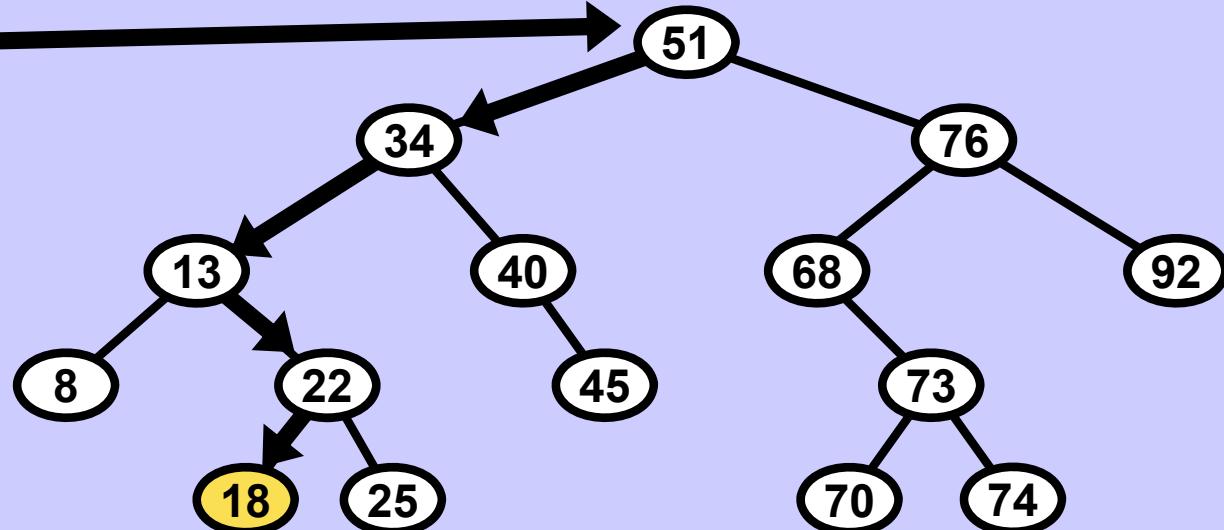
```

Node find( int k, Node node ){
    while( true ) {
        if( node == null ) return null;
        if( node->key == k ) return node;
        if( k < node->key ) node = node->left;
        else
    }
}
  
```

## Operace Find v BVS

Najdi 18

Každá operace se stromem začíná v kořeni.



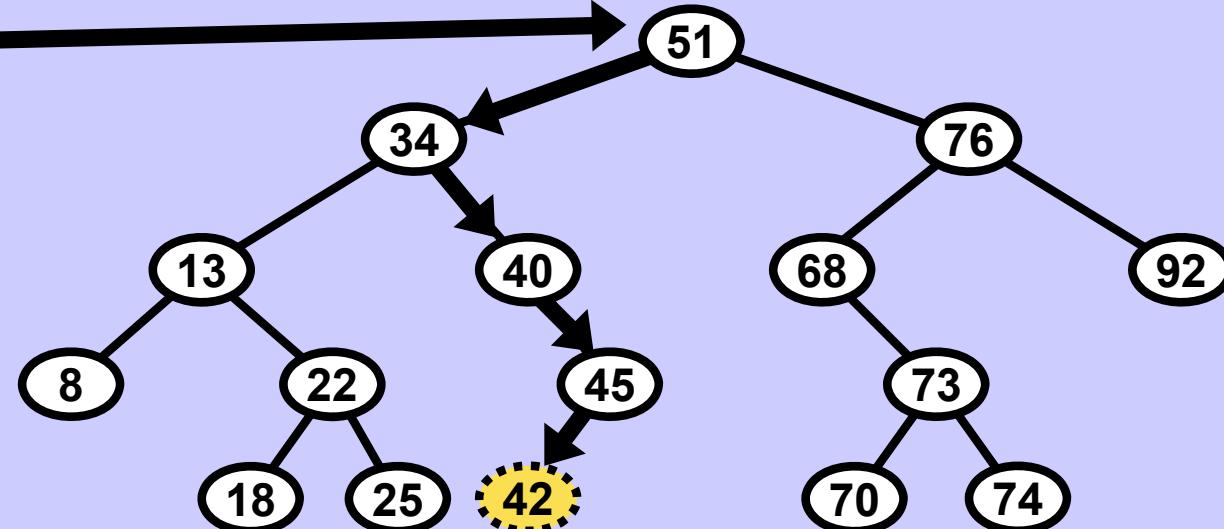
Rekurzivně

```

Node findRec( int k, Node node ){
    if( node == null ) return null;
    if( node->key == k ) return node;
    if( k < node->key ) return findRec( k, node->left );
    else return findRec( k, node->right );
} }
  
```

## Operace Insert v BVS

Vlož 42



Sem patří 42

### Insert

1. Najdi místo (jako ve Find) pro list, kam patří uzel s daným klíčem.
2. Vytvoř tento uzel a vlož jej do stromu.

## Operace Insert v BVS iterativně

```

Node insert( int k, Node node ){
    if( node == null ){                                // empty tree
        Node newNode = ...;                          // create node with key k
        return newNode;
    }
    while( true )
        if( node->key == k ) return null; //can't insert a duplicate
        if( node->key > k )
            if( node->left == null ){
                Node newNode = ...;                  // create node with key k
                node->left = newNode;
                return newNode; }
            else node = node->left;
        else
            if( node->right == null ){           // similarly to the right
                Node newNode = ...;
                node->right = newNode;
                return newNode; }
            else node = node->right; }
    } }

```

## Operace Insert v BVS rekurzivně

```

Node insertRec( int k, Node node, Node parentNode ){
    if( node == null ){                                // empty tree
        Node newNode = ...;                          // create node with key k
        if( parentNode != null ){
            if( parentNode->key > k )
                parentNode->left = newNode;
            else
                parentNode->right = newNode;
        }
        return newNode;
    }
    if( node->key == k ) return null;      // can't insert a duplicate
    if( node->key > k )                  // chose direction
        return insertRec( k, node->left, node );
    else
        return insertRec( k, node->right, node );
}

```

## Stavba BVS operací Insert

insert 40

40

insert 60

40  
—  
60

insert 50

40  
—  
60  
—  
50

insert 20

20  
—  
40  
—  
60  
—  
50

insert 70

20  
—  
40  
—  
60  
—  
50  
—  
70

insert 30

20  
—  
40  
—  
60  
—  
30  
—  
50  
—  
70

insert 10

10  
—  
20  
—  
40  
—  
60  
—  
30  
—  
50  
—  
70

Tvar budovaného BVS závisí na pořadí vkládání dat.

insert 50

A binary search tree with a single root node containing the value 50.

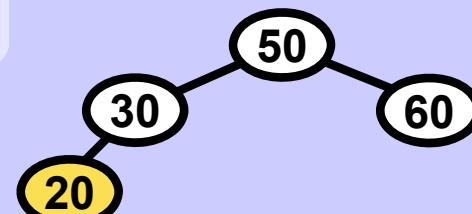
insert 30

A binary search tree with root 50. Node 30 is its left child. Both nodes are highlighted in yellow.

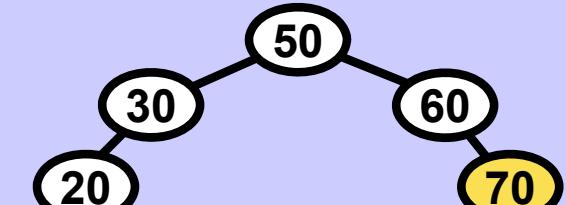
insert 60

A binary search tree with root 50. Nodes 30 (left) and 60 (right) are its children. All three nodes are highlighted in yellow.

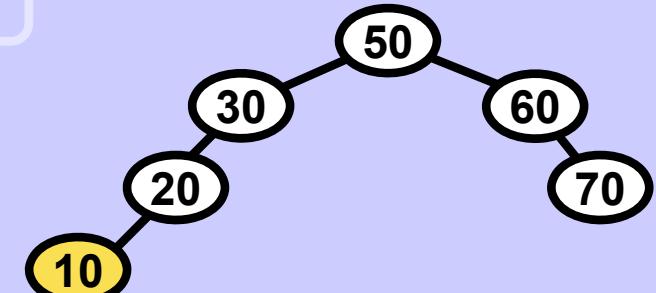
insert 20

A binary search tree with root 50. Nodes 30 (left), 60 (right), and 20 (left child of 30) are highlighted in yellow.

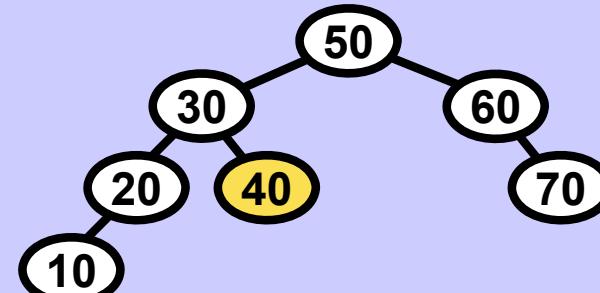
insert 70

A binary search tree with root 50. Nodes 30 (left), 60 (right), and 70 (right sibling of 60) are highlighted in yellow.

insert 10

A binary search tree with root 50. Nodes 10 (left sibling of 30), 30 (left), 60 (right), and 70 (right sibling of 60) are highlighted in yellow.

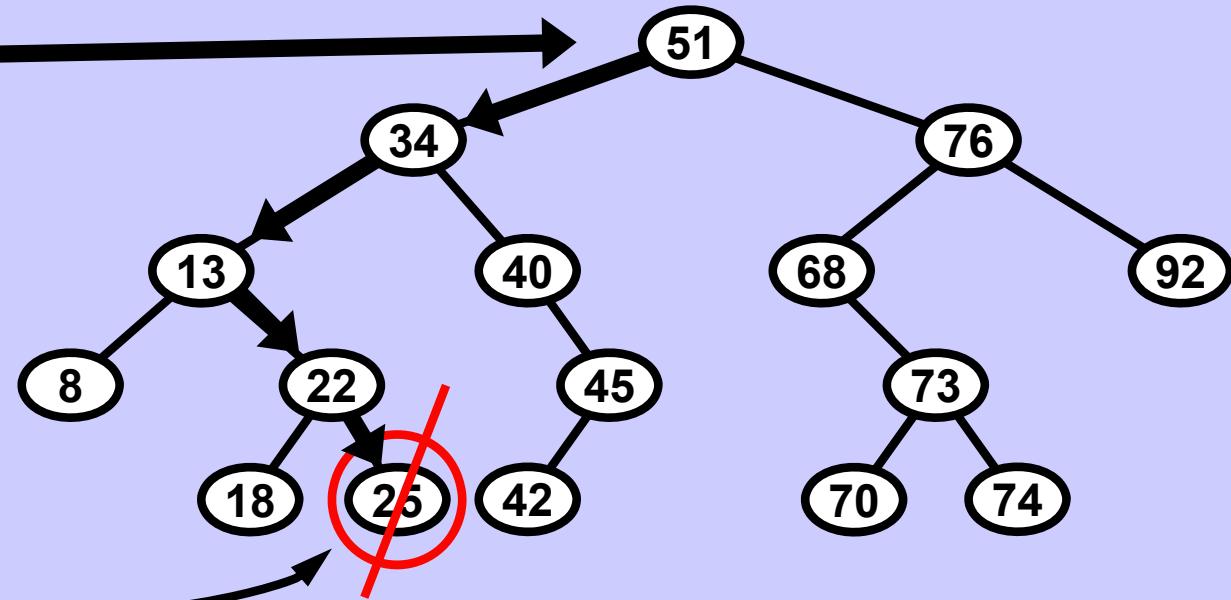
insert 40

A binary search tree with root 50. Nodes 10 (left sibling of 30), 30 (left), 40 (left child of 30), 60 (right), and 70 (right sibling of 60) are highlighted in yellow.

## Operace Delete v BVS (I.)

Smažání uzlu s 0 potomky (= listu)

Smaž 25



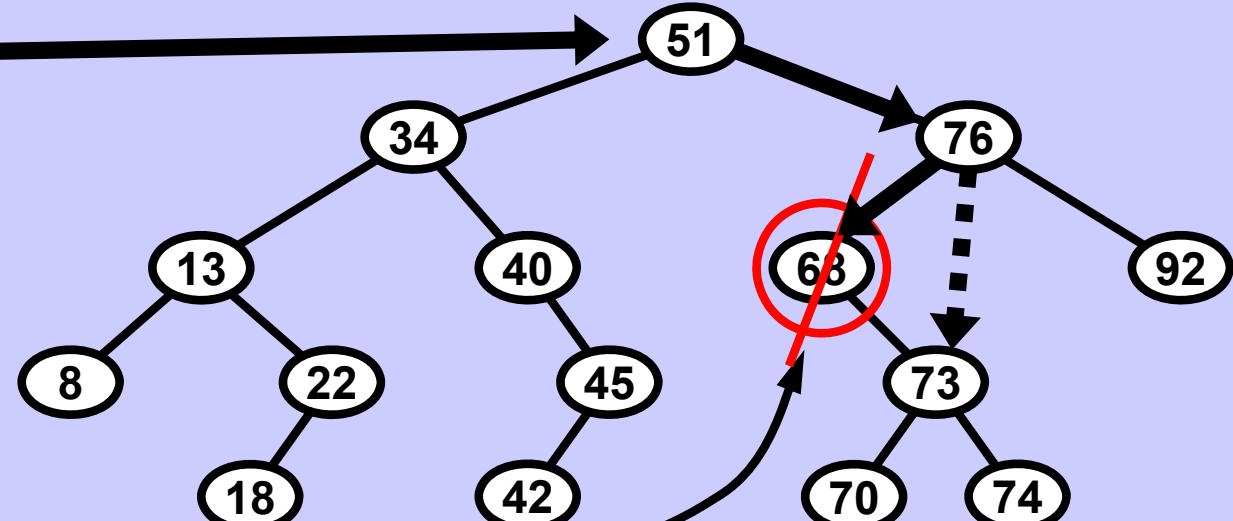
Delete I.

Najdi daný uzel (jako ve Find) a odstraň ukazatel na něj z jeho rodiče.

## Operace Delete v BVS (II.)

Smazání uzlu s 1 potomkem

Smaž 68



Odsud 25 zmizí.

Z ukazatele 76 --> 68 se stane ukazatel 76 --> 73.

Delete II.

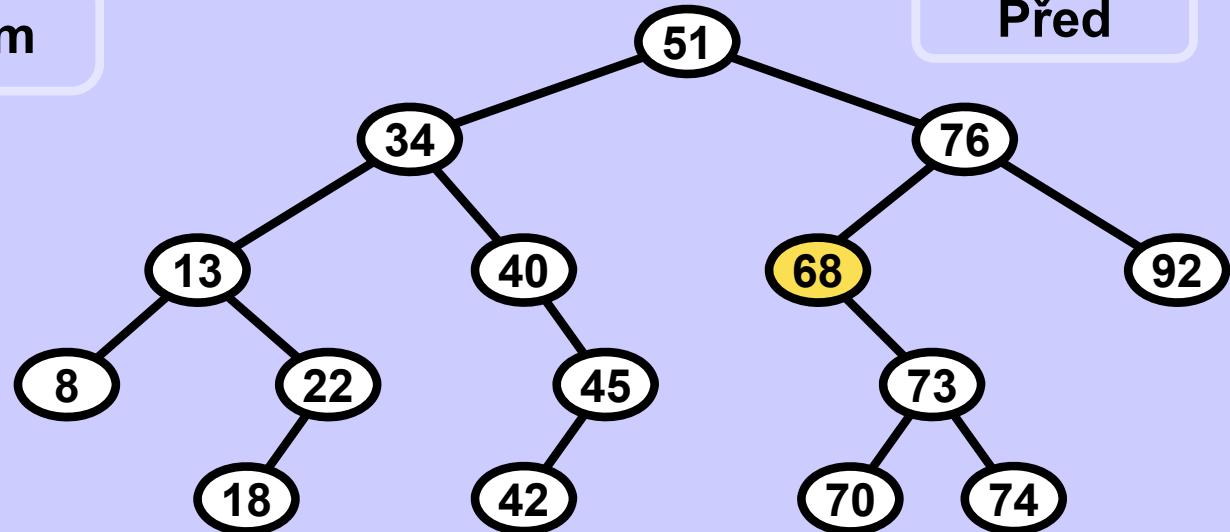
Najdi daný uzel (jako ve Find) a ukazatelem z jeho rodiče na něj ukaž na jeho (jediného!) potomka.

## Operace Delete v BVS (II.)

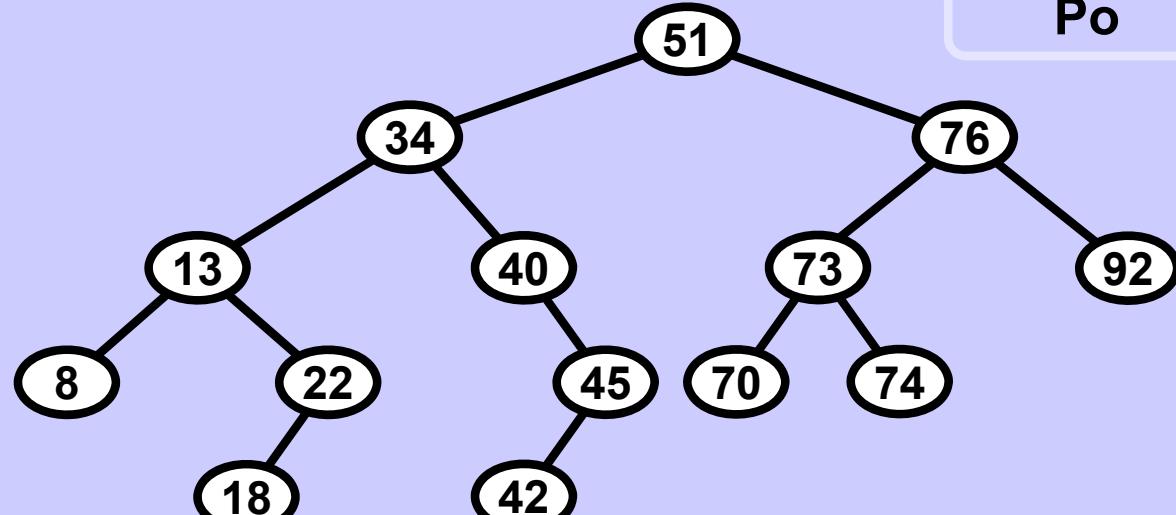
Smažání uzlu s 1 potomkem

Smaž 68

Před



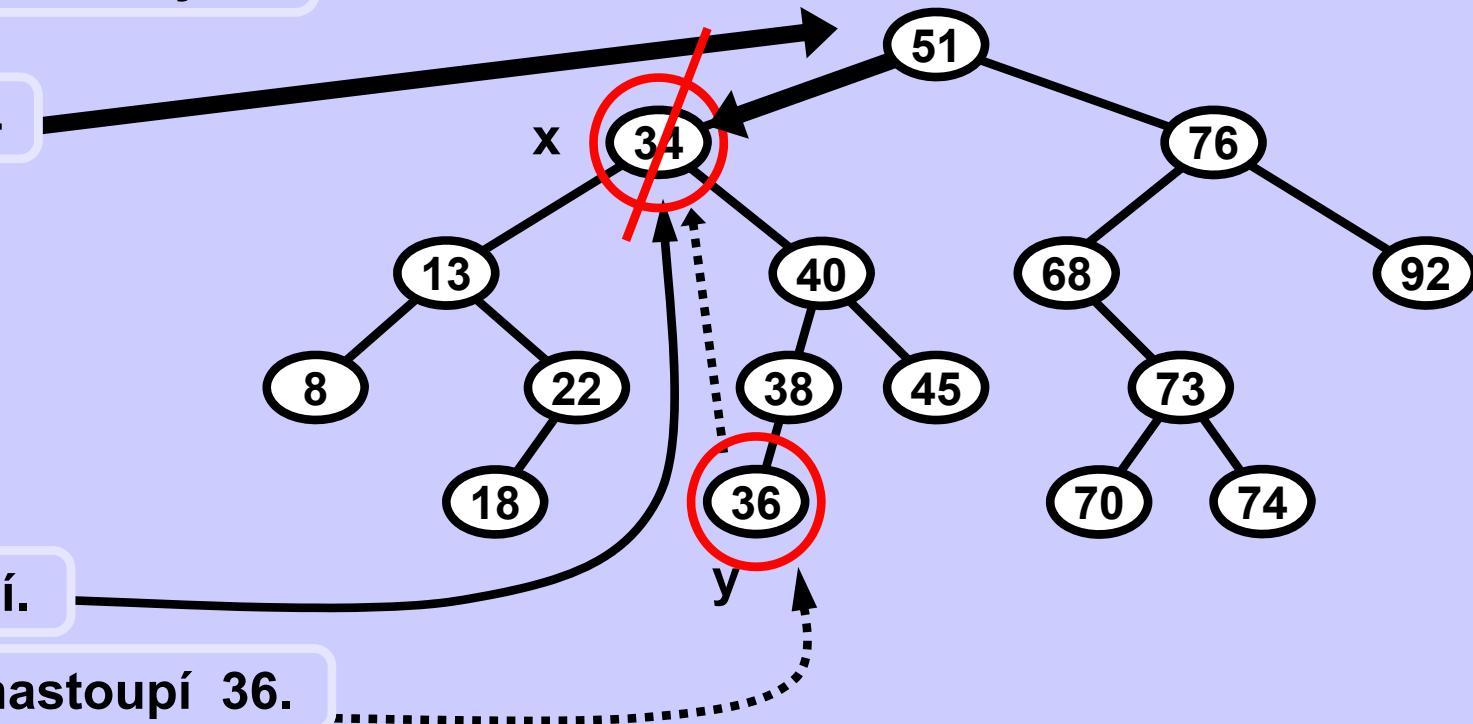
Po



## Operace Delete v BVS (IIIa.)

Smazání uzlu s 2 potomky

Smaž 34



Odsud 34 zmizí.

Na jeho místo nastoupí 36.

Delete IIIa.

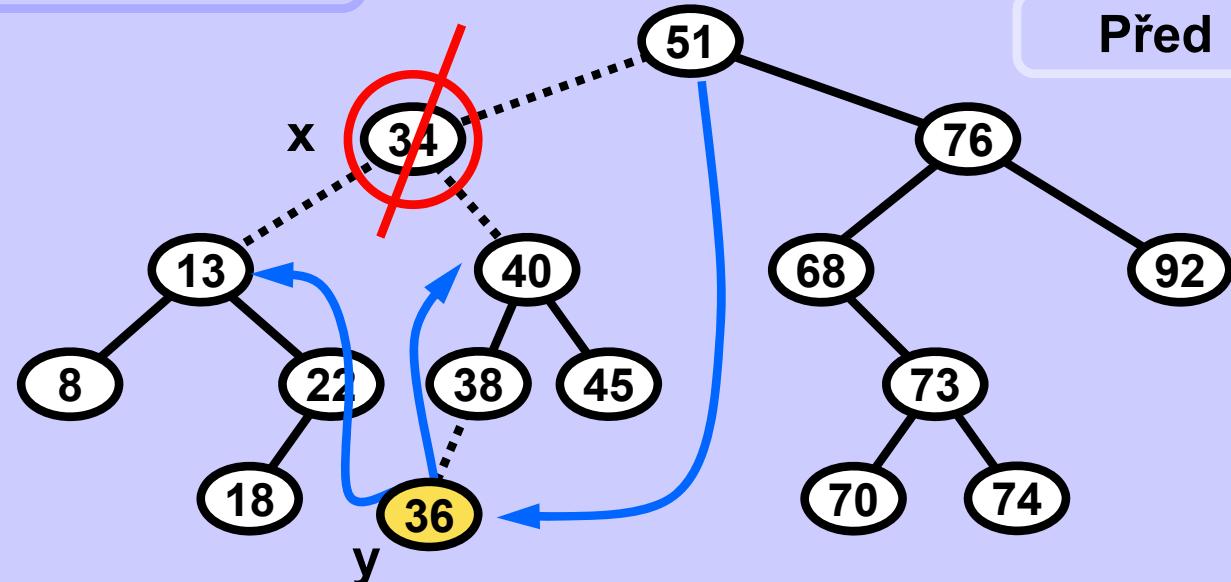
1. Najdi daný uzel x (jako ve Find) a dále najdi nejlevější (=nejmenší klíč) uzel y v pravém podstromu x.
2. Z uzlu y ukaž na potomky uzlu x, z rodiče y ukaž na potomka y místo y, z rodiče x ukaž na y.

## Operace Delete v BVS (IIIa.)

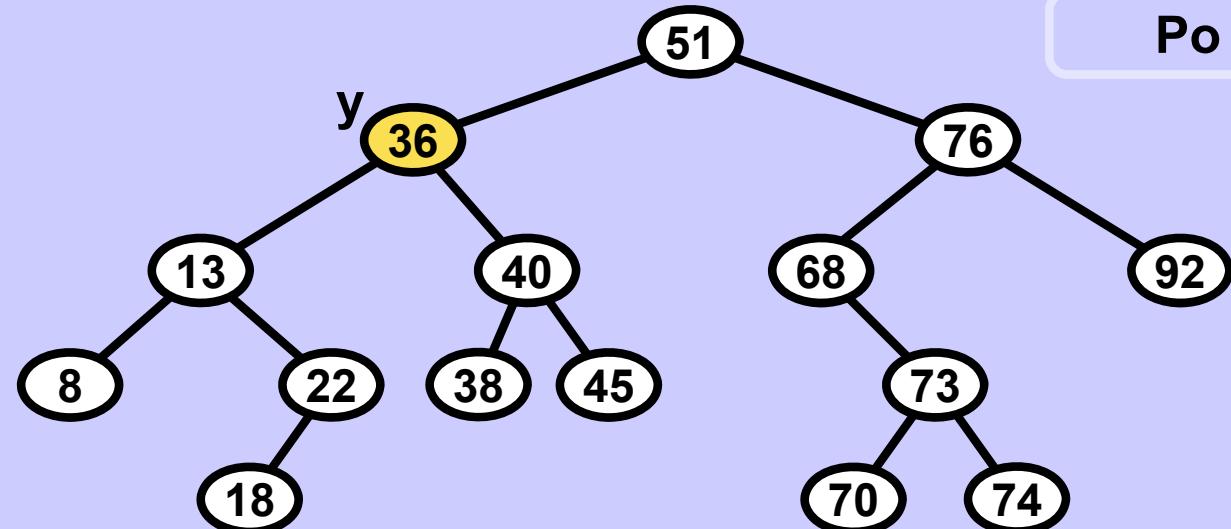
Smaž 34

zánikající  
hrany/ukazatele/reference

vznikající  
hrany/ukazatele/reference



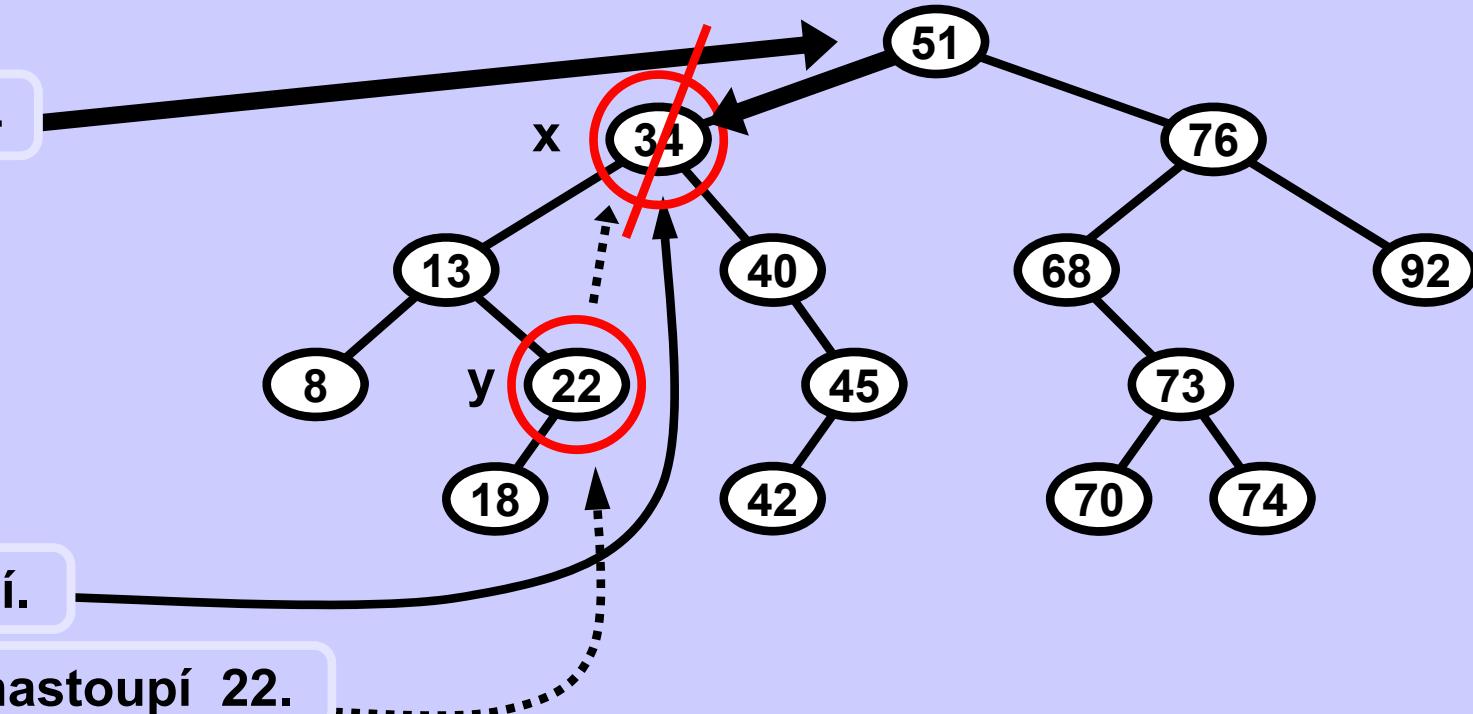
Po



Operace Delete v BVS (IIIb.) je ekvivalentní Delete IIIa.

Smazání uzlu s 2 potomky

Smaž 34



Odsud 34 zmizí.

Na jeho místo nastoupí 22.

Delete IIIb.

1. Najdi daný uzel x (jako ve Find) a dále najdi nepravější (=největší klíč) uzel y v levém podstromu x.
2. Z uzlu y ukaž na potomky uzlu x, z rodiče y ukaž na potomka y místo y, z rodiče x ukaž na y.

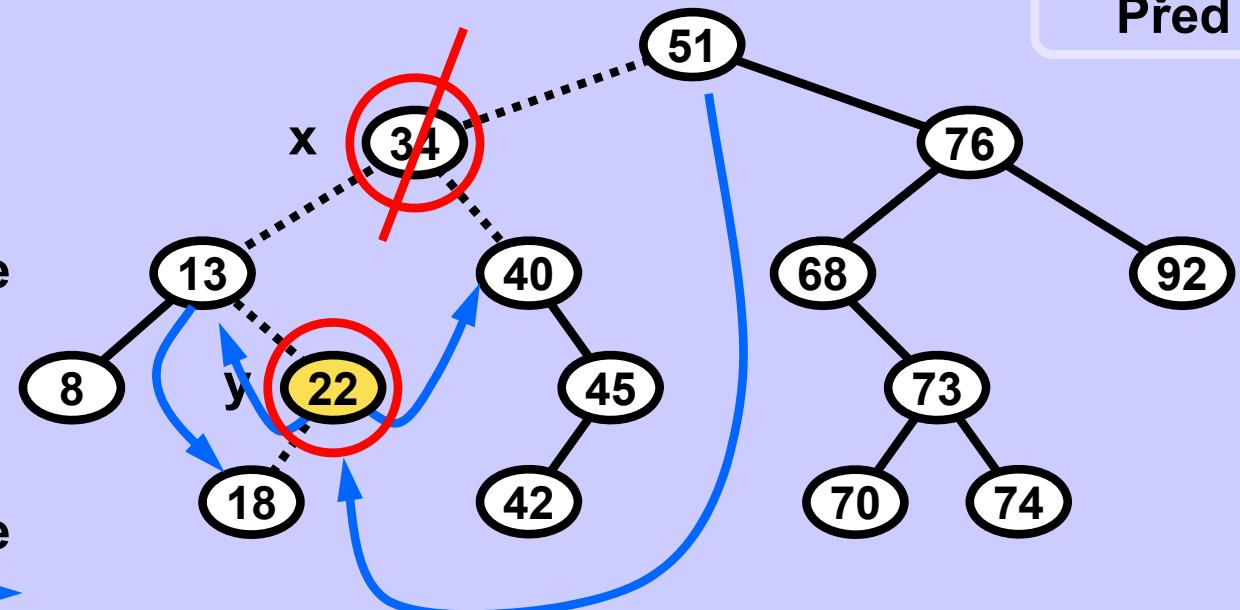
## Operace Delete v BVS (IIIb.) je ekvivalentní Delete IIIa.

Smaž 34

zanikající  
hrany/ukazatele/reference

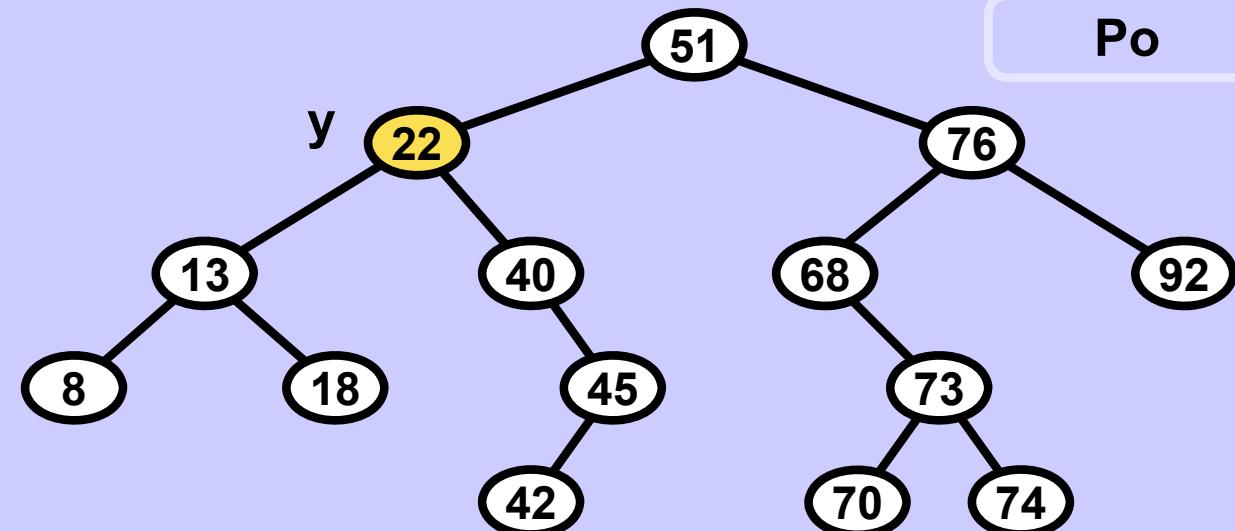
vznikající  
hrany/ukazatele/reference

Před



Je nutno ošetřit případ,  
kdy přesouvaný uzel y  
má sám následníka,  
tedy je na něj nutno  
aplikovat variantu Delete II.

Po



## Operace Delete v BVS rekurzivně

```
Node FindMin( Node root ){
    while( root->left != null ) root = root->left;
    return root;
}

Node Delete( Node root, int data ){
    if( root == null ) return root;
    else if( data < root->data )
        root->left = Delete( root->left, data );
    else if( data > root->data )
        root->right = Delete( root->right, data );
    else {
        // Case 1: No Child
        if( root->left == null && root->right == null ){
            delete root;
            root = NULL;
        }
        ...
    }
}
```

## Operace Delete v BVS rekurzivně

```
...
// Case 2: one child
} else if( root->left == null ){
    Node temp = root;
    root = root->right;
    delete temp;
} else if( root->right == null ){
    Node temp = root;
    root = root->left;
    delete temp;
// Case 3: two children
} else{
    Node temp = FindMin( root->right );
    root->data = temp->data;
    root->right = Delete( root->right, temp->data );
}
}
return root;
}
```

## Asymptotické složitosti operací Find, Insert, Delete v BVS

BVS s n uzly		
Operace	Vyvážený	Možná nevyvážený
Find	$O(\log(n))$	$O(n)$
Insert	$O(\log(n))$	$O(n)$
Delete	$O(\log(n))$	$O(n)$

# VYHLEDÁVACÍ STROMY

## AVL strom

Operace Find, Insert, Delete  
Rotace L, R, LR, RL

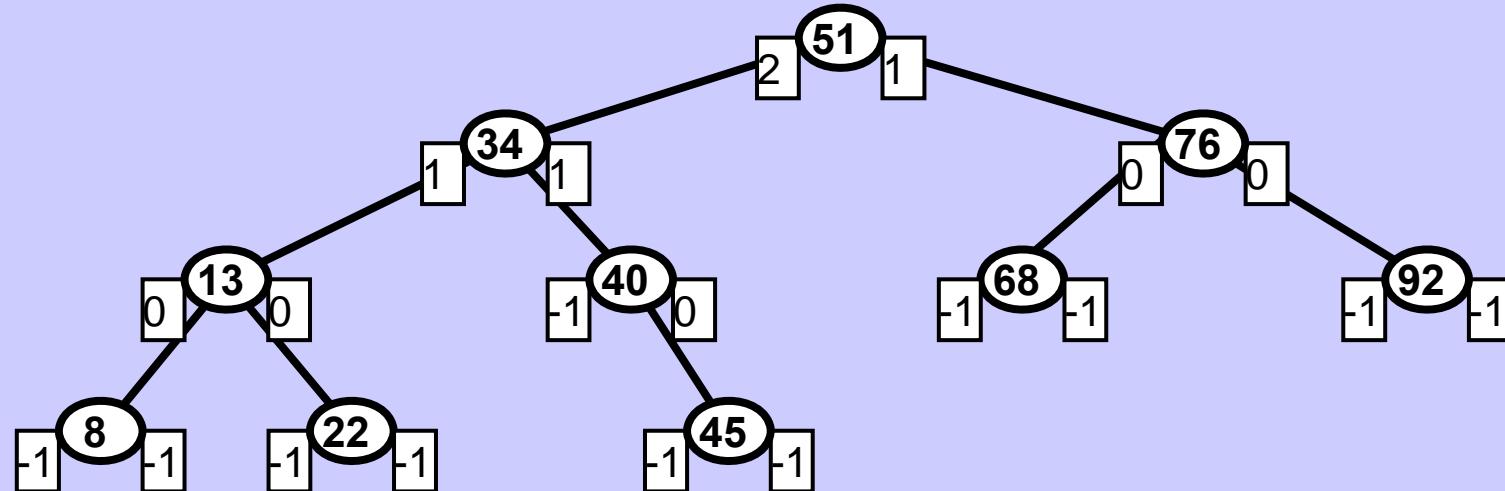
## B-strom

Operace Find, Insert, Delete

## AVL strom -- G.M. Adelson-Velskij & E.M. Landis, 1962

AVL strom je BVS s přidanými vlastnostmi, které jej udržují (téměř) vyvážený.

AVL má také operace Find, Insert, Delete.



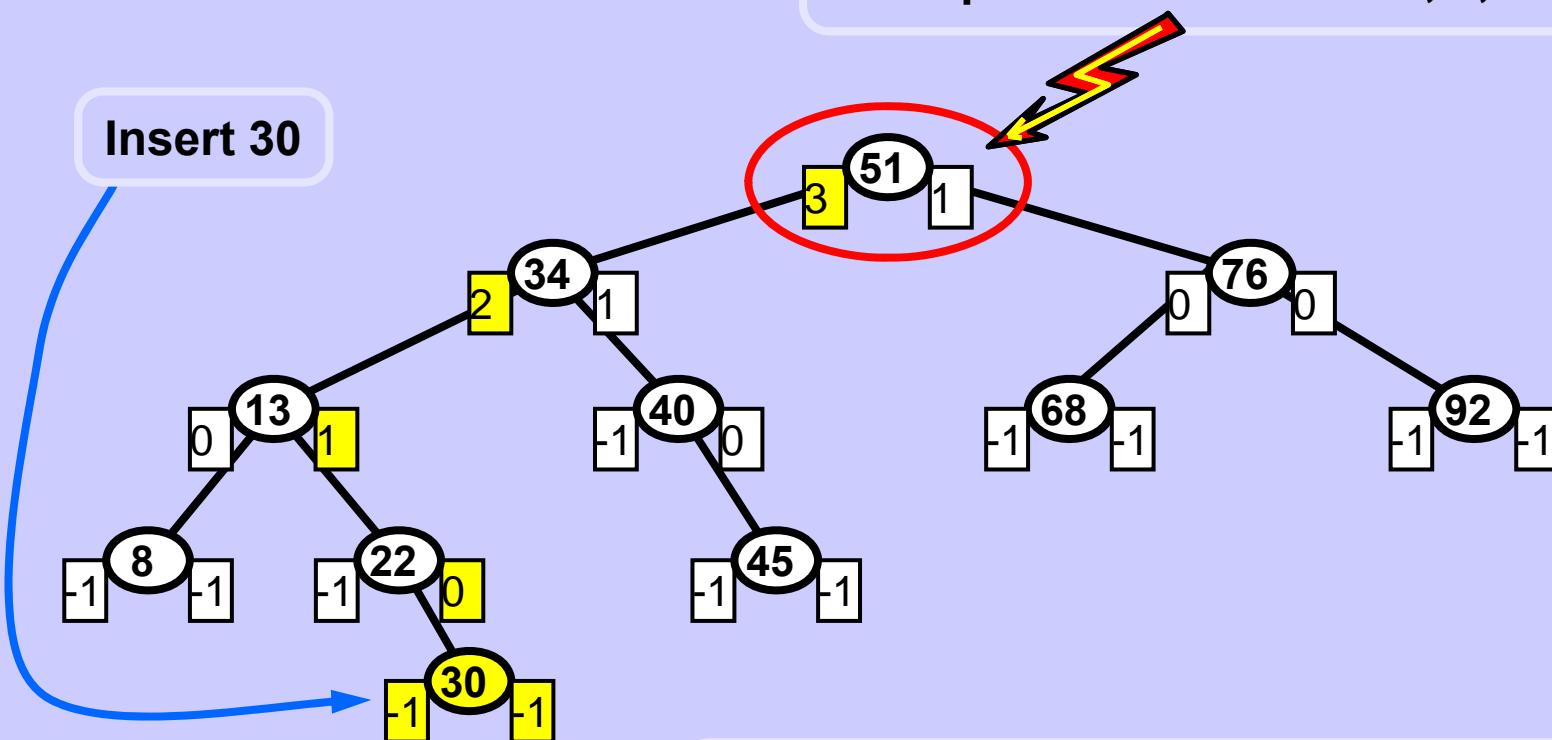
Každý uzel registruje hloubku svého levého a pravého podstromu, hloubka prázdného stromu je -1.

V každém uzlu je rozdíl výšek obou podstromů roven -1, 0, 1.

V tomto případě: Výška stromu = hloubka stromu

Vložení uzlu může způsobit rozvážení AVL stromu.

V každém uzlu má být rozdíl výšek obou podstromů roven -1, 0, 1 !!

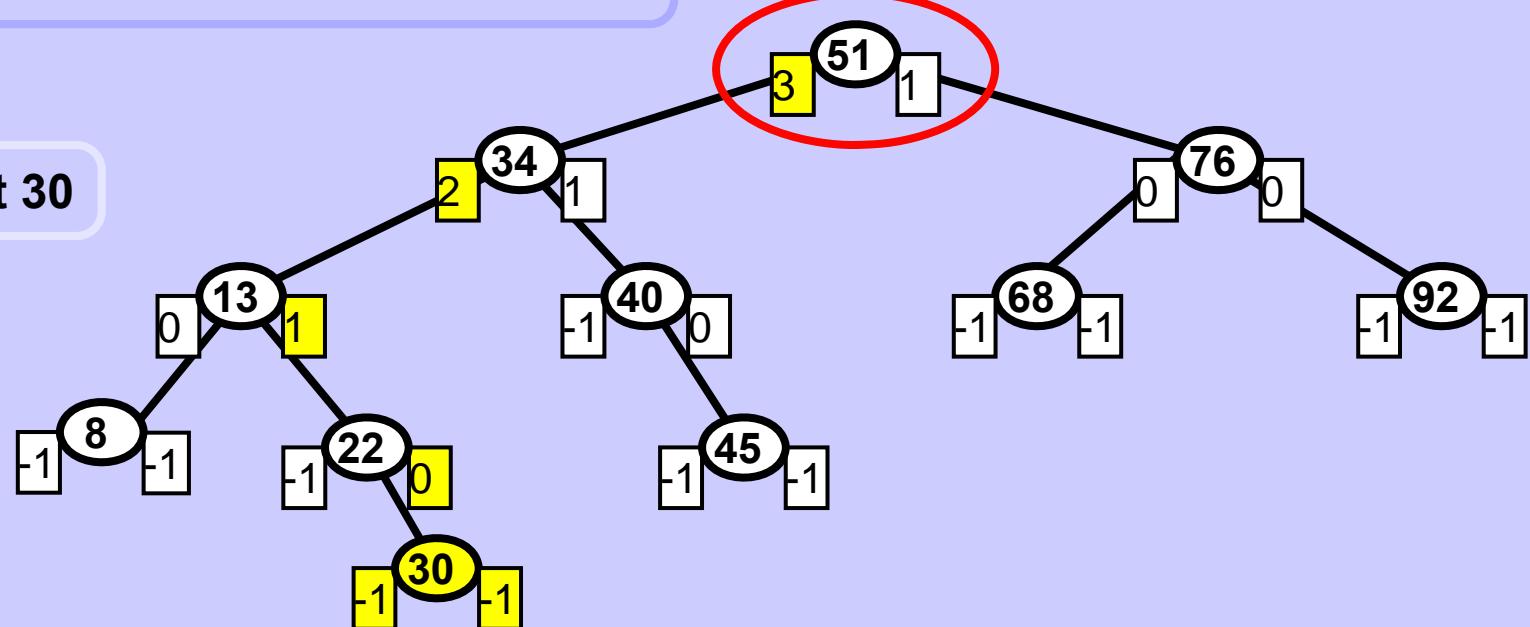


Změněné hloubky  

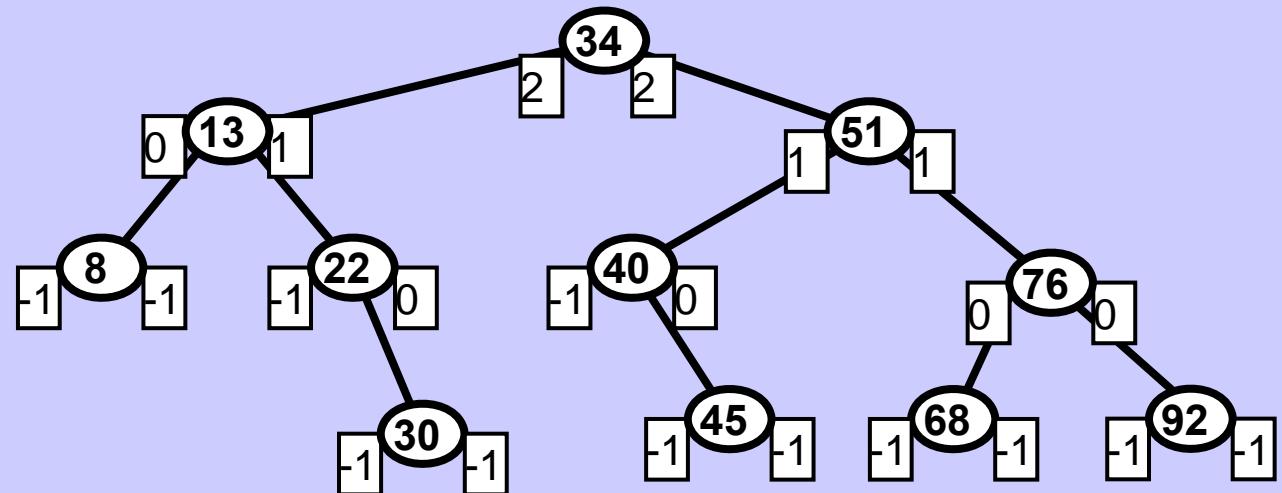
Levý podstrom uzlu 51 je příliš hluboký,  
strom přestal být AVL.

## Náprava rozvážení pomocí rotace

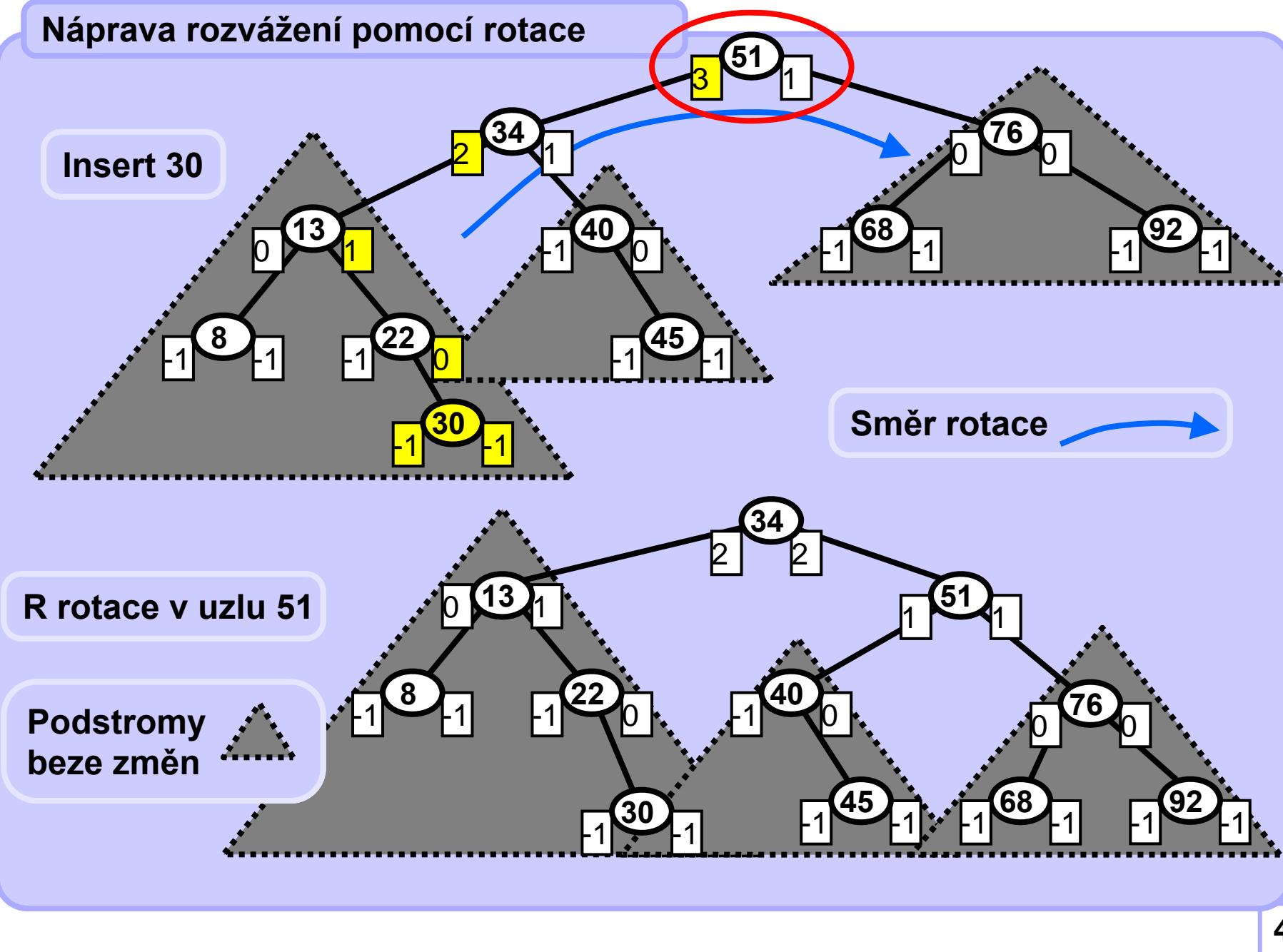
Insert 30



Vyvážený strom  
po pravé  
jednoduché rotaci,  
tzv. R rotaci

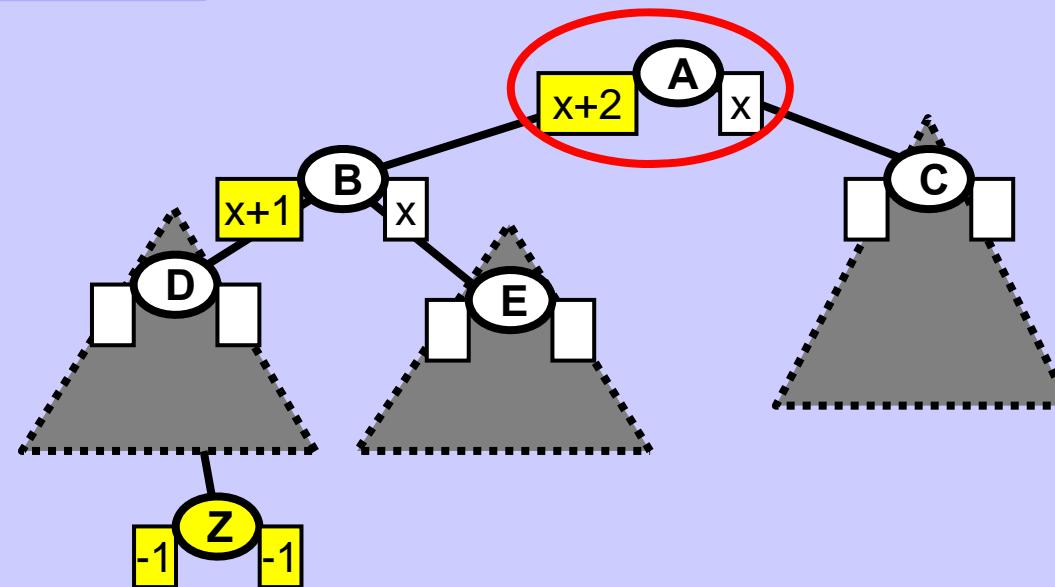


## Náprava rozvážení pomocí rotace

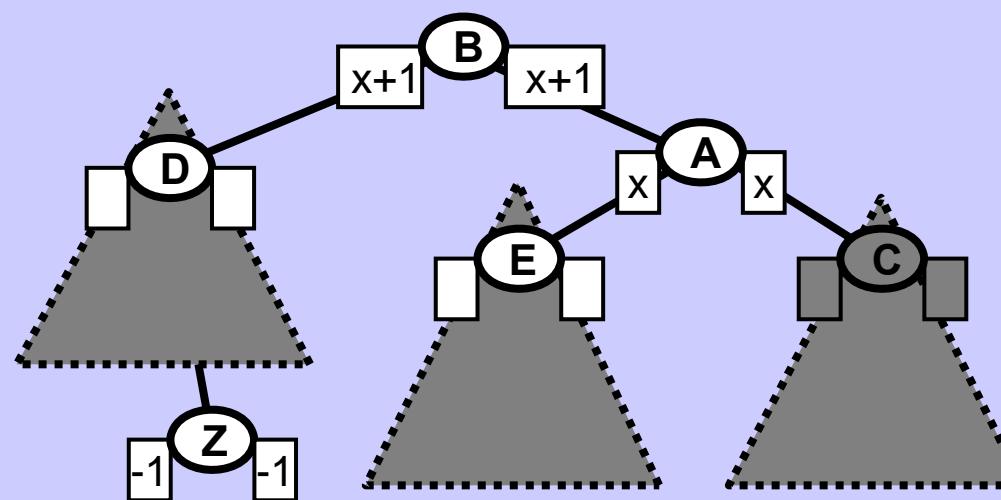


## Rotace R obecně

Před

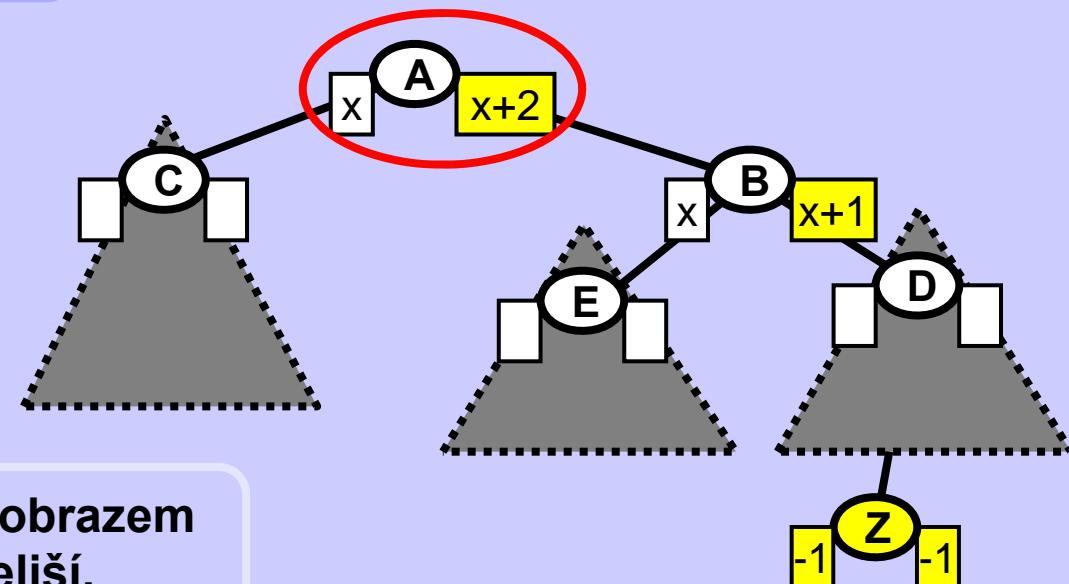


Po



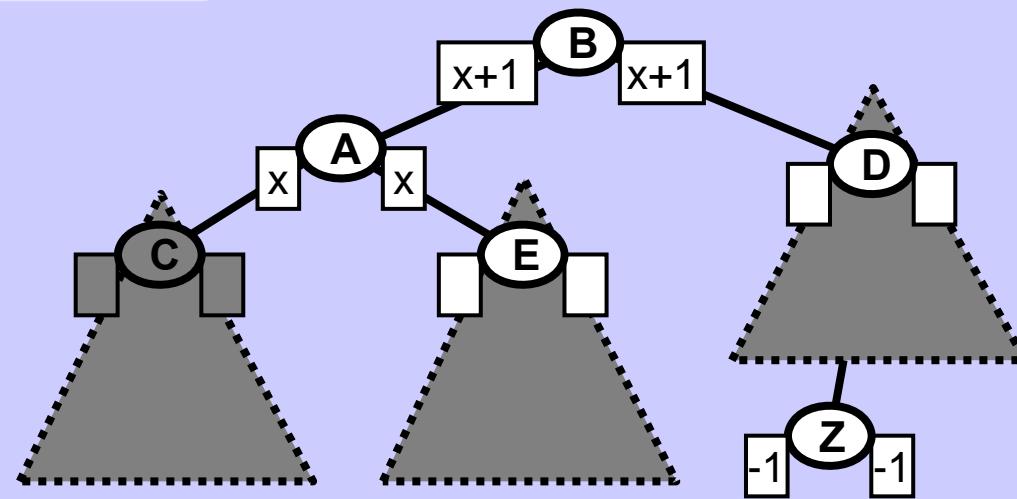
## Rotace L obecně

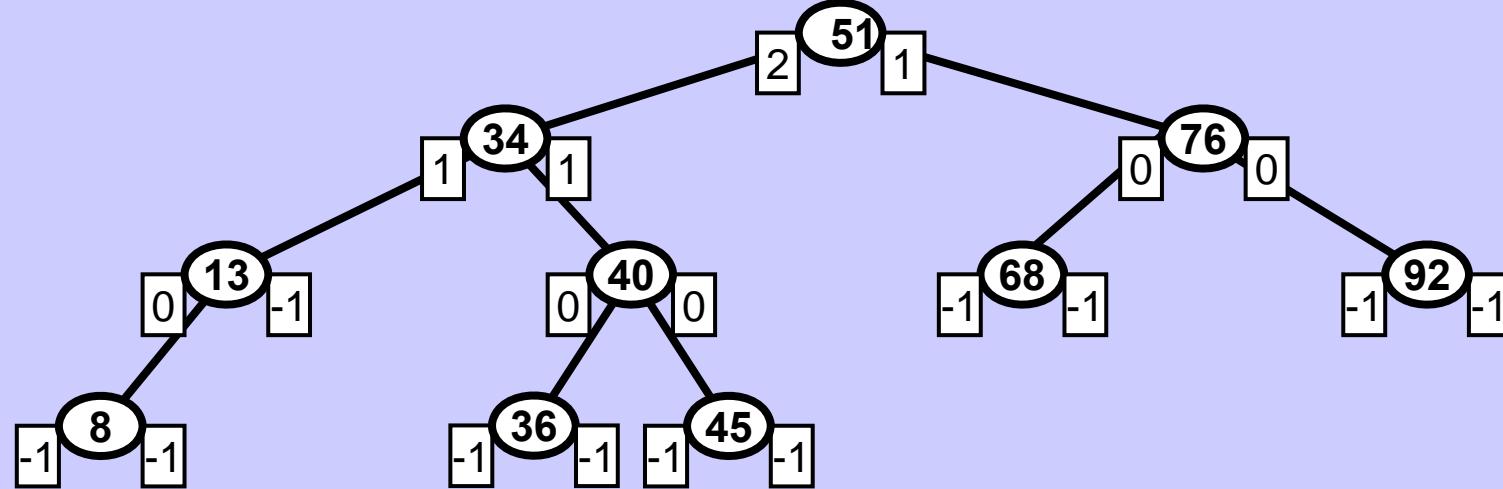
Před



**Rotace L je symetrickým obrazem  
rotace R, jinak se od ní neliší.**

Po

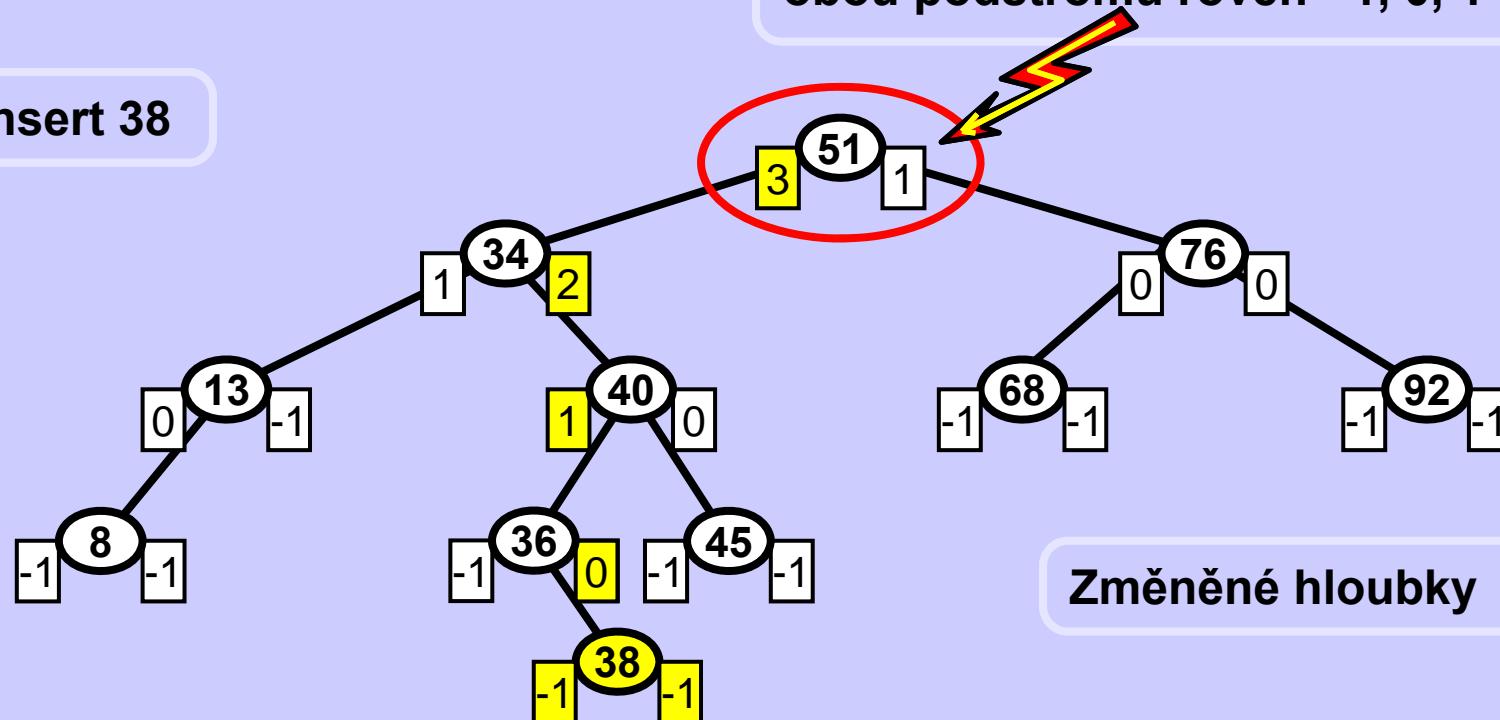


**AVL strom****Strom pro demonstraci LR rotace**

Vložení uzlu může způsobit rozvážení stromu.

V každém uzlu má být rozdíl výšek obou podstromů roven  $-1, 0, 1$  !!

Insert 38



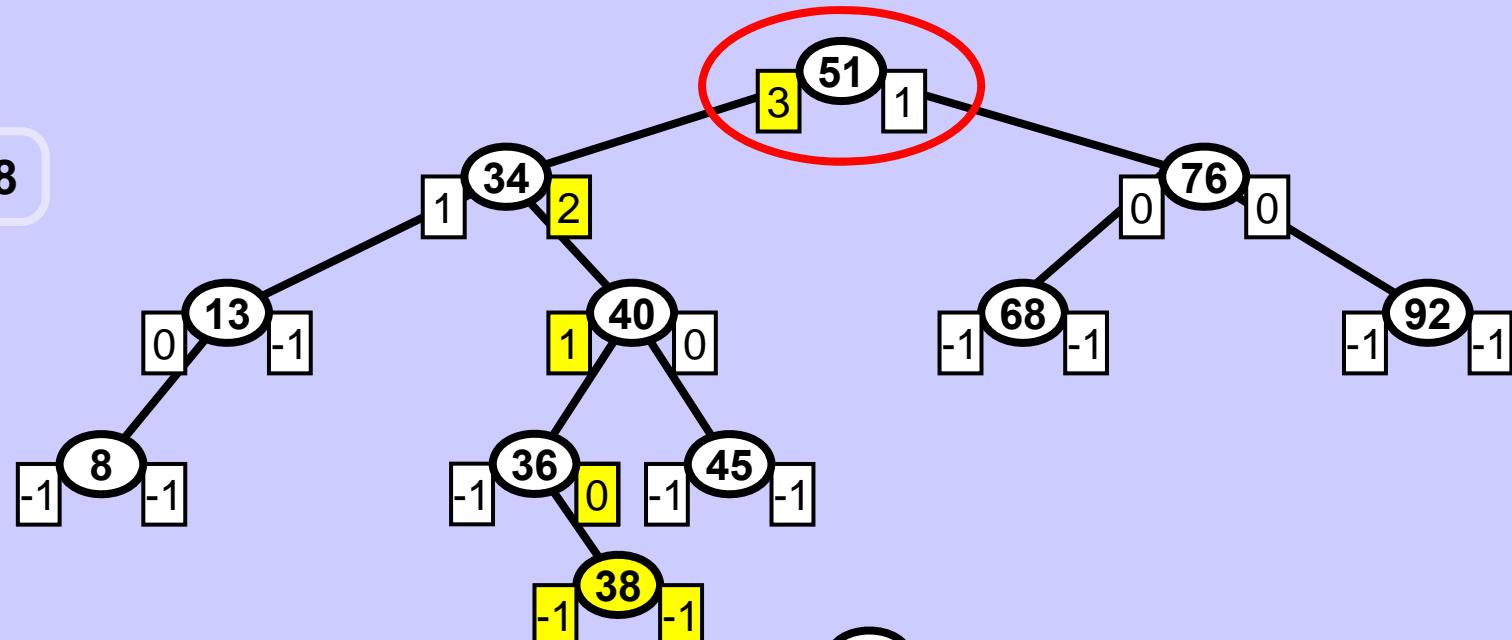
Změněné hloubky

Levý podstrom uzlu 51 je příliš hluboký, strom přestal být AVL.

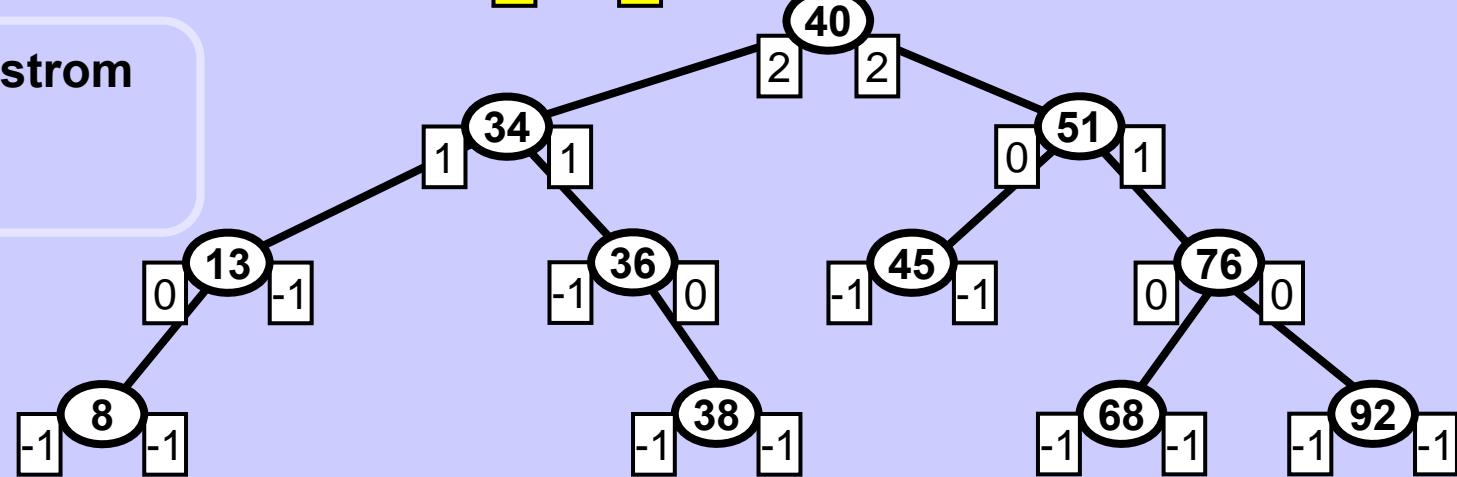
Použití rotace R by nepomohlo, příliš hlubokým by se stal původní pravý podstrom uzlu 34 díky tomu, že by se jeho hloubka vůbec nezměnila a hloubka levého podstromu 51 by klesla.

## Náprava rozvážení pomocí LRrotace

Insert 38

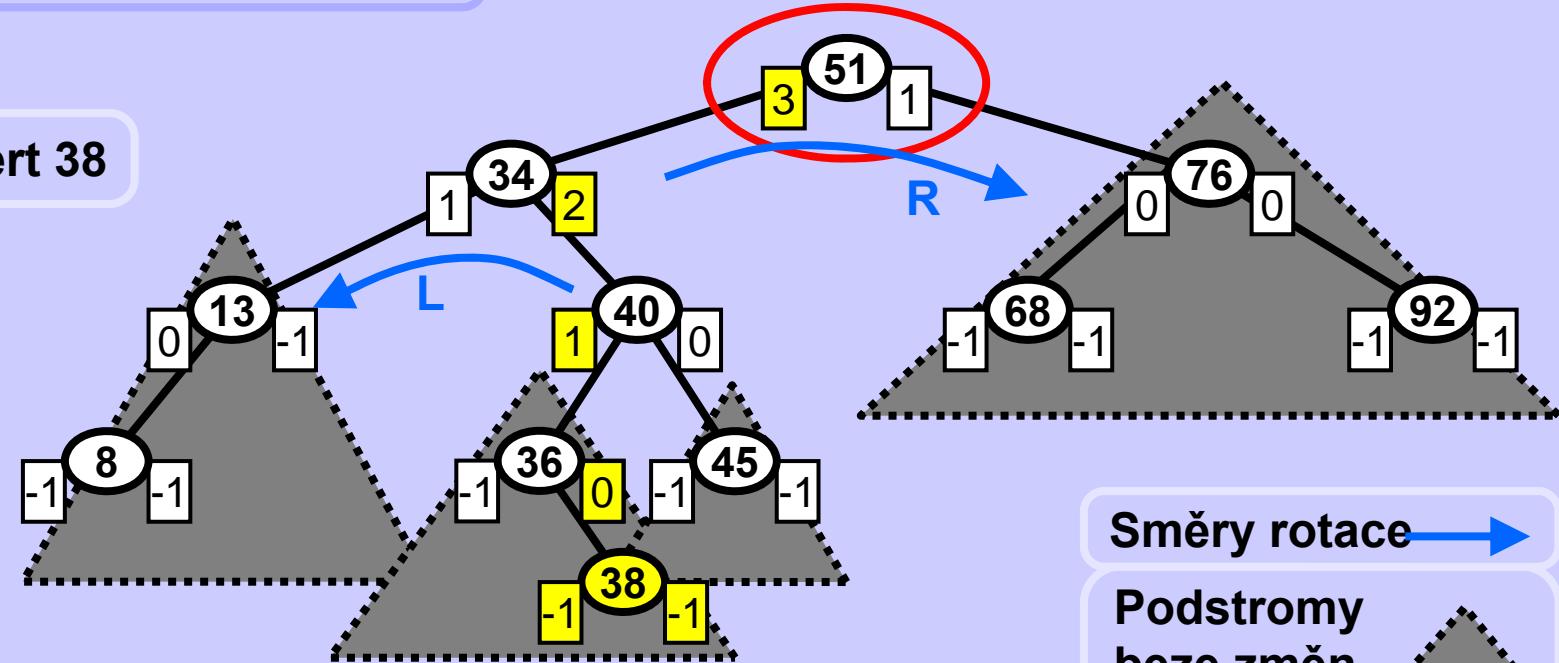


Vyvážený strom  
po dvojitě  
LR rotaci

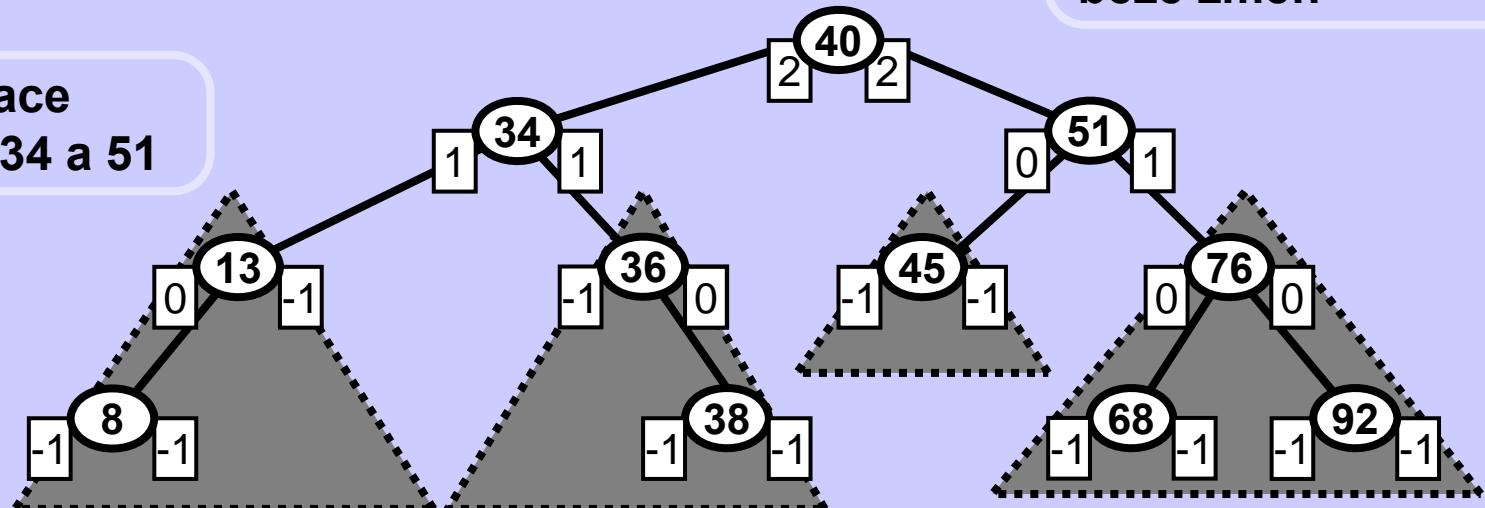


## Náprava rozvážení rotací

Insert 38

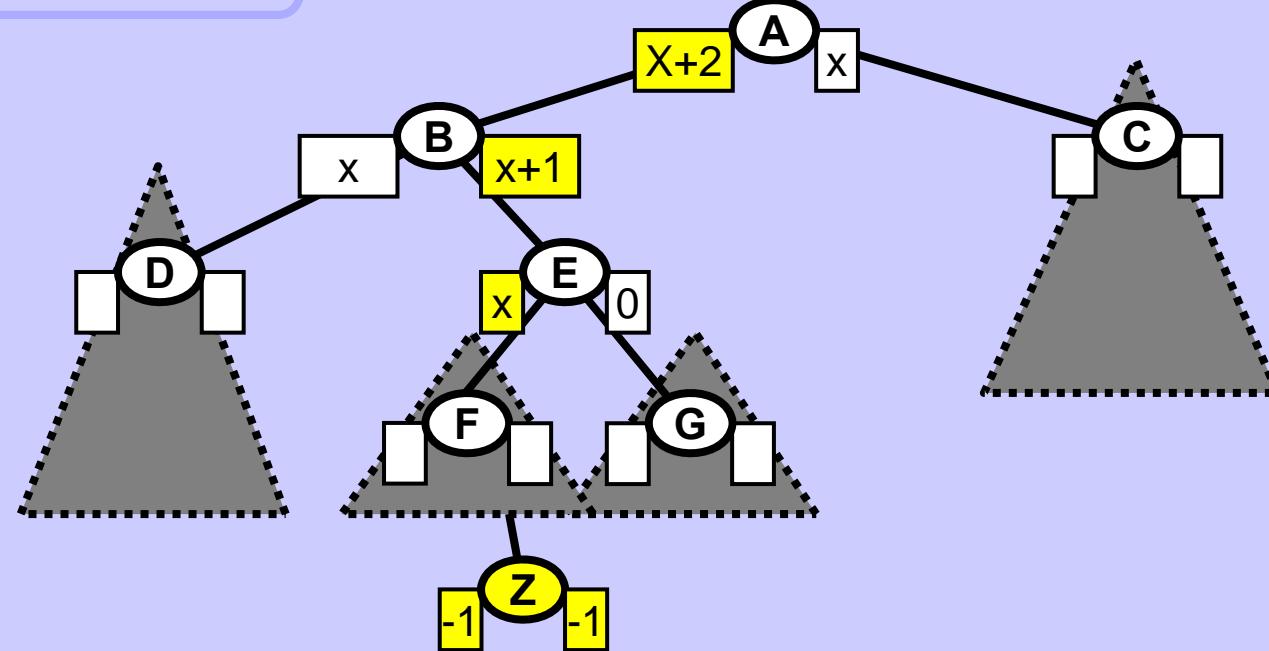


LR rotace  
v uzlu 34 a 51

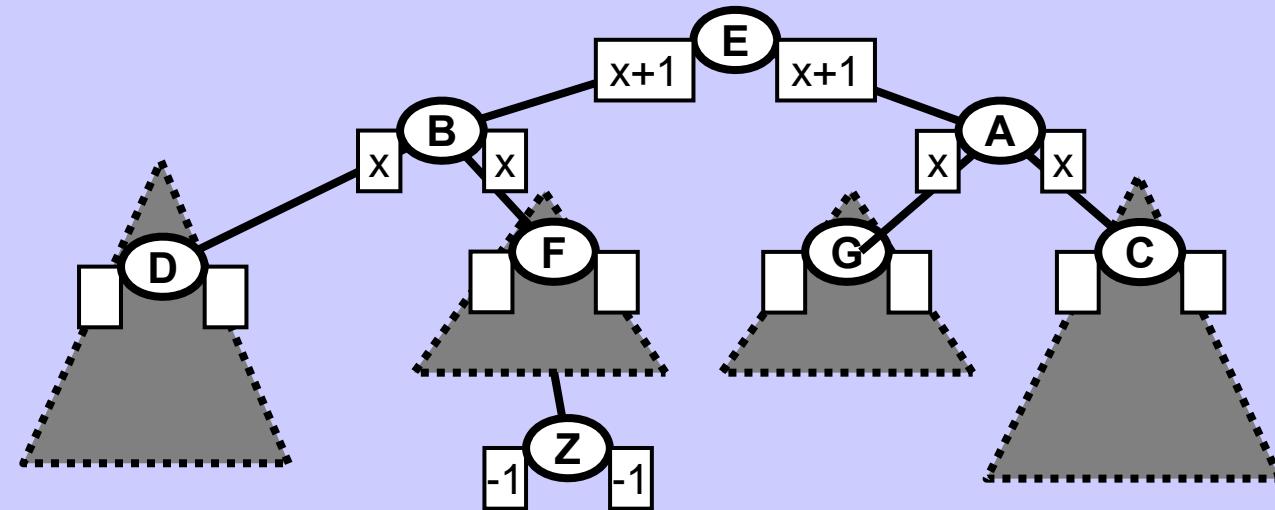


## Rotace LR obecně

Před

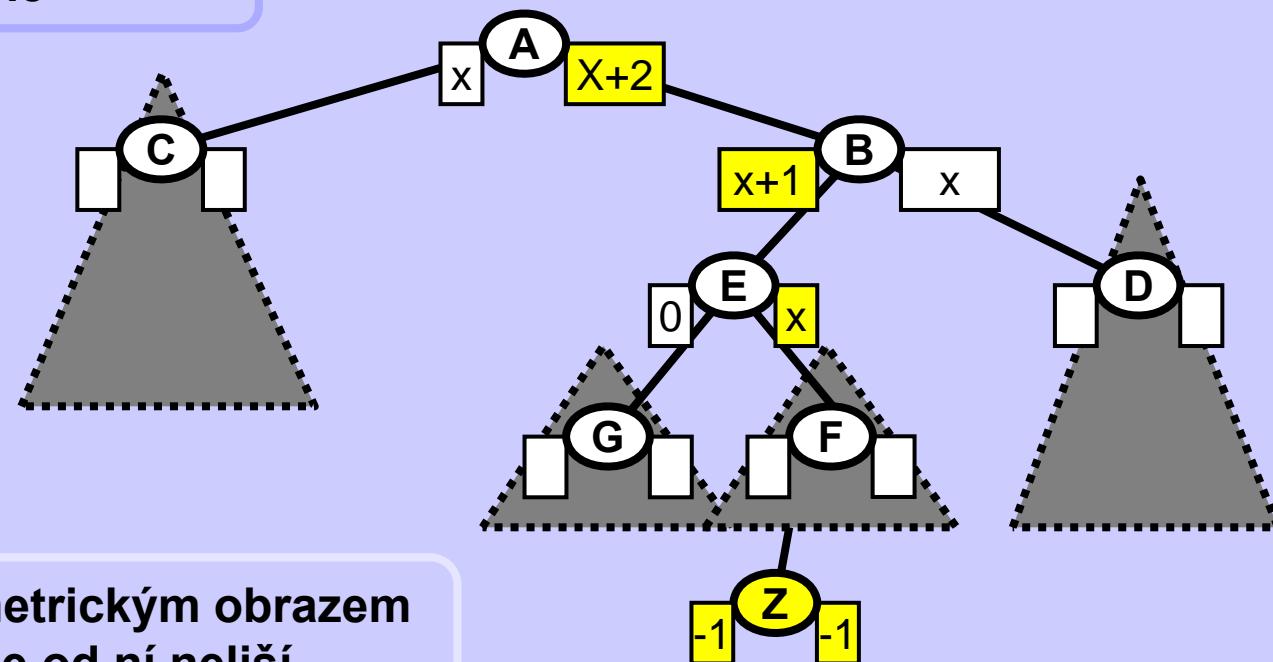


Po



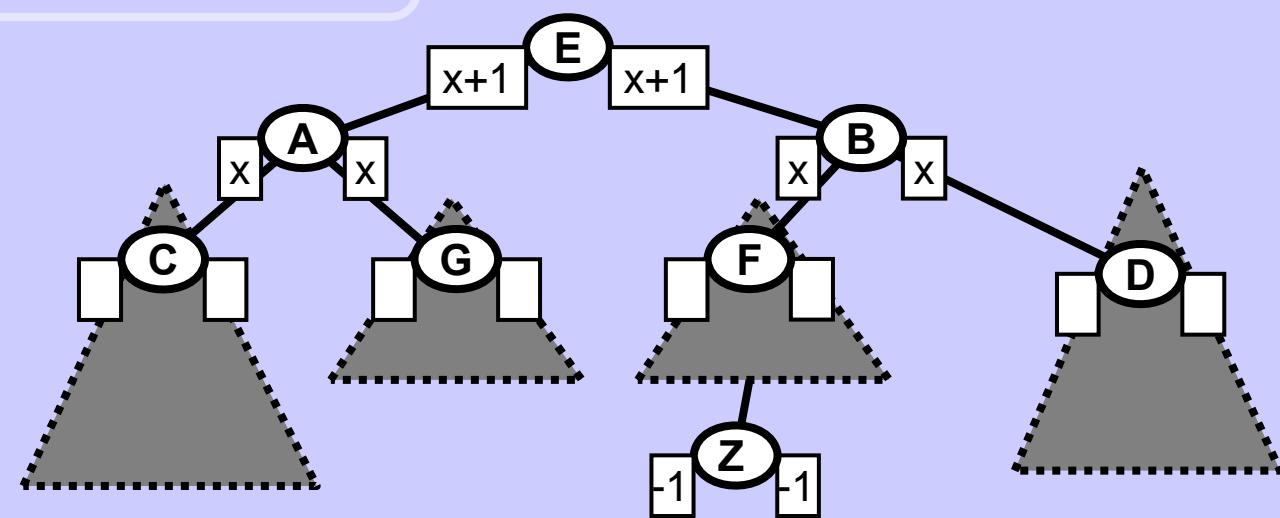
## Rotace RL obecně

Před



Rotace RL je symetrickým obrazem  
rotace LR, jinak se od ní neliší.

Po



## Pravidla pro aplikaci L, R, LR nebo RL rotací

Od přidaného uzlu postupujeme směrem ke kořeni a aktualizujeme hloubky podstromů v každém navštíveném uzlu.

Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli

- \* dvěma hranami *doprava nahoru*, provedeme v tomto uzlu **R rotaci**.
- \* dvěma hranami *doleva nahoru*, provedeme v tomto uzlu **L rotaci**.
- \* hranami *doleva a pak doprava nahoru*, provedeme v tomto uzlu **LR rotaci**.
- \* hranami *doprava a pak doleva nahoru*, provedeme v tomto uzlu **RL rotaci**.

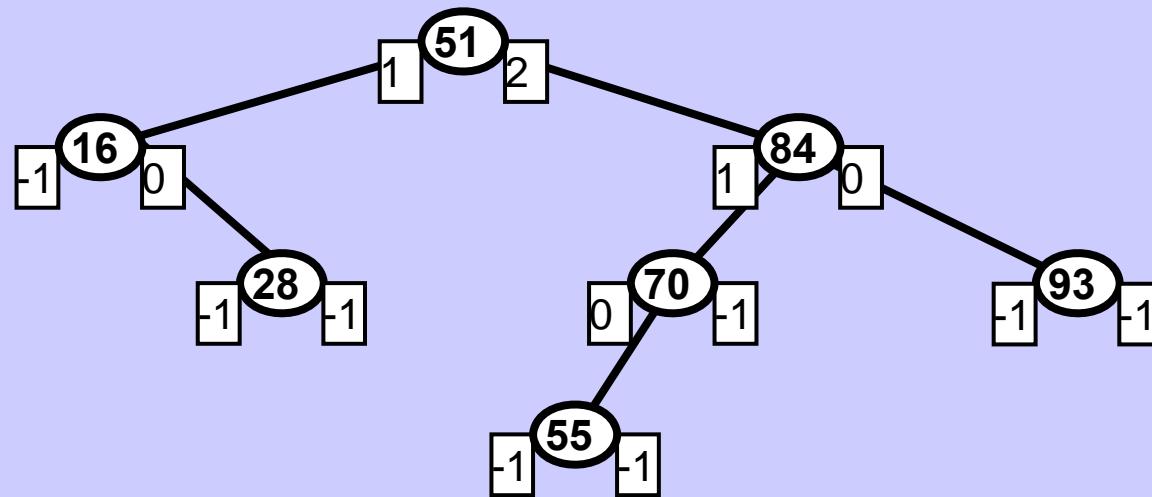
Po provedení jedné rotace po operaci Insert je AVL strom opět vyvážen.

Po provedení jedné rotace po operaci Delete (viz dále) strom vyvážen být nemusí, je nutno kontrolovat a případně aktualizovat vyvážení až ke kořeni včetně.

## Delete v AVL stromu

Strom pro demonstraci rotace po smazání uzlu

Delete 16

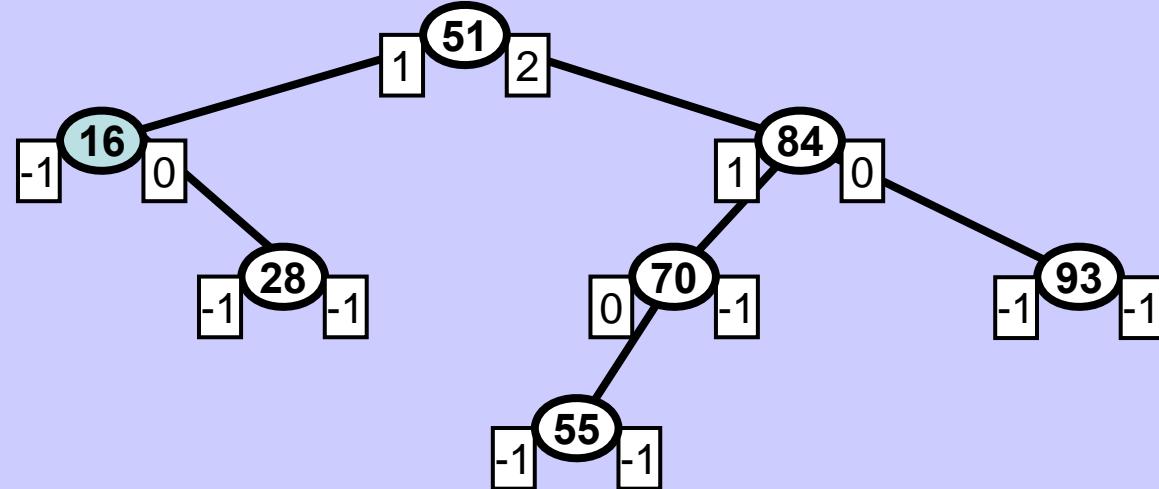


Delete proběhne standardně jako v obyčejném BVS.

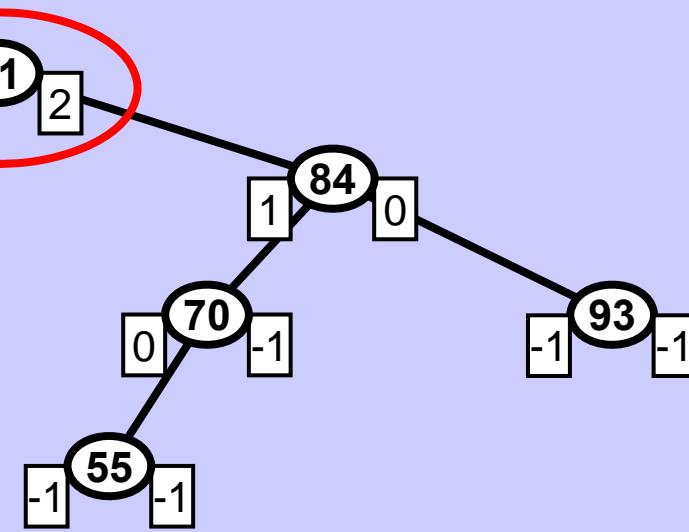
Poté postupujeme od místa smazání nahoru ke kořeni  
a aktualizujeme výšky podstromů v každém uzlu.  
Při rozvážení aplikujeme rotaci podobně jako při vkládání.

## Delete v AVL stromu

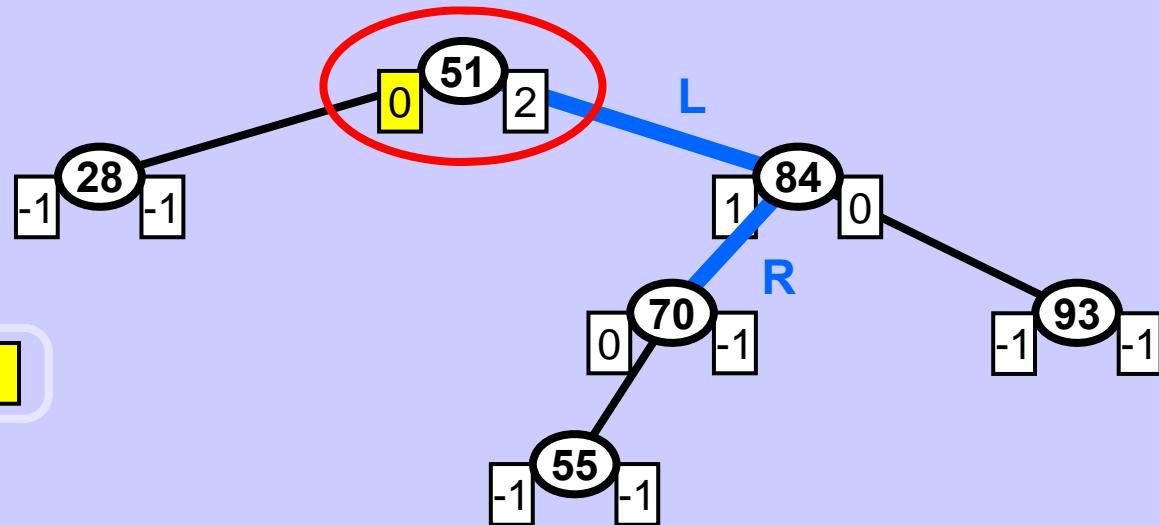
Delete 16



Změněně hloubky



## Delete v AVL stromu



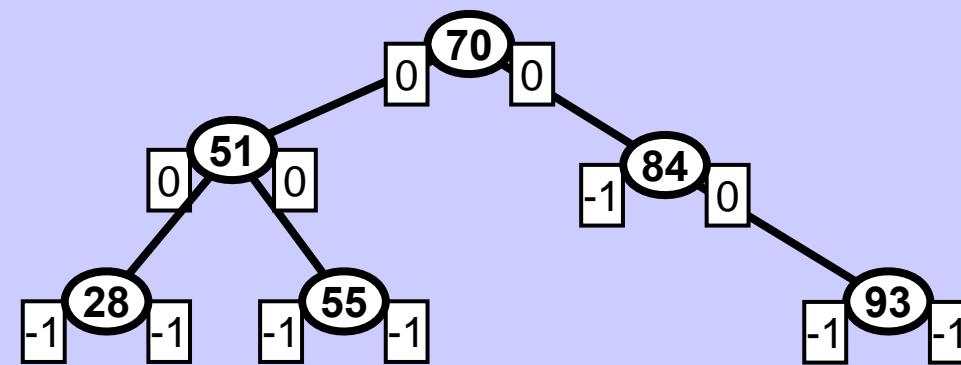
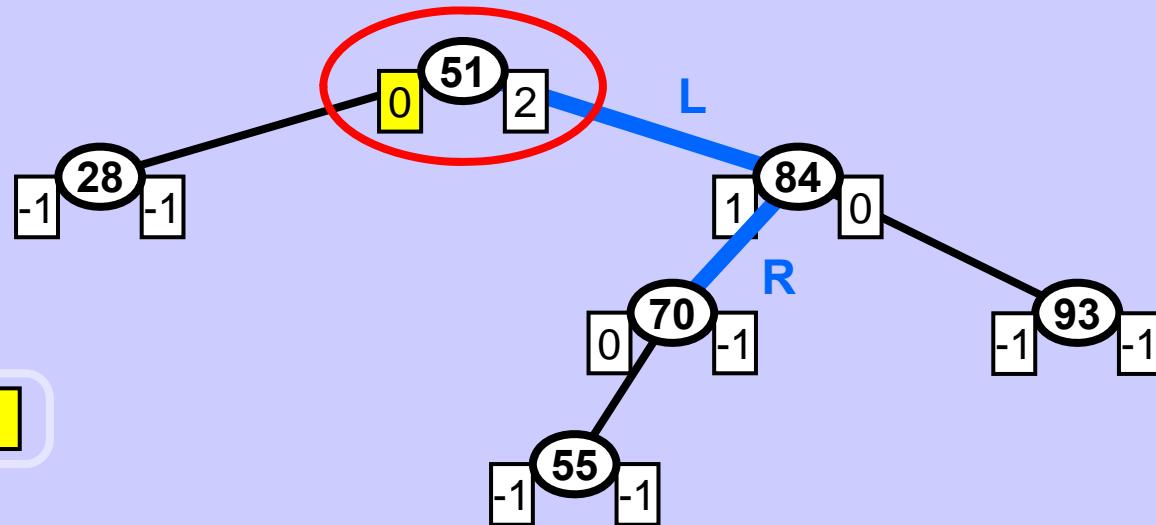
Z rozváženého uzlu 51 prozkoumáme kořen sousedního podstromu, než ze kterého jsme přišli, v tomto případě uzel 84. Má-li tento oba své podstromy stejně hluboké použijeme jednoduchou L nebo R rotaci. Má-li je různě hluboké (nejvýše se liší o 1), rozhodneme, zda použijeme L, R, LR, RL rotaci, jako kdyby rozvážení (uzel 51) vzniklo naopak přidáním uzlu do tohoto sousedního podstromu (s kořenem 84). V tomto případě použijeme RL.

## Delete v AVL stromu

Delete 16

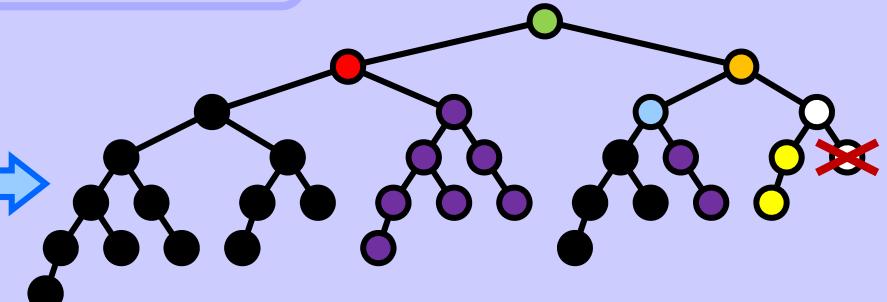
Změněné hloubky

Po rotaci RL  
v uzlu 84 a 51

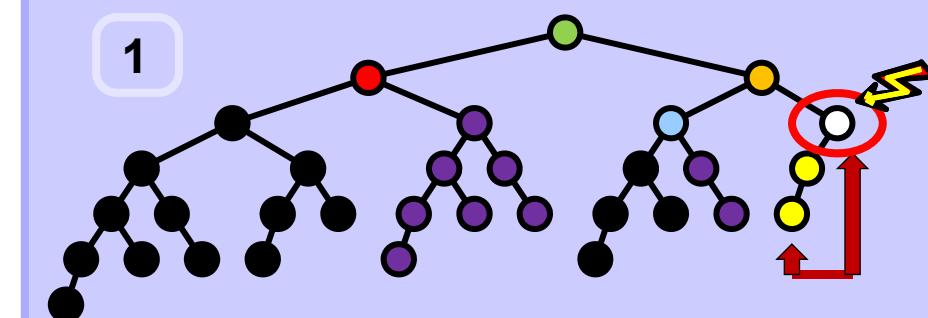


## Nutnost vícenásobných rotací po operaci Delete.

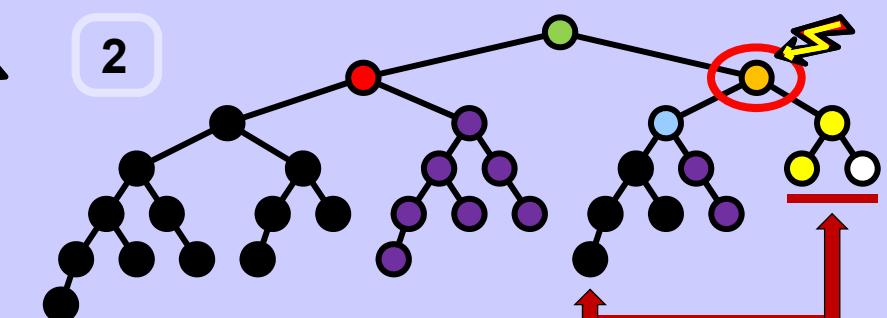
Příklad. Před operací Delete je AVL strom vyvážený.



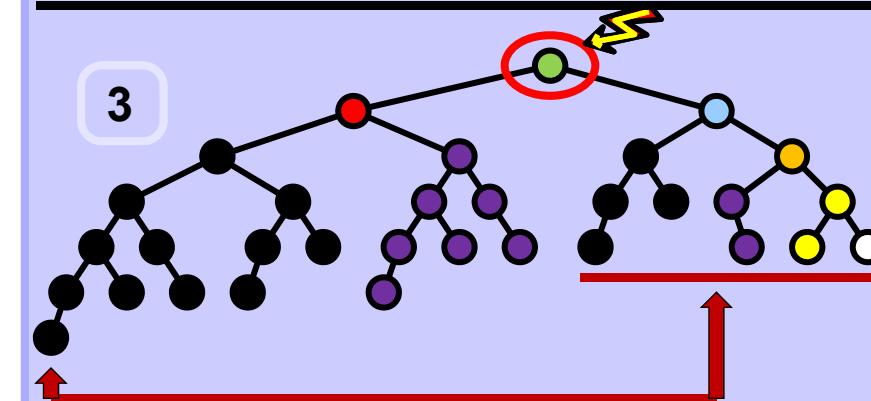
1



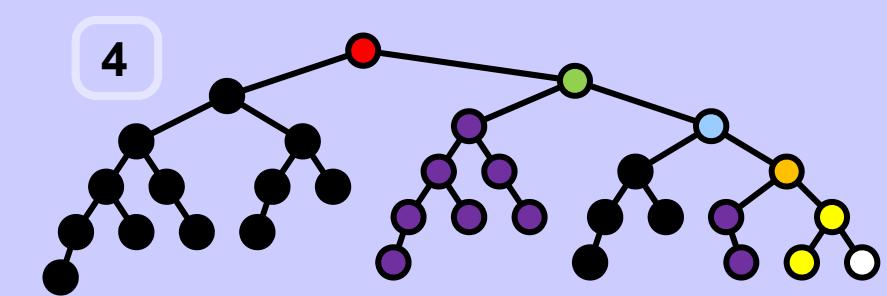
2



3



4



Vyváženo.

## Implementace operací v AVL stromu

...

// homework...

## Asymptotické složitosti operací Find, Insert, Delete v BVS a AVL

	BVS s n uzly		AVL strom s n uzly
Operace	Vyvážený	Možná nevyvážený	Vyvážený
Find	$O(\log(n))$	$O(n)$	$O(\log(n))$
Insert	$\Theta(\log(n))$	$O(n)$	$\Theta(\log(n))$
Delete	$O(\log(n))$	$O(n)$	$\Theta(\log(n))$

## B-strom -- Rudolf Bayer, Edward M. McCreight, 1972

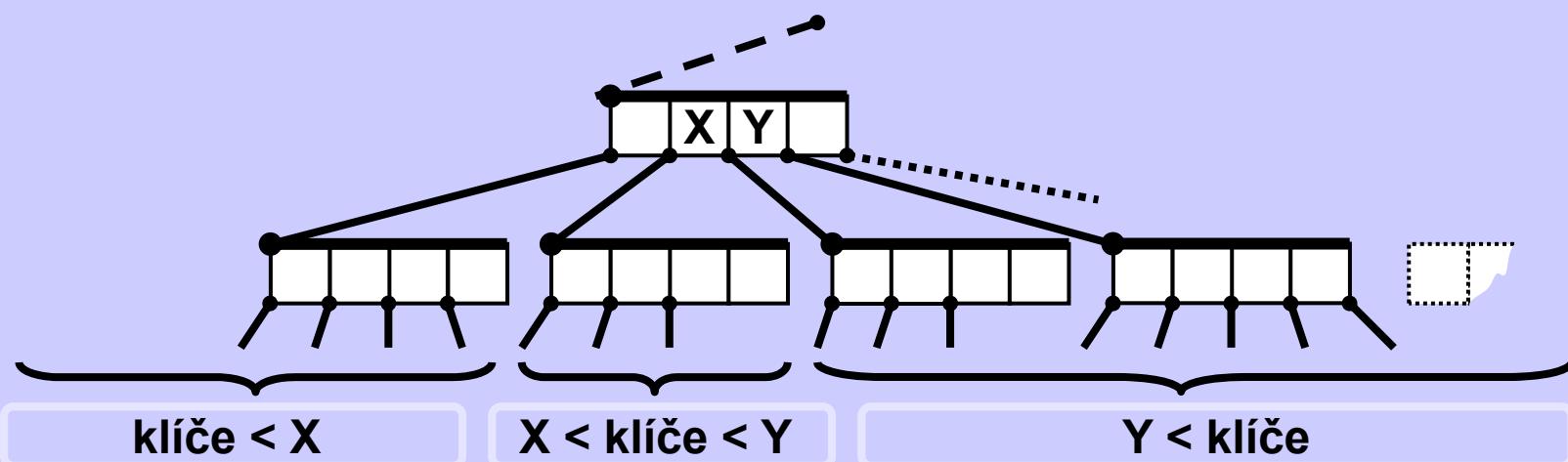
Všechny cesty z kořene do listu jsou stejně dlouhé  
tj. B-strom je ideálně vyvážený.

Klíče jsou v uzlu seřazенé.

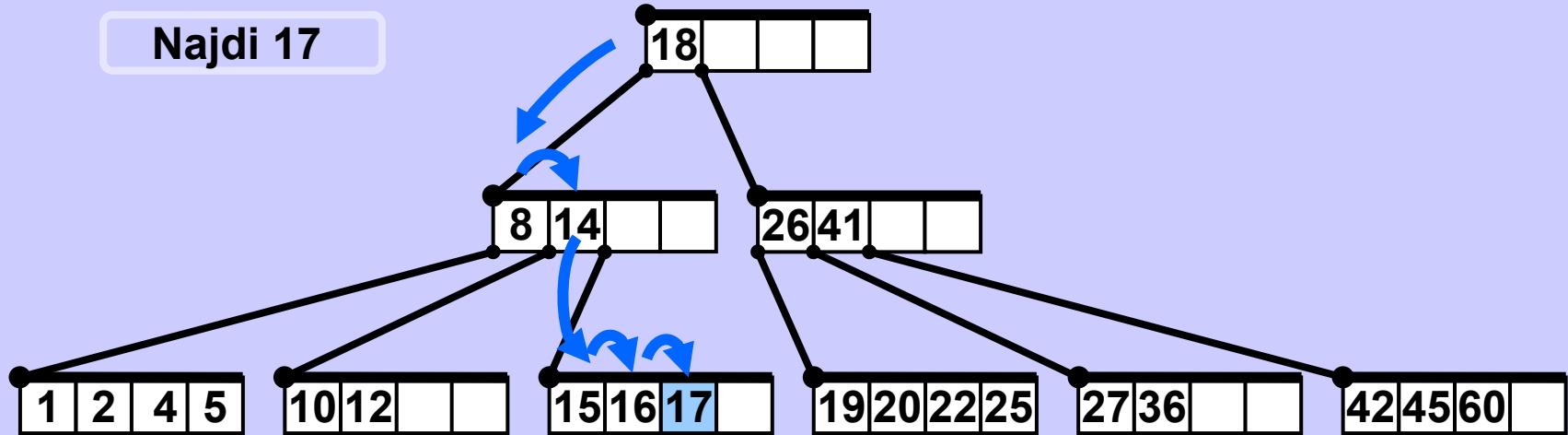
Fixní  $k > 1$  pro celý strom určuje velikost všech uzelů.

Každý uzel kromě kořene má nejméně  $k$  a nejvýše  $2k$  klíčů,  
každý vnitřní uzel tedy má nejméně  $k+1$  a nejvýše  $2k+1$  potomků.

Kořen může mít libovolný počet klíčů. Není-li zároveň listem,  
má alespoň 2 potomky.



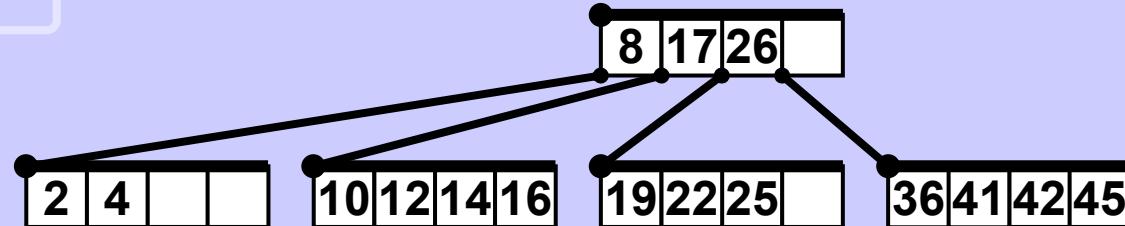
## B-strom -- Find



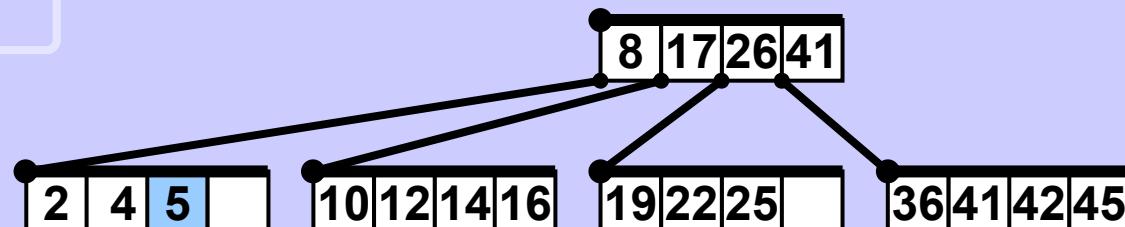
V uzlu se vyhledává sekvenčně  
 (lze také využít půlení intervalu apod).  
 Pokud uzel není listem a klíč v něm není,  
 hledání pokračuje v odpovídajícím potomku.  
 Pokud uzel je listem a klíč v něm není, nenalezeno.

## B-strom -- Insert

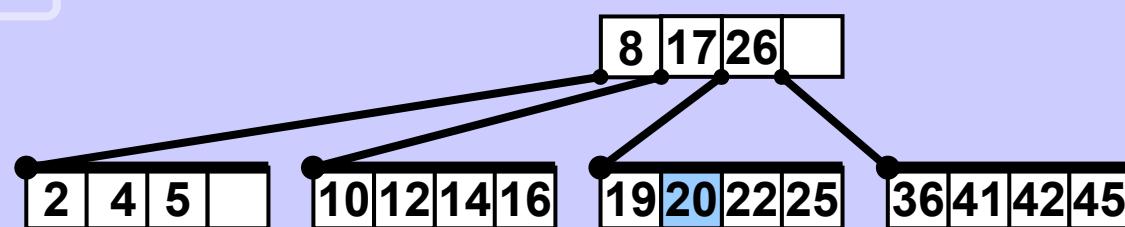
B-strom



Vlož 5

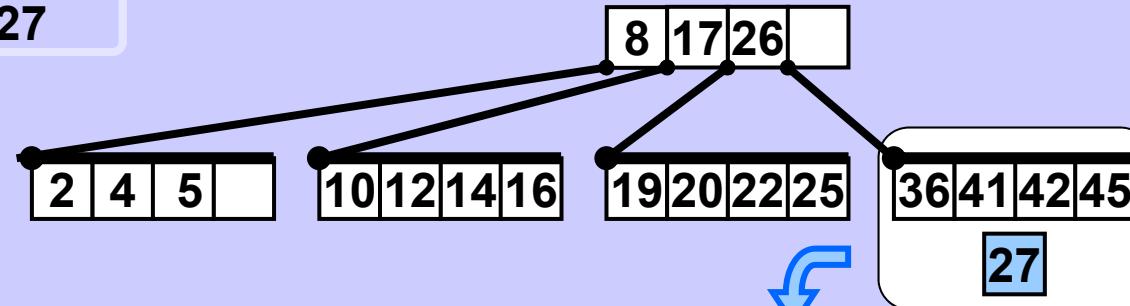


Vlož 20



## B-strom -- Insert

Vlož 27

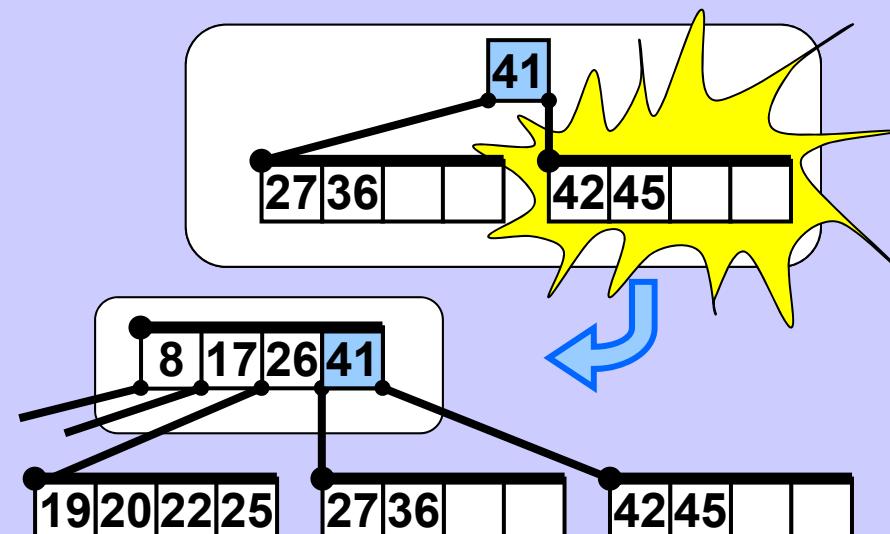


Seřad' mimo strom.

Vyber medián,  
vyvoř nový uzel,  
přesuň do něj hodnoty  
větší než medián.

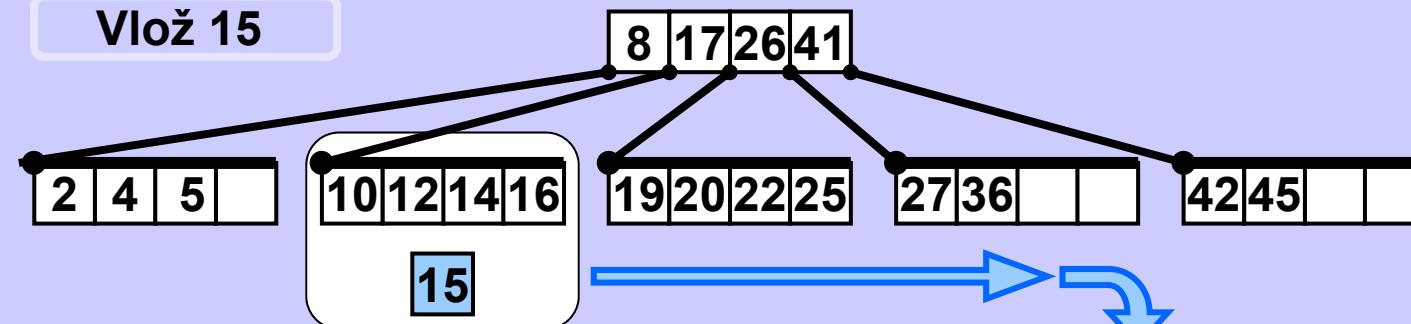
Medián zkus vložit  
do rodiče.

Zdařilo se.



## B-strom -- Insert

Vlož 15

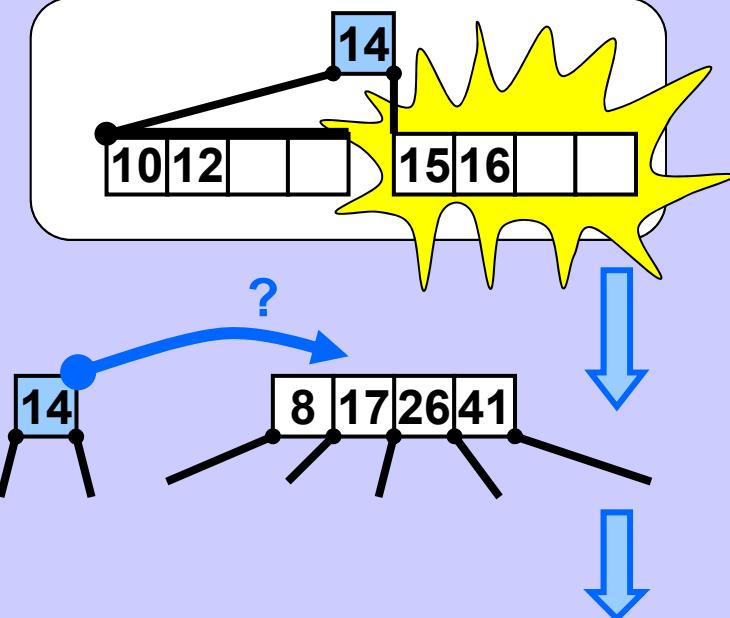


Seřad' mimo strom.

10 12 14 15 16

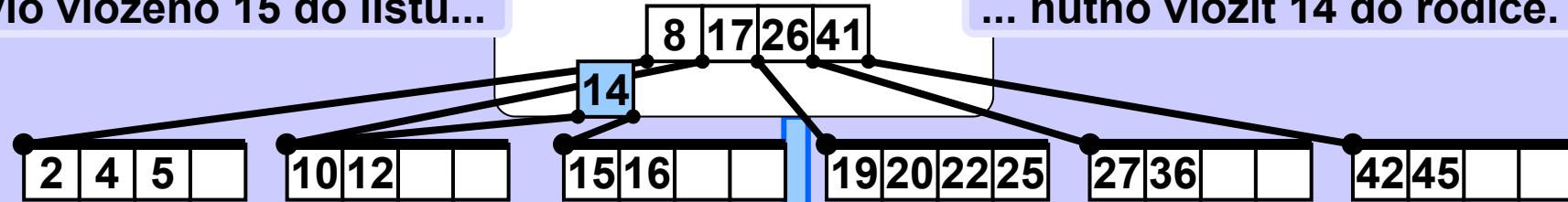
Vyber medián,  
vyvoř nový uzel,  
přesuň do něj hodnoty  
větší než medián.

Medián zkus vložit  
do rodiče.



## B-strom -- Insert

Bylo vloženo 15 do listu...



... nutno vložit 14 do rodiče.

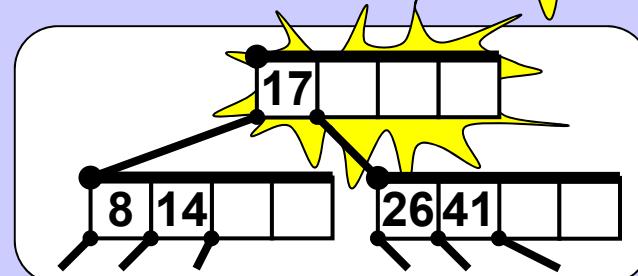
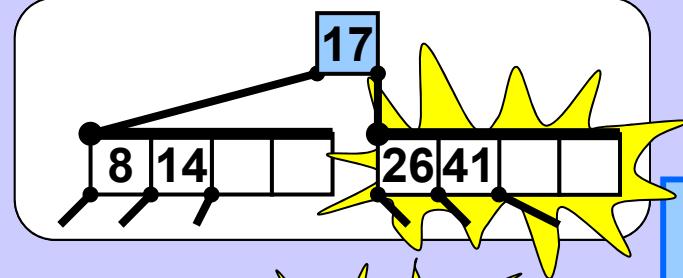
Rodič je zaplněn – Analogický další postup směrem ke kořeni.

Seřad' mimo strom.

Vyber medián,  
vyvoř nový uzel,  
přesuň do něj hodnoty  
větší než medián.

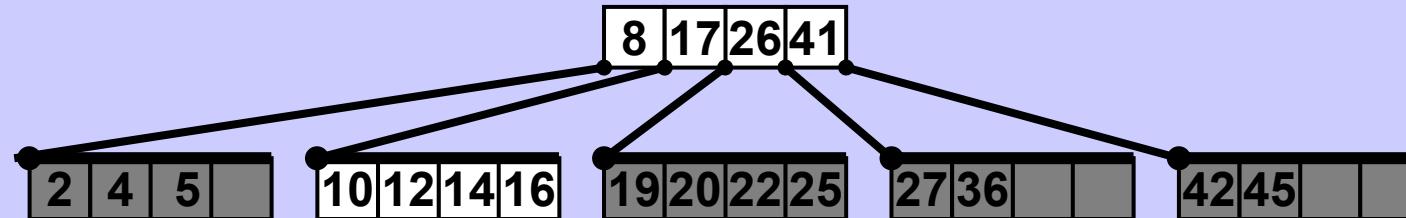
Medián nelze vložit do  
rodiče, žádný není, tedy  
se zřídí nový kořen.

8 14 17 26 41

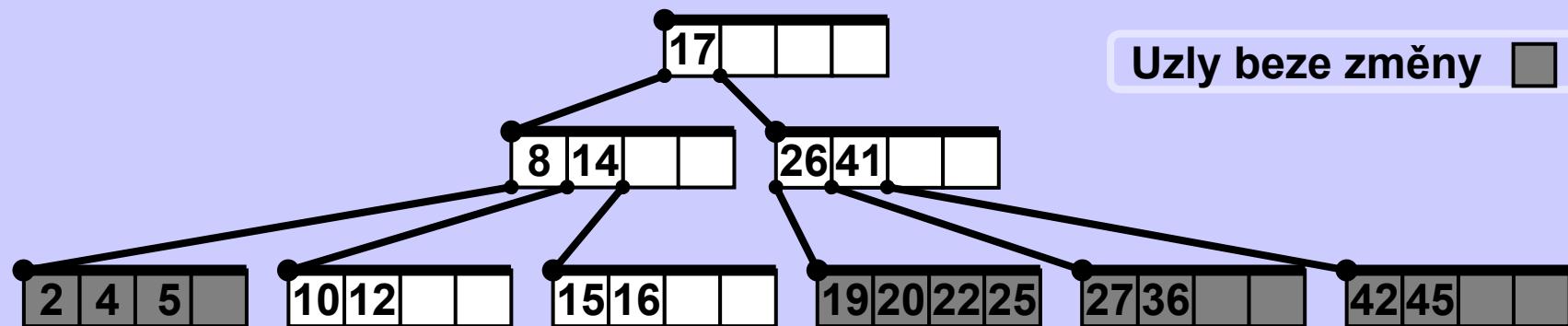


## B-strom -- Insert

### Rekapitulace - vlož 15



Vlož 15



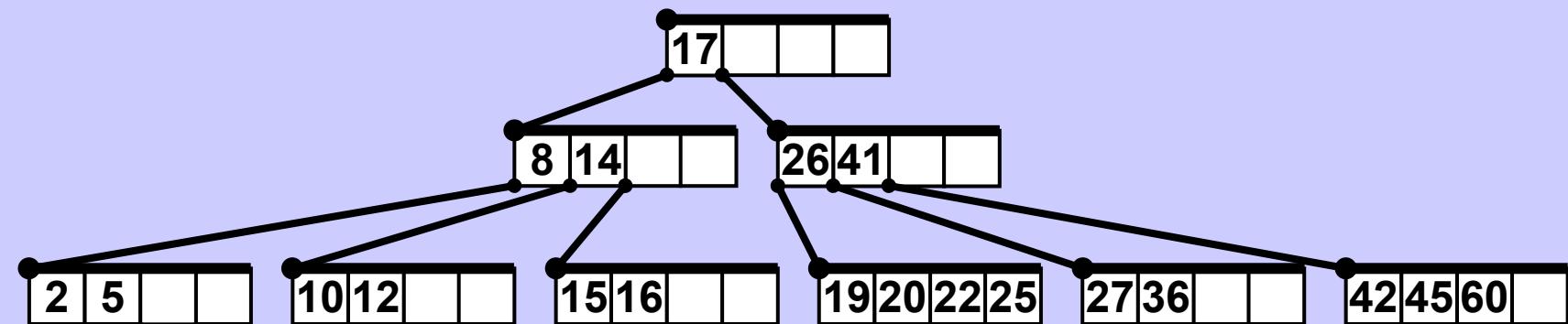
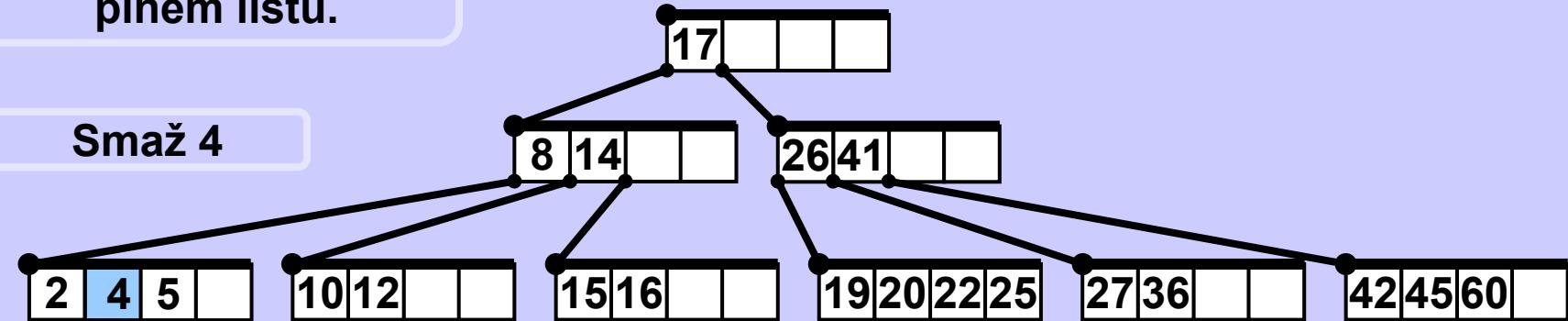
Uzly beze změny

V každém patře přibyl jeden uzel, kromě toho přibyl nový kořen, strom ale roste směrem "vzhůru", zůstává ideálně vyvážený.

## B-strom -- Delete

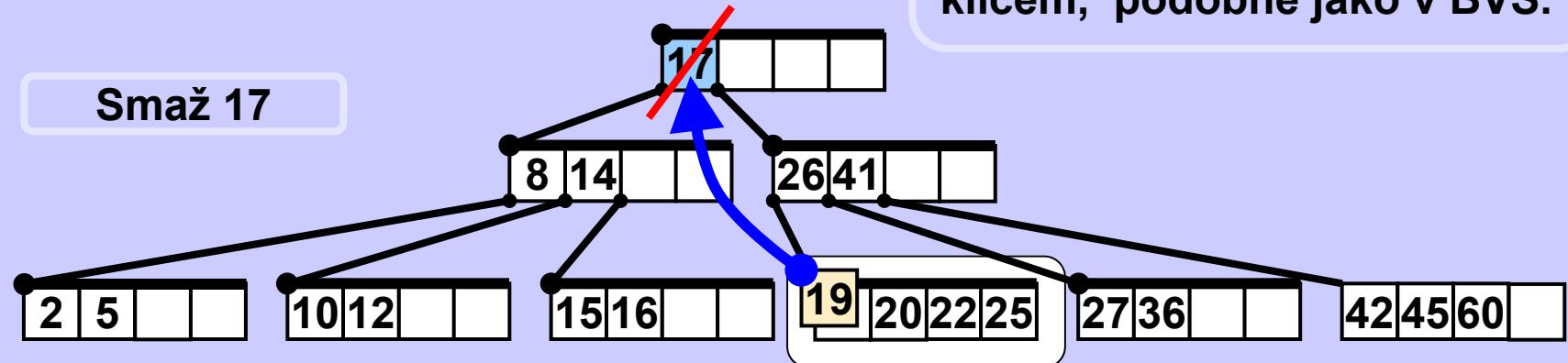
Mazání v dostatčně plném listu.

Smaž 4



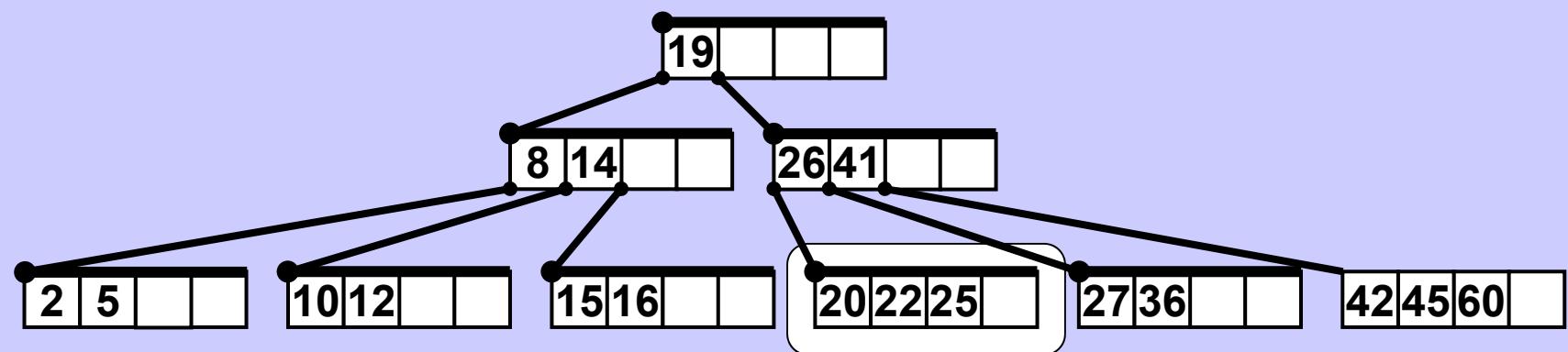
## B-strom -- Delete

Mazání ve vnitřním uzlu



Smazený klíč se nahradí nejbližším větším (menším) klíčem, podobně jako v BVS.

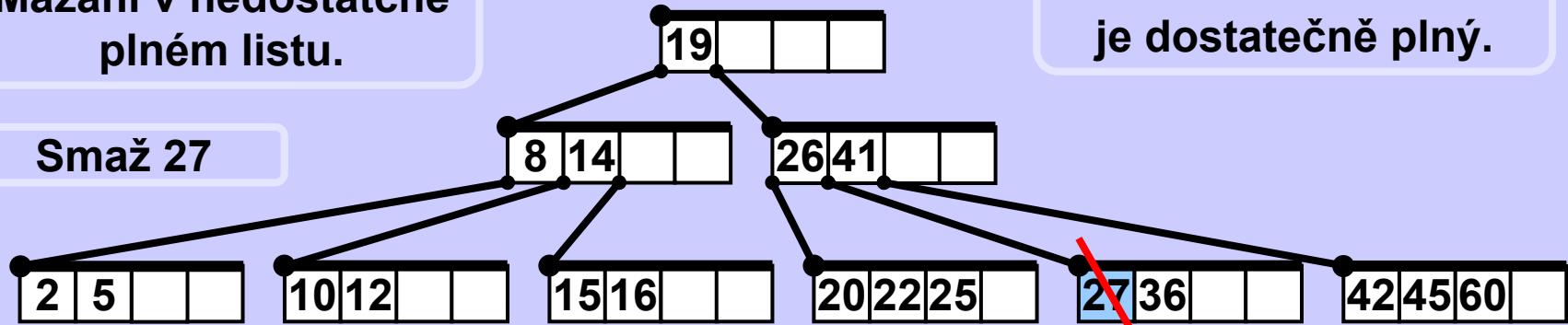
Nejbližší větší (menší) klíč je vždy v B-stromu v listu, má-li tento list dostatečný počet klíčů, jsme hotovi.



## B-strom -- Delete

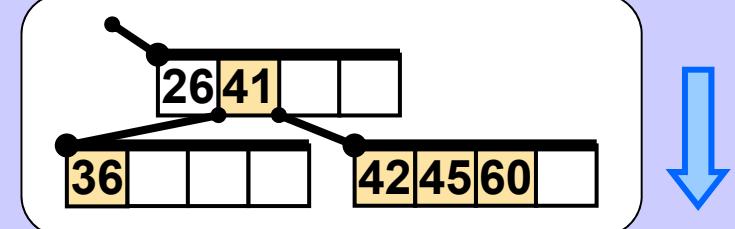
Mazání v nedostatčně plném listu.

Smaž 27



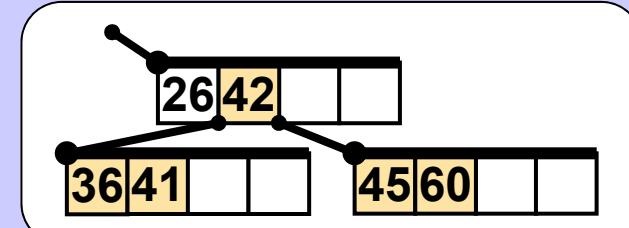
Ale sousední list je dostatečně plný.

Sjednot' klíče s klíči v sousedním listu a s dělícím klíčem v rodiči a seřad'.



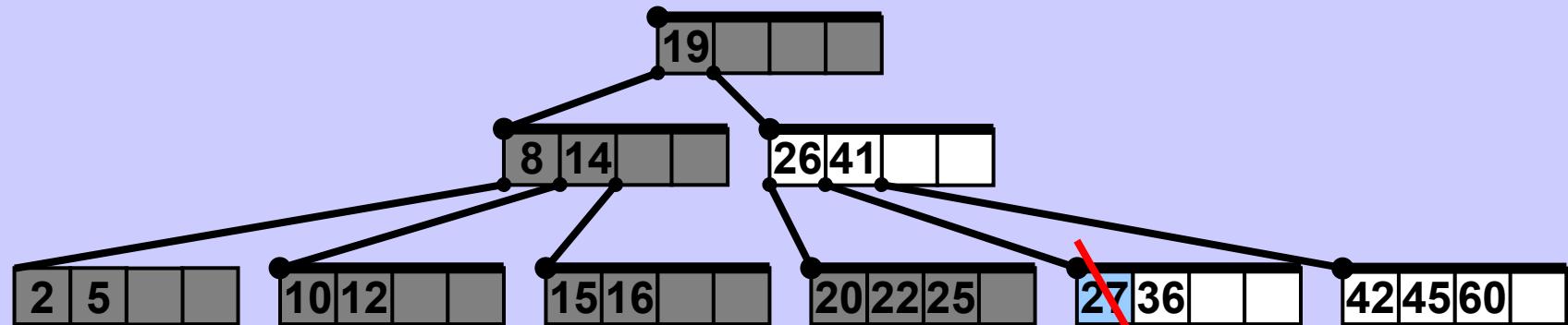
36 41 42 45 60

Medián sjednocení vlož na místo původně dělícího klíče, menší a větší klíče než medián rozděl do levého a pravého listu.



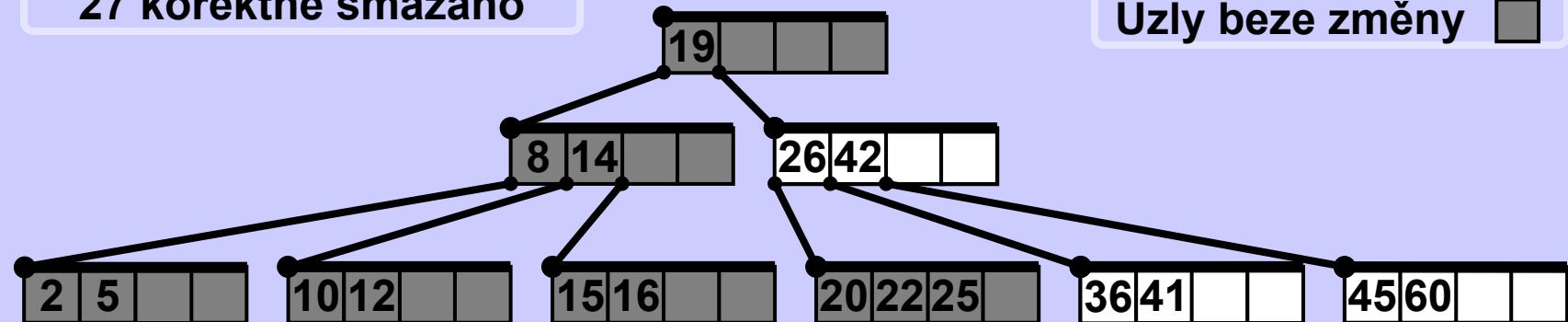
## B-strom -- Delete

Rekapitulace - smaž 27



27 korektně smazáno

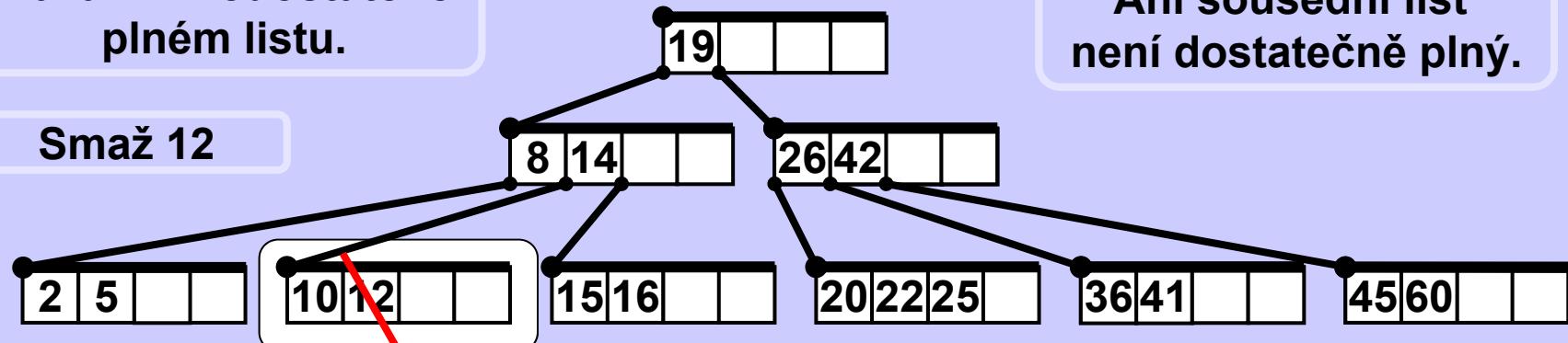
Uzly beze změny



## B-strom -- Delete

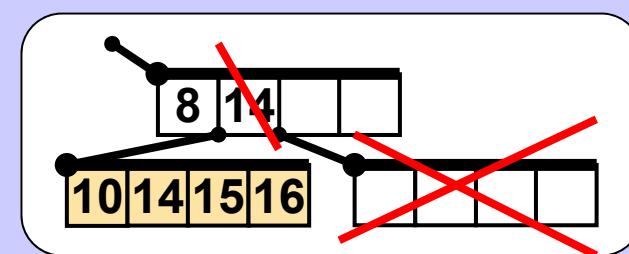
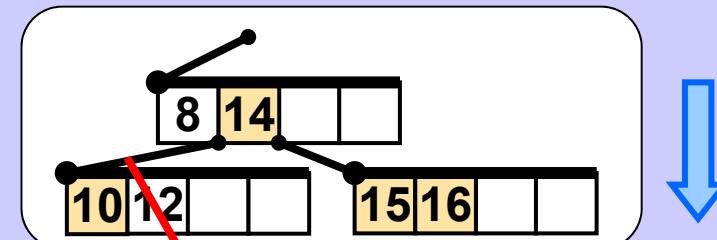
Mazání v nedostatčně plném listu.

Smaž 12



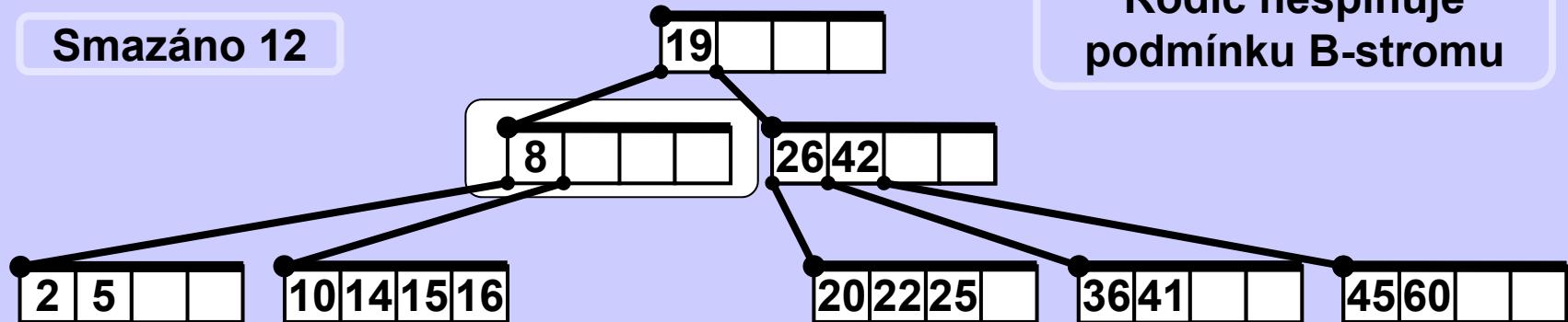
Ani sousední list  
není dostatečně plný.

Sjednot' klíče s klíči v sousedním listu  
a s dělícím klíčem v rodiči a seřad'.  
Vše vlož do původního listu,  
sousední list smaž,  
dělící klíč v rodiči také smaž.



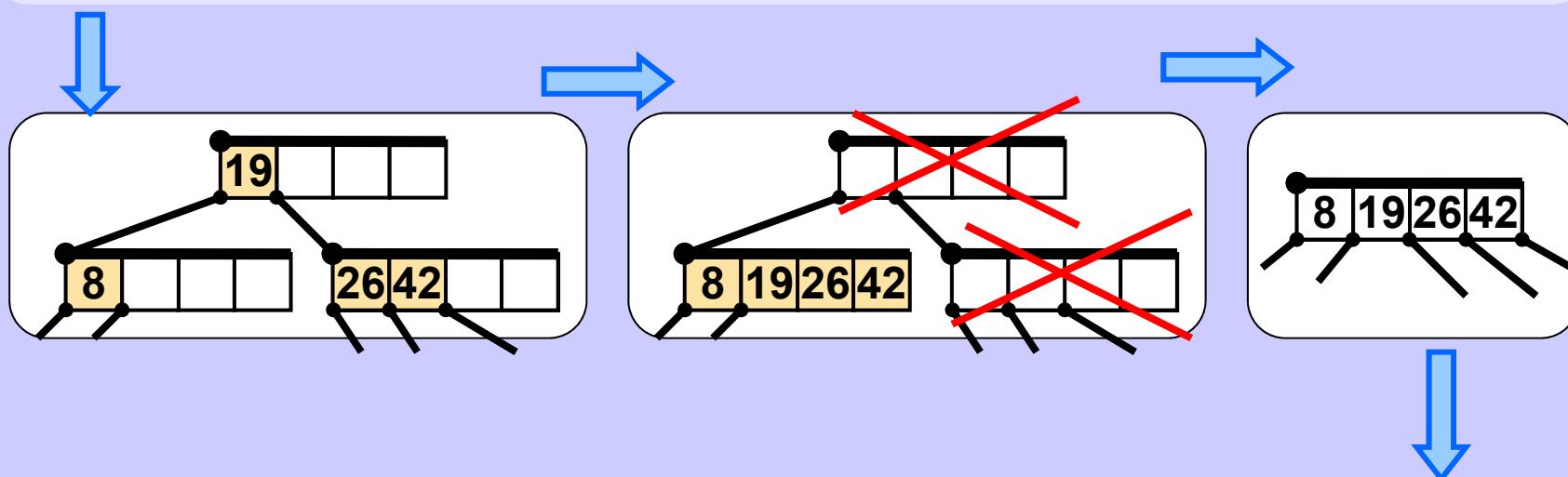
## B-strom -- Delete

Smazáno 12



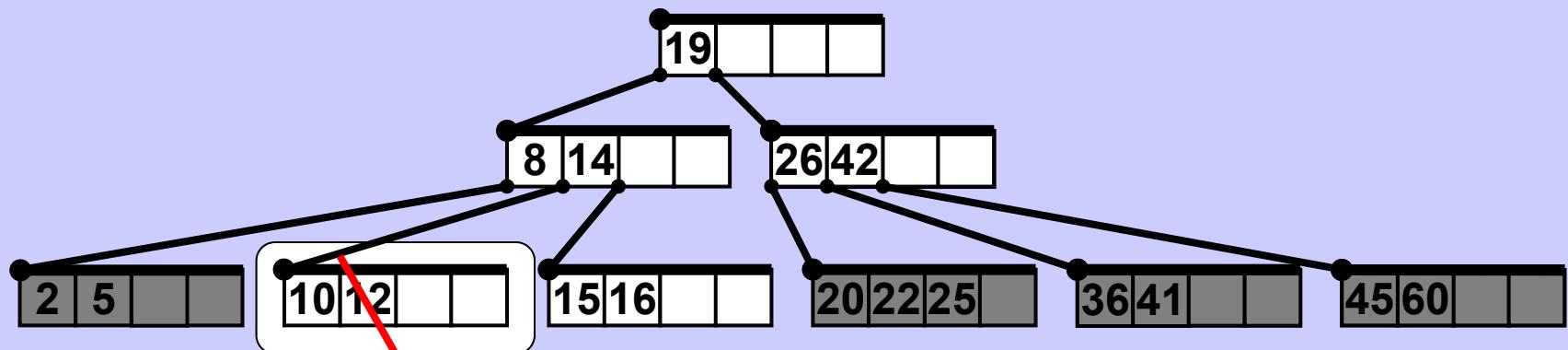
Rodič nesplňuje podmínu B-stromu

Rodič, který poskytl klíč potomku, není dostatečně plný.  
Aplikujeme na něj (a případně iterativně na jeho rodiče) tentýž postup spojení klíčů a sousedních uzlů a přesun dělícího prvku z rodiče.

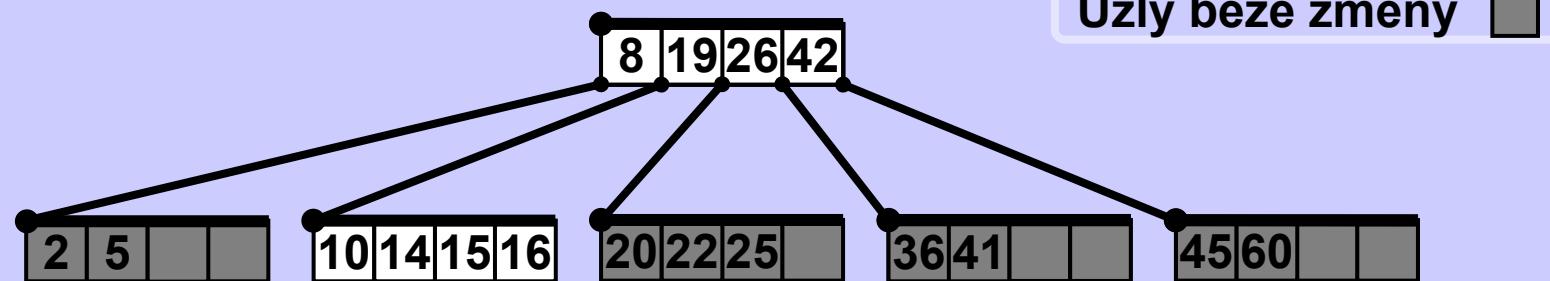


## B-strom -- Delete

Rekapitulace - smaž 12



Smažáno 12 a strom byl adekvátně restrukturován.



## ALG 07

**Selection sort** (Select sort)

**Insertion sort** (Insert sort)

**Bubble sort** deprecated

**Quicksort**

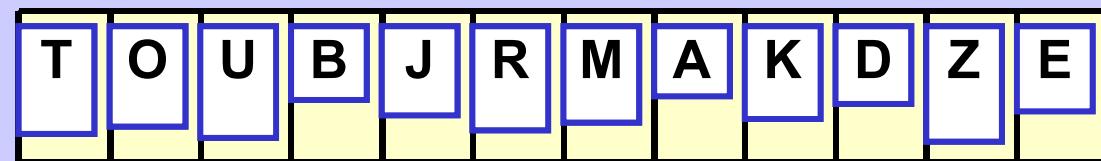
**Stabilita řazení**

## Selection sort

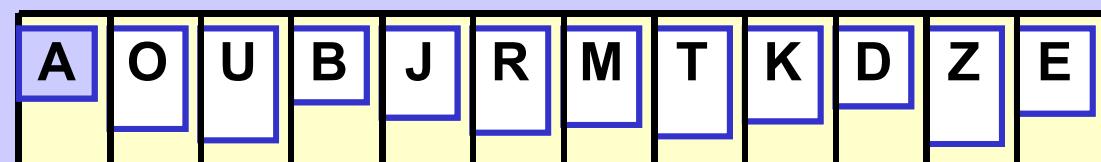
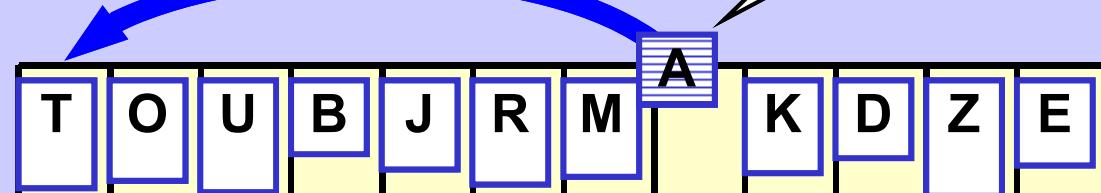
Neseřazeno

Seřazeno

Start

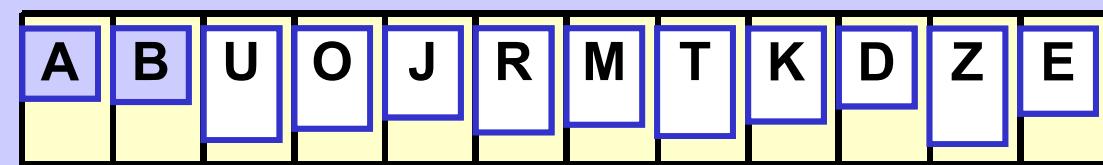
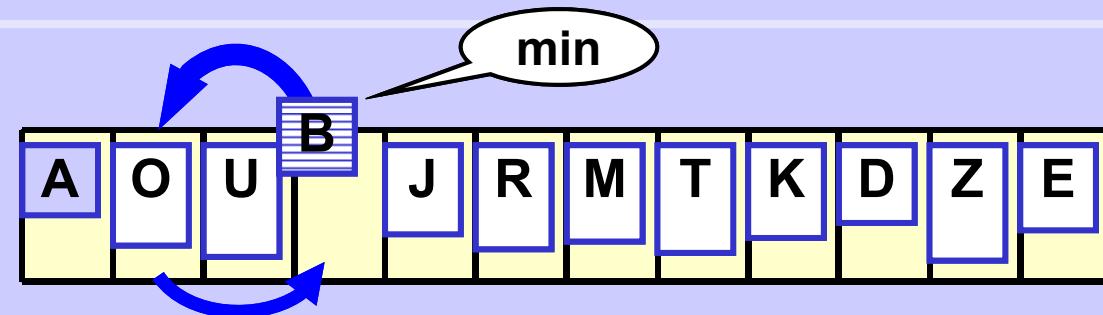


Krok1

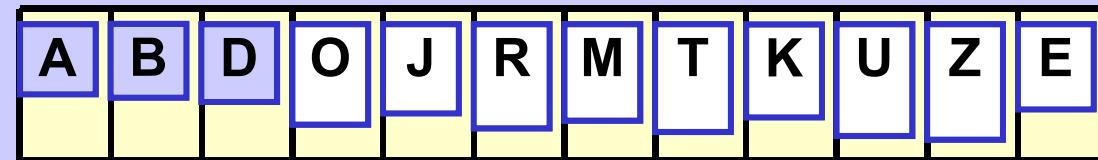
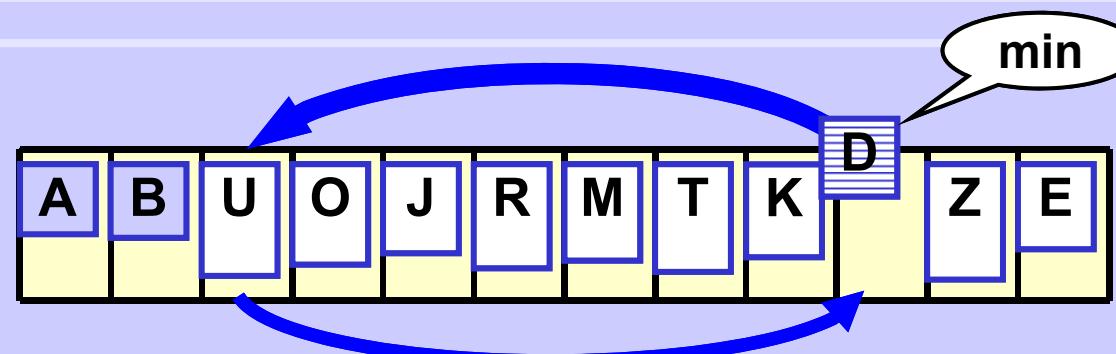


## Selection sort

Krok 2



Krok 3

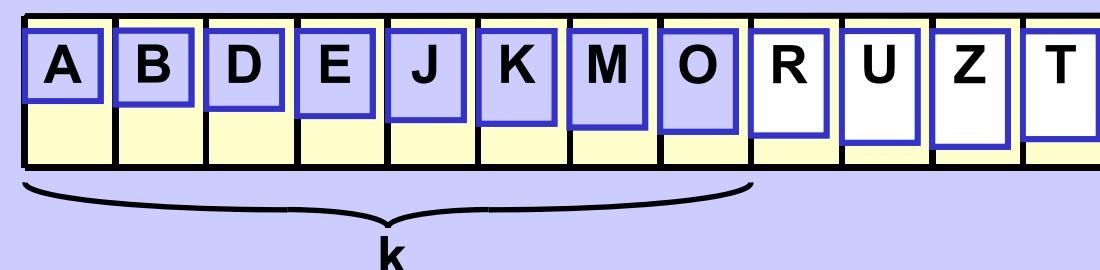
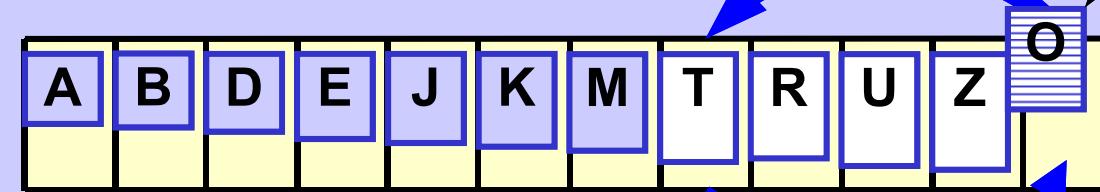


## Selection sort

...

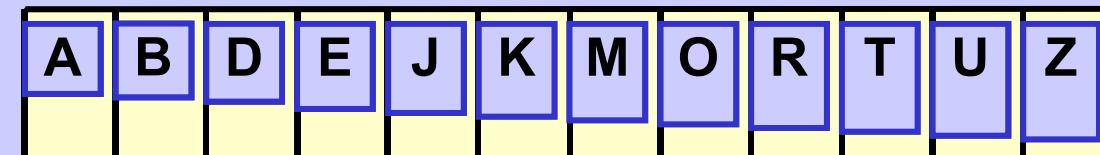
**Krok k**

...

 $k$ 

...

...

**Seřazeno**

## Selection sort

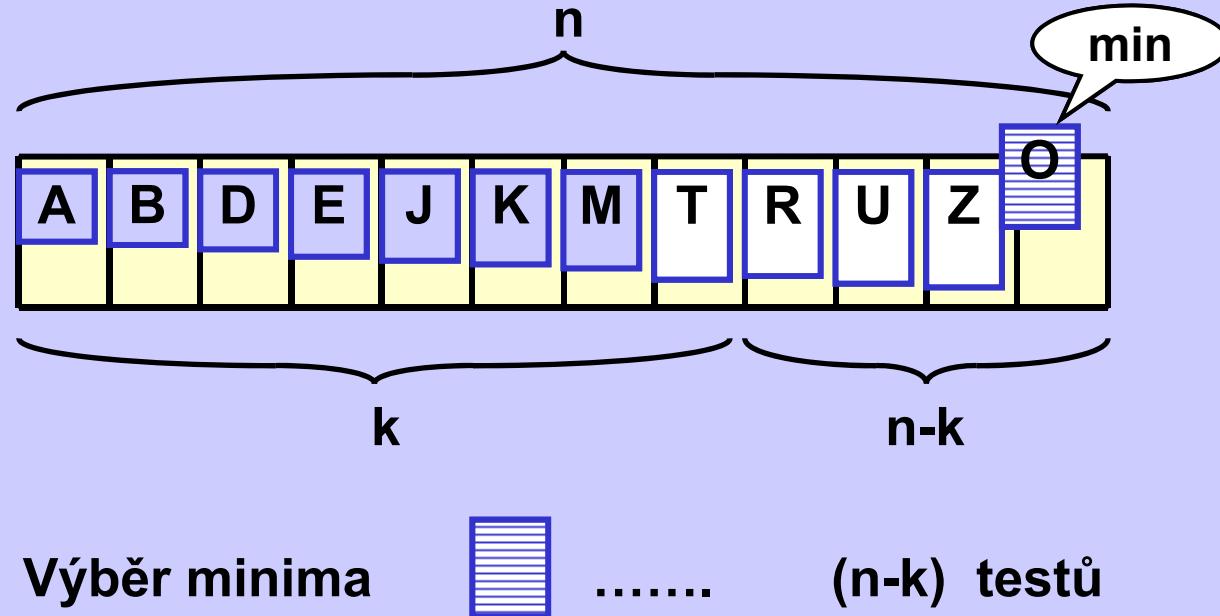
```
for( int i = 0; i < n-1; i++ ){

    // select min
    jmin = i;
    for( int j = i+1; j < n; j++ )
        if( a[j] < a[jmin] )
            jmin = j;

    // put min to its place
    min = a[jmin];
    a[jmin] = a[i];
    a[i] = min;
}
```

## Selection sort

Krok k

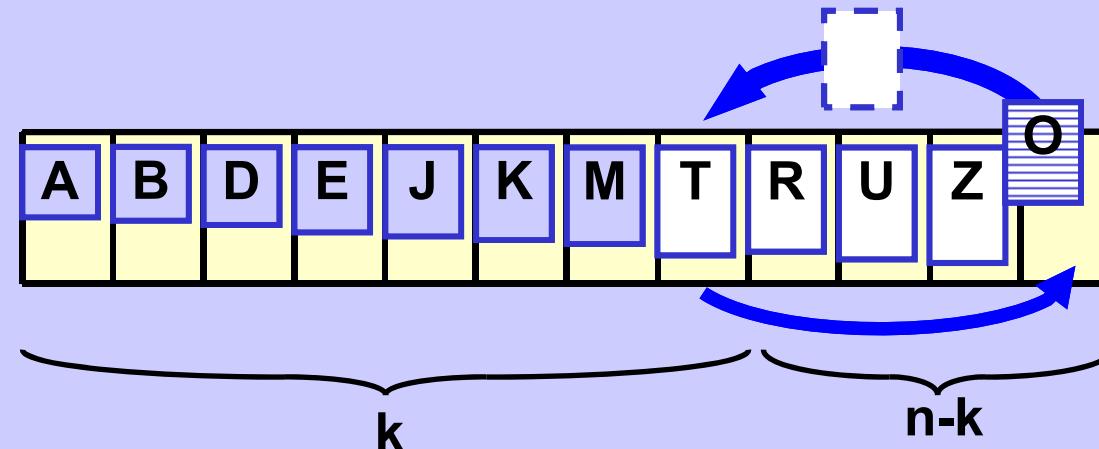


Celkem  
testů

$$\sum_{k=1}^{n-1} (n-k) = \sum_{k=1}^{n-1} n - \sum_{k=1}^{n-1} k = n(n-1) - \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$$

## Selection sort

Krok k



přesuny .....

3

Celkem  
přesunů

$$\sum_{k=1}^{n-1} 3 = 3(n - 1)$$

## Selection sort

### Shrnutí

Celkem  
testů

$$\frac{1}{2}(n^2 - n) = \Theta(n^2)$$

Celkem  
přesunů

$$3(n-1) = \Theta(n)$$

Celkem  
operací

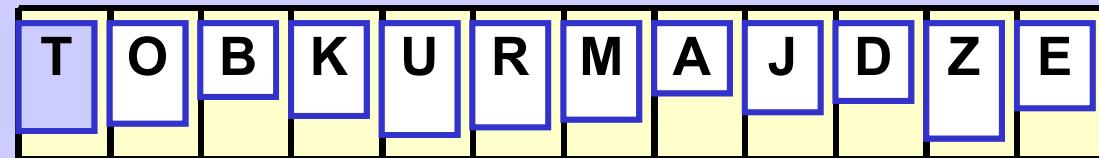
$$\frac{1}{2}(n^2 - n) + 3(n-1) = \Theta(n^2)$$

---

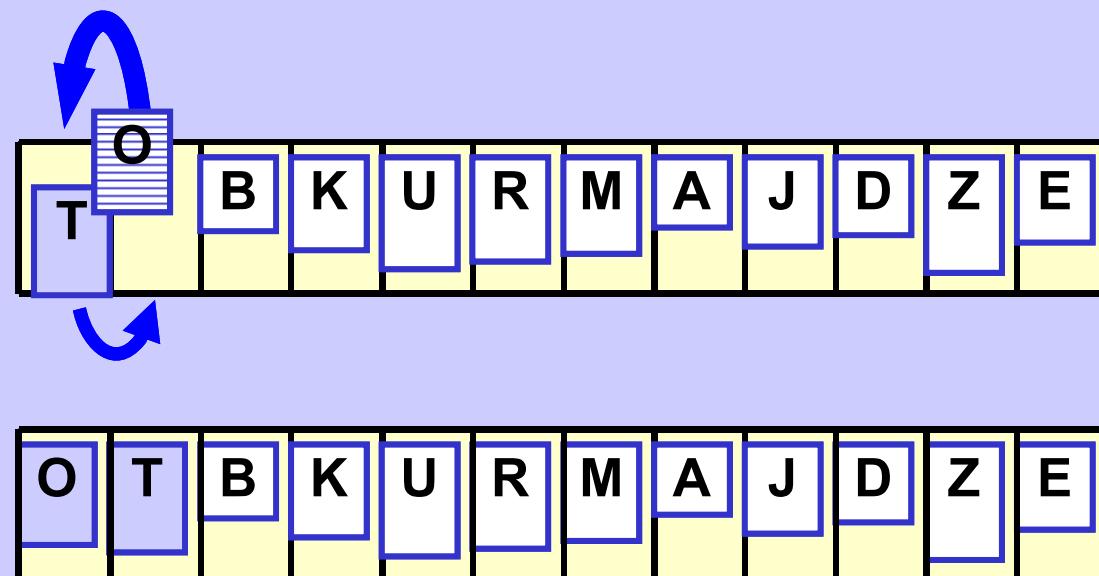
Asymptotická složitost Selection Sortu je  $\Theta(n^2)$

## Insertion sort

Start

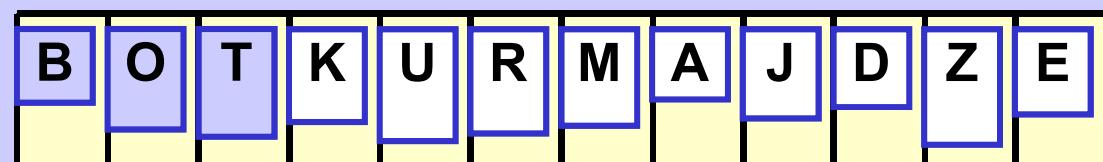
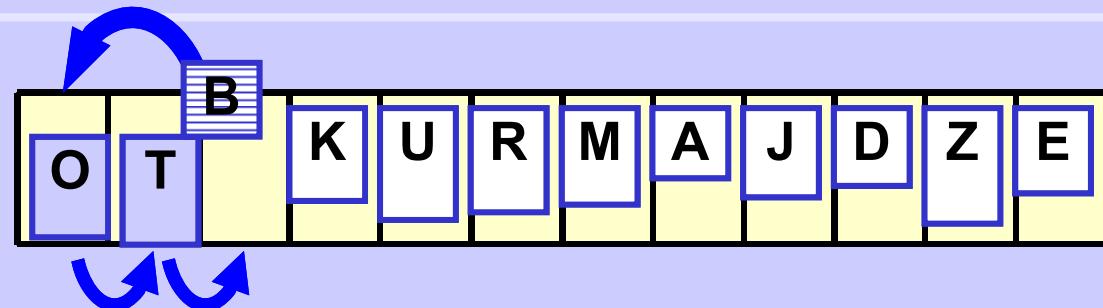


Krok1

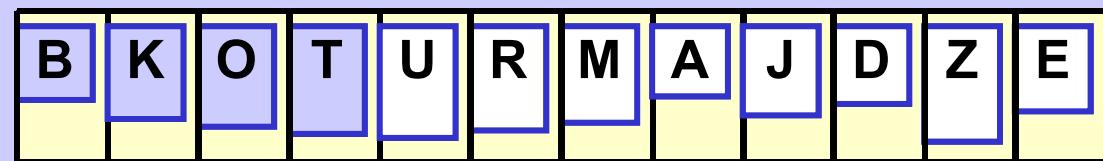
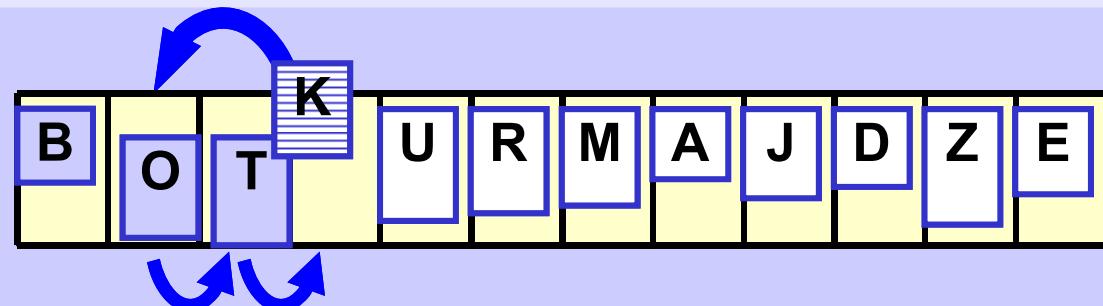


## Insertion sort

Krok 2



Krok 3



## Insertion sort

...

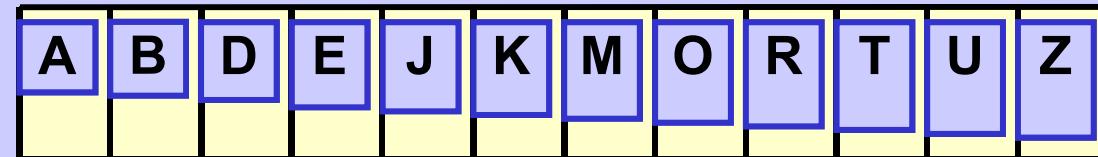
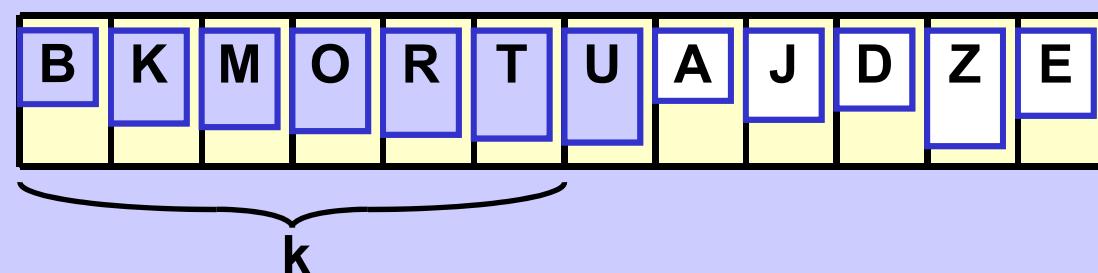
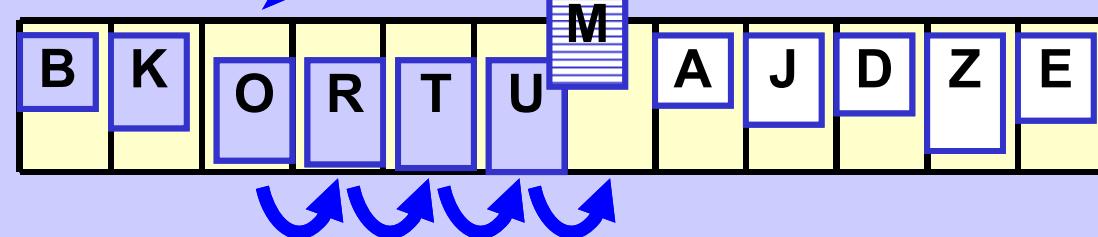
Krok k

...

Seřazeno

...

...



## Insertion sort

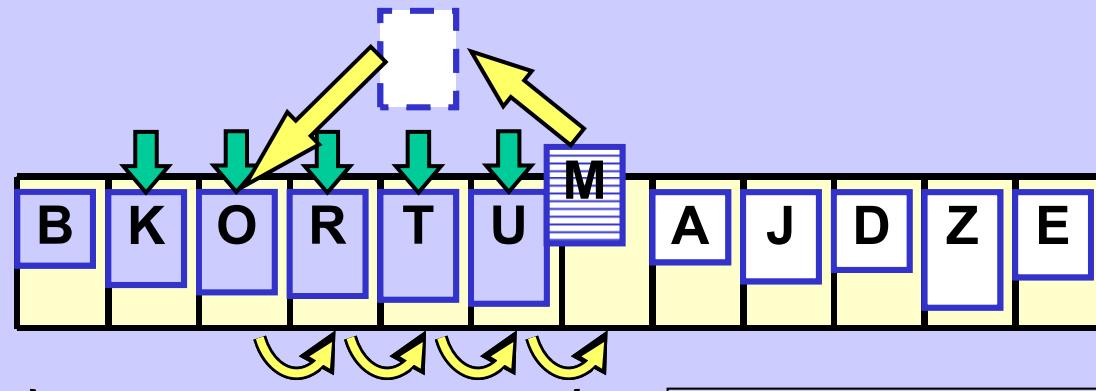
```
for( int i = 1; i < n; i++ ){

    // find & make place for a[i]
    insVal = a[i];
    int j = i-1;
    while( (j >= 0) && (a[j] > insVal) ){
        a[j+1] = a[j];
        j--;
    }

    // insert a[i]
    a[j+1] = insVal;
}
```

## Insertion sort

Krok k



$\downarrow$  testů + 1 = přesunů

testů .....	1 k $(k+1)/2$	nejlepší případ nejhorší případ průměrný případ
-------------	---------------------	---

přesunů .....	2 k+1 $(k+3)/2$	nejlepší případ nejhorší případ průměrný případ
---------------	-----------------------	---

## Insertion sort

Celkem  
testů

### Shrnutí

$$n - 1 = \Theta(n)$$

$$(n^2 - n)/2 = \Theta(n^2)$$

$$(n^2 + n - 2)/4 = \Theta(n^2)$$

nejlepší případ

nejhorší případ

průměrný případ

Celkem  
přesunů

$$2n - 2 = \Theta(n)$$

$$(n^2 + n - 2)/2 = \Theta(n^2)$$

$$(n^2 + 5n - 6)/4 = \Theta(n^2)$$

nejlepší případ

nejhorší případ

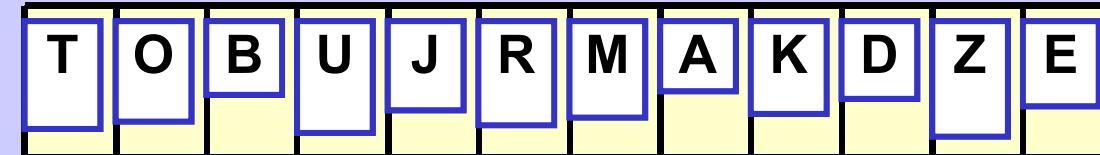
průměrný případ

---

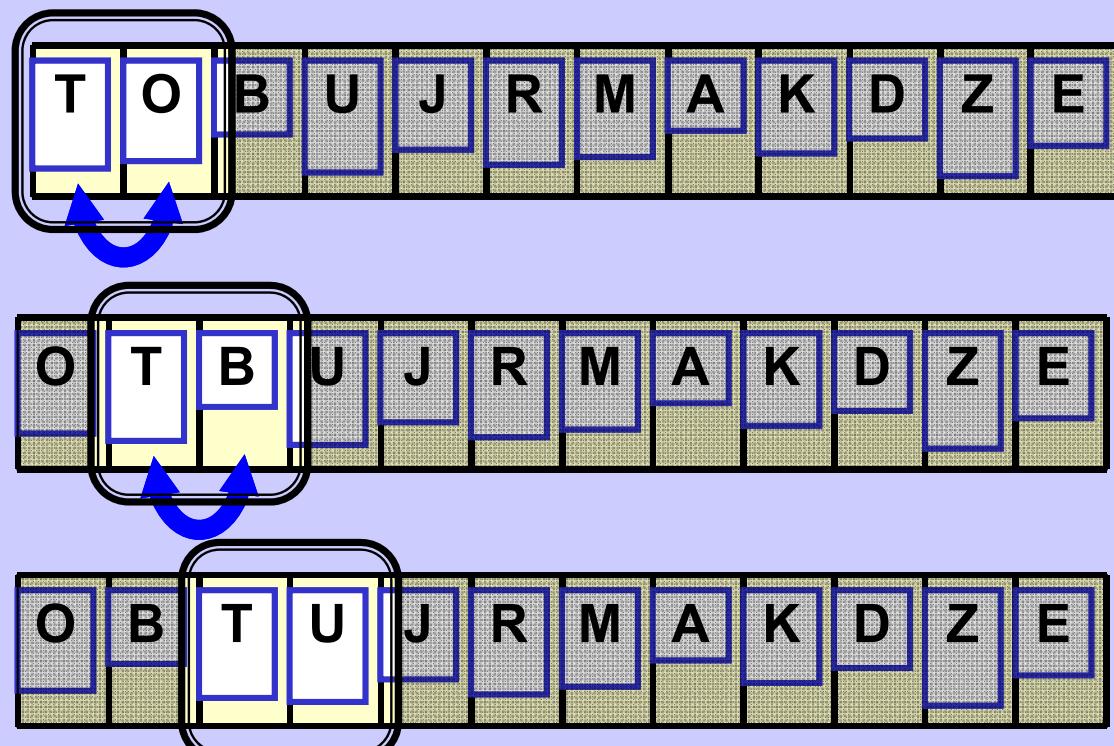
**Asymptotická složitost Insertion Sortu je  $O(n^2)$  (!!)**

## Bubble sort

Start

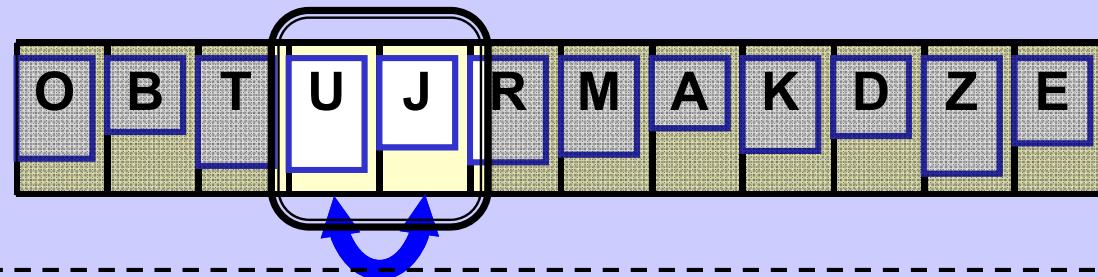


Fáze 1

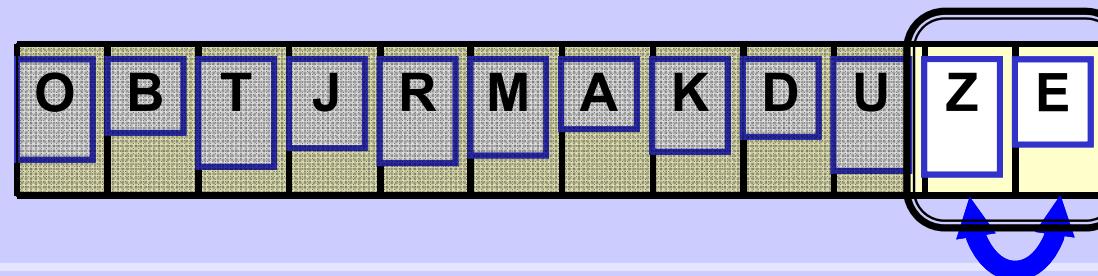


## Bubble sort

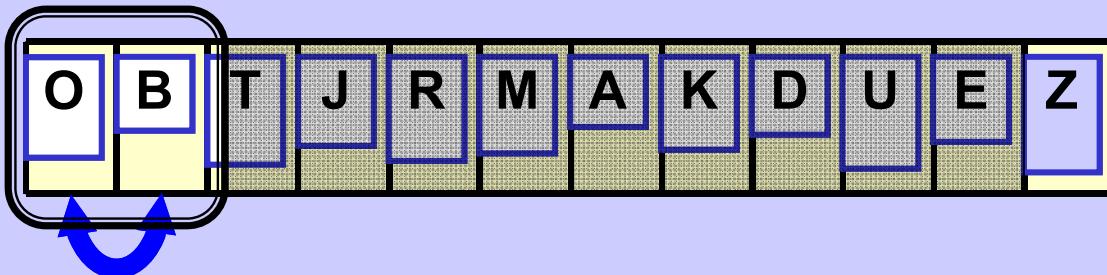
Fáze 1



... etc ...

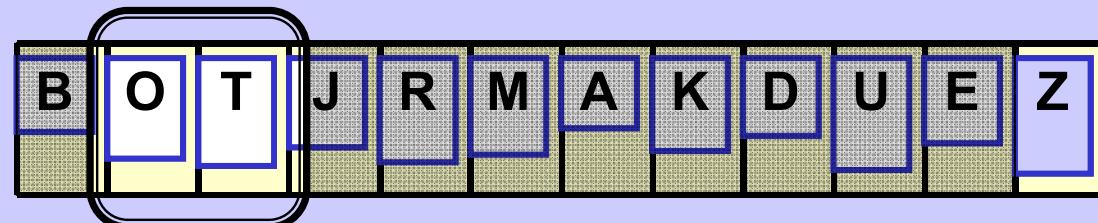


Fáze 2

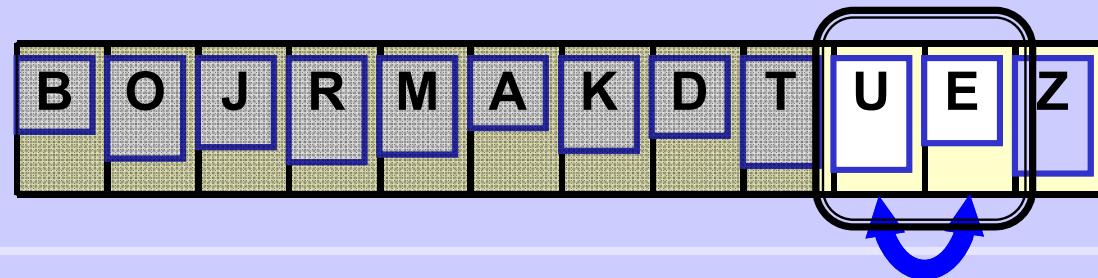


## Bubble sort

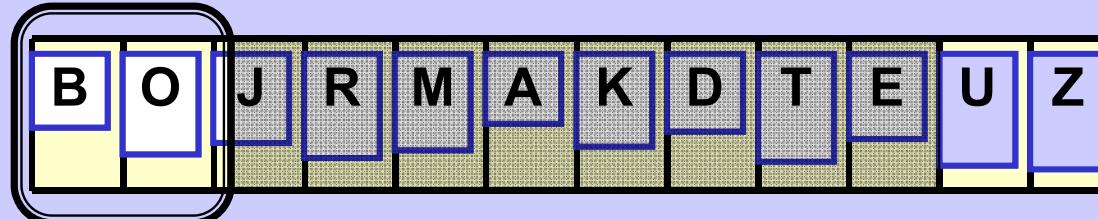
Fáze 2



...etc...



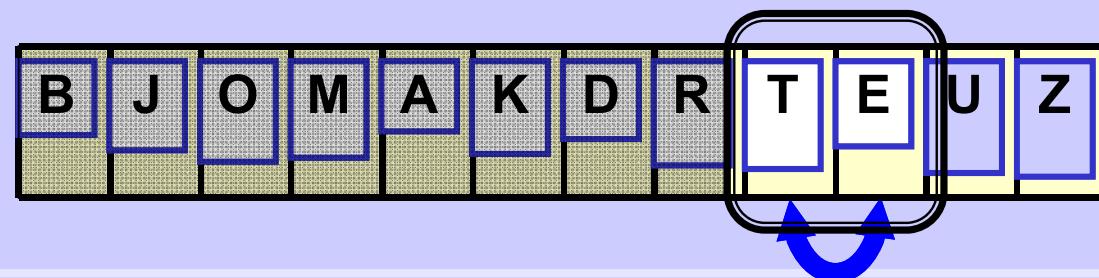
Fáze 3



## Bubble sort

Fáze 3

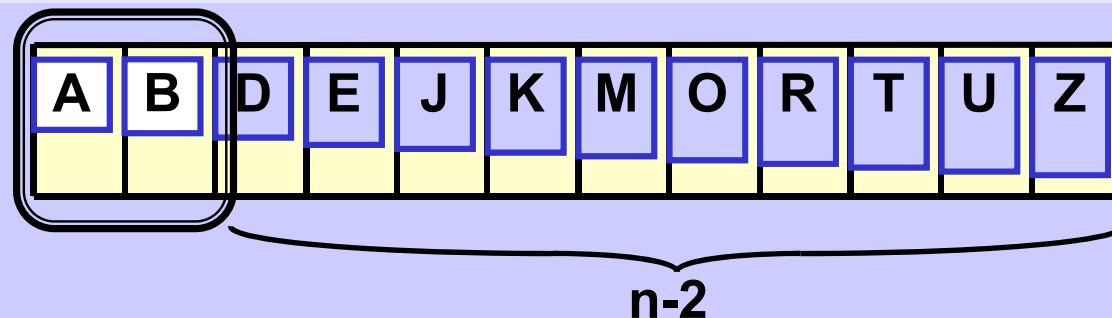
... atd ...



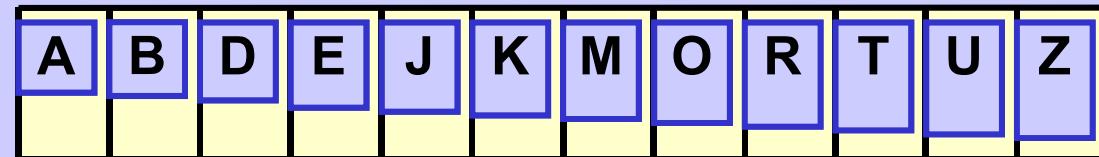
...

...

Fáze n-1



Seřazeno



## Insertion sort

```
for( int lastPos = n-1; lastPos > 0; lastPos-- )
    for( int j = 0; j < lastPos; j++ )
        if( a[j] > a[j+1] ) swap( a, j, j+1 );
```

### Shrnutí

Celkem  
testů

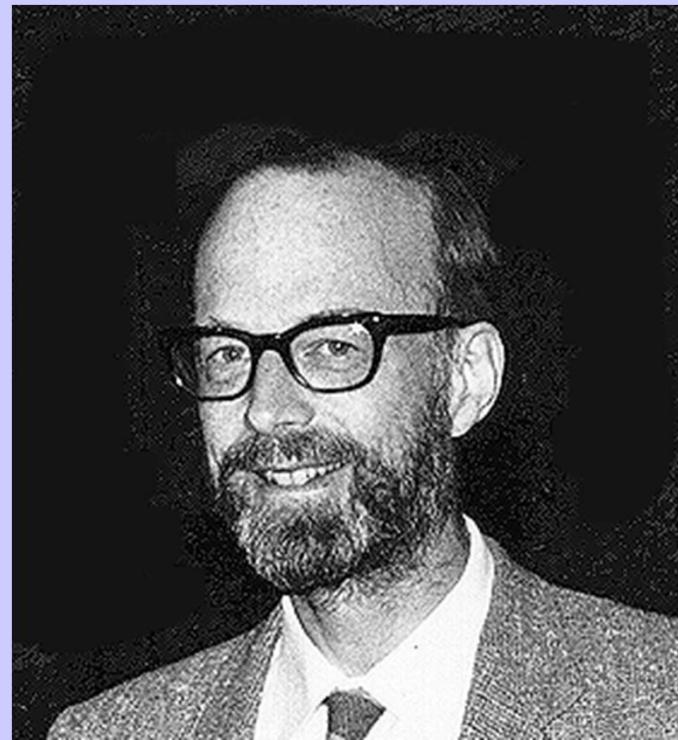
$$(n-1) + (n-2) + \dots + 2 + 1 = \frac{1}{2}(n^2 - n) = \Theta(n^2)$$

Celkem  
přesunů

$0 = \Theta(1)$	<b>nejlepší případ</b>
$\frac{1}{2}(n^2 - n) = \Theta(n^2)$	<b>nejhorší případ</b>

**Asymptotická složitost Bubble Sortu je  $\Theta(n^2)$**

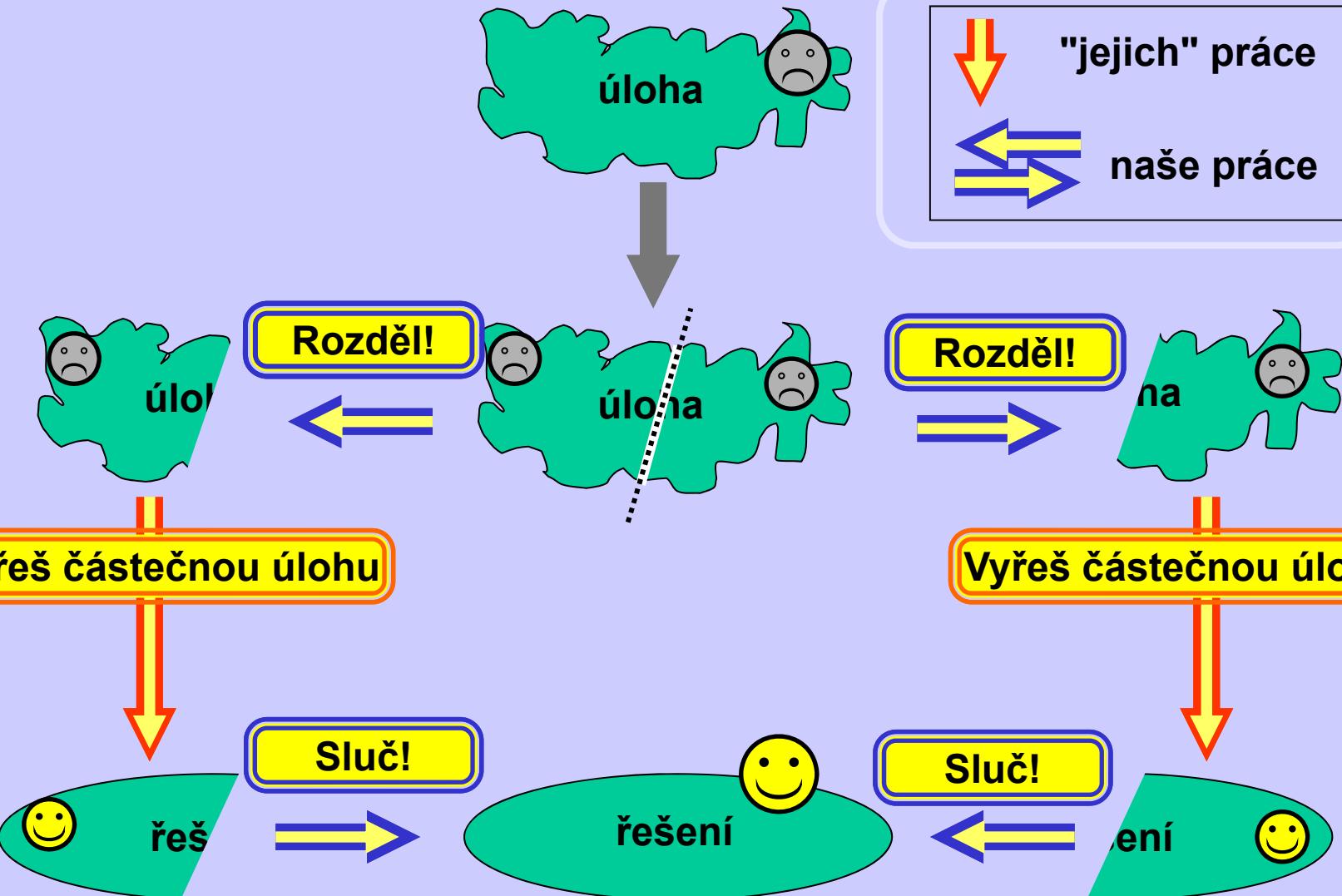
## Quicksort



**Sir Charles Antony Richard Hoare**

**C. A. R. Hoare: Quicksort. Computer Journal, Vol. 5, 1, 10-15 (1962)**

# Rozděl a panuj! Divide and conquer! Divide et impera!



## Quicksort

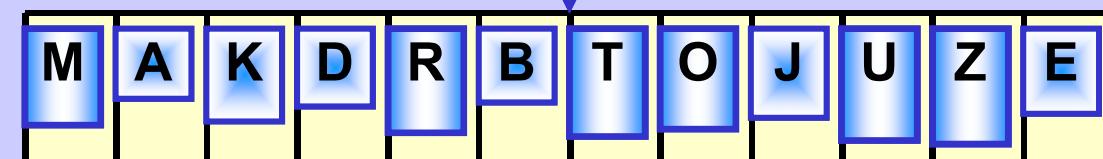
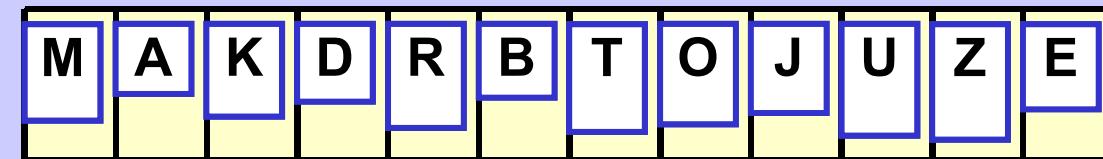
Myšlenka

Divide & Conquer!

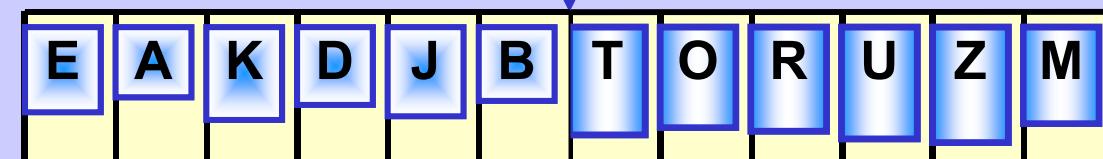
Start

Malá

Velká

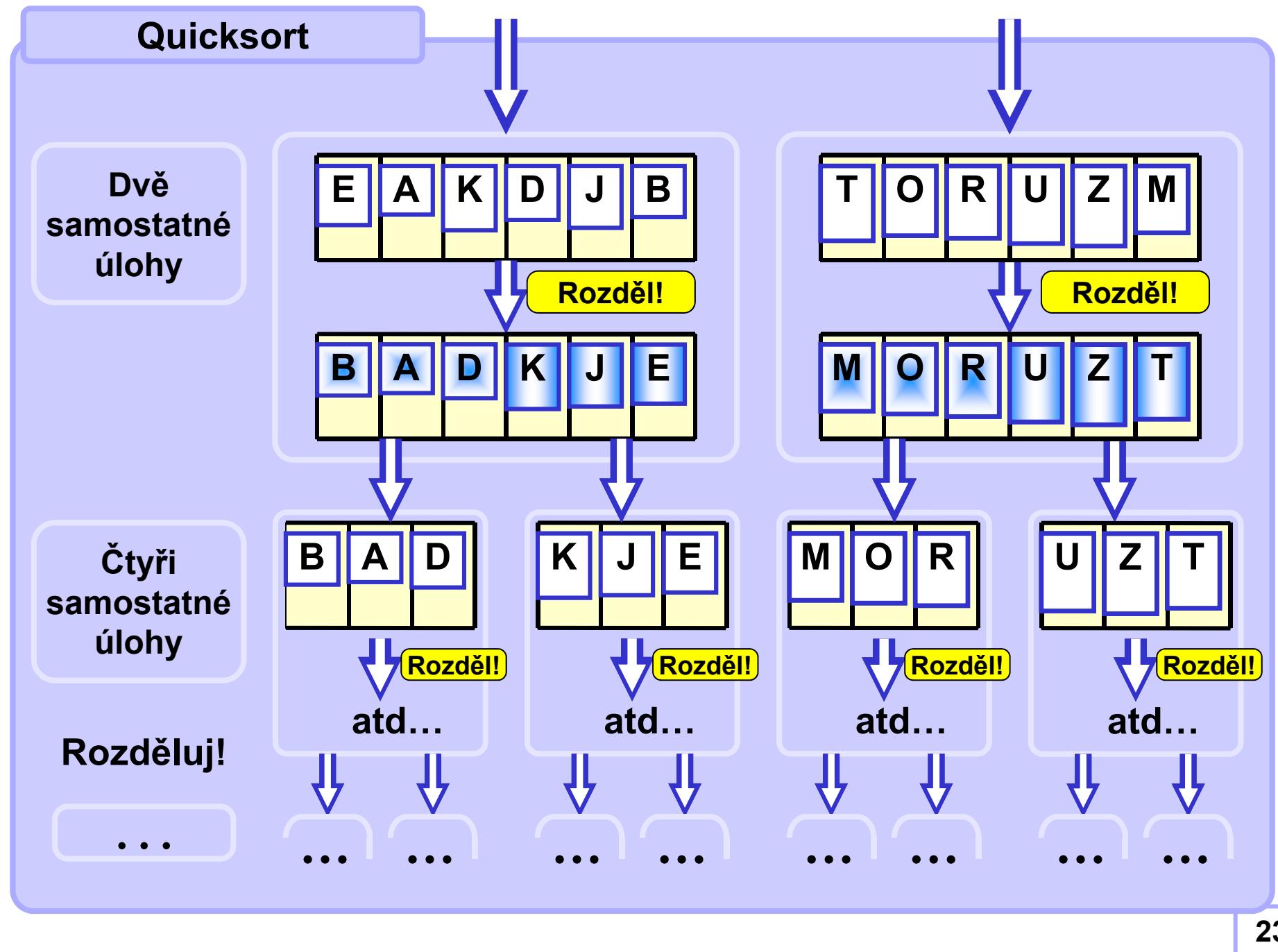


Rozděl!

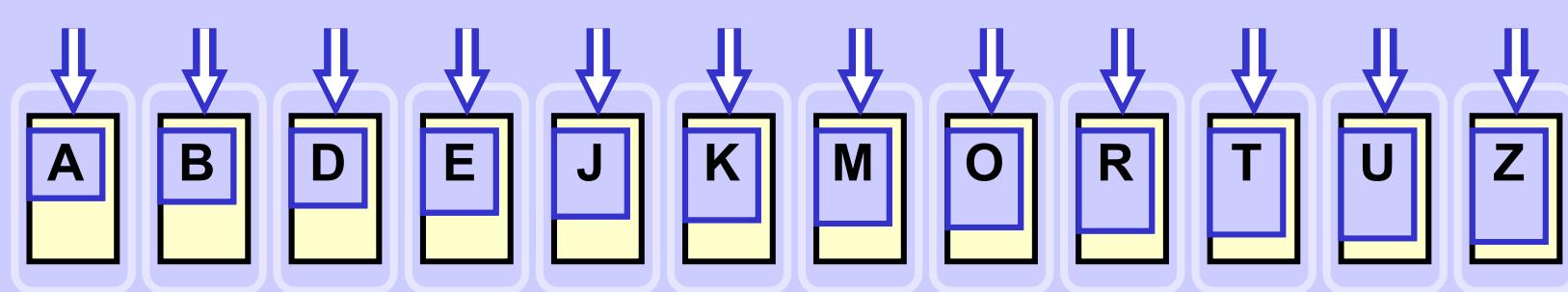


Malá

Velká



## Quicksort

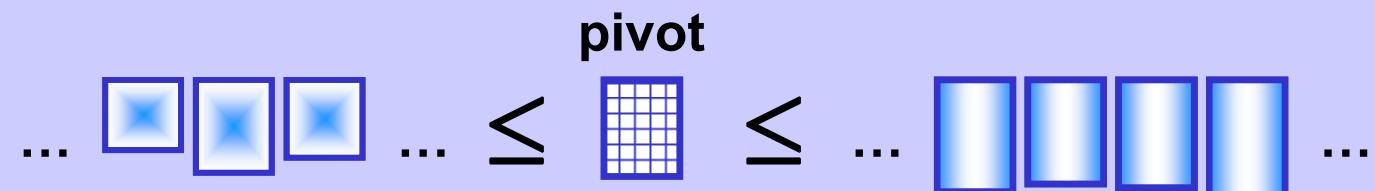


Opanováno!

## Quicksort

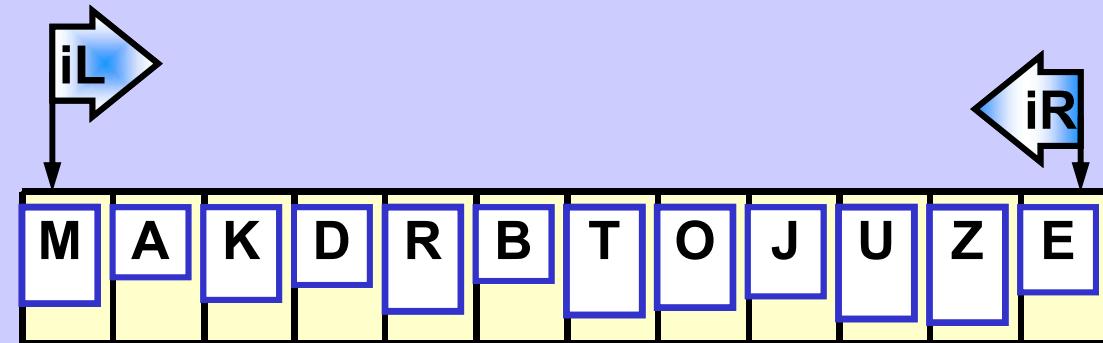
### Dělení

Pivot



Init

Pivot

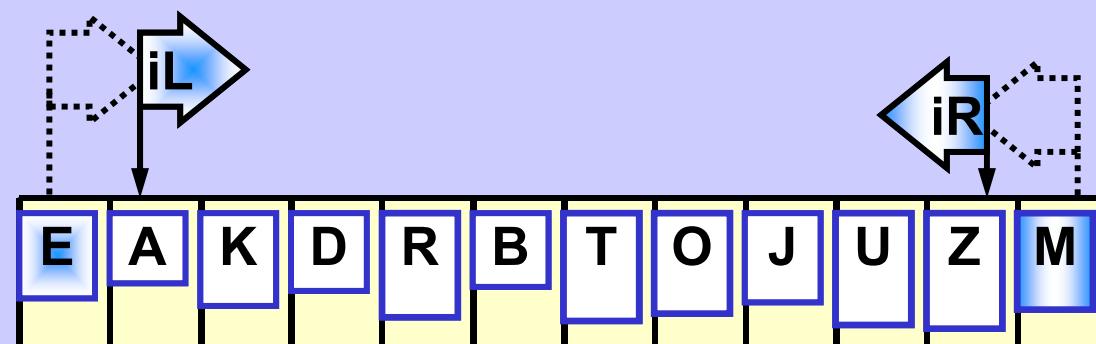
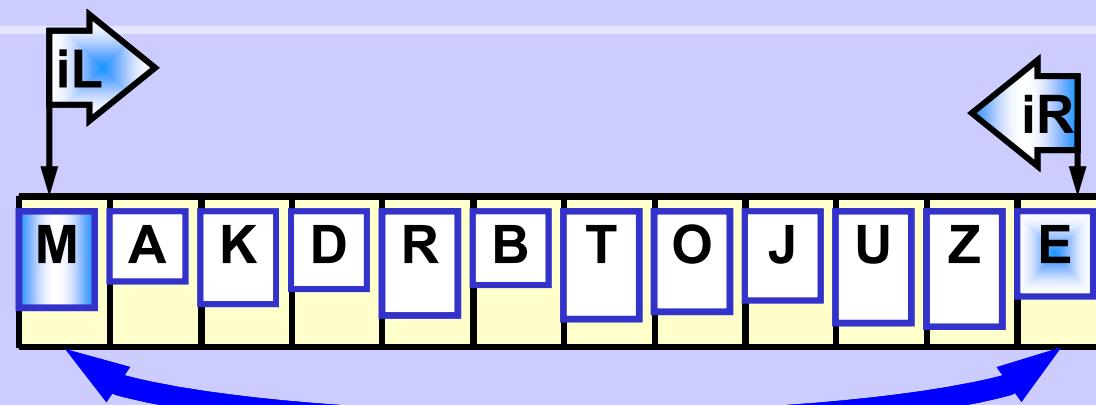
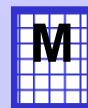


## Quicksort

### Dělení

Krok 1

Pivot

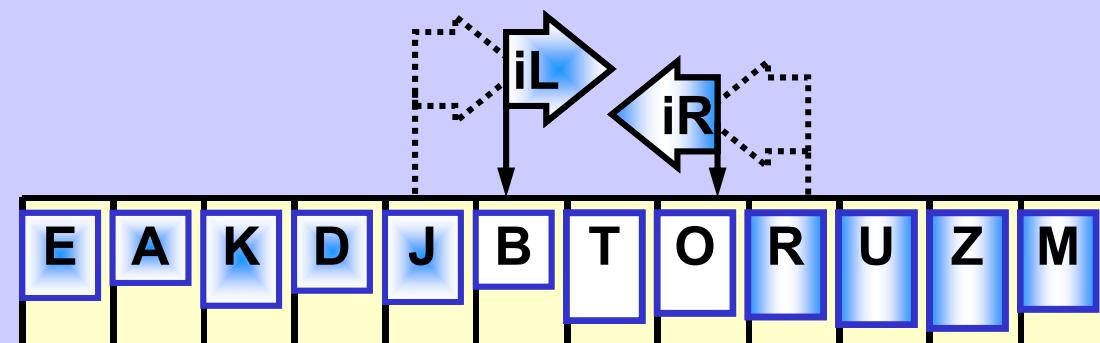
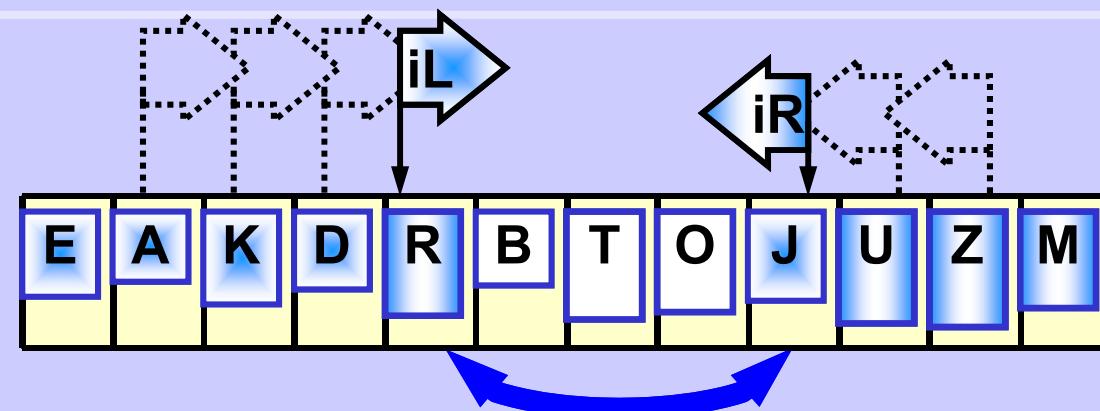


## Quicksort

### Dělení

Krok 2

Pivot

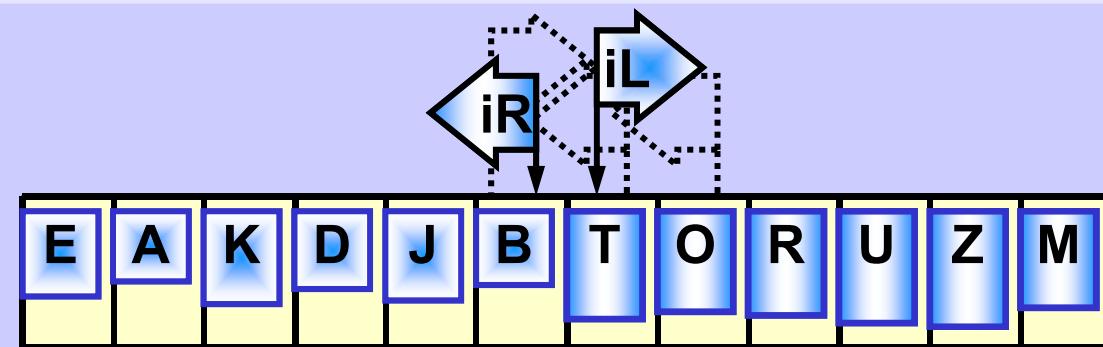


# Quicksort

## Dělení

Krok 3

Pivot

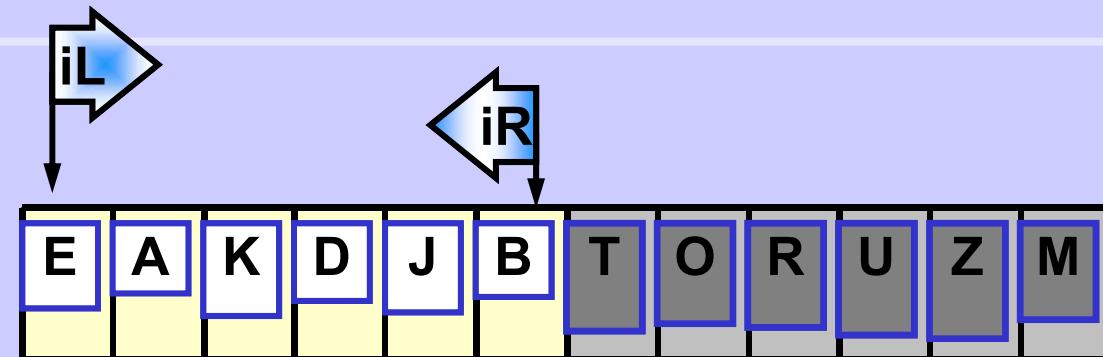
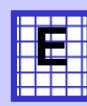


$iR < iL$  Stop

Rozděl!

Init

Pivot

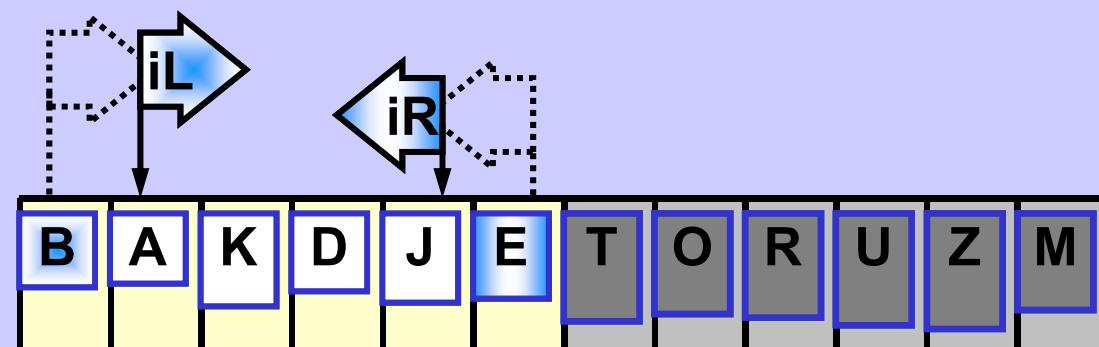
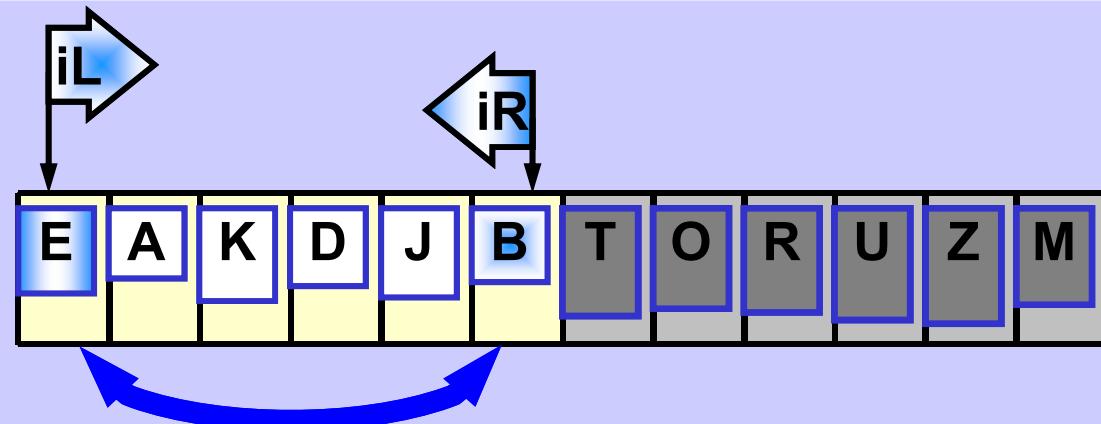


## Quicksort

### Dělení

Krok 1

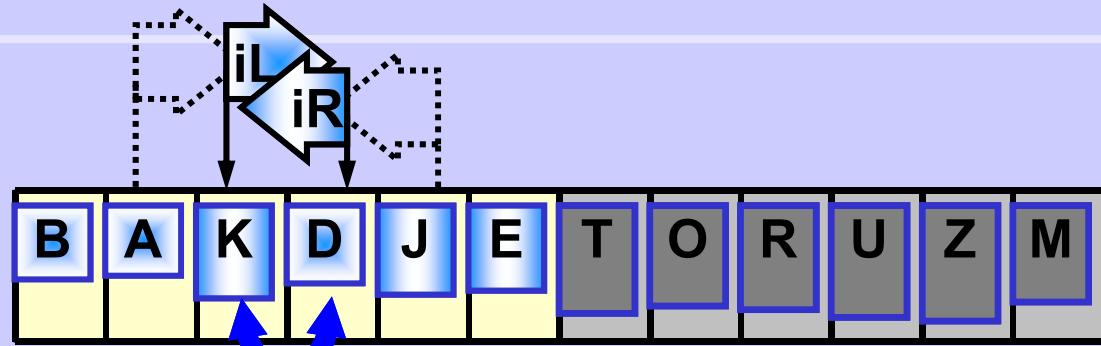
Pivot



# Quicksort

## Dělení

Pivot



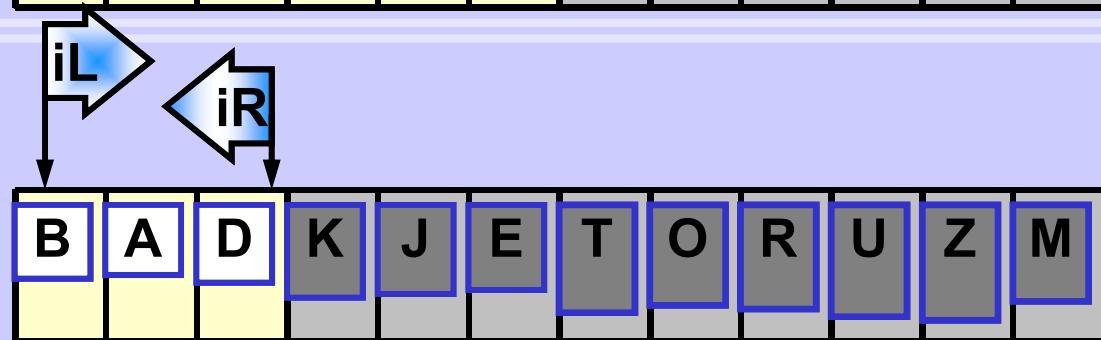
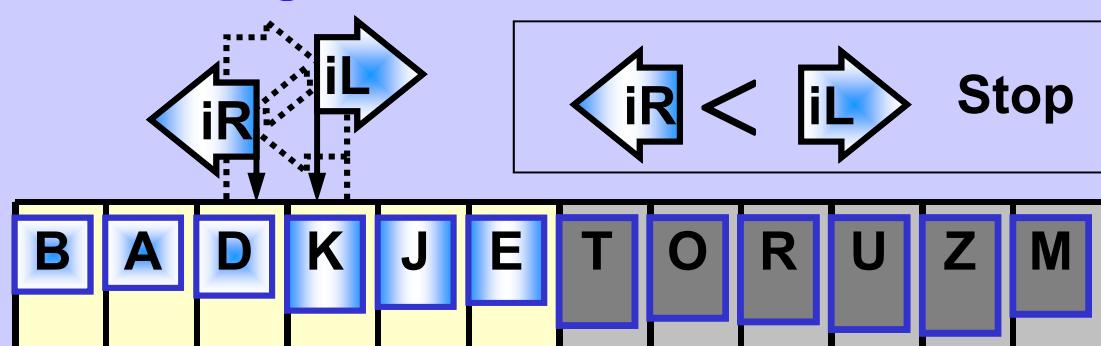
Krok 2

Rozděl!

Pivot



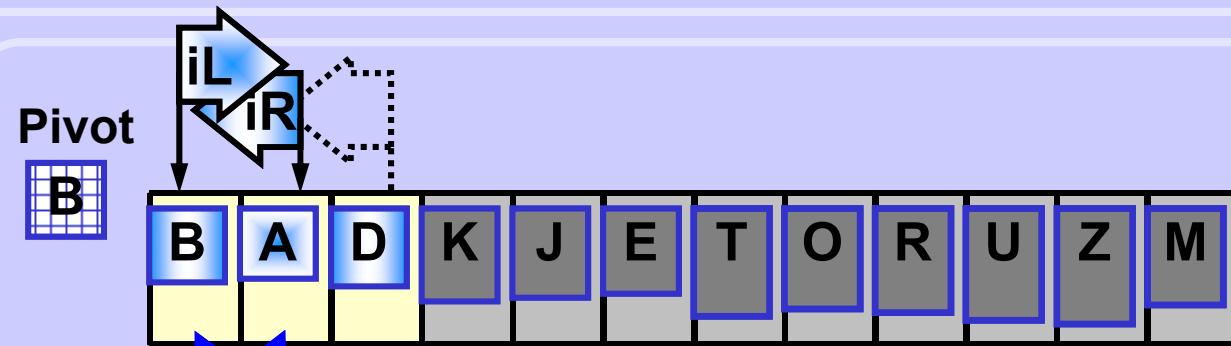
Init



# Quicksort

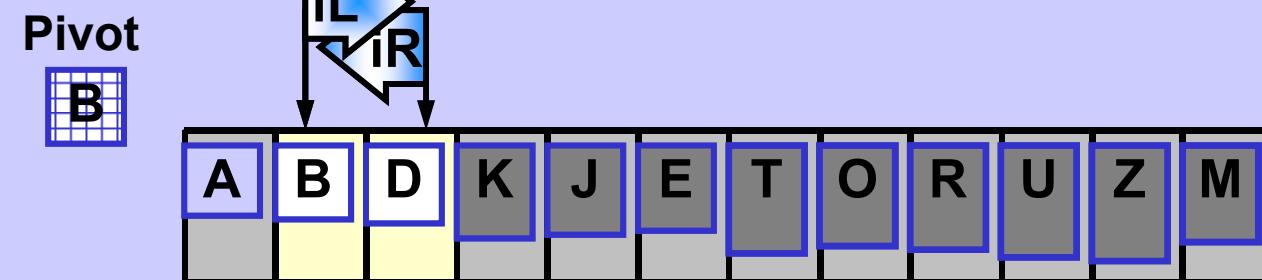
## Dělení

Krok 1



Rozděl!

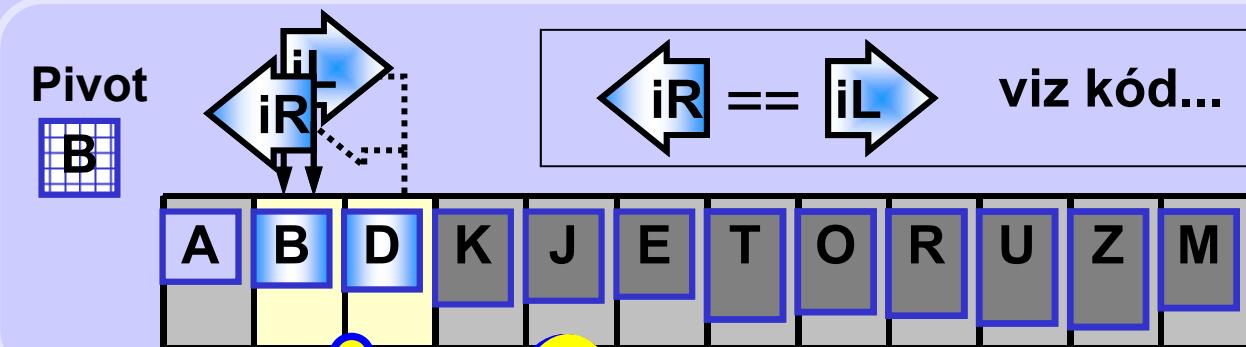
Init



# Quicksort

## Dělení

Krok 1

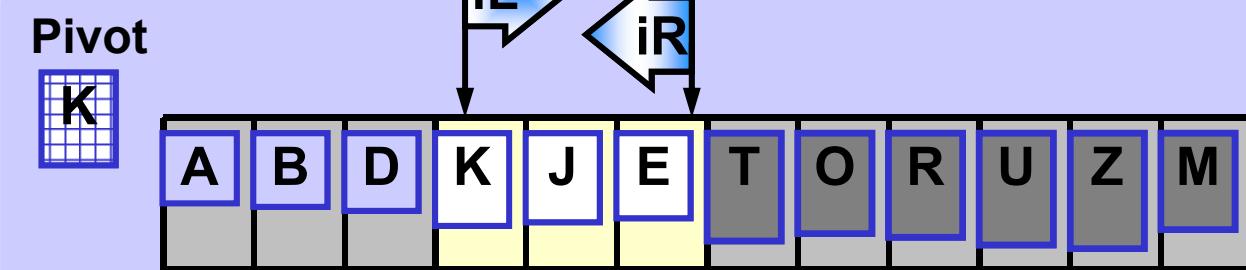


Další  
oddíl

Init

atd...

atd...



## Quicksort

```

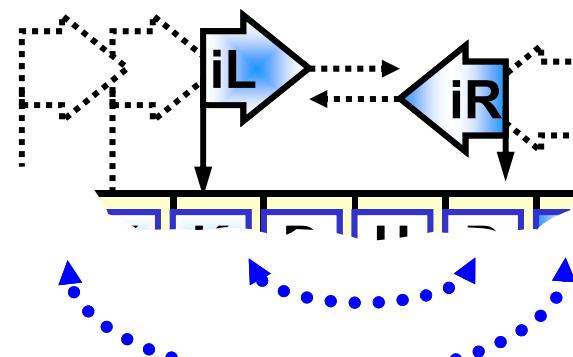
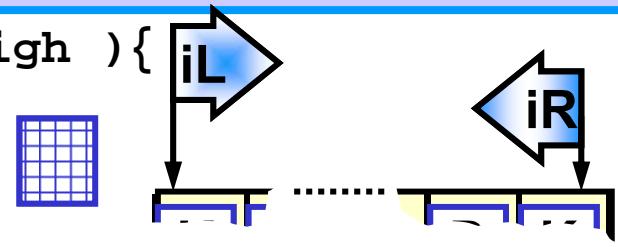
void qSort( Item a[], int low, int high ){
    int iL = low, iR = high;
    Item pivot = a[low];

    do {
        while( a[iL] < pivot ) iL++;
        while( a[iR] > pivot ) iR--;
        if( iL < iR ) {
            swap(a,iL, iR);
            iL++; iR--;
        }
        else
            if( iL == iR ) {iL++; iR--;}

    } while( iL <= iR );

    if( low < iR ) qSort( a, low, iR );
    if( iL < high ) qSort( a, iL, high );
}

```



Rozděl!

## Quicksort

Levý index se nastaví na začátek zpracovávaného úseku pole, pravý na jeho konec, zvolí se pivot.

Cyklus (rozdělení na „malé“ a „velké“) :

Levý index se pohybuje doprava

a zastaví se na prvku větším nebo rovném pivotovi.

Pravý index se pohybuje doleva

a zastaví se na prvku menším nebo rovném pivotovi.

Pokud je levý index ještě před pravým,

příslušné prvky se prohodí,

a oba indexy se posunou o 1 ve svém směru.

Jinak pokud se indexy rovnají,

jen se oba posunou o 1 ve svém směru.

Cyklus se opakuje, dokud se indexy neprekříží,

tj. pravý se dostane pred levého.

Následuje rekurzivní volání (zpracování „malých“ a „velkých“ zvlášt')

na úsek od začátku do pravého(!) indexu včetně

a na úsek od levého(!) indexu včetně až do konce,

má-li příslušný úsek délku větší než 1.

## Quicksort

### Asymptotická složitost

Celkem  
přesunů a testů

$\Theta(n \cdot \log_2(n))$  nejlepší případ

$\Theta(n \cdot \log_2(n))$  průměrný případ

$\Theta(n^2)$  nejhorší případ

Asymptotická složitost Quick Sortu je  $O(n^2)$ , ...

... ale! :

“Očekávaná” složitost Quick Sortu je  $\Theta(n \cdot \log_2(n))$  (!!)

## Quicksort



## Porovnání efektivity



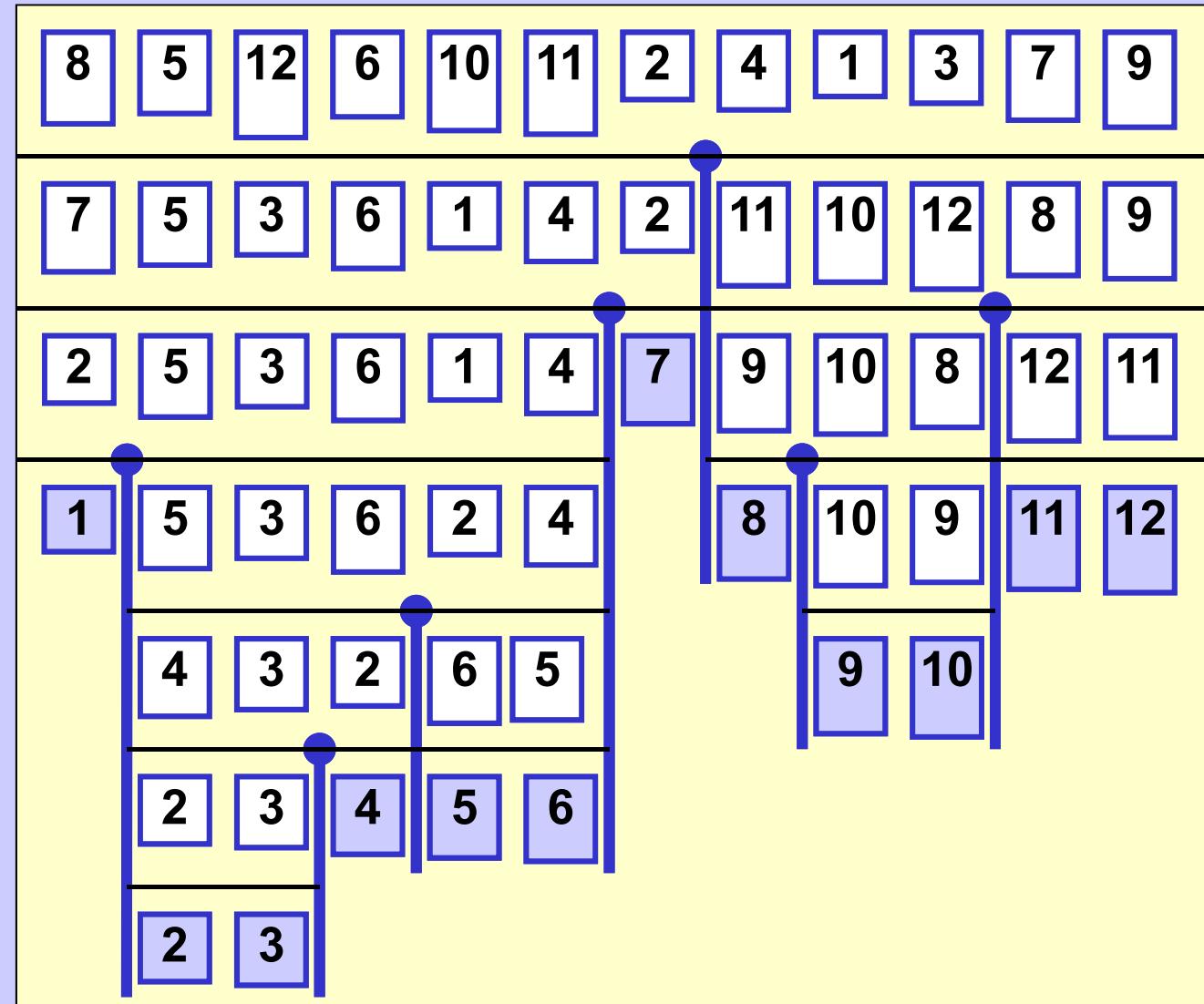
$N$	$N^2$	$N \times \log_2(N)$	$\frac{N^2}{N \times \log_2(N)}$	zpoma- lení (1~1sec)
1	1	0		
10	100	33.2	3.0	3 sec
100	10 000	6 64.4	15.1	15 sec
1 000	1 000 000	9 965.8	100.3	1.5 min
10 000	100 000 000	132 877.1	752.6	13 min
100 000	10 000 000 000	1 660 964.0	6 020.6	1.5 hod
1 000 000	1 000 000 000 000	19 931 568.5	50 171.7	14 hod
10 000 000	100 000 000 000 000	232 534 966.6	430 042.9	5 dnů

tab. 1

## Quicksort

Ukázka  
průběhu

**pivot =**  
**= první**  
**v úseku**



## Quicksort

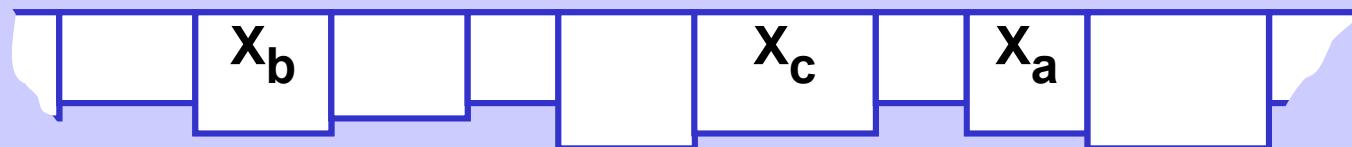


Schéma rekurzivního volání  
Quick sortu  
má podobu pravidelného  
binárního kořenového stromu.

## Stabilita řazení

**Stabilní řazení nemění pořadí prvků se stejnou hodnotou.**

Neseřazená data

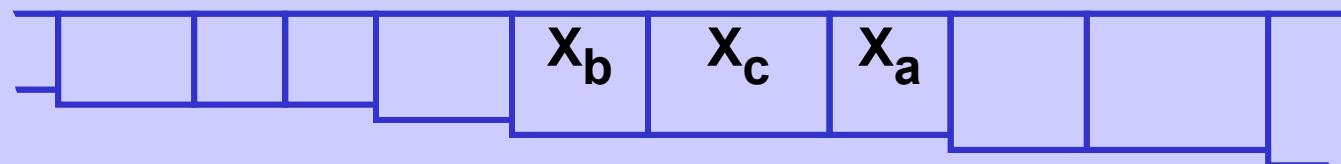


Hodnoty  $X_i$  jsou totožné



Seřad'

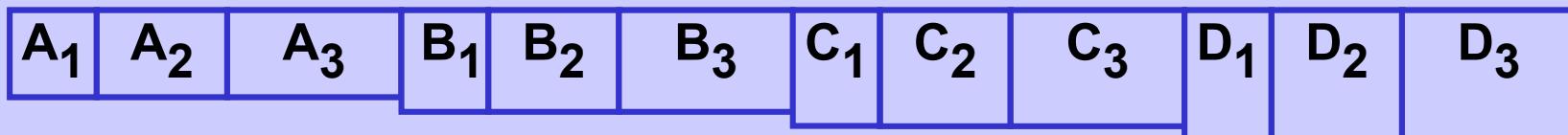
Seřazená data



## Stabilita řazení



Insert  
Bubble -- Stabilní  
implementace



Insert  
Bubble -- Nestabilní  
implementace



QuickSort      Vždy  
Select Sort      nestabilní!!



## Stabilní řazení

Záznam: **Jméno** | **Příjmení**

**Vstup:** Seznam seřazen pouze podle jména.

Andrew	Cook
Andrew	Amundsen
Andrew	Brown
Barbara	Cook
Barbara	Brown
Barbara	Amundsen
Charles	Amundsen
Charles	Cook
Charles	Brown

stabilní řazení  
 →  
 Seřad' záznamy  
 pouze podle"  
**Příjmení**

**Výstup:** Seznam seřazen podle jména i příjmení.

Andrew	Amundsen
Barbara	Amundsen
Charles	Amundsen
Andrew	Brown
Barbara	Brown
Charles	Brown
Andrew	Cook
Barbara	Cook
Charles	Cook

Pořadí záznamů se stejným příjmením se nezměnilo

## Nestabilní řazení

Záznam: **Jméno** **Příjmení**

**Vstup:** Seznam seřazen pouze podle jména.

Andrew	Cook
Andrew	Amundsen
Andrew	Brown
Barbara	Cook
Barbara	Brown
Barbara	Amundsen
Charles	Amundsen
Charles	Cook
Charles	Brown

**Výstup:** Původní pořadí jmen je ztraceno.

Barbara	Amundsen
Andrew	Amundsen
Charles	Amundsen
Barbara	Brown
Charles	Brown
Andrew	Brown
Charles	Cook
Andrew	Cook
Barbara	Cook

QuickSort

Seřad' záznamy pouze podle:  
Příjmení

Pořadí záznamů se stejným příjmením se změnilo.

## ALG 08

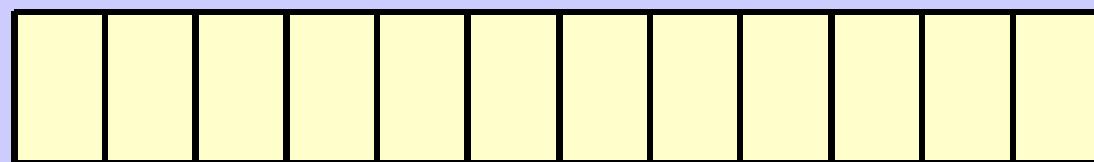
**Merge sort** -- řazení sléváním

**Heap Sort** -- řazení binární haldou

**Prioritní fronta** implementovaná binární haldou

## Merge sort

Sluč (slij?) dvě seřazená pole



Porovnávané prvky

B	K	M	T	U	Z
A	D	E	J	O	R

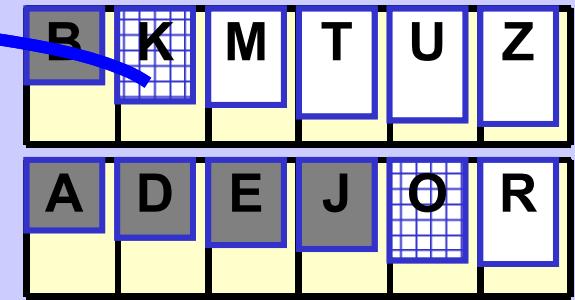
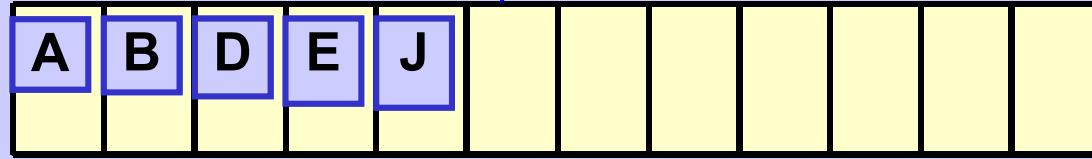
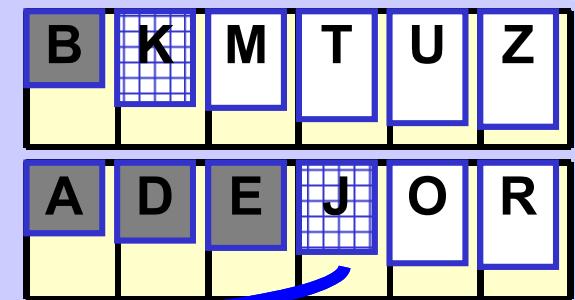
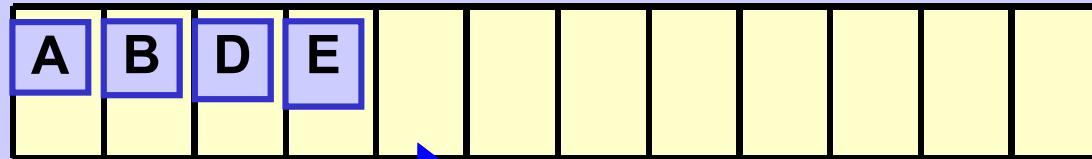
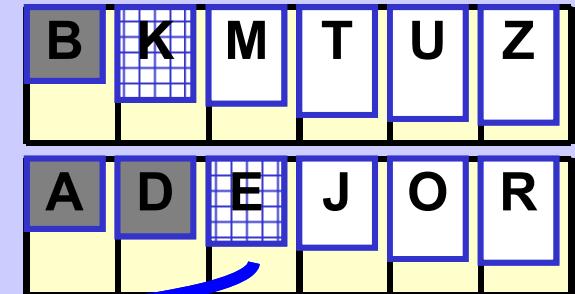
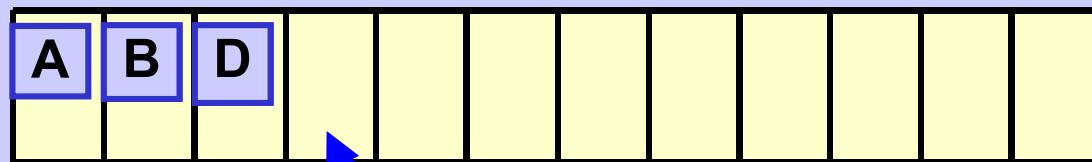
B	K	M	T	U	Z
A	D	E	J	O	R

B	K	M	T	U	Z
A	D	E	J	O	R

B	K	M	T	U	Z
A	D	E	J	O	R

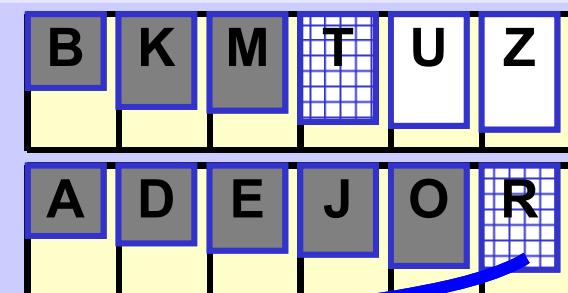
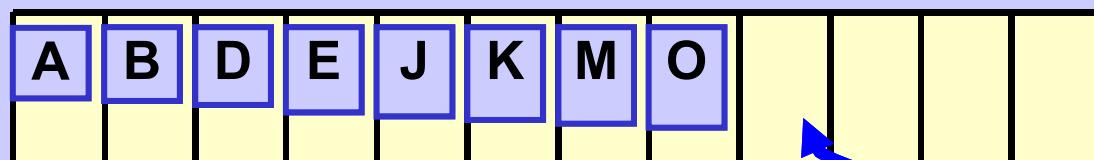
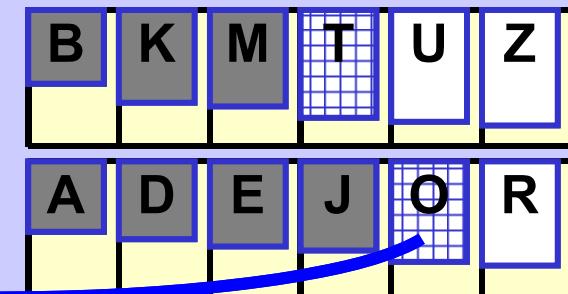
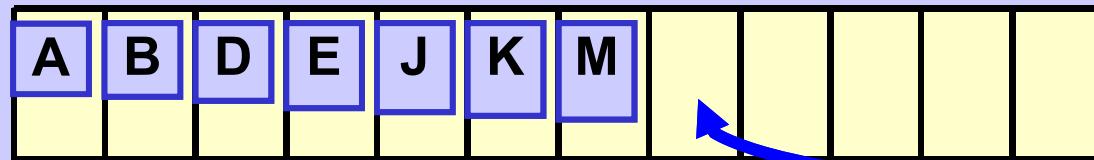
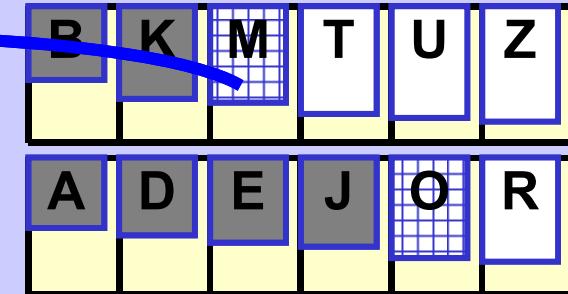
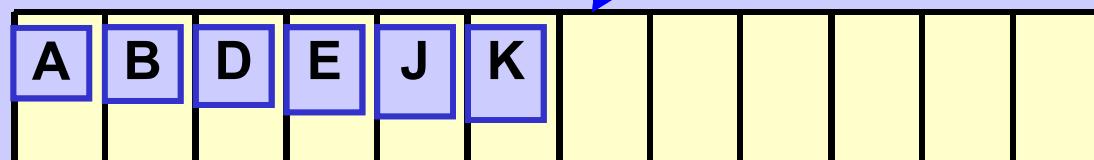
## Merge sort

Sluč dvě seřazená pole - pokr.



## Merge sort

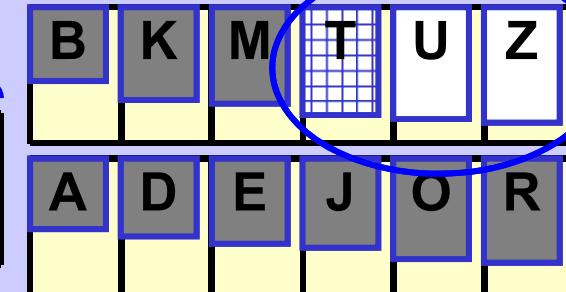
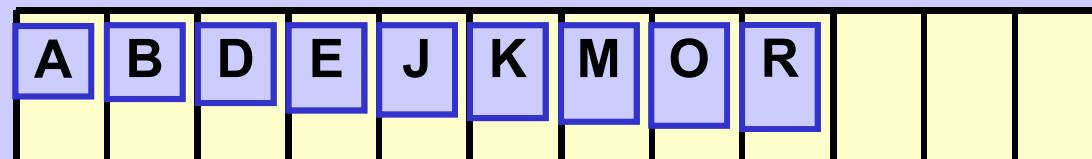
Sluč dvě seřazená pole - pokr.



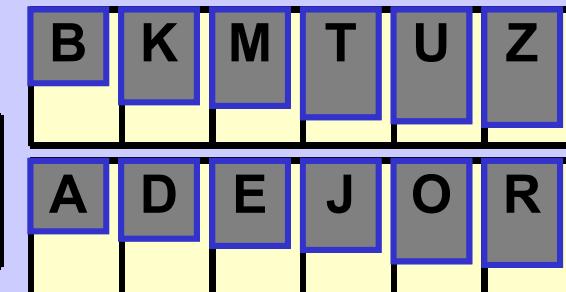
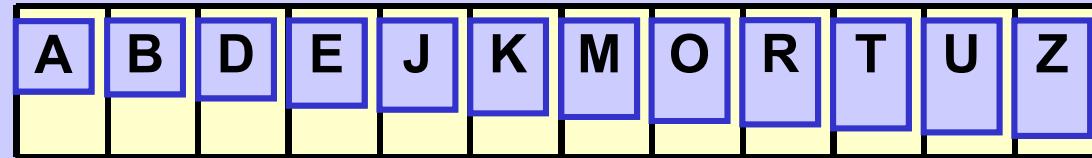
## Merge sort

Sluč dvě seřazená pole - pokr.

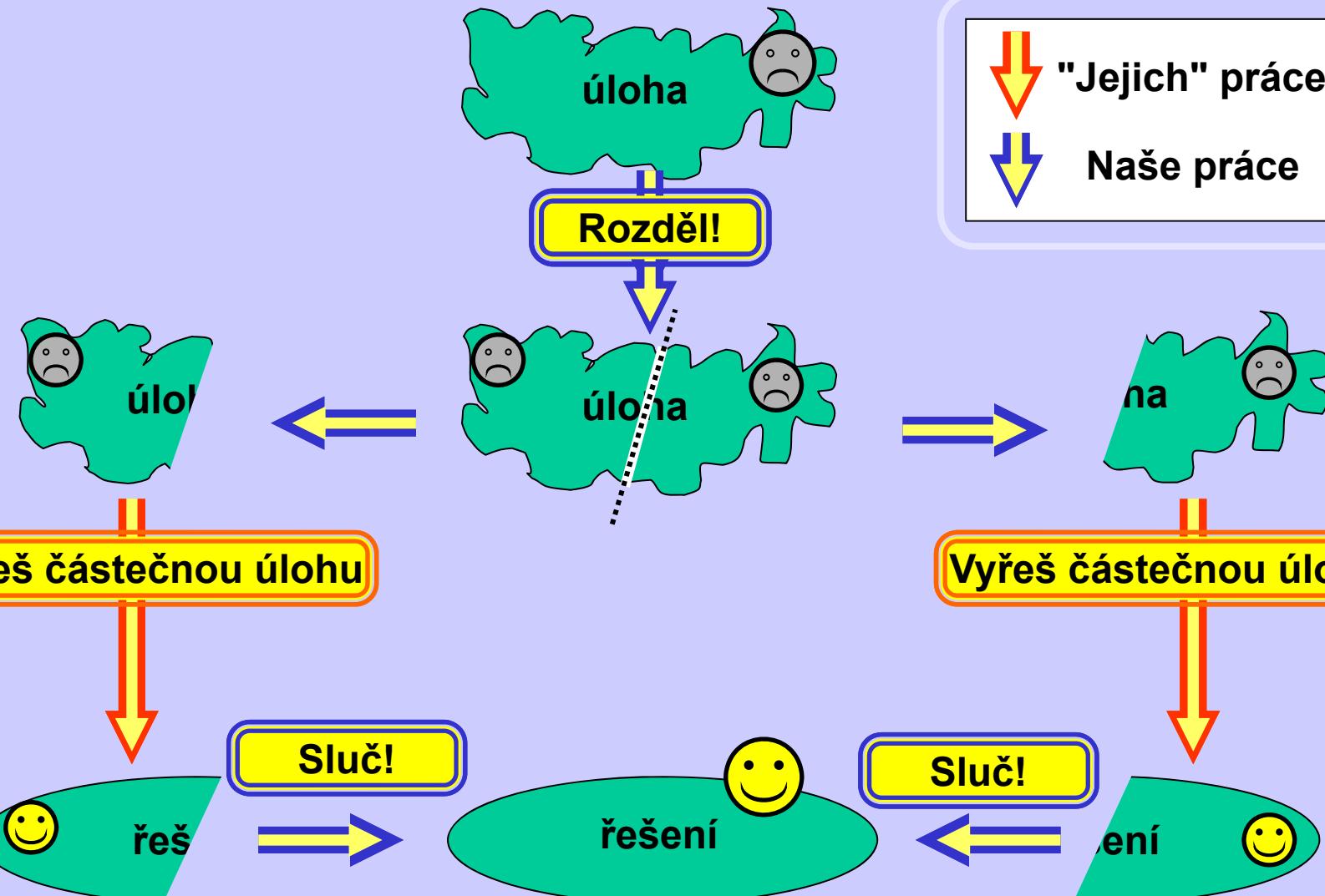
Kopíruj zbytek



Seřazeno



# Rozděl a panuj! Divide and conquer! Divide et impera!



## Merge sort

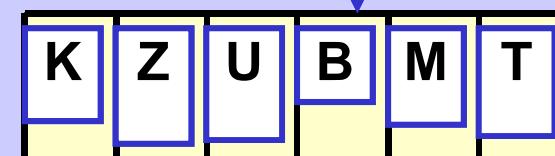
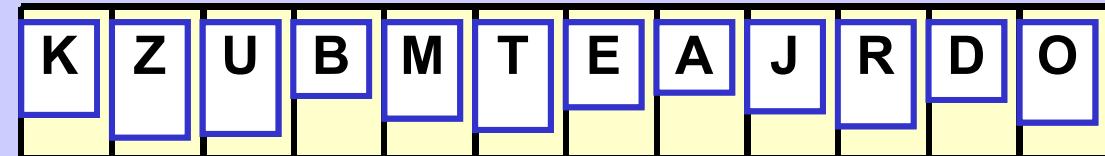
Neseřazeno

Rozděl!

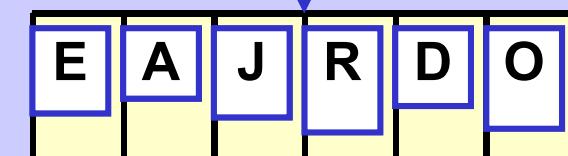
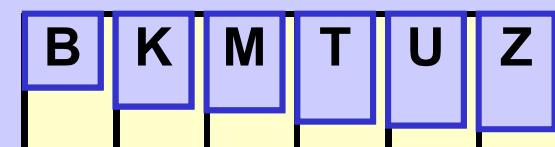
Zpracuj  
odděleně

Panuj!

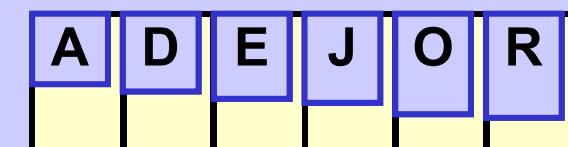
Seřazeno



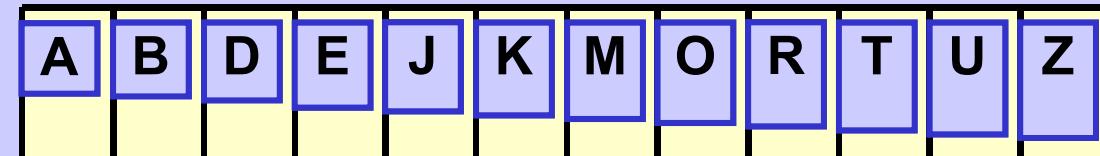
seřad'!



seřad'!



Sluč!



## Merge sort

Neseřazeno

Rozděl!

Rozděl!

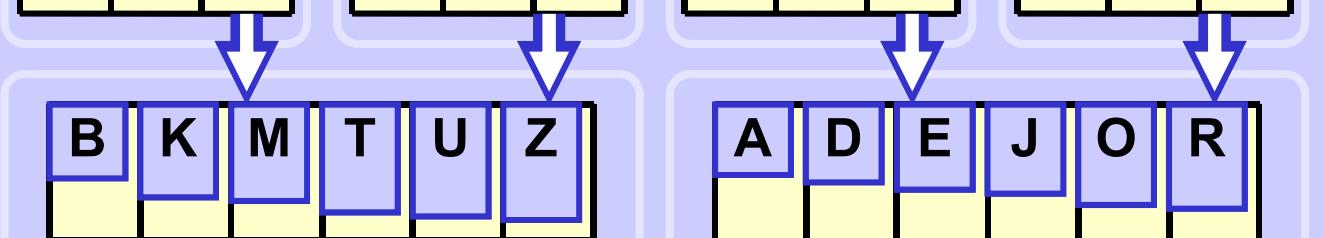
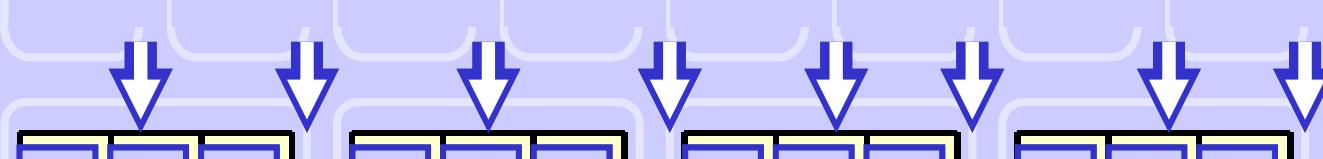
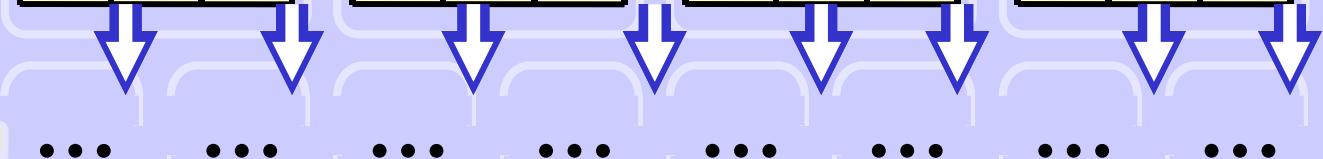
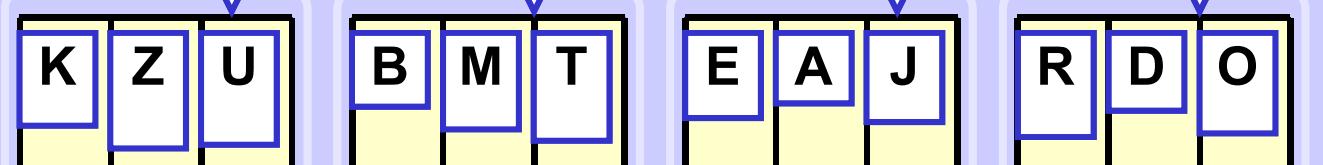
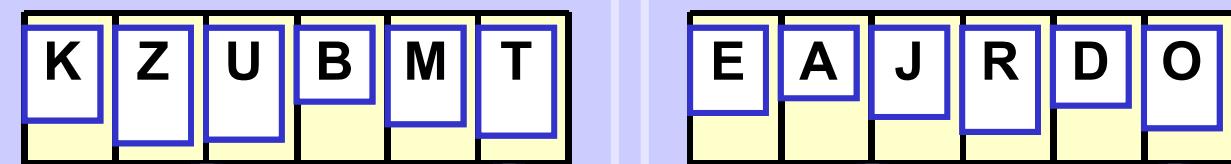
Rozděl!

Sluč!

Sluč!

Sluč!

Seřazeno



## Merge sort

```
void merge( int [] in, int [] out, int low, int high ){
    int half = (low+high)/2;
    int i1 = low;
    int i2 = half+1;
    int j = low;

                                // compare and merge
    while( (i1 <= half) && (i2 <= high) ){
        if( in[i1] <= in[i2] ){ out[j] = in[i1]; i1++; }
        else { out[j] = in[i2]; i2++; }
        j++;
    }
                                // copy the rest
    while( i1 <= half ){ out[j] = in[i1]; i1++; j++; }
    while( i2 <= high ){ out[j] = in[i2]; i2++; j++; }
}
```

## Merge sort

```
void mergeSort( int [] a, int [] aux,
                int low, int high ){
    int half = (low+high)/2;
    if( low >= high ) return; // too small!

    // sort
    mergeSort( a, aux, low, half ); // left half
    mergeSort( a, aux, half+1, high ); // right half
    merge( a, aux, low, high ); // merge halves

    // (*) put result back to a -- clumsy method!
    for( int i = low; i <= high; i++ ) a[i] = aux[i];

    // (*) better solution: swap references
    // to a and aux in even depths of recursion
}
```

## Merge sort - optimalizované využití pomocného pole

```
void mergeSort( int [] a ) {  
    int [] aux = Arrays.copyOf( a, a.length );  
    mergeSort(a, aux, 0, a.length-1, 0 );  
}  
  
void mergeSort( int [] a, int [] aux,  
                int low, int high, int depth ){  
    int half = (low+high)/2;  
    if( low >= high ) return;  
  
    mergeSort( a, aux, low, half, depth+1 );  
    mergeSort( a, aux, half+1, high, depth+1 );  
  
    // note the exchange of a and aux  
    if( d % 2 == 0 ) merge( aux, a, low, high );  
    else  
        merge( a, aux, low, high );  
}
```

## Merge sort

### Asymptotická složitost

Rozděl! .....  $\log_2(n)$  krát  $\Rightarrow$

$\Rightarrow$  Sluč! .....  $\log_2(n)$  krát

Rozděl! .....  $\Theta(1)$  operací

Sluč! .....  $\Theta(n)$  operací

---

Celkem .....  $\Theta(n) \cdot \Theta(\log_2(n)) = \Theta(n \cdot \log_2(n))$  operací

Asymptotická složitost Merge sortu je  $\Theta(n \cdot \log_2(n))$

## Merge sort

### Stabilita

Rozděl! ..... Nepohybujе prvky

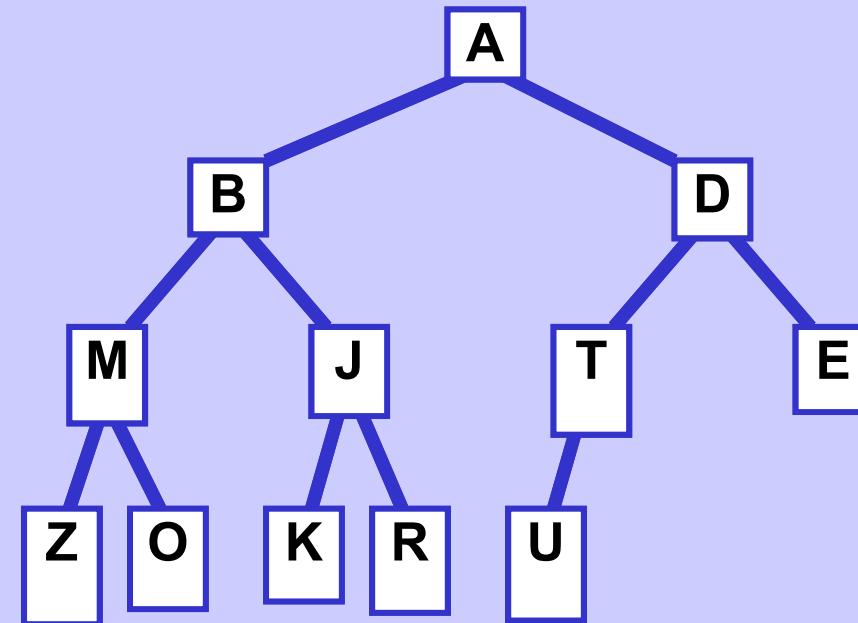
Sluč! ..... “ if ( $\text{in}[i1] \leq \text{in}[i2]$ ) {  $\text{out}[j] = \text{in}[i1]$ ; ... ”

Zařad' nejprve levý prvek,  
když slučuješ stejné hodnoty

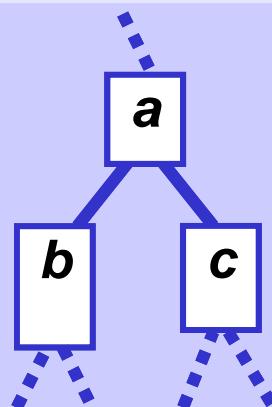
**MergeSort je stabilní**

## Heap sort

Halda



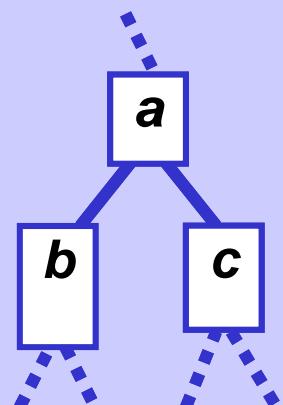
Pravidlo  
haldy



$$a \leq b \quad \&\& \quad a \leq c$$

## Heap sort

### Terminologie



**a** ..... predecessor, parent of **b** **c**

..... předchůdce, rodič

**b**, **c** ..... successor, child of **a**

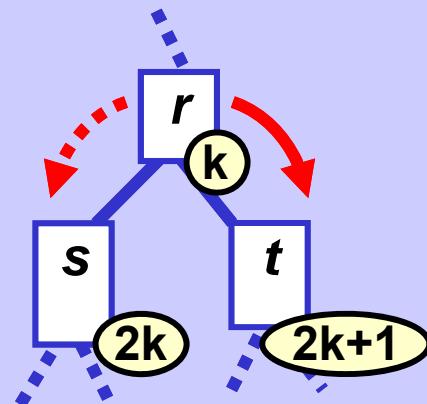
..... následník, potomek



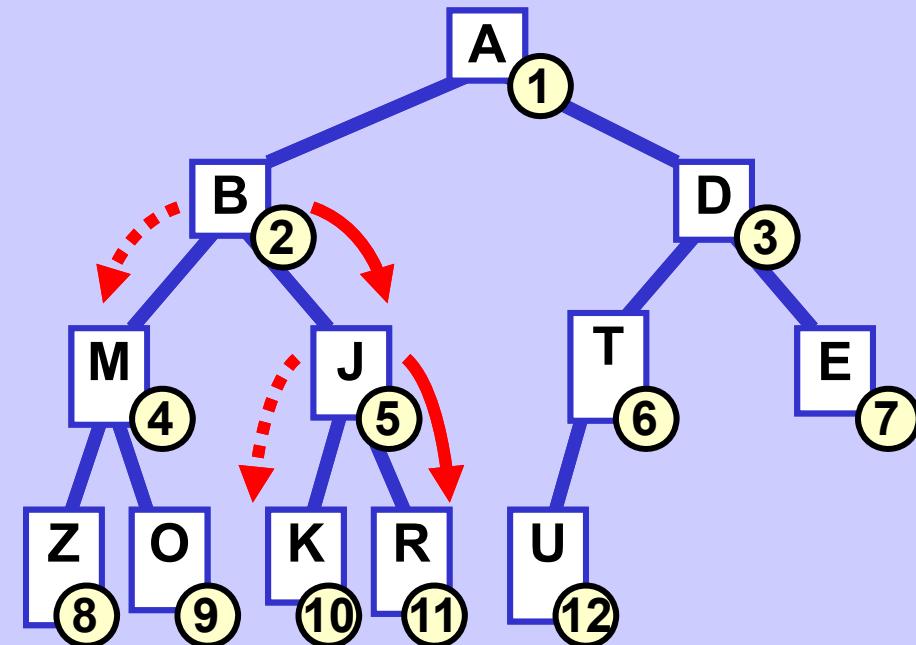
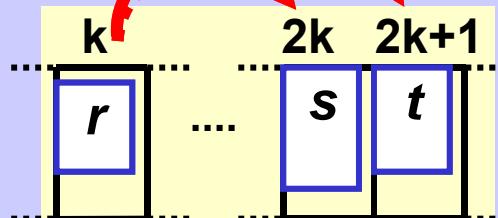
..... (heap) top ..... vrchol (haldy)

## Heap sort

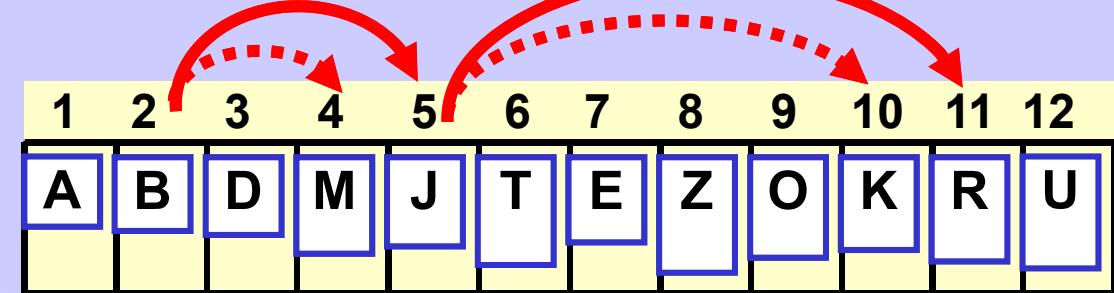
### Halda uložená v poli



následníci



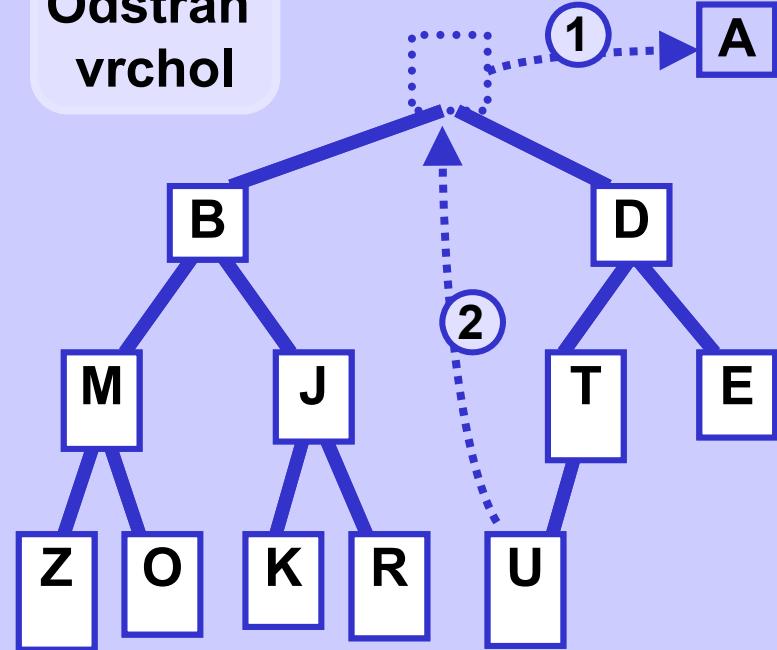
následníci



## Oprava haldy

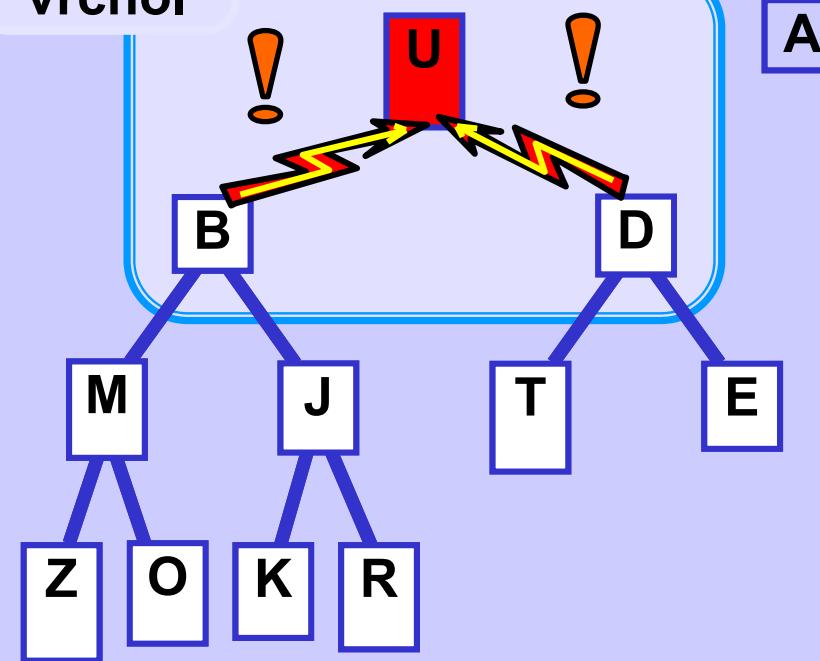
### Vrchol odstraněn (1)

① Odstraň vrchol



② poslední → první

③ Vlož na vrchol

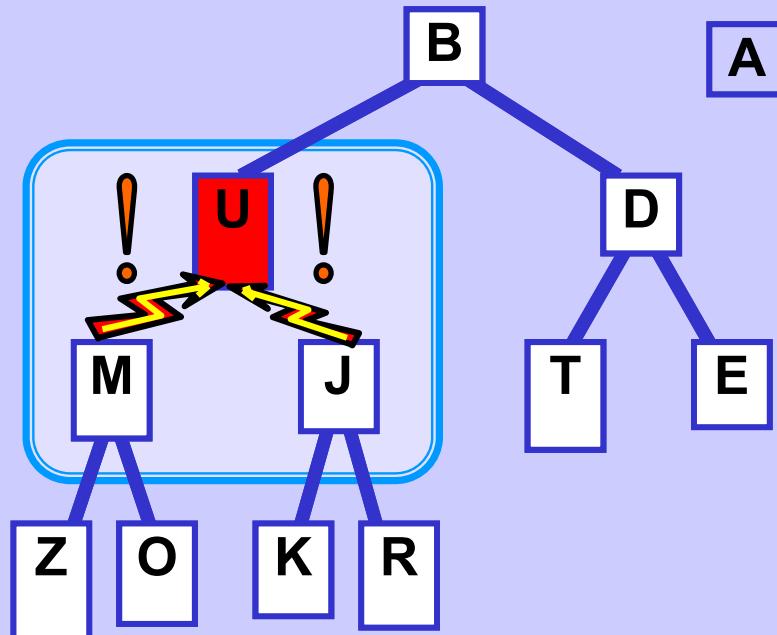


$U > B, U > D, \underline{B < D}$   
 $\Rightarrow$  prohod'  $B \leftrightarrow U$

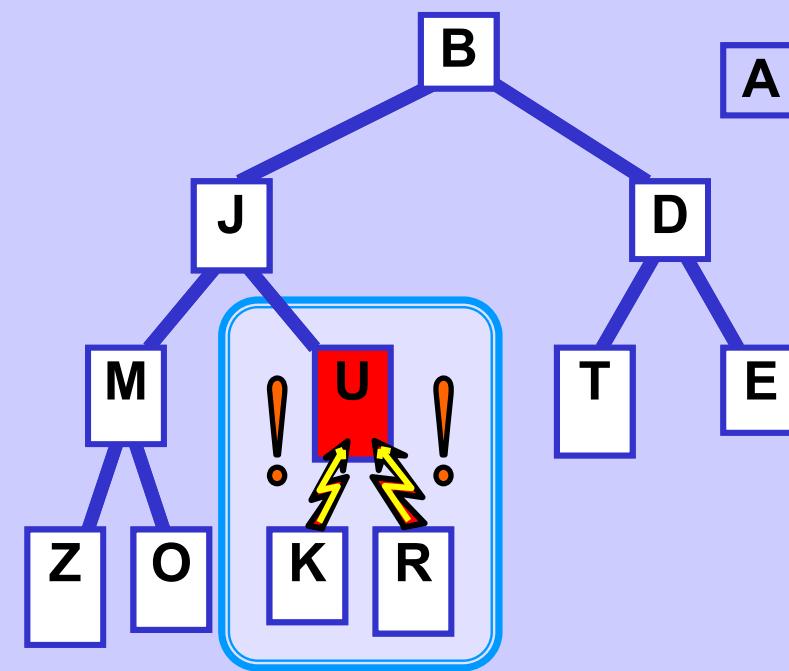
## Oprava haldy

### Vrchol odstraněn (2)

③ Vlož na vrchol - pokračování



$U > M, U > J, \underline{J < M}$   
 $\Rightarrow$  prohod'  $J \leftrightarrow U$



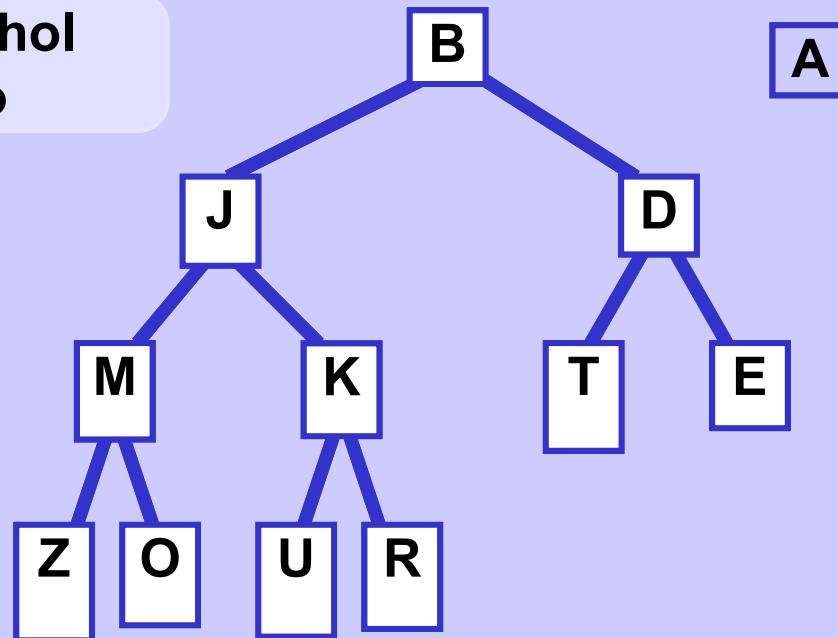
$U > K, U > R, \underline{K < R}$   
 $\Rightarrow$  prohod'  $K \leftrightarrow U$

## Oprava haldy

### Vrchol odstraněn (3)

③

Vlož na vrchol  
- hotovo

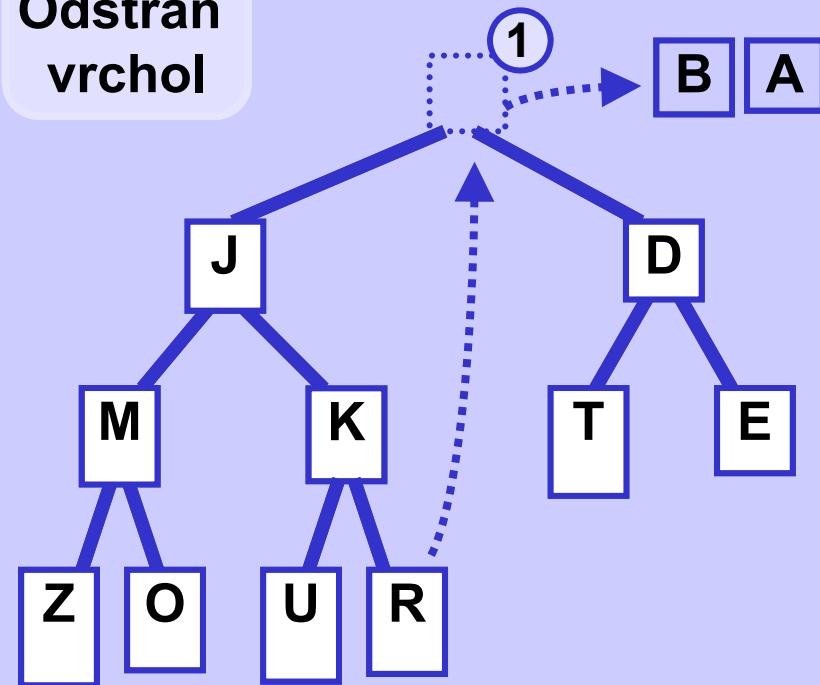


Nová halda

## Oprava haldy

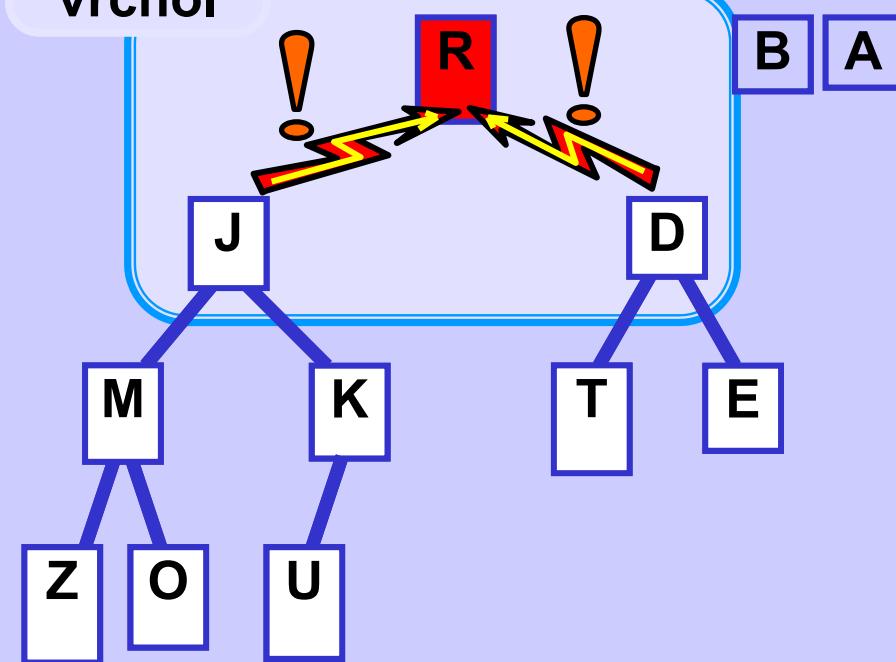
### Vrchol odstraněn II (1)

① Odstraň vrchol



② poslední → první

③ Vlož na vrchol

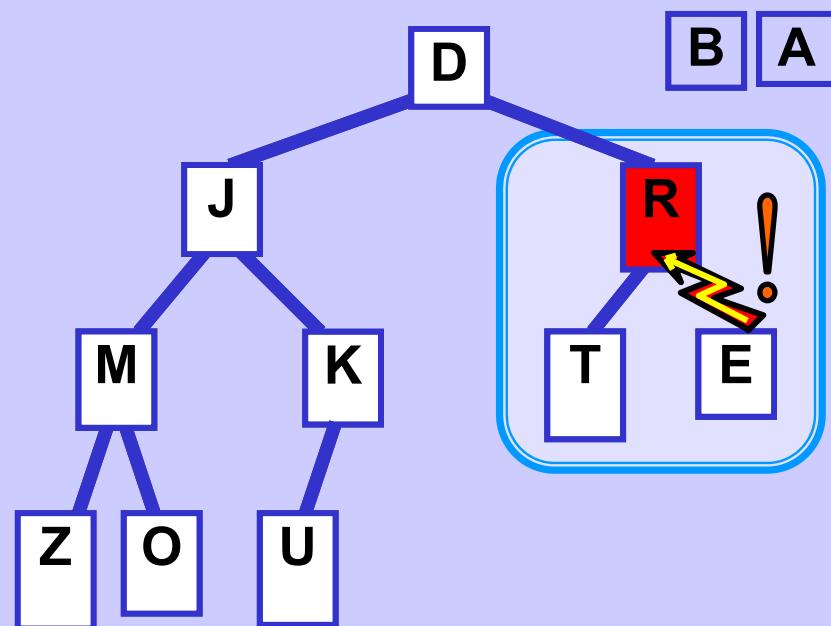


$R > J, R > D, \underline{D < J}$   
 $\Rightarrow$  prohod'  $D \leftrightarrow R$

## Oprava haldy

### Vrchol odstraněn II (2)

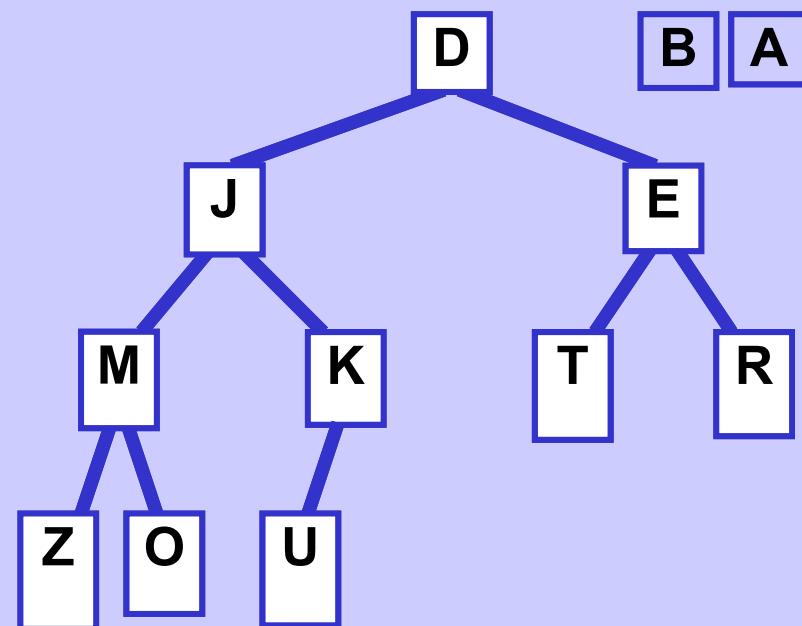
③ Vlož na vrchol - pokračování



$R < T, R > E$   
 $\Rightarrow \text{prohod'} E \leftrightarrow R$

### Vrchol odstraněn II (3)

③ Vlož na vrchol  
- hotovo

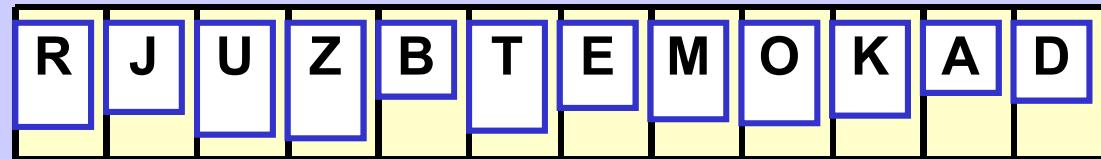


Nová hala

## Heap sort

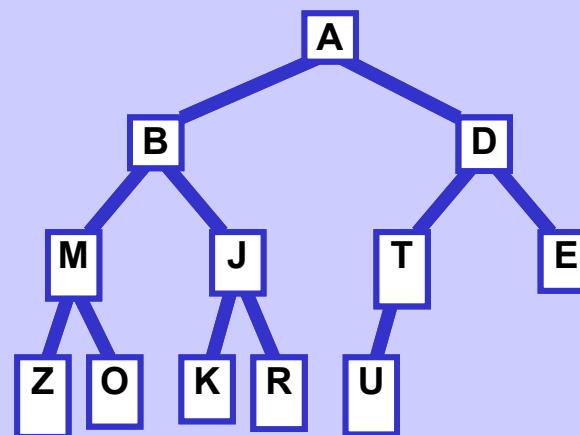
I

Neseřazeno



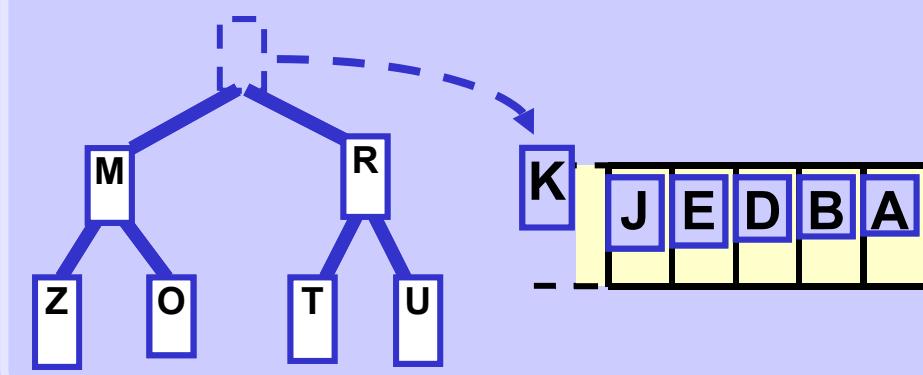
II

Vytvoř haldu



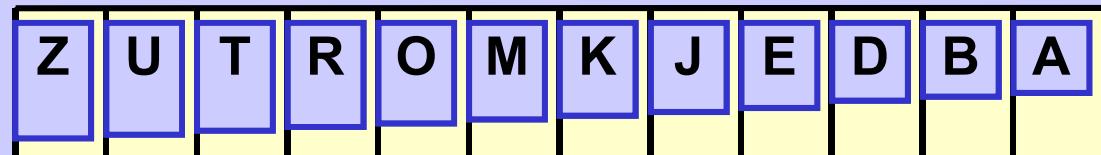
III

`for (i = 1; i <= n; i++)  
 "odstraň vrchol";`

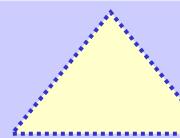
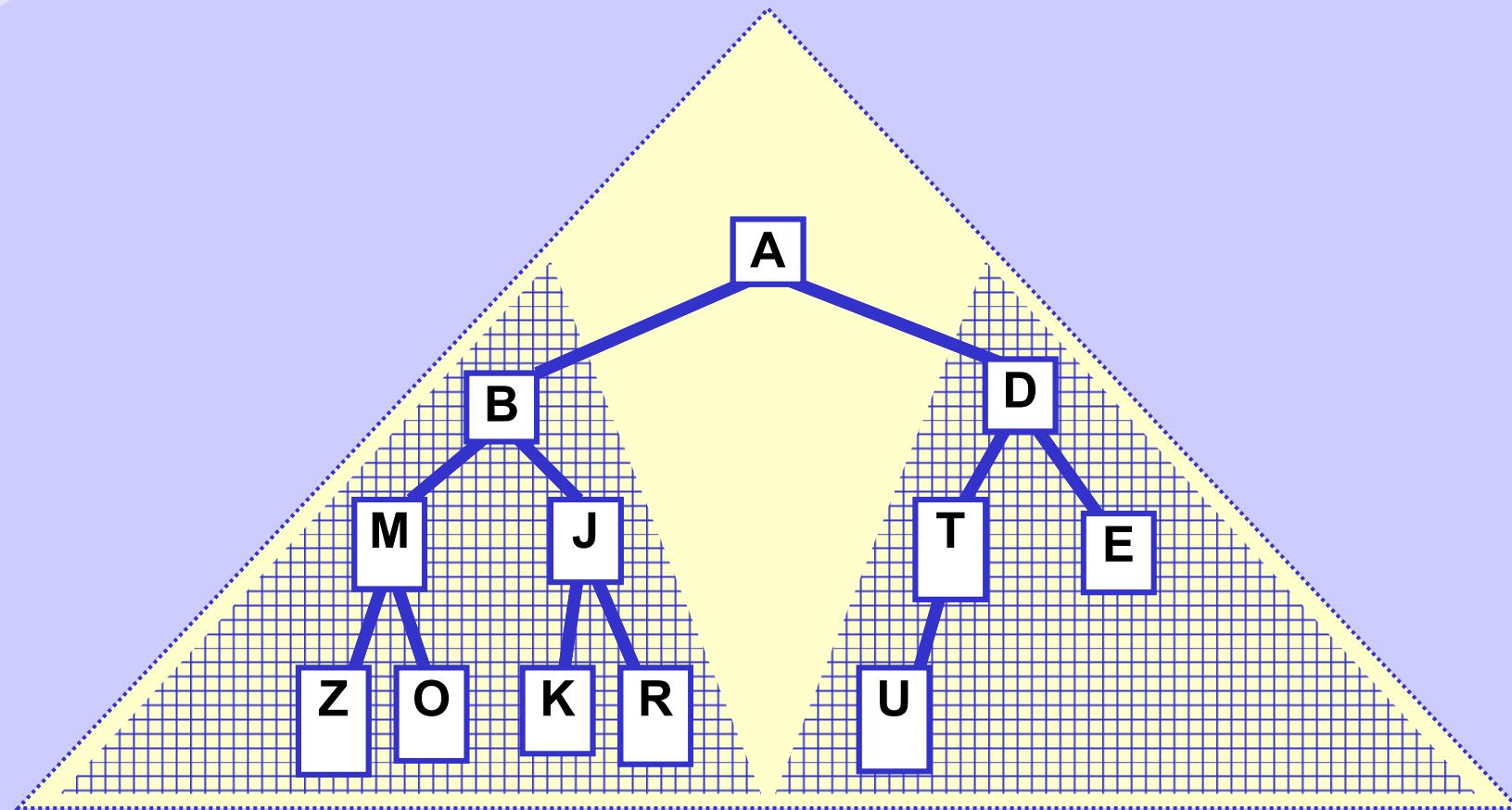


IV

Seřazeno



## Rekurzivní vlastnost "býti haldou"



je halda

$\Rightarrow$

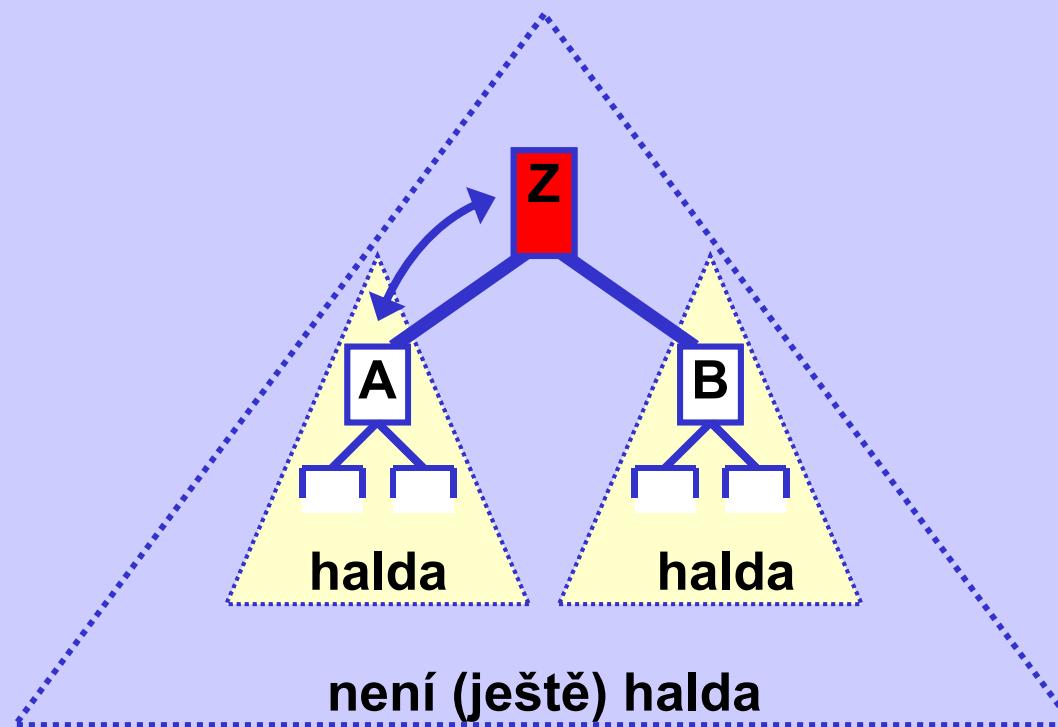


je halda a



je halda

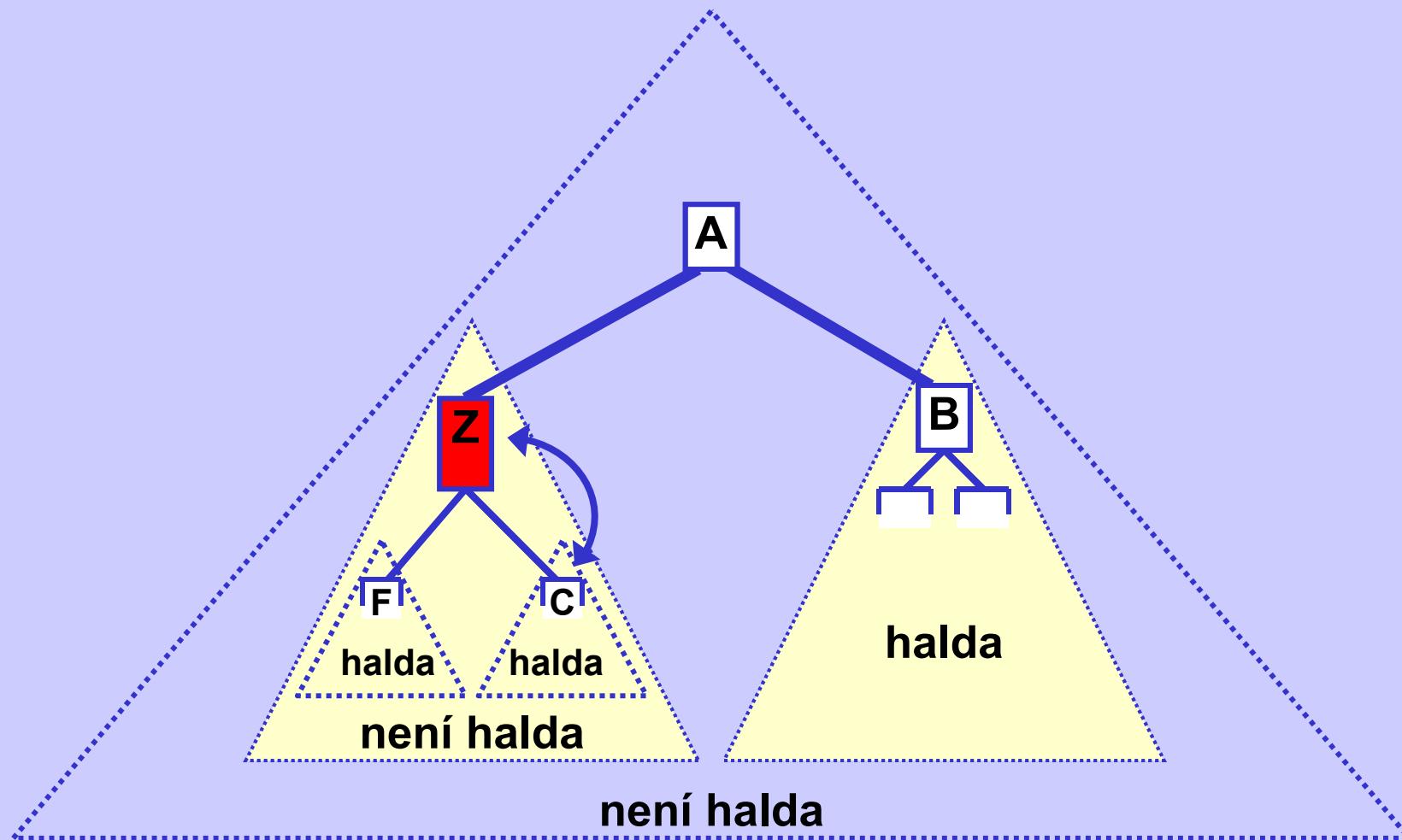
## Vytvoř jednu haldu ze dvou menších



$Z > A$  nebo  $Z > B$

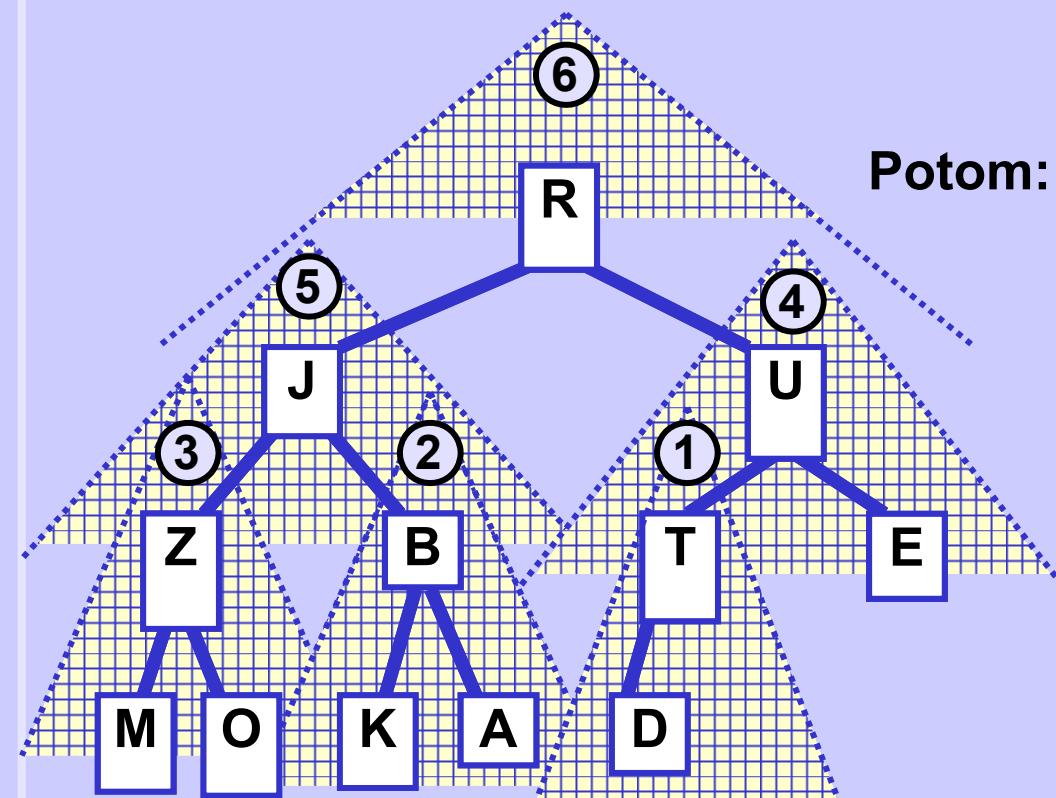
$\Rightarrow$  prohod':  $Z \leftrightarrow \min(A, B)$

## Vytvoř jednu haldu ze dvou menších



## Vytvoř haldu

### Stavba haldy zdola



Uvaž: **Každý list je samostanou (malinkou) haldou**

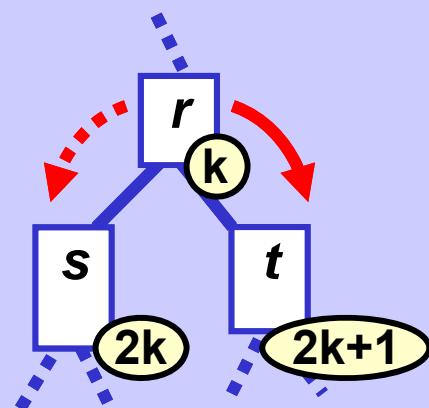
Potom:

- Vytvoř haldu v ① ...**
- ... a vytvoř haldu v ② ...**
- ... a vytvoř haldu v ③ ...**
- ... a vytvoř haldu v ④ ...**
- ... a vytvoř haldu v ⑤ ...**
- ... a vytvoř haldu v ⑥ ...**

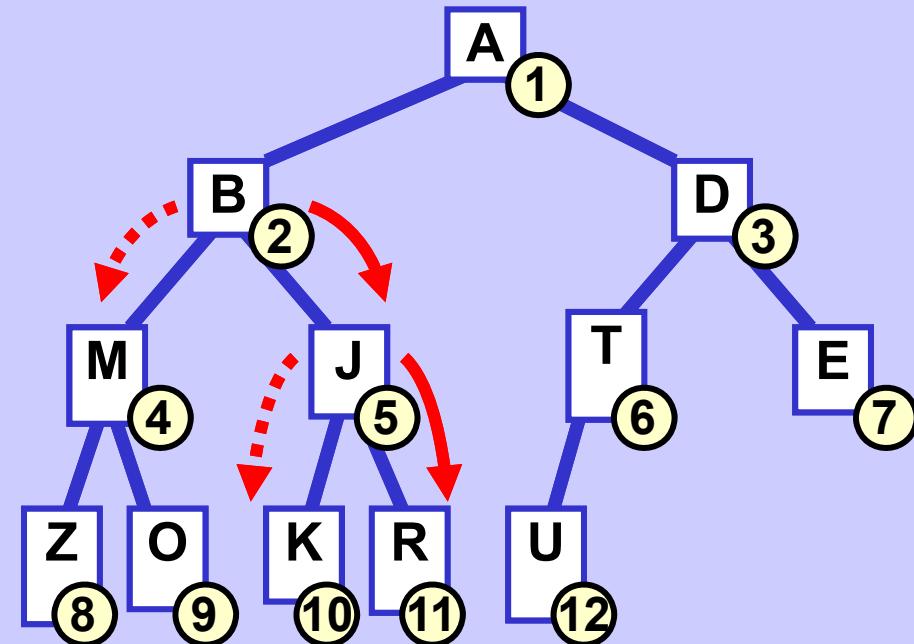
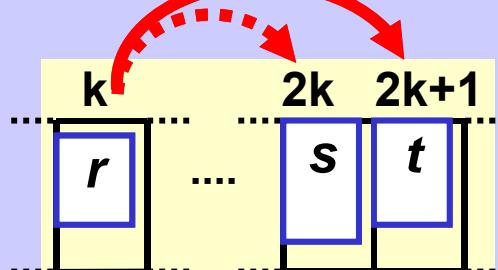
**... a celá halda je hotova.**

## Halda v poli

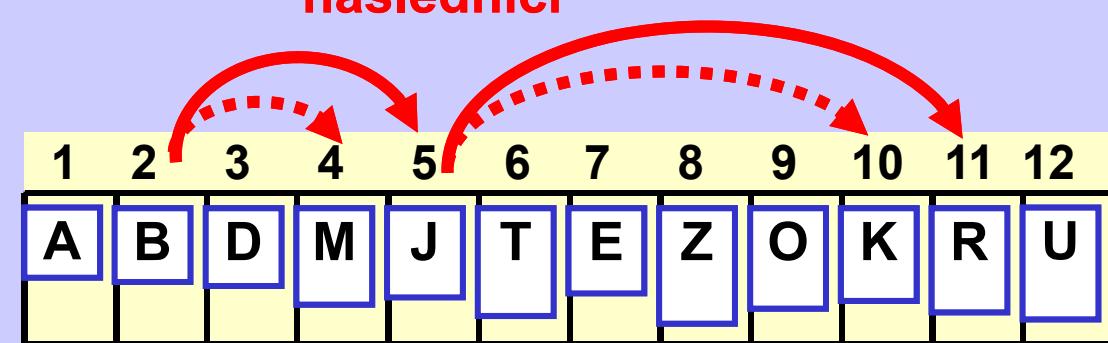
### Halda uložená v poli



následníci



následníci



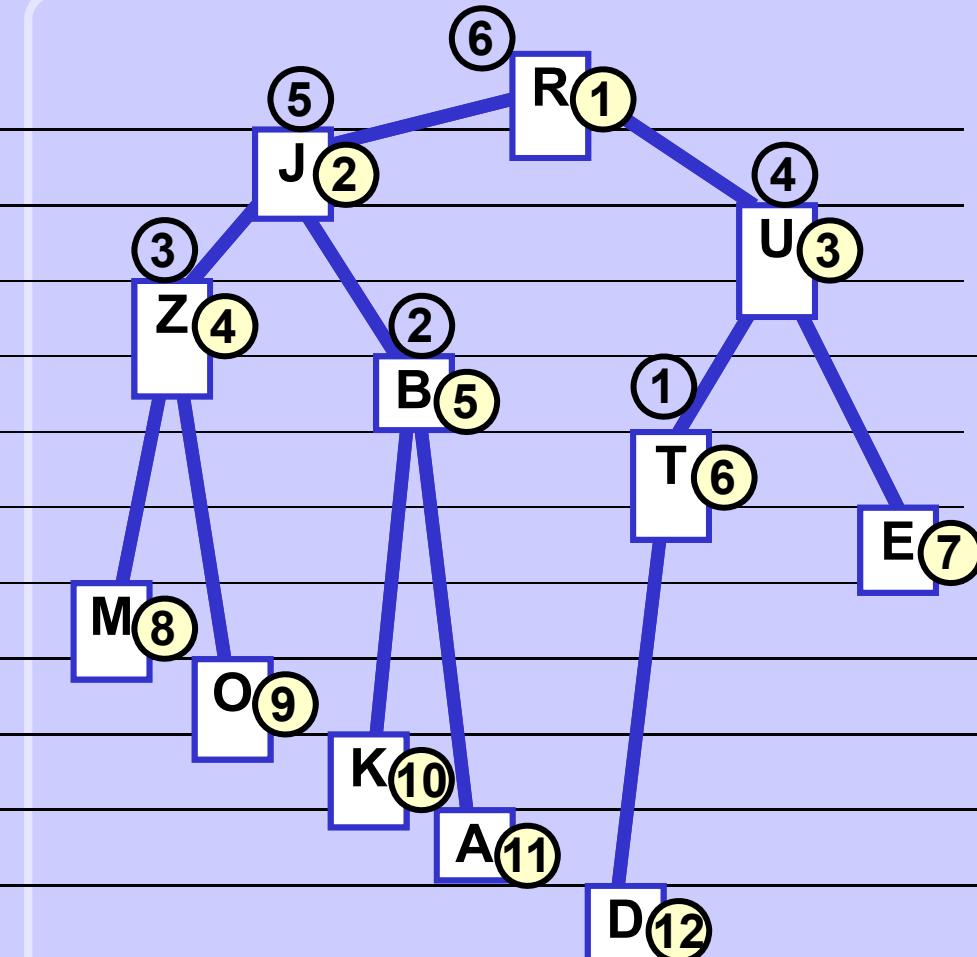
## Tvorba haldy v poli

### Pole

(6)	1	R
(5)	2	J
(4)	3	U
(3)	4	Z
(2)	5	B
(1)	6	T
	7	E
	8	M
	9	O
	10	K
	11	A
	12	D

Prvky náhodně uspořádány

Není halda



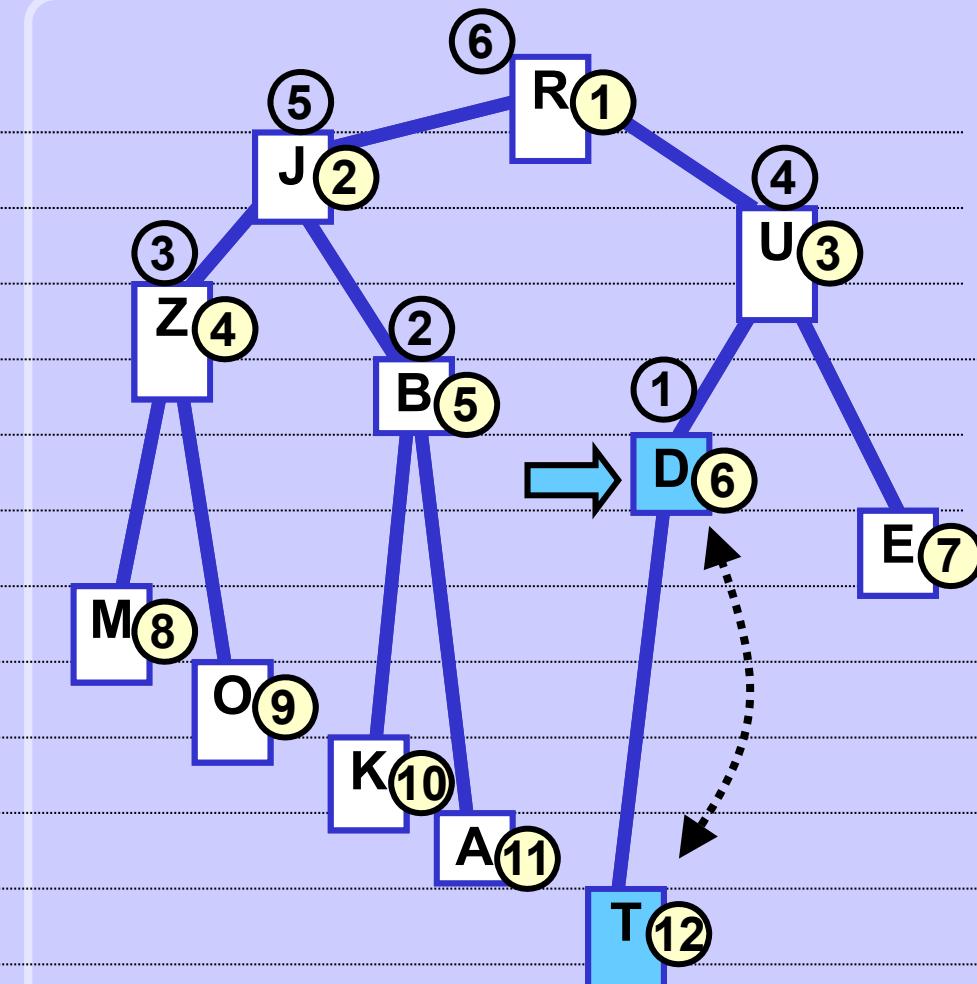
## Tvorba haldy v poli

### Pole

.....▼ Přesuny

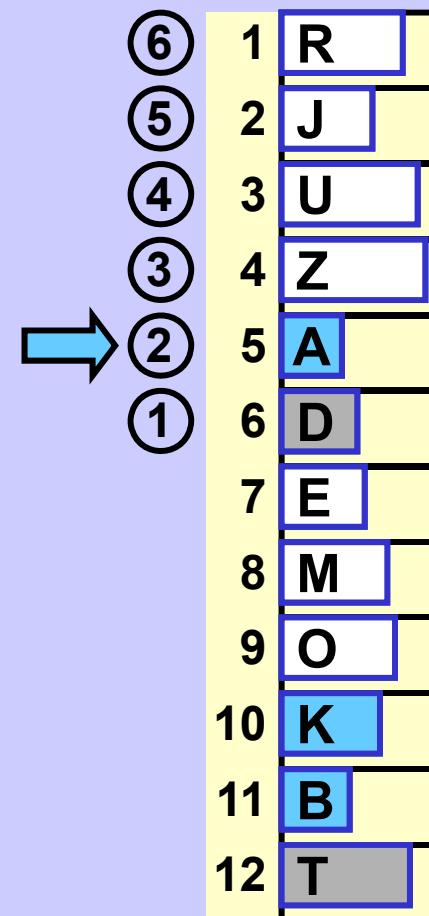
Aktuálně vytvořená halda

(6)	1	R
(5)	2	J
(4)	3	U
(3)	4	Z
(2)	5	B
(1)	6	D
	7	E
	8	M
	9	O
	10	K
	11	A
	12	T

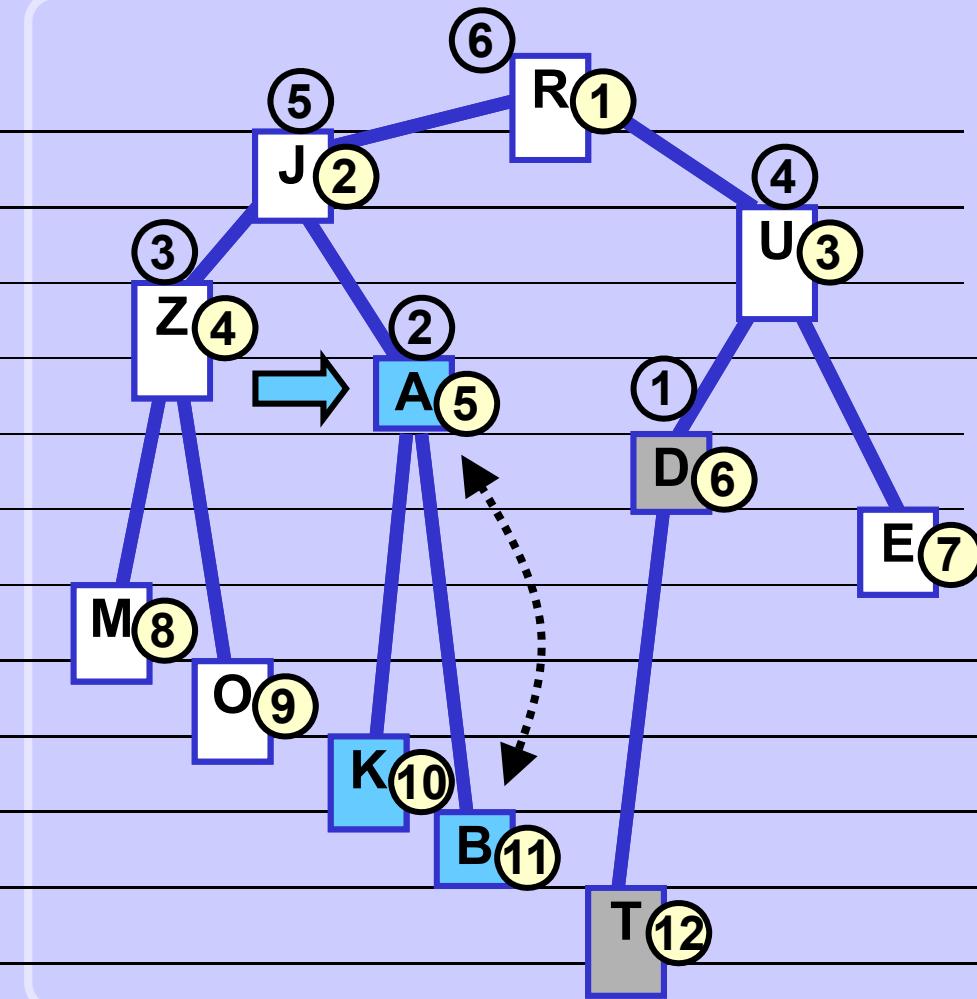


## Tvorba haldy v poli

### Pole

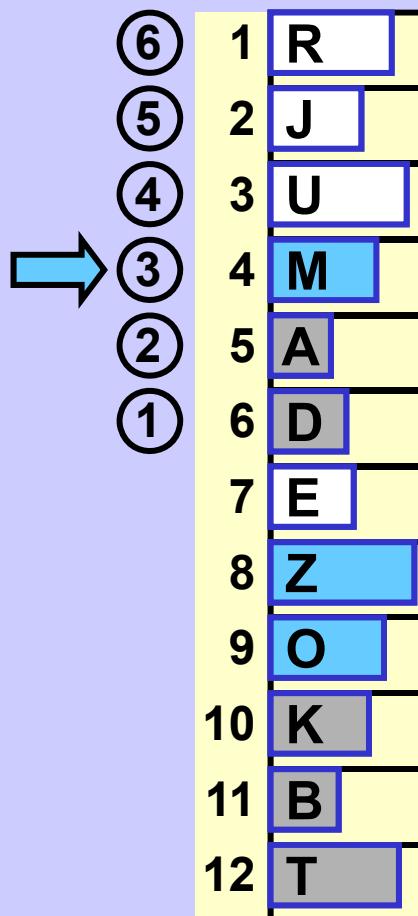


Dříve vytvořená halda .....→ Přesuny  
Aktuálně vytvořená halda

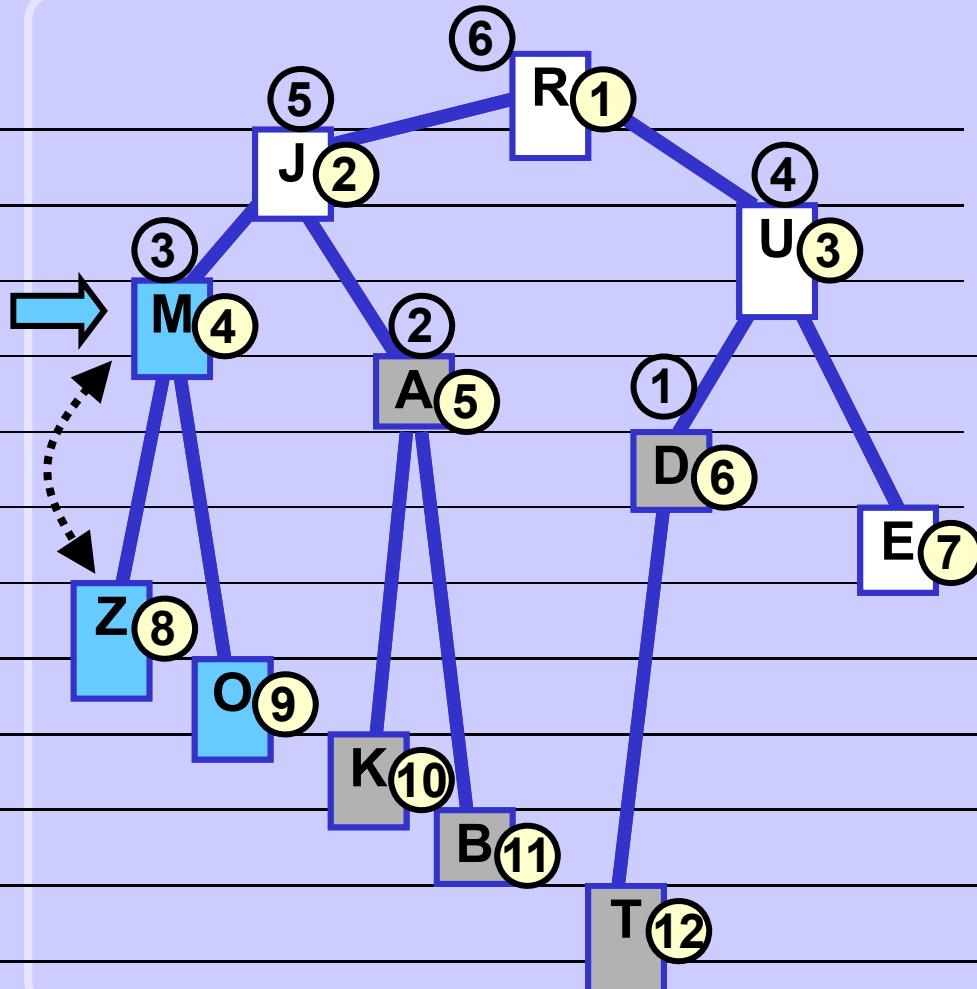


## Tvorba haldy v poli

### Pole

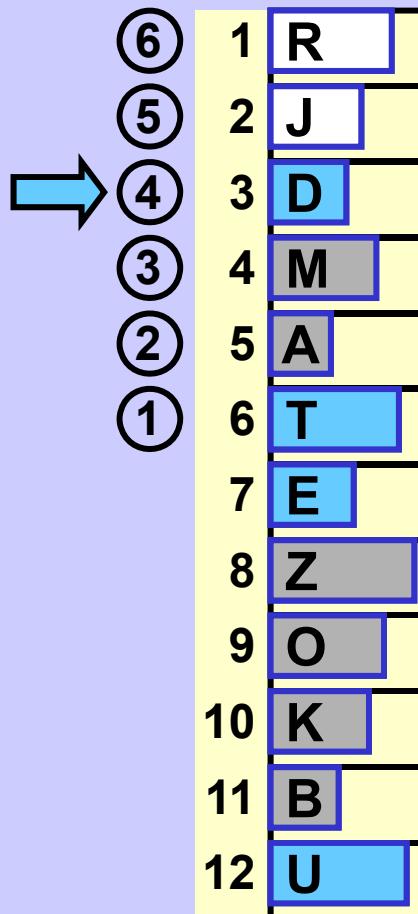


■ Dříve vytvořená hulta  
➡ Aktuálně vytvořená hulta

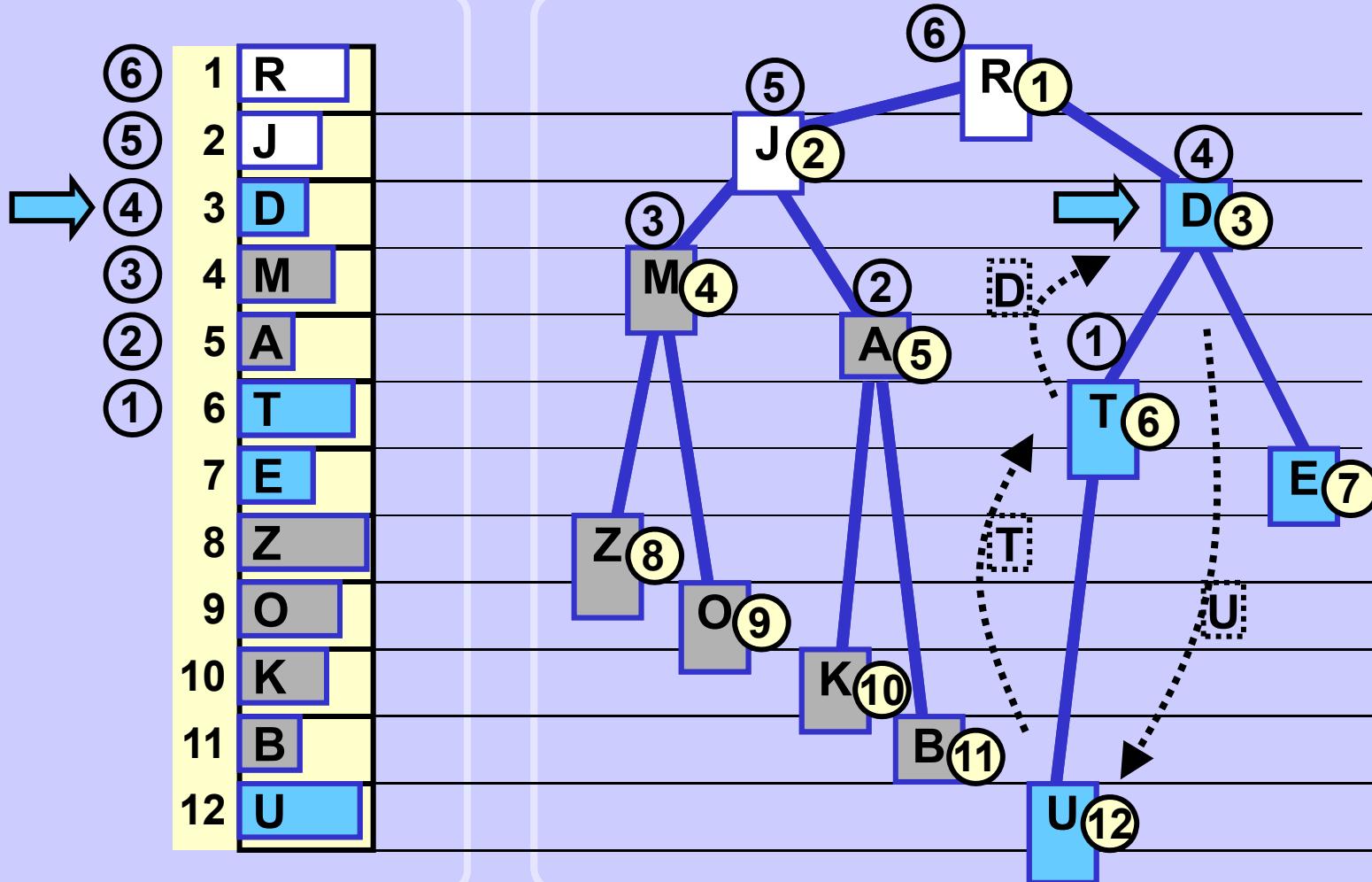


## Tvorba haldy v poli

### Pole

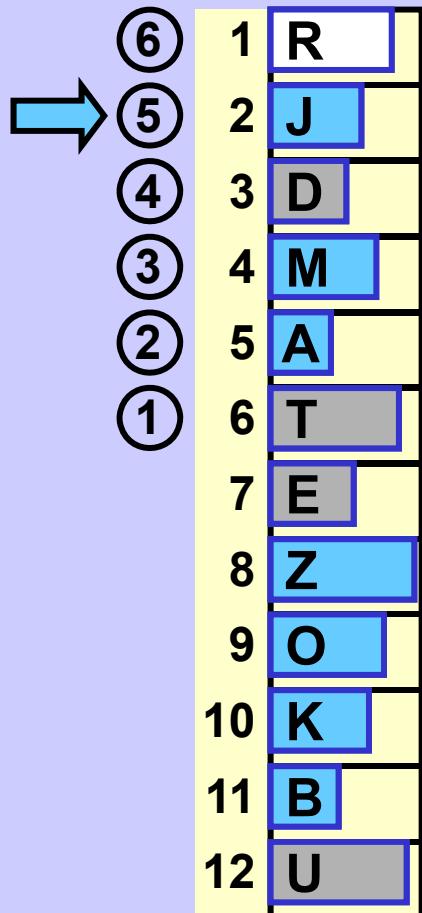


■ Dříve vytvořená halda    ..... Přesuny  
■ Aktuálně vytvořená halda

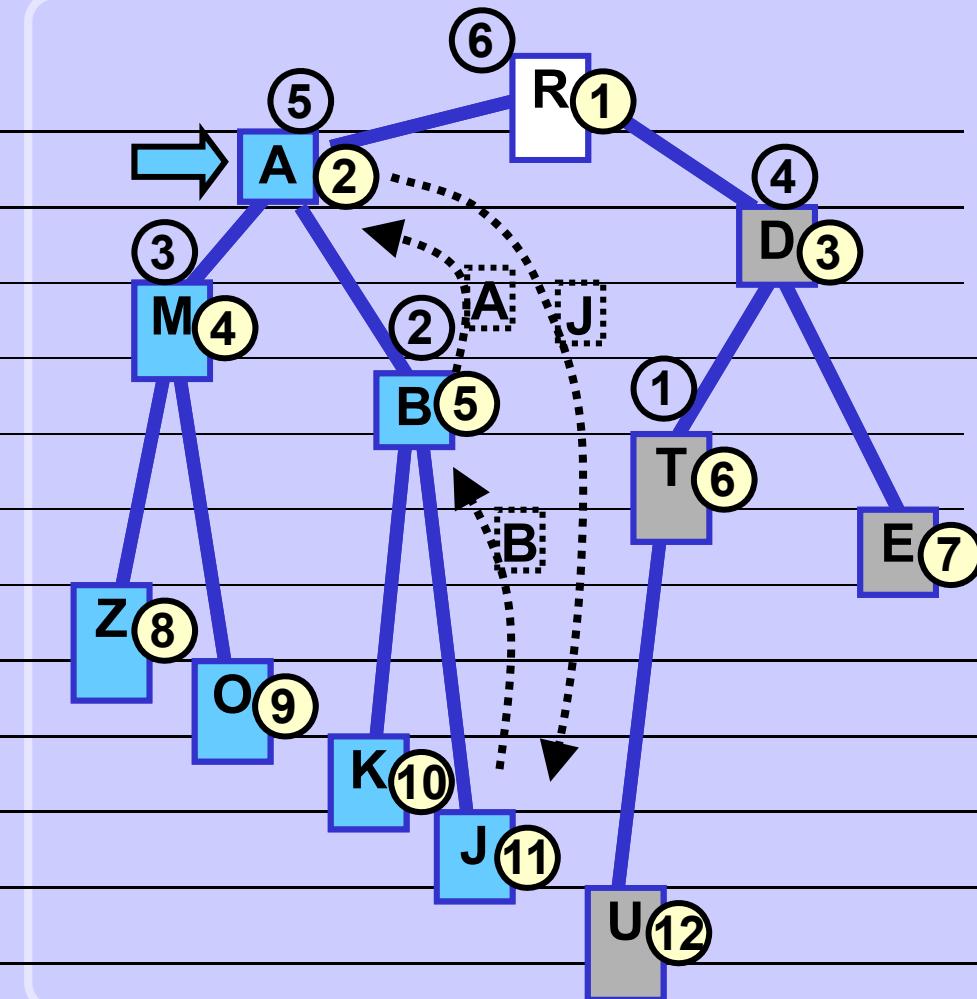


## Tvorba haldy v poli

### Pole

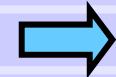


■ Dříve vytvořená hala  
→ ■ Aktuálně vytvořená hala  
..... Přesuny



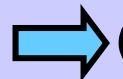
## Tvorba haldy v poli

Pole



Aktuálně vytvořená halda

.....▼ Přesuny



6

5

4

3

2

1

R

J

D

M

A

T

E

Z

O

K

B

U

6

5

4

3

2

1

A

D

M

J

Z

O

K

R

U

1

2

3

4

5

6

7

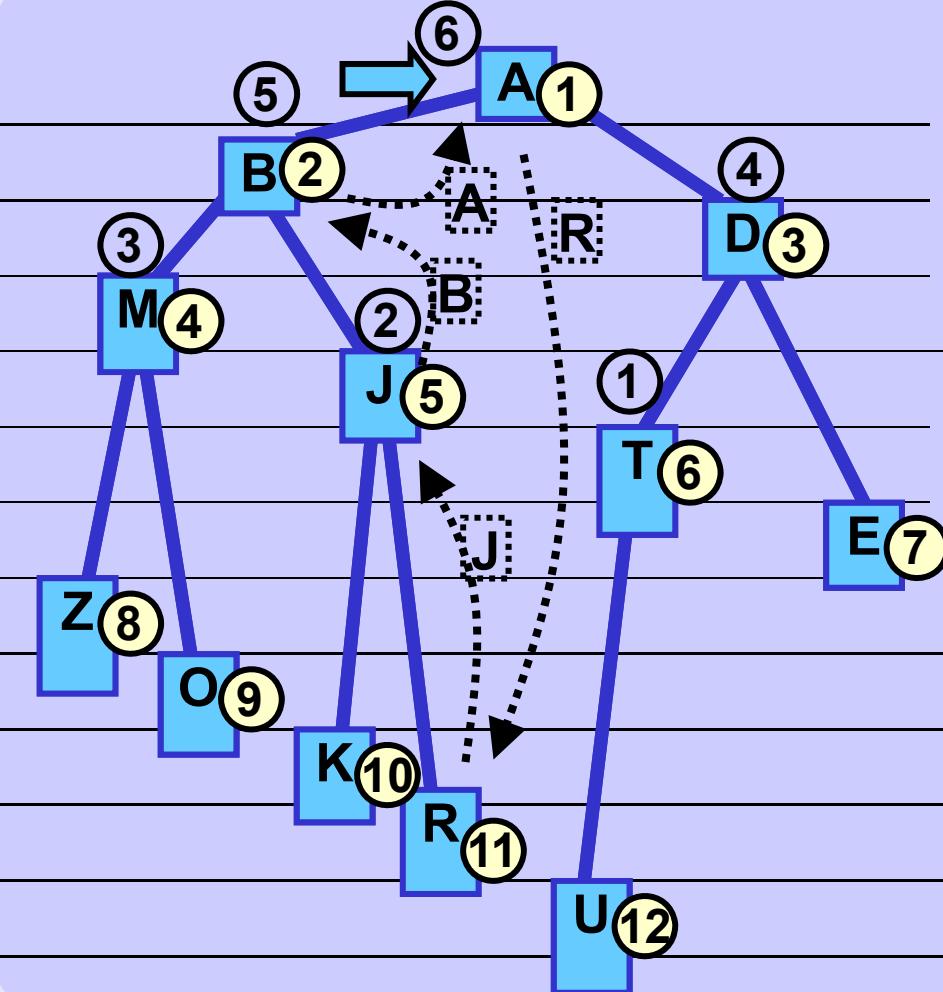
8

9

10

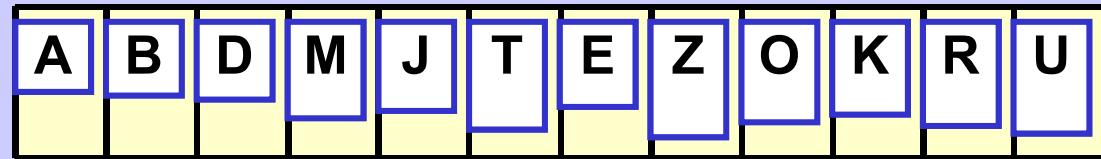
11

12

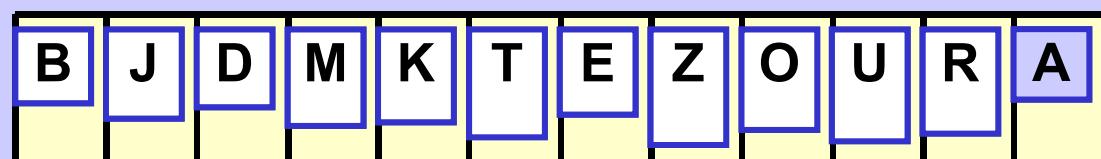
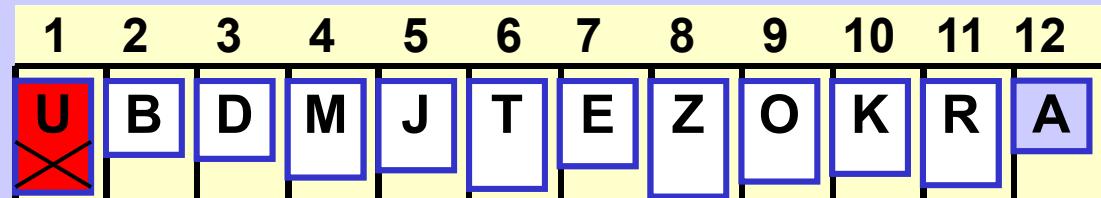
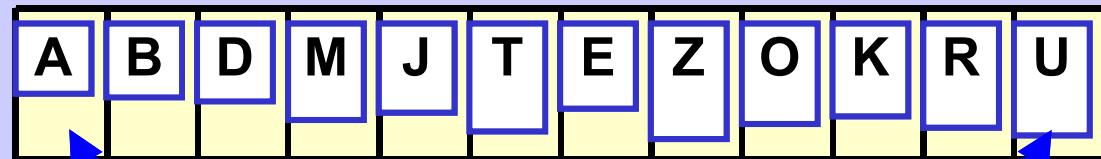


## Heap sort

Halda

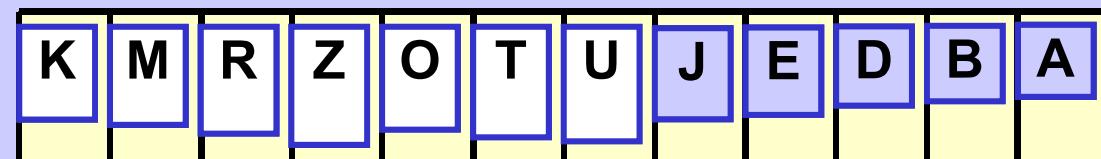
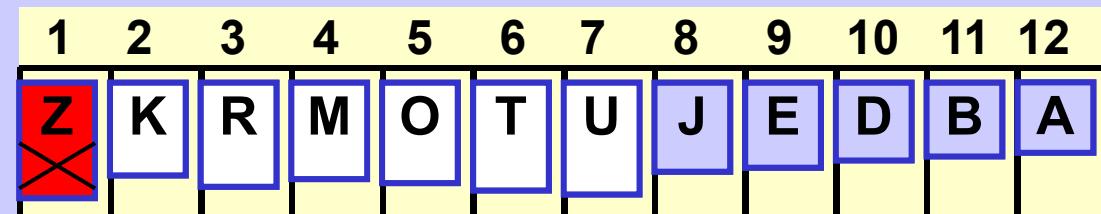
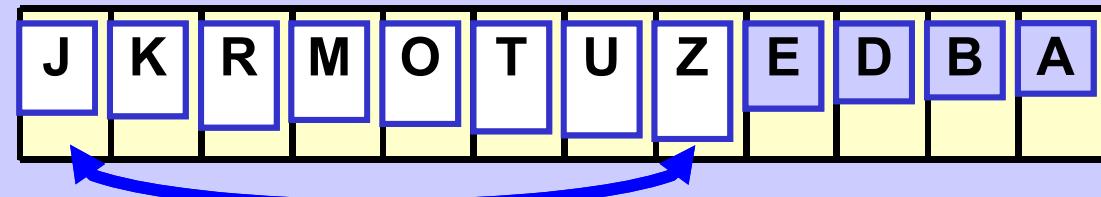


Krok 1



## Heap sort

Krok k



Halda

k

## Heap sort

```
// array: a[1]...a[n] !!!!  
  
void heapSort( Item [ ] a, int n ) {  
  
    // create a heap  
    for( int i = n/2; i > 0; i-- )  
        repairTop( a, i, n );  
  
    // sort  
    for( int i = n; i > 1; i-- ) {  
        swap( a, 1, i );  
        repairTop( a, 1, i-1 );  
    }  
}
```

## Heap sort

```

                // array: a[1]...a[n] !!!!!!
void repairTop( Item [ ] a, int top, int bottom ) {
    int i = top;           // a[2*i] and a[2*i+1]
    int j = i*2;          // are children of a[i]

    Item topVal = a[top];

                // try to find a child < topVal
    if( (j < bottom) && (a[j] > a[j+1]) ) j++;

                // while (children < topVal)
                //       move children up
    while( (j <= bottom) && (topVal > a[j]) ){
        a[i] = a[j];
        i = j; j = j*2; // skip to next child
        if( (j < bottom) && (a[j] > a[j+1]) ) j++;
    }
    a[i] = topVal;      // put topVal where it belongs
}

```

## Heap sort

**repairTop** operace nejhorší případ ...  $\log_2(n)$  ( $n$ =velikost haldy)

vytvoř haldu ...  $n/2$  **repairTop** operací

$$\log_2(n/2) + \log_2(n/2+1) + \dots + \log_2(n) \leq (n/2)(\log_2(n)) = \underline{\underline{O(n \cdot \log(n))}}$$

lze ukázat ...  $n/2$  **repairTop** operací ...  $\underline{\underline{\Theta(n)}}$

seřad' haldu ...  $n-1$  **repairTop** operací, nejhorší případ:

$$\log_2(n) + \log_2(n-1) + \dots + 1 \leq n \cdot \log_2(n) = \underline{\underline{O(n \cdot \log(n))}}$$

celkem ... vytvoř haldu + seřad' haldu =  $\underline{\underline{O(n \cdot \log(n))}}$

Asymptotická složitost Heap sortu je  $\underline{\underline{O(n \cdot \log(n))}}$

V praktických situacích se očekává  $\underline{\underline{\Theta(n \cdot \log(n))}}$

Heap sort není stabilní

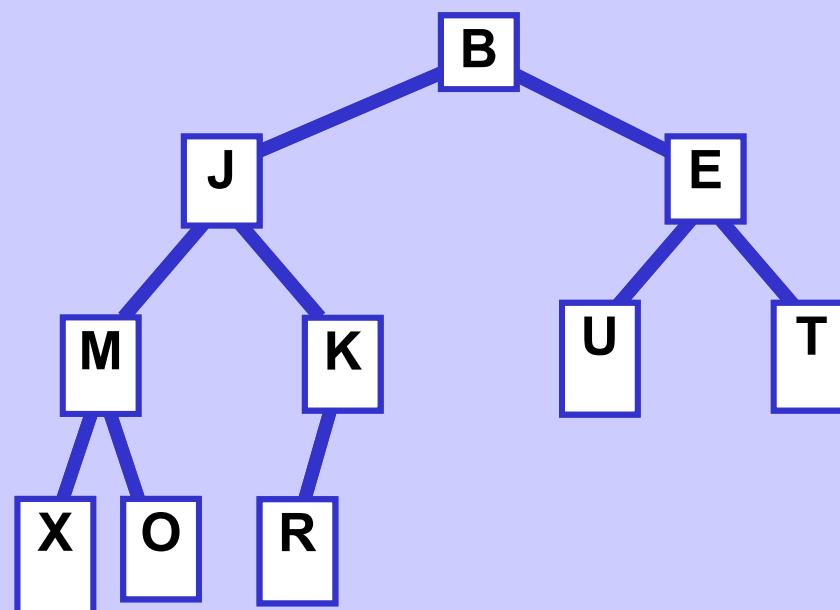
Otzáka: Pro jaká data nastane složitost menší než  $\underline{\underline{\Theta(n \cdot \log(n))}}$  ?

## Prioritní fronta

Prioritní fronta má stejné operace jako obyčejná fronta

- vlož do fronty (Insert, Enqueue, apod),
- čti první prvek (Front, Top, apod),
- smaž první prvek (Dequeue, Pop, apod).

Navíc interně zajišťuje, že na čele fronty je vždy prvek s minimální (maximální) hodnotou ze všech prvků ve frontě.



Prioritní frontu lze implementovat pomocí haldy.

Plným jménem:  
"Binární haldy".

## Prioritní fronta pomocí binární haldy -- operace

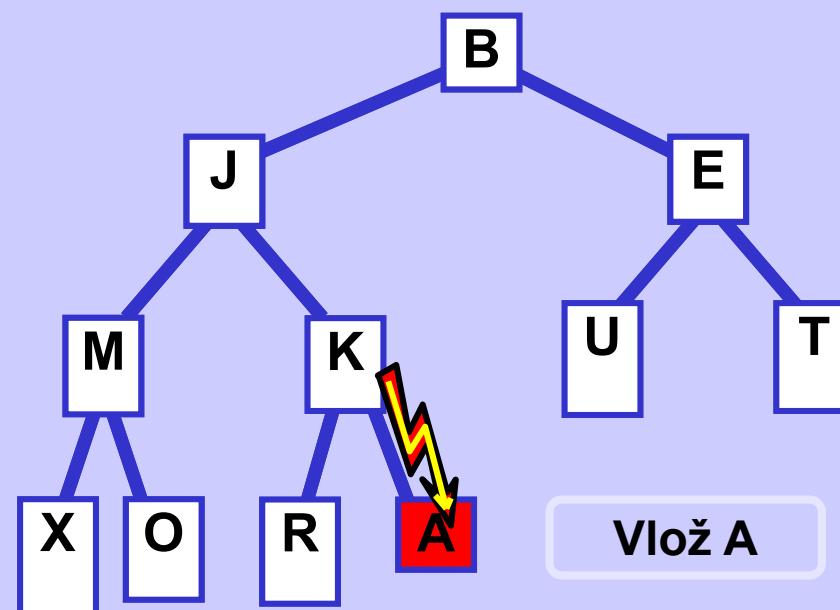
Čti první prvek (Front, Top, apod) .

Zřejmě.

Smaž první prvek (Dequeue, Pop, apod) = Odstraň vrchol a oprav haldu.

Viz výše.

Vlož do fronty (Insert, Enqueue, apod).

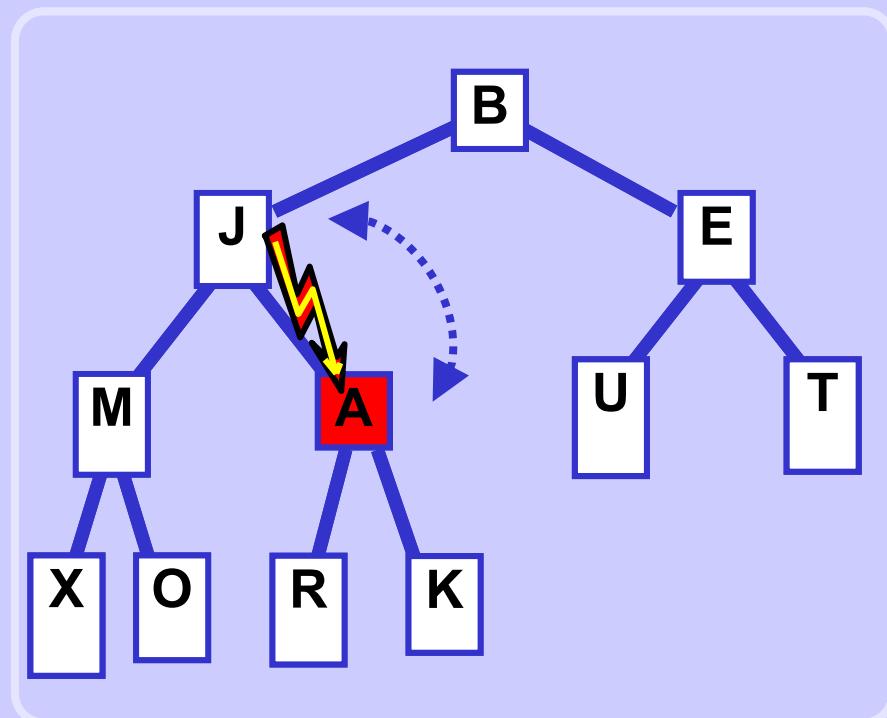
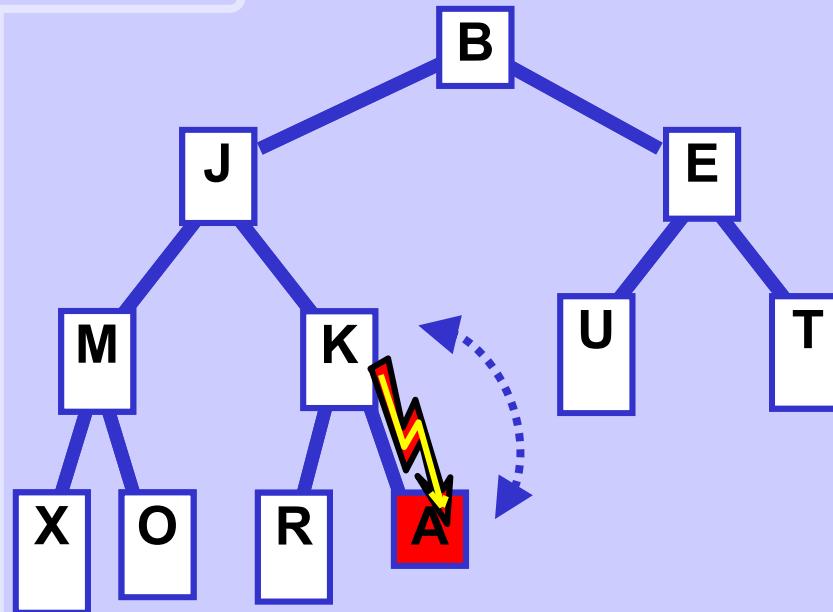


Vložíme prvek  
na konec fronty (haldy).

Ve většině případů  
se tím poruší pravidlo haldy  
a je nutno haldu opravit.

## Prioritní fronta pomocí binární haldy – vlož prvek

Vlož A

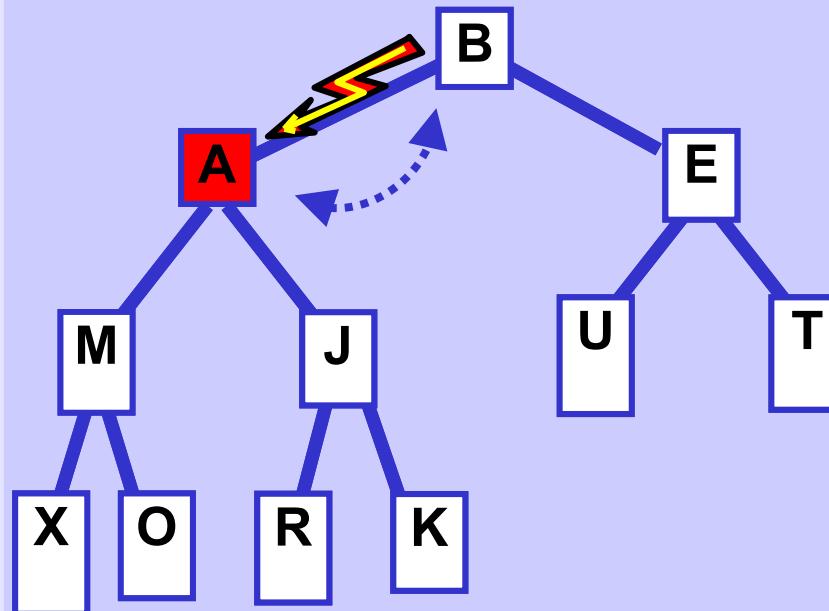


Pravidlo haldy je porušeno,  
vyměň vkládaný prvek  
s jeho rodičem.

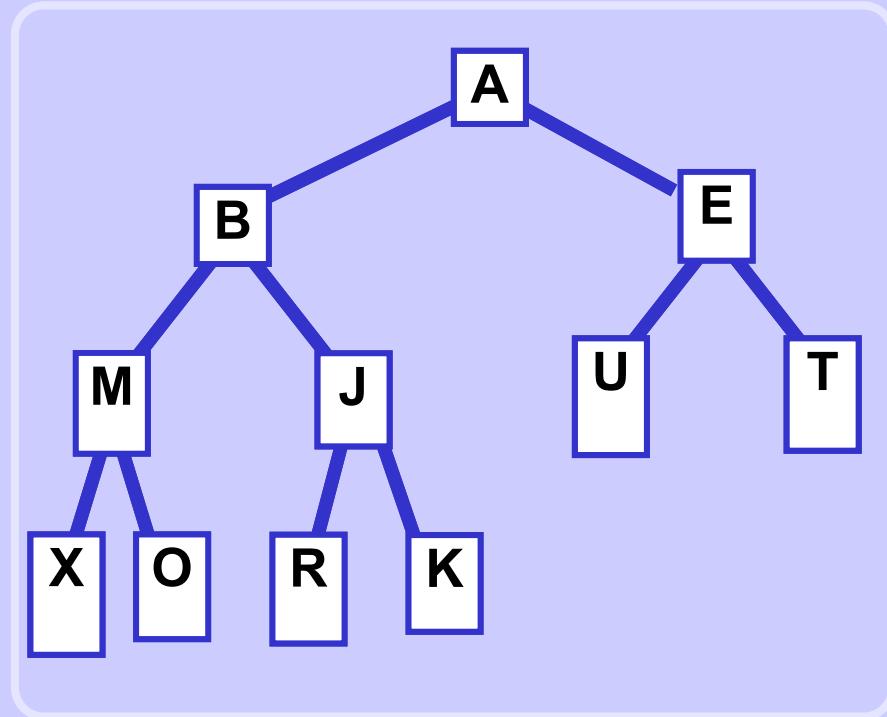
Pravidlo haldy je stále porušeno,  
vyměň vkládaný prvek  
s jeho rodičem.

## Prioritní fronta pomocí binární haldy – vlož prvek

Vkládáme A



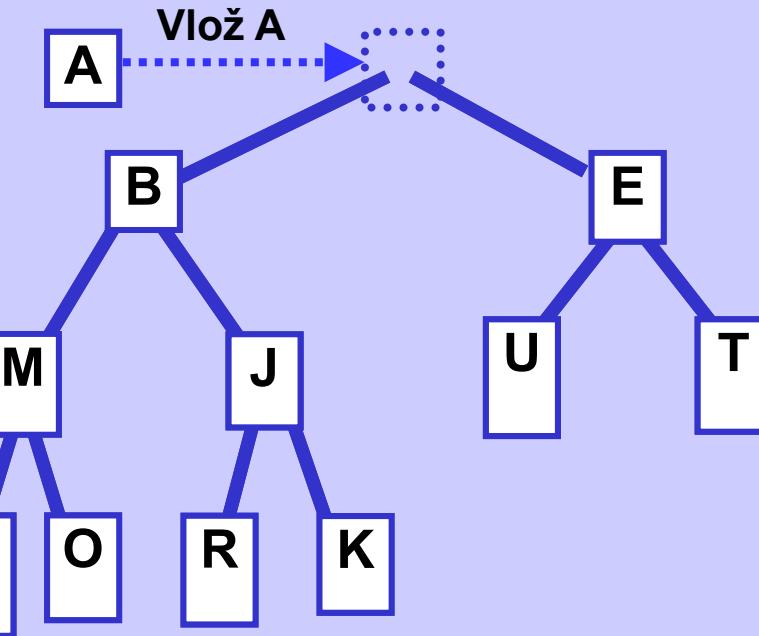
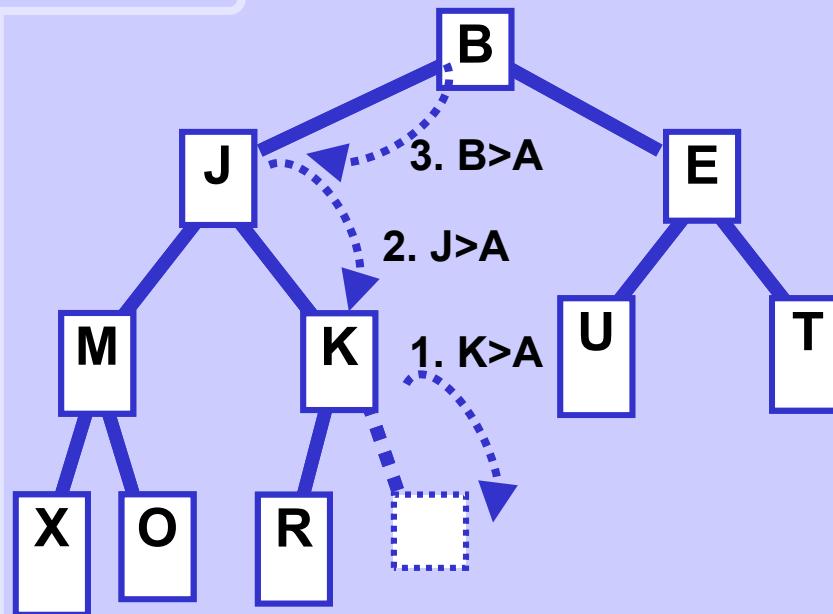
Pravidlo haldy je stále porušeno,  
vyměň vkládaný prvek  
s jeho rodičem.



Pravidlo haldy je zachováno,  
vkládaný prvek našel své místo  
v haldě.

## Binární halda – vlož prvek, efektivněji

Vlož A



Vkládaný prvek na konec haldy  
nevkládej.

Napřed zjisti, kam patří, ostatní  
(větší) prvky posuň o patro dolů ...

... a teprve nakonec  
vlož prvek na jeho místo.

## Binární halda – vlož prvek

```

                // array: a[1]...a[n] !!!!!!
int insert( Item [] a, int x, int bottom ){
    int j = ++bottom;           // expand the heap
    int i = j/2;                // parent index

    while( (i > 0) && (a[i] > x) ){
        a[j] = a[i];            // move elem down the heap
        j = i; i /= 2;           // move indices up the heap
    }
    a[i] = x;                  // put inserted elem to its place
    return bottom;
}

```

## Binární halda – Složitost operace insert

Vkládání představuje průchod vyváženým stromem s n prvky od listu nejvýše ke kořeni, složitost operace Insert je tedy  $O(\log_2(n))$ .

# ALG 09

**Radix sort (příhrádkové řazení)**

**Counting sort**

**Přehled asymptotických rychlostí jednotlivých řazení**

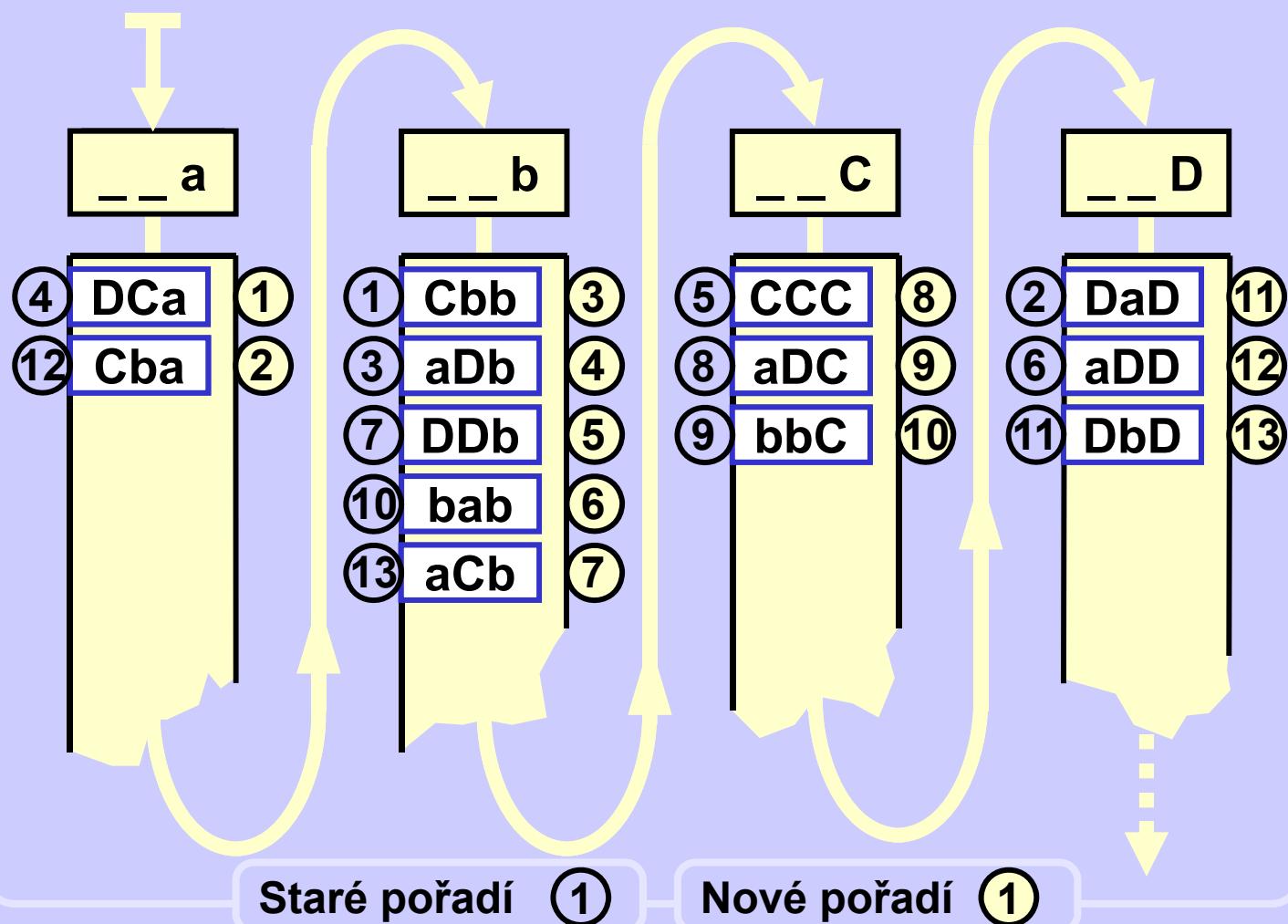
**Ilustrační experiment řazení**

## Radix sort

Neseřazeno

1	Cbb
2	DaD
3	aDb
4	DCa
5	CCC
6	aDD
7	DDb
8	aDC
9	bbC
10	bab
11	DbD
12	Cba
13	aCb

Řad' podle 3. znaku

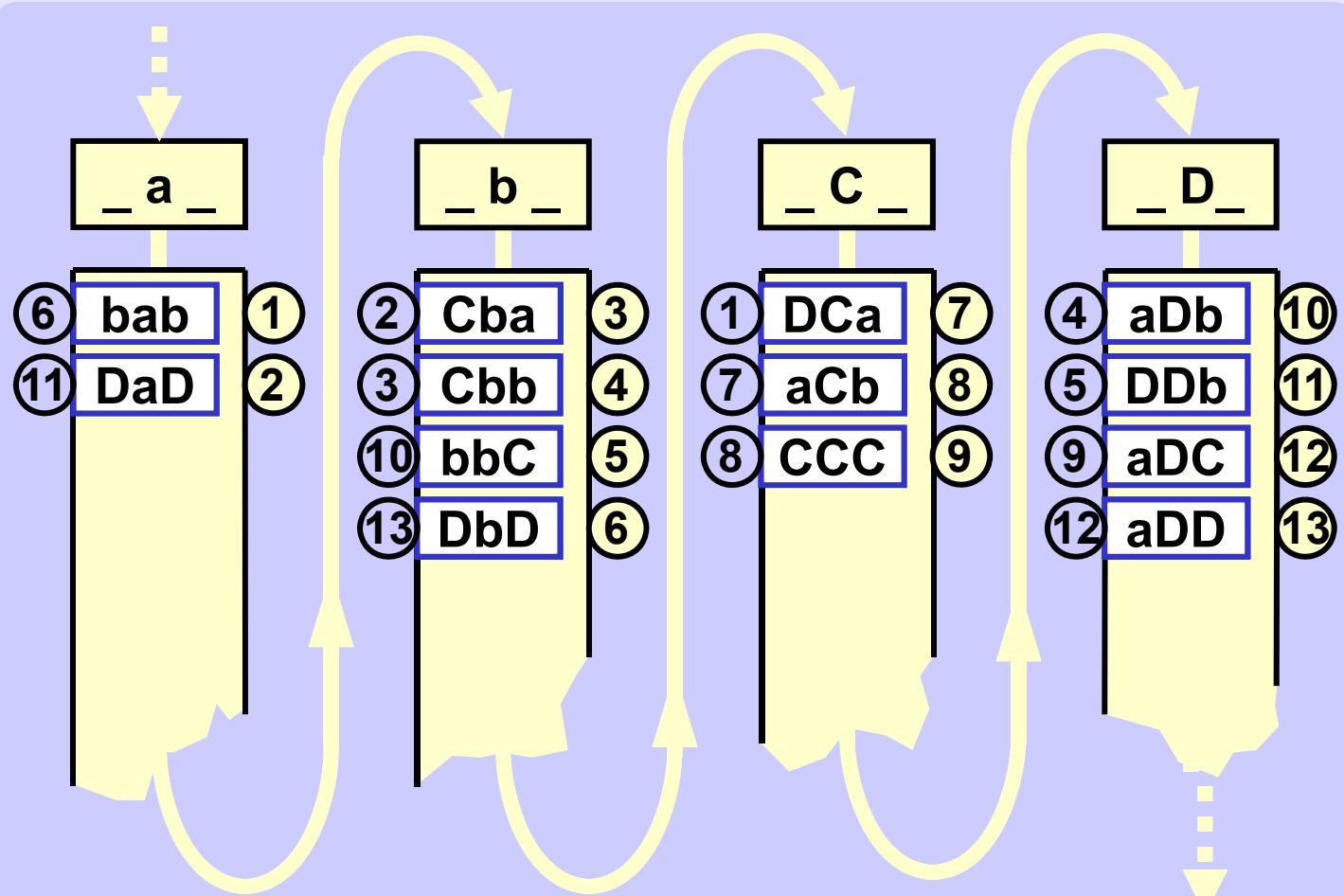


## Radix sort

Seřazeno  
od 3. znaku

1	DCa
2	Cba
3	Cbb
4	aDb
5	DDb
6	bab
7	aCb
8	CCC
9	aDC
10	bbC
11	DaD
12	aDD
13	DbD

Řad' podle 2. znaku

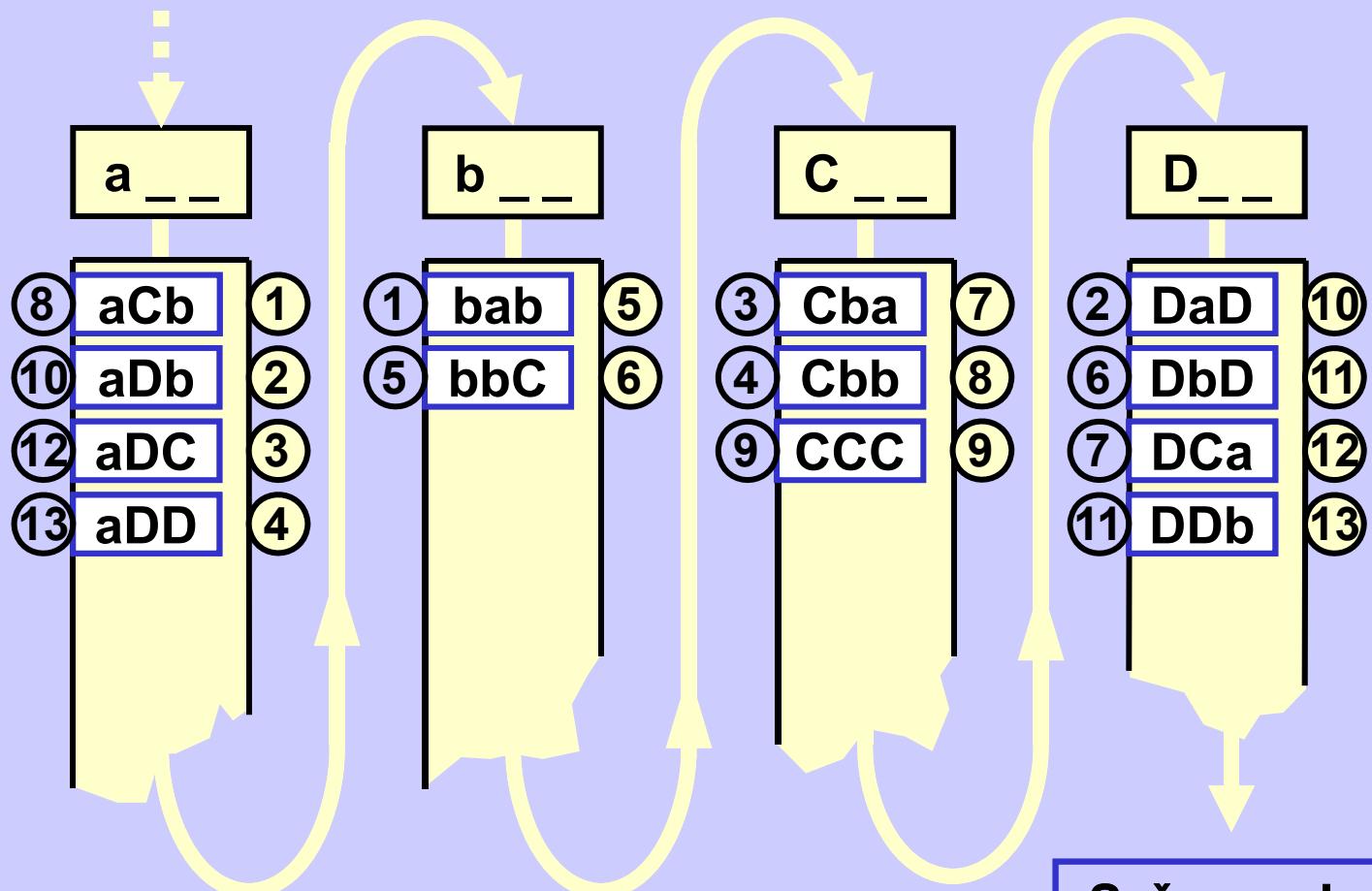


## Radix sort

Seřazeno  
od 2. znaku

1	bab
2	DaD
3	Cba
4	Cbb
5	bbC
6	DbD
7	DCa
8	aCb
9	CCC
10	aDb
11	DDb
12	aDC
13	aDD

Řad' podle 1. znaku

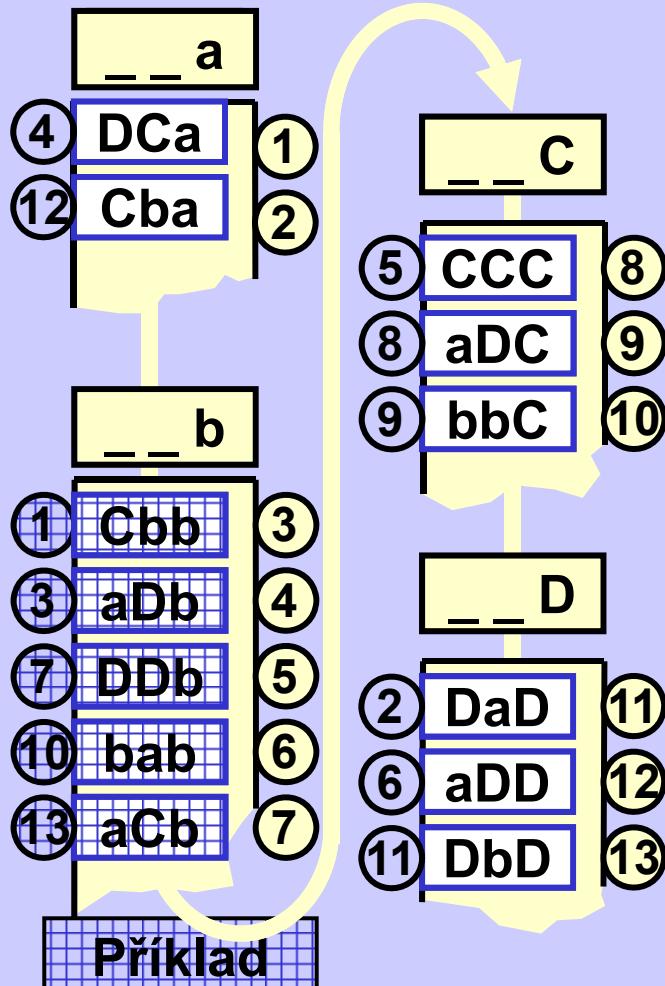


## Implementace radix sortu

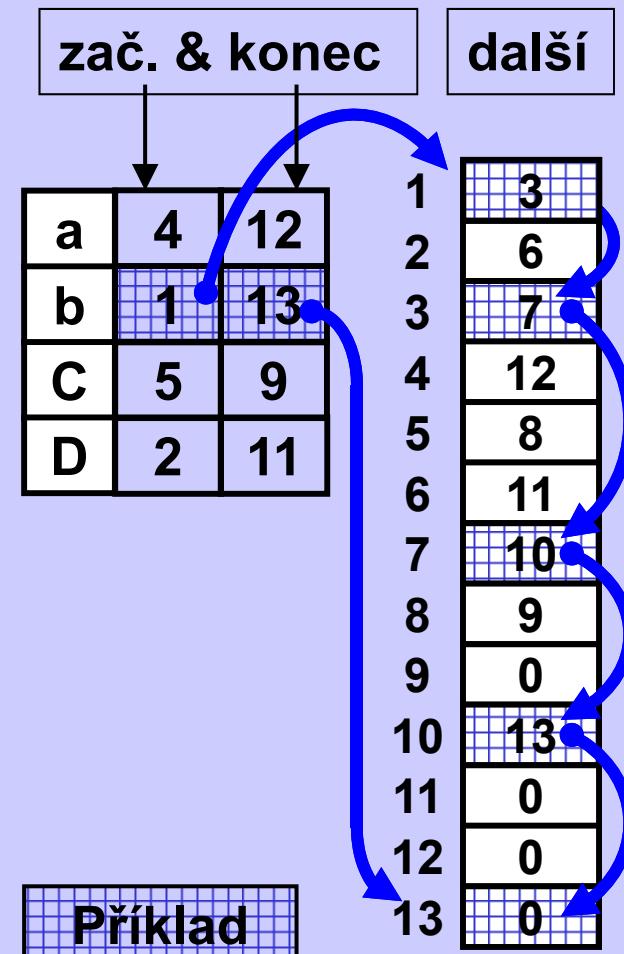
Neseřazeno

1	Cbb
2	DaD
3	aDb
4	DCa
5	CCC
6	aDD
7	DDb
8	aDC
9	bbC
10	bab
11	DbD
12	Cba
13	aCb

Seřazeno dle 3. znaku



Pomocná pole indexů registrují nové pořadí.



## Implementace radix sortu

Neseřazeno

1	Cbb
2	DaD
3	aDb
4	DCa
5	CCC
6	aDD
7	DDb
8	aDC
9	bbC
10	bab
11	DbD
12	Cba
13	aCb

Jedno pole pro všechny seznamy



Pole ukazatelů  
na začátek a  
konec seznamu  
pro každý znak

Aktuálně  
obě pole přesně  
registrují stav  
po seřazení  
podle 3. znaku.



Ukázka  
seznamu  
pro 'b'

Radix sort lze provést  
bez přesouvání původních dat,  
pouze manipulací s uvedenými  
celočíselnými poli, která  
obsahují veškerou informaci  
o aktuálním stavu řazení.

## Implementace radix sortu

Neseřazeno

1	Cbb
2	DaD
3	aDb
4	DCa
5	CCC
6	aDD
7	DDb
8	aDC
9	bbC
10	bab
11	DbD
12	Cba
13	aCb

Stav po seřazení podle 2. znaku.

9	z	k
0		
a	10	2
b	12	11
C	4	5
D	3	6

Stav po seřazení podle 1. znaku = seřazeno.

5	z	k
11		
8		
7		
0		
0		
0		
6		
0		
9		
4		
1		
3		

Ukázka seznamů pro 'b'

## Implementace radix sortu

Neseřazeno

1	Cbb
2	DaD
3	aDb
4	DCa
5	CCC
6	aDD
7	DDb
8	aDC
9	bbC
10	bab
11	DbD
12	Cba
13	aCb

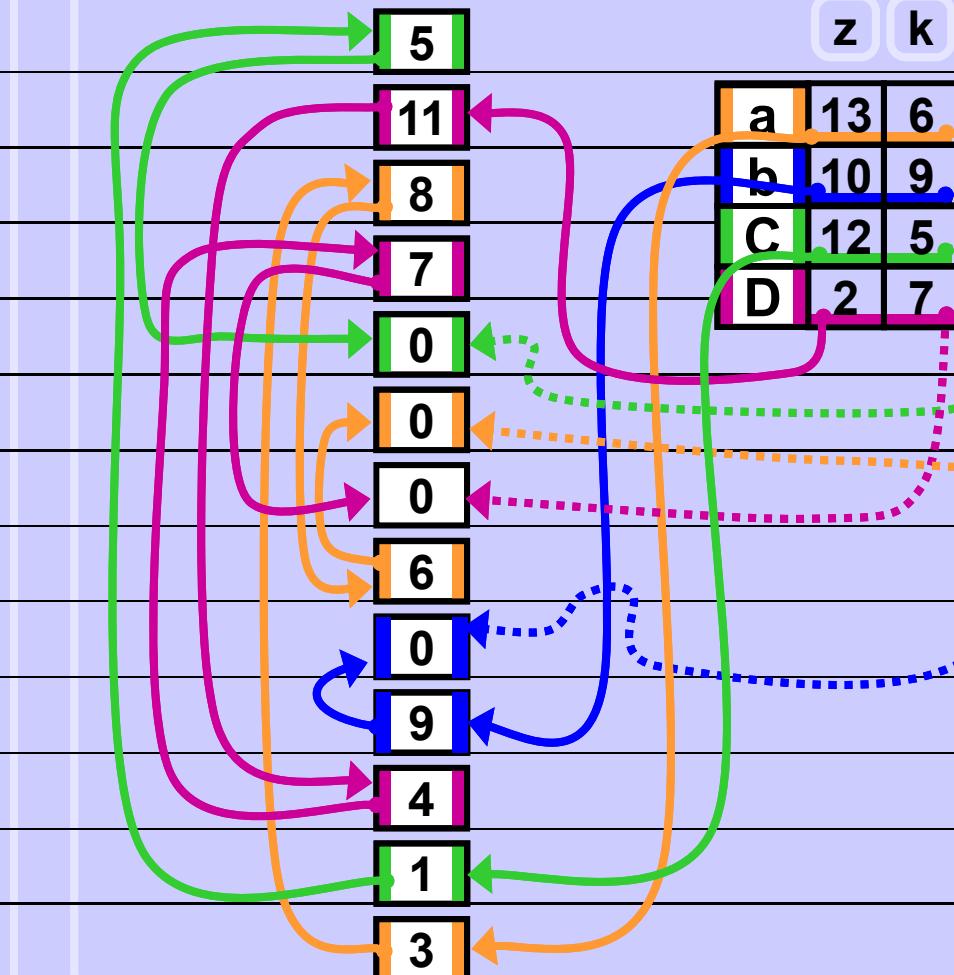
Stav po seřazení podle  
1. znaku = seřazeno.

Stačí vypsat data  
v pořadí daném seznamy:  
a → b → C → D →

z k

a	13	6
b	10	9
C	12	5
D	2	7

13	aCb
3	aDb
8	aDC
6	aDD
10	bab
9	bbC
12	Cba
1	Cbb
5	CCC
2	DaD
11	DbD
4	DCa
7	DDb



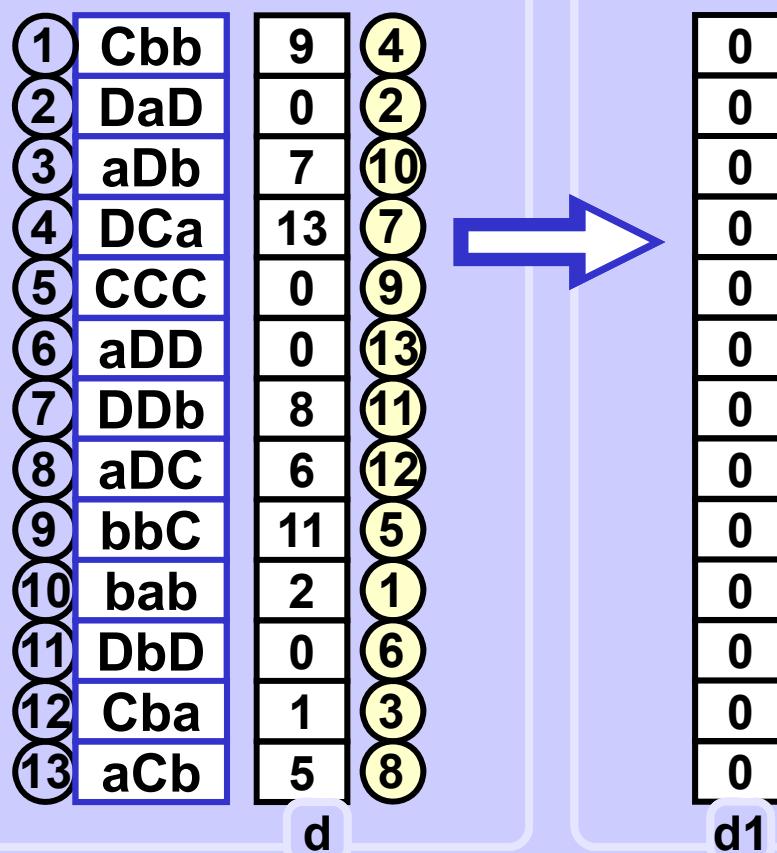
## Od seřazení podle 2. znaku k seřazení podle 1. znaku

Pole obsahují uspořádání podle 2. znaku.

	<b>z</b>	<b>k</b>
a	10	2
b	12	11
C	4	5
D	3	6

	<b>z1</b>	<b>k1</b>
a	0	0
b	0	0
C	0	0
D	0	0

Pole budou obsahovat uspořádání podle 1. znaku



Aktualizace polí z, k, d proběhne tak, že naplníme nová pole z1, k1, d1, která nakonec zkopírujeme zpět do z, k, d.

Implementačně ovšem není třeba cokoli kopírovat, stačí záměna referencí (ukazatelů, pointerů) na tato pole.

## Od seřazení podle 2. znaku k seřazení podle 1. znaku

Usp. dle 2. zn.

	z	k	a	10	2	b	12	11	C	4	5	D	3	6
1	Cbb	9	4	2										
2	DaD	0	2											
3	aDb	7	10											
4	DCa	13	7											
5	CCC	0	9											
6	aDD	0	13											
7	DDb	8	11											
8	aDC	6	12											
9	bbC	11	5											
10	bab	2	1											
11	DbD	0	6											
12	Cba	1	3											
13	aCb	5	8											
		d												

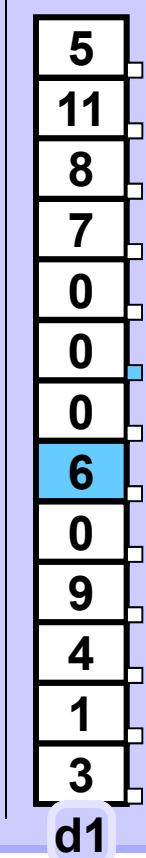
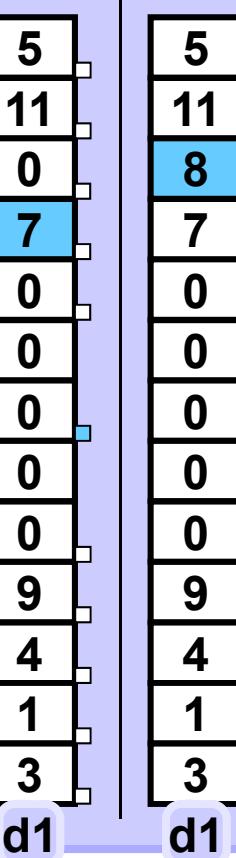
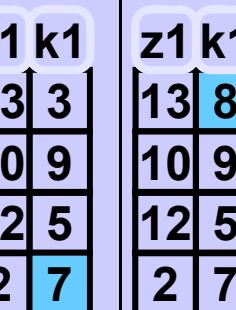
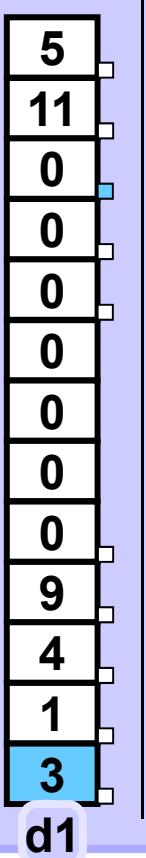
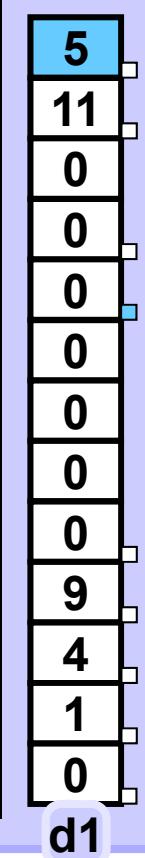
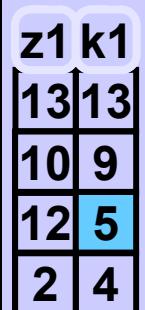
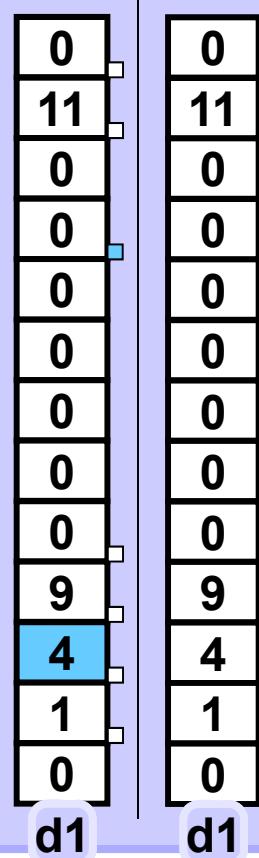
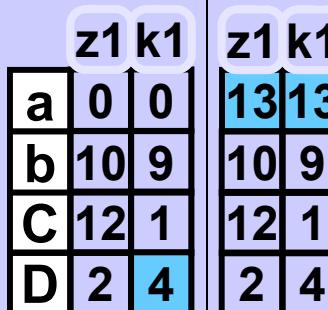
  

	z1	k1	a	0	0	b	0	0	C	0	0	D	0	0
1	Cbb	9	4	2										
2	DaD	0	2											
3	aDb	7	10											
4	DCa	13	7											
5	CCC	0	9											
6	aDD	0	13											
7	DDb	8	11											
8	aDC	6	12											
9	bbC	11	5											
10	bab	2	1											
11	DbD	0	6											
12	Cba	1	3											
13	aCb	5	8											
	d1													

**Od seřazení podle 2. znaku k seřazení podle 1. znaku**

Usp.  
dle  
2. zn.

		z	k
a	10	2	
b	12	11	
C	4	5	
D	3	6	



Hotovo

## Implementace radix sortu

```

void radix_sort( String [] a ) {
    int charCount = ...; // number of chars used (2^16? )
    int [] z = new int [charCount];
    int [] k = new int [charCount];
    int [] z1 = new int [charCount];
    int [] k1 = new int [charCount];
    int [] d = new int [a.length];
    int [] d1 = new int [a.length];
    int [] aux;

    initStep( a, z, k, d ); // 1st pass with last char

    for( int p = a[0].length()-2; p >= 0; p-- ) {
        radixStep( a, p, z, k, d, z1, k1, d1 ); // do the job
        aux = z; z = z1; z1 = aux; // just swap arrays
        aux = k; k = k1; k1 = aux; // dtto
        aux = d; d = d1; d1 = aux; // dtto
    }
    output( a, z, k, d ); // print sorted array
}

```

## Implementace radix sortu

```

void initStep( String[] a, int [] z, int [] k, int [] d ){
    int pos = a[0].length()-1;           // last char in string
    int c;                          // char as array index for Radix sort
    for( int i = 0; i < z.length; i++ )   // init arrays
        z[i] = k[i] = -1;             // empty
    for( int i = 0; i < a.length; i++ ){ // all last chars
        c = (int) a[i].charAt(pos);      // char to index
        if( z[c] == -1 )
            k[c]= z[c] = i;           // start new list
        else {
            d[k[c]] = i;             // extend existing list
            k[c] = i;
    } } }

```

Řetězce různé délky nutno uvést na stejnou délku  
připojením "nevýznamných znaků", např. mezer.

V ukázkovém kódu všechny indexy polí začínají 0,  
v obrázcích začínají 1.

## Implementace radix sortu

```

void radixStep( String [] a, int pos, int [] z, int [] k,
                int [] d, int [] z1, int [] k1, int [] d1 ){
    int j;                      // index traverses old lists
    int c;                      // char as array index for Radix sort
    for( int i = 0; i < z.length; i++ ) // init arrays
        z1[i] = k1[i] = -1;           //
    for( int i = 0; i < z.length; i++ ) // for all used chars
        if (z[i] != -1) {           // unempty list
            j = z[i];
            while( true ){          // scan the list
                c = ( int ) a[j].charAt(pos); // char to index
                if( z1[c] == -1 )
                    k1[c]= z1[c] = j;      // start new list
                else {
                    d1[k1[c]] = j;       // extend existing list
                    k1[c] = j;
                }
                if( j == k[i] ) break;
                j = d[j];               // next string index
            } } }
}

```

## Radix sort

### Shrnutí

d znaků ..... d cyklů

cyklus .....  $\Theta(n)$  operací

---

celkem .....  $\Theta(d \cdot n)$  operací

---

$d \ll n \Rightarrow \dots \Theta(n)$  operací

Radix sort nemění pořadí stejných hodnot

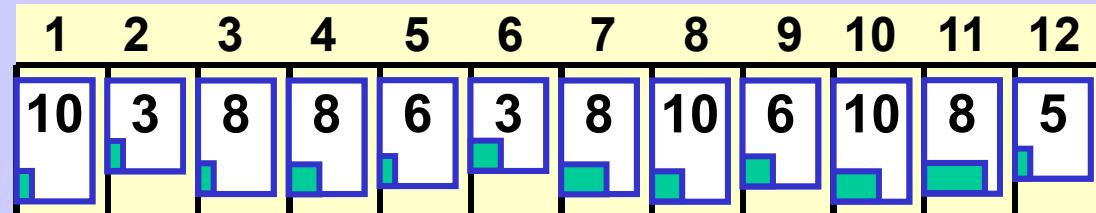
Asymptotická složitost Radix sortu je  $\Theta(n^*d)$

Pro malé konstantní d lze psát  $\Theta(n)$

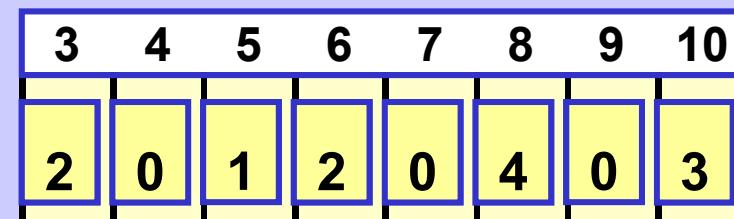
Je to stabilní řazení

## Counting sort

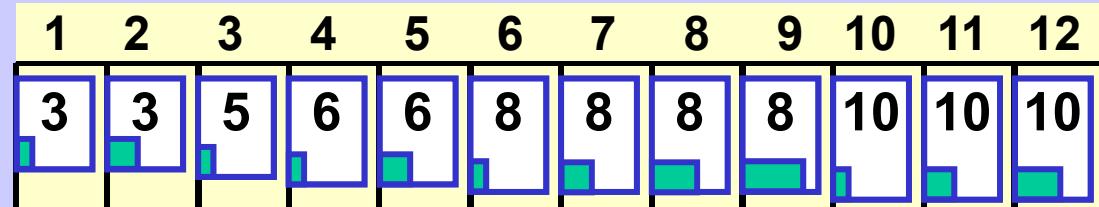
**vstup**  
**vstup.length == N**



**cetnost**  
**cetnost.length == k**  
 $k = \max(vstup) - \min(vstup) + 1$



**vystup**  
**vstup.length == N**



## Counting sort

Krok 1

Vynulování pole četností

3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0

Jeden průchod vstupním polem

10	3	8	8	6	3	8	10	6	10	8	5
----	---	---	---	---	---	---	----	---	----	---	---

Naplnění pole četností

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

## Counting sort

Krok 2

jeden průchod

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

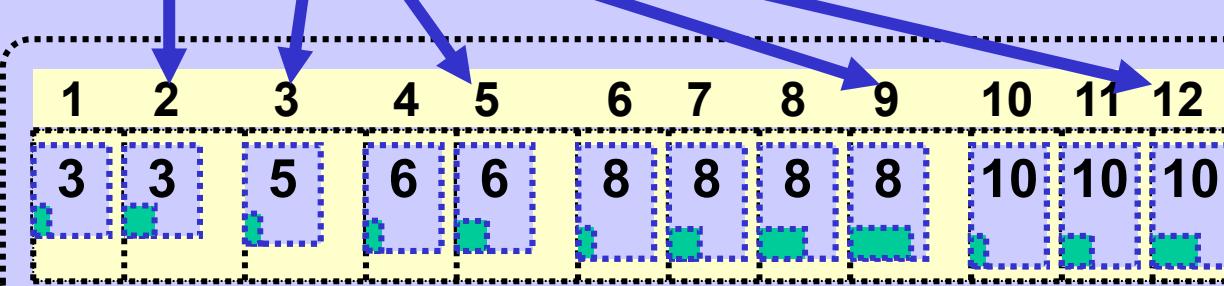
Pole četností  
mění svou roli

Úprava pole četností

```
for( int i = dMez+1; i <= hMez; i++ )
    ctnost[i] += ctnost[i-1]
```

3	4	5	6	7	8	9	10
2	2	3	5	5	9	9	12

Prvek **ctnost[ j ]** obsahuje pozici posledního prvku s hodnotou **j** v budoucím výstupním poli.

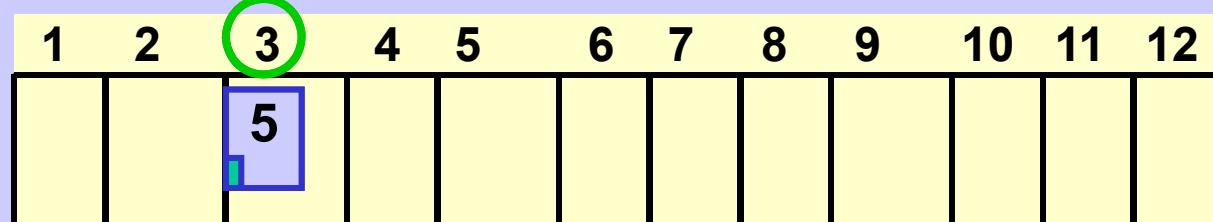
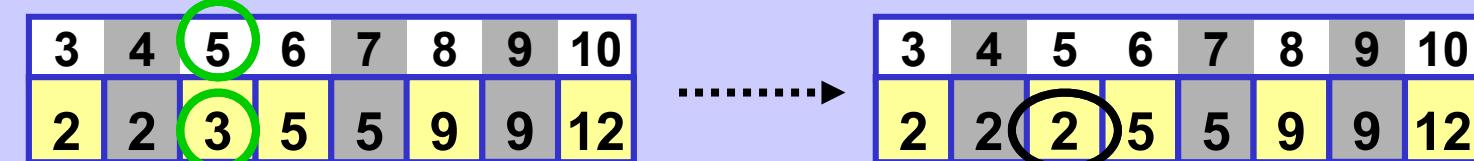
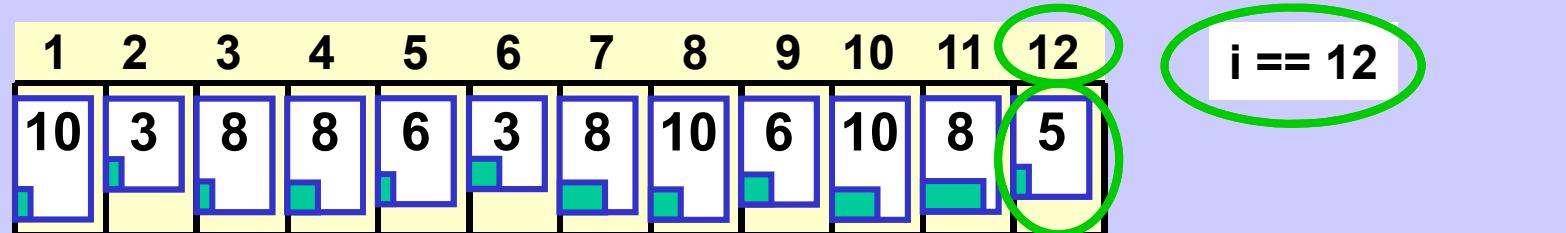


## Counting sort

Krok 3

$i == N$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnosc[vstup[i]]] = vstup[i];
    cetnosc[vstup[i]]--;
}
```

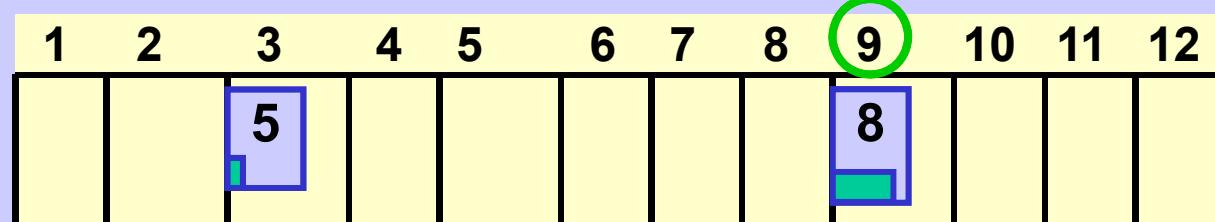
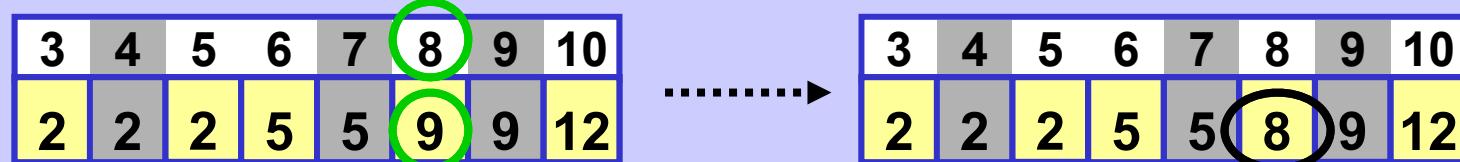
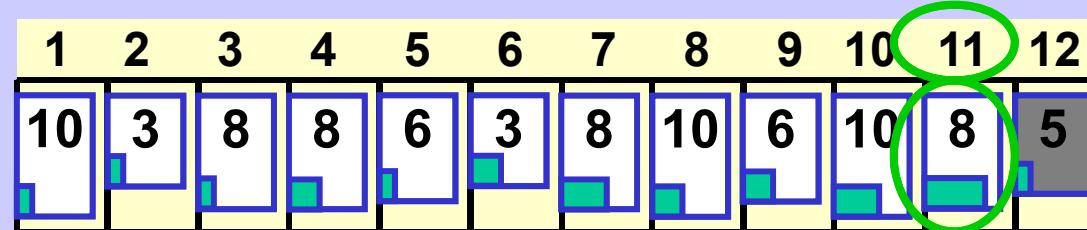


## Counting sort

Krok 3

$i == N-1$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnosc[vstup[i]]] = vstup[i];
    cetnosc[vstup[i]]--;
}
```

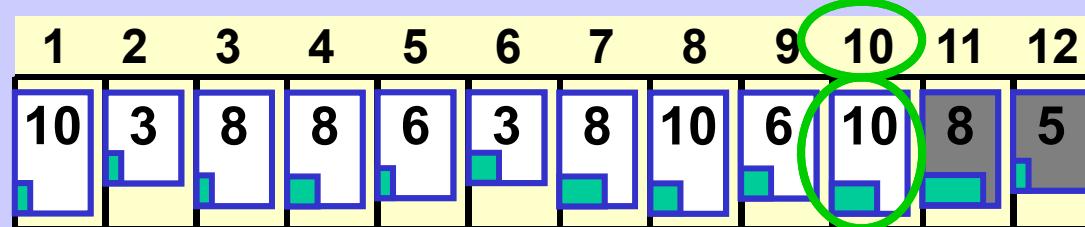


## Counting sort

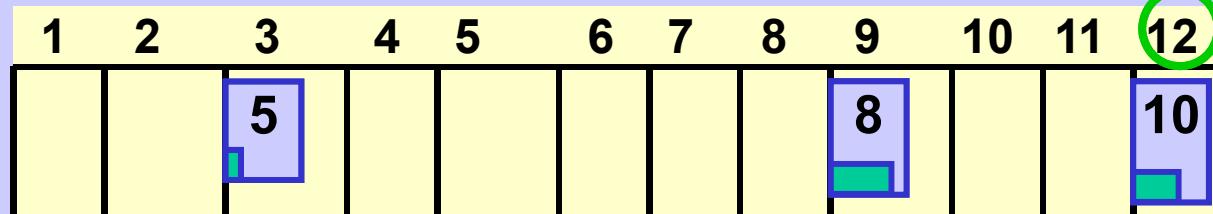
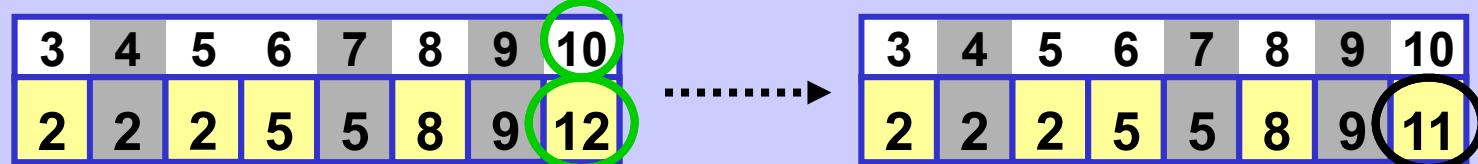
Krok 3

$i == N-2$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnosc[vstup[i]]] = vstup[i];
    cetnosc[vstup[i]]--;
}
```



$i == 10$



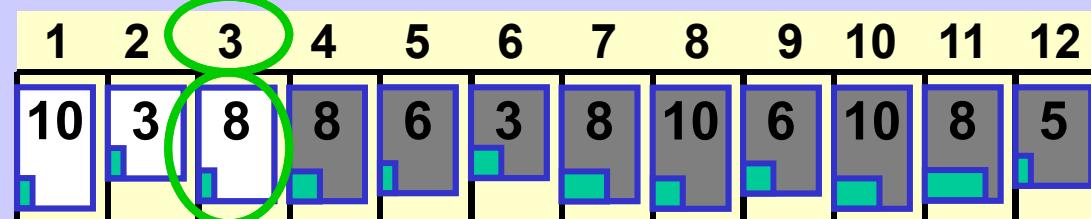
atd...

## Counting sort

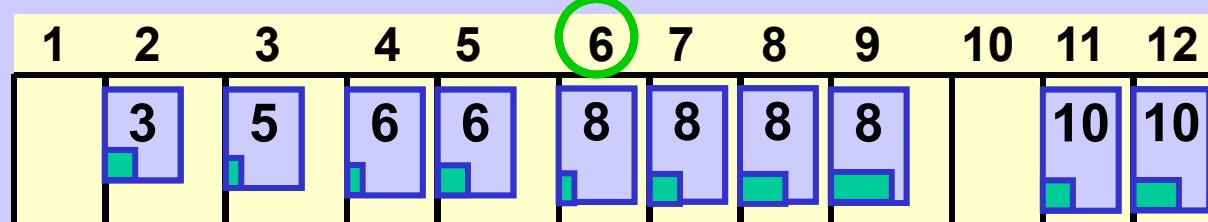
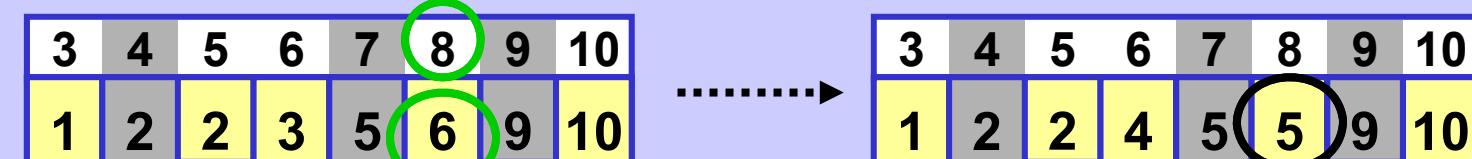
Krok 3

$i == 3$

```
for( int i = N; i > 0; i-- ) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
```



$i == 3$



atd...

## Orientační přehled vlastností řadících algoritmů

Velikost pole n	Nejhorší případ	Nejlepší případ	"typický", "běžný" případ	Stabilní	Všeobecné
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Ne	Ano
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Ano	Ano
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Ano	Ano
Quick sort	$\Theta(n^2)$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ne	Ano
Merge sort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ano	Ano
Heap sort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	Ne	Ano
Radix sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano	Ne
Counting sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano	Ne

## Ilustrační experiment řazení

### Prostředí

Intel(R) 1.8 GHz, Microsoft Windows XP SP3, jdk 1.6.0\_16.

### Organizace

Použita řazení, v nichž se prvky (double) navzájem porovnávají.

Kódy převzaty z přednášky (žádné další triky).

Řazení v jednotlivých velikostech prováděna na stejných datech.

Pole náhodně zamíchána generátorem pseudonáhodných čísel  
s rovnoměrným rozložením.

Výsledky průměrovány přes větší počet běhů.

### Závěr

Neexistuje jedno univerzální řazení, které by bylo optimální  
za všech okolností.

Hraje roli stabilita, velikost dat i stupeň předběžného seřazení dat.

## Ilustrační experiment řazení

Délka pole	% seř.	Doba běhu v milisekundách, není-li uvedeno jinak					
		Sort					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	0.0005	★ 0.0002	0.0005	0.0004	0.0009	0.0005
10	90%	0.0004	★ 0.0001	0.0004	0.0004	0.0007	0.0005
100	0%	0.028	0.016	0.043	0.081	0.014	★ 0.011
100	90%	0.026	★ 0.003	0.030	0.010	0.011	0.011
1 000	0%	2.36	1.30	4.45	★ 0.12	0.19	0.17
1 000	90%	2.31	0.18	2.86	0.16	★ 0.15	0.16
10 000	0%	228	130	450	★ 1.57	2.40	2.31
10 000	90%	229	17.5	285	1.93	★ 1.68	2.11
100 000	0%	22 900	12 800	45 000	★ 18.7	31.4	31.4
100 000	90%	22 900	1 760	28 500	27.4	★ 24.6	25.5
1 000 000	0%	38 min	22 min	75 min	★ 237	385	570
1 000 000	90%	38 min	2.9 min	47.5 min	336	★ 301	381

Seřazená část pole. Pole je nejprve seřazeno, pak náhodně vybrané prvky změně libovolně hodnotu.

## Ilustrační experiment řazení

Délka pole	% seř.	Koeficient zpomalení ( >1 ) vůči Quick sortu pro danou velikost dat a předběžné seřazení					
		Sort					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	1.3	0.7	1.4	1	2.5	1.4
10	90%	1	0.26	0.96	1	1.8	1.3
100	0%	3.4	1.8	5.4	1	1.75	1.35
100	90%	2.46	0.28	2.9	1	1.07	1.07
1 000	0%	20	11	37.5	1	1.65	1.4
1 000	90%	15	1.2	18.5	1	0.95	1.03
10 000	0%	146	83	287	1	1.53	1.48
10 000	90%	118	9.1	148	1	0.87	1.09
100 000	0%	1 220	686	2 410	1	1.7	1.7
100 000	90%	837	64.1	1 040	1	0.9	0.93
1 000 000	0%	9 960	5 400	19 000	1	1.6	2.41
1 000 000	90%	6 820	521	8 480	1	0.9	1.14

Nejrychlejší ★

Nejpomalejší ✗

Stabilní □

Select a Bubble sorty nesoutěží.

## Ilustrační experiment řazení

Délka pole	% seř.	Koeficient zpomalení ( $> 1$ ) při srovnání rychlosti řazení neseřazeného a částečně seřazeného pole					
		Sort					
		Select	Insert	Bubble	Quick	Merge	Heap
10	0%	1	1	1	1	1	1
10	90%	0.8	0.5	0.8	1	0.8	1
100	0%	1	1	1	1	1	1
100	90%	0.9	0.2	0.68	1.27	0.78	1
1 000	0%	1	1	1	1	1	1
1 000	90%	0.98	0.14	0.64	1.31	0.75	0.95
10 000	0%	1	1	1	1	1	1
10 000	90%	1.0	0.14	0.63	1.23	0.7	0.91
100 000	0%	1	1	1	1	1	1
100 000	90%	1.0	0.14	0.63	1.46	0.78	0.81
1 000 000	0%	1	1	1	1	1	1
1 000 000	90%	1.0	0.14	0.63	1.42	0.78	0.67

Stabilní

# ALG 10

## Dynamické programování

zkratka: DP

Zdroje, přehledy, ukázky viz

[https://cw.fel.cvut.cz/wiki/courses/a4b33alg/literatura\\_odkazy](https://cw.fel.cvut.cz/wiki/courses/a4b33alg/literatura_odkazy)

## Dynamické programování

### Příklady aplikací:

- Optimální rozvrhování navazujících procesů
- Přibližné vyhledávání v textu daných vzorků (bioinformatika)
- Optimální plnění ruksaku (kontejneru, nádob...)
- Hledání optimálních cest/spojení v grafech, sítích...
- Nejdelší podposloupnosti s předepsanými vlastnostmi
- Nejdelší společná podposloupnost
- Optimální pořadí násobení matic
- Optimální vyhledávací strom
- Optimální vrcholové pokrytí hran stromu
- Množství dalších....

## Dynamické programování

### Charakteristika

**Neřeší jeden konkrétní typ úlohy, je to všeobecná strategie (podobně jako Rozděl a panuj) pro řešení převážně optimalizačních úloh z různých oblastí tvorby algoritmů.**

### Významné vlastnosti

- 1. Hledané optimální řešení lze sestavit z vhodně volených optimalních řešení téže úlohy nad redukovanými daty.**
  
- 2. V rekurzivně formulovaném postupu řešení se opakovaně objevují stejně menší podproblémy.**  
DP umožňuje obejít opakovaný výpočet většinou jednoduchou tabelací výsledků menších podproblémů.

# Dynamické programování

## Seznam DP algoritmů na [en.wikipedia.org/wiki/Dynamic\\_programming](https://en.wikipedia.org/wiki/Dynamic_programming)

- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted
- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Ilustrační otisk obrazovky

## Tabelace v DP - příklad

**Definice funkce**

$$f(x,y) = \begin{cases} 1 & (x = 0) \text{ } || \text{ } (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1, y) & (x > 0) \text{ } \&\& (y > 0) \end{cases}$$

**Otzáka**

$$f(10,10) = ?$$

**Program**

```
int f( int x, int y ) {
    if ( (x == 0) || (y == 0) )
        return 1;
    return ( 2* f(x, y-1) + f(x-1,y) );
}
```

```
print( f(10,10) );
```

**Odpověď**

$$f(10,10) = 127\ 574\ 017$$



## Tabelace v DP - příklad

Jednoduchá analýza

```
int count = 0;  
  
int f( int x, int y ){  
    count++;  
    if ( (x == 0) || (y == 0) )  
        return 1;  
    return ( 2* f(x, y-1) + f(x-1,y) );  
}
```

```
xyz = f( 10,10 );  
print( count );
```

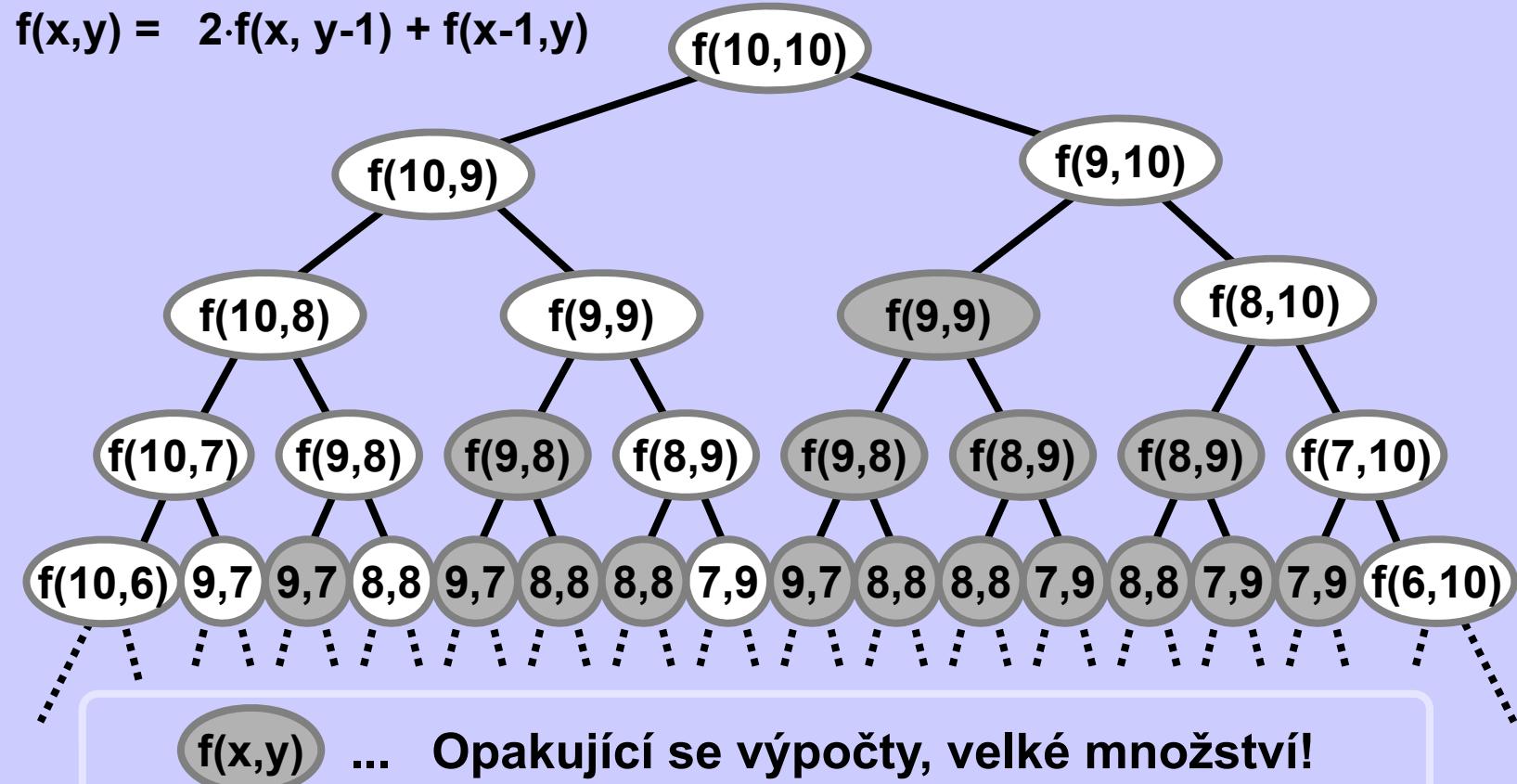
Výsledek analýzy

count = 369 511



## Tabelace v DP - příklad

### Detailnější analýza – strom rekurzivního volání

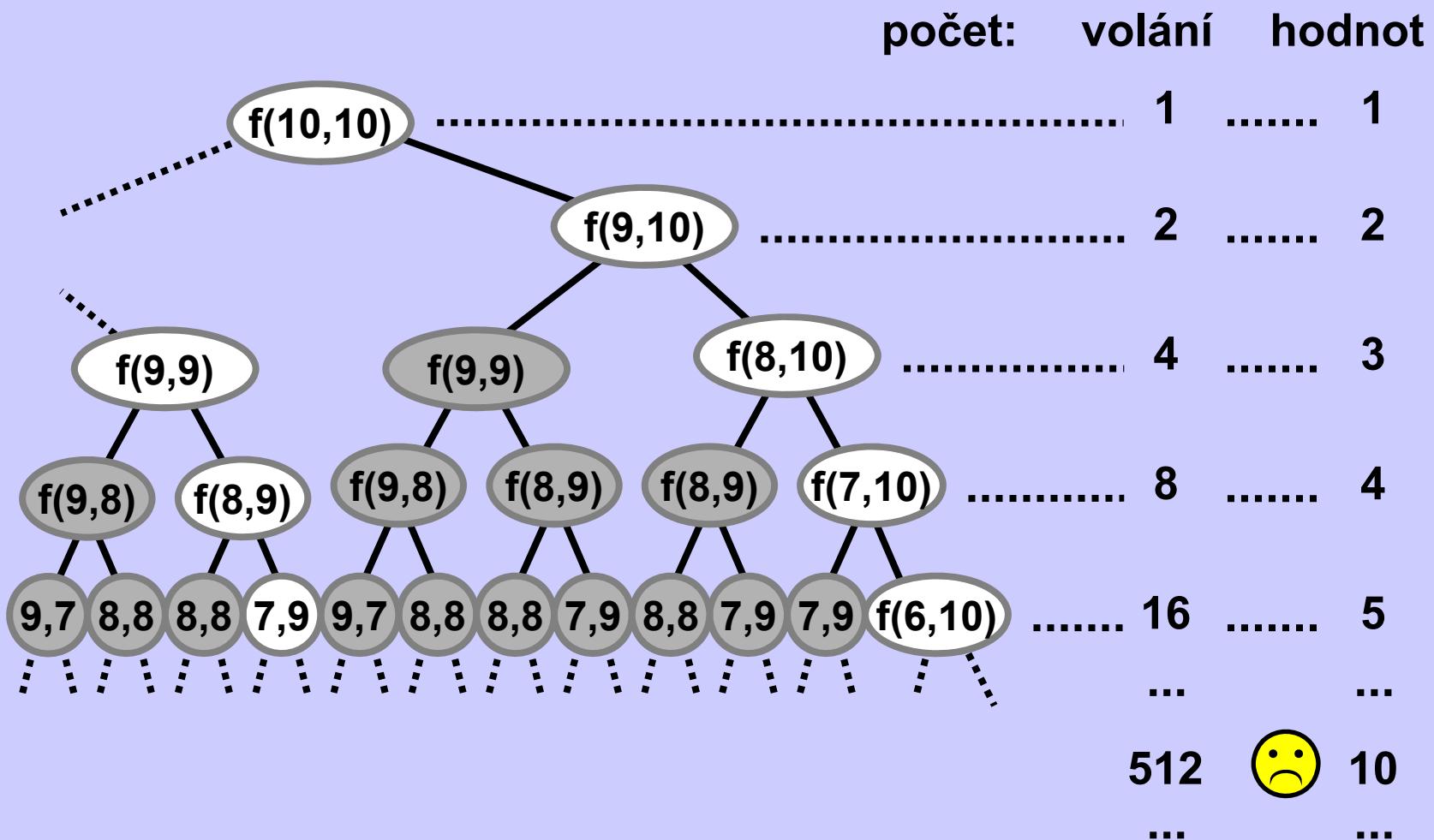


8,8  
9,7

zobrazeny jen parametry pro nedostatek místa

## Tabelace v DP - příklad

Detailnější analýza pokračuje – efektivita rekurzivního volání



## Tabelace v DP - příklad

$$f(x,y) = \begin{cases} 1 & (x = 0) \text{ || } (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1, y) & (x > 0) \text{ && } (y > 0) \end{cases}$$

## Tabulka všeobecně

A diagram illustrating a 2D grid of points  $f(x,y)$  for  $x$  and  $y$  ranging from 0 to 10. The grid is shown as a series of boxes, with the point  $f(x,y)$  highlighted in yellow. A double-headed arrow labeled "x2" indicates a horizontal shift of two units. A curved arrow shows a transition from  $f(x-1,y)$  to  $f(x,y)$ .

$x$	0	1	2	3	...	9	10	
0	$f(0,0)$	$f(1,0)$	$f(2,0)$	$f(3,0)$	...	$f(9,0)$	$f(10,0)$	
1	$f(0,1)$	$f(1,1)$	$f(2,1)$	$f(3,1)$	...	$f(9,1)$	$f(10,1)$	
2	$f(0,2)$	$f(1,2)$	$f(2,2)$	$f(3,2)$	...	$f(9,2)$	$f(10,2)$	
3	$f(0,3)$	$f(1,3)$	$f(2,3)$	$f(3,3)$	...	$f(9,3)$	$f(10,3)$	
9	$f(0,9)$	$f(1,9)$	$f(2,9)$	$f(3,9)$	...	$f(9,9)$	$f(10,9)$	
10	$f(0,10)$	$f(1,10)$	$f(2,10)$	$f(3,10)$	...	$f(8,10)$	$f(9,10)$	$f(10,10)$

## Tabelace v DP - příklad

$$f(x,y) = \begin{cases} 1 & (x = 0) \text{ || } (y = 0) \\ 2 \cdot f(x, y-1) + f(x-1, y) & (x > 0) \text{ && } (y > 0) \end{cases}$$

### Tabulka numericky

	0	1	2	3	4	9	10	x
0	1	1	1	1	1	...	1	
1	1	3	5	7	9	...	...	
2	1	7	17	31	...	...	...	
3	1	15	49	...	...	...	...	
4	1	31	...	...	...	...	...	
	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	
9	1	3	5	7	9	...	1	
10	1	15	49	31	...	...	8085505	
	...	...	...	...	...	...	16807935	32978945
	...	...	...	...	...	...	28000257	61616127
	...	...	...	...	...	...	127574017	
y								

A diagram illustrating the computation of the value at cell (4,4) in the table. The formula  $f(x,y) = 2 \cdot f(x, y-1) + f(x-1, y)$  is shown. The value 31 is calculated as  $2 \cdot f(3,3) + f(2,3)$ . The cell  $f(x-1, y)$  is highlighted in yellow, and the cell  $f(x, y-1)$  is highlighted in white with a black border. Arrows point from the formula terms to these specific cells.

## Tabelace v DP - příklad

Všechny hodnoty se předpočítají

```
int dynArr [N+1][N+1];

void fillDynArr(){

    for( int xy = 0; xy <= N; xy++ )
        dynArr[0][xy] = dynArr[xy][0] = 1;

    for( int y = 1; y <= N; y++ )
        for( int x = 1; x <= N; x++ )
            dynArr[y][x] = 2*dynArr[y-1][x] + dynArr[y][x-1];
}
```

Volání funkce

```
int f( int x, int y ){
    return dynArr[y][x];
}
```

## Hledání optimálních cest v grafu

### Značení

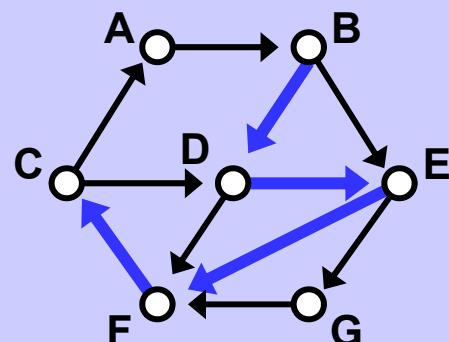
**Graf  $G = (V, E)$** , množina uzlů resp. hran:  $V(G)$  resp.  $E(G)$ ,  
 $N = |V(G)|$ ,  $M = |E(G)|$ , případně  $n = |V|$ ,  $m = |E(G)|$  apod.

### Cesta v grafu

= posloupnost na sebe navazujících hran,  
 která prochází každým uzlem nejvýše jednou.

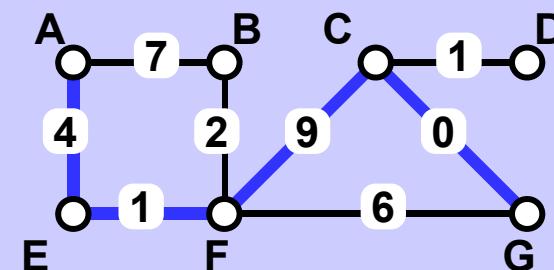
**Délka cesty v neváženém grafu**  
 = počet hran na cestě.

Př. Délka (B D E F C) = 4.



**Délka cesty ve váženém grafu**  
 = součet vah hran na cestě.

Př. Délka (A E F C G) = 14.



## Hledání optimálních cest v grafu

### Nejkratší cesty

Úloha nalezení nejkratší cesty mezi dvěma danými uzly, případně mezi některými dvojicemi uzelů nebo mezi všemi dvojicemi uzelů.

(Například minimalizace nákladů na přesun z X do Y.)

### Postupy

Je vyřešena úspěšně pro všechny praktické případy.

V neváženém obecném grafu známe BFS, pro jiné případy, zejména vážených grafů, existují specializované algoritmy -- Dijkstra, Floyd-Warshall, Johnson, Bellman-Ford, atd.

### Složitost

Asymptotická složitost je vždy polynomiální v počtu uzelů a hran, typicky nalezení jedné cesty má složitost nanejvýš  $O(N^2)$ , kde N je počet uzelů grafu.

## Hledání optimálních cest v grafu

### Nejdelší cesty

Úloha nalezení nejdelší cesty v grafu mezi dvěma danými uzly, nebo v celém grafu vůbec.

(Například maximalizace zisků při provádění navzájem závislých činností.)

Exponenciální složitost  
NP - těžký problém

Není dosud uspokojivě vyřešena v plné obecnosti.

### Možné strategie

1. Brute force -- exponenciální složitost, pro  $N > \text{cca } 30$  bezcenná.
2. Algoritmy přibližného řešení s polynomiální složitostí
  - bud' najdou optimum jen s určitou pravděpodobností
  - nebo zaručí jen nalezení suboptimálního řešení
  - typicky jsou netriviální a náročné na správnou implementaci.

## Hledání nejdelších cest v grafu

### Možné strategie

**3. Specifické typy grafů dovolují použít efektivní specifický algoritmus.**

#### Nejjednodušší případ

**3A.**

**Graf je strom (vážený ano i ne, orientovaný ano i ne).**

**Nejdelší cestu lze najít, např. s pomocí průchodu postorder, vždy v čase  $\Theta(N)$ .**

#### Příležitost pro DP

**3B.**

**Graf je orientovaný, acyklický, vážený ano i ne.**

**Standardní označení: DAG (Directed Acyclic Graph)**

## Topologické uspořádání DAG

Topologické uspořádání uzelů DAG je takové pořadí jeho uzelů, ve kterém každá hrana vede z uzlu s nižším pořadím do uzlu s vyšším pořadím.

Každý DAG lze topologicky uspořádat, většinou více způsoby.

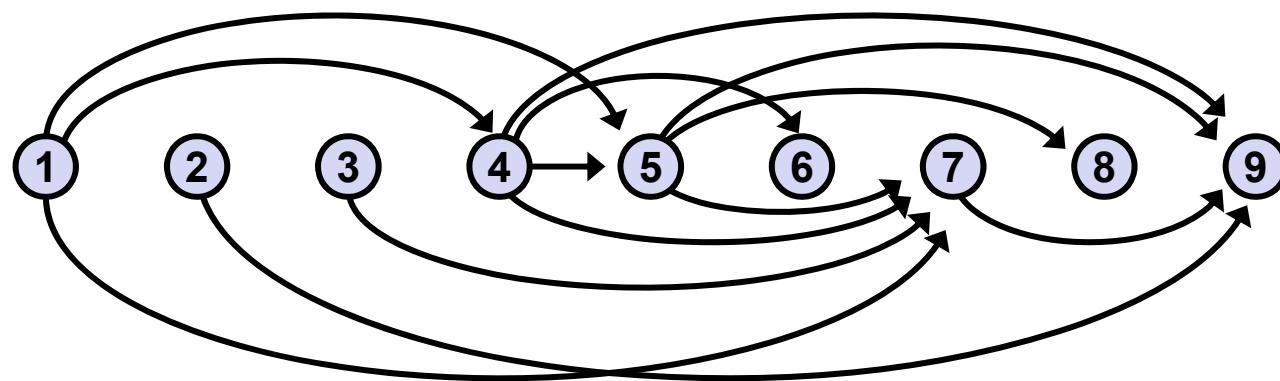
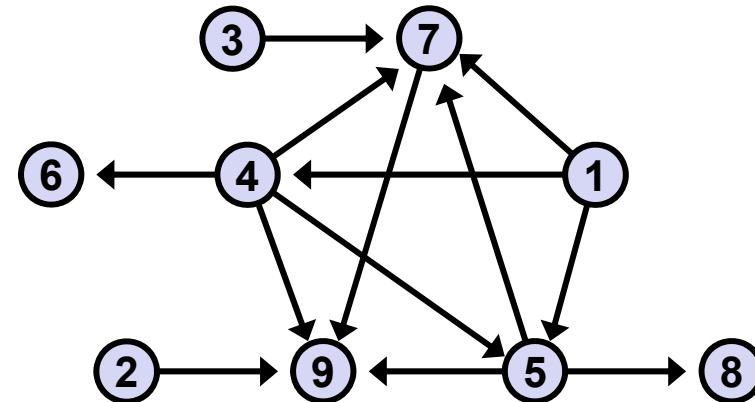
Orientovaný graf s alespoň jedním cyklem nelze topologicky uspořádat.

Mnoho úloh DP obsahuje DAG na vstupu již topologicky uspořádaný.

Topologické uspořádání DAG (alespoň jedno) lze sestavit v čase  $\Theta(M)$ , tj. v čase úměrném počtu hran DAG.

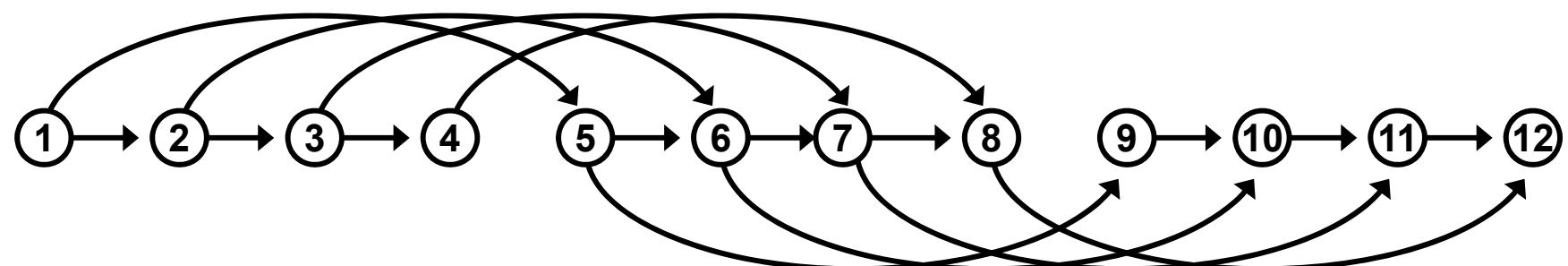
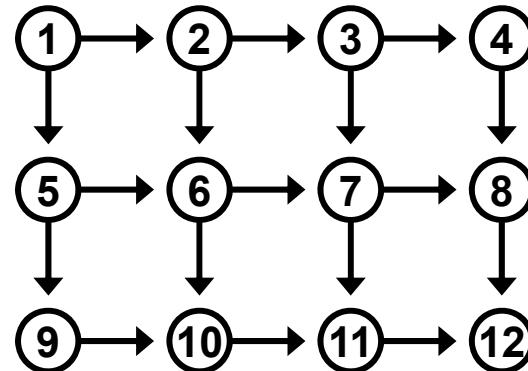
## DAG a jeho topologické uspořádání

### Příklad 1



## DAG a jeho různá topologická uspořádání

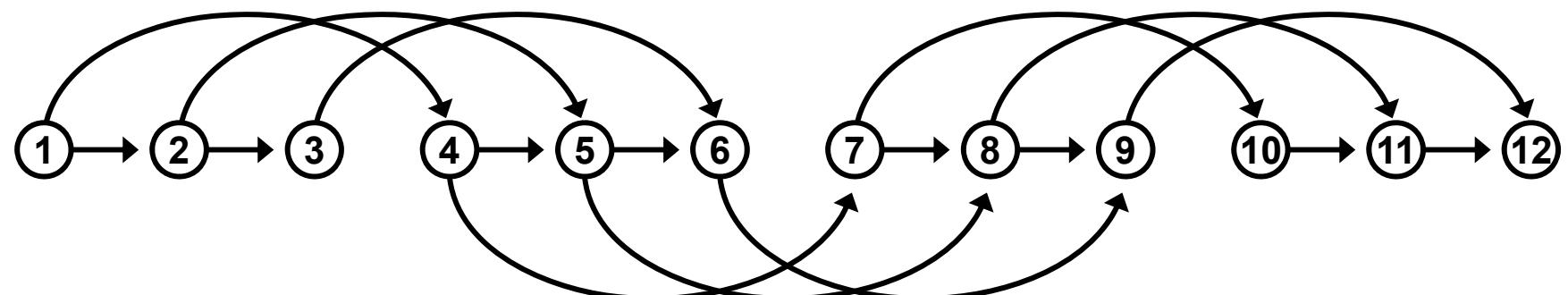
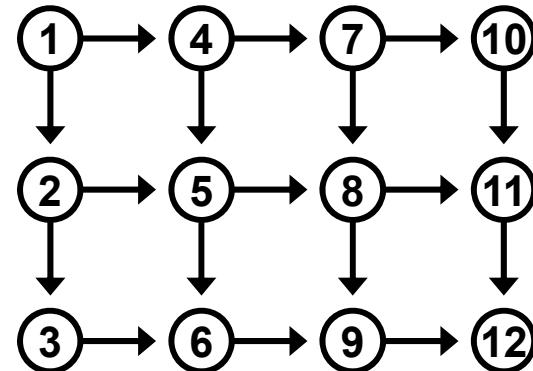
### Příklad 2a



V uzlu je zapsáno jeho pořadí v topologickém uspořádání

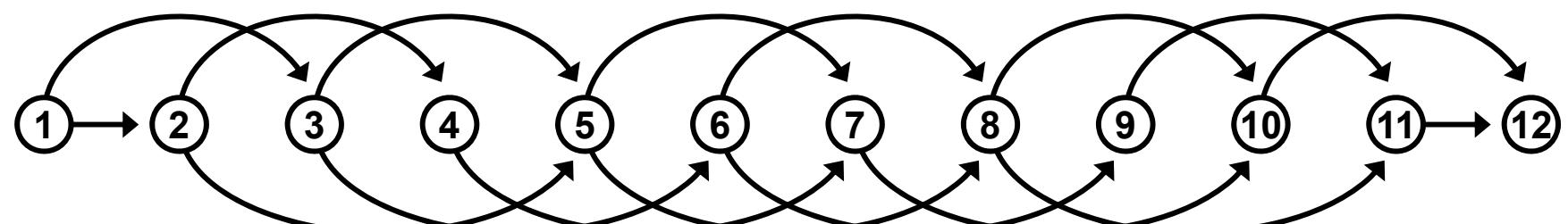
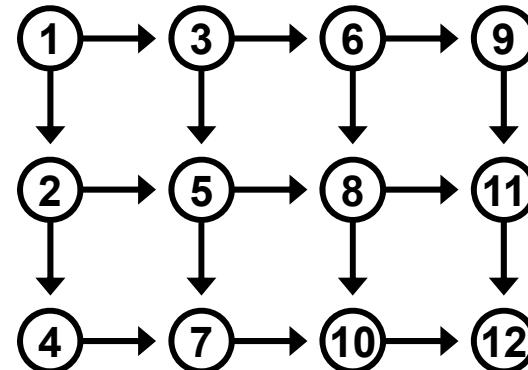
## DAG a jeho různá topologická uspořádání

### Příklad 2b



## DAG a jeho různá topologická uspořádání

### Příklad 2c



# Topologické uspořádání DAG

## Algoritmus

```

0. new queue Q of Node
   counter = 0

1. for each x in V(G)
   if (x.indegree == 0) // x is a root
     Q.insert(x)
     x.toporder = counter++

2. while (!Q.empty()) {
   Node v = Q.pop()
   for each edge (v, w) in E(G) {
     G.removeEdge((v, w))
     if (w.indegree == 0) // w is a root
       Q.insert(w)
       w.toporder = counter++
   }
}

```

## Složitost

Předpokládáme, že operace  
`G.removeEdge((v, w))`  
má konstantní složitost \*).

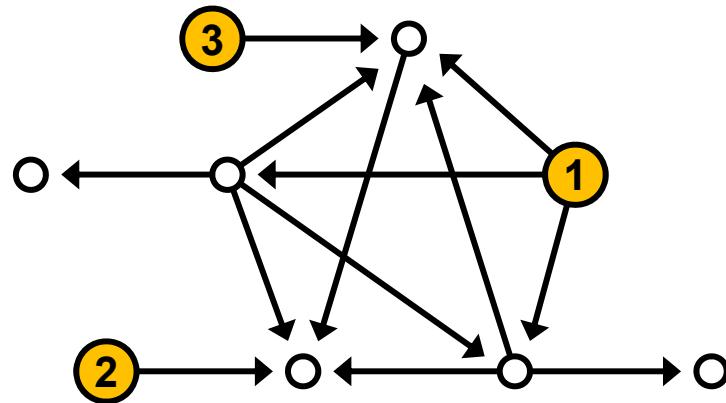
- 0. Složitost  $O(N)$
- 1. Složitost  $\Theta(N)$
- 2. Složitost  $\Theta(M)$ ,  
každá hrana je navštívena  
právě jednou  
a zpracována v konstantním čase.

Složitost:  $\Theta(N+M)$

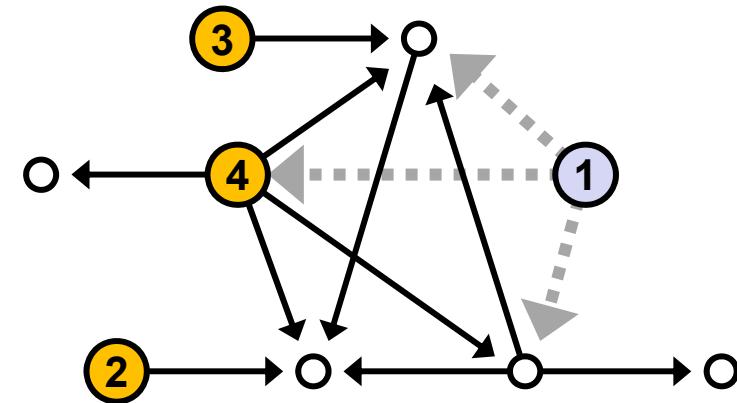
\*) Hranu fyzicky neodstraňujeme,  
jen ji vhodně označíme a změníme  
charakteristiky obou krajních uzlů.

Pořadí, ve kterém se uzly vkládají do fronty, určuje topogické uspořádání DAG.

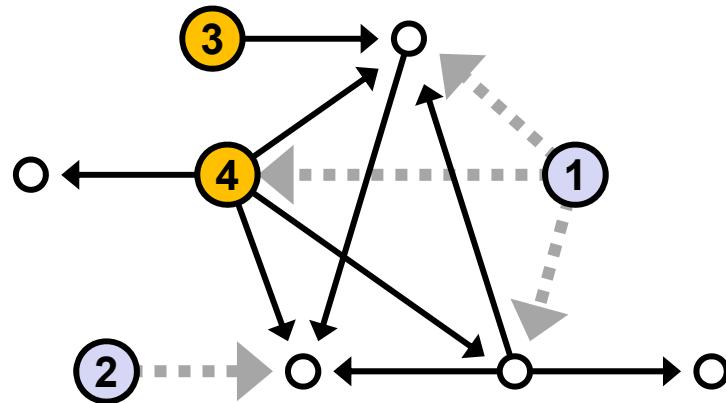
## Topologické uspořádání DAG - příklad



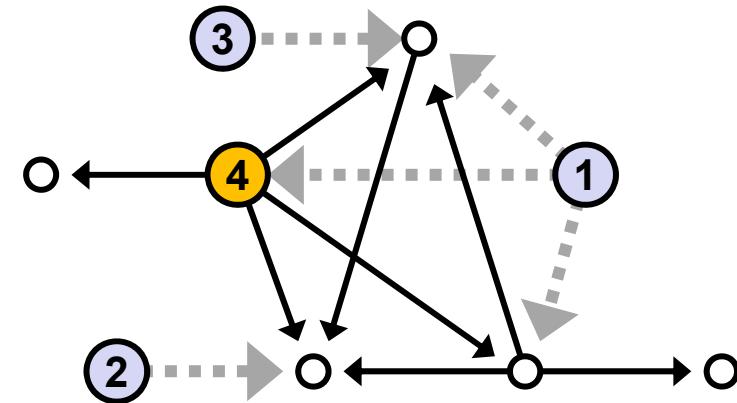
Queue: 1, 2, 3.



Queue: 2, 3, 4.

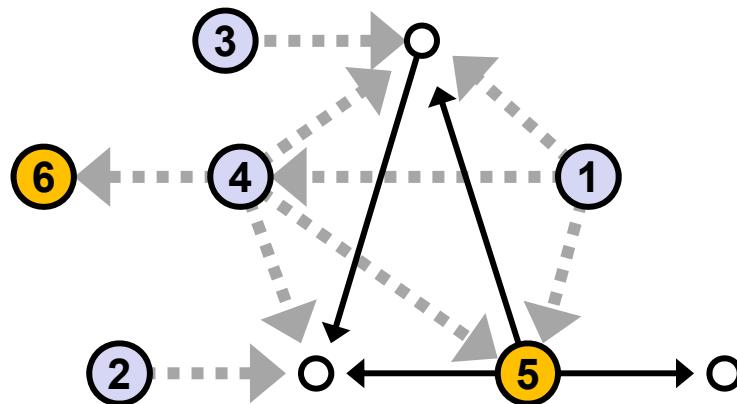


Queue: 3, 4.

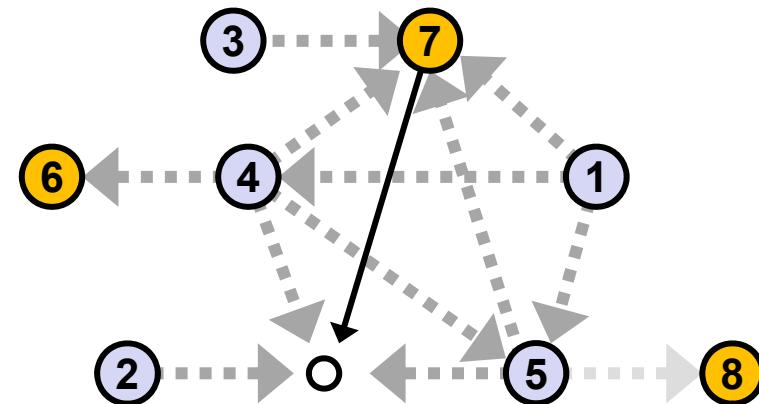


Queue: 4.

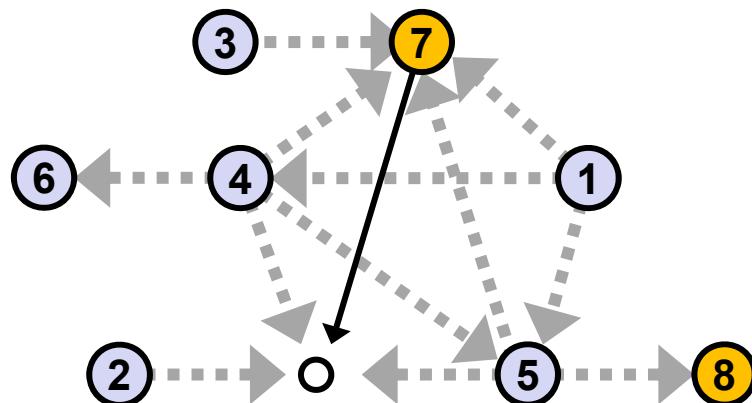
## Topologické uspořádání DAG - příklad



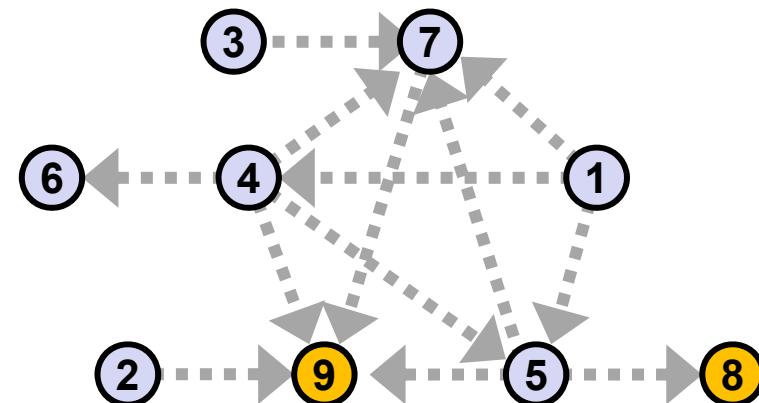
Queue: 5, 6.



Queue: 6, 7, 8.

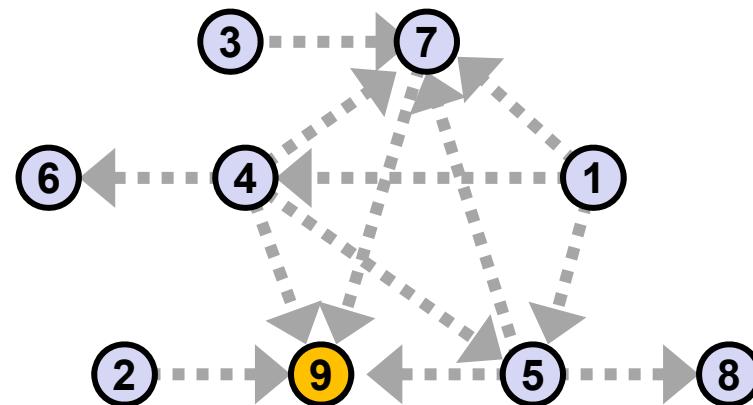


Queue: 7, 8.

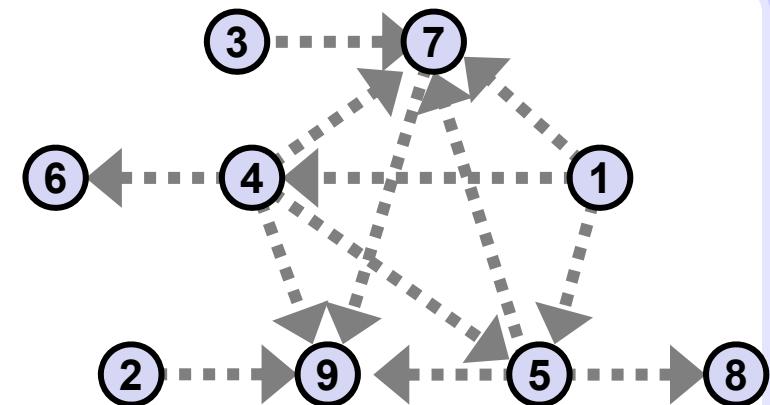


Queue: 8, 9.

## Topologické uspořádání DAG - příklad

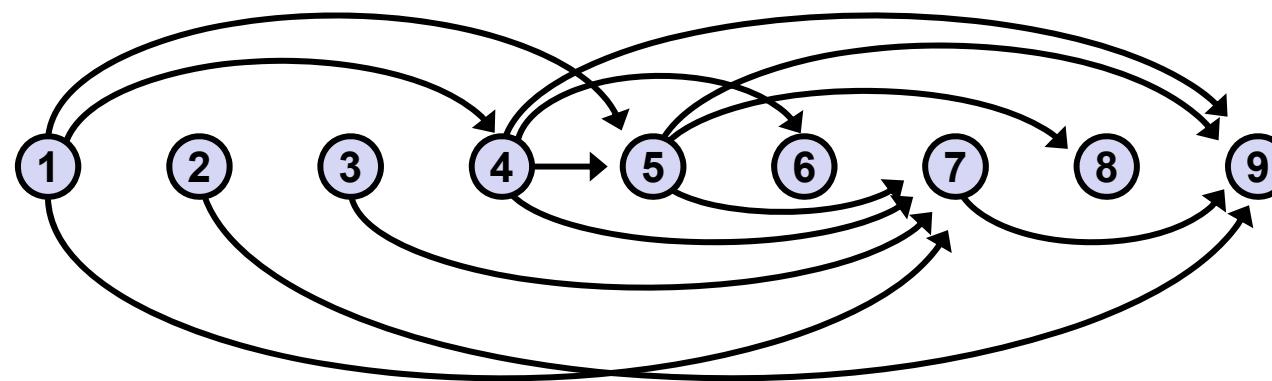


Queue: 9.



Queue: Empty.

Topologické uspořádání

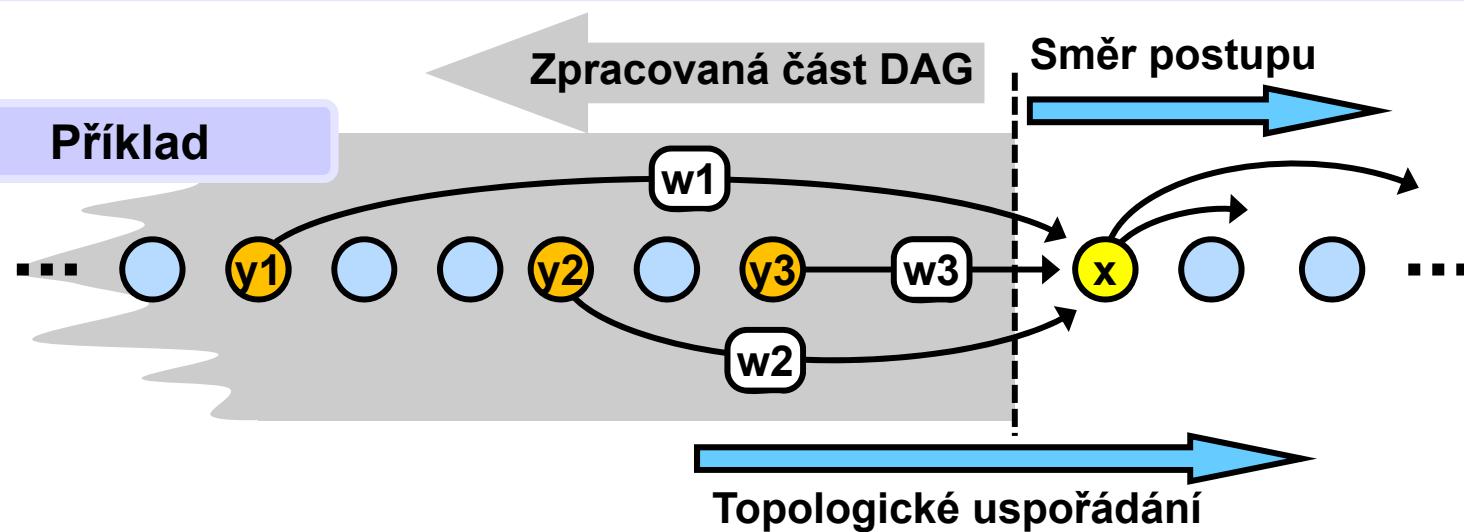


## Nejdelší cesta v DAG

Předpokládáme topologické uspořádání a v jeho směru procházíme DAG. Označme  $d[x]$  délku té cesty v DAG, která končí v  $x$  a je nejdelší možná.

Charakteristický pohled "odzadu dopředu":

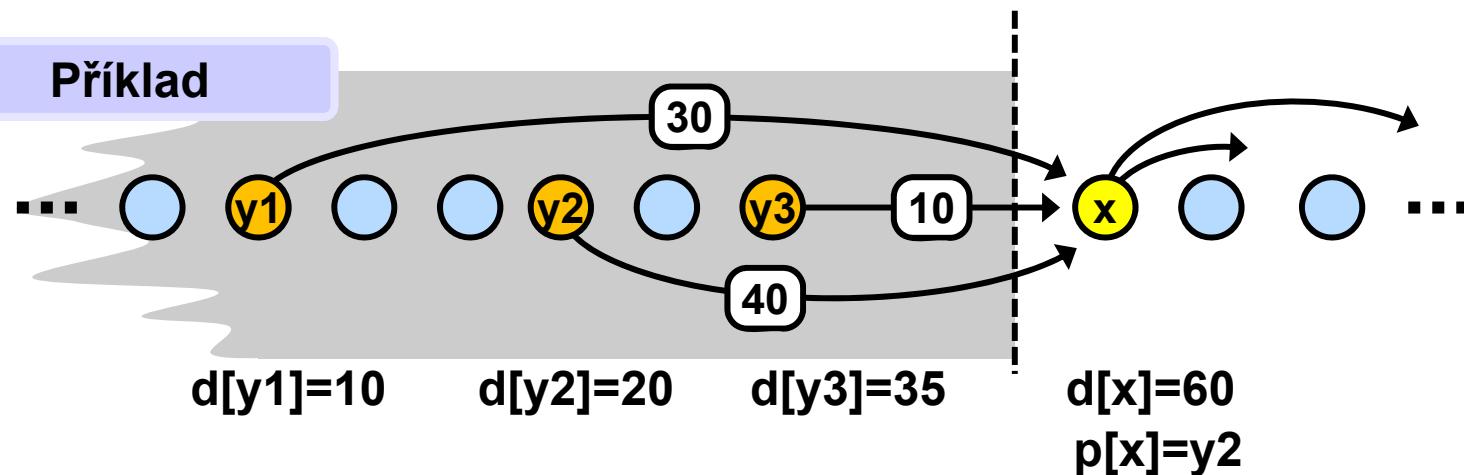
- $d[x]$  určujeme až v okamžiku, kdy jsou známy hodnoty  $d$  pro všechny předchozí (= již zpracované) uzly v topologickém uspořádání.
- $d[x]$  určíme jako maximum z hodnot
  $\{ d[y_1] + w_1, d[y_2] + w_2, \dots, d[y_k] + w_k \}$ ,  
 kde  $(y_1, x), (y_2, x), \dots, (y_k, x)$  jsou všechny hrany končící v  $x$   
 a  $w_1, w_2, \dots, w_k$  jsou jejich odpovídající váhy.



## Nejdelší cesta v DAG

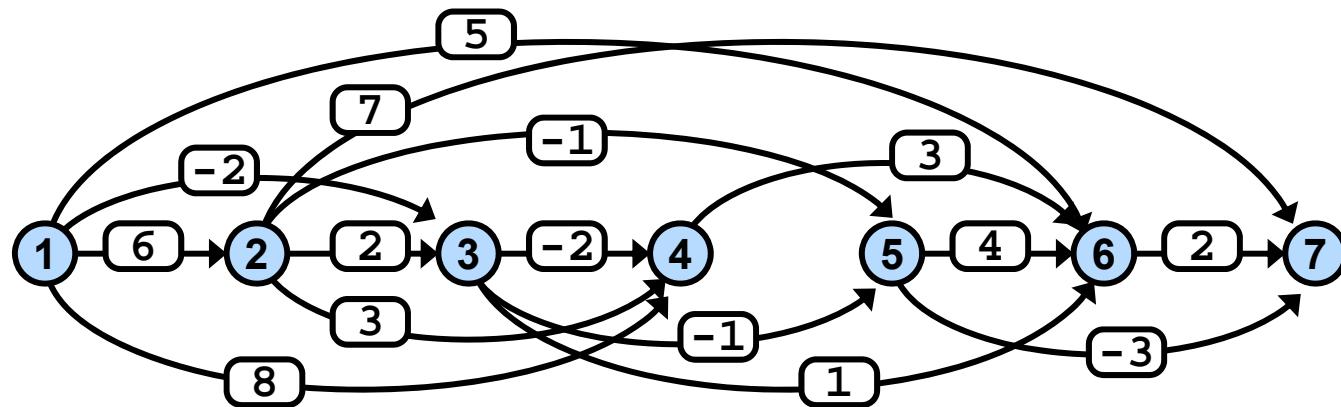
- $d[x]$  určíme jako maximum z hodnot  
 $\{ d[y_1] + w_1, d[y_2] + w_2, \dots, d[y_k] + w_k \}$ ,  
 kde  $(y_1, x), (y_2, x), \dots, (y_k, x)$  jsou všechny hrany končící v  $x$   
 a  $w_1, w_2, \dots, w_k$  jsou jejich odpovídající váhy.
- Uzel  $y_j$ , pro který je hodnota  $d[y_j] + w_j$  maximální a nezáporná,  
 ustavíme předchůdcem  $x$  na hledané nejdelší cestě.
- Pokud jsou všechny hodnoty  $\{d[y_1] + w_1, d[y_2] + w_2, \dots, d[y_k] + w_k\}$   
 záporné, nepřispívají do nejdelší cesty, pak položíme  $d[x] = 0$ ,  
 předchůdce  $x = \text{null}$ .

### Příklad



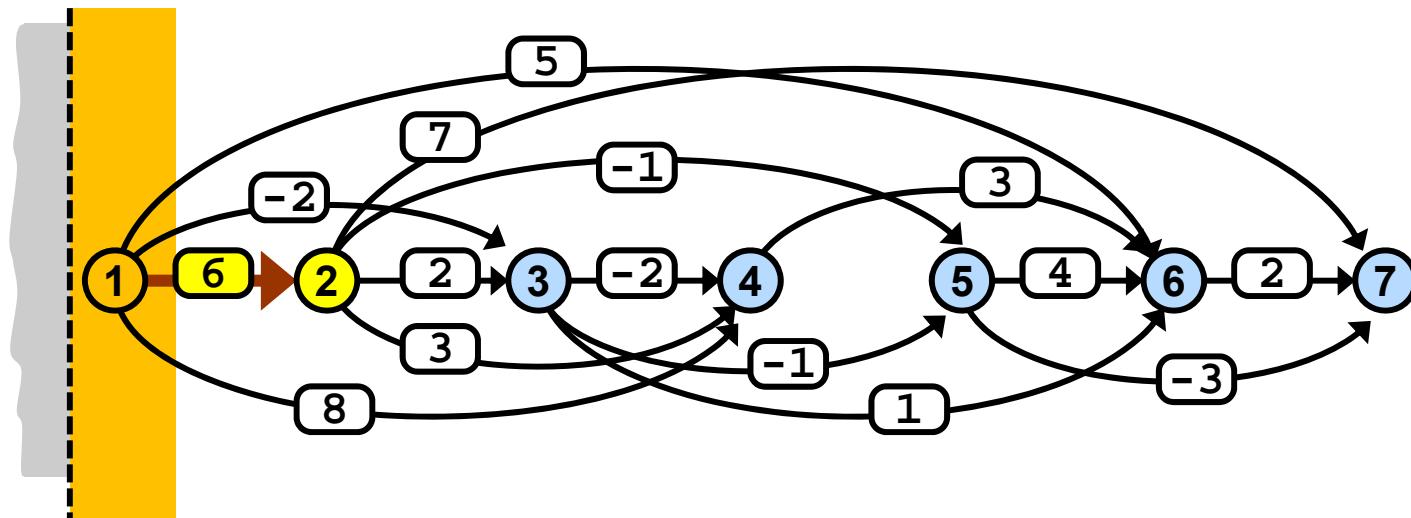
## Nejdelší cesta v DAG

### Příklad



Určete nejdelší cestu a její délku.

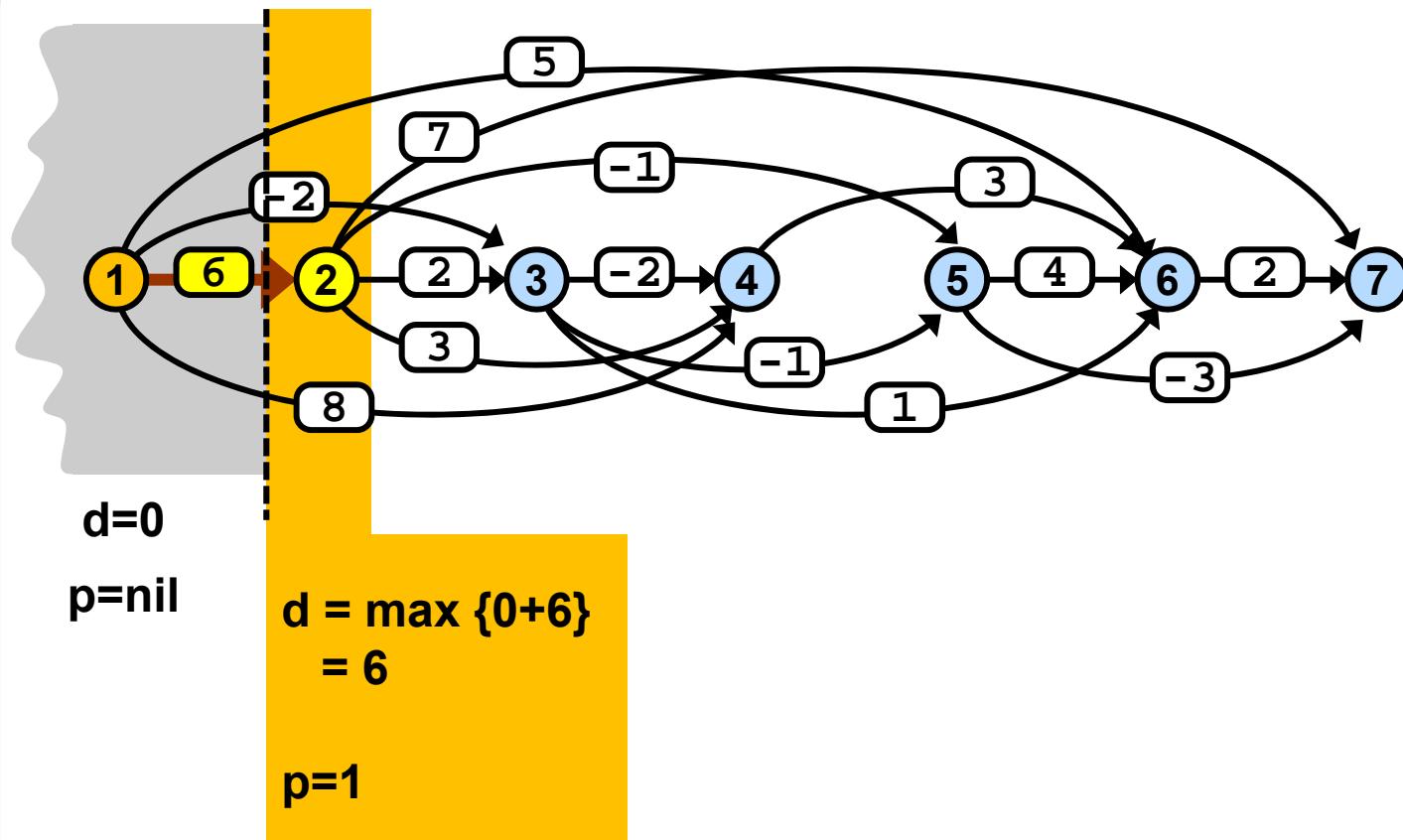
## Nejdelší cesta v DAG



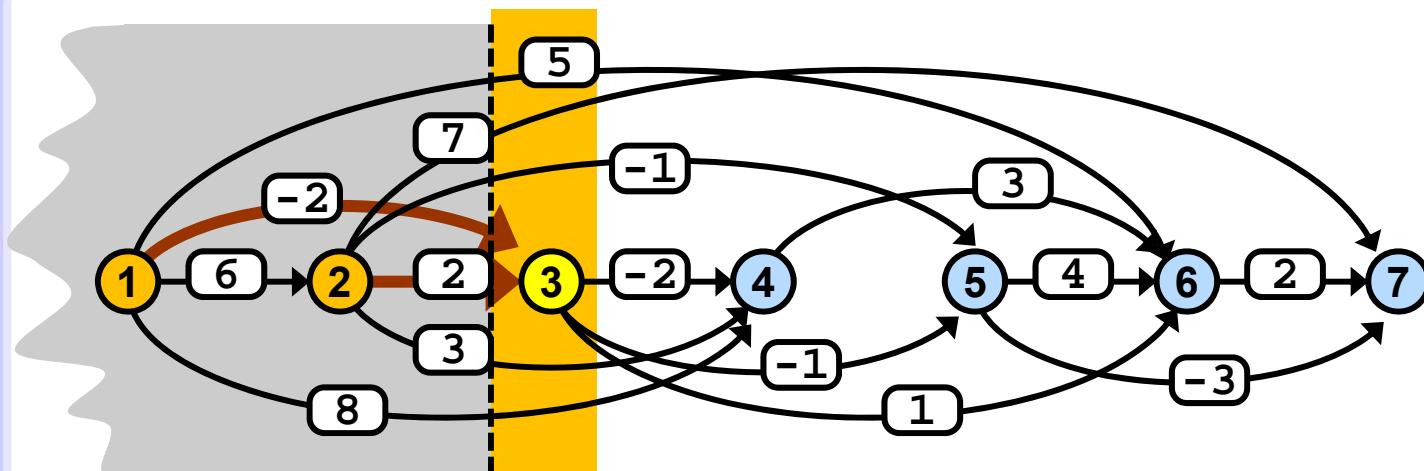
d=0

p=nil

## Nejdelší cesta v DAG



## Nejdelší cesta v DAG



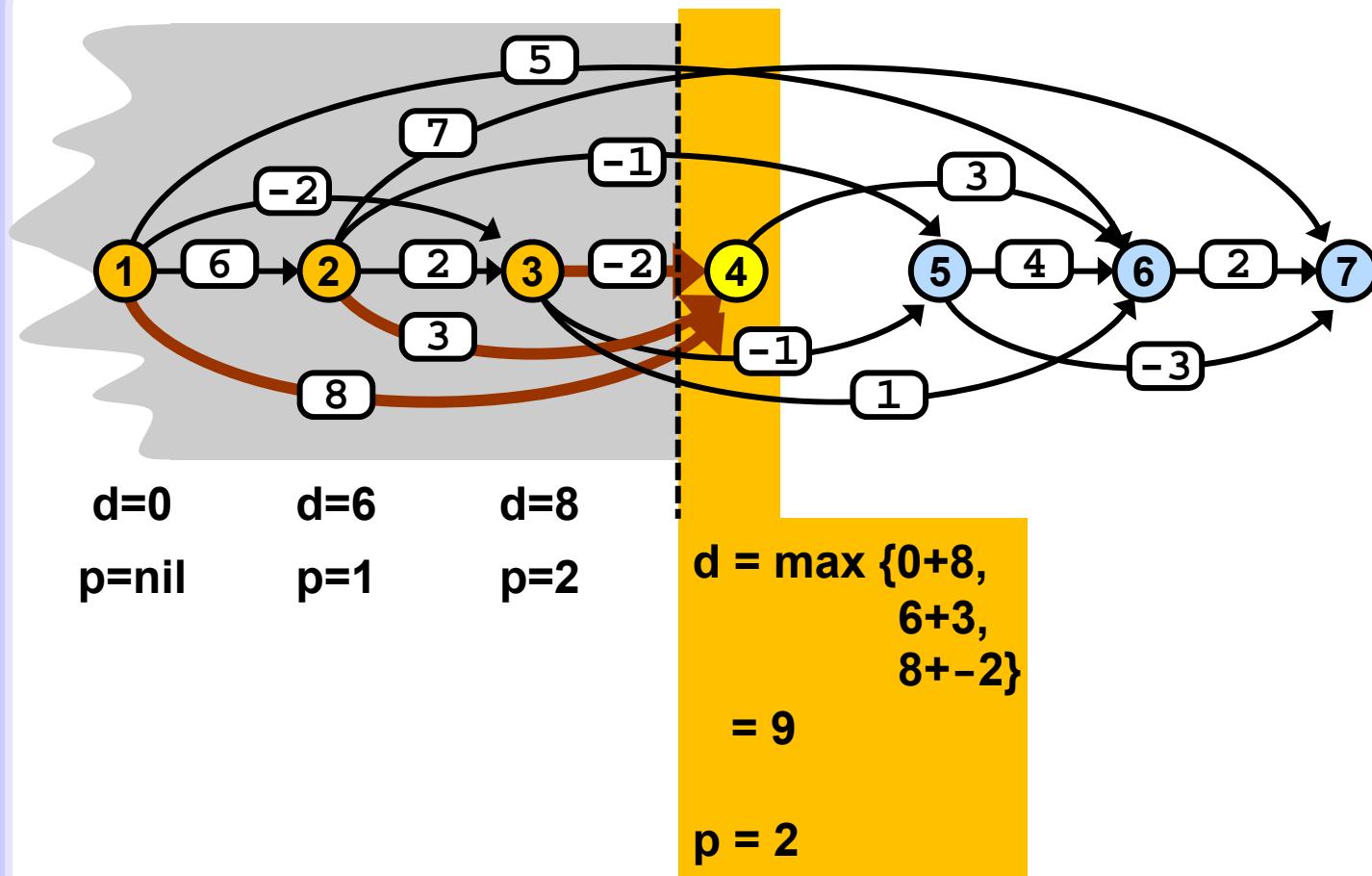
$d=0$   
 $p=nil$

$d=6$   
 $p=1$

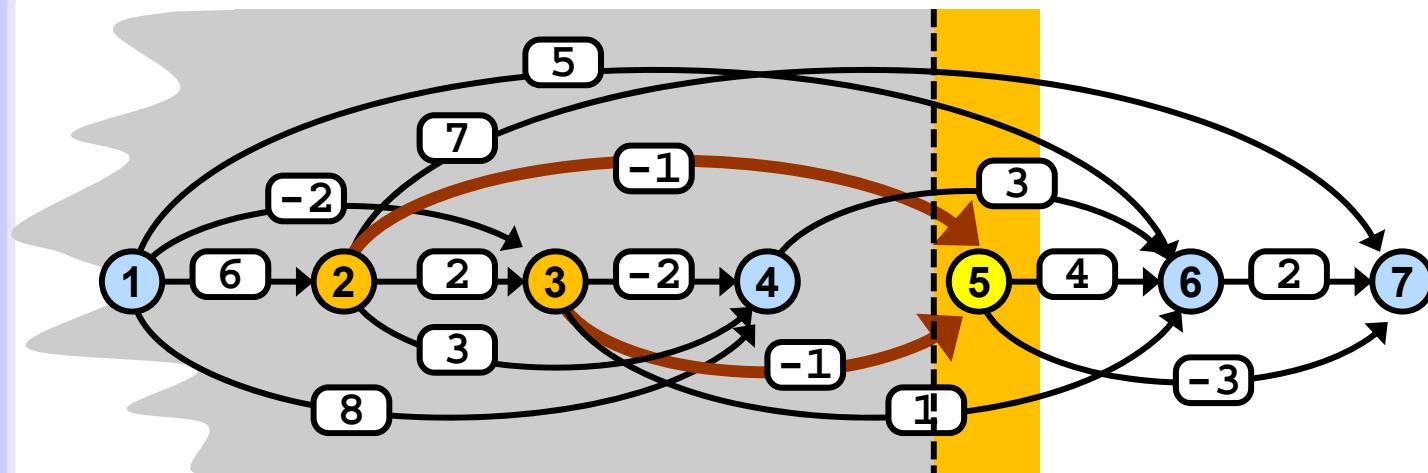
$$\begin{aligned} d &= \max \{0+(-2), \\ &\quad 6+2\} \\ &= 8 \end{aligned}$$

$p = 2$

## Nejdelší cesta v DAG



## Nejdelší cesta v DAG



$d=0$   
 $p=nil$

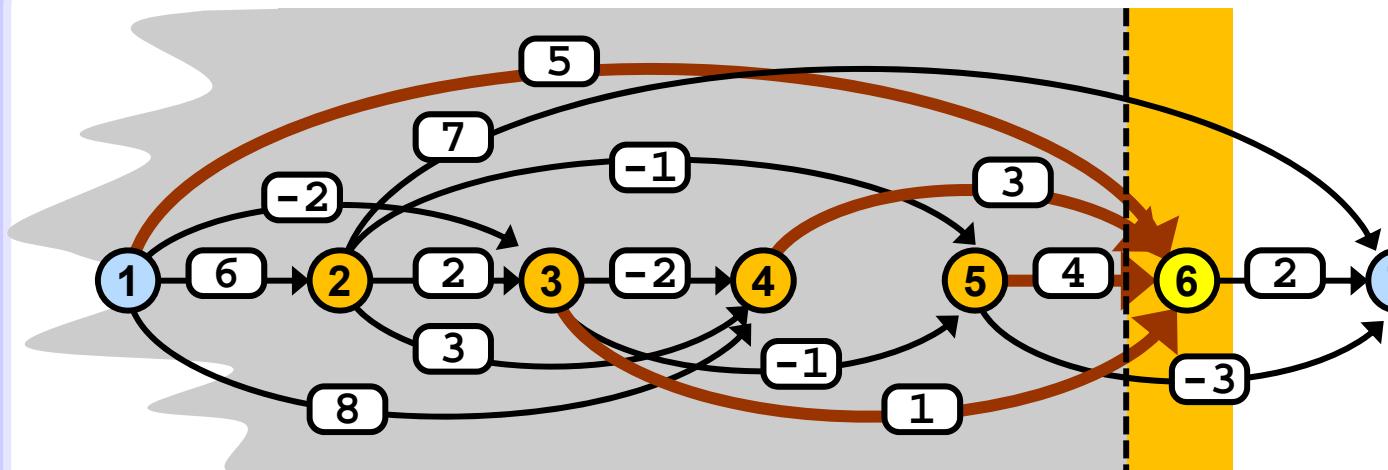
$d=6$   
 $p=1$

$d=8$   
 $p=2$

$d=9$   
 $p=2$

$$\begin{aligned}
 d &= \max \{6 + -1, \\
 &\quad 8 + -1\} \\
 &= 7 \\
 p &= 3
 \end{aligned}$$

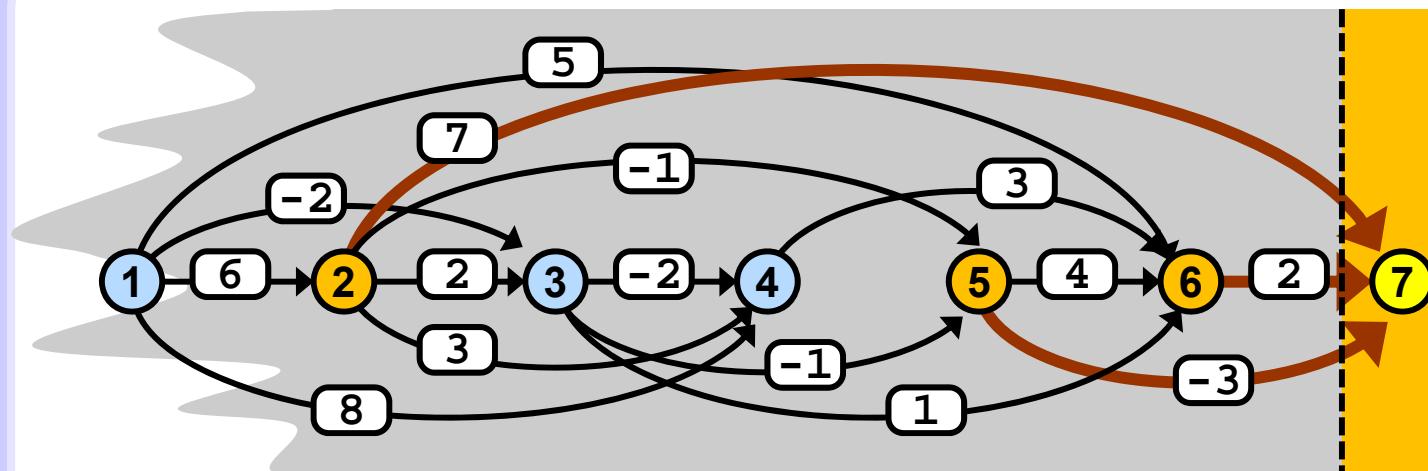
## Nejdelší cesta v DAG

 $d=0$  $p=\text{nil}$  $d=6$  $p=1$  $d=8$  $p=2$  $d=9$  $p=2$  $d=7$  $p=3$ 

$$\begin{aligned}
 d &= \max \{0+5, \\
 &\quad 8+1, \\
 &\quad 9+3, \\
 &\quad 7+4\} \\
 &= 12
 \end{aligned}$$

$$p = 4$$

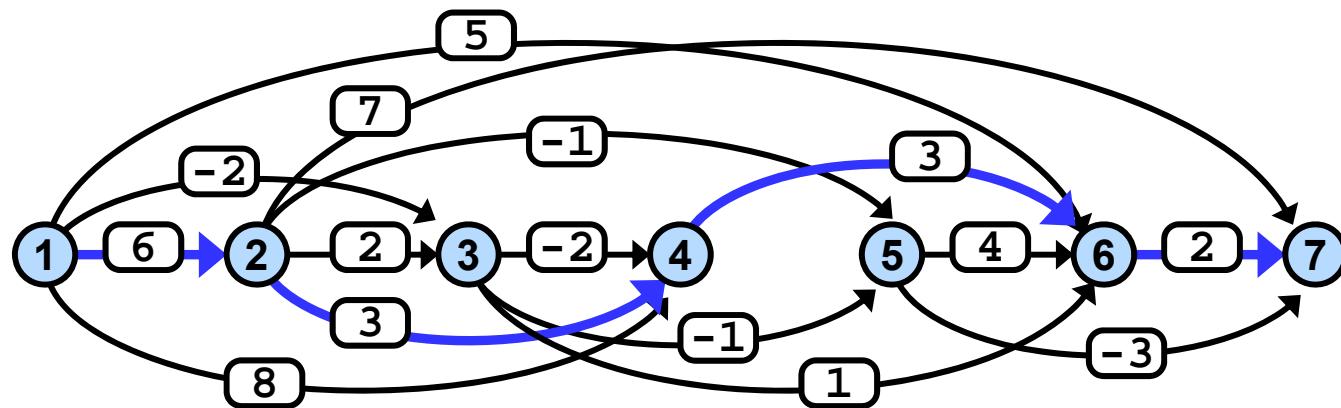
## Nejdelší cesta v DAG

 $d=0$  $p=\text{nil}$  $d=6$  $p=1$  $d=8$  $p=2$  $d=9$  $p=2$  $d=7$  $p=3$  $d=12$  $p=4$ 

$$d = \max \{6+7, 7+(-3), 12+2\} \\ = 14$$

 $p = 6$

## Nejdelší cesta v DAG



$d=0$	$d=6$	$d=8$	$d=9$	$d=7$	$d=12$	$d=14$
$p=nil$	$p=1$	$p=2$	$p=2$	$p=3$	$p=4$	$p=6$

Délka nejdelší cesty: 14  
 Nejdelší cesta: 1 -- 2 -- 4 -- 6 -- 7

## Nejdelší cesta v DAG

0. allocate memory for distance and predecessor of each node

```
1. for each node x in V(G) {
    x.dist = 0      // avoids negative path lengths
    x.pred = null
}
```

*// supposing nodes are processed  
// in ascending topological order*

```
2. for each node x in V(G) {
    for each edge e = (y, x)
    if (x. dist < y.dist + e.weight) {
        x. dist = y.dist + e.weight
        x.pred = y;
    }
}
```

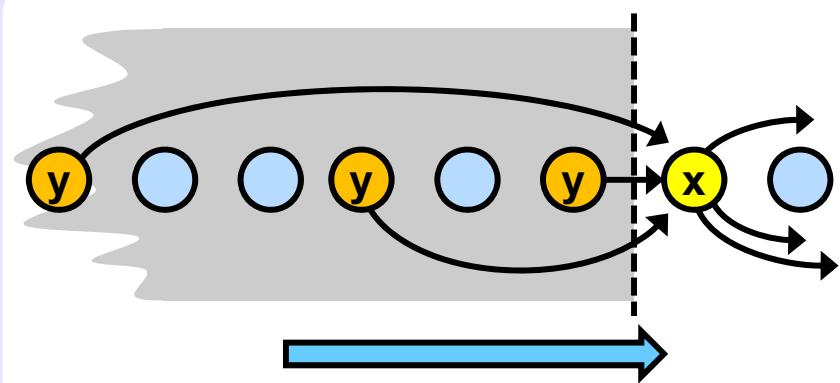
- 0. Složitost  $\Theta(N)$
- 1. Složitost  $\Theta(N)$
- 2. Složitost  $\Theta(M)$ ,  
každá hrana je navštívena  
právě jednou a zpracována  
v konstantním čase.

Složitost:  $\Theta(N+M)$

## Nejdelší cesta v DAG

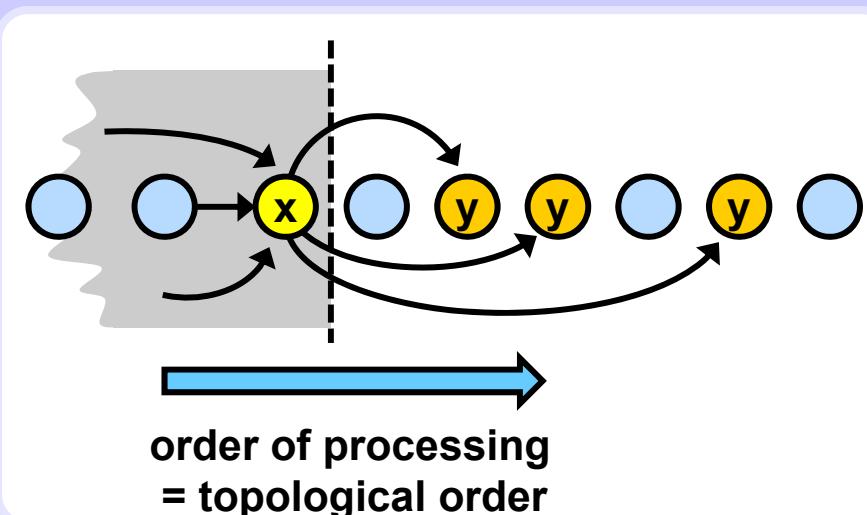
### Varianta I

```
2. for each node (x) in V(G) {
    for each edge e = (y, x) in E(G)
        if (x. dist < y.dist + e.weight) {
            x. dist = y.dist + e.weight
            x.pred = y;
        }
    }
```



### Varianta II

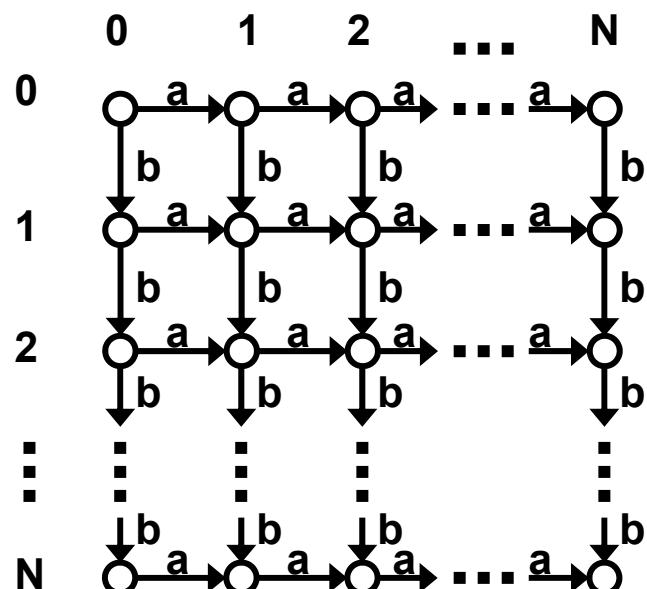
```
2. for each node (x) in V(G) {
    for each edge e = (x, y) in E(G)
        if (y. dist < x.dist + e.weight) {
            y. dist = x.dist + e.weight
            y.pred = x;
        }
    }
```



## Nejdelší cesty v DAG

Problém rekonstrukce všech optimálních cest  
– může jich být příliš mnoho.

### Ukázka



Každá cesta z kořene do listu je optimální, má cenu  $N \cdot (a+b)$ .

Počet všech těchto cest je  $\binom{2N}{N}$ ,

přičemž  $2^N < \binom{2N}{N} < 4^N$ .

Počet optimálních řešení tedy roste exponenciálně vůči hodnotě  $N$ . Např.

$N$	Počet optimálních řešení
1	2
10	184756
20	137846528820
30	118264581564861424
40	107507208733336176461620

## ALG 11

### Dynamické programování

Nejdelší rostoucí podposloupnost

Optimální pořadí násobení matic

## Nejdelší rostoucí podposloupnost

Z dané posloupnosti vyberte co nejdelší rostoucí podposloupnost.

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

Řešení: 4    5    6    7

### Jiné možné varianty

Vlastnosti hledané podposloupnosti:

Klesající, nerostoucí, neklesající, aritmetická,  
s omezenou rychlosťí růstu, s váhami prvků, ... atd., ...

[zde neprobírané](#)

### Koncepční přístup I

Převeď na známou úlohu, definuj vhodný DAG podle daných vlastností podposloupnosti, v DAG hledej nejdelší cestu.

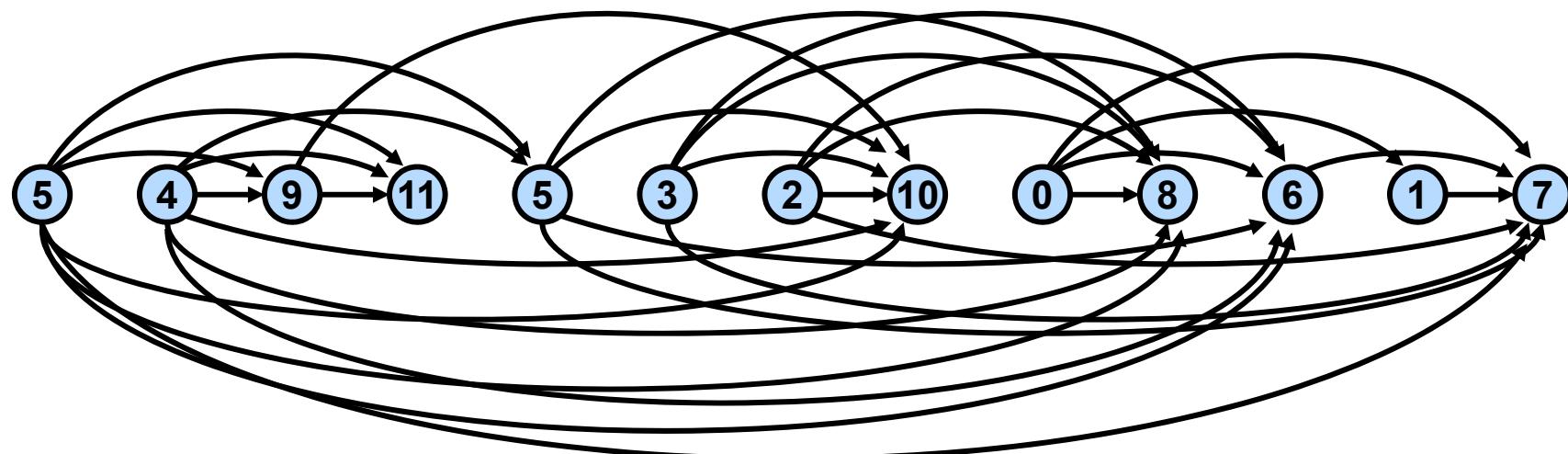
## Nejdelší rostoucí podposloupnost

Koncepční přístup I

Transformace na známou úlohu

Prvky posloupnosti budou uzly DAG, který je již topologicky uspořádán, pořadí v posloupnosti = pořadí v top. uspořádání.  
 Hrana  $x \rightarrow y$  existuje právě tehdy,  
 když  $x$  je v posloupnosti dříve než  $y$  a navíc  $x < y$ .

V tomto DAG hledáme nejdelší cestu.

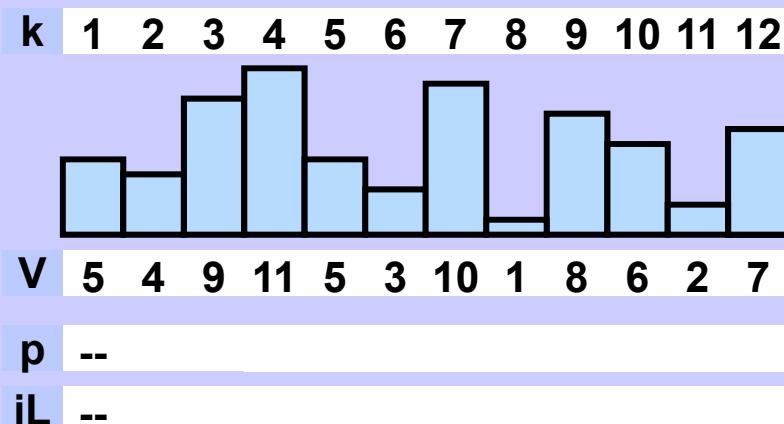


Algoritmus je znám, má složitost  $\Theta(N+M)$ , tedy  $O(N^2)$ .  
 Např. pro rostoucí posloupnost má složitost až  $\Theta(N^2)$ .

## Nejdelší rostoucí podposloupnost

### Koncepční přístup II

Sestav samostatný a potenciálně rychlejší algoritmus řešení:  
 Registrujme optimální podposloupnosti všech možných délek.  
 Postupně metodou DP aktualizujme tyto optimální podposloupnosti.



k .. index prvku  
 V .. hodnota prvku  
 p .. index předchůdce  
 iL .. index posledního prvku  
      v rostoucí podposloupnosti  
      délky d = 1, 2, ..., N.

Pro každý index k:

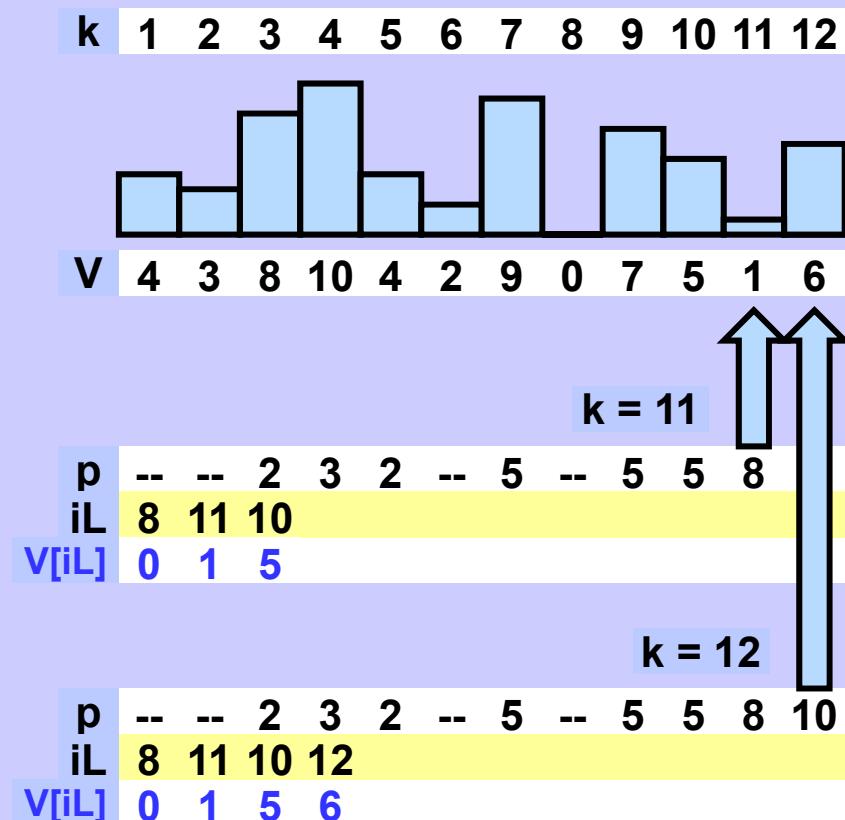
Nechť d je index největšího prvku, pro který platí  $V[iL[d]] < V[k]$ .

Potom  $iL[d+1] := k$ ,  $p[k] = iL[d]$ , pokud d existuje.

Jinak  $iL[1] := k$ ,  $p[k] = \text{null}$ .

$V[iL[d]]$ ,  $d = 1..N$  je neklesající, lze v ní hledat v čase  $O(\log N)$ .

## Nejdelší rostoucí podposloupnost



k .. index prvku  
 V .. hodnota prvku  
 p .. index předchůdce  
 iL .. index posledního prvku  
 v rostoucí podposloupnosti  
 délky  $d = 1, 2, \dots, N$ .

Pro každý index  $k$ :

Nechť  $d$  je index max. prvku, pro který platí

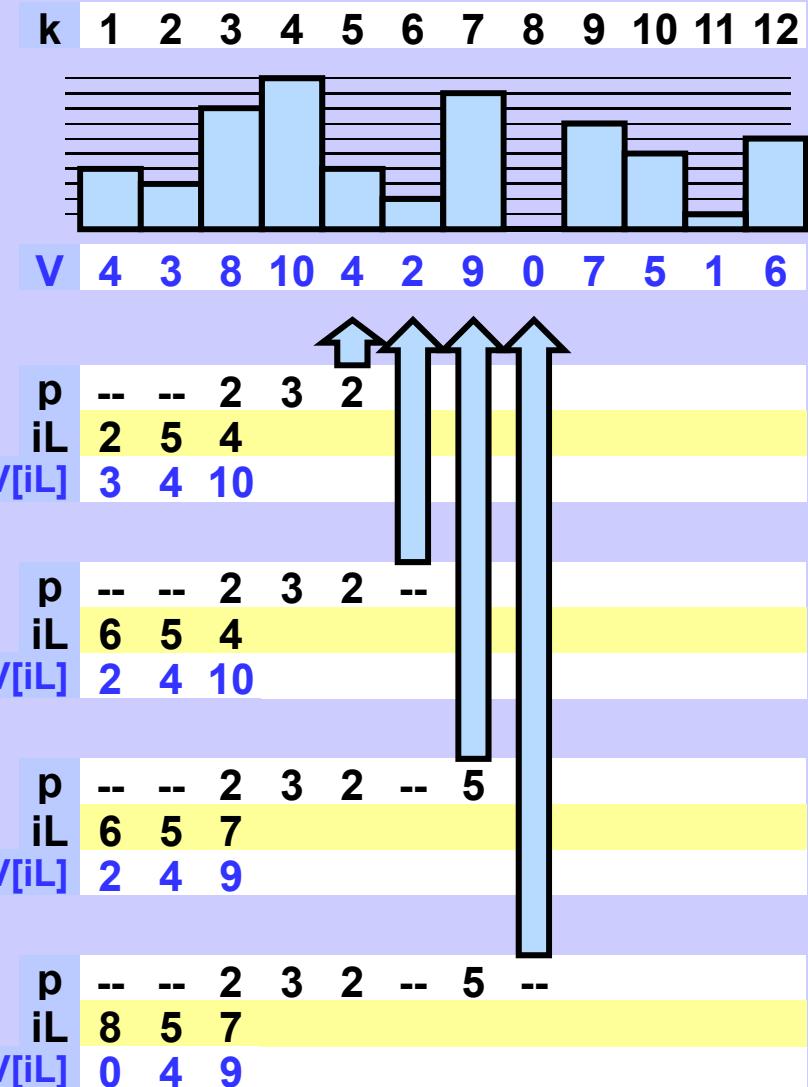
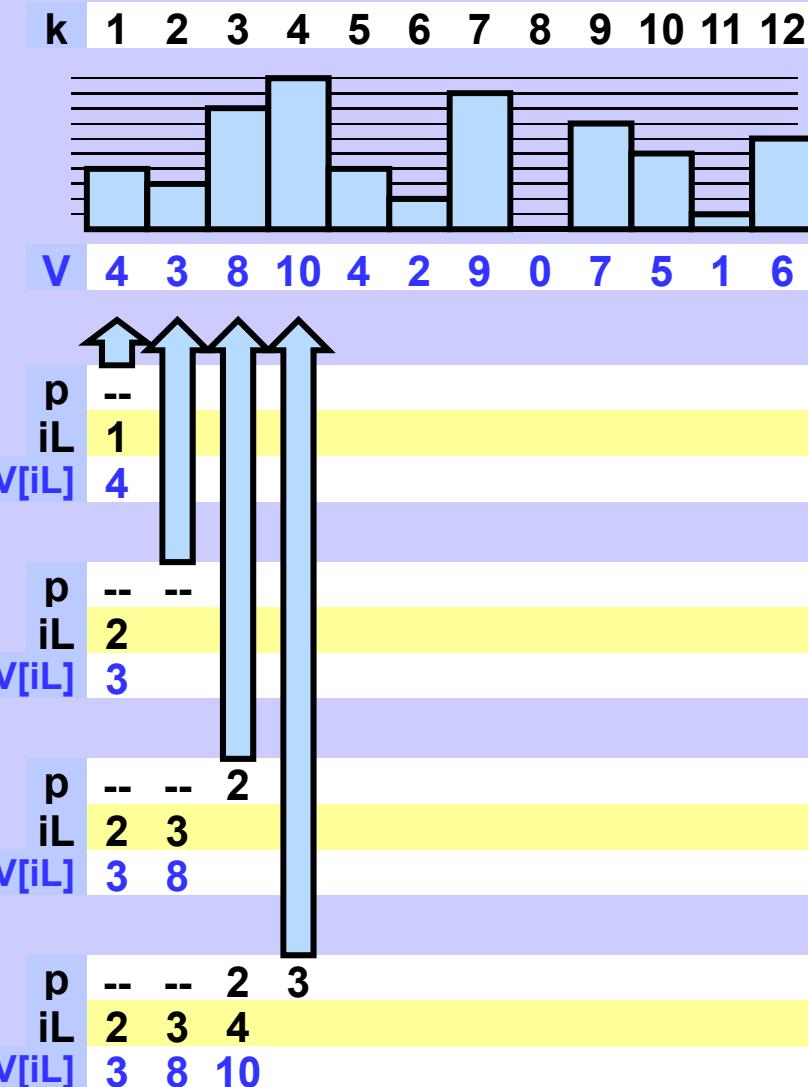
$$V[iL[d]] < V[k].$$

Potom  $iL[d+1] := k$ ,  $p[k] = iL[d]$ , pokud  $d$  existuje.

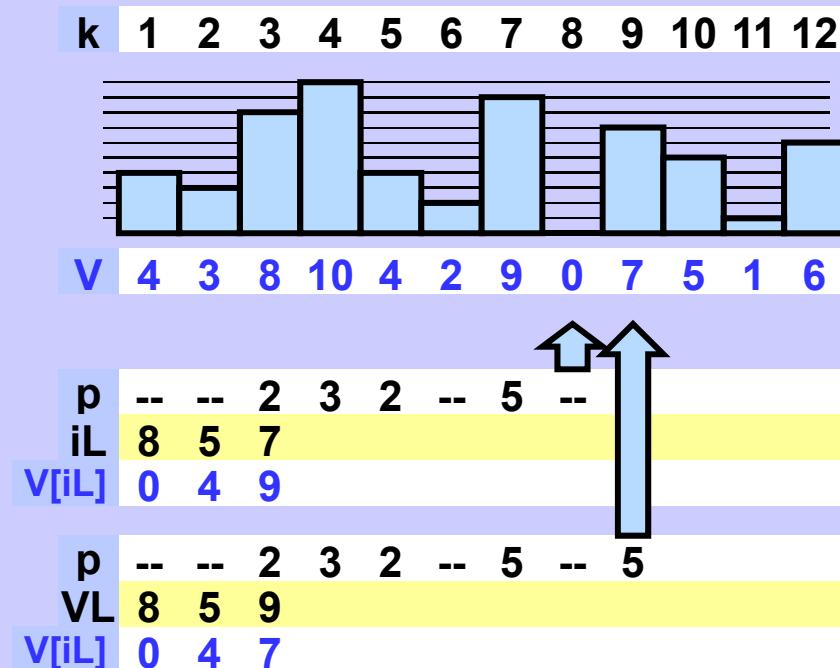
Jinak  $iL[1] := k$ ,  $p[k] = \text{null}$ .

Pro každé  $k$  je nalezneme  $d$  v čase  $O(\log N)$  půlením intervalu.  
 Aktualizace  $iL$  a  $p$  proběhne v konstantním čase.  
 Celkem je složitost  $O(N \log (N))$ .

## Nejdelší rostoucí podposloupnost

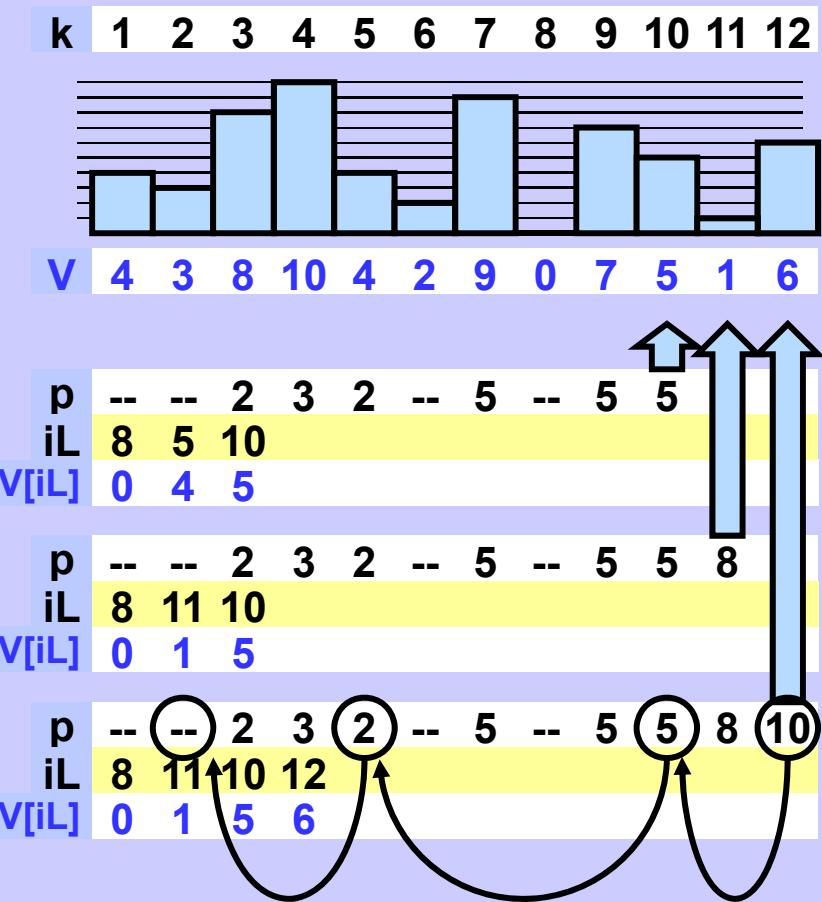


## Nejdelší rostoucí podposloupnost



### Rekonstrukce optimální cesty

Poslední definovaný prvek v iL je indexem posledního prvku jedné z optimálních podposloupností celé posloupnosti.  
Pole p určuje pomocí předchůdců tuto podposloupnost.



## Optimální pořadí násobení matic

### Instance úlohy

Máme spočítat co nejfektivněji součin reálných matic

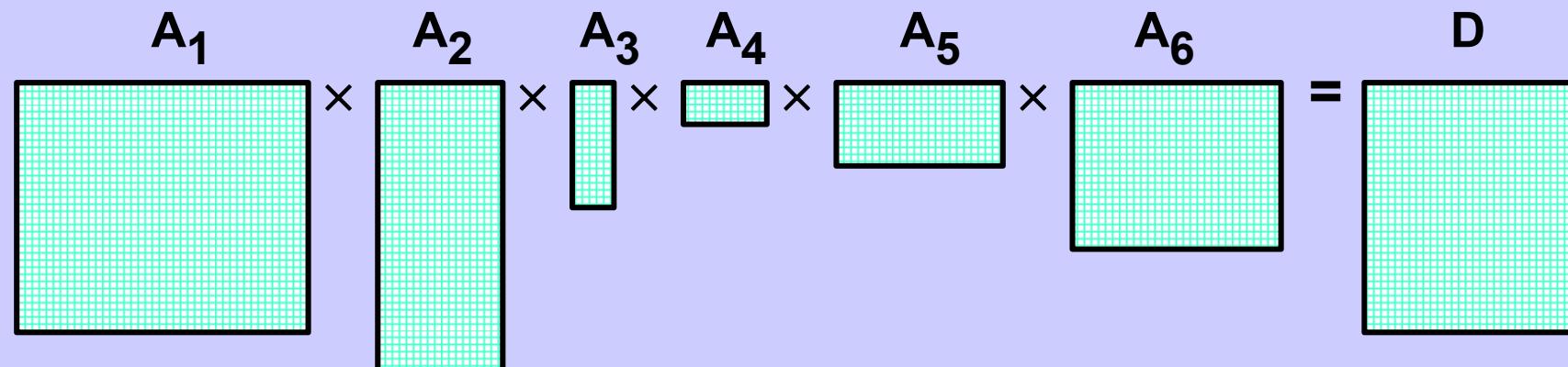
$$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6,$$

kde rozměry jednotlivých matic jsou po řadě

$30 \times 35, 35 \times 15, 15 \times 5, 5 \times 10, 10 \times 20, 20 \times 25$ .

(Výsledná matice D má rozměr  $30 \times 20$ ).

### Grafická podoba (dimenze matic ve správném poměru)



Instance převzata z [CLRS], kap. 15.

## Optimální pořadí násobení matic

### Počet operací v násobení dvou matic

$$a \begin{pmatrix} & & & \\ & \text{green} & & \\ & & & \\ & & & \\ & & & \\ \dots & & & \\ & & & \\ b & & & \end{pmatrix} \times b \begin{pmatrix} & & & \\ & \text{green} & & \\ & & & \\ & & & \\ & & & \\ \dots & & & \\ & & & \\ c & & & \end{pmatrix} = a \begin{pmatrix} & & & \\ & \text{green} & & \\ & & & \\ & & & \\ & & & \\ \dots & & & \\ & & & \\ c & & & \end{pmatrix}$$

$$1 \begin{pmatrix} & & & \\ & \text{green} & & \\ & & & \\ & & & \\ & & & \\ b & & & \end{pmatrix} \times b \begin{pmatrix} & & \\ & \text{green} & \\ & & \\ & & \\ & & \\ 1 & & \end{pmatrix} = \boxed{\text{green}}$$

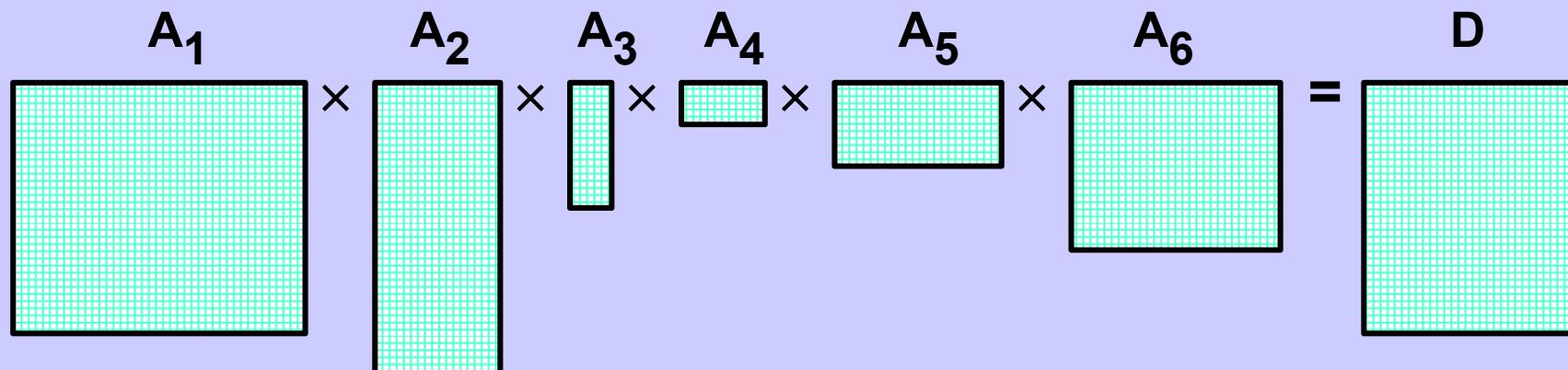
$b$  operací násobení pro výpočet jednoho prvku výsledné matice

$a * c$  prvků ve výsledné matici

Vynásobení dvou matic o rozměrech  $a \times b$  a  $b \times c$  vyžaduje celkem  $a * b * c$  operací násobení dvou prvků (čísel).

Sčítání zde neuvažujeme, lze pro něj vyvinout analogický postup.

## Optimální pořadí násobení matic

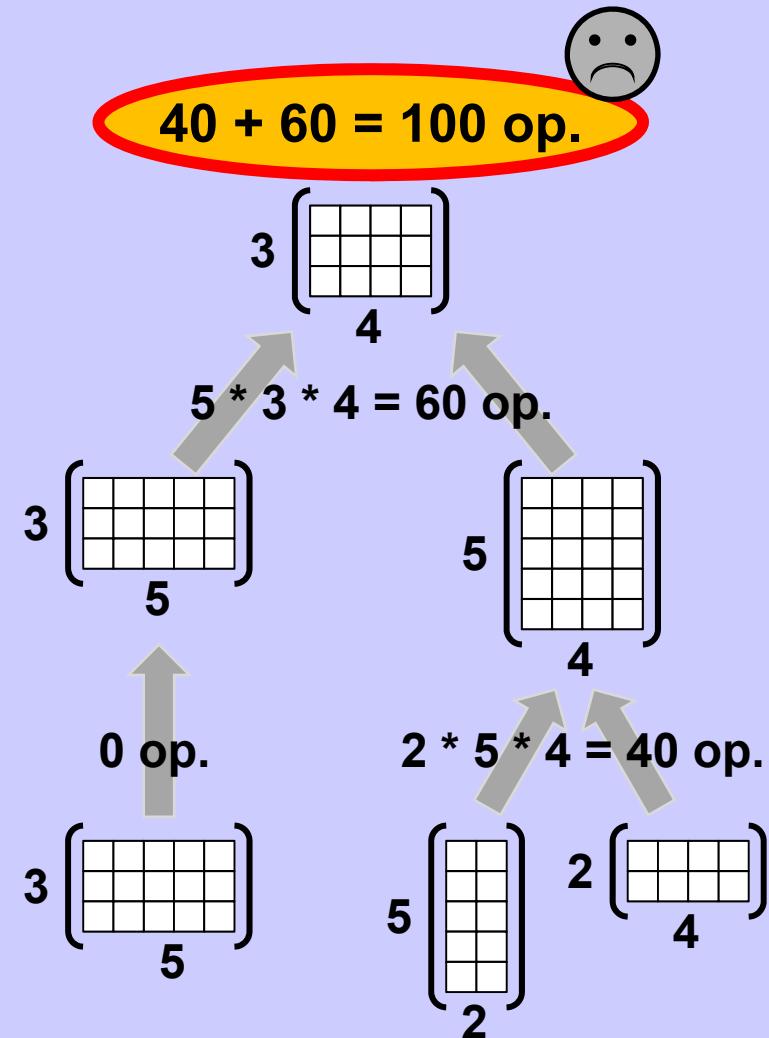
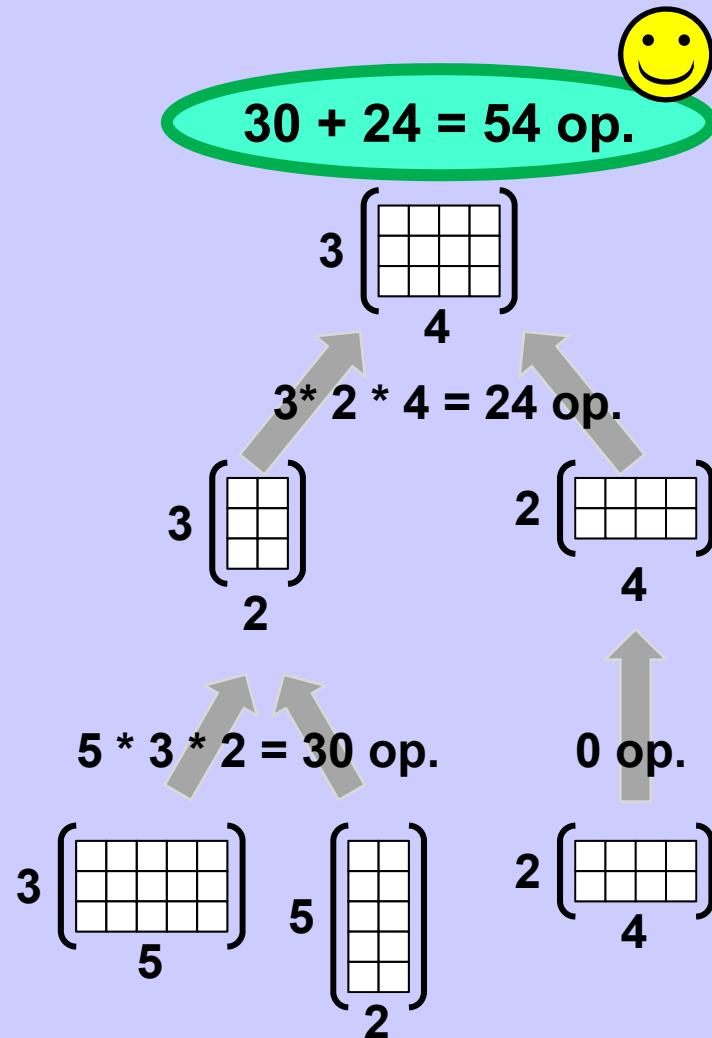


Sledujeme jen počet operací součinu dvou reálných čísel.  
Uvažujeme různé možnosti uzávorkování a tím i pořadí výpočtu.

metoda	Výraz	Počet operací
zleva doprava	$((((A_1 \times A_2) \times A_3) \times A_4) \times A_5) \times A_6$	43 500
zprava doleva	$A_1 \times (A_2 \times (A_3 \times (A_4 \times (A_5 \times A_6))))$	47 500
nejhorší	$A_1 \times ((A_2 \times ((A_3 \times A_4) \times A_5)) \times A_6)$	58 000
nejlepší	$(A_1 \times (A_2 \times A_3)) \times ((A_4 \times A_5) \times A_6)$	15 125

## Optimální pořadí násobení matic

### Příklad násobení více matic



## Optimální pořadí násobení matic

$$A_1 = 3 \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \\ 5$$

$$A_2 = 5 \begin{pmatrix} & & \\ & & \\ & & \\ & & \end{pmatrix} \\ 2$$

$$A_3 = 2 \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \\ 4$$

**Součin**  $(A_1 \times A_2) \times A_3$  vyžaduje 54 operace násobení .

**Součin**  $A_1 \times (A_2 \times A_3)$  vyžaduje 100 operací násobení.

Evidentně, na způsobu uzávorkování záleží .

### Catalanova čísla $C_N$

**Součin**  $A_1 \times A_2 \times A_3 \times \dots \times A_N$  lze uzávorkovat

$C_N = \text{Comb}(2N, N) / (N+1)$  způsoby.

$C_1, C_2, \dots, C_7 = 1, 1, 2, 5, 14, 42, 132$ .  $C_N > 2^N$  pro  $N > 7$ .

V obecném případě by mělo vyzkoušení všech uzávorkování exponenciální složitost.

## Optimální pořadí násobení matic

### Ilustrace

14 různých způsobů uzávorkování součinu 5 činitelů

$$\begin{aligned} & A_1 \times (A_2 \times (A_3 \times (A_4 \times A_5))) \\ & A_1 \times (A_2 \times ((A_3 \times A_4) \times A_5)) \\ & A_1 \times ((A_2 \times A_3) \times (A_4 \times A_5)) \\ & A_1 \times ((A_2 \times (A_3 \times A_4)) \times A_5) \\ & A_1 \times (((A_2 \times A_3) \times A_4) \times A_5) \\ & (A_1 \times A_2) \times (A_3 \times (A_4 \times A_5)) \\ & (A_1 \times A_2) \times ((A_3 \times A_4) \times A_5) \\ & (A_1 \times (A_2 \times A_3)) \times (A_4 \times A_5) \\ & ((A_1 \times A_2) \times A_3) \times (A_4 \times A_5) \\ & (A_1 \times (A_2 \times (A_3 \times A_4))) \times A_5 \\ & (A_1 \times ((A_2 \times A_3) \times A_4)) \times A_5 \\ & ((A_1 \times A_2) \times (A_3 \times A_4)) \times A_5 \\ & ((A_1 \times (A_2 \times A_3)) \times A_4) \times A_5 \\ & (((A_1 \times A_2) \times A_3) \times A_4) \times A_5 \end{aligned}$$

## Optimální pořadí násobení matic

$$\begin{aligned}
 & A_1 \times (A_2 \times A_3 \times A_4 \dots \times A_{N-1} \times A_N) \\
 & (A_1 \times A_2) \times (A_3 \times A_4 \dots \times A_{N-1} \times A_N) \\
 & (A_1 \times A_2 \times A_3) \times (A_4 \dots \times A_{N-1} \times A_N) \\
 & (A_1 \times A_2 \times A_3 \times A_4) \times (\dots \times A_{N-1} \times A_N) \\
 & \quad \vdots \\
 & (A_1 \times A_2 \times A_3 \times A_4 \times \dots) \times (A_{N-1} \times A_N) \\
 & (A_1 \times A_2 \times A_3 \times A_4 \times \dots \times A_{N-1}) \times A_N
 \end{aligned}$$

$N - 1$  možných míst,  
v nichž výraz  
rozdělíme  
a provedeme  
poslední násobení

Předpokládejme, že máme předpočítáno optimální uzávorkování pro každý modrý úsek celkového výrazu.

## Optimální pořadí násobení matic

$$A_1 \times (A_2 \times A_3 \times A_4 \dots \times A_{N-1} \times A_N) = B[1,1] \times B[2,N]$$

$$(A_1 \times A_2) \times (A_3 \times A_4 \dots \times A_{N-1} \times A_N) = B[1,2] \times B[3,N]$$

$$(A_1 \times A_2 \times A_3) \times (A_4 \dots \times A_{N-1} \times A_N) = B[1,3] \times B[4,N]$$

$$(A_1 \times A_2 \times A_3 \times A_4) \times (\dots \times A_{N-1} \times A_N) = B[1,4] \times B[5,N]$$

⋮

⋮

$$(A_1 \times A_2 \times A_3 \times A_4 \times \dots) \times (A_{N-1} \times A_N) = B[1,N-2] \times B[N-1,N]$$

$$(A_1 \times A_2 \times A_3 \times A_4 \times \dots \times A_{N-1}) \times A_N = B[1,N-1] \times B[N,N]$$

Matice  $B[i, j]$  představuje výsledek vynásobení odpovídajícího úseku.

Nechť  $r(X)$  resp.  $s(X)$  představují počet řádků resp sloupců matice  $X$ .  
 Podle pravidel násobení matic platí  
 $r(B[i, j]) = r(A_i)$ ,  $s(B[i, j]) = s(A_j)$ , pro  $1 \leq i \leq j \leq N$ .

## Optimální pořadí násobení matic

Nechť  $MO[i, j]$  představuje minimální počet operací potřebných k výpočtu matice  $B[i, j]$ , tj. minimální počet operací potřebných k výpočtu matice  $A_i \times A_{i+1} \times \dots \times A_{j-1} \times A_j$ .

$B[1,1]$	$\times$	$B[2,N]$	$MO[1,1] + r(A_1)*s(A_1)*s(A_N) + MO[2, N]$
$B[1,2]$	$\times$	$B[3,N]$	$MO[1,2] + r(A_1)*s(A_2)*s(A_N) + MO[3, N]$
$B[1,3]$	$\times$	$B[4,N]$	$MO[1,3] + r(A_1)*s(A_3)*s(A_N) + MO[4, N]$
...			
$B[1,N-2]$	$\times$	$B[N-1,N]$	$MO[1,N-2] + r(A_1)*s(A_{N-2})*s(A_N) + MO[N-1, N]$
$B[1,N-1]$	$\times$	$B[N,N]$	$MO[1,N-1] + \underbrace{r(A_1)*s(A_{N-1})*s(A_N)}_{\text{operací při násobení}} + \underbrace{MO[N, N]}_{\text{operací v pravém úseku}}$
			$B[1,.] \times B[.,N]$

Celkem dostáváme  $MO[1,N]$ :

$$MO[1,N] = \min \{ MO[1,k] + r(A_1)*s(A_k)*s(A_N) + MO[k+1, N] \mid k = 1..N-1 \}$$

## Optimální pořadí násobení matic

$$MO[1, N] = \min \{ MO[1, k] + r(A_1) * s(A_k) * s(A_N) + MO[k+1, N] \mid k = 1..N-1 \}$$

Za předpokladu znalosti  $MO[i, j]$  pro úseky kratší než  $[1, N]$ , lze řešení celé úlohy, tj. hodnotu  $MO[1, N]$ , spočít v čase  $\Theta(N)$ . (\*)

## Rekurentní využití řešení menších podúloh

Identické úvahy, jaké jsme provedli pro celý výraz

$$A_1 \times A_2 \times A_3 \times \dots \times A_N,$$

provedeme rovněž pro každý jeho souvislý úsek

$$\dots A_L \times A_{L+1} \times \dots \times A_{R-1} \times A_R \dots, \quad 1 \leq L \leq R \leq N.$$

Počet těchto souvislých úseků je stejný jako počet dvojic indexů  $(L, R)$ , kde  $1 \leq L \leq R \leq N$ . Ten je roven  $Comb(N, 2) \in \Theta(N^2)$ .

Podúlohu na úseku  $(L, R)$  lze spočít podle (\*) v čase  $O(N)$ , celou úlohu tak lze vyřešit v čase  $O(N^3)$ .

## Optimální pořadí násobení matic

\*

$$MO[L,R] = \min \{ MO[L,k] + r(A_L) * s(A_k) * s(A_R) + MO[k+1,R] \mid k = L..R-1 \}$$

Hodnoty  $MO[L,R]$  ukládáme do 2D pole na pozici s indexy  $[L][R]$ .

Při výpočtu  $MO[L,R]$  podle (\*) používáme vesměs hodnoty  $MO[x,y]$ , kde rozdíl  $y - x$  (odpovídající délce podvýrazu) je menší než rozdíl  $R - L$ .

Tabulku DP proto vyplňujeme v pořadí rostoucích rozdílů  $R - L$ .

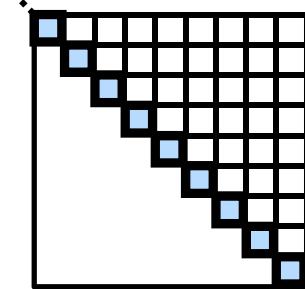
0. Vyplníme prvky s indexy  $[L][R]$ , kde  $R-L = 0$ , to je hlavní diagonála.
  1. Vyplníme prvky s indexy  $[L][R]$ , kde  $R-L = 1$ , to je diagonála těsně nad hlavní diagonálou.
  2. Vyplníme prvky s indexy  $[L][R]$ , kde  $R-L = 2$ , to je diagonála těsně nad předchozí diagonálou.
- ...

- N-1. Vyplníme prvek s indexem  $[L][R]$ , kde  $R-L = N-1$ , to je pravý horní roh tabulky.

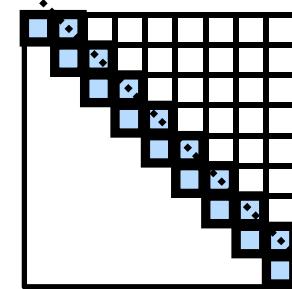
## Optimální pořadí násobení matic

### Schéma postupu výpočtu

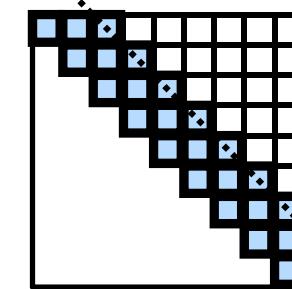
$$R - L = 0$$



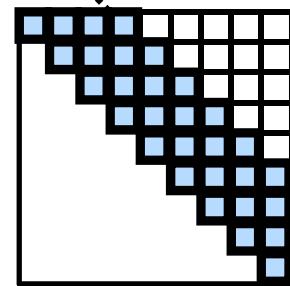
$$R - L = 1$$



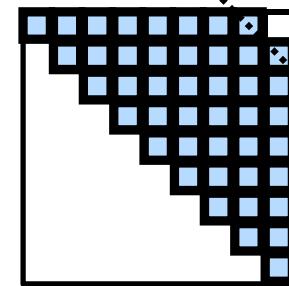
$$R - L = 2$$



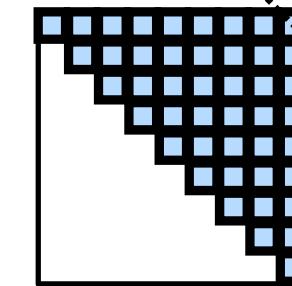
$$R - L = 3$$



$$R - L = N-2$$



$$R - L = N-1$$

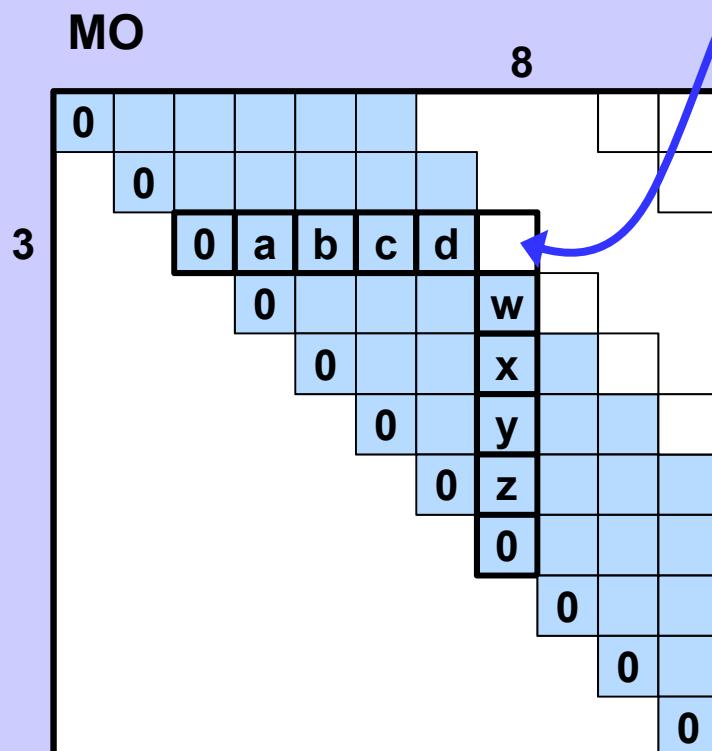


Stop

## Optimální pořadí násobení matic

$$MO[L,R] = \min \{ MO[L,k] + r(A_L) * s(A_k) * s(A_R) + MO[k+1,R] \mid k = L..R-1 \}$$

Ukázka postupu výpočtu



$$MO[3,8] = \min \{ MO[3,3] + r(A_3)*s(A_3)*s(A_8) + MO[4,8], \\ MO[3,4] + r(A_3)*s(A_4)*s(A_8) + MO[5,8], \\ MO[3,5] + r(A_3)*s(A_5)*s(A_8) + MO[6,8], \\ MO[3,6] + r(A_3)*s(A_6)*s(A_8) + MO[7,8], \\ MO[3,7] + r(A_3)*s(A_7)*s(A_8) + MO[8,8] \}$$

Označme  $P[L, R] := r(A_L)*s(A_R)$ . Potom

$$MO[3,8] = \min \{ 0 + s(A_3)*P[3,8] + w, \\ a + s(A_4)*P[3,8] + x, \\ b + s(A_5)*P[3,8] + y, \\ c + s(A_6)*P[3,8] + z, \\ d + s(A_7)*P[3,8] + 0 \}.$$

## Optimální pořadí násobení matic

### Instance úlohy

$$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 = D$$

$30 \times 35$        $35 \times 15$        $15 \times 5$        $5 \times 10$        $10 \times 20$        $20 \times 25$        $30 \times 25$

MO	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2	0	0	2625	4375	7125	10500
3	0	0	0	750	2500	5375
4	0	0	0	0	1000	3500
5	0	0	0	0	0	5000
6	0	0	0	0	0	0

optimum

## Optimální pořadí násobení matic

### Rekonstrukce uzávorkování

\*

$$MO[L,R] = \min \{ MO[L,k] + r(A_L) * s(A_k) * s(A_R) + MO[k+1,R] \mid k = L..R-1 \}$$

Při určení  $MO[L,R]$  do rekonstrukční tabulky RT stejné velikosti jako MO zaneseme na pozici  $[L][R]$  hodnotu  $k$ , v níž minimum (\*) nastalo.

Hodnota  $k$  určuje optimální rozdělení výrazu

$$(A_L \times A_{L+1} \times \dots \times A_R)$$

na dva menší optimálně uzávorkované výrazy

$$(A_L \times A_{L+1} \times \dots \times A_k) \times (A_{k+1} \times A_{k+2} \times \dots \times A_R)$$

Hodnota  $RT[1, N]$  určuje optimální rozdělení celého výrazu

$$A_1 \times A_2 \times \dots \times A_N$$

na první dva menší optimálně uzávorkované výrazy

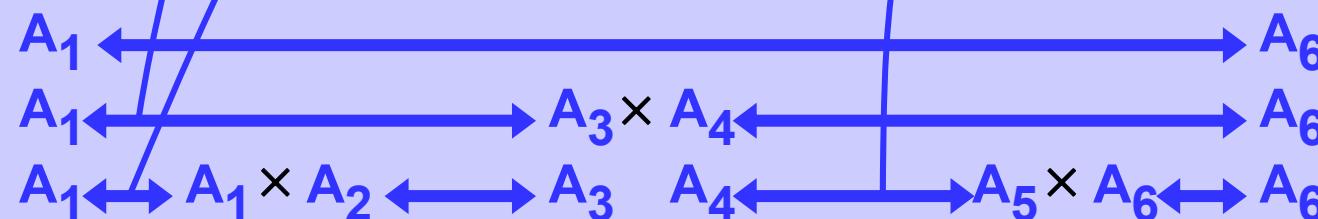
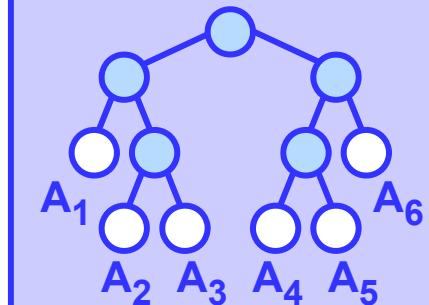
$$(A_1 \times A_2 \times \dots \times A_k) \times (A_{k+1} \times A_{k+2} \times \dots \times A_N).$$

Dále rekonstrukce optimálního uzávorkování pokračuje rekursivně analogicky pro výraz  $(A_1 \times A_2 \times \dots \times A_k)$  a pro výraz  $(A_{k+1} \times A_{k+2} \times \dots \times A_N)$  a dále pro jejich podvýrazy atd.

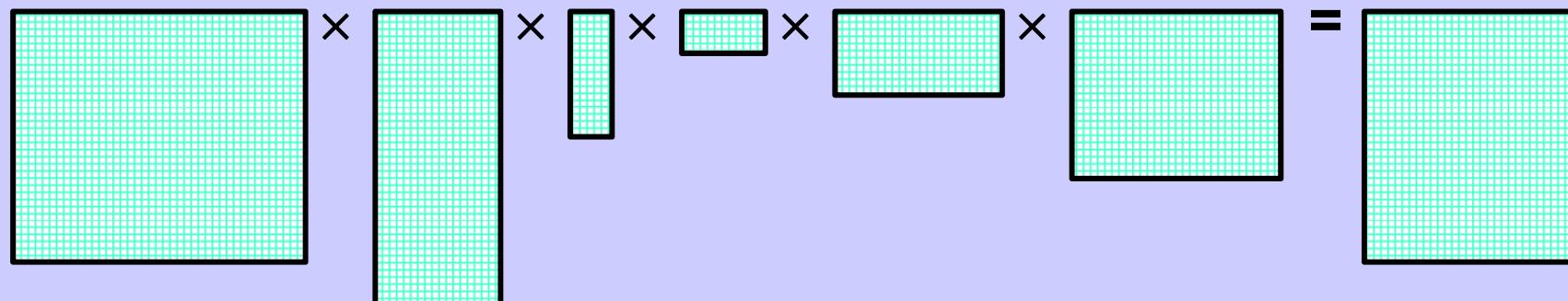
## Optimální pořadí násobení matic

RT

	1	2	3	4	5	6
1	0	1	1	3	3	3
2	0	0	2	3	3	3
3	0	0	0	3	3	3
4	0	0	0	0	4	5
5	0	0	0	0	0	5
6	0	0	0	0	0	0



$$(A_1 \times (A_2 \times A_3)) \times ((A_4 \times A_5) \times A_6) = D$$



## Optimální pořadí násobení matic

### Odvození asymptotické složitosti

index řádku	Řádkové součty	Počet buněk, z nichž je počítán obsah dané buňky v DP tabulce, je úměrný složitosti výpočtu obsahu této buňky.				
$k = N-1$	$1/2 * (N-1) * N$	1	2	3	$N-3$	$N-2$
$k = N-2$	$1/2 * (N-2) * (N-1)$		1	2	$N-4$	$N-3$
$k = N-3$	$1/2 * (N-3) * (N-2)$			1	$N-5$	$N-4$
$k = k$	$1/2 * k * (k+1)$				1	$N-k-2$
$k = 3$	$1/2 * 3 * 4$					$N-k-1$
$k = 2$	$1/2 * 2 * 3$					$N-k$
$k = 1$	$1/2 * 1 * 2$					
$\dots$						
<b>Celkový součet</b>		1	$N-k-2$	$N-k-1$	$N-k$	
			1	2	3	
				1	2	
					1	

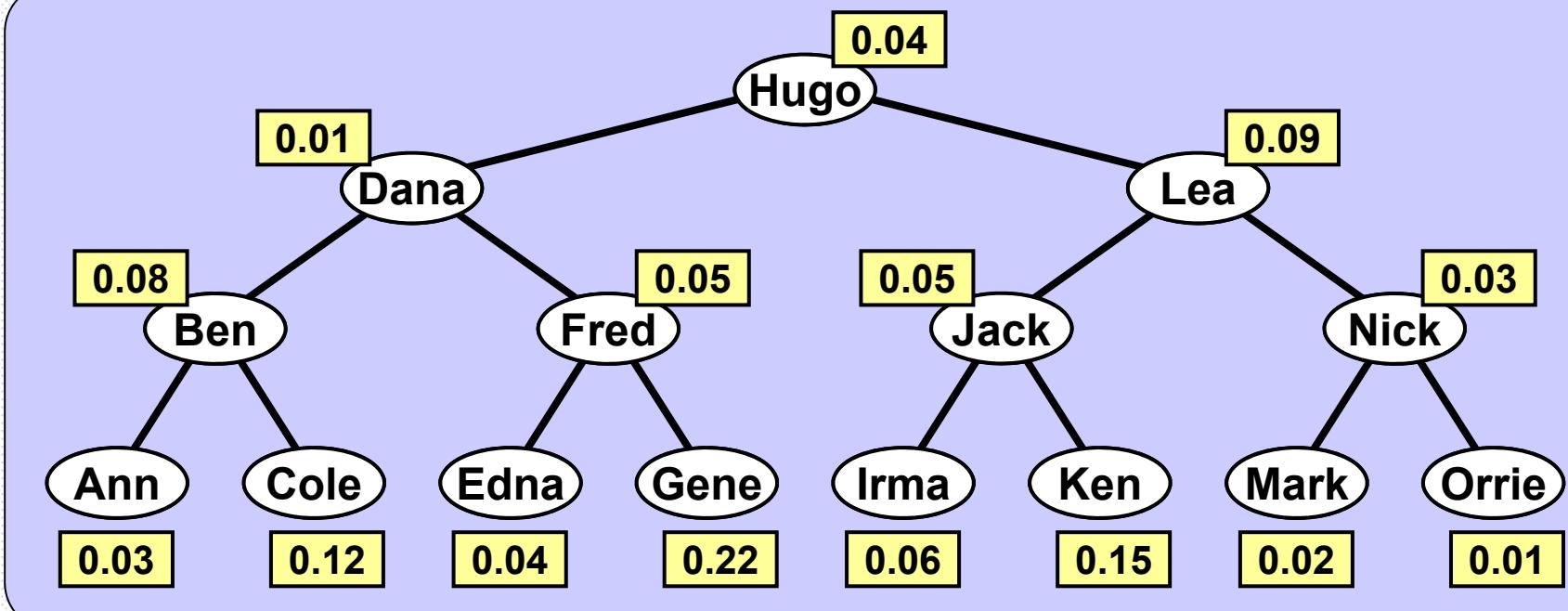
$$\begin{aligned}
 \frac{1}{2} * \sum_{k=1}^{N-1} k * (k+1) &= \frac{1}{2} * \sum_{k=1}^{N-1} k^2 + \frac{1}{2} * \sum_{k=1}^{N-1} k \\
 &= \frac{1}{2} * (N-1) * N * (2N-1)/6 + \frac{1}{2} * (N-1) * N/2 \in \Theta(N^3)
 \end{aligned}$$

# Dynamické programování

## Optimální binární vyhledávací strom

## Optimální binární vyhledávací strom

Vyvážený, ale ne optimální

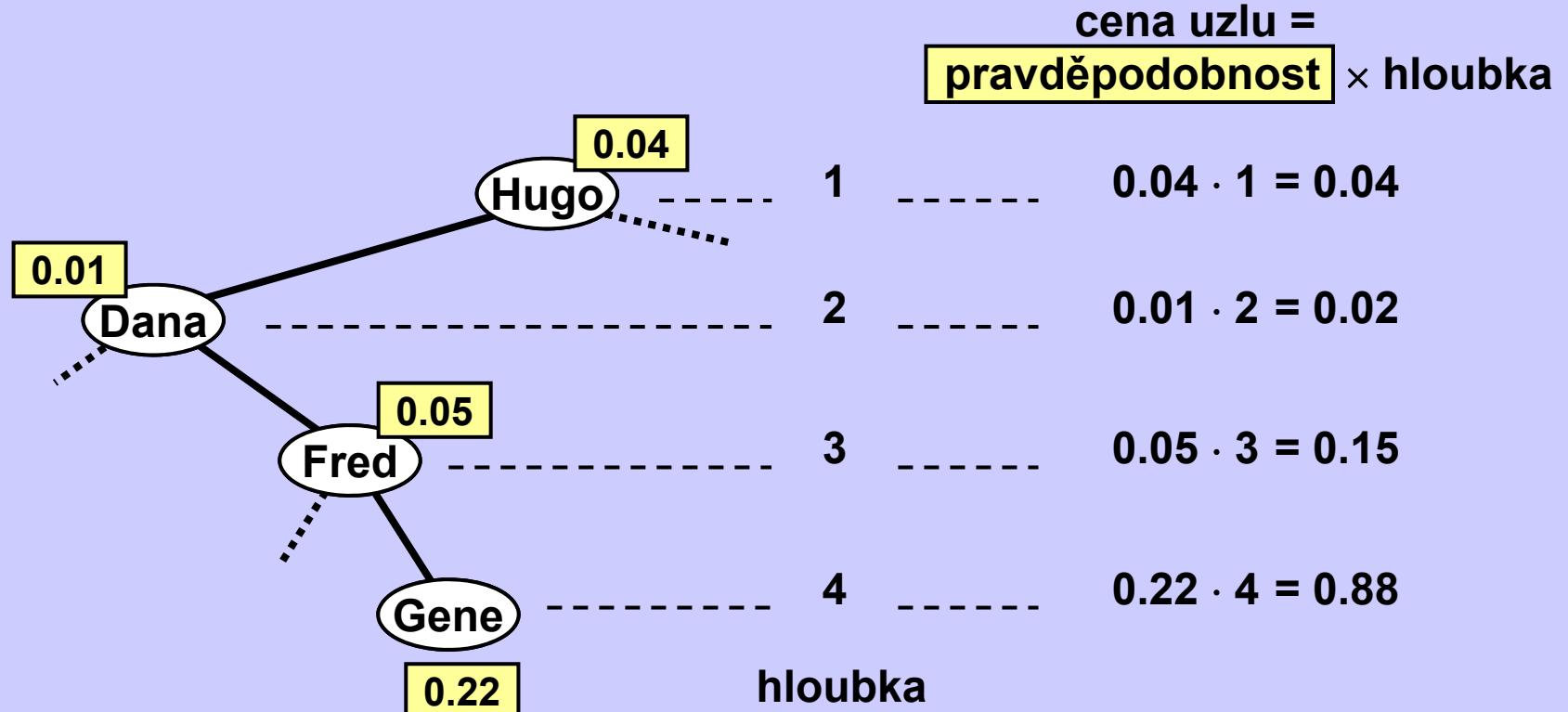


Pravděpodobnost dotazu

Klíč

## Optimální binární vyhledávací strom

### Cena jednotlivých uzlů v BVS



cena uzlu = průměrný počet testů na nalezení uzlu  
při jednom dotazu (Find)

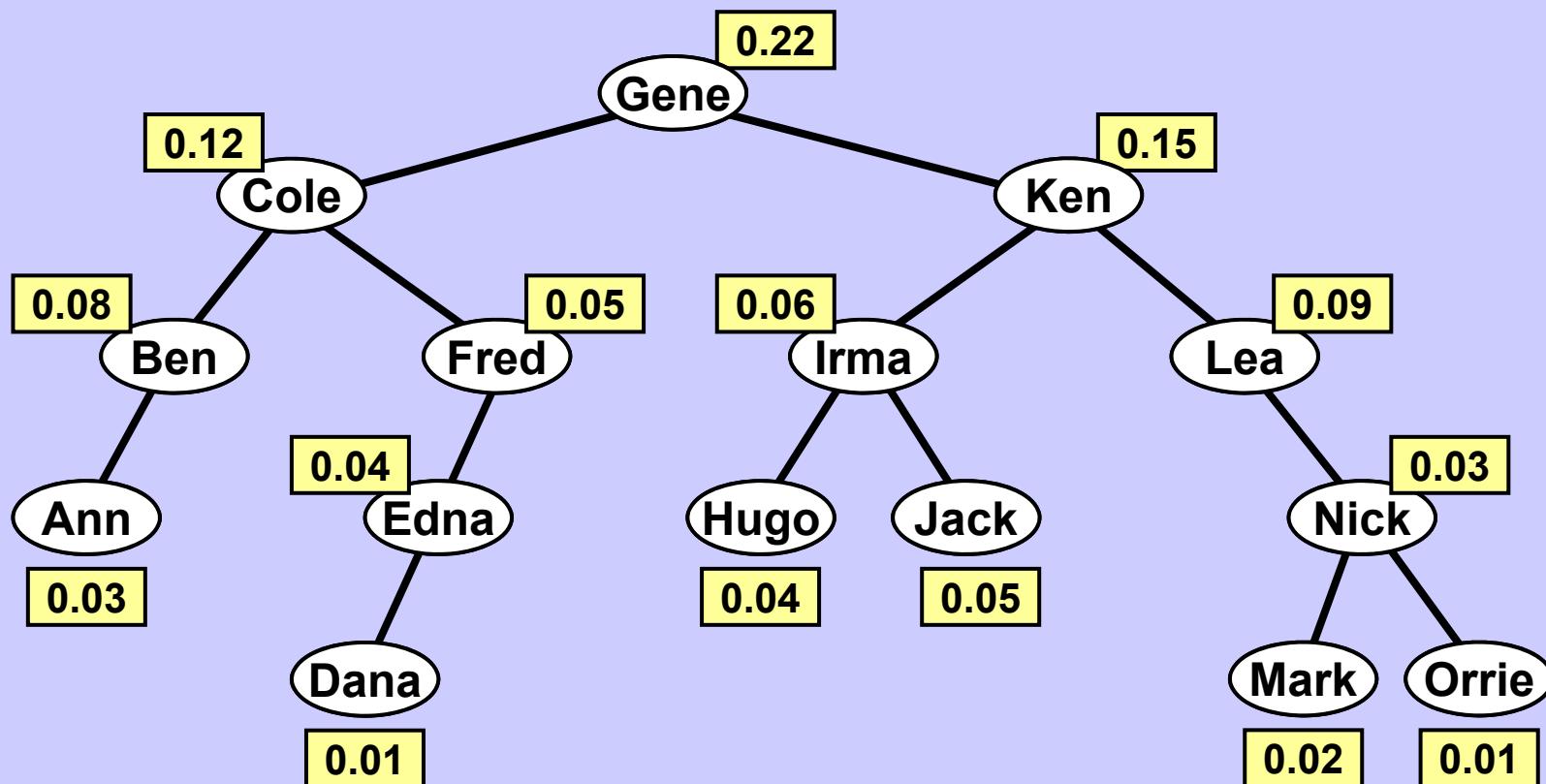
## Cena výváženého stromu

klíč	pravděp. $p_k$	hloubka $d_k$	$p_k \cdot d_k$
Ann	0.03	4	$0.03 \cdot 4 = 0.12$
Ben	0.08	3	$0.08 \cdot 3 = 0.24$
Cole	0.12	4	$0.12 \cdot 4 = 0.48$
Dana	0.01	2	$0.01 \cdot 2 = 0.02$
Edna	0.04	4	$0.04 \cdot 4 = 0.16$
Fred	0.05	3	$0.05 \cdot 3 = 0.15$
Gene	0.22	4	$0.22 \cdot 4 = 0.88$
Hugo	0.04	1	$0.04 \cdot 1 = 0.04$
Irma	0.06	4	$0.06 \cdot 4 = 0.24$
Jack	0.05	3	$0.05 \cdot 3 = 0.15$
Ken	0.15	4	$0.15 \cdot 4 = 0.60$
Lea	0.09	2	$0.09 \cdot 2 = 0.18$
Mark	0.02	4	$0.02 \cdot 4 = 0.08$
Nick	0.03	3	$0.03 \cdot 3 = 0.09$
Orrie	0.01	4	$0.01 \cdot 4 = 0.04$
Cena celkem: 3.47			

Cena celkem = prům. poč. testů na jednu operaci Find.

## Optimální BVS

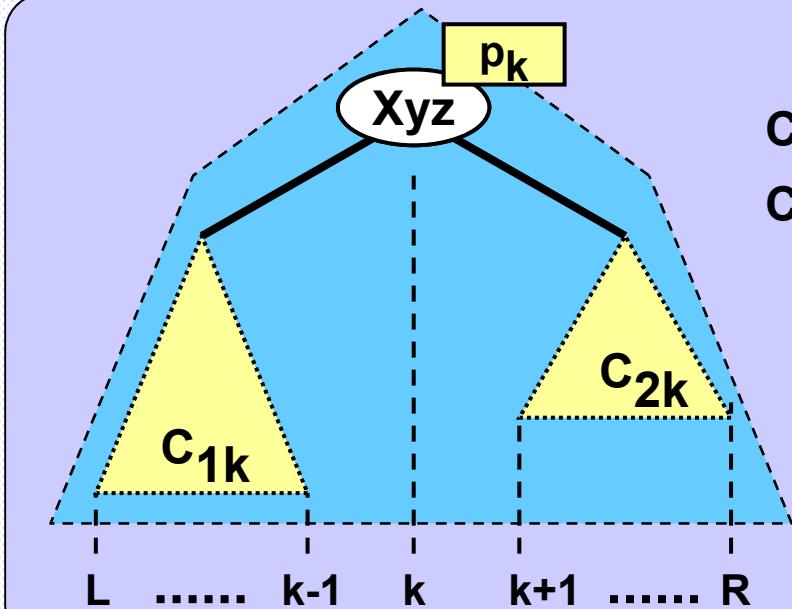
Struktura optimálního BVS s danými pravděpodobnostmi



## Cena optimálního BVS

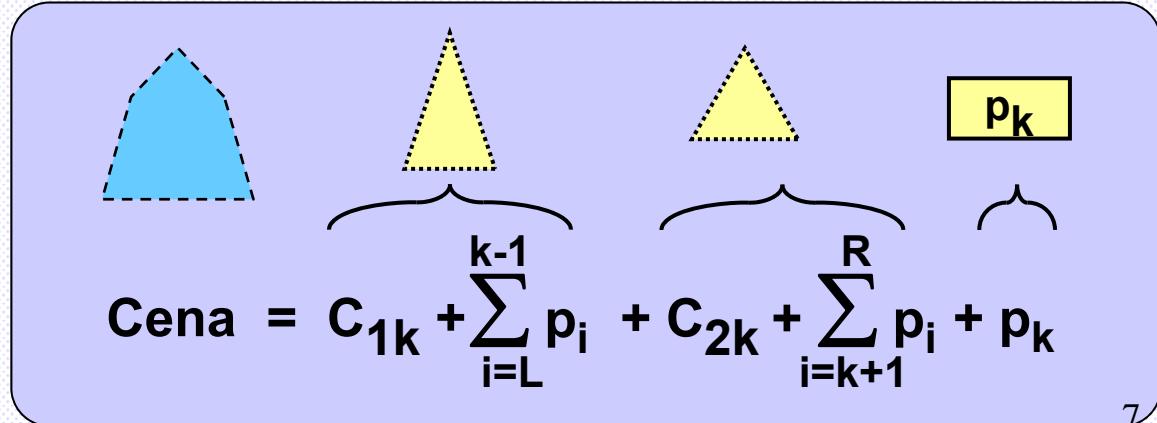
klíč	pravděp. $p_k$	hloubka $d_k$	$p_k \cdot d_k$
Ann	0.03	4	$0.03 \cdot 4 = 0.12$
Ben	0.08	3	$0.08 \cdot 3 = 0.24$
Cole	0.12	2	$0.12 \cdot 2 = 0.24$
Dana	0.01	5	$0.01 \cdot 5 = 0.05$
Edna	0.04	4	$0.04 \cdot 4 = 0.16$
Fred	0.05	3	$0.05 \cdot 3 = 0.15$
Gene	0.22	1	$0.22 \cdot 1 = 0.22$
Hugo	0.04	4	$0.04 \cdot 4 = 0.16$
Irma	0.06	3	$0.06 \cdot 3 = 0.18$
Jack	0.05	4	$0.05 \cdot 4 = 0.20$
Ken	0.15	2	$0.15 \cdot 2 = 0.30$
Lea	0.09	3	$0.09 \cdot 3 = 0.27$
Mark	0.02	5	$0.02 \cdot 5 = 0.10$
Nick	0.03	4	$0.03 \cdot 4 = 0.12$
Orrie	0.01	5	$0.01 \cdot 5 = 0.05$
<b>Cena celkem</b>			<b>2.56</b>
<b>Zrychlení</b>			<b>3.47 : 2.56 = 1 : 0.74</b>

## Výpočet ceny optimálního BVS



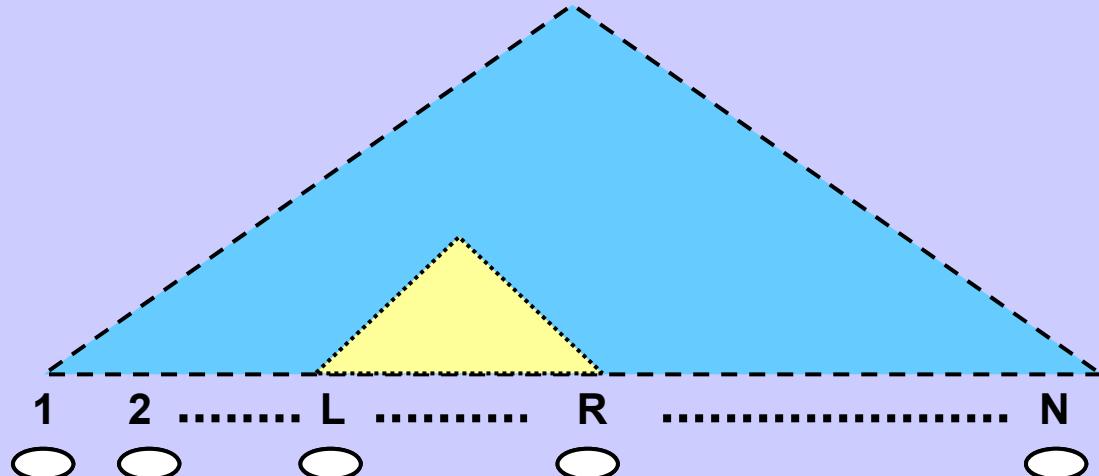
$C_{1k}$  ..... cena levého podstromu uzlu k  
 $C_{2k}$  ..... Cena pravého podstromu uzlu k

Rekurzivní myšlenka



## Výpočet ceny optimálního BVS

Malé  
optimální  
podstromy



Nad prvky s indexy od L do R  
lze jistě vytvořit jeden optimální podstrom.

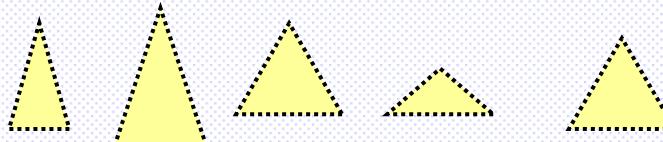
Velikost stromu  
= poč. uzelů  
=  $L-R+1$

Máme	$N$	optimalních podstromů velikosti	1
	$N-1$		2
	$N-2$		3
	⋮		⋮
	1	podstrom	$N$

Celkem máme  $N * (N+1) / 2$  různých optimálních podstromů.

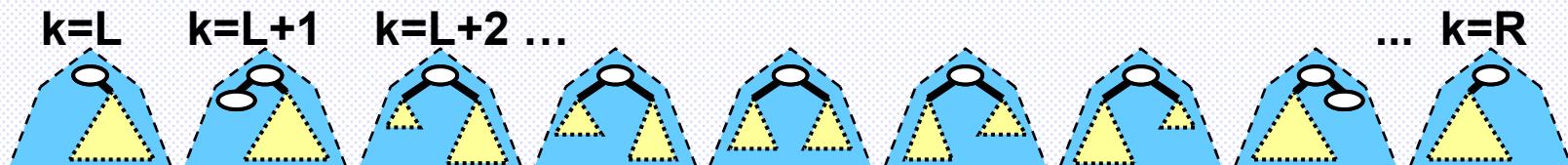
## Minimalizace ceny BVS

Idea rekurzivního řešení:



1. Předpoklad : Všechny menší optimální stromy jsou známy.

2. Zkus:  $k = L, L+1, L+2, \dots, R$



3. Zaregistruj index  $k$ , který minimalizuje cenu, tj. hodnotu

$$C_{1k} + \sum_{i=L}^{k-1} p_i + C_{2k} + \sum_{i=k+1}^R p_i + p_k$$

4. Klíč s indexem  $k$  je kořenem optimálního stromu.

## Minimalizace ceny BVS

$C(L,R)$  ..... Cena optimálního podstromu obsahujícího klíče s indexy  $L, L+1, L+2, \dots, R-1, R$

$$C(L,R) = \min_{L \leq k \leq R} \{ C(L, k-1) + \sum_{i=L}^{k-1} p_i + C(k+1, R) + \sum_{i=k+1}^R p_i + p_k \} =$$

$$= \min_{L \leq k \leq R} \{ C(L, k-1) + C(k+1, R) + \sum_{i=L}^R p_i \} =$$

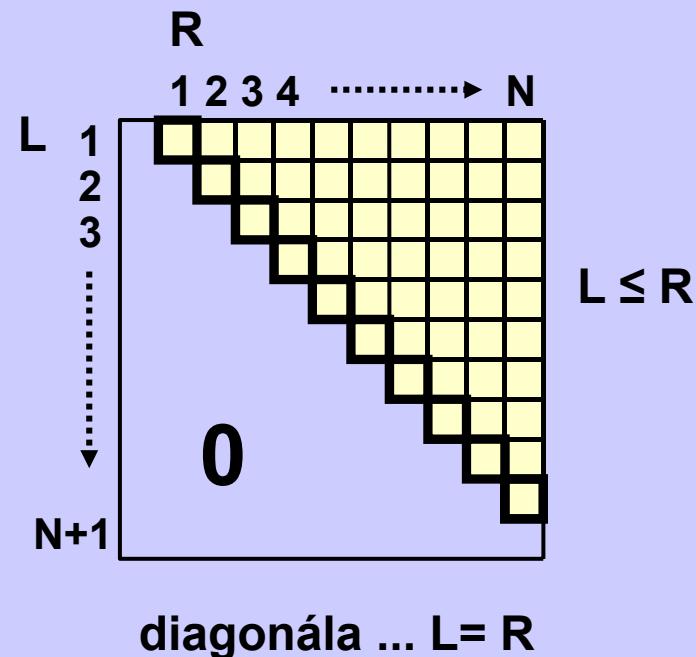
$$(*) = \min_{L \leq k \leq R} \{ C(L, k-1) + C(k+1, R) \} + \sum_{i=L}^R p_i$$

Hodnota k minimalizující (\*) je indexem kořenu optim. podstromu.

## Datové struktury pro výpočet optimálního BVS

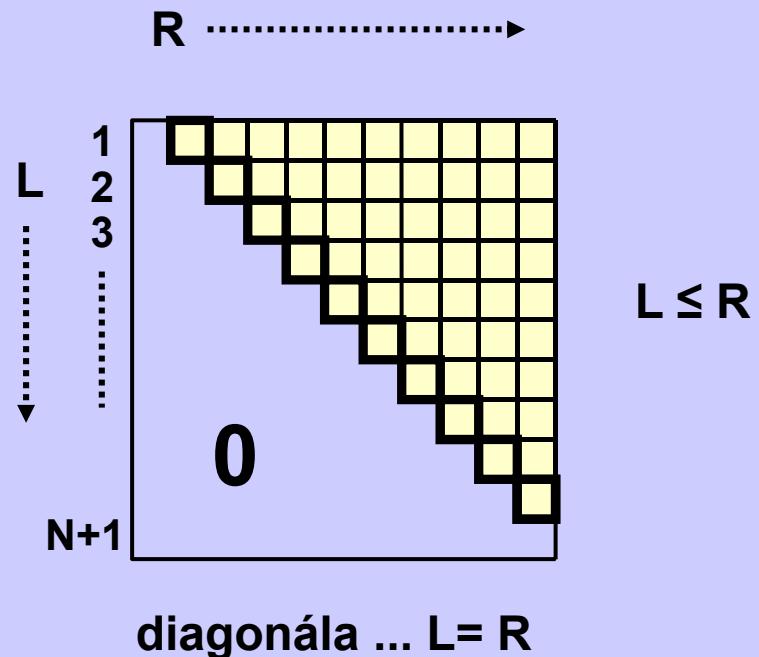
**Ceny optimálních podstromů**

**pole       $C [L][R]$       ( $L \leq R$ )**



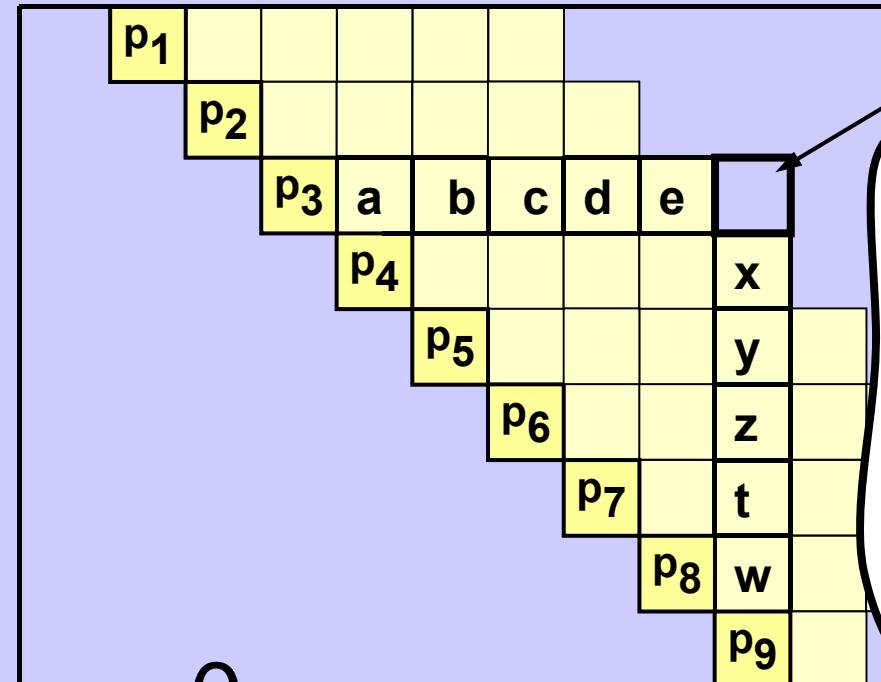
**Kořeny optimálních podstromů**

**pole      roots [L][R]      ( $L \leq R$ )**



## Výpočet optimálního BVS

### Cena konkrétního optimálního podstromu



$L=3, R=9$

$$C(L, R) = \min \{ C(L, k-1) + C(k+1, R) \} + \sum_{i=L}^R p_i$$

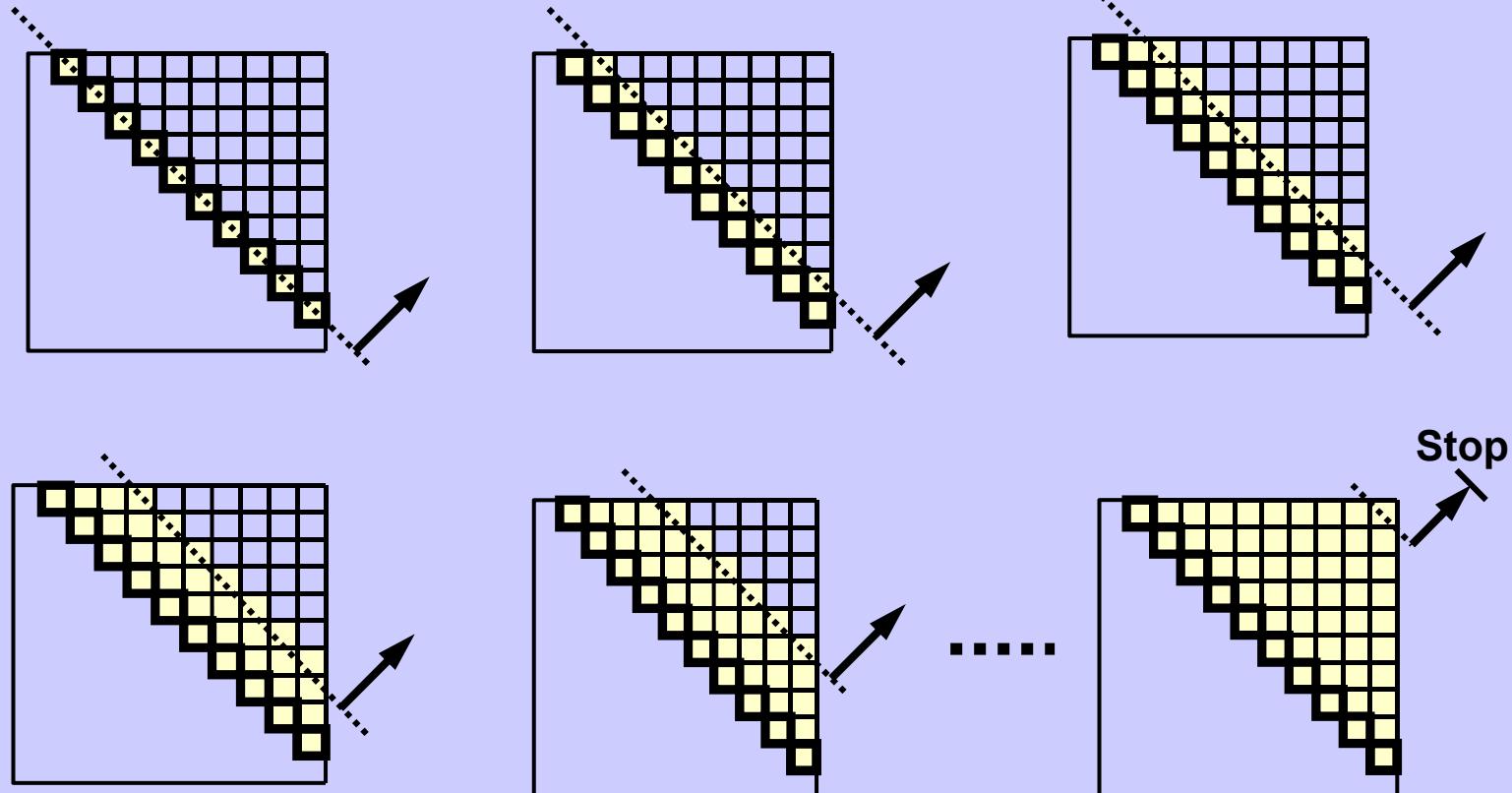
$$L \leq k \leq R$$

$$C(L, R) = \min \{ 0+x, p_3+y, a+z, b+t, c+w, d+p_9, e+0 \} + \sum_{i=L}^R p_i$$

## Výpočet optimálního BVS

Strategie DP

– nejprve se zpracují nejmenší podstromy, pak větší, atd...



## Výpočet optimálního BVS

### Výpočet DP tabulek cen a kořenů

```
void optimalTree() {
    int L, R; double min;

    // size = 1
    for( i=0; i<=N; i++ ) {
        C[i][i] = pravděpodobnost[i]; roots[i][i] = i;

    // size > 1
    for( int size = 2; size <= N; size++ ) {
        L = 1; R = size;
        while( R <= N ) {
            C[L][R] = min(C[L][k-1]+C[k+1][R], k = L..R);
            roots[L][R] = 'k minimalizující předch. řádek';
            C[L][R] += sum(C[i][i], i = L..R);
            L++; R++;
    } } }
```

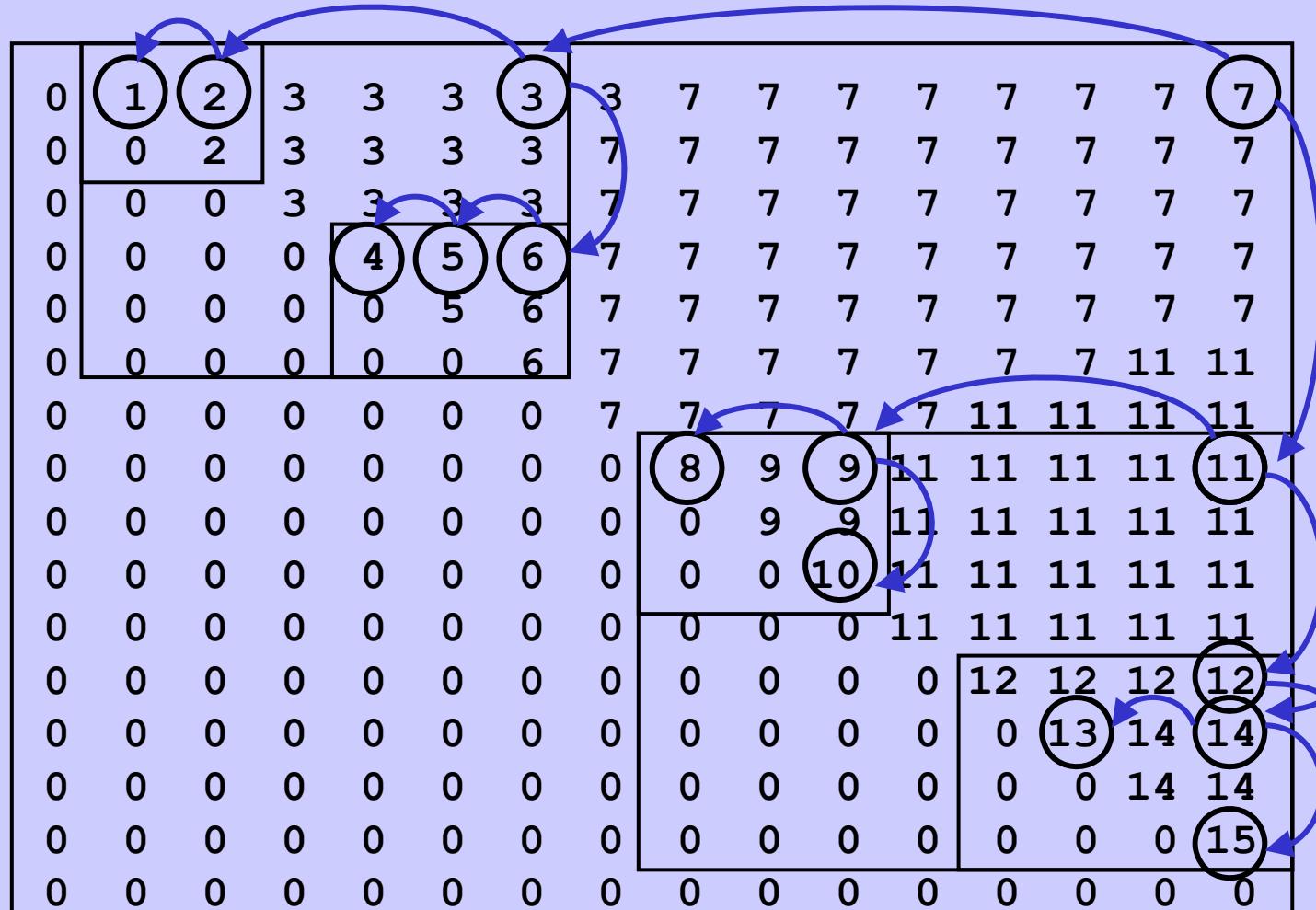
## Výpočet optimálního BVS

Vybudování optimálního stromu pomocí  
rekonstrukční tabulky kořenů

```
void buildTree( int L, int R) {  
  
    if (R < L) return;  
  
    int keyIndex = roots[L][R];  
    // keys ... sorted array of keys  
    int key = keys[roots[L][R]];  
  
    insert(root, key);      // standard BST insert  
    buildTree( L, keyIndex -1 );  
    buildTree( keyIndex +1, R );  
}
```

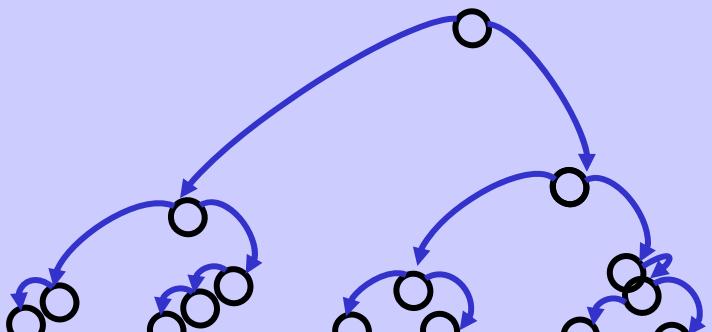
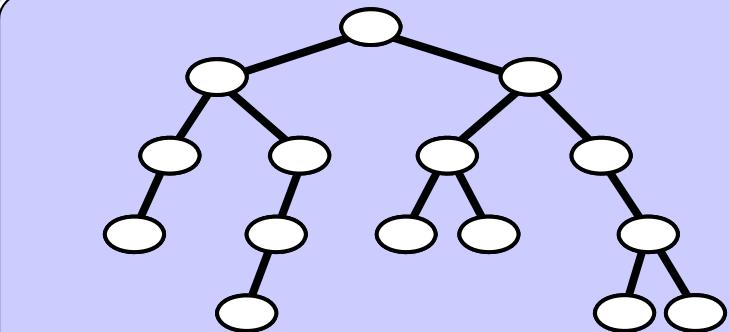
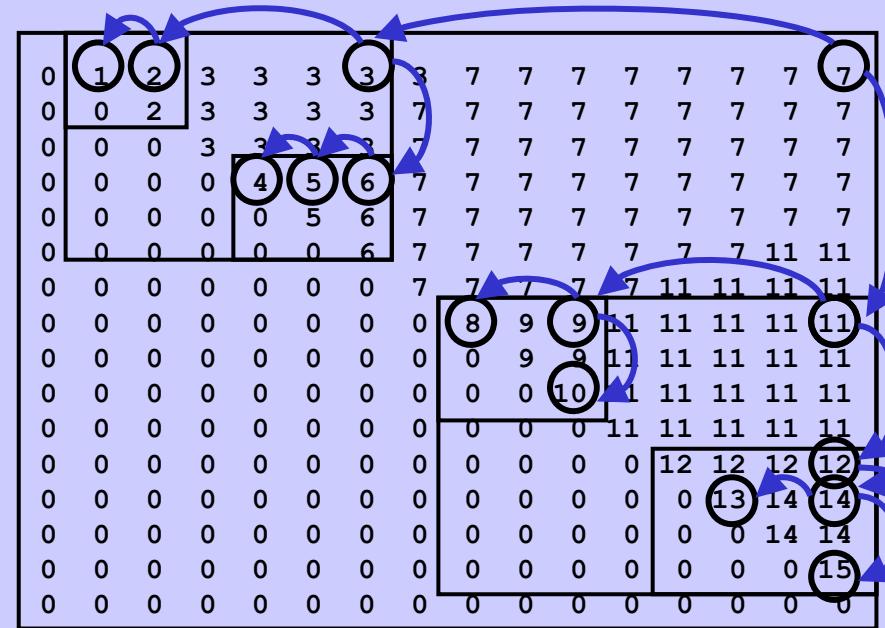
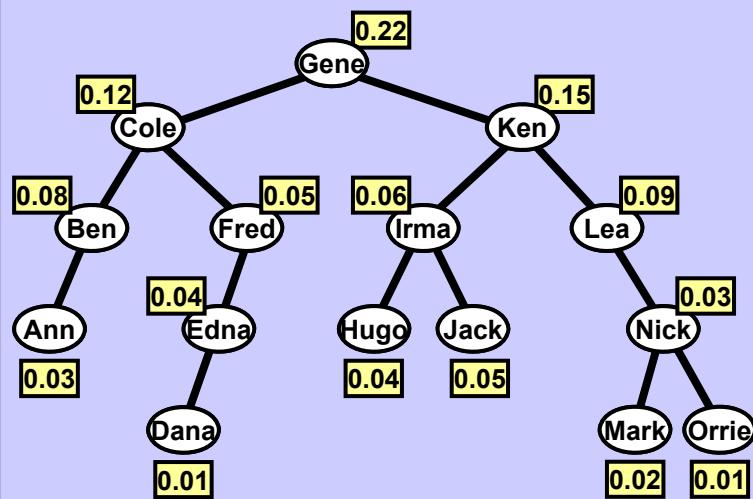
## Výpočet optimálního BVS

### Kořeny optimálních podstromů



## Výpočet optimálního BVS

### Korespondence stromů



17

## Výpočet optimálního BVS

### Ceny optimálních podstromů

	1-A	2-B	3-C	4-D	5-E	6-F	7-G	8-H	9-I	10-J	11-K	12-L	13-M	14-N	15-O
1-A	0.03	0.14	0.37	0.39	0.48	0.63	1.17	1.26	1.42	1.57	2.02	2.29	2.37	2.51	2.56
2-B	0	0.08	0.28	0.30	0.39	0.54	1.06	1.14	1.30	1.45	1.90	2.17	2.25	2.39	2.44
3-C	0	0	0.12	0.14	0.23	0.38	0.82	0.90	1.06	1.21	1.66	1.93	2.01	2.15	2.20
4-D	0	0	0	0.01	0.06	0.16	0.48	0.56	0.72	0.87	1.32	1.59	1.67	1.81	1.86
5-E	0	0	0	0	0.04	0.13	0.44	0.52	0.68	0.83	1.28	1.55	1.63	1.77	1.82
6-F	0	0	0	0	0	0.05	0.32	0.40	0.56	0.71	1.16	1.43	1.51	1.63	1.67
7-G	0	0	0	0	0	0	0.22	0.30	0.46	0.61	1.06	1.31	1.37	1.48	1.52
8-H	0	0	0	0	0	0	0	0.04	0.14	0.24	0.54	0.72	0.78	0.89	0.93
9-I	0	0	0	0	0	0	0	0	0.06	0.16	0.42	0.60	0.66	0.77	0.81
10-J	0	0	0	0	0	0	0	0	0	0.05	0.25	0.43	0.49	0.60	0.64
11-K	0	0	0	0	0	0	0	0	0	0	0.15	0.33	0.39	0.50	0.54
12-L	0	0	0	0	0	0	0	0	0	0	0	0.09	0.13	0.21	0.24
13-M	0	0	0	0	0	0	0	0	0	0	0	0	0.02	0.07	0.09
14-N	0	0	0	0	0	0	0	0	0	0	0	0	0.03	0.05	
15-O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.01

# Dynamické programování

## Nejdelší společná podposloupnost

## Nejdelší společná podposloupnost

Dvě  
posloupnosti

A: 

C	B	E	A	D	D	E	A
---	---	---	---	---	---	---	---

 $|A| = 8$   
B: 

D	E	C	D	B	D	A
---	---	---	---	---	---	---

 $|B| = 7$

Společná  
podposloupnost

A: 

C	B	E	A	D	D	E	A
---	---	---	---	---	---	---	---

  
B: 

D	E	C	D	B	D	A
---	---	---	---	---	---	---

  
C: 

C	D	A
---	---	---

 $|C| = 3$

Nejdelší  
společná  
podposloupnost  
(NSP)

A: 

C	B	E	A	D	D	E	A
---	---	---	---	---	---	---	---

  
B: 

D	E	C	D	B	D	A
---	---	---	---	---	---	---

  
C: 

E	D	D	A
---	---	---	---

 $|C| = 4$

## Nejdelší společná podposloupnost

$A_n: (a_1, a_2, \dots, a_n)$

$B_m: (b_1, b_2, \dots, b_m)$

$C_k: (c_1, c_2, \dots, c_k)$

.....  
 $C_k = \text{LCS}(A_n, B_m)$

1 2 3 4 5 6 7 8

$A_8:$  C B E A D D E A

$B_7:$  D E C D B D A

$C_4:$  E D D A

### Rekurzivní pravidla:

$(a_n = b_m) \implies (c_k = a_n = b_m) \ \& \ (C_{k-1} = \text{LCS}(A_{n-1}, B_{m-1}))$

1 2 3 4 5 6 7 8

$A_8:$  C B E A D D E A

$B_7:$  D E C D B D A

$C_4:$  E D D A

1 2 3 4 5 6 7 8

$A_7:$  C B E A D D E A

$B_6:$  D E C D B D A

$C_3:$  E D D A

## Nejdelší společná podposloupnost

$(a_n \neq b_m) \& (c_k \neq a_n) \Rightarrow (C_k = \text{LCS}(A_{n-1}, B_m))$

	1	2	3	4	5	6	7	8
$A_7:$	C	B	E	A	D	D	E	
$B_6:$	D	E	C	D	B	D		
$C_3:$	E	D	D					

	1	2	3	4	5	6	7	8
$A_6:$	C	B	E	A	D	D	E	
$B_6:$	D	E	C	D	B	D		
$C_3:$	E	D	D					

$(a_n \neq b_m) \& (c_k \neq b_m) \Rightarrow (C_k = \text{LCS}(A_n, B_{m-1}))$

	1	2	3	4	5	6	7	8
$A_5:$	C	B	E	A	D			
$B_5:$	D	E	C	D	B			
$C_2:$	E	D						

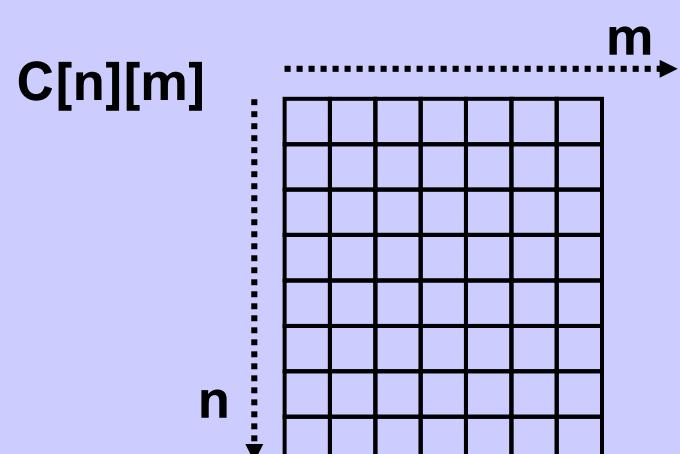
	1	2	3	4	5	6	7	8
$A_5:$	C	B	E	A	D			
$B_4:$	D	E	C	D		B		
$C_2:$	E	D						

## Nejdelší společná podposloupnost

### Rekurzivní funkce – délka LCS

$$C(n,m) = \begin{cases} 0 & n = 0 \text{ or } m = 0 \\ C(n-1, m-1) + 1 & n > 0, m > 0, a_n = b_m \\ \max\{ C(n-1, m), C(n, m-1) \} & n > 0, m > 0, a_n \neq b_m \end{cases}$$

### Strategie dynamického programování



```
for( a=1; a<=n; a++ )  
    for( b=1; b<=m; b++ )  
        C[a][b] = . . . . ;  
    }
```

## Nejdelší společná podposloupnost

### Konstrukce DP tabulek pro LCS

```
void findLCS() {
    for( int a=1; a<=n; a++ )
        for( int b=1; b<=m; b++ )
            if( A[a] == B[b] ) {
                C[a][b] = C[a-1][b-1]+1;
                arrows[a][b] = DIAG; ↑
            }
            else
                if( C[a-1][b] > C[a][b-1] ) {
                    C[a][b] = C[a-1][b];
                    arrows[a][b] = UP; ↑
                }
                else {
                    C[a][b] = C[a][b-1];
                    arrows[a][b] = LEFT; ←
                }
}
```

## Nejdelší společná podposloupnost

Pole NSP pro  
"CBEADDEA" a  
"DECDBDA"

		0	1	2	3	4	5	6	7
		B:	D	E	C	D	B	D	A
A:	0	0	0	0	0	0	0	0	0
	1	C							
	2		B						
	3			E					
	4				A				
	5					D			
	6						D		
	7							E	
	8								A

## Nejdelší společná podposloupnost

Výpis NSP -- rekuzivně :)

```
void outLCS( int a, int b ) {
    if( a == 0 || b == 0 ) return;

    if( arrows[a][b] == DIAG ) {
        outLCS(a-1, b-1);           // recursion ...
        print(A[a]);                // ... reverses the sequence!
    }
    else
        if( arrows[a][b] == UP )
            outLCS(a-1, b);
        else
            outLCS(a, b-1);
}
```

# ALG 11

## Dynamické programování

**Úloha batohu neomezená**

**Úloha batohu 0/1**

## Úloha batohu / Knapsack problem

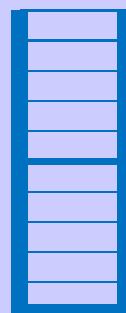
Máme  $N$  předmětů, každý s váhou  $V_i$  a cenou  $C_i$  ( $i = 1, 2, \dots, N$ ) a batoh s kapacitou váhy  $K$ .

Máme naložit batoh těmito předměty tak, aby kapacita  $K$  nebyla překročena a obsah měl maximální cenu.

**Neomezená varianta** -- Každý předmět lze použít libovolněkrát.

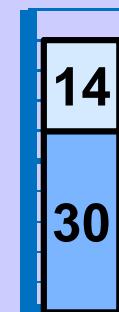
**0/1 varianta** -- Každý předmět lze použít nejvýše jednou.

Schematický batoh  
s kapacitou 10



Předměty  
s uvedenou  
cenou,  
váha ~ výška

Několik možných  
konfigurací



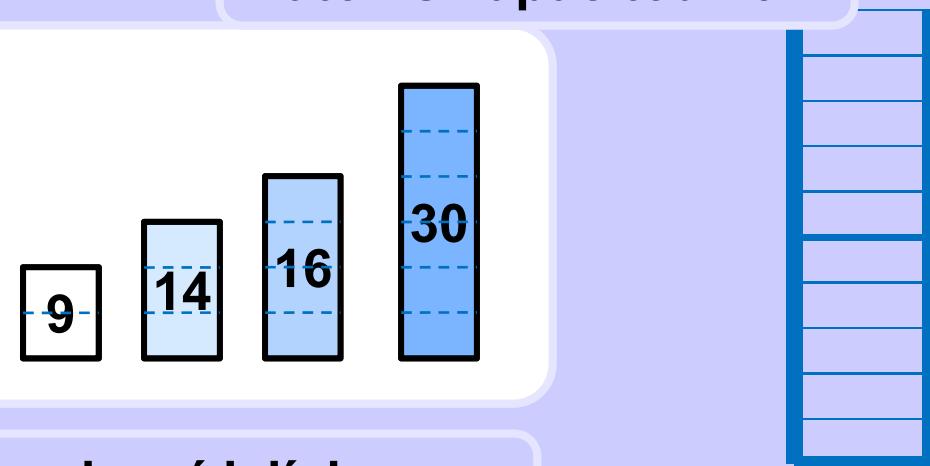
## Neomezená úloha batohu

## Batoh s kapacitou 10

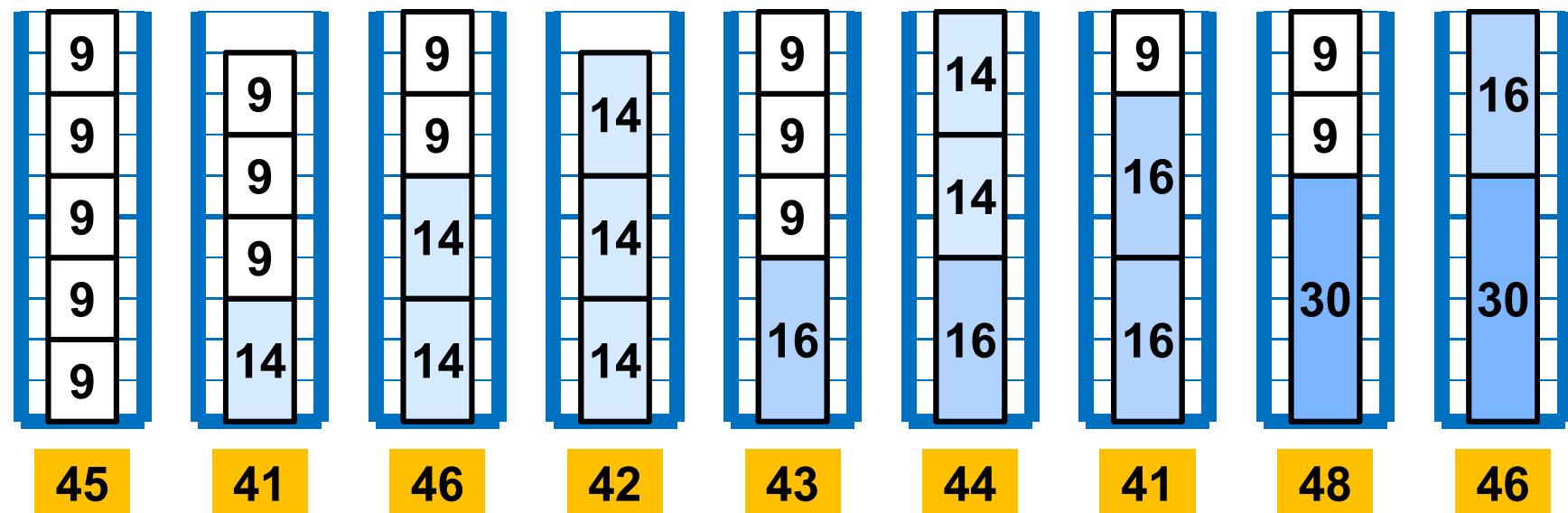
### Příklad

$$N = 4$$

Váha	2	3	4	6
Cena	9	14	16	30



Některé možnosti naplnění a odpovídající ceny



## Neomezená úloha batohu

Použijeme  $K+1$  batohů, o kapacitách  $0, 1, 2, 3, \dots, K$ .  
Hodnotu optimálního naplnění batohu s kapacitou  $K$   
lze získat jako maximum z hodnot

- (optimalní naplnění batohu o kapacitě  $K - V_1$ ) +  $C_1$ ,
- (optimalní naplnění batohu o kapacitě  $K - V_2$ ) +  $C_2$ ,
- ...
- (optimalní naplnění batohu o kapacitě  $K - V_N$ ) +  $C_N$ .

Optimální naplnění batohu o kapacitě  $K - V_i$  ( $i = 1..N$ ) je stejnou úlohou, jen s menšími daty. Hodnoty předpočítáme standardně metodou DP do 1D tabulky.

Neomezenou úlohu batohu lze přímo vyjádřit jako úlohu nalezení nejdelsí cesty v DAG. Postup řešení je identický.

## Neomezená úloha batohu -- převod na DAG

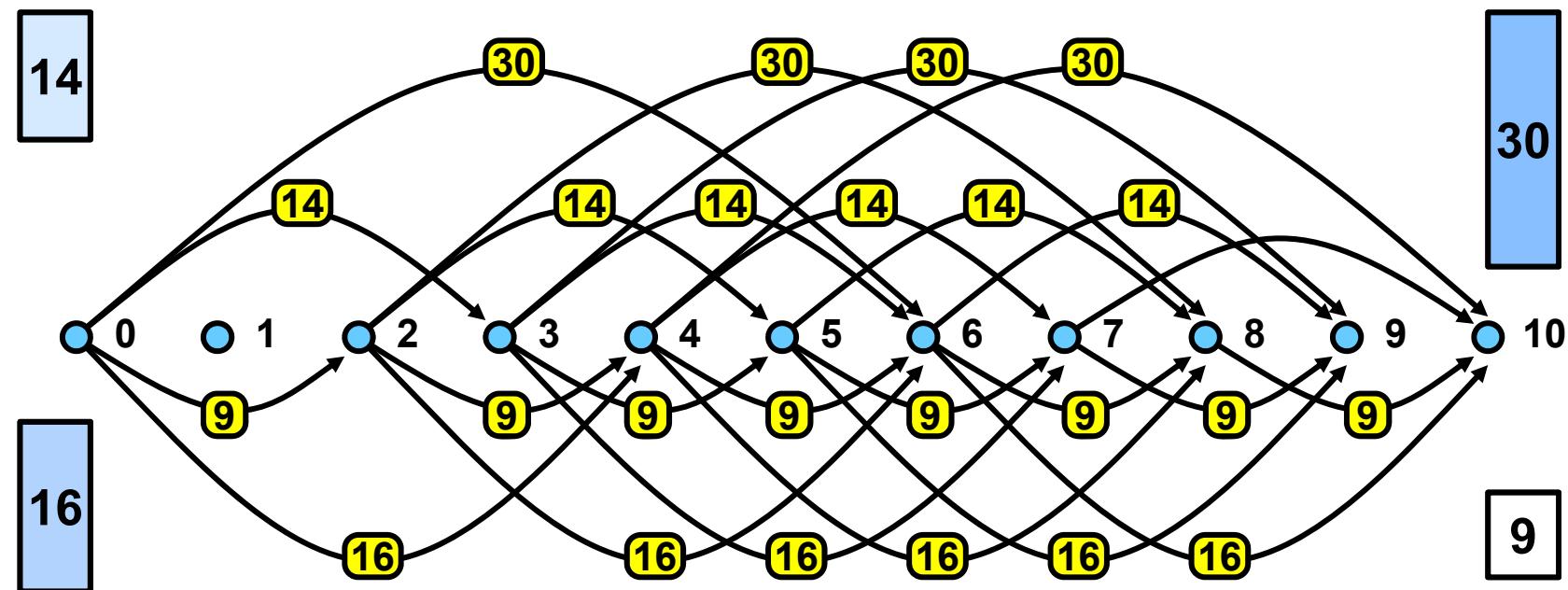
DAG:

Uzly: Kapacity 0, 1, 2, 3, ..., K.

Hrany: Z uzlu X vedou hrany po řadě do uzlů X+V1, X+V2, ..., X+VN, jsou po řadě ohodnoceny cenami C1, C2, ..., CN.

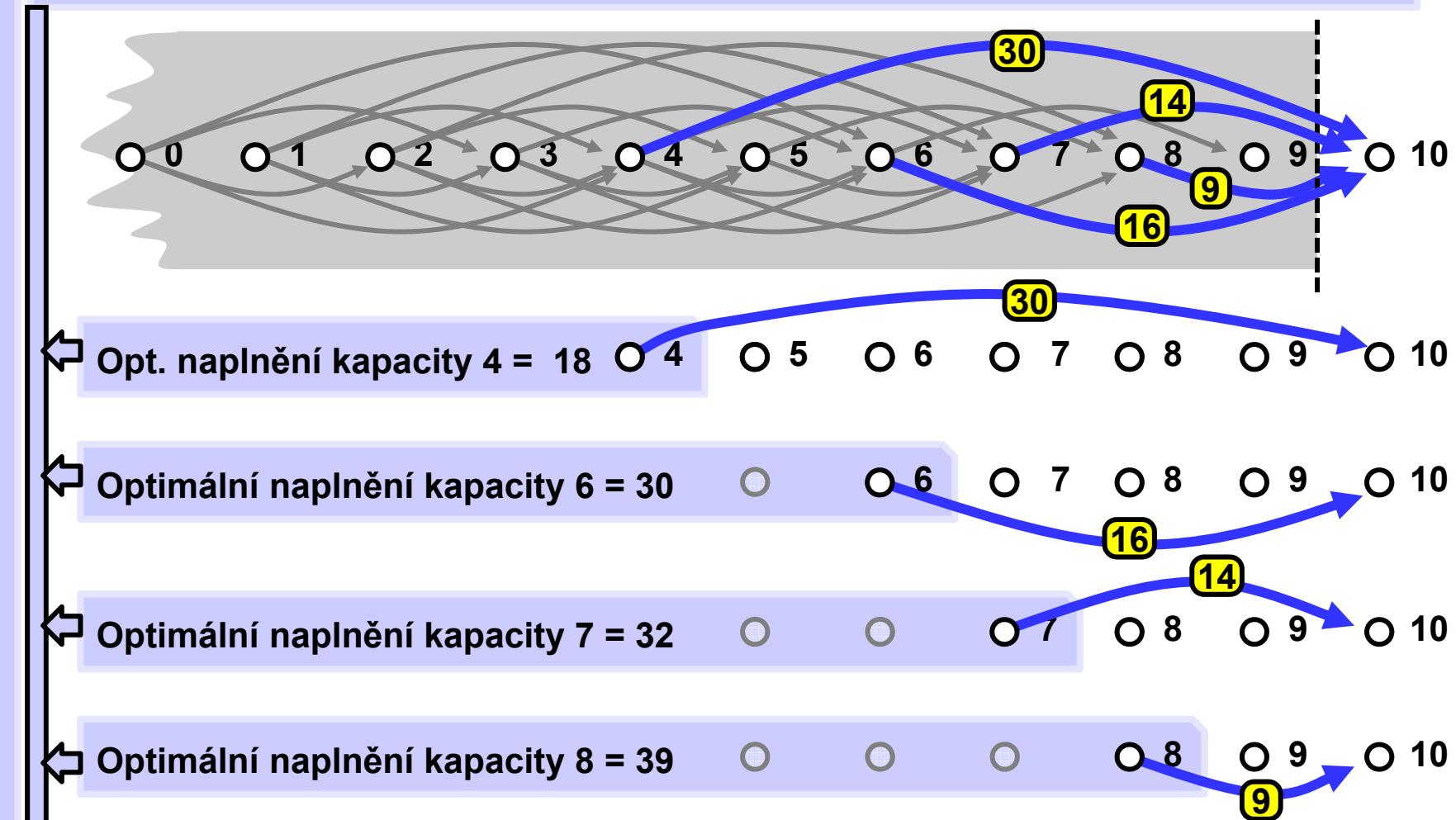
Příklad

$K = 10, N = 4, V_i = (2, 3, 4, 6), C_i = (9, 14, 16, 30), i = 1..4.$



## Neomezená úloha batohu -- jako DAG

Optimální naplnění kapacity 10 = ??

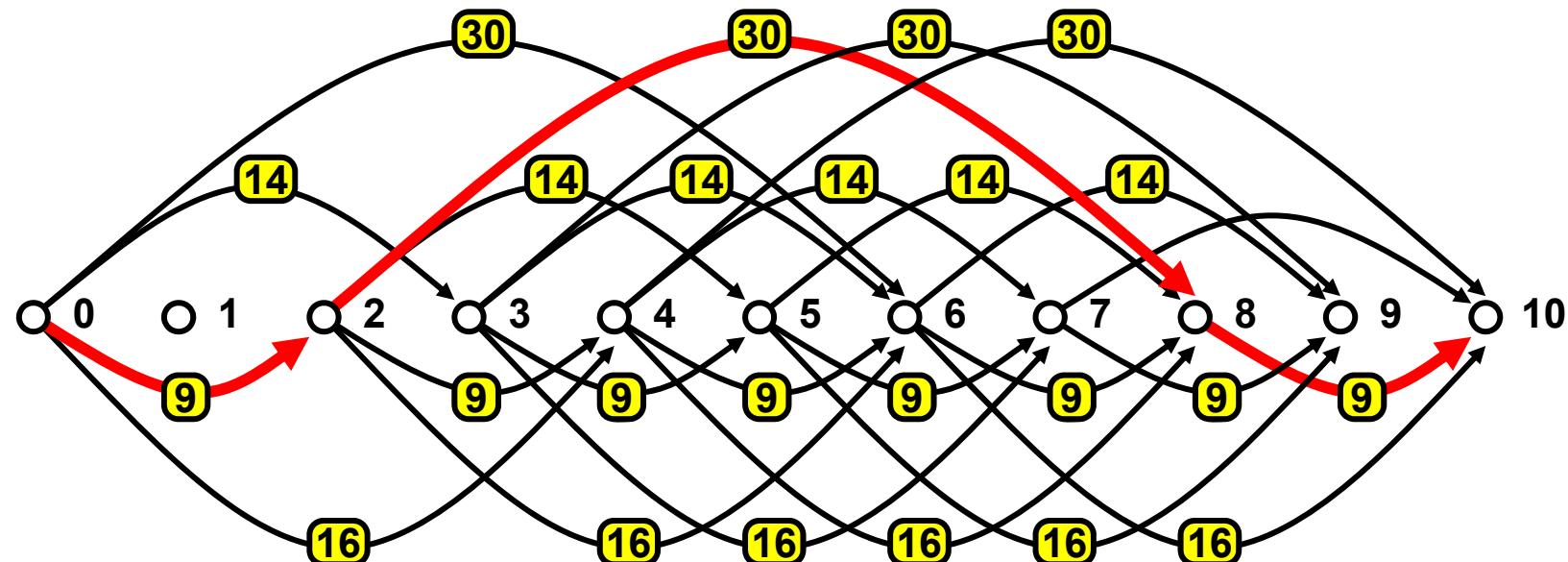


$$\text{Optimální naplnění kapacity } 10 = \max(18 + 30, 30 + 16, 32 + 14, 39 + 9) = 48$$

## Neomezená úloha batohu

Nejdelší cesta odpovídá optimálnímu naplnění batohu.  
Dvě hrany s cenou 9 a jedna hrana s cenou 30, celkem cena = 48.

Batoh optimálně naplníme dvěma předměty s váhou 2 a cenou 9  
a jedním předmětem s váhou 6 a cenou 30.

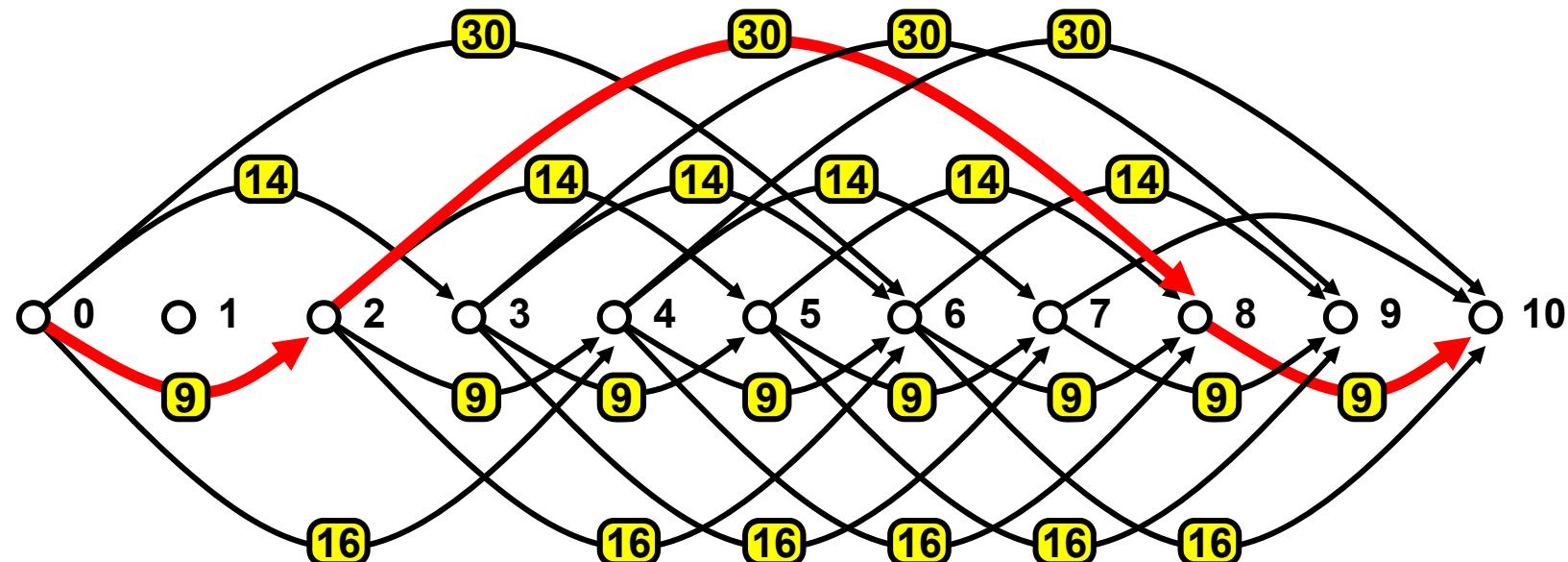


## Neomezená úloha batohu -- asymptotická složitost

DAG obsahuje  $K+1$  uzelů a méně než  $K^*N$  hran.

Má tedy  $V = \Theta(K)$  uzelů a  $E = O(K^*N)$  hran.

Asymptotická složitost hledání nejdélší cesty je  $\Theta(V+E)$ , máme tedy pro neomezenou úlohu batohu asymptotickou složitost  $O(K + K^*N) = O(K^*N)$ .



## Neomezená úloha batohu -- Asymptotická složitost

### Zdánlivá nesrovnalost

1. Literatura: NP těžký problém, není znám efektivní algoritmus.
2. ALG OI: DP řeší úlohu v čase v  $O(N^*K)$ , tedy efektivně?

**Délka výpočtu DP je lineárně závislá na velikosti kapacity K.**

#### Příklad

Velkou kapacitu  $2^{64}$  lze zadat velmi krátkým zápisem

Kapacita = 18446744073709551616.

$N = 3$ . Položky (váha, cena): (2, 345), (3, 456), (5, 678).

Data lze zapsat do cca 100 bitů < 16 Bytů < "dva longy"

Výpočet pomocí DP potrvá přes 584 roky

za předpokladu, že za 1 sec vyplní  $10^9$  prvků tabulky.

**Délka výpočtu DP je exponenciálně závislá na délce řetězce definujícího kapacitu K.**

## 0/1 úloha batohu

Každý předmět lze použít nejvýše 1 krát.

Máme vybrat vhodnou podmnožinu předmětů splňující zadání úlohy. Každé podmnožině lze přiřadit charakteristický vektor z hodnot 0/1 délky  $N$ . Pozice ve vektoru odpovídá předmětu, 0 resp. 1 odpovídá nepřítomnosti resp. přítomnosti předmětu v této podmnožině. Binárních vektorů délky  $N$  je celkem  $2^N$ , systematické probírání všech možných podmnožin bude mít exponenciální asymptotickou složitost, nehodí se.

DP poskytuje (pro relativně nevelké kapacity) výhodnější postup.

## 0/1 úloha batohu

### Příklad

$$N = 4$$

Váha Cena

2	9
3	14
4	16
6	30

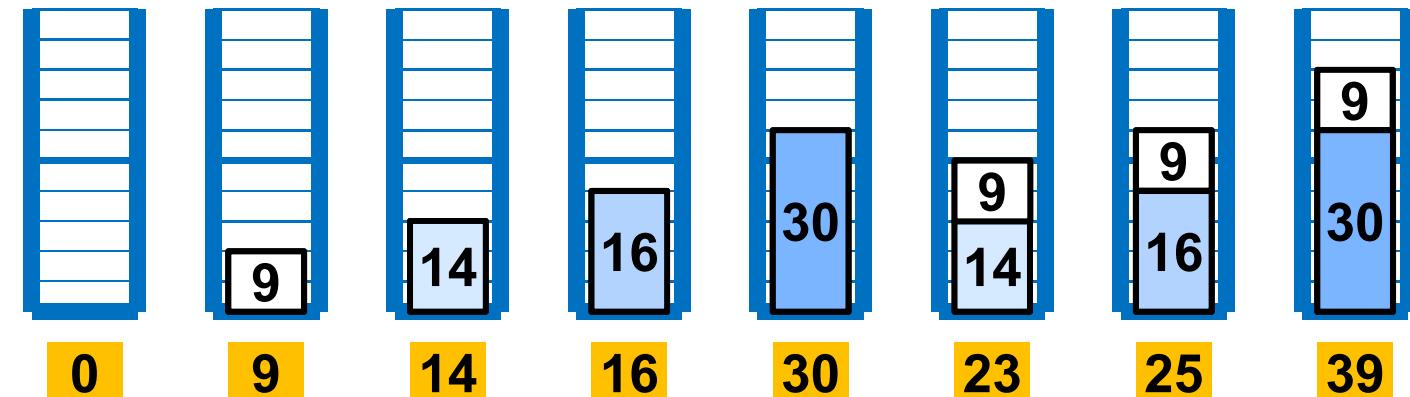
9

14

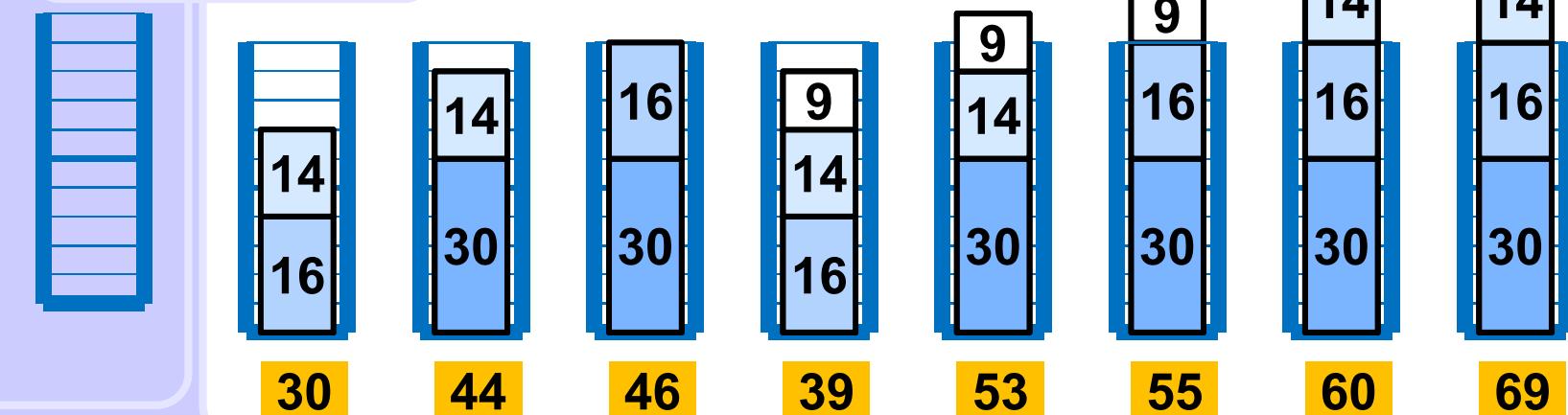
16

30

Všech 16 podmnožin čtyř předmětů a jejich ceny



Batoh  
s kapacitou 10



## 0/1 úloha batohu -- řešení

Použijeme  $K+1$  batohů, o kapacitách  $0, 1, 2, 3, \dots, K$ .

Použijeme  $N+1$  souborů předmětů.

Soubor 0 neobsahuje žádný předmět.

Soubor 1 obsahuje předmět 1.

Soubor 2 obsahuje předměty 1 a 2.

Soubor 3 obsahuje předměty 1, 2, 3.

...

Soubor  $N$  obsahuje předměty  $1, 2, 3, \dots, N$ .

Na pořadí předmětů nezáleží, je ale zafixované.

Pro každou kapacitu a pro každý soubor budeme řešit stejnou úlohu metodou DP, v pořadí od menších hodnot k větším.

## 0/1 úloha batohu -- řešení

Označme symbolem  $U(x, y)$  úlohu se souborem předmětů 1, 2, ..., x a s kapacitou batohu y a symbolem  $\text{Opt}(x, y)$  optimální řešení této úlohy.

Pro řešení  $U(x,y)$  použijeme optimální řešení úloh  $U(x-1, \_)$ :

Bud' do  $\text{Opt}(x, y)$  zahrneme předmět x nebo jej nezahrneme. V prvním případě použijeme hodnotu řešení pro batoh s kapacitou menší o velikost váhy  $Vx$ , tedy hodnotu  $\text{Opt}(x-1, y-Vx)$ , ke které přičteme cenu  $Cx$  předmětu x. V druhém případě beze změny použijeme hodnotu  $\text{Opt}(x-1, y)$ . Z obou hodnot vybereme tu výhodnější a dostaváme tak:

$$\text{Opt}(x, y) = \max(\text{Opt}(x-1, y), \text{Opt}(x-1, y-Vx) + Cx).$$

Dále zřejmě platí  $\text{Opt}(0, y) = \text{Opt}(x, 0) = 0$ , pro  $x = 0..N$ ,  $y = 0..K$ .

## 0/1 úloha batohu -- řešení

Pro  $x = 1..N$ ,  $y = 0..K$ :

$$\text{Opt}(x, y) = \max(\text{Opt}(x-1, y), \text{Opt}(x-1, y-Vx) + Cx).$$

$$\text{Opt}(0, y) = \text{Opt}(x, 0) = \text{Opt}(0, 0) = 0.$$

Pokud  $y-Vx < 0$ , položíme  $\text{Opt}(x, y-Vx) = -\infty$  (a netabelujeme).

Hodnoty  $\text{Opt}(x,y)$  tabelujeme ve 2D tabulce velikosti  $(N+1) \times (K+1)$  s řádkovým indexem  $x$  (předměty)  
a sloupcovým indexem  $y$  (kapacity menších batohů).

Pro rekonstrukci optimálního řešení použijeme tabulku předchůdců stejné velikosti jako tabulku pro  $\text{Opt}(x, y)$ .  
Předchůdce leží vždy v předchozím řádku  $x-1$ , stačí registrovat buď pozici  $y$  (beze změny) nebo pozici  $y-Vx$  (přidán předmět  $x$ ).

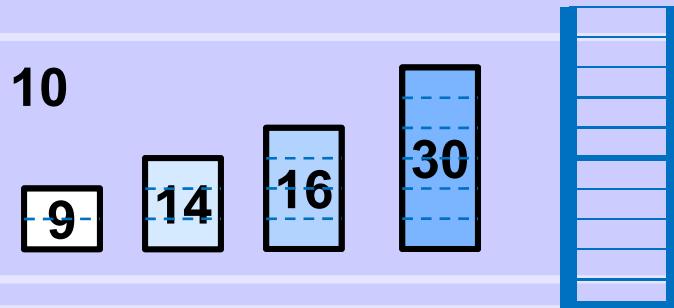
## 0/1 úloha batohu

Příklad

$N = 4$

Váha    2    3    4    6  
Cena    9    14    16    30

Kapacita = 10



$\text{Opt}(x, y)$

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	9	9	9	9	9	9	9	9	9
2	0	0	9	14	14	23	23	23	23	23	23
3	0	0	9	14	16	23	25	30	30	39	39
4	0	0	9	14	16	23	30	30	39	44	46

$\text{Pred}(x, y)$

	0	1	2	3	4	5	6	7	8	9	10
0	--	--	--	--	--	--	--	--	--	--	--
1	0	1	0	1	2	3	4	5	6	7	8
2	0	1	2	0	1	2	3	4	5	6	7
3	0	1	2	3	0	5	2	3	4	5	6
4	0	1	2	3	4	5	0	7	2	3	4

## 0/1 úloha batohu

### Vyjádření jako optimální cesty v DAG

Uzly DAG budou jednotlivé hodnoty  $\text{Opt}(x, y)$ ,  $x = 0..N$ ,  $y = 0..K$ , celkem bude mít DAG  $(N+1) \times (K+1)$  uzelů.

Do uzlu  $\text{Opt}(x, y)$  povede hrana

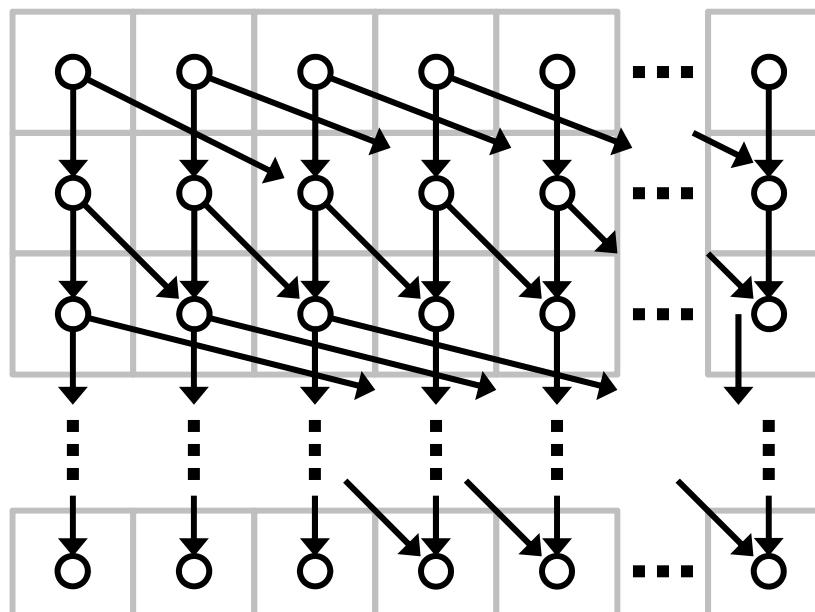
- $\text{Opt}(x-1, y) \rightarrow \text{Opt}(x, y)$   
ohodnocená 0 (žádný přidaný předmět),
- a pokud  $y - V_x \geq 0$ , také hrana  
 $\text{Opt}(x-1, y - V_x) \rightarrow \text{Opt}(x, y)$   
ohodnocená cenou  $C_x$  (cenou přidaného předmětu x).

V takto zkonstruovaném DAG hledáme nejdelší (= nejcennější) cestu standardní DP metodou.

Jaké je topologické uspořádání tohoto DAG?

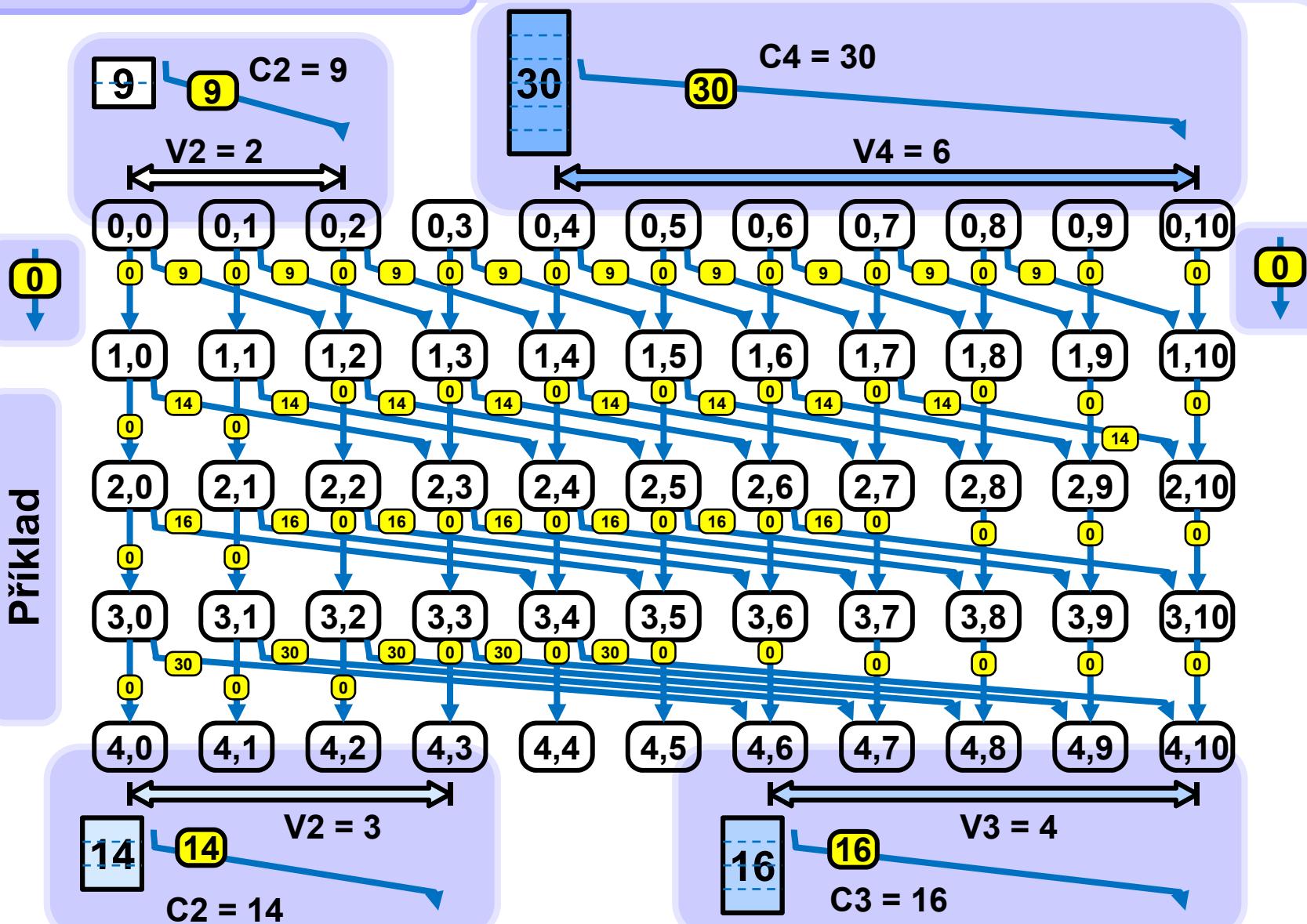
## 0/1 úloha batohu - topologické uspořádání DAG

DAG můžeme uvažovat nakreslený formálně do DP tabulky, přičemž uzel  $\text{Opt}(x, y)$  leží v buňce s indexy  $x$  a  $y$ . Pak hrany DAG vedou vždy pouze z předchozího řádku do následujícího řádku. Pokud tento DAG procházíme shora po řádcích, to jest ve stejném pořadí, v němž vyplňujeme DP tabulku, respektujeme jeho topologické uspořádání.

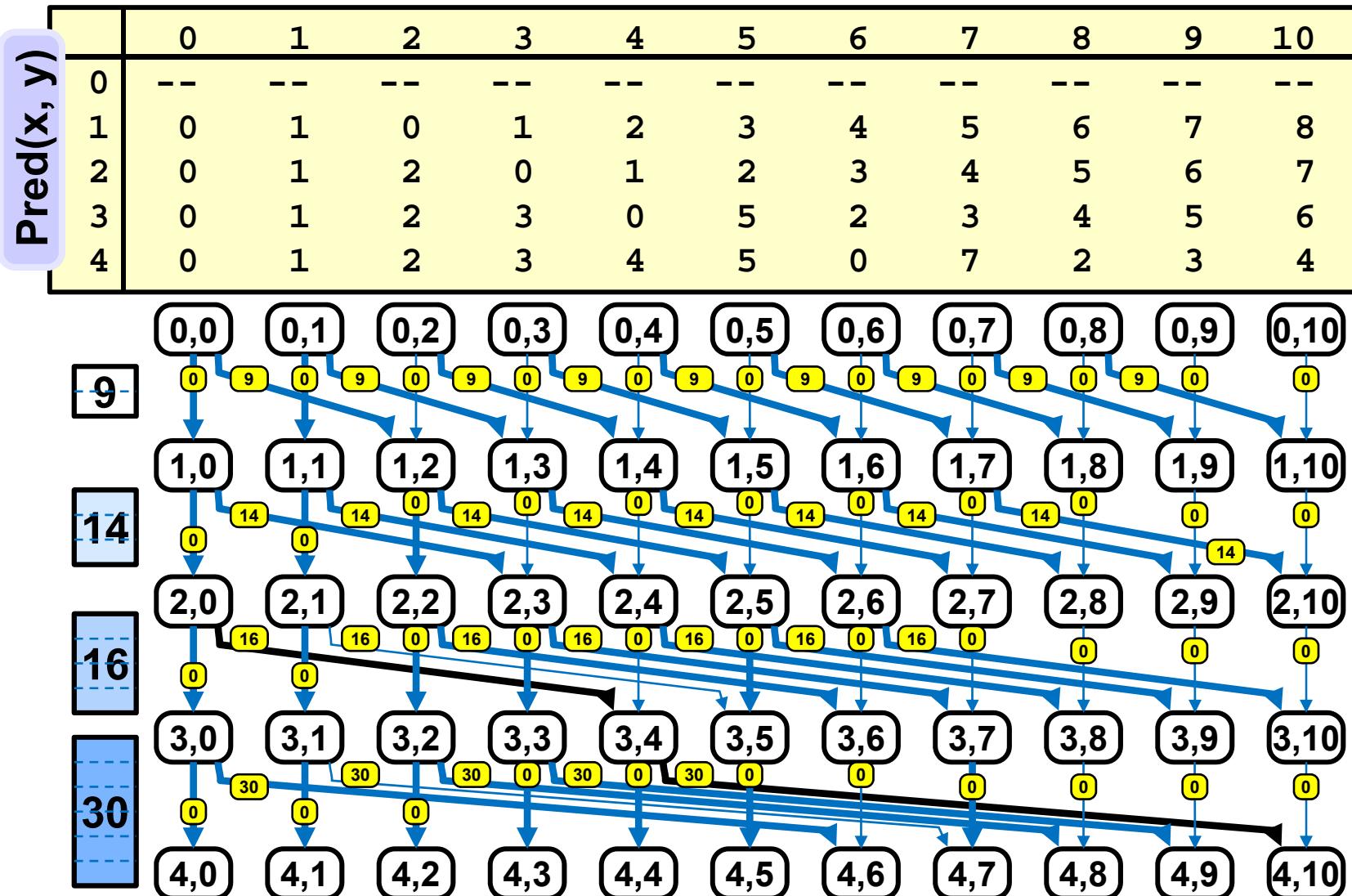


V tomto případě není nutno uzly DAG v topologickém uspořádání uvažovat v jedné přímce, "tabulkové" uspořádání je přehlednější.

## 0/1 úloha batohu -- DAG



## 0/1 úloha batohu -- rekonstrukce optimálního řešení pomocí tabulky předchůdců



**0/1 úloha batohu****Asymptotická složitost**

**Tabulka ... Velikost ...  $(N+1) \cdot (K+1) \in \Theta(N \cdot K)$**

**Vyplnění jedné buňky ...  $\Theta(1)$**

**Vyplnění tabulky ...  $\Theta(N \cdot K \cdot 1) = \Theta(N \cdot K)$ .**

**Rekonstrukce optimálního řešení  $\Theta(N)$ .**

**Celkem ...  $\Theta(N \cdot K + N) = \Theta(N \cdot K)$ .**

**DAG .... Uzlů ...  $(N+1) \cdot (K+1) \in \Theta(N \cdot K)$ .**

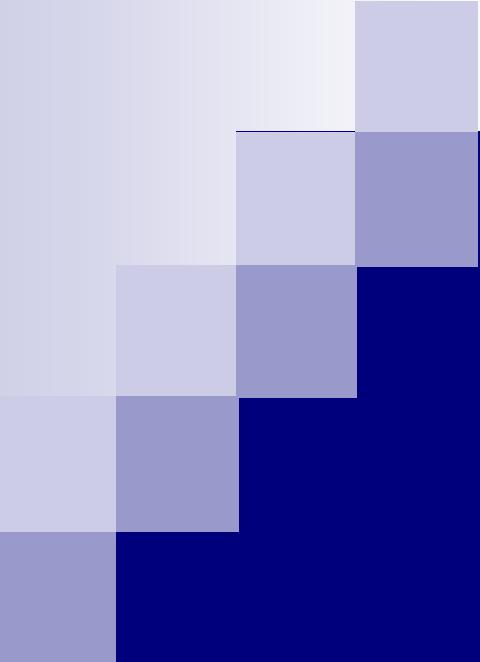
**Hran ... nejvýše  $2 \cdot (N+1) \cdot (K+1)$ , tj  $\in O(N \cdot K)$ .**

**Nalezení optimální cesty ...  $\Theta(\text{uzlů} + \text{hran}) = \Theta(N \cdot K)$ .**

**Řešení obou variant úlohy batohu, neomezené i 0/1,  
má asymptotickou složitost  $\Theta(N \cdot K)$ .**

**Přitom zárověň platí:**

**Asymptotická složitost DP řešení je exponenciální  
vzhledem k délce řetězce definujícího kapacitu K.**



# Algoritmizace

## Dynamické programování

Jiří Vyskočil, Marko Genyg-Berezovskyj

2010

# Rozděl a panuj (divide-and-conquer)

- **Rozděl (Divide):** Rozděl problém na několik podproblémů tak, aby tyto podproblémy odpovídaly původnímu problému, ale měly menší velikost.
- **Panuj (Conquer):** Rekurzivně vyřeš tyto podproblémy. Jestliže je podproblém dostatečně malý, vyřeš ho přímo (bez rekurze).
- **Kombinuj (Combine):** Zkombinuj řešení podproblémů do řešení výsledného původního problému.

# Dynamické programování

- Řeší problémy podobně jako metoda rozděl a panuj (divide-and-conquer) kombinováním řešení podproblémů.
- Slovo programování zde neznamená psaní kódu, ale použití určité metody pro řešení (podobně jako lineární programování, atd).
- Metoda rozděl a panuj typicky dělí problém rekurzivně na **nezávislé** podproblémy.
- Naproti tomu dynamické programování se používá tam, kde podproblémy nejsou nezávislé (tj. **podproblémy sdílejí společné podpodproblémy**). Metoda rozděl a panuj by v takovém případě opakovaně řešila stejné společné podpodproblémy, čímž by vykonávala zbytečnou práci navíc.
- Dynamické programování řeší každý podpodproblém pouze jednou a jeho řešení si pamatuje v tabulce. Tím zamezí opakovanému řešení stejných podproblémů.

# Dynamické programování

- Popis vývoje algoritmu dynamického programování:
  - 1) Charakterizace struktury optimálního řešení.
  - 2) Nalezení rekurzivní definice pro výpočet hodnoty optimálního řešení.
  - 3) Nalezení výpočtu hodnoty optimálního řešení zdola nahoru (od nejjednodušších podproblémů ke složitějším).
  - 4) Nalezení konstrukce optimálního řešení z vypočtených informací (tento krok může být vynechán pokud je hodnota optimálního řešení přímo vlastní optimální řešení).

# Příklad: Nejdelší společná podposloupnost

## Motivace

- V biologických aplikacích chceme často porovnávat řetězce DNA různých organismů a zjišťovat jejich podobnost.
- Řetězec DNA je posloupnost nukleotidů, kde nukleotid je buď (adenine A, guanine G, cytosine C, thymine T).
- Podobnost dvou řetězců DNA určuje délka jejich nejdelší společné podposloupnosti (dále už jen NSP).
- Podposloupnost je společná několika řetězcům, pokud je podposloupností každého řetězce jednotlivě.
- Mějme řetězec  $X = \langle x_1, x_2, \dots, x_m \rangle$ . Řetězec  $Z = \langle z_1, z_2, \dots, z_k \rangle$  je podposloupnost  $X$  pokud existuje rostoucí posloupnost indexů  $\langle i_1, i_2, \dots, i_k \rangle$  z  $X$  taková, že pro všechny  $j = 1, 2, \dots, k$ , platí  $x_{i_j} = z_j$ .  
*Příklad:*  $Z = \langle G, C, T, G \rangle$  je podposloupnost  $X = \langle A, G, C, G, T, A, G \rangle$  s korespondující posloupností indexů  $\langle 2, 3, 5, 7 \rangle$ .

## Úloha: Najděte nejdelší společnou podposloupnost pro dva zadané řetězce DNA.

# Příklad: Nejdelší společná podposloupnost

## 1) Charakterizace struktury optimálního řešení.

- Pokud bychom chtěli najít podposloupnost "hrubou" silou.
- To znamená otestovat všechny podmnožiny indexů  $\{1, 2, \dots, m\}$  menšího ze vstupních řetězců DNA ( $m$  je jeho délka).
- Takových podmnožin je ale  $2^m$ . Složitost tohoto přístupu by tedy byla exponenciální.
- Pro reálné DNA řetězce s alespoň tisícovkami nukleotidů by tedy byla tato metoda zcela nepoužitelná.
- Při detailnější analýze problému snadno zjistíme, že pro dva vstupní řetězce  $X = \langle x_1, x_2, \dots, x_m \rangle$  a  $Y = \langle y_1, y_2, \dots, y_n \rangle$  a hledanou výstupní podposloupnost  $Z = \langle z_1, z_2, \dots, z_k \rangle$  platí:
  - Pokud  $x_m = y_n$ , potom  $z_k = x_m = y_n$  a  $Z_{k-1}$  je NSP  $X_{m-1}$  a  $Y_{n-1}$ .
  - Pokud  $x_m \neq y_n$ , potom  $z_k \neq x_m$  implikuje, že  $Z$  je NSP  $X_{m-1}$  a  $Y$ .
  - Pokud  $x_m \neq y_n$ , potom  $z_k \neq y_n$  implikuje, že  $Z$  je NSP  $X$  a  $Y_{n-1}$ .

# Příklad: Nejdelší společná podposloupnost

2) Nalezení rekurzivní definice pro výpočet hodnoty optimálního řešení.

- Z předchozí analýzy vyplývá, že musíme umět řešit podúlohu pro postupně se odzadu zkracující vstupní řetězce.
- Nyní zavedeme pole  $c[i,j]$  jako délku NSP pro prefixy vstupních řetězců  $X_i$  a  $Y_j$ .
- Z předchozí analýzy si můžeme pole  $c$  definovat rekurzivně:

$$c[i,j] = \begin{cases} 0 & \text{pokud } i = 0 \text{ nebo } j = 0, \\ c[i-1, j-1] + 1 & \text{pokud } i, j > 0 \text{ a } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{pokud } i, j > 0 \text{ a } x_i \neq y_j. \end{cases}$$

# Příklad: Nejdelší společná podposloupnost

3) Nalezení výpočtu hodnoty optimálního řešení zdola nahoru.

**Function DÉLKA-NSP( $X, Y$ )**

```
1)    $m \leftarrow \text{length}[X];$ 
2)    $n \leftarrow \text{length}[Y];$ 
3)   for  $i \leftarrow 1$  to  $m$  do
4)        $c[i, 0] \leftarrow 0;$ 
5)   for  $j \leftarrow 0$  to  $n$  do
6)        $c[0, j] \leftarrow 0;$ 
7)   for  $i \leftarrow 1$  to  $m$  do
8)       for  $j \leftarrow 1$  to  $n$  do
9)           if  $x_i = y_j$ 
10)              then {  $c[i, j] \leftarrow c[i - 1, j - 1] + 1;$ 
11)                   $b[i, j] \leftarrow "\nwarrow";$  }
12)              else if  $c[i - 1, j] \geq c[i, j - 1];$ 
13)                  then {  $c[i, j] \leftarrow c[i - 1, j];$ 
14)                       $b[i, j] \leftarrow "\uparrow";$  }
15)              else {  $c[i, j] \leftarrow c[i, j - 1];$ 
16)                   $b[i, j] \leftarrow "\leftarrow";$  }
17)   return  $c$  and  $b;$ 
```

	$j$	0	1	2	3	4	5	6
$i$	$y_j$	B	D	C	A	B	A	
0	$x_i$	0	0	0	0	0	0	0
1	A	0	0	0	0	1	-1	1
2	B	0	1	-1	-1	1	2	-2
3	C	0	1	1	2	-2	2	2
4	B	0	1	1	2	2	3	-3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

## Příklad: Nejdelší společná podposloupnost

- 4) Nalezení konstrukce optimálního řešení z vypočtených informací.

### **Procedure** TISK-NSP( $b, X, i, j$ )

- ```
1) if ( $i = 0$ ) or ( $j = 0$ ) then exit;  
2) if  $b[i, j] = "↖"$  then { TISK-NSP( $b, X, i - 1, j - 1$ );  
3) print  $x_i$  ; }  
4) else if  $b[i, j] = "↑"$  then TISK-NSP( $b, X, i - 1, j$ );  
5) else TISK-NSP( $b, X, i, j - 1$ );
```

# Příklad: Nejdelší společná podposloupnost

- Asymptotická složitost algoritmu:
  - Časová:  $\Theta(mn) + O(m+n) = \Theta(mn)$
  - Paměťová:  $\Theta(mn)$
- Možná vylepšení:
  - Pomocné pole  $b$  lze nahradit testem v poli  $c$  v konstantním čase.
  - Nepotřebujeme celé pole  $c$ , ale pouze stačí v paměti udržovat předposlední a poslední řádek a sloupec.  
(Pak ale nebudeme schopni v čase  $O(m+n)$  rekonstruovat hledanou podposloupnost.)

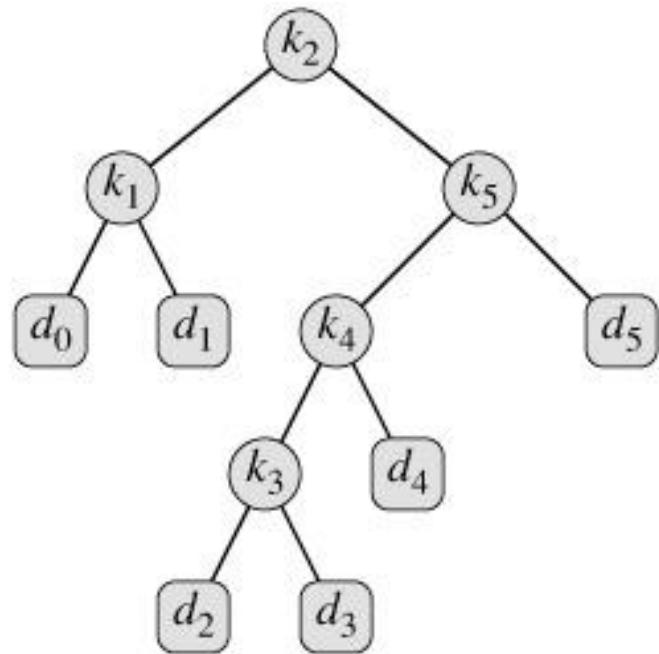
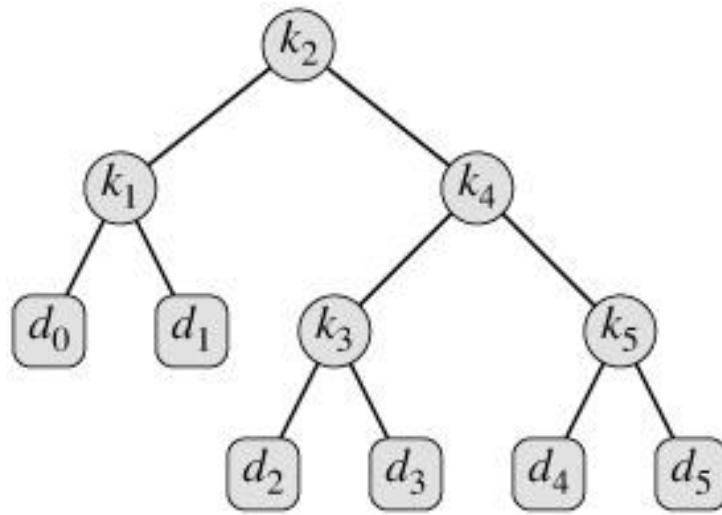
# Příklad: Optimální vyhledávací strom

- Motivace:
  - Máme vytvořit překlad textu (posloupnosti slov) z angličtiny do češtiny.
  - Přitom víme, že jak se jednotlivá anglická slova často v textu vyskytují.
  - Kdybychom sestavili pouze obyčejný vyhledávací strom s hloubkou  $O(\log n)$ , mohlo by se stát, že slovo jako "mycophagist," s velmi malou pravděpodobností výskytu by se dostalo do kořene takového stromu.
  - Naopak slova s vysokou pravděpodobností výskytu jako „the“ by měli být blízko u kořene.
- Úloha: Sestavte optimální binární vyhledávací strom (BVS), když je dána množina klíčů  $K = \langle k_1, k_2, \dots, k_n \rangle$  (tak, že  $k_1 < k_2 < \dots < k_n$ ), ke každému klíči  $k_i$  známe pravděpodobnost výskytu  $p_i$ , navíc máme seznam prázdných klíčů  $d_0, d_1, d_2, \dots, d_n$ , které reprezentují intervaly mezi klíči, které nejsou ve slovníku  $K$ , ke každému prázdnému klíči  $d_j$  známe pravděpodobnost výskytu  $q_j$ .

Navíc platí  $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$ .

# Příklad: Optimální vyhledávací strom

- Příklad dvou vyhledávacích stromů s pěti klíči.



# Příklad: Optimální vyhledávací strom

## 1) Charakterizace struktury optimálního řešení.

$$\begin{aligned}\text{celková cena vyhledávání v } T &= \sum_{i=1}^n (\text{hloubka}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{hloubka}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{hloubka}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{hloubka}_T(d_i) \cdot q_i\end{aligned}$$

- Hledáme binární vyhledávací strom jehož *celková cena vyhledávání* (tj. cena přes všechny dotazy na úspěšné i neúspěšné vyhledání slova) bude minimální.

# Příklad: Optimální vyhledávací strom

2) Nalezení rekuzivní definice pro výpočet hodnoty optimálního řešení.

- Chceme umět řešit podúlohu pro klíče  $k_i, \dots, k_j$ , kde  $i \geq 1, j \leq n$  a  $j \geq i - 1$ . (Pokud  $j = i - 1$ , potom použijeme prázdný klíč  $d_{i-1}$ .)
- Očekávanou cenu vyhledávání ve stromu  $e$  si definujeme rekuzivně

$$e[i, j] = \begin{cases} q_{i-1} & \text{pokud } j = i - 1, \\ \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w(i, j)\} & \text{pokud } i \leq j. \end{cases}$$

$$w(i, j) = \begin{cases} q_{i-1} & \text{pokud } j = i - 1, \\ w(i, j - 1) + p_j + q_j & \text{pokud } i \leq j. \end{cases}$$

- $w(i, j)$  je pravděpodobnost výskytu podstromu  $i, j$ .

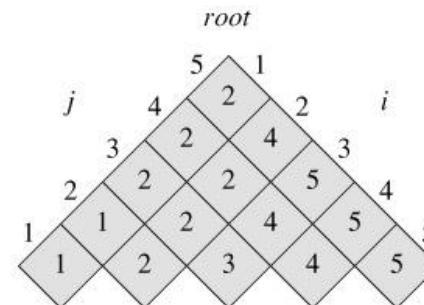
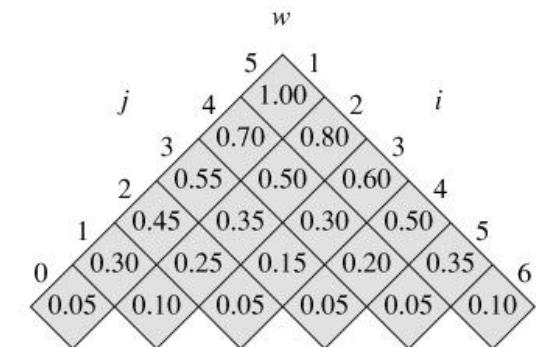
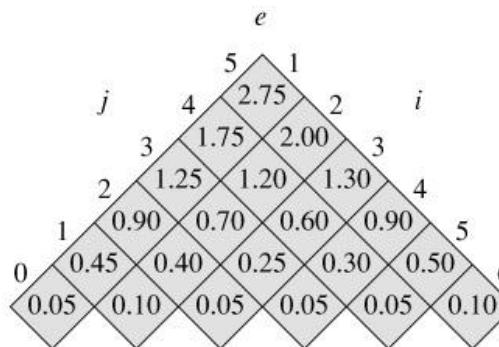
tedy  $w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i+1}^j q_l$ .

# Příklad: Optimální vyhledávací strom

3) Nalezení výpočtu hodnoty optimálního řešení zdola nahoru.

**Function** OPTIMÁLNÍ-BVS( $p, q, n$ )

```
1)   for  $i \leftarrow 1$  to  $n + 1$  do {  
2)      $e[i, i - 1] \leftarrow q_{i-1}$  ;  
3)      $w[i, i - 1] \leftarrow q_{i-1}$  ; }  
4)   for  $l \leftarrow 1$  to  $n$  do  
5)     for  $i \leftarrow 1$  to  $n - l + 1$  do {  
6)        $j \leftarrow i + l - 1$  ;  
7)        $e[i, j] \leftarrow \infty$  ;  
8)        $w[i, j] \leftarrow w[i, j - 1] + p_j + q_j$  ;  
9)       for  $r \leftarrow i$  to  $j$  do {  
10)          $t \leftarrow e[i, r - 1] + e[r + 1, j] + w[i, j]$  ;  
11)         if  $t < e[i, j]$   
12)           then {  $e[i, j] \leftarrow t$  ;  
13)              $root[i, j] \leftarrow r$  ; } } } }  
14)   return  $e$  and  $root$ 
```



# Příklad: Optimální vyhledávací strom

- Asymptotická složitost algoritmu:
  - Časová:  $O(n^3)$  a dokonce  $\Omega(n^3) \Rightarrow \Theta(n^3)$
  - Paměťová:  $\Theta(n^2)$

# Datové struktury a algoritmy

Část 11

Vyhledávání, zejména rozptylování

Petr Felkel

# Topics

## Vyhledávání

### Rozptylování (hashing)

- Rozptylovací funkce
- Řešení kolizí
  - Zřetězené rozptylování
  - Otevřené rozptylování
    - Linear Probing
    - Double hashing

# Slovník - Dictionary

Řada aplikací potřebuje

- dynamickou množinu
  - s operacemi: Search, Insert, Delete
- = **slovník**

Př. Tabulka symbolů překladače

| identifikátor | typ | adresa     |
|---------------|-----|------------|
| suma          | int | 0xFFFFDC09 |
| ...           | ... | ...        |

# Vyhledávání

Porovnáním klíčů

$\Omega(\log n)$

- Nalezeno, když klíč\_prvku = hledaný klíč
- např. sekvenční vyhledávání, BVS,...

Indexováním klíčem (přímý přístup)

$\Theta(1)$

- klíč je přímo indexem (adresou)
- rozsah klíčů ~ rozsahu indexů

Rozptylováním

průměrně  $\Theta(1)$

- výpočtem adresy z hodnoty klíče

asociativní

adresní vyhledávání

# Rozptylování - Hashing

- = kompromis mezi rychlostí a spotřebou paměti
  - ∞ času              - sekvenční vyhledávání
  - ∞ paměti          - přímý přístup  
(indexování klíčem)
  - málo času i paměti
    - hashing
    - velikost tabulky reguluje čas vyhledání

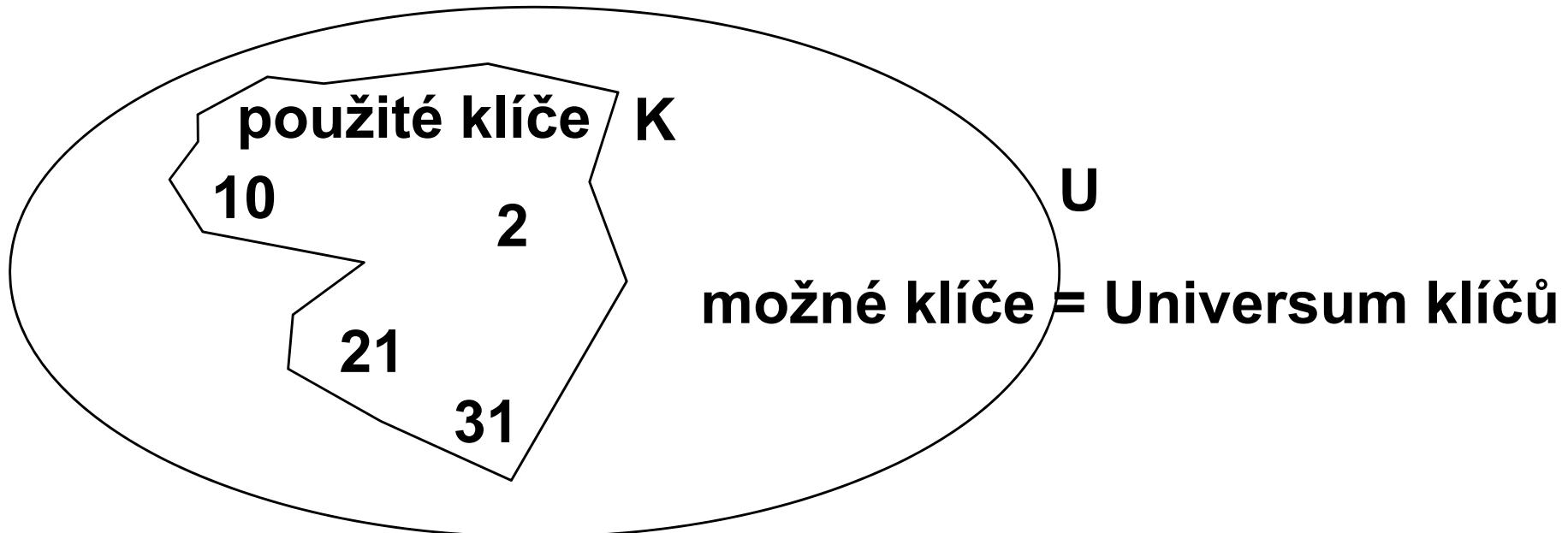
# Rozptylování - Hashing

Konstantní očekávaný čas pro *vyhledání a vkládání*  
*(search and insert) !!!*

Něco za něco:

- čas provádění ~ délce klíče
- není vhodné pro operace *výběru podmnožiny a řazení (select a sort)*

# Rozptylování

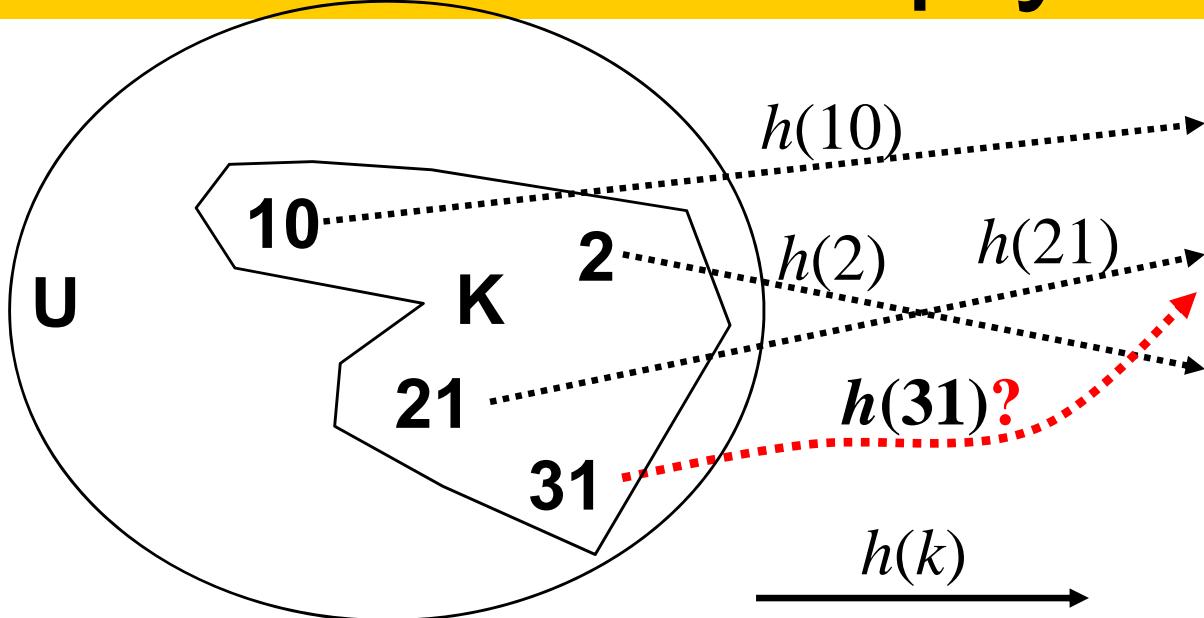


Rozptylování vhodné pro  $|K| \ll |U|$

**K** množina použitých klíčů

**U** universum klíčů

# Rozptylování



Dvě fáze

1. Výpočet rozptylovací funkce  $h(k)$   
( $h(k)$  vypočítá adresu z hodnoty klíče)
  2. Vyřešení kolizí
- $h(31)$  ..... **kolize**: index 1 již obsazen

# 1. Výpočet rozptylovací funkce $h(k)$

# Rozptylovací funkce $h(k)$

Zobrazuje

množinu klíčů  $K \in U$

do intervalu adres  $A = \langle a_{min}, a_{max} \rangle$ , obvykle  $\langle 0, M-1 \rangle$

$|U| \gg |K| \approx |A|$

( $h(k)$  Vypočítá adresu z hodnoty klíče)

**Synonyma:**  $k_1 \neq k_2$ ,  $h(k_1) = h(k_2)$   
= **kolize**

# Rozptylovací funkce $h(k)$

Je silně závislá na vlastnostech klíčů a jejich reprezentaci v paměti

Ideálně:

- výpočetně co nejjednodušší (rychlá)
- approximuje náhodnou funkci
- využije **rovnoměrně** adresní prostor
- generuje **minimum kolizí**
- proto: využívá všechny složky klíče

# Rozptylovací funkce $h(k)$ - příklady

Příklady fce  $h(k)$  pro různé typy klíčů

- reálná čísla
- celá čísla
- bitová
- řetězce

Chybná rozptylovací funkce

# Rozptylovací funkce $h(k)$ -příklady

Pro **reálná čísla** z intervalu  $<0, 1>$

- multiplikativní:  $h(k, M) = \text{round}( k * M )$   
neoddělí shluky blízkých čísel (s rozdílem  $< 1/M$ )  
 $M$  = velikost tabulky (table size)

# Rozptylovací funkce $h(k)$ -příklady

## Pro celá čísla

- multiplikativní: (kde  $M$  je prvočíslo, klíče mají  $w$  bitů)
  - $h(k, M) = \text{round}( k / 2^w * M )$
- modulární:
  - $h(k, M) = k \% M$
- kombinovaná:
  - $h(k, M) = \text{round}( c * k ) \% M, \quad c \in <0,1>$
  - $h(k, M) = (\text{int})(0.616161 * k) \% M$
  - $h(k, M) = (16161 * k) \% M \quad // \text{pozor na přetečení}$

# Rozptylovací funkce $h(k)$ -příklady

## Hash functions $h(k)$ - examples

# Rychlá, silně závislá na reprezentaci klíčů

# je totéž jako

$h(k) = k \% M$ , tj. použije x nejnižších bitů klíče

# Rozptylovací funkce $h(k)$ -příklady

Pro řetězce (*for strings*):

```
int hash( char *k, int M ) {  
    int h = 0, a = 127;  
    for( ; *k != 0; k++ )  
        h = ( a * h + *k ) % M;  
    return h;  
}
```

**Hornerovo schéma :**

$$\begin{aligned} P(a) &= k_4 * a^4 + k_3 * a^3 + k_2 * a^2 + k_1 * a^1 + k_0 * a^0 \\ &= (((k_4 * a + k_3) * a + k_2) * a + k_1) * a + k_0 \end{aligned}$$

Výpočet hodnoty polynomu  $P$  v bodě  $a$ , koeficienty  $P$  jsou jednotlivé znaky (jejich číselná hodnota) v řetězci  $*k$ .

# Rozptylovací funkce $h(k)$ -příklady

Pro řetězce (*for strings*) Java:

```
public int hashCode( String s, int M ) {  
    int h = 0;  
    for( int i = 0; i < s.length(); i++ )  
        h = 31 * h + s.charAt(i);  
    return h;  
}
```

Hodnota konstant **127**, **31** přispívá rovnoměrnému psoudonáhodnému rozptylení.

# Rozptylovací funkce $h(k)$ -příklady

Pro řetězce: (pseudo-) randomizovaná

```
int hash( char *k, int M )
{ int h = 0, a = 31415; b = 27183;
  for( ; *k != 0; k++, a = a*b % (M-1) )
    h = ( a * h + *k ) % M;
  return h;
}
```

# Rozptylovací funkce $h(k)$ -chyba

Častá chyba:

funkce vrací stále nebo většinou stejnou hodnotu

- chyba v konverzi typů
- funguje, ale vrací blízké adresy
- proto generuje hodně kolizí

=> aplikace je extrémně pomalá, řešení kolizí zdržuje.

# Shrnutí

## Rozptylovací funkce $h(k)$

- počítá adresu z hodnoty klíče

# Rozptylovací funkce $h(k)$

Každá hashovací funkce má slabá místa, kdy pro různé klíče dává stejnou adresu

## Univerzální hashování

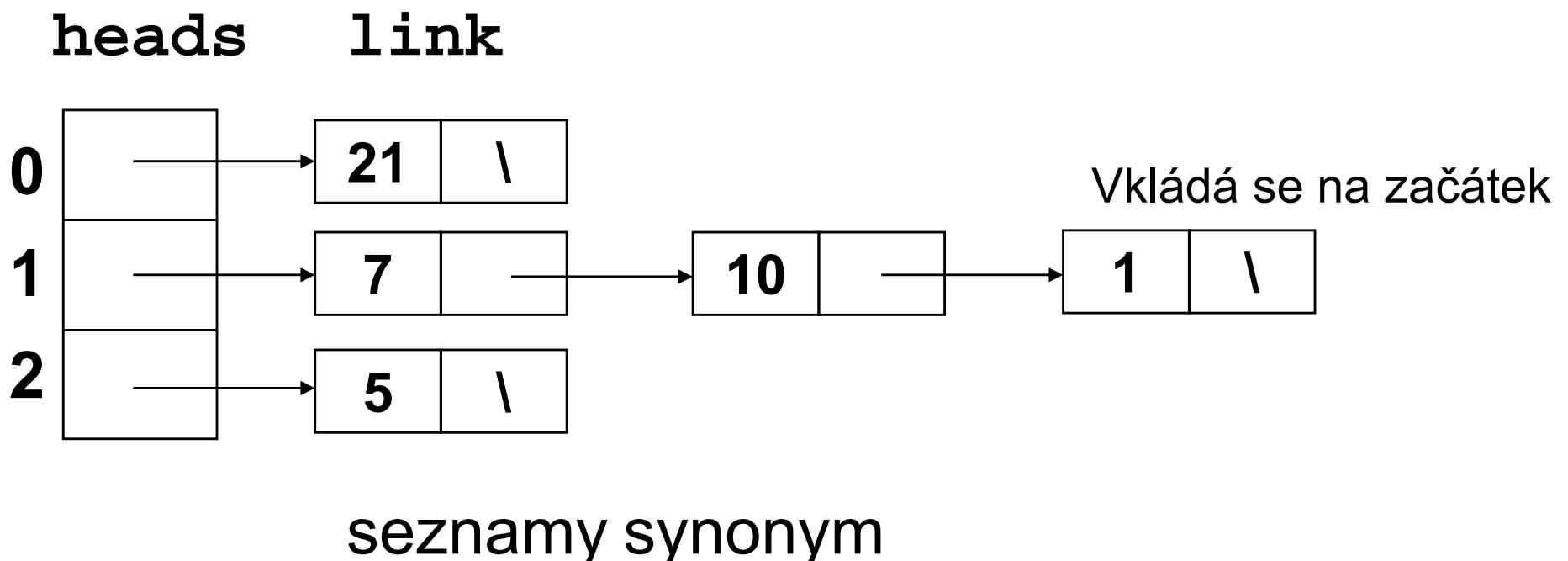
- Místo jedné hashovací funkce  $h(k)$  máme konečnou množinu  $H$  funkcí mapujících  $U$  do intervalu  $\{0, 1, \dots, m-1\}$
- Při spuštění programu jednu náhodně zvolíme
- Tato množina je univerzální, pokud pro různé klíče  $x, y \in U$  vrací stejnou adresu  $h(x) = h(y)$  přesně v  $|H|/m$  případech
- Pravděpodobnost kolize při náhodném výběru funkce  $h(k)$  je tedy přesně  $1/m$

## 2. Vyřešení kolizí

# a) Zřetězené rozptylování 1/5 Chaining

$$h(k) = k \bmod 3$$

posloupnost : 1, 5, 21, 10, 7



# a) Zřetězené rozptylování 2/5

```
private:  
    link* heads; int N,M; [Sedgewick]  
  
public:  
    init( int maxN )           // initialization  
    {  
        N=0;                  // No. of nodes  
        M = maxN / 5;          // table size  
        heads = new link[M]; // table with pointers  
        for( int i = 0; i < M; i++ )  
            heads[i] = null;  
    }  
    ...
```

# a) Zřetězené rozptylování 3/5

```
Item search( Key k )
{
    return searchList( heads[hash(k, M)], k );
}

void insert( Item item )           // Vkládá se na začátek
{
    int i = hash( item.key(), M );
    heads[i] = new node( item, heads[i] );
    N++;
}
```

# a) Zřetězené rozptylování 4/5

$n$  = počet prvků,  $m$  = velikost tabulky,  $m < n$ .

Řetěz synonym má ideálně délku  $\alpha = n/m$ ,  $\alpha > 1$  (plnění tabulky)

velmi nepravděpodobný

Insert  $I(n) = t_{\text{hash}} + t_{\text{link}} = O(1)$  extrém

Search  $Q(n) = t_{\text{hash}} + t_{\text{search}}$   
 $= t_{\text{hash}} + t_c * n/(2m) = O(n)$  průměrně  
 $O(1 + \alpha)$

Delete  $D(n) = t_{\text{hash}} + t_{\text{search}} + t_{\text{link}} = O(n)$   $O(1 + \alpha)$

pro malá  $\alpha$  (velká  $m$ ) se hodně blíží  $O(1)$  !!!

pro velká  $\alpha$  (malá  $m$ )  $m$ -násobné zrychlení vůči sekvenčnímu  
hledání.

# a) Zřetězené rozptylování 5/5

**Praxe: volit  $m = n/5$  až  $n/10 \Rightarrow$  plnění  $\alpha = 10$  prvků / řetěz**

- vyplatí se hledání sekvenčně (je krátké)
- neplýtvá nepoužitými ukazateli

Shrnutí:

- + nemusíme znát  $n$  předem
- potřebuje dynamické přidělování paměti
- potřebuje paměť na ukazatele a na tabulku[ $m$ ]

## b) Otevřené rozptylování (open-address hashing)

Známe předem počet prvků (odhad)  
nechceme ukazatele (v prvcích ani tabulku)  
=> posloupnost do pole

Podle tvaru hashovací funkce  $h(k)$  při kolizi:

1. lineární prohledávání (linear probing)
2. dvojí rozptylování (double hashing)

|   |    |
|---|----|
| 0 | 5  |
| 1 | 1  |
| 2 | 21 |
| 3 | 10 |
| 4 |    |

## b) Otevřené rozptylování

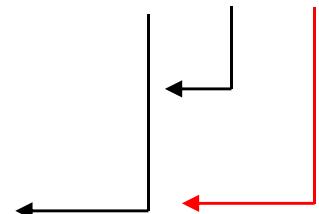
$$h(k) = k \bmod 5$$

posloupnost:

$$(h(k) = k \bmod m, m \text{ je rozměr pole})$$

1, 5, **21**, 10, 7

|   |   |
|---|---|
| 0 | 5 |
| 1 | 1 |
| 2 |   |
| 3 |   |
| 4 |   |



### Problém:

kolize - 1 blokuje místo pro 21

1. linear probing
2. double hashing

Pozn.: 1 a 21 jsou synonyma

často ale blokuje nesynonymum.

Kolize je blokování libovolným klíčem

# Test - Probe

= určení, zda pozice v tabulce obsahuje klíč shodný s hledaným klíčem

- search hit = klíč nalezen
- search miss = pozice prázdná, klíč nenalezen
- Jinak = na pozici je jiný klíč, hledej dál

## b) Otevřené rozptylování (open-addressing hashing)

Metoda řešení kolizí  
(solution of collisions)

b1) Linear probing

Lineární prohledávání

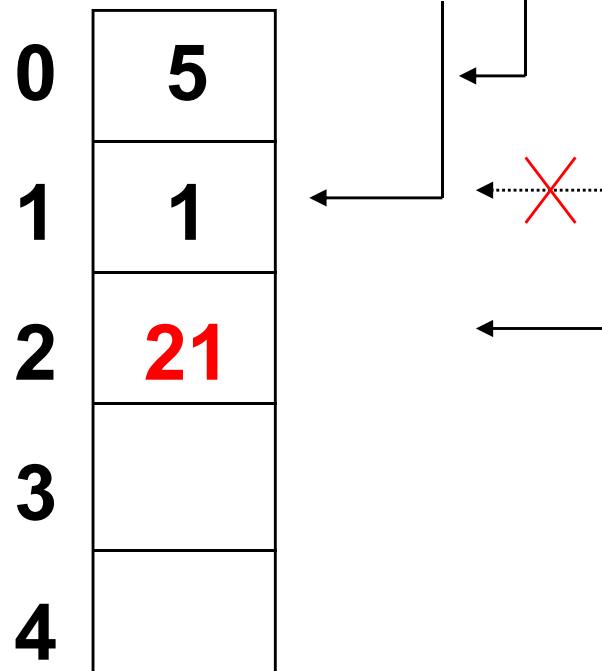
b2) Double hashing

Dvojí rozptylování

# b1) Linear probing

$$h(k) = [(k \bmod 5) + i] \bmod 5 = (k + i) \bmod 5; i = 0;$$

posloupnost: 1, 5, 21, 10, 7



kolize - 1 blokuje

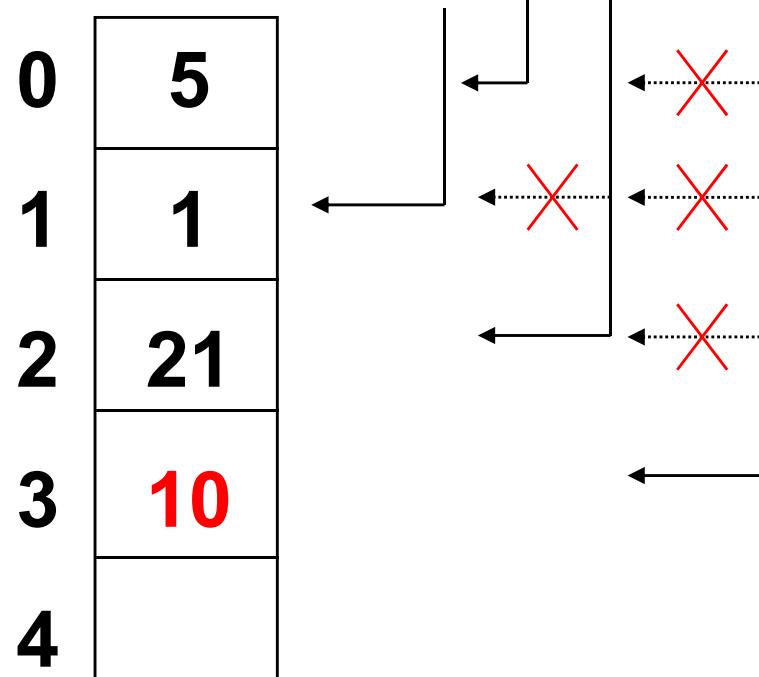
=> 1. linear probing

vlož o 1 pozici dál ( $i++ \Rightarrow i = 1$ )

# b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

posloupnost: 1, 5, 21, **10**, 7

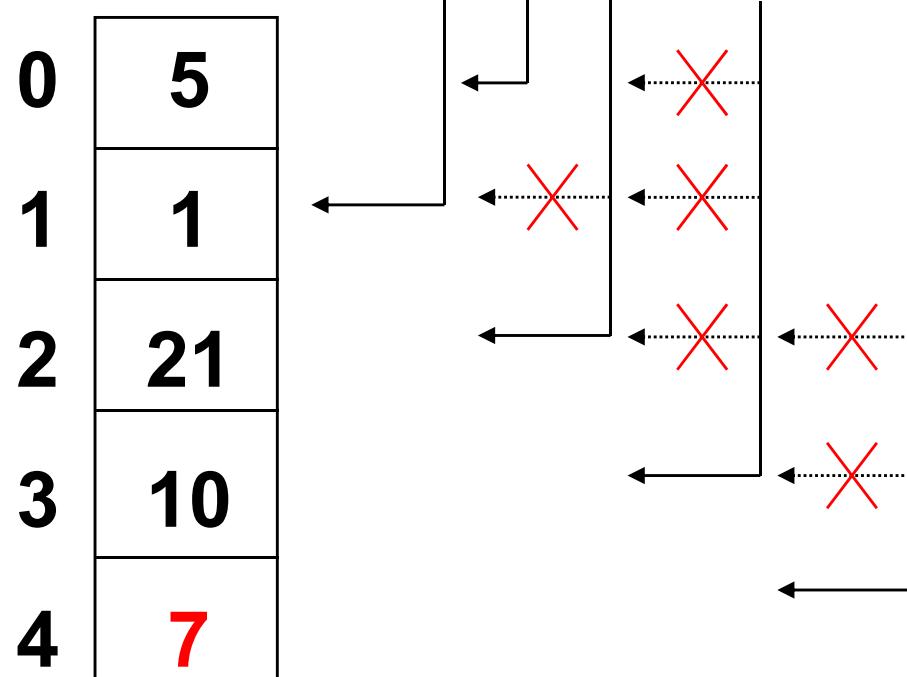


1. kolize - 5 blokuje - vlož dál
  2. kolize - 1 blokuje - vlož dál
  3. kolize - 21 blokuje - vlož dál
- vloženo o 3 pozice dál ( $i = 3$ )

# b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

posloupnost: 1, 5, 21, 10, 7



1. kolize - vlož dál ( $i++$ )
2. kolize - vlož dál ( $i++$ )
- vlož o 2 pozice dál ( $i = 2$ )

# b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

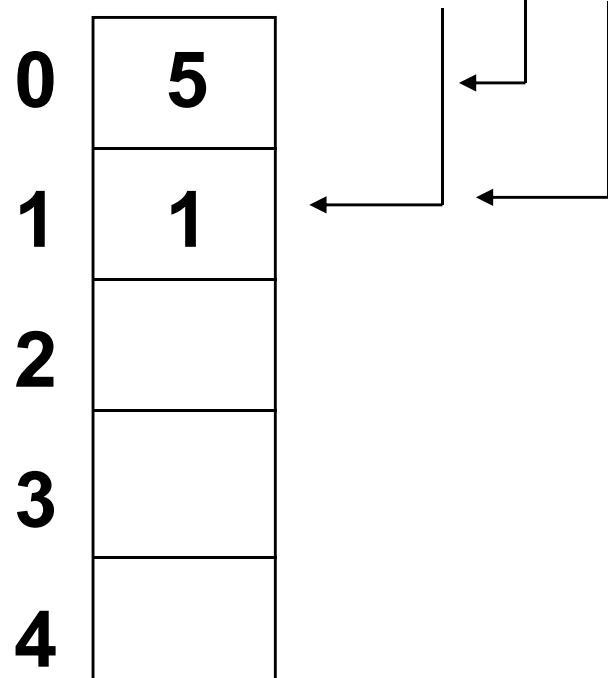
posloupnost: 1, 5, 21, 10, 7

|   |    |       |
|---|----|-------|
| 0 | 5  | i = 0 |
| 1 | 1  | i = 0 |
| 2 | 21 | i = 1 |
| 3 | 10 | i = 3 |
| 4 | 7  | i = 2 |

# b1) Linear probing

$$h(k) = k \bmod 5$$

posloupnost: 1, 5, **21**, 10, 7



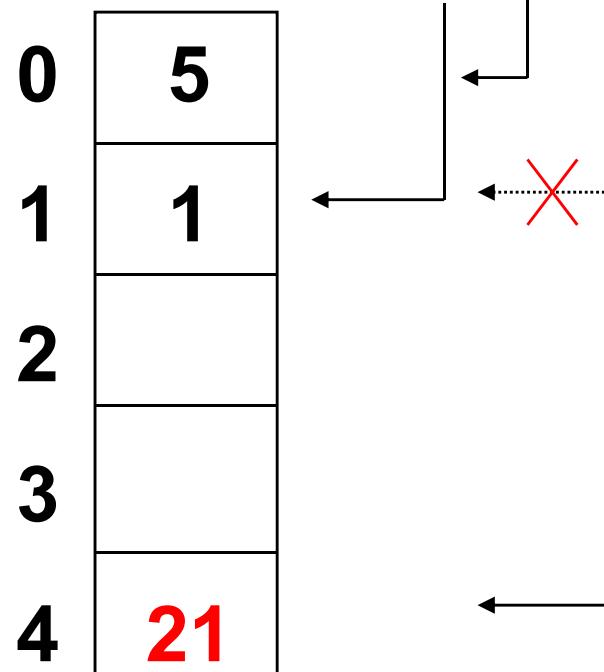
kolize - 1 blokuje (collision-blocks)

# b1) Linear probing

$$h(k) = [(k \bmod 5) + i.\text{const}] \bmod 5, \quad h(k) = (k + i \cdot 3) \bmod 5$$

posloupnost: 1, 5, **21**, 10, 7

stačí prvočíslo  $\neq m$   
nebo číslo nesoudělné s m



kolize - 1 blokuje

(collision blocks)

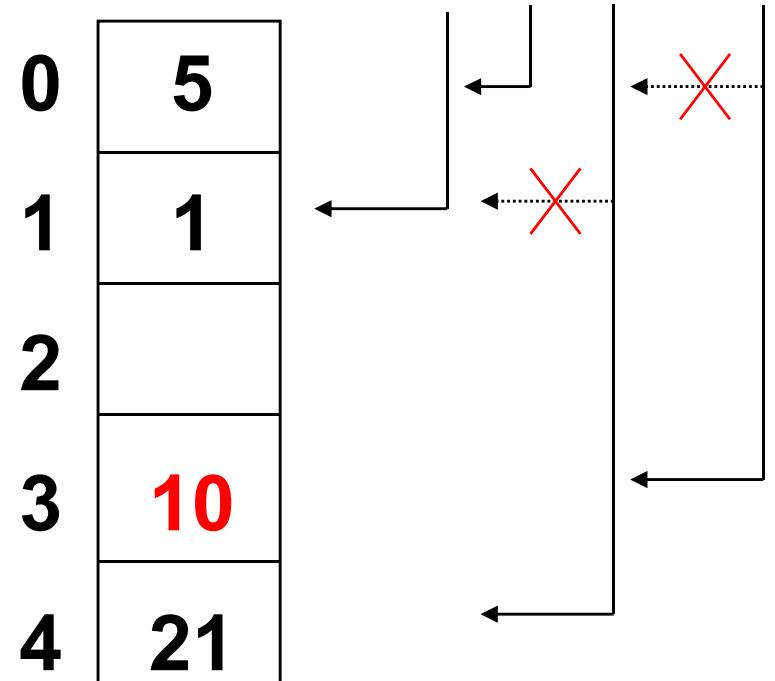
vlož o 3 pozice dál ( $i++ \Rightarrow i = 1$ )

(i je číslo pokusu)

# b1) Linear probing

$$h(k) = (k + i \cdot 3) \bmod 5$$

posloupnost: 1, 5, 21, **10**, 7



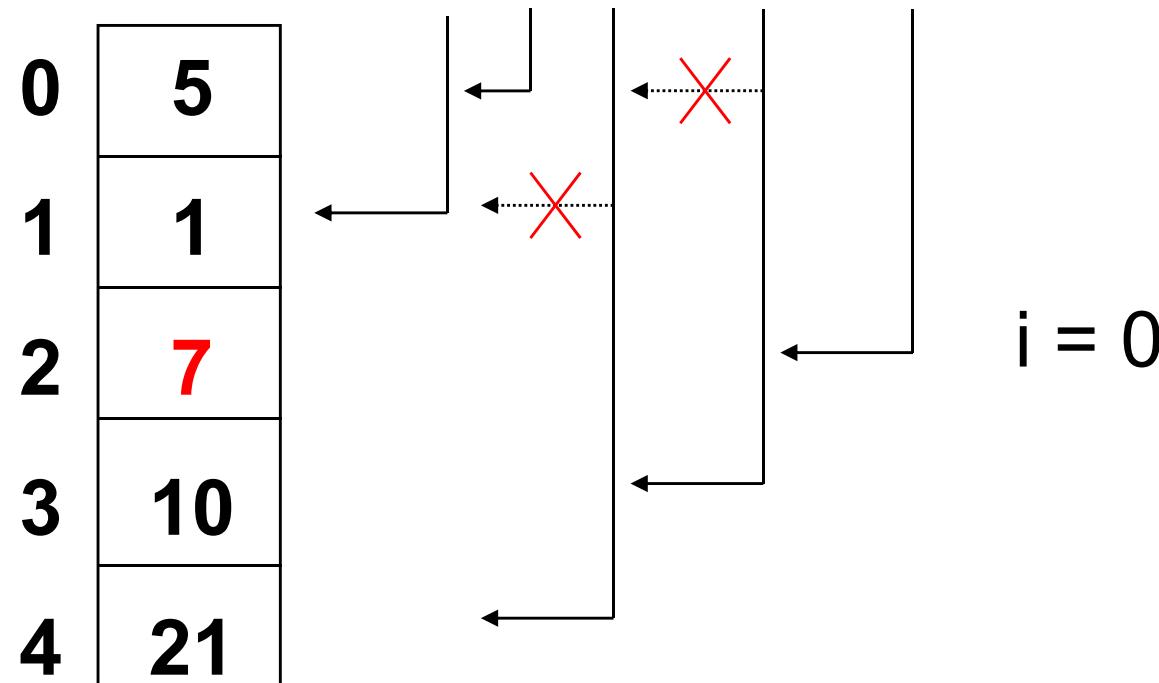
kolize - 5 blokuje - vlož dál

(vlož o 3 pozice dál ( $i = 1$ )

# b1) Linear probing

$$h(k) = (k + i \cdot 3) \bmod 5$$

posloupnost: 1, 5, 21, 10, 7



# b1) Linear probing

$$h(k) = (k + i \cdot 3) \bmod 5$$

posloupnost: 1, 5, 21, 10, 7

|   |    |       |
|---|----|-------|
| 0 | 5  | i = 0 |
| 1 | 1  | i = 0 |
| 2 | 7  | i = 0 |
| 3 | 10 | i = 1 |
| 4 | 21 | i = 1 |

# b1) Linear probing

$$h(k) = (k + i) \bmod 5$$

|   |    |         |
|---|----|---------|
| 0 | 5  | i = 0   |
| 1 | 1  | i = 0   |
| 2 | 21 | i = 1   |
| 3 | 10 | i = 3 ! |
| 4 | 7  | i = 2   |

hrozí dlouhé shluky  
(long clusters)

$$h(k) = (k + i \cdot 3) \bmod 5$$

|   |    |       |
|---|----|-------|
| 0 | 5  | i = 0 |
| 1 | 1  | i = 0 |
| 2 | 7  | i = 0 |
| 3 | 10 | i = 1 |
| 4 | 21 | i = 1 |

vhodná volba posunu  
 $i \cdot 3$  je věcí náhody

# b1) Linear probing

```
private:  
    Item *ht; int N,M; [Sedgewick]  
    Item nullItem;  
public:  
    init( int maxN )           // initialization  
    {  
        N=0;                  // Number of stored items  
        M = 2*maxN;            // load_factor < 1/2  
        ht = new Item[M];  
        for( int i = 0; i < M; i++ )  
            ht[i] = nullItem;  
    }...
```

## b1) Linear probing

```
void insert( Item item )
{
    int i = hash( item.key(), M );

    while( !ht[i].null() )
        i = (i+const) % M; // Linear probing

    ht[i] = item;
    N++;
}
```

# b1) Linear probing

```
Item search( Key k )
{
    int i = hash( k, M );

    while( !ht[i].null() ) { // !cluster end
                           // zarážka (sentinel)
        if( k == ht[i].key() )
            return ht[i];
        else
            i = (i+const) % M; // Linear probing
    }
    return nullItem;
}
```

## b) Otevřené rozptylování (open-addressing hashing)

Metoda řešení kolizí  
(solution of collisions)

b1) Linear probing

Lineární prohledávání

b2) Double hashing

Dvojí rozptylování

## b2) Double hashing

Hash function  $h(k) = [h_1(k) + i \cdot h_2(k)] \bmod m$

$$\begin{aligned} h_1(k) &= k \bmod m && // \text{initial position} \\ h_2(k) &= 1 + (k \bmod m') && // \text{offset} \end{aligned} \quad \left. \begin{array}{l} \\ \end{array} \right\} \begin{array}{l} \text{Both depend on } k \\ \Rightarrow \end{array}$$

$m$  = prime number or  $m$  = power of 2

$m'$  = slightly less  $m'$  = odd

If  $d$  = greatest common divisor  $\Rightarrow$  search  $m/d$  slots only

Each key has  
different  
probe sequence

Ex:  $k = 123456, m = 701, m' = 700$

$h_1(k) = 80, h_2(k) = 257$  Starts at 80, and every 257 % 701

## b2) Double hashing

```
void insert( Item item )
{
    Key k = item.key();
    int i = hash( k, M ),
        j = hashTwo( k, M );// different for  $k_1 \neq k_2$ 

    while( !ht[i].null() )
        i = (i+j) % M; //Double Hashing

    ht[i] = item; N++;
}
```

## b2) Double hashing

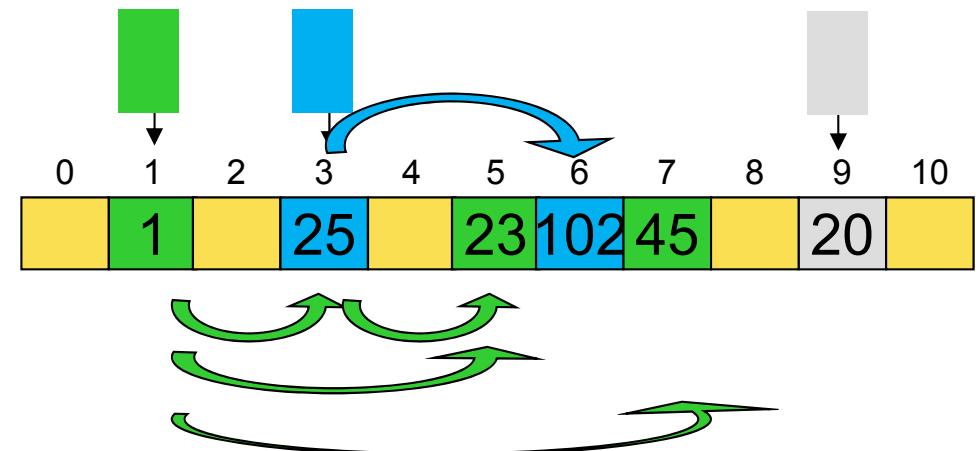
```
Item search( Key k )
{
    int i = hash( k, M ),
        j = hashTwo( k, M ); // different for  $k_1 \neq k_2$ 

    while( !ht[i].null() )
    {
        if( k == ht[i].key() )
            return ht[i];
        else
            i = (i+j) % M; // Double Hashing
    }
    return nullItem;
}
```

# Double hashing - example

b2) Double hashing  $h(k) = [h_1(k) + i.h_2(k)] \bmod m$

| Input | $h_1(k) = k \% 11$ | $h_2(k) = 1 + k \% 10$ | $i$ | $h(k)$ |
|-------|--------------------|------------------------|-----|--------|
| 1     | 1                  | 2                      | 0   | 1      |
| 25    | 3                  | 6                      | 0   | 3      |
| 23    | 1                  | 4                      | 0,1 | 1,5    |
| 45    | 1                  | 6                      | 0,1 | 1,7    |
| 102   | 3                  | 3                      | 0,1 | 3,6    |
| 20    | 9                  | 1                      | 0   | 9      |
|       |                    |                        |     |        |



$$h_1(k) = k \% 11$$

$$h_2(k) = 1 + (k \% 10)$$

## b) Otevřené rozptylování (open-addressing hashing)

$\alpha$  = plnění tabulky (*load factor of the table*)

$\alpha = n/m$ ,  $\alpha \in \langle 0, 1 \rangle$

$n$  = počet prvků (*number of items in the table*)

$m$  = velikost tabulky,  $m > n$  (*table size*)

# b) Otevřené rozptylování (open-addressing hashing)

Expected number of probes

Linear probing:

|             |                                |       |
|-------------|--------------------------------|-------|
| Search hits | $0.5 ( 1 + 1 / (1 - \alpha) )$ | found |
|-------------|--------------------------------|-------|

|               |                                  |           |
|---------------|----------------------------------|-----------|
| Search misses | $0.5 ( 1 + 1 / (1 - \alpha)^2 )$ | not found |
|---------------|----------------------------------|-----------|

Double hashing:

|             |                                         |
|-------------|-----------------------------------------|
| Search hits | $(1 / \alpha) \ln ( 1 / (1 - \alpha) )$ |
|-------------|-----------------------------------------|

|               |                    |
|---------------|--------------------|
| Search misses | $1 / (1 - \alpha)$ |
|---------------|--------------------|

$$\alpha = n/m, \alpha \in \langle 0, 1 \rangle$$

# b) Očekávaný počet testů

Linear probing:

| Plnění $\alpha$ | 1/2 | 2/3 | 3/4 | 9/10 |
|-----------------|-----|-----|-----|------|
| Search hit      | 1.5 | 2.0 | 3.0 | 5.5  |
| Search miss     | 2.5 | 5.0 | 8.5 | 55.5 |

Double hashing:

| Plnění $\alpha$ | 1/2 | 2/3 | 3/4 | 9/10 |
|-----------------|-----|-----|-----|------|
| Search hit      | 1.4 | 1.6 | 1.8 | 2.6  |
| Search miss     | 2.0 | 3.0 | 4.0 | 10.0 |

Tabulka může být více zaplněná než začne klesat výkonnost.  
K dosažení stejného výkonu stačí menší tabulka.

# References

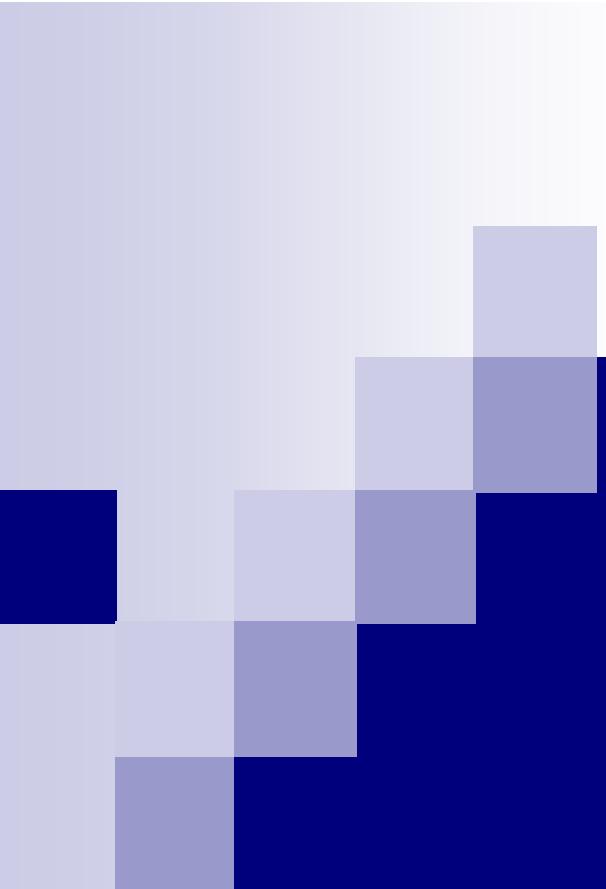
[Cormen]

Cormen, Leiserson, Rivest: Introduction to Algorithms,  
Chapter 12, McGraw Hill, 1990

or better:

[CLRS]

Cormen, Leiserson, Rivest, Stein: Introduction to  
Algorithms, third edition, Chapter 11, MIT press, 2009

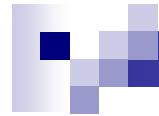


# Algoritmizace

Hashing II

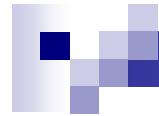
Jiří Vyskočil, Marko Genyg-Berezovskyj

2010



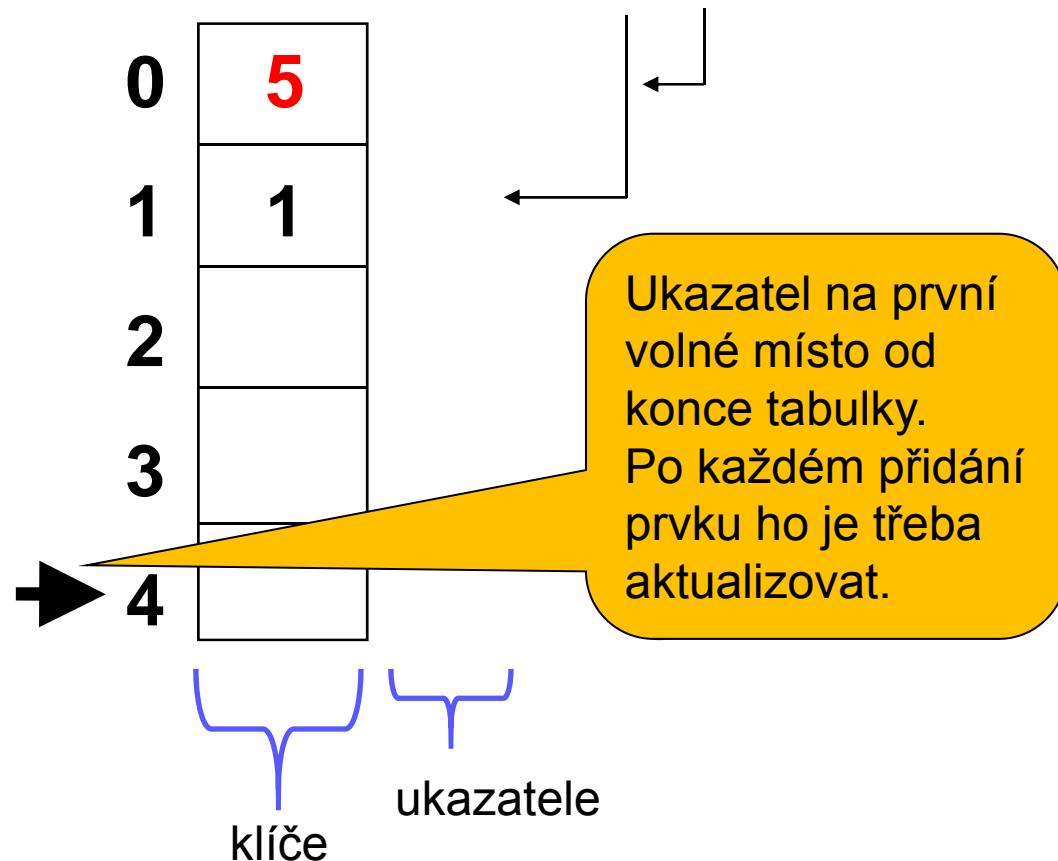
# Srůstající hashování (coalesced hashing)

- Znám předem počet prvků (odhad)
- Z důvodů efektivity nechci ukazatele (mezi prvky). Na jednu pozici tabulky připadne právě jeden ukazatel.
- => kolizní prvky (synonyma) se musejí „nějak“ ukládat přímo do hashovací tabulky. Ukazatele v tabulce umožňují procházet pouze prvky jedné skupiny. Nedochází zde k tak velkému propojování do clusterů jako v případě otevřeného hashování. Dochází zde pouze k tzv. srůstání.
  - standardní srůstající hashování
    - LISCH (late insert standard coalesced hashing)
    - EISCH (early insert standard coalesced hashing)
  - srůstající hashování s pomocnou pamětí
    - LICH (late insert coalesced hashing)
    - EICH (early insert coalesced hashing)
    - VICH (variable insert coalesced hashing)



# Standardní srůstající hashování - LISCH

- $h(k) = k \bmod 5$
- posloupnost: 1, 5, 21, 10, 15

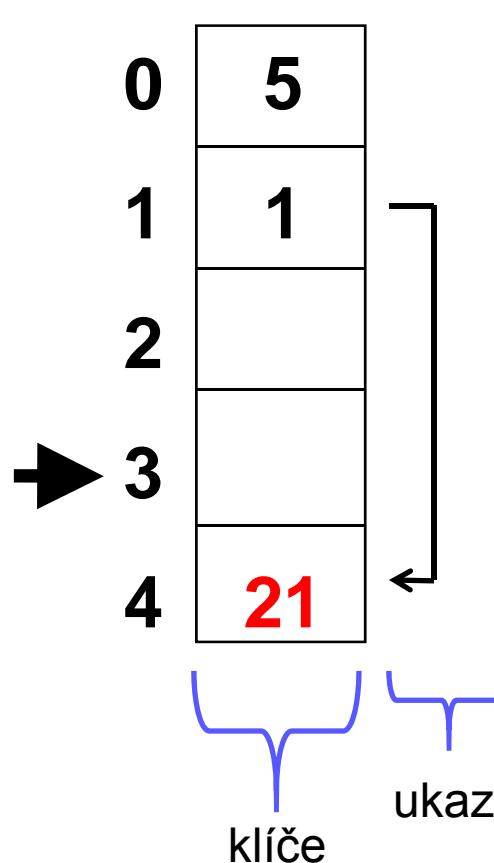


Postup:

1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce na poslední místo.

# Standardní srůstající hashování - LISCH

- $h(k) = k \bmod 5$
- posloupnost:      1, 5, **21**, 10, 15



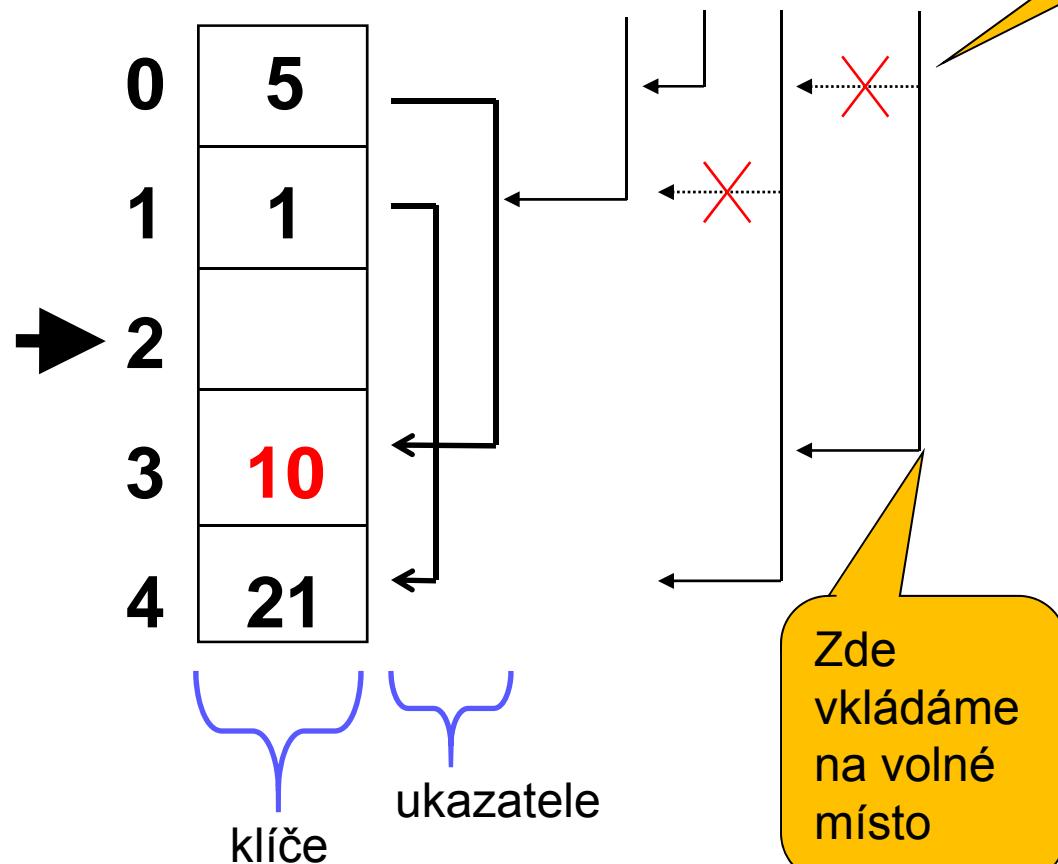
Postup:

1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce na poslední místo.

# Standardní srůstající hashování - LISCH

- $h(k) = k \bmod 5$

- posloupnost:      1, 5, 21, **10**,    15



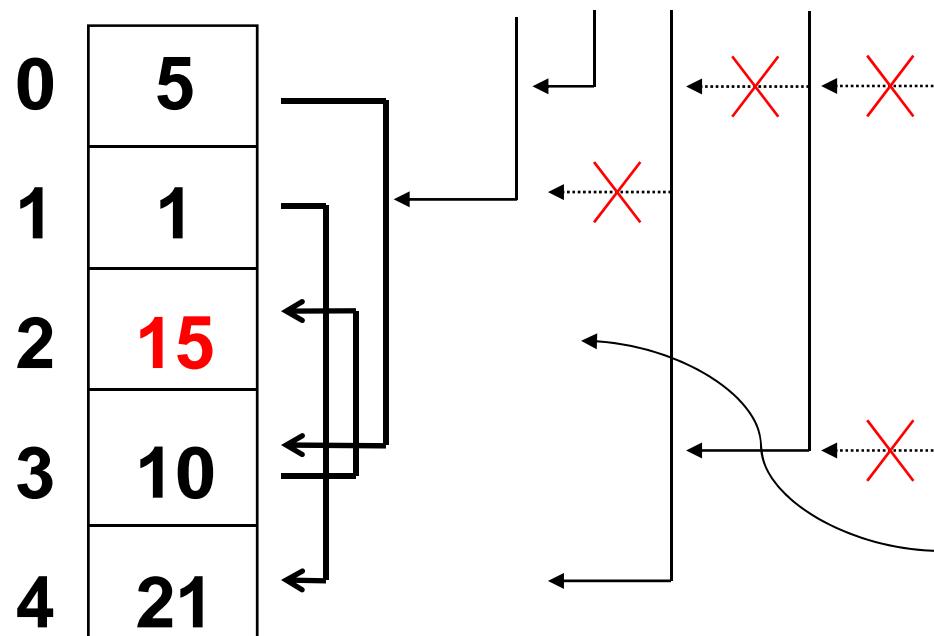
Zde procházíme řetězec prvků.

Postup:

1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce na poslední místo.

# Standardní srůstající hashování - LISCH

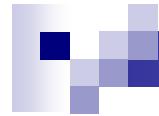
- $h(k) = k \bmod 5$
- posloupnost:      1, 5, 21, 10,    15



Postup:

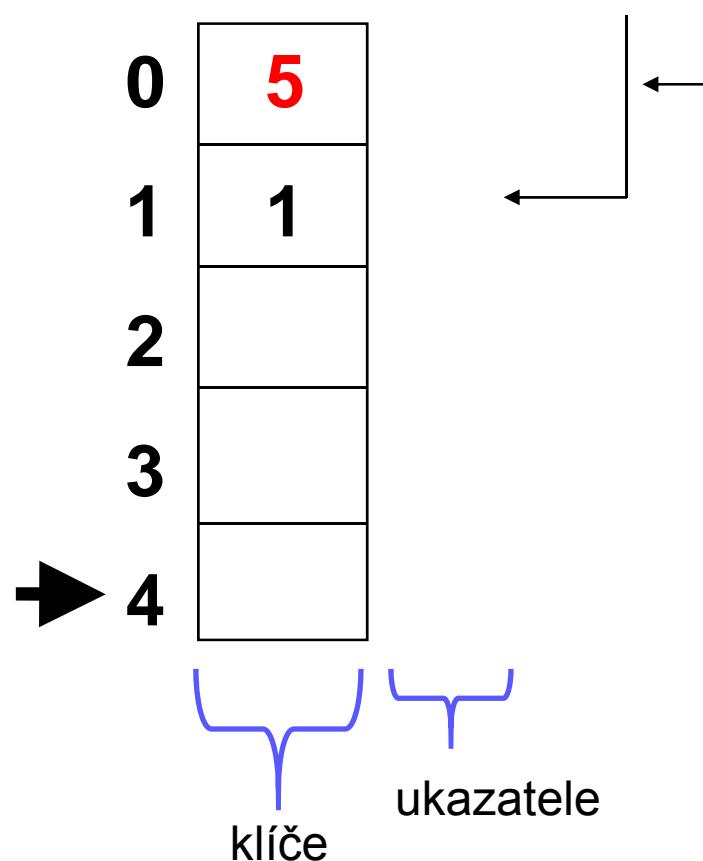
1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce na poslední místo.

→ Tabulka je zaplněna.



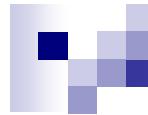
# Standardní srůstající hashování - EISCH

- $h(k) = k \bmod 5$
- posloupnost: 1, 5, 21, 10, 15



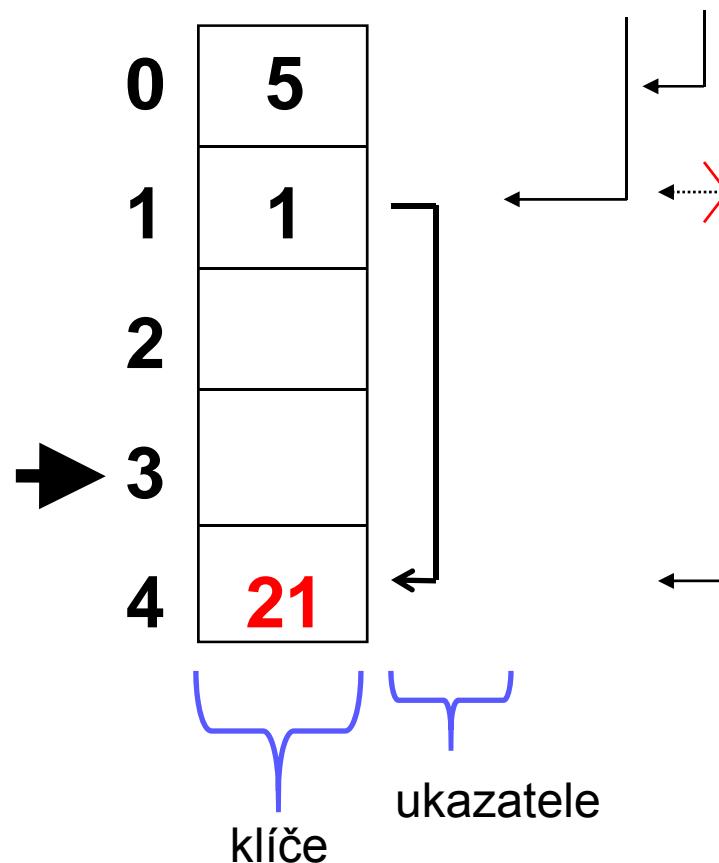
Postup:

1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce za 1. místo.



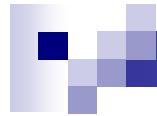
# Standardní srůstající hashování - EISCH

- $h(k) = k \bmod 5$
- posloupnost:      1, 5, **21**, 10, 15



Postup:

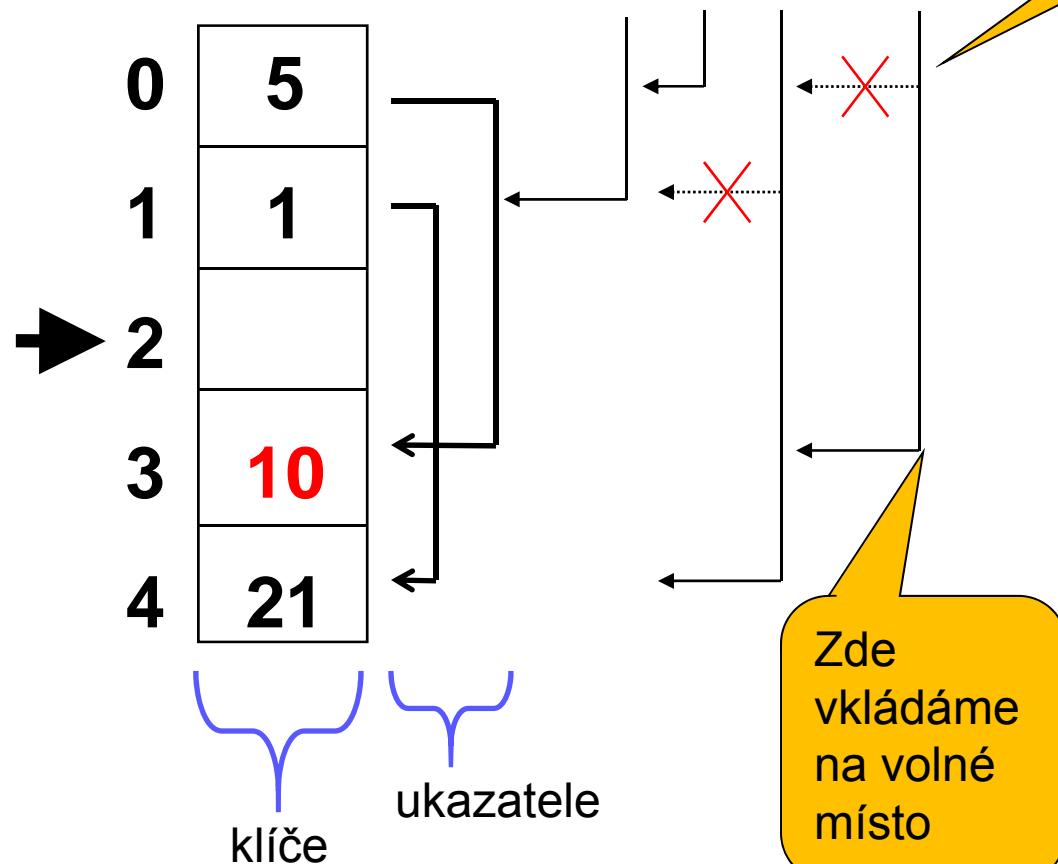
1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce za 1. místo.



# Standardní srůstající hashování - EISCH

- $h(k) = k \bmod 5$

- posloupnost:      1, 5, 21, **10**,    15

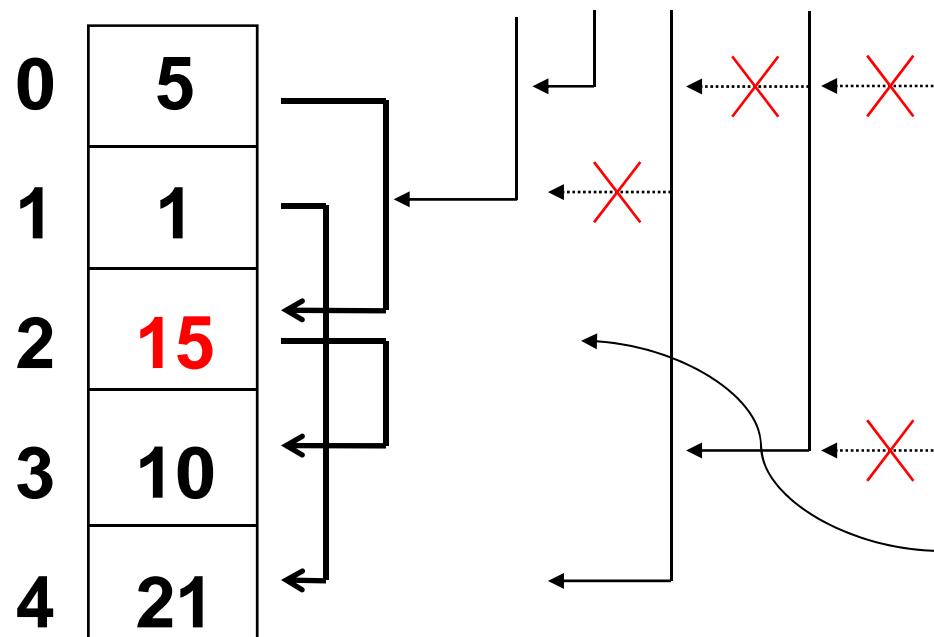


Postup:

1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce za 1. místo.

# Standardní srůstající hashování - EISCH

- $h(k) = k \bmod 5$
- posloupnost: 1, 5, 21, 10, 15



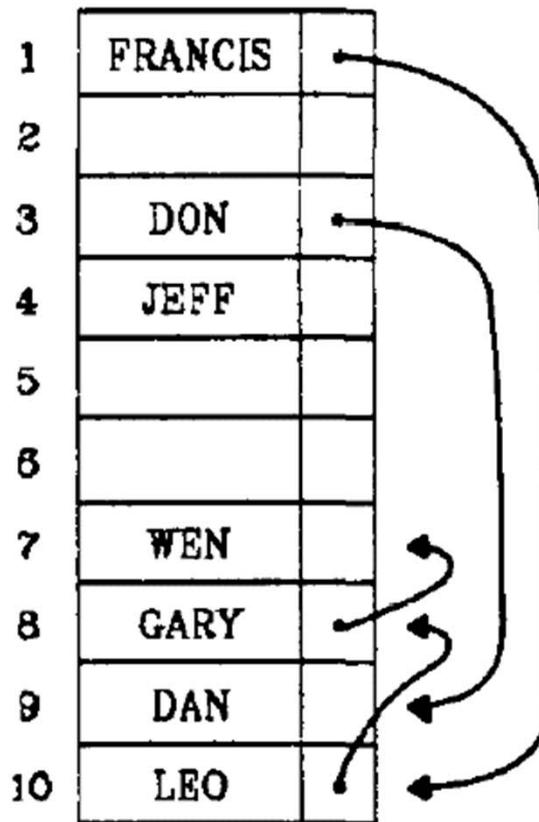
Postup:

1.  $i = h(k);$
2. Prohledej řetězec začínající na místě  $i$  a pokud nenajdeš  $k$ , přidej ho do tabulky na první volné místo od konce tabulky a připoj ho do řetězce za 1. místo.

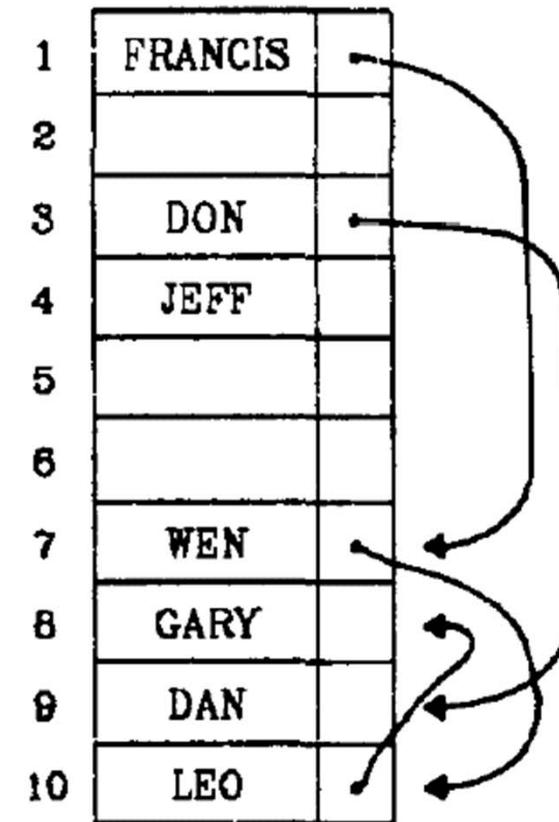
→ Tabulka je zaplněna.

# Standardní srůstající hashování – LISCH, EISCH

(a) LISCH



(b) EISCH

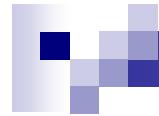


Keys:

FRANCIS    DON    LEO    JEFF    DAN    GARY    WEN

Hash Addresses: (a)(b)

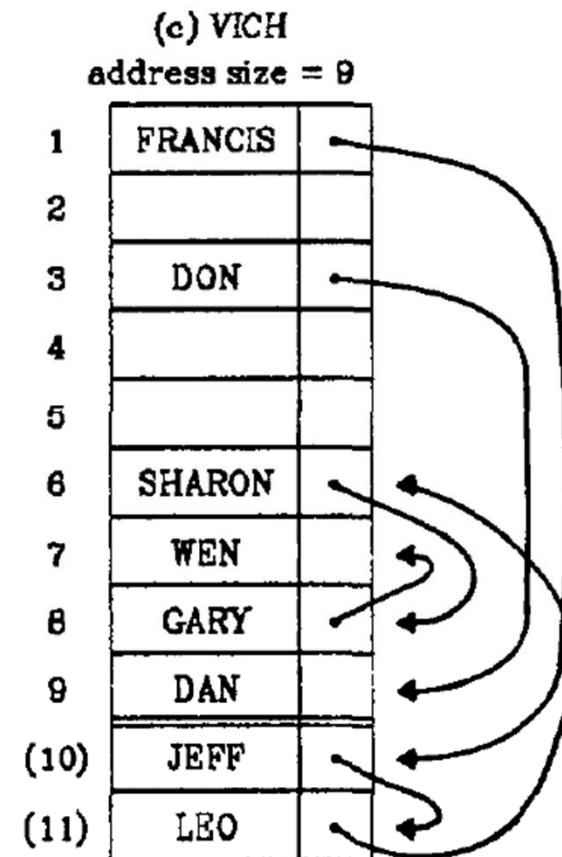
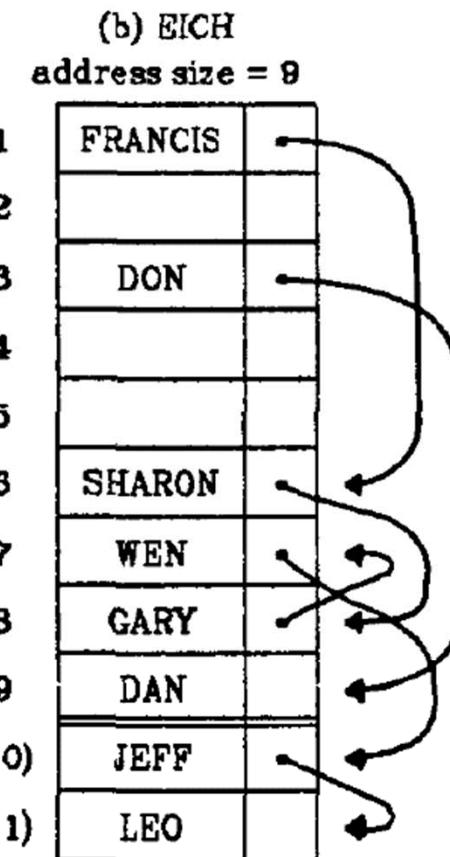
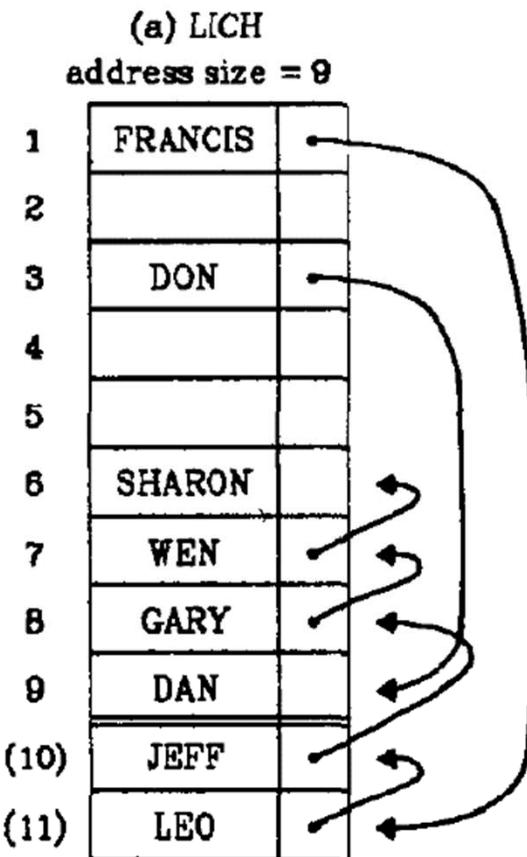
1            3            1            4            9            10            1



# Srůstající hashování s pomocnou pamětí

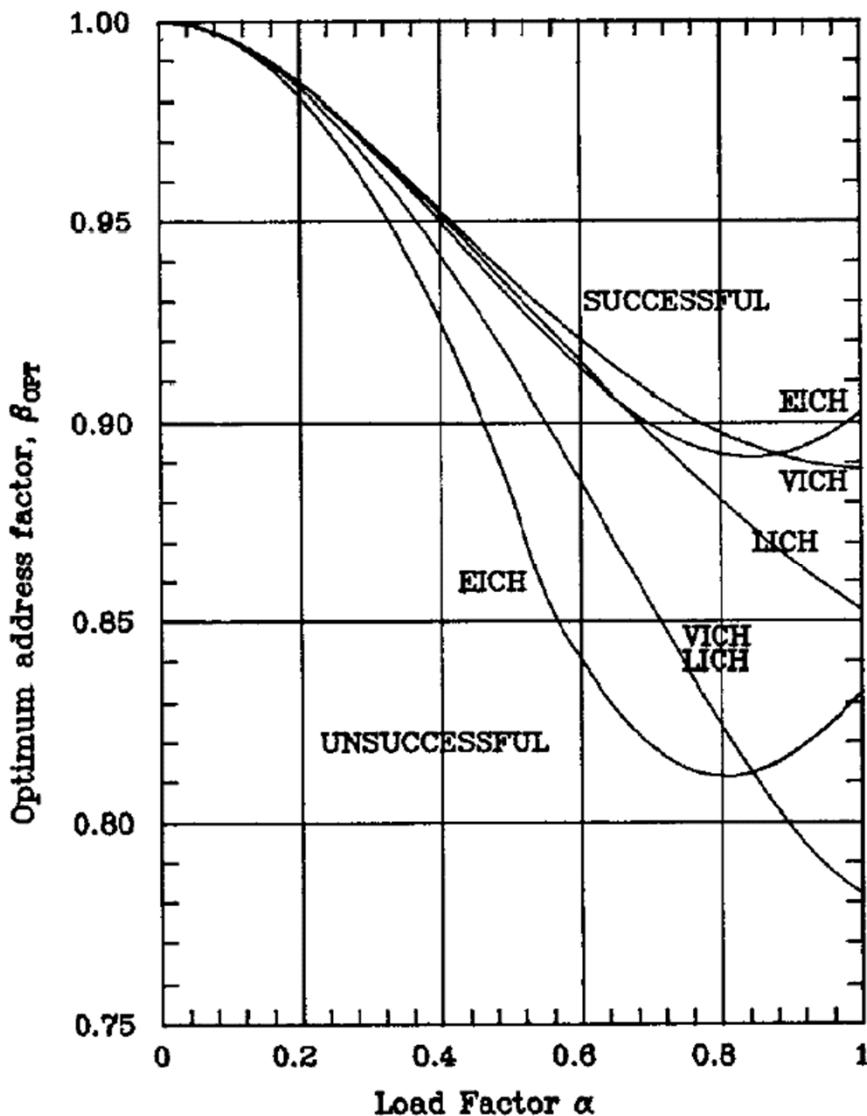
- Pro snížení srůstání a tedy zvýšení efektivity hashování se tabulka rozšiřuje o pomocnou paměť - tzv. sklep (cellar).
- Sklep je místo na konci tabulky, které není adresovatelné hashovací funkcí (má ale stejnou strukturu jako celá hashovací tabulka).
- Algoritmy LICH a EICH jsou analogické varianty algoritmů LISCH a EISCH s přidáním sklepa.
- Algoritmus VICH (variable insert coalesced hashing) připojuje prvek na konec řetězce, pokud řetězec končí ve sklepě, jinak na místo, kde řetězec opustil sklep.

# Srůstající hashování s pomocnou pamětí



Keys: FRANCIS DON LEO JEFF DAN GARY WEN SHARON  
Hash Addresses: 1 3 1 1 3 1 8 1

# Srůstající hashování s pomocnou pamětí



$\alpha$  – faktor naplnění (load factor)

$$\alpha = N/M'$$

$\beta$  – faktor adresování (address factor)

$$\beta = M/M'$$

$$K = M' - M = \text{velikost sklepa}$$

$N$  – počet vložených prvků

$M'$  – počet míst v hashovací tabulce

$M$  – počet míst adresovatelných hashovacích funkcí

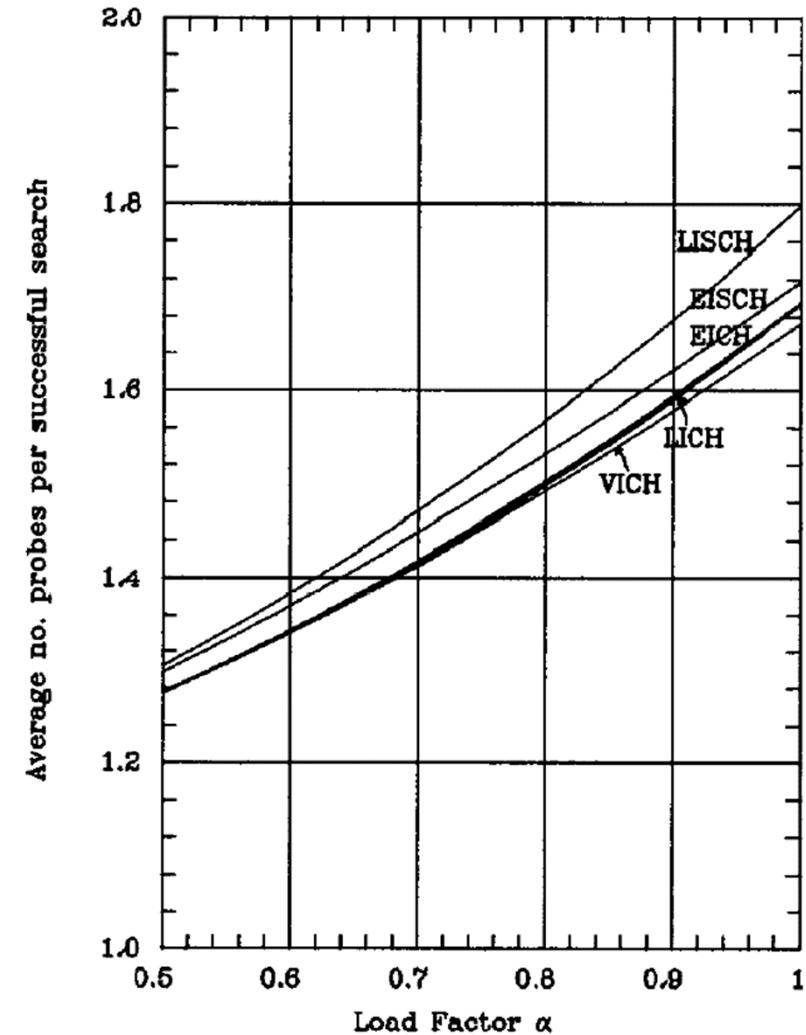
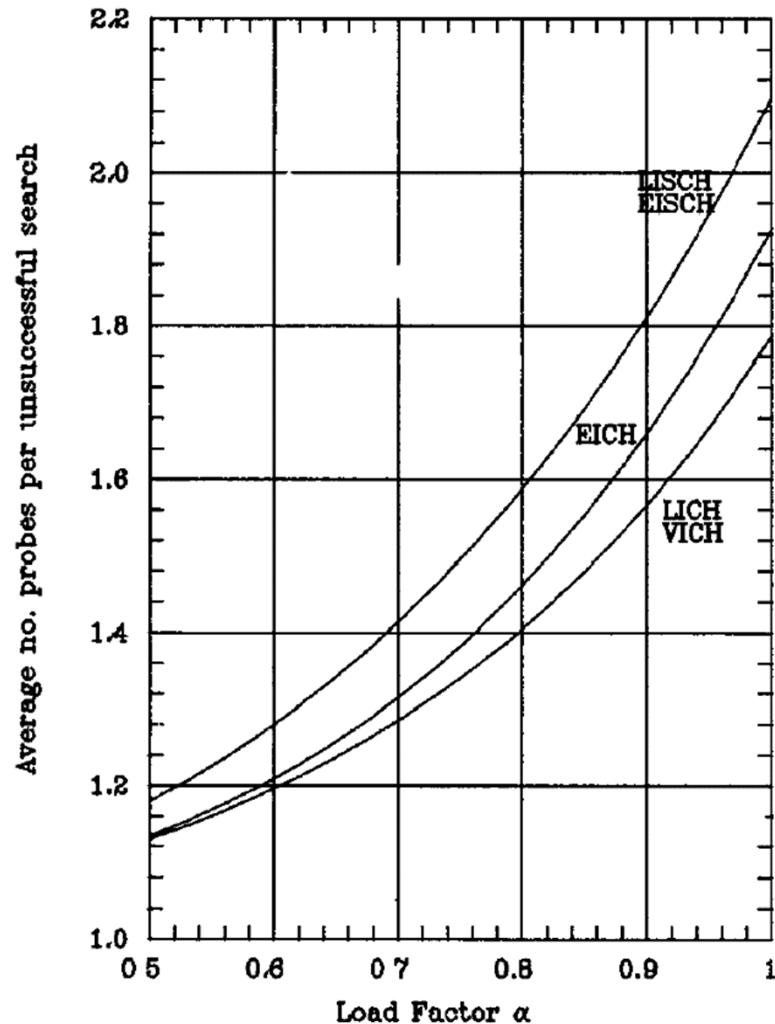
Křivky představují optimální volbu  $\beta$  v závislosti na daném  $\alpha$  při operaci FIND.

Případy:

SUCCESSFUL - klíč je v tabulce

UNSUCCESSFUL - klíč není v tabulce

# Celkové srovnání srůstajícího hashování



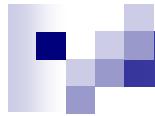
- S použitím sklepa vychází nejlépe VICH. Doporučená velikost  $\beta$  je 0,86.
- Bez sklepa vychází nejlépe EISCH.

# Efektivita srůstajícího hashování

| $\alpha$<br>method | 0.2    | 0.4    | 0.6    | 0.8    | 0.9    | 0.95   | 0.99   |
|--------------------|--------|--------|--------|--------|--------|--------|--------|
| EISCH              | 1.1065 | 1.2277 | 1.3684 | 1.5290 | 1.6182 | 1.6653 | 1.7033 |
| LISCH              | 1.1063 | 1.2316 | 1.3789 | 1.5657 | 1.6737 | 1.7337 | 1.7827 |
| BEISCH             | 1.1055 | 1.2286 | 1.3721 | 1.5336 | 1.6236 | 1.6728 | 1.7107 |
| BLISCH             | 1.1055 | 1.2341 | 1.3836 | 1.5703 | 1.6818 | 1.7423 | 1.7898 |
| REISCH             | 1.1063 | 1.2322 | 1.3693 | 1.5257 | 1.6124 | 1.6614 | 1.7014 |
| RLISCH             | 1.1085 | 1.2384 | 1.3876 | 1.5653 | 1.6723 | 1.7296 | 1.7790 |
| EICH               | 1.1116 | 1.2256 | 1.3408 | 1.4942 | 1.5867 | 1.6347 | 1.6762 |
| LICH               | 1.1116 | 1.2256 | 1.3406 | 1.4888 | 1.5801 | 1.6281 | 1.6695 |

Source: Hsiao, Yeong-Shiou, and Alan L. Tharp, "Analysis of Other New Variants of Coalesced Hashing," Technical Report TR-87-2, Computer Science Department, North Carolina State University, 1987.

- Průměrný počet navštívených klíčů při operaci FIND. Vždy se předpokládá rovnoměrné rozložení klíčů po celém oboru hodnot hashovací funkce. Nemusí nutně odpovídat reálné situaci.



# Dynamické hashování

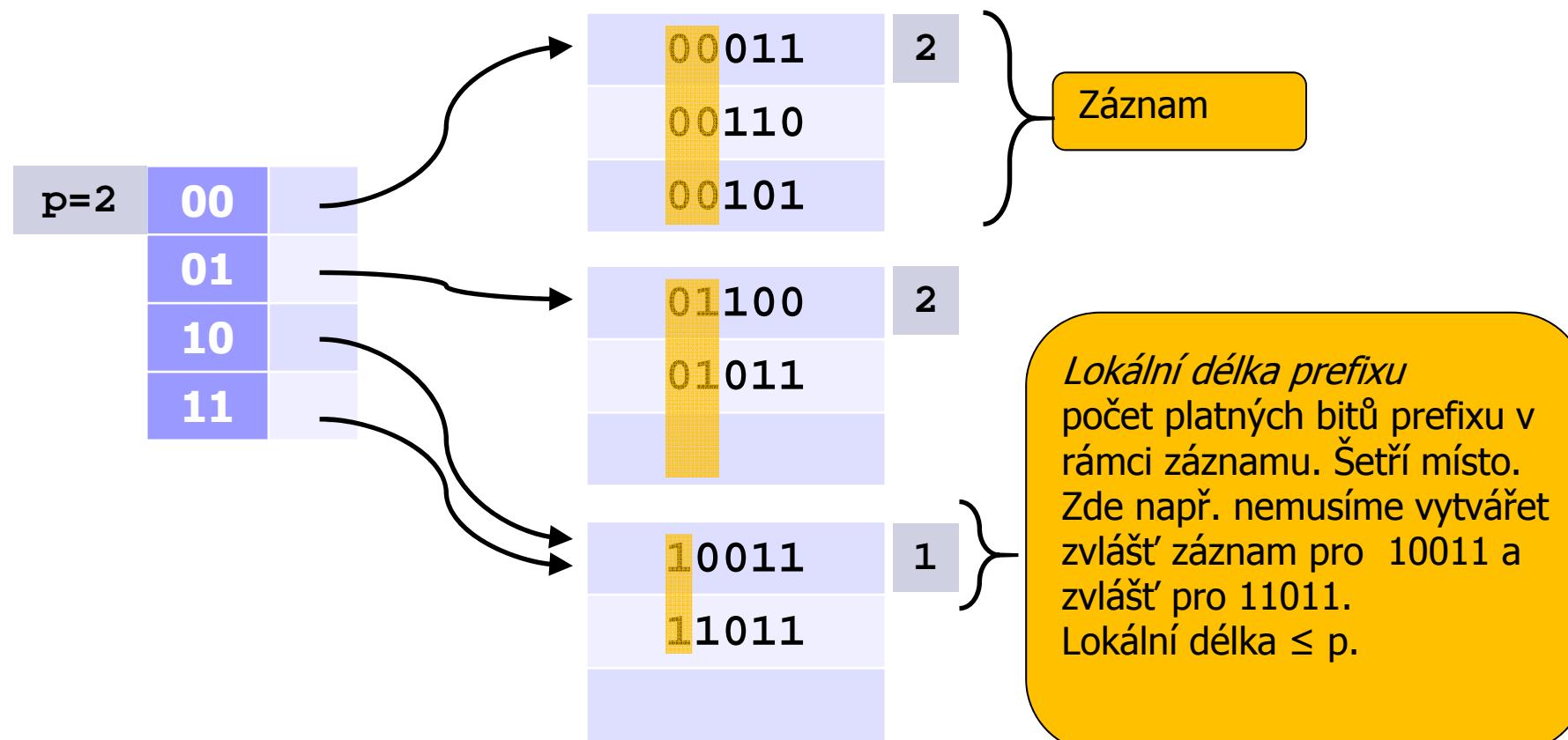
## ■ Inkrementální

- A. Při kritickém naplnění tabulky ji zvětšíme a přehashujeme.
- B. Při kritickém naplnění tabulky naalokujeme novou větší tabulku, do které začneme ukládat všechny nové prvky. Ve staré pouze vyhledáváme a rušíme prvky. Při každé operaci (ať už v nové nebo v staré tabulce) zrušíme jeden prvek ve staré tabulce a vložíme jej do nové tabulky. Po zrušení všech prvků ve staré tabulce tuto tabulku zrušíme. Toto opatření nám zajistí, že bude tabulka zrušena dříve než bude potřeba naalokovat další (tj. třetí) tabulku; a zároveň není potřeba provádět jednorázové přehashování jako v předchozím případě (takové přehashování může způsobit významné narušení plynulosti průběhu hashování).

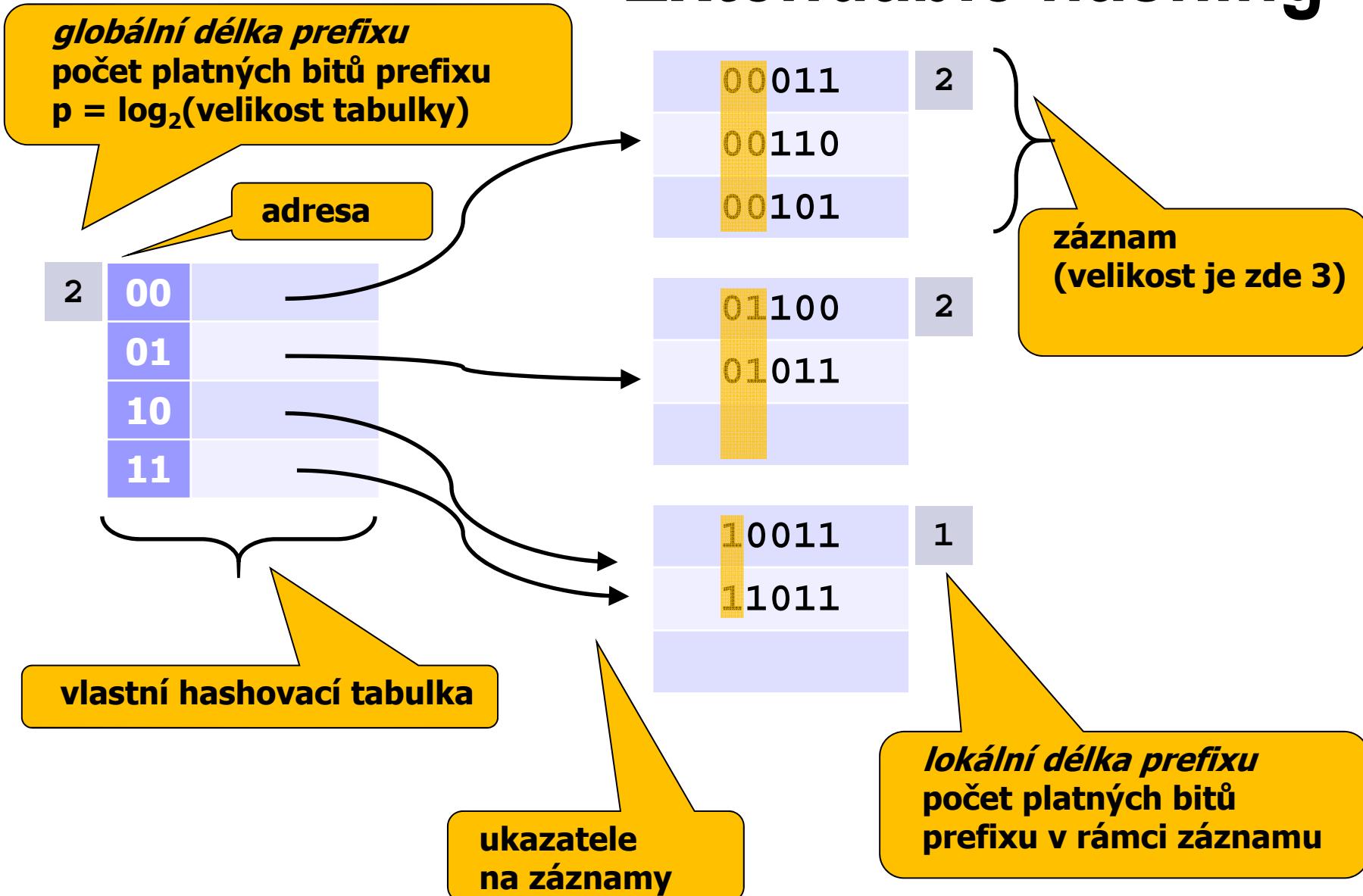
## ■ Rozšiřitelné (Extendable)

# Extendable hashing

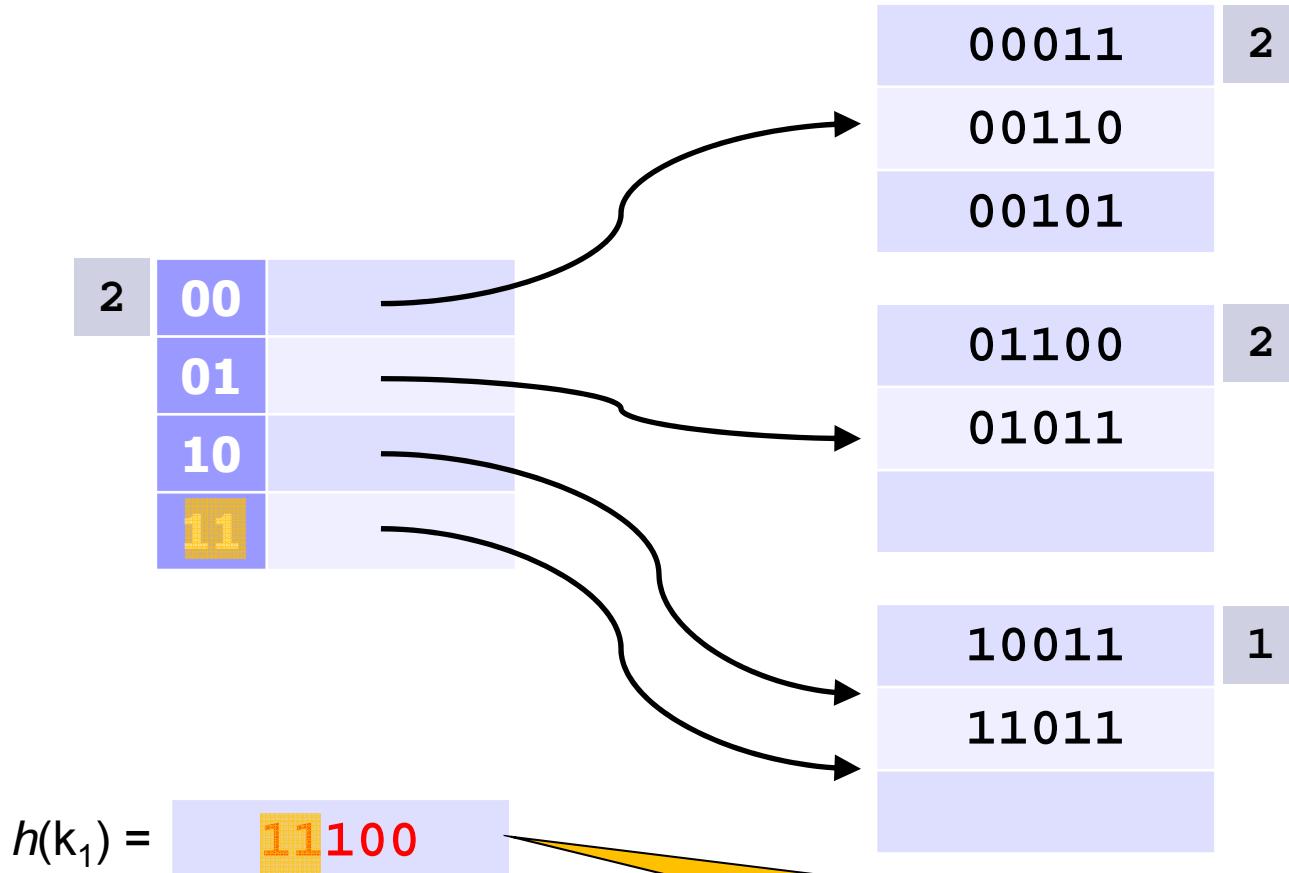
Klíč se interpretuje jako binární číslo. Tabulka má adresář a záznamy. Adresář obsahuje odkazy do záznamů a záznamy obsahují klíče. Záznamy se rozlišují prvními několika bity (prefixy) možných klíčů, klíč je v záznamu se stejným prefixem. Adresář vždy obsahuje všechny možné prefixy, jichž je  $2^p$ , kde p je délka prefixu. Hodnota p se může měnit.



# Extendable hashing

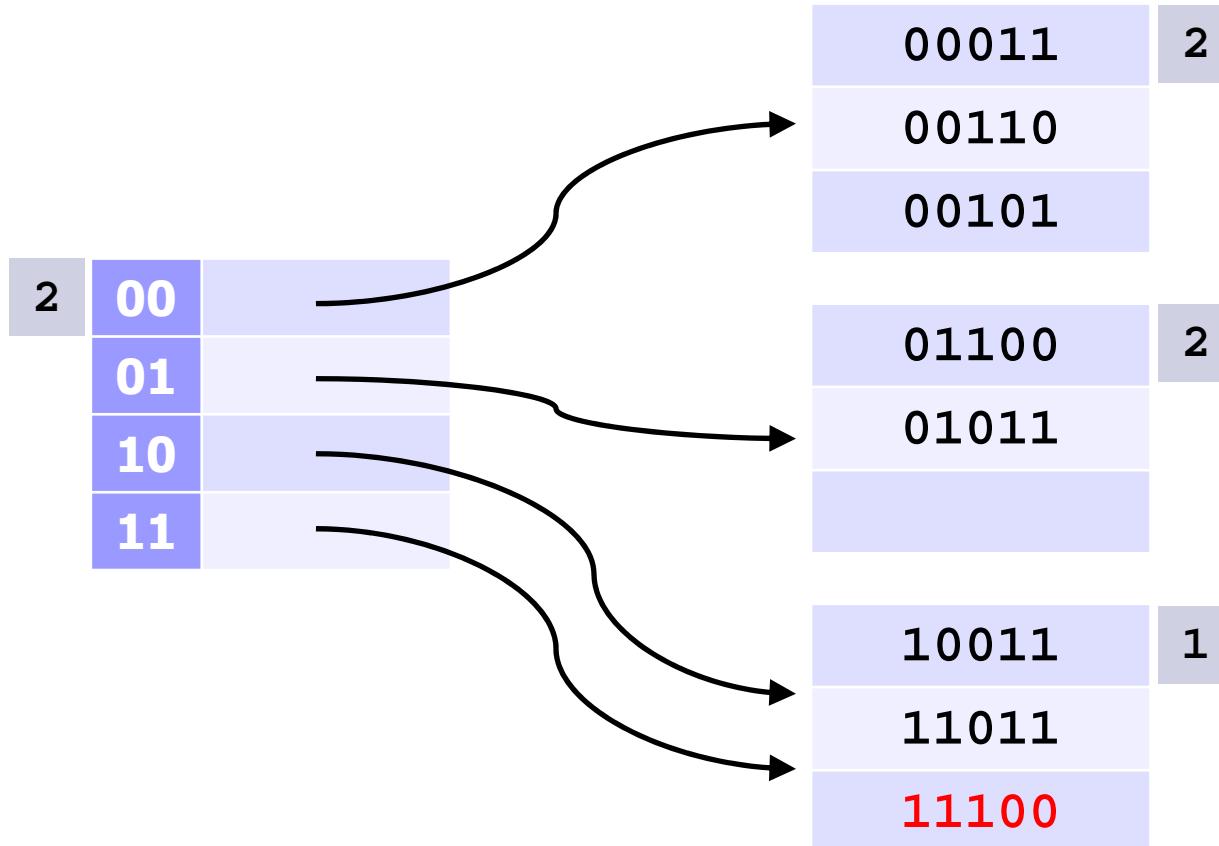


# Extendable hashing

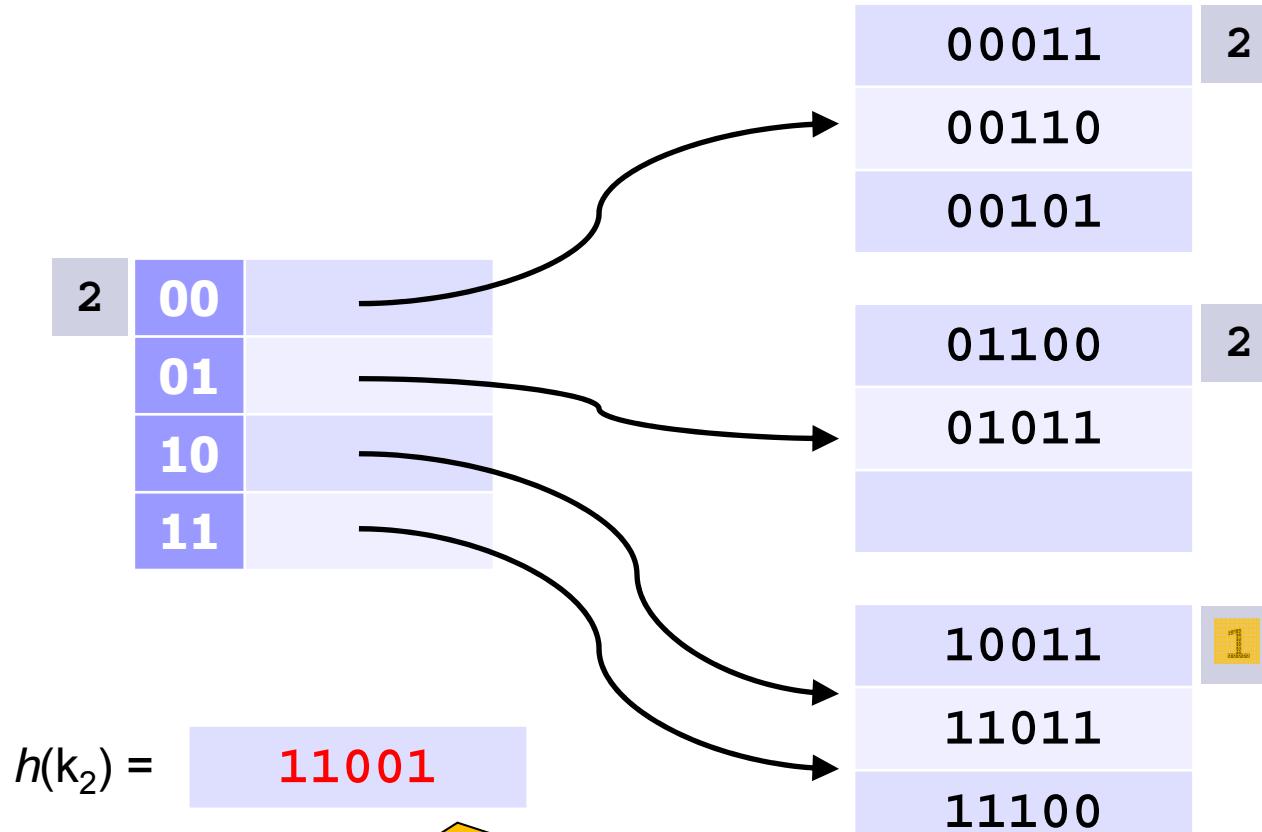


Chceme přidat nový klíč  $k_1$ .  
Záznam kam patří je určen odkazem na řádku  
hashovací tabulky s adresou se stejným prefixem  
jakou má zahashovaná hodnota klíče  $h(k_1)$ .

# Extendable hashing



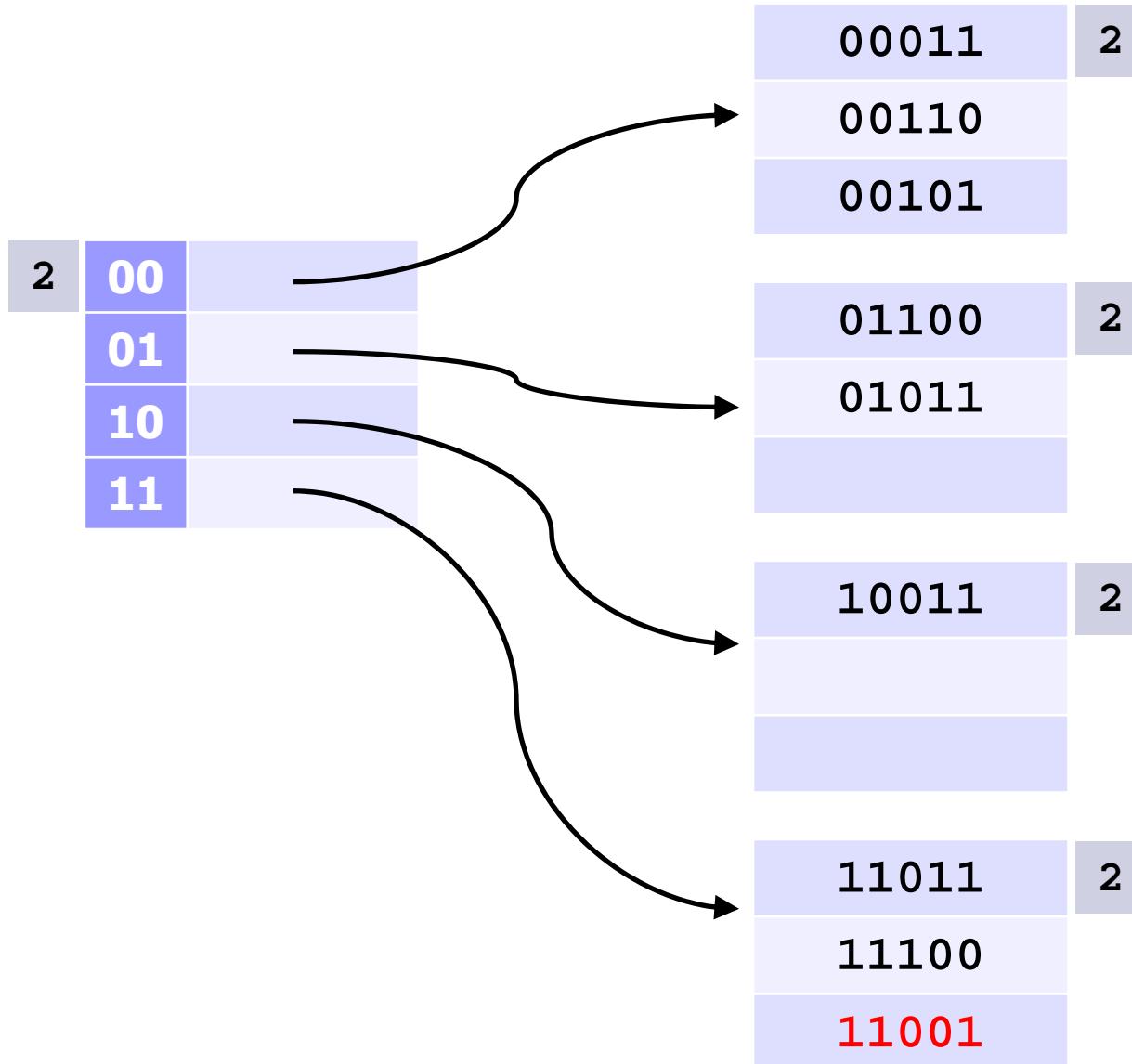
# Extendable hashing



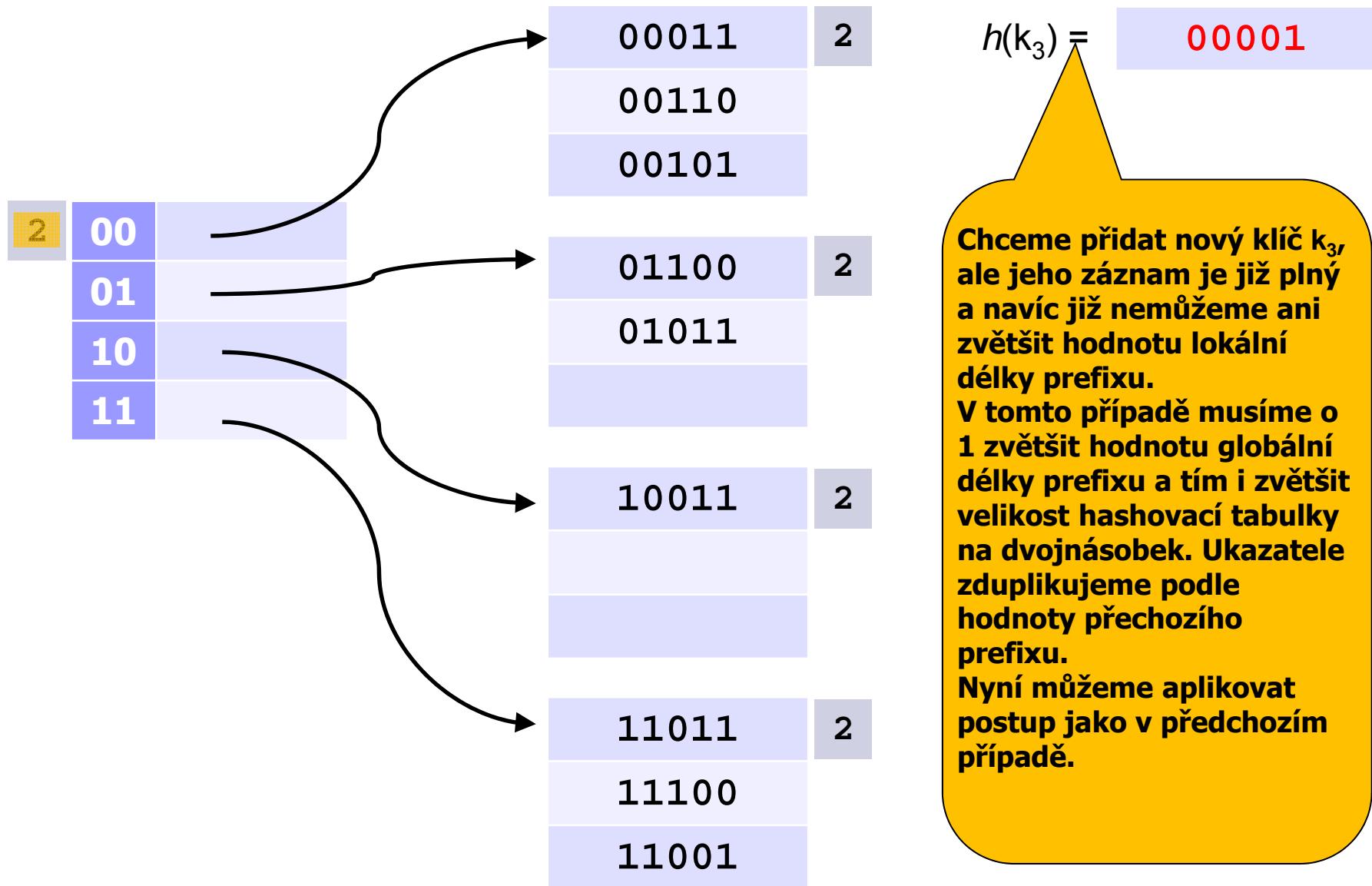
Chceme přidat nový klíč  $k_2$ , ale jeho záznam je již plný.

V tomto případě stačí zvětšit hodnotu lokální délky prefixu tohoto záznamu o 1 (lokální délka prefixu musí být vždy  $\leq$  globální délce prefixu), přidat úplně nový záznam se stejnou hodnotou prefixu do tabulky (tj. přepojit na něj první polovinu ukazatelů, které ukazovaly na původní plný záznam) a celý obsah starého plného záznamu přehashovat. Do nově vzniklého místa zahashujeme i nový klíč.

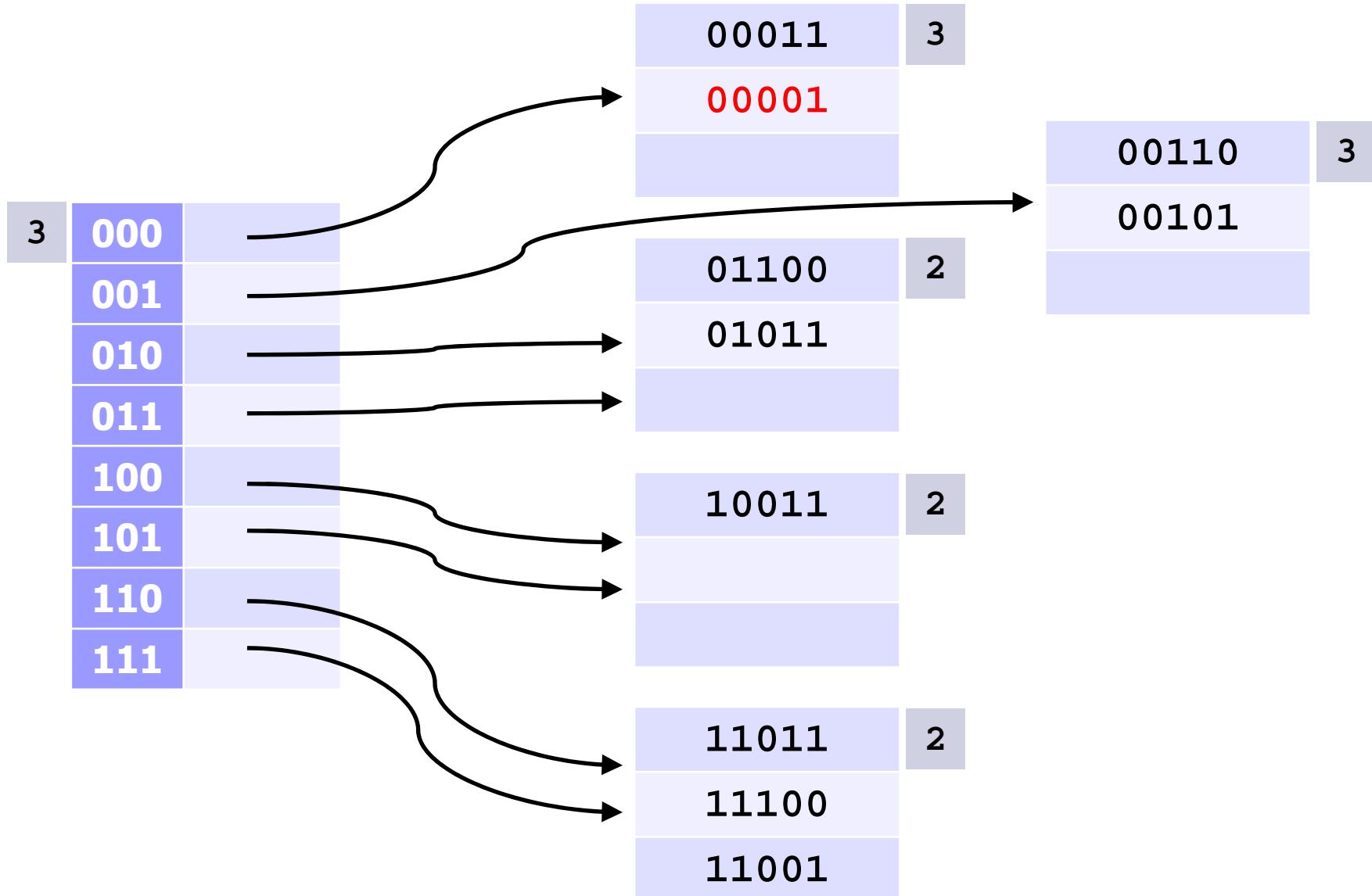
# Extendable hashing

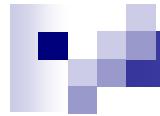


# Extendable hashing



# Extendable hashing





# Univerzální hashování

- Každá hashovací funkce má slabá místa, kdy pro různé klíče dává stejnou adresu. Proto je výhodné přizpůsobit hashovací funkci právě zpracovávaným klíčum.
- **Univerzální hashování**
  - Místo jedné hashovací funkce  $h(k)$  máme nějakou konečnou množinu  $H$  funkcí mapujících  $\mathbf{U}$  do intervalu  $\{0, 1, \dots, m-1\}$
  - Množina funkcí  $H$  je *univerzální*, pokud pro každou dvojici různých klíčů  $x, y \in \mathbf{U}$  je počet hashovacích funkcí z množiny  $H$ , pro které  $h(x) = h(y)$ , nejvýše  $|H|/m$ .
  - Důsledek: Pravděpodobnost kolize při náhodném výběru funkce  $h(k)$  z množiny univerzálních hashovacích funkcí  $H$  tedy není vyšší než pravděpodobnost kolize při náhodném a nezávislém výběru dvou stejných hodnot z intervalu  $\{0, 1, \dots, m-1\}$  tedy  $1/m$ .
  - Při prvním spuštění programu jednu náhodně zvolíme. Funkci pak náhodně měníme jen v případě, že počet kolizí převyšuje přípustnou mez. V tomto případě je samozřejmě potřeba přehashovat celou tabulku.

## ALG 13b

### Srůstající hashování

#### Ukázky

**LISCH (late insert standard coalesced hashing)**

**EISCH (early insert standard coalesced hashing)**

**LICH (late insert coalesced hashing)**

**EICH (early insert coalesced hashing)**

**VICH (variable insert coalesced hashing)**

## Srůstající hashování -- coalesced hashing

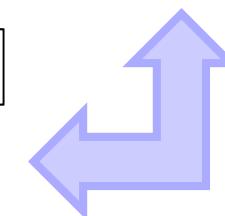
Jde o metodu řešení kolizí, nezáleží na konkrétní podobě hashovací funkce  $h(k)$ .

**Synonyma** (po kolizi) se ukádají do jednosměrného spojového seznamu **synonym**. Všechny seznamy jsou "propleteny" uloženy přímo v tabulce. Tabulka ke každému klíči obsahuje ukazatel na další klíč v seznamu. Každý klíč je součástí některého seznamu **synonym**.

Při vyhledávání se postupuje stejně jako při vkládání, v podstatě jde o lineární prohledávání spojového seznamu.



|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 6  |
| 3  |      |    |
| 4  | Irma | 8  |
| 5  | Hugo | 7  |
| 6  | Gene | 4  |
| 7  | Fred | -- |
| 8  | Edna | -- |
| 9  | Dana | 5  |
| 10 | Cole | 9  |



## LISCH (late insert standard coalesced hashing)

Hashovací funkce  $h$ , data  $d$ .

Pozice  $p := h(d)$ ;

Prohledej seznam začínající na pozici  $p$  a pokud nenajdeš  $d$ , přidej  $d$  do tabulky na první volné místo od konce tabulky a připoj ho do seznamu synonym  $d$  na poslední místo.

Ukazatel na první volné místo od konce tabulky.  
Po každém přidání prvku se aktualizuje.

|   | Name | Next |
|---|------|------|
| 0 |      |      |
| 1 |      |      |
| 2 |      |      |
| 3 |      |      |
| 4 |      |      |
| 5 |      |      |
| 6 |      |      |
| 7 |      |      |
| 8 |      |      |
| 9 |      |      |

|                  |     |     |      |      |      |      |      |      |      |
|------------------|-----|-----|------|------|------|------|------|------|------|
| data             | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| $h(\text{data})$ | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 8    | 7    |

## LISCH (late insert standard coalesced hashing)

|   |     |    |
|---|-----|----|
| 0 | Ann | -- |
| 1 |     |    |
| 2 |     |    |
| 3 |     |    |
| 4 |     |    |
| 5 |     |    |
| 6 |     |    |
| 7 |     |    |
| 8 |     |    |
| 9 |     |    |

|   |     |    |
|---|-----|----|
| 0 | Ann | -- |
| 1 |     |    |
| 2 | Ben | -- |
| 3 |     |    |
| 4 |     |    |
| 5 |     |    |
| 6 |     |    |
| 7 |     |    |
| 8 |     |    |
| 9 |     |    |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | -- |
| 3 |      |    |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 |      |    |
| 9 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 8    | 7    |

## LISCH (late insert standard coalesced hashing)

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | -- |
| 3 |      |    |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 |      |    |
| 9 | Cole | -- |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | -- |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 |      |    |
| 9 | Cole | -- |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 | Edna | -- |
| 9 | Cole | -- |

|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 8    | 7    |

## LISCH (late insert standard coalesced hashing)

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 | Edna | -- |
| 9 | Cole | -- |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 | Fred | -- |
| 8 | Edna | -- |
| 9 | Cole | 7  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 | Gene | -- |
| 7 | Fred | 6  |
| 8 | Edna | -- |
| 9 | Cole | 7  |

|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 8    | 7    |

## LISCH (late insert standard coalesced hashing)

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 | Gene | -- |
| 7 | Fred | 6  |
| 8 | Edna | -- |
| 9 | Cole | 7  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 | Hugo | -- |
| 6 | Gene | -- |
| 7 | Fred | 6  |
| 8 | Edna | 5  |
| 9 | Cole | 7  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 | Irma | -- |
| 5 | Hugo | -- |
| 6 | Gene | 5  |
| 7 | Fred | 6  |
| 8 | Edna | 5  |
| 9 | Cole | 7  |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 8    | 7    |

## EISCH (early insert standard coalesced hashing)

Hashovací funkce  $h$ , data  $d$ .

Pozice  $p := h(d)$ ;

Prohledej seznam začínající na pozici  $p$  a pokud nenajdeš  $d$ , přidej  $d$  do tabulky na první volné místo od konce tabulky a připoj ho do seznamu synonym  $d$  za první místo.

Ukazatel na první volné místo od konce tabulky.  
Po každém přidání prvku se aktualizuje.

|   | Name | Next |
|---|------|------|
| 0 |      |      |
| 1 |      |      |
| 2 |      |      |
| 3 |      |      |
| 4 |      |      |
| 5 |      |      |
| 6 |      |      |
| 7 |      |      |
| 8 |      |      |
| 9 |      |      |

|                  |     |     |      |      |      |      |      |      |      |
|------------------|-----|-----|------|------|------|------|------|------|------|
| data             | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| $h(\text{data})$ | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## EISCH (early insert standard coalesced hashing)

|   |     |    |
|---|-----|----|
| 0 | Ann | -- |
| 1 |     |    |
| 2 |     |    |
| 3 |     |    |
| 4 |     |    |
| 5 |     |    |
| 6 |     |    |
| 7 |     |    |
| 8 |     |    |
| 9 |     |    |

|   |     |    |
|---|-----|----|
| 0 | Ann | -- |
| 1 |     |    |
| 2 | Ben | -- |
| 3 |     |    |
| 4 |     |    |
| 5 |     |    |
| 6 |     |    |
| 7 |     |    |
| 8 |     |    |
| 9 |     |    |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | -- |
| 3 |      |    |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 |      |    |
| 9 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## EISCH (early insert standard coalesced hashing)

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | -- |
| 3 |      |    |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 |      |    |
| 9 | Cole | -- |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | -- |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 |      |    |
| 9 | Cole | -- |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 | Edna | -- |
| 9 | Cole | -- |

|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## EISCH (early insert standard coalesced hashing)

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 |      |    |
| 8 | Edna | -- |
| 9 | Cole | -- |

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 | Fred | -- |
| 8 | Edna | -- |
| 9 | Cole | 7  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 6  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 | Gene | 9  |
| 7 | Fred | -- |
| 8 | Edna | -- |
| 9 | Cole | 7  |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## EISCH (early insert standard coalesced hashing)

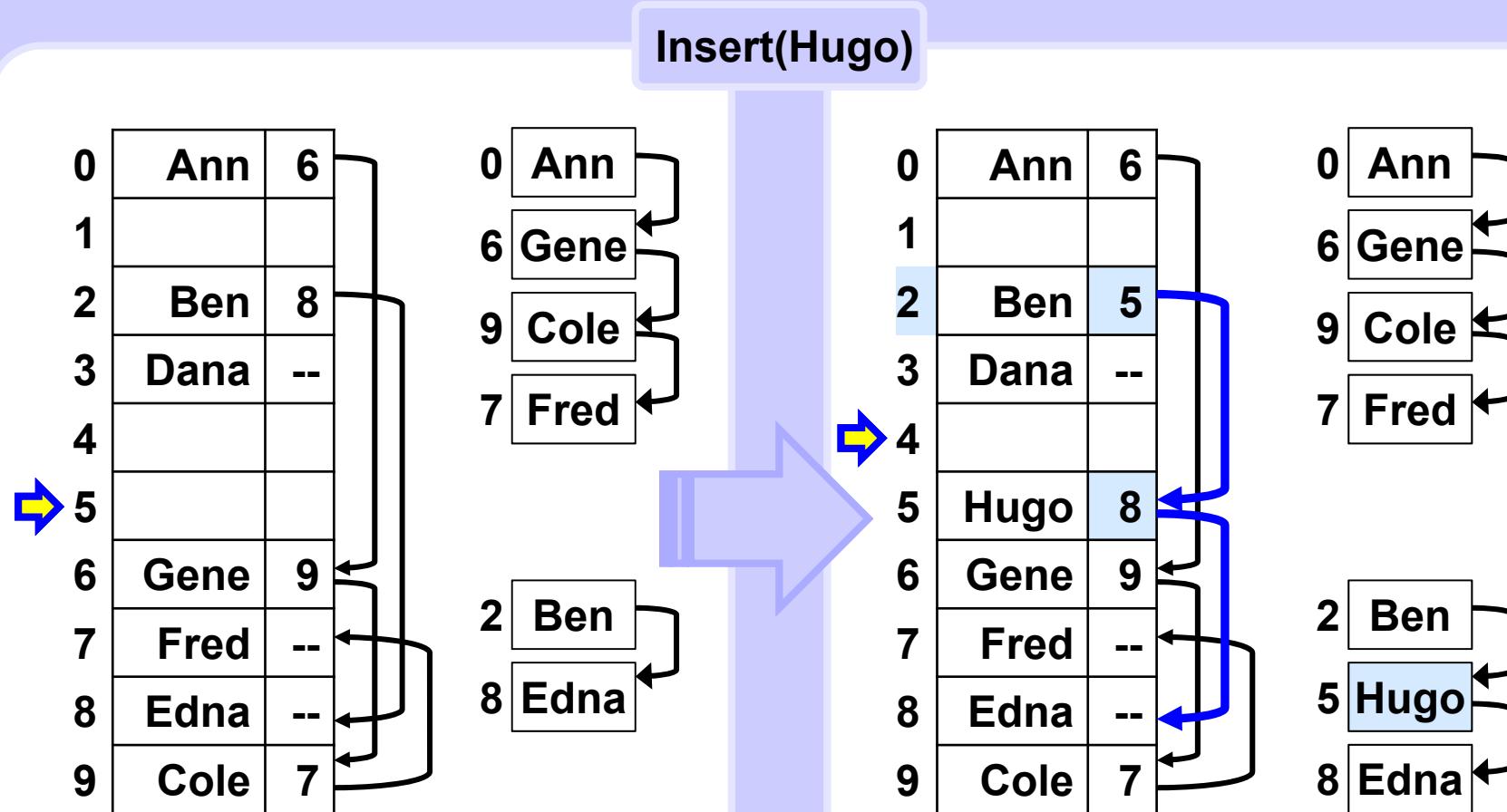
|   |      |    |
|---|------|----|
| 0 | Ann  | 6  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 |      |    |
| 6 | Gene | 9  |
| 7 | Fred | -- |
| 8 | Edna | -- |
| 9 | Cole | 7  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 6  |
| 1 |      |    |
| 2 | Ben  | 5  |
| 3 | Dana | -- |
| 4 |      |    |
| 5 | Hugo | 8  |
| 6 | Gene | 9  |
| 7 | Fred | -- |
| 8 | Edna | -- |
| 9 | Cole | 7  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 6  |
| 1 |      |    |
| 2 | Ben  | 5  |
| 3 | Dana | -- |
| 4 | Irma | 9  |
| 5 | Hugo | 8  |
| 6 | Gene | 4  |
| 7 | Fred | -- |
| 8 | Edna | -- |
| 9 | Cole | 7  |

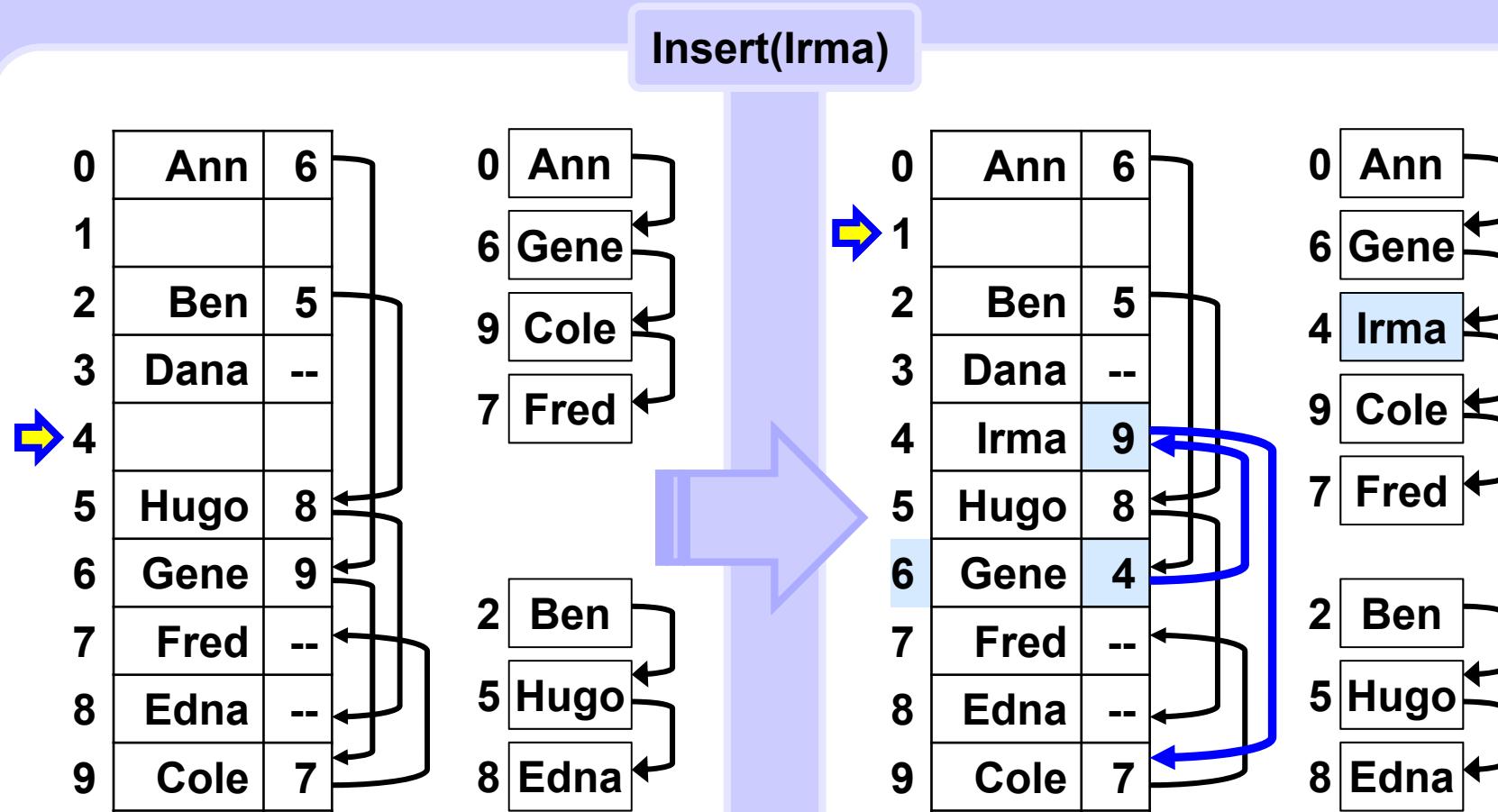
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## EISCH (early insert standard coalesced hashing)



|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## EISCH (early insert standard coalesced hashing)



|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 3    | 2    | 9    | 0    | 2    | 6    |

## Srůstající hashování s pomocnou pamětí

Pro snížení srůstání a tedy zvýšení efektivity hashování se tabulka rozšiřuje o pomocnou paměť - tzv. sklep (cellar).

Sklep je místo na konci tabulky, které není adresovatelné hashovací funkcí, má ale stejnou strukturu jako celá tabulka.

Algoritmy LICH a EICH jsou analogické varianty algoritmů LISCH a EISCH s přidáním sklepa.

Po naplnění sklepa pokračuje plnění jako v LISCH a EISCH.

Algoritmus VICH (variable insert coalesced hashing) připojuje prvek za poslední prvek seznamu, který je ještě ve sklepě. Pokud ve sklepě žádný není, vkládá jako EISCH, tj. hned za kolidující prvek v seznamu.

## LICH (late insert coalesced hashing)

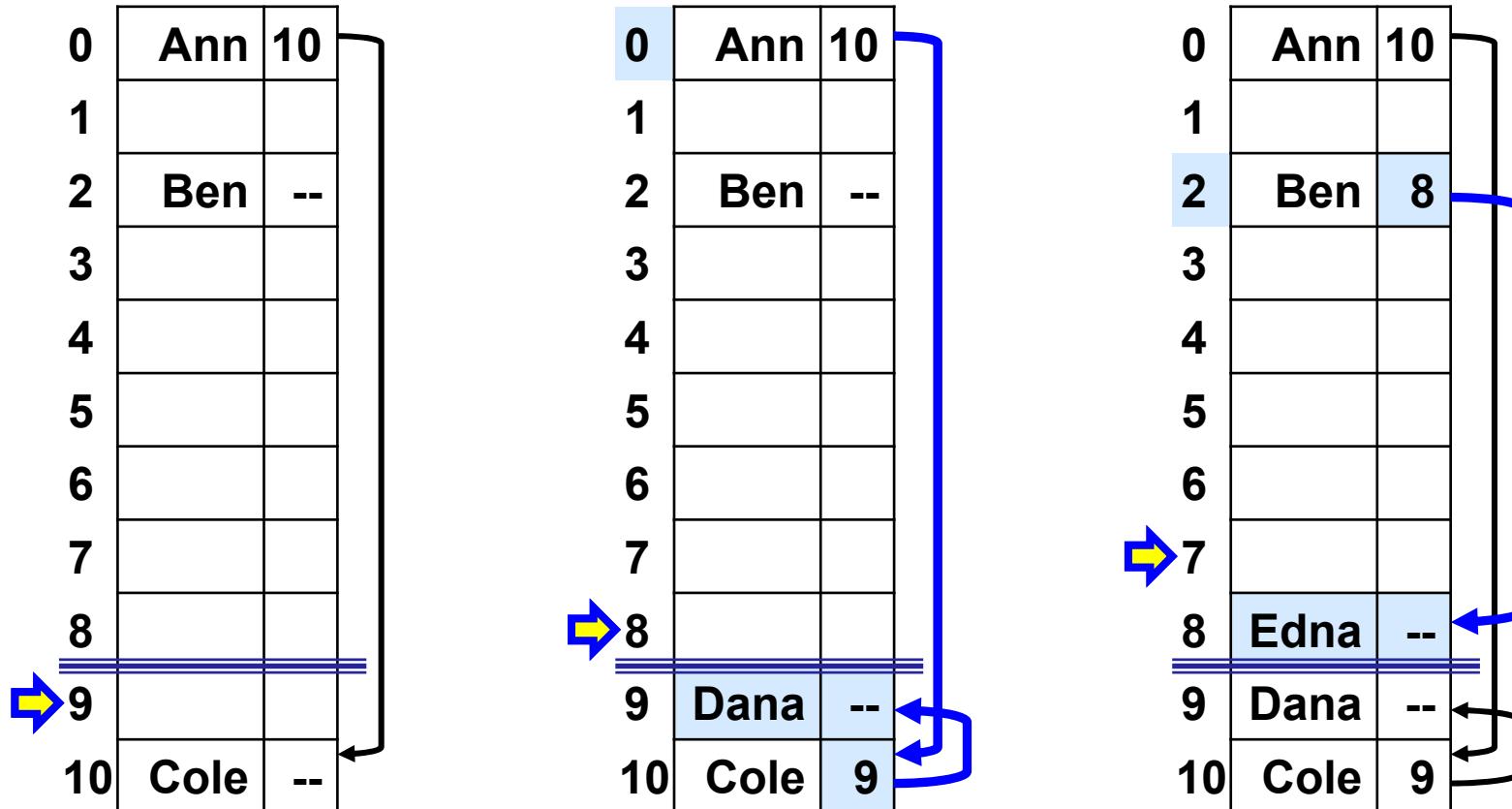
|    |     |    |
|----|-----|----|
| 0  | Ann | -- |
| 1  |     |    |
| 2  |     |    |
| 3  |     |    |
| 4  |     |    |
| 5  |     |    |
| 6  |     |    |
| 7  |     |    |
| 8  |     |    |
| 9  |     |    |
| 10 |     |    |

|    |     |    |
|----|-----|----|
| 0  | Ann | -- |
| 1  |     |    |
| 2  | Ben | -- |
| 3  |     |    |
| 4  |     |    |
| 5  |     |    |
| 6  |     |    |
| 7  |     |    |
| 8  |     |    |
| 9  |     |    |
| 10 |     |    |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | -- |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  |      |    |
| 8  |      |    |
| 9  |      |    |
| 10 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 5    | 8    |

## LICH (late insert coalesced hashing)



|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 5    | 8    |

## LICH (late insert coalesced hashing)

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  |      |    |
| 8  | Edna | -- |
| 9  | Dana | -- |
| 10 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  | Fred | -- |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  | Gene | -- |
| 7  | Fred | 6  |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 5    | 8    |

## LICH (late insert coalesced hashing)

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  | Gene | -- |
| 7  | Fred | 6  |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  | Hugo | -- |
| 6  | Gene | -- |
| 7  | Fred | 6  |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  | Irma | -- |
| 5  | Hugo | -- |
| 6  | Gene | -- |
| 7  | Fred | 6  |
| 8  | Edna | 4  |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 5    | 8    |

## EICH (early insert coalesced hashing)

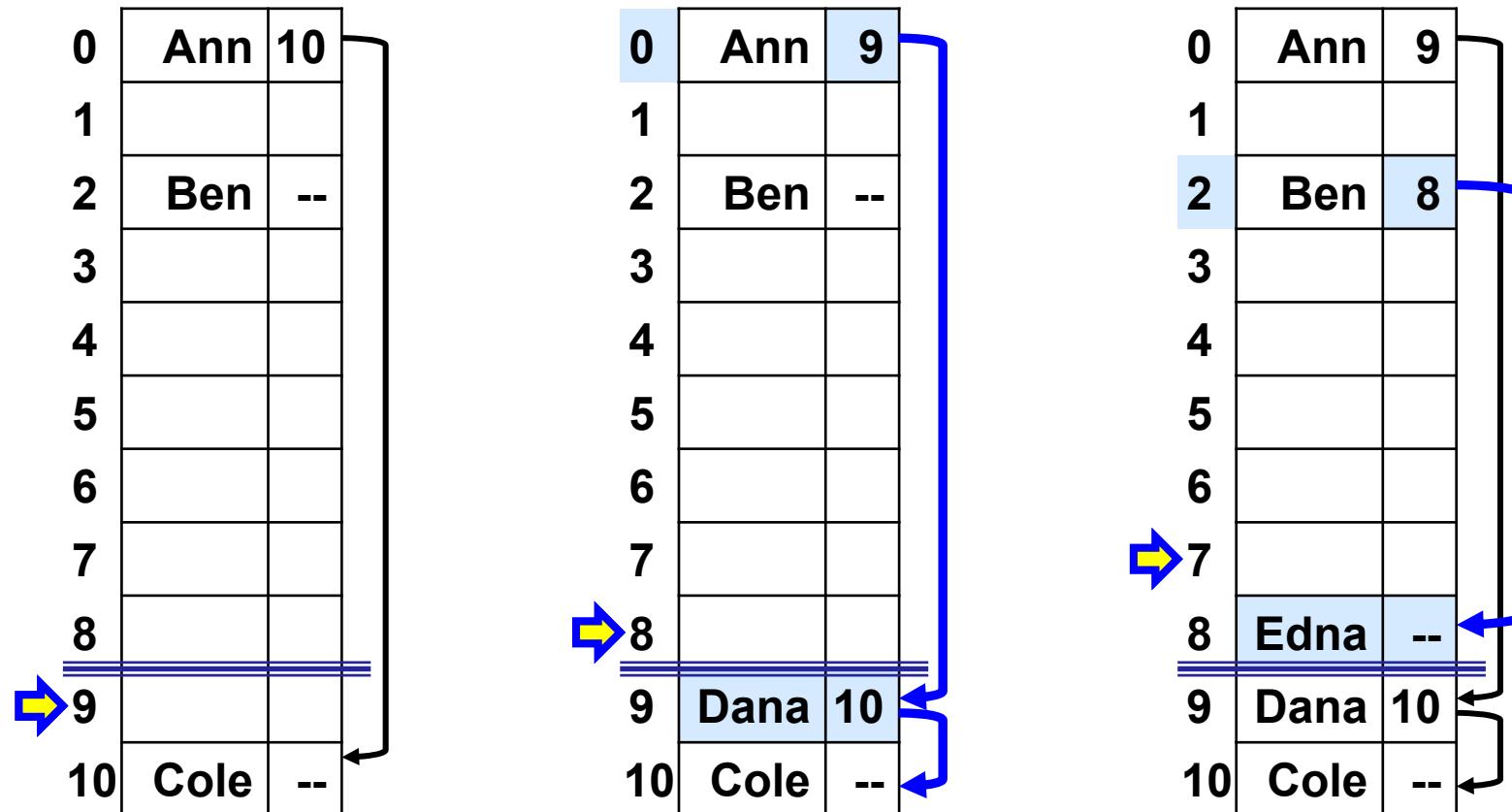
|    |     |    |
|----|-----|----|
| 0  | Ann | -- |
| 1  |     |    |
| 2  |     |    |
| 3  |     |    |
| 4  |     |    |
| 5  |     |    |
| 6  |     |    |
| 7  |     |    |
| 8  |     |    |
| 9  |     |    |
| 10 |     |    |

|    |     |    |
|----|-----|----|
| 0  | Ann | -- |
| 1  |     |    |
| 2  | Ben | -- |
| 3  |     |    |
| 4  |     |    |
| 5  |     |    |
| 6  |     |    |
| 7  |     |    |
| 8  |     |    |
| 9  |     |    |
| 10 |     |    |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | -- |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  |      |    |
| 8  |      |    |
| 9  |      |    |
| 10 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 0    | 8    |

## EICH (early insert coalesced hashing)



|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 0    | 8    |

## EICH (early insert coalesced hashing)

|   |      |    |
|---|------|----|
| 0 | Ann  | 9  |
| 1 |      |    |
| 2 | Ben  | 8  |
| 3 |      |    |
| 4 |      |    |
| 5 |      |    |
| 6 |      |    |
| 7 | Edna | -- |
| 8 | Dana | 10 |
| 9 | Cole | -- |

|    |      |    |
|----|------|----|
| 0  | Ann  | 7  |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  | Fred | 9  |
| 8  | Edna | -- |
| 9  | Dana | 10 |
| 10 | Cole | -- |

|    |      |    |
|----|------|----|
| 0  | Ann  | 7  |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  | Gene | 9  |
| 7  | Fred | 6  |
| 8  | Edna | -- |
| 9  | Dana | 10 |
| 10 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 0    | 8    |

## EICH (early insert coalesced hashing)

|    |      |    |
|----|------|----|
| 0  | Ann  | 7  |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  | Gene | 9  |
| 7  | Fred | 6  |
| 8  | Edna | -- |
| 9  | Dana | 10 |
| 10 | Cole | -- |

|    |      |    |
|----|------|----|
| 0  | Ann  | 5  |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  | Hugo | 7  |
| 6  | Gene | 9  |
| 7  | Fred | 6  |
| 8  | Edna | -- |
| 9  | Dana | 10 |
| 10 | Cole | -- |

|    |      |    |
|----|------|----|
| 0  | Ann  | 5  |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  | Irma | -- |
| 5  | Hugo | 7  |
| 6  | Gene | 9  |
| 7  | Fred | 6  |
| 8  | Edna | 4  |
| 9  | Dana | 10 |
| 10 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 7    | 0    | 8    |

## VICH (variable insert coalesced hashing)

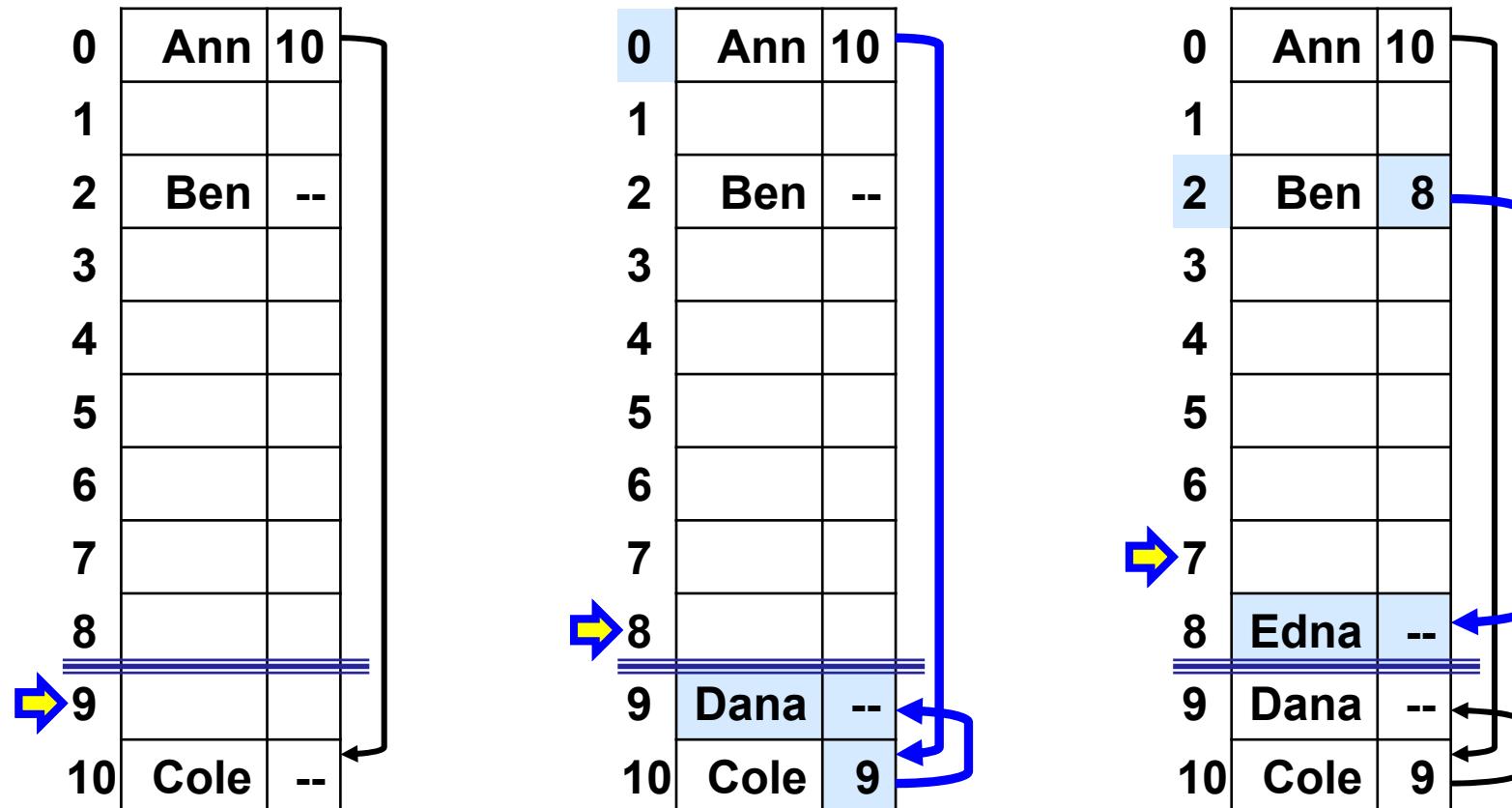
|    |     |    |
|----|-----|----|
| 0  | Ann | -- |
| 1  |     |    |
| 2  |     |    |
| 3  |     |    |
| 4  |     |    |
| 5  |     |    |
| 6  |     |    |
| 7  |     |    |
| 8  |     |    |
| 9  |     |    |
| 10 |     |    |

|    |     |    |
|----|-----|----|
| 0  | Ann | -- |
| 1  |     |    |
| 2  | Ben | -- |
| 3  |     |    |
| 4  |     |    |
| 5  |     |    |
| 6  |     |    |
| 7  |     |    |
| 8  |     |    |
| 9  |     |    |
| 10 |     |    |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | -- |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  |      |    |
| 8  |      |    |
| 9  |      |    |
| 10 | Cole | -- |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 2    | 0    | 6    |

## VICH (variable insert coalesced hashing)



|         |     |     |      |      |      |      |      |      |      |
|---------|-----|-----|------|------|------|------|------|------|------|
| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 2    | 0    | 6    |

## VICH (variable insert coalesced hashing)

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  |      |    |
| 8  | Edna | -- |
| 9  | Dana | -- |
| 10 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 8  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  |      |    |
| 7  | Fred | -- |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 6  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  | Gene | 8  |
| 7  | Fred | -- |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 2    | 0    | 6    |

## VICH (variable insert coalesced hashing)

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 6  |
| 3  |      |    |
| 4  |      |    |
| 5  |      |    |
| 6  | Gene | 8  |
| 7  | Fred | -- |
| 8  | Edna | -- |
| 9  | Dana | 7  |
| 10 | Cole | 9  |

|   |      |    |
|---|------|----|
| 0 | Ann  | 10 |
| 1 |      |    |
| 2 | Ben  | 6  |
| 3 |      |    |
| 4 | Hugo | 7  |
| 5 | Gene | 8  |
| 6 | Fred | -- |
| 7 | Edna | -- |
| 8 | Dana | 5  |
| 9 | Cole | 9  |

|    |      |    |
|----|------|----|
| 0  | Ann  | 10 |
| 1  |      |    |
| 2  | Ben  | 6  |
| 3  |      |    |
| 4  | Irma | 8  |
| 5  | Hugo | 7  |
| 6  | Gene | 4  |
| 7  | Fred | -- |
| 8  | Edna | -- |
| 9  | Dana | 5  |
| 10 | Cole | 9  |

| data    | Ann | Ben | Cole | Dana | Edna | Fred | Gene | Hugo | Irma |
|---------|-----|-----|------|------|------|------|------|------|------|
| h(data) | 0   | 2   | 0    | 0    | 2    | 0    | 2    | 0    | 6    |

## ALG 14

### Hledání k-tého nejmenšího prvku

Randomized select

CLRS varianta Partition v Quicksortu

Select metodou medián z mediánů

## Hledání k-tého nejmenšího prvku

1. Seřad' seznam/pole a vyber k-tý nejmenší,  
složitost  $\Theta(N * \log(N))$ .  
Nevýhodou je zbytečné řazení úplně všech dat.
2. Využij Randomized select založený na principu Quick sortu.  
Očekávaná složitost  $\Theta(N)$ , nejhorší možná  $\Theta(N^2)$ .
3. Využij Select založený na principu Quick sortu s výběrem pivota metodou medián z mediánů.  
Zaručená složitost  $\Theta(N)$ ,  
velká režie - vhodná jen pro velká data.

## Randomized select

**Partition -- dělení na "malé" a "velké" v Quicksortu.**

### Randomized partition

- Vyber prvek na náhodné pozici pole/úseku jako pivot.
- Proved' partition jako v Quicksortu s vybraným pivotem.

### Randomized select (k-tý nejmenší prvek)

- Když je délka úseku rovna 1, vrat' jeho jediný prvek. Jinak:
- Proved' Randomized partition tohoto úseku, vznikne levý a pravý podúsek.
- Aplikuj Randomized select rekurzivně buď na levý nebo pravý podúsek, podle toho, v kterém z nich může ležet k-tý nejmenší prvek celého pole.

## Randomized select

pozice k

k-tý nejmenší (není na pozici k !!)

### (Quicksort) Partition

Partition mění pořadí prvků v poli.

malé

velké

Alespoň k prvků

Zde k-tý prvek být nemůže

### (Quicksort) Partition

malé

velké

Partition mění pořadí prvků v poli.

Méně než k prvků, zde k-tý prvek být nemůže.

Aplikuj Partition a celý postup rekurzivně v tomto úseku.

Již nezkoumej.

## Randomized select

Příklad -- hledání 7. nejmenšího prvku

|          |    |    |    |    |           |           |           |    |           |           |    |    |           |    |    |
|----------|----|----|----|----|-----------|-----------|-----------|----|-----------|-----------|----|----|-----------|----|----|
|          | 1  | 2  | 3  | 4  | 5         | 6         | <b>7</b>  | 8  | 9         | 10        | 11 | 12 | 13        | 14 | 15 |
| Pivot 23 | 14 | 16 | 21 | 11 | 15        | 24        | <b>12</b> | 18 | 17        | 25        | 20 | 19 | <b>23</b> | 13 | 22 |
| Pivot 13 | 14 | 16 | 21 | 11 | 15        | 22        | <b>12</b> | 18 | 17        | <b>13</b> | 20 | 19 | 23        | 25 | 24 |
| Pivot 14 | 13 | 12 | 11 | 21 | 15        | 22        | <b>16</b> | 18 | 17        | <b>14</b> | 20 | 19 | 23        | 25 | 24 |
| Pivot 17 | 13 | 12 | 11 | 14 | <b>15</b> | 22        | <b>16</b> | 18 | <b>17</b> | 21        | 20 | 19 | 23        | 25 | 24 |
| Pivot 16 | 13 | 12 | 11 | 14 | <b>15</b> | <b>17</b> | <b>16</b> | 18 | 22        | 21        | 20 | 19 | 23        | 25 | 24 |
|          | 13 | 12 | 11 | 14 | 15        | 16        | <b>17</b> | 18 | 22        | 21        | 20 | 19 | 23        | 25 | 24 |

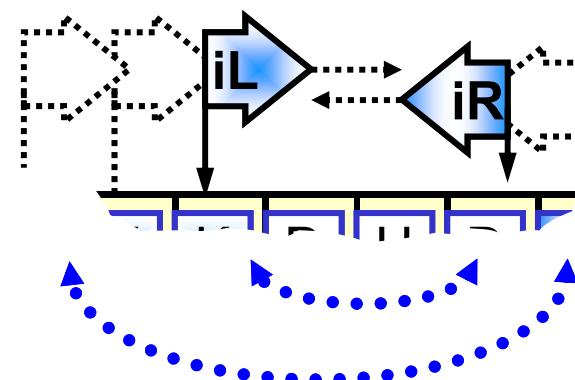
## Randomized select

Dělení úseku pole od indexu L, až po index R včetně na "malé" a "velké" hodnoty podle předem zvolené pivotní hodnoty.

Analogicky jako v Quick sortu.

```
// returns index of the first element in the "big" part
int partition( int [] a, int L, int R, int pivot) {
    int iL = L, iR = R;
    do {
        while (a[iL] < pivot) iL++;
        while (a[iR] > pivot) iR--;
        if (iL < iR)
            swap(a, iL++, iR--);
    } while(iL < iR);

    // if necessary move right by 1
    if ((a[iL] < pivot) || (iL == L)) iL++;
    return iL;
}
```



## Randomized select

```

int quickselect(int [] a, int iL, int iR, int k) {
    if (iL == iR) return a[iL];           // all done, found

    int pivot = a[randomInt(L, R)];      // random pivot
    int iMidR = partition(a, L, R, pivot);

    // recursively process **only** the part in which
    // the k-th element **surely** is present
    if (k < iMidR)
        return quickselect(a, iL, iMidR-1, k);
    else
        return quickselect(a, iMidR, iR, k);
}

```

### Příklad volání

```

int k = ...; // whatever value
int kth_element = quickselect(a, 0, a.length(), k);

```

## Randomized select

### Asymptotická složitost -- podobně jako u Quicksortu

**Nevhodná volba pivota -- složitost může degenerovat až na  $\Theta(n^2)$ , pokud by se v každém kroku zmenšila délka zpracovávaného úseku jen o malou konstantu.**

**Náhodná volba pivota činí tuto možnost velmi nepravděpodobnou, prakticky nenastává.**

### Očekávaná složitost

**Ve skoro každém kroku klesá délka zpracovávaného úseku úměrně konstantě  $\lambda < 1$ , délky úseků pak odpovídají hodnotám**

$$N, N\lambda, N\lambda^2, N\lambda^3, \dots,$$

**celková délka všech úseků je pak menší než součet řady**

$$\begin{aligned} N + N\lambda + N\lambda^2 + N\lambda^3 + \dots &= N(1 + \lambda + \lambda^2 + \lambda^3 + \dots) = \\ &= N(1/(1-\lambda)) = N/(1-\lambda) \in \Theta(N). \end{aligned}$$

## Alternativní metoda Partition

V literatuře (např. [CLRS] a na webu, např. Wikipedie) najdeme postup popsaný kódem níže. Je koncepčně jednoduchý, fakticky ale pomalejší, faktorem cca 2 až 3.

```
// returns index of the first element in the "big" part
int partitionCLRS(int [] a, int iL, int iR) {
    int pivot = a[iR];
    int iMid = iL-1;
    for (int j = iL; j < iR; j++ )
        if (a[j] <= pivot)
            swap(a, ++iMid, j);
    swap(a, ++iMid, R);
    return iMid;
}
```

Swapuje se každý prvek menší nebo roven pivotu,  
Očekáváme, že cca polovina prvků v úseku bude swapována.  
V klasické variantě se swapuje mnohem méně.

## Alternativní metoda Partition

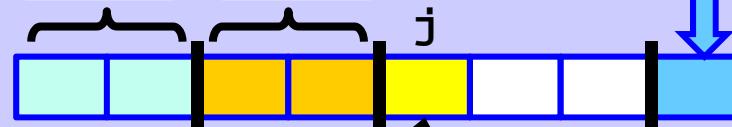
### Ukázka průběhu

Postup zleva doprava

Zpracované

Malé      Velké

Pivot

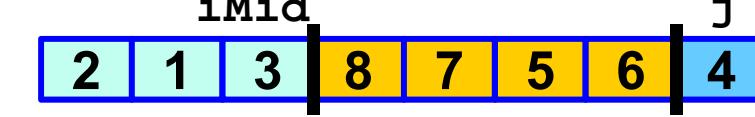
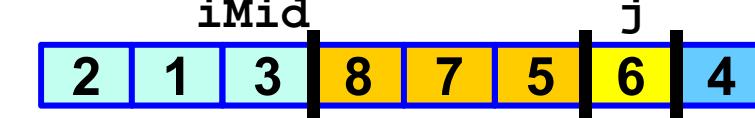
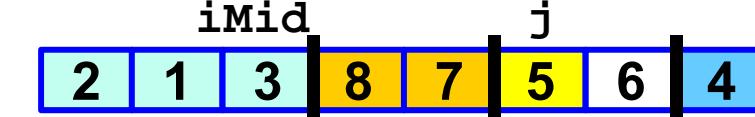
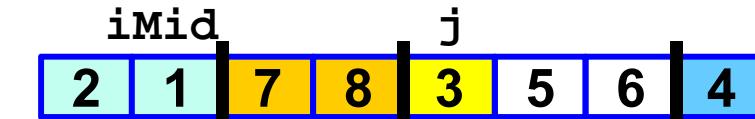
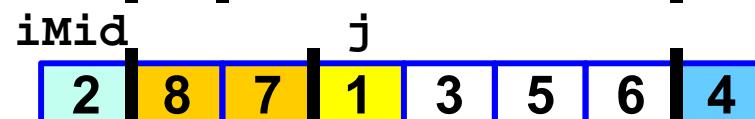
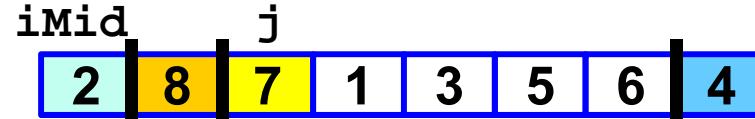
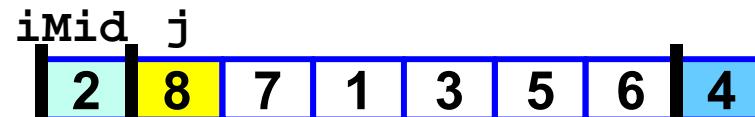
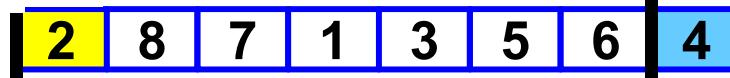


Prvek ke  
zpracování

Dosud  
nezpracované

```
int pivot = a[iR];
int iMid = iL-1;
for(int j = iL; j < iR; j++)
    if (a[j] <= pivot)
        swap(a, ++iMid, j);
swap(a, ++iMid, R);
return iMid;
```

iMid    j



## Select metodou medián z mediánů

Randomized Select se od Quicksortu liší hlavně tím, že nevolá rekurzi na úsek s "malými" a "velkými" hodnotami, ale jen na jeden z nich, podle toho, v kterém z nich lze očekávat výskyt hledaného  $k$ -tého nejmenšího prvku.

Jak si pamatujeme z Quicksortu, pokud je úsek "malých" nebo "velkých" hodnot opakováně extrémně krátký, může to vést až na kvadratickou složitost Quicksortu.

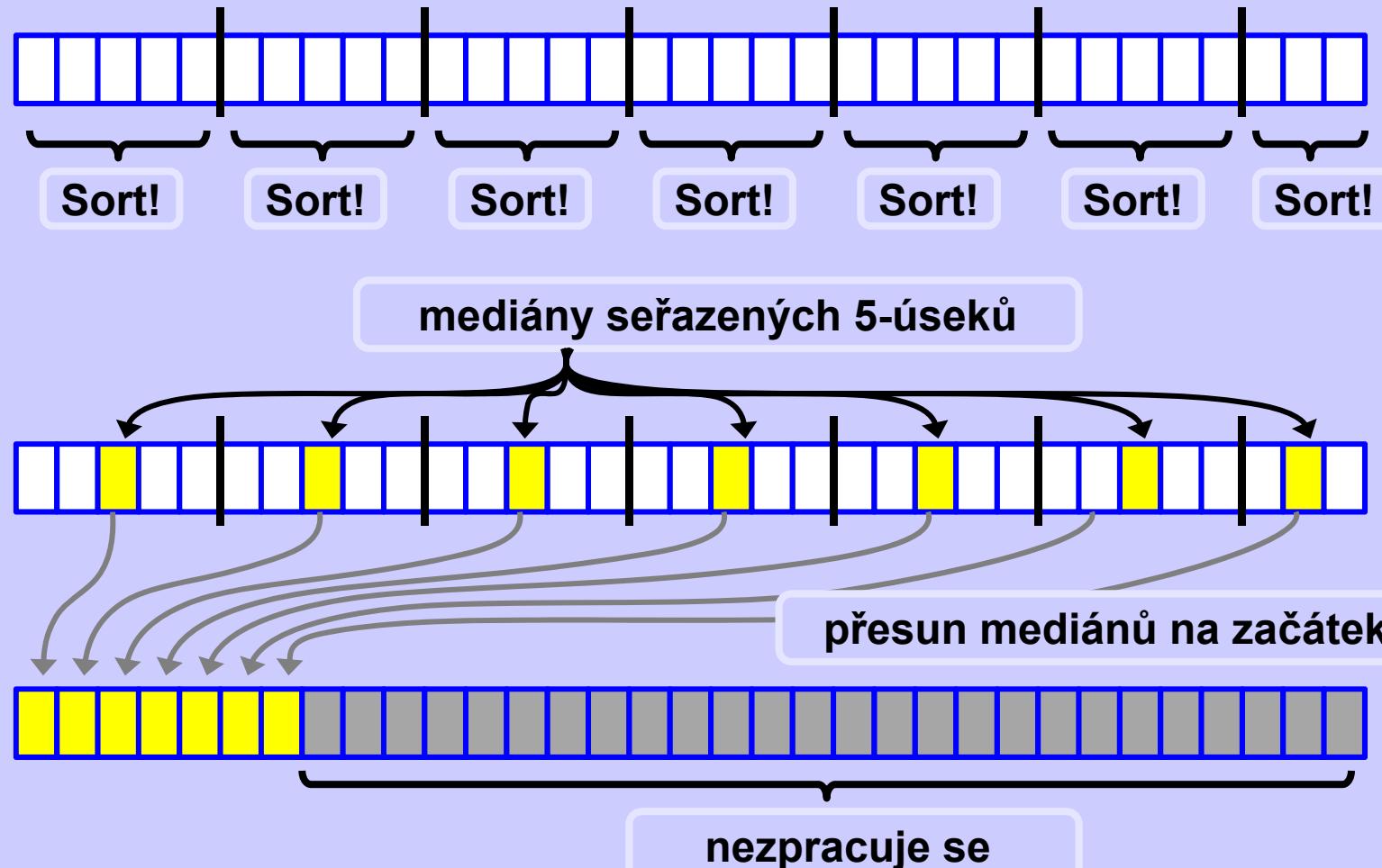
Stejná potíž by mohla nastat i metodě select.

Je nutno zajistit, aby v metodě partition byl vybrán pokaždé dostatečně vhodný pivot, tj. takový, který dobře rozdělí daný úsek na "malé" a "velké", tj. takový, který je pokud možno co nejblíže mediánu hodnot tohoto úseku.

## Select metodou medián z mediánů

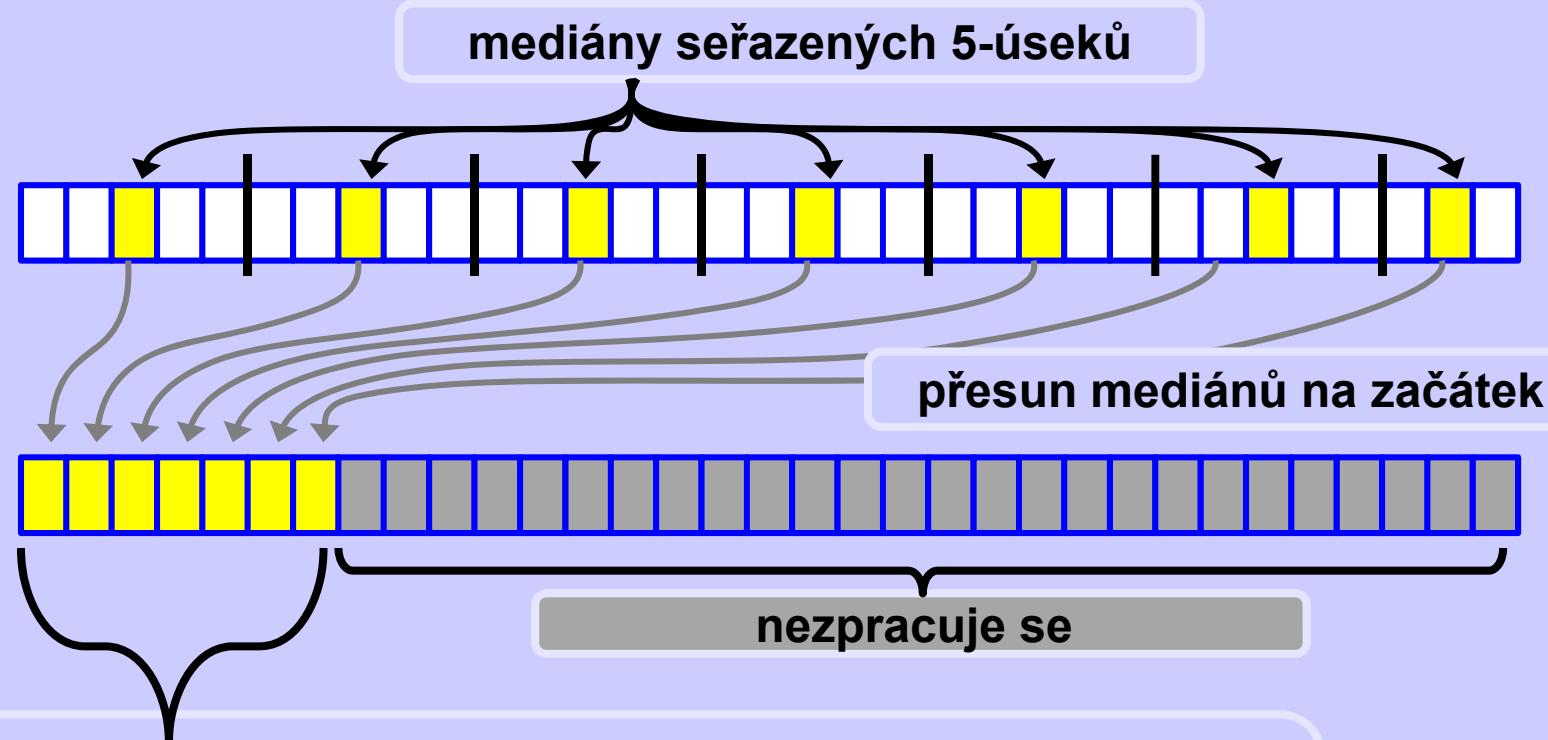
Volba pivota pro funkci Partition -- ukázka pro pole délky 33

Úseky délky 5



## Select metodou medián z mediánů

Volba pivota pro funkci Partition -- ukázka pro pole délky 33



V této části rekurzivní aplikací právě popisované metody najdi medián. To je "medián z mediánů" pro původní pole. Tento medián použij jako pivot ve funkci Partition.

## Select metodou medián z mediánů

0. Pokud má aktuální úsek pole délku 1, vrat' jeho jediný prvek.
1. Rozděl pole nebo úsek pole délky  $N$  na  $\lfloor N/5 \rfloor$  skupin po pěti prvcích a nejvýše jednu zbylou s méně než 5 prvky.
2. Každou skupinu seřaď (např. Insert sort) a vyber z ní její medián. Všechny tyto mediány přesuň na začátek pole/úseku.
3. Aplikuj rekurzivně celý postup na úsek obsahující právě nalezené mediány, tak získáš medián původních  $\lceil N/5 \rceil$  mediánů z bodu 2. Tento medián  $M$  bude pivotem.
4. Rozděl pole/úsek na "malé" a "velké" pomocí metody partition s pivotem  $M$ .
5. Podle velikosti úseků "malých" a "velkých" rozhodni, v kterém z nich se nachází  $k$ -tý nejmenší prvek celého pole a na něj aplikuj rekurzivně celý postup.

## Select metodou medián z mediánů

## Rozbor složitosti

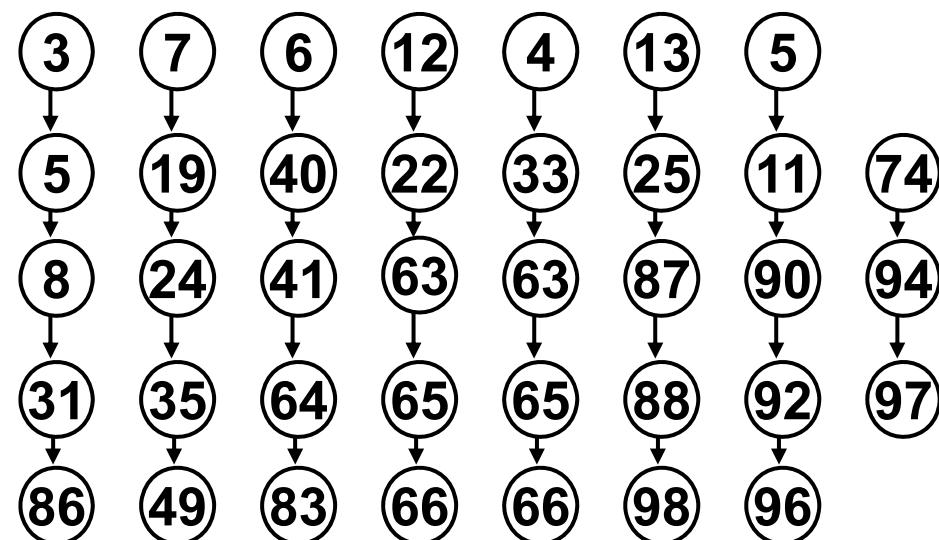
Rozděl na úseky délky 5

|   |   |    |    |   |    |    |   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |   |    |    |    |
|---|---|----|----|---|----|----|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|---|----|----|----|
| 5 | 3 | 86 | 31 | 8 | 24 | 19 | 7 | 49 | 35 | 64 | 41 | 40 | 6 | 83 | 22 | 12 | 66 | 65 | 63 | 66 | 33 | 63 | 65 | 4 | 13 | 98 | 88 | 87 | 25 | 90 | 11 | 96 | 92 | 5 | 94 | 74 | 97 |
|---|---|----|----|---|----|----|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|---|----|----|----|

Seřad' úseky délky 5

|   |   |   |    |    |   |    |    |    |    |   |    |    |    |    |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |   |    |    |    |    |    |    |    |
|---|---|---|----|----|---|----|----|----|----|---|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|
| 3 | 5 | 8 | 31 | 86 | 7 | 19 | 24 | 35 | 49 | 6 | 40 | 41 | 64 | 83 | 12 | 22 | 63 | 65 | 66 | 4 | 33 | 63 | 65 | 66 | 13 | 25 | 87 | 88 | 98 | 5 | 11 | 90 | 92 | 96 | 74 | 94 | 97 |
|---|---|---|----|----|---|----|----|----|----|---|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|

Seřazené úseky délky 5 pro úsporu místa a pro větší názornost si nakreslíme vertikálně a zachováme jejich vzájemné pořadí.



## Select metodou medián z mediánů

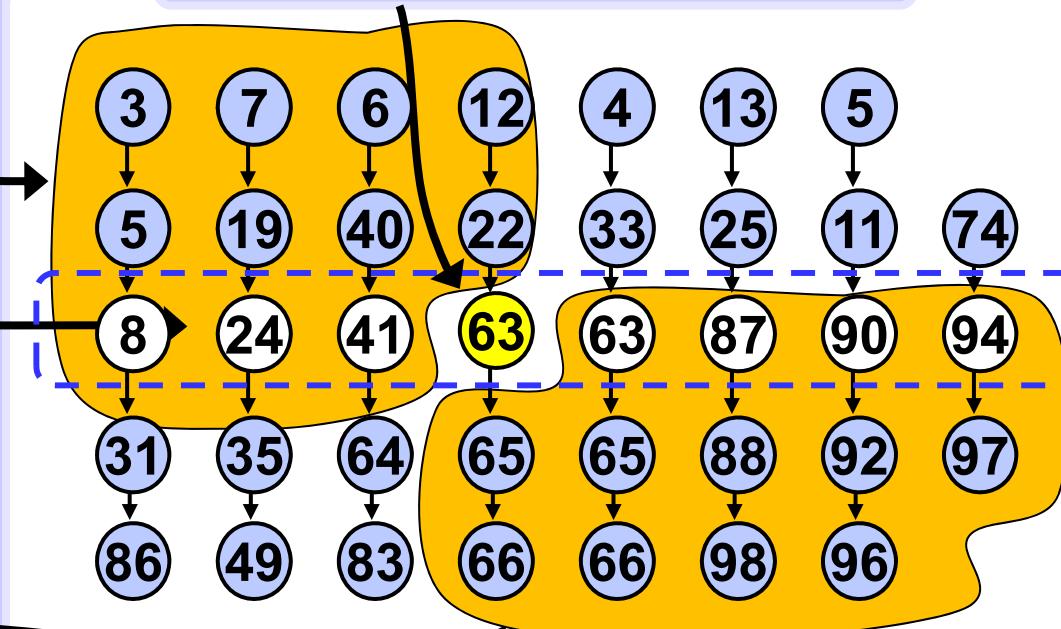
## Rozbor složitosti

$M = \text{medián z } \lceil N/5 \rceil \text{ mediánů}$

Prvky zaručeně menší než  $M$

$\lceil N/5 \rceil$  mediánů

Prvky zaručeně větší než  $M$



Prvků zaručeně větších než  $M$  je alespoň  $3N/10 - 6$ .

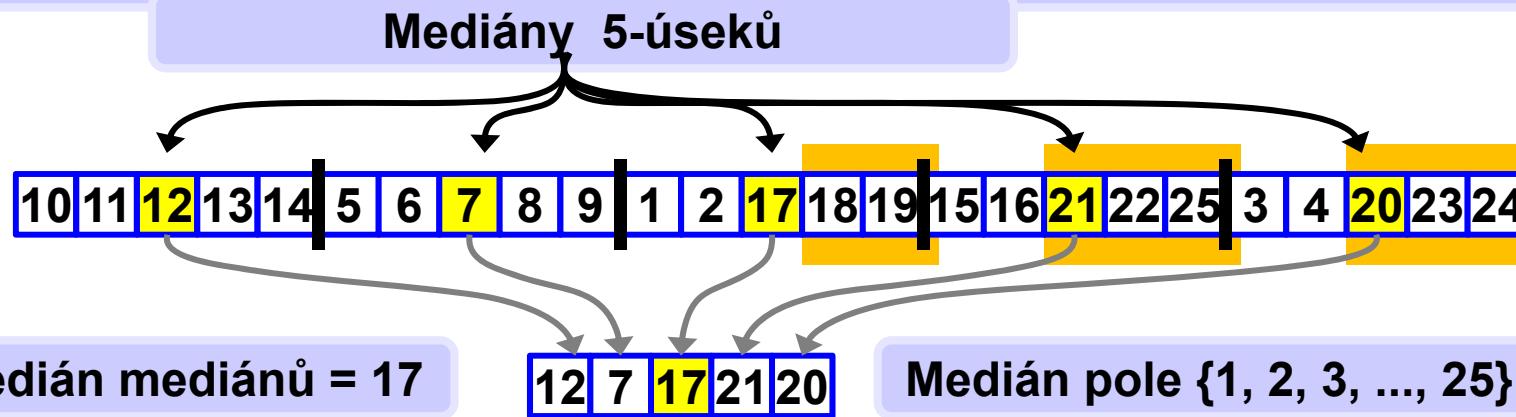
Analogicky, prvků menších než  $M$  je také alespoň  $3N/10 - 6$ .

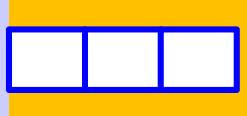
V bodě 5 se rekurze zavolá na úsek obsahující nejvýše  $N - (3N/10 - 6) = 7N/10 + 6$  prvků. Je tak zaručeno, že v nejhorším případě se úsek zkrátí o alespoň cca 30%.

## Select metodou medián z mediánů

## Rozbor složitosti

Medián mediánů není obecně roven mediánu celého souboru, nemůže být ale ani příliš velký ani malý.



Zvýrazněné hodnoty  nemohou být menší než medián mediánů. Těchto hodnot je celkem 8. Analogické zjištění platí pro hodnoty menší než medián mediánů. Pro pole délky 25 s různými prvky je zaručeno, že vybraný pivot bude co do velikosti mezi 9. -- 17. prvkem včetně, tj.  $16/25 = 64\%$  nevhodných voleb pivota odpadá.

## Select metodou medián z mediánů

Metoda subSort je obyčejný Insert Sort,  
rychlý na malých úsecích pole.

Zde slouží k nalezení mediánu hodnot v úseku délky 5 nebo menší.

```
void subSort(int [] a, int L, int R) {  
    int insVal, j;  
    for(int i = L+1; i <= R; i++) {  
        insVal = a[i];  
        j = i-1;  
        while ((j >= L) && (a[j] > insVal)) {  
            a[j+1] = a[j];  
            j--;  
        }  
        a[j+1] = insVal;  
    }  
}
```

## Select metodou medián z mediánů

Metoda select volá sama sebe dvakrát.

Poprvé pro nalezení mediánu z mediánů úseků délky 5,  
podruhé pro hledání k-tého prvku celého pole  
mezi buďto "malými nebo "velkými" hodnotami v aktuálním úseku.

```
int select(int [] a, int iL, int iR, int k) {  
    // 0. if the group is short terminate the recursion  
  
    // 1. move medians of groups of 5 to the left  
  
    // 2. also the median of the last possibly smaller group  
  
    // 3. find median M of medians of 5 recursively  
  
    // 4. use M for partition  
  
    // 5. recursively call select on the appropriate part  
}
```

## Select metodou medián z mediánů

```

int select(int [] a, int iL, int iR, int k) {

    // 0. if the group is short terminate the recursion
    if (iL == iR) return a[iL];
    if (iL+1 == iR) return (a[iL] < a[iR]) ? a[iL] : a[iR];

    // 1. move medians of groups of 5 to the left
    int lastM = iL;                      // place for medians
    int i5;                                // 5-elem groups index
    for (i5 = iL; i5 <= R-4; i5 += 5) {
        subSort(a, i5, i5+4);             // find median of 5
        swap(a, lastM++, i5+2);           // put it to the left
    }

    // 2. also the median of the last possibly smaller group
    if (i5 <= iR) {
        subSort(a, i5, iR);
        swap(a, lastM, i5+(iR-i5)/2); // lastM - last of medians
    }
    ... // continue
}

```

## Select metodou medián z mediánů

```
... // continued:  
  
// 3. find median (medmed5) of medians of 5 recursively  
int medmed5 = select(a, iL, lastM, iL+(lastM-iL)/2);  
  
// 4. use medmed5 for partition,  
//     a[iBig] = leftmost "big" value  
int iBig = partition(a, iL, iR, medmed5);  
  
// 5. recursively call select on the appropriate part  
if (iBig > k)  
    // the k-th elem is among the "small"  
    return select(a, iL, iBig-1, k);  
else  
    // the k-th elem is among the "big"  
    return select(a, iBig, iR, k);  
  
} // end of select
```

## Select metodou medián z mediánů

### Odvození složitosti

Rekurence:

$$T(N) \leq \begin{cases} O(1) & \text{pro } N < 140 \\ T(\lceil N/5 \rceil) + T(7N/10+6) + O(N) & \text{pro } N \geq 140 \end{cases}$$

**bod 3**      **bod 5**      **body 1, 2, 4**

Řešení substituční metodou, lze ukázat

$$T(N) \leq cN \in \Theta(N),$$

pro dostatečně velké  $c$  a pro  $N > 0$ .

Viz [CLRS], kap.9.

## ALG 14

**Vícedimenziونální data**

**Řazení vícedimenziونálních dat**

**Experimentální porovnání řadících algoritmů  
na vícedimenziونálních datech**

## Vícedimenziólní data

|      |
|------|
| 2.4  |
| 1.7  |
| 0.2  |
| 3.1  |
| -1.1 |
| 3.0  |



$$d = 6$$

Datový prvek je vektor  
délky  $d$  (= dimenze vektoru)

| 1    | 2    | 3    | 4    | 5 | 6 | ... |
|------|------|------|------|---|---|-----|
| 2.4  | 0.4  | -0.1 | 3.2  |   |   |     |
| 1.7  | 1.2  | -0.1 | 2.7  |   |   |     |
| 0.2  | 4.9  | -0.2 | 2.4  |   |   |     |
| 3.1  | 6.2  | 3.2  | -3.2 |   |   |     |
| -1.1 | 9.0  | 5.6  | -0.2 |   |   |     |
| 3.0  | -0.1 | -1.1 | 0.9  |   |   |     |

Řazené pole obsahuje  
(typicky) vektory stejné délky

|      |
|------|
| 2.4  |
| 1.7  |
| 0.2  |
| 3.1  |
| -1.1 |
| 3.0  |



|      |
|------|
| 2.4  |
| 1.7  |
| 0.2  |
| 3.2  |
| -5.4 |
| 2.9  |



|      |
|------|
| 2.4  |
| 1.8  |
| 0.1  |
| 3.1  |
| -5.3 |
| 2.8  |

Dvojici vektorů porovnáváme  
po složkách od začátku,  
první neshodná dvojice složek  
určuje pořadí vektorů.

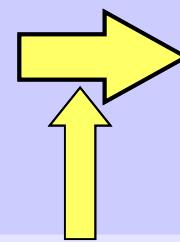
## Přímé a neefektivní řazení vícedimenzionálních dat

Vstupní pole

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 4 | 0 | 4 | 2 | 4 | 3 | 0 |
| 0 | 1 | 0 | 4 | 3 | 3 | 4 |
| 4 | 2 | 4 | 0 | 4 | 1 | 4 |
| 3 | 4 | 4 | 2 | 3 | 3 | 3 |

Seřazené pole

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 2 | 3 | 4 | 4 | 4 |
| 1 | 4 | 4 | 0 | 1 | 4 | 0 |
| 2 | 4 | 4 | 3 | 4 | 3 | 4 |
| 4 | 3 | 3 | 2 | 3 | 3 | 3 |



Řadící algoritmus

Klad



Bezprostřední přístup k datům  
je rychlý.

Zápor



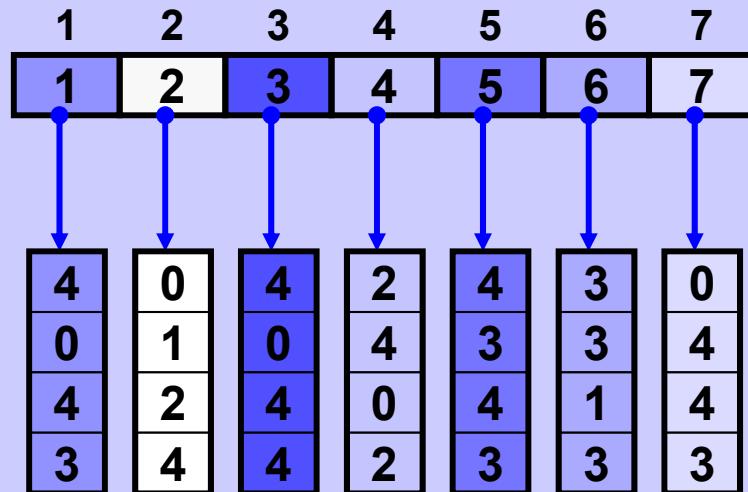
Výměna prvků je pomalá,  
musí se fyzicky přesouvat.



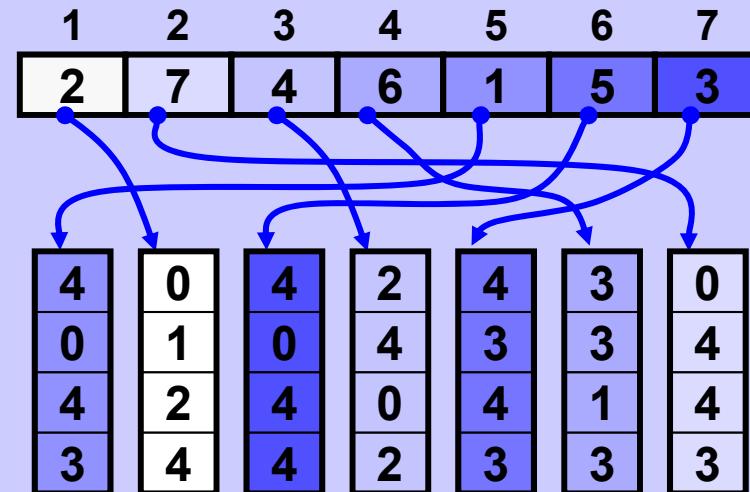
Zápor převažuje

## Řazení vícedimenziorních dat s pomocnými ukazateli

Pole pomocných ukazatelů  
na datové prvky



Řadíme pouze ukazatele podle  
velikosti jím odpovídajících dat.



Klad



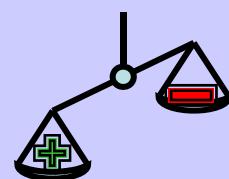
Výměna prvků je rychlá,  
děje se pouze výměnou  
ukazatelů.

Klad převažuje

Zápor



Přístup k datům pomocí  
ukazatelů je pomalejší.



## Pomocná funkce pro porovnání dvou vektorů

```
class L {  
  
    int compare( int[] a, int[] b) {  
        for( int i = 0; i < a.length; i++) {  
            if (a[i] == b[i]) continue;  
            if (a[i] < b[i]) return -1;  
            else return +1;  
        }  
        return 0;  
    }  
    ...  
}
```

## Insert sort pro 1D pole (opakování)

```
void insertSort (int [] a, int low, int high) {  
  
    int insVal, j;  
    for (int i = low+1; i <= high; i++) {  
  
        // find & make place for a[i]  
        insVal = a[i];  
        j = i-1;  
        while ((j >= low) && (a[j] > insVal)) {  
            a[j+1] = a[j];  
            j--;  
        }  
        // insert a[i]  
        a[j+1] = insVal;  
    }  
}
```

## Insert sort pro řazení pole vektorů s využitím ukazatelů

```
void insertSort( int[][] a, int[] ptrs, int low, int high){  
  
    int j;  
    int insValPtr;  
    for (int i = low+1; i <= high; i++) {  
        // find & make place for a[i]  
        insValPtr = ptrs[i];  
        j = i-1;  
        while ((j >= low) &&  
               (L.compare(a[ptrs[j]], a[insValPtr]) == 1)) {  
            ptrs[j+1] = ptrs[j];  
            j--;  
        }  
        // insert a[i]  
        ptrs[j+1] = insValPtr;  
    }  
}
```

## Merge sort pro řazení pole vektorů s využitím ukazatelů I

```
void mergeSortCore (int[][][] data, int ptr1[], int ptr2[],
                    int low, int high){
    int half = (low+high)/2;
    int i;
    if (low >= high) return;           // too small!
   // sort:
    mergeSortCore(data, ptr2, ptr1, low, half);   // left
    mergeSortCore(data, ptr2, ptr1, half+1, high); // right
    merge(data, ptr2, ptr1, low, high);           // merge halves
}

void mergeSort (int[][][] data, int [] ptr1) {
    initPtrs(ptr1);                  // init pointers
    int [] ptr2 = new int [data.length];
    initPtrs(ptr2);
    mergeSortCore(data, ptr1, ptr2, 0, data.length-1);
}
```

## Merge sort pro řazení pole vektorů s využitím ukazatelů II

```
void merge( byte [][] data,  int inptr[],  int outptr[],
            int low,  int high) {
    int half = (low+high)/2;
    int i1 = low;
    int i2 = half+1;
    int j = low;
                                // compare and merge
    while ((i1 <= half) && (i2 <= high))
        if ( compare(data[inptr[i1]], data[inptr[i2]]) <= 0 )
            outptr[j++] = inptr[i1++];
        else outptr[j++] = inptr[i2++];
                                // copy the rest
    while (i1 <= half) outptr[j++] = inptr[i1++];
    while (i2 <= high) outptr[j++] = inptr[i2++];
}
```

## Quick sort pro řazení pole vektorů s využitím ukazatelů II

```

void sortQp( int[][] a, int[] ptrs, int low, int high) {
    int iL = low, iR = high, aux;
    int pivotPtr = ptrs[low];

    do {
        while (L.compare(a[ptrs[iL]], a[pivotPtr]) == -1) iL++;
        while (L.compare(a[ptrs[iR]], a[pivotPtr]) == 1) iR--;
        if (iL < iR) {                                //swap(a,iL,iR)
            aux = ptrs[iL]; ptrs[iL] = ptrs[iR]; ptrs[iR] = aux;
            iL++; iR--;
        }
        else
            if (iL == iR) { iL++; iR--;}

    } while(iL <= iR);

    if (low < iR) sortQp(a, ptrs, low, iR);
    if (iL < high) sortQp(a, ptrs, iL, high);
}

```

## Knihovní funkce řazení

Příklad použití zabudovaného řazení v Javě  
pro řazení pole celočíselných vektorů libovolné dimenze

```
class MyCompar implements Comparator {
    public int compare( Object obj1, Object obj2 ) {
        int [] a = (int []) obj1;
        int [] b = (int []) obj2;

        for( int i = 0; i < a.length; i++ ) {
            if (a[i] == b[i]) continue;
            if (a[i] < b[i]) return -1;
            else return +1;
        }
        return 0;
    } // end of class

// usage:
int [][] MydataArray = ... ; // any initialization
Arrays.sort(MydataArray, new MyCompar());
```

## Ilustrační experiment řazení

### Prostředí

Intel(R) 2.7 GHz, Microsoft Windows 7 Enterprise, SP1, version 6.1  
Java: 1.7.0\_51; Java HotSpot(TM) 64-Bit Server VM 24.51-b03  
L2CacheSize 256 KB, L3CacheSize 4096 KB.

### Organizace

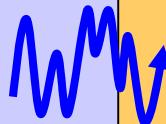
Pole celých čísel z intervalu <0,127> náhodně zamíchána generátorem pseudonáhodných čísel s rovnoměrným rozložením.  
Výsledky průměrovány přes větší počet běhů.

### Závěr

Rozhodně neexistuje jedno univerzální řazení, včetně prostředků poskytovaných ve standardních knihovnách jazyka,  
které by bylo optimální za všech okolností.

Hraje roli velikost, dimenzionalita i stupeň předběžného seřazení dat.

## Výsledky experimentu



| Náhodně uspořádaná data |           | Doba běhu v ms |         |        |        |         |
|-------------------------|-----------|----------------|---------|--------|--------|---------|
|                         |           | Délka pole     |         |        |        |         |
| Délka vektoru           | Insert    | 0.0003         | 0.010   | 1.00   | ~100   | ~20 000 |
|                         | Quick     | 0.0004         | 0.012   | 0.15   | 2.0    | 26.9    |
|                         | Merge     | 0.0007         | 0.007   | 0.12   | 1.7    | 23.0    |
|                         | Java      | 0.0004         | 0.010   | 0.13   | 1.8    | 26.0    |
|                         | Radix     | 0.0006         | ★ 0.003 | ★ 0.01 | ★ 0.14 | ★ 4.2   |
|                         | Insert    | ★ 0.0003       | 0.011   | 1.00   | ~100   | ~25 000 |
| Délka vektoru           | Quick     | 0.0006         | 0.013   | 0.16   | 2.2    | 32.9    |
|                         | Merge     | 0.0004         | ★ 0.007 | 0.12   | 1.7    | ★ 25.5  |
|                         | Java      | ★ 0.0003       | 0.009   | 0.13   | 1.9    | 30.0    |
|                         | Radix     | 0.0025         | 0.015   | ★ 0.08 | ★ 1.0  | 50.0    |
|                         | Insert    | 0.0004         | 0.011   | 1.00   | ~100   | ~30 000 |
| Délka vektoru           | Quick     | 0.0020         | 0.031   | 0.27   | 3.3    | 49.4    |
|                         | Merge     | 0.0004         | 0.011   | ★ 0.12 | ★ 1.9  | ★ 31.4  |
|                         | Java      | ★ 0.0002       | ★ 0.005 | 0.13   | 2.0    | 40.1    |
|                         | Radix     | 0.0275         | 0.154   | 0.90   | 12.2   | ~1000   |
|                         | nesoutěží |                |         |        |        |         |

## Výsledky experimentu

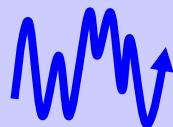


| Vzestupně<br>uspoř. data<br>s 10% šumem |        | Doba běhu v ms |         |        |        |         |
|-----------------------------------------|--------|----------------|---------|--------|--------|---------|
|                                         |        | Délka pole     |         |        |        |         |
|                                         |        | 10             | 100     | 1 000  | 10 000 | 100 000 |
| Délka<br>vektoru<br>2                   | Insert | ★ 0.0001       | 0.003   | 0.10   | 9.5    | ~100    |
|                                         | Quick  | 0.0005         | 0.010   | 0.20   | 2.5    | 30.0    |
|                                         | Merge  | 0.0002         | 0.004   | 0.06   | 0.7    | 9.9     |
|                                         | Java   | 0.0002         | 0.004   | 0.05   | 0.5    | 7.9     |
|                                         | Radix  | 0.0008         | ★ 0.003 | ★ 0.02 | ★ 0.1  | ★ 1.8   |
| Délka<br>vektoru<br>10                  | Insert | ★ 0.0001       | 0.004   | 0.11   | 9.7    | ~100    |
|                                         | Quick  | 0.0005         | 0.015   | 0.22   | 2.9    | 39.1    |
|                                         | Merge  | 0.0003         | ★ 0.004 | 0.06   | 0.8    | 10.3    |
|                                         | Java   | 0.0002         | 0.005   | ★ 0.05 | ★ 0.6  | ★ 9.1   |
|                                         | Radix  | 0.0027         | 0.015   | 0.08   | 1.0    | 47.2    |
| Délka<br>vektoru<br>100                 | Insert | ★ 0.0002       | ★ 0.004 | 0.13   | 10.5   | ~100    |
|                                         | Quick  | 0.0015         | 0.023   | 0.35   | 4.3    | 73.0    |
|                                         | Merge  | 0.0005         | ★ 0.004 | ★ 0.06 | 0.8    | 12.5    |
|                                         | Java   | 0.0004         | 0.008   | ★ 0.06 | ★ 0.7  | ★ 11.5  |
|                                         | Radix  | 0.0260         | 0.150   | 1.36   | 12.4   | ~1000   |

nesoutěží

## Výsledky experimentu

### Zjištění závislá na stupni uspořádání dat



Náhodně uspořádaná data

Při střední nebo vysoké dimenzionalitě dat je výhodné použít vlastní implementaci Merge sortu, vůči knihovnímu řazení je typicky rychlejší, o jednotky až desítky procent, zejména se zvětšujícími se daty.

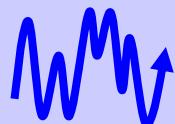


Vzestupně uspořádaná data s 10% šumem

Knihovní řazení detekuje stupeň uspořádání dat a maximálně jej využívá. V nejlepších případech dosahuje složitost  $\Theta(N)$ . Při vyšší dimenzionalitě a velkých datech bylo v experimentu mírně rychlejší než vlastní implementace Merge sortu, lze čekat, že při menším šumu v datech bude rozdíl výraznější.

## Výsledky experimentu

### Společná zjištění



Náhodně uspořádaná data



Vzestupně uspořádaná data s 10% šumem

Při velmi malém rozsahu dat (nejvýše desítky) nezávisle na dimenzi poskytuje standardně Insert sort nejlepší výkon. Také knihovní řazení Javy detekuje malá data a aplikuje Insert sort, výkon je tu srovnatelný.

Pokud lze použít Radix sort, je to výhodné zejména při nízké dimenzionalitě a velkém objemu dat, zrychlení vůči knihovnímu řazení jsme v tomto případě naměřili cca 5 až 10-i násobné.

Používání Quick sortu nespíše nelze ve většině případů doporučit.