IB002 Algoritmy a datové struktury I

Ivana Černá

Fakulta informatiky, Masarykova univerzita

Jaro 2018

Informace o předmětu

- vyučující předmětu
  - Ivana Černá *(přednášky)*
  - Vojtěch Řehák *(cvičení)*
  - Nikola Beneš, Tomáš Brázdil, Vlastimil Dohnal, Tomáš Masopust, Jan Obdržálek,
    František Blahoudek, Henrich Lauko, Dominik Velan, Radka Cieslarová, Jan Koniarik, Jan Horáček, Oliver Roch Kateřina Sloupová, Miloslav Staněk, Viktória Vozárová, Tatiana Zbončáková, Miriama Zemaníková *(vedení cvičení)*
  - Lucia Dupkaničová, Adam Matoušek,Mikoláš Stuchlík, Alena Zahradníčková, *(příprava cvičení)*

- **interaktivní osnova předmětu - kompletní informace**
  `is.muni.cz/auth/el/1433/jaro2018/IB002/index.qwarp`
  - **organizace výuky**: přednášky, cvičení, domácí úkoly, konzultace
  - **studijní materiály**: učebnice, slajdy z přednášky, skripta příkladů, zadání úkolů, rozcestníky
  - **hodnocení předmětu**: seminární odpovědníky, domácí úkoly a speciální domácí úkol, implementační a znalostní část zkoušky
- diskusní fórum předmětu v IS MU

Literatura

- T. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms*. Third Edition. MIT Press, 2009
- J. Kleinberg, and E. Tardos: *Algorithm Design*. Addison-Wesley, 2006
- S. Dasgupta, Ch. Papadimitriou, U. Vazirani: *Algorithms*. McGraw Hill, 2007.
- T. Cormen: *Algorithms Unlocked*. MIT Press, 2013
- J. Bentley: *Programming Pearls (2nd Edition)*. Addison-Wesley, 1999.

- online materiály (*odkazy viz Interaktivní osnova*)
- video přednášky (*doporučuji MIT 6.006 Intro to Algorithms*)

obrázky použité v prezentacích jsou částečně převzaty z uvedených monografií

Obsah

algoritmus způsob řešení problému

datová struktura způsob uložení informací

techniky návrhu a analýzy algoritmů

důkaz korektnosti algoritmu, analýza složitosti algoritmu, asymptotická notace, technika rozděl & panuj a rekurze

datové struktury

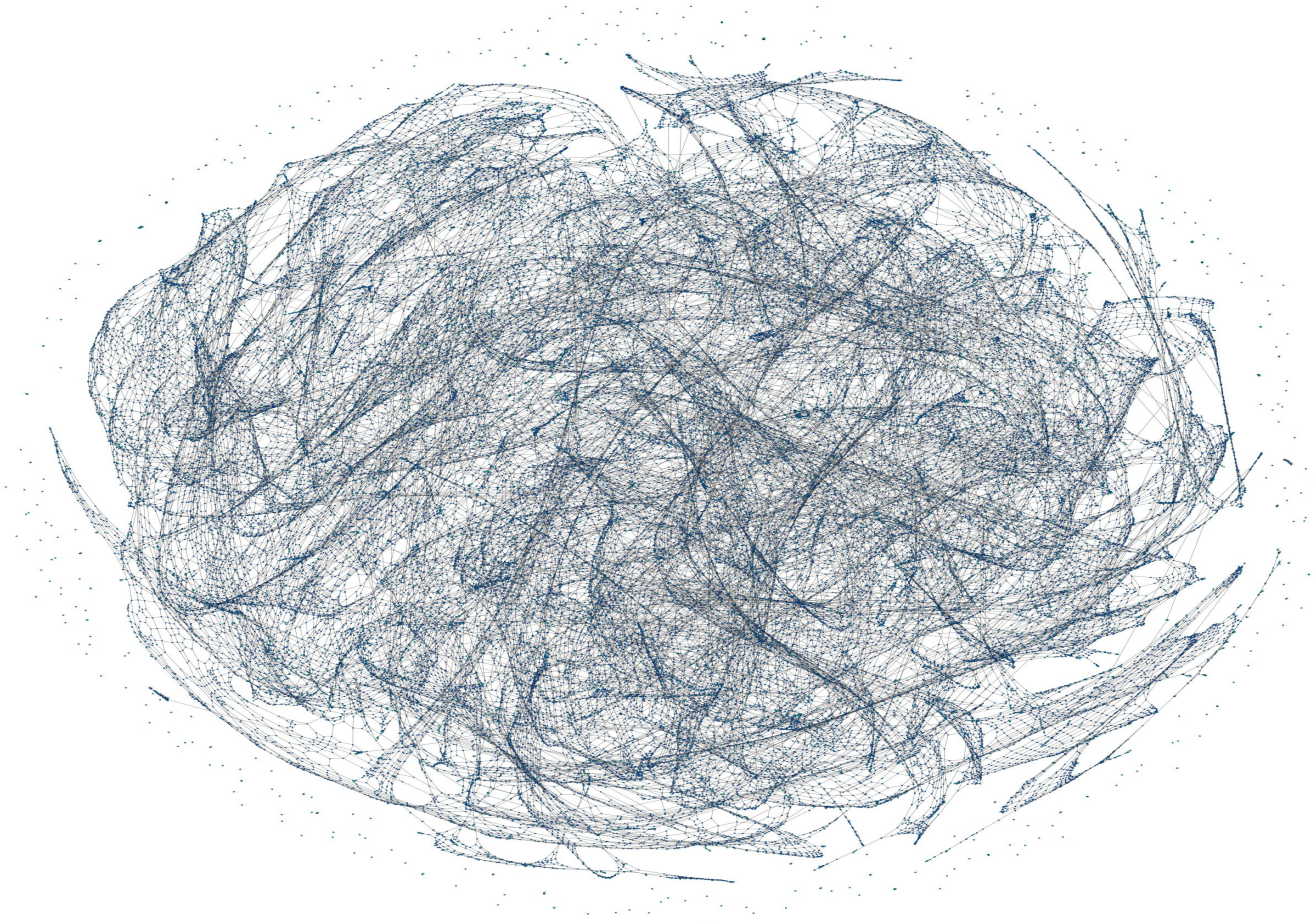
halda, prioritní fronta, vyhledávací stromy, červeno-černé stromy, B-stromy, hašovací tabulky

algoritmy řazení rozdělováním, slučováním, haldou, v lineárním čase

grafové algoritmy prohledávání grafu, souvislost, cesty

Motivace

řešit jinak neřešitelné problémy...



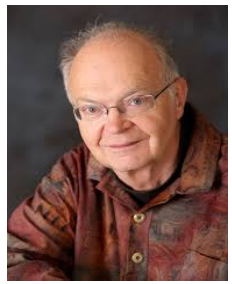
Chicago road networks, http://csun.uic.edu/dataviz.html

Motivace

naučit se něco nového...

An algorithm must be seen to be believed, and the best way to learn about what an algorithm is all about is to try it.

— Donald Knuth, The Art of Computer Programming



Algorithms: a common language for nature, human, and computer.

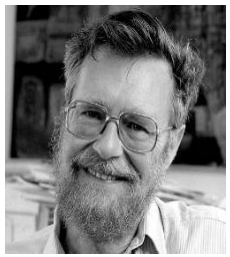
— Avi Wigderson

Motivace

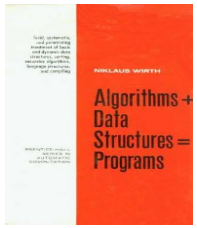
stát se dobrým programátorem...

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. — *Linus Torvalds (creator of Linux)*

Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code.
— *Edsger W. Dijkstra*



Algorithms + Data Structures = Programs.
— *Niklaus Wirth*



Motivace

široké uplatnění...

- návrh počítačů logické obvody, systém souborů, překladače
- Internet vyhledávání, distribuované sdílení, cloud computing, packet routing
- počítačová grafika virtuální realita, video, hry
- multimédia mp3, jpg, rozpoznávání obrazu
- bezpečnost šifrování, hlasování, e-obchod
- sociální sítě doporučení a predikce, reklama
- fyzika simulace
- biologie projekt lidského genomu, simulace

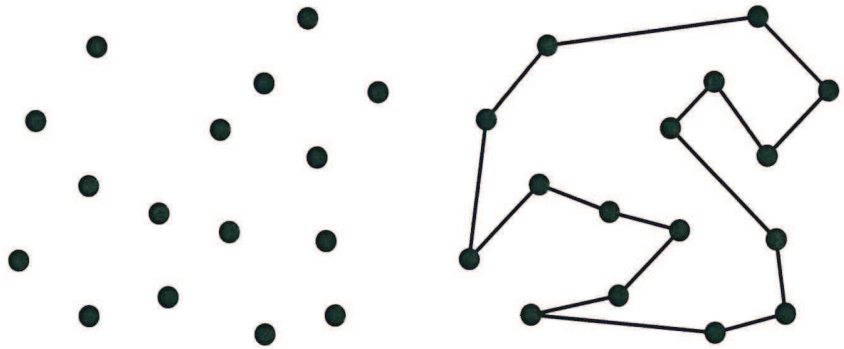
Motivace

pro zábavu a zisk...

Návrh a analýza algoritmů

Motivace

najdi nejkratší cestu pro rozvoz čerstvé pizzy



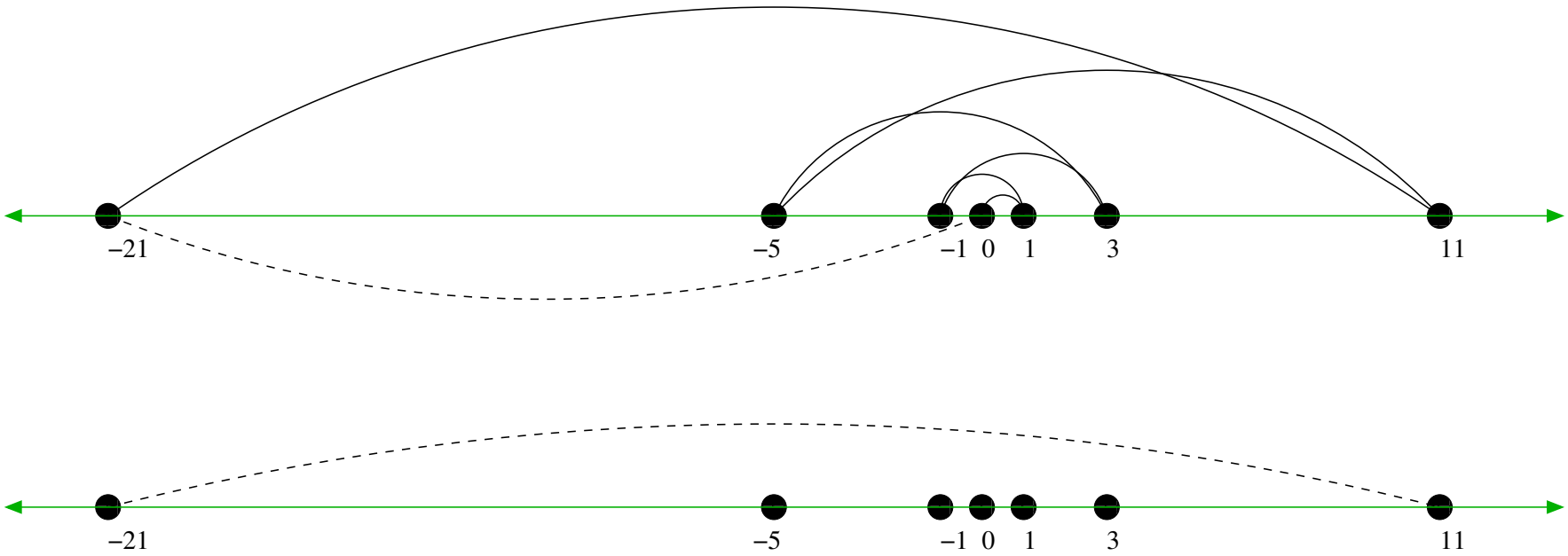
ALGORITMUS???

Řešení 1

vyber počáteční vrchol $v_0, i \leftarrow 0$
while existuje nenavštívený vrchol **do**
    $i \leftarrow i + 1$
    nechť p_i je nenavštívený vrchol nejblíž k p_{i-1}
    navštiv p_i **od**
return vrať cestu z p_0 do p_i

Řešení 1

vyber počáteční vrchol $v_0, i \leftarrow 0$
while existuje nenavštívený vrchol **do**
    $i \leftarrow i + 1$
    nechť p_i je nenavštívený vrchol nejblíž k p_{i-1}
    navštiv p_i **od**
return vrať cestu z p_0 do p_i

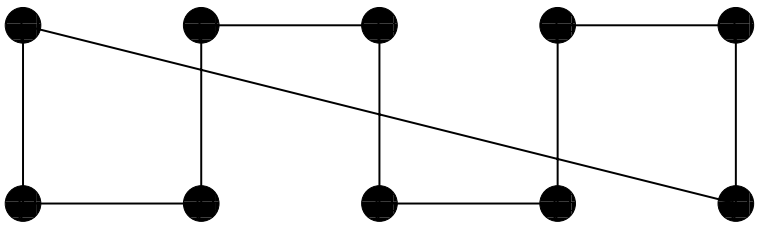


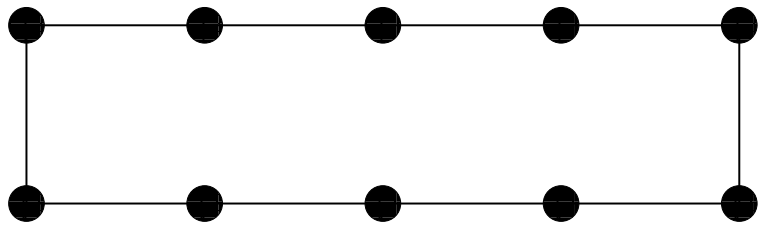
Řešení 2

nechť n je počet vrcholů
for $i = 1$ **to** $n - 1$ **do**
    $d \leftarrow \infty$
    for každou dvojici (x, y) koncových bodů částečných cest **do**
        if $dist(x, y) \leq d$ **then** $x_i \leftarrow x, y_i \leftarrow y, d \leftarrow dist(x, y)$ **fi od**
    spoj vrcholy x_i, y_i hranou **od**
spoj hranou koncové vrcholy cesty

Řešení 2

nechť n je počet vrcholů
for $i = 1$ **to** $n - 1$ **do**
    $d \leftarrow \infty$
    for každou dvojici (x, y) koncových bodů částečných cest **do**
        if $dist(x, y) \leq d$ **then** $x_i \leftarrow x, y_i \leftarrow y, d \leftarrow dist(x, y)$ **fi od**
    spoj vrcholy x_i, y_i hranou **od**
spoj hranou koncové vrcholy cesty





Korektní algoritmus

$d \leftarrow \infty$
for každou z $n!$ permutací Π_i vrcholů **do**
  if $cost(\Pi_i) \leq d$ $cost(\Pi_i) je\ cena\ cesty\ určené\ permutací\ \Pi_i$
    then $d \leftarrow cost(\Pi_i) \wedge P_{min} \leftarrow \Pi_i$ **fi od**
return P_{min}

korektnost algoritmus prověří všech $n!$ možných způsobů projití grafu

Korektní algoritmus

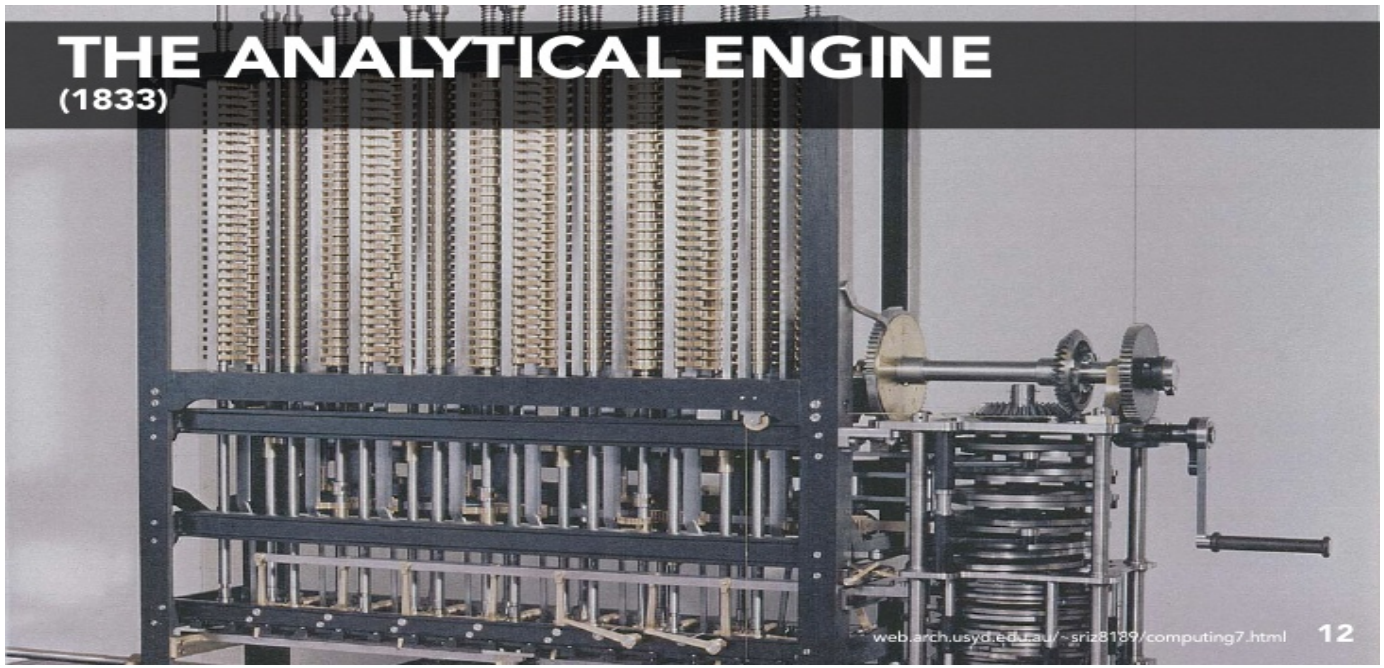
$d \leftarrow \infty$
for každou z $n!$ permutací Π_i vrcholů **do**
  if $cost(\Pi_i) \leq d$ $cost(\Pi_i) je\ cena\ cesty\ určené\ permutací\ \Pi_i$
    then $d \leftarrow cost(\Pi_i) \wedge P_{min} \leftarrow \Pi_i$ **fi od**
return P_{min}

korektnost algoritmus prověří všech $n!$ možných způsobů projití grafu
složitost algoritmus je nepoužitelný již pro velmi malé grafy
??? existuje efektivnější algoritmus ???

Složitost

Charles Babage, 1864

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any resul is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?



kolikrát musíme zatočit klikou?

Časová složitost

časová složitost **výpočtu** = součet cen všech vykonaných operací

časová složitost **algoritmu** je funkce délky vstupu

- složitost v nejhorším případě
  maximální délka výpočtu na vstupu délky n
- složitost v nejlepším případě
  minimální délka výpočtu na vstupu délky n
- průměrná složitost
  průměr složitostí výpočtů na všech vstupech délky n

složitost = časová složitost v nejhorším případě

Vyhledávání prvku v posloupnosti

Vstup posloupnost prvků $A[1 \dots n]$ a prvek x

Výstup index i takový, že $A[i] = x$, resp. hodnota No, jestliže prvek x se v posloupnosti nevyskytuje

Procedure Linear Search

```
1 answer ← No
2 for i = 1 to n do
3     if A[i] = x then answer ← i fi
4 od
5 return answer
```

Vyhledávání prvku v posloupnosti – optimalizace I

Procedure Better Linear Search

```
1 for i = 1 to n do
2     if A[i] = x then return i fi
3 od
4 return No
```

- první výskyt x ukončí prohledávání
- každý průchod cyklem znamená 2 testy: v řádku 1 testujeme rovnost $i \leq n$, v řádku 2 testujeme rovnost $A[i] = x$
- stačí 1 test?

Vyhledávání prvku v posloupnosti – optimalizace II

Procedure Sentinel Linear Search

```
1 last ← A[n]
2 A[n] ← x
3 i ← 1
4 while A[i] ≠ x do i ← i + 1 od
5 A[n] ← last
6 if i < n ∨ A[n] = x
7    then return i
8     else return No fi
```

- první výskyt x ukončí prohledávání
- sentinel (zarážka) pro případ, že pole neobsahuje prvek x
- každý průchod cyklem znamená 1 test
- 2 testy na závěr (řádek 6)

Časová složitost Linear Search

```
1 answer ← No
2 for i = 1 to n do
3     if A[i] = x
4         then answer ← i fi od
5 return answer
```

- označme t_i cenu operace na řádku i
- operace z řádků 1 a 5 se vykonají jednou
- řádek 2 se vykoná $n + 1$ krát
- řádek 3 se vykoná n krát
- přiřazení v řádku 4 se vykoná úměrně počtu výskytů x v poli

časová složitost v nejlepším případě

$$t_1 + t_2 \cdot (n + 1) + t_3 \cdot n + t_4 \cdot 0 + t_5$$

časová složitost v nejhorším případě

$$t_1 + t_2 \cdot (n + 1) + t_3 \cdot n + t_4 \cdot n + t_5$$

složitost je tvaru $c \cdot n + d$, kde c a d jsou konstanty nezávislé na n
složitost je lineární vzhledem k délce vstupu n

Časová složitost optimalizovaných algoritmů

Better Linear Search

1 **for** $i = 1$ **to** n **do if** $A[i] = x$ **then return** i **fi od**
2 **return** No

- časová složitost v nejhorším případě je lineární
- časová složitost v nejlepším případě je konstantní

Sentinel Linear Search

1 $last \leftarrow A[n], A[n] \leftarrow x, i \leftarrow 1$
2 **while** $A[i] \neq x$ **do** $i \leftarrow i + 1$ **od**
3 $A[n] \leftarrow last$
4 **if** $i < n \vee A[n] = x$ **then return** i **else return** No **fi**

- časová složitost v nejhorším případě je lineární
- časová složitost v nejlepším případě je konstantní
- rozdíl je v konstantních faktorech

Korektnost algoritmu

vstupní podmínka ze všech možných vstupů pro daný algoritmus vymezuje ty, pro které je algoritmus definován

výstupní podmínka pro každý vstup daného algoritmu splňující vstupní podmínku určuje, jak má vypadat výsledek odpovídající danému vstupu

algoritmus je (totálně) korektní jestliže pro každý vstup splňující vstupní podmínku výpočet skončí a výsledek splňuje výstupní podmínku

úplnost (konvergence) pro každý vstup splňující vstupní podmínku výpočet skončí

částečná (parciální) korektnost pro každý vstup, který splňuje vstupní podmínku a výpočet na něm skončí, výstup splňuje výstupní podmínku

Korektnost iterativního algoritmu

analyzujeme efekt jednotlivých operací

analýza efektu cyklu

- u vnořených cyklů začínáme od cyklu nejhlubší úrovně
- pro každý cyklus určíme jeho invariant
- invariantem cyklu je takové tvrzení, které platí před vykonáním a po vykonání každé iterace cyklu
- dokážeme, že invariant cyklu je pravdivý
- využitím invariantu
  - dokážeme konečnost výpočtu cyklu
  - dokážeme efekt cyklu

Invariant cyklu

Inicializace invariant je platný před začátkem vykonávání cyklu

Iterace jestliže invariant platí před iterací cyklu, zůstává v platnosti i po vykonání iterace

Ukončení cyklus skončí a po jeho ukončení platný invariant garantuje požadovaný efekt cyklu

Korektnost Better Linear Search

```
1 for i = 1 to n do if A[i] = x then return i fi od
2 return No
```

Invariant cyklu

Na začátku každé iterace cyklu platí, že jestliže prvek x se nalézá v A, tak se nalézá v části mezi pozicemi i a n.

Inicializace Na začátku je $i = 1$ a proto tvrzení platí.

Iterace Předpokládejme platnost tvrzení na začátku iterace.

Jestliže iterace nevrátí výslední hodnotu, tak $A[i] \neq x$. Proto x musí být na některé z pozic $i + 1$ až n a invariant zůstává v platnosti i po ukončení iterace (tj. před následující iterací).

Jestliže iterace vrátí hodnotu i, platnost tvrzení po ukončení iterace je zřejmá.

Ukončení Cyklus skončí buď proto, že je nalezena hodnota x anebo proto, že $i > n$. V obou případech z platnosti tvrzení po ukončení iterace cyklu plyne korektnost vypočítaného výsledku.

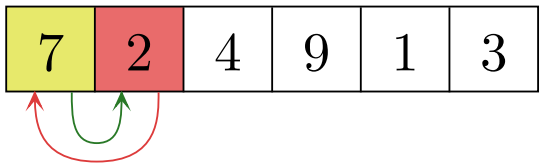
Problém řazení

Vstup posloupnost n čísel $(a_1, a_2, \ldots, a_n)$

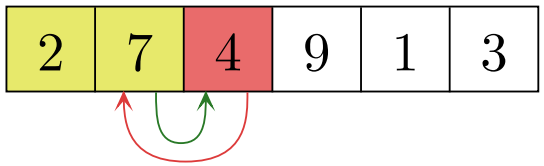
Výstup permutace (přeuspořádání) $(a'_1, a'_2, \ldots, a'_n)$ vstupní posloupnosti taková, že $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Princip řazení vkládáním

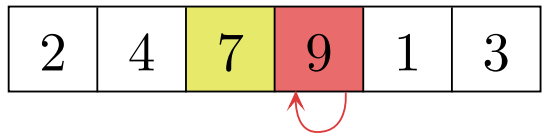
(a)



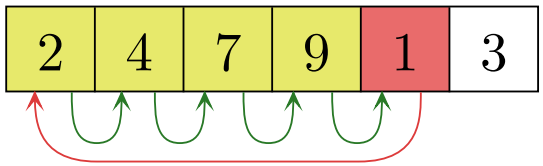
(b)



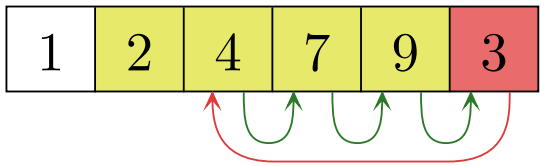
(c)



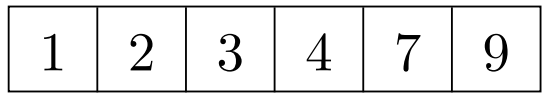
(d)



(e)



(f)



Algoritmus

Insert Sort(A)

```
1 //Vstup: A[1...n]
2 for j = 2 to A.length do
3     key ← A[j]
4     //  Vlož A[j] do seřazené postupnosti A[1...j − 1]
5     i ← j − 1
6     while i > 0 ∧ A[i] > key do
7           A[i + 1] ← A[i]
8           i ← i − 1 od
9     A[i + 1] ← key od
```

Korektnost řazení vkládáním

Invariant cyklu

Na začátku každé iterace **for** cyklu obsahuje pole $A[1 \dots j-1]$ stejné prvky jako na začátku výpočtu, ale seřazené od nejmenšího po největší.

Inicializace Před první iterací je $j = 2$ a tvrzení platí.

Iterace Předpokládejme, že tvrzení platí před iterací j, tj. prvky v $A[1 \dots j-1]$ jsou seřazeny. Jestliže $A[j] < A[j-1]$, tak prvky $A[j-1], A[j-2], A[j-3], \dots$ posouvají o jednu pozici doprava v těle cyklu se tak dlouho, až se najde vhodná pozice pro prvek $A[j]$ (ř. 5 - 7). Pole $A[1 \dots j]$ proto na konci iterace cyklu obsahuje stejné prvky jako na začátku, ale seřazené. Po navýšení hodnoty j zůstává tvrzení v platnosti.

Ukončení Cyklus skončí když $j > A.length = n$. Protože v každé iteraci se hodnota j navyšuje o 1, musí platit $j = n + 1$. Z platnosti invariantu cyklu plyne, že $A[1 \dots n]$ obsahuje stejné prvky jako na začátku výpočtu, ale seřazené.

Složitost řazení vkládáním

Insertion Sort(A)	cena	počet
1 **for** $j = 2$ **to** $A.length$ **do**	c_1	n
2 $key \leftarrow A[j]$	c_2	$n-1$
3 $i \leftarrow j-1$	c_3	$n-1$
4 **while** $i > 0 \wedge A[i] > key$ **do**	c_4	$\sum_{j=2}^{n} t_j$
5 $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6 $i \leftarrow i-1$ **od**	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $A[i+1] \leftarrow key$ **od**	c_7	$n-1$

t_j označuje počet opakování **while** cyklu pro danou hodnotu j

počet testů v hlavičce cyklu je o 1 vyšší než počet iterací cyklu

Složitost řazení vkládáním -nejlepší případ

Insertion Sort(A)	cena	počet	
1 **for** $j = 2$ **to** $A.length$ **do**	c_1	n	
2 $key \leftarrow A[j]$	c_2	$n-1$	
3 $i \leftarrow j-1$	c_3	$n-1$	
4 **while** $i > 0 \wedge A[i] > key$ **do**	c_4	$\sum_{j=2}^{n} t_j$	$t_j = 1$
5 $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$	
6 $i \leftarrow i-1$ **od**	c_6	$\sum_{j=2}^{n}(t_j - 1)$	
7 $A[i+1] \leftarrow key$ **od**	c_7	$n-1$	

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_4(n-1) + c_7(n-1)$$

$$= an + b$$

lineární složitost

Složitost řazení vkládáním -nejhorší případ

Insertion Sort(A)	**cena**	**počet**	
1 **for** $j = 2$ **to** $A.length$ **do**	c_1	n	
2 $key \leftarrow A[j]$	c_2	$n-1$	
3 $i \leftarrow j-1$	c_3	$n-1$	
4 **while** $i > 0 \wedge A[i] > key$ **do**	c_4	$\sum_{j=2}^{n} t_j$	$t_j = j$
5 $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$	
6 $i \leftarrow i-1$ **od**	c_6	$\sum_{j=2}^{n}(t_j - 1)$	
7 $A[i+1] \leftarrow key$ **od**	c_7	$n-1$	

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right)$$
$$+ c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$
$$= an^2 + bn + c$$

kvadratická složitost

Asymptotická notace

- asymptotickou notaci využíváme při popisu složitosti algoritmů
- umožňuje abstrahovat od detailů / zdůraznit podstatné

příklad

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n+1)}{2}\right) + c_6\left(\frac{n(n+1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

$$= an^2 + bn + c$$

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

Typy notací

- $f(n) = \mathcal{O}(g(n))$ znamená, že $C \times g(n)$ je **horní hranicí** pro $f(n)$
- $f(n) = \Omega(g(n))$ znamená, že $C \times g(n)$ je **dolní hranicí** pro $f(n)$
- $f(n) = \Theta(g(n))$ znamená, že $C_1 \times g(n)$ je horní hranicí pro $f(n)$ a $C_2 \times g(n)$ je dolní hranicí pro $f(n)$

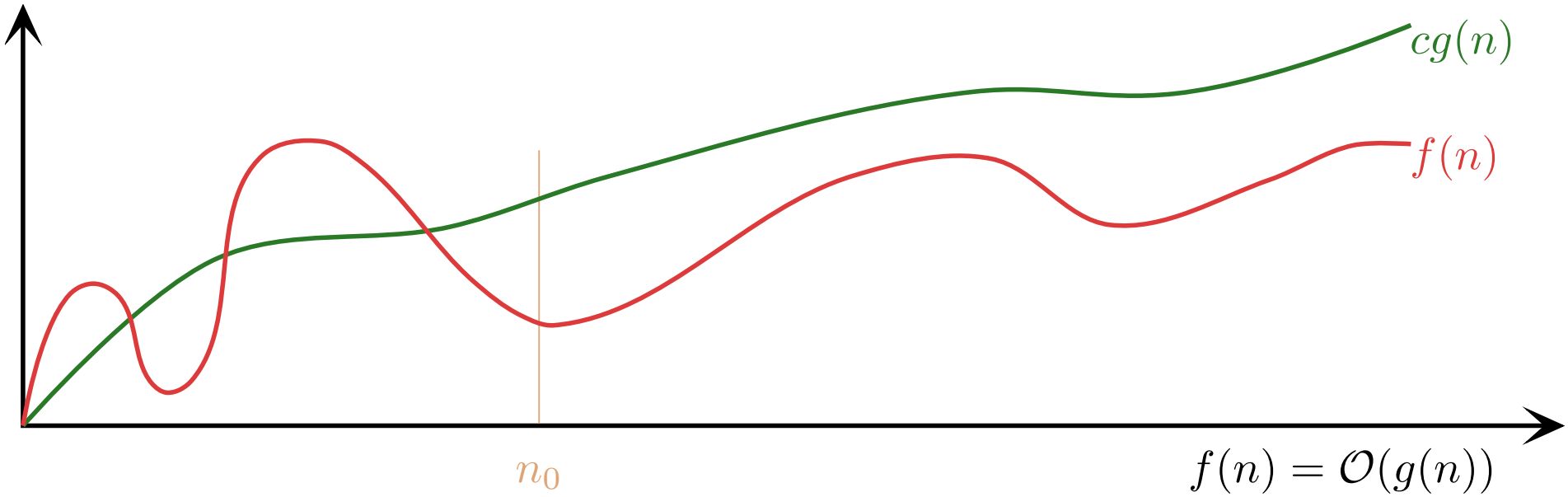
f, g jsou funkce, $f, g : \mathbb{N} \to \mathbb{N}$

C, C_1, C_2 jsou konstanty nezávislé na n

$\mathcal{O}$ notace

Definice

$f(n) = \mathcal{O}(g(n))$ právě když existují kladné konstanty n_0 a c takové, že pro všechna $n \geq n_0$ platí $f(n) \leq cg(n)$



- zápis $f(n) \in \mathcal{O}(g(n))$ vs zápis $f(n) = \mathcal{O}(g(n))$ (*historické důvody*)
- funkce $f(n)$ *neroste asymptoticky rychleji* než funkce $g(n)$
- alternativní definice $f(n) = \mathcal{O}(g(n))$ právě když $\limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$

$\mathcal{O}$ notace - příklady

- $8n^2 - 88n + 888 = \mathcal{O}(n^2)$

  protože $8n^2 - 88n + 888 < 8n^2$ pro všechna $n \geq 11$

- $8n^2 - 88n + 888 = \mathcal{O}(n^3)$

  protože $8n^2 - 88n + 8 < 1n^3$ pro všechna $n \geq 10$

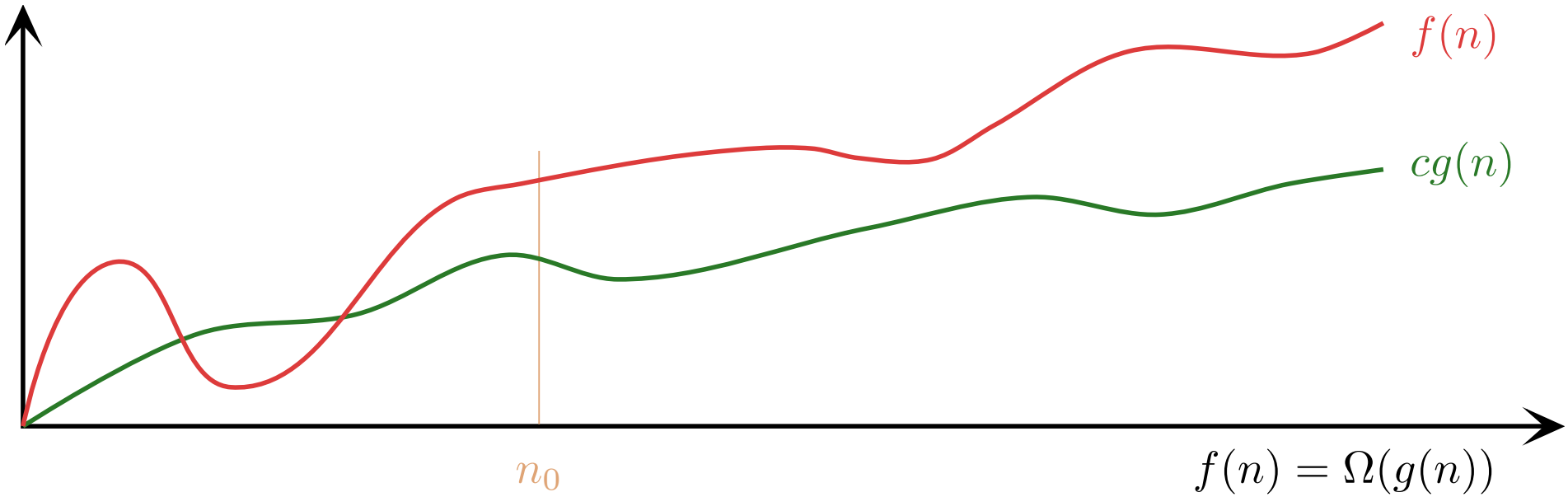
- $8n^2 - 88n + 888 \neq \mathcal{O}(n)$

  protože $cn < 8n^2 - 88n + 888$ pro $n > c$

Ω notace

Definice

$f(n) = \Omega(g(n))$ právě když existují kladné konstanty n_0 a c takové, že pro všechna $n \geq n_0$ platí $f(n) \geq cg(n)$



funkce $f(n)$ *neroste asymptoticky pomaleji* než funkce $g(n)$

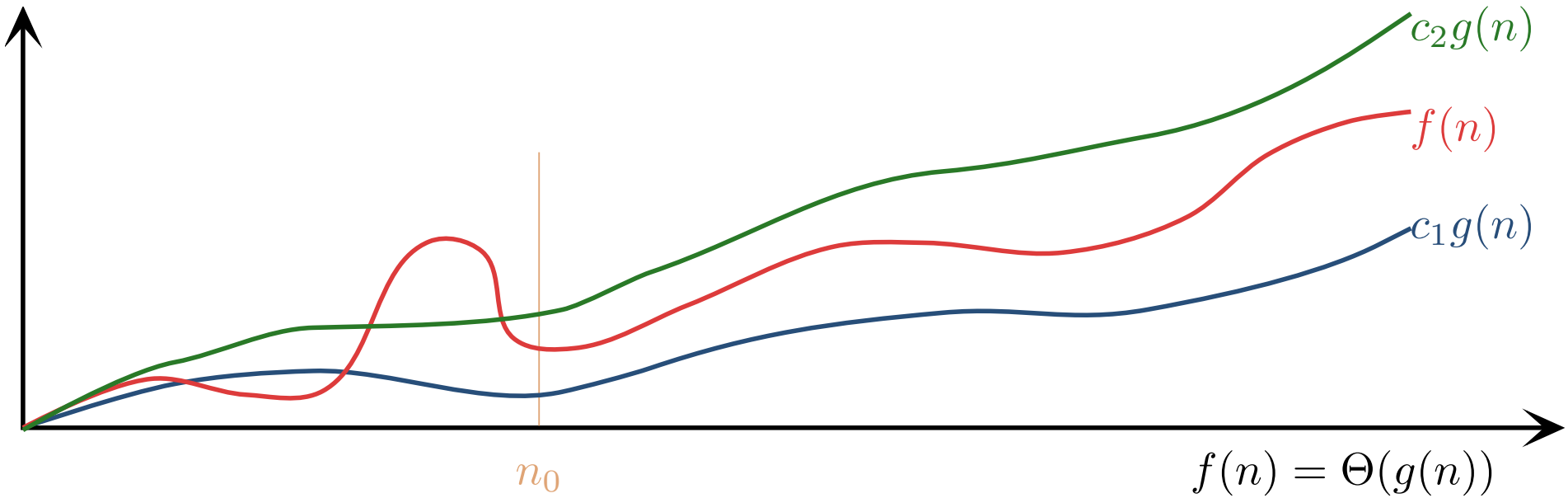
Ω notace - příklady

- $8n^2 - 88n + 8 = \Omega(n^2)$ protože $8n^2 - 88n + 8 > n^2$ pro $n > 13$
- $8n^2 - 88n + 8 \neq \Omega(n^3)$ protože $8n^2 - 88n + 8 < cn^3$ pro $n > \frac{8}{c}$
- $8n^2 - 88n + 8 = \Omega(n)$ protože $8n^2 - 88n + 8 > n$ pro $n > 11$

Θ notace

Definice

$f(n) = \Theta(g(n))$ právě když existují kladné konstanty n_0, c_1 a c_1 takové, že pro všechna $n \geq n_0$ platí $c_1 g(n) \leq f(n) \leq c_2 g(n)$



funkce $f(n)$ a $g(n)$ rostou stejně rychle

Donald E. Knuth: *Big Omicron and big Omega and big Theta*.
ACM SIGACT, Volume 8 Issue 2, April-June 1976, pp. 18 - 24.

Θ notace - příklady

- $8n^2 - 88n + 8 = \Theta(n^2)$
  protože $8n^2 - 88n + 8 = \mathcal{O}(n^2)$ a současně $8n^2 - 88n + 8 = \Omega(n^2)$
- $8n^2 - 88n + 8 \neq \Theta(n^3)$ protože $8n^2 - 88n + 8 \neq \Omega(n^3)$
- $8n^2 - 88n + 8 \neq \Theta(n)$ protože $8n^2 - 88n + 8 \neq \mathcal{O}(n)$

Θ notace - příklad

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

- musíme najít **kladné** konstanty c_1, c_2 a n_0 takové, že

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

  platí pro všechna $n \geq n_0$
- po úpravě dostáváme

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- pravá nerovnost platí pro každé $n \geq 1$ jestliže zvolíme $c_2 \geq 1/2$
- levá nerovnost platí pro každé $n \geq 7$ jestliže zvolíme $c_1 \leq 1/14$
- volba $c_1 = 1/14$, $c_2 = 1/2$ a $n_0 = 7$ dokazuje platnost vztahu $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Asymptotická notace - vlastnosti

tranzitivita

$$f(n) = \Theta(g(n)) \text{ a } g(n) = \Theta(h(n)) \text{ implikuje } f(n) = \Theta(h(n))$$
$$f(n) = \mathcal{O}(g(n)) \text{ a } g(n) = \mathcal{O}(h(n)) \text{ implikuje } f(n) = \mathcal{O}(h(n))$$
$$f(n) = \Omega(g(n)) \text{ a } g(n) = \Omega(h(n)) \text{ implikuje } f(n) = \Omega(h(n))$$

reflexivita

$$f(n) = \Theta(f(n)) \qquad \text{podobně pro } \mathcal{O} \text{ a } \Omega$$

symetrie

$$f(n) = \Theta(g(n)) \text{ právě když } g(n) = \Theta(f(n))$$

transpozice

$$f(n) = \mathcal{O}(g(n)) \text{ právě když } g(n) = \Omega(f(n))$$

poznámka: ne každá dvojice funkcí je asymptoticky srovnatelná

Návrh a analýza algoritmů

Návrh algoritmů

ideální svět návod (*algoritmus*) „jak konstruovat algoritmy“

realita osvědčené postupy

- iterativní přístup
- rekurzivní přístup (*rozděl a panuj, divide et impera, divide and conquer*)
- dynamické programování
- hladové techniky
- heuristiky
- náhodnostní techniky
- aproximativní techniky
- parametrizované techniky
- . . . . . .

Rozděl a panuj - principy

Nothing is particularly hard if you divide it into small jobs *Henry Ford*

Rozděl **[divide]** problém na podproblémy, které mají menší velikost než původní problém.

Vyřeš **[conquer]** podproblémy stejným postupem(*rekurzívně*). Jestliže velikost podproblému je malá, použij přímé řešení.

Kombinuj **[combine]** řešení podproblémů a vyřeš původní problém.

Maximální a minimální prvek

- problém nalezení maximálního a minimálního prvku posloupnosti $S[1 \ldots n]$
- složitostní kritérium - počet porovnání prvků

MaxMin Iterative(S)

```
1 max ← S[1]
2 min ← S[1]
3 for i = 2 to n do
4     if S[i] > max then max ← S[i] fi
5     if S[i] < min then min ← S[i] fi
6 od
```

celkem $2(n-1)$ porovnání

Přístup Rozděl a panuj

1. posloupnost rozděl na dvě (stejně velké) podposloupnosti
2. najdi minimální a maximální prvek v obou podposloupnostech
3. kombinuj řešení podproblémů:
   maximálním prvek posloupnosti je větší z maximálních prvků podposloupností
   minimálním prvek posloupnosti je menší z minimálních prvků podposloupností

MaxMin(S, l, r)

```
1 if r = l then return (S[l], S[r]) fi
2 if r = l + 1 then return (max(S[l], S[r]), min(S[l], S[r])) fi
3 if r > l + 1 then (A, B) ← MaxMin(S, l, ⌊(l + r)/2⌋)
4                   (C, D) ← MaxMin(S, ⌊(l + r)/2⌋ + 1, r)
5                   return (max(A, C), min(B, D)) fi
```

iniciální volání $\textsc{MaxMin}(S, 1, n)$

Korektnost

konečnost výpočtu plyne z faktu, že každé rekurzivní volání se provede pro posloupnost menší délky

správnost vypočítaného výsledku dokážeme indukcí vzhledem k délce vstupní posloupnosti

$\mathbf{n = 1, n = 2}$ provedou se příkazy v řádku 1, resp. v řádku 2

indukční předpoklad algoritmus vypočítá korektní hodnoty pro všechny posloupnosti délky nejvýše $n - 1$ $(n > 1)$

platnost tvrzení pro $\mathbf{n}$ dle indukčního předpokladu jsou čísla A a B maximálním a minimálním prvkem posloupnosti $S[1, \ldots, \lfloor (1 + n)/2 \rfloor]$, stejně tak čísla C a D jsou maximálním a minimálním prvkem posloupnosti $S[\lfloor (1 + n)/2 \rfloor + 1, \ldots, n]$
větší z čísel A, C je pak maximálním prvkem posloupnost $A[1, \ldots, n]$ a menší z čísel B, D jejím minimem

Složitost

rozděl problém na podproblémy menší velikosti

vyřeš podproblémy

kombinuj řešení podproblémů a vyřeš původní problém

n je délka vstupu, podproblémy mají velikosti $n_1, n_2, \ldots, n_k$
$T(\cdot)$ je časová složitost výpočtu na vstupu délky n

$$T(n) = \text{složitost rozdělení} + T(n_1) + T(n_2) + \ldots T(n_k) + \text{složitost kombinace}$$

konkrétně pro algoritmus MaxMin, složitost je počet porovnání prvků posloupnosti

$$T(n) = \begin{cases} 0 + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 & \text{pro } n > 2 \\ 1 & \text{pro } n = 2 \\ 0 & \text{pro } n = 1 \end{cases}$$

Složitost

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 & \text{pro } n > 2 \\ 1 & \text{pro } n = 2 \\ 0 & \text{pro } n = 1 \end{cases}$$

indukcí vzhledem k n ověříme, že pro $n > 1$ platí

$$T(n) \leq \frac{5}{3}n - 2$$

indukční základ $n = 2$ $T(2) = 1 < \frac{5}{3} \cdot 2 - 2$
indukční předpoklad nerovnost platí pro všechny hodnoty i, $2 \leq i < n$
platnost pro n

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 \qquad \textit{využijeme indukční předpoklad}$$
$$\leq \frac{5}{3}\lfloor n/2 \rfloor - 2 + \frac{5}{3}\lceil n/2 \rceil - 2 + 2 = \frac{5}{3}n - 2$$

Kdo je nejrychlejší???

Min1(S, l, r)

$minimum \leftarrow S[l]$
for $i = l + 1$ **to** r **do**
    if $minimum > S[i]$ **then** $minimum \leftarrow S[i]$ **fi od**
return $minimum$

Min2(S, l, r)

if $r = l$ **then return** $S[r]$ **fi**
if $r > l$ **then** $A \leftarrow \textsc{Min2}(S, l, \lfloor (l + r)/2 \rfloor)$
$B \leftarrow \textsc{Min2}(S, \lfloor (l + r)/2 \rfloor + 1, r)$
return $\min(A, B)$ **fi**

Min3(S, l, r)

if $r = l$ **then return** $S[r]$ **fi**
if $r > l$ **then** $A \leftarrow \textsc{Min3}(S, l, r - 1)$
return $\min(A, S[r])$ **fi**

Složitost rekurzivních algoritmů

- složitost obvykle zapíšeme pomocí **rekurentní rovnice**, která vyjadřuje složitost výpočtu na vstupu velikosti n pomocí složitosti výpočtů na menších vstupech

- označme $T(n)$ časovou složitost výpočtu na vstupu délky n
- pro dostatečně malý vstup ($n \leq c$) si přímé řešení vyžaduje konstantní čas
- velký vstup rozdělíme na a podproblémů z nichž každý má velikost $1/b$ velikosti původního vstupu
- řešení každého podproblému si vyžádá čas $T(n/b)$
- označme $D(n)$ čas potřebný na konstrukci podproblémů a $C(n)$ čas potřebný na kombinaci řešení podproblémů a nalezení řešení původního problému

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{jinak} \end{cases}$$

jak najít řešení[1] *rekurentní rovnice???*

[1] nerekurzivní popis funkce, která splňuje podmínky rovnice

Řešení rekurentních rovnic

substituční metoda „uhodneme“ řešení a dokážeme jeho správnost matematickou indukcí

metoda rekurzivního stromu konstruujeme strom, jehož vrcholy vyjadřují složitost jednotlivých rekurzivních volání; výslednou složitost vypočítáme jako sumu ohodnocení vrcholů stromu

kuchařková věta *(master method)* vzorec pro řešení rekurentní rovnice tvaru $T(n) = aT(n/b) + f(n)$

Substituční metoda

1. „uhodni“ řešení
2. matematickou indukcí dokaž jeho korektnost

Příklad

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{jinak} \end{cases}$$

1. $T(n) = \mathcal{O}(n \log n)$
2. indukcí dokážeme, že $T(n) \leq cn \log n$ pro dostatečně velké n a vhodně zvolenou konstantu c

dokazujeme $T(n) = \mathcal{O}(n \log n)$ pro rovnici

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{jinak} \end{cases}$$

Indukční krok

- předpokládejme, že tvrzení platí pro všechna $m < n$, tj. speciálně pro $m = \lfloor n/2 \rfloor$ platí $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$
- využitím indukčního předpokladu dokážeme platnost tvrzení pro n

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

dokazujeme $T(n) = \mathcal{O}(n \log n)$ pro rovnici

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{jinak} \end{cases}$$

- jako indukční základ nemůžeme použít případ $n = 1$, protože neplatí $T(1) \leq cn \log n = c1 \log 1 = 0$
- využijeme, že dokazujeme asymptotický vztah, a proto stačí, aby nerovnost platila pro všechna dostatečně velká n
- tvrzení dokážeme pro $n \geq 2$
- základem indukce bude platnost vztahu pro $n = 2$ a $n = 3$
- pro $n > 3$ závisí hodnota $T(n)$ jenom od $T(2)$ a $T(3)$

Indukční základ $n = 2$ a $n = 3$

- dosazením do rovnice zjistíme, že $T(2) = 4$ a $T(3) = 5$
- zvolíme konstantu $c \geq 1$ tak, aby pro $n = 2$ a $n = 3$ platilo $T(n) \leq cn \log n$
- dobrá volba je $c \geq 2$, protože platí $T(2) \leq c \cdot 2 \log 2$ i $T(3) \leq c \cdot 3 \log 3$

Metoda rekurzivního stromu

- „rozbalování rekurze“
- přehledný zápis pomocí stromu, jehož vrcholy vyjadřují složitost jednotlivých rekurzivních volání
- vrchol stromu je ohodnocen složitostí dekompozice a kompozice
- synové vrcholu odpovídají jednotlivým rekurzivním voláním
- výslednou složitost vypočítáme jako sumu ohodnocení vrcholů, obvykle sečítáme po jednotlivých úrovních stromu

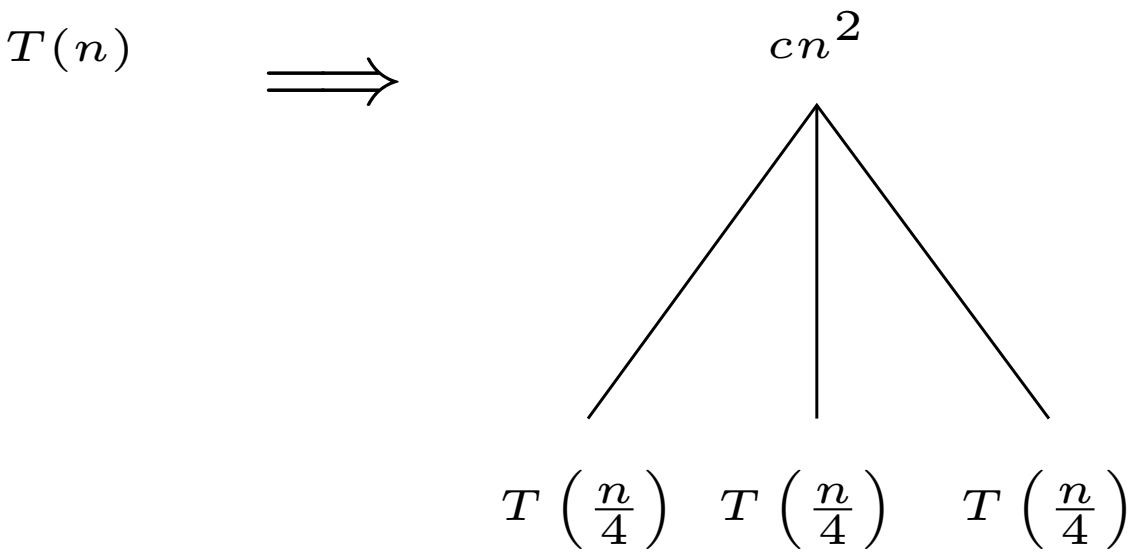
metodu můžeme použít pro

- nalezení přesného řešení (je nutné přesné počítání)
- pro získání odhadu na řešení rekurentní rovnice; pro důkaz řešení se pak použije substituční metoda

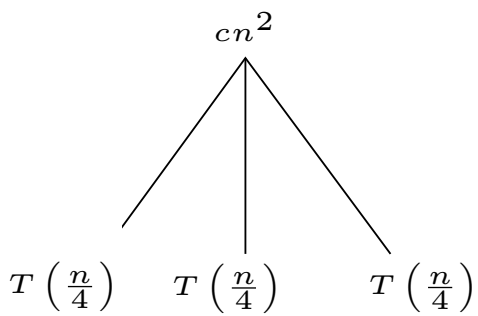
Metoda rekurzivního stromu - příklad

$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 3T(\lfloor n/4 \rfloor) + cn^2 & \text{jinak} \end{cases}$$

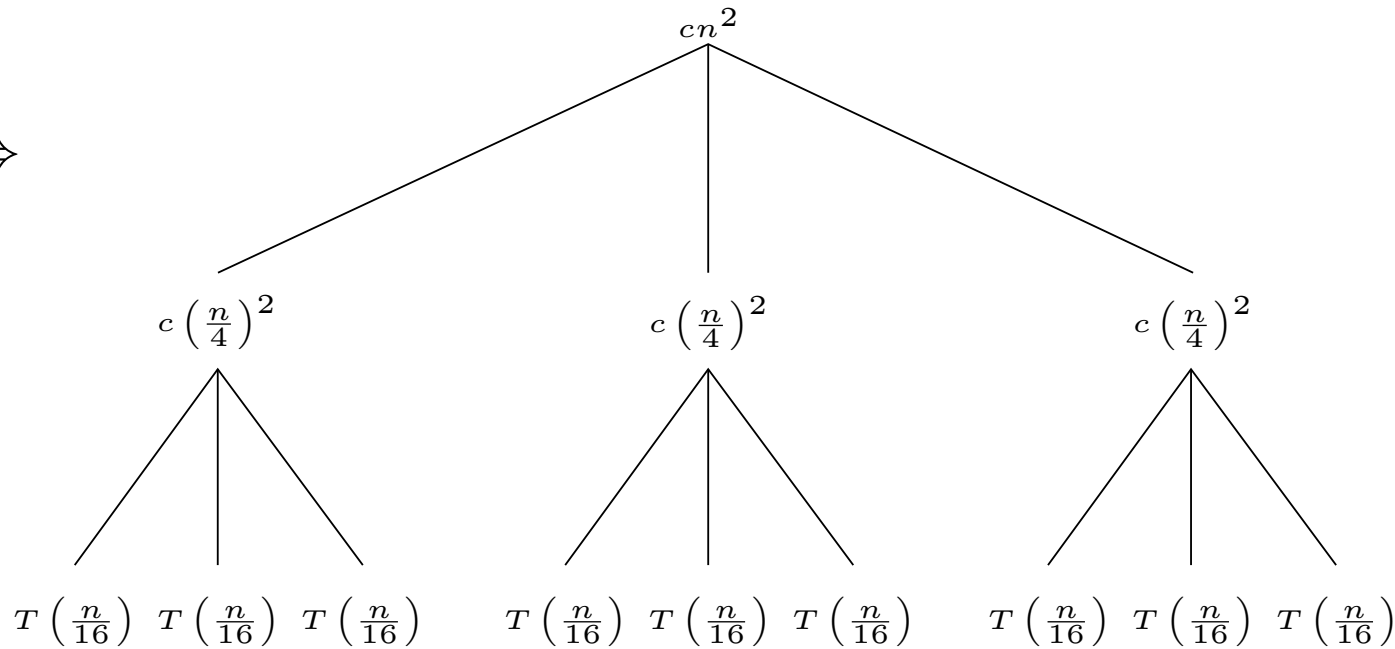
metodu použijeme pro získání odhadu řešení, můžeme proto předpokládat, že n je mocninou 4

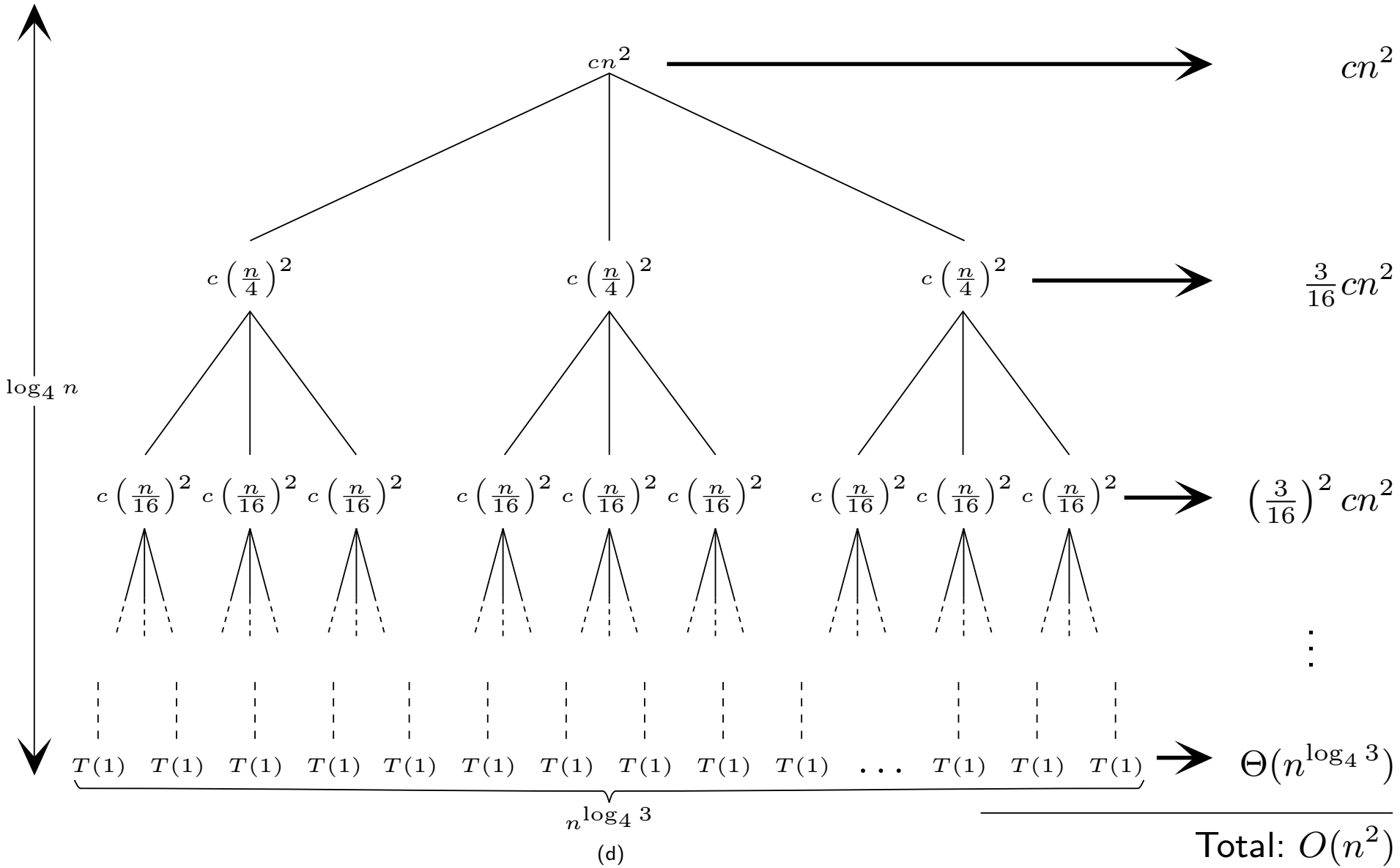


$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 3T(\lfloor n/4 \rfloor) + cn^2 & \text{jinak} \end{cases}$$



$\Longrightarrow$




cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ ... $T(1)$ $T(1)$ $T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
Total: $O(n^2)$

(d)

- **kořen** má hloubku 0

- **(vnitřní) vrchol** v hloubce i je označen složitostí $c(n/4^i)^2$
- počet vrcholů s hloubkou i je 3^i
- součet složitostí vrcholů v hloubce i je $3^i c(n/4^i)^2 = (3/16)^i cn^2$

- **list** je označen složitostí 1 (základ rekurentní rovnice) a má hloubku $i = \log_4 n$ (protože $n/4^{\log_4 n} = 1$)
- počet listů je $3^{\log_4 n} = n^{\log_4 3}$

- sumací přes všechny úrovně dostáváme

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + n^{\log_4 3}$$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13}cn^2 + n^{\log_4 3}$$

$$= \mathcal{O}(n^2)$$

hodnotu $\mathcal{O}(n^2)$ použijeme jako odhad pro substituční metodu

Kuchařková věta (Master method)

Nechť $a \geq 1$ a $b > 1$ jsou konstanty, $f(n)$ je polynomiální funkce, a nechť $T(n)$ je definována na nezáporných číslech rekurentní rovnicí

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

Potom platí

$$T(n) = \begin{cases} \Theta(f(n)) & \text{když } af(n/b) = \kappa f(n) \text{ pro nějakou konstantu } \kappa < 1 \\ \Theta(n^{\log_b a}) & \text{když } af(n/b) = Kf(n) \text{ pro nějakou konstantu } K > 1 \\ \Theta(f(n) \log_b n) & \text{když } af(n/b) = f(n) \end{cases}$$

Kuchařková věta - alternativní varianta

Nechť $a \geq 1$, $b > 1$ a $c \geq 0$ jsou konstanty a nechť $T(n)$ je definována na nezáporných číslech rekurentní rovnicí

$$T(n) = aT(n/b) + \Theta(n^c)$$

Potom platí

$$T(n) = \begin{cases} \Theta(n^c) & \text{když } a < b^c \quad \textit{případ 1} \\ \Theta(n^c \log n) & \text{když } a = b^c \quad \textit{případ 2} \\ \Theta(n^{\log_b a}) & \text{když } a > b^c \quad \textit{případ 3} \end{cases}$$

věta platí i ve variantě pro $\mathcal{O}$ a Ω

Příklady použití kuchařkové věty I

- $T(n) = 4T(n/2) + 1 \implies T(n) = \Theta(n^2)$
  případ 3, $a = 4, b = 2, c = 0,\ 4 > 2^0$
- $T(n) = 4T(n/2) + n \implies T(n) = \Theta(n^2)$
  případ 3, $a = 4, b = 2, c = 1,\ 4 > 2^1$
- $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$
  případ 2, $a = 4, b = 2, c = 2,\ 4 = 2^2$
- $T(n) = 4T(n/2) + n^3 \implies T(n) = \Theta(n^3)$
  případ 1, $a = 4, b = 2, c = 3,\ 4 < 2^3$

Příklady použití kuchařkové věty II

- $T(n) = 2T(n/2) + 1 \implies T(n) = \Theta(n)$
  případ 3, $a = 2, b = 2, c = 0,\ 2 > 2^0$
- $T(n) = 2T(n/2) + n \implies T(n) = \Theta(n \log n)$
  případ 2, $a = 2, b = 2, c = 1,\ 2 = 2^1$
- $T(n) = 2T(n/2) + n^2 \implies T(n) = \Theta(n^2)$
  případ 1, $a = 2, b = 2, c = 2,\ 2 < 2^2$
- $T(n) = 2T(n/2) + n^3 \implies T(n) = \Theta(n^3)$
  případ 1, $a = 2, b = 2, c = 3,\ 2 < 2^3$

Příklady

Hanojské věže

$T(n) = 2T(n-1) + 1$, základní případ $T(0) = 0$

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

MergeSort

$T(n) \leq 2T(n/2) + \Theta(n)$

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

násobení celých čísel

$T(n) = 4T(n/2) + \mathcal{O}(n)$

$T(n) = \mathcal{O}(n^2)$

Strassenův algoritmus pro násobení celých čísel

$T(n) = 7T(n/2) + \mathcal{O}(n^2)$

$T(n) = \mathcal{O}(n^{\log_b a}) = \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.81})$

Jak nepoužívat rekurzi - nedefinovaný výpočet

Bad_Factorial(n)

return $n \cdot \textsc{Bad_Factorial}(n-1)$

chybí základ rekurze

Jak nepoužívat rekurzi - nekonečný výpočet

Awful_Factorial(n)

if $n = 0$ **then return** 1
else return $\frac{1}{n+1}\textsc{Awful\_Factorial}(n+1)$ **fi**

Beta(n)

if $n = 1$ **then return** 1
else return $n(n-1)\textsc{Beta}(n-2)$ **fi**

Jak nepoužívat rekurzi - složitost

Fibonacciho posloupnost

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

RecFibo(n)

if $n < 2$ **then return** n
else return $\textsc{RecFibo}(n-1) + \textsc{RecFibo}(n-2)$ **fi**

časová složitost algoritmu

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1$$

$$T(n) = \Theta(\phi^n), \quad \phi = (\sqrt{5} + 1)/2$$

Jak nepoužívat rekurzi - složitost

Fibonacciho posloupnost

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

IterFibo(n)

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i = 2$ **to** n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$ **od**
return $F[n]$

časová složitost algoritmu

$$T(n) = \Theta(n)$$

Jak používat rekurzi - význam definice podproblémů

problém maximální podposloupnosti

- dané je pole celých čísel $A[1..n]$
- cílem je najít takové indexy $1 \leq i \leq j \leq n$, pro které je suma $A[i] + \cdots + A[j]$ maximální

- 13, -3, -25, 20 -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7
- řešením je 18, 20, -7, 12

- řešení *hrubou silou* - prozkoumat všechny dvojice indexů i, j
- kvadratická složitost
- existuje lepší řešení
- *rozděl a panuj*?

Rozděl a panuj

daná je posloupnost $A[low \dots high]$ a hodnota mid

hledané řešení $A[i \dots j]$

(A) je podposloupností $A[low \dots mid]$ $(low \leq i \leq j \leq mid)$

(B) je podposloupností $A[mid + 1 \dots high]$ $(mid + 1 \leq i \leq j \leq high)$

(C) zasahuje do obou podposloupností $(low \leq i \leq mid < j \leq high)$

- **(A)** a **(B)** jsou problémy stejného typu jako původní problém
- **(C)** ????

Rozděl a panuj

daná je posloupnost $A[low \dots high]$ a hodnota mid

hledané řešení $A[i \dots j]$

(A) je podposloupností $A[low \dots mid]$ ($low \leq i \leq j \leq mid$)

(B) je podposloupností $A[mid + 1 \dots high]$ ($mid + 1 \leq i \leq j \leq high$)

(C) zasahuje do obou podposloupností ($low \leq i \leq mid < j \leq high$)

- **(A)** a **(B)** jsou problémy stejného typu jako původní problém
- **(C)** ????
- pro řešení **(C)** stačí poznat podposloupnosti tvaru $A[i \dots mid]$ a $A[mid + 1 \dots j]$ s maximální sumou

Případ C

FIND_MAX_CROSSING_SUBARRAY

F_M_C_S$(A, low, mid, high)$

```
1  leftsum ← −∞
2  sum ← 0
3  for i = mid downto low do
4       sum ← sum + A[i]
5       if sum > leftsum then leftsum ← sum
6                             maxleft ← i fi od
7  rightsum ← −∞
8  sum ← 0
9  for j = mid + 1 to high do
10      sum ← sum + A[j]
11      if sum > rightsum then rightsum ← sum
12                             maxright ← j fi od
13 return (maxleft, maxright, leftsum + rightsum)
```

Algoritmus

Find_Maximum_Subarray

F_M_S$(A, low, high)$

```
1  if high = low
2     then return (low, high, A[low])
3      else mid = ⌈(low + high)/2⌉
4           (leftlow, leftthigh, leftsum) ← F_M_S(A, low, mid)
5           (rightlow, righthigh, righttsum) ← F_M_S(A, mid + 1, high)
6           (crosslow, crosshigh, crosssum) ← F_M_C_S(A, low, mid, high) fi
7  if leftsum ≥ rightsum ∧ leftsum ≥ crosssum
8     then return (leftlow, leftthigh, leftsum) fi
9  if leftsum ≤ rightsum ∧ rightsum ≥ crosssum
10    then return (rightlow, righthigh, rightsum)
11     else return (crosslow, crosshigh, crosssum) fi
```

Složitost

procedura FIND_MAX_CROSSING_SUBARRAY$(A, low, mid, high)$

- označme $n = high - low + 1$
- jedna iterace obou cyklů má konstantní složitost
- počet iterací cyklu pro levou část posloupnosti je $mid - low + 1$
- počet iterací cyklu pro pravou část posloupnosti je $high - mid$
- celková složitost je $\Theta(n)$

algoritmus FIND_MAXIMUM_SUBARRAY

dekompozice a kompozice v konstantním čase, řešení problému (C) v čase $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(n/2) + \Theta(n) & \text{jinak} \end{cases}$$

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

Rekurzivní vs iterativní přístup

pro

- intuitivní, jednoduchý návrh
- důkaz korektnosti využitím matematické indukce
- analýza složitosti využitím rekurentní rovnice
- efektivní řešení

proti

- neefektivní implementace
- ne vždy podpora ze strany programovacího jazyka
- neefektivní řešení

- každý rekurzivní algoritmus lze převést na iterativní
- simulace zásobníku volání
- *(resp. dynamické programování)*
- jednoduchý přepis v případě *tail rekurze*

Rekurzivní vs iterativní přístup

tail rekurze -speciální případ rekurze, kde se po rekurzivním volání nedělá žádný výpočet

ANO

$\mathrm{F}(x, y)$
if $y = 0$ **then return** x
else $F(x \cdot y + x, y - 1)$ **fi**

NE

$\mathrm{G}(x)$
if $y = 0$ **then return** x
else $y \leftarrow G(x - 1)$ **fi**
return $x \cdot y$

Rekurzivní vs iterativní přístup

$\mathrm{F}(x, y)$
if $y = 0$ **then return** x
else $F(x \cdot y + x, y - 1)$ **fi**

$\mathrm{F}(x, y)$
$label$: **if** $y = 0$ **then return** x
else $x \leftarrow x \cdot y + x$
$y \leftarrow y - 1$
goto $label$ **fi**

$\mathrm{F}(x, y)$
$ret \leftarrow x$
for $y = 1$ **to** y **do**
$ret \leftarrow ret \cdot y + ret$ **od**
return ret

Rekurzivní vs iterativní přístup

```
BIN SEARCH(x, A, left, right)
if right = left then return A[left] == x fi
if right < left then mid = ⌊(left + right)/2⌋
                     if A[mid] = x then return true fi
                     if A[mid] < x then left ← mid + 1
                                   else right ← mid fi
                     BIN SEARCH(x, A, left, right)
fi
```

```
BIN SEARCH(x, A, left, right)
while right < left do mid = ⌊(left + right)/2⌋
      if A[mid] = x then return true fi
      if A[mid] < x then left ← mid + 1
                    else right ← mid fi
od
```

Domácí úkol

Vstup: pole $A[1 \ldots n]$ *celých čísel*

CoToDela(n)

if $n = 1$ **then write** A
else for $i = 1$ **to** n **do**
CoToDela$(n-1)$
if n je liché **then** swap $A[1]$ a $A[n]$
else swap $A[i]$ a $A[n]$ **fi**
od fi

- co je výstupem algoritmu?
- jaká je složitost výpočtu?

Řazení

Problém řazení

- je daná množina K, nad kterou je definované úplné uspořádání
- vstupem problému řazení je posloupnost $A = (k_1, \ldots, k_n)$ prvků z K
- výstupem je posloupnost $A' = (k'_1, \ldots, k'_n)$, která je takovou permutací posloupnosti A, že $\forall i, j, 1 \leq i < j \leq n$, platí $k'_i \leq k'_j$

Stabilní algoritmy a algoritmy *in situ*

- prvky množiny K mohou být strukturované
- řazení podle **klíče**
- řazení se nazývá **stabilní** právě když zachovává vzájemné pořadí položek se stejným klíčem

- prostorová složitost algoritmů řazení je $\Omega(n)$, protože samotná vstupní posloupnost má délku n
- pro přesnější charakterizaci prostorové složitosti jednotlivých algoritmů uvažujeme tzv. **extrasekvenční prostorovou složitost**, do které nezapočítáváme paměť obsazenou vstupní posloupností
- algoritmy, jejichž extrasekvenční složitost je konstantní, se nazývají **in situ** (*in place*)

Přehled

algoritmus	časová složitost v nejhorším případě	časová složitost v průměrném případě
řazení vkládáním	$\Theta(n^2)$	$\Theta(n^2)$
řazení výběrem	$\Theta(n^2)$	$\Theta(n^2)$
řazení sléváním	$\Theta(n \log n)$	$\Theta(n \log n)$
řazení haldou	$\Theta(n \log n)$	$\Theta(n \log n)$
řazení rozdělováním	$\Theta(n^2)$	$\Theta(n \log n)$
řazení počítáním	$\Theta(k + n)$	$\Theta(k + n)$
číslicové řazení	$\Theta(d(n + k))$	$\Theta(d(n + k))$
přihrádkové řazení	$\Theta(n^2)$	$\Theta(n)$

Přehled

algoritmy založené na porovnávání prvků

vkládáním, Insertion sort in situ, stabilní

výběrem, Selection sort in situ, není stabilní

sléváním, Merge sort asymptoticky časově optimální, není in situ, stabilní

haldou, Heapsort asymptoticky časově optimální, in situ, není stabilní

rozdělováním, Quicksort není časově optimální, extrasekvenční složitost a stabilita závisí od implementace (optimálně in situ, existují stabilní implementace), velmi dobrý v praxi (průměrná složitost je $\Theta(n \log n)$)

algoritmy, které získávají informace jinak než porovnáváním prvků

počítáním, Counting sort vstupní prvky jsou z množiny $\{0, \ldots, k\}$

číslicové řazení, Radix sort zobecnění řazení počítáním

přihrádkové řazení, Bucket sort vyžaduje znalost o pravděpodobnostním rozdělení čísel na vstupu

Řazení

Řazení sléváním (Merge sort)

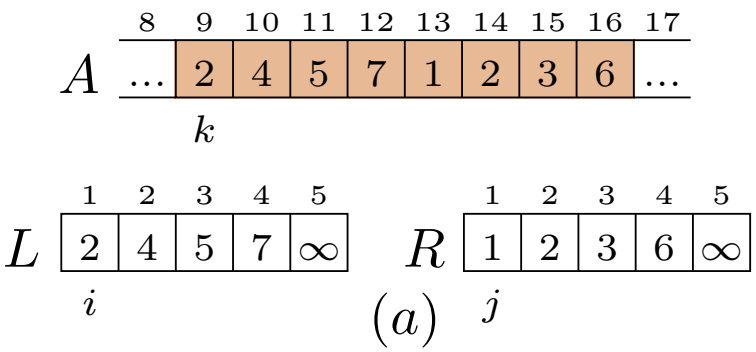
Rozděl posloupnost na dvě stejně velké podposloupnosti

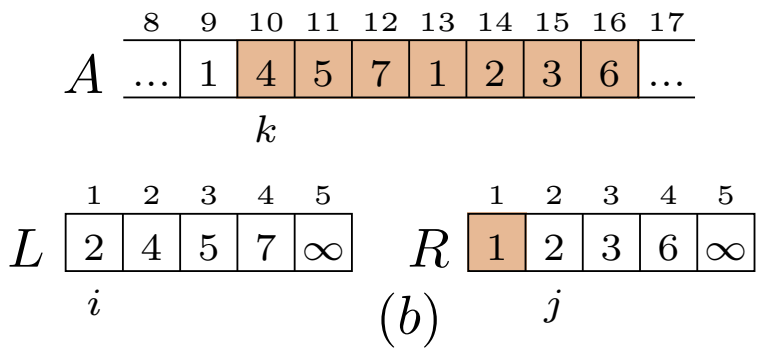
Vyřeš obě podposloupnosti (rekurzivně)

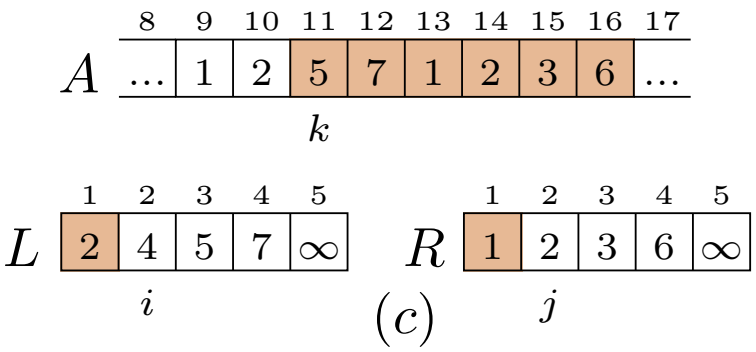
Kombinuj dvě seřazené podposloupnosti do jedné

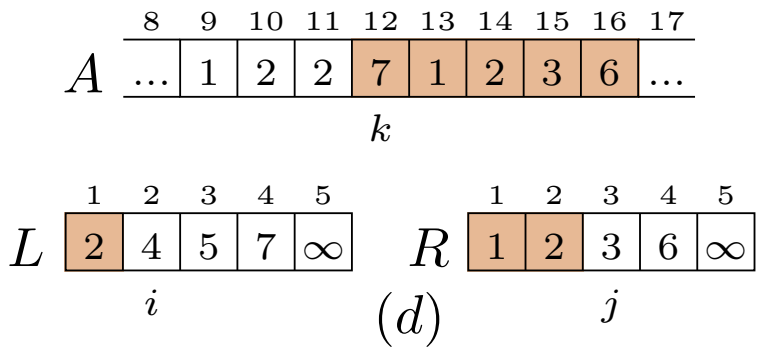
Spojení dvou seřazených posloupností - Merge

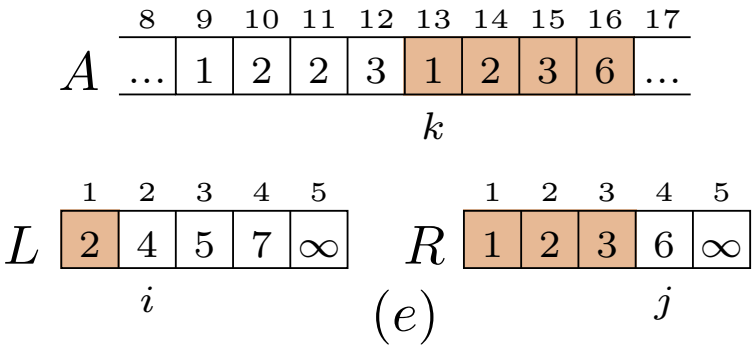
- otázkou je, jak spojit dvě seřazené posloupnosti do jedné, která bude seřazená
- při slévání porovnáváme vedoucí prvky obou posloupností
- menší z porovnávaných prvků přesuneme do výslední posloupnosti
- procedura MERGE má 4 parametry
  pole A
  indexy p, q, r takové, že $p \leq q \leq r$
  předpokládáme, že posloupnosti $A[p \ldots q]$ a $A[q+1 \ldots r]$ jsou seřazené
- pro provedení výpočtu je posloupnost $A[p \ldots r]$ seřazená
- pro zjednodušení kódu používáme *sentinel*


A
8 9 10 11 12 13 14 15 16 17
... 2 4 5 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(a)

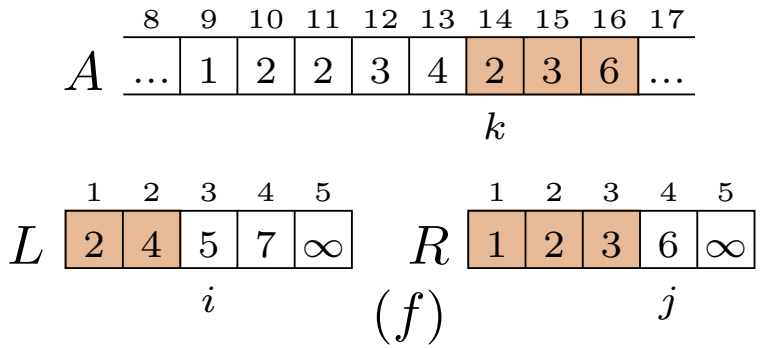

A
8 9 10 11 12 13 14 15 16 17
... 1 4 5 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(b)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 5 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(c)

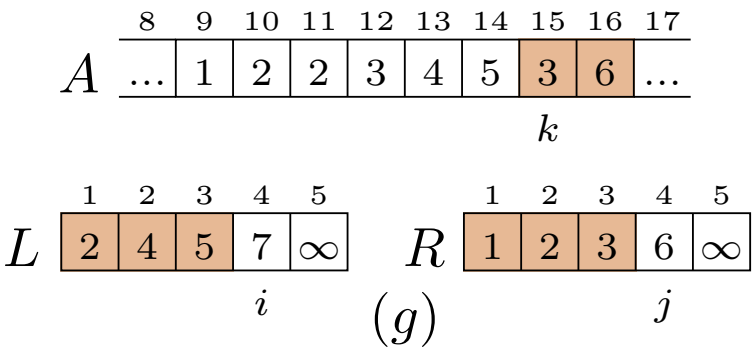

A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(d)


8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 1 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

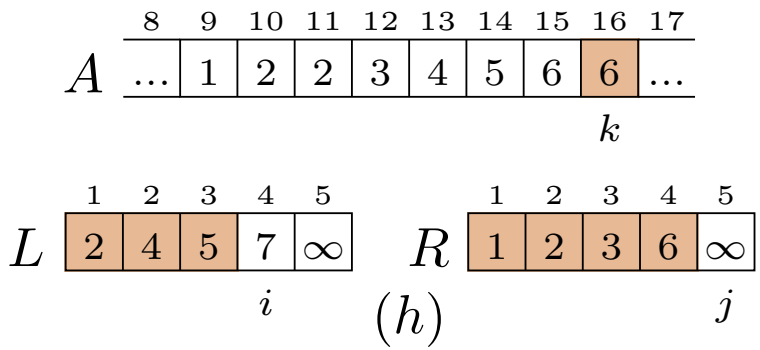
(e)


8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

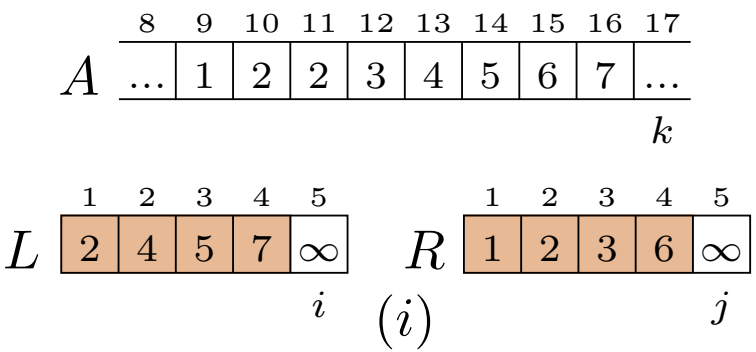
(f)


8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 5 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 5 6 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 5 6 7 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Merge

Procedure Merge(A, p, q, r)

```
1  n1 ← q − p + 1
2  n2 ← r − q
3  //nechť L[1 . . . n1 + 1] a R[1 . . . n2 + 1] jsou nová pole
4  for i = 1 to n1 do L[i] ← A[p + i − 1] od
5  for j = 1 to n2 do R[j] ← A[q + j] od
6  L[n1 + 1] ← ∞
7  R[n2 + 1] ← ∞
8  i ← 1
9  j ← 1
10 for k = p to r do
11     if L[i] ≤ R[j] then A[k] ← L[i]
12                         i ← i + 1
13                    else A[k] ← R[j]
14                         j ← j + 1 fi
15 od
```

Korektnost procedury Merge

Invariant

Na začátku každé iterace cyklu **for** v řádcích 10 - 15 posloupnost $A[p \dots k-1]$ obsahuje $k-p$ nejmenších prvků z $L[1 \dots n_1+1]$ a $R[1 \dots n_2+1]$ a to v pořadí podle velikosti. Navíc, $L[i]$ a $R[j]$ jsou nejmenší prvky mezi těmi prvky ve svých posloupnostech, které ještě nebyly zkopírované do A.

Inicializace Na začátku je $k = p$. Navíc $i = j = 1$ a tedy $L[i]$ a $R[j]$ jsou nejmenší prvky v L a R.

Iterace Předpokládejme, že $L[i] \leq R[j]$. Potom $L[i]$ je nejmenší prvek z těch, které ještě nebyly zkopírované do A. Protože $A[p \dots k-1]$ obsahuje $k-p$ nejmenších prvků, pole $A[p \dots k]$ bude obsahovat $k-p+1$ nejmenších prvků. Zvýšením k a i zaručíme platnost invariantu i po ukončení iterace.

Ukončení Cyklus končí když $k = r+1$. Z platnosti invariantu posloupnost $A[p \dots k-1] = A[p \dots r]$ obsahuje seřazených $k-p = r-p+1$ nejmenších prvků z $L[1 \dots n_1+1]$ a $R[1 \dots n_2+1]$. Pole L a R obsahují v součtu $n_1+n_2+2 = r-p+3$ prvků. Všechny prvky, s výjimkou dvou největších, byly zkopírované do A. Dva největší prvky jsou sentinely.

Složitost procedury Merge

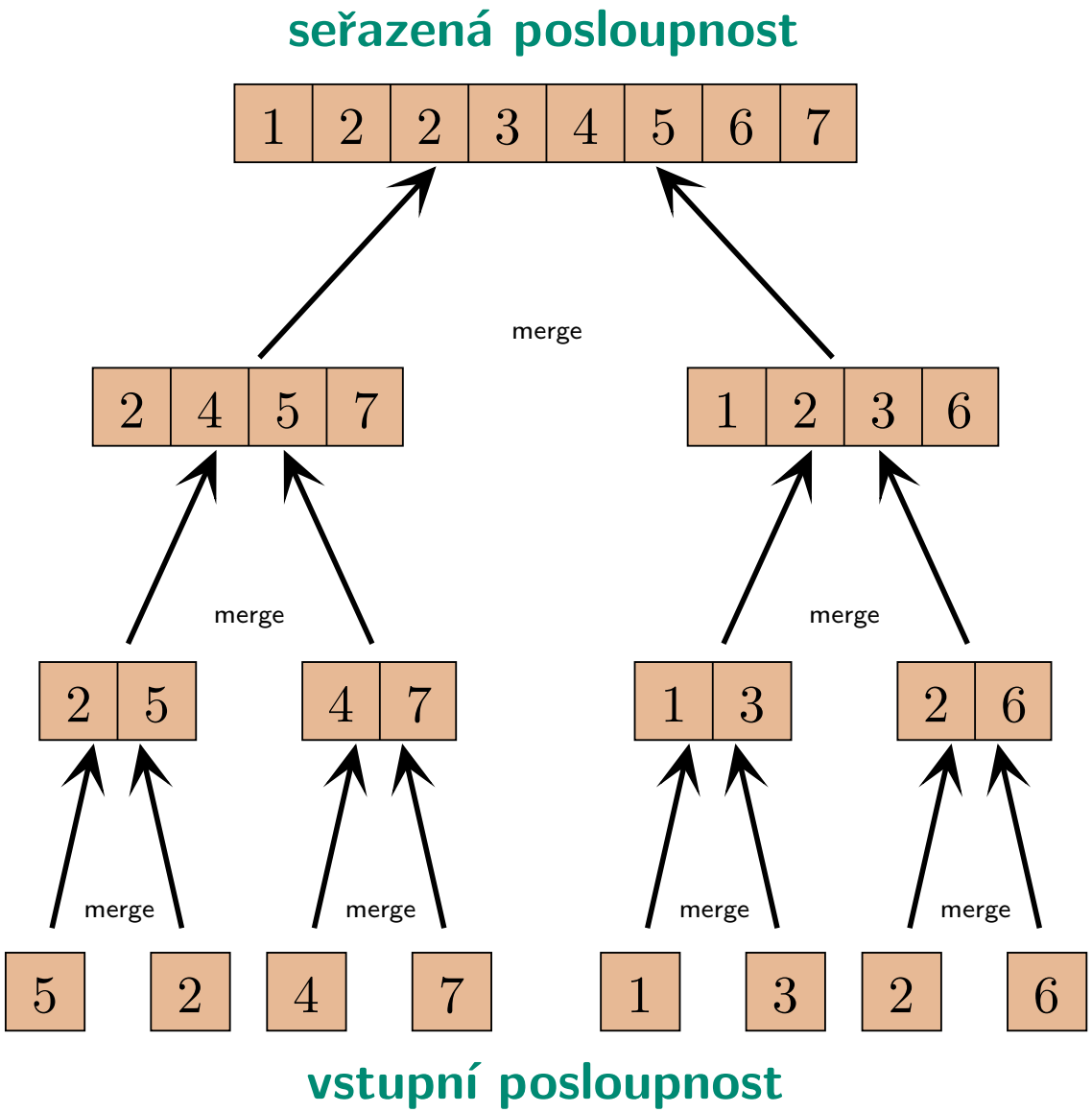
- řádky 1 - 2 a 6 - 9 mají konstantní složitost
- **for** cykly v řádcích 4 a 5 mají v součtu složitost $\Theta(n_1 + n_2) = \Theta(n)$, kde $n = r - p + 1$
- **for** cyklus v řádcích 10 - 15 iteruje n krát, všechny příkazy v řádcích 11 - 14 mají konstantní složitost
- složitost procedury MERGE je $\Theta(n)$

Merge Sort

- využívá proceduru MERGE
- pro seřazení celé posloupnosti voláme MERGE SORT$(A, 1, A.length)$

Procedure Merge Sort(A, p, r)

```
1 if p < r then q ← ⌊(p + r)/2⌋
2              MERGE SORT(A, p, q)
3              MERGE SORT(A, q + 1, r)
4              MERGE(A, p, q, r) fi
```


seřazená posloupnost
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
vstupní posloupnost

Složitost algoritmu Merge Sort

Rozděl rozdělení znamená výpočet indexu, proto má složitost $\Theta(1)$

Vyřeš rekurzívně zpracujeme dvě posloupnosti velikosti $n/2$, časová složitost je $2T(n/2)$

Kombinuj složitost procedury MERGE je $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{ak } n = 1 \\ 2T(n/2) + \Theta(n) & \text{jinak} \end{cases}$$

složitost MERGE SORT je $T(n) = \Theta(n \log n)$

Problém inverzí

motivace

porovnání seznamu preferencí

formulace problému

- je daná posloupnost vzájemně různých čísel $a_1, \ldots, a_n$
- inverzí v posloupnosti je dvojice indexů i, j takových, že $i < j$ a současně $a_i > a_j$
- úkolem je najít všechny inverze v dané posloupnosti čísel

příklad

posloupnost 1, 4, 6, 8, 2, 5 má 5 inverzí

naivní algoritmus

otestuje všechny dvojice indexů, složitost $\mathcal{O}(n^2)$

Problém inverzí - přístup Rozděl a panuj

1. **rozděl** posloupnost rozdělíme na dvě (stejně velké) podposloupnosti
2. **vyřeš** v každé z podposloupností spočítáme inverze
3. **kombinuj** k počtu inverzí z podposloupností připočítáme inverze mezi prvky různých podposloupností

- cílem je navrhnout algoritmus s lepší složitostí než je složitost naivního algoritmu ($\mathcal{O}(n^2)$)
- jestliže chceme, aby časová složitost rekurzivního algoritmu byla $T(n) = \mathcal{O}(n \log n)$, tak musí platit $T(n) \leq 2T(n/2) + \mathcal{O}(n)$, tj. složitost výpočtu v bodech 1 a 3 nesmí být větší než lineární
- jak spočítat inverze mezi prvky různých posloupností (bod 3) v čase $\mathcal{O}(n)$?

Problém inverzí - kombinuj - pokus 1

otázka jak spočítat inverze mezi prvky z posloupnosti A a posloupnosti B ?

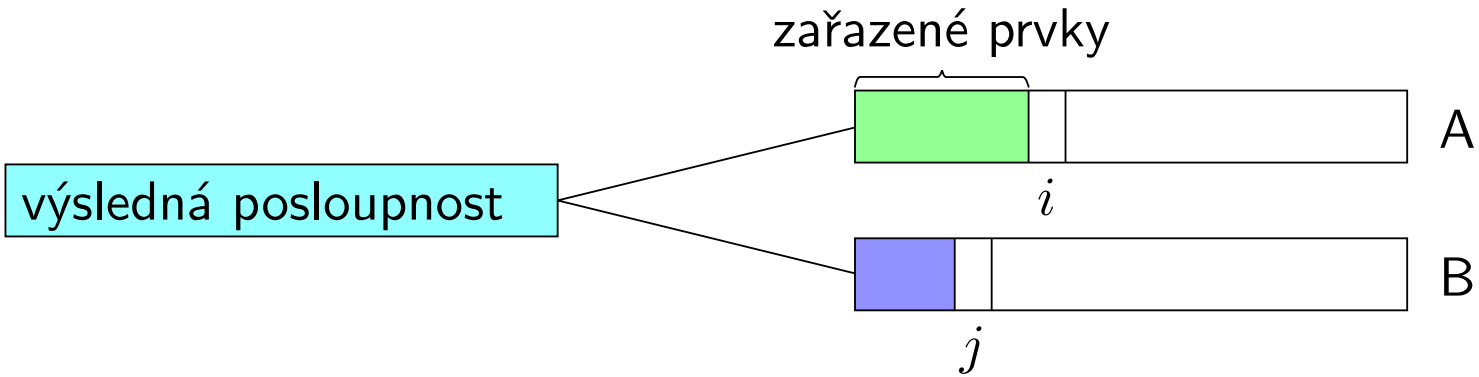
odpověď jednoduchá za předpokladu, že A i B jsou seřazené

algoritmus

- seřaď A a B
- pro každý prvek $b \in B$
  binárním vyhledáváním v A urči, kolik prvků v A je větších než b

Problém inverzí - kombinuj - pokus 2

- $A = (a_1, a_2, \ldots, a_k)$, $B = (b_1, b_2, \ldots, b_l)$
- předpokládáme, že
  - prvky v obou posloupnostech jsou seřazeny vzestupně
  - všechny prvky posloupnosti A mají ve vstupní posloupnosti menší index než prvky posloupnosti B
- postupujeme stejně jako v proceduře MERGE
- prvky $a_1, \ldots, a_{i-1}$ a $b_1, \ldots, b_{j-1}$ jsou již zařazené
- porovnáváme prvek a_i s prvkem b_j
  - menší z porovnávaných prvků zařadíme do výstupní posloupnosti
  - jestliže $\mathbf{a_i < b_j}$, tak a_i není v inverzi se žádným z prvků $b_j, b_{j+1}, \ldots, b_l$
  - jestliže $\mathbf{a_i > b_j}$, tak b_j je v inverzi se všemi prvky $a_i, \ldots, a_k$, a proto k počtu inverzí připočteme $k - i + 1$



- inverze mezi prvky zařazenými do výsledné posloupnosti jsou již započítané
- jestliže $\mathbf{a_i < b_j}$, tak do výsledné posloupnosti přesuneme a_i

  $a_i < b_j < b_{j+1} < b_{j+2} < \ldots$
  a_i není v inverzi se žádným z $b_j, b_{j+1}, b_{j+2} \ldots$
- jestliže $\mathbf{a_i > b_j}$, tak do výsledné posloupnosti přesuneme b_j

  $b_j < a_i < a_{i+1} < a_{j+2} < \ldots$
  b_j je v inverzi s každým z $a_i, a_{i+1}, a_{i+2} \ldots$

Algoritmus

Merge_and_Count(A, B)

```
1  i ← 1; j ← 1
2  //i, j jsou indexy prvních nezařazených prvků z A resp. B
3  Count ← 0
4  //Count je počet nalezených inverzí
5  while seznamy A, B jsou neprázdné do
6        porovnej a_i a b_j
7        menší z prvků zařaď do výsledného seznamu
8        if b_j < a_i then zvyš Count o počet nezařazených prvků z A fi
9        zvyš index i resp. j od
10 if jeden seznam je prázdný
11   then zařaď zbývající prvky do výsledného seznamu fi
12 return Count a výsledný seznam
```

Algoritmus

Sort_and_Count(L)

```
1 if length(L) = 1
2   then r ← 0
3    else A ← levá polovina L
4         B ← pravá polovina L
5         (r_A, A) ← Sort_and_Count(A)
6         (r_B, B) ← Sort_and_Count(B)
7         (r, L) ← Merge_and_Count(A, B)
8         r ← r + r_A + r_B fi
9 return (r, L)
```

složitost algoritmu

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(n/2) + \Theta(n) & \text{jinak} \end{cases}$$

$$T(n) = \mathcal{O}(n \log n)$$

Řazení

Řazení rozdělováním - Quicksort

Rozděl posloupnost $A[p \dots r]$ na dvě podposloupnosti $A[p \dots q-1]$ a $A[q \dots r]$ tak, aby všechny prvky v $A[p \dots q-1]$ byly menší nejvýše rovné prvkům v $A[q \dots r]$

Vyřeš obě posloupnosti (rekurzivně) seřaď

Kombinuj protože obě podposloupnosti jsou seřazené, není nutný žádný další výpočet

Mergesort

Rozděl posloupnost na dvě posloupnosti poloviční velikosti.

Vyřeš obě podposloupnosti (rekurzivně) seřaď

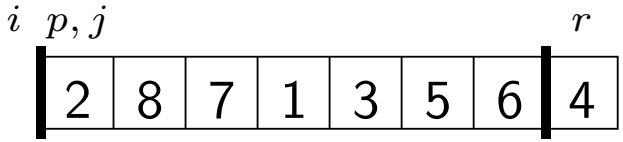
Kombinuj spoj dvě seřazené podposloupnosti do jedné

Quicksort

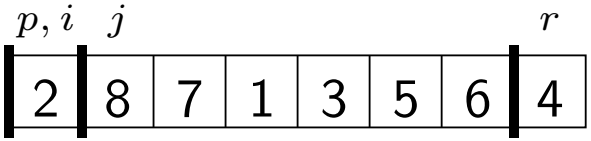
- hlavní částí algoritmu je rozdělování posloupnosti do dvou posloupností požadovaných vlastností
- při rozdělování využíváme **pivota**
- každý prvek posloupnosti porovnáváme s pivotem
- podposloupnosti prvků menších / větších než pivot

Quicksort

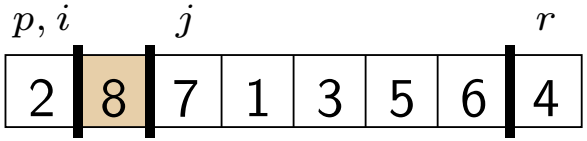
(a)



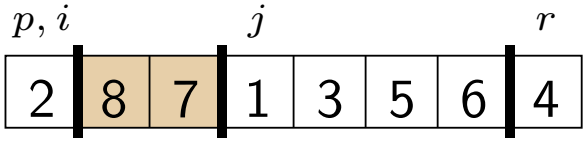
(b)



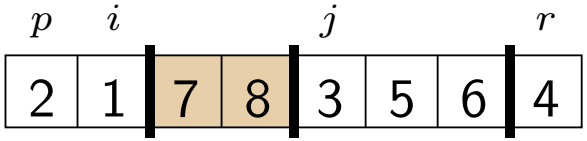
(c)



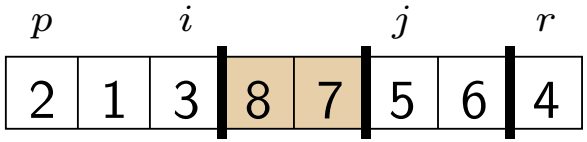
(d)



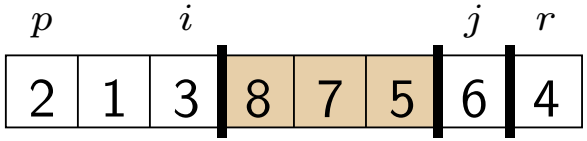
(e)



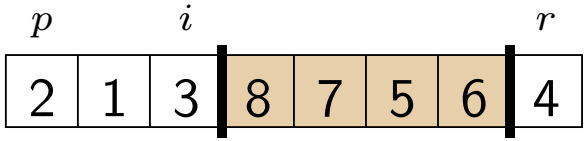
(f)



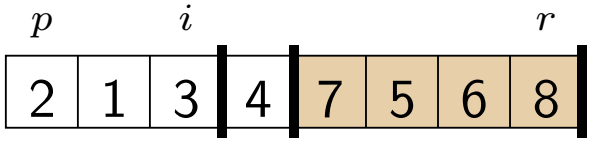
(g)



(h)



(i)



Quicksort(A, p, r)

```
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q − 1)
4         QUICKSORT(A, q + 1, r) fi
```

Partition(A, p, r)

```
1 pivot ← A[r]
2 i ← p − 1
3 for j = p to r do
4     if A[j] ≤ pivot then  i ← i + 1
5                           vyměň A[i] a A[j] fi od
6 return i
```

Korektnost

chceme dokázat, že procedura PARTITION vrátí index i takový, že

- $A[i] = pivot$
- pro $p \leq k \leq i$ platí $A[k] \leq A[i]$
- pro $i < k \leq r$ platí $A[k] > A[i]$

Invariant cyklu

na začátku každé iterace **for** cyklu v řádcích 3 - 6 platí pro každý index k

1. jestliže $p \leq k \leq i$, tak $A[k] \leq pivot$
2. jestliže $i + 1 \leq k \leq j - 1$, tak $A[k] > pivot$

Inicializace

iniciální přiřazení je $pivot \leftarrow A[r]$, $i \leftarrow p - 1$ a $j \leftarrow p$
invariant (triviálně) platí

Korektnost

Invariant cyklu

na začátku každé iterace **for** cyklu v řádcích 3 - 6 platí pro každý index k

1. jestliže $p \leq k \leq i$, tak $A[k] \leq pivot$
2. jestliže $i + 1 \leq k \leq j - 1$, tak $A[k] > pivot$

Iterace

$A[j] > pivot$ - efektem iterace cyklu je zvýšení hodnoty j o 1; invariant platí

$A[j] \leq pivot$ - efektem iterace cyklu je zvýšení hodnoty i a výměna $A[i]$ s $A[j]$, to garantuje zachování platnosti podmínky 1
zachování platnosti podmínky 2 garantuje fakt, že jsme do $A[j-1]$ přesunuli prvek větší než $pivot$

Korektnost

Invariant cyklu

na začátku každé iterace **for** cyklu v řádcích 3 - 6 platí pro každý index k

1. jestliže $p \leq k \leq i$, tak $A[k] \leq pivot$
2. jestliže $i + 1 \leq k \leq j - 1$, tak $A[k] > pivot$

Ukončení

výpočet končí když $j = r + 1$, což spolu s faktem, že po posledním provedení iterace platí invariant, garantuje, že pro $p \leq k \leq i$ platí $A[k] \leq A[i]$ a pro $i < k \leq r$ platí $A[k] > A[i]$

v poslední iteraci je $j = p$, $A[j] = pivot \leq pivot$, provede se výměna $A[i]$ s $A[j]$, což garantuje, že po ukončení výpočtu cyklu platí $A[i] = pivot$

Složitost

složitost v nejhorším případě

např. pro vstupní posloupnost, která je již seřazená, nebo která obsahuje stejné prvky

$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$

$T(n) = \Theta(n^2)$

složitost v nejlepším případě

nastává, když při každém rekurzivním volání rozdělí pivot posloupnost na dvě stejně velké podposloupnosti

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

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

průměrná složitost

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

Alternativní postup rozdělování I

- postupujeme od obou konců posloupnosti až do chvíle, než jsou detekovány dva prvky, které jsou vůči sobě v opačném pořadí; prvky si vymění svou pozici
- při tomto postupu se udělá průměrně 3 krát méně výměn
- algoritmus není stabilní

Hoare Partition(A, p, r)

```
1 x ← A[p]
2 i ← p − 1
3 j ← r + 1
4 while true do
5         repeat j ← j − 1 until A[j] ≤ x od
6         repeat i ← i + 1 until A[i] ≥ x od
7         if i < j then swap A[i] a A[j] else return j fi od
```

Quicksort(A, p, r)

```
1 if p < r then q ← Hoare Partition(A, p, r)
2                 Quicksort(A, p, q)
3                 Quicksort(A, q + 1, r) fi
```

Alternativní postup rozdělování II

- obě uvedená schémata se chovají špatně v případě, že ve vstupní posloupnosti se prvky opakují

- rozdělovací schéma, které řeší posloupnosti s opakujícími se prvky
- při rozdělování se hledají prvky menší než pivot a větší než pivot
- prvky stejné jako pivot jsou již na své pozici
- prvky menší (větší) než pivot se seřadí rekurzivně

Iterativní verze Quicksortu

algoritmus ve tvaru *tail* rekurze

Tail Recursive Quicksort(A, p, r)

```
1 while p < r do
2         q ← Partition(A, p, r)
3         Tail Recursive Quicksort(A, p, q − 1)
4         p ← q + 1 od
```

Iterative Quicksort(A, p, r)

```
1 stack = [ ]
2 stack.push(p, r)
3 while stack do
4         pos = stack.pop()
5         p, r = pos[1], pos[2]
6         q ← Partition(A, p, r)
7         if q − 1 > p then stack.push((p, q − 1)) fi
8         if q + 1 < r then stack.push((q + 1, r)) fi
9 od
```

Složitost problému řazení

složitost řadících algoritmů založených na vzájemném porovnávání prvků posloupnosti je $\Omega(n \log n)$

1. búno vstupní posloupnost $a_1, \ldots, a_n$ obsahuje vzájemně různé prvky
2. každé porovnání určí větší ze dvou prvků
3. výpočet algoritmu můžeme popsat *rozhodovacím stromem*, jehož vnitřní vrcholy jsou označeny y porovnávaných prvků a mají dva syny odpovídající vztahu $<$ a $>$
4. výpočet na konkrétním vstupu představuje cestu v rozhodovacím stromě z kořene do listu; jeho složitost je úměrná délce cesty
5. každý list jednoznačně určuje seřazení $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ vstupních prvků
6. algoritmus musí mít možnost vypočítat každou možnou permutaci vstupních prvků
7. počet různých permutací je $n!$
8. strom musí mít alespoň $n!$ listů $\Rightarrow$ má hloubku alespoň $\log(n!) = \Omega(n \log n)$

Řazení

Řazení haldou - Heapsort

idea

- cílem je seřadit posloupnost prvků
- najdeme největší prvek x posloupnosti P
- prvek x přidáme na začátek posloupnosti již seřazených prvků
- odstraníme prvek x z P
- postup opakujeme dokud posloupnost P není prázdná

co potřebujeme

- datovou strukturu nad kterou dokážeme
  1. efektivně najít největší prvek a
  2. efektivně z ní odstranit největší prvek
- **halda**

co je halda

- halda je stromová datová struktura splňující vlastnost haldy

Kořenový strom

- strom s vyznačeným vrcholem r nazýváme[2] kořenovým stromem s kořenem r
- u kořenových stromů používáme pojmy *rodič*, *děti/synové*, *sourozenci*, *potomek*
- kořen nemá žádného rodiče, ostatní vrcholy jsou potomky kořene
- listem je každý vrchol, který nemá potomky
- místo slova vrchol často používáme termín *uzel*

- podstrom určený vrcholem x je podgraf indukovaný všemi následníky vrcholu x; tento podstrom je opět kořenovým stromem s kořenem x

[2] definice viz učebný text Matematické Základy Informatiky (FI:IB000) prof. Hliněného

Kořenový strom

stupeň vrcholu v kořenovém stromě T je počet jeho synů

hloubka vrcholu x v T je délka cesty (*tj. počet hran*) z kořene do x; kořen je tedy v hloubce nula

výška vrcholu x v T je délka nejdelší cesty z x do listu; list má tedy výšku nula

hloubka stromu T = **výška stromu** T je délka nejdelší cesty od kořene k listu

k-tá hladina stromu T je množina všech vrcholů stromu T ležících v hloubce k; hladiny začínáme počítat od nulté

binární strom je strom, ve kterém má každý vrchol nejvýše dva syny; tyto často označujeme jako levého a pravého syna

k-ární strom je strom, ve kterém má každý vrchol nejvýše k synů

Stromová datová struktura

- stromová datová struktura je reprezentace stromu v počítači
- při reprezentaci stromu v počítači je důležité, abychom se z každého vrcholu uměli dostat k jeho synům a z každého vrcholu, kromě kořene, k jeho rodiči
- strom můžeme reprezentovat dynamicky pomocí ukazatelů (pointrů) a nebo staticky v poli

- v každém vrcholu v stromu si pamatujeme hodnotu $v.key$, které se říká klíč; v případě potřeby si můžeme pamatovat i další hodnoty

Halda a binární halda

halda

- je stromová datová struktura splňující **vlastnost haldy**
- kořenový strom má vlastnost haldy právě tehdy, když pro každý uzel v a pro každého jeho syna w platí $v.key \geq w.key$
- díky této vlastnosti obsahuje kořen stromu největší klíč z celé haldy

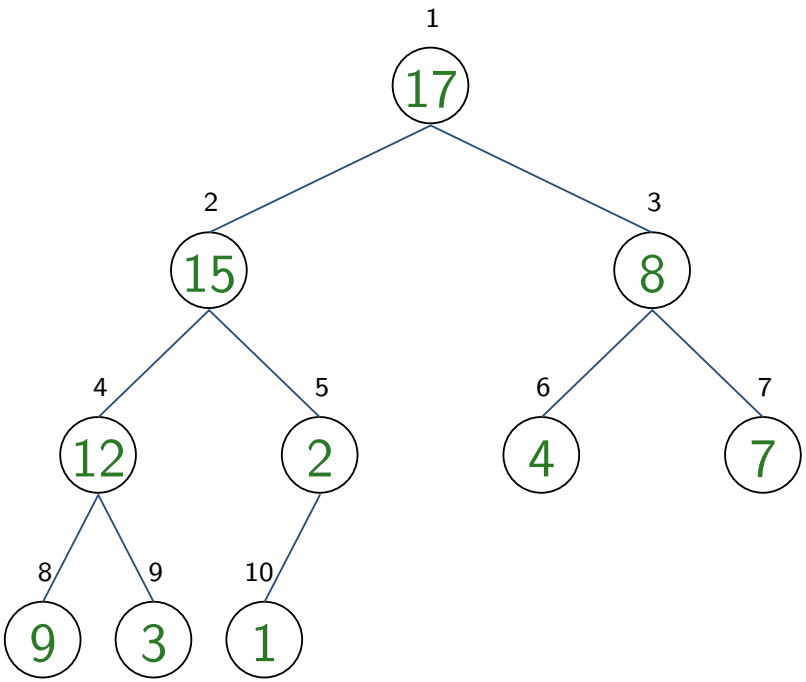
binární halda

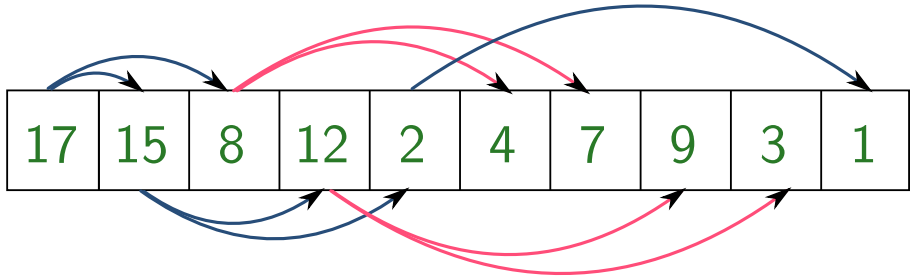
- je **úplný** binární strom s vlastností haldy
- binární strom je úplný, pokud jsou všechny jeho hladiny kromě poslední úplně zaplněny a v poslední hladině leží listy co nejvíce vlevo

maximová vs. minimová halda
d-regulární halda, binomiální halda, Fibonacciho halda

Binární halda a její reprezentace v poli

- prvky pole A odpovídají uzlům binárního stromu
- uzly očíslujeme po hladinách počínaje od jedničky; klíč z uzlu i uložíme do $A[i]$
- levý syn uzlu k bude uložen na pozici $2k$ a pravý syn uzlu k na pozici $2k + 1$
- otec uzlu k se bude nacházet na pozici $\lfloor k/2 \rfloor$





Binární halda a její reprezentace v poli

pole reprezentující haldu má atributy

- $A.length$ je počet prvků v poli
- $A.heap\_size$ je počet prvků haldy uložených v poli
- prvky haldy jsou v poli uloženy na pozicích $A[1 \ldots A.heap\_size]$

pro daný index i vypočteme indexy synů a otce uzlu $A[i]$ předpisem

$\textsc{Parent}(i)$

return $\lfloor i/2 \rfloor$

$\textsc{Left}(i)$

return $2i$

$\textsc{Right}(i)$

return $2i + 1$

Operace nad haldou

1. jak vybudovat haldu?
2. jak využít haldu k seřazení posloupnosti?

vybudování haldy

vstupem je posloupnost klíčů uložená v poli $A[1 \dots n]$
klíče v poli preuspořádáme tak, aby na konci výpočtu tvořili haldu

varianta A

v odpovídajícím binárním stromu postupujeme od listů směrem ke kořeni
operace MAX_HEAPIFY
časová složitost $\mathcal{O}(n)$

varianta B

v odpovídajícím binárním stromu postupujeme od kořene směrem k listům
operace INSERT
časová složitost $\mathcal{O}(n \log n)$

Vybudování haldy

varianta A

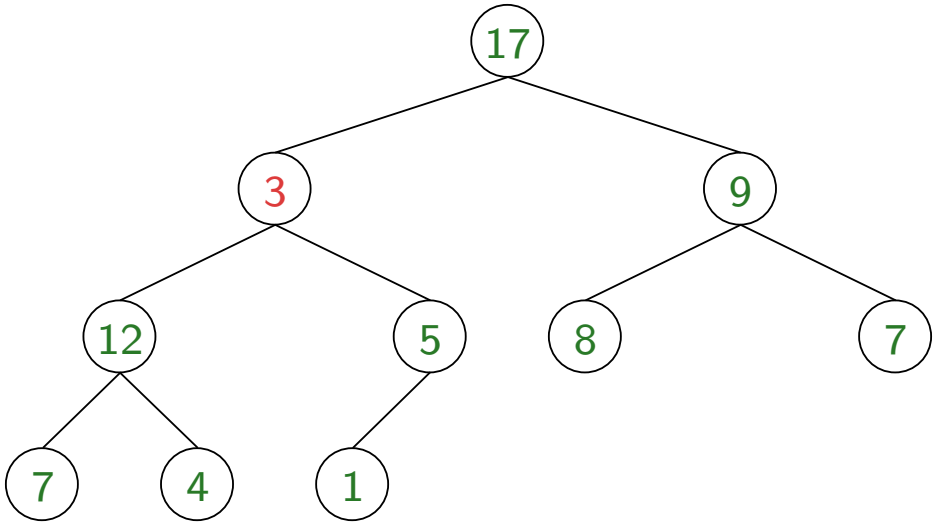
invariant

- v každém kroku algoritmu splňují všechny uzly j, pro $i \leq j \leq n$, vlastnost haldy
- v následujícím kroku necháme klíč z uzlu $i-1$ přebublat dolů takže také uzel $i-1$ bude splňovat vlastnost haldy

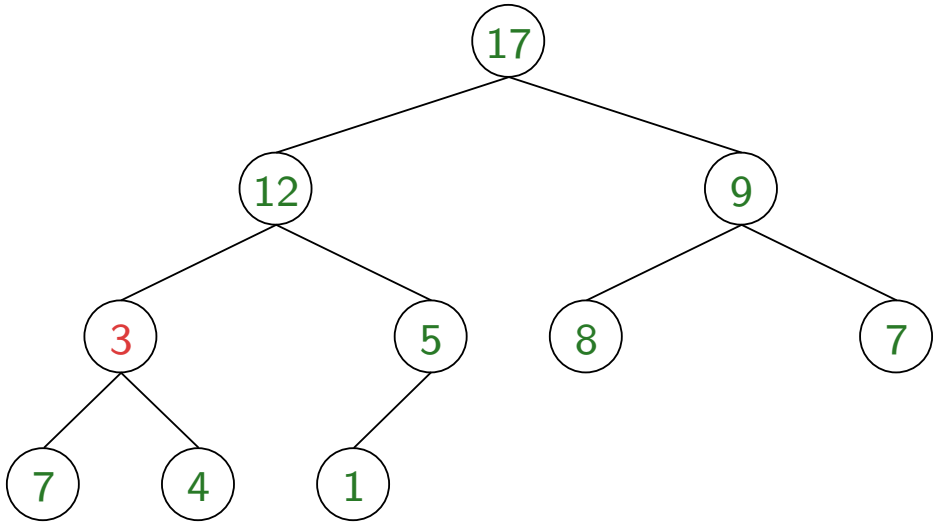
procedura $\textsc{Max\_Heapify}(A, i)$

- předpokládá, že binární stromy s kořeny $\textsc{Left}(i)$ a $\textsc{Right}(i)$ mají vlastnost haldy a že klíč $A[i]$ může být menší než jeho následníci, tj. nemusí splňovat vlastnost haldy
- procedura modifikuje A tak, že po její provedení strom s kořenem $A[i]$ má vlastnost haldy
- úprava je založena na přesunu prvku $A[i]$ směrem dolů

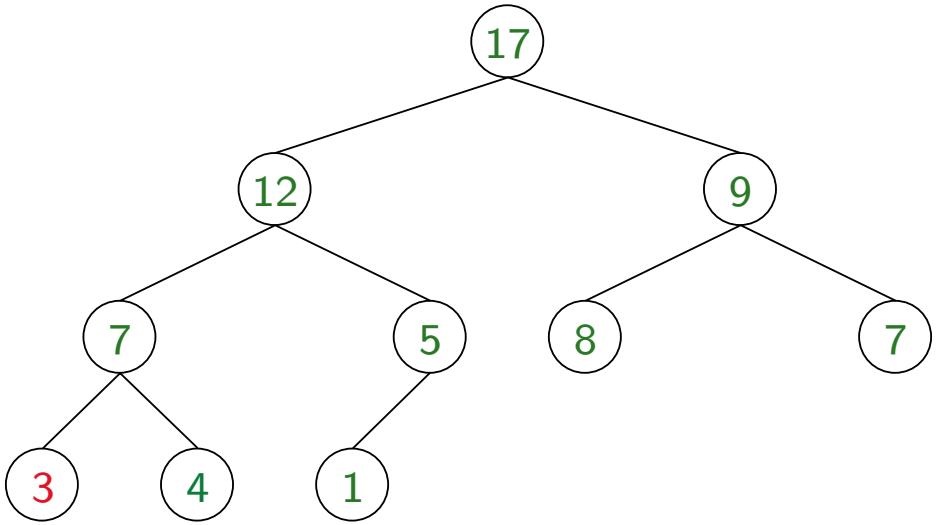
Max_Heapify



(a)



(b)



(c)

Max_Heapify

Max_Heapify(A, i)

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ A.heap_size ∧ A[l] > A[i]
4     then largest ← l
5      else largest ← i fi
6  if r ≤ A.heap_size ∧ A[r] > A[largest]
7     then largest ← r fi
8  if largest ≠ i
9     then swap A[i] a A[largest]
10          MAX_HEAPIFY(A, largest) fi
```

složitost operace je $\mathcal{O}(h)$, kde h je hloubka stromu s kořenem $A[i]$

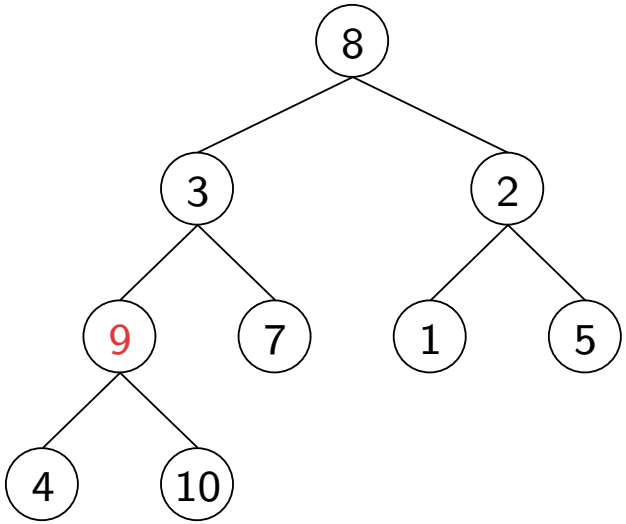
Vybudování haldy

- využitím procedury MAX_HEAPIFY zkonvertujeme pole $A[1 \dots n]$ na maximovou haldu
- prvky $A[\lfloor n/2 \rfloor + 1], A[\lfloor n/2 \rfloor + 2] \dots A[n]$ jsou listy stromu a proto každý tvoří haldu s 1 vrcholem
- proceduru MAX_HEAPIFY aplikujeme na zbylé prvky pole v pořadí odspodu směrem nahoru a na dané úrovni směrem zprava doleva

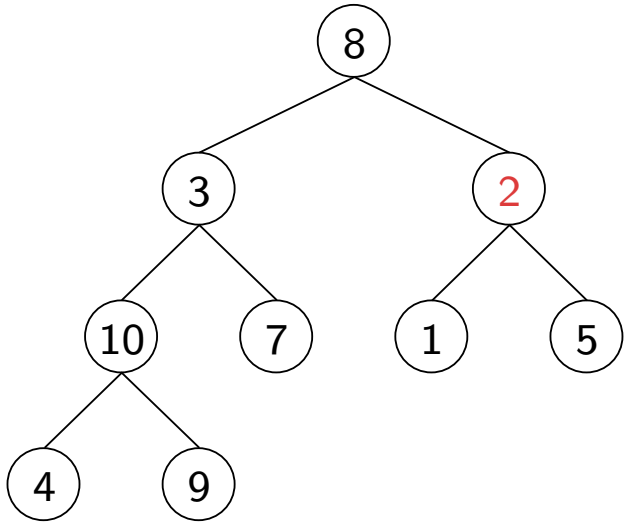
Build_Max_Heap(A)

```
1 A.heap_size ← A.length
2 for i = ⌊A.length/2⌋ downto 1 do
3     MAX_HEAPIFY(A, i) od
```

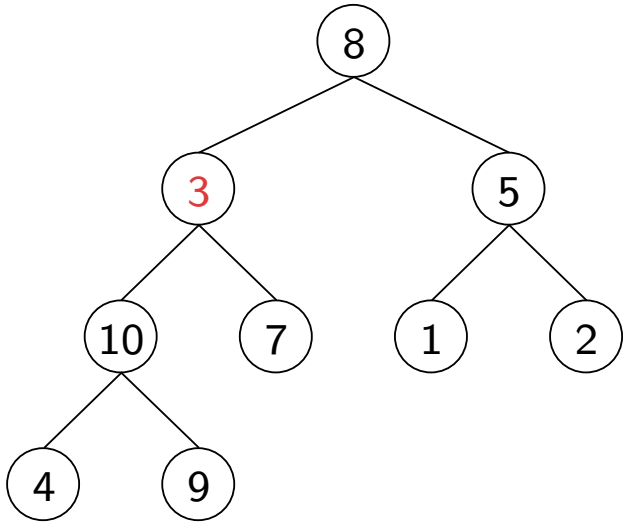
Vybudování haldy



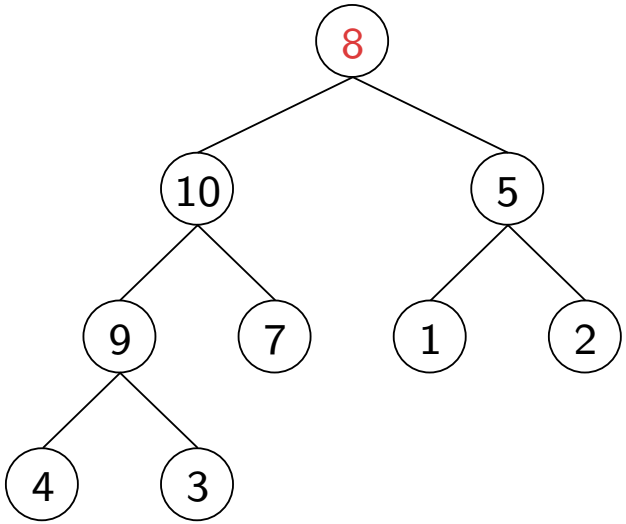
(a)



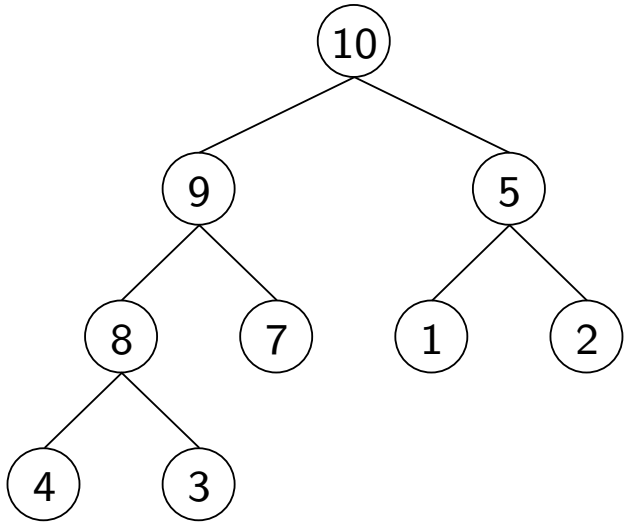
(b)



(c)



(d)



(e)

Build_Max_Heap - korektnost

Invariant cyklu

Na začátku každé iterace **for** cyklu je každý z vrcholů $A[i+1], A[i+2], \ldots, A[n]$ kořenem maximové haldy.

Inicializace na začátku je $i = \lfloor n/2 \rfloor$, vrcholy $A[\lfloor n/2 \rfloor + 1]$, $A[\lfloor n/2 \rfloor + 2]$, $\ldots$,$A[n]$ jsou listy a jsou tedy kořeny (triviální) haldy

Iterace levý a pravý podstrom vrcholu $A[i]$ jsou maximové haldy (platí pro ně invariant), vlastnost haldy může být porušena jedině hodnotou $A[i]$; procedura $\textsc{Max\_Heapify}(A, i)$ vybuduje maximovou haldu s kořenem i

Ukončení cyklus skončí když $i = 0$ a z platnosti invariantu plyne, že A je maximová haldu

Build_Max_Heap - složitost

- složitost procedury MAX_HEAPIFY je $\mathcal{O}(h)$, kde h je hloubka stromu, na který se procedura aplikuje
- počet podstromů hloubky h je nejvýše $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
- kořen má hloubku $\lfloor \log n \rfloor$
- celková složitost je proto

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = \mathcal{O}(n)$$

při zjednodušování výrazu jsme využili rovnost

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

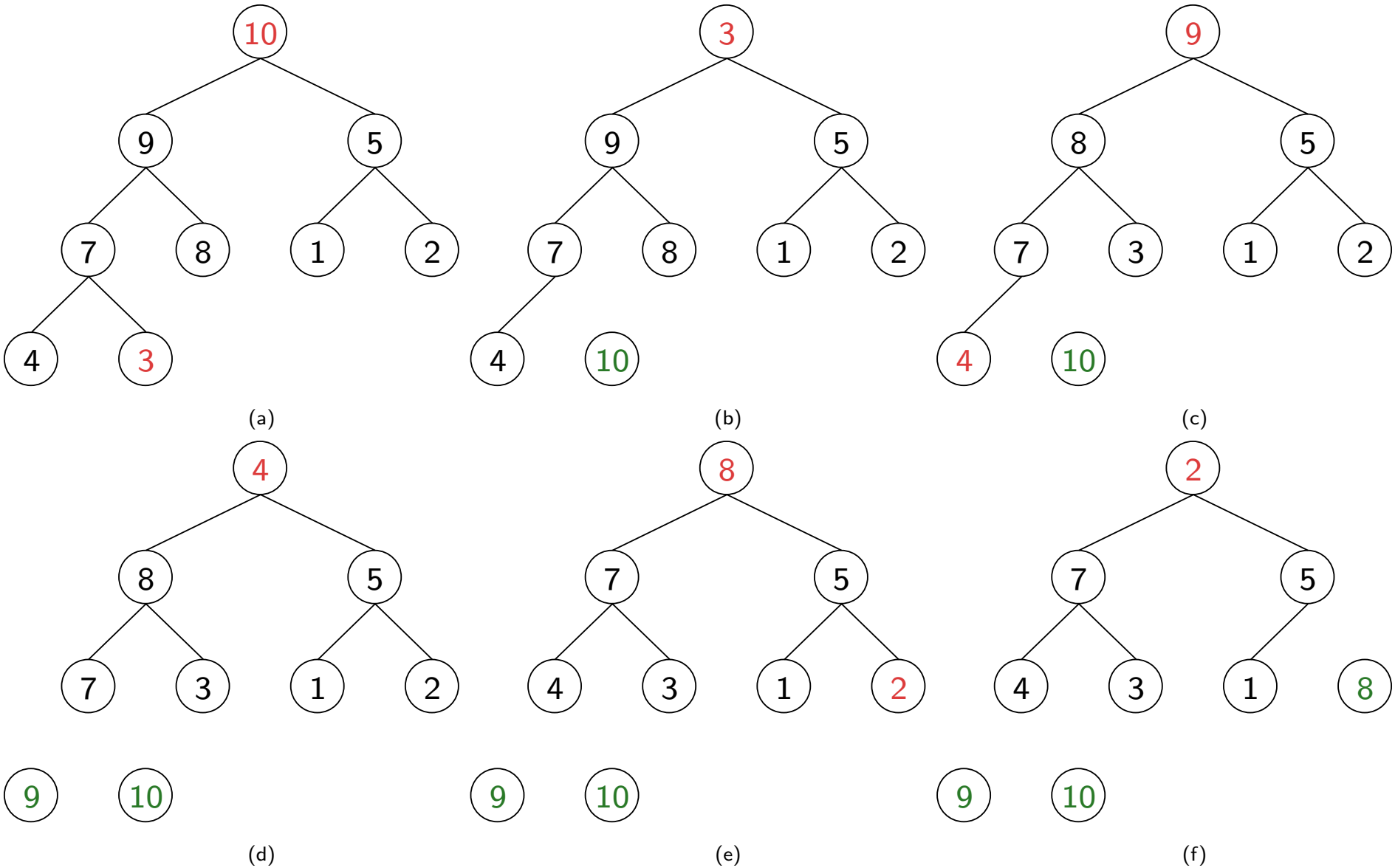
Algoritmus řazení haldou, Heapsort

- použitím procedury BUILD_MAX_HEAP vybudujeme haldu nad polem $A[1 \ldots n]$, kde $n = A.length$
- maximální prvek pole A je uložený v kořeni $A[1]$ a proto ho můžeme přesunout na jeho finální pozici $A[n]$ (vyměníme prvky $A[1]$ a $A[n]$)
- prvek, který jsme přesunuli do kořene, může porušit vlastnost haldy a pro obnovení vlastnosti haldy použijeme MAX_HEAPIFY$(A, 1)$
- celý proces opakujeme pro haldu velikosti $n - 1$

Heapsort(A)

```
1 BUILD_MAX_HEAP(A)
2 for i = A.length downto 2 do vyměň A[1] a A[i]
3     A.heap_size ← A.heap_size − 1
4     MAX_HEAPIFY(A, 1) od
```

Heapsort


10
9
5
7
8
1
2
4
3
(a)
3
9
5
7
8
1
2
4
10
(b)
9
8
5
7
3
1
2
4
10
(c)
4
8
5
7
3
1
2
9
10
(d)
8
7
5
4
3
1
2
9
10
(e)
2
7
5
4
3
1
8
9
10
(f)

Heapsort - složitost

- procedura BUILD_MAX_HEAP má složitost $\mathcal{O}(n)$
- každé z $n-1$ volaní procedury MAX_HEAPIFY má složitost $\mathcal{O}(\log n)$
- algoritmus HEAPSORT má složitost $\mathcal{O}(n \log n)$

Optimalizace a varianty

- strom vyšší arity
- ve vrcholu stromu uložených několik hodnot

- **budování haldy výměnou zdola nahoru** halda je na začátku prázdná a postupně do ní vkládáme prvky vstupní posloupnosti; prvek vložíme na poslední místo (jako list) a v případě porušení vlastnosti haldy ho (rekurzivně) zaměníme s jeho rodičem; časová složitost vybudování haldy je $\Theta(n \log n)$
- **bottom - up heapsort** optimalizuje etapu seřazování prvků; maximální prvek z kořene si vymění místo s posledním prvkem haldy a pro obnovení vlastnosti haldy výměnami se postupuje zdola nahoru
- **Smoothsort** (Edsger Dijkstra) - stejná asymptotická složitost, lepší chování pro vstupní posloupnosti, které jsou téměř uspořádané

Prioritní fronty

- datová struktura pro reprezentaci množiny, nad prvky množiny je definováno uspořádání
- umožňuje efektivní realizaci operací:
  $\textsc{Insert}(S, x)$ vloží prvek x do množiny S
  $\textsc{Maximum}(S)$ vrátí největší prvek množiny S
  $\textsc{Extract\_Max}(S)$ odstraní z množiny S největší prvek
  $\textsc{Increase\_Key}(S, x, k)$ nahradí prvek x prvkem k za předpokladu, že $k \geq x$
- alternativně můžeme definovat prioritní frontu vůči minimálnímu prvku
- **prioritní frontu implementujeme jako maximovou haldu**

Maximum a Extract_Max

- prvky množiny S tvoří haldu A
- maximální prvek haldy je v jejím kořeni; jeho nalezení má konstantní složitost

Heap_Maximum(A)

1 **return** $A[1]$

- odstranění maximálního prvku se implementuje stejně jako v algoritmu řazení
- složitost operace je $\mathcal{O}(\log n)$

Heap_Extract_Max(A)

1 **if** $A.heap\_size < 1$ **then return** prázdná fronta **fi**
2 $max \leftarrow A[1]$
3 $A[1] \leftarrow A[A.heap\_size]$
4 $A.heap\_size \leftarrow A.heap\_size - 1$
5 $\textsc{Max\_Heapify}(A, 1)$
6 **return** max

Increase_Key

- procedura HEAP_INCREASE_KEY implementuje operaci INCREASE_KEY
- index i identifikuje prvek, který má být operací nahrazen (navýšen)
- nejdříve změníme hodnotu $A[i]$ na novou hodnotu key a potom obnovíme vlastnost haldy

Heap_Increase_Key(A, i, key)

```
1 if key < A[i] then return  nová hodnota je menší než původní fi
2 A[i] ← key
3 while i > 1 ∧ A[Parent(i)] < A[i] do
4         vyměň A[i] a A[Parent(i)]
5         i ← Parent(i) od
```

složitost: $\mathcal{O}(\log n)$

Insert

- procedura MAX_HEAP_INSERT implementuje operaci INSERT
- na konec pole vložíme nový prvek, který je menší než všechny ostatní prvky, symbolicky ho označujeme $-\infty$
- zvýšíme hodnotu vloženého prvku na hodnotu prvku, který chceme vložit do fronty

Max_Heap_Insert(A, key)

1 $A.heap\_size \leftarrow A.heap\_size + 1$
2 $A[A.heap\_size] \leftarrow -\infty$
3 HEAP_INCREASE_KEY$(A, A.heap_size, key)$

složitost: $\mathcal{O}(\log n)$

Řazení

Řazení počítáním - Counting Sort

předpokládá, že vstupní posloupnost obsahuje celá čísla z intervalu $0 \ldots k$, kde k je nějaké pevně dané přirozené číslo

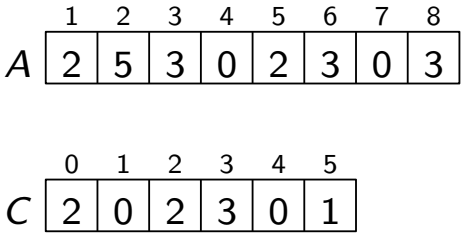
jestliže $k = \mathcal{O}(n)$, tak složitost řazení počítáním je $\Theta(n)$

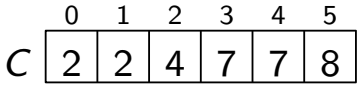
Counting sort

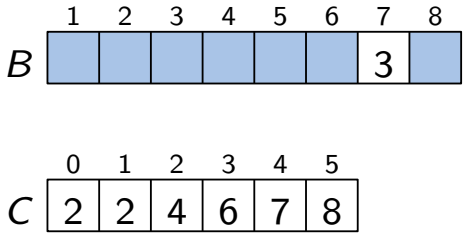
- vstupní posloupnost $A[1 \ldots n]$
- pole $B[1 \ldots n]$ obsahuje seřazenou posloupnost
- pole $C[0 \ldots k]$ se využívá v průběhu výpočtu

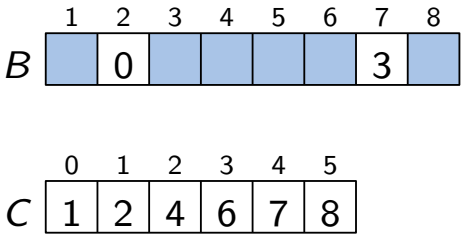
- pro každou hodnotu $i = 0, 1, \ldots, k$ spočítáme, kolik je ve vstupní posloupnosti čísel i, výsledný počet uložíme do $C[i]$
- pro každou hodnotu $i = 0, 1, \ldots, k$ spočítáme, kolik je ve vstupní posloupnosti čísel **menších nebo rovných** i, využijeme k tomu hodnoty napočítané v předcházejícím kroku a výsledný počet uložíme opět do $C[i]$
- procházíme vstupní posloupnost od konce a každé číslo uložíme do B přímo na jeho pozici, která je určená počtem menších nebo rovných čísel; hodnoty v C průběžně aktualizujeme

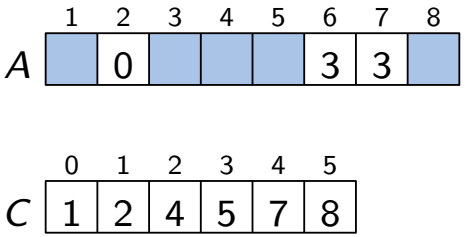
Counting sort

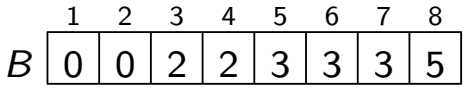












Counting_Sort(A, B, k)

```
1  //inicializace C[0...k]
2  for i = 0 to k do
3      C[i] ← 0 od
4  for j = 1 to A.length do
5      C[A[j]] ← C[A[j]] + 1 od
6  //C[i] obsahuje počet čísel rovných i
7  for i = 1 to k do
8      C[i] ← C[i] + C[i − 1] od
9  //C[i] obsahuje počet čísel menších nebo rovných i
10 for j = A.length downto 1 do
11     B[C[A[j]]] ← A[j]
12     C[A[j]] ← C[A[j]] − 1 od
```

Časová složitost

- cyklus na řádcích 2 - 3 (inicializace C) – složitost $\Theta(k)$
- cyklus na řádcích 4 - 5 (počet čísel $= i$) – složitost $\Theta(n)$
- cyklus na řádcích 7 - 8 (počet čísel $\leq i$) – složitost $\Theta(k)$
- cyklus na řádcích 10 - 12 (přesun z A do B) – složitost $\Theta(n)$
- celková složitost $\Theta(k + n)$

- v praxi používaný pro $k = \mathcal{O}(n)$
- stabilní algoritmus: prvky se stejnou hodnotou se ve výstupní posloupnosti vyskytují ve stejném pořadí jako ve vstupní posloupnosti *(důležité např. pro radix sort)*

Varianty a optimalizace

- v případě, že vstupní posloupnost obsahuje pouze čísla (a ne složitější datové objekty s klíčem), tak druhý cyklus algoritmu je možné vynechat a zapisovat do pole B přímo čísla
- algoritmus se dá využít k odstraňování duplicitních klíčů (pole C nahradíme bitovým polem)
- umožňuje efektivní paralelizaci (vstupní posloupnost rozdělíme na stejně velké podposloupnosti a pro každou z nich počítáme frekvence výskytu paralelně)
- extrasekvenční složitost algoritmu je $\mathcal{O}(n + k)$

Číslicové řazení - Radix Sort

- řazení čísel podle číslic na jednotlivých bitech
- postup zleva doprava (most significant digit, MSD) - používá se např. pro lexikografické uspořádání
- postup zprava doleva (least significant digit, LSD), stabilní řazení
- dá se použít i pro řazení položek, které nemají číselný charakter
- používá se např. když potřebujeme seřadit položky vzhledem k různým klíčům

Radix_Sort(A, d)

```
1 for i = 1 to d do
2     použij stabilní řazení a seřaď položky podle ite číslice
3 od
```

Radix Sort

Lema 1

Danou posloupnost n čísel s d číslicemi, přičemž číslice můžou nabývat k různých hodnot, seřadí RADIX_SORT *korektně v čase $\Theta(d(n+k))$ za předpokladu, že stabilní řazení, které využívá, má složitost $\Theta(n+k)$.*

- složitost je garantovaná např. při použití algoritmu Counting sort
- jestliže d je konstanta a $k = \mathcal{O}(n)$, pak časová složitost číslicového řazení je lineární

Varianty

- řazení binárních čísel (*lze zobecnit pro libovolnou číselnou soustavu*)
- nechť každé číslo má b bitů, zvolíme $r \leq b$
- číslo rozdělíme na $\lceil b/r \rceil$ skupin po r bitech
- každou skupinu chápeme jako číslo z intervalu 0 až $2^r - 1$
- při řazení postupujeme po skupinách, použijeme Counting sort pro $k = 2^r - 1$

Lema 2

Danou posloupnost n binárních b bitových čísel RADIX_SORT *korektně seřadí v čase $\Theta((b/r)(n + 2^r))$ za předpokladu, že stabilní řazení, které využívá, má složitost $\Theta(n + k)$ pro čísla z intervalu 0 až k.*

otázka vhodné volby parametru r pro dané n a b závisí od poměru veličin n a b

$\lfloor b < \log n \rfloor$ pro $r \leq b$ platí $(n + 2^r) = \Theta(n)$, optimální je proto volba $r = b$ pro kterou je celková složitost číslicového řazení $(b/b)(n + 2^b) = \Theta(n)$

$\lfloor b \geq \log n \rfloor$
- pro $r = \lfloor \log n \rfloor$ je složitost je $\Theta(bn/\log n)$
- pro $r > \lfloor \log n \rfloor$ je složitost je $\Omega(bn/\log n)$
- pro $r < \lfloor \log n \rfloor$ hodnota výrazu (b/r) klesá a hodnota výrazu $n + 2^r$ zůstává $\Theta(n)$

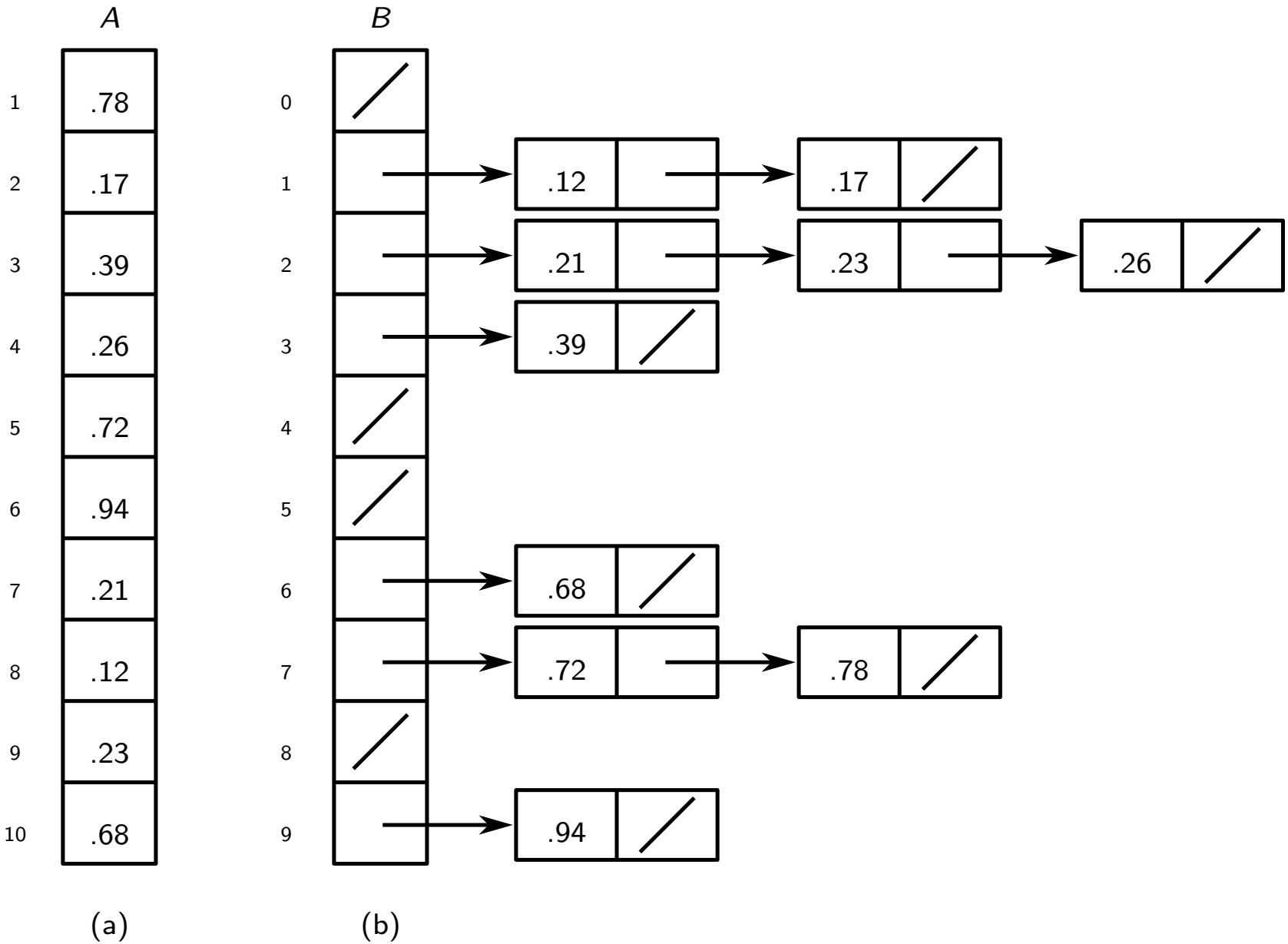
Přihrádkové řazení - Bucket Sort

předpokládá, že

- vstupní posloupnost obsahuje čísla z intervalu $[0 \ldots 1)$
- čísla rovnoměrně pokrývají celý interval

- interval $[0 \ldots 1)$ rozdělíme na stejně velké podintervaly - **koše**
- vstupní čísla rozdělíme dle jejich hodnoty do košů
- seřadíme prvky v každém koši

Bucket sort


A
B
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94
(a)
(b)

Bucket sort

Bucket_Sort(A)

```
1 //B[0...n-1] je nové pole
2 n ← A.length
3 for i = 0 to n - 1 do
4     B[i] ← prázdny seznam od
5 for i = 1 to n do
6     přidej A[i] do seznamu B[⌊n · A[i]⌋] od
7 for i = 0 to n - 1 do
8     seřaď prvky seznamu B[i] použitím řazení vkládáním od
9 spoj seznamy B[0], B[1], ..., B[n - 1] do jednoho seznamu
```

- nechť n_i označuje počet prvků v koši $B[i]$
- složitost je $T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)$
- očekávaná složitost je pro vstup s uniformně rozdělenými čísly $\Theta(n)$

Datové typy

- jaká data jsou potřebné pro řešení problému?
- jak se budou data reprezentovat?
- jaké operace se budou nad daty provádět?

Datový typ

- rozsah hodnot, které může nabývat proměnná daného datového typu
- množina operací, které jsou pro daný datový typ povolené / definované
- nezávisí na konkrétní implementaci

Datové typy a struktury

jednoduchý (skalární) datový typ
data zabírají vždy konstantní (typicky malé) množství paměti, zpřístupnění hodnoty skalárního typu trvá konstantní čas
číselné a znakové typy, typ pravdivostních hodnot, výčtový typ

složený datový typ
implementace složeného datového typu se nazývá datová struktura

- **statický** - pevná velikost; časová složitost zpřístupnění prvku je konstantní
  k-tice, pole konstantní délky
- **dynamický** - neomezená velikost; časová složitost zpřístupnění prvku je funkcí závislou na velikosti
  seznam, zásobník, fronta, slovník, strom, graf

Dynamické datové typy

- množina objektů; v průběhu výpočtu můžeme do množiny prvky přidávat a odebírat resp. množinu jinak modifikovat (*tzv. dynamická množina)*
- každý prvek dynamické množiny je reprezentovaný jako objekt, jehož atributy můžeme zkoumat a modifikovat za předpokladu, že máme ukazatel / referenci na tento objekt
- jeden z atributů objektu je jeho identifikátor - klíč key
- jestliže všechny prvky mají různé klíče, často mluvíme o množině obsahující klíče

Dynamické datové typy - základní operace

SEARCH(S, k) pro množinu S a klíč k vrátí ukazatel x takový, že $x.key = k$ resp. NIL, když objekt s klíčem k není obsažen v množině S

INSERT(S, x) do množiny S vloží objekt s ukazatelem x

DELETE(S, x) z množiny S odstraní objekt s ukazatelem x

MAXIMUM(S) pro množinu S s úplně uspořádanými objekty vrátí ukazatel x na objekt, jehož klíč je maximální

MINIMUM(S) pro množinu S s úplně uspořádanými objekty vrátí ukazatel x na objekt, jehož klíč je minimální

SUCCESSOR(S, x) pro množinu S s úplně uspořádanými objekty vrátí ukazatel na objekt, jehož klíč následuje bezprostředně za klíčem $x.key$, resp. hodnotu NIL když x je maximální

PREDECESSOR(S, x) symetricky k SUCCESSOR

Datové struktury

Problém rezervací

online rezervační systém
(*např. rezervace lékařského vyšetření, přistávací ranveje, . . .*)

- množina rezervací R
- požadavek t na rezervaci
- rezervace může být potvrzena právě když v intervalu $(t-k, t+k)$ není žádná jiná rezervace (k je délka trvání události) a současně t je aktuální
- mazání realizovaných aktualizací

příklad: $R = \{21, 26, 29, 36\}$, $k = 3$, aktuální čas 20

rezervace
24 není validní (možná), protože $26 \in R$
33 je OK
15 není validní, protože aktuální čas je 20

Problém rezervací - řešení

jaký datový typ je vhodný pro reprezentaci R a realizaci požadovaných operací???

uspořádaný seznam
ověření rezervace v čase $\mathcal{O}(n)$, záznam rezervace v čase $\mathcal{O}(1)$

uspořádané pole
ověření rezervace v čase $\mathcal{O}(\log n)$, záznam rezervace v čase $\mathcal{O}(n)$

neuspořádaný seznam / pole
ověření rezervace v čase $\mathcal{O}(n)$, záznam rezervace v čase $\mathcal{O}(1)$

minimová halda
ověření rezervace v čase $\mathcal{O}(n)$, záznam rezervace v čase $\mathcal{O}(\log n)$, aktuálnost rezervace v čase $\mathcal{O}(1)$

binární pole rezervace t je uložena v položce s indexem t – problém velikosti pole

existuje lepší řešení?? současně efektivní vyhledávání i vkládání!

Vyhledávací stromy

- umožňují efektivní implementaci operací SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- operace nad vyhledávacím stromem mají složitost **úměrnou hloubce stromu**, tj. v nejhorším případě až lineární

- **binární vyhledávací stromy** - každý vrchol stromu má nejvýše 2 následníky, tj. strom může mít až hloubku n
- **vyvážené binární vyhledávací stromy** mají logaritmickou hloubku, tj. operace mají složitost $\mathcal{O}(\log n)$

Binární vyhledávací stromy (BVS)

- datová struktura, která využívá ukazatele
- každý vrchol (uzel) stromu představuje jeden objekt
- každý vrchol obsahuje
  - klíč
  - ukazatele *left*, *right* a *p* na levého syna, pravého syna a na otce; ukazatel má hodnotu *Nil* právě když vrchol nemá příslušného syna, resp. otce
  - případné další data

v binárním vyhledávacím stromu jsou klíče vždy uloženy tak, že platí

BVS vlastnost

jestliže x je vrchol BVS a

y je vrchol v levém podstromu vrcholu x, tak platí $y.key \leq x.key$

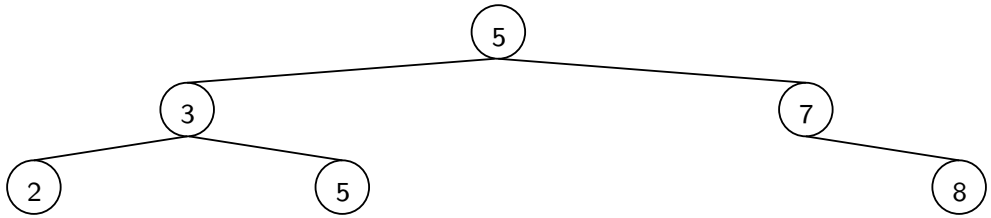
y je vrchol v pravém podstromu vrcholu x, tak platí $y.key \geq x.key$

BVS - procházení stromu

- cílem je projít strom tak, aby každý vrchol byl navštíven právě jednou
- využití: provedení operace nad každým vrcholem, výpis klíčů, kontrola vlastností stromu, ...

- strom procházíme rekurzivně
- začínáme v kořeni stromu
- (rekurzivně) navštívíme všechny vrcholy levého podstromu kořene
- (rekurzivně) navštívíme všechny vrcholy pravého podstromu kořene

BVS - výpis klíčů



klíče uložené v BVS můžeme vypsat v pořadí

inorder hodnotu klíče uloženého v kořeni vypíšeme **mezi** vypsáním klíčů uložených v jeho levém a pravém podstromě (2 3 5 5 7 8)

preorder hodnotu klíče uloženého v kořeni vypíšeme **před** vypsáním klíčů uložených v jeho levém a pravém podstromě (5 3 2 5 7 8)

postorder hodnotu klíče uloženého v kořeni vypíšeme **po** vypsání klíčů uložených v jeho levém a pravém podstromě (2 5 3 8 7 5)

Inorder

Inorder_Tree_Walk(x)

```
1 if x ≠ Nil
2    then INORDER_TREE_WALK(x.left)
3         print x.key
4         INORDER_TREE_WALK(x.right)
5 fi
```

- INORDER_TREE_WALK$(T.root)$ vypíše klíče uložené v BVS T **od nejmenšího po největší**
- časová složitost je $\Theta(n)$, kde n je počet vrcholů stromu T
- BVS SORT - časová složitost ???

BVS - vyhledávání ve stromu

- začínáme v kořeni stromu, postupujeme rekurzivně
- porovnáme hledaný klíč k s klíčem uloženým v navštíveném uzlu, jestliže se rovnají, tak vyhledávání končí úspěchem
- jestliže hledaný klíč k je **menší** než klíč $x.key$ uložený v navštíveném uzlu x, tak pokračujeme v **levém** podstromu uzlu x
- v opačném případě pokračujeme v pravém podstromu uzlu x
- vyhledávání končí neúspěchem právě když hledaný klíč není uložen ani v navštíveném listu

Tree_Search(x, k)

```
1 if x = Nil ∨ k = x.key
2    then return x fi
3 if k < x.key
4    then return TREE_SEARCH(x.left, k)
5     else return TREE_SEARCH(x.right, k) fi
```

BVS - minimální a maximální klíč

- jestliže hledáme **minimální** klíč, tak v stromu postupujeme vždy **doleva**
- jestliže hledáme **maximální** klíč, tak v stromu postupujeme vždy **doprava**

Tree_Minimum(x)

```
1 while x.left ≠ Nil do x ← x.left od
2 return x
```

Tree_Maximum(x)

```
1 while x.right ≠ Nil do x ← x.right od
2 return x
```

BVS - předchůdce a následník

- předpokládáme, že všechny klíče uložené v stromě jsou vzájemně různé[3]
- **následníkem** uzlu x je uzel, který obsahuje **nejmenší klíč větší než** $x.key$ *(successor)*
- **předchůdcem** uzlu x je uzel, který obsahuje **největší klíč menší než** $x.key$ *(predecessor)*

[3] analogicky se operace definují i pro strom, který může obsahovat uzly se stejnými klíči

následníkem uzlu x je uzel, který obsahuje **nejmenší klíč větší než** $x.key$

- jestliže uzel x má neprázdný pravý podstrom, tak jeho následníkem je nejmenší klíč uložený v jeho pravém podstromu
- jestliže pravý podstrom je prázdný, tak
  - následníkem x je uzel y takový, že $x.key$ je největším klíčem v levém podstromu uzlu y
  - uzel y je prvním uzlem na cestě z x do kořene stromu takový, že $y.key > x.key$ (*jinými slovy x patří do levého podstromu uzlu y*)

Tree_Successor(x)

```
1 if x.right ≠ Nil
2    then return TREE_MINIMUM(x.right) fi
3 y ← x.p
4 while y ≠ Nil ∧ x = y.right
5         do x ← y
6            y ← y.p
7 od
8 return y
```

BVS - přidání nového uzlu

- procházíme strom stejně jako kdybychom klíč nového uzlu vyhledávali
- hledáme vrchol, jehož příslušný podstrom je prázdný (levý podstrom, když klíč nového uzlu je menší než klíč vrcholu, pravý podstrom když je větší) a nový uzel se stane jeho příslušným synem

Tree_Insert(T, z)

```
1  y ← Nil
2  x ← T.root
3  while x ≠ Nil do
4          y ← x
5          if z.key < x.key then x ← x.left
6                           else x ← x.right fi
7  od
8  z.p ← y
9  if y = Nil then T.root ← z
10             else if z.key < y.key then y.left ← z
11                                   else y.right ← z fi
12 fi
```

BVS - odstranění uzlu - případ 1

při odstraňování uzlu z, můžou nastat 3 případy

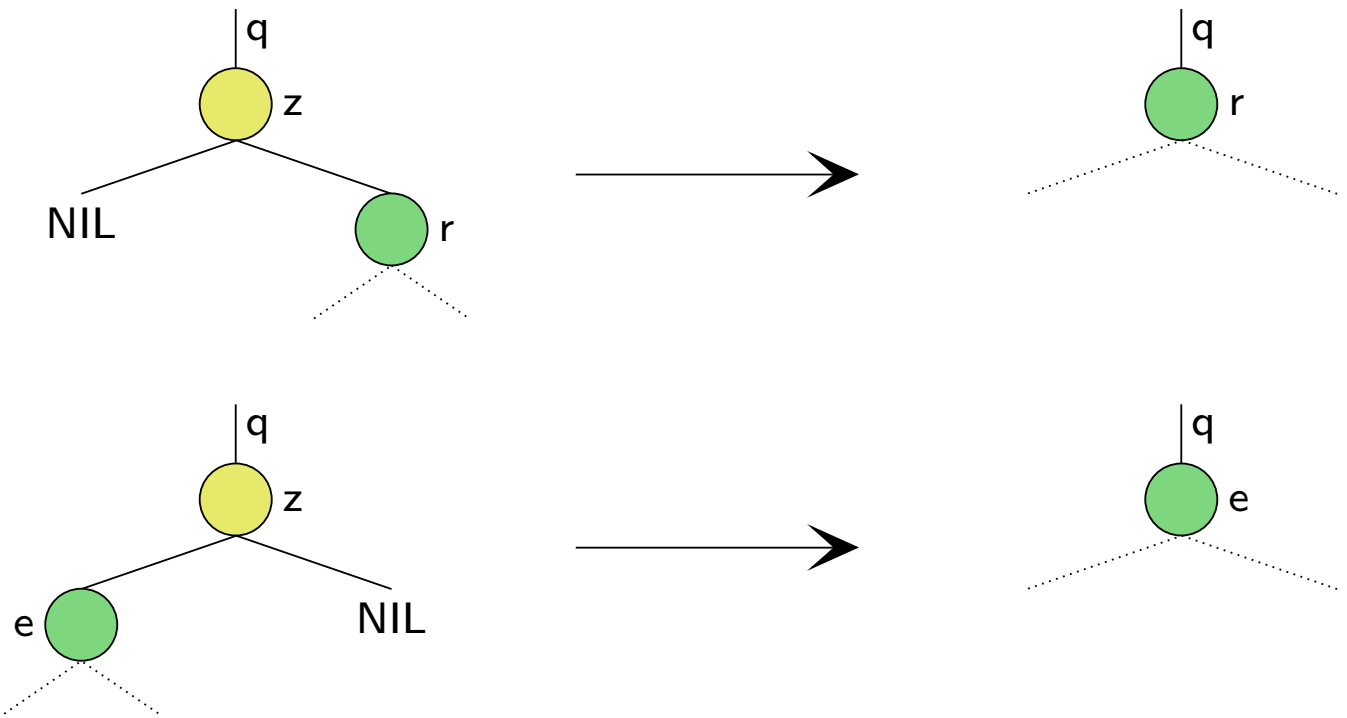
z nemá žádného syna

uzel odstraníme

BVS - odstranění uzlu - případ 2

z má jediného syna

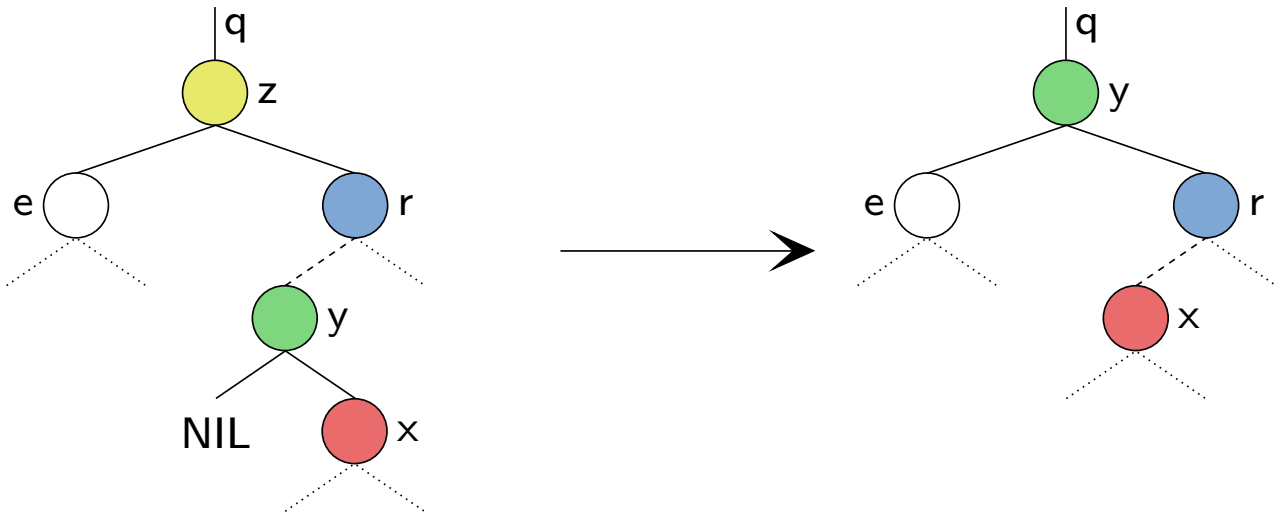
syna přesuneme na pozici uzlu z tak, že otec uzlu z se stane otcem jeho syna



BVS - odstranění uzlu - případ 3

z má dva syny

- potřebujeme najít uzel y, který nahradí uzel z
- vhodným kandidátem na y je následník uzlu z *(symetricky bychom mohli využít předchůdce uzlu z)*
- protože pravý podstrom uzlu z je neprázdný, tak následník y uzlu z je nejmenším uzlem v pravém podstromě uzlu z
- y nemá levého syna, proto ho můžeme přesunout na pozici z



BVS - přesun podstromů

- TRANSPLANT nahradí podstrom s kořenem u podstromem s kořenem v
- otcem uzlu v se stane otec uzlu u
- otec uzlu u bude mít uzel v jako svého syna

Transplant(T, u, v)

```
1 if u.p = Nil then T.root ← v
2                  else if u = u.p.left then u.p.left ← v
3                                       else u.p.right ← v
4                       fi
5 fi
6 if v ≠ Nil then v.p ← u.p fi
```

BVS - odstranění uzlu

Tree_Delete(T, z)

```
1  if z.left = Nil
2     then TRANSPLANT(T, z, z.right)
3      else if z.right = Nil
4              then TRANSPLANT(T, z, z.left)
5               else y ← TREE_MINIMUM(z.right)
6                    if y.p ≠ z then TRANSPLANT(T, y, y.right)
7                                    y.right ← z.right
8                                    y.right.p ← y
9                    fi
10                   TRANSPLANT(T, z, y)
11                   y.left ← z.left
12                   y.left.p ← y
13             fi
14 fi
```

Složitost

- všechny uvedené operace nad binárním vyhledávacím stromem mají složitost úměrnou hloubce stromu, tj. v nejhorším případě $\mathcal{O}(n)$, kde n je počet uzlů stromu
- při hledání předchůdce a následníka nemusíme vůbec porovnávat klíče
- operace se dají využít k seřazení klíčů např. tak, že najdeme minimální klíč a pak (rekurzivně) jeho následníka (složitost?!)

Vyvážené binární vyhledávací stromy

hloubka stromu je logaritmická

složitost operací je úměrná hloubce stromu

- AVL stromy
- 2 - 3 stromy
- 2 - 3 - 4 stromy
- B stromy
- červeno černé stromy

Modifikace datových struktur

- reálné situace, ve kterých potřebujeme datovou strukturu odlišnou od „*učebnicových struktur*“
- ??? účelnost návrhu úplně nové struktury
- možné řešení:
  - rozšíření některé známé struktury o nové informace
  - návrh nových operací nad takto rozšířenou strukturou
  - . . . při zachování efektivnosti původních operací

příklad: využití binárních vyhledávacích stromů pro reprezentaci množiny intervalů

Reprezentace intervalů

- číselný interval $\langle t_1, t_2 \rangle$
- objekt i s atributy $i.low$ a $i.high$
- intervaly i a i' se překrývají právě když $i.low \leq i'.high$ a současně $i'.low \leq i.high$
- pro libovolné dva intervaly i a i' platí právě jedna z možností
  - intervaly se překrývají
  - interval i je vlevo od i', tj. $i.high < i'.low$
  - interval i' je vlevo od i, tj. $i'.high < i.low$

hledáme datovou strukturu pro reprezentaci množiny intervalů nad kterou je možné efektivně implementovat operace

- INTERVAL_INSERT(T, x) – do množiny intervalů T přidá objekt reprezentující interval x
- INTERVAL_DELETE(T, x) – z množiny intervalů T odstraní objekt reprezentující interval x
- INTERVAL_SEARCH(T, i) – vrátí ukazatel na objekt, který reprezentuje interval překrývající se s intervalem i resp. hodnotu Nil, když takový objekt neexistuje

řešení 1

- seznam intervalů
- přidání intervalu v konstantním čase
- odebrání a vyhledání intervalu v čase $\mathcal{O}(n)$ *(n je počet intervalů v množině)*

řešení 2

- uspořádaný seznam intervalů
- všechny operace v čase $\mathcal{O}(n)$

intervalové stromy

- rozšíření binárních vyhledávacích stromů

Intervalové stromy

- binární vyhledávací strom
- **každý uzel má atributy** $i.low$, $i.high$, **a** $i.max$
- **jako klíč je použita hodnota** $x.low$

- $x.max$ je maximální hodnota krajního bodu intervalu uloženého v podstromu s kořenem x

$$x.max = \max\{x.high,\ x.left.max,\ x.right.max\}$$

Přidání nového intervalu

- postupujeme jako v BVS od kořene, vkládaný uzel se stane listem
- každému uzlu y na cestě z kořene do nového uzlu x aktualizujme hodnotu $y.max$ právě když $y.max < x.high$
- pro žádný vrchol neležící na cestě z kořene do nového uzlu se hodnota max nemění

Odstranění intervalu

- postupujeme jako v BVS
- na pozici odstraněného uzlu se přesune uzel y
- pro aktualizaci hodnot max procházíme cestu od původní pozice uzlu y do kořene a každému uzlu z na této cestě aktualizujeme hodnotu $z.max = \max\{z.left.max, z.right.max, z.high\}$
- složitost operace se navýší o $\mathcal{O}(n)$
- celková složitost operace odstranění intervalu zůstává asymptoticky stejná

Vyhledávání intervalu

Interval_Search(T, i)

```
1 x ← T.root
2 while x ≠ Nil ∧  intervaly i a (x.low, x.high) se nepřekrývají  do
3       if x.left ≠ Nil ∧ x.left.max ≥ i.low
4          then x ← x.left
5           else x ← x.right fi
6 od
7 return x
```

složitost

- vyhledávání začíná v kořeni
- po každé iteraci cyklu testujeme uzel, jehož hloubka je o 1 vyšší
- složitost je úměrná hloubce stromu

Vyhledávání intervalu

Interval_Search(T, i)

```
1 x ← T.root
2 while x ≠ Nil ∧  intervaly i a ⟨x.low, x.high⟩ se nepřekrývají  do
3         if x.left ≠ Nil ∧ x.left.max ≥ i.low
4            then x ← x.left
5             else x ← x.right fi od
6 return x
```

korektnost - případ 1 - ve vyhledávání postupujeme doprava

- předpokládejme, že ve vyhledávání postupujeme z uzlu x doprava a levý podstrom **není** prázdný
- platí $x.left.max < i.low$ *(jinak bychom postupovali doleva)*
- pro každý interval $\langle a, b\rangle$ z levého podstromu platí $b \leq x.left.max$ *(z definice hodnoty max)*
- $b < i.low$ znamená, že i se nepřekrývá se žádným intervalem v levém podstromu

Vyhledávání intervalu

Interval_Search(T, i)

```
1 x ← T.root
2 while x ≠ Nil ∧  intervaly i a ⟨x.low, x.high⟩ se nepřekrývají  do
3       if x.left ≠ Nil ∧ x.left.max ≥ i.low
4          then x ← x.left
5           else x ← x.right fi
6 od
7 return x
```

korektnost - případ 2 - ve vyhledávání postupujeme doleva

- předpokládejme, že žádný interval levého podstromu se nepřekrývá s i
- v levém podstromu leží interval $\langle c, d\rangle$ takový, že $d = x.left.max$
- $i.high < c$ *(i a $\langle c, d\rangle$ se nepřekrývají)*
- pro každý interval $\langle a, b\rangle$ z pravého podstromu platí $c \leq a$ *(vlastnost BVS)*
- $i.high < a$ znamená, že i se nepřekrývá se žádným intervalem v pravém podstromu

Intervalové stromy - modifikace

- vyhledávání **všech** překrývajících se intervalů
- intervaly vyšší dimenze
- namísto obecného binárního vyhledávacího stromu můžeme použít vyvážený binární vyhledávací strom

Radix trees

- využití binárních vyhledávacích stromů pro lexikografické řazení binárních řetězců
- řetězce postupně vkládáme do vyhledávacího stromu
- po vložení všech řetězců strom prohledáme a klíče vypíšeme v pořadí preorder
- časová složitost je $\Theta(n)$, kde n je součet délek všech řetězců

- zobecnění pro řetězce nad libovolnou abecedou - použijeme stromy, jejichž arita je stejná jako velikost abecedy

Datové struktury

Červeno černé stromy

Červeno černý strom je binární vyhledávací strom, jehož každý uzel je obarvený červenou anebo černou barvou a splňuje podmínky

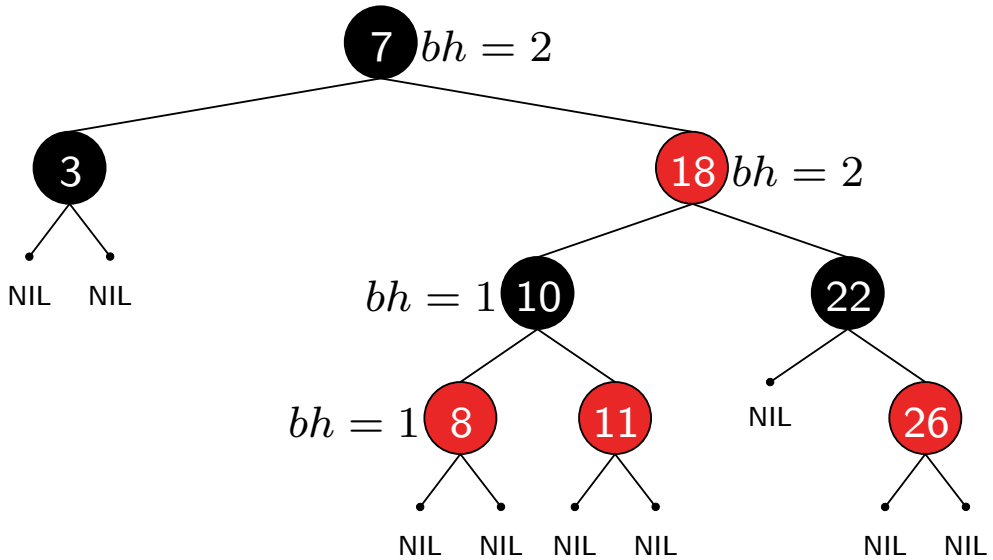
1. kořen stromu je černý
2. listy stromu nenesou žádnou hodnotu, tj. jsou označeny *Nil*, a mají černou barvu
3. když je uzel červený, tak jeho otec je černý
4. pro každý uzel x stromu platí, že všechny cesty z uzlu x do listů obsahují stejný počet černých uzlů

alternativně: oba synové červeného uzlu mají černou barvu

- každý uzel obsahuje atributy *key, color, left, right, p*
- jestliže uzel nemá některého syna anebo otce, tak příslušný atribut má hodnotu Nil

Výška uzlu a černá výška uzlu

- **výška** uzlu x je rovna počtu hran na nejdelší cestě z x do listu
- **černá výška** uzlu x, $bh(x)$, je rovna počtu **černých** uzlů na cestě z x do listu (uzel x nezapočítáváme)
  (díky vlastnosti 4 je černá výška dobře definovaná!)



Výška červeno černého stromu

Lema 3

Každý uzel s výškou h má černou výšku alespoň $h/2$.

z vlastnosti 4 plyne, že v nejhorším případě je každý druhý uzel na cestě červený

Lema 4

Pro každý uzel x platí, že podstrom s kořenem x má alespoň $2^{bh(x)} - 1$ vnitřních uzlů[4].

důkaz indukcí k výšce h uzlu x

$h = 0$ x je list $\Longrightarrow bh(x) = 0$ a současně počet vnitřních uzlů podstromu s kořenem x je 0

$h > 0$
- nechť x má výšku h a černou výšku $bh(x) = b$
- každý syn uzlu x má výšku $h - 1$ a černou výšku b anebo $b - 1$
- z indukčního předpokladu má podstrom každého syna alespoň $2^{bh(x)-1} - 1$ vnitřních uzlů
- podstrom s kořenem x má alespoň $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ vnitřních uzlů

[4] vnitřním uzlem rozumíme uzel, který nese hodnotu, tj. list není vnitřním uzlem

Výška červeno černého stromu

Věta 5

Červeno černý strom s n vnitřními uzly má výšku nejvýše $2\log_2(n+1)$.

- nechť strom má výšku h a černou výšku b
- z předchozích lemmat plyne

$$n \geq 2^b - 1 \geq 2^{h/2} - 1$$

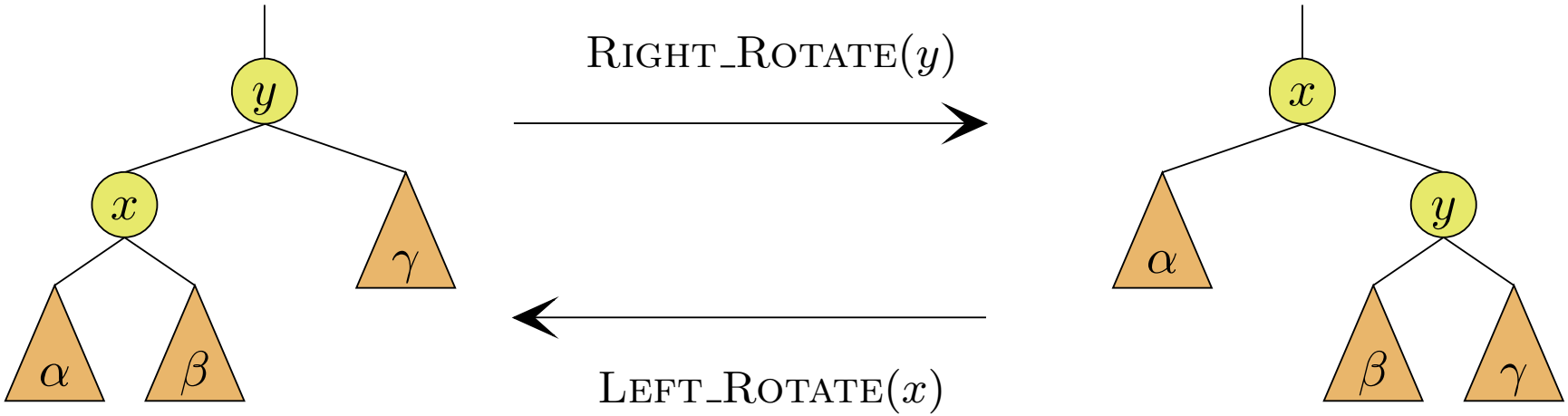
- po úpravě $\log_2(n+1) \geq h/2$, a tedy $h \leq 2\log_2(n+1)$

Červeno černé stromy - operace

- Search, Min, Max, Successor, Predecessor se implementují stejně jako pro binární vyhledávací stromy
- vyjmenované operace mají složitost $\mathcal{O}(\log n)$

- Insert a Delete modifikují strom
- modifikace může porušit vlastnosti červeno černého stromu
- jsou potřebné další kroky, které vlastnosti obnoví
- základní operací, která vede k obnovení požadovaných vlastností, je rotace

Rotace



- rotace zachovává vlastnost binárního vyhledávacího stromu

$$a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq x \leq b \leq y \leq c$$

- časová složitost $\mathcal{O}(1)$

Rotace

Left_Rotate(T, x)

```
 1 y ← x.right
 2 x.right ← y.left
 3 if y.left ≠ Nil
 4    then y.left.p ← x fi
 5 y.p ← x.p
 6 if x.p = Nil
 7    then T.root ← y
 8     else if x = x.p.left
 9             then x.p.left ← y
10              else x.p.right ← y fi fi
11 y.left ← x
12 x.p ← y
```

Přidání nového uzlu

- uzel x do stromu přidáme stejným postupem jako do binárního vyhledávací stromu
- jakou barvou máme obarvit nový uzel?
- obě možnosti mají za důsledek porušení některých vlastností červeno černého stromu

- řešení: obarvi uzel x **červenou** barvou
- vlastnosti
  1. *(černý kořen)* jestliže x je kořenem, tak vlastnost neplatí
  3. *(otec červeného uzlu je černý)* nemusí platit
  4. *(stejná černá výška)* zůstává v platnosti

Přidání nového uzlu - schéma

RB_Insert(T, a)

```
1  TREE_INSERT(T, a)
2  a.color ← red
3  while a ≠ T.root ∧ a.p.color = red
4        do if a.p = a.p.p.left
5              then d ← a.p.p.right
6                   if d.color = red
7                     then  případ 1
8                      else if a = a.p.right
9                                then  případ 2
10                                else  případ 3
11                           fi
12                  fi
13             else  stejně jako THEN se záměnou left a right
14       fi
15 od
16 T.root.color ← black
```

Přidání nového uzlu - schéma

případ 1

$a.p.color \leftarrow black$
$d.color \leftarrow black$
$a.p.p.color \leftarrow red$
$a \leftarrow a.p.p$

případ 2

$a \leftarrow a.p$
$\textsc{Left\_Rotate}(T, a)$

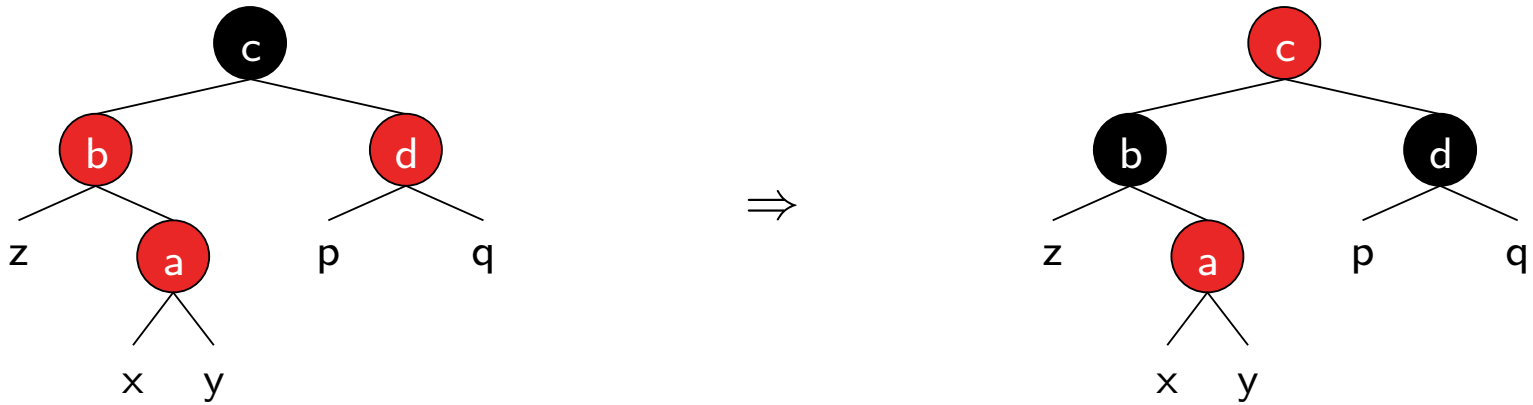
případ 3

$a.p.color \leftarrow black$
$a.p.p.color \leftarrow red$
$\textsc{Right\_Rotate}(T, a.p.p)$

Přidání nového uzlu - korekce - případ 1

- nově přidaný uzel a je červený
- jeho otec b je červený a je levým synem svého otce[5]
- strýc d uzlu a je červený
- praotec c uzlu a je **černý**

- obarvi otce (b) a strýce (d) uzlu a **černou** barvou
- obarvi praotce (c) uzlu a červenou barvou



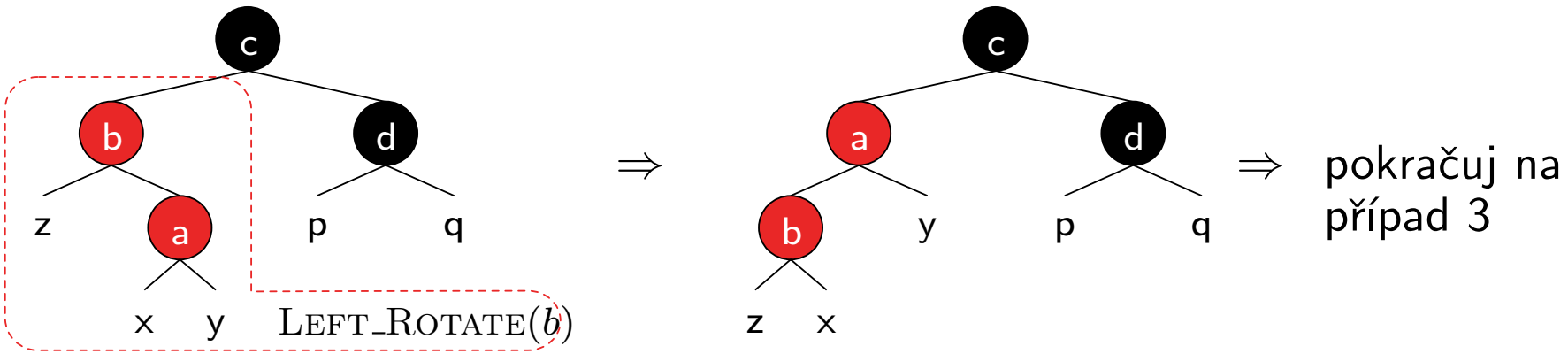
~~stromy z, x, y, p, q mají černý kořen~~ a všechny mají stejnou černou výšku

[5] situace když b je pravým synem svého otce se řeší symetricky

Přidání nového uzlu - korekce - případ 2

- uzel a je červený a je pravým synem svého otce
- jeho otec b je červený a je levým synem svého otce
- strýc d uzlu a je **černý**
- praotec c uzlu a je **černý**

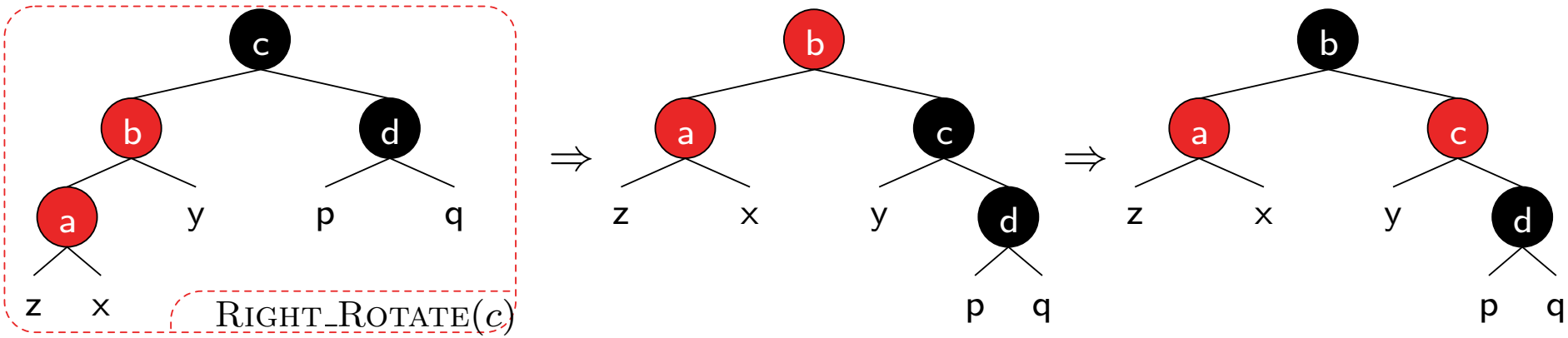
- proveď levou rotaci kolem otce (b) uzlu a
- pokračuj na případ 3



Přidání nového uzlu - korekce - případ 3

- uzel a je červený a je levým synem svého otce
- jeho otec b je červený a je levým synem svého otce
- strýc d uzlu a je **černý**
- praotec c uzlu a je **černý**

- proveď pravou rotaci kolem praotce (c) uzlu a
- vyměň obarvení mezi otcem (b) uzlu a a jeho novým bratrem (c)



Složitost přidání nového uzlu

- případ 1: změna obarvení 3 uzlů
- případy 2 a 3: jedna nebo dvě rotace a změna obarvení 2 uzlů

- v případě 1 může změna barvy praotce (c) uzlu a způsobit nový konflikt a to když otec uzlu c má červenou barvou
- v popsaném případě musíme pokračovat další iterací a korigovat barvu uzlu c
- konečnost je garantována faktem, že každou iterací se zmenšuje vzdálenost korigovaného uzlu od kořene stromu
- celková složitost $\mathcal{O}(\log n)$

Odstranění uzlu

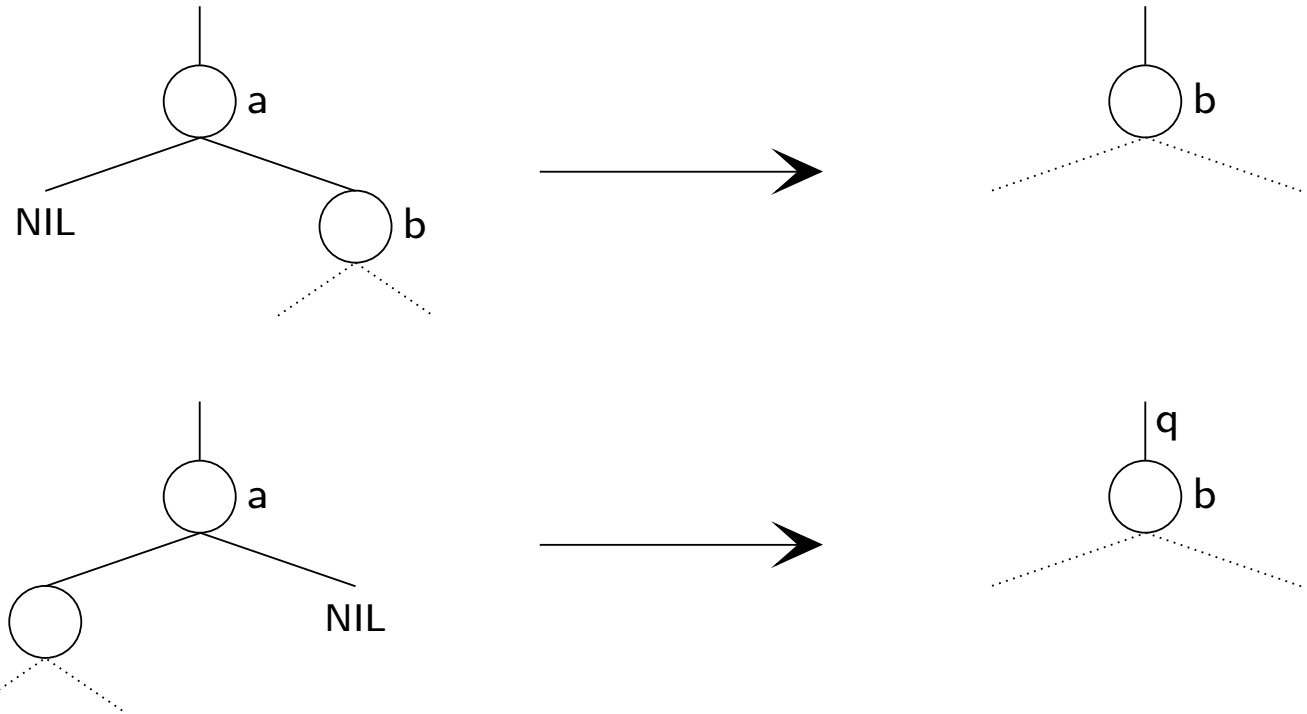
- uzel x ze stromu odstraníme stejným postupem jako z binárního vyhledávací stromu
- v případě, že odstraněný uzel měl červenou barvu, vlastnosti stromu zůstávají zachované
- v případě, že měl černou barvu, může dojít k porušení vlastnosti 4 (stejná černá výška)
- černou barvu z odstraněného uzlu přesouváme směrem ke kořenu tak, abychom obnovili platnost vlastnosti 4

Odstranění uzlu a - případy 1 a 2

a nemá levého syna

- odstraň a a nahraď ho jeho pravým synem (b)
- jestliže po přesunu uzel b a jeho otec porušují vlastnost 3 (oba jsou červené), tak uzel b obarvíme černou barvou; tím zachováme černou výšku (a musel být černý)

a nemá pravého syna - symetricky

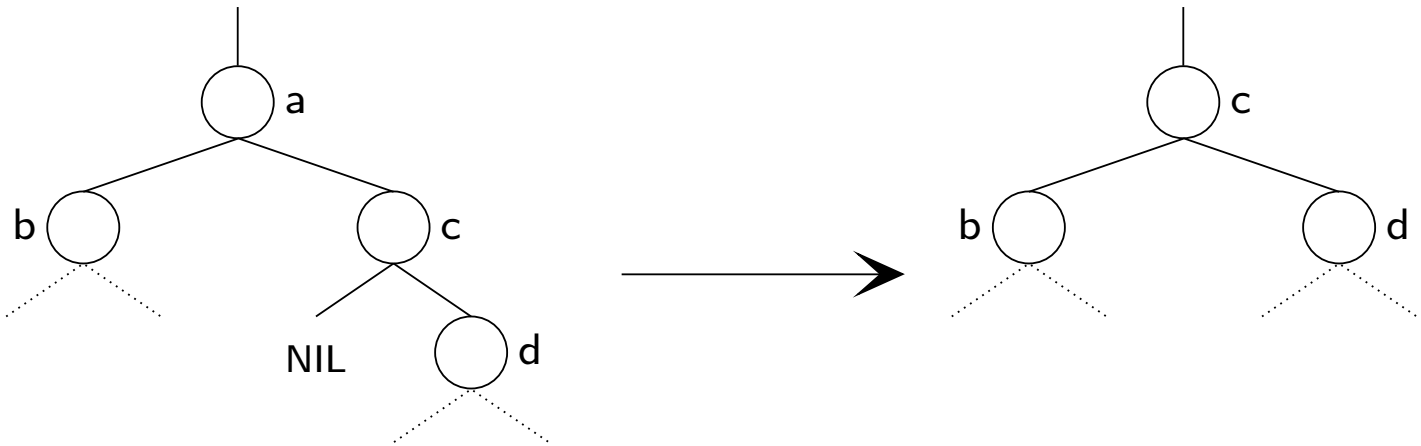


Odstranění uzlu a - případ 3

a má dva syny, následník (*successor*) **uzlu a je jeho pravým synem**

- odstraň a a nahraď ho jeho následníkem (c)
- levý syn uzlu a se stane levým synem následníka uzlu a

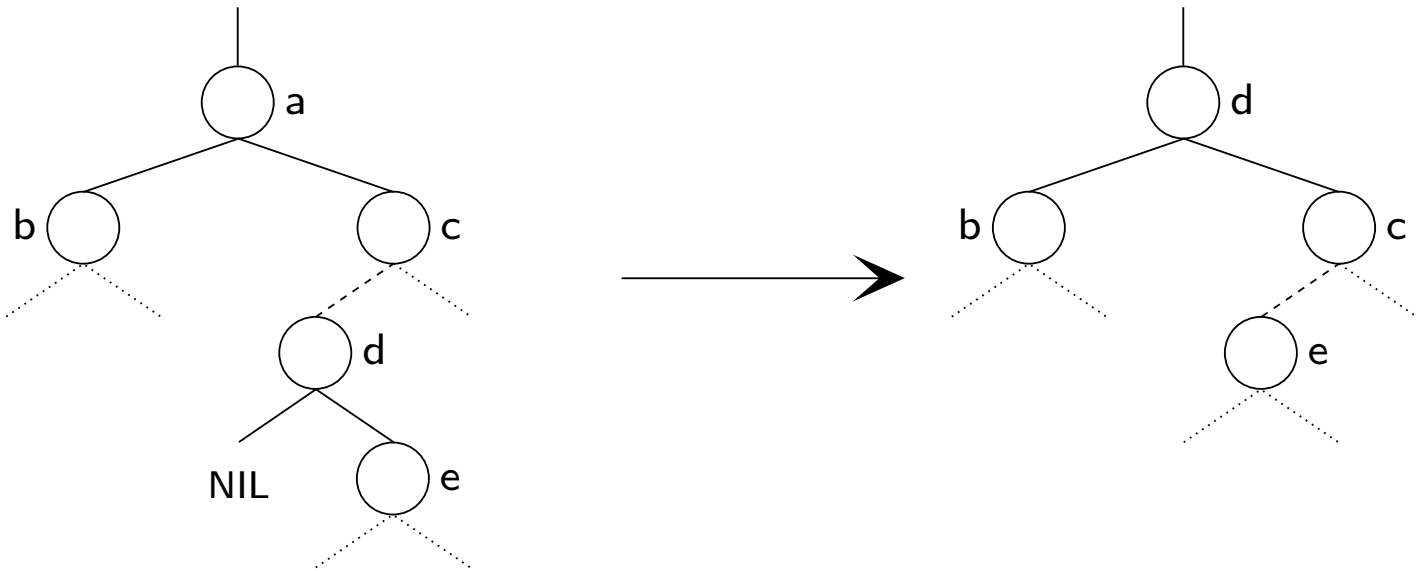
- po přesunu obarvíme následníka (c) barvou uzlu a
- jestliže následník měl původně černou barvu, tak černou barvu dostane jeho syn, tj. syn má dvě barvy (červenou a černou anebo černou a černou)
- problém dvou barev vyřešíme při korekci



Odstranění uzlu a - případ 4

a má dva syny, následník (*successor*) **uzlu a není jeho synem**

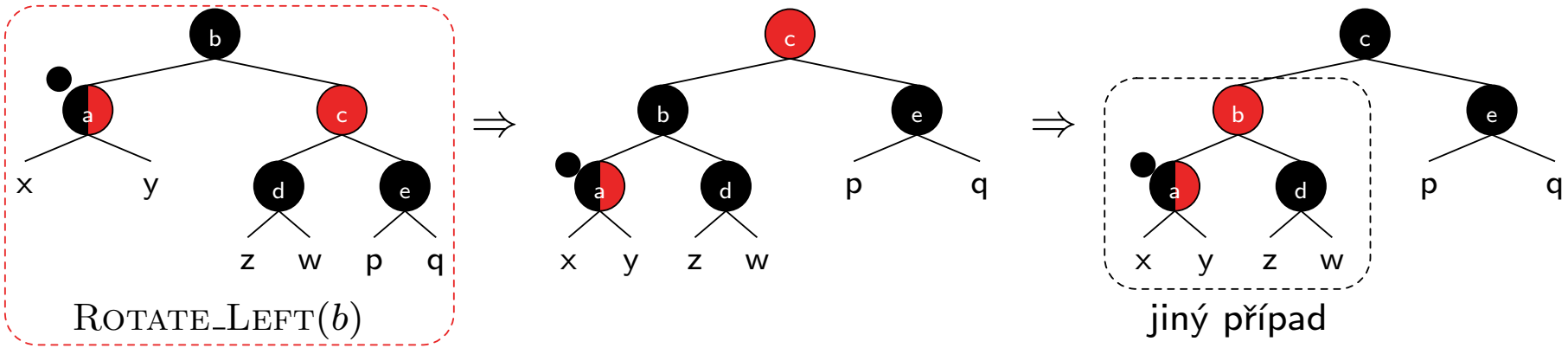
- následníka (d) nahraď jeho pravým synem (e)
- odstraň a a nahraď ho jeho následníkem (d), synové uzlu a se stanou syny následníka (d)
- po přesunu obarvíme následníka (d) barvou uzlu a
- jestliže následník měl původně černou barvu, tak černou barvu dostane jeho syn, tj. syn má dvě barvy (červenou a černou anebo černou a černou)
- problém dvou barev vyřešíme při korekci



Odstranění uzlu - korekce dvou barev - případ 1

uzel a má dvě barvy
bratr (c) uzlu a je červený

- proveď levou rotaci kolem otce (b) uzlu a
- vyměň barvy mezi otcem (b) a praotcem (c) uzlu a
- pokračuj některým z následujících případů

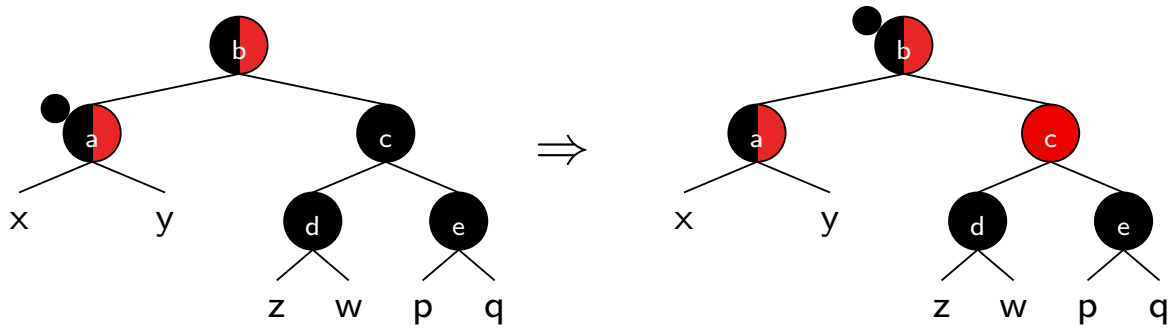


stromy x, y, z, w, p, q mají stejnou černou výšku, nemají žádný uzel s dvěma barvami a neporušují žádnou vlastnost červeno černého stromu

Odstranění uzlu - korekce dvou barev - případ 2

uzel a má dvě barvy
bratr (c) uzlu a stejně jako oba jeho synové (d, e) mají černou barvu

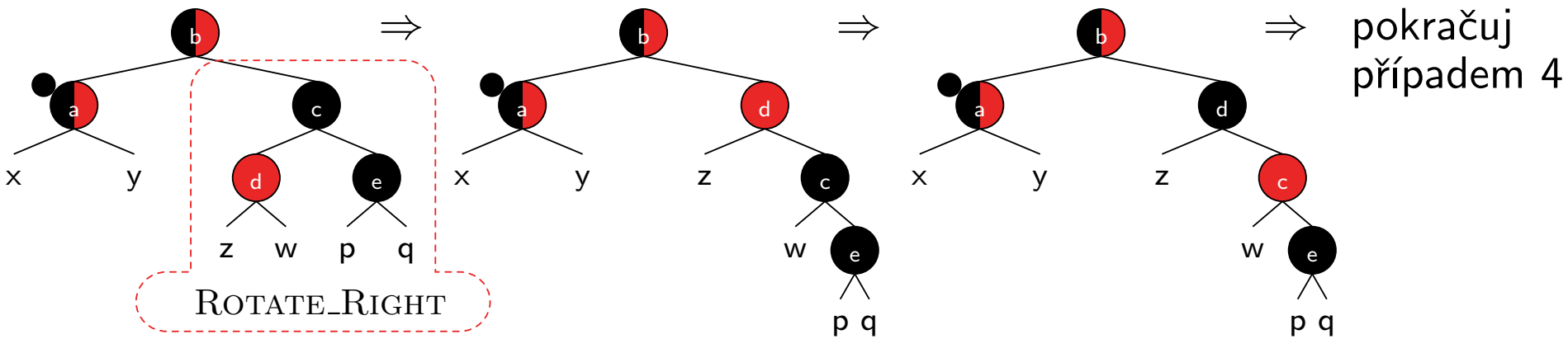
- vezmi jednu černou barvu z uzlu a a přesuň ji do jeho otce (b)
- bratr (c) uzlu a dostane červenou barvu *(aby se zachovala černá výška)*
- uzel se dvěma barvami se přesunul blíž ke kořenu, problém jeho dvou barev řešíme rekurzivně



Odstranění uzlu - korekce dvou barev - případ 3

uzel a má dvě barvy
bratr (c) uzlu a a jeho pravý syn (e) mají černou barvu, levý syn (d) je červený

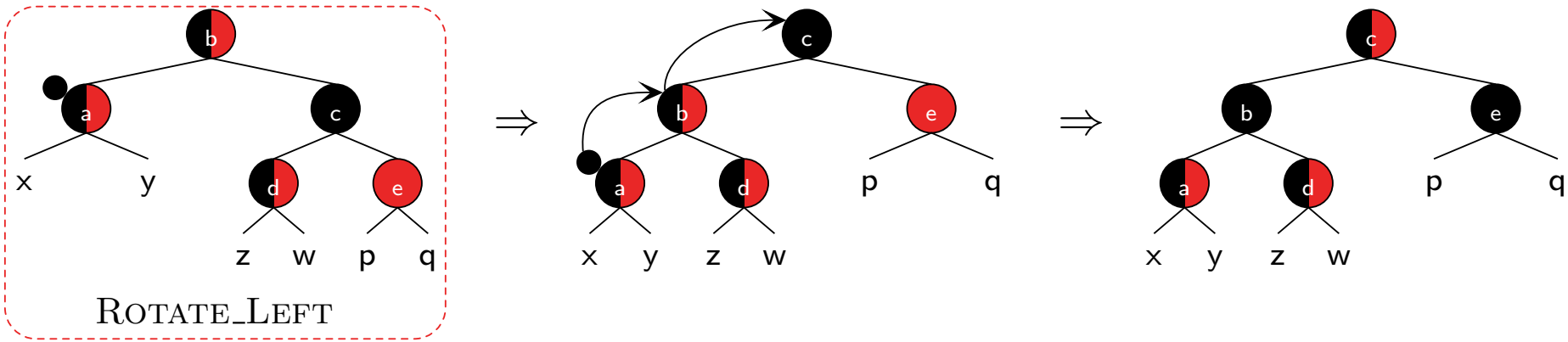
- proveď pravou rotaci kolem bratra (c) uzlu a
- vyměň barvy mezi původním a novým bratrem uzlu a (d, c)
- pokračuj případem 4



Odstranění uzlu - korekce dvou barev - případ 4

uzel a má dvě barvy
bratr (c) uzlu a má černou barvu, jeho pravý syn (e) má červenou barvu

- proveď levou rotaci kolem otce (b) uzlu a
- obarvi nového praotce (c) uzlu a barvou jeho otce (b)
- přesuň černou barvu z uzlu a na jeho otce (b), otec (b) se stane černým
- uzel (e) se stane černým



Pořadí (rank) prvku

využití červeno černých stromů při určení ranku (pořadí) prvku a vyhledávání prvku s daným rankem

- množina A obsahující n vzájemně různých čísel
- číslo $x \in A$ má rank i právě když v A existuje přesně $i - 1$ čísel menších než x

možné řešení

- jestliže prvky A jsou uložené v poli, tak v čase $\mathcal{O}(n)$ můžeme
  - najít číslo s rankem i
  - určit rank daného čísla

existuje efektivnější řešení?

při použití červeno černých stromů dokážeme oba problémy vyřešit v čase $\mathcal{O}(\log n)$

Rozšíření červeno černých stromů

požadujeme

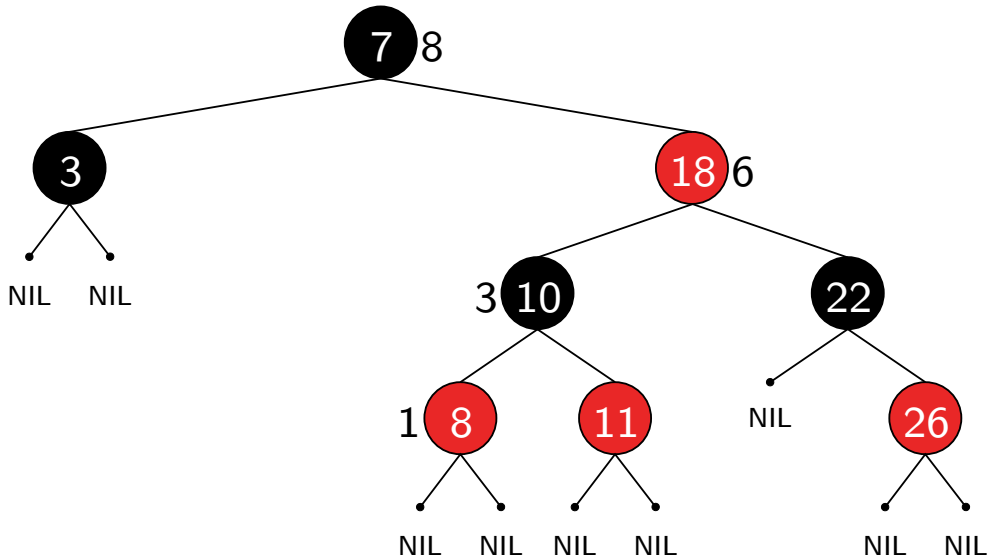
- efektivní implementaci standardních operací nad červeno černým stromem
- efektivní implementaci operace $\mathrm{RB\_SELECT}(x, i)$, která najde i-ty nejmenší klíč v podstromě s kořenem x
- efektivní implementaci operace $\mathrm{RB\_RANK}(T.x)$, která určí rank klíče uloženého v uzlu x

jestliže strom obsahuje uzly se stejnými klíči, tak rankem klíče je pořadí uzlu v INORDER *uspořádání uzlů stromu*

Princip

ke každému uzlu x přidáme atribut $x.size$ - počet (vnitřních) uzlů v podstromě s kořenem x, včetně uzlu x

$$x.size = x.left.size + x.right.size + 1$$



Vyhledání klíče s daným rankem

RB_Select(x, i)

```
1 r ← x.left.size + 1
2 if i = r then return x
3         else if i < r then return RB_Select(x.left, i)
4                        else return RB_Select(x.right, i − r) fi fi
```

korektnost

- z definice atributu $.size$ plyne, že počet uzlů v levém podstromu uzlu x navýšený o 1 (r) je přesně rank klíče uloženého v x v podstromě s kořenem x
- když $i = r$, tak x je hledaný uzel
- když $i < r$, tak i-ty nejmenší klíč se nachází v levém podstromě uzlu x a je i-tým nejmenším klíčem v tomto podstromě
- když $i > r$, tak i-ty nejmenší klíč se nachází v pravém podstromě uzlu x a jeho pořadí v tomto podstromě je i snížené o počet uzlů levého podstromu

Vyhledání klíče s daným rankem

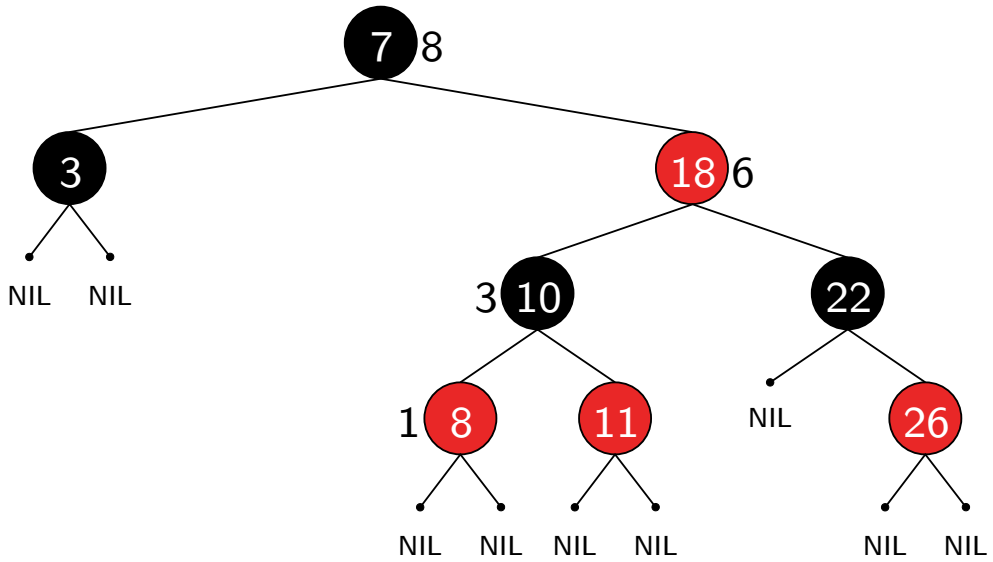
RB_Select(x, i)

```
1 r ← x.left.size + 1
2 if i = r then return x
3         else if i < r then return RB_Select(x.left, i)
4                        else return RB_Select(x.right, i − r) fi fi
```

složitost

- každé rekurzivní volání se aplikuje na strom, jehož hloubka je o 1 menší
- hloubka červeno černého stromu je $\mathcal{O}(\log n)$
- složitost RB_Select je $\mathcal{O}(\log n)$

Určení ranku daného prvku



rank prvku 11

- všechny uzly v levém podstromě uzlu 11
- sledujeme cestu od 11 do kořene
- jestliže uzel na cestě je levým synem, nemění rank prvku 11
- jestliže uzel na cestě je pravým synem, tak on sám jakož i jeho levý podstrom obsahují klíče menší než 11

Určení ranku daného prvku

RB_Rank(T, x)

```
1 r ← x.left.size + 1
2 y ← x
3 while y ≠ T.root
4       do if y = y.p.right
5             then r ← r + y.p.left.size + 1 fi
6       y ← y.p od
7 return r
```

korektnost

invariant na začátku každé iterace **while** cyklu je r rovné ranku klíče $x.key$ v podstromě s kořenem y

inicializace na začátku je r rovné ranku $x.key$ v podstromě s kořenem x a $x = y$

RB_Rank(T, x)

```
1 r ← x.left.size + 1
2 y ← x
3 while y ≠ T.root
4       do if y = y.p.right
5             then r ← r + y.p.left.size + 1 fi
6       y ← y.p od
7 return r
```

invariant na začátku každé iterace **while** cyklu je r rovné ranku klíče $x.key$ v podstromě s kořenem y

iterace

- na konci cyklu se vykoná $y \leftarrow y.p$
- po provedení cyklu proto musí platit, že r je rank $x.key$ v podstromě s kořenem $y.p$
- jestliže y je levý syn, tak všechny klíče v podstromě jeho bratra jsou větší než $x.key$ a r se nemění
- jestliže y je pravý syn, tak všechny hodnoty v podstromě jeho bratra jsou menší než $x.key$ a hodnota r se zvýší o velikost tohoto stromu plus 1 (klíč v uzlu $y.p$ je taky menší než $x.key$)

RB_Rank(T, x)

```
1 r ← x.left.size + 1
2 y ← x
3 while y ≠ T.root
4         do if y = y.p.right
5                 then r ← r + y.p.left.size + 1 fi
6         y ← y.p od
7 return r
```

invariant na začátku každé iterace **while** cyklu je r rovné ranku klíče $x.key$ v podstromě s kořenem y

ukončení

výpočet končí když $y = T.root$, z platnosti invariantu plyne korektnost algoritmu

složitost

- po každé iteraci se sníží vzdálenost y od kořene o 1
- hloubka červeno černého stromu je $\mathcal{O}(\log n)$
- složitost RB_Rank je $\mathcal{O}(\log n)$

Přidání nového uzlu

přidání uzlu postupujeme od kořene do listu, kde vytvoříme nový uzel, přitom se změní (o 1) pouze velikost podstromů těch uzlů, kterými procházíme

korekce stromu změna barvy uzlu nemění velikost podstromu

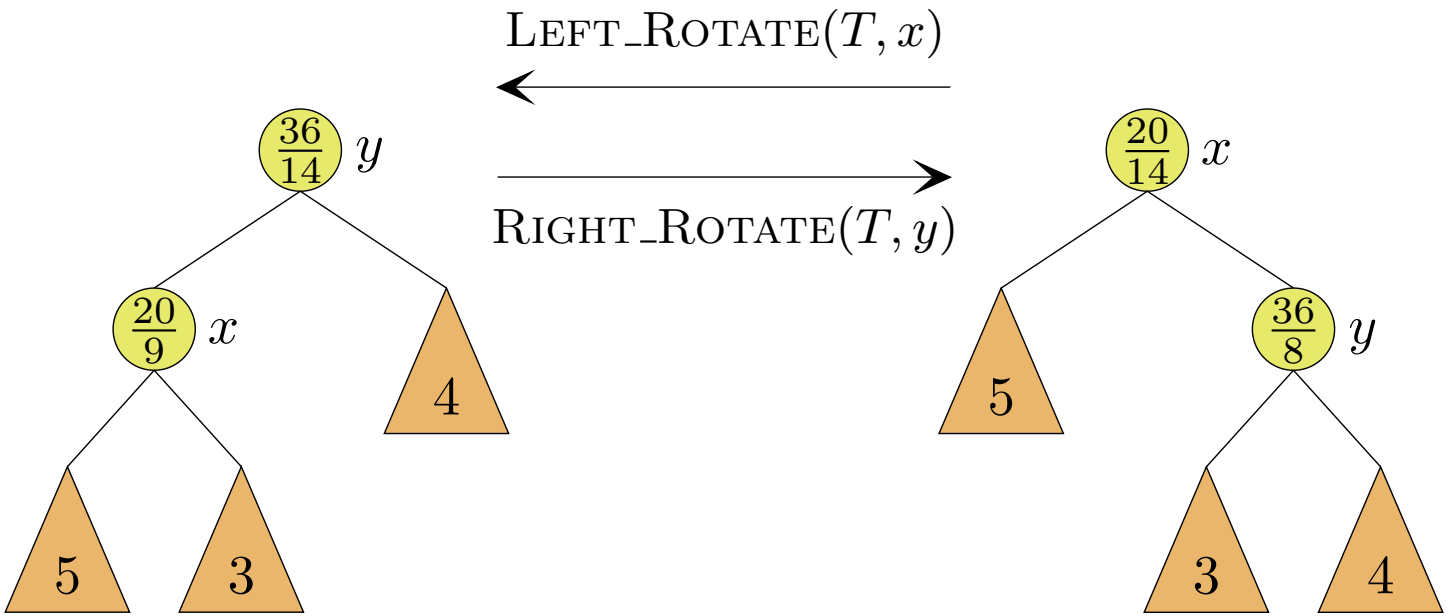
při rotaci se může změnit velikost podstromů

proceduru LEFT_ROTATE doplníme o příkazy

$y.size \leftarrow x.size$

$x.size \leftarrow x.left.size + x.right.size + 1$

symetricky pro pravou rotaci



Odstranění uzlu

první fáze odstraní uzel ze stromu

- na pozici odstraněného uzlu se přesune uzel y
- pro aktualizaci hodnot $size$ procházíme cestu od původní pozice uzlu y do kořene a každému uzlu na této cestě snížíme hodnotu $size$ o 1
- složitost operace se navýší o $\mathcal{O}(\log n)$

korekce obarvení stromu

- ke změně velikosti podstromu může dojít při rotaci, aktualizace hodnot viz přidání nového uzlu
- počet rotací je nejvýše 3, složitost se navýší o $\mathcal{O}(1)$

složitost přidávání i odstraňování uzlu zůstává asymptoticky stejná

Datové struktury

B stromy

B stromy jsou zobecněním binárních vyhledávacích stromů

- B strom je balancovaný, všechny listy mají stejnou hloubku
- vnitřní uzel stromu obsahuje $t - 1$ klíčů a má t následníků
- klíče ve vnitřních uzlech stromu zároveň vymezují t intervalů, do kterých patří klíče každého z jeho t podstromů

využití B stromů

- B stromy se typicky používají v databázových systémech a aplikacích, kde objem zpracovávaných dat není možné uchovávat v operační paměti
- počet klíčů uložených v uzlu (a tím i počet následníků) se může pohybovat od jednotek po tisíce; cílem je minimalizovat počet přístupů na disk
- v pseudokódu modelujeme přístupy operacemi DISK_READ a DISK_WRITE
- existují různé varianty, podrobněji viz např. PV062
- Bayer, McCreight 1972

B stromy vs BVS a červeno černé stromy

- zachován princip vyhledávání
- všechny uzly mají stejnou hloubku
- uzly B stromů můžou mít víc následníků
- výška B stromu je $\mathcal{O}(\log n)$, díky většímu počtu následníků může být ale výrazně menší
- operace minimalizují průchod stromem

Stupeň B stromu

minimální stupeň stromu

číslo t, které definuje dolní a horní hranici na počet klíčů uložených v uzlu

- každý uzel (s výjimkou kořene) musí obsahovat alespoň $t - 1$ klíčů
- jestliže strom je neprázdný, tak kořen musí obsahovat alespoň jeden klíč
- každý vnitřní uzel (s výjimkou kořene) musí mít alespoň t následníků

- každý uzel může obsahovat nejvýše $2t - 1$ klíčů
- každý vnitřní uzel může mít nejvýše $2t$ následníků

- uzel, který má přesně $2t - 1$ klíčů, se nazývá plný

- nejjednodušší B strom má minimální stupeň 2
- každý jeho vnitřní uzel má 2, 3 anebo 4 následníky
- obvykle se označuje jako 2-3-4 strom

Výška B stromu

všechny listy mají stejnou hloubku

B strom s $n \geq 1$ klíči a minimálním stupněm $t \geq 2$ má hloubku nejvýše

$$h \leq \log_t \frac{n+1}{2}$$

- kořen obsahuje alespoň jeden klíč, každý vnitřní uzel alespoň $t-1$ klíčů
- strom má 1 uzel hloubky 0 (kořen), alespoň 2 uzly hloubky 1, alespoň $2t$ uzlů hloubky 2, alespoň $2t^2$ uzlů hloubky 3, obecně alespoň $2t^{h-1}$ uzlů hloubky h

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\sum_{i=0}^{h-1} t^i \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1 \end{aligned}$$

- z toho $t^h \leq \frac{n+1}{2}$ a tedy $\log_t t^h \leq \log_t \frac{n+1}{2}$ □

Klíče v B stromu

- každý **uzel** x má atributy
  - $x.n$ - počet klíčů uložených v uzlu x
  - klíče $x.key_1, x.key_2, \cdots, x.key_{x.n}$, které jsou uloženy v neklesajícím pořadí
  - $x.leaf$ - booleovská proměnná nabývající hodnotu je *true* právě když uzel x je listem stromu

- každý **vnitřní uzel** x obsahuje navíc $x.n + 1$ ukazatelů $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$

- klíče $x.key_i$ definují intervaly, z kterých jsou klíče uložené v každém z podstromů; jestliže k_i je klíč uložený v podstromě s kořenem $x.c_i$, tak platí

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

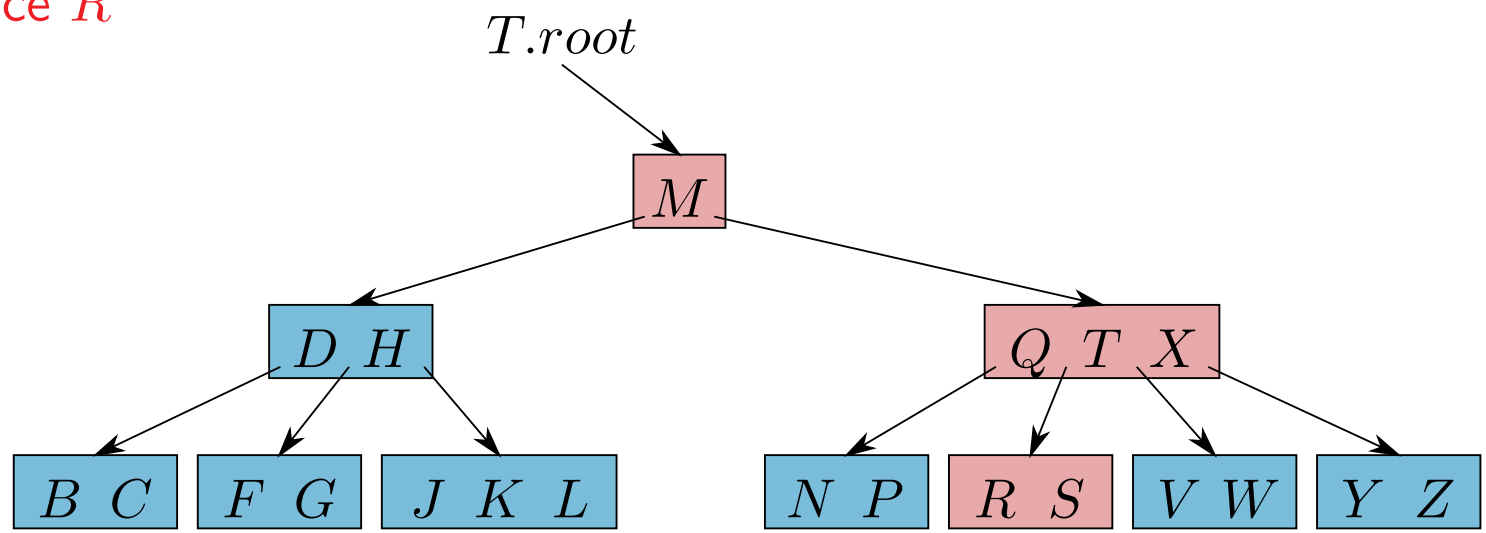
Operace nad B stromem

- vytvoření stromu; vyhledávání, přidání a odstranění klíče
- typické aplikace, které využívají B stromy, pracují s daty uloženými na externím disku
- před každou operací, která přistupuje k objektu x, se nejdříve musí vykonat operace $\textsc{Disk\_Read}(x)$, která zkopíruje objekt do operační paměti (za předpokladu, že tam není)
- symetricky operace $\textsc{Disk\_Write}(x)$ se použije pro uložení všech změn vykonaných nad objektem x
- předpokládáme, že kořen B stromu je vždy uložený v operační paměti a proto nad kořenem vykonáváme pouze operaci $\textsc{Disk\_Write}$
- asymptotická složitost všech operací je úměrná hloubce stromu, tj. $\mathcal{O}(\log n)$, kde n je počet klíčů uložených v stromu
- z důvodu optimalizace počtu přístupů na externí disk jsou všechny operace navrženy tak, aby se uzel stromu navštívil nejvýše jednou, tj. všechny operace postupují směrem od kořene dolů a nikdy se nevracejí do již navštíveného uzlu

Vyhledávání

- analogicky jako v binárním vyhledávacím stromě, vybíráme jednoho z následníků uzlu
- argumentem operace je ukazatel $T.root$ na kořen stromu a hledaný klíč k
- jestliže klíč k je v B stromě, operace vrátí dvojici (y, i), kde y je uzel a i index takový, že $y.key_i = k$
- v opačném případě vrátí hodnotu Nil

vyhledání klíče R



Vyhledávání

B-Tree_Search(x, k)

```
1 i ← 1
2 while i ≤ x.n ∧ x.key_i < k do
3         i ← i + 1 od
4 if i ≤ x.n ∧ x.key_i = k
5    then return (x, i) fi
6 if x.leaf then return Nil
7              else DISK_READ(x.c_i)
8                   return B-TREE_SEARCH(x.c_i, k) fi
```

- počet DISK_READ operací je ohraničený hloubkou stromu h
- počet opakování cyklu 2 - 3 je nejvýše $2t$ (t je minimální stupeň B stromu)
- celková složitost je $\mathcal{O}(th) = \mathcal{O}(t \log_t n)$

Vytvoření prázdného stromu

B-Tree_Create(T)

```
1 x ← Allocate_Node()
2 x.leaf ← true
3 x.n ← 0
4 Disk_Write(x)
5 T.root ← x
```

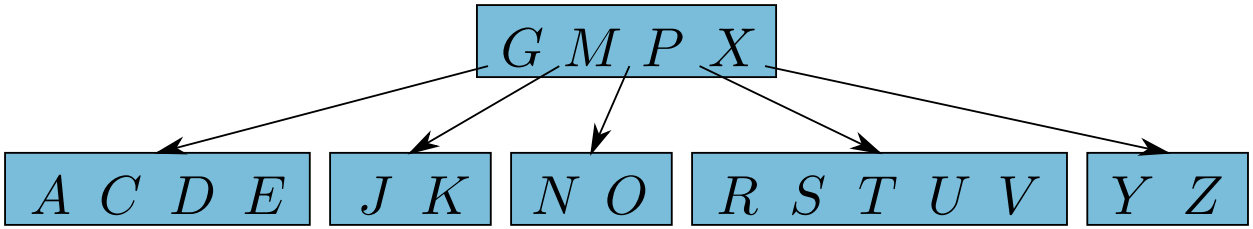
celková složitost $\mathcal{O}(1)$

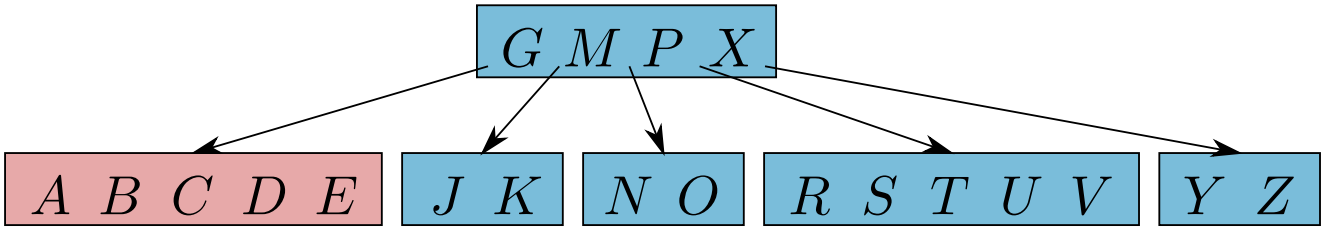
Přidání klíče

- podobně jako u BVS hledáme list, do kterého uložíme nový klíč
- nemůžeme vytvořit nový list (jako v BVS), protože bychom porušili vlastnost minimálního počtu klíčů v uzlu
- klíč vložíme do existujícího listu
- když vložením klíče dojde k porušení vlastnosti maximálního počtu klíčů, tak list rozdělíme na dva nové listy
- rozdělením se zvýší počet následníků předchůdce původního listu
- pokud se tím poruší vlastnost maximálního počtu následníků, tak musíme (rekurzivně) rozdělit i předchůdce
- proces rozdělování uzlů se v nejhorším případě zastaví až v kořeni stromu

Přidání klíče B do listu, který není plný

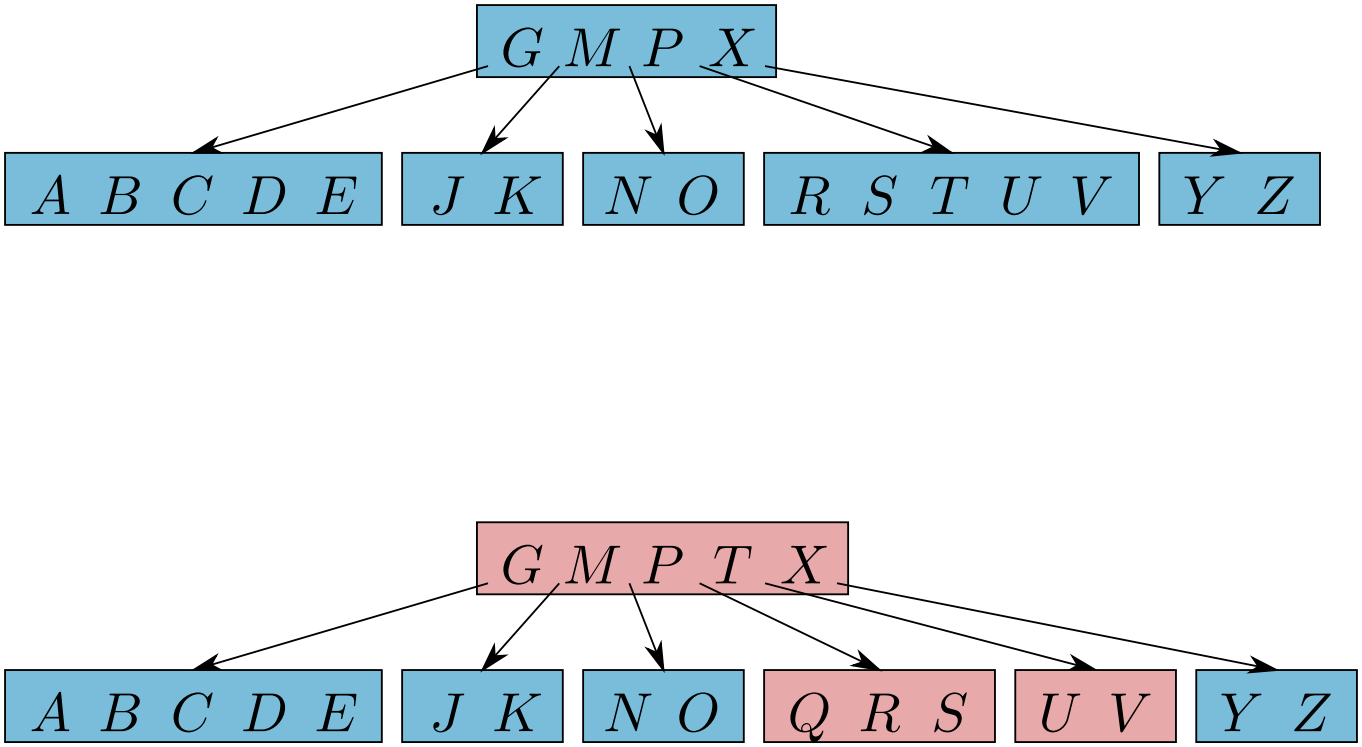
minimální stupeň stromu je 3





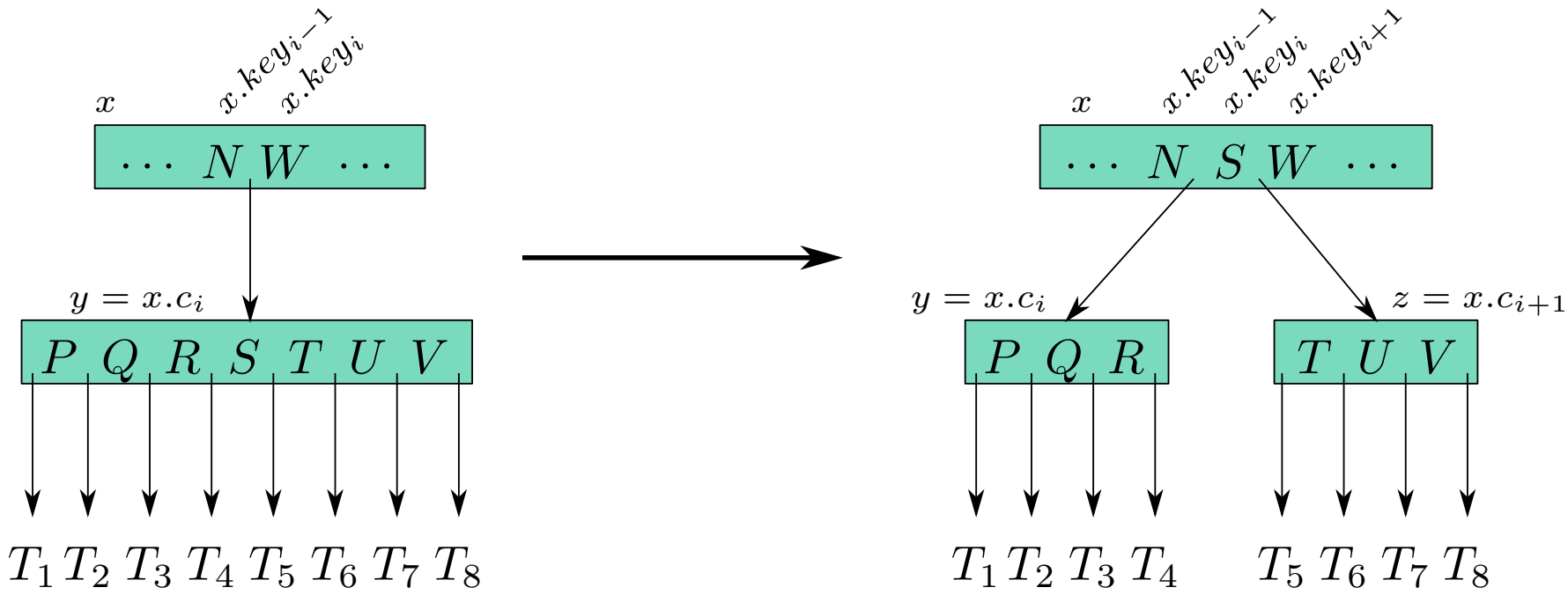
Přidání klíče Q do plného listu

minimální stupeň stromu je 3



Rozdělení uzlu - schéma

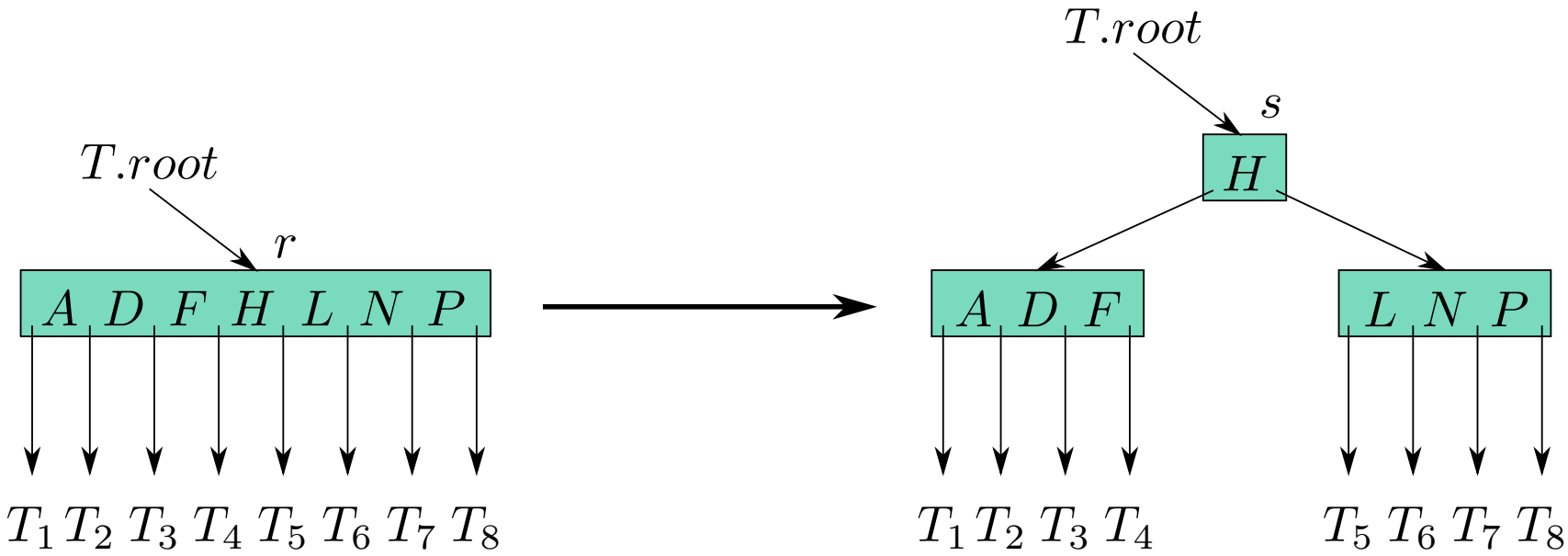
minimální stupeň stromu je 4



argumentem operace B-Tree_Split je

- vnitřní uzel x, který není plný
- index i takový, že $x.c_i$ je plný následník uzlu x

Rozdělení kořene - schéma



- když potřebujeme rozdělit kořen stromu, tak nejdříve vytvoříme nový, prázdný uzel, který se stane novým kořenem stromu
- rozdělení kořene způsobí navýšení hloubky stromu o 1

Rozdělení uzlu - implementace

B-Tree_Split(x, i)

```
1  z ← Allocate_Node()
2  y ← x.c_i
3  z.leaf ← y.leaf
4  z.n ← t − 1
5  for j = 1 to t − 1 do z.key_j ← y.key_{j+t} od
6  if ¬y.leaf then for j = 1 to t do z.c_j ← y.c_{j+t} od fi
7  y.n ← t − 1
8  for j = x.n + 1 downto i + 1 do x.c_{j+1} ← x.c_j od
9  x.c_{i+1} ← z
10 for j = x.n downto i do x.key_{j+1} ← x.key_i od
11 x.key_i ← y.key_t
12 x.n ← x.n + 1
13 Disk_Write(y)
14 Disk_Write(z)
15 Disk_Write(x)
```

Rozdělení uzlu - složitost

- rozdělujeme uzel y (řádek 2)
- když y není list, tak má před rozdělením $2t$ následníků a po rozdělení počet jeho následníků klesne na t
- z je nový uzel (řádek 1) a jeho následníky tvoří t největších následníků uzlu y
- celková složitost je $\mathcal{O}(t)$
- počet operací DISK_WRITE a DISK_READ je $\mathcal{O}(1)$

Přidání klíče - optimalizace

základní varianta

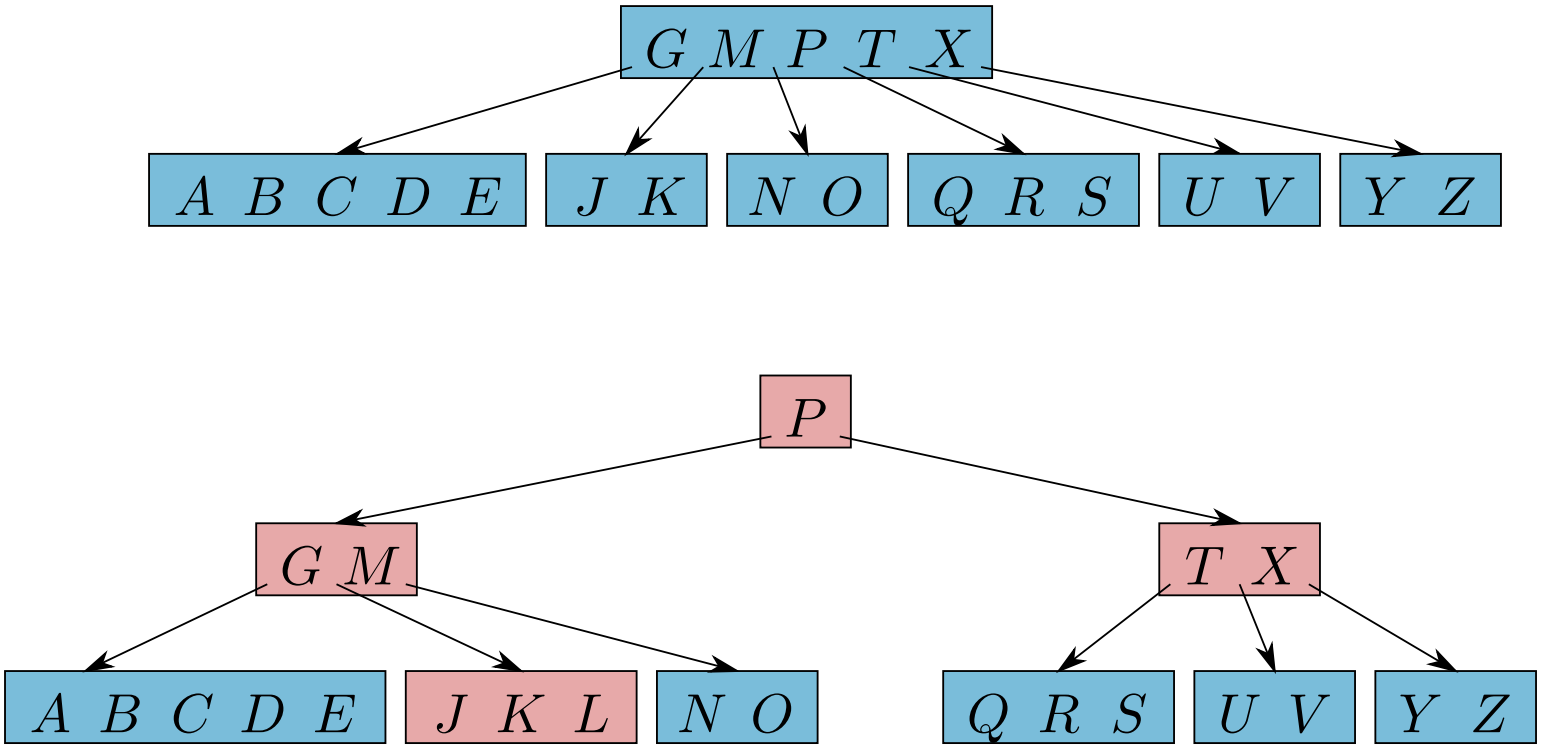
- rozdělení uzlu způsobí navýšení počtu následníků předchůdce rozdělovaného uzlu
- pokud se tím poruší vlastnost maximálního počtu následníků, tak musíme (rekurzivně) rozdělit i předchůdce
- proces rozdělování uzlů se v nejhorším případě zastaví až v kořeni stromu

optimalizace

- cílem je realizovat celou operaci přidání klíče při jednom průchodu stromu od kořene k listu *(optimalizace počtu přístupů na disk!!!)*
- rozdělování může nastat pouze u těch uzlů, které jsou plné, tj. obsahují maximální povolený počet klíčů $(2t - 1)$
- vždy, když procházíme přes plný uzel, rozdělíme ho na dva nové uzly a to tak, že každý ze dvou nových uzlů dostane $t - 1$ klíčů a jeden klíč se přesune do jejich otce
- korektnost postupu je garantována, protože předchůdce rozdělovaného uzlu není plný

Přidání klíče L - procházíme přes plný uzel

minimální stupeň stromu je 3



Přidání klíče - implementace

B-Tree_Insert(T, k)

```
1  r ← T.root
2  if r.n = 2t − 1
3     then s ← ALLOCAT_NODE()
4          T.root ← s
5          s.leaf ← false
6          s.n ← 0
7          s.c_1 ← r
8          B-TREE_SPLIT(s, 1)
9          B-TREE_INSERT_NONFULL(s, k)
10    else B-TREE_INSERT_NONFULL(r, k)
11 fi
```

- řádky 3 - 9 řeší plný kořen stromu
- na konci se volá procedura B-TREE_INSERT_NONFULL, která vloží klíč do stromu, jehož kořen není plný

Přidání klíče - implementace

B-Tree_Insert_Nonfull(x, k)

```
 1 i ← x.n
 2 if x.leaf
 3    then while i ≥ 1 ∧ x.key_i > k
 4                 do x.key_{i+1} ← x.key_i
 5                 i ← i − 1 od
 6         x.key_{i+1} ← k
 7         x.n ← x.n + 1
 8         Disk_Write(x)
 9    else while i ≥ 1 ∧ x.key_i > k do i ← i − 1 od
10         i ← i + 1
11         Disk_Read(x.c_i)
12         if x.c_i.n = 2t − 1 then B-Tree_Split(x, i)
13                             if x.key_i < k then i ← i + 1 fi fi
14         B-Tree_Insert_Nonfull(x.c_i, k)
15 fi
```

Přidání klíče - složitost

- počet operací DISK_WRITE a DISK_READ je $\mathcal{O}(h)$
  (*vždy jenom jedna mezi dvěma voláními* B-TREE_INSERT_NONFULL*)*
- celková složitost je $\mathcal{O}(th) = \mathcal{O}(t \log_t n)$
- procedura B-TREE_INSERT_NONFULL je tail - rekurzivní, a proto je počet uzlů, které musí být uloženy v operační paměti, konstantní

Odstranění klíče

odstranění probíhá analogicky jako u binárního vyhledávacího stromu

- jestliže se klíč určený k odstranění nachází v listu, odstraníme ho
- jestliže se klíč určený k odstranění nachází v uzlu, který není listem, nahradíme ho jeho následníkem (resp. předchůdcem) a následníka odstraníme z listu ve kterém se původně nacházel
- samotné mazání klíče se **vždy** realizuje v listu
- operace má stejnou asymptotickou složitost jako u BVS
- samotná implementace má ale několik speciálních případů, protože klíč může být odstraněn z libovolného uzlu
- v optimalizované variantě klíč odstraníme při jednom průchodu stromem od kořene dolů, s možnou výjimkou návratu do uzlu, ve kterém byl původně uložen odstraňovaný klíč

Odstranění klíče - základní varianta

odstranění klíče k z listu x

1. list x je současně kořenem stromu
   - klíč k odstraníme
2. list x není kořenem a obsahuje alespoň t klíčů
   - klíč k odstraníme
3. list x není kořenem a obsahuje přesně $t-1$ klíčů
   - vezmi toho bratra y listu x, který má více klíčů
   - vytvoř seznam obsahující klíče z listů x a y a navíc ten klíč z otce p listu x, který tvoří hranici mezi x a y
   - délka seznamu je $t-2$ (= počet klíčů v x) $+ 1$ (= klíč z otce) $+$ počet klíčů v $y \geq t-2+1+t-1$
   - rozlišujeme dva případy podle délky seznamu

Odstranění klíče - základní varianta - délka seznamu

3 A seznam obsahuje alespoň $2t - 1$ klíčů

- seznam rozdělíme na 3 části: *Left*, *Middle* a *Right*, kde *Middle* je medián seznamu, *Left* obsahuje klíče menší než medián a *Right* klíče větší než medián
- klíč *Middle* vrátíme do otce p, ze kterého jsme předtím odebrali hraniční klíč
- klíče *Left* a *Right* vložíme do dvou uzlů x a y
- uzly x a y mají alespoň $t - 1$ klíčů, počet klíčů v uzlu p zůstal nezměněný, hotovo

3 B seznam obsahuje právě $2t - 2$ klíčů

- uzly x a y nahradíme jediným uzlem obsahujícím všechny klíče seznamu
- nový list má povolený počet klíčů
- otec p má počet klíčů o 1 nižší než původně
- v případě, že počet klíčů v uzlu p klesl pod minimální hranici $t - 1$, opakujeme (rekurzivně) postup pro uzel p

Odstranění klíče - základní varianta

- po odstranění klíče z listu může klesnout počet klíčů v jeho uzlu pod minimální hranici
- musíme realizovat operace, které obnoví platnost podmínky minimálního počtu klíčů v uzlu
- může nastat situace, když se prochází strom od kořene k listu a potom zpátky od listu ke kořeni (*např. když všechny uzly na cestě od kořene do listu obsahujícího klíč mají stupeň přesně* t)
- podobně jako při vkládaní klíče optimalizujeme proces odstranění klíče tak, abychom minimalizovali počet přístupů na disk

Odstranění klíče - optimalizace

- postupujeme od kořene směrem k listu
- vždy, když procházíme přes uzel, který má přesně $t - 1$ klíčů, tak uděláme takovou korekci, která zvýší počet klíčů v uzlu na t
- když narazíme na uzel, ze kterého potřebujeme odstranit klíč, máme garanci, že jeho otec má alespoň t klíčů
- když odstranění klíče z uzlu způsobí snížení počtu klíčů v jeho otci, nevznikne žádný problém

Optimální odstranění klíče - pravidla

odstraňujeme klíč k

1. když klíč k je v listu x, odstraň k z x

2. když klíč k je ve vnitřním uzlu x, tak
   - **a.** jestliže syn y, který je před k v x, obsahuje alespoň t klíčů, tak najdi v podstromě s kořenem y předchůdce k' klíče k; nahraď v x klíč k klíčem k'; rekurzivně odstraň klíč k'
   - **b.** jestliže syn y má méně než t klíčů tak, symetricky, prozkoumej syna z, který následuje za k v x; v případě že z obsahuje alespoň t klíčů, tak najdi v podstromě s kořenem z následníka k' klíče k; nahraď v x klíč k klíčem k'; rekurzivně odstraň klíč k'
   - **c.** v případě, že synové y i z mají jen $t-1$ klíčů, tak do vrcholu y přesuň klíč k a všechny klíče z vrcholu z; z vrcholu x odstraň k a ukazatel na z; nový uzel y obsahuje $2t-1$ klíčů (mezi nimi i klíč k); rekurzivně odstraň k z y

Odstranění klíče - pravidla, pokračování

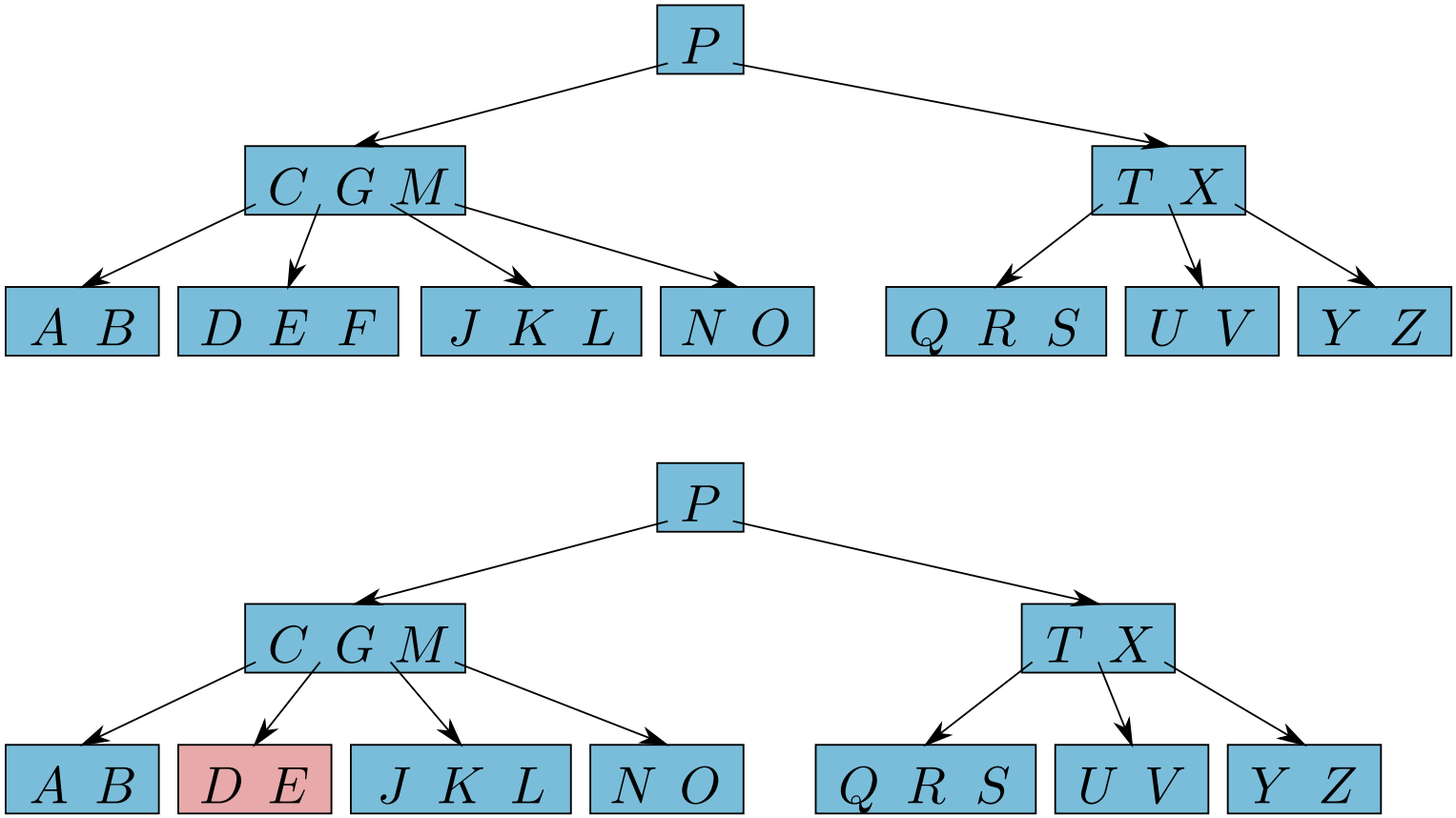
3 když klíč k není ve vnitřním uzlu x , tak urči kořen $x.c_i$ stromu, který musí obsahovat k (za předpokladu, že k je v stromě); v případě, že uzel $x.c_i$ obsahuje jen $t-1$ klíčů, pokračuj body 3.a. anebo 3.b. které zaručí, že rekurzivní volání se aplikuje na uzel obsahující alespoň t klíčů; rekurzivně odstraň klíč k z vhodného následníka uzlu x

a. v případě, že $x.c_i$ obsahuje jen $t-1$ klíčů, ale některý z jeho přímých bratrů obsahuje alespoň t klíčů, tak zvyš počet klíčů v $x.c_i$ a to tak, že přesuneš klíč z x do $x.c_i$, přesuneš klíč z bratra x a přesuneš příslušný ukazatel na následníka z bratra do uzlu $x.c_i$

b. v případě, že $x.c_i$ i jeho jeho přímí bratři obsahují jen $t-1$ klíčů, tak přesuň do $x.c_i$ jeden klíč z x a všechny klíče z jednoho z bratrů

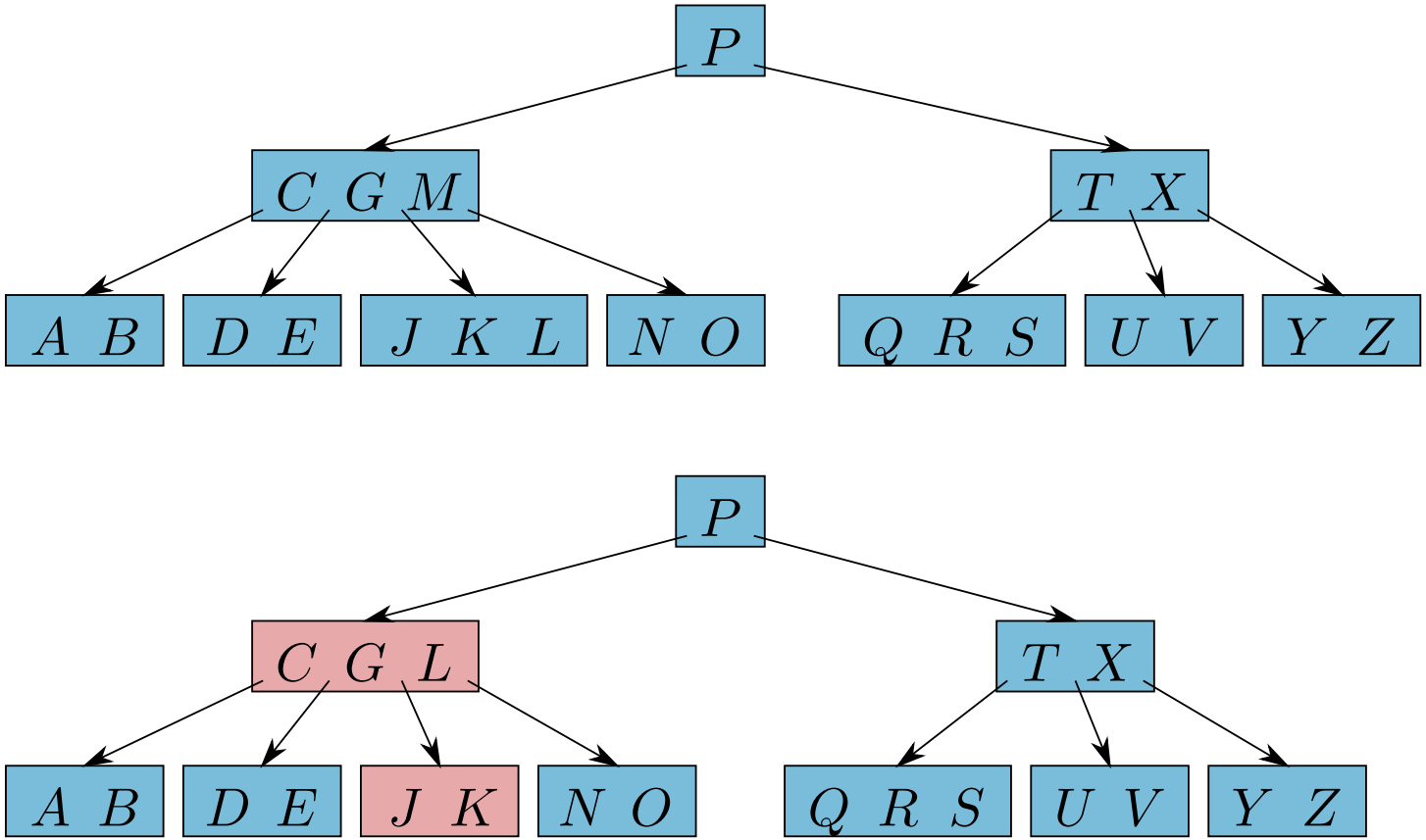
Odstranění klíče F – případ 1

$t = 3$



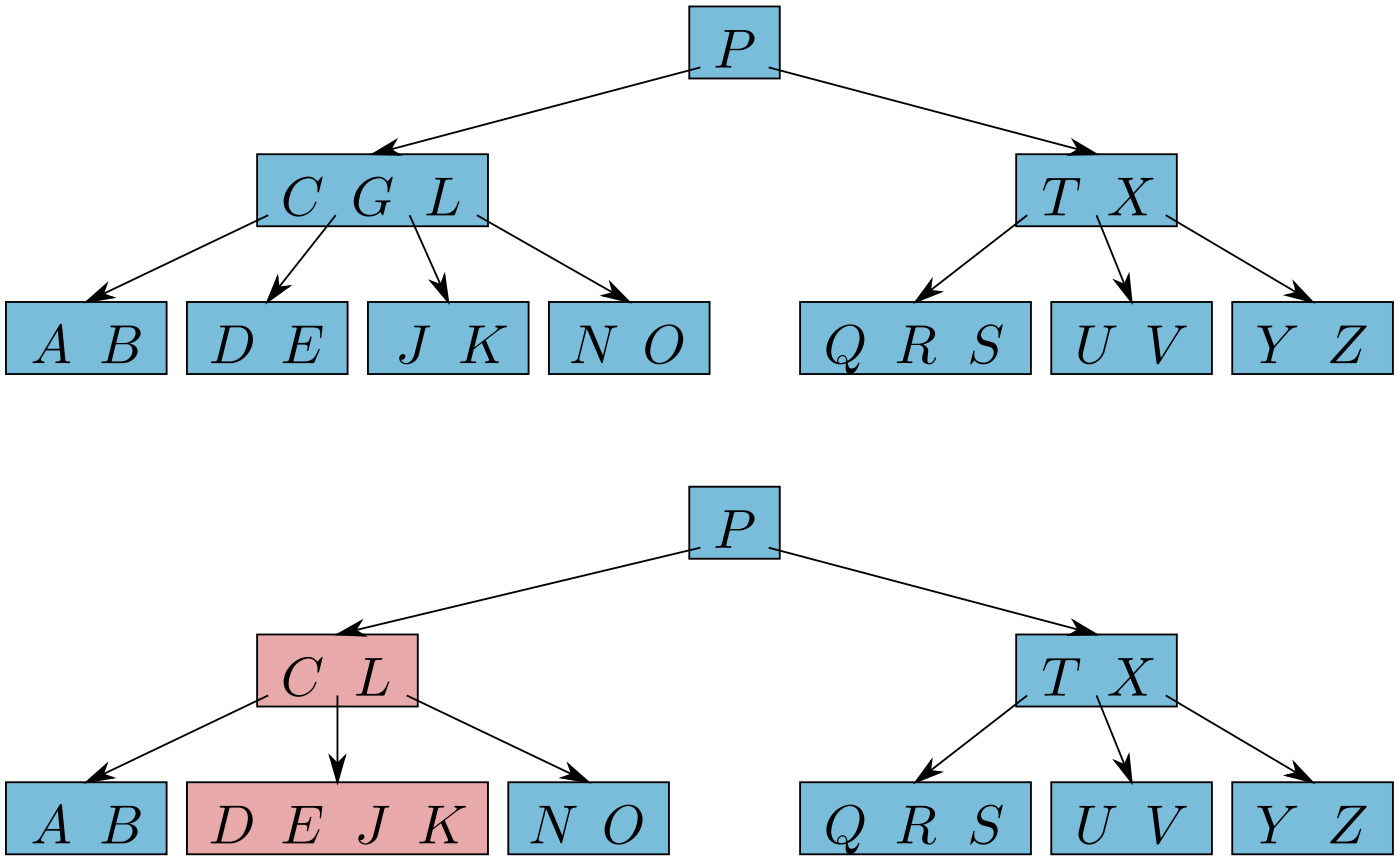
Odstranění klíče M – případ 2a

$t = 3$



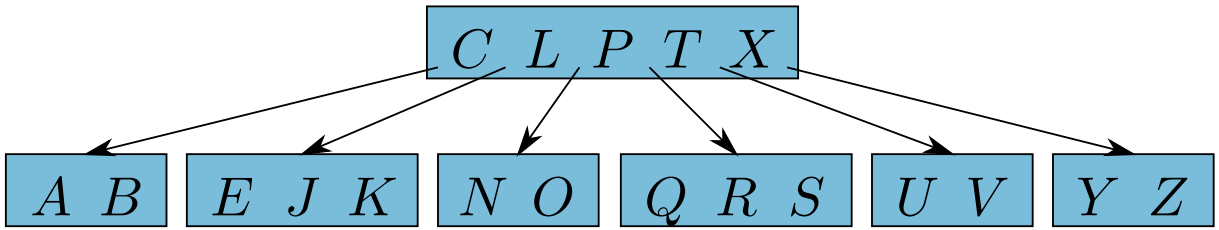
Odstranění klíče G – případ 2c

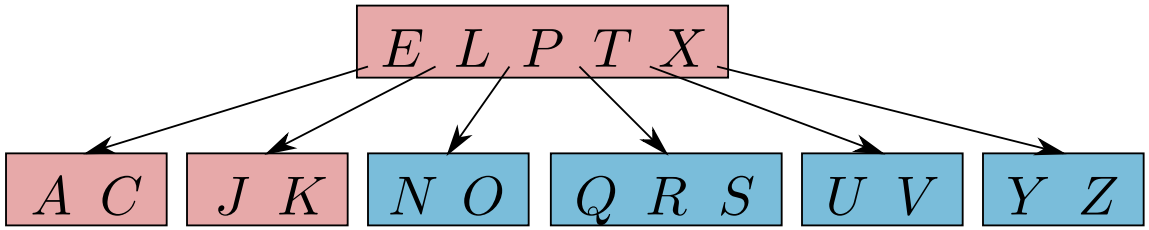
$t = 3$



Odstranění klíče B – případ 3a

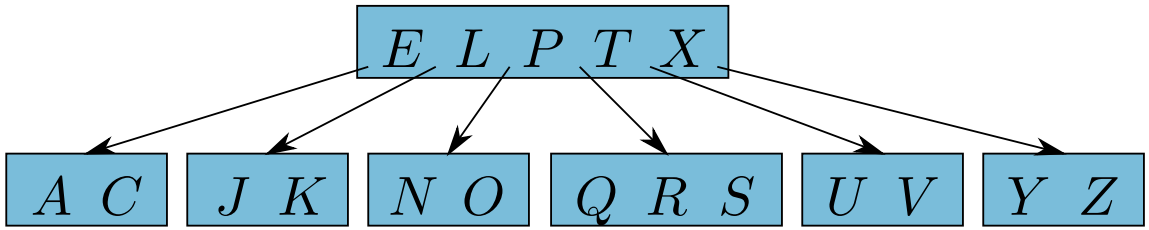
$t = 3$

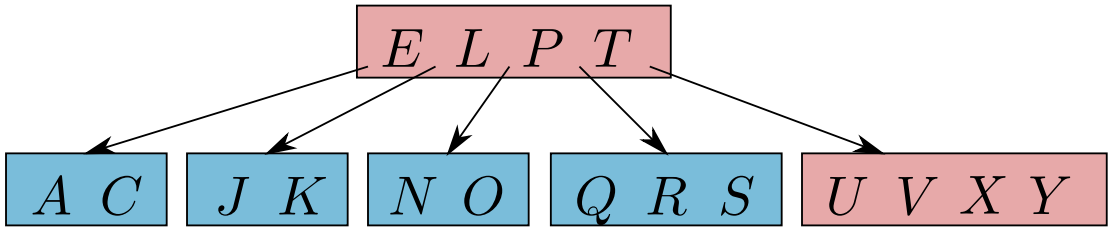




Odstranění klíče Z – případ 3b

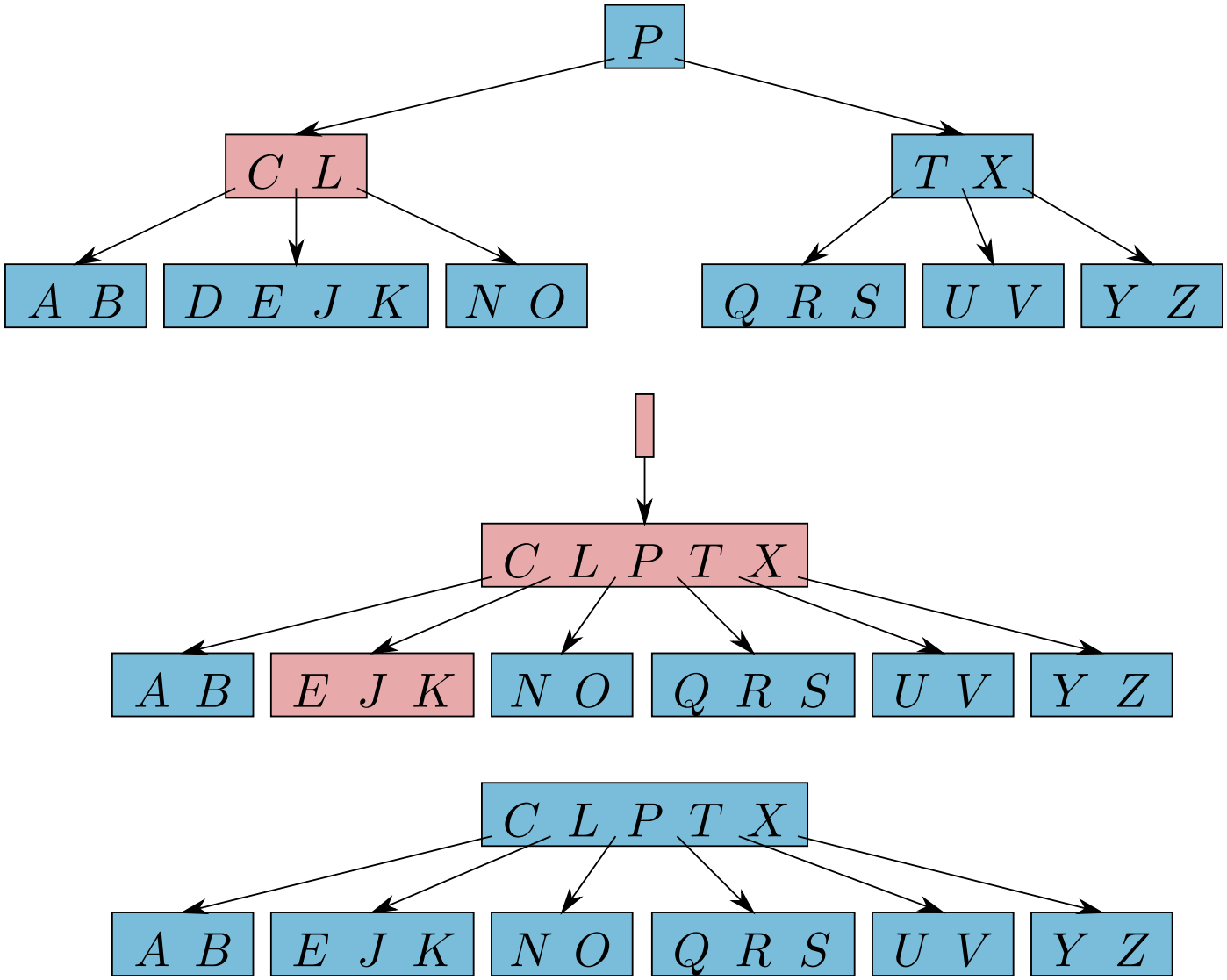
$t = 3$





Odstranění klíče D – případ 3b

$t = 3$



Odstranění klíče - složitost

- v případě, že se odstraňovaný klíč nachází v listu, procedura prochází od kořene k listu bez nutnosti návratu
- v případě, že se klíč nachází ve vnitřním uzlu, tak procedura postupuje od kořene k listu s možným návratem do vrcholu, ze kterého byl klíč odstraněn a nahrazen svým předchůdcem anebo následníkem (případy 2.a., 2.b.)
- mezi dvěma rekurzivními voláním se vykoná nanejvýš jedna operace DISK_WRITE a jedna operace DISK_READ; jejich celkový počet je proto $\mathcal{O}(h)$
- celková složitost je $\mathcal{O}(th) = \mathcal{O}(t \log_t n)$

B+ stromy

- klíče jsou uloženy pouze v listech
- zřetězení listů zachovává pořadí klíčů
- vnitřní uzly B+ stromů indexují listy

výhody a nevýhody

- klíč v B stromě se najde před dosažením listu
- vnitřní uzly B stromů jsou větší, do uzlů se proto může uložit méně klíčů a strom je hlubší
- operace vkládaní a odstraňování klíče z B stromu jsou komplikovanější
- implementace B stromu je náročnější než implementace B+stromu

Datové struktury

Slovník

- dynamický datový typ pro reprezentaci množiny objektů
- podporované operace
  $\textsc{Insert}(S, x)$ do množiny S přidá objekt x
  $\textsc{Search}(S, x)$ zjistí, zda množina S obsahuje objekt x
  $\textsc{Delete}(S, x)$ z množiny S odstraní objekt x

vhodné datové struktury pro implementaci slovníku

seznam všechny operace mají složitost $\mathcal{O}(n)$ (n je mohutnost množiny S)

vyhledávací strom se dá použít za předpokladu, že objekty mají číselný klíč, při použití vyváženého stromu je složitost operací $\mathcal{O}(\log n)$

cíl: složitost všech operací

v nejhorším případě $\Theta(n)$

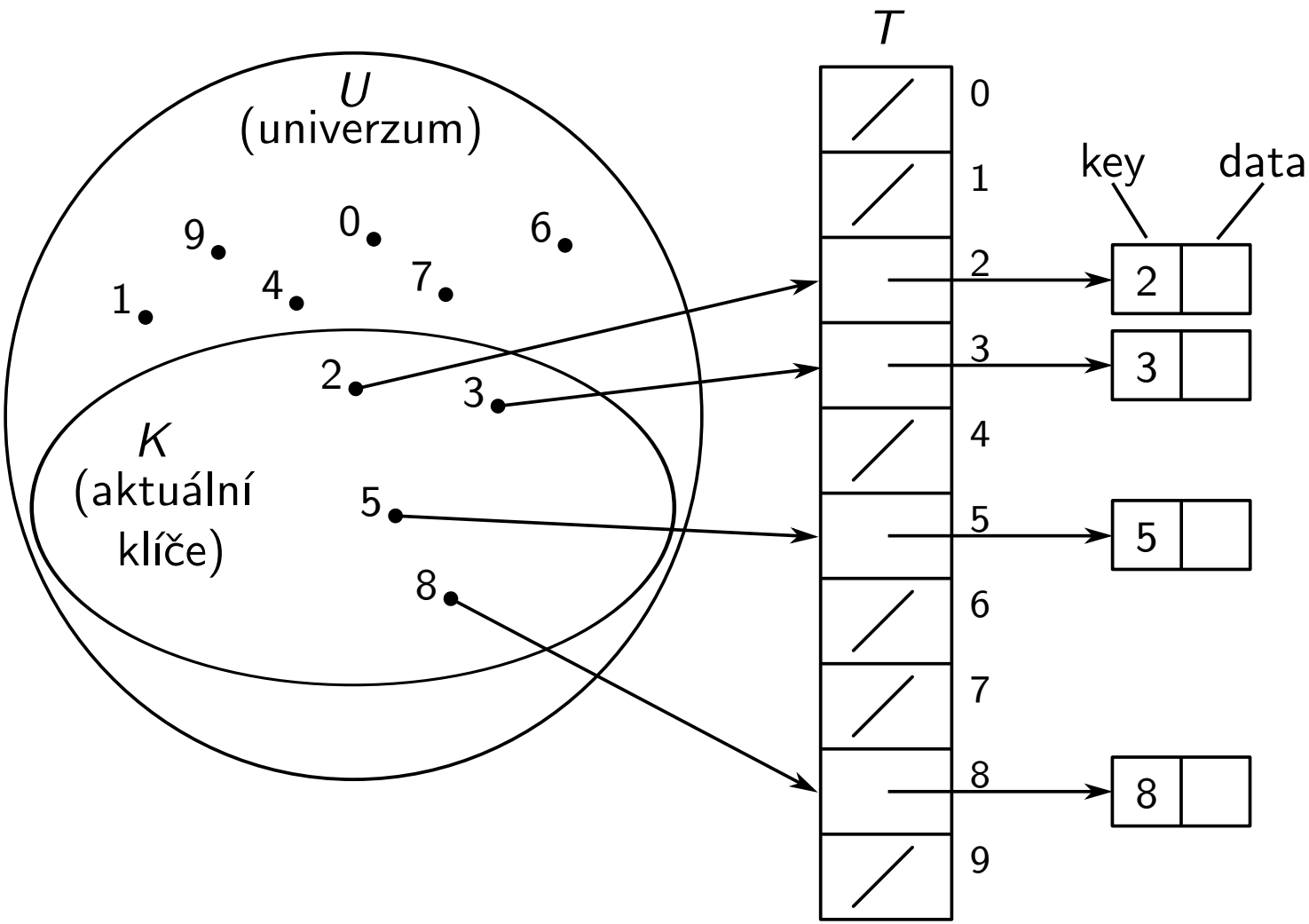
v očekávaném případě $\mathcal{O}(1)$

Přímé adresování

- každý prvek reprezentované množiny prvků má přiřazen klíč vybraný z univerza $U = \{0, 1, \ldots, m-1\}$
- **žádné dva prvky nemají přiřazený stejný klíč**

- pole $T[0 \ldots m-1]$
  - každý slot (pozice) v T odpovídá jednomu klíči z U
  - když reprezentovaná množina obsahuje prvek x s klíčem k, tak $T[k]$ obsahuje ukazatel na x
  - v opačném případě je $T[k]$ prázdné (NIL)
- složitost operací je konstantní

Přímé adresování - schéma



Výhody a nevýhody přímého adresování

výhody

- konstantní složitost všech operací
- jednoduchá implementace

nevýhody

- v případě, že univerzum U je veliké, tak uchovávání tabulky velikosti univerza je neefektivní resp. nemožné
- v případě, že množina aktuálně uložených klíčů je malá ve srovnání s velikostí univerza, tak větší část paměti alokované pro tabulku T je nevyužitá
- problém objektů se stejným klíčem

Hašovací tabulka

- v případě, že množina aktuálně uložených klíčů K je výrazně menší než U, využívá hašovací tabulka výrazně méně paměti, než tabulka s přímým přístupem
- potřebný prostor se dá redukovat až na $\Theta(|K|)$
- složitost operací zůstává konstantní avšak v *očekávaném* (a ne v nejhorším) případě

rozdíly

přímé adresování	prvek x s klíčem k uloží v tabulce na pozici $T[k]$
hašování	prvek x s klíčem k uloží v tabulce na pozici $T[h(k)]$

- h je funkce $h : U \longrightarrow \{0, 1, \ldots, m-1\}$
- h se nazývá hašovací funkce

Hašovací tabulka - problémy k řešení

1. řešení kolizí

kolize ≈ dva anebo více klíčů zahašujeme na stejnou pozici
pro $x \neq y$ *je* $h(x) = h(y)$, x *a* y *mají stejný otisk*

- zřetězené hašování *(chaining)*
- otevřená adresace *(open adressing)*

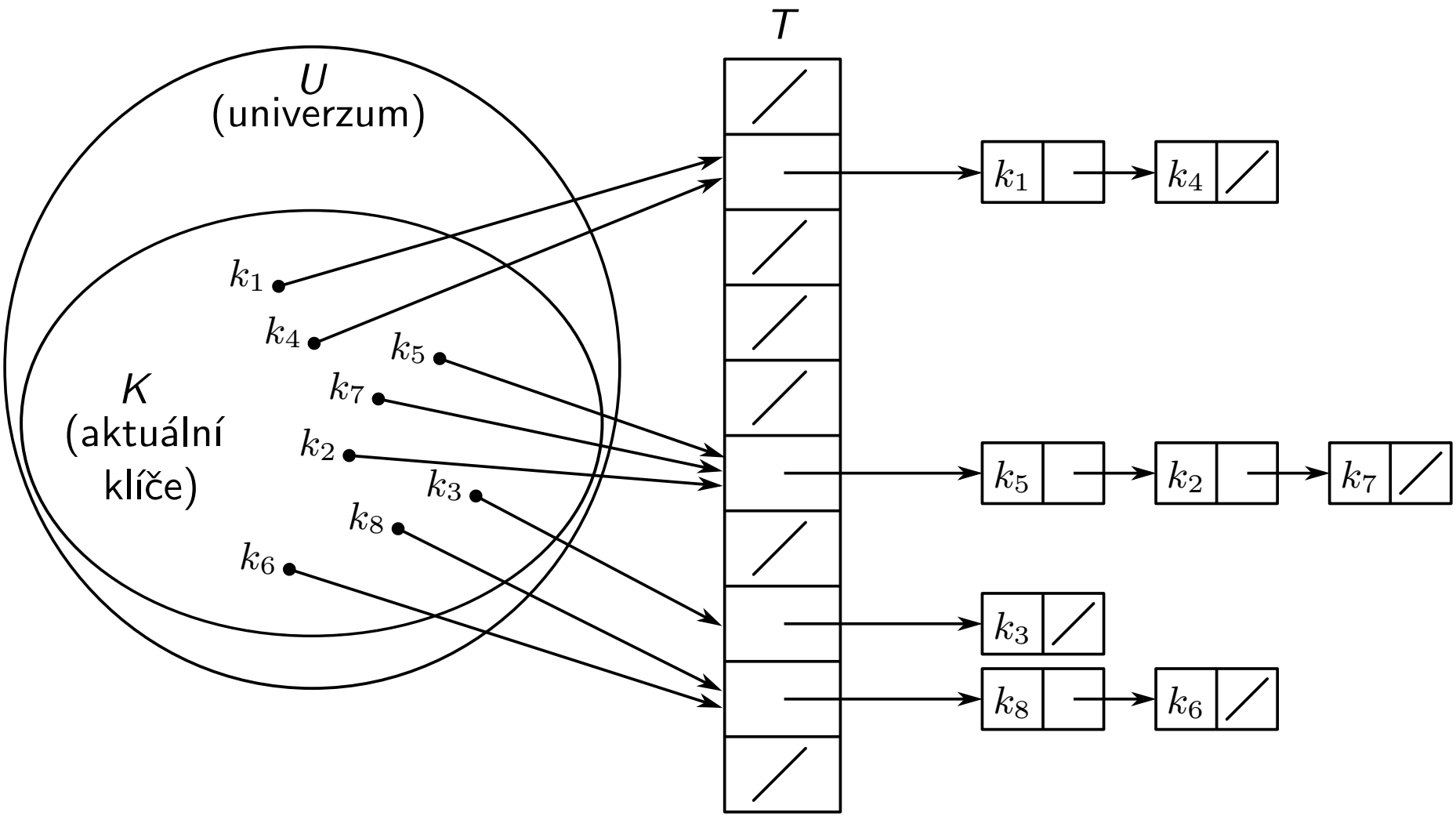
2. výběr hašovací funkce

- minimalizovat počet kolizí
- efektivní výpočet funkce

Zřetězené hašování

- každá položka tabulky obsahuje (ukazatel na) seznam prvků zahašovaných na stejnou pozici
- seznam je prázdný právě když žádný prvek nebyl zahašovaný na danou pozici
- vkládání prvku x do hašovací tabulky T se realizuje jako přidání prvku na začátek seznamu $T[h(x.key)]$
- prvek x vyhledáváme v seznamu $T[h(x.key)]$
- prvek x odstraníme vymazáním ze seznamu $T[h(x.key)]$

Zřetězené hašování - schéma



Zřetězené hašování - složitost

složitost v nejhorším případě

Insert konstantní (*za předpokladu, že vkládaný prvek není v tabulce*)

Search úměrná délce seznamu; v nejhorším případě $\Theta(n)$, kde n je počet prvků uložených v tabulce

Delete (asymptoticky) stejná jako složitost SEARCH (*za předpokladu dvousměrného seznamu*)

složitost v průměrném případě

záleží od výběru hašovací funkce

Zřetězené hašování - průměrná složitost

- předpokládáme, že hašovací funkce je **jednoduchá uniformní** *(simple uniform)*, tj. že pro každý prvek univerza je pravděpodobnost jeho zahašování na kterýkoliv index tabulky stejná (a nezávislá od toho, kam jsou zahašovány zbylé prvky univerza)
- složitost operací se vyjadřuje vzhledem k **faktoru naplnění** *(load factor)*
- pro danou tabulku s m pozicemi, ve které je uložených n prvků, definujeme faktor naplnění α předpisem $\alpha = n/m$, tj. průměrný počet prvků zahašovaných na stejnou pozici
- pro $j = 0, 1, \ldots, m-1$ nechť n_j označuje délku seznamu $T[j]$
- pro jednoduchou uniformní hašovací funkci platí, že **očekávaná délka seznamu $T[j]$ je**

$$E[n_j] = \alpha = n/m$$

- předpokládáme, že výpočet hodnoty funkce má konstantní časovou složitost

Zřetězené hašování - průměrná složitost

V hašovací tabulce, ve které jsou kolize řešeny zřetězením a ve které se používá jednoduchá uniformní funkce, má operace neúspěšného vyhledávaní prvku průměrnou časovou složitost $\Theta(1+\alpha)$.

V hašovací tabulce, ve které jsou kolize řešeny zřetězením a ve které se používá jednoduchá uniformní funkce, má operace úspěšného vyhledávaní prvku průměrnou časovou složitost $\Theta(1+\alpha)$.

- v případě, že počet pozic v tabulce je proporcionální počtu prvků v tabulce, $n = \mathcal{O}(m)$, platí $\alpha = n/m = \mathcal{O}(m)/m = \mathcal{O}(1)$
- vyhledávaní prvku má konstantní průměrnou složitost
- samotné vložení prvku a odstranění prvku ze seznamu má konstantní složitost
- **všechny operace mají za daných předpokladů konstantní průměrnou složitost**

Výběr hašovací funkce

jak vybrat dobrou hašovací funkci?

- funkce by měla mít vlastnosti jednoduché uniformní funkce: každý klíč je zahašován na všechny pozice se stejnou pravděpodobností
- v praxi je těžké ověřit podmínku uniformity, protože neznáme rozložení klíčů (a navíc jsou na sobě často závislé)
- v praxi využíváme při volbě hašovací funkce znalosti rozložení klíčů s cílem, aby se často společně se vyskytující klíče zahašovaly na různé pozice
- **příklad:** když klíče jsou vybírány náhodně s uniformním rozdělením z intervalu $\langle 0, 1)$, tak hašovací funkce $h(k) = \lfloor k \cdot m \rfloor$ je jednoduchou uniformní funkcí

Klíče jako přirozené čísla

- většina hašovacích funkcí je navržená pro univerzum - množinu přirozených čísel $\mathbb{N}$
- když klíče nejsou přirozená čísla, můžeme je interpretovat jako přirozená čísla použitím vhodného kódování

příklad

- znakový řetězec interpretujeme jako číslo (ve vhodně zvolené číselné soustavě)
- řetězec CLRS
- ASCII hodnoty: C = 67, L = 76, R = 82, S = 83
- máme 128 ASCII hodnot, volíme proto číselnou soustavu se základem 128
- CLRS interpretujeme jako $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0)$

Hašovací funkce - metoda dělení

$$h(k) = k \mod m$$

příklad $m = 20,\ k = 91 \Longrightarrow h(k) = 11$

výhody rychlost

nevýhody špatné chování pro některé m

- pro $m = 2^p$ je hodnota $h(k)$ vždy p nejpravějších bitů z k
- když k je znakový řetězec interpretovaný při základě 2^p, tak hodnota $m = 2^p - 1$ není vhodná, protože po permutaci řetězce se hodnota hašovací funkce nezmění
- dobrou volbou pro m je prvočíslo

Hašovací funkce - metoda binárního násobení

- předpoklad: univerzum je U množina binárních čísel délky w
- předpoklad: velikost m tabulky je mocninou dvojky, $m = 2^p$
- cílem je zahašovat w-bitové čísla na p-bitové čísla

- zvolíme libovolnou konstantu A, $0 < A < 1$

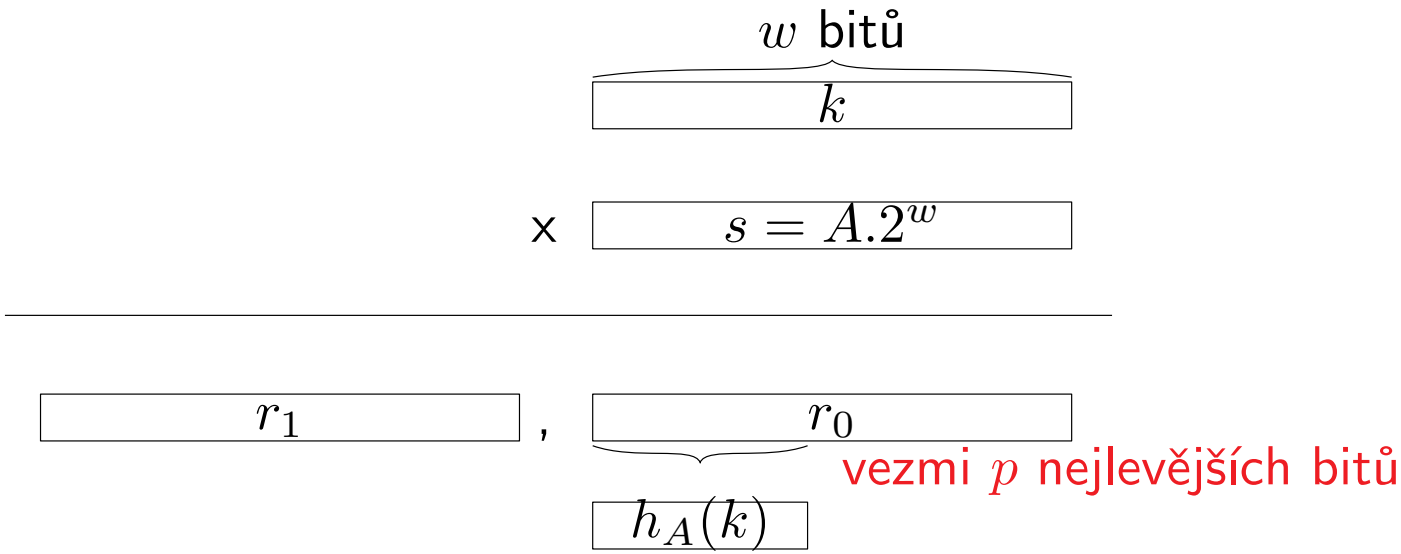
$$h_A(k) = \lfloor m \ (k \ A \mod 1) \rfloor$$

postup výpočtu

1. vynásob klíč k konstantou A a ze součinu vezmi desetinnou část
2. výsledek vynásob číslem m a ze součinu vezmi celou část

$$h_A(k) = \lfloor m \ (k \ A \mod 1) \rfloor$$

- zvolíme A tvaru $s/2^w$
- vynásobíme čísla k a s
- výsledkem násobení je $2w$ bitové číslo, kde r_1 je celočíselná část součinu kA a r_0 je desetinná část součinu *(viz obrázek)*
- pro další výpočet potřebujeme pouze r_0
- potřebujeme celou část součinu čísel r_0 a m
- vzhledem k tomu, že $m = 2^p$, násobení znamená posun o p bitů doleva
- ve skutečnosti nemusíme vůbec násobit a stačí vzít p nejvýznamnějších bitů čísla r_0



Metoda binárního násobení - příklad

- $w = 5, m = 8, w = 3$, tj. hašujeme 5 bitové čísla, velikost tabulky je $8 = 2^3$ a chceme hašovat na 3 bitové čísla
- hašujeme klíč $k = 21$
- vybíráme konstantu A tvaru $s/2^w$ a takovou, aby $0 < A < 1$ – proto musí platit $0 < s < 2^5$, vybereme $s = 13 \Longrightarrow A = 13/32$

výpočet $h_A(k)$ podle vzorce $h_A(k) = \lfloor m\ (k\ A\ \mod 1) \rfloor$

- $kA = 21 \cdot 13/32 = 8\frac{17}{32}$
- $kA\ \mod 1 = 17/32$
- $m(kA\ \mod 1) = 8 \cdot 17/32 = 4\frac{1}{4}$
- $\lfloor m(kA\ \mod 1) \rfloor = 4 = h_A(k)$

implementace

- $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17$
- $r_1 = 8, r_0 = 17$; bitový zápis r_0 je 10001
- vezmeme $p = 3$ nejvýznamnější bity r_0, tj. 100 (4 v desítkové soustavě)
- $h_A(k) = 4$

Univerzální hašování

scénář ani nejlepší hašovací funkce negarantuje dobré chování hašování v případě, že klíče určené k zahašování jsou vybrány tím nejhorším možným způsobem
(můžeme si představit útočníka, který pozná náš hašovací program a hašovací funkci a na základě toho dokáže vybrat takové klíče, které se zahašují na stejnou pozici, viz analogii s výběrem pivota pro Quicksort)

řešení při každém použití hašovacího programu vybereme náhodně jinou hašovací funkci
(když útočník neví, jaká hašovací funkce bude vybrána, nemůže záměrně vybírat vstupy, které povedou k špatnému chování)

výběr funkce samotný fakt náhodného výběru funkce ještě negarantuje efektivitu hašování; je potřebné vybírat z vhodných kandidátů

Univerzální hašování

Definice 6

Nechť $\mathcal{H}$ je konečná množina hašovacích funkcí, které mapují univerzum klíčů U na m pozic. $\mathcal{H}$ je **univerzální množinou hašovacích funkcí** *právě když pro každou dvojici klíčů $k, l \in U$, $k \neq l$, je počet hašovacích funkcí $h \in \mathcal{H}$, pro které $h(k) = h(l)$, nejvýše $|\mathcal{H}|/m$.*

Věta 7

Předpokládejme, že hašovací funkce, náhodně vybraná z univerzální množiny hašovacích funkcí, je použita pro zahašování n klíčů do tabulky s m pozicemi. Pak pro klíč k platí, že když

- *k není v tabulce, tak očekávaná délka seznamu, do kterého se zahašuje k, je nejvýše $\alpha = n/m$*
- *k je v tabulce, tak očekávaná délka seznamu, který obsahuje k, je nejvýše $\leq 1 + \alpha$.*

Univerzální hašování - složitost

Důsledek 8

Libovolná posloupnost n operací Insert, Search *a* Delete, *z nichž nejvýše $\mathcal{O}(m)$ operací je typu* Insert, *má očekávanou časovou složitost $\Theta(n)$ za předpokladu použití zřetězeného hašování, univerzální množiny hašovacích funkcí a tabulky s m pozicemi.*

Důsledek 9

Použitím univerzálního hašování a řešení kolizí řetězením v tabulce s m pozicemi zabere očekávaný čas $\Theta(n)$ jakákoliv posloupnost n operací Insert, Search *a* Delete, *která obsahuje $\mathcal{O}(m)$ operací* Insert.

Konstrukce univerzální množiny hašovacích funkcí

příklad univerzálního hašování

- zvolíme prvočíslo p takové, že žádný klíč není větší než p
- pro libovolná čísla $a \in \{1, 2, \ldots p-1\}$ a $b \in \{0, 1, \ldots p-1\}$ definujeme hašovací funkci předpisem

$$h_{ab}(k) = ((ak + b) \mod p) \mod m)$$

- množina funkcí

$$\mathcal{H}_{pm} = \{h_{ab} | a \in \{1, 2, \ldots p-1\}, b \in \{0, 1, \ldots p-1\}$$

je univerzální množinou hašovacích funkcí

- výběr prvočísla umožňuje efektivní implementaci operací $\mod$

přesné důkazy tvrzení jako i další podrobnosti týkající se univerzálního hašovaní jsou v literatuře, např. v monografii T. Cormen, Ch. Leiserson, R. Rivest, C. Stein: Introduction to Algorithms. *Third Edition. MIT Press, 2009*

Otevřená adresace

- všechny klíče ukládáme přímo do tabulky, počet klíčů nemůže přesáhnout velikost tabulky
- při vyhledávání se systematicky zkoumají pozice tabulky, dokud není nalezen hledaný klíč nebo není jasné, že v tabulce není
- nepotřebujeme seznamy a ukazatele, místo nich se počítá sekvence pozic v tabulce, které mají být prozkoumány (tzv. sondování)

Otevřená adresace - vyhledávání

- hašovací funkce je typu $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$
- pro každý klíč potřebujeme posloupnost $\langle h(k, 0), h(k, 1), \ldots h(k, m-1)\rangle$, která je permutací posloupnosti $\langle 0, 1, \ldots, m-1\rangle$
- každá pozice tabulky obsahuje buď klíč, anebo hodnotu $\textsc{Nil}$
- při hledání klíče k
  - proměnná i je rovna pořadovému číslu testu, iniciální hodnota i je 0
  - vypočítáme hodnotu $h(k, i)$ a testujeme obsah pozice $h(k, i)$
  - když pozice $h(k, i)$ obsahuje klíč k, vyhledávání je úspěšné
  - když pozice $h(k, i)$ obsahuje hodnotu $\textsc{Nil}$, vyhledávání je neúspěšné (tabulka neobsahuje klíč k)
  - když pozice $h(k, i)$ obsahuje neprázdnou hodnotu různou od k, tak zvýšíme pořadové číslo testu a vypočítáme novou pozici v tabulce jako funkci k a pořadového čísla testu a klíč hledáme pomocí této nové hašovací funkce

Otevřená adresace - vkládání

- analogicky jako při vyhledávání najdeme volnou pozici v tabulce
- vkládání skončí úspěchem když je nalezena volná pozice, na kterou se klíč vloží
- když počet testů dosáhne m, tak vkládání končí neúspěchem

Otevřená adresace - odstranění klíče

- vyhledáme klíč k v tabulce, nechť se nalézá na pozici j
- může nastat situace, že po odstranění klíče k budeme v tabulce vyhledávat klíč k', který je v tabulce uložen) a v průběhu jeho vyhledávání budeme zkoumat i pozici j
- když bychom na pozici j vložili hodnotu NIL, tak bychom při následném vyhledávání klíče k' dostali nesprávný výsledek

řešení

- místo hodnoty NIL použijeme speciální hodnotu DELETED
- operace INSERT považuje pozici s hodnotou DELETED za prázdnou
- operace SEARCH považuje pozici s hodnotou DELETED za obsazenou, ale obsahující jinou hodnotu než hledaný klíč

Otevřená adresace - výpočet sekvence sond

nejčastěji se používají k výpočtu sekvence sond tři techniky

- lineární adresace *(linear probing)*
- kvadratická adresace *(quadratic probing)*
- dvojité hašování *(double hashing)*

Otevřená adresace - lineární

využívá pomocnou hašovací funkci $h' : U \longrightarrow \{0, 1, \ldots, m-1\}$

$$h(k, i) = (h'(k) + i) \mod m$$

- pro daný klíč je nejdříve prozkoumána pozice $T[h'(k)]$, pak pozice $T[h'(k)+1]$, ..., $T[m-1]$ a pak zase od $T[0]$ až k $T[h'(k)-1]$
- problémem je tzv. primární shlukování, které může výrazně zvýšit složitost operací

Otevřená adresace - kvadratická

využívá pomocnou hašovací funkci $h' : U \longrightarrow \{0, 1, \ldots, m-1\}$ a pomocné konstanty $c_1, c_2 \neq 0$

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

- pro daný klíč je nejdříve prozkoumána pozice $T[h'(k)]$, dále pak pozice posunuta o offset závislý kvadratickým způsobem na pořadí sondy
- kvadratická adresace je obvykle lepší než lineární
- problémem je vhodný výběr konstant c_1 a c_2 a velikosti tabulky m
- když dva klíče jsou primárně zahašovány na stejnou pozici protože $h'(k_1) = h'(k_2)$, tak mají stejnou celou posloupnost sond - tzv. sekundární shlukování

Otevřená adresace - dvojité hašování

využívá dvě pomocné hašovací funkce h_1, h_2

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

- pro daný klíč je nejdříve prozkoumána pozice $T[h_1(k)]$, následující pozice je posunuta o offset $h_2(k) \mod m$
- hodnota $h_2(k)$ musí být nesoudělná s velikostí hašovací tabulky m, aby byla prohledána celá tabulka
- vhodnou volbou je vzít m jako mocninu 2 a navrhnout h_2 tak, že výsledkem bude vždy liché číslo, nebo
- zvolit m jako prvočíslo a navrhnout h_2 tak, že výsledkem bude vždy kladné číslo $< m$
- dvojité hašování je lepší než kvadratické, protože generuje $\Theta(m^2)$ posloupností sond místo $\Theta(m)$ jako kvadratická adresace

Otevřená adresace - složitost

Věta 10

Pro hašovací tabulku s otevřenou adresaci s faktorem naplnění $\alpha = n/m < 1$ je očekávaný počet sond při neúspěšném hledání nejvýše $1/(1-\alpha)$ a to za předpokladu uniformního hašování.

Věta 11

Pro hašovací tabulku s otevřenou adresaci s faktorem naplnění $\alpha = n/m < 1$ je očekávaný počet sond při úspěšném hledání nejvýše $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ a to za předpokladu uniformního hašování.

uniformní hašování je takové, že každý klíč má jako posloupnost sond se stejnou pravděpodobností libovolnou z $m!$ permutací $\langle 0, 1, \ldots, m-1 \rangle$

Kukaččí hašování (Cuckoo hashing)

- pro hašování se používají **dvě tabulky** velkosti m a **dvě hašovací funkce** $h_1, h_2 : U \longrightarrow \{0, 1, \ldots m-1\}$
- každý klíč k je zahašovaný buď na pozici $h_1(k)$ v první tabulce, anebo na pozici $h_2(k)$ v druhé tabulce
- **hledání klíče** má konstantní složitost, protože stačí otestovat dvě pozice
- **odstranění klíče** má konstantní složitost, analogicky jako jeho hledání
- při **vkládání nového klíče** k se použije *hladová strategie*: nejdříve se pokusíme vložit klíč k na pozici $h_1(k)$
- když je pozice $h_1(k)$obsazena, tak klíč y uložený na pozici $h_1(k)$ přesuneme do druhé tabulky na jeho alternativní pozici $h_2(y)$
- proces opakujeme a přepínáme se mezi tabulkami dokud nenajdeme volnou pozici, anebo se proces zacyklí

R. Pagh, F. Rodler: Cuckoo hashing. Journal of Algorithms 51 (2004) 122 - 144

Dokonalé hašování (Perfect hashing)

- hašování, které má konstantní složitost i v nejhorším případě
- předpokladem je statická množina klíčů
- využívá dvě úrovně hašování

první úroveň

v podstatě stejná, jako zřetězené hašování

druhá úroveň

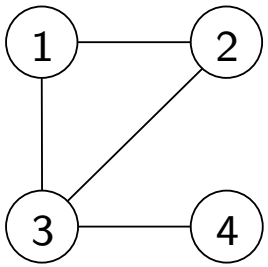
- místo seznamů použijeme sekundární hašovací tabulky S_j s asociovanou hašovací funkcí h_j, přičemž vhodným výběrem můžeme zajistit, aby na druhé úrovni nebyly žádné kolize
- velikost m_j tabulky S_j je kvadratická vůči počtu klíčů zahašovaných na pozici j
- hašovací funkce na první úrovni se vybírá z univerzální množiny hašovacích funkcí $\mathcal{H}_{pm}$, na druhé úrovni z univerzální množiny $\mathcal{H}_{pm_j}$

Grafové algoritmy

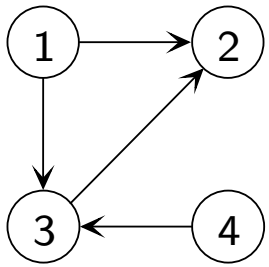
Graf a jeho reprezentace

- graf $G = (V, E)$
  - orientovaný / neorientovaný
  - ohodnocené hrany / vrcholy
  - jednoduché / násobné hrany
- reprezentace grafů
  - seznam následníků
  - matice sousednosti
- složitost grafových algoritmů je funkcí počtu vrcholů a hran
- používáme zjednodušenou notaci, např. $\mathcal{O}(V + E)$

Matice sousednosti



	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0



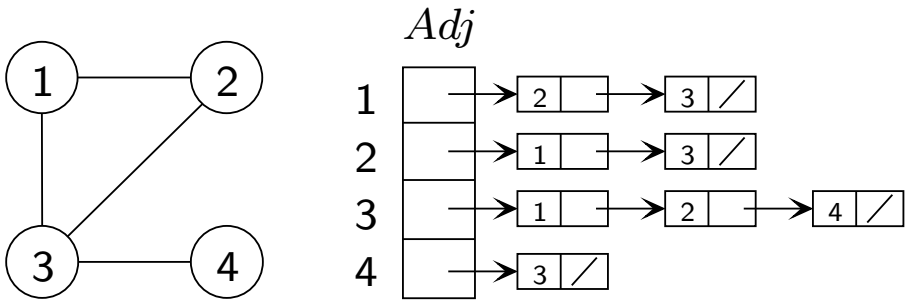
	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	0
4	0	0	1	0

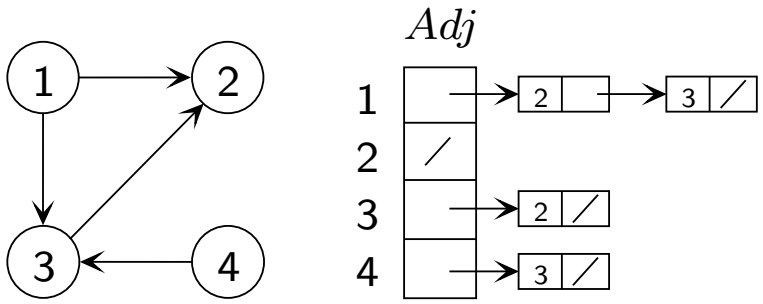
- matice $A = (a_{ij})$ rozměrů $|V| \times |V|$, kde

$$a_{ij} = \begin{cases} 1 & \text{pokud } (i,j) \in E \\ 0 & \text{jinak} \end{cases}$$

- prostorová složitost: $\Theta(V^2)$
- vhodné pro husté grafy
- časová složitost výpisu všech sousedů vrcholu u je $\Theta(V)$
- časová složitost ověření zda $(u,v) \in E$ je $\Theta(1)$

Seznam následníků





- pole Adj velikosti $|V|$
- prostorová složitost: $\Theta(V + E)$
- vhodné pro řídké grafy
- časová složitost výpisu všech sousedů vrcholu u je $\Theta(deg(u))$ ($deg(u)$ je stupeň vrcholu u)
- časová složitost ověření zda $(u, v) \in E$ je $\mathcal{O}(deg(u))$

varianta použít místo pole hašovací tabulku, zřetězené hašování nahradit otevřenou adresací

Srovnání

	matice sousednosti	seznam následníků	hašovací tabulka
test $\{u, v\} \in E$	$\mathcal{O}(1)$	$\mathcal{O}(V)$	$\mathcal{O}(1)$
test $(u, v) \in E$	$\mathcal{O}(1)$	$\mathcal{O}(V)$	$\mathcal{O}(1)$
seznam sousedů vrcholu v	$\mathcal{O}(V)$	$\mathcal{O}(1 + deg(v))$	$\mathcal{O}(1 + deg(v))$
seznam hran	$\mathcal{O}(V^2)$	$\mathcal{O}(V + E)$	$\mathcal{O}(V + E)$
přidání hrany	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^*$
odstranění hrany	$\mathcal{O}(1)$	$\mathcal{O}(V)$	$\mathcal{O}(1)^*$

* *očekávaná složitost*

Průzkum grafu

Everything on earth can be found, if only you do not let yourself be put off searching. *Philemon of Syracuse (ca. 360 BC–264 BC)*

- pro daný graf G a vrchol s grafu je cílem
  - navštívit všechny vrcholy grafu dosažitelné z vrcholu s, resp.
  - navštívit všechny vrcholy grafu
- průzkum realizovat maximálně efektivně, tj. se složitostí $\mathcal{O}(V + E)$ (*vyhnout se opakovaným návštěvám*)
- jaké další informace o grafu zjistíme v průběhu průzkumu???

Průzkum grafu do šířky vs do hloubky

Theseus si před bludištěm uváže jeden konec nitě na strom a vsoupí dovnitř. V prvním vrcholu (křižovatce) si vybere jednu možnou cestu / hranu a projde po ní do dalšího vrcholu. Aby Theseus neměl zmatek v tom, které hrany už prošel, tak si všechny hrany, které prochází označuje křídou – a to na obou koncích. V každém vrcholu, do kterého Theseus dorazí, provede následující:

- *Pokud na zemi najde položenou niť, tak ví, že už ve vrcholu byl a že se do něj při namotávání nitě zase vrátí. Odloží tedy další prozkoumávání tohoto vrcholu na později, provede čelem vzad a začne namotávat niť na klubko. To ho dovede zpátky do předchozího vrcholu.*
- *Pokud na zemi žádnou niť nenajde, tak se vydá první možnou neprošlou hranou. Pokud by taková hrana neexistovala, tak je vrchol zcela prozkoumán. V tom případě Theseus neztrácí čas a začne namotávat niť na klubko. Tím se dostane zpátky do předchozího vrcholu.*

Tímto postupem prozkoumá celé bludiště a nakonec se vrátí do výchozího vrcholu.[6]

[6] Jakub Černý: Základní grafové algoritmy `http://kam.mff.cuni.cz/~kuba/ka`

Průzkum grafu do šířky vs do hloubky

implementace

křída proměnná označující jestli jsme hranu prošli

klubko položená niť vyznačuje cestu z výchozího do aktuálního vrcholu, cestu si pamatujeme jako posloupnost vrcholů na této cestě. Pro uložení cesty použijeme zásobník. Odmotávání nitě odpovídá přidání vrcholu do zásobníka. Namotávání nitě při návratu zpět odpovídá odebrání vrcholu ze zásobníku.

Průzkum grafu do šířky vs do hloubky

Tento průchod (prohledání grafu) si můžeme představit tak, že se do výchozího vrcholu postaví miliarda trpaslíků a všichni naráz začnou prohledávat graf. Když se cesta rozdělí, tak se rozdělí i dav řítící se hranou. Předpokládáme, že všechny hrany jsou stejně dlouhé. Graf prozkoumáváme „po vlnách“. V první vlně se všichni trpaslíci dostanou do vrcholů, dokterých vede z výchozího vrcholu hrana. V druhé vlně se dostanou do vrcholů, které jsou ve vzdálenosti 2 od výchozího vrcholu. Podobně v k-té vlně se všichni trpaslíci dostanou do vrcholů ve vzdálenosti k od výchozího vrcholu. Kvůli těmto vlnám se někdy průchodu do šířky říka algoritmus vlny. [7]

implementace

V počítači vlny nasimulujeme tak, že při vstupu do nového vrcholu uložíme všechny s ním sousedící vrcholy do fronty. Frontu průběžně zpracováváme.

[7] Jakub Černý: Základní grafové algoritmy `http://kam.mff.cuni.cz/~kuba/ka`

Průzkum grafu do šířky a do hloubky

průzkum grafu do **šířky** vs do **hloubky** se liší pouze použitím **fronty** a **zásobníku**

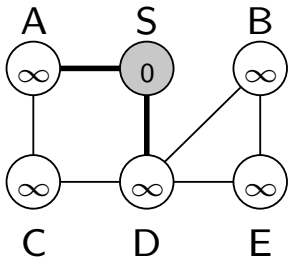
NE

Průzkum do šířky - strategie

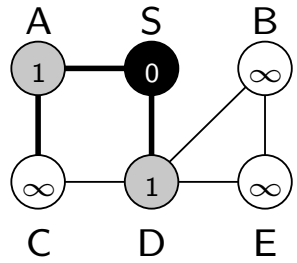
cílem je prozkoumat všechny vrcholy dosažitelné z daného iniciálního vrcholu s

- postupujeme od iniciálního vrcholu s po *vrstvách*
- nejdříve prozkoumáme všechny vrcholy dosažitelné z s po 1 hraně
- pak všechny vrcholy dosažitelné po 2 hranách, po 3 hranách atd.
- pro manipulaci s vrcholy používáme prioritní frontu Q
- $v \in Q$ právě když byl dosažen (objeven) ale ještě nebyl prozkoumán

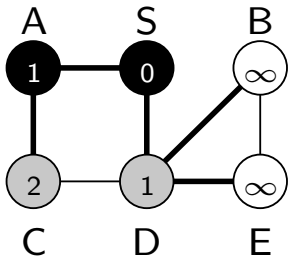
Průzkum do šířky - příklad



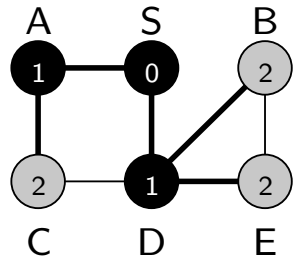
Q: S



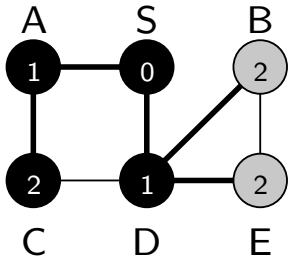
Q: AD



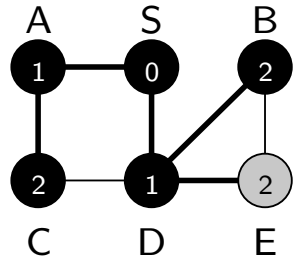
Q: DC



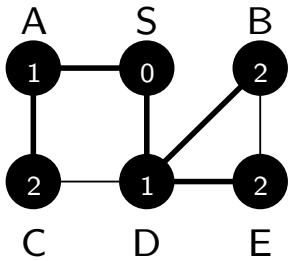
Q: CBE



Q: BE



Q: E



Q: ∅

Průzkum do šířky - atributy vrcholu

$v.color$

- v průběhu výpočtu je vrchol postupně objeven (je zařazen do fronty) a prozkoumán (všechny sousedící vrcholy jsou objeveny)
- vrchol má **černou** barvu právě když je dosažitelný z iniciálního vrcholu a byl již prozkoumán
- vrchol má **šedivou** barvu právě když je dosažitelný z iniciálního vrcholu, byl již objeven, ale nebyl ještě prozkoumán
- vrchol má **bílou** barvu právě když není dosažitelný z iniciálního vrcholu anebo ještě nebyl objeven

$v.\pi$

- vrchol, ze kterého byl v objeven

$v.d$

- délka (počet hran) cesty z s do v, na které byl v objeven
  ($=$ *délka nejkratší cesty z* s *do* v)

Průzkum do šířky - implementace

BFS(G, s)

```
1  foreach u ∈ V \ {s}
2     do u.color ← white;  u.d ← ∞;  u.π ← Nil od
3  s.color ← gray;  s.d ← 0;  s.π ← Nil
4  Q ← ∅
5  Enqueue(Q, s)
6  while Q ≠ ∅ do
7           u ← Dequeue(Q)
8           foreach v ∈ Adj[u] do
9              if v.color = white
10                then v.color ← gray
11                     v.d ← u.d + 1
12                     v.π ← u
13                     Enqueue(Q, v) fi
14             u.color ← black od
15 od
```

BFS - kompaktní verze

BFS(G, s)

```
1  foreach u ∈ V \ {s}
2     do u.d ← ∞ od
3  s.d ← 0
4  Q ← ∅
5  Enqueue(Q, s)
6  while Q ≠ ∅ do
7           u ← Dequeue(Q)
8           foreach v ∈ Adj[u] do
9              if v.d = ∞
10                then v.d ← u.d + 1
11                     Enqueue(Q, v) fi
12          od
13 od
```

BFS - složitost

- operace vložení a odstranění vrcholu z fronty mají konstantní složitost, každý vrchol je ve frontě maximálně jednou; celkově $\mathcal{O}(V)$
- seznam následníků každého vrcholu se prochází maximálně jednou; průzkum hrany má konstantní složitost; celkově $\mathcal{O}(E)$
- inicializace má složitost $\Theta(V)$
- celková složitost BFS je $\mathcal{O}(V + E)$

BFS a nejkratší cesty v neohodnoceném grafu

Nejkratší cesta v neohodnoceném grafu

Délka nejkratší cesty z s do v, značíme $\delta(s, v)$, je definována jako minimální počet hran na cestě z s do v. Když neexistuje žádná cesta z s do v, tak $\delta(s, v) = \infty$.

Nejkratší cestou z s do v je každá cesta z s do v která má $\delta(s, v)$ hran.

BFS a nejkratší cesty v neohodnoceném grafu

Věta 12

Nechť $G = (V, E)$ je graf a $s \in V$ jeho vrchol, na které aplikujeme algoritmus BFS. *Pak po ukončení výpočtu pro každý vrchol $v \in V$ platí*

$$v.d = \delta(s, v)$$

dokážeme indukcí podle hodnoty $v.d$

1. vrchol s má $s.d = 0$ a nejkratší cesta z s do s má nula hran
2. předpokládejme, že pro všechny vrcholy s hodnotou $v.d \leq k$ je $v.d$ délka nejkratší cesty do v
3. potřebujeme ukázat indukční krok a to je, že každý vrchol v s $v.d = k + 1$ leží ve vzdálenosti $k + 1$ od s
   - pokud ne, tak existuje kratší cesta z s do v a nechť (w, v) je poslední hrana na tého cestě
   - $w.d < k$
   - potom se ale měl algoritmus při zpracovávání vrcholu w podívat na hranu (w, v) a nastavit hodnotu $v.d$ na $w.d + 1$
   - $w.d + 1 < k + 1$, spor

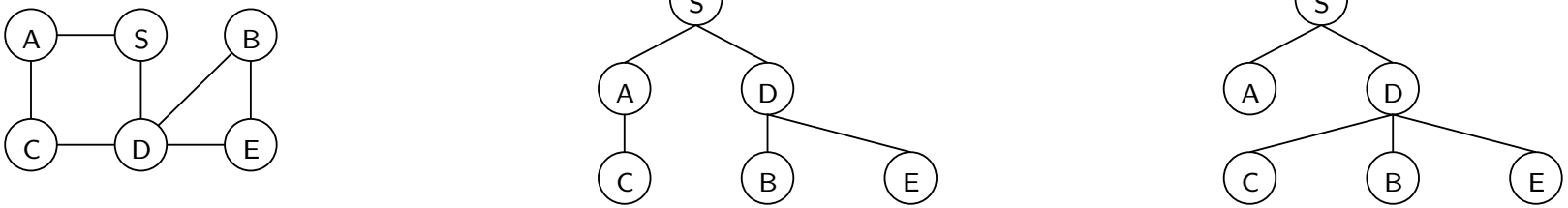
BFS strom a nejkratší cesty

- algoritmus BFS definuje přes atributy π **graf předchůdců**
- formálně: pro graf $G = (V, E)$ a iniciální vrchol s je graf předchůdců $G_\pi = (V_\pi, E_\pi)$ definovaný předpisem

$$V_\pi = \{v \in V \mid v.\pi \neq Nil\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$$

- graf předchůdců se nazývá **BFS strom**

- BFS strom je kostrou grafu
- pro každý vrchol $v \in V_\pi$ obsahuje BFS strom jedinou cestu z s do v, která je současně **nejkratší cestou z s do v**



graf a jeho dva různé BFS stromy

BFS a graf s ohodnocenými hranami

namísto fronty použijeme prioritní frontu

- do fronty vkládáme dvojici (vrchol; délka hrany, po které by objeven)
- prioritou je délka hrany
- z fronty vybíráme vždy nejkratší hranu
- **BFS strom je nejlevnější kostrou grafu**
- Primův algoritmus

- vrcholu ve frontě aktualizujeme hodnotu $v.d$ pokaždé, když je po nějaké hraně objeven
- prioritou je hodnota $v.d$
- z fronty vybíráme vždy vrchol s nejnižší hodnotou $v.d$
- **BFS strom je strom nejkratších cest z s do ostatních vrcholů grafu**
- Dijkstrův algoritmus

podrobnosti o obou algoritmech později

Aplikace a algoritmy využívající BFS

- Peer to Peer Networks
- Crawlers in Search Engines
- Social Networking Websites - hledání osob *ve vzdálenosti* nejvíce k
- GPS navigační systémy
- broadcasting
- garbage collection
- Fordův Fulkersonův algoritmus pro hledání maximálního toku v síti
- testování bipartitnosti

Bipartitní grafy

Definice 13

Neorientovaný graf se nazývá bipartitní právě když se jeho množina vrcholů dá rozdělit na dvě disjunktní množiny tak, že žádné dva vrcholy patřící do stejné množiny nejsou spojeny hranou.

alternativní formulace: vrcholy grafu je možné obarvit dvěma různými barvami tak, že každé dva vrcholy spojené hranou mají různou barvu

aplikace: vytváření dvojic, rozvrhu, ...

Testování bipartitnosti využitím BFS

Bipartitní graf neobsahuje cyklus liché délky.

- zvolíme libovolný vrchol grafu jako iniciální vrchol s
- BFS průzkum z vrcholu s definuje vrstvy $L_0, L_1, L_2, \ldots$
- do vrstvy L_i patří vrcholy, jejichž vzdálenost od s je i (t.j. $v.d = i$)

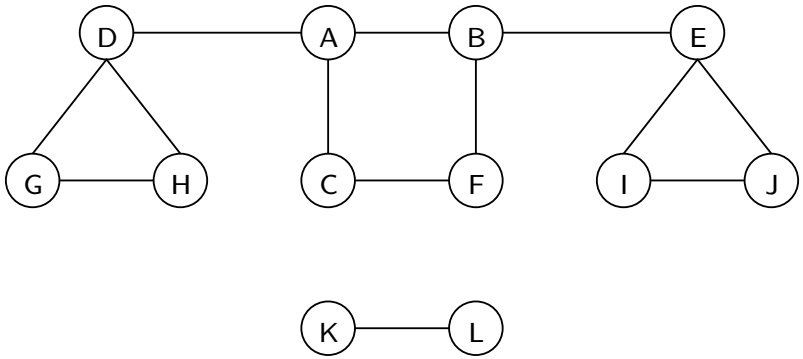
žádné dva vrcholy patřící do stejné vrstvy nejsou spojeny hranou

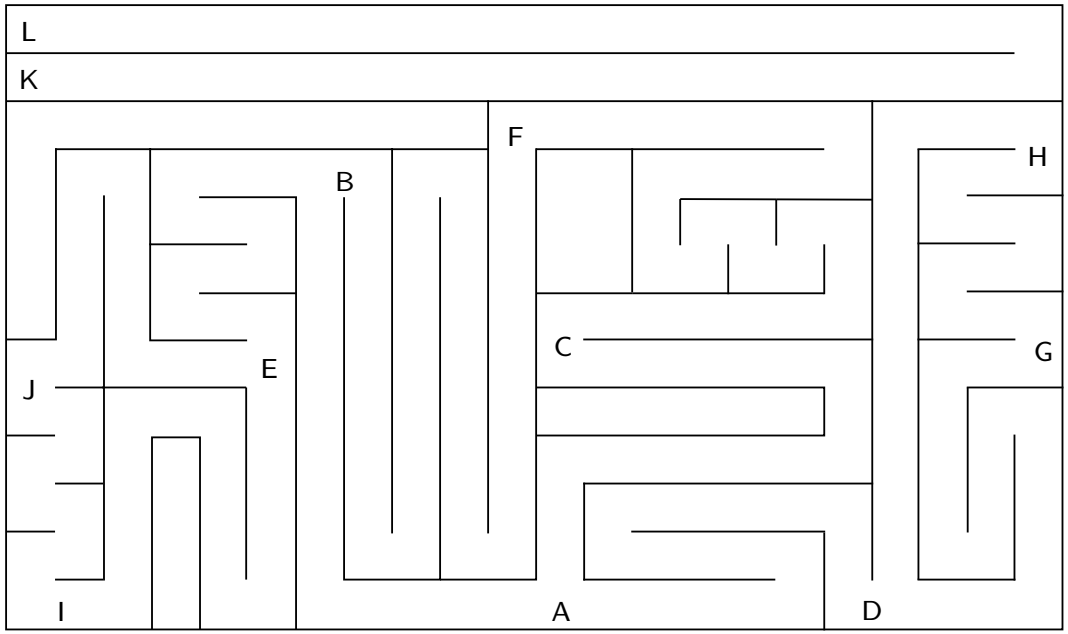
- obarvení vrcholů je určené vrstvami: vrcholy jejichž vzdálenost od s je **sudá** (**lichá**) mají **modrou** (**červenou**) barvu
- korektnost obarvení plyne z předpokladu o neexistenci hrany mezi vrcholy ze stejné vrstvy

existují dva vrcholy spojeny hranou a patřící do stejné vrstvy

- nechť u, v jsou vrcholy takové, že $u, v \in L_i$ a $\{u, v\} \in E$
- nechť y je nejmenší společný předchůdce vrcholů u a v v BFS stromu
- cesta z y do u, hrana $\{u, v\}$ a cesta z v do y tvoří cyklus, jehož délka je lichá (protože cesta z y do u a cesta z v do y mají stejnou délku)
- graf není bipartitní

Průzkum grafu do hloubky- motivace





pořadí, v němž BFS zkoumá vrcholy, netvoří souvislou cestu v grafu

Formulace problému

- průzkum do šířky a stejně tak i průzkum do hloubky je možné použít buď k prozkoumání té části grafu, která je dosažitelná z iniciálního vrcholu, anebo k prozkoumání celého grafu
- průzkum se dá aplikovat na orientované i neorientované grafy

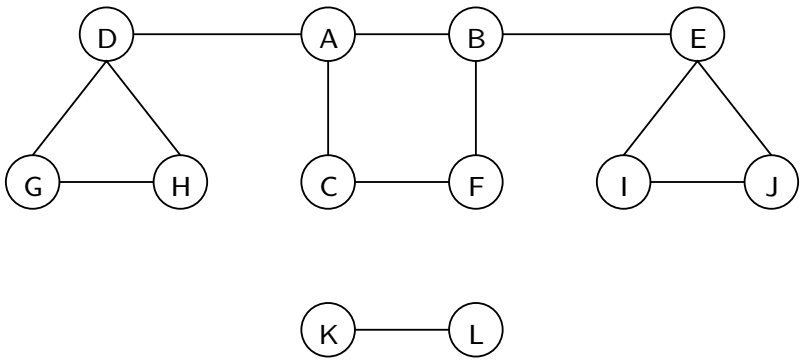
- prezentace průzkumu do hloubky předpokládá, že
  - vstupem je orientovaný graf a
  - cílem je prozkoumat celý graf

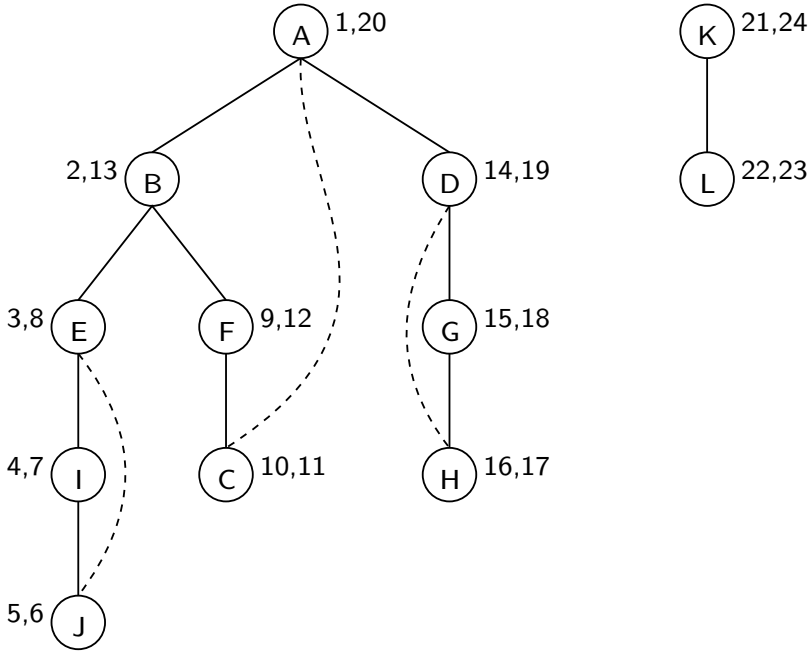
Průzkum do hloubky - strategie

- na začátku výpočtu a vždy po dokončení průzkumu vybereme jeden z dosud neprozkoumaných vrcholů a zvolíme ho za nový iniciální vrchol
- označ iniciální vrchol jako objevený
- vyber neprozkoumanou hranu (u, v), která vychází z naposledy objeveného vrcholu u, a když její koncový vrchol v ještě nebyl prozkoumán, tak ho označ jako objevený
- když všechny hrany vycházející z naposledy objeveného vrcholu u byly prozkoumány, tak ukonči průzkum vrcholu u a pokračuj vrcholem, ze kterého byl vrchol u objeven
- průzkum končí když jsou prozkoumány všechny vrcholy dosažitelné z iniciálního vrcholu

- pro manipulaci s vrcholy používáme zásobník

Průzkum grafu do hloubky - příklad





Průzkum do hloubky - atributy vrcholu

$v.color$

- vrchol má **černou** barvu právě když je dosažitelný z iniciálního vrcholu a byl již prozkoumán, tj. byly prozkoumány všechny hrany vycházející z vrcholu
- vrchol má **šedivou** barvu právě když je dosažitelný z iniciálního vrcholu, byl již objeven, ale nebyl ještě prozkoumán
- vrchol má **bílou** barvu právě když není dosažitelný z iniciálního vrcholu anebo ještě nebyl objeven

$v.\pi$

- vrchol, ze kterého byl v objeven

$v.d$

- časová značka, která zaznamenává čas první návštěvy vrcholu *(discovery time)*

$v.f$

- časová značka, která zaznamenává čas ukončení průzkumu vrcholu *(finishing time)*

Průzkum do hloubky - implementace

DFS(G)

```
1 foreach u ∈ V do u.color ← white;   u.π ← Nil od
2 time ← 0
3 foreach u ∈ V do
4     if u.color = white then DFS_Visit(G, u) fi od
```

DFS_Visit(G, u)

```
1 time ← time + 1
2 u.d ← time
3 u.color ← gray
4 foreach v ∈ Adj[u] do
5     if v.color = white then v.π ← u
6                             DFS_Visit(G, v) fi od
7 u.color ← black
8 time ← time + 1
9 u.f ← time
```

DFS - složitost

- oba cykly v DFS mají složitost $\Theta(V)$
- DFS_Visit se pro každý vrchol grafu volá jednou, protože bezprostředně po zavolání dostává vrchol šedivou barvu
- každá hrana se v cyklu procedury DFS_Visit prozkoumá právě jednou; ostatní operace mají konstantní složitost
- celková složitost DFS je $\mathcal{O}(V + E)$

Průzkum do hloubky - iterativní implementace

DFS_Iterative_Visit(G, u)

```
1  S ← ∅
2  S.push(u)
3  time ← time + 1; u.d ← time
4  u.color ← gray
5  while S ≠ ∅ do
6          u ← S.pop()
7          if  existuje hrana (u, v) taková, že v.color = white
8             then S.push(u)
9                  S.push(v)
10                 v.color ← gray
11                 v.π ← u
12                 time ← time + 1; v.d ← time
13            else u.color ← black
14                 time ← time + 1; u.f ← time fi od
```

DFS strom

- analogicky jako u BFS definují atributy $.\pi$ graf předchůdců
- protože prohledáváme celý graf, který nemusí být nutně souvislý, graf předchůdců je DFS les, který se skládá z DFS stromů
- $G_\pi = (V, E_\pi)$

  $E_\pi = \{(v.\pi, v) \mid v \in V \text{ a } v.\pi \neq Nil\}$

DFS - vlastnosti časových značek

časové značky, které DFS přiřadí vrcholům grafu, obsahují informace o struktuře grafu a DFS stromů

- pro každý vrchol u platí $u.d < u.f$
- s každým vrcholem u je asociovaný interval $[u.d, u.f]$

časové značky určují uspořádání vrcholů

preoder uspořádání podle značky $.d$ (discovery time) v rostoucím pořadí

postorder uspořádání podle značky $.f$ (finishing time) v rostoucím pořadí

reverse postorder uspořádání podle značky $.f$ (finishing time) v klesajícím pořadí

DFS - vlastnosti časových značek

podmínky správného uzávorkování

pro každé dva vrcholy u, v platí právě jedna z podmínek

- intervaly $[u.d, u.f]$ a $[v.d, v.f]$ jsou disjunktní
  u není následníkem v v DFS stromu a symetricky
  v není následníkem u v DFS stromu
- interval $[u.d, u.f]$ je celý obsažen v intervalu $[v.d, v.f]$
  u je následníkem v v DFS stromu
- interval $[v.d, v.f]$ je celý obsažen v intervalu $[u.d, u.f]$
  v je následníkem u v DFS stromu

DFS - vlastnosti časových značek

- vrchol v je dosažitelný z vrcholu u v DFS stromu grafu G právě když

$$u.d < v.d < v.f < u.f$$

- **vlastnost bílé cesty**
  v DFS stromu grafu G je vrchol v následníkem vrcholu u právě když v čase $u.d$ existuje cesta z u do v obsahující jenom bílé vrcholy

DFS - klasifikace hran

stromová hrana (*tree edge*) je hrana (u, v) obsažená v DFS lese
při průzkumu hrany je vrchol v bílý
$u.d < v.d < v.f < u.f$

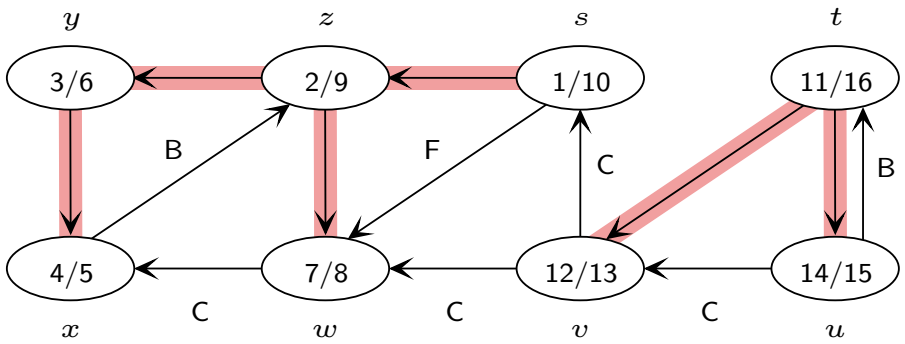
zpětná hrana (*back edge*) je hrana (u, v) která spojuje vrchol u s jeho předchůdcem v v DFS stromu
při průzkumu hrany je vrchol v šedivý
$v.d < u.d < u.f < v.f$

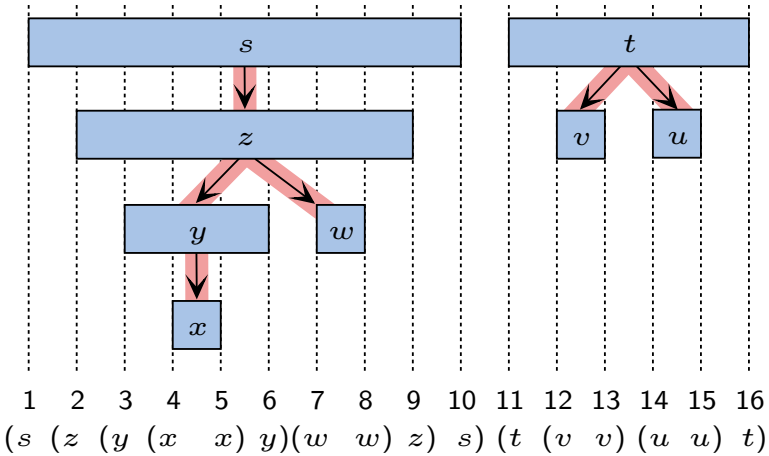
dopředná hrana (*forward edge*) je hrana (u, v), která nepatří do DFS stromu a která spojuje vrchol u s jeho následníkem v DFS stromu
při průzkumu hrany je vrchol v černý
$u.d < v.d < v.f < u.f$

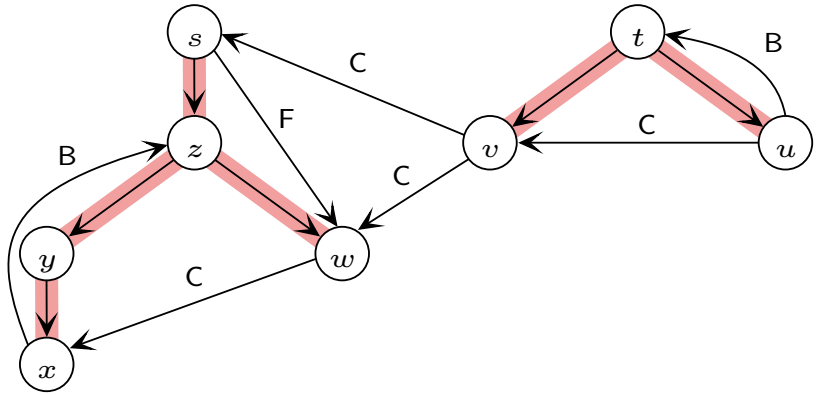
příčná hrana (*cross edge*) všechny ostatní hrany
při průzkumu hrany je vrchol v černý
$v.d < v.f < u.d < u.f$

všechny hrany v neorientovaném grafu jsou buď stromové anebo zpětné

Časové značky a klasifikace hran - příklad







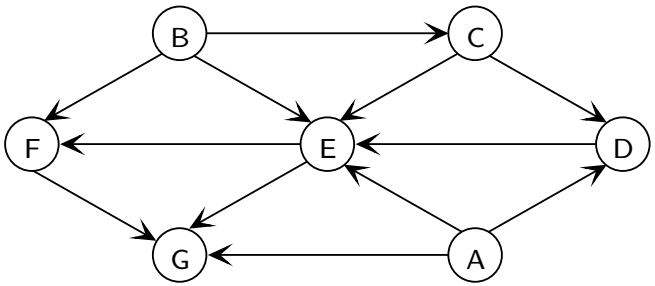
stromové hrany jsou zvýrazněné, zpětné hrany jsou označeny písmenem B, dopředné písmenem F a příčné písmenem C

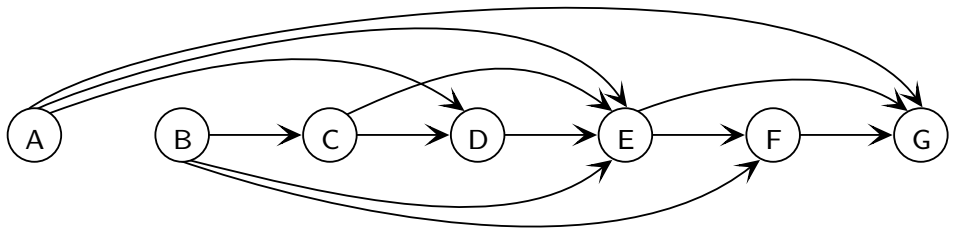
Aplikace a algoritmy využívající DFS

- topologické uspořádání
- komponenty souvislosti
- artikulace a mosty
- testování planarity
- hledání cesty v bludišti
- generování bludiště

Topologické uspořádání vrcholů grafu

Topologické uspořádání vrcholů v orientovaném acyklickém grafu je takové očíslování vrcholů čísly 1 až n (n je počet vrcholů grafu), že každá hrana vede z vrcholu s nižším číslem do vrcholu s vyšším číslem.





$A < B < C < D < E < F < G$

aplikace: modelování procesů, které jsou částečně uspořádané

problém

- existuje v daném grafu topologické uspořádání?
- když ano, tak nalezení takového uspořádání

Topologické uspořádaní - podmínky

Lema 14

Orientovaný graf G má topologické uspořádání právě když je acyklický.

Důkaz

$\Rightarrow$ z definice

$\Leftarrow$ důkaz indukcí k počtu vrcholů grafu

- tvrzení platí pro $n = 1$
- v orientovaném grafu G s $n > 1$ vrcholy najdi vrchol v, do kterého nevstupuje žádná hrana (*kdyby takový neexistoval, tak bychom mohli z libovolného vrcholu jít donekonečna „pozpátku“ a našli bychom cyklus*)
- $G - v$ je acyklický (*odstranění vrcholu nemůže způsobit vznik cyklu*)
- z indukčního předpokladu $G - v$ má topologické uspořádání
- topologické uspořádání pro G má na prvním místě v následované uspořádáním pro $G - v$

Topologické uspořádání - naivní algoritmus

Topological_Sort_Visit(G)

```
1 n ← |V|
2 for i = 1 to n do
3     v ← libovolný vrchol, do kterého nevstupuje žádná hrana
4     S[i] ← v
5     odstraň z G vrchol v a všechny jeho hrany
6 od
7 return S[1...n]
```

- algoritmus pro acyklický graf
- nalezení vrcholu do kterého nevstupuje žádná hrana má složitost ???
- celková složitost algoritmu je ???

- existuje efektivnější algoritmus (ideálně s lineární složitostí)???

- *symetricky se dá postupovat podle vrcholů z nichž nevychází žádná hrana*

Acyklický graf - testování

Lema 15

Orientovaný graf G je acyklický právě když DFS průzkum grafu neoznačí žádnou hranu jako zpětnou.

Důkaz

$\Rightarrow$ zpětná hrana (u, v) spojuje vrchol u s jeho předchůdcem v v DFS stromu, tj. uzavírá cyklus

$\Leftarrow$

- nechť v grafu existuje cyklus c, nechť v je první vrchol cyklu c navštívený při DFS průzkumu grafu a nechť (u, v) je hrana cyklu c
- v čase $v.d$ vrcholy cesty c tvoří bílou cestu z v do u co implikuje, že u je následníkem v v DFS stromu
- (u, v) je zpětná hrana

Topologické uspořádání - algoritmus

1. aplikuj DFS na G
2. když průzkum označí některou hranu jako zpětnou, tak graf nemá topologické uspořádání
3. v opačném případě vypiš vrcholy v uspořádání *reverse postorder*, tj. podle značky $.f$ (finishing time) v klesajícím pořadí

Korektnost

potřebujeme dokázat, že pro libovolnou dvojici vrcholů u, v platí

jestliže G obsahuje hranu (u, v), tak $u.f > v.f$

jaké jsou barvy vrcholů u a v při průzkumu hrany (u, v)?

- u je šedivý
- v je

  šedivý nemůže nastat, protože (u, v) by v takovém případě byla zpětnou hranou a graf by nebyl acyklický

  bílý v takovém případě je (u, v) stromová hrana, v je následníkem u v DFS stromu a $u.d < v.d < v.f < u.f$

  černý v takovém případě je průzkum vrcholu v ukončený a průzkum vrcholu u ještě není ukončený a proto $v.f < u.f$

Souvislost v orientovaném grafu

orientovaný graf $G = (V, E)$

- **vrchol v je dosažitelný z vrcholu u**, značíme $u \rightsquigarrow v$, právě když v G existuje orientovaná cesta z u do v
- $Reach(u)$ je množina **všech** vrcholů dosažitelných z u
- vrcholy u a v jsou **silně dosažitelné** (*strongly connected*) právě když u je dosažitelný z v a současně v je dosažitelný z u
- silná dosažitelnost - relace ekvivalence
- **silně souvislá komponenta** grafu je třída ekvivalence relace silné dosažitelnosti, tj. maximální množina vrcholů $C \subseteq V$ taková, že pro každé $u, v \in C$ platí $u \rightsquigarrow v$ a současně $v \rightsquigarrow u$
- graf nazýváme silně souvislý právě když má jedinou silně souvislou komponentu

problém

najít všechny silně souvislé komponenty grafu

Souvislost v neorientovaném grafu

- v neorientovaném grafu jsou pojmy dosažitelnosti a silné dosažitelnost totožné
- pro hledání silně souvislé komponenty grafu můžeme použít BFS nebo DFS
- jednotlivé DFS (BFS) stromy korespondují s komponentami souvislosti
- složitost $\mathcal{O}(V + E)$

Silně souvislé komponenty v orientovaném grafu

výpočet silně souvislé komponenty obsahující daný vrchol u

- najdi množinu $Reach(u)$ všech vrcholů dosažitelných z u aplikací $\text{DFS\_VISIT}(G, u)$
- najdi množinu $Reach^{-1}(u)$ všech vrcholů, ze kterých je dosažitelný u
- pro výpočet $Reach^{-1}(u)$ využijeme transponovaný graf[8] $rev(G)$, na který aplikujeme $\text{DFS\_VISIT}(rev(G), u)$
- silně souvislá komponenta obsahující u je průnikem $Reach(u) \cap Reach^{-1}(u)$
- časová složitost výpočtu je $\mathcal{O}(V + E)$

[8] transponovaný graf $rev(G) = (V, rev(E))$ je graf G s obrácenou orientací hran, $rev(E) = \{(x, y) \mid (y, x) \in E\}$

Silně souvislé komponenty v orientovaném grafu

výpočet všech silně souvislých komponent grafu

- můžeme *zabalit* popsaný postup (podobně jako je $\mathrm{DFS\_VISIT}(G, u)$ *zabalené* do DFS)
- v nejhorším případě má graf $|V|$ komponent souvislosti a detekce každé z nich má časovou složitost $\mathcal{O}(E)$
- celková časová složitost výpočtu je $\mathcal{O}(V \cdot E)$

- existuje efektivnější algoritmus, optimálně s lineární časovou složitostí ???
- motivace: v čase $\mathcal{O}(V + E)$ dokážeme rozhodnout, zda všechny silně souvislé komponenty grafu jsou jednoduché (obsahují pouze jeden vrchol)

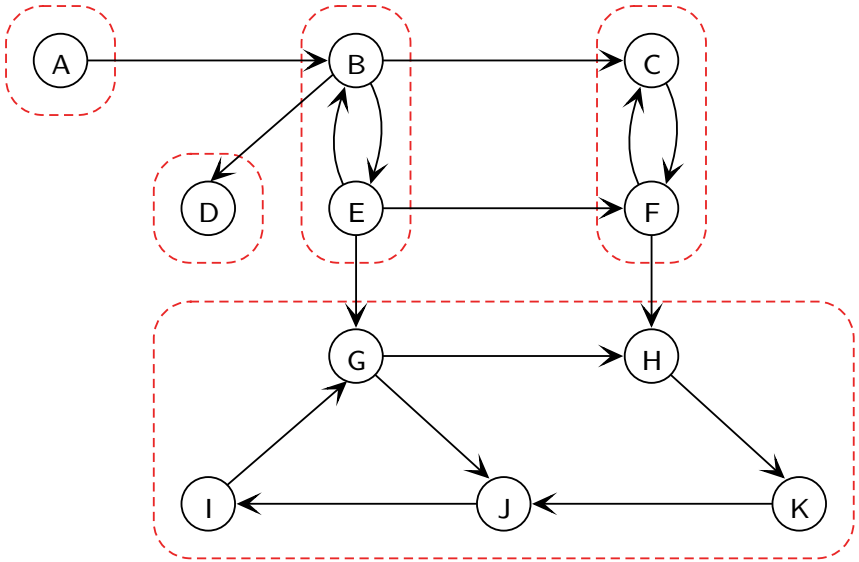
Komponentový graf

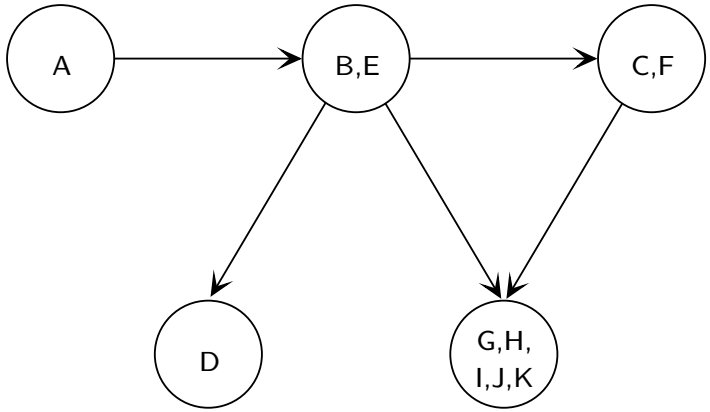
- komponentový graf (*aka graf silně souvislých komponent*) je orientovaný graf, který vznikne kontrakcí každé silně souvislé komponenty grafu do jednoho vrcholu a kontrakcí paralelních hran do jedné hrany, značíme $scc(G)$
- $scc(G)$ je orientovaný acyklický graf

- vrcholy $scc(G)$ můžeme topologicky uspořádat
- listová komponenta == list grafu $scc(G)$
- kořenová komponenta == kořen grafu $scc(G)$

- platí $rev(scc(G)) = scc(rev(G))$

Příklad





kořenová komponenta A
listové komponenty D, CF, GHIJK

Komponentový graf

- **listová komponenta grafu = list grafu** $scc(G)$
- z každého vrcholu u listové komponenty C jsou dosažitelné všechny vrcholy C, tj. DFS průzkum z u navštíví všechny vrcholy z C
- naopak, z vrcholu u není dosažitelný žádný vrchol nepatřící do C (C je listová komponenta!!), tj . DFS průzkum z u navštíví pouze všechny vrcholy z C
- na tomto pozorování můžeme postavit algoritmus pro detekci komponent

Strong_Components(G)

```
1 count ← 0
2 while G není prázdný do
3         count ← count + 1
4         v ← libovolný vrchol z listové komponenty
5         C ← One_Component(v, count)
6         odstraň z grafu G vrcholy z C a hrany vstupující do C od
```

potřebujeme (efektivně) najít vrchol patřící listové komponentě

Komponentový graf

- **potřebujeme** (efektivně) najít vrchol patřící listové komponentě
- **umíme** (efektivně) najít vrchol patřící kořenové komponentě

Věta 16

Vrchol s nejvyšší časovou značkou $.f$ leží v kořenové komponentě.

Důkaz tvrzení je důsledkem následujícího obecnějšího tvrzení

Komponentový graf

Věta 17

Nechť C_1 a C_2 jsou silně souvislé komponenty takové, že z C_1 vede hrana do C_2. Potom největší hodnota $.f$ v komponentě C_1 je větší než největší hodnota $.f$ v komponentě C_2.

Důkaz

mohou nastat dva případy

- v prvním případě navštíví DFS komponentu C_2 jako první; pak ale DFS zůstane v komponentě C_2 dokud ji celou neprozkoumá, teprve pak se dostane do C_1
- v druhém případě navštíví DFS komponentu C_1 jako první; nechť v je vrchol, který byl v C_1 navštíven jako první; DFS opustí vrchol v, až když prozkoumá všechny vrcholy, které jsou z v dosažitelné a které dosud nebyly navštíveny; proto nejprve projde celou komponentu C_2 a pad se teprve naposledy vrátí do v

Algoritmus pro detekci silně souvislých komponent v lineárním čase

- Robert Tarjan, 1972
- Harold Gabow, 2000
- Rao Kosaraju, 1978; Micha Sharir, 1981
  - vrcholy grafu uspořádej v obráceném postorder pořadí, tj. podle značky $.f$ v klesajícím pořadí
  - proveď DFS průzkum z vrcholů v daném pořadí

Kosaraju_Sharir(G)

```
1 foreach u ∈ V do u.color ← white od
2 foreach u ∈ V do if u.color = white then PUSH_DFS(G, u) fi od
3 foreach u ∈ V do u.color ← white od
4 count ← 0
5 while S ≠ ∅ do
6         u ← S.pop()
7         if u.color = white then count ← count + 1
8                                 LABEL_ONE_DFS(Rev(G), u, count) fi od
```

Push_DFS(G, u)

```
1 u.color ← gray
2 foreach v ∈ Adj[u] do
3    if v.color = white then PUSH_DFS(G, v) fi od
4 u.color ← black;    S.push(u)
```

Label_DFS$(G, u, count)$

```
1 u.color ← gray
2 foreach v ∈ Adj[u] do
3    if v.color = white then LABEL_DFS(G, v, count) fi od
4 u.color ← black;    u.label ← count
```

Grafové algoritmy

12 Průzkum grafů a grafová souvislost
- Průzkum do šířky
- Průzkum do hloubky
- Topologické uspořádání
- Silně souvislé komponenty

13 Nejkratší cesty
- Algoritmus Bellmana a Forda
- Acyklické grafy
- Dijkstrův algoritmus
- Lineární nerovnice

Cesta v grafu

cesta v grafu $G = (V, E)$ je posloupnost vrcholů $p = \langle v_0, v_1, \ldots, v_k \rangle$ taková, že $(v_{i-1}, v_i) \in E$ pro $i = 1, \ldots, k$

jednoduchá cesta je cesta, která neobsahuje dva stejné vrcholy

terminologie

cesta	jednoduchá cesta
path	simple path
walk	path
sled	cesta

$p = \langle v_0, v_1, \ldots, v_k \rangle$ je cestou z vrcholu u do vrcholu v právě když $v_0 = u$ a $v_k = v$

v je dosažitelný z u, $u \rightsquigarrow v$

Délka cesty

graf $G = (V, E)$, váhová funkce (*ohodnocení, délka hran*) $w : E \to \mathbb{R}$

délka cesty $p = \langle v_0, v_1, \ldots, v_k \rangle$ je součet délek hran cesty

$$w(p) \stackrel{\text{def}}{=} \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

délka nejkratší cesty z vrcholu u do vrcholu v je definovaná předpisem

$$\delta(u, v) \stackrel{\text{def}}{=} \begin{cases} \min\{w(p) \mid u \overset{p}{\rightsquigarrow} v\} & \text{existuje cesta z } u \text{ do } v \\ \infty & \text{jinak} \end{cases}$$

nejkratší cesta z vrcholu u do vrcholu v je libovolná cesta p z u do v pro kterou $w(p) = \delta(u, v)$

pro neohodnocené grafy se délka cesty definuje jako počet hran cesty

Varianty problému nejkratší cesty

nejkratší cesty z daného vrcholu do všech vrcholů

single source shortest path, **SSSP**

tato přednáška

nejkratší cesty ze všech vrcholů do daného vrcholu

pro neorientované grafy totožné s SSSP

pro orientované grafy redukce na SSSP změnou orientace hran

nejkratší cesta mezi danou dvojicí vrcholů speciální případ SSSP, nejsou známy asymptoticky rychlejší algoritmy než pro SSSP

tato přednáška

nejkratší cesty mezi všemi dvojicemi vrcholů

řešení opakovanou aplikací algoritmu pro SSSP

existují efektivnější algoritmy

ADS II

nejdelší, nejširší, nejspolehlivější . . . cesty

viz literatura

Algoritmy pro SSSP

orientované grafy

neohodnocený graf

průzkum do šířky, BFS

acyklický graf

průzkum do hloubky

tato přednáška

graf s nezáporným ohodnocením hran

Dijkstrův algoritmus a jiné

tato přednáška

obecný graf

algoritmus Bellmana Forda a jiné

tato přednáška

Algoritmy pro SSSP

neorientované grafy

neohodnocený graf

- průzkum do šířky, BFS

graf s nezáporným ohodnocením hran

- hranu nahradíme dvojicí orientovaný hran a převedeme na úlohu v orientovaném grafu

obecný graf

- nahrazením hrany se záporným ohodnocením dvojicí orientovaných hran vznikne cyklus záporné délky
- pokud původní graf obsahuje hrany záporné délky, ale žádný cyklus záporné délky, lze takovou úlohu převést na hledání nejlevnějšího perfektního párování
- když obsahuje cyklus záporné délky, problém je NP-těžký a umíme ho řešit pouze algoritmy exponenciální složitosti
- viz literatura

Nejkratší cesta vs nejkratší jednoduchá cesta

graf neobsahuje hrany záporné délky

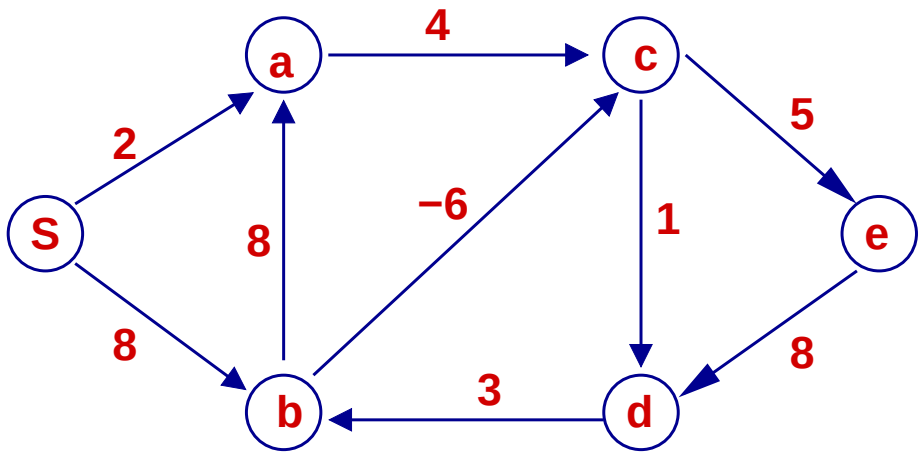
jestliže mezi dvojicí vrcholů existuje cesta,
tak mezi nima existuje taková nejkratší cesta, která je jednoduchá

nechť p je nejkratší cesta, která není jednoduchá, tj. obsahuje cyklus

- cyklus nemůže mít zápornou délku (*spor s předpokladem neexistence hran záporné délky*)
- cyklus nemůže mít kladnou délku (*spor s předpokladem, že cesta je nejkratší*)
- cyklus má nulovou délku - cyklus můžeme z cesty vypustit a dostaneme jednoduchou nejkratší cestu

Nejkratší cesta vs nejkratší jednoduchá cesta

graf obsahuje hrany záporné délky



- cyklus $\langle b, c, d, b \rangle$ má délku -2
- jestliže nějaká cesta z x do y obsahuje cyklus záporné délky, tak žádná cesta z x do y nemůže být nejkratší cestou, $\delta(x, y) = -\infty$

v případě, že graf obsahuje hrany se zápornou délkou, problém nejkratší cesty je formulovaný jako

1. rozhodni, zda graf obsahuje cyklus záporné délky
2. když ne, tak najdi nejkratší (jednoduché) cesty

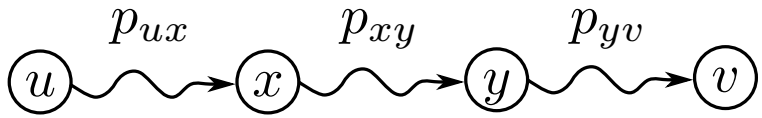
Struktura nejkratších cest

Lemma 18

Každá podcesta nejkratší cesty je nejkratší cestou.

- nechť p je nejkratší cesta z u do v

  $w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$



- předpokládejme, že existuje kratší cesta z x do y, $w(p'_{xy}) < w(p_{xy})$
- zkonstruujeme novou cestu $p' = u \overset{p_{ux}}{\rightsquigarrow} x \overset{p'_{xy}}{\rightsquigarrow} y \overset{p_{yv}}{\rightsquigarrow} v$

$$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p) \end{aligned}$$

  což je spor s předpokladem, že p je nejkratší cesta z u do v

Reprezentace nejkratších cest

strom nejkratších cest

pozor na rozdíl mezi stromem nejkratších cest a nejlevnější kostrou grafu

Reprezentace stromu nejkratších cest z vrcholu s

atribut vzdálenost *distance*, $v.d$

- iniciální nastavení $v.d = \infty$
- hodnota $v.d$ se v průběhu výpočtu snižuje
- hodnota $v.d$ je horním odhadem délky nejkratší cesty, $v.d \geq \delta(s, v)$
- na konci výpočtu je $v.d = \delta(s, v)$

atribut předchůdce *parent*, $v.\pi$

- iniciální nastavení $v.\pi = Nil$
- vrchol $v.\pi$ je předchůdcem vrcholu v na cestě z s do v délky $v.d$
- na konci výpočtu je $v.\pi$ předchůdce vrcholu v na nejkratší cestě z s do v, resp. $v.\pi = Nil$ když neexistuje cesta z s do v

graf předchůdců $G_p = (V_p, E_p)$ je definovaný hodnotami $.\pi$

$$V_p = \{v \in V \mid v.\pi \neq Nil\} \cup \{s\}$$

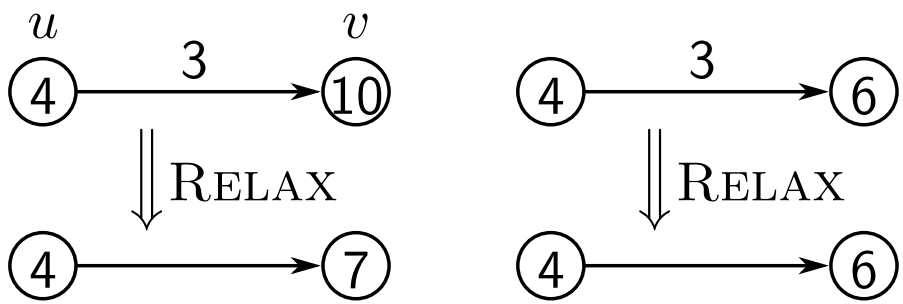
$$E_p = \{(v.\pi, v) \mid v \in V_p \setminus \{s\}\}$$

strom nejkratších cest na konci výpočtu je G_p stromem nejkratších cest

- s je kořen stromu, V_p je množina vrcholů dosažitelných z vrcholu s
- pro každý vrchol $v \in V_p$, (jediná) cesta z s do v v G_p je nejkratší cestou z s do v v G

Relaxace

- technika, kterou využívají algoritmy pro hledání nejkratších cest
- relaxací hrany (u, v) rozumíme test, zda je možné zkonstruovat kratší cestu do v tak, že projedeme přes vrchol u
- když aktuálně známá cesta do vrcholu u prodloužená o hranu (u, v) je kratší než aktuálně známá cesta do v ($u.d + w(u, v) < v.d$), tak jsme našli novou - kratší cestu do v a podle toho aktualizujeme hodnoty $v.d$ a $v.\pi$ ($v.d \leftarrow u.d + w(u, v)$ a $v.\pi \leftarrow u$)



- hranu (u, v) nazýváme napjatou právě když $u.d + w(u, v) < v.d$

Generický SSSP algoritmus

Init_SSSP(G, s)

```
1 foreach v ∈ V do v.d ← ∞; v.π ← Nil od
2 s.d ← 0
```

Relax(u, v, w)

```
1 v.d ← u.d + w(u, v)
2 v.π ← u
```

Generic_SSSP(G, w, s)

```
1 INIT_SSSP(G, s)
2 S ← s
3 while S ≠ ∅ do
4         vyber u z S
5         foreach  hranu (u, v) ∈ E do
6           if v.d > u.d + w(u, v)
7             then RELAX(u, v, w)
8                   S ← S ∪ {v} fi od
9 od
```

Korektnost generického SSSP algoritmu

- pro každý vrchol v platí, že hodnota $v.d$ je buď ∞, anebo je rovna délce nějaké cesty z s do v
  důkaz indukcí na počet relaxací
- když graf neobsahuje cyklus záporné délky, tak hodnota $v.d$ je buď ∞, anebo je rovna délce nějaké jednoduché cesty z s do v
  důkaz viz diskuse nejkratší cesta vs nejkratší jednoduchá cesta

  důsledek: **generický algoritmus pro graf bez záporných cyklů skončí, protože v grafu existuje jenom konečný počet jednoduchých cest**

- když žádná hrana grafu není napjatá, tak hodnota $v.d$ je rovna délce cesty $s \to \cdots \to v.\pi.\pi \to v.\pi \to v$
- když žádná hrana grafu není napjatá, tak cesta $s \to \cdots \to v.\pi.\pi \to v.\pi \to v$ je nejkratší cestou z s do v
  důkaz indukcí na počet relaxací

  důsledek: **když výpočet generického algoritmu skončí, tak G_p je strom nejkratších cest**

Složitost generického SSSP algoritmu

závisí od toho

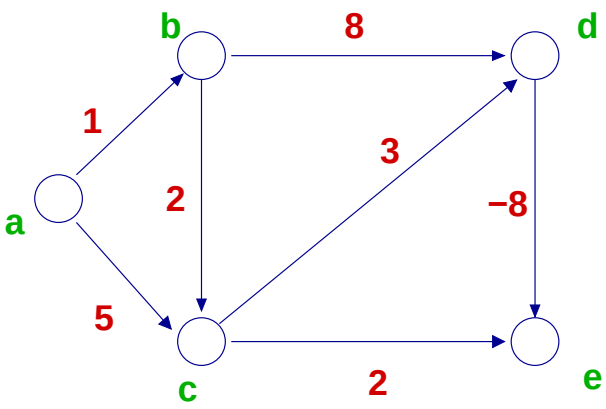
- jakou datovou strukturu použijeme pro reprezentaci množiny S obsahující vrcholy určené k prozkoumání
  (tj. vrcholy, u kterých byla změněna hodnota $.d$)
- v jakém pořadí budeme prozkoumávat vrcholy z množiny S

Bellmanův - Fordův algoritmus

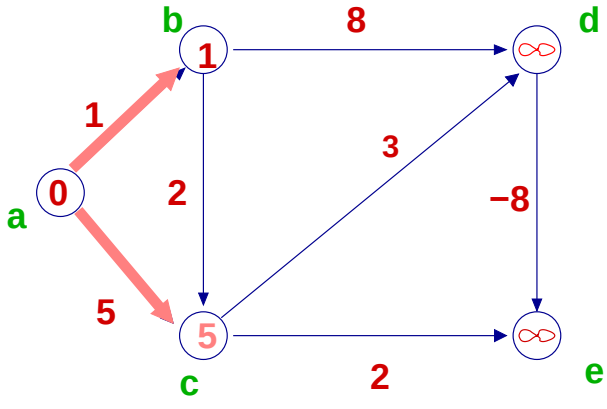
Bellman-Ford(G, w, s)

```
1  INIT_SSSP(G, s)
2  for i = 1 to |V| − 1 do
3      foreach (u, v) ∈ E do
4          if v.d > u.d + w(u, v) then RELAX(u, v, w) fi
5      od
6  od
7  foreach (u, v) ∈ E do
8      if v.d > u.d + w(u, v) then return FALSE fi
9  od
10 return TRUE
```

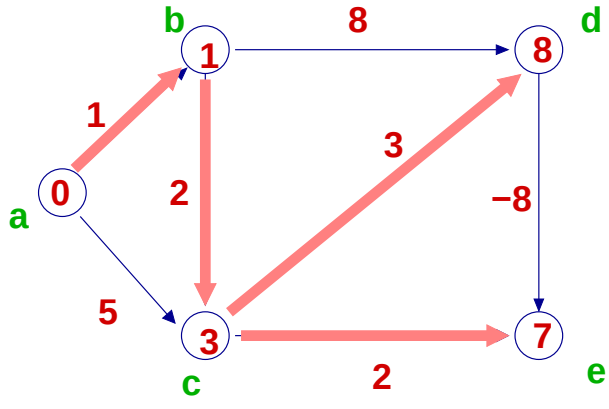
optimalizace: výpočet můžeme ukončit, jestliže v iteraci **for** *cyklu v řádcích 2 - 6 nebyla nalezena žádná napjatá hrana*



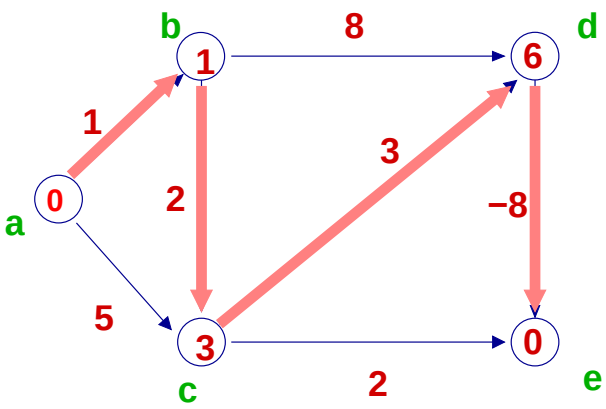
(a)



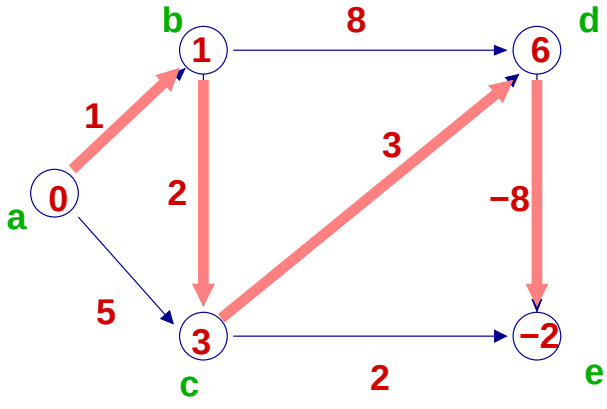
(b)



(c)



(d)



(e)

graf G_p

výpočet algoritmu BELLMAN-FORD, hledáme nejkratší cesty z vrcholu a

v každé iteraci cyklu 2 - 6 relaxujeme hrany v pořadí (c, e), (d, e), (b, d), (c, d), (b, c), (a, c), (a, b)

barevně jsou vyznačeny hrany grafu předchůdců G_p

Korektnost 1

Lema 19

Když graf G neobsahuje cyklus záporné délky dosažitelný z s, tak po $|V| - 1$ iteracích cyklu 3 - 5 pro všechny vrcholy v platí

$$v.d = \delta(s, v).$$

- nechť v je dosažitelný z s a nechť $p = \langle v_0, v_1, \ldots, v_k \rangle$ je nejkratší jednoduchá cesta z s do v ($v_0 = s$, $v_k = v$)
- protože cesta p neobsahuje cyklus, má nanejvýš $|V| - 1$ hran, tj. $k \leq |V| - 1$
- v každé iteraci cyklu se relaxují všechny hrany grafu, speciálně
  - v první iterace se relaxuje hrana (v_0, v_1)
  - v druhé iteraci se relaxuje hrana (v_1, v_2)
  - ...
  - v k-té iteraci se relaxuje hrana (v_{k-1}, v_k)
- indukcí ověříme, že po i-té iteraci platí $v_i.d = \delta(s, v_i)$

Korektnost 2

Lema 20

Když graf G neobsahuje cyklus záporné délky dosažitelný z s, tak algoritmus vrátí hodnotu TRUE.

- po ukončení výpočtu platí pro každou hranu $(u, v) \in E$

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= u.d + w(u, v) \end{aligned}$$

- žádná hrana není napjatá a test na řádku 8 nevrátí hodnotu FALSE

Korektnost 3

Lema 21

Když graf G obsahuje cyklus záporné délky dosažitelný z s, tak algoritmus vrátí hodnotu FALSE.

- nechť $c = \langle v_0, v_1, \ldots, v_k \rangle$, $v_0 = v_k$ je cyklus záporné délky dosažitelný z s, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$
- předpokládejme, že algoritmus vrátí hodnotu TRUE, tj. pro každou hranu cyklu platí $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$
- sumací přes všechny vrcholy cyklu

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- každý vrchol se vyskytuje právě jednou v součtech $\sum_{i=1}^{k} v_{i-1}.d$ a $\sum_{i=1}^{k} v_i.d$
- $$\sum_{i=1}^{k} v_{i-1}.d = \sum_{i=1}^{k} v_i.d \quad \Longrightarrow \quad 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$
- spor s předpokladem o délce cyklu c

Složitost

složitost algoritmu Bellmana a Forda

- inicializace má složitost $\Theta(V)$
- relaxace hrany má konstantní složitost
- cyklus 3 - 5 má složitost $\Theta(E)$; počet jeho opakování je $V - 1$
- celková složitost je $\mathcal{O}(VE)$
  jiný zápis: $\mathcal{O}(mn)$, kde n je počet vrcholů a m je počet hran grafu

existuje efektivnější řešení?

ne lehce najdeme graf a takové pořadí relaxace jeho hran, pro které je nutných $V - 1$ iterací

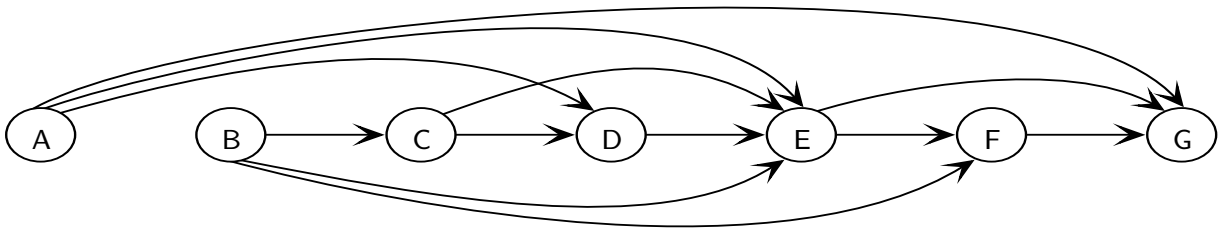
??? otázka vhodného pořadí relaxace hran

ano vhodné pořadí hran dokážeme určit pro speciální typy grafů

- acyklické grafy
- grafy bez záporných hran

Nejkratší cesty v orientovaném acyklickém grafu

- optimálně pořadí relaxace hran v Bellmanově - Fordově algoritmu je takové, že vždy relaxujeme hranu (u, v) pro kterou $u.d = \delta(s, u)$
- pro obecný graf určit pořadí relaxací tak, aby byla dodržena uvedená podmínka, může být stejně náročné jako vypočíst nejkratší cesty
- speciálně pro acyklické grafy se toto pořadí dá vypočíst jednoduše: požadovanou vlastnost má topologické uspořádání vrcholů grafu



A < B < C < D < E < F < G

Nejkratší cesty v orientovaném acyklickém grafu

Dag(G, w, s)

```
1 najdi topologické uspořádání vrcholů grafu G
2 Init_SSSP(G, s)
3 foreach vrchol u v topologickém uspořádání do
4   foreach (u, v) ∈ E do
5     if v.d > u.d + w(u, v) then Relax(u, v, w) fi
6   od
7 od
```

- časová složitost $\Theta(V + E)$
- topologické uspořádání garantuje, že hrany *každé* cesty jsou relaxované v pořadí, v jakém se vyskytují na cestě

Dijkstrův algoritmus

- pro reprezentaci množiny vrcholů určených k prozkoumání využívá **prioritní frontu**, kde priorita vrcholu v je určena hodnotou $v.d$
- *Dijkstrův algoritmus můžeme nahlížet i jako efektivní implementaci prohledávání grafu do šířky, na rozdíl od BFS neukládáme vrcholy, které mají být prozkoumané, do fronty, ale do prioritní fronty*

- řeší problém SSSP pro grafy **s nezáporným ohodnocením hran**

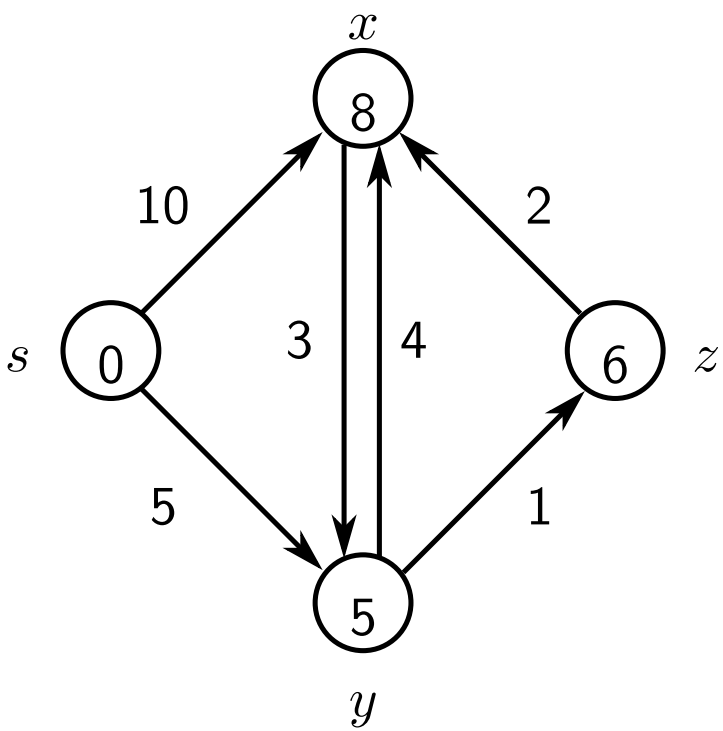
Dijkstrův algoritmus

Dijkstra(G, w, s)

```
1  INIT_SSSP(G, s)
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅ do
5        u ← EXTRACT_MIN(Q)
6        S ← S ∪ {u}
7        foreach (u, v) ∈ E do
8           if v.d > u.d + w(u, v) then RELAX(u, v, w) fi
9        od
10 od
```

- algoritmus udržuje dvě množiny
  - S vrcholy, pro které je již vypočtena délka nejkratší cesty
  - Q prioritní fronta, $Q = V \setminus S$
- algoritmus vybírá vrchol $u \in Q$ s nejmenší hodnotou $u.d$ a relaxuje hrany vycházející z u

Dijkstrův algoritmus - příklad



vrcholy se přidávají do množiny S v pořadí s, y, z, x

Korektnost

Lema 22 (Invariant cyklu while)

Na začátku iterace **while** *cyklu platí* $v.d = \delta(s, v)$ *pro všechny vrcholy* $v \in S$.

Inicializace na začátku $S = \emptyset$ a tvrzení platí triviálně

Ukončení na konci $Q = \emptyset$, tj. $S = V$ a $v.d = \delta(s, v)$ pro všechny $v \in V$

Iterace potřebujeme prokázat, že když u přesuneme do S, tak $u.d = \delta(u.s)$

- když u není dosažitelný z s tak tvrzení platí protože $u.d = \delta(s, u) = \infty$
- v opačném případě nechť p je nejkratší cesta z s do u; cestu p můžeme dekomponovat na dvě cesty $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$ tak, že bezprostředně před zařazením u do S všechny vrcholy cesty p_1 patří do S a $y \notin S$
- $x \in S \implies x.d = \delta(s, x)$
- při zařazení x do S byla relaxovaná hrana (x, y)) a $s \overset{p_1}{\rightsquigarrow} x \to y$ je nejkratší cesta do $y \implies y.d = \delta(s, y)$
- hrany mají nezápornou délku $\implies \delta(s, y) \leq \delta(s, u)$
- v dané iteraci jsme vybrali vrchol $u \implies u.d \leq y.d$
- spojením dostáváme $u.d \leq y.d = \delta(s, y) \leq \delta(s, u) \implies u.d \leq \delta(s, u)$

Složitost Dijkstrova algoritmu

$\Theta(V)$ operací INSERT (do fronty přidá nový objekt)
$\Theta(V)$ operací EXTRACT_MIN (z fronty odstraní objekt s minimálním klíčem)
$\Theta(E)$ operací DECREASE_KEY (objektu ve frontě sníží hodnotu klíče)

složitost algoritmu závisí od způsobu implementace prioritní fronty Q

pole složitost $\Theta(V \cdot V + E \cdot 1) = \Theta(V^2)$
- INSERT má složitost $\Theta(1)$
- EXTRACT_MIN má složitost $\Theta(V)$
- DECREASE_KEY má složitost $\Theta(1)$

binární halda složitost $\Theta(V \log V + E \log V)$
- INSERT má složitost $\Theta(\log V)$
- EXTRACT_MIN má složitost $\Theta(\log V)$
- DECREASE_KEY má složitost $\Theta(\log V)$

Fibonacciho halda složitost $\Theta(V \log V + E)$
- INSERT má složitost $\Theta(1)$
- EXTRACT_MIN má složitost $\Theta(\log V)$
- DECREASE_KEY má složitost $\Theta(1)$

Optimalizace Dijkstrova algoritmu pro hledání nejkratší cesty mezi dvěma vrcholy s a t

optimalizace 1

- výpočet ukončíme když vrchol t odebereme z prioritní fronty

Optimalizace Dijkstrova algoritmu pro hledání nejkratší cesty mezi dvěma vrcholy s a t

optimalizace 2 - dvousměrné hledání (*bidirectional search*)

- současně spouštíme (dopředný) výpočet Dijkstrova algoritmu z vrcholu s a (zpětný) výpočet z vrcholu t, vždy jednu iteraci každého výpočtu
- dopředný výpočet používá frontu Q_f a přiřazuje vrcholům hodnoty $.d_f$ a $.\pi_f$, zpětný frontu Q_b a přiřazuje hodnoty $.d_b$ a $.\pi_b$
- výpočet ukončíme když nějaký vrchol w je odstraněn z obou front Q_f a Q_b
- po ukončení najdeme vrchol x s minimální hodnotou $x.d_f + x.d_b$ (*pozor, nemusí to být vrchol* w)
- využitím atributů $.\pi_f$ a $.\pi_b$ najdeme nejkratší cestu z s do x a nejkratší cestu z x do t; jejich spojením dostaneme nejkratší cestu z s do t

Optimalizace Dijkstrova algoritmu pro hledání nejkratší cesty mezi dvěma vrcholy s a t

optimalizace 3 - heuristika A*

- pokud bychom dovedli spolehlivě zjistit, že nejkratší cesta z s do t nepovede přes vrchol v, mohli bychom zpracování vrcholu v a hran s ním incidentních přeskočit $\Longrightarrow$ pracovali bychom s menší grafem, a tedy rychleji
- jestliže dva vrcholy jsou stejně daleko od s, chceme při průzkumu preferovat ten, který je blíže k cílovému vrcholu t
- pro odhad preferencí používáme ohodnocení vrcholů – potenciál $h : V \to \mathbb{R}$
- Dijkstrův algoritmus s heuristikou se od klasického liší v tom, že při výběru vrcholu z prioritní fronty vybíráme vrchol s nejnižší hodnotou $v.d + h(v)$
- *jak volit potenciál a proč může urychlit výpočet?*

Heuristika A* - přípustný potenciál

Potenciál je přípustný právě když pro každou hranu $(u, v) \in E$ splňuje podmínku

$$h(u) \leq w(u, v) + h(v)$$

a pro vrchol t platí $h(t) = 0$.

- pro libovolnou cestu $p = \langle v_0 = u, v_1, \ldots, v_k = t \rangle$ z u do t a přípustný potenciál platí

$$\begin{aligned} w(p) &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \\ &\geq h(v_0) - h(v_1) + h(v_1) - h(v_2) + \ldots + h(v_{k-1}) - h(v_k) \\ &= h(u) - h(t) = h(u) \end{aligned}$$

t.j. potenciál vrcholu u je dolním odhadem délky cesty z u do t

Heuristika A* - úprava ohodnocení grafu pomocí potenciálu

- vytvoríme nové ohodnocení grafu $w' : E \to \mathbb{R}$, které definujeme jako

$$w'(u, v) = w(u, v) - h(u) + h(v)$$

- když h je přípustný potenciál, tak pro nové ohodnocení grafu a pro každou hranu grafu platí

$$w'(u, v) \geq 0$$

  t.j. pro výpočet nejkratších cest můžeme použít Dijkstrův algoritmus

- nové ohodnocení w' nemění nejkratší cesty, protože pro každou cestu p z s do t platí

$$w'(p) = w(p) + h(t) - h(s)$$

  a potenciálový rozdíl medzi s a t je pro všechny cesty z s do t stejný

Heuristika A* - úprava ohodnocení grafu pomocí potenciálu

- aby hodnoty h měly příznivý vliv na rychlost výpočtu, měli by hodnoty $h(v)$ být dolním odhadem vzdálenosti z vrcholu v do cílového vrcholu t; čím je odhad přesnější, tím je výpočet rychlejší
- Dijkstrův algoritmus s heuristikou se od klasického liší v tom, že při výběru vrcholu z prioritní fronty vybíráme vrchol s nejnižší hodnotou $v.d + h(v)$
- za předpokladu, že ohodnocení vrcholů h splňuje pro všechny hrany (x, y) grafu podmínku $w(x, y) \geq h(x) - h(y)$, Dijkstrův algoritmus s heuristikou je korektní
  důkaz probíhá analogicky jako pro Dijkstrův algoritmus

Dijkstrův algoritmus a obecné grafy

graf se záporně ohodnocenými hranami ale bez cyklů záporné délky

- algoritmus najde korektní řešení
  (po každé změně hodnoty $u.d$ vrátíme vrchol u do fronty)
- složitost výpočtu může být až exponenciální vůči velikosti grafu
- v praxi někdy rychlejší než algoritmus Bellmana a Forda

graf s cykly záporné délky

- výpočet algoritmu není konečný

Úloha lineárního programování

pro danou $m \times n$ matici A a vektory $b = (b_1, \ldots, b_m)$ a $c = (c_1, \ldots, c_n)$ najít vektor $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, který optimalizuje hodnotu účelové funkce $\sum_{i=1}^{n} c_i x_i$ při splnění ohraničení $Ax \leq b$

příklad

minimalizovat $-2x_1 - 3x_2 - x_3$

při splnění ohraničení

$$\begin{aligned} -x_1 - x_2 - x_3 &\leq 3 \\ 3x_1 + 4x_2 - 2x_3 &\leq 10 \\ 2x_1 - 4x_2 &\leq 2 \\ 4x_1 - x_2 + x_3 &\leq 0 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

přípustné řešení (0, 0, 0)

optimální řešení (0, 5, 5)

Úloha lineárního programování

význam

- mnoho problémů dokážeme vyjádřit jako úlohu lineárního programování (*např. problém nejkratší cesty*)
- pro řešení těchto problémů potom stačí použít algoritmus pro řešení úlohy lineárního programování [9]

algoritmická složitost

- existují polynomiální algoritmy pro řešení úlohy lineárního programování
- pro řešení speciálních případů úlohy lineárního programování existují mnohem rychlejší algoritmy
- problém lineárních ohraničení je příkladem takovéto speciální úlohy
- ukážeme postup založený na SSSP

[9] viz kurz IA101 Algoritmika pro těžké problémy

Problém lineárních nerovnic

- je daná množina nerovnic tvaru $x_i - x_j \leq b_k$, kde

  x jsou proměnné, $1 \leq, i, j \leq n$

  b jsou konstanty, $1 \leq k \leq m$
- úkolem je najít takové hodnoty proměnných x, které splňují všechny nerovnice (tzv. přípustné řešení; v případě, že neexistuje žádné přípustné řešení, tak výstupem je FALSE

příklad

$$\begin{aligned} x_1 - x_2 &\leq 5 \\ x_1 - x_3 &\leq 6 \\ x_2 - x_4 &\leq -1 \\ x_3 - x_4 &\leq -2 \\ x_4 - x_1 &\leq -3 \end{aligned}$$

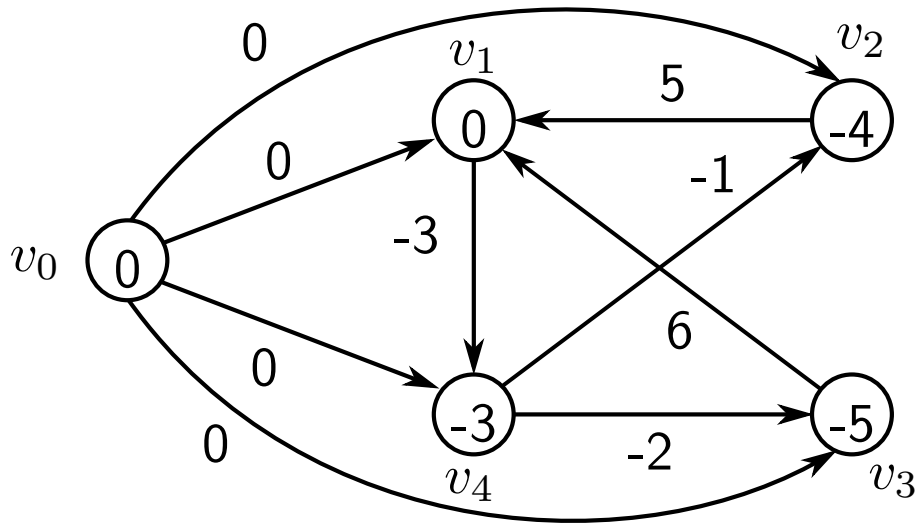
přípustné řešení $x = (0, -4, -5, -3)$

Graf lineárních nerovnic

ohodnocený, orientovaný graf $G = (V, E)$

- $V = \{v_0, v_1, \ldots, v_n\}$
  (*jeden vrchol pro každou proměnnou plus vrchol* v_0)
- $E = \{(v_i, v_j) \mid x_j - x_i \leq b_k$ je nerovnica $\} \cup$
  $\cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$
- $w(v_0, v_j) = 0$ pro všechny j
- $w(v_i, v_j) = b_k$ když $x_j - x_i \leq b_k$

$$
\begin{aligned}
x_1 - x_2 &\leq 5 \\
x_1 - x_3 &\leq 6 \\
x_2 - x_4 &\leq -1 \\
x_3 - x_4 &\leq -2 \\
x_4 - x_1 &\leq -3
\end{aligned}
$$



Nejkratší cesty v grafu lineárních nerovnic

Věta 23

Pro daný systém lineárních nerovnic a k němu odpovídající graf lineárních nerovnic $G = (V, E)$ platí:

1. *když G nemá cyklus záporné délky, tak*

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$$

   je přípustným řešením systému nerovnic;

2. *když G má cyklus záporné délky, tak systém nemá žádné přípustné řešení.*

1. z neexistence cyklu záporné délky plyne

$$\delta(v_0, v_j) \quad \leq \quad \delta(v_0, v_i) + w(v_i, v_j)$$

   po dosazení

$$\begin{aligned} x_j &\leq x_i + b_k \\ x_j - x_i &\leq b_k \end{aligned}$$

Nejkratší cesty v grafu lineárních nerovnic

Věta 24

Pro daný systém lineárních nerovnic a k němu odpovídající graf lineárních nerovnic $G = (V, E)$ platí:

1.
2. *když G má cyklus záporné délky, tak systém nemá žádné přípustné řešení.*

1.
2. nechť $c = \langle v_1, v_2, \ldots, v_k \rangle$, kde $v_1 = v_k$, je cyklus záporné délky
   hrany cyklu c odpovídají nerovnostem

$$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2) \\ x_3 - x_2 &\leq w(v_2, v_3) \\ &\vdots \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k) \end{aligned}$$

   přípustné řešení x musí splňovat všechny tyto nerovnosti
   po sečtení všech nerovností dostaneme $0 \leq w(c)$, což je spor s předpokladem
   o záporné délce cyklu c

Algoritmus pro problém lineárních nerovnic

1. vytvoř graf lineárních nerovnic
   - $n + 1$ vrcholů
   - $m + n$ hran
   - časová složitost $\Theta(m + n)$

2. najdi nejkratší cesty z vrcholu v_0 algoritmem Bellmana a Forda
   - časová složitost $\mathcal{O}((n + 1)(m + n)) = \mathcal{O}(n^2 + nm)$

3. když algoritmus vrátí hodnotu FALSE, tak problém lineárních nerovnic nemá přípustné řešení když algoritmus vrátí hodnotu TRUE, tak přípustným řešením je $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$